Bioinformatica con R

Traduzione, libero adattamento e integrazione dalla documentazione online di R (https://www.r-project.org/help.html) e dal testo: "*A little book of R for bioinformatics*" di Avril Coghlan, con licenza Creative Commons Attribution 3.0 (https://a-little-book-of-r-for-bioinformatics.readthedocs.io/en/latest/index.html).

Crescenzio Gallo
crescenzio.gallo@unifg.it

Università degli Studi di Foggia

Dipartimenti di Area Medica

21 marzo 2020 17:50

Indice

4. Allineamento a coppie di sequenze

1. La banca dati *UniProt*
2. La pagina Web *UniProt* per le sequenze proteiche
3. Recuperare sequenze proteiche dal sito Web *UniProt*
4. Recuperare sequenze proteiche in R:
    1. con il pacchetto *rentrez*
    2. con il pacchetto Bioconductor *UniProt.ws*
    3. con il pacchetto *SeqinR*
5. Confronto di due sequenze mediante un grafico a dispersione (*dotplot*)
6. Allineamento globale a coppie di sequenze di DNA mediante l'algoritmo Needleman-Wunsch
7. Allineamento globale a coppie di sequenze proteiche mediante l'algoritmo Needleman-Wunsch
8. Allineamento di sequenze *UniProt*
9. Visualizzazione di un allineamento lungo a coppie
10. Allineamento locale a coppie di sequenze proteiche mediante l'algoritmo di Smith-Waterman
11. Calcolo della significatività statistica di un allineamento globale a coppie

5. Allineamento multipli (MSA) e alberi filogenetici

1. Recuperare una lista di sequenze proteiche
2. Allineamento multiplo di sequenze dalla banca dati PDB con il pacchetto R *bio3d*
3. Installazione del software di allineamento multiplo CLUSTAL
4. Creazione di un allineamento multiplo di sequenze di proteine, DNA o mRNA mediante CLUSTAL
5. Lettura di un file di allineamento multiplo in R
6. Visualizzazione di un allineamento multiplo lungo
7. Eliminare da un allineamento multiplo le regioni scarsamente conservate
8. Calcolo delle distanze genetiche tra sequenze proteiche
9. Calcolo delle distanze genetiche tra sequenze di DNA/mRNA
10. Costruire un albero filogenetico *unrooted* per sequenze proteiche
11. Costruire un albero filogenetico *rooted* per sequenze proteiche
12. Costruire un albero filogenetico per sequenze di DNA o mRNA
13. Esempi di uso del pacchetto *ape* per la creazione di alberi filogenetici

6. Ricerca genica (*gene-finding*) computazionale

1. Il codice genetico
2. Ricerca dei codoni di inizio e fine in una sequenza di DNA
3. *Reading frame* (finestra di lettura)
4. Ricerca di *reading frame* aperti (ORF) sul *forward strand* di una sequenza di DNA
5. Predire la sequenza proteica per un ORF
6. Ricerca di *reading frame* aperti (ORF) sul *reverse strand* di una sequenza di DNA

7. Visualizzazione delle strutture proteiche

11. Analisi di dati da microarray

1. Lettura dei file CEL
2. Costruzione di un oggetto di tipo *ExpressionSet*
3. Gestione di un oggetto di tipo *AffyBatch*
4. Controllo della qualità dei dati da microarray
5. Generazione di dati di espressione artificiali
6. Normalizzazione dei dati
7. Come superare i *batch effects* nei dati di espressione
8. Analisi esplorativa dei dati con la PCA
9. Ricerca dei geni differenzialmente espressi
10. Lavorare con condizioni sperimentali multiple
11. Gestione dei dati di serie temporali
12. Le variazioni di espressione (*fold change*) nei dati da microarray
13. L'arricchimento funzionale dei dati con i termini GO
14. Clustering di dati da microarray
15. Network di co-espressione di geni
16. Visualizzazione dei dati di espressione genica

12. Analisi di dati GWAS (Genome Wide Association Study)

1. L'analisi delle associazioni SNP
2. Esecuzione di scansioni delle associazioni SNP
3. Analisi delle associazioni SNP dell'intero GENOMA
4. Importazione di dati GWAS in formato PLINK
5. Gestione dei dati con il pacchetto GWASTools
6. Manipolazione di altri formati di dati GWAS
7. Test dei dati per l'equilibrio di Hardy-Weinberg
8. Test di associazione con dati CNV
9. Visualizzazioni negli studi GWAS

13. Analisi dei dati di Spettrometria di Massa (MS)

1. Lettura dei dati MS nel formato mzXML/mzML
2. Lettura dei dati MS nel formato Bruker
3. Conversione dei dati MS dal formato mzXML in MALDIquant
4. Estrazione di elementi dagli oggetti dei dati MS
5. Pre-elaborazione dei dati MS
6. Rilevamento dei picchi nei dati MS
7. Allineamento dei picchi con i dati MS
8. Identificazione dei peptidi nei dati MS
9. Analisi di quantificazione delle proteine

10. Analisi di più gruppi nei dati MS
11. Visualizzazioni utili per l'analisi dei dati MS

14. Analisi dei dati NGS (Next Generation Sequencing)

1. Interrogazione del database SRA
2. Download dei dati dal database SRA
3. Lettura di file FASTQ in R
4. Lettura dei dati di allineamento
5. Pre-elaborazione dei dati NGS
6. Analisi dei dati RNAseq con il pacchetto *edgeR*
7. Analisi differenziale dei dati NGS con il pacchetto *limma*
8. Arricchimento dei dati RNAseq con i termini GO
9. L'arricchimento KEGG dei dati di sequenza
10. Analisi dei dati di metilazione
11. Analisi dei dati ChipSeq
12. Visualizzazione dei dati NGS

15. Il Machine Learning nella Bioinformatica

1. Clustering dei dati mediante *k*-means e clustering gerarchico
2. Visualizzazione dei cluster
3. Apprendimento supervisionato per la classificazione
4. Apprendimento probabilistico con Naïve Bayes
5. Il bootstrap nel machine learning
6. La cross-validation per i classificatori
7. Misurare le performance dei classificatori
8. Visualizzazione di una curva ROC in R
9. Identificazione di biomarcatori con dati da microarray

Appendice

Premessa

Le malattie tropicali "trascurate" sono malattie gravi che colpiscono molte persone nei paesi tropicali e che sono state relativamente poco studiate. L'Organizzazione Mondiale della Sanità elenca le seguenti malattie tropicali trascurate: *tracoma*, *lebbra*, *schistosomiasi*, *elminti trasmessi dal suolo*, *filariosi linfatica*, *oncocercosi*, *ulcera di Buruli*, *framboesia*, *morbo di Chagas*, *tripanosomiasi africana*, *leishmaniosi*, *febbre di Dengue*, *rabbia*, *Dracunculiasi* (malattia dei vermi della Guinea) e *Fascioliasis* (vedi https://www.who.int/neglected_diseases/diseases/en/).

I genomi di molti degli organismi che causano malattie tropicali trascurate sono stati completamente sequenziati, o sono attualmente in fase di sequenziamento, tra cui:

- il batterio *Chlamydia trachomatis*, che causa il tracoma;
- il batterio *Mycobacterium leprae*, che causa la lebbra;
- il batterio *Mycobacterium ulcerans*, che causa l'ulcera di Buruli;
- il batterio *Treponema pallidum subsp. pertenue*, che provoca la framboesia;
- il *Trypanosoma cruzi protista*, che causa il morbo di Chagas;
- il *Trypanosoma brucei protista*, che provoca la tripanosomiasi africana;
- la *Leishmania major* e le specie di *Leishmania* affini, che causano la leishmaniosi;
- il verme *Schistosoma mansoni*, che provoca la schistosomiasi;
- i vermi nematodi *Brugia malayi* e *Wuchereria bancrofti*, che provocano la filariosi linfatica;
- il verme nematode *Loa loa*, che provoca la filariosi sottocutanea;
- il verme nematode *Onchocerca volvulus*, che provoca l'oncocercosi;
- il verme nematode *Necator americanus*, che provoca l'elmintiasi trasmessa dal suolo;
- il virus *Dengue*, che causa la febbre Dengue;
- il virus *Rabies*, che causa la rabbia.

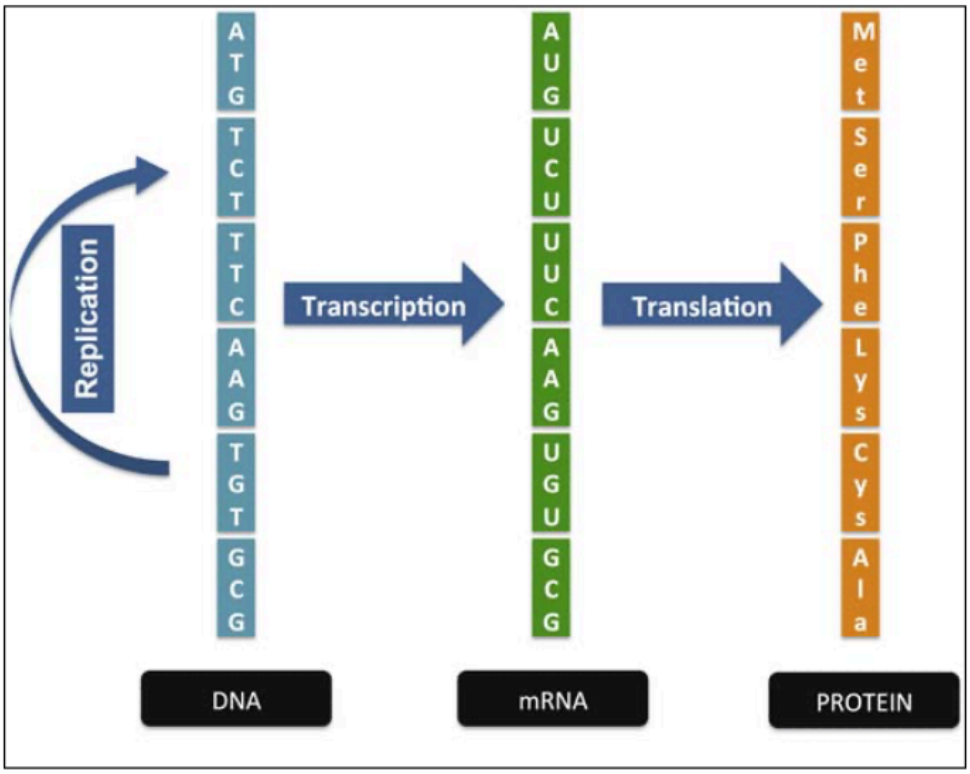
Per incoraggiare la ricerca su questi organismi, molti degli esempi in questo testo si basano sull'analisi di questi genomi.

Introduzione

Gli acidi nucleici e le sequenze di proteine sono onnipresenti oggi in biologia. Gli acidi nucleici rappresentano geni, RNA e così via, mentre le proteine sono considerate i mattoni della vita. Queste biomolecole rappresentano il contenuto informativo di un sistema vivente in termini di sequenze di caratteri. Il famoso dogma centrale della biologia molecolare può essere percepito come la conversione di un tipo di un insieme di caratteri in un altro a livello di sequenza. Ad esempio, gli RNA messaggeri (mRNA) vengono convertiti in proteine o, meglio, polipeptidi tramite traduzione.

Quindi, è la sequenza del DNA che alla fine determinerà la sequenza delle proteine. Ciò rende l'analisi delle sequenze (nucleotidiche e proteiche) importante per varie applicazioni che vanno dal confronto delle biomolecole per lo studio dell'evoluzione, alla mutazione e all'identificazione di siti interessanti nelle biomolecole, e così via.

È stata l'enorme crescita dei dati delle sequenze che ha aperto la strada all'evoluzione della *bioinformatica*. I dati sono di solito un flusso di caratteri; nel caso del DNA sono le lettere ATGC (che rappresentano le basi nei nucleotidi — ricordiamo che l'RNA ha U invece di T), mentre — nel caso delle proteine — sono costituiti dalle lettere che rappresentano gli aminoacidi. Analizzare le proprietà di queste molecole e capire il meccanismo di base della trasformazione di un tipo di informazione in un altro o nel livello successivo è una delle chiavi di volta per decifrare i sistemi viventi. La figura seguente illustra come l'informazione genetica viene trasferita ad una proteina, processo che avviene mediante il trasferimento di un tipo di sequenza (nucleotidi) ad un altro (aminoacidi) attraverso il processo di *trascrizione* (transcription) e *traduzione* (translation):



L'analisi delle sequenze è il compito più elementare della bioinformatica. In generale, si riferisce all'elaborazione dei dati di sequenza del DNA o delle proteine per sfruttare le informazioni di base sulla funzione, la struttura o l'evoluzione della biomolecola.

Le proteine sono biomolecole molto versatili, i cui ruoli strutturali e funzionali sono stati studiati attraverso studi sperimentali e numerose tecniche di calcolo. L'analisi delle sequenze proteiche riguarda principalmente il matching e gli allineamenti; tuttavia le proteine, a differenza dei nucleotidi, mostrano una grande varietà di conformazioni tridimensionali necessarie per i loro diversi ruoli strutturali e funzionali. Gli aminoacidi in una proteina dettano importanti termini nella sua struttura tridimensionale. La frequenza di particolari residui di aminoacidi nella sequenza gioca un ruolo importante nel determinare le strutture secondarie della proteina (elica α, foglietti β e ripiegamento; per i dettagli di questa dipendenza, visitare il sito http://www.ncbi.nlm.nih.gov/books/NBK22342/table/A354/?report=objectonly).

Oltre che dalla struttura, le proprietà fisico-chimiche di una proteina sono anche determinate dalle proprietà analoghe degli aminoacidi (cioè la sequenza) in essa contenuti. Ad esempio, l'acidità, la basicità, e così via, possono essere decise in base alla frazione di aminoacidi acidi o basici presenti nella proteina. Tutte le proprietà delle proteine, dalla struttura alle proprietà fisico-chimiche, sono fondamentali per il loro ruolo biologico.

L'analisi delle sequenze ci permette di trovare somiglianze o dissimiglianze tra di loro a fini di confronto. Si possono utilizzare i dati di una sequenza per identificare le sue proprietà chimiche in base al suo contenuto, o anche per calcolarne la struttura. Inoltre, si possono utilizzare per creare un modello basato sull'omologia per predire strutture tridimensionali sconosciute di proteine che possono essere utilizzate nella scoperta di farmaci. Gli scopi non si limitano comunque solamente agli aspetti menzionati.

In questo testo introdurremo i diversi tipi di analisi delle sequenze che possono essere fatte con R e le corrispondenti librerie e pacchetti, esaminando anche gli aspetti di genomica computazionale e comparata, nonché l'analisi evolutiva.

1. Analisi statistica delle sequenze biologiche — Parte I

1.1 Uso di R per la Bioinformatica

Questo testo spiega come utilizzare il software R per effettuare alcune semplici analisi comuni nella bioinformatica. In particolare, l'analisi computazionale dei dati delle sequenze biologiche, come le sequenze del genoma e le sequenze proteiche.

Si presuppone che il lettore abbia alcune conoscenze di base di biologia, ma non necessariamente di bioinformatica. L'obiettivo è quello di spiegare i concetti base dell'analisi bioinformatica, e come effettuare queste analisi utilizzando R.

1.2 I pacchetti R per la bioinformatica: "Bioconductor" e *SeqinR*

Esistono molti pacchetti R per eseguire un'ampia varietà di analisi. Questi non vengono forniti con l'installazione standard di R, ma devono essere installati e caricati come "add-on", normalmente con il comando: `install.packages(…)`.

Vi sono diversi pacchetti R per la bioinformatica. In particolare:

- **Bioconductor** (https://www.bioconductor.org), un sito che contiene diversi pacchetti con molte funzioni R per l'analisi di set di dati biologici come i dati da microarray;
- **SeqinR** (ftp://http://pbil.univ-lyon1.fr/pub/seqinr/), un pacchetto di funzioni R per recuperare ed analizzare sequenze di DNA e proteine da database specializzati.

Per utilizzare le funzioni del pacchetto *SeqinR*, dobbiamo prima installarlo con il comando: `install.packages("seqinr")`. Una volta installato, è possibile caricarlo per l'utilizzo digitando: `library(seqinr)`.

1.3 Il formato FASTA

Il formato FASTA è un formato semplice e ampiamente utilizzato per la memorizzazione di sequenze biologiche (DNA o proteine). È stato utilizzato per la prima volta dal programma FASTA per l'allineamento delle sequenze. Inizia con una riga di descrizione che inizia con il carattere ">", seguita da righe di sequenze. Ecco un esempio di file FASTA:

```
> A06852 183 residues
MPRLFSYLLGVWLLLSQLPREIPGQSTNDFIKACGRELVRLWVEICGSVSWGRTALSLEE
PQLETGPPAETMPSSITKDAEILKMMLEFVPNLPQELKATLSERQPSLRELQQSASKDSN
LNFEEFKKIILNRQNEAEDKSLLELKNLGLDKHSRKKRLFRMTLSEKCCQVGCIRKDIAR
LC
```

1.4 Il database di sequenze NCBI

Il National Centre for Biotechnology Information (NCBI) (https://www.ncbi.nlm.nih.gov) negli Stati Uniti gestisce un enorme database di tutti i dati raccolti sulle sequenze di DNA e proteine, il NCBI Sequence

Database. Vi è anche un database simile in Europa, il database delle sequenze del Laboratorio Europeo di Biologia Molecolare (EMBL-EBI) (https://www.ebi.ac.uk), e anche un database simile in Giappone, la DNA Data Bank of Japan (DDBJ; https://www.ddbj.nig.ac.jp). Queste tre banche dati si scambiano dati ogni notte, quindi in ogni momento contengono dati quasi identici.

Ogni sequenza nel NCBI Sequence Database è memorizzata in un *record* separato, cui è assegnato un identificatore univoco che può essere utilizzato per fare riferimento a quella sequenza. L'identificatore è noto come "*accession*" e consiste in una stringa di numeri e lettere. Ad esempio, il virus Dengue, che causa la febbre omonima, è presente in quattro tipi: DEN-1, DEN-2, DEN-3 e DEN-4. Gli *accession* NCBI per le sequenze di DNA dei virus DEN-1, DEN-2, DEN-3 e DEN-4 sono rispettivamente NC_001477, NC_001474, NC_001475 e NC_002640.

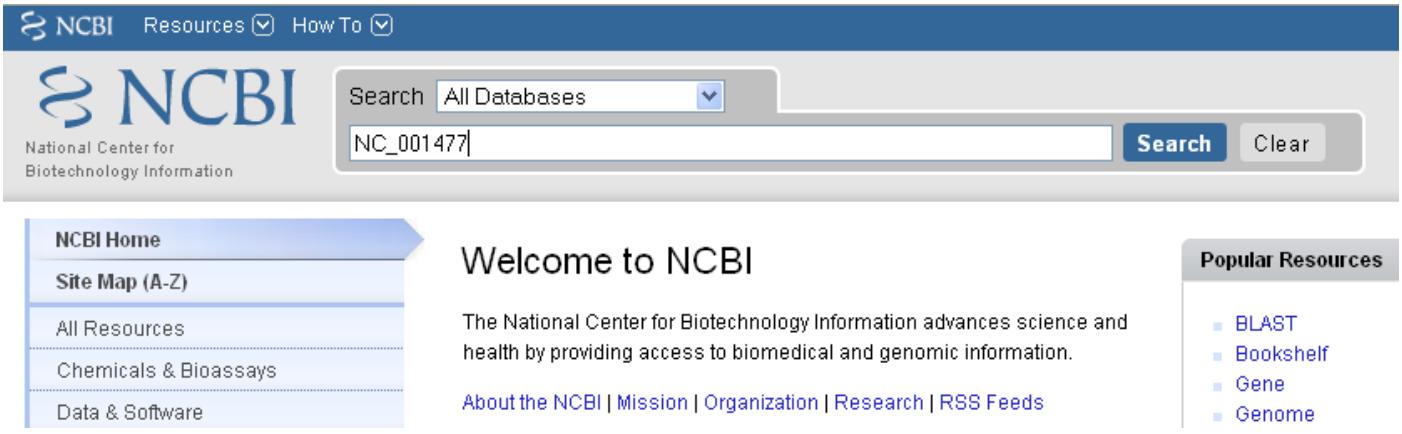
Poiché le banche dati NCBI, EMBL e DDBJ si sincronizzano ogni notte, le sequenze del virus Dengue saranno presenti in tutte e tre le banche dati, ma avranno diversi *accession*, in quanto ciascuna banca dati utilizza i propri sistemi di numerazione per fare riferimento ai propri record di sequenze.

1.5 Recupero di sequenze genomiche

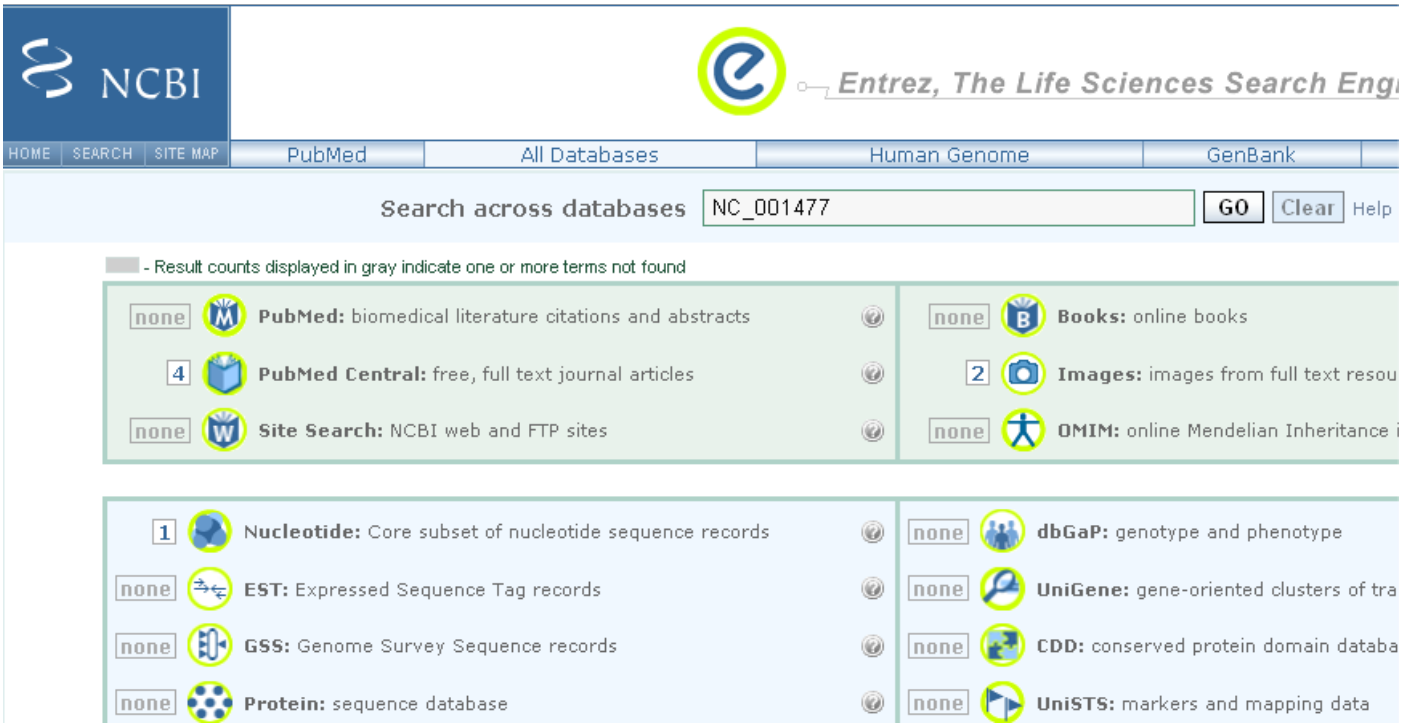
1.5.1 Recupero di sequenze genomiche dal sito web NCBI

È possibile recuperare facilmente i dati relativi al DNA o alle sequenze di proteine dal database di sequenze NCBI tramite il suo sito web https://www.ncbi.nlm.nih.gov.

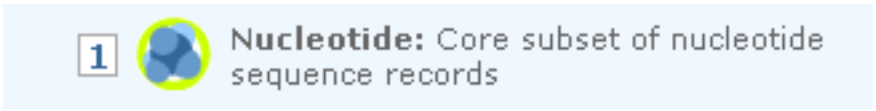
La sequenza di DNA Dengue DEN-1 è una sequenza di DNA virale con *accession* NCBI NC_001477. Per recuperare la sequenza di DNA dal sito web del NCBI, digitare "NC_001477" nella casella di ricerca della pagina e premere il pulsante <Search>:



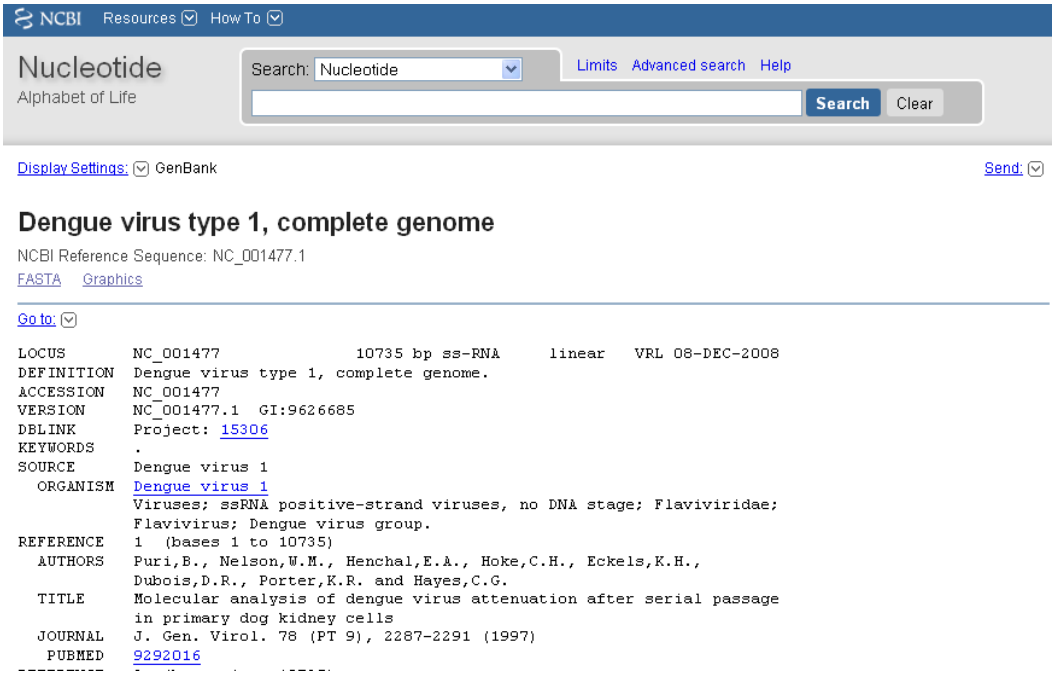
Nella pagina dei risultati vedrete il numero di risultati per "NC_001477" in ciascuno dei database dell'NCBI. Ad esempio, il database "PubMed" contiene estratti da documenti scientifici, il database "Nucleotide" contiene dati di sequenze di DNA e RNA, "Protein" contiene dati di sequenze di proteine, e così via. L'immagine sottostante mostra come dovrebbe essere la pagina dei risultati per la ricerca NC_001477, con un risultato nel database "Nucleotide":



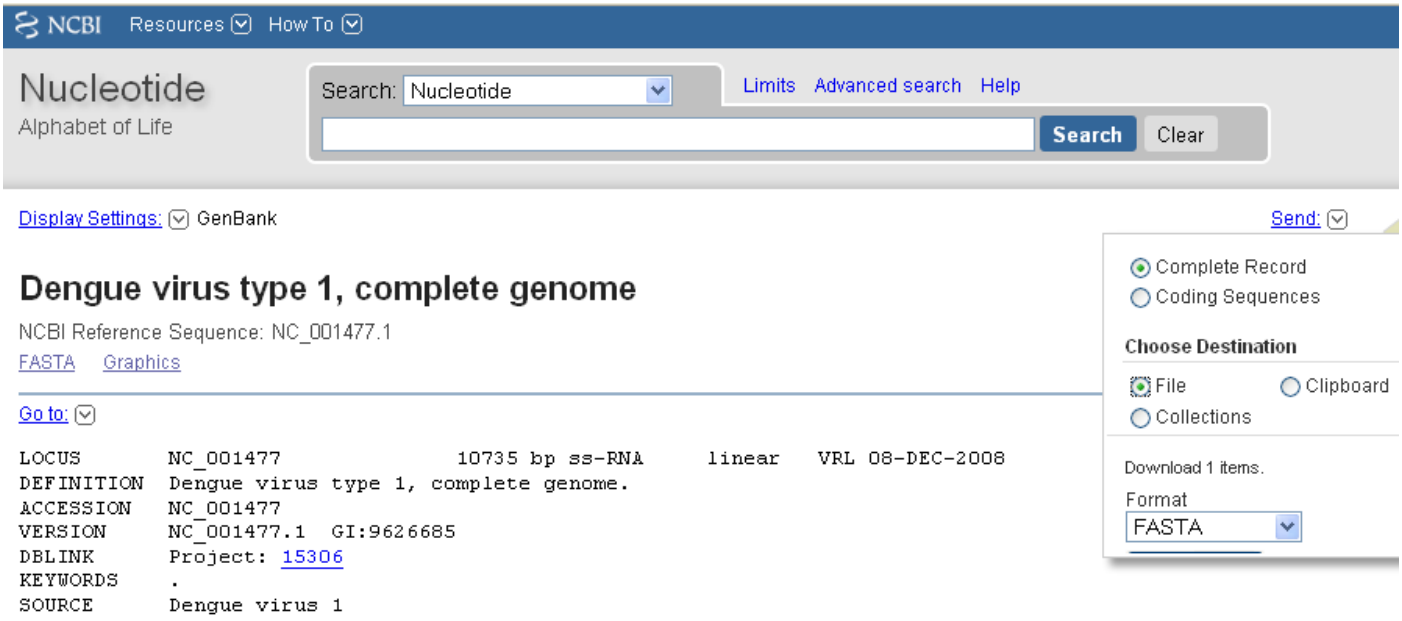
Per esaminare l'unica sequenza trovata nella banca dati dei nucleotidi, è necessario cliccare sull'icona corrispondente nella pagina dei risultati della ricerca:



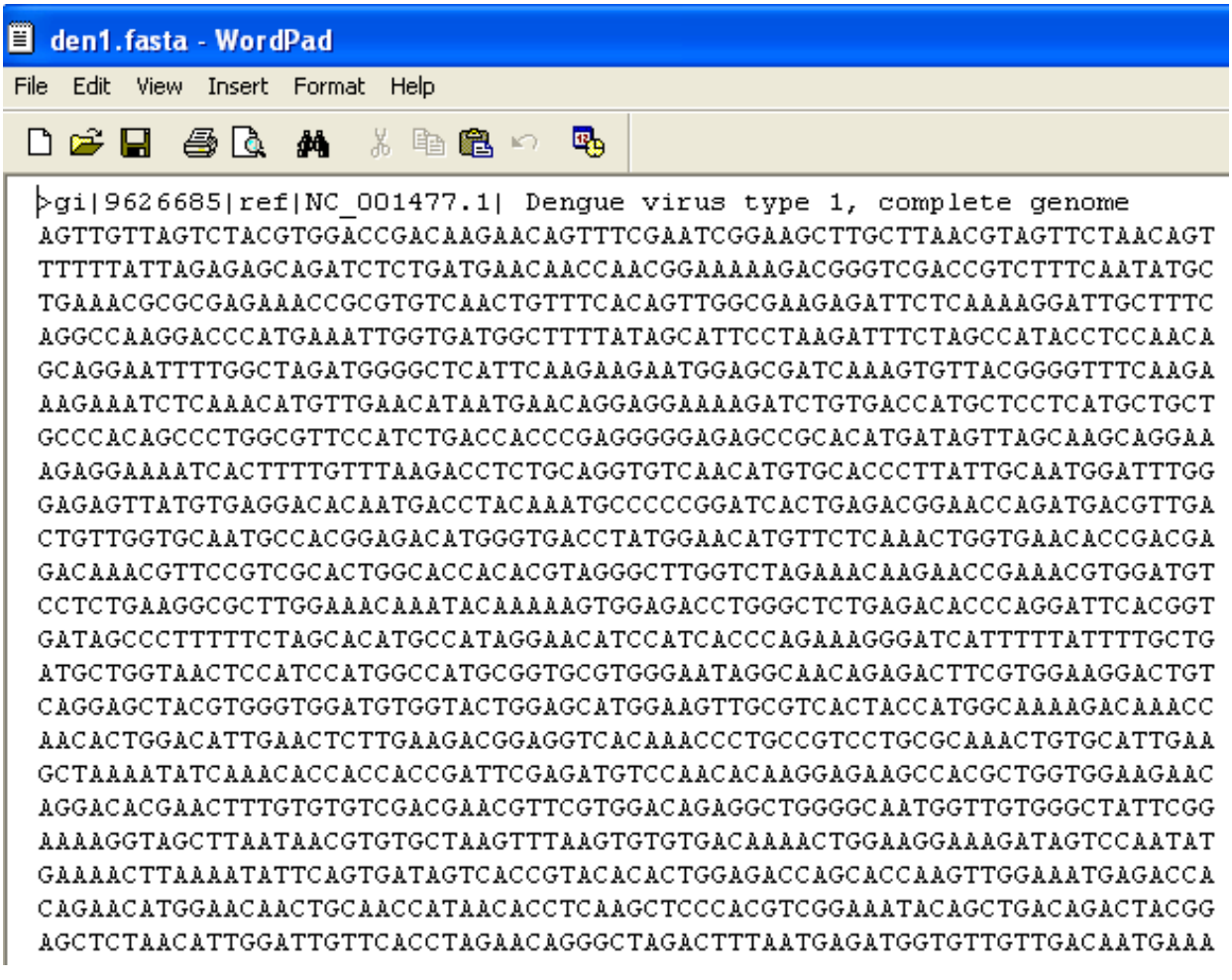
Quando si clicca sull'icona per il database dei nucleotidi NCBI, si arriva al record per NC_001477 nel database dei nucleotidi NCBI. Questo conterrà il nome e l'*accession* NCBI della sequenza, così come altri dettagli come eventuali documenti che descrivono la sequenza:



Per recuperare la sequenza del DNA del virus DEN-1 Dengue come file in formato FASTA, cliccare sul pulsante <Send> in alto a destra della pagina web, quindi scegliere "File" nel menu che appare, scegliere FASTA dal menu "Format" e cliccare infine su <Create file>, Apparirà un riquadro che chiede il nome del file e dove salvarlo:



È possibile aprire il file FASTA contenente la sequenza del genoma del virus DEN-1 utilizzando un qualunque editor di testo, come ad es. blocco note o WordPad. Il contenuto del file in formato FASTA dovrebbe ora essere visualizzato come segue:



1.5.2 Recupero di sequenze genomiche tramite il pacchetto SeqinR

I dati di una sequenza possono essere recuperati sia visitando il sito web NCBI come visto in precedenza, sia accedendo al database all'interno di R tramite i comandi/funzioni del pacchetto *SeqinR* (che a sua volta utilizza il sistema pubblico ACNUC http://doua.prabi.fr/databases/acnuc). Per utilizzarli, carichiamo prima la libreria con il seguente comando:

```
> install.packages("seqinr")
> library(seqinr)
```

Quindi, per selezionare la banca dati necessaria per ottenere la sequenza, possiamo controllare quelle disponibili con il comando `choosebank()`:

```
> choosebank()
 [1] "genbankseqinr"   "genbank"         "embl"            "emblwgs"         "swissprot"       "ensembl"
 [7] "hogenom7dna"     "hogenom7"        "hogenom"         "hogenomdna"      "hovergendna"     "hovergen"
[13] "hogenom5"        "hogenom5dna"     "hogenom4"        "hogenom4dna"     "homolens"        "homolensdna"
[19] "hobacnucl"       "hobacprot"       "phever2"         "phever2dna"      "refseq"          "refseq16s"
[25] "greviews"        "bacterial"       "archaeal"        "protozoan"       "ensprotists"     "ensfungi"
[31] "ensmetazoa"      "ensplants"       "ensemblbacteria" "mito"            "polymorphix"     "emglib"
[37] "refseqViruses"   "ribodb"          "taxodb"
```

Nel nostro caso scegliamo la banca dati genomica "genbank" (https://www.ncbi.nlm.nih.gov/genbank/) come segue (il parametro `timeout` serve qualora i tempi di risposta del server remoto siano lenti):

```
> choosebank("genbank", timeout=20)
```

Una volta aperta la banca dati remota, la possiamo interrogare con il seguente comando (questa fase richiede tempo per essere completata), che cerca le sequenze registrare per il gene BRCA1 della specie *Homo sapiens* (questo gene codifica una fosfoproteina nucleare che svolge un ruolo nel mantenimento della stabilità genomica, e agisce anche come soppressore di tumori):

```
> BRCA1 <- query("BRCA1", "SP=Homo sapiens AND K=BRCA1")
```

Esaminiamo le componenti dell'oggetto risultato dell'interrogazione:

```
> attributes(BRCA1)
$names
[1] "call"     "name"     "nelem"    "typelist" "req"      "socket"
$class
[1] "qaw"
```

Per controllare l'insieme di tutte le sequenze che sono state recuperate, digitiamo il seguente comando per ottenere il nome, la lunghezza e altri attributi per ogni sequenza disponibile:

```
> BRCA1$req
[[1]]
           name          length           frame           ncbicg
"AB621825.BRCA1"            "71"             "0"             "1"
[[2]]
           name          length           frame           ncbicg
"AF005068.BRCA1"          "5379"             "0"             "1"
. . .
```

Leggiamo nella variabile *myseqs* tutte le sequenze recuperate:

```
> myseqs <- getSequence(BRCA1)
```

Per ottenere una sequenza specifica in base al suo *accession number* utilizziamo l'attributo "AC" nel comando di interrogazione e quindi il comando `getSequence`:

```
> tmp <- query("tmp", "SP=Homo sapiens AND AC=U61268")
> myseq1 <- getSequence(tmp$req[[1]])
```

Diamo un'occhiata alla sequenza che è stata recuperata:

```
> myseq1
   [1] "t" "c" "g" "c" "t" "a" "g" "a" "a" "c" "c" "c" "g" "g" "g" "a" "g" "g" "c" "g" "g" "a" "g" "g" "t"
  [26] "t" "g" "c" "a" "g" "t" "g" "a" "g" "c" "c" "g" "a" "g" "a" "t" "c" "g" "c" "g" "c" "c" "a" "t" "t"
. . .
[1301] "c" "t" "g" "g" "c" "c" "a" "a" "c" "a" "t" "g" "g" "t" "g" "a" "a" "a" "a" "c" "c" "c" "c" "c" "t"
[1326] "c" "t" "c" "c" "a" "c" "t" "a" "a" "a" "a" "a" "t"
```

Il pacchetto *seqinr* funziona in modo simile a *biomaRt*. La banca dati selezionata viene interrogata tramite l'insieme dei termini di ricerca inseriti (K per parola chiave e SP per specie) e tutte le sequenze disponibili per gli attributi di ricerca indicati sono assegnate all'oggetto definito (nel nostro caso, "BRCA1"). Da questa lista possiamo filtrare i risultati rilevanti tramite i suoi attributi, come ad es. il nome o la posizione della sequenza (nel nostro caso abbiamo usato la posizione `[[1]]`). Il comando `getSequence` recupera le sequenze dall'oggetto restituito dalla query.

Oltre all'attributo K (Keyword) per la query, ci sono altri possibili attributi che possono essere utilizzati (allo scopo basta visualizzare l'aiuto del comando "query"). È possibile recuperare ad esempio l'annotazione (gli altri attributi di una sequenza) con il comando `getAnnot`:

```
> annots <- getAnnot(BRCA1$req[[1]])
> annots
 [1] "LOCUS       HSU61268                1338 bp    DNA     linear   PRI 16-JAN-1997"
 [2] "DEFINITION  Human breast and ovarian cancer susceptibility (BRCA1) gene, exon"
 [3] "            2, partial flanking introns, and partial cds."
 [4] "ACCESSION   U61268"
. . .
```

Possiamo infine cercare gli identificatori delle sequenze (e visualizzarne il numero totale) come segue:

```
> mynames <- getName(BRCA1)
> length(mynames)
[1] 582
```

È sempre consigliabile chiudere la banca dati dopo l'interrogazione per evitare di avere più connessioni aperte. A tal fine è necessario utilizzare la seguente funzione di chiusura della banca dati (senza argomenti):

```
> closebank()
```

1.5.3 Recupero di sequenze genomiche tramite il pacchetto dati BSgenome

Oltre alle possibilità mostrate, può essere utile l'impiego di *BSgenome*, un'infrastruttura software Bioconductor condivisa da tutti i pacchetti di dati sul genoma basati su *Biostrings*. In particolare, il pacchetto *BSgenome.Hsapiens.NCBI.GRCh38* consente di leggere sequenze genomiche complete per l'Homo sapiens fornite da NCBI (versione GRCh38 del 17/12/2013) e memorizzate in oggetti *Biostrings*.

Innanzitutto, installiamo e carichiamo il pacchetto e osserviamo i dati disponibili:

```
> BiocManager::install("BSgenome.Hsapiens.NCBI.GRCh38")
> library(BSgenome.Hsapiens.NCBI.GRCh38)
> Hsapiens
Human genome:
# organism: Homo sapiens (Human)
# provider: NCBI
# provider version: GRCh38
# release date: 2013-12-17
# release name: Genome Reference Consortium Human Build 38
# 455 sequences:
#   1                                   2
#   3                                   4
#   5                                   6
#   7                                   8
#   9                                   10
#   ...                                 ...
#   HSCHR19KIR_FH05_B_HAP_CTG3_1        HSCHR19KIR_FH06_A_HAP_CTG3_1
#   HSCHR19KIR_FH06_BA1_HAP_CTG3_1      HSCHR19KIR_FH08_A_HAP_CTG3_1
#   HSCHR19KIR_FH08_BAX_HAP_CTG3_1      HSCHR19KIR_FH13_A_HAP_CTG3_1
#   HSCHR19KIR_FH13_BA2_HAP_CTG3_1      HSCHR19KIR_FH15_A_HAP_CTG3_1
#   HSCHR19KIR_RP5_B_HAP_CTG3_1
# (use 'seqnames()' to see all the sequence names, use the '$' or '[[' operator to
access a given sequence)
```

Estraiamo ad esempio la sequenza dei nucleotidi dalla posizione 35.623.354 alla posizione 35.637.951 (lunga 14.598 basi) del forward strand del cromosoma 22 (un potenziale promotore del gene MB):

```
> seq <- getSeq(Hsapiens, "22", start=35623354, end=35637951, strand="+")
> seq
  14598-letter "DNAString" instance
seq: CTTTGATTGTTTATTTTTGGATTGAGTCTGCCCAGGGTTTTGGCTTGCT…
GTTGCTCCTCTACCCAACCTCATCAATGCTCTC
```

Se necessario, possiamo invertire la sequenza mediante la funzione `rev()`:

```
> seq = rev(seq)
> seq
  14598-letter "DNAString" instance
seq:
CTCTCGTAACTACTCCAGTCCTTCGGAGGACAACCCATCTCCTCGTTGA...CGTTCGGTTTTGGGACCCTTATTTGTTAGTTTC
```

Si osservi che la sequenza estratta dalla funzione `getSeq()` viene restituita in un oggetto di tipo DNAString; se si vuole un vettore di caratteri occorre specificare il parametro `as.character=TRUE`.

1.6 Registrare i dati di una sequenza in un file FASTA

Dopo aver recuperato delle sequenze (e i relativi nomi), è possibile salvarle in un file in formato FASTA in R utilizzando la funzione `write.fasta()` del pacchetto *SeqinR*. La funzione `write.fasta()` richiede il nome del file di output mediante l'argomento `file.out`. È inoltre necessario specificare la variabile R che contiene le sequenze (argomento `sequences`) e i nomi delle sequenze (argomento `names`).

Ad esempio, possiamo registrare le sequenze del gene BRCA1 recuperate nel paragrafo precedente (memorizzate nelle variabili *myseqs* e *mynames*) nel file in formato FASTA "brca1.fasta" digitando:

```
> write.fasta(sequences = myseqs, names = mynames, file.out = "brca1.fasta")
```

1.7 Leggere una sequenza da un file FASTA in R

Utilizzando il pacchetto *SeqinR* è possibile leggere facilmente una sequenza di DNA da un file FASTA in R. Per esempio, supponendo di voler leggere la sequenza del genoma del virus DEN-1 dal file in formato FASTA "den1.fasta", basta digitare il comando:

```
> library("seqinr")
> dengue <- read.fasta(file = "den1.fasta")
```

Si noti che R legge e scrive i file nella cartella corrente, che si può impostare tramite il comando "Cambia directory di lavoro" del menu "Misc".

Il comando sopra riportato legge il contenuto del file "den1.fasta" in un oggetto R di tipo lista chiamato *dengue*, che contiene le informazioni del file FASTA (il nome dato alla sequenza nel file e la sequenza di DNA stessa). Il primo elemento della lista *dengue* contiene la sequenza DNA; quindi, possiamo memorizzare la sequenza di DNA per il virus DEN-1 in una variabile *dengueseq* digitando:

```
> dengueseq <- dengue[[1]]
```

che copia la sequenza di nucleotidi nel vettore *dengueseq*.

1.8 Lunghezza di una sequenza DNA

Una volta recuperata una sequenza DNA, possiamo ottenere alcune semplici statistiche, come la lunghezza totale della sequenza in nucleotidi. Nell'esempio precedente, abbiamo recuperato la sequenza del genoma del virus DEN-1 e l'abbiamo memorizzata nella variabile vettore *dengueseq*. Per ottenere successivamente la lunghezza della sequenza del genoma, utilizziamo la funzione `length()`:

```
> length(dengueseq)
[1] 10735
```

La funzione `length()` restituisce il numero di elementi del vettore fornito in input, e quindi la lunghezza della sequenza memorizzata nella variabile *dengueseq*: 10735 nucleotidi.

1.9 Composizione di base di una sequenza DNA

Una prima analisi ovvia di qualsiasi sequenza di DNA è quella di contare il numero di occorrenze dei quattro diversi nucleotidi ("A", "C", "G" e "T") nella sequenza. Questo può essere fatto usando la funzione `table()`. Per esempio, per trovare il numero di A, C, G e T nella sequenza del virus DEN-1 (memorizzata in precedenza nella variabile *dengueseq*), occorre digitare:

```
> table(dengueseq)
dengueseq
   a    c    g    t
3426 2240 2770 2299
```

da cui si deduce che la sequenza ha 3426 nucleotidi A, 2240 C, 2770 G e 2299 T.

1.10 Contenuto GC di una sequenza DNA

Una volta recuperata una sequenza, possiamo ad esempio conoscere la frequenza nucleotidica o aminoacidica, e le basi nucleotidiche della guanina e della citosina (contenuto di GC). Il contenuto di una sequenza aiuta anche a determinare alcune proprietà dell'intera molecola, per esempio, l'idrofobicità della basicità acida nelle proteine basate sugli aminoacidi presenti nella sequenza.

L'interesse del contenuto di GC si riferisce alla frazione di Guanina (G) e Citosina (C) nella sequenza, e alcuni genomi, soprattutto tra i batteri, mostrano una differenza significativa su questa scala e variazioni in termini di regioni genomiche. Alcuni Actinobatteri possono avere più del 70% di GC, mentre alcuni Proteobatteri possono avere meno del 20% di GC. Inoltre, il contenuto di GC è anche utilizzato per prevedere la temperatura di *annealing* della sequenza durante gli esperimenti PCR. Questi aspetti rendono importante l'analisi del contenuto di una sequenza che esaminiamo in questa sezione.

Il contenuto di nucleotidi GC è quindi la frazione della sequenza che consiste di G e C, cioè la %(G+C) = (numero di G + numero di C)*100/(lunghezza del genoma). Per esempio, se il genoma è 100 bp, e 20 basi sono G e 21 basi sono C, allora il contenuto di GC è (20 + 21)*100/100 = 41%.

Si può facilmente calcolare il contenuto di GC in base al numero di A, G, C e T nella sequenza del genoma. Per esempio, per la sequenza del genoma del virus DEN-1 sappiamo, usando la funzione `table()`, che il genoma contiene 3426 A, 2240 C, 2770 G e 2299 T. Pertanto, possiamo calcolare il contenuto di GC utilizzando il comando:

```
> (2240+2770)*100/(3426+2240+2770+2299)
[1] 46.66977
```

In alternativa, si può usare la funzione `GC()` del pacchetto *SeqinR*, che dà la frazione di basi nella sequenza che sono G o C:

```
> GC(dengueseq)
[1] 0.4666977
```

Vediamo un altro esempio. Recuperiamo due sequenze di RNA polimerasi (sottounità beta) da GenBank che appartengono a due specie diverse, Mycobacterium tuberculosis (Actinobacterium) e Escherichia coli (Proteobacterium) e verifichiamone il numero totale:

```
> library(seqinr)
> choosebank("genbank")
> actino <- query(listname = "actino", query="SP=Mycobacterium tuberculosis AND
K=rpoB")
> proteo <- query(listname = "proteo", query="SP=Escherichia coli AND K=rpoB")
> length(actino$req)
[1] 1202
> length(proteo$req)
[1] 987
```

Scegliamo una sequenza da ogni specie. Nel nostro caso, JX303316 (la n. 702) nella prima e AJ854258 (la n. 4) nella seconda come segue:

```
> myActino <- getSequence(actino$req[[702]])
> myProteo <- getSequence(proteo$req[[4]])
```

Calcoliamo il numero di ogni base nella sequenza con la funzione `table()`:

```
> table(myActino)
myActino
  a   c   g   t
112 201 228 108
> table(myProteo)
myProteo
   a    c    g    t
1020 1032 1102  908
```

Per ottenere la frazione per ogni base, dividiamo il risultato precedente per la lunghezza della sequenza:

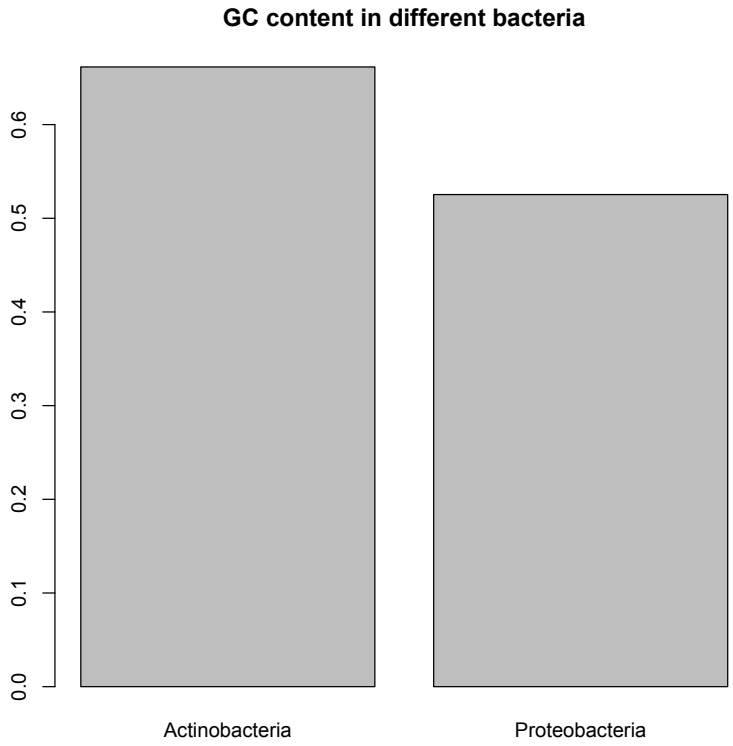
```
> table(myActino)/length(myActino)
myActino
        a         c         g         t
0.1725732 0.3097072 0.3513097 0.1664099
> table(myProteo)/length(myProteo)
myProteo
        a         c         g         t
0.2511078 0.2540620 0.2712949 0.2235352
```

Per calcolare in maniera più semplice il contenuto GC delle sequenze, è sufficiente utilizzare il comando GC del pacchetto *seqinr* come segue:

```
> GC(myActino)
[1] 0.6610169
> GC(myProteo)
[1] 0.525357
```

Confrontiamo il contenuto di GC delle due diverse sequenze con un semplice grafico a barre come mostrato nel comando seguente:

```
> barplot(c(Actinobacteria=GC(myActino), Proteobacteria=GC(myProteo)),
      main="GC content in different bacteria")
```



Possiamo anche conoscere la frequenza di ogni possibile coppia di nucleotidi nella sequenza utilizzando la funzione `count()` come mostrato nell'esempio seguente (naturalmente è possibile farlo anche per le triple e così via scegliendo il giusto valore per l'argomento `wordsize`):

```
> seqinr::count(myActino, wordsize=2)
aa ac ag at ca cc cg ct ga gc gg gt ta tc tg tt
18 40 30 24 36 59 83 22 51 59 69 49  7 43 45 13
> seqinr::count(myProteo, wordsize=2)
 aa  ac  ag  at  ca  cc  cg  ct  ga  gc  gg  gt  ta  tc  tg  tt
329 268 225 197 209 235 342 246 312 269 223 298 169 260 312 167
```

In questo esempio abbiamo usato la RNA polimerasi beta di due specie batteriche (JX303316 e AJ854258) che rappresentano proteine simili e sono di lunghezza comparabile. Le funzioni utilizzate si basano principalmente sulla corrispondenza dei caratteri e sul conteggio, che può essere effettuato per parole di dimensioni diverse, ad esempio 2 (per il calcolo del contenuto GC), 3 (usato poiché rappresenta le terzine nei codoni) e così via.

In questa sezione abbiamo parlato soprattutto di sequenze nucleotidiche; le sequenze proteiche saranno trattate in dettaglio nel seguito.

1.11 "Parole" DNA

Oltre alla frequenza di ciascuno dei singoli nucleotidi ("A", "G", "T", "C") in una sequenza di DNA, è anche interessante conoscere la frequenza delle "parole" più lunghe del DNA. I singoli nucleotidi sono parole del DNA di lunghezza 1, ma potremmo anche voler conoscere la frequenza delle parole del DNA che sono lunghe 2 nucleotidi (cioè "AA", "AG", "AC", "AT", "CA", "CG", "CC", "CT", "GA", "GG", "GC", "GT", "TA", "TG", "TC" e "TT"), 3 nucleotidi (ad es. "AAA", "AAT", "ACG", …), ecc.

Per trovare il numero di occorrenze di parole del DNA di una particolare lunghezza, possiamo usare la funzione `count()` del pacchetto R *SeqinR*. Per esempio, per trovare il numero di occorrenze di parole del DNA che sono lunghe 1 nucleotide nella sequenza *dengueseq*, digitiamo:

```
> count(dengueseq, 1)
  a    c    g    t
3426 2240 2770 2299
```

che come previsto, ci dà il numero di occorrenze dei singoli nucleotidi. Per trovare il numero di occorrenze delle parole del DNA che sono lunghe 2 nucleotidi, digitiamo:

```
> count(dengueseq, 2)
  aa   ac   ag   at   ca   cc   cg   ct   ga   gc   gg   gt   ta   tc   tg   tt
1108  720  890  708  901  523  261  555  976  500  787  507  440  497  832  529
```

Si noti che per impostazione predefinita la funzione `count()` include tutte le parole del DNA che si sovrappongono in una sequenza. Pertanto, ad esempio, si ritiene che la sequenza "ATG" contenga due parole lunghe due nucleotidi: "AT" e "TG".

Il risultato della funzione `count()` è un oggetto tabellare. Ciò significa che si possono usare le doppie parentesi quadre per estrarre i valori degli elementi dalla tabella. Per esempio, per estrarre il valore del terzo elemento (il numero di G nella sequenza del virus DEN-1), si può digitare:

```
> denguetable <- count(dengueseq,1)
> denguetable[[3]]
 [1] 2770
```

Il comando `denguetable[[3]]` estrae il terzo elemento della tabella prodotta da `count(dengueseq,1)`, memorizzata nella variabile *denguetable*.

In alternativa, è possibile trovare il valore dell'elemento della tabella nella colonna "g" digitando:

```
> denguetable[["g"]]
 [1] 2770
```

2. Analisi statistica delle sequenze biologiche — Parte II

2.1 Lettura di sequenze con il pacchetto *SeqinR*

Nel capitolo precedente abbiamo visto come cercare e scaricare i dati di sequenza per un dato *accession* dal database NCBI. Abbiamo ad esempio scaricato i dati di sequenza del virus Dengue DEN-1 (*accession* NCBI NC_001477) e li abbiamo memorizzati nel file "den1.fasta".

Possiamo recuperare direttamente la sequenza NCBI NC_001477 anche utilizzando la funzione `getncbiseq()` (vedi Appendice), che utilizza il pacchetto *SeqinR* per scandire i principali database ACNUC e ritrovare la sequenza con l'*accession* indicato:

```
> dengueseq <- getncbiseq("NC_001477")
> dengueseq
    [1] "a" "g" "t" "t" "g" "t" "t" "a" "g" "t" "c" "t" "a" "c" "g" "t" "g" "g" "a" "c" "c"
   [22] "g" "a" "c" "a" "a" "g" "a" "a" "c" "a" "g" "t" "t" "t" "c" "g" "a" "a" "t" "c" "g"
…
[10711] "t" "g" "g" "t" "g" "c" "t" "g" "t" "t" "g" "a" "a" "t" "c" "a" "a" "c" "a" "g" "g"
[10732] "t" "t" "c" "t"
> write.fasta(names="DEN-1", sequences=dengueseq, file.out="den1.fasta")
```

Una volta scaricati e memorizzati i dati della sequenza, li possiamo in un secondo tempo leggere in R utilizzando la funzione `read.fasta()` del pacchetto *SeqinR*. Ad esempio, per rileggere la sequenza del virus DEN-1 dal file "den1.fasta" basta digitare:

```
> library("seqinr")                              # Load the SeqinR package.
> dengue <- read.fasta(file = "den1.fasta")      # Read in the file "den1.fasta".
> dengueseq <- dengue[[1]]                       # Put the sequence in a vector␣
↪called "dengueseq".
```

Una volta recuperata una sequenza dal database NCBI e memorizzata in una variabile vettore, è possibile estrarre la successione della sequenza digitando il nome del vettore (ad es. *dengueseq*) seguito dalle parentesi quadre che contengono gli indici per quei nucleotidi. Per esempio, per ottenere i nucleotidi 452–535 del genoma del virus DEN-1, possiamo digitare:

```
> dengueseq[452:535]
 [1] "c" "g" "a" "g" "g" "g" "g" "g" "a" "g" "a" "g" "c" "c" "g" "c" "a" "c" "a"
[20] "t" "g" "a" "t" "a" "g" "t" "t" "a" "g" "c" "a" "a" "g" "c" "a" "g" "g" "a"
[39] "a" "a" "g" "a" "g" "g" "a" "a" "a" "a" "t" "c" "a" "c" "t" "t" "t" "t" "g"
[58] "t" "t" "t" "a" "a" "g" "a" "c" "c" "t" "c" "t" "g" "c" "a" "g" "g" "t" "g"
[77] "t" "c" "a" "a" "c" "a" "t" "g"
```

2.2 Variazione locale nel contenuto di GC

Abbiamo visto in precedenza che, per scoprire il contenuto GC di una sequenza genomica, si può usare la funzione `GC()` del pacchetto *SeqinR*. Per esempio, per trovare il contenuto GC della sequenza del virus DEN-1 che abbiamo memorizzato nel vettore *dengueseq*, basta digitare:

```
> GC(dengueseq)
[1] 0.4666977
```

L'output di `GC()` è la frazione di nucleotidi in una sequenza che sono G o C, quindi per convertirla in una percentuale dobbiamo moltiplicarla per 100. Così, il contenuto di GC del genoma del virus DEN-1 è circa 0,467 cioè 46,7%.

Anche se il contenuto di GC dell'intera sequenza del genoma del virus DEN-1 è circa il 46,7%, probabilmente c'è una variazione locale del contenuto di GC all'interno del genoma. Cioè, alcune regioni della sequenza genomica possono avere un contenuto di GC molto più alto o molto più basso del 46,7%. Le fluttuazioni locali del contenuto di GC all'interno della sequenza genomica possono fornire diverse informazioni interessanti, ad esempio, possono rivelare casi di trasferimento orizzontale o rivelare alterazioni nella mutazione.

Se un pezzo di DNA si è spostato per trasferimento orizzontale dal genoma di una specie a basso contenuto di GC ad una specie ad alto contenuto di GC, il pezzo di DNA trasferito orizzontalmente potrebbe essere rilevato come una regione con un contenuto di GC insolitamente basso nel genoma ricevente ad alto contenuto di GC.

D'altra parte, una regione con un contenuto di GC insolitamente basso in un genoma altrimenti ad alto contenuto di GC potrebbe anche sorgere a causa di distorsioni nella mutazione in quella regione del genoma; per esempio, se le mutazioni da G/C a T/A sono più comuni per qualche motivo in quella regione che nel resto del genoma.

2.3 Analisi "sliding window" del contenuto di GC

Per studiare la variazione locale del contenuto di GC all'interno di una sequenza genomica, potremmo calcolare il contenuto di GC per piccoli pezzi della sequenza stessa. La sequenza del genoma del virus DEN-1 è lunga 10.735 nucleotidi. Per studiare la variazione del contenuto di GC all'interno della sequenza del genoma, potremmo calcolare il contenuto di GC di pezzi del genoma del virus DEN-1, ad esempio, per ogni pezzo di 2.000 nucleotidi della sequenza:

```
> GC(dengueseq[1:2000])       # Calculate the GC content of nucleotides 1-2000 of␣
↪the Dengue genome
[1] 0.465
> GC(dengueseq[2001:4000])    # Calculate the GC content of nucleotides 2001-4000␣
↪of the Dengue genome
[1] 0.4525
> GC(dengueseq[4001:6000])    # Calculate the GC content of nucleotides 4001-6000␣
↪of the Dengue genome
[1] 0.4705
> GC(dengueseq[6001:8000])    # Calculate the GC content of nucleotides 6001-8000␣
↪of the Dengue genome
[1] 0.479
> GC(dengueseq[8001:10000])   # Calculate the GC content of nucleotides 8001-10000␣
↪of the Dengue genome
[1] 0.4545
> GC(dengueseq[10001:10735]) # Calculate the GC content of nucleotides 10001-10735␣
↪of the Dengue genome
[1] 0.4993197
```

Dall'output precedente si vede che la regione 1–2.000 del genoma del virus DEN-1 ha un contenuto di GC del 46,5%, mentre la regione 2.001–4.000 ha un contenuto di GC di circa il 45,3%. Quindi, sembra esserci una certa variazione locale nel contenuto di GC all'interno della sequenza del genoma del virus Dengue.

Invece di digitare i comandi di cui sopra, possiamo usare un ciclo per eseguire gli stessi calcoli prendendo pezzi di 2.000 nucleotidi del genoma del virus DEN-1 e calcolando il contenuto di GC di ogni parte:

```
> starts <- seq(1, length(dengueseq)-2000, by = 2000)
> starts
[1]    1 2001 4001 6001 8001
> n <- length(starts)     # Find the length of the vector "starts"
> for (i in 1:n) {
        chunk <- dengueseq[starts[i]:(starts[i]+1999)]
        chunkGC <- GC(chunk)
        print (chunkGC)
    }
[1] 0.465
[1] 0.4525
[1] 0.4705
[1] 0.479
[1] 0.4545
```

Il comando "starts <- seq(1, length(dengueseq)-2000, by=2000)" memorizza il risultato della funzione `seq()` nel vettore *starts*, che contiene i valori 1, 2001, 4001, 6001 e 8001.

Abbiamo impostato la variabile *n* in modo che sia uguale al numero di elementi del vettore *starts*, cioè 5. La riga `for (i in 1:n)` significa che il contatore *i* prenderà valori 1–5 nelle iterazioni successive del ciclo. Il ciclo di cui sopra è distribuito su più righe; tuttavia, R non eseguirà i comandi all'interno del ciclo fino a quando non viene digitato l'ultima parentesi di chiusura "`}`" alla fine del ciclo `for` e premuto "Invio".

Ciascuno dei tre comandi all'interno del ciclo viene eseguito ad ogni iterazione. Nella prima iterazione, *i* vale 1 e la variabile *chunk* viene utilizzata per memorizzare la regione dei nucleotidi 1–2.000; quindi il contenuto di GC di quella regione viene calcolato e memorizzato nella variabile *chunkGC* il cui valore viene visualizzato.

Nella seconda iterazione, *i* vale 2, la variabile *chunkGC* viene usata per memorizzare la regione 2.001–4.000 della sequenza del virus Dengue, il contenuto di GC di quella regione viene calcolato e memorizzato nella variabile *chunkGC*, e il valore di *chunkGC* viene stampato. Il ciclo continua fino a quando il valore di *i* è 5, con la stampa del contenuto di GC della regione 8.001–10.000.

Si noti che il ciclo termina sulla regione 8.001–10.000, invece di continuare l'esame dei nucleotidi 10.001–12.000. Ciò poiché la lunghezza della sequenza del genoma del virus di Dengue è di soli 10.735 nucleotidi, quindi non vi è una regione completa di 2.000 nucleotide dal nucleotide 10.001 alla fine della sequenza al nucleotide 10.735.

L'analisi della variazione locale del contenuto di GC vista è nota come "analisi a finestra scorrevole" (*sliding window*) del contenuto di GC. Calcolando il contenuto di GC in ogni pezzo di 2.000 nucleotidi del genoma

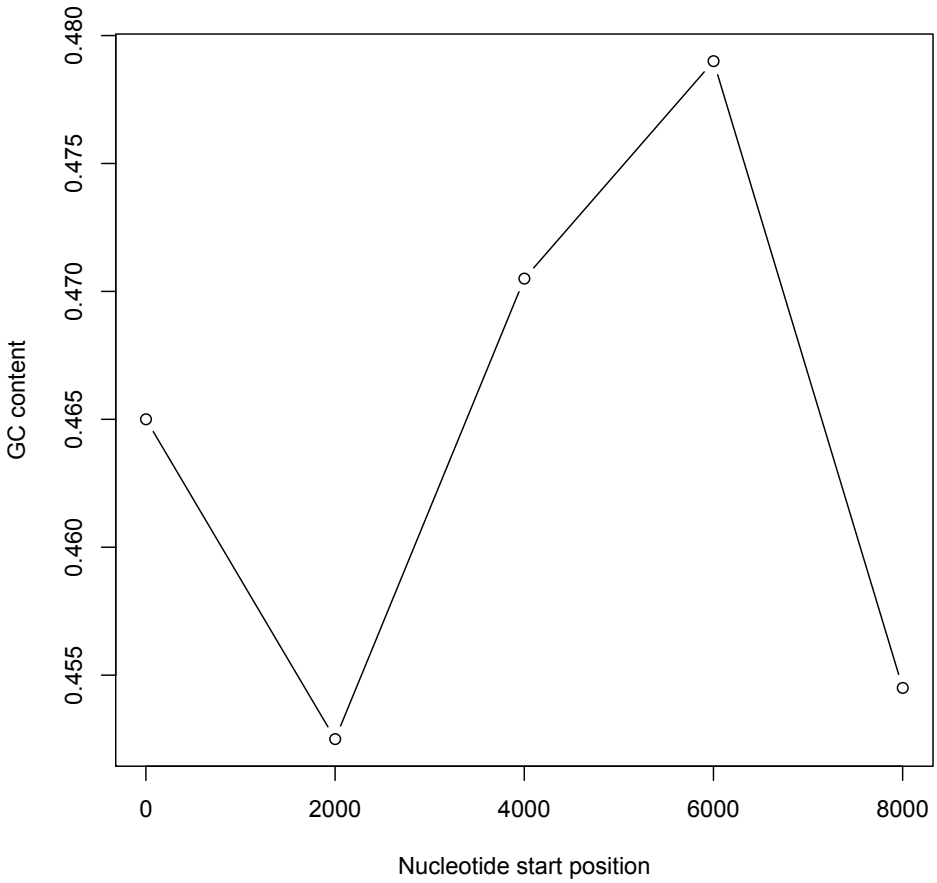
del virus Dengue, si fa scorrere una finestra di 2.000 nucleotidi lungo la sequenza di DNA dall'inizio alla fine, calcolando il contenuto di GC in ogni finestra non sovrapposta.

L'analisi del contenuto di GC mostrata è una versione leggermente semplificata del metodo solitamente eseguito in bioinformatica. In questa versione semplificata, abbiamo calcolato il contenuto di GC in finestre non sovrapposte lungo una sequenza di DNA. Tuttavia, è più usuale calcolare il contenuto di GC in finestre sovrapposte, anche se questo rende il codice leggermente più complicato.

2.4 Grafico "sliding window" del contenuto di GC

È possibile utilizzare i dati generati da un'analisi *sliding window* per creare una serie di grafici del contenuto di GC. Per creare tali grafici, si traccia il contenuto locale di GC in ogni finestra del genoma, rispetto alla posizione nucleotidica dell'inizio di ogni finestra. Possiamo creare un grafico a finestra scorrevole (*sliding window*) del contenuto di GC digitando:

```
> starts <- seq(1, length(dengueseq)-2000, by = 2000)
> n <- length(starts)     # Find the length of the vector "starts"
> chunkGCs <- numeric(n) # Make a vector of the same length as vector "starts",␣
↪but just containing zeroes
> for (i in 1:n) {
        chunk <- dengueseq[starts[i]:(starts[i]+1999)]
        chunkGC <- GC(chunk)
        print(chunkGC)
        chunkGCs[i] <- chunkGC
    }
> plot(starts,chunkGCs,type="b",xlab="Nucleotide start position",ylab="GC content")
```



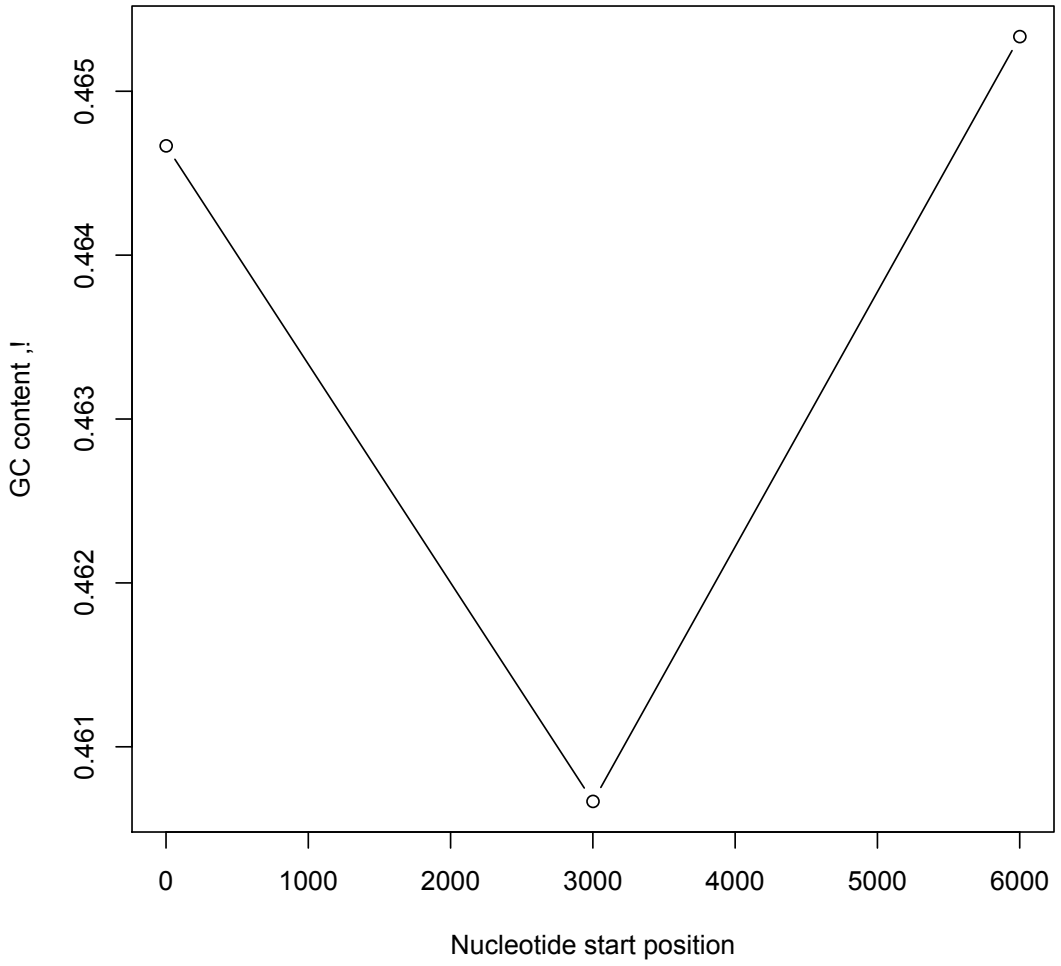
Nel codice di cui sopra, la riga "`chunkGCs <- numeric(n)`" crea un nuovo vettore *chunkGCs* che ha lo stesso numero di elementi del vettore *starts* (5 elementi). Questo vettore viene poi utilizzato all'interno del ciclo per memorizzare il contenuto GC di ogni pezzo di DNA.

Dopo il ciclo, il vettore *starts* può essere tracciato rispetto al vettore *chunkGCs* usando la funzione `plot()`, per ottenere un grafico (a finestra scorrevole) del contenuto di GC rispetto alla posizione del nucleotide nella sequenza del genoma.

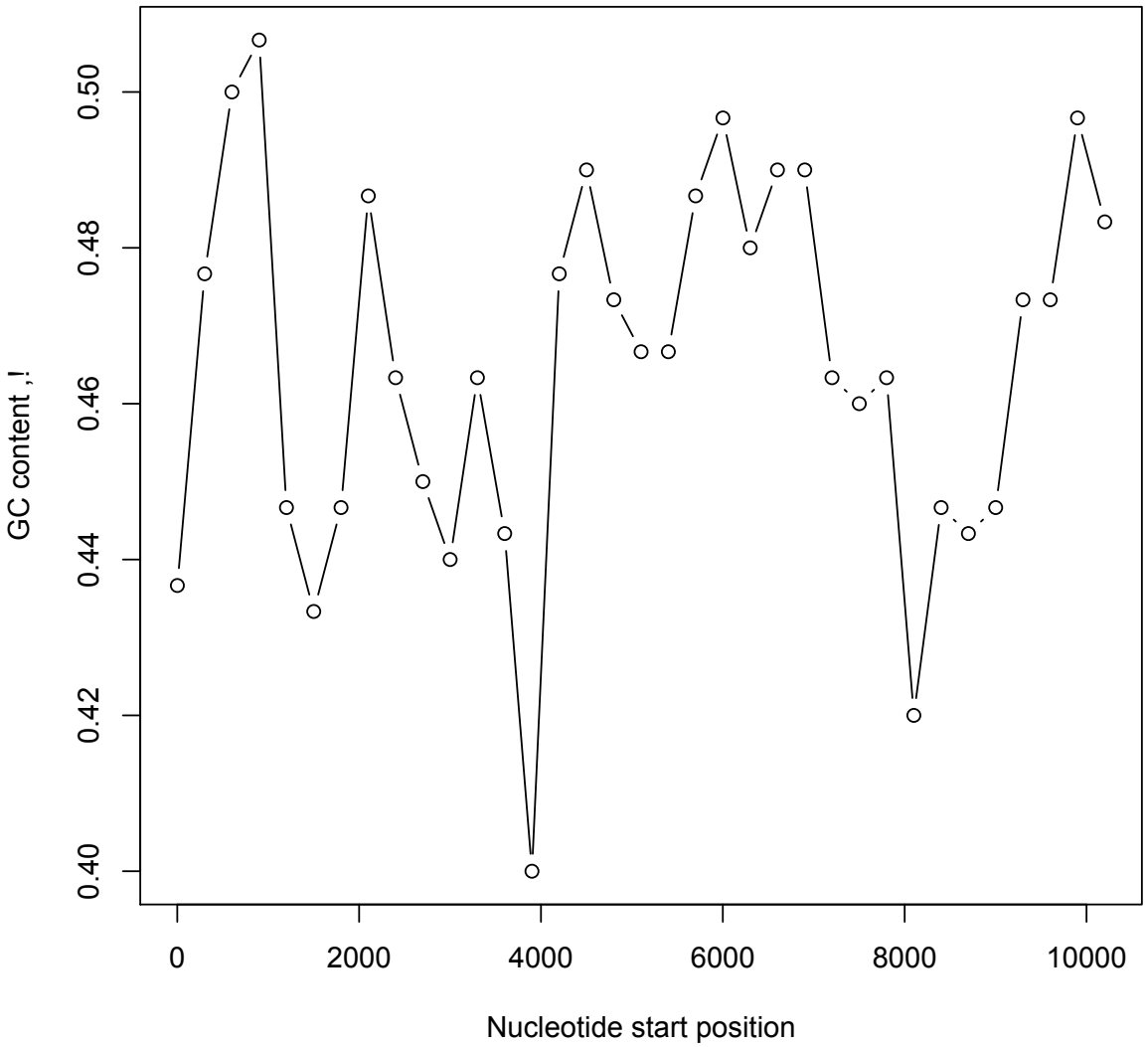
Si potrebbe usare il codice precedente per creare grafici *sliding window* del contenuto di GC di genomi di specie diverse, usando diverse dimensioni di finestra. La funzione `slidingwindowplot()` (vedi Appendice) consente di tracciare tali grafici, fornendo in input la dimensione della finestra che l'utente vuole usare e la sequenza da studiare.

Questa funzione può quindi essere usata per fare dei grafici *sliding window* del contenuto di GC per diverse sequenze di DNA, con diverse dimensioni di finestra. Per esempio, si possono creare due diversi grafici della sequenza del genoma del virus DEN-1, usando dimensioni di finestra di 3.000 e 300 nucleotidi, rispettivamente:

```
> slidingwindowplot(3000, dengueseq)
```



```
> slidingwindowplot(300, dengueseq)
```



2.5 Parole DNA sovra e sottorappresentate (statistica *Rho*)

Nel capitolo precedente abbiamo visto che la funzione `count()` del pacchetto *SeqinR* può calcolare la frequenza di tutte le parole di una certa lunghezza in una sequenza di DNA. Ad esempio, per conoscere la frequenza di tutte le parole di 2 nucleotidi di lunghezza nella sequenza del genoma del virus Dengue, si può digitare:

```
> count(dengueseq, 2)
  aa   ac   ag   at   ca   cc   cg   ct   ga   gc   gg   gt   ta   tc   tg   tt
1108  720  890  708  901  523  261  555  976  500  787  507  440  497  832  529
```

È interessante identificare le parole di due nucleotidi ("dinucleotidi", cioè "AT", "AC", ecc.) che sono sovra o sottorappresentate in una sequenza di DNA. Se una particolare parola è sovrarappresentata in una sequenza, significa che si verifica molte più volte di quanto ci si sarebbe aspettati per puro caso. Allo stesso

modo, se una particolare parola del DNA è sottorappresentata in una sequenza, significa che si verifica molte meno volte nella sequenza di quanto ci si sarebbe atteso.

Una statistica chiamata ρ (Rho) è usata per misurare quanto è sovrarappresentata o sottorappresentata una particolare parola di DNA. Per un dinucleotide ρ è calcolato come: $\rho(xy) = f_{xy} / (f_x \cdot f_y)$, dove f_{xy} e f_x sono le frequenze delle parole DNA xy e x nella sequenza di DNA in studio. Ad esempio, il valore di ρ per la parola DNA "TA" può essere calcolato come: $\rho(TA) = f_{TA} / (f_T \cdot f_A)$, dove f_{TA}, f_T e f_A sono le frequenze delle parole "TA", "T" e "A" nella sequenza di DNA.

L'idea alla base della statistica ρ è che, se una sequenza di DNA ha una frequenza f_x di una parola di DNA a 1 nucleotide "x", e una frequenza f_y di una parola di DNA a 1 nucleotide "y", allora ci aspettiamo che la frequenza della parola di DNA a 2 nucleotidi "xy" sia $f_x \cdot f_y$. Vale a dire, le frequenze delle parole a 2 nucleotidi in una sequenza dovrebbero essere uguali ai prodotti delle frequenze specifiche dei due nucleotidi che li compongono. Se questo fosse vero, allora ρ sarebbe uguale a 1. Se troviamo che ρ è molto maggiore di 1 per un particolare dinucleotide, ciò indica che quella parola a 2 nucleotidi è molto più comune in quella sequenza di quanto ci si aspettasse (cioè è *sovrarappresentata*).

Supponiamo per esempio che la sequenza abbia solo il 5% di T (cioè $f_T = 0{,}05$). In una sequenza di DNA casuale con 5% di T, ci si aspetterebbe di vedere la parola "TT" molto raramente. Infatti, ci aspetteremmo che solo lo $0{,}05 \cdot 0{,}05 = 0{,}0025$ (0,25%) di parole a 2 nucleotidi siano TT (cioè ci aspettiamo $f_{TT} = f_T \cdot f_T$). Questo perché le T sono rare, quindi ci si aspetta che siano adiacenti l'una all'altra molto raramente se le poche T sono sparse in modo casuale nel DNA. Pertanto, se si vedono molti dinucleotidi TT nella sequenza di input reale (ad es. $f_{TT} = 0{,}3$, quindi $\rho = 0{,}3/0{,}0025 = 120$), si sospetterebbe che la selezione naturale abbia agito per aumentare il numero di occorrenze della parola TT nella sequenza (presumibilmente perché ha una qualche funzione biologica benefica).

Per trovare i dinucleotidi sovra e sottorappresentati nella sequenza del virus DEN-1, possiamo calcolare la statistica ρ per ogni dinucleotide nella sequenza. Per esempio, dato il numero di occorrenze dei singoli nucleotidi A, C, G e T nella sequenza di Dengue, e il numero di occorrenze della parola DNA GC nella sequenza (500, come visto all'inizio della sezione), possiamo calcolare il valore di ρ per il dinucleotide "GC" usando la formula $\rho(GC) = f_{GC} / (f_G \cdot f_C)$, dove f_{GC}, f_G e f_C sono le frequenze delle parole "GC", "G" e "C" nella sequenza di DNA:

```
> count(dengueseq, 1) # Get the number of occurrences of 1-nucleotide DNA words
    a     c     g     t
 3426 2240 2770 2299
> 2770/(3426+2240+2770+2299) # Get fG
[1] 0.2580345
> 2240/(3426+2240+2770+2299) # Get fC
[1] 0.2086633
> count(dengueseq, 2) # Get the number of occurrences of 2-nucleotide DNA words
  aa   ac   ag   at   ca   cc   cg   ct   ga   gc   gg   gt   ta   tc   tg   tt
1108  720  890  708  901  523  261  555  976  500  787  507  440  497  832  529
> 500/(1108+720+890+708+901+523+261+555+976+500+787+507+440+497+832+529) # Get fGC
[1] 0.04658096
> 0.04658096/(0.2580345*0.2086633) # Get rho(GC)
[1] 0.8651364
```

Calcoliamo un valore di ρ(GC) di circa 0,865. Ciò significa che la parola DNA "GC" è circa 0,865 volte più comune nella sequenza del virus DEN-1 rispetto al previsto. Cioè, sembra essere leggermente sottorappresentata.

Si noti che se il rapporto tra la frequenza osservata e quella prevista di una particolare parola del DNA è molto basso o molto alto, allora sospetteremmo che ci sia una sottorappresentazione statistica o una sovrarappresentazione di quella parola del DNA. Per effettuare lo stesso calcolo in maniera immediata è disponibile la funzione `rho()` del pacchetto *seqinr*:

```
> rho(dengueseq)
       aa        ac        ag        at        ca        cc        cg        ct
ga
1.0134622 1.0072555 1.0068514 0.9650492 1.2604683 1.1190466 0.4516013 1.1570403
1.1041427
       gc        gg        gt        ta        tc        tg        tt
0.8651367 1.1011784 0.8547355 0.5997481 1.0361244 1.4026428 1.0745342
```

3. Database di sequenze biologiche

3.1 La banca dati NCBI di sequenze biologiche

Tutte le sequenze di genoma pubblicate sono disponibili su Internet, in quanto è un requisito di ogni rivista scientifica che ogni sequenza di DNA o RNA o proteina pubblicata sia depositata in un database pubblico. Le principali risorse per l'archiviazione e la distribuzione dei dati delle sequenze sono tre grandi banche dati: la banca dati NCBI (https://www.ncbi.nlm.nih.gov), la banca dati del Laboratorio Europeo di Biologia Molecolare (EMBL-EBI) (https://www.ebi.ac.uk) e la banca dati del DNA del Giappone (DDBJ) (https://www.ddbj.nig.ac.jp). Queste banche dati raccolgono tutti i dati disponibili al pubblico sulle sequenze di DNA, RNA e proteine e li rendono disponibili gratuitamente. Si scambiano dati ogni notte, quindi contengono essenzialmente gli stessi dati.

In questo capitolo parleremo della banca dati NCBI. Si noti, tuttavia, che essa contiene essenzialmente gli stessi dati di EMBL e DDBJ.

Le sequenze nella banca dati NCBI (o EMBL/DDBJ) sono identificate da un numero di registrazione (*accession number*). Si tratta di un numero unico associato ad una sola sequenza. Ad esempio, il numero di accesso NC_001477 corrisponde alla sequenza del genoma del virus Dengue DEN-1: tale numero identifica la sequenza ed è riportato nei documenti scientifici che la descrivono.

Oltre alla sequenza stessa, per ogni sequenza la banca dati memorizza anche alcuni dati di annotazione aggiuntivi, come il nome della specie da cui proviene, i riferimenti alle pubblicazioni che descrivono tale sequenza, ecc. Alcuni di questi dati di annotazione sono stati aggiunti dalla persona che ha tracciato la sequenza e l'ha presentata al database dell'NCBI, mentre altri possono essere stati aggiunti successivamente da un curatore umano che lavora per l'NCBI.

La banca dati NCBI contiene diversi database, i più importanti dei quali sono:

- il database NCBI dei *Nucleotidi*: contiene sequenze di DNA e RNA;
- il database NCBI delle *Proteine*: contiene sequenze proteiche;
- EST: contiene *Expressed Sequence Tag* (tag di sequenza espressa), che sono brevi sequenze derivate da mRNA;
- il database NCBI del *Genoma*: contiene sequenze di DNA per interi genomi;
- *PubMed*: contiene dati su pubblicazioni scientifiche.

3.2 Ricerca di un *accession number* nella banca dati NCBI

Nel capitolo **1. Analisi statistica delle sequenze DNA — Parte I**, abbiamo visto come ottenere un file FASTA contenente la sequenza del DNA corrispondente ad un particolare numero di accesso, ad es. il numero NC_001477 (la sequenza del genoma del virus Dengue DEN-1) tramite il sito web della NCBI.

Come spiegato in precedenza, il formato FASTA è un formato di file comunemente utilizzato per memorizzare le informazioni sulla sequenza. La prima riga inizia con il carattere **>** seguito da un nome e/o una descrizione della sequenza. Le righe successive contengono la sequenza stessa.

```
>mysequence1
ACATGAGACAGACAGACCCCCAGAGACAGACCCCTAGACACAGAGAGAG
TATGCAGGACAGGGTTTTTGCCCAGGGTGGCAGTATG
```

Un file FASTA può contenere più di una sequenza. Se un file FASTA contiene molte sequenze, allora per ogni sequenza vi sarà una riga di intestazione che inizia con **>** seguita dalla sequenza stessa.

```
>mysequence1
ACATGAGACAGACAGACCCCCAGAGACAGACCCCTAGACACAGAGAGAG
TATGCAGGACAGGGTTTTTGCCCAGGGTGGCAGTATG
>mysequence2
AGGATTGAGGTATGGGTATGTTCCCGATTGAGTAGCCAGTATGAGCCAG
AGTTTTTTACAAGTATTTTTCCCAGTAGCCAGAGAGAGAGTCACCCAGT
ACAGAGAGC
```

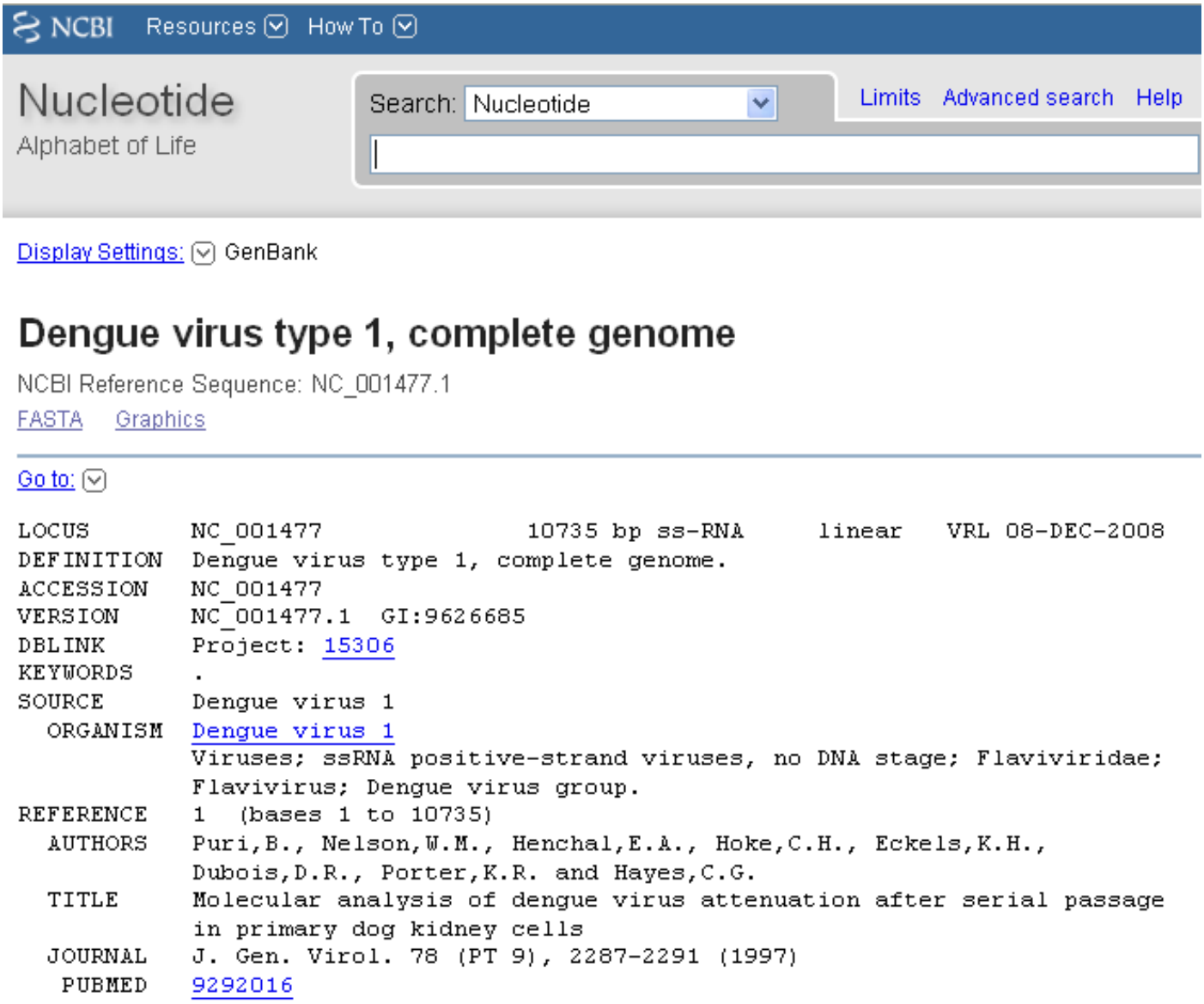
3.3 Il formato NCBI per le sequenze biologiche

Come accennato in precedenza, per ogni sequenza la banca dati NCBI memorizza alcune informazioni supplementari come le specie da cui proviene, le pubblicazioni che descrivono la sequenza, ecc. Queste informazioni sono memorizzate nel record NCBI per la sequenza, che può essere visualizzato cercando nella banca dati NCBI il numero di accesso per quella sequenza. Le voci NCBI per le sequenze sono memorizzate in un formato particolare, noto come formato NCBI.

Per visualizzare la voce NCBI per il virus DEN-1 (che ha *accession* NC_001477), seguire questi passi:

1. Andare al sito web NCBI (https://www.ncbi.nlm.nih.gov).
2. Cercare il numero di accesso NC_001477.
3. Nella pagina dei risultati, se la sequenza da ricercare corrisponde ad una sequenza nucleotidica (DNA o RNA), dovrebbe esserci un risultato nel database dei nucleotidi; basta quindi cliccare sulla parola 'Nucleotide' per visualizzare la voce NCBI corrispondente. Allo stesso modo, se la sequenza da ricercare corrisponde ad una sequenza proteica, dovrebbe esserci un risultato nel database delle proteine, e basta cliccare sulla parola 'Protein' per visualizzare la voce NCBI relativa.
4. Dopo aver cliccato su 'Nucleotide' o 'Protein' nella fase precedente, apparirà la voce NCBI per il numero di accesso ricercato.

Ad esempio, la voce NCBI per la sequenza del genoma del virus DEN-1 (*accession* NCBI NC_001477) appare così:



La voce NCBI per un *accession* contiene molte informazioni sulla sequenza, come i documenti che la descrivono, le caratteristiche della sequenza, ecc. Il campo "DEFINITION" fornisce una breve descrizione della sequenza. Il campo "ORGANISM" identifica la specie da cui proviene la sequenza. Il campo "REFERENCE" contiene pubblicazioni scientifiche che descrivono la sequenza. Il campo "FEATURES" contiene informazioni sulla localizzazione delle caratteristiche di interesse all'interno della sequenza, come le sequenze di regolazione o i geni che si trovano all'interno della sequenza. Il campo "ORIGIN" fornisce la sequenza stessa.

3.4 Il database NCBI *RefSeq*

Quando si effettuano ricerche nella banca dati NCBI, è importante tenere presente che la banca dati può contenere sequenze ridondanti per lo stesso gene che sono state sequenziate da laboratori diversi (perché molti laboratori diversi hanno sequenziato il gene e hanno inviato le loro sequenze alla banca dati).

Ci sono anche molti tipi diversi di sequenze di nucleotidi e di proteine nella banca dati NCBI. Per quanto riguarda le sequenze nucleotidiche, alcune sono sequenze di DNA genomiche intere, altre possono essere

mRNA, e altre ancora possono essere sequenze di qualità inferiore, come i tag di sequenza espressa (EST, che derivano da parti di mRNA), o sequenze di DNA di contigui derivanti da progetti genomici. Inoltre, alcune sequenze possono essere curate manualmente in modo che le voci associate contengano informazioni aggiuntive, ma la maggior parte delle sequenze non sono curate.

Come detto sopra, la banca dati NCBI spesso contiene informazioni ridondanti per un gene, contiene sequenze di qualità variabile, e contiene sia dati non curati che dati curati. Di conseguenza, l'NCBI ha creato un database speciale chiamato *RefSeq* (Reference Sequence Database), che è un sottoinsieme della banca dati NCBI. I dati in RefSeq sono curati manualmente, sono dati di sequenza di alta qualità e non sono ridondanti: questo significa che ogni gene (o *splice-form*, forma di giunzione di un gene, nel caso degli eucarioti), proteina o sequenza genomica è rappresentata una sola volta.

I dati in RefSeq sono curati e sono di qualità molto più elevata rispetto al resto del Sequence Database NCBI. Comunque, purtroppo, a causa dell'alto livello di cura manuale richiesto, RefSeq non copre tutte le specie, e non è completo per le specie che sono coperte finora.

Si può facilmente capire che una sequenza proviene da RefSeq perché il suo numero di accesso inizia con una particolare sequenza di lettere: gli *accession* per sequenze RefSeq corrispondenti a record proteici di solito iniziano con "NP_", e quelli di sequenze complete del genoma curate da RefSeq di solito iniziano con "NC_" o "NS_".

3.5 Interrogazione della banca dati NCBI

Potrebbe essere necessario interrogare la banca dati NCBI per trovare particolari sequenze o un insieme di sequenze che corrispondono a determinati criteri, come ad esempio:

- la sequenza con l'*accession* NC_001477;
- le sequenze pubblicate in Nature 460:352-358;
- tutte le sequenze di *Chlamydia trachomatis*;
- sequenze presentate da Matthew Berriman;
- sequenze di flagelline o fibrinogeni;
- il gene della glutammina sintetasi da *Mycobacteriuma leprae*;
- la regione di controllo a monte (*upstream control region*) del gene *Mycobacterium leprae dnaA*;
- la sequenza della proteina Mycobacterium leprae dnaA;
- la sequenza genomica del *Trypanosoma cruzi*;
- tutte le sequenze nucleotidiche umane associate alla malaria.

Ci sono due modi principali per interrogare il database NCBI per trovare questi insiemi di sequenze. La prima possibilità è quella di effettuare ricerche sul sito web dell'NCBI. La seconda possibilità è quella di effettuare ricerche da R (ad es. mediante i pacchetti R *seqinr* e *rentrez*).

Di seguito spiegheremo come utilizzare il primo metodo per effettuare le ricerche sul database NCBI. In generale, i due metodi dovrebbero dare lo stesso risultato, ma in alcuni casi non lo danno, per vari motivi essenzialmente dipendenti dalla modalità di ricerca implementate dal pacchetto utilizzato.

3.5.1 Interrogare la banca dati NCBI tramite il sito web

Se si effettuano ricerche sul sito web NCBI, per restringere le ricerche a specifici tipi di sequenze o a specifici organismi è necessario utilizzare i *tag di ricerca*. Ad esempio, i tag di ricerca "[PROP]" e "[ORGN]" consentono di limitare la ricerca a un sottoinsieme specifico del database delle sequenze NCBI o alle sequenze di una particolare tassonomia, rispettivamente. Ecco un elenco di tag di ricerca utili, che spiegheremo di seguito come utilizzare:

Search tag	Example	Restricts your search to sequences:
[AC]	NC_001477[AC]	With a particular accession number
[ORGN]	Fungi[ORGN]	From a particular organism or taxon
[PROP]	biomol_mRNA[PROP]	Of a specific type (eg. mRNA) or from a specific database (eg. RefSeq)
[JOUR]	Nature[JOUR]	Described in a paper published in a particular journal
[VOL]	531[VOL]	Described in a paper published in a particular journal volume
[PAGE]	27[PAGE]	Described in a paper with a particular start-page in a journal
[AU]	"Smith J"[AU]	Described in a paper, or submitted to NCBI, by a particular author

Per effettuare ricerche nella banca dati NCBI, è necessario prima di tutto andare sul sito web dell'NCBI e digitare la query di ricerca nella casella di ricerca in alto. Ad esempio, per cercare tutte le sequenze di *Fungi*, si deve digitare "Fungi[ORGN]".

È possibile combinare i tag di ricerca di cui sopra utilizzando "AND", per effettuare ricerche più complesse. Per esempio, per trovare tutte le sequenze mRNA di *Fungi*, si può digitare "Fungi[ORGN] AND biomol_mRNA[PROP]" nella casella di ricerca.

Allo stesso modo, è anche possibile combinare i tag di ricerca utilizzando "OR", ad esempio, per cercare tutte le sequenze mRNA di funghi o batteri, si può digitare "(Fungi[ORGN] OR Bacteria[ORGN]) AND biomol_mRNA[PROP]". Si noti che è necessario mettere delle parentesi attorno a "Fungi[ORGN] OR Bacteria[ORGN]" per specificare che la parola "OR" si riferisce a questi due tag di ricerca.

Ecco alcuni esempi di ricerche, alcune delle quali effettuate combinando i termini di ricerca con "AND":

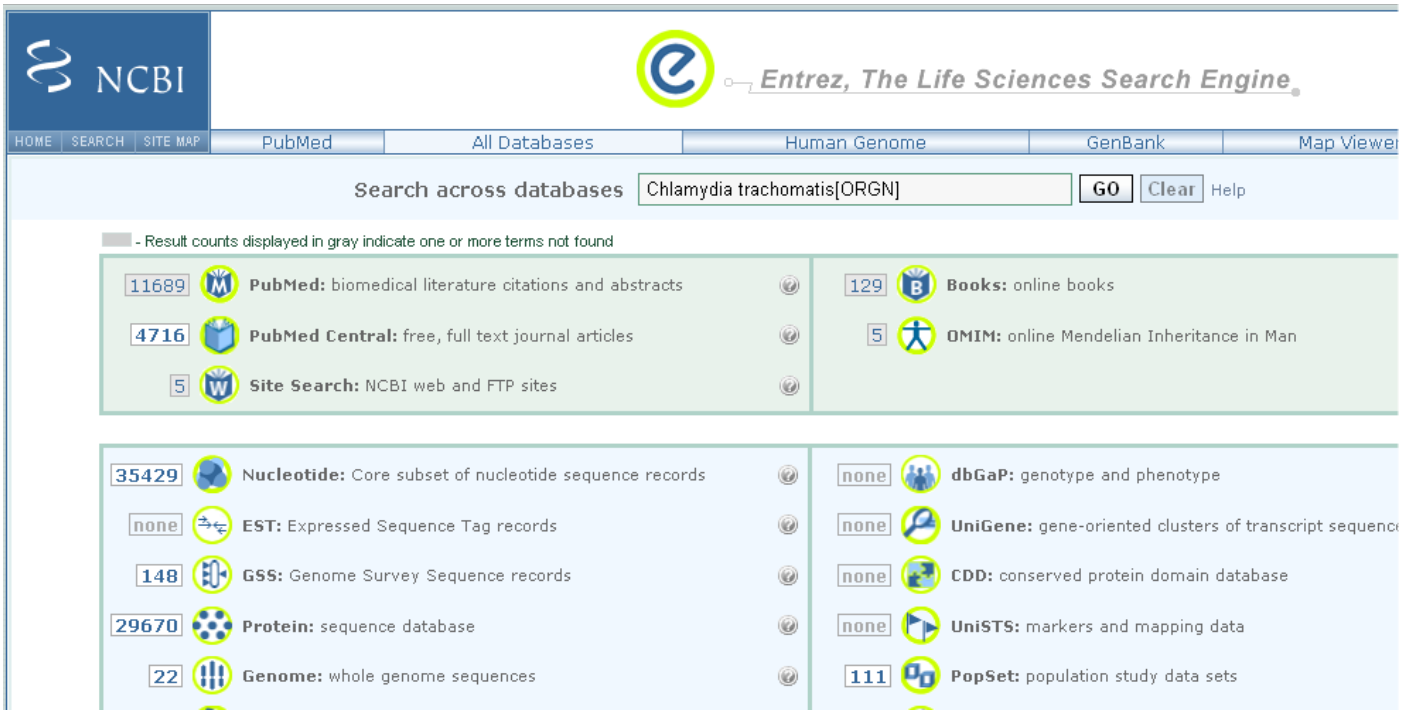
Typed in the search box	Searches for sequences:
NC_001477[AC]	With accession number NC_001477
Nature[JOUR] AND 460[VOL] AND 352[PAGE]	Published in *Nature* **460**:352-358
"Chlamydia trachomatis"[ORGN]	From the bacterium *Chlamydia trachomatis*
"Berriman M"[AU]	Published in a paper, or submitted to NCBI, by M. Berriman
flagellin OR fibrinogen	Which contain the word 'flagellin' or 'fibrinogen' in their NCBI record
"Mycobacterium leprae"[ORGN] AND dnaA	Which are from *M. leprae*, and contain "dnaA" in their NCBI record

"Homo sapiens"[ORGN] AND "colon cancer"	Which are from human, and contain "colon cancer" in their NCBI record
"Homo sapiens"[ORGN] AND malaria	Which are from human, and contain "malaria" in their NCBI record
"Homo sapiens"[ORGN] AND biomol_mrna[PROP]	Which are mRNA sequences from human
"Bacteria"[ORGN] AND srcdb_refseq[PROP]	Which are RefSeq sequences from Bacteria
"colon cancer" AND srcdb_refseq[PROP]	From RefSeq, which contain "colon cancer" in their NCBI record

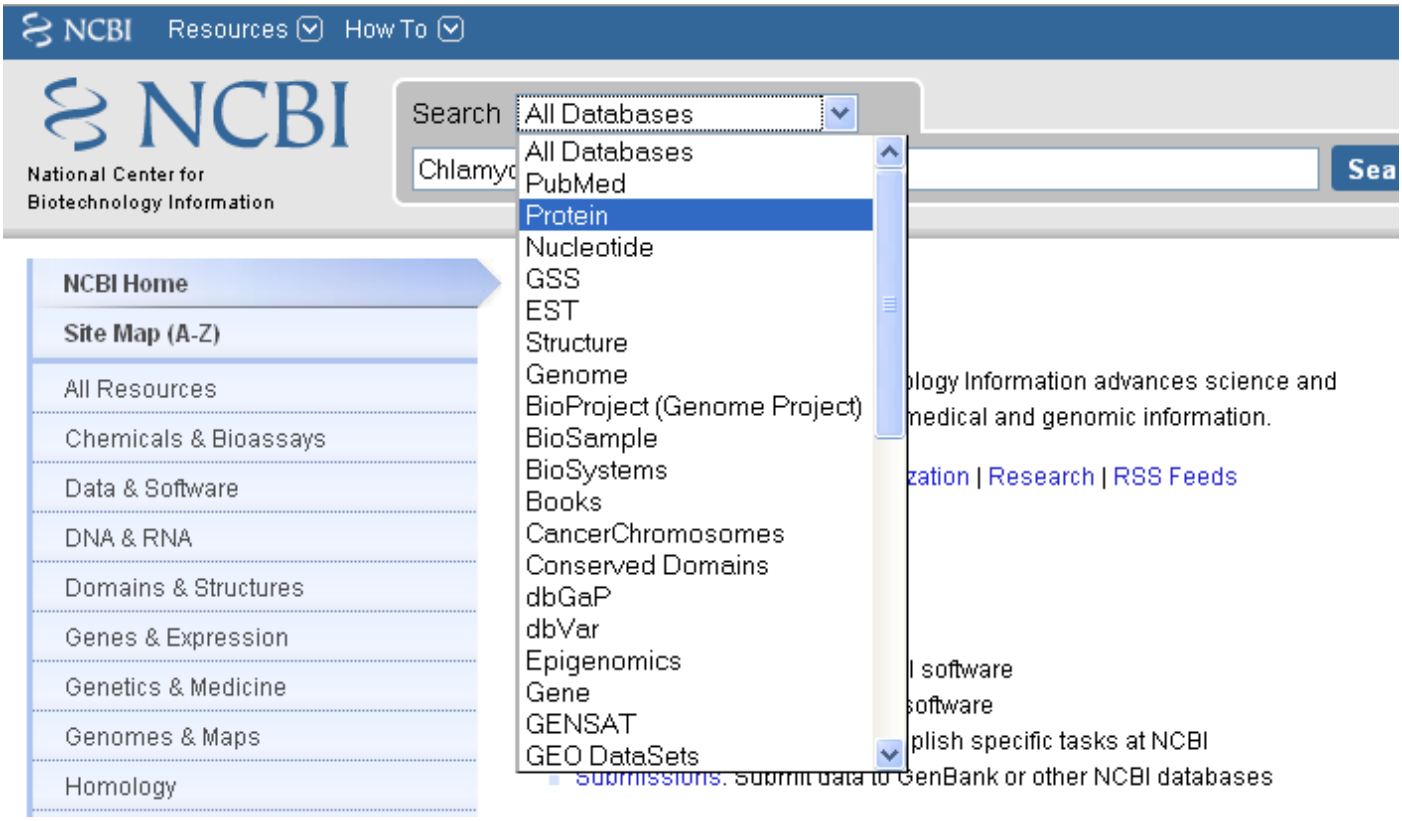
Si noti che se si cerca una frase come "colon cancer" o "Chlamydia trachomatis", è necessario inserire la frase tra virgolette (ricerca letterale). Questo perché se si digita la frase nella casella di ricerca senza usare le virgolette, la ricerca sarà per i record NCBI che contengono una delle due parole "colon" o "cancer" (o una delle due parole "Chlamydia" o "trachomatis"), non necessariamente entrambe le parole.

Come già detto, la banca dati NCBI contiene diverse sottobasi di dati, tra cui la banca dati dei Nucleotidi e la banca dati delle Proteine. Digitando sul sito web del NCBI una delle query di ricerca di cui sopra nella casella di ricerca, la pagina dei risultati dirà quanti record NCBI corrispondenti sono stati trovati in ciascuna delle sottobasi di dati NCBI.

Ad esempio, se si cerca "Chlamydia trachomatis[ORGN]", si otterranno corrispondenze con le proteine di *C. trachomatis* nel database delle proteine, corrispondenze con sequenze di DNA e RNA di *C. trachomatis* nel database dei nucleotidi, corrispondenze con sequenze di genomi interi per i ceppi di *C. trachomatis* nel database del genoma, e così via:

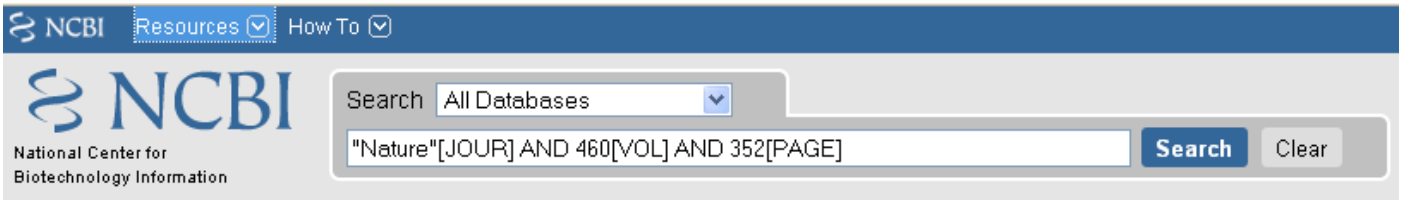


In alternativa, se si sa in anticipo che si vuole cercare un particolare sotto-database, per esempio, il database delle Proteine, quando si va sul sito web dell'NCBI è possibile selezionare quel sotto-database dall'elenco a discesa sopra la casella di ricerca:



Esempio: trovare le sequenze pubblicate in *Nature* 460:352-358

Ad esempio, se si desidera trovare le sequenze pubblicate in *Nature* 460:352-358, è possibile utilizzare i termini di ricerca "[JOUR]", "[VOL]" e "[PAGE]" digitando nella casella di ricerca: "Nature[JOUR] AND 460[VOL] AND 352[PAGE]", dove "[JOUR]" specifica il nome della rivista, "[VOL]" il volume della rivista in cui si trova l'articolo e "[PAGE]" il numero di pagina:



Questo dovrebbe far apparire una pagina di risultati con "50890" accanto alla parola "Nucleotide", e "1" accanto alla parola "Genoma", e "25701" accanto alla parola "Protein", che indica che vi sono 50.890 record di sequenza nel database dei Nucleotidi, che contiene sequenze di DNA e RNA, 1 risultato nel database del Genoma e 25.701 risultati nel database delle Proteine:

Se si clicca sulla parola "Nucleotide", si aprirà una pagina web con un elenco di link ai record di sequenze NCBI per quei 50.890 risultati, tutti connessi al verme *Schistosoma mansoni* (un verme parassita responsabile della schistosomiasi). Allo stesso modo, se si clicca sulla parola "Protein", si aprirà una pagina web con un elenco di link ai 25.701 record di sequenze NCBI che sono tutte proteine previste per *Schistosoma mansoni*. Se si clicca sulla parola "Genome", si otterranno i record NCBI per le sequenze del genoma di *Schistosoma mansoni*, che ha l'*accession* NS_00200. Si noti che l'*accession* inizia con "NS_", il che indica che si tratta di un *accession* RefSeq.

Pertanto, nel volume 460 *Nature*, pagina 352, è stata pubblicata la sequenza del genoma di *Schistosoma mansoni*, insieme a tutte le sequenze di DNA connesse che sono state sequenziate per il progetto del genoma, e tutte le proteine previste per le predizioni geniche effettuate nella sequenza del genoma. È possibile consultare il documento originale sul sito web di *Nature* all'indirizzo http://www.nature.com/nature/journal/v460/n7253/abs/nature08160.html.

3.5.2 Ricerca della sequenza genomica per una particolare specie

I genomi microbici sono generalmente più piccoli dei genomi eucarioti (*Escherichia coli* ha circa 5 milioni di coppie di basi nel suo genoma, mentre il genoma umano è di circa 3 miliardi di coppie di basi). Poiché sono notevolmente meno costosi da sequenziare, molti progetti di sequenziamento del genoma microbico sono stati completati.

Se non si conosce l'*accession* per una sequenza genomica (ad es. per *Mycobacterium leprae*, il batterio che causa la lebbra), come si fa a scoprirlo? Il modo più semplice per farlo è consultare il sito web NCBI del Genoma, che elenca tutti i genomi completamente sequenziati e fornisce i numeri di accesso per le sequenze di DNA corrispondenti.

Se ad es. non conosciamo il numero di accesso per il genoma del *Mycobacterium leprae*, possiamo trovarlo sul sito web NCBI del Genoma seguendo questi passi:

1. Aprire il sito web NCBI del genoma (http://www.ncbi.nlm.nih.gov/sites/entrez?db=Genome)
2. La homepage del sito web NCBI del Genoma fornisce link alle principali suddivisioni del database del Genoma, che includono Eukaryota, Prokaryota (Batteri e Archaea) e Virus. Cliccare su 'Prokaryota' (poiché il *Mycobacterium leprae* è un batterio): in questo modo verrà visualizzato un elenco di tutti i genomi batterici completamente sequenziati, con i numeri di accesso corrispondenti. Si noti che più di un genoma (di vari ceppi) può essere stato sequenziato per una particolare specie.
3. Usare 'Trova' nel menu 'Modifica' del browser web per cercare *Mycobacterium leprae* sulla pagina web. Si dovrebbero trovare i genomi dei diversi ceppi di *M. leprae* che sono stati sequenziati: uno di questi è *M. leprae* TN, che ha *accession* NC_002677.

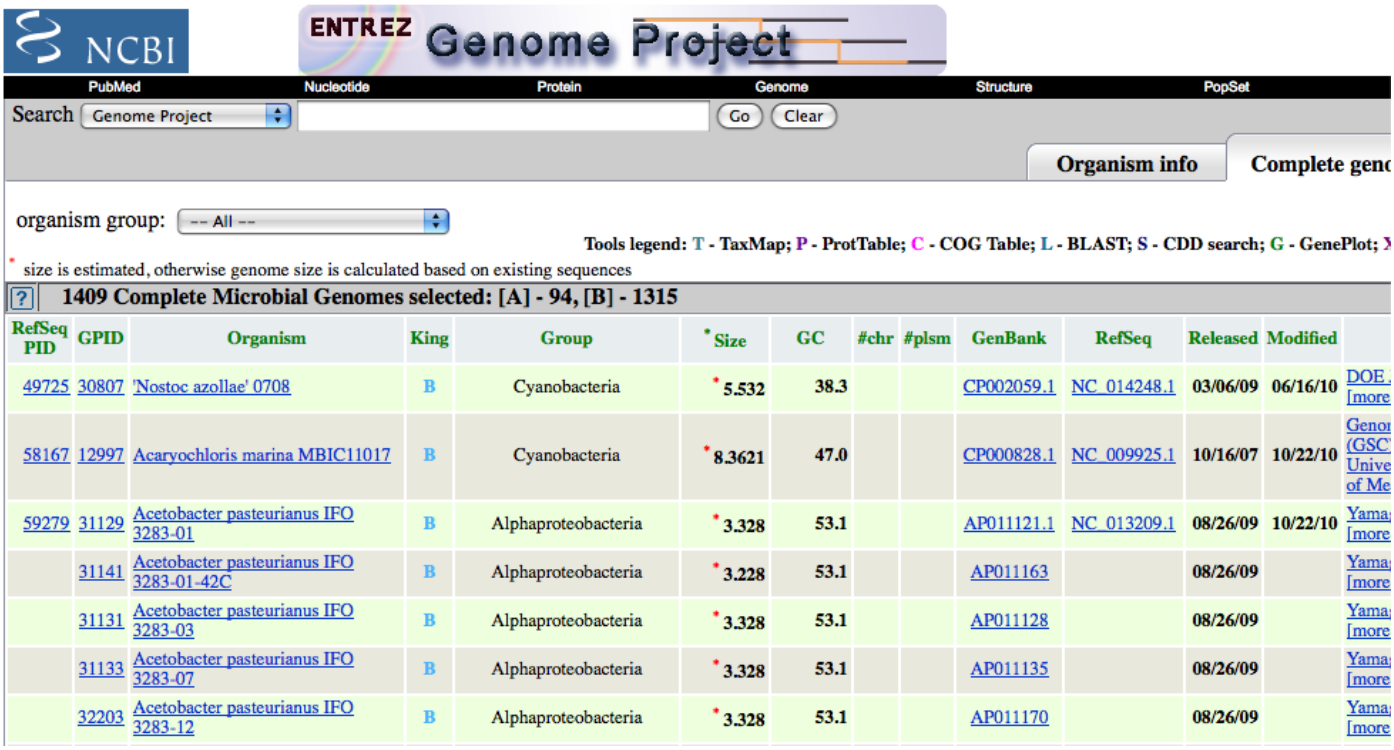
L'elenco dei genomi sequenziati sul sito web NCBI non è un elenco definitivo; alcuni genomi sequenziati potrebbero mancare. Se si vuole scoprire se un particolare genoma è stato sequenziato, ma non lo si trova nell'elenco del sito web NCBI, bisogna cercarlo seguendo questi passi:

1. Andare sul sito web NCBI (https://www.ncbi.nlm.nih.gov).
2. Selezionare "Genome" dall'elenco a discesa sopra la casella di ricerca.
3. Digitare nella casella di ricerca il nome della specie a cui si è interessati (ad es. "Mycobacterium leprae"[ORGN]).
4. Premere il pulsante <Search>.

Si noti che si può trovare la sequenza del genoma del *Mycobacterium leprae* anche cercando nel database NCBI dei Nucleotidi, poiché il database del genoma NCBI è solo un sottoinsieme del database NCBI dei Nucleotidi.

3.5.3 Quanti genomi sono stati sequenziati o si stanno tuttora sequenziando?

Sul sito web NCBI del Genoma (http://www.ncbi.nlm.nih.gov/sites/entrez?db=Genome), la prima pagina contiene link ad un elenco di tutti i genomi sequenziati nei gruppi *Eukaryota*, *Prokaryota* (Batteri e Archaea) e *Virus*. Se si clicca su uno di questi link (ad es. Prokaryota), nella parte superiore della pagina viene indicato il numero di genomi sequenziati in quel gruppo (ad es. numero di genomi procarioti sequenziati). Per esempio, in questa schermata (nel Gennaio 2011), vediamo che c'erano 1.409 genomi procarioti completi (94 archeali, 1.315 batteri):


NCBI ENTREZ Genome Project

PubMed Nucleotide Protein Genome Structure PopSet

Search Genome Project Go Clear

Organism info Complete genc

organism group: -- All --

Tools legend: T - TaxMap; P - ProtTable; C - COG Table; L - BLAST; S - CDD search; G - GenePlot;

* size is estimated, otherwise genome size is calculated based on existing sequences

1409 Complete Microbial Genomes selected: [A] - 94, [B] - 1315

RefSeq PID	GPID	Organism	King	Group	*Size	GC	#chr	#plsm	GenBank	RefSeq	Released	Modified	
49725	30807	'Nostoc azollae' 0708	B	Cyanobacteria	*5.532	38.3			CP002059.1	NC_014248.1	03/06/09	06/16/10	DOE [more
58167	12997	Acaryochloris marina MBIC11017	B	Cyanobacteria	*8.3621	47.0			CP000828.1	NC_009925.1	10/16/07	10/22/10	Genor (GSC) Unive of Me
59279	31129	Acetobacter pasteurianus IFO 3283-01	B	Alphaproteobacteria	*3.328	53.1			AP011121.1	NC_013209.1	08/26/09	10/22/10	Yama [more
	31141	Acetobacter pasteurianus IFO 3283-01-42C	B	Alphaproteobacteria	*3.228	53.1			AP011163		08/26/09		Yama [more
	31131	Acetobacter pasteurianus IFO 3283-03	B	Alphaproteobacteria	*3.328	53.1			AP011128		08/26/09		Yama [more
	31133	Acetobacter pasteurianus IFO 3283-07	B	Alphaproteobacteria	*3.328	53.1			AP011135		08/26/09		Yama [more
	32203	Acetobacter pasteurianus IFO 3283-12	B	Alphaproteobacteria	*3.328	53.1			AP011170		08/26/09		Yama [more

Un altro utile sito web che elenca i progetti di sequenziamento dei genomi è il Genomes OnLine Database (GOLD), che elenca i genomi che sono stati completamente sequenziati, o sono attualmente in fase di sequenziamento. Per trovare il numero di progetti di sequenziamento batterico completi o in corso, seguire questi passi:

1. Andare al sito web GOLD (http://genomesonline.org).
2. Cliccare sul pulsante 'Statistics' in alto nella pagina web. Nella pagina successiva, si otterrà il numero di progetti di sequenziamento del genoma batterico, archeologico ed eucariotico in corso.
3. Nella pagina principale cliccando sul pusante <Search> è possibile vedere l'elenco dei progetti di sequenziamento del genoma di interesse (ad es. batterico) in corso, con indicazioni sull'università o l'istituto in cui il genoma è stato sequenziato e le informazioni tassonomiche per l'organismo e i link ai dati della sequenza.

È possibile identificare i dati di sequenza del genoma nel database NCBI. Il database GOLD fornisce anche alcune informazioni sui progetti genomici in corso. Spesso, il database GOLD elenca alcuni progetti in corso che non sono ancora presenti nel database del genoma NCBI, perché i dati della sequenza non sono ancora stati inviati al database NCBI. Se si è interessati a scoprire quanti genomi sono stati sequenziati o sono attualmente in fase di sequenziamento per una particolare specie (ad es. *Mycobacterium leprae*), è una buona idea dare un'occhiata sia al database NCBI del genoma che a GOLD.

3.6 Interrogazione della suite di banche dati BioMart

Il pacchetto *biomaRt* (presente in BioConductor https://www.bioconductor.org) fornisce un'interfaccia ad una raccolta di database che implementa la suite BioMart (http://www.biomart.org), come *Ensembl* (geni e genomi), *Uniprot* (informazioni sulle proteine), *HGNC* (nomenclatura dei geni), *Gramene* (genomica funzionale delle piante) e *Wormbase* (informazioni su *Caenorhabditis elegans* e altri nematodi).

Installiamo prima Bioconductor e quindi il pacchetto *biomaRt* con i seguenti comandi:

```
> if (!requireNamespace("BiocManager", quietly = TRUE))
      install.packages("BiocManager")
> BiocManager::install(version = "3.10")
> BiocManager::install("biomaRt")
> library(biomaRt)
```

Selezioniamo il mart (database) appropriato per la ricerca di geni dell'*ensembl* "homo sapiens" mediante la funzione *useMart* come segue:

```
> mart <- useMart(biomart = "ensembl", dataset = "hsapiens_gene_ensembl")
```

Otteniamo quindi l'elenco degli identificatori ("hgnc_symbol") dei geni dal dataset scelto in precedenza, memorizzando i risultati nella variabile *res*:

```
> res <- getBM(attributes = c("hgnc_symbol"), mart = mart)
> str(res)
'data.frame':     37884 obs. of  1 variable:
 $ hgnc_symbol: chr  "MT-TF" "MT-RNR1" "MT-TV" "MT-RNR2" ...
```

dalla quale estraiamo un campione casuale di 15 geni:

```
> sam <- sample(res$hgnc_symbol,15)
> sam
 [1] "FUZ"          "RN7SL712P"    "CPLANE2"      "YEATS2"       "RNU6ATAC14P"
 [6] "ACCS"         "RNU7-26P"     "RBM3"         "HK3"          "OR5M8"
[11] "SIGLEC9"      "LINC00113"    "SHQ1P1"       "MED15P9"      "PRXL2AP1"
```

Il pacchetto *biomaRt* può essere utilizzato ad es. per recuperare le sequenze peptidiche di un gene:

```
> seq <- getSequence(id="AHSG", type="hgnc_symbol", seqType="peptide", mart=mart);
> seq
peptide
1
MKSLVLLLCLAQLWGCHSAPHGPGLIYRQPNCDDPETEEAALVAIDYINQNLPWGYKHTLNQIDEVKVWPQQPSGELFEIEIDTL
ETTCHVLDPTPVARCSVRQLKEHAVEGDCDFQLLKLDGKFSVVYAKCDSSPDSAEDVRKVCQDCPLLAPLNDTRVVHAAKAALAA
FNAQNNGSNFQLEEISRAQLVPLPPSTYVEFTVSGTDCVAKEATEAAKCNLLAEKQYGFCKATLSEKLGGAEVAVTCMVFQTQPV
SSQPQPEGANEAVPTPVVDPDAPPSPPLGAPGLPPAGSPPDSHVLLAAPPGHQLHRAHYDLRHTFMGVVSLGSPSGEVSHPRKTR
TVVQPSVGAAAGPVVPPCPGRIRHFKV*
…
```

Digitare `help(getSequence)` per una lista completa delle opzioni per gli argomenti `type` e `seqtype`.

Per recuperare una sequenza di nucleotidi specificando la posizione nel cromosoma, possiamo usare l'argomento `upstream` (o `downstream`) come segue:

```
seq2 <- getSequence(id="ENST00000520540",
      type="ensembl_transcript_id", seqType="gene_flank", upstream=30, mart=mart)
```

Le regioni "gene_flank" sono regioni di DNA adiacenti all'estremità 5′ (*upstream*) o 3′ (*downstream*) del gene che interagiscono con le proteine note come fattori di trascrizione che non vengono trascritte in RNA. La prima contiene il promotore e può contenere stimolatori o altri siti di legame delle proteine.

Nell'esempio, abbiamo estratto la sequenza *upstream* (cioè vicina alla parte iniziale) di 30 nucleotidi del gene ENST00000520540. Per vedere la sequenza, utilizziamo la funzione `show()` come segue:

```
> show(seq2)
                      gene_flank ensembl_transcript_id
1 AATGAAAAGAGGTCTGCCCGAGCGTGCGAC       ENST00000520540
```

4. Allineamento a coppie di sequenze

Come accennato nell'introduzione, mentre si analizzano i geni o le sequenze di proteine spesso è necessario confrontarle per conoscerne le similarità e le differenze: questo serve ad esempio per gli studi evolutivi e per capire la struttura e la funzione di una nuova sequenza confrontandola con quelle conosciute. Prima di fare affermazioni comparative su due sequenze, dobbiamo produrre un loro allineamento a coppie, che si riferisce al modo ottimale di organizzare due sequenze al fine di identificarne le regioni di somiglianza (allineamento ottimale). Alcune delle domande che emergono sono le seguenti:

- Come dobbiamo ottimizzare l'allineamento delle due sequenze?
- Come misurare le similarità e le differenze?
- Le sequenze di proteine vanno valutate diversamente dalle sequenze di DNA?

Ci sono diversi algoritmi e metriche che cercano di rispondere a queste domande e sono progettati per eseguire tali allineamenti, e la scelta di uno di essi dipende dalla domanda biologica a cui si deve rispondere. Esistono metodi di allineamento globale che mirano ad allineare ogni residuo nelle sequenze e vengono utilizzati quando le sequenze sono simili e di lunghezza comparabile (non è necessario che siano uguali): un esempio di tale metodo è l'algoritmo di Needleman-Wunsch. L'algoritmo di Smith-Waterman è invece una tecnica di allineamento locale che tenta di allineare regioni di elevata somiglianza nelle sequenze.

Ci sono molti altri metodi e algoritmi per l'allineamento delle sequenze, anche se discuterne in dettaglio va oltre lo scopo di questo testo. Qui introdurremo alcuni metodi basati su R per eseguire l'allineamento a coppie di due sequenze.

4.1 La banca dati *UniProt*

Nel capitolo precedente abbamo visto come recuperare le sequenze di DNA e proteine da NCBI, una banca dati chiave in bioinformatica perché contiene essenzialmente tutte le sequenze di DNA sinora sequenziate.

Come menzionato nel capitolo precedente, una sottosezione della banca dati NCBI chiamata *RefSeq* è costituita da dati di alta qualità di sequenze di DNA e proteine. Inoltre, le voci NCBI per le sequenze *RefSeq* sono state curate manualmente, il che significa che i biologi impiegati dalla NCBI hanno aggiunto ulteriori informazioni alle voci NCBI per quelle sequenze, come i dettagli dei documenti scientifici che descrivono le sequenze.

Un'altra banca dati estremamente importante curata manualmente è *UniProt* (https://www.uniprot.org), che si concentra sulle sequenze proteiche e contiene informazioni curate manualmente su tutte le sequenze proteiche conosciute. Mentre molte delle sequenze proteiche di *UniProt* sono presenti anche in *RefSeq*, la quantità e la qualità delle informazioni curate manualmente in *UniProt* è molto superiore a quella di *RefSeq*.

Per ogni proteina di *UniProt*, i curatori leggono tutti i documenti scientifici che possono trovare su quella proteina e aggiungono le informazioni di questi documenti alla voce *UniProt* della proteina. Per esempio, per una proteina umana, la voce *UniProt* per la proteina di solito include informazioni sulla funzione biologica della proteina, in quali tessuti umani si esprime, se interagisce con altre proteine umane e molto

altro ancora. Tutte queste informazioni sono state raccolte manualmente dai curatori di *UniProt* da documenti scientifici, e i documenti in cui si trovano le informazioni sono sempre elencati nella voce *UniProt* per la proteina.

Proprio come NCBI, anche *UniProt* assegna un *accession* ad ogni sequenza nel database. Anche se la stessa sequenza proteica può apparire sia nella banca dati NCBI che in *UniProt*, essa avrà *accession* diversi. Tuttavia, di solito c'è un link sulla voce NCBI della sequenza proteica alla voce *UniProt* e viceversa.

4.2 La pagina Web *UniProt* per le sequenze proteiche

Conoscendo l'*accession UniProt* per una proteina, per trovare la relativa voce bisogna prima andare sul sito Web di *UniProt*, https://www.uniprot.org. Nella parte superiore della homepage del sito c'è la casella di ricerca, nella quale occorre digitare l'*accession* della proteina che si sta cercando e poi cliccare sul pulsante <Search>.

Ad esempio, se si vuole trovare la sequenza della proteina *Mycobacterium leprae* che ha *accession UniProt* Q9CD83, basta digitare "Q9CD83" nella casella di ricerca e premere <Search>:

La voce *UniProt* per l'*accession* Q9CD83 apparirà quindi nel browser. L'immagine sottostante mostra la parte superiore della voce *UniProt* per l'*accession* Q9CD83, con le informazioni sulla proteina relativa:

Names and origin

Protein names	*Recommended name:* **Chorismate--pyruvate lyase** EC=4.1.3.- *Alternative name(s):* 4-HB synthase p-hydroxybenzoic acid synthase
Gene names	Ordered Locus Names:ML0133
Organism	**Mycobacterium leprae** [Complete proteome] [HAMAP]
Taxonomic identifier	1769 [NCBI]
Taxonomic lineage	Bacteria › Actinobacteria › Actinobacteridae › Actinomycetales › Corynebacterineae › Mycobacteriace

Protein attributes

Sequence length	210 AA.
Sequence status	Complete.
Protein existence	Inferred from homology.

General annotation (Comments)

Function	Removes the pyruvyl group from chorismate to provide 4-hydroxybenzoate (4HB). Involved in the synth (p-HBADs) and phenolic glycolipids (PGL) that play important roles in the pathogenesis of mycobacte
Catalytic activity	Chorismate = 4-hydroxybenzoate + pyruvate.
Sequence similarities	Belongs to the chorismate--pyruvate lyase type 2 family.

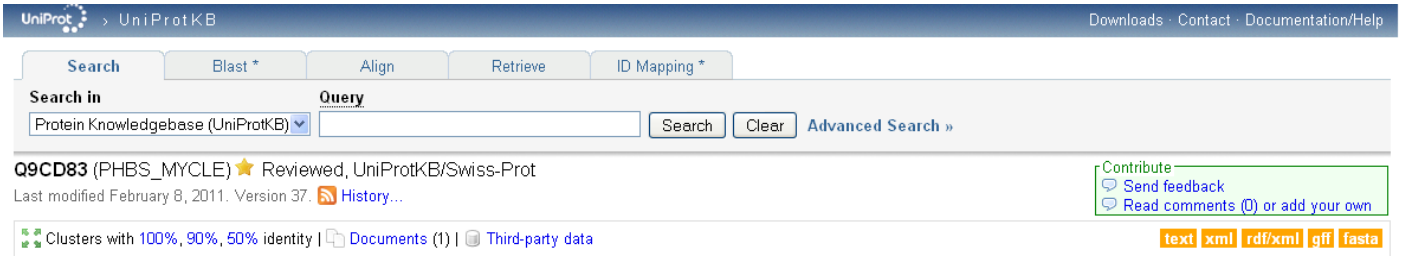
Alla voce "Organism" si può vedere che l'organismo è indicato come *Mycobacterium leprae*. La voce "Taxonomic lineage" (discendenza tassonomica) mostra "*Bacteria* > *Actinobacteria* > *Actinobacteridae* > *Actinomycetales* > *Corynebacterineae* > *Mycobacteriaceae* > *Mycobacterium leprae*". Questo ci dice che il *Mycobacterium* è una specie di batteri che appartiene ad un gruppo di batteri imparentati chiamati *Mycobacteriaceae*, a sua volta appartenente ai gruppi più ampi chiamati *Corynebacterineae*, *Actinomycetales*, *Actinobacteridae* e *Actinobacteria*.

Alla voce "Sequence length" (lunghezza della sequenza) vediamo che la sequenza è lunga 210 aminoacidi. Più in basso, la voce "Function" (funzione) dice che questa proteina "Rimuove il gruppo piruvilico dal corismato per fornire 4-idrossibenzoato (4HB)". Questo ci dice che la proteina è un enzima (una proteina che aumenta il tasso di una specifica reazione biochimica), e qual è la particolare reazione biochimica in cui questo enzima è coinvolto.

Più in basso nella pagina vi sono molte più informazioni, così come molti link a pagine web in altri database biologici, come NCBI. L'enorme quantità di informazioni sulle proteine in *UniProt* significa che se si vuole scoprire una particolare proteina, la pagina *UniProt* è un ottimo punto di partenza.

4.3 Recuperare sequenze proteiche dal sito Web *UniProt*

Per recuperare un file in formato FASTA contenente la sequenza di una particolare proteina, è necessario guardare in alto a destra nella homepage del sito Web *UniProt* dopo aver effettuato la ricerca. Vi è un piccolo pulsante con l'etichetta "FASTA", su cui bisogna cliccare:



La sequenza in formato FASTA apparirà ora nel browser web. Per salvarla come file, andare nel menu "File" del browser, scegliere "Salva pagina con nome" e salvare il file.

Ad esempio, è possibile recuperare le sequenze proteiche per le proteine *Mycobacterium leprae* (che ha l'*accession UniProt* Q9CD83) e *Mycobacterium ulcerans* (*accession UniProt* A0PQ23), e salvarle come file in formato FASTA (ad es. "Q9CD83.fasta" e "A0PQ23.fasta").

Si noti che il *Mycobacterium leprae* è il batterio che causa la lebbra, mentre il *Mycobacterium ulcerans* è un batterio correlato che causa l'ulcera di Buruli, entrambi classificati dall'OMS come malattie tropicali trascurate. Le proteine *M. leprae* e *M. ulcerans* sono un esempio di una coppia di proteine omologhe (affini) in due specie di batteri imparentate.

Una volta scaricate le sequenze proteiche per gli *accession* UniProt Q9CD83 e A0PQ23 e salvate come file in formato FASTA (ad es. "Q9CD83.fasta" e "A0PQ23.fasta"), le possiamo leggerle in R usando la funzione `read.fasta()` del pacchetto *SeqinR*.

Per esempio, i seguenti comandi leggono i file in formato FASTA Q9CD83.fasta e A0PQ23.fasta in R, e memorizzano le due sequenze proteiche in due vettori *lepraeseq* e *ulceransseq*:

```
> library("seqinr")
> leprae <- read.fasta(file = "Q9CD83.fasta")
> ulcerans <- read.fasta(file = "A0PQ23.fasta")
> lepraeseq <- leprae[[1]]
> ulceransseq <- ulcerans[[1]]
> lepraeseq # Display the contents of the vector "lepraeseq"
  [1] "m" "t" "n" "r" "t" "l" "s" "r" "e" "e" "i" "r" "k" "l" "d" "r" "d" "l"
 [19] "r" "i" "l" "v" "a" "t" "n" "g" "t" "l" "t" "r" "v" "l" "n" "v" "v" "a"
 [37] "n" "e" "e" "i" "v" "v" "d" "i" "i" "n" "q" "q" "l" "l" "d" "v" "a" "p"
 [55] "k" "i" "p" "e" "l" "e" "n" "l" "k" "i" "g" "r" "i" "l" "q" "r" "d" "i"
 [73] "l" "l" "k" "g" "q" "k" "s" "g" "i" "l" "f" "v" "a" "a" "e" "s" "l" "i"
 [91] "v" "i" "d" "l" "l" "p" "t" "a" "i" "t" "t" "y" "l" "t" "k" "t" "h" "h"
[109] "p" "i" "g" "e" "i" "m" "a" "a" "s" "r" "i" "e" "t" "y" "k" "e" "d" "a"
[127] "q" "v" "w" "i" "g" "d" "l" "p" "c" "w" "l" "a" "d" "y" "g" "y" "w" "d"
[145] "l" "p" "k" "r" "a" "v" "g" "r" "r" "y" "r" "i" "i" "a" "g" "g" "q" "p"
[163] "v" "i" "i" "t" "t" "e" "y" "f" "l" "r" "s" "v" "f" "q" "d" "t" "p" "r"
[181] "e" "e" "l" "d" "r" "c" "q" "y" "s" "n" "d" "i" "d" "t" "r" "s" "g" "d"
[199] "r" "f" "v" "l" "h" "g" "r" "v" "f" "k" "n" "l"
```

4.4 Recuperare sequenze proteiche in R

Vi sono molti strumenti software promettenti per l'analisi e la visualizzazione della struttura delle proteine; anche R offre pacchetti che servono allo scopo. Nel seguito illustreremo come recuperare sequenze proteiche da UniProt mediante i pacchetti *rentrez*, *UniProt.ws* e *SeqinR*.

4.4.1 Recuperare sequenze proteiche in R con il pacchetto rentrez

Un metodo alternativo per recuperare una sequenza proteica in R è quello di utilizzare il pacchetto *rentrez* per interrogare il sotto-database NCBI "swissprot", che contiene sequenze proteiche di *UniProt*.

NCBI rende accessibili i dati attraverso l'API (Application Programming Interaface) chiamata "Entrez Utilities" (*Eutils* in breve), disponibile in R mediante il pacchetto *rentrez* che mette a disposizione delle funzioni che consentono all'utente di raccogliere dati da vari database NCBI.

In primo luogo, è possibile utilizzare la funzione `entrez_dbs()` per trovare l'elenco dei database disponibili:

```
> install.packages("rentrez") # if not yet installed
> library(rentrez) # use package 'rentrez'
> entrez_dbs()
 [1] "pubmed"          "protein"         "nuccore"         "ipg"             "nucleotide"
 [6] "structure"       "sparcle"         "genome"          "annotinfo"       "assembly"
[11] "bioproject"      "biosample"       "blastdbinfo"     "books"           "cdd"
[16] "clinvar"         "gap"             "gapplus"         "grasp"           "dbvar"
[21] "gene"            "gds"             "geoprofiles"     "homologene"      "medgen"
[26] "mesh"            "ncbisearch"      "nlmcatalog"      "omim"            "orgtrack"
[31] "pmc"             "popset"          "probe"           "proteinclusters" "pcassay"
[36] "biosystems"      "pccompound"      "pcsubstance"     "seqannot"        "snp"
[41] "sra"             "taxonomy"        "biocollections"  "gtr"
```

Nel nostro caso utilizzeremo il database "protein". Altre funzioni utili del pacchetto sono:

- `entrez_db_summary()`: recupera informazioni sintetiche su una banca dati NCBI
- `entrez_db_searchable()`: elenca i campi di ricerca disponibili per una determinata banca dati
- `entrez_search()`: esegue ricerche in un determinato database NCBI
- `entrez_summary()`: ottiene riepiloghi di oggetti in dataset NCBI mediante un ID univoco
- `entrez_fetch()`: recupera i dati dalle banche dati NCBI

Utilizzando tali funzioni possiamo recuperare le sequenze proteiche per gli *accession* UniProt Q9CD83 e A0PQ23. Vediamo innanzitutto le informazioni generali sul database "protein":

```
> entrez_db_summary("protein") # view sinthetic informations about database "protein"
DbName: protein
MenuName: Protein
Description: Protein sequence record
DbBuild: Build191208-1236m.1
Count: 809583046
LastUpdate: 2019/12/10 08:29
```

Ritroviamo quindi le parole chiave di ricerca ammesse sul database (ci interessa l'*accession*):

```
> entrez_db_searchable("protein")
Searchable fields for database 'protein'
  ALL  All terms from all searchable fields
       …
  ACCN Accession number of sequence
       …
```

Cerchiamo l'ID univoco corrispondente agli *accession* UniProt Q9CD83 e ABL04442 (alias di A0PQ23 in Entrez) utilizzando la parola chiave ACCN:

```
> r_search <- entrez_search(db="protein", term="Q9CD83[ACCN]"); r_search$id
[1] "81537264"
> entrez_summary(db="protein", id=81537264)
…
> r_search <- entrez_search(db="protein", term="ABL04442[ACCN]"); r_search$id
[1] "118569691"
> entrez_summary(db="protein", id= 118569691)
…
```

Recuperiamo infine le sequenze proteiche in formato FASTA e registriamole su file:

```
> tmp <- entrez_fetch(db="protein", id="81537264", rettype="fasta")
> write(tmp, file="Q9CD83.fasta")
> leprae <- read.fasta("Q9CD83.fasta") # needed to rebuild the fasta structure
> lepraeseq <- leprae[[1]]
> tmp <- entrez_fetch(db="protein", id="118569691", rettype="fasta")
> write(tmp, file="A0PQ23.fasta")
> ulcerans <- read.fasta("A0PQ23.fasta") # needed to rebuild the fasta structure
> ulceransseq <- ulcerans[[1]]
```

4.4.2 Recuperare sequenze proteiche in R con il pacchetto UniProt.ws

L'alternativa qui presentata è il pacchetto *UniProt.ws* (della suite Bioconductor), un'interfaccia R per i servizi web di UniProt. I vari tipi di dati che possono essere estratti da UniProt includono le sequenze peptidiche, l'annotazione GO, le interazioni, le famiglie di proteine e così via. Caricando il pacchetto in una sessione R si crea un oggetto UniProt.ws.

