

The Bioconductor 2018 Workshop Compilation

Contents

1	Introduction	5
1.1	The Workshops	5
2	100: R and Bioconductor for everyone: an introduction	7
2.1	Overview	7
2.2	Introduction to R	9
2.3	Introduction to Bioconductor	18
2.4	Summary	30
3	101: Introduction to Bioconductor annotation resources	31
3.1	Instructors	31
3.2	Workshop Description	31
3.3	Workshop goals and objectives	32
3.4	Annotation Workshop	32
4	102: Solving common bioinformatic challenges using GenomicRanges	53
4.1	Instructor name and contact information	53
4.2	Workshop Description	53
4.3	Workshop goals and objectives	54
4.4	Introduction	54
4.5	Setup	55
4.6	<i>GRanges</i> : Genomic Ranges	55
4.7	Constructing a <i>GRanges</i> object from data.frame	55
4.8	Loading a <i>GRanges</i> object from a standard file format	56
4.9	Basic manipulation of <i>GRanges</i> objects	56
4.10	Subsetting <i>GRanges</i> objects	58
4.11	Splitting and combining <i>GRanges</i> objects	60
4.12	Aggregating <i>GRanges</i> objects	62
4.13	Basic interval operations for <i>GRanges</i> objects	62
4.14	Interval set operations for <i>GRanges</i> objects	67
4.15	Finding overlaps between <i>GRanges</i> objects	68
4.16	Exercises	70
4.17	Example: exploring BigWig files from AnnotationHub	71
4.18	Worked example: coverage analysis of BAM files	74
4.19	Conclusions	77
5	103: Public data resources and Bioconductor	79
5.1	Instructor names and contact information	79
5.2	Syllabus	79
5.3	Overview	81
5.4	GEOquery	81
5.5	GenomicDataCommons	84

5.6	Querying metadata	89
5.7	Sequence Read Archive	97
5.8	Accessing The Cancer Genome Atlas (TCGA)	104
5.9	recount: Reproducible RNA-seq Analysis Using recount2	106
5.10	curated*Data packages for standardized cancer transcriptomes	107
5.11	Microbiome data	107
5.12	Pharmacogenomics	109
5.13	Bibliography	112
6	200: RNA-seq analysis is easy as 1-2-3 with limma, Glimma and edgeR	113
6.1	Instructor name and contact information	113
6.2	Workshop Description	113
6.3	Workshop goals and objectives	114
6.4	Introduction	115
6.5	Data packaging	115
6.6	Data pre-processing	119
6.7	Differential expression analysis	124
7	201: RNA-seq data analysis with DESeq2	133
7.1	Overview	133
7.2	Preparing data for <i>DESeq2</i>	134
7.3	Importing into R with <i>tximport</i>	136
7.4	Exploratory data analysis	140
7.5	Differential expression analysis	144
7.6	<i>AnnotationHub</i>	148
7.7	Building reports	151
7.8	Integration with <i>ZINB-WaVE</i>	152
8	202: Analysis of single-cell RNA-seq data: Dimensionality reduction, clustering, and lineage inference	159
8.1	Overview	159
8.2	Getting started	160
8.3	The <i>SingleCellExperiment</i> class	162
8.4	Pre-processing	165
8.5	Normalization and dimensionality reduction: ZINB-WaVE	167
8.6	Cell clustering: RSEC	168
8.7	Cell lineage and pseudotime inference: Slingshot	174
8.8	Differential expression analysis along lineages	179
9	210: Functional enrichment analysis of high-throughput omics data	181
9.1	Instructor names and contact information	181
9.2	Workshop Description	181
9.3	Goals and objectives	182
9.4	Where does it all come from?	182
9.5	Gene expression-based enrichment analysis	183
9.6	A primer on terminology, existing methods & statistical theory	183
9.7	Data types	185
9.8	Differential expression analysis	187
9.9	Gene sets	189
9.10	GO/KEGG overrepresentation analysis	190
9.11	Functional class scoring & permutation testing	192
9.12	Network-based enrichment analysis	194
9.13	Genomic region enrichment analysis	199
10	220: Workflow for multi-omics analysis with MultiAssayExperiment	203

10.1 Instructor names and contact information	203
10.2 Workshop Description	203
10.3 Workshop goals and objectives	204
10.4 Overview of key data classes	205
10.5 Working with RaggedExperiment	208
10.6 Working with MultiAssayExperiment	213
10.7 API cheat sheet	213
10.8 MultiAssayExperiment Subsetting	217
10.9 Complete cases	218
10.10 Row names that are common across assays	219
10.11 Extraction	219
10.12 Summary of slots and accessors	219
10.13 Transformation / reshaping	220
10.14 MultiAssayExperiment class construction and concatenation	222
10.15 The Cancer Genome Atlas (TCGA) as MultiAssayExperiment objects	223
10.16 Utilities for TCGA	224
10.17 Plotting, correlation, and other analyses	226
11 230: Cytoscape automation in R using RCy3	237
11.1 Overview	237
11.2 Background	238
11.3 Translating biological data into Cytoscape using RCy3	242
11.4 Set up	242
11.5 Cytoscape Basics	246
11.6 Example Data Set	248
11.7 Finding Network Data	248
11.8 Use Case 1 - How are my top genes related?	248
11.9 Use Case 2 - Which genes have similar expression.	258
11.10 Use Case 3 - Functional Enrichment of Omics set.	266
12 240: Fluent genomic data analysis with plyranges	277
12.1 Instructor names and contact information	277
12.2 Workshop Description	277
12.3 Workshop goals and objectives	278
12.4 Workshop	278
12.5 Introduction	278
12.6 Setup	279
12.7 What are GRanges objects?	279
12.8 The Grammar	280
12.9 Data import and creating pipelines	292
12.10 What's next?	299
12.11 Appendix	299
13 250: Working with genomic data in R with the DECIIPHER package	301
13.1 Overview	301
13.2 Workshop goals and objectives	302
14 260: Biomarker discovery from large pharmacogenomics datasets	321
14.1 Instructors:	321
14.2 Workshop Description	321
14.3 Workshop goals and objectives	322
14.4 Abstract	322
14.5 Introduction	323
14.6 Reproducibility	324
14.7 Replication	326

14.8 Machine Learning and Biomarker Discovery	332
14.9 Session Info	337
15 500: Effectively using the DelayedArray framework to support the analysis of large datasets	339
15.1 Overview	339
15.2 Introductory material	341
15.3 Overview of DelayedArray framework	344
15.4 What's out there already?	368
15.5 Incorporating DelayedArray into a package	368
15.6 Questions and discussion	368
15.7 TODOs	368
16 510: Maintaining your Bioconductor package	371
16.1 Instructor name and contact information	371
16.2 Workshop Description	371
16.3 Introduction	372
16.4 Outline how to use infrastructure	373
16.5 Round up of resources available	382
16.6 Acknowledgements	382
17 Bibliography	383

Chapter 1

Introduction

Author: Martin Morgan¹. Last modified: 19 July, 2018.

Welcome to Bioc2018. This year's conference includes a wide array of workshops for audiences ranging from beginner to advance users and developers. Workshop materials are available as a book in html, pdf, and eBook format at <https://bioconductor.github.io/BiocWorkshops/>. Workshops are organized by level and topic according to numbers, as described below. Every workshop starts with a syllabus that will help you to decide whether it matches your learning goals.

1.1 The Workshops

This book contains workshops for *R* / *Bioconductor* training. The workshops are divided into 3 sections:

- **Learn** (100-series chapters) contains material for beginning users of *R* and *Bioconductor*. However, even experienced *R* and *Bioconductor* users may find something new here.
 - 100: *R* and *Bioconductor* for everyone: an introduction
 - 101: Introduction to Bioconductor annotation resources
 - 102: Solving common bioinformatic challenges using GenomicRanges
 - 103: Public data resources and Bioconductor
- **Use** (200-series chapters) contains workshops emphasizing use of *Bioconductor* for common tasks, e.g., RNA-seq differential expression, single-cell analysis, gene set enrichment, multi'omics analysis, genome analysis, network analysis, and pharmacogenomics.
 - 200: RNA-seq analysis is easy as 1-2-3 with limma, Glimma and edgeR
 - 201: RNA-seq data analysis with DESeq2
 - 202: Analysis of single-cell RNA-seq data: Dimensionality reduction, clustering, and lineage inference
 - 210: Functional enrichment analysis of high-throughput omics data
 - 220: Workflow for multi-omics analysis with MultiAssayExperiment
 - 230: Cytoscape automation in R using Rcy3
 - 240: Fluent genomic data analysis with plyranges
 - 250: Working with genomic data in R with the DECIPHER package
 - 260: Biomarker discovery from large pharmacogenomics datasets
- **Develop** (500-series chapters) contains workshops to help expert users hone their skills and contribute their domain-specific knowledge to the *Bioconductor* community. These workshops are presented on “Developer Day”.
 - 500: Effectively using the DelayedArray framework to support the analysis of large datasets
 - 510: Maintaining your Bioconductor package

¹Roswell Park Comprehensive Cancer Center, Buffalo, NY

Chapter 2

100: *R* and *Bioconductor* for everyone: an introduction

Authors: Martin Morgan¹, Lori Shepherd. Last modified: 17 July 2018

2.1 Overview

2.1.1 Description

This workshop is intended for those with little or no experience using *R* or *Bioconductor*. In the first portion of the workshop, we will explore the basics of using *RStudio*, essential *R* data types, composing short scripts and using functions, and installing and using packages that extend base *R* functionality. The second portion of the workshop orients participants to the *Bioconductor* collection of *R* packages for analysis and comprehension of high-throughput genomic data. We will describe how to discover, install, and learn to use *Bioconductor* packages, and will explore some of the unique ways in which *Bioconductor* represents genomic data. The workshop will primarily be instructor-led live demos, with participants following along in their own *RStudio* sessions.

2.1.2 Pre-requisites

This workshop is meant to be introductory, and has no pre-requisites.

Participants will maximize benefit from the workshop by pursuing elementary instructions for using *R*, such as the introductory course from DataCamp.

2.1.3 Participation

Participants will use *RStudio* interactively as the instructor moves carefully through short examples of *R* and *Bioconductor* code.

2.1.4 *R* / *Bioconductor* packages used

- Base *R* packages, e.g., `stats`, `graphics`; `ggplot2`

¹Roswell Park Comprehensive Cancer Center, Buffalo, NY

- Essential *Bioconductor* packages, e.g., rtracklayer, GenomicRanges, SummarizedExperiment

2.1.5 Time outline

An example for a 45-minute workshop:

Activity	Time
Introduction to <i>R</i>	45m
- Using <i>RStudio</i>	
- <i>R</i> vectors and data frames	
- Data input and manipulation	
- Scripts and functions	
- Using <i>R</i> packages	
Introduction to <i>Bioconductor</i>	60m
- Project history	
- Discovering and using packages	
- Working with objects	

2.1.6 Workshop goals and objectives

2.1.7 Learning goals

Part 1: *R*

- Import text (e.g., ‘comma-separate value’) files, into *R*.
- Perform manipulations of *R* data frames.
- Apply *R* functions for statistical analysis and visualization.
- Use *R* packages to extend basic functionality.

Part 2: *Bioconductor*

- Find, install, and learn how to use *Bioconductor* packages.
- Import and manipulate genomic files and *Bioconductor* data objects.
- Start an RNA-seq differential expression work flow.

2.1.8 Learning objectives

Part 1: *R*

- Import the ‘pData.csv’ file describing samples from an experiment.
- ‘Clean’ the pData to include only a subset of values.
- Perform a t-test assessing the relationship between ‘age’ and ‘mol.biol’ (presence of BCR/ABL) amongst samples.
- Visualize the relationship between ‘age’ and ‘mol.biol’ using base *R*’s `boxplot()` function.
- Load the `ggplot2` package.
- Visualize the relationship between two variables using `ggplot()`.

Part 2: *Bioconductor*

- Discover, install, and read the vignette of the DESeq2 package.
- Discover the ‘single cell sequencing’ vignette
- Import BED and GTF files into *Bioconductor*
- Find regions of overlap between the BED and GTF files.

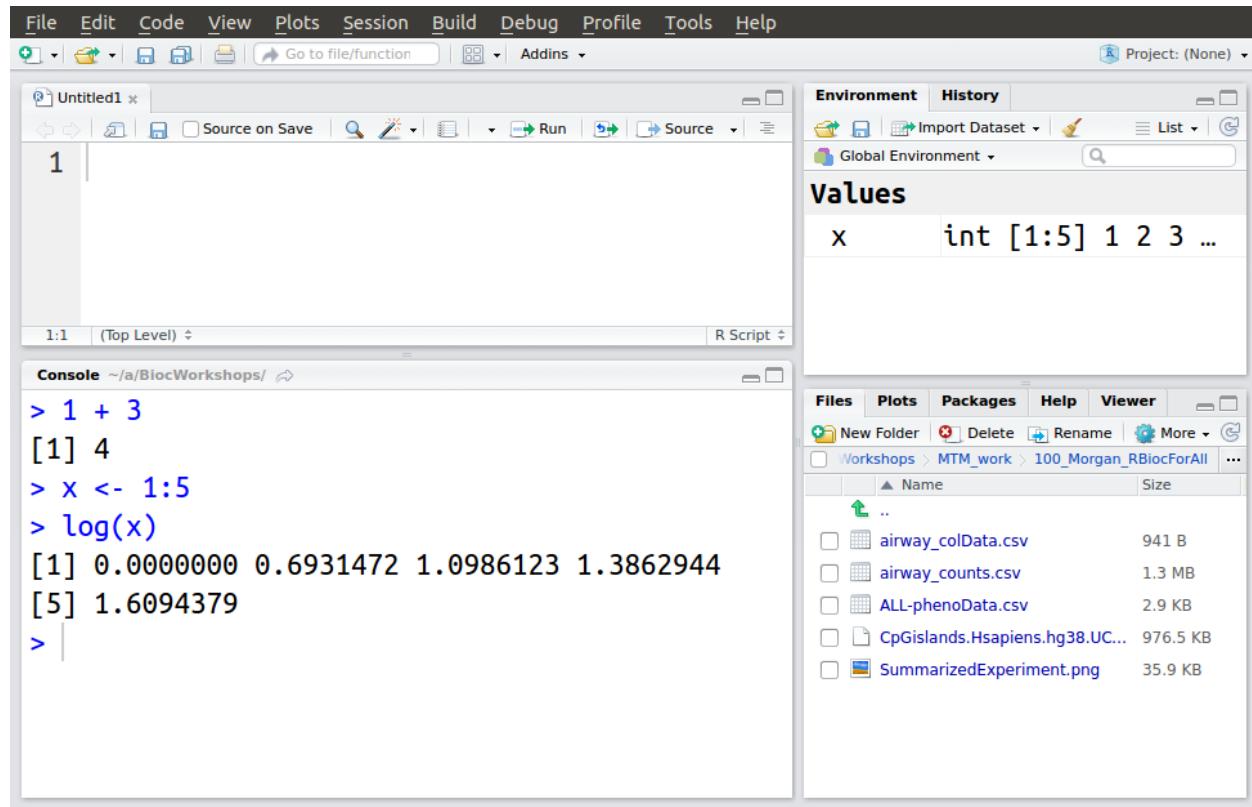


Figure 2.1:

- Import a matrix and data.frame into *Bioconductor*'s `SummarizedExperiment` object for RNA-Seq differential expression.

2.2 Introduction to *R*

2.2.1 *RStudio* orientation

2.2.2 Very basics

R works by typing into the console. A very simple example is to add the numbers 1 and 2.

```
1 + 2
#> [1] 3
```

R works best on *vectors*, and we can construct vectors ‘by hand’ using the function `c()` (perhaps for `_c_`oncatenate). Here is a vector of the numbers 1, 3, and 5.

```
c(1, 3, 5)
#> [1] 1 3 5
```

R has many shortcuts. A very commonly used shortcut is to create a vector representing a sequence of numbers using `:`. For instance, here is a vector representing the numbers 1 through 5.

```
1:5
#> [1] 1 2 3 4 5
```

It would be pretty tedious to continually have to enter vectors by hand. *R* allows objects such as the vector returned by `c()` to be assigned to variables. The variables can then be referenced at subsequent points in the script.

```
x <- c(1, 3, 5)
x
#> [1] 1 3 5
```

Since *R* knows about vectors, it can be very easy to transform all elements of the vector, e.g., taking the `log()` of `x`:

```
log(x)
#> [1] 0.000000 1.098612 1.609438
```

The return value of `log(x)` is itself a vector, and can be assigned to another variable.

```
y <- log(x)
y
#> [1] 0.000000 1.098612 1.609438
```

2.2.3 Data input and manipulation

Read data files into *R* `data.frame` objects. We start by reading a simple file into *R*. The file is a ‘csv’ (comma-separated value) file that could have been exported from a spreadsheet. The first few lines of the file include:

```
"Sample", "sex", "age", "mol.biol"
"01005", "M", 53, "BCR/ABL"
"01010", "M", 19, "NEG"
"03002", "F", 52, "BCR/ABL"
"04006", "M", 38, "ALL1/AF4"
"04007", "M", 57, "NEG"
```

The file describes samples used in a classic microarray experiment. It consists of four columns:

- `Sample`: a unique identifier
- `sex`: the sex of each patient
- `age`: the age of each patient
- `mol.biol`: cytological characterization of each patient, e.g., “BCR/ABL” indicates presence of the classic BCR/ABL translocation, while “NEG” indicates no special cytological information.

We start by asking are to find the *path* to the file, using a function `file.choose()`. This will open a dialog box, and we will navigate to the location of a file named “`ALL-phenoData.csv`”. Print the value of `fname` to view the path that you choose.

```
fname <- file.choose()
fname
```

Confirm that the file exists using the `file.exists()` command.

```
file.exists(fname)
#> [1] TRUE
```

Read the data from the csv file using `read.csv()`; assign the input to a variable called `pdata`.

```
pdata <- read.csv(fname)
```

Viewing the `data.frame`

We could print the entire data frame to the screen using

```
pdata
```

but a smarter way to explore the data is to ask about its dimensions (rows and columns) using `dim()`, to look at the `head()` and `tail()` of the data, and to ask *R* for a brief `summary()`.

```
dim(pdata)    # 128 rows x 4 columns
#> [1] 128   4
head(pdata)  # first six rows
#>   Sample sex age mol.biol
#> 1 01005 M 53 BCR/ABL
#> 2 01010 M 19 NEG
#> 3 03002 F 52 BCR/ABL
#> 4 04006 M 38 ALL1/AF4
#> 5 04007 M 57 NEG
#> 6 04008 M 17 NEG
tail(pdata) # last six rows
#>   Sample sex age mol.biol
#> 123 49004 M 24 NEG
#> 124 56007 M 37 NEG
#> 125 64005 M 19 NEG
#> 126 65003 M 30 NEG
#> 127 83001 M 29 NEG
#> 128 LAL4 <NA> NA NEG
summary(pdata)
#>   Sample      sex       age      mol.biol
#> 01003 : 1   F :42   Min.   : 5.00  ALL1/AF4:10
#> 01005 : 1   M :83   1st Qu.:19.00  BCR/ABL :37
#> 01007 : 1   NA's: 3 Median :29.00  E2A/PBX1: 5
#> 01010 : 1           Mean  :32.37  NEG     :74
#> 02020 : 1           3rd Qu.:45.50  NUP-98  : 1
#> 03002 : 1           Max.   :58.00  p15/p16 : 1
#> (Other):122        NA's    :5
```

The `age` element of `pdata` is a vector of integers and the summary provides a quantitative description of the data. Some values are missing, and these have the special value `NA`.

The `sex` and `mol.biol` vectors are *factors*. The `sex` variable has two *levels* (`M`, `F`) while the `mol.biol` vector has 6 levels. A factor is a statistical concept central to describing data; the levels describe the universe of possible values that the variable can take.

`Sample` is also a `factor`, but it should probably be considered a `character` vector used to identify each sample; we clean this up in just a minute.

In addition to `summary()`, it can be helpful to use `class()` to look at the class of each object or column, e.g.,

```
class(fname)
#> [1] "character"
class(pdata)
#> [1] "data.frame"
```

Reading data, round two

We noted that `Sample` has been read by *R* as a factor. Consult the (complicated!) help page for `read.csv()` and try to understand how to use `colClasses` to specify how each column of data should be input.

To navigate to the help page, use

```
?read.csv
```

Focus on the `colClasses` argument. It is a vector that describes the class each column should be interpreted as. We'd like to read the columns as character, factor, integer, factor. Start by creating a vector of desired values

```
c("character", "factor", "integer", "factor")
#> [1] "character" "factor"    "integer"   "factor"
```

Then add another argument to our use of `read.csv()`, specifying the desired column classes as this vector.

```
pdata <- read.csv(
  fname,
  colClasses = c("character", "factor", "integer", "factor")
)
summary(pdata)
#>      Sample           sex         age        mol.biol
#>  Length:128      F :42   Min.   : 5.00  ALL1/AF4:10
#>  Class :character M :83   1st Qu.:19.00  BCR/ABL :37
#>  Mode  :character NA's: 3   Median :29.00  E2A/PBX1: 5
#>                               Mean   :32.37  NEG     :74
#>                               3rd Qu.:45.50  NUP-98  : 1
#>                               Max.   :58.00  p15/p16 : 1
#>                               NA's    :5
```

Subsetting

A basic operation in *R* is to subset data. A `data.frame` is a two-dimensional object, so it is subset by specifying the desired rows and columns. Several different ways of specifying rows and columns are possible. Row selection often involves numeric vectors, logical vectors, and character vectors (selecting the rows with the corresponding `rownames()`). Column selection is most often with a character vector, but can also be numeric.

```
pdata[1:5, c("sex", "mol.biol")]
#>   sex mol.biol
#> 1  M  BCR/ABL
#> 2  M    NEG
#> 3  F  BCR/ABL
#> 4  M ALL1/AF4
#> 5  M    NEG
pdata[1:5, c(2, 3)]
#>   sex age
#> 1  M  53
#> 2  M  19
#> 3  F  52
#> 4  M  38
#> 5  M  57
```

Omitting either the row or column index selects all rows or columns

```
pdata[1:5, ]
#>   Sample sex age mol.biol
#> 1 01005  M  53 BCR/ABL
#> 2 01010  M  19    NEG
#> 3 03002  F  52 BCR/ABL
#> 4 04006  M  38 ALL1/AF4
#> 5 04007  M  57    NEG
```

The subset operator [generally returns the same type of object as being subset, e.g., above all return values are `data.frame` objects. It is sometimes desirable to select a single column as a vector. This can be done using \$ or [[; note that some values are NA, indicating that the patient's age is "not available".

```
pdata$age
#> [1] 53 19 52 38 57 17 18 16 15 40 33 55 5 18 41 27 27 46 37 36 53 39 53
#> [24] 20 44 28 58 43 48 58 19 26 19 32 17 45 20 16 51 57 29 16 32 15 NA 21
#> [47] 49 38 17 26 48 16 18 17 22 47 21 54 26 19 47 18 52 27 52 18 18 23 16
#> [70] NA 54 25 31 19 24 23 NA 41 37 54 18 19 43 53 50 54 53 49 20 26 22 36
#> [93] 27 50 NA 31 16 48 17 40 22 30 18 22 50 41 40 28 25 16 31 14 24 19 37
#> [116] 23 30 48 22 41 52 32 24 37 19 30 29 NA
pdata[["age"]]
#> [1] 53 19 52 38 57 17 18 16 15 40 33 55 5 18 41 27 27 46 37 36 53 39 53
#> [24] 20 44 28 58 43 48 58 19 26 19 32 17 45 20 16 51 57 29 16 32 15 NA 21
#> [47] 49 38 17 26 48 16 18 17 22 47 21 54 26 19 47 18 52 27 52 18 18 23 16
#> [70] NA 54 25 31 19 24 23 NA 41 37 54 18 19 43 53 50 54 53 49 20 26 22 36
#> [93] 27 50 NA 31 16 48 17 40 22 30 18 22 50 41 40 28 25 16 31 14 24 19 37
#> [116] 23 30 48 22 41 52 32 24 37 19 30 29 NA
```

We can use `class()` to determine the type of each column

```
class(pdata$age)
#> [1] "integer"
```

as well as other functions to manipulate or summarize the data. What do each of the following lines do?

```
table(pdata$mol.biol)
#>
#> ALL1/AF4 BCR/ABL E2A/PBX1      NEG    NUP-98 p15/p16
#>      10      37       5      74        1        1
table(is.na(pdata$age))
#>
#> FALSE  TRUE
#> 123     5
levels(pdata$sex)
#> [1] "F" "M"
```

Basic data manipulations

We've seen (e.g., `log()`, above) that numeric vectors can be transformed by mathematical functions. Similar functions are available for other data types. A very common scenario is to create logical vectors that are then used to subset objects. Here we compare each element of the `sex` column to "F". The result indicates which rows in the `pdata` correspond to individuals with sex "F".

```
pdata$sex == "F"
#> [1] FALSE FALSE TRUE FALSE FALSE FALSE TRUE FALSE FALSE FALSE
#> [12] FALSE FALSE FALSE FALSE TRUE FALSE FALSE FALSE TRUE FALSE FALSE
#> [23] FALSE FALSE FALSE FALSE TRUE TRUE TRUE TRUE TRUE FALSE
#> [34] FALSE TRUE TRUE TRUE TRUE TRUE FALSE TRUE FALSE TRUE FALSE
#> [45] NA FALSE TRUE FALSE TRUE TRUE TRUE FALSE FALSE FALSE FALSE
#> [56] TRUE FALSE TRUE FALSE TRUE FALSE FALSE TRUE FALSE FALSE FALSE
#> [67] FALSE FALSE FALSE FALSE TRUE FALSE FALSE TRUE TRUE FALSE FALSE
#> [78] FALSE TRUE FALSE FALSE FALSE TRUE TRUE TRUE FALSE FALSE FALSE
#> [89] FALSE TRUE FALSE FALSE TRUE FALSE NA FALSE TRUE TRUE FALSE
#> [100] TRUE FALSE FALSE FALSE FALSE FALSE TRUE TRUE FALSE TRUE
#> [111] FALSE FALSE FALSE TRUE FALSE FALSE TRUE FALSE FALSE FALSE FALSE
#> [122] FALSE FALSE FALSE FALSE FALSE FALSE NA
```

A more elaborate example identifies rows corresponding to females greater than 50 years of age. Note that some comparisons are NA; why is that?

```
(pdata$sex == "F") & (pdata$age > 50)
#> [1] FALSE FALSE TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
#> [12] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
#> [23] FALSE FALSE FALSE FALSE TRUE FALSE FALSE TRUE FALSE FALSE FALSE
#> [34] FALSE FALSE FALSE FALSE FALSE TRUE FALSE FALSE FALSE FALSE FALSE
#> [45] NA FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
#> [56] FALSE FALSE TRUE FALSE FALSE FALSE TRUE FALSE FALSE FALSE FALSE
#> [67] FALSE FALSE FALSE FALSE TRUE FALSE FALSE FALSE FALSE FALSE FALSE
#> [78] FALSE FALSE FALSE FALSE FALSE FALSE TRUE FALSE FALSE FALSE FALSE
#> [89] FALSE FALSE FALSE FALSE FALSE NA FALSE FALSE FALSE FALSE
#> [100] FALSE FALSE
#> [111] FALSE FALSE
#> [122] FALSE FALSE FALSE FALSE FALSE NA
```

Another elaborate comparison is `%in%`, which asks whether each element in the vector on the left-hand side of `%in%` is contained in the set of values on the right-hand side, for example, which elements of the `pdata$mol.biol` column are in the set "BCR/ABL" or "NEG"?

```
table( pdata$mol.biol )
#>
#> ALL1/AF4 BCR/ABL E2A/PBX1      NEG      NUP-98 p15/p16
#>      10      37       5     74       1       1
pdata$mol.biol %in% c("BCR/ABL", "NEG")
#> [1] TRUE TRUE TRUE FALSE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
#> [12] TRUE FALSE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
#> [23] TRUE TRUE FALSE TRUE FALSE TRUE FALSE TRUE TRUE TRUE TRUE
#> [34] TRUE TRUE FALSE TRUE TRUE TRUE TRUE TRUE FALSE TRUE TRUE
#> [45] TRUE TRUE TRUE FALSE FALSE TRUE TRUE TRUE TRUE FALSE TRUE TRUE
#> [56] TRUE TRUE TRUE TRUE TRUE FALSE TRUE FALSE TRUE TRUE TRUE
#> [67] TRUE TRUE TRUE TRUE TRUE FALSE TRUE TRUE FALSE TRUE TRUE
#> [78] TRUE FALSE
#> [89] TRUE TRUE TRUE TRUE TRUE TRUE FALSE TRUE FALSE TRUE TRUE
#> [100] TRUE TRUE
#> [111] TRUE TRUE
#> [122] TRUE TRUE TRUE TRUE TRUE TRUE TRUE
```

The `subset()` function can be used to perform the subset, e.g., selecting all records of females older than 50

```
subset(pdata, sex == "F" & age > 50)
#>   Sample sex age mol.biol
#> 3 03002 F 52 BCR/ABL
#> 27 16004 F 58 ALL1/AF4
#> 30 20002 F 58 BCR/ABL
#> 39 24011 F 51 BCR/ABL
#> 58 28021 F 54 BCR/ABL
#> 63 28032 F 52 ALL1/AF4
#> 71 30001 F 54 BCR/ABL
#> 84 57001 F 53 NEG
```

The return value of `subset()` can be assigned to a variable, so the below creates a subset of `pdata` corresponding to individuals whose `mol.biol` is either "BCR/ABL" or "NEG"; the result is assigned to a variable `bcrabl`.

```
bcrabl <- subset(pdata, mol.biol %in% c("BCR/ABL", "NEG"))
dim( bcrabl )
#> [1] 111   4
```

Here's a tabular summary of the levels of the `mol.biol` factor in `bcrabl`

```
table(bcrabl$mol.biol)
#>
#> ALL1/AF4  BCR/ABL E2A/PBX1      NEG    NUP-98  p15/p16
#>          0       37       0       74       0       0
```

Note that the factor includes levels that are not actually present in the subset. For our work below, we'd like to remove the unused levels. This can be done by calling the `factor()` function

```
factor(bcrabl$mol.biol)
#> [1] BCR/ABL NEG      BCR/ABL NEG      NEG      NEG      NEG      NEG
#> [9] BCR/ABL BCR/ABL NEG      NEG      BCR/ABL NEG      BCR/ABL BCR/ABL
#> [17] BCR/ABL BCR/ABL NEG      BCR/ABL BCR/ABL NEG      BCR/ABL NEG
#> [25] BCR/ABL NEG      BCR/ABL NEG      BCR/ABL BCR/ABL NEG      BCR/ABL
#> [33] BCR/ABL BCR/ABL NEG      BCR/ABL NEG      NEG      NEG      BCR/ABL
#> [41] BCR/ABL BCR/ABL NEG      NEG      NEG      NEG      BCR/ABL BCR/ABL
#> [49] NEG      NEG      NEG      NEG      BCR/ABL NEG      NEG      NEG
#> [57] NEG      NEG      BCR/ABL BCR/ABL NEG      NEG      BCR/ABL BCR/ABL
#> [65] NEG      NEG      NEG      NEG      BCR/ABL NEG      BCR/ABL BCR/ABL
#> [73] BCR/ABL NEG      NEG      BCR/ABL NEG      BCR/ABL BCR/ABL NEG
#> [81] NEG      NEG      NEG      NEG      NEG      NEG      NEG      NEG
#> [89] NEG      NEG      NEG      NEG      NEG      NEG      NEG      NEG
#> [97] NEG      NEG      NEG      NEG      NEG      NEG      NEG      NEG
#> [105] NEG     NEG      NEG      NEG      NEG      NEG      NEG      NEG
#> Levels: BCR/ABL NEG
```

We can then update the `mol.biol` column of `bcrabl` by assigning new values to it with `<-`.

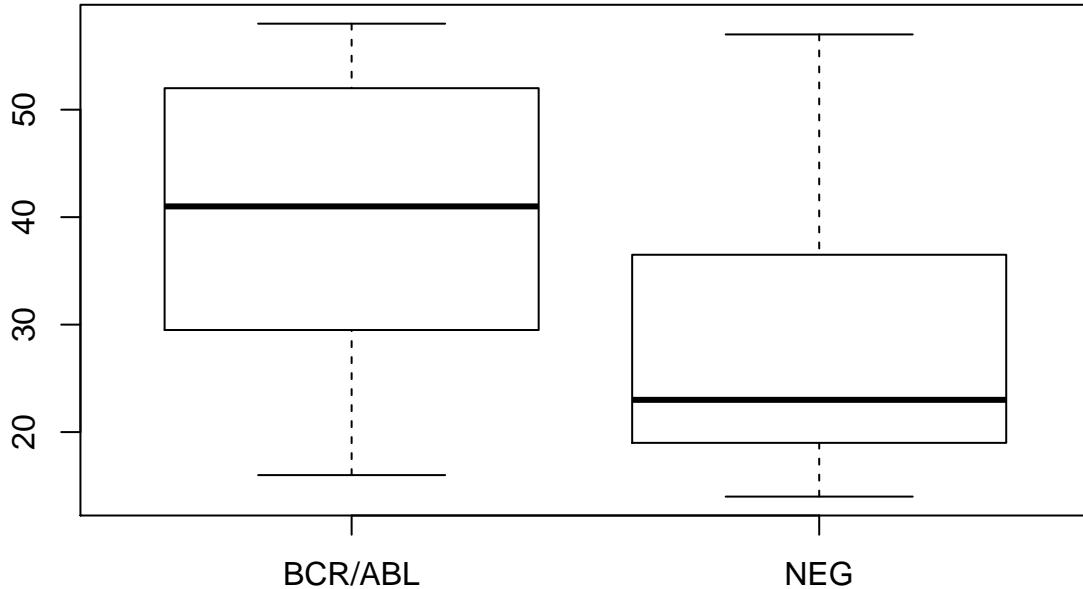
```
bcrabl$mol.biol <- factor(bcrabl$mol.biol)
table(bcrabl$mol.biol)
#>
#> BCR/ABL      NEG
#>      37      74
```

Exploratory data analysis

A very useful concept in *R* is the `formula`. This is specified in a form like `lhs ~ rhs`. The left-hand side is typically a response (dependent) variable, and the right-hand side a description of the independent variables that one is interested in. I 'say' a formula like `age ~ mol.biol` as "age as a function of mol.biol".

To get a sense of the use of formulas in *R*, and the ease with which one can explore data, use the `boxplot()` function to visualize the relationship between age as a function of molecular biology.

```
boxplot(age ~ mol.biol, bcrabl)
```



Consult the help page `?boxplot` to figure out how to add “Age” as the y-axis label.

The boxplot suggests that in our sample individuals with BCR/ABL are on average older than individuals classified as NEG. Confirm this using `t.test()`.

```
t.test(age ~ mol.biol, bcrabl)
#>
#> Welch Two Sample t-test
#>
#> data: age by mol.biol
#> t = 4.8172, df = 68.529, p-value = 8.401e-06
#> alternative hypothesis: true difference in means is not equal to 0
#> 95 percent confidence interval:
#> 7.13507 17.22408
#> sample estimates:
#> mean in group BCR/ABL      mean in group NEG
#>          40.25000           28.07042
```

It might be surprising that the `df` (degrees of freedom) are not a round number, but 68.592. This is because by default *R* performs a t-test that allows for variances to differ between groups. Consult the `?t.test` help page to figure out to perform a simpler form of the t-test, where variances are equal.

2.2.4 Writing *R* scripts

2.2.5 Using packages

R functions are made available in *packages*. All the functions that we have used so far are implemented in packages that are distributed with *R*. There are actually more than 15,000 *R* packages available for a wide variety of purposes. The general challenge we explore here is how to discover, install, and use one of these packages.

Discover

The most common place to find *R* packages is on the comprehensive *R* archive network, CRAN.

Visit CRAN at <https://cran.r-project.org>, click on the ‘Packages’ link, and the table of available packages sorted by name. Find the `ggplot2` package and peruse the information you are presented with. What does

this package do?

The large number of *R* packages can make finding the ‘best’ packages for a particular purpose difficult to identify. From <https://cran.r-project.org>, click on the ‘Task Views’ link and explore a topic interesting to you.

Install (once only per *R* installation)

One useful packages have been identified, they need to be installed into *R*. This only needs to be done once per *R* installation. The following installs the ggplot2 and tibble packages; you DO NOT need to do this command because ggplot2 is already installed on your AMI.

```
## No need to do this...
install.packages(c("ggplot2", "tibble"))
```

Use

Packages that are installed on your system are not available for use until they are attached to the current *R* session. Use `library()` to attach the ggplot2 package.

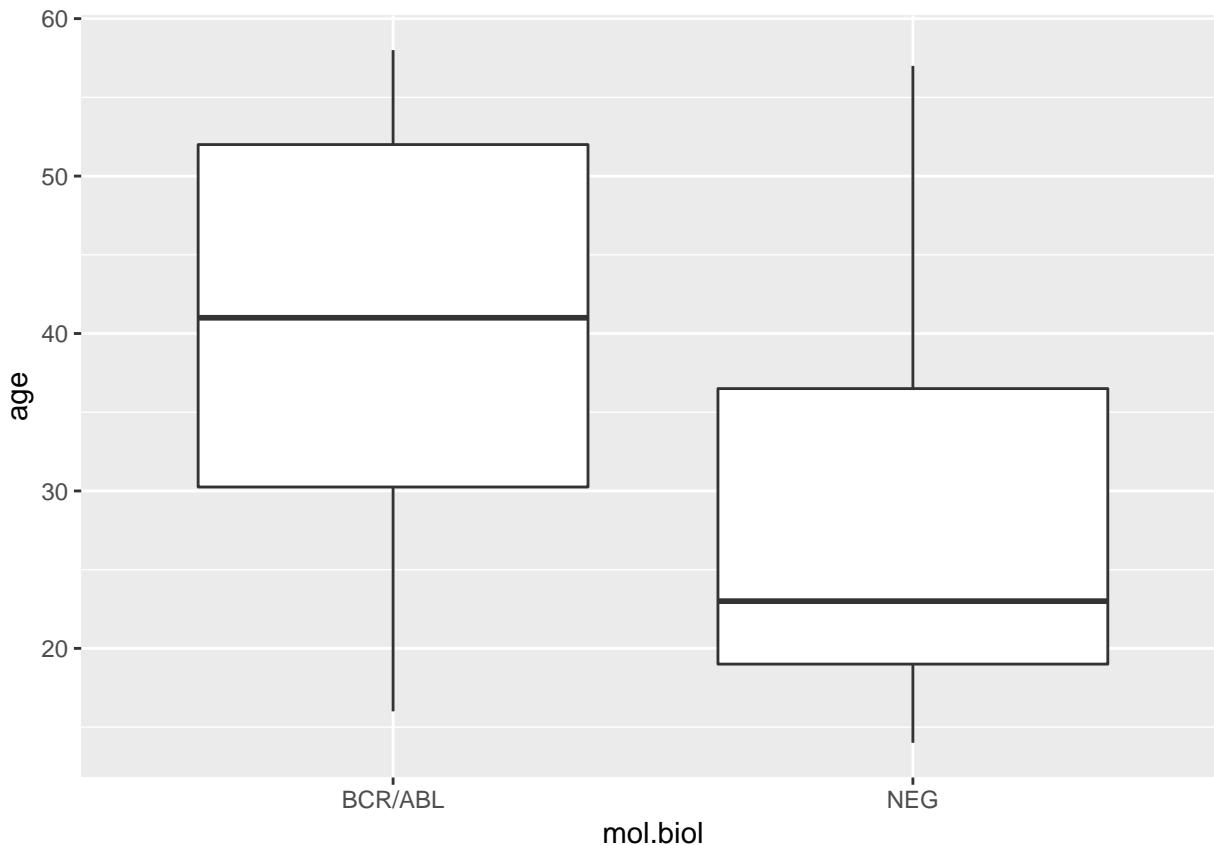
```
library("ggplot2")
```

Look up help on `?ggplot`. Peruse especially the examples at the bottom of the help page. The idea is that one identifies the data set to be used, and the variables with the data to be used as the x and y ‘aesthetics’

```
ggplot(brcabl, aes(x = mol.biol, y = age))
```

One then adds ‘geoms’ to the plot, for instance the boxplot geom

```
ggplot(brcabl, aes(x = mol.biol, y = age)) + geom_boxplot()
#> Warning: Removed 4 rows containing non-finite values (stat_boxplot).
```



Use `xlab()` and `ylab()` to enhance the plot with more meaningful x and y labels; see the help page `?xlab`,

especially the example section, for hints on how to modify the graph.

The grammar of graphics implemented by `ggplot2` is a very powerful and expressive system for producing appealing visualizations, and is widely used in the *R* community.

2.3 Introduction to *Bioconductor*

Bioconductor is a collection of more than 1,500 packages for the statistical analysis and comprehension of high-throughput genomic data. Originally developed for microarrays, *Bioconductor* packages are now used in a wide range of analyses, including bulk and single-cell RNA-seq, ChIP seq, copy number analysis, microarray methylation and classic expression analysis, flow cytometry, and many other domains.

This section of the workshop introduces the essential of *Bioconductor* package discovery, installation, and use.

2.3.1 Discovering, installing, and learning how to use *Bioconductor* packages

Discovery

The web site at <https://bioconductor.org> contains descriptions of all *Bioconductor* packages, as well as essential reference material for all levels of user.

Packages available in *Bioconductor* are summarized at <https://bioconductor.org/packages>, also linked from the front page of the web site. The widget on the left summarizes four distinct types of *Bioconductor* packages

- ‘Software’ packages implement particular analyses or other functionality, e.g., querying web-based resources or importing common file formats to *Bioconductor* objects.
- ‘Annotation’ packages contain data that can be helpful in placing analysis results in context, for example: mapping between gene symbols such as “BRCA1” and Ensembl or Entrez gene identifiers; classifying genes in terms of gene ontology; describing the genomic coordinates of exons, transcripts, and genes; and representing whole genome sequences of common organisms.
- ‘Experiment data’ packages contain highly curated data sets that can be useful for learning and teaching (e.g., the airway package and data set used in the DESeq2 package for the analysis of bulk RNA-seq differential expression) or placing results in context (e.g., the curatedTCGAData package for conveniently accessing TCGA data in a way that allows very smooth integration into *Bioconductor* analyses).
- ‘Workflow’ packages that summarize common work flows, e.g., simpleSingleCell for single-cell RNA-seq expression analysis.

Installation

Like CRAN packages, *Bioconductor* packages need to be installed only once per *R* installation, and then attached to each session where they are going to be used.

Bioconductor packages are installed slightly differently from CRAN packages. The first step is to install the `BiocManager` package from CRAN.

```
if (!"BiocManager" %in% rownames(installed.packages()))
  install.packages("BiocManager", repos = "https://cran.r-project.org")
```

The next step is to install the desired *Bioconductor* packages. The syntax to install the `rtracklayer` and `GenomicRanges` packages is

```
BiocManager::install(c("rtracklayer", "GenomicRanges"))
```

Bioconductor packages tend to depend on one another quite a lot, so it is important that the correct versions of all packages are installed. Validate your installation (not necessary during the course) with

```
BiocManager::valid()
```

A convenient function in BiocManager is `available()`, which accepts a regular expression to find matching packages. The following finds all ‘TxDb’ packages (describing exon, transcript, and gene coordinates) for *Homo sapiens*.

```
BiocManager::available("TxDb.Hsapiens")
#> [1] "TxDb.Hsapiens.BioMart.igis"
#> [2] "TxDb.Hsapiens.UCSC.hg18.knownGene"
#> [3] "TxDb.Hsapiens.UCSC.hg19.knownGene"
#> [4] "TxDb.Hsapiens.UCSC.hg19.lincRNAsTranscripts"
#> [5] "TxDb.Hsapiens.UCSC.hg38.knownGene"
```

Learning and support

Each package is linked to a ‘landing page’, e.g., DESeq2 that contains a description of the package, authors, perhaps literature citations where the software is described, and installation instructions.

An important part of *Bioconductor* packages are ‘vignettes’ which describe how the package is to be used. Vignettes are linked from package landing pages, and are available from within *R* using

```
browseVignettes("simpleSingleCell")
```

Users can get support on using packages at <https://support.bioconductor.org>, a question-and-answer style forum where responses usually come quickly and often from very knowledgeable users or the package developer. There are many additional sources of support, including course material linked from the home page.

2.3.2 Working with Genomic Ranges

This section introduces two useful packages for general-purpose work on genomic coordinates. The rtracklayer package provides the `import()` function to read many types of genomic files (e.g., BED, GTF, VCF, FASTA) into *Bioconductor* objects. The GenomicRanges package provides functions for manipulating genomic ranges, i.e., descriptions of exons, genes, ChIP peaks, called variants, … as coordinates in genome space.

Start by attaching the rtracklayer and GenomicRanges packages to our session.

```
library("rtracklayer")
library("GenomicRanges")
```

Importing data

We’ll read in a BED file derived from the UCSC genome browser. The file contains the coordinates of all CpG islands in the human genome, and is described at the UCSC table browser. Here are the first few lines of the file, giving the chromosome, start and end coordinates, and identifier of each CpG island.

```
chr1    155188536  155192004  CpG:_361
chr1    2226773 2229734 CpG:_366
chr1    36306229   36307408  CpG:_110
chr1    47708822   47710847  CpG:_164
chr1    53737729   53739637  CpG:_221
chr1    144179071  144179313 CpG:_20
```

Use `file.choose()` to find the file

```
fname <- file.choose()  # CpGislands.Hsapiens.hg38.UCSC.bed
```

```
fname
#> [1] "100_Morgan_RBiocForAll/CpGislands.Hsapiens.hg38.UCSC.bed"
file.exists(fname)
#> [1] TRUE
```

Then use `import()` from `rtracklayer` to read the data into *R*. The end result is a `GenomicRanges` object describing each CpG island.

```
cpg <- import(fname)
cpg
#> GRanges object with 30477 ranges and 1 metadata column:
#>           seqnames      ranges strand |      name
#>           <Rle>      <IRanges> <Rle> / <character>
#> [1]     chr1  155188537-155192004   * /  CpG:_361
#> [2]     chr1  2226774-2229734    * /  CpG:_366
#> [3]     chr1  36306230-36307408   * /  CpG:_110
#> [4]     chr1  47708823-47710847   * /  CpG:_164
#> [5]     chr1  53737730-53739637   * /  CpG:_221
#> ...
#> [30473] chr22_KI270734v1_random  131010-132049   * /  CpG:_102
#> [30474] chr22_KI270734v1_random  161257-161626   * /  CpG:_55
#> [30475] chr22_KI270735v1_random  172221-18098   * /  CpG:_100
#> [30476] chr22_KI270738v1_random  4413-5280    * /  CpG:_80
#> [30477] chr22_KI270738v1_random  6226-6467    * /  CpG:_34
#> -----
#> seqinfo: 254 sequences from an unspecified genome; no seqlengths
```

Closely compare the coordinates of the first few ranges from the file with the first few ranges in the *Bioconductor* representation. The BED format specification says that coordinates are 0-based, and intervals are half-open (the ‘start’ coordinate is in the range, the ‘end’ coordinate is immediately after the range; this makes some computations easy). *Bioconductor*’s convention is that coordinates are 1-based and closed (i.e., both start and end coordinates are included in the range). `rtracklayer`’s `import()` function has adjusted coordinates to follow *Bioconductor* conventions.

Working with genomic ranges

For convenience and to illustrate functionality, let’s work only with the ‘standard’ chromosomes 1 - 22 autosomal, X, and Y chromosomes. Look up the help page `?keepStandardChromosomes` for an explanation of `pruning.mode=`.

```
cpg <- keepStandardChromosomes(cpg, pruning.mode = "coarse")
cpg
#> GRanges object with 27949 ranges and 1 metadata column:
#>           seqnames      ranges strand |      name
#>           <Rle>      <IRanges> <Rle> / <character>
#> [1]     chr1  155188537-155192004   * /  CpG:_361
#> [2]     chr1  2226774-2229734    * /  CpG:_366
#> [3]     chr1  36306230-36307408   * /  CpG:_110
#> [4]     chr1  47708823-47710847   * /  CpG:_164
#> [5]     chr1  53737730-53739637   * /  CpG:_221
#> ...
#> [27945] chr22  50704375-50704880   * /  CpG:_38
#> [27946] chr22  50710878-50711294   * /  CpG:_41
#> [27947] chr22  50719959-50721632   * /  CpG:_180
#> [27948] chr22  50730600-50731304   * /  CpG:_65
#> [27949] chr22  50783345-50783889   * /  CpG:_63
```

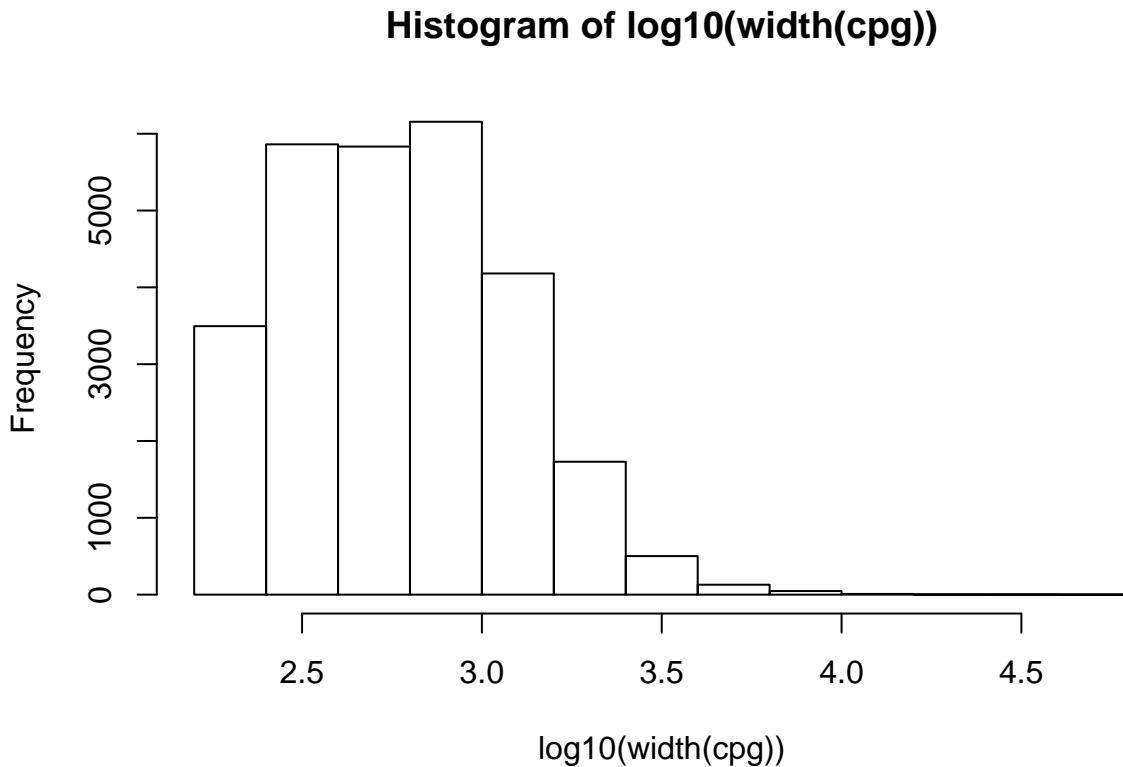
```
#> -----
#> seqinfo: 24 sequences from an unspecified genome; no seqlengths
```

There are two parts to a `GenomicRanges` object. The `seqnames` (chromosomes, in the present case), start and end coordinates, and strand are *required*. Additional elements such as `name` in the current example are optional. Required components are accessed by functions such as `start()`, `end()` and `width()`. Optional components can be accessed using the `$` notation.

```
head( start(cpg) )
#> [1] 155188537 2226774 36306230 47708823 53737730 144179072
head( cpg$name )
#> [1] "CpG:_361" "CpG:_366" "CpG:_110" "CpG:_164" "CpG:_221" "CpG:_20"
```

Use the `width()` accessor function to extract a vector of widths of each CpG island. Transform the values using `log10()`, and visualize the distribution using `hist()`.

```
hist(log10(width(cpg)))
```



Use `subset()` to select the CpG islands on chromosomes 1 and 2.

```
subset(cpg, seqnames %in% c("chr1", "chr2"))
#> GRanges object with 4217 ranges and 1 metadata column:
#>           seqnames          ranges strand |      name
#>           <Rle>          <IRanges> <Rle> / <character>
#> [1]     chr1 155188537-155192004   * /    CpG:_361
#> [2]     chr1 2226774-2229734     * /    CpG:_366
#> [3]     chr1 36306230-36307408   * /    CpG:_110
#> [4]     chr1 47708823-47710847   * /    CpG:_164
#> [5]     chr1 53737730-53739637   * /    CpG:_221
#> ...
#> [4213]   chr2 242003256-242004412  * /    CpG:_79
```

```
#> [4214] chr2 242006590-242010686      * / CpG:_263
#> [4215] chr2 242045491-242045723      * / CpG:_16
#> [4216] chr2 242046615-242047706      * / CpG:_170
#> [4217] chr2 242088150-242089411      * / CpG:_149
#> -----
#> seqinfo: 24 sequences from an unspecified genome; no seqlengths
```

Genomic annotations

Earlier we mentioned ‘Annotation data’ packages. An example is the TxDb.* family of packages. These packages contain information on the genomic coordinates of exons, genes, transcripts, etc. Attach the TxDb package corresponding to the *Homo sapiens* hg38 genome build using the UCSC ‘knownGene’ track.

```
library("TxDb.Hsapiens.UCSC.hg38.knownGene")
```

Extract the coordinates of all transcripts

```
tx <- transcripts(TxDb.Hsapiens.UCSC.hg38.knownGene)
tx
#> GRanges object with 197782 ranges and 2 metadata columns:
#>           seqnames      ranges strand |   tx_id    tx_name
#>           <Rle>      <IRanges> <Rle> | <integer> <character>
#> [1]     chr1    29554-31097    + /      1 uc057aty.1
#> [2]     chr1    30267-31109    + /      2 uc057atz.1
#> [3]     chr1    30366-30503    + /      3 uc031tlb.1
#> [4]     chr1    69091-70008    + /      4 uc001aal.1
#> [5]     chr1    160446-161525   + /      5 uc057aum.1
#> ...
#> [197778] chrUn_KI270750v1 148668-148843    + /    197778 uc064xrp.1
#> [197779] chrUn_KI270752v1      144-268    + /    197779 uc064xrq.1
#> [197780] chrUn_KI270752v1  21813-21944    + /    197780 uc064xrt.1
#> [197781] chrUn_KI270752v1  3497-3623    - /    197781 uc064xrr.1
#> [197782] chrUn_KI270752v1  9943-10067    - /    197782 uc064xrs.1
#> -----
#> seqinfo: 455 sequences (1 circular) from hg38 genome
```

Keep only the standard chromosomes, to work smoothly with our cpg object.

```
tx <- keepStandardChromosomes(tx, pruning.mode="coarse")
tx
#> GRanges object with 182435 ranges and 2 metadata columns:
#>           seqnames      ranges strand |   tx_id    tx_name
#>           <Rle>      <IRanges> <Rle> | <integer> <character>
#> [1]     chr1    29554-31097    + /      1 uc057aty.1
#> [2]     chr1    30267-31109    + /      2 uc057atz.1
#> [3]     chr1    30366-30503    + /      3 uc031tlb.1
#> [4]     chr1    69091-70008    + /      4 uc001aal.1
#> [5]     chr1    160446-161525   + /      5 uc057aum.1
#> ...
#> [182431]   chrM    5826-5891    - /    182431 uc064xpa.1
#> [182432]   chrM    7446-7514    - /    182432 uc064xpb.1
#> [182433]   chrM    14149-14673   - /    182433 uc064xpm.1
#> [182434]   chrM    14674-14742   - /    182434 uc022bqv.3
#> [182435]   chrM    15956-16023   - /    182435 uc022bqx.2
#> -----
#> seqinfo: 25 sequences (1 circular) from hg38 genome
```

Overlaps

A very useful operation is to count overlaps in two distinct genomic ranges objects. The following counts the number of CpG islands that overlap each transcript. Related functions include `findOverlaps()`, `nearest()`, `precede()`, and `follow()`.

```
olaps <- countOverlaps(tx, cpg)
length(olaps)      # 1 count per transcript
#> [1] 182435
table(olaps)
#> olaps
#>   0   1   2   3   4   5   6   7   8   9   10  11
#> 94621 70551 10983 3228 1317 595 351 214 153 93 64 39
#> 12   13   14   15   16   17   18   19   20   21  22  23
#> 41   31   21   20   17   8    14   8    7    3    6    6
#> 24   25   26   27   28   29   30   31   32   33   34  35
#> 3    2    4    1    3    2    6    5    3    3    1    2
#> 36   37   38   63
#> 3    1    1    4
```

Calculations such as `countOverlaps()` can be added to the `GRanges` object, tightly coupling derived data with the original annotation.

```
tx$cpgOverlaps <- countOverlaps(tx, cpg)
tx
#> GRanges object with 182435 ranges and 3 metadata columns:
#>           seqnames      ranges strand | tx_id      tx_name
#>           <Rle>      <IRanges>  <Rle> | <integer> <character>
#> [1] chr1    29554-31097    + /      1 uc057aty.1
#> [2] chr1    30267-31109    + /      2 uc057atz.1
#> [3] chr1    30366-30503    + /      3 uc031tlb.1
#> [4] chr1    69091-70008    + /      4 uc001aal.1
#> [5] chr1    160446-161525   + /      5 uc057aum.1
#> ...
#> [182431] chrM    5826-5891    - /    182431 uc064xpa.1
#> [182432] chrM    7446-7514    - /    182432 uc064xpb.1
#> [182433] chrM    14149-14673   - /    182433 uc064xpm.1
#> [182434] chrM    14674-14742   - /    182434 uc022bqv.3
#> [182435] chrM    15956-16023   - /    182435 uc022bqx.2
#>           cpgOverlaps
#>           <integer>
#> [1]      1
#> [2]      0
#> [3]      0
#> [4]      0
#> [5]      0
#> ...
#> [182431] 0
#> [182432] 0
#> [182433] 0
#> [182434] 0
#> [182435] 0
#> -----
#> seqinfo: 25 sequences (1 circular) from hg38 genome
```

It is then possible to perform coordinated actions, e.g., subsetting the `GRanges` objects for transcripts

satisfying particular conditions, in a coordinated fashion where the software does all the book-keeping to makes sure the correct ranges are selected.

```
subset(tx, cpgOverlaps > 10)
#> GRanges object with 265 ranges and 3 metadata columns:
#>   seqnames      ranges strand |  tx_id    tx_name
#>   <Rle>      <IRanges>  <Rle> | <integer> <character>
#> [1] chr1  2050470-2185395    + /     213 uc001aiq.3
#> [2] chr1  2050485-2146108    + /     214 uc057bkd.1
#> [3] chr1  2073462-2185390    + /     219 uc001air.4
#> [4] chr1  2073986-2185190    + /     221 uc010nyw.3
#> [5] chr1  2104716-2185393    + /     227 uc001ais.4
#> ...
#> ...
#> ...
#> [261] chrX 40051246-40177329    - /    179887 uc004deo.4
#> [262] chrX 40051248-40177329    - /    179889 uc004dep.5
#> [263] chrX 40062955-40177320    - /    179892 uc064you.1
#> [264] chry  333963-386710     - /    182211 uc004fot.4
#> [265] chry  344896-386955     - /    182217 uc011nah.3
#>   cpgOverlaps
#>   <integer>
#> [1]    15
#> [2]    11
#> [3]    13
#> [4]    13
#> [5]    11
#> ...
#> ...
#> [261]   11
#> [262]   11
#> [263]   11
#> [264]   14
#> [265]   12
#> -----
#> seqinfo: 25 sequences (1 circular) from hg38 genome
```

Can you think of other situations where one might calculate derived values and couple these with `GRanges` or similar objects?

2.3.3 Working with summarized experimental data

This section introduces another broadly useful package and data structure, the `SummarizedExperiment` package and `SummarizedExperiment` object.

The `SummarizedExperiment` object has matrix-like properties – it has two dimensions and can be subset by ‘rows’ and ‘columns’. The `assay()` data of a `SummarizedExperiment` experiment contains one or more matrix-like objects where rows represent features of interest (e.g., genes), columns represent samples, and elements of the matrix represent results of a genomic assay (e.g., counts of reads overlaps genes in each sample of an bulk RNA-seq differential expression assay).

Object construction

The `SummarizedExperiment` coordinates assays with (optional) descriptions of rows and columns. We start by reading in a simple `data.frame` describing 8 samples from an RNASeq experiment looking at dexamethasone treatment across 4 human smooth muscle cell lines; use `browseVignettes("airway")` for a more complete description of the experiment and data processing. Read the column data in using `file.choose()` and `read.csv()`.

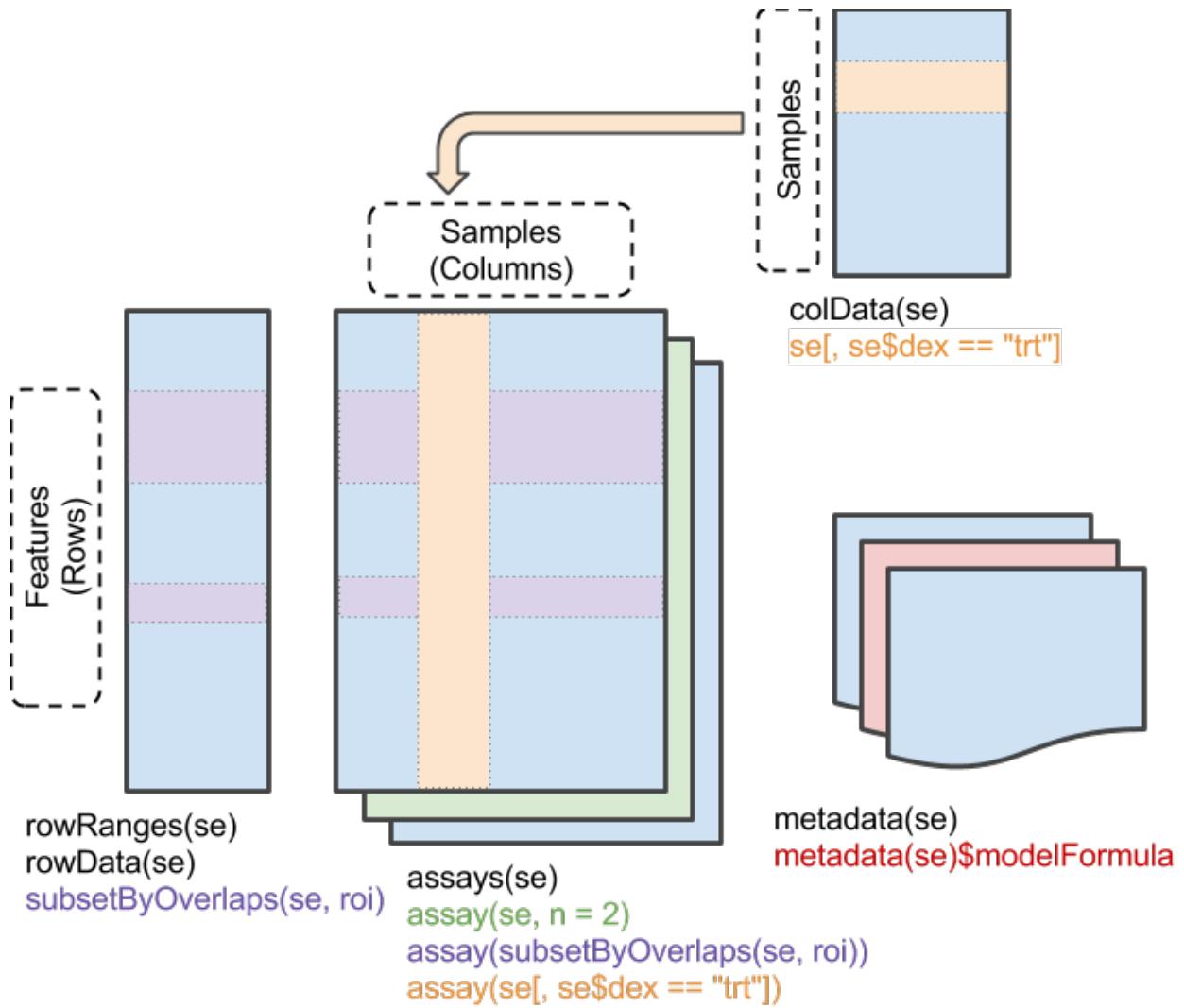


Figure 2.2:

```
fname <- file.choose() # airway_colData.csv
fname
```

We want the first column the the data to be treated as row names (sample identifiers) in the `data.frame`, so `read.csv()` has an extra argument to indicate this.

```
colData <- read.csv(fname, row.names = 1)
colData
#>      SampleName   cell   dex albut      Run avgLength Experiment
#> SRR1039508 GSM1275862 N61311 untrt untrt SRR1039508      126  SRX384345
#> SRR1039509 GSM1275863 N61311    trt untrt SRR1039509      126  SRX384346
#> SRR1039512 GSM1275866 N052611 untrt untrt SRR1039512      126  SRX384349
#> SRR1039513 GSM1275867 N052611    trt untrt SRR1039513      87   SRX384350
#> SRR1039516 GSM1275870 N080611 untrt untrt SRR1039516     120   SRX384353
#> SRR1039517 GSM1275871 N080611    trt untrt SRR1039517      126  SRX384354
#> SRR1039520 GSM1275874 N061011 untrt untrt SRR1039520     101   SRX384357
#> SRR1039521 GSM1275875 N061011    trt untrt SRR1039521      98   SRX384358
#>
#>      Sample   BioSample
#> SRR1039508 SRS508568 SAMN02422669
#> SRR1039509 SRS508567 SAMN02422675
#> SRR1039512 SRS508571 SAMN02422678
#> SRR1039513 SRS508572 SAMN02422670
#> SRR1039516 SRS508575 SAMN02422682
#> SRR1039517 SRS508576 SAMN02422673
#> SRR1039520 SRS508579 SAMN02422683
#> SRR1039521 SRS508580 SAMN02422677
```

The data are from the Short Read Archive, and the row names, `SampleName`, `Run`, `Experiment`, `Sample`, and `BioSample` columns are classifications from the archive. Additional columns include:

- `cell`: the cell line used. There are four cell lines.
- `dex`: whether the sample was untreated, or treated with dexamethasone.
- `albut`: a second treatment, which we ignore
- `avgLength`: the sample-specific average length of the RNAseq reads estimated in the experiment.

Assay data

Now import the assay data from the file “airway_counts.csv”

```
fname <- file.choose() # airway_counts.csv
fname

counts <- read.csv(fname, row.names=1)
```

Although the data are read as a `data.frame`, all columns are of the same type (integer-valued) and represent the same attribute; the data is really a `matrix` rather than `data.frame`, so we coerce to matrix using `as.matrix()`.

```
counts <- as.matrix(counts)
```

We see the dimensions and first few rows of the counts matrix

```
dim(counts)
#> [1] 33469      8
head(counts)
#>      SRR1039508 SRR1039509 SRR1039512 SRR1039513 SRR1039516
#> ENSG00000000003      679      448      873      408      1138
#> ENSG00000000419      467      515      621      365      587
#> ENSG00000000457      260      211      263      164      245
```

```
#> ENSG000000000460      60      55      40      35      78
#> ENSG000000000938      0       0       2       0       1
#> ENSG000000000971    3251    3679    6177    4252    6721
#>           SRR1039517 SRR1039520 SRR1039521
#> ENSG000000000003    1047    770     572
#> ENSG000000000419    799     417     508
#> ENSG000000000457    331     233     229
#> ENSG000000000460     63      76      60
#> ENSG000000000938     0       0       0
#> ENSG000000000971   11027   5176    7995
```

It's interesting to think about what the counts mean – for ENSG000000000003, sample SRR1039508 had 679 reads that overlapped this gene, sample SRR1039509 had 448 reads, etc. Notice that for this gene there seems to be a consistent pattern – within a cell line, the read counts in the untreated group are always larger than the read counts for the treated group. This and other basic observations from 'looking at' the data motivate many steps in a rigorous RNASeq differential expression analysis.

Creating a `SummarizedExperiment` object

We saw earlier that there was considerable value in tightly coupling the count of CpG islands overlapping each transcript with the `GRanges` describing the transcripts. We can anticipate that close coupling of the column data with the assay data will have similar benefits, e.g., reducing the chances of bookkeeping errors as we work with our data.

Attach the `SummarizedExperiment` library to our *R* session.

```
library("SummarizedExperiment")
```

Use the `SummarizedExperiment()` function to coordinate the assay and column data; this function uses row and column names to make sure the correct assay columns are described by the correct column data rows.

```
se <- SummarizedExperiment(assay = counts, colData = colData)
se
#> class: SummarizedExperiment
#> dim: 33469 8
#> metadata(0):
#> assays(1): ''
#> rownames(33469): ENSG000000000003 ENSG000000000419 ...
#>   ENSG00000273492 ENSG00000273493
#> rowData names(0):
#> colnames(8): SRR1039508 SRR1039509 ... SRR1039520 SRR1039521
#> colData names(9): SampleName cell ... Sample BioSample
```

It is straight-forward to use `subset()` on `SummarizedExperiment` to create subsets of the data in a coordinated way. Remember that a `SummarizedExperiment` is conceptually two-dimensional (matrix-like), and in the example below we are subsetting on the second dimension.

```
subset(se, , dex == "trt")
#> class: SummarizedExperiment
#> dim: 33469 4
#> metadata(0):
#> assays(1): ''
#> rownames(33469): ENSG000000000003 ENSG000000000419 ...
#>   ENSG00000273492 ENSG00000273493
#> rowData names(0):
#> colnames(4): SRR1039509 SRR1039513 SRR1039517 SRR1039521
#> colData names(9): SampleName cell ... Sample BioSample
```

As with **GRanges**, there are accessors that extract data from the **SummarizedExperiment**. For instance, we can use **assay()** to extract the count matrix, and **colSums()** to calculate the library size (total number of reads overlapping genes in each sample).

```
colSums(assay(se))
#> SRR1039508 SRR1039509 SRR1039512 SRR1039513 SRR1039516 SRR1039517
#> 20637971 18809481 25348649 15163415 24448408 30818215
#> SRR1039520 SRR1039521
#> 19126151 21164133
```

Note that library sizes differ by a factor of 2 from largest to smallest; how would this influence the interpretation of counts in individual cells of the assay data?

As with **GRanges**, it might be useful to remember important computations in a way that is robust, e.g.,

```
se$lib.size <- colSums(assay(se))
colData(se)
#> DataFrame with 8 rows and 10 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 lib.size
#>           <factor> <factor> <factor> <numeric>
#> SRR1039508 SRX384345 SRS508568 SAMN02422669 20637971
#> SRR1039509 SRX384346 SRS508567 SAMN02422675 18809481
#> SRR1039512 SRX384349 SRS508571 SAMN02422678 25348649
#> SRR1039513 SRX384350 SRS508572 SAMN02422670 15163415
#> SRR1039516 SRX384353 SRS508575 SAMN02422682 24448408
#> SRR1039517 SRX384354 SRS508576 SAMN02422673 30818215
#> SRR1039520 SRX384357 SRS508579 SAMN02422683 19126151
#> SRR1039521 SRX384358 SRS508580 SAMN02422677 21164133
```

2.3.4 Down-stream analysis

In this final section we quickly hint at down-stream analysis, and the way in which skills learned in working with *Bioconductor* objects in one package translate to working with objects in other packages.

