Bioconductor

An Introduction to Core Technologies

Kasper D. Hansen

Bioconductor

An Introduction to Core Technologies

Kasper D. Hansen

This book is for sale at http://leanpub.com/bioconductor

This version was published on 2016-05-23

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Contents

Preface

My first real exposure to the Bioconductor project was in 2003 where I attended, and helped organize, a workshop on Bioconductor at Kolle Kolle in Denmark taught by Sandrine Dudoit and Vince Carey. Looking back, this workshop had a profound impact on my career. It led to Sandrine becoming my Ph.D. advisor and started my ongoing involvement with the Bioconductor project.

At that time I had a Masters in Statistics and was working in the Department of Biostatistics at University of Copenhagen as a research assistant. I knew next to nothing about computational biology, except that I had gotten tired of using statistics to do epidemiology and I was looking for some other application area for my statistical skills. But I had a good background in R, having used it since at least version 0.65, so I was asked to help with the workshop by the local organizer, Peter Dalgaard (then and now a member of R-core), who was a colleague at the Department of Biostatistics.

During my Ph.D. I developed several R packages released through the Bioconductor project. I became involved in the user community, first through the email list and later by attending the yearly Bioconductor meeting. Eventually, I joined the technical advisory board for the project.

My example is an illustration of the Bioconductor goal of "users are encouraged to become developers". Over the years I have seen other participants from the email lists (and now online support forum) go from beginners to experienced developers. I have benefited immensely from my involvement in the Bioconductor community and I highly encourage other users to take the same path. I hope this book will be a guide in this process for some of you.

1. What is Bioconductor

The Bioconductor[1] project is an open source and open development platform for computational biology. It is a also a repository of packages for the R[2] programming language, focused on computational biology.

"Open source" means that anyone can view (and modify) the source code.

"Open development" means that anyone can join in developing software for Bioconductor, although the software needs to satisfy certain minimal requirements.

In practice, Bioconductor is a loosely structured collection of software packages, developed by different groups all over the world. The software packages differ enormously in size and quality, although there are certain (enforced) minimum requirements. In general, these requirements mean that the software can be used on many different platforms and comes with a decent level of documentation. It is common that a single task is addressed by multiple packages, that may then compete with each other for users. Anyone is allowed to contribute a new package to the project, even if the package contains functionality covered by an existing package. This ensures some amount of healthy competition in the project.

The Bioconductor website and repository is maintained by a core group of developers, the "core team". Beside maintaining the website, support forum and the software repository, the core team is also actively involved in package development. They usually focus on infrastructure packages, which are widely useful to a significant class of users and developers.

The repository contains three types of packages: software, annotation and experimental data packages. Software packages are classic R packages addressing a specific problem with computational and statistical methods. Annotation packages are bundles of annotation which are compiled for use in Bioconductor, for example microarray annotation files. And finally experimental data packages are cleaned and processed data for common use, for example in package documentation and testing, or in teaching.

1.1 What is this book

This book provides an overview of the core technologies used in Bioconductor. It is my opinion that **anyone** who uses Bioconductor ought to have some understanding of the subjects covered here.

A main component of Bioconductor's success has been that it provides very useful statistical and graphical functionality for Genomic Data Science. This book – perhaps surprisingly – does not cover

[1] http://www.bioconductor.org

[2] http://www.r-project.org

much of this functionality. Using Bioconductor for Genomic Data Science requires some knowledge of statistics and genomics, and I have tried to write this book book without assuming much (if any) knowledge of these subjects. As such, most readers will need to learn specific packages and workflows for the types of analysis they want to do; the last chapters provide some (hopefully) useful examples of this.

This book is not a comprehensive text on the core technologies of Bioconductor. Instead I have tried to provide an **overview** which should give the interested reader a "big picture" idea of how the core components fit together. After reading this book, you'll probably find that there are many additional important details. To learn this, you will have to start using the different package vignettes and help pages; this book will give you a firm foundation to start doing this.

I hope you'll enjoy the book; feel free to give feedback on twitter or by email.

2. Installing Bioconductor

2.1 Installing Bioconductor

The one true way to install Bioconductor is by using the `biocLite` script. You get access to this script by sourcing it from the Bioconductor website

```
> source("http://www.bioconductor.org/biocLite.R")
```

The first time you run the script without arguments, it will install a core set of Bioconductor packages.

```
> biocLite()
```

When you run this script, it will autodetect if any of your installed packages are out of date; it will aggressively ask you to update your packages.

Because of this, the way you update a Bioconductor installation, is just by running `biocLite()` without any argument.

You can check if your installation is fully up-to-date, by running `biocValid()`; it will return `TRUE` if everything is current.

You install a new package by using `biocLite` with the package name, for example

```
> biocLite("limma")
```

The reason why you want to use `biocLite` - and only this function - to install and update Bioconductor, is because one of the top problems users have is when they mix and match Bioconductor packages from different releases. Using `biocLite` ensures that everything is synchronized.

The way you update Bioconductor itself when a new release comes out, is by updating R itself and then run `biocLite()`.

There are more comprehensive installation instructions on the Bioconductor site[1].

[1] http://bioconductor.org/install

3. Online Resources for Bioconductor

Watch video 1[1] and video 2[2] of this chapter.

3.1 Overview

Bioconductor is very well documented, compared to most other pieces of academic software. But sometimes the documentation is hard to find or hard to get started with. One of the problems is that sometimes useful documentation is scattered around different sources and sometimes it is hard to find exactly what you want to accomplish a given task. This document has a short overview of some of the more useful web sites and resources. The intention is that, as you learn more and more, you will return to some of these sources to get the gory details.

3.2 Bioconductor packages and documentation

Bioconductor is organized into packages and there are minimal requirements for the documentation of a package. All we can really check is whether the documentation is there, not whether it is useful. But it is my experience that most packages are very well documented, although they sometimes assume some basic familiarity with the conventions of the project.

Bioconductor has been a leader in the R community wrt. package *vignettes*. A vignette is a small manual, typically giving a holistic overview of the package and its capabilities. The first thing I do, when I examine a new package, is to skim the vignette to get some idea of what I can accomplish with the software. A package can contain multiple vignettes.

In addition to the package vignettes, there are the man pages. These help pages describes the details of each function, how to use it and what the arguments are. But most of the time, the different help pages does not provide a good idea of how to put it all together to achieve a task. This is the intention of the vignette(s).

Package vignettes are installed inside of R. You can access vignettes in the following ways

1. Through the online help (for example in RStudio).
2. Through the `vignettes()` function in R, but that requires you know the name of the vignette.
3. Through the `browseVignettes()` function in *Biobase* which shows a list of installed vignettes in a browser (this is a different interface to the RStudio help interface).

[1] https://youtu.be/TIj2ckwJmqM
[2] https://youtu.be/290_-Ca5iAk

In addition, the vignettes are available from the Bioconductor site[3]; this is often how I access them.

A useful trick is to get a listing of all help pages in a given package. You can do this through the online HTML help (not all ways of interacting with R gives you access to the HTML help system). Another useful trick (which I use all the time) is to do

```
> library(NAME = help)
```

where `NAME` is the package name.

3.3 The Bioconductor site

The Bioconductor site[4] has a wealth of great information. Here are some pointers

- Workflows[5]. This is a new addition to Bioconductor; the intention is to provide across-package description of useful functionality.
- Software packages[6]; note the use of `biocViews` to the left of the page, this might be useful if you are searching for specific functionality.

There are also developer HOWTOs, which are very useful for developing packages. And you can browse an exhaustive list of the different packages.

3.4 Other resources

- The Bioconductor support site[7]; this is a great place to ask questions.
- The posting guide[8] for the Bioconductor support site. Read this before asking questions; it will maximize your chance of getting a useful answer.
- Stack Overflow (R)[9]; this is a popular help site for computer programming.
- Stack Overflow (Bioconductor)[10]; this is a popular help site for computer programming.
- R Documentation[11]; a search engine for all documentation from all packages from CRAN and Bioconductor.
- R Seek[12]; like "R Documentation" but it also search a few other sites and is based on a different search engine.

[3] http://www.bioconductor.org
[4] http://www.bioconductor.org
[5] http://bioconductor.org/help/workflows/
[6] http://bioconductor.org/packages/release/BiocViews.html#___Software
[7] https://support.bioconductor.org
[8] http://bioconductor.org/help/support/posting-guide/
[9] http://stackoverflow.com/questions/tagged/r
[10] http://stackoverflow.com/questions/tagged/bioconductor
[11] http://www.rdocumentation.org/
[12] http://rseek.org

4. An overview of base types in R

Watch a video[1] of this chapter.

4.1 Dependencies

This document has no dependencies.

4.2 Overview

Bioconductor has a rich class of different types of objects. Some are used to represent entire experiments (such as `ExpressionSet`, covered later) and some are used to represent simpler structures. It is my experience that a prerequisite for understanding and using many Bioconductor objects efficiently, is a good understanding of the different base types in R. This document contains a brief overview of these objects and how to subset and manipulate them.

4.3 Atomic Vectors

The most basic object in R is an atomic vector. Examples includes `numeric`, `integer`, `logical`, `character` and `factor`. These objects have a single length and can have names, which can be used for indexing

```
> x <- 1:10
> names(x) <- letters[1:10]
> class(x)
[1] "integer"
> x[1:3]
a b c
1 2 3
> x[c("a", "b")]
a b
1 2
```

The following types of atomic vectors are used frequently

[1] https://youtu.be/bw55cuD6bqA

- `numeric` - for numeric values.
- `integer` - for integer values.
- `character` - for characters (strings).
- `factor` - for factors.
- `logical` - for logical values.

All vectors can have missing values.

Note: names of vectors does not need to be unique. This can lead to subsetting problems:

```
> x <- 1:3
> names(x) <- c("A", "A", "B")
> x
A A B
1 2 3
> x["A"]
A
1
```

Note that you donâ€™t even get a warning, so watch out for non-unique names! You can check for unique names by using the functions `unique`, `duplicated` or (easiest) `anyDuplicated`.

```
> anyDuplicated(names(x))
[1] 2
> names(x) <- c("A", "B", "C")
> anyDuplicated(names(x))
[1] 0
```

`anyDuplicated` returns the index of the first duplicated name, so `0` indicates nothing is duplicated.

Integers in R

The default in R is to represent numbers as `numeric`, NOT `integer`. This is something that can usually be ignored, but you might run into some issues in Bioconductor with this. Note that even constructions that looks like `integer` are really `numeric`:

```
> x <- 1
> class(x)
[1] "numeric"
> x <- 1:3
> class(x)
[1] "integer"
```

The way to make sure to get an `integer` in R is to append `L` to the numbers

```
> x <- 1L
> class(x)
[1] "integer"
```

So why the distinguishing between `integer` and `numeric`? Internally, the way computers represents and calculates numbers are different between `integer` and `numeric`.

- `integer` mathematics are different.
- `numeric` can hold much larger values than `integer`.
- `numeric` takes up slightly more RAM (but nothing to worry about).

Point 2 is something you can sometimes run into, in Bioconductor. The maximum `integer` is

```
> .Machine$integer.max
[1] 2147483647
> 2^31 -1 == .Machine$integer.max
[1] TRUE
> round(.Machine$integer.max / 10^6, 1)
[1] 2147.5
```

This number is smaller than the number of bases in the human genome. So we sometimes (accidentally) add up numbers which exceeds this. The fix is to use `as.numeric` to convert the `integer` to `numeric`.

This number is also the limit for how long an atomic vector can be. So you cannot have a single vector which is as long as the human genome. In R we are beginning to get support for something called “long vectors” which basically are â€¦ long vectors. But the support for long vectors is not yet pervasive.

4.4 Matrices

`matrix` is a two-dimensional object. All values in a `matrix` has to have the same type (`numeric` or `character` or any of the other atomic vector types). It is optional to have `rownames` or `colnames` and these names does not have to be unique.

```
> x <- matrix(1:9, ncol = 3, nrow = 3)
> rownames(x) <- c("A","B", "B")
> x
  [,1] [,2] [,3]
A    1    4    7
B    2    5    8
B    3    6    9
> dim(x)
[1] 3 3
> nrow(x)
[1] 3
> ncol(x)
[1] 3
```

Subsetting is two-dimensional; the first dimension is rows and the second is columns. You can even subset with a matrix of the same dimension, but watch out for the return object.

```
> x[1:2,]
  [,1] [,2] [,3]
A    1    4    7
B    2    5    8
> x["B",]
[1] 2 5 8
> x[x >= 5]
[1] 5 6 7 8 9
```

(note how subsetting with a non-unique name does not lead to an error). If you grab a single row or a single column from a `matrix` you get a vector. Sometimes, it is really nice to get a `matrix`; you do that by using `drop=FALSE` in the subsetting:

```
> x[1,]
[1] 1 4 7
> x[1,,drop=FALSE]
  [,1] [,2] [,3]
A    1    4    7
```

There are a lot of mathematical operations working on matrices, for example `rowSums`, `colSums` and things like `eigen` for eigenvector decomposition. I am a heavy user of the package *matrixStats*[2] for the full suite of `rowXX` and `colXX` with `XX` being any standard statistical function such as `sd()`, `var()`, `quantiles()` etc.

Internally, a `matrix` is just a `vector` with a dimension attribute. In R we have column-first orientation, so the columns are filled up first:

[2] http://cran.fhcrc.org/web/packages/matrixStats/index.html

```
> matrix(1:9, 3, 3)
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
> matrix(1:9, 3, 3, byrow = TRUE)
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
[3,]    7    8    9
```

4.5 Lists

`lists` are like `vectors`, but can hold together objects of arbitrary kind.

```
> x <- list(1:3, letters[1:3], is.numeric)
> x
[[1]]
[1] 1 2 3

[[2]]
[1] "a" "b" "c"

[[3]]
function (x)  .Primitive("is.numeric")
> names(x) <- c("numbers", "letters", "function")
> x[1:2]
$numbers
[1] 1 2 3

$letters
[1] "a" "b" "c"
> x[1]
$numbers
[1] 1 2 3
> x[[1]]
[1] 1 2 3
```

See how subsetting creates another `list`. To get to the actual content of the first element, you need double brackets `[[`. The distinction between `[` and `[[` is critical to understand.

You can use `$` on a named list. However, R has something called "partial" matching for `$`:

```
> x$letters
[1] "a" "b" "c"
> x["letters"]
$letters
[1] "a" "b" "c"
> x$let
[1] "a" "b" "c"
> x["let"]
$<NA>
NULL
```

Trick: sometimes you want a list where each element is a single number. Use `as.list`:

```
> as.list(1:3)
[[1]]
[1] 1

[[2]]
[1] 2

[[3]]
[1] 3
> list(1:3)
[[1]]
[1] 1 2 3
```

lapply and sapply

It is quite common to have a `list` where each element is of the same kind, for example a `numeric` vector. You can apply a function to each element in the `list` by using `lapply()`; this returns another `list` which is named if the input is.

```
> x <- list(a = rnorm(3), b = rnorm(3))
> lapply(x, mean)
$a
[1] -0.5085716

$b
[1] 0.3014588
```

If the output of the function is of the same kind, you can simplify the output using `sapply` (simplify apply). This is particularly useful if the function in question returns a single number.

```
> sapply(x, mean)
         a          b
-0.5085716  0.3014588
```

4.6 Data frames

`data.frame` are fundamental to data analysis. They look like matrices, but each column can be a separate type, so you can mix and match different data types. They are required to have unique column and row names. If no rowname is given, itâ€™ll use `1:nrow`.

```
> x <- data.frame(sex = c("M", "M", "F"), age = c(32,34,29))
> x
  sex age
1   M  32
2   M  34
3   F  29
```

You access columns by `$` or `[[`:

```
> x$sex
[1] M M F
Levels: F M
> x[["sex"]]
[1] M M F
Levels: F M
```

Note how `sex` was converted into a `factor`. This is a frequent source of errors, so much that I highly encourage users to make sure they never have `factors` in their `data.frames`. This conversion can be disabled by `stringsAsfactors=FALSE`:

```
> x <- data.frame(sex = c("M", "M", "F"), age = c(32,34,29), stringsAsFactors = \
FALSE)
> x$sex
[1] "M" "M" "F"
```

Behind the scenes, a `data.frame` is really a `list`. Why does this matter? Well, for one, it allows you to use `lapply` and `sapply` across the columns:

```
> sapply(x, class)
        sex         age
"character"   "numeric"
```

4.7 Conversion

We often have to convert R objects from one type to another. For basic R types (as described above), you have the `as.XX` family of functions, with `XX` being all the types of objects listed above.

```
> x
  sex age
1   M  32
2   M  34
3   F  29
> as.matrix(x)
     sex age
[1,] "M" "32"
[2,] "M" "34"
[3,] "F" "29"
> as.list(x)
$sex
[1] "M" "M" "F"

$age
[1] 32 34 29
```

When we convert the `data.frame` to a `matrix` it becomes a `character` matrix, because there is a `character` column and this is the only way to keep the contents.

For more "complicated" objects there is a suite of `as()` functions, which you use as follows

```
> library(methods)
> as(x, "matrix")
     sex age
[1,] "M" "32"
[2,] "M" "34"
[3,] "F" "29"
```

This is how you convert most Bioconductor objects.

4.8 Other Resources

"An Introduction to R" ships with R and can also be access on the web (HTML[3] | PDF[4]). This introduction contains a lot of useful material but it is written very terse; you will need to pay close attention to the details. It is useful to re-read this introduction after you have used R for a while; you are likely to learn new details you had missed at first.

[3] https://cran.r-project.org/doc/manuals/r-release/R-intro.html
[4] https://cran.r-project.org/doc/manuals/r-release/R-intro.pdf

5. IRanges - Basic Usage

Watch a video[1] of this chapter.

5.1 Dependencies

This document has the following dependencies:

```
> library(IRanges)
```

Use the following commands to install these packages in R.

```
> source("http://www.bioconductor.org/biocLite.R")
> biocLite(c("IRanges"))
```

5.2 Overview

A surprising amount of objects/tasks in computational biology can be formulated in terms of integer intervals, manipulation of integer intervals and overlap of integer intervals.

Objects: A transcript (a union of integer intervals), a collection of SNPs (intervals of width 1), transcription factor binding sites, a collection of aligned short reads.

Tasks: Which transcription factor binding sites hit the promoter of genes (overlap between two sets of intervals), which SNPs hit a collection of exons, which short reads hit a predetermined set of exons.

`IRanges` are collections of integer intervals. `GRanges` are like `IRanges`, but with an associated chromosome and strand, taking care of some book keeping.

Here we discuss `IRanges`, which provides the foundation for `GRanges`. This package implements (amongst other things) an algebra for handling integer intervals.

5.3 Basic IRanges

Specify IRanges by 2 of start, end, width (SEW).

[1] https://youtu.be/YB2WRH3sFHs

```
> ir1 <- IRanges(start = c(1,3,5), end = c(3,5,7))
> ir1
IRanges object with 3 ranges and 0 metadata columns:
          start       end     width
      <integer> <integer> <integer>
  [1]         1         3         3
  [2]         3         5         3
  [3]         5         7         3
> ir2 <- IRanges(start = c(1,3,5), width = 3)
> all.equal(ir1, ir2)
[1] TRUE
```

An `IRanges` consist of separate intervals; each interval is called a range. So `ir1` above contains 3 ranges.

Assessor methods: `start()`, `end()`, `width()` and also replacement methods.

```
> start(ir1)
[1] 1 3 5
> width(ir2) <- 1
> ir2
IRanges object with 3 ranges and 0 metadata columns:
          start       end     width
      <integer> <integer> <integer>
  [1]         1         1         1
  [2]         3         3         1
  [3]         5         5         1
```

They may have names

```
> names(ir1) <- paste("A", 1:3, sep = "")
> ir1
IRanges object with 3 ranges and 0 metadata columns:
         start       end     width
     <integer> <integer> <integer>
  A1         1         3         3
  A2         3         5         3
  A3         5         7         3
```

They have a single dimension

```
> dim(ir1)
NULL
> length(ir1)
[1] 3
```

Because of this, subsetting works like a vector

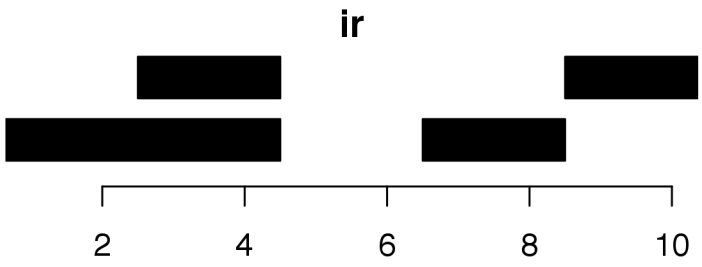
```
> ir1[1]
IRanges object with 1 range and 0 metadata columns:
         start       end     width
     <integer> <integer> <integer>
  A1         1         3         3
> ir1["A1"]
IRanges object with 1 range and 0 metadata columns:
         start       end     width
     <integer> <integer> <integer>
  A1         1         3         3
```

Like vectors, you can concatenate two `IRanges` with the `c()` function

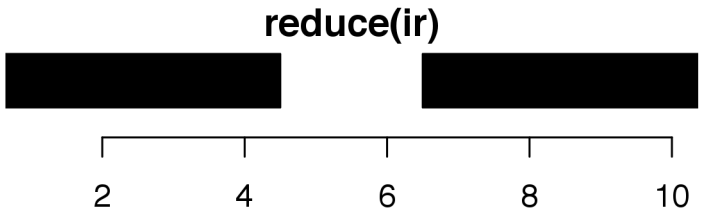
```
> c(ir1, ir2)
IRanges object with 6 ranges and 0 metadata columns:
         start       end     width
     <integer> <integer> <integer>
  A1         1         3         3
  A2         3         5         3
  A3         5         7         3
             1         1         1
             3         3         1
             5         5         1
```

5.4 Normal IRanges

A normal IRanges is a minimal representation of the IRanges viewed as a set. Each integer only occur in a single range and there are as few ranges as possible. In addition, it is ordered. Many functions produce a normal `IRanges`. Created by `reduce()`.



An IRanges consisting of 4 ranges.



reduce() applied to the previous IRanges produces a normal IRanges.

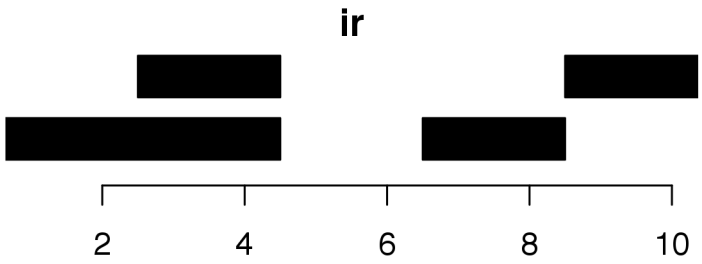
```
> ir
IRanges object with 4 ranges and 0 metadata columns:
          start       end     width
      <integer> <integer> <integer>
  [1]         1         4         4
  [2]         3         4         2
  [3]         7         8         2
  [4]         9        10         2
> reduce(ir)
IRanges object with 2 ranges and 0 metadata columns:
          start       end     width
      <integer> <integer> <integer>
  [1]         1         4         4
  [2]         7        10         4
```

Answers: “Given a set of overlapping exons, which bases belong to an exon?”

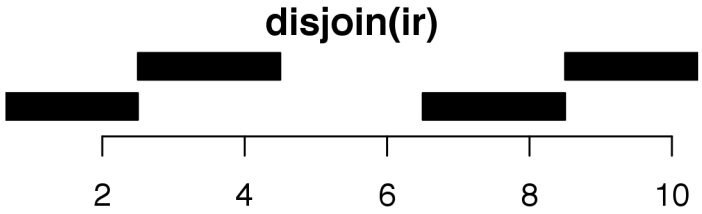
5.5 Disjoin

From some perspective, `disjoin()` is the opposite of `reduce()`. An example explains better:

```
> disjoin(ir1)
```



An IRanges



disjoin() applied to the previous IRanges.

Answers: "Give a set of overlapping exons, which bases belong to the same set of exons?"

5.6 Manipulating IRanges, intra-range

"Intra-range" manipulations are manipulations where each original range gets mapped to a new range.

Examples of these are: `shift()`, `narrow()`, `flank()`, `resize()`, `restrict()`.

For example, `resize()` can be extremely useful. It has a `fix` argument controlling where the resizing occurs from. Use `fix="center"` to resize around the center of the ranges; I use this a lot.

```
> resize(ir, width = 1, fix = "start")
IRanges object with 4 ranges and 0 metadata columns:
          start       end     width
      <integer> <integer> <integer>
  [1]         1         1         1
  [2]         3         3         1
  [3]         7         7         1
  [4]         9         9         1
> resize(ir, width = 1, fix = "center")
IRanges object with 4 ranges and 0 metadata columns:
          start       end     width
      <integer> <integer> <integer>
  [1]         2         2         1
  [2]         3         3         1
  [3]         7         7         1
  [4]         9         9         1
```

The help page is `?"intra-range-methods"` (note that there is both a help page in *IRanges* and *GenomicRanges*).

5.7 Manipulating IRanges, as sets

Manipulating `IRanges` as sets means that we view each `IRanges` as a set of integers; individual integers is either contained in one or more ranges or they are not. This is equivalent to calling `reduce()` on the `IRanges` first.

Once this is done, we can use standard: `union()`, `intersect()`, `setdiff()`, `gaps()` between two `IRanges` (which all returns normalized `IRanges`).

```
> ir1 <- IRanges(start = c(1, 3, 5), width = 1)
> ir2 <- IRanges(start = c(4, 5, 6), width = 1)
> union(ir1, ir2)
IRanges object with 2 ranges and 0 metadata columns:
          start       end     width
      <integer> <integer> <integer>
  [1]         1         1         1
  [2]         3         6         4
> intersect(ir1, ir2)
IRanges object with 1 range and 0 metadata columns:
          start       end     width
      <integer> <integer> <integer>
  [1]         5         5         1
```

Because they return normalized `IRanges`, an alternative to `union()` is

```
> reduce(c(ir1, ir2))
IRanges object with 2 ranges and 0 metadata columns:
          start       end     width
      <integer> <integer> <integer>
  [1]         1         1         1
  [2]         3         6         4
```

There is also an element-wise (pair-wise) version of these: `punion()`, `pintersect()`, `psetdiff()`, `pgap()`; this is similar to say `pmax` from base R. In my experience, these functions are seldom used.

5.8 Finding Overlaps

Finding (pairwise) overlaps between two `IRanges` is done by `findOverlaps()`. This function is very important and amazingly fast!

```
> ir1 <- IRanges(start = c(1,4,8), end = c(3,7,10))
> ir2 <- IRanges(start = c(3,4), width = 3)
> ov <- findOverlaps(ir1, ir2)
> ov
Hits object with 3 hits and 0 metadata columns:
      queryHits subjectHits
      <integer>   <integer>
  [1]         1           1
  [2]         2           1
  [3]         2           2
  -------
  queryLength: 3 / subjectLength: 2
```

It returns a `Hits` object which describes the relationship between the two `IRanges`. This object is basically a two-column matrix of indicies into the two `IRanges`.

The two columns of the hits object can be accessed by `queryHits()` and `subjectHits()` (often used with `unique()`).

For example, the first row of the matrix describes that the first range of `ir1` overlaps with the first range of `ir2`. Or said differently, they have a non-empty intersection:

```
> intersect(ir1[queryHits(ov)[1]],
+           ir2[subjectHits(ov)[2]])
IRanges object with 1 range and 0 metadata columns:
          start       end     width
      <integer> <integer> <integer>
  [1]         3         3         1
```

The elements of `unique(queryHits)` gives you the indices of the query ranges which actually had an overlap; you need `unique` because a query range may overlap multiple subject ranges.

```
> queryHits(ov)
[1] 1 2 2
> unique(queryHits(ov))
[1] 1 2
```

The list of arguments to `findOverlaps()` is long; there are a few hidden treasures here. For example, you can ask to only get an overlap if two ranges overlap by a certain number of bases.

```
> args(findOverlaps)
function (query, subject, maxgap = 0L, minoverlap = 1L, type = c("any",
    "start", "end", "within", "equal"), select = c("all", "first",
    "last", "arbitrary"), ...)
NULL
```

5.9 Counting Overlaps

For efficiency, there is also `countOverlaps()`, which just returns the number of overlaps. This function is more efficient than `findOverlaps()` because it does not have to keep track of which ranges overlap, just the number of overlaps.

```
> countOverlaps(ir1, ir2)
[1] 1 2 0
```

5.10 Finding nearest IRanges

Sometimes you have two sets of `IRanges` and you need to know which ones are closest to each other. Functions for this include `nearest()`, `precede()`, `follow()`. Watch out for ties!

```
> ir1
IRanges object with 3 ranges and 0 metadata columns:
          start       end     width
      <integer> <integer> <integer>
  [1]         1         3         3
  [2]         4         7         4
  [3]         8        10         3
> ir2
IRanges object with 2 ranges and 0 metadata columns:
          start       end     width
      <integer> <integer> <integer>
  [1]         3         5         3
  [2]         4         6         3
> nearest(ir1, ir2)
[1] 1 1 2
```

5.11 Other Resources

The vignette titled "An Introduction to IRanges" from the IRanges package[2].

[2] http://bioconductor.org/packages/IRanges

6. GenomicRanges - GRanges

Watch a video[1] of this chapter.

6.1 Dependencies

This document has the following dependencies:

```
> library(GenomicRanges)
```

Use the following commands to install these packages in R.

```
> source("http://www.bioconductor.org/biocLite.R")
> biocLite(c("GenomicRanges"))
```

6.2 GRanges

`GRanges` are like `IRanges` with strand and chromosome. Strand can be `+`, `-` and `*`. The value `*` indicates 'unknown strand' or 'unstranded'. This value usually gets treated as a third strand, which is sometimes confusing to users (examples below).

They get created with the `GRanges` constructor:

```
> gr <- GRanges(seqnames = "chr1", strand = c("+", "-", "+"),
+               ranges = IRanges(start = c(1,3,5), width = 3))
>
```

Natural accessor functions: `strand()`, `seqnames()`, `ranges()`, `start()`, `end()`, `width()`.

Because the have strand, we now have operations which are relative to the direction of transcription (`upstream()`, `downstream()`):

[1] https://youtu.be/et3zeBXnpdc

```
> flank(gr, 2, start = FALSE)
GRanges object with 3 ranges and 0 metadata columns:
      seqnames    ranges strand
         <Rle> <IRanges>  <Rle>
  [1]     chr1    [4, 5]      +
  [2]     chr1    [1, 2]      -
  [3]     chr1    [8, 9]      +
  -------
  seqinfo: 1 sequence from an unspecified genome; no seqlengths
```

6.3 GRanges, seqinfo

`GRanges` operate within a universe of sequences (chromosomes/contigs) and their lengths.

This is described through `seqinfo`:

```
> seqinfo(gr)
Seqinfo object with 1 sequence from an unspecified genome; no seqlengths:
  seqnames seqlengths isCircular genome
  chr1             NA         NA   <NA>
> seqlengths(gr) <- c("chr1" = 10)
> seqinfo(gr)
Seqinfo object with 1 sequence from an unspecified genome:
  seqnames seqlengths isCircular genome
  chr1             10         NA   <NA>
> seqlevels(gr)
[1] "chr1"
> seqlengths(gr)
chr1
  10
```

Especially the length of the chromosomes are used by some functions. For example `gaps()` return the stretches of the genome not covered by the `GRanges`.

```
> gaps(gr)
GRanges object with 5 ranges and 0 metadata columns:
      seqnames    ranges strand
         <Rle> <IRanges>  <Rle>
  [1]     chr1   [4,  4]      +
  [2]     chr1   [8, 10]      +
  [3]     chr1   [1,  2]      -
  [4]     chr1   [6, 10]      -
  [5]     chr1   [1, 10]      *
  -------
  seqinfo: 1 sequence from an unspecified genome
```

In this example, we know that the last gap stops at 10, because that is the length of the chromosome. Note how a range on the * strand appears in the result.

Let us expand the `GRanges` with another chromosome

```
> seqlevels(gr) <- c("chr1", "chr2")
> seqnames(gr) <- c("chr1", "chr2", "chr1")
```

When you `sort()` a `GRanges`, the sorting order of the chromosomes is determined by their order in `seqlevel`. This is nice if you want the sorting "chr1", "chr2", ..., "chr10", ...

```
> sort(gr)
GRanges object with 3 ranges and 0 metadata columns:
      seqnames    ranges strand
         <Rle> <IRanges>  <Rle>
  [1]     chr1    [1, 3]      +
  [2]     chr1    [5, 7]      +
  [3]     chr2    [3, 5]      -
  -------
  seqinfo: 2 sequences from an unspecified genome
> seqlevels(gr) <- c("chr2", "chr1")
> sort(gr)
GRanges object with 3 ranges and 0 metadata columns:
      seqnames    ranges strand
         <Rle> <IRanges>  <Rle>
  [1]     chr2    [3, 5]      -
  [2]     chr1    [1, 3]      +
  [3]     chr1    [5, 7]      +
  -------
  seqinfo: 2 sequences from an unspecified genome
```

You can associate a genome with a `GRanges`.

```
> genome(gr) <- "hg19"
> gr
GRanges object with 3 ranges and 0 metadata columns:
      seqnames    ranges strand
         <Rle> <IRanges>  <Rle>
  [1]     chr1    [1, 3]      +
  [2]     chr2    [3, 5]      -
  [3]     chr1    [5, 7]      +
  -------
  seqinfo: 2 sequences from hg19 genome
```

This becomes valuable when you deal with data from different genome versions (as we all do), because it allows R to throw an error when you compare two `GRanges` from different genomes, like

```
> gr2 <- gr
> genome(gr2) <- "hg18"
> findOverlaps(gr, gr2)
Error in mergeNamedAtomicVectors(genome(x), genome(y), what = c("sequence", : se\
quences chr2, chr1 have incompatible genomes:
  - in 'x': hg19, hg19
  - in 'y': hg18, hg18
```

The fact that each sequence may have its own genome is more esoteric. One usecase is for experiments where the experimenter have spiked in sequences exogenous to the original organism.

6.4 Other Resources

- The vignettes from the GenomicRanges package[2].
- The package is described in a paper in PLOS Computational Biology (Lawrence et al. 2013).

6.5 References

Lawrence, Michael, Wolfgang Huber, Hervé Pagès, Patrick Aboyoun, Marc Carlson, Robert Gentleman, Martin T Morgan, and Vincent J Carey. 2013. "Software for computing and annotating genomic ranges." *PLoS Computational Biology* 9 (8): e1003118. doi:10.1371/journal.pcbi.1003118.

[2] http://bioconductor.org/packages/GenomicRanges

7. GenomicRanges - Basic GRanges Usage

Watch a video[1] of this chapter.