Il database contiene dati sulle proteine di diverse specie, che possono essere elencate (insieme ai loro ID) con la funzione `availableUniprotSpecies()`; quindi è possibile selezionare l'ID tassonomico desiderato. La funzione più importante del pacchetto è la funzione `select()` che estrae i dati da UniProt. La funzione necessita di tre argomenti: `keys` (chiavi), `columns` (colonne) e `keytype` (tipo di chiave), insieme all'oggetto UniProt.ws. Le colonne indicano quali tipi di dati devono essere restituiti nel data frame di output; le chiavi servono a recuperare i record desiderati dal database. L'argomento `keytype` può essere usato per restituire chiavi specifiche: ad esempio, `keytype = "UNIPROTKB"` definisce il tipo di chiave di accesso di UniProt, mentre `ENTREZ_GENE` corrisponde al tipo di accesso Entrez ID.

Per utilizzare il pacchetto lo dobbiamo installare e caricare come segue:

```
> BiocManager::install("UniProt.ws")
> library(UniProt.ws)
```

Una volta caricata la libreria nella sessione, possiamo controllare le specie disponibili in UniProt (in questo caso, con il pattern "sapiens") come segue:

```
> availableUniprotSpecies(pattern="sapiens")
Downloading: 690 kB
  taxon ID                                                                Species name
1   742918 Human associated cyclovirus 1 (isolate Homo sapiens/Pakistan/PK5510/2007)
2    63221                                                 Homo sapiens neanderthalensis
3     9606                                                                Homo sapiens
```

Impostiamo l'ID tassonomico per la specie umana e controlliamo le informazioni disponibili nel database selezionato con i seguenti comandi:

```
> up <- UniProt.ws(taxId=9606)
> head(keytypes(up))
[1] "AARHUS/GHENT-2DPAGE" "AGD"                 "ALLERGOME"           "ARACHNOSERVER"
[5] "BIOCYC"              "CGD"
> head(columns(up))
[1] "3D"                  "AARHUS/GHENT-2DPAGE" "AGD"                 "ALLERGOME"
[5] "ARACHNOSERVER"       "BIOCYC"
```

Ora selezioniamo le informazioni "PDB", "UNIPROTKB", "SEQUENCE" per l'ID Entrez O95544 (keytype "UNIPROTKB") e recuperiamo la sequenza desiderata con il comando `select()`:

```
> myProtein <- select(up, "O95544", c("PDB", "UNIPROTKB", "SEQUENCE"), "UNIPROTKB")
Getting mapping data for O95544 ... and PDB_ID
Getting extra data for O95544
'select()' returned 1:1 mapping between keys and columns
```

Visualizziamo infine la sequenza:

```
> myProtein$SEQUENCE
[1]
"MEMEQEKMTMNKELSPDAAAYCCSACHGDETWSYNHPIRGRAKSRSLSASPALGSTKEFRRTRSLHGPCPVTTFGPKACVLQNPQTIMHIQDP
ASQRLTWNKSPKSVLVIKKMRDASLLQPFKELCTHLMEENMIVYVEKKVLEDPAIASDESFGAVKKKFCTFREDYDDISNQIDFIICLGGDGTL
LYASSLFQGSVPPVMAFHLGSLGFLTPFSFENFQSQVTQVIEGNAAVVLRSRLKVRVVKELRGKKTAVHNGLGENGSQAAGLDMDVGKQAMQYQ
VLNEVVIDRGPSSYLSNVDVYLDGHLITTVQGDGVIVSTPTGSTAYAAAAGASMIHPNVPAIMITPICPHSLSFRPIVVPAGVELKIMLSPEAR
NTAWVSFDGRKRQEIRHGDSISITTSCYPLPSICVRDPVSDWFESLAQCLHWNVRKKQAHFEEEEEEEEEG"
```

4.4.3 Recuperare sequenze proteiche in R con il pacchetto SeqinR

Per recuperare una sequenza proteica è anche possibile utilizzare il pacchetto *SeqinR* per interrogare il sotto-database ACNUC "swissprot", che contiene sequenze proteiche di UniProt. Ad esempio, per recuperare le sequenze proteiche per gli *accession* UniProt Q9CD83 e A0PQ23, digitiamo:

```
> library(seqinr)
> choosebank("swissprot")
> leprae <- query("leprae", "AC=Q9CD83")
> lepraeseq <- getSequence(leprae$req[[1]])
> ulcerans <- query("ulcerans", "AC=A0PQ23")
> ulceransseq <- getSequence(ulcerans$req[[1]])
> closebank()
> lepraeseq # Display the contents of "lepraeseq"
  [1] "M" "T" "N" "R" "T" "L" "S" "R" "E" "E" "I" "R" "K" "L" "D" "R" "D" "L" "R" "I" "L" "V"
 [23] "A" "T" "N" "G" "T" "L" "T" "R" "V" "L" "N" "V" "V" "A" "N" "E" "E" "I" "V" "V" "D" "I"
 [45] "I" "N" "Q" "Q" "L" "L" "D" "V" "A" "P" "K" "I" "P" "E" "L" "E" "N" "L" "K" "I" "G" "R"
 [67] "I" "L" "Q" "R" "D" "I" "L" "L" "K" "G" "Q" "K" "S" "G" "I" "L" "F" "V" "A" "A" "E" "S"
 [89] "L" "I" "V" "I" "D" "L" "L" "P" "T" "A" "I" "T" "T" "Y" "L" "T" "K" "T" "H" "H" "P" "I"
[111] "G" "E" "I" "M" "A" "A" "S" "R" "I" "E" "T" "Y" "K" "E" "D" "A" "Q" "V" "W" "I" "G" "D"
[133] "L" "P" "C" "W" "L" "A" "D" "Y" "G" "Y" "W" "D" "L" "P" "K" "R" "A" "V" "G" "R" "R" "Y"
[155] "R" "I" "I" "A" "G" "G" "Q" "P" "V" "I" "I" "T" "T" "E" "Y" "F" "L" "R" "S" "V" "F" "Q"
[177] "D" "T" "P" "R" "E" "E" "L" "D" "R" "C" "Q" "Y" "S" "N" "D" "I" "D" "T" "R" "S" "G" "D"
[199] "R" "F" "V" "L" "H" "G" "R" "V" "F" "K" "N" "L"
```

4.5 Confronto di due sequenze mediante un grafico a dispersione (*dotplot*)

Come primo passo per confrontare due sequenze di proteine, RNA o DNA, è una buona idea fare un grafico a dispersione (*dotplot*), un metodo grafico che permette di confrontare due sequenze di proteine o di DNA e di identificare le regioni di stretta somiglianza tra loro. Un *dotplot* è essenzialmente una matrice bidimensionale, con le sequenze delle proteine che vengono confrontate disposte lungo gli assi verticale e orizzontale.

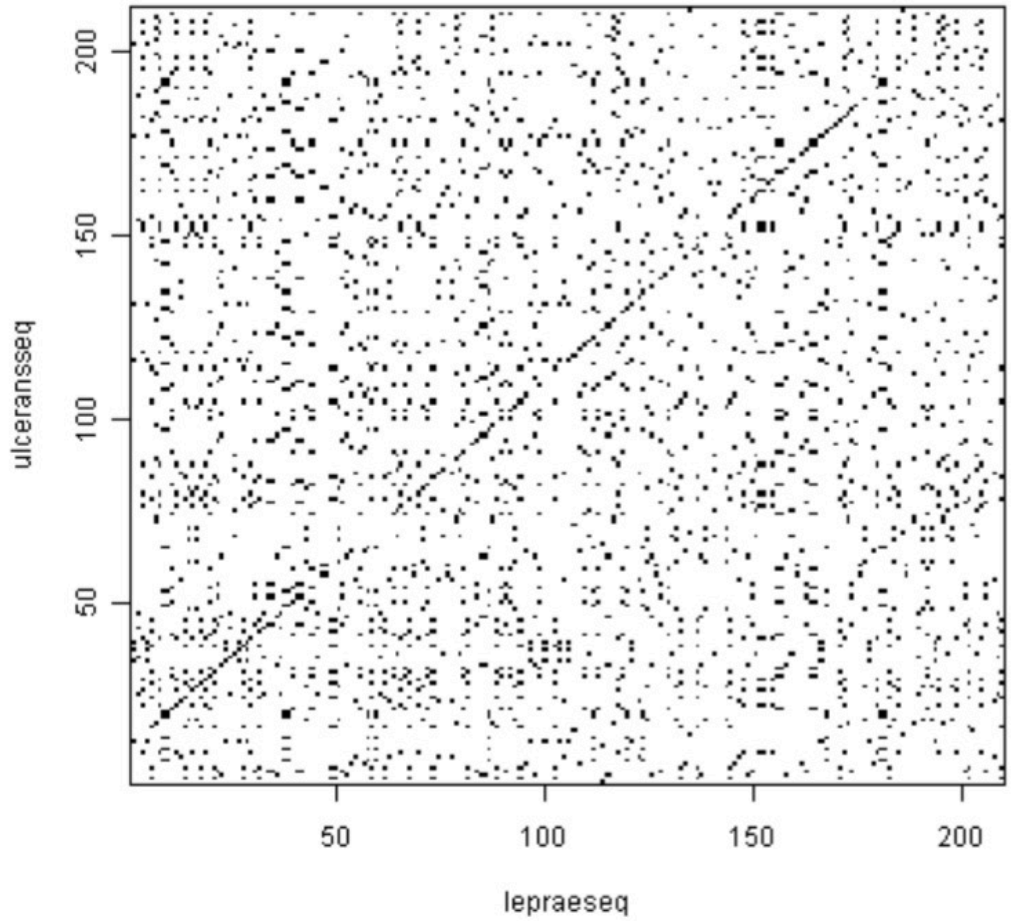
Al fine di realizzare un semplice *dotplot* per rappresentare la similarità tra due sequenze, le singole celle della matrice vengono annerite se i residui sono identici, in modo che i segmenti di sequenza corrispondenti appaiono come tratti di linee diagonali attraverso la matrice. Le proteine identiche avranno una linea esattamente sulla diagonale principale del grafico, che si estende attraverso l'intera matrice di punti.

Per le proteine che non sono identiche, ma condividono regioni di somiglianza, il grafico avrà linee più corte che possono trovarsi sulla diagonale principale, o fuori dalla diagonale principale della matrice. In sostanza, un *dotplot* rivelerà se ci sono regioni che sono chiaramente molto simili in due sequenze di proteine (o DNA).

Possiamo creare un grafico *dotplot* per due sequenze usando la funzione `dotPlot()` del pacchetto *SeqinR*.

Ad esempio, se vogliamo creare un *dotplot* delle sequenze per le proteine del *Mycobacterium leprae* e *Mycobacterium ulcerans*, scriveremo:

```
> dotPlot(lepraeseq, ulceransseq)
```



Nel grafico di cui sopra, la sequenza *M. leprae* viene tracciata lungo l'asse *x* (orizzontale), e la sequenza *M. ulcerans* lungo l'asse *y* (verticale). Il grafico mostra un quadratino nero nei punti in cui c'è un identico aminoacido nelle due sequenze.

Per esempio, se l'aminoacido 53 nella sequenza *M. leprae* è lo stesso aminoacido (ad es. "A") dell'aminoacido 63 nella sequenza *M. ulcerans*, allora il grafico mostrerà un punto nella posizione corrispondente a *x*=53 e *y*=63.

In questo caso si possono vedere molti punti lungo una linea diagonale, il che indica che le due sequenze proteiche contengono molti aminoacidi identici nelle stesse posizioni (o molto simili) lungo la loro lunghezza. Questo è quello che ci si aspetterebbe, perché sappiamo che queste due proteine sono *omologhe* (proteine correlate).

Un utile tool online per tracciare *dotplot* è Dotlet JS, accessibile all'indirizzo: https://dotlet.vital-it.ch

4.6 Allineamento globale a coppie di sequenze di DNA mediante l'algoritmo Needleman-Wunsch

Se si studia una particolare coppia di geni o proteine, una domanda importante è in che misura le due sequenze sono simili. Per quantificare la similarità, è necessario allineare le due sequenze, per calcolare un punteggio di somiglianza in base all'allineamento.

Ci sono due tipi di allineamento in generale. Un *allineamento globale* è un allineamento dell'intera lunghezza di due sequenze, per esempio, di due sequenze di proteine o di due sequenze di DNA. Un *allineamento locale* è un allineamento di parte di una sequenza con parte di un'altra sequenza.

Il primo passo nel calcolo di un allineamento (globale o locale) è quello di decidere un sistema di punteggio (*score*). Per esempio, possiamo decidere di dare un punteggio di +2 a una corrispondenza e una penalità di -1 a una mancata corrispondenza, e una penalità di -2 a un intervallo (*gap*). Così, per l'allineamento:

```
G A A T T C
G A T T - A
```

calcoleremmo un punteggio di 2+2-1+2-2-1 = 2. Allo stesso modo, il punteggio per il seguente allineamento è 2+2-2+2+2-1 = 5:

```
G A A T T C
G A - T T A
```

Il sistema di punteggio sopra descritto può essere rappresentato da una *score matrix* (matrice di punteggio, nota anche come *matrice di sostituzione*). La matrice di punteggio ha una riga e una colonna per ogni possibile lettera del nostro alfabeto di lettere (ad es. 4 righe e 4 colonne per le sequenze di DNA). L'elemento (i, j) della matrice ha un valore +2 in caso di corrispondenza e -1 in caso di disallineamento.

Possiamo realizzare una *score matrix* in R utilizzando la funzione `nucleotideSubstitutionMatrix()` del pacchetto *Biostrings*, che fa parte di una serie di pacchetti R per l'analisi bioinformatica nota come "Bioconductor" (https://www.bioconductor.org). Per utilizzare il pacchetto *Biostrings*, è necessario prima installare il pacchetto nel seguente modo:

```
> if (!requireNamespace("BiocManager", quietly = TRUE))
    install.packages("BiocManager") # install Bioconductor Manager
> BiocManager::install(version = "3.10") # install Bioconductor core packages
> BiocManager::install("Biostrings")
```

Gli argomenti (input) per la funzione `nucleotideSubstitutionMatrix()` sono il punteggio che vogliamo assegnare ad una corrispondenza (*match*) e il punteggio che vogliamo assegnare ad un disallineamento (*mismatch*). Possiamo anche specificare che vogliamo usare solo le quattro lettere che rappresentano i quattro nucleotidi (cioè A, C, G, T) impostando '`baseOnly=TRUE`', o se vogliamo usare anche le lettere che rappresentano casi ambigui in cui non siamo sicuri di quale sia il nucleotide (es. 'N' = A/C/G/T).

Per creare una matrice che assegna un punteggio +2 ad un match e -1 ad un mismatch, e memorizzarla nella variabile *sigma*, digitiamo:

```
> library(Biostrings)
> sigma <- nucleotideSubstitutionMatrix(match = 2, mismatch = -1, baseOnly = TRUE)
> sigma # Print out the matrix
   A  C  G  T
A  2 -1 -1 -1
C -1  2 -1 -1
G -1 -1  2 -1
T -1 -1 -1  2
```

Invece di assegnare la stessa penalità ad ogni *gap*, è comune invece assegnare una penalità di apertura del *gap* alla prima posizione di un intervallo, e una penalità di estensione del *gap* più piccola ad ogni posizione successiva nello stesso intervallo.

La ragione per ciò è che è probabile che posizioni di *gap* adiacenti siano state create dallo stesso evento di inserimento o cancellazione, piuttosto che da diversi eventi indipendenti di inserimento o cancellazione. Pertanto, non vogliamo penalizzare un gap di 3 lettere tanto quanto penalizzeremmo tre separati gap di 1 lettera, in quanto il gap di 3 lettere potrebbe essere sorto a causa di un solo evento di inserimento o cancellazione, mentre i 3 separati gap di 1 lettera probabilmente sono sorti a causa di tre eventi indipendenti di inserimento o cancellazione.

Ad esempio, se vogliamo calcolare il punteggio per un allineamento globale delle due brevi sequenze di DNA "GAATTC" e "GATTA", possiamo usare l'algoritmo Needleman-Wunsch per calcolare l'allineamento con il punteggio più alto usando una particolare funzione di punteggio. La funzione `pairwiseAlignment()` nel pacchetto *Biostrings* trova il punteggio per l'allineamento globale ottimale tra due sequenze utilizzando l'algoritmo Needleman-Wunsch, dato un particolare sistema di punteggio.

Come argomenti (input), la funzione `pairwiseAlignment()` prende le due sequenze che si vogliono allineare, la matrice di punteggio, la penalità di apertura del gap e la penalità di estensione del gap. Si può anche dire alla funzione che si vuole avere solo il punteggio dell'allineamento globale ottimale impostando "`scoreOnly = TRUE`", oppure che si vuole avere sia l'allineamento globale ottimale che il suo punteggio impostando "`scoreOnly = FALSE`".

Per esempio, per trovare il punteggio per l'allineamento globale ottimale tra le sequenze "GAATTC" e "GATTA", digitiamo:

```
> s1 <- "GAATTC"
> s2 <- "GATTA"
> globalAligns1s2 <- pairwiseAlignment(s1, s2, substitutionMatrix = sigma,␣
↪gapOpening = -2,
gapExtension = -8, scoreOnly = FALSE)
> globalAligns1s2 # Print out the optimal alignment and its score
Global Pairwise Alignment (1 of 1)
pattern: [1] GAATTC
subject: [1] GA-TTA
score: -3
```

I comandi di cui sopra producono l'allineamento globale ottimale per le due sequenze e il relativo punteggio.

Si noti che abbiamo impostato "`gapOpening`" a -2 e "`gapExtension`" a -8, il che significa che alla prima posizione di uno spazio viene assegnato un punteggio di -8-2 = -10, e ad ogni posizione successiva in un intervallo viene assegnato un punteggio di -8. Qui l'allineamento contiene quattro corrispondenze, un mismatch, e un intervallo di lunghezza 1, quindi il suo punteggio è [4 · 2] + [1 · (-1)] + [1 · (-10)] = -3.

4.7 Allineamento globale a coppie di sequenze proteiche mediante l'algoritmo Needleman-Wunsch

Oltre agli allineamenti del DNA, è anche possibile effettuare allineamenti di sequenze proteiche. In questo caso è necessario utilizzare matrici di *score* per gli aminoacidi piuttosto che per i nucleotidi, che sono più grandi e dall'aspetto più complicato essendovi una diversa probabilità per ogni coppia di aminoacidi. Sono state proposte diverse matrici per le sostituzioni degli aminoacidi nelle sequenze proteiche.

Ci sono diverse matrici ben note disponibili in R, come la serie di matrici BLOSUM (Block Substitution Matrix), una famiglia di matrici i cui membri sono calcolati sulla base delle sostituzioni che si incontrano in un insieme di proteine conosciute. Un'altra famiglia popolare di matrici di sostituzione è la PAM, che qui non tratteremo.

Esistono matrici BLOSUM differenti, identificate con numeri diversi: quelle con numeri alti sono progettate per confrontare sequenze strettamente correlate, mentre quelle con numeri bassi sono progettate per confrontare sequenze poco correlate. Ad esempio, BLOSUM62 è usata per allineamenti poco divergenti (allineamenti di sequenze che differiscono poco), e BLOSUM30 è usata per allineamenti più divergenti (allineamenti di sequenze che differiscono molto).

Il pacchetto *Biostrings* contiene diverse matrici di sostituzione. Si può usare la funzione "`data()`" in R per caricare il dataset di matrici BLOSUM fornito con il pacchetto; ad esempio, per caricare la matrice BLOSUM50 si digita (sono visualizzate solo le prime 10 righe):

```
> data(BLOSUM50)
> BLOSUM50 # Print out the data
   A  R  N  D  C  Q  E  G  H  I  L  K  M  F  P  S  T  W  Y  V  B  Z  X  *
A  5 -2 -1 -2 -1 -1 -1  0 -2 -1 -2 -1 -1 -3 -1  1  0 -3 -2  0 -2 -1 -1 -5
R -2  7 -1 -2 -4  1  0 -3  0 -4 -3  3 -2 -3 -3 -1 -1 -3 -1 -3 -1  0 -1 -5
N -1 -1  7  2 -2  0  0  0  1 -3 -4  0 -2 -4 -2  1  0 -4 -2 -3  4  0 -1 -5
D -2 -2  2  8 -4  0  2 -1 -1 -4 -4 -1 -4 -5 -1  0 -1 -5 -3 -4  5  1 -1 -5
C -1 -4 -2 -4 13 -3 -3 -3 -3 -2 -2 -3 -2 -2 -4 -1 -1 -5 -3 -1 -3 -3 -2 -5
Q -1  1  0  0 -3  7  2 -2  1 -3 -2  2  0 -4 -1  0 -1 -1 -1 -3  0  4 -1 -5
E -1  0  0  2 -3  2  6 -3  0 -4 -3  1 -2 -3 -1 -1 -1 -3 -2 -3  1  5 -1 -5
G  0 -3  0 -1 -3 -2 -3  8 -2 -4 -4 -2 -3 -4 -2  0 -2 -3 -3 -4 -1 -2 -2 -5
H -2  0  1 -1 -3  1  0 -2 10 -4 -3  0 -1 -1 -2 -1 -2 -3  2 -4  0  0 -1 -5
I -1 -4 -3 -4 -2 -3 -4 -4 -4  5  2 -3  2  0 -3 -3 -1 -3 -1  4 -4 -3 -1 -5
```

…

È possibile ottenere un elenco delle matrici di *score* disponibili fornite con il pacchetto *Biostrings* utilizzando la funzione `data()`, che prende come argomento il nome del pacchetto per il quale si vuole conoscere i set di dati che lo accompagnano:

```
> data(package="Biostrings")
Data sets in package 'Biostring':
BLOSUM100                                   Scoring matrices
BLOSUM45                                    Scoring matrices
BLOSUM50                                    Scoring matrices
BLOSUM62                                    Scoring matrices
BLOSUM80                                    Scoring matrices
```

Per trovare l'allineamento globale ottimale tra le sequenze proteiche "PAWHEAE" e "HEAGAWGHEE" utilizzando l'algoritmo Needleman-Wunsch con la matrice BLOSUM50, digitiamo:

```
> data(BLOSUM50)
> s3 <- "PAWHEAE"
> s4 <- "HEAGAWGHEE"
> globalAligns3s4 <- pairwiseAlignment(s3, s4, substitutionMatrix = "BLOSUM50",␣
↪gapOpening = -2,
gapExtension = -8, scoreOnly = FALSE)
> globalAligns3s4 # Print out the optimal global alignment and its score
Global Pairwise Alignment (1 of 1)
pattern: [1] P---AWHEAE
subject: [1] HEAGAWGHEE
score: -5
```

Abbiamo impostato "`gapOpening`" a -2 e "`gapExtension`" a -8, il che significa che alla prima posizione di un *gap* viene assegnato un punteggio di -8-2 = -10, e ad ogni posizione successiva in un *gap* viene assegnato un punteggio di -8. Ciò significa che al *gap* verrà assegnato un punteggio di -10-8-8 = -26.

4.8 Allineamento di sequenze UniProt

Abbiamo discusso in precedenza di come è possibile cercare gli *accession* UniProt e recuperare le sequenze proteiche corrispondenti, sia tramite il sito web di UniProt che utilizzando pacchetti R.

Negli esempi sopra riportati, abbiamo visto come recuperare le sequenze per le proteine del *Mycobacterium leprae* (UniProt Q9CD83) e *Mycobacterium ulcerans* (UniProt A0PQ23), e leggerle in R e memorizzarle nei vettori *lepraeseq* e *ulceransseq*. È possibile allineare queste sequenze usando la funzione `pairwiseAlignment()` del pacchetto *Biostrings*.

Come input, la funzione `pairwiseAlignment()` richiede che le sequenze siano in forma di singola stringa (ad es. "ACGTA"), piuttosto che come vettore di caratteri (ad es. un vettore con il primo elemento come "A", il secondo come "C", ecc.). Pertanto, per allineare le proteine *M. leprae* e *M. ulcerans* dobbiamo prima convertire i vettori *lepraeeq* e *ulceransseq* in stringhe. Possiamo farlo usando la funzione `c2s()` del pacchetto *SeqinR*:

```
> lepraeseqstring <- c2s(lepraeseq)      # Make a string that contains the sequence␣
↪in "lepraeseq"
> ulceransseqstring <- c2s(ulceransseq) # Make a string that contains the sequence␣
↪in "ulceransseq"
```

Inoltre, `pairwiseAlignment()` richiede che le sequenze siano memorizzate come caratteri maiuscoli. Pertanto, se non sono già in maiuscolo, dobbiamo usare la funzione `toupper()` per convertire le variabili *lepraeseqstring* e *ulceransseqstring* in maiuscolo:

```
> lepraeseqstring <- toupper(lepraeseqstring)
> ulceransseqstring <- toupper(ulceransseqstring)
> lepraeseqstring # Print out the content of "lepraeseqstring"
  [1]
↪"MTNRTLSREEIRKLDRDLRILVATNGTLTRVLNVVANEEIVVDIINQQLLDVAPKIPELENLKIGRILQRDILLKGQKSGII
↪"
```

Ora possiamo allineare le sequenze delle proteine *M. leprae* e *M. ulcerans* utilizzando la funzione `pairwiseAlignment()`:

```
> globalAlignLepraeUlcerans <- pairwiseAlignment(lepraeseqstring,␣
↪ulceransseqstring,
  substitutionMatrix = BLOSUM50, gapOpening = -2, gapExtension = -8, scoreOnly =␣
↪FALSE)
> globalAlignLepraeUlcerans # Print out the optimal global alignment and its score
  Global PairwiseAlignedFixedSubject (1 of 1)
  pattern: [1] MT-----NR--T---LSREEIRKLDRDLRILVATN...
↪QDTPREELDRCQYSNDIDTRSGDRFVLHGRVFKN
  subject: [1] MLAVLPEKREMTECHLSDEEIRKLNRDLRILIATN...EDNSREEPIRHQRS--VGT-SA-R---
↪SGRSICT
  score: 627
```

Poiché l'allineamento è molto lungo, quando si digita *globalAlignLepraeUlcerans*, si vedono solo l'inizio e la fine dell'allineamento (vedi sopra). Pertanto, abbiamo bisogno di una funzione per stampare l'intero allineamento.

4.9 Visualizzazione di un allineamento lungo a coppie

Se si desidera visualizzare un allineamento lungo a coppie come quello tra le proteine di *M. leprae* e *M. ulerans*, è conveniente stampare l'allineamento in blocchi. La funzione R `printPairwiseAlignment()` (vedi Appendice) serve allo scopo.

Per utilizzare questa funzione è necessario prima attivarla mediante il comando:

```
> source("printPairwiseAlignment.R")
```

nel quale si suppone che il file di testo contenente la funzione si chiami "printPairwiseAlignment.R" e sia contenuto nella directory di lavoro.

È ora possibile utilizzare la funzione per stampare l'allineamento tra le proteine di *M. leprae* e *M. ulcerans* (che abbiamo memorizzato prima nella variabile *globalAlignLepraeUlcerans*), in blocchi di 60 colonne di allineamento:

```
> printPairwiseAlignment(globalAlignLepraeUlcerans, 60)
[1] "MT-----NR--T---LSREEIRKLDRDLRILVATNGTLTRVLNVVANEEIVVDIINQQLL 50"
[1] "MLAVLPEKREMTECHLSDEEIRKLNRDLRILIATNGTLTRILNVLANDEIVVEIVKQQIQ 60"
[1] " "
[1] "DVAPKIPELENLKIGRILQRDILLKGQKSGILFVAAESLIVIDLLPTAITTYLTKTHHPI 110"
[1] "DAAPEMDGCDHSSIGRVLRRDIVLKGRRSGIPFVAAESFIAIDLLPPEIVASLLETHRPI 120"
[1] " "
[1] "GEIMAASRIETYKEDAQVWIGDLPCWLADYGYWDLPKRAVGRRYRIIAGGQPVIITTEYF 170"
[1] "GEVMAASCIETFKEEAKVWAGESPAWLELDRRRNLPPKVVGRQYRVIAEGRPVIIITEYF 180"
[1] " "
[1] "LRSVFQDTPREELDRCQYSNDIDTRSGDRFVLHGRVFKN 219"
[1] "LRSVFEDNSREEPIRHQRS--VGT-SA-R---SGRSICT 219"
[1] " "
```

La posizione nella proteina dell'aminoacido che si trova alla fine di ogni riga dell'allineamento è indicata dopo la fine della riga stessa. Ad esempio, la prima riga dell'allineamento sopra indicata termina alla posizione dell'aminoacido 50 nella proteina *M. leprae* e alla posizione dell'aminoacido 60 nella proteina *M. ulcerans*. Poiché l'allineamento contiene dei *gap* nelle prime 60 colonne di allineamento, queste terminano prima del 60esimo aminoacido nella sequenza *M. leprae*.

4.10 Allineamento locale a coppie di sequenze proteiche mediante l'algoritmo di Smith-Waterman

È possibile utilizzare la funzione `pairwiseAlignment()` per trovare l'allineamento locale ottimale di due sequenze, cioè il miglior allineamento interno, mediante l'argomento `"type=local"` in `pairwiseAlignment()`. Tale opzione utilizza il classico algoritmo bioinformatico di Smith-Waterman per trovare l'allineamento locale ottimale, cioè regioni ad alta similarità in entrambe le sequenze. Ad esempio, per trovare il miglior allineamento locale tra le proteine *M. leprae* e *M. ulcerans*, possiamo digitare:

```
> localAlignLepraeUlcerans <- pairwiseAlignment(lepraeseqstring, ulceransseqstring,
      substitutionMatrix = BLOSUM50, gapOpening = -2, gapExtension = -8,
      scoreOnly = F, type = "local")
> localAlignLepraeUlcerans
Local PairwiseAlignmentsSingleSubject (1 of 1)
pattern:  [1]
MTNRTLSREEIRKLDRDLRILVATNGTLTRVLNVVANEEI...AGGQPVIITTEYFLRSVFQDTPREELDRCQYSNDIDTRSG
subject: [11]
MTECHLSDEEIRKLNRDLRILIATNGTLTRILNVLANDEI...AEGRPVIIITEYFLRSVFEDNSREEPIRHQRSVGTSARSG
score: 761
> printPairwiseAlignment(localAlignLepraeUlcerans)
[1] "MTNRTLSREEIRKLDRDLRILVATNGTLTRVLNVVANEEIVVDIINQQLLDVAPKIPELE 60"
[1] "MTECHLSDEEIRKLNRDLRILIATNGTLTRILNVLANDEIVVEIVKQQIQDAAPEMDGCD 60"
[1] " "
[1] "NLKIGRILQRDILLKGQKSGILFVAAESLIVIDLLPTAITTYLTKTHHPIGEIMAASRIE 120"
[1] "HSSIGRVLRRDIVLKGRRSGIPFVAAESFIAIDLLPPEIVASLLETHRPIGEVMAASCIE 120"
[1] " "
[1] "TYKEDAQVWIGDLPCWLADYGYWDLPKRAVGRRYRIIAGGQPVIITTEYFLRSVFQDTPR 180"
[1] "TFKEEAKVWAGESPAWLELDRRRNLPPKVVGRQYRVIAEGRPVIIITEYFLRSVFEDNSR 180"
[1] " "
[1] "EELDRCQYSNDIDTRSG 197"
[1] "EEPIRHQRSVGTSARSG 197"
[1] " "
```

Vediamo che l'allineamento locale ottimale è abbastanza simile all'allineamento globale ottimale in questo caso, tranne che esclude una breve regione di sequenza scarsamente allineata all'inizio e alla fine delle due proteine.

4.11 Calcolo della significatività statistica di un allineamento globale a coppie

Abbiamo visto che le sequenze proteiche "PAWHEAE" e "HEAGAWGHEE" hanno una certa similarità, e il punteggio per il loro allineamento globale ottimale è -5. Ma questo allineamento è statisticamente significativo? In altre parole, questo allineamento è migliore di quanto ci aspetteremmo tra due proteine casuali?

L'algoritmo di allineamento Needleman-Wunsch produrrà un allineamento globale anche con due sequenze proteiche casuali non correlate, pur con un punteggio di allineamento basso. Pertanto dovremmo chiederci: il punteggio per il nostro allineamento è migliore di quanto ci si aspetterebbe tra due sequenze casuali delle stesse lunghezze e composizioni di aminoacidi?

È ragionevole aspettarsi che se il punteggio di allineamento è statisticamente significativo, allora sarà più alto dei punteggi ottenuti dall'allineamento di coppie di sequenze proteiche casuali che hanno le stesse lunghezze e composizioni aminoacidiche delle due sequenze originali.

Pertanto, per valutare se il punteggio per il nostro allineamento tra la sequenza proteica "PAWHEAE" e "HEAGAWGHEE" è statisticamente significativo, un primo passo è quello di generare alcune sequenze casuali che hanno la stessa composizione aminoacidica e la stessa lunghezza di una delle due sequenze iniziali.

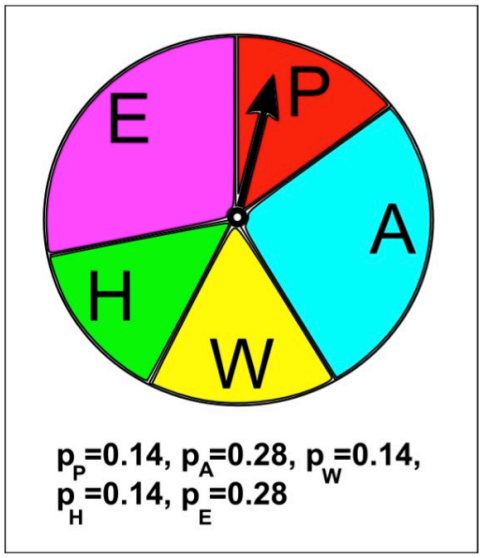
Come possiamo ottenere sequenze casuali della stessa composizione aminoacidica e lunghezza della sequenza "PAWHEAE"? Un modo è quello di generare sequenze utilizzando un *modello multinomiale per le sequenze di proteine* in cui le probabilità dei diversi aminoacidi sono impostate per essere uguali alle loro frequenze nella sequenza "PAWHEAE".

Ossia, possiamo generare sequenze utilizzando un modello multinomiale per le proteine, in cui ad es. la probabilità di "P" è impostata a 0,1428571 (1/7), la probabilità di "A" è impostata a 0,2857143 (2/7), la probabilità di "W" è impostata a 0,1428571 (1/7), la probabilità di "H" è impostata a 0,1428571 (1/7), quella di "E" è impostata a 0,2857143 (2/7) e le probabilità degli altri 15 aminoacidi sono impostate a 0.

Per generare una sequenza con questo modello multinomiale, scegliamo la lettera per ogni posizione della sequenza in base a tali probabilità. Questo equivale a una roulette in cui 1/7 della ruota è occupato da "P", 2/7 da "A", 1/7 da "W", 1/7 da "H" e 2/7 da "E".

Per generare una sequenza utilizzando il modello multinomiale, continuiamo a far girare la freccia al centro della ruota della roulette, e scriviamo la lettera su cui la freccia si ferma dopo ogni giro. Per generare una sequenza di 7 lettere, possiamo far girare la freccia 7 volte. Per generare 1.000 sequenze di 7 lettere

ciascuna, possiamo far girare la freccia 7.000 volte, dove le lettere scelte formano 1.000 sequenze di aminoacidi di 7 lettere.



Per generare un certo numero (ad esempio 1.000) sequenze di aminoacidi casuali di una certa lunghezza usando un modello multinomiale, si può usare la funzione `generateSeqsWithMultinomialModel()` (vedi Appendice) che genera *X* sequenze casuali con un modello multinomiale, dove le probabilità delle diverse lettere sono impostate uguali alle loro frequenze in una sequenza di ingresso, che viene passata alla funzione come una stringa di caratteri (ad es. "PAWHEAE").

La funzione restituisce *X* sequenze casuali sotto forma di un vettore di *X* elementi, dove il primo elemento contiene la prima sequenza, il secondo elemento la seconda sequenza, e così via. Possiamo usare questa funzione per generare 1.000 sequenze di aminoacidi di 7 lettere usando un modello multinomiale in cui le probabilità delle lettere sono impostate uguali alle loro frequenze in "PAWHEAE" (cioè probabilità 1/7 per "P", 2/7 per "A", 1/7 per "W", 1/7 per "H" e 2/7 per "E"), digitando:

```
> randomseqs <- generateSeqsWithMultinomialModel('PAWHEAE',1000)
> randomseqs[1:10] # Print out the first 10 random sequences
  [1] "EHHEWEA" "EAEEEAH" "WAHAWEP" "PPAPAAW" "HEPWWAA" "APAAAAA" "EAHAPHP"
  [8] "AAPEEWE" "HEAAAAP" "EWAAPEP"
```

Le 1.000 sequenze casuali sono memorizzate nel vettore *randomseqs* di 1.000 elementi, ognuno dei quali contiene una delle sequenze casuali. Possiamo quindi utilizzare l'algoritmo Needleman-Wunsch per allineare la sequenza "HEAGAWGHEE" con una delle 1.000 sequenze casuali generate utilizzando il modello multinomiale con probabilità 1/7 per "P", 2/7 per "A", 1/7 per "W", 1/7 per "H" e 2/7 per "E".

Ad esempio, per allineare "HEAGAWGHEE" alla prima delle 1.000 sequenze casuali ("EHHEWEA"), digitiamo:

```
> s4 <- 'HEAGAWGHEE'
> pairwiseAlignment(s4, randomseqs[1], substitutionMatrix = BLOSUM50,
      gapOpening = -2, gapExtension = -8, scoreOnly = FALSE)
Global PairwiseAlignmentsSingleSubject (1 of 1)
pattern: HEAGAWGHEE
subject: -EHHEW--EA
score: -7
```

Se usiamo la funzione `pairwiseAlignment()` con l'argomento `scoreOnly=TRUE`, ci darà solo il punteggio per l'allineamento:

```
> pairwiseAlignment(s4, randomseqs[1], substitutionMatrix = BLOSUM50,
      gapOpening = -2, gapExtension = -8, scoreOnly = TRUE)
[1] -7
```

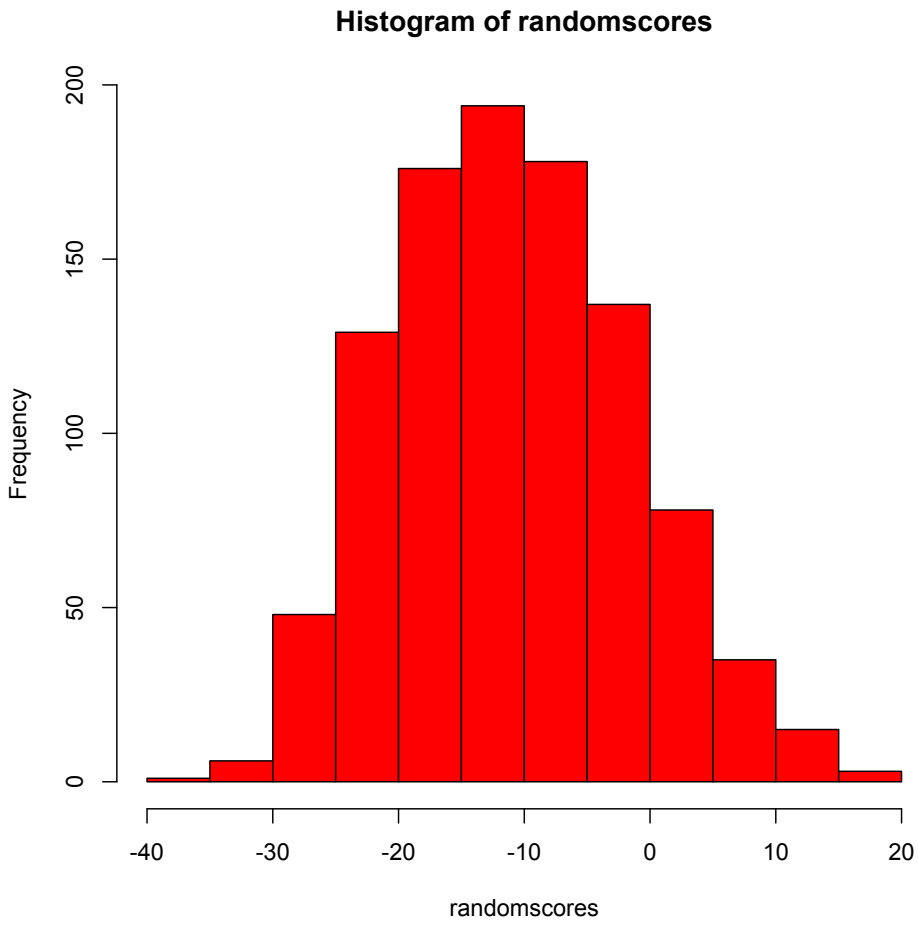
Se ripetiamo l'allineamento 1.000 volte, cioè per ognuna delle 1.000 sequenze casuali del vettore *randomseqs*, possiamo ottenere una distribuzione dei punteggi di allineamento previsti per l'allineamento di "HEAGAWGHEE" a sequenze casuali della stessa lunghezza e (approssimativamente la stessa) composizione aminoacidica di "PAWHEAE". Possiamo quindi confrontare il punteggio effettivo per l'allineamento di "PAWHEAE" a "HEAGAWGHEE" (cioè -5) con la distribuzione dei punteggi per l'allineamento di "HEAGAWGHEE" alle sequenze casuali:

```
> randomscores <- double(1000) # Create a numeric vector with 1000 elements
> for (i in 1:1000) {
      score <- pairwiseAlignment(s4, randomseqs[i], substitutionMatrix = "BLOSUM50",
      gapOpening = -2, gapExtension = -8, scoreOnly = TRUE)
      randomscores[i] <- score
  }
```

Il codice di cui sopra usa prima la funzione `double()` per creare il vettore *randomscores* di 1.000 numeri reali che sarà usato per memorizzare i punteggi di 1.000 allineamenti tra "HEAGAWGHEE" e le 1.000 diverse sequenze casuali generate usando il modello multinomiale.

Il ciclo `for` prende ognuna delle 1.000 diverse sequenze casuali, allinea ciascuna di esse a "HEAGAWGHEE", e memorizza i 1.000 punteggi di allineamento nel vettore *randomscores*. Una volta eseguito il ciclo, possiamo fare un istogramma dei 1.000 punteggi nel vettore *randomscores* digitando:

```
> hist(randomscores, col="red") # Draw a red histogram
```



Possiamo vedere dall'istogramma che molte delle sequenze casuali sembrano avere punteggi di allineamento più alti di -5 quando allineate a "HEAGAWGHEE" (dove -5 è il punteggio di allineamento per "PAWHEAE" e "HEAGAWGHEE"). Possiamo usare il vettore *randomscores* dei punteggi per 1.000 allineamenti di sequenze casuali a "HEAGAWGHEE" per calcolare la probabilità di ottenere per caso un punteggio almeno pari al reale punteggio di allineamento di "PAWHEAE" e "HEAGAWGHEE" (cioè -5):

```
> sum(randomscores >= -5)
[1] 266
```

Vediamo che 266 dei 1.000 allineamenti di sequenze casuali a "HEAGAWGHEE" hanno punteggi di allineamento pari o superiori a -5. Così, possiamo stimare che la probabilità (*p*-value) di ottenere per caso un punteggio almeno pari al reale punteggio di allineamento sia 266/1.000 = 0,266. Questa probabilità è abbastanza alta (quasi il 30%), quindi possiamo concludere che è abbastanza probabile che si possa ottenere solo per caso un punteggio di allineamento almeno pari a -5. Ciò indica che le sequenze "HEAGAWGHEE" e "PAWHEAE" non sono più simili di due sequenze casuali, e quindi probabilmente non sono sequenze correlate.

Un altro modo per dire questo è che il *p*-value che abbiamo calcolato è alto (0,266), e di conseguenza concludiamo che il punteggio di allineamento per le sequenze "HEAGAWGHEE" e "PAWHEAE" non è statisticamente significativo. Generalmente, se il *p*-value che calcoliamo per un allineamento di due sequenze è > 0,05 concludiamo che il punteggio di allineamento non è statisticamente significativo, e che le sequenze non sono probabilmente correlate. D'altra parte, se il *p*-value è inferiore o uguale a 0,05 si conclude che il punteggio di allineamento è statisticamente significativo, e che le sequenze sono molto probabilmente correlate (omologhe).

5. Allineamenti multipli (MSA) e alberi filogenetici

Vi sono occasioni nell'analisi delle sequenze in cui dobbiamo confrontare molte sequenze l'una con l'altra (allineamenti multipli o MSA, Multiple Sequence Alignment). Un esempio generale di questo caso è l'analisi filogenetica, che sarà discussa nelle prossime sezioni, dopo aver esaminato come fare l'allineamento di sequenze multiple (MSA) utilizzando R.

L'analisi filogenetica riguarda la ricerca della relazione evolutiva tra le specie (organismi), nel nostro caso, sulla base dei dati di sequenza. Una volta che abbiamo un insieme di sequenze provenienti da fonti diverse, diventa molto interessante capire quanto siano vicine o lontane in termini di evoluzione molecolare. Come risultato delle mutazioni durante l'evoluzione, emergono differenze a livello di sequenza. Queste differenze possono essere rappresentate in termini di misure di distanza. Queste misure possono quindi essere utilizzate per stimare le relazioni evolutive tra le specie, spesso rappresentate come alberi filogenetici. Finora, abbiamo esaminato i vari aspetti del recupero di sequenze, dell'allineamento e dell'analisi. In questa sezione illustreremo come eseguire un'analisi filogenetica sui dati di sequenza.

5.1 Recuperare una lista di sequenze proteiche

Nei capitoli precedenti, abbiamo imparato a cercare sequenze di DNA o proteine nel database di sequenze NCBI o UniProt utilizzando il pacchetto *rentrez*. Spesso è utile recuperare diverse sequenze in una sola volta se si dispone di un elenco di *accession UniProt*. La funzione `retrieventrezseqs()` (vedi Appendice) è utile per questo scopo.

Come input, bisogna dare alla funzione un vettore che contenga gli *accession* per le sequenze che si vogliono recuperare, così come il nome del sotto-database dal quale le sequenze devono essere recuperate. In questo caso, vogliamo recuperare le sequenze proteiche, quindi le sequenze dovrebbero essere nel sotto-database NCBI "protein".

La funzione `retrieventrezseqs()` restituisce una variabile lista, in cui ogni elemento è un vettore contenente una delle sequenze cercate.

Ad esempio, per recuperare le sequenze proteiche per gli *accession* UniProt P06747, P0C569, O56773 e Q5VKP1 (la fosfoproteina del virus della rabbia, la fosfoproteina del virus Mokola, la fosfoproteina del virus del pipistrello del Lagos e la fosfoproteina del virus del pipistrello del Caucaso occidentale), è possibile digitare:

```
> seqnames <- c("P06747", "P0C569", "O56773", "Q5VKP1") # Make a vector containing the names of the sequences
> seqs <- retrieventrezseqs(seqnames,"protein") # Retrieve the sequences and store them in the list variable "seqs"
> length(seqs) # Print out the number of sequences retrieved
 [1] 4
> seq1 <- seqs[[1]] # Get the first sequence
> seq1[1:20] # Print out the first 20 letters of the first sequence
 [1] "M" "S" "K" "I" "F" "V" "N" "P" "S" "A" "I" "R" "A" "G" "L" "A" "D" "L" "E" "M"
> seq2 <- seqs[[2]] # Get the second sequence
> seq2[1:20] # Print out the first 20 letters of the second sequence
 [1] "M" "S" "K" "D" "L" "V" "H" "P" "S" "L" "I" "R" "A" "G" "I" "V" "E" "L" "E" "M"
```

I comandi precedenti usano la funzione `retrieventrezseqs()` per recuperare sequenze di proteine nella variabile lista *seqs*. Ogni elemento della variabile *seqs* contiene un vettore che memorizza una delle sequenze.

Una volta recuperate le sequenze, è possibile utilizzare la funzione `write.fasta()` del pacchetto *seqinr* per scrivere le sequenze in un file in formato FASTA.

Come argomenti la funzione ha la variabile lista contenente le sequenze, un vettore contenente i nomi delle sequenze, e il nome che si vuole dare al file in formato FASTA. Per esempio:

```
> write.fasta(seqs, seqnames, file="phosphoproteins.fasta")
```

Il comando scrive le sequenze nella variabile *seqs* in un file in formato FASTA chiamato "phosphoproteins.fasta" nella directory di lavoro.

5.2 Allineamento multiplo di sequenze dalla banca dati PDB con il pacchetto R *bio3d*

Un altro modo per recuperare le sequenze proteiche è dalla banca dati PDB (Protein Data Bank; https://www.rcsb.org), che si concentra principalmente su informazioni relative a strutture di proteine, acidi nucleici e assemblaggi complessi determinati sperimentalmente. Possiamo quindi recuperare direttamente i dati delle sequenze da PDB ed eseguire l'allineamento multiplo in R.

La posizione di un allineamento di sequenze multiple di proteine gioca un ruolo importante; il colore è spesso usato per indicare le proprietà degli aminoacidi e per aiutare a giudicare la conservazione di una data sostituzione di aminoacidi. Questo influenzerà in ultima analisi le proprietà strutturali, funzionali e fisico-chimiche delle proteine. Presenteremo un modo semplice per recuperare le sequenze mediante il pacchetto *bio3d*, creare l'allineamento multiplo ed esaminare il risultato osservando la colorazione dell'allineamento.

Il pacchetto *bio3d* è specializzato nella gestione delle sequenze proteiche e gestisce i file PDB, che contengono una rappresentazione dei dati della struttura macromolecolare derivata da studi di diffrazione dei raggi X e NMR. Un file PDB contiene le sequenze e le coordinate dei residui di aminoacidi. La funzione `read.pdb()` legge dalla banca dati PDB il file corrispondente alla proteina indicata.

Nel nostro esempio, per prima cosa recupereremo le sequenze proteiche dal database PDB. I residui di aminoacidi sono rappresentati con tre lettere e la funzione `aa321()` li converte in rappresentazioni di una lettera. Un'altra funzione simile, `aa123()`, effettua le conversioni al contrario. Combineremo poi le sequenze in una matrice con la funzione `seqbind()` e infine eseguiremo l'allineamento multiplo vero e proprio con la funzione `seqaln()` che utilizza il programma MUSCLE per allineare le sequenze.

Innanzitutto occorre quindi installare il software MUSCLE (MUltiple Sequence Comparison by Log-Expectation) sul proprio sistema. Ad esempio, su un sistema Linux basta digitare dalla riga di comando: `$ sudo apt-get install muscle`; per altri sistemi operativi, riferirsi al sito web del produttore (https://www.drive5.com/muscle/).

Installiamo e carichiamo quindi il pacchetto R *bio3d* secondo la procedura standard:

```
> install.packages("bio3d", dependencies = TRUE)
> library(bio3d)
```

Recuperiamo l'insieme delle proteine (provenienti da tre diversi organismi, associate ai microtubuli, che possono avere un ruolo nel trasporto degli organuli) da PDB usando la funzione `read.pdb()` come segue:

```
> pdb1 <- read.pdb("1BG2")
> pdb2 <- read.pdb("2VVG")
> pdb3 <- read.pdb("1MKJ")
```

Estraiamo da queste proteine le sequenze da allineare utilizzando la funzione `aa321()`, che legge la rappresentazione di tre lettere per gli aminoacidi e la converte in codici di una lettera:

```
> s1 <- aa321(as.character(pdb1$seqres))
> s2 <- aa321(as.character(pdb2$seqres))
> s3 <- aa321(as.character(pdb3$seqres))
```

Per allineare le sequenze, le inseriamo in una matrice con la funzione `seqbind()` (che restituisce un oggetto `fasta`) ed effettuiamo l'allineamento con `seqaln()` come segue:

```
> seqsmat <- seqbind(s1,s2,s3)
> aln <- seqaln(seqsmat, id=c("1BG2","2VVG","1MKJ"))
```

Salviamo infine l'allineamento come file HTML nella directory di lavoro con la funzione `aln2html()`:

```
aln2html(aln, append=FALSE, file="Myalign.html")
```

Apriamo il file HTML in un browser per visualizzare i risultati. La seguente schermata mostra una parte dell'allineamento (scorrere orizzontalmente la finestra del browser per vedere l'intero allineamento) per le tre sequenze di proteine 1BG2, 2VVG e 1MKJ con i relativi codici di colore standard CLUSTAL (vedi la sezione successiva); l'allineamento mostra le differenze così come i domini conservati (a colori):

5.3 Installazione del software di allineamento multiplo CLUSTAL

Un compito comune nella bioinformatica è quello di scaricare un insieme di sequenze correlate da un database, e quindi di allineare tali sequenze utilizzando un software di allineamento multiplo. Questo è il primo passo nella maggior parte delle analisi filogenetiche.

Un pacchetto software di allineamento multiplo comunemente usato è CLUSTAL[1]. Per costruire un allineamento usando CLUSTAL, è necessario prima installare il programma seguendo questi passi:

- Andare al sito web http://www.clustal.org/download/current/.
- Cliccare con il tasto destro del mouse sul link del file "clustalx-2.1-…" (con l'estensione a seconda del sistema operativo), scegliere "Scarica file collegato col nome..." e poi salvare il file nella cartella desiderata.
- Una volta che il file è stato scaricato, fare doppio clic sulla sua icona.
- Verrà chiesto "Siete sicuri di voler eseguire questo software? Premere "Esegui".
- Si vedrà quindi "Benvenuti nella procedura guidata di configurazione di ClustalX2". Premete "Avanti".
- Verrà chiesto dove installare ClustalX2. Selezionare la cartella desiderata.
- Continuare a premere "Sì" o "Avanti" fino a quando lo schermo non dice "Completamento della procedura guidata di installazione di ClustalX2". Poi premere "Fine".

CLUSTAL dovrebbe ora essere installato sul computer.

5.4 Creazione di un allineamento multiplo di sequenze di proteine, DNA o mRNA mediante CLUSTAL

Una volta installato CLUSTAL, è ora possibile allineare le sequenze utilizzando CLUSTAL seguendo questi passi:

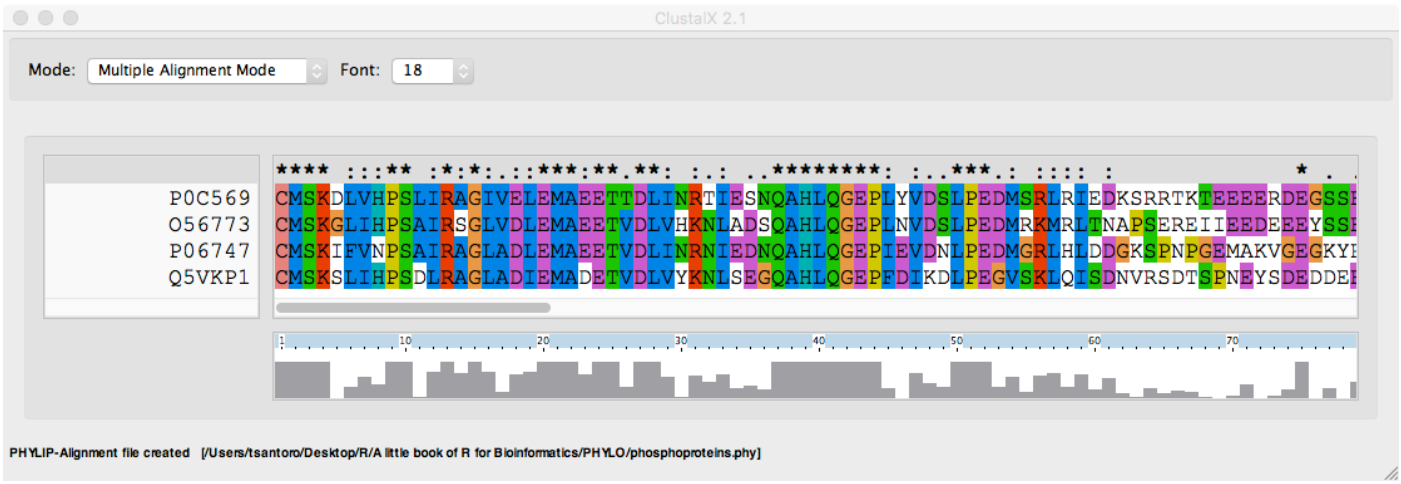
- Avviare il programma CLUSTAL.
- Per caricare le sequenze di DNA o proteine che si desidera allineare in CLUSTAL, andare al menu "File" di CLUSTAL e scegliere "Load sequences".
- Selezionare il file in formato FASTA contenente le sequenze (ad es. "phosphoproteins.fasta") per caricarlo in CLUSTAL.
- Le sequenze non sono state ancora allineate, ma saranno visualizzate nella finestra di CLUSTAL.
- È possibile utilizzare la barra di scorrimento a destra per scorrere verso il basso e guardare tutte le sequenze, oppure la barra di scorrimento in basso per scorrere da sinistra a destra e guardare lungo la lunghezza delle sequenze.

---

[1] Altri software open-source di allineamento multiplo di sequenze sono **JalView** (http://www.jalview.org, un programma gratuito e open source in Java sviluppato per l'editing interattivo, l'analisi e la visualizzazione di allineamenti di sequenze multiple. Può anche lavorare con l'annotazione delle sequenze, le informazioni sulle strutture secondarie, gli alberi filogenetici e le strutture molecolari 3D) e l'ottimo **MEGA-X** (https://www.megasoftware.net/web_help_10/index.htm#t=Introduction.htm; fornisce strumenti per esplorare, scoprire e analizzare le sequenze di DNA e proteine da una prospettiva evolutiva; ha inoltre capacità prestazionali avanzate per la costruzione di alberi di bootstrap, la selezione di modelli e metodi per la costruzione di alberi temporali).

- Prima di allineare le sequenze, è necessario dire a CLUSTAL di impostare il file di allineamento di uscita nel formato di allineamento PHYLIP, in modo da poterlo poi leggere in R. Per fare questo, andare al menu "Alignment" in CLUSTAL, scegliere "Output Format Options". Apparirà un modulo, e in questo modulo dovrete selezionare "PHYLIP format" e deselezionare "CLUSTAL format", e poi premere "OK".
- Per allineare le sequenze con CLUSTAL, andare al menu "Alignment" di CLUSTAL e scegliere "Do complete alignment".
- Apparirà una casella di menu che chiederà dove salvare il file di uscita *guide-tree* (con estensione ".dnd") e il file di output di allineamento (con estensione ".phy").
- A questo punto CLUSTAL allineerà le sequenze. Questo richiederà qualche minuto; nella parte inferiore della finestra CLUSTAL dirà quale coppia di sequenze sta allineando in un determinato momento. Se i numeri continuano a cambiare, significa che CLUSTAL sta ancora lavorando, e l'allineamento non è ancora finito.

Una volta che CLUSTAL ha finito di effettuare l'allineamento, esso verrà visualizzato nella finestra centrale. Ad esempio, ecco l'allineamento CLUSTAL per la fosfoproteina del virus Mokola (P0C569), del virus del pipistrello del Lagos (O56773), del virus della rabbia (P06747) e del virus del pipistrello del Caucaso occidentale (Q5VKP1):



L'allineamento visualizzato in CLUSTAL ha una riga per ciascuna delle sequenze. CLUSTAL colora le serie di aminoacidi chimicamente simili con colori simili. Ad esempio, la tirosina (Y) è di colore blu-verde, mentre l'aminoacido fenilalanina (F) chimicamente simile è di colore blu.

Sotto l'allineamento, si può vedere un grafico grigio che mostra il livello di similarità in ogni punto della sequenza. Vi è una barra grigia alta se la similarità in una regione è alta (c'è un'alta percentuale di similarità tra le sequenze), e una barra grigia bassa se c'è una bassa percentuale di similarità. Nel nostro caso, questo può dare un'idea di quali siano le regioni meglio conservate dell'allineamento.

Per esempio, per l'allineamento delle quattro fosfoproteine virali, possiamo vedere che la regione nelle colonne di allineamento da 35 a 45 circa è molto ben conservata, mentre la regione nelle colonne di allineamento da 60 a 70 è scarsamente conservata.

L'allineamento CLUSTAL può essere salvato in un file chiamato ad es. "phosphoproteins.phy", un file di allineamento in formato PHYLIP che può essere letto in R per ulteriori analisi.

5.5 Lettura di un file di allineamento multiplo in R

Per leggere un allineamento di sequenze in R da un file, si può usare la funzione `read.alignment()` del pacchetto *seqinr*. Per esempio, per leggere l'allineamento di sequenze multiple delle *fosfoproteine* virali in R, digitiamo:

```
> virusaln  <- read.alignment(file = "phosphoproteins.phy", format = "phylip")
```

La variabile *virusaln* è una variabile lista che memorizza l'allineamento. Una variabile lista in R può avere elementi denominati, e si può accedere a tali elementi di una variabile digitando il nome della variabile, seguito da "$", seguito dal nome dell'elemento denominato.

La variabile *virusaln* ha gli elementi denominati "nb", "nam", "seq" e "com". Infatti, l'elemento denominato "seq" contiene l'allineamento, che è possibile visualizzare digitando:

```
> virusaln$seq
  [[1]]
  [1] "mskdlvhpsliragivelemaeettdlinrtiesnqahlqgeplyvdslpedmsrlriedksrrtk...
  [[2]]
  [1] "mskglihpsairsglvdlemaeetvdlvhknladsqahlqgeplnvdslpedmrkmrltnapsere...
  [[3]]
  [1] "mskifvnpsairagladlemaeetvdlinrniednqahlqgepievdnlpedmgrlhlddgkspnp...
  [[4]]
  [1] "mskslihpsdlragladiemadetvdlvyknlsegqahlqgepfdikdlpegvsklqisdnvrsdt...
```

Qui viene mostrata solo la prima parte dell'allineamento memorizzata in *virusaln$seq*, data la sua lunghezza.

5.6 Visualizzazione di un allineamento multiplo lungo

Se si desidera visualizzare un allineamento multiplo lungo, è conveniente visualizzare l'allineamento multiplo in blocchi. La funzione "`printMultipleAlignment()`" (vedi Appendice) serve allo scopo.

La funzione ha come argomenti l'allineamento in input e il numero di colonne da visualizzare per ogni blocco. Ad esempio, per esaminare l'allineamento multiplo delle *fosfoproteine* virali (memorizzato nella variabile *virusaln*) in blocchi di 60 colonne, digitiamo:

```
> printMultipleAlignment(virusaln, 60)
```

Otteniamo così la lista visualizzata nella pagina seguente:

```
> printMultipleAlignment(virusaln, 60)
  [1] "MSKDLVHPSLIRAGIVELEMAEETTDLINRTIESNQAHLQGEPLYVDSLPEDMSRLRIED 60"
  [1] "MSKGLIHPSAIRSGLVDLEMAEETVDLVHKNLADSQAHLQGEPLNVDSLPEDMRKMRLTN 60"
  [1] "MSKIFVNPSAIRAGLADLEMAEETVDLINRNIEDNQAHLQGEPIEVDNLPEDMGRLHLDD 60"
  [1] "MSKSLIHPSDLRAGLADIEMADETVDLVYKNLSEGQAHLQGEPFDIKDLPEGVSKLQISD 60"
  [1] " "
  [1] "KSRRTKTEEEERDEGSSEEDNYLSEGQDPLIPFQNFLDEIGARAVKRLKTGEGFFRVWSA 120"
  [1] "APSEREIIEEDEEEYSSEDEYYLSQGQDPMVPFQNFLDELGTQIVRRMKSGDGFFKIWSA 120"
  [1] "GKSPNPGEMAKVGEGKYREDFQMDEGEDPSLLFQSYLDNVGVQIVRQIRSGERFLKIWSQ 120"
  [1] "NVRSDTSPNEYSDEDDEEGEDEYEEVYDPVSAFQDFLDETGSYLISKLKKGEKIKKTWSE 120"
  [1] " "
  [1] "LSDDIKGYVSTNIM-TSGERDTKSIQIQTEPTASVSSGNESRHDSESMHDPNDKKDHTPD 179"
  [1] "ASEDIKGYVLSTFM-KPETQATVSKPTQTDSLSVPRPSQGYTSVPRDKPSNSESQGGGVK 179"
  [1] "TVEEIISYVAVNFP-NPPGKSSEDKSTQTTGRELKKETTPTPSQRESQSSKARMAAQTAS 179"
  [1] "VSRVIYSYVMSNFPPRPPKPTTKDIAVQADLKKPNEIQKISEHKSKSEPSPREPVVEMHK 180"
  [1] " "
  [1] "HDVVPDIESSTDKGEIRDIEGEVAHQVAESFSKKYKFPSRSSGIFLWNFEQLKMNLDDIV 239"
  [1] "PKKVQKSEWTRDTDEISDIEGEVAHQVAESFSKKYKFPSRSSGIFLWNFEQLKMNLDDIV 239"
  [1] "GPPALEWSATNEEDDLS-VEAEIAHQIAESFSKKYKFPSRSSGILLYNFEQLKMNLDDIV 238"
  [1] "HATLE-----NPEDDEGALESEIAHQVAESYSKKYKFPSKSSGIFLWNFEQLKMNLDDIV 235"
  [1] " "
  [1] "KAAMNVPGVERIAEKGGKLPLRCILGFVALDSSKRFRLLADNDKVARLIQEDINSYMARL 299"
  [1] "KTSMNVPGVDKIAEKGGKLPLRCILGFVSLDSSKRFRLLADTDKVARLMQDDIHNYMTRI 299"
  [1] "KEAKNVPGVTRLARDGSKLPLRCVLGWVALANSKKFQLLVESNKLSKIMQDDLNRYTSC- 297"
  [1] "QVARGVPGISQIVERGGKLPLRCMLGYVGLETSKRFRSLVNQDKLCKLMQEDLNAYSVSS 295"
  [1] " "
  [1] "EEAE-- 357"
  [1] "EEIDHN 359"
  [1] "------ 351"
  [1] "NN---- 351"
  [1] " "
```

5.7 Eliminare da un allineamento multiplo le regioni scarsamente conservate

Spesso è una buona idea eliminare le regioni scarsamente conservate da un allineamento mulitplo prima di costruire un albero filogenetico, in quanto le regioni scarsamente conservate sono probabilmente regioni che non sono omologhe tra le sequenze considerate (e quindi non aggiungono alcun segnale filogenetico), oppure sono omologhe ma sono così divergenti che sono molto difficili da allineare accuratamente (e quindi possono aggiungere rumore all'analisi filogenetica, e diminuire la precisione dell'albero dedotto).

Per scartare regioni molto poco conservate da un allineamento multiplo, si può usare la funzione `cleanAlignment()` (vedi Appendice) che ha tre argomenti: l'allineamento in input; la percentuale minima di lettere in una colonna di allineamento che deve essere di caratteri non *gap* perché la colonna venga mantenuta; la percentuale minima di coppie di lettere in una colonna di allineamento che deve essere identica per la colonna da mantenere.

Per esempio, se abbiamo una colonna con le lettere "T", "A", "T", "-" (in quattro sequenze), allora il 75% (3/4) delle lettere sono caratteri non spaziati; le coppie di lettere non *gap* sono "T,A", "T,T", e "A,T", e il 33% (1/3) delle coppie di lettere sono identiche.

Possiamo usare la funzione `cleanAlignment()` per scartare da un allineamento multiplo le colonne scarsamente conservate.

Ad esempio, se si guarda l'allineamento multiplo per le sequenze delle *fosfoproteine* virali (che abbiamo visualizzato prima utilizzando la funzione `printMultipleAlignment()`), possiamo vedere che le ultime colonne sono mal allineate (contengono molti *gap* e disallineamenti), e probabilmente aggiungono rumore all'analisi filogenetica.

Pertanto, per filtrare le colonne dell'allineamento ben conservate, possiamo digitare:

```
> cleanedvirusaln <- cleanAlignment(virusaln, 30, 30)
```

In questo caso, abbiamo richiesto che almeno il 30% delle lettere in una colonna non sia costituito da *gap*, e che almeno il 30% delle coppie di lettere in una colonna di allineamento deve essere identica. Possiamo ottenere l'allineamento filtrato digitando:

```
> printMultipleAlignment(cleanedvirusaln)
  [1] "MSKLVHPSIRAGIVELEMAEETTDLIRTIQAHLQGEPVDLPEDMRLIDREEEDEGDPFQF 60"
  [1] "MSKLIHPSIRSGLVDLEMAEETVDLVKNLQAHLQGEPVDLPEDMKMLNSEEEEQGDPFQF 60"
  [1] "MSKFVNPSIRAGLADLEMAEETVDLIRNIQAHLQGEPVDLPEDMRLLDSAERDEGDPFQY 60"
  [1] "MSKLIHPSLRAGLADIEMADETVDLVKNLQAHLQGEPIKLPEGVKLIDREEEEEVDPFQF 60"
  [1] " "
  [1] "LDEGVKGEFRWSSIGYVNIMSTSIQTHSDESGEDEEVAHQVAESFSKKYKFPSRSSGIFL 120"
  [1] "LDEGVKGDFKWSSIGYVTFMPTSKQTSDSETDEDEEVAHQVAESFSKKYKFPSRSSGIFL 120"
  [1] "LDNGVRGEFKWSVISYVNFPPSDKQTSSSSTDD-EEIAHQIAESFSKKYKFPSRSSGILL 119"
  [1] "LDEGIKGEIKWSSISYVNFPPTDIQAHSS--DDAEEIAHQVAESYSKKYKFPSKSSGIFL 118"
  [1] " "
  [1] "WNFEQLKMNLDDIVKANVPGVIAEGGKLPLRCLGVLSKRFRLLADKVRLIQEDINYEE 180"
  [1] "WNFEQLKMNLDDIVKSNVPGVIAEGGKLPLRCLGVLSKRFRLLADKVRLMQDDIHYEE 180"
  [1] "YNFEQLKMNLDDIVKANVPGVLARGSKLPLRCLGVLSKKFQLLVNKLKIMQDDLNY-- 177"
  [1] "WNFEQLKMNLDDIVQAGVPGIIVEGGKLPLRCLGVLSKRFRSLVDKLKLMQEDLNYNN 178"
  [1] " "
```

L'allineamento filtrato è più breve perché mancano alcune delle regioni mal conservate dell'allineamento originale.

Si noti che non è una buona idea filtrare troppo l'allineamento, in quanto se restano poche colonne nell'allineamento filtrato, l'albero filogenetico sarà basato su un allineamento molto breve (pochi dati) e quindi potrebbe essere inaffidabile. Pertanto, è necessario raggiungere un equilibrio tra l'eliminazione delle parti poco allineate e il mantenimento di un numero sufficiente di colonne dell'allineamento su cui basare l'albero filogenetico.

5.8 Calcolo delle distanze genetiche tra sequenze proteiche

Un primo passo comune nell'esecuzione di un'analisi filogenetica è il calcolo delle distanze genetiche a coppie tra sequenze. La distanza genetica è una stima della divergenza tra due sequenze, ed è solitamente misurata in quantità di cambiamento evolutivo (una stima del numero di mutazioni che si sono verificate da quando le due sequenze hanno condiviso un antenato comune).

Possiamo calcolare le distanze genetiche tra sequenze di proteine usando la funzione `dist.alignment()` del pacchetto *seqinr*. La funzione prende come input un allineamento multiplo. Sulla base dell'allineamento multiplo fornito, `dist.alignment()` calcola la distanza genetica tra ogni

coppia di proteine nell'allineamento multiplo. Per esempio, per calcolare le distanze genetiche tra le *fosfoproteine* virali in base all'allineamento di sequenze multiple memorizzato in *virusaln*, digitiamo:

```
> virusdist <- dist.alignment(virusaln)                    # Calculate the␣
↪genetic distances
> virusdist                                                # Print out the␣
↪genetic distance matrix
              P0C569     O56773     P06747
O56773     0.4142670
P06747     0.4678196  0.4714045
Q5VKP1     0.4828127  0.5067117  0.5034130
```

La matrice delle distanze genetiche sopra mostra la distanza genetica tra ogni coppia di proteine.

Le sequenze sono riferite ai loro *accession* UniProt. Ricordiamo che P06747 è la fosfoproteina del virus della rabbia, P0C569 è la fosfoproteina del virus Mokola, O56773 è la fosfoproteina del virus del pipistrello del Lagos e Q5VKP1 è la fosfoproteina del virus del pipistrello del Caucaso occidentale.

Sulla base della matrice di distanza genetica di cui sopra, possiamo vedere che la distanza genetica tra la fosfoproteina del virus del pipistrello del Lagos (O56773) e la fosfoproteina del virus Mokola (P0C569) è la più piccola (circa 0,414). Allo stesso modo, la distanza genetica tra la fosfoproteina del virus del pipistrello del Caucaso occidentale (Q5VKP1) e la fosfoproteina del virus del pipistrello del Lagos (O56773) è la più grande (circa 0,507).

Più grande è la distanza genetica tra due sequenze, più cambiamenti di aminoacidi o inserimenti/cancellazioni si sono verificati da quando hanno condiviso un antenato comune, e più tempo fa il loro antenato comune probabilmente è vissuto.

5.9 Calcolo delle distanze genetiche tra sequenze di DNA/mRNA

Proprio come per le sequenze di proteine, è possibile calcolare le distanze genetiche tra sequenze di DNA (o mRNA) sulla base di un loro allineamento.

Ad esempio, l'*accession* NCBI AF049118 contiene la sequenza mRNA V241 per la fosfoproteina del virus Mokola, AF049114 contiene la sequenza mRNA V006 per la fosfoproteina del pipistrello del Lagos, AF049119 contiene la sequenza mRNA V267 per la fosfoproteina del virus del pipistrello del Lagos, mentre AF049115 contiene la sequenza mRNA V008 per la fosfoproteina del virus Duvenhage.

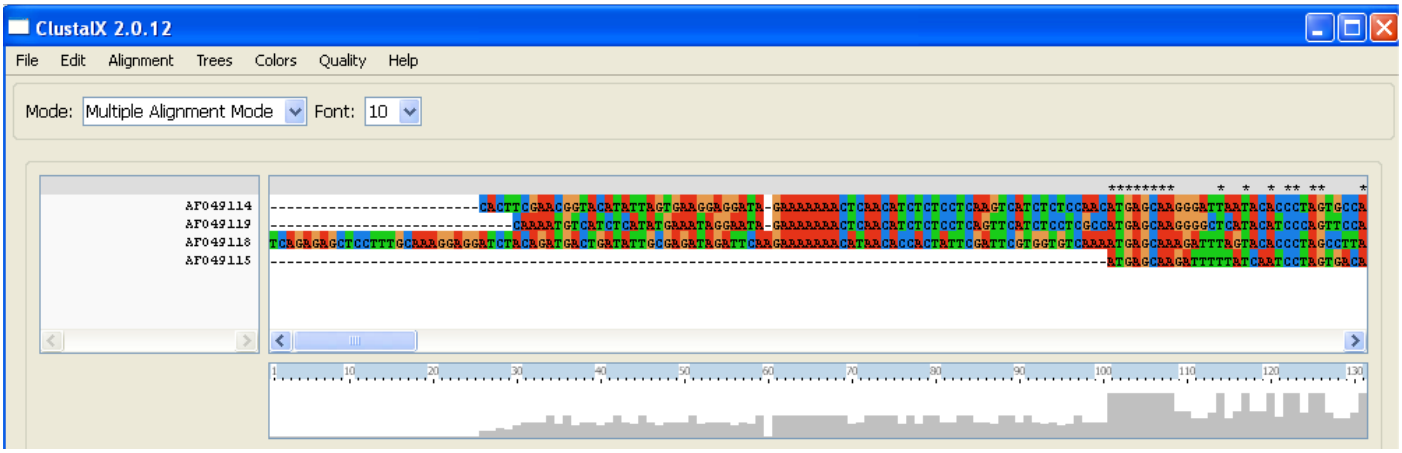
Per recuperare queste sequenze dal database NCBI, possiamo cercare nel sotto-database NretrieventrezCBI "nucleotide" (poiché si tratta di sequenze nucleotidiche), digitando:

```
> seqnames <- c("AF049118", "AF049114", "AF049119", "AF049115")
> seqs <- retrieventrezseqs(seqnames,"nucleotide")
[1] "Retrieving sequence AF049118 ..."
[1] "Retrieving sequence AF049114 ..."
[1] "Retrieving sequence AF049119 ..."
[1] "Retrieving sequence AF049115 ..."
```

Possiamo quindi scrivere le sequenze in un file in formato FASTA digitando:

```
> write.fasta(seqs, seqnames, file="virusmRNA.fasta")
```

Usiamo CLUSTAL per creare un allineamento in formato PHYLIP delle sequenze e memorizzarlo nel file "virusmRNA.phy". Questa schermata mostra una parte dell'allineamento:



Possiamo leggere l'allineamento in R:

```
> virusmRNAaln  <- read.alignment(file = "virusmRNA.phy", format = "phylip")
```

Abbiamo visto che la funzione `dist.alignment()` può essere utilizzata per calcolare una matrice di distanze genetiche basata sull'allineamento di una sequenza proteica.

È possibile calcolare una distanza genetica per le sequenze di DNA o mRNA utilizzando la funzione `dist.dna()` del pacchetto *ape*. Tale funzione prende come input un allineamento multiplo di sequenze di DNA o mRNA e calcola la distanza genetica tra ogni coppia di sequenze di DNA nell'allineamento multiplo.

La funzione `dist.dna()` richiede che l'allineamento in input sia in un formato speciale noto come formato "DNAbin", quindi dobbiamo usare la funzione `as.DNAbin()` per convertire il nostro allineamento di DNA in questo formato prima di usare la funzione `dist.dna()`.

Ad esempio, per calcolare la distanza genetica tra ogni coppia di sequenze mRNA per le *fosfoproteine* virali indicate in precedenza, digitiamo:

```
> virusmRNAalnbin <- as.DNAbin(virusmRNAaln) # Convert the alignment to "DNAbin"␣
↪format
> virusmRNAdist <- dist.dna(virusmRNAalnbin) # Calculate the genetic distance␣
↪matrix
> virusmRNAdist                              # Print out the genetic distance␣
↪matrix
            AF049114  AF049119  AF049118
AF049119 0.3400576
AF049118 0.5235850 0.5637372
AF049115 0.6854129 0.6852311 0.7656023
```

5.10 Costruire un albero filogenetico *unrooted* per sequenze proteiche

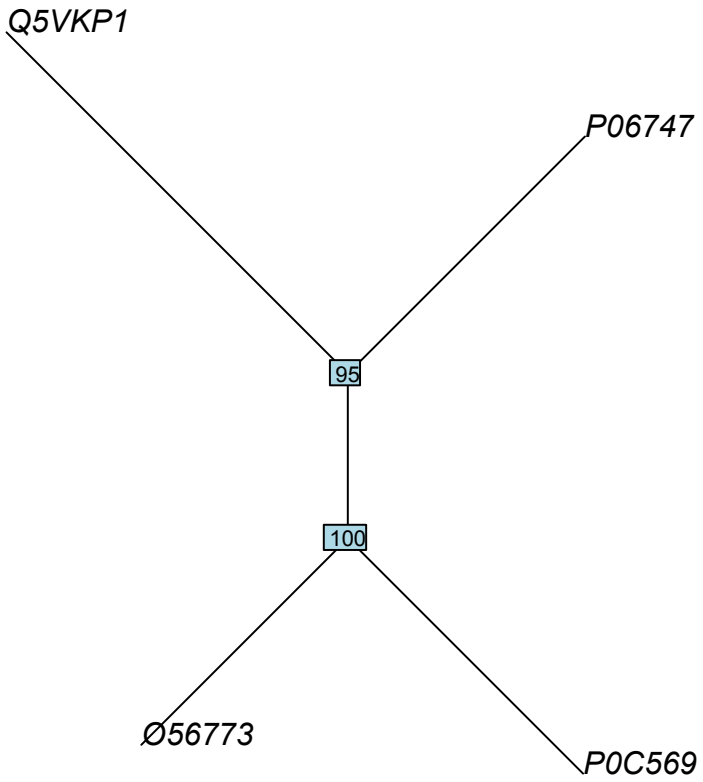
Una volta che abbiamo una matrice che dà le distanze a coppie tra tutte le nostre sequenze proteiche, possiamo costruire un albero filogenetico basato su tale matrice di distanza utilizzando il metodo dell'algoritmo *neighbour-joining* (unione dei vicini).

È possibile costruire un albero filogenetico utilizzando tale algoritmo tramite il pacchetto R *ape*. Per prima cosa è necessario installare e caricare il pacchetto *ape* mediante i comandi R:

```
> install.packages("ape")
> library(ape)
```

La funzione `unrootedNJtree()` (vedi Appendice) costruisce un albero filogenetico basato su un allineamento di sequenze, usando l'algoritmo *neighbour-joining* mediante le funzioni del pacchetto *ape*. La funzione prende un allineamento di sequenze in input, calcola le distanze a coppie tra le sequenze in base all'allineamento e poi costruisce un albero filogenetico basato sulle distanze a coppie, restituendo l'albero filogenetico e anche una sua rappresentazione grafica:

```
> virusalntree <- unrootedNJtree(virusaln,type="protein")
Running bootstraps:       100 / 100
Calculating bootstrap values... done.
```



Si noti che è necessario specificare che si stanno utilizzando sequenze di proteine impostando il parametro "`type=protein`".

Possiamo vedere che sono state raggruppate nell'albero le proteine Q5VKP1 (la fosfoproteina del virus del pipistrello del Caucaso occidentale) e P06747 (la fosfoproteina del virus della rabbia) e le proteine O56773 (la fosfoproteina del virus del pipistrello del Lagos) e P0C569 (la fosfoproteina del virus Mokola). Ciò è coerente con quello che abbiamo visto prima nella matrice delle distanze genetiche, dove la distanza genetica tra la fosfoproteina del virus del pipistrello del Lagos (O56773) e la fosfoproteina del virus Mokola (P0C569) è relativamente piccola.

I numeri nelle caselle azzurre al centro dell'albero sono valori di *bootstrap* per i nodi dell'albero. Un valore di bootstrap per un particolare nodo dell'albero dà un'idea della "confidenza" che abbiamo nel gruppo definito da quel nodo dell'albero. Se un nodo ha un valore di bootstrap alto (vicino al 100%) allora siamo molto fiduciosi che il gruppo definito dal nodo sia corretto, mentre se ha un valore di bootstrap basso (vicino allo 0%) allora non siamo altrettanto confidenti.

Si noti che il fatto che un valore di bootstrap per un nodo sia alto non garantisce necessariamente che il gruppo definito dal nodo sia corretto, ma ci dice solo che è abbastanza probabile che lo sia.

I valori di bootstrap sono calcolati facendo molti (ad es. 100) "ricampionamenti" casuali dell'allineamento su cui si è basato l'albero filogenetico. Ogni "campione" consiste in un certo numero *x* (ad esempio 200) di colonne campionate in modo casuale dall'allineamento e forma una sorta di finto allineamento a sé stante su cui basare l'albero filogenetico. Possiamo fare 100 ricampionamenti casuali dell'allineamento e costruire 100 alberi filogenetici basati sui 100 campioni. Questi 100 alberi sono noti come "*alberi di bootstrap*". Per ogni raggruppamento dell'albero filogenetico originale possiamo contare in quanti dei 100 alberi di bootstrap appare. Questo è conosciuto come il "*valore di bootstrap*" del raggruppamento del nostro albero filogenetico originale.

Ad esempio, se calcoliamo 100 ricampionamenti casuali dell'allineamento delle fosfoproteine virali e costruiamo 100 alberi filogenetici sulla base di questi campioni, possiamo calcolare i valori di bootstrap per ogni raggruppamento dell'albero filogenetico delle fosfoproteine virali.

In questo caso, il valore di bootstrap per il nodo che definisce il gruppo contenente Q5VKP1 (fosfoproteina del virus del pipistrello del Caucaso occidentale) e P06747 (fosfoproteina del virus della rabbia) è del 95%, mentre il valore di bootstrap per il nodo che definisce il gruppo della fosfoproteina del virus del pipistrello del Lagos (O56773) e della fosfoproteina del virus Mokola (P0C569) è del 100%. I valori di bootstrap per ciascuno di questi raggruppamenti sono la percentuale di 100 alberi di bootstrap in cui il raggruppamento appare.

Pertanto, siamo molto fiduciosi che le fosfoproteine del virus del pipistrello del Lagos e del virus Mokola debbano essere raggruppate nell'albero; non lo siamo altrettanto per le fosfoproteine del virus del pipistrello del Caucaso occidentale e del virus della rabbia.

La lunghezza dei rami nel grafico dell'albero è proporzionale alla quantità di cambiamento evolutivo (numero stimato di mutazioni) lungo i rami.

In questo caso, i rami che portano alla fosfoproteina del virus del pipistrello del Lagos (O56773) e alla fosfoproteina del virus Mokola (P0C569) dal nodo che rappresenta il loro antenato comune sono leggermente più corti dei rami che portano al virus del pipistrello del Caucaso occidentale (Q5VKP1) e alle fosfoproteine del virus della rabbia (P06747) dal nodo che rappresenta il loro antenato comune.

Questo suggerisce che ci potrebbero essere state più mutazioni nelle fosfoproteine del virus del pipistrello del Caucaso occidentale (Q5VKP1) e del virus della rabbia (P06747) a partire dall'antenato comune, che nelle fosfoproteine del virus del pipistrello del Lagos (O56773) e della fosfoproteina del virus Mokola (P0C569) dal rispettivo antenato comune.

L'albero delle fosfoproteine virali è un albero filogenetico senza radici (*unrooted*) in quanto non contiene una sequenza fuori gruppo (*outgroup*), cioè una sequenza di una proteina che è nota per essere più lontana dalle altre proteine dell'albero di quanto non lo siano l'una dall'altra.

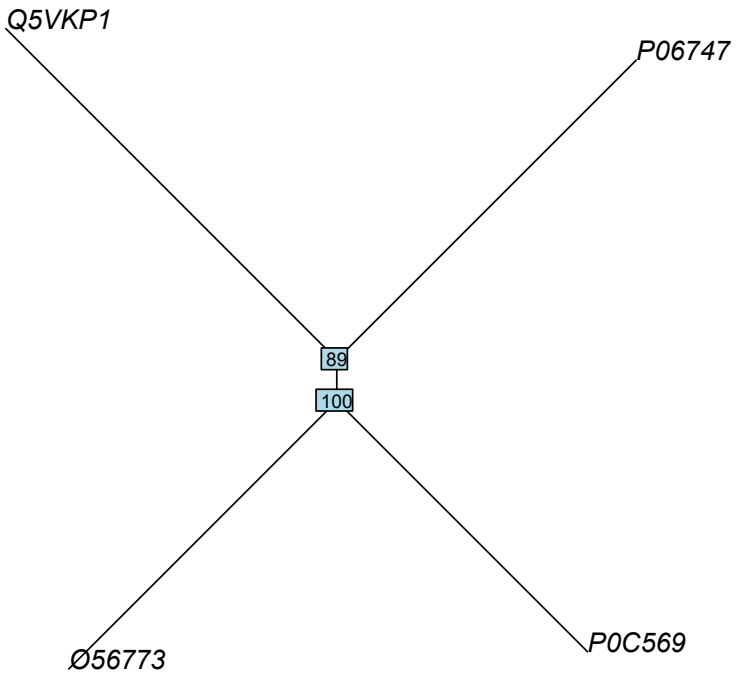
Di conseguenza, non possiamo dire in quale direzione il tempo è evoluto lungo i rami interni dell'albero. Per esempio, non possiamo dire se il nodo che rappresenta l'antenato comune di (O56773, P0C569) era un antenato del nodo che rappresenta l'antenato comune di (Q5VKP1, P06747), o viceversa.

Per costruire un albero filogenetico con radice (*rooted*), abbiamo bisogno di una sequenza di *outgroup* nel nostro albero. Nel caso delle fosfoproteine virali questo purtroppo non è possibile, in quanto (ad ora) non è nota alcuna proteina che sia più lontana dalle quattro proteine già presenti nel nostro albero di quanto lo siano l'una dall'altra.

Tuttavia, in molti altri casi è noto un *outgroup* — una sequenza nota per essere più lontanamente relazionata alle altre sequenze dell'albero di quanto non lo siano l'una all'altra — e quindi è possibile costruire un albero filogenetico *rooted*.

Abbiamo visto in precedenza che è una buona idea indagare se l'eliminazione delle regioni mal conservate di un allineamento multiplo abbia un effetto sull'analisi filogenetica. In precedenza abbiamo fatto una copia filtrata dell'allineamento multiplo e l'abbiamo memorizzata nella variabile *cleanvirusaln*. Possiamo costruire ora un albero filogenetico basato su questo allineamento filtrato e vedere se concorda con l'albero filogenetico basato sull'allineamento originale:

```
> cleanedvirusalntree <- unrootedNJtree(cleanedvirusaln,type="protein")
```



Qui le proteine (O56773, P0C569) e (Q5VKP1, P06747) sono raggruppate insieme, come nell'albero filogenetico basato sull'allineamento multiplo grezzo (non filtrato) di prima. Quindi, il filtro sull'allineamento multiplo non ha effetto sull'albero.

Se avessimo trovato una differenza negli alberi realizzati con gli allineamenti multipli non filtrati e filtrati, avremmo dovuto esaminare gli allineamenti multipli da vicino per vedere se l'allineamento multiplo non

filtrato contiene molte regioni molto poco allineate che potrebbero aggiungere rumore all'analisi filogenetica (se questo fosse vero, l'albero basato sull'allineamento filtrato sarebbe probabilmente più affidabile).

5.11 Costruire un albero filogenetico *rooted* per sequenze proteiche

Per convertire l'albero non radicato (*unrooted tree*) in un albero con radice (*rooted tree*), dobbiamo aggiungere una sequenza di *outgroup*. Normalmente, la sequenza di outgroup è una sequenza che sappiamo da alcune conoscenze precedenti essere più lontana dalle altre sequenze in studio quanto non lo siano l'una sequenza dall'altra.

Ad esempio, la proteina Fox-1 è coinvolta nella determinazione del sesso di un embrione nel verme nematode *Caenorhabditis elegans* (*accession* UniProt Q10572). Proteine correlate si trovano in altri nematodi, tra cui *Caenorhabditis remanei* (UniProt E3M2K8), *Caenorhabditis briggsae* (A8WS01), *Loa loa* (E1FUV2) e *Brugia malayi* (UniProt A8NSK3).

Si noti che *Caenorhabditis elegans* è un organismo modello comunemente studiato in biologia molecolare. I nematodi *Loa loa* e *Brugia malayi* sono nematodi parassiti che causano la filariosi.

Il database UniProt contiene una sequenza lontanamente correlata con la *Drosophila melanogaster* (mosca della frutta; *accession* UniProt Q9VT99). Se dovessimo costruire un albero filogenetico degli omologhi del verme nematode Fox-1, la sequenza lontanamente correlata dalla *Drosophila melanogaster* sarebbe probabilmente una buona scelta di *outgroup*, poiché la proteina proviene da un gruppo animale diverso (insetti) rispetto ai vermi nematodi. Pertanto, è probabile che la proteina della mosca della frutta sia più lontanamente correlata con tutte le proteine dei nematodi di quanto non lo siano l'una con l'altra.

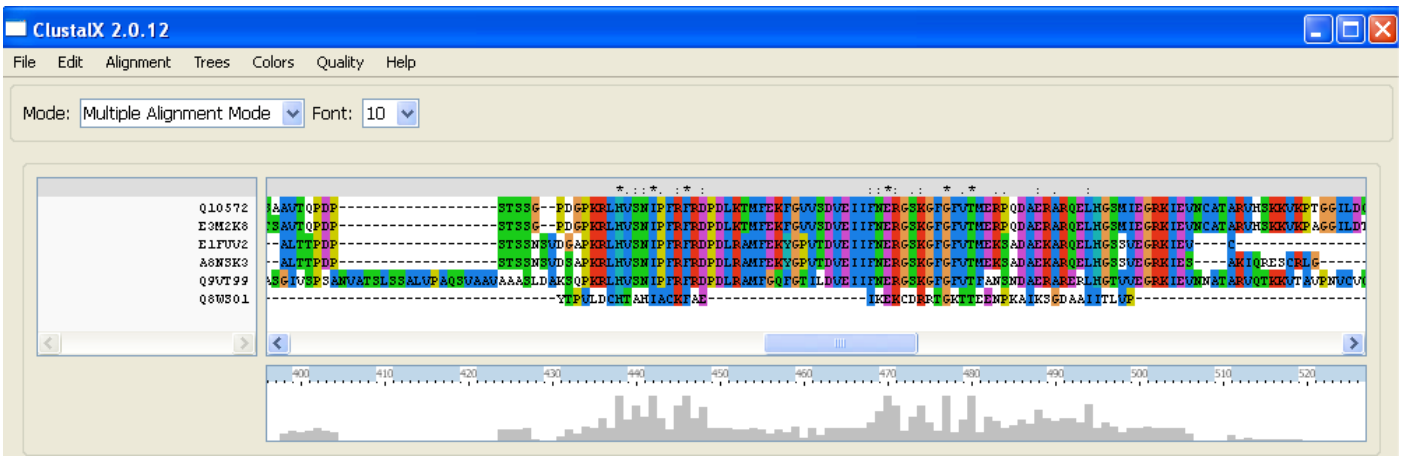
Per recuperare le sequenze da UniProt possiamo usare la funzione `retrieventrezseqs()`:

```
> seqnames <- c("Q10572","E3M2K8","Q8WS01","E1FUV2","A8NSK3","Q9VT99")
> seqs <- retrieventrezseqs(seqnames,"protein")
[1] "Retrieving sequence Q10572 ..."
[1] "Retrieving sequence E3M2K8 ..."
[1] "Retrieving sequence Q8WS01 ..."
[1] "Retrieving sequence E1FUV2 ..."
[1] "Retrieving sequence A8NSK3 ..."
[1] "Retrieving sequence Q9VT99 ..."
```

Possiamo quindi scrivere le sequenze in un file FASTA:

```
> seqinr::write.fasta(seqs, seqnames, file="fox1.fasta")
```

Usiamo ora CLUSTAL per creare un allineamento in formato PHYLIP delle sequenze e memorizzarlo nel file "fox1.phy". Questa schermata mostra una parte del lungo allineamento:



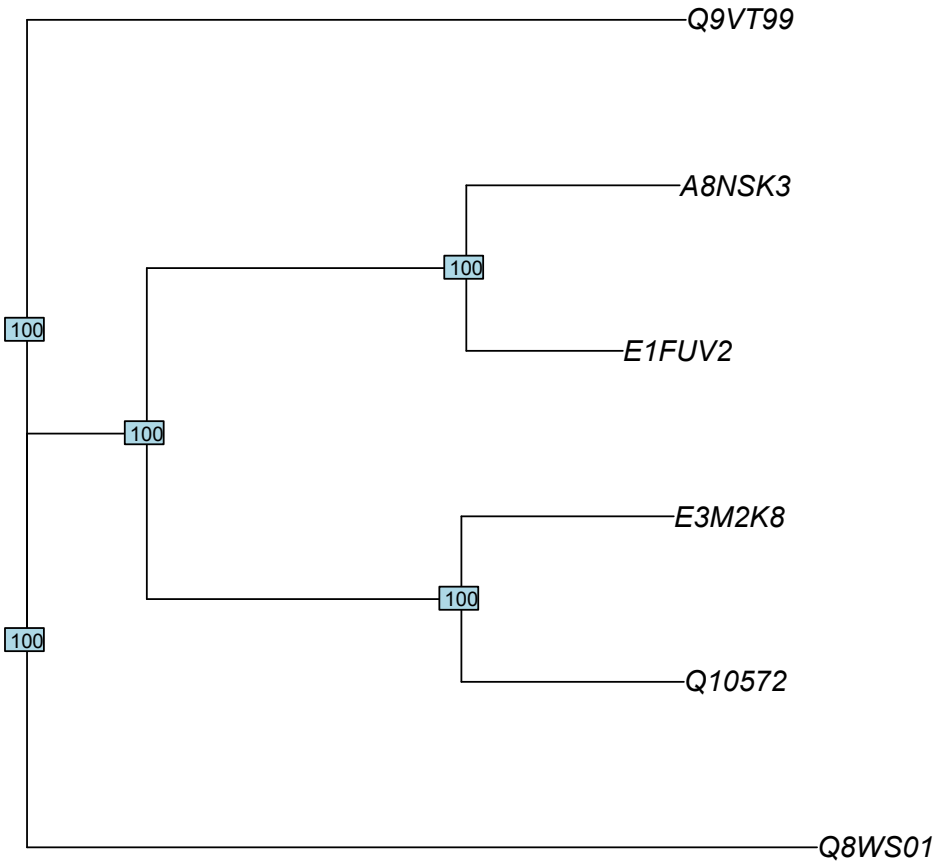
Leggiamo infine l'allineamento in R:

```
> fox1aln <- seqinr::read.alignment(file = "fox1.phy", format = "phylip")
```

Il passo successivo è quello di costruire un albero filogenetico delle proteine, che anche in questo caso possiamo fare usando l'algoritmo *neighbour-joining*.

Questa volta abbiamo un *outgroup* nel nostro set di sequenze, in modo da poter costruire un albero con radice. La funzione `rootedNJtree()` (vedi Appendice) può essere usata per costruire un albero *rooted*. Essa restituisce l'albero filogenetico, e ne fa anche un grafico; prende come input l'allineamento e il nome dell'*outgroup*. Ad esempio, per realizzare un albero filogenetico della proteina Fox-1 *C. elegans* e dei suoi omologhi, utilizzando come *outgroup* la proteina della mosca della frutta (UniProt Q9VT99), digitiamo:

```
> fox1alntree <- rootedNJtree(fox1aln, "Q9VT99",type="protein")
```



Qui possiamo vedere che E3M2K8 (omologo Fox-1 *C. remanei*) e Q10572 (Fox-1 *C. elegans*) sono stati raggruppati con bootstrap 100%, così come A8NSK3 (omologo Fox-1 *Brugia malayi*) e E1FUV2 (omologo Fox-1 *Loa loa*). Queste quattro proteine sono state anche unite in un gruppo più grande con bootstrap 100%.

Rispetto a queste quattro proteine, Q8WS01 (omologo Fox-1 *C. briggsae*) e Q9VT99 (*outgroup* mosca della frutta) sembrano essere relativamente lontani.

Trattandosi di un albero *rooted*, conosciamo la direzione evolutiva nel tempo. Supponiamo di chiamare l'antenato delle quattro sequenze (E3M2K8, Q10572, A8NSK3, E1FUV2) *antenato1*, l'antenato delle due sequenze (E3M2K8, Q10572) *antenato2* e l'antenato delle due sequenze (A8NSK3, E1FUV2) *antenato3*.

Poiché si tratta di un albero con radice, sappiamo che il tempo scorre da sinistra verso destra lungo i rami dell'albero, per cui *antenato1* era l'antenato di *antenato2* e *antenato3*. In altre parole, *antenato1* viveva prima di *antenato2* o *antenato3*, quindi *antenato2* e *antenato3* sono discendenti di *antenato1*.

Si può anche dire che E3M2K8 e Q10572 hanno condiviso un antenato comune più recentemente rispetto a A8NSK3 e E1FUV2.

Le lunghezze dei rami di questo albero sono proporzionali alla quantità di cambiamento evolutivo (numero stimato di mutazioni) che si è verificato lungo i rami. I rami che portano indietro da E3M2K8 e Q10572 al loro ultimo antenato comune sono leggermente più lunghi dei rami che portano indietro da A8NSK3 e E1FUV2 al loro ultimo antenato comune.

Ciò indica che c'è stato un cambiamento evolutivo maggiore nelle proteine E3M2K8 (omologo Fox-1 *C. remanei*) e Q10572 (Fox-1 *C. elegans*) da quando si sono allontanate, rispetto alle proteine A8NSK3 (omologo Fox-1 *Brugia malayi*) e E1FUV2 (omologo Fox-1 *Loa loa*) da quando si sono allontanate.

5.12 Costruire un albero filogenetico per sequenze di DNA o mRNA

Nell'esempio sopra riportato, è stato costruito un albero filogenetico per sequenze proteiche. I genomi di organismi imparentati alla lontana, come i vertebrati, avranno accumulato molte mutazioni da quando si sono allontanati. A volte, si sono verificate così tante mutazioni da quando gli organismi si sono allontanati che le loro sequenze di DNA sono difficili da allineare correttamente ed è anche difficile stimare con precisione le distanze evolutive dagli allineamenti di quelle sequenze di DNA.

Al contrario, poiché molte mutazioni a livello di DNA sono sinonimi a livello di proteine, le sequenze di proteine divergono ad un ritmo più lento rispetto alle sequenze di DNA. Questo è il motivo per cui per organismi ragionevolmente correlati, come i vertebrati, è di solito preferibile utilizzare le sequenze proteiche per le analisi filogenetiche.

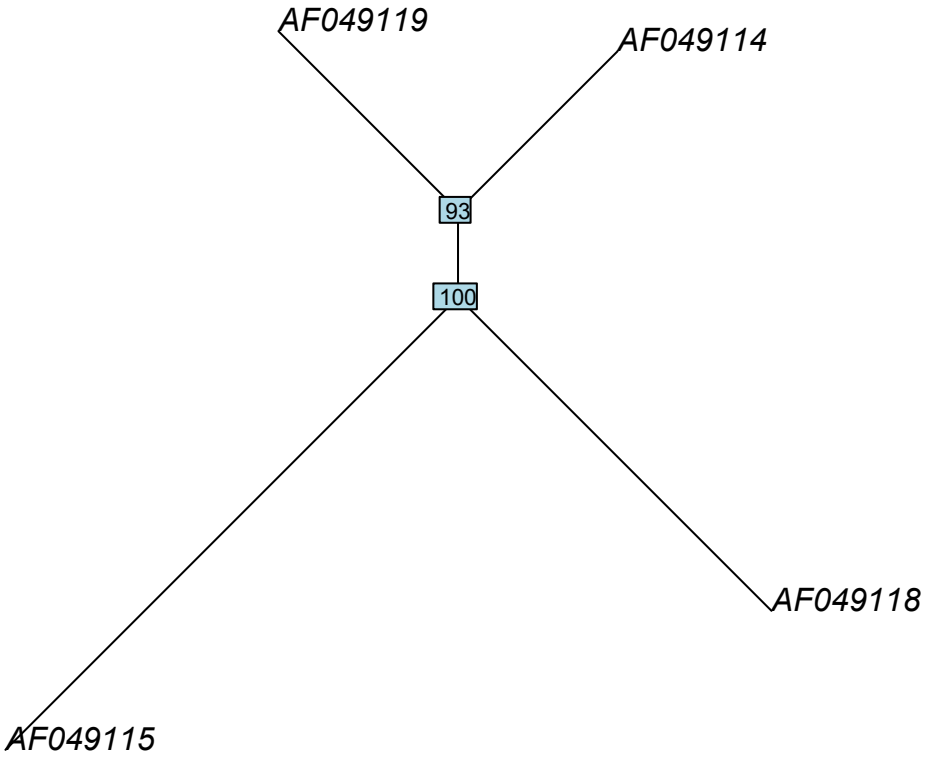
Se si studiano organismi strettamente imparentati come i primati, da quando si sono allontanati si sono verificate poche mutazioni. Di conseguenza, se si utilizzano le sequenze proteiche per un'analisi filogenetica, ci possono essere troppo poche sostituzioni di aminoacidi per fornire un "segnale" sufficiente da utilizzare per l'analisi filogenetica. Pertanto, è spesso preferibile utilizzare le sequenze di DNA per un'analisi filogenetica di organismi strettamente correlati come i primati.

Possiamo utilizzare le funzioni `unrootedNJtree()` e `rootedNJtree()` descritte sopra per costruire alberi filogenetici senza radice o con radici vicine che si uniscono in base ad un allineamento delle sequenze di DNA o mRNA. In questo caso, dobbiamo usare "`type=DNA`" come argomento in queste funzioni per dire loro che stiamo costruendo un albero di sequenze nucleotidiche, non di proteine.

Ad esempio, per costruire un albero filogenetico *unrooted* basato sull'allineamento delle sequenze di mRNA delle fosfoproteine virali, digitiamo in R:

```
> virusmRNAaln  <- read.alignment(file = "virusmRNA.phy", format = "phylip")
> virusmRNAalntree <- unrootedNJtree(virusmRNAaln, type="DNA")
```

ottenendo il seguente albero filogenetico:



5.13 Esempi di uso del pacchetto *ape* per la creazione di alberi filogenetici

Vediamo ora alcuni esempi applicativi di analisi filogenetica con il pacchetto *ape*. Iniziamo con l'installazione e il caricamento del pacchetto digitando i seguenti comandi:

```
> install.packages("ape")
> library(ape)
```

Definiamo le sequenze di nostro interesse con i loro ID e recuperiamole dal dataset *GenBank*, memorizzandole in una variabile *mySeqs* di tipo DNAbin:

```
> mySet <- c("U15717", "U15718", "U15719", "U15720", "U15721", "U15722", "U15723",
"U15724")
> mySeqs <- read.GenBank(mySet)
> mySeqs
8 DNA sequences in binary format stored in a list.
All sequences of same length: 1045
Labels: U15717 U15718 U15719 U15720 U15721 U15722 ...
Base composition:
    a     c     g     t
0.267 0.351 0.134 0.247
(Total: 8.36 kb)
```

Calcoliamo la matrice di distanza per le sequenze mediante la funzione `dist.dna()`:

```
> myDist <- dist.dna(mySeqs)
> myDist
             U15717       U15718       U15719       U15720       U15721       U15722       U15723
U15718 0.0963968720
U15719 0.0519601191 0.0821667377
U15720 0.0155113115 0.0932071967 0.0489261666
U15721 0.0624466459 0.0797534702 0.0498979851 0.0551246867
U15722 0.0164965334 0.0920721224 0.0478790458 0.0009578547 0.0540676038
U15723 0.0699980300 0.0885445872 0.0647927035 0.0636627810 0.0396102719 0.0625875118
U15724 0.0722326361 0.0842641651 0.0562718695 0.0615634135 0.0397830924 0.0604916017 0.0417080298
```

Creiamo l'albero filogenetico (oggetto di tipo `phylo`) mediante la funzione `nj()` del pacchetto *ape* che utilizza il metodo *Neighbor-Joining Tree Estimation* per ricostruire l'albero:

```
> myPhylo <- nj(myDist)
> myPhylo
Phylogenetic tree with 8 tips and 6 internal nodes.
Tip labels:
      U15717, U15718, U15719, U15720, U15721, U15722, ...
Unrooted; includes branch lengths.
```

Effettuiamo ora un *bootstrap* sull'oggetto *myPhylo* utilizzando la funzione `boot.phylo()`:

```
> fun <- function(xx) nj(dist.dna(xx))
> bootmat = matrix(nrow=100, ncol=6, byrow=TRUE)
> for (i in 1:100) bootmat[i,] <- (boot.phylo(myPhylo, as.matrix(mySeqs), fun,
      quiet = TRUE))
> boot <- numeric(6)
> for (i in 1:6) boot[i] <- mean(bootmat[,i])
> boot
[1] 100.00  66.54  91.83  93.33 100.00  96.29
```

Creiamo e visualizziamo (vedi pagina seguente) i diversi tipi di alberi filogenetici per l'analisi digitando:

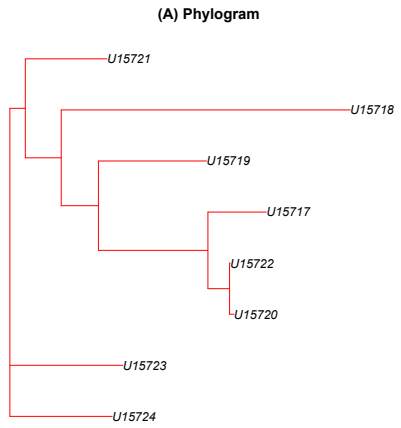
```
> plot(myPhylo, type="phylogram", edge.color="red", cex=1, edge.width=1,
      main="(A) Phylogram")
> plot(myPhylo, type="cladogram", edge.color="red", cex=1, edge.width=1,
      main="(B) Cladogram")
> plot(myPhylo, type="fan", edge.color="red", cex=1, edge.width=1, main="(C) Fan")
> plot(myPhylo, type="unrooted", edge.color="red", cex=1, edge.width=1,
      main="(D) Unrooted")
```

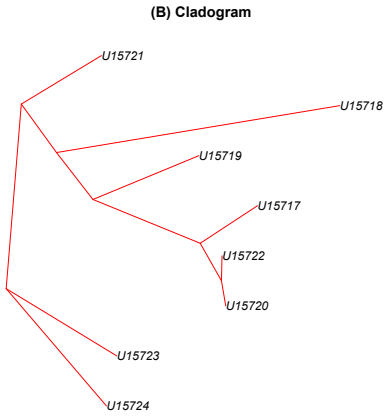
Tracciamo infine il grafico dell'albero filogenetico con i valori di bootstrap calcolati in precedenza:

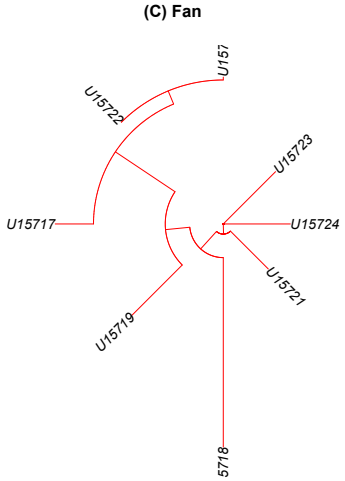
```
> layout(1)
> par(mar = rep(2, 4))
> plot(myPhylo, main = "Phylogenetic tree")
> nodelabels(boot)
```

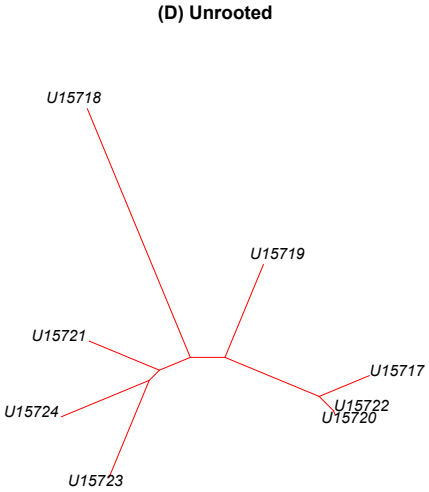
Riassumiamo per grandi linee il processo fin qui seguito. Una volta che le sequenze vengono recuperate dal database che le contiene, inizia il processo di analisi filogenetica.

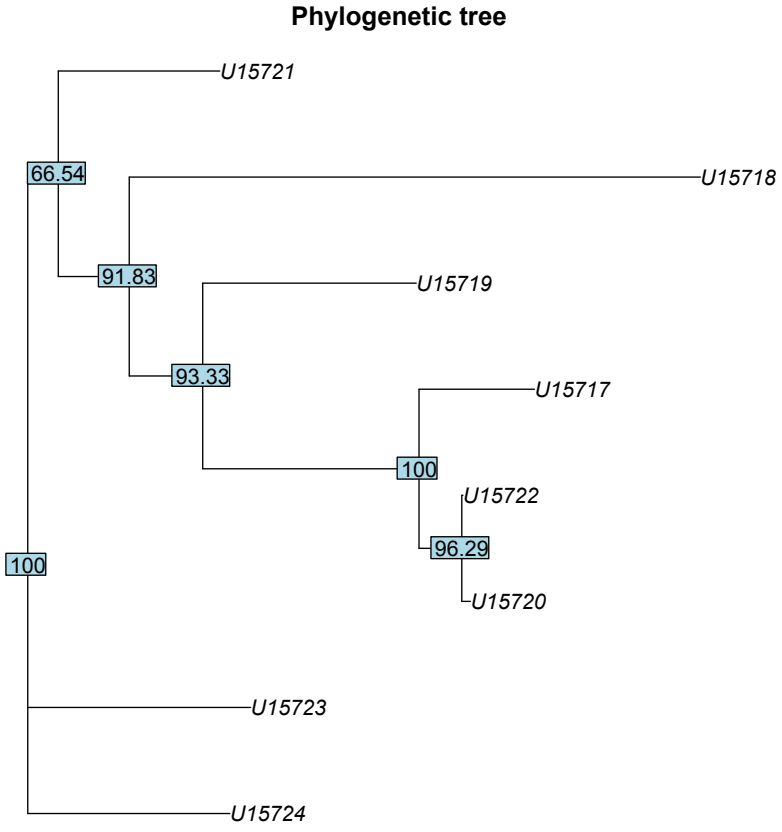
Il primo passo è il calcolo delle distanze a coppie tra le sequenze. Questo si basa su diversi modelli di evoluzione del DNA: ad esempio il modello di Kimura, il modello di Jukes-Cantor e così via. Le distanze si basano sulla "velocità" con la quale un nucleotide sostituisce un altro durante l'evoluzione, e sono state calcolate mediante a funzione `ape::dist.dna()`. Questa funzione produce un oggetto della classe "dist"; per ottenere invece il risultato sotto forma di matrice impostare l'argomento `as.matrix` a `TRUE`. L'oggetto "dist" viene poi convertito in un oggetto "phylo" che può essere usato per tracciare alberi filogenetici. Queste distanze rappresentano quanto sono lontane le specie, o meglio, le sequenze sulla scala evolutiva. Abbiamo utilizzato il metodo *Neighbor-Joining* per creare l'albero filogenetico, che unisce i due ceppi più vicini in modo che siano il più lontano possibile dai rimanenti, ed abbiamo anche eseguito un bootstrap di controllo su 100 alberi casuali.


(A) Phylogram
U15721
U15718
U15719
U15717
U15722
U15720
U15723
U15724


(B) Cladogram
U15721
U15718
U15719
U15717
U15722
U15720
U15723
U15724


(C) Fan
U15722
U15717
U15723
U15724
U15721
U15719
5718


(D) Unrooted
U15718
U15719
U15721
U15717
U15724
U15722
U15720
U15723


Phylogenetic tree
U15721
66.54
U15718
91.83
U15719
93.33
U15717
100
U15722
96.29
100
U15720
U15723
U15724

Abbiamo anche visto quattro modi diversi per tracciare il grafico dell'albero filogenetico. Il "filogramma" (*phylogram*) mostra la storia evolutiva in cui la lunghezza di ogni tronco di ramo indica la quantità di evoluzione (come il numero di sostituzioni nucleotidiche avvenute tra i punti di ramo collegati). Il secondo tipo di albero che abbiamo tracciato è un "cladogramma" (*cladogram*): esso non rappresenta la relazione effettiva tra le specie, ma le ramificazioni durante l'evoluzione e i rami si uniscono a ipotetici antenati. Il terzo tipo di albero è un albero "a ventaglio" (*fan*), che mostra i rami radicali per le specie. L'ultimo tipo è l'albero "non radicato" (*unrooted*), che illustra la correlazione dei nodi foglia (terminali), ignorando completamente l'ascendenza.

Durante l'esecuzione di un'analisi filogenetica, è importante tenere presente la domanda biologica a cui si cerca di rispondere. Ad esempio, quando si studiano specie strettamente correlate a livello proteico, i segnali saranno molto piccoli, poiché le proteine evolvono ad un ritmo molto più lento: è quindi consigliabile utilizzare sequenze nucleotidiche per tale analisi. Al contrario, per le specie lontane, il segnale del DNA è troppo "rumoroso" e quindi si preferiscono le sequenze proteiche.

Infatti, le sequenze nucleotidiche di una coppia di geni omologhi hanno un contenuto di informazioni come mutazioni più elevato rispetto alle corrispondenti sequenze di aminoacidi, poiché i cambiamenti non sinonimi che interessano il DNA potrebbero non influenzare la sequenza di aminoacidi.

Osserviamo infine che un'altra importante caratteristica del pacchetto *ape* è la possibilità di salvare l'albero filogenetico come file in formato Newick usando la funzione `write.tree()`:

```
> write.tree(myPhylo, "myPhylogeneticTree.tre")
```

6. Ricerca genica (*gene-finding*) computazionale

6.1 Il codice genetico

Un gene che codifica le proteine inizia con una sequenza "ATG", seguita da un numero intero di *codoni* (triplette di DNA) che codificano gli aminoacidi, e termina con "TGA", "TAA" o "TAG". Cioè, il codone di partenza di un gene è sempre "ATG", mentre il codone di arresto di un gene può essere "TGA", "TAA" o "TAG".

In R, è possibile visualizzare il codice genetico standard, la corrispondenza tra i codoni e gli aminoacidi in cui sono tradotti, utilizzando la funzione `tablecode()` del pacchetto *SeqinR*:

```
> tablecode()
```

Genetic code 1 : standard

T T T	Phe	T C T	Ser	T A T	Tyr	T G T	Cys
T T C	Phe	T C C	Ser	T A C	Tyr	T G C	Cys
T T A	Leu	T C A	Ser	T A A	Stp	T G A	Stp
T T G	Leu	T C G	Ser	T A G	Stp	T G G	Trp
C T T	Leu	C C T	Pro	C A T	His	C G T	Arg
C T C	Leu	C C C	Pro	C A C	His	C G C	Arg
C T A	Leu	C C A	Pro	C A A	Gln	C G A	Arg
C T G	Leu	C C G	Pro	C A G	Gln	C G G	Arg
A T T	Ile	A C T	Thr	A A T	Asn	A G T	Ser
A T C	Ile	A C C	Thr	A A C	Asn	A G C	Ser
A T A	Ile	A C A	Thr	A A A	Lys	A G A	Arg
A T G	Met	A C G	Thr	A A G	Lys	A G G	Arg
G T T	Val	G C T	Ala	G A T	Asp	G G T	Gly
G T C	Val	G C C	Ala	G A C	Asp	G G C	Gly
G T A	Val	G C A	Ala	G A A	Glu	G G A	Gly
G T G	Val	G C G	Ala	G A G	Glu	G G G	Gly

Da questa tabella si può vedere che "ATG" è tradotto in Met (l'aminoacido metionina), e che "TAA", "TGA" e "TAG" corrispondono a Stp (codici di stop, che non sono tradotti in nessun aminoacido, ma segnalano la fine della traslazione).

6.2 Ricerca dei codoni di inizio e fine in una sequenza di DNA

Per cercare tutti i potenziali codoni di inizio (*start*) e di fine (*stop*) in una sequenza di DNA, dobbiamo trovare tutte le triplette "ATG", "TGA", "TAA" e "TAG" nella sequenza.

Per fare questo, possiamo usare la funzione `matchPattern()` del pacchetto *Biostrings* di Bioconductor, che identifica tutte le ricorrenze di un particolare motivo (ad es. "ATG") in una sequenza. Come input, la funzione richiede che le sequenze siano sotto forma di una stringa di caratteri.

Ad esempio, possiamo cercare tutti gli "ATG" nella sequenza "AAAATGCAGTAACCCATGCCC" digitando:

```
> library("Biostrings")
> s1 <- "aaaatgcagtaacccatgccc"
> matchPattern("atg", s1) # Find all ATGs in the sequence s1
  Views on a 21-letter BString subject
subject: aaaatgcagtaacccatgccc
views:
    start end width
[1]     4   6     3 [atg]
[2]    16  18     3 [atg]
```

L'output di `matchPattern()` dice che ci sono due triplette "ATG" nella sequenza, ai nucleotidi 4–6 e 16–18. In effetti, basta esaminare la sequenza "AAAATGCAGTAACCCATGCCC".

Allo stesso modo, se si usa `matchPattern()` per trovare le posizioni di "TAA", "TGA" e "TAG" nella sequenza "AAAATGCAGTAACCCATGCCC", si troverà che vi è un "TAA" ai nucleotidi 10–12, ma nessun "TAG" o "TGA".

La funzione `findPotentialStartsAndStops()` (vedi Appendice) può essere usata per trovare tutti i potenziali codoni di *start* e *stop* in una sequenza di DNA. Ad esempio, possiamo utilizzare questa funzione per trovare potenziali codoni di *start* e *stop* nella sequenza *s1*:

```
> s1 <- "aaaatgcagtaacccatgccc"
> findPotentialStartsAndStops(s1)
[[1]]
[1]  4 10 16

[[2]]
[1] "atg" "taa" "atg"
```

Il risultato della funzione viene restituito come variabile lista che contiene due elementi: il primo è un vettore contenente le posizioni dei potenziali codoni di *start* e *stop* nella sequenza di input, e il secondo è un vettore contenente il tipo dei codoni di *start*/*stop* ("atg", "taa", "tag", o "tga").

L'output ci dice che la sequenza *s1* ha un "ATG" a partire dal nucleotide 4, un "TAA" a partire dal nucleotide 10, e un altro "ATG" a partire dal nucleotide 16.

Possiamo usare la funzione `findPotentialStartsAndStops()` per trovare tutti i potenziali codoni di *start* e *stop* in sequenze più lunghe. Ad esempio, supponiamo di voler trovare tutti i potenziali codoni di *start*/*stop* nei primi 500 nucleotidi della sequenza genomica del virus Dengue DEN-1 (*accession* NCBI NC_001477).

In precedenza abbamo visto come recuperare una sequenza per un *accession* NCBI utilizzando la funzione `retrieventrezseqs()`. Così, per recuperare la sequenza del genoma del virus Dengue DEN-1 (*accession* NCBI NC_001477), possiamo digitare:

```
> seqs = retrieventrezseqs(list("NC_001477"), db = "nucleotide")
> dengueseq = seqs[[1]]
```

La variabile *dengueseq* è un vettore, e ogni lettera della sequenza di DNA del virus Dengue DEN-1 è memorizzata in un elemento del vettore.

Possiamo prendere i primi 500 nucleotidi della sequenza del virus DEN-1 dai primi 500 elementi di questo vettore:

```
> # Take the first 500 nucleotides of the DEN-1 Dengue sequence
> denguseqstart <- dengueseq[1:500]
> # Find the length of the "denguseqstart" start vector
> length(denguseqstart)
[1] 500
```

Ora vogliamo trovare i potenziali codoni di *start*/*stop* nei primi 500 nucleotidi della sequenza del virus Dengue. Possiamo farlo usando la funzione `findPotentialStartsAndStops()` descritta sopra. Tuttavia, la funzione richiede che la sequenza di input sia nel formato di una stringa di caratteri, piuttosto che di un vettore. Pertanto, dobbiamo prima convertire il vettore *dengueseqstart* in una stringa di caratteri. Possiamo farlo usando la funzione `c2s()` del pacchetto *seqinr*:

```
> # Convert the vector "denguseqstart" to a string of characters
> denguseqstartstring <- c2s(denguseqstart)
> # Print out the variable string of characters "denguseqstartstring"
> denguseqstartstring
[1]
"AGTTGTTAGTCTACGTGGACCGACAAGAACAGTTTCGAATCGGAAGCTTGCTTAACGTAGTTCTAACAGTTTTTTATTAGAGAG
CAGATCTCTGATGAACAACCAACGGAAAAAGACGGGTCGACCGTCTTTCAATATGCTGAAACGCGCGAGAAACCGCGTGTCAACT
GTTTCACAGTTGGCGAAGAGATTCTCAAAAGGATTGCTTTCAGGCCAAGGACCCATGAAATTGGTGATGGCTTTTATAGCATTCC
TAAGATTTCTAGCCATACCTCCAACAGCAGGAATTTTGGCTAGATGGGGCTCATTCAAGAAGAATGGAGCGATCAAAGTGTTACG
GGGTTTCAAGAAAGAAATCTCAAACATGTTGAACATAATGAACAGGAGGAAAAGATCTGTGACCATGCTCCTCATGCTGCTGCCC
ACAGCCCTGGCGTTCCATCTGACCACCCGAGGGGGAGAGCCGCACATGATAGTTAGCAAGCAGGAAAGAGGAAAAT"
```

Possiamo quindi trovare i potenziali codoni di *start*/*stop* nei primi 500 nucleotidi della sequenza del virus DEN-1 digitando:

```
> findPotentialStartsAndStops(denguseqstartstring)
[[1]]
 [1]   7  53  58  64  78  93  95  96 137 141 224 225 234 236 246 255 264 295 298 318 365 369
[23] 375 377 378 399 404 413 444 470 471 474 478
[[2]]
 [1] "TAG" "TAA" "TAG" "TAA" "TAG" "TGA" "ATG" "TGA" "ATG" "TGA" "ATG" "TGA" "TGA" "ATG"
[15] "TAG" "TAA" "TAG" "TAG" "ATG" "ATG" "ATG" "TGA" "TAA" "ATG" "TGA" "TGA" "ATG" "ATG"
[29] "TGA" "ATG" "TGA" "TAG" "TAG"
```

Vediamo che la sequenza ha molti diversi potenziali codoni di start e di stop; ad esempio, un potenziale codone di arresto (TAG) al nucleotide 7, un potenziale codone di arresto (TAA) al nucleotide 53, un potenziale codone di arresto (TAG) al nucleotide 58, e così via.

6.3 *Reading frame* (finestra di lettura)

I potenziali codoni di *start*/*stop* in una sequenza di DNA possono essere in tre diversi possibili *frame* (finestre) di lettura (*reading frame*). Un potenziale codone di *start*/*stop* si dice essere nel *frame* di lettura +1 se c'è un numero intero di triplette *x* tra il primo nucleotide della sequenza e l'inizio del codone di *start*/*stop*. Così, un potenziale codone che inizia con i nucleotidi 1 (0 triplette), 4 (1 tripletta), 7 (2 triplette), ... sarà nel *frame* di lettura +1.

Se c'è un numero intero di triplette *x* più un nucleotide (cioè *x*.3 triplette) tra il primo nucleotide della sequenza e l'inizio del codone di *start*/*stop*, allora il codone è nel *frame* di lettura +2. Un potenziale codone di *start*/*stop* che inizia ai nucleotidi 2 (0.3 triplette), 5 (1.3 triplette), 8 (2.3 triplette), ... si trova nel *frame* di lettura +2.

Allo stesso modo, se c'è un numero intero di triplette *x* più due nucleotidi (cioè *x*.6 triplette) tra i primi nucleotidi della sequenza e l'inizio del codone di *start*/*stop*, il codone si trova nel *frame* di lettura +3. Quindi un potenziale codone di *start*/*stop* che inizia con i nucleotidi 3 (0.6 triplette), 6 (1.6 triplette), 9 (2.6 triplette),… è nel *frame* di lettura +3.

Affinché un potenziale codone di *start*/*stop* faccia parte dello stesso gene, deve trovarsi nello stesso *frame* di lettura.

Dall'output di `findPotentialStartsAndStops()` per i primi 500 nucleotidi del genoma del virus Dengue DEN-1 si può vedere che c'è un potenziale codone di *start* (ATG) che parte dal nucleotide 137, e un potenziale codone di *stop* (TGA) che parte dal nucleotide 141. La regione individuata dai nucleotidi 137–143 potrebbe essere un gene?

Possiamo isolare la regione dal nucleotide 137 al 143 della sequenza *denguseqstartstring* per dare un'occhiata usando la funzione `substring()`. I suoi argomenti sono il nome della variabile che contiene la stringa di caratteri (cioè, la sequenza di DNA) e le coordinate della sottostringa desiderata:

```
> substring(dengueseqstartstring,137,143)
[1] "ATGCTGA"
```

Se guardiamo la sequenza dei nucleotidi 137–143, "ATGCTGA", vediamo che inizia con un potenziale codone di *start* (ATG) e finisce con un potenziale codone di *stop* (TGA).

Tuttavia, il ribosoma legge la sequenza scansionando i codoni (triplette) uno per uno da sinistra a destra, e quando scomponiamo la sequenza in codoni vediamo che non contiene un numero intero di triplette: "ATG CTG A".

Ciò significa che anche il ribosoma non riconoscerà la regione da 137–143 come gene potenziale, poiché l'ATG al nucleotide 137 non è separato dal TGA al nucleotide 141 da un numero intero di codoni. Cioè, i codoni potenziali di *start*/*stop* "ATG" e "TGA" non sono nello stesso frame di lettura, e quindi non possono essere il codone di *start* e di *stop* dello stesso gene.

Il codone potenziale di *start* al nucleotide 137 della sequenza *dengueseqstartstring* è nel *frame* di lettura +2 (infatti (137-1) %% 3 + 1 = 2) poiché c'è un numero intero di triplette, più un nucleotide, tra l'inizio della sequenza e l'inizio del codone di *start* (cioè le triplette 1–3, 4–6, 7–9, 10–12, 13–15, 16–18, 19–21, 22–24, 25–27, 28–30, ..., 133–135, e il singolo nucleotide 136).

Tuttavia, il codone potenziale di *stop* al nucleotide 141 è nel *frame* di lettura +3 (infatti (141-1) %% 3 + 1 = 3) poiché ci sono due nucleotidi più un numero intero di triplette tra l'inizio della sequenza e l'inizio del codone di *stop* (cioè le triplette 1–3, 4–6, 7–9, 10–12, 13–15, 16–18, 19–21, 22–24, 25–27, 28–30, 31–33, 34–36, 37–39, 40–42, 43–45, ..., 133–135, 136–138 e i due nucleotidi 139 e 140).

Poiché il codone potenziale di *start* al nucleotide 137 e il codone potenziale di *stop* al nucleotide 141 si trovano in diversi *frame* di lettura, non sono separati da un numero intero di codoni, e quindi non possono essere parte dello stesso gene.

6.4 Ricerca di *reading frame* aperti (ORF) sul *forward strand* di una sequenza di DNA

Per trovare i potenziali geni, dobbiamo cercare un codone di *start*, seguito da un numero intero di triplette, seguito da un codone di *stop*. Questo equivale alla ricerca di un codone di *start* seguito da un codone di *stop* che si trovi nello stesso *reading frame*. Un tale tratto di DNA è conosciuto come un *reading frame* aperto (ORF, Open Reading Frame), ed è un buon candidato per un gene potenziale.

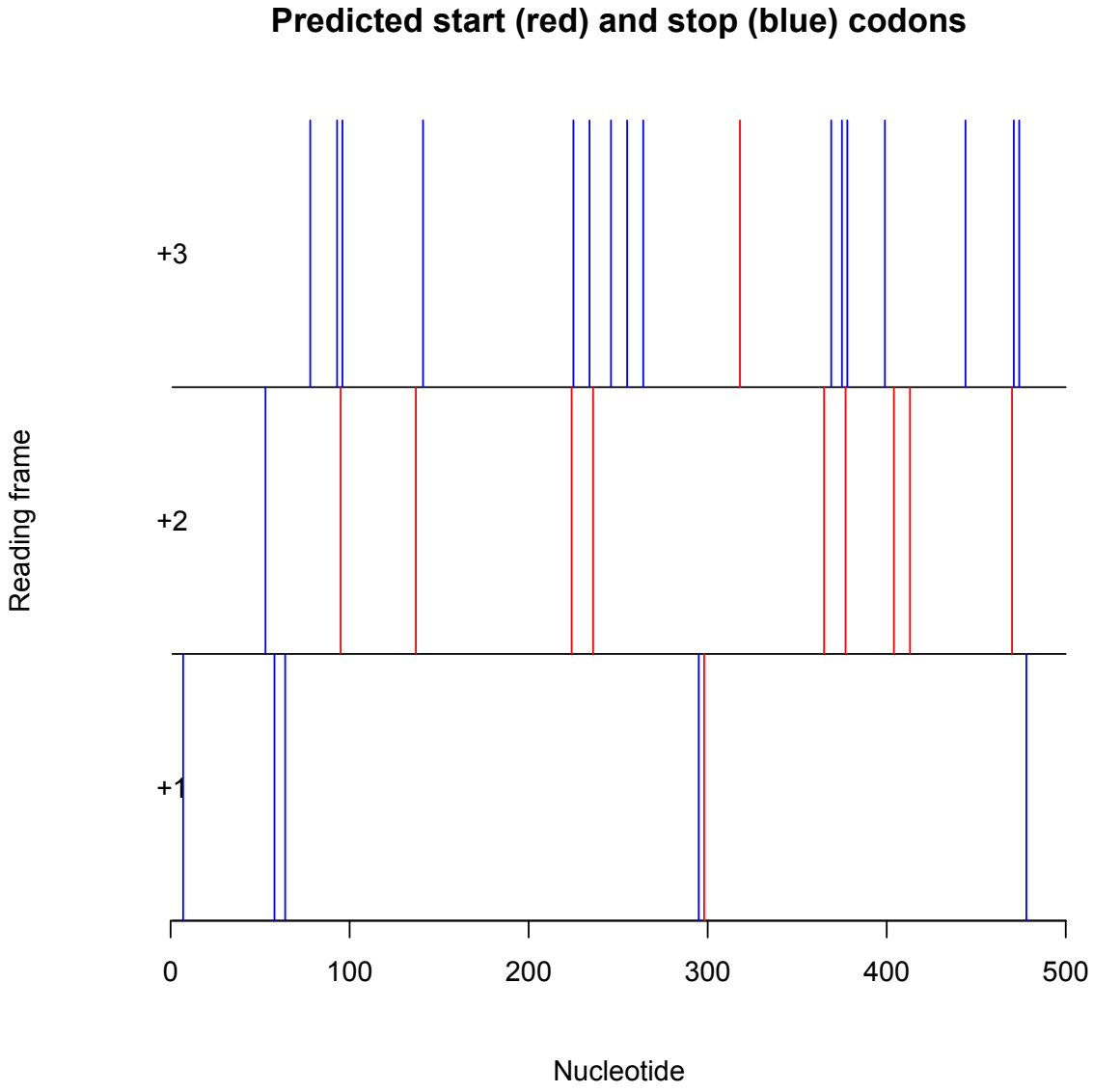
La funzione `plotPotentialStartsAndStops()` (vedi Appendice) traccia i potenziali codoni di *start* e di *stop* nei tre diversi *reading frame* di una sequenza di DNA. Ad esempio, per tracciare i codoni di *start*/*stop* nei primi 500 nucleotidi del genoma del virus Dengue DEN-1, digitiamo:

```
> plotPotentialStartsAndStops(dengueseqstartstring)
```

Nell'immagine prodotta da `plotPotentialStartsAndStops()` l'asse *x* rappresenta la sequenza di input (contenuta nella variabile *dengueseqstartstring*). I codoni potenziali di *start* sono rappresentati da linee rosse verticali, e quelli di *stop* da linee blu verticali.

I tre diversi livelli nell'immagine mostrano i codoni di *start*/*stop* nei *reading frame* +1 (livello inferiore), +2 (livello intermedio) e +3 (livello superiore).

Possiamo vedere che il codone di *start* al nucleotide 137 è rappresentato da una linea rossa verticale nello livello corrispondente al *reading frame* +2 (livello intermedio). Non ci sono potenziali codoni di *stop* nel *reading frame* +2 a destra di quel codone di *start*. Quindi, il codone di *start* al nucleotide 137 non sembra far parte di un *reading frame* aperto.



Possiamo comunque osservare che nel *reading frame* +3 (livello superiore) c'è un possibile codone di *start* (linea rossa) in posizione 318, seguito da un possibile codone di *stop* (linea blu) in posizione 371. Così, la regione dai nucleotidi 318 a 371 potrebbe essere un gene potenziale nel *reading frame* +3. In altre parole, la regione dai nucleotidi 318 a 371 è un *reading frame* aperto, cioè un ORF.

La funzione `findORFsinSeq()` (vedi Appendice) trova gli ORF in una sequenza di input. Ad esempio, possiamo usarla per trovare tutti gli ORF nella sequenza *s1*:

```
> s1 <- "aaaatgcagtaacccatgccc"
> findORFsinSeq(s1)
[[1]]
[1] 4

[[2]]
[1] 12

[[3]]
[1] 9
```

La funzione restituisce una variabile lista, dove il primo elemento è il vettore delle posizioni iniziali degli ORF, il secondo elemento è il vettore delle posizioni finali di quegli ORF, e il terzo elemento è il vettore contenente le lunghezze degli ORF.

L'output della funzione `findORFsinSeq()` ci dice che c'è un ORF nella sequenza *s1*, che il potenziale codone di *start* inizia al nucleotide 4 nella sequenza e che il potenziale codone di *stop* termina al nucleotide 12 nella sequenza.

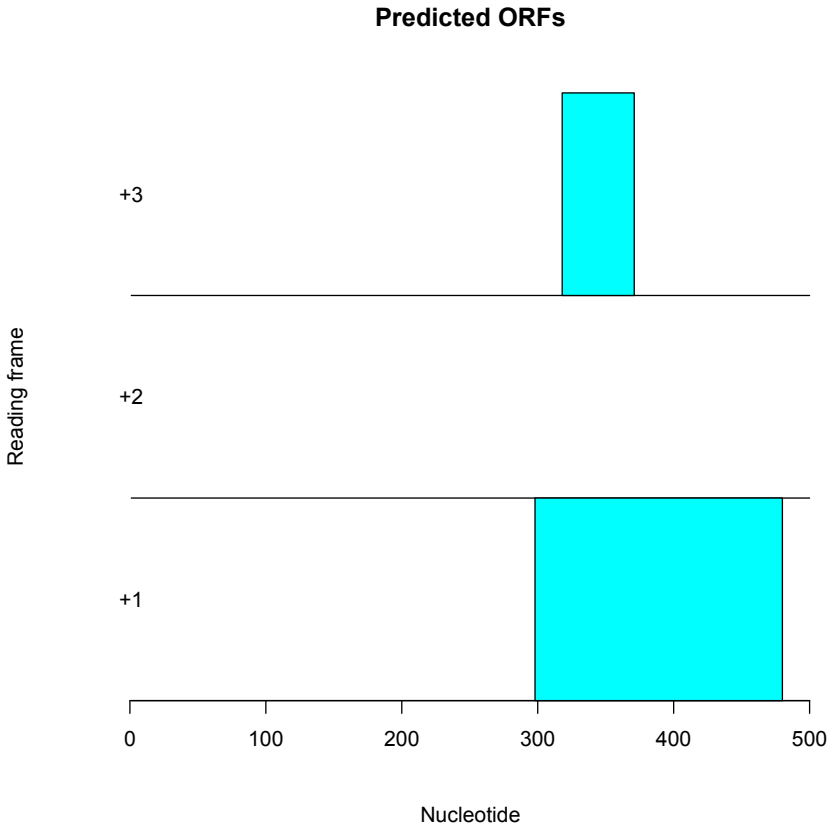
Possiamo usare la funzione `findORFsinSeq()` per trovare gli ORF nei primi 500 nucleotidi della sequenza del genoma del virus Dengue DEN-1 digitando:

```
> findORFsinSeq(denguseqstartstring)
 [[1]]
  [1] 298 318
 [[2]]
  [1] 480 371
 [[3]]
  [1] 183 54
```

Il risultato di `findORFsinSeq()` indica che ci sono due ORF nei primi 500 nucleotidi del genoma del virus, ai nucleotidi 298–480 (codone di *start* a 298–300, codone di *stop* a 478–480) e 318–371 (codone di *start* a 318–320, codone di *stop* a 369–371).

La funzione `plotORFsinSeq()` (vedi Appendice) traccia le posizioni degli ORF in una sequenza. Con questa funzione è possibile tracciare le posizioni degli ORF presenti in *dengueseqstartstring* digitando:

```
> plotORFsinSeq(denguseqstartstring)
```



L'immagine prodotta da `plotORFsinSeq()` rappresenta i due ORF nei primi 500 nucleotidi del genoma del virus DEN-1 come rettangoli azzurri.

Uno degli ORF si trova nel *reading frame* +3 e l'altro nel *reading frame* +1. Non ci sono ORF nel *reading frame* +2, in quanto non ci sono potenziali codoni di *stop* a destra (3') dei potenziali codoni di *start* nel *reading frame* +2, come possiamo vedere dall'immagine prodotta sopra da `plotPotentialStartsAndStops()`.