We start by loading the **DESeq2** package, a very popular facility for analysing bulk RNAseq differential expression data.

```
library("DESeq2")
```

The package requires count data like that in the **SummarizedExperiment** we have been working with, in addition to a **formula** describing the experimental design. Some of the observations above suggest that we should include cell line as a covariate, and dexamethazone treatment as the main factor that we are interested in.

```
dds <- DESeqDataSet(se, design = ~ cell + dex)
#> renaming the first element in assays to 'counts'
```

```
dds
#> class: DESeqDataSet
#> dim: 33469 8
#> metadata(1): version
#> assays(1): counts
#> rownames(33469): ENSG00000000003 ENSG000000000419 ...
#>   ENSG00000273492 ENSG00000273493
#> rowData names(0):
#> colnames(8): SRR1039508 SRR1039509 ... SRR1039520 SRR1039521
#> colData names(10): SampleName cell ... BioSample lib.size
```

The `dds` object can be manipulated very much like a `SummarizedExperiment`.

The essential DESeq work flow is summarized by a single function call, which performs advanced statistical analysis on the data in the `dds` object.

```
dds <- DESeq(dds)
#> estimating size factors
#> estimating dispersions
#> gene-wise dispersion estimates
#> mean-dispersion relationship
#> final dispersion estimates
#> fitting model and testing
```

A table summarizing measures of differential expression can be extracted from the object, and visualized or manipulated using commands we learned earlier today.

```
results(dds)
#> log2 fold change (MLE): dex untrt vs trt
#> Wald test p-value: dex untrt vs trt
#> DataFrame with 33469 rows and 6 columns
#>
#>           baseMean    log2FoldChange      lfcSE
#>           <numeric>      <numeric>      <numeric>
#> ENSG00000000003  708.602169691234  0.381253982523047 0.100654411865821
#> ENSG000000000419  520.297900552084 -0.206812601085051 0.112218646507256
#> ENSG000000000457  237.163036796015 -0.037920431205081 0.143444654934088
#> ENSG000000000460  57.9326331250967  0.0881681758701336 0.287141822937769
#> ENSG000000000938  0.318098378392895  1.37822703433213  3.49987280259267
#> ...
#>           ...          ...          ...
#> ENSG00000273487  8.1632349843654 -1.04640654119647 0.698966822033151
#> ENSG00000273488  8.58447903624707 -0.107830154768606 0.638099034888812
#> ENSG00000273489  0.275899382507797 -1.48373838425916 3.51394583815493
#> ENSG00000273492  0.105978355992386  0.463679009985687 3.52308267589304
#> ENSG00000273493  0.106141666408122  0.521354507611989 3.53139024554344
#>
#>           stat      pvalue
#>           <numeric>      <numeric>
#> ENSG00000000003  3.78775232457058 0.000152016273044953
#> ENSG000000000419 -1.84294328546976 0.0653372915097615
#> ENSG000000000457 -0.264355832725208 0.79150574160836
#> ENSG000000000460  0.307054454722334 0.758801923982929
#> ENSG000000000938  0.393793463954221 0.693733530342183
#> ...
#>           ...          ...
#> ENSG00000273487 -1.49707612466166 0.134373451371486
#> ENSG00000273488 -0.168986550477068 0.865807220420437
#> ENSG00000273489 -0.422242815511986 0.672847792938571
```

```
#> ENSG00000273492 0.131611731157615 0.895291406125436
#> ENSG00000273493 0.147634351165219 0.882631343839791
#>                               padj
#>                               <numeric>
#> ENSG00000000003 0.00127423101752341
#> ENSG000000000419 0.195433088560704
#> ENSG000000000457 0.910602138773154
#> ENSG000000000460 0.893977566267183
#> ENSG000000000938 NA
#> ...
#> ENSG00000273487 0.327756206316493
#> ENSG00000273488 0.944617232242564
#> ENSG00000273489 NA
#> ENSG00000273492 NA
#> ENSG00000273493 NA
```

2.4 Summary

Chapter 3

101: Introduction to Bioconductor annotation resources

3.1 Instructors

- James W. MacDonald (jmacdon@uw.edu)
- Lori Shepherd (lori.shepherd@roswellpark.org)

3.2 Workshop Description

There are various annotation packages provided by the Bioconductor project that can be used to incorporate additional information to results from high-throughput experiments. This can be as simple as mapping Ensembl IDs to corresponding HUGO gene symbols, to much more complex queries involving multiple data sources. In this workshop we will cover the various classes of annotation packages, what they contain, and how to use them efficiently.

3.2.1 Prerequisites

- Basic knowledge of R syntax
- Basic understanding of the various annotation sources (NCBI, EBI/EMBL)

Useful background reading

- The AnnotationDbi vignette.
- The biomaRt vignette.
- The GenomicFeatures vignette.

3.2.2 Workshop Participation

After each type of annotation package is introduced, students will be given the opportunity to practice making their own queries.

3.2.3 *R / Bioconductor* packages used

- AnnotationDbi
- AnnotationHub
- BSgenome
- biomaRt
- ensemblDb
- org.Hs.eg.db
- TxDb.Hsapiens.UCSC.hg19.knownGene
- EnsDb.Hsapiens.v79
- EnsDb.Mmusculus.v79
- Homo.sapiens
- BSgenome.Hsapiens.UCSC.hg19
- hugene20sttranscriptcluster.db

3.3 Workshop goals and objectives

Annotating data is a complex task. For any high-throughput experiment the analyst usually starts with a set of identifiers for each thing that was measured, and in order to make the results useful to collaborators these identifiers need to be mapped to other identifiers that are either more familiar to collaborators, or that can be used for further analyses. As an example, RNA-Seq data may only have Entrez Gene IDs for each gene measured, and as part of the output you may want to include the gene symbols, which are more likely to be familiar to a Biologist.

3.3.1 Learning goals

- Understand what sort of annotation data are available
- Understand the difference between annotation sources (NCBI and EBI/EMBL)
- Gain familiarity with the various ways to query annotation packages
- Get some practice making queries

3.3.2 Learning objectives

- Be able to use select and mapIds to map between identifiers
- Be able to extract data from TxDb and EnsDb packages
- Be able to make queries using biomaRt
- Extract and utilize various data from AnnotationHub

3.4 Annotation Workshop

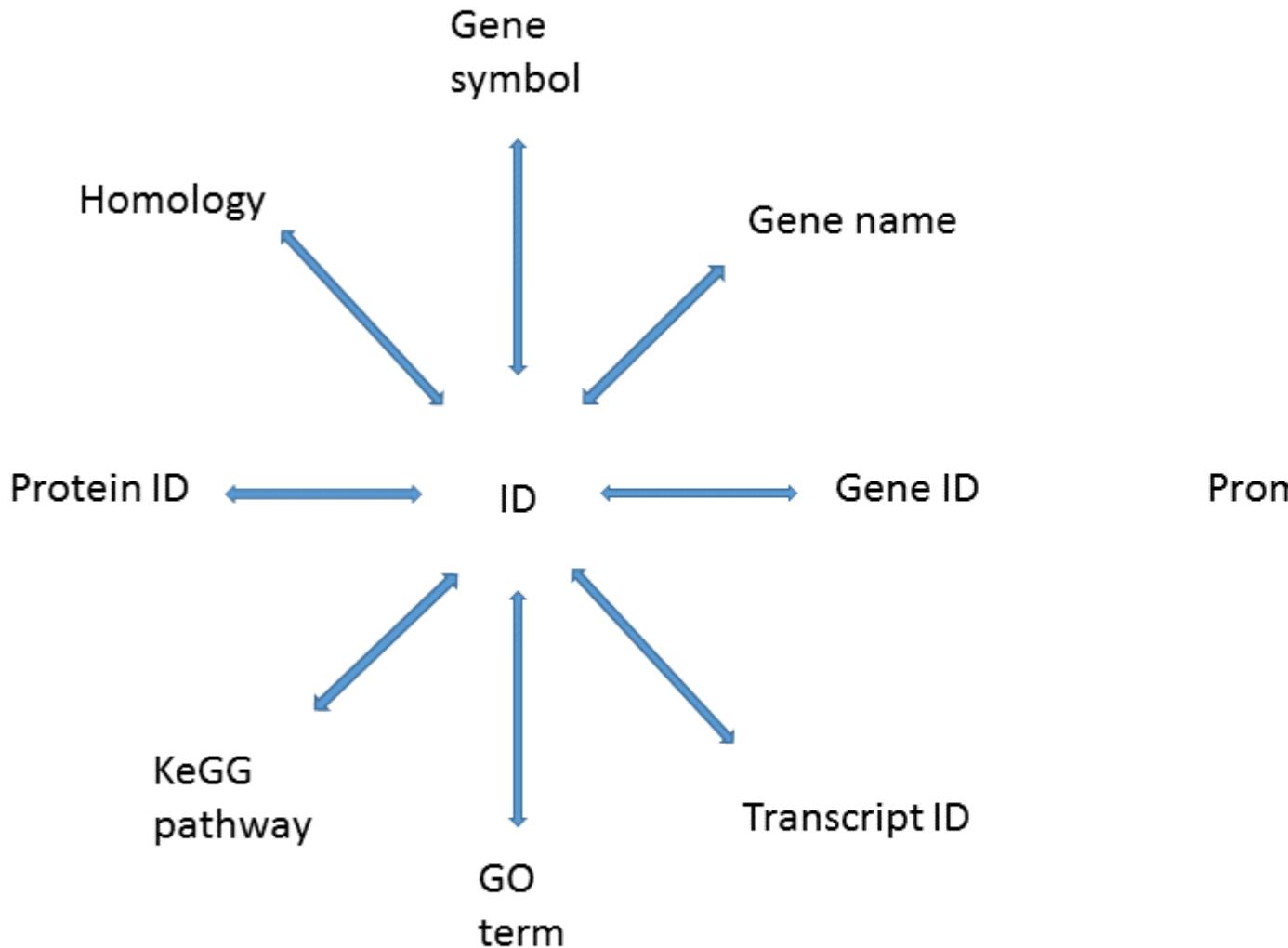
3.4.1 Goals for this workshop

- Learn about various annotation package types
- Learn the basics of querying these resources
- Discuss annotations in regard to Bioc data structures
- Get in some practice

3.4.2 What do we mean by annotation?

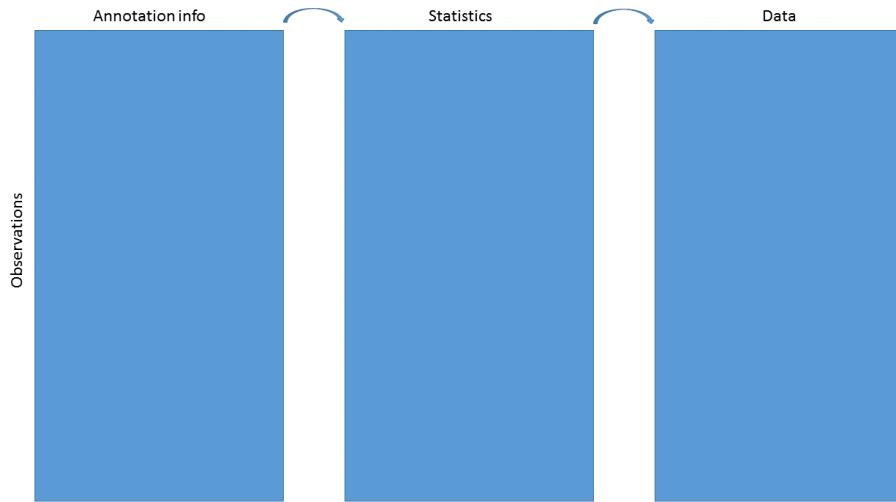
Map a known ID to other functional or positional information

Functional Annotation



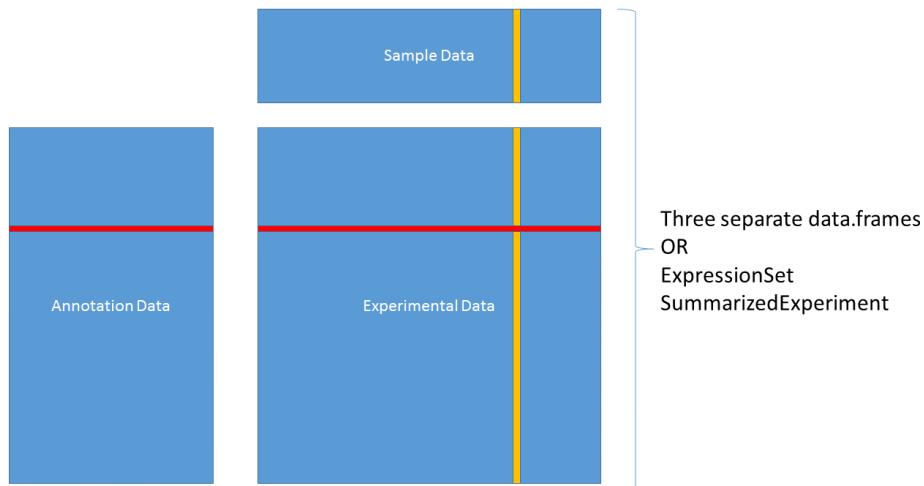
3.4.3 Specific goal

We have data and statistics, and we want to add other useful information



The end result might be as simple as a data.frame or HTML table, or as complex as a `RangedSummarizedExperiment`

3.4.4 Data containers



3.4.5 ExpressionSet

```
load(system.file("extdata/eset.Rdata", package = "Bioc2018Anno"))
eset
#> ExpressionSet (storageMode: lockedEnvironment)
#> assayData: 33552 features, 6 samples
#> element names: exprs
#> protocolData: none
#> phenoData
#>   sampleNames: GSM2194079 GSM2194080 ... GSM2194084 (6 total)
#>   varLabels: title characteristics_ch1.1
#>   varMetadata: labelDescription
```

```
#> featureData
#>   featureNames: 16657436 16657440 ... 17118478 (33552 total)
#>   fvarLabels: PROBEID ENTREZID SYMBOL GENENAME
#>   fvarMetadata: labelDescription
#> experimentData: use 'experimentData(object)'
#> Annotation: pd.hugene.2.0.st
```

3.4.6 ExpressionSet (continued)

```
head(exprs(eset))
#>          GSM2194079  GSM2194080  GSM2194081  GSM2194082  GSM2194083  GSM2194084
#> 16657436    8.505158   9.046577   8.382674   9.115481   8.715343   8.566301
#> 16657440    7.948860   8.191222   7.901911   8.459781   8.191793   8.219658
#> 16657450   10.932934  11.228553  10.948120  11.462231  11.300046  11.300886
#> 16657469    9.172462   9.344630   9.193450   9.465584   9.464020   9.135715
#> 16657473    6.222049   6.551035   6.000246   6.398798   5.892654   5.592125
#> 16657476    8.514300   8.474073   8.407196   8.811238   8.780833   8.874606
head(pData(phenoData(eset)))
#>                      title characteristics_ch1.1
#> GSM2194079  SW620-miR625-rep1      shRNA: miR-625-3p
#> GSM2194080  SW620-miR625-rep2      shRNA: miR-625-3p
#> GSM2194081  SW620-miR625-rep3      shRNA: miR-625-3p
#> GSM2194082  SW620-scramble-rep1      shRNA: scramble
#> GSM2194083  SW620-scramble-rep2      shRNA: scramble
#> GSM2194084  SW620-scramble-rep3      shRNA: scramble
```

3.4.7 ExpressionSet (continued)

```
head(pData(featureData(eset)))
#>          PROBEID  ENTREZID      SYMBOL
#> 16657436 16657436    84771    DDX11L2
#> 16657440 16657440  100302278   MIR1302-2
#> 16657450 16657450    402483   LINC01000
#> 16657469 16657469    140849  LINC00266-1
#> 16657473 16657473    729759    OR4F29
#> 16657476 16657476    388574  RPL23AP87
#>
#>                               GENENAME
#> 16657436          DEAD/H-box helicase 11 like 2
#> 16657440          microRNA 1302-2
#> 16657450          long intergenic non-protein coding RNA 1000
#> 16657469          long intergenic non-protein coding RNA 266-1
#> 16657473 olfactory receptor family 4 subfamily F member 29
#> 16657476          ribosomal protein L23a pseudogene 87
```

3.4.8 BioC containers vs basic structures

3.4.8.1 Pros

- Validity checking
- Subsetting
- Function dispatch
- Automatic behaviors

3.4.8.2 Cons

- Difficult to create
- Cumbersome to extract data by hand
- Useful only within R

3.4.9 Annotation sources

Package type	Example
ChipDb	hugene20sttranscriptcluster.db
OrgDb	org.Hs.eg.db
TxDb/EnsDb	TxDb.Hsapiens.UCSC.hg19.knownGene; EnsDb.Hsapiens.v75
OrganismDb	Homo.sapiens
BSgenome	BSgenome.Hsapiens.UCSC.hg19
Others	GO.db; KEGG.db
AnnotationHub	Online resource
biomaRt	Online resource

3.4.10 Interacting with AnnoDb packages

The main function is `select`:

```
select(annopkg, keys, columns, keytype)
```

Where

- *annopkg* is the annotation package
- *keys* are the IDs that we **know**
- *columns* are the values we **want**
- *keytype* is the type of key used
 - if the *keytype* is the **central** key, it can remain unspecified

3.4.11 Simple example

Say we have analyzed data from an Affymetrix Human Gene ST 2.0 array and want to know what the genes are. For purposes of this lab, we just select some IDs at random.

```
library(hugene20sttranscriptcluster.db)
set.seed(12345)
ids <- featureNames(eset)[sample(1:25000, 5)]
ids
#> [1] "16908472" "16962185" "16920686" "16965513" "16819952"
select(hugene20sttranscriptcluster.db, ids, "SYMBOL")
#> 'select()' returned 1:1 mapping between keys and columns
#>   PROBEID      SYMBOL
#> 1 16908472 LINC01494
#> 2 16962185     ALG3
#> 3 16920686    <NA>
#> 4 16965513    <NA>
#> 5 16819952    CBFB
```

3.4.12 Questions!

How do you know what the central keys are?

- If it's a ChipDb, the central key are the manufacturer's probe IDs
- It's sometimes in the name - org.Hs.eg.db, where 'eg' means Entrez Gene ID
- You can see examples using e.g., head(keys(*annopkg*)), and infer from that
- But note that it's never necessary to know the central key, as long as you specify the keytype

3.4.13 More questions!

What keytypes or columns are available for a given annotation package?

```
keytypes(hugene20sttranscriptcluster.db)
#> [1] "ACCCNUM"        "ALIAS"          "ENSEMBL"        "ENSEMBLPROT"
#> [5] "ENSEMBLTRANS"   "ENTREZID"       "ENZYME"         "EVIDENCE"
#> [9] "EVIDENCEALL"    "GENENAME"       "GO"             "GOALL"
#> [13] "IPI"           "MAP"            "OMIM"           "ONTOLOGY"
#> [17] "ONTOLOGYALL"   "PATH"           "PFAM"           "PMID"
#> [21] "PROBEID"       "PROSITE"        "REFSEQ"         "SYMBOL"
#> [25] "UCSCKG"        "UNIGENE"        "UNIPROT"
columns(hugene20sttranscriptcluster.db)
#> [1] "ACCCNUM"        "ALIAS"          "ENSEMBL"        "ENSEMBLPROT"
#> [5] "ENSEMBLTRANS"   "ENTREZID"       "ENZYME"         "EVIDENCE"
#> [9] "EVIDENCEALL"    "GENENAME"       "GO"             "GOALL"
#> [13] "IPI"           "MAP"            "OMIM"           "ONTOLOGY"
#> [17] "ONTOLOGYALL"   "PATH"           "PFAM"           "PMID"
#> [21] "PROBEID"       "PROSITE"        "REFSEQ"         "SYMBOL"
#> [25] "UCSCKG"        "UNIGENE"        "UNIPROT"
```

3.4.14 Another example

There is one issue with `select` however.

```
ids <- c('16737401','16657436' , '16678303')
select(hugene20sttranscriptcluster.db, ids, c("SYMBOL","MAP"))
#> 'select()' returned 1:many mapping between keys and columns
#>   PROBEID      SYMBOL      MAP
#> 1 16737401      TRAF6    11p12
#> 2 16657436      DDX11L1  1p36.33
#> 3 16657436 LOC102725121 1p36.33
#> 4 16657436      DDX11L2  2q14.1
#> 5 16657436      DDX11L9  15q26.3
#> 6 16657436      DDX11L10 16p13.3
#> 7 16657436      DDX11L5  9p24.3
#> 8 16657436      DDX11L16 Xq28
#> 9 16657436      DDX11L16 Yq12
#> 10 16678303     ARF1    1q42.13
```

3.4.15 The `mapIds` function

An alternative to `select` is `mapIds`, which gives control of duplicates

- Same arguments as `select` with slight differences
 - The columns argument can only specify one column
 - The keytype argument **must** be specified
 - An additional argument, multiVals used to control duplicates

```
mapIds(hugene20sttranscriptcluster.db, ids, "SYMBOL", "PROBEID")
#> 'select()' returned 1:many mapping between keys and columns
#> 16737401 16657436 16678303
#> "TRAF6" "DDX11L1" "ARF1"
```

3.4.16 Choices for `multiVals`

Default is `first`, where we just choose the first of the duplicates. Other choices are `list`, `CharacterList`, `filter`, `asNA` or a user-specified function.

```
mapIds(hugene20sttranscriptcluster.db, ids, "SYMBOL", "PROBEID", multiVals = "list")
#> 'select()' returned 1:many mapping between keys and columns
#> $`16737401` 
#> [1] "TRAF6"
#>
#> $`16657436` 
#> [1] "DDX11L1"      "LOC102725121" "DDX11L2"      "DDX11L9"
#> [5] "DDX11L10"     "DDX11L5"       "DDX11L16"    
#>
#> $`16678303` 
#> [1] "ARF1"
```

3.4.17 Choices for multiVals (continued)

```
mapIds(hugene20sttranscriptcluster.db, ids, "SYMBOL", "PROBEID", multiVals = "CharacterList")
#> 'select()' returned 1:many mapping between keys and columns
#> CharacterList of length 3
#> [[["16737401"]]] TRAF6
#> [[["16657436"]]] DDX11L1 LOC102725121 DDX11L2 DDX11L9 DDX11L10 DDX11L5 DDX11L16
#> [[["16678303"]]] ARF1
mapIds(hugene20sttranscriptcluster.db, ids, "SYMBOL", "PROBEID", multiVals = "filter")
#> 'select()' returned 1:many mapping between keys and columns
#> 16737401 16678303
#> "TRAF6"    "ARF1"
mapIds(hugene20sttranscriptcluster.db, ids, "SYMBOL", "PROBEID", multiVals = "asNA")
#> 'select()' returned 1:many mapping between keys and columns
#> 16737401 16657436 16678303
#> "TRAF6"      NA     "ARF1"
```

3.4.18 ChipDb/OrgDb questions

Using either the hugene20sttranscriptcluster.db or org.Hs.eg.db package,

- What gene symbol corresponds to Entrez Gene ID 1000?
- What is the Ensembl Gene ID for PPARG?
- What is the UniProt ID for GAPDH?
- How many of the probesets from the ExpressionSet (eset) we loaded map to a single gene? How many don't map to a gene at all?

3.4.19 TxDb packages

TxDb packages contain positional information; the contents can be inferred by the package name

TxDb.Species.Source.Build.Table

- TxDb.Hsapiens.UCSC.hg19.knownGene
 - *Homo sapiens*
 - UCSC genome browser
 - hg19 (their version of GRCh37)
 - knownGene table

TxDb.Dmelanogaster.UCSC.dm3.ensGene TxDb.Athaliana.BioMart.plantsmart22

3.4.20 EnsDb packages

EnsDb packages are similar to TxDb packages, but based on Ensembl mappings

EnsDb.Hsapiens.v79 EnsDb.Mmusculus.v79 EnsDb.Rnorvegicus.v79

3.4.21 Transcript packages

As with ChipDb and OrgDb packages, `select` and `mapIds` can be used to make queries

```
select(TxDb.Hsapiens.UCSC.hg19.knownGene, c("1", "10"),
       c("TXNAME", "TXCHROM", "TXSTART", "TXEND"), "GENEID")
#> 'select()' returned 1:many mapping between keys and columns
#>   GENEID      TXNAME TXCHROM TXSTART    TXEND
#> 1     1 uc002qsd.4    chr19 58858172 58864865
#> 2     1 uc002qsf.2    chr19 58859832 58874214
#> 3    10 uc003wyw.1    chr8 18248755 18258723
select(EnsDb.Hsapiens.v79, c("1", "10"),
       c("GENEID", "GENENAME", "SEQNAME", "GENESEQSTART", "GENESEQEND"), "ENTREZID")
#>   ENTREZID      GENEID GENENAME SEQNAME GENESEQSTART GENESEQEND
#> 1           1 ENSG00000121410     A1BG      19    58345178  58353499
#> 2          10 ENSG00000156006     NAT2       8    18391245  18401218
```

But this is not how one normally uses them...

3.4.22 GRanges

The normal use case for transcript packages is to extract positional information into a `GRanges` or `GRangesList` object. An example is the genomic position of all genes:

```
gns <- genes(TxDb.Hsapiens.UCSC.hg19.knownGene)
gns
#> GRanges object with 23056 ranges and 1 metadata column:
#>   seqnames      ranges strand |  gene_id
#>   <Rle>      <IRanges>  <Rle> / <character>
#> 1   chr19 58858172-58874214 - / 1
#> 10  chr8 18248755-18258723 + / 10
#> 100 chr20 43248163-43280376 - / 100
#> 1000 chr18 25530930-25757445 - / 1000
#> 10000 chr1 243651535-244006886 - / 10000
#> ...
#> 9991 chr9 114979995-115095944 - / 9991
#> 9992 chr21 35736323-35743440 + / 9992
#> 9993 chr22 19023795-19109967 - / 9993
#> 9994 chr6 90539619-90584155 + / 9994
#> 9997 chr22 50961997-50964905 - / 9997
#> -----
#> seqinfo: 93 sequences (1 circular) from hg19 genome
```

3.4.23 GRangesList

Or the genomic position of all transcripts by gene:

```
txs <- transcriptsBy(TxDb.Hsapiens.UCSC.hg19.knownGene)
txs
#> GRangesList object of length 23459:
#> $1
#> GRanges object with 2 ranges and 2 metadata columns:
```

```

#>           seqnames      ranges strand |   tx_id    tx_name
#>           <Rle>        <IRanges>  <Rle> | <integer> <character>
#> [1] chr19 58858172-58864865      - |    70455 uc002qsd.4
#> [2] chr19 58859832-58874214      - |    70456 uc002qsf.2
#>
#> $10
#> GRanges object with 1 range and 2 metadata columns:
#>           seqnames      ranges strand | tx_id    tx_name
#>           [1] chr8 18248755-18258723      + | 31944 uc003uryw.1
#>
#> $100
#> GRanges object with 1 range and 2 metadata columns:
#>           seqnames      ranges strand | tx_id    tx_name
#>           [1] chr20 43248163-43280376     - | 72132 uc002xmj.3
#>
#> ...
#> <23456 more elements>
#> -----
#> seqinfo: 93 sequences (1 circular) from hg19 genome

```

3.4.24 Other accessors

- Positional information can be extracted for transcripts, genes, coding sequences (cds), promoters and exons.
- Positional information can be extracted for most of the above, grouped by a second element. For example, our `transcriptsBy` call was all transcripts, grouped by gene.
- More detail on these *Ranges objects is beyond the scope of this workshop, but why we want them is not.

3.4.25 Why *Ranges objects

The main rationale for *Ranges objects is to allow us to easily select and subset data based on genomic position information. This is really powerful!

`GRanges` and `GRangesLists` act like `data.frames` and lists, and can be subsetted using the `[` function. As a really artificial example:

```

txs[txs %over% gns[1:2,]]
#> GRangesList object of length 3:
#> $1
#> GRanges object with 2 ranges and 2 metadata columns:
#>           seqnames      ranges strand |   tx_id    tx_name
#>           <Rle>        <IRanges>  <Rle> | <integer> <character>
#> [1] chr19 58858172-58864865      - |    70455 uc002qsd.4
#> [2] chr19 58859832-58874214      - |    70456 uc002qsf.2
#>
#> $10
#> GRanges object with 1 range and 2 metadata columns:
#>           seqnames      ranges strand | tx_id    tx_name
#>           [1] chr8 18248755-18258723      + | 31944 uc003uryw.1
#>

```

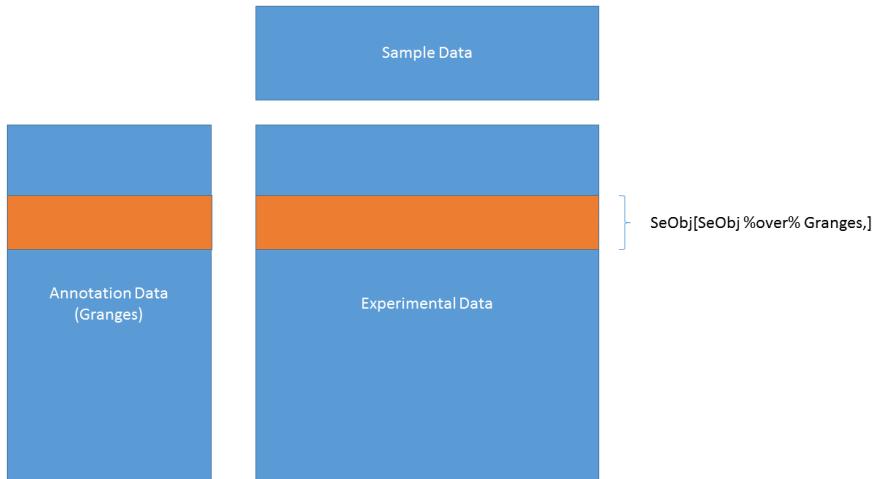
```
#> $162968
#> GRanges object with 2 ranges and 2 metadata columns:
#>   seqnames      ranges strand | tx_id    tx_name
#>   [1] chr19 58865723-58874214      - / 70457 uc002qsh.2
#>   [2] chr19 58865723-58874214      - / 70458 uc002qsi.2
#>
#> -----
#> seqinfo: 93 sequences (1 circular) from hg19 genome
```

3.4.26 *Ranges use cases

- Gene expression changes near differentially methylated CpG islands
- Closest genes to a set of interesting SNPs
- Genes near DNaseI hypersensitivity clusters
- Number of CpGs measured over Gene X by Chip Y

3.4.27 SummarizedExperiment objects

SummarizedExperiment objects are like ExpressionSets, but the row-wise annotations are GRanges, so you can subset by genomic locations:



SummarizedExperiment objects are popular objects for representing expression data and other rectangular data (feature x sample data). Incoming packages are now strongly recommended to use this class representation instead of ExpressionSet.

3.4.28 TxDb exercises

- How many transcripts does PPARG have, according to UCSC?
- Does Ensembl agree?
- How many genes are between 2858473 and 3271812 on chr2 in the hg19 genome?
 - Hint: you make a GRanges like this - GRanges("chr2", IRanges(2858473, 3271812))

3.4.29 OrganismDb packages

OrganismDb packages are meta-packages that contain an OrgDb, a TxDb, and a GO.db package and allow cross-queries between those packages.

All previous accessors work; `select`, `mapIds`, `transcripts`, etc.

```
library(Homo.sapiens)
Homo.sapiens
#> OrganismDb Object:
#> # Includes GODb Object: GO.db
#> # With data about: Gene Ontology
#> # Includes OrgDb Object: org.Hs.eg.db
#> # Gene data about: Homo sapiens
#> # Taxonomy Id: 9606
#> # Includes TxDb Object: TxDb.Hsapiens.UCSC.hg19.knownGene
#> # Transcriptome data about: Homo sapiens
#> # Based on genome: hg19
#> # The OrgDb gene id ENTREZID is mapped to the TxDb gene id GENEID .
```

3.4.30 OrganismDb packages

- Updateable - can change TxDb object
- columns and keytypes span all underlying objects
- Calls to TxDb accessors include a ‘columns’ argument

```
head(genes(Homo.sapiens, columns = c("ENTREZID", "ALIAS", "UNIPROT")), 4)
#> 'select()' returned 1:many mapping between keys and columns
#> GRanges object with 4 ranges and 3 metadata columns:
#>          seqnames      ranges strand /           ALIAS
#>          <Rle>        <IRanges> <Rle> /           <CharacterList>
#>    1     chr19 58858172-58874214      - /       A1B, ABG, GAB, ...
#>   10    chr8 18248755-18258723      + /     AAC2, NAT-2, PNAT, ...
#>  100   chr20 43248163-43280376      - /           ADA
#> 1000  chr18 25530930-25757445      - / CD325, CDHN, CDw325, ...
#>                               UNIPROT      ENTREZID
#>                               <CharacterList> <FactorList>
#>    1            P04217, V9HWD8           1
#>   10           A4Z6T7, P11245          10
#>  100          AOA0S2Z381, P00813, F5GWI4      100
#> 1000          P19022, AOA024RC42         1000
#> -----
#> seqinfo: 93 sequences (1 circular) from hg19 genome
```

3.4.31 OrganismDb exercises

- Get all the GO terms for BRCA1
- What gene does the UCSC transcript ID uc002fai.3 map to?
- How many other transcripts does that gene have?

- Get all the transcripts from the hg19 genome build, along with their Ensembl gene ID, UCSC transcript ID and gene symbol

3.4.32 Organism.dplyr package

- Combines the data from TxDb and Org.Db associated packages into local database.
- Allows functions from both *org.** and *TxDb.**
 - `keytypes()`, `select()`, ...
 - `exons()`, `promoters()`, ...
- Allows for filtering and display of combined TxDb and Org.Db information through `dplyr` functions.

```
library(Organism.dplyr)
#> Loading required package: dplyr
#>
#> Attaching package: 'dplyr'
#> The following objects are masked from 'package:stats':
#>
#>     filter, lag
#> The following objects are masked from 'package:base':
#>
#>     intersect, setdiff, setequal, union
#> Loading required package: AnnotationFilter

# src = src_organism("TxDb.Hsapiens.UCSC.hg19.knownGene")
src <- src_organism(dbpath = hg38light())
src

#> src: sqlite 3.22.0 [/usr/local/lib/R/site-library/Organism.dplyr/extdata/light.hg38.knownGene.sqlite]
#> tbls: id, id_accession, id_go, id_go_all, id_omim_pm, id_protein,
#>     id_transcript, ranges_cds, ranges_exon, ranges_gene, ranges_tx
```

3.4.33 Organism.dplyr

Get promoters from a TxDb object (we use a small version)

```
options(width = 120)
promoters(src)
#> <SQL>
#> SELECT *
#> FROM `ranges_tx`
#> GRanges object with 88 ranges and 2 metadata columns:
#>           seqnames      ranges strand |   tx_id    tx_name
#>           <Rle>        <IRanges> <Rle> | <integer> <character>
#> uc001hzz.2      chr1 243843037-243845236   - |    15880  uc001hzz.2
#> uc021plu.1      chr1 243843385-243845584   - |    15881  uc021plu.1
#> uc001iab.3      chr1 243843083-243845282   - |    15882  uc001iab.3
#> uc057qvr.1      chr1 243849929-243852128   - |    15883  uc057qvr.1
#> uc057qvt.1      chr1 243614947-243617146   - |    15884  uc057qvt.1
#> ...
#> uc064xqh.1 chrUn_GL000220v1 110025-112224   + |    197741  uc064xqh.1
#> uc064xqi.1 chrUn_GL000220v1 112151-114350   + |    197742  uc064xqi.1
#> uc064xqj.1 chrUn_GL000220v1 115428-117627   + |    197743  uc064xqj.1
```

```
#> uc064xqk.1 chrUn_GL000220v1      116197-118396      + / 197744 uc064xqk.1
#> uc033dnj.2 chrUn_GL000220v1      153997-156196      + / 197750 uc033dnj.2
#> -----
#> seqinfo: 455 sequences (1 circular) from hg38 genome
```

3.4.34 Organism.dplyr

Extract a table from the underlying database

```
tbl(src, "id")
#> # Source: table<id> [?? x 6]
#> # Database: sqlite 3.22.0 []
#>   entrez map      ensembl      symbol genename      alias
#>   <chr> <chr>      <chr>      <chr> <chr>      <chr>
#> 1 1    19q13.4 ENSG00000121410 A1BG alpha-1-B glycoprotein A1B
#> 2 1    19q13.4 ENSG00000121410 A1BG alpha-1-B glycoprotein ABG
#> 3 1    19q13.4 ENSG00000121410 A1BG alpha-1-B glycoprotein GAB
#> 4 1    19q13.4 ENSG00000121410 A1BG alpha-1-B glycoprotein HYST2477
#> 5 1    19q13.4 ENSG00000121410 A1BG alpha-1-B glycoprotein A1BG
#> 6 10   8p22    ENSG00000156006 NAT2 N-acetyltransferase 2 AAC2
#> 7 10   8p22    ENSG00000156006 NAT2 N-acetyltransferase 2 NAT-2
#> 8 10   8p22    ENSG00000156006 NAT2 N-acetyltransferase 2 PNAT
#> 9 10   8p22    ENSG00000156006 NAT2 N-acetyltransferase 2 NAT2
#> 10 100  20q13.12 ENSG00000196839 ADA adenosine deaminase ADA
#> # ... with more rows
```

3.4.35 Organism.dplyr

Make a complex query between tables in the underlying database

```
inner_join(tbl(src, "id"), tbl(src, "ranges_gene")) %>%
  filter(symbol %in% c("ADA", "NAT2")) %>%
  dplyr::select(gene_chrom, gene_start, gene_end,
                gene_strand, symbol, alias, map)
#> Joining, by = "entrez"
#> # Source: lazy query [?? x 7]
#> # Database: sqlite 3.22.0 []
#>   gene_chrom gene_start gene_end gene_strand symbol alias map
#>   <chr>      <int>     <int>      <chr>      <chr> <chr> <chr>
#> 1 chr8       18391245 18401218 +      NAT2 AAC2 8p22
#> 2 chr8       18391245 18401218 +      NAT2 NAT-2 8p22
#> 3 chr8       18391245 18401218 +      NAT2 PNAT 8p22
#> 4 chr8       18391245 18401218 +      NAT2 NAT2 8p22
#> 5 chr20      44619522 44651742 -     ADA ADA 20q13.12
```

3.4.36 Organism.dplyr exercises

- How many supported organisms are implemented in Organism.dplyr?
- Display the ensembl Id and genename description for symbol “NAT2”.

- Show all the alias for “NAT2” in the database.
 - Display Gene ontology (GO) information for gene symbol “NAT2”.

3.4.37 BSgenome packages

BSeqneme packages contain sequence information for a given species/build. There are many such packages - you can get a listing using `available.genomes`

3.4.38 BSgenome packages

We can load and inspect a BSgenome package

```

library(BSgenome.Hsapiens.UCSC.hg19)
Hsapiens
#> Human genome:
#> # organism: Homo sapiens (Human)
#> # provider: UCSC
#> # provider version: hg19
#> # release date: Feb. 2009
#> # release name: Genome Reference Consortium GRCh37
#> # 93 sequences:
#> #   chr1           chr2           chr3           chr4           chr5
#> #   chr6           chr7           chr8           chr9           chr10
#> #   chr11          chr12          chr13          chr14          chr15
#> #   chr16          chr17          chr18          chr19          chr20
#> #   chr21          chr22          chrX            chrY           chrM
#> #   ...
#> #   ...
#> #   ...
#> #   chrUn_g1000227 chrUn_g1000228 chrUn_g1000229 chrUn_g1000230 chrUn_gl
#> #   chrUn_g1000232 chrUn_g1000233 chrUn_g1000234 chrUn_g1000235 chrUn_gl
#> #   chrUn_g1000237 chrUn_g1000238 chrUn_g1000239 chrUn_g1000240 chrUn_gl
#> #   chrUn_g1000242 chrUn_g1000243 chrUn_g1000244 chrUn_g1000245 chrUn_gl
#> #   chrUn_g1000247 chrUn_g1000248 chrUn_g1000249
#> # (use 'seqnames()' to see all the sequence names, use the '$' or '[[' operator to access a given se

```

3.4.39 BSgenome packages

The main accessor is `getSeq`, and you can get data by sequence (e.g., entire chromosome or unplaced scaffold), or by passing in a GRanges object, to get just a region.

```
#>      width seq
#> [1] 85634 GCGGAGCGTGTGACGCTGCGGCCGCCGGACCTGGGGATTAA...ACTTTAAATAAATCGGAATTAAATATTAAGAGCTGACTGGAA
```

The Biostrings package contains most of the code for dealing with these `*StringSet` objects - please see the Biostrings vignettes and help pages for more information.

3.4.40 BSgenome exercises

- Get the sequences for all transcripts of the TP53 gene

3.4.41 AnnotationHub

AnnotationHub is a package that allows us to query and download many different annotation objects, without having to explicitly install them.

```
library(AnnotationHub)
hub <- AnnotationHub()
#> snapshotDate(): 2018-06-27
hub
#> AnnotationHub with 44925 records
#> # snapshotDate(): 2018-06-27
#> # $dataprovder: BroadInstitute, Ensembl, UCSC, ftp://ftp.ncbi.nlm.nih.gov/gene/DATA/, Haemcode, Inp
#> # $species: Homo sapiens, Mus musculus, Drosophila melanogaster, Bos taurus, Pan troglodytes, Rattus
#> # $rdataclass: GRanges, BigWigFile, FaFile, TwoBitFile, Rle, OrgDb, ChainFile, EnsDb, Inparanoid8Db,
#> # additional mcols(): taxonomyid, genome, description, coordinate_1_based, maintainer, rdatadateadded
#> #   preparerclass, tags, rdatapath, sourceurl, sourcetype
#> # retrieve records with, e.g., 'object[["AH2"]]' 
#>
#>           title
#> AH2    / Ailuropoda_melanoleuca.ailMeli.69.dna.toplevel.fa
#> AH3    / Ailuropoda_melanoleuca.ailMeli.69.dna_rm.toplevel.fa
#> AH4    / Ailuropoda_melanoleuca.ailMeli.69.dna_sm.toplevel.fa
#> AH5    / Ailuropoda_melanoleuca.ailMeli.69.ncrna.fa
#> AH6    / Ailuropoda_melanoleuca.ailMeli.69.pep.all.fa
#> ...
#> ...
#> AH63655 / phastCons46wayPrimates.UCSC.hg19.chrX.rds
#> AH63656 / phastCons46wayPrimates.UCSC.hg19.chrY.rds
#> AH63657 / Alternative Splicing Annotation for Homo sapiens (Human)
#> AH63658 / Allele data from the IPD IMGT/HLA database
#> AH63659 / Allele data from the IPD KIR database
```

3.4.42 Querying AnnotationHub

Finding the ‘right’ resource on AnnotationHub is like using Google - a well posed query is necessary to find what you are after. Useful queries are based on

- Data provider
- Data class
- Species
- Data source

```
names(mcols(hub))
#> [1] "title"                      "dataprovider"      "species"          "taxonomyid"
#> [6] "description"                 "coordinate_1_based" "maintainer"       "rdatadateadded"
#> [11] "tags"                       "rdataclass"        "rdatapath"        "sourceurl"
#> [16] "url"                        "genome"           "preparercl"      "sourcetype
```

3.4.43 AnnotationHub Data providers

```
unique(hub$dataprovider)
#> [1] "Ensembl"                     "UCSC"
#> [3] "RefNet"                      "Inparanoid8"
#> [5] "NHLBI"                       "ChEA"
#> [7] "Pazar"                       "NIH Pathway Interaction Database"
#> [9] "Haemcode"                    "BroadInstitute"
#> [11] "PRIDE"                      "Gencode"
#> [13] "CRIBI"                      "Genoscope"
#> [15] "MISO, VAST-TOOLS, UCSC"    "UWashington"
#> [17] "Stanford"                   "dbSNP"
#> [19] "BioMart"                    "GeneOntology"
#> [21] "KEGG"                       "URGI"
#> [23] "ftp://ftp.ncbi.nlm.nih.gov/gene/DATA/" "EMBL-EBI"
```

3.4.44 AnnotationHub Data classes

```
unique(hub$rdataclass)
#> [1] "FaFile"                      "GRanges"           "data.frame"
#> [4] "Inparanoid8Db"                "TwoBitFile"        "ChainFile"
#> [7] "SQLiteConnection"             "biopax"            "BigWigFile"
#> [10] "AAStringSet"                 "MSnSet"            "mzRpwiz"
#> [13] "mzRident"                   "list"              "TxDb"
#> [16] "Rle"                         "EnsDb"             "VcfFile"
#> [19] "igraph"                     "OrgDb"             "data.frame, DNAStringS"
```

3.4.45 AnnotationHub Species

```
head(unique(hub$species))
#> [1] "Ailuropoda melanoleuca" "Anolis carolinensis"   "Bos taurus"          "Caenorhabditis elegans"
#> [5] "Callithrix jacchus"      "Canis familiaris"    "Danio rerio"         "Drosophila melanogaster"
length(unique(hub$species))
#> [1] 1971
```

3.4.46 AnnotationHub Data sources

```
unique(hub$sourcetype)
#> [1] "FASTA"                      "UCSC track"       "GTF"               "TSV"               "Inparanoid"        "TwoBit"            "Chai
#> [8] "GRASP"                       "Zip"              "CSV"               "BioPax"            "BioPaxLevel2"     "RData"             "BED"
```

```
#> [15] "BigWig"           "tab"          "mzTab"        "mzML"        "mzid"        "GFF"
#> [22] "VCF"              "NCBI/ensembl" "NCBI/UniProt" "ensembl"
```

3.4.47 AnnotationHub query

```
qry <- query(hub, c("granges", "homo sapiens", "ensembl"))
qry
#> AnnotationHub with 56 records
#> # snapshotDate(): 2018-06-27
#> # $dataprovier: Ensembl, UCSC
#> # $species: Homo sapiens
#> # $rdataclass: GRanges
#> # additional mcols(): taxonomyid, genome, description, coordinate_1_based, maintainer, rdatadateadded
#> #   preparerclass, tags, rdatapath, sourceurl, sourcetype
#> # retrieve records with, e.g., 'object[["AH5046"]]'
```

#>

#>	title
#>	AH5046 / Ensembl Genes
#>	AH5160 / Ensembl Genes
#>	AH5311 / Ensembl Genes
#>	AH5434 / Ensembl Genes
#>	AH5435 / Ensembl EST Genes
#>
#>	AH60085 / Homo_sapiens.GRCh38.91.gtf
#>	AH61125 / Homo_sapiens.GRCh38.92.ab initio.gtf
#>	AH61126 / Homo_sapiens.GRCh38.92.chr.gtf
#>	AH61127 / Homo_sapiens.GRCh38.92.chr_patch_hapl_scaff.gtf
#>	AH61128 / Homo_sapiens.GRCh38.92.gtf

3.4.48 AnnotationHub query

```
qry$sourceurl
#> [1] "rtracklayer://hgdownload.cse.ucsc.edu/goldenpath/hg19/database/ensGene"
#> [2] "rtracklayer://hgdownload.cse.ucsc.edu/goldenpath/hg18/database/ensGene"
#> [3] "rtracklayer://hgdownload.cse.ucsc.edu/goldenpath/hg17/database/ensGene"
#> [4] "rtracklayer://hgdownload.cse.ucsc.edu/goldenpath/hg16/database/ensGene"
#> [5] "rtracklayer://hgdownload.cse.ucsc.edu/goldenpath/hg16/database/ensEstGene"
#> [6] "ftp://ftp.ensembl.org/pub/release-70/gtf/homo_sapiens/Homo_sapiens.GRCh37.70.gtf.gz"
#> [7] "ftp://ftp.ensembl.org/pub/release-69/gtf/homo_sapiens/Homo_sapiens.GRCh37.69.gtf.gz"
#> [8] "ftp://ftp.ensembl.org/pub/release-71/gtf/homo_sapiens/Homo_sapiens.GRCh37.71.gtf.gz"
#> [9] "ftp://ftp.ensembl.org/pub/release-72/gtf/homo_sapiens/Homo_sapiens.GRCh37.72.gtf.gz"
#> [10] "ftp://ftp.ensembl.org/pub/release-73/gtf/homo_sapiens/Homo_sapiens.GRCh37.73.gtf.gz"
#> [11] "ftp://ftp.ensembl.org/pub/release-74/gtf/homo_sapiens/Homo_sapiens.GRCh37.74.gtf.gz"
#> [12] "ftp://ftp.ensembl.org/pub/release-75/gtf/homo_sapiens/Homo_sapiens.GRCh37.75.gtf.gz"
#> [13] "ftp://ftp.ensembl.org/pub/release-78/gtf/homo_sapiens/Homo_sapiens.GRCh38.78.gtf.gz"
#> [14] "ftp://ftp.ensembl.org/pub/release-76/gtf/homo_sapiens/Homo_sapiens.GRCh38.76.gtf.gz"
#> [15] "ftp://ftp.ensembl.org/pub/release-79/gtf/homo_sapiens/Homo_sapiens.GRCh38.79.gtf.gz"
#> [16] "ftp://ftp.ensembl.org/pub/release-77/gtf/homo_sapiens/Homo_sapiens.GRCh38.77.gtf.gz"
#> [17] "ftp://ftp.ensembl.org/pub/release-80/gtf/homo_sapiens/Homo_sapiens.GRCh38.80.gtf.gz"
```

```
#> [18] "ftp://ftp.ensembl.org/pub/release-81/gtf/homo_sapiens/Homo_sapiens.GRCh38.81.gtf.gz"
#> [19] "ftp://ftp.ensembl.org/pub/release-82/gtf/homo_sapiens/Homo_sapiens.GRCh38.82.gtf.gz"
#> [20] "ftp://ftp.ensembl.org/pub/release-83/gtf/homo_sapiens/Homo_sapiens.GRCh38.83.gtf.gz"
#> [21] "ftp://ftp.ensembl.org/pub/release-84/gtf/homo_sapiens/Homo_sapiens.GRCh38.84.ab initio.gtf.gz"
#> [22] "ftp://ftp.ensembl.org/pub/release-84/gtf/homo_sapiens/Homo_sapiens.GRCh38.84.chr.gtf.gz"
#> [23] "ftp://ftp.ensembl.org/pub/release-84/gtf/homo_sapiens/Homo_sapiens.GRCh38.84.chr_patch_hapl_sc"
#> [24] "ftp://ftp.ensembl.org/pub/release-84/gtf/homo_sapiens/Homo_sapiens.GRCh38.84.gtf.gz"
#> [25] "ftp://ftp.ensembl.org/pub/release-85/gtf/homo_sapiens/Homo_sapiens.GRCh38.85.ab initio.gtf.gz"
#> [26] "ftp://ftp.ensembl.org/pub/release-85/gtf/homo_sapiens/Homo_sapiens.GRCh38.85.chr.gtf.gz"
#> [27] "ftp://ftp.ensembl.org/pub/release-85/gtf/homo_sapiens/Homo_sapiens.GRCh38.85.chr_patch_hapl_sc"
#> [28] "ftp://ftp.ensembl.org/pub/release-85/gtf/homo_sapiens/Homo_sapiens.GRCh38.85.gtf.gz"
#> [29] "ftp://ftp.ensembl.org/pub/release-86/gtf/homo_sapiens/Homo_sapiens.GRCh38.86.ab initio.gtf.gz"
#> [30] "ftp://ftp.ensembl.org/pub/release-86/gtf/homo_sapiens/Homo_sapiens.GRCh38.86.chr.gtf.gz"
#> [31] "ftp://ftp.ensembl.org/pub/release-86/gtf/homo_sapiens/Homo_sapiens.GRCh38.86.chr_patch_hapl_sc"
#> [32] "ftp://ftp.ensembl.org/pub/release-86/gtf/homo_sapiens/Homo_sapiens.GRCh38.86.gtf.gz"
#> [33] "ftp://ftp.ensembl.org/pub/release-87/gtf/homo_sapiens/Homo_sapiens.GRCh38.87.ab initio.gtf.gz"
#> [34] "ftp://ftp.ensembl.org/pub/release-87/gtf/homo_sapiens/Homo_sapiens.GRCh38.87.chr.gtf.gz"
#> [35] "ftp://ftp.ensembl.org/pub/release-87/gtf/homo_sapiens/Homo_sapiens.GRCh38.87.chr_patch_hapl_sc"
#> [36] "ftp://ftp.ensembl.org/pub/release-87/gtf/homo_sapiens/Homo_sapiens.GRCh38.87.gtf.gz"
#> [37] "ftp://ftp.ensembl.org/pub/release-88/gtf/homo_sapiens/Homo_sapiens.GRCh38.88.ab initio.gtf.gz"
#> [38] "ftp://ftp.ensembl.org/pub/release-88/gtf/homo_sapiens/Homo_sapiens.GRCh38.88.chr.gtf.gz"
#> [39] "ftp://ftp.ensembl.org/pub/release-88/gtf/homo_sapiens/Homo_sapiens.GRCh38.88.chr_patch_hapl_sc"
#> [40] "ftp://ftp.ensembl.org/pub/release-88/gtf/homo_sapiens/Homo_sapiens.GRCh38.88.gtf.gz"
#> [41] "ftp://ftp.ensembl.org/pub/release-89/gtf/homo_sapiens/Homo_sapiens.GRCh38.89.ab initio.gtf.gz"
#> [42] "ftp://ftp.ensembl.org/pub/release-89/gtf/homo_sapiens/Homo_sapiens.GRCh38.89.chr.gtf.gz"
#> [43] "ftp://ftp.ensembl.org/pub/release-89/gtf/homo_sapiens/Homo_sapiens.GRCh38.89.chr_patch_hapl_sc"
#> [44] "ftp://ftp.ensembl.org/pub/release-89/gtf/homo_sapiens/Homo_sapiens.GRCh38.89.gtf.gz"
#> [45] "ftp://ftp.ensembl.org/pub/release-90/gtf/homo_sapiens/Homo_sapiens.GRCh38.90.ab initio.gtf.gz"
#> [46] "ftp://ftp.ensembl.org/pub/release-90/gtf/homo_sapiens/Homo_sapiens.GRCh38.90.chr.gtf.gz"
#> [47] "ftp://ftp.ensembl.org/pub/release-90/gtf/homo_sapiens/Homo_sapiens.GRCh38.90.chr_patch_hapl_sc"
#> [48] "ftp://ftp.ensembl.org/pub/release-90/gtf/homo_sapiens/Homo_sapiens.GRCh38.90.gtf.gz"
#> [49] "ftp://ftp.ensembl.org/pub/release-91/gtf/homo_sapiens/Homo_sapiens.GRCh38.91.ab initio.gtf.gz"
#> [50] "ftp://ftp.ensembl.org/pub/release-91/gtf/homo_sapiens/Homo_sapiens.GRCh38.91.chr.gtf.gz"
#> [51] "ftp://ftp.ensembl.org/pub/release-91/gtf/homo_sapiens/Homo_sapiens.GRCh38.91.chr_patch_hapl_sc"
#> [52] "ftp://ftp.ensembl.org/pub/release-91/gtf/homo_sapiens/Homo_sapiens.GRCh38.91.gtf.gz"
#> [53] "ftp://ftp.ensembl.org/pub/release-92/gtf/homo_sapiens/Homo_sapiens.GRCh38.92.ab initio.gtf.gz"
#> [54] "ftp://ftp.ensembl.org/pub/release-92/gtf/homo_sapiens/Homo_sapiens.GRCh38.92.chr.gtf.gz"
#> [55] "ftp://ftp.ensembl.org/pub/release-92/gtf/homo_sapiens/Homo_sapiens.GRCh38.92.chr_patch_hapl_sc"
#> [56] "ftp://ftp.ensembl.org/pub/release-92/gtf/homo_sapiens/Homo_sapiens.GRCh38.92.gtf.gz"
```

3.4.49 Selecting AnnotationHub resource

```
whatIwant <- qry[["AH50377"]]
```

We can use these data as they are, or convert to a TxDb format:

```
GRCh38TxDb <- makeTxDbFromGRanges(whatIwant)
GRCh38TxDb
#> TxDb object:
#> # Db type: TxDb
#> # Supporting package: GenomicFeatures
#> # Genome: GRCh38
```

```
#> # transcript_nrow: 199184
#> # exon_nrow: 675836
#> # cds_nrow: 270225
#> # Db created by: GenomicFeatures package from Bioconductor
#> # Creation time: 2018-07-30 05:22:56 +0000 (Mon, 30 Jul 2018)
#> # GenomicFeatures version at creation time: 1.33.0
#> # RSQLite version at creation time: 2.1.1
#> # DBSCHEMAVERSION: 1.2
```

3.4.50 AnnotationHub exercises

- How many resources are on AnnotationHub for Atlantic salmon (*Salmo salar*)?
- Get the most recent Ensembl build for domesticated dog (*Canis familiaris*) and make a TxDb

3.4.51 biomaRt

The biomaRt package allows queries to an Ensembl Biomart server. We can see the choices of servers that we can use:

```
library(biomaRt)
listMarts()
#> biomart          version
#> 1 ENSEMBL_MART_ENSEMBL    Ensembl Genes 93
#> 2 ENSEMBL_MART_MOUSE      Mouse strains 93
#> 3 ENSEMBL_MART_SNPs       Ensembl Variation 93
#> 4 ENSEMBL_MART_FUNCGEN    Ensembl Regulation 93
```

3.4.52 biomaRt data sets

And we can then check for the available data sets on a particular server.

```
mart <- useMart("ENSEMBL_MART_ENSEMBL")
head(listDatasets(mart))
#> dataset           description      version
#> 1 acarolinensis_gene_ensembl Anole lizard genes (AnoCar2.0) AnoCar2.0
#> 2 amelanoleuca_gene_ensembl Panda genes (ailMeli1) ailMeli1
#> 3 amexicanus_gene_ensembl Cave fish genes (AstMex102) AstMex102
#> 4 anancymaae_gene_ensembl Ma's night monkey genes (Anan_2.0) Anan_2.0
#> 5 aplatyrhynchos_gene_ensembl Duck genes (BGI_duck_1.0) BGI_duck_1.0
#> 6 btaurus_gene_ensembl Cow genes (UMD3.1) UMD3.1
```

3.4.53 biomaRt queries

After setting up a `mart` object pointing to the server and data set that we care about, we can make queries. We first set up the `mart` object.

```
mart <- useMart("ENSEMBL_MART_ENSEMBL", "hsapiens_gene_ensembl")
```

Queries are of the form

```
getBM(attributes, filters, values, mart)
```

where

- attributes are the things we **want**
- filters are the *types of* IDs we **have**
- values are the IDs we **have**
- mart is the **mart** object we set up

3.4.54 biomaRt attributes and filters

Both attributes and filters have rather inscrutable names, but a listing can be accessed using

```
atrib <- listAttributes(mart)
filts <- listFilters(mart)
head(atrib)
#>                               name      description      page
#> 1      ensembl_gene_id      Gene stable ID feature_page
#> 2      ensembl_gene_id_version  Gene stable ID version feature_page
#> 3      ensembl_transcript_id  Transcript stable ID feature_page
#> 4      ensembl_transcript_id_version Transcript stable ID version feature_page
#> 5      ensembl_peptide_id    Protein stable ID feature_page
#> 6      ensembl_peptide_id_version Protein stable ID version feature_page
head(filt)
#>                               name      description
#> 1 chromosome_name Chromosome/scaffold name
#> 2      start          Start
#> 3      end            End
#> 4      band_start     Band Start
#> 5      band_end       Band End
#> 6      marker_start   Marker Start
```

3.4.55 biomaRt query

A simple example query

```
afyids <- c("1000_at", "1001_at", "1002_f_at", "1007_s_at")
getBM(c("affy_hg_u95av2", "hgnc_symbol"), c("affy_hg_u95av2"), afyids, mart)
#> affy_hg_u95av2 hgnc_symbol
#> 1      1000_at      MAPK3
#> 2      1007_s_at    DDR1
#> 3      1002_f_at    CYP2C19
#> 4      1002_f_at    TIE1
#> 5      1001_at      TIE1
```

3.4.56 biomaRt exercises

- Get the Ensembl gene IDs and HUGO symbol for Entrez Gene IDs 672, 5468 and 7157
- What do you get if you query for the ‘gene_exon’ for GAPDH?

Chapter 4

102: Solving common bioinformatic challenges using GenomicRanges

4.1 Instructor name and contact information

- Michael Lawrence (michafla@gene.com)

4.2 Workshop Description

We will introduce the fundamental concepts underlying the GenomicRanges package and related infrastructure. After a structured introduction, we will follow a realistic workflow, along the way exploring the central data structures, including GRanges and SummarizedExperiment, and useful operations in the ranges algebra. Topics will include data import/export, computing and summarizing data on genomic features, overlap detection, integration with reference annotations, scaling strategies, and visualization. Students can follow along, and there will be plenty of time for students to ask questions about how to apply the infrastructure to their particular use case. Michael Lawrence (Genentech).

4.2.1 Pre-requisites

- Solid understanding of R
- Basic familiarity with GRanges objects
- Basic familiarity with packages like S4Vectors, IRanges, GenomicRanges, rtracklayer, etc.

4.2.2 Workshop Participation

Describe how students will be expected to participate in the workshop.

4.2.3 *R / Bioconductor* packages used

- S4Vectors
- IRanges
- GenomicRanges
- rtracklayer

- GenomicFeatures
- SummarizedExperiment
- GenomicAlignments

4.2.4 Time outline

Activity	Time
Intro slides	30m
Workflow(s)	1hr
Remaining questions	30m

4.3 Workshop goals and objectives

4.3.1 Learning goals

- Understand how to apply the *Ranges infrastructure to real-world problems
- Gain insight into the design principles of the infrastructure and how it was meant to be used

4.3.2 Learning objectives

- Manipulate GRanges and related objects
- Use the ranges algebra to analyze genomic ranges
- Implement efficient workflows based on the *Ranges infrastructure

4.4 Introduction

4.4.1 What is the Ranges infrastructure?

The Ranges framework of packages provide data structures and algorithms for analyzing genomic data. This includes standard genomic data containers like GRanges and SummarizedExperiment, optimized data representations like Rle, and fast algorithms for computing overlaps, finding nearest neighbors, summarizing ranges and metadata, etc.

4.4.2 Why use the Ranges infrastructure?

Hundreds of Bioconductor packages operate on Ranges data structures, enabling the construction of complex workflows integrating multiple packages and data types. The API directly supports data analysis as well the construction of new genomic software. Code evolves easily from analysis script to generalized package extending the Bioconductor ecosystem.

4.4.3 Who is this workshop for?

If you still think of R as a programming language and want to write new bioinformatics algorithms and/or build interoperable software on top of formal genomic data structures, this workshop is for you. For the tidyverse analog of this workshop, see the plyranges tutorial by Stuart Lee.



Figure 4.1: An illustration of genomic ranges. GRanges represents a set genomic ranges in terms of the sequence name (typically the chromosome), start and end coordinates (as an IRanges object), and strand (either positive, negative, or unstranded). GRanges holds information about its universe of sequences (typically a genome) and an arbitrary set of metadata columns with information particular to the dataset.

4.5 Setup

To participate in this workshop you'll need to have R ≥ 3.5 and install the GenomicRanges, AnnotationHub, and airway Bioconductor 3.7 packages (Morgan (2018); Love (2018)). You can achieve this by installing the BiocManager package from CRAN, loading it then running the install command:

```
install.packages("BiocManager")
library(BiocManager)
install(c("GenomicRanges", "AnnotationHub", "airway"))
```

4.6 *GRanges*: Genomic Ranges

The central genomic data structure is the *GRanges* class, which represents a collection of genomic ranges that each have a single start and end location on the genome. It can be used to store the location of genomic features such as binding sites, read alignments and transcripts.

4.7 Constructing a *GRanges* object from `data.frame`

If we have a `data.frame` containing scores on a set of genomic ranges, we can call `makeGRangesFromDataFrame()` to promote the `data.frame` to a *GRanges*, thus adding semantics, formal constraints, and range-specific functionality. For example,

```
suppressPackageStartupMessages({
  library(BiocStyle)
  library(GenomicRanges)
})

df <- data.frame(
  seqnames = rep(c("chr1", "chr2", "chr1", "chr3"), c(1, 3, 2, 4)),
  start = c(101, 105, 125, 132, 134, 152, 153, 160, 166, 170),
  end = c(104, 120, 133, 132, 155, 154, 159, 166, 171, 190),
  strand = rep(strand(c("-", "+", "*", "+", "-")), c(1, 2, 2, 3, 2)),
  score = 1:10,
  GC = seq(1, 0, length=10),
  row.names = head(letters, 10))
gr <- makeGRangesFromDataFrame(df, keep.extra.columns=TRUE)
```

creates a *GRanges* object with 10 genomic ranges. The output of the `GRanges show()` method separates the information into a left and right hand region that are separated by | symbols. The genomic coordinates

(seqnames, ranges, and strand) are located on the left-hand side and the metadata columns (annotation) are located on the right. For this example, the metadata is comprised of "score" and "GC" information, but almost anything can be stored in the metadata portion of a *GRanges* object.

4.8 Loading a *GRanges* object from a standard file format

We often obtain data on genomic ranges from standard track formats, like BED, GFF and BigWig. The rtracklayer package parses those files directly into *GRanges* objects. The GenomicAlignments package parses BAM files into *GAlignments* objects, which behave much like *GRanges*, and it is easy to convert a *GAlignments* to a *GRanges*. We will see some examples of loading data from files later in the tutorial.

The `seqnames()`, `ranges()`, and `strand()` accessor functions extract the components of the genomic coordinates,

4.9 Basic manipulation of *GRanges* objects

```
seqnames(gr)
#> factor-Rle of length 10 with 4 runs
#>   Lengths: 1 3 2 4
#>   Values : chr1 chr2 chr1 chr3
#> Levels(3): chr1 chr2 chr3
ranges(gr)
#> IRanges object with 10 ranges and 0 metadata columns:
#>   start     end     width
#>   <integer> <integer> <integer>
#>   a       101      104      4
#>   b       105      120      16
#>   c       125      133      9
#>   d       132      132      1
#>   e       134      155      22
#>   f       152      154      3
#>   g       153      159      7
#>   h       160      166      7
#>   i       166      171      6
#>   j       170      190      21
strand(gr)
#> factor-Rle of length 10 with 5 runs
#>   Lengths: 1 2 2 3 2
#>   Values : - + * + -
#> Levels(3): + - *
```

The `granges()` function extracts genomic ranges without corresponding metadata,

```
granges(gr)
#> GRanges object with 10 ranges and 0 metadata columns:
#>   seqnames     ranges strand
#>   <Rle> <IRanges> <Rle>
#>   a       chr1    101-104    -
#>   b       chr2    105-120    +
#>   c       chr2    125-133    +
#>   d       chr2    132        *
#>   e       chr1    134-155    *
```

```
#>   f    chr1  152-154      +
#>   g    chr3  153-159      +
#>   h    chr3  160-166      +
#>   i    chr3  166-171      -
#>   j    chr3  170-190      -
#> -----
#> seqinfo: 3 sequences from an unspecified genome; no seqlengths
```

The `start()`, `end()`, `width()`, and `range` functions extract basic interval characteristics,

```
start(gr)
#> [1] 101 105 125 132 134 152 153 160 166 170
end(gr)
#> [1] 104 120 133 132 155 154 159 166 171 190
width(gr)
#> [1] 4 16 9 1 22 3 7 7 6 21
```

The `mcols()` accessor extracts the metadata as a *DataFrame*,

```
mcols(gr)
#> DataFrame with 10 rows and 2 columns
#>       score          GC
#>       <integer>     <numeric>
#> a       1             1
#> b       2 0.8888888888888889
#> c       3 0.7777777777777778
#> d       4 0.6666666666666667
#> e       5 0.5555555555555556
#> f       6 0.4444444444444444
#> g       7 0.3333333333333333
#> h       8 0.2222222222222222
#> i       9 0.1111111111111111
#> j      10            0
mcols(gr)$score
#> [1] 1 2 3 4 5 6 7 8 9 10
score(gr)
#> [1] 1 2 3 4 5 6 7 8 9 10
```

The lengths and other properties of the sequences containing the ranges can (and should) be stored in the *GRanges* object. Formal tracking of the sequence universe, typically the genome build, ensures data integrity and prevents accidental mixing of ranges from incompatible contexts. Assuming these data are of *Homo sapiens*, we could add the sequence information like this:

```
seqinfo(gr) <- Seqinfo(genome="hg38")
```

The `Seqinfo()` function automatically loads the sequence information for the specified `genome=` by querying the UCSC database.