7.1 Dependencies

This document has the following dependencies:

```
> library(GenomicRanges)
```

Use the following commands to install these packages in R.

```
> source("http://www.bioconductor.org/biocLite.R")
> biocLite(c("GenomicRanges"))
```

7.2 DataFrame

The *S4Vectors* package introduced the `DataFrame` class. This class is very similar to the base `data.frame` class from R, but it allows columns of any class, provided a number of required methods are supported. For example, `DataFrame` can have `IRanges` as columns, unlike `data.frame`:

```
> ir <- IRanges(start = 1:2, width = 3)
> df1 <- DataFrame(iranges = ir)
> df1
DataFrame with 2 rows and 1 column
    iranges
  <IRanges>
1    [1, 3]
2    [2, 4]
> df1$iranges
IRanges object with 2 ranges and 0 metadata columns:
          start       end     width
      <integer> <integer> <integer>
```

[1] https://youtu.be/dxoIvuRLGuk

```
  [1]         1         3         3
  [2]         2         4         3
> df2 <- data.frame(iranges = ir)
> df2
  iranges.start iranges.end iranges.width
1             1           3             3
2             2           4             3
```

In the `data.frame` case, the `IRanges` gives rise to 4 columns, whereas it is a single column when a `DataFrame` is used.

Think of this as an expanded and more versatile class.

7.3 GRanges, metadata

`GRanges` (unlike `IRanges`) may have associated metadata. This is immensely useful. The formal way to access and set this metadata is through `values` or `elementMetadata` or `mcols`, like

```
> gr <- GRanges(seqnames = "chr1", strand = c("+", "-", "+"),
+               ranges = IRanges(start = c(1,3,5), width = 3))
> values(gr) <- DataFrame(score = c(0.1, 0.5, 0.3))
> gr
GRanges object with 3 ranges and 1 metadata column:
      seqnames    ranges strand |     score
         <Rle> <IRanges>  <Rle> | <numeric>
  [1]     chr1    [1, 3]      + |       0.1
  [2]     chr1    [3, 5]      - |       0.5
  [3]     chr1    [5, 7]      + |       0.3
  -------
  seqinfo: 1 sequence from an unspecified genome; no seqlengths
```

A much easier way to set and access metadata is through the `$` operator

```
> gr$score
[1] 0.1 0.5 0.3
> gr$score2 = gr$score * 0.2
> gr
GRanges object with 3 ranges and 2 metadata columns:
      seqnames    ranges strand |     score    score2
         <Rle> <IRanges>  <Rle> | <numeric> <numeric>
  [1]     chr1    [1, 3]      + |       0.1      0.02
  [2]     chr1    [3, 5]      - |       0.5       0.1
  [3]     chr1    [5, 7]      + |       0.3      0.06
  -------
  seqinfo: 1 sequence from an unspecified genome; no seqlengths
```

7.4 findOverlaps

`findOverlaps` works exactly as for `IRanges`. But the `strand` information can be confusing. Let us make an example

```
> gr2 <- GRanges(seqnames = c("chr1", "chr2", "chr1"), strand = "*",
+                ranges = IRanges(start = c(1, 3, 5), width = 3))
> gr2
GRanges object with 3 ranges and 0 metadata columns:
      seqnames    ranges strand
         <Rle> <IRanges>  <Rle>
  [1]     chr1    [1, 3]      *
  [2]     chr2    [3, 5]      *
  [3]     chr1    [5, 7]      *
  -------
  seqinfo: 2 sequences from an unspecified genome; no seqlengths
> gr
GRanges object with 3 ranges and 2 metadata columns:
      seqnames    ranges strand |     score    score2
         <Rle> <IRanges>  <Rle> | <numeric> <numeric>
  [1]     chr1    [1, 3]      + |       0.1      0.02
  [2]     chr1    [3, 5]      - |       0.5       0.1
  [3]     chr1    [5, 7]      + |       0.3      0.06
  -------
  seqinfo: 1 sequence from an unspecified genome; no seqlengths
```

Note how the `ranges` in the two `GRanges` object are the same coordinates, they just have different `seqnames` and `strand`. Let us try to do a standard `findOverlaps`:

```
> findOverlaps(gr, gr2)
Hits object with 4 hits and 0 metadata columns:
      queryHits subjectHits
      <integer>   <integer>
  [1]         1           1
  [2]         2           1
  [3]         2           3
  [4]         3           3
  -------
  queryLength: 3 / subjectLength: 3
```

Notice how the * strand overlaps both + and -. There is an argument `ignore.strand` to `findOverlaps` which will ... ignore the strand information (so + overlaps -). Several other functions in `GenomicRanges` have an `ignore.strand` argument as well.

7.5 subsetByOverlaps

A common operation is to select only certain ranges from a `GRanges` which overlap something else. Enter the convenience function `subsetByOverlaps`

```
> subsetByOverlaps(gr, gr2)
GRanges object with 3 ranges and 2 metadata columns:
      seqnames    ranges strand |     score    score2
         <Rle> <IRanges>  <Rle> | <numeric> <numeric>
  [1]     chr1    [1, 3]      + |       0.1      0.02
  [2]     chr1    [3, 5]      - |       0.5       0.1
  [3]     chr1    [5, 7]      + |       0.3      0.06
  -------
  seqinfo: 1 sequence from an unspecified genome; no seqlengths
```

7.6 makeGRangesFromDataFrame

A common situation is that you have data which looks like a `GRanges` but is really stored as a classic `data.frame`, with `chr`, `start` etc. The `makeGRangesFromDataFrame` converts this `data.frame` into a `GRanges`. An argument tells you whether you want to keep any additional columns.

```
> df <- data.frame(chr = "chr1", start = 1:3, end = 4:6, score = 7:9)
> makeGRangesFromDataFrame(df)
GRanges object with 3 ranges and 0 metadata columns:
      seqnames    ranges strand
         <Rle> <IRanges>  <Rle>
  [1]     chr1    [1, 4]      *
  [2]     chr1    [2, 5]      *
  [3]     chr1    [3, 6]      *
  -------
  seqinfo: 1 sequence from an unspecified genome; no seqlengths
> makeGRangesFromDataFrame(df, keep.extra.columns = TRUE)
GRanges object with 3 ranges and 1 metadata column:
      seqnames    ranges strand |     score
         <Rle> <IRanges>  <Rle> | <integer>
  [1]     chr1    [1, 4]      * |         7
  [2]     chr1    [2, 5]      * |         8
  [3]     chr1    [3, 6]      * |         9
  -------
  seqinfo: 1 sequence from an unspecified genome; no seqlengths
```

7.7 Biology usecase I

Suppose we want to identify transcription factor (TF) binding sites that overlaps known SNPs.

Input objects are
snps: a `GRanges` (of width 1)
TF: a `GRanges`

pseudocode:

```
> findOverlaps(snps, TF)
```

(watch out for strand)

7.8 Biology usecase II

Suppose we have a set of differentially methylation regions (DMRs) (think genomic regions) and a set of CpG Islands and we want to find all DMRs within 10kb of a CpG Island.

Input objects are
dmrs: a `GRanges`
islands: a `GRanges`

pseudocode:

```
> big_islands <- resize(islands, width = 20000 + width(islands), fix = "center")
> findOverlaps(dmrs, big_islands)
```

(watch out for strand)

7.9 Other Resources

- The vignettes from the GenomicRanges package[2].
- The package is described in a paper in PLOS Computational Biology (Lawrence et al. 2013).

7.10 References

Lawrence, Michael, Wolfgang Huber, Hervé Pagès, Patrick Aboyoun, Marc Carlson, Robert Gentleman, Martin T Morgan, and Vincent J Carey. 2013. "Software for computing and annotating genomic ranges." *PLoS Computational Biology* 9 (8): e1003118. doi:10.1371/journal.pcbi.1003118.

[2] http://bioconductor.org/packages/GenomicRanges

8. GenomicRanges - More on seqinfo

Watch a video[1] of this chapter.

8.1 Dependencies

This document has the following dependencies:

```
> library(GenomeInfoDb)
> library(GenomicRanges)
```

Use the following commands to install these packages in R.

```
> source("http://www.bioconductor.org/biocLite.R")
> biocLite(c("GenomeInfoDb", "GenomicRanges"))
```

8.2 Overview

The `GRanges` class contains `seqinfo` information about the length and the names of the chromosomes. Here we will briefly discuss strategies for harmonizing this information.

The *GenomeInfoDb* package addresses a seemingly small, but consistent problem: different online resources uses different naming conventions for chromosomes. In more technical jargon, this package helps keeping your `seqinfo` and `seqlevels` harmonized.

8.3 Drop and keep seqlevels

It is common to want to remove `seqlevels` from a `GRanges` object. Here are some equivalent methods

[1] https://youtu.be/nEJIvoUmuBM

```
> gr <- GRanges(seqnames = c("chr1", "chr2"),
+               ranges = IRanges(start = 1:2, end = 4:5))
> seqlevels(gr, force=TRUE) <- "chr1"
> gr
GRanges object with 1 range and 0 metadata columns:
      seqnames    ranges strand
         <Rle> <IRanges>  <Rle>
  [1]     chr1    [1, 4]      *
  -------
  seqinfo: 1 sequence from an unspecified genome; no seqlengths
```

In *GenomeInfoDb* (loaded when you load *GenomicRanges*) you find `dropSeqlevels()` and `keepSeqlevels()`.

```
> gr <- GRanges(seqnames = c("chr1", "chr2"),
+               ranges = IRanges(start = 1:2, end = 4:5))
> dropSeqlevels(gr, "chr1")
GRanges object with 1 range and 0 metadata columns:
      seqnames    ranges strand
         <Rle> <IRanges>  <Rle>
  [1]     chr2    [2, 5]      *
  -------
  seqinfo: 1 sequence from an unspecified genome; no seqlengths
> keepSeqlevels(gr, "chr2")
GRanges object with 1 range and 0 metadata columns:
      seqnames    ranges strand
         <Rle> <IRanges>  <Rle>
  [1]     chr2    [2, 5]      *
  -------
  seqinfo: 1 sequence from an unspecified genome; no seqlengths
```

You can also just get rid of weird looking chromosome names with `keepStandardChromosomes()`.

```
> gr <- GRanges(seqnames = c("chr1", "chrU345"),
+               ranges = IRanges(start = 1:2, end = 4:5))
> keepStandardChromosomes(gr)
GRanges object with 1 range and 0 metadata columns:
      seqnames    ranges strand
         <Rle> <IRanges>  <Rle>
  [1]     chr1    [1, 4]      *
  -------
  seqinfo: 1 sequence from an unspecified genome; no seqlengths
```

8.4 Changing style

It is an inconvenient truth that different online resources uses different naming convention for chromosomes. This can even be different from organism to organism. For example, for the fruitfly (Drosophila Melanogaster) NCBI and Ensembl uses "2L" and UCSC uses "chr2L". But NCBI and Ensembl differs on some contigs: NCBI uses "Un" and Ensembl used "U".

```
> gr <- GRanges(seqnames = "chr1", ranges = IRanges(start = 1:2, width = 2))
```

Let us remap

```
> newStyle <- mapSeqlevels(seqlevels(gr), "NCBI")
> gr <- renameSeqlevels(gr, newStyle)
```

This can in principle go wrong, if the original set of `seqlevels` are inconsistent (not a single style).

The *GenomeInfoDb* also contains information for dropping / keeping various classes of chromosomes:

8.5 Using information from BSgenome packages

`BSgenome` packages contains `seqinfo` on their genome objects. This contains `seqlengths` and other information. An easy trick is to use these packages to correct your `seqinfo` information.

Hopefully, in Bioconductor 3.2 we will get support for `seqlengths` in *GenomeInfoDb* so we can avoid using the big genome packages.

8.6 Other Resources

- The vignette from the GenomeInfoDb package[2].

[2] http://bioconductor.org/packages/GenomeInfoDb

9. AnnotationHub

Watch a video[1] of this chapter.

9.1 Dependencies

This document has the following dependencies:

```
> library(AnnotationHub)
```

Use the following commands to install these packages in R.

```
> source("http://www.bioconductor.org/biocLite.R")
> biocLite(c("AnnotationHub"))
```

9.2 Overview

Annotation information is extremely important for putting your data into context. There are many online resources for doing this, and many different databases organizes different information suing different approaches.

There are multiple ways to access annotation information in Bioconductor.

Here we discuss a new way of doing so, through the package *AnnotationHub*. This package provides access to a ton of online resources through a unified interface. However, each data resource has its own peculiarities, so a user still needs to understand what the different datasets are.

In a recent paper I was involved in (Hansen et al. 2014), I used *AnnotationHub* to interrogate my data against all transcription factor data available through the ENCODE project. I managed to write the code and conduct the analysis in the matter of a single evening, which I think is pretty awesome.

9.3 Usage

First we create an `AnnotationHub` instance. The first time you do this, it will create a local cache on your system, so that repeat queries for the same information (even in different R sessions) will be very fast.

[1] https://youtu.be/bw55cuD6bqA

```
> ah <- AnnotationHub()
> ah
```

As you can see, `ah` contains tons of information. The information content is constantly changing, which is why there is a `snapshotDate`. While the object is big, it actually only contains pointers to online information. Actually downloading all the resources available in an `AnnotationHub` is prohibitive.

The object is organized as a vector, with single-dimension indexing. You can get information about a single resource by indexing with a single `[`; using two brackets (`[[`) downloads the object:

```
> ah[1]
AnnotationHub with 1 record
# snapshotDate(): 2016-05-12
# names(): AH2
# $dataprovider: Ensembl
# $species: Ailuropoda melanoleuca
# $rdataclass: FaFile
# $title: Ailuropoda_melanoleuca.ailMel1.69.dna.toplevel.fa
# $description: FASTA DNA sequence for Ailuropoda melanoleuca
# $taxonomyid: 9646
# $genome: ailMel1
# $sourcetype: FASTA
# $sourceurl: ftp://ftp.ensembl.org/pub/release-69/fasta/ailuropoda_mel...
# $sourcelastmodifieddate: 2012-10-12
# $sourcesize: 693412448
# $tags: FASTA, ensembl, sequence
# retrieve record with 'object[["AH2"]]'
```

The way you use *AnnotationHub* is by using various tools to narrow down your hub to a single or a small number of datasets. Then you download these datasets for your own usage.

Let us first explore some high-level features of the hub:

```
> unique(ah$dataprovider)
 [1] "Ensembl"
 [2] "EncodeDCC"
 [3] "UCSC"
 [4] "RefNet"
 [5] "Inparanoid8"
 [6] "NCBI"
 [7] "NHLBI"
 [8] "ChEA"
```

```
 [9] "Pazar"
[10] "NIH Pathway Interaction Database"
[11] "Haemcode"
[12] "GEO"
[13] "BroadInstitute"
[14] "ftp://ftp.ncbi.nlm.nih.gov/gene/DATA/"
[15] "PRIDE"
[16] "Gencode"
[17] "dbSNP"
> unique(ah$rdataclass)
 [1] "FaFile"           "GRanges"          "data.frame"
 [4] "Inparanoid8Db"    "OrgDb"            "TwoBitFile"
 [7] "ChainFile"        "SQLiteConnection" "biopax"
[10] "BigWigFile"       "ExpressionSet"    "AAStringSet"
[13] "MSnSet"           "mzRpwiz"          "mzRident"
[16] "VcfFile"
```

(we will discuss many of these data classes in future sessions).

You can narrow down the hub by using one (or more) of the following strategies

- Use `subset` (or `[`) to do a specific subsetting operation.
- Use `query` to do a command-line search over the metadata of the hub.
- Use `display` to get a Shiny interface in a browser, so you can browse the object.

It is often useful to start with a very rough subsetting, for example to data from a specific species. The `subset` function is useful for doing a standard R subsetting (the function also works on `data.frames`).

```
> ah <- subset(ah, species == "Homo sapiens")
> ah
AnnotationHub with 30411 records
# snapshotDate(): 2016-05-12
# $dataprovider: BroadInstitute, EncodeDCC, UCSC, Ensembl, Gencode, NIH...
# $species: Homo sapiens
# $rdataclass: GRanges, BigWigFile, ChainFile, FaFile, TwoBitFile, data...
# additional mcols(): taxonomyid, genome, description, tags,
#   sourceurl, sourcetype
# retrieve records with, e.g., 'object[["AH133"]]'

            title
  AH133   | Homo_sapiens.GRCh37.69.cdna.all.fa
  AH134   | Homo_sapiens.GRCh37.69.dna.toplevel.fa
```

```
  AH135   | Homo_sapiens.GRCh37.69.dna_rm.toplevel.fa
  AH136   | Homo_sapiens.GRCh37.69.dna_sm.toplevel.fa
  AH137   | Homo_sapiens.GRCh37.69.ncrna.fa
  ...       ...
  AH50558 | Homo_sapiens.GRCh38.cdna.all.2bit
  AH50559 | Homo_sapiens.GRCh38.dna.primary_assembly.2bit
  AH50560 | Homo_sapiens.GRCh38.dna_rm.primary_assembly.2bit
  AH50561 | Homo_sapiens.GRCh38.dna_sm.primary_assembly.2bit
  AH50562 | Homo_sapiens.GRCh38.ncrna.2bit
```

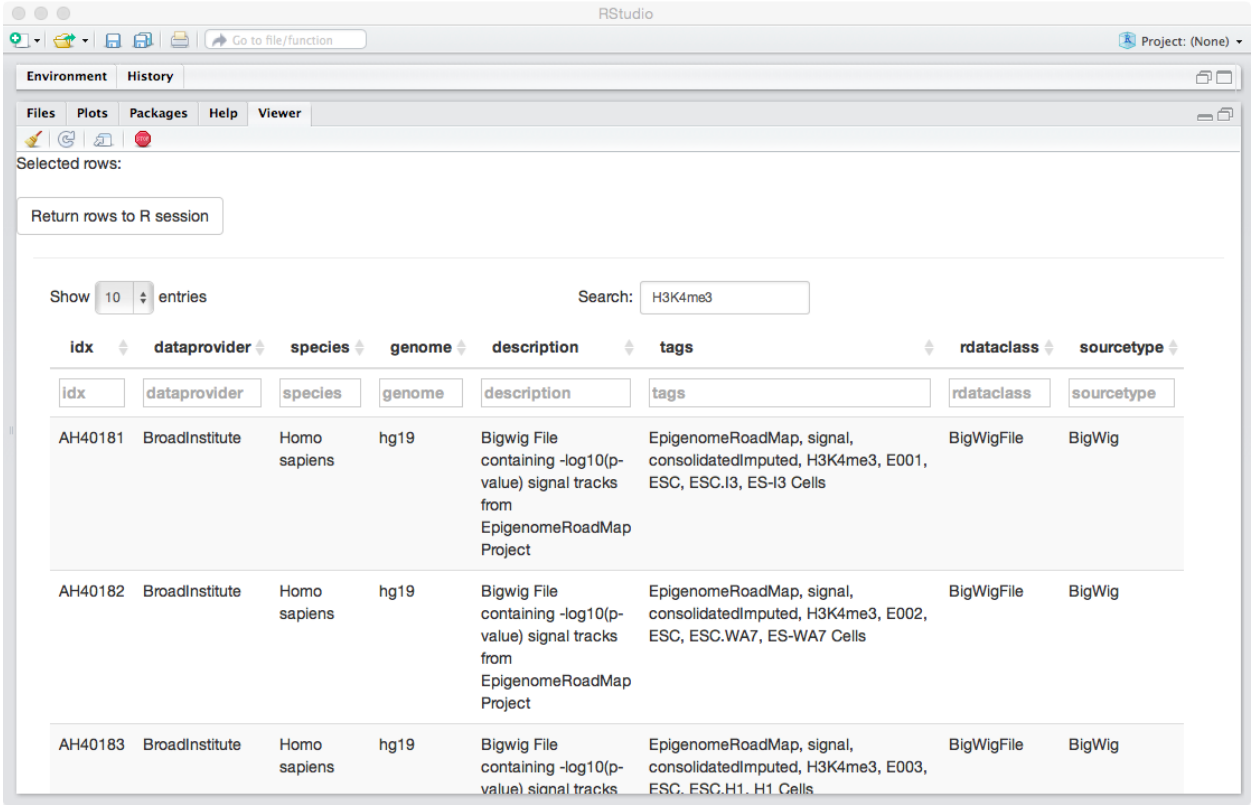
We can use `query` to search the hub. The (possible) drawback to query is that it searches over different fields in the hub, so watch out with using a search term which is very non-specific. The query is a regular expression, which by default is case-insensitive. Here we locate all datasets on the **H3K4me3** histone modification (in H. sapiens because we selected this species above)

```
> query(ah, "H3K4me3")
AnnotationHub with 2308 records
# snapshotDate(): 2016-05-12
# $dataprovider: BroadInstitute, EncodeDCC, UCSC
# $species: Homo sapiens
# $rdataclass: GRanges, BigWigFile
# additional mcols(): taxonomyid, genome, description, tags,
#   sourceurl, sourcetype
# retrieve records with, e.g., 'object[["AH706"]]'

            title
  AH706   | wgEncodeBroadHistoneA549H3k04me3Dex100nmPk
  AH707   | wgEncodeBroadHistoneA549H3k04me3Etoh02Pk
  AH730   | wgEncodeBroadHistoneDnd41H3k04me3Pk
  AH742   | wgEncodeBroadHistoneGm12878H3k04me3StdPkV2
  AH749   | wgEncodeBroadHistoneGm12878H3k4me3StdPk
  ...       ...
  AH46826 | UW.Fetal_Muscle_Leg.H3K4me3.H-24644.Histone.DS21536.gappedP...
  AH46833 | UW.Fetal_Muscle_Trunk.H3K4me3.H-24851.Histone.DS23302.gappe...
  AH46839 | UW.Fetal_Placenta.H3K4me3.H-24996.Histone.DS23300.gappedPea...
  AH46845 | UW.Fetal_Stomach.H3K4me3.H-24639.Histone.DS22598.gappedPeak.gz
  AH46851 | UW.Fetal_Thymus.H3K4me3.H-24644.Histone.DS21539.gappedPeak.gz
```

Another way of searching a hub is by using a browser. Notice how we assign the output of `display` to make sure that we can capture our selection in the browser

```
> hist <- display(ah)
```



display(ah)

9.4 Other Resources

- The *AnnotationHub* vignette from the AnnotationHub package[2].
- The Annotation Resources[3] workflow on Bioconductor contains material on *AnnotationHub*.
- The BioC 2015 conference had a tutorial on *AnnotationHub*; material is available through the course materials[4] site.

9.5 References

Hansen, Kasper D, Sarven Sabunciyan, Ben Langmead, Noemi Nagy, Rebecca Curley, Georg Klein, Eva Klein, Daniel Salamon, and Andrew P Feinberg. 2014. “Large-scale hypomethylated blocks associated with Epstein-Barr virus-induced B-cell immortalization.” *Genome Research* 24 (2): 177â€“84. doi:10.1101/gr.157743.113.

[2] http://bioconductor.org/packages/AnnotationHub

[3] http://www.bioconductor.org/help/workflows/annotation/Annotation_Resources/

[4] http://www.bioconductor.org/help/course-materials/

10. Usecase - Basic GRanges and AnnotationHub

Watch video 1[1] and video 2[2] of this chapter.

10.1 Dependencies

This document has the following dependencies:

```
> library(GenomicRanges)
> library(rtracklayer)
> library(AnnotationHub)
```

Use the following commands to install these packages in R.

```
> source("http://www.bioconductor.org/biocLite.R")
> biocLite(c("GenomicRanges", "rtracklayer", "AnnotationHub"))
```

10.2 Overview

We are going to use *AnnotationHub* and *GenomicRanges* to access ENCODE data on the H3K4me3 histone modification in a specific cell line. This histone modification is believed to mark active promoters, and we are going to attempt to verify this statement. This involves

1. Getting the ENCODE histone data using *AnnotationHub*.
2. Getting promoters using *AnnotationHub*.
3. Comparing the histone data and promoters using `findOverlaps` in *GenomicRanges*.

10.3 Accomplishing our goals

First we use *AnnotationHub* to get data on homo sapiens.

[1] https://youtu.be/5XVfLe8GtdI
[2] https://youtu.be/08r_l0x4L1k

```
> ah <- AnnotationHub()
> ah <- subset(ah, species == "Homo sapiens")
```

Next we search for two keywords: **H3K4me3** and **Gm12878** which is the name of the cell line we are interested in.

```
> qhs <- query(ah, "H3K4me3")
> qhs <- query(qhs, "Gm12878")
```

(Note: I like to keep my full annotation hub around, so I can re-do my query with a different search term in case I end up with no hits. This is why I start assigning output to the `qhs` object and not `ah`).

Lets have a look

```
> qhs
AnnotationHub with 17 records
# snapshotDate(): 2016-05-12
# $dataprovider: BroadInstitute, EncodeDCC, UCSC
# $species: Homo sapiens
# $rdataclass: GRanges, BigWigFile
# additional mcols(): taxonomyid, genome, description, tags,
#   sourceurl, sourcetype
# retrieve records with, e.g., 'object[["AH742"]]'

            title
  AH742   | wgEncodeBroadHistoneGm12878H3k04me3StdPkV2
  AH749   | wgEncodeBroadHistoneGm12878H3k4me3StdPk
  AH4472  | wgEncodeUwHistoneGm12878H3k4me3StdHotspotsRep1
  AH4473  | wgEncodeUwHistoneGm12878H3k4me3StdHotspotsRep2
  AH4474  | wgEncodeUwHistoneGm12878H3k4me3StdPkRep1
  ...       ...
  AH30747 | E116-H3K4me3.narrowPeak.gz
  AH31690 | E116-H3K4me3.gappedPeak.gz
  AH32869 | E116-H3K4me3.fc.signal.bigwig
  AH33901 | E116-H3K4me3.pval.signal.bigwig
  AH40294 | E116-H3K4me3.imputed.pval.signal.bigwig
```

Note how some of these hits don't contain **Gm12878** in their title. This is a useful illustration of how `query` searches over multiple fields.

Lets have a closer look at this

```
> qhs$title
 [1] "wgEncodeBroadHistoneGm12878H3k04me3StdPkV2"
 [2] "wgEncodeBroadHistoneGm12878H3k4me3StdPk"
 [3] "wgEncodeUwHistoneGm12878H3k4me3StdHotspotsRep1"
 [4] "wgEncodeUwHistoneGm12878H3k4me3StdHotspotsRep2"
 [5] "wgEncodeUwHistoneGm12878H3k4me3StdPkRep1"
 [6] "wgEncodeUwHistoneGm12878H3k4me3StdPkRep2"
 [7] "wgEncodeBroadHistoneGm12878H3k4me3StdPk.broadPeak.gz"
 [8] "wgEncodeUwHistoneGm12878H3k4me3StdHotspotsRep1.broadPeak.gz"
 [9] "wgEncodeUwHistoneGm12878H3k4me3StdHotspotsRep2.broadPeak.gz"
[10] "wgEncodeUwHistoneGm12878H3k4me3StdPkRep1.narrowPeak.gz"
[11] "wgEncodeUwHistoneGm12878H3k4me3StdPkRep2.narrowPeak.gz"
[12] "E116-H3K4me3.broadPeak.gz"
[13] "E116-H3K4me3.narrowPeak.gz"
[14] "E116-H3K4me3.gappedPeak.gz"
[15] "E116-H3K4me3.fc.signal.bigwig"
[16] "E116-H3K4me3.pval.signal.bigwig"
[17] "E116-H3K4me3.imputed.pval.signal.bigwig"
> qhs$dataprovider
 [1] "EncodeDCC"      "EncodeDCC"      "EncodeDCC"      "EncodeDCC"
 [5] "EncodeDCC"      "EncodeDCC"      "UCSC"           "UCSC"
 [9] "UCSC"           "UCSC"           "UCSC"           "BroadInstitute"
[13] "BroadInstitute" "BroadInstitute" "BroadInstitute" "BroadInstitute"
[17] "BroadInstitute"
```

This result is a great illustration of the mess of public data. It turns our that **E116** is a Roadmap Epigenetics code for the **Gm12878** cell line. The first 5 hits are from ENCODE, hosted at UCSC and the last 6 hits are from Roadmap Epigenomics hosted at the Broad Institute. The Roadmap data is different representation (and peaks) from the same underlying data. For the ENCODE data, two different centers did the same experiment in the same cell line (`Broad`, hit 1) and (`Uw`, hit 2-5), where `Uw` exposed data on two replicates (whatever that means). These two experiments seems to be analyzed using different algorithms. It is even possible that the Roadmap data is from the same raw data but just analyzed using different algorithms.

Lets take a look at the `narrowPeak` data:

```
> gr1 <- subset(qhs, title == "wgEncodeUwHistoneGm12878H3k4me3StdPkRep1.narrowPe\
ak.gz")[[1]]
> gr1
GRanges object with 74470 ranges and 6 metadata columns:
          seqnames                ranges strand |        name     score
             <Rle>             <IRanges>  <Rle> | <character> <numeric>
      [1]     chr1       [713301, 713450]      * |        <NA>         0
      [2]     chr1       [713501, 713650]      * |        <NA>         0
      [3]     chr1       [713881, 714030]      * |        <NA>         0
      [4]     chr1       [714181, 714330]      * |        <NA>         0
      [5]     chr1       [714481, 714630]      * |        <NA>         0
      ...      ...                    ...    ... .         ...       ...
  [74466]     chrX [154492741, 154492890]      * |        <NA>         0
  [74467]     chrX [154493401, 154493550]      * |        <NA>         0
  [74468]     chrX [154560441, 154560590]      * |        <NA>         0
  [74469]     chrX [154562121, 154562270]      * |        <NA>         0
  [74470]     chrX [154842061, 154842210]      * |        <NA>         0
          signalValue    pValue    qValue      peak
            <numeric> <numeric> <numeric> <numeric>
      [1]          91  112.7680        -1        -1
      [2]          25   26.7181        -1        -1
      [3]          81   77.4798        -1        -1
      [4]          32  106.5650        -1        -1
      [5]         122  153.8320        -1        -1
      ...         ...       ...       ...       ...
  [74466]          43  52.20930        -1        -1
  [74467]         122 203.16800        -1        -1
  [74468]           8   4.49236        -1        -1
  [74469]           8   4.41978        -1        -1
  [74470]         125 170.20100        -1        -1
  -------
  seqinfo: 93 sequences (1 circular) from hg19 genome
> gr2 <- subset(qhs, title == "E116-H3K4me3.narrowPeak.gz")[[1]]
> gr2
GRanges object with 76188 ranges and 6 metadata columns:
          seqnames                ranges strand |        name     score
             <Rle>             <IRanges>  <Rle> | <character> <numeric>
      [1]     chr9 [123553464, 123557122]      * |      Rank_1       623
      [2]     chr3 [ 53196213,  53197995]      * |      Rank_2       622
      [3]    chr18 [  9137534,   9142676]      * |      Rank_3       583
      [4]    chr11 [ 75110593,  75111943]      * |      Rank_4       548
      [5]    chr13 [ 41343776,  41345943]      * |      Rank_5       543
```

```
    ...      ...                   ...    ... .         ...        ...
  [76184]     chr8 [ 98881411,  98881613]      * |  Rank_76184        20
  [76185]     chr9 [ 26812360,  26812533]      * |  Rank_76185        20
  [76186]     chrX [ 49019816,  49020016]      * |  Rank_76186        20
  [76187]     chrX [ 55932872,  55933045]      * |  Rank_76187        20
  [76188]     chr5 [118322235, 118322408]      * |  Rank_76188        20
         signalValue    pValue    qValue      peak
           <numeric> <numeric> <numeric> <numeric>
     [1]    23.09216  62.34132  52.85911      1736
     [2]    24.91976  62.22835  52.85911       439
     [3]    22.48938  58.39180  50.67302       442
     [4]    22.47056  54.84914  47.77176       752
     [5]    20.82804  54.31837  47.29083       909
     ...         ...       ...       ...       ...
  [76184]     2.63471   2.02539   0.27458        94
  [76185]     2.63471   2.02539   0.27458        32
  [76186]     2.63471   2.02539   0.27458        82
  [76187]     2.63471   2.02539   0.27458        41
  [76188]     2.62206   2.00970   0.26790        52
  -------
  seqinfo: 93 sequences (1 circular) from hg19 genome
```

(Note: I use this code, where I use `title` to refer to the different resources, to make this script more robust over time).

This gives us two `GRanges` objects. Let us look at the distribution of peak widths:

```
> summary(width(gr1))
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  150.0   150.0   150.0   150.3   150.0   410.0
> table(width(gr1))

  150   210   230   250   270   290   390   410
74313     1    13    24    37    77     4     1
> summary(width(gr2))
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
    174     243     387     662     770   12340
```

Turns out that almost all peaks in `gr1` have a width of 150bp, whereas `gr2` is much more variable. This is likely a product of the data processing algorithm; it can be very hard to figure out the details of this.

At this time one can spend a lot of time thinking about which datasets is best. We will avoid this (important) discussion; since we referred to ENCODE data above (in the Overview section), we will stick with `gr1`.

Now we need to get some promoter coordinates. There are multiple ways to do this in Bioconductor, but here I will do a quick lookup for RefSeq in my annotation hub. RefSeq is a highly curated (aka conservative) collection of genes.

Lets get started

```
> qhs <- query(ah, "RefSeq")
> qhs
AnnotationHub with 8 records
# snapshotDate(): 2016-05-12
# $dataprovider: UCSC
# $species: Homo sapiens
# $rdataclass: GRanges
# additional mcols(): taxonomyid, genome, description, tags,
#   sourceurl, sourcetype
# retrieve records with, e.g., 'object[["AH5040"]]'

            title
  AH5040 | RefSeq Genes
  AH5041 | Other RefSeq
  AH5155 | RefSeq Genes
  AH5156 | Other RefSeq
  AH5306 | RefSeq Genes
  AH5307 | Other RefSeq
  AH5431 | RefSeq Genes
  AH5432 | Other RefSeq
```

So this gives us multiple datasets, all with very similar names. We probably need the thing called `RefSeq Genes` and not `Other RefSeq` but why are there multiple resources with the same name?