6.5 Predire la sequenza proteica per un ORF

Se si trova un ORF in una sequenza DNA, è interessante estrarre la sequenza dei nucleotidi dell'ORF. Abbiamo visto tramite la funzione `findORFsinSeq()` che c'è un ORF ai nucleotidi 4–12 della sequenza *s1* (aaaatgcagtaacccatgccc). Per osservare la sequenza del DNA dell'ORF, possiamo usare la funzione `substring()` per isolare il pezzo di DNA. Ad esempio, per isolare la sottostringa della sequenza *s1* che corrisponde all'ORF dei nucleotidi 4–12, digitiamo:

```
> s1 <- "aaaatgcagtaacccatgccc"
> myorf <- substring(s1, 4, 12)
> myorf # Print out the sequence of "myorf"
  [1] "atgcagtaa"
```

Come si può vedere, l'ORF inizia con un potenziale codone di *start* (ATG), è seguito da un numero intero di codoni (un solo codone, CAG, in questo caso), e termina con un potenziale codone di *stop* (TAA).

Se si dispone della sequenza del DNA di un ORF, è possibile prevedere la sequenza proteica per l'ORF utilizzando la funzione `translate()` del pacchetto *seqinr*. Si noti che, poiché esiste una funzione con lo stesso nome in entrambi i pacchetti *Biostrings* e *seqinr*, dobbiamo digitare `seqinr::translate()` per specificare che vogliamo usare la funzione `translate()` del pacchetto *seqinr*.

La funzione richiede che la sequenza di input sia sotto forma di vettore di caratteri. Se la sequenza è sotto forma di stringa di caratteri, è possibile convertirla in un vettore di caratteri usando la funzione `s2c()` di *seqinr*. Ad esempio, per prevedere la sequenza proteica dell'ORF *myorf*, si dovrebbe digitare:

```
> myorfvector <- s2c(myorf) # Convert the sequence of characters to a vector
> myorfvector # Print out the value of "myorfvector"
[1] "a" "t" "g" "c" "a" "g" "t" "a" "a"
> seqinr::translate(myorfvector)
[1] "M" "Q" "*"
```

Dall'output della funzione `seqinr::translate()` vediamo che il potenziale codone di *start* (ATG) è tradotto come *metionina* (M) ed è seguito da una *glutammina* (Q). Il potenziale codone di *stop* è rappresentato come "*" in quanto non è tradotto in nessun aminoacido.

6.6 Ricerca di *reading frame* aperti (ORF) sul *reverse strand* di una sequenza di DNA

I geni in una sequenza genomica possono verificarsi sia sul *forward strand* del DNA che sul *reverse strand* (vedi Appendice). Per trovare gli ORF sul *reverse strand* di una sequenza, dobbiamo prima dedurre la sequenza dal *forward strand* e poi usare la funzione `findORFsinSeq()` per trovare gli ORF sul *reverse strand*.

La sequenza del *reverse strand* può essere facilmente dedotta da quella del *forward strand* poiché è sempre il complemento inverso della stessa. Possiamo usare la funzione `comp()` del pacchetto *SeqinR* per calcolare il complemento di una sequenza, e la funzione `rev()` per invertire tale sequenza in modo da ottenere quella del complemento inverso.

Le funzioni `comp()` e `rev()` richiedono che la sequenza di input sia sotto forma di vettore di caratteri. La funzione `s2c()` può essere usata per convertire una stringa di caratteri in un vettore, mentre la funzione `c2s()` è utile per riconvertire un vettore in una stringa di caratteri.

Ad esempio, se la nostra sequenza sul *forward strand* è "AAAATGCTTAAACCATTGCCC", e vogliamo trovare la sequenza sul *reverse strand*, digitiamo:

```
> forward <- "AAAATGCTTAAACCATTGCCC"
> forwardvector <- s2c(forward) # Convert the string of characters to a vector
> forwardvector # Print out the vector containing the forward strand sequence
 [1] "A" "A" "A" "A" "T" "G" "C" "T" "T" "A" "A" "A" "C" "C" "A" "T" "T" "G" "C" "C" "C"
> reversevector <- toupper(rev(comp(forwardvector))) # Find the reverse strand
sequence, by finding the reverse complement
> reversevector # Print out the vector containing the reverse strand sequence
 [1] "G" "G" "G" "C" "A" "A" "T" "G" "G" "T" "T" "T" "A" "A" "G" "C" "A" "T" "T" "T" "T"
> reverse <- c2s(reversevector) # Convert the vector to a string of characters
> reverse # Print out the string of characters containing the reverse strand sequence
[1] "GGGCAATGGTTTAAGCATTTT"
```

Nel comando `reversevector <- toupper(rev(comp(forwardvector)))` sopra abbiamo prima usato la funzione `comp()` per trovare il complemento della sequenza. Quindi abbiamo usato la funzione `rev()` per prendere la sequenza di uscita data da `comp()` e invertire l'ordine delle lettere della sequenza, trasformando infine tutte le lettere in maiuscole mediante la funzione `toupper()`.

Una volta dedotta la sequenza complementare, possiamo utilizzare la funzione `findORFsinSeq()` per trovare gli ORF nella stessa:

```
> findORFsinSeq(reverse)
[[1]]
[1] 6

[[2]]
[1] 14

[[3]]
[1] 9
```

Questo indica che c'è un ORF di lunghezza 9 nel *reverse strand* della sequenza "AAAATGCTTAAACCATTGCCC", che ha un codone di *start* previsto che inizia al nucleotide 6 e un codone di *stop* previsto che termina al nucleotide 14.

6.7 Lunghezza degli *Open Reading Frame*

Come si può vedere dall'immagine che mostra il codice genetico realizzato con `tablecode()` (sezione 6.1), tre dei 64 diversi codoni sono di *stop*. Ciò significa che in una sequenza casuale di DNA la probabilità che un qualsiasi codone sia un potenziale codone di *stop* è di 3/64 (circa il 5%).

Pertanto, ci si potrebbe aspettare che a volte i potenziali codoni di *start* e di *stop* possano verificarsi in una sequenza di DNA solo per caso, non perché sono in realtà parte di un qualsiasi gene reale che viene trascritto e tradotto in una proteina. Di conseguenza, molti degli ORF in una sequenza di DNA possono non corrispondere a geni reali, ma essere solo tratti di DNA tra i potenziali codoni di *start* e di *stop* che si trovano per caso nella sequenza.

In altre parole, un *Open Reading Frame* (ORF) è solo una previsione genica, o un gene potenziale. Può corrispondere a un gene reale (vero positivo), ma può non esserlo (falso positivo).

Come possiamo dire se i potenziali codoni di *start* e di *stop* di un ORF lo sono davvero, cioè se un ORF probabilmente corrisponde a un gene reale che viene trascritto e tradotto in una proteina? In realtà, non possiamo dirlo con i soli metodi bioinformatici (abbiamo bisogno di fare alcuni esperimenti di laboratorio per saperlo), ma possiamo fare una previsione abbastanza attendibile in base alla lunghezza dell'ORF.

Per definizione, un ORF è un tratto di DNA che inizia con un potenziale codone di *start*, e finisce con un potenziale codone di *stop* nello stesso *frame* di lettura, e quindi non ha codoni di *stop* interni in quel *frame*. Poiché circa il 5% in una sequenza casuale di DNA è un potenziale codone di *stop* solo per caso, se vediamo un ORF molto lungo (centinaia di codoni) sarebbe sorprendente che non ci fossero codoni di *stop* interni in un così lungo tratto di DNA se l'ORF non fosse un vero gene.

In altre parole, è improbabile che si verifichino ORF lunghi centinaia di codoni solo per caso; e quindi possiamo essere abbastanza sicuri che ORF così lunghi corrispondano probabilmente a geni reali.

6.8 Identificare *Open Reading Frame* significativi

Quanto deve essere lungo un ORF per poter essere sicuri che corrisponda a un gene reale? Questa è una domanda difficile.

Un approccio per rispondere a questa domanda è quello di chiedersi: *Qual è l'ORF più lungo che si trova in una sequenza casuale della stessa lunghezza e composizione nucleotidica della nostra sequenza originale?*

Gli ORF in una sequenza casuale non corrispondono a geni reali, ma sono dovuti solo a potenziali codoni di *start*/*stop* che si sono verificati per caso in quelle sequenze (poiché, per definizione, una sequenza casuale è generata in modo accidentale piuttosto che dall'evoluzione, come in un organismo reale).

Così, osservando le lunghezze degli ORF nella sequenza casuale, possiamo vedere qual è l'ORF più lungo che si verifica per caso.

Ma come possiamo ottenere sequenze casuali? In precedenza abbiamo visto che è possibile generare sequenze casuali usando un *modello multinomiale* con una particolare probabilità di ogni lettera (A, C, G e T nel caso di sequenze di DNA).

Abbiamo usato la funzione `generateSeqsWithMultinomialModel()` per generare sequenze casuali usando un modello multinomiale in cui la probabilità di ogni lettera è uguale alla frazione di quella lettera in una sequenza di input. Questa funzione prende due argomenti, la sequenza di input e il numero delle sequenze casuali che si vogliono generare.

Per esempio, per creare una sequenza casuale della stessa lunghezza di "AAAATGCTTAAACCATTGCCC", utilizzando un modello multinomiale in cui le probabilità di A, C, G e T sono uguali alle loro frazioni in questa sequenza, digitiamo:

```
> myseq <- "AAAATGCTTAAACCATTGCCC"
> generateSeqsWithMultinomialModel(myseq, 1) # generate one random sequence
[1] "CACTATAAACTAACACCTCTC"
```

Possiamo quindi utilizzare la funzione `findORFsinSeq()` per trovare gli ORF in questa sequenza casuale. Se lo ripetiamo 10 volte, possiamo trovare le lunghezze degli ORF nelle 10 sequenze casuali, da confrontare con la lunghezza della sequenza originale.

Per esempio, per confrontare le lunghezze degli ORF trovati nella sequenza del genoma del virus Dengue DEN-1 *dengueseq* con le lunghezze degli ORF trovati in 10 sequenze casuali generate usando un modello multinomiale (in cui le probabilità delle quattro basi sono uguali alle loro frazioni nella sequenza del virus), digitiamo:

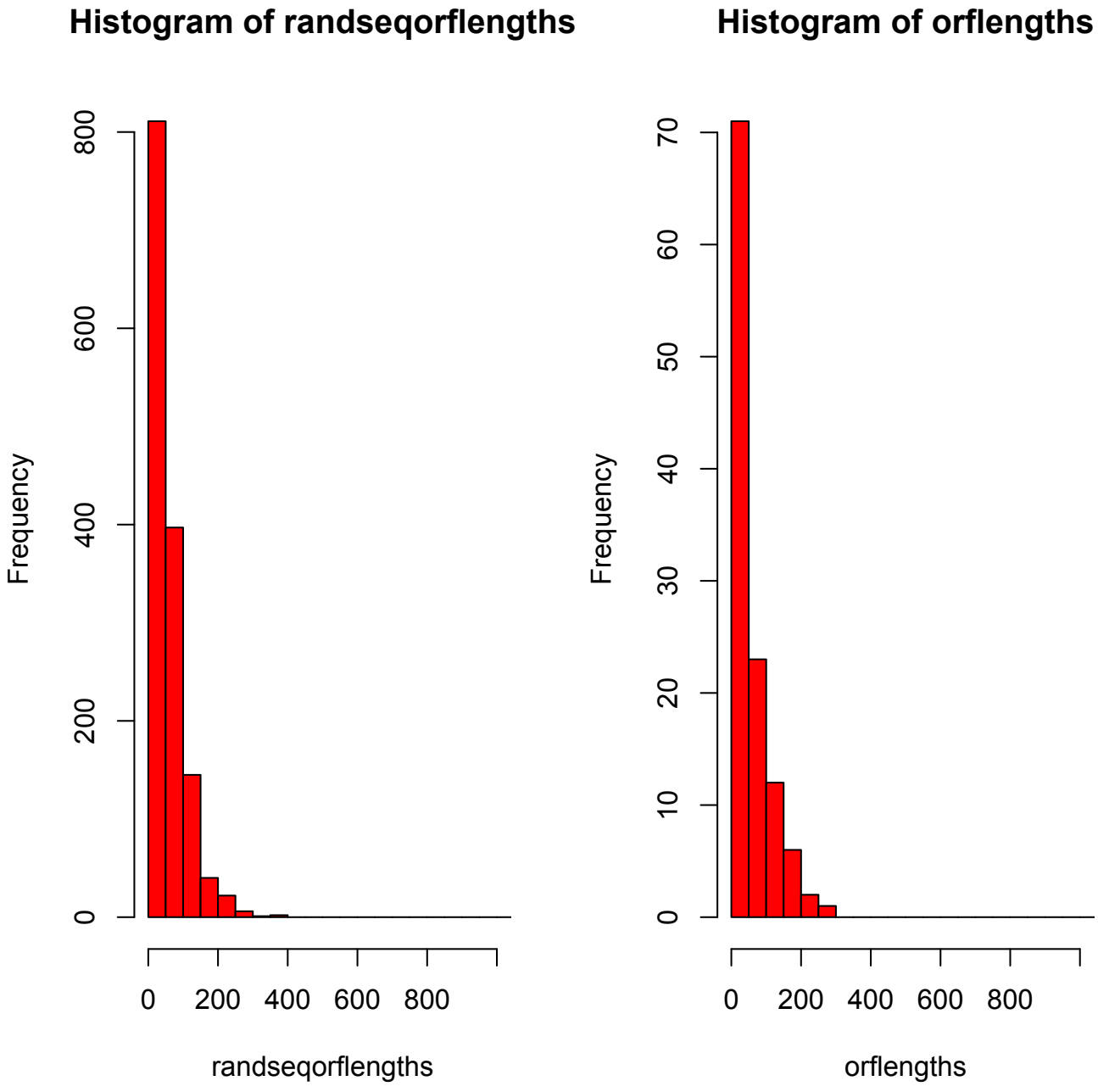
```
> denguseqstring <- c2s(dengueseq) # Convert the Dengue sequence to a string of characters
> mylist <- findORFsinSeq(denguseqstring) # Find ORFs in "denguseqstring"
> orflengths <- mylist[[3]] # Find the lengths of ORFs in "denguseqstring"
> randseqs <- generateSeqsWithMultinomialModel(denguseqstring, 10) # Generate 10 random
sequences using the multinomial model
> randseqorflengths <- numeric() # Make a new vector of numbers
> for (i in 1:10) {
      print(i)
      randseq <- randseqs[i] # Get the i-th random sequence
      mylist <- findORFsinSeq(randseq) # Find ORFs in "randseq"
      lengths <- mylist[[3]] # Find the lengths of ORFs in "randseq"
      randseqorflengths <- append(randseqorflengths, lengths,
            after=length(randseqorflengths))
}
```

Può occorrere un po' di tempo per l'esecuzione; tuttavia, il ciclo `for` stampa il valore di *i* per ogni iterazione in modo da poter controllare l'esecuzione.

Nel codice sopra recuperiamo le lunghezze degli ORF trovati dalla funzione `findORFsinSeq()` prendendo il terzo elemento della lista restituita dalla funzione, che è un vettore contenente le lunghezze di tutti gli ORF trovati nella sequenza di input.

Possiamo quindi tracciare un istogramma delle lunghezze degli ORF nella sequenza reale del genoma DEN-1 (*orflengths*) accanto ad un istogramma delle lunghezze degli ORF nelle 10 sequenze casuali (*randseqorflengths*):

```
> par(mfrow = c(1,2)) # Make a picture with two plots side-by-side (one row, two columns)
> bins <- seq(0,11000,50) # Set the bins for the histogram
> hist(randseqorflengths, breaks=bins, col="red", xlim=c(0,1000))
> hist(orflengths, breaks=bins, col="red", xlim=c(0,1000))
```



In altre parole, l'istogramma delle lunghezze degli ORF nelle 10 sequenze casuali dà un'idea della distribuzione delle lunghezze degli ORF che ci si aspetterebbe in una sequenza casuale di DNA (generata da un modello multinomiale in cui le probabilità delle quattro basi sono impostate uguali alle loro frequenze nella sequenza del genoma del virus DEN-1).

Possiamo calcolare il più lungo degli ORF che si verifica nelle sequenze casuali, utilizzando la funzione R `max()`, che può essere utilizzata per trovare l'elemento più grande in un vettore di numeri:

```
> max(randseqorflengths)
[1] 342
```

Ciò indica che l'ORF più lungo che si verifica nelle sequenze casuali è di 342 nucleotidi. Quindi, è possibile che un ORF fino a 342 nucleotidi si verifichi solo per caso in una sequenza casuale della stessa lunghezza e più o meno della stessa composizione del genoma del virus DEN-1.

Pertanto, potremmo usare 342 nucleotidi come soglia e scartare tutti gli ORF trovati nel genoma del virus DEN-1 che sono più corti di questo, partendo dal presupposto che probabilmente sono sorti per caso e probabilmente non corrispondono a geni reali. Quanti ORF rimarrebbero nella sequenza del genoma del virus DEN-1 se usassimo 342 nucleotidi come soglia?

```
> summary(orflengths>342)
   Mode   FALSE    TRUE
logical     115       1
```

Se usassimo 342 nucleotidi come soglia, rimarrebbe solo 1 ORF nel genoma del virus DEN-1. Alcuni dei 115 ORF più corti che abbiamo scartato potrebbero però corrispondere a geni reali.

In generale, non vogliamo perdere molti geni reali e potremmo perciò voler usare una soglia più tollerante. Ad esempio, invece di scartare tutti gli ORF di Dengue che sono più corti degli ORF più lunghi trovati nelle 10 sequenze casuali, potremmo scartare tutti gli ORF di Dengue che sono più corti del 99% degli ORF più lunghi nelle sequenze casuali.

Possiamo usare la funzione `quantile()` per trovare i quantili di un insieme di numeri. Il 99° quantile di un insieme di numeri è il valore *x* tale che il 99% dei numeri dell'insieme ha valori inferiori a *x*. Ad esempio, per trovare il 99° quantile di *randseqorflengths*, digitiamo:

```
> quantile(randseqorflengths, probs=c(0.99))
99%
248.07
```

Ciò significa che il 99% degli ORF nelle sequenze casuali ha lunghezze inferiori a 248 nucleotidi. In altre parole, il più lungo del 99% degli ORF più lunghi nelle sequenze casuali è costituito da 248 nucleotidi.

Quindi, se usassimo questo come soglia, scarteremmo tutti gli ORF del genoma DEN-1 lunghi 248 nucleotidi o meno. Questo risulterà in un minor numero di ORF scartati che se usassimo la soglia più rigorosa di 342 nucleotidi, quindi probabilmente avremo scartato meno ORF che corrispondono a geni reali. Sfortunatamente, ciò significa che probabilmente avremo anche mantenuto più falsi positivi allo stesso tempo, cioè ORF che non corrispondono a geni reali.

7. Genomica comparativa

7.1 Introduzione

La genomica comparativa è il campo della bioinformatica che consiste nel confrontare i genomi di due specie diverse, o di due ceppi diversi della stessa specie.

Una delle prime domande da porsi quando si confrontano i genomi di due specie è: le due specie hanno lo stesso numero di geni (cioè lo stesso contenuto genico)? Poiché tutta la vita sulla terra ha condiviso un antenato comune a un certo punto, qualsiasi specie, ad esempio quella umana e la mosca della frutta, deve essere discendente da una specie antenata comune.

Dal tempo dell'antenato comune di due specie (ad es. umana e topo), alcuni dei geni che erano presenti nella specie antenata comune possono essere andati perduti in una delle due discendenze. Inoltre, i due ceppi discendenti possono aver acquisito geni che non erano presenti nella specie antenata comune.

7.2 Utilizzo del pacchetto R *biomaRt* per interrogare la banca dati Ensembl

Per effettuare analisi genomiche comparative di due specie animali i cui genomi sono stati completamente sequenziati (ad es. l'uomo e il topo), è utile analizzare i dati presenti nella banca dati Ensembl (https://www.ensembl.org).

Il database principale Ensembl contiene i geni di vertebrati completamente sequenziati, nonché *Saccharomyces cerevisiae* (lievito) e un piccolo numero di altri animali di organismi modello (ad es. il verme nematode *Caenorhabditis elegans* e la mosca della frutta *Drosophila melanogaster*).

Esistono anche database Ensembl per altri gruppi di organismi, ad esempio "Ensembl Protists" (http://protists.ensembl.org/index.html) per Protisti, "Ensembl Metazoa" (http://metazoa.ensembl.org/index.html) per Metazoani, "Ensembl Bacteria" (http://bacteria.ensembl.org/index.html) per Batteri, "Ensembl Plants" (http://plants.ensembl.org/index.html) per Piante e "Ensembl Fungi" (http://fungi.ensembl.org/index.html) per Funghi.

È possibile effettuare analisi in R sulla banca dati Ensembl utilizzando il pacchetto *biomaRt* (che fa parte di Bioconductor), che può collegarsi alla banca dati Ensembl ed eseguire interrogazioni sui dati. Una volta installato il pacchetto *biomaRt* è possibile ottenere un elenco di database (*mart*) che possono essere interrogati digitando:

```
> library(biomaRt) # Load the biomaRt package in R
listMarts(host = "ensembl.org") # List the main marts that can be queried
               biomart                version
1 ENSEMBL_MART_ENSEMBL      Ensembl Genes 99
2   ENSEMBL_MART_MOUSE      Mouse strains 99
3     ENSEMBL_MART_SNP  Ensembl Variation 99
4 ENSEMBL_MART_FUNCGEN Ensembl Regulation 99
> listMarts(host = "protists.ensembl.org") # List the protists marts that can be queried
              biomart                          version
1       protists_mart      Ensembl Protists Genes 46
2 protists variations Ensembl Protists Variations 46
```

Qui parleremo dell'uso del pacchetto *biomaRt* per interrogare il database Ensembl, ma vale la pena ricordare che esso può essere usato anche per eseguire interrogazioni su altri database come UniProt.

Si può vedere sopra che *biomaRt* indica quale versione di ogni database può essere ricercata. Ad esempio, la versione del database principale Ensembl che può essere ricercata è Ensembl 99, mentre la versione del database Ensembl Protists che può essere ricercata è Ensembl Protists 46.

Se si desidera eseguire una ricerca nel database Ensembl utilizzando *biomaRt*, occorre innanzitutto specificare il database che si desidera interrogare. Si può fare utilizzando la funzione `useMart()` del pacchetto:

```
ensemblprotists <- useMart("protists_mart", host = "protists.ensembl.org")
```

Questo indica a *biomaR* che si desidera interrogare la banca dati Ensembl Protists, che contiene informazioni genomiche per diverse specie di protisti i cui genomi sono stati completamente sequenziati. Per vedere quali dataset si possono interrogare nel database selezionato, si può digitare:

```
> listDatasets(ensemblprotists)
                   dataset
1       alaibachii_eg_gene
2          bnatans_eg_gene
3      ddiscoideum_eg_gene
4     ehistolytica_eg_gene
5         ehuxleyi_eg_gene
6         glamblia_eg_gene
7           gtheta_eg_gene
8   harabidopsidis_eg_gene
9           lmajor_eg_gene
10 paphanidermatum_eg_gene
…
```

Viene restituito un lungo elenco degli organismi per i quali il database Ensembl Protists dispone di dati sul genoma, tra cui *Plasmodium vivax* e *Plasmodium falciparium* (che causano la malaria), e *Leishmania major*, che causa la leishmaniosi.

Per eseguire una query sul database Ensembl utilizzando *biomaRt*, è necessario innanzitutto specificare a quale dataset Ensembl si riferisce la query utilizzando la funzione `useDataset()`. Ad esempio, per specificare che si desidera eseguire una query sul dataset principale Ensembl *Leishmania*, si deve digitare:

```
> ensembllleishmania <- useDataset("lmajor_eg_gene", mart=ensemblprotists)
```

Si noti che il nome del dataset Ensembl del *Leishmania major* è "lmajor_eg_gene", che è stato elencato quando abbiamo digitato `listDatasets(ensemblprotists)` in precedenza.

Una volta specificato il particolare dataset Ensemble su cui si desidera eseguire una query, è possibile eseguire l'interrogazione mediante la funzione `getBM()`.

Di solito, si vuole eseguire una query per un particolare insieme di attributi del dataset. Quali attributi si possono cercare? Lo si può scoprire utilizzando la funzione `listAttributes()` di *biomaRt*:

```
> leishmaniaattributes <- listAttributes(ensemblleishmania)
```

La funzione `listAttributes()` restituisce un oggetto lista, il cui primo elemento è un vettore di tutti gli attributi che è possibile selezionare, e il secondo elemento è un vettore che contiene le spiegazioni di tutte queste caratteristiche:

```
> attributenames <- leishmaniaattributes[[1]]
> attributedescriptions <- leishmaniaattributes[[2]]
> length(attributenames)
[1] 757
> attributenames[1:10]
 [1] "ensembl_gene_id"       "ensembl_transcript_id" "ensembl_peptide_id"
 [4] "ensembl_exon_id"       "description"           "chromosome_name"
 [7] "start_position"        "end_position"          "strand"
[10] "band"
> attributedescriptions[1:10]
 [1] "Gene stable ID"           "Transcript stable ID"     "Protein stable ID"
 [4] "Exon stable ID"           "Gene description"         "Chromosome/scaffold name"
 [7] "Gene start (bp)"          "Gene end (bp)"            "Strand"
[10] "Karyotype band"
```

La funzione `listAttributes()` restituisce una lunga lista di 757 attributi nel dataseset Ensembl del *Leishmania major* che possiamo interrogare, come geni, trascrizioni (mRNA), peptidi (proteine), cromosomi, termini GO (Gene Ontologia) e così via.

Quando si esegue una query sul dataset principale *Leishmania* utilizzando `getBM()`, è necessario specificare quali attributi si desidera recuperare. Per esempio, si può vedere dall'output di `listAttributes()` che un possibile tipo di caratteristica che possiamo cercare sono i geni *Leishmania major*. Per recuperare una lista di tutti i geni principali della *Leishmania* dal dataset Ensembl del *Leishmania major* dobbiamo solo digitare:

```
> leishmaniagenes <- getBM(attributes = c("ensembl_gene_id"), mart=ensemblleishmania)
```

ottenendo una variabile lista *leishmaniagenes*, il cui primo elemento è un vettore che contiene i nomi di tutti i geni principali della *Leishmania*. Così, per trovare il numero di geni, e stampare i nomi dei primi dieci geni memorizzati nel vettore, possiamo digitare:

```
> leishmaniagenenames <- leishmaniagenes[[1]]
> length(leishmaniagenenames)
[1] 10030
> leishmaniagenenames[1:10]
 [1] "ENSRNAG00049765313" "ENSRNAG00049765315" "ENSRNAG00049765317" "ENSRNAG00049765319"
 [5] "ENSRNAG00049765321" "ENSRNAG00049765323" "ENSRNAG00049765325" "ENSRNAG00049765327"
 [9] "ENSRNAG00049765329" "ENSRNAG00049765333"
```

Questo ci dice che ci sono 10.030 diversi geni principali della *Leishmania* nel dataset del *L. major*. Si noti che questo include vari tipi di geni, tra cui i geni che codificano le proteine (sia i geni "noti" che quelli "nuovi", dove i geni "nuovi" sono predizioni di geni che non hanno somiglianza di sequenza con le sequenze presenti nei database di sequenze), i geni RNA e gli pseudogeni.

E se fossimo interessati solo ai geni che codificano le proteine? Se si guarda l'output di `listAttribuutes(ensemblleishmania)` si vede che uno degli attributi è "gene_biotype", che ci dice di che tipo è ogni gene (ad es. "protein_coding", "pseudogene", ecc.):

```
> leishmaniagenes2 <- getBM(attributes = c("ensembl_gene_id", "gene_biotype"),
mart=ensemblleishmania)
```

In questo caso, la funzione `getBM()` restituisce una variabile lista *leishmaniagenes2* il cui primo elemento è un vettore che contiene i nomi di tutti i geni principali della *Leishmania*, e il secondo è un vettore che contiene i tipi di questi geni:

```
> leishmaniagenenames2 <- leishmaniagenes2[[1]]
> leishmaniagenebiotypes2 <- leishmaniagenes2[[2]]
```

Possiamo fare una tabella di tutti i diversi tipi di geni utilizzando la funzione `table()`:

```
> table(leishmaniagenebiotypes2)
leishmaniagenebiotypes2
         ncRNA protein_coding     pseudogene           rRNA          snRNA           tRNA
          1130           8315             94             92            233            166
```

da cui si deduce che ci sono 8.315 geni che codificano le proteine, 94 pseudogeni e vari tipi di geni RNA (geni tRNA, geni rRNA, geni snRNA, ecc.).

7.3 Confronto del numero di geni in due specie

Ensembl è una risorsa molto utile per confrontare il contenuto genico delle diverse specie. Ad esempio, una semplice domanda che possiamo porre analizzando i dati di Ensembl è: quanti geni codificanti le proteine sono presenti in *Leishmania major* e quanti in *Plasmodium falciparum*?

Sappiamo quanti geni codificanti le proteine sono presenti nella Leishmania major (8.315; vedi sopra), ma che dire del *Plasmodium falciparum*? Per rispondere a questa domanda, dobbiamo prima dire a *biomaRt* che vogliamo fare una query sul dataset Ensembl *Plasmodium falciparum* usando la funzione `useDataset()`.

```
> ensemblpfalciparum <- useDataset("pfalciparum_eg_gene", mart=ensemblprotists)
```

Si noti che il nome Ensembl del dataset *Plasmodium falciparum* è "pfalciparum_eg_gene", come visto in precedenza.

Possiamo quindi usare `getBM()` come prima per recuperare i nomi di tutti i geni che codificano le proteine del *Plasmodium falciparum*. Questa volta dobbiamo impostare l'opzione "`mart`" nella funzione `getBM()`

su "*ensemblpfalciparum*", per specificare che vogliamo interrogare il dataset Ensembl del *Plasmodium falciparum* piuttosto che il dataset del *Leishmania major*:

```
> pfalciparumgenes <- getBM(attributes = c("ensembl_gene_id", "gene_biotype"),
      mart=ensemblpfalciparum)
> pfalciparumgenenames <- pfalciparumgenes[[1]]
> length(pfalciparumgenenames)
[1] 5767
> pfalciparumgenebiotypes <- pfalciparumgenes[[2]]
> table(pfalciparumgenebiotypes)
pfalciparumgenebiotypes
             ncRNA nontranslating_CDS     protein_coding         pseudogene
               102                   4               5358                153
              rRNA               snRNA               sRNA               tRNA
                44                  10                 17                 79
```

Questo ci dice che ci sono 5.358 geni che codificano le proteine del *Plasmodium falciparum*. Cioè, il *Plasmodium falciparum* sembra avere meno geni che codificano le proteine rispetto alla *Leishmania major* (8.315).

È interessante chiedersi: perché il *Plasmodium falciparum* ha meno geni codificanti per le proteine rispetto alla *Leishmania major*? Ci sono diverse possibili spiegazioni: (*a*) che ci sono state duplicazioni di geni nel ceppo principale della *Leishmania* da quando la *Leishmania* e il *Plasmodium* condividevano un antenato comune, che hanno dato origine a nuovi geni principali della *Leishmania*; (*b*) che nella discendenza della *Leishmania major* sono sorti geni completamente nuovi (che non sono imparentati con nessun altro gene della *Leishmania major*) da quando la *Leishmania* e il *Plasmodium* hanno condiviso un antenato comune; oppure (*c*) che ci sono geni persi dal genoma *Plasmodium falciparum* da quando la *Leishmania* e il *Plasmodium* hanno condiviso un antenato comune.

Per capire quale di queste spiegazioni è più probabile che sia corretta, dobbiamo capire come i principali geni codificanti delle proteine della *Leishmania* siano correlati ai geni codificanti delle proteine del *Plasmodium falciparum*.

7.4 Identificazione di geni omologhi tra due specie

La banca dati Ensembl raggruppa i geni omologhi (correlati) in famiglie di geni. Se un gene della *Leishmania major* e un gene del *Plasmodium falciparum* sono correlati, dovrebbero essere messi insieme nella stessa famiglia di geni nel database "Protists". Infatti, se un gene della *Leishmania major* ha qualche omologo in altri protisti, dovrebbe essere inserito in qualche famiglia di geni nel database "Protists".

Per tutti i geni della *Leishmania major* e del *Plasmodium falciparum* che sono in una stessa famiglia, Ensembl classifica la relazione tra ogni coppia di geni della *Leishmania major* e del *Plasmodium falciparum* come "ortologhi" (geni imparentati che condividono un antenato comune nell'antenato della *Leishmania* e del *Plasmodium*, e sono sorti a causa dell'evento di speciazione *Leishmania—Plasmodium*) o "paraloghi" (geni correlati che sono sorti a causa di un evento di duplicazione all'interno di una specie, per esempio a causa di un evento di duplicazione in *Leishmania major* o un evento di duplicazione nell'antenato *Leishmania—Plasmodium*).

Se si digita di nuovo `listAttributes(ensemblleishmania)` si vedrà che un possibile attributo che si può cercare è "pfalciparum_eg_homolog_ensembl_gene", che è l'ortologo *Plasmodium falciparum* di un gene della *Leishmania major*.

Un altro possibile attributo che si può cercare è "pfalciparum_eg_homolog_orthology_type", che descrive il tipo di relazione ortologica tra un particolare gene della *Leishmania major* e il suo ortologo *Plasmodium falciparum*. Per esempio, se un particolare gene della *Leishmania major* ha due ortologhi *Plasmodium falciparum*, la relazione tra il gene della *Leishmania major* e ciascuno degli ortologhi *Plasmodium falciparum* sarà "ortholog_one2many" (ortologia uno-a-molti).

Ciò può verificarsi nel caso in cui ci sia stata una duplicazione nel ceppo *Plasmodium falciparum* dopo la divergenza tra *Plasmodium* e *Leishmania*, il che significa che due diversi geni *Plasmodium falciparum* (che sono paraloghi l'uno dell'altro) sono entrambi ortologhi dello stesso gene *Leishmania major*.

Pertanto, possiamo recuperare gli identificatori Ensembl degli ortologhi del *Plasmodium falciparum* di tutti i geni principali della *Leishmania* digitando:

```
> leishmaniagenes <- getBM(attributes = c("ensembl_gene_id",
      "pfalciparum_eg_homolog_ensembl_gene",
      "pfalciparum_eg_homolog_orthology_type"), mart=ensemblleishmania)
```

Questo restituirà una variabile lista *leishmaniagenes*, il cui primo elemento è un vettore di identificatori Ensembl per tutti i principali geni codificanti della *Leishmania*, il secondo elemento è un vettore di identificatori Ensembl per i loro ortologhi *Plasmodium falciparum* e il terzo è un vettore con informazioni sui tipi di ortologia.

Possiamo visualizzare i nomi dei primi 10 geni principali della *Leishmania* e dei loro ortologhi *Plasmodium falciparum*, e dei loro tipi di ortologia, digitando:

```
> leishmaniagenenames <- leishmaniagenes[[1]]
> leishmaniaPforthologues <- leishmaniagenes[[2]]
> leishmaniaPforthologuetypes <- leishmaniagenes[[3]]
> leishmaniagenenames[1:10]
 [1] "LMJF_01_0780" "LMJF_01_0350" "LMJF_01_0280" "LMJF_01_0290" "LMJF_01_0335"
 [6] "LMJF_01_0600" "LMJF_01_0220" "LMJF_01_0220" "LMJF_01_0220" "LMJF_01_0220"
> leishmaniaPforthologues[1:10]
 [1] ""              ""              ""              ""              ""
 [6] ""              "PF3D7_0528700" "PF3D7_1116300" "PF3D7_0804800" "PF3D7_1423200"
> leishmaniaPforthologuetypes[1:10]
 [1] ""                    ""                    ""                    ""
 [5] ""                    ""                    "ortholog_many2many" "ortholog_many2many"
 [9] "ortholog_many2many" "ortholog_many2many"
```

Non tutti i geni principali della *Leishmania* hanno ortologhi *Plasmodium falciparum*; ecco perché quando stampiamo i primi 10 elementi del vettore *leishmaniaPforthologues* alcuni elementi sono vuoti. Per scoprire quanti geni della *Leishmania major* hanno ortologhi nel *Plasmodium falciparum*, possiamo prima trovare gli indici degli elementi del vettore *leishmaniaPforthologues* che sono vuoti:

```
> myindex <- leishmaniaPforthologues==""
```

Possiamo quindi scoprire i nomi dei geni del *Leishmania* corrispondenti a questi indici:

```
> leishmaniagenenames2 <- leishmaniagenenames[myindex]
> length(leishmaniagenenames2)
[1] 8191
```

Questo ci dice che 8.191 geni principali della *Leishmania* non hanno ortologhi *Plasmodium falciparum*.

Quanti degli 8.191 geni della *Leishmania major* che non hanno gli ortologhi *Plasmodium falaciparum* sono geni che codificano proteine? Per rispondere a questa domanda, possiamo unire insieme le informazioni delle variabili *leishmaniagenes2* (che contiene informazioni sul nome di ogni gene della *Leishmania major* e sul suo tipo; vedi sopra) e *leishmaniagenes* (che contiene informazioni sul nome di ogni gene *L. major* e sui suoi ortologhi *Plasmodium falciparum*).

Ricordando che *leishmaniagenes2* è stata creata con la tipizzazione:

```
> leishmaniagenes2 <- getBM(attributes = c("ensembl_gene_id", "gene_biotype"),
      mart=ensemblleishmania)
```

per combinare *leishmaniagenes* e *leishmaniagenes2* possiamo usare la funzione `merge()` di R, che può fondere insieme due variabili che contengono alcuni elementi denominati in comune (in questo caso, entrambe le variabili hanno una colonna con i nomi dei geni principali della *Leishmania*):

```
> leishmaniagenes3 <- merge(leishmaniagenes2, leishmaniagenes)
```

La prima colonna del data.frame *leishmaniagenes3* contiene i nomi dei geni principali della *Leishmania*, la seconda i tipi di questi geni (ad es. codifica delle proteine, pseudogene, ecc.) e la terza i nomi degli ortologhi *Plasmodium falciparum*. Possiamo quindi scoprire quanti geni principali della *Leishmania* che codificano proteine sono privi di ortologhi *Plasmodium falciparum* digitando:

```
> leishmaniagenenames <- leishmaniagenes3[[1]]
> leishmaniagenebiotypes <- leishmaniagenes3[[2]]
> leishmaniaPforthologues <- leishmaniagenes3[[3]]
> myindex <- leishmaniaPforthologues=="" & leishmaniagenebiotypes=="protein_coding"
> leishmaniagenenames2 <- leishmaniagenenames[myindex]
> length(leishmaniagenenames2)
[1] 6476
```

Si vede che ci sono 6.476 geni della *Leishmania* che codificano le proteine principali che mancano di ortologhi *Plasmodium falciparum*.

7.5 Estendere *biomaRt* con il nuovo sistema di interrogazione *biomartr*

Per iniziare con *biomartr*

La metodologia di interrogazione fornita da Ensembl *Biomart* e dal pacchetto R *biomaRt* è un approccio molto ben definito per un accurato recupero delle annotazioni. Tuttavia, questa metodologia di

interrogazione non è molto intuitiva dal punto di vista dell'utente. Il pacchetto *biomartr* fornisce un'altra metodologia di interrogazione che mira ad essere più centrata sull'organismo.

Il flusso di lavoro in *biomartr* permette agli utenti di eseguire query *BioMart* veloci per gli attributi utilizzando la funzione `biomart()` implementata in questo pacchetto; tipicamente la sequenza di lavoro è la seguente:

1. Ottenere attributi, dataset e mart (database) tramite la funzione: `organismAttributes()`
2. Scegliere le caratteristiche biologiche disponibili (filtri) tramite la funzione: `organismFilters()`
3. Specificare un set di geni da interrogare mediante le funzioni `getGenome()`, `getProteome()` o `getCDS()`[2]
4. Specificare tutti gli argomenti della funzione `biomart()` utilizzando i passi 1–3 ed eseguire una query *BioMart*

Recuperare mart, dataset, attributi e filtri con *biomartr*

La funzione `getMarts()` permette agli utenti di elencare tutti i database disponibili a cui si può accedere attraverso le interfacce *BioMart*:

```
> # load the biomartr package
> library(biomartr)
>
> # list all available databases
> biomartr::getMarts()
# A tibble: 15 x 2
   mart                 version
   <chr>                <chr>
 1 ENSEMBL_MART_ENSEMBL  Ensembl Genes 99
 2 ENSEMBL_MART_MOUSE    Mouse strains 99
 3 ENSEMBL_MART_SEQUENCE Sequence
 4 ENSEMBL_MART_ONTOLOGY Ontology
 5 ENSEMBL_MART_GENOMIC  Genomic features 99
 6 ENSEMBL_MART_SNP      Ensembl Variation 99
 7 ENSEMBL_MART_FUNCGEN  Ensembl Regulation 99
 8 plants_mart           Ensembl Plants Genes 45
 9 plants_variations     Ensembl Plants Variations 46
10 fungi_mart            Ensembl Fungi Genes 46
11 fungi_variations      Ensembl Fungi Variations 46
12 protists_mart         Ensembl Protists Genes 46
13 protists_variations   Ensembl Protists Variations 46
14 metazoa_mart          Ensembl Metazoa Genes 46
15 metazoa_variations    Ensembl Metazoa Variations 46
```

Il tipo di dato `"tibble"` restituito dalla funzione `getMarts()` è un'estensione del tipo di dato `"data.frame"`, con alcune caratteristiche di impostazione tipografica che qui non esamineremo.

L'utente può selezionare un database specifico per elencare tutti i relativi dataset disponibili. In questo esempio scegliamo il database ENSEMBL_MART_ENSEMBL:

[2] CDS = Coding DNA Sequence, sequenza codificante di DNA

```
> biomartr::getDatasets(mart = "ENSEMBL_MART_ENSEMBL")
# A tibble: 202 x 3
    dataset                      description                                                version
    <chr>                        <chr>                                                      <chr>
  1       pcinereus_gene_ensembl                     Koala genes (phaCin_unsw_v4.1)         phaCin_unsw_v4.1
  2         ggallus_gene_ensembl                            Chicken genes (GRCg6a)                 GRCg6a
  3          hhucho_gene_ensembl                         Huchen genes (ASM331708v1)              ASM331708v1
  4       lcoronata_gene_ensembl Blue-crowned manakin genes (Lepidothrix_coronata-1.0) Lepidothrix_coronata-1.0
  5 dnovaehollandiae_gene_ensembl                              Emu genes (droNov1)                   droNov1
…
 98 hsapiens_gene_ensembl          Human genes (GRCh38.p13)                                   GRCh38.p13
…
199    loculatus_gene_ensembl            Spotted gar genes (LepOcu1)        LepOcu1
```

Ora è possibile selezionare il dataset *hsapiens_gene_ensembl* ed elencare tutti gli attributi disponibili:

```
> head(biomartr::getAttributes(mart = "ENSEMBL_MART_ENSEMBL",
      dataset = "hsapiens_gene_ensembl"), 5)
                           name                  description
1               ensembl_gene_id               Gene stable ID
2       ensembl_gene_id_version       Gene stable ID version
3         ensembl_transcript_id         Transcript stable ID
4 ensembl_transcript_id_version Transcript stable ID version
5             ensembl_peptide_id             Protein stable ID
```

Poiché vi sono ben 3446 attributi, abbiamo utilizzato la funzione `head()` per visualizzare solo i primi cinque.

Infine, la funzione `getFilters()` permette agli utenti di elencare i filtri disponibili (434 in totale) per uno specifico dataset che possono essere utilizzati per una query con la funzione `biomart()`:

```
> # List the available filters for dataset: hsapiens_gene_ensembl
> biomartr::getFilters(mart="ENSEMBL_MART_ENSEMBL", dataset="hsapiens_gene_ensembl")
                name                                 description
1                        chromosome_name                                  Chromosome/scaffold name
2                                  start                                                     Start
3                                    end                                                       End
4                             band_start                                                Band Start
5                               band_end                                                  Band End
6                           marker_start                                              Marker Start
7                             marker_end                                                Marker End
8                          encode_region                                             Encode region
9                                 strand                                                    Strand
10                     chromosomal_region                       e.g. 1:100:10000:-1, 1:100000:200000:1
11                               with_ccds                                          With CCDS ID(s)
12                            with_chembl                                       With ChEMBL ID(s)
13          with_clone_based_ensembl_gene                  With Clone-based (Ensembl) gene ID(s)
14    with_clone_based_ensembl_transcript              With Clone-based (Ensembl) transcript ID(s)
15                            with_dbass3            With DataBase of Aberrant 3' Splice Sites ID(s)
16                            with_dbass5            With DataBase of Aberrant 5' Splice Sites ID(s)
17                with_ens_hs_transcript                       With Ensembl Human Transcript ID(s)
18               with_ens_hs_translation                     With Ensembl Human Translation ID(s)
19            with_entrezgene_trans_name                   With EntrezGene transcript name ID(s)
20                               with_embl                 With European Nucleotide Archive ID(s)
21                     with_arrayexpress                          With Expression Atlas ID(s)
22                           with_genedb                                    With GeneDB ID(s)
23                              with_go                                        With GO ID(s)
24                       with_goslim_goa                             With GOSlim GOA ID(s)
25                             with_hgnc                           With HGNC Symbol ID(s)
26                              with_hpa                  With Human Protein Atlas ID(s)
…
```

Recuperare informazioni specifiche per un organismo

Nella maggior parte dei casi, si lavora con un singolo organismo o un insieme di organismi modello. In questo processo si è per lo più interessati ad annotazioni specifiche per un particolare organismo modello. La funzione `organismBM()` affronta questo problema e fornisce una query, centrata sull'organismo, ai database e dataset per un particolare organismo di interesse.

Si noti che quando si eseguono tali funzioni per la prima volta, la procedura di recupero dei dati richiede un certo tempo, a causa dell'accesso remoto a *BioMart*. Il risultato corrispondente viene quindi salvato in un file di tipo testo denominato "_biomart/listDatasets.txt" all'interno della cartella temporanea di lavoro (visualizzabile con il comando `tempdir()`), consentendo di eseguire le query successive molto più velocemente. La cartella `tempdir()`, tuttavia, verrà cancellata dopo la creazione di una sessione R. All'avvio quindi di una nuova sessione la chiamata iniziale di queste funzioni richiederà di nuovo tempo per recuperare tutti i dati specifici dell'organismo dal database *BioMart*.

Il concetto di memorizzazione locale di tutti i dati specifici dell'organismo disponibili in *BioMart* in un file interno consente agli utenti di accelerare notevolmente le successive interrogazioni per quel particolare organismo:

```
> # retrieving all available datasets and biomart connections for a specific query organism (scientific name)
> biomartr::organismBM(organism = "Homo sapiens")
# A tibble: 16 x 5
   organism_name description                                                    mart              dataset                 version
   <chr>         <chr>                                                          <chr>             <chr>                   <chr>
 1 hsapiens      Human genes (GRCh38.p13)                                       ENSEMBL_MART_ENS… hsapiens_gene_ensembl   GRCh38.p…
 2 hsapiens      Human sequences (GRCh38.p13)                                   ENSEMBL_MART_SEQ… hsapiens_genomic_sequ… GRCh38.p…
 3 hsapiens      marker_feature                                                 ENSEMBL_MART_GEN… hsapiens_marker_start   GRCh38.p…
 4 hsapiens      marker_feature_end                                             ENSEMBL_MART_GEN… hsapiens_marker_end     GRCh38.p…
 5 hsapiens      karyotype_end                                                  ENSEMBL_MART_GEN… hsapiens_karyotype_end  GRCh38.p…
 6 hsapiens      karyotype_start                                                ENSEMBL_MART_GEN… hsapiens_karyotype_st… GRCh38.p…
 7 hsapiens      encode                                                         ENSEMBL_MART_GEN… hsapiens_encode         GRCh38.p…
 8 hsapiens      Human Somatic Structural Variants (GRCh38.p13)                 ENSEMBL_MART_SNP  hsapiens_structvar_som GRCh38.p…
 9 hsapiens      Human Somatic Short Variants (SNPs and indels excluding flagged v… ENSEMBL_MART_SNP  hsapiens_snp_som     GRCh38.p…
10 hsapiens      Human Short Variants (SNPs and indels excluding flagged variants)… ENSEMBL_MART_SNP  hsapiens_snp           GRCh38.p…
11 hsapiens      Human Structural Variants (GRCh38.p13)                         ENSEMBL_MART_SNP  hsapiens_structvar      GRCh38.p…
12 hsapiens      Human Regulatory Features (GRCh38.p12)                         ENSEMBL_MART_FUN… hsapiens_regulatory_f… GRCh38.p…
13 hsapiens      Human Regulatory Evidence (GRCh38.p12)                         ENSEMBL_MART_FUN… hsapiens_peak           GRCh38.p…
14 hsapiens      Human Other Regulatory Regions (GRCh38.p12)                    ENSEMBL_MART_FUN… hsapiens_external_fea… GRCh38.p…
15 hsapiens      Human Binding Motifs (GRCh38.p12)                              ENSEMBL_MART_FUN… hsapiens_motif_feature GRCh38.p…
16 hsapiens      Human miRNA Target Regions (GRCh38.p12)                        ENSEMBL_MART_FUN… hsapiens_mirna_target… GRCh38.p…
```

Il risultato della funzione `organismBM()` è una tabella che mostra tutti i database e i dataset da cui è possibile recuperare le annotazioni per l'organismo indicato (*Homo sapiens*). Inoltre, viene restituita una breve descrizione e la versione del dataset a cui si accede (molto utile per le pubblicazioni).

Si noti, tuttavia, che i nomi scientifici degli organismi devono essere scritti correttamente! Per es. "Homo Sapiens" sarà trattato in modo diverso (non riconosciuto) rispetto a "Homo sapiens" (riconosciuto).

Analogamente alla metodologia di interrogazione del pacchetto *biomaRt*, occorre specificare gli attributi e i filtri per poter eseguire query accurate sul database *BioMart*. Le funzioni `organismAttributes()` e `organismFilters()` forniscono un'intefaccia concettuale utile e intuitiva.

Nell'esempio seguente si osservi che la funzione `organismAttributes()` restituisce un data.frame che memorizza i nomi degli attributi, i dataset e i database disponibili per l'*Homo sapiens* (talvolta la funzione

potrebbe visualizzare messaggi di avviso per informare l'utente quando alcuni database forniti da Ensembl non funzionano ancora correttamente):

```
> # return the first 20 available attributes for "Homo sapiens"
> head(biomartr::organismAttributes("Homo sapiens"), 20)
# A tibble: 20 x 4
   name                          description                                dataset               mart
   <chr>                         <chr>                                      <chr>                 <chr>
 1 ensembl_gene_id               Gene stable ID                             hsapiens_gene_ensembl ENSEMBL_MART_ENSEMBL
 2 ensembl_gene_id_version       Gene stable ID version                     hsapiens_gene_ensembl ENSEMBL_MART_ENSEMBL
 3 ensembl_transcript_id         Transcript stable ID                       hsapiens_gene_ensembl ENSEMBL_MART_ENSEMBL
 4 ensembl_transcript_id_version Transcript stable ID version               hsapiens_gene_ensembl ENSEMBL_MART_ENSEMBL
 5 ensembl_peptide_id            Protein stable ID                          hsapiens_gene_ensembl ENSEMBL_MART_ENSEMBL
 6 ensembl_peptide_id_version    Protein stable ID version                  hsapiens_gene_ensembl ENSEMBL_MART_ENSEMBL
 7 ensembl_exon_id               Exon stable ID                             hsapiens_gene_ensembl ENSEMBL_MART_ENSEMBL
 8 description                   Gene description                           hsapiens_gene_ensembl ENSEMBL_MART_ENSEMBL
 9 chromosome_name               Chromosome/scaffold name                   hsapiens_gene_ensembl ENSEMBL_MART_ENSEMBL
10 start_position                Gene start (bp)                            hsapiens_gene_ensembl ENSEMBL_MART_ENSEMBL
11 end_position                  Gene end (bp)                              hsapiens_gene_ensembl ENSEMBL_MART_ENSEMBL
12 strand                        Strand                                     hsapiens_gene_ensembl ENSEMBL_MART_ENSEMBL
13 band                          Karyotype band                             hsapiens_gene_ensembl ENSEMBL_MART_ENSEMBL
14 transcript_start              Transcript start (bp)                      hsapiens_gene_ensembl ENSEMBL_MART_ENSEMBL
15 transcript_end                Transcript end (bp)                        hsapiens_gene_ensembl ENSEMBL_MART_ENSEMBL
16 transcription_start_site      Transcription start site (TSS)             hsapiens_gene_ensembl ENSEMBL_MART_ENSEMBL
17 transcript_length             Transcript length (including UTRs and CDS) hsapiens_gene_ensembl ENSEMBL_MART_ENSEMBL
18 transcript_tsl                Transcript support level (TSL)             hsapiens_gene_ensembl ENSEMBL_MART_ENSEMBL
19 transcript_gencode_basic      GENCODE basic annotation                   hsapiens_gene_ensembl ENSEMBL_MART_ENSEMBL
20 transcript_appris             APPRIS annotation                          hsapiens_gene_ensembl ENSEMBL_MART_ENSEMBL
```

Una caratteristica aggiuntiva fornita da `organismAttributes()` è l'argomento `topic`, che permette agli utenti di cercare specifici attributi, argomenti o categorie per un filtraggio più veloce. Ad esempio, per ricercare i nomi di attributi che hanno `"id"` come parte del loro nome basta digitare:

```
> # search for the first 10 attribute topic "id"
> head(biomartr::organismAttributes("Homo sapiens", topic = "id"), 10)
# A tibble: 10 x 4
   name                          description                dataset               mart
   <chr>                         <chr>                      <chr>                 <chr>
 1 ensembl_gene_id               Gene stable ID             hsapiens_gene_ensembl ENSEMBL_MART_ENSEMBL
 2 ensembl_gene_id_version       Gene stable ID version     hsapiens_gene_ensembl ENSEMBL_MART_ENSEMBL
 3 ensembl_transcript_id         Transcript stable ID       hsapiens_gene_ensembl ENSEMBL_MART_ENSEMBL
 4 ensembl_transcript_id_version Transcript stable ID version hsapiens_gene_ensembl ENSEMBL_MART_ENSEMBL
 5 ensembl_peptide_id            Protein stable ID          hsapiens_gene_ensembl ENSEMBL_MART_ENSEMBL
 6 ensembl_peptide_id_version    Protein stable ID version  hsapiens_gene_ensembl ENSEMBL_MART_ENSEMBL
 7 ensembl_exon_id               Exon stable ID             hsapiens_gene_ensembl ENSEMBL_MART_ENSEMBL
 8 peptide_version               Version (protein)          hsapiens_gene_ensembl ENSEMBL_MART_ENSEMBL
 9 study_external_id             Study external reference   hsapiens_gene_ensembl ENSEMBL_MART_ENSEMBL
10 go_id                         GO term accession          hsapiens_gene_ensembl ENSEMBL_MART_ENSEMBL
```

Oppure, per ricercare i nomi di attributi che hanno `"homolog"` come parte del loro nome:

```
> # search for the first 10 attribute topic "homolog"
> head(biomartr::organismAttributes("Homo sapiens", topic = "homolog"), 10)
# A tibble: 10 x 4
   name                                description                                                dataset           mart
   <chr>                               <chr>                                                      <chr>             <chr>
 1 cabingdonii_homolog_ensembl_gene    Abingdon island giant tortoise gene stable ID              hsapiens_gene_en… ENSEMBL_MART_EN…
 2 cabingdonii_homolog_associated_gene_n… Abingdon island giant tortoise gene name               hsapiens_gene_en… ENSEMBL_MART_EN…
 3 cabingdonii_homolog_ensembl_peptide Abingdon island giant tortoise protein or transcript sta… hsapiens_gene_en… ENSEMBL_MART_EN…
 4 cabingdonii_homolog_chromosome      Abingdon island giant tortoise chromosome/scaffold name    hsapiens_gene_en… ENSEMBL_MART_EN…
 5 cabingdonii_homolog_chrom_start     Abingdon island giant tortoise chromosome/scaffold start… hsapiens_gene_en… ENSEMBL_MART_EN…
 6 cabingdonii_homolog_chrom_end       Abingdon island giant tortoise chromosome/scaffold end (… hsapiens_gene_en… ENSEMBL_MART_EN…
 7 cabingdonii_homolog_canonical_transcr… Query protein or transcript ID                          hsapiens_gene_en… ENSEMBL_MART_EN…
 8 cabingdonii_homolog_subtype         Last common ancestor with Abingdon island giant tortoise   hsapiens_gene_en… ENSEMBL_MART_EN…
 9 cabingdonii_homolog_orthology_type  Abingdon island giant tortoise homology type               hsapiens_gene_en… ENSEMBL_MART_EN…
10 cabingdonii_homolog_perc_id         %id. target Abingdon island giant tortoise gene identica… hsapiens_gene_en… ENSEMBL_MART_EN…
```

Analogamente alla funzione `organismAttributes()`, la funzione `organismFilters()` restituisce tutti i filtri disponibili per un interrogare un organismo di interesse:

```
> # return the first 10 available filters for "Homo sapiens"
> head(biomartr::organismFilters("Homo sapiens"), 10)
# A tibble: 10 x 4
   name               description                              dataset               mart
   <chr>              <chr>                                    <chr>                 <chr>
 1 chromosome_name    Chromosome/scaffold name                 hsapiens_gene_ensembl ENSEMBL_MART_ENSEMBL
 2 start              Start                                    hsapiens_gene_ensembl ENSEMBL_MART_ENSEMBL
 3 end                End                                      hsapiens_gene_ensembl ENSEMBL_MART_ENSEMBL
 4 band_start         Band Start                               hsapiens_gene_ensembl ENSEMBL_MART_ENSEMBL
 5 band_end           Band End                                 hsapiens_gene_ensembl ENSEMBL_MART_ENSEMBL
 6 marker_start       Marker Start                             hsapiens_gene_ensembl ENSEMBL_MART_ENSEMBL
 7 marker_end         Marker End                               hsapiens_gene_ensembl ENSEMBL_MART_ENSEMBL
 8 encode_region      Encode region                            hsapiens_gene_ensembl ENSEMBL_MART_ENSEMBL
 9 strand             Strand                                   hsapiens_gene_ensembl ENSEMBL_MART_ENSEMBL
10 chromosomal_region e.g. 1:100:10000:-1, 1:100000:200000:1 hsapiens_gene_ensembl ENSEMBL_MART_ENSEMBL
```

La funzione `organismFilters()` consente inoltre agli utenti di cercare i filtri che corrispondono a un determinato argomento o categoria:

```
> # search for the first 10 filter topic "id"
> head(biomartr::organismFilters("Homo sapiens", topic = "id"), 10)
# A tibble: 10 x 4
   name                          description                                                  dataset              mart
   <chr>                         <chr>                                                        <chr>                <chr>
 1 with_protein_id               With INSDC protein ID ID(s)                                  hsapiens_gene_ensem… ENSEMBL_MART_ENSEM…
 2 with_mim_morbid               With MIM morbid ID(s)                                        hsapiens_gene_ensem… ENSEMBL_MART_ENSEM…
 3 with_refseq_peptide           With RefSeq peptide ID(s)                                    hsapiens_gene_ensem… ENSEMBL_MART_ENSEM…
 4 with_refseq_peptide_predicted With RefSeq peptide predicted ID(s)                          hsapiens_gene_ensem… ENSEMBL_MART_ENSEM…
 5 ensembl_gene_id               Gene stable ID(s) [e.g. ENSG00000000003]                     hsapiens_gene_ensem… ENSEMBL_MART_ENSEM…
 6 ensembl_gene_id_version       Gene stable ID(s) with version [e.g. ENSG00000000003.15]     hsapiens_gene_ensem… ENSEMBL_MART_ENSEM…
 7 ensembl_transcript_id         Transcript stable ID(s) [e.g. ENST00000000233]               hsapiens_gene_ensem… ENSEMBL_MART_ENSEM…
 8 ensembl_transcript_id_version Transcript stable ID(s) with version [e.g. ENST00000000233.… hsapiens_gene_ensem… ENSEMBL_MART_ENSEM…
 9 ensembl_peptide_id            Protein stable ID(s) [e.g. ENSP00000000233]                  hsapiens_gene_ensem… ENSEMBL_MART_ENSEM…
10 ensembl_peptide_id_version    Protein stable ID(s) with version [e.g. ENSP00000000233.5]   hsapiens_gene_ensem… ENSEMBL_MART_ENSEM…
```

Costruire query sul database *BioMart* con il pacchetto *biomartr*

Le funzionalità di `organismBM()`, `organismAttributes()` e `organismFilters()` consente di eseguire le query *BioMart* in modo molto intuitivo e centrato sull'organismo di interesse. La funzione principale per eseguire le query sul database Ensembl *BioMart* è `biomart()`.

Nell'esempio all'inizio della pagina seguente supponiamo di essere interessati all'annotazione di specifici geni del proteoma *Homo sapiens*. Vogliamo mappare l'ID del gene *RefSeq* corrispondente a un insieme di altri ID genici utilizzati in altri database (a questo scopo, utilizziamo prima la funzione `organismAttributes()`).

La funzione `biomart()` prende come argomenti un insieme di geni (gli "ID type" dei geni sono specificati nell'argomento `filters`), il database e il dataset corrispondente, così come gli attributi che devono essere restituiti.

```
> head(biomartr::organismAttributes("Homo sapiens", topic = "id"))
# A tibble: 6 x 4
   name                        description                 dataset              mart
   <chr>                       <chr>                       <chr>                <chr>
 1 ensembl_gene_id             Gene stable ID              hsapiens_gene_ensembl ENSEMBL_MART_ENSEMBL
 2 ensembl_gene_id_version     Gene stable ID version      hsapiens_gene_ensembl ENSEMBL_MART_ENSEMBL
 3 ensembl_transcript_id       Transcript stable ID        hsapiens_gene_ensembl ENSEMBL_MART_ENSEMBL
 4 ensembl_transcript_id_version Transcript stable ID version hsapiens_gene_ensembl ENSEMBL_MART_ENSEMBL
 5 ensembl_peptide_id          Protein stable ID           hsapiens_gene_ensembl ENSEMBL_MART_ENSEMBL
 6 ensembl_peptide_id_version  Protein stable ID version   hsapiens_gene_ensembl ENSEMBL_MART_ENSEMBL
> # retrieve the proteome of Homo sapiens from refseq
> file_path <- biomartr::getProteome(db = "refseq", organism = "Homo sapiens",
      path = file.path("_ncbi_downloads","proteomes"))
> Hsapiens_proteome <- biomartr::read_proteome(file_path, format = "fasta")
# remove splice variants from id
gene_set <- unlist(sapply(strsplit(Hsapiens_proteome@ranges@NAMES[1:5], ".",
      fixed = TRUE), function(x) x[1]))
> result_BM <- biomartr::biomart(
      genes = gene_set, # genes retrieved using getGenome()
      mart  = "ENSEMBL_MART_ENSEMBL", # marts selected with getMarts()
      dataset = "hsapiens_gene_ensembl", # datasets selected with getDatasets()
      attributes = c("ensembl_gene_id","ensembl_peptide_id"), # attributes selected with getAttributes()
      filters = "refseq_peptide") # specify what ID type was stored in the fasta file retrieved with biomartr::getGenome()
> result_BM
  refseq_peptide ensembl_gene_id ensembl_peptide_id
1      NP_000005 ENSG00000175899    ENSP00000323929
2      NP_000006 ENSG00000156006    ENSP00000286479
3      NP_000007 ENSG00000117054    ENSP00000359878
4      NP_000008 ENSG00000122971    ENSP00000242592
5      NP_000009 ENSG00000072778    ENSP00000349297
```

Gene Ontology

Il pacchetto *biomartr* consente anche un recupero rapido e intuitivo dei termini GO (Gene Ontology) e delle informazioni aggiuntive tramite la funzione `getGO()`. È possibile selezionare diversi database per recuperare informazioni di annotazione GO per un insieme di geni da interrogare. Ad ora, la funzione `getGO()` consente il recupero di informazioni GO dal database Ensembl *Biomart*.

Nell'esempio che segue recupereremo le informazioni GO per un insieme di geni *Homo sapiens* memorizzati come "hgnc_symbol".

La funzione `getGO()` ha diversi argomenti di input per recuperare informazioni da *BioMart*. In primo luogo, è necessario specificare il nome scientifico dell'organismo di interesse. Inoltre, è necessario specificare un insieme di ID di geni e la loro corrispondente notazione filtro (gli ID del gene GUCA2A hanno il simbolo "hgnc_symbol" di notazione filtro; vedere `organismFilters()` per i dettagli).

Come si vede, per ogni "gene ID" la tabella risultante mostra tutti i termini GO annotati (*accession*) presenti in Ensembl *Biomart*:

```
> # search for GO terms of an example Homo sapiens gene
> GO_tbl <- getGO(organism="Homo sapiens", genes="GUCA2A", filters="hgnc_symbol")
> GO_tbl
  hgnc_symbol                        goslim_goa_description goslim_goa_accession
1      GUCA2A                            biological_process           GO:0008150
2      GUCA2A                           signal transduction           GO:0007165
3      GUCA2A                            cellular_component           GO:0005575
4      GUCA2A                          extracellular region           GO:0005576
5      GUCA2A                            molecular_function           GO:0003674
6      GUCA2A                          biosynthetic process           GO:0009058
7      GUCA2A              small molecule metabolic process           GO:0044281
8      GUCA2A cellular nitrogen compound metabolic process           GO:0034641
9      GUCA2A                      enzyme regulator activity           GO:0030234
```

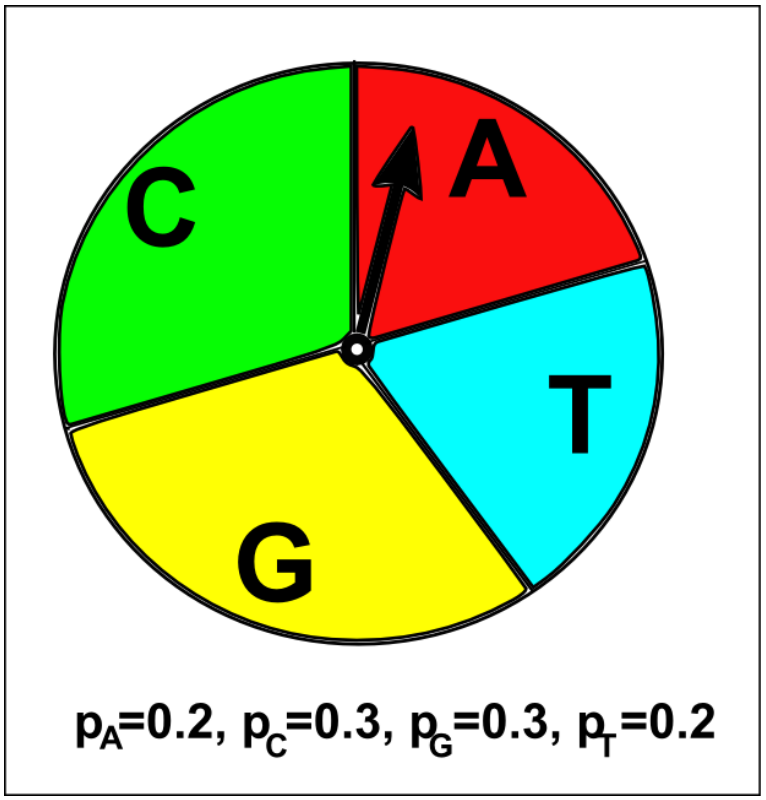
8. Modelli nascosti di Markov (*Hidden Markov Models*)

8.1 Un modello multinomiale di evoluzione di una sequenza di DNA

Il modello più semplice dell'evoluzione di una sequenza di DNA presuppone che la sequenza sia stata prodotta da un processo casuale che ha scelto a caso uno qualsiasi dei quattro nucleotidi in ogni posizione della sequenza, dove la probabilità dipende da una distribuzione predeterminata. Cioè, i quattro nucleotidi sono scelti rispettivamente con p_A, p_C, p_G e p_T. Questo è noto come modello di sequenza multinomiale (*multinomial sequence model*).

Un modello multinomiale per l'evoluzione di una sequenza di DNA ha quattro parametri: le probabilità dei quattro nucleotidi p_A, p_C, p_G e p_T. Ad esempio, diciamo di voler creare un modello multinomiale dove p_A=0,2, p_C=0,3, p_G=0,3 e p_T=0,2. Ciò significa che la probabilità di scegliere una A in una particolare posizione della sequenza è impostata a 0,2, la probabilità di scegliere una C è 0,3, la probabilità di scegliere una G è 0,3 e la probabilità di scegliere una T è 0,2. Si noti che $p_A + p_C + p_G + p_T = 1$ poiché la somma delle probabilità dei quattro diversi tipi di nucleotidi deve essere uguale a 1, essendovi solo quattro possibili tipi di nucleotidi.

Il modello di sequenza multinomiale equivale a una ruota della roulette divisa in quattro diversi settori etichettati "A", "T", "G" e "C", dove p_A, p_T, p_C e p_G sono le frazioni della ruota occupate dai settori con queste quattro etichette. Se si fa girare la freccia attaccata al centro della ruota della roulette, la probabilità che essa si fermi nel settore con una particolare etichetta (es. il settore etichettato "A") dipende solo dalla frazione della ruota presa da quel settore (p_A nell'esempio dell'immagine sotto riportata).



8.2 Generazione di una sequenza di DNA utilizzando un modello multinomiale

Possiamo usare R per generare una sequenza di DNA utilizzando un particolare modello multinomiale. Per prima cosa dobbiamo impostare i valori dei quattro parametri del modello multinomiale, le probabilità p_A, p_C, p_G e p_T di scegliere i nucleotidi A, C, G e T, rispettivamente, in una particolare posizione nella sequenza. Ad esempio, diciamo di decidere di impostare p_A=0,2, p_C=0,3, p_G=0,3 e p_T=0,2. Possiamo usare la funzione `sample()` di R per generare una sequenza di DNA di una certa lunghezza, selezionando un nucleotide in ogni posizione in base a questa distribuzione di probabilità:

```
> nucleotides <- c("A", "C", "G", "T") # Define the alphabet of nucleotides
> probabilities1 <- c(0.2, 0.3, 0.3, 0.2) # Set the values of the probabilities
> seqlength <- 30 # Set the length of the sequence
> sample(nucleotides, seqlength, rep=TRUE, prob=probabilities1) # Generate a sequence
 [1] "T" "C" "C" "C" "G" "C" "T" "T" "T" "C" "A" "C" "G" "C" "C" "C" "C" "A" "T" "A" "G" "A"
[23] "T" "G" "A" "A" "G" "G" "G" "G"
```

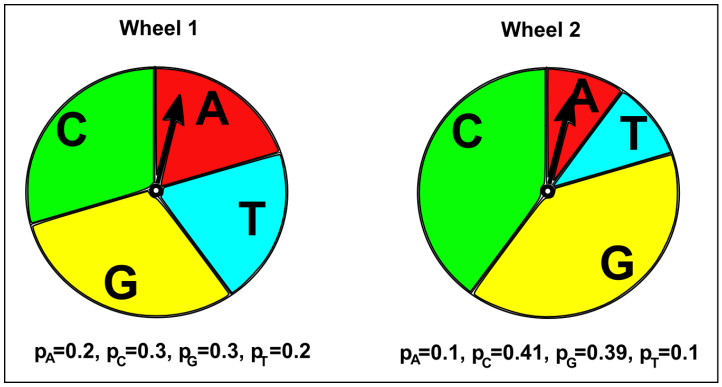
Gli input della funzione sono il vettore da cui prelevare il campione (*nucleotides*), la dimensione del campione (lunghezza della sequenza *seqlength*, nell'esempio 30) e il vettore di probabilità per ottenere gli elementi del vettore da campionare (*probabilities1*). Se usiamo la funzione `sample()` per generare di nuovo una sequenza, otterremo una sequenza diversa usando lo stesso modello multinomiale:

```
> sample(nucleotides, seqlength, rep=TRUE, prob=probabilities1) # Generate another sequence
 [1] "C" "A" "A" "A" "G" "T" "T" "T" "A" "G" "T" "G" "G" "T" "T" "G" "A" "G" "G" "A" "C" "T"
[23] "G" "T" "A" "C" "A" "A" "T" "C"
```

Allo stesso modo, possiamo generare una sequenza utilizzando un modello multinomiale diverso, dove p_A=0,1, p_C=0,41, p_G=0,39 e p_T=0,1:

```
> probabilities2 <- c(0.1, 0.41, 0.39, 0.1) # Set the values of the probabilities for the new model
> sample(nucleotides, seqlength, rep=TRUE, prob=probabilities2) # Generate a sequence
 [1] "G" "C" "G" "C" "C" "G" "G" "G" "C" "G" "G" "T" "C" "A" "G" "T" "G" "C" "G" "T" "C" "C"
[23] "G" "G" "G" "C" "T" "C" "C" "G"
```

Come ci si aspetterebbe, le sequenze generate utilizzando questo secondo modello multinomiale hanno una frazione maggiore di C e G rispetto alle sequenze generate utilizzando il primo modello multinomiale di cui sopra. Questo perché le probabilità p_C e p_G sono più alte per questo secondo modello rispetto al primo. Cioè, nel secondo modello multinomiale stiamo usando una "roulette" con settori grandi etichettati "C" e "G", mentre nel primo modello multinomiale stavamo usando una "roulette" con settori "C" e "G" relativamente più piccoli (vedi l'immagine qui accanto).



8.3 Un modello di Markov dell'evoluzione di una sequenza di DNA

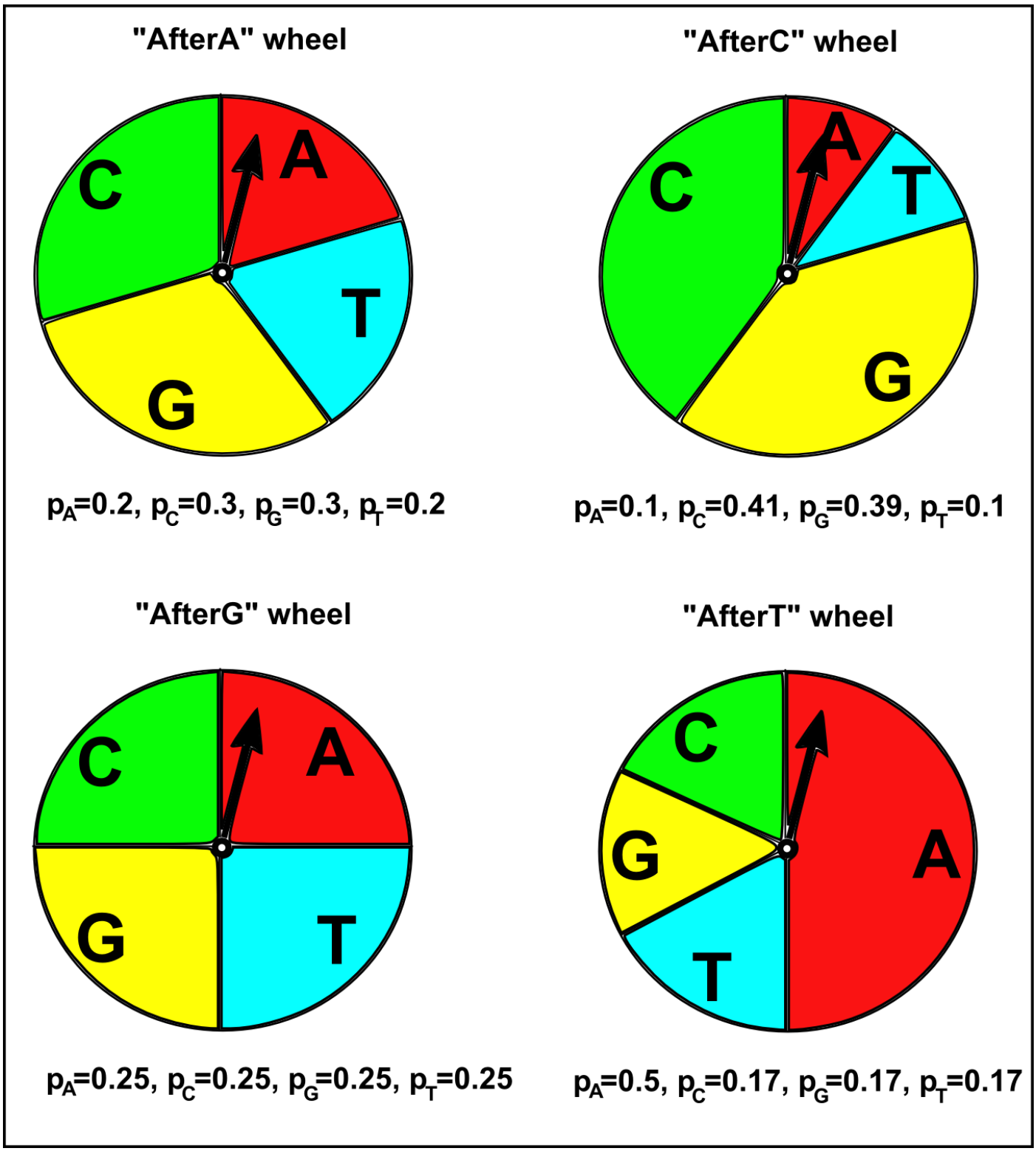
Un modello multinomiale di evoluzione delle sequenze di DNA è un buon modello dell'evoluzione di molte sequenze di DNA. Tuttavia, per alcune sequenze un modello multinomiale non è una rappresentazione accurata di come le sequenze si sono evolute. Una ragione è che un modello multinomiale presuppone che ogni parte della sequenza (ad es. i primi 100 nucleotidi della sequenza, i secondi 100 nucleotidi, i terzi 100 nucleotidi, ecc.) abbiano la stessa frequenza di ogni tipo di nucleotide (la stessa p_A, p_C, p_G e p_T), e questo può non essere vero per una particolare sequenza di DNA se ci sono notevoli differenze nelle frequenze dei nucleotidi in diverse parti della sequenza.

Un'altra ipotesi di un modello multinomiale di evoluzione della sequenza di DNA è che la probabilità di scegliere un particolare nucleotide (ad es. "A") in una particolare posizione nella sequenza dipende solo dalla frequenza predeterminata di quel nucleotide (cioè p_A), e non dipende affatto dai nucleotidi che si trovano in posizioni adiacenti nella sequenza. Questa ipotesi vale per molte sequenze di DNA. Tuttavia, per alcune sequenze non è vero, perché la probabilità di trovare un particolare nucleotide in una particolare posizione nella sequenza dipende da quali nucleotidi si trovano in posizioni adiacenti nella sequenza. In questo caso, un diverso modello di sequenza di DNA chiamato modello di sequenza di Markov (*Markov sequence model*) è una rappresentazione più accurata dell'evoluzione della sequenza.

Un modello di sequenza di Markov presuppone che la sequenza sia stata prodotta da un processo che ha scelto uno qualsiasi dei quattro nucleotidi della sequenza, dove la probabilità di scegliere uno qualsiasi dei quattro nucleotidi in una particolare posizione dipende dal nucleotide scelto per la posizione precedente. Cioè, se "A" è stato scelto nella posizione precedente, allora la probabilità di scegliere uno qualsiasi dei quattro nucleotidi nella posizione attuale dipende da una distribuzione di probabilità predeterminata. In altre parole, dato che "A" è stato scelto nella posizione precedente, i quattro nucleotidi "A", "C", "G", o "T" sono scelti nella posizione corrente con probabilità, rispettivamente, ad es. p_A=0,2, p_C=0,3, p_G=0,3 e p_T=0,2. Al contrario, se "C" è stato scelto nella posizione precedente, allora la probabilità di scegliere uno qualsiasi dei quattro nucleotidi nella posizione attuale dipende da una diversa distribuzione di probabilità predeterminata (ad es. p_A=0,1, p_C=0,41, p_G=0,39 e p_T=0,1).

Un modello di sequenza di Markov è come avere quattro diverse roulette, etichettate "dopo_A", "dopo_T", "dopo_G" e "dopo_C", per i casi in cui "A", "T", "G" o "C" siano state rispettivamente scelte nella posizione precedente della sequenza. Ognuna delle quattro ruote ha quattro settori etichettati "A", "T", "G" e "C", ma in ogni ruota una diversa frazione è occupata dai quattro settori. Cioè, ogni ruota della roulette ha una diversa p_A, p_T, p_G e p_C. Se stiamo generando una nuova sequenza di DNA usando un modello di sequenza di Markov per decidere quale nucleotide scegliere in una particolare posizione della sequenza, si fa girare la freccia al centro di una ruota della roulette, e si vede in quale settore si ferma la freccia. Ci sono quattro roulette, e la particolare ruota che usiamo in una particolare posizione nella sequenza dipende dal nucleotide scelto per la posizione precedente nella sequenza. Per esempio, se "T" è stato scelto nella posizione precedente, usiamo la ruota della roulette "dopo_T" per scegliere il nucleotide per la posizione corrente. La probabilità di scegliere un particolare nucleotide nella posizione corrente (ad es. "A") dipende quindi dalla

frazione della ruota "dopo_T" presa dal settore etichettato con quel nucleotide (ad es. p_A; vedi l'immagine sotto).



8.4 La matrice di transizione per un modello di Markov

Un modello multinomiale di evoluzione delle sequenze di DNA ha solo quattro parametri: le probabilità p_A, p_C, p_G e p_T. Al contrario, un modello di Markov ha molti più parametri: quattro serie di probabilità p_A, p_C, p_G e p_T, che differiscono a seconda che il precedente nucleotide fosse "A", "C", "G" o "T". I simboli p_{AA}, p_{AC},

p_{AG} e p_{AT} sono solitamente usati per rappresentare le quattro probabilità per il caso in cui il nucleotide precedente fosse "A", i simboli p_{CA}, p_{CC}, p_{CG} e p_{CT} per il caso in cui il nucleotide precedente fosse "C" e così via.

È comune memorizzare i parametri di probabilità per un modello di Markov di una sequenza di DNA in una matrice quadrata, nota come *matrice di transizione* di Markov. Le righe della matrice di transizione rappresentano il nucleotide trovato nella posizione precedente nella sequenza, mentre le colonne rappresentano i nucleotidi che potrebbero essere trovati nella posizione corrente nella sequenza. In R, è possibile creare una matrice utilizzando il comando `matrix()`; le funzioni `rownames()` e `colnames()` possono essere utilizzate per etichettare le righe e le colonne della matrice. Ad esempio, per creare una matrice di transizione, possiamo digitare:

```
> nucleotides <- c("A", "C", "G", "T") # Define the alphabet of nucleotides
> afterAprobs <- c(0.2, 0.3, 0.3, 0.2) # Set the values of the probabilities beign "A" the previous nucleotide
> afterCprobs <- c(0.1, 0.41, 0.39, 0.1) # Set the values of the probabilities beign "C" the previous nucleotide
> afterGprobs <- c(0.25, 0.25, 0.25, 0.25) # Set the values of the probabilities beign "G" the previous nucleotide
> afterTprobs <- c(0.5, 0.17, 0.17, 0.17) # Set the values of the probabilities beign "T" the previous nucleotide
> mytransitionmatrix <- matrix(c(afterAprobs, afterCprobs, afterGprobs, afterTprobs),
      4, 4, byrow = TRUE) # Create a 4x4 matrix
> rownames(mytransitionmatrix) <- nucleotides
> colnames(mytransitionmatrix) <- nucleotides
> mytransitionmatrix # Print out the transition matrix
     A    C    G    T
A 0.20 0.30 0.30 0.20
C 0.10 0.41 0.39 0.10
G 0.25 0.25 0.25 0.25
T 0.50 0.17 0.17 0.17
```

Le righe 1, 2, 3 e 4 della matrice di transizione danno le probabilità p_A, p_C, p_G e p_T per i casi in cui il precedente nucleotide fosse rispettivamente "A", "C", "G" o "T". Cioè, l'elemento in una particolare riga e colonna della matrice di transizione (ad es. 0,30 nella riga "A" e colonna "C") contiene la probabilità (p_{AC}) di scegliere un particolare nucleotide ("C") nella posizione corrente nella sequenza, dato un particolare nucleotide ("A") nella posizione precedente della sequenza.

8.5 Generare una sequenza di DNA utilizzando un modello di Markov

Così come è possibile generare una sequenza di DNA utilizzando un particolare modello multinomiale, è possibile generare una sequenza di DNA utilizzando un particolare modello di Markov. Quando si genera una sequenza di DNA usando un modello di Markov, il nucleotide scelto in ogni posizione della sequenza dipende dal nucleotide scelto nella posizione precedente. Poiché non c'è un nucleotide precedente nella prima posizione della nuova sequenza, dobbiamo definire le probabilità di scegliere "A", "C", "G" o "T" per la prima posizione. I simboli Π_A, Π_C, Π_G e Π_T sono usati per rappresentare le probabilità di scegliere "A", "C", "G" o "T" per la prima posizione.

La funzione `generatemarkovseq()` (vedi Appendice) genera una sequenza di DNA usando un particolare modello di Markov. Prende come argomenti la matrice di transizione per il particolare modello di Markov, un vettore contenente i valori Π_A, Π_C, Π_G e Π_T e la lunghezza della sequenza di DNA da generare.

Le probabilità di scegliere ciascuno dei quattro nucleotidi in una posizione successiva alla prima dipendono dal nucleotide scelto nella posizione precedente.

Possiamo usare la funzione `generatemarkovseq()` per generare una sequenza usando un particolare modello di Markov. Per esempio, per creare una sequenza di 30 nucleotidi usando il modello di Markov descritto nella matrice di transizione *mytransitionmatrix*, usando probabilità di partenza uniformi (ad es. $\Pi_A = \Pi_C = \Pi_G = \Pi_T = 0{,}25$), digitiamo:

```
> myinitialprobs <- c(0.25, 0.25, 0.25, 0.25)
> generatemarkovseq(mytransitionmatrix, myinitialprobs, 30)
 [1] "C" "C" "G" "A" "C" "G" "G" "G" "G" "A" "T" "A" "T" "C" "C" "G" "T" "A" "A" "G" "T" "A"
[23] "G" "T" "G" "A" "C" "C" "T" "A"
```

Come si può vedere, ci sono molte "A" dopo le "T" nella sequenza. Questo perché la p_{TA} ha un valore elevato (0,5) nella matrice di transizione di Markov *mytransitionmatrix*. Allo stesso modo, ci sono poche "A" o "T" dopo le "C", il che è dovuto al fatto che p_{CA} e p_{CT} hanno valori bassi (0,1).

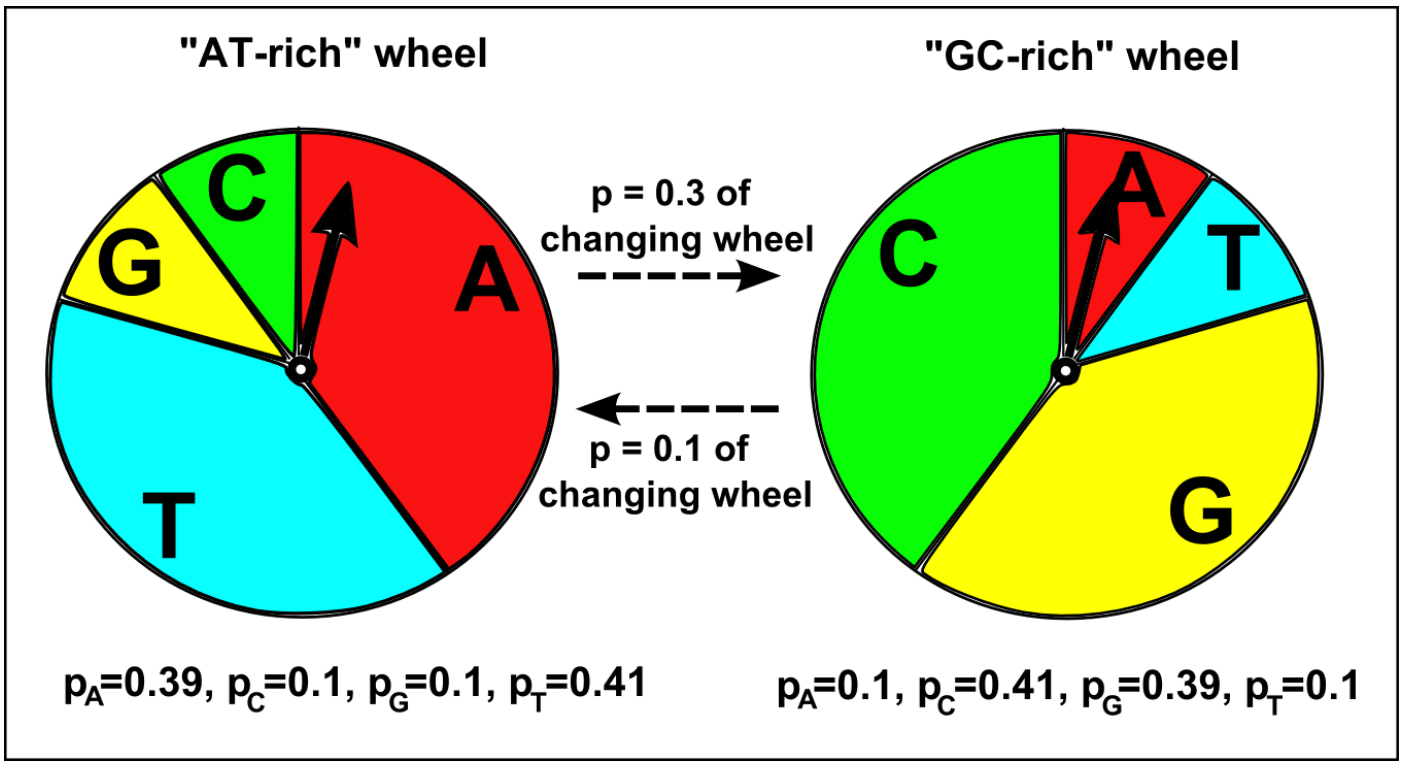
8.6 Un *Hidden Markov Model* dell'evoluzione di una sequenza di DNA

In un modello di Markov, il nucleotide in una particolare posizione in una sequenza dipende dal nucleotide trovato nella posizione precedente. Al contrario, in un modello di Markov nascosto (HMM), il nucleotide trovato in una particolare posizione in una sequenza dipende dallo stato del nucleotide precedente nella sequenza. Lo stato in una posizione della sequenza è una proprietà di quella posizione della sequenza; ad esempio, un particolare HMM può modellare le posizioni lungo una sequenza come appartenenti ad uno dei due stati, "GC-denso" o "AT-denso". Un HMM più complesso può modellare le posizioni lungo una sequenza come appartenenti a diversi stati possibili, come "promotore", "esone", "introne" e "DNA intergenico".

Un HMM è come avere diverse roulette, una per ogni stato dell'HMM: ad esempio, una roulette "GC-densa" e una roulette "AT-densa". Ciascuna delle ruote della roulette ha quattro settori etichettati "A", "T", "G" e "C", e in ogni ruota una diversa frazione della ruota è occupata dai quattro settori. Cioè, le ruote delle roulette "GC-densa" e "AT-densa" hanno valori diversi di p_A, p_C, p_G e p_T. Se stiamo generando una nuova sequenza di DNA usando un HMM, per decidere quale nucleotide scegliere in una particolare posizione della sequenza facciamo girare la freccia di una particolare ruota della roulette e vediamo in quale settore si ferma.

Come decidiamo quale roulette utilizzare? Se ci sono due roulette, tendiamo ad usare la stessa roulette che abbiamo usato per scegliere il nucleotide precedente nella sequenza; ma c'è anche una certa piccola probabilità di passare all'altra roulette. Per esempio, se abbiamo usato la ruota della roulette "GC-densa" per scegliere il nucleotide precedente nella sequenza, ci può essere una probabilità del 90% che useremo di nuovo la stessa roulette per scegliere il nucleotide nella posizione corrente, ma una probabilità del 10% che passeremo ad usare la roulette "AT-densa". Allo stesso modo, se abbiamo usato la roulette "AT-densa" per scegliere il nucleotide nella posizione precedente, ci potrebbe essere il 70% di probabilità che useremo di

nuovo la stessa roulette, ma il 30% di probabilità che passeremo ad usare la roulette "GC-densa" per scegliere il nucleotide nella posizione corrente.



8.7 La matrice di transizione e la matrice di emissione di un HMM

Un HMM ha due importanti matrici che contengono i suoi parametri. La prima è la *matrice di transizione* HMM, che contiene le probabilità di passare da uno stato all'altro. Per esempio, in un HMM con due stati, uno stato AT-denso e uno stato GC-denso, la matrice di transizione conterrà le probabilità di passare dallo stato AT-denso allo stato GC-denso e viceversa. Ad esempio, se il nucleotide precedente era nello stato AT-denso ci può essere una probabilità di 0,3 che il nucleotide attuale sia nello stato GC-denso; e se il nucleotide precedente era nello stato GC-denso ci può essere una probabilità di 0,1 che il nucleotide attuale sia nello stato AT-denso:

```
> states <- c("AT-rich", "GC-rich") # Define the names of the states
> ATrichprobs <- c(0.7, 0.3) # Set the probabilities of switching states, where the previous state was "AT-rich"
> GCrichprobs <- c(0.1, 0.9) # Set the probabilities of switching states, where the previous state was "GC-rich"
> thetransitionmatrix <- matrix(c(ATrichprobs, GCrichprobs), 2, 2, byrow = TRUE) #
Create a 2x2 matrix
> rownames(thetransitionmatrix) <- states
> colnames(thetransitionmatrix) <- states
> thetransitionmatrix # Print out the transition matrix
        AT-rich GC-rich
AT-rich     0.7     0.3
GC-rich     0.1     0.9
```

C'è una riga nella matrice di transizione per ciascuno dei possibili stati nella posizione precedente nella sequenza nucleotidica. Per esempio, in questa matrice di transizione, la prima riga corrisponde al caso in cui la posizione precedente era nello stato "AT-denso", e la seconda riga corrisponde al caso in cui la posizione

precedente era nello stato "GC-denso". Le colonne danno le probabilità di passare a diversi stati nella posizione corrente. Per esempio, il valore nella seconda riga e nella prima colonna della matrice di transizione di cui sopra è 0,1, che è la probabilità di passare allo stato AT-denso se la posizione precedente della sequenza era nello stato GC-denso.

La seconda matrice importante è la *matrice di emissione* HMM, che contiene le probabilità di scegliere i quattro nucleotidi "A", "C", "G" e "T", in ciascuno degli stati. In un HMM con uno stato AT-denso e uno stato GC-denso, la matrice di emissione conterrà la probabilità di scegliere ciascuno dei quattro nucleotidi "A", "C", "G" e "T" nello stato AT-denso (per esempio, p_A=0,39, p_C=0,1, p_G=0,1 e p_T=0,41 per lo stato AT-denso), e le probabilità di scegliere "A", "C", "G" e "T" nello stato GC-denso (per esempio, p_A=0,1, p_C=0,41, p_G=0,39 e p_T=0,1 per lo stato GC-denso).

```
> nucleotides <- c("A", "C", "G", "T")
> ATrichstateprobs <- c(0.39, 0.1, 0.1, 0.41)
> GCrichstateprobs <- c(0.1, 0.41, 0.39, 0.1)
> theemissionmatrix <- matrix(c(ATrichstateprobs, GCrichstateprobs), 2, 4,
      byrow = TRUE)
> rownames(theemissionmatrix) <- states
> colnames(theemissionmatrix) <- nucleotides
> theemissionmatrix
           A    C    G    T
AT-rich 0.39 0.10 0.10 0.41
GC-rich 0.10 0.41 0.39 0.10
```

C'è una riga nella matrice di emissione per ogni possibile stato, e le colonne danno la probabilità di scegliere ciascuno dei quattro possibili nucleotidi quando ci si trova in un particolare stato. Per esempio, il valore nella seconda riga e nella terza colonna della matrice di emissione di cui sopra è 0,39, che corrisponde alla probabilità di scegliere una "G" quando ci si trova nello stato "GC-denso" (cioè quando si usa la roulette "GC-densa").

8.8 Generare una sequenza di DNA utilizzando un HMM

La funzione `generatehmmseq()` (vedi Appendice) può essere utilizzata per generare una sequenza di DNA utilizzando un particolare HMM. Come suoi argomenti richiede i parametri dell'HMM: la *matrice di transizione* e la *matrice di emissione*.

Quando si genera una sequenza di DNA utilizzando un HMM, il nucleotide viene scelto in ogni posizione a seconda dello stato nella posizione precedente della sequenza. Poiché non esiste un nucleotide precedente nella prima posizione della sequenza, la funzione `generatehmmseq()` richiede anche le probabilità di scegliere ciascuno degli stati nella prima posizione (ad es. $\Pi_{\text{AT-denso}}$ e $\Pi_{\text{GC-denso}}$ essendo la probabilità di scegliere gli stati "AT-denso" o "GC-denso" nella prima posizione per un HMM con questi due stati).

Possiamo usare la funzione `generatehmmseq()` per generare una sequenza utilizzando un particolare HMM. Ad esempio, per creare una sequenza di 30 nucleotidi utilizzando l'HMM con gli stati "AT-denso" e "GC-denso" descritti nella matrice di transizione *thetransitionmatrix*, la matrice di emissione *theemissionmatrix* e le probabilità di partenza uniformi (cioè $\Pi_{\text{AT-denso}} = 0{,}5$, $\Pi_{\text{GC-denso}} = 0{,}5$), digitiamo:

```
> theinitialprobs <- c(0.5, 0.5)
> generatehmmseq(thetransitionmatrix, theemissionmatrix, theinitialprobs, 30)
[1] "Position 1 , State GC-rich , Nucleotide =  C"
[1] "Position 2 , State GC-rich , Nucleotide =  G"
[1] "Position 3 , State GC-rich , Nucleotide =  G"
[1] "Position 4 , State AT-rich , Nucleotide =  A"
[1] "Position 5 , State GC-rich , Nucleotide =  C"
[1] "Position 6 , State GC-rich , Nucleotide =  G"
[1] "Position 7 , State GC-rich , Nucleotide =  A"
[1] "Position 8 , State GC-rich , Nucleotide =  G"
[1] "Position 9 , State GC-rich , Nucleotide =  C"
[1] "Position 10 , State GC-rich , Nucleotide =  G"
[1] "Position 11 , State GC-rich , Nucleotide =  C"
[1] "Position 12 , State GC-rich , Nucleotide =  C"
[1] "Position 13 , State AT-rich , Nucleotide =  A"
[1] "Position 14 , State GC-rich , Nucleotide =  A"
[1] "Position 15 , State AT-rich , Nucleotide =  T"
[1] "Position 16 , State AT-rich , Nucleotide =  A"
[1] "Position 17 , State AT-rich , Nucleotide =  T"
[1] "Position 18 , State AT-rich , Nucleotide =  A"
[1] "Position 19 , State GC-rich , Nucleotide =  A"
[1] "Position 20 , State GC-rich , Nucleotide =  G"
[1] "Position 21 , State GC-rich , Nucleotide =  G"
[1] "Position 22 , State GC-rich , Nucleotide =  T"
[1] "Position 23 , State GC-rich , Nucleotide =  G"
[1] "Position 24 , State GC-rich , Nucleotide =  C"
[1] "Position 25 , State AT-rich , Nucleotide =  T"
[1] "Position 26 , State AT-rich , Nucleotide =  C"
[1] "Position 27 , State GC-rich , Nucleotide =  T"
[1] "Position 28 , State AT-rich , Nucleotide =  C"
[1] "Position 29 , State GC-rich , Nucleotide =  C"
[1] "Position 30 , State GC-rich , Nucleotide =  C"
```

Come si può vedere, i nucleotidi generati dallo stato GC-denso sono per lo più, ma non tutti, "G" e "C" (a causa degli alti valori di p_G e p_C per lo stato GC-denso nella matrice di emissione HMM), mentre i nucleotidi generati dallo stato AT-denso sono per lo più, ma non tutti, "A" e "T" (a causa degli alti valori di p_T e p_A per lo stato AT-denso nella matrice di emissione HMM).

Inoltre, tendono ad esserci sequenze di nucleotidi che sono tutte nello stesso stato, poiché la matrice di transizione specifica che le probabilità di passare dallo stato AT-denso allo stato GC-denso (probabilità 0.3) o al contrario (probabilità 0.1) sono relativamente basse.

8.9 Inferire gli stati di un HMM che ha generato una sequenza di DNA

Se abbiamo un HMM con due stati, "GC-denso" e "AT-denso", e conosciamo le matrici di transizione e di emissione dell'HMM, possiamo prendere una nuova sequenza di DNA e capire quale stato (GC-denso o AT-denso) è il più probabile che abbia generato ogni posizione nucleotidica in quella sequenza di DNA? Questo è un problema comune nella bioinformatica, cioè il problema di trovare il "percorso di stato più probabile" (*most probable state path*) in quanto consiste essenzialmente nell'assegnare lo stato più probabile ad ogni posizione nella sequenza di DNA. Il problema di trovare il percorso di stato più probabile è anche chiamato a volte *segmentazione*. Per esempio, data una sequenza di DNA di 1.000 nucleotidi, si potrebbe

voler usare l'HMM per segmentare la sequenza in blocchi che sono stati probabilmente generati dallo stato "GC-denso" o dallo stato "AT-denso".

Il problema di trovare il percorso di stato più probabile dato un HMM e una sequenza (cioè il problema di *segmentare* una sequenza usando un HMM), può essere risolto con l'algoritmo di *Viterbi*. Come suo risultato, l'algoritmo fornisce per ogni posizione nucleotidica in una sequenza di DNA lo stato dell'HMM che molto probabilmente ha generato il nucleotide in quella posizione. Per esempio, se si segmenta una particolare sequenza di DNA di 1.000 nucleotidi usando un HMM con stati "AT-denso" e "GC-denso", l'algoritmo di Viterbi può dire che i nucleotidi 1–343 sono stati probabilmente generati dallo stato AT-denso, i nucleotidi 344–900 sono stati probabilmente generati dallo stato GC-denso e i nucleotidi 901–1.000 sono stati probabilmente generati dallo stato AT-denso.

La funzione `viterbi()` (vedi Appendice) implementa l'algoritmo di Viterbi e fa uso della funzione `makeViterbimat()` (vedi Appendice). Dato un HMM, e una particolare sequenza di DNA, è possibile utilizzare la funzione `viterbi()` per trovare lo stato dell'HMM che più probabilmente ha generato il nucleotide in ogni posizione della sequenza di DNA:

```
> myseq <- c("A", "A", "G", "C", "G", "T", "G", "G", "G", "G", "C", "C", "C", "C",
"G", "G", "C", "G", "A", "C", "A", "T", "G", "G", "G", "G", "T", "G", "T", "C")
> viterbi(myseq, thetransitionmatrix, theemissionmatrix)
[1] "Positions 1 - 2 Most probable state =  AT-rich"
[1] "Positions 3 - 21 Most probable state =  GC-rich"
[1] "Positions 22 - 22 Most probable state =  AT-rich"
[1] "Positions 23 - 30 Most probable state =  GC-rich"
```

8.10 Un *Hidden Markov Model* dell'evoluzione di una sequenza proteica

Finora abbiamo parlato di usare gli HMM per modellare l'evoluzione di sequenze di DNA. Tuttavia, è naturalmente possibile utilizzare HMMs per modellare l'evoluzione di sequenze proteiche. Quando si usa un HMM per modellare l'evoluzione di una sequenza di DNA, possiamo avere stati come "AT-denso" e "GC-denso". Allo stesso modo, quando si utilizza un HMM per modellare l'evoluzione di una sequenza proteica, possiamo avere stati come "idrofobo" e "idrofilo". In una proteina con stati "idrofobo" e "idrofilo", lo stato "idrofobo" avrà probabilità p_A, p_R, p_C ... di scegliere ciascuno dei 20 aminoacidi alanina (A), arginina (R), cisteina (C), ecc. in quello stato. Allo stesso modo, lo stato "idrofilo" avrà diverse probabilità p_A, p_R, p_C ... di scegliere ciascuno dei 20 aminoacidi. La probabilità di scegliere un aminoacido idrofobico come l'alanina sarà più alta nello stato "idrofobo" che nello stato "idrofilo" (cioè p_A dello stato "idrofobo" sarà più alta di p_A dello stato "idrofilo", dove A rappresenta l'alanina). Un HMM dell'evoluzione della sequenza proteica definisce anche una certa probabilità di passare dallo stato "idrofilo" allo stato "idrofobo" e viceversa.

9. Grafi di interazione proteica

9.1 I grafi in R

Nei capitoli precedenti abbiamo utilizzato il software di statistica R per l'analisi di molti tipi diversi di dati. In questa parte, utilizzeremo R per l'analisi dei dati di interazione proteica. Tuttavia, prima esamineremo alcune caratteristiche di R che saranno utili allo scopo.

Una cosa utile da sapere in R è che molti pacchetti vengono forniti con dataset di esempio, che possono essere utilizzati per familiarizzare con le relative funzioni. Per elencare i dataset che vengono forniti con un particolare pacchetto R, si può usare la funzione `data()`. Ad esempio, per trovare i dataset forniti con il pacchetto *graph*, digitare:

```
> library("graph")
> data(package = "graph")
Data sets in package 'graph':
IMCAAttrs (integrinMediatedCellAdhesion)
                            KEGG Integrin Mediated Cell Adhesion graph
IMCAGraph (integrinMediatedCellAdhesion)
                            KEGG Integrin Mediated Cell Adhesion graph
MAPKsig                     A graph encoding parts of the MAPK signaling pathway
MAPKsig (defunctGraph)      A graph encoding parts of the MAPK signaling pathway
apopGraph                   KEGG apoptosis pathway graph
biocRepos                   A graph representing the Bioconductor package
                            repository
esetsFemale                 MultiGraph edgeSet data
esetsMale                   MultiGraph edgeSet data
graphExamples               A List Of Example Graphs
pancrCaIni                  A graph encoding parts of the pancreatic cancer
                            initiation pathway
```

È quindi possibile caricare uno qualsiasi di questi dataset in R digitando, ad esempio, per caricare il dataset "apopGraph":

```
> data("apopGraph")
```

Utilizzeremo inoltre la funzione `table()` per creare tabelle di dati memorizzati in vettori. Se si dispone di un vettore contenente valori numerici, la funzione `table()` è utile per creare una tabella che dice quanti elementi nel vettore hanno ciascuno dei valori. Per esempio:

```
> y <- c(10, 10, 20, 20, 20, 20, 20, 30) # Make a numeric vector "y"
> table(y)
y
10 20 30
 2  5  1
```

I risultati della funzione `table()` ci dicono che due degli elementi nel vettore *y* hanno il valore 10, cinque elementi hanno il valore 20 e un elemento ha il valore 30.

Un altro uso della funzione `table()` è trovare quanti elementi in un vettore numerico hanno un particolare valore. Per esempio, se vogliamo sapere quanti elementi nel vettore *y* hanno il valore 20, possiamo digitare:

```
> table(y == 20)
FALSE  TRUE
    3     5
```

Il risultato ci dice che cinque elementi del vettore *y* hanno il valore 20.

In questo capitolo utilizzeremo funzioni di diversi pacchetti, che però a volte hanno lo stesso nome. Ad esempio, c'è una funzione chiamata `degree()` in entrambi i pacchetti *igraph* e *graph*. Pertanto, è necessario specificare quale funzione si vuole usare anteponendo al suo nome quello del pacchetto, seguito da `"::"`. Ad esempio, per usare la funzione `degree()` del pacchetto *graph* si può digitare `graph::degree()`, mentre per usare la funzione `degree()` del pacchetto *igraph* occorre digitare `igraph::degree()`.

9.2 Grafi per i dati di interazione proteica in R

I dati di interazione proteica possono essere descritti in termini di grafi. In questa sezione esploreremo una serie di dati curata di interazioni proteina-proteina, utilizzando pacchetti R per l'analisi e la visualizzazione di grafi.

Utilizzeremo tre pacchetti principali che sono stati scritti per la gestione dei grafi biologici: il pacchetto *graph*, il pacchetto *RBGL* e il pacchetto *Rgraphviz*. Questi tre pacchetti fanno parte di Bioconductor, quindi devono essere installati di conseguenza (vedere la pagina Web di Bioconductor all'indirizzo https://master.bioconductor.org/install/ per i dettagli).

Per prima cosa analizzeremo un dataset curato di interazioni proteiche nel lievito *Saccharomyces cerevisiae* estratto da documenti pubblicati. Questo dataset proviene da un pacchetto R chiamato *yeastExpData*, che contiene il dataset "litG". Questi dati sono stati descritti per la prima volta in un articolo di Ge *et al* (2001) in *Nature Genetics* (https://www.nature.com/ng/journal/v29/n4/full/ng776.html).

Per leggere il dataset "litG" in R, dobbiamo prima caricare il pacchetto *yeastExpData*, e poi possiamo usare la funzione `data()` per leggere il dataset:

```
> BiocManager::install("yeastExpData")
> library(yeastExpData)
> data(litG)
```

Quando si legge il dataset "litG" mediante la funzione `data()`, esso viene memorizzato come grafo in R. In questo grafo, i vertici (nodi) sono proteine, e le connessioni tra i vertici indicano che due proteine interagiscono. Nel grafo ci sono 2.885 vertici, che rappresentano 2.885 proteine diverse.

È possibile visualizzare il numero di vertici e archi (connessioni) in un grafo semplicemente digitandone il nome, ad esempio:

```
> litG
A graphNEL graph with undirected edges
Number of Nodes = 2885
Number of Edges = 315
```

Questo indica che il grafo "litG" ha 2.885 vertici e 315 archi. I 315 archi nel grafo rappresentano 315 interazioni tra 315 coppie di proteine.

9.3 Trovare i nomi dei vertici nei grafi per i dati di interazione proteica

Il pacchetto *graph* di R contiene molte funzioni per l'analisi dei dati di un grafo. Ad esempio, la funzione `nodes()` del pacchetto può essere usata per recuperare i nomi dei vertici (nodi) del grafo. Ad esempio, possiamo recuperare i nomi dei vertici nel grafo "litG" e memorizzarli in un vettore *mynodes* digitando:

```
> library("graph")          # Load the graph package
> mynodes <- nodes(litG)    # Retrieve the names of the vertices in the litG graph
```

Possiamo quindi visualizzare i nomi dei primi 10 vertici del grafo digitando:

```
> mynodes[1:10]             # Print the names of the first 10 vertices in the litG␣
↪graph
 [1] "YBL072C" "YBL083C" "YBR009C" "YBR010W" "YBR031W" "YBR093C" "YBR106W"
 [8] "YBR118W" "YBR188C" "YBR191W"
```

Questo fornisce i nomi delle proteine del lievito corrispondenti ai primi 10 vertici del grafo "litG". Si noti che l'ordine in cui le proteine sono memorizzate nel grafo non ha alcun significato; queste 10 proteine sono semplicemente le prime 10 memorizzate nel grafo litG. Poiché *mynodes* è un vettore che contiene un elemento per ogni vertice nel grafo "litG", il numero di elementi di *mynodes* dovrebbe essere uguale al numero di vertici nel grafo:

```
> length(mynodes)           # Find the number of vertices in the litG graph
[1] 2885
```

9.4 Trovare i nomi delle proteine con cui una particolare proteina interagisce

Se interessa una particolare proteina in un grafo di interazione proteica, si può voler visualizzare l'elenco delle proteine con cui quella proteina interagisce. Per fare questo, possiamo usare la funzione `adj()` del pacchetto *graph*. Ad esempio, per vedere le proteine con cui la proteina YBR009C interagisce nel grafo "litG", è possibile digitare:

```
> adj(litG, "YBR009C")
$YBR009C
[1] "YBR010W" "YNL031C" "YDR227W"
```

Questo ci dice che la proteina YBR009C interagisce con altre tre proteine nel grafo "litG", cioè YBR010W, YNL031C e YDR227W.

9.5 Calcolo della distribuzione dei gradi per un grafo in R

Il grado di un vertice (nodo) in un grafo è il numero di connessioni che ha con gli altri vertici del grafo. La distribuzione dei gradi per un grafo è la distribuzione dei valori dei gradi per tutti i suoi vertici, cioè il numero di vertici nel grafo che hanno grado 0, 1, 2, 3, ecc.

In termini di un grafo di interazione proteica, ogni vertice nel grafo rappresenta una proteina, e il grado di un particolare vertice è il numero di interazioni che quella proteina ha con altre proteine.

È possibile calcolare i gradi di tutti i vertici di un grafo utilizzando la funzione `degree()` del pacchetto *graph*. La funzione restituisce un vettore contenente i gradi di ciascuno dei vertici del grafo. Poiché c'è una funzione `degree()` in entrambi i pacchetti *graph* e *igraph*, se sono stati caricati entrambi i pacchetti occorre specificare che si vuole usare la funzione `degree()` del pacchetto *graph* scrivendo `graph::degree()`.

Ad esempio, per calcolare i gradi dei vertici nel grafo "litG" digitiamo:

```
> mydegrees <- graph::degree(litG)
> mydegrees # Print out the values in the vector "mydegrees"
  YBL072C   YBL083C   YBR009C   YBR010W   YBR031W   YBR093C   YBR106W   YBR118W
        0         0         3         3         0         0         0         2
```

Vediamo che la proteina del lievito YBL072C non interagisce con altre proteine, mentre la proteina del lievito YBR118W interagisce con altre due. Viene mostrata solo la prima riga dei risultati, poiché ci sono 2.885 vertici nel grafo "litG".

È possibile ordinare il vettore *mydegrees* utilizzando la funzione `sort()`:

```
> sort(mydegrees)
  YBL072C   YBL083C   YBR031W   YBR093C   YBR106W   YBR188C   YBR191W   YBR206W
↪YCL007C   YCL018W
        0         0         0         0         0         0         0         0
↪       0         0
...
  YBR198C   YDR392W   YDR448W   YBR160W   YFL039C
        8         8         9        10        12
```

Vengono mostrate solo la prima e l'ultima riga dell'output. Si può vedere dall'ultima riga che ci sono alcuni vertici che hanno gradi elevati. Per esempio, il vertice corrispondente alla proteina YFL039C ha grado 12. Ciò significa che la proteina YFL039C interagisce con altre 12 proteine. Le proteine altamente connesse in un grafo di interazione proteica sono a volte chiamati proteine *hub*.

Possiamo calcolare la distribuzione dei gradi per un grafo utilizzando la funzione `table()` per fare una tabella di quanti vertici nel grafo hanno grado 0, 1, 2, 3, ecc. Ad esempio, per calcolare la distribuzione dei gradi per il grafo "litG" possiamo digitare:

```
> table(mydegrees)
mydegrees
   0    1    2    3    4    5    6    7    8    9   10   12
2587  159   58   38   19    7    3    7    4    1    1    1
```

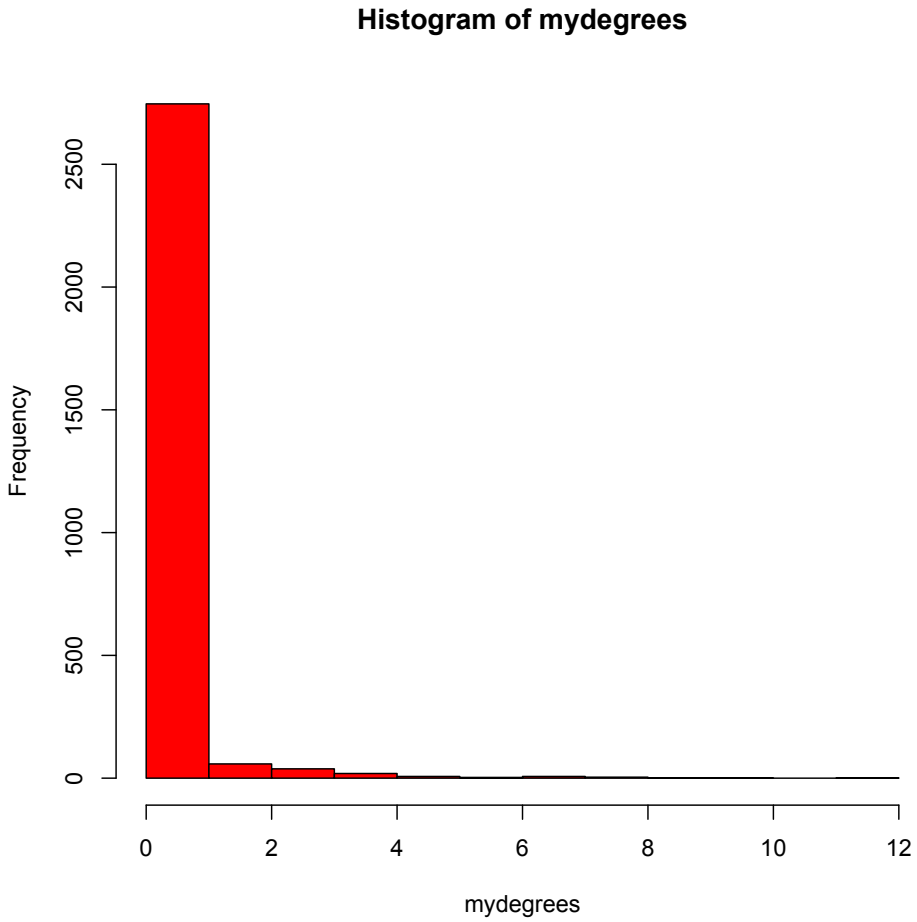
Questo indica che 2.587 vertici nel grafo "litG" non sono collegati ad altri vertici, 159 vertici sono collegati ad un altro vertice, 58 vertici sono collegati ad altri due vertici, e così via. È possibile calcolare il grado medio dei vertici utilizzando la funzione `mean()` di R:

```
> mean(mydegrees)
[1] 0.2183709
```

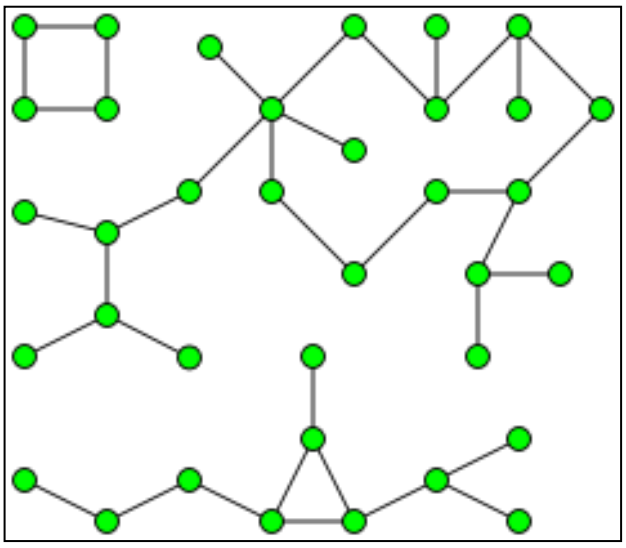
Il grado medio del grafo "litG" è solo di circa 0,22 poiché la maggior parte delle proteine non interagiscono con altre proteine.

È utile visualizzare la distribuzione dei gradi per un grafo tracciando un istogramma (mediante la funzione `hist()` di R):

```
> hist(mydegrees, col="red")
```



9.6 Trovare le componenti connesse nei grafi per i dati di interazione proteica



Se si sta analizzando un grafo molto grande, esso può contenere diversi sottografi dove i vertici all'interno di ogni sottografo sono collegati tra loro, ma non ci sono connessioni tra i vertici in sottografi diversi. In questo caso, i sottografi sono noti come *componenti connesse* (chiamati anche *sottografi massimamemte connessi*).

Ad esempio, il grafo a lato contiene tre componenti connesse.

È possibile trovare le componenti connesse di un grafo mediante la funzione `connectedComp()` del pacchetto *RBGL*

.

Ad esempio, per trovare le componenti connesse nel grafo "litG" digitiamo:

```
> library("RBGL")
> myconnectedcomponents <- connectedComp(litG)
```

I comandi sopra riportati memorizzano le componenti connesse nel grafo "litG" in una lista *myconnectedcomponents*. Ogni componente connessa è memorizzata in un elemento della lista; cioè, ogni elemento della lista *myconnectedcomponents* è un vettore che contiene i nomi delle proteine in una particolare componente connessa.

Possiamo visualizzare le proteine del lievito che sono i vertici delle prime tre componenti connesse esaminando i primi tre elementi della lista *myconnectedcomponents*. Si ricordi che è necessario utilizzare le doppie parentesi quadre per accedere agli elementi di una variabile lista in R:

```
> myconnectedcomponents[[1]]
[1] "YBL072C"
> myconnectedcomponents[[2]]
[1] "YBL083C"
> myconnectedcomponents[[3]]
 [1] "YBR009C" "YBR010W" "YNL030W" "YNL031C" "YOL139C" "YAR007C" "YBR073W"
 [8] "YER095W" "YJL173C" "YNL312W" "YBL084C" "YDR146C" "YLR127C" "YNL172W"
[15] "YLR134W" "YMR284W" "YER179W" "YIL144W" "YML104C" "YOR191W" "YDL008W"
[22] "YDL030W" "YDL042C" "YDR004W" "YGR162W" "YMR117C" "YDR386W" "YDR485C"
[29] "YDL043C" "YDR118W" "YMR106C" "YML032C" "YDR076W" "YDR180W" "YDL013W"
[36] "YDR227W"
```

Cioè, le prime due componenti connesse contengono una sola proteina ciascuna. Queste due proteine non hanno interazioni con nessuna delle altre proteine del lievito nel grafo "litG". La terza componente connessa contiene 36 proteine. Queste 36 proteine non sono necessariamente tutte collegate tra loro, ma ognuna delle 36 proteine deve essere collegata ad almeno una delle altre 35 della componente connessa. Si noti che le componenti connesse non sono memorizzate nell'elenco *myconnectedcomponents* in un ordine particolare; queste sono solo le prime tre componenti connesse memorizzate nell'elenco.

Per trovare il numero totale di componenti connesse nel grafo "litG", basta visualizzare la lunghezza della lista *myconnectedcomponents*:

```
> length(myconnectedcomponents)
[1] 2642
```

Nel grafo "litG" ci sono quindi 2.642 diverse componenti connesse. Si tratta di 2.642 sottografi del grafo, con collegamenti tra i vertici all'interno di un sottografo, ma non tra i 2.642 sottografi.

È interessante sapere qual è la più grande componente connessa in un grafo. Come possiamo calcolarla per il grafo "litG"? Ogni elemento del grafo contiene un vettore che memorizza le proteine in una particolare componente connessa, e la lunghezza di questo vettore è il numero di proteine in quella componente connessa. Quindi, per calcolare le dimensioni di tutte le componenti connesse nel grao "litG" possiamo usare un ciclo "`for`" per calcolare la lunghezza di ciascuno dei vettori in *myconnectedcomponents*:

```
> componentsizes <- numeric(2642) # Make a vector for storing the sizes of the␣
↪2642 connected components
> for (i in 1:2642) {
   component <- myconnectedcomponents[[i]] # Store the connected component in a␣
↪vector "component"
   componentsize <- length(component)      # Find the number of vertices in this␣
↪connected component
   componentsizes[i] <- componentsize      # Store the size of this component
}
```

Nel codice di cui sopra, la riga `componentsizes <- numeric(2642)` crea un nuovo vettore che ha lo stesso numero di elementi del numero di componenti connesse del grafo "litG". Questo vettore viene poi utilizzato all'interno del ciclo per memorizzare la dimensione di ogni componente connessa. Possiamo ora trovare la dimensione della componente conessa più grande del grafo "litG" utilizzando la funzione `max()`:

```
> max(componentsizes)
[1] 88
```

Ovvero, la massima componente connessa del grafo "litG" ha 88 diverse proteine.

Possiamo anche usare la funzione `table()` per costruire una tabella del numero di componenti connesse di diverse dimensioni:

```
> table(componentsizes)
componentsizes
```

1	2	3	4	5	6	7	8	12	13	36	88
2587	29	10	7	1	1	2	1	1	1	1	1

Questo ci dice che c'è solo una componente connessa con 88 proteine. Inoltre, vediamo che ci sono 2.587 componenti connesse che contengono solo 1 proteina ciascuna. Queste proteine presumibilmente non hanno alcuna interazione nota con qualsiasi altra proteina nel dataset "litG".

Per trovare la componente connessa cui appartiene una particolare proteina, è possibile utilizzare la funzione `findcomponent()` (vedi Appendice) che restituisce un vettore contenente i nomi delle proteine nella componente connessa. Ad esempio, per trovare la componente connessa contenente la proteina YBR009C, è possibile digitare:

```
> mycomponent <- findcomponent(litG, "YBR009C")
> mycomponent # Print out the members of this connected component.
   [1] "YBR009C" "YBR010W" "YNL030W" "YNL031C" "YOL139C" "YAR007C" "YBR073W"
↪"YER095W" "YJL173C" "YNL312W"
  [11] "YBL084C" "YDR146C" "YLR127C" "YNL172W" "YLR134W" "YMR284W" "YER179W"
↪"YIL144W" "YML104C" "YOR191W"
  [21] "YDL008W" "YDL030W" "YDL042C" "YDR004W" "YGR162W" "YMR117C" "YDR386W"
↪"YDR485C" "YDL043C" "YDR118W"
  [31] "YMR106C" "YML032C" "YDR076W" "YDR180W" "YDL013W" "YDR227W"
```

9.7 Estrarre un sottografo da un grafo in R

Se si vuole estrarre un particolare sottografo da un grafo si può usare la funzione `subGraph()` del pacchetto *graph*. Come argomenti di input, la funzione accetta un vettore che contiene i vertici (nodi) del sottografo da estrarre e il grafo a cui il sottografo appartiene.

Per esempio, se vogliamo estrarre il sottografo (del grafo "litG") che contiene la terza componente connessa nel vettore *myconnectedcomponents*, digitiamo:

```
> myconnectedcomponents <- connectedComp(litG)
> component3 <- myconnectedcomponents[[3]]
> component3 # Print out the proteins in connected component 3
 [1] "YBR009C" "YBR010W" "YNL030W" "YNL031C" "YOL139C" "YAR007C" "YBR073W" "YER095W"
 [9] "YJL173C" "YNL312W" "YBL084C" "YDR146C" "YLR127C" "YNL172W" "YLR134W" "YMR284W"
[17] "YER179W" "YIL144W" "YML104C" "YOR191W" "YDL008W" "YDL030W" "YDL042C" "YDR004W"
[25] "YGR162W" "YMR117C" "YDR386W" "YDR485C" "YDL043C" "YDR118W" "YMR106C" "YML032C"
[33] "YDR076W" "YDR180W" "YDL013W" "YDR227W"
> mysubgraph <- subGraph(component3, litG)
> mysubgraph
A graphNEL graph with undirected edges
Number of Nodes = 36
Number of Edges = 48
```

I comandi sopra riportati memorizzano il sottografo corrispondente a *component3* nella variabile di tipo grafo *mysubgraph* che contiene 36 vertici e 48 connessioni.

9.8 Rappresentare in R i grafi per i dati di interazione proteica

Il pacchetto *Rgraphviz* contiene funzioni utili per tracciare grafi o parti di grafi (sottografi).

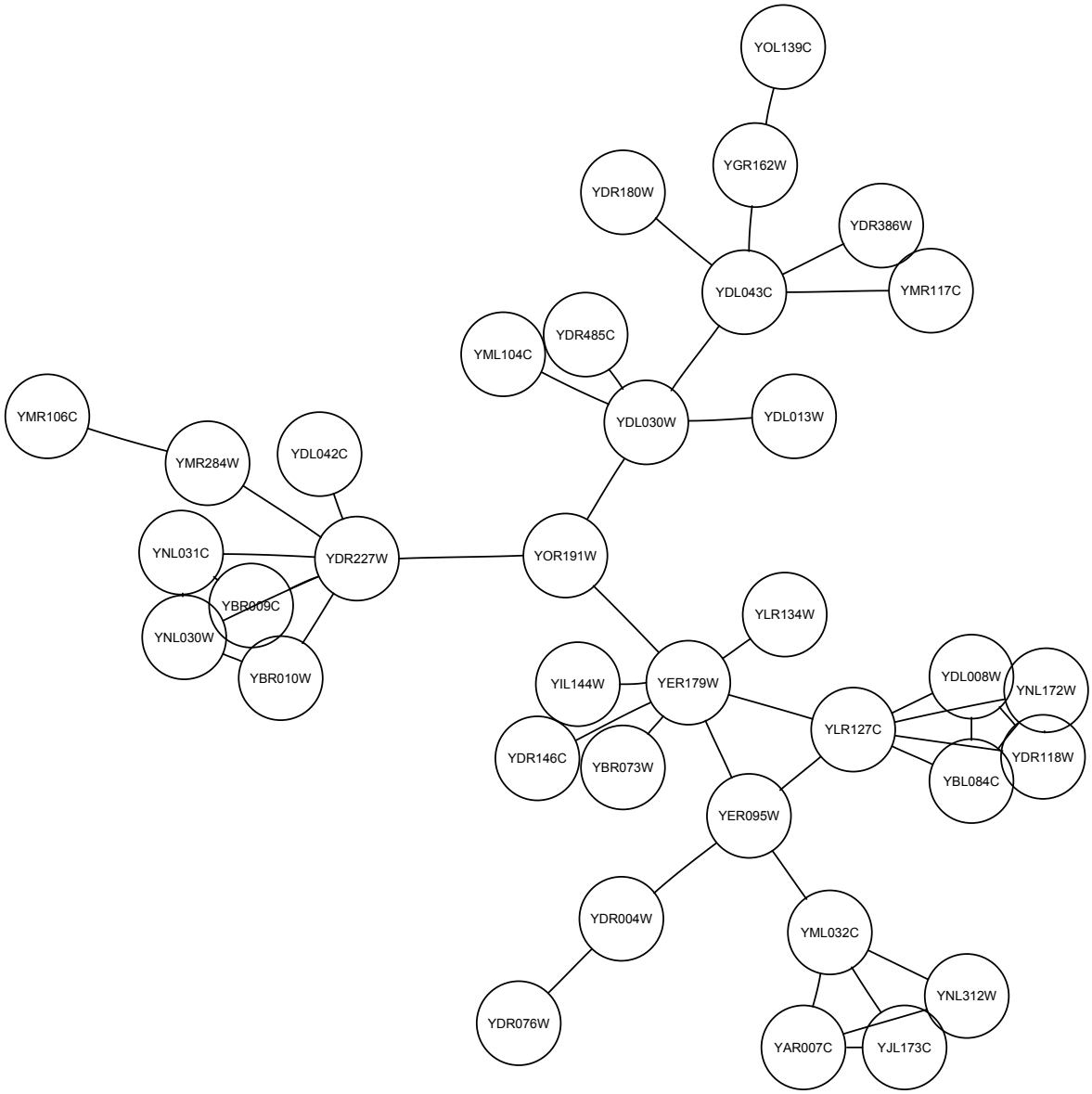
Le funzioni `layoutGraph()` e `renderGraph()` del pacchetto *Rgraphviz* possono essere usate per realizzare un efficace diagramma di un (sotto)grafo. Vi sono molte opzioni per i colori da usare per i vertici e gli archi.

Ad esempio, se vogliamo visualizzare il sottografo corrispondente alla terza componente connessa nel vettore *myconnectedcomponents*, possiamo digitare:

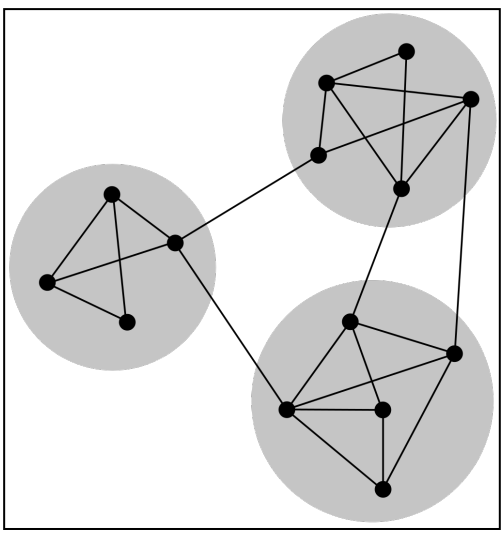
```
> BiocManager::install("Rgraphviz")
> library(Rgraphviz)
> mysubgraph <- subGraph(component3, litG)
> mygraphplot <- layoutGraph(mysubgraph, layoutType="neato")
> renderGraph(mygraphplot)
```

Il grafico della pagina seguente mostra un diagramma della terza componente connessa nel grafo "litG".

Vi sono 36 vertici, corrispondenti a 36 diverse proteine del lievito. I nomi delle proteine sono mostrati nei cerchi che rappresentano i vertici. Le connessioni tra i vertici rappresentano le interazioni tra coppie di proteine.



9.9 Rilevare le comunità in un grafo di interazione proteica utilizzando R



Una proprietà comune a molti tipi di grafi, compresi i grafi di interazione proteica, è la *struttura a comunità*. Una *comunità* è spesso definita come un sottoinsieme dei vertici del grafo in modo che le connessioni tra i vertici siano più dense rispetto alle connessioni con il resto del grafo. Per esempio, il grafo a lato è costituito da una componente connessa. Tuttavia, all'interno di tale componente, possiamo vedere tre sottografi densamente collegati; si potrebbe dire che si tratta di tre diverse comunità all'interno del grafo.

In termini di reti di interazione tra proteine, se vi sono diverse comunità all'interno di una componente connessa (ad esempio, tre

comunità, come nella figura sopra), queste potrebbero rappresentare tre diversi gruppi di proteine, dove le proteine all'interno di una comunità interagiscono molto di più tra loro che con le proteine nelle altre comunità.

Rilevando le comunità all'interno di un grafo di interazione tra proteine, possiamo rilevare i complessi proteici putativi, cioè gruppi di proteine associate che sono probabilmente abbastanza stabili nel tempo. In altre parole, i complessi proteici possono essere rilevati cercando gruppi di proteine tra cui ci sono molte interazioni, e dove i membri del complesso hanno poche interazioni con altre proteine che non appartengono al complesso.

Vi sono molti metodi diversi disponibili per rilevare le comunità in un grafo, e ogni metodo darà risultati leggermente diversi. Cioè, il particolare metodo usato per rilevare le comunità deciderà come dividere una componente connessa in una o più comunità.

La funzione `findcommunities()` (vedi Appendice) identifica le comunità all'interno di un grafo (o sottografo di un grafo). Richiede una seconda funzione, `findcommunities2()`, anch'essa in Appendice.

`findcommunities()` utilizza la funzione `spinglass.community()` del pacchetto *igraph* per identificare le comunità in un grafo o in un sottografo. Come argomenti, la funzione `findcommunities()` prende il grafo/sottografo in cui si vogliono trovare le comunità e il numero minimo di vertici che una comunità deve avere per essere rilevata.

Per esempio, per trovare comunità all'interno del sottografo corrispondente alla terza componente connessa del grafo "litG", possiamo digitare:

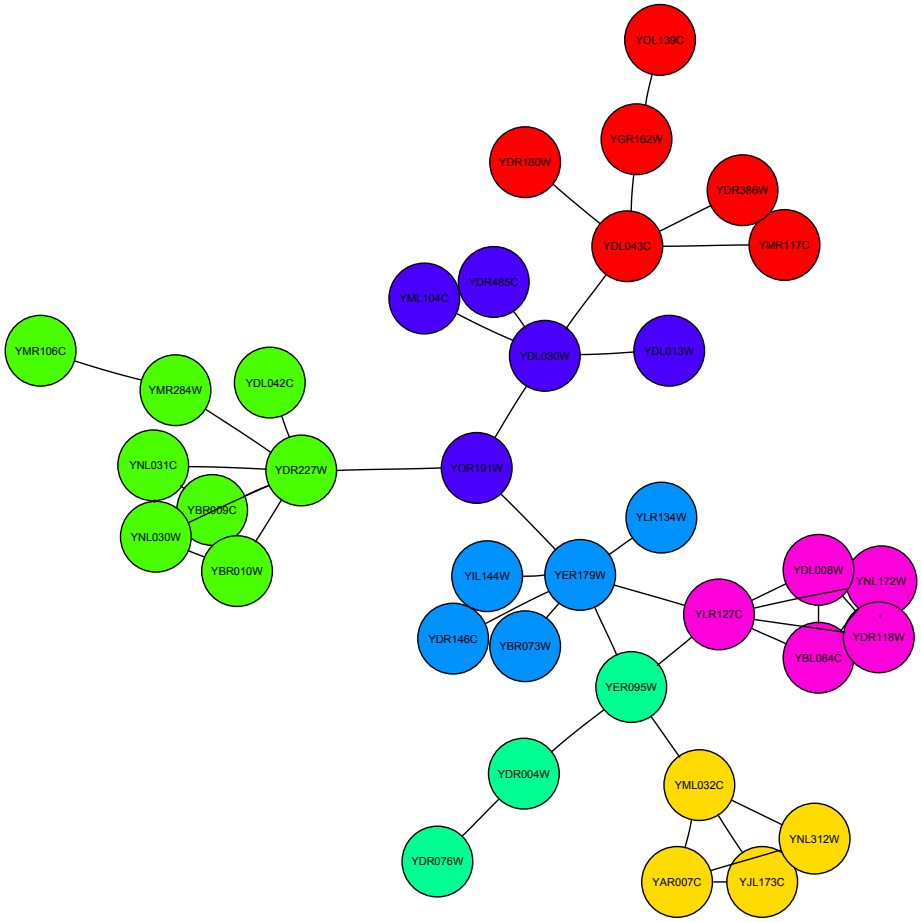
```
> mysubgraph <- subGraph(component3, litG)
> findcommunities(mysubgraph, 1)
[1] "Community 1 : YAR007C YER095W YJL173C YNL312W YDR004W YML032C YDR076W"
[1] "Community 2 : YOL139C YGR162W YMR117C YDR386W YDL043C YDR180W"
[1] "Community 3 : YMR284W YMR106C"
[1] "Community 4 : YBR073W YDR146C YLR134W YER179W YIL144W"
[1] "Community 5 : YBL084C YLR127C YNL172W YDL008W YDR118W"
[1] "Community 6 : YML104C YOR191W YDL030W YDR485C YDL013W"
[1] "Community 7 : YBR009C YBR010W YNL030W YNL031C YDL042C YDR227W"
[1] "There were 7 communities in the input graph"
```

Si vede dall'output precedente che ci sono sette diverse comunità nel sottografo corrispondente alla terza componente connessa del grafo "litG".

Si noti che se si esegue `findcommunities()` più volte sullo stesso grafo di input, potrebbero risultare ogni volta insiemi di comunità leggermente diversi. Questo perché la funzione utilizza un generatore di numeri casuali nel metodo che utilizza per identificare le comunità, e il numero casuale utilizzato sarà diverso ogni volta che si esegue la funzione, il che significa che si otterranno risposte leggermente diverse ogni volta. Le risposte dovrebbero essere molto simili; tuttavia si potrebbe notare una piccola differenza: ad esempio, una comunità grande potrebbe essere divisa in due comunità più piccole.

```
> mysubgraph <- subGraph(component3, litG)
> plotcommunities(mysubgraph)
```

È possibile tracciare una mappa delle comunità in un grafo o in un sottografo utilizzando la funzione `plotcommunities()` (vedi Appendice):



Nel grafo di cui sopra, le sette comunità nella terza componente connessa del grafo "litG" sono colorate con sette diversi colori.

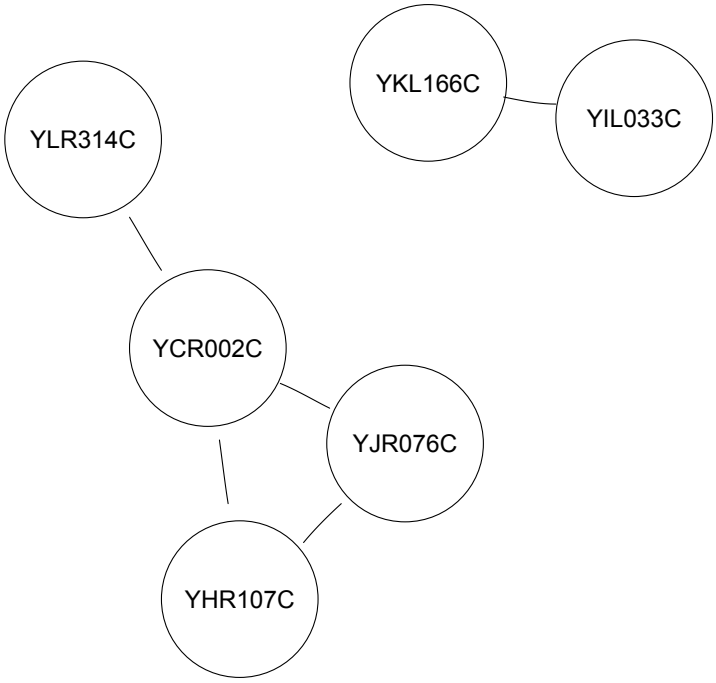
9.10 Lettura dei dati di interazione proteica in R

Nell'esempio sopra riportato, abbiamo esaminato il dataset "litG" delle interazioni tra proteine, un insieme di dati fornito con il pacchetto R *yeastExpData*. Ma cosa succede se si vuole esaminare un dataset di interazioni proteiche che non proviene da R?