And then retrieves as:

```
seqinfo(gr)
#> Seqinfo object with 455 sequences (1 circular) from hg38 genome:
#>   seqnames      seqlengths isCircular genome
#>   chr1        248956422    FALSE  hg38
#>   chr2        242193529    FALSE  hg38
#>   chr3        198295559    FALSE  hg38
#>   chr4        190214555    FALSE  hg38
```

```
#> chr5          181538259    FALSE hg38
#> ...           ...       ...   ...
#> chrUn_KI270753v1 62944    FALSE hg38
#> chrUn_KI270754v1 40191    FALSE hg38
#> chrUn_KI270755v1 36723    FALSE hg38
#> chrUn_KI270756v1 79590    FALSE hg38
#> chrUn_KI270757v1 71251    FALSE hg38
```

Methods for accessing the `length` and `names` have also been defined.

```
names(gr)
#> [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j"
length(gr)
#> [1] 10
```

4.10 Subsetting *GRanges* objects

GRanges objects act like vectors of ranges, with the expected vector-like subsetting operations available

```
gr[2:3]
#> GRanges object with 2 ranges and 2 metadata columns:
#>   seqnames      ranges strand |      score      GC
#>     <Rle> <IRanges> <Rle> / <integer>      <numeric>
#>   b      chr2    105-120    + /      2 0.888888888888889
#>   c      chr2    125-133    + /      3 0.777777777777778
#>   -----
#>   seqinfo: 455 sequences (1 circular) from hg38 genome
```

A second argument to the `[` subset operator specifies which metadata columns to extract from the *GRanges* object. For example,

```
gr[2:3, "GC"]
#> GRanges object with 2 ranges and 1 metadata column:
#>   seqnames      ranges strand |      GC
#>     <Rle> <IRanges> <Rle> /      <numeric>
#>   b      chr2    105-120    + /  0.888888888888889
#>   c      chr2    125-133    + /  0.777777777777778
#>   -----
#>   seqinfo: 455 sequences (1 circular) from hg38 genome
```

The `subset()` function provides an easy way to subset based on attributes of the ranges and columns in the metadata. For example,

```
subset(gr, strand == "+" & score > 5, select = score)
#> GRanges object with 3 ranges and 1 metadata column:
#>   seqnames      ranges strand |      score
#>     <Rle> <IRanges> <Rle> / <integer>
#>   f      chr1    152-154    + /      6
#>   g      chr3    153-159    + /      7
#>   h      chr3    160-166    + /      8
#>   -----
#>   seqinfo: 455 sequences (1 circular) from hg38 genome
```

Elements can also be assigned to the *GRanges* object. This example replaces the the second row of a *GRanges* object with the first row of `gr`.

```

grMod <- gr
grMod[2] <- gr[1]
head(grMod, n=3)
#> GRanges object with 3 ranges and 2 metadata columns:
#>   seqnames      ranges strand |  score          GC
#>   <Rle> <IRanges> <Rle> / <integer>    <numeric>
#>   a     chr1    101-104      - /        1             1
#>   b     chr1    101-104      - /        1             1
#>   c     chr2    125-133      + /        3 0.7777777777777778
#> -----
#> seqinfo: 455 sequences (1 circular) from hg38 genome

```

There are methods to repeat, reverse, or select specific portions of *GRanges* objects.

```

rep(gr[2], times = 3)
#> GRanges object with 3 ranges and 2 metadata columns:
#>   seqnames      ranges strand |  score          GC
#>   <Rle> <IRanges> <Rle> / <integer>    <numeric>
#>   b     chr2    105-120      + /        2 0.8888888888888889
#>   b     chr2    105-120      + /        2 0.8888888888888889
#>   b     chr2    105-120      + /        2 0.8888888888888889
#> -----
#> seqinfo: 455 sequences (1 circular) from hg38 genome
rev(gr)
#> GRanges object with 10 ranges and 2 metadata columns:
#>   seqnames      ranges strand |  score          GC
#>   <Rle> <IRanges> <Rle> / <integer>    <numeric>
#>   j     chr3    170-190      - /        10            0
#>   i     chr3    166-171      - /        9 0.111111111111111
#>   h     chr3    160-166      + /        8 0.2222222222222222
#>   g     chr3    153-159      + /        7 0.333333333333333
#>   f     chr1    152-154      + /        6 0.444444444444444
#>   e     chr1    134-155      * /        5 0.555555555555556
#>   d     chr2    132          * /        4 0.666666666666667
#>   c     chr2    125-133      + /        3 0.777777777777778
#>   b     chr2    105-120      + /        2 0.888888888888889
#>   a     chr1    101-104      - /        1             1
#> -----
#> seqinfo: 455 sequences (1 circular) from hg38 genome
head(gr, n=2)
#> GRanges object with 2 ranges and 2 metadata columns:
#>   seqnames      ranges strand |  score          GC
#>   <Rle> <IRanges> <Rle> / <integer>    <numeric>
#>   a     chr1    101-104      - /        1             1
#>   b     chr2    105-120      + /        2 0.888888888888889
#> -----
#> seqinfo: 455 sequences (1 circular) from hg38 genome
tail(gr, n=2)
#> GRanges object with 2 ranges and 2 metadata columns:
#>   seqnames      ranges strand |  score          GC
#>   <Rle> <IRanges> <Rle> / <integer>    <numeric>
#>   i     chr3    166-171      - /        9 0.111111111111111
#>   j     chr3    170-190      - /        10            0
#> -----

```

```
#> seqinfo: 455 sequences (1 circular) from hg38 genome
window(gr, start=2, end=4)
#> GRanges object with 3 ranges and 2 metadata columns:
#>   seqnames      ranges strand |  score          GC
#>     <Rle> <IRanges>  <Rle> / <integer>    <numeric>
#>   b     chr2    105-120      + /           2 0.8888888888888889
#>   c     chr2    125-133      + /           3 0.7777777777777778
#>   d     chr2        132      * /           4 0.6666666666666667
#> -----
#> seqinfo: 455 sequences (1 circular) from hg38 genome
gr[IRanges(start=c(2,7), end=c(3,9))]
#> GRanges object with 5 ranges and 2 metadata columns:
#>   seqnames      ranges strand |  score          GC
#>     <Rle> <IRanges>  <Rle> / <integer>    <numeric>
#>   b     chr2    105-120      + /           2 0.8888888888888889
#>   c     chr2    125-133      + /           3 0.7777777777777778
#>   g     chr3    153-159      + /           7 0.3333333333333333
#>   h     chr3    160-166      + /           8 0.2222222222222222
#>   i     chr3    166-171      - /           9 0.1111111111111111
#> -----
#> seqinfo: 455 sequences (1 circular) from hg38 genome
```

4.11 Splitting and combining *GRanges* objects

The `split()` function divides a *GRanges* into groups, returning a *GRangesList*, a class that we will describe and demonstrate later.

```
sp <- split(gr, rep(1:2, each=5))
sp
#> GRangesList object of length 2:
#> $1
#> GRanges object with 5 ranges and 2 metadata columns:
#>   seqnames      ranges strand |  score          GC
#>     <Rle> <IRanges>  <Rle> / <integer>    <numeric>
#>   a     chr1    101-104      - /           1 1
#>   b     chr2    105-120      + /           2 0.8888888888888889
#>   c     chr2    125-133      + /           3 0.7777777777777778
#>   d     chr2        132      * /           4 0.6666666666666667
#>   e     chr1    134-155      * /           5 0.5555555555555556
#>
#> $2
#> GRanges object with 5 ranges and 2 metadata columns:
#>   seqnames      ranges strand |  score          GC
#>     <Rle> <IRanges>  <Rle> / <integer>    <numeric>
#>   f     chr1    152-154      + /           6 0.4444444444444444
#>   g     chr3    153-159      + /           7 0.3333333333333333
#>   h     chr3    160-166      + /           8 0.2222222222222222
#>   i     chr3    166-171      - /           9 0.1111111111111111
#>   j     chr3    170-190      - /           10 0
#> -----
#> seqinfo: 455 sequences (1 circular) from hg38 genome
```

We can split the ranges by metadata columns, like strand,

```
split(gr, ~ strand)
#> GRangesList object of length 3:
#> $+
#>   GRanges object with 5 ranges and 2 metadata columns:
#>     seqnames      ranges strand | score          GC
#>       <Rle> <IRanges> <Rle> / <integer>      <numeric>
#>     b      chr2    105-120    + /      2 0.8888888888888889
#>     c      chr2    125-133    + /      3 0.7777777777777778
#>     f      chr1    152-154    + /      6 0.4444444444444444
#>     g      chr3    153-159    + /      7 0.3333333333333333
#>     h      chr3    160-166    + /      8 0.2222222222222222
#>
#> $-
#>   GRanges object with 3 ranges and 2 metadata columns:
#>     seqnames      ranges strand | score          GC
#>       <Rle> <IRanges> <Rle> / <integer>      <numeric>
#>     a      chr1    101-104    - /      1 1
#>     i      chr3    166-171    - /      9 0.1111111111111111
#>     j      chr3    170-190    - /      10 0
#>
#> $*
#>   GRanges object with 2 ranges and 2 metadata columns:
#>     seqnames      ranges strand | score          GC
#>       <Rle> <IRanges> <Rle> / <integer>      <numeric>
#>     d      chr2      132    * /      4 0.6666666666666667
#>     e      chr1    134-155    * /      5 0.5555555555555556
#>
#> -----
#> seqinfo: 455 sequences (1 circular) from hg38 genome
```

The `c()` and `append()` functions combine two (or more in the case of `c()`) `GRanges` objects.

```
c(sp[[1]], sp[[2]])
#> GRanges object with 10 ranges and 2 metadata columns:
#>   seqnames      ranges strand | score          GC
#>     <Rle> <IRanges> <Rle> / <integer>      <numeric>
#>     a      chr1    101-104    - /      1 1
#>     b      chr2    105-120    + /      2 0.8888888888888889
#>     c      chr2    125-133    + /      3 0.7777777777777778
#>     d      chr2      132    * /      4 0.6666666666666667
#>     e      chr1    134-155    * /      5 0.5555555555555556
#>     f      chr1    152-154    + /      6 0.4444444444444444
#>     g      chr3    153-159    + /      7 0.3333333333333333
#>     h      chr3    160-166    + /      8 0.2222222222222222
#>     i      chr3    166-171    - /      9 0.1111111111111111
#>     j      chr3    170-190    - /      10 0
#>
#> -----
#> seqinfo: 455 sequences (1 circular) from hg38 genome
```

The `stack()` function stacks the elements of a `GRangesList` into a single `GRanges` and adds a column indicating the origin of each element,

```
stack(sp, index.var="group")
#> GRanges object with 10 ranges and 3 metadata columns:
#>   seqnames      ranges strand | group      score          GC
#>     <Rle> <IRanges> <Rle> / <Rle> <integer>      <numeric>
```

```
#>   a    chr1  101-104      - /     1      1          1
#>   b    chr2  105-120      + /     1      2 0.8888888888888889
#>   c    chr2  125-133      + /     1      3 0.7777777777777778
#>   d    chr2       132      * /     1      4 0.6666666666666667
#>   e    chr1  134-155      * /     1      5 0.5555555555555556
#>   f    chr1  152-154      + /     2      6 0.4444444444444444
#>   g    chr3  153-159      + /     2      7 0.3333333333333333
#>   h    chr3  160-166      + /     2      8 0.2222222222222222
#>   i    chr3  166-171      - /     2      9 0.1111111111111111
#>   j    chr3  170-190      - /     2      10          0
#> -----
#> seqinfo: 455 sequences (1 circular) from hg38 genome
```

4.12 Aggregating *GRanges* objects

Like other tabular data structures, we can aggregate *GRanges* objects, for example,

```
aggregate(gr, score ~ strand, mean)
#> DataFrame with 3 rows and 2 columns
#>   strand           score
#>   <factor>     <numeric>
#> 1      +         5.2
#> 2      - 6.66666666666667
#> 3      *         4.5
```

The `aggregate()` function also supports a syntax similar to `summarize()` from `dplyr`,

```
aggregate(gr, ~ strand, n_score = lengths(score), mean_score = mean(score))
#> DataFrame with 3 rows and 4 columns
#>   grouping strand n_score      mean_score
#>   <ManyToOneGrouping> <factor> <integer>     <numeric>
#> 1      2,3,6,...    +      5          5.2
#> 2      1,9,10       -      3 6.66666666666667
#> 3      4,5          *      2          4.5
```

Note that we need to call `lengths(score)` instead of `length(score)` because `score` is actually a list-like object in the aggregation expression.

4.13 Basic interval operations for *GRanges* objects

There are many functions for manipulating *GRanges* objects. The functions can be classified as *intra-range functions*, *inter-range functions*, and *between-range functions*.

Intra-range functions operate on each element of a *GRanges* object independent of the other ranges in the object. For example, the `flank` function can be used to recover regions flanking the set of ranges represented by the *GRanges* object. So to get a *GRanges* object containing the ranges that include the 10 bases upstream according to the direction of “transcription” (indicated by the strand):

```
r g <- gr[1:3] g <- append(g, gr[10]) flank(g, 10) #> GRanges object with 4 ranges
and 2 metadata columns: #>           seqnames      ranges strand |      score          GC
#>           <Rle> <IRanges> <Rle> | <integer>     <numeric> #>   a      chr1  105-114
#>           1                  1  #>   b      chr2  95-104      + |      2 0.8888888888888889
```

```
#>   c      chr2    115-124      + |      3 0.7777777777777778 #>   j      chr3    191-200
- |      10          0  #>   ----- #>   seqinfo: 455 sequences (1 circular)
from hg38 genome
```

And to include the downstream bases:

```
flank(g, 10, start=FALSE)
#> GRanges object with 4 ranges and 2 metadata columns:
#>   seqnames ranges strand | score      GC
#>     <Rle> <IRanges> <Rle> / <integer>      <numeric>
#>   a     chr1    91-100    - /      1           1
#>   b     chr2    121-130    + /      2 0.8888888888888889
#>   c     chr2    134-143    + /      3 0.7777777777777778
#>   j     chr3    160-169    - /      10          0
#>   -----
#>   seqinfo: 455 sequences (1 circular) from hg38 genome
```

A common use case for `flank()` is generating promoter regions based on the transcript ranges. There is a convenience function that by default generates a region starting 2000bp upstream and 200bp downstream of the TSS,

```
promoters(g)
#> Warning in valid.GenomicRanges.seqinfo(x, suggest.trim = TRUE): GRanges object contains 4 out-of-bound
#>   chr1, chr2, and chr3. Note that ranges located on a sequence whose
#>   length is unknown (NA) or on a circular sequence are not
#>   considered out-of-bound (use seqlengths() and isCircular() to get
#>   the lengths and circularity flags of the underlying sequences).
#>   You can use trim() to trim these ranges. See
#>   ?`trim,GenomicRanges-method` for more information.
#> GRanges object with 4 ranges and 2 metadata columns:
#>   seqnames ranges strand | score      GC
#>     <Rle> <IRanges> <Rle> / <integer>      <numeric>
#>   a     chr1    -95-2104   - /      1           1
#>   b     chr2    -1895-304   + /      2 0.8888888888888889
#>   c     chr2    -1875-324   + /      3 0.7777777777777778
#>   j     chr3    -9-2190    - /      10          0
#>   -----
#>   seqinfo: 455 sequences (1 circular) from hg38 genome
```

To ignore strand/transcription and assume the orientation of left to right use `unstrand()`,

```
flank(unstrand(g), 10)
#> GRanges object with 4 ranges and 2 metadata columns:
#>   seqnames ranges strand | score      GC
#>     <Rle> <IRanges> <Rle> / <integer>      <numeric>
#>   a     chr1    91-100    * /      1           1
#>   b     chr2    95-104    * /      2 0.8888888888888889
#>   c     chr2    115-124    * /      3 0.7777777777777778
#>   j     chr3    160-169    * /      10          0
#>   -----
#>   seqinfo: 455 sequences (1 circular) from hg38 genome
```

Other examples of intra-range functions include `resize()` and `shift()`. The `shift()` function will move the ranges by a specific number of base pairs, and the `resize()` function will set a specific width, by default fixing the “transcription” start (or just the start when strand is “*”). The `fix=` argument controls whether the “start”, “end” or “center” is held constant.

```

shift(g, 5)
#> GRanges object with 4 ranges and 2 metadata columns:
#>   seqnames      ranges strand |  score          GC
#>     <Rle> <IRanges> <Rle> / <integer>     <numeric>
#>   a    chr1  106-109      - /        1             1
#>   b    chr2  110-125      + /    2 0.888888888888889
#>   c    chr2  130-138      + /    3 0.777777777777778
#>   j    chr3  175-195      - /       10            0
#> -----
#> seqinfo: 455 sequences (1 circular) from hg38 genome
resize(g, 30)
#> GRanges object with 4 ranges and 2 metadata columns:
#>   seqnames      ranges strand |  score          GC
#>     <Rle> <IRanges> <Rle> / <integer>     <numeric>
#>   a    chr1  75-104      - /        1             1
#>   b    chr2  105-134      + /    2 0.888888888888889
#>   c    chr2  125-154      + /    3 0.777777777777778
#>   j    chr3  161-190      - /       10            0
#> -----
#> seqinfo: 455 sequences (1 circular) from hg38 genome

```

The *GenomicRanges* help page `?"intra-range-methods"` summarizes these methods.

Inter-range functions involve comparisons between ranges in a single *GRanges* object and typically aggregate ranges. For instance, the `reduce()` function will merge overlapping and adjacent ranges to produce a minimal set of ranges representing the regions covered by the original set.

```

reduce(gr)
#> GRanges object with 8 ranges and 0 metadata columns:
#>   seqnames      ranges strand
#>     <Rle> <IRanges> <Rle>
#>   [1] chr1  152-154      +
#>   [2] chr1  101-104      -
#>   [3] chr1  134-155      *
#>   [4] chr2  105-120      +
#>   [5] chr2  125-133      +
#>   [6] chr2      132      *
#>   [7] chr3  153-166      +
#>   [8] chr3  166-190      -
#> -----
#> seqinfo: 455 sequences (1 circular) from hg38 genome
reduce(gr, ignore.strand=TRUE)
#> GRanges object with 5 ranges and 0 metadata columns:
#>   seqnames      ranges strand
#>     <Rle> <IRanges> <Rle>
#>   [1] chr1  101-104      *
#>   [2] chr1  134-155      *
#>   [3] chr2  105-120      *
#>   [4] chr2  125-133      *
#>   [5] chr3  153-190      *
#> -----
#> seqinfo: 455 sequences (1 circular) from hg38 genome

```

Rarely, it is useful to complement the (reduced) ranges. Note that the universe is taken as the entire sequence span in all three strands (+, -, *), which is often surprising when working with unstranded ranges.

```

gaps(g)
#> GRanges object with 1369 ranges and 0 metadata columns:
#>           seqnames      ranges strand
#>           <Rle>      <IRanges>  <Rle>
#> [1]     chr1    1-248956422    +
#> [2]     chr1      1-100      -
#> [3]     chr1  105-248956422    -
#> [4]     chr1    1-248956422    *
#> [5]     chr2      1-104      +
#> ...
#> ...
#> [1365] chrUn_KI270756v1    1-79590      -
#> [1366] chrUn_KI270756v1    1-79590      *
#> [1367] chrUn_KI270757v1    1-71251      +
#> [1368] chrUn_KI270757v1    1-71251      -
#> [1369] chrUn_KI270757v1    1-71251      *
#> -----
#> seqinfo: 455 sequences (1 circular) from hg38 genome

```

The `disjoin` function breaks up the ranges so that they do not overlap but still cover the same regions:

```

disjoin(g)
#> GRanges object with 4 ranges and 0 metadata columns:
#>           seqnames      ranges strand
#>           <Rle>      <IRanges>  <Rle>
#> [1]     chr1    101-104      -
#> [2]     chr2    105-120      +
#> [3]     chr2    125-133      +
#> [4]     chr3    170-190      -
#> -----
#> seqinfo: 455 sequences (1 circular) from hg38 genome

```

The `coverage` function counts how many ranges overlap each position in the sequence universe of a `GRanges` object.

```

cov <- coverage(g)
cov[1:3]
#> RleList of length 3
#> $chr1
#> integer-Rle of length 248956422 with 3 runs
#>   Lengths:      100          4 248956318
#>   Values :      0            1            0
#>
#> $chr2
#> integer-Rle of length 242193529 with 5 runs
#>   Lengths:      104          16           4           9 242193396
#>   Values :      0            1            0            1            0
#>
#> $chr3
#> integer-Rle of length 198295559 with 3 runs
#>   Lengths:      169          21 198295369
#>   Values :      0            1            0

```

The coverage is stored compactly as an `RleList`, with one `Rle` vector per sequence. We can convert it to a `GRanges`,

```

cov_gr <- GRanges(cov)
cov_gr
#> GRanges object with 463 ranges and 1 metadata column:
#>           seqnames      ranges strand |  score
#>           <Rle>      <IRanges>  <Rle> / <integer>
#> [1]     chr1        1-100    * /      0
#> [2]     chr1       101-104   * /      1
#> [3]     chr1  105-248956422   * /      0
#> [4]     chr2        1-104    * /      0
#> [5]     chr2       105-120   * /      1
#> ...
#> ...
#> [459] chrUn_KI270753v1    1-62944   * /      0
#> [460] chrUn_KI270754v1    1-40191   * /      0
#> [461] chrUn_KI270755v1    1-36723   * /      0
#> [462] chrUn_KI270756v1    1-79590   * /      0
#> [463] chrUn_KI270757v1    1-71251   * /      0
#> -----
#> seqinfo: 455 sequences from an unspecified genome

```

and even convert the *GRanges* form back to an *RleList* by computing a weighted coverage,

```
cov <- coverage(cov_gr, weight="score")
```

The *GRanges* derivative *GPos*, a compact representation of width 1 ranges, is useful for representing coverage, although it cannot yet represent the coverage for the entire human genome (or any genome with over ~ 2 billion bp).

```

GPos(cov[1:3])
#> GPos object with 689445510 positions and 0 metadata columns:
#>           seqnames      pos strand
#>           <Rle> <integer>  <Rle>
#> [1]     chr1        1    *
#> [2]     chr1        2    *
#> [3]     chr1        3    *
#> [4]     chr1        4    *
#> [5]     chr1        5    *
#> ...
#> ...
#> [689445506]  chr3 198295555   *
#> [689445507]  chr3 198295556   *
#> [689445508]  chr3 198295557   *
#> [689445509]  chr3 198295558   *
#> [689445510]  chr3 198295559   *
#> -----
#> seqinfo: 3 sequences from an unspecified genome

```

These inter-range functions all generate entirely new sets of ranges. The return value is left unannotated, since there is no obvious way to carry the metadata across the operation. The user is left to map the metadata to the new ranges. Functions like `reduce()` and `disjoin()` facilitate this by optionally including in the returned metadata a one-to-many reverse mapping from the aggregate ranges to input ranges. For example, to average the score over a reduction,

```

rg <- reduce(gr, with.revmap=TRUE)
rg$score <- mean(extractList(gr$score, rg$revmap))

```

See the *GenomicRanges* help page `?inter-range-methods` for additional help.

4.14 Interval set operations for *GRanges* objects

Between-range functions calculate relationships between different *GRanges* objects. Of central importance are `findOverlaps` and related operations; these are discussed below. Additional operations treat *GRanges* as mathematical sets of coordinates; `union(g, g2)` is the union of the coordinates in `g` and `g2`. Here are examples for calculating the `union`, the `intersect` and the asymmetric difference (using `setdiff`).

```
g2 <- head(gr, n=2)
union(g, g2)
#> GRanges object with 4 ranges and 0 metadata columns:
#>   seqnames      ranges strand
#>   <Rle> <IRanges> <Rle>
#>   [1] chr1    101-104      -
#>   [2] chr2    105-120      +
#>   [3] chr2    125-133      +
#>   [4] chr3    170-190      -
#>   -----
#>   seqinfo: 455 sequences (1 circular) from hg38 genome
intersect(g, g2)
#> GRanges object with 2 ranges and 0 metadata columns:
#>   seqnames      ranges strand
#>   <Rle> <IRanges> <Rle>
#>   [1] chr1    101-104      -
#>   [2] chr2    105-120      +
#>   -----
#>   seqinfo: 455 sequences (1 circular) from hg38 genome
setdiff(g, g2)
#> GRanges object with 2 ranges and 0 metadata columns:
#>   seqnames      ranges strand
#>   <Rle> <IRanges> <Rle>
#>   [1] chr2    125-133      +
#>   [2] chr3    170-190      -
#>   -----
#>   seqinfo: 455 sequences (1 circular) from hg38 genome
```

Related functions are available when the structure of the *GRanges* objects are ‘parallel’ to one another, i.e., element 1 of object 1 is related to element 1 of object 2, and so on. These operations all begin with a `p`, which is short for parallel. The functions then perform element-wise, e.g., the union of element 1 of object 1 with element 1 of object 2, etc. A requirement for these operations is that the number of elements in each *GRanges* object is the same, and that both of the objects have the same seqnames and strand assignments throughout.

```
g3 <- g[1:2]
ranges(g3[1]) <- IRanges(start=105, end=112)
punion(g2, g3)
#> GRanges object with 2 ranges and 0 metadata columns:
#>   seqnames      ranges strand
#>   <Rle> <IRanges> <Rle>
#>   a     chr1    101-112      -
#>   b     chr2    105-120      +
#>   -----
#>   seqinfo: 455 sequences (1 circular) from hg38 genome
pintersect(g2, g3)
#> GRanges object with 2 ranges and 3 metadata columns:
#>   seqnames      ranges strand /      score          GC      hit
#>   <Rle> <IRanges> <Rle> / <integer> <numeric> <logical>
```

```
#>   a      chr1  105-104      - /      1      1      TRUE
#>   b      chr2  105-120      + /      2 0.888888888888889      TRUE
#>   -----
#>   seqinfo: 455 sequences (1 circular) from hg38 genome
psetdiff(g2, g3)
#> GRanges object with 2 ranges and 0 metadata columns:
#>   seqnames      ranges strand
#>   <Rle> <IRanges> <Rle>
#>   a      chr1  101-104      -
#>   b      chr2  105-104      +
#>   -----
#>   seqinfo: 455 sequences (1 circular) from hg38 genome
```

For more information on the `GRanges` classes be sure to consult the manual page.

`?GRanges`

A relatively comprehensive list of available functions is discovered with

`methods(class="GRanges")`

4.15 Finding overlaps between `GRanges` objects

Interval overlapping is the process of comparing the ranges in two objects to determine if and when they overlap. As such, it is perhaps the most common operation performed on `GRanges` objects. To this end, the `GenomicRanges` package provides a family of interval overlap functions. The most general of these functions is `findOverlaps()`, which takes a query and a subject as inputs and returns a `Hits` object containing the index pairings for the overlapping elements.

Let us assume that we have three random data.frame objects, each with annoyingly differing ways of naming the columns defining the ranges,

```
set.seed(66+105+111+99+49+56)

pos <- sample(1:200, size = 30L)
size <- 10L
end <- size + pos - 1L
chrom <- sample(paste0("chr", 1:3), size = 30L, replace = TRUE)
query_df <- data.frame(chrom = chrom,
                       start = pos,
                       end = end)
query_dfs <- split(query_df, 1:3)
q1 <- rename(query_dfs[[1L]], start = "pos")
q2 <- rename(query_dfs[[2L]], chrom = "ch", start = "st")
q3 <- rename(query_dfs[[3L]], end = "last")
```

The `makeGRangesFromDataFrame()` function can guess some of these, but not all of them, so we help it out,

```
q1 <- makeGRangesFromDataFrame(q1, start.field = "pos")
q2 <- makeGRangesFromDataFrame(q2, seqnames.field = "ch",
                               start.field = "st")
q3 <- makeGRangesFromDataFrame(q3, end.field = "last")
query <- mstack(q1, q2, q3, .index.var="replicate")
sort(query, by = ~ start)
#> GRanges object with 30 ranges and 1 metadata column:
```

```
#>      seqnames    ranges strand / replicate
#>      <Rle> <IRanges>  <Rle> /     <Rle>
#>  25   chr1     11-20    * /      1
#>  22   chr1     16-25    * /      1
#>  2    chr2     21-30    * /      2
#>  30   chr3     50-59    * /      3
#>  6    chr2     51-60    * /      3
#> ...
#> ...
#> 26   chr3     160-169   * /      2
#> 13   chr1     169-178   * /      1
#> 29   chr2     183-192   * /      2
#> 27   chr3     190-199   * /      3
#> 24   chr3     197-206   * /      3
#> -----
#> seqinfo: 3 sequences from an unspecified genome; no seqlengths
```

Above, we use the convenient `mstack()` function, which stacks its arguments, populating the `.index.var`= column with the origin of each range (using the argument names or positions).

Perhaps the simplest overlap-based operation is `subsetByOverlaps()`, which extracts the elements in the query (the first argument) that overlap at least one element in the subject (the second).

```
subject <- gr
subsetByOverlaps(query, subject, ignore.strand=TRUE)
#> GRanges object with 9 ranges and 1 metadata column:
#>      seqnames    ranges strand / replicate
#>      <Rle> <IRanges>  <Rle> /     <Rle>
#>  10   chr2     120-129   * /      1
#>  28   chr1     151-160   * /      1
#>  5    chr2     128-137   * /      2
#>  14   chr1     149-158   * /      2
#>  23   chr3     153-162   * /      2
#>  26   chr3     160-169   * /      2
#>  9    chr1     92-101    * /      3
#>  21   chr1     148-157   * /      3
#>  27   chr3     190-199   * /      3
#> -----
#> seqinfo: 3 sequences from an unspecified genome; no seqlengths
```

In every call to an overlap operation, it is necessary to specify `ignore.strand=TRUE`, except in rare cases when we do not want ranges on opposite strands to be considered overlapping.

To generally compute on the overlaps, we call `findOverlaps()` to return a `Hits` object, which is essentially a bipartite graph matching query ranges to overlapping subject ranges.

```
hits <- findOverlaps(query, subject, ignore.strand=TRUE)
```

We typically use the hits to perform one of two operations: join and aggregate. For example, we could inner join the scores from the subject using the query and subject indexes,

```
joined <- query[queryHits(hits)]
joined$score <- subject$score[subjectHits(hits)]
```

The above carries over a single metadata column from the subject. Similar code would carry over other columns and even the ranges themselves.

Sometimes, we want to merge the matched query and subject ranges, typically by finding their intersection,

```
ranges(joined) <- ranges(pintersect(joined, subject[subjectHits(hits)]))
```

The typical aggregation is counting the number of hits overlapping a query. In general, aggregation starts by grouping the subject hits by query hits, which we express as a coercion to a *List*,

```
hitsByQuery <- as(hits, "List")
```

The result is an *IntegerList*, a type of *AtomicList*. *AtomicList* objects have many methods for efficient aggregation. In this case, we just call `lengths()` to get the count:

```
counts <- lengths(hitsByQuery)
```

Since this a common operation, there are shortcuts,

```
counts <- countQueryHits(hits)
```

or even shorter and more efficient,

```
counts <- countOverlaps(query, subject, ignore.strand=TRUE)
unname(counts)
#> [1] 0 0 0 2 0 0 0 0 0 2 0 2 0 0 2 0 0 2 2 0 0 0 1 0 0 0 2 0 1 0
```

Often, we want to combine joins and aggregations. For example, we may want to annotate each query with the maximum score among the subject hits,

```
query$maxScore <- max(extractList(subject$score, hitsByQuery))
subset(query, maxScore > 0)
#> GRanges object with 9 ranges and 2 metadata columns:
#>   seqnames      ranges strand | replicate maxScore
#>   <Rle> <IRanges> <Rle> /     <Rle> <integer>
#> 10  chr2    120-129    * /       1        3
#> 28  chr1    151-160    * /       1        6
#>  5  chr2    128-137    * /       2        4
#> 14  chr1    149-158    * /       2        6
#> 23  chr3    153-162    * /       2        8
#> 26  chr3    160-169    * /       2        9
#>  9  chr1    92-101    * /       3        1
#> 21  chr1    148-157    * /       3        6
#> 27  chr3    190-199    * /       3       10
#> -----
#> seqinfo: 3 sequences from an unspecified genome; no seqlengths
```

In rare cases, we can more or less arbitrarily select one of the subject hits. The `select=` argument to `findOverlaps()` automatically selects an “arbitrary”, “first” (in subject order) or “last” subject range,

```
hits <- findOverlaps(query, subject, select="first", ignore.strand=TRUE)
hits <- findOverlaps(query, subject, select="arbitrary", ignore.strand=TRUE)
hits
#> [1] NA NA NA  2 NA NA NA NA NA  5 NA  3 NA NA  5 NA NA  7  8 NA NA NA  1
#> [24] NA NA NA  5 NA 10 NA
```

4.16 Exercises

- Find the average intensity of the X and Y measurements for each each replicate over all positions in the query object

2. Add a new column to the intensities object that is the distance from each position to its closest gene (hint `IRanges::distance()`)
3. Find flanking regions downstream of the genes in `gr` that have width of 8bp
4. Are any of the intensities positions within the flanking region?

4.17 Example: exploring BigWig files from AnnotationHub

In the workflow of ChIP-seq data analysis, we are often interested in finding peaks from islands of coverage over a chromosome. Here we will use plyranges to explore ChIP-seq data from the Human Epigenome Roadmap project Roadmap Epigenomics Consortium et al. (2015).

4.17.1 Extracting data from AnnotationHub

This data is available on Bioconductor's AnnotationHub. First we construct an AnnotationHub, and then `query()` for all bigWigFiles related to the project that correspond to the following conditions:

1. are from methylation marks (H3K4ME in the title)
2. correspond to primary T CD8+ memory cells from peripheral blood
3. correspond to unimputed log10 P-values

First we construct a hub that contains all references to the EpigenomeRoadMap data and extract the metadata as a data.frame:

```
library(AnnotationHub)
ah <- AnnotationHub()
#> snapshotDate(): 2018-06-27
roadmap_hub <- query(ah, "EpigenomeRoadMap")
metadata <- query(ah, "Metadata")[[1L]]
#> downloading 0 resources
#> loading from cache
#>      '/home/mramos//.AnnotationHub/47270'
head(metadata)
#>   EID     GROUP    COLOR      MNEMONIC
#> 1 E001     ESC #924965  ESC.I3
#> 2 E002     ESC #924965  ESC.WA7
#> 3 E003     ESC #924965  ESC.H1
#> 4 E004 ES-deriv #4178AE ESDR.H1.BMP4.MESO
#> 5 E005 ES-deriv #4178AE ESDR.H1.BMP4.TROP
#> 6 E006 ES-deriv #4178AE      ESDR.H1.MSC
#>                               STD_NAME
#> 1                         ES-I3 Cells
#> 2                         ES-WA7 Cells
#> 3                         H1 Cells
#> 4 H1 BMP4 Derived Mesendoderm Cultured Cells
#> 5 H1 BMP4 Derived Trophoblast Cultured Cells
#> 6      H1 Derived Mesenchymal Stem Cells
#>                               EDACC_NAME      ANATOMY        TYPE
#> 1          ES-I3_Cell_Line  ESC PrimaryCulture
#> 2          ES-WA7_Cell_Line  ESC PrimaryCulture
#> 3          H1_Cell_Line    ESC PrimaryCulture
#> 4 H1_BMP4_Derived_Mesendoderm_Cultured_Cells ESC_DERIVED  ESCDerived
#> 5 H1_BMP4_Derived_Trophoblast_Cultured_Cells ESC_DERIVED  ESCDerived
#> 6      H1_Derived_Mesenchymal_Stem_Cells ESC_DERIVED  ESCDerived
```

```
#> AGE SEX SOLID_LIQUID ETHNICITY SINGLEDONOR_COMPOSITE
#> 1 CL Female <NA> <NA> SD
#> 2 CL Female <NA> <NA> SD
#> 3 CL Male <NA> <NA> SD
#> 4 CL Male <NA> <NA> SD
#> 5 CL Male <NA> <NA> SD
#> 6 CL Male <NA> <NA> SD
```

To find out the name of the sample corresponding to primary memory T-cells we can filter the data.frame. We extract the sample ID corresponding to our filter.

```
primary_tcells <- subset(metadata,
                           ANATOMY == "BLOOD" & TYPE == "PrimaryCell" &
                           EDACC_NAME == "CD8_Memory_Primary_Cells")$EID
primary_tcells <- as.character(primary_tcells)
```

Now we can take our roadmap hub and query it based on our other conditions:

```
methylation_files <- query(roadmap_hub,
                            c("BigWig", primary_tcells, "H3K4ME[1-3]",
                              "pval.signal"))

methylation_files
#> AnnotationHub with 5 records
#> # snapshotDate(): 2018-06-27
#> # $dataprovicer: BroadInstitute
#> # $species: Homo sapiens
#> # $rdataclass: BigWigFile
#> # additional mcols(): taxonomyid, genome, description,
#> #   coordinate_1_based, maintainer, rdatadateadded, preparerclass,
#> #   tags, rdatapath, sourceurl, sourcetype
#> # retrieve records with, e.g., 'object[["AH33454"]]'
```

#>

```
#>           title
#> AH33454 | E048-H3K4me1.pval.signal.bigwig
#> AH33455 | E048-H3K4me3.pval.signal.bigwig
#> AH39974 | E048-H3K4me1.imputed.pval.signal.bigwig
#> AH40101 | E048-H3K4me2.imputed.pval.signal.bigwig
#> AH40228 | E048-H3K4me3.imputed.pval.signal.bigwig
```

So we'll take the first two entries and download them as BigWigFiles:

```
bw_files <- lapply(methylation_files[1:2], `[[`, 1L)
#> require("rtracklayer")
#> downloading 0 resources
#> loading from cache
#>   '/home/mramos//.AnnotationHub/38894'
#> downloading 0 resources
#> loading from cache
#>   '/home/mramos//.AnnotationHub/38895'
```

We have our desired BigWig files so now we can we can start analyzing them.

4.17.2 Reading BigWig files

For this analysis, we will call peaks from a score vector over chromosome 10.

First, we extract the genome information from the first BigWig file and filter to get the range for chromosome 10. This range will be used as a filter when reading the file.

```
chr10_ranges <- Seqinfo(genome="hg19")["chr10"]
```

Then we read the BigWig file only extracting scores if they overlap chromosome 10.

```
library(rtracklayer)
chr10_scores <- lapply(bw_files, import, which = chr10_ranges,
                        as = "RleList")
chr10_scores[[1]]$chr10
#> numeric-Rle of length 135534747 with 5641879 runs
#>   Lengths:           60612                  172 ...
#>   Values :  0.0394200012087822  0.154219999909401 ...
#>                                         ...               0
```

Each element of the list is a run-length encoded vector of the scores for a particular signal type.

We find the islands by slicing the vectors,

```
islands <- lapply(chr10_scores, slice, lower=1L)
```

where the islands are represented as *Views* objects, i.e., ranges of interest over a genomic vector. Then we find the summit within each island,

```
summits <- lapply(islands, viewRangeMaxs)
```

using the optimized `viewRangeMaxs()` function. Each element of the `summits` list is a *RangesList* object, holding the ranges for each summit. The structure of the *RangesList* keeps track of the chromosome (10) of the summits (there could be multiple chromosomes in general). We broaden the summits and reduce them in order to smooth the peak calls and provide some context,

```
summits <- lapply(lapply(summits, `+`, 50L), reduce)
```

After this preprocessing, we want to convert the result to a more familiar and convenient GRanges object containing an *RleList* “score” column containing the score vector for each summit,

```
summits_grs <- lapply(summits, GRanges)
score_grs <- mapply(function(scores, summits) {
  summits$score <- scores[summits]
  seqlengths(summits) <- lengths(scores)
  summits
}, chr10_scores, summits_grs)
score_gr <- stack(GenomicRangesList(score_grs), index.var="signal_type")
```

One problem with *RangesList* is that it does not keep track of the sequence lengths, so we need to add those after forming the *GRanges*.

We could then find summits with the maximum summit height within each signal type:

```
score_gr$score_max <- max(score_gr$score)
chr10_max_score_region <- aggregate(score_gr, score_max ~ signal_type, max)
```

4.17.3 Exercises

1. Use the `reduce_ranges()` function to find all peaks for each signal type.
2. How could you annotate the scores to find out which genes overlap each peak found in 1.?
3. Plot a 1000nt window centred around the maximum scores for each signal type using the `ggbio` or `Gviz` package.

4.18 Worked example: coverage analysis of BAM files

A common quality control check in a genomics workflow is to perform coverage analysis over features of interest or over the entire genome. Here we use the data from the airway package to operate on read alignment data and compute coverage histograms.

First let's gather all the BAM files available to use in airway (see `browseVignettes("airway")` for more information about the data and how it was prepared):

```
library(tools)
bams <- list_files_with_exts(system.file("extdata", package = "airway"), "bam")
names(bams) <- sub("_[^_]+$", "", basename(bams))
library(Rsamtools)
#> Loading required package: Biostrings
#> Loading required package: XVector
#>
#> Attaching package: 'Biostrings'
#> The following object is masked from 'package:base':
#>
#>     strsplit
bams <- BamFileList(bams)
```

Casting the vector of filenames to a formal `BamFileList` is critical for informing the following code about the nature of the files.

To start let's look at a single BAM file (containing only reads from chr1). We can compute the coverage of the alignments over all contigs in the BAM as follows:

```
first_bam <- bams[[1L]]
first_bam_cvg <- coverage(first_bam)
```

The result is a list of `Rle` objects, one per chromosome. Like other `AtomicList` objects, we can pass our `RleList` to `table()` to compute the coverage histogram by chromosome,

```
head(table(first_bam_cvg)[1L,])
#>      0       1       2       3       4       5
#> 249202844 15607 5247 3055 2030 1280
```

For RNA-seq experiments we are often interested in splitting up alignments based on whether the alignment has skipped a region from the reference (that is, there is an “N” in the cigar string, indicating an intron). We can represent the nested structure using a `GRangesList` object.

To begin we read the BAM file into a `GAlignments` object using `readGAlignments()` and extract the ranges, chopping by introns, using `grlist()`,

```
library(GenomicAlignments)
#> Loading required package: SummarizedExperiment
#> Loading required package: Biobase
#> Welcome to Bioconductor
#>
#> Vignettes contain introductory material; view with
#> 'browseVignettes()'. To cite Bioconductor, see
#> 'citation("Biobase")', and for packages 'citation("pkgname")'.
#>
#> Attaching package: 'Biobase'
#> The following object is masked from 'package:AnnotationHub':
#>
#>     cache
```

```
#> Loading required package: DelayedArray
#> Loading required package: matrixStats
#>
#> Attaching package: 'matrixStats'
#> The following objects are masked from 'package:Biobase':
#>
#>     anyMissing, rowMedians
#> Loading required package: BiocParallel
#>
#> Attaching package: 'DelayedArray'
#> The following objects are masked from 'package:matrixStats':
#>
#>     colMaxs, colMins, colRanges, rowMaxs, rowMins, rowRanges
#> The following object is masked from 'package:Biostrings':
#>
#>     type
#> The following objects are masked from 'package:base':
#>
#>     aperm, apply
reads <- grlist(readGAlignments(first_bam))
```

Finally, we can find the junction reads:

```
reads[lengths(reads) >= 2L]
#> GRangesList object of length 3833:
#> [[1]]
#> GRanges object with 2 ranges and 0 metadata columns:
#>     seqnames           ranges strand
#>             <Rle>           <IRanges>  <Rle>
#>     [1]      1 11072744-11072800      +
#>     [2]      1 11073773-11073778      +
#>
#> [[2]]
#> GRanges object with 2 ranges and 0 metadata columns:
#>     seqnames           ranges strand
#>             <Rle>           <IRanges>  <Rle>
#>     [1]      1 11072745-11072800      -
#>     [2]      1 11073773-11073779      -
#>
#> [[3]]
#> GRanges object with 2 ranges and 0 metadata columns:
#>     seqnames           ranges strand
#>             <Rle>           <IRanges>  <Rle>
#>     [1]      1 11072746-11072800      +
#>     [2]      1 11073773-11073780      +
#>
#> ...
#> <3830 more elements>
#> -----
#> seqinfo: 84 sequences from an unspecified genome
```

We typically want to count how many reads overlap each gene. First, we get the transcript structures as a *GRangesList* from Ensembl,

```
library(GenomicFeatures)
#> Loading required package: AnnotationDbi
library(EnsDb.Hsapiens.v75)
```

```
#> Loading required package: ensemblldb
#> Loading required package: AnnotationFilter
#>
#> Attaching package: 'ensemldb'
#> The following object is masked from 'package:stats':
#>
#>     filter
tx <- exonsBy(EnsDb.Hsapiens.v75, "gene")
```

Finally, we count how many reads overlap each transcript,

```
reads <- keepStandardChromosomes(reads)
counts <- countOverlaps(tx, reads, ignore.strand=TRUE)
head(counts[counts > 0])
#> ENSG00000009724 ENSG00000116649 ENSG00000120942 ENSG00000120948
#>           89          2006           436          5495
#> ENSG00000171819 ENSG00000171824
#>           8          2368
```

To do this over every sample, we use the `summarizeOverlaps()` convenience function,

```
airway <- summarizeOverlaps(features=tx, reads=bams,
                             mode="Union", singleEnd=FALSE,
                             ignore.strand=TRUE, fragments=TRUE)
airway
#> class: RangedSummarizedExperiment
#> dim: 64102 8
#> metadata(0):
#> assays(1): counts
#> rownames(64102): ENSG0000000003 ENSG0000000005 ... LRG_98 LRG_99
#> rowData names(0):
#> colnames(8): SRR1039508 SRR1039509 ... SRR1039520 SRR1039521
#> colData names(0):
```

The `airway` object is a `SummarizedExperiment` object, the central Bioconductor data structure for storing results summarized per feature and sample, along with sample and feature metadata. It is at the point of summarization that workflows switch focus from the ranges infrastructure to Bioconductor modeling packages, most of which consume the `SummarizedExperiment` data structure, so this is an appropriate point to end this tutorial.

4.18.1 Exercises

1. Compute the total depth of coverage across all features.
2. How could you compute the proportion of bases covered over an entire genome? (hint: `seqinfo` and `S4Vectors::merge`)
3. How could you compute the strand specific genome wide coverage?
4. Create a workflow for computing the strand specific coverage for all BAM files.
5. For each sample plot total breadth of coverage against the number of bases covered faceted by each sample name.

4.19 Conclusions

The Bioconductor ranges infrastructure is rich and complex, and it can be intimidating to new users. However, the effort invested will pay dividends, especially when generalizing a series of bespoke analyses into a reusable contribution to Bioconductor.

Chapter 5

103: Public data resources and Bioconductor

5.1 Instructor names and contact information

- Levi Waldron <levi.waldron at sph.cuny.edu> (City University of New York, New York, NY, USA)
- Benjamin Haibe-Kains <benjamin.haibe.kains at utoronto.ca> (Princess Margaret Cancer Center, Toronto, Canada)
- Sean Davis (Center for Cancer Research, National Cancer Institute, National Institutes of Health, Bethesda, MD, USA)

5.2 Syllabus

5.2.1 Workshop Description

The goal of this workshop is to introduce Bioconductor packages for finding, accessing, and using large-scale public data resources including the Gene Expression Omnibus GEO, Sequence Read Archive SRA, the Genomic Data Commons GDC, and Bioconductor-hosted curated data resources for metagenomics, pharmacogenomics PharmacoDB, and The Cancer Genome Atlas.

5.2.2 Pre-requisites

- Basic knowledge of R syntax
- Familiarity with the ExpressionSet and SummarizedExperiment classes
- Basic familiarity with 'omics technologies such as microarray and NGS sequencing

Interested students can prepare by reviewing vignettes of the packages listed in “R/Bioconductor packages used” to gain background on aspects of interest to them.

Some more general background on these resources is published in Kannan et al. (2016).

5.2.3 Workshop Participation

Each component will include runnable examples of typical usage that students are encouraged to run during demonstration of that component.

5.2.4 R/Bioconductor packages used

- *GEOquery*: Access to the NCBI Gene Expression Omnibus (GEO), a public repository of gene expression (primarily microarray) data.
- *GenomicDataCommons*: Access to the NIH / NCI Genomic Data Commons RESTful service.
- *SRAdbV2*: A compilation of metadata from the NCBI Sequence Read Archive, the largest public repository of sequencing data from the next generation of sequencing platforms, and tools
- *curatedTCGAData*: Curated data from The Cancer Genome Atlas (TCGA) as MultiAssayExperiment Objects
- *curatedMetagenomicData*: Curated metagenomic data of the human microbiome
- *HMP16SData*: Curated metagenomic data of the human microbiome
- *Pharmacogenomics*: Curated large-scale preclinical pharmacogenomic data and basic analysis tools

5.2.5 Time outline

This is a 1h45m workshop.

Activity	Time
Overview	10m
GEOquery	15m
GenomicDataCommons	20m
Sequence Read Archive	20m
curatedTCGAData	10m
curatedMetagenomicData and HMP16SData	15m
Pharmacogenomics	20m

5.2.6 Workshop goals and objectives

Bioconductor provides access to significant amounts of publicly available experimental data. This workshop introduces students to Bioconductor interfaces to the NCBI's Gene Expression Omnibus, Genomic Data Commons, Sequence Read Archive and PharmacogenomicsDB. It additionally introduces curated resources providing The Cancer Genome Atlas, the Human Microbiome Project and other microbiome studies, and major pharmacogenomic studies, as native Bioconductor objects ready for analysis and comparison to in-house datasets.

5.2.7 Learning goals

- search NCBI resources for publicly available 'omics data
- quickly use data from the TCGA and the Human Microbiome Project

5.2.8 Learning objectives

- find and download processed microarray and RNA-seq datasets from the Gene Expression Omnibus
- find and download 'omics data from the Genomic Data Commons and Sequence Read Archive
- download and manipulate data from The Cancer Genome Atlas and Human Microbiome Project
- download and explore pharmacogenomics data

5.3 Overview

Before proceeding, ensure that the following packages are installed.

```
required_pkgs = c(
  "TCGAbiolinks",
  "GEOquery",
  "GenomicDataCommons",
  "limma",
  "curatedTCGAData",
  "recount",
  "curatedMetagenomicData",
  "phyloseq",
  "HMP16SDData",
  "caTools",
  "piano",
  "isa",
  "VennDiagram",
  "downloader",
  "gdata",
  "AnnotationDbi",
  "hgu133a.db",
  "PharmacoGx")
BiocManager::install(required_pkgs)
```

5.4 GEOquery

(Davis and Meltzer 2007)

The NCBI Gene Expression Omnibus (GEO) serves as a public repository for a wide range of high-throughput experimental data. These data include single and dual channel microarray-based experiments measuring mRNA, genomic DNA, and protein abundance, as well as non-array techniques such as serial analysis of gene expression (SAGE), mass spectrometry proteomic data, and high-throughput sequencing data. The *GEOquery* package (Davis and Meltzer 2007) forms a bridge between this public repository and the analysis capabilities in Bioconductor.

5.4.1 Overview of GEO

At the most basic level of organization of GEO, there are four basic entity types. The first three (Sample, Platform, and Series) are supplied by users; the fourth, the dataset, is compiled and curated by GEO staff from the user-submitted data. See the GEO home page for more information.

5.4.1.1 Platforms

A Platform record describes the list of elements on the array (e.g., cDNAs, oligonucleotide probesets, ORFs, antibodies) or the list of elements that may be detected and quantified in that experiment (e.g., SAGE tags, peptides). Each Platform record is assigned a unique and stable GEO accession number (GPLxxx). A Platform may reference many Samples that have been submitted by multiple submitters.

5.4.1.2 Samples

A Sample record describes the conditions under which an individual Sample was handled, the manipulations it underwent, and the abundance measurement of each element derived from it. Each Sample record is assigned a unique and stable GEO accession number (GSMxxx). A Sample entity must reference only one Platform and may be included in multiple Series.

5.4.1.3 Series

A Series record defines a set of related Samples considered to be part of a group, how the Samples are related, and if and how they are ordered. A Series provides a focal point and description of the experiment as a whole. Series records may also contain tables describing extracted data, summary conclusions, or analyses. Each Series record is assigned a unique and stable GEO accession number (GSExxx). Series records are available in a couple of formats which are handled by GEOquery independently. The smaller and new GSEMatrix files are quite fast to parse; a simple flag is used by GEOquery to choose to use GSEMatrix files (see below).

5.4.1.4 Datasets

GEO DataSets (GDSxxx) are curated sets of GEO Sample data. There are hundreds of GEO datasets available, but GEO discontinued creating GDS records several years ago. We mention them here for completeness only.

5.4.2 Getting Started using GEOquery

Getting data from GEO is really quite easy. There is only one command that is needed, `getGEO`. This one function interprets its input to determine how to get the data from GEO and then parse the data into useful R data structures.

```
library(GEOquery)
```

With the library loaded, we are free to access any GEO accession.

5.4.3 Use case: MDS plot of cancer data

The data we are going to access are from this paper.

Background: The tumor microenvironment is an important factor in cancer immunotherapy response. To further understand how a tumor affects the local immune system, we analyzed immune gene expression differences between matching normal and tumor tissue.
Methods: We analyzed public and new gene expression data from solid cancers and isolated immune cell populations. We also determined the correlation between CD8, FoxP3 IHC, and our gene signatures.
Results: We observed that regulatory T cells (Tregs) were one of the main drivers of immune gene expression differences between normal and tumor tissue. A tumor-specific CD8 signature was slightly lower in tumor tissue compared with normal of most (12 of 16) cancers, whereas a Treg signature was higher in tumor tissue of all cancers except liver. Clustering by Treg signature found two groups in colorectal cancer datasets. The high Treg cluster had more samples that were consensus molecular subtype 1/4, right-sided, and microsatellite-unstable, compared with the low Treg cluster. Finally, we found that the correlation between signature and IHC was low in our small dataset, but samples in the high Treg cluster had significantly more CD8+ and FoxP3+ cells compared with the low Treg cluster.
Conclusions: Treg gene expression is highly indicative of the overall tumor immune environment.
Impact: In comparison with the consensus molecular subtype and microsatellite status, the Treg signature identifies more colorectal tumors with high immune activation that may benefit from cancer immunotherapy.

In this little exercise, we will:

1. Access public omics data using the GEOquery package
2. Convert the public omics data to a `SummarizedExperiment` object.
3. Perform a simple unsupervised analysis to visualize these public data.

Use the GEOquery package to fetch data about GSE103512.

```
gse = getGEO("GSE103512")[[1]]
```

Note that `getGEO`, when used to retrieve *GSE* records, returns a list. The members of the list each represent one *GEO Platform*, since each *GSE* record can contain multiple related datasets (eg., gene expression and DNA methylation). In this case, the list is of length one, but it is still necessary to grab the first element.

The first step—a detail—is to convert from the older Bioconductor data structure (GEOquery was written in 2007), the `ExpressionSet`, to the newer `SummarizedExperiment`. One line suffices.

```
library(SummarizedExperiment)
se = as(gse, "SummarizedExperiment")
```

Examine two variables of interest, cancer type and tumor/normal status.

```
with(colData(se), table(`cancer.type.ch1`, `normal.ch1`))
#>           normal.ch1
#> cancer.type.ch1 no yes
#>          BC    65   10
#>          CRC   57   12
#>          NSCLC 60    9
#>          PCA   60    7
```

Filter gene expression by variance to find most informative genes.

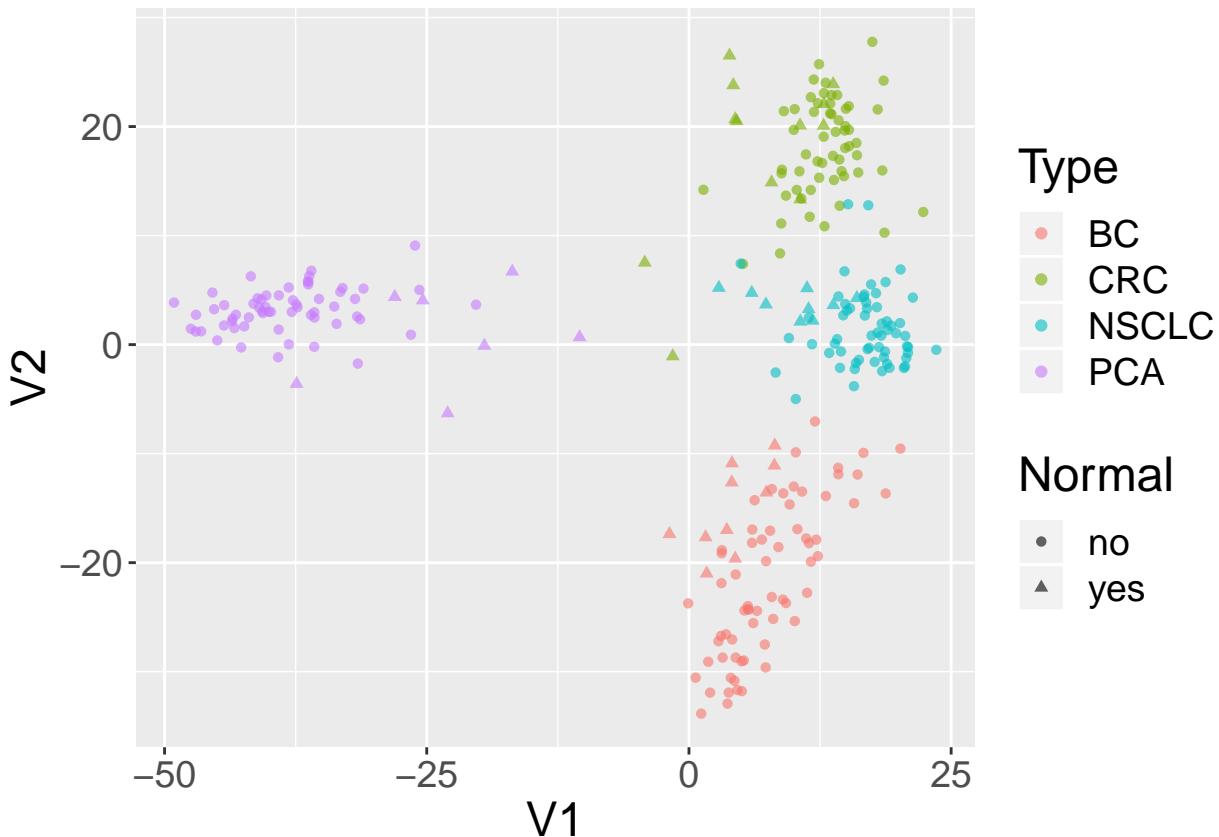
```
sds = apply(assay(se, 'exprs'), 1, sd)
dat = assay(se, 'exprs')[order(sds, decreasing = TRUE)[1:500],]
```

Perform multidimensional scaling and prepare for plotting. We will be using ggplot2, so we need to make a `data.frame` before plotting.

```
mdsvals = cmdscale(dist(t(dat)))
mdsvals = as.data.frame(mdsvals)
mdsvals$Type= factor(colData(se)[, 'cancer.type.ch1'])
mdsvals$Normal = factor(colData(se)[, 'normal.ch1'])
head(mdsvals)
#>           V1      V2 Type Normal
#> GSM2772660 8.531331 -18.57115 BC    no
#> GSM2772661 8.991591 -13.63764 BC    no
#> GSM2772662 10.788973 -13.48403 BC    no
#> GSM2772663 3.127105 -19.13529 BC    no
#> GSM2772664 13.056599 -13.88711 BC    no
#> GSM2772665 7.903717 -13.24731 BC    no
```

And do the plot.

```
library(ggplot2)
ggplot(mdsvals, aes(x=V1,y=V2,shape=Normal,color=Type)) +
  geom_point( alpha=0.6) + theme(text=element_text(size = 18))
```



5.4.4 Accessing Raw Data from GEO

NCBI GEO accepts (but has not always required) raw data such as .CEL files, .CDF files, images, etc. It is also not uncommon for some RNA-seq or other sequencing datasets to supply *only* raw data (with accompanying sample information, of course), necessitating Sometimes, it is useful to get quick access to such data. A single function, `getGEOSuppFiles`, can take as an argument a GEO accession and will download all the raw data associate with that accession. By default, the function will create a directory in the current working directory to store the raw data for the chosen GEO accession.

5.5 GenomicDataCommons

From the Genomic Data Commons (GDC) website:

The National Cancer Institute's (NCI's) Genomic Data Commons (GDC) is a data sharing platform that promotes precision medicine in oncology. It is not just a database or a tool; it is an expandable knowledge network supporting the import and standardization of genomic and clinical data from cancer research programs. The GDC contains NCI-generated data from some of the largest and most comprehensive cancer genomic datasets, including The Cancer Genome Atlas (TCGA) and Therapeutically Applicable Research to Generate Effective Therapies (TARGET). For the first time, these datasets have been harmonized using a common set of bioinformatics pipelines, so that the data can be directly compared. As a growing knowledge system for cancer, the GDC also enables researchers to submit data, and harmonizes these data for import into the GDC. As more researchers add clinical and genomic data to the GDC, it will become an even more powerful tool for making discoveries about the molecular basis of cancer that may lead to better care for patients.

The data model for the GDC is complex, but it worth a quick overview and a graphical representation is included here.

The GDC API exposes these nodes and edges in a somewhat simplified set of RESTful endpoints.

5.5.1 Quickstart

This quickstart section is just meant to show basic functionality. More details of functionality are included further on in this vignette and in function-specific help.

To report bugs or problems, either submit a new issue or submit a `bug.report(package='GenomicDataCommons')` from within R (which will redirect you to the new issue on GitHub).

5.5.1.1 Installation

Installation of the *GenomicDataCommons* package is identical to installation of other Bioconductor packages.

```
install.packages('BiocManager')
BiocManager::install('GenomicDataCommons')
```

After installation, load the library in order to use it.

```
library(GenomicDataCommons)
```

5.5.1.2 Check connectivity and status

The *GenomicDataCommons* package relies on having network connectivity. In addition, the NCI GDC API must also be operational and not under maintenance. Checking `status` can be used to check this connectivity and functionality.

```
GenomicDataCommons::status()
#> $commit
#> [1] "e9e20d6f97f2bf6dd3b3261e36ead57c56a4c7cc"
#>
#> $data_release
#> [1] "Data Release 12.0 - June 13, 2018"
#>
#> $status
#> [1] "OK"
#>
#> $tag
#> [1] "1.14.1"
#>
#> $version
#> [1] 1
```

5.5.1.3 Find data

The following code builds a `manifest` that can be used to guide the download of raw data. Here, filtering finds gene expression files quantified as raw counts using HTSeq from ovarian cancer patients.

```
ge_manifest = files() %>%
  filter(~ cases.project.project_id == 'TCGA-OV' &
        type == 'gene_expression' &
```

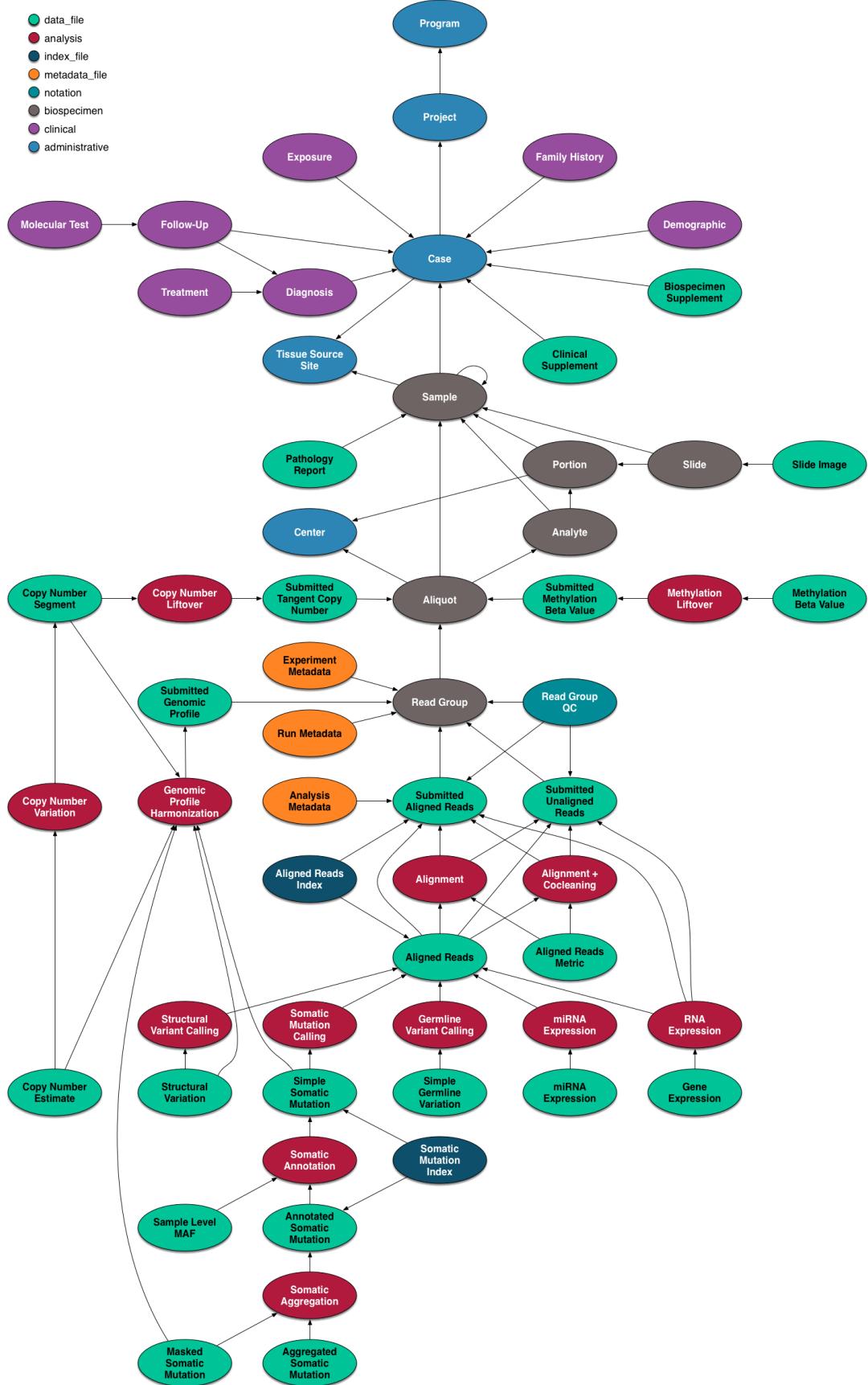


Figure 5.1: The data model is encoded as a so-called property graph. Nodes represent entities such as Projects, Cases, Diagnoses, Files (various kinds), and Annotations. The relationships between these entities are maintained as edges. Both nodes and edges may have Properties that supply instance details.

```
analysis.workflow_type == 'HTSeq - Counts') %>%
manifest()
```

5.5.1.4 Download data

After the 379 gene expression files specified in the query above. Using multiple processes to do the download very significantly speeds up the transfer in many cases. On a standard 1Gb connection, the following completes in about 30 seconds. The first time the data are downloaded, R will ask to create a cache directory (see `?gdc_cache` for details of setting and interacting with the cache). Resulting downloaded files will be stored in the cache directory. Future access to the same files will be directly from the cache, alleviating multiple downloads.

```
fnames = lapply(ge_manifest$id[1:20], gdcdata)
```

If the download had included controlled-access data, the download above would have needed to include a `token`. Details are available in the authentication section below.

5.5.1.5 Metadata queries

The *GenomicDataCommons* can access the significant clinical, demographic, biospecimen, and annotation information contained in the NCI GDC.

```
expands = c("diagnoses", "annotations",
           "demographic", "exposures")
projResults = projects() %>%
  results(size=10)
str(projResults, list.len=5)
#> List of 8
#> $ dbgap_accession_number: chr [1:10] "phs001179" "phs000470" NA NA ...
#> $ disease_type :List of 10
#>   ..$ FM-AD : chr [1:23] "Germ Cell Neoplasms" "Acinar Cell Neoplasms" "Miscellaneous Tumors" "The...
#>   ..$ TARGET-RT: chr "Rhabdoid Tumor"
#>   ..$ TCGA-UCS : chr "Uterine Carcinosarcoma"
#>   ..$ TCGA-LUSC: chr "Lung Squamous Cell Carcinoma"
#>   ..$ TCGA-BRCA: chr "Breast Invasive Carcinoma"
#>   ... [list output truncated]
#> $ released : logi [1:10] TRUE TRUE TRUE TRUE TRUE TRUE ...
#> $ state : chr [1:10] "open" "open" "open" "open" ...
#> $ primary_site :List of 10
#>   ..$ FM-AD : chr [1:42] "Kidney" "Testis" "Unknown" "Other and unspecified parts of biliary trac...
#>   ..$ TARGET-RT: chr "Kidney"
#>   ..$ TCGA-UCS : chr "Uterus"
#>   ..$ TCGA-LUSC: chr "Lung"
#>   ..$ TCGA-BRCA: chr "Breast"
#>   ... [list output truncated]
#>   [list output truncated]
#> - attr(*, "row.names")= int [1:10] 1 2 3 4 5 6 7 8 9 10
#> - attr(*, "class")= chr [1:3] "GDCprojectsResults" "GDCResults" "list"
names(projResults)
#> [1] "dbgap_accession_number" "disease_type"
#> [3] "released" "state"
#> [5] "primary_site" "project_id"
```

```
#> [7] "id"           "name"
# or listviewer::jsonedit(clinResults)
```

5.5.2 Basic design

This package design is meant to have some similarities to the “hadleyverse” approach of dplyr. Roughly, the functionality for finding and accessing files and metadata can be divided into:

1. Simple query constructors based on GDC API endpoints.
2. A set of verbs that when applied, adjust filtering, field selection, and facetting (fields for aggregation) and result in a new query object (an endomorphism)
3. A set of verbs that take a query and return results from the GDC

In addition, there are auxiliary functions for asking the GDC API for information about available and default fields, slicing BAM files, and downloading actual data files. Here is an overview of functionality¹.

- Creating a query
 - `projects()`
 - `cases()`
 - `files()`
 - `annotations()`
- Manipulating a query
 - `filter()`
 - `facet()`
 - `select()`
- Introspection on the GDC API fields
 - `mapping()`
 - `available_fields()`
 - `default_fields()`
 - `grep_fields()`
 - `field_picker()`
 - `available_values()`
 - `available_expand()`
- Executing an API call to retrieve query results
 - `results()`
 - `count()`
 - `response()`
- Raw data file downloads
 - `gdcdata()`
 - `transfer()`
 - `gdc_client()`
- Summarizing and aggregating field values (faceting)
 - `aggregations()`
- Authentication
 - `gdc_token()`
- BAM file slicing
 - `slicing()`

5.5.3 Usage

There are two main classes of operations when working with the NCI GDC.

¹See individual function and methods documentation for specific details.

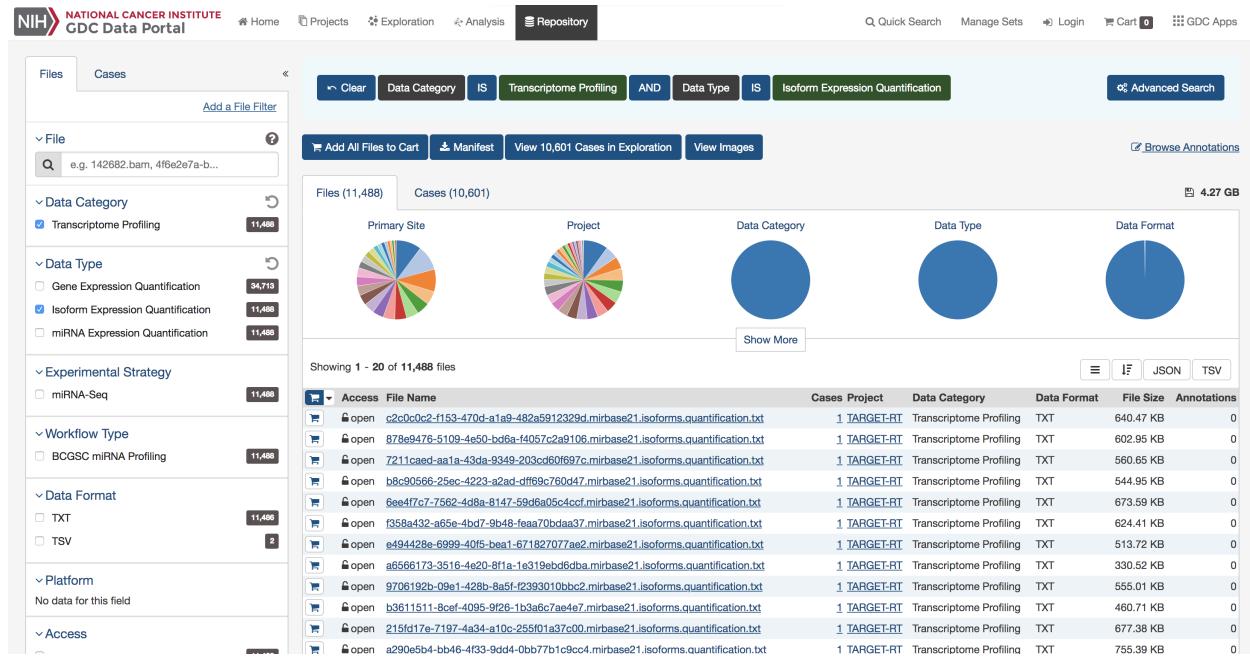


Figure 5.2: A screenshot of an example query of the GDC repository portal.

1. Querying metadata and finding data files (e.g., finding all gene expression quantifications data files for all colon cancer patients).
2. Transferring raw or processed data from the GDC to another computer (e.g., downloading raw or processed data)

Both classes of operation are reviewed in detail in the following sections.

5.6 Querying metadata

Vast amounts of metadata about cases (patients, basically), files, projects, and so-called annotations are available via the NCI GDC API. Typically, one will want to query metadata to either focus in on a set of files for download or transfer *or* to perform so-called aggregations (pivot-tables, facets, similar to the R `table()` functionality).