Turns out the answer makes sense:

```
> qhs$genome
[1] "hg19" "hg19" "hg18" "hg18" "hg17" "hg17" "hg16" "hg16"
```

This looks like the same resources, but in different genome builds. We have

```
> genome(gr1)
                  chr1                   chr2                   chr3
                "hg19"                 "hg19"                 "hg19"
                  chr4                   chr5                   chr6
                "hg19"                 "hg19"                 "hg19"
                  chr7                   chr8                   chr9
                "hg19"                 "hg19"                 "hg19"
                 chr10                  chr11                  chr12
                "hg19"                 "hg19"                 "hg19"
                 chr13                  chr14                  chr15
                "hg19"                 "hg19"                 "hg19"
                 chr16                  chr17                  chr18
                "hg19"                 "hg19"                 "hg19"
                 chr19                  chr20                  chr21
                "hg19"                 "hg19"                 "hg19"
                 chr22                    chrX                   chrY
                "hg19"                 "hg19"                 "hg19"
                  chrM   chr1_gl000191_random   chr1_gl000192_random
                "hg19"                 "hg19"                 "hg19"
       chr4_ctg9_hap1   chr4_gl000193_random   chr4_gl000194_random
                "hg19"                 "hg19"                 "hg19"
        chr6_apd_hap1          chr6_cox_hap2          chr6_dbb_hap3
                "hg19"                 "hg19"                 "hg19"
       chr6_mann_hap4          chr6_mcf_hap5          chr6_qbl_hap6
                "hg19"                 "hg19"                 "hg19"
       chr6_ssto_hap7   chr7_gl000195_random   chr8_gl000196_random
                "hg19"                 "hg19"                 "hg19"
 chr8_gl000197_random   chr9_gl000198_random   chr9_gl000199_random
                "hg19"                 "hg19"                 "hg19"
 chr9_gl000200_random   chr9_gl000201_random chr11_gl000202_random
                "hg19"                 "hg19"                 "hg19"
      chr17_ctg5_hap1 chr17_gl000203_random chr17_gl000204_random
                "hg19"                 "hg19"                 "hg19"
chr17_gl000205_random chr17_gl000206_random chr18_gl000207_random
                "hg19"                 "hg19"                 "hg19"
chr19_gl000208_random chr19_gl000209_random chr21_gl000210_random
                "hg19"                 "hg19"                 "hg19"
       chrUn_gl000211         chrUn_gl000212         chrUn_gl000213
                "hg19"                 "hg19"                 "hg19"
       chrUn_gl000214         chrUn_gl000215         chrUn_gl000216
                "hg19"                 "hg19"                 "hg19"
       chrUn_gl000217         chrUn_gl000218         chrUn_gl000219
```

```
        "hg19"              "hg19"              "hg19"
chrUn_gl000220      chrUn_gl000221      chrUn_gl000222
        "hg19"              "hg19"              "hg19"
chrUn_gl000223      chrUn_gl000224      chrUn_gl000225
        "hg19"              "hg19"              "hg19"
chrUn_gl000226      chrUn_gl000227      chrUn_gl000228
        "hg19"              "hg19"              "hg19"
chrUn_gl000229      chrUn_gl000230      chrUn_gl000231
        "hg19"              "hg19"              "hg19"
chrUn_gl000232      chrUn_gl000233      chrUn_gl000234
        "hg19"              "hg19"              "hg19"
chrUn_gl000235      chrUn_gl000236      chrUn_gl000237
        "hg19"              "hg19"              "hg19"
chrUn_gl000238      chrUn_gl000239      chrUn_gl000240
        "hg19"              "hg19"              "hg19"
chrUn_gl000241      chrUn_gl000242      chrUn_gl000243
        "hg19"              "hg19"              "hg19"
chrUn_gl000244      chrUn_gl000245      chrUn_gl000246
        "hg19"              "hg19"              "hg19"
chrUn_gl000247      chrUn_gl000248      chrUn_gl000249
        "hg19"              "hg19"              "hg19"
```

so we know which one to get:

```
> refseq <- qhs[qhs$genome == "hg19" & qhs$title == "RefSeq Genes"]
> refseq
AnnotationHub with 1 record
# snapshotDate(): 2016-05-12
# names(): AH5040
# $dataprovider: UCSC
# $species: Homo sapiens
# $rdataclass: GRanges
# $title: RefSeq Genes
# $description: GRanges object from UCSC track 'RefSeq Genes'
# $taxonomyid: 9606
# $genome: hg19
# $sourcetype: UCSC track
# $sourceurl: rtracklayer://hgdownload.cse.ucsc.edu/goldenpath/hg19/dat...
# $sourcelastmodifieddate: NA
# $sourcesize: NA
# $tags: refGene, UCSC, track, Gene, Transcript, Annotation
# retrieve record with 'object[["AH5040"]]'
> refseq <- refseq[[1]] ## Downloads
```

Lets have a look

```
> refseq
UCSC track 'refGene'
UCSCData object with 50066 ranges and 5 metadata columns:
                     seqnames              ranges strand |         name
                        <Rle>           <IRanges>  <Rle> |  <character>
      [1]                chr1 [66999825, 67210768]      + |    NM_032291
      [2]                chr1 [ 8378145,  8404227]      + | NM_001080397
      [3]                chr1 [48998527, 50489626]      - |    NM_032785
      [4]                chr1 [16767167, 16786584]      + | NM_001145277
      [5]                chr1 [16767167, 16786584]      + | NM_001145278
      ...                 ...                 ...    ... .          ...
  [50062] chr19_gl000209_random     [ 57209,  68123]      + |    NM_002255
  [50063] chr19_gl000209_random     [ 46646,  68123]      + | NM_001258383
  [50064] chr19_gl000209_random     [ 98135, 112667]      + |    NM_012313
  [50065] chr19_gl000209_random     [ 70071,  84658]      + | NM_001083539
  [50066] chr19_gl000209_random     [131433, 145745]      + |    NM_012312
            score     itemRgb               thick
        <numeric> <character>           <IRanges>
      [1]         0        <NA> [67000042, 67208778]
      [2]         0        <NA> [ 8378169,  8404073]
      [3]         0        <NA> [48999845, 50489468]
      [4]         0        <NA> [16767257, 16785491]
      [5]         0        <NA> [16767257, 16785385]
      ...       ...         ...                 ...
  [50062]         0        <NA>     [ 57249,  67717]
  [50063]         0        <NA>     [ 57132,  67717]
  [50064]         0        <NA>     [ 98146, 112480]
  [50065]         0        <NA>     [ 70108,  83979]
  [50066]         0        <NA>     [131468, 145120]
                                                 blocks
                                            <IRangesList>
      [1] [    1,   227] [91706, 91769] [98929, 98953] ...
      [2]       [   1,  102] [6222, 6642] [7214, 7306] ...
      [3]       [   1, 1439] [2036, 2062] [6788, 6884] ...
      [4]       [   1,  182] [2961, 3061] [7199, 7303] ...
      [5]       [   1,  104] [2961, 3061] [7199, 7303] ...
      ...                                           ...
  [50062]       [   1,   80] [ 280,  315] [1182, 1466] ...
  [50063] [    1,    86] [10414, 10643] [10843, 10878] ...
  [50064]       [   1,   46] [1523, 1557] [4002, 4301] ...
  [50065]       [   1,   71] [1071, 1106] [1851, 2135] ...
```

```
  [50066]       [   1,   69] [ 862,  897] [3334, 3633] ...
  -------
  seqinfo: 93 sequences (1 circular) from hg19 genome
```

Let us look at the number of isoforms per gene name:

```
> table(table(refseq$name))

    1     2     3     4     5     6     7     8     9    10    12    13
45178   568   102    27    74    52   171   129    13    19     1     1
   19
    5
```

(the `table(table())` construction may seems weird at first, but its a great way to get a quick tabular summary of occurrences with the same name). This shows that almost all genes have a single transcript, which reflects how conservative RefSeq is.

Notice that each transcript is a separate range. Therefore, each transcript is represented as the "outer" coordinates of the gene; the ranges does not exclude introns. Because we got this from UCSC I happen to know that the `$blocks` metadata really contains the coordinates of the different exons. In a later example we will introduce a gene representation where we keep track of exons, introns and transcripts from the same gene, through something known as a `TxDb` (transcript database) object.

For now, we just need the promoters, so we don't really care about introns. We need to keep track of which strand each transcript is on, to get the transcription start site.

Or we could just use the convenience function `promoters()`:

```
> promoters <- promoters(refseq)
> table(width(promoters))

 2200
50066
> args(promoters)
NULL
```

There are many definitions of promoters based on transcription start site. The default in this function is to use 2kb upstream and 200bp downstream of the start site. Let's keep this.

Now we compute which promoters have a H3K4me3 peak in them:

```
> ov <- findOverlaps(promoters, gr1)
> ov
Hits object with 46022 hits and 0 metadata columns:
          queryHits subjectHits
          <integer>   <integer>
      [1]         2         387
      [2]         3        2523
      [3]         4         774
      [4]         5         774
      [5]         6        1114
      ...       ...         ...
  [46018]     47457       47491
  [46019]     47459       47493
  [46020]     47459       47494
  [46021]     47460       47493
  [46022]     47460       47494
  -------
  queryLength: 50066 / subjectLength: 74470
```

Let us compute how many percent of the peaks are in a promoter

```
> length(unique(queryHits(ov))) / length(gr1)
[1] 0.3333691
```

and how many percent of promoters have a peak in them

```
> length(unique(subjectHits(ov))) / length(promoters)
[1] 0.440878
```

My rule of thumb is that any cell type has at most 50% of genes expressed, which fits well with these numbers. We also see that there are many H3K4me3 peaks which do not lie in a genic promoter. This is actually expected.

Now, the overlap is non-empty, but is it interesting or "significant". Answering this question thoroughly requires thinking deeply about background distributions etc. We will avoid a careful statistical approach to this question, and will instead do a few back-of-the-envelope calculations to get a feel for it.

First we notice that both promoters and peaks are small compared to the size of the human genome:

```
> sum(width(reduce(gr1))) / 10^6
[1] 11.19044
> sum(width(reduce(promoters))) / 10^6
[1] 64.3005
```

(result is in megabases; the human genome is 3000 megabases). Just this fact alone should convince you that the overlap is highly unlikely happen purely by chance. Let us calculate the overlap in megabases:

```
> sum(width(intersect(gr1, promoters))) / 10^6
[1] 0
```

That's weird; why is the overlap empty when we know (using `findOverlaps`) that there is plenty of overlap? Turns out the answer is strand. We need to ignore the strand when we do this calculation:

```
> sum(width(intersect(gr1, promoters, ignore.strand = TRUE))) / 10^6
[1] 3.019608
```

This brings us back to the question of the size of the promoter set above; this is also affected by strand when there is a promoter on each strand which overlaps. Contrast

```
> sum(width(reduce(promoters))) / 10^6
[1] 64.3005
> sum(width(reduce(promoters, ignore.strand = TRUE))) / 10^6
[1] 62.27957
```

Strand can be troublesome!

Let us compute a small 2x2 matrix for which bases are in promoters and/or peaks:

```
> prom <- reduce(promoters, ignore.strand = TRUE)
> peaks <- reduce(gr1)
> both <- intersect(prom, peaks)
> only.prom <- setdiff(prom, both)
> only.peaks <- setdiff(peaks, both)
> overlapMat <- matrix(0,, ncol = 2, nrow = 2)
> colnames(overlapMat) <- c("in.peaks", "out.peaks")
> rownames(overlapMat) <- c("in.promoters", "out.promoter")
> overlapMat[1,1] <- sum(width(both))
> overlapMat[1,2] <- sum(width(only.prom))
> overlapMat[2,1] <- sum(width(only.peaks))
```

```
> overlapMat[2,2] <- 3*10^9 - sum(overlapMat)
> round(overlapMat / 10^6, 2)
             in.peaks out.peaks
in.promoters     3.02     59.26
out.promoter     8.17   2929.55
```

Here we have just used the genome size of 3 billion bases. This is not correct. There are many of these bases which have not been sequenced and in addition, there are many bases which cannot be mapped using short reads. This will reduce the genome size.

Nevertheless, let us compute an odds-ratio for this table:

```
> oddsRatio <- overlapMat[1,1] * overlapMat[2,2] / (overlapMat[2,1] * overlapMat\
[1,2])
> oddsRatio
[1] 18.26938
```

This odds-ratio shows an enrichment of peaks in promoters. We can get a feel for how much the genome size (which we use incorrectly) affects our result by using a lower bound on the genome size. Let us say 1.5 billion bases:

```
> overlapMat[2,2] <- 1.5*10^9
> oddsRatio <- overlapMat[1,1] * overlapMat[2,2] / (overlapMat[2,1] * overlapMat\
[1,2])
> oddsRatio
[1] 9.354362
```

The odds-ratio got smaller, but it is still bigger than 1.

Here we basically says that each base can be assigned to peaks or promoters independently, which is definitely false. So there are many, many reasons why this calculation is not the “right” one. Nevertheless, it appears that we have some enrichment, as expected.

11. Biostrings

Watch a video[1] of this chapter.

11.1 Dependencies

This document has the following dependencies:

```
> library(Biostrings)
```

Use the following commands to install these packages in R.

```
> source("http://www.bioconductor.org/biocLite.R")
> biocLite(c("Biostrings"))
```

11.2 Overview

The *Biostrings* package contains classes and functions for representing biological strings such as DNA, RNA and amino acids. In addition the package has functionality for pattern matching (short read alignment) as well as a pairwise alignment function implementing Smith-Waterman local alignments and Needleman-Wunsch global alignments used in classic sequence alignment (see (Durbin et al. 1998) for a description of these algorithms). There are also functions for reading and writing output such as FASTA files.

11.3 Representing sequences

There are two basic types of containers for representing strings. One container represents a single string (say a chromosome or a single short read) and the other container represents a set of strings (say a set of short reads). There are different classes intended to represent different types of sequences such as DNA or RNA sequences.

[1] https://youtu.be/lTXsZ1glvUY

```
> dna1 <- DNAString("ACGT-N")
> dna1
  6-letter "DNAString" instance
seq: ACGT-N
> DNAStringSet("ACG")
  A DNAStringSet instance of length 1
    width seq
[1]     3 ACG
> dna2 <- DNAStringSet(c("ACGT", "GTCA", "GCTA"))
> dna2
  A DNAStringSet instance of length 3
    width seq
[1]     4 ACGT
[2]     4 GTCA
[3]     4 GCTA
```

Note that the alphabet of a `DNAString` is an extended alphabet: `-` (for insertion) and `N` are allowed. In fact, IUPAC codes are allowed (these codes represent different characters, for example the code "M" represents either and "A" or a "C"). A list of IUPAC codes can be obtained by

```
> IUPAC_CODE_MAP
     A      C      G      T      M      R      W      S      Y      K
   "A"    "C"    "G"    "T"   "AC"   "AG"   "AT"   "CG"   "CT"   "GT"
     V      H      D      B      N
 "ACG"  "ACT"  "AGT"  "CGT" "ACGT"
```

Indexing into a `DNAString` retrieves a subsequence (similar to the standard R function `substr`), whereas indexing into a `DNAStringSet` gives you a subset of sequences.

```
> dna1[2:4]
  3-letter "DNAString" instance
seq: CGT
> dna2[2:3]
  A DNAStringSet instance of length 2
    width seq
[1]     4 GTCA
[2]     4 GCTA
```

Note that `[[` allows you to get a single element of a `DNAStringSet` as a `DNAString`. This is very similar to `[` and `[[` for lists.

```
> dna2[[2]]
  4-letter "DNAString" instance
seq: GTCA
```

`DNAStringSet` objects can have names, like ordinary vectors

```
> names(dna2) <- paste0("seq", 1:3)
> dna2
  A DNAStringSet instance of length 3
    width seq                                               names
[1]     4 ACGT                                              seq1
[2]     4 GTCA                                              seq2
[3]     4 GCTA                                              seq3
```

The full set of string classes are

- `DNAString[Set]`: DNA sequences.
- `RNAString[Set]`: RNA sequences.
- `AAString[Set]`: Amino Acids sequences (protein).
- `BString[Set]`: "Big" sequences, using any kind of letter.

In addition you will often see references to `XString[Set]` in the documentation. An `XString[Set]` is basically any of the above classes.

These classes seem very similar to standard `characters()` from base R, but there are important differences. The differences are mostly about efficiencies when you deal with either (a) many sequences or (b) very long strings (think whole chromosomes).

11.4 Basic functionality

Basic character functionality is supported, like

- `length`, `names`.
- `c` and `rev` (reverse the sequence).
- `width`, `nchar` (number of characters in each sequence).
- `==`, `duplicated`, `unique`.
- `as.charcater` or `toString`: converts to a base `character()` vector.
- `sort`, `order`.
- `chartr`: convert some letters into other letters.
- `subseq`, `subseq<-`, `extractAt`, `replaceAt`.
- `replaceLetterAt`.

Examples

```
> width(dna2)
[1] 4 4 4
> sort(dna2)
  A DNAStringSet instance of length 3
    width seq                                               names
[1]     4 ACGT                                              seq1
[2]     4 GCTA                                              seq3
[3]     4 GTCA                                              seq2
> rev(dna2)
  A DNAStringSet instance of length 3
    width seq                                               names
[1]     4 GCTA                                              seq3
[2]     4 GTCA                                              seq2
[3]     4 ACGT                                              seq1
> rev(dna1)
  6-letter "DNAString" instance
seq: N-TGCA
```

Note that `rev` on a `DNAStringSet` just reverse the order of the elements, whereas `rev` on a `DNAString` actually reverse the string.

11.5 Biological functionality

There are also functions which are related to the biological interpretation of the sequences, including

- `reverse`: reverse the sequence.
- `complement`, `reverseComplement`: (reverse) complement the sequence.
- `translate`: translate the DNA or RNA sequence into amino acids.

```
> translate(dna2)
  A AAStringSet instance of length 3
    width seq                                               names
[1]     1 T                                                 seq1
[2]     1 V                                                 seq2
[3]     1 A                                                 seq3
> reverseComplement(dna1)
  6-letter "DNAString" instance
seq: N-ACGT
```

11.6 Counting letters

We very often want to count sequences in various ways. Examples include:

- Compute the GC content of a set of sequences.
- Compute the frequencies of dinucleotides in a set of sequences.
- Compute a position weight matrix from a set of aligned sequences.

There is a rich set of functions for doing this quickly.

- `alphabetFrequency`, `letterFrequency`: Compute the frequency of all characters (`alphabetFrequency`) or only specific letters (`letterFrequency`).
- `dinucleotideFrequency`, `trinucleotideFrequency`, `oligonucleotideFrequeny`: compute frequencies of dinucleotides (2 bases), trinucleotides (3 bases) and oligonucleotides (general number of bases).
- `letterFrequencyInSlidingView`: letter frequencies, but in sliding views along the string.
- `consensusMatrix`: consensus matrix; almost a position weight matrix.

Letâ€™s look at some examples, note how the output expands to a matrix when you use the functions on a `DNAStringSet`:

```
> alphabetFrequency(dna1)
A C G T M R W S Y K V H D B N - + .
1 1 1 1 0 0 0 0 0 0 0 0 0 0 1 1 0 0
> alphabetFrequency(dna2)
     A C G T M R W S Y K V H D B N - + .
[1,] 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0
[2,] 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0
[3,] 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0
> letterFrequency(dna2, "GC")
     G|C
[1,]   2
[2,]   2
[3,]   2
> consensusMatrix(dna2, as.prob = TRUE)
       [,1]      [,2]      [,3]      [,4]
A 0.3333333 0.0000000 0.0000000 0.6666667
C 0.0000000 0.6666667 0.3333333 0.0000000
G 0.6666667 0.0000000 0.3333333 0.0000000
T 0.0000000 0.3333333 0.3333333 0.3333333
```

```
M 0.0000000 0.0000000 0.0000000 0.0000000
R 0.0000000 0.0000000 0.0000000 0.0000000
W 0.0000000 0.0000000 0.0000000 0.0000000
S 0.0000000 0.0000000 0.0000000 0.0000000
Y 0.0000000 0.0000000 0.0000000 0.0000000
K 0.0000000 0.0000000 0.0000000 0.0000000
V 0.0000000 0.0000000 0.0000000 0.0000000
H 0.0000000 0.0000000 0.0000000 0.0000000
D 0.0000000 0.0000000 0.0000000 0.0000000
B 0.0000000 0.0000000 0.0000000 0.0000000
N 0.0000000 0.0000000 0.0000000 0.0000000
- 0.0000000 0.0000000 0.0000000 0.0000000
+ 0.0000000 0.0000000 0.0000000 0.0000000
. 0.0000000 0.0000000 0.0000000 0.0000000
```

(most functions allows the return of probabilities with `as.prob = TRUE`).

11.7 References

Durbin, Richard M, Sean R Eddy, Anders Krogh, Graeme Mitchison, and Sean R Eddy. 1998. *Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids*. Cambridge University Press.

12. BSgenome

Watch a video[1] of this chapter.

12.1 Dependencies

This document has the following dependencies:

```
> library(BSgenome)
> library(BSgenome.Scerevisiae.UCSC.sacCer2)
```

Use the following commands to install these packages in R.

```
> source("http://www.bioconductor.org/biocLite.R")
> biocLite(c("BSgenome", "BSgenome.Scerevisiae.UCSC.sacCer2"))
```

12.2 Overview

The *BSgenome* package contains infrastructure for representing genome sequences in Bioconductor.

12.3 Genomes

The *BSgenome* package provides support for genomes. In Bioconductor, we have special classes for genomes, because the chromosomes can get really big. For example, the human genome takes up several GB of memory.

The `available.genomes()` function lists which genomes are currently available from from Bioconductor (it is possible to make your own genome package). Note that there are several so-called "masked" genomes, where some parts of the genome are masked. We will avoid this subject for now. We can `grep()` for known organisms.

[1] https://youtu.be/cNJ2wbObRl8

```
> allgenomes <- available.genomes()
> grep("Hsapiens", allgenomes, value = TRUE)
 [1] "BSgenome.Hsapiens.1000genomes.hs37d5"
 [2] "BSgenome.Hsapiens.NCBI.GRCh38"
 [3] "BSgenome.Hsapiens.UCSC.hg17"
 [4] "BSgenome.Hsapiens.UCSC.hg17.masked"
 [5] "BSgenome.Hsapiens.UCSC.hg18"
 [6] "BSgenome.Hsapiens.UCSC.hg18.masked"
 [7] "BSgenome.Hsapiens.UCSC.hg19"
 [8] "BSgenome.Hsapiens.UCSC.hg19.masked"
 [9] "BSgenome.Hsapiens.UCSC.hg38"
[10] "BSgenome.Hsapiens.UCSC.hg38.masked"
> grep("Scerevisiae", allgenomes, value = TRUE)
[1] "BSgenome.Scerevisiae.UCSC.sacCer1" "BSgenome.Scerevisiae.UCSC.sacCer2"
[3] "BSgenome.Scerevisiae.UCSC.sacCer3"
```

Let us load the latest yeast genome

```
> library(BSgenome.Scerevisiae.UCSC.sacCer2)
> Scerevisiae
Yeast genome:
# organism: Saccharomyces cerevisiae (Yeast)
# provider: UCSC
# provider version: sacCer2
# release date: June 2008
# release name: SGD June 2008 sequence
# 18 sequences:
#   chrI    chrII   chrIII  chrIV   chrV    chrVI   chrVII  chrVIII chrIX
#   chrX    chrXI   chrXII  chrXIII chrXIV  chrXV   chrXVI  chrM    2micron
# (use 'seqnames()' to see all the sequence names, use the '$' or '[['
# operator to access a given sequence)
```

A *BSgenome* package contains a single object which is the second component of the name. At first, nothing is loaded into memory, which makes it very fast. You can get the length and names of the chromosomes without actually loading them.

```
> seqlengths(Scerevisiae)
   chrI   chrII  chrIII   chrIV    chrV   chrVI  chrVII chrVIII   chrIX
 230208  813178  316617 1531919  576869  270148 1090947  562643  439885
   chrX   chrXI  chrXII chrXIII  chrXIV   chrXV  chrXVI    chrM 2micron
 745742  666454 1078175  924429  784333 1091289  948062   85779    6318
> seqnames(Scerevisiae)
 [1] "chrI"    "chrII"   "chrIII"  "chrIV"   "chrV"    "chrVI"   "chrVII"
 [8] "chrVIII" "chrIX"   "chrX"    "chrXI"   "chrXII"  "chrXIII" "chrXIV"
[15] "chrXV"   "chrXVI"  "chrM"    "2micron"
```

We load a chromosome by using the `[[` or `$` operators:

```
> Scerevisiae$chrI
  230208-letter "DNAString" instance
seq: CCACACCACACCCACACACCCACACACCACACCA...GTGTGGGTGTGGTGTGGGTGTGGTGTGTGTGGG
```

We can now do things like compute the GC content of the first chromosome

```
> letterFrequency(Scerevisiae$chrI, "CG", as.prob = TRUE)
      C|G
0.3927361
```

To iterate over chromosomes seems straightforward with `lapply`. However, this function may end up using a lot of memory because the entire genome is loaded. Instead there is the `bsapply` function which handles loading and unloading of different chromosomes. The interface to `bsapply` is weird at first; you set up a `BSparams` object which contains which function you are using and which genome you are using it on (and a bit more information). This paradigm is being used in other packages these days, for example *BiocParallel*. An example will make this clear:

```
> param <- new("BSParams", X = Scerevisiae, FUN = letterFrequency)
> head(bsapply(param, letters = "GC"))
$chrI
  G|C
90411

$chrII
   G|C
311807

$chrIII
   G|C
```

```
121998

$chrIV
   G|C
580699

$chrV
   G|C
222141

$chrVI
   G|C
104636
```

note how the additional argument `letters` to the `letterFrequency` function is given as an argument to `bsapply`, not to the `BSParams` object. This gives us a list; you can simplify the output (like the difference between `lapply` and `sapply`) by

```
> param <- new("BSParams", X = Scerevisiae, FUN = letterFrequency, simplify = TR\
UE)
> bsapply(param, letters = "GC")
   chrI.G|C   chrII.G|C  chrIII.G|C   chrIV.G|C    chrV.G|C   chrVI.G|C
      90411      311807      121998      580699      222141      104636
 chrVII.G|C chrVIII.G|C   chrIX.G|C    chrX.G|C   chrXI.G|C  chrXII.G|C
     415227      216586      171122      286167      253728      414843
chrXIII.G|C  chrXIV.G|C   chrXV.G|C  chrXVI.G|C    chrM.G|C 2micron.G|C
     353167      303042      416443      360871       14676        2463
```

Note how the mitochondria chromosome is very different. To conclude, the GC percentage of the genome is

```
> sum(bsapply(param, letters = "GC")) / sum(seqlengths(Scerevisiae))
[1] 0.3814872
```

13. Biostrings - Matching

Watch a video[1] of this chapter.

13.1 Dependencies

This document has the following dependencies:

```
> library(Biostrings)
> library(BSgenome)
> library(BSgenome.Scerevisiae.UCSC.sacCer2)
```

Use the following commands to install these packages in R.

```
> source("http://www.bioconductor.org/biocLite.R")
> biocLite(c("Biostrings", "BSgenome",
+            "BSgenome.Scerevisiae.UCSC.sacCer2", "AnnotationHub"))
```

13.2 Overview

We continue our treatment of *Biostrings* and *BSgenome*, focusing on searching the genome.

13.3 Pattern matching

We often want to find patterns in (long) sequences. *Biostrings* have a number of functions for doing so

- `matchPattern` and `vmatchPattern`: match a single sequence against one sequence (`matchPattern`) or more than one (`vmatchPattern`) sequences.
- `matchPDict` and `vmatchPDict`: match a (possibly large) set of sequences against one sequence (`matchPDict`) or more than one (`vmatchPDict`) sequences.

[1] https://youtu.be/wFfaF4M8sqM

These functions allows a small set of mismatches and some small indels. The `Dict` term is used because the function builds a "dictionary" over the sequences.

There are also functions with similar naming using `count` instead of `match` (eg. `countPatterns`). These functions returns the number of matches instead of precise information about where the matches occur.

In many ways, these functions are similar to using short read aligners like Bowtie. But these functions are designed to be comprehensive (return all matches satisfying certain criteria). Having this functionality available in Bioconductor can sometimes be very useful.

```
> dnaseq <- DNAString("ACGTACGT")
> matchPattern(dnaseq, Scerevisiae$chrI)
  Views on a 230208-letter DNAString subject
subject: CCACACCACACCCACACACCCACACACCACAC...GTGGGTGTGGTGTGGGTGTGGTGTGTGTGGG
views:
    start   end width
[1] 57932 57939     8 [ACGTACGT]
> countPattern(dnaseq, Scerevisiae$chrI)
[1] 1
> vmatchPattern(dnaseq, Scerevisiae)
GRanges object with 170 ranges and 0 metadata columns:
        seqnames           ranges strand
           <Rle>        <IRanges>  <Rle>
    [1]     chrI [ 57932,  57939]      +
    [2]     chrI [ 57932,  57939]      -
    [3]    chrII [ 49581,  49588]      +
    [4]    chrII [411291, 411298]      +
    [5]    chrII [491129, 491136]      +
    ...      ...              ...    ...
  [166]   chrXVI [195477, 195484]      -
  [167]   chrXVI [683620, 683627]      -
  [168]   chrXVI [837296, 837303]      -
  [169]   chrXVI [906938, 906945]      -
  [170]   chrXVI [943045, 943052]      -
  -------
  seqinfo: 18 sequences from an unspecified genome
> head(vcountPattern(dnaseq, Scerevisiae))
  seqname strand count
1    chrI      +     1
2    chrI      -     1
3   chrII      +     4
4   chrII      -     4
```

```
5  chrIII      +     3
6  chrIII      -     3
```

See how we use `vmatchPattern` to examine across all chromosomes.

First, note how the return object of `vmatchPattern` is a `GRanges` given the exact information of where the string matches. Note sequence we search for is its own reverse complement, so we get hits on both strands (which makes sense). Obviously, not all sequences are like this

```
> dnaseq == reverseComplement(dnaseq)
[1] TRUE
```

Second, note how the return object of `matchPattern` looks like an `IRanges` but is really something called a `Views` (see another session).

13.4 Specialized alignments

There are a number of other, specialized, alignment functions in *Biostrings*. They include

- `matchPWM`: a position weight matrix is a common way to represent for example a transcription factor binding motif (think sequence logos). This function allows you to search for such motifs in the genome.
- `pairwiseAlignment`: This function implements pairwise alignments using dynamic programming; providing an interface to both the Smith-Waterman local alignment problem and the Needleman-Wunsch global alignment problems, see a thorough description in (Durbin et al. 1998).
- `trimLRpattern` (trim left-right pattern): Takes a set of sequences and looks for whether they start or end with a given (other sequence), for example a sequencing adapter. Used for trimming reads based on adapter sequences.

For now, we will avoid further discussion of these functions.

One note: `pairwiseAlignment` allows you to do pairwise alignments of millions of short reads against a single sequence, for example a gene or a transposable element. Few people use these algorithms for short read data, because the algorithms scale badly with the length of the sequence (ie. the genome), but they work fine for millions of reads as long as the reference sequence is short. In my opinion this approach might be very fruitful if you are particular interested in high-quality alignments to a specific small gene or region of the genome.

13.5 References

Durbin, Richard M, Sean R Eddy, Anders Krogh, Graeme Mitchison, and Sean R Eddy. 1998. *Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids*. Cambridge University Press.

14. BSgenome - Views

Watch a video[1] of this chapter.

14.1 Dependencies

This document has the following dependencies:

```
> library(BSgenome)
> library(BSgenome.Scerevisiae.UCSC.sacCer2)
> library(AnnotationHub)
```

Use the following commands to install these packages in R.

```
> source("http://www.bioconductor.org/biocLite.R")
> biocLite(c("BSgenome",
+            "BSgenome.Scerevisiae.UCSC.sacCer2", "AnnotationHub"))
```

14.2 Overview

We continue our treatment of *Biostrings* and *BSgenome*

14.3 Views

`Views` are used when you have a single big object (think chromosome or other massive dataset) and you need to deal with (many) subsets of this object. `Views` are not restricted to genome sequences; we will discuss `Views` on other types of objects in a different session.

Technically, a `Views` is like an `IRanges` couple with a pointer to the massive object. The `IRanges` contains the indexes. Let's look at `matchPattern` again:

[1] https://youtu.be/fPBxqAPXQCE

```
> library(BSgenome.Scerevisiae.UCSC.sacCer2)
> dnaseq <- DNAString("ACGTACGT")
> vi <- matchPattern(dnaseq, Scerevisiae$chrI)
> vi
  Views on a 230208-letter DNAString subject
subject: CCACACCACACCCACACACCCACACACCACAC...GTGGGTGTGGTGTGGGTGTGGTGTGTGTGGG
views:
    start   end width
[1] 57932 57939     8 [ACGTACGT]
```

We can get the `IRanges` component by

```
> ranges(vi)
IRanges object with 1 range and 0 metadata columns:
          start       end     width
      <integer> <integer> <integer>
  [1]     57932     57939         8
```

The `IRanges` gives us indexes into the underlying subject (here chromosome I). To be clear, compare these two:

```
> vi
  Views on a 230208-letter DNAString subject
subject: CCACACCACACCCACACACCCACACACCACAC...GTGGGTGTGGTGTGGGTGTGGTGTGTGTGGG
views:
    start   end width
[1] 57932 57939     8 [ACGTACGT]
> Scerevisiae$chrI[ start(vi):end(vi) ]
  8-letter "DNAString" instance
seq: ACGTACGT
```

The `Views` object also look a bit like a `DNAStringSet`; we can do things like

```
> alphabetFrequency(vi)
     A C G T M R W S Y K V H D B N - + .
[1,] 2 2 2 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

The advantage of `Views` is that they don't duplicate the sequence information from the subject; all they keep track of are indexes into the subject (stored as `IRanges`). This makes it very (1) fast, (2) low-memory and makes it possible to do things like

```
> shift(vi, 10)
  Views on a 230208-letter DNAString subject
subject: CCACACCACACCCACACACCCACACACCACAC...GTGGGTGTGGTGTGGGTGTGGTGTGTGTGGG
views:
    start   end width
[1] 57942 57949     8 [AAGCTTTG]
```

where we now get the sequence 10 bases next to the original match. This could not be done if all we had were the bases of the original subsequence.

`Views` are especially powerful when there are many of them. A usecase I often have are the set of all exons (or promoters) of all genes in the genome. You can use `GRanges` as `Views` as well. Lets look at the hits from `vmatchPattern`.

```
> gr <- vmatchPattern(dnaseq, Scerevisiae)
> vi2 <- Views(Scerevisiae, gr)
```

Now, let us do something with this. First let us get gene coordinates from *AnnotationHub*.