È comune memorizzare i dati sulle interazioni proteiche in un file di testo con due colonne, dove ogni riga del file contiene una coppia di proteine che interagiscono tra loro. Ad esempio, un file di questo tipo può avere questo aspetto: YKL166C YIL033C YCR002C YHR107C YCR002C YJR076C YCR002C YLR314C YJR076C YHR107C. Questo indica che ci sono 5 interazioni proteina-proteina, tra le proteine YKL166C e YIL033C, YCR002C e YHR107C, YCR002C e YJR076C, YCR002C e YLR314C e tra YJR076C e YHR107C.

La funzione `makeproteingraph()` (vedi Appendice) realizza un grafo basato su un file di input delle interazioni tra proteine, dove le prime due colonne del file di input indicano le coppie di proteine che interagiscono. Ad esempio, supponendo che il file "ExampleInteractionData.txt" nella directory di lavoro contenga le cinque coppie di proteine interagenti sopra elencate, è possibile leggerle e creare un grafo per queste proteine digitando:

```
> thegraph <- makeproteingraph("ExampleInteractionData.txt")
> mygraphplot <- layoutGraph(thegraph, layoutType="neato")
> renderGraph(mygraphplot)
```



9.11 Creazione di grafi *random* in R

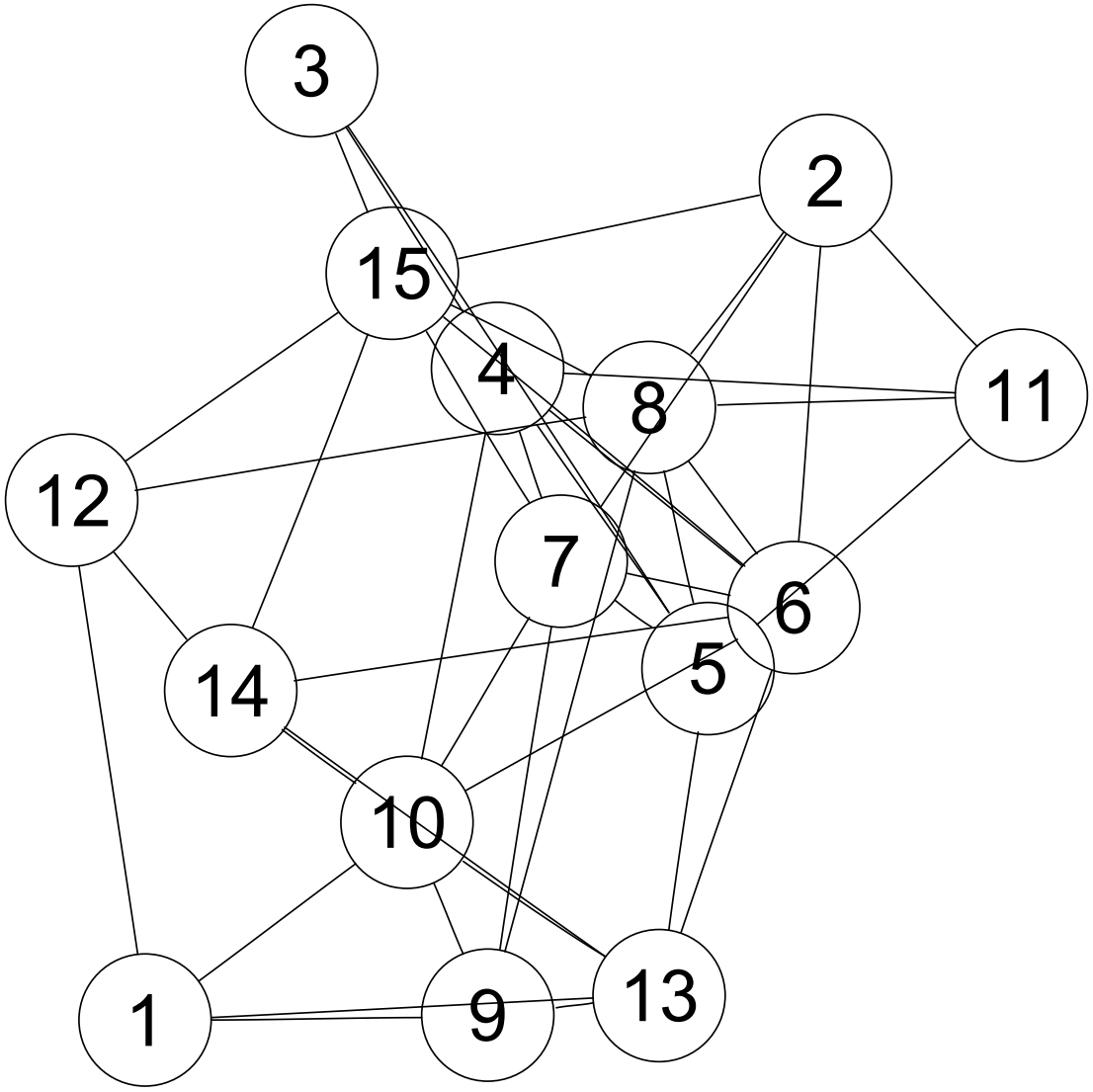
Un *grafo random* è un grafo generato da un processo casuale, in cui si inizia con un certo numero di vertici (nodi) e gli archi vengono aggiunti scegliendo casualmente coppie di vertici e creando un collegamento tra i due membri di ciascuna di queste coppie (metodo conosciuto come *modello Erdös-Renyi per i grafi random*). In un grafo *random*, i vertici con molte connessioni sono ugualmente probabili come i vertici con pochissime connessioni. Cioè, se si calcola il grado medio dei vertici in un grafo casuale, si scopre che i gradi della maggior parte dei vertici nel grafo sono vicini alla media.

Spesso è utile e interessante confrontare le proprietà dei *network* biologici con quelle dei grafi *random* ("undirected", cioè non orientati). Per fare questo, è necessario essere in grado di generare alcuni grafi *random*. La funzione `makerandomgraph()` (vedi Appendice) crea un grafo *random*, e ha come argomento il numero di nodi e di archi che si vuole che il grafo debba avere. Per esempio, per creare un grafo *random* con 15 nodi e 43 connessioni, digitiamo:

```
> myrandomgraph <- makerandomgraph(15, 43)
> myrandomgraph # Print out the number of vertices and edges in the graph
A graphNEL graph with undirected edges
Number of Nodes = 15
Number of Edges = 43
```

Nel codice precedente vengono assegnate ai vertici le etichette da 1 a 15. Possiamo tracciare il grafo *random* mediante i comandi seguenti:

```
> myrandomgraphplot <- layoutGraph(myrandomgraph, layoutType="neato")
> renderGraph(myrandomgraphplot)
```



10. Estrazione delle caratteristiche strutturali delle proteine

Una sequenza di aminoacidi può fornire alcune informazioni sulla proteina. Le sue proprietà includono la composizione degli aminoacidi, l'idrofobicità e così via. Una volta recuperata una sequenza, possiamo saperne di più su di essa, come la frequenza nucleotidica e il contenuto di GC (vedi Capitoli 1 e 2) presente in essa. In questo capitolo esamineremo come ottenere informazioni sulla proteina dai dati della sequenza.

10.1 Estrazione di caratteristiche proteiche tramite il pacchetto *protR*

Il pacchetto *protR* consente l'estrazione delle caratteristiche di una sequenza proteica, che possono essere utilizzate per vari scopi. Implementa la maggior parte delle funzioni di utilità per calcolare i descrittori della sequenza proteica. La *composizione aminoacidica* (AAC) è la frazione di ogni tipo di aminoacido all'interno di una proteina, calcolata mediante la funzione `extractAAC()`. Allo stesso modo, la maggior parte delle caratteristiche che illustreremo in questa sezione sono calcolate in base al contenuto della sequenza. I descrittori cercano i singoli aminoacidi, e il descrittore complessivo è calcolato in base a questi.

Per ottenere le caratteristiche dai dati di sequenza, utilizziamo quindi il pacchetto *protR* disponibile su CRAN, che può essere installato, caricato (e attivato insieme ai pacchetti da cui dipende) come qualsiasi altro pacchetto R come segue:

```
> install.packages("protr")
> library(bio3d)
> library(seqinr)
> library(protr)
```

Per analizzare una sequenza, la recuperiamo da PDB (vedi par. 5.2) digitando i seguenti comandi:

```
> pdb1 <- read.pdb("1BG2")
> s1 <- aa321(as.character(pdb1$seqres))
```

Il pacchetto *protR* ha bisogno della sequenza come stringa di caratteri. Pertanto, occorre prima convertire il vettore di caratteri in una stringa come segue:

```
> s1 <- c2s(s1)
> s1
[1]
"MADLAECNIKVMCRFRPLNESEVNRGDKYIAKFQGEDTVVIASKPYAFDRVFQSSTSQEQVYNDCAKKIVKDVLEGYNGTIFAY
GQTSSGKTHTMEGKLHDPEGMGIIPRIVQDIFNYIYSMDENLEFHIKVSYFEIYLDKIRDLLDVSKTNLSVHEDKNRVPYVKGCT
ERFVCSPDEVMDTIDEGKSNRHVAVTNMNEHSSRSHSIFLINVKQENTQTEQKLSGKLYLVDLAGSEKVSKTGAEGAVLDEAKNI
NKSLSALGNVISALAEGSTYVPYRDSKMTRILQDSLGGNCRTTIVICCSPSSYNESETKSTLLFGQRAKTI"
```

Ora possiamo estrarre e calcolare le caratteristiche da questa sequenza. Iniziamo con la composizione degli aminoacidi usando la funzione `extractAAC()`:

```
> extractAAC(s1)
         A          R          N          D          C          E          Q          G          H
0.04923077 0.04307692 0.05846154 0.06153846 0.02461538 0.07384615 0.03384615 0.06153846 0.02153846
         I          L          K          M          F          P          S          T          W
0.06769231 0.06769231 0.07692308 0.02461538 0.03384615 0.02461538 0.09538462 0.06153846 0.00000000
         Y          V
0.04307692 0.07692308
```

Controlliamo se i tipi di aminoacidi della sequenza proteica sono nei 20 tipi predefiniti utilizzando la seguente funzione:

```
> protcheck(s1)
[1] TRUE
```

Calcoliamo la caratteristica successiva, cioè la composizione anfifilica[3] pseudo aminoacidica, che dà un'idea relativa all'ordine della sequenza di una proteina e alla distribuzione degli aminoacidi idrofobici e idrofili lungo la sua catena, con l'aiuto del seguente comando:

```
> extractAPAAC(s1, props = c("Hydrophobicity", "Hydrophilicity"), lambda = 30, w =
0.05, customprops = NULL)
                Pc1.A                 Pc1.R                 Pc1.N                 Pc1.D
         1.559274e+01          1.364365e+01          1.851638e+01          1.949093e+01
                Pc1.C                 Pc1.E                 Pc1.Q                 Pc1.G
         7.796371e+00          2.338911e+01          1.072001e+01          1.949093e+01
                Pc1.H                 Pc1.I                 Pc1.L                 Pc1.K
         6.821824e+00          2.144002e+01          2.144002e+01          2.436366e+01
                Pc1.M                 Pc1.F                 Pc1.P                 Pc1.S
         7.796371e+00          1.072001e+01          7.796371e+00          3.021094e+01
                Pc1.T                 Pc1.W                 Pc1.Y                 Pc1.V
         1.949093e+01          0.000000e+00          1.364365e+01          2.436366e+01
 Pc2.Hydrophobicity.1  Pc2.Hydrophilicity.1  Pc2.Hydrophobicity.2  Pc2.Hydrophilicity.2
        -2.191252e-03          1.807624e-03         -3.394759e-03         -4.434121e-03
 Pc2.Hydrophobicity.3  Pc2.Hydrophilicity.3  Pc2.Hydrophobicity.4  Pc2.Hydrophilicity.4
         2.390839e-03          2.071590e-03          1.402134e-04          4.170478e-03
 Pc2.Hydrophobicity.5  Pc2.Hydrophilicity.5  Pc2.Hydrophobicity.6  Pc2.Hydrophilicity.6
        -5.645002e-03         -2.576934e-03         -3.816560e-03         -2.961727e-03
 Pc2.Hydrophobicity.7  Pc2.Hydrophilicity.7  Pc2.Hydrophobicity.8  Pc2.Hydrophilicity.8
         1.969807e-03          2.997077e-03         -7.562103e-04          1.802072e-03
 Pc2.Hydrophobicity.9  Pc2.Hydrophilicity.9 Pc2.Hydrophobicity.10 Pc2.Hydrophilicity.10
        -1.472387e-04          1.203361e-03          1.238568e-04          2.514355e-03
Pc2.Hydrophobicity.11 Pc2.Hydrophilicity.11 Pc2.Hydrophobicity.12 Pc2.Hydrophilicity.12
         8.217123e-04          2.047829e-03         -2.758131e-03          4.449484e-05
Pc2.Hydrophobicity.13 Pc2.Hydrophilicity.13 Pc2.Hydrophobicity.14 Pc2.Hydrophilicity.14
        -3.793538e-04          9.105654e-04          6.384032e-04          3.388078e-03
Pc2.Hydrophobicity.15 Pc2.Hydrophilicity.15 Pc2.Hydrophobicity.16 Pc2.Hydrophilicity.16
        -8.639649e-04         -2.602823e-03         -3.454223e-06          2.735599e-03
Pc2.Hydrophobicity.17 Pc2.Hydrophilicity.17 Pc2.Hydrophobicity.18 Pc2.Hydrophilicity.18
        -1.030651e-03          3.611545e-03          6.392462e-03          5.413131e-03
Pc2.Hydrophobicity.19 Pc2.Hydrophilicity.19 Pc2.Hydrophobicity.20 Pc2.Hydrophilicity.20
        -8.004659e-04          1.371016e-03         -4.732215e-03         -1.104242e-03
Pc2.Hydrophobicity.21 Pc2.Hydrophilicity.21 Pc2.Hydrophobicity.22 Pc2.Hydrophilicity.22
        -5.754892e-04         -2.596259e-04         -1.783870e-03          3.242798e-03
Pc2.Hydrophobicity.23 Pc2.Hydrophilicity.23 Pc2.Hydrophobicity.24 Pc2.Hydrophilicity.24
        -2.631277e-03          1.774054e-03         -3.114488e-03         -3.884162e-03
Pc2.Hydrophobicity.25 Pc2.Hydrophilicity.25 Pc2.Hydrophobicity.26 Pc2.Hydrophilicity.26
         1.073171e-04         -9.120840e-04          4.154261e-03          4.638141e-03
Pc2.Hydrophobicity.27 Pc2.Hydrophilicity.27 Pc2.Hydrophobicity.28 Pc2.Hydrophilicity.28
        -9.529463e-04          3.295043e-03          3.107037e-03          1.640724e-03
Pc2.Hydrophobicity.29 Pc2.Hydrophilicity.29 Pc2.Hydrophobicity.30 Pc2.Hydrophilicity.30
        -9.585703e-04          4.136713e-04          3.394123e-03          6.391990e-03
```

Calcoliamo alcune altre caratteristiche come il descrittore di composizione, il descrittore di transizione e la composizione del dipeptide con l'aiuto dei seguenti comandi:

[3] Una molecola è detta **anfifilica** quando contiene sia un gruppo idrofilo sia uno idrofobo. Queste caratteristiche molecolari fanno sì che molecole anfifiliche, immerse in un liquido acquoso, tendono a formare spontaneamente un doppio strato, nel quale le teste idrofile sono rivolte verso l'esterno e le code idrofobe verso l'interno.

```
> extractCTDC(s1)
 hydrophobicity.Group1  hydrophobicity.Group2  hydrophobicity.Group3 normwaalsvolume.Group1
             0.3476923              0.3569231              0.2953846              0.3784615
normwaalsvolume.Group2 normwaalsvolume.Group3        polarity.Group1        polarity.Group2
             0.3784615              0.2430769              0.3384615              0.2923077
       polarity.Group3  polarizability.Group1  polarizability.Group2  polarizability.Group3
             0.3692308              0.3292308              0.4276923              0.2430769
         charge.Group1          charge.Group2          charge.Group3 secondarystruct.Group1
             0.1200000              0.7446154              0.1353846              0.3907692
secondarystruct.Group2 secondarystruct.Group3   solventaccess.Group1   solventaccess.Group2
             0.3076923              0.3015385              0.3815385              0.3476923
  solventaccess.Group3
             0.2707692
```

```
> extractCTDD(s1)
  prop1.G1.residue0  prop1.G1.residue25  prop1.G1.residue50  prop1.G1.residue75
          0.9230769          20.9230769          48.3076923          68.3076923
prop1.G1.residue100   prop1.G2.residue0  prop1.G2.residue25  prop1.G2.residue50
         99.3846154           0.6153846          27.0769231          56.0000000
 prop1.G2.residue75 prop1.G2.residue100   prop1.G3.residue0  prop1.G3.residue25
         80.0000000          99.6923077           0.3076923          22.7692308
 prop1.G3.residue50  prop1.G3.residue75 prop1.G3.residue100   prop2.G1.residue0
         45.5384615          71.3846154         100.0000000           0.6153846
 prop2.G1.residue25  prop2.G1.residue50  prop2.G1.residue75 prop2.G1.residue100
         25.5384615          54.1538462          80.0000000          99.6923077
  prop2.G2.residue0  prop2.G2.residue25  prop2.G2.residue50  prop2.G2.residue75
          1.2307692          24.0000000          49.2307692          72.6153846
prop2.G2.residue100   prop2.G3.residue0  prop2.G3.residue25  prop2.G3.residue50
        100.0000000           0.3076923          20.9230769          41.5384615
 prop2.G3.residue75 prop2.G3.residue100   prop3.G1.residue0  prop3.G1.residue25
         64.0000000          99.3846154           0.3076923          22.7692308
 prop3.G1.residue50  prop3.G1.residue75 prop3.G1.residue100   prop3.G2.residue0
         44.6153846          70.7692308         100.0000000           0.6153846
 prop3.G2.residue25  prop3.G2.residue50  prop3.G2.residue75 prop3.G2.residue100
         26.7692308          59.3846154          81.8461538          99.6923077
  prop3.G3.residue0  prop3.G3.residue25  prop3.G3.residue50  prop3.G3.residue75
          0.9230769          22.1538462          48.3076923          68.0000000
prop3.G3.residue100   prop4.G1.residue0  prop4.G1.residue25  prop4.G1.residue50
         99.3846154           0.6153846          26.1538462          56.0000000
 prop4.G1.residue75 prop4.G1.residue100   prop4.G2.residue0  prop4.G2.residue25
         79.6923077          99.6923077           1.2307692          23.0769231
 prop4.G2.residue50  prop4.G2.residue75 prop4.G2.residue100   prop4.G3.residue0
         49.8461538          73.2307692         100.0000000           0.3076923
 prop4.G3.residue25  prop4.G3.residue50  prop4.G3.residue75 prop4.G3.residue100
         20.9230769          41.5384615          64.0000000          99.3846154
  prop5.G1.residue0  prop5.G1.residue25  prop5.G1.residue50  prop5.G1.residue75
          3.0769231          20.6153846          48.9230769          72.9230769
prop5.G1.residue100   prop5.G2.residue0  prop5.G2.residue25  prop5.G2.residue50
         99.3846154           0.3076923          25.5384615          51.3846154
 prop5.G2.residue75 prop5.G2.residue100   prop5.G3.residue0  prop5.G3.residue25
         76.3076923         100.0000000           0.9230769          22.1538462
 prop5.G3.residue50  prop5.G3.residue75 prop5.G3.residue100   prop6.G1.residue0
         44.3076923          66.1538462          95.6923077           0.3076923
 prop6.G1.residue25  prop6.G1.residue50  prop6.G1.residue75 prop6.G1.residue100
         22.7692308          49.5384615          73.8461538          99.3846154
  prop6.G2.residue0  prop6.G2.residue25  prop6.G2.residue50  prop6.G2.residue75
          2.1538462          23.6923077          45.5384615          73.2307692
prop6.G2.residue100   prop6.G3.residue0  prop6.G3.residue25  prop6.G3.residue50
        100.0000000           0.9230769          27.0769231          53.8461538
…
```

```
> extractDC(s1)
         AA          RA          NA          DA          CA          EA          QA          GA
0.000000000 0.003086420 0.000000000 0.000000000 0.003086420 0.003086420 0.000000000 0.006172840
         HA          IA          LA          KA          MA          FA          PA          SA
0.000000000 0.006172840 0.009259259 0.000000000 0.003086420 0.003086420 0.000000000 0.006172840
         TA          WA          YA          VA          AR          RR          NR          DR
0.000000000 0.000000000 0.003086420 0.003086420 0.000000000 0.000000000 0.009259259 0.003086420
         CR          ER          QR          GR          HR          IR          LR          KR
0.006172840 0.003086420 0.003086420 0.000000000 0.000000000 0.003086420 0.000000000 0.000000000
         MR          FR          PR          SR          TR          WR          YR          VR
0.000000000 0.003086420 0.003086420 0.003086420 0.003086420 0.000000000 0.003086420 0.000000000
         AN          RN          NN          DN          CN          EN          QN          GN
0.000000000 0.000000000 0.000000000 0.000000000 0.003086420 0.006172840 0.000000000 0.006172840
         HN          IN          LN          KN          MN          FN          PN          SN
0.000000000 0.006172840 0.003086420 0.006172840 0.003086420 0.003086420 0.000000000 0.003086420
         TN          WN          YN          VN          AD          RD          ND          DD
0.006172840 0.000000000 0.009259259 0.003086420 0.003086420 0.006172840 0.003086420 0.000000000
         CD          ED          QD          GD          HD          ID          LD          KD
0.000000000 0.006172840 0.006172840 0.003086420 0.003086420 0.003086420 0.009259259 0.003086420
         MD          FD          PD          SD          TD          WD          YD          VD
0.006172840 0.003086420 0.003086420 0.000000000 0.000000000 0.000000000 0.000000000 0.003086420
         AC          RC          NC          DC          CC          EC          QC          GC
0.000000000 0.000000000 0.003086420 0.003086420 0.003086420 0.003086420 0.000000000 0.003086420
         HC          IC          LC          KC          MC          FC          PC          SC
0.000000000 0.003086420 0.000000000 0.000000000 0.003086420 0.000000000 0.000000000 0.000000000
         TC          WC          YC          VC          AE          RE          NE          DE
0.000000000 0.000000000 0.000000000 0.003086420 0.009259259 0.000000000 0.009259259 0.012345679
         CE          EE          QE          GE          HE          IE          LE          KE
0.000000000 0.000000000 0.006172840 0.003086420 0.003086420 0.000000000 0.006172840 0.000000000
         ME          FE          PE          SE          TE          WE          YE          VE
0.003086420 0.003086420 0.003086420 0.009259259 0.006172840 0.000000000 0.000000000 0.000000000
         AQ          RQ          NQ          DQ          CQ          EQ          QQ          GQ
0.000000000 0.000000000 0.000000000 0.000000000 0.000000000 0.006172840 0.000000000 0.006172840
         HQ          IQ          LQ          KQ          MQ          FQ          PQ          SQ
0.000000000 0.000000000 0.003086420 0.003086420 0.000000000 0.006172840 0.000000000 0.003086420
         TQ          WQ          YQ          VQ          AG          RG          NG          DG
0.003086420 0.000000000 0.000000000 0.003086420 0.003086420 0.003086420 0.003086420 0.000000000
         CG          EG          QG          GG          HG          IG          LG          KG
0.000000000 0.018518519 0.003086420 0.003086420 0.000000000 0.000000000 0.006172840 0.003086420
         MG          FG          PG          SG          TG          WG          YG          VG
0.003086420 0.003086420 0.000000000 0.006172840 0.003086420 0.000000000 0.003086420 0.000000000
         AH          RH          NH          DH          CH          EH          QH          GH
0.000000000 0.003086420 0.000000000 0.000000000 0.000000000 0.003086420 0.000000000 0.000000000
         HH          IH          LH          KH          MH          FH          PH          SH
0.000000000 0.000000000 0.003086420 0.000000000 0.000000000 0.003086420 0.000000000 0.003086420
         TH          WH          YH          VH          AI          RI          NI          DI
0.003086420 0.000000000 0.000000000 0.003086420 0.000000000 0.006172840 0.006172840 0.003086420
         CI          EI          QI          GI          HI          II          LI          KI
0.000000000 0.003086420 0.000000000 0.003086420 0.003086420 0.003086420 0.003086420 0.006172840
         MI          FI          PI          SI          TI          WI          YI          VI
0.000000000 0.000000000 0.000000000 0.003086420 0.012345679 0.000000000 0.006172840 0.009259259
         AL          RL          NL          DL          CL          EL          QL          GL
0.006172840 0.000000000 0.006172840 0.009259259 0.000000000 0.000000000 0.000000000 0.000000000
         HL          IL          LL          KL          ML          FL          PL          SL
0.000000000 0.003086420 0.006172840 0.009259259 0.000000000 0.003086420 0.003086420 0.006172840
         TL          WL          YL          VL          AK          RK          NK          DK
0.003086420 0.000000000 0.006172840 0.006172840 0.012345679 0.000000000 0.003086420 0.009259259
         CK          EK          QK          GK          HK          IK          LK          KK
0.000000000 0.003086420 0.003086420 0.012345679 0.000000000 0.006172840 0.000000000 0.003086420
         MK          FK          PK          SK          TK          WK          YK          VK
0.000000000 0.000000000 0.000000000 0.012345679 0.003086420 0.000000000 0.000000000 0.009259259
         AM          RM          NM          DM          CM          EM          QM          GM
0.000000000 0.000000000 0.003086420 0.000000000 0.000000000 0.000000000 0.000000000 0.003086420
         HM          IM          LM          KM          MM          FM          PM          SM
0.000000000 0.000000000 0.000000000 0.003086420 0.000000000 0.000000000 0.000000000 0.003086420
…
```

10.2 Gestione dei file PDB

PDB è un archivio mondiale dei dati strutturali delle macromolecole biologiche e dispone di una notevole quantità di dati proteici che comprende sia la struttura che la sequenza. Il formato dei dati PDB è lo standard per i file contenenti coordinate atomiche insieme a sequenze e altre informazioni. La gestione di un file PDB è il primo passo per analizzare una struttura proteica.

Questa sezione illustra in dettaglio la funzione `read.pdb()` del pacchetto *bio3d* che legge i dati PDB in base all'ID in input come oggetto PDB. L'oggetto creato ha tutti i record relativi alle coordinate dei residui, alle sequenze e agli angoli di torsione. Per estrarre le sequenze o altri elementi dalla proteina, utilizziamo semplicemente i nomi degli oggetti presenti all'interno dell'oggetto PDB.

Per prima cosa, carichiamo le librerie *protr* e *bio3d*:

```
> library(protr)
> library(bio3d)
```

Leggiamo quindi un file PDB per il dominio del motore proteico della kinesina umana con l'ID 1BG2 come segue:

```
> pdb <- read.pdb("1BG2")
  Note: Accessing on-line PDB file
```

Controlliamo le diverse parti e componenti dell'oggetto PDB creato:

```
> class(pdb)
[1] "pdb" "sse"
> attributes(pdb)
$names
[1] "atom"   "xyz"    "seqres" "helix"  "sheet"  "calpha" "remark" "call"
$class
[1] "pdb" "sse"
> head(pdb)
$atom
       type eleno elety  alt resid chain resno insert      x       y      z o      b segid elesy
1      ATOM     1     N <NA>   ASP     A     3   <NA> 43.743  -2.106 39.408 1 100.00  <NA>     N
2      ATOM     2    CA <NA>   ASP     A     3   <NA> 45.053  -2.661 39.856 1 100.00  <NA>     C
3      ATOM     3     C <NA>   ASP     A     3   <NA> 45.305  -2.401 41.340 1 100.00  <NA>     C
4      ATOM     4     O <NA>   ASP     A     3   <NA> 46.119  -3.083 41.957 1 100.00  <NA>     O
5      ATOM     5    CB <NA>   ASP     A     3   <NA> 46.204  -2.067 39.034 1 100.00  <NA>     C
…
> head(pdb$atom[, c("x","y","z")])
       x      y      z
1 43.743 -2.106 39.408
2 45.053 -2.661 39.856
3 45.305 -2.401 41.340
4 46.119 -3.083 41.957
5 46.204 -2.067 39.034
```

Per ottenere le coordinate C-α nella molecola della proteina, è sufficiente accedere al record corrispondente dell'oggetto PDB digitando il seguente comando:

```
> head(pdb$atom[pdb$calpha, c("resid", "elety", "x","y","z")])
   resid elety      x      y      z
2    ASP    CA 45.053 -2.661 39.856
10   LEU    CA 44.791 -1.079 43.319
18   ALA    CA 42.451 -3.691 44.790
23   GLU    CA 40.334 -4.572 41.737
32   CYS    CA 36.711 -3.532 42.171
38   ASN    CA 36.940 -0.636 44.620
```

Per consultare la sequenza nell'oggetto PDB, accediamo al record della sequenza come segue:

```
> c2s(aa321(pdb$seqres))
[1]
"MADLAECNIKVMCRFRPLNESEVNRGDKYIAKFQGEDTVVIASKPYAFDRVFQSSTSQEQVYNDCAKKIVKDVLEGYNGTIFAY
GQTSSGKTHTMEGKLHDPEGMGIIPRIVQDIFNYIYSMDENLEFHIKVSYFEIYLDKIRDLLDVSKTNLSVHEDKNRVPYVKGCT
ERFVCSPDEVMDTIDEGKSNRHVAVTNMNEHSSRSHSIFLINVKQENTQTEQKLSGKLYLVDLAGSEKVSKTGAEGAVLDEAKNI
NKSLSALGNVISALAEGSTYVPYRDSKMTRILQDSLGGNCRTTIVICCSPSSYNESETKSTLLFGQRAKTI"
```

Utilizziamo la funzione `write.pdb()` per registrare localmente un file PDB, che può essere letto successivamente con la funzione `read.pdb()` nella sessione R come segue:

```
> write.pdb(pdb, file="myPDBfile.pdb")
> read.pdb("myPDBfile.pdb")

 Call:  read.pdb(file = "myPDBfile.pdb")

   Total Models#: 1
     Total Atoms#: 2733,  XYZs#: 8199  Chains#: 1  (values: A)

     Protein Atoms#: 2527  (residues/Calpha atoms#: 323)
     Nucleic acid Atoms#: 0  (residues/phosphate atoms#: 0)

     Non-protein/nucleic Atoms#: 206  (residues: 174)
     Non-protein/nucleic resid values: [ ACT (3), ADP (1), HOH (169), MG (1) ]

   Protein sequence:
      DLAECNIKVMCRFRPLNESEVNRGDKYIAKFQGEDTVVIASKPYAFDRVFQSSTSQEQVY
      NDCAKKIVKDVLEGYNGTIFAYGQTSSGKTHTMEGKLHDPEGMGIIPRIVQDIFNYIYSM
      DENLEFHIKVSYFEIYLDKIRDLLDVSKTNLSVHEDKNRVPYVKGCTERFVCSPDEVMDT
      IDEGKSNRHVAVTNMNEHSSRSHSIFLINVKQENTQTEQKLSGKL...<cut>...AKTI

+ attr: atom, xyz, calpha, call
```

10.3 Lavorare con le annotazioni del dominio *InterPro*

InterPro è una risorsa che fornisce l'analisi funzionale delle sequenze proteiche classificandole in famiglie e prevedendo la presenza di domini e siti importanti. Per classificare le proteine in questo modo, *InterPro* utilizza modelli predittivi chiamati "firme" provenienti da diversi database che compongono il consorzio *InterPro*. Esso comprende Gene3D, PANTHER, PRINTS, ProDom, PROSITE, TIGRFAM e così via. *InterPro* è un database di famiglie di proteine, domini e siti funzionali, in cui le caratteristiche identificabili trovate nelle proteine conosciute possono essere applicate a nuove sequenze proteiche per caratterizzarle funzionalmente.

Vediamo un esempio di ritrovamento di annotazioni di proteine da *InterPro*. Innanzitutto, attiviamo la libreria UniProt.ws per accedere ai dati delle sequenze proteiche UniProt e cerchiamo l'ID tassonomico della specie *Homo sapiens*:

```
> library(UniProt.ws)
> availableUniprotSpecies(pattern="sapiens")
Downloading: 690 kB
  taxon ID                                                                  Species name
1   742918 Human associated cyclovirus 1 (isolate Homo sapiens/Pakistan/PK5510/2007)
2    63221                                                 Homo sapiens neanderthalensis
3     9606                                                                 Homo sapiens
```

Attiviamo quindi un oggetto UniProt.ws sul dataset *Homo sapiens* e impostiamo l'ID Ensembl della *NAD chinasi*, la cui annotazione intendiamo recuperare:

```
> up <- UniProt.ws(taxId=9606)
> ensemblID <- "ENSG00000008130"
```

Per recuperare l'annotazione InterPro (visualizzando le colonne INTERPRO e GO), utilizziamo la funzione `select()` di UniProt.ws come segue:

```
> res <- select(up, keytype = "ENSEMBL", keys = c(ensemblID),
      columns = c("INTERPRO","GO"))
Getting mapping data for ENSG00000008130 ... and ACC
Getting extra data for A0A0A0MR98, J3KSP9, J3KTI3... (6 total)
'select()' returned 1:many mapping between keys and columns
```

L'oggetto recuperato è di tipo data.frame, con ogni riga che rappresenta un'annotazione per l'ID specificato. L'output è il seguente:

```
> res
          ENSEMBL                                        INTERPRO
1 ENSG00000008130  IPR017438; IPR017437; IPR016064; IPR002504;
2 ENSG00000008130                                           <NA>
3 ENSG00000008130             IPR017438; IPR016064; IPR002504;
4 ENSG00000008130             IPR017437; IPR016064; IPR002504;
5 ENSG00000008130  IPR017438; IPR017437; IPR016064; IPR002504;

GO
1 NAD+ kinase activity [GO:0003951]; NAD metabolic process [GO:0019674]; NADP
biosynthetic process [GO:0006741]
2 <NA>
3 NAD+ kinase activity [GO:0003951]; NADP biosynthetic process [GO:0006741]
4 NAD+ kinase activity [GO:0003951]; NAD metabolic process [GO:0019674]; NADP
biosynthetic process [GO:0006741]
5 cytosol [GO:0005829]; ATP binding [GO:0005524]; metal ion binding [GO:0046872];
NAD+ kinase activity [GO:0003951]; ATP metabolic process [GO:0046034]; NAD metabolic
process [GO:0019674]; NADP biosynthetic process [GO:0006741]; phosphorylation
[GO:0016310]; positive regulation of insulin secretion involved in cellular response
to glucose stimulus [GO:0035774]
```

10.4 Comprendere il *grafico di Ramachandran*

Il *grafico di Ramachandran* è un modo per visualizzare le combinazioni degli angoli diedri Ψ (psi) e Φ (phi) dei residui aminoacidici ammesse all'interno di una struttura polipeptidica. In linea di principio, mostra tutte le possibili conformazioni degli angoli Ψ e Φ per i residui di aminoacidi in una proteina dovuti a vincoli sterici. Serve anche per la validazione della struttura della proteina. Nel grafico sono riportati in ordinata i valori di Ψ ed in ascissa i valori di Φ, che rappresentano rispettivamente l'angolo Cα-C' e l'angolo N-Cα di un legame peptidico Cα-C'-N-Cα fra due aminoacidi adiacenti.

In questa sezione mostreremo come tracciare tale grafico per una proteina leggendo un file PDB ed estraendo gli angoli Ψ e Φ. Il file PDB, come si è visto in precedenza, ha vari elementi: gli angoli di torsione Ψ e Φ sono estratti dalla funzione `torsion.pdb()`.

Per visualizzare il grafico di Ramachandran per la proteina 1BG2 procediamo come di seguito illustrato. Per prima cosa, carichiamo la libreria *bio3d* nella sessione R come segue:

```
> library(bio3d)
```

Quindi leggiamo il file PDB della proteina 1BG2:

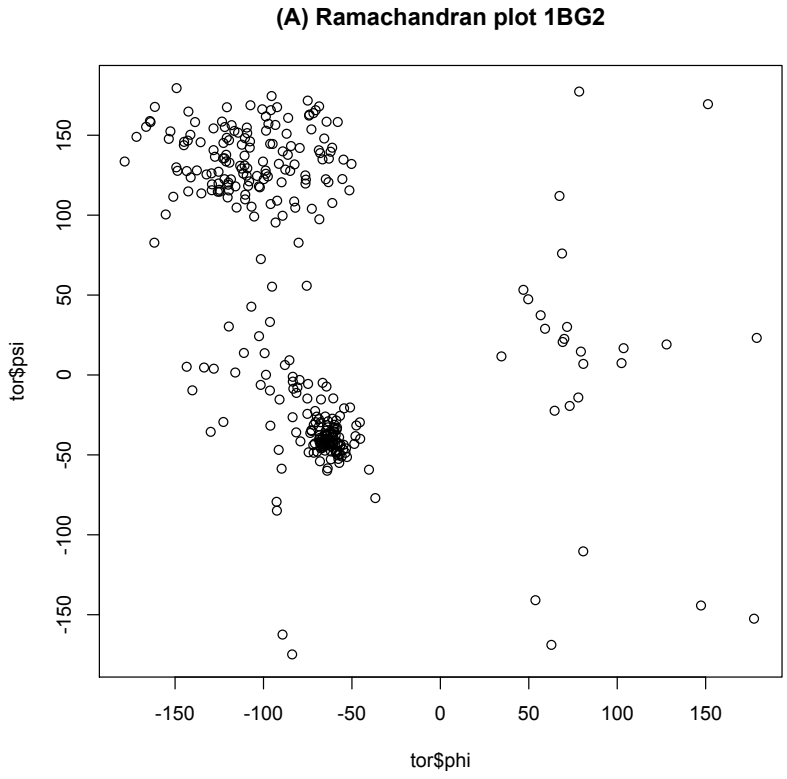
```
> pdb <- read.pdb("1BG2")
  Note: Accessing on-line PDB file
```

Estraiamo gli angoli di torsione Ψ e Φ con la seguente funzione:

```
> tor <- torsion.pdb(pdb)
```

Ora, è sufficiente tracciare i componenti degli angoli di torsione con la funzione `plot()` di R come segue:

```
plot(tor$phi, tor$psi, main="(A) Ramachandran plot 1BG2")
```



È possibile migliorare la visualizzazione con un codice leggermente più sofisticato; prima occorre però estrarre gli angoli di torsione separatamente come segue:

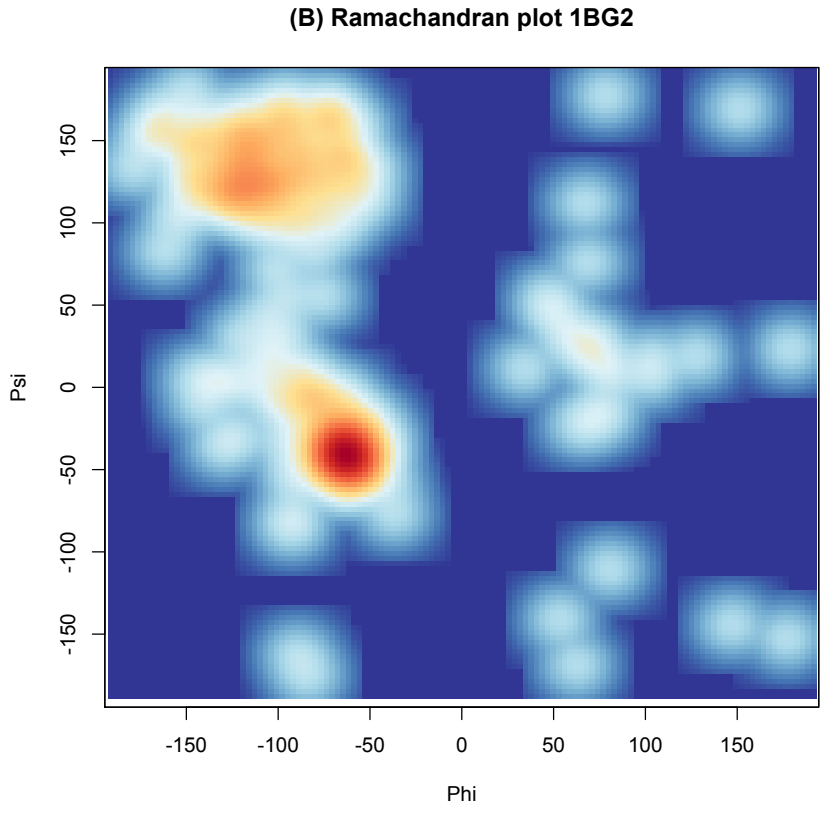
```
> scatter.psi <- tor$psi
> scatter.phi <- tor$phi
```

Poi, creiamo il colore dei contorni per il nuovo grafico di Ramachandran con l'aiuto dei seguenti comandi:

```
> library(RColorBrewer) # load RColourBrewer package
> k <- 10 # define number of colours
> my.cols <- rev(brewer.pal(k, "RdYlBu")) # Brew color palette
```

Con questi colori e gli angoli di torsione precedentemente estratti, creiamo un grafico di Ramachandran più gradevole usando il seguente comando:

```
> smoothScatter(x=scatter.phi, y=scatter.psi, colramp=colorRampPalette(my.cols),
      xlim=c(-180,180), ylim=c(-180,180), xlab="Phi", ylab="Psi",
      main="(B) Ramachandran plot 1BG2", pch=19, cex=0.00)
```



Si noti che sia il primo grafico in bianco e nero (tracciato con la funzione `plot()`) che quello successivo a colori (tracciato con la funzione `smoothScatter()`) mostrano gli angoli Ψ lungo l'asse *y* e gli angoli Φ lungo l'asse *x*, con le aree corrispondenti che possono essere facilmente confrontate. I punti nel grafico in bianco e nero si riferiscono alla coppia di angoli di torsione osservati nella struttura proteica. Nel grafico colorato, le tonalità blu rappresentano lo spazio non occupato dagli angoli di torsione, mentre le tonalità bianche e rosse rappresentano le parti dove la struttura mostra i corrispondenti angoli di torsione. Entrambi i grafici complessivamente delineano le conformazioni preferite della proteina.

10.5 Ricerca di proteine simili (BLAST)

Una volta che abbiamo una proteina candidata, può essere interessante cercare le eventuali proteine simili. È possibile ottenere questo risultato mediante il tool BLAST (Basic Local Alignment Search Tool). Se lo si esegue online (ad es. sul sito web NCBI https://www.ncbi.nlm.nih.gov/BLAST/), si cercano sequenze simili in un database di sequenze di destinazione e si utilizzano le corrispondenze per cercare proteine simili rilevanti in base ai loro *similarity scores* (punteggi di somiglianza). R ci permette di fare qualcosa di simile. In questa sezione illustreremo il metodo per fare una ricerca BLAST per un file PDB direttamente da una sessione R.

Per prima cosa, attiviamo la libreria *bio3d* e leggiamo il file PDB della proteina 1BG2 come segue:

```
> library(bio3d)
> pdb <- read.pdb("1BG2")
```

Quindi estraiamo il record di sequenza dall'oggetto PDB con la rappresentazione a singola lettera degli aminoacidi utilizzando la funzione `aa321()`:

```
> mySeq <- aa321(pdb$seqres)
```

Eseguiamo BLAST utilizzando la funzione `blast.pdb()` con la sequenza estratta in precedenza (potrebbe richiedere un po' di tempo):

```
> myBlast <- blast.pdb(mySeq)
 Searching ... please wait (updates every 5 seconds) RID = 1VBNZAE2016
 ..
 Reporting 101 hits
> summary(myBlast)
        Length Class      Mode
hit.tbl 16     data.frame list
raw     13     data.frame list
url      1     -none-     character
```

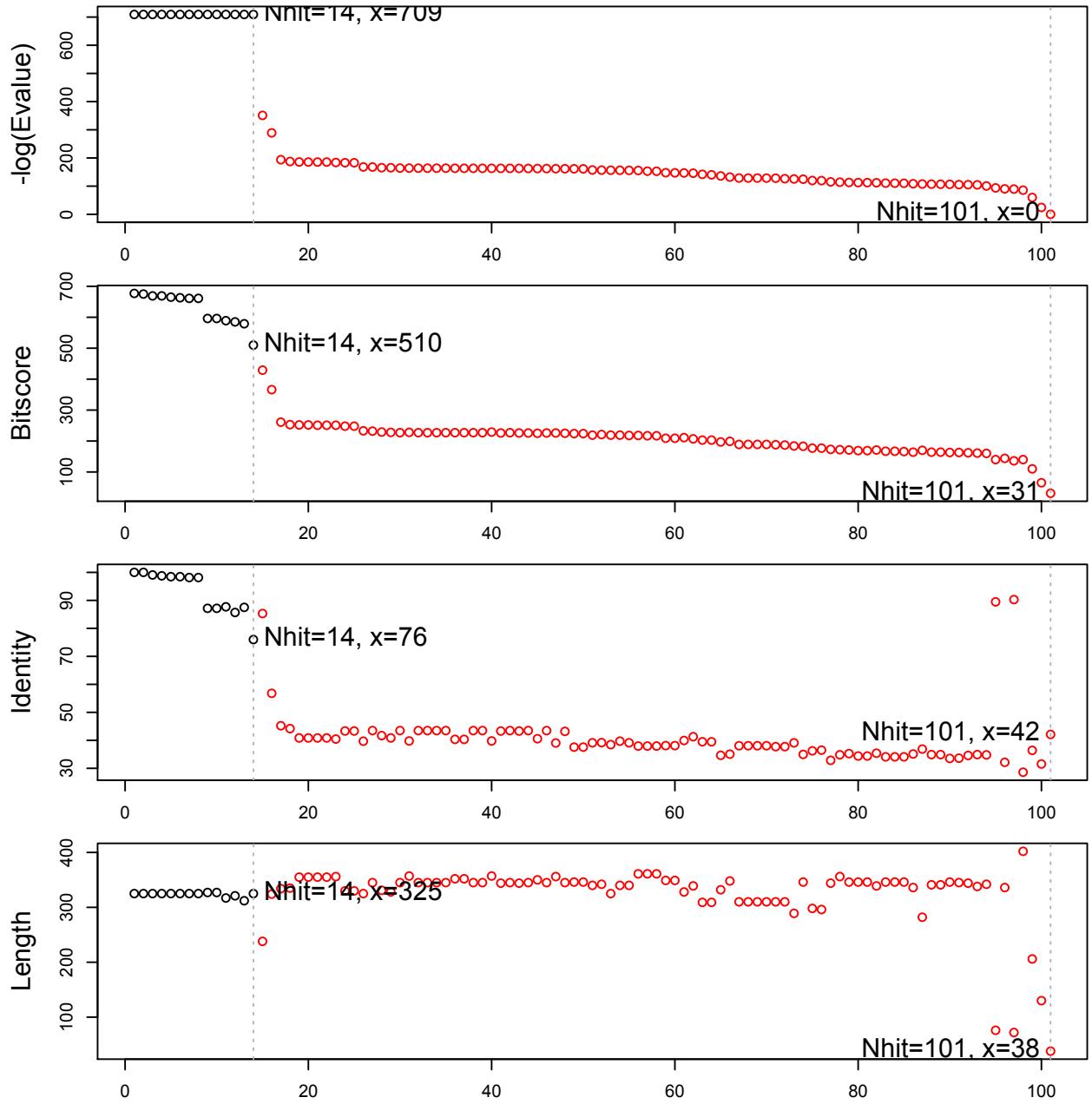
Possiamo vedere la parte inziale del risultato della ricerca digitando il seguente comando:

```
> head(myBlast$hit.tbl)
      queryid subjectids identity alignmentlength mismatches gapopens q.start q.end s.start s.end
1 Query_24211     1MKJ_A  100.000             325          0        0       1   325       1   325
2 Query_24211     1BG2_A  100.000             325          0        0       1   325       1   325
3 Query_24211     3J8X_K   99.077             325          3        0       1   325       1   325
4 Query_24211     4ATX_C   98.769             325          4        0       1   325       1   325
5 Query_24211     4HNA_K   98.462             325          5        0       1   325       1   325
 evalue bitscore positives mlog.evalue pdb.id    acc
1      0      677    100.00    709.1962 1MKJ_A 1MKJ_A
2      0      675    100.00    709.1962 1BG2_A 1BG2_A
3      0      669     99.08    709.1962 3J8X_K 3J8X_K
4      0      669     99.38    709.1962 4ATX_C 4ATX_C
5      0      665     98.46    709.1962 4HNA_K 4HNA_K
```

Tracciamo le sequenze più simili come segue:

```
> top.hits <- plot(myBlast)
  * Possible cutoff values:    709 0
            Yielding Nhits:    14 101

  * Chosen cutoff value of:    709
            Yielding Nhits:    14
```



Esaminiamo la lista delle sequenze più simili:

```
> head(top.hits$hits)
  pdb.id   acc      group
1 "1MKJ_A" "1MKJ_A" "1"
2 "1BG2_A" "1BG2_A" "1"
3 "3J8X_K" "3J8X_K" "1"
4 "4ATX_C" "4ATX_C" "1"
5 "4HNA_K" "4HNA_K" "1"
```

Abbiamo effettuato in R una ricerca molto simile all'esecuzione di BLAST su web. L'elaborazione inizia con l'estrazione della sequenza dal file PDB desiderato, che viene poi inviata al sito remoto NCBI per una ricerca BLASTP, cioè una ricerca rispetto alle sequenze di proteine nel repository dei dati. I risultati della ricerca consistono in una tabella che contiene le informazioni sugli ID PDB delle proteine trovate e altri ID insieme al loro raggruppamento. Per ulteriori informazioni, controllare l'help della funzione `blast.pdb()`.

10.6 Analisi delle caratteristiche strutturali secondarie delle proteine

Per procedere abbiamo bisogno del programma DSSP o STRIDE, utilizzati per determinare la struttura secondaria di una proteina prendendo come input le coordinate tridimensionali dei suoi atomi. Il programma viene invocato tramite la libreria *bio3d*, che rende più facile l'analisi della struttura proteica.

Per prima cosa, occorre installare in locale il pacchetto *dssp* (http://swift.cmbi.ru.nl/gv/dssp/) o *stride* (http://webclu.bio.wzw.tum.de/stride/). Carichiamo quindi la libreria *bio3d* e leggiamo i dati della proteina 12AS:

```
> library(bio3d)
> pdb <- read.pdb("12as")
  Note: Accessing on-line PDB file
> pdb
 Call:  read.pdb(file = "12as")
   Total Models#: 1
     Total Atoms#: 5385,  XYZs#: 16155  Chains#: 2  (values: A B)
     Protein Atoms#: 5136  (residues/Calpha atoms#: 656)
     Nucleic acid Atoms#: 0  (residues/phosphate atoms#: 0)
     Non-protein/nucleic Atoms#: 249  (residues: 205)
     Non-protein/nucleic resid values: [ AMP (2), HOH (203) ]
   Protein sequence:
      AYIAKQRQISFVKSHFSRQLEERLGLIEVQAPILSRVGDGTQDNLSGAEKAVQVKVKALP
      DAQFEVVHSLAKWKRQTLGQHDFSAGEGLYTHMKALRPDEDRLSPLHSVYVDQWDWERVM
      GDGERQFSTLKSTVEAIWAGIKATEAAVSEEFGLAPFLPDQIHFVHSQELLSRYPDLDAK
      GRERAIAKDLGAVFLVGIGGKLSDGHRHDVRAPDYDDWSTPSELG...<cut>...LLNN
+ attr: atom, xyz, seqres, helix, sheet, calpha, remark, call
```

Estraiamo e visualizziamo le informazioni sulla struttura secondaria della proteina con la funzione `dssp()`:

```
> pdb <- dssp(pdb, exepath="/usr/local/bin/")
/usr/local/bin/dssp
Warning message:
In dssp.pdb(pdb, exepath = "/usr/local/bin/") :
  Non-protein residues detected in input PDB: AMP, HOH
> attributes(pdb)
$names
[1] "helix"  "sheet"  "hbonds" "turn"   "phi"    "psi"    "acc"    "sse"   "call"
$class
[1] “sse"
> pdb$helix
$start
  1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17  18  19  20  21
  5  76 130 170 182 258 277 297 320   5  76 130 170 182 258 277 297 320 273 310 271
$end
  1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17  18  19  20  21
 27  83 154 176 193 268 283 305 325  26  83 155 176 193 268 283 304 325 275 312 275
$length
 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22
23  8 25  7 12 11  7  9  6 22  8 26  7 12 11  7  8  6  3  3  5  3
$chain
  1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17  18  19  20  21
"A" "A" "A" "A" "A" "A" "A" "A" "A" "B" "B" "B" "B" "B" "B" "B" "B" "B" "A" "A" "B"
$type
  1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17  18  19  20  21
"H" "H" "H" "H" "H" "H" "H" "H" "H" "H" "H" "H" "H" "H" "H" "H" "H" "H" "G" "G" "G"
```

Questo ci dice che la prima elica va dal residuo 5 al residuo 27 (entrambi inclusi).

Eseguiamo ora analoghe operazioni di estrazione delle caratteristiche della struttura secondaria mediante la funzione `stride()`[4]:

```
> pdb <- read.pdb("12as")
  Note: Accessing on-line PDB file
> pdb
 Call:  read.pdb(file = "12as")
   Total Models#: 1
     Total Atoms#: 5385,  XYZs#: 16155  Chains#: 2  (values: A B)
     Protein Atoms#: 5136  (residues/Calpha atoms#: 656)
     Nucleic acid Atoms#: 0  (residues/phosphate atoms#: 0)
     Non-protein/nucleic Atoms#: 249  (residues: 205)
     Non-protein/nucleic resid values: [ AMP (2), HOH (203) ]
   Protein sequence:
      AYIAKQRQISFVKSHFSRQLEERLGLIEVQAPILSRVGDGTQDNLSGAEKAVQVKVKALP
      DAQFEVVHSLAKWKRQTLGQHDFSAGEGLYTHMKALRPDEDRLSPLHSVYVDQWDWERVM
      GDGERQFSTLKSTVEAIWAGIKATEAAVSEEFGLAPFLPDQIHFVHSQELLSRYPDLDAK
      GRERAIAKDLGAVFLVGIGGKLSDGHRHDVRAPDYDDWSTPSELG...<cut>...LLNN
+ attr: atom, xyz, seqres, helix, sheet, calpha, remark, call
> stride(pdb)
Call:
  stride(pdb = pdb)
Class:
  sse
Helices:  22
         5-27 (A)      73-84 (A)    130-155 (A)    170-176 (A)    182-193 (A)
      258-266 (A)    277-284 (A)    297-305 (A)    320-325 (A)      4-27 (B)
        75-84 (B)    130-155 (B)    170-176 (B)    182-193 (B)    258-268 (B)
      277-283 (B)    297-305 (B)    320-325 (B)    273-275 (A)    310-312 (A)
      271-275 (B)    310-312 (B)
Sheets:   24
        29-31 (A)      37-39 (A)      56-57 (A)      67-69 (A)      90-99 (A)
      113-122 (A)    166-169 (A)    195-199 (A)    204-205 (A)    209-210 (A)
      233-240 (A)    245-254 (A)    290-296 (A)      29-30 (B)      37-39 (B)
        56-57 (B)      67-69 (B)      90-99 (B)    113-122 (B)    166-169 (B)
      195-199 (B)    233-240 (B)    245-254 (B)    290-296 (B)
Turns:   38
        40-42 (A)      47-52 (A)      59-65 (A)      70-72 (A)      87-89 (A)
      100-106 (A)    108-109 (A)    111-112 (A)    124-129 (A)    162-165 (A)
      177-180 (A)    194-194 (A)    200-203 (A)    206-208 (A)    211-213 (A)
      215-218 (A)    220-223 (A)    225-228 (A)    241-244 (A)    267-272 (A)
        40-42 (B)      47-52 (B)      59-65 (B)      70-73 (B)      87-89 (B)
      100-103 (B)    107-112 (B)    124-129 (B)    162-165 (B)    177-180 (B)
      194-194 (B)    200-203 (B)    205-208 (B)    211-213 (B)    216-218 (B)
      220-223 (B)    225-228 (B)    241-244 (B)
Output is provided in residue numbers
```

L'oggetto restituito è una tabella di residui con la loro struttura secondaria sotto forma di elica α (Helices), foglietti β (Sheets) e torsioni (Turns). In particolare, La sezione "`Helices:`" dei risultati conferma che la prima elica va dal residuo 5 al residuo 27 incluso.

10.7 Visualizzazione delle strutture proteiche

La visualizzazione della struttura tridimensionale di una proteina è fondamentale per diversi scopi, come la progettazione di farmaci e la modellazione delle proteine. Esistono molti strumenti per la visualizzazione

[4] E' anche disponibile una versione online di STRIDE all'indirizzo http://webclu.bio.wzw.tum.de/cgi-bin/stride/stridecgi.py

delle strutture e, soprattutto, per la visualizzazione 3D interattiva. Esempi comuni sono RasMol (http://www.openrasmol.org), Jmol (http://jmol.sourceforge.net), Cn3D (https://www.ncbi.nlm.nih.gov/Structure/CN3D/cn3d.shtml) e VMD (https://www.ks.uiuc.edu/Research/vmd/). Il software RasMol accetta i formati PDB e MMDB, mentre Cn3D legge il formato ASN.1, più raffinato. Oltre a questo, il Cn3D può anche leggere la struttura (sotto forma di file ASN.1) direttamente dalle banche dati remote e fornisce un rendering più veloce. VMD è un programma per la visualizzazione, l'animazione e l'analisi di grandi sistemi biomolecolari che utilizza grafica 3D e scripting integrato. Al momento R non fornisce strumenti sofisticati come quelli indicati; tuttavia, per completezza, in questa sezione introduciamo un metodo semplice per esaminare la struttura di una proteina in R.

Abbiamo bisogno del pacchetto *Rknots* (progettato per l'analisi topologica dei polimeri) per la visualizzazione dei dati proteici di nostro interesse, che a sua volta necessita dei pacchetti *rgl*, *rSymPy*, *rJython*, *rJava*, *rjson* e *bio3d*. Perciò, installiamo e carichiamo il pacchetto *Rknots* dal repository CRAN (insieme ai pacchetti da cui dipende) come segue:

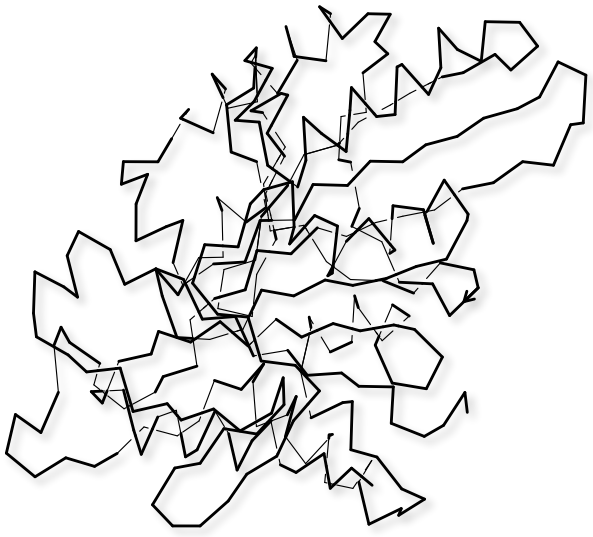
```
> install.packages("Rknots", dependencies = TRUE)
> library(Rknots)
Carico il pacchetto richiesto: rgl
Carico il pacchetto richiesto: rSymPy
Carico il pacchetto richiesto: rJython
Carico il pacchetto richiesto: rJava
Carico il pacchetto richiesto: rjson
Carico il pacchetto richiesto: bio3d
```

Carichiamo quindi la proteina da PDB:

```
> myprot <- loadProtein("1BG2")
  Note: Accessing on-line PDB file
Summary of the distance vector
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  3.755   3.793   3.804   3.803   3.813   3.841
No gap found
```

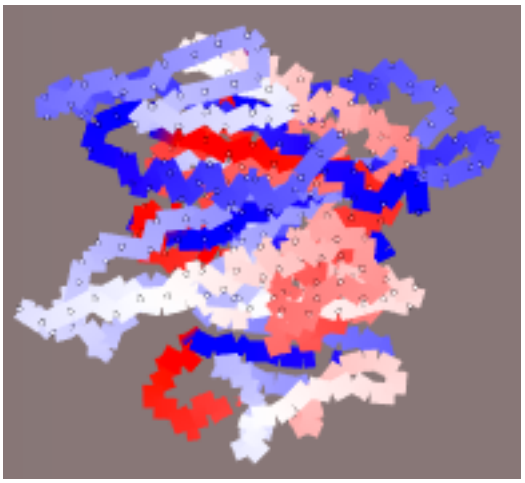
Possiamo osservare la struttura 2D della proteina con la funzione `plotDiagram()` (l'attributo myprot$A contiene le coordinate tridimensionali della proteina; il parametro `lwd` definisce lo spessore del tratto, mentre `ends=c()` serve a tracciare le linee unite e non separate):

```
> plotDiagram(myprot$A, ends = c(), lwd = 2.5)
```



Il pacchetto *rgl* consente una semplice visualizzazione 3D a colori con la funzione `plotKnot3D()`:

```
> ramp <- colorRamp(c('blue', ' white', ' red' ))
> pal <- rgb(ramp(seq(0,1,length=100)), max=255)
> plotKnot3D(myprot$A, colors=list(pal), lwd=8, radius=0.4, showNC=TRUE, text=FALSE)
```

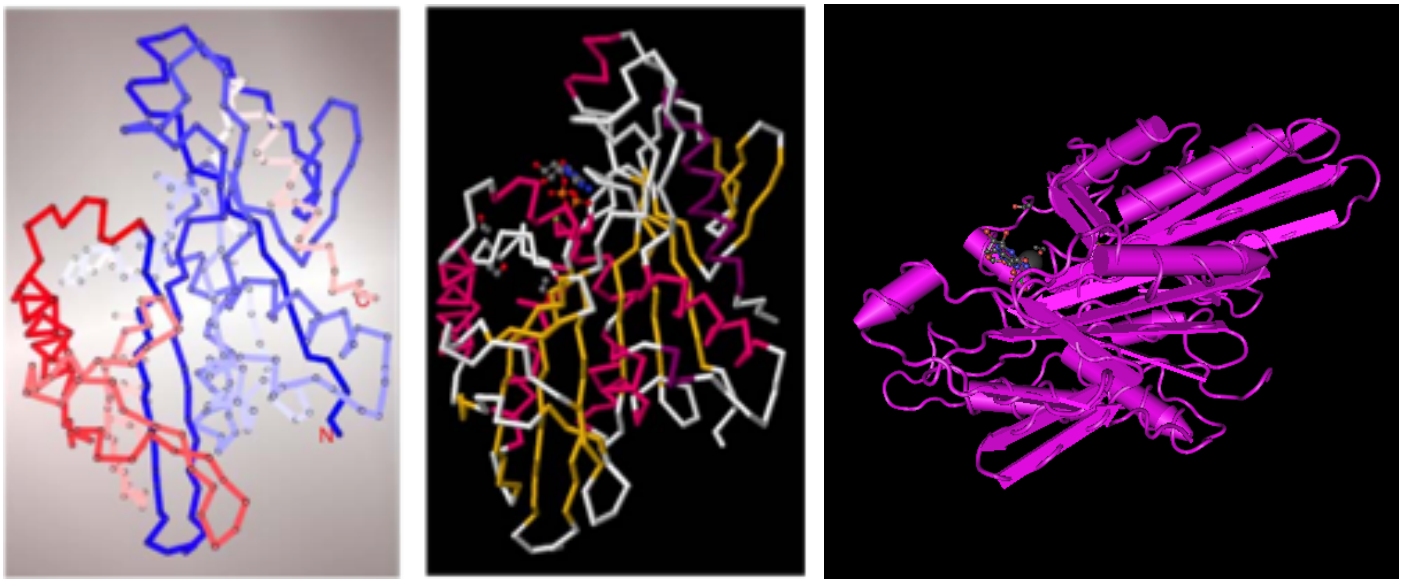


Abbiamo quindi visto che la funzione `loadProtein()` recupera le informazioni PDB per l'ID specificato, mentre `plotDiagram()` crea la struttura in 2D. Per visualizzarla, abbiamo utilizzato una tavolozza di colori con diverse tonalità dal rosso al blu (suddivisi in 100 intervalli uguali). Quindi abbiamo tracciato la struttura 3D mediante la grafica RGL della funzione `plotKnot3D()`.

Si consiglia di utilizzare gli altri strumenti già citati (*RasMol*, *Jmol*, *Cn3D* o *VMD*) per una migliore visualizzazione delle strutture proteiche. In ogni caso, è possibile scrivere i dati su un file PDB locale utilizzando la funzione `write.pdb()` (oppure scaricando direttamente il file da PDB) e poi visualizzarli con lo strumento preferito:

```
> pdb <- read.pdb( "1bg2" )
  Note: Accessing on-line PDB file
> write.pdb(pdb, file = "1bg2.pdb")
```

La seguente schermata mostra la visualizzazione della proteina 1BG2 che utilizza il pacchetto *Rknots* (a sinistra), il software *Jmol* (al centro) e *Cn3D* (a destra):



Il sito web RCSB PDB (Research Collaboratory for Structural Bioinformatics PDB, https://www.rcsb.org) è un ottimo punto di partenza per la sintesi delle proteine, la ricerca e la formazione in biologia molecolare, biologia strutturale, biologia computazionale e oltre.

11. Analisi di dati da microarray

I microarray sono uno degli strumenti più utili per comprendere i meccanismi biologici attraverso misurazioni su larga scala di campioni, tipicamente DNA, RNA o proteine. La tecnica è stata utilizzata per una serie di scopi nella ricerca nel campo delle scienze della vita, che vanno dal profilo dell'espressione genica all'identificazione di SNP (Single-Nucleotide Polymorphism) o altri biomarcatori e, inoltre, per capire le relazioni tra i geni e le loro attività su larga scala. Tuttavia, raggiungere tali inferenze dall'enorme quantità di dati disponibili su un singolo chip è impegnativo sia in termini di biologia che di statistica. Il dominio dell'analisi dei dati da microarray ha rivoluzionato la ricerca in biologia nell'ultimo decennio. In questo capitolo, esploreremo i diversi metodi che vengono utilizzati per analizzare i dati di espressione genica da microarray che hanno diversi obiettivi biologici. Ci possono essere molti tipi di microarray a seconda del campione biologico utilizzato. Allo stesso modo, ci sono diverse tecniche in uso per produrre questi array; ad es. Affymetrix, Illumina e così via. Ci concentreremo principalmente sui microarray per la misurazione dell'espressione genica con campioni di acido nucleico e utilizzeremo i dati dei file CEL Affymetrix per gli esempi esplicativi. Tuttavia, la maggior parte delle tecniche può essere utilizzata su altre piattaforme con solo leggere modifiche.

Il capitolo inizia con la lettura e il caricamento dei dati dei microarray, seguito dalla loro preelaborazione, analisi, estrazione e, infine, la loro visualizzazione utilizzando R. Successivamente, esamineremo le soluzioni relative all'uso biologico dei dati.

11.1 Lettura dei file CEL

Un file CEL contiene tutte le informazioni relative all'intensità dei pixel dell'array, come l'intensità stessa, la deviazione standard, il numero di pixel e altre meta-informazioni. Per averne un'idea, possiamo aprire il file CEL con un qualsiasi editor di testo e dargli un'occhiata. Ogni esperimento di solito ha più campioni e repliche; quindi, ci sarà un file CEL presente per ogni campione o replica. I file CEL devono essere letti in maniera opportuna per ottenere i dati in un formato utilizzabile in R.

In questa sezione spiegheremo come leggere un file CEL nell'area di lavoro R, utilizzando come esempio un semplice dataset dalla banca dati GEO di NCBI sul cancro al seno. Inoltre, avremo bisogno di alcuni pacchetti R che esamineremo in dettaglio nella sezione seguente.

Scarichiamo il dataset GSE24460 da NCBI-GEO (https://www.ncbi.nlm.nih.gov/geo/query/acc.cgi?acc=GSE24460). Otteniamo un file chiamato GSE24460_RAW.tar, da registrare nella directory di lavoro e scompattare per ottenere i file CEL veri e propri nella sottocartella "GSE24460_RAW". Successivamente, installiamo e carichiamo la libreria *affy* di Bioconductor nella sessione R come segue:

```
> BiocManager::install("affy")
> library(affy)
```

Per leggere tutti i file nella directory, utilizziamo la funzione `ReadAffy()` del pacchetto:

```
> myData <- ReadAffy(celfile.path = "GSE24460_RAW")
```

```
> myData
AffyBatch object
size of arrays=732x732 features (19 kb)
cdf=HG-U133A_2 (22277 affyids)
number of samples=4
number of genes=22277
annotation=hgu133a2
notes=
> str(myData)
Formal class 'AffyBatch' [package "affy"] with 10 slots
  ..@ cdfName          : chr "HG-U133A_2"
  ..@ nrow             : Named int 732
  .. ..- attr(*, "names")= chr "Rows"
  ..@ ncol             : Named int 732
  .. ..- attr(*, "names")= chr "Cols"
  ..@ assayData        :<environment: 0x7fe2a73d9460>
  ..@ phenoData        :Formal class 'AnnotatedDataFrame' [package "Biobase"] with 4 slots
  .. .. ..@ varMetadata      :'data.frame':    1 obs. of  1 variable:
  .. .. .. ..$ labelDescription: chr "arbitrary numbering"
  .. .. ..@ data             :'data.frame':    4 obs. of  1 variable:
  .. .. .. ..$ sample: int [1:4] 1 2 3 4
  .. .. ..@ dimLabels        : chr [1:2] "sampleNames" "sampleColumns"
  .. .. ..@ .__classVersion__:Formal class 'Versions' [package "Biobase"] with 1 slot
  .. .. .. .. ..@ .Data:List of 1
  .. .. .. .. .. ..$ : int [1:3] 1 1 0
  ..@ featureData      :Formal class 'AnnotatedDataFrame' [package "Biobase"] with 4 slots
  .. .. ..@ varMetadata      :'data.frame':    0 obs. of  1 variable:
  .. .. .. ..$ labelDescription: chr(0)
  .. .. ..@ data             :'data.frame':    535824 obs. of  0 variables
  .. .. ..@ dimLabels        : chr [1:2] "featureNames" "featureColumns"
  .. .. ..@ .__classVersion__:Formal class 'Versions' [package "Biobase"] with 1 slot
  .. .. .. .. ..@ .Data:List of 1
  .. .. .. .. .. ..$ : int [1:3] 1 1 0
  ..@ experimentData   :Formal class 'MIAME' [package "Biobase"] with 13 slots
  .. .. ..@ name             : chr ""
  .. .. ..@ lab              : chr ""
  .. .. ..@ contact          : chr ""
  .. .. ..@ title            : chr ""
  .. .. ..@ abstract         : chr ""
  .. .. ..@ url              : chr ""
  .. .. ..@ pubMedIds        : chr ""
  .. .. ..@ samples          : list()
  .. .. ..@ hybridizations   : list()
  .. .. ..@ normControls     : list()
  .. .. ..@ preprocessing    :List of 2
  .. .. .. ..$ filenames  : chr [1:4] "GSE24460_RAW/GSM602658_MCF71.CEL.gz" “GSE24460_RAW/
…
```

L’oggetto acquisito è di tipo "AffyBatch" (come possiamo vedere digitandone semplicemente il nome) e contiene 4 campioni di 22.277 geni ciascuno:

```
> myData
AffyBatch object
size of arrays=732x732 features (19 kb)
cdf=HG-U133A_2 (22277 affyids)
number of samples=4
number of genes=22277
annotation=hgu133a2
notes=
```

Se invece si desidera leggere un batch costituito di un solo (o solo alcuni) file, basta specificarne il nome:

```
> mySomeData <- ReadAffy(filenames = "GSE24460_RAW/GSM602658_MCF71.CEL.gz")
> attributes(mySomeData)
$.__classVersion__
       R   Biobase      eSet AffyBatch
  "3.6.1"  "2.46.0"   "1.3.0"   "1.2.0"
$cdfName
[1] "HG-U133A_2"
$nrow
Rows
 732
$ncol
Cols
 732
$assayData
<environment: 0x7fe2f76be7d8>
$phenoData
An object of class 'AnnotatedDataFrame'
  sampleNames: GSM602658_MCF71.CEL
  varLabels: sample
  varMetadata: labelDescription
$featureData
An object of class 'AnnotatedDataFrame': none
$experimentData
Experiment data
  Experimenter name:
  Laboratory:
  Contact information:
  Title:
  URL:
  PMIDs:
  No abstract available.
  Information is available on: preprocessing
  notes:
$annotation
[1] "hgu133a2"
$protocolData
An object of class 'AnnotatedDataFrame'
  sampleNames: GSM602658_MCF71.CEL
  varLabels: ScanDate
  varMetadata: labelDescription
$class
[1] "AffyBatch"
attr(,"package")
[1] "affy"
```

I dati letti provengono da uno studio sulla cellula del cancro al seno. La funzione `ReadAffy()` utilizza un'altra funzione chiamata `read.affybatch()` che permette la lettura del file CEL insieme ad altri dati rilevanti in un oggetto di tipo AffyBatch. La funzione `ReadAffy()` legge l'elenco di tutti i file CEL (anche i file compressi) presenti nella directory indicata e lo passa alla funzione `read.affybatch()` per creare il batch dei file indicati. È possibile anche fornire argomenti per altre informazioni (come ad es. `phenoData`) e specifici nomi di file CEL da leggere. Quando non vengono passati argomenti, la funzione legge tutti i file CEL nella directory di lavoro.

In questa sezione ci siamo concentrati principalmente sui file CEL di Affymetrix. L'uso di file di testo strutturati (CSV e simili) sta diventando popolare per i dati di espressione genica; ciò consente di leggere i dati utilizzando `read.csv()` o altre funzioni simili, con la possibilità di estrarre le colonne di interesse dai dati di espressione e utilizzarle di conseguenza.

11.2 Costruzione di un oggetto di tipo *ExpressionSet*

La classe "ExpressionSet" in Bioconductor rappresenta una combinazione di diverse fonti di informazione in un'unica struttura di dati. Per un array contiene le intensità, i dati fenotipici, le informazioni sugli esperimenti e le informazioni sulle annotazioni. Quando leggiamo un insieme di file CEL usando la funzione `ReadAffy()` o `read.affybatch()`, viene creato un oggetto di tipo AffyBatch che estende la struttura di ExpressionSet. L'oggetto AffyBatch è un dato a livello di sonda, mentre ExpressionSet è un dato a livello di set di sonde, che viene esteso al livello di sonda da AffyBatch.

A volte, abbiamo valori di intensità sotto forma di tabella, matrice o dati insieme ai dati fenotipici, ai dettagli dell'esperimento e alle annotazioni come oggetti (o file) separati. Dobbiamo creare un oggetto ExpressionSet da questi singoli file partendo da zero per facilitare il lavoro di analisi. Questa sezione illustra come risolvere il problema.

Per procedere, abbiamo bisogno di file con diversi tipi di informazioni, come dati di analisi, metadati fenotipici, annotazioni e metadati delle caratteristiche e una descrizione dell'esperimento. Possiamo creare questi file semplicemente scrivendo su file diversi le opportune componenti dell'oggetto creato nella sezione precedente, insieme alle informazioni sulle annotazioni del chip. Per costruire l'oggetto ExpressionSet, procediamo come di seguito descritto.

Installiamo e carichiamo la libreria *Biobase*, se non è già stato fatto (viene caricata di default quando si carica la libreria *affy*):

```
> BiocManager::install("Biobase")
> library(Biobase)
```

Come file demo di espressione (*exprs*) e dei dati fenotipici (*pData*) useremo i dati incorporati della libreria *Biobase*, la cui posizione nell'ambiente R può essere recuperata come segue:

```
> DIR <- system.file("extdata", package="Biobase")
> exprsLoc <- file.path(DIR, "exprsData.txt")
> pDataLoc <- file.path(DIR, "pData.txt")
```

Leggiamo la variabile *exprs* dal file di testo che contiene i valori di espressione utilizzando la funzione `read.csv()` e controlliamo che l'oggetto letto sia una matrice:

```
> exprs <- as.matrix(read.csv(exprsLoc, header = TRUE, sep = "\t", row.names = 1,
            as.is = TRUE))
> class(exprs)
[1] "matrix"
> dim(exprs)
[1] 500  26
```

Quindi, leggiamo il file di informazioni *pData* sul fenotipo in modo simile:

```
> pData <- read.table(pDataLoc, row.names = 1, header = TRUE, sep = "\t")
> pData <- new("AnnotatedDataFrame", data = pData)
```

Compiliamo le informazioni dell'esperimento, un oggetto della classe MIAME con gli slot per il nome dello sperimentatore, il nome del laboratorio e così via come segue:

```
> exData <- new("MIAME", name="Dr. XYZ", lab="XYZ Lab", contact="abc@xyz", title="",
      abstract="", url="www.xyz")
```

È anche importante conoscere l'annotazione del chip in quanto fa parte dell'oggetto "ExpressionSet" per questi dati; utilizziamo allo scopo l'annotazione del chip hgu133a2. Creiamo un nuovo oggetto "ExpressionSet" usando le informazioni compilate nei passi precedenti come segue:

```
> exampleSet <- new("ExpressionSet", exprs = exprs, phenoData = pData,
      experimentData = exData, annotation = "hgu133a2")
```

Per controllare il nostro oggetto, ne digitiamo semplicemente il nome o controlliamo la struttura con la funzione `str()` come segue:

```
> str (exampleSet)
Formal class 'ExpressionSet' [package "Biobase"] with 7 slots
  ..@ experimentData   :Formal class 'MIAME' [package "Biobase"] with 13 slots
  .. .. ..@ name             : chr "ABCabc"
  .. .. ..@ lab              : chr "XYZ Lab"
  .. .. ..@ contact          : chr "abc@xyz"
  .. .. ..@ title            : chr ""
  .. .. ..@ abstract         : chr ""
  .. .. ..@ url              : chr "www.xyz"
  .. .. ..@ pubMedIds        : chr ""
  .. .. ..@ samples          : list()
  .. .. ..@ hybridizations   : list()
  .. .. ..@ normControls     : list()
  .. .. ..@ preprocessing    : list()
  .. .. ..@ other            : list()
  .. .. ..@ .__classVersion__:Formal class 'Versions' [package "Biobase"] with 1 slot
  .. .. .. .. ..@ .Data:List of 2
  .. .. .. .. .. ..$ : int [1:3] 1 0 0
  .. .. .. .. .. ..$ : int [1:3] 1 1 0
  ..@ assayData        :<environment: 0x7fe2d633d838>
```

Controlliamo la validità dell'oggetto creato, prima di continuare l'analisi, come segue:

```
> validObject(exampleSet)
[1] TRUE
```

Per convertire un oggetto "AffyBatch" in "ExpressionSet", è sufficiente utilizzare i componenti "AffyBatch" direttamente per creare un nuovo oggetto "ExpressionSet", come mostrato nei punti precedenti.

Riassumendo, in questa sezione abbiamo visto come leggere diversi file di informazioni (in una matrice o in un frame di dati) utilizzando la funzione `read.csv()`. I dati di espressione sono organizzati in una matrice che contiene le intensità misurate, mentre i dati fenotipici contengono informazioni sulle condizioni (per esempio, controllo o malattia) dei dati e dei campioni. I dati sperimentali hanno una serie di informazioni formali che non è obbligatorio compilare. Essendo l'ordine delle informazioni molto importante per l'oggetto *exampleSet* finale, abbiamo controllato la validità dell'oggetto creato.

L'annotazione utilizzata per il chip è "hgu133a2" poiché i dati provengono dal chip hgu133a2 Affymetrix. Ad esempio, se i nomi del campione nei dati di espressione e i dati fenotipici sono diversi, la funzione restituirà l'oggetto come non valido. Questi singoli oggetti vengono poi assemblati creando un nuovo oggetto di tipo "ExpressionSet", nel quale ogni componente ha un proprio ruolo. L'oggetto *exprs* contiene i dati di espressione, i dati fenotipici in *pData* riassumono le informazioni sui campioni (ad esempio, il sesso, l'età e lo stato di trattamento a cui si fa riferimento come *covariate*), e le annotazioni forniscono gli strumenti di base per la manipolazione dei dati da perte dei pacchetti di metadati. Il tutto può essere fatto con qualsiasi piattaforma, sia essa Affymetrix o Illumina.

11.3 Gestione di un oggetto di tipo *AffyBatch*

L'oggetto di tipo AffyBatch sarà utilizzato in tutto il capitolo per l'analisi dei dati di espressione. Come abbiamo visto, può essere creato leggendo i file CEL di un esperimento insieme alle altre informazioni. L'oggetto AffyBatch ha varie componenti, che esamineremo in questa sezione, utilizzando l'oggetto *myData* creato in precedenza. Possiamo controllare un oggetto AffyBatch semplicemente digitandone il nome o controllando la struttura come segue:

```
> myData
AffyBatch object
size of arrays=732x732 features (19 kb)
cdf=HG-U133A_2 (22277 affyids)
number of samples=4
number of genes=22277
annotation=hgu133a2
notes=
> str(myData)
Formal class 'AffyBatch' [package "affy"] with 10 slots
  ..@ cdfName          : chr "HG-U133A_2"
  ..@ nrow             : Named int 732
  .. ..- attr(*, "names")= chr "Rows"
  ..@ ncol             : Named int 732
  .. ..- attr(*, "names")= chr "Cols"
  ..@ assayData        :<environment: 0x7f8adb07b078>
  ..@ phenoData        :Formal class 'AnnotatedDataFrame' [package "Biobase"] with 4 slots
  .. .. ..@ varMetadata      :'data.frame':        1 obs. of  1 variable:
  .. .. .. ..$ labelDescription: chr "arbitrary numbering"
…
```

Controlliamo i dati fenotipici dell'oggetto utilizzando le funzioni `pData()` e `phenoData()`:

```
> pData(myData)
                         sample
GSM602658_MCF71.CEL           1
GSM602659_MCF72.CEL           2
GSM602660_MCF7226ng.CEL       3
GSM602661_MCF7262ng.CEL       4
> phenoData(myData)
An object of class 'AnnotatedDataFrame'
  sampleNames: GSM602658_MCF71.CEL GSM602659_MCF72.CEL GSM602660_MCF7226ng.CEL
    GSM602661_MCF7262ng.CEL
  varLabels: sample
  varMetadata: labelDescription
```

Per visualizzare i dati di espressione, basta usare la funzione `exprs()`:

```
> exprs(myData)
       GSM602658_MCF71.CEL GSM602659_MCF72.CEL GSM602660_MCF7226ng.CEL GSM602661_MCF7262ng.CEL
1                       88                  93                      81                      207
2                     7068                5899                    3602                     2931
3                       99                  91                      94                      188
4                     7452                6565                    3850                     3297
5                       87                 143                     119                      159
6                       80                  76                      92                      213
7                     7794                6193                    3879                     3648
8                      101                  99                      87                      194
9                     7493                6344                    3792                     3364
10                      95                  90                      87                      214
…
24999                   88                 116                      98                      262
 [ reached getOption("max.print") -- omitted 510825 rows ]
```

Questi valori possono essere scritti in un file separato con la funzione `write.csv()`. Per ottenere l'annotazione dell'oggetto, utilizziamo la funzione `annotation()`:

```
> annotation(myData)
[1] "hgu133a2"
```

Possiamo infine ottenere i nomi delle sonde o dei campioni mediante le funzioni `probeNames()` e `sampleNames()`:

```
> probeNames(myData)
    [1] "1007_s_at"   "1007_s_at"   "1007_s_at"   "1007_s_at"   "1007_s_at"   "1007_s_at"
    [7] "1007_s_at"   "1007_s_at"   "1007_s_at"   "1007_s_at"   "1007_s_at"   "1007_s_at"
   [13] "1007_s_at"   "1007_s_at"   "1007_s_at"   "1007_s_at"   "1053_at"     "1053_at"
   [19] "1053_at"     "1053_at"     "1053_at"     "1053_at"     "1053_at"     "1053_at"
   [25] "1053_at"     "1053_at"     "1053_at"     "1053_at"     "1053_at"     "1053_at"
…
[99997] "209592_s_at" "209592_s_at" "209592_s_at"
 [ reached getOption("max.print") -- omitted 147900 entries ]
> sampleNames(myData)
[1] "GSM602658_MCF71.CEL"     "GSM602659_MCF72.CEL"     "GSM602660_MCF7226ng.CEL"
[4] "GSM602661_MCF7262ng.CEL"
```

L'oggetto AffyBatch ha una struttura complessa che consiste di molte componenti e sottocomponenti (come si può vedere osservandone la struttura), che — come i dati di espressione — possono essere estratti e scritti in un file separato usando la funzione `write.csv()` o simili.

11.4 Controllo della qualità dei dati da microarray

La qualità dei dati provenienti da microarray può essere influenzata in ogni fase della pipeline di esperimenti. Problemi legati alla qualità potrebbero derivare dall'ibridazione a causa di una fluorescenza irregolare sul chip che causa distribuzioni di intensità variabile. Un legame non specifico o altre ragioni biologiche/tecniche possono creare rumore di fondo nei dati. Un'altra possibile situazione può essere un disegno sperimentale inappropriato che può influenzare il dataset nel suo complesso. L'uso di tali dati porterà ad un'inferenza errata o inconcludente durante l'analisi. Pertanto, occorre garantire la qualità dei dati prima di iniziarne l'analisi. Questo si ottiene controllando gli array, le distribuzioni all'interno degli array, gli errori dei lotti e così via. Ci sono varie analisi e diagrammi diagnostici che possono essere utilizzati per

calcolare queste misure che spiegano la qualità dei dati degli array sotto esame. Questa sezione illustra varie fasi diagnostiche per il controllo della qualità dei dati da microarray.

Per cominciare, abbiamo bisogno dei dati dell'array: useremo gli stessi dati che abbiamo usato nelle sezioni precedenti. Introdurremo anche un nuovo pacchetto R utile per la maggior parte dei processi di valutazione della qualità. Per esaminare la qualità dei dati, creeremo alcuni test diagnostici e grafici di controllo; iniziamo installando e caricando la libreria *arrayQualityMetrics* dal repository Bioconductor:

```
> BiocManager::install("arrayQualityMetrics")
> library(arrayQualityMetrics)
```

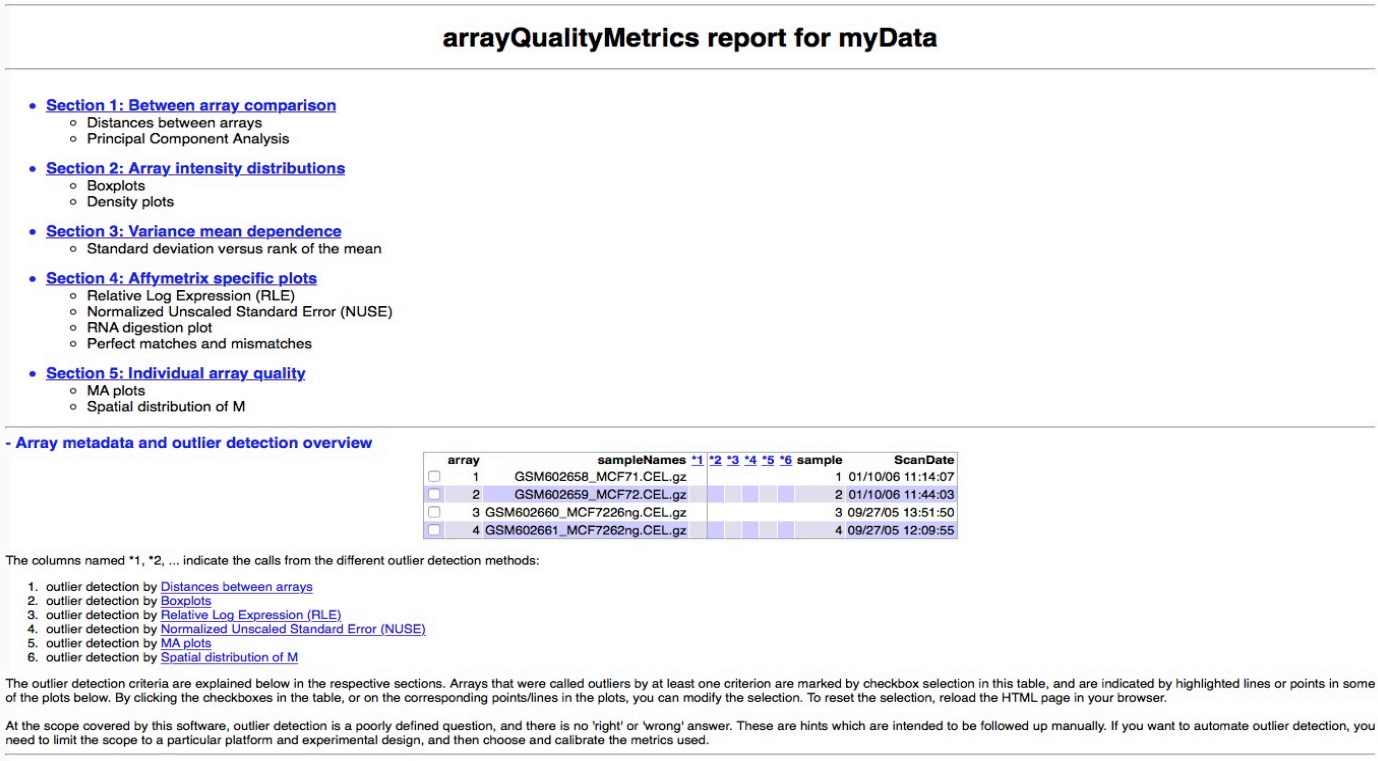
Utilizziamo la funzione `arrayQualityMetrics()` per creare grafici per valutare la qualità dei dati:

```
> arrayQualityMetrics(myData, outdir="quality_assessment")
```

Nella sottocartella "quality_assessment" creata nella directory di lavoro possiamo controllare la pagina HTML riassuntiva (index.html) e i grafici di controllo.

Apriamo il file index.html nel browser mediante il comando:

```
> browseURL(file.path("quality_assessment", "index.html"))
```



arrayQualityMetrics report for myData

- Section 1: Between array comparison
  - Distances between arrays
  - Principal Component Analysis
- Section 2: Array intensity distributions
  - Boxplots
  - Density plots
- Section 3: Variance mean dependence
  - Standard deviation versus rank of the mean
- Section 4: Affymetrix specific plots
  - Relative Log Expression (RLE)
  - Normalized Unscaled Standard Error (NUSE)
  - RNA digestion plot
  - Perfect matches and mismatches
- Section 5: Individual array quality
  - MA plots
  - Spatial distribution of M

- Array metadata and outlier detection overview

	array	sampleNames	*1	*2	*3	*4	*5	*6	sample	ScanDate
☐	1	GSM602658_MCF71.CEL.gz							1	01/10/06 11:14:07
☐	2	GSM602659_MCF72.CEL.gz							2	01/10/06 11:44:03
☐	3	GSM602660_MCF7226ng.CEL.gz							3	09/27/05 13:51:50
☐	4	GSM602661_MCF7262ng.CEL.gz							4	09/27/05 12:09:55

The columns named *1, *2, ... indicate the calls from the different outlier detection methods:

1. outlier detection by Distances between arrays
2. outlier detection by Boxplots
3. outlier detection by Relative Log Expression (RLE)
4. outlier detection by Normalized Unscaled Standard Error (NUSE)
5. outlier detection by MA plots
6. outlier detection by Spatial distribution of M

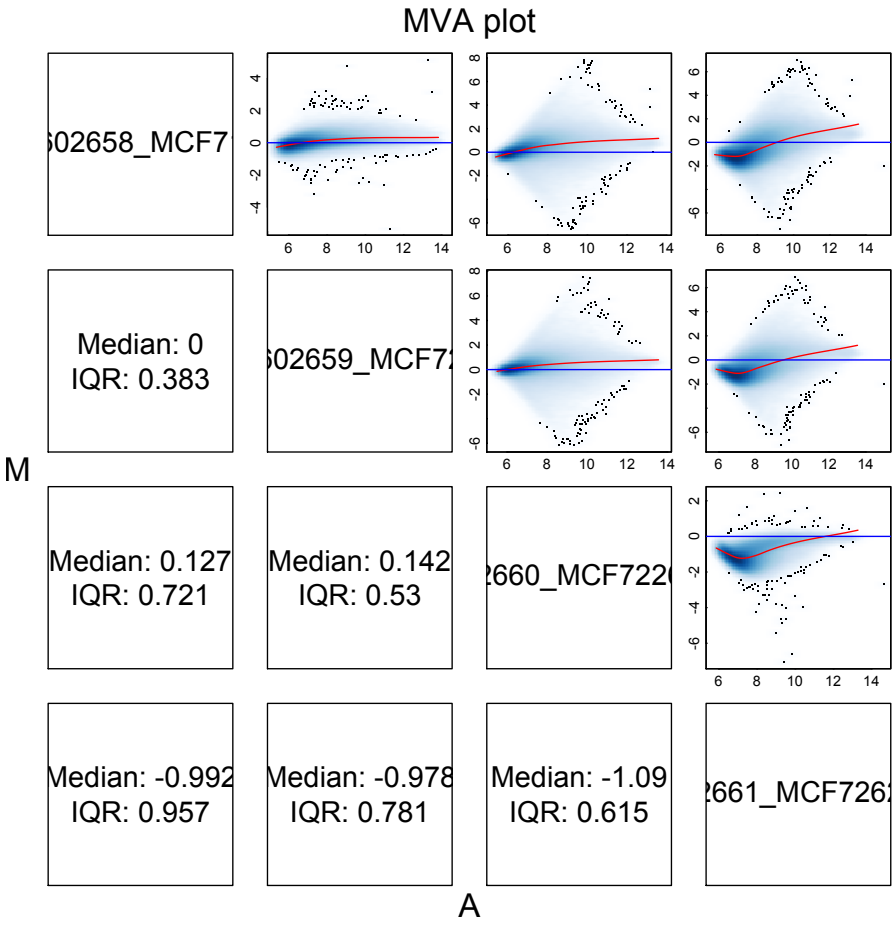
The outlier detection criteria are explained below in the respective sections. Arrays that were called outliers by at least one criterion are marked by checkbox selection in this table, and are indicated by highlighted lines or points in some of the plots below. By clicking the checkboxes in the table, or on the corresponding points/lines in the plots, you can modify the selection. To reset the selection, reload the HTML page in your browser.

At the scope covered by this software, outlier detection is a poorly defined question, and there is no 'right' or 'wrong' answer. These are hints which are intended to be followed up manually. If you want to automate outlier detection, you need to limit the scope to a particular platform and experimental design, and then choose and calibrate the metrics used.

È anche possibile creare i grafici di controllo e le valutazioni singolarmente. Ad esempio, per creare un grafico MA[5], utilizziamo la funzione `MAplot()` come segue:

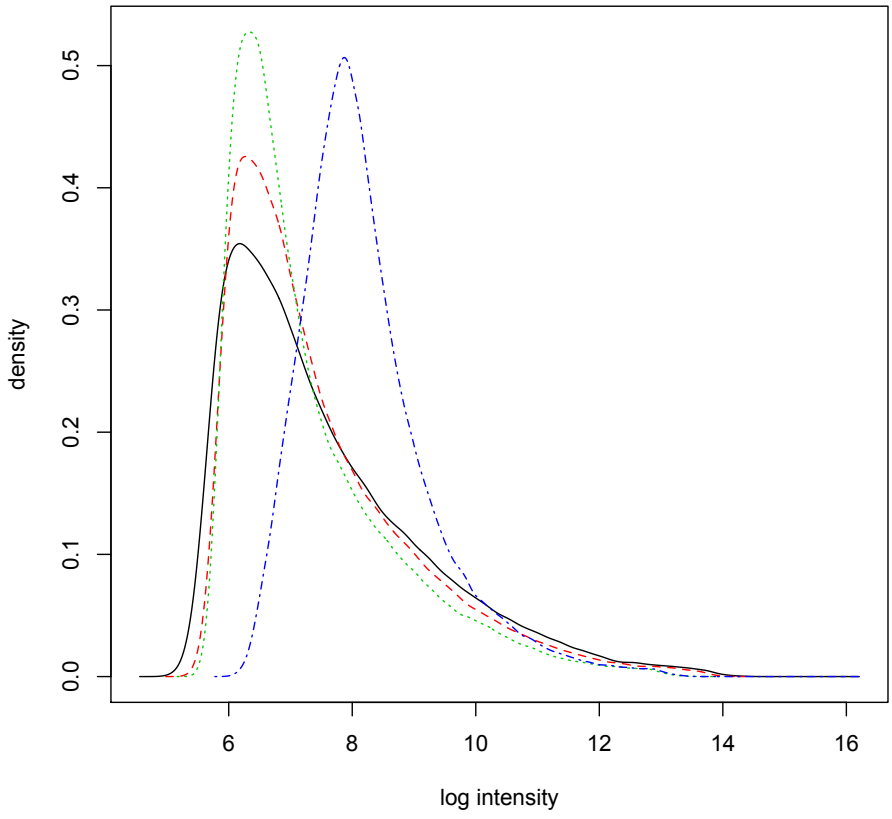
```
> MAplot(myData, pairs=TRUE, plot.method="smoothScatter")
```

[5] Il grafico MA è un diagramma di dispersione in cui l'asse *y* e l'asse *x* mostrano rispettivamente $M=\log_2(R_i/G_i)$ e $A=\log_2(R_i*G_i)$ dove R_i e G_i rappresentano l'intensità dell'i-esimo gene nei campioni R e G.



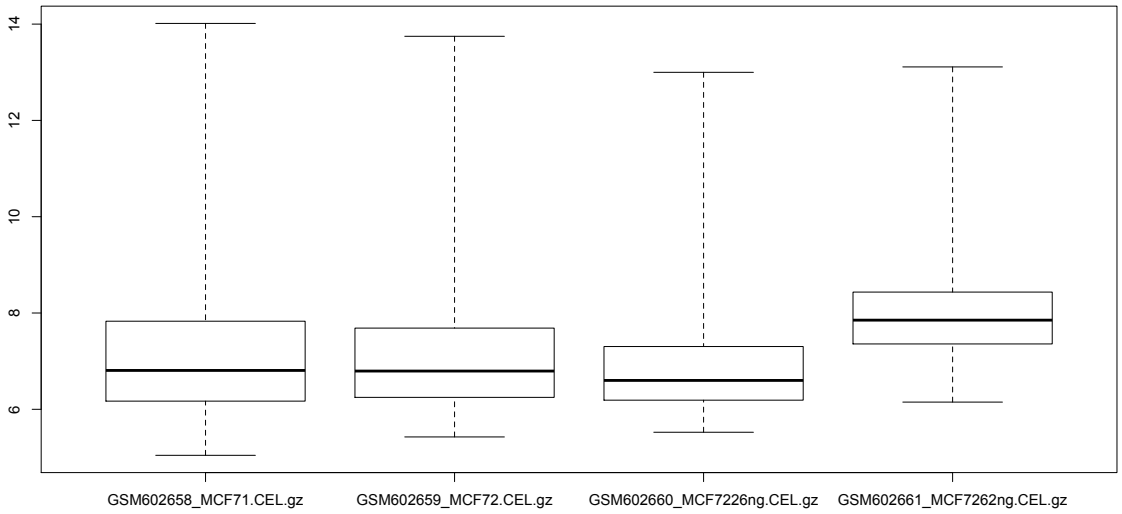
Mentre, per tracciare le log-densità digitiamo il seguente comando:

```
> plotDensity.AffyBatch(myData)
```



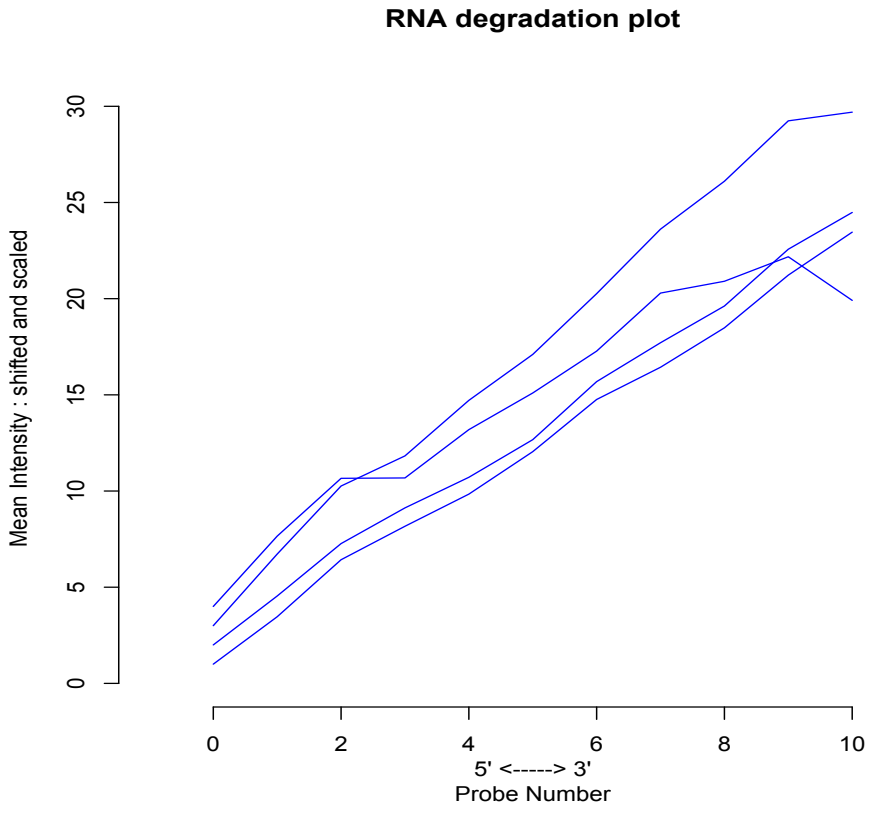
Possiamo creare i boxplot mediante la funzione omonima sull'oggetto AffyBatch *myData* come segue:

```
> boxplot(myData)
```



Per ottenere il diagramma di degradazione dell'RNA, utilizziamo le funzioni `AffyRNAdeg()` e `plotAffyRNAdeg()` e infine controlliamo i dettagli dell'oggetto *rnaDeg* come illustrato di seguito:

```
> rnaDeg <- AffyRNAdeg(myData)
> plotAffyRNAdeg(rnaDeg)
> summaryAffyRNAdeg(rnaDeg)
       GSM602658_MCF71.CEL.gz GSM602659_MCF72.CEL.gz GSM602660_MCF7226ng.CEL.gz
slope                 2.19e+00               2.22e+00                    2.73e+00
pvalue                2.87e-13               5.01e-13                    3.40e-11
       GSM602661_MCF7262ng.CEL.gz
slope                      1.74e+00
pvalue                     1.51e-06
```



La funzione `ArrayQualityMetrics()` prende i dati degli array ed esegue diversi tipi di controlli, che includono la misurazione delle distanze tra gli array, l'analisi delle componenti principali (PCA), i diagrammi di densità, i grafici MA e i diagrammi di degradazione dell'RNA, che consentono anche il rilevamento degli outlier negli array. Una descrizione dettagliata è disponibile nel file HTML prodotto. Ad esempio, per misurare le distanze d_{ij} tra gli array i e j, possiamo usare la seguente formula:

$$d_{ij} = \text{mean} (| I_{ik} - I_{jk} |)$$

dove I_{ik} e I_{jk} sono le misure di intensità della k-esima sonda rispettivamente per gli array i e j.

Il **grafico MA** (*MA plot*), una derivazione del gafico di Bland-Altman, si basa su due componenti. La componente M rappresenta il rapporto tra due canali (o due array), dando così un'indicazione di quale colore è più legato a un dato *spot*. La componente A è una misura dell'intensità (trasformata in $\log_2$) nello *spot*. Tracciando queste due componenti in due dimensioni (di solito M lungo l'asse y e A lungo l'asse x) si ha un'idea della distorsione dell'intensità nei dati.

Ciò significa che le differenze in due canali o due array possono essere utili per rilevare problemi di background o outlier nei dati. Ad esempio, possiamo utilizzarle per il confronto a coppie di array o per confrontare le intensità di due coloranti nei dati a due canali. Una deviazione dalla linea M=0 (distribuzione asimmetrica lungo l'asse M=0) indica una distorsione dell'intensità, outlier o anche geni espressi differenzialmente (DE, *differential expression*). La deviazione nel grafico può essere corretta in qualche misura con la normalizzazione (infatti, è spesso usata come indicatore per la normalizzazione insieme al boxplot, che sarà discusso in seguito). Qualsiasi tendenza verso il lato sinistro (A inferiore) indica la presenza di un background, mentre una tendenza verso il lato destro (A superiore) indica saturazione.

La maggior parte degli spot nel grafico sono di solito intorno alla linea M=0, con piccoli intervalli interquartile attraverso gli array, e probabilmente rappresentano geni non espressi in modo differenziale (non DE). Tuttavia, si dovrebbero raggiungere tali conclusioni solo dopo la correzione del background e la normalizzazione.

Il successivo tipo di grafico che abbiamo prodotto è il **grafico delle intensità**, che mostra le stime della densità dei dati e ne fornisce i diversi tipi di informazioni. I dati validi hanno una forma e un intervallo simile tra gli array. Dati con un elevato rumore di fondo sposteranno l'intera distribuzione verso destra; se la distribuzione mostra una coda destra meno accentuata indica una perdita di segnale. Il rigonfiamento superiore della distribuzione indica la saturazione del segnale. Una forma multimodale (cioè più di un picco) nel grafico rappresenta un artefatto spaziale (come la polarizzazione regionale nell'array).

I **boxplot** di dati di alta qualità mostrano larghezza e posizioni simili e rappresentano la distribuzione delle intensità di segnale nei dati. La distribuzione viene di solito effettuata su scala logaritmica per rendere il grafico più leggibile. Una deviazione importante nel boxplot potrebbe rappresentare un difetto sperimentale o un rumore in quel particolare array. La statistica di Kolmogorov-Smirnov (KS) su queste distribuzioni viene utilizzata per rilevare valori anomali nei dati. Un'altra importante caratteristica del boxplot è che la

deviazione può essere superata, nella maggior parte dei casi, dalla normalizzazione dei dati (che sarà esaminata nelle prossime sezioni).

Il **diagramma di degradazione dell'RNA** fornisce un'indicazione della qualità dei campioni utilizzati nell'ibridazione degli array. Generalmente, gli mRNA hanno una certa durata di vita, dopo la quale sono degradati e quindi non sono efficaci per misurare i livelli di espressione. La degradazione inizia all'estremità 5' del filamento e si sposta verso l'estremità 3'. Pertanto, come effetto di questa degradazione, le intensità dovrebbero essere più basse all'estremità 5' rispetto che alla 3'. L'espressione (misura dell'intensità) di tutte le sonde su un array fornisce il livello di degradazione nel campione, che viene rappresentato come un diagramma dove le sonde sono numerate in sequenza dall'estremità 5' all'estremità 3' della molecola. Quindi, il grafico delle intensità dovrebbe mostrare una tendenza al rialzo lungo i numeri delle sonde (più degradazione all'estremità 5', quindi bassa intensità, e viceversa). Se le linee nel grafico seguono un trend costante i dati sono validi; una deviazione indica problemi con il campione utilizzato per l'ibridazione.

Tutte questi grafici, più alcuni altri con una breve descrizione, sono riportati nel file HTML prodotto nella directory indicata. Il significato generale di ogni grafico è spiegato nel file HTML stesso; tuttavia, è consigliabile esaminarli in maniera critica, poiché ogni esperimento può essere diverso dall'altro e dovrebbe essere analizzato di conseguenza.

11.5 Generazione di dati di espressione artificiali

Lo sviluppo di nuovi metodi di analisi dei dati di espressione richiede test e controlli di performance adeguati su grandi dataset di alta qualità ottenuti da molte condizioni sperimentali. Ciò richiede dati di riferimento con parametri noti. Tali dati provenienti da esperimenti non sono solitamente disponibili, e l'esecuzione di tali esperimenti in laboratorio non è economica. Pertanto, è necessario generare set di dati sintetici ben caratterizzati, che permettano di testare accuratamente gli algoritmi di calcolo in modo rapido e riproducibile. È pratica comune utilizzare set di dati simulati (artificiali) per tali scopi, come vedremo in questa sezione.

Prima di iniziare a generare i dati artificiali, dobbiamo deciderne le caratteristiche in termini di limiti (superiore e inferiore), frazione di geni differenzialmente espressi (DE), quantità di dati e parametri statistici associati, come vedremo meglio alla fine della sezione.

Generiamo un dataset di 35.000 geni con il 2% di geni DE come di seguito indicato. Prima di iniziare, dobbiamo installare e caricare la libreria *madsim* dal repository CRAN:

```
> install.packages("madsim")
> library(madsim)
```

Definiamo ora la prima serie di parametri statistici per il processo di simulazione e distribuzione dei dati artificiali negli array come segue:

```
> fparams <- data.frame(m1 = 7, m2 = 7, shape2 = 4, lb = 4, ub = 14, pde = 0.02,
      sym = 0.5)
```

Impostiamo il secondo set di parametri statistici che definiscono il livello di espressione nei geni:

```
> dparams <- data.frame(lambda1 = 0.13, lambda2 = 2, muminde = 1, sdde = 0.5)
> sdn <- 0.4
> rseed <- 50
```

Definiamo quindi il numero di geni desiderati nei dati di espressione:

```
> n <- 35000
```

Ora, generiamo i dati sintetici come segue:

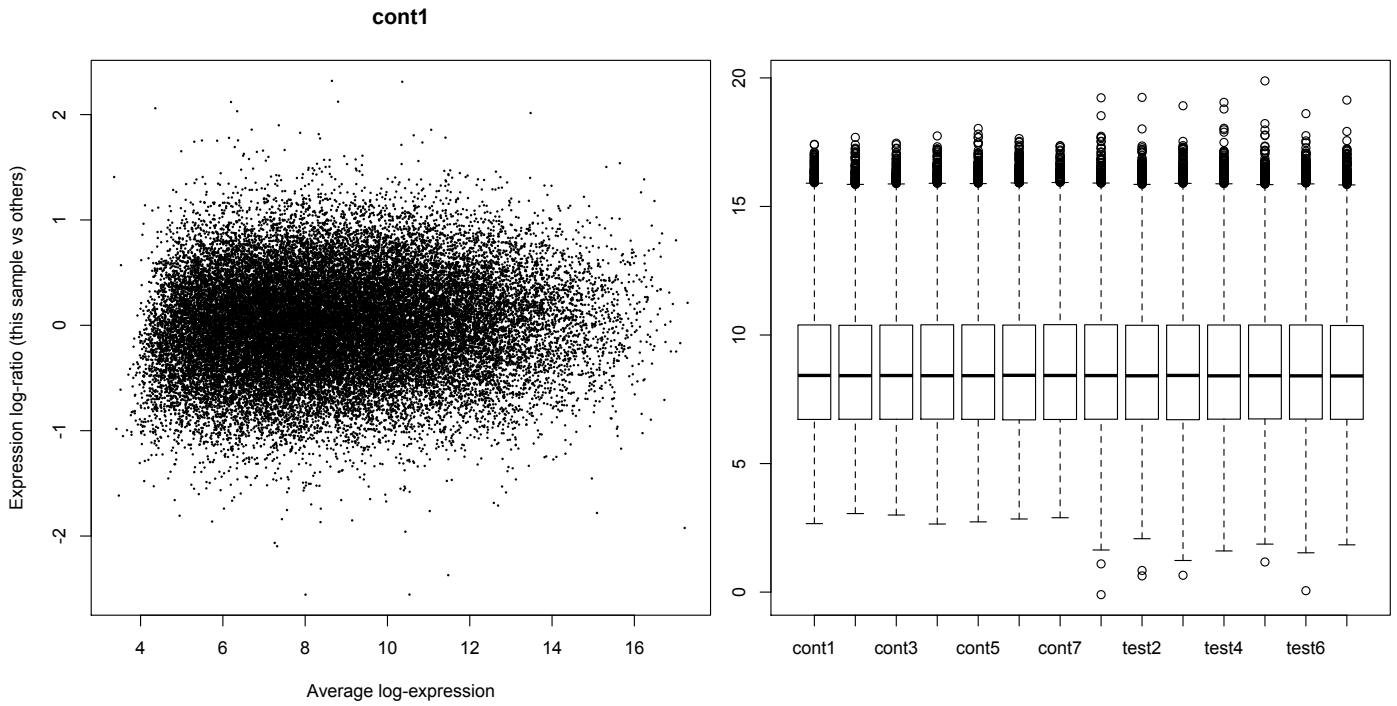
```
> myData <- madsim(mdata=NULL, n=35000, ratio=0, fparams, dparams, sdn, rseed)
```

Osservando la struttura dell'oggetto creato da `madsim()`, si può notare che ha tre componenti:

```
> str(myData)
List of 3
 $ xdata: num [1:35000, 1:14] 11.12 6.52 8.82 10.25 4.46 ...
  ..- attr(*, "dimnames")=List of 2
  .. ..$ : NULL
  .. ..$ : chr [1:14] "cont1" "cont2" "cont3" "cont4" ...
 $ xid  : num [1:35000, 1] 0 0 0 0 0 0 0 0 -1 0 ...
 $ xsd  : num 2.5
```

Per visualizzare i dati creiamo un grafico MA, ad esempio per il campione 1, e tracciamo anche il boxplot dei dati:

```
> library(limma)
> plotMA(myData[[1]], 1)
> boxplot(myData[[1]])
```



Il pacchetto *madsim* genera dati per due condizioni biologiche quando le caratteristiche sono note in termini di parametri statistici. I parametri utilizzati nel modello consentono all'utente di generare dati con caratteristiche variabili. Essi utilizzano i seguenti quattro componenti per generare i valori di espressione di un gene: (*a*) i livelli di espressione dei geni non-DE; (*b*) i livelli di espressione dei geni DE; (*c*) rumore; (*d*) rumore tecnico.

Il livello di espressione complessivo è una funzione (nel nostro caso, la somma) di queste quattro componenti. La funzione `madsim()` utilizza una distribuzione *Beta* per generare *n* valori compresi tra 0 e 1. I parametri di forma nell'argomento `fparams` definiscono questa distribuzione *Beta* — una distribuzione statistica continua, definita da due parametri. Essa viene poi scalata per adattarsi ai limiti superiore e inferiore definiti. Viene estratto un insieme di geni DE selezionati in modo casuale secondo l'argomento `pde` in `fparams`. Le proprietà per generare i valori di espressione sono specificate usando l'argomento `dparams` (deviazione standard e media) e servono alla simulazione dell'espressione differenziale dei geni.

Ci sono altri modi per simulare i dati microarray. Il pacchetto chiamato *optBiomarker* ha la funzione `simData()` che può anch'essa simulare dati artificiali. Il pacchetto può essere installato dal repository CRAN standard (ha bisogno del software BWidget). Per saperne di più, consultare il manuale di aiuto del pacchetto e i dettagli disponibili all'indirizzo https://cran.r-project.org/web/packages/optBiomarker/.

11.6 Normalizzazione dei dati

I microarray sono metodi ad alto rendimento che misurano i livelli di espressione di migliaia di geni simultaneamente. Ogni campione corrisponde a condizioni differenti. Una piccola differenza nelle quantità di RNA e/o errori sperimentali può far variare il livello di intensità da un replicato all'altro, indipendentemente dall'espressione biologica dei geni. La gestione di questo problema intrinseco richiede la corretta normalizzazione dei dati, in modo da minimizzare gli errori tecnici e rendere i dati comparabili. In questa sezione esploreremo alcuni dei molti metodi disponibili in R per la normalizzazione dei dati da microarray.

Per iniziare, dobbiamo definire i dati da normalizzare come oggetto "ExpressionSet" o "AffyBatch". Useremo l'oggetto *myData* di tipo "AffyBatch" contenente i dati sul cancro al seno, creandolo nuovamente come abbiamo già visto nella sezione inziale del capitolo:

```
> library(affy)
> myData <- ReadAffy(celfile.path="GSE24460_RAW")
```

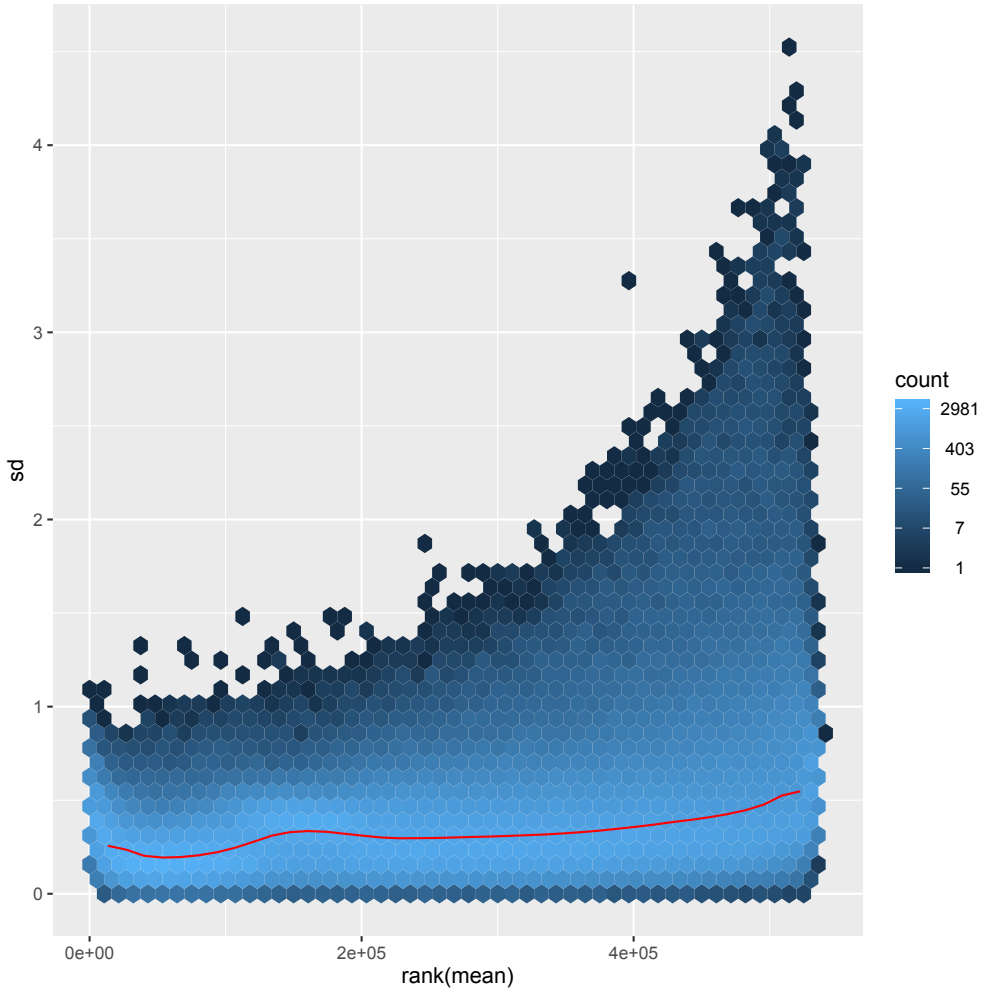
Ci sono molti tipi possibili di normalizzazione. Ci concentreremo su tre metodi: le normalizzazioni *VSN*, *Loess* e *Quantile*.

Per effettuare una normalizzazione *VSN* è necessario il pacchetto omonimo che andiamo a installare e caricare nella sessione R come segue:

```
> BiocManager::install("vsn")
> library(vsn)
```

Eseguiamo e verifichiamo la normalizzazione *VSN* sull'oggetto AffyBatch *myData* con i seguenti comandi:

```
> myData.vsn <- vsn2(myData)
vsn2: 535824 x 4 matrix (1 stratum).
Please use 'meanSdPlot' to verify the fit.
> meanSdPlot(myData.vsn)
```



L'oggetto *myData.vsn* è di tipo vsn; la funzione `meanSdPlot()` traccia un grafico delle deviazioni standard dei dati di input rispetto alle medie (vedi immagine precedente). La deviazione standard e la media sono calcolate riga per riga dalla matrice di espressione contenuta in *myData.vsn*. Il diagramma di dispersione (*scatterplot*) della deviazione standard rispetto alla media consente di verificare visivamente se esiste una dipendenza della deviazione standard (o varianza) dalla media.

La linea rossa rappresenta lo stimatore della mediana: se non c'è una dipendenza della varianza dalla media, la linea dovrebbe essere approssimativamente orizzontale.

Creiamo un boxplot per i dati normalizzati VSN (grafico (B) nella pagina seguente) e confrontiamolo con il boxplot dei dati non normalizzati della sezione 11.4 (grafico (A) nella pagina seguente) per verificarne le differenze:

```
> myData.vsn <- justvsn(myData) # creates an AffyBatch object
> boxplot(myData.vsn, las=2) # vertical labels
```

La normalizzazione *Loess* utilizza la libreria *affy*. Per effettuare la normalizzazione *Loess*, utilizziamo la funzione `normalize.AffyBatch.loess()` come segue:

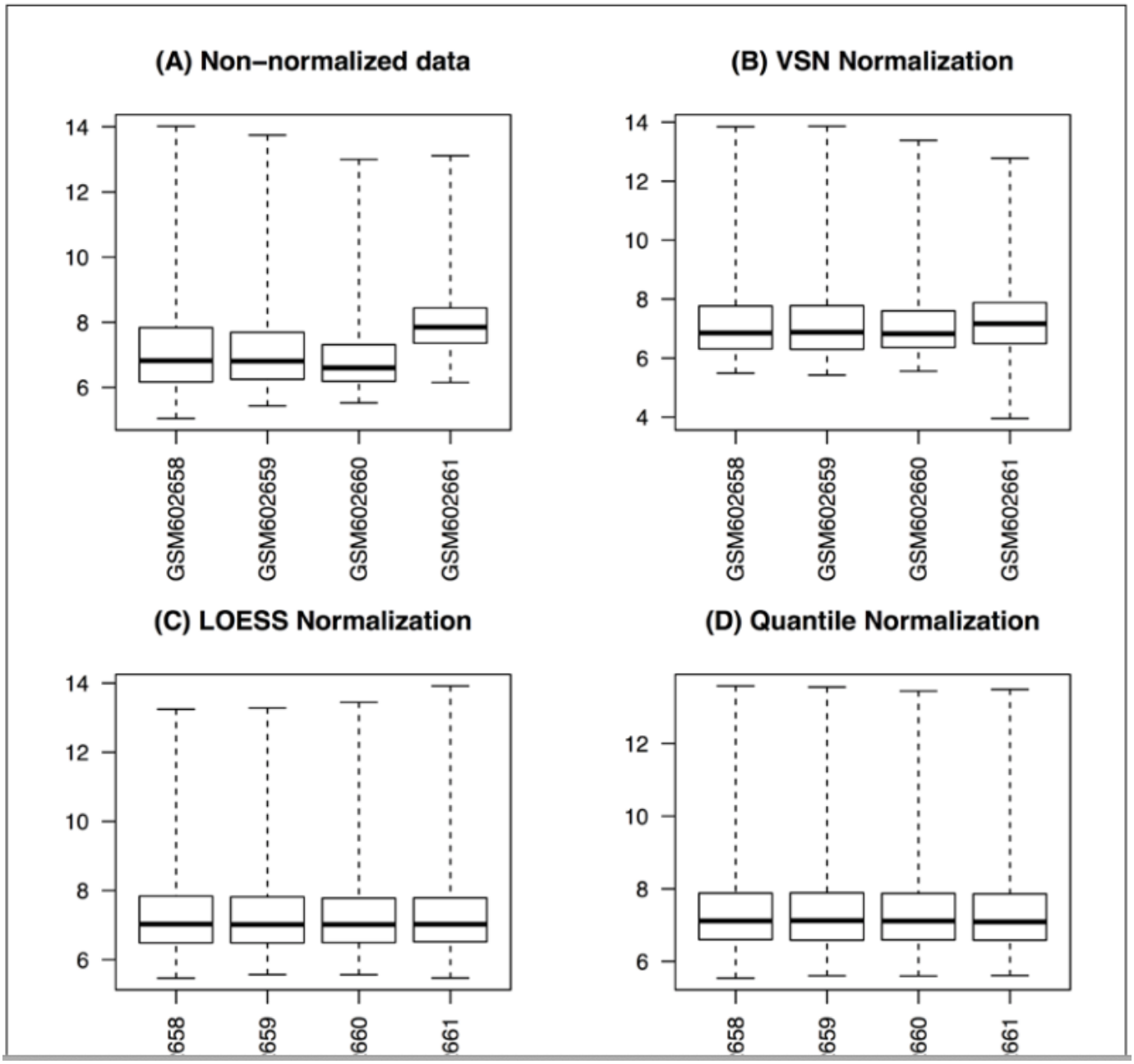
```
> myData.loess <- normalize.AffyBatch.loess(myData)
```

La normalizzazione *Quantile* utilizza anch'essa la libreria *affy* mediante la funzione `normalize.AffyBatch.quantiles()`:

```
> myData.quantile <- normalize.AffyBatch.quantiles(myData)
```

Tracciamo il boxplot anche per questi due metodi (grafici (C) e (D) nella figura) in modo simile a quello *VSN* e confrontiamoli per esaminare le differenze tra i dati normalizzati e non normalizzati:

```
> boxplot(myData.loess, las=2)
> boxplot(myData.quantile, las=2)
```



Possiamo infine eseguire un secondo ciclo di controllo della qualità dei dati normalizzati e osservare l'effetto della normalizzazione rispetto ai dati non normalizzati utilizzando la funzione `arrayQualityMetrics()`.

La stabilizzazione e normalizzazione della varianza (*VSN*, Variance Stabilization and Normalization) si basa sul presupposto che la varianza dei dati da microarray dipende dall'intensità del segnale ed esiste una trasformazione che mantiene la varianza approssimativamente costante. Ciò significa che il metodo *VSN* trova nei dati una trasformazione delle misure di intensità in modo da mantenere la varianza dell'intensità approssimativamente indipendente dalla sua media. La funzione `justvsn()` vista prima è in realtà un *wrapper* per la funzione `vsn2()` nella libreria *vsn* che restituisce un oggetto AffyBatch.

Il secondo metodo, la normalizzazione *Loess*, utilizza una regressione ponderata localmente per normalizzare i dati (*Local Polynomial Regression Fitting*). Il metodo adatta una curva di *smoothing* ad un set di dati. Il grado di *smoothing* è determinato dal parametro della larghezza della finestra. Una larghezza maggiore risulta in una curva più uniforme, mentre una finestra più piccola risulta in una maggiore variazione locale. La funzione `normalize.AffyBatch.loess()` utilizza effettivamente la funzione `loess()` di R per adattare e smussare i dati. La dimensione della finestra usata di default è di 2/3, ma può essere modificata con l'argomento `span`.

Infine, la normalizzazione dei quantili utilizza il concetto più semplice di regolazione dei quantili della distribuzione in un array per renderli tutti uguali con un centro mediano comune. Questo fa sì che gli istogrammi degli array si assomiglino.

Ci sono alcune altre fasi di pre-elaborazione che possono essere eseguite sui dati grezzi di un microarray. Esse includono la correzione del *background*, la trasformazione log e così via. Sono per lo più inclusi nei metodi di normalizzazione. La correzione del *background* nel pacchetto *affy* può essere esaminata con la funzione "bgcorrect.methods":

```
> bgcorrect.methods()
[1] "bg.correct" "mas"        "none"       "rma"
```

La trasformazione log può essere impostata sui valori booleani VERO o FALSO cambiando l'argomento `log.it` nelle funzioni di normalizzazione del pacchetto *affy*.

Ci sono altri metodi di normalizzazione all'interno della libreria *affy*, come `qspline`, `invariantset`, `contrasts` e così via. Per verificare i vari metodi disponibili, digitare il seguente comando nella sessione R con la libreria *affy* precaricata:

```
> normalize.AffyBatch.methods()
[1] "constant"          "contrasts"         "invariantset"      "loess"
[5] "methods"           "qspline"           "quantiles"         "quantiles.robust"
```

Il metodo di normalizzazione è una questione di scelta e di necessità. Dipende molto dalla piattaforma di dati e dagli esperimenti biologici. I metodi illustrati in questa sezione sono i più utilizzati in bioinformatica.

11.7 Come superare i *batch effects* nei dati di espressione

I *batch effects* (effetti dei lotti) riguardano gli errori sistematici causati quando i campioni vengono processati in lotti diversi, che rappresentano le differenze non biologiche tra i campioni in un esperimento. La ragione può essere la differenza nella preparazione del campione o nel protocollo di ibridazione, e così via. Possono essere ridotti, in una certa misura, da un'attenta progettazione sperimentale, ma non possono essere eliminati completamente a meno che lo studio non venga eseguito in un unico lotto. I *batch effects* rendono difficile il compito di combinare i dati di diversi lotti per la loro elaborazione complessiva, riducendo la *potenza* dell'analisi statistica dei dati. Il problema richiede un'adeguata pre-elaborazione prima che i lotti vengano combinati. Vedremo di seguito tali tecniche di pre-elaborazione dei dati provenienti da più lotti.

Per iniziare scaricheremo un dataset ("bladderbatch", che consiste di cinque lotti) che mostra l'effetto dei lotti senza alcuna pre-elaborazione, insieme a tutte le librerie necessarie:

```
> BiocManager::install(c("sva", "bladderbatch"))
> library(sva) # contains batch removing utilities
> library(bladderbatch) # the data to be used
> data(bladderdata)
```

Estraiamo la matrice dei dati di espressione e i dati fenotipici, e diamo un'occhiata alle informazioni fenotipiche digitando i seguenti comandi:

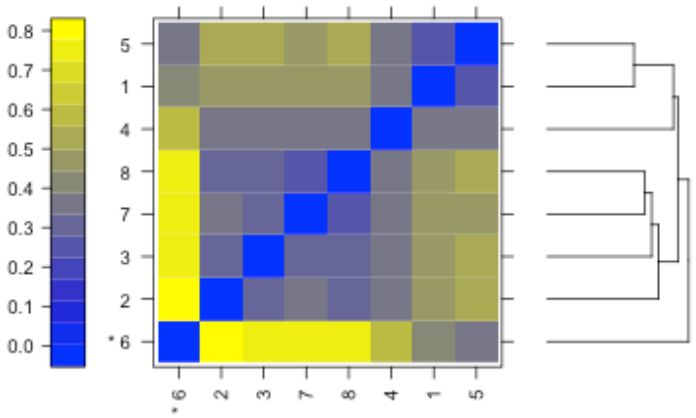
```
> edata <- exprs(bladderEset)
> pheno <- pData(bladderEset)
> pheno
             sample  outcome batch cancer
GSM71019.CEL      1   Normal     3 Normal
GSM71020.CEL      2   Normal     2 Normal
GSM71021.CEL      3   Normal     2 Normal
GSM71022.CEL      4   Normal     3 Normal
GSM71023.CEL      5   Normal     3 Normal
GSM71024.CEL      6   Normal     3 Normal
GSM71025.CEL      7   Normal     2 Normal
GSM71026.CEL      8   Normal     2 Normal
GSM71028.CEL      9 sTCC+CIS     5 Cancer
GSM71029.CEL     10 sTCC-CIS     2 Cancer
…
```

Possiamo vedere che i primi otto campioni sono cellule normali ma divise in due lotti (lotto numero 2 e 3). Utilizzeremo questi otto campioni per mostrare come rimuovere i *batch effects*. Per selezionare questi campioni, utilizziamo la seguente funzione:

```
> myData <- bladderEset[,sampleNames(bladderEset)[1:8]]
```

Per dare un'occhiata al *batch effect*, eseguiamo un controllo di qualità dei dati con la funzione `arrayQualityMetrics()` nella cartella "qc_be" ed esaminiamo la heatmap e l'albero di clustering prodotto per i campioni come mostrato di seguito:

```
> library(arrayQualityMetrics) # if not already loaded
> arrayQualityMetrics(myData, outdir="qc_be")
> browseURL(file.path("qc_be", "index.html"))
```



Creiamo ora la matrice di analisi del modello (tramite la funzione `model.matrix`; consultarne l'help per i dettagli) per il set di dati come segue (si noti che sono utilizzate solo la prima e la terza colonna della matrice del modello in quanto i dati hanno la sola condizione "cancer"):

```
> pData(myData)
             sample outcome batch cancer
GSM71019.CEL      1  Normal     3 Normal
GSM71020.CEL      2  Normal     2 Normal
GSM71021.CEL      3  Normal     2 Normal
GSM71022.CEL      4  Normal     3 Normal
GSM71023.CEL      5  Normal     3 Normal
GSM71024.CEL      6  Normal     3 Normal
GSM71025.CEL      7  Normal     2 Normal
GSM71026.CEL      8  Normal     2 Normal
> mod1 <- model.matrix(~as.factor(cancer), data=pData(myData))[,c(1,3)]
```

Definiamo quindi i lotti per i campioni (nella variabile *batch*) ed estraiamo la matrice dei dati di espressione *edata* (dove devono essere rimossi i *batch effects*) dall'oggetto di tipo "ExpressionSet" *myData* come segue:

```
> batch <- pData(myData)$batch
> edata <- exprs(myData)
```

Una volta che tutti gli oggetti sono pronti, applichiamo la funzione `ComBat()` (che regola l'effetto dei lotti utilizzando un modello empirico di Bayes) al modello *mod1* come segue:

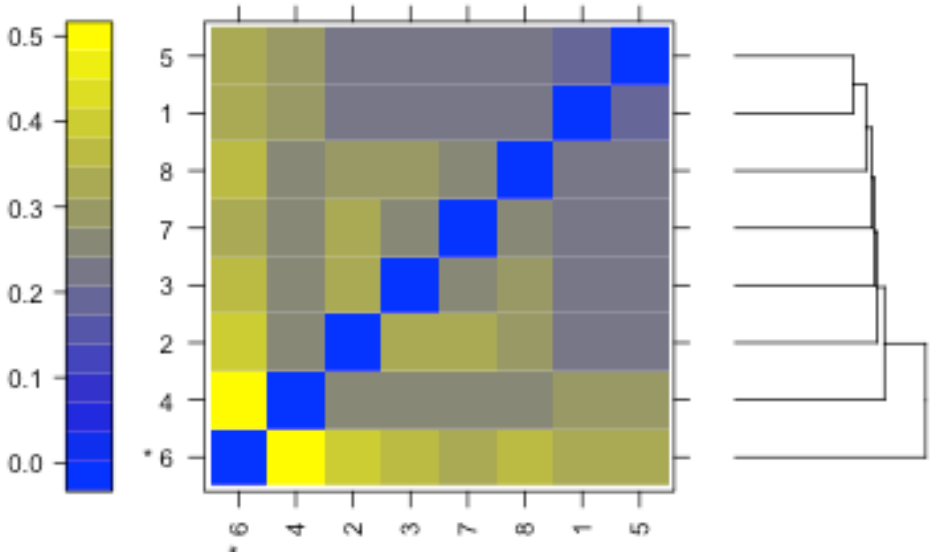
```
> combat_edata <- ComBat(dat=edata, batch=batch, mod=mod1, par.prior=TRUE)
Found2batches
Adjusting for0covariate(s) or covariate level(s)
Standardizing Data across genes
Fitting L/S model and finding priors
Finding parametric adjustments
Adjusting the Data
```

Creiamo ora un oggetto "ExpressionSet" con tutti i dati di input originali, tranne la matrice dei dati di espressione — che viene sostituita dalla matrice ottenuta come risultato della funzione `ComBat()`:

```
> myData2 <- myData
> exprs(myData2) <- combat_edata
```

Eseguiamo infine nuovamente la funzione `arrayQualityMetrics()` per verificare l'eliminazione dell'effetto dei lotti sul nuovo oggetto *myData2* e diamo un'occhiata alla heatmap e all'albero di clustering generati dalla funzione per verificare che i *batch effects* nei dati siano stati eliminati:

```
> arrayQualityMetrics(myData2, outdir = "qc_nbe")
> browseURL(file.path("qc_nbe", "index.html"))
```

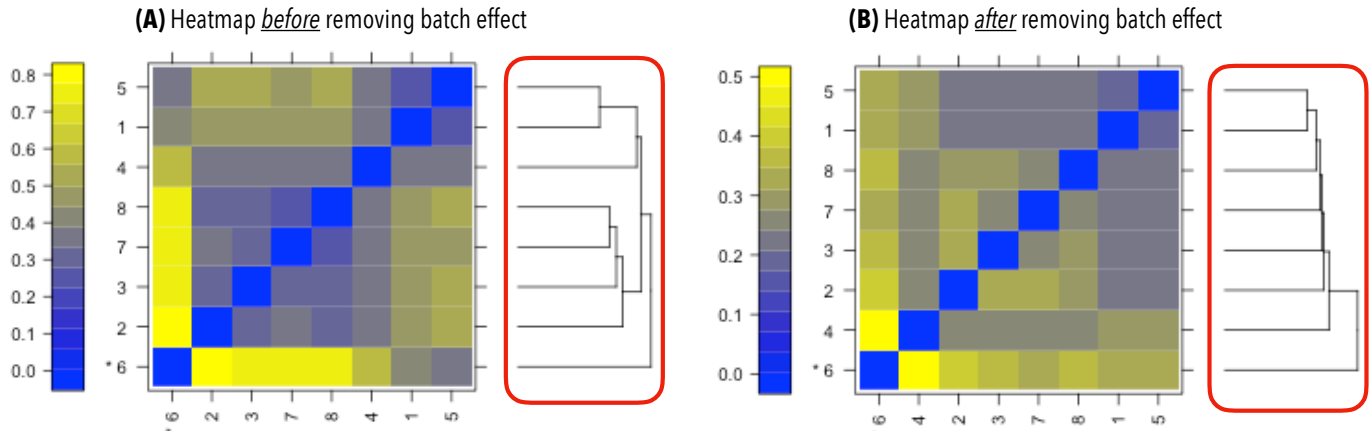


I dati che abbiamo utilizzato provengono da uno studio sul cancro della vescica, dove il profilo di espressione è stato utilizzato per esaminare i modelli di espressione genica nel *carcinoma a cellule transizionali superficiali* (STCC) con e senza circostante *carcinoma in situ* (CIS). I dati sono stati prodotti in date diverse e, di conseguenza, mostrano tipicamente notevoli effetti a lotti; questi possono portare a conclusioni biologiche confuse o errate, a causa dell'influenza di artefatti tecnici.

La funzione `ComBat()` accetta la matrice di dati che contiene i valori di espressione, il numero del lotto per ogni campione nei dati di input e la matrice del modello che rappresenta le informazioni dei campioni (per gli altri argomenti di input opzionali, digitare `?ComBat` nella console R). La funzione utilizza un modello empirico di Bayes per combinare i lotti; essa stima i parametri per la localizzazione e la regolazione della scala (LS) di ogni lotto per ogni gene in modo indipendente. La messa in comune di informazioni per più geni in ogni lotto fa emergere il modello che i geni con espressioni simili seguono. Queste informazioni vengono poi utilizzate per regolare i lotti al fine di eliminarne gli effetti.

I modelli nella heatmap possono mostrare due fattori: l'effetto biologico previsto o il *batch effect* non previsto. I lotti possono essere osservati nella heatmap e nell'albero di clustering. Possiamo vedere le ramificazioni e gli schemi di colore distinti nella heatmap per ogni lotto, mentre dopo aver eseguito la funzione `ComBat()` alcuni di questi rami negli alberi di clustering si fondono.

Nella figura seguente sono riportate le heatmap per i primi otto campioni (cellule normali) dei dati della vescica, che mostrano gli alberi di clustering prima (A) e dopo (B) la rimozione del *batch effect* per enfatizzarne visivamente le differenze:



11.8 Analisi esplorativa dei dati con la PCA

Supponiamo di aver esaminato circa 20.000 geni umani in 10 campioni e di aver ottenuto una matrice di 20.000 × 10 misure. Ora, immaginiamo questi 20.000 geni (le *features*, o caratteristiche, da analizzare) come una nuvola di punti in uno spazio multidimensionale. Scoprire un *pattern* tra i geni in un tale spazio multidimensionale è alquanto difficile; occorre perciò trasformare lo spazio multidimensionale in uno con meno dimensioni per spiegare e rappresentare graficamente i *pattern* nei dati. Organizzare e combinare le caratteristiche per spiegare la massima variabilità dei dati può contribuire a raggiungere questo obiettivo. L'Analisi delle Componenti Principali (PCA) è un metodo che permette di raggiungere questo obiettivo eseguendo un'analisi di covarianza tra i fattori. Trova le componenti ortogonali che rappresentano i dati e ogni componente (chiamata componente principale) che rappresenta la dimensione in cui le caratteristiche sono più estese.

La PCA proietta i dati su uno spazio con un numero inferiore di dimensioni e può essere utilizzata come metodo esplorativo per scopi come la ricerca di modelli (*pattern*) nei dati e la riduzione del rumore. In questa sezione ci occuperemo della PCA per i dati da microarray.