Querying metadata starts with creating a “blank” query. One will often then want to `filter` the query to limit results prior to retrieving results. The `GenomicDataCommons` package has helper functions for listing fields that are available for filtering.

In addition to fetching results, the GDC API allows facetting, or aggregating,, useful for compiling reports, generating dashboards, or building user interfaces to GDC data (see GDC web query interface for a non-R-based example).

5.6.0.1 Creating a query

The `GenomicDataCommons` package accesses the *same* API as the *GDC* website. Therefore, a useful approach, particularly for beginning users is to examine the filters available on the GDC repository pages to find appropriate filtering criteria. From there, converting those checkboxes to a `GenomicDataCommons` `query()` is relatively straightforward. Note that only a small subset of the `available_fields()` are available by default on the website.

A query of the GDC starts its life in R. Queries follow the four metadata endpoints available at the GDC. In particular, there are four convenience functions that each create `GDCQuery` objects (actually, specific subclasses of `GDCQuery`):

- `projects()`
- `cases()`
- `files()`
- `annotations()`

```
pquery = projects()
```

The `pquery` object is now an object of (S3) class, `GDCQuery` (and `gdc_projects` and `list`). The object contains the following elements:

- fields: This is a character vector of the fields that will be returned when we retrieve data. If no fields are specified to, for example, the `projects()` function, the default fields from the GDC are used (see `default_fields()`)
- filters: This will contain results after calling the `filter()` method and will be used to filter results on retrieval.
- facets: A character vector of field names that will be used for aggregating data in a call to `aggregations()`.
- archive: One of either “default” or “legacy”.
- token: A character(1) token from the GDC. See the authentication section for details, but note that, in general, the token is not necessary for metadata query and retrieval, only for actual data download.

Looking at the actual object (get used to using `str()!`), note that the query contains no results.

```
str(pquery)
#> List of 5
#> $ fields : chr [1:16] "awg_review" "dbgap_accession_number" "disease_type" "in_review" ...
#> $ filters: NULL
#> $ facets : NULL
#> $ legacy : logi FALSE
#> $ expand : NULL
#> - attr(*, "class")= chr [1:3] "gdc_projects" "GDCQuery" "list"
```

5.6.0.2 Retrieving results

[GDC pagination documentation]

[GDC sorting documentation]

With a query object available, the next step is to retrieve results from the GDC. The `GenomicDataCommons` package. The most basic type of results we can get is a simple `count()` of records available that satisfy the filter criteria. Note that we have not set any filters, so a `count()` here will represent all the project records publicly available at the GDC in the “default” archive”

```
pcount = count(pquery)
# or
pcount = pquery %>% count()
pcount
#> [1] 40
```

The `results()` method will fetch actual results.

```
presets = pquery %>% results()
```

These results are returned from the GDC in JSON format and converted into a (potentially nested) list in R. The `str()` method is useful for taking a quick glimpse of the data.

```
str(presults)
#> List of 8
#> $ dbgap_accession_number: chr [1:10] "phs001179" "phs000470" NA NA ...
#> $ disease_type      :List of 10
#>   ..$ FM-AD       : chr [1:23] "Germ Cell Neoplasms" "Acinar Cell Neoplasms" "Miscellaneous Tumors" "Th...
#>   ..$ TARGET-RT     : chr "Rhabdoid Tumor"
#>   ..$ TCGA-UCS     : chr "Uterine Carcinosarcoma"
#>   ..$ TCGA-LUSC    : chr "Lung Squamous Cell Carcinoma"
#>   ..$ TCGA-BRCA    : chr "Breast Invasive Carcinoma"
#>   ..$ TCGA-SKCM    : chr "Skin Cutaneous Melanoma"
#>   ..$ TARGET-OS     : chr "Osteosarcoma"
#>   ..$ TCGA-THYM    : chr "Thymoma"
#>   ..$ TARGET-WT     : chr "High-Risk Wilms Tumor"
#>   ..$ TCGA-ESCA    : chr "Esophageal Carcinoma"
#> $ released          : logi [1:10] TRUE TRUE TRUE TRUE TRUE TRUE ...
#> $ state             : chr [1:10] "open" "open" "open" "open" ...
#> $ primary_site      :List of 10
#>   ..$ FM-AD       : chr [1:42] "Kidney" "Testis" "Unknown" "Other and unspecified parts of biliary trac...
#>   ..$ TARGET-RT     : chr "Kidney"
#>   ..$ TCGA-UCS     : chr "Uterus"
#>   ..$ TCGA-LUSC    : chr "Lung"
#>   ..$ TCGA-BRCA    : chr "Breast"
#>   ..$ TCGA-SKCM    : chr "Skin"
#>   ..$ TARGET-OS     : chr "Bone"
#>   ..$ TCGA-THYM    : chr "Thymus"
#>   ..$ TARGET-WT     : chr "Kidney"
#>   ..$ TCGA-ESCA    : chr "Esophagus"
#> $ project_id        : chr [1:10] "FM-AD" "TARGET-RT" "TCGA-UCS" "TCGA-LUSC" ...
#> $ id                : chr [1:10] "FM-AD" "TARGET-RT" "TCGA-UCS" "TCGA-LUSC" ...
#> $ name              : chr [1:10] "Foundation Medicine Adult Cancer Clinical Dataset (FM-AD)" "R...
#> - attr(*, "row.names")= int [1:10] 1 2 3 4 5 6 7 8 9 10
#> - attr(*, "class")= chr [1:3] "GDCprojectsResults" "GDCResults" "list"
```

A default of only 10 records are returned. We can use the `size` and `from` arguments to `results()` to either page through results or to change the number of results. Finally, there is a convenience method, `results_all()` that will simply fetch all the available results given a query. Note that `results_all()` may take a long time and return HUGE result sets if not used carefully. Use of a combination of `count()` and `results()` to get a sense of the expected data size is probably warranted before calling `results_all()`

```
length(ids(presults))
#> [1] 10
presults = pquery %>% results_all()
length(ids(presults))
#> [1] 40
# includes all records
length(ids(presults)) == count(pquery)
#> [1] TRUE
```

Extracting subsets of results or manipulating the results into a more conventional R data structure is not easily generalizable. However, the purrr, rlist, and data.tree packages are all potentially of interest for manipulating complex, nested list structures. For viewing the results in an interactive viewer, consider the listviewer package.

5.6.0.3 Fields and Values

[GDC fields documentation]

Central to querying and retrieving data from the GDC is the ability to specify which fields to return, filtering by fields and values, and faceting or aggregating. The GenomicDataCommons package includes two simple functions, `available_fields()` and `default_fields()`. Each can operate on a character(1) endpoint name (“cases”, “files”, “annotations”, or “projects”) or a `GDCQuery` object.

```
default_fields('files')
#> [1] "access"                  "acl"
#> [3] "batch_id"                "created_datetime"
#> [5] "data_category"           "data_format"
#> [7] "data_type"                "error_type"
#> [9] "experimental_strategy"   "file_autocomplete"
#> [11] "file_id"                 "file_name"
#> [13] "file_size"               "file_state"
#> [15] "imaging_date"            "magnification"
#> [17] "md5sum"                  "origin"
#> [19] "platform"                "read_pair_number"
#> [21] "revision"                "state"
#> [23] "state_comment"           "submitter_id"
#> [25] "tags"                     "type"
#> [27] "updated_datetime"

# The number of fields available for files endpoint
length(available_fields('files'))
#> [1] 703

# The first few fields available for files endpoint
head(available_fields('files'))
#> [1] "access"                  "acl"
#> [3] "analysis.analysis_id"    "analysis.analysis_type"
#> [5] "analysis.batch_id"        "analysis.created_datetime"
```

The fields to be returned by a query can be specified following a similar paradigm to that of the `dplyr` package. The `select()` function is a verb that resets the fields slot of a `GDCQuery`; note that this is not quite analogous to the `dplyr` `select()` verb that limits from already-present fields. We *completely replace* the fields when using `select()` on a `GDCQuery`.

```
# Default fields here
qcases = cases()
qcases$fields
#> [1] "aliquot_ids"              "analyte_ids"
#> [3] "batch_id"                 "case_autocomplete"
#> [5] "case_id"                  "created_datetime"
#> [7] "days_to_index"             "days_to_lost_to_followup"
#> [9] "disease_type"              "index_date"
#> [11] "lost_to_followup"         "portion_ids"
#> [13] "primary_site"             "sample_ids"
#> [15] "slide_ids"                "state"
#> [17] "submitter_aliquot_ids"    "submitter_analyte_ids"
#> [19] "submitter_id"               "submitter_portion_ids"
#> [21] "submitter_sample_ids"      "submitter_slide_ids"
#> [23] "updated_datetime"

# set up query to use ALL available fields
# Note that checking of fields is done by select()
qcases = cases() %>% GenomicDataCommons::select(available_fields('cases'))
```

```
head(qcases$fields)
#> [1] "case_id"                      "aliquot_ids"
#> [3] "analyte_ids"                  "annotations.annotation_id"
#> [5] "annotations.batch_id"          "annotations.case_id"
```

Finding fields of interest is such a common operation that the GenomicDataCommons includes the `grep_fields()` function and the `field_picker()` widget. See the appropriate help pages for details.

5.6.0.4 Facets and aggregation

[GDC facet documentation]

The GDC API offers a feature known as aggregation or faceting. By specifying one or more fields (of appropriate type), the GDC can return to us a count of the number of records matching each potential value. This is similar to the R `table` method. Multiple fields can be returned at once, but the GDC API does not have a cross-tabulation feature; all aggregations are only on one field at a time. Results of `aggregation()` calls come back as a list of data.frames (actually, tibbles).

```
# total number of files of a specific type
res = files() %>% facet(c('type','data_type')) %>% aggregations()
res$type
#>
#> key doc_count
#> 1 simple_somatic_mutation 64015
#> 2 annotated_somatic_mutation 63580
#> 3 aligned_reads 45985
#> 4 copy_number_segment 44752
#> 5 gene_expression 34713
#> 6 slide_image 30036
#> 7 biospecimen_supplement 25151
#> 8 mirna_expression 22976
#> 9 clinical_supplement 12496
#> 10 methylation_beta_value 12359
#> 11 aggregated_somatic_mutation 186
#> 12 masked_somatic_mutation 132
```

Using `aggregations()` is an also easy way to learn the contents of individual fields and forms the basis for faceted search pages.

5.6.0.5 Filtering

[GDC filtering documentation]

The GenomicDataCommons package uses a form of non-standard evaluation to specify R-like queries that are then translated into an R list. That R list is, upon calling a method that fetches results from the GDC API, translated into the appropriate JSON string. The R expression uses the formula interface as suggested by Hadley Wickham in his vignette on non-standard evaluation

It's best to use a formula because a formula captures both the expression to evaluate and the environment where the evaluation occurs. This is important if the expression is a mixture of variables in a data frame and objects in the local environment [for example].

For the user, these details will not be too important except to note that a filter expression must begin with a “~”.

```
qfiles = files()
qfiles %>% count() # all files
#> [1] 356381
```

To limit the file type, we can refer back to the section on facetting to see the possible values for the file field “type”. For example, to filter file results to only “gene_expression” files, we simply specify a filter.

```
qfiles = files() %>% filter(~ type == 'gene_expression')
# here is what the filter looks like after translation
str(get_filter(qfiles))
#> List of 2
#> $ op      : 'scalar' chr "="
#> $ content:List of 2
#>   ..$ field: chr "type"
#>   ..$ value: chr "gene_expression"
```

What if we want to create a filter based on the project (‘TCGA-OVCA’, for example)? Well, we have a couple of possible ways to discover available fields. The first is based on base R functionality and some intuition.

```
grep('pro',available_fields('files'),value=TRUE)
#> [1] "cases.diagnoses.progression_free_survival"
#> [2] "cases.diagnoses.progression_free_survival_event"
#> [3] "cases.diagnoses.progression_or_recurrence"
#> [4] "cases.project.awg_review"
#> [5] "cases.project.dbgap_accession_number"
#> [6] "cases.project.disease_type"
#> [7] "cases.project.in_review"
#> [8] "cases.project.intended_release_date"
#> [9] "cases.project.is_legacy"
#> [10] "cases.project.name"
#> [11] "cases.project.primary_site"
#> [12] "cases.project.program.dbgap_accession_number"
#> [13] "cases.project.program.name"
#> [14] "cases.project.program.program_id"
#> [15] "cases.project.project_id"
#> [16] "cases.project.releasable"
#> [17] "cases.project.release_requested"
#> [18] "cases.project.released"
#> [19] "cases.project.request_submission"
#> [20] "cases.project.state"
#> [21] "cases.project.submission_enabled"
#> [22] "cases.samples.days_to_sample_procurement"
#> [23] "cases.samples.method_of_sample_procurement"
#> [24] "cases.samples.portions.slides.number_proliferating_cells"
#> [25] "cases.tissue_source_site.project"
```

Interestingly, the project information is “nested” inside the case. We don’t need to know that detail other than to know that we now have a few potential guesses for where our information might be in the files records. We need to know where because we need to construct the appropriate filter.

```
files() %>% facet('cases.project.project_id') %>% aggregations()
#> $cases.project.project_id
#>   key doc_count
#> 1      FM-AD      36134
#> 2      TCGA-BRCA    31511
#> 3      TCGA-LUAD    17051
```

```
#> 4   TCGA-UCEC    16130
#> 5   TCGA-HNSC    15266
#> 6   TCGA-OV      15057
#> 7   TCGA-THCA    14420
#> 8   TCGA-LUSC    15323
#> 9   TCGA-LGG     14723
#> 10  TCGA-KIRC    15082
#> 11  TCGA-PRAD    14287
#> 12  TCGA-COAD    14270
#> 13  TCGA-GBM     11973
#> 14  TCGA-SKCM    12724
#> 15  TCGA-STAD    12845
#> 16  TCGA-BLCA    11710
#> 17  TCGA-LIHC    10814
#> 18  TCGA-CESC    8593
#> 19  TCGA-KIRP    8506
#> 20  TCGA-SARC    7493
#> 21  TCGA-PAAD    5306
#> 22  TCGA-ESCA    5270
#> 23  TCGA-PCPG    5032
#> 24  TCGA-READ    4918
#> 25  TCGA-TGCT    4217
#> 26  TCGA-THYM    3444
#> 27  TCGA-LAML    3960
#> 28  TARGET-NBL    2795
#> 29  TCGA-ACC     2546
#> 30  TCGA-KICH    2324
#> 31  TCGA-MESO    2330
#> 32  TARGET-AML    2170
#> 33  TCGA-UVM    2179
#> 34  TCGA-UCS    1658
#> 35  TARGET-WT    1406
#> 36  TCGA-DLBC    1330
#> 37  TCGA-CHOL    1348
#> 38  TARGET-OS     47
#> 39  TARGET-RT     174
#> 40  TARGET-CCSK    15
```

We note that `cases.project.project_id` looks like it is a good fit. We also note that TCGA-OV is the correct project_id, not TCGA-OVCA. Note that *unlike with dplyr and friends, the filter() method here replaces the filter and does not build on any previous filters.*

```
qfiles = files() %>%
  filter(~ cases.project.project_id == 'TCGA-OV' & type == 'gene_expression')
str(get_filter(qfiles))
#> List of 2
#> $ op      : 'scalar' chr "and"
#> $ content:List of 2
#>   ..$ :List of 2
#>     ...$ op      : 'scalar' chr "="
#>     ...$ content:List of 2
#>       ...$ field: chr "cases.project.project_id"
#>       ...$ value: chr "TCGA-OV"
#>     ..$ :List of 2
```

```
#> ... . $ op      : 'scalar' chr "="
#> ... . $ content: List of 2
#> ... . . $ field: chr "type"
#> ... . . $ value: chr "gene_expression"
qfiles %>% count()
#> [1] 1137
```

Asking for a `count()` of results given these new filter criteria gives `r qfiles %>% count()` results. Generating a manifest for bulk downloads is as simple as asking for the manifest from the current query.

```
manifest_df = qfiles %>% manifest()
head(manifest_df)
#> # A tibble: 6 x 5
#>   id              filename          md5        size state
#>   <chr>            <chr>           <chr>      <int> <chr>
#> 1 567ced20-00cf-46~ b2552f6f-dd15-410f-a621~~ 9af0d993c40aec~ 258324 live
#> 2 05692746-1770-47~ 701b8c71-6c05-4e5b-ac10~~ 8e9816f4d9b871~ 526537 live
#> 3 e2d47640-8565-43~ b2552f6f-dd15-410f-a621~~ e05190ed65c8a8~ 543367 live
#> 4 bc6dab72-dc5a-4c~ a1c4f19e-079e-47e7-8939~~ 110d8cda0ccdf6~ 253059 live
#> 5 0a176c20-f3f3-4b~ 01eac123-1e21-440d-9495~~ b40921f17128a9~ 540592 live
#> 6 2ae73487-7acf-42~ 12c8b289-b9d0-4697-b3a6~~ 4d3c2b951d94f0~ 549437 live
```

Note that we might still not be quite there. Looking at filenames, there are suspiciously named files that might include “FPKM”, “FPKM-UQ”, or “counts”. Another round of `grep` and `available_fields`, looking for “type” turned up that the field “analysis.workflow_type” has the appropriate filter criteria.

```
qfiles = files() %>% filter( ~ cases.project.project_id == 'TCGA-OV' &
                           type == 'gene_expression' &
                           analysis.workflow_type == 'HTSeq - Counts')
manifest_df = qfiles %>% manifest()
nrow(manifest_df)
#> [1] 379
```

The GDC Data Transfer Tool can be used (from R, `transfer()` or from the command-line) to orchestrate high-performance, restartable transfers of all the files in the manifest. See the bulk downloads section for details.

5.6.1 Authentication

[GDC authentication documentation]

The GDC offers both “controlled-access” and “open” data. As of this writing, only data stored as files is “controlled-access”; that is, metadata accessible via the GDC is all “open” data and some files are “open” and some are “controlled-access”. Controlled-access data are only available after going through the process of obtaining access.

After controlled-access to one or more datasets has been granted, logging into the GDC web portal will allow you to access a GDC authentication token, which can be downloaded and then used to access available controlled-access data via the `GenomicDataCommons` package.

The `GenomicDataCommons` uses authentication tokens only for downloading data (see `transfer` and `gdcdat` documentation). The package includes a helper function, `gdc_token`, that looks for the token to be stored in one of three ways (resolved in this order):

1. As a string stored in the environment variable, `GDC_TOKEN`
2. As a file, stored in the file named by the environment variable, `GDC_TOKEN_FILE`

3. In a file in the user home directory, called `.gdc_token`

As a concrete example:

```
token = gdc_token()
transfer(..., token=token)
# or
transfer(..., token=get_token())
```

5.6.2 Datafile access and download

The `gdcdat` function takes a character vector of one or more file ids. A simple way of producing such a vector is to produce a `manifest` data frame and then pass in the first column, which will contain file ids.

```
fnames = gdcdat(manifest_df$id[1:2], progress=FALSE)
```

Note that for controlled-access data, a GDC authentication token is required. Using the `BiocParallel` package may be useful for downloading in parallel, particularly for large numbers of smallish files.

The bulk download functionality is only efficient (as of v1.2.0 of the GDC Data Transfer Tool) for relatively large files, so use this approach only when transferring BAM files or larger VCF files, for example. Otherwise, consider using the approach shown above, perhaps in parallel.

```
fnames = gdcdat(manifest_df$id[3:10], access_method = 'client')
```

5.7 Sequence Read Archive

The SRAdbV2 package is currently available from GitHub and is under active development. Either the `devtools` package or the `BiocManager` package can be used for easy installation.

```
install.packages('BiocManager')
BiocManager::install('seandavi/SRAdbV2')
```

5.7.1 Usage

5.7.1.1 Loading the library

```
library(SRAdbV2)
#> Loading required package: R6
```

5.7.1.2 The Omicidx

The entrypoint for using the SRAdbV2 system is the `Omicidx`, an R6 class. To start, create a new instance of the class.

```
oidx = Omicidx$new()
```

Typing `oidx$` and then TAB will give possible completions. Note the “search” completion.

5.7.1.3 Queries

Once an instance of `Omicidx` is created (here we will call the instance `oidx`), search capabilities are available via `oidx$search()`. The one interesting parameter is the `q` parameter. This parameter takes a string formatted as a Lucene query string. See below for Query syntax.

```
query=paste(
  paste0('sample_taxon_id:', 10116),
  'AND experiment_library_strategy:"rna seq"',
  'AND experiment_library_source:transcriptomic',
  'AND experiment_platform:illumina')
z = oidx$search(q=query,entity='full',size=100L)
```

The `entity` parameter is one of the SRA entity types available via the API. The `size` parameter is the number of records that will be returned in each “chunk”.

5.7.1.4 Fetching results

Because result sets can be large, we have a special method that allows us to “scroll” through the results or to simply get them *en bloc*. The first step for result retrieval, then, is to get a `Scroller`.

```
s = z$scroll()
s
#> <Scroller>
#>   Public:
#>     clone: function (deep = FALSE)
#>     collate: function (limit = Inf)
#>     count: active binding
#>     fetched: active binding
#>     has_next: function ()
#>     initialize: function (search, progress = interactive())
#>     reset: function ()
#>     yield: function ()
#>   Private:
#>     .count: NULL
#>     .fetched: 0
#>     .last: FALSE
#>     progress: FALSE
#>     scroll: 1m
#>     scroll_id: NULL
#>     search: Searcher, R6
```

Methods such as `s$count` allow introspection into the available number of results, in this case, 8886 records.

The `Scroller` provides two different approaches to accessing the resulting data.

5.7.1.4.1 Collating entire result sets

The first approach to getting results of a query back into R is the most convenient, but for large result sets, the entire dataset is loaded into memory and may take significant time if network connections are slow.

```
# for VERY large result sets, this may take
# quite a bit of time and/or memory. An
# alternative is to use s$chunk() to retrieve
# one batch of records at a time and process
# incrementally.
```

```

res = s$collate(limit = 1000)
head(res)
#> # A tibble: 6 x 85
#>   experiment_Insdc experiment_LastMet~ experiment_LastUpd~
#>   <lgl>           <dttm>           <dttm>
#> 1 TRUE            2018-04-10 11:28:04 2018-04-10 11:52:20
#> 2 TRUE            2018-05-06 06:12:59 2018-05-06 06:42:38
#> 3 TRUE            2013-08-28 07:50:02 2014-04-13 06:08:21
#> 4 TRUE            2018-04-10 11:28:00 2018-04-10 11:34:38
#> 5 TRUE            2016-12-07 17:16:11 2017-06-21 23:05:24
#> 6 TRUE            2016-10-21 15:03:02 2016-10-22 13:55:27
#> # ... with 82 more variables: experiment_Published <dttm>,
#> #   experiment_Received <dttm>, experiment_Status <chr>,
#> #   experiment_accession <chr>, experiment_alias <chr>,
#> #   experiment_center_name <chr>, experiment_identifiers <list>,
#> #   experiment_instrument_model <chr>,
#> #   experiment_library_construction_protocol <chr>,
#> #   experiment_library_layout <chr>, experiment_library_name <chr>,
#> #   experiment_library_selection <chr>, experiment_library_source <chr>,
#> #   experiment_library_strategy <chr>, experiment_platform <chr>,
#> #   experiment_title <chr>, run_FileDate <dbl>, run_FileMd5 <chr>,
#> #   run_FileSize <int>, run_Insdc <lgl>, run_LastMetaUpdate <dttm>,
#> #   run_LastUpdate <dttm>, run_Published <dttm>, run_Received <dttm>,
#> #   run_Status <chr>, run_accession <chr>, run_alias <chr>,
#> #   run_attributes <list>, run_bases <dbl>, run_center_name <chr>,
#> #   run_identifiers <list>, run_nreads <int>, run_reads <list>,
#> #   run_spot_length <int>, run_spots <int>, sample_BioSample <chr>,
#> #   sample_Insdc <lgl>, sample_LastMetaUpdate <dttm>,
#> #   sample_LastUpdate <dttm>, sample_Published <dttm>,
#> #   sample_Received <dttm>, sample_Status <chr>, sample_accession <chr>,
#> #   sample_alias <chr>, sample_attributes <list>,
#> #   sample_center_name <chr>, sample_identifiers <list>,
#> #   sample_organism <chr>, sample_taxon_id <int>, sample_title <chr>,
#> #   study_BioProject <chr>, study_Insdc <lgl>,
#> #   study_LastMetaUpdate <dttm>, study_LastUpdate <dttm>,
#> #   study_Published <dttm>, study_Received <dttm>, study_Status <chr>,
#> #   study_abstract <chr>, study_accession <chr>, study_alias <chr>,
#> #   study_attributes <list>, study_center_name <chr>,
#> #   study_description <chr>, study_identifiers <list>, study_title <chr>,
#> #   study_type <chr>, experiment_attributes <list>,
#> #   experiment_broker_name <chr>, experiment_design <chr>,
#> #   run_broker_name <chr>, run_center <chr>, sample_broker_name <chr>,
#> #   sample_description <chr>, study_broker_name <chr>,
#> #   run_file_addons <list>, experiment_library_layout_length <dbl>,
#> #   experiment_library_layout_sdev <chr>, sample_xrefs <list>,
#> #   experiment_xrefs <list>, sample_GEO <chr>, study_GEO <chr>,
#> #   study_xrefs <list>

```

Note that the scroller now reports that it has fetched (`s$fetched`) 1000 records.

To reuse a `Scroller`, we must reset it first.

```

s$reset()
s
#> <Scroller>

```

```
#>   Public:
#>     clone: function (deep = FALSE)
#>     collate: function (limit = Inf)
#>     count: active binding
#>     fetched: active binding
#>     has_next: function ()
#>     initialize: function (search, progress = interactive())
#>     reset: function ()
#>     yield: function ()
#>   Private:
#>     .count: 8886
#>     .fetched: 0
#>     .last: FALSE
#>     progress: FALSE
#>     scroll: 1m
#>     scroll_id: NULL
#>     search: Searcher, R6
```

5.7.1.4.2 Yielding chunks

The second approach is to iterate through results using the `yield` method. This approach allows the user to perform processing on chunks of data as they arrive in R.

```
j = 0
## fetch only 500 records, but
## `yield` will return NULL
## after ALL records have been fetched
while(s$fetched < 500) {
  res = s$yield()
  # do something interesting with `res` here if you like
  j = j + 1
  message(sprintf('total of %d fetched records, loop iteration # %d', s$fetched, j))
}
#> total of 100 fetched records, loop iteration # 1
#> total of 200 fetched records, loop iteration # 2
#> total of 300 fetched records, loop iteration # 3
#> total of 400 fetched records, loop iteration # 4
#> total of 500 fetched records, loop iteration # 5
```

The `Scroller` also has a `has_next()` method that will report TRUE if the result set has not been fully fetched. Using the `reset()` method will move the cursor back to the beginning of the result set.

5.7.2 Query syntax

5.7.2.1 Terms

A query is broken up into terms and operators. There are two types of terms: Single Terms and Phrases. A Single Term is a single word such as “test” or “hello”. A Phrase is a group of words surrounded by double quotes such as “hello dolly”. Multiple terms can be combined together with Boolean operators to form a more complex query (see below).

5.7.2.2 Fields

Queries support fielded data. When performing a search you can either specify a field, or use the default field. The field names and default field is implementation specific. You can search any field by typing the field name followed by a colon ":" and then the term you are looking for. As an example, let's assume a Lucene index contains two fields, title and abstract. If you want to find the document entitled "The Right Way" which contains the text "don't go this way" in the abstract, you can enter:

```
title:"The Right Way" AND abstract:go
```

or

Note: The field is only valid for the term that it directly precedes, so the query

```
title:Do it right
```

will only find "Do" in the title field. It will find "it" and "right" in any other fields.

5.7.2.3 Wildcard Searches

Lucene supports single and multiple character wildcard searches within single terms (not within phrase queries). To perform a single character wildcard search use the "?" symbol. To perform a multiple character wildcard search use the "*" symbol. The single character wildcard search looks for terms that match that with the single character replaced. For example, to search for "text" or "test" you can use the search:

```
te?t
```

Multiple character wildcard searches looks for 0 or more characters. For example, to search for test, tests or tester, you can use the search:

```
test*
```

You can also use the wildcard searches in the middle of a term.

```
te*t
```

Note: You cannot use a * or ? symbol as the first character of a search.

5.7.2.4 Fuzzy Searches

Lucene supports fuzzy searches based on the Levenshtein Distance, or Edit Distance algorithm. To do a fuzzy search use the tilde, "~", symbol at the end of a Single word Term. For example to search for a term similar in spelling to "roam" use the fuzzy search:

```
roam~
```

This search will find terms like foam and roams.

Starting with Lucene 1.9 an additional (optional) parameter can specify the required similarity. The value is between 0 and 1, with a value closer to 1 only terms with a higher similarity will be matched. For example:

```
roam~0.8
```

The default that is used if the parameter is not given is 0.5.

5.7.2.5 Proximity Searches

Lucene supports finding words are a within a specific distance away. To do a proximity search use the tilde, "~", symbol at the end of a Phrase. For example to search for a "apache" and "jakarta" within 10 words of each other in a document use the search:

```
"jakarta apache"~10
```

5.7.2.6 Range Searches

Range Queries allow one to match documents whose field(s) values are between the lower and upper bound specified by the Range Query. Range Queries can be inclusive or exclusive of the upper and lower bounds. Sorting is done lexicographically.

```
mod_date:[20020101 TO 20030101]
```

This will find documents whose mod_date fields have values between 20020101 and 20030101, inclusive. Note that Range Queries are not reserved for date fields. You could also use range queries with non-date fields:

```
title:{Aida TO Carmen}
```

This will find all documents whose titles are between Aida and Carmen, but not including Aida and Carmen. Inclusive range queries are denoted by square brackets. Exclusive range queries are denoted by curly brackets.

5.7.2.7 Boolean Operators

Boolean operators allow terms to be combined through logic operators. Lucene supports AND, “+”, OR, NOT and “-” as Boolean operators (Note: Boolean operators must be ALL CAPS).

5.7.2.7.1 OR

The OR operator is the default conjunction operator. This means that if there is no Boolean operator between two terms, the OR operator is used. The OR operator links two terms and finds a matching document if either of the terms exist in a document. This is equivalent to a union using sets. The symbol || can be used in place of the word OR.

To search for documents that contain either “jakarta apache” or just “jakarta” use the query:

```
"jakarta apache" jakarta
```

or

```
"jakarta apache" OR jakarta
```

The AND operator matches documents where both terms exist anywhere in the text of a single document. This is equivalent to an intersection using sets. The symbol && can be used in place of the word AND. To search for documents that contain “jakarta apache” and “Apache Lucene” use the query:

```
"jakarta apache" AND "Apache Lucene"
```

5.7.2.7.2 +

The “+” or required operator requires that the term after the “+” symbol exist somewhere in a the field of a single document. To search for documents that must contain “jakarta” and may contain “lucene” use the query:

```
+jakarta lucene
```

5.7.2.7.3 NOT

The NOT operator excludes documents that contain the term after NOT. This is equivalent to a difference using sets. The symbol ! can be used in place of the word NOT. To search for documents that contain “jakarta apache” but not “Apache Lucene” use the query:

```
"jakarta apache" NOT "Apache Lucene"
```

Note: The NOT operator cannot be used with just one term. For example, the following search will return no results:

```
NOT "jakarta apache"
```

5.7.2.7.4 -

The “-” or prohibit operator excludes documents that contain the term after the “-” symbol. To search for documents that contain “jakarta apache” but not “Apache Lucene” use the query:

```
"jakarta apache" -"Apache Lucene"
```

5.7.2.8 Grouping

Lucene supports using parentheses to group clauses to form sub queries. This can be very useful if you want to control the boolean logic for a query. To search for either “jakarta” or “apache” and “website” use the query:

```
(jakarta OR apache) AND website
```

This eliminates any confusion and makes sure you that website must exist and either term jakarta or apache may exist.

Lucene supports using parentheses to group multiple clauses to a single field.

To search for a title that contains both the word “return” and the phrase “pink panther” use the query:

```
title:(+return +"pink panther")
```

5.7.2.9 Escaping Special Characters

Lucene supports escaping special characters that are part of the query syntax. The current list special characters are

```
+ - && || ! ( ) { } [ ] ^ " ~ * ? : \
```

To escape these character use the \ before the character. For example to search for (1+1):2 use the query:

```
\(1\+1\)\:2
```

5.7.3 Using the raw API without R/Bioconductor

The SRAdBV2 is a client to a high-performance web-based API. As such, the web API is perfectly usable from either a simple test page, accessible here:

```
sra_browse_API()
```

The web-based API provides a useful interface for experiment with queries. It also returns URLs associated with the example queries, facilitating querying with other tools like curl or wget

The API is described using the OpenAPI standard, also known as Swagger. Tooling exists to quickly scaffold clients in any language (basically) based on the json available here:

```
sra_get_swagger_json_url()
#> [1] "https://api-omicididx.cancerdatasci.org/sra/1.0/swagger.json"
```

5.8 Accessing The Cancer Genome Atlas (TCGA)

We summarize two approaches to accessing TCGA data: 1. *TCGAbiolinks*: a. data access through *GenomicDataCommons* b. provides data both from the legacy Firehose pipeline used by the TCGA publications (alignments based on hg18 and hg19 builds²), and the GDC harmonized GRCh38 pipeline³. c. downloads files from the Genomic Data Commons, and provides conversion to `(Ranged)SummarizedExperiment` where possible 2. *curatedTCGAData*: a. data access through *ExperimentHub* b. provides data from the legacy Firehose pipeline⁴ c. provides individual assays as `(Ranged)SummarizedExperiment` and `RaggedExperiment`, integrates multiple assays within and across cancer types using `MultiAssayExperiment`

5.8.1 TCGAbiolinks

We demonstrate here generating a `RangedSummarizedExperiment` for RNA-seq data from adrenocortical carcinoma (ACC). For additional information and options, see the TCGAbiolinks vignettes⁵.

Load packages:

Search for matching data:

```
library(TCGAbiolinks)
library(SummarizedExperiment)
query <- GDCquery(project = "TCGA-ACC",
                   data.category = "Gene expression",
                   data.type = "Gene expression quantification",
                   platform = "Illumina HiSeq",
                   file.type = "normalized_results",
                   experimental.strategy = "RNA-Seq",
                   legacy = TRUE)
```

Download data and convert it to `RangedSummarizedExperiment`:

```
gdmdir <- file.path("Waldron_PublicData", "GDCdata")
GDCdownload(query, method = "api", files.per.chunk = 10,
            directory = gdmdir)
ACCse <- GDCprepare(query, directory = gdmdir)
ACCse
```

5.8.2 curatedTCGAData: Curated Data From The Cancer Genome Atlas as MultiAssayExperiment Objects

curatedTCGAData does not interface with the Genomic Data Commons, but downloads data from Bioconductor's *ExperimentHub*.

```
library(curatedTCGAData)
library(MultiAssayExperiment)
```

By default, the `curatedTCGAData()` function will only show available datasets, and not download anything. The arguments are shown here only for demonstration, the same result is obtained with no arguments:

```
curatedTCGAData(diseaseCode = "*", assays = "*")
#> Please see the list below for available cohorts and assays
```

²<https://confluence.broadinstitute.org/display/GDAC/FAQ#FAQ-Q%C2%A0Whatreferencegenomebuildareyouusing>

³<https://gdc.cancer.gov/about-data/data-harmonization-and-generation/genomic-data-harmonization-0>

⁴<https://confluence.broadinstitute.org/display/GDAC/FAQ#FAQ-Q%C2%A0Whatreferencegenomebuildareyouusing>

⁵https://bioconductor.org/packages/release/bioc/vignettes/TCGAbiolinks/inst/doc/download_prepare.html

```
#> Available Cancer codes:
#> ACC BLCA BRCA CESC CHOL COAD DLBC ESCA GBM HNSC KICH
#> KIRC KIRP LAML LGG LIHC LUAD LUSC MESO OV PAAD PCPG
#> PRAD READ SARC SKCM STAD TGCT THCA THYM UCEC UCS UVM
#> Available Data Types:
#> CNACGH CNASeq CNASNP CNVSNP GISTIC GISTICT
#> Methylation miRNAArray miRNASEqGene mRNAArray
#> Mutation RNASeq2GeneNorm RNASeqGene RPPAArray
```

Check potential files to be downloaded for adrenocortical carcinoma (ACC):

```
curatedTCGAData(diseaseCode = "ACC")
#> ACC_CNASNP
#> "ACC_CNASNP-20160128.rda"
#> ACC_CNVSNP
#> "ACC_CNVSNP-20160128.rda"
#> ACC_GISTIC_AllByGene
#> "ACC_GISTIC_AllByGene-20160128.rda"
#> ACC_GISTIC_ThresholdedByGene
#> "ACC_GISTIC_ThresholdedByGene-20160128.rda"
#> ACC_Methylation
#> "ACC_Methylation-20160128.rda"
#> ACC_miRNASEqGene
#> "ACC_miRNASEqGene-20160128.rda"
#> ACC_Mutation
#> "ACC_Mutation-20160128.rda"
#> ACC_RNASeq2GeneNorm
#> "ACC_RNASeq2GeneNorm-20160128.rda"
#> ACC_RPPAArray
#> "ACC_RPPAArray-20160128.rda"
```

Actually download the reverse phase protein array (RPPA) and RNA-seq data for ACC

```
ACCmae <- curatedTCGAData("ACC", c("RPPAArray", "RNASeq2GeneNorm"),
                           dry.run=FALSE)
ACCmae
#> A MultiAssayExperiment object of 2 listed
#> experiments with user-defined names and respective classes.
#> Containing an ExperimentList class object of length 2:
#> [1] ACC_RNASeq2GeneNorm-20160128: SummarizedExperiment with 20501 rows and 79 columns
#> [2] ACC_RPPAArray-20160128: SummarizedExperiment with 192 rows and 46 columns
#> Features:
#> experiments() - obtain the ExperimentList instance
#> colData() - the primary/phenotype DataFrame
#> sampleMap() - the sample availability DataFrame
#> `$, `[, `[[` - extract colData columns, subset, or experiment
#> *Format() - convert into a long or wide DataFrame
#> assays() - convert ExperimentList to a SimpleList of matrices
```

Note. Data will be downloaded the first time the above command is run; subsequent times it will be loaded from local cache.

This object contains 822 columns of clinical, pathological, specimen, and subtypes data in its `colData`, merged from all available data levels (1-4) of the Firehose pipeline:

```
dim(colData(ACCmae))
#> [1] 79 822
head(colnames(colData(ACCmae)))
#> [1] "patientID"           "years_to_birth"      "vital_status"
#> [4] "days_to_death"        "days_to_last_followup" "tumor_tissue_site"
```

See the *MultiAssayExperiment* vignette (Ramos et al. 2017) and the *Workflow for Multi-omics Analysis with MultiAssayExperiment* workshop for details on using this object.

5.8.2.1 Subtype information

Some cancer datasets contain associated subtype information within the clinical datasets provided. This subtype information is included in the metadata of `colData` of the `MultiAssayExperiment` object. To obtain these variable names, run the `metadata` function on the `colData` of the object such as:

```
head(metadata(colData(ACCmae)))[["subtypes"]]
#>             ACC_annotations    ACC_subtype
#> 1            Patient_ID       SAMPLE
#> 2 histological_subtypes   Histology
#> 3            mrna_subtypes   C1A/C1B
#> 4            mrna_subtypes   mRNA_K4
#> 5                  cimp     MethyLevel
#> 6 microrna_subtypes miRNA cluster
```

5.9 recount: Reproducible RNA-seq Analysis Using `recount2`

The `recount`(Collado-Torres et al. 2017) package provides uniformly processed `RangedSummarizedExperiment` objects at the gene, exon, or exon-exon junctions level, the raw counts, the phenotype metadata used, the urls to sample coverage bigWig files and mean coverage bigWig file, for every study available. The `RangedSummarizedExperiment` objects can be used for differential expression analysis. These are also accessible through a web interface.⁶

```
#> No methods found in package 'IRanges' for request: 'subset' when loading 'derfinder'
```

`recount` provides a search function:

```
library(recount)
project_info <- abstract_search('GSE32465')
```

It is not an ExperimentHub package, so downloading and serializing is slightly more involved in involves two steps: first, download the gene-level `RangedSummarizedExperiment` data:

```
download_study(project_info$project)
#> 2018-07-25 18:02:12 downloading file rse_gene.Rdata to SRP009615
```

followed by loading the data

```
load(file.path(project_info$project, 'rse_gene.Rdata'))
```

⁶<https://jhbiostatistics.shinyapps.io/recount/>

5.10 curated*Data packages for standardized cancer transcriptomes

There are focused databases of cancer microarray data for several cancer types, which can be useful for researchers of those cancer types or for methodological development: * *curatedOvarianData*(Ganzfried et al. 2013): Clinically Annotated Data for the Ovarian Cancer Transcriptome (data available with additional options through the *MetaGxOvarian* package). * *curatedBladderData*: Clinically Annotated Data for the Bladder Cancer Transcriptome * *curatedCRCDATA*: Clinically Annotated Data for the Colorectal Cancer Transcriptome

These provide data from the Gene Expression Omnibus and other sources, but use a formally vocabulary for clinicopathological data and use a common pipeline for preprocessing of microarray data (for Affymetrix, other for other platforms the processed data are provided as processed by original study authors), merging probesets, and mapping to gene symbols. The pipeline is described by Ganzfried et al. (2013).

5.11 Microbiome data

Bioconductor provides curated resources of microbiome data. Most microbiome data are generated either by targeted amplicon sequencing (usually of variable regions of the 16S ribosomal RNA gene) or by metagenomic shotgun sequencing (MGX). These two approaches are analyzed by different sequence analysis tools, but downstream statistical and ecological analysis can involve any of the following types of data: * taxonomic abundance at different levels of the taxonomic hierarchy * phylogenetic distances and the phylogenetic tree of life * metabolic potential of the microbiome * abundance of microbial genes and gene families

A review of types and properties of microbiome data is provided by (Morgan and Huttenhower 2012).

5.11.1 curatedMetagenomicData: Curated and processed metagenomic data through ExperimentHub

curatedMetagenomicData(Pasolli et al. 2017) provides 6 types of processed data for >30 publicly available whole-metagenome shotgun sequencing datasets (obtained from the Sequence Read Archive):

1. Species-level taxonomic profiles, expressed as relative abundance from kingdom to strain level
2. Presence of unique, clade-specific markers
3. Abundance of unique, clade-specific markers
4. Abundance of gene families
5. Metabolic pathway coverage
6. Metabolic pathway abundance

Types 1-3 are generated by MetaPhlAn2; 4-6 are generated by HUMAnN2.

Currently, *curatedMetagenomicData* provides:

- 6386 samples from 31 datasets, primarily of the human gut but including body sites profiled in the Human Microbiome Project
- Processed data from whole-metagenome shotgun metagenomics, with manually-curated metadata, as integrated and documented Bioconductor ExpressionSet objects
- ~80 fields of specimen metadata from original papers, supplementary files, and websites, with manual curation to standardize annotations
- Processing of data through the MetaPhlAn2 pipeline for taxonomic abundance, and HUMAnN2 pipeline for metabolic analysis
- These represent ~100TB of raw sequencing data, but the processed data provided are much smaller.

These datasets are documented in the reference manual.

This is an *ExperimentHub* package, and its main workhorse function is `curatedMetagenomicData()`:

The manually curated metadata for all available samples are provided in a single table `combined_metadata`:

```
library(curatedMetagenomicData)
?combined_metadata
View(data.frame(combined_metadata))
```

The main function provides a list of `ExpressionSet` objects:

```
oral <- c("BritoIL_2016.metaphlan_bugs_list.oralcavity",
        "Castro-NallarE_2015.metaphlan_bugs_list.oralcavity")
esl <- curatedMetagenomicData(oral, dryrun = FALSE)
#> Working on BritoIL_2016.metaphlan_bugs_list.oralcavity
#> snapshotDate(): 2018-07-17
#> see ?curatedMetagenomicData and browseVignettes('curatedMetagenomicData') for documentation
#> downloading 0 resources
#> loading from cache
#>     '/home/mramos//.ExperimentHub/1179'
#> Working on Castro-NallarE_2015.metaphlan_bugs_list.oralcavity
#> snapshotDate(): 2018-07-17
#> see ?curatedMetagenomicData and browseVignettes('curatedMetagenomicData') for documentation
#> downloading 0 resources
#> loading from cache
#>     '/home/mramos//.ExperimentHub/391'

esl
#> List of length 2
#> names(2): BritoIL_2016.metaphlan_bugs_list.oralcavity ...
```

These `ExpressionSet` objects can also be converted to `phyloseq` object for ecological analysis and differential abundance analysis using the `DESeq2` package, using the `ExpressionSet2phyloseq()` function:

```
ExpressionSet2phyloseq( esl[[1]], phylogenetictree = TRUE)
#> Loading required namespace: phyloseq
#> phyloseq-class experiment-level object
#> otu_table()    OTU Table:      [ 535 taxa and 140 samples ]
#> sample_data()  Sample Data:    [ 140 samples by 17 sample variables ]
#> tax_table()    Taxonomy Table: [ 535 taxa by 8 taxonomic ranks ]
#> phy_tree()     Phylogenetic Tree: [ 535 tips and 534 internal nodes ]
```

See the documentation of `phyloseq` for more on ecological and differential abundance analysis of the microbiome.

5.11.2 HMP16SData: 16S rRNA Sequencing Data from the Human Microbiome Project

```
suppressPackageStartupMessages(library(HMP16SData))
#> snapshotDate(): 2018-07-17
```

`HMP16SData`(Schiffer et al. 2018) is a Bioconductor `ExperimentData` package of the Human Microbiome Project (HMP) 16S rRNA sequencing data. Taxonomic count data files are provided as downloaded from the HMP Data Analysis and Coordination Center from its QIIME pipeline. Processed data is provided as `SummarizedExperiment` class objects via *ExperimentHub*. Like other *ExperimentHub*-based packages, a convenience function does downloading, automatic local caching, and serializing of a Bioconductor data

class. This returns taxonomic counts from the V1-3 variable region of the 16S rRNA gene, along with the unrestricted participant data and phylogenetic tree.

```
V13()
#> snapshotDate(): 2018-07-17
#> see ?HMP16SData and browseVignettes('HMP16SData') for documentation
#> downloading 0 resources
#> loading from cache
#> '/home/mramos//.ExperimentHub/1117'
#> class: SummarizedExperiment
#> dim: 43140 2898
#> metadata(2): experimentData phylogeneticTree
#> assays(1): 16SrRNA
#> rownames(43140): OTU_97.1 OTU_97.10 ... OTU_97.9997 OTU_97.9999
#> rowData names(7): CONSENSUS_LINEAGE SUPERKINGDOM ... FAMILY GENUS
#> colnames(2898): 700013549 700014386 ... 700114963 700114965
#> colData names(7): RSID VISITNO ... HMP_BODY_SUBSITE SRS_SAMPLE_ID
```

This can also be converted to *phyloseq* for ecological and differential abundance analysis; see the *HMP16SData* vignette for details.

5.12 Pharmacogenomics

Pharmacogenomics holds great promise for the development of biomarkers of drug response and the design of new therapeutic options, which are key challenges in precision medicine. However, such data are scattered and lack standards for efficient access and analysis, consequently preventing the realization of the full potential of pharmacogenomics. To address these issues, we implemented *PharmacoGx*, an easy-to-use, open source package for integrative analysis of multiple pharmacogenomic datasets. ~PharmacoGx‘ provides a unified framework for downloading and analyzing large pharmacogenomic datasets which are extensively curated to ensure maximum overlap and consistency.

Examples of *PharmacoGx* usage in biomedical research can be found in the following publications: * Smirnov et al. *PharmacoGx: an R package for analysis of large pharmacogenomic datasets.*" *Bioinformatics* (2015): 1244-1246. * Safikhani et al., *Assessment of pharmacogenomic agreement*, *F1000 Research* (2017) * Safikhani et al., *Revisiting inconsistency in large pharmacogenomic studies*, *F1000 Research* (2017) * Yao et al., *Tissue specificity of in vitro drug sensitivity*, *JAMIA* (2017) * Safikhani et al., *Gene isoforms as expression-based biomarkers predictive of drug response in vitro*, *Nature Communications* (2017) * El-Hachem et al., *Integrative Cancer Pharmacogenomics to Infer Large-Scale Drug Taxonomy*, *Cancer Research* (2017)

5.12.1 Getting started

Let us first load the *PharmacoGx* library.

```
library(PharmacoGx)
#> Warning in fun(libname, pkgname): couldn't connect to display ":0"
```

We can now access large-scale preclinical pharmacogenomic datasets that have been fully curated for ease of use.

5.12.2 Overview of *PharmacoGx* datasets (*PharmacoSets*)

To efficiently store and analyze large pharmacogenomic datasets, we developed the *PharmacoSet* class (also referred to as *PSet*), which acts as a data container storing pharmacological and molecular data along with

experimental metadata (detailed structure provided in Supplementary materials). This class enables efficient implementation of curated annotations for cell lines, drug compounds and molecular features, which facilitates comparisons between different datasets stored as `PharmacoSet` objects.

We have made the `PharmacoSet` objects of the curated datasets available for download using functions provided in the package. A table of available `PharmacoSet` objects can be obtained by using the `availablePSets` function. Any of the `PharmacoSets` in the table can then be downloaded by calling `downloadPSet`, which saves the datasets into a directory of the users choice, and returns the data into the R session.

Structure of the `PharmacoSet` class

To get a list of all the available `PharmacoSets` in `PharmacoGx`, we can use the `availablePSets` function, which returns a table providing key information for each dataset.

```
availablePSets(saveDir=file.path(".", "Waldron_PublicData"))
#>          PSet.Name Dataset.Type Available.Molecular.Profiles
#> CCLE_2013      CCLE_2013  sensitivity             rna/mutation
#> CCLE           CCLE      sensitivity     rna/rnaseq/mutation/cnv
#> GDSC_2013      GDSC_2013  sensitivity             rna/mutation
#> GDSC           GDSC      sensitivity rna/rna2/mutation/fusion/cnv
#> GDSC1000       GDSC1000  sensitivity             rna
#> gCSI            gCSI     sensitivity             rnaseq/cnv
#> FIMM            FIMM    sensitivity
#> CTRPv2          CTRPv2   sensitivity
#> CMAP            CMAP    perturbation
#> L1000_compounds L1000_compounds perturbation
#> L1000_genetic   L1000_genetic perturbation
#>                  Date.Updated
#> CCLE_2013      Tue Sep 15 18:50:07 2015
#> CCLE           Thu Dec 10 18:17:14 2015
#> GDSC_2013      Mon Oct  5 16:07:54 2015
#> GDSC           Wed Dec 30 10:44:21 2015
#> GDSC1000       Thu Aug 25 11:13:00 2016
#> gCSI            Mon Jun 13 18:50:12 2016
#> FIMM            Mon Oct  3 17:14:00 2016
#> CTRPv2          Thu Aug 25 11:15:00 2016
#> CMAP            Mon Sep 21 02:38:45 2015
#> L1000_compounds Mon Jan 25 12:51:00 2016
#> L1000_genetic   Mon Jan 25 12:51:00 2016
#>
#> CCLE_2013      https://www.ncbi.nlm.nih.gov/bhklab/sites/default/files/downloads/CCLE_Nature2013.RData
#> CCLE           https://www.ncbi.nlm.nih.gov/bhklab/sites/default/files/downloads/CCLE.RData
#> GDSC_2013      https://www.ncbi.nlm.nih.gov/bhklab/sites/default/files/downloads/CGP_Nature2013.RData
#> GDSC           https://www.ncbi.nlm.nih.gov/bhklab/sites/default/files/downloads/GDSC.RData
#> GDSC1000       https://www.ncbi.nlm.nih.gov/bhklab/sites/default/files/downloads/GDSC1000.RData
#> gCSI            https://www.ncbi.nlm.nih.gov/bhklab/sites/default/files/downloads/gCSI.RData
#> FIMM            https://www.ncbi.nlm.nih.gov/bhklab/sites/default/files/downloads/FIMM.RData
#> CTRPv2          https://www.ncbi.nlm.nih.gov/bhklab/sites/default/files/downloads/CTRPv2.RData
#> CMAP            https://www.ncbi.nlm.nih.gov/bhklab/sites/default/files/downloads/CMAP.RData
#> L1000_compounds https://www.ncbi.nlm.nih.gov/bhklab/sites/default/files/downloads/L1000_compounds.RData
#> L1000_genetic   https://www.ncbi.nlm.nih.gov/bhklab/sites/default/files/downloads/L1000_genetic.RData
```

There are currently 11 datasets available in `PharmacoGx`, including 8 sensitivity datasets and 3 perturbation datasets (see below).

5.12.2.1 Drug Sensitivity Datasets

Drug sensitivity datasets refer to pharmacogenomic data where cancer cells are molecularly profiled at baseline (before drug treatment), and the effect of drug treatment on cell viability is measured using a pharmacological assay (e.g., Cell Titer-Glo). These datasets can be used for biomarker discovery by correlating the molecular features of cancer cells to their response to drugs of interest.

Schematic view of the drug sensitivity datasets.

```
psets <- availablePSets(saveDir=file.path(".", "Waldron_PublicData"))
psets[psets[ , "Dataset.Type"] == "sensitivity", ]
#>          PSet.Name Dataset.Type Available.Molecular.Profiles
#> CCLE_2013 CCLE_2013 sensitivity           rna/mutation
#> CCLE       CCLE   sensitivity           rna/rnaseq/mutation/cnv
#> GDSC_2013 GDSC_2013 sensitivity           rna/mutation
#> GDSC       GDSC   sensitivity rna/rna2/mutation/fusion/cnv
#> GDSC1000  GDSC1000 sensitivity           rna
#> gCSI        gCSI   sensitivity           rnaseq/cnv
#> FIMM        FIMM   sensitivity           rnaseq/cnv
#> CTRPv2     CTRPv2  sensitivity           Date.Updated
#> CCLE_2013 Tue Sep 15 18:50:07 2015
#> CCLE       Thu Dec 10 18:17:14 2015
#> GDSC_2013 Mon Oct  5 16:07:54 2015
#> GDSC       Wed Dec 30 10:44:21 2015
#> GDSC1000  Thu Aug 25 11:13:00 2016
#> gCSI        Mon Jun 13 18:50:12 2016
#> FIMM        Mon Oct  3 17:14:00 2016
#> CTRPv2     Thu Aug 25 11:15:00 2016
#>
#> CCLE_2013 https://www.pmgcgenomics.ca/bhklab/sites/default/files/downloads/CCLE_Nature2013.RData
#> CCLE       https://www.pmgcgenomics.ca/bhklab/sites/default/files/downloads/CCLE.RData
#> GDSC_2013  https://www.pmgcgenomics.ca/bhklab/sites/default/files/downloads/GDSC_Nature2013.RData
#> GDSC       https://www.pmgcgenomics.ca/bhklab/sites/default/files/downloads/GDSC.RData
#> GDSC1000   https://www.pmgcgenomics.ca/bhklab/sites/default/files/downloads/GDSC1000.RData
#> gCSI        https://www.pmgcgenomics.ca/bhklab/sites/default/files/downloads/gCSI.RData
#> FIMM        https://www.pmgcgenomics.ca/bhklab/sites/default/files/downloads/FIMM.RData
#> CTRPv2     https://www.pmgcgenomics.ca/bhklab/sites/default/files/downloads/CTRPv2.RData
```

Notably, the Genomics of Drug Sensitivity in Cancer GDSC and the Cancer Cell Line Encyclopedia CCLE are large drug sensitivity datasets published in seminal studies in Nature, Garnett et al., <https://www.nature.com/articles/nature11005>, Nature (2012) and Barretina et al., The Cancer Cell Line Encyclopedia enables predictive modelling of anticancer drug sensitivity, Nature (2012), respectively.

5.12.2.2 Drug Perturbation Datasets

Drug perturbation datasets refer to pharmacogenomic data where gene expression profiles are measured before and after short-term (e.g., 6h) drug treatment to identify genes that are up- and down-regulated due to the drug treatment. These datasets can be used to classify drug (drug taxonomy), infer their mechanism of action, or find drugs with similar effects (drug repurposing).

Schematic view of drug perturbation datasets

```
psets <- availablePSets(saveDir=file.path(".", "Waldron_PublicData"))
psets[psets[ , "Dataset.Type"] == "perturbation", ]
```

```

#>          PSet.Name Dataset.Type Available.Molecular.Profiles
#> CMAP           CMAP perturbation                      rna
#> L1000_compounds L1000_compounds perturbation        rna
#> L1000_genetic   L1000_genetic perturbation         rna
#>          Date.Updated
#> CMAP           Mon Sep 21 02:38:45 2015
#> L1000_compounds Mon Jan 25 12:51:00 2016
#> L1000_genetic   Mon Jan 25 12:51:00 2016
#>
#> CMAP           https://www.pmgeneomics.ca/bhklab/sites/default/files/downloads/CMAP.RData
#> L1000_compounds https://www.pmgeneomics.ca/bhklab/sites/default/files/downloads/L1000_compounds.RData
#> L1000_genetic   https://www.pmgeneomics.ca/bhklab/sites/default/files/downloads/L1000_genetic.RData

```

Large drug perturbation data have been generated within the Connectivity Map Project CAMP, with CMAPv2 and CMAPv3 available from *PharmacoGx*, published in Lamb et al., The Connectivity Map: Using Gene-Expression Signatures to Connect Small Molecules, Genes, and Disease, *Science* (2006) and Subramanian et al., A Next Generation Connectivity Map: L1000 Platform and the First 1,000,000 Profiles, *Cell* (2017), respectively.

5.12.3 Exploring drug sensitivity datasets

The Biomarker discovery from large pharmacogenomics datasets workshop demonstrates analyses of *PharmacoGx* data.

5.13 Bibliography

Chapter 6

200: RNA-seq analysis is easy as 1-2-3 with limma, Glimma and edgeR

6.1 Instructor name and contact information

- Charity Law (law@wehi.edu.au)

6.2 Workshop Description

In this instructor-led live demo, we analyse RNA-sequencing data from the mouse mammary gland, demonstrating use of the popular **edgeR** package to import, organise, filter and normalise the data, followed by the **limma** package with its voom method, linear modelling and empirical Bayes moderation to assess differential expression and graphical representations. This pipeline is further enhanced by the **Glimma** package which enables interactive exploration of the results so that individual samples and genes can be examined by the user. The complete analysis offered by these three packages highlights the ease with which researchers can turn the raw counts from an RNA-sequencing experiment into biological insights using Bioconductor. The complete workflow is available at <http://master.bioconductor.org/packages/release/workflows/html/RNAseq123.html>.

6.2.1 Pre-requisites

- Basic knowledge of RNA-sequencing
- Basic knowledge of R syntax, R object classes and object manipulation

6.2.2 Workshop Participation

Participants can watch the live demo, or may prefer to follow the demonstration by bringing their laptops along. To follow the analysis on their own laptops, participants need to install the *RNAseq123* workflow by running

```
source("https://bioconductor.org/biocLite.R")
biocLite("RNAseq123")
```

in R. The relevant sequencing data should also be download in advance.

```

url <- "https://www.ncbi.nlm.nih.gov/geo/download/?acc=GSE63310&format=file"
utils::download.file(url, destfile="GSE63310_RAW.tar", mode="wb")
utils::untar("GSE63310_RAW.tar", exdir = ".")
files <- c("GSM1545535_10_6_5_11.txt", "GSM1545536_9_6_5_11.txt", "GSM1545538_purep53.txt",
"GSM1545539_JMS8-2.txt", "GSM1545540_JMS8-3.txt", "GSM1545541_JMS8-4.txt",
"GSM1545542_JMS8-5.txt", "GSM1545544_JMS9-P7c.txt", "GSM1545545_JMS9-P8c.txt")
for(i in paste(files, ".gz", sep=""))
  R.utils::gunzip(i, overwrite=TRUE)

```

Due to time restraints, extra help regarding R package installation and coding errors will not be addressed during the workshop.

6.2.3 *R / Bioconductor* packages used

Bioconductor: limma, Glimma, edgeR, Mus.musculus
CRAN: RColorBrewer, gplots

6.2.4 Time outline

Activity	Time
Introduction	5mins
Data packaging	10mins
Data pre-processing	15mins
Differential expression analysis	30mins

6.3 Workshop goals and objectives

The key steps to RNA-seq data analysis are described in this workshop with basic statistical theory of methods used. The goal is to allow beginner-analysts of RNA-seq data to become familiar with each of the steps involved, as well as completing a standard analysis pipeline from start to finish.

6.3.1 Learning goals

- learn how to analyse RNA-seq data
- identify methods for pre-processing data
- understand linear models used in differential expression analysis
- examine plots for data exploration and result representation

6.3.2 Learning objectives

- read in count data and format as a DGEList-object
- annotate Entrez gene identifiers with gene information
- filter out lowly expressed genes
- normalise gene expression values
- unsupervised clustering of samples (standard and interactive plots)
- linear modelling for comparisons of interest
- remove heteroscedascity
- examine the number of differentially expressed genes

- mean-difference plots (standard and interactive plots)
- heatmaps

6.4 Introduction

RNA-sequencing (RNA-seq) has become the primary technology used for gene expression profiling, with the genome-wide detection of differentially expressed genes between two or more conditions of interest one of the most commonly asked questions by researchers. The **edgeR** and **limma** packages available from the Bioconductor project offer a well-developed suite of statistical methods for dealing with this question for RNA-seq data.

In this article, we describe an **edgeR - limma** workflow for analysing RNA-seq data that takes gene-level counts as its input, and moves through pre-processing and exploratory data analysis before obtaining lists of differentially expressed (DE) genes and gene signatures. This analysis is enhanced through the use of interactive graphics from the **Glimma** package, that allows for a more detailed exploration of the data at both the sample and gene-level than is possible using static **R** plots.

The experiment analysed in this workflow is from Sheridan *et al.* (2015) and consists of three cell populations (basal, luminal progenitor (LP) and mature luminal (ML)) sorted from the mammary glands of female virgin mice, each profiled in triplicate. RNA samples were sequenced across three batches on an Illumina HiSeq 2000 to obtain 100 base-pair single-end reads. The analysis outlined in this article assumes that reads obtained from an RNA-seq experiment have been aligned to an appropriate reference genome and summarised into counts associated with gene-specific regions. In this instance, reads were aligned to the mouse reference genome (mm10) using the **R** based pipeline available in the **Rsubread** package (specifically the **align** function followed by **featureCounts** for gene-level summarisation based on the in-built *mm10* RefSeq-based annotation).

Count data for these samples can be downloaded from the Gene Expression Omnibus (GEO) <http://www.ncbi.nlm.nih.gov/geo/> using GEO Series accession number GSE63310. Further information on experimental design and sample preparation is also available from GEO under this accession number.

```
suppressPackageStartupMessages({
  library(limma)
  library(Glimma)
  library(edgeR)
  library(Mus.musculus)
})
```

6.5 Data packaging

```
dir.create("Law_RNAseq123")
#> Warning in dir.create("Law_RNAseq123"): 'Law_RNAseq123' already exists
setwd("Law_RNAseq123")

url <- "https://www.ncbi.nlm.nih.gov/geo/download/?acc=GSE63310&format=file"
utils::download.file(url, destfile="GSE63310_RAW.tar", mode="wb")
utils::untar("GSE63310_RAW.tar", exdir = ".")

files <- c("GSM1545535_10_6_5_11.txt", "GSM1545536_9_6_5_11.txt",
          "GSM1545538_purep53.txt", "GSM1545539_JMS8-2.txt", "GSM1545540_JMS8-3.txt",
          "GSM1545541_JMS8-4.txt", "GSM1545542_JMS8-5.txt", "GSM1545544_JMS9-P7c.txt",
          "GSM1545545_JMS9-P8c.txt")
```

```
for(i in paste(files, ".gz", sep=""))
  R.utils::gunzip(i, overwrite=TRUE)
```

6.5.1 Reading in count-data

To get started with this analysis, download the file *GSE63310_RAW.tar* available online from <https://www.ncbi.nlm.nih.gov/geo/download/?acc=GSE63310&format=file>, and extract the relevant files from this archive. Each of these text files contains the raw gene-level counts for a given sample. Note that our analysis only includes the basal, LP and ML samples from this experiment (see associated file names below).

```
read.delim(file.path("Law_RNAseq123", files[1]), nrow=5)
#>   EntrezID GeneLength Count
#> 1    497097      3634     1
#> 2  100503874     3259     0
#> 3  100038431     1634     0
#> 4    19888       9747     0
#> 5    20671       3130     1
```

Whilst each of the nine text files can be read into **R** separately and combined into a matrix of counts, **edgeR** offers a convenient way to do this in one step using the **readDGE** function. The resulting DGEList-object contains a matrix of counts with 27,179 rows associated with unique Entrez gene identifiers (IDs) and nine columns associated with the individual samples in the experiment.

```
x <- readDGE(file.path("Law_RNAseq123", files), columns=c(1,3))
class(x)
#> [1] "DGEList"
#> attr("package")
#> [1] "edgeR"
dim(x)
#> [1] 27179     9
```

If the counts from all samples were stored in a single file, the data can be read into **R** and then converted into a DGEList-object using the **DGEList** function.

6.5.2 Organising sample information

For downstream analysis, sample-level information related to the experimental design needs to be associated with the columns of the counts matrix. This should include experimental variables, both biological and technical, that could have an effect on expression levels. Examples include cell type (basal, LP and ML in this experiment), genotype (wild-type, knock-out), phenotype (disease status, sex, age), sample treatment (drug, control) and batch information (date experiment was performed if samples were collected and analysed at distinct time points) to name just a few.

Our DGEList-object contains a **samples** data frame that stores both cell type (or **group**) and batch (sequencing **lane**) information, each of which consists of three distinct levels. Note that within **x\$samples**, library sizes are automatically calculated for each sample and normalisation factors are set to 1. For simplicity, we remove the GEO sample IDs (GSM*) from the column names of our DGEList-object **x**.

```
samplenames <- substring(colnames(x), 12, nchar(colnames(x)))
samplenames
#> [1] "23/GSM1545535_10_6_5_11" "23/GSM1545536_9_6_5_11"
#> [3] "23/GSM1545538_purep53"    "23/GSM1545539_JMS8-2"
#> [5] "23/GSM1545540_JMS8-3"    "23/GSM1545541_JMS8-4"
#> [7] "23/GSM1545542_JMS8-5"    "23/GSM1545544_JMS9-P7c"
```

```
#> [9] "23/GSM1545545_JMS9-P8c"
colnames(x) <- sampenames
group <- as.factor(c("LP", "ML", "Basal", "Basal", "ML", "LP",
                     "Basal", "ML", "LP"))
x$samples$group <- group
lane <- as.factor(rep(c("L004", "L006", "L008"), c(3, 4, 2)))
x$samples$lane <- lane
x$samples
#>                                         files group
#> 23/GSM1545535_10_6_5_11 Law_RNAseq123/GSM1545535_10_6_5_11.txt LP
#> 23/GSM1545536_9_6_5_11 Law_RNAseq123/GSM1545536_9_6_5_11.txt ML
#> 23/GSM1545538_purep53 Law_RNAseq123/GSM1545538_purep53.txt Basal
#> 23/GSM1545539_JMS8-2 Law_RNAseq123/GSM1545539_JMS8-2.txt Basal
#> 23/GSM1545540_JMS8-3 Law_RNAseq123/GSM1545540_JMS8-3.txt ML
#> 23/GSM1545541_JMS8-4 Law_RNAseq123/GSM1545541_JMS8-4.txt LP
#> 23/GSM1545542_JMS8-5 Law_RNAseq123/GSM1545542_JMS8-5.txt Basal
#> 23/GSM1545544_JMS9-P7c Law_RNAseq123/GSM1545544_JMS9-P7c.txt ML
#> 23/GSM1545545_JMS9-P8c Law_RNAseq123/GSM1545545_JMS9-P8c.txt LP
#> lib.size norm.factors lane
#> 23/GSM1545535_10_6_5_11 32863052 1 L004
#> 23/GSM1545536_9_6_5_11 35335491 1 L004
#> 23/GSM1545538_purep53 57160817 1 L004
#> 23/GSM1545539_JMS8-2 51368625 1 L006
#> 23/GSM1545540_JMS8-3 75795034 1 L006
#> 23/GSM1545541_JMS8-4 60517657 1 L006
#> 23/GSM1545542_JMS8-5 55086324 1 L006
#> 23/GSM1545544_JMS9-P7c 21311068 1 L008
#> 23/GSM1545545_JMS9-P8c 19958838 1 L008
```

6.5.3 Organising gene annotations

A second data frame named **genes** in the DGEList-object is used to store gene-level information associated with rows of the counts matrix. This information can be retrieved using organism specific packages such as **Mus.musculus** for mouse (or **Homo.sapiens** for human) or the **biomaRt** package which interfaces the Ensembl genome databases in order to perform gene annotation.

The type of information that can be retrieved includes gene symbols, gene names, chromosome names and locations, Entrez gene IDs, Refseq gene IDs and Ensembl gene IDs to name just a few. **biomaRt** primarily works off Ensembl gene IDs, whereas **Mus.musculus** packages information from various sources and allows users to choose between many different gene IDs as the key.