```
> ahub <- AnnotationHub()
> qh <- query(ahub, c("sacCer2", "genes"))
> qh
AnnotationHub with 2 records
# snapshotDate(): 2016-05-12
# $dataprovider: UCSC
# $species: Saccharomyces cerevisiae
# $rdataclass: GRanges
# additional mcols(): taxonomyid, genome, description, tags,
#   sourceurl, sourcetype
# retrieve records with, e.g., 'object[["AH7048"]]'

           title
  AH7048 | SGD Genes
  AH7049 | Ensembl Genes
> genes <- qh[[which(qh$title == "SGD Genes")]]
> genes
GRanges object with 6717 ranges and 5 metadata columns:
         seqnames           ranges strand |        name     score
            <Rle>        <IRanges>  <Rle> | <character> <numeric>
     [1]     chrI [130802, 131986]      + |     YAL012W         0
     [2]     chrI [   335,    649]      + |     YAL069W         0
     [3]     chrI [   538,    792]      + |   YAL068W-A         0
```

```
    [4]     chrI [  1807,   2169]      - |     YAL068C         0
    [5]     chrI [  2480,   2707]      + |   YAL067W-A         0
    ...      ...             ...    ... .         ...       ...
[6713]  chrXIII [923492, 923800]      - |     YMR326C         0
[6714]  2micron [   252,   1523]      + |      R0010W         0
[6715]  2micron [  1887,   3008]      - |      R0020C         0
[6716]  2micron [  3271,   3816]      + |      R0030W         0
[6717]  2micron [  5308,   6198]      - |      R0040C         0
          itemRgb            thick        blocks
      <character>        <IRanges> <IRangesList>
   [1]        <NA> [130802, 131986]     [1, 1185]
   [2]        <NA> [   335,    649]      [1, 315]
   [3]        <NA> [   538,    792]      [1, 255]
   [4]        <NA> [  1807,   2169]      [1, 363]
   [5]        <NA> [  2480,   2707]      [1, 228]
   ...         ...             ...          ...
[6713]        <NA> [923492, 923800]      [1, 309]
[6714]        <NA> [   252,   1523]     [1, 1272]
[6715]        <NA> [  1887,   3008]     [1, 1122]
[6716]        <NA> [  3271,   3816]      [1, 546]
[6717]        <NA> [  5308,   6198]      [1, 891]
-------
seqinfo: 18 sequences (2 circular) from sacCer2 genome
```

Let us compute the GC content of all promoters in the yeast genome.

```
> prom <- promoters(genes)
> head(prom, n = 3)
GRanges object with 3 ranges and 5 metadata columns:
     seqnames           ranges strand |        name     score     itemRgb
        <Rle>        <IRanges>  <Rle> | <character> <numeric> <character>
  [1]     chrI [128802, 131001]      + |     YAL012W         0        <NA>
  [2]     chrI [ -1665,    534]      + |     YAL069W         0        <NA>
  [3]     chrI [ -1462,    737]      + |   YAL068W-A         0        <NA>
              thick        blocks
          <IRanges> <IRangesList>
  [1] [130802, 131986]     [1, 1185]
  [2] [   335,    649]      [1, 315]
  [3] [   538,    792]      [1, 255]
  -------
  seqinfo: 18 sequences (2 circular) from sacCer2 genome
```

We get a `warning` that some of these promoters are out-of-band (see the the second and third element in the `prom` object; they have negative values for their ranges). We clean it up and continue

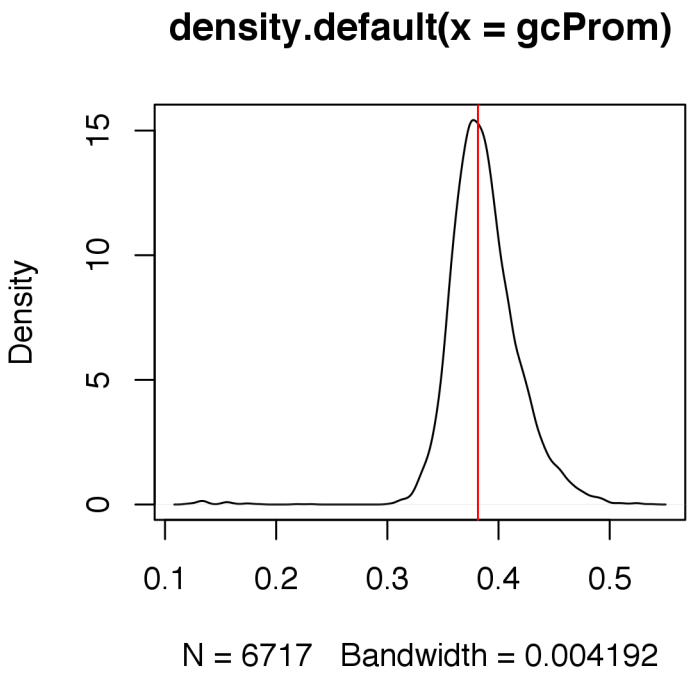
```
> prom <- trim(prom)
> promViews <- Views(Scerevisiae, prom)
> gcProm <- letterFrequency(promViews, "GC", as.prob = TRUE)
> head(gcProm)
          G|C
[1,] 0.4668182
[2,] 0.4868914
[3,] 0.4572592
[4,] 0.3731818
[5,] 0.3859091
[6,] 0.3309091
```

In the previous *Biostrings* session we computed the GC content of the yeast genome. Let us do it again, briefly

```
> params <- new("BSParams", X = Scerevisiae, FUN = letterFrequency, simplify = T\
RUE)
> gccontent <- bsapply(params, letters = "GC")
> gcPercentage <- sum(gccontent) / sum(seqlengths(Scerevisiae))
> gcPercentage
[1] 0.3814872
```

Let us compare this genome percentage to the distribution of GC content for promoters

```
> plot(density(gcProm))
> abline(v = gcPercentage, col = "red")
```



The distribution of GC content of promoters.

At first glance, the GC content of the promoters is not very different from the genome-wide GC content (perhaps shifted a bit to the right).

15. GenomicRanges - Rle

Watch a video[1] of this chapter.

15.1 Dependencies

This document has the following dependencies:

```
> library(GenomicRanges)
```

Use the following commands to install these packages in R.

```
> source("http://www.bioconductor.org/biocLite.R")
> biocLite(c("GenomicRanges"))
```

15.2 Overview

In this session we will discuss a data representation class called `Rle` (run length encoding). This class is great for representation genome-wide sequence coverage.

15.3 Coverage

In high-throughput sequencing, coverage is the number of reads overlapping each base. In other words, it associates a number (the number of reads) to every base in the genome.

This is a fundamental quantity for many high-throughout sequencing analyses. For variant calling (DNA sequencing) it tells you how much power (information) you have to call a variant at a given location. For ChIP sequencing it is the primary signal; areas with high coverage are thought to be enriched for a given protein.

A file format which is often used to represent coverage data is `Wig` or the modern version `BigWig`.

[1] https://youtu.be/w7ZPnO-jB9o

15.4 Rle

An `Rle` (run-length-encoded) vector is a specific representation of a vector. The *IRanges* package implements support for this class. Watch out: there is also a base R class called `rle` which has much less functionality.

The run-length-encoded representation of a vector, represents the vector as a set of distinct runs with their own value. Let us take an example

```
> rl <- Rle(c(1,1,1,1,2,2,3,3,2,2))
> rl
numeric-Rle of length 10 with 4 runs
  Lengths: 4 2 2 2
  Values : 1 2 3 2
> runLength(rl)
[1] 4 2 2 2
> runValue(rl)
[1] 1 2 3 2
> as.numeric(rl)
 [1] 1 1 1 1 2 2 3 3 2 2
```

Note the accessor functions `runLength()` and `runValue()`.

This is a very efficient representation if

- the vector is very long
- there are a lot of consecutive elements with the same value

This is especially useful for genomic data which is either piecewise constant, or where most of the genome is not covered (eg. RNA sequencing in mammals).

In many ways `Rles` function as normal vectors, you can do arithmetic with them, transform them etc. using standard R functions like + and `log2`.

There are also `RleList` which is a list of `Rles`. This class is used to represent a genome wide coverage track where each element of the list is a different chromosome.

15.5 Useful functions for Rle

A standard usecase is that you have a number of regions (say `IRanges`) and you want to do something to your `Rle` over each of these regions. Enter `aggregate()`.

```
> ir <- IRanges(start = c(2,6), width = 2)
> aggregate(rl, ir, FUN = mean)
[1] 1.0 2.5
```

It is also possible to covert an `IRanges` to a `Rle` by the `coverage()` function. This counts, for each integer, how many ranges overlap the integer.

```
> ir <- IRanges(start = 1:10, width = 3)
> rl <- coverage(ir)
> rl
integer-Rle of length 12 with 5 runs
  Lengths: 1 1 8 1 1
  Values : 1 2 3 2 1
```

You can select high coverage regions by the `slice()` function:

```
> slice(rl, 2)
Views on a 12-length Rle subject

views:
    start end width
[1]     2  11    10 [2 3 3 3 3 3 3 3 3 2]
```

This outputs a `Views` object, see next section.

15.6 Views and Rles

In the sessions on the *Biostrings* package we learned about `Views` on genomes. `Views` can also be instantiated on `Rles`.

```
> vi <- Views(rl, start = c(3,7), width = 3)
> vi
Views on a 12-length Rle subject

views:
    start end width
[1]     3   5     3 [3 3 3]
[2]     7   9     3 [3 3 3]
```

with `Views` you can now (again) apply functions:

```
> mean(vi)
[1] 3 3
```

This is very similar to using `aggregate()` described above.

15.7 RleList

An `RleList` is simply a list of `Rle`. It is similar to a `GRangesList` in concept.

15.8 Rles and GRanges

`Rle`â€™s can also be constructed from `GRanges`.

This often involves `RleList` where each element of the list is a chromosome. Surprisingly, we do not yet have an `RleList` type structure which also contains information about say the length of the different chromosomes.

Let us see some examples

```
> gr <- GRanges(seqnames = "chr1", ranges = IRanges(start = 1:10, width = 3))
> rl <- coverage(gr)
> rl
RleList of length 1
$chr1
integer-Rle of length 12 with 5 runs
  Lengths: 1 1 8 1 1
  Values : 1 2 3 2 1
```

Let us consider a `GRanges` which we want to use to index into the `Rle`:

```
> grView <- GRanges("chr1", ranges = IRanges(start = c(2,5), end = c(3,6)))
```

We can subset directly into the `Rle` by

```
> rl[grView]
RleList of length 2
$chr1
integer-Rle of length 2 with 2 runs
  Lengths: 1 1
  Values : 2 3

$chr1
integer-Rle of length 2 with 1 run
  Lengths: 2
  Values : 3
```

But using `Views` on such an object exposes some missing functionality

```
> grView <- GRanges("chr1", ranges = IRanges(start = 2, end = 7))
> vi <- Views(rl, grView)
Error in RleViewsList(rleList = subject, rangesList = start): 'rangesList' must \
be a RangesList object
```

We get an error, mentioning some object called a `RangesList`. This type of object is similar to a `GRanges` and could be considered succeeded by the later class. We sometimes see instances of this popping around.

```
> vi <- Views(rl, as(grView, "RangesList"))
> vi
RleViewsList of length 1
names(1): chr1
> vi[[1]]
Views on a 12-length Rle subject

views:
    start end width
[1]     2   7     6 [2 3 3 3 3 3]
```

15.9 Biology Usecase

Suppose we want to compute the average coverage of bases belonging to (known) exons.

Input objects are
reads: a `GRanges`.
exons: a `GRanges`.

pseudocode:

```
> bases <- reduce(exons)
> nBases <- sum(width(bases))
> nCoverage <- sum(Views(coverage(reads), bases))
> nCoverage / nBases
```

(watch out for strand)

16. GenomicRanges - Lists

Watch a video[1] of this chapter.

16.1 Dependencies

This document has the following dependencies:

```
> library(GenomicRanges)
```

Use the following commands to install these packages in R.

```
> source("http://www.bioconductor.org/biocLite.R")
> biocLite(c("GenomicRanges"))
```

16.2 Overview

In this session we will discuss `GRangesList` which is a list of `GRanges` (whoa; blinded by the insight here!).

16.3 Why

The *IRanges* and *GenomicRanges* packages introduced a number of classes Iâ€™ll call `XXList`; an example is `GRangesList`.

These looks like standard `lists` from base R, but they require that every element of the list is of the same class. This is convenient from a data structure perspective; we know exactly what is in the list.

But things are also happening behind the scenes. These types of lists often have additional compression build into them. Because of this, it is best to use specific methods/functions on them, as opposed to the standard toolbox of `sapply/lapply` that we use for normal lists. This will be clearer below.

An important usecase specifically for `GRangesList` is the representation of a set of **transcripts**. Each transcript is an element in the list and the **exons** of the transcript is represented as a `GRanges`.

16.4 GrangesList

Let us make a `GRangesList`:

[1] https://youtu.be/Ba3_nX2L1gI

```
> gr1 <- GRanges(seqnames = "chr1", ranges = IRanges(start = 1:4, width = 3))
> gr2 <- GRanges(seqnames = "chr2", ranges = IRanges(start = 1:4, width = 3))
> gL <- GRangesList(gr1 = gr1, gr2 = gr2)
> gL
GRangesList object of length 2:
$gr1
GRanges object with 4 ranges and 0 metadata columns:
      seqnames    ranges strand
         <Rle> <IRanges>  <Rle>
  [1]     chr1    [1, 3]      *
  [2]     chr1    [2, 4]      *
  [3]     chr1    [3, 5]      *
  [4]     chr1    [4, 6]      *

$gr2
GRanges object with 4 ranges and 0 metadata columns:
      seqnames ranges strand
  [1]     chr2 [1, 3]      *
  [2]     chr2 [2, 4]      *
  [3]     chr2 [3, 5]      *
  [4]     chr2 [4, 6]      *

seqinfo: 2 sequences from an unspecified genome; no seqlengths
```

A number of standard `GRanges` functions work, but returns (for example) `IntegerLists`

```
> start(gL)
IntegerList of length 2
[["gr1"]] 1 2 3 4
[["gr2"]] 1 2 3 4
> seqnames(gL)
RleList of length 2
$gr1
factor-Rle of length 4 with 1 run
  Lengths:    4
  Values : chr1
Levels(2): chr1 chr2

$gr2
factor-Rle of length 4 with 1 run
  Lengths:    4
  Values : chr2
Levels(2): chr1 chr2
```

I very often want to get the length of each of the elements. Surprisingly it is very slow to get this using `sapply(gL, length)` (or at least it used to be very slow). There is a dedicated function for this:

```
> elementLengths(gL)
gr1 gr2
  4   4
```

We have a new `XXapply` function with the fancy name `endoapply`. This is used when you want to apply a function which maps a `GRanges` into a `GRanges`, say a `shift` or `resize`.

```
> shift(gL, 10)
GRangesList object of length 2:
$gr1
GRanges object with 4 ranges and 0 metadata columns:
      seqnames    ranges strand
         <Rle> <IRanges>  <Rle>
  [1]     chr1  [11, 13]      *
  [2]     chr1  [12, 14]      *
  [3]     chr1  [13, 15]      *
  [4]     chr1  [14, 16]      *

$gr2
GRanges object with 4 ranges and 0 metadata columns:
      seqnames   ranges strand
  [1]     chr2 [11, 13]      *
  [2]     chr2 [12, 14]      *
  [3]     chr2 [13, 15]      *
  [4]     chr2 [14, 16]      *

seqinfo: 2 sequences from an unspecified genome; no seqlengths
```

`findOverlaps` works slightly different. For `GRangesLists`, we think of each element is a union of ranges. So we get an overlap if any range overlaps. Lets us see

```
> findOverlaps(gL, gr2)
Hits object with 4 hits and 0 metadata columns:
      queryHits subjectHits
      <integer>   <integer>
  [1]         2           1
  [2]         2           2
  [3]         2           3
  [4]         2           4
  -------
  queryLength: 2 / subjectLength: 4
```

Note how the `queryLength` is 2 and not 20. What we know from the first row of this output is that some range in `gL[[2]]` overlaps the range `gr[1]`.

This is actually a feature if we think of the `GRangesList` as a set of transcript, where each `GRanges` gives you the exon of the transcript. With this interpretation, `findOverlaps` tells you whether or not the **transcript** overlaps some region of interest, and this is true if any of the **exons** of the transcript overlaps the region.

16.5 Other Lists

There are many other types of `XXList`, including

- `RleList`
- `IRangesList`
- `IntegerList`
- `CharacterList`
- `LogicalList`

and many others.

17. GenomicFeatures

Watch a video[1] of this chapter.

17.1 Dependencies

This document has the following dependencies:

```
> library(GenomicFeatures)
> library(TxDb.Hsapiens.UCSC.hg19.knownGene)
```

Use the following commands to install these packages in R.

```
> source("http://www.bioconductor.org/biocLite.R")
> biocLite(c("GenomicFeatures", "TxDb.Hsapiens.UCSC.hg19.knownGene"))
```

17.2 Overview

The *GenomicFeatures* package contains functionality for so-called transcript database or *TxDb* objects. These objects contains a coherent interface to transcripts. Transcripts are complicated because higher organisms usually have many different transcripts for each gene locus.

17.3 Examples

We will show the `TxDb` functionality by examining a database of human transcripts. Unlike genomes in Bioconductor, there is no shorthand object; we reassign the long name to a shorter for convenience:

[1] https://youtu.be/9sAhB4Bs43k

```
> library(TxDb.Hsapiens.UCSC.hg19.knownGene)
> txdb <- TxDb.Hsapiens.UCSC.hg19.knownGene
> txdb
TxDb object:
# Db type: TxDb
# Supporting package: GenomicFeatures
# Data source: UCSC
# Genome: hg19
# Organism: Homo sapiens
# Taxonomy ID: 9606
# UCSC Table: knownGene
# Resource URL: http://genome.ucsc.edu/
# Type of Gene ID: Entrez Gene ID
# Full dataset: yes
# miRBase build ID: GRCh37
# transcript_nrow: 82960
# exon_nrow: 289969
# cds_nrow: 237533
# Db created by: GenomicFeatures package from Bioconductor
# Creation time: 2015-10-07 18:11:28 +0000 (Wed, 07 Oct 2015)
# GenomicFeatures version at creation time: 1.21.30
# RSQLite version at creation time: 1.0.0
# DBSCHEMAVERSION: 1.1
```

A `TxDb` object is really an interface to a `SQLite` database. You can query the database using a number of tools detailed in the package vignette, but usually you use convenience functions to extract the relevant information.

Extract basic quantities

- `genes()`
- `transcripts()`
- `cds()`
- `exons()`
- `microRNAs()`
- `tRNAs()`
- `promoters()`

Extract quantities and group

- `transcriptsBy(by = c("gene", "exon", "cds"))`
- `cdsBy(by = c("tx", "gene"))`

- `exonsBy(by = c("tx", "gene"))`
- `intronsByTranscript()`
- `fiveUTRsByTranscript()`
- `threeUTRsByTranscript()`

(Note: there are grouping functions without the non-grouping function and vice versa; there is for example no `introns()` function.

Other functions

- `transcriptLengths()` (optionally include CDS length etc).
- `XXByOverlaps()` (select features based on overlaps with `XX` being `transcript`, `cds` or `exon`).

Mapping between genome and transcript coordinates

- `extractTranscriptSeqs()` (getting RNA sequencing of the transcripts).

17.4 Caution: Terminology

The `TxDb` object approach is powerful but it suffers (in my opinion) from a lack of clearly defined terminology. Even worse, the meaning of terminology changes depending on the function. For example `transcript` is sometimes used to refer to un-spliced transcripts (pre-mRNA) and sometimes to splices transcripts.

17.5 Gene, exons and transcripts

Let us start by examining genes, exons and transcripts. Let us focus on a single gene on chr1: DDX11L1.

```
> gr <- GRanges(seqnames = "chr1", strand = "+", ranges = IRanges(start = 11874,\
 end = 14409))
> subsetByOverlaps(genes(txdb), gr)
GRanges object with 1 range and 1 metadata column:
            seqnames          ranges strand |     gene_id
               <Rle>       <IRanges>  <Rle> | <character>
  100287102     chr1 [11874, 14409]      + |   100287102
  -------
  seqinfo: 93 sequences (1 circular) from hg19 genome
> subsetByOverlaps(genes(txdb), gr, ignore.strand = TRUE)
GRanges object with 2 ranges and 1 metadata column:
```

```
           seqnames         ranges strand |     gene_id
              <Rle>      <IRanges>  <Rle> | <character>
  100287102     chr1 [11874, 14409]      + |   100287102
     653635     chr1 [14362, 29961]      - |      653635
  -------
  seqinfo: 93 sequences (1 circular) from hg19 genome
```

The genes() output contains a single gene with these coordinates, overlapping another gene on the opposite strand. Note that the gene is represented as a single range; so this output tells us nothing about exons and splicing. There is a single identifier called gene_id. If you look at the output of txdb youâ€™ll see that this is an “Entrex Gene ID”.

```
> subsetByOverlaps(transcripts(txdb), gr)
GRanges object with 3 ranges and 2 metadata columns:
      seqnames         ranges strand |     tx_id     tx_name
         <Rle>      <IRanges>  <Rle> | <integer> <character>
  [1]     chr1 [11874, 14409]      + |         1  uc001aaa.3
  [2]     chr1 [11874, 14409]      + |         2  uc010nxq.1
  [3]     chr1 [11874, 14409]      + |         3  uc010nxr.1
  -------
  seqinfo: 93 sequences (1 circular) from hg19 genome
```

The gene has 3 transcripts; again we only have coordinates of the pre-mRNA here. There are 3 different transcript names (tx_name) which are identifiers from UCSC and then we have a TxDb specific transcript id (tx_id) which is an integer. Letâ€™s look at exons:

```
> subsetByOverlaps(exons(txdb), gr)
GRanges object with 6 ranges and 1 metadata column:
      seqnames         ranges strand |   exon_id
         <Rle>      <IRanges>  <Rle> | <integer>
  [1]     chr1 [11874, 12227]      + |         1
  [2]     chr1 [12595, 12721]      + |         2
  [3]     chr1 [12613, 12721]      + |         3
  [4]     chr1 [12646, 12697]      + |         4
  [5]     chr1 [13221, 14409]      + |         5
  [6]     chr1 [13403, 14409]      + |         6
  -------
  seqinfo: 93 sequences (1 circular) from hg19 genome
```

Here we get 6 exons, but no indication of which exons makes up which transcripts. To get this, we can do

```
> subsetByOverlaps(exonsBy(txdb, by = "tx"), gr)
GRangesList object of length 3:
$1
GRanges object with 3 ranges and 3 metadata columns:
      seqnames         ranges strand |   exon_id   exon_name exon_rank
         <Rle>      <IRanges>  <Rle> | <integer> <character> <integer>
  [1]     chr1 [11874, 12227]      + |         1        <NA>         1
  [2]     chr1 [12613, 12721]      + |         3        <NA>         2
  [3]     chr1 [13221, 14409]      + |         5        <NA>         3

$2
GRanges object with 3 ranges and 3 metadata columns:
      seqnames         ranges strand | exon_id exon_name exon_rank
  [1]     chr1 [11874, 12227]      + |       1      <NA>         1
  [2]     chr1 [12595, 12721]      + |       2      <NA>         2
  [3]     chr1 [13403, 14409]      + |       6      <NA>         3

$3
GRanges object with 3 ranges and 3 metadata columns:
      seqnames         ranges strand | exon_id exon_name exon_rank
  [1]     chr1 [11874, 12227]      + |       1      <NA>         1
  [2]     chr1 [12646, 12697]      + |       4      <NA>         2
  [3]     chr1 [13221, 14409]      + |       5      <NA>         3

seqinfo: 93 sequences (1 circular) from hg19 genome
```

Here we now finally see the structure of the three transcripts in the form of a `GRangesList`.

Let us include the coding sequence (CDS). Now, it can be extremely hard to computationally infer the coding sequence from a fully spliced mRNA.

```
> subsetByOverlaps(cds(txdb), gr)
GRanges object with 3 ranges and 1 metadata column:
      seqnames         ranges strand |    cds_id
         <Rle>      <IRanges>  <Rle> | <integer>
  [1]     chr1 [12190, 12227]      + |         1
  [2]     chr1 [12595, 12721]      + |         2
  [3]     chr1 [13403, 13639]      + |         3
  -------
  seqinfo: 93 sequences (1 circular) from hg19 genome
> subsetByOverlaps(cdsBy(txdb, by = "tx"), gr)
GRangesList object of length 1:
$2
```

```
GRanges object with 3 ranges and 3 metadata columns:
      seqnames          ranges strand |    cds_id    cds_name exon_rank
         <Rle>       <IRanges>  <Rle> | <integer> <character> <integer>
  [1]     chr1 [12190, 12227]      + |         1        <NA>         1
  [2]     chr1 [12595, 12721]      + |         2        <NA>         2
  [3]     chr1 [13403, 13639]      + |         3        <NA>         3

seqinfo: 93 sequences (1 circular) from hg19 genome
```

The output of `cds()` is not very useful by itself, since each range is part of a CDS, not the entire cds. We need to know how these ranges together form a CDS, and for that we need `cdsBy(by = "tx")`. We can see that only one of the three transcripts has a CDS by looking at their CDS lengths:

```
> subset(transcriptLengths(txdb, with.cds_len = TRUE), gene_id == "100287102")
  tx_id    tx_name   gene_id nexon tx_len cds_len
1     1 uc001aaa.3 100287102     3   1652       0
2     2 uc010nxq.1 100287102     3   1488     402
3     3 uc010nxr.1 100287102     3   1595       0
```

(here we subset a `data.frame`).

Note: as an example of terminology mixup, consider that the output of `transcripts()` are coordinates for the unspliced transcript, whereas `extractTranscriptSeqs()` is the RNA sequence of the spliced transcripts.

17.6 Other Resources

- The vignette from the GenomicFeatures package[2].

[2] http://bioconductor.org/packages/GenomicFeatures

18. Using the rtracklayer package for data import

Watch a video[1] of this chapter.

18.1 Dependencies

This document has the following dependencies:

```
> library(rtracklayer)
> library(AnnotationHub)
> library(Rsamtools)
```

Use the following commands to install these packages in R.

```
> source("http://www.bioconductor.org/biocLite.R")
> biocLite(c("rtracklayer", "AnnotationHub", "Rsamtools"))
```

18.2 Overview

The *rtracklayer* package interfaces to (UCSC) Genome Browser. It contains functions for importing and exporting data to this browser.

This includes functionality for parsing file formats associated the UCSC Genome Browser such as BED, Wig, BigBed and BigWig.

18.3 The import function

The function to parse data formats is `import()`. This function has a `format` argument taking values such as `BED` or `BigWig`.

Note that there is a help page for the general `import()` function, but there are also file format specific help pages. The easiest way to get to these help pages is to look for `XXFile` with `XX` being the format.

[1] https://youtu.be/BGlXm0kCwf4

```
> ?import
> ?BigWigFile
```

There are often format specific arguments.

18.4 BED files

Most BED files are small and can be read as a single object. The output of `import(format = "BED")` is a `GRanges`.

You can specify `genome` (for example `hg19`) and the function will try to make an effort to populated the `seqinfo` of the `GRanges`.

You can also use the `which` argument to selectively parse a subset of the file which overlaps a `GRanges`. This becomes much more efficient if the file has been tabix-indexed (see below).

18.5 BigWig files

BigWig files typically store whole-genome coverage vectors (or at least whole-genome data). For this reason, the R representation of a BigWig file is usually quite big, so it might be necessary to read it into R in small chunks.

As for BED files, `import(format="BigWig")` supports a `which` argument which is a `GRanges`. It output data type is a `GRanges` per default, but using the `as` agurment you can have `as="Rle"` and a few other options.

The `import(format="BigWig")` does not support a `genome` argument.

18.6 Other file formats

- GFF
- TwoBit
- Wig
- bedGRaph

18.7 Extensive example

Let us start an `AnnotationHub`:

```
> library(AnnotationHub)
> ahub <- AnnotationHub()
> table(ahub$rdataclass)
```

```
     AAStringSet       BigWigFile           biopax        ChainFile
               1            10247                9             1113
      data.frame    ExpressionSet           FaFile          GRanges
              24                1             5122            23577
   Inparanoid8Db           MSnSet         mzRident          mzRpwiz
             268                1                1                1
           OrgDb SQLiteConnection       TwoBitFile          VcfFile
            2164                1             1179                8
```

At this point, you should have seen several of these file formats mentioned. The `GRanges` are usually directly constructed from BED file, and the `seqinfo` information is fully populated:

```
> ahub.gr <- subset(ahub, rdataclass == "GRanges" & species == "Homo sapiens")
> gr <- ahub.gr[[1]]
> gr
GRanges object with 12011 ranges and 6 metadata columns:
          seqnames                ranges strand |        name     score
             <Rle>             <IRanges>  <Rle> | <character> <integer>
      [1]    chr17 [ 77967164,  77967908]      * |           .         0
      [2]    chr14 [ 31698761,  31699349]      * |           .         0
      [3]     chr2 [ 46635926,  46636811]      * |           .         0
      [4]     chr2 [102577636, 102578715]      * |           .         0
      [5]    chr21 [ 46387790,  46388451]      * |           .         0
      ...      ...                   ...    ... .         ...       ...
  [12007]     chr1   [87692002, 87692333]      * |           .         0
  [12008]     chr1   [61388054, 61388436]      * |           .         0
  [12009]    chr22   [37554384, 37554752]      * |           .         0
  [12010]    chr22   [36220362, 36220685]      * |           .         0
  [12011]    chr18   [36821553, 36821903]      * |           .         0
          signalValue    pValue        qValue      peak
            <numeric> <numeric>     <numeric> <integer>
      [1]      17.672   243.175 6.941353e-239       401
      [2]      16.782   240.744 9.362981e-237       346
      [3]      18.917   226.703 6.859981e-223       241
      [4]      20.104   222.061 2.256234e-218       827
      [5]      66.188   213.519 6.287444e-210       371
      ...         ...       ...          ...       ...
  [12007]       9.326     8.537  2.508599e-08       164
```

```
  [12008]       9.326     8.537  2.508808e-08       190
  [12009]       8.041     8.537  2.509017e-08       177
  [12010]       8.041     8.537  2.509225e-08       183
  [12011]       8.607     8.537  2.509434e-08       166
  -------
  seqinfo: 23 sequences from hg19 genome
> seqinfo(gr)
Seqinfo object with 23 sequences from hg19 genome:
  seqnames seqlengths isCircular genome
  chr1      249250621      FALSE   hg19
  chr2      243199373      FALSE   hg19
  chr3      198022430      FALSE   hg19
  chr4      191154276      FALSE   hg19
  chr5      180915260      FALSE   hg19
  ...             ...        ...    ...
  chr19      59128983      FALSE   hg19
  chr20      63025520      FALSE   hg19
  chr21      48129895      FALSE   hg19
  chr22      51304566      FALSE   hg19
  chrX      155270560      FALSE   hg19
```

Perhaps more interesting is the data in form of BigWig files.

```
> ahub.bw <- subset(ahub, rdataclass == "BigWigFile" & species == "Homo sapiens")
> ahub.bw
AnnotationHub with 9932 records
# snapshotDate(): 2016-05-12
# $dataprovider: BroadInstitute
# $species: Homo sapiens
# $rdataclass: BigWigFile
# additional mcols(): taxonomyid, genome, description, tags,
#   sourceurl, sourcetype
# retrieve records with, e.g., 'object[["AH32002"]]'

            title
  AH32002 | E001-H3K4me1.fc.signal.bigwig
  AH32003 | E001-H3K4me3.fc.signal.bigwig
  AH32004 | E001-H3K9ac.fc.signal.bigwig
  AH32005 | E001-H3K9me3.fc.signal.bigwig
  AH32006 | E001-H3K27me3.fc.signal.bigwig
  ...       ...
  AH49540 | E058_mCRF_FractionalMethylation.bigwig
```

```
  AH49541 | E059_mCRF_FractionalMethylation.bigwig
  AH49542 | E061_mCRF_FractionalMethylation.bigwig
  AH49543 | E081_mCRF_FractionalMethylation.bigwig
  AH49544 | E082_mCRF_FractionalMethylation.bigwig
> bw <- ahub.bw[[1]]
> bw
BigWigFile object
resource: /Users/khansen/.AnnotationHub/37442
```

This returns us a file name, ready for use by `import`.