Utilizzeremo il dataset sul cancro al seno per eseguire la PCA sui dati e mostrare le prime componenti principali:

```
> library(affy)
> myData <- ReadAffy(celfile.path="GSE24460_RAW")
```

Quindi, selezioniamo i dati per l'analisi PCA estraendo i valori di espressione genica dal dataset:

```
> myData.pca <- exprs(myData)
> dim(myData.pca)
[1] 535824      4
```

Abbiamo 732×732 = 535.824 valori di espressione (per tutti i geni e tutte le sonde) ripartiti su 4 campioni. Eseguiamo la PCA sui valori di espressione mediante la funzione `prcomp()`:

```
> myPCA <- prcomp(myData.pca, scale=TRUE)
```

Per controllare le componenti principali, diamo un'occhiata al riepilogo degli oggetti creati come segue:

```
> myPCA
Standard deviations (1, .., p=4):
[1] 1.9305486 0.4601109 0.1782064 0.1718213

Rotation (n x k) = (4 x 4):
                                PC1        PC2        PC3        PC4
GSM602658_MCF71.CEL     0.4981676 -0.5377661  0.2850430 -0.6175655
GSM602659_MCF72.CEL     0.5023913 -0.4495057 -0.4387600  0.5941694
GSM602660_MCF7226ng.CEL 0.5043762  0.3979131  0.6710449  0.3700925
GSM602661_MCF7262ng.CEL 0.4950118  0.5919606 -0.5252988 -0.3586183
> summary(myPCA)
Importance of components:
                          PC1     PC2     PC3     PC4
Standard deviation     1.9305 0.46011 0.17821 0.17182
Proportion of Variance 0.9317 0.05293 0.00794 0.00738
Cumulative Proportion  0.9317 0.98468 0.99262 1.00000
```

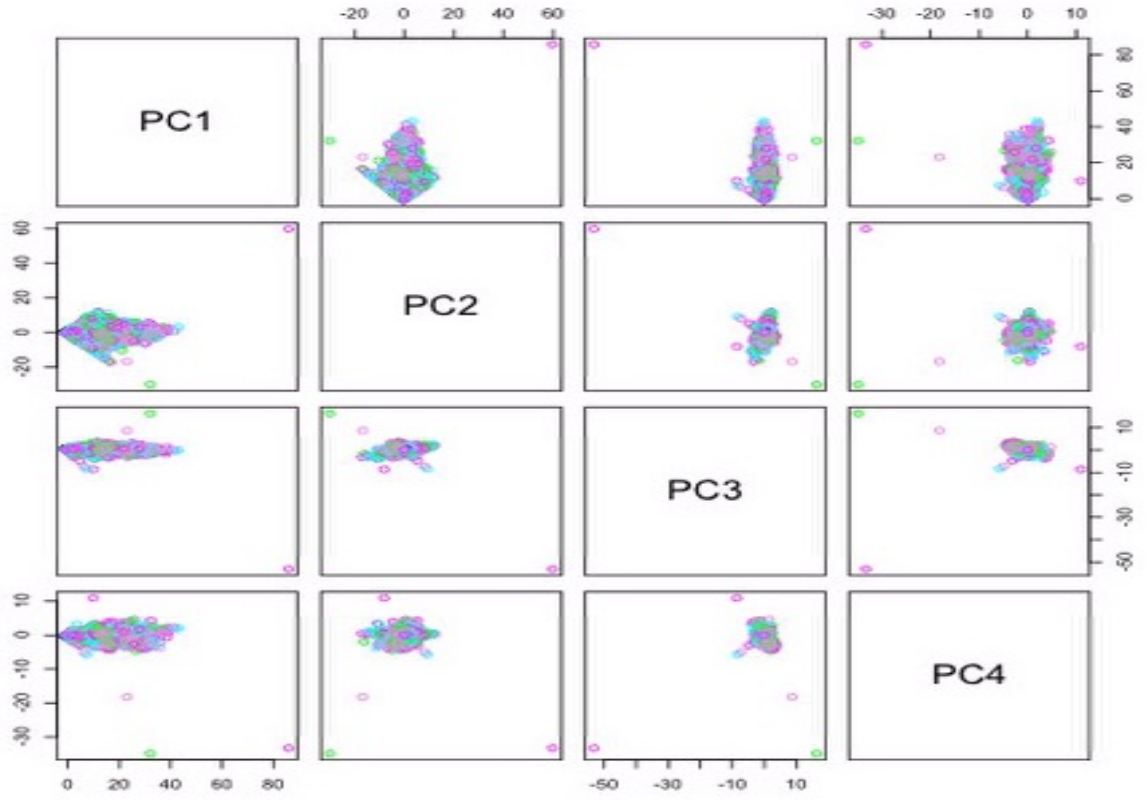
Creiamo un vettore di colori per ogni campione:

```
> colors <- c("green","cyan","violet","magenta")
```

Per tracciare il grafico delle componenti principali, utilizziamo la funzione `pairs()` come segue (si noti che potrebbe richiedere un certo tempo a seconda della dimensione dei dati):

```
> pairs(myPCA$x, col=colors)
```

La funzione `prcomp()` esegue l'analisi delle componenti principali sulla matrice dei dati. Essa restituisce le componenti principali, le loro deviazioni standard (le radici quadrate degli autovalori) e la rotazione (contenente gli autovettori). La seguente schermata mostra come i dati appaiono quando vengono visualizzati lungo le coppie di componenti principali selezionate:



11.9 Ricerca dei geni differenzialmente espressi

A livello di genoma, il contenuto di ogni cellula è identico, il che significa che geni simili sono presenti (con poche eccezioni) in cellule simili. La domanda che sorge allora è: cosa rende le cellule (per esempio, i campioni di controllo e quelli trattati) diverse l'una dall'altra? La risposta è nel concetto di *espressione genica differenziale* (DE, *differential gene expression*). In ogni cellula viene espressa solo una frazione di un genoma, e questo fenomeno di espressione selettiva di geni basati su tipi di cellule è alla base del concetto di espressione genica differenziale. Pertanto, è importante trovare quali geni mostrano l'espressione differenziale in una particolare cellula. Questo si ottiene confrontando la cellula in studio con un riferimento, solitamente chiamato *controllo*. Questa sezione illustra come trovare i geni differenziamente espressi (DE) per una cellula in base ai livelli di espressione dei controlli e delle cellule di trattamento.

Per eseguire l'elaborazione abbiamo bisogno dei dati di espressione normalizzati per i campioni di trattamento e di controllo. Un numero maggiore di repliche è sempre statisticamente rilevante per tali scopi analitici.

Come accennato in precedenza, la normalizzazione rende gli array comparabili e, quindi, l'utilizzo dei dati normalizzati rende il processo di ricerca delle differenze imparziale e scientificamente verificabile. In seguito useremo i dati normalizzati col metodo *Quantile*. Oltre a questo, abbiamo bisogno dei dettagli dell'esperimento e del fenotipo, che fanno parte dell'oggetto "AffyBatch" o "ExpressionSet". Useremo anche il pacchetto R *limma* che ospita uno dei metodi più popolari per l'analisi dell'espressione genica differenziale. Per gli esempi, useremo i normali dati *affy* preprocessati per il cancro al colon dal pacchetto *antiProfilesData*:

```
> BiocManager::install(c("limma", "antiProfilesData"))
> library(affy) # package for affy data handling
> library(antiProfilesData) # Package containing input data
> library(affyPLM) # normalization package for eSet
> library(limma) # limma analysis package
```

Prendiamo i dati sul cancro al colon dal pacchetto *antiProfilesData* e selezioniamo i primi 16 campioni, che rappresentano i campioni normali e quelli tumorali (otto ciascuno), come abbiamo fatto in precedenza digitando i seguenti comandi:

```
> data(apColonData)
> myData <- apColonData[, sampleNames(apColonData)[1:16]]
> myData_quantile <- normalize.ExpressionSet.quantiles(myData)
```

Prepariamo una matrice di progetto basata sui dettagli dell'esperimento e sui dati fenotipici come segue:

```
design <- model.matrix(~0 + pData(myData)$Status)
```

Adattiamo un modello lineare utilizzando i dati di espressione e la matrice *design* e controlliamo il modello risultante come segue:

```
> fit <- lmFit(myData_quantile,design)
> fit
An object of class "MArrayLM"
```

Avendo il modello lineare, calcoliamo le statistiche relative utilizzando la funzione `eBayes()` che, dato un modello lineare adattato sui dati di un microarray, calcola la statistica *t*, la statistica F e i log-odds dell'espressione differenziale con il modello empirico di Bayes degli errori standard verso un valore comune:

```
> fitE <- eBayes(fit)
```

Il risultato viene riordinato (in senso decrescente in base ai log-odds dell'espressione differenziale, colonna B della tabella *gene_list*) e i geni più significativi (cioè più differenzialmente espressi; nell'esempio 5.339) possono essere estratti come segue:

```
> gene_list <- topTable(fitE, adjust="fdr", sort.by="B", number=Inf)
> dim(gene_list)
[1] 5339    6
> gene_list
                    logFC       AveExpr             t      P.Value    adj.P.Val            B
210372_s_at   5.200189e+00  2.936906e+00  1.858682e+01 2.082126e-12 1.111647e-08 17.909405944
205470_s_at   6.255458e+00  3.462435e+00  1.452345e+01 9.365043e-11 2.499998e-07 14.665745205
204855_at     1.334243e+01  8.332598e+00  1.251168e+01 8.820153e-10 1.260731e-06 12.634754710
219795_at     6.773997e+00  3.269661e+00  1.245411e+01 9.445448e-10 1.260731e-06 12.571572337
220133_at     3.223162e+00  1.561743e+00  1.212799e+01 1.399296e-09 1.494168e-06 12.207825072
…
1553534_at   -2.792301e-04 -4.404650e-02 -2.503923e-03 9.980325e-01 9.984065e-01 -6.748846042
237320_at     4.826854e-05 -2.136316e-01  2.714129e-04 9.997867e-01 9.999727e-01 -6.748849286
207067_s_at  -7.696790e-06  3.347532e-01 -3.479360e-05 9.999727e-01 9.999727e-01 -6.748849324
```

Possiamo aggiungere delle condizioni sui *p*-value o altre condizioni per filtrare i geni DE di interesse. Ad esempio, per ottenere solo i geni DE che abbiano un *p*-value inferiore a 0.01 (1.211 geni) digiteremo:

```
> filtered_gene_list <- gene_list[gene_list$adj.P.Val<0.01,]
> dim(filtered_gene_list)
[1] 1211    6
```

Infine, se vogliamo solo i geni DE con un *p*-value inferiore a 0.01 e un log-FC (log Fold Change = fattore logaritmico di espressione differenziale) maggiore di 2 (cioè un rapporto di aumento/diminuzione superiore a $4 = 2^2$ volte, essendo 2 la base dei logaritmi utilizzata) digiteremo:

```
> filtered_gene_list <- gene_list[gene_list$adj.P.Val<0.01
      & abs(gene_list$logFC)>2,]
> dim(filtered_gene_list)
[1] 558   6
```

In questo caso, vengono filtrati 558 geni dagli iniziali 5.339 differenzialmente espressi.

La libreria *limma* viene utilizzata per analizzare i dati di espressione genica, in particolare l'uso di modelli lineari per l'analisi dei dati di espressione genica. Il termine *limma* sta per *linear models for microarray data* (modelli lineari per i dati da microarray). Implementa diversi metodi di modellizzazione lineare per i dati da microarray che possono essere usati per identificare i geni DE. L'approccio descritto in questa sezione adatta prima di tutto un modello lineare per ogni gene nei dati del microarray. Successivamente, utilizza un metodo empirico di Bayes per valutare l'espressione differenziale. Questo calcola il test statistico e il punteggio

corrispondente sotto forma di *p*-value, log fold change (log-FC) e altro. La funzione `lmFit()` che abbiamo utilizzato prende come input la matrice dei dati di espressione e la matrice di progetto; l'input può anche essere un oggetto "ExpressionSet". Si noti che abbiamo assegnato manualmente la matrice *design* di progetto per sfruttare le conoscenze pregresse sui dati. Tuttavia, tale matrice può anche essere creata direttamente dai dati fenotipici normalizzati come segue:

```
> design <- model.matrix(~-1 + factor(pData(myData_quantile)$SubType))
```

La matrice di progetto descrive la condizione dell'esperimento in ciascuna delle sue colonne. Nel nostro caso, abbiamo solo due condizioni ("normal" e "adenoma") e quindi funziona anche la matrice a colonna singola con l'indicazione dell'esperimento appropriato (`$SubType`). Per saperne di più sulla creazione di matrici di progetto, digitare `help(model.matrix)` nella console R. Per confronti multipli tra le diverse condizioni, si può usare la funzione `makeContrasts()` (vedere il relativo aiuto `?makeContrasts`). La matrice di contrasto può essere usata allo stesso modo della matrice di progetto.

Ci sono diversi altri pacchetti per l'analisi dell'espressione genica differenziale. Uno di questi è EMA, che può essere utilizzato per molti scopi (lo utilizzeremo per il clustering e la generazione di heatmap). Per saperne di più sul'uso del pacchetto EMA, controllare il file di aiuto corrispondente come segue:

```
> BiocManager::install(c("siggenes","gcrma"))
> install.packages("EMA")
> library(EMA)
> help(package="EMA")
```

11.10 Lavorare con condizioni sperimentali multiple

Fino ad ora abbiamo eseguito analisi con due soli gruppi sperimentali, tipicamente "trattamento" e "controllo". Tuttavia, ci sono progetti sperimentali in cui potremmo aver bisogno di confrontare più di due gruppi. Per illustrare la problematica, consideriamo una situazione in cui abbiamo tre condizioni e dobbiamo confrontarle sistematicamente l'una con l'altra. Questa sezione illustra in dettaglio come procedere.

Useremo un altro dataset del pacchetto *leukemiasEset*. I dati provengono da 60 campioni di midollo osseo di pazienti con uno dei quattro principali tipi di leucemia (ALL, AML, CLL e CML) e controlli non leucemici. Tuttavia, a scopo dimostrativo, useremo solo tre campioni di ciascuna di queste categorie.

Per prima cosa, installiamo e carichiamo le librerie necessarie (in questo caso, *leukemiasEset*) come segue:

```
> BiocManager::install("leukemiasEset")
> library(limma)
> library(leukemiasEset)
> data(leukemiasEset)
> pheno <- pData(leukemiasEset)
> pheno
             Project     Tissue LeukemiaType                LeukemiaTypeFullName                                  Subtype
GSM330151.CEL   Mile1 BoneMarrow          ALL         Acute Lymphoblastic Leukemia                 c_ALL/Pre_B_ALL without t(9 22)
…
GSM331677.CEL   Mile1 BoneMarrow          NoL Non-leukemia and healthy bone marrow
```

Selezioniamo ora tre campioni da ogni set utilizzando gli indici corrispondenti (ALL=1:3, AML=13:15, CLL=25:27, Controllo=49:51):

```
> pheno[c(1:3,13:15,25:27,49:51),1:4]
              Project     Tissue LeukemiaType                LeukemiaType FullName
GSM330151.CEL   Mile1 BoneMarrow          ALL         Acute Lymphoblastic Leukemia
GSM330153.CEL   Mile1 BoneMarrow          ALL         Acute Lymphoblastic Leukemia
GSM330154.CEL   Mile1 BoneMarrow          ALL         Acute Lymphoblastic Leukemia
GSM330532.CEL   Mile1 BoneMarrow          AML           Acute Myelogenous Leukemia
GSM330546.CEL   Mile1 BoneMarrow          AML           Acute Myelogenous Leukemia
GSM330559.CEL   Mile1 BoneMarrow          AML           Acute Myelogenous Leukemia
GSM330933.CEL   Mile1 BoneMarrow          CLL         Chronic Lymphocytic Leukemia
GSM330934.CEL   Mile1 BoneMarrow          CLL         Chronic Lymphocytic Leukemia
GSM330969.CEL   Mile1 BoneMarrow          CLL         Chronic Lymphocytic Leukemia
GSM331660.CEL   Mile1 BoneMarrow          NoL Non-leukemia and healthy bone marrow
GSM331661.CEL   Mile1 BoneMarrow          NoL Non-leukemia and healthy bone marrow
GSM331663.CEL   Mile1 BoneMarrow          NoL Non-leukemia and healthy bone marrow
> myData <- leukemiasEset[, sampleNames(leukemiasEset)[c(1:3, 13:15, 25:27, 49:51)]]
> myData
ExpressionSet (storageMode: lockedEnvironment)
assayData: 20172 features, 12 samples
  element names: exprs, se.exprs
protocolData
  sampleNames: GSM330151.CEL GSM330153.CEL ... GSM331663.CEL (12 total)
  varLabels: ScanDate
  varMetadata: labelDescription
phenoData
  sampleNames: GSM330151.CEL GSM330153.CEL ... GSM331663.CEL (12 total)
  varLabels: Project Tissue ... Subtype (5 total)
  varMetadata: labelDescription
featureData: none
experimentData: use 'experimentData(object)'
Annotation: genemapperhgu133plus2
```

Creiamo la matrice di progetto sulla base delle variabili di condizione, come spiegato in precedenza, utilizzando i dati fenotipici come segue:

```
> design <- model.matrix(~0 + factor(pData(myData)$LeukemiaType))
```

Rinominiamo le colonne della matrice di progetto e la controlliamo come segue:

```
> colnames(design) <- unique(as.character(pData(myData)$LeukemiaType))
> design
   ALL AML CLL NoL
1    1   0   0   0
2    1   0   0   0
3    1   0   0   0
4    0   1   0   0
5    0   1   0   0
6    0   1   0   0
7    0   0   1   0
8    0   0   1   0
9    0   0   1   0
10   0   0   0   1
11   0   0   0   1
12   0   0   0   1
```

Procediamo ora con l'applicazione di un modello lineare ai dati:

```
> fit <- lmFit(myData, design)
```

Per i confronti a coppie, creiamo una matrice dei confronti come segue:

```
> contrast.matrix <- makeContrasts(NoL-ALL, NoL-AML, NoL-CLL, levels = design)
> contrast.matrix
      Contrasts
Levels NoL - ALL NoL - AML NoL - CLL
   ALL        -1          0         0
   AML         0         -1         0
   CLL         0          0        -1
   NoL         1          1         1
```

Adattiamo il modello lineare utilizzando questa matrice con l'aiuto del seguente comando:

```
> fit2 <- contrasts.fit(fit, contrast.matrix)
```

Eseguiamo l'analisi empirica di Bayes del modello digitando il comando:

```
> fit2 <- eBayes(fit2)
```

Estraiamo i geni differenzialmente espressi per ciascuno dei confronti a coppie usando l'argomento `coef` nella funzione `topTable()`. Ad esempio, per il primo confronto a coppie impostiamo `coef=1` per confrontare i controlli con i casi di leucemia linfoblastica acuta (ALL):

```
> DE_genes_1 <- topTable(fit2,adjust="fdr",sort.by="B",number=Inf, coef=1)
> dim(DE_genes_1)
[1] 20172     6
> DE_genes_1
                     logFC   AveExpr           t      P.Value    adj.P.Val             B
ENSG00000152078  4.51050741  4.856523  28.1398756 4.463747e-11 9.004270e-07 14.0147239244
ENSG00000117519 -4.18517540  4.791585 -22.7388822 3.878292e-10 3.911645e-06 12.6973810483
ENSG00000145850  4.14223626  4.507655  17.3863633 5.759942e-09 2.925048e-05 10.7278231532
…
ENSG00000113196 -0.09279082  4.254868  -0.3762420 7.143743e-01 8.646296e-01 -6.7112607239
 [ reached 'max' / getOption("max.print") -- omitted 3506 rows ]
```

Otteniamo così 20.172 geni differenzialmente espressi per la coppia Controlli—ALL.

Se vogliamo filtrare solo quelli aventi *p*-value < 0.01 digitiamo:

```
> filtered_DE_genes_1 <- DE_genes_1[DE_genes_1$adj.P.Val < 0.01,]
> dim(filtered_DE_genes_1)
[1] 252   6
```

Come si può vedere, restano solo 252 geni degli iniziali 20.172.

La differenza principale in questa analisi multipla rispetto al confronto di due soli gruppi è nell'uso della matrice dei confronti e dell'argomento `coef` nella funzione `topTable()`. La matrice permette il confronto a coppie per il calcolo dei *p*-value: in tal modo il modello adattato restituisce un insieme di *p*-

value per ogni confronto, consentendo l'eventuale estrazione dei geni con la soglia di *p*-value desiderata. Modulando opportunamente il valore del parametro `coef` si possono eseguire i confronti tra tutte le coppie dei campioni presenti nel dataset.

11.11 Gestione dei dati di serie temporali

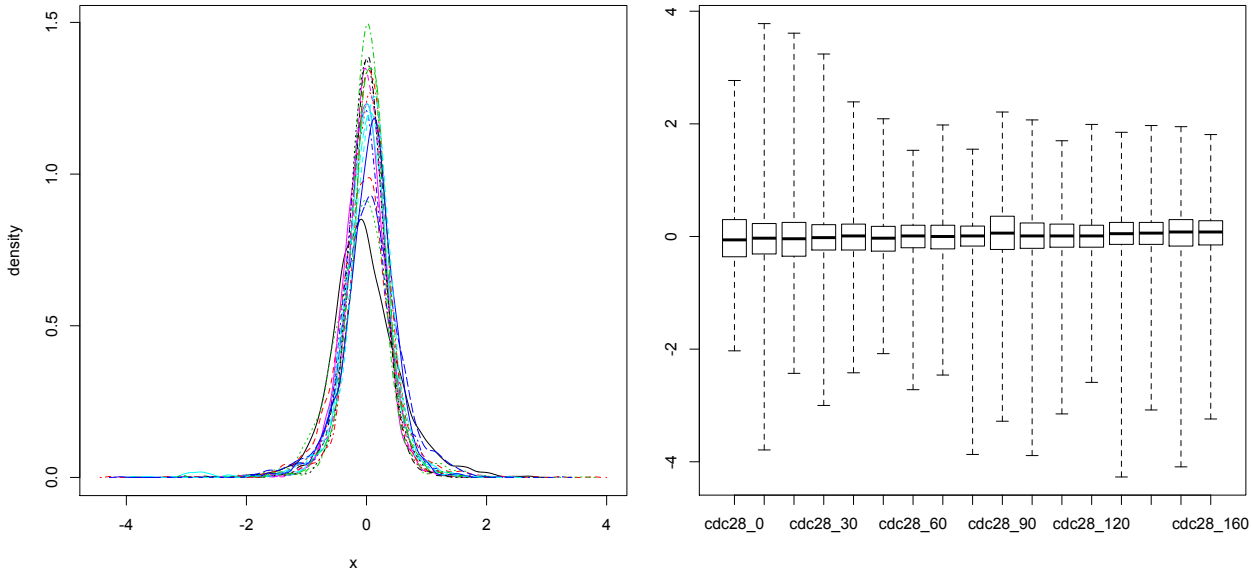
Avendo sinora discusso di diverse condizioni o trattamenti durante gli esperimenti, l'uso del tempo come condizione sperimentale è tra i metodi più diffusi. Un campione di cellule viene sottoposto a un certo trattamento e la sua espressione può cambiare nel corso del tempo. A titolo di esempio, possiamo considerare che durante lo sviluppo di cellule staminali o embrionali, l'espressione dei geni in diversi punti del tempo possa variare. Il trattamento di tali dati di espressione nel corso del tempo, anche se non molto diverso dal protocollo standard descritto in precedenza, necessita di piccole modifiche nell'analisi.

In questa sezione useremo il dataset del lievito (*Saccharomyces cerevisae*) del pacchetto Bioconductor *Mfuzz*, che installiamo e attiviamo insieme al pacchetto *affyPLM* dovendo gestire dati di tipo "ExpressionSet":

```
> BiocManager::install("Mfuzz")
> library(Mfuzz)
> library(affyPLM)
> data(yeast)
> yeast
ExpressionSet (storageMode: lockedEnvironment)
assayData: 3000 features, 17 samples
  element names: exprs
protocolData: none
phenoData
  sampleNames: cdc28_0 cdc28_10 ... cdc28_160 (17 total)
  varLabels: index label time
  varMetadata: labelDescription
featureData: none
experimentData: use 'experimentData(object)'
Annotation: YAL001C ...
```

Controlliamo la qualità dei dati tracciandone il grafico di densità e il boxplot come segue:

```
> plotDensity(yeast)
> boxplot(yeast)
```

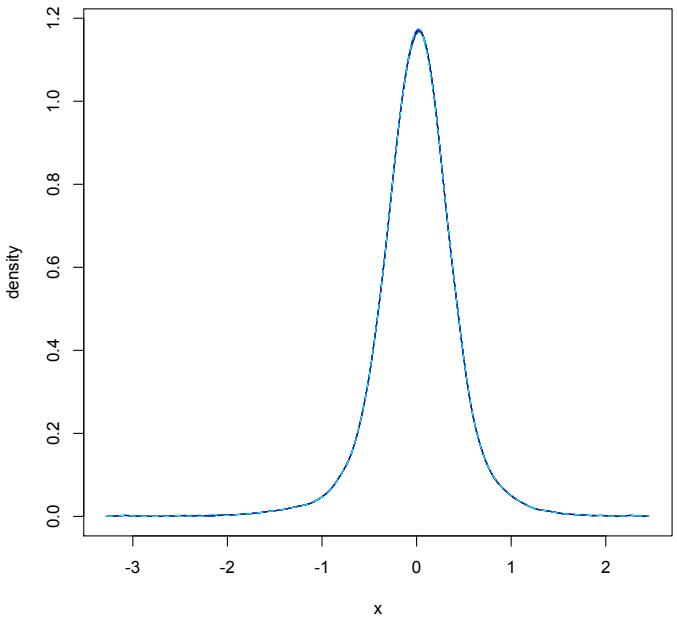


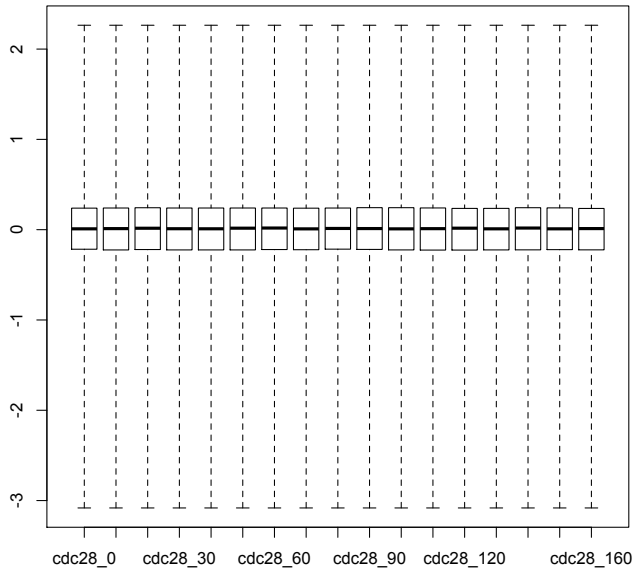
Per normalizzare i dati, usiamo la funzione `normalize.ExpressionSet.quantiles()` del pacchetto *affyPLM* come segue:

```
> yeast_norm <- normalize.ExpressionSet.quantiles(yeast)
```

Eseguiamo nuovamente la valutazione della qualità dei dati normalizzati, come prima, mediante il grafico della densità e il boxplot:

```
> plotDensity(yeast_norm)
> boxplot(yeast_norm)
```





Per controllare gli attributi dei dati, leggiamo i dettagli dell'oggetto ExpressionSet *yeast_norm*:

```
> pData(yeast_norm)
          index      label time
cdc28_0       0   cdc28_0    0
cdc28_10      1  cdc28_10   10
cdc28_20      2  cdc28_20   20
cdc28_30      3  cdc28_30   30
cdc28_40      4  cdc28_40   40
cdc28_50      5  cdc28_50   50
cdc28_60      6  cdc28_60   60
cdc28_70      7  cdc28_70   70
cdc28_80      8  cdc28_80   80
cdc28_90      9  cdc28_90   90
cdc28_100    10 cdc28_100  100
cdc28_110    11 cdc28_110  110
cdc28_120    12 cdc28_120  120
cdc28_130    13 cdc28_130  130
cdc28_140    14 cdc28_140  140
cdc28_150    15 cdc28_150  150
cdc28_160    16 cdc28_160  160
```

Si può vedere che i dati sono costituiti da 17 campioni, a partire dal tempo 0 al tempo 160 con un intervallo di 10 unità.

Creiamo ora la matrice di progetto per una serie temporale in cui ci saranno controlli e tempi, considerando i controlli al tempo T0 da confrontare con i restanti tempi T1…T16 (rinominiamo di conseguenza le colonne della matrice come T0…T16):

```
> times <- pData(yeast_norm)$time
> times <- as.factor(times)
> design <- model.matrix(~0 + factor(pData(yeast_norm)$time))
> colnames(design)[1:17] <- paste("T", 0:16, sep="")
```

Creiamo la matrice dei confronti usando il punto 0 come controllo e i tempi restanti come trattamento (di solito viene fatto per i campioni della stessa coltura), come segue:

```
> cont <- makeContrasts(T0-T1, T0-T2, T0-T3, T0-T4, T0-T5, T0-T6, T0-T7, T0-T8, T0-T9,
      T0-T10, T0-T11, T0-T12, T0-T13, T0-T14, T0-T15, T0-T16, levels=design)
```

Utilizziamo questa matrice per adattare il modello lineare, seguito dalla funzione eBayes per calcolare le statistiche:

```
> fit <- lmFit(yeast_norm, cont)
> fitE <- eBayes(fit)
```

Filtriamo infine i geni più differenzialmente espressi utilizzando la funzione `topTable()` come segue:

```
> DE_genes <- topTable(fitE, adjust="fdr", sort.by="F", number=100)
> DE_genes <- DE_genes[DE_genes$adj.P.Val< 0.05,]
```

Visualizziamo i primi 10 geni solo per i primi quattro confronti dei tempi:

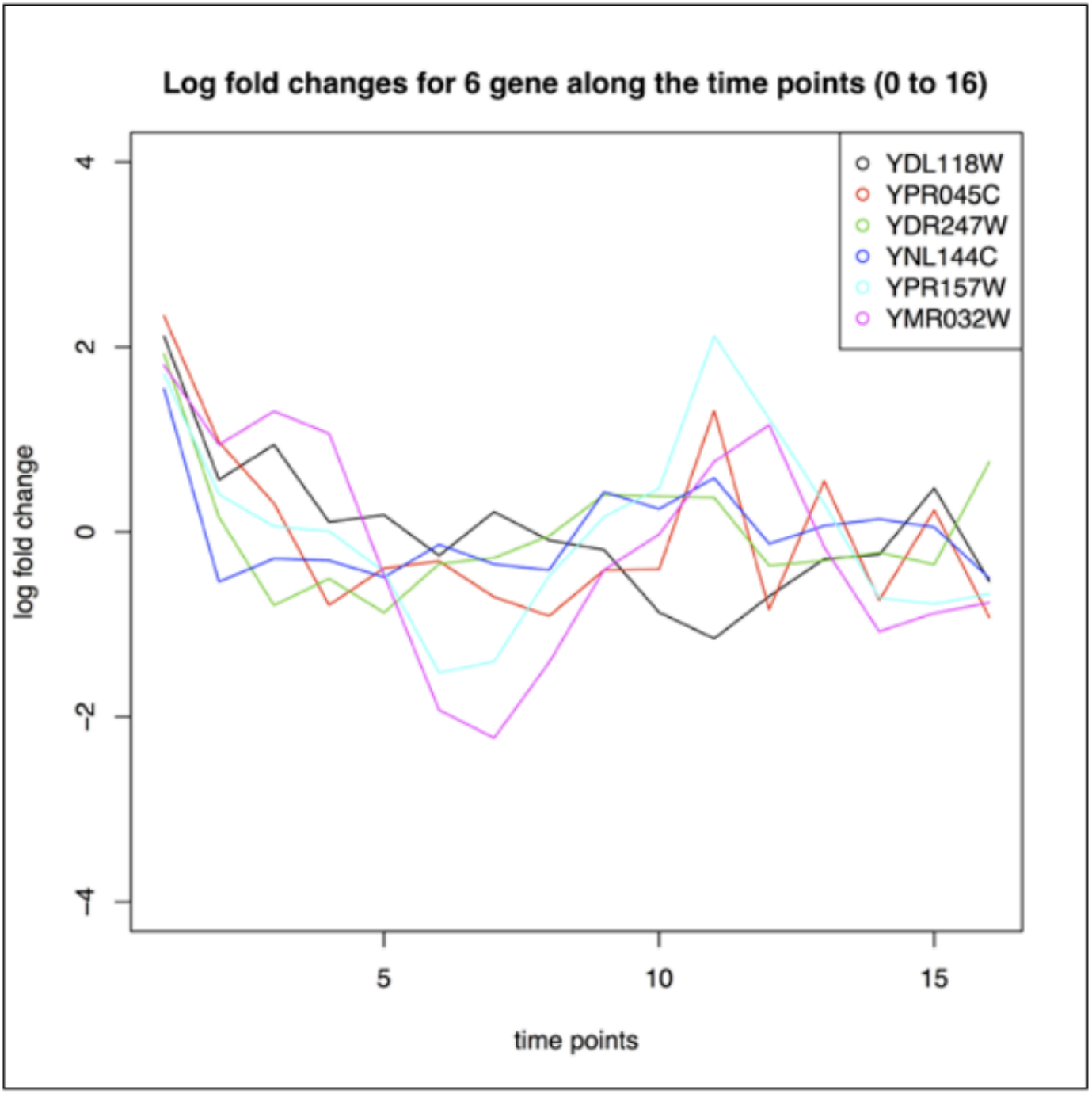
```
> DE_genes[1:10,c(1:4,17,18,19,20)]

            T0...T1    T0...T2    T0...T3    T0...T4       AveExpr        F      P.Value  adj.P.Val
YDR535C  0.045373350 -0.3547546  3.0812557  3.0812557 -0.0005090025 256.5624 2.244303e-05 0.02482905
YMR317W -1.361618986 -0.6120542  0.1716029 -1.2922435 -0.0283052587 193.5176 4.037131e-05 0.02482905
YHR137W  0.085269790  0.9183755  1.1131313  0.9793744  0.0120720297 163.5566 5.728278e-05 0.02482905
YML047C -0.459388638 -0.7911001  0.2786879  1.8677038  0.0045702974 153.9416 6.497142e-05 0.02482905
YMR032W  1.801686846  0.9413805  1.3044926  1.0622749  0.0350770571 146.6830 7.183158e-05 0.02482905
YGL184C -0.001825009 -0.2441705 -1.6977556 -1.9718563 -0.0103381969 130.2669 9.191468e-05 0.02482905
YAL067C -0.006592829  0.1164244 -1.6635528 -2.2965168 -0.0323991276 118.9541 1.109897e-04 0.02482905
YOL023W  0.619830018  1.9858828  1.5984868  0.1094692  0.0110757348 109.1761 1.326050e-04 0.02482905
YHR185C -1.060100057 -0.1223435 -0.6438810  0.4720617 -0.0037853298 108.9022 1.332975e-04 0.02482905
YLR374C -0.134356153 -0.4125907 -0.9739728 -0.3610631  0.0085810480 106.0072 1.409565e-04 0.02482905
```

Riassumendo, l'analisi effettuata è molto simile a quella dei dati statici vista nella sezione precedente. L'unica differenza in questo caso è l'utilizzo del fattore tempo nella creazione delle matrici di progetto e di confronto. Tali dati possono anche essere analizzati usando la sola matrice di progetto in condizioni semplici e con dati completi.

Il grafico nella pagina seguente mostra il log-FC per il dataset *yeast* nei vari tempi per i primi sei geni con le relative oscillazioni.

La libreria *affy* ha anche una funzione `justRMA()` che può eseguire la normalizzazione RMA (Robust Multichip Average) dei dati di espressione. Utilizza solo la normalizzazione quantile e può lavorare direttamente sui file CEL, senza bisogno di AffyObject.



In Bioconductor sono presenti molti altri metodi per filtrare i geni differenzialmente espressi nel tempo, che si basano su approcci statistici simili (Bayes) o differenti. L'analista può utilizzare il miglior metodo possibile a seconda dei dati, dell'esperimento e del quesito biologico.

L'elenco dettagliato dei pacchetti è disponibile sulla pagina web di Bioconductor all'indirizzo: https://bioconductor.org/packages/release/BiocViews.html#___DifferentialExpression.

11.12 Le variazioni di espressione (*fold change*) nei dati da microarray

La variazione di espressione (*fold change*) si riferisce al rapporto tra il valore finale e il valore iniziale di espressione del gene nelle condizioni confrontate. In termini di espressione genica, il *fold change* può essere definito come il rapporto tra la quantificazione finale (ad es. i tempi successivi o le condizioni di trattamento) dell'mRNA e il contenuto iniziale (ad es. il tempo iniziale o i controlli).

Rappresenta il cambiamento relativo piuttosto che una (ambigua) quantità assoluta: quando si estraggono i geni differenzialmente espressi da un dataset, il *fold change* serve come identificatore più riproducibile per l'individuazine dei geni più "interessanti". Questa sezione illustra l'uso del *fold change* per tali scopi.

Continueremo a utilizzare il dataset sulla leucemia visto in precedenza (questa volta prendendo in esame tutti i tipi di leucemia), riprendendo direttamente i risultati delle sezioni 11.9 e 11.10 derivanti dall'applicazione del pacchetto *limma* (la tabella generata con `topTable()` ha una colonna per il *fold change* associato alle sonde). Riproduciamo i comandi necessari a creare la tabella dei geni differenzialmente espressi in base al dataset della leucemia:

```
> library(limma)
> library(leukemiasEset)
> data(leukemiasEset)
> pheno <- pData(leukemiasEset)
> myData <- leukemiasEset[, sampleNames(leukemiasEset)[c(1:3, 13:15, 25:27, 49:51)]]
> design <- model.matrix(~0 + factor(pData(myData)$LeukemiaType))
> colnames(design) <- unique(as.character(pData(myData)$LeukemiaType))
> fit <- lmFit(myData, design)
> contrast.matrix <- makeContrasts(NoL- ALL, NoL- AML, NoL- CLL, levels = design)
> fit2 <- contrasts.fit(fit, contrast.matrix)
> fit2 <- eBayes(fit2)
> genes <- topTable(fit2, adjust="fdr", sort.by="B", number=Inf, coef=1)
> DE_genes <- genes[genes$adj.P.Val < 0.01,] # if we wanto to filter the most DE genes
> head(DE_genes)
                    logFC  AveExpr         t      P.Value    adj.P.Val        B
ENSG00000152078  4.510507 4.856523  28.13988 4.463747e-11 9.004270e-07 14.01472
ENSG00000117519 -4.185175 4.791585 -22.73888 3.878292e-10 3.911645e-06 12.69738
ENSG00000145850  4.142236 4.507655  17.38636 5.759942e-09 2.925048e-05 10.72782
ENSG00000170180  5.681327 5.734169  17.37423 5.800214e-09 2.925048e-05 10.72231
ENSG00000087586  3.952183 5.720789  16.45393 9.977396e-09 3.111188e-05 10.28705
ENSG00000047597  5.362419 5.108415  16.32474 1.079114e-08 3.111188e-05 10.22315
```

Sopra sono rappresentati i primi geni più significativi dopo l'analisi *limma*. Estraiamo le colonne più importanti in un data.frame separato per le prime 10.000 sonde come segue:

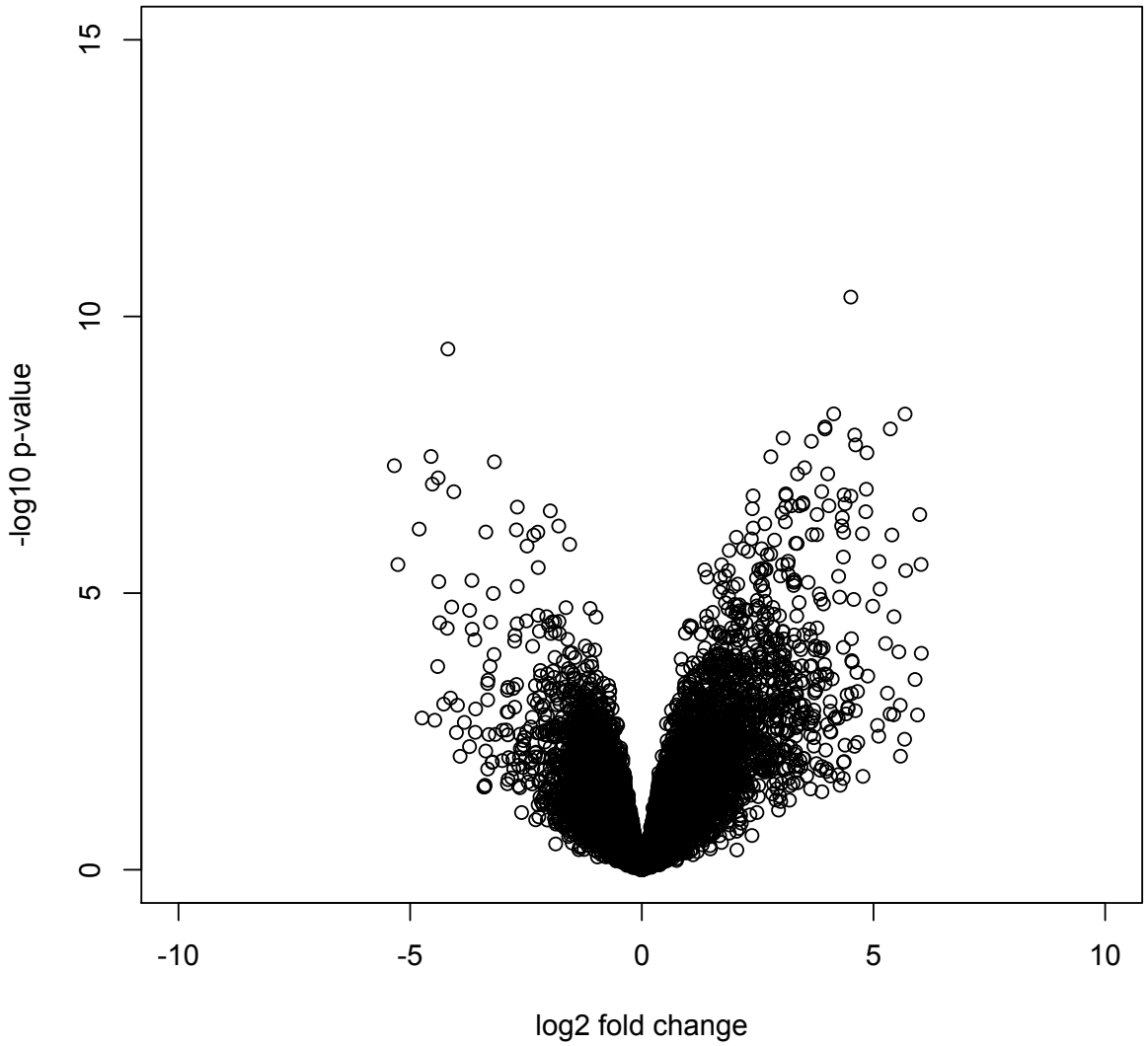
```
> myTable <- topTable(fit2, number=10000)
> logratio <- genes$logFC
```

La libreria *gtools* può essere utilizzata per trasformare il *fold change* in valori log e viceversa:

```
> library(gtools)
> FC <- logratio2foldchange(logratio, base=2) # applicable to the previous logratio variable
> LR <- foldchange2logratio(foldchange, base=2) # not applicable in this case
```

Visualizziamo ora il log *fold change* e i *p*-value nel grafico (*volcano plot*) seguente:

```
> plot(genes$logFC, -log10(genes$P.Value),xlim=c(-10, 10), ylim=c(0, 15),
      xlab="log2 fold change", ylab="-log10 p-value")
```



Possiamo inoltre selezionare i geni significativi dalla tabella utilizzando la colonna del log *fold change* (logFC>1.5) come criterio insieme alla colonna del *p*-value (P.Val<0.05):

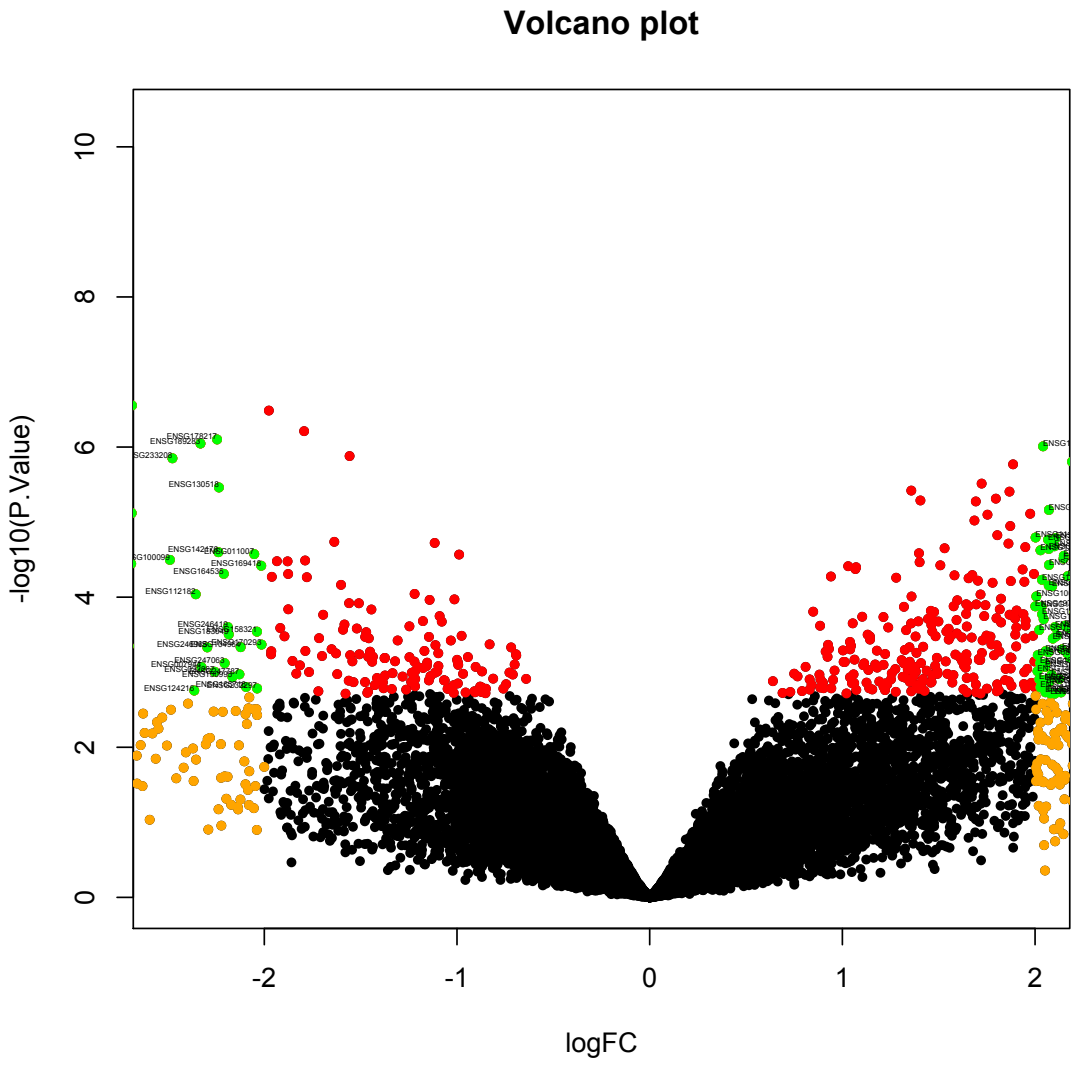
```
> myTable[genes$P.Val<0.05 & genes$logFC>1.5,]
                   NoL...ALL     NoL...AML    NoL...CLL   AveExpr          F      P.Value
ENSG00000152078  4.5105074099  4.8130880711  4.238709078  4.856523 402.014054 5.861217e-11
ENSG00000104043  4.5945506150 -0.6587315157  4.729545071  5.877417 201.108790 1.977755e-09
ENSG00000170180  5.6813272073  5.7321101457  5.979642373  5.734169 157.495813 6.783896e-09
ENSG00000145850  4.1422362637  4.2917293951  4.109112072  4.507655 154.209151 7.542751e-09
ENSG00000111863 -0.1351353287  0.1905309987 -4.187692726  4.764159 153.272247 7.777368e-09
```

Il funzionamento del codice precedente è semplice e autoesplicativo. Il *fold change* è calcolato in base ai valori finali (trattamento) e iniziali (controllo). Sono utilizzati i logaritmi in base 2. Il grafico precedente è un semplice scatterplot (chiamato *volcano plot*) con il $\log_2$ del *fold change* lungo l'asse *x* e il $-\log_{10}$(*p*-value) lungo l'asse *y*. Trasformando i *p*-value in scala logaritmica si ottiene una migliore risoluzione per la visualizzazione nel grafico.

Un *volcano plot* tipicamente traccia la misura dell'effetto sull'asse *x* (nel nostro caso il $\log_2$ del *fold change*) e la significatività statistica sull'asse *y* (nel nostro caso il $-\log_{10}$ del *p*-value). I geni che sono altamente disregolati sono più lontani a sinistra (*down*) e a destra (*up*), mentre i *fold change* statisticamente significativi appaiono nella parte alta del grafico.

Possiamo migliorare il grafico evidenziando i geni up/down regolati e statisticamente significativi con i seguenti comandi (facciamo uso del pacchetto *calibrate* per etichettare i geni più importanti):

```
> install.packages("calibrate")
> library(calibrate)
> Gene <- rownames(genes)
> Gene <- paste(substr(Gene,1,4),substr(Gene,10,15),sep="")
> gene_table <- cbind(Gene,genes)
> with(gene_table, plot(logFC, -log10(P.Value), pch=20, main="Volcano plot",
      xlim=c(-2.5,2)))
> red_points <- subset(gene_table, adj.P.Val<0.05)
> with(red_points, points(logFC, -log10(P.Value), pch=20, col="red"))
> orange_points <- subset(gene_table, abs(logFC)>2)
> with(orange_points, points(logFC, - log10(P.Value), pch=20, col="orange"))
> green_points <- subset(gene_table, adj.P.Val<0.05 & abs(logFC)>2)
> with(green_points, points(logFC, -log10(P.Value), pch=20, col="green"))
> with(green_points, textxy(logFC, -log10(P.Value), labs=Gene, cex=0.3, offset=0.5))
```



Dopo aver copiato i dati dei geni nella tabella di comodo *gene_table* (alla quale abbiamo aggiunto la colonna *Gene* dei nomi abbreviati dei geni), abbiamo ricreato il grafico *volcano plot* di base e colorato i geni in base al loro *fold change* (in arancione quelli maggiormente up/down regolati, con `abs(logFC)>2`) e alla loro significatività statistica (in rosso, con `adj.P.Val<0.05`). Il pacchetto *calibrate* ci consente — nell'ultima riga di comando — di etichettare i geni di interesse, che sono sia up/down regolati ($\log_2$(*fold change*)>2, cioè con un *fold change* di almeno 4 volte) che statisticamente significativi (*p*-value<0.05), il cui numero (430) ed elenco possiamo esaminare come segue:

```
> length(green_points$Gene)
[1] 430
> green_points$Gene
  [1] ENSG152078 ENSG117519 ENSG145850 ENSG170180 ENSG087586 ENSG047597 ENSG175449
  [8] ENSG104043 ENSG024526 ENSG115641 ENSG188672 ENSG133063 ENSG164330 ENSG173372
 [15] ENSG234955 ENSG102935 ENSG159189 ENSG107562 ENSG086506 ENSG130635 ENSG066923
 [22] ENSG162692 ENSG198336 ENSG109501 ENSG168497 ENSG080819 ENSG130821 ENSG162599
 [29] ENSG173391 ENSG071967 ENSG197721 ENSG196188 ENSG173369 ENSG144407 ENSG171840
 [36] ENSG134369 ENSG164398 ENSG006074 ENSG090269 ENSG109255 ENSG164879 ENSG166947
 [43] ENSG112077 ENSG164010 ENSG130208 ENSG112212 ENSG101935 ENSG177455 ENSG183166
 [50] ENSG221968 ENSG178217 ENSG163710 ENSG133048 ENSG143942 ENSG240583 ENSG008394
. . .
[400] ENSG127951 ENSG196415 ENSG165716 ENSG149516 ENSG150867 ENSG163737 ENSG120696
[407] ENSG164506 ENSG232297 ENSG214882 ENSG122694 ENSG138678 ENSG120265 ENSG168461
[414] ENSG124216 ENSG134242 ENSG099882 ENSG136824 ENSG198736 ENSG178695 ENSG164111
[421] ENSG138160 ENSG168496 ENSG163751 ENSG133193 ENSG153140 ENSG110077 ENSG165071
[428] ENSG174099 ENSG121104 ENSG084234
```

11.13 L'arricchimento funzionale dei dati con i termini GO

Una volta che conosciamo i geni differenzialmente espressi dai dati dell'array, abbiamo tutti i geni che in qualche modo hanno un ruolo significativo nella cellula. Per conoscere meglio questo insieme di geni a livello biologico, dobbiamo conoscerne il ruolo in termini di funzione svolta mediante l'analisi delle categorie di GO (Gene Ontology), una raccolta di circa 25.000 termini di riferimento che descrivono la biologia di un prodotto genico in qualsiasi organismo. La raccolta GO è organizzata gerachicamente in 3 categorie indipendenti: MF=Molecular Function, BP=Biological Process, CC=Cellular Component. Un gene può essere presente in una qualsiasi delle ontologie (MF / BP / CC) ed avere assegnati più termini GO; la struttura gerarchica implica che se un gene è membro di un termine è anche membro dei termini "genitori" (visitare il sito http://www.geneontology.org per maggiori dettagli).

Questa sezione riguarda l'arricchimento di un set di geni con i termini GO verificato attraverso un test statistico. Avendo in input un elenco di geni (ad es. up/down regolati) si vuole verificare se i termini GO associati sono sovrarappresentati nel set di geni dato rispetto a un set di geni campionato casualmente. Il metodo statistico seguito è il *test ipergeometrico*, basato sulla distribuzione ipergeometrica che deriva dal campionamento da una popolazione fissata.

La successiva elaborazione utilizza i dati del set di geni che provengono dagli esempi precedenti e il pacchetto di annotazioni indicato nell'oggetto "ExpressionSet" *hgu95av2.db*. Per prima cosa, installiamo e carichiamo il database delle annotazioni e la libreria *GOstats* come segue:

```
> BiocManager::install(c("hgu95av2.db", "GOstats"))
> library(hgu95av2.db)
> library(GOstats)
> library(biomaRt)
```

Prendiamo i dati di input dai risultati dell'analisi dei dati sulla leucemia (sezione 11.12) e creiamo due insiemi di geni; uno che consiste di tutti i geni nel dataset e l'altro che consiste di geni differenzialmente espressi:

```
> all_genes <- rownames(genes)
> sel_genes <- rownames(DE_genes)
```

Mappiamo questi set di geni sui loro ID Entrez come segue:

```
> mart <- useDataset("hsapiens_gene_ensembl", useMart("ensembl")) # set the mart Homo sapiens
> all_genes <- c(getBM(filters="ensembl_gene_id", attributes=c("entrezgene_id"),
      values=all_genes, mart=mart)) # get entrez IDs for all genes
> sel_genes <- c(getBM(filters="ensembl_gene_id", attributes=c("entrezgene_id"),
      values=sel_genes, mart=mart)) # get entrez IDs for DE genes
```

Definiamo una soglia di significatività per il successivo test statistico:

```
> hgCutoff <- 0.05
```

Abbiamo ora bisogno di un oggetto di tipo "GOHyperGParams" (per i dettagli consultare la documentazione con `help.search("GOHyperGParams")`), che verrà utilizzato come parametro di input per l'elaborazione del *GO enrichment* (arricchimento Gene Ontology) di tipo BP. Può essere creato mediante la seguente funzione:

```
> params <- new("GOHyperGParams", geneIds=unlist(sel_genes),
      universeGeneIds=unlist(all_genes), annotation="hgu95av2.db", ontology="BP",
      pvalueCutoff=hgCutoff, conditional=FALSE, testDirection="over")
```

Una volta pronto l'oggetto *params* di tipo "GOHyperGParams", eseguiamo il test ipergeometrico per ottenere il *p*-value per le annotazioni GO come segue:

```
> hgOver <- hyperGTest(params)
> class(hgOver)
[1] "GOHyperGResult"
attr(,"package")
[1] "GOstats"
```

Osserviamo il numero di geni associati alle diverse categorie:

```
> geneCounts(hgOver)
GO:0046501 GO:0006779 GO:0006778 GO:0033014 GO:0042168 GO:0006782 GO:0006783 GO:0033013 GO:0019755 GO:0042440 GO:0048821 GO:0051188
         6          7          8          7          7          5          6          8          4          7          5         12
GO:0046148 GO:0030218 GO:0055065 GO:1905446 GO:0055080 GO:0002262 GO:0042592 GO:0098771 GO:0034101 GO:0050801 GO:0007052 GO:0055072
         6          8         19          3         20          9         36         20          8         21          7          6
. . .
GO:0001817 GO:0006396 GO:0080135 GO:0007268 GO:0098916 GO:0099537 GO:0099536 GO:0034655 GO:1905114 GO:0016071 GO:0007267 GO:0008150
         2          2          2          2          2          2          2          1          1          1          6        125
```

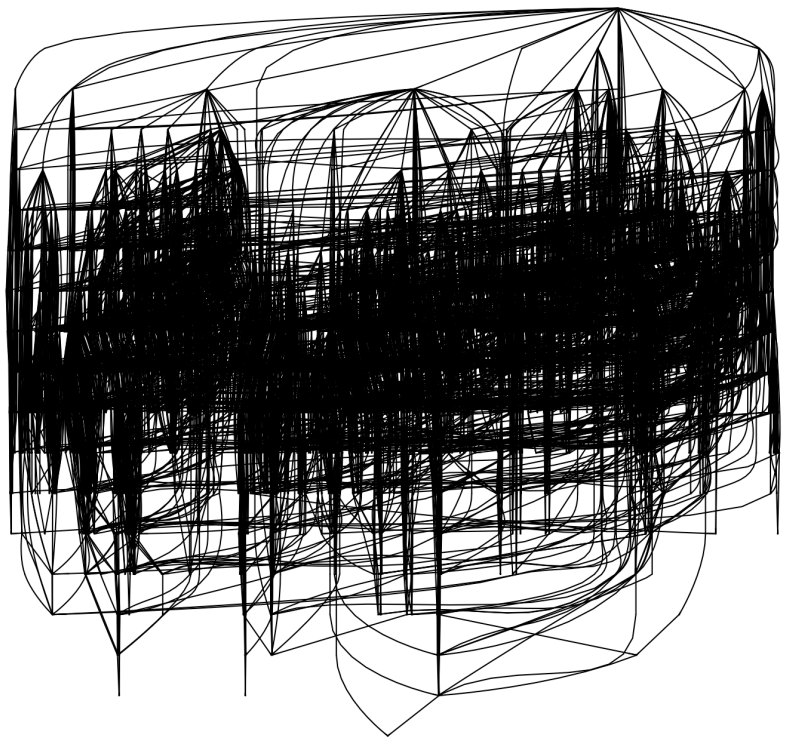
```
> universeCounts(hgOver)
GO:0046501 GO:0006779 GO:0006778 GO:0033014 GO:0042168 GO:0006782 GO:0006783 GO:0033013 GO:0019755 GO:0042440 GO:0048821 GO:0051188 GO:0046148 GO:0030218
        10         17         26         18         21          8         15         38          9         49         24        191         42         86
GO:0055065 GO:1905446 GO:0055080 GO:0002262 GO:0042592 GO:0098771 GO:0034101 GO:0050801 GO:0007052 GO:0055072 GO:0032536 GO:0035378 GO:0048878 GO:0055076
       449          6        491        116       1221        499         93        543         72         51          8          2        758         81
. . .
GO:1901361 GO:0001817 GO:0006396 GO:0080135 GO:0007268 GO:0098916 GO:0099537 GO:0099536 GO:0034655 GO:1905114 GO:0016071 GO:0007267 GO:0008150
       456        459        464        478        486        486        491        495        396        409        489       1106       7919
```

Controlliamo il riepilogo dell'oggetto creato *hgOver* per una panoramica complessiva del test:

```
> summary(hgOver)
         GOBPID       Pvalue  OddsRatio    ExpCount Count Size
1   GO:0046501 2.735588e-09  98.193277  0.15784821     6   10
2   GO:0006779 3.519029e-09  46.176271  0.26834196     7   17
3   GO:0006778 3.788979e-09  29.538462  0.41040535     8   26
4   GO:0033014 5.683340e-09  41.973035  0.28412678     7   18
5   GO:0042168 1.996459e-08  32.966102  0.33148125     7   21
6   GO:0006782 4.877567e-08 108.208333  0.12627857     5    8
…
443 GO:0019637 4.960904e-02   1.630979 11.09672939    17  703
                                                                  Term
1                                   protoporphyrinogen IX metabolic process
2                                porphyrin-containing compound biosynthetic process
3                                porphyrin-containing compound metabolic process
4                                          tetrapyrrole biosynthetic process
5                                                  heme metabolic process
6                                protoporphyrinogen IX biosynthetic process
…
443                                          organophosphate metabolic process
```

Tracciamo anche il grafico aciclico diretto (DAG, Direct Acyclic Graph) del GO come segue:

```
plot(goDag(hgOver))
```



Infine, generiamo il report come file HTML che può essere letto con qualsiasi browser e illustra i termini relativi alla circolazione sanguigna nei dati sulla leucemia:

```
> htmlReport(hgOver, file="ALL_hgco.html")
```

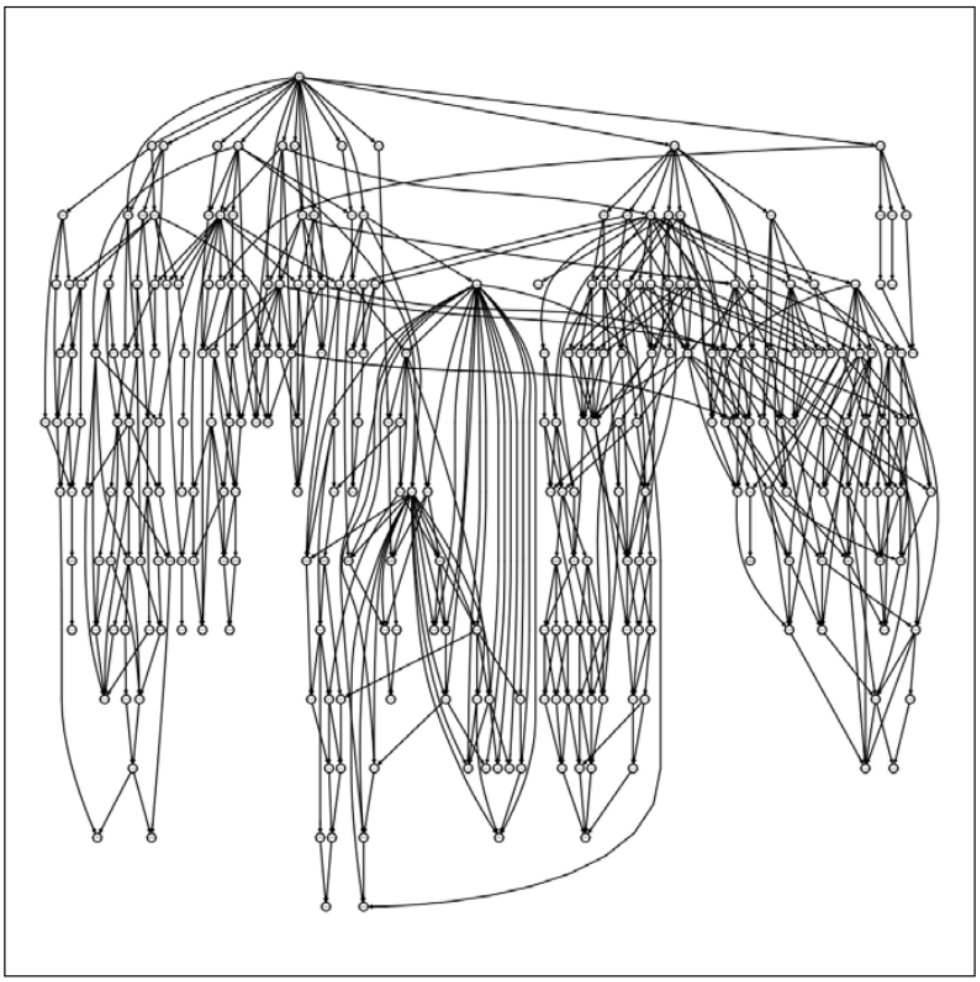
Gene to GO BP test for over-representation

GOBPID	Pvalue	OddsRatio	ExpCount	Count	Size	Term
GO:0046501	0.000	98.193	0	6	10	protoporphyrinogen IX metabolic process
GO:0006779	0.000	46.176	0	7	17	porphyrin-containing compound biosynthetic process
GO:0006778	0.000	29.538	0	8	26	porphyrin-containing compound metabolic process
GO:0033014	0.000	41.973	0	7	18	tetrapyrrole biosynthetic process
GO:0042168	0.000	32.966	0	7	21	heme metabolic process
GO:0006782	0.000	108.208	0	5	8	protoporphyrinogen IX biosynthetic process
GO:0006783	0.000	43.613	0	6	15	heme biosynthetic process
GO:0033013	0.000	17.696	1	8	38	tetrapyrrole metabolic process
GO:0019755	0.000	51.498	0	4	9	one-carbon compound transport
GO:0042440	0.000	10.949	1	7	49	pigment metabolic process
GO:0048821	0.000	17.050	0	5	24	erythrocyte development
GO:0051188	0.000	4.518	3	12	191	cofactor biosynthetic process
GO:0046148	0.000	10.866	1	6	42	pigment biosynthetic process
GO:0030218	0.000	6.764	1	8	86	erythrocyte differentiation
GO:0055065	0.000	3.070	7	19	449	metal ion homeostasis
GO:1905446	0.000	63.861	0	3	6	regulation of mitochondrial ATP synthesis coupled electron transport
GO:0055080	0.000	2.961	8	20	491	cation homeostasis
GO:0002262	0.000	5.574	2	9	116	myeloid cell homeostasis
GO:0042592	0.000	2.256	19	36	1221	homeostatic process

L'oggetto GOHyperGParams racchiude i parametri del test, rendendo più facile organizzare ed eseguire il test ipergeometrico sulle annotazioni GO per il set di geni. L'oggetto ha degli *slot* (entrate) per la categoria GO (BP, MF o CC), i geni (ID di Entrez) e la condizione e l'annotazione della struttura GO. La funzione `hyperGTest()` implementa il test ipergeometrico utilizzando l'insieme dei parametri dell'oggetto GOHyperGParams. Essa calcola la sovrarappresentazione o la sottorappresentazione (argomento `testDirection`) dei termini GO nel set di geni (tuttavia, il calcolo ignora la struttura GO, trattando ogni annotazione come indipendente). Se l'argomento `conditional` è impostato a `TRUE`, la funzione utilizza la struttura del grafico GO per stimare, per ogni termine, se sia statisticamente sovrarappresentato in base ai suoi discendenti. La tabella riportata sopra è per le annotazioni più importanti dei termini GO. La prima colonna riporta gli ID GO, la seconda il *p*-value prodotto dal test ipergeometrico, la terza il rapporto di probabilità (*Odds Ratio*) e il resto riguarda rispettivamente il conteggio previsto, il conteggio effettivo, la dimensione e il termine effettivo.

KEGG è un'altra fonte di informazioni sui pathway e sulle funzioni, ma non è disponibile in R nella sua forma aggiornata. Si tratta di una banca dati completa di vari pathway (per esempio di trasduzione del segnale, metabolici e così via). Per visualizzare il database, visitare l'indirizzo https://www.genome.jp/kegg/. Possiamo usare l'oggetto "KEGGHyperGParams" al posto di "GOHyperGParams" per l'arricchimento dei geni KEGG. Durante la creazione dell'oggetto, tutti i parametri GOHyperGParams vengono sostituiti con KEGGHyperGParams e l'argomento `conditional` va impostato su FALSE. L'analisi necessita anche del pacchetto *KEGG.db* (non più aggiornato dal 2014; per informazioni visitare il sito http://www.bioconductor.org/packages/2.13/data/annotation/manuals/KEGG.db/man/KEGG.db.pdf).

Si può utilizzare anche il pacchetto *MLP* per l'arricchimento dei geni con vari database di pathway. Per i dettagli, consultare la home page di Bioconductor all'indirizzo http://bioconductor.org/packages/release/bioc/html/MLP.html. Il seguente grafico mostra parte dell'albero precedente delle categorie GO:



11.14 Clustering di dati da microarray

Il clustering consiste nell'aggregazione di geni (o campioni) simili in un gruppo (chiamato *cluster*) e lontano da altri gruppi simili. Quando i geni vengono raggruppati insieme significa che seguono un modello simile basato sui dati di espressione nelle condizioni date. Questa sezione presenta il concetto ampiamente utilizzato del *clustering gerarchico* nell'analisi dell'espressione genica, utilizzando una parte (per velocità di elaborazione) dei dati sulla leucemia già visti nelle sezioni precedenti.

Per prima cosa, creiamo il dataset per il clustering a partire dai dati sulla leucemia, utilizzando solo le prime 100 istanze a scopo dimostrativo:

```
> library(leukemiasEset)
> data(leukemiasEset)
> c.data <- exprs(leukemiasEset[1:100,])
```

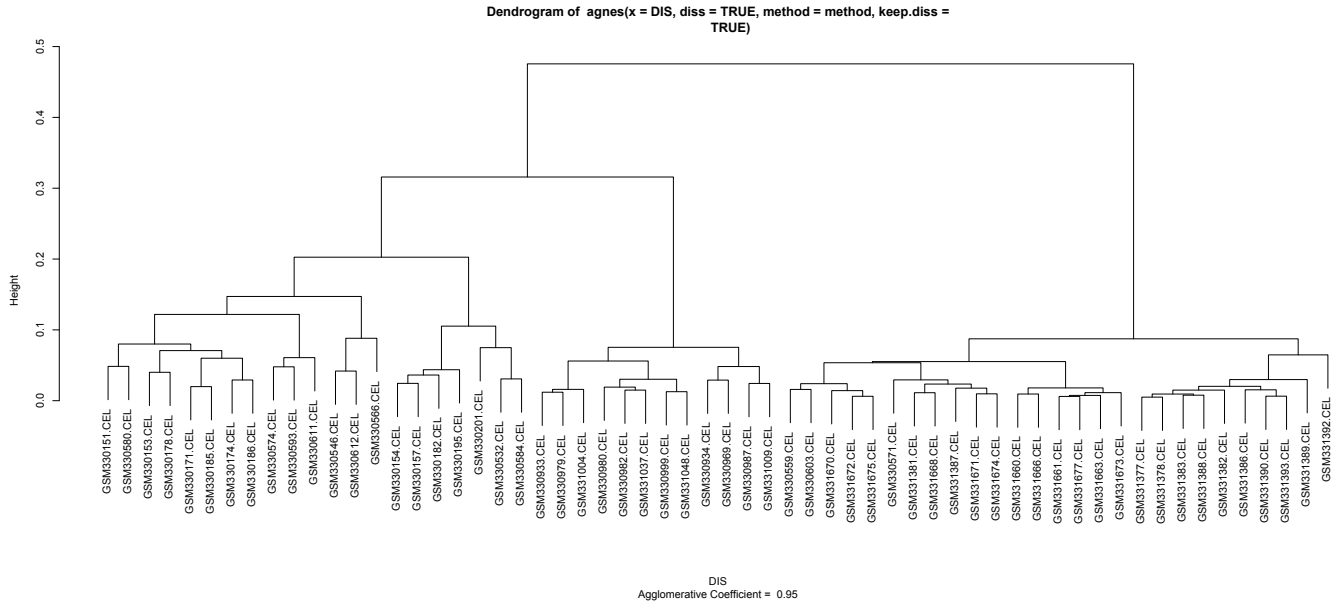
Eseguiamo il clustering dei campioni (file CEL) presenti nella matrice *c.data* utilizzando il pacchetto *EMA* (Easy Microarray Analysis); il risultato è memorizzato nella variabile *c.array*, un oggetto di tipo "agnes" (Agglomerative Nesting):

```
> install.packages("EMA")
> library(EMA)
> c.array <- clustering(data=c.data, metric="pearson", method="ward")
> c.array
Call:  agnes(x = DIS, diss = TRUE, method = method, keep.diss = TRUE)
Agglomerative coefficient:  0.9491618
Order of objects:
 [1] GSM330151.CEL GSM330580.CEL GSM330153.CEL GSM330178.CEL GSM330171.CEL GSM330185.CEL
 [7] GSM330174.CEL GSM330186.CEL GSM330574.CEL GSM330593.CEL GSM330611.CEL GSM330546.CEL
[13] GSM330612.CEL GSM330566.CEL GSM330154.CEL GSM330157.CEL GSM330182.CEL GSM330195.CEL
[19] GSM330201.CEL GSM330532.CEL GSM330584.CEL GSM330933.CEL GSM330979.CEL GSM331004.CEL
[25] GSM330980.CEL GSM330982.CEL GSM331037.CEL GSM330999.CEL GSM331048.CEL GSM330934.CEL
[31] GSM330969.CEL GSM330987.CEL GSM331009.CEL GSM330559.CEL GSM330603.CEL GSM331670.CEL
[37] GSM331672.CEL GSM331675.CEL GSM330571.CEL GSM331381.CEL GSM331668.CEL GSM331387.CEL
[43] GSM331671.CEL GSM331674.CEL GSM331660.CEL GSM331666.CEL GSM331661.CEL GSM331677.CEL
[49] GSM331663.CEL GSM331673.CEL GSM331377.CEL GSM331378.CEL GSM331383.CEL GSM331388.CEL
[55] GSM331382.CEL GSM331386.CEL GSM331390.CEL GSM331393.CEL GSM331389.CEL GSM331392.CEL
Height (summary):
    Min.  1st Qu.   Median     Mean  3rd Qu.     Max.
0.005031 0.014946 0.029234 0.052161 0.057994 0.475575

Available components:
[1] "order"     "height"    "ac"        "merge"     "diss"      "call"      "method"
[8] "order.lab"
```

Tracciamo il grafico del dendrogramma per il clustering dei campioni con il comando seguente:

```
> plot(c.array)
```

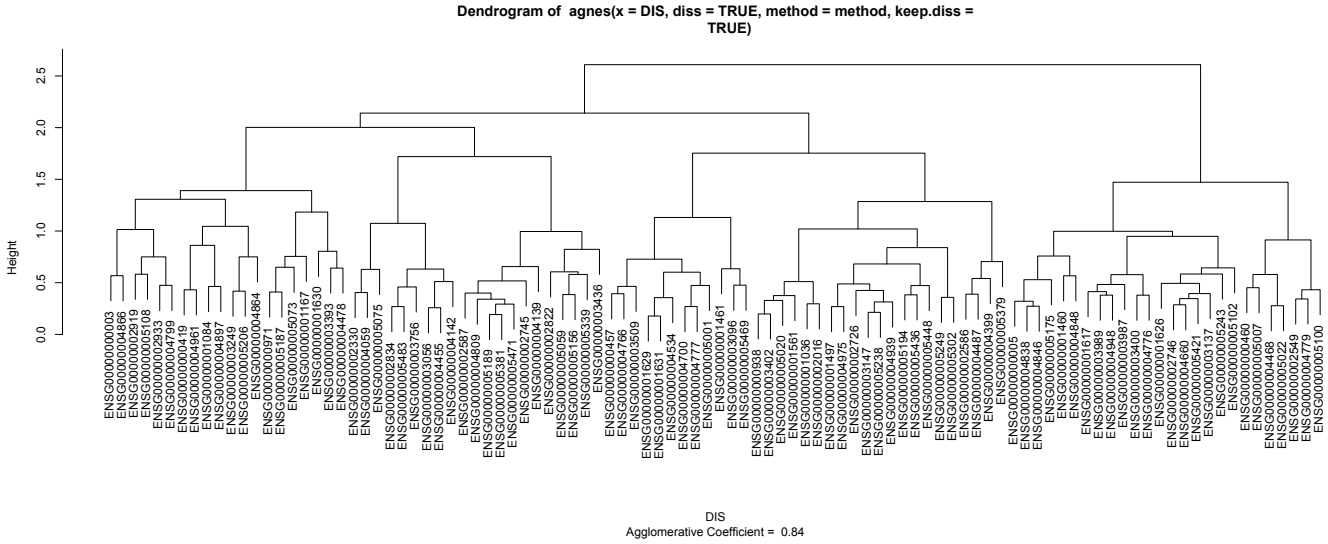


Per raggruppare i geni, è sufficiente trasporre la matrice dei dati e utilizzarla come argomento di input nella funzione di raggruppamento, definendo inoltre la metrica di similarità (argomento `metric`) e il metodo di raggruppamento (argomento `method`) come segue:

```
> c.gene <- clustering(data=t(c.data), metric="pearsonabs", method="ward")
```

Tracciamo il grafico risultante come segue (si ricordi che, per problemi di leggibilità, avevamo selezionato solo 100 geni):

```
> plot(c.gene)
```



Una visualizzazione più dettagliata sotto forma di *heatmap* è presentata nella sezione 11.16 del capitolo.

Il metodo di clustering presentato in questa sezione calcola la similarità tra i dati (campioni o geni) dai valori di espressione. Le misure di similarità che possono essere utilizzate sono il coefficiente di correlazione di Pearson o di Spearman, la distanza euclidea, di Manhattan e di Jaccard. Sulla base dei punteggi di similarità, viene generata una matrice di distanza utilizzata per generare i cluster. Sono stati implementati vari metodi di aggregazione dei cluster, che includono i metodi Average, Single, Complete o Ward. Si possono anche utilizzare altri metodi come k-Means o PAM (Partition Around Medoids).

11.15 Network di co-espressione di geni

La generazione di network che rappresentano le relazioni tra i geni a partire dai dati di espressione riveste un'importanza sempre maggiore nell'analisi di dati da microarray. Vi sono molti modi possibili per evidenziare queste relazioni; in questa sezione esploreremo le relazioni basate sulla *correlazione* tra i geni facendo uso del pacchetto R *WGCNA*. Selezioneremo una piccola frazione del set di dati per rendere il processo più veloce e computazionalmente meno dispendioso. Inoltre, è buona pratica ridurre tali network ai geni più importanti, rendendo così la rete meno "rumorosa".

Per iniziare, installiamo e attiviamo le seguenti librerie nella nostra sessione R:

```
> BiocManager::install("impute")
> BiocManager::install("RBGL")
> BiocManager::install("Rgraphviz")
> install.packages("WGCNA")
> library(RBGL)
> library(WGCNA)
> library(Rgraphviz)
```

Prendiamo solo 25 geni significativi (cfr. variabile *DE_genes* sezione 11.13) dal dataset in questa fase, perché in tal modo si riduce il rumore nei dati e contemporaneamente si velocizza l'elaborazione:

```
> myData.Sel <- exprs(leukemiasEset[rownames(DE_genes)[1:25],1:3])
```

La matrice ha i nomi dei campioni sulle colonne e i geni sulle righe. Trasponiamo i dati come segue:

```
> myData.Sel <- t(myData.Sel)
```

Per calcolare la matrice di adiacenza, possiamo calcolare le correlazioni (ma richiede molto tempo) o utilizziamo la funzione di adiacenza della libreria *WGCNA*:

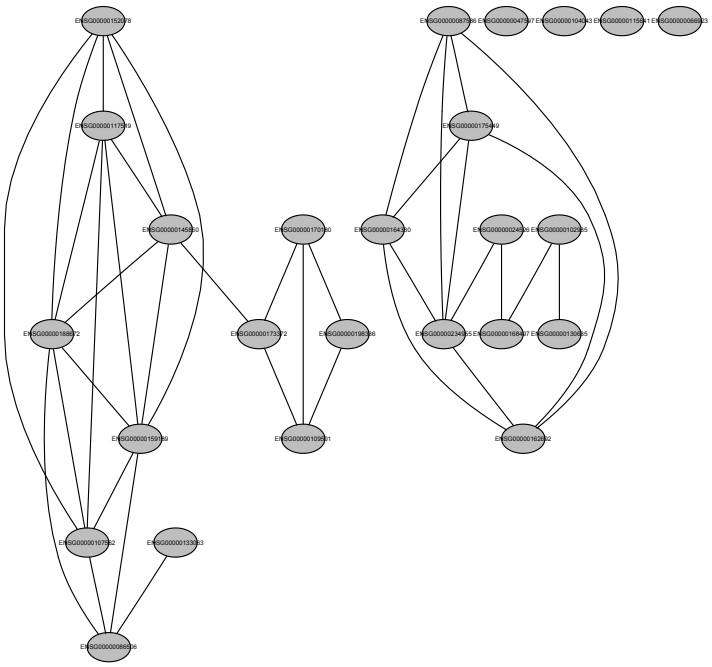
```
> myMat <- adjacency(myData.Sel, type="signed")
```

Il risultato è una matrice quadrata di dimensioni pari al numero di geni. Gli elementi della matrice corrispondono alle connessioni tra i geni corrispondenti; possono essere impostati a 0 (connessione assente) o 1 (connessione presente) tramite dicotomizzazione in diversi modi. Un metodo semplice, ma non ottimale, consiste nell'utilizzare una soglia (ad es. di 0,90) per connettere solo i vertici maggiormente correlati:

```
> adjMat <- myMat
> adjMat[abs(adjMat)>0.90] <- 1
> adjMat[abs(adjMat)<=0.90] <- 0
> diag(adjMat) <- 0
```

La matrice di adiacenza finale può essere convertita in un oggetto graphNEL da visualizzare come grafo con nodi e connessioni:

```
> myGraph <- as(adjMat, "graphNEL")
> myGraph
A graphNEL graph with undirected edges
Number of Nodes = 25
Number of Edges = 38
> plot(myGraph, nodeAttrs=makeNodeAttrs(myGraph,fontsize=28, fillcolor="grey"))
```



La figura precedente mostra la rete di co-espressione genica tra i primi 25 geni ottenuti dopo un'analisi DE (differential expression) delle cellule normali nella leucemia.

Il metodo ilustrato in questa sezione si basa sul calcolo delle relazioni tra i geni in termini di correlazione o misure di similarità. La funzione di adiacenza calcola le similarità o correlazioni tra tutte le coppie di geni in base ai dati di espressione e le restituisce sotto forma di matrice. La soglia impostata definisce solo i geni altamente correlati o simili, i cui nodi sono poi connessi nella rete. La matrice di adiacenza viene poi utilizzata per ottenere la rappresentazione grafica della rete di co-espressione dei geni.

Oltre al pacchetto *WGCNA* (che semplifica il processo), è comunque possibile calcolare autonomamente le distanze o le correlazioni per tali reti, che possono essere dedotte a seconda dell'esperimento e del tipo di dati. Molti di questi metodi sono supportati in R tramite specifici pacchetti, che vanno oltre i nostri scopi. Per informazioni dettagliate si può fare riferimento all'articolo dal titolo *Inferring cellular networks - a review* di Markowetz e Spang (http://www.biomedcentral.com/1471-2105/8/S6/S5/) che presenta una panoramica su alcuni di questi metodi.

11.16 Visualizzazione dei dati di espressione genica

Questa sezione presenta alcune interessanti e utili visualizzazioni per i dati di espressione genica: *heatmap*, diagrammi di Venn e *volcano plot*, per i quali useremo i dataset già visti nel capitolo o creeremo dati artificiali.

Iniziamo con le ***heatmap*** (mappe termiche). Dalla sezione 11.14 sul clustering dei dati da microarray utilizziamo i cluster generati, cioè le variabili *c.array* e *c.gene*. Per creare una *heatmap* utilizziamo la funzione `clustering.plot()` della libreria *EMA*, passando come argomento i cluster precedentemente menzionati (la mappa è visualizzata all'inizio della pagina seguente):

```
> library(EMA)
> clustering.plot(tree=c.array, tree.sup=c.gene, data=c.data)
```

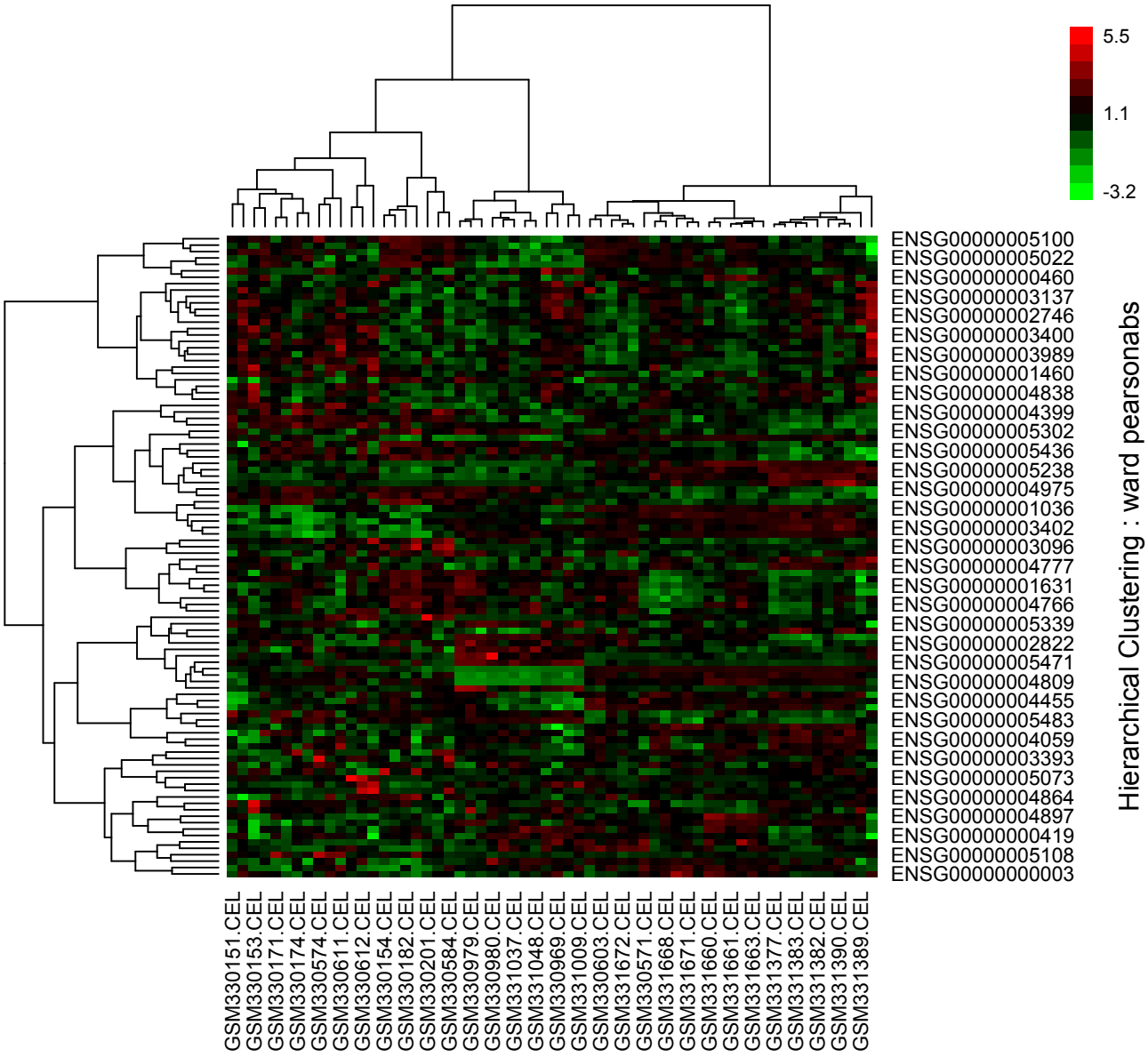
Questo genera la mappa desiderata. Si può anche utilizzare la funzione di base `heatmap()` per ottenere lo stesso risultato (visualizzato a pag. 190):

```
> heatmap(c.data)
```

Entrambe le figure mostrano una *heatmap* per i geni selezionati dai dati di espressione della leucemia, che presenta le cellule normali e i diversi tipi di leucemia, con i campioni sulle colonne e i geni sulle righe.

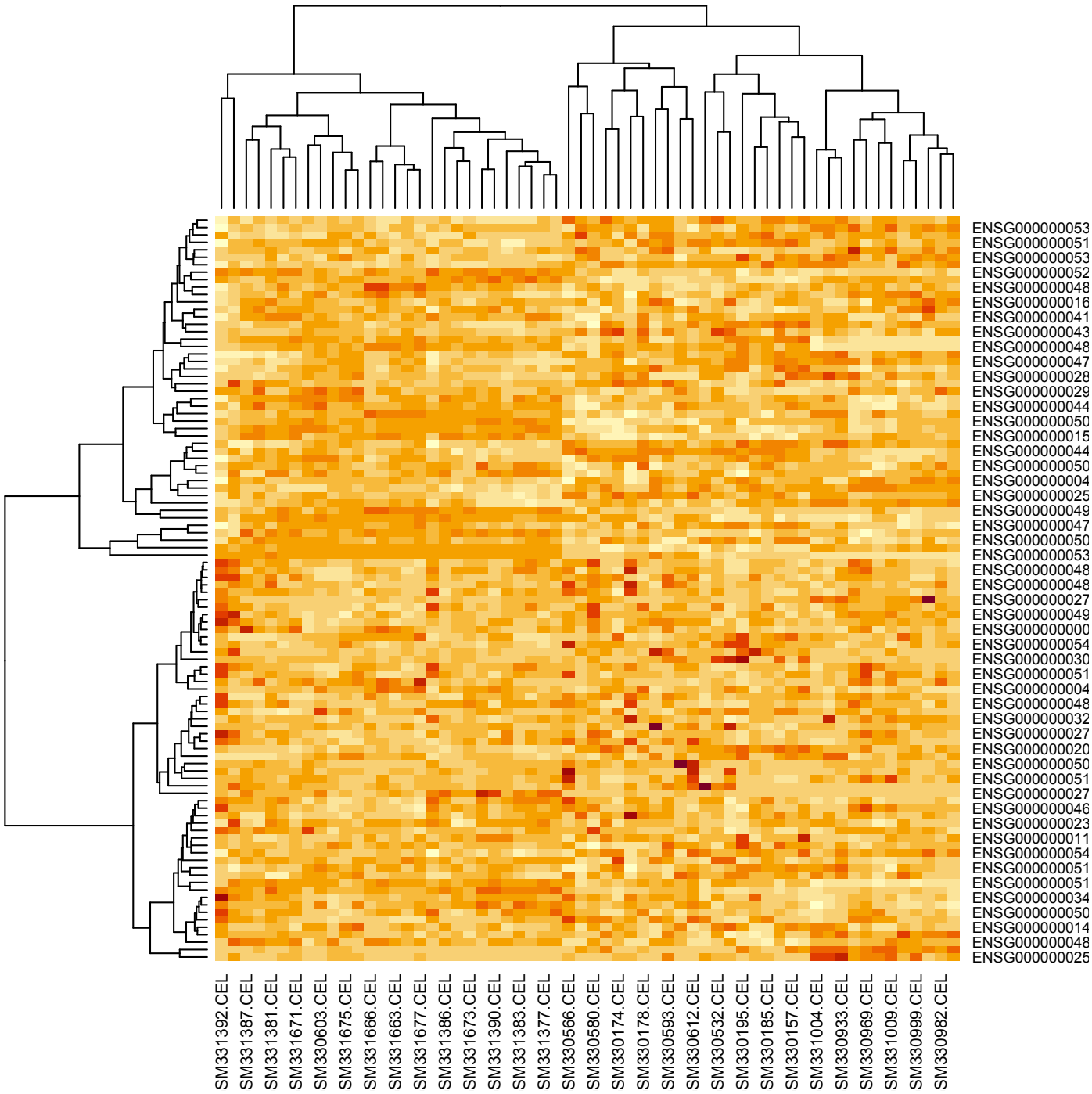
Il successivo tipo di grafico da esaminare è il ***diagramma di Venn***, che può essere usato per mostrare il contenuto comune e unico di due variabili. Richiede il pacchetto *VennDiagram*, che può essere installato e caricato come segue:

```
> install.packages("VennDiagram")
> library(VennDiagram)
```



Creiamo ora dei dati artificiali che consistono in una lista denominata di cinque insiemi chiamata *set*, digitando i seguenti comandi:

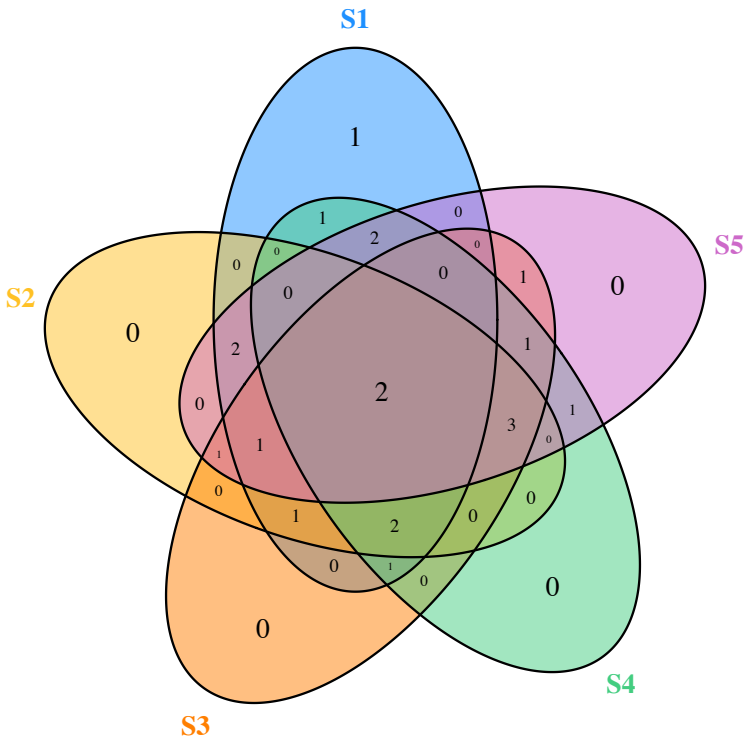
```
> set <- list()
> for (i in 1:5) {set[[i]] <- sample(LETTERS[1:20],replace=TRUE,prob=rep(0.05,20))}
> names(set) <- c(paste("S", 1:5, sep=""))
> set
$S1
 [1] "N" "S" "P" "L" "F" "C" "Q" "B" "R" "T" "C" "C" "R" "R" "S" "D" "C" "R" "A" "K"
$S2
 [1] "J" "D" "O" "S" "C" "I" "T" "T" "C" "I" "K" "S" "R" "M" "I" "B" "L" "M" "O" "S"
$S3
 [1] "M" "I" "M" "H" "J" "R" "J" "J" "O" "H" "B" "D" "C" "L" "D" "T" "A" "D" "G" "H"
$S4
 [1] "I" "F" "T" "A" "A" "I" "E" "D" "B" "Q" "Q" "P" "L" "B" "G" "P" "J" "D" "G" "M"
$S5
 [1] "I" "J" "J" "D" "P" "M" "I" "M" "T" "F" "O" "O" "S" "F" "K" "G" "J" "R" "H" "E"
```



Per tracciare il diagramma di Venn, utilizziamo i cinque insiemi della lista *set* creata in precedenza e creiamo l'oggetto *venn.plot* di tipo *gList* come segue:

```
> venn.plot <- venn.diagram(x = set, filename = NULL, cat.cex = 1.5, alpha = 0.50,
      col = "black", fill = c("dodgerblue", "goldenrod1", "darkorange1", "seagreen3",
      "orchid3"), cex = c(1.5, 1.5, 1.5, 1.5, 1.5, 1, 0.8, 1, 0.8, 1, 0.8, 1, 0.8, 1,
      0.8, 1, 0.55, 1, 0.55, 1, 0.55, 1, 0.55, 1, 0.55, 1, 1, 1, 1, 1, 1.5),
      cat.col = c("dodgerblue", "goldenrod1", "darkorange1", "seagreen3",
      "orchid3"), cat.fontface = "bold", margin = 0.05, category =
      c("S1","S2","S3","S4","S5"))
> pdf("venn.pdf") #starts the graphics device driver for producing PDF graphics on the external file "venn.pdf"
> grid.draw(venn.plot) #produces graphical output from object venn.plot.
> dev.off() #closes the PDf file
```

La figura seguente mostra il diagramma di Venn per i cinque insiemi di lettere S1…S5 creati artificialmente:



I ***volcano plot*** (diagrammi "a vulcano", già visti nella sezione 11.12) sono utilizzati per tracciare i *fold change* con i relativi *p*-value. Vedremo in questa sezione una rappresentazione più intuitiva di tale tipo di grafico utilizzando la libreria *ggplot2*:

```
> library(ggplot2)
```

Visualizziamo i primi geni contenuti nella variabile *genes* — creata mediante l'analisi *limma* della sezione 11.12 — digitando il seguente comando:

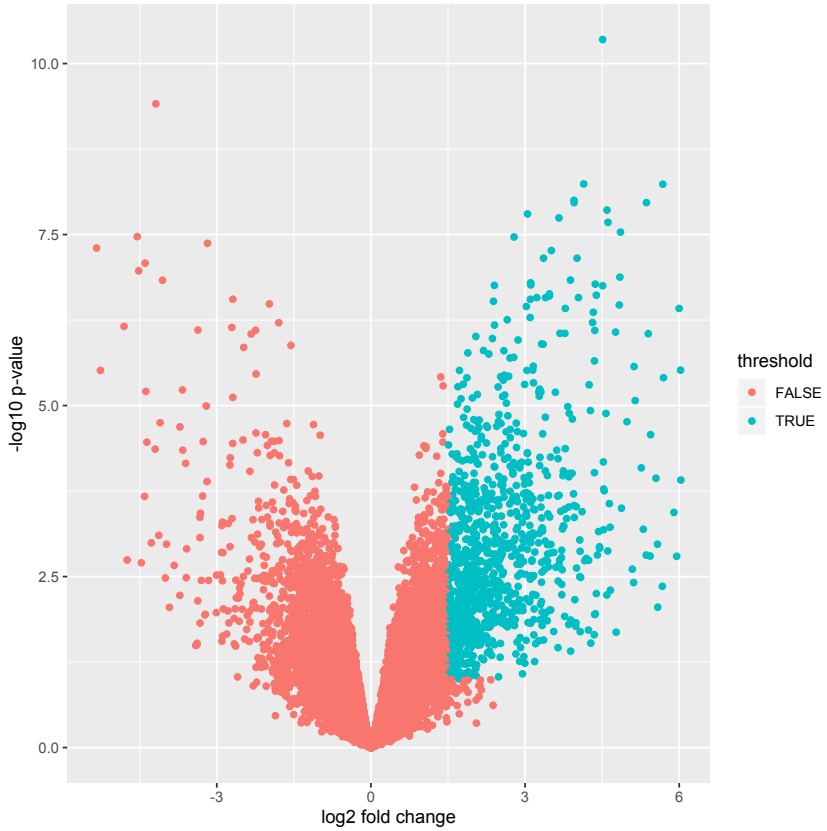
```
> head(genes)
                    logFC  AveExpr         t      P.Value    adj.P.Val        B
ENSG00000152078  4.510507 4.856523  28.13988 4.463747e-11 9.004270e-07 14.01472
ENSG00000117519 -4.185175 4.791585 -22.73888 3.878292e-10 3.911645e-06 12.69738
ENSG00000145850  4.142236 4.507655  17.38636 5.759942e-09 2.925048e-05 10.72782
ENSG00000170180  5.681327 5.734169  17.37423 5.800214e-09 2.925048e-05 10.72231
ENSG00000087586  3.952183 5.720789  16.45393 9.977396e-09 3.111188e-05 10.28705
ENSG00000047597  5.362419 5.108415  16.32474 1.079114e-08 3.111188e-05 10.22315
```

Utilizziamo come soglia i *p*-value e il log *fold change* per definire i colori nel grafico:

```
> threshold <- as.factor(genes$logFC>1.5 & genes$P.Value<0.1)
```

Utilizziamo ora i dati e l'oggetto *threshold* per creare — mediante la funzione `ggplot()` — e visualizzare il grafico *g* come segue:

```
> g <- ggplot(data = genes, aes(x = logFC, y = -log10(P.Value), colour =
threshold)) + geom_point() + xlab("log2 fold change") + ylab("-log10 p-value")
> g
```



Riassumendo, nella parte relativa alle *heatmap* abbiamo utilizzato i risultati del clustering bidirezionale per i campioni e i geni. Si possono utilizzare più argomenti per aggiungere informazioni al grafico, come etichette e titoli, e anche la tavolozza dei colori può essere cambiata. Per ulteriori informazioni, è possibile visualizzare l'aiuto in R digitando `?clustering.plot`.

Nella visualizzazione del *diagramma di Venn* abbiamo utilizzato un oggetto lista contenente cinque insiemi (di lettere generate in modo casuale) per i quali il diagramma è stato tracciato. Per poter utilizzare la funzione `venn.diagram()` per un numero diverso di insiemi, è necessario modificare parametri come `cat.col` e `cex`. Esempi con un diverso numero di insiemi sono illustrati nel file di aiuto della funzione (digitare `?venn.diagram` per una spiegazione dettagliata). Questi grafici possono essere usati per mostrare diversi attributi dei risultati nell'analisi dei dati. Ad esempio, se si stanno confrontando più trattamenti con un controllo, possiamo visualizzare quanti geni/sonde differenzialmente espressi sono condivisi negli esperimenti. Un'altra situazione possibile può essere la condivisione di termini GO sovraespressi in due o più analisi.

Il terzo tipo di grafico (*volcano plot*) è piuttosto semplice e utilizza la funzione `ggplot()` invece della normale funzione `plot()`. Essa separa i geni in termini di colori, mostrando i geni con *p*-value inferiori a 0,1 e variazione assoluta di log *fold change* maggiore di 1,5.

12. Analisi di dati GWAS (Genome Wide Association Study)

Gli studi di associazione a livello genomico (GWAS, Genome Wide Association Study) sono un metodo per identificare i loci di suscettibilità per le malattie complesse. Si basa sulla tecnica di scansione dei genomi di molti soggetti al fine di identificare la variazione genetica eventualmente responsabile di una malattia attraverso test statistici. La tecnica necessita di un gran numero di soggetti per identificare in modo affidabile la variazione. Con l'avvento delle tecnologie microarray ad alta densità per il rilevamento dei polimorfismi a singolo nucleotide (SNP, Single Nucleotide Polymorphism) e il progetto HapMap (https://hapmap.ncbi.nlm.nih.gov) insieme al Progetto Genoma Umano (HGP, https://www.genome.gov/human-genome-project), questa tecnica è stata resa possibile.

Attualmente, il progetto HapMap è stato ritirato ed è confluito nel progetto 1KG (1000 Genomi, https://www.ncbi.nlm.nih.gov/variation/tools/1000genomes/), che a sua volta è ricompreso nel progetto IGSR (The International Genome Sample Resource, https://www.internationalgenome.org), lo standard della ricerca per la genetica delle popolazioni e la genomica. La missione originale del progetto internazionale HapMap era quella di sviluppare una mappa aplotipica del genoma umano, HapMap, che descrivesse i modelli comuni di variazione della sequenza del DNA umano. Attraverso questa ricerca sono stati scoperti milioni di SNP e molti studi GWAS hanno utilizzato questo set di dati nella ricerca per l'associazione di malattie. Questo progetto è stato un trampolino di lancio necessario per il progetto 1KG che utilizza molte delle stesse popolazioni. I dati HapMap archiviati continuano ad essere disponibili via FTP all'indirizzo ftp://ftp.ncbi.nlm.nih.gov/hapmap/.

Una serie di studi SNP sono disponibili su SNPedia (https://snpedia.com/index.php/SNPedia), che condivide informazioni sugli effetti delle variazioni del DNA sulla base della letteratura peer-reviewed. Un database popolare per gli SNP è dbSNP, disponibile alla pagina NCBI https://www.ncbi.nlm.nih.gov/snp/. OMIM (Online Mendelian Inheritance in Man) è un'altra fonte di informazioni sui geni umani e fenotipi genetici, disponibile su https://omim.org.

L'analisi GWAS utilizza due gruppi di soggetti: i controlli (cioè i soggetti senza la malattia o i tratti desiderati), e i casi (malati o con i tratti desiderati). Gli SNP vengono rilevati dai soggetti e si riferiscono ad una variazione nella sequenza del DNA in cui un singolo nucleotide differisce tra due fenotipi o cromosomi. Ad esempio, in una sequenza ATCG**T**ACG la variante può essere ATCG**C**ACG, dove la T al centro diventa C. Per trovare l'associazione di questi SNP con alcuni fenotipi, la presenza di tali SNP viene verificata statisticamente con campioni di questi fenotipi. Ad esempio, gli SNP significativamente più frequenti nelle persone che soffrono della malattia, rispetto a quelli che non lo sono, sono associati alla malattia. Nello studio GWA per il controllo di una patologia, viene calcolato il conteggio allelico degli SNP per i casi/controlli e poi viene eseguito un test statistico — come ad es. il test Chi-quadro — per identificare le varianti associate alla malattia in esame. Ciò comporta l'analisi di un'enorme quantità di dati genotipici, che possono essere delle dimensioni di centinaia di MB fino a GB.

Prima di iniziare a lavorare con i dati GWAS, vediamo come sono costituiti. I dati GWAS in qualsiasi formato sono ottenuti da studi sul genoma; includono anche informazioni sulle varianti, come la mappatura

sui cromosomi per ogni individuo, e sono strutturati in modi diversi a seconda del formato dei dati, come ad esempio tabelle o semplici file. Alcuni di questi formati di dati saranno discussi nel corso di questo capitolo. Per avere un'idea dettagliata dei dati GWAS, consultare l'articolo sugli studi di associazione a livello di genoma di Bush e Moore all'indirizzo https://journals.plos.org/ploscompbiol/article?id=10.1371/journal.pcbi.1002822. In questo capitolo illustreremo l'analisi dei dati GWAS al fine di effettuare inferenze biologiche.

12.1 L'analisi delle associazioni SNP

Il GWAS ha guadagnato popolarità come approccio per identificare varianti genetiche associate a un fenotipo di malattia attraverso l'analisi statistica dei dati. In questa sezione vedremo alcuni metodi di analisi statistica per stabilire l'associazione tra ogni singolo SNP e il fenotipo osservato, utilizzando i dati disponibili in alcuni pacchetti.

Iniziamo con l'installazione del pacchetto *SNPassoc* da CRAN:

```
> install.packages("SNPassoc") # a package for association studies
> library(SNPassoc)
```

Utilizziamo i dati disponibili in questo pacchetto, che possono essere caricati ed esaminati come segue:

```
> data(SNPs)
> head(SNPs)[,1:10]
  id casco    sex blood.pre  protein snp10001 snp10002 snp10003 snp10004 snp10005
1  1     1 Female      13.7 75640.52       TT       CC       GG       GG       GG
2  2     1 Female      12.7 28688.22       TT       AC       GG       GG       AG
3  3     1 Female      12.9 17279.59       TT       CC       GG       GG       GG
4  4     1   Male      14.6 27253.99       CT       CC       GG       GG       GG
5  5     1 Female      13.4 38066.57       TT       AC       GG       GG       GG
6  6     1 Female      11.3  9872.46       TT       CC       GG       GG       GG
> head(SNPs.info.pos)
       snp  chr     pos
1 snp10001 Chr1 2987398
2 snp10002 Chr1 1913558
3 snp10003 Chr1 1982067
4 snp10004 Chr1  447403
5 snp10005 Chr1 2212031
6 snp10006 Chr1 2515720
```

Creiamo quindi un oggetto SNP usando la funzione `setupSNP()` con le informazioni nelle colonne da 6 a 40 ed eseguiamo un test di associazione per la caratteristica e le variabili di interesse:

```
> mySNP <- setupSNP(SNPs, 6:40, sep="")
> myres <- association(casco~sex+snp10001+blood.pre, data = mySNP, model.interaction
      = c("dominant","codominant"))
```

Esaminiamo i risultati digitando il nome della variabile:

```
> myres

SNP: snp10001  adjusted by: sex blood.pre
               0    %   1    %   OR lower upper p-value   AIC
Codominant
T/T           24 51.1  68 61.8 1.00             0.15410 195.8
C/T           21 44.7  32 29.1 0.55  0.26  1.14
C/C            2  4.3  10  9.1 1.74  0.35  8.63
Dominant
T/T           24 51.1  68 61.8 1.00             0.22859 196.1
C/T-C/C       23 48.9  42 38.2 0.65  0.32  1.31
Recessive
T/T-C/T       45 95.7 100 90.9 1.00             0.28494 196.4
C/C            2  4.3  10  9.1 2.22  0.46 10.70
Overdominant
T/T-C/C       26 55.3  78 70.9 1.00             0.07188 194.3
C/T           21 44.7  32 29.1 0.52  0.25  1.06
log-Additive
0,1,2         47 29.9 110 70.1 0.87  0.51  1.49 0.60861 197.3
```

La variabile *SNPs* è un oggetto data.frame che consiste di 157 righe e 40 colonne. Ogni riga rappresenta un campione, che può essere un caso o un controllo. Le prime cinque colonne rappresentano gli attributi di ogni campione; le informazioni SNP sono contenute nelle colonne dalla 6 alla 40, che abbiamo appunto usato durante la creazione dell'oggetto *mySNP* di tipo snp. Le prime cinque colonne contengono: gli identificatori dei campioni, la tipologia "casco", cioè caso (0) o controllo (1), il genere, le informazioni sulla pressione sanguigna e sul livello di proteine.

L'altra variabile *SNPs.info.pos* contiene informazioni sul nome SNP, il nome del cromosoma e la posizione genomica.

Anche l'oggetto *mySNP* che abbiamo creato ha una struttura data.frame, ma ha un aspetto leggermente diverso per quanto riguarda la codifica delle informazioni SNP:

```
> head(mySNP)[,1:10]
  id casco    sex blood.pre  protein snp10001 snp10002 snp10003 snp10004 snp10005
1  1     1 Female      13.7 75640.52      T/T      C/C      G/G      G/G      G/G
2  2     1 Female      12.7 28688.22      T/T      A/C      G/G      G/G      A/G
3  3     1 Female      12.9 17279.59      T/T      C/C      G/G      G/G      G/G
4  4     1   Male      14.6 27253.99      C/T      C/C      G/G      G/G      G/G
5  5     1 Female      13.4 38066.57      T/T      A/C      G/G      G/G      G/G
6  6     1 Female      11.3  9872.46      T/T      C/C      G/G      G/G      G/G
```

La funzione `association()` esegue un'analisi di associazione tra un singolo SNP e una variabile dipendente tramite l'adattamento del modello e il calcolo delle statistiche. Gli argomenti includono la formula del modello da adattare nella forma: "caratteristica di interesse"~"variabile in esame" (nel nostro esempio `casco~sex+snp10001+blood.pre`). È possibile aggiungere altre variabili alla formula dopo

la prima utilizzando l'operatore +, ma l'analisi viene fatta solo per la prima e regolata sulle altre. La funzione di associazione può modellare la dipendenza sulla base dei modelli genetici di ereditarietà codominante, dominante, recessiva, sovradominante e log-additiva. Questo viene fornito alla funzione con l'argomento `model.interaction`.

In questa sezione, abbiamo cercato di trovare la dipendenza del casi/controlli sul primo SNP (snp10001), il sesso e la pressione sanguigna. Abbiamo usato due modelli genetici: dominante e codominante. Il risultato è una matrice le cui colonne contengono informazioni sulla dimensione del campione e le percentuali per ogni genotipo, il rapporto di probabilità, il suo intervallo di confidenza al 95% (che prende come riferimento il genotipo omozigote più frequente) e il *p*-value corrispondente al test *likelihood ratio* ottenuto da un confronto con il modello nullo. Inoltre, la matrice contiene anche il criterio di informazione Akaike (AIC) di ogni modello genetico. La tabella *myres* riflette la dipendenza dei fenotipi dall'SNP (insieme ad altri fattori) come si può vedere nella schermata all'inizio della pagina precedente.

Esistono altri pacchetti per effettuare analisi simili, come *snp.plotter* e *GenABEL*, che esamineremo in seguito.

12.2 Esecuzione di scansioni delle associazioni SNP

Nella sezione precedente abbiamo presentato l'approccio per scoprire la dipendenza di un fenotipo da un singolo SNP. Tuttavia, nel caso di fenotipi come le malattie complesse, bisogna esaminare tutti gli SNP e scandire l'intero dataset. Questa sezione illustra il calcolo della significatività della dipendenza dei fenotipi da tutti gli SNP disponibili nei dati, continuando a lavorare sullo stesso dataset *SNPs* e sulla stessa libreria *SNPassoc*.

Iniziamo dall'oggetto *mySNP* creato in precedenza per studiare un altro fenotipo, il livello di proteine (nella formula è quindi inutile specificare dopo la caratteristica anche le variabili, dovendole esaminare tutte). Eseguiamo la funzione `WGassociation()` sull'oggetto *mySNP* per tutti i modelli genetici e diamo un'occhiata ai risultati come segue:

```
> myres <- WGassociation(protein, data = mySNP, model = "all")
> myres
          comments    codominant dominant recessive overdominant log-additive
snp10001  -           0.00586    0.00516  0.01605   0.12306      0.00140
snp10002  -           0.78555    0.93303  0.48704   0.87288      0.76846
snp10003  Monomorphic -          -        -         -            -
snp10004  Monomorphic -          -        -         -            -
snp10005  -           0.63306    0.43881  0.50130   0.55427      0.37267
snp10006  Monomorphic -          -        -         -            -
snp10007  Monomorphic -          -        -         -            -
snp10008  -           0.20627    0.30005  0.08652   0.83655      0.13493
snp10009  -           0.74736    0.87204  0.47815   0.68152      0.93616
snp100010 Monomorphic -          -        -         -            -
…
snp100030 Monomorphic -          -        -         -            -
snp100031 Genot 65%   -          -        -         -            -
snp100032 -           0.01943    0.00656  0.09656   0.23993      0.00717
snp100033 -           0.01253    0.00455  0.06839   0.26544      0.00428
snp100034 -           0.00859    0.02183  0.00548   0.47601      0.00333
snp100035 Monomorphic -          -        -         -            -
```

Per osservare i *p*-value per specifici modelli genetici, utilizziamo come argomento il nome della modalità come funzione e la matrice completa dei risultati, come segue:

```
> dominant(myres)
 [1] 0.005164929 0.933029300          NA          NA 0.438808500          NA          NA
 [8] 0.300054230 0.872038400          NA 0.129205100 0.583599200 0.098778790 0.015114890
[15]          NA          NA 0.791264500 0.886923300 0.007494522 0.313808950          NA
[22]          NA 0.995438100 0.017737798          NA          NA 0.714963100 0.005040410
[29] 0.021833942          NA          NA 0.006561046 0.004546931 0.021833942          NA
> recessive(myres)
 [1] 0.016047565 0.487039400          NA          NA 0.501301800          NA          NA
 [8] 0.086524570 0.478153600          NA 0.272885900 0.479292900 0.110399960 0.119523580
[15]          NA          NA 0.459659700 0.478153600 0.147754655 0.086524570          NA
[22]          NA 0.481948100 0.005475794          NA          NA 0.949004900 0.087582190
[29] 0.005475794          NA          NA 0.096555480 0.068387908 0.005475794          NA
```

Per ottenere il risultato dettagliato per ogni SNP, utilizziamo la funzione `WGstats()`:

```
> WGstats(myres)
$snp10001

SNP: snp10001  adjusted by:
               n    me   se    dif  lower   upper  p-value  AIC
Codominant
T/T           92 47419 2393      0                 0.005861 3602
C/T           53 38987 3177  -8432 -16165  -698.2
C/C           12 27413 6061 -20006 -33770 -6241.5
Dominant
T/T           92 47419 2393      0                 0.005165 3603
C/T-C/C       65 36851 2858 -10568 -17870 -3266.8
Recessive
T/T-C/T      145 44337 1935      0                 0.016048 3605
C/C           12 27413 6061 -16924 -30549 -3298.9
Overdominant
T/T-C/C      104 45111 2308      0                 0.123062 3608
C/T           53 38987 3177  -6123 -13864  1617.1
log-Additive
0,1,2                       -9332 -14955 -3708.6 0.001404 3601

$snp10002

SNP: snp10002  adjusted by:
               n    me   se    dif  lower upper p-value  AIC
Codominant
C/C           74 42876 2890    0.0                0.7855 3612
A/C           78 42740 2576 -135.8  -7648  7377
A/A            5 50262 6879 7385.6 -14006 28777
Dominant
C/C           74 42876 2890    0.0                0.9330 3611
A/C-A/A       83 43193 2456  317.3  -7072  7706
Recessive
C/C-A/C      152 42806 1924    0.0                0.4870 3610
A/A            5 50262 6879 7455.3 -13518 28429
Overdominant
C/C-A/A       79 43343 2742    0.0                0.8729 3611
A/C           78 42740 2576 -603.2  -7980  6773
log-Additive
0,1,2                        996.5  -5626  7619  0.7685 3611
…
```

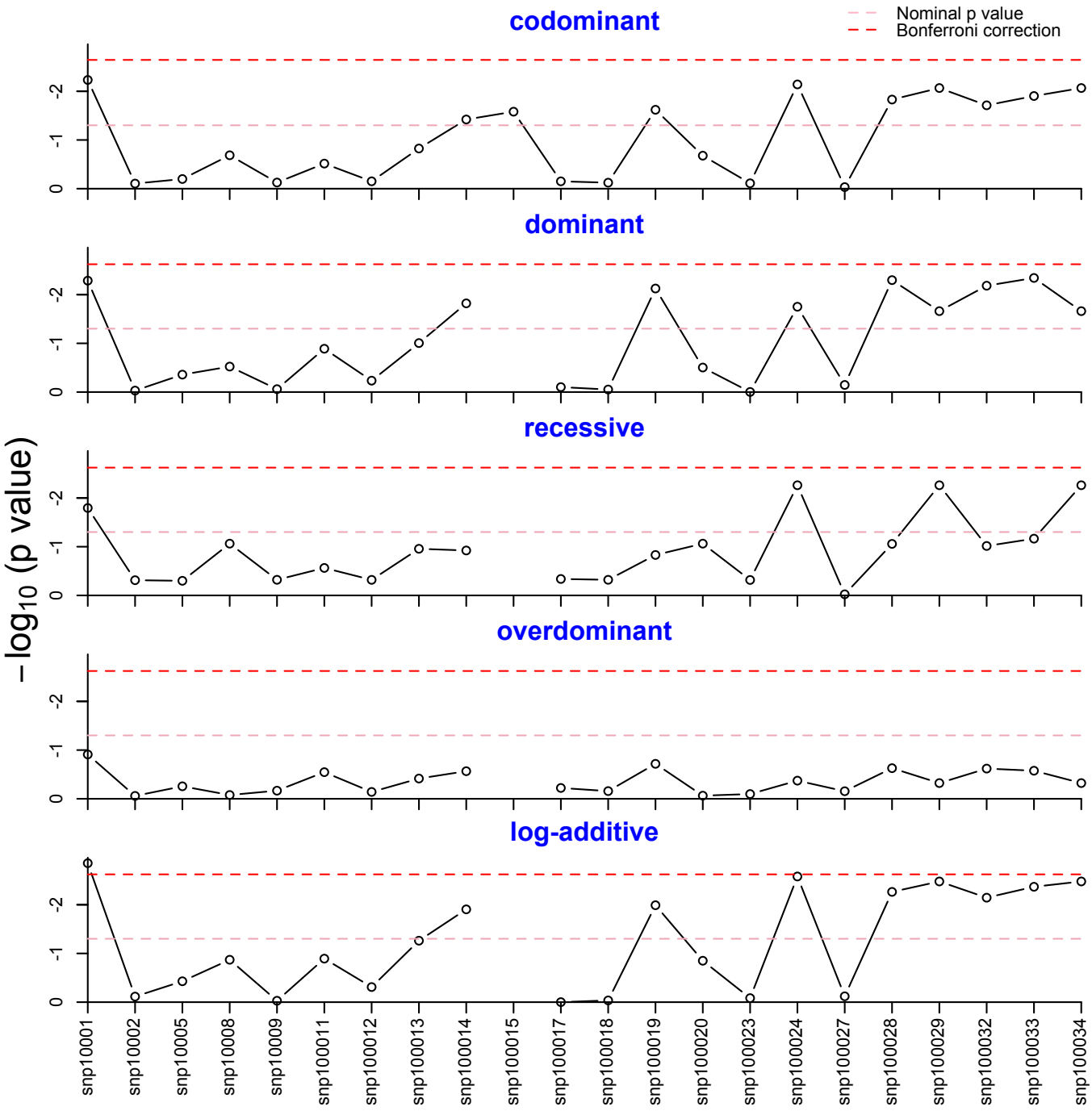
Visualizziamo il riepilogo generale con la funzione `summary()`:

```
> summary(myres)
 SNPs (n) Genot error (%) Monomorphic (%) Significant* (n) (%)
       35               0            34.3                 0   0

 *Number of statistically significant associations at level 1e-06
```

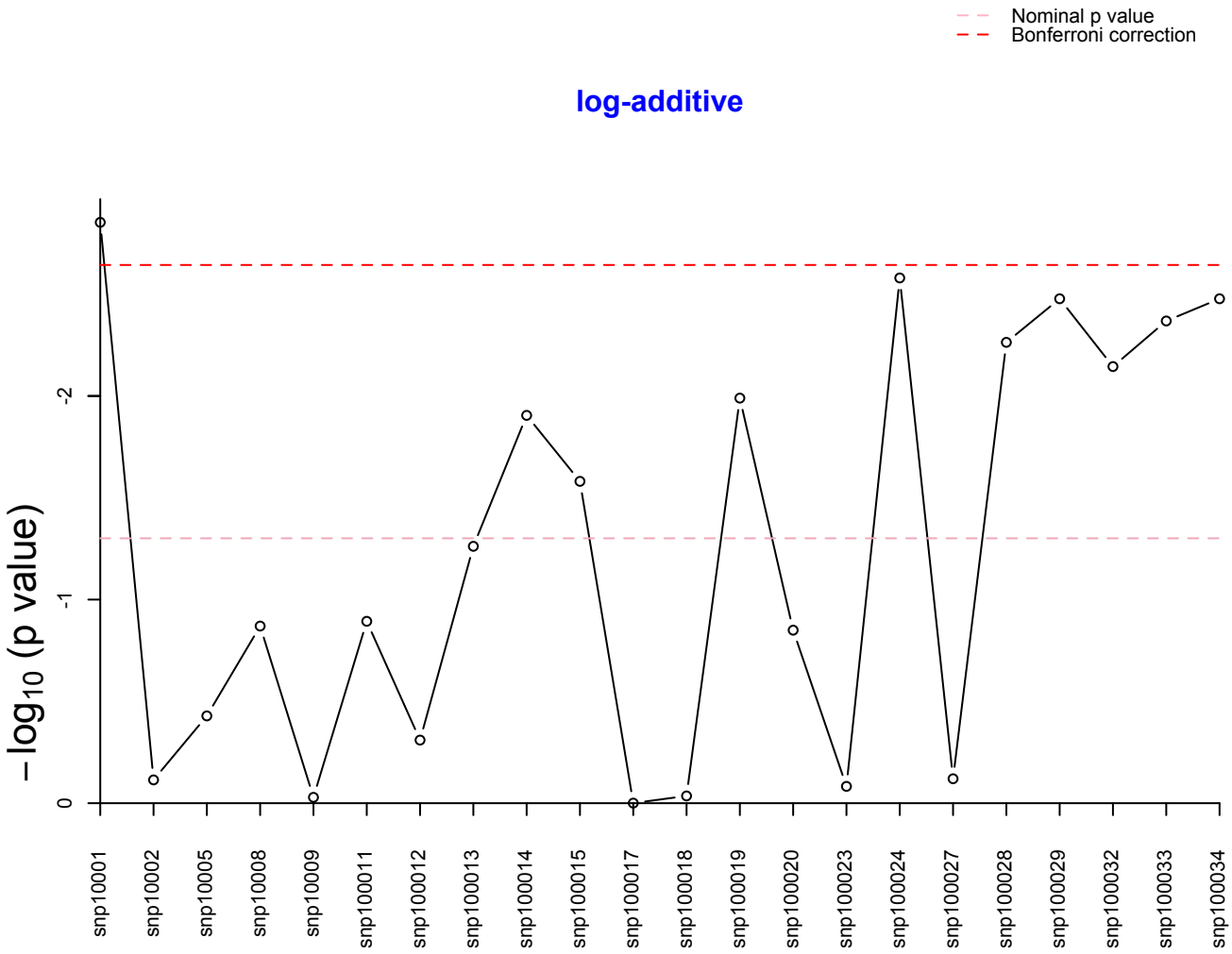
Tracciamo i *p*-value (nessuno significativo) per tutti i modelli e SNP come segue:

```
> plot(myres)
```



Utilizziamo infine la funzione `WGassociation()` per un'analisi dell'intera associazione genomica calcolando solo i *p*-value per ogni SNP corrispondenti al modello *log-additivo*:

```
> resHapMap <- WGassociation(protein, data=mySNP, model="log-additive")
> plot(resHapMap)
```



Abbiamo visto che le funzioni `association()` e `WGassociation()` operano in modo simile. La differenza principale è che la seconda esegue molti test di associazione in parallelo usando la funzione `mclapply()`, che prende ogni SNP come variabile da cui dipende il fenotipo ed esegue l'associazione con ognuno di essi in parallelo. Il risultato è una matrice per ciascun SNP, da cui possono essere estratti i *p*-value corrispondenti.

I grafici mostrano i *p*-value per ogni SNP in scala logaritmnica assoluta lungo l'asse *y* e gli SNP (per modello genetico) lungo l'asse *x*. I *p*-value sono accompagnati dalla correzione di Bonferroni, con l'indicazione di quanti SNP sono significativi per ogni modello di ereditarietà. Nel nostro caso non abbiamo riscontrato alcuna associazione significativa per il modello cercato.

12.3 Analisi delle associazioni SNP dell'intero genoma

Finora abbiamo lavorato con alcuni dati SNP e abbiamo fatto dei tentativi per comprendere la dipendenza fenotipica da questi SNP. Tuttavia, dobbiamo ancora coprire la parte più interessante dell'analisi delle associazioni, cioè l'analisi di associazione di un tratto sull'intero genoma.

Abbiamo sinora utilizzato un piccolo set di dati. Ora passiamo ad un dataset molto più grande chiamato "HapMap", disponibile all'interno del pacchetto *SNPassoc* di R:

```
> library(SNPassoc)
> data(HapMap)
```

Verifichiamo l'aspetto dei dati (esaminando solo le prime voci) con la funzione `str()`:

```
> str(HapMap)
'data.frame':     120 obs. of  9307 variables:
 $ id        : Factor w/ 120 levels "NA06985","NA06993",..: 1 2 3 4 5 6 7 8 9 10 ...
 $ group     : Factor w/ 2 levels "CEU","YRI": 1 1 1 1 1 1 1 1 1 1 ...
 $ rs10399749: Factor w/ 1 level "CC": 1 1 1 1 1 1 1 1 1 1 ...
 $ rs11260616: Factor w/ 3 levels "AA","AT","TT": 1 2 1 2 1 1 1 1 2 2 ...
 $ rs4648633 : Factor w/ 3 levels "CC","CT","TT": 3 2 3 3 2 3 3 2 2 NA …
…
> str(HapMap.SNPs.pos)
'data.frame':     9305 obs. of  3 variables:
 $ snp       : chr  "rs10399749" "rs11260616" "rs4648633" "rs6659552" ...
 $ chromosome: chr  "chr1" "chr1" "chr1" "chr1" ...
 $ position  : int  45162 1794167 2352864 2902617 3170389 3382844 3766548 4038743 ...
```

Il passo successivo è quello di creare l'oggetto di tipo SNP con il seguente comando:

```
> myHapMap <- setupSNP(HapMap, colSNPs=3:9307, sort=TRUE, info=HapMap.SNPs.pos,
sep="")
```

Eseguiamo ora la funzione `WGassociation()` su questo oggetto dati per il modello genetico di nostro interesse (nel nostro caso, usiamo "dominant"):

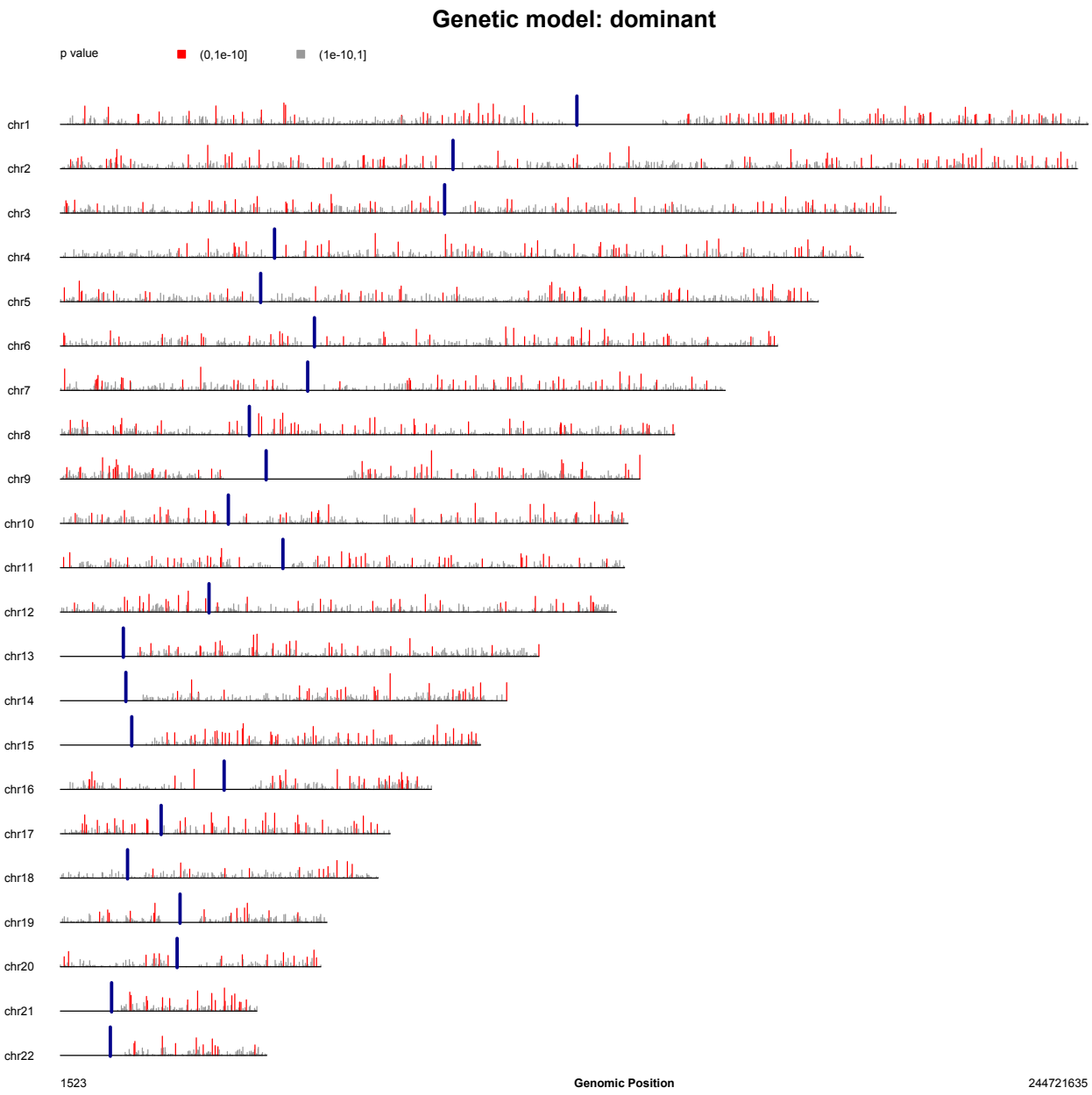
```
> myHapMapres <- WGassociation(group, data= myHapMap, model="dominant")
```

Il risultato *myHapMapres* è un dataframe; controlliamone il contenuto:

```
> myHapMapres
           comments    dominant
rs10399749 Monomorphic -
rs11260616 -           0.09851
rs4648633  -           0.00000
rs6659552  -           0.00000
rs7550396  -           1.00000
rs12239794 Monomorphic -
…
rs11574907 Monomorphic -
rs12000915 Monomorphic -
rs7851392  -           0.00000
```

Digitiamo infine il seguente comando per tracciare graficamente l'intero risultato:

```
> plot(myHapMapres, whole=TRUE)
```



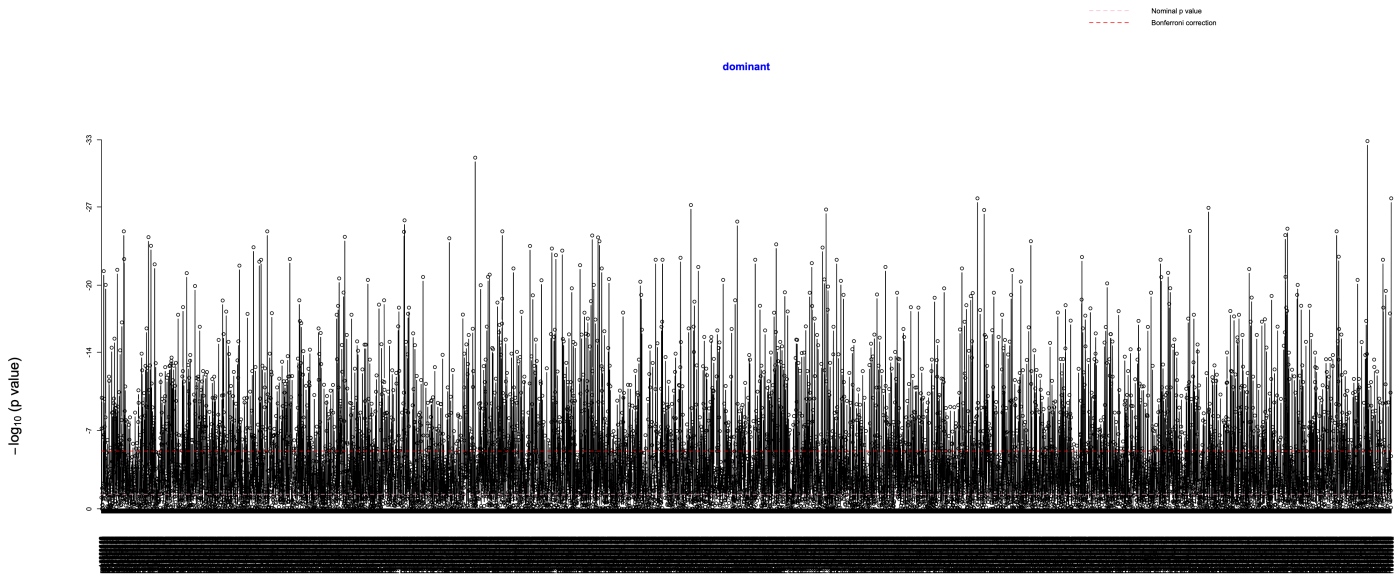
I dati HapMap utilizzati in questa sezione sono costituiti da due gruppi di popolazione, CEU (residenti nello Utah con antenati dell'Europa settentrionale e occidentale) e YRI (Yoruba a Ibadan, Nigeria). Ogni gruppo è rappresentato da 60 campioni (in totale 120 campioni). La seguente schermata mostra la parte iniziale dei dati *HapMap* con il gruppo di identificazione del campione e le coppie nucleotidiche presso i siti SNP:

```
       id group rs10399749 rs11260616 rs4648633 rs6659552 rs7550396 rs12239794 rs6688969
1 NA06985   CEU         CC         AA         TT         GG         GG         GG         CC
2 NA06993   CEU         CC         AT         CT         CG         GG         GG         CT
3 NA06994   CEU         CC         AA         TT         CG         GG         GG         CT
4 NA07000   CEU         CC         AT         TT         GG         GG       <NA>         CC
5 NA07022   CEU         CC         AA         CT         GG         GG         GG         CT
6 NA07034   CEU         CC         AA         TT         CG         GG         GG         CC
…
```

Il data frame *HapMap.SNPs.pos* contiene le informazioni di mappatura genomica per gli SNP sui cromosomi umani, come mostrato nella seguente schermata:

```
         snp chromosome position
1 rs10399749       chr1    45162
2 rs11260616       chr1  1794167
3  rs4648633       chr1  2352864
4  rs6659552       chr1  2902617
5  rs7550396       chr1  3170389
6 rs12239794       chr1  3382844
…
```

Il principio alla base di questa analisi è per lo più lo stesso della precedente. La funzione `setupSNP()` converte un oggetto data.frame nella classe snp per essere utilizzato dalla funzione `WGassociation()`. L'input richiesto è l'oggetto data.frame e il numero delle colonne dove si trovano gli SNP nell'oggetto data.frame. La differenza è nel grafico che è stato creato in questa sezione, che mostra i *p*-value dell'associazione dei fenotipi mappati sulla loro corrispondente posizione genomica sul cromosoma umano. L'asse *y* rappresenta i -$\log_{10}$(*p*-value) per l'intera analisi lungo tutti i cromosomi. Ogni linea verticale corrisponde ad una posizione o ad un SNP su quel cromosoma specifico, mentre le linee blu rappresentano il centromero. Gli SNP significativi sono in rosso (gli altri sono in grigio); si noti che il modello assunto è un modello genetico dominante. Il grafico visualizza tutti gli SNP e il loro significato lungo le posizioni cromosomiche; possiamo impostare l'argomento `whole` su `FALSE` per mettere i risultati insieme lungo l'asse orizzontale:



Prima di convertire i dati in un oggetto SNP, è sempre opportuno esaminarli per capire quali sono i descrittori e le variabili disponibili. Può essere utile anche nei casi in cui i dati fenotipici non sono disponibili e bisogna crearli manualmente (artificialmente o da altre fonti). Si osservi che l'analisi di dati di grandi dimensioni come il dataset *HapMap* può richiedere molto tempo e memoria di calcolo.

I dati *HapMap* sono un'enorme raccolta di varianti genetiche di diverse popolazioni; CEU e YRI (che abbiamo utilizzato nel nostro esempio) sono due di queste popolazioni. Un dettaglio completo di altri descrittori è disponibile all'indirizzo https://www.ncbi.nlm.nih.gov/variation/news/NCBI_retiring_HapMap/.

12.4 Importazione di dati GWAS in formato PLINK

Nelle sezioni precedenti abbiamo utilizzato i dati GWAS nel formato "data.frame", contenente normalmente i dati del genotipo (che hanno informazioni sulle varianti SNP) e le informazioni di mappatura (che contengono i nomi SNP e le informazioni correlate). Uno dei formati di dati GWAS più diffusi (ma non semplice come i data.frame) è il formato PLINK, costituito da due file separati: uno contenente le informazioni SNP (con estensione ".ped") e l'altro le informazioni di mappatura (con estensione ".map"). Per l'analisi delle dipendenze, questo formato può essere combinato con i dati fenotipici. Un tool software (PLINK 2.00 https://www.cog-genomics.org/plink/2.0/) di recente implementazione per la gestione ed analisi di dati GWAS in formato PLINK è stato sviluppato dal Dipartimento di Scienze dei Dati Biomedici dell'università di Stanford (http://med.stanford.edu/dbds.html), in grado di gestire anche il recente formato VCF del progetto 1000 Genomes (ftp://ftp.1000genomes.ebi.ac.uk/vol1/ftp/).

In questa sezone vediamo come utilizzare i dati GWAS in formato PLINK "tradizionale" importandoli appositamente in R. Per prima cosa abbiamo bisogno dei file .map e .ped da leggere in R, che possiamo scaricare dal sito dei dati HapMap archiviati dall'NCBI all'indirizzo ftp://ftp.ncbi.nlm.nih.gov/hapmap/ (esaminare la cartella "genotypes").