The Entrez gene IDs available in our dataset were annotated using the **Mus.musculus** package to retrieve associated gene symbols and chromosome information.

```
geneid <- rownames(x)
genes <- select(Mus.musculus, keys=geneid, columns=c("SYMBOL", "TXCHROM"),
                 keytype="ENTREZID")
head(genes)
#>   ENTREZID SYMBOL TXCHROM
#> 1    497097  Xkr4    chr1
#> 2 100503874  Gm19938    <NA>
#> 3 100038431  Gm10568    <NA>
#> 4    19888    Rp1    chr1
```

```
#> 5      20671   Sox17    chr1
#> 6      27395   Mrpl15   chr1
```

As with any gene ID, Entrez gene IDs may not map one-to-one to the gene information of interest. It is important to check for duplicated gene IDs and to understand the source of duplication before resolving them. Our gene annotation contains 28 genes that map to multiple chromosomes (e.g. gene Gm1987 is associated with *chr4* and *chr4_JH584294_random* and microRNA Mir5098 is associated with *chr2*, *chr5*, *chr8*, *chr11* and *chr17*). To resolve duplicate gene IDs one could combine all chromosome information from the multi-mapped genes, such that gene Gm1987 would be assigned to *chr4 and chr4_JH584294_random*, or select one of the chromosomes to represent the gene with duplicate annotation. For simplicity we do the latter, keeping only the first occurrence of each gene ID.

```
genes <- genes[!duplicated(genes$ENTREZID),]
```

In this example, the gene order is the same in both the annotation and the data object. If this is not the case due to missing and/or rearranged gene IDs, the `match` function can be used to order genes correctly. The data frame of gene annotations is then added to the data object and neatly packaged in a `DGEList`-object containing raw count data with associated sample information and gene annotations.

```
x$genes <- genes
x
#> An object of class "DGEList"
#> $samples
#>
#> 23/GSM1545535_10_6_5_11 Law_RNAseq123/GSM1545535_10_6_5_11.txt LP
#> 23/GSM1545536_9_6_5_11 Law_RNAseq123/GSM1545536_9_6_5_11.txt ML
#> 23/GSM1545538_purep53 Law_RNAseq123/GSM1545538_purep53.txt Basal
#> 23/GSM1545539_JMS8-2 Law_RNAseq123/GSM1545539_JMS8-2.txt Basal
#> 23/GSM1545540_JMS8-3 Law_RNAseq123/GSM1545540_JMS8-3.txt ML
#> 23/GSM1545541_JMS8-4 Law_RNAseq123/GSM1545541_JMS8-4.txt LP
#> 23/GSM1545542_JMS8-5 Law_RNAseq123/GSM1545542_JMS8-5.txt Basal
#> 23/GSM1545544_JMS9-P7c Law_RNAseq123/GSM1545544_JMS9-P7c.txt ML
#> 23/GSM1545545_JMS9-P8c Law_RNAseq123/GSM1545545_JMS9-P8c.txt LP
#>
#> lib.size norm.factors lane
#> 23/GSM1545535_10_6_5_11 32863052 1 L004
#> 23/GSM1545536_9_6_5_11 35335491 1 L004
#> 23/GSM1545538_purep53 57160817 1 L004
#> 23/GSM1545539_JMS8-2 51368625 1 L006
#> 23/GSM1545540_JMS8-3 75795034 1 L006
#> 23/GSM1545541_JMS8-4 60517657 1 L006
#> 23/GSM1545542_JMS8-5 55086324 1 L006
#> 23/GSM1545544_JMS9-P7c 21311068 1 L008
#> 23/GSM1545545_JMS9-P8c 19958838 1 L008
#>
#> $counts
#>           Samples
#> Tags          23/GSM1545535_10_6_5_11 23/GSM1545536_9_6_5_11
#> 497097          1                  2
#> 100503874        0                  0
#> 100038431        0                  0
#> 19888            0                  1
#> 20671            1                  1
#>
#>           Samples
#> Tags          23/GSM1545538_purep53 23/GSM1545539_JMS8-2
#> 497097          342                 526
```

```

#> 100503874      5      6
#> 100038431      0      0
#> 19888          0      0
#> 20671          76     40
#>           Samples
#> Tags      23/GSM1545540_JMS8-3 23/GSM1545541_JMS8-4 23/GSM1545542_JMS8-5
#> 497097        3      3      535
#> 100503874      0      0      5
#> 100038431      0      0      1
#> 19888          17     2      0
#> 20671          33     14     98
#>           Samples
#> Tags      23/GSM1545544_JMS9-P7c 23/GSM1545545_JMS9-P8c
#> 497097        2      0
#> 100503874      0      0
#> 100038431      0      0
#> 19888          1      0
#> 20671          18     8
#> 27174 more rows ...
#>
#> $genes
#>   ENTREZID SYMBOL TXCHROM
#> 1 497097  Xkr4  chr1
#> 2 100503874 Gm19938 <NA>
#> 3 100038431 Gm10568 <NA>
#> 4 19888    Rp1   chr1
#> 5 20671    Sox17  chr1
#> 27174 more rows ...

```

6.6 Data pre-processing

6.6.1 Transformations from the raw-scale

For differential expression and related analyses, gene expression is rarely considered at the level of raw counts since libraries sequenced at a greater depth will result in higher counts. Rather, it is common practice to transform raw counts onto a scale that accounts for such library size differences. Popular transformations include counts per million (CPM), log2-counts per million (log-CPM), reads per kilobase of transcript per million (RPKM), and fragments per kilobase of transcript per million (FPKM).

In our analyses, CPM and log-CPM transformations are used regularly although they do not account for feature length differences which RPKM and FPKM values do. Whilst RPKM and FPKM values can just as well be used, CPM and log-CPM values can be calculated using a counts matrix alone and will suffice for the type of comparisons we are interested in. Assuming that there are no differences in isoform usage between conditions, differential expression analyses look at gene expression changes between conditions rather than comparing expression across multiple genes or drawing conclusions on absolute levels of expression. In other words, gene lengths remain constant for comparisons of interest and any observed differences are a result of changes in condition rather than changes in gene length.

Here raw counts are converted to CPM and log-CPM values using the `cpm` function in `edgeR`, where log-transformations use a prior count of 0.25 to avoid taking the log of zero. RPKM values are just as easily calculated as CPM values using the `rpkpm` function in `edgeR` if gene lengths are available.

```
cpm <- cpm(x)
lcpm <- cpm(x, log=TRUE)
```

6.6.2 Removing genes that are lowly expressed

All datasets will include a mix of genes that are expressed and those that are not expressed. Whilst it is of interest to examine genes that are expressed in one condition but not in another, some genes are unexpressed throughout all samples. In fact, 19% of genes in this dataset have zero counts across all nine samples.

```
table(rowSums(x$counts==0)==9)
#>
#> FALSE TRUE
#> 22026 5153
```

Genes that are not expressed at a biologically meaningful level in any condition should be discarded to reduce the subset of genes to those that are of interest, and to reduce the number of tests carried out downstream when looking at differential expression. Upon examination of log-CPM values, it can be seen that a large proportion of genes within each sample is unexpressed or lowly-expressed (shown in panel A of the next figure). Using a nominal CPM value of 1 (which is equivalent to a log-CPM value of 0) genes are deemed to be expressed if their expression is above this threshold, and unexpressed otherwise. Genes must be expressed in at least one group (or in at least three samples across the entire experiment) to be kept for downstream analysis.

Although any sensible value can be used as the expression cutoff, typically a CPM value of 1 is used in our analyses as it separates expressed genes from unexpressed genes well for most datasets. Here, a CPM value of 1 means that a gene is *expressed* if it has at least 20 counts in the sample with the lowest sequencing depth (JMS9-P8c, library size approx. 20 million) or at least 76 counts in the sample with the greatest sequencing depth (JMS8-3, library size approx. 76 million). If sequence reads are summarised by exons rather than genes and/or experiments have low sequencing depth, a lower CPM cutoff may be considered.

```
keep.exprs <- rowSums(cpm>1)>=3
x <- x[keep.exprs,, keep.lib.sizes=FALSE]
dim(x)
#> [1] 14165 9
```

Using this criterion, the number of genes is reduced to approximately half the number that we started with (14,165 genes, panel B of the next figure). Note that subsetting the entire DGEList-object removes both the counts as well as the associated gene information. Code to produce the figure is given below.

```
library(RColorBrewer)
nsamples <- ncol(x)
col <- brewer.pal(nsamples, "Paired")
par(mfrow=c(1,2))
plot(density(lcpm[,1]), col=col[1], lwd=2, ylim=c(0,0.21), las=2,
     main="", xlab="")
title(main="A. Raw data", xlab="Log-cpm")
abline(v=0, lty=3)
for (i in 2:nsamples){
  den <- density(lcpm[,i])
  lines(den$x, den$y, col=col[i], lwd=2)
}
legend("topright", samplenames, text.col=col, bty="n")
lcpm <- cpm(x, log=TRUE)
plot(density(lcpm[,1]), col=col[1], lwd=2, ylim=c(0,0.21), las=2,
     main="", xlab="")
```

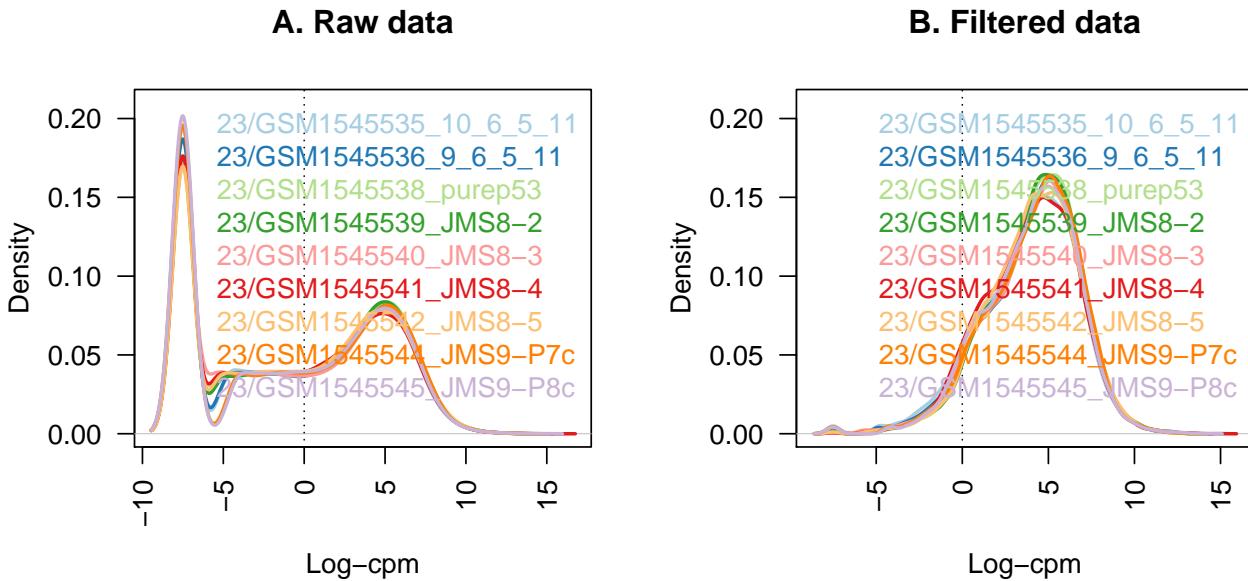


Figure 6.1: The density of log-CPM values for raw pre-filtered data (A) and post-filtered data (B) are shown for each sample. Dotted vertical lines mark the log-CPM of zero threshold (equivalent to a CPM value of 1) used in the filtering step.

```
title(main="B. Filtered data", xlab="Log-cpm")
abline(v=0, lty=3)
for (i in 2:nsamples){
  den <- density(lcpm[,i])
  lines(den$x, den$y, col=col[i], lwd=2)
}
legend("topright", samplenames, text.col=col, bty="n")
```

6.6.3 Normalising gene expression distributions

During the sample preparation or sequencing process, external factors that are not of biological interest can affect the expression of individual samples. For example, samples processed in the first batch of an experiment can have higher expression overall when compared to samples processed in a second batch. It is assumed that all samples should have a similar range and distribution of expression values. Normalisation is required to ensure that the expression distributions of each sample are similar across the entire experiment.

Any plot showing the per sample expression distributions, such as a density or boxplot, is useful in determining whether any samples are dissimilar to others. Distributions of log-CPM values are similar throughout all samples within this dataset (panel B of the figure above).

Nonetheless, normalisation by the method of trimmed mean of M-values (TMM) is performed using the `calcNormFactors` function in `edgeR`. The normalisation factors calculated here are used as a scaling factor for the library sizes. When working with `DGEList`-objects, these normalisation factors are automatically stored in `x$samples$norm.factors`. For this dataset the effect of TMM-normalisation is mild, as evident in the magnitude of the scaling factors, which are all relatively close to 1.

```
x <- calcNormFactors(x, method = "TMM")
x$samples$norm.factors
#> [1] 0.8957309 1.0349196 1.0439552 1.0405040 1.0323599 0.9223424 0.9836603
#> [8] 1.0827381 0.9792607
```

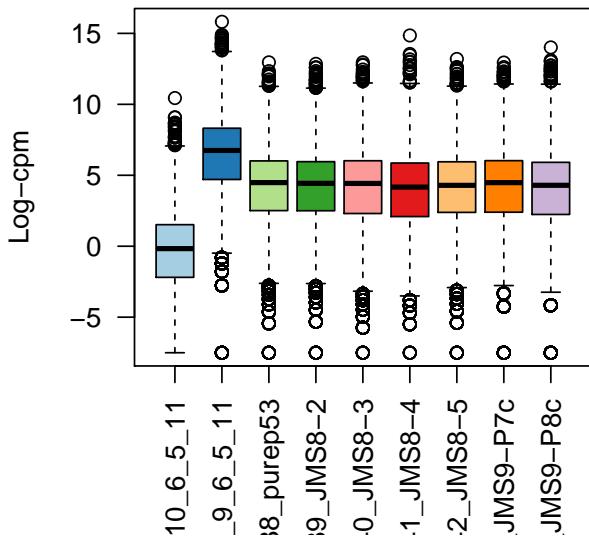
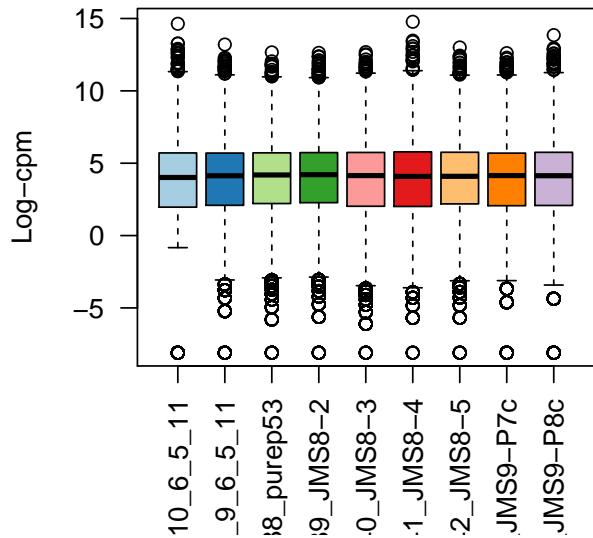
A. Example: Unnormalised data**B. Example: Normalised data**

Figure 6.2: Example data: Boxplots of log-CPM values showing expression distributions for unnormalised data (A) and normalised data (B) for each sample in the modified dataset where the counts in samples 1 and 2 have been scaled to 5% and 500% of their original values respectively.

To give a better visual representation of the effects of normalisation, the data was duplicated then adjusted so that the counts of the first sample are reduced to 5% of their original values, and in the second sample they are inflated to be 5-times larger.

```
x2 <- x
x2$samples$norm.factors <- 1
x2$counts[,1] <- ceiling(x2$counts[,1]*0.05)
x2$counts[,2] <- x2$counts[,2]*5
```

The figure below shows the expression distribution of samples for unnormalised and normalised data, where distributions are noticeably different pre-normalisation and are similar post-normalisation. Here the first sample has a small TMM scaling factor of 0.05, whereas the second sample has a large scaling factor of 6.13 – neither values are close to 1.

```
par(mfrow=c(1,2))
lcpm <- cpm(x2, log=TRUE)
boxplot(lcpm, las=2, col=col, main="")
title(main="A. Example: Unnormalised data",ylab="Log-cpm")
x2 <- calcNormFactors(x2)
x2$samples$norm.factors
#> [1] 0.05472223 6.13059440 1.22927355 1.17051887 1.21487709 1.05622968
#> [7] 1.14587663 1.26129350 1.11702264
lcpm <- cpm(x2, log=TRUE)
boxplot(lcpm, las=2, col=col, main="")
title(main="B. Example: Normalised data",ylab="Log-cpm")
```

6.6.4 Unsupervised clustering of samples

In our opinion, one of the most important exploratory plots to examine for gene expression analyses is the multi-dimensional scaling (MDS) plot, or similar. The plot shows similarities and dissimilarities between samples in an unsupervised manner so that one can have an idea of the extent to which differential expression can be detected before carrying out formal tests. Ideally, samples would cluster well within the primary condition of interest, and any sample straying far from its group could be identified and followed up for sources of error or extra variation. If present, technical replicates should lie very close to one another.

Such a plot can be made in **limma** using the `plotMDS` function. The first dimension represents the leading-fold-change that best separates samples and explains the largest proportion of variation in the data, with subsequent dimensions having a smaller effect and being orthogonal to the ones before it. When experimental design involves multiple factors, it is recommended that each factor is examined over several dimensions. If samples cluster by a given factor in any of these dimensions, it suggests that the factor contributes to expression differences and is worth including in the linear modelling. On the other hand, factors that show little or no effect may be left out of downstream analysis.

In this dataset, samples can be seen to cluster well within experimental groups over dimension 1 and 2, and then separate by sequencing lane (sample batch) over dimension 3 (shown in the plot below). Keeping in mind that the first dimension explains the largest proportion of variation in the data, notice that the range of values over the dimensions become smaller as we move to higher dimensions.

Whilst all samples cluster by groups, the largest transcriptional difference is observed between basal and LP, and basal and ML over dimension 1. For this reason, it is expected that pairwise comparisons between cell populations will result in a greater number of DE genes for comparisons involving basal samples, and relatively small numbers of DE genes when comparing ML to LP. In other datasets, samples that do not cluster by their groups of interest may also show little or no evidence of differential expression in the downstream analysis.

To create the MDS plots, different colour groupings are assigned to factors of interest. Dimensions 1 and 2 are examined using the color grouping defined by cell types.

Dimensions 3 and 4 are examined using the colour grouping defined by sequencing lanes (batch).

```
lcpm <- cpm(x, log=TRUE)
par(mfrow=c(1,2))
col.group <- group
levels(col.group) <- brewer.pal(nlevels(col.group), "Set1")
col.group <- as.character(col.group)
col.lane <- lane
levels(col.lane) <- brewer.pal(nlevels(col.lane), "Set2")
col.lane <- as.character(col.lane)
plotMDS(lcpm, labels=group, col=col.group)
title(main="A. Sample groups")
plotMDS(lcpm, labels=lane, col=col.lane, dim=c(3,4))
title(main="B. Sequencing lanes")
```

Alternatively, the **Glimma** package offers the convenience of an interactive MDS plot where multiple dimensions can be explored. The `g1MDSPlot` function generates an html page (that is opened in a browser if `launch=TRUE`) with an MDS plot in the left panel and a barplot showing the proportion of variation explained by each dimension in the right panel. Clicking on the bars of the bar plot changes the pair of dimensions plotted in the MDS plot, and hovering over the individual points reveals the sample label. The colour scheme can be changed as well to highlight cell population or sequencing lane (batch). An interactive MDS plot of this dataset can be found at <http://bioinf.wehi.edu.au/folders/limmaWorkflow/glimma-plots/MDS-Plot.html>.

```
g1MDSPlot(lcpm, labels=paste(group, lane, sep="_"),
          groups=x$samples[,c(2,5)], launch=FALSE)
```

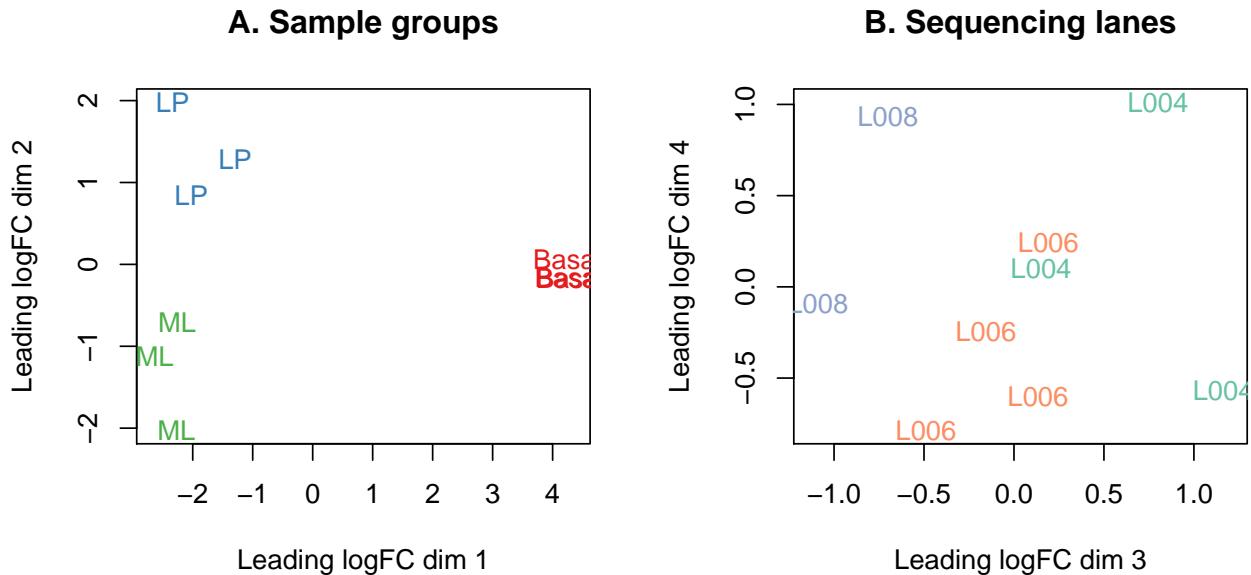


Figure 6.3: MDS plots of log-CPM values over dimensions 1 and 2 with samples coloured and labeled by sample groups (A) and over dimensions 3 and 4 with samples coloured and labeled by sequencing lane (B). Distances on the plot correspond to the leading fold-change, which is the average (root-mean-square) log2-fold-change for the 500 genes most divergent between each pair of samples by default.

6.7 Differential expression analysis

6.7.1 Creating a design matrix and contrasts

In this study, it is of interest to see which genes are expressed at different levels between the three cell populations profiled. In our analysis, linear models are fitted to the data with the assumption that the underlying data is normally distributed. To get started, a design matrix is set up with both the cell population and sequencing lane (batch) information.

```
design <- model.matrix(~0+group+lane)
colnames(design) <- gsub("group", "", colnames(design))
design
#>   Basal LP ML laneL006 laneL008
#> 1    0  1  0      0      0
#> 2    0  0  1      0      0
#> 3    1  0  0      0      0
#> 4    1  0  0      1      0
#> 5    0  0  1      1      0
#> 6    0  1  0      1      0
#> 7    1  0  0      1      0
#> 8    0  0  1      0      1
#> 9    0  1  0      0      1
#> attr(,"assign")
#> [1] 1 1 1 2 2
#> attr(,"contrasts")
#> attr(,"contrasts")$group
#> [1] "contr.treatment"
#>
#> attr(,"contrasts")$lane
```

```
#> [1] "contr.treatment"
```

For a given experiment, there are usually several equivalent ways to set up an appropriate design matrix. For example, `~0+group+lane` removes the intercept from the first factor, `group`, but an intercept remains in the second factor `lane`. Alternatively, `~group+lane` could be used to keep the intercepts in both `group` and `lane`. Understanding how to interpret the coefficients estimated in a given model is key here. We choose the first model for our analysis, as setting up model contrasts is more straight forward in the absence of an intercept for `group`. Contrasts for pairwise comparisons between cell populations are set up in **limma** using the `makeContrasts` function.

```
contr.matrix <- makeContrasts(
  BasalvsLP = Basal-LP,
  BasalvsML = Basal - ML,
  LPvsML = LP - ML,
  levels = colnames(design))
contr.matrix
#>           Contrasts
#> Levels      BasalvsLP BasalvsML LPvsML
#> Basal          1        1        0
#> LP            -1       0        1
#> ML             0       -1       -1
#> laneL006       0        0        0
#> laneL008       0        0        0
```

A key strength of **limma**'s linear modelling approach, is the ability accommodate arbitrary experimental complexity. Simple designs, such as the one in this workflow, with cell type and batch, through to more complicated factorial designs and models with interaction terms can be handled relatively easily. Where experimental or technical effects can be modelled using a random effect, another possibility in **limma** is to estimate correlations using `duplicateCorrelation` by specifying a `block` argument for both this function and in the `lmFit` linear modelling step.

6.7.2 Removing heteroscedascity from count data

It has been shown that for RNA-seq count data, the variance is not independent of the mean – this is true of raw counts or when transformed to log-CPM values. Methods that model counts using a Negative Binomial distribution assume a quadratic mean-variance relationship. In **limma**, linear modelling is carried out on the log-CPM values which are assumed to be normally distributed and the mean-variance relationship is accommodated using precision weights calculated by the `voom` function.

When operating on a `DGEList`-object, `voom` converts raw counts to log-CPM values by automatically extracting library sizes and normalisation factors from `x` itself. Additional normalisation to log-CPM values can be specified within `voom` using the `normalize.method` argument.

The mean-variance relationship of log-CPM values for this dataset is shown in the left-hand panel of the next figure. Typically, the *voom-plot* shows a decreasing trend between the means and variances resulting from a combination of technical variation in the sequencing experiment and biological variation amongst the replicate samples from different cell populations. Experiments with high biological variation usually result in flatter trends, where variance values plateau at high expression values. Experiments with low biological variation tend to result in sharp decreasing trends.

Moreover, the *voom-plot* provides a visual check on the level of filtering performed upstream. If filtering of lowly-expressed genes is insufficient, a drop in variance levels can be observed at the low end of the expression scale due to very small counts. If this is observed, one should return to the earlier filtering step and increase the expression threshold applied to the dataset.

Where sample-level variation is evident from earlier inspections of the MDS plot, the `voomWithQualityWeights` function can be used to simultaneously incorporate sample-level weights together with the abundance dependent weights estimated by `voom`. For an example of this, see Liu *et al.* (2016).

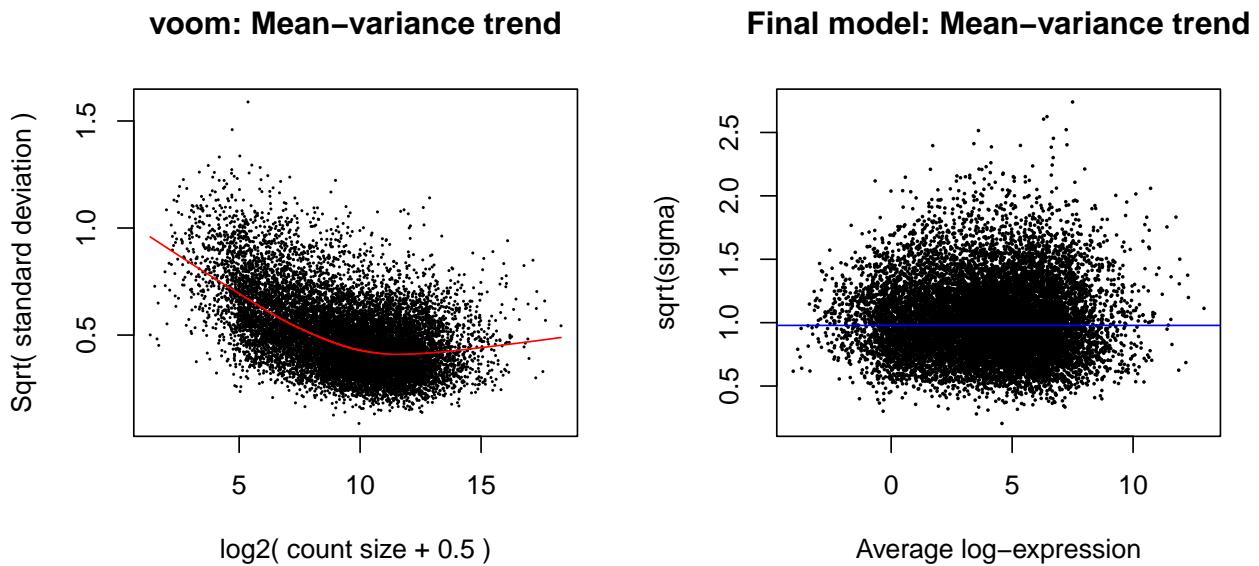
```
par(mfrow=c(1,2))
v <- voom(x, design, plot=TRUE)
v
#> An object of class "EList"
#> $genes
#>   ENTREZID SYMBOL TXCHROM
#> 1    497097 Xkr4    chr1
#> 6    27395 Mrpl15   chr1
#> 7    18777 Lypla1   chr1
#> 9    21399 Tceal1   chr1
#> 10   58175 Rgs20   chr1
#> 14160 more rows ...
#>
#> $targets
#>
#>                                         files group
#> 23/GSM1545535_10_6_5_11 Law_RNAseq123/GSM1545535_10_6_5_11.txt LP
#> 23/GSM1545536_9_6_5_11  Law_RNAseq123/GSM1545536_9_6_5_11.txt ML
#> 23/GSM1545538_purep53  Law_RNAseq123/GSM1545538_purep53.txt Basal
#> 23/GSM1545539_JMS8-2   Law_RNAseq123/GSM1545539_JMS8-2.txt Basal
#> 23/GSM1545540_JMS8-3   Law_RNAseq123/GSM1545540_JMS8-3.txt ML
#> 23/GSM1545541_JMS8-4   Law_RNAseq123/GSM1545541_JMS8-4.txt LP
#> 23/GSM1545542_JMS8-5   Law_RNAseq123/GSM1545542_JMS8-5.txt Basal
#> 23/GSM1545544_JMS9-P7c Law_RNAseq123/GSM1545544_JMS9-P7c.txt ML
#> 23/GSM1545545_JMS9-P8c Law_RNAseq123/GSM1545545_JMS9-P8c.txt LP
#> lib.size norm.factors lane
#> 23/GSM1545535_10_6_5_11 29409426  0.8957309 L004
#> 23/GSM1545536_9_6_5_11  36528591  1.0349196 L004
#> 23/GSM1545538_purep53  59598629  1.0439552 L004
#> 23/GSM1545539_JMS8-2   53382070  1.0405040 L006
#> 23/GSM1545540_JMS8-3   78175314  1.0323599 L006
#> 23/GSM1545541_JMS8-4   55762781  0.9223424 L006
#> 23/GSM1545542_JMS8-5   54115150  0.9836603 L006
#> 23/GSM1545544_JMS9-P7c 23043111  1.0827381 L008
#> 23/GSM1545545_JMS9-P8c 19525423  0.9792607 L008
#>
#> $E
#>   Samples
#> Tags      23/GSM1545535_10_6_5_11 23/GSM1545536_9_6_5_11
#> 497097    -4.293244            -3.869026
#> 27395     3.875010            4.400568
#> 18777     4.707695            5.559334
#> 21399     4.784462            4.741999
#> 58175     3.943567            3.294875
#>   Samples
#> Tags      23/GSM1545538_purep53 23/GSM1545539_JMS8-2 23/GSM1545540_JMS8-3
#> 497097    2.522753            3.302006            -4.481286
#> 27395     4.521172            4.570624            4.322845
#> 18777     5.400569            5.171235            5.627798
#> 21399     5.374548            5.130925            4.848030
#> 58175     -1.767924           -1.880302           2.993289
```

```

#>      Samples
#> Tags   23/GSM1545541_JMS8-4 23/GSM1545542_JMS8-5 23/GSM1545544_JMS9-P7c
#> 497097 -3.993876          3.306782          -3.204336
#> 27395  3.786547          3.918878          4.345642
#> 187777 5.081794          5.080061          5.757404
#> 21399  4.944024          5.158292          5.036933
#> 58175  3.357379          -2.114104          3.142621
#>      Samples
#> Tags   23/GSM1545545_JMS9-P8c
#> 497097 -5.287282
#> 27395  4.132678
#> 187777 5.150470
#> 21399  4.987679
#> 58175  3.523290
#> 14160 more rows ...
#>
#> $weights
#>      [,1]      [,2]      [,3]      [,4]      [,5]      [,6]      [,7]
#> [1,] 1.183974 1.183974 20.526779 20.97747 1.773562 1.217142 21.125740
#> [2,] 20.879554 26.561871 31.596323 29.66102 32.558344 26.745293 29.792090
#> [3,] 28.003202 33.695540 34.845507 34.45673 35.148529 33.550527 34.517259
#> [4,] 27.670233 29.595778 34.901302 34.43298 34.841349 33.159425 34.493456
#> [5,] 19.737381 18.658333 3.184207 2.62986 24.191635 24.014937 2.648747
#>      [,8]      [,9]
#> [1,] 1.183974 1.183974
#> [2,] 21.900102 17.150677
#> [3,] 31.440457 25.228325
#> [4,] 26.136796 24.502247
#> [5,] 13.149278 14.351930
#> 14160 more rows ...
#>
#> $design
#> Basal LP ML laneL006 laneL008
#> 1 0 1 0 0 0
#> 2 0 0 1 0 0
#> 3 1 0 0 0 0
#> 4 1 0 0 1 0
#> 5 0 0 1 1 0
#> 6 0 1 0 1 0
#> 7 1 0 0 1 0
#> 8 0 0 1 0 1
#> 9 0 1 0 0 1
#> attr(,"assign")
#> [1] 1 1 1 2 2
#> attr(,"contrasts")
#> attr(,"contrasts")$group
#> [1] "contr.treatment"
#>
#> attr(,"contrasts")$lane
#> [1] "contr.treatment"
vfit <- lmFit(v, design)
vfit <- contrasts.fit(vfit, contrasts=contr.matrix)
efit <- eBayes(vfit)

```

```
plotSA(efit, main="Final model: Mean-variance trend")
```



Note that the other data frames stored within the DGEList-object that contain gene- and sample-level information, are retained in the EList-object `v` created by `voom`. The `v$genes` data frame is equivalent to `x$genes`, `v$targets` is equivalent to `x$samples`, and the expression values stored in `v$E` is analogous to `x$counts`, albeit on a transformed scale. In addition to this, the `voom` EList-object has a matrix of precision weights `v$weights` and stores the design matrix in `v$design`.

6.7.3 Fitting linear models for comparisons of interest

Linear modelling in `limma` is carried out using the `lmFit` and `contrasts.fit` functions originally written for application to microarrays. The functions can be used for both microarray and RNA-seq data and fit a separate model to the expression values for each gene. Next, empirical Bayes moderation is carried out by borrowing information across all the genes to obtain more precise estimates of gene-wise variability. The model's residual variances are plotted against average expression values in the next figure. It can be seen from this plot that the variance is no longer dependent on the mean expression level.

6.7.4 Examining the number of DE genes

For a quick look at differential expression levels, the number of significantly up- and down-regulated genes can be summarised in a table. Significance is defined using an adjusted p -value cutoff that is set at 5% by default. For the comparison between expression levels in basal and LP, 4,127 genes are found to be down-regulated in basal relative to LP and 4,298 genes are up-regulated in basal relative to LP – a total of 8,425 DE genes. A total of 8,510 DE genes are found between basal and ML (4,338 down- and 4,172 up-regulated genes), and a total of 5,340 DE genes are found between LP and ML (2,895 down- and 2,445 up-regulated). The larger numbers of DE genes observed for comparisons involving the basal population are consistent with our observations from the MDS plots.

```
summary(decideTests(efit))
#>      BasalvsLP BasalvsML LPvsML
#> Down      4127     4338    2895
#> NotSig    5740     5655    8825
#> Up        4298     4172    2445
```

Some studies require more than an adjusted *p*-value cut-off. For a stricter definition on significance, one may require log-fold-changes (log-FCs) to be above a minimum value. The *treat* method can be used to calculate *p*-values from empirical Bayes moderated *t*-statistics with a minimum log-FC requirement. The number of differentially expressed genes are reduced to a total of 3,135 DE genes for basal versus LP, 3,270 DE genes for basal versus ML, and 385 DE genes for LP versus ML when testing requires genes to have a log-FC that is significantly greater than 1 (equivalent to a 2-fold difference between cell types on the original scale).

```
tfit <- treat(vfit, lfc=1)
dt <- decideTests(tfit)
summary(dt)

#>      BasalvsLP BasalvsML LPvsML
#> Down      1417     1512    203
#> NotSig    11030    10895   13780
#> Up        1718     1758    182
```

Genes that are DE in multiple comparisons can be extracted using the results from *decideTests*, where 0s represent genes that are not DE, 1s represent genes that are up-regulated, and -1s represent genes that are down-regulated. A total of 2,409 genes are DE in both basal versus LP and basal versus ML, twenty of which are listed below. The *write.fit* function can be used to extract and write results for all three comparisons to a single output file.

```
de.common <- which(dt[,1] != 0 & dt[,2] != 0)
length(de.common)
#> [1] 2409
head(tfit$genes$SYMBOL[de.common], n=20)
#> [1] "Xkr4"          "Rgs20"         "Cpa6"          "Sulf1"
#> [5] "Eya1"          "Msc"           "Sbspon"        "Pi15"
#> [9] "Crispld1"       "Kcnq5"         "Ptpn18"        "Arhgef4"
#> [13] "2010300C02Rik" "Aff3"          "Npas2"         "Tbc1d8"
#> [17] "Creg2"         "Il1r1"         "Il18r1"        "Il18rap"
vennDiagram(dt[,1:2], circle.col=c("turquoise", "salmon"))

write.fit(tfit, dt, file="results.txt")
```

6.7.5 Examining individual DE genes from top to bottom

The top DE genes can be listed using *topTreat* for results using *treat* (or *topTable* for results using *eBayes*). By default *topTreat* arranges genes from smallest to largest adjusted *p*-value with associated gene information, log-FC, average log-CPM, moderated *t*-statistic, raw and adjusted *p*-value for each gene. The number of top genes displayed can be specified, where *n=Inf* includes all genes. Genes *Cldn7* and *Rasef* are amongst the top DE genes for both basal versus LP and basal versus ML.

```
basal.vs.lp <- topTreat(tfit, coef=1, n=Inf)
basal.vs.ml <- topTreat(tfit, coef=2, n=Inf)
head(basal.vs.lp)

#>      ENTREZID SYMBOL TXCHROM      logFC AveExpr          t      P.Value
#> 12759    12759   Clu   chr14 -5.442877 8.857907 -33.44429 3.990899e-10
#> 53624    53624  Cldn7  chr11 -5.514605 6.296762 -32.94533 4.503694e-10
#> 242505   242505  Rasef  chr4 -5.921741 5.119585 -31.77625 6.063249e-10
#> 67451    67451   Pkp2  chr16 -5.724823 4.420495 -30.65370 8.010456e-10
#> 228543   228543  Rhov   chr2 -6.253427 5.486640 -29.46244 1.112729e-09
#> 70350    70350   Basp1  chr15 -6.073297 5.248349 -28.64890 1.380545e-09
#>          adj.P.Val
#> 12759  2.703871e-06
#> 53624  2.703871e-06
```

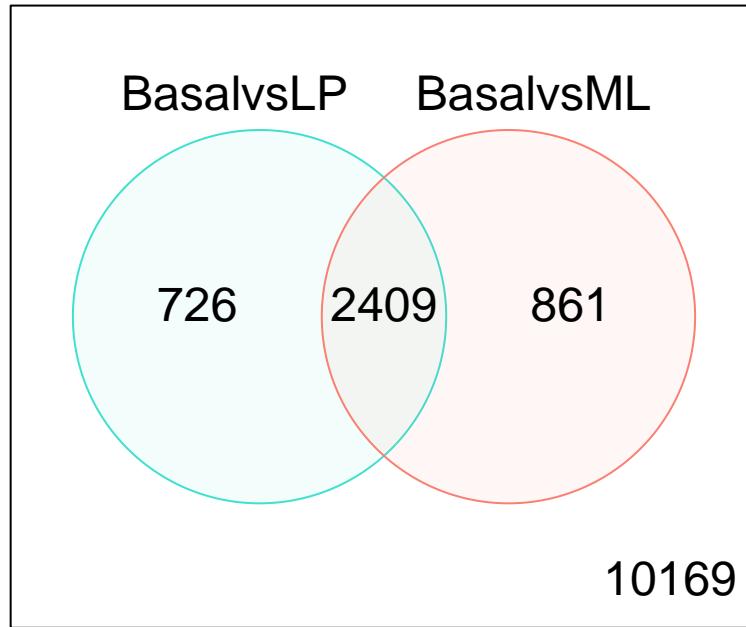


Figure 6.4: Venn diagram showing the number of genes DE in the comparison between basal versus LP only (left), basal versus ML only (right), and the number of genes that are DE in both comparisons (center). The number of genes that are not DE in either comparison are marked in the bottom-right.

```
#> 242505 2.703871e-06
#> 67451 2.703871e-06
#> 228543 2.703871e-06
#> 70350 2.703871e-06
head(basal.vs.ml)
#>      ENTREZID SYMBOL TXCHROM      logFC AveExpr          t     P.Value
#> 242505 242505 Rasef   chr4 -6.510470 5.119585 -35.49093 2.573575e-10
#> 53624  53624 Cldn7  chr11 -5.469160 6.296762 -32.52520 4.978446e-10
#> 12521  12521 Cd82   chr2 -4.667737 7.070963 -31.82187 5.796191e-10
#> 71740  71740 Nectin4 chr1 -5.556046 5.166292 -31.29987 6.760578e-10
#> 20661  20661 Sort1  chr3 -4.908119 6.705784 -31.23083 6.761331e-10
#> 15375  15375 Foxa1  chr12 -5.753884 5.625064 -28.34612 1.487280e-09
#>      adj.P.Val
#> 242505 1.915485e-06
#> 53624  1.915485e-06
#> 12521  1.915485e-06
#> 71740  1.915485e-06
#> 20661  1.915485e-06
#> 15375  2.281914e-06
```

6.7.6 Useful graphical representations of differential expression results

To summarise results for all genes visually, mean-difference plots, which display log-FCs from the linear model fit against the average log-CPM values can be generated using the `plotMD` function, with the differentially expressed genes highlighted.

```
plotMD(tfit, column=1, status=dt[,1], main=colnames(tfit)[1],
       xlim=c(-8,13))
```

Glimma extends this functionality by providing an interactive mean-difference plot via the `g1MDPlot` function. The output of this function is an html page, with summarised results in the left panel (similar to what is output by `plotMD`), and the log-CPM values from individual samples in the right panel, with a table of results below the plots. This interactive display allows the user to search for particular genes based on their Gene symbol, which is not possible in a static **R** plot. The `g1MDPlot` function is not limited to mean-difference plots, with a default version allowing a data frame to be passed with the user able to select the columns of interest to plot in the left panel.

```
g1MDPlot(tfit, coef=1, status=dt, main=colnames(tfit)[1],
          side.main="ENTREZID", counts=x$counts, groups=group, launch=FALSE)
```

The mean-difference plot generated by the command above is available online (see <http://bioinf.wehi.edu.au/folders/limmaWorkflow/glimma-plots/MD-Plot.html>). The interactivity provided by the **Glimma** package allows additional information to be presented in a single graphical window. **Glimma** is implemented in **R** and Javascript, with the **R** code generating the data which is converted into graphics using the Javascript library D3 (<https://d3js.org>), with the Bootstrap library handling layouts and Datatables generating the interactive searchable tables. This allows plots to be viewed in any modern browser, which is convenient for including them as linked files from an Rmarkdown report of the analysis.

Plots shown previously include either all of the genes that are expressed in any one condition (such as the Venn diagram of common DE genes or mean-difference plot) or look at genes individually (log-CPM values shown in right panel of the interactive mean-difference plot). Heatmaps allow users to look at the expression of a subset of genes. This can be give useful insight into the expression of individual groups and samples without losing perspective of the overall study when focusing on individual genes, or losing resolution when examining patterns averaged over thousands of genes at the same time.

A heatmap is created for the top 100 DE genes (as ranked by adjusted p-value) from the basal versus LP contrast using the `heatmap.2` function from the **gplots** package. The heatmap correctly clusters samples into cell type and rearranges the order of genes to form blocks of similar expression. From the heatmap, we observe that the expression of ML and LP samples are very similar for the top 100 DE genes between basal and LP.

```
library(gplots)
basal.vs.lp.topgenes <- basal.vs.lp$ENTREZID[1:100]
i <- which(v$genes$ENTREZID %in% basal.vs.lp.topgenes)
mycol <- colorpanel(1000,"blue","white","red")
heatmap.2(v$E[i,], scale="row",
          labRow=v$genes$SYMBOL[i], labCol=group,
          col=mycol, trace="none", density.info="none",
          margin=c(8,6), lhei=c(2,10), dendrogram="column")
```

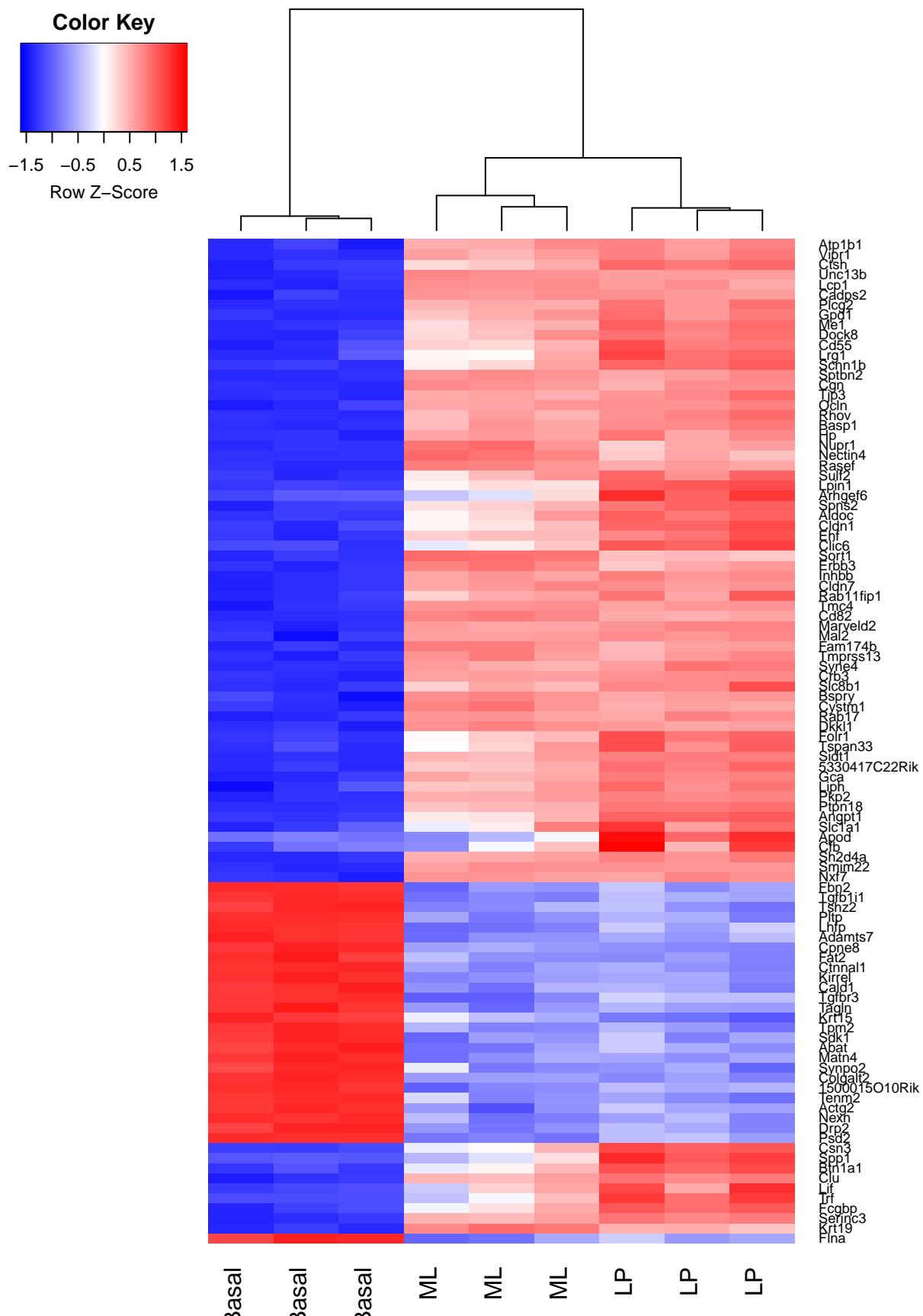


Figure 6.5: Heatmap of log-CPM values for top 100 genes DE in basal versus LP. Expression across each gene (or row) have been scaled so that mean expression is zero and standard deviation is one. Samples with relatively high expression of a given gene are marked in red and samples with relatively low expression are marked in blue. Lighter shades and white represent genes with intermediate expression levels. Samples and genes have been reordered by the method of hierarchical clustering. A dendrogram is shown for the sample

Chapter 7

201: RNA-seq data analysis with DESeq2

Authors: Michael I. Love¹, Simon Anders², Wolfgang Huber³ Last modified: 25 June, 2018.

7.1 Overview

7.1.1 Description

In this workshop, we will give a quick overview of the most useful functions in the DESeq2 package, and a basic RNA-seq analysis. We will cover: how to quantify transcript expression from FASTQ files using Salmon, import quantification from Salmon with tximport and tximeta, generate plots for quality control and exploratory data analysis EDA (also using MultiQC), perform differential expression (DE) (also using apegelm), overlap with other experimental data (using AnnotationHub), and build reports (using ReportingTools and Glimma). We will give a short example of integration of DESeq2 with the zinbwave package for single-cell RNA-seq differential expression. The workshop is designed to be a lab with plenty of time for questions throughout the lab.

7.1.2 Pre-requisites

- Basic knowledge of R syntax

Non-essential background reading:

- DESeq2 paper: <https://www.ncbi.nlm.nih.gov/pmc/articles/PMC4302049/>
- tximport paper: <https://f1000research.com/articles/4-1521/v2>
- apegelm paper: <https://www.biorxiv.org/content/early/2018/04/17/303255>

7.1.3 Participation

Students will participate by following along an Rmarkdown document, and asking questions throughout the workshop.

¹UNC-Chapel Hill, NC, US

²ZMBH Heidelberg, Germany

³EMBL Heidelberg, Germany

7.1.4 R / Bioconductor packages used

- DESeq2
- tximport
- apeglm
- AnnotationHub
- ReportingTools
- Glimma
- splatter
- zinbwave

7.1.5 Time outline

Activity	Time
Overview of packages	20m
Quantification and import	20m
EDA and DE	20m
Downstream analysis & reports	20m
ZINB-WaVE integration	20m
Additional questions	20m

7.1.6 Workshop goals and objectives

Learning goals

- Visually assess quality of RNA-seq data
- Perform basic differential analysis of RNA-seq data
- Compare RNA-seq results with other experimental data

Learning objectives

- Quantify transcript expression from FASTQ files
- Import quantification into R/Bioconductor
- Perform quality control and exploratory data analysis
- Perform differential expression
- Overlap with other experimental data
- Build dynamic reports
- Integrate DESeq2 and zinbwave for single-cell RNA-seq data

7.2 Preparing data for *DESeq2*

7.2.1 Experimental data

The data used in this workflow is stored in the *airway* package that summarizes an RNA-seq experiment wherein airway smooth muscle cells were treated with dexamethasone, a synthetic glucocorticoid steroid with anti-inflammatory effects (Himes et al. 2014). Glucocorticoids are used, for example, by people with asthma to reduce inflammation of the airways. In the experiment, four primary human airway smooth muscle cell lines were treated with 1 micromolar dexamethasone for 18 hours. For each of the four cell lines, we have a treated and an untreated sample. For more description of the experiment see the PubMed entry 24926665 and for raw data see the GEO entry GSE52778.

We will show how to import RNA-seq quantification data using an alternative dataset (the *tximportData* package which is used in the *tximport* vignette). Afterward we will load counts for the *airway* dataset, which were counted using *summarizeOverlaps* from the *GenomicAlignments* package. As described below, we recommend the *tximport* pipeline for producing count matrices, but we do not yet have a Bioconductor package containing the necessary quantification files for the *airway* dataset.

7.2.2 Modeling count data

As input, the count-based statistical methods, such as *DESeq2* (Love, Huber, and Anders 2014), *edgeR* (Robinson, McCarthy, and Smyth 2009), *limma* with the voom method (Law et al. 2014), *DSS* (Wu, Wang, and Wu 2013), *EBSeq* (Leng et al. 2013) and *baySeq* (Hardcastle and Kelly 2010), expect input data as obtained, e.g., from RNA-seq or another high-throughput sequencing experiment, in the form of a matrix of counts. The value in the i -th row and the j -th column of the matrix tells how many reads (or fragments, for paired-end RNA-seq) have been assigned to gene i in sample j . Analogously, for other types of assays, the rows of the matrix might correspond e.g., to binding regions (with ChIP-Seq), species of bacteria (with metagenomic datasets), or peptide sequences (with quantitative mass spectrometry).

The values in the matrix should be counts of sequencing reads/fragments. This is important for the statistical models used by *DESeq2* and *edgeR* to hold, as only counts allow assessing the measurement precision correctly. It is important to not provide counts that were pre-normalized for sequencing depth (also called library size), as the statistical model is most powerful when applied to un-normalized counts and is designed to account for library size differences internally.

7.2.3 Transcript abundances

In this workflow, we will show how to use transcript abundances as quantified by the *Salmon* (Patro et al. 2017) software package. *Salmon* and other methods, such as *Sailfish* (Patro, Mount, and Kingsford 2014), *kallisto* (Bray et al. 2016), or *RSEM* (Bo Li and Dewey 2011), estimate the relative abundances of all (known, annotated) transcripts without aligning reads. Because estimating the abundance of the transcripts involves an inference step, the counts are *estimated*. Most methods either use a statistical framework called Estimation-Maximization or Bayesian techniques to estimate the abundances and counts. Following quantification, we will use the *tximport* (Soneson, Love, and Robinson 2015) package for assembling estimated count and offset matrices for use with Bioconductor differential gene expression packages.

The advantages of using the transcript abundance quantifiers in conjunction with *tximport* to produce gene-level count matrices and normalizing offsets, are:

1. this approach corrects for any potential changes in gene length across samples (e.g. from differential isoform usage) (Trapnell et al. 2013)
2. some of these methods are substantially faster and require less memory and less disk usage compared to alignment-based methods
3. it is possible to avoid discarding those fragments that can align to multiple genes with homologous sequence (Robert and Watson 2015).

Note that transcript abundance quantifiers skip the generation of large files which store read alignments (SAM or BAM files), instead producing smaller files which store estimated abundances, counts and effective lengths per transcript. For more details, see the manuscript describing this approach (Soneson, Love, and Robinson 2015) and the *tximport* package vignette for software details.

A full tutorial on how to use the *Salmon* software for quantifying transcript abundance can be found here.

7.2.4 *Salmon* quantification

We begin by providing *Salmon* with the sequence of all of the reference transcripts, which we will call the *reference transcriptome*. We recommend to use the GENCODE human transcripts, which can be downloaded from the GENCODE website. On the command line, creating the transcriptome index looks like:

```
salmon index -i gencode.v99_salmon_0.10.0 -t gencode.v99.transcripts.fa.gz
```

The 0.10.0 refers to the version of *Salmon* that was used, and is useful to put into the index name.

To quantify an individual sample, `sample_01`, the following command can be used:

```
salmon quant -i gencode.v99_salmon_0.10.0 -p 6 --libType A \
--gcBias --biasSpeedSamp 5 \
-1 sample_01_1.fastq.gz -2 sample_01_2.fastq.gz \
-o sample_01
```

In simple English, this command says to “quantify a sample using this transcriptome index, with 6 threads, using automatic library type detection, using GC bias correction (the bias speed part is now longer needed with current versions of *Salmon*), here are the first and second read, and use this output directory.” The output directory will be created if it doesn’t exist, though if earlier parts of the path do not exist, it will give an error. A single sample of human RNA-seq usually takes ~5 minutes with the GC bias correction.

Rather than writing the above command on the command line multiple times for each sample, it is possible to loop over files using a bash loop, or more advanced workflow management systems such as Snakemake (Köster and Rahmann 2012) or Nextflow (Di Tommaso et al. 2017).

7.3 Importing into R with *tximport*

7.3.1 Specifying file locations

Following quantification, we can use *tximport* to import the data into R and perform statistical analysis using Bioconductor packages. Normally, we would simply point *tximport* to the `quant.sf` files on our machine. However, because we are distributing these files as part of an R package, we have to do some extra steps, to figure out where the R package, and so the files, are located on *your* machine.

We will show how to import *Salmon* quantification files using the data in the *tximportData* package. The quantified samples are six samples from the GEUVADIS Project (Lappalainen et al. 2013). The output directories from the above *Salmon* quantification calls has been stored in the `extdata` directory of the *tximportData* package. The R function `system.file` can be used to find out where on your computer the files from a package have been installed. Here we ask for the full path to the `extdata` directory, where R packages store external data, that is part of the *tximportData* package.

```
library("tximportData")
dir <- system.file("extdata", package="tximportData")
list.files(dir)
#> [1] "cufflinks"                  "kallisto"
#> [3] "kallisto_boot"              "rsem"
#> [5] "sailfish"                   "salmon"
#> [7] "salmon_dm"                 "salmon_gibbs"
#> [9] "samples_extended.txt"      "samples.txt"
#> [11] "tx2gene.csv"                "tx2gene.gencode.v27.csv"
```

The *Salmon* quantification directories are in the `salmon` directory.

```
list.files(file.path(dir, "salmon"))
#> [1] "ERR188021" "ERR188088" "ERR188288" "ERR188297" "ERR188329" "ERR188356"
```

The identifiers used here are the *ERR* identifiers from the European Nucleotide Archive. We need to create a named vector pointing to the quantification files. We will create a vector of filenames first by reading in a table that contains the sample IDs, and then combining this with `dir` and `"quant.sf.gz"`. (We gzipped the quantification files to make the data package smaller, this is not a problem for R functions that we use to import the files.)

```
samples <- read.table(file.path(dir,"samples.txt"), header=TRUE)
samples
#>   pop center      assay sample experiment      run
#> 1 TSI  UNIGE NA20503.1.M_111124_5 ERS185497 ERX163094 ERR188297
#> 2 TSI  UNIGE NA20504.1.M_111124_7 ERS185242 ERX162972 ERR188088
#> 3 TSI  UNIGE NA20505.1.M_111124_6 ERS185048 ERX163009 ERR188329
#> 4 TSI  UNIGE NA20507.1.M_111124_7 ERS185412 ERX163158 ERR188288
#> 5 TSI  UNIGE NA20508.1.M_111124_2 ERS185362 ERX163159 ERR188021
#> 6 TSI  UNIGE NA20514.1.M_111124_4 ERS185217 ERX163062 ERR188356
files <- file.path(dir, "salmon", samples$run, "quant.sf.gz")
names(files) <- paste0("sample",1:6)
all(file.exists(files))
#> [1] TRUE
```

7.3.2 Mapping transcripts to genes

Transcripts need to be associated with gene IDs for gene-level summarization. We therefore will construct a *data.frame* called `tx2gene` with two columns: 1) transcript ID and 2) gene ID. The column names do not matter but this column order must be used. The transcript ID must be the same one used in the abundance files. This can most easily be accomplished by downloading the GTF file at the same time that the transcriptome FASTA is downloaded, and generating `tx2gene` from the GTF file using Bioconductor's *TxDb* infrastructure.

Generating a *TxDb* from a GTF file can be easily accomplished with the `makeTxDbFromGFF` function. This step requires a few minutes of waiting, and a large file. We therefore skip this step, but show the code that is used to create the `tx2gene` table, assuming the correct *TxDb* object has been created.

Creating the `tx2gene` *data.frame* can be accomplished by calling the `select` function from the *AnnotationDbi* package on a *TxDb* object. The following code could be used to construct such a table:

```
library("TxDb.Hsapiens.UCSC.hg38.knownGene")
#> Loading required package: GenomicFeatures
#> Loading required package: BiocGenerics
#> Loading required package: parallel
#>
#> Attaching package: 'BiocGenerics'
#> The following objects are masked from 'package:parallel':
#>
#>   clusterApply, clusterApplyLB, clusterCall, clusterEvalQ,
#>   clusterExport, clusterMap, parApply, parCapply, parLapply,
#>   parLapplyLB, parRapply, parSapply, parSapplyLB
#> The following objects are masked from 'package:stats':
#>
#>   IQR, mad, sd, var, xtabs
#> The following objects are masked from 'package:base':
#>
#>   anyDuplicated, append, as.data.frame, basename, cbind,
#>   colMeans, colnames, colSums, dirname, do.call, duplicated,
#>   eval, evalq, Filter, Find, get, grep, grepl, intersect,
```

```

#>      is.unsorted, lapply, lengths, Map, mapply, match, mget, order,
#>      paste, pmax, pmax.int, pmin, pmin.int, Position, rank, rbind,
#>      Reduce, rowMeans, rownames, rowSums, sapply, setdiff, sort,
#>      table, tapply, union, unique, unsplit, which, which.max,
#>      which.min
#> Loading required package: S4Vectors
#> Loading required package: stats4
#>
#> Attaching package: 'S4Vectors'
#> The following object is masked from 'package:base':
#>
#>      expand.grid
#> Loading required package: IRanges
#> Loading required package: GenomeInfoDb
#> Loading required package: GenomicRanges
#> Loading required package: AnnotationDbi
#> Loading required package: Biobase
#> Welcome to Bioconductor
#>
#>      Vignettes contain introductory material; view with
#>      'browseVignettes()'. To cite Bioconductor, see
#>      'citation("Biobase")', and for packages 'citation("pkgname")'.
txdb <- TxDb.Hsapiens.UCSC.hg38.knownGene
k <- keys(txdb, keytype="TXNAME")
tx2gene <- select(txdb, k, "GENEID", "TXNAME")
#> 'select()' returned 1:1 mapping between keys and columns

```

In this case, we've used the Gencode v27 CHR transcripts to build our *Salmon* index, and we used `makeTxDbFromGFF` and code similar to the chunk above to build the `tx2gene` table. We then read in a pre-constructed `tx2gene` table:

```

library("readr")
tx2gene <- read_csv(file.path(dir, "tx2gene.gencode.v27.csv"))
#> Parsed with column specification:
#> cols(
#>   TXNAME = col_character(),
#>   GENEID = col_character()
#> )
head(tx2gene)
#> # A tibble: 6 x 2
#>   TXNAME          GENEID
#>   <chr>            <chr>
#> 1 ENST00000456328.2 ENSG00000223972.5
#> 2 ENST00000450305.2 ENSG00000223972.5
#> 3 ENST00000473358.1 ENSG00000243485.5
#> 4 ENST00000469289.1 ENSG00000243485.5
#> 5 ENST00000607096.1 ENSG00000284332.1
#> 6 ENST00000606857.1 ENSG00000268020.3

```

7.3.3 `tximport` command

Finally the following line of code imports *Salmon* transcript quantifications into R, collapsing to the gene level using the information in `tx2gene`.

```
library("tximport")
library("jsonlite")
library("readr")
txi <- tximport(files, type="salmon", tx2gene=tx2gene)
#> reading in files with read_tsv
#> 1 2 3 4 5 6
#> summarizing abundance
#> summarizing counts
#> summarizing length
```

The `txi` object is simply a list of matrices (and one character vector):

```
names(txi)
#> [1] "abundance"           "counts"            "length"
#> [4] "countsFromAbundance"
txi$counts[1:3,1:3]
#>                  sample1   sample2   sample3
#> ENSG00000000003.14  2.58012   2.000   27.09648
#> ENSG00000000005.5  0.00000   0.000   0.00000
#> ENSG00000000419.12 1056.99960 1337.997 1452.99497
txi$length[1:3,1:3]
#>                  sample1   sample2   sample3
#> ENSG00000000003.14 3621.7000 735.4220 2201.6223
#> ENSG00000000005.5 195.6667 195.6667 195.6667
#> ENSG00000000419.12 871.5077 905.0540 845.7278
txi$abundance[1:3,1:3]
#>                  sample1   sample2   sample3
#> ENSG00000000003.14 0.0354884 0.119404 0.411491
#> ENSG00000000005.5 0.0000000 0.000000 0.000000
#> ENSG00000000419.12 60.4173800 64.909276 57.441353
txi$countsFromAbundance
#> [1] "no"
```

If we were continuing with the GEUVADIS samples, we would then create a *DESeqDataSet* with the following line of code. Because there are no differences among the samples (same population and same sequencing batch), we specify a *design formula* of ~1, meaning we can only fit an intercept term – so we cannot perform differential expression analysis with these samples.

```
library("DESeq2")
#> Loading required package: SummarizedExperiment
#> Loading required package: DelayedArray
#> Loading required package: matrixStats
#>
#> Attaching package: 'matrixStats'
#> The following objects are masked from 'package:Biobase':
#>
#>     anyMissing, rowMedians
#> Loading required package: BiocParallel
#>
#> Attaching package: 'DelayedArray'
#> The following objects are masked from 'package:matrixStats':
#>
#>     colMaxs, colMins, colRanges, rowMaxs, rowMins, rowRanges
#> The following objects are masked from 'package:base':
#>
```

```
#>      aperm, apply
dds <- DESeqDataSetFromTximport(tximport, samples, ~1)
#> using counts and average transcript lengths from tximport
dds$center
#> [1] UNIGE UNIGE UNIGE UNIGE UNIGE UNIGE
#> Levels: UNIGE
dds$pop
#> [1] TSI TSI TSI TSI TSI TSI
#> Levels: TSI
```

7.4 Exploratory data analysis

7.4.1 Simple EDA

We will now switch over to the *airway* experiment, counts of which are already prepared in a *SummarizedExperiment* object. In this case, the object that we load is the output of the *summarizeOverlaps* function in the *GenomicAlignments* package, and the exact code used to produce this object can be seen by typing `vignette("airway")` into the R session, to pull up the *airway* software vignette. There are multiple ways to produce a count table and import it into *DESeq2*, and these are summarized in this section of the RNA-seq gene-level workflow.

```
library("airway")
data("airway")
```

We want to specify that `untrt` is the reference level for the `dex` variable:

```
airway$dex <- relevel(airway$dex, "untrt")
airway$dex
#> [1] untrt trt   untrt trt   untrt trt   untrt trt
#> Levels: untrt trt
```

We can quickly check the millions of fragments that uniquely aligned to the genes (the second argument of `round` tells how many decimal points to keep).

```
round( colSums(assay(airway)) / 1e6, 1 )
#> SRR1039508 SRR1039509 SRR1039512 SRR1039513 SRR1039516 SRR1039517
#>      20.6      18.8      25.3      15.2      24.4      30.8
#> SRR1039520 SRR1039521
#>      19.1      21.2
```

We can inspect the information about the samples, by pulling out the `colData` slot of the *SummarizedExperiment*:

```
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





```

If we had not already loaded *DESeq2*, we would do this, and then create a *DESeqDataSet*. We want to control for the cell line, while testing for differences across dexamethasone treatment, so we use a design of `~ cell + dex`:

```
library("DESeq2")
dds <- DESeqDataSet(airway, design = ~ cell + dex)
```

We will perform a minimal filtering to reduce the size of the dataset. We do not need to retain genes if they do not have a count of 5 or more for 4 or more samples as these genes will have no statistical power to detect differences, and no information to compute distances between samples.

```
keep <- rowSums(counts(dds) >= 5) >= 4

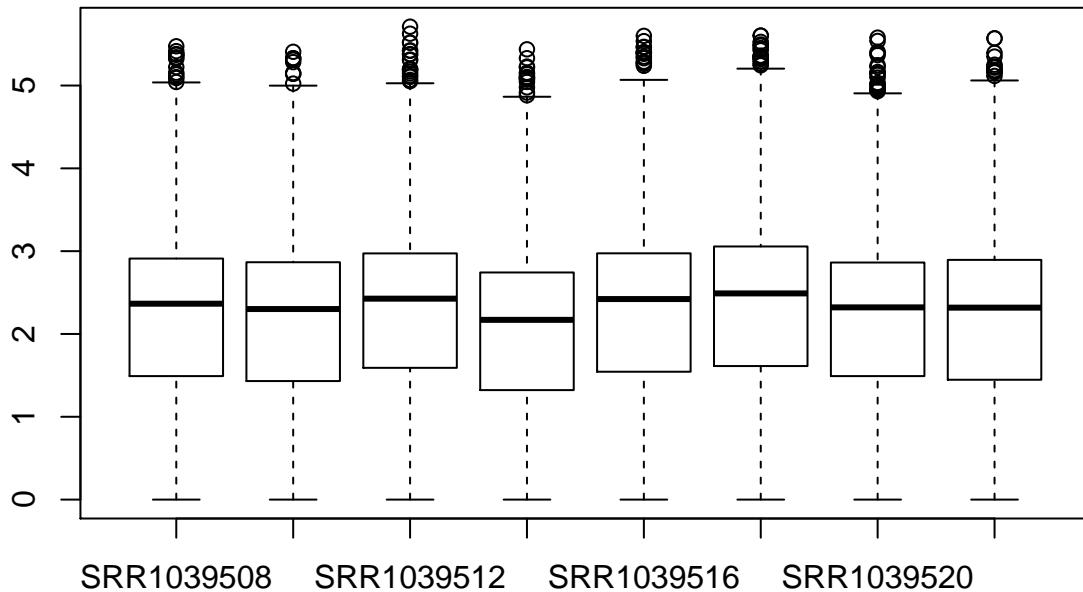




```

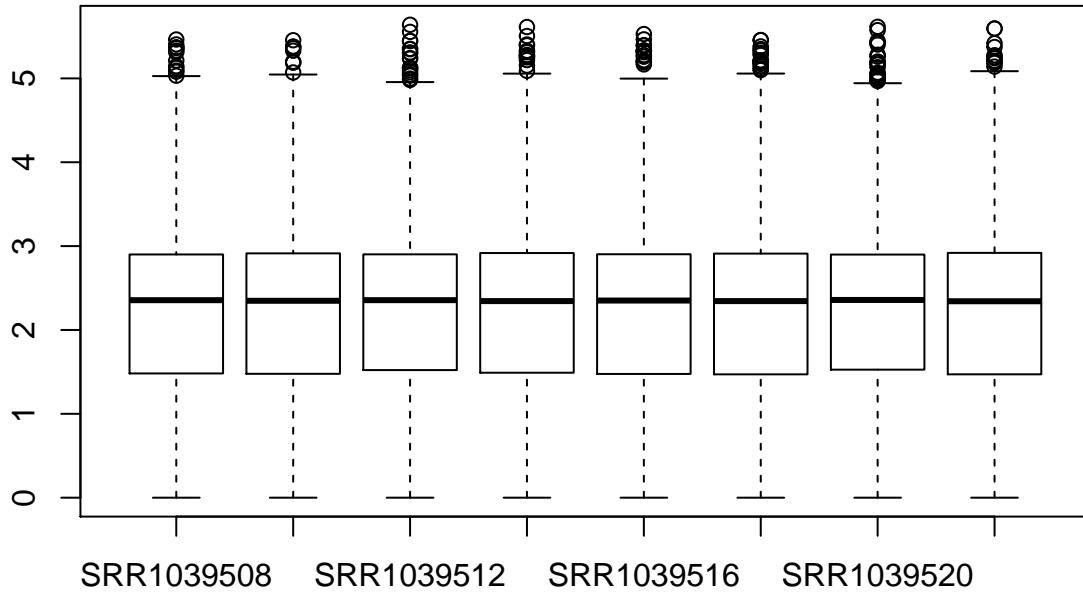
Some very basic exploratory analysis is to examine a boxplot of the counts for each sample. We will take the logarithm so that large counts do not dominate the boxplot:

```
boxplot(log10(counts(dds)+1))
```



The main function in *DESeq2* involves computation of *size factors* which normalize for differences in sequencing depth among samples. We can also compute these size factors manually, so that the *normalized counts* are available for plotting:

```
dds <- estimateSizeFactors(dds)
boxplot(log10(counts(dds,normalized=TRUE)+1))
```



7.4.2 Data transformation for EDA

Taking the logarithm of counts plus a pseudocount of 1 is a common transformation, but it tends to inflate the sampling variance of low counts such that it is even larger than biological variation across groups of samples. In *DESeq2* we therefore provide transformations which produce log-scale data such that the systematic trends have been removed. Our recommended transformation is the variance-stabilizing transformation, or VST, and it can be called with the *vst* function:

```
vsd <- vst(dds)
class(vsd)
#> [1] "DESeqTransform"
#> attr(,"package")
#> [1] "DESeq2"
```

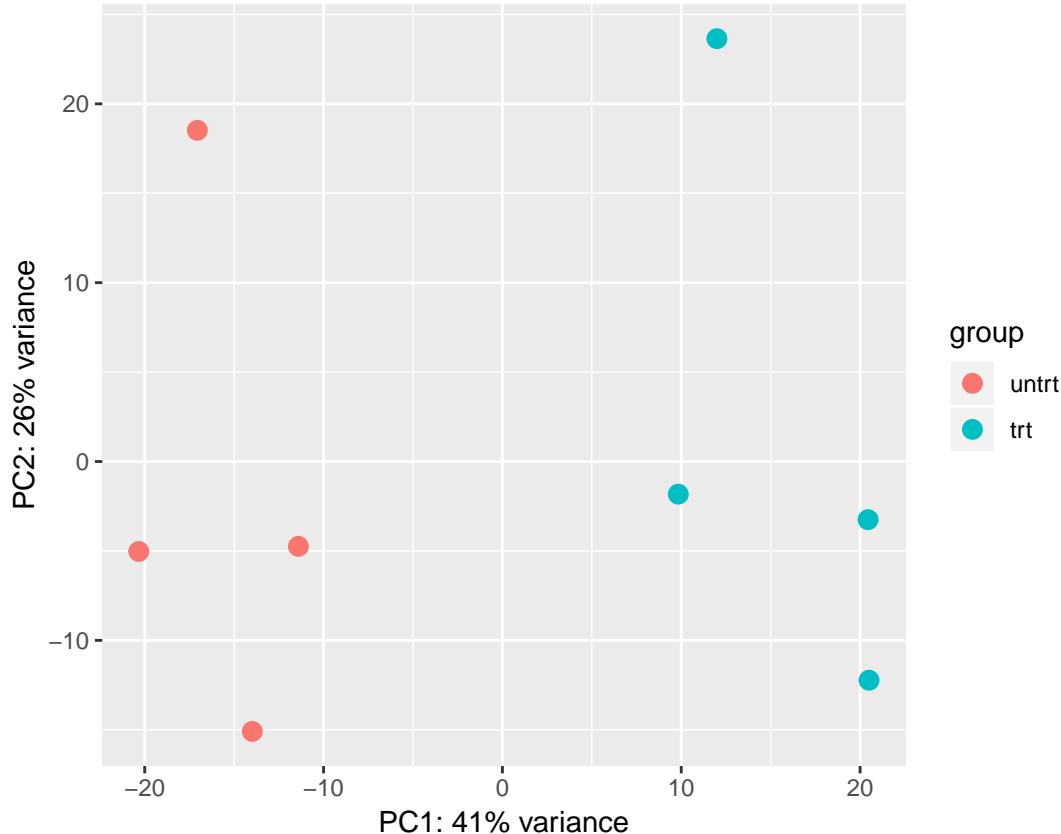
This function does not return a *DESeqDataSet*, because it does not return counts, but instead continuous values (on the log2 scale). We can access the transformed data with *assay*:

```
assay(vsd)[1:3,1:3]
#>           SRR1039508 SRR1039509 SRR1039512
#> ENSG000000000003  9.456925  9.074623  9.608160
#> ENSG000000000419  8.952752  9.262092  9.145782
#> ENSG000000000457  8.193711  8.098664  8.032656
```

7.4.3 Principal components plot

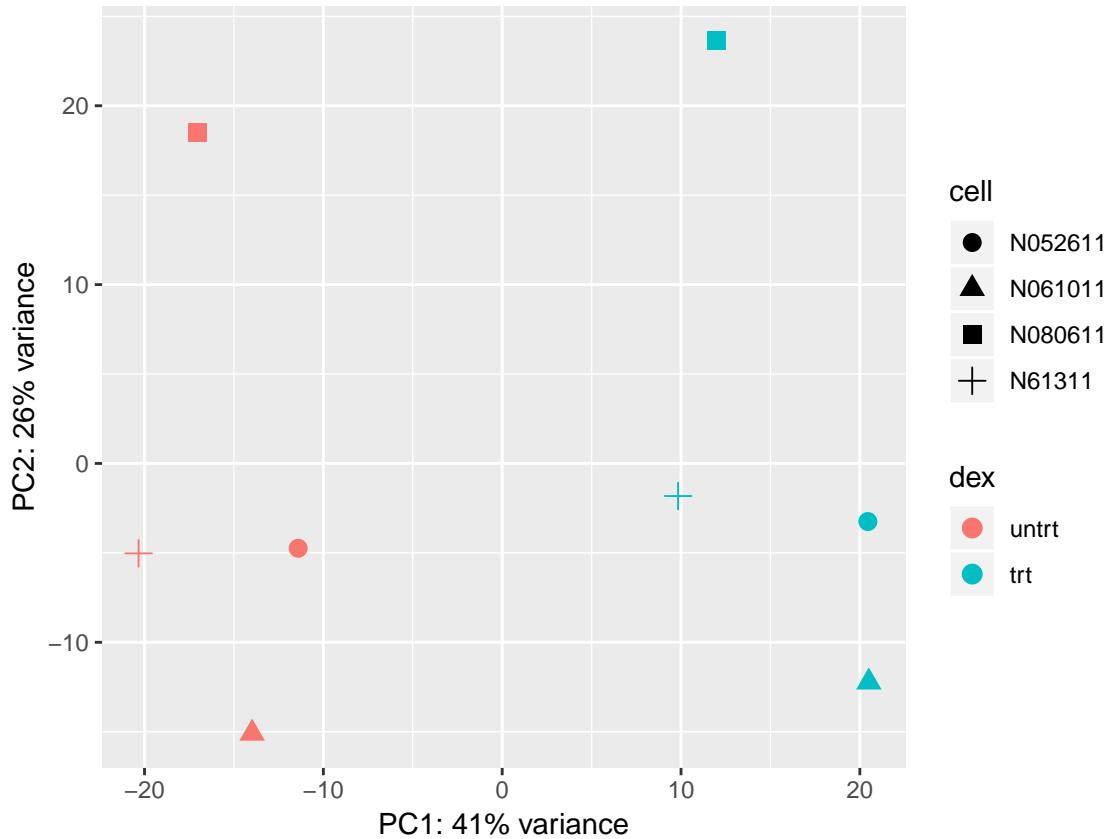
The VST data is appropriate for calculating distances between samples or for performing PCA. More information about PCA and distance calculation can be found in the RNA-seq gene-level workflow. In short, PCA plots allow us to visualize the most dominant axes of variation in our data, which is useful for both quality control, and to get a sense of how large the inter-sample differences are across and within conditions. Here we see that PC1 (the primary axis of variation in the data) separates the treated and untreated samples:

```
plotPCA(vsd, "dex")
```



With some additional *ggplot2* code, we can also indicate which samples belong to which cell line:

```
library("ggplot2")
pcaData <- plotPCA(vsd, intgroup = c( "dex", "cell"), returnData = TRUE)
percentVar <- round(100 * attr(pcaData, "percentVar"))
ggplot(pcaData, aes(x = PC1, y = PC2, color = dex, shape = cell)) +
  geom_point(size =3) +
  xlab(paste0("PC1: ", percentVar[1], "% variance")) +
  ylab(paste0("PC2: ", percentVar[2], "% variance")) +
  coord_fixed()
```



Note that we do not recommend working with the transformed data for the primary differential expression analysis. Instead we will use the original counts and a *generalized linear model* (GLM) which takes into account the expected variance from either low or high counts. For statistical details, please refer to the *DESeq2* methods paper (Love, Huber, and Anders 2014).