```
> gr1 <- gr[1:3]
> out.gr <- import(bw, which = gr1)
> out.gr
GRanges object with 350 ranges and 1 metadata column:
        seqnames              ranges strand |             score
           <Rle>           <IRanges>  <Rle> |         <numeric>
    [1]    chr14 [31698761, 31698771]      * | 0.911909997463226
    [2]    chr14 [31698772, 31698777]      * |   0.8612300157547
    [3]    chr14 [31698778, 31698817]      * |   1.1482800245285
    [4]    chr14 [31698818, 31698826]      * |  1.08782994747162
    [5]    chr14 [31698827, 31698839]      * |   0.8158900141716
    ...      ...                 ...    ... .               ...
  [346]     chr2 [46636756, 46636762]      * |  1.96861004829407
  [347]     chr2 [46636763, 46636767]      * |  2.21465992927551
  [348]     chr2 [46636768, 46636788]      * |  2.32524991035461
  [349]     chr2 [46636789, 46636800]      * |  2.06692004203796
  [350]     chr2 [46636801, 46636811]      * |  1.80850994586945
  -------
  seqinfo: 25 sequences from an unspecified genome
```

This gives us the content in the form of a `GRanges`. Often, an `Rle` might be appropriate:

```
> out.rle <- import(bw, which = gr1, as = "Rle")
> out.rle
RleList of length 25
$chr1
numeric-Rle of length 249250621 with 1 run
  Lengths: 249250621
  Values :         0

$chr10
numeric-Rle of length 135534747 with 1 run
  Lengths: 135534747
  Values :         0

$chr11
numeric-Rle of length 135006516 with 1 run
  Lengths: 135006516
  Values :         0

$chr12
numeric-Rle of length 133851895 with 1 run
  Lengths: 133851895
  Values :         0

$chr13
numeric-Rle of length 115169878 with 1 run
  Lengths: 115169878
  Values :         0

<20 more elements>
```

You can get all of `chr22` by

```
> gr.chr22 <- GRanges(seqnames = "chr22",
+                     ranges = IRanges(start = 1, end = seqlengths(gr)["chr22"]))
> out.chr22 <- import(bw, which = gr.chr22, as = "Rle")
> out.chr22[["chr22"]]
numeric-Rle of length 51304566 with 1381642 runs
  Lengths:          16050196               194 ...             61301
  Values :                 0 0.465829998254776 ...                 0
```

18.8 LiftOver

LiftOver is a popular tool from the UCSC Genome Browser for converting between different genome versions. The *rtracklayer* package also exposes this function through the `liftOver`. To use `liftOver` you need a so-called "chain" file describing how to convert from one genome to another. This can be obtained by hand from UCSC, or directly from *AnnotationHub*.

We can re-use our `AnnotationHub`:

```
> ahub.chain <- subset(ahub, rdataclass == "ChainFile" & species == "Homo sapien\
s")
> query(ahub.chain, c("hg18", "hg19"))
AnnotationHub with 2 records
# snapshotDate(): 2016-05-12
# $dataprovider: UCSC
# $species: Homo sapiens
# $rdataclass: ChainFile
# additional mcols(): taxonomyid, genome, description, tags,
#   sourceurl, sourcetype
# retrieve records with, e.g., 'object[["AH14149"]]'

            title
  AH14149 | hg19ToHg18.over.chain.gz
  AH14220 | hg18ToHg19.over.chain.gz
> chain <- ahub.chain[ahub.chain$title == "hg19ToHg18.over.chain.gz"]
> chain <- chain[[1]]
> gr.hg18 <- liftOver(gr, chain)
> gr.hg18
GRangesList object of length 12011:
[[1]]
GRanges object with 1 range and 6 metadata columns:
      seqnames              ranges strand |        name     score
         <Rle>           <IRanges>  <Rle> | <character> <integer>
  [1]    chr17 [75581759, 75582503]      * |           .         0
      signalValue    pValue        qValue      peak
        <numeric> <numeric>     <numeric> <integer>
  [1]      17.672   243.175 6.941353e-239       401

[[2]]
GRanges object with 1 range and 6 metadata columns:
      seqnames              ranges strand | name score signalValue
  [1]    chr14 [30768512, 30769100]      * |    .     0      16.782
       pValue        qValue peak
```

```
  [1] 240.744 9.362981e-237  346

[[3]]
GRanges object with 1 range and 6 metadata columns:
      seqnames              ranges strand | name score signalValue
  [1]     chr2 [46489430, 46490315]      * |    .     0      18.917
       pValue        qValue peak
  [1] 226.703 6.859981e-223  241

<12008 more elements>
seqinfo: 23 sequences from an unspecified genome; no seqlengths
```

This converts a `GRanges` into a `GRangesList`, why? This is because a single range (interval) may be split into multiple intervals in the other genome. So each element in the output correspond to a single range in the input. If the ranges are small, most ranges should be mapped to a single range. Let us look at the number of elements in output:

```
> table(elementLengths(gr.hg18))

    0     1     2     3
    7 11996     5     3
```

Only a few ranges were not mapped and only a few were split.

18.9 Importing directly from UCSC

Using *rtracklayer* you can import tables and tracks directly from the UCSC Genome Browser. However, it is now possible to get a lot (all?) of this data from *AnnotationHub* and this later package seems friendlier.

It is possible that not all tracks / tables and/or all information from the different track / tables from UCSC are exposed in *AnnotationHub*.

See a detailed exposition in the package vignette.

18.10 Tabix indexing

Tabix indexing is a way to index a text file with chromosomal positions for random access. This will greatly speed up any querying of such a file. The tabix[2] functionality was introduced in the SAMtools library; this library was later renamed to htslib.

[2] http://www.htslib.org/doc/tabix.html

Tabix indexing is usually something you do at the command line, but there is also the convenient possibility of doing it from inside Bioconductor using `indexTabix` from the *Rsamtools* package. First however, the file needs to be bgzip2 compressed, which you can do using the `bgzip2` function. A full pipeline, using an example SAM file from *Rsamtools* is

```
> library(Rsamtools)
> from <- system.file("extdata", "ex1.sam", package="Rsamtools",
+                     mustWork=TRUE)
> from
[1] "/Library/Frameworks/R.framework/Versions/3.3/Resources/library/Rsamtools/ex\
tdata/ex1.sam"
> to <- tempfile()
> zipped <- bgzip(from, to)
> idx <- indexTabix(zipped, "sam")
```

see also the help page for `indexTabix`.

18.11 Other Resources

- The vignette from the rtracklayer package[3].

[3] http://bioconductor.org/packages/rtracklayer

19. ExpressionSet

Watch a video[1] of this chapter.

19.1 Dependencies

This document has the following dependencies:

```
> library(Biobase)
> library(ALL)
> library(hgu95av2.db)
```

Use the following commands to install these packages in R.

```
> source("http://www.bioconductor.org/biocLite.R")
> biocLite(c("Biobase", "ALL", "hgu95av2.db"))
```

19.2 Overview

We will examine how to use and manipulate the `ExpressionSet` class; a fundamental data container in Bioconductor. The class is defined in the *Biobase* package.

This class has inspired many other data containers in Bioconductor. It is a great example of *programming with data*.

19.3 Data Containers

R/Bioconductor has a rich set of data containers, or types of objects for storing different types of genomic data. Data containers might seem a boring subject at first, but in my opinion, they have been critical to the success of Bioconductor. Don't just take my word for it; the following quote is from Robert Gentleman (co-creator of R and founder of Bioconductor) on successful computational biology software (Altschul et al. 2013):

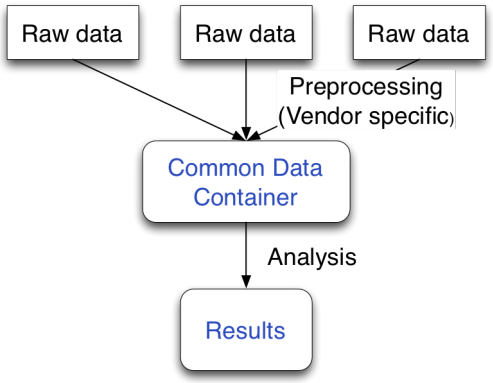
> If everybody puts their gene expression data into the same kind of box, it doesn't matter how the data came about, but that box is the same and can be used by analytic tools

[1] https://youtu.be/wVFxRsz2zGQ

The classic example is gene expression microarray data. In Bioconductor, data from such an experiment goes through the following pipeline

1. Data is read into Bioconductor from a vendor specific file format and stored as raw data in a vendor specific data container.
2. The raw data is preprocessed, using statistical and computational methods which are specific to the vendor specific data.
3. After preprocessing, the data is stored in an `ExpressionSet` data container.

For example, data from Affymetrix microarrays and Agilent microarrays have (in their raw version) aspects of the data which are different to each vendor. Affymetrix uses short oligonucleotide probes and therefore uses multiple probes to measure a single gene. In contrast, Agilent uses long(er) probes and typically uses a single probe to measure a gene. But after preprocessing, both platforms have a set of measurements on a set of genes. At this level, the data becomes similar.

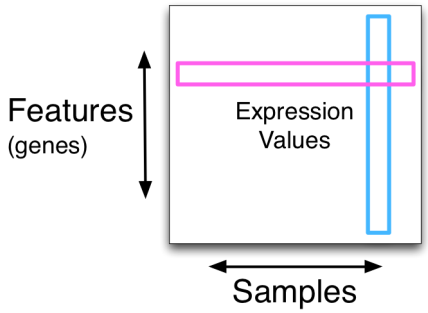


Multiple types of raw data are unified in a single data container.

Having well designed data containers allows developer to write methods for these data containers which will be applicable to many types of use cases. And it helps users by making the data easier to manipulate; thereby reducing errors.

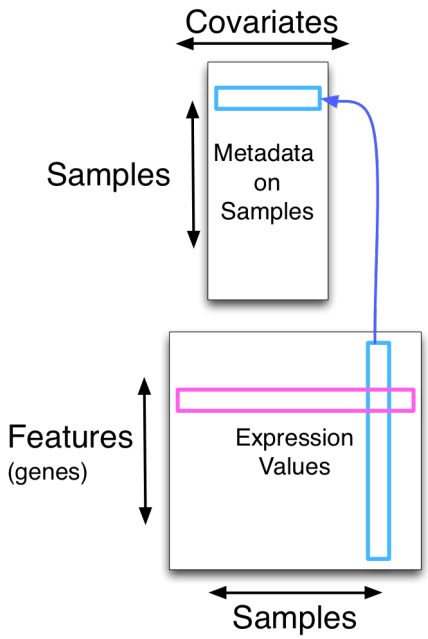
19.4 The structure of an ExpressionSet

Gene expression data can be thought of as a data matrix of expression values, one for each (gene, sample) pair. Tradition demands that rows are genes and columns are samples. Genes are sometimes called features. An example of this is



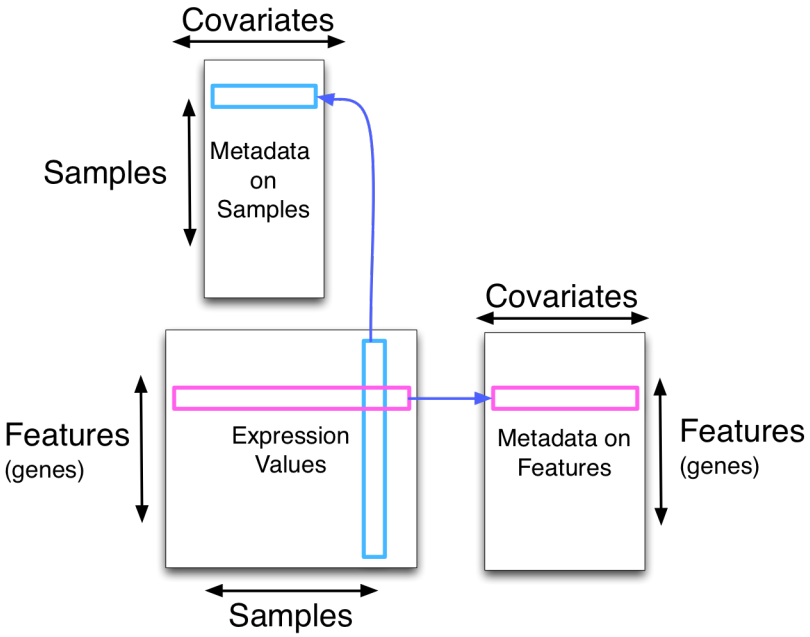
Gene by sample matrix of expression values. The pink rectangle denotes the measurement of all genes in one sample, and the blue rectangle denote the measurement of one gene across all samples.

We always have additional data on the samples, often called covariates or phenotype data. This is represented in a new data matrix; this time samples are in the rows and covariates are in the columns. There is an implicit link between columns of the gene expression data matrix and rows of the phenotype matrix. An example of this is



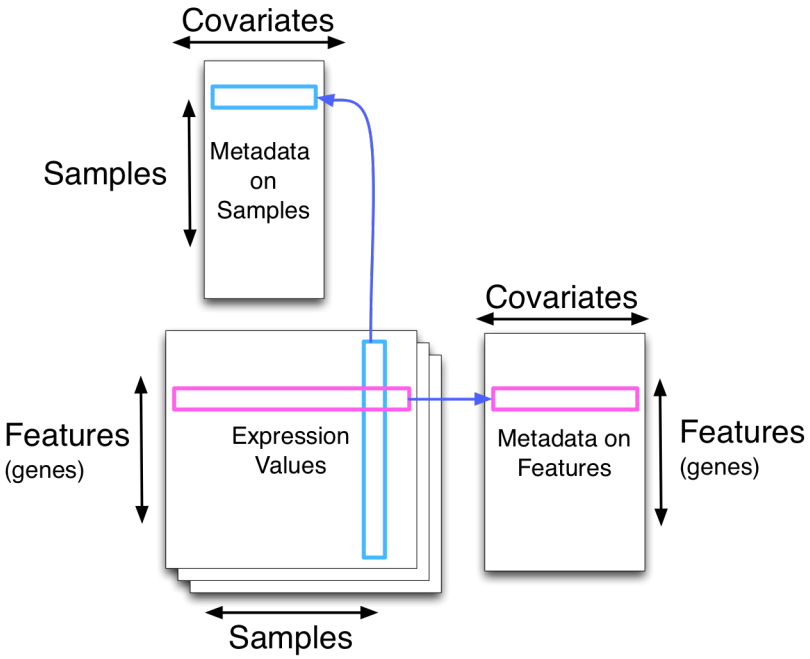
Gene by sample matrix linked to a sample by covariate matrix.

You can think of the phenotype information as annotation on the columns of the expression matrix. We also allow annotation on the rows of the expression matrix. This is additional information on each *gene* in the matrix, such as name, location and GC content. Historically, this has been used less in Bioconductor. An full depiction of an `ExpressionSet` is seen here:



An ExpressionSet, linking a gene expression matrix to metadata on both columns (samples) and rows (features).

The `ExpressionSet` construction has been used for many other ttypes of data, besides gene expression data. A general class, allowing for multiple data matrices, is an `eSet` depicted below.



An eSet

Examples of the need for multiple data matrices are DNA methylation microarrays where two measurements are obtained for each feature (a methylated and an unmethylated channel).

19.5 Example

An example dataset, stored as an `ExpressionSet` is available in the *ALL*[2] package. This package is an example of an "experimental data" package"; a bundling of a full experiment inside an R package. These experimental data packages are used for teaching, testing and illustration, but can also serve as the documentation of a data analysis.

```
> library(ALL)
> data(ALL)
> ALL
ExpressionSet (storageMode: lockedEnvironment)
assayData: 12625 features, 128 samples
  element names: exprs
protocolData: none
phenoData
  sampleNames: 01005 01010 ... LAL4 (128 total)
  varLabels: cod diagnosis ... date last seen (21 total)
  varMetadata: labelDescription
featureData: none
experimentData: use 'experimentData(object)'
  pubMedIds: 14684422 16243790
Annotation: hgu95av2
```

(you don't always have to explicitly call `data()` on all datasets in R; that depends on the choice of the package author).

This is an experiment done on an Affymetrix HGU 95av2 gene expression microarray; the authors profiled 128 samples.

Because this is data from an experiment package, there is documentation in the help page for the dataset, see

```
> ?ALL
```

From the printed description of `ALL` you can see that 12625 features (in this case genes) were measured on 128 samples. The object contains information about the experiment; look at

[2] http://bioconductor.org/packages/ALL

```
> experimentData(ALL)
Experiment data
  Experimenter name: Chiaretti et al.
  Laboratory: Department of Medical Oncology, Dana-Farber Cancer Institute, Depa\
rtment of Medicine, Brigham and Women's Hospital, Harvard Medical School, Boston\
, MA 02115, USA.
  Contact information:
  Title: Gene expression profile of adult T-cell acute lymphocytic leukemia iden\
tifies distinct subsets of patients with different response to therapy and survi\
val.
  URL:
  PMIDs: 14684422 16243790

  Abstract: A 187 word abstract is available. Use 'abstract' method.
```

Two papers (pubmed IDs or PMIDs) are associated with the data.

There is no `protocolData` in the object (this is intended to describe the experimental protocol); this is typical in my experience (although it would be great to have this information easily available to the end user).

The core of the object are two matrices

- the `exprs` matrix containing the 12625 gene expression measurements on the 128 samples (a 12625 by 128 numeric matrix).
- the `pData` `data.frame` containing phenotype data on the samples.

You get the expression data by

```
> exprs(ALL)[1:4, 1:4]
             01005    01010    03002    04006
1000_at   7.597323 7.479445 7.567593 7.384684
1001_at   5.046194 4.932537 4.799294 4.922627
1002_f_at 3.900466 4.208155 3.886169 4.206798
1003_s_at 5.903856 6.169024 5.860459 6.116890
```

Note how this matrix has column and rownames. These are `sampleNames` and `featureNames`. Get them by

```
> head(sampleNames(ALL))
[1] "01005" "01010" "03002" "04006" "04007" "04008"
> head(featureNames(ALL))
[1] "1000_at"   "1001_at"   "1002_f_at" "1003_s_at" "1004_at"   "1005_at"
```

To get at the `pData` information, using the `pData` accessor.

```
> head(pData(ALL))
       cod diagnosis sex age BT remission CR   date.cr t(4;11) t(9;22)
01005 1005 5/21/1997   M  53 B2        CR CR  8/6/1997   FALSE    TRUE
01010 1010 3/29/2000   M  19 B2        CR CR 6/27/2000   FALSE   FALSE
03002 3002 6/24/1998   F  52 B4        CR CR 8/17/1998      NA      NA
04006 4006 7/17/1997   M  38 B1        CR CR  9/8/1997    TRUE   FALSE
04007 4007 7/22/1997   M  57 B2        CR CR 9/17/1997   FALSE   FALSE
04008 4008 7/30/1997   M  17 B1        CR CR 9/27/1997   FALSE   FALSE
      cyto.normal        citog mol.biol fusion protein mdr   kinet   ccr
01005       FALSE      t(9;22)  BCR/ABL           p210 NEG dyploid FALSE
01010       FALSE  simple alt.      NEG           <NA> POS dyploid FALSE
03002          NA         <NA>  BCR/ABL           p190 NEG dyploid FALSE
04006       FALSE      t(4;11) ALL1/AF4           <NA> NEG dyploid FALSE
04007       FALSE      del(6q)      NEG           <NA> NEG dyploid FALSE
04008       FALSE complex alt.      NEG           <NA> NEG hyperd. FALSE
      relapse transplant               f.u date last seen
01005   FALSE       TRUE BMT / DEATH IN CR           <NA>
01010    TRUE      FALSE               REL      8/28/2000
03002    TRUE      FALSE               REL     10/15/1999
04006    TRUE      FALSE               REL      1/23/1998
04007    TRUE      FALSE               REL      11/4/1997
04008    TRUE      FALSE               REL     12/15/1997
```

You can access individual columns of this `data.frame` by using the `$` operator:

```
> head(pData(ALL)$sex)
[1] M M F M M M
Levels: F M
> head(ALL$sex)
[1] M M F M M M
Levels: F M
```

19.6 Subsetting

Subsetting of this object is an important operation. The subsetting has two dimensions; the first dimension is genes and the second is samples. It keeps track of how the expression measures are matched to the pheno data.

```
> ALL[,1:5]
ExpressionSet (storageMode: lockedEnvironment)
assayData: 12625 features, 5 samples
  element names: exprs
protocolData: none
phenoData
  sampleNames: 01005 01010 ... 04007 (5 total)
  varLabels: cod diagnosis ... date last seen (21 total)
  varMetadata: labelDescription
featureData: none
experimentData: use 'experimentData(object)'
  pubMedIds: 14684422 16243790
Annotation: hgu95av2
> ALL[1:10,]
ExpressionSet (storageMode: lockedEnvironment)
assayData: 10 features, 128 samples
  element names: exprs
protocolData: none
phenoData
  sampleNames: 01005 01010 ... LAL4 (128 total)
  varLabels: cod diagnosis ... date last seen (21 total)
  varMetadata: labelDescription
featureData: none
experimentData: use 'experimentData(object)'
  pubMedIds: 14684422 16243790
Annotation: hgu95av2
> ALL[1:10,1:5]
ExpressionSet (storageMode: lockedEnvironment)
assayData: 10 features, 5 samples
  element names: exprs
protocolData: none
phenoData
  sampleNames: 01005 01010 ... 04007 (5 total)
  varLabels: cod diagnosis ... date last seen (21 total)
  varMetadata: labelDescription
featureData: none
```

```
experimentData: use 'experimentData(object)'
  pubMedIds: 14684422 16243790
Annotation: hgu95av2
```

We can even change the order of the samples

```
> ALL[, c(3,2,1)]
ExpressionSet (storageMode: lockedEnvironment)
assayData: 12625 features, 3 samples
  element names: exprs
protocolData: none
phenoData
  sampleNames: 03002 01010 01005
  varLabels: cod diagnosis ... date last seen (21 total)
  varMetadata: labelDescription
featureData: none
experimentData: use 'experimentData(object)'
  pubMedIds: 14684422 16243790
Annotation: hgu95av2
> ALL$sex[c(1,2,3)]
[1] M M F
Levels: F M
> ALL[, c(3,2,1)]$sex
[1] F M M
Levels: F M
```

This gives us a lot of confidence that the data is properly matched to the phenotypes.

19.7 featureData and annotation

You can think of `pData` as providing information on the columns (samples) of the measurement matrix. Similar to `pData`, we have `featureData` which is meant to provide information on the features (genes) of the array.

However, while this slot has existed for many years, it often goes unused:

```
> featureData(ALL)
An object of class 'AnnotatedDataFrame': none
```

So this leaves us with a key question: so far the `ALL` object has some useful description of the type of experiment and which samples were profiled. But how do we get information on which genes were profiled?

For commercially available microarrays, the approach in Bioconductor has been to make so-called annotation packages which links `featureNames` to actual useful information. Conceptually, this information could have been stored in `featureData`, but isn't.

Example

```
> ids <- featureNames(ALL)[1:5]
> ids
[1] "1000_at"   "1001_at"   "1002_f_at" "1003_s_at" "1004_at"
```

these are ids named by Affymetrix, the microarray vendor. We can look them up in the annotation package by

```
> library(hgu95av2.db)
> as.list(hgu95av2ENTREZID[ids])
$`1000_at`
[1] "5595"

$`1001_at`
[1] "7075"

$`1002_f_at`
[1] "1557"

$`1003_s_at`
[1] "643"

$`1004_at`
[1] "643"
```

This gives us the Entrez ID associated with the different measurements. There are a number of so-called "maps" like `hgu95av2XX` with `XX` having multiple values. This approach is very specific to Affymetrix chips. An alternative to using annotation packages is to use the *biomaRt* package to get the microarray annotation from Ensembl (an online database). This will be discussed elsewhere.

We will leave the annotation subject for now.

19.8 Note: phenoData and pData

For this type of object, there is a difference between `phenoData` (an object of class `AnnotatedDataFrame`) and `pData` (an object of class `data.frame`).

The idea behind `AnnotatedDataFrame` was to include additional information on what a `data.frame` contains, by having a list of descriptions called `varLabels`.

```
> pD <- phenoData(ALL)
> varLabels(pD)
 [1] "cod"            "diagnosis"      "sex"            "age"
 [5] "BT"             "remission"      "CR"             "date.cr"
 [9] "t(4;11)"        "t(9;22)"        "cyto.normal"    "citog"
[13] "mol.biol"       "fusion protein" "mdr"            "kinet"
[17] "ccr"            "relapse"        "transplant"     "f.u"
[21] "date last seen"
```

But these days, it seems that `varLabels` are constrained to be equal to the column names of the `data.frame` making the entire `AnnotatedDataFrame` construction unnecessary:

```
> varLabels(pD)[2] <- "Age at diagnosis"
> pD
An object of class 'AnnotatedDataFrame'
  sampleNames: 01005 01010 ... LAL4 (128 total)
  varLabels: cod Age at diagnosis ... date last seen (21 total)
  varMetadata: labelDescription
> colnames(pD)[1:3]
[1] "cod"              "Age at diagnosis" "sex"
> varLabels(pD)[1:3]
[1] "cod"              "Age at diagnosis" "sex"
```

So now we have exposed what is arguably a bug, together with some un-used abstraction. This happens.

19.9 The eSet class

The `ExpressionSet` class is an example of an `eSet`. The `ExpressionSet` class has a single measurement matrix which we access by `exprs`. In contrast to this, the `eSet` class can have any number of measurement matrices with arbitrary names, although all matrices needs to have the same dimension.

An example of another `eSet` derived class is `NChannelSet` which is meant to store multichannel microarray measurements; an example is a two-color array where the microarray has been imaged in two different colors.

Another example of classes are from the *minfi* package for DNA methylation arrays; here there are classes such as `RGChannelSet` for a two color microarray and `MethylSet` for methylation measurements.

19.10 Other Resources

- The "An Introduction to Bioconductor's ExpressionSet Class" vignette from the Biobase webpage[3].
- The "Notes for eSet Developers" vignette from the Biobase webpage[4] contains advanced information.

Altschul, Stephen, Barry Demchak, Richard Durbin, Robert Gentleman, Martin Krzywinski, Heng Li, Anton Nekrutenko, et al. 2013. "The Anatomy of Successful Computational Biology Software." *Nature Biotechnology* 31 (10): 894–97. doi:10.1038/nbt.2721[5].

[3] http://bioconductor.org/packages/Biobase

[4] http://bioconductor.org/packages/Biobase

[5] https://doi.org/10.1038/nbt.2721

20. SummarizedExperiment

Watch a video[1] of this chapter.

20.1 Dependencies

This document has the following dependencies:

```
> library(SummarizedExperiment)
> library(GenomicRanges)
> library(airway)
```

Use the following commands to install these packages in R.

```
> source("http://www.bioconductor.org/biocLite.R")
> biocLite(c("SummarizedExperiment", "GenomicRanges", "airway"))
```

20.2 Overview

We will present the `SummarizedExperiment` class from *GenomicRanges* package; an extension of the `ExpressionSet` class to include `GRanges`.

This class is suitable for storing processed data particularly from high-throughout sequencing assays.

20.3 Details

An example dataset, stored as a `SummarizedExperiment` is available in the *airway*[2] package. This data represents an RNA sequencing experiment, but we will use it only for illustrating the class.

[1] https://youtu.be/D8IVRmbMjyc
[2] http://bioconductor.org/packages/airway

```
> library(airway)
> data(airway)
> airway
class: RangedSummarizedExperiment
dim: 64102 8
metadata(1): ''
assays(1): counts
rownames(64102): ENSG00000000003 ENSG00000000005 ... LRG_98 LRG_99
rowData names(0):
colnames(8): SRR1039508 SRR1039509 ... SRR1039520 SRR1039521
colData names(9): SampleName cell ... Sample BioSample
```

This looks similar to - and yet different from - the `ExpressionSet`. Things now have different names, representing that these classes were designed about 10 years apart.

We have 8 samples and 64102 features (genes).

Some aspects of the object are very similar to `ExpressionSet`, although with slightly different names and types:

`colData` contains phenotype (sample) information, like `pData` for `ExpressionSet`. It returns a `DataFrame` instead of a `data.frame`:

```
> colData(airway)
DataFrame with 8 rows and 9 columns
            SampleName     cell      dex    albut        Run avgLength
              <factor> <factor> <factor> <factor>   <factor> <integer>
SRR1039508 GSM1275862   N61311    untrt    untrt SRR1039508       126
SRR1039509 GSM1275863   N61311      trt    untrt SRR1039509       126
SRR1039512 GSM1275866  N052611    untrt    untrt SRR1039512       126
SRR1039513 GSM1275867  N052611      trt    untrt SRR1039513        87
SRR1039516 GSM1275870  N080611    untrt    untrt SRR1039516       120
SRR1039517 GSM1275871  N080611      trt    untrt SRR1039517       126
SRR1039520 GSM1275874  N061011    untrt    untrt SRR1039520       101
SRR1039521 GSM1275875  N061011      trt    untrt SRR1039521        98
            Experiment    Sample    BioSample
              <factor>  <factor>     <factor>
SRR1039508  SRX384345 SRS508568 SAMN02422669
SRR1039509  SRX384346 SRS508567 SAMN02422675
SRR1039512  SRX384349 SRS508571 SAMN02422678
SRR1039513  SRX384350 SRS508572 SAMN02422670
SRR1039516  SRX384353 SRS508575 SAMN02422682
SRR1039517  SRX384354 SRS508576 SAMN02422673
SRR1039520  SRX384357 SRS508579 SAMN02422683
SRR1039521  SRX384358 SRS508580 SAMN02422677
```

You can still use `$` to get a particular column:

```
> airway$cell
[1] N61311  N61311  N052611 N052611 N080611 N080611 N061011 N061011
Levels: N052611 N061011 N080611 N61311
```

`metadata` is like `experimentData` from `ExpressionSet`. This slot is often un-used; this is the case for this object:

```
> metadata(airway)
[[1]]
Experiment data
  Experimenter name: Himes BE
  Laboratory: NA
  Contact information:
  Title: RNA-Seq transcriptome profiling identifies CRISPLD2 as a glucocorticoid\
 responsive gene that modulates cytokine function in airway smooth muscle cells.\

  URL: http://www.ncbi.nlm.nih.gov/pubmed/24926665
  PMIDs: 24926665

  Abstract: A 226 word abstract is available. Use 'abstract' method.
```

`colnames` are like `sampleNames` from `ExpressionSet`; `rownames` are like `featureNames`.

```
> colnames(airway)
[1] "SRR1039508" "SRR1039509" "SRR1039512" "SRR1039513" "SRR1039516"
[6] "SRR1039517" "SRR1039520" "SRR1039521"
> head(rownames(airway))
[1] "ENSG00000000003" "ENSG00000000005" "ENSG00000000419" "ENSG00000000457"
[5] "ENSG00000000460" "ENSG00000000938"
```

The measurement data are accessed by `assay` and `assays`. A `SummarizedExperiment` can contain multiple measurement matrices (all of the same dimension). You get all of them by `assays` and you select a particular one by `assay(OBJECT, NAME)` where you can see the names when you print the object or by using `assayNames`. In this case there is a single matrix called `counts`:

```
> airway
class: RangedSummarizedExperiment
dim: 64102 8
metadata(1): ''
assays(1): counts
rownames(64102): ENSG00000000003 ENSG00000000005 ... LRG_98 LRG_99
rowData names(0):
colnames(8): SRR1039508 SRR1039509 ... SRR1039520 SRR1039521
colData names(9): SampleName cell ... Sample BioSample
> assayNames(airway)
[1] "counts"
> assays(airway)
List of length 1
names(1): counts
> head(assay(airway, "counts"))
                SRR1039508 SRR1039509 SRR1039512 SRR1039513 SRR1039516
ENSG00000000003        679        448        873        408       1138
ENSG00000000005          0          0          0          0          0
ENSG00000000419        467        515        621        365        587
ENSG00000000457        260        211        263        164        245
ENSG00000000460         60         55         40         35         78
ENSG00000000938          0          0          2          0          1
                SRR1039517 SRR1039520 SRR1039521
ENSG00000000003       1047        770        572
ENSG00000000005          0          0          0
ENSG00000000419        799        417        508
ENSG00000000457        331        233        229
ENSG00000000460         63         76         60
ENSG00000000938          0          0          0
```

So far, this is all information which could be stored in an `ExpressionSet`. The new thing is that `SummarizedExperiment` allows for a `rowRanges` (or `granges`) data representing the different features. The idea is that these `GRanges` tells us which part of the genome is summarized for a particular feature. Let us take a look

```
> length(rowRanges(airway))
[1] 64102
> dim(airway)
[1] 64102     8
> rowRanges(airway)
GRangesList object of length 64102:
$ENSG00000000003
GRanges object with 17 ranges and 2 metadata columns:
       seqnames              ranges strand |   exon_id       exon_name
          <Rle>           <IRanges>  <Rle> | <integer>     <character>
   [1]        X [99883667, 99884983]      - |    667145 ENSE00001459322
   [2]        X [99885756, 99885863]      - |    667146 ENSE00000868868
   [3]        X [99887482, 99887565]      - |    667147 ENSE00000401072
   [4]        X [99887538, 99887565]      - |    667148 ENSE00001849132
   [5]        X [99888402, 99888536]      - |    667149 ENSE00003554016
   ...      ...                 ...    ... .       ...             ...
  [13]        X [99890555, 99890743]      - |    667156 ENSE00003512331
  [14]        X [99891188, 99891686]      - |    667158 ENSE00001886883
  [15]        X [99891605, 99891803]      - |    667159 ENSE00001855382
  [16]        X [99891790, 99892101]      - |    667160 ENSE00001863395
  [17]        X [99894942, 99894988]      - |    667161 ENSE00001828996

<64101 more elements>
seqinfo: 722 sequences (1 circular) from an unspecified genome
```

See how `rowRanges` is a `GRangesList` (it could also be a single `GRanges`). Each element of the list represents a feature and the `GRanges` of the feature tells us the coordinates of the exons in the gene (or transcript). Because these are genes, for each `GRanges`, all the ranges should have the same `strand` and `seqnames`, but that is not enforced.

In total we have around 64k "genes" or "transcripts" and around 745k different exons.

```
> length(rowRanges(airway))
[1] 64102
> sum(elementLengths(rowRanges(airway)))
[1] 745593
```

For some operations, you donâ€™t need to use `rowRanges` first, you can use the operation directly on the object. Here is an example with `start()`:

```
> start(rowRanges(airway))
IntegerList of length 64102
[["ENSG00000000003"]] 99883667 99885756 99887482 ... 99891790 99894942
[["ENSG00000000005"]] 99839799 99840228 99848621 ... 99854013 99854505
[["ENSG00000000419"]] 49551404 49551404 49551433 ... 49574900 49574900
[["ENSG00000000457"]] 169818772 169821804 ... 169862929 169863148
[["ENSG00000000460"]] 169631245 169652610 ... 169821931 169821931
[["ENSG00000000938"]] 27938575 27938803 27938811 ... 27961576 27961576
[["ENSG00000000971"]] 196621008 196621186 ... 196716241 196716241
[["ENSG00000001036"]] 143815948 143817763 ... 143832548 143832548
[["ENSG00000001084"]] 53362139 53362139 53362940 ... 53429548 53481684
[["ENSG00000001167"]] 41040684 41040722 41046768 ... 41065096 41065096
<64092 more elements>
> start(airway)
IntegerList of length 64102
[["ENSG00000000003"]] 99883667 99885756 99887482 ... 99891790 99894942
[["ENSG00000000005"]] 99839799 99840228 99848621 ... 99854013 99854505
[["ENSG00000000419"]] 49551404 49551404 49551433 ... 49574900 49574900
[["ENSG00000000457"]] 169818772 169821804 ... 169862929 169863148
[["ENSG00000000460"]] 169631245 169652610 ... 169821931 169821931
[["ENSG00000000938"]] 27938575 27938803 27938811 ... 27961576 27961576
[["ENSG00000000971"]] 196621008 196621186 ... 196716241 196716241
[["ENSG00000001036"]] 143815948 143817763 ... 143832548 143832548
[["ENSG00000001084"]] 53362139 53362139 53362940 ... 53429548 53481684
[["ENSG00000001167"]] 41040684 41040722 41046768 ... 41065096 41065096
<64092 more elements>
```

You can use `granges()` as synonymous for `rowRanges()`.

Subsetting works like `ExpressionSet`: there are two dimensions, the first dimension is features (genes) and the second dimension is samples.

Because the `SummarizedExperiment` contains a `GRanges[List]` you can also use `subsetByOverlaps`, like

```
> gr <- GRanges(seqnames = "1", ranges = IRanges(start = 1, end = 10^7))
> subsetByOverlaps(airway, gr)
class: RangedSummarizedExperiment
dim: 329 8
metadata(1): ''
assays(1): counts
rownames(329): ENSG00000007923 ENSG00000008128 ... ENSG00000272512
  ENSG00000273443
rowData names(0):
colnames(8): SRR1039508 SRR1039509 ... SRR1039520 SRR1039521
colData names(9): SampleName cell ... Sample BioSample
```

21. GEOquery

Watch a video[1] of this chapter.

21.1 Dependencies

This document has the following dependencies:

```
> library(GEOquery)
```

Use the following commands to install these packages in R.