I file possono essere di dimensioni enormi, e la loro elaborazione potrebbe richiedere molto tempo; pertanto, per il nostro esempio, utilizzeremo dei file di dimensioni ridotte (scaricabili facendo click sul loro nome) che si chiamano myPed.ped, myMap.map e myPheno.ph. Iniziamo con l'installazione e il caricamento della libreria GenABEL:

```
> install.packages("GenABEL")
> library(GenABEL)
```

Convertiamo quindi i file "ped" e "map" in dati di genotipo mediante la seguente funzione:

```
> convert.snp.ped(pedfile="myPed.ped", mapfile="myMap.map", out="myOut.out")
Reading map from file 'myMap.map' ...
... done.  Read positions of 214 markers from file 'myMap.map'
Reading genotypes from file 'myPed.ped' ...
...done.  Read information for 90 people from file 'myPed.ped'
Analysing marker information ...
Writing to file 'myOut.out' ...
... done.
```

Abbiamo anche bisogno del file fenotipico "myPheno.ph" — avente come colonne i nomi dei campioni, il sesso e il flag del tratto di interesse (0=assente / 1=presente) — per caricare in R i dati fenotipici e genotipici completi da elaborare:

```
> myData <- load.gwaa.data(phenofile="myPheno.ph", genofile="myOut.out")
ids loaded...
marker names loaded...
chromosome data loaded...
map data loaded...
allele coding data loaded...
strand data loaded...
genotype data loaded...
snp.data object created...
assignment of gwaa.data object FORCED; X-errors were not checked!
```

Osserviamo la struttura dell'oggetto *myData*:

```
> str(myData)
Formal class 'gwaa.data' [package "GenABEL"] with 2 slots
  ..@ phdata:'data.frame':     90 obs. of  3 variables:
  .. ..$ id : chr [1:90] "NA06985" "NA06991" "NA06993" "NA06994" ...
  .. ..$ sex: int [1:90] 1 0 0 0 1 1 0 0 1 1 ...
  .. ..$ aff: int [1:90] 1 0 1 0 1 0 0 1 0 1 ...
  ..@ gtdata:Formal class 'snp.data' [package "GenABEL"] with 11 slots
  .. .. ..@ nbytes    : num 23
  .. .. ..@ nids      : int 90
  .. .. ..@ nsnps     : int 214
  .. .. ..@ idnames   : chr [1:90] "NA06985" "NA06991" "NA06993" "NA06994" ...
  .. .. ..@ snpnames  : chr [1:214] "rs679153" "rs9965482" "rs16943926" "rs16943929" ...
  .. .. ..@ chromosome: Factor w/ 1 level "18": 1 1 1 1 1 1 1 1 1 1 ...
  .. .. .. ..- attr(*, "names")= chr [1:214] "rs679153" "rs9965482" "rs16943926"
"rs16943929" ...
  .. .. ..@ map       : Named num [1:214] 2859916 2860891 2861667 2861811 2862091 ...
  .. .. .. ..- attr(*, "names")= chr [1:214] "rs679153" "rs9965482" "rs16943926"
"rs16943929" ...
  .. .. ..@ coding    :Formal class 'snp.coding' [package "GenABEL"] with 1 slot
  .. .. .. .. ..@ .Data: raw [1:214] 17 1b 1b 1b ...
  .. .. ..@ strand    :Formal class 'snp.strand' [package "GenABEL"] with 1 slot
  .. .. .. .. ..@ .Data: raw [1:214] 00 00 00 00 ...
  .. .. ..@ male      : Named int [1:90] 1 0 0 0 1 1 0 0 1 1 ...
  .. .. .. ..- attr(*, "names")= chr [1:90] "NA06985" "NA06991" "NA06993" "NA06994" ...
  .. .. ..@ gtps      :Formal class 'snp.mx' [package "GenABEL"] with 1 slot
  .. .. .. .. ..@ .Data: raw [1:23, 1:214] 97 6a e9 e6 ...
```

Consultiamo i dati del genotipo e del fenotipo con l'aiuto delle seguenti funzioni:

```
> phdata(myData)
             id sex aff
NA06985 NA06985   1   1
NA06991 NA06991   0   0
NA06993 NA06993   0   1
NA06994 NA06994   0   0
NA07000 NA07000   1   1
…
> gtdata(myData)
@nids = 90
@nsnps = 214
@nbytes = 23
@idnames = NA06985 NA06991 NA06993 NA06994 NA07000 NA07019 NA07022 NA07029 NA07034 …
@snpnames = rs679153 rs9965482 rs16943926 rs16943929 rs612071 rs9950998 rs613796 …
@chromosome = 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 …
@coding =  17 1b 1b 1b 01 17 17 17 17 01 01 01 01 01 17 01 17 17 01 1b 17 17 01 1b 01 17 …
@strand =  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 …
@map = 2859916 2860891 2861667 2861811 2862091 2862368 2862449 2862586 2862679 2862721 …
@male = 1 0 0 0 1 1 0 0 1 1 0 1 1 0 0 0 1 1 0 0 1 1 1 1 0 0 0 1 0 1 1 0 0 0 1 0 0 0 1 …
@gtps = 80 40 40 40 80 40 80 80 40 40 40 40 40 40 80 40 40 80 40 40 c0 40 80 40 40 c0 c0 …
```

I dati HapMap sono costituiti da dati di genotipi e fenotipi individuali. I dati scaricati dal progetto HapMap sono costituiti da due diversi tipi di informazioni. Il primo è il file .ped, che contiene le informazioni sul genotipo, e il secondo è il file .map, che consiste nei nomi SNP e nelle loro mappature. I dati del nostro esempio riguardano i residenti dello Utah con antenati dell'Europa settentrionale e occidentale (CEU). I dati sono composti da 90 individui e dai corrispondenti SNP sul diciottesimo cromosoma. I dati fenotipici contengono informazioni sul sesso e sul fenotipo di ogni individuo presente nei dati. Durante la lettura di

questi file, la funzione `load.gwaa.data()` ricava dai dati le informazioni sul genotipo e sul marcatore, come abbiamo visto prima nella console R.

L'oggetto finale *myData* dispone di slot per i dati SNP, mappature e dati fenotipici. Nel caso in cui si abbiano dati su qualche altro fenotipo dei pazienti, è possibile creare il file fenotipo come una tabella avente come colonne i nomi dei campioni, il sesso e i tratti di interesse. Va notato che tutti gli ID utilizzati nel file .ped devono corrispondere a quelli del file .ph; in caso contrario, si verificherebbe un errore e i dati non verrebbero caricati. Ad esempio, se supponiamo di avere gli ID dei campioni nella variabile *samples*, i dati del sesso nella variabile *sex* e i dati del tratto di interesse nella variabile *trait*, possiamo registrare il file fenotipo myPheno.ph nel seguente modo:

```
> write.csv(data.frame(id=samples, sex=sex, traitOfInterest=trait), file="myPheno.ph")
```

12.5 Gestione dei dati con il pacchetto *GWASTools*

Sinora ci siamo occupati di due diversi formati di dati GWAS. In questa sezione introdurremo il pacchetto Bioconductor *GWASTools* (con il relativo dataset *GWASdata*) che non solo ci permette di lavorare con i dati grezzi, ma ci aiuta anche nei controlli di qualità e in molte fasi di analisi.

Iniziamo con l'installazione del pacchetto dal repository Bioconductor digitando i seguenti comandi:

```
> BiocManager::install("GWASTools")
> BiocManager::install(c("GWASdata", "gdsfmt"))
> library(GWASTools)
> library(GWASdata)
```

Ora carichiamo i dati grezzi *affy* dal pacchetto GWASdata (i due dataset "affY_scan_anot" e "affy_snp_annot"):

```
> data(affy_scan_annot)
> data(affy_snp_annot)
```

Utilizziamo i dati inclusi nel pacchetto. Per ottenere il percorso completo e il nome dei dati dal pacchetto, digitiamo i seguenti comandi:

```
> file <- system.file("extdata", "affy_geno.nc", package="GWASdata")
> file
[1] "/Library/Frameworks/R.framework/Versions/3.6/Resources/library/GWASdata/extdata/affy_geno.nc"
```

Usiamo il file come argomento della funzione `NcdfGenotypeReader()` per importare il file nell'area di lavoro R e controlliamo il numero di scansioni e SNP nei dati:

```
> nc <- NcdfGenotypeReader(file)
Loading required namespace: ncdf4
> nscan(nc)
[1] 47
> nsnp(nc)
[1] 3300
```

Per estrarre i componenti del genotipo dai dati, eseguiamo la funzione `getGenotype()` per le scansioni e gli SNP desiderati (nel nostro caso, 5 e 20):

```
> geno <- getGenotype(nc, scan=c(1,5), snp=c(1,20))
```

Carichiamo i dati di scansione e SNP nell'area di lavoro come segue:

```
> data(affyScanADF)
> data(affySnpADF)
```

Per recuperare i dati fenotipici, digitiamo il seguente comando:

```
> varMetadata(affySnpADF)

labelDescription
snpID
unique integer ID for SNPs
chromosome integer code for chromosome: 1:22=autosomes, 23=X, 24=pseudoautosomal,
25=Y, 26=Mitochondrial, 27=Unknown
position                                                                base pair
position on chromosome (build 36)
rsID
RS identifier
probeID
unique ID from Affymetrix
missing.n1       fraction of genotype calls missing over all samples except that
females are excluded for Y chr SNPs
```

Creiamo un oggetto dati con attributi scan e SNP mediante la funzione `GenotypeData()` (a questo livello il componente di annotazione per i dati non è comunque obbligatorio):

```
> myGenoData <- GenotypeData(nc, snpAnnot=affySnpADF, scanAnnot=affyScanADF)
```

Abbiamo creato un oggetto della classe "GenotypeData". Osserviamone la struttura:

```
> str(myGenoData)
Formal class 'GenotypeData' [package "GWASTools"] with 3 slots
  ..@ data     :Formal class 'NcdfGenotypeReader' [package "GWASTools"] with 14 slots
  .. .. ..@ snpDim       : chr "snp"
  .. .. ..@ scanDim      : chr "sample"
  .. .. ..@ snpIDvar     : chr "snp"
  .. .. ..@ chromosomeVar: chr "chromosome"
  .. .. ..@ positionVar  : chr "position"
  .. .. ..@ scanIDvar    : chr "sampleID"
  .. .. ..@ genotypeVar  : chr "genotype"
  .. .. ..@ autosomeCode : int [1:22] 1 2 3 4 5 6 7 8 9 10 ...
  .. .. ..@ XchromCode   : int 23
  .. .. ..@ YchromCode   : int 25
  .. .. ..@ XYchromCode  : int 24
  .. .. ..@ MchromCode   : int 26
  .. .. ..@ filename     : chr "/Library/Frameworks/R.framework/Versions/3.6/
Resources/library/GWASdata/extdata/affy_geno.nc"
…
```

Calcoliamo il *Missing Call Rate* (MCR = tasso di mancanti, la frazione di mancanti per ciascun SNP sui campioni o la frazione di mancanti per campione sugli SNP) per tutti i cromosomi come segue:

```
> mcr_chr <- missingGenotypeByScanChrom(myGenoData)
> head(mcr_chr$missing.counts)
   21 22 X XY   Y M
3   2  4 2  0 100 0
5   0  4 2  0   0 0
14  2  6 2  1 100 0
15  4  3 6  1 100 0
17  2  4 1  0   0 1
28  1  1 0  0   0 0
```

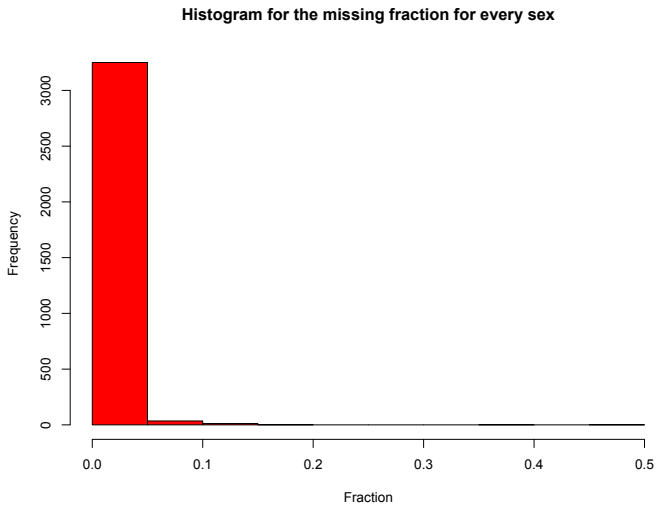
Per calcolare il MCR del sesso nella genotipizzazione come misura della qualità, utilizziamo la seguente funzione:

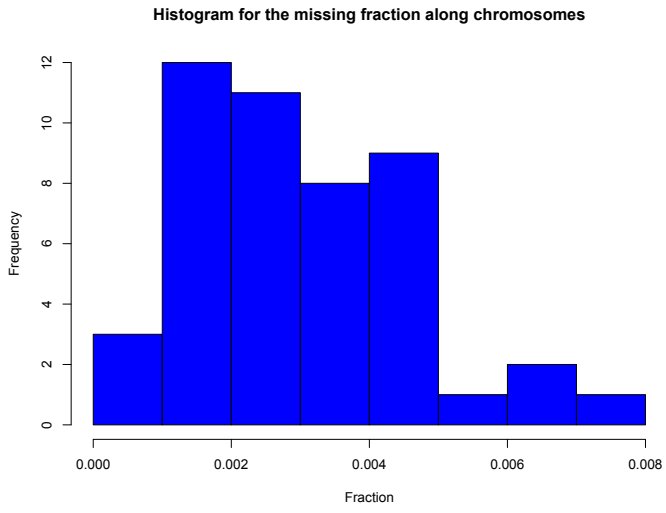
```
> mcr_sex <- missingGenotypeBySnpSex(myGenoData)
```

La funzione `NcdfGenotypeReader()` legge i dati del genotipo memorizzati nei file NetCDF (un file NetCDF memorizza i dati del genotipo e dell'intensità e ha dimensioni pari a "N.ro di SNP × Dimensione del campione"); a questo scopo utilizza il pacchetto *netcdf*. Una volta che i dati sono letti dal file NetCDF, possono essere usati per creare un oggetto dati completo con cui lavorare. Il principio in questione è lo stesso di un ExpressionSet che abbiamo visto nell'analisi dell'espressione genica nel capitolo 11.

I genotipi mancanti possono influenzare l'intero dataset verso un genotipo o un altro. Un *Missing Call Rate* (MCR) è definito come la frazione di SNP mancanti su più campioni (o per campione). È una delle misure di qualità comuni per i blocchi di genotipi. La funzione `missingGenotypeBySnpSex()` calcola gli SNP mancanti, gli alleli e gli eterozigoti per ogni sesso. Si noti che la funzione esclude le femmine quando si calcola il tasso di SNP mancanti per il cromosoma Y. Con questi risultati, possiamo ottenere il totale dei mancanti, il numero di scansioni e la frazione mancante. Possiamo eseguire un istogramma delle frazioni utilizzando i seguenti comandi:

```
> par(mfrow=c(1,2))
> hist(mcr_sex$missing.fraction, xlab="Fraction", main="Histogram for the missing
fraction for every sex", col="red")
> hist(mcr_chr$missing.fraction, xlab="Fraction", main="Histogram for the missing
fraction along chromosomes", col="blue")
```





La figura precedente mostra l'istogramma delle frazioni di mancanti per ogni sesso (a sinistra) e per tutti i cromosomi (a destra) (si noti che abbiamo la frazione di mancanti lungo l'asse *x*; l'asse *y* rappresenta la frequenza dell'occorrenza nei dati).

Per calcolare il numero di SNP con MCR superiore a 0,05, possiamo usare il seguente comando:

```
> sum(mcr_sex$missing.fraction>0.05)
[1] 50
```

12.6 Manipolazione di altri formati di dati GWAS

Finora abbiamo lavorato con dati strutturati (data.frame) e file PLINK (.ped e .map). Tuttavia, ci sono diversi formati per i dati GWAS, con l'utilizzo di diverse librerie per l'analisi dei differenti formati. Questa sezione si focalizza sulla lettura di altri formati di dati e sulla loro conversione in formati utilizzabili. Tali dati provengono da esperimenti su diverse piattaforme (Affymetrix, Illumina, …). Nel seguito utilizzeremo solo dei nomi di riferimento per i file; è possibile modificare tali nomi con i propri per la concreta applicazione degli esempi.

Iniziamo con alcune funzioni per convertire file da diversi tipi di dati genotipici a dati grezzi (raw). La funzione "convert.snp.<formato desiderato>" può essere utilizzata per convertire i formati in file raw, come visto nella sezione precedente.

Digitare il seguente comando per convertire un file Affymetrix nel formato raw:

```
> dir <- "<path/to/affymetrix/directory>"
> library(GenABEL)
> convert.snp.affymetrix(dir, map, outfile, skipaffym)
```

La funzione `convert.snp.affymetrix()` converte i dati genotipici da Affymetrix in formato interno grezzo (raw). Il parametro `dir` specifica il percorso dove sono contenuti i file Affymetrix; `map` indica il nome del file con le annotazioni di mappatura degli SNP sui cromosomi; `outfile` è il file di destinazione; `skipaffym` indica il numero di righe iniziali del file Affymetrix da saltare.

Analogamente all'esempio precedente, utilizziamo le seguenti funzioni per convertire dati genotipici dal formato Illumina (.illumina), premakeped/mach (.ped) e intero (.text) nel formato raw:

```
> convert.snp.illumina(infile="file.illumina", out="myRaw.raw", strand="+")
> convert.snp.ped(ped="myPed.ped", mapfile="myMap.map", out="myRaw.raw")
> convert.snp.text("genos.dat", "myRaw.raw")
```

Consultare l'aiuto dei singoli comandi per i dettagli di utilizzo.

Per leggere un file .ped come data.frame, utilizzare la funzione generica `read.csv()` o la funzione `read.pedfile()` del pacchetto Bioconductor *trio*:

```
> library(trio)
> mySNP <- read.pedfile(file="myPed.ped")
```

Esaminiamo le righe iniziali del data.frame creato:

```
> head(mySNP)
  famid     pid fatid motid sex affected SNP1.1 SNP1.2 SNP2.1 SNP2.2 SNP3.1 SNP3.2 SNP4.1
1     1 NA06985     0     0   2        2      1      2      1      1      1      1      1
2     1 NA06991     0     0   2        1      2      2      1      1      1      1      1
3     1 NA06993     0     0   2        1      2      2      1      1      1      1      1
4     1 NA06994     0     0   1        2      1      1      1      1      1      1      1
5     1 NA07000     0     0   2        1      2      2      1      1      1      1      1
6     1 NA07019     0     0   2        2      1      2      1      1      1      1      1
```

Controlliamo anche i nomi delle colonne dell'oggetto *mySNP* per verificare la differenza tra questo data.frame e quello che abbiamo usato col pacchetto SNPassoc (vedi sezione 12.1):

```
> colnames(mySNP)
  [1] "famid"    "pid"      "fatid"    "motid"    "sex"      "affected" "SNP1.1"   "SNP1.2"
  [9] "SNP2.1"   "SNP2.2"   "SNP3.1"   "SNP3.2"   "SNP4.1"   "SNP4.2"   "SNP5.1"   "SNP5.2"
. . .
[425] "SNP210.1" "SNP210.2" "SNP211.1" "SNP211.2" "SNP212.1" "SNP212.2" "SNP213.1" "SNP213.2"
[433] "SNP214.1" "SNP214.2"
```

I dati del file "myPed.ped" letti nella variabile m*ySNP* devono essere convertiti in un oggetto data.frame utilizzabile dalla funzione del pacchetto *SNPassoc*. Per fare questo, estraiamo alcune caratteristiche dalla matrice *mySNP* come segue:

```
> allsnps <- unlist(strsplit(colnames(mySNP)[7:ncol(mySNP)],"[.]"))[seq(1,68,2)]
> onlysnp <- mySNP [,-c(1:6)]
> odds <- seq(1,ncol(onlysnp),2)
> allsnps <- allsnps[odds]
> evens <- odds+1
```

Una volta che disponiamo di tutte le dimensioni e le specifiche dei dati .ped originali, utilizziamo i seguenti comandi per creare un data.frame vuoto che corrisponda alle dimensioni richieste:

```
> x <- matrix(0, nrow(onlysnp), length(allsnps))
> x <- data.frame(x)
> colnames(x) <- allsnps
```

Utilizziamo ora i seguenti comandi per riempire il data.frame *x* con gli alleli SNP:

```
> for (i in 1:length(odds)) {
+ p <- as.factor(paste(onlysnp[,odds[i]], onlysnp[,evens[i]], sep=""))
+ x[,i] <- p
+ }
```

Infine, uniamo il data.frame *x* con gli ID e le altre informazioni variabili del file .ped originale e controlliamo i dati riformattati come segue:

```
> mySNP <- data.frame(mySNP[, c(1:6)],x)
> head(mySNP)
  famid     pid fatid motid sex affected SNP1 SNP2 SNP3 SNP4 SNP5 SNP6 SNP7 SNP8 SNP9 SNP10 …
1     1 NA06985     0     0   2        2   12   11   11   11   12   22   12   12   22    11
2     1 NA06991     0     0   2        1   22   11   11   11   11   22   22   11   22    11
3     1 NA06993     0     0   2        1   22   11   11   11   11   22   22   11   22    11
4     1 NA06994     0     0   1        2   11   11   11   11   22   22   11   22   22    11
5     1 NA07000     0     0   2        1   22   11   11   11   11   22   22   11   22    11
6     1 NA07019     0     0   2        2   12   11   11   11   22   22   11   22   22    11
…
```

Questo data.frame può essere utilizzato per l'ulteriore elaborazione con il pacchetto *SNPassoc* come abbiamo visto in precedenza.

Le funzioni di conversione mostrate in questa sezione leggono le informazioni genotipiche del file che è stato assegnato e convertono le informazioni per il pacchetto *GenABEL* nel formato grezzo interno (raw). Il formato è composto da diverse componenti, strutturate come data.frame.

Nella seconda parte della sezione abbiamo visto come convertire un file .ped da PLINK in data.frame. Il file .ped può essere letto come oggetto data.frame in molti modi, anche con la funzione generica `read.csv()` oppure con la funzione specifica `read.pedfile()`. La seguente schermata mostra la parte iniziale del file .ped prima della conversione:

	famid	pid	fatid	motid	sex	affected	SNP1.1	SNP1.2	SNP2.1	SNP2.2	SNP3.1	SNP3.2	SNP4.1
1	CH18526	NA18526	0	0	2	1	G	G	C	C	T	T	A
2	CH18524	NA18524	0	0	1	1	G	G	C	C	T	T	A
3	CH18529	NA18529	0	0	2	1	C	G	C	C	T	T	C
4	CH18558	NA18558	0	0	1	1	G	G	C	C	G	T	A
5	CH18532	NA18532	0	0	2	1	G	G	C	C	T	T	A
6	CH18561	NA18561	0	0	1	1	G	G	G	C	G	T	C
7	CH18562	NA18562	0	0	1	1	G	G	C	C	T	T	A
8	CH18537	NA18537	0	0	2	2	G	G	G	C	G	T	C
9	CH18603	NA18603	0	0	1	2	G	G	C	C	T	T	A
10	CH18540	NA18540	0	0	2	1	G	G	C	C	T	T	A

Per rendere questo data.frame utilizzabile nel pacchetto *SNPassoc* abbiamo eseguito alcune manipolazioni, che si limitano a fondere gli alleli in due colonne consecutive come genotipi per ogni SNP. Infine, tutte le altre informazioni sono state fuse in un unico data.frame per essere completate. La seguente schermata mostra la parte iniziale del file .ped convertito:

	famid	pid	fatid	motid	sex	affected	SNP1	SNP2	SNP3	SNP4	SNP5	SNP6	SNP7	SNP8	SNP9	SNP10
1	CH18526	NA18526	0	0	2	1	GG	CC	TT	AA	GG	GG	TA	GG	TG	CC
2	CH18524	NA18524	0	0	1	1	GG	CC	TT	AA	GG	AG	AA	GA	GG	CC
3	CH18529	NA18529	0	0	2	1	CG	CC	TT	CA	GG	GG	TA	GG	TG	CC
4	CH18558	NA18558	0	0	1	1	GG	CC	GT	AA	GG	GG	AA	GA	TG	CC
5	CH18532	NA18532	0	0	2	1	GG	CC	TT	AA	GG	GG	AA	GA	GG	CC
6	CH18561	NA18561	0	0	1	1	GG	GC	GT	CA	GG	AG	TA	GG	TG	CC
7	CH18562	NA18562	0	0	1	1	GG	CC	TT	AA	GG	GG	AA	GA	GG	CC
8	CH18537	NA18537	0	0	2	2	GG	GC	GT	CA	GG	GG	AA	AA	GG	CC
9	CH18603	NA18603	0	0	1	2	GG	CC	TT	AA	GG	GG	TA	GA	TG	CC
10	CH18540	NA18540	0	0	2	1	GG	CC	TT	AA	GG	GG	TA	GG	TG	CC

Una volta che questo data.frame è pronto, si può eseguire la funzione `setupSNP()` e preparare i dati per le prove di associazione.

12.7 Test dei dati per l'equilibrio di Hardy-Weinberg

L'equilibrio di Hardy-Weinberg (HWE) è un principio che afferma che la variazione genetica in una popolazione rimarrà costante da una generazione all'altra in assenza di fattori di disturbo. Quando si verifica un processo di accoppiamento casuale in una popolazione numerosa senza fattori di disturbo, la legge prevede che sia il genotipo che le frequenze alleliche resteranno costanti perché sono in equilibrio. Tuttavia,

le mutazioni, la selezione naturale, l'accoppiamento non casuale, la deriva genetica e il flusso genico disturbano questo equilibrio. Qualsiasi deviazione da questo equilibrio indicherà quindi questi fenomeni, oltre alla qualità della genotipizzazione. Pertanto, l'esecuzione di un test HWE può essere utile durante l'analisi dei dati genotipici. I test che illustriamo in questa sezione per tutti i genotipi SNP o una loro selezione indicano se la deviazione dall'HWE è significativa e quindi indicativa di uno dei fenomeni qui menzionati. La verifica è di solito realizzata con il test esatto di Fisher o il Chi-quadro.

Eseguiremo il test HWE per due diversi tipi di dati: prima per dati simulati; successivamente per dati SNP sull'Alzheimer (disponibili per il download al seguente indirizzo: Alzheimers.rda). Questi ultimi indicano che la malattia di Alzheimer ad insorgenza tardiva è prevalente in una popolazione giapponese e che un certo numero di SNP sono stati genotipizzati nella regione del gene APOE; questi SNP sono elencati in ordine fisico, con la distanza tra il primo e l'ultimo SNP di circa 55Kb.

Per prima cosa, carichiamo la libreria *GWASExactHW* nella sessione R:

```
> install.packages("GWASExactHW")
> library(GWASExactHW)
```

Simuliamo ora alcuni dati generando artificialmente delle varianti SNP attraverso una distribuzione multinomiale. Utilizziamo le condizioni di HWE per generare i dati come segue:

```
> pA <- runif(1)  # a random number between 0 and 1
> pAA <- pA^2
> pAa <- 2*pA*(1-pA)
> paa <- (1-pA)^2
> myCounts <- rmultinom(100, 500, c(pAA, pAa, paa))  # generate 100 multinomially distributed random number columns (each summing to 500) for each of the three probabilities pAA, pAa, paa (summing to 1)
```

Creiamo un oggetto data.frame di dimensione 100×3 che contiene i conteggi (frequenze) di tutte le varianti:

```
> genotypes <- data.frame(t(myCounts))
> colnames(genotypes)= c("nAA", "nAa", "naa")
```

Utilizziamo infine la funzione `HWExact()` per calcolare i *p*-value dei 100 genotipi artificiali come segue:

```
> hwPvalues <- HWExact(genotypes)
```

Carichiamo ora nella sessione di lavoro i dati SNP reali (che possono essere scaricati dall'indirizzo Alzheimers.rda); viene caricato l'oggetto data.frame *Alzheimers* che contiene le frequenze di ciascuno dei 3 alleli AA, Aa, aa (nelle colonne) per 70 SNP (nelle righe) come segue:

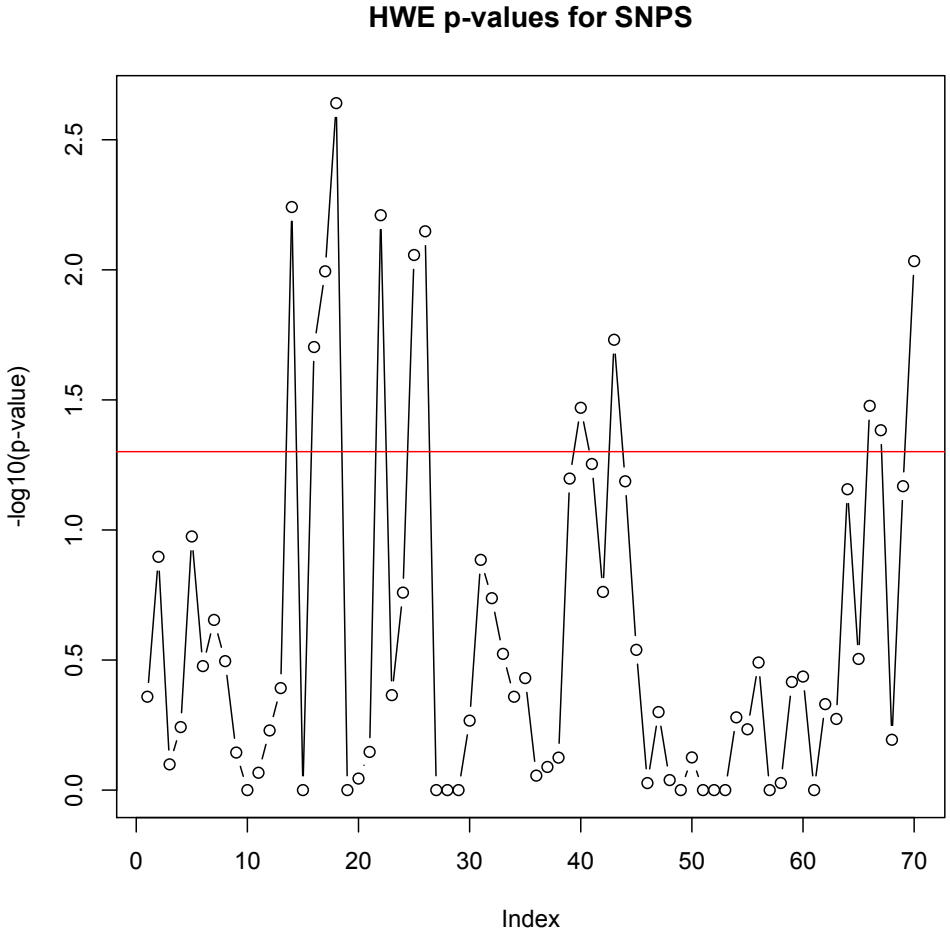
```
> load("Alzheimers.rda")
> head(Alzheimers)
                nAA nAa naa
rs419010.cases  112 278 150
rs394221.cases  149 269  91
rs4803766.cases 136 273 130
rs395908.cases  228 240  70
rs519113.cases  237 226  73
rs12776.cases   245 230  65
```

Utilizziamo i dati nella variabile *Alzheimers* per calcolare tramite il test esatto di Fisher i 70 *p*-value:

```
> myTest <- HWExact(Alzheimers)
> names(myTest) <- rownames(Alzheimers)
> head(myTest)
 rs419010.cases  rs394221.cases rs4803766.cases  rs395908.cases  rs519113.cases  rs12776.cases
      0.4375541       0.1268150       0.7963606       0.5724517       0.1059180      0.3336875
```

Tracciamo i *p*-value ottenuti dal test:

```
> plot(-log10(myTest), type="b", ylab="-log10(p-value)", main="HWE p-values for SNPS")
> abline(h=-log10(0.05), col="red")
```



Per conoscere il numero degli SNP significativi conteggiamo i *p*-value inferiori a 0,05 (12, nella parte superiore del grafico precedente):

```
> sum(myTest<0.05)
[1] 12
```

Visualizziamo i nomi dei 12 SNP significativi:

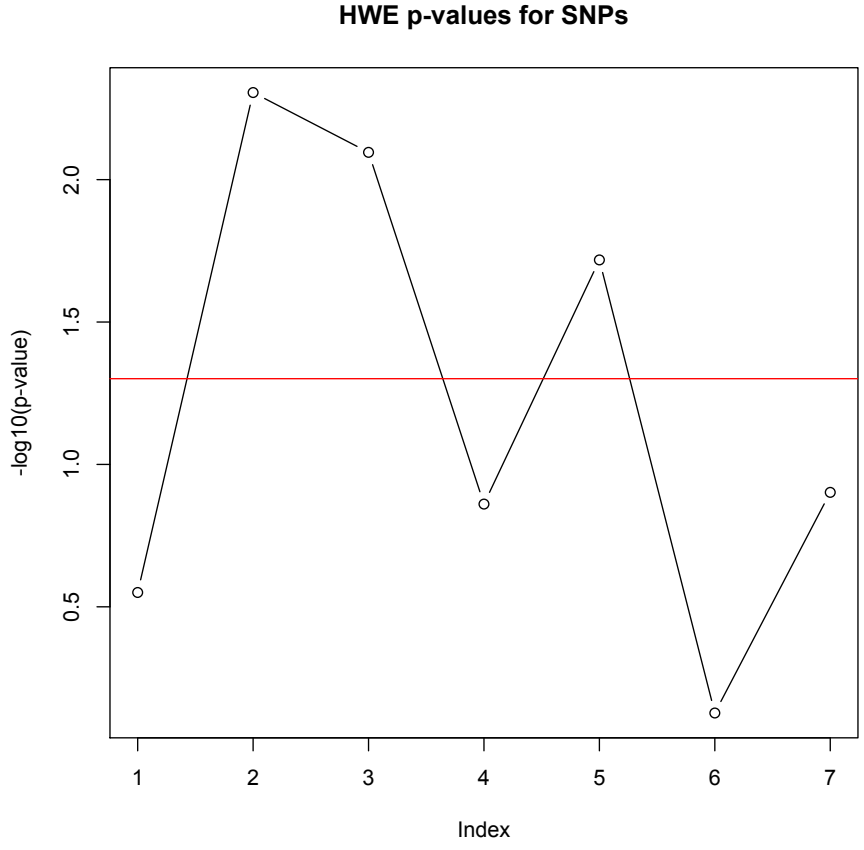
```
> names(myTest)[which(myTest<0.05)]
 [1] "rs157583.cases"     "rs157587.cases"     "rs283817.cases"     "rs573199.cases"
 [5] "rs394819.cases"     "rs446037.cases"     "rs434132.cases"     "rs519113.controls"
 [9] "rs3852860.controls" "rs429358.controls"  "rs7256200.controls" "rs4420638.controls"
```

Immaginiamo ora di non avere le frequenze, ma piuttosto gli alleli di SNP in diversi campioni (individui) e richiamiamo i dati SNP della sezione 12.1. La funzione `SNPfreqCalc()` (vedi Appendice) calcola le frequenze a partire dai dati SNP come segue:

```
> library(SNPassoc)
> data(SNPs)
> freq2 <- SNPfreqCalc(SNPs, prefix="snp")
> freq2
               nAA      nAa      naa
snp10001    12.000    53.00    92.00
snp10002     5.000    78.00    74.00
snp10005     3.000    70.00    84.00
snp10008   104.000    44.00     9.00
snp100011    1.000     2.00   154.00
snp100019   32.000    75.00    50.00
snp100020    9.000    43.00   105.00
```

In questo modo otteniamo la tabella di frequenza *freq2*, sulla quale eseguiamo la funzione `HWExact()` per ottenere le statistiche desiderate:

```
> myTestSNPs <- HWExact(freq2)
> names(myTestSNPs) <- rownames(freq2)
> myTestSNPs
   snp10001    snp10002    snp10005    snp10008   snp100011   snp100019   snp100020
0.281639248 0.004944837 0.008019904 0.137802395 0.019138510 0.746284208 0.125354659
> sum(myTestSNPs<0.05)
[1] 3
> plot(-log10(myTestSNPs), type="b", ylab="-log10(p-value)",
      main="HWE p-values for SNPs")
> abline(h=-log10(0.05), col="red")
```

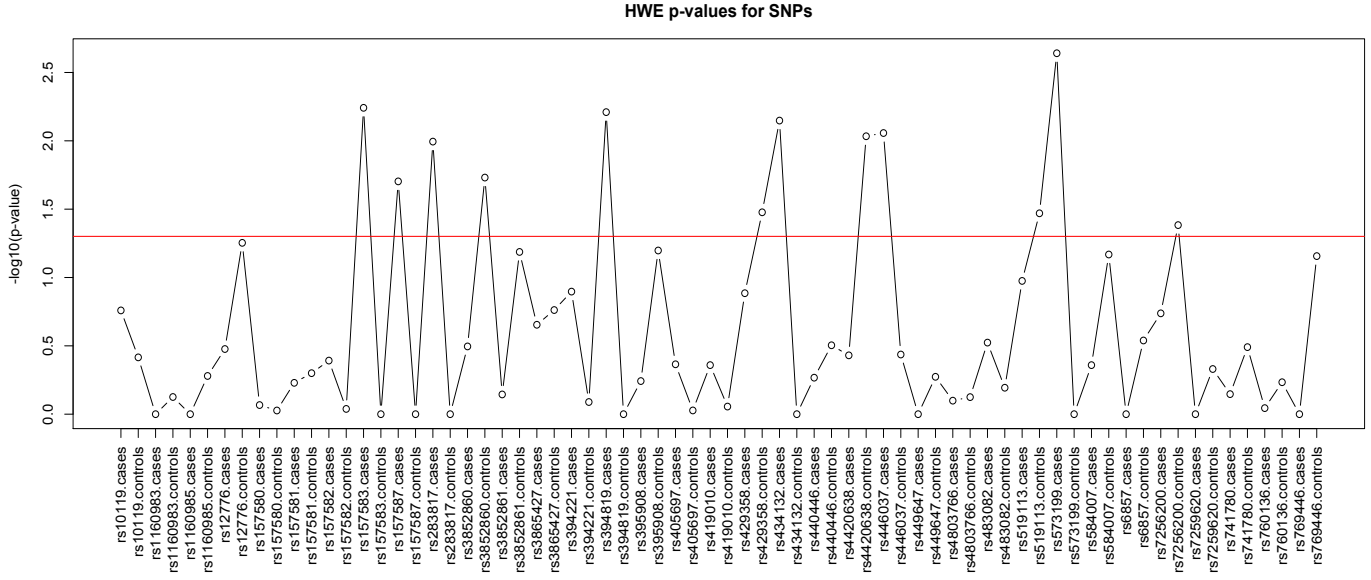


Come abbiamo visto, la funzione `HWExact()` esegue un test Fisher esatto su tutti gli SNP dei dati e restituisce i *p*-value corrispondenti. Nel primo caso dei dati simulati, abbiamo ottenuto delle probabilità casuali per le varianti utilizzandole quindi per generare i dati da una distribuzione multinomiale. Abbiamo infine calcolato la tabella di frequenza per le varianti ottenute, che funge da input per la funzione `HWExact()`.

Nel secondo caso abbiamo utilizzato il dataset dell'Alzheimer con frequenze alleliche per ogni SNP. La funzione `HWExact()`, applicata ai dati, calcola i punteggi esatti per le frequenze alleliche e restituisce un *p*-value per ogni SNP. Abbiamo poi tracciato i *p*-value ottenuti su una scala logaritmica per evidenziare gli SNP significativamente devianti secondo il test HWE. È noto che, nel caso dell'Alzheimer, la regione del gene APOE[6] gioca un ruolo importante. I dati *Alzheimers* che abbiamo utilizzato hanno gli SNP da "rs446037" a "rs429358" che si trovano nel gene APOE. Gli SNP "rs429358" e "rs7412" (non genotipizzati, ma adiacenti a "rs429358") formano insieme gli alleli ε2, ε3 e ε4 del gene APOE. I risultati ottenuti mostrano gli SNP che si discostano dall'HWE.

Per il terzo tipo di dati, abbiamo utilizzato la funzione `SNPfreqCalc()` che calcola una tabella di frequenza dai file SNP che contengono gli alleli. La funzione calcola la frequenza di ogni allele per ogni SNP in tutti i campioni (individui) e infine rimuove le informazioni fenotipiche disponibili nei dati cercando uno schema di nomi fornito per gli ID SNP (abbiamo utilizzato "snp" come schema, fornendolo come valore all'argomento `prefix` della funzione; avremmo tuttavia potuto usare altri pattern, come ad es. "rs", per identificare gli SNP).

La seguente schermata mostra il grafico dei *p*-value ottenuti per gli SNP dell'Alzheimer dopo il test HWE applicando il pattern "rs" (lungo l'asse *x* abbiamo gli SNP ordinati e raggruppati per caso/controllo; l'asse *y* rappresenta il -$\log_{10}$ dei *p*-value):



[6] Le persone con la variante genica *APOE4* hanno livelli più alti di proteine amiloidi, e sono a più elevato rischio di Alzheimer. Questa variante infatti altera il metabolismo dei lipidi nei neuroni e negli astrociti (le cellule cerebrali che alimentano i neuroni), e riduce notevolmente l'attività della microglia, la cui principale funzione è la rimozione di presenze indesiderate, dai patogeni alle proteine amiloidi di scarto.

12.8 Test di associazione con dati CNV

La variazione del numero di copie (CNV, *Copy Number Variation*) si riferisce ad un fenomeno in cui per alcuni geni esiste una normale variazione del numero di copie di una o più sezioni del DNA. I CNV sono una componente importante della variazione strutturale del genoma umano e sono potenzialmente critici in termini di contributo genetico al rischio di malattie complesse comuni. Proprio come l'analisi di associazione basata sull'SNP che abbiamo visto finora, l'analisi di associazione CNV può svolgere un ruolo cruciale nella comprensione di molti fenotipi e malattie complesse. In questa sezione vedremo alcuni metodi per eseguire tale analisi.

Utilizzeremo il pacchetto *CNVassoc* disponibile nel repository di archivio del CRAN (https://cran.r-project.org/src/contrib/Archive/CNVassoc/). I dati di input per la nostra analisi sono quelli dell'esperimento MLPA per l'associazione tra CNV e malattia integrati nel pacchetto. Iniziamo con il caricamento della libreria *CNVassoc* e dei relativi dati:

```
> library(CNVassoc)
> data(dataMLPA)
```

Diamo un'occhiata al data.frame *dataMLPA*:

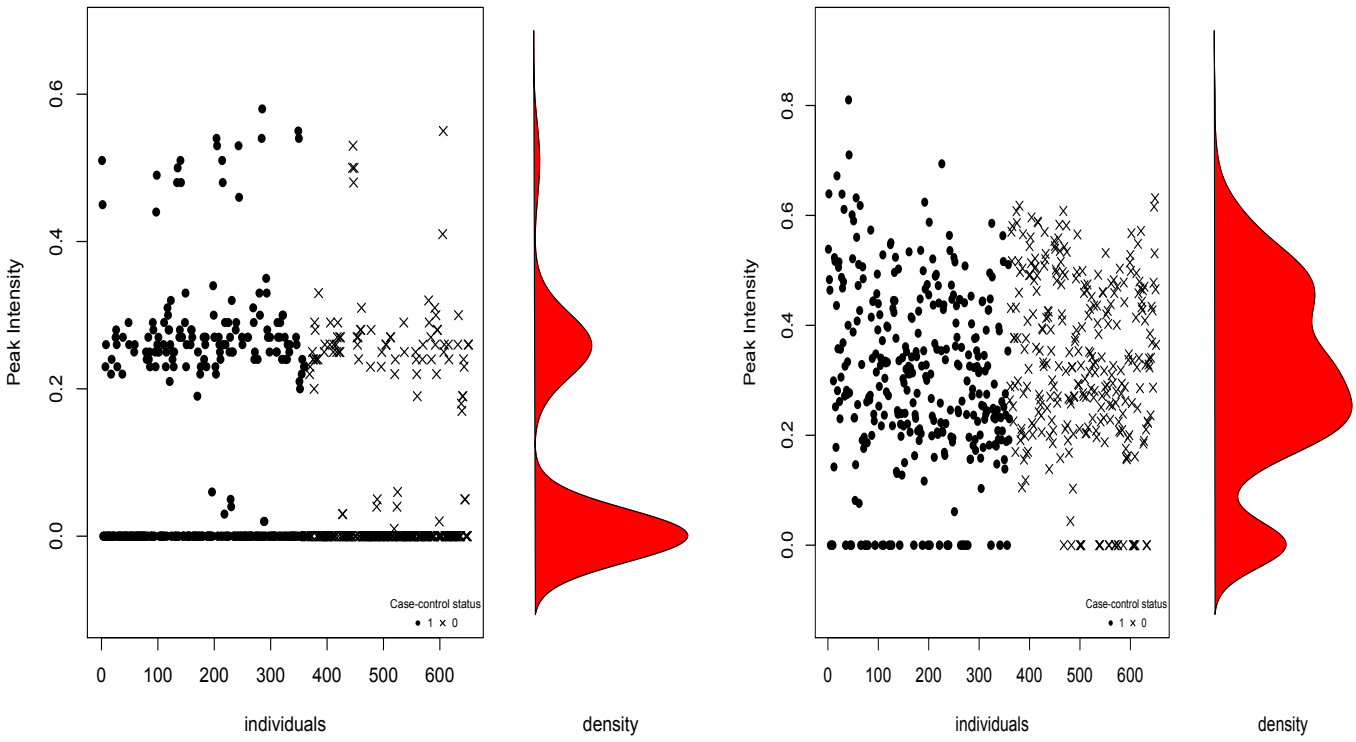
```
> str(dataMLPA)
'data.frame':     651 obs. of  8 variables:
 $ id       : Factor w/ 346 levels "H238","H239",..: 1 1 2 2 3 3 4 4 5 5 ...
 $ casco    : int  1 1 1 1 1 1 1 1 1 1 ...
 $ Gene1    : num  0.51 0.45 0 0 0 0 0.23 0.26 0 0 ...
 $ Gene2    : num  0.539 0.639 0.483 0.464 0 ...
 $ PCR.Gene1: Factor w/ 3 levels "del","ht","wt": 3 3 1 1 1 1 2 2 1 1 ...
 $ PCR.Gene2: Factor w/ 3 levels "del","ht","wt": 3 3 3 3 1 1 1 1 1 1 ...
 $ quanti   : num  -0.61 -0.13 -0.57 -1.4 0.83 -2.07 -1.68 -1.4 1.09 0.55 ...
 $ cov      : num  10.83 10.69 9.63 9.87 10.25 ...
> head(dataMLPA)
    id casco Gene1     Gene2 PCR.Gene1 PCR.Gene2 quanti   cov
1 H238     1  0.51 0.5385080        wt        wt  -0.61 10.83
2 H238     1  0.45 0.6392029        wt        wt  -0.13 10.69
3 H239     1  0.00 0.4831572       del        wt  -0.57  9.63
4 H239     1  0.00 0.4640072       del        wt  -1.40  9.87
5 H276     1  0.00 0.0000000       del       del   0.83 10.25
6 H276     1  0.00 0.0000000       del       del  -2.07 10.40
```

Per verificare le intensità che corrispondono al fenotipo, tracciamo le densità per entrambi i geni (gene 1 a sinistra, gene 2 a destra; vedi grafici pagina seguente):

```
> plotSignal(dataMLPA$Gene1, case.control = dataMLPA$casco)
> plotSignal(dataMLPA$Gene2, case.control = dataMLPA$casco)
```

Ora, dopo aver esaminato le intensità di picco dei dati, è necessario creare un oggetto CNV per il gene che si vuole studiare (il gene 2); creiamo l'oggetto CNV come segue:

```
> myCNV <- cnv(x = dataMLPA$Gene2, threshold.0 = 0.01, mix.method = "mixdist")
```



Utilizziamo l'oggetto CNV per calcolare l'associazione mediante la funzione `CNVassoc()`:

```
> myModel <- CNVassoc(formula=casco ~ myCNV, data = dataMLPA, model = "mul")
```

Per esaminare i parametri stimati, digitiamo il seguente comando:

```
> summary(myModel)

Call:
CNVassoc(formula = casco ~ myCNV, data = dataMLPA, model = "mul")

Deviance: 876.396
Number of parameters: 3
Number of individuals: 651

Coefficients:
          OR lower.lim upper.lim      SE    stat pvalue
CNV0  1.0000
CNV1  0.4772    0.2742    0.8304  0.2827 -2.6172  0.009
CNV2  0.3169    0.1834    0.5477  0.2791 -4.1169  0.000

(Dispersion parameter for  binomial  family taken to be  1 )

Covariance between coefficients:
     CNV0    CNV1    CNV2
CNV0  0.0613  0.0000  0.0000
CNV1          0.0186 -0.0032
CNV2                  0.0166
```

Per fare un'ANOVA (con un modello additivo) o il calcolo della probabilità, utilizziamo la funzione corrispondente:

```
> myModel2 <- CNVassoc(formula=casco ~ myCNV, data = dataMLPA, model= “add")
> anova(myModel, myModel2)

--- Likelihood ratio test comparing 2 CNVassoc models:

Model 1 call:  CNVassoc(formula = casco ~ myCNV, data = dataMLPA, model = "mul")
Model 2 call:  CNVassoc(formula = casco ~ myCNV, data = dataMLPA, model = "add")
Chi= 0.6645798 (df= 1 )  p-value= 0.4149477
  Note: the 2 models must be nested, and this function doesn't check this!
> logLik(myModel)
  logLik       df
-438.198    3.000
```

Infine, per determinare se un CNV è associato al fenotipo, eseguiamo un test di Wald o di probabilità sul modello (che restituisce il *p*-value per l'associazione in base al test che è stato eseguito):

```
> CNVtest(myModel , type = "LRT")
----CNV Likelihood Ratio Test----
Chi= 18.75453 (df= 2 ) , pvalue= 8.462633e-05

> CNVtest(myModel, type = "Wald")
----CNV Wald test----
Chi= 17.32966 (df= 2 ) , pvalue= 0.0001725492
```

Il dataset MLPA contiene lo stato caso/controllo tra il CNV e la malattia. I dati contengono le intensità di picco per i due geni che emergono dal test MLPA su 360 casi e 291 controlli. I dati contengono i due geni, la fase di studio e le covariate corrispondenti. La seguente schermata ne mostra la parte iniziale:

```
   id casco Gene1     Gene2 PCR.Gene1 PCR.Gene2 quanti   cov
1 H238     1  0.51 0.5385080        wt        wt  -0.61 10.83
2 H238     1  0.45 0.6392029        wt        wt  -0.13 10.69
3 H239     1  0.00 0.4831572       del        wt  -0.57  9.63
4 H239     1  0.00 0.4640072       del        wt  -1.40  9.87
5 H276     1  0.00 0.0000000       del       del   0.83 10.25
6 H276     1  0.00 0.0000000       del       del  -2.07 10.40
```

La funzione `cnv()` utilizza il segnale quantitativo delle singole sonde per dedurre lo stato del numero di copie dei campioni. La funzione si basa sulle ipotesi del modello normale di mixture.

I grafici all'inizio della pagina precedente (ottenuti con la funzione `plotSignal()`) tracciano il segnale rispetto ai picchi di intensità nei dati CNV per i due geni. Lungo l'asse *x* abbiamo gli individui e lungo l'asse *y* l'intensità di picco. Ciascun grafico mostra i picchi per i casi e i controlli; le densità sono tracciate lungo l'asse verticale. Le intensità dei picchi mostrano le diverse densità per i diversi numeri di copie stimati.

La funzione `CNVassoc()` utilizza un modello di classe latente per eseguire un'analisi di associazione tra un CNV e il fenotipo; essa tratta il fenotipo come una variabile binaria. Durante l'utilizzo della funzione, il valore per l'argomento `family` dovrebbe essere basato sulla distribuzione della variabile di risposta. Infine, la funzione `CNVtest()` calcola il *p*-value di significato globale per l'associazione.

Si noti che i dati Illumina o Affymetrix, dove sono disponibili i rapporti $\log_2$ al posto delle intensità di picco, possono essere analizzati nello stesso modo che abbiamo illustrato in questa sezione. Come detto in

precedenza, la funzione `CNVassoc()` utilizza una classe latente per il test di associazione: questa classe latente incorpora incertezze che modellano il grado di errata classificazione dello stato del numero di copie. Maggiori informazioni sulla classe latente sono disponibili nell'articolo *Accounting for uncertainty when assessing association between copy number and disease: a latent class model* di Gonalez e altri su http://www.biomedcentral.com/1471-2105/10/172.

La funzione `multiCNVassoc()` può essere utilizzata per testare più CNV. Per saperne di più sulla funzione, digitare `?multiCNVassoc` nella console R.

Per eseguire un'analisi CNV a livello genomico, può essere utile il pacchetto *patchwork*. Può essere utilizzato per determinare il numero di copie di sequenze omologhe in tutto il genoma ed è utile per campioni di aneuploidia con copertura moderata. Per ulteriori dettagli, consultare l'articolo *Patchwork: allele-specific copy number analysis of whole-genome sequenced tumor tissue* di Mayrhofer e altri (http://genomebiology.com/content/14/3/R24).

12.9 Visualizzazioni negli studi GWAS

Per presentare i risultati ottenuti dall'analisi GWAS possiamo eseguire diverse visualizzazioni intuitive, rapide e interessanti. Le più comunemente utilizzate sono i grafici Q-Q (Quantile-Quantile), il grafico di Manhattan, i grafici delle associazioni locali e così via. Questa sezione presenta i metodi per generare alcuni di questi grafici.

Iniziamo con un **grafico Q-Q**. Per ottenerlo carichiamo prima la libreria *GWASTools*:

```
> library(GWASTools)
```

Come input prendiamo i risultati dell'associazione sui dati SNP dell'intero genoma dalla sezione precedente, che ricalcoliamo per comodità:

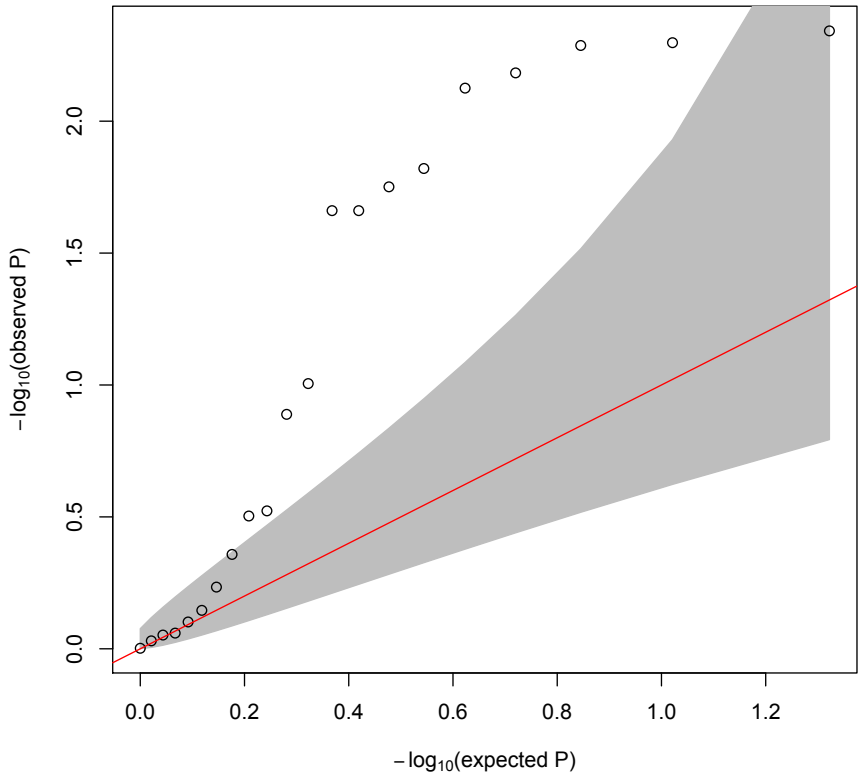
```
> mySNP <- setupSNP(SNPs, 6:40, sep="")
> myres <- WGassociation(protein, data=mySNP, model="all")
```

Estraiamo i *p*-value per il modello dominante di eredità dai risultati e utilizziamo la funzione `qqPlot()` per generare un grafico Q-Q (vedi inizio pagina seguente):

```
> pvals <- dominant(myres)
> qqplot(pvals)
```

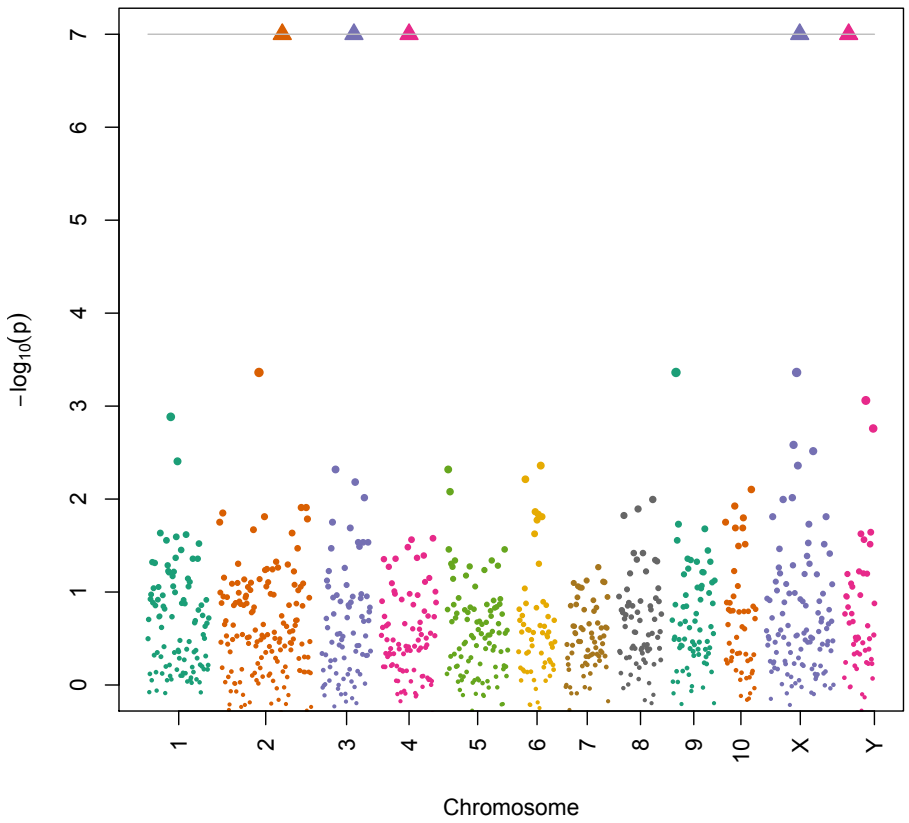
Il prossimo sarà un **grafico di Manhattan**. Iniziamo simulando i *p*-value di 1000 SNP e assegniamoli in maniera casuale ai cromosomi come segue:

```
> n <- 1000
> pvals <- sample(-log10((1:n)/n), n, replace=TRUE)
> chromosome <- c(rep(1,100), rep(2,150), rep(3,80), rep(4,90), rep(5,100),
rep(6,60), rep(7,70), rep(8,70), rep(9,70), rep(10,50), rep("X",110), rep("Y",50))
```



Per creare il grafico, richiamiamo la funzione `manhattanPlot()` dalla libreria *GWASTools*:

```
> manhattanPlot(pvals, chromosome)
```



È possibile creare un grafico simile con risultati reali. Allo scopo, riutilizziamo i risultati del dataset HapMap e iniziamo con l'estrazione dei *p*-value per il modello di ereditarietà dominante dai risultati dell'analisi dell'associazione SNP dell'intero genoma (trasformandoli nel logaritmo base 10 negativo):

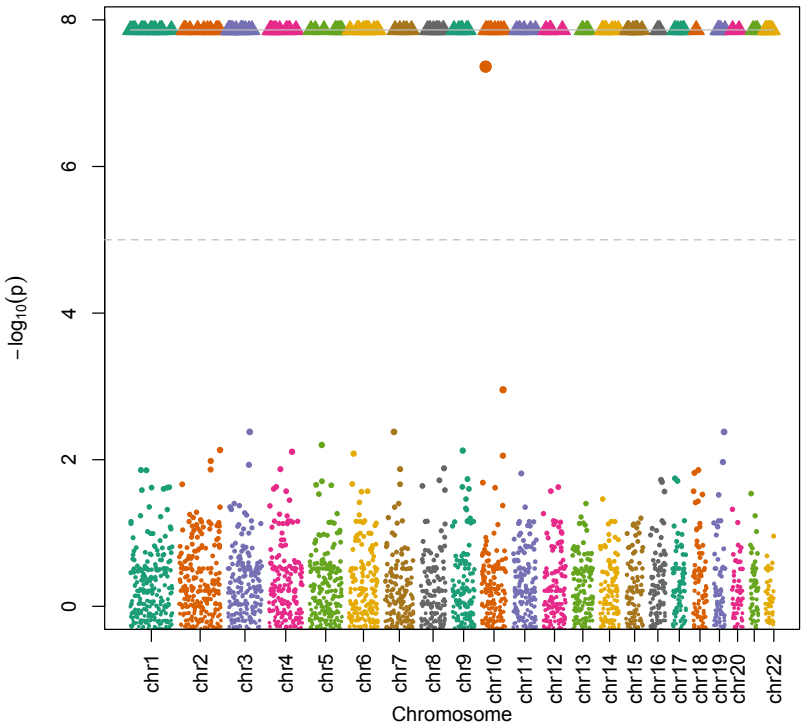
```
> data(HapMap)
> myHapMap <- setupSNP(HapMap, colSNPs= 3:9307, sort=TRUE,
      info=HapMap.SNPs.pos,sep="")
> myHapMapres <- WGassociation(group, data=myHapMap, model="dominant")
> pvals <- dominant(myHapMapres)
> pvals <- -log10(pvals)
```

Utilizziamo quindi i dati di posizione HapMap per ottenere informazioni sulla posizione cromosomica degli SNP:

```
> chromosome <- HapMap.SNPs.pos$chromosome
```

Come in precedenza, eseguiamo la funzione `manhattanPlot()` con questi *p*-value e le informazioni cromosomiche:

```
> manhattanPlot(pvals, chromosome, signif=1e-5)
```



Vediamo infine il grafico di associazione locale (**regional plot**). Per eseguirlo, carichiamo il pacchetto *gap* indieme ai dati CDKN:

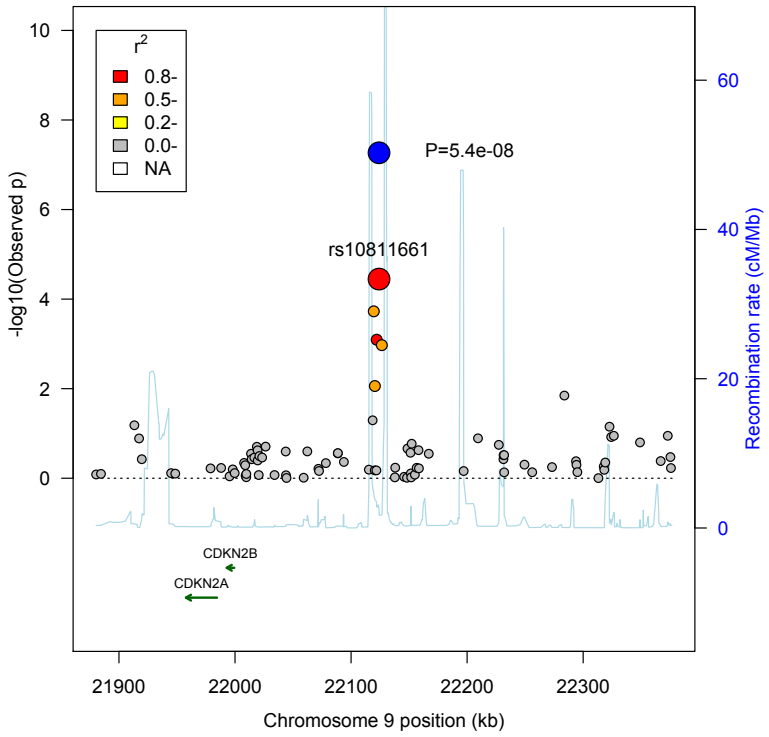
```
> install.packages(c("gap","gap.datasets"))
> library(gap)
> library(gap.datasets)
> data(CDKN)
```

La libreria *gap* carica il dataset CDKN, che ha tre subset (CDKNlocus, CDKNmap, CDKNgenes). Per esaminare la parte iniziale di ciascun subset, utilizziamo la funzione `head()`:

```
> head(CDKNlocus)
  CHR      POS       NAME       PVAL  RSQR
1   9 21880326  rs7865071 0.82418370 0.020
2   9 21884495  rs7389178 0.79907230 0.017
3   9 21913279 rs10811638 0.06569925 0.001
4   9 21917327  rs4977749 0.12945710 0.001
5   9 21919666  rs2518713 0.37621220 0.002
6   9 21944953 rs10757261 0.76992020 0.010
> head(CDKNmap)
       POS      THETA     DIST
1 19999135  0.1778173  0.00000
2 20000312  0.1791786 40.55383
3 20001576  6.9411662 40.55406
4 20001821 10.9959827 40.55576
5 20002125 11.3206207 40.55910
6 20002593  9.2593987 40.56440
> head(CDKNgenes)
      START      STOP STRAND    GENE
1 116267060 117256871      -   ASTN2
2  27938527  28709303      -  LINGO2
3   8307267   9008737      -   PTPRD
4  70379521  70966068      -   TRPM3
5 123221486 123771971      - DENND1A
6  98129920  98551034      -  GABBR2
```

L'oggetto "CDKNlocus" fornisce tutte le informazioni per creare il grafico. La funzione `asplot()` utilizza le informazioni di locus, mappa e gene di questi set di dati per creare un grafico di associazione locale (*regional association plot*) come segue:

```
> asplot(CDKNlocus, CDKNmap, CDKNgenes, best.pval=5.4e-8, sf=c(3,6))
```



Il primo grafico che abbiamo creato in questa sezione è una *grafico Q-Q* (Quantile-Quantile plot). I dati utilizzati provengono dagli studi di associazione della sezione precedente. L'obiettivo è quello di trovare un'associazione tra una certa proteina e il suo genotipo. I grafici Q-Q in GWAS mostrano la distribuzione prevista delle statistiche dei test di associazione (asse *x*) tra i milioni di SNP rispetto ai valori osservati (asse *y*). Una deviazione dalla diagonale $X = Y$ si riferisce ad una differenza costante tra i casi e i controlli su tutto il genoma. Pertanto, un grafico Q-Q può essere utilizzato per caratterizzare la misura in cui la distribuzione osservata della statistica del test segue la distribuzione attesa (nulla) nei dati forniti. Il grafico Q-Q per i dati SNP mostra i valori osservati lungo l'asse *y* e i valori attesi lungo l'asse *x* (statistiche del test); si noti che la linea rossa corrisponde a $X = Y$.

Il secondo tipo di grafico che abbiamo visto è il *grafico di Manhattan*. Abbiamo usato i risultati ottenuti analizzando i dati HapMap nella sezione per l'analisi dell'associazione SNP per l'intero genoma. Il grafico rappresenta la significatività dell'associazione degli SNP con il tratto o fenotipo in studio. L'asse *y* è la trasformazione -log10 dei *p*-value, dove i *p*-value rappresentano una misura dell'intensità di associazione. Il grafico è codificato a colori per cromosoma. Nel nostro caso, abbiamo creato prima un grafico con dati simulati, quindi abbiamo estratto i risultati dallo studio HapMap dell'intero genoma e abbiamo creato un secondo grafico dai dati risultanti.

Infine, il grafico di associazione locale (*regional association plot*) che abbiamo creato utilizza tre diversi data.frame: (*i*) l'SNP e i dettagli della sua associazione (CDKNlocus); (*ii*) la posizione e i dettagli della mappa dell'SNP (CDKNmap); (*iii*) i dettagli dei geni sul genoma (CDKNgenes). I dati provengono da uno studio delle regioni CDKN del nono cromosoma per studi di associazione provenienti da Studi di Genetica del Diabete. Il grafico risultante riguarda un particolare locus che si basa sulle informazioni relative ai tassi di ricombinazione, ai disequilibri di collegamento (LD) tra l'SNP di interesse e quelli vicini e ai *p*-value dei test di associazione single-point. Questo tipo di grafico (non molto comune) mostra un'annotazione per i *p*-value dei nomi SNP e la loro posizione sul cromosoma insieme alle misurazioni LD. La funzione `asplot()` accetta il locus, i geni, la mappa e i risultati delle analisi (dei punteggi statistici) per creare il grafico. Esaminiamo la struttura dei singoli oggetti usati come input per ottenere i dettagli del grafico della pagina precedente. L'asse *x* rappresenta il sito di localizzazione nel nono cromosoma in termini di kbasi, mentre l'asse *y* mostra il logaritmo negativo dei *p*-value osservati. Il tasso di ricombinazione è presente lungo l'asse *y* e mostra la struttura di disequilibrio del collegamento locale intorno all'SNP associato. Nel grafico i cerchi rappresentano gli SNP e il loro colore rappresenta l'indicatore statistico di significatività r^2.

13. Analisi dei dati di Spettrometria di Massa (MS)

La spettrometria di massa (MS) è una tecnologia chiave di analisi utilizzata nella proteomica attuale per generare grandi quantità di dati di alta qualità, affrontando i problemi di identificazione delle proteine, l'annotazione di modifiche secondarie e la determinazione della presenza delle proteine. I dati consistono nell'intensità rispetto al rapporto massa/carica (normalmente chiamato rapporto m/z) nel campione biochimico o chimico, mostrando il modello di distribuzione dei componenti nel campione (in questo capitolo ci occuperemo principalmente di campioni biochimici). Tra le molte applicazioni della MS, la scoperta di modelli di biomarcatore ha suscitato un notevole interesse.

Tuttavia, le sfide nell'analisi dei dati MS vanno da quelle statistiche (gestione del rumore, normalizzazione, ecc.) a quelle biologiche (identificazione peptidica, quantificazione, ecc.). Pertanto, il ruolo della bioinformatica diventa fondamentale per l'elaborazione dei dati di MS e per lo screening delle conoscenze biologiche nei dati. Questo capitolo si propone di presentare soluzioni ad alcuni problemi chiave nell'analisi dei dati sulla MS per gli studi di proteomica.

Per lavorare con i dati di MS, è estremamente importante conoscere i relativi formati. Nel corso degli anni, diversi produttori di spettrometri di massa hanno sviluppato diversi formati di dati per la gestione e la presentazione dei dati. Questo rende difficile avere un protocollo uniforme per manipolare direttamente i dati. Per ovviare a questa limitazione, diversi formati di dati aperti basati su XML sono stati recentemente sviluppati nell'ambito del "Trans-Proteomic Pipeline" (TPP, un software open source di analisi dei dati di proteomica prodotto dal gruppo di ricerca del Prof. Ruedi Aebersold dell'Institute for Systems Biology — ISB presso il Seattle Proteome Center) e come innovazione nel settore pubblico.

Il formato di dati utilizzato nella MS dipende in realtà dall'hardware utilizzato per produrre i dati. Tuttavia, c'è stato un recente sviluppo nel rendere questi formati più o meno coerenti per facilitare la manipolazione dei dati da parte della comunità scientifica. I due formati più usati per i dati MS sono mzXML e mzML.

Il formato mzXML è un formato di file basato su XML per dati di spettrometria di massa proteomica. Questo formato di dati ha informazioni che vanno dall'hardware all'elaborazione e ai picchi proteici. Negli ultimi tempi, mzXML si è affermato come formato diffuso grazie alla sua flessibilità e all'efficiente gestione dei diversi attributi dei dati. Maggiori dettagli sul formato possono essere trovati nell'articolo intitolato *A common open representation of mass spectrometry data and its application to proteomics research* di Pedrioli *et al*. L'articolo è disponibile all'indirizzo http://www.nature.com/nbt/journal/v22/n11/full/nbt1031.html.

Il formato mzML è un formato ibrido di mzXML e un formato obsoleto chiamato mzData. Si possono trovare maggiori dettagli sulle specifiche del formato nell'articolo *mzML: A single, unifying data format for mass spectrometer output* di Eric Deutsch all'indirizzo http://onlinelibrary.wiley.com/doi/10.1002/pmic.200890049/abstract.

Nella seguito ci concentreremo su questi due formati di dati, che sono popolari nella comunità di proteomica, e sui file binari disponibili all'interno dei pacchetti R. Tuttavia, daremo un'occhiata alle informazioni necessarie per lavorare anche con altri formati di dati MS.

13.1 Lettura dei dati MS nel formato mzXML/mzML

I dati disponibili nella comunità MS sono di solito sotto forma di file XML (un formato contenente diversi campi con le informazioni assegnate). Per iniziare a lavorare con i dati MS è necessario importarli nella sessione R: i seguenti passi illustrano l'approccio da adottare per importare dati MS basati su XML nella nostra sessione di lavoro.

Innanzitutto scarichiamo nella directory di lavoro R i dati (di circa 26MB) tramite un browser dall'indirizzo https://regis-web.systemsbiology.net/rawfiles/lcq/7MIX_STD_110802_1.mzXML. Sarà anche necessario caricare alcuni pacchetti per la gestione dei dati MALDI[7], che possiamo installare dal repository CRAN e caricare nella sessione R come segue:

```
> install.packages("readMzXmlData", dependencies = TRUE)
> install.packages("MALDIquant")
> install.packages("MALDIquantForeign")
> library(readMzXmlData)
> library(MALDIquant)
> library(MALDIquantForeign)
```

Utilizziamo ora la funzione `importMzXml()` per leggere il file mzXML scaricato (può anche essere fornita una directory contenente i file mzXML) nella variabile lista *myData*:

```
> myData <- importMzXml(path="7MIX_STD_110802_1.mzXML", centroided=TRUE)
```

Controlliamo la dimensione della lista *myData* ed esaminiamo la struttura della prima componente:

```
> length(myData)
[1] 7143
> head(summary(myData))
     Length Class     Mode
[1,]  705   MassPeaks S4
[2,]   74   MassPeaks S4
[3,]   91   MassPeaks S4
[4,]  686   MassPeaks S4
[5,]   44   MassPeaks S4
[6,]   92   MassPeaks S4
> dim(summary(myData))
[1] 7143    3
> str(myData[[1]])
Formal class 'MassPeaks' [package "MALDIquant"] with 4 slots
  ..@ snr      : num [1:705] NA NA NA NA NA NA NA NA NA NA ...
  ..@ mass     : num [1:705] 401 402 403 404 406 ...
  ..@ intensity: num [1:705] 722826 95972 1126969 607818 504474 ...
  ..@ metaData :List of 20
```

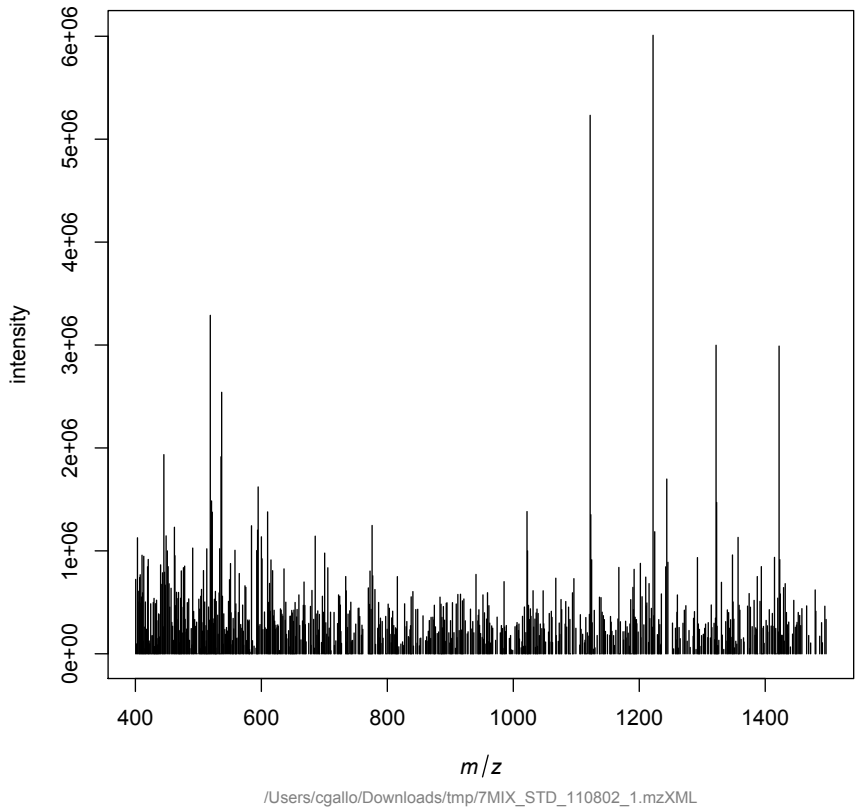
[7] MALDI è l'acronimo di Matrix-Assisted Laser Desorption/Ionization, una tecnologia che produce dati MS. Permette di analizzare grandi molecole organiche come le proteine.

Per poter leggere file di tipo mzML (ad es. i file contenuti nella directory "exampledata" del pacchetto *MALDIquantForeign*) utilizziamo la funzione `importMzMl()`:

```
> DIR <- system.file("exampledata", package="MALDIquantForeign")
> myDataMzMl <- importMzMl(path=DIR)
```

Tutti i formati di file MS discussi in questa sezione condividono le loro strutture con una serie di metadati, seguiti da una lista di spettri con le masse e le intensità. Inoltre, ogni spettro ha il proprio set di metadati, come il tempo di ritenzione e i parametri di acquisizione. Le masse e le intensità possono essere tracciate rispettivamente sugli assi *x* e *y* con la funzione `plot()`; il grafico mostra l'intensità (asse *y*) ottenuta per le diverse masse (asse *x*) fornendo un indizio dei picchi nei dati MS del file 7MIX_STD_110802_1.mzXML:

```
> plot(myData[[1]], col="black")
```



Il grafico mostra diversi frammenti con i loro corrispondenti rapporti m/z lungo l'asse *x* e le intensità osservate lungo l'asse *y*.

13.2 Lettura dei dati MS nel formato Bruker

I dati MS sono solitamente disponibili come file basati su XML, come visto in precedenza. Tuttavia, è possibile utilizzare i dati ottenuti direttamente dallo strumento: "Bruker XMASS" e "flexAnalysis" registrano le liste dei picchi in un semplice formato XML. È possibile utilizzare convertitori, come CompaxxXport, per convertire questi file nel formato mzXML. Tuttavia, è possibile anche leggere questi file direttamente in R, come vedremo in questa sezione.

Per prima cosa, installiamo e carichiamo la libreria *readBrukerFlexData* dal repository CRAN:

```
> install.packages("readBrukerFlexData")
> library(readBrukerFlexData)
```

Ora è necessario definire la directory in cui si trovano i file di dati. Utilizziamo il file di esempio del pacchetto *readBrukerFlexData*; il percorso del file può essere estratto utilizzando la funzione `system.file()` in R come segue:

```
> DIR <- system.file("Examples", package="readBrukerFlexData")
```

Per leggere tutti i file nella directory assegnata in precedenza, utilizziamo il seguente comando:

```
> myData_Bruker <- readBrukerFlexDir(file.path(DIR))
```

Diamo un'occhiata ai dati e alla loro struttura:

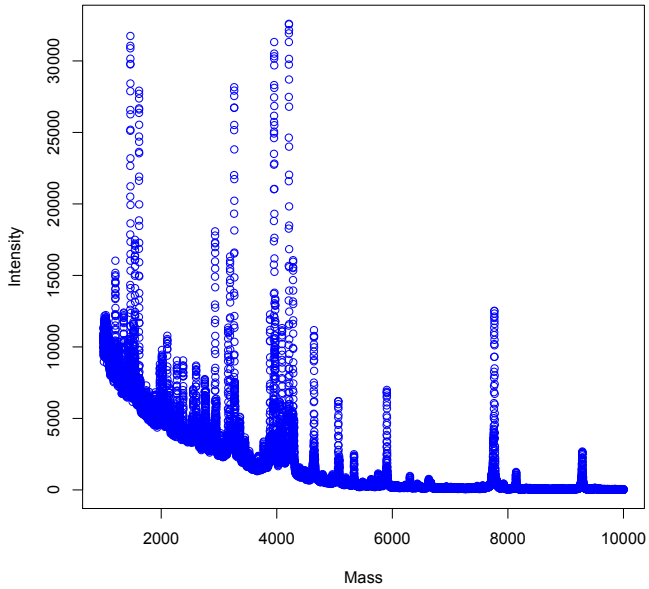
```
> summary(myData_Bruker)
                                              Length Class  Mode
s2010_05_19_Gibb_C8_A1.A1.T_0209513_0020253_98 2      -none- list
s2010_05_19_Gibb_C8_A1.A2.T_0209513_0020253_98 2      -none- list
sfid.A20.T_0209520_0020303_0                   2      -none- list
> str(myData_Bruker)
List of 3
 $ s2010_05_19_Gibb_C8_A1.A1.T_0209513_0020253_98:List of 2
  ..$ spectrum:List of 3
  .. ..$ tof      : num [1:22431] 21021 21023 21025 21027 21029 ...
  .. ..$ mass     : num [1:22431] 1000 1000 1000 1001 1001 ...
  .. ..$ intensity: num [1:22431] 11278 11350 10879 10684 10740 ...
  ..$ metaData:List of 48
  .. ..$ byteOrder              : chr "little"
  .. ..$ number                 : num 22431
  .. ..$ timeDelay              : num 21021
  .. ..$ timeDelta              : num 2
  .. ..$ calibrationConstants   : Named num [1:3] 2.32e+06 2.74e+02 -1.30e-03
  .. .. ..- attr(*, "names")= chr [1:3] "c1" "c2" "c3"
  .. ..$ hpcLimits              : Named num [1:2] 0 0
  .. .. ..- attr(*, "names")= chr [1:2] "minMass" "maxMass"
  .. ..$ hpcOrder               : num 0
  .. ..$ hpcUse                 : logi FALSE
…
```

Per tracciare il valore dell'intensità rispetto alla massa utilizziamo i campi corrispondenti dei dati:

```
> plot(myData_Bruker[[1]]$spectrum$mass, myData_Bruker[[1]]$spectrum$intensity,
      xlab="Mass", ylab="Intensity", col="blue")
```

Nel grafico ottenuto i picchi corrispondono alle intensità più elevate per i frammenti con il relativo rapporto m/z.

La struttura dei dati importati dalla funzione `readBrukerFlexData()` è leggermente diversa da quella dei formati visti in precedenza. I dati mzXML e Bruker hanno informazioni simili, ma sono organizzati in modo diverso. I dati Bruker restituiscono un oggetto in cui ogni elemento della lista ha delle sottoliste, cioè



lo spettro e i metaDati, con massa e intensità che sono i sottocomponenti dello slot dello spettro, mentre i dati mzXML hanno tre slot chiamati mass, intensity e metaData.

13.3 Conversione dei dati MS dal formato mzXML in MALDIquant

MALDIquant e il suo pacchetto aggiuntivo *MALDIquantForeign* sono pacchetti R che contengono molte funzioni per un'analisi di base dei dati MS. Pertanto, è essenziale poter convertire i dati in un formato appropriato per la loro elaborazione.

Per prima cosa, è opportuno installare e attivare i due pacchetti:

```
> install.packages(c("MALDIquant","MALDIquantForeign"))
> library(MALDIquant)
> library(MALDIquantForeign)
```

Iniziamo con l'importazione di un file nell'area di lavoro R dalla directory dei file di esempio del pacchetto:

```
> DIR <- system.file("exampledata", package="MALDIquantForeign")
> myData_MALDI <- import(DIR, type="auto", centroided=TRUE)
```

Esaminiamo la classe dell'oggetto importato:

```
> class(myData_MALDI[[1]])
[1] "MassPeaks"
attr(,"package")
[1] "MALDIquant"
```

Osservando la struttura di uno dei componenti (spettri) dei dati importati, si scopre che è molto simile all'oggetto mzXML che abbiamo creato nella sezione 13.1:

```
> str(myData_MALDI[[1]])
Formal class 'MassPeaks' [package "MALDIquant"] with 4 slots
  ..@ snr      : num [1:5] NA NA NA NA NA
  ..@ mass     : num [1:5] 1 2 3 4 5
  ..@ intensity: num [1:5] 6 7 8 9 10
  ..@ metaData :List of 18
  .. ..$ file           : chr "/Library/Frameworks/R.framework/Versions/3.6/Resources/library/
MALDIquantForeign/exampledata/tiny1-centroided.mzXML3.0.mzXML"
  .. ..$ scanCount      : num 1
  .. ..$ parentFile     :List of 1
  .. .. ..$ :List of 3
  .. .. .. ..$ fileName: chr "file://tiny.RAW"
  .. .. .. ..$ fileType: chr "RAW"
  .. .. .. ..$ fileSha1: chr "0000000000000000000000000000000000000000"
  .. ..$ msInstrument   :List of 6
  .. .. ..$ msManufacturer: chr "FooBar"
  .. .. ..$ msModel       : chr "FooBar Model1"
  .. .. ..$ msIonisation  : chr "MALDI"
  .. .. ..$ msMassAnalyzer: chr "TOF"
  .. .. ..$ msDetector    : chr "msD"
  .. .. ..$ software      :List of 3
  .. .. .. ..$ type   : chr "acquisition"
  .. .. .. ..$ name   : chr "AcquisitionSoftware"
  .. .. .. ..$ version: chr "1.0.0"
  .. ..$ num            : num 1
  .. ..$ peaksCount     : num 5
  .. ..$ msLevel        : num 1
  .. ..$ polarity       : chr "+"
  .. ..$ scanType       : chr "Full"
  .. ..$ centroided     : num 1
  .. ..$ lowMz          : num 1
  .. ..$ highMz         : num 5
  .. ..$ totIonCurrent  : num 40
  .. ..$ precision      : num 64
  .. ..$ byteOrder      : chr "network"
  .. ..$ contentType    : chr "m/z-int"
  .. ..$ compressionType: chr "none"
  .. ..$ compressedLen  : num 0
```

Il pacchetto *MALDIquantForeign* è un pacchetto ausiliario del pacchetto *MALDIquant*. Legge e scrive diversi formati di file di dati MS da e verso oggetti MALDIquant; supporta le funzioni di importazione ed esportazione. In questa sezione ci siamo concentrati sulla funzione di importazione, che analizza i file di input e scrive il contenuto nel formato di dati richiesto.

13.4 Estrazione di elementi dagli oggetti dei dati MS

Nelle sezioni precedenti ci siamo concentrati su diversi formati di dati MS. Come abbiamo visto, i dati MS hanno componenti o slot diversi. Ogni slot contiene un tipo specifico di informazione. Questa sezione presenta i metodi che possono essere utilizzati per ottenere contenuti specifici dai dati.

Attiviamo innanzitutto il pacchetto *MALDIquant* nella sessione R:

```
> library(MALDIquant)
```

Ora carichiamo i dati disponibili nel pacchetto *MALDIquant* e esaminiamone il contenuto:

```
> data("fiedler2009subset", package="MALDIquant")
> summary(fiedler2009subset)
                                                 Length Class        Mode
sPankreas_HB_L_061019_G10.M19.T_0209513_0020740_18 42388  MassSpectrum S4
sPankreas_HB_L_061019_G10.M20.T_0209513_0020740_18 42388  MassSpectrum S4
sPankreas_HB_L_061019_H7.O14.T_0209513_0020740_18  42388  MassSpectrum S4
sPankreas_HB_L_061019_H7.P13.T_0209513_0020740_18  42388  MassSpectrum S4
sPankreas_HB_L_061019_F10.L19.T_0209513_0020740_18 42388  MassSpectrum S4
sPankreas_HB_L_061019_F10.L20.T_0209513_0020740_18 42388  MassSpectrum S4
sPankreas_HB_L_061019_F9.L17.T_0209513_0020740_18  42388  MassSpectrum S4
sPankreas_HB_L_061019_F9.L18.T_0209513_0020740_18  42388  MassSpectrum S4
sPankreas_HB_L_061019_A6.A11.T_0209513_0020740_18  42388  MassSpectrum S4
sPankreas_HB_L_061019_A6.A12.T_0209513_0020740_18  42388  MassSpectrum S4
sPankreas_HB_L_061019_A8.A15.T_0209513_0020740_18  42388  MassSpectrum S4
sPankreas_HB_L_061019_A8.A16.T_0209513_0020740_18  42388  MassSpectrum S4
sPankreas_HB_L_061019_C4.F7.T_0209513_0020740_18   42388  MassSpectrum S4
sPankreas_HB_L_061019_C4.F8.T_0209513_0020740_18   42388  MassSpectrum S4
sPankreas_HB_L_061019_D9.G17.T_0209513_0020740_18  42388  MassSpectrum S4
sPankreas_HB_L_061019_D9.G18.T_0209513_0020740_18  42388  MassSpectrum S4
```

Per ulteriori analisi, utilizzeremo solo il primo elemento della lista:

```
> myData <- fiedler2009subset[[1]]
```

I dati sono un'istanza della classe "MassSpectrum":

```
> class(myData)
[1] "MassSpectrum"
attr(,"package")
[1] "MALDIquant"
```

È necessario conoscere la struttura dei dati per estrarne gli elementi utili. Diamo un'occhiata alla struttura dettagliata dell'oggetto *MyData* (che è composto dei tre slot mass, intensity e metaData):

```
> str(myData)
Formal class 'MassSpectrum' [package "MALDIquant"] with 3 slots
  ..@ mass     : num [1:42388] 1000 1000 1000 1000 1000 ...
  ..@ intensity: int [1:42388] 3149 3134 3127 3131 3170 3162 3162 3152 3174 3137 ...
  ..@ metaData :List of 42
  .. ..$ byteOrder           : chr "little"
  .. ..$ number              : num 42388
  .. ..$ timeDelay           : num 19886
  .. ..$ timeDelta           : num 1
…
  .. ..$ file                : chr "/data/set A - discovery leipzig/control/
Pankreas_HB_L_061019_G10/0_m19/1/1SLin/fid"
  .. ..$ sampleName          : chr "Pankreas_HB_L_061019_G10"
  .. ..$ fullName            : chr "Pankreas_HB_L_061019_G10.M19"
  .. ..$ name                : chr "Pankreas_HB_L_061019_G10.M19"
```

Per estrarre il rapporto m/z, prendiamo la componente della massa (slot `@mass`) come segue:

```
> m_z <- myData@mass
```

Analogamente prendiamo le intensità dalla relativa componente:

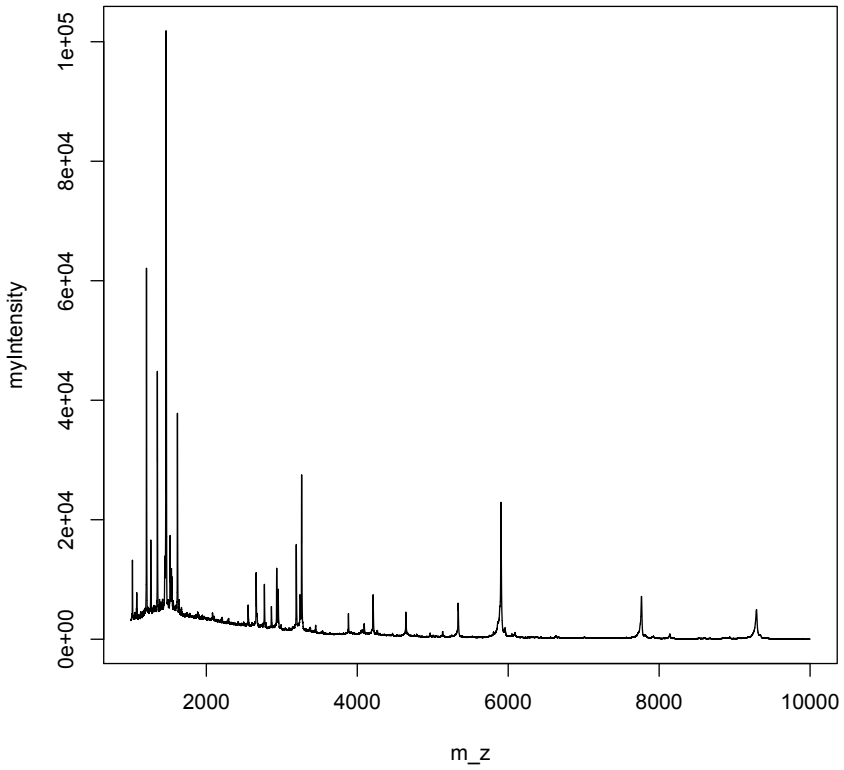
```
> myIntensity <- myData@intensity
```

Diamo anche un'occhiata all'esperimento e ai dettagli del campione esaminando lo slot `@metaData`:

```
> myData@metaData
$byteOrder
[1] "little"
$number
[1] 42388
$timeDelay
[1] 19886
$timeDelta
[1] 1
$calibrationConstants
          c1           c2           c3
 2.59729e+06  2.68443e+02 -4.43352e-03
…
$instrument
[1] "AUTOFLEX"
$instrumentId
[1] "25001.00183"
$instrumentType
[1] "0"
$massError
[1] 1000
$laserShots
[1] 450
$patch
[1] "M19"
$path
[1] "D:\\data\\Rohdaten\\serum\\control\\MatrixOne\\serum_450\\Eluate 1\
\Pankreas_HB_L_061019_G10\\0_M19\\1\\1SLin"
…
$targetCount
[1] 18
$targetIdString
[1] "T_0209513_0020740_18"
$targetSerialNumber
[1] "0020740"
$targetTypeNumber
[1] "0209513"
$file
[1] "/data/set A - discovery leipzig/control/Pankreas_HB_L_061019_G10/0_m19/1/1SLin/fid"
$sampleName
[1] "Pankreas_HB_L_061019_G10"
$fullName
[1] "Pankreas_HB_L_061019_G10.M19"
$name
[1] "Pankreas_HB_L_061019_G10.M19"
```

I grafici realizzati nelle precedenti sezioni per la massa e l'intensità possono essere realizzati semplicemente utilizzando i singoli componenti con il seguente comando:

```
> plot(m_z, myIntensity, type="l", col="black")
```



L'analisi qui illustrata si basa sulla struttura dei dati che provengono da un esperimento su otto soggetti (quattro soggetti di controllo sani e quattro soggetti malati) con due repliche di ciascuno, con l'obiettivo di studiare il peptide associato al cancro al pancreas. La struttura dei dati permette di estrarre intuitivamente gli elementi di interesse. Possiamo ad esempio tabulare le intensità in ogni campione in una tabella costituita di due colonne: i dati di massa (m_z) e le intensità (intensity):

```
> myTab = matrix(nrow=length(m_z), ncol=2)
> myTab[,1] = m_z
> myTab[,2] = myIntensity
> colnames(myTab) = c("m_z","intensity")
> head(myTab)
          m_z intensity
[1,] 1000.015      3149
[2,] 1000.117      3134
[3,] 1000.219      3127
[4,] 1000.321      3131
[5,] 1000.423      3170
[6,] 1000.525      3162
```

13.5 Pre-elaborazione dei dati MS

Ogni volta che una proteina o un peptide ionizzato colpisce il rivelatore, lo strumento lo registra. Questo frammento ionizzato ha un rapporto m/z, e i rivelatori misurano l'intensità di questo frammento chiamato *spettro di massa*. Gli spettri di massa hanno diverse imperfezioni, che possono complicarne l'interpretazione. Il rilevamento dei picchi in questi spettri è impegnativa perché alcuni peptidi con bassa abbondanza possono essere nascosti dal rumore, causando un'alta percentuale di picchi falsi positivi. Il rumore chimico, di ionizzazione e elettronico spesso si traduce in una curva decrescente nel background dei

dati MS, che viene indicata come una linea di base (*baseline*). La sua esistenza produce una forte distorsione nel rilevamento dei picchi: pertanto, è auspicabile normalizzare, livellare e rimuovere la linea di base prima del rilevamento dei picchi. Questi sono alcuni problemi comuni con cui l'analista dei dati MS ha a che fare. Ci occuperemo di alcune di queste fasi di pre-elaborazione in questa sezione, con particolare attenzione alla tecnica MALDI.

Attiviamo la libreria *MALDIquant* e carichiamo i relativi dati (prendiamo il primo dei 16 spettri):

```
> library(MALDIquant)
> data("fiedler2009subset", package="MALDIquant")
> myData <- fiedler2009subset[[1]]
```

Iniziamo con la trasformazione dell'intensità dei dati mediante la funzione `transformIntensity()`:

```
> spec <- transformIntensity(myData, method="sqrt")
```

Altre possibili opzioni per l'argomento `method` sono `log`, `log2` e `log10`. Utilizzeremo l'intensità trasformata come input per i passi successivi.

Livelliamo l'intensità usando la funzione `smoothIntensity()` (la funzione può utilizzare diversi filtri di riduzione del rumore, tra cui SavitzkyGolay e MovingAverage):

```
> spec <- smoothIntensity(spec, method="MovingAverage")
```

Eseguiamo ora la correzione della linea di base mediante la funzione `removeBaseline()`:

```
> spec <- removeBaseline(spec)
```

Infine, normalizziamo i dati con la funzione `calibrateIntensity()`:

```
> spec <- calibrateIntensity(spec, method="TIC")
```

Stimiamo il rumore dei dati MS come segue:

```
> n <- estimateNoise(spec)
> head(n)
         mass   intensity
[1,] 1000.015 3.77133e-05
[2,] 1000.117 3.77133e-05
[3,] 1000.219 3.77133e-05
[4,] 1000.321 3.77133e-05
[5,] 1000.423 3.77133e-05
[6,] 1000.525 3.77133e-05
```

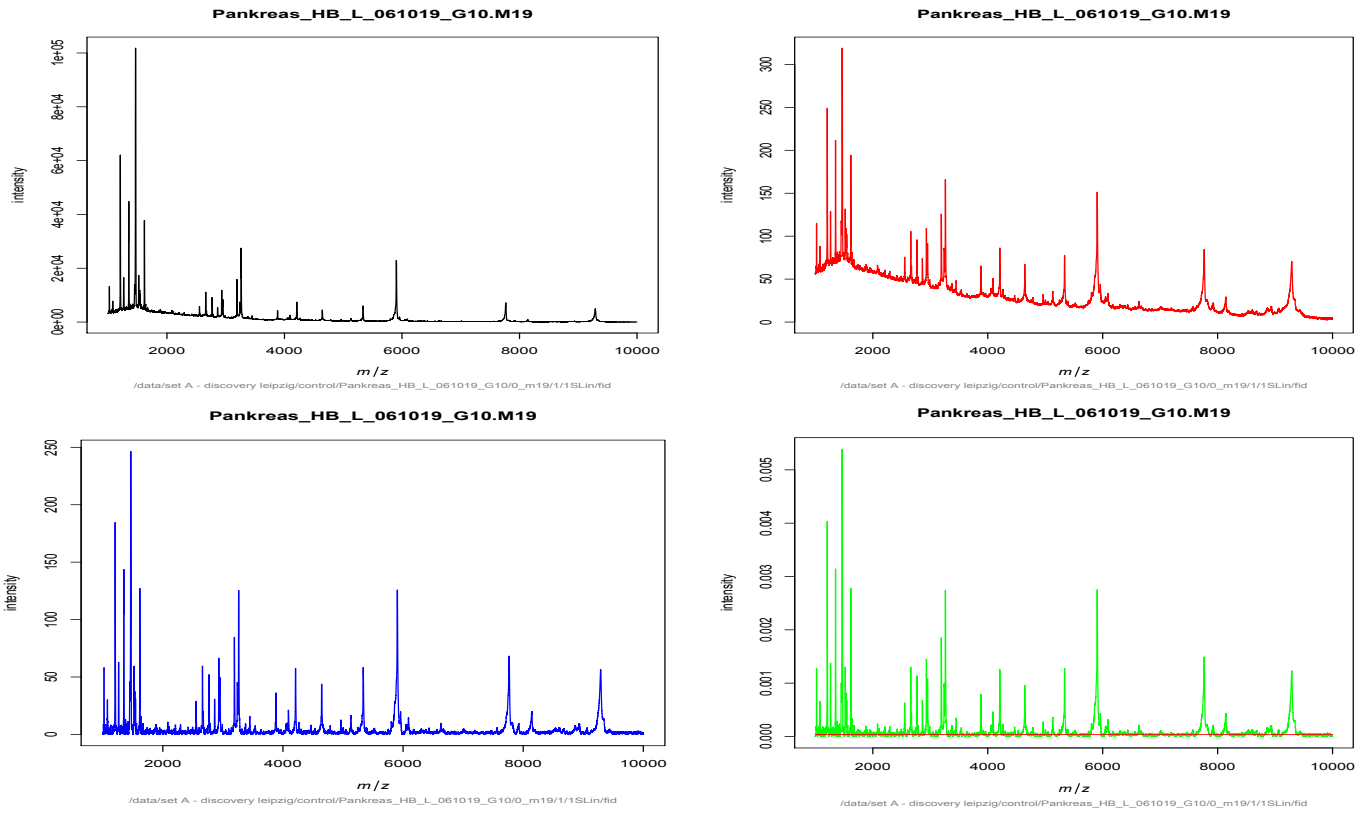
Trasformando l'intensità si migliora la risoluzione dei dati: ciò significa che i dati diventano più distinguibili per i classificatori e altri metodi. Come abbiamo visto, sono possibili diversi tipi di trasformazione, tra cui la trasformazione della radice quadrata e la trasformazione logaritmica (con base 2, 10 e naturale). La trasformazione con la radice quadrata evita l'asimmetria nella distribuzione, mentre una trasformazione

logaritmica può risultare in una distribuzione leggermente asimmetrica; tuttavia, un errore significativo è raro. Pertanto, la scelta della trasformazione dipende in gran parte dall'utente. Se la trasformazione logaritmica dà risultati inaspettatamente asimmetrici, si dovrebbe provare la trasformazione con la radice quadrata.

I dati MS possono avere del rumore dovuto al progetto dell'esperimento o ad altri fattori. Il livellamento (*smoothing*) riduce il rumore in tutto lo spettro. Può utilizzare vari metodi: il metodo MovingAverage esegue una semplice media mobile su due lati; lo smoothing SavitzkyGolay utilizza un filtro Savitzky-Golay basato sui minimi quadrati parziali. La correzione della linea di base è un altro metodo di riduzione del rumore per i dati MS; utilizza uno dei metodi SNIP, TopHat, ConvexHull e mediano.

La normalizzazione dei dati viene eseguita mediante la calibrazione attraverso la funzione `calibrateIntensity()`. I metodi implementati per la normalizzazione includono TIC, PQN e mediano. Il metodo TIC imposta la corrente ionica totale (TIC, Total Ionic Current) dello spettro a uno per la normalizzazione, mentre il metodo mediano imposta la mediana delle intensità a uno. Il metodo PQN è più complicato: calcola la calibrazione TIC per i dati e calcola anche lo spettro mediano. Poi, i quozienti di tutte le intensità vengono calcolati con la mediana di riferimento, e viene calcolata la mediana per questo spettro. Infine, le intensità vengono divise per la mediana ottenuta nell'ultimo passo per ottenere i dati normalizzati. Possiamo tracciare lo spettro calcolato ad ogni passo usando la funzione `plot()`. Un riferimento più dettagliato per l'approccio PQN è disponibile nell'articolo *Probabilistic Quotient Normalization as Robust Method to Account for Dilution of Complex Biological Mixtures. Application in ^{1}H NMR Metabonomics* di Dieterle *et al*. su http://pubs.acs.org/doi/abs/10.1021/ac051632c.

La schermata seguente mostra i valori grezzi iniziali (in nero), la trasformazione della radice quadrata (in rosso), la correzione della linea di base(in blu) e l'intensità calibrata (in verde, con il rumore in rosso):



13.6 Rilevamento dei picchi nei dati MS

Di solito, i segnali peptidici appaiono come massimi locali chiamati picchi negli spettri MS. Questi picchi contengono informazioni su molecole con concentrazioni elevate. Una procedura di rilevamento dei picchi calcola il cosiddetto *spettro di linea* per uno spettro grezzo, che è un elenco delle posizioni e intensità dei picchi. Il rilevamento dei picchi è di solito una fase iniziale e cruciale dell'analisi dei dati MS. Le fasi di preelaborazione viste nella sezione precedente rendono più facile eseguire il rilevamento dei picchi.

Continueremo ad utilizzare il pacchetto *MALDIquant* per il rilevamento dei picchi. In questa sezione useremo i dati dello stesso pacchetto a scopo dimostrativo. Iniziamo caricando la libreria *MALDIquant* e i dati come segue:

```
> library(MALDIquant)
> data("fiedler2009subset")
> myData <- fiedler2009subset[[1]]
```

È necessario pre-elaborare i dati prima del rilevamento dei picchi. Eseguiamo quindi preliminarmente (*i*) la trasformazione dei valori delle intensità alle loro radici quadrate, (*ii*) il livellamento delle intensità trasformate (questa volta utilizzando il filtro Savitzky-Golay), (*iii*) la correzione della linea di base e infine (*iv*) la normalizzazione dei dati come segue:

```
> spec <- transformIntensity(myData, method="sqrt")
> spec <- smoothIntensity(spec, method="SavitzkyGolay")
> spec <- removeBaseline(spec)
> spec <- calibrateIntensity(spec, method= "TIC")
```

Richiamiamo la funzione `detectPeaks()` per rilevare i picchi nei dati normalizzati:

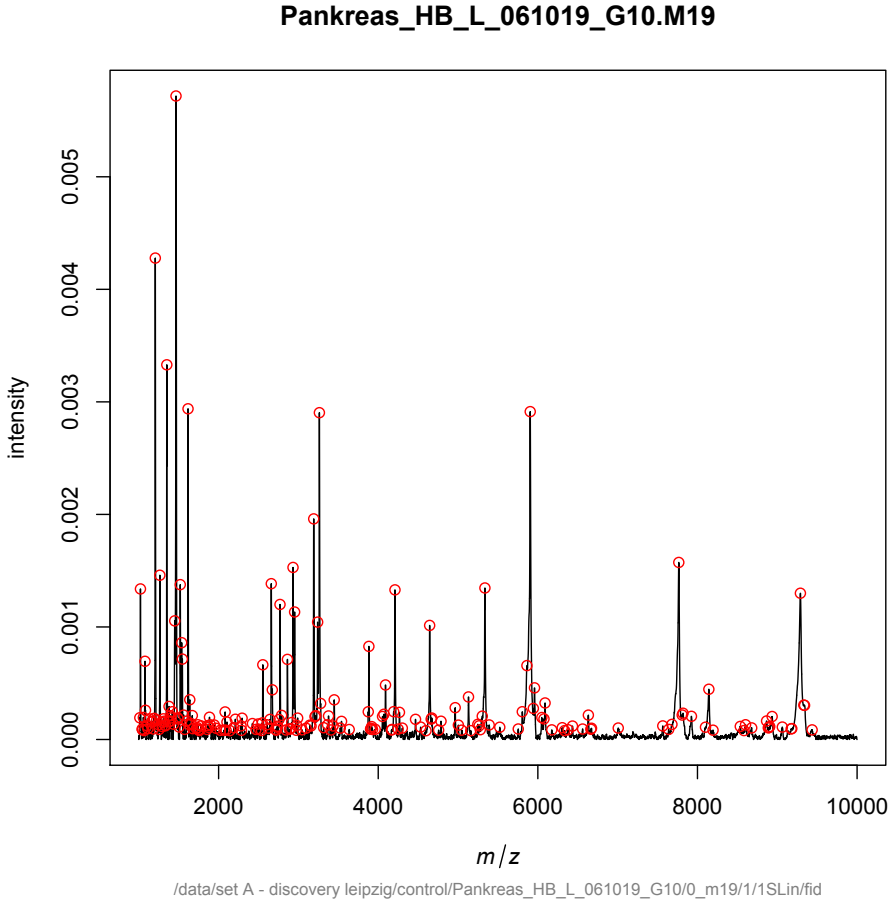
```
> peak <- detectPeaks(spec, method="MAD", SNR=2)
```

Se esaminiamo la struttura dell'oggetto vediamo che è simile a quella dei dati originali:

```
> str(peak)
Formal class 'MassPeaks' [package "MALDIquant"] with 4 slots
  ..@ snr      : num [1:207] 5.01 34.73 2.41 2.26 5.08 ...
  ..@ mass     : num [1:207] 1013 1021 1042 1046 1057 ...
  ..@ intensity: num [1:207] 1.93e-04 1.34e-03 9.27e-05 8.69e-05 1.96e-04 ...
  ..@ metaData :List of 42
  .. ..$ byteOrder           : chr "little"
  .. ..$ number              : num 42388
  .. ..$ timeDelay           : num 19886
  .. ..$ timeDelta           : num 1
  .. ..$ calibrationConstants: Named num [1:3] 2.60e+06 2.68e+02 -4.43e-03
  .. .. ..- attr(*, "names")= chr [1:3] "c1" "c2" "c3"
  .. ..$ hpcLimits           : Named num [1:2] 0 0
  .. .. ..- attr(*, "names")= chr [1:2] "minMass" "maxMass"
  .. ..$ hpcOrder            : num 0
  .. ..$ hpcUse              : logi TRUE
  .. ..$ dataType            : chr "CONTINUOUS MASS SPECTRUM"
  .. ..$ dataSystem          : chr "Bruker Flex Series"
…
```

Per visualizzare i picchi, tracciamo prima il grafico delle intensità e poi aggiungiamo i punti dei picchi:

```
> plot(spec)
> points(peak, col="red")
```



All'inizio della sezione abbiamo eseguito la pre-elaborazione dei dati come nella sezione precedente. Il metodo `detectPeaks()` cerca i picchi nei dati MS (rappresentati dall'oggetto *spec* di tipo MassSpectrum). Un picco è un massimo locale al di sopra di una soglia di rumore definita dall'utente. La funzione stima il rumore nei dati utilizzando l'argomento `method`, che accetta due valori, MAD e SuperSmoother. L'altro argomento è `SNR` che si riferisce al rapporto segnale-rumore nei dati e necessita di un valore numerico (il valore predefinito è 2). La funzione calcola prima il rumore del segnale; per identificare un massimo locale come picco, il valore deve essere più alto secondo la formula:

$$peak > noise \times SNR$$

Tutti i valori per i quali la condizione precedente è soddisfatta vengono rilevati come picchi. Il calcolo della deviazione assoluta mediana (MAD) stima il rumore nei dati. Il metodo SuperSmoother utilizza invece l'approccio super smoother di Friedman per calcolare il rumore.

13.7 Allineamento dei picchi con i dati MS

Per il confronto dei picchi in diversi spettri, è essenziale effettuare l'allineamento. Inoltre, la risoluzione dello strumento o la sua calibrazione può influenzare la qualità dei dati a causa della variazione del rapporto

massa/carica in qualsiasi punto. Pertanto, è necessario anche l'allineamento degli spettri all'interno dei set di dati. Questa sezione si occupa dell'allineamento dei picchi nei dati MS, utilizzando la stessa libreria e gli stessi dati (con relativa pre-elaborazione) della sezione precedente:

```
> library(MALDIquant)
> data("fiedler2009subset")
> myData <- fiedler2009subset
> spec <- transformIntensity(myData, method="sqrt")
> spec <- smoothIntensity(spec, method="SavitzkyGolay")
> spec <- removeBaseline(spec)
> spec <- calibrateIntensity(spec, method="TIC")
```

Utilizziamo infine la funzione `alignSpectra()` per allineare i picchi elaborati e normalizzati:

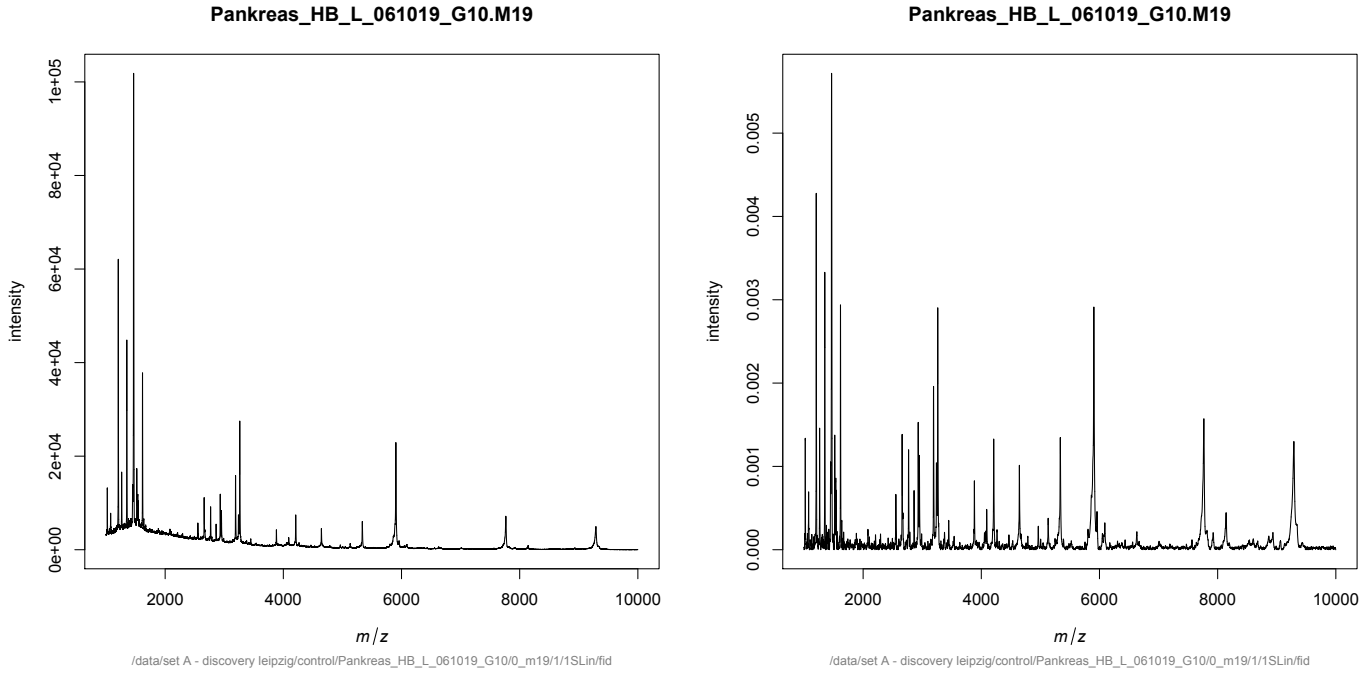
```
> spectra <- alignSpectra(spec)
```

Controlliamo gli spettri originali e quelli allineati (si tratta di una lista di 16 spettri ciascuno):

```
> summary(myData)
                                                  Length Class         Mode
sPankreas_HB_L_061019_G10.M19.T_0209513_0020740_18 42388  MassSpectrum S4
sPankreas_HB_L_061019_G10.M20.T_0209513_0020740_18 42388  MassSpectrum S4
sPankreas_HB_L_061019_H7.O14.T_0209513_0020740_18  42388  MassSpectrum S4
sPankreas_HB_L_061019_H7.P13.T_0209513_0020740_18  42388  MassSpectrum S4
sPankreas_HB_L_061019_F10.L19.T_0209513_0020740_18 42388  MassSpectrum S4
sPankreas_HB_L_061019_F10.L20.T_0209513_0020740_18 42388  MassSpectrum S4
sPankreas_HB_L_061019_F9.L17.T_0209513_0020740_18  42388  MassSpectrum S4
sPankreas_HB_L_061019_F9.L18.T_0209513_0020740_18  42388  MassSpectrum S4
sPankreas_HB_L_061019_A6.A11.T_0209513_0020740_18  42388  MassSpectrum S4
sPankreas_HB_L_061019_A6.A12.T_0209513_0020740_18  42388  MassSpectrum S4
sPankreas_HB_L_061019_A8.A15.T_0209513_0020740_18  42388  MassSpectrum S4
sPankreas_HB_L_061019_A8.A16.T_0209513_0020740_18  42388  MassSpectrum S4
sPankreas_HB_L_061019_C4.F7.T_0209513_0020740_18   42388  MassSpectrum S4
sPankreas_HB_L_061019_C4.F8.T_0209513_0020740_18   42388  MassSpectrum S4
sPankreas_HB_L_061019_D9.G17.T_0209513_0020740_18  42388  MassSpectrum S4
sPankreas_HB_L_061019_D9.G18.T_0209513_0020740_18  42388  MassSpectrum S4
> summary(spectra)
      Length Class         Mode
 [1,] 42388  MassSpectrum S4
 [2,] 42388  MassSpectrum S4
 [3,] 42388  MassSpectrum S4
 [4,] 42388  MassSpectrum S4
 [5,] 42388  MassSpectrum S4
 [6,] 42388  MassSpectrum S4
 [7,] 42388  MassSpectrum S4
 [8,] 42388  MassSpectrum S4
 [9,] 42388  MassSpectrum S4
[10,] 42388  MassSpectrum S4
[11,] 42388  MassSpectrum S4
[12,] 42388  MassSpectrum S4
[13,] 42388  MassSpectrum S4
[14,] 42388  MassSpectrum S4
[15,] 42388  MassSpectrum S4
[16,] 42388  MassSpectrum S4
```

Tracciamo i relativi grafici come segue (a sinistra il grafico del primo spettro originario; a destra quello del primo spettro allineato):

```
> plot(myData[[1]])
> plot(spectra[[1]])
```



Per far corrispondere i picchi che hanno gli stessi valori di massa, *MALDIquant* utilizza un approccio basato sulla regressione statistica. In primo luogo, vengono identificati i picchi di riferimento, che si verificano nella maggior parte degli spettri. Successivamente, viene calcolata una funzione di deformazione non lineare per ogni spettro, adattando una regressione locale ai picchi di riferimento abbinati. Questo permette anche di fondere gli spettri allineati da repliche tecniche.

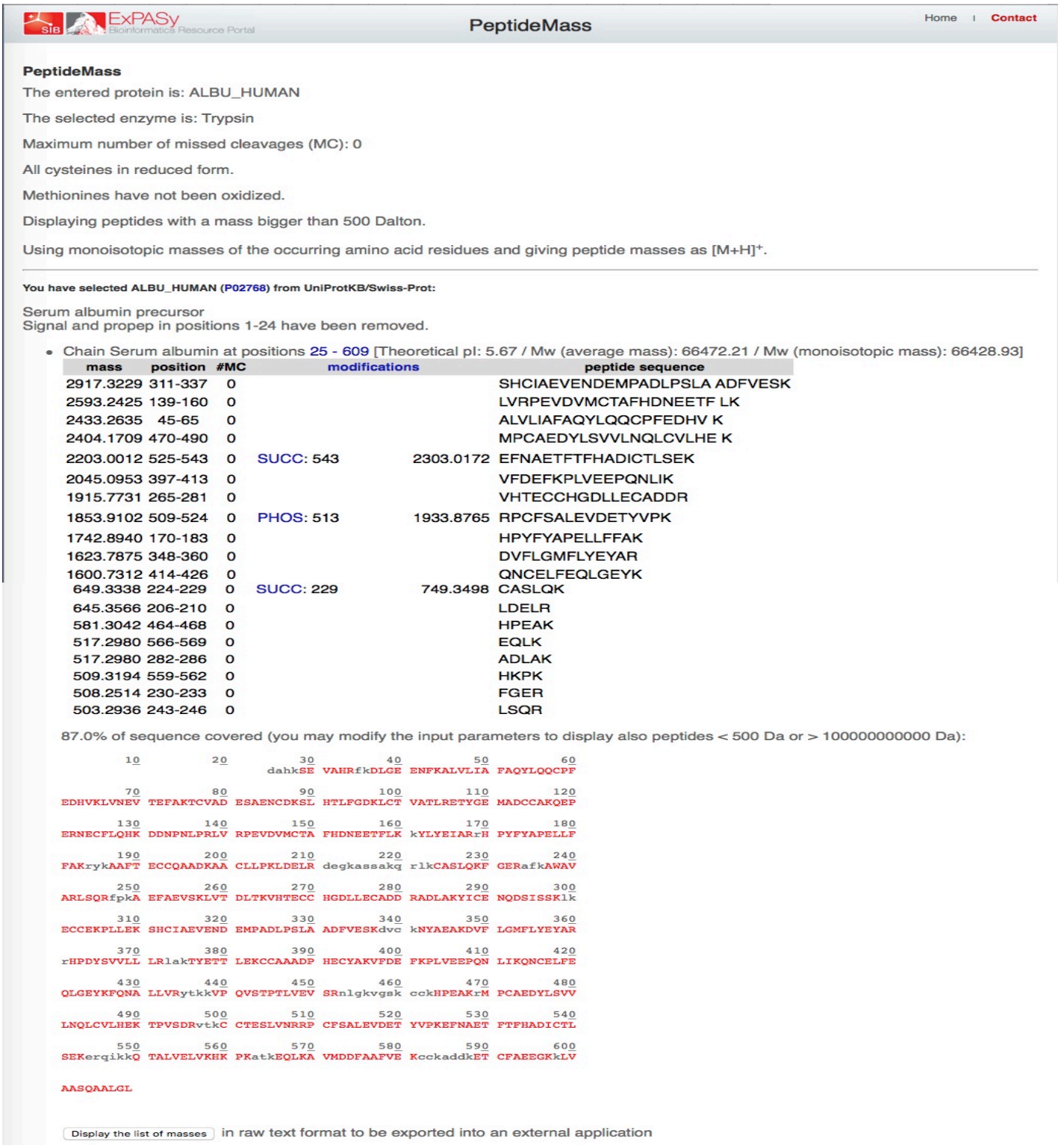
Il metodo di allineamento implementato nel pacchetto *MALDIquant* è un ibrido di due metodi diffusi, il DISCO e il *warping* autocalibrato. Per conoscere i dettagli concettuali dell'approccio DISCO fare riferimento all'articolo *DISCO: Distance and Spectrum Correlation Optimization Alignment for Two-Dimensional Gas Chromatography Time-of-Flight Mass Spectrometry-Based Metabolomics* di Wang *et al*. (http://pubs.acs.org/doi/abs/10.1021/ac100064b). Il secondo metodo è spiegato in dettaglio nell'articolo *Self-Calibrated Warping for Mass Spectra Alignment* di Peter He *et al*. (http://www.ncbi.nlm.nih.gov/pmc/articles/PMC3085421/).

13.8 Identificazione dei peptidi nei dati MS

Esaminiamo ora l'inferenza biologica dei dati MS. Identificare i peptidi dai dati MS è uno dei compiti più diffusi nell'analisi dei dati MS. Attualmente l'identificazione delle proteine viene regolarmente eseguita con la *spettrometria di massa tandem* (MS-MS). A causa della difficoltà di misurare le proteine intatte con MS-MS, una proteina viene tipicamente digerita in peptidi con enzimi e viene misurato lo spettro MS-MS di ogni peptide. Qui presentiamo una tecnica per identificare i peptidi nei dati MS-MS.

Introdurremo il nuovo pacchetto R *protViz*. I dati MS che usiamo in questa sezione sono creati artificialmente dai dati precedenti. Il peptide che cerchiamo di rilevare è l'albumina sierica umana (ALB).

Iniziamo ottenendo i dati peptidici adatti dal sito web SwissProt-ExPASy per la proteina dell'albumina umana: https://web.expasy.org/peptide_mass. Inseriamo l'identificatore UniProt KB per l'albumina (ALBU_HUMAN) nella casella di interrogazione e facciamo clic sul pulsante <Perform> (lasciare inalterate le impostazioni degli altri parametri). Quello che otteniamo è il risultato della digestione della proteina ALBU_HUMAN da parte dell'enzima *tripsina*. La seguente schermata mostra la pagina web di ExPASy risultante:



ExPASy Bioinformatics Resource Portal — PeptideMass — Home | Contact

PeptideMass

The entered protein is: ALBU_HUMAN

The selected enzyme is: Trypsin

Maximum number of missed cleavages (MC): 0

All cysteines in reduced form.

Methionines have not been oxidized.

Displaying peptides with a mass bigger than 500 Dalton.

Using monoisotopic masses of the occurring amino acid residues and giving peptide masses as $[M+H]^+$.

You have selected ALBU_HUMAN (P02768) from UniProtKB/Swiss-Prot:

Serum albumin precursor
Signal and propep in positions 1-24 have been removed.

- Chain Serum albumin at positions 25 - 609 [Theoretical pI: 5.67 / Mw (average mass): 66472.21 / Mw (monoisotopic mass): 66428.93]

mass	position	#MC	modifications		peptide sequence
2917.3229	311-337	0			SHCIAEVENDEMPADLPSLA ADFVESK
2593.2425	139-160	0			LVRPEVDVMCTAFHDNEETF LK
2433.2635	45-65	0			ALVLIAFAQYLQQCPFEDHV K
2404.1709	470-490	0			MPCAEDYLSVVLNQLCVLHE K
2203.0012	525-543	0	SUCC: 543	2303.0172	EFNAETFTFHADICTLSEK
2045.0953	397-413	0			VFDEFKPLVEEPQNLIK
1915.7731	265-281	0			VHTECCHGDLLECADDR
1853.9102	509-524	0	PHOS: 513	1933.8765	RPCFSALEVDETYVPK
1742.8940	170-183	0			HPYFYAPELLFFAK
1623.7875	348-360	0			DVFLGMFLYEYAR
1600.7312	414-426	0			QNCELFEQLGEYK
649.3338	224-229	0	SUCC: 229	749.3498	CASLQK
645.3566	206-210	0			LDELR
581.3042	464-468	0			HPEAK
517.2980	566-569	0			EQLK
517.2980	282-286	0			ADLAK
509.3194	559-562	0			HKPK
508.2514	230-233	0			FGER
503.2936	243-246	0			LSQR

87.0% of sequence covered (you may modify the input parameters to display also peptides < 500 Da or > 100000000000 Da):

```
        10         20         30         40         50         60
                  dahkSE VAHRfkDLGE ENFKALVLIA FAQYLQQCPF

        70         80         90        100        110        120
EDHVKLVNEV TEFAKTCVAD ESAENCDKSL HTLFGDKLCT VATLRETYGE MADCCAKQEP

       130        140        150        160        170        180
ERNECFLQHK DDNPNLPRLV RPEVDVMCTA FHDNEETFLK kYLYEIARrH PYFYAPELLF

       190        200        210        220        230        240
FAKrykAAFT ECCQAADKAA CLLPKLDELR degkassakq rlkCASLQKF GERafkAWAV

       250        260        270        280        290        300
ARLSQRfpkA EFAEVSKLVT DLTKVHTECC HGDLLECADD RADLAKYICE NQDSISSKlk

       310        320        330        340        350        360
ECCEKPLLEK SHCIAEVEND EMPADLPSLA ADFVESKdvc kNYAEAKDVF LGMFLYEYAR

       370        380        390        400        410        420
rHPDYSVVLL LRlakTYETT LEKCCAAADP HECYAKVFDE FKPLVEEPQN LIKQNCELFE

       430        440        450        460        470        480
QLGEYKFQNA LLVRytkkVP QVSTPTLVEV SRnlgkvgsk cckHPEAKrM PCAEDYLSVV

       490        500        510        520        530        540
LNQLCVLHEK TPVSDRvtkC CTESLVNRRP CFSALEVDET YVPKEFNAET FTFHADICTL

       550        560        570        580        590        600
SEKerqikkQ TALVELVKHK PKatkEQLKA VMDDFAAFVE KcckaddkET CFAEEGKkLV

AASQAALGL
```

Display the list of masses in raw text format to be exported into an external application

Per la successiva elaborazione è necessario il pacchetto R standard *protViz*:

```
> install.packages("protViz", dependencies=TRUE)
> library(protViz)
```

Per le ulteriori analisi prendiamo solo i primi peptidi dal risultato della ricerca effettuata sul sito web SwissProt-ExPASy e inseriamoli in una stringa di caratteri come segue:

```
> myPeptide <- "SHCIAEVENDEMPADLPSLAADFVESK"
```

Creiamo i dati dello spettro artificiale (che inseriremo nella variabile *mySpec*). Per comodità, utilizziamo uno degli spettri visti finora, che sottoponiamo alle stesse fasi di pre-elaborazione viste in precedenza prima di creare lo spettro artificiale:

```
> myData <- fiedler2009subset[[1]]
> spec = transformIntensity(myData, method="sqrt")
> spec = smoothIntensity(spec, method="SavitzkyGolay")
> spec = removeBaseline(spec)
> spec = calibrateIntensity(spec, method="TIC")
> mySpec <- list(title="artificial", charge=2, mz=spec@mass,
      intensity=spec@intensity)
```

Una volta pronti i dati da confrontare, ricaviamo il numero di caratteri del peptide mediante la funzione `nchar()`:

```
> n <- nchar(myPeptide)
```

Calcoliamo gli ioni frammento della sequenza peptidica ed estraiamo gli ioni di nostra scelta (gli ioni *b* e *y*, che etichettiamo di conseguenza) digitando i seguenti comandi:

```
> fi <- fragmentIon(myPeptide)
> by_mz <- c(fi[[1]]$b, fi[[1]]$y)
> by_label <- c(paste("b", 1:n, sep=""),paste("y", 1:n, sep=""))
```

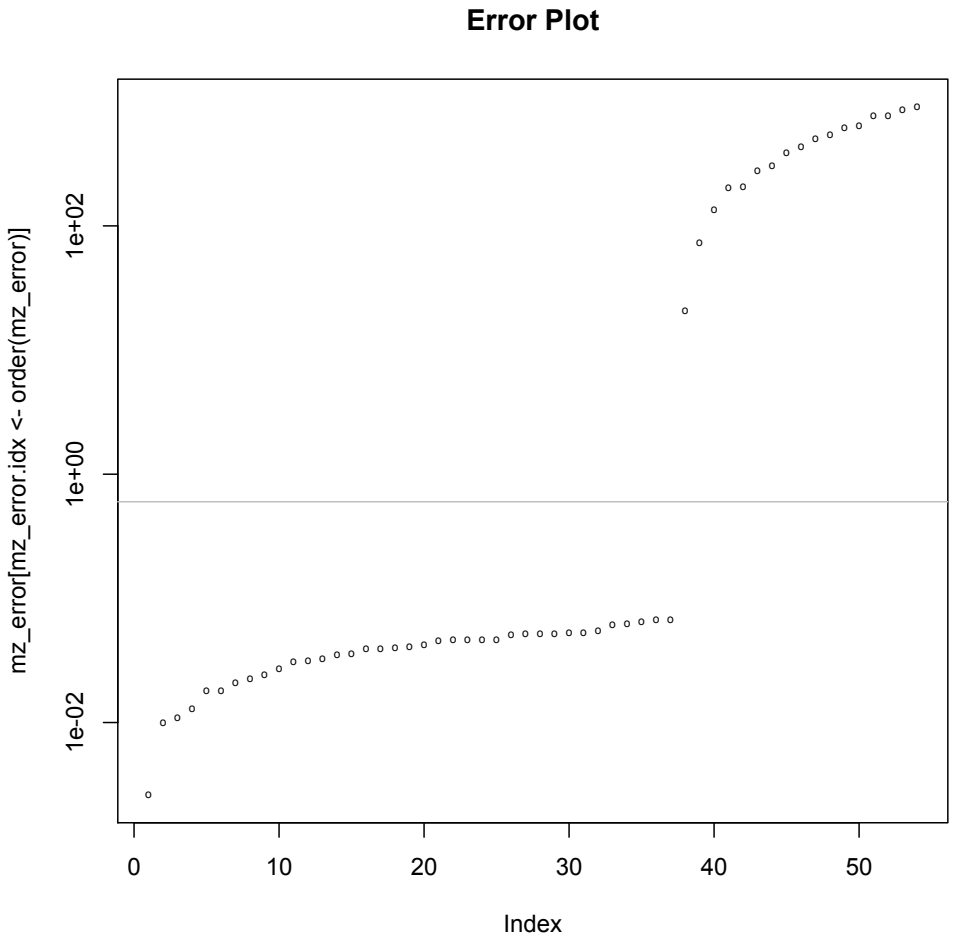
Troviamo quindi gli indici dei vicini più prossimi (*nearest neighbors*) dei dati ionici negli spettri e calcoliamo l'errore nelle corrispondenze m/z trovate nei passi precedenti:

```
> idx <- findNN(by_mz, mySpec$mz)
> mz_error <- abs(mySpec$mz[idx]-by_mz)
```

Per tracciare il grafico dell'errore utilizziamo la funzione `plot()`:

```
> plot(mz_error[mz_error.idx<-order(mz_error)], main="Error Plot", pch='o', cex=0.5,
      log='y')
> abline(h=0.6, col="grey")
```

Una sequenza peptidica viene rilevata negli spettri attraverso l'accoppiamento di frammenti di ioni. Si considera un hit se questo errore è inferiore ad una soglia chiamata `fragmentIonError`. Per calcolare



l'errore, utilizziamo gli ioni nella sequenza peptidica mediante la funzione `fragmentIon()`. Mappiamo i rapporti m/z calcolati per questi ioni sugli spettri con la corrispondenza più vicina (*nearest neighbor*) con la funzione `findNN()`. Possiamo scegliere gli ioni che devono essere mappati sui dati dello spettro (abbiamo usato gli ioni *b* e *y*). Se la differenza tra la massa calcolata e la massa misurata (nei dati MS) è inferiore alla soglia, viene definita come un hit.

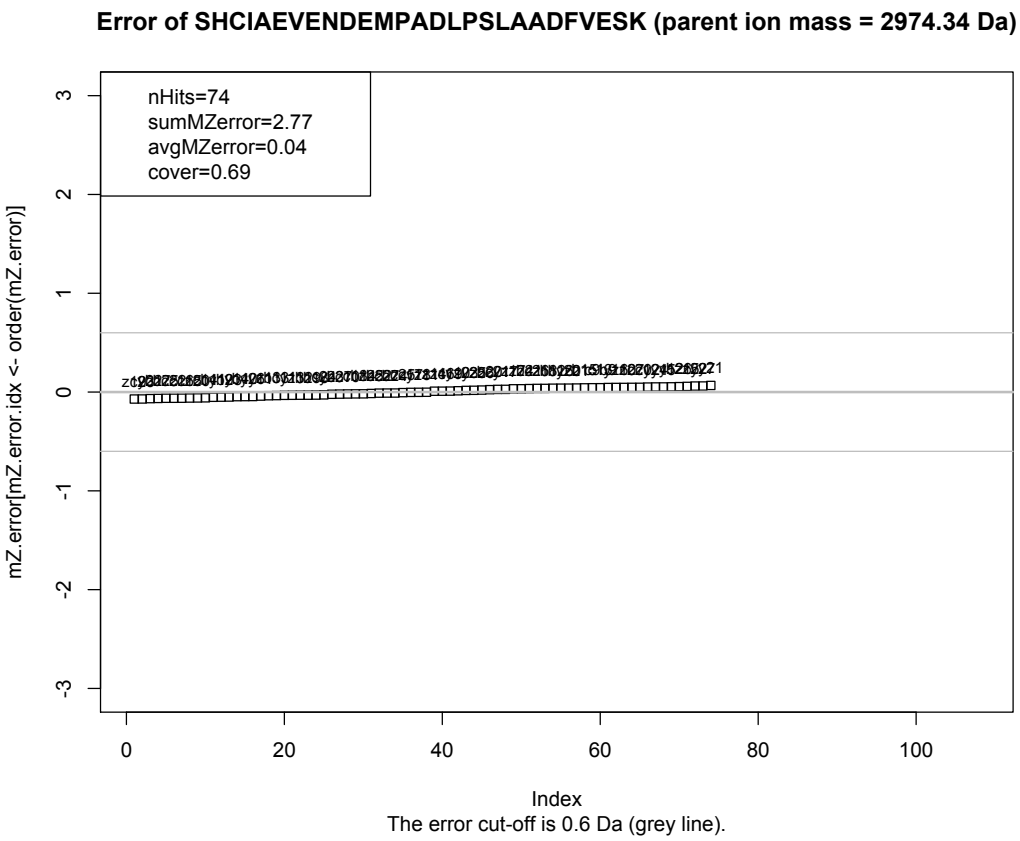
Un tasso di hit più alto e un tasso di errore più basso aumentano la fiducia nella presenza del peptide in esame. Possiamo vedere l'alto tasso di errore nella grafico precedente, che mostra l'errore di assegnazione tra i dati MS e gli ioni frammento dell'oggetto *myPeptide* caricato manualmente. Più gli ioni teorici corrispondono alle serie di ioni effettivamente misurati, più è probabile che lo spettro misurato provenga effettivamente dal peptide dedotto, e quindi la proteina viene identificata.

Per il calcolo dell'errore è anche disponibile la funzione `psm()`. Tuttavia, è consigliabile utilizzare la tecnica qui descritta per calcolare la corrispondenza tra lo spettro e il peptide poiché essa fornisce una maggiore flessibilità e risoluzione.

La funzione `psm()` può essere utilizzata nel seguente modo:

```
> match <- psm(myPeptide, mySpec)
```

Se confrontiamo i due grafici, la risoluzione nell'analisi è evidente.



I tipi di ioni nel frammento peptidico seguono un sistema di nomenclatura denominato *b*, *y*, *c* e *z*. Per saperne di più su questa nomenclatura, fare riferimento all'articolo *Proposal for a common nomenclature for sequence ions in mass spectra of peptides* di Roepstorff e Fohlman (https://www.scienceopen.com/document?vid=0362922f-ae77-42dc-8307-530561ad95f7). Ulteriori informazioni sull'identificazione dei peptidi nei dati MS sono disponibili nel libro *Protein Sequencing and Identification Using Tandem Mass Spectrometry* di Kinter e Nicholas (https://books.google.com/books?hl=it&lr=&id=1fmdXMG-72AC&oi=fnd&pg=PR7&dq=Protein+Sequencing+and+Identification+Using+Tandem+Mass+Spectrometry+di+Kinter+e+Nicholas&ots=EmKl5e7XvE&sig=N2hAtytw_EFR9WGm6h3cL4KvoXY).

13.9 Analisi di quantificazione delle proteine

Finora abbiamo parlato in termini di uso qualitativo dei dati MS. Tuttavia, l'analisi proteomica è passata dall'approccio qualitativo a quello quantitativo, e la tecnica della spettroscopia di massa ha giocato un ruolo chiave in questo passaggio. Questa sezione si concentrerà sugli aspetti quantitativi dell'analisi dei dati MS. Continueremo ad utilizzare la libreria *protViz*; i dati inclusi nel questo pacchetto serviranno come input per la quantificazione basata sui dati MS.

Iniziamo caricando la libreria *protViz* e i relativi dati nella sessione R:

```
> library(protViz)
> data(pgLFQfeature)
```

Osserviamo la struttura dell'oggetto *pgLFQfeature* per capire le sue caratteristiche e il suo contenuto:

```
> str(pgLFQfeature)
List of 9
 $ grouping                     : Named chr [1:24] "WT_NI" "WT_NI" "WT_NI" "WT_NI" …
  ..- attr(*, "names")= chr [1:24] "V13" "V14" "V15" "V16" …
 $ scoreNames                   : Named chr [1:5] "Normalized abundance" "Raw abundance"
"Intensity" "Sample retention time (min)" …
  ..- attr(*, "names")= chr [1:5] "V13" "V37" "V61" "V85" …
 $ output                       :'data.frame':  1239 obs. of  12 variables:
  ..$ #                          : int [1:1239] 2 63 4 8 5 10 11 6 9 168 …
  ..$ m/z                        : num [1:1239] 422 365 1112 706 489 …
  ..$ Retention time (min)       : num [1:1239] 58.9 20.9 147.8 82.6 38.2 …
  ..$ Retention time window (min): num [1:1239] 3.397 0.643 8.969 4.693 3.609 …
  ..$ Mass                       : num [1:1239] 842 1092 5557 1411 975 …
  ..$ Charge                     : int [1:1239] 2 3 5 2 2 2 3 3 3 2 …
  ..$ Max fold change            : chr [1:1239] "1.13762357642608" "1.9919014348284" …
  ..$ Highest mean condition     : chr [1:1239] "Inf_3h" "WT_NI" "WT_NI" "Inf_2h" …
  ..$ Lowest mean condition      : chr [1:1239] "Inf_1h" "Inf_1h" "Inf_3h" "Inf_1h" …
  ..$ Anova                      : num [1:1239] 0.79419 0.63518 0.06618 0.14803 0.00849 ...
  ..$ Included                   : chr [1:1239] "yes" "yes" "yes" "yes" ...
  ..$ Maximum CV                 : num [1:1239] 34.6 126.5 23.7 12.9 24.7 …
 $ peptideInfo                  :'data.frame':  1239 obs. of  10 variables:
  ..$ Notes                                    : logi [1:1239] NA NA NA NA NA NA …
…
 $ Normalized abundance         :'data.frame':  1239 obs. of  24 variables:
  ..$ 20120809_01_WT_NI_3_excl          : num [1:1239] 1.56e+07 1.29e+06 2.74e+08 2.16e+07 …
 $ Raw abundance                :'data.frame':  1239 obs. of  24 variables:
  ..$ 20120809_01_WT_NI_3_excl          : num [1:1239] 9.89e+06 8.16e+05 1.74e+08 1.37e+07 …
$ Intensity                    :'data.frame':   1239 obs. of  24 variables:
  ..$ 20120809_01_WT_NI_3_excl          : num [1:1239] 20853106 30445706 30740312 20000506 …
$ Sample retention time (min) :'data.frame':   1239 obs. of  24 variables:
  ..$ 20120809_01_WT_NI_3_excl          : num [1:1239] 58.8 20.9 149.5 83.6 38.8 …
 $ Best peptide spectral counts:'data.frame':  1239 obs. of  24 variables:
  ..$ 20120809_01_WT_NI_3_excl          : int [1:1239] 0 0 0 0 0 0 0 0 1 0 …
```

Si può notare che i dati contengono valori di abbondanza m/z e informazioni sulle proteine.