7.5 Differential expression analysis

7.5.1 Standard DE steps

Differential expression analysis in *DESeq2* is performed by calling the following two functions:

```
dds <- DESeq(dds)
#> using pre-existing size factors
#> estimating dispersions
#> gene-wise dispersion estimates
#> mean-dispersion relationship
```

```
#> final dispersion estimates
#> fitting model and testing
res <- results(dds)
```

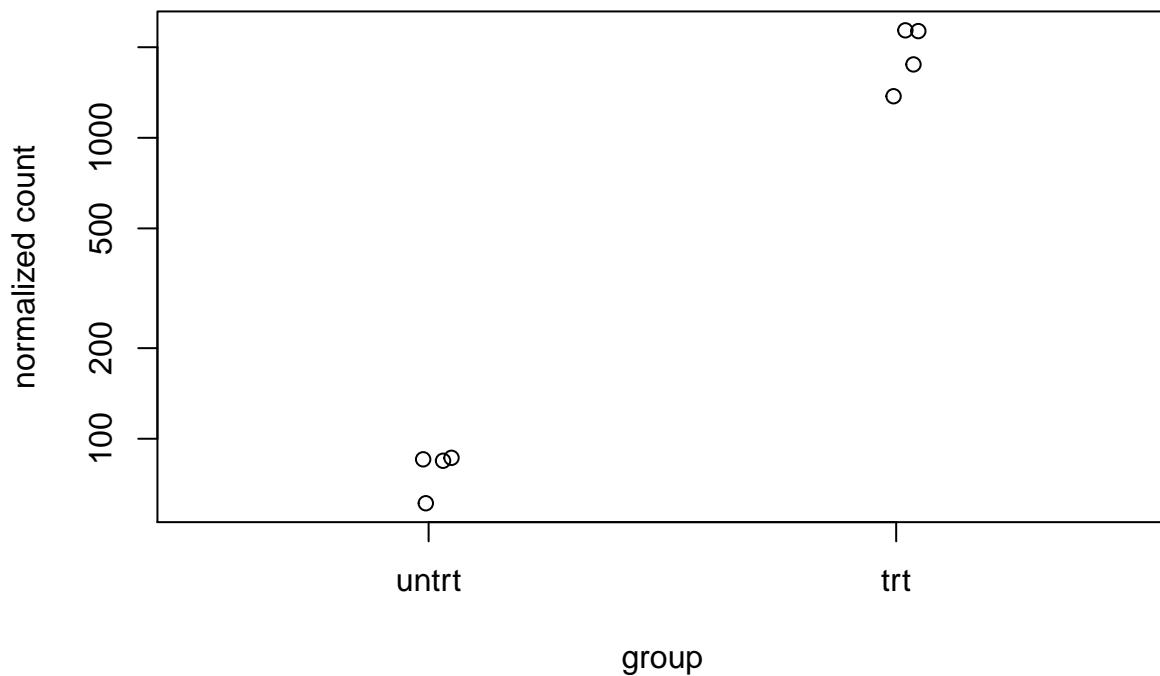
The results table `res` contains the results for each gene (in the same order as in the `DESeqDataSet`). If we want to see the top genes, we can order it like so:

```
head(res[order(res$pvalue),])
#> log2 fold change (MLE): dex trt vs untrt
#> Wald test p-value: dex trt vs untrt
#> DataFrame with 6 rows and 6 columns
#>           baseMean    log2FoldChange      lfcSE
#>           <numeric>     <numeric>     <numeric>
#> ENSG00000152583 997.522193389904 4.57431721934494 0.183934560931089
#> ENSG00000165995 495.289924523775 3.29060376160862 0.132397954572965
#> ENSG00000120129 3409.85238036378 2.94725094348016 0.122471899817052
#> ENSG00000101347 12707.320121355 3.7664043884955 0.156934450326662
#> ENSG00000189221 2342.17328482568 3.35311264853444 0.142537730315705
#> ENSG00000211445 12292.1234547129 3.72983474166181 0.167361554620974
#>           stat        pvalue
#>           <numeric>     <numeric>
#> ENSG00000152583 24.8692643524384 1.60060793892601e-136
#> ENSG00000165995 24.853886695018 2.34741061794036e-136
#> ENSG00000120129 24.0647115614501 5.85598005614604e-128
#> ENSG00000101347 23.9998571419828 2.79035129961566e-127
#> ENSG00000189221 23.5243864281245 2.2964743349727e-122
#> ENSG00000211445 22.2860904352187 5.04142773173702e-110
#>           padj
#>           <numeric>
#> ENSG00000152583 2.03426604150712e-132
#> ENSG00000165995 2.03426604150712e-132
#> ENSG00000120129 3.38319487777077e-124
#> ENSG00000101347 1.20905921812347e-123
#> ENSG00000189221 7.96049863474937e-119
#> ENSG00000211445 1.45630042410777e-106
```

We can plot the counts for the top gene using `plotCounts`:

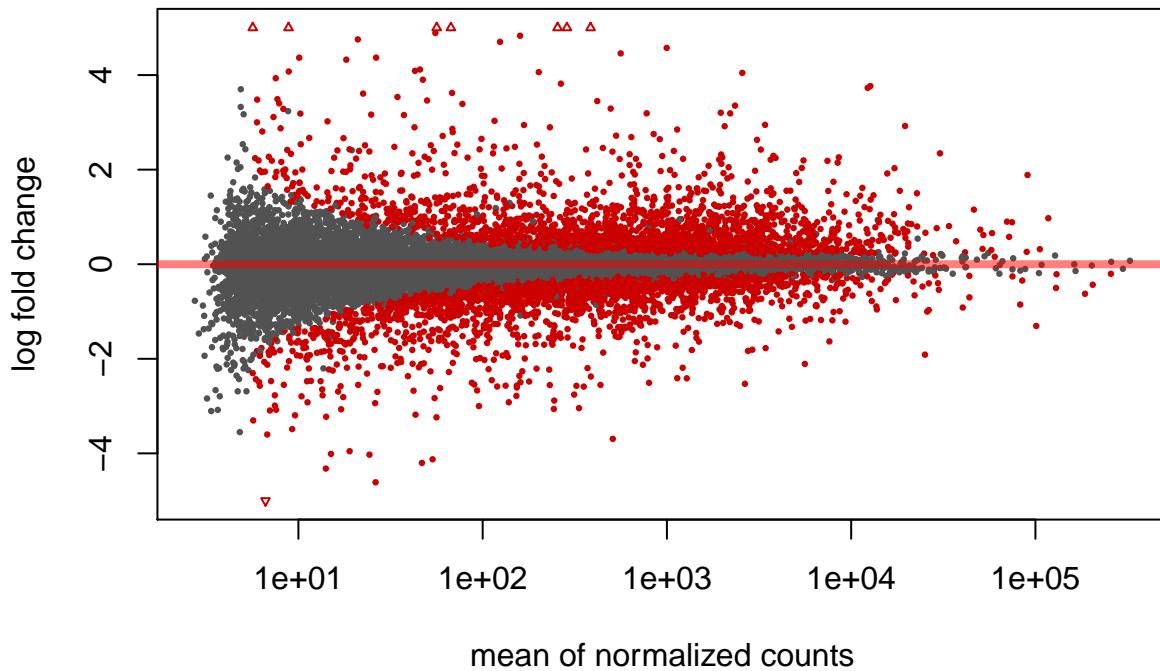
```
plotCounts(dds, which.min(res$pvalue), "dex")
```

ENSG00000152583



We can examine all the log₂ fold changes (LFC) due to dexamethasone treatment over the mean of counts using `plotMA`:

```
plotMA(res, ylim=c(-5,5))
```

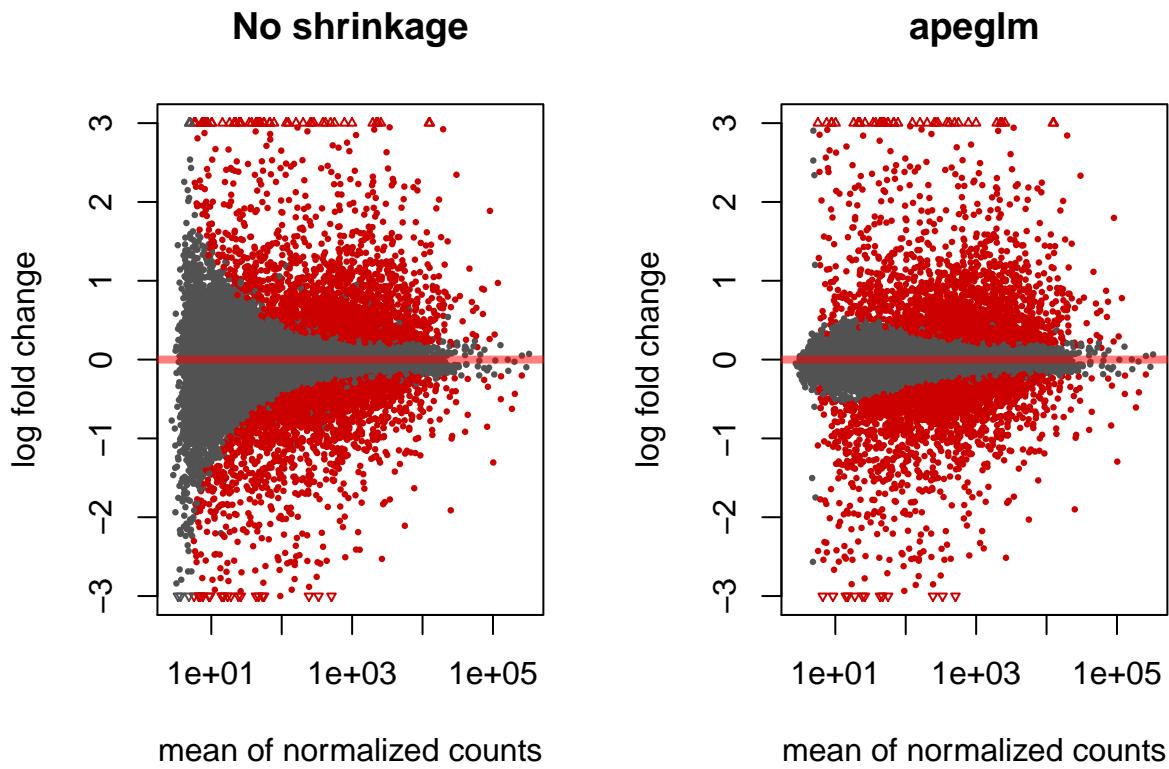


Note that there are many large LFC which are not significant (grey points) on the left side of the MA-plot above. These obtain a large LFC because of the imprecision of log counts. For more informative visualization and more accurate ranking of genes by effect size (the log fold change may sometimes be referred to as an *effect size*), we recommend to use *DESeq2*'s functionality for shrinking LFCs. Our most recent methodological

development is the *apeglm* shrinkage estimator, which is available in *DESeq2*'s *lfcShrink* function:

```
library("apeglm")
resultsNames(dds)
#> [1] "Intercept"           "cell_N061011_vs_N052611"
#> [3] "cell_N080611_vs_N052611" "cell_N61311_vs_N052611"
#> [5] "dex_trt_vs_untrt"
res2 <- lfcShrink(dds, coef="dex_trt_vs_untrt", type="apeglm")
#> using 'apeglm' for LFC shrinkage. If used in published research, please cite:
#>   Zhu, A., Ibrahim, J.G., Love, M.I. (2018) Heavy-tailed prior distributions for
#>   sequence count data: removing the noise and preserving large differences.
#>   bioRxiv. https://doi.org/10.1101/303255

par(mfrow=c(1,2))
plotMA(res, ylim=c(-3,3), main="No shrinkage")
plotMA(res2, ylim=c(-3,3), main="apeglm")
```



7.5.2 Minimum effect size

If we don't want to report as significant genes with small LFC, we can specify a minimum *biologically meaningful* effect size, by choosing an LFC and testing against this. We can either perform such a threshold test using the unshrunken LFCs or the LFCs provided by *lfcShrink* using the *apeglm* method:

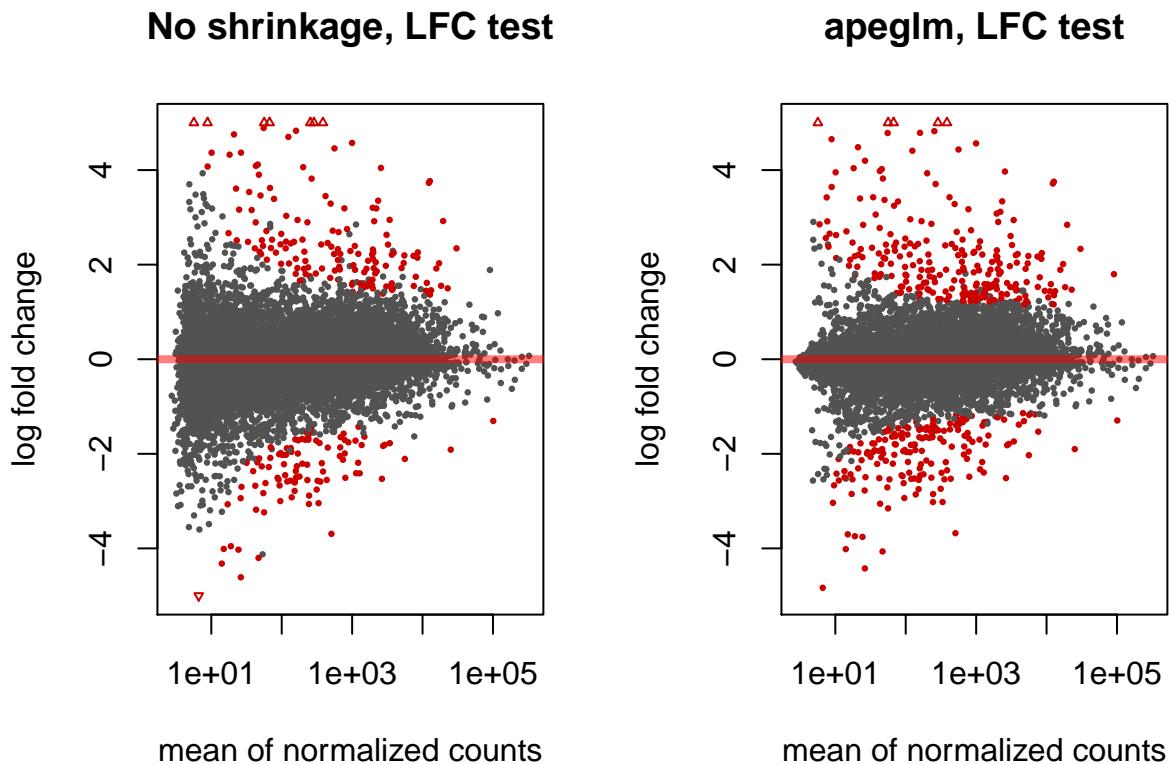
```
res.lfc <- results(dds, lfcThreshold=1)
res.lfc2 <- lfcShrink(dds, coef="dex_trt_vs_untrt", type="apeglm",
                      lfcThreshold=1)
#> using 'apeglm' for LFC shrinkage. If used in published research, please cite:
#>   Zhu, A., Ibrahim, J.G., Love, M.I. (2018) Heavy-tailed prior distributions for
#>   sequence count data: removing the noise and preserving large differences.
```

```
#>     bioRxiv. https://doi.org/10.1101/303255
#> computing FSOS 'false sign or small' s-values (T=1)
```

Note that *testing* against an LFC threshold is not equivalent to testing against a null hypothesis of 0 and then filtering on LFC values. We prefer the former, as discussed in Love, Huber, and Anders (2014) and Zhu, Ibrahim, and Love (2018).

The *apeglm* method provides s-values (Stephens 2016) when `svalue=TRUE` or when we supply a minimum effect size as above. These are analogous to q-values or adjusted p-values, in that the genes with s-values less than α should have an aggregate rate of false sign or being smaller in absolute value than our given LFC threshold, which is bounded by α .

```
par(mfrow=c(1,2))
plotMA(res.lfc, ylim=c(-5,5), main="No shrinkage, LFC test")
plotMA(res.lfc2, ylim=c(-5,5), main="apeglm, LFC test", alpha=0.01)
```



7.6 AnnotationHub

7.6.1 Querying AnnotationHub

We will use the *AnnotationHub* package to attach additional information to the results table. *AnnotationHub* provides an easy-to-use interface to more than 40,000 annotation records. A record may be peaks from a ChIP-seq experiment from ENCODE, the sequence of the human genome, a *TxDb* containing information about transcripts and genes, or an *OrgDb* containing general information about biological identifiers for a particular organism.

```
library("AnnotationHub")
#>
#> Attaching package: 'AnnotationHub'
```

```
#> The following object is masked from 'package:Biobase':
#>
#>   cache
ah <- AnnotationHub()
#> snapshotDate(): 2018-06-27
```

The following code chunk, un-evaluated here, launches a browser for navigating all the records available through *AnnotationHub*.

```
display(ah)
```

We can also query using keywords with the *query* function:

```
query(ah, c("OrgDb", "Homo sapiens"))
#> AnnotationHub with 1 record
#> # snapshotDate(): 2018-06-27
#> # names(): AH61777
#> # $dataprov...: ftp://ftp.ncbi.nlm.nih.gov/gene/DATA/
#> # $species: Homo sapiens
#> # $rdataclass: OrgDb
#> # $rdataadateadded: 2018-04-19
#> # $title: org.Hs.eg.db.sqlite
#> # $description: NCBI gene ID based annotations about Homo sapiens
#> # $taxonid: 9606
#> # $genome: NCBI genomes
#> # $sourcetype: NCBI/ensembl
#> # $sourceurl: ftp://ftp.ncbi.nlm.nih.gov/gene/DATA/, ftp://ftp.ensembl.....
#> # $sourcesize: NA
#> # $tags: c("NCBI", "Gene", "Annotation")
#> # retrieve record with 'object[["AH61777"]]'
```

To pull down a particular record we use double brackets and the *name* of the record:

```
hs <- ah[["AH61777"]]
#> downloading 0 resources
#> loading from cache
#>   '/home/mramos//.AnnotationHub/68523'
hs
#> OrgDb object:
#> / DBSCHEMAVERSION: 2.1
#> / Db type: OrgDb
#> / Supporting package: AnnotationDbi
#> / DBSCHEMA: HUMAN_DB
#> / ORGANISM: Homo sapiens
#> / SPECIES: Human
#> / EGSRCDATE: 2018-Apr4
#> / EGSRCENAME: Entrez Gene
#> / EGSRCEURL: ftp://ftp.ncbi.nlm.nih.gov/gene/DATA
#> / CENTRALID: EG
#> / TAXID: 9606
#> / GOSRCENAME: Gene Ontology
#> / GOSRCEURL: ftp://ftp.geneontology.org/pub/go/godatabase/archive/latest-lite/
#> / GOSRCDATE: 2018-Mar28
#> / GOEGSRCDATE: 2018-Apr4
#> / GOEGSRCENAME: Entrez Gene
#> / GOEGSRCEURL: ftp://ftp.ncbi.nlm.nih.gov/gene/DATA
```

```
#> / KEGGSOURCENAME: KEGG GENOME
#> / KEGGSOURCEURL: ftp://ftp.genome.jp/pub/kegg/genomes
#> / KEGGSOURCEDATE: 2011-Mar15
#> / GPSOURCENAME: UCSC Genome Bioinformatics (Homo sapiens)
#> / GPSOURCEURL:
#> / GPSOURCEDATE: 2018-Mar26
#> / ENSOURCEDATE: 2017-Dec04
#> / ENSOURCENAME: Ensembl
#> / ENSOURCEURL: ftp://ftp.ensembl.org/pub/current_fasta
#> / UPSOURCENAME: Uniprot
#> / UPSOURCEURL: http://www.UniProt.org/
#> / UPSOURCEDATE: Mon Apr  9 20:58:54 2018
#>
#> Please see: help('select') for usage information
```

7.6.2 Mapping IDs

The *rownames* of the results table are Ensembl IDs, and most of these are entries in *OrgDb* (although thousands are not).

```
columns(hs)
#> [1] "ACCCNUM"        "ALIAS"          "ENSEMBL"         "ENSEMLPROT"
#> [5] "ENSEMBLTRANS"   "ENTREZID"       "ENZYME"         "EVIDENCE"
#> [9] "EVIDENCEALL"    "GENENAME"       "GO"              "GOALL"
#> [13] "IPI"           "MAP"            "OMIM"           "ONTOLOGY"
#> [17] "ONTOLOGYALL"   "PATH"           "PFAM"           "PMID"
#> [21] "PROSITE"       "REFSEQ"         "SYMBOL"         "UCSCKG"
#> [25] "UNIGENE"       "UNIPROT"
table(rownames(res) %in% keys(hs, "ENSEMBL"))
#>
#> FALSE  TRUE
#> 3492 14540
```

We can use the *mapIds* function to add gene symbols, using ENSEMBL as the keytype, and requesting the column SYMBOL.

```
res$symbol <- mapIds(hs, rownames(res), column="SYMBOL", keytype="ENSEMBL")
#> 'select()' returned 1:many mapping between keys and columns
head(res)
#> log2 fold change (MLE): dex trt vs untrt
#> Wald test p-value: dex trt vs untrt
#> DataFrame with 6 rows and 7 columns
#>                                baseMean      log2FoldChange      lfcSE
#>                                <numeric>      <numeric>      <numeric>
#> ENSG000000000003 708.84032163709 -0.38188890052585 0.100800595908449
#> ENSG00000000419 520.444343803335 0.206203578138288 0.111340654960802
#> ENSG00000000457 237.237392013978 0.0373231179429298 0.140524882321204
#> ENSG00000000460 57.9518862998956 -0.0907678445557818 0.276878121334175
#> ENSG00000000971 5819.01711439455 0.425781621579506 0.0897314693828554
#> ENSG000000001036 1282.59042750161 -0.241675199658759 0.0898743244338321
#>                                stat      pvalue      padj
#>                                <numeric>      <numeric>      <numeric>
#> ENSG000000000003 -3.78855796519988 0.000151524241731238 0.00121584173966936
```

```
#> ENSG00000000419 1.85200615364516 0.0640249392675146 0.184546856375281
#> ENSG00000000457 0.265597930604338 0.790548879512378 0.903592643032674
#> ENSG00000000460 -0.32782599115598 0.743043234238656 0.878207188857688
#> ENSG000000000971 4.74506463014478 2.08439772491752e-06 2.5495258552061e-05
#> ENSG00000001036 -2.68903495165281 0.00716589158352556 0.0334768821902062
#>                               symbol
#>                               <character>
#> ENSG000000000003      TSPAN6
#> ENSG00000000419       DPM1
#> ENSG00000000457       SCYL3
#> ENSG00000000460       C1orf112
#> ENSG000000000971      CFH
#> ENSG00000001036       FUCA2
```

7.7 Building reports

7.7.1 ReportingTools

There are many packages for building interactive reports from Bioconductor. Two of these are *ReportingTools* and *Glimma*, which both provide HTML reports that allow for collaborators to examine the top genes (or whatever features of interest) from a genomic analysis.

The code for compiling a *ReportingTools* report is:

```
library("ReportingTools")
#> Loading required package: knitr
#>
#>
tmp <- tempdir() # you would instead use a meaningful path here
rep <- HTMLReport(shortName="airway", title="Airway DGE",
                  basePath=tmp, reportDirectory="report")
publish(res, rep, dds, n=20, make.plots=TRUE, factor=dds$dex)
finish(rep)
#> [1] "/tmp/Rtmp5eNWI7/report/airway.html"
```

This last line, un-evaluated would launch the report in a web browser:

```
browseURL(file.path(tmp, "report", "airway.html"))
```

7.7.2 Glimma

Another package which can generate interactive reports is *Glimma*. The *glMDPlot* constructs an interactive MA-plot where hovering over a gene in the MA-plot on the left side will display the counts for the samples on the right hand side. Clicking will bring up the gene's information in a tooltip and in a list at the bottom of the screen. Hovering on a sample on the right hand side will give the sample ID in a tooltip.

```
library("Glimma")
status <- as.numeric(res$padj < .1)
anno <- data.frame(GeneID=rownames(res), symbol=res$symbol)
glMDPlot(res2, status=status, counts=counts(dds, normalized=TRUE),
          groups=dds$dex, transform=FALSE,
          samples=colnames(dds), anno=anno,
          path=tmp, folder="glimma", launch=FALSE)
```

This last line would launch the report in a web browser:

```
browseURL(file.path(tmp, "glimma", "MD-Plot.html"))
```

7.8 Integration with *ZINB-WaVE*

7.8.1 Simulate with *splatter*

In this last section, we show that *DESeq2* can be integrated with another Bioconductor package *zinbwave* (Risso et al. 2018a) in order to model and account for additional zeros (more than expected by the Negative Binomial model). This can be useful for single cell RNA-seq experiments.

Here we use the *splatter* package to simulate single-cell RNA-seq data (Zappia, Phipson, and Oshlack 2017). We then use the methods defined in Van den Berge et al. (2018) to combine *zinbwave* observation weights with *DESeq2* modeling of negative binomial counts.

From Van den Berge et al. (2018):

It is important to note that while methods such as ZINB-WaVE and ZINGER can successfully identify excess zeros, they cannot, however, readily discriminate between their underlying causes, i.e., between technical (e.g., dropout) and biological (e.g., bursting) zeros.

The above note implies that the zero-inflation weighting approach outlined below can be used when the interesting signal is not in the zero component. That is, if you wanted to find biological differences in transcriptional bursting across groups of cells, the below approach would not help you find these differences. It instead helps to uncover differences in counts besides the zero component (whether those zeros be biological or technical).

7.8.2 Simulate single-cell count data with *splatter*

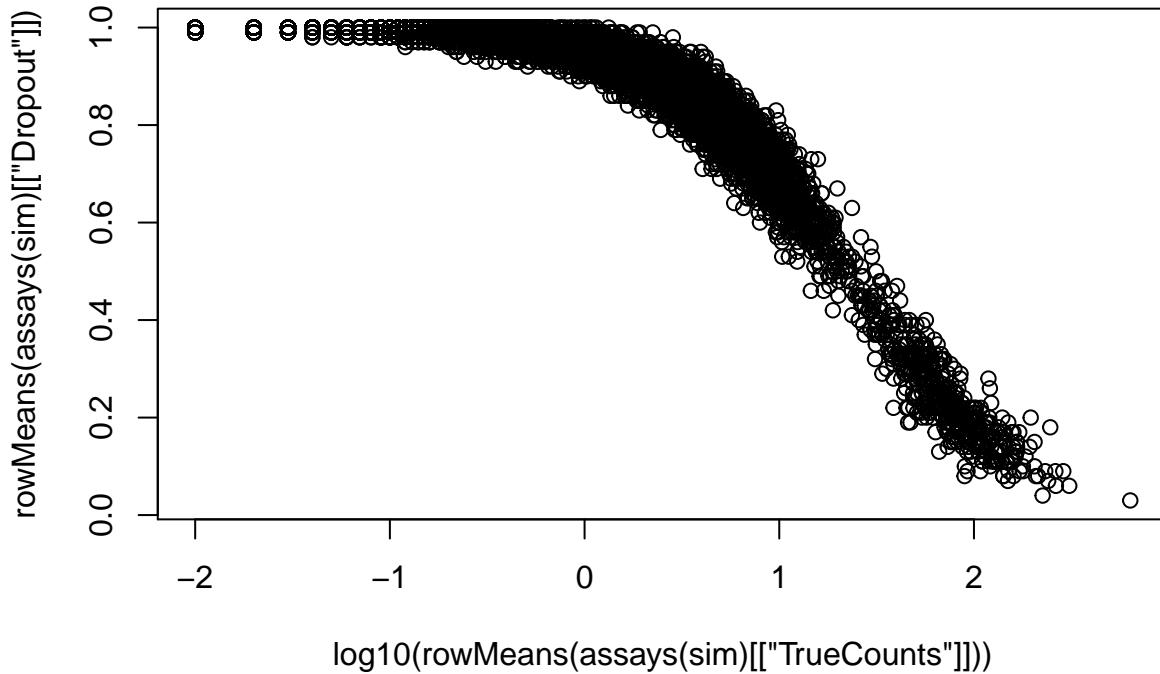
The following chunks of code create a *splatter* simulation:

```
library("splatter")
#> Loading required package: SingleCellExperiment
params <- newSplatParams()
params <- setParam(params, "de.facLoc", 1)
params <- setParam(params, "de.facScale", .25)
params <- setParam(params, "dropout.type", "experiment")
params <- setParam(params, "dropout.mid", 3)

set.seed(1)
sim <- splatSimulate(params, group.prob=c(.5,.5), method="groups")
#> Getting parameters...
#> Creating simulation object...
#> Simulating library sizes...
#> Simulating gene means...
#> Simulating group DE...
#> Simulating cell means...
#> Simulating BCV...
#> Simulating counts...
#> Simulating dropout (if needed)...
#> Done!
```

We can plot the amount of dropouts over the true counts:

```
plot(log10(rowMeans(assays(sim)[["TrueCounts"]])),
     rowMeans(assays(sim)[["Dropout"]]))
```

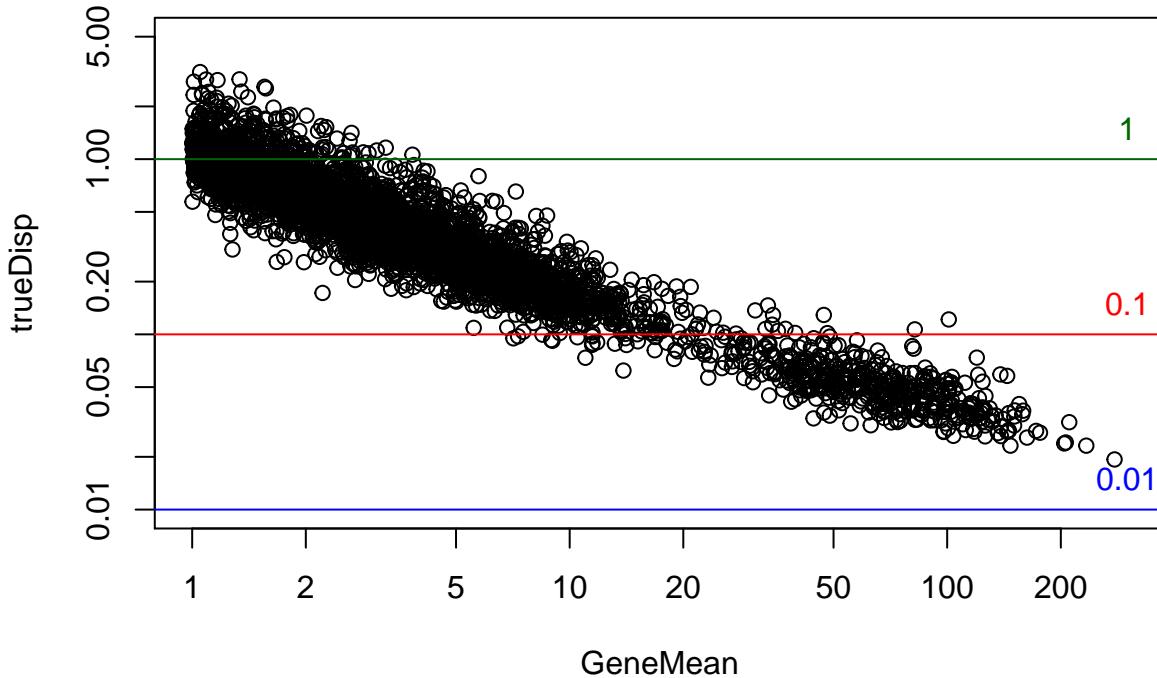


We will store the true log2 fold change for comparison:

```
rowData(sim)$log2FC <- with(rowData(sim), log2(DEFacGroup2/DEFacGroup1))
```

The true dispersion for the Negative Binomial component, over the mean:

```
rowData(sim)$trueDisp <- rowMeans(assays(sim)[["BCV"]])^2
gridlines <- c(1e-2, 1e-1, 1); cols <- c("blue", "red", "darkgreen")
with(rowData(sim)[rowData(sim)$GeneMean > 1,],
     plot(GeneMean, trueDisp, log="xy", xlim=c(1,300), ylim=c(.01,5)))
abline(h=gridlines, col=cols)
text(300, gridlines, labels=gridlines, col=cols, pos=3)
```



7.8.3 Model zeros with *zinbwave*

The following code subsets the dataset and creates a `condition` variable that we will use to test for differential expression:

```
library(zinbwave)
keep <- rowSums(counts(sim) >= 5) >= 25
table(keep)
#> keep
#> FALSE TRUE
#> 9025 975
zinb <- sim[keep,]
zinb$condition <- factor(zinb$Group)
```

We need to re-arrange the assays in the `zinb` object such that "counts" is the first assay:

```
nms <- c("counts", setdiff(assayNames(zinb), "counts"))
assays(zinb) <- assays(zinb)[nms]
```

Finally we fit the *ZINB-WAVE* model. See `?zinbwave` and the `zinbwave` vignette for more details, including options on parallelization. It runs in less than a minute on this simulated dataset (with not so many cells).

```
zinb <- zinbwave(zinb, K=0, BPPARAM=SerialParam(), epsilon=1e12)
```

7.8.4 Model non-zeros with *DESeq2*

Now we import the `zinb` object using `DESeqDataSet` (which works because the `SingleCellExperiment` object builds on top of the `SummarizedExperiment`). All of the simulation information comes along in the metadata columns of the object.

Van den Berge et al. (2018) and others have shown the LRT may perform better for null hypothesis testing, so we use the LRT. In order to use the Wald test, it is recommended to set `useT=TRUE`.

```

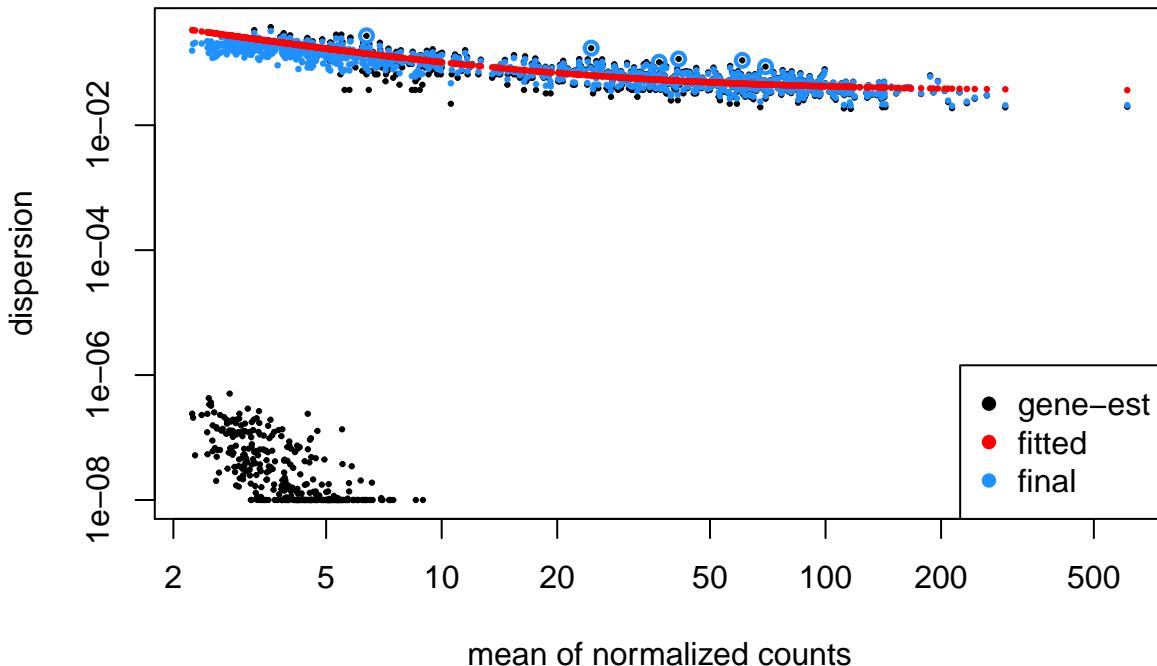
zdds <- DESeqDataSet(zinb, design=~condition)
zdds <- DESeq(zdds, test="LRT", reduced=~1,
               sfType="poscounts", minmu=1e-6, minRep=Inf)
#> estimating size factors
#> estimating dispersions
#> gene-wise dispersion estimates
#> mean-dispersion relationship
#> final dispersion estimates
#> fitting model and testing

```

7.8.5 Plot dispersion estimates

It is recommended to plot the dispersion estimates for *DESeq2* on single-cell data. As discussed in the *DESeq2* paper, it becomes difficult to accurately estimate the dispersion when the counts are very small, because the Poisson component of the variance is dominant. Therefore we see some very low dispersion estimates here, although the trend is still accurately capturing the upper proportion. So here everything looks good.

```
plotDispEts(zdds)
```

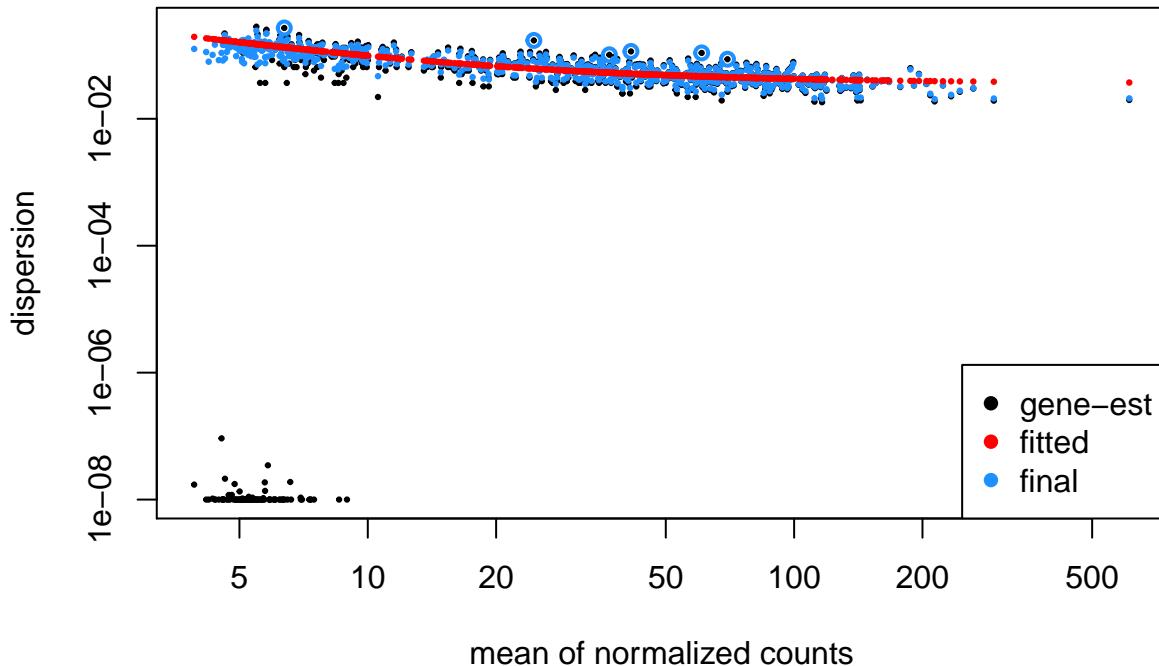


If the parametric trend fails to fit (there would be a warning in this case), one should check the dispersion plot as above. If it looks like the dispersion fit is being thrown off by the low count genes with low dispersion estimates at the bottom of the plot, there is a relatively easy solution: one can filter out more of the low count genes only for the dispersion estimation step, so that the trend still captures the upper portion. This is pretty easy to do in *DESeq2*, to filter genes solely for the dispersion trend estimation, but to use a larger set for the rest of the analysis. An example of how this can be done:

```

keepForDispTrend <- rowSums(counts(zdds) >= 10) >= 25
zdds2 <- estimateDispersionsFit(zdds[keepForDispTrend,])
plotDispEts(zdds2)

```



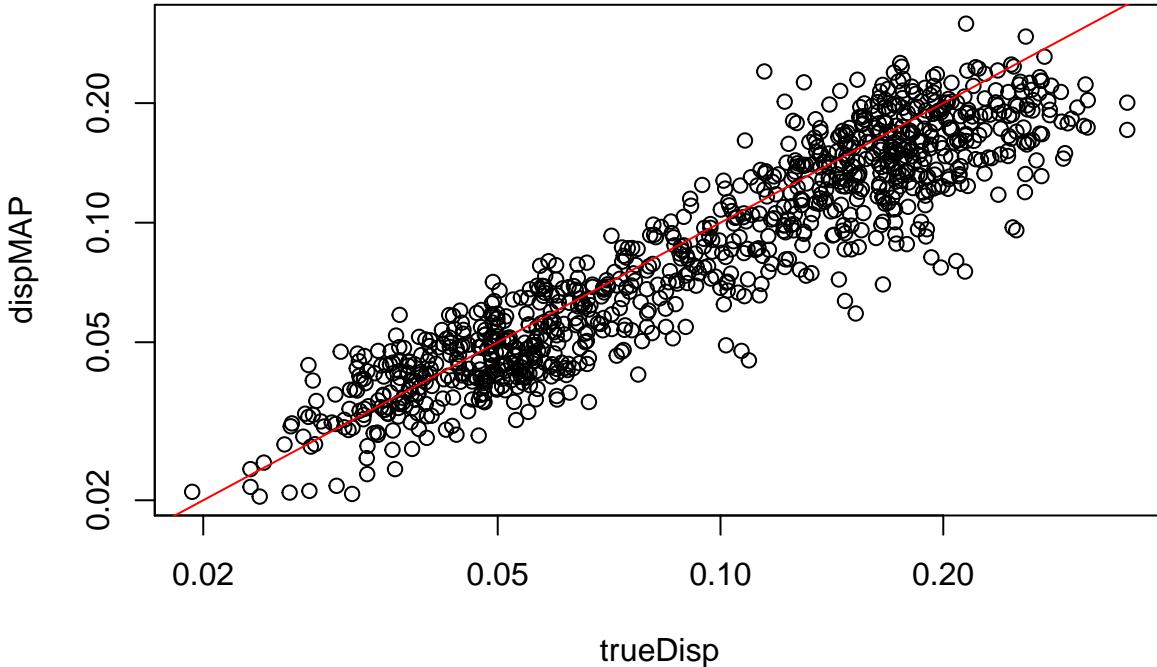
One would then assign the dispersion function to the original dataset, re-estimate final dispersions, check `plotDispEsts`, and then either re-run the Wald or LRT function:

```
dispersionFunction(zdds) <- dispersionFunction(zdds2)
zdds <- estimateDispersionsMAP(zdds)
#> found already estimated dispersions, removing these
zdds <- nbinomLRT(zdds, reduced=~1, minmu=1e-6)
#> found results columns, replacing these
```

7.8.6 Evaluation against truth

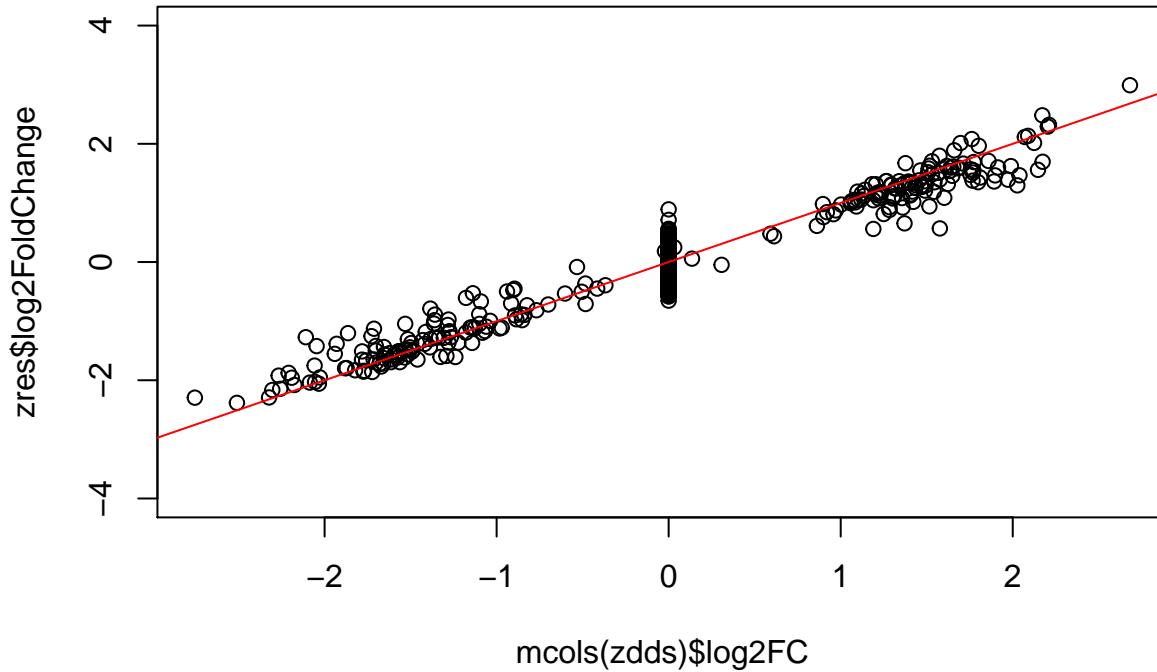
Compare dispersion on the non-zero-component counts to the true value used for simulation.

```
with(mcols(zdds), plot(trueDisp, dispMAP, log="xy"))
abline(0,1,col="red")
```



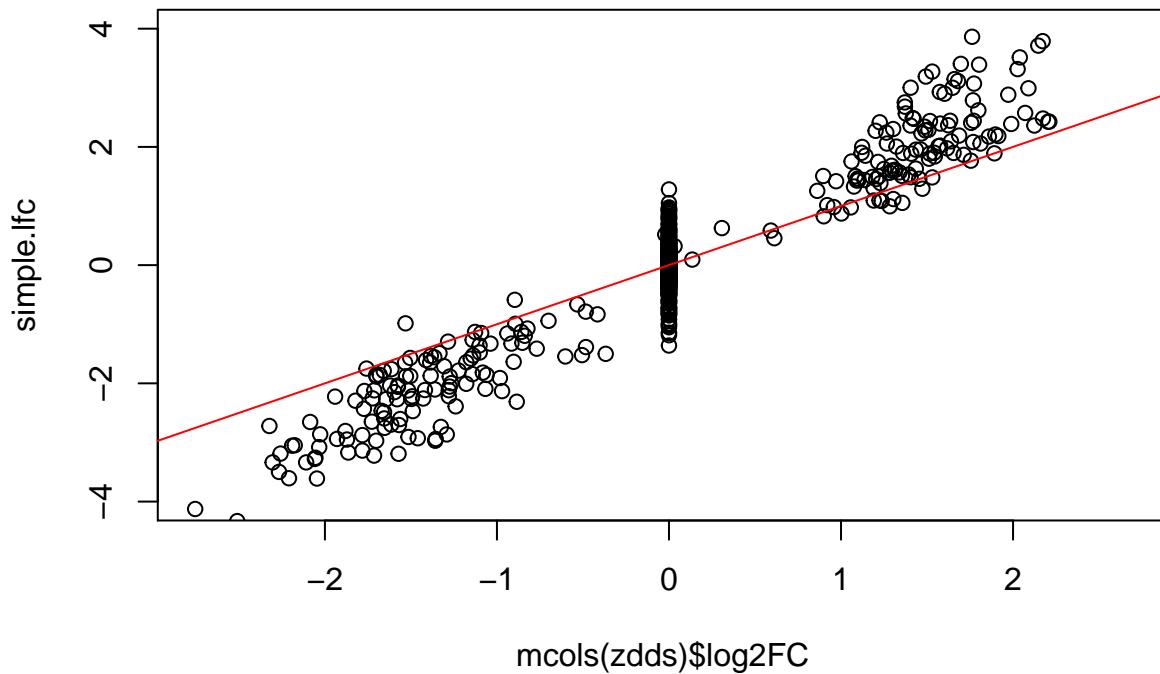
Extract results table:

```
zres <- results(zdds, independentFiltering=FALSE)
plot(mcols(zdds)$log2FC, zres$log2FoldChange, ylim=c(-4,4)); abline(0,1,col="red")
```



Below we show that the “simple” LFC does not work - it over-estimates the true DE LFC because of the dropout zeros in the group with the smaller mean. It also has a lot of noise for the null genes.

```
ncts <- counts(zdds, normalized=TRUE)
simple.lfc <- log2(rowMeans(ncts[,zdds$condition == "Group2"])/
                     rowMeans(ncts[,zdds$condition == "Group1"]))
plot(mcols(zdds)$log2FC, simple.lfc, ylim=c(-4,4)); abline(0,1,col="red")
```



How well do we do in null hypothesis testing:

```
tab <- table(sig=zres$padj < .05, DE.status=mcols(zdds)$log2FC != 0)
tab
#>      DE.status
#> sig    FALSE TRUE
#> FALSE   727  22
#> TRUE     8 218
round(prop.table(tab, 1), 3)
#>      DE.status
#> sig    FALSE  TRUE
#> FALSE 0.971 0.029
#> TRUE  0.035 0.965
```

Chapter 8

202: Analysis of single-cell RNA-seq data: Dimensionality reduction, clustering, and lineage inference

Authors: Diya Das¹, Kelly Street², Davide Risso³ Last modified: 28 June, 2018.

8.1 Overview

8.1.1 Description

This workshop will be presented as a lab session (brief introduction followed by hands-on coding) that instructs participants in a Bioconductor workflow for the analysis of single-cell RNA-sequencing data, in three parts:

1. dimensionality reduction that accounts for zero inflation, over-dispersion, and batch effects
2. cell clustering that employs a resampling-based approach resulting in robust and stable clusters
3. lineage trajectory analysis that uncovers continuous, branching developmental processes

We will provide worked examples for lab sessions, and a set of stand-alone notes in this repository.

Note: A previous version of this workshop was well-attended at BioC 2017, but the tools presented have been significantly updated for interoperability (most notably, through the use of the `SingleCellExperiment` class), and we have been receiving many requests to provide an updated workflow. We plan to incorporate feedback from this workshop into a revised version of our F1000 Workflow.

8.1.2 Pre-requisites

We expect basic knowledge of R syntax. Some familiarity with S4 objects may be helpful, but not required. More importantly, participants should be familiar with the concept and design of RNA-sequencing experiments. Direct experience with single-cell RNA-seq is not required, and the main challenges of single-cell RNA-seq compared to bulk RNA-seq will be illustrated.

¹University of California at Berkeley, Berkeley, CA, USA

²University of California at Berkeley, Berkeley, CA, USA

³Weill Cornell Medicine, New York, NY, USA

8.1.3 Participation

This will be a hands-on workshop, in which each student, using their laptop, will analyze a provided example datasets. The workshop will be a mix of example code that the instructors will show to the students (available through this repository) and short exercises.

8.1.4 *R / Bioconductor* packages used

1. *zinbwave* : <https://bioconductor.org/packages/zinbwave>
2. *clusterExperiment*: <https://bioconductor.org/packages/clusterExperiment>
3. *slingshot*: <https://bioconductor.org/packages/slingshot>

8.1.5 Time outline

Activity	Time
Intro to single-cell RNA-seq analysis	15m
zinbwave (dimensionality reduction)	30m
clusterExperiment (clustering)	30m
slingshot (lineage trajectory analysis)	30m
Questions / extensions	15m

8.1.6 Workshop goals and objectives

Learning goals

- describe the goals of single-cell RNA-seq analysis
- identify the main steps of a typical single-cell RNA-seq analysis
- evaluate the results of each step in terms of model fit
- synthesize results of successive steps to interpret biological significance and develop biological models
- apply this workflow to carry out a complete analysis of other single-cell RNA-seq datasets

Learning objectives

- compute and interpret low-dimensional representations of single-cell data
- identify and remove sources of technical variation from the data
- identify sub-populations of cells (clusters) and evaluate their robustness
- infer lineage trajectories corresponding to differentiating cells
- order cells by developmental “pseudotime”
- identify genes that play an important role in cell differentiation

8.2 Getting started

The workflow presented in this workshop consists of four main steps:

1. dimensionality reduction accounting for zero inflation and over-dispersion and adjusting for gene and cell-level covariates, using the **zinbwave** Bioconductor package;
2. robust and stable cell clustering using resampling-based sequential ensemble clustering, as implemented in the **clusterExperiment** Bioconductor package;
3. inference of cell lineages and ordering of the cells by developmental progression along lineages, using the **slingshot** R package; and
4. DE analysis along lineages.

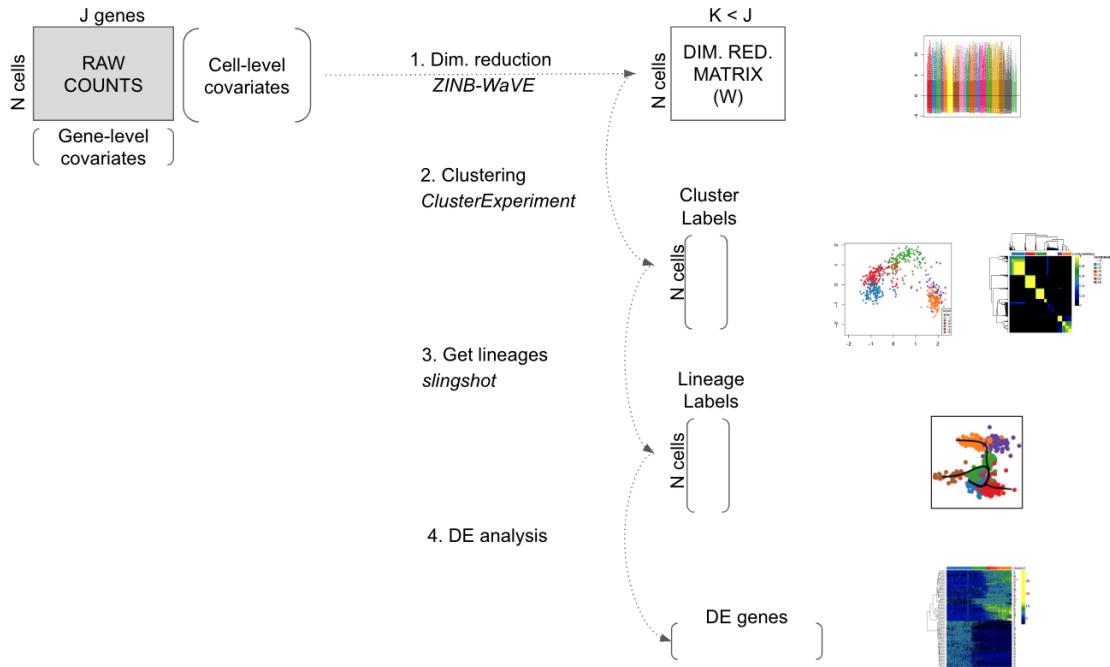


Figure 8.1: Workflow for analyzing scRNA-seq datasets. On the right, main plots generated by the workflow.

Throughout the workflow, we use a single `SingleCellExperiment` object to store the scRNA-seq data along with any gene or cell-level metadata available from the experiment.

8.2.1 The data

This workshop uses data from a scRNA-seq study of stem cell differentiation in the mouse olfactory epithelium (OE) (Fletcher et al. 2017). The olfactory epithelium contains mature olfactory sensory neurons (mOSN) that are continuously renewed in the epithelium via neurogenesis through the differentiation of globose basal cells (GBC), which are the actively proliferating cells in the epithelium. When a severe injury to the entire tissue happens, the olfactory epithelium can regenerate from normally quiescent stem cells called horizontal basal cells (HBC), which become activated to differentiate and reconstitute all major cell types in the epithelium.

The scRNA-seq dataset we use as a case study was generated to study the differentiation of HBC stem cells into different cell types present in the olfactory epithelium. To map the developmental trajectories of the multiple cell lineages arising from HBCs, scRNA-seq was performed on FACS-purified cells using the Fluidigm C1 microfluidics cell capture platform followed by Illumina sequencing. The expression level of each gene in a given cell was quantified by counting the total number of reads mapping to it. Cells were then assigned to different lineages using a statistical analysis pipeline analogous to that in the present workflow. Finally, results were validated experimentally using *in vivo* lineage tracing. Details on data generation and statistical methods are available in (Fletcher et al. 2017; Risso et al. 2018b; Street et al. 2018; Risso et al. 2018a).

In this workshop, we describe a sequence of steps to recover the lineages found in the original study, starting from the genes x cells matrix of raw counts publicly-available at <https://www.ncbi.nlm.nih.gov/geo/query/acc.cgi?acc=GSE95601>.

The following packages are needed.

```
suppressPackageStartupMessages({
  # Bioconductor
  library(BiocParallel)
```

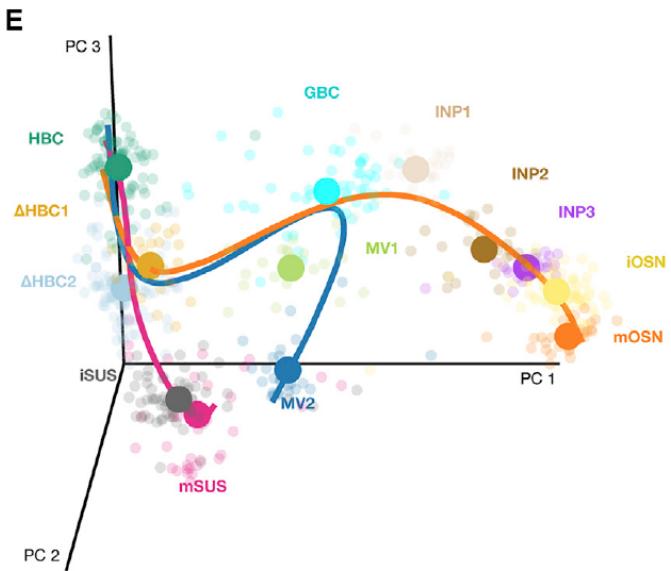


Figure 8.2: Stem cell differentiation in the mouse olfactory epithelium. This figure was reproduced with kind permission from Fletcher et al. (2017).

```

library(SingleCellExperiment)
library(clusterExperiment)
library(scone)
library(zinbwave)
library(slingshot)
# CRAN
library(gam)
library(RColorBrewer)
})
#> Warning: replacing previous import 'SingleCellExperiment::weights' by
#> 'stats::weights' when loading 'slingshot'
#> Warning in rgl.init(initValue, onlyNULL): RGL: unable to open X11 display
#> Warning: 'rgl_init' failed, running with rgl.useNULL = TRUE
set.seed(20)

```

8.2.2 Parallel computing

The `BiocParallel` package can be used to allow for parallel computing in `zinbwave`. Here, we use a single CPU to run the function, registering the serial mode of `BiocParallel`. Users that have access to more than one core in their system are encouraged to use multiple cores to increase speed.

```
register(SerialParam())
```

8.3 The `SingleCellExperiment` class

Counts for all genes in each cell are available as part of the GitHub R package `drisso/fletcher2017data`. Before filtering, the dataset has 849 cells and 28,361 detected genes (i.e., genes with non-zero read counts).

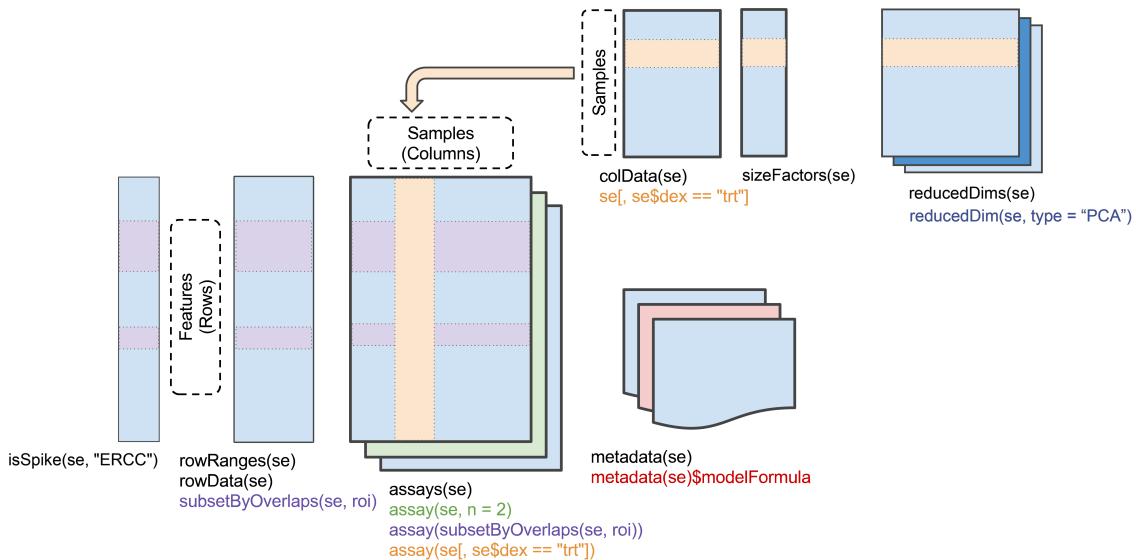


Figure 8.3: Schematic view of the SingleCellExperiment class.

```

library(fletcher2017data)

data(fletcher)
fletcher
#> class: SingleCellExperiment
#> dim: 28284 849
#> metadata(0):
#> assays(1): counts
#> rownames(28284): Xkr4 LOC102640625 ... Ggcx.1 eGFP
#> rowData names(0):
#> colnames(849): OEP01_N706_S501 OEP01_N701_S501 ... OEL23_N704_S503
#> OEL23_N703_S502
#> colData names(19): Experiment Batch ... CreER ERCC_reads
#> reducedDimNames(0):
#> spikeNames(0):

```

Throughout the workshop, we use the class `SingleCellExperiment` to keep track of the counts and their associated metadata within a single object.

The cell-level metadata contain quality control measures, sequencing batch ID, and cluster and lineage labels from the original publication (Fletcher et al. 2017). Cells with a cluster label of -2 were not assigned to any cluster in the original publication.

```

colData(fletcher)
#> DataFrame with 849 rows and 19 columns
#>
#>           Experiment    Batch publishedClusters     NREADS
#>           <factor> <factor>      <numeric> <numeric>
#> OEP01_N706_S501 K5ERRY_UI_96HPT    Y01          1  3313260
#> OEP01_N701_S501 K5ERRY_UI_96HPT    Y01          1  2902430
#> OEP01_N707_S507 K5ERRY_UI_96HPT    Y01          1  2307940
#> OEP01_N705_S501 K5ERRY_UI_96HPT    Y01          1  3337400
#> OEP01_N704_S507 K5ERRY_UI_96HPT    Y01         -2  117892
#> ...
#> ...