```
> source("http://www.bioconductor.org/biocLite.R")
> biocLite(c("GEOquery"))
```

21.2 Overview

NCBI Gene Expression Omnibus (GEO) has a lot of high-throughput genomics datasets publicly available. Despite the name, this database is not exclusively for gene expression data.

21.3 NCBI GEO

NCBI GEO is organised as samples which are grouped into series. For bigger experiments there are both SubSeries and SuperSeries. A SuperSeries is all the experiments for a single paper; a SuperSeries can be decomposed into SubSeries which are different technologies. As an example, look at the SuperSeries GSE19486[2]. In this paper they used 2 different platforms (this is a weird name; a platform is a combination of a technology and a species). And they did RNA-seq and ChIP-seq for two different factors (NFkB-II and Pol II). This results in 4 SubSeries (2 for RNA-seq and 2 for ChIP-seq).

A simpler setup is for example GSE994[3] where samples from current and former smokers were compared, using an Affymetrix microarray.

[1] https://youtu.be/hO4ORyp9FDo

[2] http://www.ncbi.nlm.nih.gov/geo/query/acc.cgi?acc=GSE19486

[3] http://www.ncbi.nlm.nih.gov/geo/query/acc.cgi?acc=GSE994

Data submitted to NCBI GEO can be both "raw" and "proceseed". Let us focus on gene expression data for the moment. "Processed" data is normalized and quantified, typically at the gene level, and is usually provided in the form of a gene by sample matrix. "Raw" data can be anything, from sequencing reads to microarray image files. There can even be different states of "raw" data, for example for a RNA-seq dataset you may have

- FASTQ files (raw reads)
- BAM files (aligned reads)
- gene by sample expression matrix (unnormalized)
- gene by sample expression matrix (normalized)

So what is "raw" and what is "processed" can be very context dependent.

In some cases there is a consensus in the field. For Affymetrix gene expression microarrays, "raw" files are so-called CEL files (a file format invented by Affymetrix) and "processed" data is normalized and quantified data, summarized at the probeset level.

At the end of the day, GEO has series identifiers (like GSE19486) and sample identiers (GSM486297). Note the GSE vs GSM in the same. A user is almost always interested in all the samples in a given series, so the starting point is the series identifier, also called the accession number.

21.4 GEOquery

All you need to download data from GEO is the accession number. Let us use GSE11675[4] which is a very small scale Affymetrix gene expression array study (6 samples).

```
> eList <- getGEO("GSE11675")
> class(eList)
[1] "list"
> length(eList)
[1] 1
> names(eList)
[1] "GSE11675_series_matrix.txt.gz"
> eData <- eList[[1]]
> eData
ExpressionSet (storageMode: lockedEnvironment)
assayData: 12625 features, 6 samples
  element names: exprs
protocolData: none
phenoData
```

[4] http://www.ncbi.nlm.nih.gov/geo/query/acc.cgi?acc=GSE11675

```
  sampleNames: GSM296630 GSM296635 ... GSM296639 (6 total)
  varLabels: title geo_accession ... data_row_count (34 total)
  varMetadata: labelDescription
featureData
  featureNames: 1000_at 1001_at ... AFFX-YEL024w/RIP1_at (12625
    total)
  fvarLabels: ID GB_ACC ... Gene Ontology Molecular Function (16
    total)
  fvarMetadata: Column Description labelDescription
experimentData: use 'experimentData(object)'
Annotation: GPL8300
```

The function returns a `list` because you might be getting multiple SubSeries. In this case there is only one, and the list element is an `ExpressionSet` ready for usage! The phenotype data (which GEO knows about) is contained inside the `pData` slot; there is usually a lot of unnecessary stuff here:

```
> names(pData(eData))
 [1] "title"                  "geo_accession"
 [3] "status"                 "submission_date"
 [5] "last_update_date"       "type"
 [7] "channel_count"          "source_name_ch1"
 [9] "organism_ch1"           "characteristics_ch1"
[11] "characteristics_ch1.1"  "characteristics_ch1.2"
[13] "molecule_ch1"           "extract_protocol_ch1"
[15] "label_ch1"              "label_protocol_ch1"
[17] "taxid_ch1"              "hyb_protocol"
[19] "scan_protocol"          "description"
[21] "data_processing"        "platform_id"
[23] "contact_name"           "contact_email"
[25] "contact_phone"          "contact_department"
[27] "contact_institute"      "contact_address"
[29] "contact_city"           "contact_zip/postal_code"
[31] "contact_country"        "supplementary_file"
[33] "supplementary_file.1"   "data_row_count"
```

However, what we got here was processed data. Users (including me) often wants access to more raw data. This is called "supplementary files" in GEO language and we can get those as well. For display purposes we only keep the `basename()` of the files.

```
> eList2 <- getGEOSuppFiles("GSE11675")
> rownames(eList2) <- basename(rownames(eList2))
> eList2
                     size isdir mode               mtime
GSE11675_RAW.tar 45803520 FALSE  644 2016-05-23 10:40:26
filelist.txt         740 FALSE  644 2016-05-23 10:40:26
                                ctime               atime uid gid   uname
GSE11675_RAW.tar 2016-05-23 10:40:26 2016-05-23 10:39:33 501  20 khansen
filelist.txt     2016-05-23 10:40:26 2016-05-21 13:05:45 501  20 khansen
                 grname
GSE11675_RAW.tar  staff
filelist.txt      staff
> tarArchive <- rownames(eList2)[1]
> tarArchive
[1] "GSE11675_RAW.tar"
```

This is now a `data.frame` of file names. A single TAR archive was downloaded. You can expand the TAR achive using standard tools; inside there is a list of 6 CEL files and 6 CHP files. You can then read the 6 CEL files into R using functions from *affy* or *oligo*.

It is also possible to use *GEOquery* to query GEO as a database (ie. looking for datasets); more information in the package vignette.

21.5 Other packages

There are other packages for accessing other online repositories with public data; they include *SRAdb* for the Short Read Archive (SRA) and *ArrayExpress* (ArrayExpress; a similar database to NCBI GEO but hosted at the European Bioinformatics Institute (EBI)).

21.6 Other Resources

- The vignette from the GEOquery package[5].
- GEO documentation[6].

[5] http://bioconductor.org/packages/GEOquery
[6] http://www.ncbi.nlm.nih.gov/geo/info/overview.html

22. biomaRt

Watch a video[1] of this chapter.

22.1 Dependencies

This document has the following dependencies:

```
> library(biomaRt)
```

Use the following commands to install these packages in R.

```
> source("http://www.bioconductor.org/biocLite.R")
> biocLite(c("biomaRt"))
```

22.2 Overview

We use a large number of different databases in computational biology. "Biomart" is a flexible interface to a biological database. The idea is that any kind of resource can setup a Biomart and then users can access the data using a single set of tools to access multiple databases.

The *biomaRt* package implements such an interface.

Databases supporting the Biomart interface includes Ensembl (from EBI), HapMap and UniProt.

22.3 Specifiying a mart and a dataset

To use *biomaRt* you need a mart (database) and a dataset inside the database. This is somewhat similar to *AnnotationHub*.

[1] https://youtu.be/-EXanoy2CGk

```
> head(listMarts())
               biomart               version
1 ENSEMBL_MART_ENSEMBL      Ensembl Genes 84
2     ENSEMBL_MART_SNP  Ensembl Variation 84
3 ENSEMBL_MART_FUNCGEN Ensembl Regulation 84
4    ENSEMBL_MART_VEGA               Vega 64
> mart <- useMart("ensembl")
> mart
Object of class 'Mart':
 Using the ENSEMBL_MART_ENSEMBL BioMart database
 Using the  dataset
> head(listDatasets(mart))
                         dataset
1         oanatinus_gene_ensembl
2        cporcellus_gene_ensembl
3        gaculeatus_gene_ensembl
4 itridecemlineatus_gene_ensembl
5         lafricana_gene_ensembl
6        choffmanni_gene_ensembl
                                description version
1     Ornithorhynchus anatinus genes (OANA5)   OANA5
2           Cavia porcellus genes (cavPor3) cavPor3
3     Gasterosteus aculeatus genes (BROADS1) BROADS1
4 Ictidomys tridecemlineatus genes (spetri2) spetri2
5         Loxodonta africana genes (loxAfr3) loxAfr3
6        Choloepus hoffmanni genes (choHof1) choHof1
> ensembl <- useDataset("hsapiens_gene_ensembl", mart)
> ensembl
Object of class 'Mart':
 Using the ENSEMBL_MART_ENSEMBL BioMart database
 Using the hsapiens_gene_ensembl dataset
```

You can see that the different datasets are organized by species; we have select *Homo sapiens*.

You access this database over the internet. Sometimes you need to specify a proxy server for this to work; details are in the *biomaRt* vignette; I have never encountered this.

22.4 Building a query

There is one main function in *biomaRt*: `getBM()` (get Biomart). This function retrives data from a Biomart based on a query. So it is important to understand how to build queries.

A Biomart query consists of 3 things: “attributes”, “filters” and “values”. Let us do an example. Let us say we want to annotate an Affymetrix gene expression microarray. We have Affymetrix probe ids in R and we want to retrieve gene names. In this case gene names is an “attribute” and Affymetrix probe ids is a “Filter”. Finally, the “values” are the actual values of the “filter”, ie. the ids.

An example might be (not run)

```
> values <- c("202763_at","209310_s_at","207500_at")
> getBM(attributes = c("ensembl_gene_id", "affy_hg_u133_plus_2"),
+       filters = "affy_hg_u133_plus_2", values = values, mart = ensembl)
  ensembl_gene_id affy_hg_u133_plus_2
1 ENSG00000137757           207500_at
2 ENSG00000164305           202763_at
3 ENSG00000196954         209310_s_at
```

Note that I list `affy_hg_133_plus_2` under both `attributes` and `filters`. It is listed under `attributes` because otherwise it would not appear in the return value of the function. If I don’t have the `aafy_hg_133_plus_2` column in my `data.frame` I wouldn’t know where genes are mapped to which probe ids. I would just have a list of which genes were measured on the array.

An example of a filter that might not appear in the `attributes` is if you want to only select autosomal genes. You may not care about which chromosomes the different genes appear on, just that they are on autosomal chromosomes.

Important note: Biomart (at least Ensembl) logs how often you query. If you query to many times, it disable access for a while. So the trick is to make a single vectorized query using a long list of `values` and not call `getBM()` for each individual value (doing this is also much, much slower).

A major part of using *biomaRt* is figuring out which `attributes` and which `filters` to use. You can get a description of this using `listAttributes()` and `listFilters()`; taht returns a very long `data.frame`.

```
> attributes <- listAttributes(ensembl)
> head(attributes)
                  name           description         page
1      ensembl_gene_id       Ensembl Gene ID feature_page
2 ensembl_transcript_id Ensembl Transcript ID feature_page
3   ensembl_peptide_id    Ensembl Protein ID feature_page
4      ensembl_exon_id       Ensembl Exon ID feature_page
5          description           Description feature_page
6      chromosome_name       Chromosome Name feature_page
> nrow(attributes)
[1] 1319
> filters <- listFilters(ensembl)
```

```
> head(filters)
             name     description
1 chromosome_name Chromosome name
2           start Gene Start (bp)
3             end   Gene End (bp)
4      band_start      Band Start
5        band_end        Band End
6    marker_start    Marker Start
> nrow(filters)
[1] 297
```

A lot of the `attributes` are gene names for the corresponding gene in a different organism. All these entries makes it a bit hard to get a good idea of what is there.

In Biomart, data is organized into pages (if you know about databases, this is a non-standard design). Each page contains a subset of attributes. You can get a more understandable set of attributes by using pages.

```
> attributePages(ensembl)
[1] "feature_page" "structure"    "homologs"     "snp"
[5] "snp_somatic"  "sequences"
> attributes <- listAttributes(ensembl, page = "feature_page")
> head(attributes)
                  name           description         page
1       ensembl_gene_id       Ensembl Gene ID feature_page
2 ensembl_transcript_id Ensembl Transcript ID feature_page
3    ensembl_peptide_id    Ensembl Protein ID feature_page
4       ensembl_exon_id       Ensembl Exon ID feature_page
5           description           Description feature_page
6       chromosome_name       Chromosome Name feature_page
> nrow(attributes)
[1] 178
```

All the homologs I complain about above are part of the ... `homologs` page.

An `attribute` can be part of multiple pages. It turns out that `getBM()` can only return a query which uses `attributes` from a single page. If you want to combine `attributes` from multiple pages you need to do multiple queries and then merge them.

Another aspect of working with `getBM()` is that sometimes the return `data.frame` contains duplicated rows. This is a consequence of the internal structure of the database and how queries are interpreted.

The *biomaRt* vignette is very useful and readable and contains a lot of example tasks, which can inspire future use. As a help, I have listed some of them here:

1. Annotate a set of Affymetrix identifiers with HUGO symbol and chromosomal locations of corresponding genes.
2. Annotate a set of EntrezGene identifiers with GO annotation.
3. Retrieve all HUGO gene symbols of genes that are located on chromosomes 17, 20 or Y, and are associated with one the following GO terms: “GO:0051330”, “GO:0000080”, “GO:0000114”, “GO:0000082”.
4. Annotate set of idenfiers with INTERPRO pro- tein domain identifiers.
5. Select all Affymetrix identifiers on the hgu133plus2 chip and Ensembl gene identifiers for genes located on chromosome 16 between basepair 1100000 and 1250000.
6. Retrieve all entrezgene identifiers and HUGO gene symbols of genes which have a “MAP kinase activity” GO term associated with it.
7. Given a set of EntrezGene identifiers, retrieve 100bp upstream promoter sequences.
8. Retrieve all 5’ UTR sequences of all genes that are located on chromosome 3 between the positions 185514033 and 185535839
9. Retrieve protein sequences for a given list of EntrezGene identifiers.
10. Retrieve known SNPs located on the human chromosome 8 between positions 148350 and 148612.
11. Given the human gene TP53, retrieve the human chromosomal location of this gene and also retrieve the chromosomal location and RefSeq id of it’s homolog in mouse.

22.5 Other Resources

- The vignette from the biomaRt package[2].

[2] http://bioconductor.org/packages/biomaRt

23. R - S4 Classes and Methods

Watch video 1[1] and video 2[2] of this chapter.

23.1 Dependencies

This document has the following dependencies:

```
> library(ALL)
> library(GenomicRanges)
```

Use the following commands to install these packages in R.

```
> source("http://www.bioconductor.org/biocLite.R")
> biocLite(c("ALL", "GenomicRanges"))
```

23.2 Overview

The S4 system in R is a system for object oriented programing. Confusingly, R has support for at least 3 different systems for object oriented programming: S3, S4 and S5 (also known as reference classes).

The S4 system is heavily used in Bioconductor, whereas it is very lightly used in "traditional" R and in packages from CRAN. As a user it can be useful to recognize S4 objects and to learn some facts about how to explore, manipulate and use the help system when encountering S4 classes and methods.

Important note for programmers

If you have experience with object oriented programming in other languages, for example java, you need to understand that in R, S4 objects and methods are completely separate. You can use S4 classes without every using S4 methods and vice versa.

[1] https://youtu.be/CeP-A__FroY
[2] https://youtu.be/wm-VCagXwj4

23.3 S3 and S4 classes

Based on years of experience in Bioconductor, it is fair to say that S4 classes have been very successful in this project. S4 classes has allowed us to construct rich and complicated data representations that nevertheless seems simple to the end user. An example, which we will return to, are the data containers `ExpressionSet` and `SummarizedExperiment`.

Let us look at a S3 object, the output of the linear model function `lm` in base R:

```
> df <- data.frame(y = rnorm(10), x = rnorm(10))
> lm.object <- lm(y ~ x, data = df)
> lm.object

Call:
lm(formula = y ~ x, data = df)

Coefficients:
(Intercept)            x
    -0.5459       0.4826
> names(lm.object)
 [1] "coefficients"  "residuals"     "effects"       "rank"
 [5] "fitted.values" "assign"        "qr"            "df.residual"
 [9] "xlevels"       "call"          "terms"         "model"
> class(lm.object)
[1] "lm"
```

In standard R, an S3 object is essentially a `list` with a `class` attribute on it. The problem with S3 is that we can assign any class to any list, which is nonsense. Let us try an example

```
> xx <- list(a = letters[1:3], b = rnorm(3))
> xx
$a
[1] "a" "b" "c"

$b
[1]  0.2504613  1.0273194 -1.2370133
> class(xx) <- "lm"
> xx

Call:
NULL

No coefficients
```

At least we don't get an error when we print it.

S4 classes have a formal definition and formal validity checking. To the end user, this gurantees validity of the object.

Let us load an S4 object:

```
> library(ALL)
> data(ALL)
> ALL
ExpressionSet (storageMode: lockedEnvironment)
assayData: 12625 features, 128 samples
  element names: exprs
protocolData: none
phenoData
  sampleNames: 01005 01010 ... LAL4 (128 total)
  varLabels: cod diagnosis ... date last seen (21 total)
  varMetadata: labelDescription
featureData: none
experimentData: use 'experimentData(object)'
  pubMedIds: 14684422 16243790
Annotation: hgu95av2
> class(ALL)
[1] "ExpressionSet"
attr(,"package")
[1] "Biobase"
> isS4(ALL)
[1] TRUE
```

The last function call checks whether the object is S4.

23.4 Constructors and getting help

The proper way of finding help on a class is to do one of the following

```
> ?"ExpressionSet-class"
> class?ExpressionSet
```

Note how you need to put the `ExpressionSet-class` in quotes.

A constructor function is a way to construct objects of the given class. You have already used constructor functions for base R classes, such as

```
> xx <- list(a = 1:3)
```

here `list()` is a constructor function. The Bioconductor coding standards suggests that an S4 class should have a name that begins with a capital letter and a constructor function with the same name as the class.

This is true for `ExpressionSet`:

```
> ExpressionSet()
ExpressionSet (storageMode: lockedEnvironment)
assayData: 0 features, 0 samples
  element names: exprs
protocolData: none
phenoData: none
featureData: none
experimentData: use 'experimentData(object)'
Annotation:
```

It is common that the constructor function is documented on the same help page as the class; this is why getting using

```
> ?ExpressionSet
```

works to give you detail on the class. *This does not always work.*

You can always use the function `new()` to construct an instance of a class. This is now frowned upon in Bioconductor, since it is not a good idea for complicated classes (... years of experience left out here). But in old documents on Bioconductor you can sometimes see calls like

```
> new("ExpressionSet")
ExpressionSet (storageMode: lockedEnvironment)
assayData: 0 features, 0 samples
  element names: exprs
protocolData: none
phenoData: none
featureData: none
experimentData: use 'experimentData(object)'
Annotation:
```

An example of a class in Bioconductor that does not have a constructor function is the `BSParams` class from *BSgenome* used for constructing calls to the `bsapply` function (applying functions over whole genomes).

23.5 Slots and accessor functions

You can get the class definition as

```
> getClass("ExpressionSet")
Class "ExpressionSet" [package "Biobase"]

Slots:

Name:      experimentData          assayData          phenoData
Class:              MIAME          AssayData AnnotatedDataFrame

Name:         featureData         annotation       protocolData
Class: AnnotatedDataFrame          character AnnotatedDataFrame

Name:    .__classVersion__
Class:           Versions

Extends:
Class "eSet", directly
Class "VersionedBiobase", by class "eSet", distance 2
Class "Versioned", by class "eSet", distance 3
```

In this output you'll see two things

1. A number of `slots` are mentioned together with a name and a `class`.
2. The class "extends" the class `eSet` "directly".

First, let us discuss (1). Data inside an S4 class are organized into slots. You access slots by using either '@' or the 'slots()' function, like

```
> ALL@annotation
[1] "hgu95av2"
> slot(ALL, "annotation")
[1] "hgu95av2"
```

However, as a user you **should never have to access slots directly**. This is important to understand. You should get data out of the class using "accessor" functions. Frequently accessor functions are named as the slot or perhaps `get` and the slot name.

```
> annotation(ALL)
[1] "hgu95av2"
```

(the `get` version of this name is `getAnnotation()` - different package authors use different styles). Not all slots have an accessor function, because slots may contain data which is not useful to the user.

Traditionally, accessor functions are documented on the same help page as the class itself.

Accessor functions does not always precisely refer to a slot. For example, for `ExpressionSet` we use `exprs()` to get the expression matrix, but there is no slot called `exprs` in the class definition. We still refer to `exprs()` as an accessor function.

By only using accessor functions you are protecting yourself (and your code) against future changes in the class definition; accessor functions should always work.

23.6 Class inheritance

Class inheritance is used a lot in Bioconductor. Class inheritance is used when you define a new class which "is almost like this other class but with a little twist". For example `ExpressionSet` inherits from `eSet`, and when you look at the class definition you cannot easily see a difference. The difference is that `ExpressionSet` is meant to contain expression data and has the `exprs()` accessor.

To make the usefulness of this more obvious, let me describe (briefly) the `MethylSet` class from the *minfi* (which I have authored). This class is very similar to `ExpressionSet` except it contains methylation data. Methylation is commonly measured using two channels "methylation" and "unmethylation" as opposed to the single channel exposed by `ExpressionSet`. Both `ExpressionSet` and `MethylSet` inherits from `eSet` (which actually represents most of the code of these classes) but `ExpressionSet` has a single `exprs()` accessor and `MethylSet` has two methylation accessors `getMeth()` and `getUnmeth()`.

This is useful to know because the documentation for a class might often refer to its parent class. For example, in the documentation for `ExpressionSet` you find the phrase "see `eSet`" a lot.

23.7 Outdated S4 classes

It occasionally happens that an S4 class definition gets updated. This might affect you in the following scenario

- You do an analysis in a given version of Bioconductor and you save your objects.
- 6 months later your work has to be revised, but Bioconductor has been updated in the meantime.
- When you `load` the old object, it doesn't seem to work.

The solution to this problem is the function `updateObject`. When a programmer updates their class definition, they are supposed to provide an `updateObject` function which will update old objects to new objects. Note the "supposed", this is not guranteed to happen, but feel free to report this as a bug if you encounter it.

Usage is easy

```
> new_object <- updateObject(old_object)
```

In practice, you tend to not want to keep the `old_object` around so you do

```
> object <- updateObject(object)
```

As an added hint, you can always run validity checking on an S4 objects if you think something funny is going on. It should return `TRUE`:

```
> validObject(ALL)
[1] TRUE
```

Notes on class version

In the early days of Bioconductor, efforts were made to version S4 classes. This was done in anticipation of changes in class definitions. This actually happens. For example, the `ExpressionSet` class has changed definition at least one time, and at the time of writing, the `SummarizedExperiment` class is undergoing changes to its internal structure between Bioconductor 3.1 and 3.2. It was later realized that we do seldom change class definitions, so the versioning was abandoned. You see debris from this in the `.__classVersion__` slot of the `ExpressionSet` class.

23.8 S4 Methods

You can think of S4 methods as simple functions. A method is a function which can look at its arguments and decide what to do. One way to mimic a method is by a function definition like the following

```
> mimicMethod <- function(x) {
+     if (is(x, "matrix"))
+         method1(x)
+     if (is(x, "data.frame"))
+         method2(x)
+     if (is(x, "IRanges"))
+         method3(x)
+ }
```

This function examines the `x` argument and runs different sets of code (`method1`, `method2`, `method3`) depending on which class `x` is.

An example of this is `as.data.frame`.

```
> as.data.frame
standardGeneric for "as.data.frame" defined from package "BiocGenerics"

function (x, row.names = NULL, optional = FALSE, ...)
standardGeneric("as.data.frame")
<environment: 0x7f9f81a1d108>
Methods may be defined for arguments: x
Use  showMethods("as.data.frame")  for currently available ones.
```

In the output you can see that it is so-called "generic" method involved something called `standardGeneric()`. Don't be distracted by this lingo; this is just like the `mimicMethod` function defined above. To see `method1`, `method2` etc you do

```
> showMethods("as.data.frame")
Function: as.data.frame (package BiocGenerics)
x="ANY"
x="DataFrame"
x="DataTable"
x="GappedRanges"
x="GenomicRanges"
x="GPos"
x="GroupedIRanges"
x="Hits"
x="List"
x="Pairs"
x="RangedData"
x="Ranges"
x="Rle"
x="Seqinfo"
x="Vector"
```

The different values of `x` here are called "signatures".

Actually, this does not show you the actual methods, it just shows you which values of `x` a method has been defined for. To see the code, do

```
> getMethod("as.data.frame", "DataFrame")
Method Definition:

function (x, row.names = NULL, optional = FALSE, ...)
{
    if (length(list(...)))
        warning("Arguments in '...' ignored")
    l <- as(x, "list")
    if (is.null(row.names))
        row.names <- rownames(x)
    if (!length(l) && is.null(row.names))
        row.names <- seq_len(nrow(x))
    l <- lapply(l, function(y) {
        if (is(y, "SimpleList") || is(y, "CompressedList"))
            y <- as.list(y)
        if (is.list(y))
            y <- I(y)
        y
    })
    IRanges.data.frame <- injectIntoScope(data.frame, as.data.frame)
    do.call(IRanges.data.frame, c(l, list(row.names = row.names),
        check.names = !optional, stringsAsFactors = FALSE))
}
<environment: namespace:S4Vectors>

Signatures:
        x
target  "DataFrame"
defined "DataFrame"
```

Lingo - `as.data.frame` is a generic method. It operatures on different signatures (values of `x`) and each signature has an associated method. This method is said to "dispatch" on `x`.

Many Bioconductor packages uses S4 classes extensively and S4 methods sparringly; I tend to follow this paradigm. S4 methods are particularly useful when

1. there are many different values if the argument which needs to be handled (like `as.data.frame` above.

2. you want to mimic functions from base R.

The second point is the case for, for example, the *IRanges* and*GenomicRanges* packages. The `IRanges` class looks very much like a standard vector and extensive work has gone into making it feel like a standard vector.

For `as.data.frame`, you can see that the value of this function in base R is not a method, by

```
> base::as.data.frame
function (x, row.names = NULL, optional = FALSE, ...)
{
    if (is.null(x))
        return(as.data.frame(list()))
    UseMethod("as.data.frame")
}
<bytecode: 0x7f9f808f5298>
<environment: namespace:base>
```

What happens is that the *BiocGenerics* converts the base R function `as.data.frame` into a generic method. This is what you get notified about when the following is printed when you load *BiocGenerics* (typically as by-product of loading another Biconductor package such as *IRanges*.

```
The following objects are masked from 'package:base':

    Filter, Find, Map, Position, Reduce, anyDuplicated, append,
    as.data.frame, as.vector, cbind, colnames, do.call, duplicated,
    eval, evalq, get, intersect, is.unsorted, lapply, mapply, match,
    mget, order, paste, pmax, pmax.int, pmin, pmin.int, rank, rbind,
    rep.int, rownames, sapply, setdiff, sort, table, tapply, union,
    unique, unlist, unsplit
```

There are drwabacks to methods:

1. It is hard (but not impossible) to get the actual code.
2. The help system can be confusing.
3. They are hard to debug for non-package authors.

We have addressed (1) above. The problem with the help system is that each method of `as.data.frame` may have its own help page, sometimes in different packages. Furthermore, each method may have different arguments.

The correct way to look up a help page for a method is

```
> method?"as.data.frame,DataFrame"
> ?"as.data.frame-method,DataFrame"
```

which is quite a mothful. This becomes worse when there is dispatching on multiple arguments; a great example is

```
> showMethods("findOverlaps")
Function: findOverlaps (package IRanges)
query="ANY", subject="Pairs"
query="GenomicRanges", subject="GenomicRanges"
query="GenomicRanges", subject="GRangesList"
query="GRangesList", subject="GenomicRanges"
query="GRangesList", subject="GRangesList"
query="integer", subject="Ranges"
query="Pairs", subject="ANY"
query="Pairs", subject="missing"
query="Pairs", subject="Pairs"
query="RangedData", subject="GenomicRanges"
query="RangedData", subject="RangedData"
query="RangedData", subject="RangesList"
query="Ranges", subject="Ranges"
query="RangesList", subject="RangedData"
query="RangesList", subject="RangesList"
query="Vector", subject="missing"
query="Vector", subject="Views"
query="Vector", subject="ViewsList"
query="Views", subject="Vector"
query="Views", subject="Views"
query="ViewsList", subject="Vector"
query="ViewsList", subject="ViewsList"
```

Finding the right help page for a method is (in my opinion) currently much harder than it ought to be; console yourself that many people struggle with this.

`findOverlaps` is also an example where two different methods of the generic have different arguments, although it becomes extremely confusing to illustrate how `findOverlaps` only accepts `ignore.strand` when the argument is a `GRanges` and not an `IRanges`. You cannot see it in the method arguments; you need to read the code itself (or the help page):

```
> getMethod("findOverlaps", signature(query = "Ranges", subject = "Ranges"))
Method Definition:

function (query, subject, maxgap = 0L, minoverlap = 1L, type = c("any",
    "start", "end", "within", "equal"), select = c("all", "first",
    "last", "arbitrary"), ...)
{
    .local <- function (query, subject, maxgap = 0L, minoverlap = 1L,
        type = c("any", "start", "end", "within", "equal"), select = c("all",
            "first", "last", "arbitrary"))
    {
        type <- match.arg(type)
        select <- match.arg(select)
        findOverlaps_NCList(query, subject, maxgap = maxgap,
            minoverlap = minoverlap, type = type, select = select)
    }
    .local(query, subject, maxgap, minoverlap, type, select,
        ...)
}
<environment: namespace:IRanges>

Signatures:
        query    subject
target  "Ranges" "Ranges"
defined "Ranges" "Ranges"
> getMethod("findOverlaps", signature(query = "GenomicRanges", subject = "Genomi\
cRanges"))
Method Definition:

function (query, subject, maxgap = 0L, minoverlap = 1L, type = c("any",
    "start", "end", "within", "equal"), select = c("all", "first",
    "last", "arbitrary"), ...)
{
    .local <- function (query, subject, maxgap = 0L, minoverlap = 1L,
        type = c("any", "start", "end", "within", "equal"), select = c("all",
            "first", "last", "arbitrary"), ignore.strand = FALSE)
    {
        type <- match.arg(type)
        select <- match.arg(select)
        findOverlaps_GNCList(query, subject, maxgap = maxgap,
            minoverlap = minoverlap, type = type, select = select,
            ignore.strand = ignore.strand)
```

```
    }
    .local(query, subject, maxgap, minoverlap, type, select,
        ...)
}
<environment: namespace:GenomicRanges>

Signatures:
        query           subject
target  "GenomicRanges" "GenomicRanges"
defined "GenomicRanges" "GenomicRanges"
```

This is (in some ways) a great illustration of how confusing methods can be! The good thing is that they tend to "just work".

24. Getting Data into Bioconductor

Watch a video[1] of this chapter.

24.1 Dependencies

This document has no dependencies.

24.2 Overview

How do you get your data into R/Bioconductor? The answer obviously depends on the file format of the data, but also what what you want to do with the data. Generally speaking, you need access to the data file and then you need to put the data into a relevant **data container**. Examples of data containers are `ExpressionSet` and `SummarizedExperiment`, but also classes such as `GRanges`.

Bioinformatics has jokingly been referred to as "The Science of Inventing New File Formats". This joke exemplifies the myriad of different file formats in use. Because we use many file formats and different types of data, it is hard to comprehensively cover all file formats and data types.

In general, a lot of useful solutions exists in domain / application specific packages. As an example of this paradigm, the *affxparser* package provides tools for parsing Affymetrix CEL files. However, this package is a parsing library and returns the data in a less useful representation. An end-user should instead use the *oligo* package which uses *affxparser* to read the data and then puts the data inside a useful data container; ready for downstream analysis.

24.3 Application Area

Microarray Data

Most microarray data is available to end users through a vendor specific file format such as CEL (Affymetrix) or IDAT (Illumina). These file formats can be read using vendor specific packages such as

- *affyio*
- *affxparser*

[1] https://youtu.be/mXg_YqSVpwM

- *illuminaio*

These packages are very low-level. In practice, many analysis specific packages supports import of these files into useful data structures, and you are much better off using one of those packages. For example

- *affy* for Affymetrix Gene Expression data.
- *oligo* for Affymetrix and Nimblegen expression and SNP array data.
- *lumi* for Illumina arrays.
- *minfi* for Illumina DNA methylation arrays (the 450k and 27k arrays).

High-throughput sequencing

Raw (unmapped) reads are typically available in the FASTQ format.

The first step in most analyses is mapping the reads onto a genome. For aligned reads, the BAM (SAM) format is now a clear standard.

However BAM (and SAM and FASTQ) files are quite big and still represents the data in a format which requires further processing before analysis. However, this further processing vary by application area (ChIP, RNA, DNA etc). Additionally, there are very few standard processed file formats; an example of such a standard format is BigWig. As an example of the lack of standards, there is still no standard file format representing RNA-seq reads summarized at the gene or transcript level; different pipelines provide different sorts of file. Luckily, these files are usually text files and can be read with standard tools for processing text files.ation from UCSC including UCSC tables can be accessed from the same package, for example by using the functions `getTable()` and `ucscTableQuery()`.

There is also support for parsing GFF (Genome File Format) in *rtracklayer*.

24.4 File types

FASTQ files

These file represent sequencing reads, often from an Illumina sequencer. See the *ShortRead* package.

BAM / SAM files

This fileformat contains reads aligned to a reference genome. See the `Biocpkg("Rsamtools")` package.

VCF files

VCF (Variant Call Format) files represents genotype files, typically produced by running a genotyping pipeline on high-throughout sequencing data. This format has a binary version called BCF. Use the functionality in *VariantAnnotation* to access these files.

UCSC Genome Browser formats

These formats include

- Wig and BigWig
- Bed and BigBed
- bedGraph

and can be read using the *rtracklayer* package, which also contains support for GFF files (annotation files).

Text files

An important special case is simple text files, either separated by `TAB` or `,` and then often named TSV (tab separated values) or CSV (comma separated values).

The base R function for reading these types of files is the versatile, but slow, `read.table()`. It has a large number of arguments, and can be customized to read most files. Pay attention to the following arguments

- `sep`: the separator
- `comment.char`: commenting out lines, for example header line.
- `colClasses`: if you know the class of the different columns in the file, you can speed up the function substantially.
- `quote`: the default value is `'"'` which can cause problems in genomics due to the use of 3â€™ and 5â€™.
- `row.names`, `col.names`
- `skip`, `nrows`, `fill`: reading part of the file.

For extremely complicated files you can use `readLines()` which reads the file into a character vector.

While `read.table()` is a classic, there are never, faster and more convenient functions which you should get to know.

The *readr*[2] package has functions `read_tsv()`, `read_csv()` and more general `read_delim()`. These functions are much faster than `read.table()` and support connections.

[2] http://cran.fhcrc.org/web/packages/readr/index.html

The *data.table*[3] package has the `fread()` function which is the fastest parser I know of, but is less flexible than the functions in *readr*[4].

24.5 Get data from databases of publicly available data

A number of data repositories have software packages dedicated to accessing the data inside of them:

- NCBI GEO[5] (Gene Expression Omnibus): the *GEOquery* package.
- NCBI SRA[6] (Short Read Archive): the *SRAdb* package.
- EBI ArrayExpress[7]: the *ArrayExpress* package.

[3] http://cran.fhcrc.org/web/packages/data.table/index.html
[4] http://cran.fhcrc.org/web/packages/readr/index.html
[5] http://www.ncbi.nlm.nih.gov/geo/
[6] http://www.ncbi.nlm.nih.gov/sra
[7] https://www.ebi.ac.uk/arrayexpress/

25. ShortRead

Watch a video[1] of this chapter.

25.1 Dependencies

This document has the following dependencies:

```
> library(ShortRead)
```

Use the following commands to install these packages in R.