Per calcolare il volume di ogni proteina associata alle caratteristiche nei dati, eseguiamo la funzione `pgLFQtNpq()` che produce un oggetto di tipo matrice che fornisce i dettagli di abbondanza delle proteine in ogni campione:

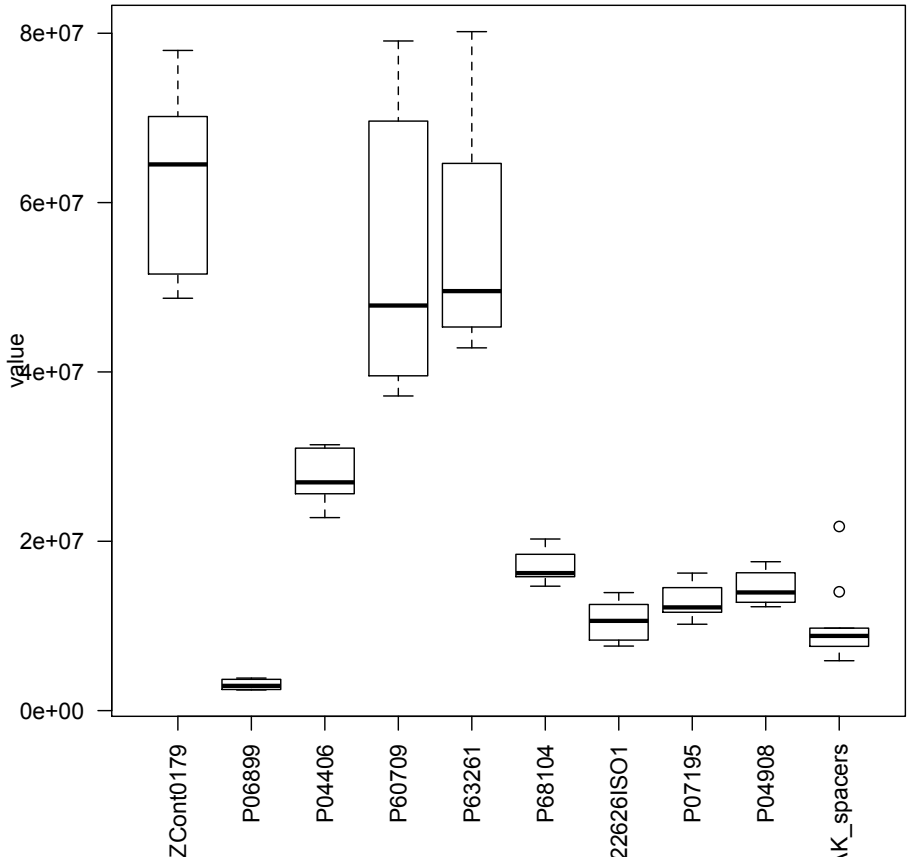
```
> myAbundance <- pgLFQtNpq(QuantitativeValue=pgLFQfeature$"Normalized abundance",
      peptide=pgLFQfeature$peptideInfo$Sequence,
      protein=pgLFQfeature$peptideInfo$Protein, N=2, plot=FALSE)
> class(myAbundance)
[1] "matrix"
```

La seguente schermata mostra le prime sei righe e otto colonne dell'oggetto *myAbundance*:

```
> head(myAbundance[1:6,1:8])
                    [,1]     [,2]     [,3]     [,4]     [,5]     [,6]     [,7]     [,8]
ZZ_FGCZCont0179 63903627 59534454 50423729 70165609 51562584 77970377 48701488 65112693
P06899           2436975  3692945  2494733  3841880  2488065  3833229  2572906  3275921
P04406          31252475 31000263 27021699 31399587 25355491 27057299 22794612 26195901
P60709          67652202 69624753 43747487 78348612 51937499 79087413 39542716 42958682
P63261          76490709 64628409 47910815 80188664 44058618 59892262 42843683 51180369
P68104          15832621 16368242 14930934 16116918 14694761 15813410 17859790 19490708
```

Creiamo infine un boxplot per le abbondanze come segue (solo per le prime 10 proteine):

```
> boxplot(t(myAbundance)[1:10,1:10], xlab="", ylab="value", las=2)
```



I dati contengono le proteine digerite provenienti da cellule umane *HeLa* infettate con il batterio *Shigella* coltivate durante un certo periodo di tempo. Sono strutturati nelle misure di sei repliche biologiche da quattro condizioni (Not_infected, Infected_1hr, Infected_2hr e Infected_3hr), avendo così 24 campioni in totale.

La precedente schermata mostra i boxplot con l'abbondanza delle proteine (sull'asse *y*) solo per i primi 10 peptidi visualizzati lungo l'asse *x*. La funzione `pgLFQtNpq()` adotta una strategia TopN, che utilizza solo le prime *N* caratteristiche di intensità per calcolare il volume proteico (nel nostro caso *N*=2). Questo approccio dovrebbe rivelare un valore quantitativo di proteine, che dovrebbe rendere la proteina stessa comparabile all'interno di una condizione. Ciò permette di stimare la stechiometria della proteina e semplifica la modellazione e i calcoli con i numeri di copie per cellula. L'argomento booleano `plot`, se impostato a `TRUE`, oltre al calcolo della matrice crea un grafico matriciale per le proteine.

L'articolo *Implementation and evaluation of relative and absolute quantification in shotgun proteomics with label-free methods* di Grossman *et al.* (http://www.sciencedirect.com/science/article/pii/S1874391910001703) fornisce informazioni sui dataset utilizzati in questa sezione e illustra anche la strategia TopN per rilevare l'abbondanza di proteine.

13.10 Analisi di più gruppi nei dati MS

Un'altra analisi importante per i dati MS (o di proteomica) dal punto di vista biologico è il confronto tra due gruppi. I gruppi possono essere campioni in due condizioni diverse, come il trattamento e il controllo. Il confronto avrà lo scopo di scoprire se alcune proteine o molecole hanno un'abbondanza significativamente diversa in uno dei gruppi. In questa sezione illustreremo l'approccio per affrontare tali situazioni.

Non abbiamo bisogno di pacchetti specifici per l'analisi, ma per i dati useremo il dataset **Isobaric Tags for Relative and Absolute Quantitation** (tag isobarici per la qantizzazione relativa e assoluta, iTRAQ) di *protViz*:

```
> library(protViz)
> data(iTRAQ)
> head(iTRAQ)
    prot       peptide  area113  area114  area115  area116  area117  area118  area119  area121
1 P02654   EFGNTLEDKAR  1705.43  1459.10   770.65  3636.40  3063.48  4046.73  2924.49  5767.87
2 P02654     EWFSETFQK  2730.41  1852.90  1467.65  2266.88  2269.57  3572.32  2064.82  2208.92
3 P02654     IKQSELSAK 28726.38 15409.81 19050.13 58185.02 51416.05 70721.05 38976.42 60359.72
4 P02654   LKEFGNTLEDK  4221.31  4444.28  2559.23  6859.71  5545.12 11925.66  6371.50 15656.92
5 P02654 LKEFGNTLEDKAR 20209.66 14979.02 12164.94 37572.56 30687.57 39176.99 34417.66 54439.22
6 P02654     EFGNTLEDK  4504.97  4871.88  2760.53  9213.41  6728.62 14761.96  7796.29 18681.60
                                                                                        desc
1 Apolipoprotein C-I OS=Homo sapiens GN=APOC1 PE=1 SV=1 sp|P02654|APOC1_HUMA
2 Apolipoprotein C-I OS=Homo sapiens GN=APOC1 PE=1 SV=1 sp|P02654|APOC1_HUMA
3 Apolipoprotein C-I OS=Homo sapiens GN=APOC1 PE=1 SV=1 sp|P02654|APOC1_HUMA
4 Apolipoprotein C-I OS=Homo sapiens GN=APOC1 PE=1 SV=1 sp|P02654|APOC1_HUMA
5 Apolipoprotein C-I OS=Homo sapiens GN=APOC1 PE=1 SV=1 sp|P02654|APOC1_HUMA
6 Apolipoprotein C-I OS=Homo sapiens GN=APOC1 PE=1 SV=1 sp|P02654|APOC1_HUMA
```

Per osservare il contenuto del dataset (11 colonne), visualizziamo le colonne digitando il seguente comando (si noti che le colonne da 3 a 10 hanno i valori dei dati e le colonne 1, 2 e 11 hanno informazioni sulla proteina):

```
> colnames(iTRAQ)
 [1] "prot"    "peptide" "area113" "area114" "area115" "area116" "area117" "area118" "area119" "area121"
[11] "desc"
```

I dati hanno due gruppi in condizioni diverse. Creiamo due vettori vuoti per contenere i dati di queste due condizioni:

```
> condition_1 <- numeric()
> condition_2 <- numeric()
```

Le colonne da 3 a 6 nei dati appartengono ad un gruppo, e da 7 a 10 appartengono all'altro. Riempiamo i due vettori creati in precedenza con i valori delle colonne corrispondenti:

```
> for (i in c(3:6)) {
+ condition_1 = cbind(condition_1,
+ asinh(tapply(iTRAQ[,i], paste(iTRAQ$prot), sum, na.rm=TRUE)))
+ }
> for (i in 7:10) {
+ condition_2 = cbind(condition_2,
+ asinh(tapply(iTRAQ[,i], paste(iTRAQ$prot), sum, na.rm=TRUE)))
+ }
```

Al termine avremo i dati di cinque proteine per due gruppi con quattro campioni in ogni gruppo.

Ora, per verificare la significatività della differenza tra i due gruppi, eseguiamo un *t*-test. A tal fine, creiamo prima un vettore vuoto in cui memorizzare il risultato di ciascun test:

```
> pv <- c()
```

Eseguiamo quindi il *t*-test per ogni proteina e aggiungiamo i *p*-value al vettore creato:

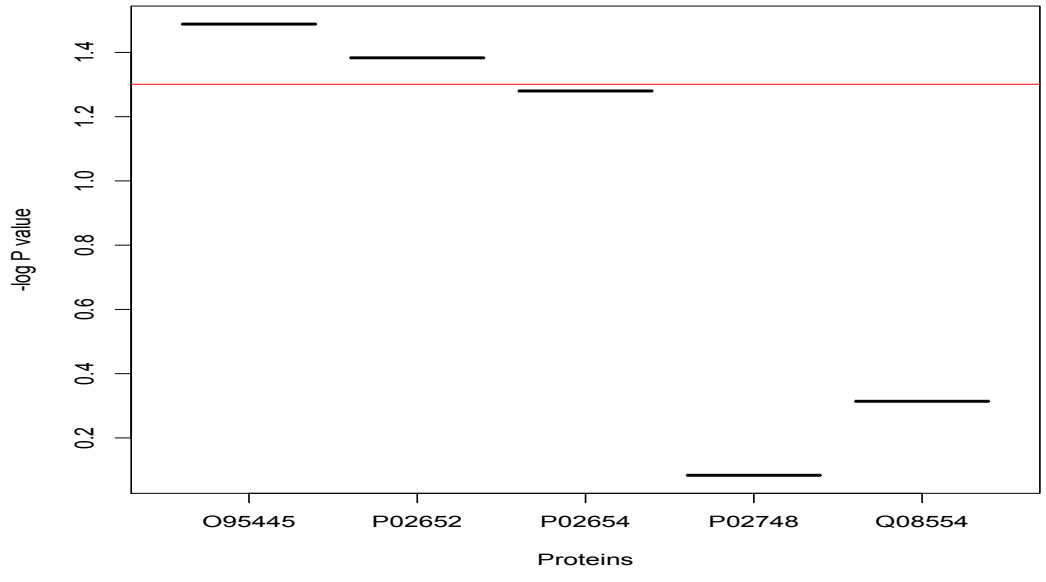
```
> for (i in 1:nrow(condition_2)) {
  pv = c(pv,  t.test(as.numeric(condition_1[i,]), as.numeric(condition_2[i,]))$p.value)
  }
> pv
[1] 0.03249315 0.04138090 0.05247586 0.82471440 0.48526520
```

I *p*-value sono i risultati del test che definiscono se la differenza di abbondanza dei peptidi tra i due gruppi è significativo o meno. Convenzionalmente, si utilizza un cut-off di 0,05 per determinare la significatività: si possono vedere due proteine, O95445 e P02652, che mostrano una differenza significativa con *p*-value di circa 0,032 e 0,041. Se necessario, si può effettuare una correzione con test multipli.

I dati iTRAQ utilizzati in questa sezione provengono da due gruppi, i casi e i controlli, per l'artrite reumatoide. I dati contengono i peptidi di cinque proteine identificate. In primo luogo, abbiamo misurato ogni proteina sommando i peptidi corrispondenti in ogni campione e registrando i risultati nei due vettori *condition_1* e *condition_2*. Quindi abbiamo calcolato i punteggi del *t*-test, che ha restituito il *p*-value della differenza nella misurazione di ogni proteina nelle due diverse condizioni: se il valore restituito per una proteina è inferiore a 0,05 accettiamo che la differenza sia significativa.

Possiamo creare un grafico per illustrare i risultati (usiamo il $-\log_{10}$ del *p*-value per una migliore visualizzazione). Le due proteine sono le Apolipoproteine (possiamo controllare gli ID UniProt sul sito https://www.uniprot.org):

```
> names(pv) <- levels(iTRAQ$prot)
> plot(x=factor(names(pv)), y=-log10(pv), xlab="Proteins", ylab="-log P value")
> abline(h=-log10(0.05), col="red")
```



Nella grafico precedente la linea rossa indica la soglia 0,05 del *p*-value ($-\log_{10}(0.05) \approx 1.3$) e l'asse *x* indica le proteine (con il loro ID UniProt).

L'articolo *Discovery of serum proteomic biomarkers for prediction of response to infliximab (a monoclonal anti-TNF antibody) treatment in rheumatoid arthritis: An exploratory analysis* di Ortea *et al*. (http://www.sciencedirect.com/science/article/pii/S1874391912006550) fornisce maggiori informazioni sul dataset che abbiamo utilizzato.

13.11 Visualizzazioni utili per l'analisi dei dati MS

In questa sezione illustreremo alcune interessanti visualizzazioni dei dati MS che includono la *peak labeling visualization* (visualizzazione delle etichette dei picchi), il *multiple spectra plotting* (grafico di spettri multipli) e la *fragment ion visualization* (visualizzazione di frammenti di ioni). Utilizzeremo gli stessi dati e le stesse librerie già viste in precedenza.

Come al solito, carichiamo la libreria *MALDIquant* e il primo spettro del dataset fiedler2009subset eseguendo la relativa pre-elaborazione:

```
> library(MALDIquant)
> data("fiedler2009subset", package = "MALDIquant")
> spec <- fiedler2009subset[[1]]
> spec <- transformIntensity(spec, method = "sqrt")
> spec <- smoothIntensity(spec, method = "MovingAverage")
> spec <- removeBaseline(spec)
```

Tracciamo il grafico inziale dello spettro:

```
> plot(spec)
```

Quindi, eseguiamo il rilevamento dei picchi e aggiungiamoli come punti del grafico:

```
> p <- detectPeaks(spec)
> points(p)
```

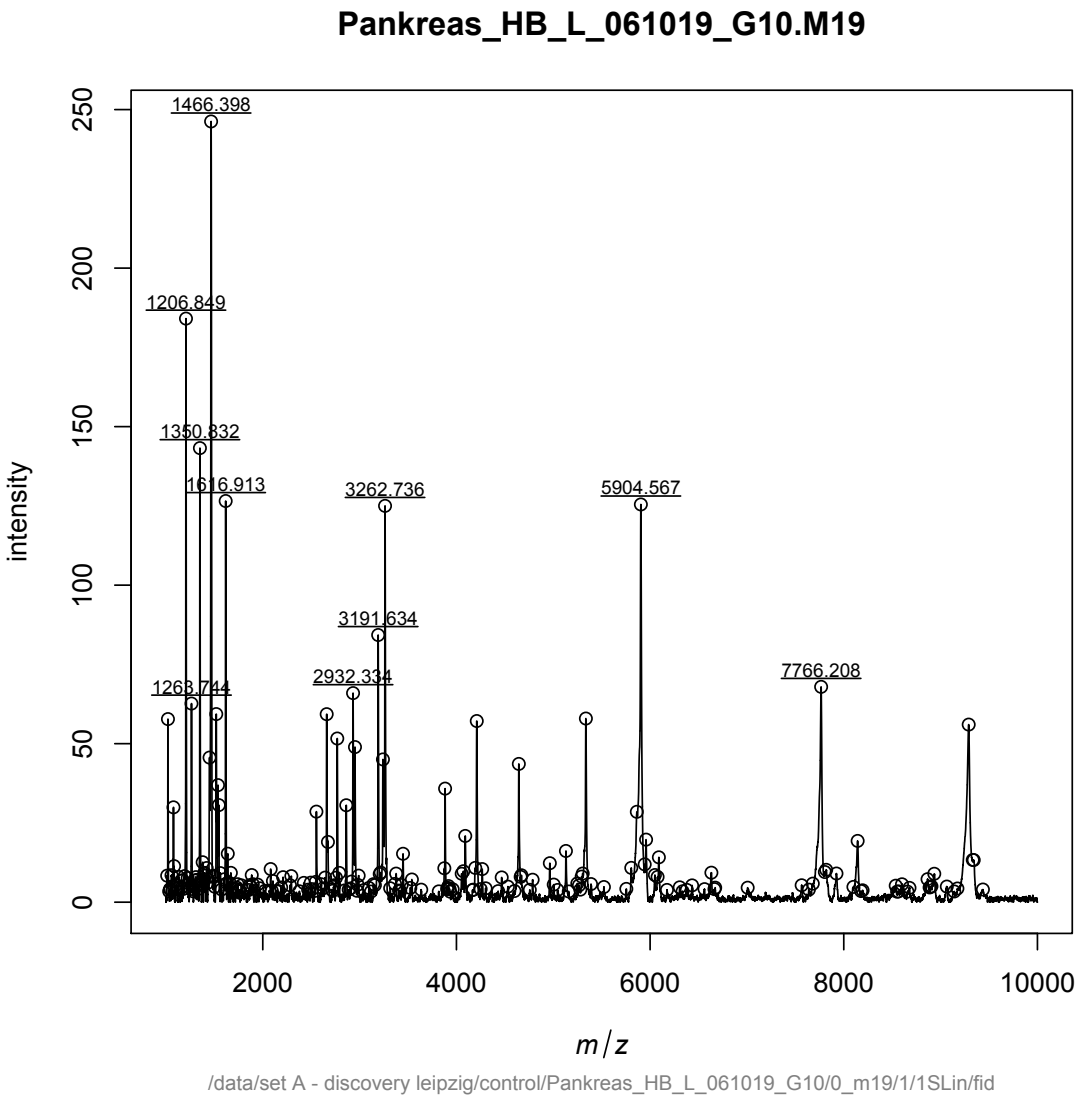
Etichettiamo ora i primi 10 picchi dello spettro nel grafico con le loro masse. Per identificarli, dobbiamo prima ordinarli:

```
> top10 <- intensity(p) %in% sort(intensity(p), decreasing = TRUE)[1:10]
```

Quindi eseguiamo la funzione `labelPeaks()` per aggiungere le etichette nel grafico:

```
> labelPeaks(p, index = top10)
```

La seguente schermata mostra il grafico con i picchi etichettati:



La prossima visualizzazione riguarda il grafico di spettri diversi sulla stessa schermata, che può essere usato specialmente per mostrare l'allineamento degli spettri. Anche in questo caso utilizziamo gli spettri nel dataset fiedler2009subset (li assegniamo alla variabile *spectra* e il loro numero alla variabile *l*):

```
> spectra <- fiedler2009subset
> l <-length(spectra)
```

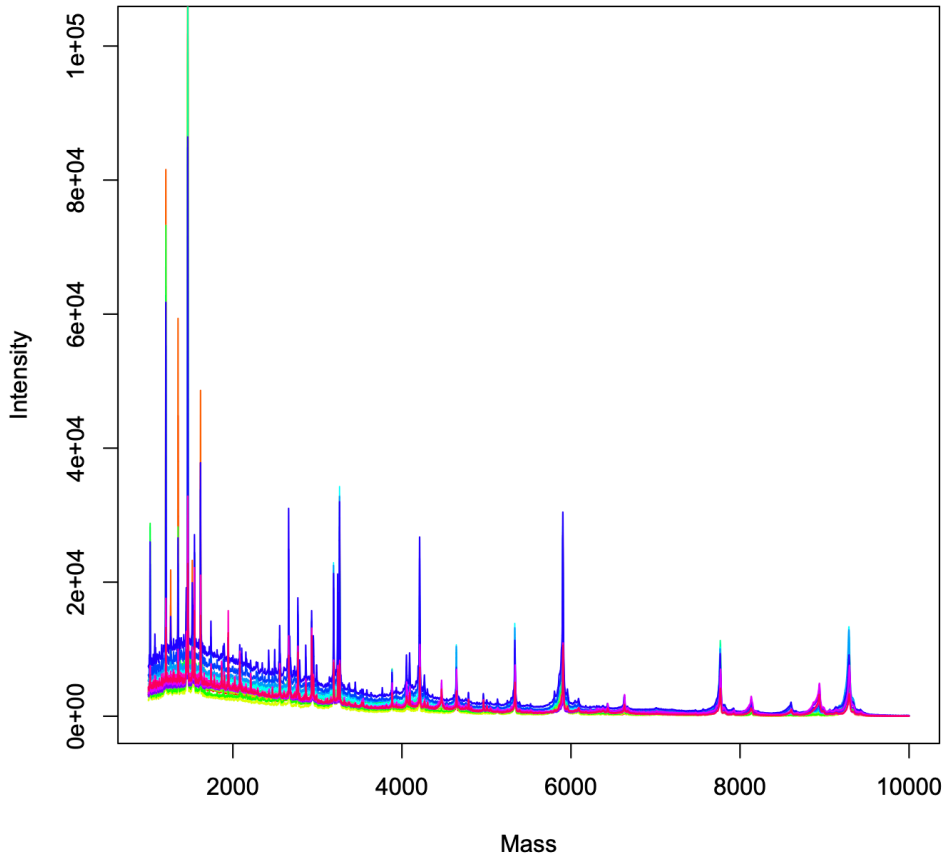
Iniziamo un grafico per il primo spettro come segue (impostiamo l'argomento `type` della funzione `plot()` a `"n"` per non tracciare ancora la grafica):

```
> plot(spectra[[1]]@mass, spectra[[1]]@intensity, type = "n", ,xlab = "Mass",
      ylab = "Intensity")
```

Aggiungiamo le righe in diversi colori per tutti gli altri dati degli spettri della lista *spectra* tramite un ciclo come segue (digitare `?rainbow` in R per informazioni sui colori utilizzati):

```
> for (i in 1:l) {
+ lines(spectra[[i]]@mass, spectra[[i]]@intensity, type = "l", col = rainbow(l)[i])
+ }
```

Nella seguente schermata, possiamo vedere le intensità degli spettri multipli tracciate insieme con colori diversi:



Il grafico successivo è quello della visualizzazione dei rapporti m/z per i diversi ioni in un peptide. Impostiamo manualmente i peptidi (prendiamo i dati dalla precedente query effettuata sul sito ExPASy per ALBU_HUMAN):

```
> myPeptide <- c("SHCIAEVENDEMPADLPSLAADFVESK", "LVRPEVDVMCTAFHDNEETFLK",
      "ALVLIAFAQYLQQCPFEDHVK")
```

Calcoliamo la massa dello "ione genitore" per questi peptidi come segue:

```
> pim <- parentIonMass(myPeptide)
```

Calcoliamo inoltre gli ioni frammento per i nostri peptidi (per ogni peptide si ottengono gli ioni *b*, *y*, *c* e *z*):

```
> fi <- fragmentIon(myPeptide)
```

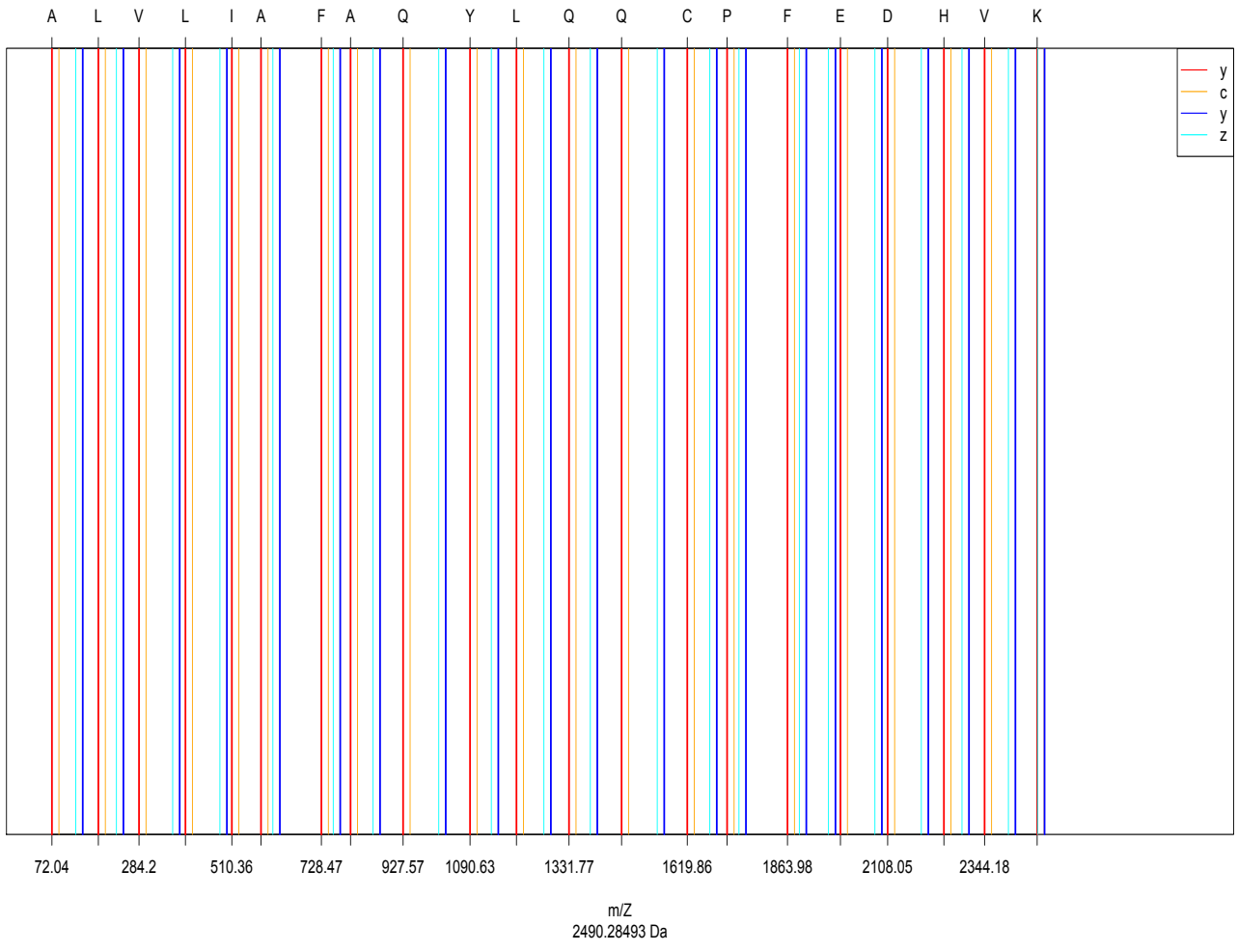
Possiamo visualizzare i rapporti m/z per ognuno di questi tipi di ioni in un colore diverso per ogni peptide (ad esempio il terzo) come segue:

```
> i = 3 # plot the 3rd peptide
> plot(0, 0, xlab = 'm/Z',  ylab = '',  xlim = c(min(c(fi[i][[1]]$b,fi[i][[1]]$y)),
      max(c(fi[i][[1]]$b,fi[i][[1]]$y))+350), ylim = c(0,1), type = 'n',
      axes = FALSE, sub = paste(pim[i],"Da")) # creates the plot area
> box() # adds a box to the plot area
> axis(1, fi[i][[1]]$b, round(fi[i][[1]]$b,2)) # add axis label
> pepSeq <- strsplit(myPeptide[i],"") # splits the peptide residues
> axis(3, fi[i][[1]]$b, pepSeq[[1]]) # adds residue labels at top
> abline(v = fi[i][[1]]$b, col='red', lwd=2) # adds plot for the b type ions
> abline(v = fi[i][[1]]$c, col='orange') # adds plot for the c type ions
> abline(v = fi[i][[1]]$y, col='blue', lwd=2) # adds plot for the y type ions
> abline(v = fi[i][[1]]$z, col='cyan') # adds plot for the z type ions
```

Aggiungiamo una legenda al grafico:

```
> legend("topright", legend = c("y","c","y", "z"),
      col = c("red","orange","blue","cyan"), lty = 1, merge = TRUE)
```

La schermata seguente mostra il rapporto m/z per diversi tipi di ioni (in diversi colori) per il primo peptide, "ALVLIAFAQYLQQCPFEDHVK":



Il primo grafico che abbiamo visto è l'etichettatura dei picchi mediante la funzione `labelPeaks()` del pacchetto *MALDIquant* (possiamo anche utilizzare la funzione `peakplot()` del pacchetto *protViz*; per ulteriori informazioni consultare il file di aiuto del pacchetto *protViz*).

Il secondo grafico sfrutta semplicemente le funzioni `plot()` e `lines()` di R applicate iterativamente. Ad ogni ciclo viene aggiunto al grafico un dataset che può essere usato per mostrare uno spettro dopo e prima dell'allineamento per i picchi di due diverse condizioni.

La terza e ultima visualizzazione presentata nella sezione mostra la posizione degli ioni frammentati di un peptide nello spazio m/z, con i diversi tipi di ioni in differenti colori.

14. Analisi dei dati NGS (Next Generation Sequencing)

Il progetto HGP (Human Genome Project) mirava a determinare le sequenze che compongono il DNA umano. È stato completato nel 2003, prima del previsto e nei limiti del budget, e ha non solo ha dato il via ai numerosi progetti di sequenziamento, ma ha anche incoraggiato lo sviluppo di tecnologie in grado di consentire un sequenziamento più rapido ed economico dei genomi. Un unico genoma umano non è sufficiente per interpretare le informazioni genetiche sulla malattia; sono piuttosto necessari molti genomi per tali studi. Le tecnologie utilizzate durante l'HGP erano lente e costose. La richiesta di metodi di sequenziamento più economici e veloci ha guidato lo sviluppo del sequenziamento di prossima generazione (NGS, Next Generation Sequencing). Le piattaforme NGS eseguono un sequenziamento massivo, dove milioni di frammenti di DNA di un singolo campione sono esaminati in parallelo facilitando il sequenziamento ad alta velocità. Questo permette di sequenziare un intero genoma più velocemente e ad un costo inferiore.

NGS è un termine usato per descrivere una serie di diverse tecnologie moderne di sequenziamento. Esso comprende alcune tecnologie ormai diffuse, come:

- Sequenziamento *Illumina* (Solexa)
- Sequenziamento *Roche 454*
- *Ion Torrent* (sequenziamento protonico e PGM)
- Sequenziamento *SOLiD*

Queste tecnologie permettono un sequenziamento del DNA e dell'RNA più veloce e meno costoso rispetto al sequenziamento Sanger utilizzato in precedenza, e come tali hanno rivoluzionato lo studio della genomica e della biologia molecolare. Questo capitolo tratterà i risultati ottenuti da queste tecnologie e da alcune tecnologie correlate. Per saperne di più sulle tecnologie NGS e la loro applicazione, si può fare riferimento all'articolo *Sequencing technologies — the next generation* di Metzker (http://www.nature.com/nrg/journal/v11/n1/full/nrg2626.html).

Uno dei formati più popolari di dati di sequenza NGS è il formato FASTQ. Prima di approfondire in dettaglio l'analisi dei dati NGS, è utile esaminare il formato di un file FASTQ, composto da quattro righe. La prima riga descrive il nome della sequenza, la seconda contiene la sequenza stessa, la terza informazioni opzionali sulla sequenza e la quarta la misura di confidenza o precisione delle basi. La seguente schermata mostra un'istanza del formato dati FASTQ:

```
@ERR056989.2 GRJP5WI01AODNT/2
GCGAAGTAGCATGAGCAGGACGCGATGACGAGCAGCAGGAGCATGACCATGAGCGTCTGCGCGGCAGCGC
+
:9;00012333358995../07;=;;;;=?@@@@@??<<<=@??;;;@@@BB@@??==@?;511111371
```

Poiché tutte le altre informazioni nel formato FASTQ sono piuttosto semplici e dovrebbero essere facilmente comprensibili, parliamo delle misure di qualità. La qualità dei dati in NGS è misurata in termini di una metrica chiamata *Phred score*. Un punteggio Phred viene assegnato ad ogni base durante il processo di sequenziamento; pertanto, nei dati FASTQ abbiamo un carattere corrispondente ad ogni base. Matematicamente, il *Phred score* (*Q*) è definito come segue:

$$Q = -10 \cdot \log_{10}(P)$$

Nella formula precedente, *P* è la probabilità di errore stimata nell'attribuzione della base (processo di assegnazione delle basi ai picchi). Questo stabilisce una relazione logaritmica tra la qualità e l'errore della base, che permette di lavorare con errori molto piccoli (vicini allo zero) e di trattarli con alta precisione numerica. Ad esempio il 99,999 per cento (1 su 100.000) di precisione nell'attribuzione delle basi corrisponde a un punteggio pari a 50 ($-10 \cdot \log_{10}(0{,}00001) = 50$). Per saperne di più sul *Phred score* fare riferimento all'articolo *Base-Calling of Automated Sequencer Traces Using Phred. II. Error Probabilities* di Ewing e Green (http://genome.cshlp.org/content/8/3/186.long).

Il formato FASTQ visualizza questa misura di qualità in termini di caratteri ASCII (di solito viene utilizzato il (33+*Q*)-esimo carattere ASCII per rappresentare un valore *Q*). Ecco perché vediamo tali caratteri in ogni quarta riga di un file FASTQ.

I dati FASTQ consentono di memorizzare la sequenza e le informazioni di qualità per ogni lettura in un formato compatto basato su testo. Per saperne di più sul formato FASTQ, consultare l'articolo The *Sanger FASTQ file format for sequences with quality scores, and the Solexa/Illumina FASTQ variants* di Cock *et al*. (http://nar.oxfordjournals.org/content/38/6/1767.full).

14.1 Interrogazione del database SRA

"Sequence Read Archive" (SRA) è un database che contiene dati di sequenziamento del DNA in termini di brevi letture generate da sequenziamenti ad alta velocità ottenuti attraverso diverse piattaforme. Le sequenze sono solitamente di lunghezza inferiore a 1.000 coppie di basi. Il database è disponibile all'indirizzo https://www.ncbi.nlm.nih.gov/sra. Questa sezione mostra come interrogare e accedere ai dati che risiedono nel database SRA dall'interno di R.

Per prima cosa, installiamo e carichiamo la libreria *SRAdb* del repository Bioconductor nella sessione R:

```
> BiocManager::install("SRAdb")
> library(SRAdb)
```

Scarichiamo ora il file dei metadati dal server SRA (trattandosi di un file di alcuni Gigabytes potrebbe richiedere un po' di tempo):

```
> sqlFile <- getSRAdbFile()
```

Se invece lo abbiamo già scaricato nella directory di lavoro, ne impostiamo semplicemente il percorso:

```
> sqlFile = paste(getwd(),"/SRAmetadb.sqlite",sep="")
```

Creiamo una connessione al file SQLite per l'interrogazione:

```
> sraCon <- dbConnect(SQLite(),sqlFile)
```

Utilizziamo la seguente funzione per determinare i campi nei dati:

```
> sraTables <- dbListTables(sraCon)
> dbListFields(sraCon,"study")
 [1] "study_ID"              "study_alias"          "study_accession"      "study_title"
 [5] "study_type"            "study_abstract"       "broker_name"          "center_name"
 [9] "center_project_name"   "study_description"    "related_studies"      "primary_study"
[13] "sra_link"              "study_url_link"       "xref_link"            "study_entrez_link"
[17] "ddbj_link"             "ena_link"             "study_attribute"   "submission_accession"
[21] "sradb_updated"
```

Ora, impostiamo la nostra prima ricerca per ottenere gli *accession* e i titoli dello studio SRA dove il tipo di studio contiene la parola chiave `embryo`. L'interrogazione può essere vista come una tipica query SQL (che restituisce due record):

```
> myHit <- dbGetQuery(sraCon, "select study_accession,study_title from study where
study_description like '%embryo'")
> myHit
  study_accession
1       SRP001353
2       ERP014605
study_title
1 GSE17621: RNA-Seq of Drosophila developmental stage 16-18 hr
2 Ultra-deep sequencing of ribosome-associated poly-adenylated RNA in early
Drosophila embryo reveals hundreds of conserved translating sORFs
```

Effettuiamo una ricerca a testo libero del termine "brain" (la query restituisce 309.071 righe e 23 colonne in totale, di cui visualizziamo solo una prima parte):

```
> myHit <- getSRA(search_terms = "brain", out_types = c('run','study'), sraCon)
> dim(myHit)
[1] 309071     23
> head(myHit)
  run_alias       run   run_date updated_date   spots     bases run_center experiment_name …
1 DRR000060 DRR000060 2008-09-25   2015-02-06 6219174 223890264     UT-MGS       DRX000023 …
2 DRR000061 DRR000061 2008-09-25   2015-02-06 6082236 218960496     UT-MGS       DRX000023 …
3 DRR000063 DRR000063 2008-09-25   2015-02-06 6062563 218252268     UT-MGS       DRX000023 …
4 DRR000062 DRR000062 2008-09-25   2015-02-06 6095377 219433572     UT-MGS       DRX000023 …
5 DRR000072 DRR000072 2009-02-02   2015-02-06 9192512 330930432     UT-MGS       DRX000026 …
6 DRR000073 DRR000073 2009-02-02   2015-02-06 9306929 335049444     UT-MGS       DRX000026 …
…
```

Combiniamo le parole chiave per una ricerca composta come segue:

```
> myHit <- getSRA(search_terms='"ALZHEIMERS" OR "EPILEPSY"', out_types=c('sample'), sraCon)
> dim(myHit)
[1] 29916    10
> myHit[1:4,1:6]
                    sample_alias     sample taxon_id common_name anonymized_name individual_name
1 qiita_sid_10317:10317.000069625 ERS1846633   408170        <NA>              NA              NA
2 qiita_sid_10317:10317.000066637 ERS1846567   408170        <NA>              NA              NA
3 qiita_sid_10317:10317.000062125 ERS1846482   408170        <NA>              NA              NA
4 qiita_sid_10317:10317.000004786  ERS915949   408170        <NA>              NA              NA
```

Il pacchetto *SRAdb* rende molto più facile e veloce l'accessibilità ai metadati associati alla registrazione, allo studio, al campione, all'esperimento e all'esecuzione del sequenziamento. La funzione `dbConnect()` collega prima di tutto l'ambiente R a questi sistemi di database locali, e tutte le richieste vengono elaborate localmente sulla base dei dati scaricata dal sito SRA di NCBI. In questa sezione le ricerche che abbiamo eseguito con la funzione `dbGetQuery()` vengono passate sotto forma di query SQL, secondo lo schema "SELECT … FROM … WHERE …". Questa parte richiede in realtà il pacchetto *RSQLite*, che viene installato automaticamente quando si installa il pacchetto *SRAdb*. La funzione `getSRA()` effettua una ricerca di testo completo nei dati SRA (sempre via *RSQLite*) e recupera i dati nei campi selezionati per la query.

Il manuale SRA NCBI (https://www.ncbi.nlm.nih.gov/books/NBK47528/) fornisce informazioni dettagliate sulla banca dati SRA nel framework Entrez. L'articolo *SRAdb: query and use public next-generation sequencing data from within R* di Zhu *et al*. all'indirizzo https://www.biomedcentral.com/1471-2105/14/19, fornisce informazioni sul pacchetto *SRAdb*.

14.2 Download dei dati dal database SRA

Abbiamo appena visto come interrogare il database SRA. Possiamo utilizzare il pacchetto *SRAdb* anche per scaricare i dati FASTQ in base alle nostre richieste. Questa sezione mostra come affrontare questo problema.

Iniziamo con il caricamento della libreria *SRAdb*:

```
> library(SRAdb)
```

Impostiamo una ricerca. Ad esempio, usiamo la stessa query della sezione precedente per cercare ALZHEIMERS oppure EPILEPSY (si noti che il passo richiede la connessione SQLite *sraCon* creata nella sezione precedente):

```
> myHit <- getSRA(search_terms='"ALZHEIMERS" OR "EPILEPSY"', out_types=c('sample'), sraCon)
```

Scegliamo ora due dei risultati (i campioni ERS354366 e SRS266589) e cerchiamo le relative informazioni utilizzando la funzione `sraConvert()`:

```
> conversion <- sraConvert(c('ERS354366','SRS266589'), sra_con = sraCon)
> conversion
     sample submission     study experiment       run
1 ERS354366  ERA252779 ERP003987  ERX324104 ERR351268
2 SRS266589  SRA046961 SRP008797  SRX100465 SRR351672
3 SRS266589  SRA046961 SRP008797  SRX100465 SRR351673
```

Per ottenere informazioni su uno degli esperimenti, possiamo utilizzare la funzione `getSRAinfo()`:

```
> rs <- getSRAinfo(c("SRX100465"), sraCon, sraType = "sra")
```

Oppure possiamo accedere alle informazioni desiderate tramite il browser web:

```
> browseURL("https://www.ncbi.nlm.nih.gov/sra/?term=SRX100465")
```

Questo ci fornisce i dettagli dell'esperimento:

Full

SRX100465: GSM803420: HudsonAlpha_ChipSeq_GM12878_MEF2C_(SC-13268)_v041610.1
2 ILLUMINA (Illumina Genome Analyzer) runs: 48.8M spots, 1.8G bases, 1.9Gb downloads

Submitted by: Gene Expression Omnibus (GEO)

Study: GSE32465: Transcription Factor Binding Sites by ChIP-seq from ENCODE/HAIB
• SRP008797 • All experiments • All runs

Sample: HudsonAlpha_ChipSeq_GM12878_MEF2C_(SC-13268)_v041610.1
SAMN00738382 • SRS266589 • All experiments • All runs
Organism: Homo sapiens

Library:
Name: GSM803420: HudsonAlpha_ChipSeq_GM12878_MEF2C_(SC-13268)_v041610.1
Instrument: Illumina Genome Analyzer
Strategy: ChIP-Seq
Source: GENOMIC
Selection: ChIP
Layout: SINGLE

Spot descriptor:
1 forward

Experiment attributes:
GEO Accession: GSM803420

Links:
External link: GEO Web Link

Runs: 2 runs, 48.8M spots, 1.8G bases, 1.9Gb

Run	# of Spots	# of Bases	Size	Published
SRR351672	22,020,897	792.8M	812.3Mb	2011-10-11
SRR351673	26,765,383	963.6M	1.1Gb	2011-10-11

ID: 116951

Per scaricare i dati di nostro interesse utilizziamo la funzione `getSRAfile()` come segue (potrebbe richiedere un certo tempo per il download):

```
> getSRAfile(c("SRR351672", "SRR351673"), sraCon, fileType ='fastq')
        run submission     study    sample experiment
1 SRR351672  SRA046961 SRP008797 SRS266589  SRX100465
2 SRR351673  SRA046961 SRP008797 SRS266589  SRX100465
                                                                  ftp
1 ftp://ftp.sra.ebi.ac.uk/vol1/fastq/SRR351/SRR351672/SRR351672.fastq.gz
2 ftp://ftp.sra.ebi.ac.uk/vol1/fastq/SRR351/SRR351673/SRR351673.fastq.gz
```

Questa sezione è un'estensione della precedente. La differenza è nella conversione degli ID dai risultati della query per la ricerca a testo libero, che fornisce gli ID di accesso per i risultati recuperati dal database SRA. Successivamente, abbiamo raccolto le informazioni sui dati scaricati. Infine, la funzione `getSRAfile()` recupera il file dal database SRA, scaricando effettivamente i dati dall'host remoto. Possiamo anche usare l'indirizzo FTP direttamente per scaricare il file. Per esaminare altri metodi per recuperare i file, consultare il manuale del pacchetto *SRAdb* disponibile all'indirizzo https://www.bioconductor.org/packages/release/bioc/manuals/SRAdb/man/SRAdb.pdf.

14.3 Lettura di file FASTQ in R

Un file FASTQ, come detto in precedenza, è costotuito da una successione di diverse brevi sequenze di lettura. È il formato universalmente accettato nella comunità NGS e per la maggior parte dei programmi di allineamento come Bowtie, che utilizza questi file come input. Per poter analizzare i dati, dobbiamo leggerli nell'area di lavoro R. Questa sezione affronta il problema della lettura dei file FASTQ in R.

Per proseguire avremo bisogno di un file FASTQ da leggere e del pacchetto *ShortRead*. Possiamo scaricare un file dal database SRA oppure utilizzare il file di esempio del pacchetto *ShortRead*. La seguente schermata mostra un file FASTQ:

Scarichiamo il file SRR000648.fastq mediante la funzione `getFASTQfile()`:

```
> getFASTQfile(in_acc=c("SRR000648"),sraCon,destDir=getwd(),srcType='ftp',ascpCMD=NULL)
Files are saved to:
'/Users/…/Downloads/tmp'

provo con l'URL 'ftp://ftp.sra.ebi.ac.uk/vol1/fastq/SRR000/SRR000648/SRR000648.fastq.gz'
Content type 'unknown' length 84837 bytes (82 KB)
==================================================
        run submission     study    sample experiment
1 SRR000648  SRA000241 SRP000098 SRS000290  SRX000122
                                                                    ftp
1 ftp://ftp.sra.ebi.ac.uk/vol1/fastq/SRR000/SRR000648/SRR000648.fastq.gz
```

Installiamo e carichiamo il pacchetto *ShortRead*:

```
> BiocManager::install("ShortRead")
> library(ShortRead)
```

Utilizziamo la funzione `readFastq()` per leggere come oggetto R il file FASTQ scaricato. Si noti che i file FASTQ sono solitamente molto grandi e potrebbero richiedere molta memoria durante la lettura (potrebbero essere necessari circa 8 GB di RAM):

```
> myFastq <- readFastq(getwd(), pattern="SRR000648.fastq")
```

Esaminiamo le prime quattro righe del file con la funzione R `readLines()`:

```
> readLines("SRR000648.fastq", 4)
[1] "@SRR000648.1.2 EUEMSW405C5XBJ length=262"
[2]
"TTGCGTGGGAGCATTTTATTTCATTTACAAGGCCGTGGGAAGTCATAAAATTAATTTCGGCATAACCATGTGTTCAGTGTGTAT
ACAGTGAGAGTAGCTCCCGCTGCCCATTGTTTCAAGACTTCTGCTTCTTGTGGGTTGTGGCTGGAATGAAGACACCGGTCTTCAC
TGCAATAAGGTTGTTGTTTCTGATTTCCAACCCCAAATCCGTCCAGCTTCTGAGACACGCAACAGGGGATAGGCAAGGCACACAG
GGGATAGG"
[3] "+SRR000648.1.2 EUEMSW405C5XBJ length=262"
[4] "C==<==FB1==<<HD8+<FB1==FB1<=B;C<C<<<EA/
A9=;<<=FC6'C=C=FB1<C<<<=C=C<<=<==C=<<<=<<<=<====<===<=<==<<;<FB0==<<GB1<C=<FB0=C<=:=C=
<===C=;B;;=B=)C<8<?
6<=B;B;<<B;=8:;C<C=;;C=<<<===C;;A9C<C=8B;;FB0<<=<GC3C=C<FB4$A=(=C=:<A9=;<B<=8<8:99<;<
=A<<8HD8,<<<A<:A;B<97==<<HD8,<9;A;"
```

La funzione `readFastq()` legge tutti i file di formato FASTQ che corrispondono ad un determinato schema (argomento `pattern`) nella directory da scansionare (nel nostro caso `getwd()`, che corrisponde alla directory di lavoro dove abbiamo scaricato in precedenza il file). La funzione restituisce in un oggetto R (nel nostro caso *myFastq*) la sequenza e le misure di qualità del file FASTQ letto.

Maggiori informazioni sul pacchetto *ShortRead* sono disponibili agli indirizzi http://darrenjw.wordpress.com/2010/11/29/a-quick-introduction-to-the-bioconductor-shortread-package-for-the-analysis-of-ngs-data/ e http://www.bioconductor.org/packages/release/bioc/html/ShortRead.html.

14.4 Lettura dei dati di allineamento

Gli output degli esperimenti NGS sono letture di sequenze che devono essere allineate e mappate su un genoma di riferimento. Il primo passo nell'analisi dei dati NGS è quello di allineare le letture al genoma di riferimento. Questo compito di allineamento è computazionalmente impegnativo a causa degli enormi volumi dei dati NGS e dei genomi di riferimento. Tuttavia, ci sono strumenti — oltre a R — per fare questo. Gli strumenti di allineamento più comunemente usati includono BWA e Bowtie. È oltre lo scopo del testo entrare nei dettagli di questi due metodi.

Dopo aver mappato le letture del file FASTQ sul genoma di riferimento, si ottiene una mappa di allineamento della sequenza (SAM) o un file BAM (versione binaria del SAM). Questa sezione mostra come leggere/caricare questi file nell'area di lavoro R. Avremo bisogno del pacchetto *Rsamtools* per leggere i file

BAM. I dati di esempio utilizzati in questa sezione — un campione di sequenza (NA12878) del progetto 1000 Genome Project — sono disponibili presso l'UCSC a scopo dimostrativo.

Iniziamo con il download del file di esempio dal sito UCSC:

```
> download.file(url="http://genome.ucsc.edu/goldenPath/help/examples/bamExample.bam",
      destfile = "bamExample.bam")
```

Per leggere il file BAM utilizziamo la funzione `scanBam()` del pacchetto *Rsamtools* come segue:

```
> library(Rsamtools)
> bam <- scanBam("bamExample.bam")
```

Esaminiamo gli attributi del primo elemento della lista dei dati letti:

```
> names(bam[[1]])
 [1] "qname"  "flag"   "rname"  "strand" "pos"    "qwidth" "mapq"   "cigar"  "mrnm"   "mpos"   "isize"
[12] "seq"    "qual"
```

Controlliamo il conteggio dei record nei dati:

```
> countBam("bamExample.bam")
  space start end width           file records nucleotides
1    NA    NA  NA    NA bamExample.bam   36142     1474950
```

Se vogliamo leggere solo alcuni attributi li impostiamo come parametri digitando i seguenti comandi:

```
> what <- c("rname", "strand", "pos", "qwidth", "seq")
> param <- ScanBamParam(what=what)
> bam2 <- scanBam("bamExample.bam", param=param)
> names(bam2[[1]])
[1] "rname"  "strand" "pos"    "qwidth" "seq"
```

Leggiamo i dati come oggetto DataFrame:

```
> bam_df <- do.call("DataFrame", bam[[1]])
> head(bam_df)
DataFrame with 6 rows and 13 columns
                               qname      flag    rname   strand       pos    qwidth      mapq       cigar
                         <character> <integer> <factor> <factor> <integer> <integer> <integer> <character>
1                  SRR010939.15011799        35       21        +  33019936        76        99         76M
2                  SRR010939.15011799        19       21        -  33019947        76        99         76M
3                   SRR006419.2418801        16       21        -  33019958        51        76         51M
4 -XAT_0001_FC208BFAAXX:5:149:585:182        16       21        -  33019960        47        91         47M
5 -XAH_0003_FC203BTAAXX:1:191:243:169        16       21        -  33019963        47        90         47M
6                   ERR001302.6114800        19       21        -  33019965        36        99         36M
      mrnm      mpos     isize                      seq                     qual
  <factor> <integer> <integer>          <DNAStringSet>          <PhredQuality>
1       21  33019947        87 TAAAGATATA...GTATAGAGAA =789=7D69;...75-.-$+$*,
2       21  33019936       -87 TCAGTAACTC...CATATACTTT ,/06.-856$...?D:B1A=623
3       NA         0         0 ACAAATCCCA...TGTGTATATA A5"4;-&24G...@?>>AB>A=C
4       NA         0         0 TAATCCCAAC...TATGTGTATA 63*269:A:>...@@B<&??@=D
5       NA         0         0 AACAAACACT...GTGTATATAT #$6#&7:6<1...+F>;BFBD-I
6       21  33019766      -235 CCAACACTAG...CATATATATG 4:>@?>??CA...=@=?<>;==@
```

Estraiamo infine da questo oggetto DataFrame le sequenze che soddisfano determinate condizioni:

```
> table(bam_df$rname == '21' & bam_df$flag == 16)
FALSE  TRUE
31894  4248
```

La funzione `scanBam()` insieme all'altra funzione `countBam()` importa le mappe di allineamento binario di un'analisi NGS. L'argomento `what` definisce gli attributi che devono essere importati dai dati. Ulteriori informazioni sulla funzione sono disponibili nel file di aiuto (`?scanBam`). I file importati sono memorizzati come una lista, e ogni elemento è un allineamento. La lista può essere facilmente trasformata in un oggetto data.frame; ogni voce della lista è una riga e gli attributi dell'allineamento sono mostrati in termini di colonne. Infine, usiamo la funzione `table()` per determinare quante delle sequenze nel file soddisfano le condizioni desiderate.

Nell'elenco degli attributi del data.frame *bam_df* `rname` si riferisce al nome della sequenza di riferimento e `flag` indica il flag bitwise, che definisce le diverse proprietà dei segmenti nei dati; cioè, segmenti multipli, segmenti non mappati, il primo segmento, e così via. In questa sezione, per il flag abbiamo usato un valore di 16 che indica le letture per i filamenti inversi (*reverse strand*).

L'articolo *The Sequence Alignment/Map format and SAMtools* di Li *et al*. all'indirizzo http://bioinformatics.oxfordjournals.org/content/25/16/2078.full fornisce maggiori dettagli sui file BAM e SAM. La pagina del Broad Institute all'indirizzo http://software.broadinstitute.org/software/igv/bam fornisce informazioni utili sui file BAM e SAM. La pagina SAMtools all'indirizzo http://samtools.sourceforge.net fornisce alcuni utili strumenti per manipolare gli allineamenti nel formato SAM.

14.5 Pre-elaborazione dei dati NGS

I dati FASTQ hanno le sequenze di basi con corrispondenti punteggi di qualità (Phred) in termini di caratteri ASCII, come spiegato nella parte introduttiva del capitolo. Una volta letti nell'area di lavoro R, i dati sono pronti per essere analizzati. È però necessaria una certa pre-elaborazione per soddisfare le condizioni richieste sulla qualità e sul filtro dei dati: ad esempio, potremmo aver bisogno di punteggi Phred più alti e di un particolare *strand*. Questa sezione si occupa di questi aspetti, in particolare del filtro e dei controlli di qualità dei dati; useremo i dati scaricati dal database SRA e continueremo anche ad utilizzare la libreria *ShortRead*.

Prima di tutto, scarichiamo i file richiesti (continuiamo ad utilizzare la connessione *sraCon* già aperta):

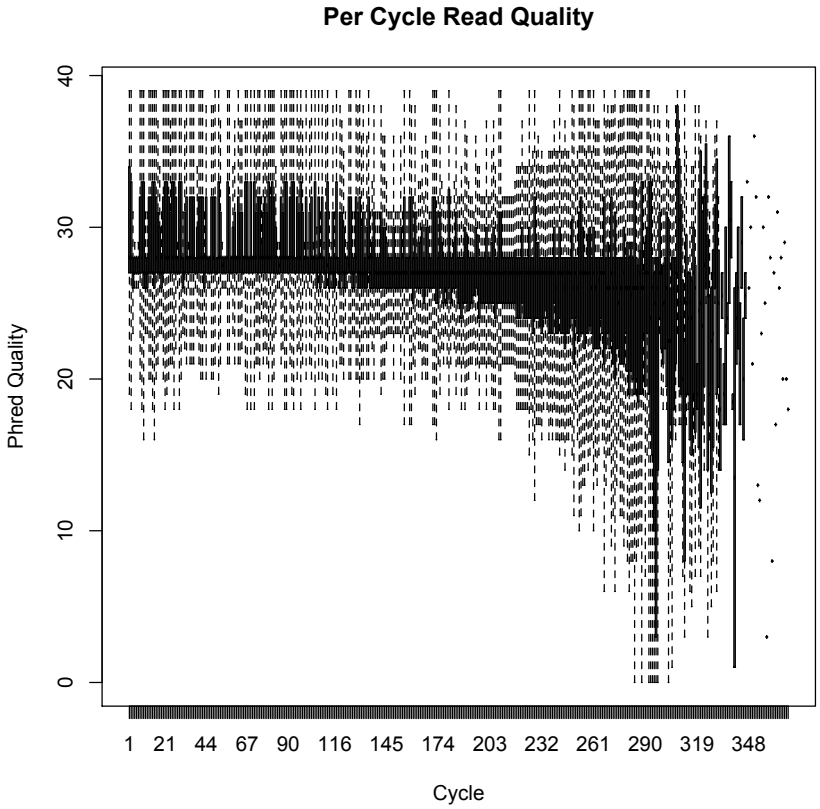
```
> library(SRAdb)
> getSRAfile(c("SRR000648", "SRR000657"), sraCon, fileType ="fastq")
        run submission     study    sample experiment
1 SRR000648  SRA000241 SRP000098 SRS000290  SRX000122
2 SRR000657  SRA000241 SRP000098 SRS000290  SRX000122
                                                                    ftp
1 ftp://ftp.sra.ebi.ac.uk/vol1/fastq/SRR000/SRR000648/SRR000648.fastq.gz
2 ftp://ftp.sra.ebi.ac.uk/vol1/fastq/SRR000/SRR000657/SRR000657.fastq.gz
```

Per valutare la qualità, utilizziamo la funzione `FastQuality()` della libreria *ShortRead*:

```
> library(ShortRead)
> myFiles <- list.files(getwd(), "fastq", full=TRUE)
> myFQ <- lapply(myFiles, readFastq)
> myQual <- FastqQuality(quality(quality(myFQ[[1]])))
> head(myQual)
class: FastqQuality
quality:
  A BStringSet instance of length 6
    width seq
[1]   262 C==<==FB1==<<HD8+<FB1==FB1<=B;C<C<<<EA/A9=;<<=FC...;<B<=8<8:99<;<=A<<8HD8,<<<A<:A;B<97==<<HD8,<9;A;
[2]   262 ==<C;==<FB0C<=<==<<=<C=<<<<==<C<<<C<===<=<==;<<F...<FC3=<;HC:2,%=A=(;GC6'=<HD8+;<:6<<;9<;=;:<;?:#;<
[3]   260 =<C<=;FB5&<;<=C<<==C<<=C<C<<=C;<=<<=C<<C<=<=<C==...@8B<@7<<<C<FC6&8GC3:=@879;C<6:8:<=:;=:98C=<<HD8+
[4]   287 A.;<<<;:<EA3#8<==EA3#;9;<B;8:=C<A<(==<<<;B;===<=...;<<:8<=;;;<<B;<8=C<GC2C=GC7(8=HD7)<)8:B<B<:====2
[5]   206 <<<<C<<=<C<<<=FC6';=<GC2<<=<=<<<<=EA/<FB0<====C=...B0<<=;:;<<<=C==:FB5&<<=C=;C=C=;;=;<=GC6(=;=B<!!!
[6]   312 =<C=C=GC3<B<;<<=C=C;C<=<C<<=C=;<;=;8C<<C=HC91*#:...FB02;C=GC3FB3;C=HD7)A;<9<;7A9<GC2C==:<B;=96=7:33
```

Convertiamo le misure di qualità in una matrice e visualizziamo i risultati come boxplot:

```
> readM <- as(myQual, "matrix")
> boxplot(readM, outline=FALSE, main="Per Cycle Read Quality", xlab="Cycle",
      ylab="Phred Quality")
```



Nel boxplot l'asse *x* mostra i cicli e l'asse *y* rappresenta le misure di qualità Phred disponibili all'interno del file FASTQ.

Un'altra interessante fase di pre-elaborazione prevede il filtro delle sequenze durante la lettura dei dati di allineamento. Per questo, prima di tutto scarichiamo nella directory di lavoro i dati di allineamento di esempio dall'indirizzo https://www.crescenziogallo.it/pub/myBowtie.txt e quindi carichiamoli in R:

```
> myData <- readAligned("myBowtie.txt", type = "Bowtie")
> myData
class: AlignedRead
length: 1000 reads; width: 35 cycles
chromosome: chr5 chr10 ... chr15 chr16
position: 151311502 35505989 ... 25552487 33204792
strand: - - ... + +
alignQuality: NumericQuality
alignData varLabels: similar mismatch
```

Ci sono diversi filtri che possono essere utilizzati, ad es. per i cromosomi, la lunghezza della sequenza, gli *strand* e così via. Per utilizzarli, definiamo prima il filtro richiesto (in questo caso, un filtro per lo *strand* +):

```
> strand <- strandFilter("+")
```

Quindi, applichiamo il filtro creato:

```
> myRead_strand <- readAligned("myBowtie.txt", filter=strand, type = "Bowtie")
> myRead_strand
class: AlignedRead
length: 521 reads; width: 35 cycles
chromosome: chr4 chr2 ... chr15 chr16
position: 94552391 98506780 ... 25552487 33204792
strand: + + ... + +
alignQuality: NumericQuality
alignData varLabels: similar mismatch
```

Possiamo combinare più filtri e poi utilizzarli con la funzione `compose()` come segue:

```
> chromosome <- chromosomeFilter("3")
> myFilt <- compose(strand, chromosome)
> myRead_filt <- readAligned("myBowtie.txt", filter=myFilt, type = "Bowtie")
> myRead_filt
class: AlignedRead
length: 55 reads; width: 35 cycles
chromosome: chr3 chr13 ... chr3 chr13
position: 149312235 27275680 ... 136314155 52437523
strand: + + ... + +
alignQuality: NumericQuality
alignData varLabels: similar mismatch
```

La funzione di controllo qualità descritta in questa sezione utilizza gli *score* Phred assegnati in termini di caratteri ASCII nei file FASTQ e calcola il punteggio di qualità effettivo per ogni ciclo. Questo si riflette in quanto abbiamo visto nel boxplot precedente. La funzione di filtro controlla semplicemente i dati ed estrae le letture di allineamento (*sequence read*) che soddisfano le condizioni. Gli altri filtri che possono essere utilizzati includono "idFilter", "positionFilter" e così via. Possiamo vedere la differenza tra l'oggetto *myData* che ha 1.000 letture con entrambi gli *strand* + e -, mentre l'oggetto filtrato, *myRead_filt*, ha solo 55 letture con il solo *strand* +.

14.6 Analisi dei dati RNAseq con il pacchetto *edgeR*

Per determinare se il conteggio per un trascritto è significativamente diverso o differentemente espresso nella condizione di trattamento, dobbiamo fare un'analisi differenziale del conteggio dei dati RNAseq, analogamente all'analisi differenziale dei geni vista nel capitolo 11. Eseguiremo questa analisi utilizzando i pacchetti *edgeR* e *goseq*, utilizzando i dataset inclusi negli stessi. I dati provengono da un esperimento che ha esaminato l'effetto della stimolazione degli androgeni su una linea di cellule tumorali della prostata umana, LNCaP. I dati hanno quattro controlli e tre campioni trattati.

Iniziamo con l'installazione e il caricamento delle librerie necessarie:

```
> BiocManager::install(c("edgeR","goseq"))
> library(edgeR)
> library(goseq)
```

Leggiamo i dati che ci interessano della libreria *goseq* e diamo un'occhiata alla parte iniziale:

Le prime quattro colonne sono i controlli e le ultime tre sono i campioni di trattamento. Assegniamo questi

```
> myData <- read.table(system.file("extdata", "Li_sum.txt", package="goseq"),
      sep = "\t", header = TRUE, stringsAsFactors = FALSE, row.names=1)
> head(myData)
                lane1 lane2 lane3 lane4 lane5 lane6 lane8
ENSG00000215688     0     0     0     0     0     0     0
ENSG00000215689     0     0     0     0     0     0     0
ENSG00000220823     0     0     0     0     0     0     0
ENSG00000242499     0     0     0     0     0     0     0
ENSG00000224938     0     0     0     0     0     0     0
ENSG00000239242     0     0     0     0     0     0     0
```

attributi ai dati come segue:

```
> myTreat <- factor(rep(c("Control","Treatment"),times = c(4,3)))
```

Creiamo ora un oggetto ti tipo "DGElist" utilizzando tutti i dati di conteggio e le informazioni di trattamento:

```
> myDG <- DGEList(myData,lib.size = colSums(myData),group = myTreat)
> myDG
An object of class "DGEList"
$counts
                lane1 lane2 lane3 lane4 lane5 lane6 lane8
ENSG00000215688     0     0     0     0     0     0     0
ENSG00000215689     0     0     0     0     0     0     0
ENSG00000220823     0     0     0     0     0     0     0
ENSG00000242499     0     0     0     0     0     0     0
ENSG00000224938     0     0     0     0     0     0     0
49501 more rows ...
$samples
          group lib.size norm.factors
lane1   Control  1178832            1
lane2   Control  1384945            1
lane3   Control  1716355            1
lane4   Control  1767927            1
lane5 Treatment  2127868            1
lane6 Treatment  2142158            1
lane8 Treatment   816171            1
```

L'oggetto *myDG* è una lista con due componenti: conteggi e campioni (informazioni sul trattamento), come mostrato nella schermata precedente.

Stimiamo la dispersione dei dati ed eseguiamo un test esatto di Fisher sulla stima ottenuta:

```
> myDisp <- estimateCommonDisp(myDG)
> myTest <- exactTest(myDisp)
```

Estraiamo ed esaminiamo i primi tag DE in base al *p*-value (o al *fold change* logaritmico assoluto) utilizzando la funzione `topTags()`:

```
> myRes <- topTags(mytest, sort.by = "PValue")
> head(myRes)
Comparison of groups:  Treatment-Control
                    logFC   logCPM       PValue          FDR
ENSG00000127954 11.557868 6.680748 2.574972e-80 1.274766e-75
ENSG00000151503  5.398963 8.499530 1.781732e-65 4.410322e-61
ENSG00000096060  4.897600 9.446705 7.983756e-60 1.317479e-55
ENSG00000091879  5.737627 6.282646 1.207655e-54 1.494654e-50
ENSG00000132437 -5.880436 7.951910 2.950042e-52 2.920896e-48
ENSG00000166451  4.564246 8.458467 7.126763e-52 5.880292e-48
```

Il pacchetto *edgeR* utilizza i dati di conteggio modellandoli attraverso un modello di Poisson sovradisperso, assumendo che si tratti di una distribuzione binomiale negativa. Segue quindi una procedura Empirical Bayes per moderare il grado di iperdispersione tra i geni tramite la massima probabilità condizionale, condizionata al conteggio totale per il gene (funzione `DGEList()`). Per calcolare l'espressione differenziale di un tag/gene, viene eseguita una fase di test esatto di Fisher, ottenendo i corrispondenti score statistici. Infine vengno calcolati i tag più significativi secondo i rispettivi *p*-value. La funzione `topTags()`, per default, restituisce i primi 10 risultati; possiamo ottenere il numero desiderato di tag impostando l'argomento *n* della funzione.

L'articolo *edgeR: a Bioconductor package for differential expression analysis of digital gene expression data* di Robinson *et al*. (https://academic.oup.com/bioinformatics/article/26/1/139/182458) fornisce ulteriori dettagli sul pacchetto *edgeR*.

Per una dettagliata illustrazione dei metodi statistici adottati nell'analisi dei dati consultare gli articoli *Moderated statistical tests for assessing differences in tag abundance* (http://bioinformatics.oxfordjournals.org/content/23/21/2881.abstract?ijkey=02b1e386cfb11240f48d2bafc520b091f53bfd95&keytype2=tf_ipsecsha) e *Small-sample estimation of negative binomial dispersion, with applications to SAGE data* (http://biostatistics.oxfordjournals.org/content/9/2/321.abstract?ijkey=7c4f41ecdfbc114236c79cea46db9d733c624d69&keytype2=tf_ipsecsha) di Robinson *et al*.

L'articolo *Determination of tag density required for digital transcriptome analysis: Application to an androgen-sensitive prostate cancer model* di Li *et al*. (http://www.pnas.org/content/early/2008/12/16/0807121105) fornisce informazioni sui dataset utilizzati in questa sezione.

14.7 Analisi differenziale dei dati NGS con il pacchetto *limma*

Abbiamo visto l'analisi differenziale dell'espressione genica nel capitolo 11 insieme al pacchetto *limma*, che può gestire esperimenti multipli tramite i metodi statistici Empirical Bayes e utilizza conteggi di lettura normalizzati per ogni gene. Questa sezione spiegherà l'uso del pacchetto *limma* di Bioconductor per l'analisi differenziale dei geni con i dati NGS.

Oltre alla libreria *limma* utilizzeremo il dataset *Pasilla* che può essere scaricato da Bioconductor e consiste in conteggi di sequenza da un esperimento di perturbazione nella *Drosophila*. Per saperne di più, fare riferimento all'articolo *Conservation of an RNA regulatory map between Drosophila and mammals* di Brooks *et al*. (http://genome.cshlp.org/content/early/2010/10/04/gr.108662.110).

Iniziamo scaricando e attivando i pacchetti contenenti i dati dal repository Bioconductor:

```
> BiocManager::install(c("DESeq","pasilla"))
> library(limma)
> library(DESeq)
> library(pasilla)
> data(pasillaGenes)
```

Controlliamo i dati:

```
> pasillaGenes
CountDataSet (storageMode: environment)
assayData: 14470 features, 7 samples
  element names: counts
protocolData: none
phenoData
  sampleNames: treated1fb treated2fb ... untreated4fb (7 total)
  varLabels: sizeFactor condition type
  varMetadata: labelDescription
featureData: none
experimentData: use 'experimentData(object)'
  pubMedIds: 20921232
Annotation:
```

Creiamo ora una matrice dei dati di espressione utilizzando i conteggi del dataset *pasillaGenes* come segue:

```
> eset <- counts(pasillaGenes)
```

Il dataset ha sette campioni; i primi tre sono i campioni di trattamento e gli ultimi quattro sono i controlli. Assegniamo le etichette appropriate (T_1…T_3 e C_1…C_4) ai dati della matrice *eset*:

```
> colnames(eset) <- c(paste("T", 1:3, sep="_"), paste("C", 1:4, sep="_"))
```

Diamo un'occhiata ai dati di espressione:

```
> head(eset)
            T_1  T_2  T_3  C_1  C_2  C_3  C_4
FBgn0000003   0    0    1    0    0    0    0
FBgn0000008  78   46   43   47   89   53   27
FBgn0000014   2    0    0    0    0    1    0
FBgn0000015   1    0    1    0    1    1    2
FBgn0000017 3187 1672 1859 2445 4615 2063 1711
FBgn0000018  369  150  176  288  383  135  174
```

Creiamo anche una matrice di progetto dell'esperimento per i sette campioni come segue:

```
> design <- cbind(Intercept=1,trt=c(1,1,1,0,0,0,0))
> design
     Intercept trt
[1,]         1   1
[2,]         1   1
[3,]         1   1
[4,]         1   0
[5,]         1   0
[6,]         1   0
[7,]         1   0
```

Eseguiamo una trasformazione con la funzione `voom()` (un acronimo per *mean-variance modelling at the observational level*) utilizzando la matrice di progettazione dell'esperimento:

```
> eset_voom <- voom(eset, design, plot=FALSE)
```

La funzione trasforma i dati di conteggio in *log2-counts per million* (logCPM), stima la relazione media-varianza e la utilizza per calcolare i pesi appropriati a livello di osservazione: i dati sono quindi pronti per la modellazione lineare. I dati di conteggio mostrano sempre marcate relazioni media-varianza; i conteggi grezzi mostrano una varianza crescente con l'aumento della dimensione del conteggio, mentre i log-conteggi mostrano tipicamente una tendenza alla diminuzione della media-varianza. Questa funzione stima l'andamento della variazione media per i log-conteggi, quindi assegna un peso ad ogni osservazione in base alla sua varianza prevista. I pesi sono poi utilizzati nel processo di modellazione lineare per regolare l'*eteroschedasticità* (un campione è eteroschedastico se al suo interno esistono sotto-popolazioni che abbiano varianze differenti; in tal caso vengono meno alcune delle ipotesi classiche del modello di regressione lineare).

Adattiamo un modello lineare sui dati *eset* e sulla matrice di progettazione:

```
> fit <- lmFit(eset_voom,design)
```

Per eseguire il calcolo delle statistiche, utilizziamo la funzione `eBayes()`:

```
> fitE <- eBayes(fit)
```

Per trovare i geni più significativi e filtrarli in base ai corrispondenti *p*-value digitiamo i seguenti comandi:

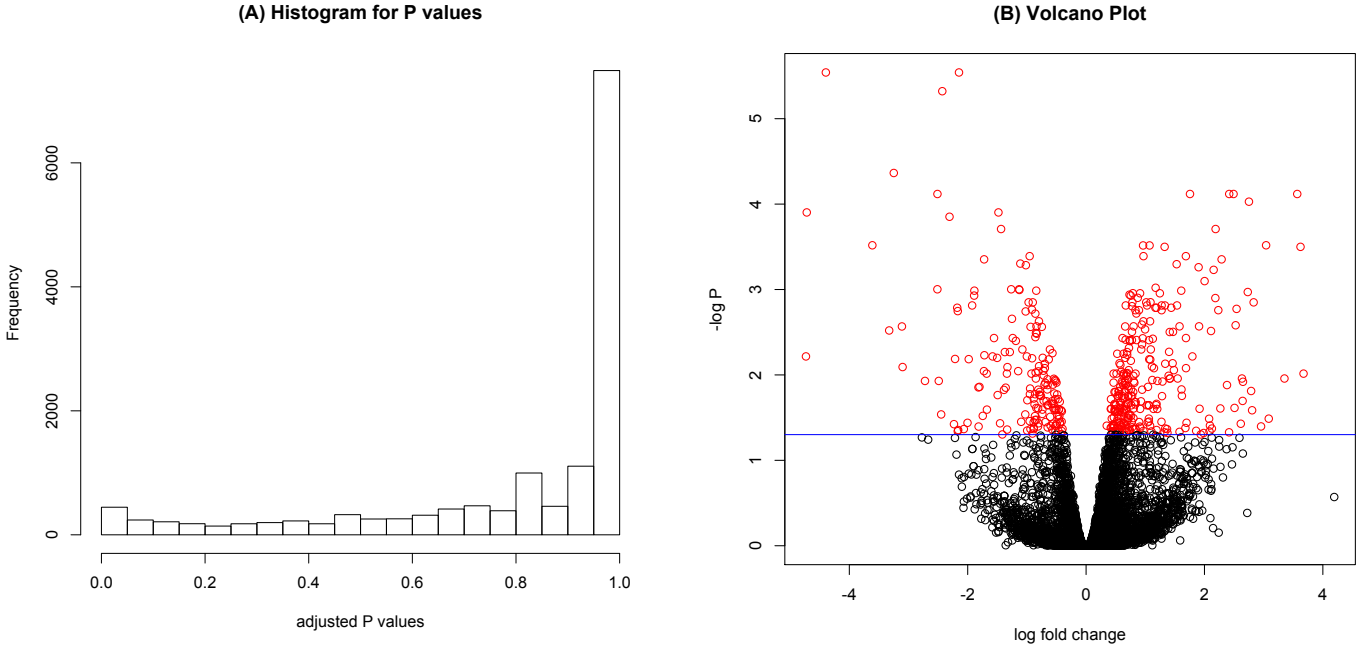
```
> topAll <- topTable(fitE, n=nrow(eset), coef = 2, adjust = "BH")
> DEgenes <- rownames(topAll[which(topAll$adj.P.Val<0.05),])
> head(DEgenes)
[1] "FBgn0029167" "FBgn0035085" "FBgn0039155" "FBgn0001226" "FBgn0011260"
```

L'analisi *limma* è già stata descritta nel capitolo 11. In questa sezione abbiamo usato i dati di conteggio al posto dei valori di espressione normalizzati. Un'altra differenza è la trasformazione *voom* che stima la relazione media-varianza per i log-conteggi, che genera in modo robusto un peso di precisione per ogni

singola osservazione normalizzata. Per conoscere i dettagli del pacchetto *limma*, fare riferimento al capitolo 11.

I grafici seguenti mostrano un istogramma dei tag con diversi intervalli di *p*-value e un *volcano plot* per l'analisi *limma*. Il codice per generare i grafici è il seguente:

```
> hist(topAll$adj.P.Val, xlab="adjusted P values", main="(A) Histogram for P values")
> clr <- rep("black",nrow(topAll)) # creates a vector for color
> clr[which(topAll$adj.P.Val<0.05)] <- "red" # sets color for DE to red
> plot(x = topAll$logFC, y = -log10(topAll$adj.P.Val), col = clr,
    xlab = "log fold change", ylab = "-log P", main = "(B) Volcano Plot") # do a volcano plot
> abline(h = -log10(0.05), col = "blue") # draw a horizontal line marking a P value threshold for 0.05
```



I risultati mostrano che abbiamo circa 500 tag che sono significativamente più espressi tra le due condizioni. Possiamo verificare quanti di questi mostrano un *fold change* più alto (ad es. 2):

```
> rownames(topAll[which(topAll$adj.P.Val<0.05&abs(topAll$logFC) > 2),])
 [1] "FBgn0029167" "FBgn0035085" "FBgn0039155" "FBgn0011260" "FBgn0034736" "FBgn0029896" "FBgn0000071"
 [8] "FBgn0051092" "FBgn0026562" "FBgn0003501" "FBgn0033764" "FBgn0035189" "FBgn0034434" "FBgn0037290"
[15] "FBgn0260011" "FBgn0024288" "FBgn0051642" "FBgn0034438" "FBgn0038832" "FBgn0032405" "FBgn0020248"
[22] "FBgn0052407" "FBgn0261284" "FBgn0040827" "FBgn0038198" "FBgn0030598" "FBgn0039827" "FBgn0050463"
[29] "FBgn0039593" "FBgn0037754" "FBgn0030763" "FBgn0085359" "FBgn0033065" "FBgn0030041" "FBgn0010387"
[36] "FBgn0038237" "FBgn0032436" "FBgn0053318" "FBgn0038012" "FBgn0063667" "FBgn0030964" "FBgn0033760"
[43] "FBgn0051555" "FBgn0046258" "FBgn0037143" "FBgn0259236" "FBgn0037223" "FBgn0037191" "FBgn0002578"
[50] "FBgn0051663" "FBgn0052700" "FBgn0039937" "FBgn0033733" "FBgn0050324" "FBgn0034898" "FBgn0028939"
[57] "FBgn0051776" "FBgn0032770" "FBgn0020639"
```

Possiamo vedere che 59 tag mostrano un $\log_2$ *fold change* maggiore di due (in valore assoluto), il che rende questi geni importanti per lo studio.

L'articolo *voom: precision weights unlock linear model analysis tools for RNA-seq read count* di Law *et al.* (http://genomebiology.com/2014/15/2/R29) fornisce maggiori dettagli sulla trasformazione *voom*.

14.8 Arricchimento dei dati RNAseq con i termini GO

I dati RNAseq provenienti da NGS forniscono grandi dettagli sul quadro trascrizionale cellulare. Oltre a misurare i livelli di espressione delle trascrizioni, forniscono informazioni sullo splicing alternativo, sulle espressioni allele-specifiche e così via. I dati RNAseq forniscono quindi un quadro più completo dell'espressione differenziale nelle cellule; tuttavia, l'aggiunta degli aspetti funzionali può raffinare questa analisi ulteriormente con la robustezza statistica. Questa sezione mostra l'evidenziazione dei dati RNAseq in base ai termini GO.

Per la nostra esposizione continueremo ad utilizzare la libreria *goseq* e i dati inclusi. Iniziamo con il caricamento dei pacchetti richiesti, *goseq* e *edgeR*, e diamo un'occhiata ai genomi supportati dai pacchetti:

```
> library(goseq)
> library(edgeR)
> supportedGenomes()
         db           species      date                                                name
2      hg19             Human Feb. 2009                Genome Reference Consortium GRCh37
3      hg18             Human Mar. 2006                                     NCBI Build 36.1
4      hg17             Human  May 2004                                       NCBI Build 35
5      hg16             Human Jul. 2003                                       NCBI Build 34
21  felCat3               Cat Mar. 2006                          Broad Institute Release 3
…
AvailableGeneIDs
2 ccdsGene,ensGene,exoniphy,geneSymbol,knownGene,nscanGene,refGene,xenoRefGene
3 acembly,acescan,ccdsGene,ensGene,exoniphy,geneSymbol,geneid,genscan,knownGene,knownGeneOld3,refGene,sgpGene,sibGene,xenoRefGene
4 acembly,acescan,ccdsGene,ensGene,exoniphy,geneSymbol,geneid,genscan,knownGene,refGene,sgpGene,vegaGene,vegaPseudoGene,xenoRefGene
5 acembly,ensGene,exoniphy,geneSymbol,geneid,genscan,knownGene,refGene,sgpGene
21 ensGene,geneSymbol,geneid,genscan,nscanGene,refGene,sgpGene,xenoRefGene
…
```

Carichiamo dal pacchetto *goseq* i dati da utilizzare per l'analisi:

```
> myData <- read.table(system.file("extdata", "Li_sum.txt", package="goseq"),
      sep = '\t', header = TRUE, stringsAsFactors = FALSE, row.names=1)
```

Le prime quattro colonne del data.frame *myData* sono i controlli e le ultime tre sono i campioni di trattamento. Etichettiamo le colonne con gli attributi corrispondenti:

```
> myTreat <- factor(rep(c("Control","Treatment"),times = c(4,3)))
```

Creiamo ora un oggetto DGElist tramite la libreria edgeR utilizzando i dati di conteggio e le informazioni di trattamento:

```
> myDG <- DGEList(myData, lib.size = colSums(myData), group = myTreat)
```

Stimiamo con l'oggetto DGElist la dispersione nei dati seguita da un test esatto di Fisher:

```
> myDisp <- estimateCommonDisp(myDG)
> myTest <- exactTest(myDisp)
```

Utilizziamo i geni di questa analisi per l'arricchimento GO. Estriamo perciò i geni con i *p*-value desiderati e la condizione di log *fold change* con i nomi dei geni corrispondenti, creando un vettore come segue:

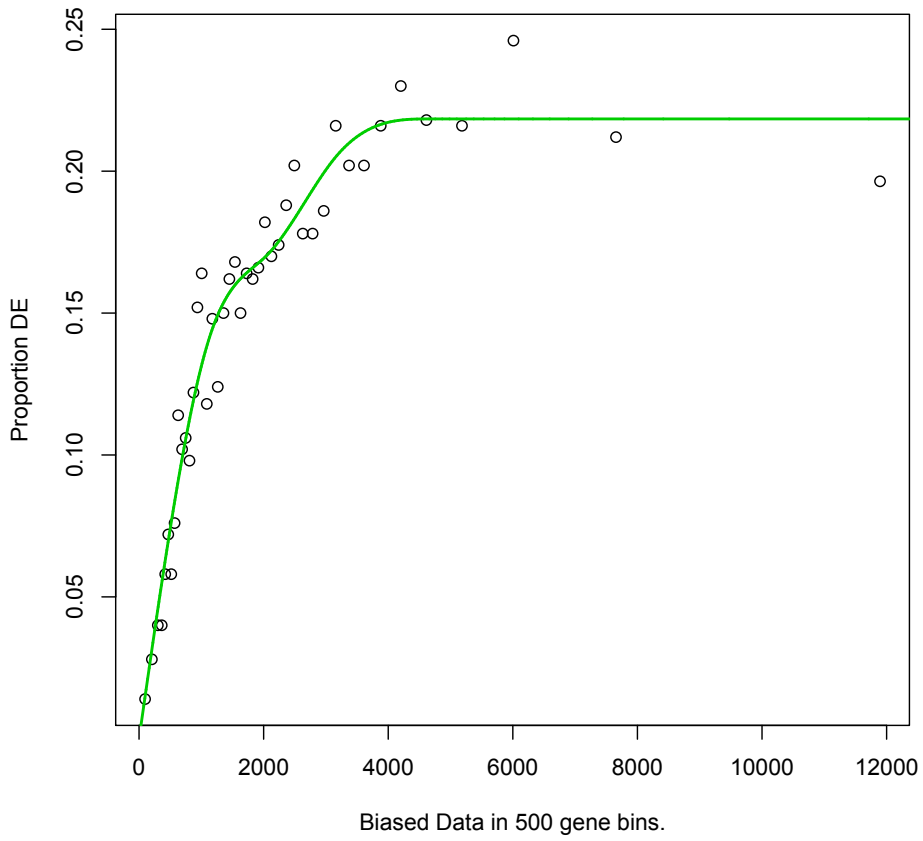
```
> myTags <- as.integer(p.adjust(myTest$table$PValue[myTest$table$logFC!=0],
      method="BH") < 0.05)
> names(myTags) <- row.names(myTest$table[myTest$table$logFC!=0,])
```

Il vettore *myTags* creato nella fase precedente riporta lo stato di ogni gene in termini di tag 1 (*differentially expressed*, 3.208 geni) o 0 (*non differentially expressed*, 19.535 geni):

```
> table(myTags)
myTags
    0     1
19535  3208
```

Calcoliamo ora la funzione di ponderazione delle probabilità per un insieme di geni in base al loro stato:

```
> wtFunc <- nullp(myTags,"hg19","ensGene")
Loading hg19 length data...
> head(wtFunc)
                DEgenes bias.data        pwf
ENSG00000230758       0       247 0.03757470
ENSG00000182463       0      3133 0.20436865
ENSG00000124208       0      1978 0.16881769
ENSG00000230753       0       466 0.06927243
ENSG00000224628       0      1510 0.15903532
ENSG00000125835       0       954 0.12711992
```



L'istruzione `nullp()` calcola una funzione di ponderazione della probabilità per un insieme di geni (nel nostro caso contenuti in *myTags* e identificati dall'ID "ensGene") di un genoma di riferimento (nel nostro caso "hg19") sulla base di un insieme di dati parziali (di solito la lunghezza dei geni) e lo stato di ogni gene come differenzialmente espresso o meno. Produce come risultato un data frame di 3 colonne denominate "DEgenes", "bias.data" e "pwf" con i nomi delle righe impostati sui nomi dei geni. Ogni riga corrisponde a un gene con la colonna "DEgenes" che specifica se il gene è DE (1 per DE, 0 per non DE), la colonna "bias.data" che fornisce il valore numerico del bias DE di cui si tiene conto (di solito la lunghezza del gene o il numero di conteggi) e la colonna "pwf" (Probability Weighting Function) che fornisce il valore dei geni sulla funzione di ponderazione delle probabilità. Questo oggetto viene di solito passato a *goseq* per calcolare l'arricchimento delle categorie GO o a *plotPWF* per ulteriori visualizzazioni grafiche.

Utilizziamo i risultati ottenuti per calcolare l'arricchimento come segue (occorre caricare preliminarmente il pacchetto *org.Hs.eg.db* di Bioconductor):

```
> BiocManager::install("org.Hs.eg.db")
> myEnrich_wall <- goseq(wtFunc,"hg19","ensGene", test.cats=c("GO:BP"))
Fetching GO annotations...
Loading required package: AnnotationDbi
For 9426 genes, we could not find any categories. These genes will be excluded.
To force their use, please run with use_genes_without_cat=TRUE (see documentation).
This was the default behavior for version 1.15.1 and earlier.
Calculating the p-values...
'select()' returned 1:1 mapping between keys and columns
```

Diamo infine un'occhiata alla parte iniziale dell'arricchimento GO ottenuto:

```
> head(myEnrich_wall)
       category over_represented_pvalue under_represented_pvalue numDEInCat numInCat
1674 GO:0006793            3.599915e-08                         1        620     2554
1550 GO:0006614            4.732072e-08                         1         34       95
57   GO:0000278            5.978181e-08                         1        243      881
1675 GO:0006796            6.736033e-08                         1        612     2528
1549 GO:0006613            7.959581e-08                         1         35       99
2127 GO:0008283            8.843440e-08                         1        345     1314
                                                      term ontology
1674                               phosphorus metabolic process       BP
1550 SRP-dependent cotranslational protein targeting to membrane       BP
57                                           mitotic cell cycle       BP
1675             phosphate-containing compound metabolic process       BP
1549               cotranslational protein targeting to membrane       BP
2127                                         cell proliferation       BP
```

Per esaminare il significato di una categoria GO (ad es. la prima) utilizziamo il pacchetto *GO.db*:

```
> library(GO.db)
> GOTERM[[myEnrich_wall$category[1]]]
GOID: GO:0006793
Term: phosphorus metabolic process
Ontology: BP
Definition: The chemical reactions and pathways involving the nonmetallic element
    phosphorus or compounds that contain phosphorus, usually in the form of a
    phosphate group (PO4).
Synonym: phosphorus metabolism
```

Gran parte della sezione attuale è simile alla precedente, che riguarda il pacchetto *edgeR*. Nei primi passi abbiamo calcolato i geni o tag di interesse e abbiamo attribuito gli ID Ensembl ai tag; la funzione *goseq* utilizzata recupera quindi la categoria GO per i geni Ensembl ottenuti. In questa sezione abbiamo usato solo la categoria ontologica BP (Biological Process), ma per le altre categorie possiamo usare ad es. i valori GO:CC o GO:MF. Oltre a recuperare l'annotazione GO, la funzione *goseq* calcola anche l'arricchimento in termini di sovra e sotto rappresentazione dei *p*-value. Infine, il pacchetto *GO.db* ha fornito i dettagli effettivi dei termini GO per le categorie recuperate.

Abbiamo qui visto solo uno dei metodi di arricchimento (Wallenius, quello di default usato dalla funzione `goseq()`), ma ci sono altri metodi possibili come Sampling e Hypergeometric (però sconsigliati per la loro lunghezza computazionale o inadeguatezza alle applicazioni biologiche). Per saperne di più, digitare `?goseq` nella console R.

L'articolo *Gene ontology analysis for RNA-seq: accounting for selection bias* di Young *et al*. all'indirizzo https://genomebiology.biomedcentral.com/articles/10.1186/gb-2010-11-2-r14?report=reader fornisce dettagli sui metodi *goseq*.

14.9 L'arricchimento KEGG dei dati di sequenza

La sezione precedente riguardava l'arricchimento dei tag con le categorie GO; tuttavia, possiamo fare un'analisi simile anche in termini di annotazioni KEGG. In questa sezione ci occupiamo dell'arricchimento KEGG dei dati di sequenza, continuando ad utilizzare gli stessi dati sella sezione precedente.

Iniziamo con il caricamento delle librerie e dei dati necessari:

```
> library(goseq)
> library(edgeR)
> library(org.Hs.eg.db)
> myData <- read.table(system.file("extdata", "Li_sum.txt", package = "goseq"),
      sep = "\t", header = TRUE, stringsAsFactors = FALSE, row.names = 1)
```

Le prime quattro colonne dei dati sono i controlli e le ultime tre sono i campioni di trattamento. Assegniamo questi attributi ai dati ed eseguiamo il calcolo dei tag differenziali:

```
> myTreat <- factor(rep(c("Control","Treatment"),times = c(4,3)))
> myDG <- DGEList(myData, lib.size = colSums(myData), group = myTreat)
> myDisp <- estimateCommonDisp(myDG)
> myTest <- exactTest(myDisp)
```

Per l'arricchimento utilizziamo i geni con il *p*-value e il log *fold change* desiderati, con i corrispondenti nomi dei geni, creando un vettore come segue:

```
> myTags <- as.integer(p.adjust(myTest$table$PValue[myTest$table$logFC!=0],
      method = "BH") < 0.05)
> names(myTags) <- row.names(myTest$table[myTest$table$logFC!=0,])
```

Impostiamo ora i dati KEGG per l'arricchimento, iniziando con la conversione degli ID ENSEMBL in Entrez:

```
> en2eg <- as.list(org.Hs.egENSEMBL2EG)
```

Otteniamo gli ID KEGG per gli ID Entrez compilati con il seguente comando:

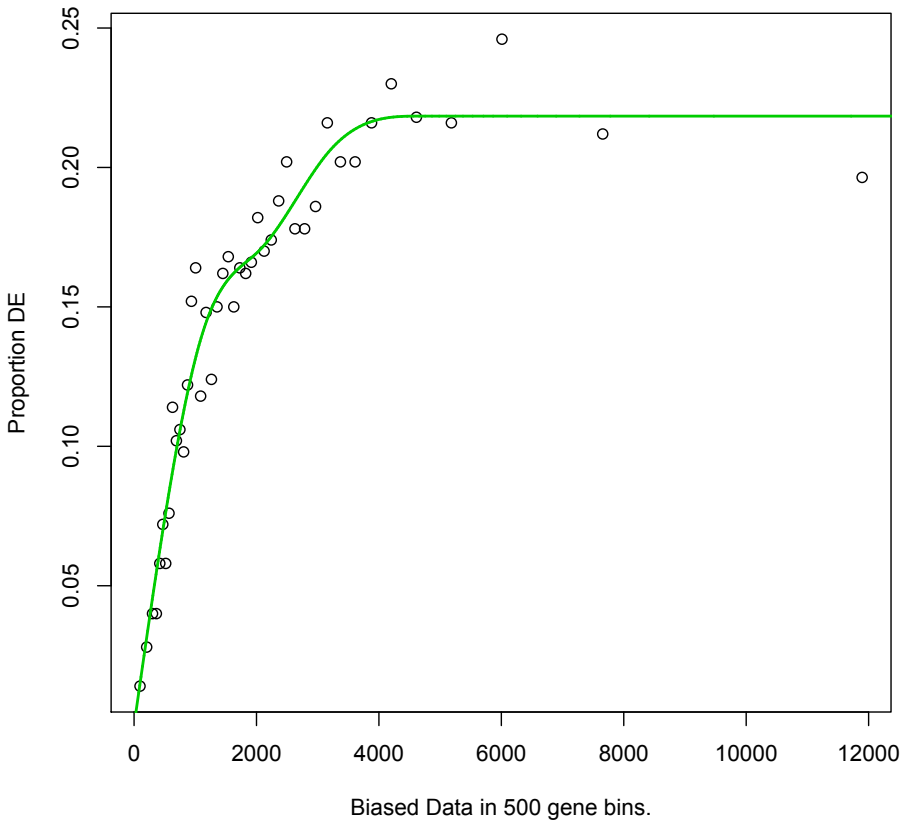
```
> eg2kegg <- as.list(org.Hs.egPATH)
```

Per ottenere la mappatura degli ID KEGG e Entrez, digitiamo i seguenti comandi:

```
> grepKEGG <- function(id,mapkeys) { unique(unlist(mapkeys[id], use.names = FALSE)) }
> kegg <- lapply(en2eg,grepKEGG,eg2kegg)
```

Calcoliamo la funzione di ponderazione delle probabilità:

```
> pwf <- nullp(myTags,"hg19","ensGene")
```



Utilizziamo infine la funzione *goseq* con le mappature KEGG per l'arricchimento dei tag:

```
> KEGG <- goseq(pwf,gene2cat = kegg)
Using manually entered categories.
For 18658 genes, we could not find any categories. These genes will be excluded.
To force their use, please run with use_genes_without_cat=TRUE (see documentation).
This was the default behavior for version 1.15.1 and earlier.
Calculating the p-values...
```

La schermata seguente mostra l'arricchimento KEGG dei dati:

```
> head(KEGG)
    category over_represented_pvalue under_represented_pvalue numDEInCat numInCat
88      03010             6.331426e-06                0.9999980         29        87
77      00900             2.393680e-04                0.9999710         10        15
113     04115             8.178449e-04                0.9996829         26        64
175     04964             2.152488e-03                0.9995921         10        17
27      00330             3.673147e-03                0.9986576         18        44
20      00250             5.204967e-03                0.9984341         13        28
```

In questa sezione abbiamo visto la ricerca delle annotazioni KEGG al posto delle categorie GO. Abbiamo inizialmente recuperato i tag differenzialmente espressi dai dati; quindi abbiamo estratto i geni di interesse in base al *p*-value e al log *fold change*. Il passo successivo è consistito nella mappatura Entrez dei geni ENSEMBL, seguita dalla corrispondente mappatura KEGG. Infine, abbiamo estratto le annotazioni KEGG dei geni significativi e calcolato gli score di arricchimento.

14.10 Analisi dei dati di metilazione

I dati di metilazione del DNA possono servire come strumento per rilevare un marcatore epigenetico per il quale è stato descritto un meccanismo dettagliato di ereditarietà mitotica. Di solito viene effettuata tramite il sequenziamento del DNA specifico per la metilazione o tramite la regolazione epigenetica dei dati da microarray. I recenti progressi nella tecnologia NGS e microarray rendono possibile la mappatura della metilazione del DNA a livello genomico ad alta risoluzione e in un gran numero di campioni. Uno degli obiettivi comuni nell'analisi dei dati di metilazione è quello di identificare le regioni differenzialmente metilate (DMR) quando si confrontano le condizioni di controllo e di trattamento.

In questa sezione verrà illustrato come affrontare questo problema: utilizzeremo allo scopo il pacchetto *methyAnalysis* e il dataset integrato.

Iniziamo con l'installazione del pacchetto da Bioconductor (se necessario) e il caricamento della libreria e del dataset nell'area di lavoro R:

```
> BiocManager::install(c("methyAnalysis", "TxDb.Hsapiens.UCSC.hg19.knownGene"))
> library(methyAnalysis)
> data(exampleMethyGenoSet)
```

Per ottenere alcune informazioni, come ad esempio i campioni e la posizione dei dati, esaminiamo le diverse componenti del dataset *exampleMethyGenoSet* (ci sono otto componenti):

```
> exampleMethyGenoSet
class: MethyGenoSet
dim: 4243 8
metadata(0):
assays(4): unmethylated methylated detection exprs
rownames(4243): cg17035109 cg06187584 ... cg07468397 cg08821909
rowData names(1): ID
colnames(8): Sample1 Sample2 ... Sample7 Sample8
colData names(1): SampleType
> slotNames(exampleMethyGenoSet)
[1] "history"         "annotation"      "rowRanges"       "colData"         "assays"
[6] "NAMES"           "elementMetadata" "metadata"
```

```
> colData(exampleMethyGenoSet)
DataFrame with 8 rows and 1 column
         SampleType
        <character>
Sample1       Type1
Sample2       Type1
Sample3       Type1
Sample4       Type1
Sample5       Type2
Sample6       Type2
Sample7       Type2
Sample8       Type2
> str(exampleMethyGenoSet)
Formal class 'MethyGenoSet' [package "methyAnalysis"] with 8 slots
  ..@ history        :'data.frame':     0 obs. of  4 variables:
  .. ..$ submitted  : 'AsIs' logi(0)
  .. ..$ finished   : 'AsIs' logi(0)
  .. ..$ command    : 'AsIs' logi(0)
  .. ..$ lumiVersion: 'AsIs' logi(0)
  ..@ annotation     : chr "IlluminaHumanMethylation450k.db"
  ..@ rowRanges      :Formal class 'GRanges' [package "GenomicRanges"] with 7 slots
…
 ..@ colData        :Formal class 'DataFrame' [package "S4Vectors"] with 6 slots
  .. .. ..@ rownames       : chr [1:8] "Sample1" "Sample2" "Sample3" "Sample4" ...
  .. .. ..@ nrows          : int 8
  .. .. ..@ listData       :List of 1
  .. .. .. ..$ SampleType: chr [1:8] "Type1" "Type1" "Type1" "Type1" ...
  .. .. ..@ elementType    : chr "ANY"
  .. .. ..@ elementMetadata: NULL
  .. .. ..@ metadata       : list()
  ..@ assays         :Reference class 'ShallowSimpleListAssays' [package
"SummarizedExperiment"] with 1 field
 …
..@ NAMES          : NULL
..@ elementMetadata:Formal class 'DataFrame' [package "S4Vectors"] with 6 slots
.. .. ..@ rownames       : NULL
.. .. ..@ nrows          : int 4243
.. .. ..@ listData       : Named list()
.. .. ..@ elementType    : chr "ANY"
.. .. ..@ elementMetadata: NULL
.. .. ..@ metadata       : list()
..@ metadata       : list()
```

I dati hanno due condizioni di campionamento, che sono Type1 e Type2. Livelliamo i dati di ingresso con una finestra di smoothing di ampiezza 200:

```
> methylSmooth <- smoothMethyData(exampleMethyGenoSet, winSize = 200)
Smoothing Chromosome chr21 ...
```

Estraiamo le condizioni dei campioni dal dataset:

```
> conditions <- exampleMethyGenoSet$SampleType
```

Rileviamo le DMR (Differentially Methylated Regions) nei dati in base ai dati di ingresso e al tipo di campione:

```
> myDMR <- detectDMR.slideWin(exampleMethyGenoSet, sampleType=conditions)
Smoothing Chromosome chr21 ...
```

Esaminiamo le prime sei componenti del risultato ottenuto:

```
> head(myDMR)
GRanges object with 6 ranges and 11 metadata columns:
             seqnames    ranges strand |    PROBEID         difference            p.value
                <Rle> <IRanges>  <Rle> |   <factor>          <numeric>          <numeric>
  cg17035109    chr21  10882029      * | cg17035109  -1.84116050891237 0.0627644943115837
  cg06187584    chr21  10883548      * | cg06187584 -0.456605878544024  0.416014860951571
  cg12459059    chr21  10884748      * | cg12459059 -0.359117942730594  0.365421521707119
  cg25450479    chr21  10884967      * | cg25450479 -0.359117942730594  0.365421521707119
  cg23347501    chr21  10884969      * | cg23347501 -0.359117942730594  0.365421521707119
  cg03661019    chr21  10885409      * | cg03661019 -0.353266150910433  0.380656000521138
                     p.adjust            tscore startWinIndex endWinIndex startLocation
                    <numeric>          <numeric>     <numeric>   <numeric>     <integer>
  cg17035109  0.18884880818101  -2.28040907618597             1           1      10882029
  cg06187584  0.61495957661053 -0.873425189908341             2           2      10883548
  cg12459059 0.562790422031127 -0.978927000428536             3           5      10884748
  cg25450479 0.562790422031127 -0.978927000428536             3           5      10884748
  cg23347501 0.562790422031127 -0.978927000428536             3           5      10884748
  cg03661019 0.578209920562373 -0.946031414573684             6           6      10885409
             endLocation         mean_Type1          mean_Type2
               <integer>          <numeric>           <numeric>
  cg17035109    10882029  -2.41837749804746  -0.577216989135089
  cg06187584    10883548  -2.22975671527783    -1.7731508367338
  cg12459059    10884969  0.259415093217181   0.618533035947775
  cg25450479    10884969  0.259415093217181   0.618533035947775
  cg23347501    10884969  0.259415093217181   0.618533035947775
  cg03661019    10885409 -0.417036279735772 -0.0637701288253387
  -------
  seqinfo: 1 sequence from hg19 genome; no seqlengths
```

Per identificare i DMR significativi, applichiamo la funzione `identifySigDMR()` all'output precedente:

```
> mySigDMR <- identifySigDMR(myDMR)
```

Prendiamo le informazioni di annotazione per le regioni dai dati UCSC (https://genome.ucsc.edu/cgi-bin/hgSession):

```
> dmr_anno <- annotateDMRInfo(mySigDMR, "TxDb.Hsapiens.UCSC.hg19.knownGene")
Loading required package: TxDb.Hsapiens.UCSC.hg19.knownGene
Loading required package: GenomicFeatures
```

Infine, esportiamo i risultati dell'analisi (verranno creati i file DMRdata_testExample_<*data_odierna*>.csv, DMRInfo_testExample_<*data_odierna*>.csv e DMRInfo_testExample_<*data_odierna*>.bed):

```
> export.DMRInfo(dmr_anno, savePrefix="testExample")
```

I dati di esempio che abbiamo usato in questa sezione sono un'estensione della classe eSet (utilizzata per contenere dati relativi a test ad alta produttività e metadati sperimentali), con slot per diversi tipi di informazioni come campioni, cromosomi e range genomici. Abbiamo iniziato la nostra analisi con il livellamento (*smoothing*) dei dati; la funzione `smoothMethyData()` considera la correlazione nei siti CpG vicini in termini della finestra definita (nel nostro caso di ampiezza 200), riducendo così il rumore. Con i dati meno rumorosi, `detectDMR.slideWin()` controlla se la regione (livellata) è differenzialmente

metilata in base al *t*-test (parametrico) o al test Wilcoxon (non parametrico); la regione viene poi fusa per ottenere le regioni continue mediante la funzione `getContinuousRegion()`. I risultati annotati con i dati del genoma sono stati infine esportati come file CSV, di cui vediamo uno stralcio:

CHROMOSOME	POSITION	PROBEID	difference	p.value	p.adjust	tscore	startWinIndex	endWinIndex	mean_Type1	mean_Type2	Transcript	EntrezID	GeneSymbol	distance2TSS	nearestTx	PROMOTER
chr21	19191096	cg12430776	-1.07152346805949	0.000123592435561639	0.00706138813614358	-8.7470211196085	279	281	-4.09227128083324	-3.02074781277375	uc002yko.4	54149	C21orf91	607	uc002ykq.4	FALSE
chr21	34522588	ch.21.33444458F	-1.28480473970052	7.6775226703329e-06	0.00309388312095657	-14.1827491036894	998	998	-4.51155528155554	-3.22675054185502	uc002yra.4	728409	LINC01548	19953	uc002yra.4	FALSE
chr21	37851847	cg02417033	-1.49225161667501	9.41612079823841e-05	0.00676843530598798	-9.17962630644639	1486	1486	0.620124385677425	2.11237600235243	uc002yvl.1	23562	CLDN14	541	uc002yvm.1	FALSE
chr21	38066047	cg10445315	-2.54808102022505	1.13173411956149e-05	0.00369206492389252	-13.2706343739005	1514	1514	0.577260922659085	3.12534194288413	uc002yvp.3	6493	SIM2	-5944	uc002yvp.3	FALSE
chr21	38075599	cg22711869	-1.55216154010044	0.000233210962154705	0.0099004441919398 24	-7.80531772537336	1555	1555	0.967205777453885	2.51936731755432	uc002yvp.3	6493	SIM2	3608	uc002yvr.2	FALSE
chr21	38076709	cg22289831	-3.51662897665358	2.60932603388828e-07	0.00055330758548601	-25.1381604575894	1556	1557	0.239312950325912	3.75594192697949	uc002yvp.3	6493	SIM2	4718	uc002yvr.2	FALSE
chr21	38076869	cg21697851	-3.46488568856872	4.10491597563916e-05	0.00435792551679 58	-10.6205297147535	1556	1558	0.57098982192709	4.03587551049581	uc002yvp.3	6493	SIM2	4878	uc002yvr.2	FALSE
chr21	38080975	cg15750546	-2.73678458570224	8.03105705019188e-07	0.00056766188249773	-20.8036095530873	1574	1576	0.0969041594080431	2.83368874511028	uc002yvp.3	6493	SIM2	8984	uc002yvr.2	FALSE
chr21	38081100	cg20349024	-2.73678458570224	8.03105705019188e-07	0.00056766188249773	-20.8036095530873	1574	1576	0.0969041594080431	2.83368874511028	uc002yvp.3	6493	SIM2	9109	uc002yvr.2	FALSE
chr21	38081193	cg01090834	-2.73678458570224	8.03105705019188e-07	0.00056766188249773	-20.8036095530873	1574	1576	0.0969041594080431	2.83368874511028	uc002yvp.3	6493	SIM2	9202	uc002yvr.2	FALSE
chr21	39285679	cg01360586	-3.87518312872458	2.53151754388646e-05	0.0043579255167958	-11.5511443779546	1748	1748	-1.45518525764503	2.41999787107955	uc002ywo.3	3763	KCNJ6	3062	uc011aej.2	FALSE
chr21	39748803	cg24018174	-2.80130749997503	0.000142434454218468	0.00784499377065615	-8.52849659641358	1803	1803	-0.175928079607726	2.6253794203673	uc021wjd.1	2078	ERG	284901	uc010gny.1	FALSE
chr21	40033892	cg17274064	-3.66493706892538	1.50974410295012e-08	6.40282474061146e-05	-40.5235124560946	1823	1823	-2.85508375497808	0.809853313947298	uc021wjd.1	2078	ERG	-188	uc021wjd.1	TRUE
chr21	42217001	cg02475236	-2.03975700494258	7.22780967530002e-05	0.00567650756165692	-9.61834409769408	2026	2026	0.467076402359827	2.50683340730241	uc002yyq.1	1826	DSCAM	2038	uc031rvr.1	FALSE
chr21	43652704	cg01881899	-2.00828027888908	0.000219018441191438	0.00990044419139824	-7.89439481899973	2384	2384	0.309978231149814	2.31825851003889	uc002zar.3	9619	ABCG1	12696	uc011aev.2	FALSE
chr21	45139229	cg00784703	-1.0405871865453	0.000207371936813841	0.00990044419139824	-7.9726152102145	2848	2849	-4.9126547367552	-3.8720675502099	uc002zdm.4	8566	PDXK	251	uc002zdn.4	FALSE
chr21	45139379	cg14522549	-1.0405871865453	0.000207371936813841	0.00990044419139824	-7.9726152102145	2848	2849	-4.9126547367552	-3.8720675502099	uc002zdm.4	8566	PDXK	401	uc002zdn.4	FALSE
chr21	47876058	cg09387528	-2.77916428787659	4.40541732684654e-06	0.00233542186039452	-15.5926292201797	4167	4167	0.114951040929133	2.89411532880572	uc002zji.3	23181	DIP2A	-2804	uc002zji.3	FALSE
chr21	47878552	cg19247551	-1.12351819086554	5.007089857557e-07	0.00056766188249773	-22.5278412185324	4172	4174	-5.48201828197236	-4.35850009110682	uc002zji.3	23181	DIP2A	-310	uc002zji.3	TRUE
chr21	47878727	cg15775835	-1.02145678363949	8.75420831206763e-06	0.00309388312095657	-13.8678237143095	4172	4176	-4.96909268492811	-3.94763590128863	uc002zji.3	23181	DIP2A	-135	uc002zji.3	TRUE
chr21	47878746	cg12533308	-1.02145678363949	8.75420831206763e-06	0.00309388312095657	-13.8678237143095	4172	4176	-4.96909268492811	-3.94763590128863	uc002zji.3	23181	DIP2A	-116	uc002zji.3	TRUE

Una regione differenzialmente metilata (DMR) è una regione in cui la maggior parte dei siti CpG sono metilati. In questa sezione abbiamo eseguito il rilevamento dello stato di metilazione delle sonde nell'oggetto *exampleMethyGenoSet*. Infine, abbiamo mappato queste regioni sulle mappe cromosomiche. Nella schermata precedente vediamo nella colonna "GeneSymbol" i valori SIM2, ERG, DIP2, e così via, regioni sostanzialmente metilate sul cromosoma 21 dove il *p*-value era significativo (cioè inferiore a 0,05). I dati di input utilizzati erano di tipo "eSet"; tuttavia possono essere in altri formati, come file CSV o file di testo (abbiamo già visto nel capitolo 11 come creare oggetti eSet da questi file).

L'articolo *Analysing and interpreting DNA methylation data* di Bock (http://www.nature.com/nrg/journal/v13/n10/full/nrg3273.html) fornisce ulteriori dettagli sull'uso dei dati di metilazione.

L'articolo *Comparison of Beta-value and M-value methods for quantifying methylation levels by microarray analysis* di Du *et al*. (http://www.biomedcentral.com/1471-2105/11/587) presenta i dettagli teorici del pacchetto *methyAnalysis*.

14.11 Analisi dei dati ChipSeq

ChipSeq (*Chromatin immuneprecipitation sequencing*, sequenziamento mediante immunoprecipitazione della cromatina) è un potente metodo per identificare i siti di legame del DNA a livello genomico per una proteina di interesse. È spesso usato per determinare i siti di legame per i fattori di trascrizione, gli enzimi di legame del DNA, gli istoni, gli accompagnatori o i nucleosomi. Il flusso di lavoro per produrre i dati ChipSeq parte dalle proteine legate ai legami incrociati e dalla cromatina. La cromatina viene frammentata, ed i frammenti di DNA legati ad una proteina vengono catturati utilizzando un anticorpo specifico per essa. Le estremità dei frammenti catturati sono sequenziati utilizzando tecniche di NGS. La mappatura computazionale del DNA sequenziato porta all'identificazione delle posizioni genomiche di questi frammenti, chiarendo il loro ruolo nelle interazioni delle proteine del DNA e nella ricerca epigenetica.

I dati ChipSeq consistono in letture brevi (*short read*) in formato file FASTQ. Vi è una lettura breve dopo ogni quinta riga del file.

Questa sezione si occupa dell'analisi dei dati ChipSeq in R e richiede il pacchetto *ChipSeq* con i dati inclusi. Utilizzeremo anche i dati del genoma del topo per la mappatura biologica dei risultati.

Iniziamo con il caricamento delle librerie R *chipseq* e *TxDb.Mmusculus.UCSC.mm9.knownGene*:

```
> BiocManager::install(c("chipseq", "TxDb.Mmusculus.UCSC.mm9.knownGene"))
> library(TxDb.Mmusculus.UCSC.mm9.knownGene)
> library(chipseq)
```

Per l'analisi dei dati a scopo dimostrativo utilizziamo il dataset integrato nel pacchetto denominato *cstest*:

```
> data(cstest)
> cstest
GRangesList object of length 2:
$ctcf
GRanges object with 450096 ranges and 0 metadata columns:
           seqnames              ranges strand
              <Rle>           <IRanges>  <Rle>
       [1]    chr10     3012936-3012959      +
       [2]    chr10     3012941-3012964      +
       [3]    chr10     3012944-3012967      +
       [4]    chr10     3012955-3012978      +
       [5]    chr10     3012963-3012986      +
       ...      ...                 ...    ...
  [450092]    chr12 121239376-121239399      -
  [450093]    chr12 121245849-121245872      -
  [450094]    chr12 121245895-121245918      -
  [450095]    chr12 121246344-121246367      -
  [450096]    chr12 121253499-121253522      -
  -------
  seqinfo: 35 sequences from an unspecified genome

$gfp
GRanges object with 295385 ranges and 0 metadata columns:
           seqnames              ranges strand
              <Rle>           <IRanges>  <Rle>
       [1]    chr10     3002512-3002535      +
       [2]    chr10     3009093-3009116      +
       [3]    chr10     3020716-3020739      +
       [4]    chr10     3023026-3023049      +
       [5]    chr10     3024629-3024652      +
       ...      ...                 ...    ...
  [295381]    chr12 121213126-121213149      -
  [295382]    chr12 121216905-121216928      -
  [295383]    chr12 121216967-121216990      -
  [295384]    chr12 121251805-121251828      -
  [295385]    chr12 121253426-121253449      -
  -------
  seqinfo: 35 sequences from an unspecified genome
```

Stimiamo la lunghezza dei frammenti nei dati:

```
> estimate.mean.fraglen(cstest$ctcf)
   chr10    chr11    chr12
179.6794 172.4884 181.6732
```

Estendiamo i frammenti in modo da coprire i siti di *binding* nelle sequenze. Utilizziamo una lunghezza di estensione basata sulle lunghezze dei frammenti dell'ultimo passaggio (lunghezza = 200 > `estimate.mean.fraglen(cstest$ctcf)`):

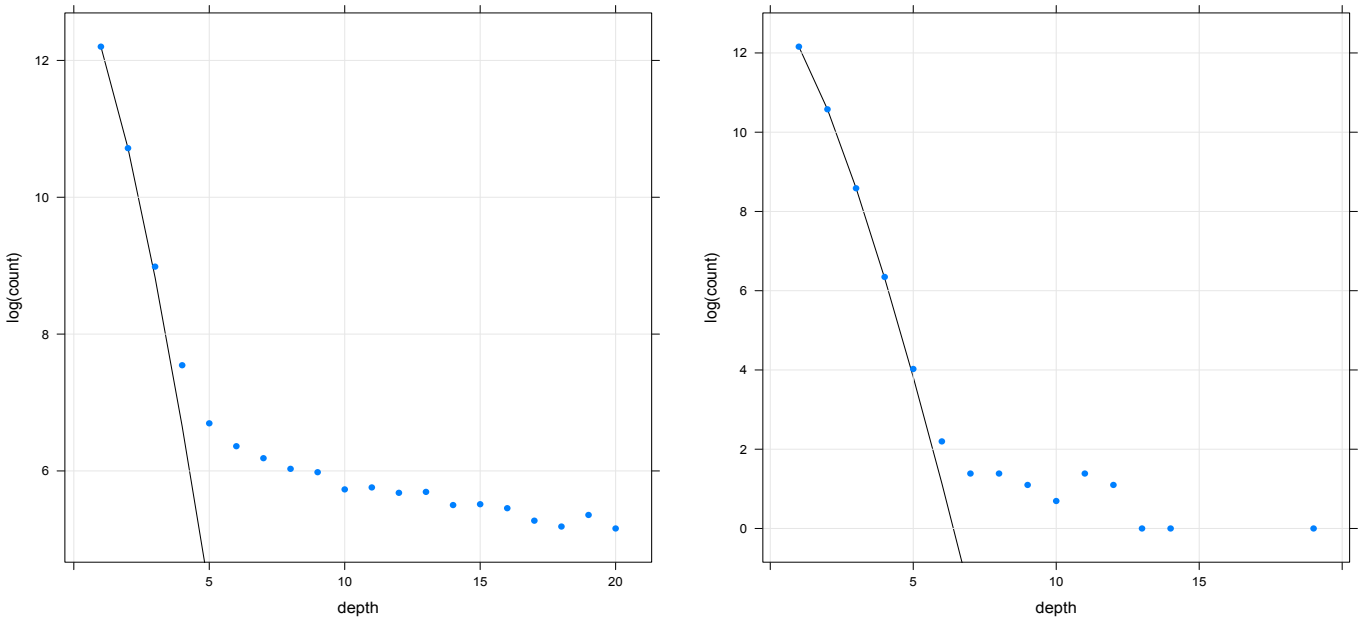
```
> ctcf_ext <- resize(cstest$ctcf, width = 200)
> gfp_ext <- resize(cstest$gfp, width = 200)
```

Un'utile sintesi di queste informazioni è la copertura (cioè quante volte ogni base del genoma è stata coperta da uno di questi intervalli) e può essere calcolata come segue:

```
> cov_ctcf <- coverage(ctcf_ext)
> cov_gfp <- coverage(gfp_ext)
```

Creiamo ora grafici "IslandDepth" per le regioni di interesse (il grafico traccia la distribuzione delle profondità delle isole utilizzando punti per le isole osservate e una linea per la stima del rumore di Poisson):

```
> library(lattice)
> islandDepthPlot(cov_ctcf)
> islandDepthPlot(cov_gfp)
```



Nei grafici l'asse *x* mostra la profondità, mentre l'asse *y* mostra i log-conteggi corrispondenti (nel complesso, il grafico mostra la copertura per i dati dei campioni).

Calcoliamo ora il cut-off dei picchi per un FDR (False Discovery Rate, tasso di errore di tipo I) desiderato (0,01):

```
> peakCutoff(cov_ctcf, fdr = 0.01)
[1] 5.091909
> peakCutoff(cov_gfp, fdr = 0.01)
[1] 6.979273
```

In base a questo valore, stabiliamo un valore di cut-off (7) da utilizzare per ottenere i picchi con alta copertura nei segmenti dei dati per entrambe le linee ctcf e gfp (si noti che il cut-off scelto si basa sul calcolo effettuato in precedenza):

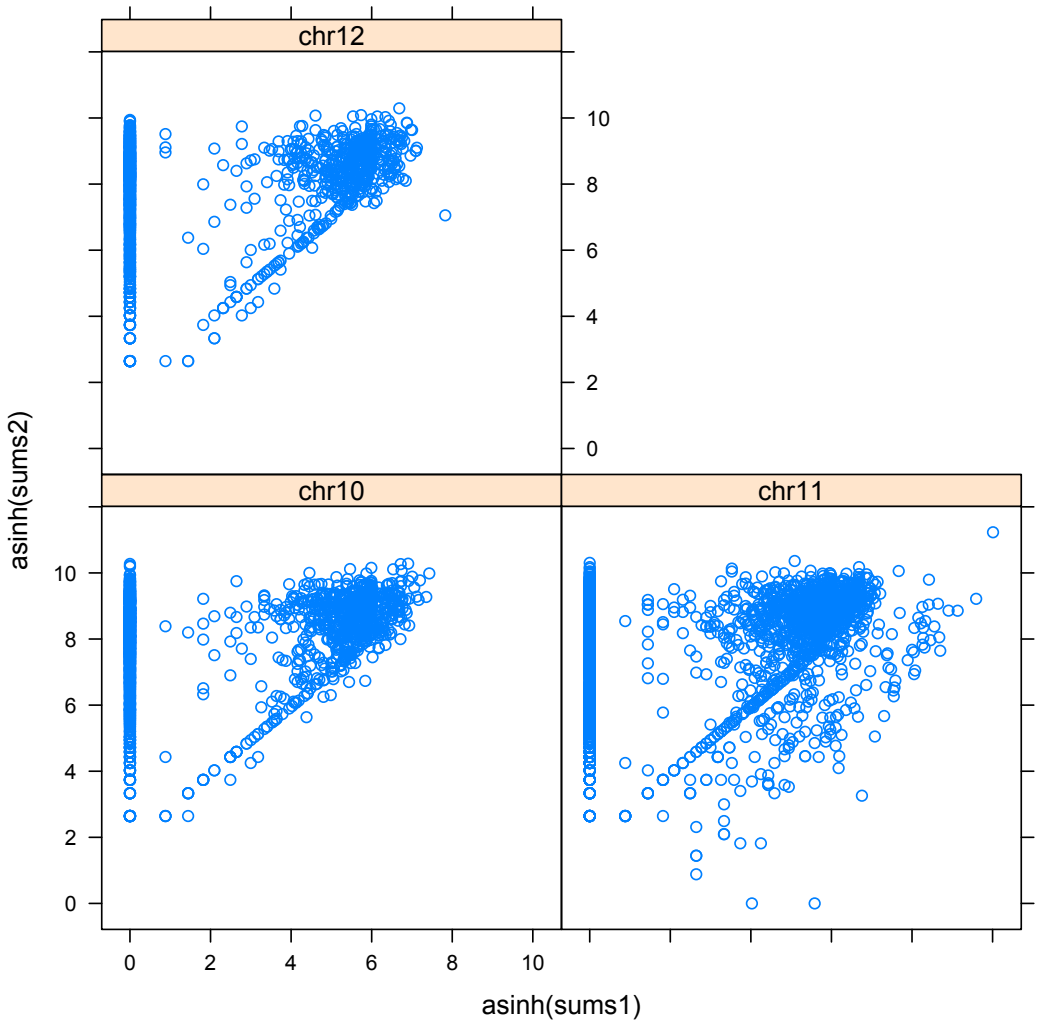
```
> peaks_ctcf <- slice(cov_ctcf, lower = 7)
> peaks_gfp <- slice(cov_gfp, lower = 7)
```

Calcoliamo ora i picchi differenziali con la seguente funzione per determinare quali picchi sono diversi nei due campioni:

```
> peakSummary <- diffPeakSummary(peaks_gfp, peaks_ctcf)
> head(data.frame(peakSummary))
  seqnames    start      end width strand comb.max sums1 sums2 maxs1 maxs2
1    chr10 3012944 3013140   197      *       11     0  1911     0    11
2    chr10 3135027 3135029     3      *        7     0    21     0     7
3    chr10 3234798 3234896    99      *       10     0   910     0    10
4    chr10 3234924 3234933    10      *        7     0    70     0     7
5    chr10 3270010 3270301   292      *       20   164  4072     1    19
6    chr10 3277660 3277861   202      *       13     0  1897     0    13
```

Visualizziamo i risultati in un grafico XY che mostra il riepilogo dei picchi per tre cromosomi::

```
> xyplot(asinh(sums2) ~ asinh(sums1) | seqnames, data = as.data.frame(peakSummary))
```



Verifichiamo se i picchi si trovano nella regione di interesse (regione del promotore) utilizzando i seguenti comandi:

```
> gregions <- transcripts(TxDb.Mmusculus.UCSC.mm9.knownGene)
> promoters <- flank(gregions, 1000, both = TRUE)
> peakSummary$inPromoter <- peakSummary %over% promoters
```

Diamo un'occhiata ai picchi nella regione del promotore esaminando l'oggetto creato nella fase precedente:

```
> which(peakSummary$inPromoter)
  [1]    2   14   41   51  100  101  110  124  126  130  133  141  155  156  157  158
 [17]  175  176  177  180  184  185  211  212  213  217  218  257  273  276  299  306
…
[657] 5962 5979 5996 6022 6047 6068 6130 6131 6148 6161 6182 6224 6259 6267 6268 6309
[673] 6313 6318 6322 6327 6328 6349 6361 6362
```

In questa sezione i dati di input provengono da tre cromosomi (10, 11 e 12) e in due linee che rappresentano la proteina `ctcf` (CCCTC binding factor) e `gfp` (Green Fluorescent Protein) nel topo. Lo scopo della nostra analisi è trovare quali picchi sono diversi in due linee (campioni). Il metodo spiegato in questa sezione trova i vettori di copertura dei dati e calcola i punteggi statistici per i picchi in essi presenti. Abbiamo iniziato con la ricerca della copertura per entrambe le linee estendendo le letture per coprire i siti di *binding*. Successivamente, abbiamo definito i picchi sulla base di un valore di cut-off secondo il FDR desiderato (0,01). La funzione `diffPeakSummary()` ha quindi combinato e riassunto i picchi per le due linee. Infine, abbiamo mappato i picchi sul genoma di riferimento e selezionato i picchi che si trovano nella regione del promotore, che sono di interesse per trovare i siti di legame.

L'articolo *ChIP-seq: welcome to the new frontier* di Elaine R. Mardis (http://www.nature.com/nmeth/journal/v4/n8/abs/nmeth0807-613.html) fornisce una descrizione dettagliata del flusso ChipSeq.

L'articolo *Practical Guidelines for the Comprehensive Analysis of ChIP-seq Data* di Bailey *et al*. (https://journals.plos.org/ploscompbiol/article?id=10.1371/journal.pcbi.1003326) illustra i dettagli del protocollo di analisi dei dati ChipSeq.

L'articolo *ChIP-seq Analysis in R (CSAR): An R package for the statistical detection of protein-bound genomic regions* di Muino *et al*. (http://www.newton.ac.uk/programmes/CGR/Muino,%20J.pdf) documenta il pacchetto *chipseq*.

L'articolo *BayesPeak — an R package for analysing ChIP-seq data* di Cairns *et al*. (http://bioinformatics.oxfordjournals.org/content/27/5/713.long) fornisce informazioni sul metodo BayesPeak.

14.12 Visualizzazione dei dati NGS

In questa sezione introdurremo alcune nuove visualizzazioni che possono essere utilizzate per rappresentare alcuni risultati dell'analisi NGS. Presenteremo due grafici: (*a*) un grafico di valutazione della qualità NGS; (*b*) una mappa per i dati di metilazione.

Per prima cosa, scarichiamo nella directory di lavoro alcuni file in formato FASTQ che ci serviranno per il grafico di valutazione della qualità:

```
> download.file(url = "https://www.crescenziogallo.it/pub/SRR038845.fastq",
      destfile="SRR038845.fastq")
> download.file(url = "https://www.crescenziogallo.it/pub/SRR038846.fastq",
      destfile="SRR038846.fastq")
> download.file(url = "https://www.crescenziogallo.it/pub/SRR038848.fastq",
      destfile="SRR038848.fastq")
> download.file(url = "https://www.crescenziogallo.it/pub/SRR038850.fastq",
      destfile="SRR038850.fastq")
```

Attiviamo la libreria *ggplot2* e carichiamo le funzioni `seeFastq()` e `seeFastPlot()` (vedi Appendice) per disegnare i grafici:

```
> library(ggplot2)
> source("seeFastq.R")
> source("seeFastqPlot.R")
```

Creiamo ora la lista dei file FASTQ scaricati in precedenza e assegniamo un nome descrittivo ad ogni file:

```
> fastq <- list.files(getwd(),"*.fastq$")
> names(fastq) <- paste("flowcell_6_lane", 1:4, sep="_")
> fastq
flowcell_6_lane_1 flowcell_6_lane_2 flowcell_6_lane_3 flowcell_6_lane_4
"SRR038845.fastq" "SRR038846.fastq" "SRR038848.fastq" “SRR038850.fastq"
```

Calcoliamo quindi i punteggi relativi alla qualità per ogni file mediante la funzione `seeFastq()`:

```
> fqlist <- seeFastq(fastq=fastq, batchsize=1000, klength=5)
```

Per visualizzare i risultati, tracciamo il grafico delle misure di qualità solo per il primo campione mediante la funzione `seeFastqPlot()` (vedi inizio pagina successiva):

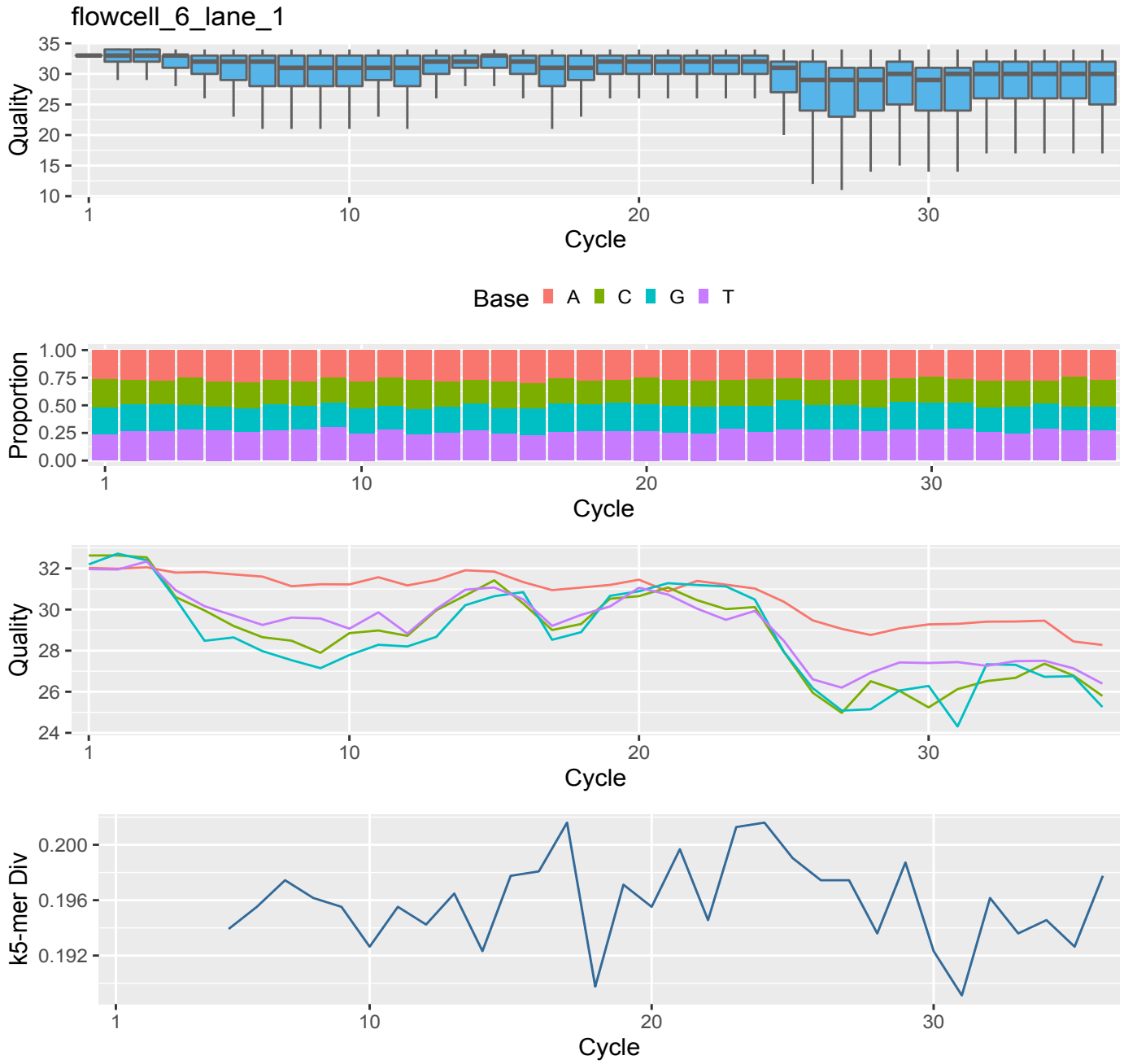
```
> seeFastqPlot(fqlist[1], arrange=seq(along=fqlist))
```

Il secondo grafico è una visualizzazione dettagliata dei dati di metilazione. A tal fine, utilizziamo i risultati dell’analisi svolta nella sezione 14.10 "Analisi dei dati di metilazione”:

```
 > library(methyAnalysis)
 > data(exampleMethyGenoSet)
```

Esaminiamo il file CSV esportato in precedenza nella sezione 14.10 e scegliamo il gene da visualizzare sulla mappa cromosomica (nel nostro caso il gene 54149, il primo gene della lista). Per tracciare la mappa, usiamo la funzione `plotMethylationHeatmapByGene()` del pacchetto *methyAnalysis* come segue:

```
> plotMethylationHeatmapByGene("54149", methyGenoSet = exampleMethyGenoSet,
      phenoData = colData(exampleMethyGenoSet), includeGeneBody = TRUE,
      genomicFeature = "TxDb.Hsapiens.UCSC.hg19.knownGene")
Ploting  C21orf91 (GeneID:54149)
```

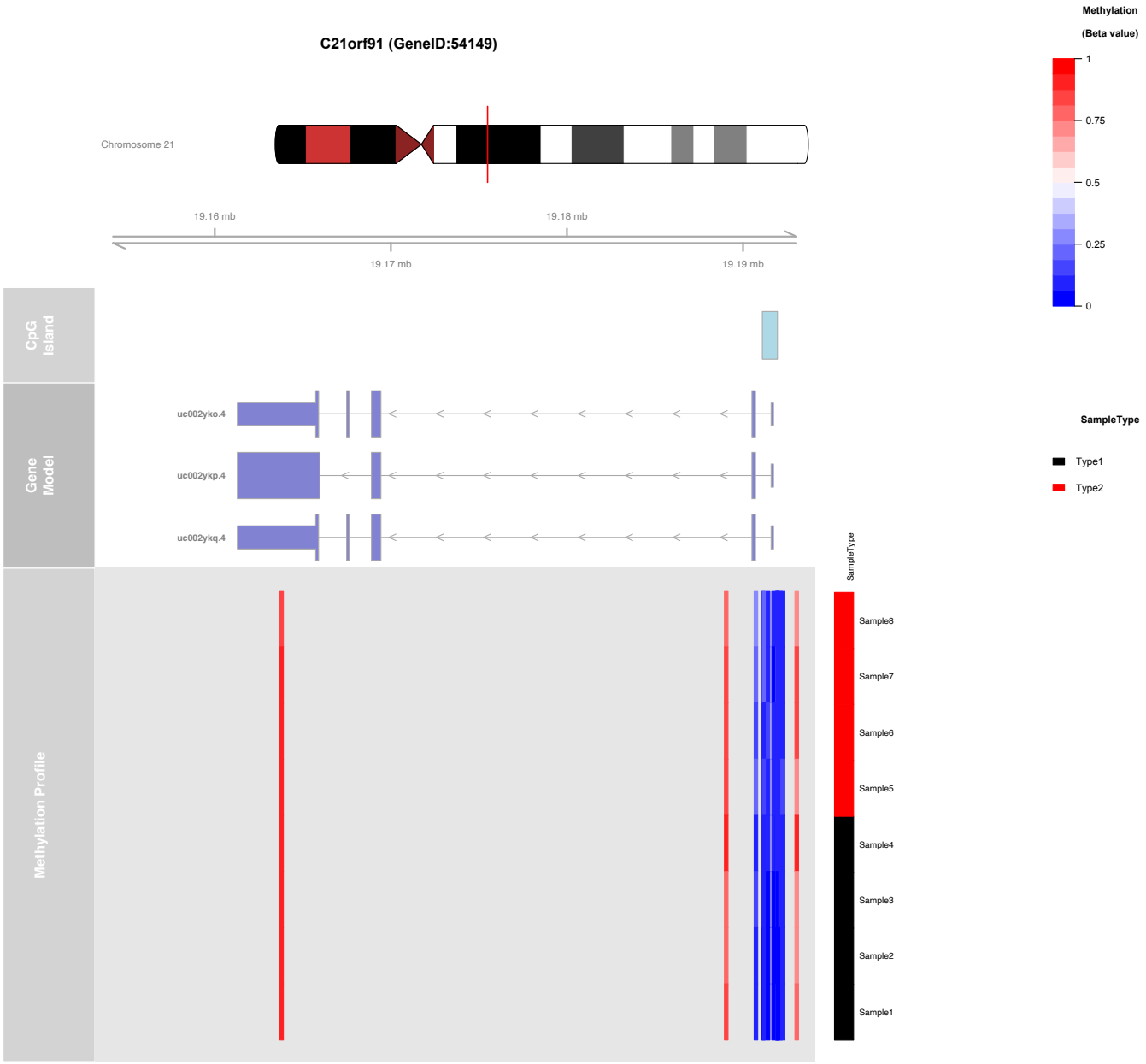


Il grafico della qualità creato (il primo della sezione) mostra la qualità e le composizioni delle basi in ogni ciclo e in particolare i seguenti caratteri:

- un boxplot della qualità (punteggio Phred);
- la proporzione delle basi (A, G, C e T);
- la qualità media delle basi;
- la diversità dei *k*-meri.

Possiamo esaminare il contenuto delle metriche calcolate per il campione (linea) *i* (ad es. 2) e la metrica *j* (ad es. 1) come segue:

```
> head(fqlist[[2]][[1]])
batchsize    nReads   klength
     1000    391970         5
```



Il secondo grafico mappa il gene 5149 sul cromosoma 21 in base alla sua posizione nei dati della mappa del genoma "TxDb.Hsapiens.UCSC.hg19.knownGene". Oltre a questo, traccia anche i valori di metilazione beta o M nella heatmap adiacente. La schermata sopra riportata mostra la heatmap per i dati di metilazione con la posizione cromosomica.

15. Il Machine Learning nella Bioinformatica

La crescita esponenziale della quantità di dati biologici disponibili pone due problemi, quello dell'archiviazione e della gestione dei dati e quello della estrazione delle informazioni utili da tali dati. Il secondo problema è una delle principali sfide della biologia computazionale; richiede lo sviluppo di strumenti e metodi in grado di trasformare tutti questi dati eterogenei in conoscenza biologica del meccanismo sottostante. Si pensi ad esempio ai dati di espressione genica che consistono in valori di espressione provenienti da una serie di esperimenti di microarray di DNA. I dati provengono da pazienti affetti da una malattia, ad esempio il cancro. Con questo tipo di dati a disposizione, è interessante cercare risposte alle seguenti domande:

- Quali campioni mostrano una maggiore similarità nei loro profili di espressione?
- C'è qualche somiglianza intra-campione tra i geni?
- Ci sono alcuni geni che caratterizzano certi tipi di cancro?

Il *Machine Learning* (apprendimento automatico) è un dominio dell'intelligenza artificiale che permette ai computer di apprendere questi aspetti, e altri ancora, da dati di esempio o da esperienze passate. L'apprendimento automatico può creare e ottimizzare modelli in grado di prevedere i risultati sulla base delle caratteristiche dei dati conosciuti (di solito chiamati *training data*, dati di addestramento).

Le due impostazioni principali nell'apprendimento automatico sono l'apprendimento *supervisionato* e quello *non supervisionato*. Nell'apprendimento supervisionato abbiamo i valori dei campioni nel training set e usiamo questa conoscenza per predire i valori per il campione sconosciuto (il *test* set). La *classificazione* è un apprendimento supervisionato in cui abbiamo le categorie (classi) dei campioni noti nel training set e creiamo i modelli in base a questo set per predire le classi dei casi sconosciuti. L'apprendimento non supervisionato utilizza invece un insieme di istanze di dati senza risultati noti a priori. Esso scopre dei modelli (pattern, cluster) nei dati e quindi li suddivide in sottoinsiemi significativi (queste categorie non sono predefinite come nel caso dell'apprendimento supervisionato). Il *clustering* è un esempio di apprendimento non supervisionato in cui l'analisi esplorativa dei dati viene eseguita per trovare modelli per identificare le somiglianze e le differenze tra le istanze di dati.

In questo capitolo approfondiremo come affrontare i metodi di machine learning in bioinformatica utilizzando R. Tuttavia, prima di passare a problemi specifici in bioinformatica e alle loro soluzioni attraverso il machine learning, occorre prima definire un obiettivo chiaro, con i dati in grado di soddisfare i requisiti posti. Le questioni chiave che devono essere affrontate prima di iniziare a formulare modelli di apprendimento sono le seguenti:

- **Valutazione della qualità dei dati.** La correttezza e la completezza dei dati è necessaria per far funzionare il modello nel modo desiderato. Ciò include la verifica della coerenza dei valori e dei tipi di attributi, la verifica dei valori mancanti e la ricerca dei valori anomali (*outliers*).

- **Normalizzazione dei dati.** A seconda dei dati di input e dell'algoritmo scelto, a volte è necessario standardizzare i dati per renderli confrontabili. Tuttavia, potrebbe non essere necessario nei casi in cui la scala originaria dei dati è significativa. Allo stesso modo, il metodo di normalizzazione può essere diverso a seconda del tipo di caratteristiche (*feature*). Un altro aspetto importante è che si dovrebbero prima normalizzare i dati di training e poi applicare questi parametri di normalizzazione per ridimensionare i dati di validazione.

- **Selezione delle caratteristiche**. Per un modello efficiente occorre selezionare le caratteristiche che mostrano una dipendenza tra l'istanza dei dati e l'obiettivo (per esempio, le etichette di classe nel caso della classificazione). A seconda dei dati, dell'obiettivo e dell'algoritmo di apprendimento, potrebbero non essere necessarie tutte le caratteristiche per creare un modello; le feature superflue aggiungono rumore, ridondanza o causano inefficienza di calcolo per il modello. Pertanto, è necessario identificare le caratteristiche utili e significative che possono definire correttamente la funzione obiettivo.