```

```

#> OEL23_N704_S510 K5ERP63CKO_UI_14DPT P14 -2 2407440
#> OEL23_N705_S502 K5ERP63CKO_UI_14DPT P14 -2 2308940
#> OEL23_N706_S502 K5ERP63CKO_UI_14DPT P14 12 2215640
#> OEL23_N704_S503 K5ERP63CKO_UI_14DPT P14 12 2673790
#> OEL23_N703_S502 K5ERP63CKO_UI_14DPT P14 7 2450320
#> NALIGNED RALIGN TOTAL_DUP PRIMER
#> <numeric> <numeric> <numeric> <numeric>
#> OEP01_N706_S501 3167600 95.6035 47.9943 0.0154566
#> OEP01_N701_S501 2757790 95.0167 45.015 0.0182066
#> OEP01_N707_S507 2178350 94.3852 43.7832 0.0219196
#> OEP01_N705_S501 3183720 95.3952 43.2688 0.0183041
#> OEP01_N704_S507 98628 83.6596 18.0576 0.0623744
#> ...
#> OEL23_N704_S510 2305060 95.7472 47.1489 0.0159111
#> OEL23_N705_S502 2203300 95.4244 62.5638 0.0195812
#> OEL23_N706_S502 2108490 95.1637 50.6643 0.0182207
#> OEL23_N704_S503 2568300 96.0546 60.5481 0.0155611
#> OEL23_N703_S502 2363500 96.4567 48.4164 0.0122704
#> PCT_RIBOSOMAL_BASES PCT_CODING_BASES PCT_UTR_BASES
#> <numeric> <numeric> <numeric>
#> OEP01_N706_S501 2e-06 0.20013 0.230654
#> OEP01_N701_S501 0 0.182461 0.20181
#> OEP01_N707_S507 0 0.152627 0.207897
#> OEP01_N705_S501 2e-06 0.169514 0.207342
#> OEP01_N704_S507 1.4e-05 0.110724 0.199174
#> ...
#> OEL23_N704_S510 0 0.287346 0.314104
#> OEL23_N705_S502 0 0.337264 0.297077
#> OEL23_N706_S502 7e-06 0.244333 0.262663
#> OEL23_N704_S503 0 0.343203 0.338217
#> OEL23_N703_S502 8e-06 0.259367 0.238239
#> PCT_INTRONIC_BASES PCT_INTERGENIC_BASES PCT_MRNA_BASES
#> <numeric> <numeric> <numeric>
#> OEP01_N706_S501 0.404205 0.165009 0.430784
#> OEP01_N701_S501 0.465702 0.150027 0.384271
#> OEP01_N707_S507 0.511416 0.12806 0.360524
#> OEP01_N705_S501 0.457556 0.165586 0.376856
#> OEP01_N704_S507 0.489514 0.200573 0.309898
#> ...
#> OEL23_N704_S510 0.250658 0.147892 0.60145
#> OEL23_N705_S502 0.230214 0.135445 0.634341
#> OEL23_N706_S502 0.355899 0.137097 0.506997
#> OEL23_N704_S503 0.174696 0.143885 0.68142
#> OEL23_N703_S502 0.376091 0.126294 0.497606
#> MEDIAN_CV_COVERAGE MEDIAN_5PRIME_BIAS MEDIAN_3PRIME_BIAS
#> <numeric> <numeric> <numeric>
#> OEP01_N706_S501 0.843857 0.061028 0.521079
#> OEP01_N701_S501 0.91437 0.03335 0.373993
#> OEP01_N707_S507 0.955405 0.014606 0.49123
#> OEP01_N705_S501 0.81663 0.101798 0.525238
#> OEP01_N704_S507 1.19978 0 0.706512
#> ...
#> OEL23_N704_S510 0.698455 0.198224 0.419745

```

```
#> OEL23_N705_S502      0.830816    0.105091    0.398755
#> OEL23_N706_S502      0.805627    0.103363    0.431862
#> OEL23_N704_S503      0.745201    0.118615    0.38422
#> OEL23_N703_S502      0.711685    0.196725    0.377926
#>                         CreER   ERCC_reads
#>                         <numeric> <numeric>
#> OEP01_N706_S501        1       10516
#> OEP01_N701_S501      3022      9331
#> OEP01_N707_S507      2329      7386
#> OEP01_N705_S501        717      6387
#> OEP01_N704_S507        60       992
#> ...
#> ...
#> OEL23_N704_S510      659       0
#> OEL23_N705_S502      1552       0
#> OEL23_N706_S502        0       0
#> OEL23_N704_S503        0       0
#> OEL23_N703_S502      2222       0
```

8.4 Pre-processing

Using the Bioconductor package `scone`, we remove low-quality cells according to the quality control filter implemented in the function `metric_sample_filter` and based on the following criteria (Figure 8.4): (1) Filter out samples with low total number of reads or low alignment percentage and (2) filter out samples with a low detection rate for housekeeping genes. See the `scone` vignette for details on the filtering procedure.

8.4.1 Sample filtering

```
# QC-metric-based sample-filtering
data("housekeeping")
hk = rownames(fletcher)[toupper(rownames(fletcher)) %in% housekeeping$V1]

mfilt <- metric_sample_filter(counts(fletcher),
                               nreads = colData(fletcher)$NREADS,
                               ralign = colData(fletcher)$RALIGN,
                               pos_controls = rownames(fletcher) %in% hk,
                               zcut = 3, mixture = FALSE,
                               plot = TRUE)

# Simplify to a single logical
mfilt <- !apply(simplify2array(mfilt[!is.na(mfilt)]), 1, any)
filtered <- fletcher[, mfilt]
dim(filtered)
#> [1] 28284    747
```

After sample filtering, we are left with 747 good quality cells.

Finally, for computational efficiency, we retain only the 1,000 most variable genes. This seems to be a reasonable choice for the illustrative purpose of this workflow, as we are able to recover the biological signal found in the published analysis ((Fletcher et al. 2017)). In general, however, we recommend care in selecting a gene filtering scheme, as an appropriate choice is dataset-dependent.

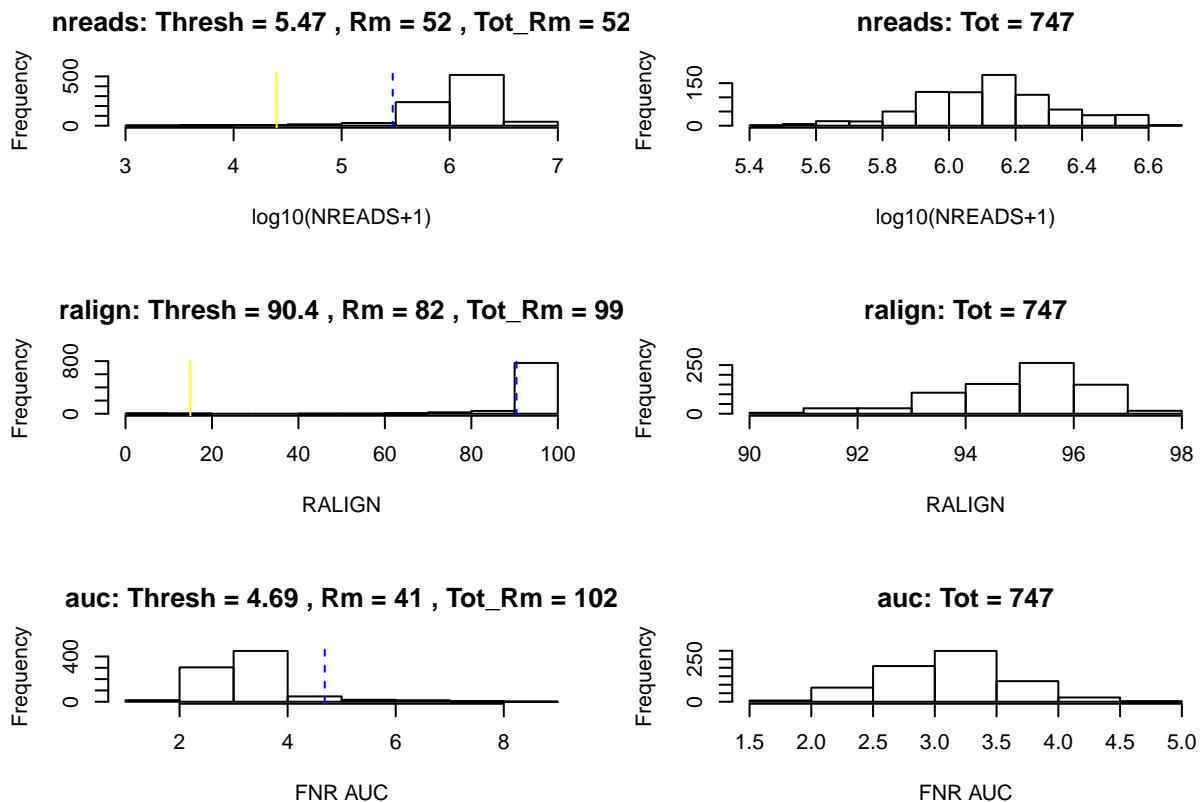


Figure 8.4: SCONE: Filtering of low-quality cells.

We can use functions from the `clusterExperiment` package to compute a filter statistics based on the variance (`makeFilterStats`) and to apply the filter to the data (`filterData`).

```
filtered <- makeFilterStats(filtered, filterStats="var", transFun = log1p)
filtered <- filterData(filtered, percentile=1000, filterStats="var")
filtered
#> class: SingleCellExperiment
#> dim: 1000 747
#> metadata(0):
#> assays(1): counts
#> rownames(1000): Cbr2 Cyp2f2 ... Rnf13 Atp7b
#> rowData names(1): var
#> colnames(747): OEP01_N706_S501 OEP01_N701_S501 ... OEL23_N704_S503
#>     OEL23_N703_S502
#> colData names(19): Experiment Batch ... CreER ERCC_reads
#> reducedDimNames(0):
#> spikeNames(0):
```

In the original work (Fletcher et al. 2017), cells were clustered into 14 different clusters, with 151 cells not assigned to any cluster (i.e., cluster label of -2).

```
publishedClusters <- colData(filtered)[, "publishedClusters"]
col_clus <- c("transparent", "#1B9E77", "antiquewhite2", "cyan", "#E7298A",
            "#A6CEE3", "#666666", "#E6AB02", "#FFED6F", "darkorchid2",
            "#B3DE69", "#FF7F00", "#A6761D", "#1F78B4")
names(col_clus) <- sort(unique(publishedClusters))
```

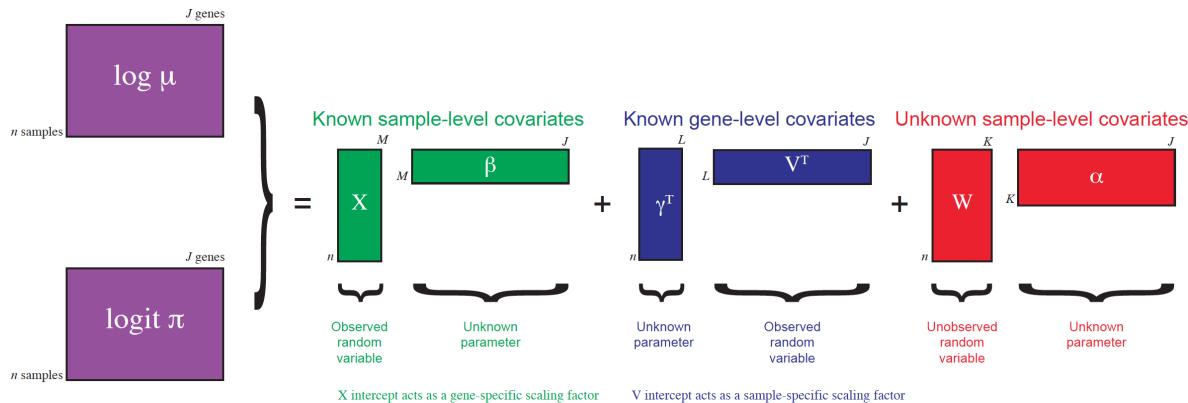


Figure 8.5: ZINB-WaVE: Schematic view of the ZINB-WaVE model. This figure was reproduced with kind permission from Risso et al. (2017).

```
table(publishedClusters)
#> publishedClusters
#> -2 1 2 3 4 5 7 8 9 10 11 12 14 15
#> 151 90 25 54 35 93 58 27 74 26 21 35 26 32
```

8.5 Normalization and dimensionality reduction: ZINB-WaVE

In scRNA-seq analysis, dimensionality reduction is often used as a preliminary step prior to downstream analyses, such as clustering, cell lineage and pseudotime ordering, and the identification of DE genes. This allows the data to become more tractable, both from a statistical (cf. curse of dimensionality) and computational point of view. Additionally, technical noise can be reduced while preserving the often intrinsically low-dimensional signal of interest (Dijk et al. 2017; Pierson and Yau 2015; Risso et al. 2018b).

Here, we perform dimensionality reduction using the zero-inflated negative binomial-based wanted variation extraction (ZINB-WaVE) method implemented in the Bioconductor R package `zinbwave`. The method fits a ZINB model that accounts for zero inflation (dropouts), over-dispersion, and the count nature of the data. The model can include a cell-level intercept, which serves as a global-scaling normalization factor. The user can also specify both gene-level and cell-level covariates. The inclusion of observed and unobserved cell-level covariates enables normalization for complex, non-linear effects (often referred to as batch effects), while gene-level covariates may be used to adjust for sequence composition effects (e.g., gene length and GC-content effects). A schematic view of the ZINB-WaVE model is provided in Figure 8.5. For greater detail about the ZINB-WaVE model and estimation procedure, please refer to the original manuscript (Risso et al. 2018b).

As with most dimensionality reduction methods, the user needs to specify the number of dimensions for the new low-dimensional space. Here, we use $K = 50$ dimensions and adjust for batch effects via the matrix X .

```
clustered <- zinbwave(filtered, K = 50, X = "~ Batch", residuals = TRUE, normalizedValues = TRUE))
```

Note that the `fletcher2017data` package includes the object `clustered` that already contains the ZINB-WaVE factors. We can load such objects to avoid waiting for the computations.

```
data(clustered)
```

8.5.1 Normalization

The function `zinbwave` returns a `SingleCellExperiment` object that includes normalized expression measures, defined as deviance residuals from the fit of the ZINB-WaVE model with user-specified gene- and cell-level covariates. Such residuals can be used for visualization purposes (e.g., in heatmaps, boxplots). Note that, in this case, the low-dimensional matrix W is not included in the computation of residuals to avoid the removal of the biological signal of interest.

```
assayNames(clustered)
#> [1] "normalizedValues" "residuals"           "counts"
norm <- assay(clustered, "normalizedValues")
norm[1:3,1:3]
#>      OEP01_N706_S501 OEP01_N701_S501 OEP01_N707_S507
#> Cbr2        4.531898     4.369185    -4.142982
#> Cyp2f2      4.359680     4.324476     4.124527
#> Gstm1       4.724216     4.621898     4.403587
```

8.5.2 Dimensionality reduction

The `zinbwave` function's main use is to perform dimensionality reduction. The resulting low-dimensional matrix W is stored in the `reducedDim` slot named `zinbwave`.

```
reducedDimNames(clustered)
#> [1] "zinbwave"
W <- reducedDim(clustered, "zinbwave")
dim(W)
#> [1] 747 50
W[1:3, 1:3]
#>          W1         W2         W3
#> OEP01_N706_S501 0.5494761 1.1745361 -0.93175747
#> OEP01_N701_S501 0.4116797 0.3015379 -0.46922527
#> OEP01_N707_S507 0.7394759 0.3365600 -0.07959226
```

The low-rank matrix W can be visualized in two dimensions by performing multi-dimensional scaling (MDS) using the Euclidian distance. To verify that W indeed captures the biological signal of interest, we display the MDS results in a scatterplot with colors corresponding to the original published clusters (Figure 8.6).

```
W <- reducedDim(clustered)
d <- dist(W)
fit <- cmdscale(d, eig = TRUE, k = 2)
plot(fit$points, col = col_clus[as.character(publishedClusters)], main = "",
     pch = 20, xlab = "Component 1", ylab = "Component 2")
legend(x = "topleft", legend = unique(names(col_clus)), cex = .5, fill = unique(col_clus), title = "Sample")
```

8.6 Cell clustering: RSEC

The next step is to cluster the cells according to the low-dimensional matrix W computed in the previous step. We use the resampling-based sequential ensemble clustering (RSEC) framework implemented in the `RSEC` function from the Bioconductor R package `clusterExperiment`. Specifically, given a set of user-supplied base clustering algorithms and associated tuning parameters (e.g., k -means, with a range of values for k), RSEC generates a collection of candidate clusterings, with the option of resampling cells and using a sequential tight clustering procedure as in (Tseng and Wong 2005). A consensus clustering is obtained based on the levels of co-clustering of samples across the candidate clusterings. The consensus clustering is further condensed by

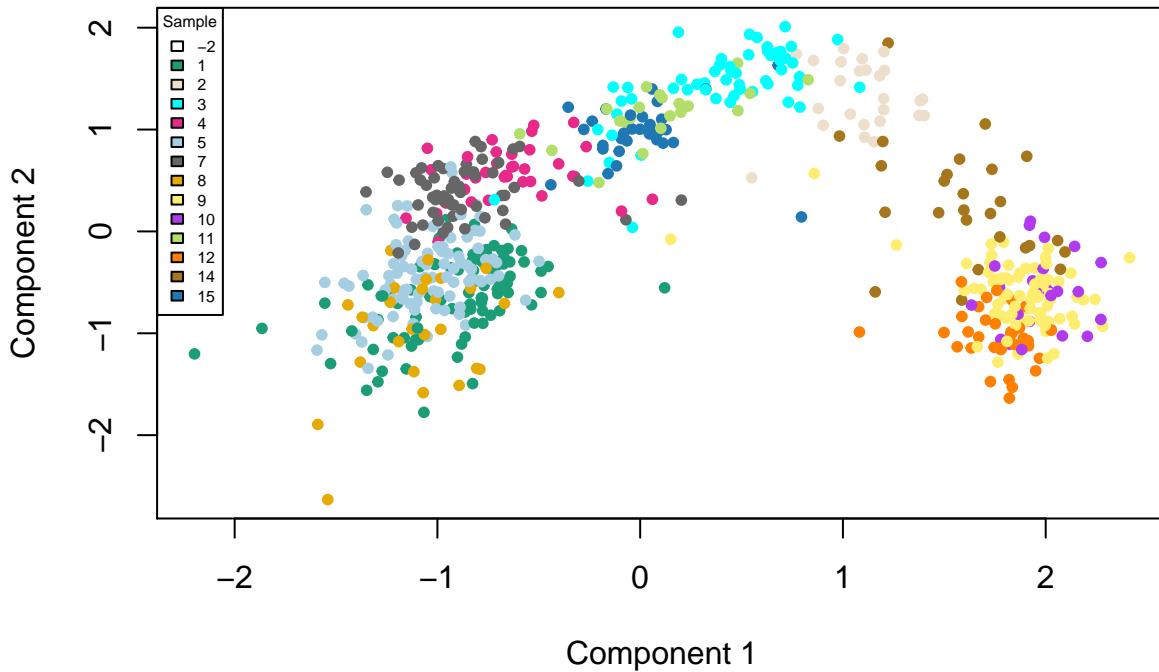


Figure 8.6: ZINB-WaVE: MDS of the low-dimensional matrix W , where each point represents a cell and cells are color-coded by original published clustering.

merging similar clusters, which is done by creating a hierarchy of clusters, working up the tree, and testing for differential expression between sister nodes, with nodes of insufficient DE collapsed. As in supervised learning, resampling greatly improves the stability of clusters and considering an ensemble of methods and tuning parameters allows us to capitalize on the different strengths of the base algorithms and avoid the subjective selection of tuning parameters.

```
clustered <- RSEC(clustered, k0s = 4:15, alphas = c(0.1),
                     betas = 0.8, reduceMethod="zinbwave",
                     clusterFunction = "hierarchical01", minSizes=1,
                     ncores = NCORES, isCount=FALSE,
                     dendroReduce="zinbwave",
                     subsampleArgs = list(resamp.num=100,
                                          clusterFunction="kmeans",
                                          clusterArgs=list(nstart=10)),
                     verbose=TRUE,
                     consensusProportion = 0.7,
                     mergeMethod = "none", random.seed=424242,
                     consensusMinSize = 10)
```

Again, the previously loaded `clustered` object already contains the results of the `RSEC` run above, so we do not evaluate the above chunk here.

```
clustered
#> class: ClusterExperiment
#> dim: 1000 747
#> reducedDimNames: zinbwave
#> filterStats: var
#> -----
#> Primary cluster type: makeConsensus
#> Primary cluster label: makeConsensus
```

```
#> Table of clusters (of primary clustering):
#> -1 c1 c2 c3 c4 c5 c6 c7
#> 184 145 119 105 100 48 33 13
#> Total number of clusterings: 13
#> Dendrogram run on 'makeConsensus' (cluster index: 1)
#> -----
#> Workflow progress:
#> clusterMany run? Yes
#> makeConsensus run? Yes
#> makeDendrogram run? Yes
#> mergeClusters run? No
```

Note that the results of the `RSEC` function is an object of the `ClusterExperiment` class, which extends the `SingleCellExperiment` class, by adding additional information on the clustering results.

```
is(clustered, "SingleCellExperiment")
#> [1] TRUE
slotNames(clustered)
#> [1] "transformation"           "clusterMatrix"
#> [3] "primaryIndex"             "clusterInfo"
#> [5] "clusterTypes"             "dendro_samples"
#> [7] "dendro_clusters"          "dendro_index"
#> [9] "dendro_outbranch"         "coClustering"
#> [11] "clusterLegend"           "orderSamples"
#> [13] "merge_index"              "merge_dendrocluster_index"
#> [15] "merge_method"             "merge_demethod"
#> [17] "merge_cutoff"             "merge_nodeProp"
#> [19] "merge_nodeMerge"          "int_elementMetadata"
#> [21] "int_colData"              "int_metadata"
#> [23] "reducedDims"              "rowRanges"
#> [25] "colData"                  "assays"
#> [27] "NAMES"                    "elementMetadata"
#> [29] "metadata"
```

The resulting candidate clusterings can be visualized using the `plotClusters` function (Figure 8.7), where columns correspond to cells and rows to different clusterings. Each sample is color-coded based on its clustering for that row, where the colors have been chosen to try to match up clusters that show large overlap across rows. The first row correspond to a consensus clustering across all candidate clusterings.

```
plotClusters(clustered)
```

The `plotCoClustering` function produces a heatmap of the co-clustering matrix, which records, for each pair of cells, the proportion of times they were clustered together across the candidate clusters (Figure 8.8).

```
plotCoClustering(clustered)
```

The distribution of cells across the consensus clusters can be visualized in Figure 8.9 and is as follows:

```
table(primaryClusterNamed(clustered))
#>
#> -1 c1 c2 c3 c4 c5 c6 c7
#> 184 145 119 105 100 48 33 13
```

```
plotBarplot(clustered, legend = FALSE)
```

The distribution of cells in our clustering overall agrees with that in the original published clustering (Figure 8.10), the main difference being that several of the published clusters were merged here into single clusters.

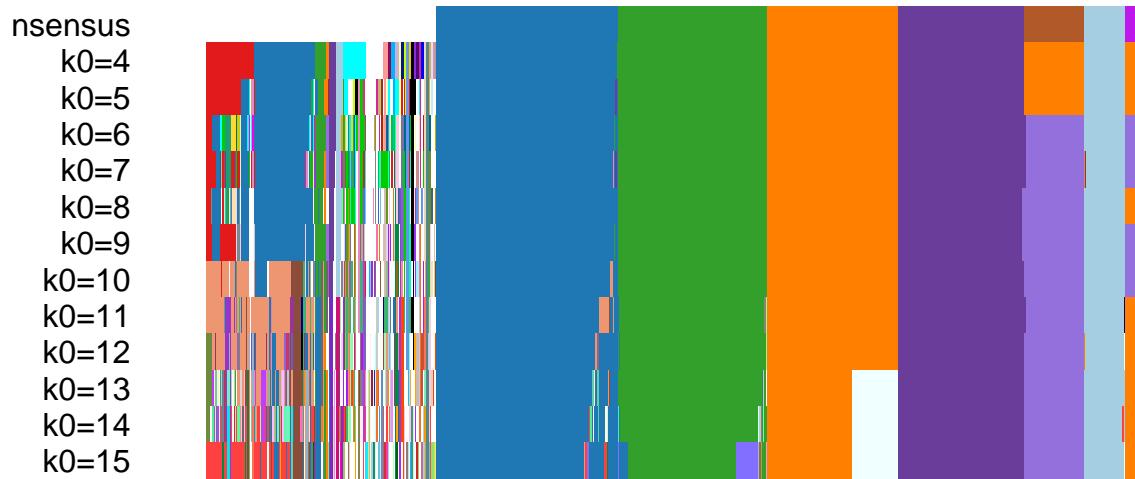


Figure 8.7: RSEC: Candidate clusterings found using the function RSEC from the clusterExperiment package.

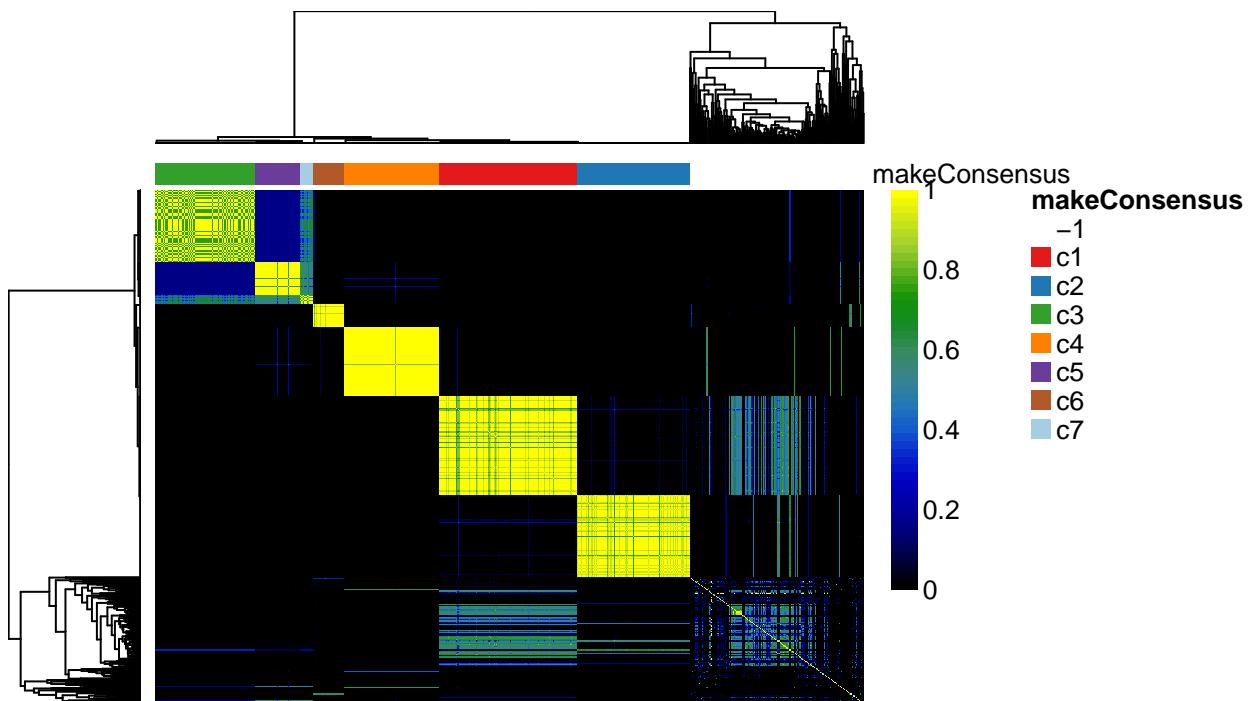


Figure 8.8: RSEC: Heatmap of co-clustering matrix.

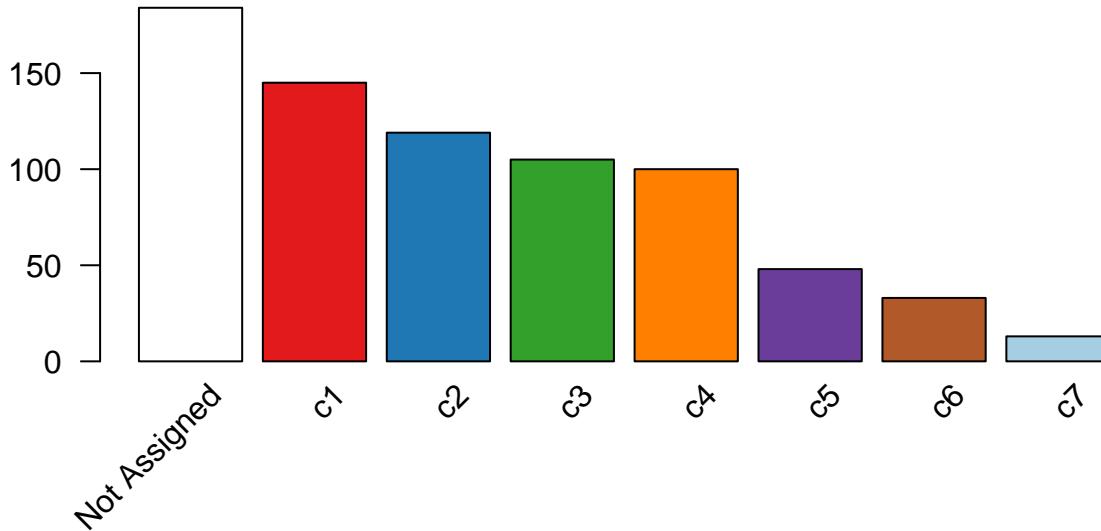


Figure 8.9: RSEC: Barplot of number of cells per cluster for our workflow’s RSEC clustering.

This discrepancy is likely caused by the fact that we started with the top 1,000 genes, which might not be enough to discriminate between closely related clusters.

```
clustered <- addClusterings(clustered, colData(clustered)$publishedClusters,
                             clusterLabel = "publishedClusters")

## change default color to match with Figure 7
clusterLegend(clustered)$publishedClusters[, "color"] <-
  col_clus[clusterLegend(clustered)$publishedClusters[, "name"]]

plotBarplot(clustered, whichClusters=c("makeConsensus", "publishedClusters"),
            xlab = "", legend = FALSE, missingColor="white")

plotClustersTable(clustered, whichClusters=c("makeConsensus", "publishedClusters"))
```

Figure 8.12 displays a heatmap of the normalized expression measures for the 1,000 most variable genes, where cells are clustered according to the RSEC consensus.

```
# Set colors for additional sample data
experimentColors <- bigPalette[1:nlevels(colData(clustered)$Experiment)]
batchColors <- bigPalette[1:nlevels(colData(clustered)$Batch)]
metaColors <- list("Experiment" = experimentColors,
                   "Batch" = batchColors)

plotHeatmap(clustered,
            whichClusters = c("makeConsensus", "publishedClusters"), clusterFeaturesData = "all",
            clusterSamplesData = "dendrogramValue", breaks = 0.99,
            colData = c("Batch", "Experiment"),
            clusterLegend = metaColors, annLegend = FALSE, main = "")
```

Finally, we can visualize the cells in a two-dimensional space using the MDS of the low-dimensional matrix W and coloring the cells according to their newly-found RSEC clusters (Figure 8.13); this is analogous to Figure 8.6 for the original published clusters.

```
plotReducedDims(clustered, whichCluster="primary", reducedDim="zinbwave", pch=20,
                xlab = "Component1", ylab = "Component2", legendTitle="Sample", main="",
```

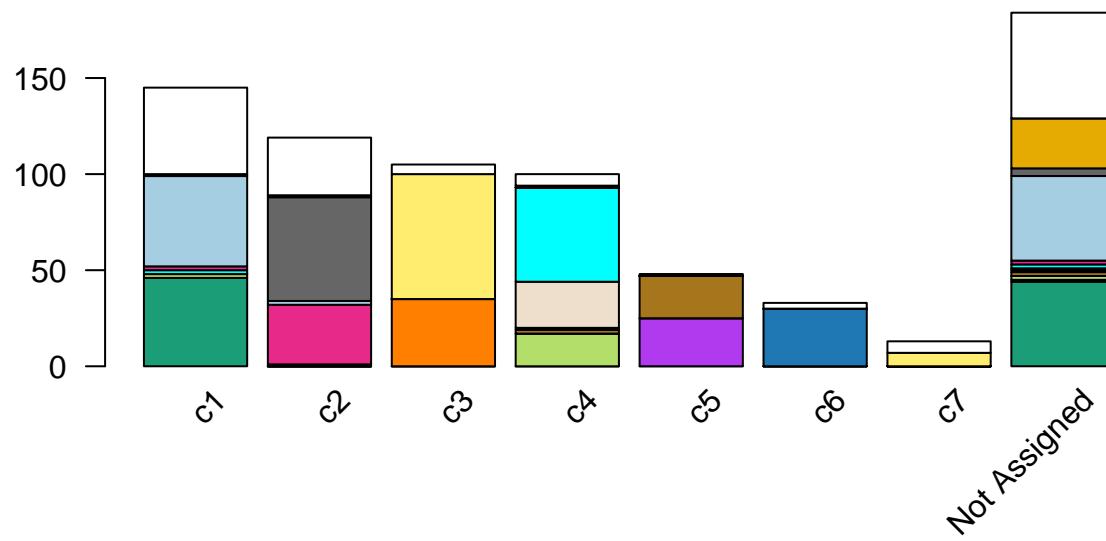


Figure 8.10: RSEC: Barplot of number of cells per cluster, for our workflow’s RSEC clustering, color-coded by original published clustering.

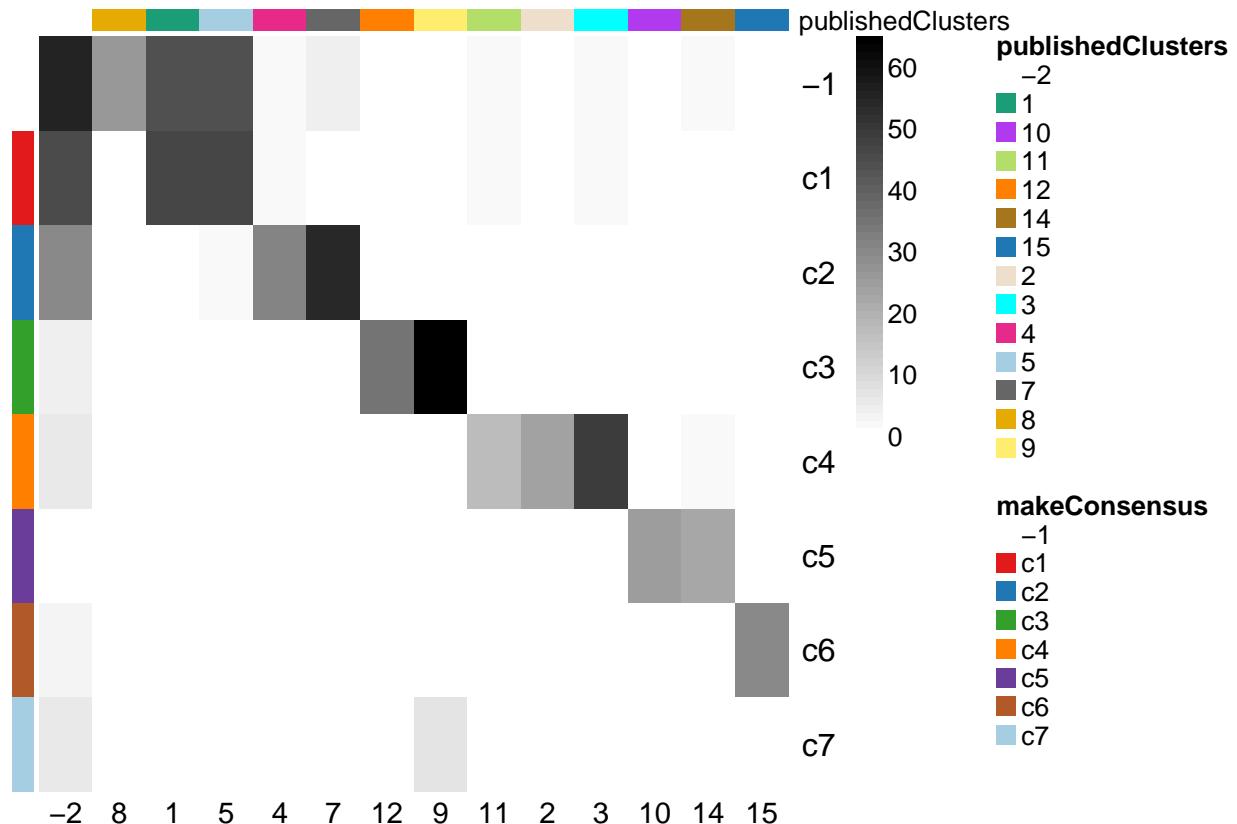


Figure 8.11: RSEC: Confusion matrix of number of cells per cluster, for our workflow’s RSEC clustering and the original published clustering.

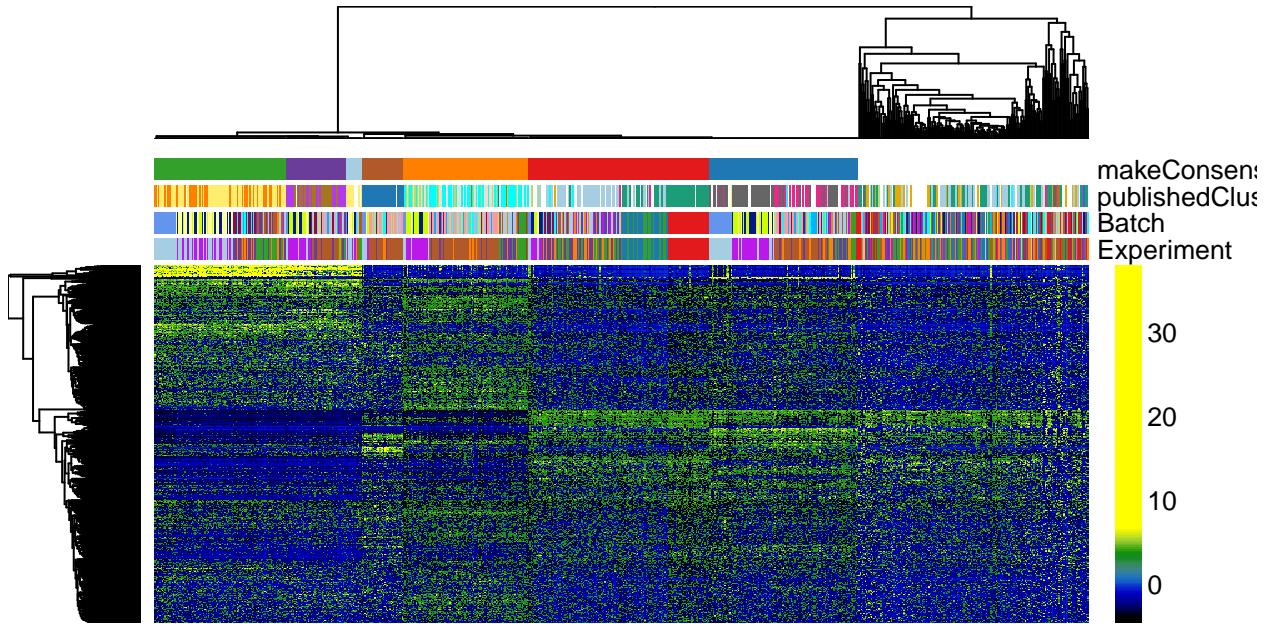


Figure 8.12: RSEC: Heatmap of the normalized expression measures for the 1,000 most variable genes, where rows correspond to genes and columns to cells ordered by RSEC clusters.

```
plotUnassigned=FALSE
)
```

8.7 Cell lineage and pseudotime inference: Slingshot

We now demonstrate how to use the Bioconductor package `slingshot` to infer branching cell lineages and order cells by developmental progression along each lineage. The method, proposed in (Street et al. 2018), comprises two main steps: (1) The inference of the global lineage structure (i.e., the number of lineages and where they branch) using a minimum spanning tree (MST) on the clusters identified above by RSEC and (2) the inference of cell pseudotime variables along each lineage using a novel method of simultaneous principal curves. The approach in (1) allows the identification of any number of novel lineages, while also accommodating the use of domain-specific knowledge to supervise parts of the tree (e.g., known terminal states); the approach in (2) yields robust pseudotimes for smooth, branching lineages.

This analysis is performed out by the `slingshot` function and the results are stored in a `SlingshotDataSet` object. The minimal input to this function is a low-dimensional representation of the cells and a set of cluster labels; these can be separate objects (ie. a matrix and a vector) or, as below, components of a `SingleCellExperiment` object. When a `SingleCellExperiment` object is provided as input, the output will be an updated object containing a `SlingshotDataSet` as an element of the `int_metadata` list, which can be accessed through the `SlingshotDataSet` function. For more low-level control of the lineage inference procedure, the two steps may be run separately via the functions `getLineages` and `getCurves`.

From the original published work, we know that the starting cluster should correspond to HBCs and the end clusters to MV, mOSN, and mSUS cells. Additionally, we know that GBCs should be at a junction before the differentiation between MV and mOSN cells (Figure 8.2). The correspondence between the clusters we found here and the original clusters is as follows.

```
table(data.frame(original = publishedClusters, ours = primaryClusterNamed(clustered)))
#>      ours
```

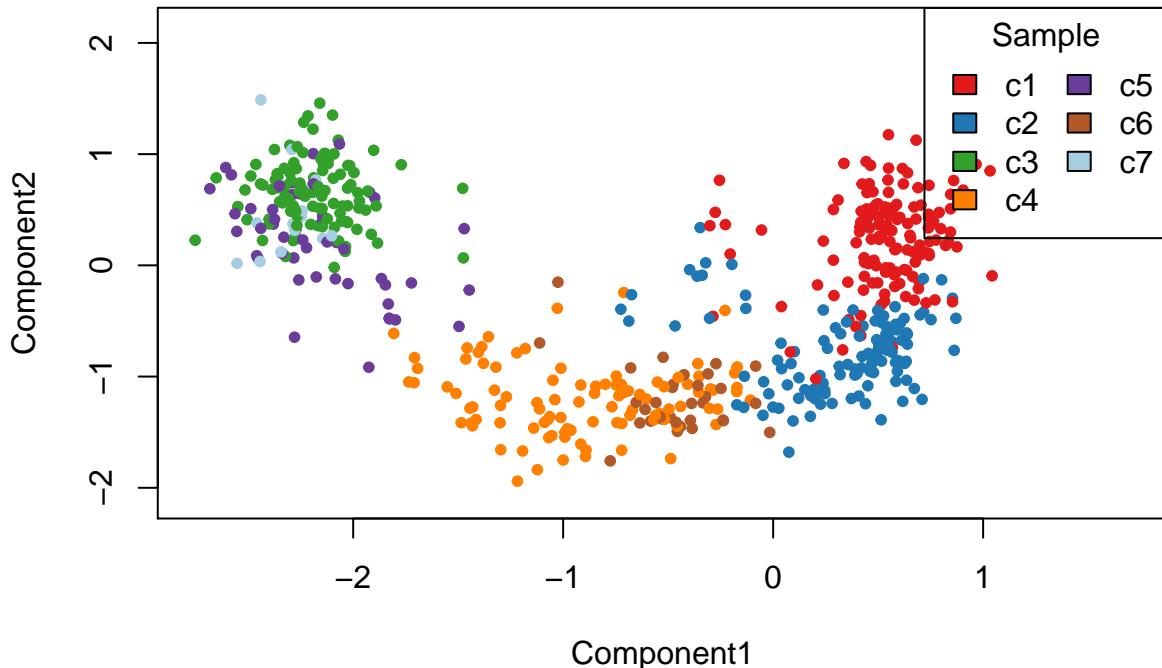


Figure 8.13: RSEC: MDS of the low-dimensional matrix W , where each point represents a cell and cells are color-coded by RSEC clustering.

```
#> original -1 c1 c2 c3 c4 c5 c6 c7
#>      -2 55 45 30  5  6  1  3  6
#>      1 44 46  0  0  0  0  0  0
#>      2  1  0  0  0 24  0  0  0
#>      3  2  2  1  0 49  0  0  0
#>      4  2  2 31  0  0  0  0  0
#>      5 44 47  2  0  0  0  0  0
#>      7  4  0 54  0  0  0  0  0
#>      8 26  1  0  0  0  0  0  0
#>      9  0  0  1 65  1  0  0  7
#>     10  1  0  0  0  0 25  0  0
#>     11  2  2  0  0 17  0  0  0
#>     12  0  0  0 35  0  0  0  0
#>     14  2  0  0  0 22  0  0  0
#>     15  1  0  0  0  1  0 30  0
```

Cluster name	Description	Color	Correspondence
c1	HBC	red	original 1, 5
c2	mSUS	blue	original 4, 7
c3	mOSN	green	original 9, 12
c4	GBC	orange	original 2, 3, 11
c5	Immature Neuron	purple	original 10, 14
c6	MV	brown	original 15
c7	mOSN	light blue	original 9

To infer lineages and pseudotimes, we will apply Slingshot to the 4-dimensional MDS of the low-dimensional matrix W . We found that the Slingshot results were robust to the number of dimensions k for the MDS (we tried k from 2 to 5). Here, we use a semi-supervised version of Slingshot, where we only provide the identity of the start cluster but not of the end clusters.

```
pseudoCe <- clustered[, !primaryClusterNamed(clustered) %in% c("-1")]
X <- reducedDim(pseudoCe, type = "zinbwave")
mds <- cmdscale(dist(X), eig = TRUE, k = 4)
lineages <- slingshot(mds$points, clusterLabels = primaryClusterNamed(pseudoCe), start.clus = "c1")
```

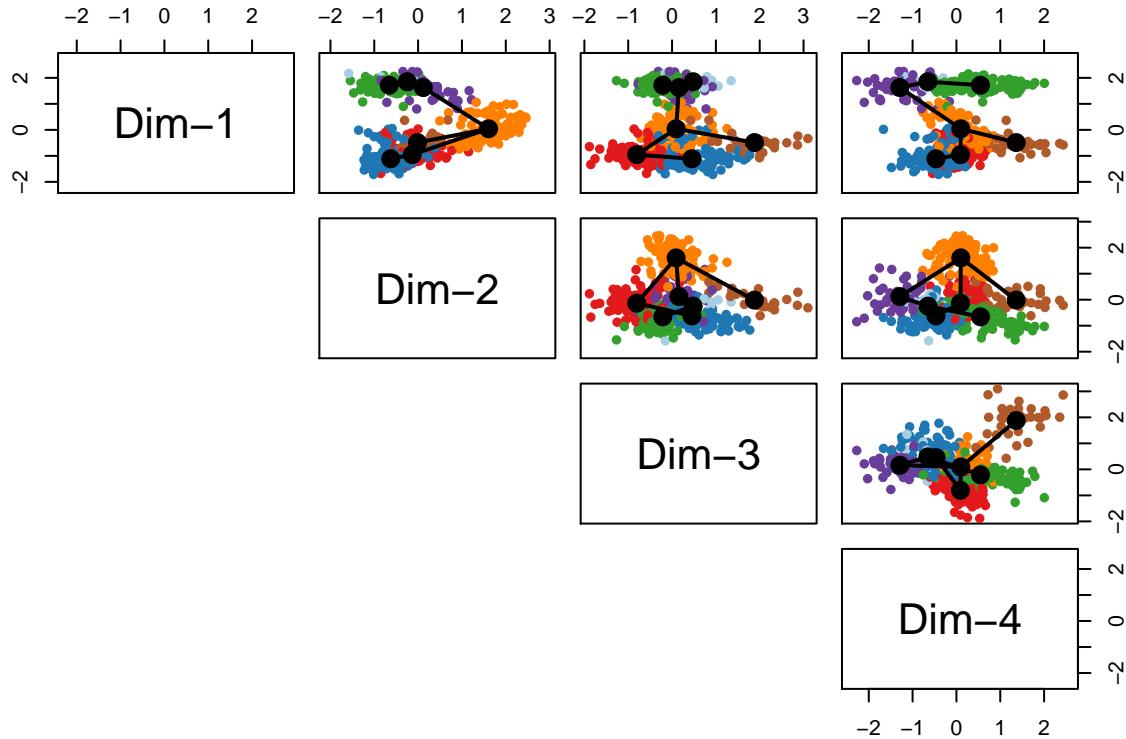


Figure 8.14: Slingshot: Cells color-coded by cluster in a 4-dimensional MDS space, with connecting lines between cluster centers representing the inferred global lineage structure.

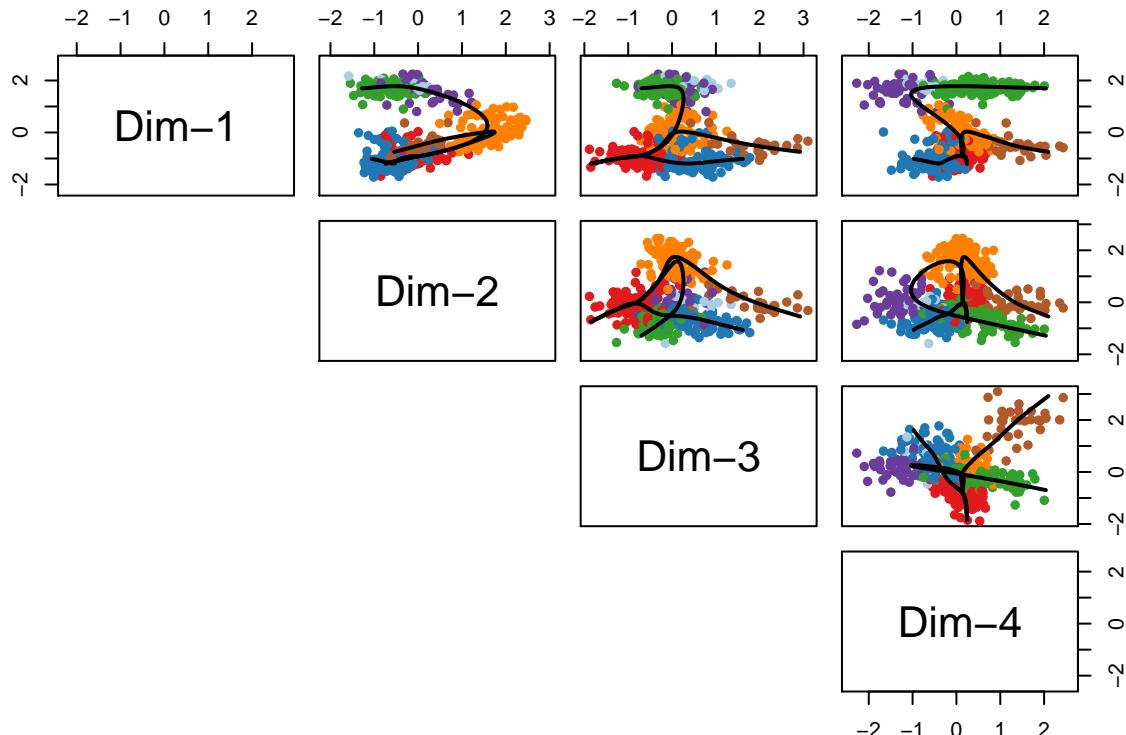


Figure 8.15: Slingshot: Cells color-coded by cluster in a 4-dimensional MDS space, with smooth curves representing each inferred lineage.

```

lineages
#> class: SlingshotDataSet
#>
#>   Samples Dimensions
#>      563          4
#>
#>   lineages: 3
#> Lineage1: c1  c4  c5  c7  c3
#> Lineage2: c1  c4  c6
#> Lineage3: c1  c2
#>
#>   curves: 3
#> Curve1: Length: 9.5238  Samples: 361.21
#> Curve2: Length: 7.8221  Samples: 234.14
#> Curve3: Length: 4.2828  Samples: 254.85

```

As an alternative, we could have incorporated the MDS results into the `clustered` object and applied `slingshot` directly to it. Here, we need to specify that we want to use the MDS results, because `slingshot` would otherwise use the first element of the `reducedDims` list (in this case, the 10-dimensional W matrix from `zinbwave`).

```

reducedDim(pseudoCe, "MDS") <- mds$points
pseudoCe <- slingshot(pseudoCe, reducedDim = "MDS", start.clus = "c1")
#> Using full covariance matrix
pseudoCe
#> class: ClusterExperiment
#> dim: 1000 563
#> reducedDimNames: zinbwave MDS
#> filterStats: var
#> -----
#> Primary cluster type: makeConsensus
#> Primary cluster label: makeConsensus
#> Table of clusters (of primary clustering):
#>   c1  c2  c3  c4  c5  c6  c7
#> 145 119 105 100  48  33  13
#> Total number of clusterings: 14
#> No dendrogram present
#> -----
#> Workflow progress:
#> clusterMany run? Yes
#> makeConsensus run? Yes
#> makeDendrogram run? No
#> mergeClusters run? No
colData(pseudoCe)
#> DataFrame with 563 rows and 23 columns
#>           Experiment     Batch publishedClusters    NREADS
#>           <factor> <factor>        <numeric> <numeric>
#> OEP01_N706_S501 K5ERRY_UI_96HPT Y01            1 3313260
#> OEP01_N701_S501 K5ERRY_UI_96HPT Y01            1 2902430
#> OEP01_N707_S507 K5ERRY_UI_96HPT Y01            1 2307940
#> OEP01_N705_S501 K5ERRY_UI_96HPT Y01            1 3337400
#> OEP01_N702_S508 K5ERRY_UI_96HPT Y01           -2 525096
#> ...
#> OEL23_N704_S510 K5ERP63CKO_UI_14DPT    P14           -2 2407440

```

```

#> OEL23_N705_S502 K5ERP63CKO_UI_14DPT P14 -2 2308940
#> OEL23_N706_S502 K5ERP63CKO_UI_14DPT P14 12 2215640
#> OEL23_N704_S503 K5ERP63CKO_UI_14DPT P14 12 2673790
#> OEL23_N703_S502 K5ERP63CKO_UI_14DPT P14 7 2450320
#> NALIGNED RALIGN TOTAL_DUP PRIMER
#> <numeric> <numeric> <numeric> <numeric>
#> OEP01_N706_S501 3167600 95.6035 47.9943 0.0154566
#> OEP01_N701_S501 27577790 95.0167 45.015 0.0182066
#> OEP01_N707_S507 2178350 94.3852 43.7832 0.0219196
#> OEP01_N705_S501 3183720 95.3952 43.2688 0.0183041
#> OEP01_N702_S508 484847 92.3349 18.806 0.0248804
#> ...
#> OEL23_N704_S510 2305060 95.7472 47.1489 0.0159111
#> OEL23_N705_S502 2203300 95.4244 62.5638 0.0195812
#> OEL23_N706_S502 2108490 95.1637 50.6643 0.0182207
#> OEL23_N704_S503 2568300 96.0546 60.5481 0.0155611
#> OEL23_N703_S502 2363500 96.4567 48.4164 0.0122704
#> PCT_RIBOSOMAL_BASES PCT_CODING_BASES PCT_UTR_BASES
#> <numeric> <numeric> <numeric>
#> OEP01_N706_S501 2e-06 0.20013 0.230654
#> OEP01_N701_S501 0 0.182461 0.20181
#> OEP01_N707_S507 0 0.152627 0.207897
#> OEP01_N705_S501 2e-06 0.169514 0.207342
#> OEP01_N702_S508 0 0.130247 0.230848
#> ...
#> OEL23_N704_S510 0 0.287346 0.314104
#> OEL23_N705_S502 0 0.337264 0.297077
#> OEL23_N706_S502 7e-06 0.244333 0.262663
#> OEL23_N704_S503 0 0.343203 0.338217
#> OEL23_N703_S502 8e-06 0.259367 0.238239
#> PCT_INTRONIC_BASES PCT_INTERGENIC_BASES PCT_MRNA_BASES
#> <numeric> <numeric> <numeric>
#> OEP01_N706_S501 0.404205 0.165009 0.430784
#> OEP01_N701_S501 0.465702 0.150027 0.384271
#> OEP01_N707_S507 0.511416 0.12806 0.360524
#> OEP01_N705_S501 0.457556 0.165586 0.376856
#> OEP01_N702_S508 0.477167 0.161738 0.361095
#> ...
#> OEL23_N704_S510 0.250658 0.147892 0.60145
#> OEL23_N705_S502 0.230214 0.135445 0.634341
#> OEL23_N706_S502 0.355899 0.137097 0.506997
#> OEL23_N704_S503 0.174696 0.143885 0.68142
#> OEL23_N703_S502 0.376091 0.126294 0.497606
#> MEDIAN_CV_COVERAGE MEDIAN_5PRIME_BIAS MEDIAN_3PRIME_BIAS
#> <numeric> <numeric> <numeric>
#> OEP01_N706_S501 0.843857 0.061028 0.521079
#> OEP01_N701_S501 0.91437 0.03335 0.373993
#> OEP01_N707_S507 0.955405 0.014606 0.49123
#> OEP01_N705_S501 0.81663 0.101798 0.525238
#> OEP01_N702_S508 1.13937 0 0.671565
#> ...
#> OEL23_N704_S510 0.698455 0.198224 0.419745
#> OEL23_N705_S502 0.830816 0.105091 0.398755

```

```

#> OEL23_N706_S502          0.805627      0.103363      0.431862
#> OEL23_N704_S503          0.745201      0.118615      0.38422
#> OEL23_N703_S502          0.711685      0.196725      0.377926
#>                               CreER ERCC_reads slingClusters slingPseudotime_1
#>                               <numeric> <numeric> <character> <numeric>
#> OEP01_N706_S501           1       10516        c1      NA
#> OEP01_N701_S501           3022      9331        c1  1.17234765019331
#> OEP01_N707_S507           2329      7386        c1  1.05857338852636
#> OEP01_N705_S501           717       6387        c1  1.60463630065258
#> OEP01_N702_S508            6       1218        c1  1.15934111346639
#> ...
#> OEL23_N704_S510           659        0          c2      NA
#> OEL23_N705_S502           1552       0          c2      NA
#> OEL23_N706_S502            0        0          c3  8.14552572709918
#> OEL23_N704_S503            0        0          c3  8.53595122837615
#> OEL23_N703_S502           2222       0          c2      NA
#>                               slingPseudotime_2 slingPseudotime_3
#>                               <numeric> <numeric>
#> OEP01_N706_S501             NA  0.692789182245903
#> OEP01_N701_S501           1.16364108574816  1.14696263207989
#> OEP01_N707_S507           1.0611741023521  1.03755897807688
#> OEP01_N705_S501           1.61033070603652  1.4465916227885
#> OEP01_N702_S508           1.1674954947514  1.4206584204892
#> ...
#> OEL23_N704_S510             NA  2.01842841303396
#> OEL23_N705_S502             NA  3.75230631265741
#> OEL23_N706_S502             NA      NA
#> OEL23_N704_S503             NA      NA
#> OEL23_N703_S502             NA  2.74576551163381

```

The result of `slingshot` applied to a `ClusterExperiment` object is still of class `ClusterExperiment`. Note that we did not specify a set of cluster labels, implying that `slingshot` should use the default `primaryClusterNamed` vector.

In the workflow, we recover a reasonable ordering of the clusters using the largely unsupervised version of `slingshot`. However, in some other cases, we have noticed that we need to give more guidance to the algorithm to find the correct ordering. `getLineages` has the option for the user to provide known end cluster(s), which represents a constraint on the MST requiring those clusters to be leaf nodes. Here is the code to use `slingshot` in a supervised setting, where we know that clusters `c3`, `c6` and `c2` represent terminal cell fates.

```

pseudoCeSup <- slingshot(pseudoCe, reducedDim = "MDS", start.clus = "c1",
                           end.clus = c("c3", "c6", "c2"))

```

8.8 Differential expression analysis along lineages

After assigning the cells to lineages and ordering them within lineages, we are interested in finding genes that have non-constant expression patterns over pseudotime.

More formally, for each lineage, we use the robust local regression method loess to model in a flexible, non-linear manner the relationship between a gene's normalized expression measures and pseudotime. We then can test the null hypothesis of no change over time for each gene using the `gam` package. We implement this approach for the neuronal lineage and display the expression measures of the top 100 genes by p-value in

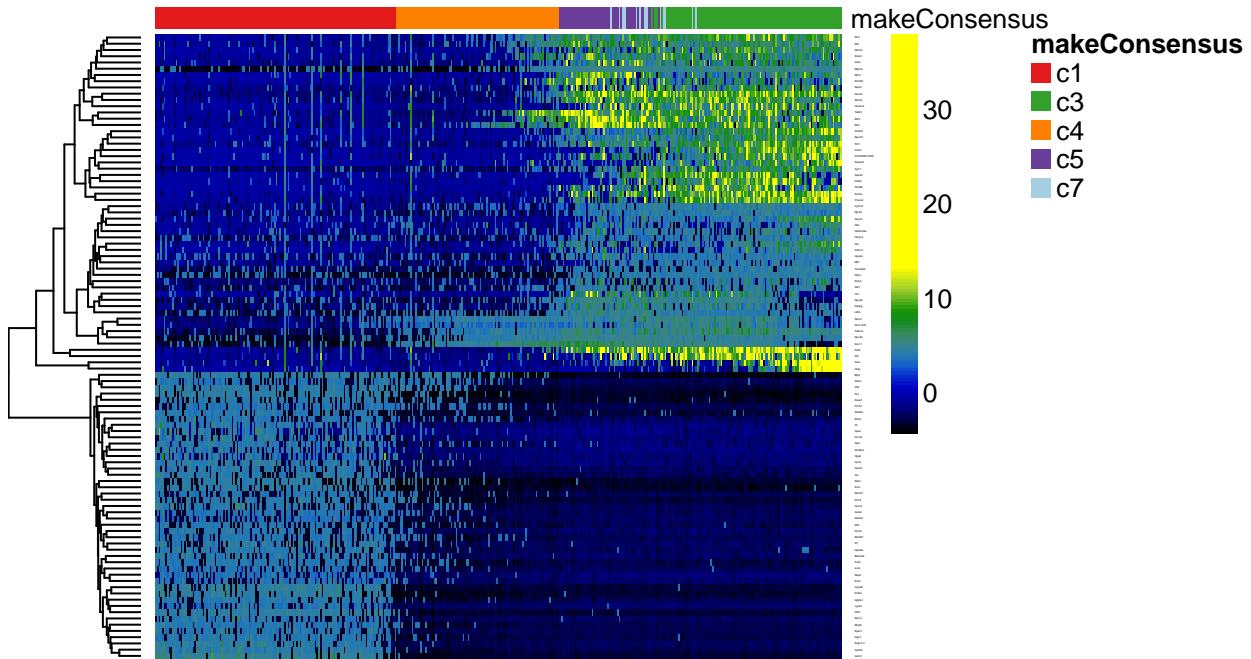


Figure 8.16: DE: Heatmap of the normalized expression measures for the 100 most significantly DE genes for the neuronal lineage, where rows correspond to genes and columns to cells ordered by pseudotime.

the heatmap of Figure 8.16.

```
t <- colData(pseudoCe)$slingPseudotime_1
y <- transformData(pseudoCe)
gam.pval <- apply(y, 1, function(z){
  d <- data.frame(z=z, t=t)
  tmp <- gam(z ~ lo(t), data=d)
  p <- summary(tmp)[4][[1]][1,5]
  p
})

topgenes <- names(sort(gam.pval, decreasing = FALSE))[1:100]

pseudoCe1 <- pseudoCe[, !is.na(t)]
orderSamples(pseudoCe1) <- order(t[!is.na(t)])

plotHeatmap(pseudoCe1[topgenes,], clusterSamplesData = "orderSamplesValue", breaks = .99)
```

Chapter 9

210: Functional enrichment analysis of high-throughput omics data

9.1 Instructor names and contact information

- Ludwig Geistlinger (Ludwig.Geistlinger@sph.cuny.edu)
- Levi Waldron

CUNY School of Public Health 55 W 125th St, New York, NY 10027

9.2 Workshop Description

This workshop gives an in-depth overview of existing methods for enrichment analysis of gene expression data with regard to functional gene sets, pathways, and networks. The workshop will help participants understand the distinctions between assumptions and hypotheses of existing methods as well as the differences in objectives and interpretation of results. It will provide code and hands-on practice of all necessary steps for differential expression analysis, gene set- and network-based enrichment analysis, and identification of enriched genomic regions and regulatory elements, along with visualization and exploration of results.

9.2.1 Pre-requisites

- Basic knowledge of R syntax
- Familiarity with the SummarizedExperiment class
- Familiarity with the GenomicRanges class
- Familiarity with high-throughput gene expression data as obtained with microarrays and RNA-seq
- Familiarity with the concept of differential expression analysis (with e.g. limma, edgeR, DESeq2)

9.2.2 Workshop Participation

Execution of example code and hands-on practice

9.2.3 *R / Bioconductor* packages used

- EnrichmentBrowser
- regioneR
- airway
- ALL
- hgu95av2.db
- BSgenome.Hsapiens.UCSC.hg19.masked

9.2.4 Time outline

Activity	Time
Background	30m
Differential expression analysis	15m
Gene set analysis	30m
Gene network analysis	15m
Genomic region analysis	15m

9.3 Goals and objectives

Theory:

- Gene sets, pathways & regulatory networks
- Resources
- Gene set analysis vs. gene set enrichment analysis
- Underlying null: competitive vs. self-contained
- Generations: ora, fcs & topology-based

Practice:

- Data types: microarray vs. RNA-seq
- Differential expression analysis
- Defining gene sets according to GO and KEGG
- GO/KEGG overrepresentation analysis
- Functional class scoring & permutation testing
- Network-based enrichment analysis
- Genomic region enrichment analysis

9.4 Where does it all come from?

Test whether known biological functions or processes are over-represented (= enriched) in an experimentally-derived gene list, e.g. a list of differentially expressed (DE) genes. See Goeman and Buehlmann, 2007 for a critical review.

Example: Transcriptomic study, in which 12,671 genes have been tested for differential expression between two sample conditions and 529 genes were found DE.

Among the DE genes, 28 are annotated to a specific functional gene set, which contains in total 170 genes. This setup corresponds to a 2x2 contingency table,

```
deTable <-  
  matrix(c(28, 142, 501, 12000),
```

```

nrow = 2,
dimnames = list(c("DE", "Not.DE"),
                 c("In.gene.set", "Not.in.gene.set")))
deTable
#>      In.gene.set Not.in.gene.set
#> DE          28           501
#> Not.DE     142         12000

```

where the overlap of 28 genes can be assessed based on the hypergeometric distribution. This corresponds to a one-sided version of Fisher's exact test, yielding here a significant enrichment.

```

fisher.test(deTable, alternative = "greater")
#>
#> Fisher's Exact Test for Count Data
#>
#> data: deTable
#> p-value = 4.088e-10
#> alternative hypothesis: true odds ratio is greater than 1
#> 95 percent confidence interval:
#> 3.226736      Inf
#> sample estimates:
#> odds ratio
#> 4.721744

```

This basic principle is at the foundation of major public and commercial enrichment tools such as DAVID and Pathway Studio.

Although gene set enrichment methods have been primarily developed and applied on transcriptomic data, they have recently been modified, extended and applied also in other fields of genomic and biomedical research. This includes novel approaches for functional enrichment analysis of proteomic and metabolomic data as well as genomic regions and disease phenotypes, Lavallee and Yates, 2016, Chagoyen et al., 2016, McLean et al., 2010, Ried et al., 2012.

9.5 Gene expression-based enrichment analysis

The first part of the workshop is largely based on the EnrichmentBrowser package, which implements an analysis pipeline for high-throughput gene expression data as measured with microarrays and RNA-seq. In a workflow-like manner, the package brings together a selection of established Bioconductor packages for gene expression data analysis. It integrates a wide range of gene set enrichment analysis methods and facilitates combination and exploration of results across methods.

```
suppressPackageStartupMessages(library(EnrichmentBrowser))
```

Further information can be found in the vignette and publication.

9.6 A primer on terminology, existing methods & statistical theory

Gene sets, pathways & regulatory networks

Gene sets are simple lists of usually functionally related genes without further specification of relationships between genes.

Pathways can be interpreted as specific gene sets, typically representing a group of genes that work together in a biological process. Pathways are commonly divided in metabolic and signaling pathways. Metabolic pathways such as glycolysis represent biochemical substrate conversions by specific enzymes. Signaling pathways such as the MAPK signaling pathway describe signal transduction cascades from receptor proteins to transcription factors, resulting in activation or inhibition of specific target genes.

Gene regulatory networks describe the interplay and effects of regulatory factors (such as transcription factors and microRNAs) on the expression of their target genes.

Resources

GO and KEGG annotations are most frequently used for the enrichment analysis of functional gene sets. Despite an increasing number of gene set and pathway databases, they are typically the first choice due to their long-standing curation and availability for a wide range of species.

GO: The Gene Ontology (GO) consists of three major sub-ontologies that classify gene products according to molecular function (MF), biological process (BP) and cellular component (CC). Each ontology consists of GO terms that define MFs, BPs or CCs to which specific genes are annotated. The terms are organized in a directed acyclic graph, where edges between the terms represent relationships of different types. They relate the terms according to a parent-child scheme, i.e. parent terms denote more general entities, whereas child terms represent more specific entities.

KEGG: The Kyoto Encyclopedia of Genes and Genomes (KEGG) is a collection of manually drawn pathway maps representing molecular interaction and reaction networks. These pathways cover a wide range of biochemical processes that can be divided in 7 broad categories: metabolism, genetic and environmental information processing, cellular processes, organismal systems, human diseases, and drug development. Metabolism and drug development pathways differ from pathways of the other 5 categories by illustrating reactions between chemical compounds. Pathways of the other 5 categories illustrate molecular interactions between genes and gene products.

Gene set analysis vs. gene set enrichment analysis

The two predominantly used enrichment methods are:

- Overrepresentation analysis (ORA), testing whether a gene set contains disproportional many genes of significant expression change, based on the procedure outlined in the first section
- Gene set enrichment analysis (GSEA), testing whether genes of a gene set accumulate at the top or bottom of the full gene vector ordered by direction and magnitude of expression change Subramanian et al., 2005

However, the term *gene set enrichment analysis* nowadays subsumes a general strategy implemented by a wide range of methods Huang et al., 2009. Those methods have in common the same goal, although approach and statistical model can vary substantially Goeman and Buehlmann, 2007, Khatri et al., 2012.

To better distinguish from the specific method, some authors use the term *gene set analysis* to denote the general strategy. However, there is also a specific method from Efron and Tibshirani, 2007 of this name.

Underlying null: competitive vs. self-contained

Goeman and Buehlmann, 2007 classified existing enrichment methods into *competitive* and *self-contained* based on the underlying null hypothesis.

- *Competitive* null hypothesis: the genes in the set of interest are at most as often DE as the genes not in the set,
- *Self-contained* null hypothesis: no genes in the set of interest are DE.

Although the authors argue that a self-contained null is closer to the actual question of interest, the vast majority of enrichment methods is competitive.

Goeman and Buehlmann further raise several critical issues concerning the 2x2 ORA:

- rather arbitrary classification of genes in DE / not DE

- based on gene sampling, although sampling of subjects is appropriate
- unrealistic independence assumption between genes, resulting in highly anti-conservative p -values

With regard to these statistical concerns, GSEA is considered superior:

- takes all measured genes into account
- subject sampling via permutation of class labels
- the incorporated permutation procedure implicitly accounts for correlations between genes

However, the simplicity and general applicability of ORA is unmet by subsequent methods improving on these issues. For instance, GSEA requires the expression data as input, which is not available for gene lists derived from other experiment types. On the other hand, the involved sample permutation procedure has been proven inaccurate and time-consuming Efron and Tibshirani, 2007, Phipson and Smyth, 2010, Larson and Owen, 2015.

Generations: ora, fcs & topology-based

Khatri et al., 2012 have taken a slightly different approach by classifying methods along the timeline of development into three generations:

1. Generation: ORA methods based on the 2x2 contingency table test,
2. Generation: functional class scoring (FCS) methods such as GSEA, which compute gene set (= functional class) scores by summarizing per-gene DE statistics,
3. Generation: topology-based methods, explicitly taking into account interactions between genes as defined in signaling pathways and gene regulatory networks (Geistlinger et al., 2011 for an example).

Although topology-based (also: network-based) methods appear to be most realistic, their straightforward application can be impaired by features that are not-detectable on the transcriptional level (such as protein-protein interactions) and insufficient network knowledge Geistlinger et al., 2013, Bayerlova et al., 2015.

Given the individual benefits and limitations of existing methods, cautious interpretation of results is required to derive valid conclusions. Whereas no single method is best suited for all application scenarios, applying multiple methods can be beneficial. This has been shown to filter out spurious hits of individual methods, thereby reducing the outcome to gene sets accumulating evidence from different methods Geistlinger et al., 2016, Alhamdoosh et al., 2017.

9.7 Data types

Although RNA-seq (read count data) has become the *de facto* standard for transcriptomic profiling, it is important to know that many methods for differential expression and gene set enrichment analysis have been originally developed for microarray data (intensity measurements).

However, differences in data distribution assumptions (microarray: quasi-normal, RNA-seq: negative binomial) made adaptations in differential expression analysis and, to some extent, also in gene set enrichment analysis necessary.

Thus, we consider two example datasets - a microarray and a RNA-seq dataset, and discuss similarities and differences of the respective analysis steps.

For microarray data, we consider expression measurements of patients with acute lymphoblastic leukemia Chiaretti et al., 2004. A frequent chromosomal defect found among these patients is a translocation, in which parts of chromosome 9 and 22 swap places. This results in the oncogenic fusion gene BCR/ABL created by positioning the ABL1 gene on chromosome 9 to a part of the BCR gene on chromosome 22.