```
> source("http://www.bioconductor.org/biocLite.R")
> biocLite(c("ShortRead"))
```

25.2 Overview

The *ShortRead* package contains functionality for reading and examining raw sequence reads (typically in FASTQ format).

25.3 ShortRead

The ShortRead package was one of the first Bioconductor packages to deal with low-level analysis of high-throughput sequencing data. Some of its functionality has now been superseded by other packages, but there is still relevant functionality left.

25.4 Reading FASTQ files

The FASTQ file format is the standard way of representing raw (unaligned) next generation sequencing reads, particular for the Illumina platform. The format basically consists of 4 lines per read, with the lines containing

1. Read name (sometimes includes flowcell ID or other information).

[1] https://youtu.be/dEDlER0ZNfA

2. Read nucleotides
3. Either empty or a repeat of line 1
4. Encoded read quality scores

Paired-end reads are stored in two separate files, where the reads are ordered the same (this is obviously fragile; what if reads are re-ordered in one file and not the other).

These files are read by `readFastq()` which produces an object of class `ShortReadQ`

```
> fastqDir <- system.file("extdata", "E-MTAB-1147", package = "ShortRead")
> fastqPath <- list.files(fastqDir, pattern = ".fastq.gz$", full = TRUE)[1]
> reads <- readFastq(fastqPath)
> reads
class: ShortReadQ
length: 20000 reads; width: 72 cycles
```

Here we directly point the function to the file path. A paradigm which is often used in Bioconductor is to first put the file path into an object which represents a specific file type and then read it; see

```
> fqFile <- FastqFile(fastqPath)
> fqFile
class: FastqFile
path: /Library/Frameworks/R.framework/Versio.../ERR127302_1_subset.fastq.gz
isOpen: FALSE
> reads <- readFastq(fqFile)
```

This appears to make little sense in this situation, but for really big files it makes sense to access them in chunks, see below for a BAM file example.

The `ShortReadQ` class is very similar to a `DNAStringSet` but it has two sets of strings: one for the read nucleotides and one for the base qualities. They are accessed as

```
> sread(reads)[1:2]
  A DNAStringSet instance of length 2
    width seq
[1]    72 GTCTGCTGTATCTGTGTCGGCTGTCTCGCGG...CAATGAAGGCCTGGAATGTCACTACCCCCAG
[2]    72 CTAGGGCAATCTTTGCAGCAATGAATGCCAA...GTGGCTTTTGAGGCCAGAGCAGACCTTCGGG
> quality(reads)[1:2]
class: FastqQuality
quality:
  A BStringSet instance of length 2
    width seq
```

```
[1]    72 HHHHHHHHHHHHHHHHHHHHHHEBDBB?B:BBG...FEFBDBD@DDECEE3>:?;@@@>?=BAB?##
[2]    72 IIIIHIIIGIIIIIIIHIIIIEGBGHIIIIH...IHIIIHIIIIIGIIIEGIIGBGE@DDGGGIG
> id(reads)[1:2]
  A BStringSet instance of length 2
    width seq
[1]    53 ERR127302.8493430 HWI-EAS350_0441:1:34:16191:2123#0/1
[2]    53 ERR127302.21406531 HWI-EAS350_0441:1:88:9330:2587#0/1
```

25.5 A word on quality scores

Note how the quality scores are listed as characters. You can convert them into standard 0-40 integer quality scores by

```
> as(quality(reads), "matrix")[1:2,1:10]
     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
[1,]   39   39   39   39   39   39   39   39   39    39
[2,]   40   40   40   40   39   40   40   40   38    40
```

In this conversion, each letter is matched to an integer between 0 and 40. This matching is known as the "encoding" of the quality scores and there has been different ways to do this encoding. Unfortunately, it is not stored in the FASTQ file which encoding is used, so you have to know or guess the encoding. The ShortRead package does this for you.

These numbers are supposed to related to the probability that the reported base is different from the template fragment (ie. a sequence error). One should be aware that this probabilistic interpretation is not always true; methods such as "quality-remapping" helps to ensure this.

25.6 Reading alignment files

In the early days of next generation sequencing, there was no standardized alignment output format. different aligners produced different output file, including Bowtie and MAQ. Later on, the SAM / BAM format was introduced and this is now the standard alignment output. ShortRead contains tools for reading these older alignment formats through the `readAligned()` function (the `type` argument support options such as `type="Bowtie"` and `type="MAQMap"` and `type="MAQMapShort"`).

The package has some very old support for parsing BAM files, but use *Rsamtools* and *GenomicAlignments* for this task instead.

25.7 Other Resources

- The vignettes from the ShortRead package[2].

[2] http://bioconductor.org/packages/ShortRead

26. Rsamtools

Watch a video[1] of this chapter.

26.1 Dependencies

This document has the following dependencies:

```
> library(Rsamtools)
```

Use the following commands to install these packages in R.

```
> source("http://www.bioconductor.org/biocLite.R")
> biocLite(c("Rsamtools"))
```

26.2 Overview

The *Rsamtools* packages contains functionality for reading and examining aligned reads in the BAM format.

26.3 Rsamtools

The Rsamtools package is an interface to the widely used `samtools/htslib` library. The main functionality of the package is support for reading BAM files.

26.4 The BAM / SAM file format

The SAM format is a text based representation of alignments. The BAM format is a binary version of SAM which is smaller and much faster. In general, always work with BAM.

The format is quite complicated, which means the R representation is also a bit complicated. This complication happens because of the following features of the file format

- It may contain unaligned sequences.

[1] https://youtu.be/3T4mDPQ5hU8

- Each sequence may be aligned to multiple locations.
- It supports spliced (split) alignments.
- It may contain reads from multiple samples.

We will not attempt to fully understanding all the intricacies of this format.

A BAM file can be sorted in multiple ways. If it is sorted according to genomic location and if it is also "indexed" it is possible to retrieve all reads mapping to a genomic location, something which can be very useful. In *Rsamtools* this is done by the functions `sortBam()` and `indexBam()`.

26.5 scanBam

How to read a BAM file goes conceptually like this

1. A pointer to the file is created by the `BamFile()` constructor.
2. (Optional) Parameters for which reads to report is constructed by `ScanBamParams()`.
3. The file is being read according to these parameters by `scanBam()`.

First we setup a `BamFile` object:

```
> bamPath <- system.file("extdata", "ex1.bam", package="Rsamtools")
> bamFile <- BamFile(bamPath)
> bamFile
class: BamFile
path: /Library/Frameworks/R.framework/Versions/3.3/Resources/lib.../ex1.bam
index: /Library/Frameworks/R.framework/Versions/3.3/Resource.../ex1.bam.bai
isOpen: FALSE
yieldSize: NA
obeyQname: FALSE
asMates: FALSE
qnamePrefixEnd: NA
qnameSuffixStart: NA
```

Some high-level information can be accessed here, like

```
> seqinfo(bamFile)
Seqinfo object with 2 sequences from an unspecified genome:
  seqnames seqlengths isCircular genome
  seq1           1575         NA   <NA>
  seq2           1584         NA   <NA>
```

(obviously, `seqinfo()` and `seqlevels()` etc. are supported as well).

Now we read all the reads in the file using `scanBam()`, ignoring the possibility of selecting reads using `ScanBamParams()` (we will return to this below).

```
> aln <- scanBam(bamFile)
> length(aln)
[1] 1
> class(aln)
[1] "list"
```

We get back a list of length 1; this is because `scanBam()` can return output from multiple genomic regions, and here we have only one (everything). We therefore subset the output; this again gives us a list and we show the information from the first alignment

```
> aln <- aln[[1]]
> names(aln)
 [1] "qname"  "flag"   "rname"  "strand" "pos"    "qwidth" "mapq"
 [8] "cigar"  "mrnm"   "mpos"   "isize"  "seq"    "qual"
> lapply(aln, function(xx) xx[1])
$qname
[1] "B7_591:4:96:693:509"

$flag
[1] 73

$rname
[1] seq1
Levels: seq1 seq2

$strand
[1] +
Levels: + - *

$pos
[1] 1
```

```
$qwidth
[1] 36

$mapq
[1] 99

$cigar
[1] "36M"

$mrnm
[1] <NA>
Levels: seq1 seq2

$mpos
[1] NA

$isize
[1] NA

$seq
  A DNAStringSet instance of length 1
    width seq
[1]    36 CACTAGTGGCTCATTGTAAATGTGTGGTTTAACTCG

$qual
  A PhredQuality instance of length 1
    width seq
[1]    36 <<<<<<<<<<<<<<;<<<<<<<<<5<<<<<;:<;7
```

Notice how the `scanBam()` function returns a basic R object, instead of an S4 class. Representing the alignments as S4 object is done by the *GenomicAlignments* package; this is especially useful for access to spliced alignments from RNA sequencing data.

The names of the `aln` list are basically the names used in the BAM specification. Here is a quick list of some important ones

- `qname`: The name of the read.
- `rname`: The name of the chromosome / sequence / contig it was aligned to.
- `strand`: The strand of the alignment.
- `pos`: The coordinate of the left-most part of the alignment.
- `qwidth`: The length of the read.

- `mapq`: The mapping quality of the alignment.
- `seq`: The actual sequence of the alignment.
- `qual`: The quality string of the alignment.
- `cigar`: The CIGAR string (below).
- `flag`: The flag (below).

26.6 Reading in parts of the file

BAM files can be extremely big and it is there often necessary to read in parts of the file. You can do this in different ways

1. Read a set number of records (alignments).
2. Only read alignments satisfying certain criteria.
3. Only read alignments in certain genomic regions.

Let us start with the first of this. By specifying `yieldSize` when you use `BamFile()`, every invocation of `scanBam()` will only read `yieldSize` number of alignments. You can then invoke `scanBam()` again to get the next set of alignments; this requires you to `open()` the file first (otherwise you will keep read the same alignments).

```
> yieldSize(bamFile) <- 1
> open(bamFile)
> scanBam(bamFile)[[1]]$seq
  A DNAStringSet instance of length 1
    width seq
[1]    36 CACTAGTGGCTCATTGTAAATGTGTGGTTTAACTCG
> scanBam(bamFile)[[1]]$seq
  A DNAStringSet instance of length 1
    width seq
[1]    35 CTAGTGGCTCATTGTAAATGTGTGGTTTAACTCGT
> ## Cleanup
> close(bamFile)
> yieldSize(bamFile) <- NA
```

The other two ways of reading in parts of a BAM file is to use `ScanBamParams()`, specifically the `what` and `which` arguments.

```
> gr <- GRanges(seqnames = "seq2",
+               ranges = IRanges(start = c(100, 1000), end = c(1500,2000)))
> params <- ScanBamParam(which = gr, what = scanBamWhat())
> aln <- scanBam(bamFile, param = params)
> names(aln)
[1] "seq2:100-1500"  "seq2:1000-2000"
> head(aln[[1]]$pos)
[1] 66 68 68 72 73 77
```

Notice how the `pos` is less than what is specified in the `which` argument; this is because the alignments overlap the `which` argument. The `what=scanBamWhat()` tells the function to read everything. Often, you may not be interested in the actual sequence of the read or its quality scores. These takes up a lot of space so you may consider disabling reading this information.

The CIGAR string

The "CIGAR" is how the BAM format represents spliced alignments. For example, the format stored the left most coordinate of the alignment. To get to the right coordinate, you have to parse the CIGAR string. In this example "36M" means that it has been aligned with no insertions or deletions. If you need to work with spliced alignments or alignments containing insertions or deletions, you should use the *GenomicAlignments* package.

Flag

An alignment may have a number of "flags" set or unset. These flags provide information about the alignment. The flag integer is a representation of multiple flags simultanously. An example of a flag is indicating (for a paired end alignment) whether both pairs have been properly aligned. For more information, see the BAM specification.

In *Rsamtools* there is a number of helper functions dealing with only reading certain flags; use these.

26.7 BAM summary

Sometimes you want a quick summary of the alignments in a BAM file:

```
> quickBamFlagSummary(bamFile)
                                group |    nb of |    nb of | mean / max
                                   of |  records |   unique | records per
                              records | in group |   QNAMEs | unique QNAME
All records..................... A |     3307 |     1699 | 1.95 / 2
  o template has single segment.... S |        0 |        0 |   NA / NA
  o template has multiple segments. M |     3307 |     1699 | 1.95 / 2
      - first segment.............. F |     1654 |     1654 |    1 / 1
      - last segment............... L |     1653 |     1653 |    1 / 1
      - other segment.............. O |        0 |        0 |   NA / NA

Note that (S, M) is a partitioning of A, and (F, L, O) is a partitioning of M.
Indentation reflects this.

Details for group M:
  o record is mapped.............. M1 |     3271 |     1699 | 1.93 / 2
      - primary alignment......... M2 |     3271 |     1699 | 1.93 / 2
      - secondary alignment....... M3 |        0 |        0 |   NA / NA
  o record is unmapped............ M4 |       36 |       36 |    1 / 1

Details for group F:
  o record is mapped.............. F1 |     1641 |     1641 |    1 / 1
      - primary alignment......... F2 |     1641 |     1641 |    1 / 1
      - secondary alignment....... F3 |        0 |        0 |   NA / NA
  o record is unmapped............ F4 |       13 |       13 |    1 / 1

Details for group L:
  o record is mapped.............. L1 |     1630 |     1630 |    1 / 1
      - primary alignment......... L2 |     1630 |     1630 |    1 / 1
      - secondary alignment....... L3 |        0 |        0 |   NA / NA
  o record is unmapped............ L4 |       23 |       23 |    1 / 1
```

26.8 Other functionality from Rsamtools

BamViews

Instead of reading a single file, it is possible to construct something called a `BamViews`, a link to multiple files. In many ways, it has the same `Views` functionality as other views. A quick example should suffice, first we read everything;

```
> bamView <- BamViews(bamPath)
> aln <- scanBam(bamView)
> names(aln)
[1] "ex1.bam"
```

This gives us an extra list level on the return object; first level is files, second level is ranges.

We can also set `bamRanges()` on the `BamViews` to specify that only certain ranges are read; this is similar to setting a `which` argument to `ScanBamParams()`.

```
> bamRanges(bamView) <- gr
> aln <- scanBam(bamView)
> names(aln)
[1] "ex1.bam"
> names(aln[[1]])
[1] "seq2:100-1500"  "seq2:1000-2000"
```

countBam

Sometimes, all you want to do is countâ€¦ use `countBam()` instead of `scanBam()`.

26.9 Other Resources

- The vignettes from the Rsamtools package[2].
- For representing more complicated alignments (specifically spliced alignments from RNA-seq), see the *GenomicAlignments* package.

[2] http://bioconductor.org/packages/Rsamtools

27. The oligo package

Watch a video[1] of this chapter.

27.1 Dependencies

This document has the following dependencies:

```
> library(oligo)
> library(GEOquery)
```

Use the following commands to install these packages in R.

```
> source("http://www.bioconductor.org/biocLite.R")
> biocLite(c("oligo", "GEOquery"))
```

27.2 Overview

This document presents the *oligo* package for handling Affymetrix and Nimblegen microarrays, especially gene expression, exon expression and SNP arrays.

27.3 Getting the data

We will use the dataset deposited as GEO accession number "GSE38792". In this dataset, the experimenters profiled fat biopsies from two different conditions: 10 patients with obstructive sleep apnea (OSA) and 8 healthy controls.

The profiling was done using the Affymetrix Human Gene ST 1.0 array.

First we need to get the raw data; this will be a set of binary files in CEL format. There will be one file per sample. The CEL files are accessible as supplementary information from GEO; we get the files using *GEOquery*.

[1] https://youtu.be/_pAtq0OC8Ro

```
> library(GEOquery)
> getGEOSuppFiles("GSE38792")
> list.files("GSE38792")
[1] "CEL"              "filelist.txt"     "GSE38792_RAW.tar"
> untar("GSE38792/GSE38792_RAW.tar", exdir = "GSE38792/CEL")
> list.files("GSE38792/CEL")
 [1] "GSM949164_Control1.CEL.gz" "GSM949166_Control2.CEL.gz"
 [3] "GSM949168_Control3.CEL.gz" "GSM949169_Control4.CEL.gz"
 [5] "GSM949170_Control5.CEL.gz" "GSM949171_Control6.CEL.gz"
 [7] "GSM949172_Control7.CEL.gz" "GSM949173_Control8.CEL.gz"
 [9] "GSM949174_OSA1.CEL.gz"     "GSM949175_OSA2.CEL.gz"
[11] "GSM949176_OSA3.CEL.gz"     "GSM949177_OSA4.CEL.gz"
[13] "GSM949178_OSA5.CEL.gz"     "GSM949179_OSA6.CEL.gz"
[15] "GSM949180_OSA7.CEL.gz"     "GSM949181_OSA8.CEL.gz"
[17] "GSM949182_OSA9.CEL.gz"     "GSM949183_OSA10.CEL.gz"
```

oligo and many other packages of its kind has convenience functions for reading in many files at once. In this case we construct a vector of filenames and feed it to `read.celfiles()`.

```
> library(oligo)
> celfiles <- list.files("GSE38792/CEL", full = TRUE)
> rawData <- read.celfiles(celfiles)
Reading in : GSE38792/CEL/GSM949164_Control1.CEL.gz
Reading in : GSE38792/CEL/GSM949166_Control2.CEL.gz
Reading in : GSE38792/CEL/GSM949168_Control3.CEL.gz
Reading in : GSE38792/CEL/GSM949169_Control4.CEL.gz
Reading in : GSE38792/CEL/GSM949170_Control5.CEL.gz
Reading in : GSE38792/CEL/GSM949171_Control6.CEL.gz
Reading in : GSE38792/CEL/GSM949172_Control7.CEL.gz
Reading in : GSE38792/CEL/GSM949173_Control8.CEL.gz
Reading in : GSE38792/CEL/GSM949174_OSA1.CEL.gz
Reading in : GSE38792/CEL/GSM949175_OSA2.CEL.gz
Reading in : GSE38792/CEL/GSM949176_OSA3.CEL.gz
Reading in : GSE38792/CEL/GSM949177_OSA4.CEL.gz
Reading in : GSE38792/CEL/GSM949178_OSA5.CEL.gz
Reading in : GSE38792/CEL/GSM949179_OSA6.CEL.gz
Reading in : GSE38792/CEL/GSM949180_OSA7.CEL.gz
Reading in : GSE38792/CEL/GSM949181_OSA8.CEL.gz
Reading in : GSE38792/CEL/GSM949182_OSA9.CEL.gz
Reading in : GSE38792/CEL/GSM949183_OSA10.CEL.gz
```

```
> rawData
GeneFeatureSet (storageMode: lockedEnvironment)
assayData: 1102500 features, 18 samples
  element names: exprs
protocolData
  rowNames: GSM949164_Control1.CEL.gz GSM949166_Control2.CEL.gz
    ... GSM949183_OSA10.CEL.gz (18 total)
  varLabels: exprs dates
  varMetadata: labelDescription channel
phenoData
  rowNames: GSM949164_Control1.CEL.gz GSM949166_Control2.CEL.gz
    ... GSM949183_OSA10.CEL.gz (18 total)
  varLabels: index
  varMetadata: labelDescription channel
featureData: none
experimentData: use 'experimentData(object)'
Annotation: pd.hugene.1.0.st.v1
```

This is in the form of an `GeneFeatureSet`; which is an `ExpressionSet`-like container. Knowing a bit of S4, we can see this through the class definition

```
> getClass("GeneFeatureSet")
Class "GeneFeatureSet" [package "oligoClasses"]

Slots:

Name:        manufacturer      intensityFile          assayData
Class:          character          character          AssayData

Name:           phenoData        featureData     experimentData
Class: AnnotatedDataFrame AnnotatedDataFrame              MIAxE

Name:          annotation       protocolData  .__classVersion__
Class:          character AnnotatedDataFrame           Versions

Extends:
Class "FeatureSet", directly
Class "NChannelSet", by class "FeatureSet", distance 2
Class "eSet", by class "FeatureSet", distance 3
Class "VersionedBiobase", by class "FeatureSet", distance 4
Class "Versioned", by class "FeatureSet", distance 5
```

We see that this is a special case of a `FeatureSet` which is a special case of `NChannelSet` which is an `eSet`. We can see the intensity measures by

```
> exprs(rawData)[1:4,1:3]
  GSM949164_Control1.CEL.gz GSM949166_Control2.CEL.gz
1                      9411                      9917
2                       255                       200
3                      9171                      9202
4                       229                       220
  GSM949168_Control3.CEL.gz
1                      8891
2                       181
3                      9266
4                       202
```

We see this is raw intensity data; the unit of measure is integer measurements on a 16 bit scanner, so we get values between 0 and $2^16=65,536$. This is easily verifiable:

```
> max(exprs(rawData))
[1] 65534
```

Note the large number of features in this dataset, more than 1 million. Because of the manufacturing technology, Affymetrix can only make very short oligos (around 25bp) but can make them cheaply and at high quality. The short oligos means that the binding specificity of the oligo is not very good. To compensate for this, Affymetrix uses a design where a gene is being measured by many different probes simultaneously; this is called a probeset. As part of the preprocessing step for Affymetrix arrays, the measurements for all probes in a probeset needs to be combined into one expression measure.

Let us clean up the phenotype information for `rawData`.

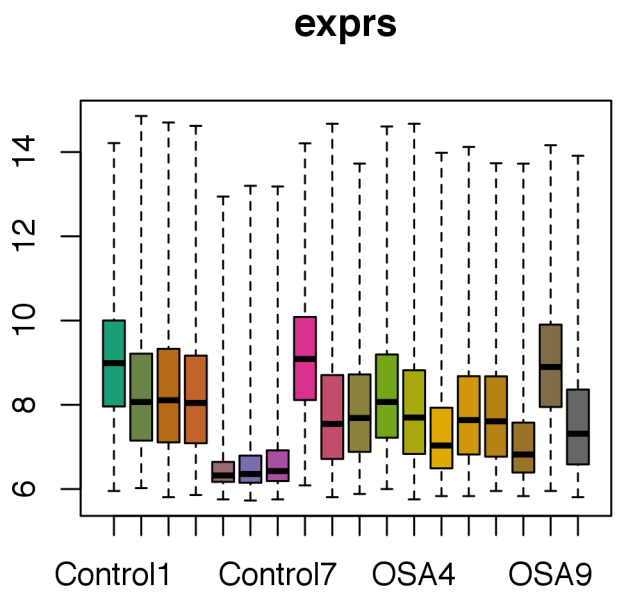
```
> filename <- sampleNames(rawData)
> pData(rawData)$filename <- filename
> sampleNames <- sub(".*_", "", filename)
> sampleNames <- sub(".CEL.gz$", "", sampleNames)
> sampleNames(rawData) <- sampleNames
> pData(rawData)$group <- ifelse(grepl("^OSA", sampleNames(rawData)),
+                                "OSA", "Control")
> pData(rawData)
         index                  filename   group
Control1     1 GSM949164_Control1.CEL.gz Control
Control2     2 GSM949166_Control2.CEL.gz Control
```

```
Control3     3 GSM949168_Control3.CEL.gz Control
Control4     4 GSM949169_Control4.CEL.gz Control
Control5     5 GSM949170_Control5.CEL.gz Control
Control6     6 GSM949171_Control6.CEL.gz Control
Control7     7 GSM949172_Control7.CEL.gz Control
Control8     8 GSM949173_Control8.CEL.gz Control
OSA1         9     GSM949174_OSA1.CEL.gz     OSA
OSA2        10     GSM949175_OSA2.CEL.gz     OSA
OSA3        11     GSM949176_OSA3.CEL.gz     OSA
OSA4        12     GSM949177_OSA4.CEL.gz     OSA
OSA5        13     GSM949178_OSA5.CEL.gz     OSA
OSA6        14     GSM949179_OSA6.CEL.gz     OSA
OSA7        15     GSM949180_OSA7.CEL.gz     OSA
OSA8        16     GSM949181_OSA8.CEL.gz     OSA
OSA9        17     GSM949182_OSA9.CEL.gz     OSA
OSA10       18    GSM949183_OSA10.CEL.gz     OSA
```

27.4 Normalization

Let us look at the probe intensities across the samples, using the `boxplot()` function.

```
> boxplot(rawData, target = "core")
```



Boxplots of the raw data.

Boxplots are great for comparing many samples because it is easy to display many box plots side by side. We see there is a large difference in both location and spread between samples. There are three samples with very low intensities; almost all probes have intensities less than 7 on the log2 scale. From experience with Affymetrix microarrays, I know this is an extremely low intensity. Perhaps the array hybridization failed for these arrays. To determine this will require more investigation.

A classic and powerful method for preprocessing Affymetrix gene expression arrays is the RMA method. Experience tells us that RMA essentially always performs well so many people prefer this method; one can argue that it is better to use a method which always does well as opposed to a method which does extremely well on some datasets and poorly on others.

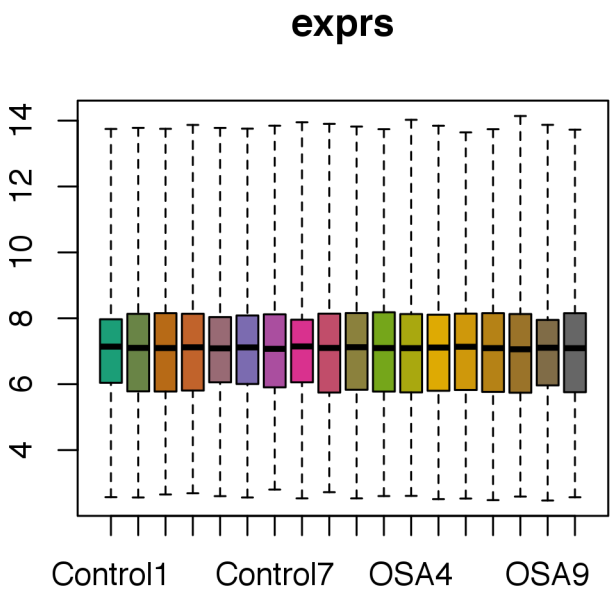
The RMA method was originally implemented in the *affy* package which has later been supplanted by the *oligo* package. The data we are analyzing comes from a "new" style Affymetrix array based on random priming; the *affy* package does not support these types of arrays. It is extremely easy to run RMA:

```
> normData <- rma(rawData)
Background correcting
Normalizing
Calculating Expression
> normData
ExpressionSet (storageMode: lockedEnvironment)
assayData: 33297 features, 18 samples
  element names: exprs
protocolData
  rowNames: Control1 Control2 ... OSA10 (18 total)
  varLabels: exprs dates
  varMetadata: labelDescription channel
phenoData
  rowNames: Control1 Control2 ... OSA10 (18 total)
  varLabels: index filename group
  varMetadata: labelDescription channel
featureData: none
experimentData: use 'experimentData(object)'
Annotation: pd.hugene.1.0.st.v1
```

Note how `normData` has on the order of 33k features which is closer to the number of genes in the human genome.

We can check the performance of RMA by looking at boxplots again.

```
> boxplot(normData)
```



Boxplots of the data preprocessed using RMA.

Here, it is important to remember that the first set of boxplots is at the probe level (\∼1M probes) whereas the second set of boxplots is at the probeset level (\∼33k probesets), so they display data at different summarization levels. However, what matters for analysis is that the probe distributions are normalized across samples and at a first glance it looks ok. One can see that the 3 suspicious samples from before still are slightly different, but that at least 2 more samples are similar to those.

For the normalization-interested person, note that while the distributions are similar, they are not identical despite the fact that RMA includes quantile normalization. This is because quantile normalization is done prior to probe summarization; if you quantile normalize different distributions they are guaranteed to have the same distribution afterwards.

The data is now ready for differential expression analysis.

27.5 Other Resources

- The vignettes from the oligo package[2].

[2] http://bioconductor.org/packages/oligo

28. limma

Watch a video[1] of this chapter.

28.1 Dependencies

This document has the following dependencies:

```
> library(limma)
> library(leukemiasEset)
```

Use the following commands to install these packages in R.

```
> source("http://www.bioconductor.org/biocLite.R")
> biocLite(c("limma", "leukemiasEset"))
```

28.2 Overview

limma is a very popular package for analyzing microarray and RNA-seq data.

LIMMA stands for "linear models for microarray data". Perhaps unsurprisingly, limma contains functionality for fitting a broad class of statistical models called "linear models". Examples of such models include linear regression and analysis of variance. While most of the functionality of limma has been developed for microarray data, the model fitting routines of limma are useful for many types of data, and is not limited to microarrays. For example, I am using limma in my research on analysis of DNA methylation.

28.3 Analysis Setup and Design

(The discussion in this section is not specific to limma.)

A very common analysis setup is having access to a matrix of numeric values representing some measurements; an example is gene expression. Traditionally in Bioconductor, and in computational biology more generally, columns of this matrix are samples and rows of the matrix are features. Features can be many things; in gene expression a feature is a gene. The feature by sample layout

[1] https://youtu.be/ZRet1oeGiUU

in Bioconductor is the transpose of the layout in classic statistics where the matrix is samples by features; this sometimes cause confusion.

A very common case of this type of data is gene expression data (either from microarrays or from RNA sequencing) where the features are individual genes.

Samples are usually few (usually less than a hundred, almost always less than a thousand) and are often grouped; arguably the most common setup is having samples from two different groups which we can call cases and controls. The objective of an analysis is frequently to discover which features (genes) are different between groups or stated differently: to discover which genes are differentially expressed between cases and controls. Of course, more complicated designs are also used; sometimes more than two groups are considered and sometimes there are additional important covariates such as age or sex. An example of a more complicated design is a time series experiment where each time point is a group and where it is sometimes important to account for the time elapsed between time points.

Broadly speaking, how samples are distributed between groups determines the **design** of the study. In addition to the design, there is one or more question(s) of interest(s) such as the difference between two groups. Such questions are usually formalized as **contrasts**; an example of a contrast is indeed the difference between two groups.

Samples are usually assumed to be independent but are sometimes paired; an example of pairing is when a normal and a cancer sample from the same patient is available. Pairing allows for more efficient inference because each sample has a sample specific control. If only some samples are paired, or if multiple co-linked samples exists, it becomes harder (but usually possible) to account for this structure in the statistical model.

As stated above, the most common design is a two group design with unpaired samples.

Features are often genes or genomic intervals, for example different promoters or genomic bins. The data is often gene expression data but could be histone modification abundances or even measurements from a Hi-C contact matrix.

Common to all these cases is the rectangular data structure (the matrix) with samples on columns and features on rows. This is exactly the data structure which is represented by an `ExpressionSet` or a `SummarizedExperiment`.

A number of different packages allows us to fit common types of models to this data structure

- *limma* fits a so-called linear model; examples of linear models are (1) linear regression, (2) multiple linear regression and (3) analysis of variance.
- *edgeR*, *DESeq* and *DESeq2* fits generalized linear models, specifically models based on the negative binomial distribution.

Extremely simplified, *limma* is useful for continuous data such as microarray data and *edgeR* / *DESeq* / *DESeq2* are useful for count data such as high-throughput sequencing. But that is a very simplified statement.

In addition to the distributional assumptions, all of these packages uses something called empirical Bayes techniques to borrow information across features. As stated above, usually the number of samples is small and the number of features is large. It has been shown time and time again that you can get better results by borrowing information across features, for example by modeling a mean-variance relationship. This can be done in many ways and often depends on the data characteristics of a specific type of data. For example both edgeR and DESeq(2) are very popular in the analysis of RNA-seq data and all three packages uses models based on the negative binomial distribution. For a statistical point of view, a main difference between these packages (models) is how they borrow information across genes.

Fully understanding these classes of models as well as their strengths and limitations are beyond our scope. But we will still introduce aspects of these packages because they are so widely used.

28.4 A two group comparison

Obtaining data

Let us use the `leukemiasEset` dataset from the *leukemiasEset*[2] package; this is an `ExpressionSet`.

```
> library(leukemiasEset)
> data(leukemiasEset)
> leukemiasEset
ExpressionSet (storageMode: lockedEnvironment)
assayData: 20172 features, 60 samples
  element names: exprs, se.exprs
protocolData
  sampleNames: GSM330151.CEL GSM330153.CEL ... GSM331677.CEL (60
    total)
  varLabels: ScanDate
  varMetadata: labelDescription
phenoData
  sampleNames: GSM330151.CEL GSM330153.CEL ... GSM331677.CEL (60
    total)
  varLabels: Project Tissue ... Subtype (5 total)
  varMetadata: labelDescription
featureData: none
experimentData: use 'experimentData(object)'
Annotation: genemapperhgu133plus2
> table(leukemiasEset$LeukemiaType)
```

[2] http://bioconductor.org/packages/leukemiasEset

```
ALL AML CLL CML NoL
 12  12  12  12  12
```

This is data on different types of leukemia. The code `NoL` means not leukemia, ie. normal controls.

Let us ask which genes are differentially expressed between the `ALL` type and normal controls. First we subset the data and clean it up

```
> ourData <- leukemiasEset[, leukemiasEset$LeukemiaType %in% c("ALL", "NoL")]
> ourData$LeukemiaType <- factor(ourData$LeukemiaType)
```

A linear model

Now we do a standard limma model fit

```
> design <- model.matrix(~ ourData$LeukemiaType)
> fit <- lmFit(ourData, design)
> fit <- eBayes(fit)
> topTable(fit)
                    logFC  AveExpr        t      P.Value    adj.P.Val
ENSG00000163751 4.089587 5.819472 22.51729 9.894742e-18 1.733025e-13
ENSG00000104043 4.519488 5.762115 21.98550 1.718248e-17 1.733025e-13
ENSG00000008394 5.267835 7.482490 20.08250 1.374231e-16 9.240332e-13
ENSG00000165140 3.206807 6.560163 19.41855 2.959391e-16 1.492421e-12
ENSG00000204103 4.786273 7.774809 19.04041 4.628812e-16 1.867448e-12
ENSG00000145569 2.845963 5.958707 18.46886 9.239404e-16 3.067090e-12
ENSG00000038427 5.047670 6.496822 18.35375 1.064328e-15 3.067090e-12
ENSG00000173391 4.282498 5.293222 17.89645 1.881511e-15 4.744229e-12
ENSG00000138449 5.295928 6.999716 17.79655 2.134448e-15 4.784010e-12
ENSG00000105352 2.521351 7.054018 17.62423 2.657074e-15 5.359850e-12
                       B
ENSG00000163751 30.15253
ENSG00000104043 29.65066
ENSG00000008394 27.73589
ENSG00000165140 27.02056
ENSG00000204103 26.60138
ENSG00000145569 25.95084
ENSG00000038427 25.81729
ENSG00000173391 25.27803
ENSG00000138449 25.15836
ENSG00000105352 24.95031
```

What happens here is a common limma (and friends) workflow. First, the comparison of interest (and the design of the experiment) is defined through a so-called "design matrix". This matrix basically encompasses everything we know about the design; in this case there are two groups (we have more to say on the design below). Next, the model is fitted. This is followed by borrowing strength across genes using a so-called empirical Bayes procedure (this is the step in limma which really works wonders). Because this design only has two groups there is only one possible comparison to make: which genes differs between the two groups. This question is examined by the `topTable()` function which lists the top differentially expressed genes. In a more complicated design, the `topTable()` function would need to be told which comparison of interest to summarize.

An important part of the output is `logFC` which is the log fold-change. To interpret the sign of this quantity you need to know if this is `ALL-NoL` (in which case positive values are up-regulated in ALL) or the reverse. In this case this is determined by the reference level which is the first level of the factor.