Ci sono molti altri aspetti da considerare, che saranno illustrati quando necessario. Si consiglia di consultare un testo di base sul Machine Learning per maggiori informazioni (ad esempio *The Elements of Statistical Learning* di Hastie, Tibshirani e Friedman, disponibile all'indirizzo http://statweb.stanford.edu/~tibs/ElemStatLearn/printings/ESLII_print10.pdf). Anche l'articolo *Machine learning in bioinformatics* di Larrañaga *et al*. (http://bib.oxfordjournals.org/content/7/1/86.long) offre una buona panoramica del settore nel contesto della bioinformatica.

I dati che utilizzeremo nelle sezioni successive sono già pre-elaborati, per semplificare le attività di costruzione e validazione dei modelli.

15.1 Clustering dei dati mediante *k*-means e clustering gerarchico

Il clustering è un processo di apprendimento non supervisionato nel data mining. Esso mira a raggruppare una serie di istanze di dati in modo tale che quelle dello stesso gruppo siano più simili tra loro rispetto a quelle di altri gruppi. Questi gruppi non sono predefiniti e sono chiamati *cluster*. L'obiettivo del clustering è quello di ridurre al minimo le distanze tra i dati all'interno del cluster e massimizzare le distanze tra i cluster. Vi sono diverse funzioni disponibili in R per eseguire vari tipi di clustering: questa sezione ne illustrerà alcune.

Per eseguire il clustering, abbiamo bisogno di un dataset su cui operare; bisogna anche, a seconda del metodo, decidere a priori il numero di gruppi in cui suddividere i dati. Come dataset, utilizzeremo i dati del lievito sui siti di localizzazione delle proteine nei batteri gram-negativi (che può essere scaricato dall'indirizzo https://www.crescenziogallo.it/pub/yeast.rda), selezionati per quattro localizzazioni e casi ed estrapolati dai dati originali all'indirizzo http://archive.ics.uci.edu/ml/machine-learning-databases/yeast/. Abbiamo quattro siti di localizzazione delle proteine, ovvero mitocondri, citosol, nucleo e membrana.

Per prima cosa, carichiamo il dataset di input per il clustering.

Useremo i dati del lievito descritti in precedenza, selezionando le colonne da 2 a 9 che corrispondono alle otto features:

mcg (metodo di McGeoch per il riconoscimento della sequenza di segnali);

gvh (metodo di von Heijne per il riconoscimento della sequenza di segnali);

alm (score del programma di predizione della regione di estensione della membrana ALOM);

mit (score dell'analisi discriminante del contenuto di aminoacidi della regione N-terminale — lunga 20 residui — di proteine mitocondriali e non mitocondriali);

erl (presenza/assenza della sottostringa "HDEL", che si ritiene agisca come segnale di ritenzione nel lume del reticolo endoplasmatico);

pox (segnale di individuazione del perossisoma nel C-terminale);

vac (score dell'analisi discriminante del contenuto di aminoacidi nelle proteine vacuolari ed extracellulari);

nuc (score dell'analisi discriminante dei segnali di localizzazione di proteine nucleari e non nucleari).

La decima colonna è la classe, cioè il sito di localizzazione:

```
> load("yeast.rda")
> head(yeast[,2:10])
    mcg  gvh  alm  mit erl pox  vac  nuc class
6  0.51 0.40 0.56 0.17 0.5 0.5 0.49 0.22     1
10 0.40 0.39 0.60 0.15 0.5 0.0 0.58 0.30     1
13 0.40 0.42 0.57 0.35 0.5 0.0 0.53 0.25     1
16 0.46 0.44 0.52 0.11 0.5 0.0 0.50 0.22     1
17 0.47 0.39 0.50 0.11 0.5 0.0 0.49 0.40     1
21 0.45 0.40 0.50 0.16 0.5 0.0 0.50 0.22     1
> myData <- yeast[,2:9]
```

Definiamo il numero di cluster desiderato (4 perché ci sono quattro siti di localizzazione delle proteine):

```
> k <- 4
```

Iniziamo con il clustering *k*-means. Per eseguirlo utilizziamo la funzione `kmeans()`, prendendo come input i dati e il numero di cluster ed esaminiamo la struttura del risultato:

```
> kmeans_result <- kmeans(myData, k)
> str(kmeans_result)
List of 9
 $ cluster     : Named int [1:1299] 3 3 3 3 3 3 3 4 3 4 ...
  ..- attr(*, "names")= chr [1:1299] "6" "10" "13" "16" ...
 $ centers     : num [1:4, 1:8] 0.427 0.505 0.395 0.555 0.436 ...
  ..- attr(*, "dimnames")=List of 2
  .. ..$ : chr [1:4] "1" "2" "3" "4"
  .. ..$ : chr [1:8] "mcg" "gvh" "alm" "mit" ...
 $ totss       : num 88.7
 $ withinss    : num [1:4] 6.49 12.21 17.34 14.48
 $ tot.withinss: num 50.5
 $ betweenss   : num 38.2
 $ size        : int [1:4] 123 228 498 450
 $ iter        : int 4
 $ ifault      : int 0
 - attr(*, "class")= chr "kmeans"
```

Controlliamo ora il numero di proteine in ogni cluster:

```
> table(kmeans_result$cluster)

  1   2   3   4
123 228 498 450
```

Visualizziamo i cluster sullo scatterplot delle due variabili "mit" e "gvh":

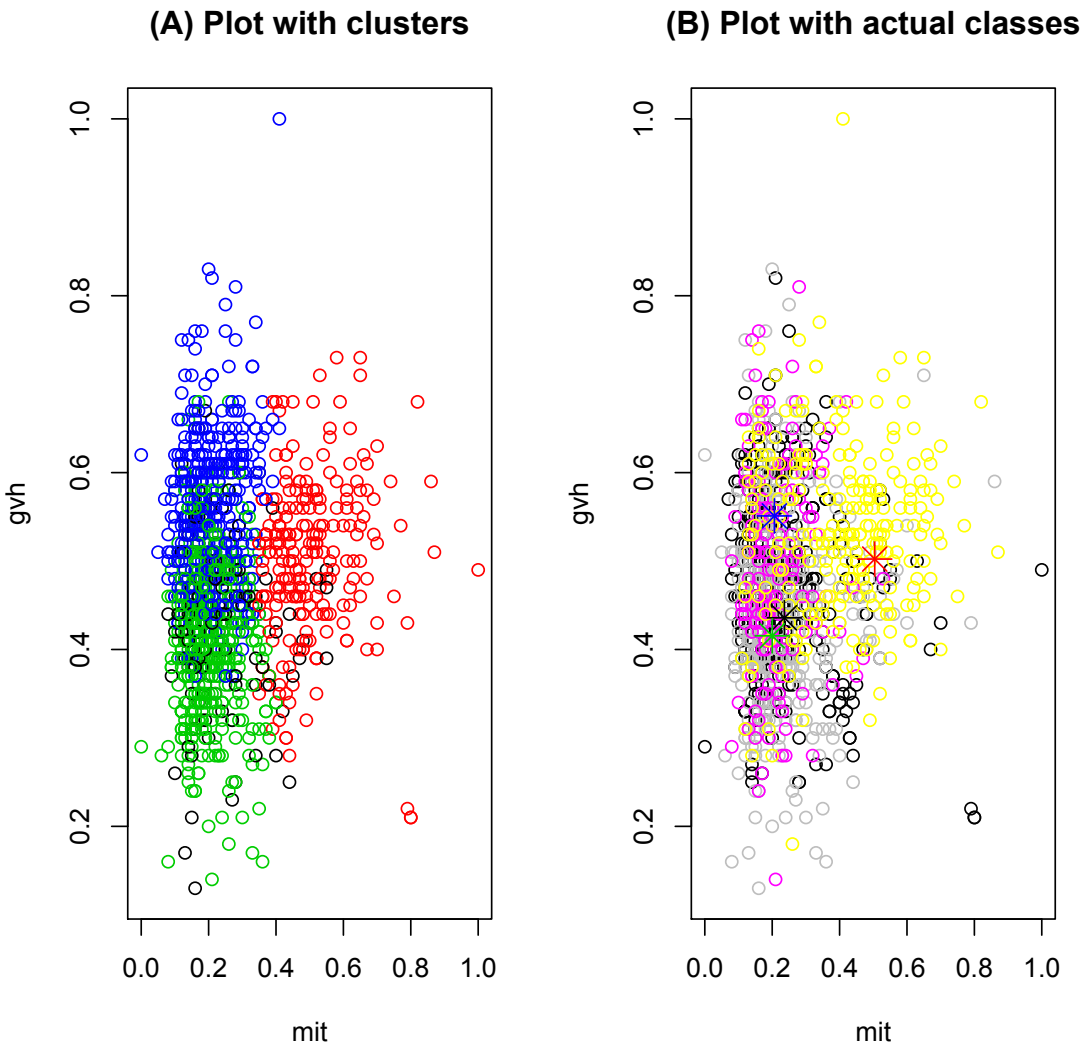
```
> par(mfrow=c(1,2))
> plot(myData[c("mit", "gvh")], col = kmeans_result$cluster,
      main = "(A) Plot with clusters")
```

Tracciamo la classe (sito di localizzazione delle proteine) e confrontiamo i risultati del clustering con i dati reali:

```
> plot(myData[c("mit", "gvh")], col = yeast$class,
      main = "(B) Plot with actual classes")
```

Aggiungiamo i centroidi dei cluster nel diagramma:

```
> points(kmeans_result$centers[,c("mit", "gvh")], col = 1:4, pch = 8, cex=2)
```



Nel grafico precedente, i colori rappresentano il cluster calcolato nel dataset (A) e le classi effettive nei dati di input (B). I centroidi dei cluster sono mostrati come stelle — i colori corrispondono ai cluster in (A) — insieme ai punti dati in (B). In entrambi i diagrammi le feature *mit* e *gvh* sono rappresentate rispettivamente lungo gli assi *x* e *y*.

Il passo successivo è il clustering gerarchico. Creiamo prima (per semplificare l'elaborazione successiva) una versione ridotta del dataset di soli 100 elementi nella variabile *yeast100*:

```
> dim(yeast)
[1] 1299   10
> yeast100 = yeast[sample(seq(1:1299), 100, replace = FALSE),]
> yeast100
       Sequence  mcg  gvh  alm  mit erl pox  vac  nuc class
1377 YBG6_YEAST 0.28 0.41 0.46 0.18 0.5   0 0.47 0.39     8
946  RPB3_YEAST 0.54 0.57 0.55 0.18 0.5   0 0.50 0.22     8
1046 DHSB_YEAST 0.53 0.51 0.55 0.57 0.5   0 0.46 0.22     7
…
1216 SUP1_YEAST 0.41 0.49 0.55 0.13 0.5   0 0.55 0.22     1
```

Ora creiamo una matrice di distanza tra le features (esclusa la classe):

```
> myData_100 <- yeast100[,2:9]
> myDist <- dist(myData_100)
```

Effettuiamo il clustering gerarchico applicando la funzione `hclust()` alla matrice di distanza (viene applicato il metodo *average linkage*, nel quale la distanza tra due cluster è definita come la distanza media tra ciascun punto di un cluster rispetto a ciascun punto nell'altro cluster):

```
> hc <- hclust(myDist, method="ave")
```

Controlliamo i cluster ottenuti e limitiamo l'albero gerarchico a tre cluster:

```
> groups <- cutree(hc, k=3)
> table(groups)
groups
 1  2  3
89  6  5
```

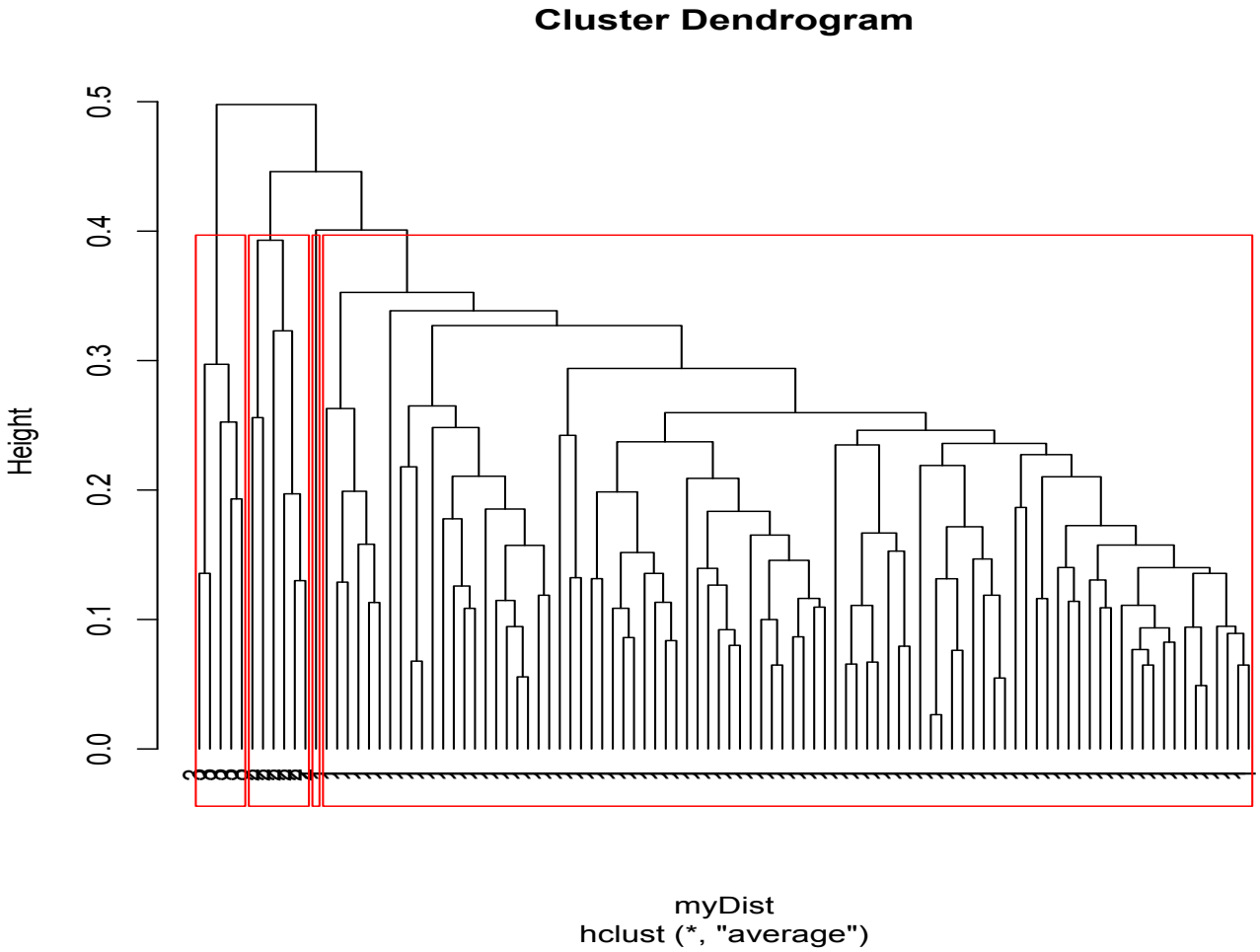
Il clustering gerarchico così formato può essere tracciato come dendrogramma:

```
> plot(hc, hang = -1, labels = groups)
```

Per una migliore visualizzazione, tracciamo un rettangolo attorno ai cluster:

```
> rect.hclust(hc, k = 4, which = NULL, x = NULL, h = NULL, border = 2,
      cluster = NULL)
```

Nel grafico seguente i cluster sono racchiusi all'interno di rettangoli rossi:



Gli approcci di clustering mostrati in precedenza utilizzano due metodi diversi, ovvero *k*-means e il clustering gerarchico. Il primo passo è quello di preparare i dati e importarli nella sessione R. Abbiamo selezionato le feature dei dati (colonne da 2 a 9, la decima è la classe) e impostato il numero di cluster *k* a 4, avendo quattro classi nei dati. L'attributo di classe è stato quindi rimosso dai dati, potendo altrimenti causare distorsioni nel processo di clustering. Una volta pronti i dati, abbiamo applicato gli algoritmi di clustering implementati nelle funzioni R incorporate, come `kmeans()` e `hclust()`.

L'algoritmo *k*-means assegna in modo casuale i cluster ai punti dati in ingresso e poi calcola i centroidi dei cluster. Questo processo viene aggiornato e rieseguito fino a quando il clustering converge.

Il clustering gerarchico calcola prima le distanze tra i punti dati in base alle misurazioni delle caratteristiche disponibili. Per impostazione predefinita, la metrica di distanza utilizzata nella funzione `dist()` è quella euclidea (le altre opzioni sono Manhattan, Maximum, e così via). La matrice di distanza viene poi usata per costruire un dendrogramma. Una volta che il dendrogramma è pronto, viene determinato un taglio ottimale per ottenere il numero desiderato di cluster nei dati (di solito conviene provare diversi numeri di cluster e osservare i risultati).

Per effettuare il clustering è consigliabile usare prima il clustering gerarchico per visualizzare il risultato e decidere quanti gruppi (*k*) devono essere considerati; quindi utilizzare il valore *k* per calcolare i cluster con altri metodi (ad es. *k*-means).

Ci sono altri metodi di clustering in R; ad esempio, i pacchetti *som* e *kohonen* per le mappe auto-organizzanti. I metodi di clustering basati sulla densità sono implementati nel pacchetto *fpc*. Il clustering fuzzy può essere eseguito con il pacchetto *e1071* tramite la funzione `cmeans()`. Il pacchetto *e1071* è disponibile nel repository CRAN all'indirizzo http://cran.r-project.org/web/packages/e1071/index.html.

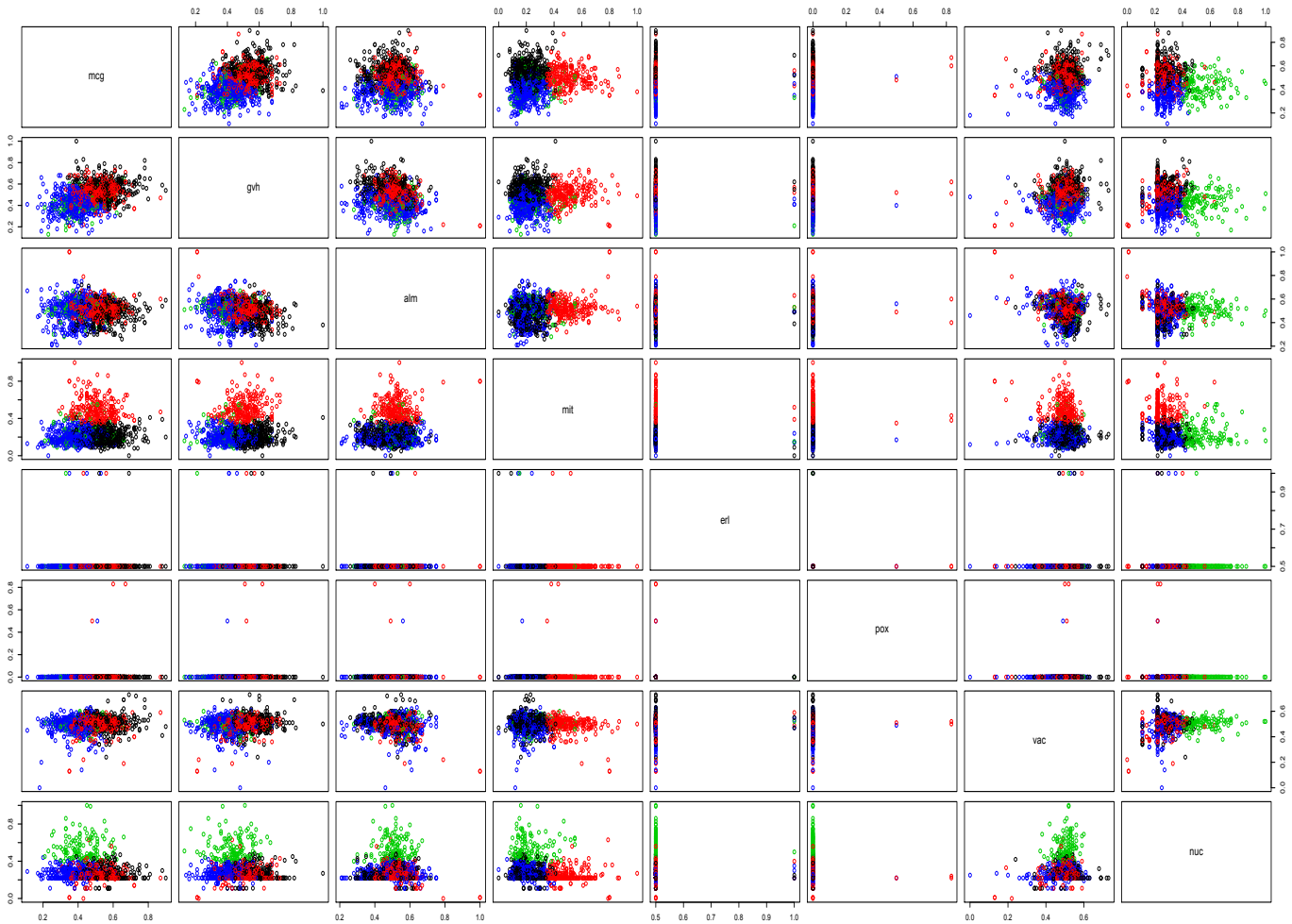
Il capitolo *A Survey of Clustering Data Mining Techniques* di Berkhin all'indirizzo http://link.springer.com/chapter/10.1007/3-540-28349-8_2 presenta una rassegna delle tecniche di clustering.

15.2 Visualizzazione dei cluster

La visualizzazione dei cluster in uno spazio bidimensionale o tridimensionale fornisce informazioni più intuitive all'osservatore in termini di posizione dei punti dati nello spazio delle feature e della separazione e aggregazione di questi dati. La visualizzazione consiste solitamente nella posizione di un punto dati lungo l'asse della feature. Un altro interessante grafico per il clustering è il grafico di *silhouette*, che illustra la qualità del clustering. In questa sezione vedremo alcuni tipi di visualizzazione dei cluster *k*-means come già visto per la visualizzazione del clustering gerarchico.

Creiamo un semplice grafico dei cluster ottenuti, tracciando i dati rispetto ai cluster e colorando i punti in base ai cluster:

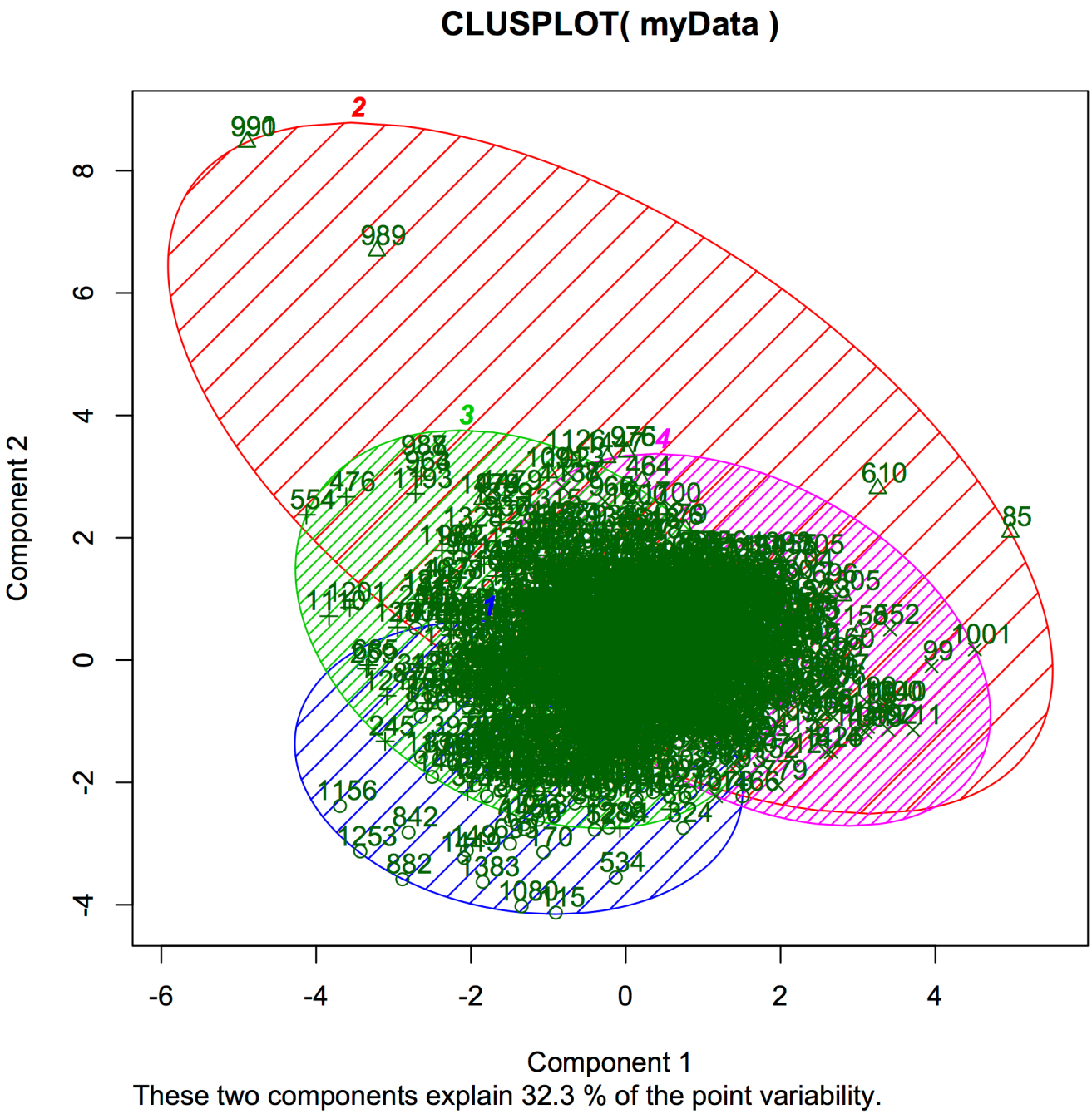
```
> plot(myData, col = kmeans_result$cluster)
```



Per eseguire l'ombreggiatura dell'area di clustering, utilizziamo il pacchetto *cluster* e ombreggiamo le aree con i colori corrispondenti mediante la funzione `clusplot()`:

```
> library(cluster)
> clusplot(myData, kmeans_result$cluster, color=T, shade=T, labels=2, lines=0)
```

Nella schermata seguente, gli assi *x* e *y* rappresentano le due componenti che spiegano la quantità massima di variabilità:



Per osservare le posizioni dei centroidi insieme ai dati abbiamo bisogno del pacchetto *vegan*:

```
> install.packages("vegan")
> library(vegan)
```

Assegniamo i gruppi dai risultati del clustering e iniziamo il grafico:

```
> groups <- levels(factor(kmeans_result$cluster))
> ordiplot(cmdscale(dist(myData)), type = "n")
```

Assegniamo i colori ai cluster:

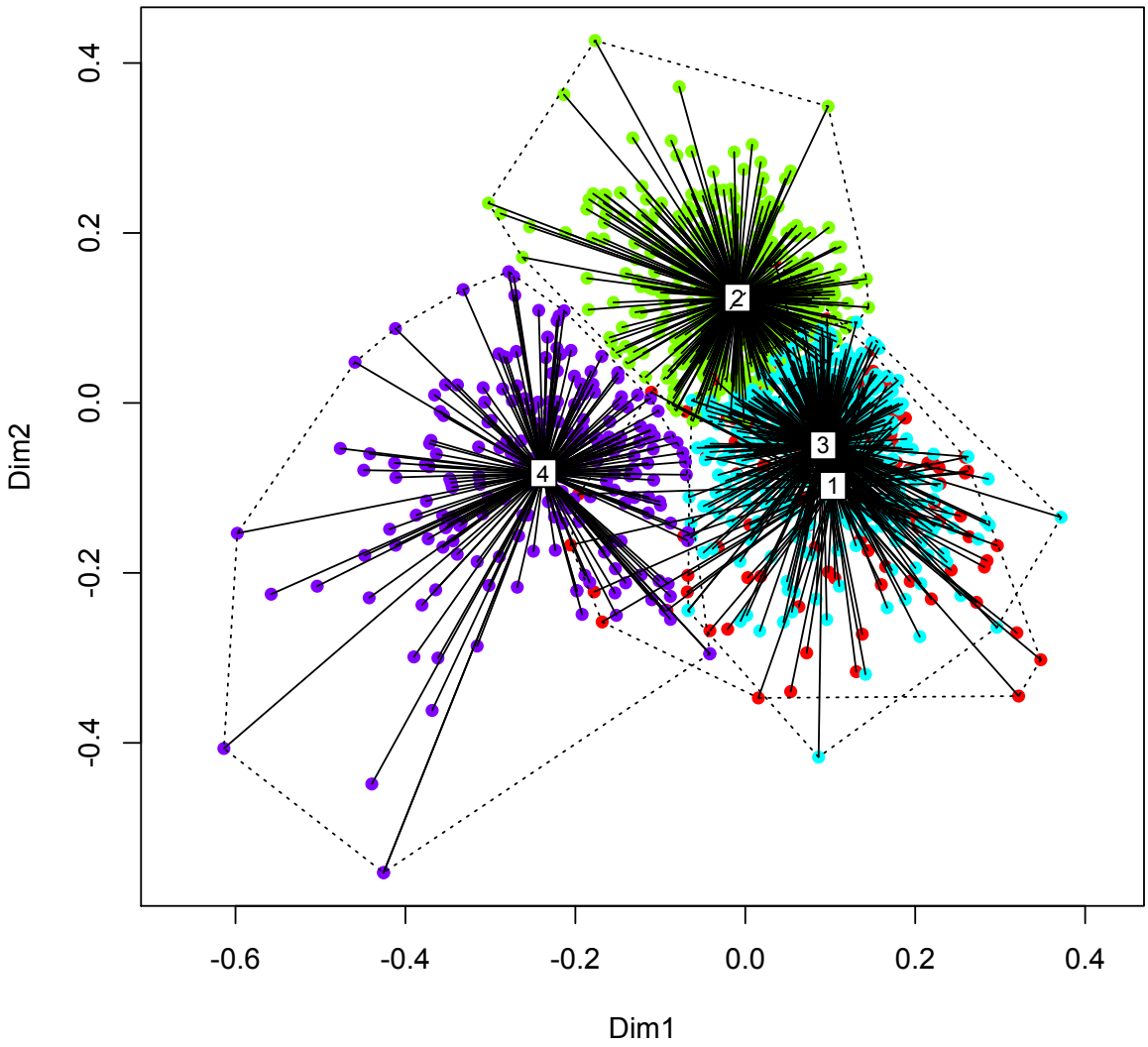
```
> cols <- rainbow(nlevels(factor(kmeans_result$cluster)))
```

Quindi tracciamo i punti dati all'interno del grafico:

```
> for (i in seq_along(groups)) {
      points(cmdscale(dist(myData))[factor(kmeans_result$cluster) ==
            groups[i], ], col = cols[i], pch = 16)
  }
```

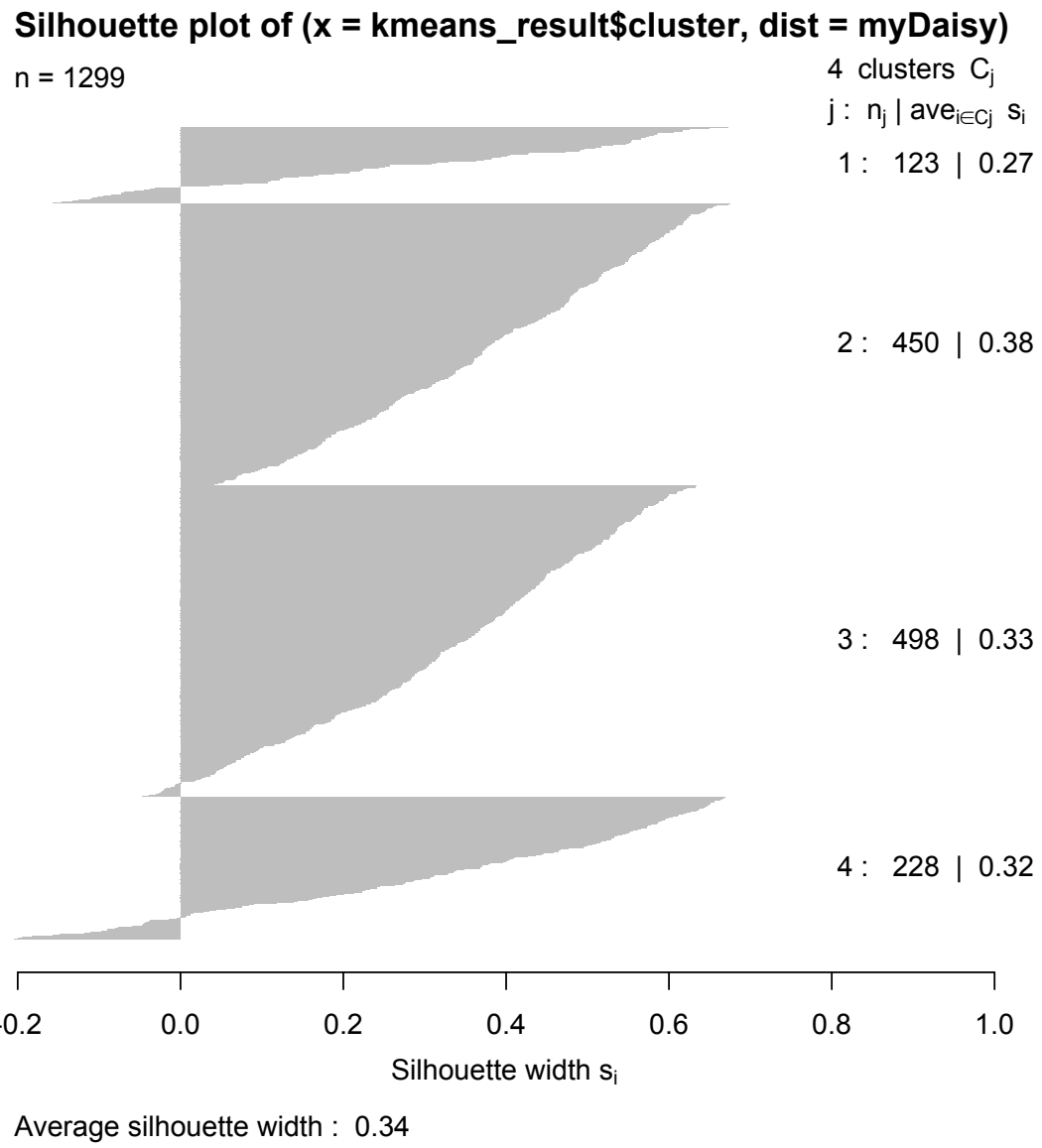
Colleghiamo i punti dati di ogni cluster con i loro centroidi:

```
> ordispider(cmdscale(dist(myData)), factor(kmeans_result$cluster), label = TRUE)
> ordihull(cmdscale(dist(myData)), factor(kmeans_result$cluster), lty = "dotted")
```



Il *silhouette plot* mostra la qualità del clustering in termini di separazione e aggregazione dei punti dati. Può essere tracciato come segue:

```
> myDaisy <- (daisy(myData))^2
> mySil <- silhouette(kmeans_result$cluster, myDaisy)
> plot(mySil)
```



Il primo grafico creato visualizza gli scatterplot sulle coppie delle otto feature della matrice dei dati e colora i punti a seconda dei cluster ad essi assegnati.

Nel grafico successivo, mediante la funzione `clusplot()` del pacchetto *cluster* abbiamo evidenziato la regione per ogni cluster intorno al suo centroide.

Il terzo grafico utilizza il pacchetto *vegan* e mostra il centroide e le reti di connessione di ciascun punto dati con il centroide del cluster di appartenenza. La libreria *vegan* ha diverse utilità per calcolare le matrici di distanza; abbiamo utilizzato la funzione `dist()` per calcolare le distanze tra le coppie di dati. La funzione `cmdplot()` calcola il classico scaling multidimensionale (analisi delle coordinate principali) di una matrice di dati e restituisce un insieme di punti in modo che le distanze tra i punti siano

approssimativamente uguali alle loro distanze. Il grafico vuoto iniziale è stato riempito con i punti dati (argomento `type=n` della funzione `ordiplot()`) e ad esso sono state aggiunte le connessioni tra i punti dati e i centroidi dei cluster. Questo grafico può essere utilizzato per visualizzare la distribuzione dei dati nello spazio delle feature.

Infine il grafico della silhouette (*silhouette plot*) rappresenta i seguenti aspetti:

- il *numero* di cluster, ciascuno rappresentato da un'area grigia orizzontale;
- l'*altezza* di queste regioni grigie è proporzionale al numero di punti dati all'interno del gruppo corrispondente;
- l'*area* grigia è formata da un insieme di dati all'interno di ogni cluster rappresentato da linee che si estendono orizzontalmente fino alla larghezza della silhouette (SW, *silhouette width*), che rappresentano l'appertenenza di un punto ad un cluster rispetto agli altri. Più alta è la SW, migliore è l'adattamento;
- l'*indice di silhouette* è la similarità media di ogni punto con il proprio cluster meno la similarità media con il cluster successivo più vicino.

Il *silhouette plot* utilizza la libreria *cluster* di R per calcolare le distanze nel dataset e creare l'oggetto silhouette per i risultati del cluster. Si possono quindi tracciare i risultati utilizzando la normale funzione `plot()` di R. L'indice di silhouette fornisce una misura del grado di appartenenza di ogni elmento all'interno del suo cluster: nel nostro caso il grafico mostra l'indice di silhouette più alto (0,38) per il cluster 2.

L'articolo *Silhouettes: A graphical aid to the interpretation and validation of cluster analysis* di Peter J. Rousseeuw (http://www.sciencedirect.com/science/article/pii/0377042787901257) illustra l'indice di silhouette e il relativo grafico.

15.3 Apprendimento supervisionato per la classificazione

Come il clustering, anche la classificazione riguarda la categorizzazione delle istanze di dati, ma in questo caso le categorie sono note e vengono definite come *classi*. L'obiettivo è quello di identificare la categoria (classe) cui appartiene un nuovo elemento dato; si utilizza un dataset in cui le classi sono note per costruire il modello predittivo. La classificazione è un caso di apprendimento supervisionato in cui l'algoritmo di apprendimento prende un insieme noto di dati di input e le risposte corrispondenti (etichette di classe) e costruisce un modello predittore che genera previsioni ragionevoli per le classi nei dati sconosciuti. Ad esempio, si supponga di avere dei dati di espressione genica di pazienti affetti da cancro e di pazienti sani. Il modello di espressione genica di questi ultimi può aiutare a definire un modello per decidere se un paziente ha il cancro o meno. In questo caso, se abbiamo un insieme di campioni per i quali conosciamo il tipo di tumore, i dati possono essere utilizzati per costruire un modello in grado di identificare il tipo di tumore. In seguito, questo modello può essere applicato per predire il tipo di tumore in casi sconosciuti.

Vi sono alcune considerazioni da tenere a mente quando si esegue la classificazione. Bisogna innanzitutto assicurarsi di avere abbastanza dati per addestrare il modello. L'apprendimento con insiemi insufficienti di

dati non permetterà al classificatore di apprendere con sufficiente capacità di generalizzazione, dando luogo ad una classificazione imprecisa. Inoltre, le fasi di pre-elaborazione (come la normalizzazione) per i dati di addestramento (*training*) e di validazione (*test*) dovrebbero essere gli stessi. È anche importante mantenere distinti i dati relativi al training e al test: se si effettua l'apprendimento su tutti i dati e poi si utilizza una parte dei dati per i test porta ad un fenomeno chiamato *overfitting*.

Esistono diversi metodi di classificazione. In questa sezione ne vedremo alcuni: l'analisi discriminante lineare (LDA, *linear discriminant analysis*), l'albero decisionale (DT, *decision tree*) e la *support vector machine* (SVM).

Per eseguire la classificazione abbiamo bisogno in primo luogo di un dataset con classi note (*training set*); in secondo luogo, il dataset di prova su cui deve essere testato il classificatore (*test set*). Oltre a questo, useremo alcuni pacchetti R che saranno discussi quando necessario. Il dataset che useremo contiene 83 record, ciascuno relativo a 2.308 geni di cellule tumorali e identificato da uno di quattro diversi tipi di tumori (EW=sarcome di Ewing, BL=linfoma di Burkitt, NB=neuroblastoma e RM=rhabdomyo sarcoma, usati come etichette di classe). Selezioneremo 60 record per l'addestramento e i restanti 23 per il test. Per maggiori informazioni sul dataset consultare l'articolo *Classification and diagnostic prediction of cancers using gene expression profiling and artificial neural networks* di Khan *et al*. (http://research.nhgri.nih.gov/microarray/Supplement/). Il dataset è stato pre-elaborato in un formato facilmente utilizzabile in R (un data.frame *mldata* con 83 righe e 2.309 colonne, l'ultima è la classe tumorale) ed è disponibile per il download all'indirizzo https://www.crescenziogallo.it/pub/cancer.rda.

Per prima cosa, carichiamo la libreria *MASS* che comprende alcune delle funzioni di classificazione:

```
> library(MASS)
```

Ora abbiamo bisogno dei dati per addestrare e testare i classificatori. Carichiamo il data.frame *mldata* dal file "cancer.rda" (supponendo che sia stato scaricato nella directory di lavoro):

```
> load("cancer.rda")
> str(mldata)
'data.frame':     83 obs. of  2309 variables:
 $ 21652   : num  1.15 3.204 2.319 1.984 0.141 ...
…

$ 323577  : num  1.503 2.283 1.918 1.539 0.671 ...
  [list output truncated]
```

Estraiamo un campione di 60 record per il training e i restanti 23 per il test, assicurandoci che i due set di dati non si sovrappongano e non siano condizionati da alcun tipo specifico di tumore (campionamento casuale):

```
> train_row <- sample(1:83, 60)
> train <- mldata[train_row,] # use sampled indexes to extract training data
> test <- mldata[-train_row,] # test set is select by selecting all the other data points
```

Conserviamo la classe (per i confronti nella fase di training), che corrisponde alla colonna `tumor`, e la rimuoviamo invece dai dati di test:

```
> testClass <- test$tumor
> test$tumor <- NULL
```

Proviamo ora il metodo `lda` (analisi discriminante lineare) per ottenere il modello di classificazione:

```
> myLD <- lda(tumor ~ ., train)
```

Usiamo il classificatore ottenuto per predire le etichette sul set di prova:

```
> testRes_lda <- predict(myLD, newdata=test)
```

Per verificare il numero di previsioni corrette ed errate, è sufficiente confrontare le classi previste con la variabile *testClass* creata in precedenza:

```
> sum(testRes_lda$class == testClass) # correct prediction
[1] 9
> sum(testRes_lda$class != testClass) # incorrect prediction
[1] 14
```

Proviamo ora un altro semplice classificatore, il *decision tree* (DT), per il quale è necessario il pacchetto *rpart*:

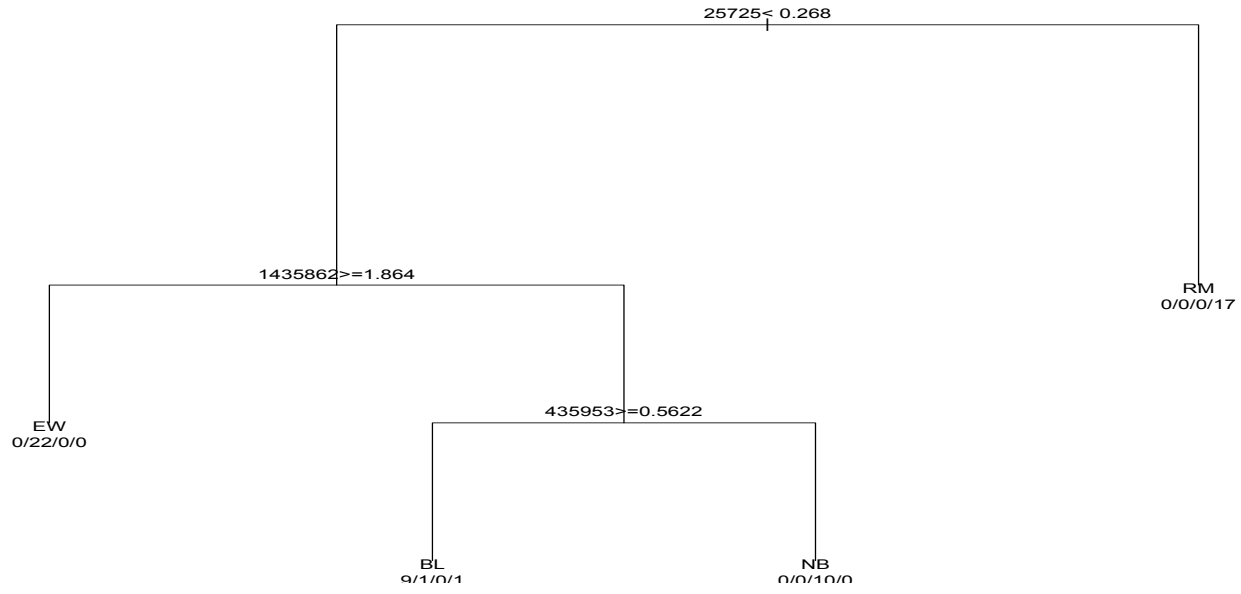
```
> library(rpart)
```

Creiamo l'albero decisionale sulla base dei dati di training:

```
> myDT <- rpart(tumor ~ ., data = train, control = rpart.control(minsplit = 10))
```

Tracciamo il grafico dell'albero mediante i comandi seguenti:

```
> plot(myDT)
> text(myDT, use.n=T)
```



Il grafico mostra il criterio di suddivisione per ogni feature (sui rami) per differenziare le classi. Verifichiamo il classificatore dell'albero decisionale sui dati di test utilizzando la funzione di previsione:

```
> testRes_dt <- predict(myDT, newdata=test)
```

Diamo un'occhiata alle predizioni del classificatore per alcune istanze (record) di dati (1 se il tipo di tumore è previsto nel record, 0 altrimenti):

```
> classes <- round(testRes_dt)
> head(classes)
   BL EW NB RM
1   0  0  0  1
4   0  0  0  1
14  0  0  0  1
17  0  0  0  1
19  0  1  0  0
25  0  0  0  1
```

Conteggiamo il numero di "successi" previsionali per i dati di test:

```
> classlab = c("BL","EW","NB","RM")
> names(classlab) = c(8,4,2,1) # binary powers corresponding to tumor labels
> ok = 0
> for (i in 1:23) {
      lab = classes[i,4]*2^0+classes[i,3]*2^1+classes[i,2]*2^2+classes[i,1]*2^3
      if (testClass[i] == classlab[paste(lab,sep="")]) {ok = ok+1}
  }
> ok # no. of correct predictions
[1] 17
```

Infine, utilizziamo la SVM. Per fare questo è necessario un altro pacchetto R denominato *e1071*:

```
> library(e1071)
```

Creiamo il classificatore `svm` dai dati di training:

```
> mySVM <- svm(tumor ~ ., data = train)
```

Utilizziamo quindi il classificatore, l'oggetto *mySVM*, per i dati di test. Esaminiamo le classi previste per ogni record e conteggiamo il numero di "successi" del classificatore:

```
> testRes_svm <- predict(mySVM, newdata=test)
> testRes_svm
 1  4 14 17 19 25 32 37 43 45 48 55 56 59 61 62 66 68 69 75 77 78 80
EW RM NB RM EW NB EW EW EW EW EW BL BL NB NB NB NB NB RM RM RM RM RM
> testClass
NB RM NB RM EW NB EW EW EW EW EW BL BL NB NB NB NB NB RM RM RM RM RM
sum(testRes_svm == testClass)
[1] 22
```

Abbiamo iniziato la sezione caricando il dataset sui tumori. I metodi di apprendimento supervisionati che abbiamo visto hanno utilizzato due set di dati: il *training* set e il *test* set. Il training set riporta le informazioni della classe e viene utilizzato per creare un classificatore e addestrarlo a distinguere le classi. Questo modello viene poi applicato sul test set per predire la classe. Per identificare i due set abbiamo campionato casualmente 60 record di dati per il training e utilizzato i restanti 23 per il test.

I metodi di apprendimento supervisionati illustrati in questa sezione seguono un principio diverso. LDA cerca di modellare la differenza tra le classi in base alla combinazione lineare delle sue caratteristiche. Questa funzione di combinazione costituisce il modello in base al training set e viene utilizzata per prevedere le classi nel test set. Il modello addestrato su 60 casi viene poi utilizzato per predire i restanti 23.

DT è un metodo diverso. Esso costruisce alberi di regressione che formano un insieme di regole per distinguere una classe dall'altra. L'albero addestrato sul training set viene poi applicato per predire le classi del test set o altri set di dati simili.

La SVM è una tecnica di classificazione relativamente complessa. Essa mira a creare uno o più iperpiani nello spazio delle feature, rendendo i punti dati separabili lungo questi piani. Anche qui l'addestramento viene fatto sul training set e il classificatore ottenuto viene poi utilizzato per assegnare le classi ai nuovi dati. In generale, la LDA usa la combinazione lineare e la SVM usa le dimensioni multiple come iperpiano per la distinzione dei dati. In questa sezione abbiamo usato la funzione `svm()` del pacchetto *e1071*, che offre numerose opzioni per il training e può essere utilizzata sia per problemi di classificazione che di regressione.

Negli esempi precedenti il classificatore migliore è risultato quello SVM, con 22 predizioni corrette su 23, pari a circa il 96%.

Uno dei tool più popolari nella comunità machine learning è WEKA. È uno strumento basato su Java e implementa molte librerie per eseguire attività di classificazione (e regressione) mediante i metodi DT, LDA, Random Forest e così via. R supporta un'interfaccia per WEKA tramite una libreria chiamata *RWeka*. È disponibile sul repository CRAN all'indirizzo https://cran.r-project.org/web/packages/RWeka/ e si basa su *RWekajars*, un pacchetto separato, per utilizzare le librerie Java in esso contenute che implementano diversi classificatori.

Il testo *Elements of Statistical Learning* di Hastie, Tibshirani e Friedman (disponibile all'indirizzo http://statweb.stanford.edu/~tibs/ElemStatLearn/printings/ESLII_print10.pdf) fornisce ulteriori informazioni sui metodi LDA, DT e SVM.

15.4 Apprendimento probabilistico con Naïve Bayes

Abbiamo sinora visto alcuni metodi di classificazione non probabilistici. È opportuno dare anche un'occhiata ai metodi probabilistici di classificazione e di data mining. In questa sezione descriviamo *Naïve Bayes*, un metodo di classificazione probabilistico estremamente semplice che si basa sull'assunzione di caratteristiche indipendenti nel set di dati. Ciò significa che la presenza o l'assenza di una particolare caratteristica (*feature*) è indipendente (non correlata) dalla presenza (o assenza) di qualsiasi altra caratteristica. A scopo illustrativo, immaginiamo che una malattia D possa essere causata dalla mutazione di

tre geni a, b e c. Supponendo che questi geni siano indipendenti, le informazioni di questi geni possono essere combinate in un classificatore Naïve Bayes con un nodo per la malattia D come "progenitore" per le variabili indipendenti a, b e c.

Il metodo Naïve Bayes richiede piccoli training set per stimare i parametri (le medie e le varianze delle variabili) necessari per la classificazione. Poiché si basa sul presupposto che le variabili sono indipendenti, è necessario calcolare solo le varianze delle variabili per ogni classe e non l'intera matrice di covarianza.

I prerequisiti sono gli stessi delle sezioni precedenti: il training set e il test set (continueremo ad utilizzare i set creati per i classificatori precedenti). Oltre a questo abbiamo bisogno anche della libreria R *e1071*, che andiamo a caricare:

```
> library(e1071)
```

Addestriamo il modello di classificazione utilizzando la funzione `naiveBayes()` del pacchetto *e1071*:

```
> model <- naiveBayes(tumor ~ ., data = train)
```

Una volta ottenuto il modello, lo usiamo per predire le classi nel test set:

```
> testRes_nb <- predict(model, newdata=test)
```

Confrontiamo le classi previste (*testRes_nb*) con le classi effettive (*testClass*) formando una tabella di contingenza (nota anche come *matrice di confusione*), da cui si evince che il classificatore ha ottenuto un'accuratezza del 100%:

```
> table(testRes_nb, testClass)
          testClass
testRes_nb BL EW NB RM
        BL  5  0  0  0
        EW  0  6  0  0
        NB  0  0  5  0
        RM  0  0  0  7
```

La libreria *e1071* implementa il metodo di classificazione Naïve Bayes. Il metodo calcola le probabilità a posteriori condizionate della classe sulla base della regola di Bayes, dati i valori delle feature che sono considerati indipendenti l'uno dall'altro. Queste probabilità creano il modello, che viene poi utilizzato per predire le classi nel test set o per i nuovi dati. L'argomento formula `"tumor ~ ."` nella funzione rappresenta la classe e le variabili da cui essa dipende. I valori della classe devono essere presenti nel training set, ma naturalmente mancano nel test set.

L'articolo *Understanding Probabilistic Classifiers* di Garg e Roth (http://link.springer.com/chapter/10.1007%2F3-540-44795-4_16) fornisce ulteriori dettagli sui classificatori probabilistici.

15.5 Il bootstrap nel machine learning

La verifica sul test set è un modo per valutare la robustezza di un algoritmo di apprendimento automatico e migliorarne la precisione: è un approccio semplice (noto come metodo *hold-out*) per la stima dell'accuratezza e fornisce il *bias* (o varianza) dell'estimatore. Un approccio più completo è il *bootstrap* (che abbiamo già visto nel capitolo 5), che itera la verifica su una serie di test set e migliora progressivamente le prestazioni del modello. In questa sezione illustreremo il bootstrap e come realizzarlo in R.

Per eseguire il bootstrap abbiamo bisogno della funzione (ad esempio il tipo di classificatore che vogliamo testare) e del set di dati su cui effettuarlo. In questa sezione useremo il classico dataset *iris*.

```
> data(iris)
> str(iris)
'data.frame':     150 obs. of  5 variables:
 $ Sepal.Length: num  5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
 $ Sepal.Width : num  3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...
 $ Petal.Length: num  1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 ...
 $ Petal.Width : num  0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 ...
 $ Species     : Factor w/ 3 levels "setosa","versicolor",..: 1 1 1 1 1 1 1 1 1 1 ...
> myData = iris[c(1:150),]
```

Creiamo una funzione `corPred()` che sarà applicata a tutto il set di dati. La funzione implementa un classificatore SVM e restituisce il numero di veri positivi (TP), che servirà come indicatore di performance del modello (selezionato per semplicità; si possono usare anche misure più complete delle performance come vedremo nelle prossime sezioni):

```
> corPred <- function(data, label, indices) {
+ train = myData[indices,] # indexes for training data
+ test = myData[-indices,] # indexes for test data
+ testClass = test [,label] # assigns class labels (species)
+ colnames(train)[ncol(train)] = "Class"
+ mySVM = svm(Class ~ ., data = train, cost = 100, gamma = 1) # learn model using SVM
+ myPred = predict(mySVM, newdata = test) # prediction on test set
+ TP = sum(myPred == testClass) # calculate True positives
+ return(TP)
+ }
```

Scriviamo ora la funzione che esegue il bootstrap sui dati, avente come argomenti i dati, la classe e il numero di iterazioni:

```
> myboot <- function(d, label, iter){
+ bootres = c()
+ for(i in 1:iter){
+ indices = sample(1:nrow(d), floor((2*nrow(d))/3)) # samples indexes
+ res = corPred (d, label, indices) # runs corPred function
+ bootres = c(bootres, res) # append results
+ }
+ return(list(BOOT.RES = bootres, BOOT.MEAN = mean(bootres), BOOT.SD = sd(bootres)))
+ }
```

Eseguiamo infine la funzione di bootstrap come segue:

```
> res.bs <- myboot(d = myData, label = "Species", iter = 10000)
> str(res.bs)
List of 3
 $ BOOT.RES : int [1:10000] 45 47 47 44 45 48 47 48 48 45 ...
 $ BOOT.MEAN: num 46.7
 $ BOOT.SD  : num 1.47
```

I risultati consistono nel valore TP ottenuto per ogni iterazione (BOOT.RES), nella media di tutte le iterazioni (BOOT.MEAN) e la corrispondente deviazione standard (BOOT.SD).

La parte principale della sezione consiste nelle due funzioni `corPred()` e `myboot()`. La prima estrae i dati per il training e il test dai dati originali e li utilizza per l'apprendimento e la verifica, calcolando anche il numero totale di veri positivi come indicatore di performance. È molto semplice modificarlo per calcolare altri indicatori come la sensibilità e la specificità (sarà discusso nelle prossime sezioni). La seconda funzione esegue effettivamente il bootstrap e crea anche un vettore degli indicatori di performance aggiungendo progressivamente i risultati delle iterazioni; il risultato complessivo è una lista degli indicatori di performance e della loro media e deviazione standard: una media più alta e una deviazione standard più bassa indicano un classificatore più accurato.

15.6 La cross-validation per i classificatori

Costruire un modello basato sull'apprendimento non è sufficiente per ottenere risultati ottimali. La domanda successiva è: "Quanto bene funziona il modello quando viene applicato per effettuare nuove previsioni su istanze di dati non note?" (in termini tecnici le *prestazioni predittive*). Una prima possibilità, come abbiamo già visto, è quella di riservare una parte dei dati disponibili per il test (*hold-out*). Dopo l'addestramento, questo set di prova può essere utilizzato per testare le prestazioni del modello appreso. Questa idea di base, insieme ad un'intera classe di metodi di valutazione dei modelli, viene definita *cross-validation* (CV, convalida incrociata). La CV è utile per superare il problema dell'overfitting, che si riferisce ad una condizione in cui il modello richiede più informazioni di quante i dati possano fornire.

Ci sono diversi approcci per fare il CV, il più semplice è quello che abbiamo appena descritto, cioè l'*hold-out*; gli altri metodi sono la cross-validation "*k*-fold" e la cross-validation "leave-one-out". Il primo approccio può essere visto come un'iterazione multipla dell'hold-out, nel quale i dati sono suddivisi in k sottoinsiemi e si iterano il training e il test un numero k di volte, utilizzando i primi k-1 elementi per il training e il k-esimo per il test (da cui il nome k-fold cross-validation). L'ultimo approccio "leave-one-out" è analogo al k-fold, con il valore di k impostato sul numero totale n di dati ($k = n$). Si selezionano quindi n-1 istanze per l'addestramento e l'ennesima per il test. Il test set più piccolo può portare ad una maggiore variabilità delle prestazioni predittive nel caso del metodo leave-one-out (meno combinazioni di campioni), oltre ad essere più probabile che sia più costoso dal punto di vista computazionale. In questi casi, si preferisce la cross-validation k-fold.

La domanda a questo punto è: come si definisce il numero k? Questo dipende molto da quante istanze di dati abbiamo; con un numero maggiore di istanze di dati disponibili, abbiamo più libertà di scegliere k. I valori

più accettati per *k* sono 5 e 10. Inoltre, possiamo includere la selezione delle feature all'interno del CV, formando una cross-validation *k*-fold nidificata (che non approfondiremo).

Questa sezione riguarda l'esecuzione di una cross-validation *k*-fold in R per un metodo di apprendimento a nostra scelta. Per questo scopo utilizziamo il dataset *iris*, ma solo le prime 100 istanze con due classi, in modo da semplificare l'elaborazione. Per l'algoritmo di apprendimento usiamo il metodo SVM del pacchetto *e1071*, che andiamo a caricare:

```
> library(e1071)
```

Creiamo il training set e il test set estraendo le prime 100 istanze dal dataset *iris*:

```
> data(iris)
> myData <- iris[1:100,]
> str(myData)
'data.frame':     100 obs. of  5 variables:
 $ Sepal.Length: num  5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
 $ Sepal.Width : num  3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...
 $ Petal.Length: num  1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 ...
 $ Petal.Width : num  0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 ...
 $ Species     : Factor w/ 3 levels "setosa","versicolor",..: 1 1 1 1 1 1 1 1 1 1 ...
```

Apportiamo una piccola modifica ai dati estratti per limitare il numero di classi a 2 (il dataset *iris* originario contiene le tre specie "setosa", "versicolor", "virginica"; il nostro dataset ridotto *myData* contiene solo le specie "setosa" e "versicolor"):

```
> myData$Species <- factor(as.character(myData$Species))
> str(myData)
'data.frame':     100 obs. of  5 variables:
 $ Sepal.Length: num  5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
 $ Sepal.Width : num  3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...
 $ Petal.Length: num  1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 ...
 $ Petal.Width : num  0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 ...
 $ Species     : Factor w/ 2 levels "setosa","versicolor": 1 1 1 1 1 1 1 1 1 1 ...
```

Impostiamo ora il numero di iterazioni per il CV *k*-fold a 10 e creiamo gli indici in base al numero di iterazioni campionando i dati con sostituzione come segue:

```
> k <- 10
> index <- sample(1:k, nrow(myData), replace = TRUE)
> folds <- 1:k
```

Quindi, inizializziamo un data.frame per memorizzare i risultati del CV:

```
> myRes <- data.frame()
```

Il seguente blocco di codice è il ciclo principale all'interno del quale avviene il campionamento dei dati, l'apprendimento del modello e il CV:

```
> for (i in folds) {
+ training = subset(myData, index %in% folds[-i]) # create training set
+ test = subset(myData, index %in% c(i)) # create test set
+ mymodel = svm(training$Species ~ ., data = training) # train model
+ actual = test[,ncol(test)] # get actual labels
+ temp = data.frame(predict(mymodel, test [,-ncol(test)])) # run model on test set
+ colnames(temp) = "Predicted"
+ results = data.frame(Predicted=temp, Actual=actual) # create data.frame for results
+ myRes = rbind(myRes, results) # append results for each iteration
+ }
```

Infine, costruiamo la tabella di contingenza (*matrice di confusione*) dei risultati complessivi del CV:

```
> str(myRes)
'data.frame':     100 obs. of  2 variables:
 $ Predicted: Factor w/ 2 levels "setosa","versicolor": 1 1 1 1 1 1 1 1 1 1 ...
 $ Actual   : Factor w/ 2 levels "setosa","versicolor": 1 1 1 1 1 1 1 1 1 1 ...
> table(myRes)
            Actual
Predicted    setosa versicolor
  setosa         50          0
  versicolor      0         50
```

Nelle fasi iniziali abbiamo visto come prepare i dati per l'elaborazione, con l'impostazione dei parametri come ad esempio il numero *k* di fold per il CV. All'interno del ciclo di esecuzione del CV vengono creati e campionati i set di training e test in base al numero di fold *k*. Il modello SVM viene quindi addestrato in base al training set e utilizzato per le predizioni sul test set. I risultati previsti ed effettivi vengono combinati durante le iterazioni e memorizzati nell'oggetto *myRes*, che infine abbiamo utilizzato per visualizzare la matrice di confusione del modello di classificazione.

È possibile utilizzare diversi pacchetti per eseguire il CV durante i processi di apprendimento. Le librerie R, come *randomForest* e *nlcv*, consentono questo contemporaneamente al processo di apprendimento del modello specificando particolari argomenti durante l'esecuzione della funzione di apprendimento. Si noti che *randomForest*, oltre che un pacchetto R, è anche il nome di un metodo di classificazione; il pacchetto non è inteso solo per la classificazione e la regressione basata su una foresta di alberi random, ma contiene anche gli strumenti per il CV. Tuttavia, in questa sezione abbiamo creato un nostro set di funzioni per il CV, che possono essere utilizzate per validare qualsiasi metodo di apprendimento o test statistico.

Il già citato testo *The Elements of Statistical Learning* di Hastie, Tibshirani e Friedman (http://statweb.stanford.edu/~tibs/ElemStatLearn/printings/ESLII_print10.pdf) illustra in dettaglio le tecniche di cross-validation.

L'articolo *Nested Loop Cross Validation for Classification using nlcv* di Talloen (https://r-forge.r-project.org/scm/viewvc.php/*checkout*/pkg/nlcv/inst/doc/nlcv.pdf?root=nlcv) descrive la nested loop cross-validation.

15.7 Misurare le performance dei classificatori

La misurazione della precisione degli algoritmi di apprendimento è importante quanto gli algoritmi stessi. Queste misurazioni delle prestazioni definiscono la precisione con cui i dati sono stati classificati o raggruppati. In questa sezione descriveremo il calcolo di alcune di queste misurazioni in R.

Per cominciare, abbiamo bisogno di alcuni dei risultati precedenti. Abbiamo visto la misura delle prestazioni per il clustering nella sezione 15.2 quando abbiamo parlato di *silhouette plot*. In questa sezione calcoliamo alcuni indicatori di prestazione per valutare la bontà del classificatore. Utilizzeremo per semplicità una classificazione binaria (con due etichette di classe); l'utilizzo di più classi sarà spiegato successivamente.

Per prima cosa, installiamo e carichiamo la libreria *caret*:

```
> install.packages("caret")
> library(caret)
```

Prepariamo i dati e il classificatore come segue:

Per calcolare le prestazioni del classificatore, prendiamo i risultati delle previsioni del classificatore SVM

```
> data (iris)
> myData <- iris
> indices <- sample(1:nrow(myData), floor((2*nrow(myData))/3))
> train <- myData [indices,]
> test <- myData [-indices,]
> testClass <- test [,"Species"]
> mySVM <- svm (Species ~ ., data = train, cost = 100, gamma = 1)
```

insieme alle classi effettive:

```
> myPred <- predict(mySVM, test)
> myLabels <- testClass
```

Quindi, calcoliamo la sensibilità e la specificità del classificatore utilizzando la funzione `confusionMatrix()` del pacchetto *caret*:

```
> myCM <- confusionMatrix(myPred, myLabels)
```

Estraiamo la matrice di confusione dall'oggetto creato in precedenza:

```
> CMtable <- myCM$table
> CMtable
            Reference
Prediction   setosa versicolor virginica
  setosa         15          0         0
  versicolor      0         20         2
  virginica       1          1        11
```

Estraiamo infine la sensibilità, la specificità e gli altri parametri di valutazione del classificatore per ogni classe come segue:

```
> myPerf2 <- myCM$byClass
> myPerf2
                     Sensitivity Specificity Pos Pred Value Neg Pred Value Precision
Class: setosa          0.9375000   1.0000000      1.0000000      0.9714286 1.0000000
Class: versicolor      0.9523810   0.9310345      0.9090909      0.9642857 0.9090909
Class: virginica       0.8461538   0.9459459      0.8461538      0.9459459 0.8461538
                        Recall        F1 Prevalence Detection Rate Detection Prevalence
Class: setosa       0.9375000 0.9677419       0.32           0.30                 0.30
Class: versicolor   0.9523810 0.9302326       0.42           0.40                 0.44
Class: virginica    0.8461538 0.8461538       0.26           0.22                 0.26
                    Balanced Accuracy
Class: setosa               0.9687500
Class: versicolor           0.9417077
Class: virginica            0.8960499
```

Il contenuto di questa sezione è molto semplice. Le funzioni di sensibilità e specificità della libreria *caret* mettono a confronto le etichette di classe previste e quelle originali.

La funzione `prediction()` del pacchetto *ROCR* (come vedremo più avanti) accetta due argomenti principali in termini di classi previste e di classi originali (in forma numerica) e crea un oggetto `prediction` con tutte le misure di performance. La misura di interesse viene estratta con la funzione `performance()`, che accetta l'oggetto `prediction` e il nome della misura come argomento di input. Possiamo usare il pacchetto *ROCR* solo se abbiamo classi binarie: per questo prendiamo solo i primi 34

```
> preds <- prediction(as.numeric(myPred[1:34]), as.numeric(myLabels[1:34]))
> performance(preds, "sens", "spec")
An object of class "performance"
Slot "x.name":
[1] "Specificity"
Slot "y.name":
[1] "Sensitivity"
Slot "alpha.name":
[1] "Cutoff"
Slot "x.values":
[[1]]
[1] 1.0000 0.9375 0.0000
Slot "y.values":
[[1]]
[1] 0 1 1
Slot "alpha.values":
[[1]]
[1] Inf   2   1
```

elementi (che contengono solo le classi setosa e versicolor), convertiamo la classe (categorica) in tipo numerico e poi usiamo la funzione `prediction()` del pacchetto *ROCR* come segue:

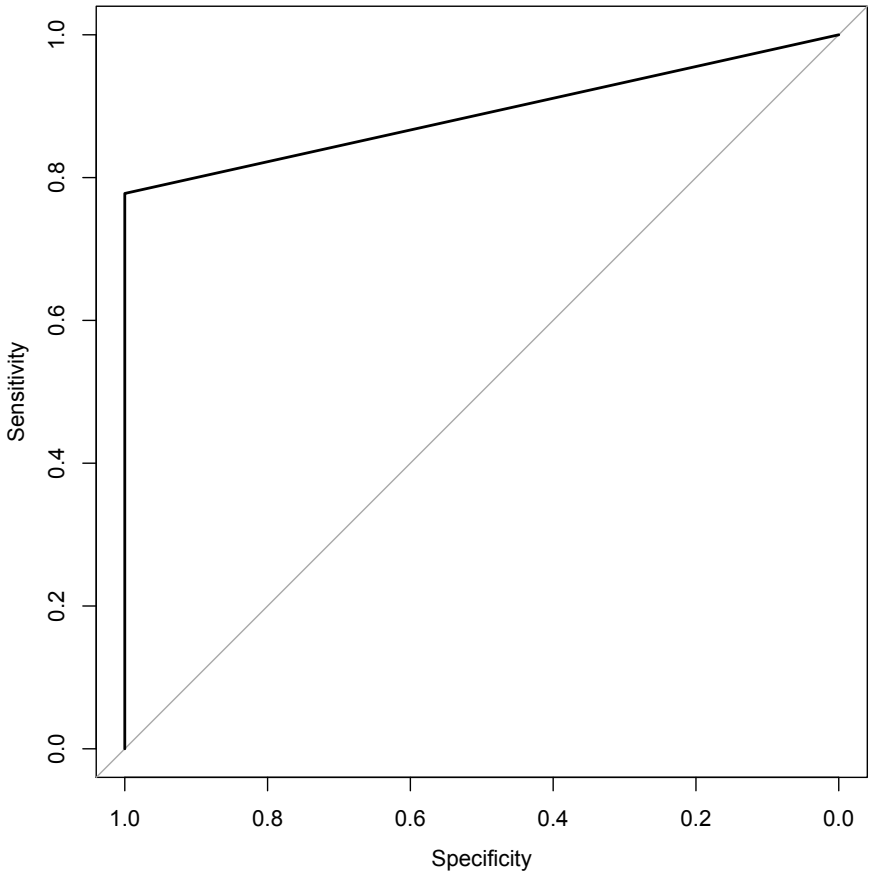
15.8 Visualizzazione di una curva ROC in R

La curva ROC fornisce la misura completa delle prestazioni; traccia un grafico della *specificità* rispetto a 1-*sensibilità*, rappresentate rispettivamente sugli assi *x* e *y*. Questo tipo di curva offre la visione completa della

```
> install.packages("ROCR")
> library(ROCR)
> library(pROC)
> library(MASS)
> load("cancer.rda")
> train_row <- sample(1:83, 60)
> train <- mldata[train_row,]
> test <- mldata[-train_row,]
> testClass <- test$tumor
> test$tumor <- NULL
> my_lda <- lda(tumor ~ ., train)
> testRes_lda <- predict(my_lda, test)
```

precisione (sensibilità e specificità) del classificatore in un unico grafico, nel quale si osserva l'area sotto la curva tracciata: più grande è l'area, migliori sono le prestazioni (maggiore è la sensibilità e la specificità).

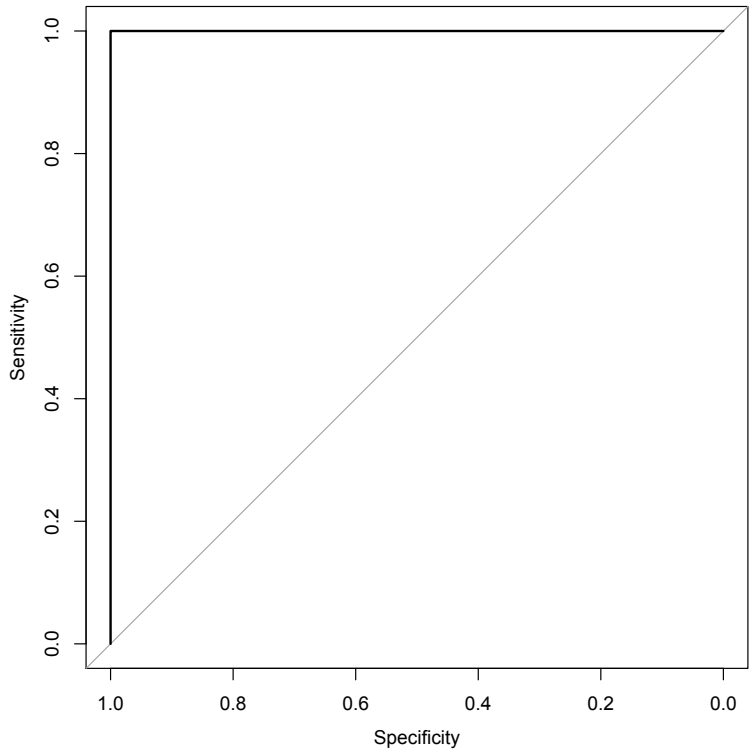
```
> library(e1071)
> mySVM <- svm(tumor ~ ., data = train)
> testRes_svm <- predict(mySVM, test)
> roc_svm <- plot.roc(as.numeric(testRes_svm), as.numeric(testClass))
```



Innanzitutto installiamo e carichiamo i pacchetti e i dati necessari e ricalcoliamo il classificatore LDA:

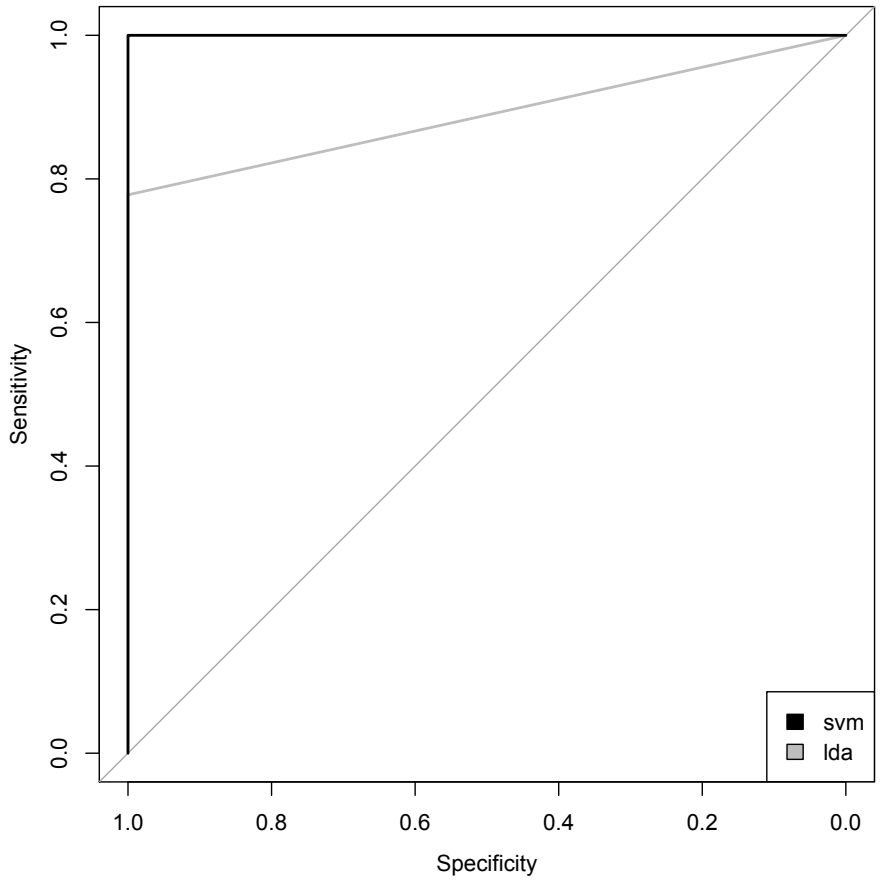
Quindi, calcoliamo l'oggetto di tipo *roc* per il classificatore LDA dei dati del tumore e tracciamo la curva:

A scopo di confronto, calcoliamo l'oggetto *roc* per un altro classificatore (SVM) come segue:



Mettiamo a confronto le curve ROC per i due metodi LDA e SVM:

```
> plot(roc_lda, col="grey")
> lines(roc_svm, col="black")
> legend("bottomright", c("svm", "lda"), fill = c("black","grey"))
```



Il pacchetto *pROC* ha anche la funzione `multiclass.roc()` che può trattare specificamente più classi; possiamo utilizzarla come segue:

```
> multiclass.roc(as.numeric(testRes_svm), as.numeric(testClass), percent=TRUE)

Call:
multiclass.roc.default(response = as.numeric(testRes_svm), predictor =
      as.numeric(testClass),     percent = TRUE)

Data: as.numeric(testClass) with 4 levels of as.numeric(testRes_svm): 1, 2, 3, 4.
Multi-class area under the curve: 92.42%
```

Il grafico precedente mostra una perfetta classificazione da parte del classificatore SVM. Questo è un caso ideale, con l'area sotto la curva (AUC, Area Under ROC Curve) pari a 1, e mostra una predizione accurata per ogni punto di dati (cosa che raramente accade nella realtà). La linea diagonale rappresenta il 50% dell'area sotto la curva (AUC) e non appartiene ad alcun classificatore (se non alla classica monetina…).

Come indicato nella sezione precedente, è possibile estrarre dall'oggetto di previsione le due misure di performance True Positive Rate (tpr) e False Positive Rate (fpr): questi parametri sono utilizzati per tracciare la curva ROC. La funzione `plot.roc()` del pacchetto *pROC* lo fa in un unico passaggio.

Il pacchetto *ROCR* funziona solo per la classificazione binaria. Nel caso di più classi possiamo usare il pacchetto *pROC*, che fornisce l'opzione di una curva ROC multiclasse. Esso presenta il valore medio AUC per tutte le classi. La funzione da usare è `multiclass.roc()`, come visto prima. Per ulteriori dettagli digitare il comando `?multiclass.roc`.

15.9 Identificazione di biomarcatori con dati da microarray

Fino ad ora abbiamo visto alcuni metodi di machine learning in R. È interessante esaminare la loro applicazione in bioinformatica. Nelle scienze della vita, confrontare gruppi di individui per trovare differenze significative sta diventando sempre più importante: può essere ad esempio una misura o una sostanza che indica la condizione biologica e può servire da biomarcatore.

Immaginiamo di avere dati da microarray sull'espressione genica di pazienti e controlli sani. I geni identificati sulla base di questi dati di espressione possono aiutare a distinguere tra campioni malati e sani, potendo servire come indicatori o biomarcatori per uno stato biologico come una malattia. Questa sezione mostra la ricerca di biomarcatori utilizzando alcune tecniche di apprendimento e metodi statistici.

Genereremo artificialmente il set di dati nel quale cercare il biomarcatore; inoltre utilizzeremo il pacchetto R *BioMark*.

Iniziamo con l'installazione e il caricamento del pacchetto:

```
> install.packages("BioMark")
> library(BioMark)
```

Una volta caricata la libreria, utilizziamo la funzione `gen.data()` per simulare un set di dati da utilizzare. Per impostazione predefinita vengono simulati i dati per cinque biomarcatori; questo può essere modificato attraverso l'argomento `nbiom` (numero di biomarcatori, cioè il numero di variabili da modificare nella classe di trattamento rispetto alla classe di controllo; le variabili che vengono modificate sono sempre le prime variabili della matrice dei dati):

```
> simdata <- gen.data(ncontrol=10, nbiom=5, nvar=500, nsimul=1, group.diff=1.5)
```

Controlliamo la composizione del dataset artificiale creato:

```
> str(simdata)
List of 3
 $ X          : num [1:20, 1:500, 1] 1.028 -0.11 0.72 -0.605 0.304 ...
 $ Y          : Factor w/ 2 levels "control","treated": 1 1 1 1 1 1 1 1 1 1 ...
 $ nbiomarkers: num 5
```

I biomarcatori reali nel dataset generato sono le prime cinque variabili; pertanto, è possibile assegnarli ad un oggetto R a fini di valutazione come segue:

```
> myrealMarkers <- c(1,2,3,4,5)
```

Utilizziamo ora la funzione `get.biom()` per cercare i biomarcatori nel dataset. La funzione restituisce un oggetto della classe "BMark", una lista contenente un elemento per ogni valore impostato per l'argomento `fmethod` (metodo di modellizzazione), così come un elemento `info`. I singoli elementi contengono informazioni a seconda del tipo scelto: per `type=="coef"` l'unico elemento restituito è una matrice contenente le dimensioni dei coefficienti; per `type=="HC"` e `type=="stab"` viene restituita una lista contenente gli elementi `biom.indices`, e gli elementi `pvals` (per `type=="HC"`) o `fraction.selected` (per `type=="stab"`). L'elemento `biom.indices` contiene gli indici delle variabili selezionate, e può essere estratto utilizzando la funzione `selection()`. L'elemento `pvals` contiene i *p*-value utilizzati nell'applicazione della soglia HC; questi sono presentati nell'ordine originale delle variabili, e possono essere ottenuti ad esempio direttamente tramite il *t*-test o il campionamento con permutazione. L'elemento `fraction.selected` indica in quale frazione delle iterazioni di selezione della stabilità è stata selezionata una particolare variabile. Più spesso è stata selezionata, più è stabile come biomarcatore. La funzione generica `coef.biom()` estrae i coefficienti del modello, i *p*-value o le frazioni di stabilità per i tipi "coef", "HC" e "stab" rispettivamente.

Utilizziamo quindi la funzione invocandola con il metodo `"pls"` e il tipo `"HC"` (*Higher Criticism approach*) come segue:

```
> myBiom <- get.biom(X=simdata$X[1:20,,1], Y=simdata$Y, fmethod="pls", type="HC")
```

Una volta selezionati i biomarcatori, diamo un'occhiata agli indici della variabile classificata:

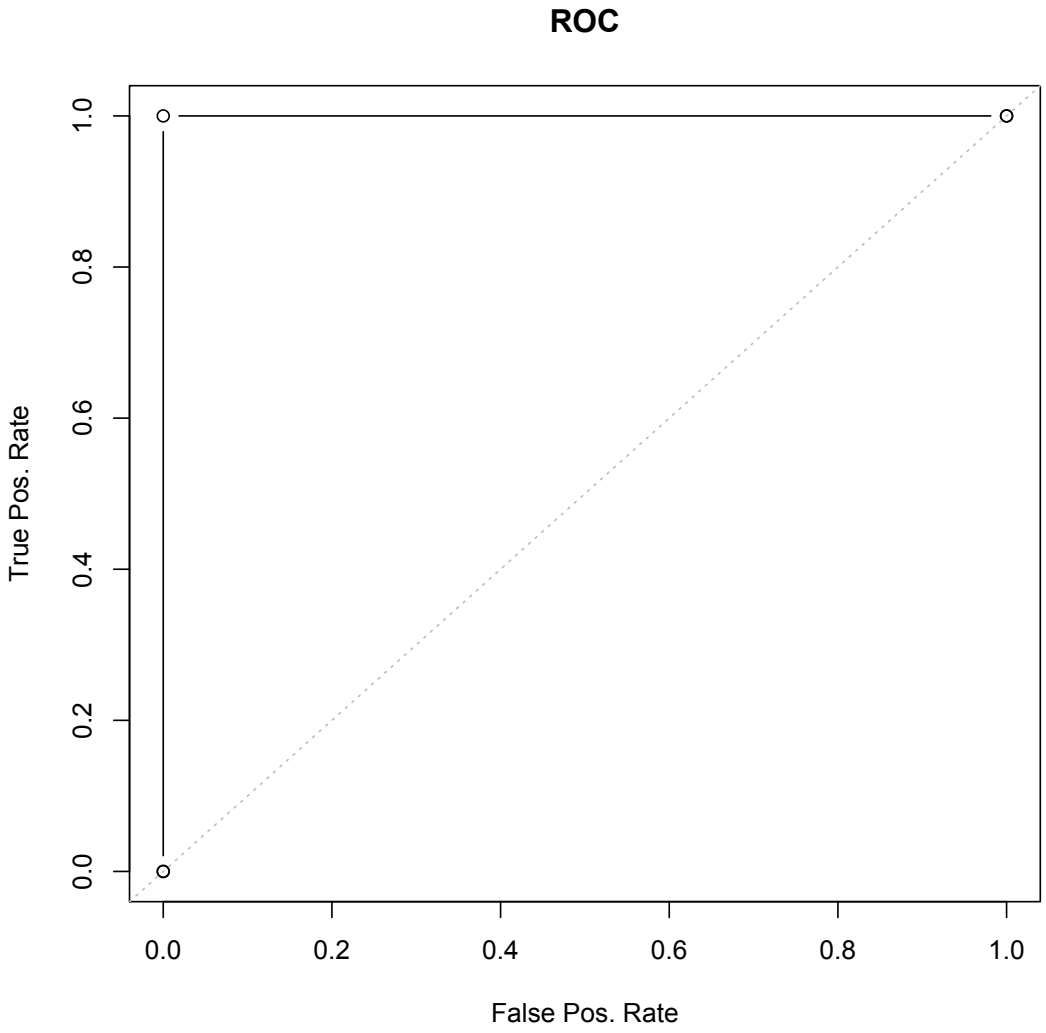
```
> selection(myBiom)
$pls
$pls[[1]]
[1]  52   4 372
```

La funzione `selection()` mostra gli indici della selezione in un oggetto di tipo *BioMark*. Osserviamo la struttura e il contenuto dell'oggetto *myBiom* creato dalla funzione `get.biom()`:

```
> myBiom
Result of HC-based biomarker selection using 1 modelling method.
> str(myBiom)
List of 2
 $ pls :List of 1
  ..$ :List of 2
  .. ..$ biom.indices: int [1:3] 52 4 372
  .. ..$ pvals       : num [1:500] 0.056 0.0513 0.0874 0.0012 0.0309 ...
 $ info:List of 4
  ..$ call   : language get.biom(X = simdata$X[1:20, , 1], Y = simdata$Y, fmethod =
      "pls", type = "HC")
  ..$ type   : chr "HC"
  ..$ fmethod: chr "pls"
  ..$ nvar   : int 500
 - attr(*, "class")= chr "BMark"
```

Una volta ottenuti i risultati, esaminiamo l'indicatore di performance in termini di curva ROC per i primi cinque biomarcatori selezionati tra i risultati:

```
> myROC <- ROC(1/coef(myBiom)$pls[[1]][1:5], myrealMarkers)
> plot(myROC)
```



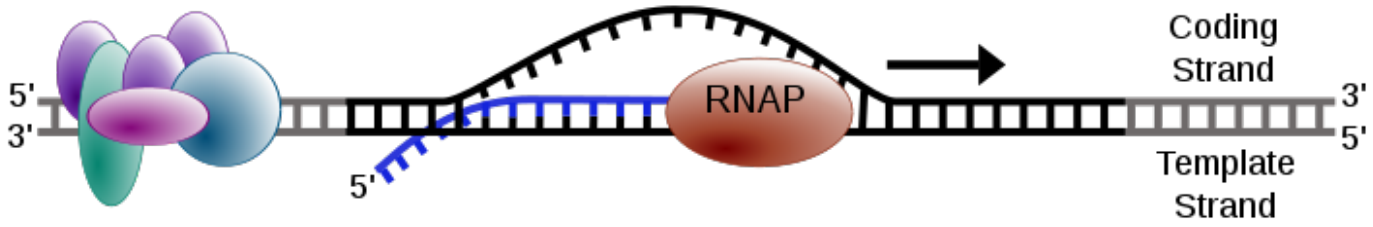
La libreria *BioMark* seleziona un biomarcatore da un set di dati sulla base di test statistici o di metodi basati sulla regressione e i loro coefficienti corrispondenti. La funzione `gen.data()` genera un dataset con una quantità definita di controlli (parametro `ncontrol`) e, per default, lo stesso numero di trattamenti (per modificarlo, utilizzare l'argomento `ntreated`). Questo genera un certo numero di matrici di dati, che abbiamo impostato a 1 tramite l'argomento `nsimul`. Il risultato è un elenco di etichette delle classi di dati e un numero di biomarcatori (il valore predefinito è 5). Per utilizzare il dataset, estraiamo i dati da utilizzare selezionando un solo dataset, specificando gli indici dell'array appropriato (l'argomento X). La funzione `get.biom()` esegue il test richiesto o l'apprendimento per filtrare l'insieme di biomarcatori nel dataset. È possibile utilizzare tre diversi metodi per dedurre i biomarcatori: oltre alla regressione parziale ai minimi quadrati (PLS, Partial Least Squares) che abbiamo utilizzato, gli altri metodi possibili sono la PCR e i *t*-test. Il valore restituito è un elenco ordinato di indici delle variabili (nel nostro caso, geni o proteine) individuati come biomarcatori nel dataset.

In questa sezione abbiamo utilizzato il pacchetto *BioMark*. Tuttavia, il processo può essere personalizzato per un metodo specifico, come SVM o altre tecniche: ciò però comporta una nuova implementazione, che qui non esaminiamo.

Appendice

La struttura a doppia elica del DNA

Il DNA ha una struttura *double-stranded* (a doppia elica, o filamento). Per convenzione, per un cromosoma di riferimento, un intero *strand* è designato come *forward strand* ("filamento in avanti") e l'altro come *reverse strand* ("filamento inverso"). Questa designazione è arbitraria. A volte si usano invece i termini "*plus strand*" e "*minus strand*".



Visivamente (non stiamo ancora parlando della trascrizione), di solito si legge la sequenza di uno *strand* nella direzione 5'—3'. Per il *forward strand* questo significa leggere da sinistra a destra, e per il *reverse strand* significa da destra a sinistra.

Un gene può essere posizionato su un filamento di DNA in uno dei due orientamenti. Si dice che il gene abbia uno *strand* di codifica (noto anche come *coding strand* o *sense strand*) e uno *strand* modello (noto anche come *template strand* o *antisense strand*). Per il 50% dei geni, lo *strand di* codifica corrisponde al *forward strand* del cromosoma, e per l'altro 50% corrisponde al *reverse strand*.

La sequenza mRNA (e le proteine) di un gene corrisponde alla sequenza del DNA letto (visivamente) dallo *strand* di codifica del gene. Così la sequenza mRNA corrisponde sempre alla sequenza di codifica 5'—3' di un gene.

La polimerasi dell'RNA si muove lungo il DNA nell'orientamento 5'—3' dello *strand* di codifica (ad esempio da sinistra a destra per un gene che si trova sul *forward strand*). Vengono lette le basi dal *template strand* (quindi nella direzione 3'–5') e viene man mano costruito l'mRNA. Ciò significa che l'mRNA corrisponde alla sequenza di codifica del gene, non alla sequenza del *template*.

Funzione R `cleanAlignment()`

```
cleanAlignment <- function(alignment, minpcnongap, minpcid) {

# Make a copy of the alignment to store the new alignment in:
newalignment <- alignment

# Find the number of sequences in the alignment:
numseqs <- alignment$nb

# Empty the alignment in "newalignment":
for (j in 1:numseqs) { newalignment$seq[[j]] <- "" }

# Find the length of the alignment:
alignmentlen <- nchar(alignment$seq[[1]])

# Look at each column of the alignment in turn:
for (i in 1:alignmentlen) {
	# see what percent of the letters in this column are non-gaps:
	nongap <- 0
	for (j in 1:numseqs) {
		seqj <- alignment$seq[[j]]
		letterij <- substr(seqj,i,i)
		if (letterij != "-") { nongap <- nongap + 1}
	}
	pcnongap <- (nongap*100)/numseqs
	# Only consider this column if at least minpcnongap % of the letters are not gaps:
	if (pcnongap >= minpcnongap) {
		# see what percent of the pairs of letters in this column are identical:
		numpairs <- 0; numid <- 0
		# find the letters in all of the sequences in this column:
		for (j in 1:(numseqs-1)) {
			seqj <- alignment$seq[[j]]
			letterij <- substr(seqj,i,i)
			for (k in (j+1):numseqs) {
				seqk <- alignment$seq[[k]]
				letterkj <- substr(seqk,i,i)
				if (letterij != "-" && letterkj != "-") {
					numpairs <- numpairs + 1
					if (letterij == letterkj) { numid <- numid + 1}
				}
			}
		}
		pcid <- (numid*100)/(numpairs)
		# Only consider this column if at least "minpcid" of the pairs of letters are identical:
		if (pcid >= minpcid) {
			for (j in 1:numseqs) {
				seqj <- alignment$seq[[j]]
				letterij <- substr(seqj,i,i)
				newalignmentj <- newalignment$seq[[j]]
				newalignmentj <- paste(newalignmentj,letterij,sep="")
				newalignment$seq[[j]] <- newalignmentj
			}
		}
	}
}

return(newalignment)
}
```

Funzione R `findcommunities()`

```
##
# Function to find network communities in a graph
##

findcommunities <- function(mygraph,minsize) {

# Load up the igraph library:
require("igraph")

# Set the counter for the number of communities:
cnt <- 0

# First find the connected components in the graph:
myconnectedcomponents <- connectedComp(mygraph)

# For each connected component, find the communities within that connected component:
numconnectedcomponents <- length(myconnectedcomponents)
for (i in 1:numconnectedcomponents) {
      component <- myconnectedcomponents[[i]]
      # Find the number of nodes in this connected component:
      numnodes <- length(component)
      if (numnodes > 1) { # We can only find communities if there is more than one node
            mysubgraph <- subGraph(component, mygraph)
            # Find the communities within this connected component:
            print(component)
            myvector <- vector()
            mylist <- findcommunities2(mysubgraph,cnt,"FALSE",myvector,minsize)
            cnt <- mylist[[1]]
            myvector <- mylist[[2]]
      }
}

print(paste("There were",cnt,"communities in the input graph"))

}
```

Funzione R `findcommunities2()`

```
##
# Function to find network communities in a connected component of a graph
##

findcommunities2 <- function(mygraph,cnt,plot,myvector,minsize) {

# Find the number of nodes in the input graph
nodes <- nodes(mygraph)
numnodes <- length(nodes)

# Record the vertex number for each vertex name
myvector <- vector()
for (i in 1:numnodes) {
      node <- nodes[i] # "node" is the vertex name, "i" is the vertex number
      myvector[`node`] <- i # Add named element to myvector
}

# Create a graph in the "igraph" library format, with numnodes nodes:
newgraph <- graph.empty(n=numnodes,directed=FALSE)

# First record which edges we have seen already in the "mymatrix" matrix, so that we don't add any edge twice:
mymatrix <- matrix(nrow=numnodes,ncol=numnodes)
for (i in 1:numnodes) {
      for (j in 1:numnodes) {
          mymatrix[i,j] = 0
          mymatrix[j,i] = 0
    }
}

# Now add edges to the graph "newgraph":
for (i in 1:numnodes) {
      node <- nodes[i] # "node" is the vertex name, i is the vertex number
      # Find the nodes that this node is joined to:
      neighbours <- adj(mygraph, node)
      neighbours <- neighbours[[1]] # Get the list of neighbours
      numneighbours <- length(neighbours)
      if (numneighbours >= 1) { # If this node "node" has some edges to other nodes
            for (j in 1:numneighbours) {
                  neighbour <- neighbours[j]
                  # Get the vertex number
                  neighbourindex <- myvector[neighbour]
                  neighbourindex <- neighbourindex[[1]]
                  # Add a node in the new graph "newgraph" between vertices "i" and "neighbourindex"
                  # In graph "newgraph", the vertices are counted from 0 upwards:
                  indexi <- i
                  indexj <- neighbourindex
                  # If we have not seen this edge already:
                  if (mymatrix[indexi,indexj] == 0 && mymatrix[indexj,indexi] == 0) {
                        mymatrix[indexi,indexj] <- 1
                        mymatrix[indexj,indexi] <- 1
                        # Add edges to the graph "newgraph"
                        newgraph <- add.edges(newgraph, c(i, neighbourindex))
                  }
            }
      }
}
```

```
# Set the names of the vertices in graph "newgraph":
newgraph <- set.vertex.attribute(newgraph, "name", value=nodes)

# Now find communities in the graph:
communities <- spinglass.community(newgraph)

# Find how many communities there are:
sizecommunities <- communities$csize
numcommunities <- length(sizecommunities)

# Find which vertices belong to which communities:
membership <- communities$membership

# Get the names of vertices in the graph "newgraph":
vertexnames <- get.vertex.attribute(newgraph, "name")

# Print out the vertices belonging to each community:
for (i in 1:numcommunities) {
      cnt <- cnt + 1
      nummembers <- 0
      printout <- paste("Community",cnt,":")
      for (j in 1:length(membership)) {
            community <- membership[j]
            if (community == i) {  # If vertex j belongs to the i-th community
                  vertexname <- vertexnames[j]
                  if (plot == FALSE) {
                        nummembers <- nummembers + 1
                        # Print out the vertices belonging to the community
                        printout <- paste(printout,vertexname)
                  }
                  else {
                        # Colour in the vertices belonging to the community
                        myvector[`vertexname`] <- cnt
                  }
            }
      }
      if (plot == FALSE && nummembers >= minsize) {
            print(printout)
      }
}

return(list(cnt,myvector))

}
```

Funzione R `findcomponent()`

```
##
# Function to find the connected component that contains a particular vertex
##

findcomponent <- function(graph,vertex) {

require("RBGL")

found <- 0
myconnectedcomponents <- connectedComp(graph)
numconnectedcomponents <- length(myconnectedcomponents)

for (i in 1:numconnectedcomponents) {
      componenti <- myconnectedcomponents[[i]]
      numvertices <- length(componenti)
      for (j in 1:numvertices) {
            vertexj <- componenti[j]
            if (vertexj == vertex) {
                  found <- 1
                  return(componenti)
            }
      }
}

print("ERROR: did not find vertex in the graph")

}
```

Funzione R `findORFsinSeq()`

```
findORFsinSeq <- function(sequence) {

require(Biostrings)

# Make vectors "positions" and "types" containing information on the positions of ATGs in the sequence:
mylist <- findPotentialStartsAndStops(sequence)
positions <- mylist[[1]]
types <- mylist[[2]]

# Make vectors "orfstarts" and "orfstops" to store the predicted start and stop codons of ORFs
orfstarts <- numeric()
orfstops <- numeric()
# Make a vector "orflengths" to store the lengths of the ORFs
orflengths <- numeric()
# Print out the positions of ORFs in the sequence:
numpositions <- length(positions) # find the length of vector "positions"

# There must be at least one start codon and one stop codon to have an ORF.
if (numpositions >= 2) {
      for (i in 1:(numpositions-1)) {
            posi <- positions[i]
            typei <- types[i]
            found <- 0
            while (found == 0) {
                  for (j in (i+1):numpositions) {
                        posj <- positions[j]
                        typej <- types[j]
                        posdiff <- posj - posi
                        posdiffmod3 <- posdiff %% 3
                        # Add in the length of the stop codon
                        orflength <- posj - posi + 3
                        if (typei == "ATG" && (typej == "TAA" || typej == "TAG" ||
typej == "TGA") && posdiffmod3 == 0) {
                              # Check if we have already used the stop codon at posj+2 in an ORF
                              numorfs <- length(orfstops)
                              usedstop <- -1
                              if (numorfs > 0) {
                                    for (k in 1:numorfs) {
                                          orfstopk <- orfstops[k]
                                          if (orfstopk == (posj + 2)) { usedstop <- 1 }
                                    }
                              }
                              if (usedstop == -1) {
                                    orfstarts <- append(orfstarts, posi,
after=length(orfstarts))
                                    orfstops <- append(orfstops, posj+2,
after=length(orfstops)) # including the stop codon
                                    orflengths <- append(orflengths, orflength,
after=length(orflengths))
                              }
                              found <- 1
                              break
                        }
                        if (j == numpositions) { found <- 1 }
                  } # end for (j in (i+1):numpositions)
            } # end while (found == 0)
      } # end for (i in 1:(numpositions-1))
```

```
} # end if (numpositions >= 2)

# Sort the final ORFs by start position:
indices <- order(orfstarts)
orfstarts <- orfstarts[indices]
orfstops <- orfstops[indices]

# Find the lengths of the ORFs that we have
orflengths <- numeric()
numorfs <- length(orfstarts)
for (i in 1:numorfs) {
      orfstart <- orfstarts[i]
      orfstop <- orfstops[i]
      orflength <- orfstop - orfstart + 1
      orflengths <- append(orflengths,orflength,after=length(orflengths))
} # end for (i in 1:numorfs)

mylist <- list(orfstarts, orfstops, orflengths)
return(mylist)

}
```

Funzione R `findPotentialStartsAndStops()`

```
findPotentialStartsAndStops <- function(sequence) {

# Define a vector with the sequences of potential start and stop codons
codons <- c("ATG", "TAA", "TAG", "TGA")
sequence = toupper(sequence)

# Find the number of occurrences of each type of potential start or stop codon
for (i in 1:4) {
	codon <- codons[i]
	# Find all occurrences of codon "codon" in sequence "sequence"
	occurrences <- matchPattern(codon, sequence)
	# Find the start positions of all occurrences of "codon" in sequence "sequence"
	codonpositions <- attr(attr(occurrences,"ranges"),"start")
	# Find the total number of potential start and stop codons in sequence "sequence"
	numoccurrences <- length(codonpositions)
	if (i == 1) {
		# Make a copy of vector "codonpositions" called "positions"
		positions <- codonpositions
		# Make a vector "types" containing "numoccurrences" copies of "codon"
		types <- rep(codon, numoccurrences)
	}
	else {
		positions <- append(positions, codonpositions, after=length(positions))
		# Add the vector "rep(codon, numoccurrences)" to the end of vector "types":
		types <- append(types, rep(codon, numoccurrences), after=length(types))
	}
}

# Sort the vectors "positions" and "types" in order of position along the input sequence:
indices <- order(positions)
positions <- positions[indices]
types <- types[indices]

# Return a list variable including vectors "positions" and "types":
mylist <- list(positions,types)
return(mylist)
}
```

Funzione R `generatehmmseq()`

```
##
# Function to generate a DNA sequence, given a HMM and the length of the sequence to be generated.
##

generatehmmseq <- function(transitionmatrix, emissionmatrix, initialprobs, seqlength) {

nucleotides <- c("A", "C", "G", "T") # Define the alphabet of nucleotides
states <- rownames(emissionmatrix) # Define the names of the states
mysequence <- character() # Create a vector for storing the new sequence
mystates <- character() # Create a vector for storing the state that each position in the new sequence was generated by

# Choose the state for the first position in the sequence:
firststate <- sample(states, 1, rep=TRUE, prob=initialprobs)

# Get the probabilities of the current nucleotide, given that we are in the state "firststate":
probabilities <- emissionmatrix[firststate,]

# Choose the nucleotide for the first position in the sequence:
firstnucleotide <- sample(nucleotides, 1, rep=TRUE, prob=probabilities)
mysequence[1] <- firstnucleotide # Store the nucleotide for the first position of the sequence
mystates[1] <- firststate # Store the state that the first position in the sequence was generated by

for (i in 2:seqlength) {
      prevstate <- mystates[i-1] # Get the state that the previous nucleotide in the sequence was generated by
      # Get the probabilities of the current state, given that the previous nucleotide was generated by state "prevstate"
      stateprobs <- transitionmatrix[prevstate,]
      # Choose the state for the ith position in the sequence:
      state <- sample(states, 1, rep=TRUE, prob=stateprobs)
      # Get the probabilities of the current nucleotide, given that we are in the state "state":
      probabilities <- emissionmatrix[state,]
      # Choose the nucleotide for the ith position in the sequence:
      nucleotide <- sample(nucleotides, 1, rep=TRUE, prob=probabilities)
      mysequence[i] <- nucleotide # Store the nucleotide for the current position of the sequence
      mystates[i] <- state # Store the state that the current position in the sequence was generated by
}

for (i in 1:length(mysequence)) {
      nucleotide <- mysequence[i]
      state <- mystates[i]
      print(paste("Position", i, ", State", state, ", Nucleotide = ", nucleotide))
}

}
```

Funzione R `generatemarkovseq()`

```
generatemarkovseq <- function(transitionmatrix, initialprobs, seqlength) {

nucleotides <- c("A", "C", "G", "T") # Define the alphabet of nucleotides
mysequence <- character() # Create a vector for storing the new sequence

# Choose the nucleotide for the first position in the sequence:
firstnucleotide <- sample(nucleotides, 1, rep=TRUE, prob=initialprobs)
mysequence[1] <- firstnucleotide # Store the nucleotide for the first position of the sequence
for (i in 2:seqlength) {
      prevnucleotide <- mysequence[i-1] # Get the previous nucleotide in the new sequence
      # Get the probabilities of the current nucleotide, given previous nucleotide "prevnucleotide":
      probabilities <- transitionmatrix[prevnucleotide,]
      # Choose the nucleotide at the current position of the sequence:
      nucleotide <- sample(nucleotides, 1, rep=TRUE, prob=probabilities)
      mysequence[i] <- nucleotide # Store the nucleotide for the current position of the sequence
}

return(mysequence)

}
```

Funzione R `generateSeqsWithMultinomialModel()`

```
generateSeqsWithMultinomialModel <- function(inputsequence, X) {

# Change the input sequence into a vector of letters
require("seqinr") # This function requires the SeqinR package
inputsequencevector <- s2c(inputsequence)

# Find the frequencies of the letters in the input sequence "inputsequencevector":
mylength <- length(inputsequencevector)
mytable <- table(inputsequencevector)

# Find the names of the letters in the sequence
letters <- rownames(mytable)
numletters <- length(letters)
probabilities <- numeric() # Make a vector to store the probabilities of letters
for (i in 1:numletters) {
      letter <- letters[i]
      count <- mytable[[i]]
      probabilities[i] <- count/mylength
}

# Make X random sequences using the multinomial model with probabilities "probabilities"
seqs <- numeric(X)
for (j in 1:X) {
      seq <- sample(letters, mylength, rep=TRUE, prob=probabilities) # Sample with replacement
      seq <- c2s(seq)
      seqs[j] <- seq
}

# Return the vector of random sequences
return(seqs)
}
```

Funzione R `getncbiseq()`

```
##
# To retrieve a sequence with a particular NCBI accession using SeqinR
##

getncbiseq <- function(accession) {

require("seqinr") # this function requires the SeqinR R package

# First find which ACNUC database the accession is stored in:
dbs <- c("swissprot","genbank","refseq","refseqViruses","bacterial")
numdbs <- length(dbs)

for (i in 1:numdbs) {
      db <- dbs[i]
      choosebank(db,timeout=20)
      # check if the sequence is in ACNUC database 'db':
      resquery <- try(query(".tmpquery", paste("AC=", accession)), silent = TRUE)
      if (!(inherits(resquery, "try-error"))) {
            queryname <- "query2"
            thequery <- paste("AC=",accession,sep="")
            query2 <- query(`queryname`,`thequery`)
            # see if a sequence was retrieved:
            seq <- getSequence(query2$req[[1]])
            closebank()
            return(seq)
      }
    closebank()
}

print(paste("ERROR: accession",accession,"was not found")) }
```

Funzione R `makeproteingraph()`

```
##
# Function to make a graph based on protein-protein interaction data in an input file
##

makeproteingraph <- function(myfile) {

require("graph")
mytable <- read.table(file(myfile)) # Store the data in a data frame
proteins1 <- mytable$V1
proteins2 <- mytable$V2
protnames <- c(levels(factor(proteins1)),levels(factor(proteins2)))

# Find out how many pairs of proteins there are:
numpairs <- length(proteins1)

# Find the unique protein names:
uniquenames <- unique(protnames)

# Make a graph for these proteins with no edges:
mygraph <- new("graphNEL", nodes = uniquenames)

# Add edges to the graph (see http://rss.acs.unt.edu/Rdoc/library/graph/doc/graph.pdf for more examples):
weights <- rep(1,numpairs)
mygraph2 <- addEdge(as.vector(proteins1),as.vector(proteins2),mygraph,weights)

return(mygraph2)

}
```

Funzione R `makerandomgraph()`

```
##
# Function to make a random graph
##

makerandomgraph <- function(numvertices,numedges) {

require("graph")

# Make a vector with the names of the vertices
mynames <- sapply(seq(1,numvertices),toString)
myrandomgraph <- randomEGraph(mynames, edges = numedges)

return(myrandomgraph)

}
```

Funzione R `makeViterbimat()`

```
##
# This makes the matrix v using the Viterbi algorithm.
# Adapted from "Applied Statistics for Bioinformatics using R" by Wim P. Krijnen, page 209
# ( cran.r-project.org/doc/contrib/Krijnen-IntroBioInfStatistics.pdf )
##

makeViterbimat <- function(sequence, transitionmatrix, emissionmatrix) {

# Change the sequence to uppercase
sequence <- toupper(sequence)

# Find out how many states are in the HMM
numstates <- dim(transitionmatrix)[1]

# Make a matrix with as many rows as positions in the sequence, and as many columns as states in the HMM
v <- matrix(NA, nrow = length(sequence), ncol = dim(transitionmatrix)[1])

# Set the values in the first row of matrix v (representing the first position of the sequence) to 0
v[1, ] <- 0

# Set the value in the first row of matrix v, first column to 1
v[1,1] <- 1

# Fill in the matrix v:
for (i in 2:length(sequence)) {  # For each position in the DNA sequence:
      for (l in 1:numstates) {  # For each of the states of in the HMM:
            # Find the probabilility, if we are in state "l", of choosing the nucleotide at position "i" in the sequence
            statelprobnucleotidei <- emissionmatrix[l,sequence[i]]

            # v[(i-1),] gives the values of v for the (i-1)th row of v, ie. the (i-1)th position in the sequence.
            # In v[(i-1),] there are values of v at the (i-1)th row of the sequence for each possible state k.
            # v[(i-1),k] gives the value of v at the (i-1)th row of the sequence for a particular state k.

            # transitionmatrix[l,] gives the values in the lth row of the transition matrix, xx should not be transitionmatrix[,l]?
            # probabilities of changing from a previous state k to a current state l.

            # max(v[(i-1),] * transitionmatrix[l,]) is the maximum probability for the nucleotide observed
            # at the previous position in the sequence in state k, followed by a transition from previous
            # state k to current state l at the current nucleotide position.

            # Set the value in matrix v for row i (nucleotide position i), column l (state l) to be:
            v[i,l] <- statelprobnucleotidei * max(v[(i-1),] * transitionmatrix[,l])
      }
}

return(v)

}
```

Funzione R `plotcommunities()`

```
##
# Function to plot network communities in a graph
##

plotcommunities <- function(mygraph) {

# Load the "igraph" package:
require("igraph")

# Make a plot of the graph
graphplot <- layoutGraph(mygraph, layoutType="neato")
renderGraph(graphplot)

# Get the names of the nodes in the graph:
vertices <- nodes(mygraph)
numvertices <- length(vertices)

# Now record the colour of each vertex in a vector "myvector":
myvector <- vector()
colour <- "red"

for (i in 1:numvertices) {
      vertex <- vertices[i]
      myvector[`vertex`] <- colour # Add named element to myvector
}

# Set the counter for the number of communities:
cnt <- 0

# First find the connected components in the graph:
myconnectedcomponents <- connectedComp(mygraph)

# For each connected component, find the communities within that connected component:
numconnectedcomponents <- length(myconnectedcomponents)
for (i in 1:numconnectedcomponents) {
      component <- myconnectedcomponents[[i]]
      # Find the number of nodes in this connected component:
      numnodes <- length(component)
      if (numnodes > 1) { # We can only find communities if there is more than one node
            mysubgraph <- subGraph(component, mygraph)
            # Find the communities within this connected component:
            mylist <- findcommunities2(mysubgraph,cnt,"TRUE",myvector,0)
            cnt <- mylist[[1]]
            myvector <- mylist[[2]]
      }
}

# Get a set of cnt colours, where cnt is equal to the number of communities found:
mycolours <- rainbow(cnt)

# Set the colour of the vertices, so that vertices in each community are of same colour,
# and vertices in different communities are different colours:
myvector2 <- vector()
for (i in 1:numvertices) {
      vertex <- vertices[i]
      community <- myvector[vertex]
```

```
    mycolour <- mycolours[community]
    myvector2[`vertex`] <- mycolour
}

nodeRenderInfo(graphplot) = list(fill=myvector2)
renderGraph(graphplot)

}
```

Funzione R `plotORFsinSeq()`

```
plotORFsinSeq <- function(sequence) {

# Make vectors "positions" and "types" containing information on the positions of ATGs in the sequence:
mylist <- findPotentialStartsAndStops(sequence)
positions <- mylist[[1]]
types <- mylist[[2]]

# Make vectors "orfstarts" and "orfstops" to store the predicted start and stop codons of ORFs
orfstarts <- numeric()
orfstops <- numeric()

# Make a vector "orflengths" to store the lengths of the ORFs
orflengths <- numeric()

# Print out the positions of ORFs in the sequence:
numpositions <- length(positions) # find the length of vector "positions"

# There must be at least one start codon and one stop codon to have an ORF.
if (numpositions >= 2) {
      for (i in 1:(numpositions-1)) {
            posi <- positions[i]
            typei <- types[i]
            found <- 0
            while (found == 0) {
                  for (j in (i+1):numpositions) {
                        posj <- positions[j]
                        typej <- types[j]
                        posdiff <- posj - posi
                        posdiffmod3 <- posdiff %% 3
                        orflength <- posj - posi + 3 # add in the length of the stop codon
                        if (typei == "ATG" && (typej == "TAA" || typej == "TAG" ||
typej == "TGA") && posdiffmod3 == 0) {
                        # Check if we have already used the stop codon at posj+2 in an ORF
                        numorfs <- length(orfstops)
                        usedstop <- -1
                        if (numorfs > 0) {
                              for (k in 1:numorfs) {
                                    orfstopk <- orfstops[k]
                                    if (orfstopk == (posj + 2)) { usedstop <- 1 } }
                              }
                              if (usedstop == -1) {
                                    orfstarts <- append(orfstarts, posi,
after=length(orfstarts))
                                    orfstops <- append(orfstops, posj+2,
after=length(orfstops)) # including the stop codon
                                    orflengths <- append(orflengths, orflength,
after=length(orflengths))
                              }
                              found <- 1
                              break
                        }
                        if (j == numpositions) { found <- 1 }
                  }
            }
      }
}
```

```
# Sort the final ORFs by start position:
indices <- order(orfstarts)
orfstarts <- orfstarts[indices]
orfstops <- orfstops[indices]

# Make a plot showing the positions of ORFs in the input sequence:
# Draw a line at y=0 from 1 to the length of the sequence:
x <- c(1,nchar(sequence))
y <- c(0,0)
plot(x, y, ylim=c(0,3), type="l", axes=FALSE, xlab="Nucleotide", ylab="Reading frame",
main="Predicted ORFs")
segments(1,1,nchar(sequence),1)
segments(1,2,nchar(sequence),2)

# Add the x-axis at y=0:
axis(1, pos=0)

# Add the y-axis labels:
text(0.9,0.5,"+1")
text(0.9,1.5,"+2")
text(0.9,2.5,"+3")

# Make a plot of the ORFs in the sequence:
numorfs <- length(orfstarts)
for (i in 1:numorfs) {
      orfstart <- orfstarts[i]
      orfstop <- orfstops[i]
      remainder <- (orfstart-1) %% 3
      if (remainder == 0) { # +1 reading frame
            rect(orfstart,0,orfstop,1,col="cyan",border="black")
      }
      else if (remainder == 1) { # +2 reading frame
            rect(orfstart,1,orfstop,2,col="cyan",border="black")
      }
      else if (remainder == 2) { # +3 reading frame
            rect(orfstart,2,orfstop,3,col="cyan",border="black")
      }
}

}
```

Funzione R `plotPotentialStartsAndStops()`

```
plotPotentialStartsAndStops <- function(sequence) {

# Define a vector with the sequences of potential start and stop codons
codons <- c("ATG", "TAA", "TAG", "TGA")

# Find the number of occurrences of each type of potential start or stop codon
for (i in 1:4) {
      codon <- codons[i]
      # Find all occurrences of codon "codon" in sequence "sequence"
      occurrences <- matchPattern(codon, sequence)
      # Find the start positions of all occurrences of "codon" in sequence "sequence"
      codonpositions <- attr(attr(occurrences,"range"),"start")
      # Find the total number of potential start and stop codons in sequence "sequence"
      numoccurrences <- length(codonpositions)
      if (i == 1) {
            # Make a copy of vector "codonpositions" called "positions"
            positions <- codonpositions
            # Make a vector "types" containing "numoccurrences" copies of "codon"
            types <- rep(codon, numoccurrences)
      }
      else {
            positions <- append(positions, codonpositions, after=length(positions))
            # Add the vector "rep(codon, numoccurrences)" to the end of vector "types":
            types <- append(types, rep(codon, numoccurrences), after=length(types))
      }
}

# Sort the vectors "positions" and "types" in order of position along the input sequence:
indices <- order(positions)
positions <- positions[indices]
types <- types[indices]

# Make a plot showing the positions of the start and stop codons in the input sequence:
# Draw a line at y=0 from 1 to the length of the sequence:
x <- c(1,nchar(sequence))
y <- c(0,0)
plot(x, y, ylim=c(0,3), type="l", axes=FALSE, xlab="Nucleotide", ylab="Reading frame",
main="Predicted start (red) and stop (blue) codons")
segments(1,1,nchar(sequence),1)
segments(1,2,nchar(sequence),2)

# Add the x-axis at y=0:
axis(1, pos=0)

# Add the y-axis labels:
text(0.9,0.5,"+1")
text(0.9,1.5,"+2")
text(0.9,2.5,"+3")

# Draw in each predicted start/stop codon:
numcodons <- length(positions)
for (i in 1:numcodons) {
      position <- positions[i]
      type <- types[i]
      remainder <- (position-1) %% 3
      if (remainder == 0) { # reading frame +1
```

```
            if (type == "ATG") { segments(position,0,position,1,lwd=1,col="red") }
            else { segments(position,0,position,1,lwd=1,col="blue")}
      }
      else if (remainder == 1) { # reading frame +2
            if (type == "ATG") { segments(position,1,position,2,lwd=1,col="red") }
            else { segments(position,1,position,2,lwd=1,col="blue")}
      }
      else if (remainder == 2) { # reading frame +3
            if (type == "ATG") { segments(position,2,position,3,lwd=1,col="red") }
            else { segments(position,2,position,3,lwd=1,col="blue")}
      }
}

}
```

Funzione R `printMultipleAlignment()`

```
printMultipleAlignment <- function(alignment, chunksize=60) {

# This function requires the Biostrings package:
require("Biostrings")

# Find the number of sequences in the alignment :
numseqs <- alignment$nb

# Find the length of the alignment:
alignmentlen <- nchar(alignment$seq[[1]])
starts <- seq(1, alignmentlen, by=chunksize)
n <- length(starts)

# Get the alignment for each of the sequences:
aln <- vector()
lettersprinted <- vector()
for (j in 1:numseqs) {
      alignmentj <- alignment$seq[[j]]
      aln[j] <- alignmentj
      lettersprinted[j] <- 0
}

# Print out the alignment in blocks of 'chunksize' columns:
for (i in 1:n) { # for each of n chunks
      for (j in 1:numseqs) {
            alnj <- aln[j]
            chunkseqjaln <- substring(alnj, starts[i], starts[i]+chunksize-1)
            chunkseqjaln <- toupper(chunkseqjaln)
            # Find out how many gaps there are in chunkseqjaln:
            gapsj <- countPattern("-",chunkseqjaln) # countPattern() is from Biostrings package
            # Calculate how many residues of the first sequence we have printed so far in the alignment:
            lettersprinted[j] <- lettersprinted[j] + chunksize - gapsj
            print(paste(chunkseqjaln,lettersprinted[j]))
      }
      print(paste(' '))
}

}
```

Funzione R `printPairwiseAlignment()`

```
printPairwiseAlignment <- function(alignment, chunksize=60, returnlist=FALSE) {

require(Biostrings) # This function requires the Biostrings package
seq1aln <- pattern(alignment) # Get the alignment for the first sequence
seq2aln <- subject(alignment) # Get the alignment for the second sequence
alnlen <- nchar(seq1aln) # Find the number of columns in the alignment
starts <- seq(1, alnlen, by=chunksize)

n <- length(starts)
seq1alnresidues <- 0
seq2alnresidues <- 0
for (i in 1:n) {
      chunkseq1aln <- substring(seq1aln, starts[i], starts[i]+chunksize-1)
      chunkseq2aln <- substring(seq2aln, starts[i], starts[i]+chunksize-1)
      # Find out how many gaps there are in chunkseq1aln:
      gaps1 <- countPattern("-",chunkseq1aln) # countPattern() is from Biostrings package
      # Find out how many gaps there are in chunkseq2aln:
      gaps2 <- countPattern("-",chunkseq2aln) # countPattern() is from Biostrings package
      # Calculate how many residues of the first sequence we have printed so far in the alignment:
      seq1alnresidues <- seq1alnresidues + chunksize - gaps1
      # Calculate how many residues of the second sequence we have printed so far in the alignment:
      seq2alnresidues <- seq2alnresidues + chunksize - gaps2
      if (returnlist == 'FALSE') {
            print(paste(chunkseq1aln,min(nchar(seq1aln),seq1alnresidues)))
            print(paste(chunkseq2aln,min(nchar(seq2aln),seq2alnresidues)))
            print(paste(' '))
      }
}

if (returnlist == 'TRUE') {
      vector1 <- s2c(substring(seq1aln, 1, nchar(seq1aln)))
      vector2 <- s2c(substring(seq2aln, 1, nchar(seq2aln)))
      mylist <- list(vector1, vector2)
      return(mylist)
}

}
```

Funzione R `retrieventrezseqs()`

```
##
# Function to retrieve sequences with names passed in parameter `seqnames` from the specified database `dbname`
##

retrieventrezseqs <- function(seqnames,dbname) {

require("rentrez") # This function requires the Entrez package
require("seqinr") # This function requires the SeqinR package

myseqs <- list() # Make a list to store the sequences
for (i in 1:length(seqnames)) {
      seqname <- seqnames[i]
      print(paste("Retrieving sequence",seqname,"..."))
      tmp_search <- entrez_search(db=dbname, term=seqname)
      if (tmp_search$count==0) {
            seq <- NULL
            print(paste("   ... no sequence found for",seqname,"!"))
      }
      else {
            tmp_id <- tmp_search$ids[1]
            tmp_fasta <- entrez_fetch(db=dbname, id=tmp_id, rettype="fasta")
            tmp_file <- tempfile()
            write(tmp_fasta, file=tmp_file)
            tmp_seq <- seqinr::read.fasta(file = tmp_file, as.string = FALSE)
            seq = tmp_seq[[1]][1:length(tmp_seq[[1]])]
            print(paste("   ... found sequence for",seqname,"with ID",tmp_id))
      }
      myseqs[[i]] <- toupper(seq)
}

return(myseqs)
}
```

Funzione R `rootedNJtree()`

```
rootedNJtree <- function(alignment, theoutgroup, type) {

# Load the ape and seqinR packages:
require("ape")
require("seqinr")

# Define a function for making a tree:
makemytree <- function(alignmentmat, outgroup=`theoutgroup`) {
      alignment <- ape::as.alignment(alignmentmat)
      if (type == "protein") {
            mydist <- dist.alignment(alignment)
      }
      else if (type == "DNA") {
            alignmentbin <- as.DNAbin(alignment)
            mydist <- dist.dna(alignmentbin)
      }
      mytree <- nj(mydist)
      mytree <- makeLabel(mytree, space="") # get rid of spaces in tip names
      myrootedtree <- root(mytree, outgroup, r=TRUE)
      return(myrootedtree)
}

# Infer a tree:
mymat <- as.matrix.alignment(alignment)
myrootedtree <- makemytree(mymat, outgroup=theoutgroup)

# Bootstrap the tree:
myboot <- boot.phylo(myrootedtree, mymat, makemytree)

# Plot the tree:
plot.phylo(myrootedtree, type="p") # plot the rooted phylogenetic tree
nodelabels(myboot,cex=0.7) # plot the bootstrap values
myrootedtree$node.label <- myboot # make the bootstrap values be the node labels

return(myrootedtree)

}
```

Funzione R `seeFastq()`

```
#######################
## FASTQ Quality Plots ##
#######################
##    Plots quality report for set of FASTQ files including
##         (A) Per cycle box plot of quality
##         (B) Per cycle base proportion
##         (C) Per cycle mean base quality
##         (D) Relative k-mer diversity: unique_k-mers / all_possible_k-mers
##         (E) Number of reads where all Phred scores are above a minimum cutoff
##         (F) Distribution of mean quality of reads
##         (G) Read length distribution
##         (H) Read occurrence distribution

## (A) Compute quality stats and store them in list

seeFastq <- function(fastq, batchsize, klength=8) {
       ## Processing of single fastq file
       seeFastqSingle <- function(fastq, batchsize, klength) {
          ## Random sample N reads from fastq file (N=batchsize)
          f <- FastqSampler(fastq, batchsize)
          fq <- yield(f)
          nReads <- f$status()[["total"]] # Total number of reads in fastq file
          close(f)

          ## If reads are not of constant width then inject them into a matrix pre-populated with
          ## N/NA values and of dimensions N_rows = number_of_reads and N_columns = length_of_longest_read.
          if(length(unique(width(fq))) == 1) {
              q <- as.matrix(PhredQuality(quality(fq)))
              s <- as.matrix(sread(fq))
          } else {
              mymin <- min(width(fq)); mymax <- max(width(fq))
              s <- matrix("N", length(fq), mymax)
              q <- matrix(NA, length(fq), mymax)
              for(i in mymin:mymax) {
                  index <- width(fq)==i
                  if(any(index)) {
                      s[index, 1:i] <- as.matrix(DNAStringSet(sread(fq)[index], start=1,
end=i))
                      q[index, 1:i] <- as.matrix(PhredQuality(quality(fq))[index])
                  }
              }
          }
          s[s=="N"] <- NA
          row.names(q) <- paste("s", 1:length(q[,1]), sep=""); colnames(q) <-
1:length(q[1,])

          ## (A) Per cycle quality box plot
          ## Generate box plot from precomputed stats
          bpl <- boxplot(q, plot=FALSE)
          astats <- data.frame(bpl$names, t(matrix(bpl$stats, dim(bpl$stats))))
          colnames(astats) <- c("Cycle", "min", "low", "mid", "top", "max")
          astats[,1] <- factor(astats[,1], levels=unique(astats[,1]), ordered=TRUE)

          ## (B) Per cycle base proportion
```

```
        bstats <- apply(s, 2, function(x) table(factor(x, levels=c("A", "C", "G",
"T"))))
        colnames(bstats) <- 1:length(bstats[1,])
        bstats <- t(apply(bstats, 1, function(x) x/colSums(bstats)))
        bstats <- data.frame(Nuc=rownames(bstats), bstats)
        convertDF <- function(df=df, mycolnames) {
            myfactor <- rep(colnames(df)[-1], each=length(df[,1]))
            mydata <- as.vector(as.matrix(df[,-1]))
            df <- data.frame(df[,1], mydata, myfactor)
            colnames(df) <- mycolnames
            return(df)
        }
        bstats <- convertDF(bstats, mycolnames=c("Base", "Frequency", "Cycle"))
        bstats[,3] <- as.numeric(gsub("X", "", bstats[,3]))
        bstats[,3] <- factor(bstats[,3], levels=unique(bstats[,3]), ordered=TRUE)

        ## (C) Per cycle average quality of each base type
        A <- q; A[s %in% c("T", "G", "C")] <- NA; A <- colMeans(A, na.rm=TRUE)
        T <- q; T[s %in% c("A", "G", "C")] <- NA; T <- colMeans(T, na.rm=TRUE)
        G <- q; G[s %in% c("T", "A", "C")] <- NA; G <- colMeans(G, na.rm=TRUE)
        C <- q; C[s %in% c("T", "G", "A")] <- NA; C <- colMeans(C, na.rm=TRUE)
        cstats <- data.frame(Quality=c(A, C, G, T), Base=rep(c("A", "C", "G", "T"),
each=length(A)), Cycle=c(names(A), names(C), names(G), names(T)))
        cstats[,3] <- factor(cstats[,3], levels=unique(cstats[,3]), ordered=TRUE)

        ## (D) Relative K-mer Diversity
        dna <- sread(fq)
        loopv <- 1:(min(width(dna)) - (klength-1))
        kcount <- sapply(loopv, function(x) length(unique(DNAStringSet(start=x,
end=x+klength-1, dna))))
        reldiv <- kcount/(5^klength) # 5 instead of 4 because of Ns
        reldiv <- c(rep(NA, klength-1), reldiv) # Adds dummy NAs to align with sequencing cycles
        names(reldiv) <- 1:length(reldiv)
        dstats <- data.frame(RelDiv=reldiv, Method=rep(c(1), each=length(reldiv)),
Cycle=names(reldiv))
        dstats[,3] <- factor(dstats[,3], levels=unique(dstats[,3]), ordered=TRUE)

        ## (E) Number of reads where all Phred scores are above a minimum cutoff
        ev <- c("0"=0, "1"=10, "2"=20, "3"=30, "4"=40)
        edf <- sapply(ev, function(x) sapply(as.numeric(names(ev)), function(y)
sum(rowSums(q >= x, na.rm=TRUE) >= (rowSums(!is.na(q))-y))))
        rownames(edf) <- names(ev); colnames(edf) <- ev
        edf <- edf/max(edf)*100
        edf <- data.frame(Percent=paste(">", colnames(edf), sep=""), t(edf),
check.names=FALSE)
        estats <- convertDF(edf, mycolnames=c("minQuality", "Percent", "Outliers"))
        estats[,1] <- factor(estats[,1], levels=unique(estats[,1]), ordered=TRUE)
        estats[,3] <- factor(estats[,3], levels=unique(estats[,3]), ordered=TRUE)

        ## (F) Distribution of mean quality of reads
        qv <- table(round(rowMeans(q)))[as.character(0:max(q, na.rm=TRUE))]
        qv[is.na(qv)] <- 0; names(qv) <- 0:max(q, na.rm=TRUE)
        fstats <- data.frame(Quality=names(qv), Percent=as.numeric(qv))
        fstats[,2] <- as.numeric(as.vector(fstats[,2])) / length(q[,1]) * 100
        fstats[,1] <- factor(fstats[,1], levels=unique(fstats[,1]), ordered=TRUE)

        ## (G) Read length distribution
        l <- rep(0, max(width(fq))); names(l) <- 1:length(l)
        lv <- table(width(fq))
        l[names(lv)] <- lv
```

```
        gstats <- data.frame(Cycle=names(l), Percent=l)
        gstats[,2] <- gstats[,2] / sum(gstats[,2]) * 100
        gstats[,1] <- factor(gstats[,1], levels=unique(gstats[,1]), ordered=TRUE)

        ## (H) Read occurrence distribution
        qa1 <- qa(fq, basename(fastq))
        hstats <- qa1[["sequenceDistribution"]][,1:2]
        hstats <- data.frame(nOccurrences=hstats[,1], Percent=hstats[,1] * hstats[,2] /
batchsize * 100)
        hstats[,1] <- factor(hstats[,1], levels=unique(hstats[,1]), ordered=TRUE)

        ## Assemble results in list
        return(list(fqstats=c(batchsize=batchsize, nReads=nReads, klength=klength),
astats=astats, bstats=bstats, cstats=cstats, dstats=dstats, estats=estats,
fstats=fstats, gstats=gstats, hstats=hstats))
    }
    ## Loop to run seeFastqSingle on one or many fastq files
    fqlist <- lapply(names(fastq), function(x) seeFastqSingle(fastq=fastq[x],
batchsize=batchsize, klength=klength))
    names(fqlist) <- names(fastq)
    return(fqlist)
}
## Alias
# fastqQuality <- seeFastq
```

Funzione R `seeFastqPlot()`

```
## (B) Plot seeFastq results

seeFastqPlot <- function(fqlist, arrange=c(1,2,3,4,5,8,6,7), ...) {
    ## Create plotting instances from fqlist
    fastqPlot <- function(x=fqlist) {
        ## (A) Per cycle quality box plot
        astats <- x[[1]][["astats"]]
        a <- ggplot(astats, aes(x=Cycle, ymin = min, lower = low, middle = mid,
            upper = top, ymax = max)) +
            geom_boxplot(stat = "identity", color="#606060", fill="#56B4E9") +
            scale_x_discrete(breaks=c(1, seq(0, length(astats[,1]), by=10)[-1])) +
            ylab("Quality") +
            theme(legend.position = "none", plot.title = element_text(size = 12)) +
            ggtitle(names(x))

        ## (B) Per cycle base proportion
        bstats <- x[[1]][["bstats"]]
        b <- ggplot(bstats, aes(Cycle, Frequency, fill=Base), color="black") +
            scale_x_discrete(breaks=c(1, seq(0, length(unique(bstats$Cycle)),
                  by=10)[-1])) +
            geom_bar(stat="identity") +
            theme(legend.position="top", legend.key.size=unit(0.2, "cm")) +
            ylab("Proportion")

        ## (C) Per cycle average quality of each base type
        cstats <- x[[1]][["cstats"]]
        c <- ggplot(cstats, aes(Cycle, Quality, group=Base, color=Base)) +
            geom_line() +
            scale_x_discrete(breaks=c(1, seq(0, length(unique(bstats$Cycle)),
                  by=10)[-1])) +
            theme(legend.position = "none")

        ## (D) Relative K-mer Diversity
        dstats <- x[[1]][["dstats"]]
        d <- ggplot(dstats, aes(Cycle, RelDiv, group=Method, color=Method)) +
            geom_line() +
            scale_x_discrete(breaks=c(1, seq(0, length(unique(bstats$Cycle)),
                  by=10)[-1])) +
            ylab(paste("k", x[[1]][["fqstats"]][["klength"]], "-mer Div", sep="")) +
            theme(legend.position = "none")

        ## (E) Number of reads where all Phred scores are above a minimum cutoff
        estats <- x[[1]][["estats"]]
        e <- ggplot(estats, aes(minQuality, Percent, fill = Outliers)) +
            geom_bar(position="dodge", stat="identity") +
            theme(legend.position="top", legend.key.size=unit(0.2, "cm")) +
            xlab("All Bases Above Min Quality") +
            ylab("% Reads")

        ## (F) Distribution of mean quality of reads
        fstats <- x[[1]][["fstats"]]
        f <- ggplot(fstats, aes(Quality, Percent)) +
            geom_bar(fill="#0072B2", stat="identity") +
            theme(legend.position = "none", plot.title = element_text(size = 9)) +
            ggtitle(paste(formatC(x[[1]][["fqstats"]][["batchsize"]], big.mark = ",",
            format="f", digits=0), "of", formatC(x[[1]][["fqstats"]][["nReads"]],
            big.mark = ",", format="f", digits=0), "Reads")) +
```

```
                    scale_x_discrete(breaks=c(0, seq(0, length(unique(fstats$Quality)),
by=5)[-1])) +
            xlab("Mean Quality") +
            ylab("% Reads")

        ## (G) Read length distribution
        gstats <- x[[1]][["gstats"]]
        g <- ggplot(gstats, aes(Cycle, Percent)) +
            geom_bar(fill="#0072B2", stat="identity") +
            theme(legend.position = "none") +
            scale_x_discrete(breaks=c(1, seq(0, length(unique(gstats$Cycle)),
                  by=10)[-1])) +
            xlab("Read Length") +
            ylab("% Reads")

        ## (H) Read occurrence distribution
        hstats <- x[[1]][["hstats"]]
        myintervals <- data.frame(labels=c("1", "2-10", "11-100", "101-1k", "1k-10k",
">10k"), lower=c(1,2,11,101,1001,10001), upper=c(2,11,101,1001,10001,Inf))
        iv <- sapply(seq(along=myintervals[,1]), function(x)
sum(hstats[as.numeric(as.vector(hstats$nOccurrences)) >= myintervals[x,2] &
as.numeric(as.vector(hstats$nOccurrences)) < myintervals[x,3], "Percent"]))
        hstats <- data.frame(labels=myintervals[,1], Percent=iv)
        hstats[,1] <- factor(hstats[,1], levels=unique(hstats[,1]), ordered=TRUE)
        h <- ggplot(hstats, aes(labels, Percent)) +
            geom_bar(fill="#0072B2", stat="identity") +
            theme(legend.position = "none") +
            xlab("Read Occurrence") +
            ylab("% Reads")

        ## Assemble results in list
        return(list(a=a, b=b, c=c, d=d, g=g, e=e, f=f, h=h))
    }
    ## Loop to run fastqPlot and store instances in list
    fqplot <- lapply(names(fqlist), function(z) fastqPlot(x=fqlist[z]))
    names(fqplot) <- names(fqlist)

    ## Final plot
    grid.newpage() # open a new page on grid device
    pushViewport(viewport(layout = grid.layout(length(arrange), length(fqplot))))
    for(i in seq(along=fqplot)) {
        for(j in seq(along=arrange)) {
            suppressWarnings(print(fqplot[[i]][[arrange[j]]],
                  vp = viewport(layout.pos.row = j, layout.pos.col = i)))
        }
    }
}
## Alias
# plotFQ <- seeFastqPlot
```

Funzione R `slidingwindowplot()`

```
slidingwindowplot <- function(windowsize, inputseq) {

starts <- seq(1, length(inputseq)-windowsize, by = windowsize)
n <- length(starts) # Find the length of the vector "starts"
chunkGCs <- numeric(n) # Make a vector of the same length as vector "starts", but just containing zeroes

for (i in 1:n) {
      chunk <- inputseq[starts[i]:(starts[i]+windowsize-1)]
      chunkGC <- GC(chunk)
      print(chunkGC)
      chunkGCs[i] <- chunkGC
}

plot(starts,chunkGCs,type="b",xlab="Nucleotide start position",ylab="GC content")

}
```

Funzione R SNP`freqCalc()`

```
SNPfreqCalc <- function(df, prefix){

require(SNPassoc)
require(GWASExactHW)

freq <- data.frame() # initialize a data.frame
x <- c() # initialize an empty vector counting for NAs
for (i in 1:ncol(df)){ # iterate over the data for each snp
      t <- as.numeric(summary(df[,i])) # get value for each SNP
      freq <- rbind(freq,t) # append to the data.frame
      if (any(is.na(d[,i])) | length(t)<3){ # removes NA values
            x <- c(x,i)
      }
}

rownames(freq) <- colnames(df)
colnames(freq) <- c("nAA", "nAa", "naa")
freq <- freq[-x,]
freq <- freq[, c("nAA", "nAa", "naa")]
freq <- freq[which(substr(rownames(freq),1,nchar(prefix))==prefix),]
return(freq)

}
```

Funzione R `unrootedNJtree()`

```
unrootedNJtree <- function(alignment,type) {

# This function requires the ape and seqinR packages:
require("ape")
require("seqinr")

# Define a function for making a tree:
makemytree <- function(alignmentmat) {
      alignment <- ape::as.alignment(alignmentmat)
      if (type == "protein") {
            mydist <- dist.alignment(alignment)
      }
      else if (type == "DNA") {
            alignmentbin <- as.DNAbin(alignment)
            mydist <- dist.dna(alignmentbin)
      }
      mytree <- nj(mydist)
      mytree <- makeLabel(mytree, space="") # get rid of spaces in tip names
      return(mytree)
}

# Infer a tree:
mymat  <- as.matrix.alignment(alignment)
mytree <- makemytree(mymat)

# Bootstrap the tree:
myboot <- boot.phylo(mytree, mymat, makemytree)

# Plot the tree:
plot.phylo(mytree,type="u") # plot the unrooted phylogenetic tree
nodelabels(myboot,cex=0.7) # plot the bootstrap values
mytree$node.label <- myboot # make the bootstrap values be the node labels

return(mytree)

}
```

Funzione R `viterbi()`

```
##
# This carries out the Viterbi algorithm.
# Adapted from "Applied Statistics for Bioinformatics using R" by Wim P. Krijnen, page 209
# ( cran.r-project.org/doc/contrib/Krijnen-IntroBioInfStatistics.pdf )
##

viterbi <- function(sequence, transitionmatrix, emissionmatrix) {

# Get the names of the states in the HMM:
states <- rownames(emissionmatrix)

# Make the Viterbi matrix v:
v <- makeViterbimat(sequence, transitionmatrix, emissionmatrix)

# Go through each of the rows of the matrix v (where each row represents a position in
# the DNA sequence), and find out which column has the maximum value for that row (where
# each column represents one state of the HMM):
mostprobablestatepath <- apply(v, 1, function(x) which.max(x))

# Print out the most probable state path:
prevnucleotide <- sequence[1]
prevmostprobablestate <- mostprobablestatepath[1]
prevmostprobablestatename <- states[prevmostprobablestate]
startpos <- 1
for (i in 2:length(sequence)) {
      nucleotide <- sequence[i]
      mostprobablestate <- mostprobablestatepath[i]
      mostprobablestatename <- states[mostprobablestate]
      if (mostprobablestatename != prevmostprobablestatename) {
            print(paste("Positions",startpos,"-",(i-1), "Most probable state = ",
prevmostprobablestatename))
            startpos <- i
      }
      prevnucleotide <- nucleotide
      prevmostprobablestatename <- mostprobablestatename
}

print(paste("Positions",startpos,"-",i, "Most probable state = ",
prevmostprobablestatename))

}
```