```
> ourData$LeukemiaType
 [1] ALL ALL ALL ALL ALL ALL ALL ALL ALL ALL ALL ALL NoL NoL NoL NoL NoL
[18] NoL NoL NoL NoL NoL NoL NoL
Levels: ALL NoL
```

we see the reference level is `ALL` so positive values means it is down-regulated in cancer. You can change the reference level of a factor using the `relevel()` function. You can also confirm this by computing the `logFC` by hand, which is useful to know. Let's compute the fold-change of the top differentially expressed gene:

```
> topTable(fit, n = 1)
                   logFC  AveExpr        t      P.Value    adj.P.Val
ENSG00000163751 4.089587 5.819472 22.51729 9.894742e-18 1.733025e-13
                       B
ENSG00000163751 30.15253
> genename <- rownames(topTable(fit, n=1))
> typeMean <- tapply(exprs(ourData)[genename,], ourData$LeukemiaType, mean)
> typeMean
     ALL      NoL
3.774678 7.864265
> typeMean["NoL"] - typeMean["ALL"]
     NoL
4.089587
```

confirming the statement. It is sometimes useful to check things by hand to make sure you have the right interpretation. Finally, note that limma doesn't do anything different from a difference of means when it computes `logFC`; all the statistical improvements centers on computing better t-statistics and p-values.

The reader who has some experience with statistics will note that all we are doing is comparing two groups; this is the same setup as the classic t-statistic. What we are computing here is indeed a t-statistic, but one where the variance estimation (the denominator of the t-statistics) is *moderated* by borrowing strength across genes (this is what `eBayes()` does); this is called a moderated t-statistic.

The output from `topTable()` includes

- `logFC`: the log fold-change between cases and controls.
- `t`: the t-statistic used to assess differential expression.
- `P.Value`: the p-value for differential expression; this value is not adjusted for multiple testing.
- `adj.P.Val`: the p-value adjusted for multiple testing. Different adjustment methods are available, the default is Benjamini-Horchberg.

How to setup and interpret a design matrix for more complicated designs is beyond the scope of this course. The limma User's Guide is extremely helpful here. Also, note that setting up a design matrix for an experiment is a standard task in statistics (and requires very little knowledge about genomics), so other sources of help is a local, friendly statistician or text books on basic statistics.

28.5 More on the design

In the analysis in the preceding section we setup our model like this

```
> design <- model.matrix(~ ourData$LeukemiaType)
```

and the we use F-statistics to get at our question of interest. We can it this easily because there is only really one interesting question for this design: is there differential expression between the two groups. But this formulation did not use contrasts; in the "Analysis Setup and Design" section we discussed how one specifies the question of interest using contrasts and we did not really do this here.

Let's try. A contrast is interpreted relative to the design matrix one uses. One conceptual design may be represented by different design matrices, which is one of the reasons why design matrices and contrasts take a while to absorb.

Let's have a look

```
> head(design)
  (Intercept) ourData$LeukemiaTypeNoL
1           1                       0
2           1                       0
3           1                       0
4           1                       0
5           1                       0
6           1                       0
```

This matrix has two columns because there are two parameters in this conceptual design: the expression level in each of the two groups. In this **parametrization** column 1 represents the expression of the ALL group and column 2 represents the difference in expression level from the NoL group to the ALL group. Testing that the two groups have the same expression level is done by testing whether the second parameter (equal to the difference in expression between the two groups) is equal to zero.

A different parametrization is

```
> design2 <- model.matrix(~ ourData$LeukemiaType - 1)
> head(design2)
  ourData$LeukemiaTypeALL ourData$LeukemiaTypeNoL
1                       1                       0
2                       1                       0
3                       1                       0
4                       1                       0
5                       1                       0
6                       1                       0
> colnames(design2) <- c("ALL", "NoL")
```

In this design, the two parameters corresponding to the two columns of the design matrix, represents the expression levels in the two groups. And the scientific question gets translated into asking whether or not these two parameters are the same. Let us see how we form a contrast matrix for this

```
> fit2 <- lmFit(ourData, design2)
> contrast.matrix <- makeContrasts("ALL-NoL", levels = design2)
> contrast.matrix
      Contrasts
Levels ALL-NoL
   ALL       1
   NoL      -1
```

Here we say we are interested in ALL-NoL which has the opposite sign of what we were doing above (where it was NoL-ALL; since NoL is the natural reference group this makes a lot more sense. Now we fit

```
> fit2C <- contrasts.fit(fit2, contrast.matrix)
> fit2C <- eBayes(fit2C)
> topTable(fit2C)
                    logFC  AveExpr         t      P.Value    adj.P.Val
ENSG00000163751 -4.089587 5.819472 -22.51729 9.894742e-18 1.733025e-13
ENSG00000104043 -4.519488 5.762115 -21.98550 1.718248e-17 1.733025e-13
ENSG00000008394 -5.267835 7.482490 -20.08250 1.374231e-16 9.240332e-13
ENSG00000165140 -3.206807 6.560163 -19.41855 2.959391e-16 1.492421e-12
ENSG00000204103 -4.786273 7.774809 -19.04041 4.628812e-16 1.867448e-12
ENSG00000145569 -2.845963 5.958707 -18.46886 9.239404e-16 3.067090e-12
ENSG00000038427 -5.047670 6.496822 -18.35375 1.064328e-15 3.067090e-12
ENSG00000173391 -4.282498 5.293222 -17.89645 1.881511e-15 4.744229e-12
ENSG00000138449 -5.295928 6.999716 -17.79655 2.134448e-15 4.784010e-12
ENSG00000105352 -2.521351 7.054018 -17.62423 2.657074e-15 5.359850e-12
                       B
ENSG00000163751 30.15253
ENSG00000104043 29.65066
ENSG00000008394 27.73589
ENSG00000165140 27.02056
ENSG00000204103 26.60138
ENSG00000145569 25.95084
ENSG00000038427 25.81729
ENSG00000173391 25.27803
ENSG00000138449 25.15836
ENSG00000105352 24.95031
```

Note that this is exactly the same output from `topTable()` as above, except for the sign of the `logFC` column.

28.6 Background: Data representation in limma

As we see above, *limma* works directly on `ExpressionSets`. It also works directly on matrices. But limma also have a class `RGList` which represents a two-color microarray. The basic data stored in this class is very `ExpressionSet`-like, but it has at least two matrices of expression measurements `R` (Red) and `G` (Green) and optionally two additional matrices of background estimates (`Rb` and `Gb`). It has a slot called `genes` which is basically equivalent to `featureData` for `ExpressionSets` (ie. information about which genes are measured on the microarray) as well as a `targets` slot which is basically the `pData` information from `ExpressionSet`.

28.7 Background: The targets file

limma introduced the concept of a so-called `targets` file. This is just a simple text file (usually TAB or comma-separated) which holds the phenotype data. The idea is that it is easier for many users to create this text file in a spreadsheet program, and then read it into R and stored the information in the data object.

28.8 Other Resources

The limma User's Guide from the limma webpage[3]. This is an outstanding piece of documentation which has (rightly) been called "the best resource on differential expression analysis available".

[3] http://bioconductor.org/packages/limma

29. Analysis of 450k DNA methylation data with minfi

Watch a video[1] of this chapter.

29.1 Dependencies

This document has the following dependencies:

```
> library(minfi)
> library(GEOquery)
> library(IlluminaHumanMethylation450kmanifest)
> library(IlluminaHumanMethylation450kanno.ilmn12.hg19)
```

Use the following commands to install these packages in R.

```
> source("http://www.bioconductor.org/biocLite.R")
> biocLite(c("minfi", "GEOquery"))
```

29.2 Overview

The Illumina 450k DNA methylation microarray is a cheap and relatively comprehensive array for assaying DNA methylation. It is the platform of choice for large sample profiling of DNA methylation, especially for so-called EWAS (Epigenome-wide association studies). The studies are like GWAS (genome-wide association studies) but instead of associated a phenotype (like disease) with genotype (typically measured on SNP arrays or exome/whole-genome sequencing) they associated phenotype with epigenotype.

29.3 DNA methylation

DNA methylation is a chemical modification of DNA. In humans, it occurs at CpG dinucleotides where the 'C' can be methylated or not. The methylation state of a given locus in a single cell is binary (technically tertiary since we have two copies of most chromosomes) but we measure DNA

[1] https://youtu.be/0llfypt9FAM

methylation across a population of cells. We observe that some loci has intermediate methylation values (between 0 and 1) and we use the methylation percentage (or Beta-value) to describe this.

The goal of most analyses of DNA methylation is to associate changes in phenotype with changes in methylation in one or more loci.

29.4 Array Design

The 450k array has a very unusual design, which to some extent impact analysis. It is really a mixture of a two-color array and two one-color arrays. There are two main types of probes (type I and type II) and the probe design affects the signal distribution of the probe.

The raw data format for the 450k array is known as IDAT. Because the array is measured in two different colors, there are two files for each sample, typically with the extention `_Grn.idat` and `_Red.idat`. Illumina's software suite for analysis of this array is called GenomeStudio. It is not unusual for practitioners to only have access to processed data from GenomeStudio instead of the raw IDAT files, but I and others have shown that there is information in the IDAT files which are beneficial to analysis.

29.5 Data

We will access a dataset created with the intention of studying acute mania. Serum samples were obtained from individuals hospitalized with acute mania as well as unaffected controls.

We want to obtain the IDAT files which are available as supplementary data. Far from all 450k datasets on GEO has IDAT files available. First we download the files.

```
> library(GEOquery)
> getGEOSuppFiles("GSE68777")
> untar("GSE68777/GSE68777_RAW.tar", exdir = "GSE68777/idat")
> head(list.files("GSE68777/idat", pattern = "idat"))
[1] "GSM1681154_5958091019_R03C02_Grn.idat"
[2] "GSM1681154_5958091019_R03C02_Grn.idat.gz"
[3] "GSM1681154_5958091019_R03C02_Red.idat"
[4] "GSM1681154_5958091019_R03C02_Red.idat.gz"
[5] "GSM1681155_5935446005_R05C01_Grn.idat"
[6] "GSM1681155_5935446005_R05C01_Grn.idat.gz"
```

Currently *minfi* does not support reading compressed IDAT files. This is clearly a needed functionality and (as the maintainer of this package) I will address this. But for now we will need to decompress the files.

```
> idatFiles <- list.files("GSE68777/idat", pattern = "idat.gz$", full = TRUE)
> head(sapply(idatFiles, gunzip, overwrite = TRUE), n = 3)
GSE68777/idat/GSM1681154_5958091019_R03C02_Grn.idat.gz
                                                8091452
GSE68777/idat/GSM1681154_5958091019_R03C02_Red.idat.gz
                                                8091452
GSE68777/idat/GSM1681155_5935446005_R05C01_Grn.idat.gz
                                                8091452
```

Now we read the IDAT files using `read.450k.exp()` which (in this case) reads all the IDAT files in a directory.

```
> rgSet <- read.450k.exp("GSE68777/idat")
> rgSet
RGChannelSet (storageMode: lockedEnvironment)
assayData: 622399 features, 40 samples
  element names: Green, Red
An object of class 'AnnotatedDataFrame': none
Annotation
  array: IlluminaHumanMethylation450k
  annotation: ilmn12.hg19
> pData(rgSet)
data frame with 0 columns and 40 rows
> head(sampleNames(rgSet))
[1] "GSM1681154_5958091019_R03C02" "GSM1681155_5935446005_R05C01"
[3] "GSM1681156_5958091020_R01C01" "GSM1681157_5958091020_R03C02"
[5] "GSM1681158_5935403032_R05C01" "GSM1681159_5958091019_R04C02"
```

Now we have the data, but note that we have no pheno data. And the filenames are very unhelpful here. These names consists of a GEO identifier (the `GSM` part) followed by a standard IDAT naming convention with a 10 digit number which is an array identifier followed by an identifier of the form `R01C01`. This is because each array actually allows for the hybridization of 12 samples in a 6x2 arrangement. The `5958091020_R01C0` means row 1 and column 1 on chip `5958091020`. This is all good, but does not help us understand which samples are cases and which are controls.

We now get the standard GEO representation to get the phenotype data stored in GEO. Most of the columns in this phenotype data are irrelevant (contains data such as the address of the person who submitted the data); we keep the useful ones. Then we clean it.

```
> geoMat <- getGEO("GSE68777")
> pD.all <- pData(geoMat[[1]])
> pD <- pD.all[, c("title", "geo_accession", "characteristics_ch1.1", "character\
istics_ch1.2")]
> head(pD)
                      title geo_accession characteristics_ch1.1
GSM1681154 5958091019_R03C02    GSM1681154      diagnosis: Mania
GSM1681155 5935446005_R05C01    GSM1681155      diagnosis: Mania
GSM1681156 5958091020_R01C01    GSM1681156        diagnosis: Ctr
GSM1681157 5958091020_R03C02    GSM1681157        diagnosis: Ctr
GSM1681158 5935403032_R05C01    GSM1681158      diagnosis: Mania
GSM1681159 5958091019_R04C02    GSM1681159      diagnosis: Mania
           characteristics_ch1.2
GSM1681154          Sex: Female
GSM1681155          Sex: Female
GSM1681156            Sex: Male
GSM1681157          Sex: Female
GSM1681158          Sex: Female
GSM1681159            Sex: Male
> names(pD)[c(3,4)] <- c("group", "sex")
> pD$group <- sub("^diagnosis: ", "", pD$group)
> pD$sex <- sub("^Sex: ", "", pD$sex)
```

We now need to merge this pheno data into the methylation data. To do so, we need a common sample identifier and we make sure we re-order the phenotype data in the same order as the methylation data. Finally we put the phenotype data inside the methylation data.

```
> sampleNames(rgSet) <- sub(".*_5", "5", sampleNames(rgSet))
> rownames(pD) <- pD$title
> pD <- pD[sampleNames(rgSet),]
> pData(rgSet) <- pD
> rgSet
RGChannelSet (storageMode: lockedEnvironment)
assayData: 622399 features, 40 samples
  element names: Green, Red
An object of class 'AnnotatedDataFrame'
  sampleNames: 5958091019_R03C02 5935446005_R05C01 ...
    5935403032_R04C01 (40 total)
  varLabels: title geo_accession group sex
  varMetadata: labelDescription
Annotation
  array: IlluminaHumanMethylation450k
  annotation: ilmn12.hg19
```

29.6 Preprocessing

The `rgSet` object is a class called `RGChannelSet` which represents two color data with a green and a red channel, very similar to an `ExpressionSet`.

The first step is usually to preprocess the data, using a number of functions including

- `preprocessRaw()` : do nothing.
- `preprocessIllumina()` : use Illumina's standard processing choices.
- `preprocessQuantile()` : use a version of quantile normalization adapted to methylation arrays.
- `preprocessNoob()` : use the NOOB background correction method.
- `preprocessSWAN()` : use the SWAN method.
- `preprocessFunnorm()` : use functional normalization.

These functions output different types of objects.

The class hierarchy in minfi is as follows: data can be stored in an Methylation and Unmethylation channel or in a percent methylation (called Beta) channel. For the first case we have the class `MethylSet`, for the second case we have the class `RatioSet`. When you have methylation / unmethylation values you can still compute Beta values on the fly. You convert from a `MethylSet` to a `RatioSet` with `ratioConvert()`.

In addition to these two classes, we have `GenomicMethylSet` and `GenomicRatioSet`. The `Genomic` indicates that the data has been associated with genomic coordinates using the `mapToGenome()` function.

The starting point for most analyses ought to be a `GenomicRatioSet` class. If your preprocessing method of choice does not get you there, use `ratioConvert()` and `mapToGenome()` to go the last steps.

Let us run `preprocessQuantile()` which arrives at a `GenomicRatioSet`:

```
> grSet <- preprocessQuantile(rgSet)
[preprocessQuantile] Mapping to genome.
[preprocessQuantile] Fixing outliers.
[preprocessQuantile] Quantile normalizing.
> grSet
class: GenomicRatioSet
dim: 485512 40
metadata(0):
assays(2): M CN
rownames(485512): cg13869341 cg14008030 ... cg08265308 cg14273923
rowData names(0):
```

```
colnames(40): 5958091019_R03C02 5935446005_R05C01 ...
  5958091020_R02C02 5935403032_R04C01
colData names(5): title geo_accession group sex predictedSex
Annotation
  array: IlluminaHumanMethylation450k
  annotation: ilmn12.hg19
Preprocessing
  Method: Raw (no normalization or bg correction)
  minfi version: 1.18.2
  Manifest version: 0.4.0
```

This is like a `SummarizedExperiment`; we can get the location of the CpGs by

```
> granges(grSet)
GRanges object with 485512 ranges and 0 metadata columns:
             seqnames              ranges strand
                <Rle>           <IRanges>  <Rle>
  cg13869341     chr1       [15865, 15865]      *
  cg14008030     chr1       [18827, 18827]      *
  cg12045430     chr1       [29407, 29407]      *
  cg20826792     chr1       [29425, 29425]      *
  cg00381604     chr1       [29435, 29435]      *
         ...      ...                  ...    ...
  cg17939569     chrY [27009430, 27009430]      *
  cg13365400     chrY [27210334, 27210334]      *
  cg21106100     chrY [28555536, 28555536]      *
  cg08265308     chrY [28555550, 28555550]      *
  cg14273923     chrY [28555912, 28555912]      *
  -------
  seqinfo: 24 sequences from hg19 genome; no seqlengths
```

The usual methylation measure is called "Beta" values; equal to percent methylation and defined as `Meth` divided by `Meth + Unmeth`.

```
> getBeta(grSet)[1:3,1:3]
           5958091019_R03C02 5935446005_R05C01 5958091020_R01C01
cg13869341         0.7485333         0.7696497         0.7322275
cg14008030         0.5300410         0.5893653         0.6044354
cg12045430         0.0912596         0.1007678         0.1057603
```

CpGs forms clusters known as "CpG Islands". Areas close to CpG Islands are known as CpG Shores, followed by CpG Shelfs and finally CpG Open Sea probes. An easy way to get at this is to use

```
> head(getIslandStatus(grSet))
[1] "OpenSea" "OpenSea" "Island"  "Island"  "Island"  "OpenSea"
```

29.7 Differential Methylation

Once the data has been normalized, one possibility is to identify differentially methylated CpGs by using *limma* on the Beta values.

Another possibility is to look for clusters of CpGs all changing in the same direction. One method for doing this is through the `bumphunter()` function which interfaces to the *bumphunter* package.

29.8 Other Resources

The vignette from the minfi package[2].

[2] http://bioconductor.org/packages/minfi

30. Count Based RNA-seq analysis

Watch a video[1] of this chapter.

30.1 Dependencies

This document has the following dependencies:

```
> library(DESeq2)
> library(edgeR)
> library(airway)
```

Use the following commands to install these packages in R.

```
> source("http://www.bioconductor.org/biocLite.R")
> biocLite(c("DESeq2", "edgeR", "airway"))
```

30.2 Overview

RNA seq data is often analyzed by creating a count matrix of gene counts per sample. This matrix is analyzed using count-based models, often built on the negative binomial distribution. Popular packages for this includes *edgeR* and *DESeq* / *DESeq2*.

This type of analysis discards part of the information in the RNA sequencing reads, but we have a good understanding of how to analyze this type of data.

30.3 RNA-seq count data

One simple way of analyzing RNA sequencing data is to make it look like microarray data. This is done by counting how many reads in each sample overlaps a gene. There are many ways to do this. It obviously depends on the annotation used, but also on how it is decided that a read overlaps a region. Of specific concern is which genomic regions are part of a gene with multiple transcripts.

There are no consensus on this process and the different choices one make is known to affect the outcome.

Tools for doing gene counting includes

[1] https://youtu.be/zAXHn_Y_NhI

- by using `featureCounts()` from the *Rsubread* package.
- the HTSeq[2] package (this is a python package, not a Bioconductor package).
- by using `summarizeOverlaps()` from the *GenomicAlignments* package.

and there are other alternatives. Many people seem to write their own counting pipeline.

Reducing RNA sequencing data to a single integer per gene is obvious a simplification. Indeed it ignores some of the main reasons for doing RNA sequencing, including assessing alternative splicing. On the other hand, we understand the statistical properties of this procedure well, and it delivers a basic insight into something that most researcher wants to know. Finally, this approach requires the different genomic regions to be known beforehand.

30.4 Statistical issues

In RNA-seq data analysis we often see that many genes (up to 50%) have little or no expression. It is common to pre-filter (remove) these genes prior to analysis. In general genomics filtering might be beneficial to your analysis, but this discussion is outside the scope of this document.

Note: The analysis presented below is **extremely** superficial. Consider this a very quick introduction to the workflow of these two packages.

30.5 The Data

We will be using the *airway*[3] dataset which contains RNA-seq data in the form of a `SummarizedExperiment`. Lets load the data and have a look

```
> library(airway)
> data(airway)
> airway
class: RangedSummarizedExperiment
dim: 64102 8
metadata(1): ''
assays(1): counts
rownames(64102): ENSG00000000003 ENSG00000000005 ... LRG_98 LRG_99
rowData names(0):
colnames(8): SRR1039508 SRR1039509 ... SRR1039520 SRR1039521
colData names(9): SampleName cell ... Sample BioSample
> assay(airway, "counts")[1:3, 1:3]
                SRR1039508 SRR1039509 SRR1039512
```

[2] http://www-huber.embl.de/users/anders/HTSeq/

[3] http://bioconductor.org/packages/airway

```
ENSG00000000003        679        448        873
ENSG00000000005          0          0          0
ENSG00000000419        467        515        621
> airway$dex
[1] untrt trt   untrt trt   untrt trt   untrt trt
Levels: trt untrt
```

The main variable of interest is `dex` which takes on levels `trt` (treated) and `untrt` (untreated). The first level will be the reference level for this factor, so we use `relevel()` to set the `untrt` level as reference; this is much easier to interpret.

```
> airway$dex <- relevel(airway$dex, "untrt")
> airway$dex
[1] untrt trt   untrt trt   untrt trt   untrt trt
Levels: untrt trt
```

There is rich information about which gene model was used for each gene:

```
> granges(airway)
GRangesList object of length 64102:
$ENSG00000000003
GRanges object with 17 ranges and 2 metadata columns:
       seqnames              ranges strand |   exon_id       exon_name
          <Rle>           <IRanges>  <Rle> | <integer>     <character>
   [1]        X [99883667, 99884983]      - |    667145 ENSE00001459322
   [2]        X [99885756, 99885863]      - |    667146 ENSE00000868868
   [3]        X [99887482, 99887565]      - |    667147 ENSE00000401072
   [4]        X [99887538, 99887565]      - |    667148 ENSE00001849132
   [5]        X [99888402, 99888536]      - |    667149 ENSE00003554016
   ...      ...                 ...    ... .       ...             ...
  [13]        X [99890555, 99890743]      - |    667156 ENSE00003512331
  [14]        X [99891188, 99891686]      - |    667158 ENSE00001886883
  [15]        X [99891605, 99891803]      - |    667159 ENSE00001855382
  [16]        X [99891790, 99892101]      - |    667160 ENSE00001863395
  [17]        X [99894942, 99894988]      - |    667161 ENSE00001828996

<64101 more elements>
seqinfo: 722 sequences (1 circular) from an unspecified genome
```

30.6 edgeR

The *edgeR* is very similar in terms of data structures and functionality to the *limma*. Whereas *limma* allows us to operate directy on `ExpressionSets`, edgeR does not work directly with `SummarizedExperiment`. We first need to put out data into an edgeR specific container.

```
> library(edgeR)
> dge <- DGEList(counts = assay(airway, "counts"),
+                group = airway$dex)
> dge$samples <- merge(dge$samples,
+                      as.data.frame(colData(airway)),
+                      by = 0)
> dge$genes <- data.frame(name = names(rowRanges(airway)),
+                         stringsAsFactors = FALSE)
```

This object has something called the `group` which is the basic experimental group for each sample. It also has `$samples` (the pheno data) which - weirdly - cannot be set when you create the `DGEList` object, so we set it afterwards. The `$genes` is a `data.frame` so we cannot include the rich gene model information we had in the `SummarizedExperiment`.

Having set up the input object, we now proceed as follows.

First we estimate the normalization factors or effective library sizes

```
> dge <- calcNormFactors(dge)
```

Next we setup the design matrix and estimate the dispersion (variance). There are multiple ways to do this, and the weird two-step procedure is necessary.

```
> design <- model.matrix(~dge$samples$group)
> dge <- estimateGLMCommonDisp(dge, design)
> dge <- estimateGLMTagwiseDisp(dge, design)
```

Now we do a `glmFit()`, similar to *limma*

```
> fit <- glmFit(dge, design)
```

Now it is time to do a test and extract the top hits

```
> lrt <- glmLRT(fit, coef = 2)
> topTags(lrt)
Coefficient:  dge$samples$grouptrt
                 name     logFC   logCPM       LR       PValue
9658  ENSG00000152583  4.584952 5.536758 286.3965 3.032129e-64
14922 ENSG00000179593 10.100345 1.663884 180.1177 4.568028e-41
3751  ENSG00000109906  7.128577 4.164217 170.6604 5.307950e-39
44236 ENSG00000250978  6.166269 1.405150 168.8572 1.314558e-38
14827 ENSG00000179094  3.167788 5.177666 161.6348 4.971441e-37
17245 ENSG00000189221  3.289112 6.769370 138.9111 4.606056e-32
5054  ENSG00000120129  2.932939 7.310875 137.0461 1.178199e-31
2529  ENSG00000101347  3.842550 9.207551 131.4672 1.956855e-30
2071  ENSG00000096060  3.921841 6.899072 123.3973 1.141438e-28
14737 ENSG00000178695 -2.515219 6.959338 122.9711 1.414932e-28
                FDR
9658  1.943655e-59
14922 1.464099e-36
3751  1.134167e-34
44236 2.106644e-34
14827 6.373586e-33
17245 4.920957e-28
5054  1.078927e-27
2529  1.567979e-26
2071  8.129829e-25
14737 9.069997e-25
```

30.7 DESeq2

Like *edgeR*, DESeq2 requires us to put the data into a package-specific container (a `DESeqDataSet`). But unlike edgeR, it is pretty easy.

```
> library(DESeq2)
> dds <- DESeqDataSet(airway, design = ~ dex)
```

Note that the design of the experiment is stored inside the object. The last variable (in case multiple variables are list) will be the variable of interest which is report in the different results outputs.

Fitting the model is simple

```
> dds <- DESeq(dds)
```

and then all we need to do is get the results. Note that the results are not ordered, so we do that.

```
> res <- results(dds)
> res <- res[order(res$padj),]
> res[1:10,]
log2 fold change (MAP): dex trt vs untrt
Wald test p-value: dex trt vs untrt
DataFrame with 10 rows and 6 columns
                  baseMean log2FoldChange      lfcSE      stat
                 <numeric>      <numeric>  <numeric> <numeric>
ENSG00000152583   997.4398       4.280694 0.19572061  21.87145
ENSG00000148175 11193.7188       1.434429 0.08325248  17.22987
ENSG00000179094   776.5967       2.981009 0.18833478  15.82825
ENSG00000109906   385.0710       5.095376 0.32987788  15.44625
ENSG00000134686  2737.9820       1.368175 0.08974798  15.24463
ENSG00000125148  3656.2528       2.126258 0.14207457  14.96579
ENSG00000120129  3409.0294       2.760597 0.18885833  14.61729
ENSG00000189221  2341.7673       3.039185 0.20995474  14.47543
ENSG00000178695  2649.8501      -2.372770 0.16979309 -13.97448
ENSG00000101347 12703.3871       3.406507 0.24761309  13.75738
                      pvalue          padj
                   <numeric>     <numeric>
ENSG00000152583 4.858346e-106 9.020004e-102
ENSG00000148175  1.585139e-66  1.471484e-62
ENSG00000179094  1.986835e-56  1.229586e-52
ENSG00000109906  7.996137e-54  3.711407e-50
ENSG00000134686  1.787492e-52  6.637314e-49
ENSG00000125148  1.228541e-50  3.801516e-47
ENSG00000120129  2.178980e-48  5.779276e-45
ENSG00000189221  1.732453e-47  4.020589e-44
ENSG00000178695  2.231464e-44  4.603262e-41
ENSG00000101347  4.599152e-43  8.538786e-40
```

and then we print the first 10 hits.

30.8 Comments

We see that amongst the top 5 genes, 3 are shared between edgeR and DESeq2, with some small variation in the estimated fold-change. The two methods are both being continually developed (and probably bench-marked against each other by the authors). At any given time it is difficult to decide which one to prefer.

30.9 Other Resources

- The vignette from the edgeR webpage[4].
- The vignette from the DESeq2 webpage[5].
- The RNA-seq workflow[6].

[4] http://bioconductor.org/packages/edgeR
[5] http://bioconductor.org/packages/DESeq2
[6] http://bioconductor.org/help/workflows/rnaseqGene/

Details on R and Bioconductor

The version of R used and all packages used to produce the output of this book, are detailed in the following call to `sessionInfo()`:

```
## R version 3.3.0 (2016-05-03)
## Platform: x86_64-apple-darwin13.4.0 (64-bit)
## Running under: OS X 10.11.5 (El Capitan)
##
## locale:
## [1] en_US.UTF-8/en_US.UTF-8/en_US.UTF-8/C/en_US.UTF-8/en_US.UTF-8
##
## attached base packages:
## [1] stats4    parallel  methods   stats     graphics  grDevices utils
## [8] datasets  base
##
## other attached packages:
##  [1] oligo_1.36.1
##  [2] oligoClasses_1.34.0
##  [3] leukemiasEset_1.8.0
##  [4] hgu95av2.db_3.2.2
##  [5] org.Hs.eg.db_3.3.0
##  [6] edgeR_3.14.0
##  [7] limma_3.28.4
##  [8] biomaRt_2.28.0
##  [9] airway_0.106.0
## [10] TxDb.Hsapiens.UCSC.hg19.knownGene_3.2.2
## [11] ShortRead_1.30.0
## [12] GenomicAlignments_1.8.0
## [13] BiocParallel_1.6.2
## [14] Rsamtools_1.24.0
## [15] IlluminaHumanMethylation450kmanifest_0.4.0
## [16] IlluminaHumanMethylation450kanno.ilmn12.hg19_0.2.1
## [17] minfi_1.18.2
## [18] bumphunter_1.12.0
## [19] locfit_1.5-9.1
## [20] iterators_1.0.8
## [21] foreach_1.4.3
## [22] lattice_0.20-33
```

```
## [23] GenomicFeatures_1.24.2
## [24] AnnotationDbi_1.34.2
## [25] GEOquery_2.38.4
## [26] DESeq2_1.12.2
## [27] SummarizedExperiment_1.2.2
## [28] BSgenome.Scerevisiae.UCSC.sacCer2_1.4.0
## [29] BSgenome_1.40.0
## [30] rtracklayer_1.32.0
## [31] Biostrings_2.40.0
## [32] XVector_0.12.0
## [33] GenomicRanges_1.24.0
## [34] GenomeInfoDb_1.8.1
## [35] IRanges_2.6.0
## [36] S4Vectors_0.10.0
## [37] AnnotationHub_2.4.2
## [38] ALL_1.14.0
## [39] Biobase_2.32.0
## [40] BiocGenerics_0.18.0
## [41] BiocStyle_2.0.2
## [42] knitr_1.13
##
## loaded via a namespace (and not attached):
##  [1] colorspace_1.2-6             hwriter_1.3.2
##  [3] siggenes_1.46.0              mclust_5.2
##  [5] base64_2.0                   affyio_1.42.0
##  [7] interactiveDisplayBase_1.10.3 codetools_0.2-14
##  [9] splines_3.3.0                geneplotter_1.50.0
## [11] Formula_1.2-1                annotate_1.50.0
## [13] cluster_2.0.4                shiny_0.13.2
## [15] httr_1.1.0                   Matrix_1.2-6
## [17] formatR_1.4                  acepack_1.3-3.3
## [19] htmltools_0.3.5              tools_3.3.0
## [21] gtable_0.2.0                 affxparser_1.44.0
## [23] doRNG_1.6                    Rcpp_0.12.5
## [25] multtest_2.28.0              preprocessCore_1.34.0
## [27] nlme_3.1-127                 stringr_1.0.0
## [29] mime_0.4                     rngtools_1.2.4
## [31] XML_3.98-1.4                 beanplot_1.2
## [33] zlibbioc_1.18.0              MASS_7.3-45
## [35] scales_0.4.0                 BiocInstaller_1.22.2
## [37] RColorBrewer_1.1-2           gridExtra_2.2.1
## [39] ggplot2_2.1.0                pkgmaker_0.22
```

```
## [41] rpart_4.1-10                reshape_0.8.5
## [43] latticeExtra_0.6-28         stringi_1.0-1
## [45] RSQLite_1.0.0               genefilter_1.54.2
## [47] chron_2.3-47                matrixStats_0.50.2
## [49] bitops_1.0-6                nor1mix_1.2-1
## [51] evaluate_0.9                bit_1.1-12
## [53] plyr_1.8.3                  magrittr_1.5
## [55] R6_2.1.2                    Hmisc_3.17-4
## [57] DBI_0.4-1                   foreign_0.8-66
## [59] survival_2.39-2             RCurl_1.95-4.8
## [61] nnet_7.3-12                 grid_3.3.0
## [63] data.table_1.9.6            digest_0.6.9
## [65] xtable_1.8-2                ff_2.2-13
## [67] httpuv_1.3.3                illuminaio_0.14.0
## [69] openssl_0.9.3               munsell_0.4.3
## [71] registry_0.3                quadprog_1.5-5
```

About the Author

Kasper D. Hansen is an Assistant Professor in the Department of Biostatistics[7] and the Institute of Genetic Medicine[8] at Johns Hopkins University. He is a co-director of the Johns Hopkins Specialization in Genomic Data Science[9] and a member of the technical advisory board for the Bioconductor[10] project. He can be found on Twitter @KasperDHansen[11]. His scientific work can be found though his lab page[12].

[7] http://www.biostat.jhsph.edu
[8] https://igm.jhmi.edu/
[9] https://www.coursera.org/specializations/genomic-data-science
[10] http://www.bioconductor.org
[11] https://twitter.com/kasperdhansen
[12] http://www.hansenlab.org