We load the ALL dataset

```
library(ALL)
data(ALL)
```

and select B-cell ALL patients with and without the BCR/ABL fusion, as described previously Gentleman et al., 2005.

```
ind.bs <- grep("^B", ALL$BT)
ind.mut <- which(ALL$mol.biol %in% c("BCR/ABL", "NEG"))
sset <- intersect(ind.bs, ind.mut)
all.eset <- ALL[, sset]
```

We can now access the expression values, which are intensity measurements on a log-scale for 12,625 probes (rows) across 79 patients (columns).

```
dim(all.eset)
#> Features Samples
#> 12625 79
exprs(all.eset)[1:4,1:4]
#> 01005 01010 03002 04007
#> 1000_at 7.597323 7.479445 7.567593 7.905312
#> 1001_at 5.046194 4.932537 4.799294 4.844565
#> 1002_f_at 3.900466 4.208155 3.886169 3.416923
#> 1003_s_at 5.903856 6.169024 5.860459 5.687997
```

As we often have more than one probe per gene, we compute gene expression values as the average of the corresponding probe values.

```
allSE <- probe2gene(all.eset)
#> Loading required package: hgu95av2.db
#> Loading required package: AnnotationDbi
#> Loading required package: org.Hs.eg.db
#>
#>
head(names(allSE))
#> [1] "5595" "7075" "1557" "643" "1843" "4319"
```

For RNA-seq data, we consider transcriptome profiles of four primary human airway smooth muscle cell lines in two conditions: control and treatment with dexamethasone Himes et al., 2014.

We load the airway dataset

```
library(airway)
data(airway)
```

For further analysis, we only keep genes that are annotated to an ENSEMBL gene ID.

```
airSE <- airway[grep("^ENSG", names(airway)), ]
dim(airSE)
#> [1] 63677 8

assay(airSE)[1:4,1:4]
#> SRR1039508 SRR1039509 SRR1039512 SRR1039513
#> ENSG000000000003 679 448 873 408
#> ENSG000000000005 0 0 0 0
#> ENSG00000000419 467 515 621 365
#> ENSG00000000457 260 211 263 164
```

9.8 Differential expression analysis

Normalization of high-throughput expression data is essential to make results within and between experiments comparable. Microarray (intensity measurements) and RNA-seq (read counts) data typically show distinct features that need to be normalized for. As this is beyond the scope of this workshop, we refer to limma for microarray normalization and EDASeq for RNA-seq normalization. See also `EnrichmentBrowser::normalize`, which wraps commonly used functionality for normalization.

The `EnrichmentBrowser` incorporates established functionality from the `limma` package for differential expression analysis. This involves the `voom` transformation when applied to RNA-seq data. Alternatively, differential expression analysis for RNA-seq data can also be carried out based on the negative binomial distribution with `edgeR` and `DESeq2`.

This can be performed using the function `EnrichmentBrowser::deAna` and assumes some standardized variable names:

- **GROUP** defines the sample groups being contrasted,
- **BLOCK** defines paired samples or sample blocks, as e.g. for batch effects.

For more information on experimental design, see the `limma` user's guide, chapter 9.

For the ALL dataset, the **GROUP** variable indicates whether the BCR-ABL gene fusion is present (1) or not (0).

```
allSE$GROUP <- ifelse(allSE$mol.biol == "BCR/ABL", 1, 0)
table(allSE$GROUP)
#>
#> 0 1
#> 42 37
```

For the airway dataset, it indicates whether the cell lines have been treated with dexamethasone (1) or not (0).

```
airSE$GROUP <- ifelse(colData(airway)$dex == "trt", 1, 0)
table(airSE$GROUP)
#>
#> 0 1
#> 4 4
```

Paired samples, or in general sample batches/blocks, can be defined via a **BLOCK** column in the `colData` slot. For the airway dataset, the sample blocks correspond to the four different cell lines.

```
airSE$BLOCK <- airway$cell
table(airSE$BLOCK)
#>
#> N052611 N061011 N080611 N61311
#> 2 2 2 2
```

For microarray data, the `EnrichmentBrowser::deAna` function carries out differential expression analysis based on functionality from the `limma` package. Resulting log2 fold changes and *t*-test derived *p*-values for each gene are appended to the `rowData` slot.

```
allSE <- deAna(allSE)
rowData(allSE, use.names=TRUE)
#> DataFrame with 9010 rows and 3 columns
#>          FC          ADJ.PVAL      limma.STAT
#>      <numeric>      <numeric>      <numeric>
#> 5595  0.0429698599842595 0.8992468173107715 0.734679177472013
#> 7075  0.0320835027449625 0.949001013642671 0.4546910829318
```

```
#> 1557 -0.0439401425131442 0.818330132411339 -1.06578261967549
#> 643 -0.0277543539240438 0.929148567589577 -0.567394394416651
#> 1843 -0.427302534257363 0.566034751753148 -1.75050227190017
#> ...
#> ... ...
#> 6300 -0.026651766164237 0.922828548631225 -0.608608859328046
#> 7297 -0.124257678078831 0.804578494190681 -1.11279493778184
#> 2246 0.0522428857778935 0.748021044717352 1.27408420746691
#> 7850 -0.00908229596065303 0.991826450687159 -0.102406339091096
#> 1593 -0.00747713820802068 0.989532971314233 -0.145650256847251
```

Nominal p -values are already corrected for multiple testing (ADJ.PVAL) using the method from Benjamini and Hochberg implemented in `stats:::p.adjust`.

For RNA-seq data, the `deAna` function can be used to carry out differential expression analysis between the two groups either based on functionality from `limma` (that includes the `voom` transformation), or alternatively, the frequently used `edgeR` or `DESeq2` package. Here, we use the analysis based on `edgeR`.

```
airSE <- deAna(airSE, de.method="edgeR")
#> Excluding 50740 genes not satisfying min.cpm threshold

rowData(airSE, use.names=TRUE)
#> DataFrame with 12937 rows and 3 columns
#>          FC           ADJ.PVAL
#>          <numeric>      <numeric>
#> ENSG000000000003 -0.404945626610932 0.00213458295385943
#> ENSG000000000419 0.182985434777532 0.0915691945172958
#> ENSG000000000457 0.0143477674070903 0.922279475399443
#> ENSG000000000460 -0.141173372957311 0.619013213521635
#> ENSG000000000971 0.402240426474172 0.00403820532305827
#> ...
#> ... ...
#> ENSG00000273270 -0.129793853337261 0.495892935815041
#> ENSG00000273290 0.505580471641003 0.006392183877102899
#> ENSG00000273311 0.00161557580855148 0.996356136956657
#> ENSG00000273329 -0.222817127090519 0.388294594068803
#> ENSG00000273344 0.0151704005097403 0.9627771106053257
#>          edgeR.STAT
#>          <numeric>
#> ENSG000000000003 35.8743710016552
#> ENSG000000000419 5.90960619951737
#> ENSG000000000457 0.0233923316990905
#> ENSG000000000460 0.492929955080604
#> ENSG000000000971 27.8509962017407
#> ...
#> ... ...
#> ENSG00000273270 0.901598359265205
#> ENSG00000273290 23.0905678847871
#> ENSG00000273311 8.04821152029429e-05
#> ENSG00000273329 1.42723325850597
#> ENSG00000273344 0.00543503273765429
```

Exercise: Compare the number of differentially expressed genes as obtained on the `airSE` with `limma`/`voom`, `edgeR`, and `DESeq2`.

9.9 Gene sets

We are now interested in whether pre-defined sets of genes that are known to work together, e.g. as defined in the Gene Ontology or the KEGG pathway annotation, are coordinately differentially expressed.

The function `getGenesets` can be used to download gene sets from databases such as GO and KEGG. Here, we use the function to download all KEGG pathways for a chosen organism (here: *Homo sapiens*) as gene sets.

```
kegg.gs <- getGenesets(org="hsa", db="kegg")
```

Analogously, the function `getGenesets` can be used to retrieve GO terms of a selected ontology (here: biological process, BP) as defined in the `GO.db` annotation package.

```
go.gs <- getGenesets(org="hsa", db="go", go.onto="BP", go.mode="GO.db")
#>
```

If provided a file, the function `getGenesets` parses user-defined gene sets from GMT file format. Here, we use this functionality for reading a list of already downloaded KEGG gene sets for *Homo sapiens* containing NCBI Entrez Gene IDs.

```
data.dir <- system.file("extdata", package="EnrichmentBrowser")
gmt.file <- file.path(data.dir, "hsa_kegg_gs.gmt")
hsa.gs <- getGenesets(gmt.file)
hsa.gs[1:2]
#> $hsa05416_Viral_myocarditis
#> [1] "100509457" "101060835" "1525"      "1604"      "1605"
#> [6] "1756"       "1981"       "1982"      "25"        "2534"
#> [11] "27"         "3105"       "3106"      "3107"      "3108"
#> [16] "3109"       "3111"       "3112"      "3113"      "3115"
#> [21] "3117"       "3118"       "3119"      "3122"      "3123"
#> [26] "3125"       "3126"       "3127"      "3133"      "3134"
#> [31] "3135"       "3383"       "3683"      "3689"      "3908"
#> [36] "4624"       "4625"       "54205"     "5551"      "5879"
#> [41] "5880"       "5881"       "595"       "60"        "637"
#> [46] "6442"       "6443"       "6444"      "6445"      "71"
#> [51] "836"        "841"        "842"       "857"       "8672"
#> [56] "940"        "941"        "942"       "958"       "959"
#>
#> $`hsa04622_RIG-I-like_receptor_signaling_pathway`
#> [1] "10010"      "1147"      "1432"      "1540"      "1654"      "23586"    "26007"
#> [8] "29110"      "338376"    "340061"    "3439"      "3440"      "3441"      "3442"
#> [15] "3443"       "3444"       "3445"      "3446"      "3447"      "3448"      "3449"
#> [22] "3451"       "3452"       "3456"      "3467"      "3551"      "3576"      "3592"
#> [29] "3593"       "3627"       "3661"      "3665"      "4214"      "4790"      "4792"
#> [36] "4793"       "5300"       "54941"     "55593"     "5599"      "5600"      "5601"
#> [43] "5602"       "5603"       "56832"     "57506"     "5970"      "6300"      "64135"
#> [50] "64343"      "6885"      "7124"      "7186"      "7187"      "7189"      "7706"
#> [57] "79132"      "79671"     "80143"     "841"       "843"       "8517"      "8717"
#> [64] "8737"       "8772"       "9140"      "9474"      "9636"      "9641"      "9755"
hsa.gs[1:2]
#> $hsa05416_Viral_myocarditis
#> [1] "100509457" "101060835" "1525"      "1604"      "1605"
#> [6] "1756"       "1981"       "1982"      "25"        "2534"
#> [11] "27"         "3105"       "3106"      "3107"      "3108"
```

```
#> [16] "3109"      "3111"      "3112"      "3113"      "3115"
#> [21] "3117"      "3118"      "3119"      "3122"      "3123"
#> [26] "3125"      "3126"      "3127"      "3133"      "3134"
#> [31] "3135"      "3383"      "3683"      "3689"      "3908"
#> [36] "4624"      "4625"      "54205"     "5551"      "5879"
#> [41] "5880"      "5881"      "595"       "60"        "637"
#> [46] "6442"      "6443"      "6444"      "6445"      "71"
#> [51] "836"       "841"       "842"       "857"       "8672"
#> [56] "940"       "941"       "942"       "958"       "959"
#>
#> $`hsa04622_RIG-I-like_receptor_signaling_pathway`
#> [1] "10010"     "1147"      "1432"      "1540"      "1654"      "23586"     "26007"
#> [8] "29110"     "338376"    "340061"    "3439"      "3440"      "3441"      "3442"
#> [15] "3443"      "3444"      "3445"      "3446"      "3447"      "3448"      "3449"
#> [22] "3451"      "3452"      "3456"      "3467"      "3551"      "3576"      "3592"
#> [29] "3593"      "3627"      "3661"      "3665"      "4214"      "4790"      "4792"
#> [36] "4793"      "5300"      "54941"     "55593"     "5599"      "5600"      "5601"
#> [43] "5602"      "5603"      "56832"     "57506"     "5970"      "6300"      "64135"
#> [50] "64343"     "6885"      "7124"      "7186"      "7187"      "7189"      "7706"
#> [57] "79132"     "79671"     "80143"     "841"       "843"       "8517"      "8717"
#> [64] "8737"      "8772"      "9140"      "9474"      "9636"      "9641"      "9755"
```

See also the MSigDb for additional gene set collections.

9.10 GO/KEGG overrepresentation analysis

A variety of gene set analysis methods have been proposed Khatri et al., 2012. The most basic, yet frequently used, method is the over-representation analysis (ORA) with gene sets defined according to GO or KEGG. As outlined in the first section, ORA tests the overlap between DE genes (typically DE p -value < 0.05) and genes in a gene set based on the hypergeometric distribution. Here, we choose a significance level $\alpha = 0.2$ for demonstration.

```
ora.all <- sbea(method="ora", se=allSE, gs=hsa.gs, perm=0, alpha=0.2)
gsRanking(ora.all)
#> DataFrame with 7 rows and 4 columns
#>                                         GENE.SET
#>                                         <character>
#> 1          hsa05202_Transcriptional_misregulation_in_cancer
#> 2 hsa05412_Arrhythmogenic_right_ventricular_cardiomyopathy_(ARVC)
#> 3                                     hsa05144_Malaria
#> 4          hsa04670_Leukocyte_transendothelial_migration
#> 5          hsa05100_Bacterial_invasion_of_epithelial_cells
#> 6 hsa04622_RIG-I-like_receptor_signaling_pathway
#> 7          hsa05130_Pathogenic_Escherichia_coli_infection
#>   NR.GENES NR.SIG.GENES P.VALUE
#>   <numeric>   <numeric>   <numeric>
#> 1      153         17  0.0351
#> 2       63          8  0.0717
#> 3       45          6  0.0932
#> 4       94         10  0.122
#> 5       64          7  0.162
#> 6       54          6  0.178
#> 7       43          5  0.184
```

Such a ranked list is the standard output of most existing enrichment tools. Using the `eaBrowse` function creates a HTML summary from which each gene set can be inspected in more detail.

```
eaBrowse(ora.all)
#> Creating gene report ...
#>
#> Creating set view ...
#> Creating kegg view ...
#> Loading required package: pathview
```

The resulting summary page includes for each significant gene set

- a gene report, which lists all genes of a set along with fold change and DE *p*-value (click on links in column `NR.GENES`),
- interactive overview plots such as heatmap and volcano plot (column `SET.VIEW`, supports mouse-over and click-on),
- for KEGG pathways: highlighting of differentially expressed genes on the pathway maps (column `PATH.VIEW`, supports mouse-over and click-on).

As ORA works on the list of DE genes and not the actual expression values, it can be straightforward applied to RNA-seq data. However, as the gene sets here contain NCBI Entrez gene IDs and the airway dataset contains ENSEMBL gene ids, we first map the airway dataset to Entrez IDs.

```
airSE <- idMap(airSE, org="hsa", from="ENSEMBL", to="ENTREZID")
#> 'select()' returned 1:many mapping between keys and columns
#> Excluded 1133 genes without a corresponding to.ID
#> Encountered 8 from.IDs with >1 corresponding to.ID (a single to.ID was chosen for each of them)

ora.air <- sbea(method="ora", se=airSE, gs=hsa.gs, perm=0)
gsRanking(ora.air)
#> DataFrame with 9 rows and 4 columns
#>          GENE.SET
#>          <character>
#> 1      hsa05206_MicroRNAs_in_cancer
#> 2      hsa05218_Melanoma
#> 3      hsa05214_Glioma
#> 4      hsa05131_Shigellosis
#> 5      hsa05410_Hypertrophic_cardiomyopathy_(HCM)
#> 6      hsa04670_Leukocyte_transendothelial_migration
#> 7      hsa05100_Bacterial_invasion_of_epithelial_cells
#> 8      hsa04514_Cell_adhesion_molecules_(CAMs)
#> 9      hsa05412_Arrhythmogenic_right_ventricular_cardiomyopathy_(ARVC)
#>   NR.GENES NR.SIG.GENES P.VALUE
#>   <numeric>   <numeric>   <numeric>
#> 1      118      68  0.000508
#> 2      50       33  0.000662
#> 3      48       30  0.00419
#> 4      53       31  0.0142
#> 5      53       31  0.0142
#> 6      62       34  0.0353
#> 7      60       33  0.036
#> 8      60       33  0.036
#> 9      48       27  0.0402
```

Note #1: Young et al., 2010, have reported biased results for ORA on RNA-seq data due to over-detection

of differential expression for long and highly expressed transcripts. The goseq package and `limma::goana` implement possibilities to adjust ORA for gene length and abundance bias.

Note #2: Independent of the expression data type under investigation, overlap between gene sets can result in redundant findings. This is well-documented for GO (parent-child structure, Rhee et al., 2008) and KEGG (pathway overlap/crosstalk, Donato et al., 2013). The topGO package (explicitly designed for GO) and mgsa (applicable to arbitrary gene set definitions) implement modifications of ORA to account for such redundancies.

9.11 Functional class scoring & permutation testing

A major limitation of ORA is that it restricts analysis to DE genes, excluding genes not satisfying the chosen significance threshold (typically the vast majority).

This is resolved by gene set enrichment analysis (GSEA), which scores the tendency of gene set members to appear rather at the top or bottom of the ranked list of all measured genes Subramanian et al., 2005. The statistical significance of the enrichment score (ES) of a gene set is assessed via sample permutation, i.e. (1) sample labels (= group assignment) are shuffled, (2) per-gene DE statistics are recomputed, and (3) the enrichment score is recomputed. Repeating this procedure many times allows to determine the empirical distribution of the enrichment score and to compare the observed enrichment score against it. Here, we carry out GSEA with 1000 permutations.

```
gsea.all <- sbea(method="gsea", se=allSE, gs=hsa.gs, perm=1000)
#> Permutations: 1 -- 100
#> Permutations: 101 -- 200
#> Permutations: 201 -- 300
#> Permutations: 301 -- 400
#> Permutations: 401 -- 500
#> Permutations: 501 -- 600
#> Permutations: 601 -- 700
#> Permutations: 701 -- 800
#> Permutations: 801 -- 900
#> Permutations: 901 -- 1000
#> Processing ...
```

```
gsRanking(gsea.all)
#> DataFrame with 20 rows and 4 columns
#>                                         GENE.SET
#>                                         <character>
#> 1   hsa05412_Arrhythmogenic_right_ventricular_cardiomyopathy_(ARVC)
#> 2                               hsa04670_Leukocyte_transendothelial_migration
#> 3                               hsa04520_Adherens_junction
#> 4                               hsa04390_Hippo_signaling_pathway
#> 5                               hsa05323_Rheumatoid_arthritis
#> ...
#> 16                             hsa05217_Basal_cell_carcinoma
#> 17                             hsa04210_Apoptosis
#> 18                             hsa05130_Pathogenic_Escherichia_coli_infection
#> 19                             hsa05410_Hypertrophic_cardiomyopathy_(HCM)
#> 20                             hsa05131_Shigellosis
#>             ES      NES    P.VALUE
#>     <numeric> <numeric> <numeric>
#> 1      0.511    1.92      0
#> 2      0.499    1.78      0
```

```
#> 3      0.488    1.74      0
#> 4      0.459    1.67      0
#> 5      0.574    1.66    0.0019
#> ...    ...
#> 16     0.559    1.64    0.0248
#> 17     0.424    1.44    0.0336
#> 18     0.486    1.54    0.0347
#> 19     0.386    1.45    0.0406
#> 20     0.479    1.49    0.0436
```

As GSEA's permutation procedure involves re-computation of per-gene DE statistics, adaptations are necessary for RNA-seq. The EnrichmentBrowser implements an accordingly adapted version of GSEA, which allows incorporation of limma/voom, edgeR, or DESeq2 for repeated DE re-computation within GSEA. However, this is computationally intensive (for limma/voom the least, for DESeq2 the most). Note the relatively long running times for only 100 permutations having used edgeR for DE analysis.

```
gsea.air <- sbea(method="gsea", se=airSE, gs=hsa.gs, perm=100)
#> 100 permutations completed
```

While it might be in some cases necessary to apply permutation-based GSEA for RNA-seq data, there are also alternatives avoiding permutation. Among them is ROtAtion gene Set Testing (ROAST), which uses rotation instead of permutation Wu et al., 2010.

```
roast.air <- sbea(method="roast", se=airSE, gs=hsa.gs)
gsRanking(roast.air)
#> DataFrame with 27 rows and 4 columns
#>
#>          GENE.SET    NR.GENES      DIR
#>          <character> <numeric> <numeric>
#> 1  hsa05410_Hypertrophic_cardiomyopathy_(HCM)      53      1
#> 2  hsa05134_Legionellosis                  35      1
#> 3  hsa05416_Viral_myocarditis                 33      1
#> 4  hsa00790_Folate_biosynthesis                11      1
#> 5  hsa03030_DNA_replication                  33     -1
#> ...
#>       ...
#> 23  hsa04150_mTOR_signaling_pathway              50      1
#> 24  hsa04350_TGF-beta_signaling_pathway            63      1
#> 25  hsa00561_Glycerolipid_metabolism               39      1
#> 26  hsa04621_NOD-like_receptor_signaling_pathway   40     -1
#> 27  hsa04514_Cell_adhesion_molecules_(CAMs)        60     -1
#>          P.VALUE
#>          <numeric>
#> 1      0.001
#> 2      0.001
#> 3      0.001
#> 4      0.001
#> 5      0.001
#> ...
#>       ...
#> 23     0.027
#> 24     0.029
#> 25     0.032
#> 26     0.033
#> 27     0.035
```

A selection of additional methods is also available:

```
sbeaMethods()
#> [1] "ora"          "safe"         "gsea"        "gsa"         "padog"
#> [6] "globaltest"   "roast"        "camera"      "gsva"        "samgs"
#> [11] "ebm"         "mgsa"
```

Exercise: Carry out a GO overrepresentation analysis for the `allSE` and `airSE`. How many significant gene sets do you observe in each case?

9.12 Network-based enrichment analysis

Having found gene sets that show enrichment for differential expression, we are now interested whether these findings can be supported by known regulatory interactions.

For example, we want to know whether transcription factors and their target genes are expressed in accordance to the connecting regulations (activation/inhibition). Such information is usually given in a gene regulatory network derived from specific experiments or compiled from the literature (Geistlinger et al., 2013 for an example).

There are well-studied processes and organisms for which comprehensive and well-annotated regulatory networks are available, e.g. the RegulonDB for *E. coli* and YeastRACT for *S. cerevisiae*.

However, there are also cases where such a network is missing or at least incomplete. A basic workaround is to compile a network from regulations in pathway databases such as KEGG.

```
hsa.grn <- compileGRN(org="hsa", db="kegg")
head(hsa.grn)
#>    FROM      TO      TYPE
#> [1,] "10000"  "100132074"  "-"
#> [2,] "10000"  "1026"      "-"
#> [3,] "10000"  "1026"      "+"
#> [4,] "10000"  "1027"      "-"
#> [5,] "10000"  "10488"    "+"
#> [6,] "10000"  "107"       "+"
```

Signaling pathway impact analysis (SPIA) is a network-based enrichment analysis method, which is explicitly designed for KEGG signaling pathways Tarca et al., 2009. The method evaluates whether expression changes are propagated across the pathway topology in combination with ORA.

```
spia.all <- nbea(method="spia", se=allSE, gs=hsa.gs, grn=hsa.grn, alpha=0.2)
#>
#> Done pathway 1 : RNA transport..
#> Done pathway 2 : RNA degradation..
#> Done pathway 3 : PPAR signaling pathway..
#> Done pathway 4 : Fanconi anemia pathway..
#> Done pathway 5 : MAPK signaling pathway..
#> Done pathway 6 : ErbB signaling pathway..
#> Done pathway 7 : Calcium signaling pathway..
#> Done pathway 8 : Cytokine-cytokine receptor int..
#> Done pathway 9 : Chemokine signaling pathway..
#> Done pathway 10 : NF-kappa B signaling pathway..
#> Done pathway 11 : Phosphatidylinositol signaling..
#> Done pathway 12 : Neuroactive ligand-receptor in..
#> Done pathway 13 : Cell cycle..
#> Done pathway 14 : Oocyte meiosis..
#> Done pathway 15 : p53 signaling pathway..
```

```
#> Done pathway 16 : Sulfur relay system..
#> Done pathway 17 : SNARE interactions in vesicula..
#> Done pathway 18 : Regulation of autophagy..
#> Done pathway 19 : Protein processing in endoplas..
#> Done pathway 20 : Lysosome..
#> Done pathway 21 : mTOR signaling pathway..
#> Done pathway 22 : Apoptosis..
#> Done pathway 23 : Vascular smooth muscle contrac..
#> Done pathway 24 : Wnt signaling pathway..
#> Done pathway 25 : Dorso-ventral axis formation..
#> Done pathway 26 : Notch signaling pathway..
#> Done pathway 27 : Hedgehog signaling pathway..
#> Done pathway 28 : TGF-beta signaling pathway..
#> Done pathway 29 : Axon guidance..
#> Done pathway 30 : VEGF signaling pathway..
#> Done pathway 31 : Osteoclast differentiation..
#> Done pathway 32 : Focal adhesion..
#> Done pathway 33 : ECM-receptor interaction..
#> Done pathway 34 : Cell adhesion molecules (CAMs)..
#> Done pathway 35 : Adherens junction..
#> Done pathway 36 : Tight junction..
#> Done pathway 37 : Gap junction..
#> Done pathway 38 : Complement and coagulation cas..
#> Done pathway 39 : Antigen processing and present..
#> Done pathway 40 : Toll-like receptor signaling p..
#> Done pathway 41 : NOD-like receptor signaling pa..
#> Done pathway 42 : RIG-I-like receptor signaling ..
#> Done pathway 43 : Cytosolic DNA-sensing pathway..
#> Done pathway 44 : Jak-STAT signaling pathway..
#> Done pathway 45 : Natural killer cell mediated c..
#> Done pathway 46 : T cell receptor signaling path..
#> Done pathway 47 : B cell receptor signaling path..
#> Done pathway 48 : Fc epsilon RI signaling pathwa..
#> Done pathway 49 : Fc gamma R-mediated phagocytos..
#> Done pathway 50 : Leukocyte transendothelial mig..
#> Done pathway 51 : Intestinal immune network for ..
#> Done pathway 52 : Circadian rhythm - mammal..
#> Done pathway 53 : Long-term potentiation..
#> Done pathway 54 : Neurotrophin signaling pathway..
#> Done pathway 55 : Retrograde endocannabinoid sig..
#> Done pathway 56 : Glutamatergic synapse..
#> Done pathway 57 : Cholinergic synapse..
#> Done pathway 58 : Serotonergic synapse..
#> Done pathway 59 : GABAergic synapse..
#> Done pathway 60 : Dopaminergic synapse..
#> Done pathway 61 : Long-term depression..
#> Done pathway 62 : Olfactory transduction..
#> Done pathway 63 : Taste transduction..
#> Done pathway 64 : Phototransduction..
#> Done pathway 65 : Regulation of actin cytoskelet..
#> Done pathway 66 : Insulin signaling pathway..
#> Done pathway 67 : GnRH signaling pathway..
#> Done pathway 68 : Progesterone-mediated oocyte m..
```

```
#> Done pathway 69 : Melanogenesis..
#> Done pathway 70 : Adipocytokine signaling pathwa..
#> Done pathway 71 : Type II diabetes mellitus..
#> Done pathway 72 : Type I diabetes mellitus..
#> Done pathway 73 : Maturity onset diabetes of the..
#> Done pathway 74 : Aldosterone-regulated sodium r..
#> Done pathway 75 : Endocrine and other factor-reg..
#> Done pathway 76 : Vasopressin-regulated water re..
#> Done pathway 77 : Salivary secretion..
#> Done pathway 78 : Gastric acid secretion..
#> Done pathway 79 : Pancreatic secretion..
#> Done pathway 80 : Carbohydrate digestion and abs..
#> Done pathway 81 : Bile secretion..
#> Done pathway 82 : Mineral absorption..
#> Done pathway 83 : Alzheimer's disease..
#> Done pathway 84 : Parkinson's disease..
#> Done pathway 85 : Amyotrophic lateral sclerosis ..
#> Done pathway 86 : Huntington's disease..
#> Done pathway 87 : Prion diseases..
#> Done pathway 88 : Cocaine addiction..
#> Done pathway 89 : Amphetamine addiction..
#> Done pathway 90 : Morphine addiction..
#> Done pathway 91 : Alcoholism..
#> Done pathway 92 : Bacterial invasion of epitheli..
#> Done pathway 93 : Vibrio cholerae infection..
#> Done pathway 94 : Epithelial cell signaling in H..
#> Done pathway 95 : Pathogenic Escherichia coli in..
#> Done pathway 96 : Shigellosis..
#> Done pathway 97 : Salmonella infection..
#> Done pathway 98 : Pertussis..
#> Done pathway 99 : Legionellosis..
#> Done pathway 100 : Leishmaniasis..
#> Done pathway 101 : Chagas disease (American trypa..
#> Done pathway 102 : African trypanosomiasis..
#> Done pathway 103 : Malaria..
#> Done pathway 104 : Toxoplasmosis..
#> Done pathway 105 : Amoebiasis..
#> Done pathway 106 : Staphylococcus aureus infectio..
#> Done pathway 107 : Tuberculosis..
#> Done pathway 108 : Hepatitis C..
#> Done pathway 109 : Measles..
#> Done pathway 110 : Influenza A..
#> Done pathway 111 : HTLV-I infection..
#> Done pathway 112 : Herpes simplex infection..
#> Done pathway 113 : Epstein-Barr virus infection..
#> Done pathway 114 : Pathways in cancer..
#> Done pathway 115 : Transcriptional misregulation ..
#> Done pathway 116 : Viral carcinogenesis..
#> Done pathway 117 : Colorectal cancer..
#> Done pathway 118 : Renal cell carcinoma..
#> Done pathway 119 : Pancreatic cancer..
#> Done pathway 120 : Endometrial cancer..
#> Done pathway 121 : Glioma..
```

```

#> Done pathway 122 : Prostate cancer..
#> Done pathway 123 : Thyroid cancer..
#> Done pathway 124 : Basal cell carcinoma..
#> Done pathway 125 : Melanoma..
#> Done pathway 126 : Bladder cancer..
#> Done pathway 127 : Chronic myeloid leukemia..
#> Done pathway 128 : Acute myeloid leukemia..
#> Done pathway 129 : Small cell lung cancer..
#> Done pathway 130 : Non-small cell lung cancer..
#> Done pathway 131 : Asthma..
#> Done pathway 132 : Autoimmune thyroid disease..
#> Done pathway 133 : Systemic lupus erythematosus..
#> Done pathway 134 : Rheumatoid arthritis..
#> Done pathway 135 : Allograft rejection..
#> Done pathway 136 : Graft-versus-host disease..
#> Done pathway 137 : Arrhythmogenic right ventricular..
#> Done pathway 138 : Dilated cardiomyopathy..
#> Done pathway 139 : Viral myocarditis..
#> Finished SPIA analysis
gsRanking(spia.all)
#> DataFrame with 7 rows and 6 columns
#>
#>          GENE.SET      SIZE      NDE
#>          <character> <numeric> <numeric>
#> 1  hsa04620_Toll-like_receptor_signaling_pathway    74       6
#> 2  hsa05202_Transcriptional_misregulation_in_cancer    89       9
#> 3           hsa05416_Viral_myocarditis    45       5
#> 4  hsa04630_Jak-STAT_signaling_pathway    72       8
#> 5  hsa04910_Insulin_signaling_pathway   115       5
#> 6  hsa05143_African_trypansomiasis    22       4
#> 7  hsa04978_Mineral_absorption      3       1
#>
#>     T.ACT      STATUS     P.VALUE
#>     <numeric> <numeric> <numeric>
#> 1    -3.62      -1    0.0497
#> 2    -0.209     -1    0.0726
#> 3     2.98       1    0.0768
#> 4    -1.06      -1    0.0861
#> 5    -7.35      -1    0.153
#> 6     0.263      1    0.193
#> 7      0       -1    0.199

```

More generally applicable is gene graph enrichment analysis (GGEA), which evaluates consistency of interactions in a given gene regulatory network with the observed expression data Geistlinger et al., 2011.

```

ggea.all <- nbea(method="ggea", se=allSE, gs=hsa.gs, grn=hsa.grn)
gsRanking(ggea.all)
#> DataFrame with 9 rows and 5 columns
#>
#>          GENE.SET
#>          <character>
#> 1  hsa04390_Hippo_signaling_pathway
#> 2           hsa05416_Viral_myocarditis
#> 3  hsa05217_Basal_cell_carcinoma
#> 4           hsa04520_Adherens_junction
#> 5  hsa04350_TGF-beta_signaling_pathway
#> 6  hsa05412_Arrhythmogenic_right_ventricular_cardiomyopathy_(ARVC)

```

```

#> 7                               hsa04910_Insulin_signaling_pathway
#> 8                               hsa04622_RIG-I-like_receptor_signaling_pathway
#> 9                               hsa04210_Apoptosis
#>      NR.RELS RAW.SCORE NORM.SCORE   P.VALUE
#>      <numeric> <numeric> <numeric> <numeric>
#> 1       62     22.8    0.367    0.002
#> 2        7      3.3     0.471    0.003
#> 3       18     6.92    0.385    0.00799
#> 4       11      4.5     0.409    0.014
#> 5       14     5.36    0.383    0.024
#> 6        4      1.75    0.437    0.032
#> 7       31     11.1    0.359    0.039
#> 8       35     12.5    0.356    0.041
#> 9       47     16.5    0.35     0.044

nbeaMethods()
#> [1] "ggea"          "spia"           "pathnet"         "degraph"        "ganpa"
#> [6] "cepa"           "topologygsa"   "netgsa"

```

Note #1: As network-based enrichment methods typically do not involve sample permutation but rather network permutation, thus avoiding DE re-computation, they can likewise be applied to RNA-seq data.

Note #2: Given the various enrichment methods with individual benefits and limitations, combining multiple methods can be beneficial, e.g. combined application of a set-based and a network-based method. This has been shown to filter out spurious hits of individual methods and to reduce the outcome to gene sets accumulating evidence from different methods Geistlinger et al., 2016, Alhamdoosh et al., 2017.

The function `combResults` implements the straightforward combination of results, thereby facilitating seamless comparison of results across methods. For demonstration, we use the ORA and GSEA results for the ALL dataset from the previous section:

```

res.list <- list(ora.all, gsea.all)
comb.res <- combResults(res.list)
gsRanking(comb.res)
#> DataFrame with 20 rows and 6 columns
#>                                         GENE.SET
#>                                         <character>
#> 1  hsa05412_Arrhythmogenic_right_ventricular_cardiomyopathy_(ARVC)
#> 2  hsa05202_Transcriptional_misregulation_in_cancer
#> 3  hsa04670_Leukocyte_transendothelial_migration
#> 4  hsa04520_Adherens_junction
#> 5  hsa05206_MicroRNAs_in_cancer
#> ...
#> 16  hsa05410_Hypertrophic_cardiomyopathy_(HCM)
#> 17  hsa04514_Cell_adhesion_molecules_(CAMs)
#> 18  hsa04621_NOD-like_receptor_signaling_pathway
#> 19  hsa04622_RIG-I-like_receptor_signaling_pathway
#> 20  hsa04350_TGF-beta_signaling_pathway
#>      ORA.RANK GSEA.RANK MEAN.RANK  ORA.PVAL GSEA.PVAL
#>      <numeric> <numeric> <numeric> <numeric> <numeric>
#> 1       5.1     10.3     7.7     0.0717     0
#> 2       2.6     17.9    10.3     0.0351  0.00374
#> 3      10.3     10.3     10.3     0.122      0
#> 4      20.5     10.3     15.4     0.201      0
#> 5      23.1     25.6     24.4     0.225  0.00781

```

```
#> ...
#> 16      41     48.7    44.9    0.403   0.0406
#> 17     69.2    23.1    46.2    0.649   0.0058
#> 18     61.5    33.3    47.4    0.631   0.0172
#> 19     15.4    84.6     50     0.178   0.354
#> 20     38.5    61.5     50     0.39    0.113
```

Exercise: Carry out SPIA and GGEA for the airSE and combine the results. How many gene sets are rendered significant by both methods?

9.13 Genomic region enrichment analysis

Microarrays and next-generation sequencing are also widely applied for large-scale detection of variable and regulatory genomic regions, e.g. single nucleotide polymorphisms, copy number variations, and transcription factor binding sites.

Such experimentally-derived genomic region sets are raising similar questions regarding functional enrichment as in gene expression data analysis.

Of particular interest is thereby whether experimentally-derived regions overlap more (enrichment) or less (depletion) than expected by chance with regions representing known functional features such as genes or promoters.

The regioneR package implements a general framework for testing overlaps of genomic regions based on permutation sampling. This allows to repeatedly sample random regions from the genome, matching size and chromosomal distribution of the region set under study. By recomputing the overlap with the functional features in each permutation, statistical significance of the observed overlap can be assessed.

```
suppressPackageStartupMessages(library(regioneR))
```

To demonstrate the basic functionality of the package, we consider the overlap of gene promoter regions and CpG islands in the human genome. We expect to find an enrichment as promoter regions are known to be GC-rich. Hence, is the overlap between CpG islands and promoters greater than expected by chance?

We use the collection of CpG islands described in Wu et al., 2010 and restrict them to the set of canonical chromosomes 1-23, X, and Y.

```
cpgHMM <- toGRanges("http://www.haowulab.org/software/makeCGI/model-based-cpg-islands-hg19.txt")
cpgHMM <- filterChromosomes(cpgHMM, chr.type="canonical")
cpgHMM <- sort(cpgHMM)
cpgHMM

#> GRanges object with 63705 ranges and 5 metadata columns:
#>           seqnames      ranges strand |  length  CpGcount
#>           <Rle>      <IRanges> <Rle> | <integer> <integer>
#> [1]   chr1    10497-11241    * /      745     110
#> [2]   chr1    28705-29791    * /     1087     115
#> [3]   chr1    135086-135805   * /      720      42
#> [4]   chr1    136164-137362   * /     1199      71
#> [5]   chr1    137665-138121   * /      457      22
#> ...
#> [63701] chrY 59213702-59214290   * /      589      43
#> [63702] chrY 59240512-59241057   * /      546      40
#> [63703] chrY 59348047-59348370   * /      324      17
#> [63704] chrY 59349137-59349565   * /      429      31
#> [63705] chrY 59361489-59362401   * /      913     128
```

```

#>      GCcontent     pctGC    obsExp
#>      <integer> <numeric> <numeric>
#> [1]      549     0.737    1.106
#> [2]      792     0.729    0.818
#> [3]      484     0.672    0.548
#> [4]      832     0.694    0.524
#> [5]      301     0.659    0.475
#> ...
#> [63701]   366     0.621    0.765
#> [63702]   369     0.676    0.643
#> [63703]   193     0.596    0.593
#> [63704]   276     0.643    0.7
#> [63705]   650     0.712    1.108
#> -----
#> seqinfo: 24 sequences from an unspecified genome; no seqlengths

```

Analogously, we load promoter regions in the *hg19* human genome assembly as available from UCSC:

```

promoters <- toGRanges("http://gattaca.imppc.org/regioner/data/UCSC.promoters.hg19.bed")
promoters <- filterChromosomes(promoters, chr.type="canonical")
promoters <- sort(promoters)
promoters

#> GRanges object with 49049 ranges and 3 metadata columns:
#>      seqnames      ranges strand | V4      V5      V6
#>      <Rle>      <IRanges> <Rle> | <factor> <factor> <factor>
#> [1] chr1    9873-12073 * / TSS . +
#> [2] chr1    16565-18765 * / TSS . -
#> [3] chr1    17551-19751 * / TSS . -
#> [4] chr1    17861-20061 * / TSS . -
#> [5] chr1    19559-21759 * / TSS . -
#> ...
#> [49045] chrY 59211948-59214148 * / TSS . +
#> [49046] chrY 59328251-59330451 * / TSS . +
#> [49047] chrY 59350972-59353172 * / TSS . +
#> [49048] chrY 59352984-59355184 * / TSS . +
#> [49049] chrY 59360654-59362854 * / TSS . -
#> -----
#> seqinfo: 24 sequences from an unspecified genome; no seqlengths

```

To speed up the example, we restrict analysis to chromosomes 21 and 22. Note that this is done for demonstration only. To make an accurate claim, the complete region set should be used (which, however, runs considerably longer).

```

cpg <- cpgHMM[seqnames(cpgHMM) %in% c("chr21", "chr22")]
prom <- promoters[seqnames(promoters) %in% c("chr21", "chr22")]

```

Now, we are applying an overlap permutation test with 100 permutations (`ntimes=100`), while maintaining chromosomal distribution of the CpG island region set (`per.chromosome=TRUE`). Furthermore, we use the option `count.once=TRUE` to count an overlapping CpG island only once, even if it overlaps with 2 or more promoters.

Note that we use 100 permutations for demonstration only. To draw robust conclusions a minimum of 1000 permutations should be carried out.

```

pt <- overlapPermTest(cpg, prom, genome="hg19", ntimes=100, per.chromosome=TRUE, count.once=TRUE)

```

```
#> $numOverlaps  
#> P-value: 0.0099009900990099  
#> Z-score: 45.6077  
#> Number of iterations: 100  
#> Alternative: greater  
#> Evaluation of the original region set: 719  
#> Evaluation function: numOverlaps  
#> Randomization function: randomizeRegions  
#>  
#> attr(,"class")  
#> [1] "permTestResultsList"  
  
summary(pt[[1]]$permuted)  
#>   Min. 1st Qu. Median   Mean 3rd Qu.   Max.  
#> 138.0 161.0 169.0 169.8 178.2 202.0
```

The resulting permutation *p*-value indicates a significant enrichment. Out of the 2859 CpG islands, 719 overlap with at least one promoter. In contrast, when repeatedly drawing random regions matching the CpG islands in size and chromosomal distribution, the mean number of overlapping regions across permutations was 117.7 ± 11.8 .

Note #1: The function `regioneR::permTest` allows to incorporate user-defined functions for randomizing regions and evaluating additional measures of overlap such as total genomic size in bp.

Note #2: The LOLA package implements a genomic region ORA, which assesses genomic region overlap based on the hypergeometric distribution using a library of pre-defined functional region sets.

Chapter 10

220: Workflow for multi-omics analysis with MultiAssayExperiment

10.1 Instructor names and contact information

- Marcel Ramos^{1,2} (marcel.ramos@roswellpark.org)
- Ludwig Geistlinger³
- Levi Waldron⁴

10.2 Workshop Description

This workshop demonstrates data management and analyses of multiple assays associated with a single set of biological specimens, using the `MultiAssayExperiment` data class and methods. It introduces the `RaggedExperiment` data class, which provides efficient and powerful operations for representation of copy number and mutation and variant data that are represented by different genomic ranges for each specimen.

10.2.1 Pre-requisites

- Basic knowledge of R syntax
- Familiarity with the `GRanges` and `SummarizedExperiment` classes
- Familiarity with 'omics data types including copy number and gene expression

10.2.2 Workshop Participation

Students will have a chance to build a `MultiAssayExperiment` object from scratch, and will also work with more complex objects provided by the `curatedTCGAData` package.

¹City University of New York, New York, NY, USA

²Roswell Park Comprehensive Cancer Center, Buffalo, NY

³City University of New York, New York, NY, USA

⁴City University of New York, New York, NY, USA

10.2.3 R/Bioconductor packages used

- MultiAssayExperiment
- GenomicRanges
- RaggedExperiment
- curatedTCGAData
- SummarizedExperiment
- TCGAutils
- UpSetR
- AnnotationFilter
- EnsDb.Hsapiens.v86
- survival
- survminer
- pheatmap

```
library(MultiAssayExperiment)
library(GenomicRanges)
library(RaggedExperiment)
library(curatedTCGAData)
library(GenomicDataCommons)
library(SummarizedExperiment)
library(SingleCellExperiment)
library(TCGAutils)
library(UpSetR)
library(mirbase.db)
library(AnnotationFilter)
library(EnsDb.Hsapiens.v86)
library(survival)
library(survminer)
library(pheatmap)
```

10.2.4 Time outline

1h 45m total

Activity	Time
Overview of key data classes	25m
Working with RaggedExperiment	20m
Building a MultiAssayExperiment from scratch	10m
TCGA multi-assay dataset	10m
Subsetting and reshaping multi-assay data	20m
Plotting, correlation, and other analyses	20m

10.3 Workshop goals and objectives

10.3.1 Learning goals

- identify appropriate data structures for different 'omics data types
- gain familiarity with GRangesList and RaggedExperiment

10.3.2 Learning objectives

- use curatedTCGAData to create custom TCGA MultiAssayExperiment objects
- create a MultiAssayExperiment for TCGA or other multi'omics data
- perform subsetting, reshaping, growing, and extraction of a MultiAssayExperiment
- link MultiAssayExperiment data with packages for differential expression, machine learning, and plotting

10.4 Overview of key data classes

This section summarizes three fundamental data classes for the representation of multi-omics experiments.

10.4.1 (Ranged)SummarizedExperiment

`SummarizedExperiment` is the most important Bioconductor class for matrix-like experimental data, including from RNA sequencing and microarray experiments. It can store multiple experimental data matrices of identical dimensions, with associated metadata on the rows/genes/transcripts/other measurements (`rowData`), column/sample phenotype or clinical data (`colData`), and the overall experiment (`metadata`). The derivative class `RangedSummarizedExperiment` associates a `GRanges` or `GRangesList` vector with the rows. These classes supersede the use of `ExpressionSet`. Note that many other classes for experimental data are actually derived from `SummarizedExperiment`; for example, the `SingleCellExperiment` class for single-cell RNA sequencing experiments extends `RangedSummarizedExperiment`, which in turn extends `SummarizedExperiment`:

```
library(SingleCellExperiment)
extends("SingleCellExperiment")
#> [1] "SingleCellExperiment"           "RangedSummarizedExperiment"
#> [3] "SummarizedExperiment"          "Vector"
#> [5] "Annotated"
```

Thus, although `SingleCellExperiment` provides additional methods over `RangedSummarizedExperiment`, it also inherits all the methods of `SummarizedExperiment` and `RangedSummarizedExperiment`, so everything you learn about `SummarizedExperiment` will be applicable to `SingleCellExperiment`.

10.4.2 RaggedExperiment

`RaggedExperiment` is a flexible data representation for segmented copy number, somatic mutations such as represented in .vcf files, and other ragged array schema for genomic location data. Like the `GRangesList` class from `GenomicRanges`, `RaggedExperiment` can be used to represent *differing* genomic ranges on each of a set of samples. In fact, `RaggedExperiment` contains a `GRangesList`:

```
showClass("RaggedExperiment")
#> Class "RaggedExperiment" [package "RaggedExperiment"]
#>
#> Slots:
#>
#> Name:      assays      rowidx      colidx      metadata
#> Class: GRangesList    integer     integer      list
#>
#> Extends: "Annotated"
```

However, `RaggedExperiment` provides a flexible set of `Assay` methods to support transformation of such data to matrix format.

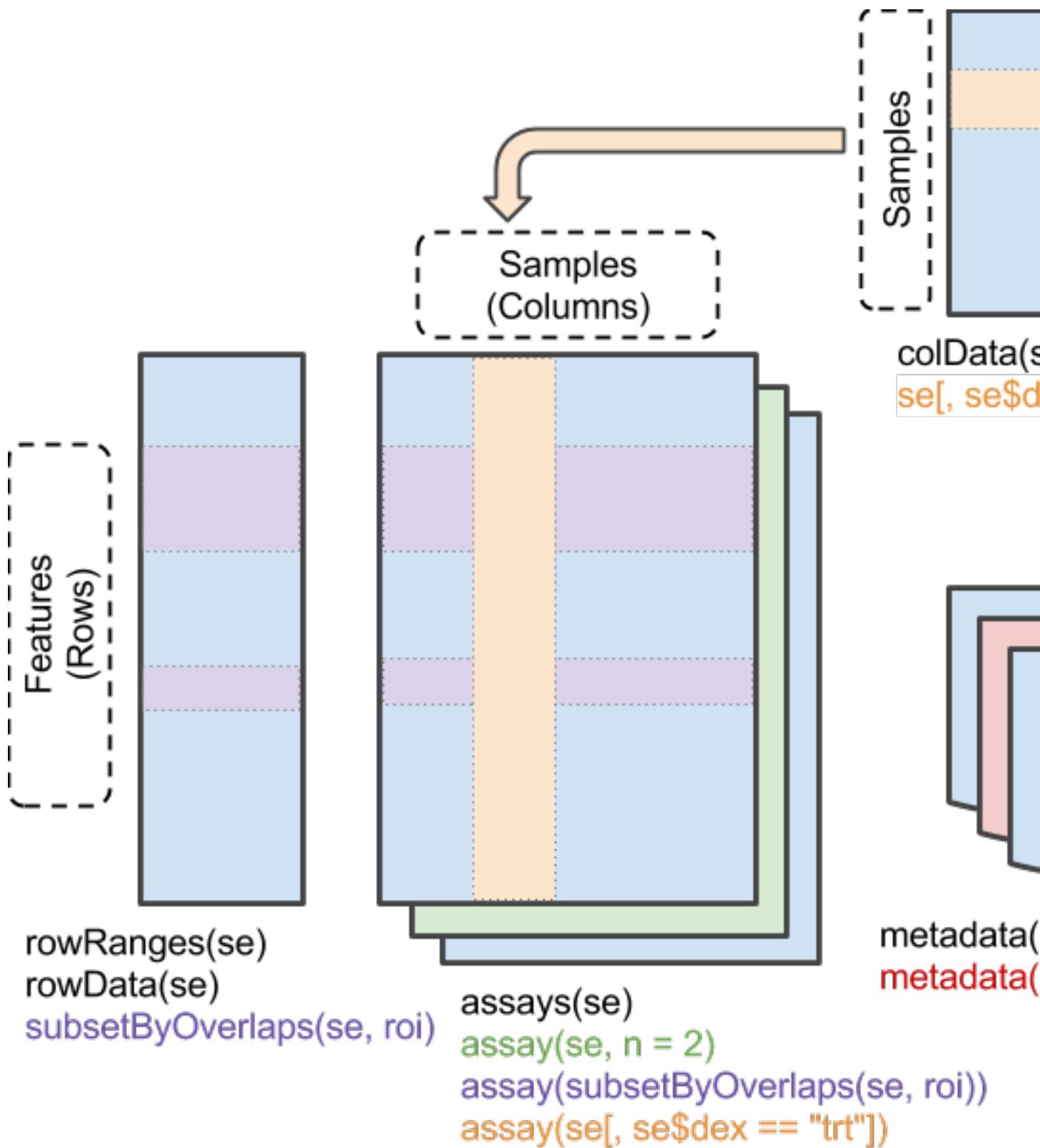


Figure 10.1: A matrix-like container where rows represent features of interest and columns represent samples. The objects contain one or more assays, each represented by a matrix-like object of numeric or other mode.

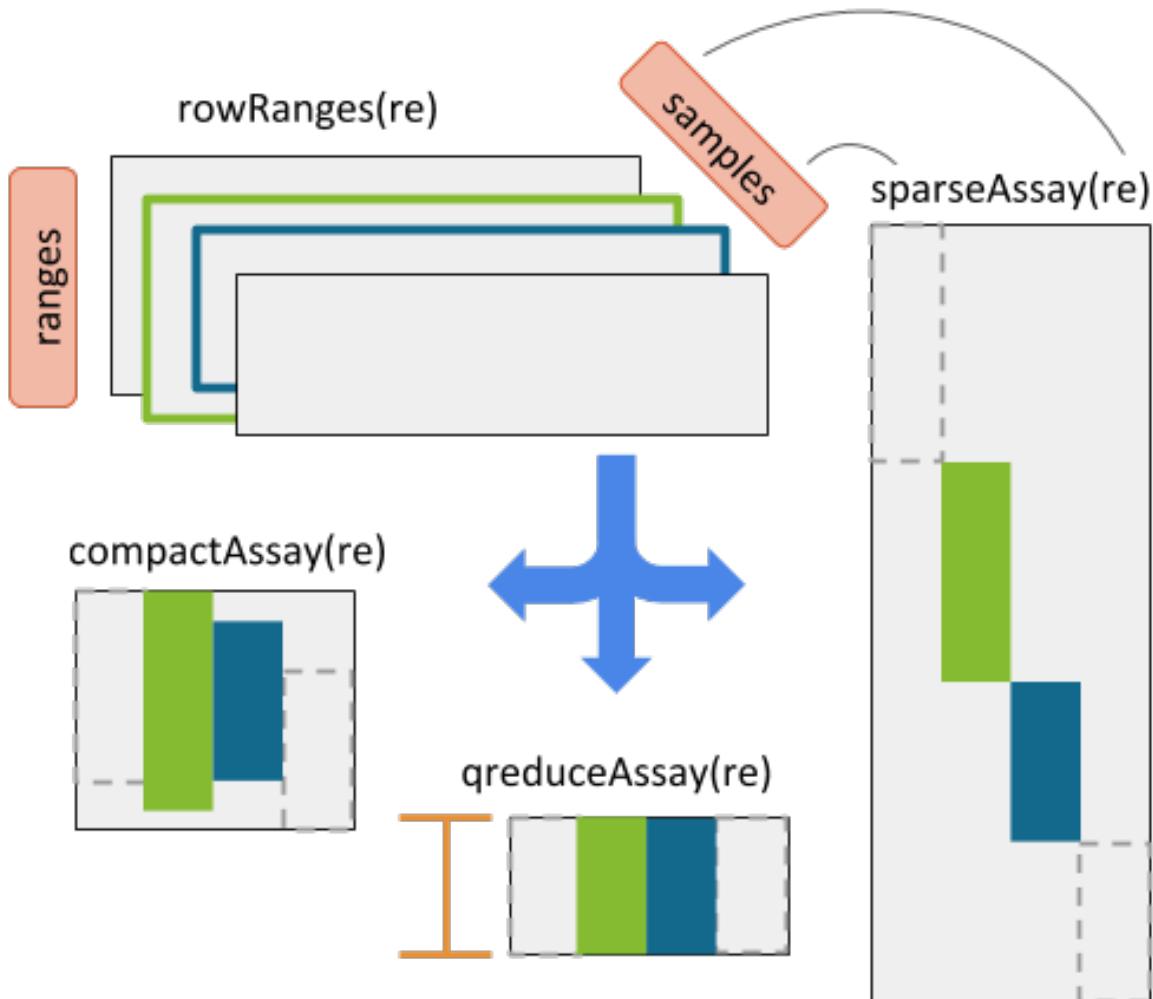


Figure 10.2: `RaggedExperiment` object schematic. Rows and columns represent genomic ranges and samples, respectively. Assay operations can be performed with (from left to right) `compactAssay`, `qreduceAssay`, and `sparseAssay`.

10.4.3 MultiAssayExperiment

`MultiAssayExperiment` is an integrative container for coordinating multi-omics experiment data on a set of biological specimens. As much as possible, its methods adopt the same vocabulary as `SummarizedExperiment`. A `MultiAssayExperiment` can contain any number of assays with different representations. Assays may be *ID-based*, where measurements are indexed identifiers of genes, microRNA, proteins, microbes, etc. Alternatively, assays may be *range-based*, where measurements correspond to genomic ranges that can be represented as `GRanges` objects, such as gene expression or copy number.

For ID-based assays, there is no requirement that the same IDs be present for different experiments. For range-based assays, there is also no requirement that the same ranges be present for different experiments; furthermore, it is possible for different samples within an experiment to be represented by different ranges. The following data classes have been tested to work as elements of a `MultiAssayExperiment`:

1. `matrix`: the most basic class for ID-based datasets, could be used for example for gene expression summarized per-gene, microRNA, metabolomics, or microbiome data.
2. `SummarizedExperiment` and derived methods: described above, could be used for miRNA, gene expression, proteomics, or any matrix-like data where measurements are represented by IDs.
3. `RangedSummarizedExperiment`: described above, could be used for gene expression, methylation, or other data types referring to genomic positions.
4. `ExpressionSet`: Another rich representation for ID-based datasets, supported only for legacy reasons
5. `RaggedExperiment`: described above, for non-rectangular (ragged) ranged-based datasets such as segmented copy number, where segmentation of copy number alterations occurs and different genomic locations in each sample.
6. `RangedVcfStack`: For VCF archives broken up by chromosome (see `VcfStack` class defined in the `GenomicFiles` package)
7. `DelayedMatrix`: An on-disk representation of matrix-like objects for large datasets. It reduces memory usage and optimizes performance with delayed operations. This class is part of the `DelayedArray` package.

Note that any data class extending these classes, and in fact any data class supporting row and column names and subsetting can be used as an element of a `MultiAssayExperiment`.

10.5 Working with RaggedExperiment

You can skip this section if you prefer to focus on the functionality of `MultiAssayExperiment`. In most use cases, you would likely convert a `RaggedExperiment` to matrix or `RangedSummarizedExperiment` using one of the `Assay` functions below, and either concatenate this rectangular object to the `MultiAssayExperiment` or use it to replace the `RaggedExperiment`.

10.5.1 Constructing a RaggedExperiment object

We start with a toy example of two `GRanges` objects, providing ranges on two chromosomes in two samples:

```
sample1 <- GRanges(
  c(A = "chr1:1-10:-", B = "chr1:8-14:+", C = "chr1:15-18:+"),
  score = 3:5, type=c("germline", "somatic", "germline"))
sample2 <- GRanges(
  c(D = "chr1:1-10:-", E = "chr1:11-18:+"),
  score = 11:12, type=c("germline", "somatic"))
```

Include column data `colData` to describe the samples:

```
colDat <- DataFrame(id=1:2, status = factor(c("control", "case")))
```

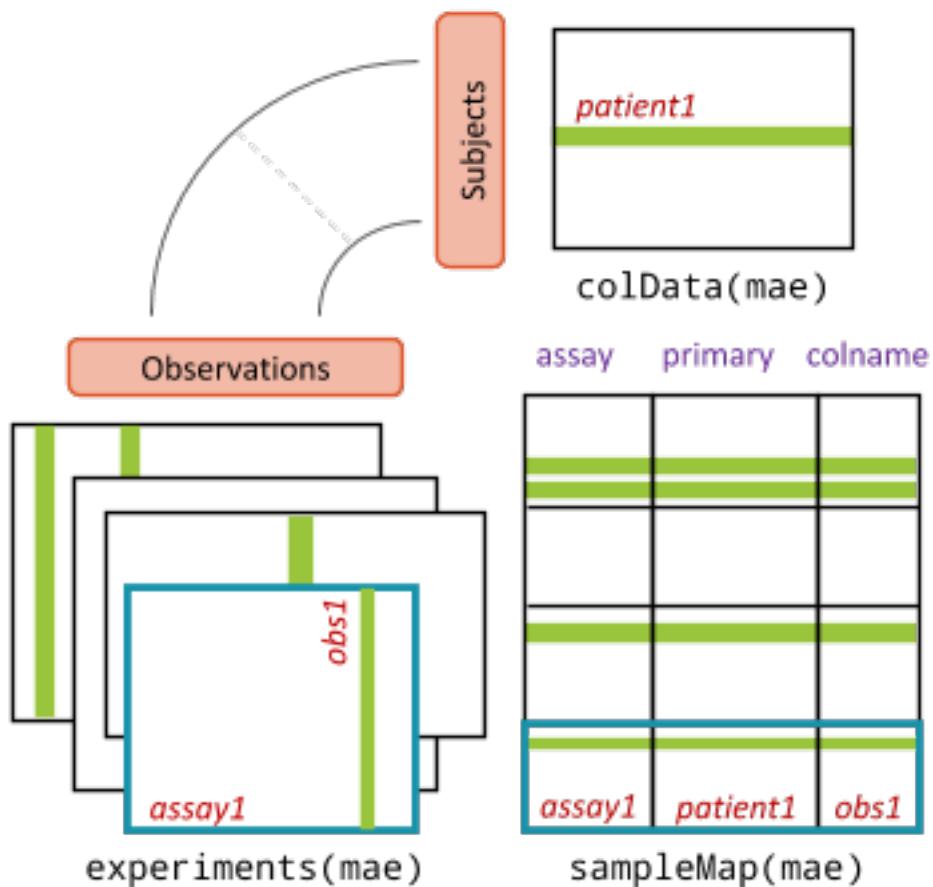


Figure 10.3: MultiAssayExperiment object schematic. `colData` provides data about the patients, cell lines, or other biological units, with one row per unit and one column per variable. The experiments are a list of assay datasets of arbitrary class. The `sampleMap` relates each column (observation) in `ExperimentList` to exactly one row (biological unit) in `colData`; however, one row of `colData` may map to zero, one, or more columns per assay, allowing for missing and replicate assays. `sampleMap` allows for per-assay sample naming conventions. Metadata can be used to store information in arbitrary format about the MultiAssayExperiment. Green stripes indicate a mapping of one subject to multiple observations across experiments.

The `RaggedExperiment` can be constructed from individual `Granges`:

```
(ragexp <- RaggedExperiment(
  sample1 = sample1,
  sample2 = sample2,
  colData = colDat))
#> class: RaggedExperiment
#> dim: 5 2
#> assays(2): score type
#> rownames(5): A B C D E
#> colnames(2): sample1 sample2
#> colData names(2): id status
```

Or from a `GRangesList`:

```
grl <- GRangesList(sample1=sample1, sample2=sample2)
ragexp2 <- RaggedExperiment(grl, colData = colDat)
identical(ragexp, ragexp2)
#> [1] TRUE
```

Note that the original ranges are represented as the `rowRanges` of the `RaggedExperiment`:

```
rowRanges(ragexp)
#> GRanges object with 5 ranges and 0 metadata columns:
#>   seqnames      ranges strand
#>   <Rle> <IRanges>  <Rle>
#>   A     chr1      1-10    -
#>   B     chr1      8-14    +
#>   C     chr1      15-18   +
#>   D     chr1      1-10    -
#>   E     chr1      11-18   +
#>   -----
#>   seqinfo: 1 sequence from an unspecified genome; no seqlengths
```

10.5.2 *Assay functions

A suite of `*Assay` operations allow users to resize the matrix-like representation of ranges to varying row dimensions (see `RaggedExperiment` Figure for a visual example).

The four main `Assay` functions for converting to matrix are:

- `sparseAssay`: leave ranges exactly as-is
- `compactAssay`: combine identical ranges
- `disjoinAssay`: disjoin ranges that overlap across samples
- `qreduceAssay`: find overlaps with provided “query” ranges

These each have a corresponding function for conversion to `RangedSummarizedExperiment`.

10.5.2.1 sparseAssay

The most straightforward matrix representation of a `RaggedExperiment` will return a matrix with the number of rows equal to the total number of ranges defined across all samples. *i.e.* the 5 rows of the `sparseAssay` result:

```
sparseAssay(ragexp)
#> sample1 sample2
```

```
#> A      3      NA
#> B      4      NA
#> C      5      NA
#> D      NA     11
#> E      NA     12
```

correspond to the ranges of the unlisted GRangesList:

```
unlist(gr1)
#> GRanges object with 5 ranges and 2 metadata columns:
#>           seqnames    ranges strand |   score      type
#>           <Rle>    <IRanges>  <Rle> | <integer> <character>
#> sample1.A    chr1    1-10     - /     3  germline
#> sample1.B    chr1    8-14     + /     4  somatic
#> sample1.C    chr1    15-18    + /     5  germline
#> sample2.D    chr1    1-10     - /    11  germline
#> sample2.E    chr1    11-18    + /    12  somatic
#> -----
#> seqinfo: 1 sequence from an unspecified genome; no seqlengths
```

The rownames of the `sparseAssay` result are equal to the names of the `GRanges` elements. The values in the matrix returned by `sparseAssay` correspond to the first columns of the `mcols` of each `GRangesList` element, in this case the “score” column.

Note, this is the default `assay()` method of `RaggedExperiment`:

```
assay(ragexp, "score")
#> sample1 sample2
#> A      3      NA
#> B      4      NA
#> C      5      NA
#> D      NA     11
#> E      NA     12
assay(ragexp, "type")
#> sample1   sample2
#> A "germline" NA
#> B "somatic"  NA
#> C "germline" NA
#> D NA        "germline"
#> E NA        "somatic"
```

10.5.2.2 compactAssay

The dimensions of the `compactAssay` result differ from that of the `sparseAssay` result only if there are identical ranges in different samples. Identical ranges are placed in the same row in the output. Ranges with any difference in start, end, or strand, will be kept on different rows. Non-disjoint ranges are **not** collapsed.

```
compactAssay(ragexp)
#>           sample1 sample2
#> chr1:8-14:+      4      NA
#> chr1:11-18:+     NA     12
#> chr1:15-18:+     5      NA
#> chr1:1-10:-     3      11
compactAssay(ragexp, "type")
#>           sample1 sample2
```

```
#> chr1:8-14:+ "somatic" NA
#> chr1:11-18:+ NA "somatic"
#> chr1:15-18:+ "germline" NA
#> chr1:1-10:- "germline" "germline"
```

Note that row names are constructed from the ranges, and the names of the `GRanges` vectors are lost, unlike in the `sparseAssay` result.

10.5.2.3 disjoinAssay

This function is similar to `compactAssay` except the rows are *disjoint*⁵ ranges. Elements of the matrix are summarized by applying the `simplifyDisjoin` functional argument to assay values of overlapping ranges.

```
disjoinAssay(ragexp, simplifyDisjoin = mean)
#>           sample1 sample2
#> chr1:8-10:+      4     NA
#> chr1:11-14:+      4     12
#> chr1:15-18:+      5     12
#> chr1:1-10:-      3     11
```

10.5.2.4 qreduceAssay

The `qreduceAssay` function is the most complicated but likely the most useful of the `RaggedExperiment Assay` functions. It requires you to provide a `query` argument that is a `GRanges` vector, and the rows of the resulting matrix correspond to the elements of this `GRanges`. The returned matrix will have dimensions `length(query)` by `ncol(x)`. Elements of the resulting matrix correspond to the overlap of the i th `query` range in the j th sample, summarized according to the `simplifyReduce` functional argument. This can be useful, for example, to calculate per-gene copy number or mutation status by providing the genomic ranges of every gene as the `query`.

The `simplifyReduce` argument in `qreduceAssay` allows the user to summarize overlapping regions with a custom method for the given “query” region of interest. We provide one for calculating a weighted average score per query range, where the weight is proportional to the overlap widths between overlapping ranges and a query range.

Note that there are three arguments to this function. See the documentation for additional details.

```
weightedmean <- function(scores, ranges, qranges)
{
  isects <- pintersect(ranges, qranges)
  sum(scores * width(isects)) / sum(width(isects))
}
```

The call to `qreduceAssay` calculates the overlaps between the ranges of each sample:

```
gr1
#> GRangesList object of length 2:
#> $sample1
#> GRanges object with 3 ranges and 2 metadata columns:
#>   seqnames      ranges strand |   score      type
#>   <Rle> <IRanges> <Rle> | <integer> <character>
#>   A      chr1      1-10    - /     3      germline
#>   B      chr1      8-14    + /     4      somatic
#>   C      chr1      15-18   + /     5      germline
```

⁵A *disjoint* set of ranges has no overlap between any ranges of the set.

```
#>
#> $sample2
#> GRanges object with 2 ranges and 2 metadata columns:
#>   seqnames ranges strand | score      type
#>   D       chr1    1-10      - /     11 germline
#>   E       chr1    11-18     + /     12 somatic
#>
#> -----
#> seqinfo: 1 sequence from an unspecified genome; no seqlengths
```

with the query ranges (an arbitrary set is defined here for demonstration): First create a demonstration “query” region of interest:

```
(query <- GRanges(c("chr1:1-14:-", "chr1:15-18:+")))
#> GRanges object with 2 ranges and 0 metadata columns:
#>   seqnames      ranges strand
#>   <Rle> <IRanges> <Rle>
#>   [1]   chr1      1-14      -
#>   [2]   chr1      15-18     +
#>
#> -----
#> seqinfo: 1 sequence from an unspecified genome; no seqlengths
```

using the `simplifyReduce` function to resolve overlapping ranges and return a matrix with rows corresponding to the query:

```
qreduceAssay(ragexp, query, simplifyReduce = weightedmean)
#>           sample1 sample2
#> chr1:1-14:-      3      11
#> chr1:15-18:+     5      12
```

10.5.3 Conversion to RangedSummarizedExperiment

These methods all have corresponding methods to return a `RangedSummarizedExperiment` and preserve the `colData`:

```
sparseSummarizedExperiment(ragexp)
compactSummarizedExperiment(ragexp)
disjoinSummarizedExperiment(ragexp, simplify = mean)
qreduceSummarizedExperiment(ragexp, query=query, simplify=weightedmean)
```

10.6 Working with MultiAssayExperiment

10.7 API cheat sheet

10.7.1 The MultiAssayExperiment miniACC demo object

Get started by trying out `MultiAssayExperiment` using a subset of the TCGA adrenocortical carcinoma (ACC) dataset provided with the package. This dataset provides five assays on 92 patients, although all five assays were not performed for every patient:

1. `RNASeq2GeneNorm`: gene mRNA abundance by RNA-seq
2. `gistic`: GISTIC genomic copy number by gene

Table 1. Summary of the MultiAssayExperiment API (Ramos et al. Can. Res. 2017; DOI: 10.1158/0008-5472.CAN-17-0344)

Category and Function	Description	Returned class
Constructors		
MultiAssayExperiment	Create a MultiAssayExperiment object	MultiAssayExperiment
ExperimentList	Create an ExperimentList from a List or list	ExperimentList
Accessors		
colData	Get or set data that describe patients / biological units	DataFrame
experiments	Get or set the list of experimental data objects as original classes	ExperimentList
assays	Get the list of experimental data numeric matrices	SimpleList
assay	Get the first experimental data numeric matrix	matrix, matrix-like
sampleMap	Get or set the map relating observations to subjects	DataFrame
metadata	Get or set additional data descriptions	list
rownames	Get row names for all experiments	CharacterList
colnames	Get column names for all experiments	CharacterList
Subsetting		
mae[i, j, k]	Get rows, columns, and/or experiments	MultiAssayExperiment
mae[i, ,]	i: GRanges, character, integer, logical, List, list	MultiAssayExperiment
mae[, j,]	j: character, integer, logical, List, list	MultiAssayExperiment
mae[, , k]	k: character, integer, logical	MultiAssayExperiment
mae[[n]]	Get or set object of arbitrary class from experiments	(varies)
mae[[n]]	n: character, integer, logical	
mae\$column	Get or set colData column	vector (varies)
Management		
complete.cases	Identify subjects with complete data in all experiments	vector (logical)
duplicated replicated*	Identify subjects with replicate observations per experiment	list of LogicalLists
anyReplicated	Displays whether there are any replicate observations in each assay	vector (logical)
mergeReplicates	Merge replicate observations within each experiment, using function	MultiAssayExperiment
intersectRows	Return features that are present for all experiments	MultiAssayExperiment
intersectColumns	Return subjects with data available for all experiments	MultiAssayExperiment
prepMultiAssay	Troubleshoot common problems when constructing main class	list
Reshaping		
longFormat	Return a long and tidy DataFrame with optional colData columns	DataFrame
wideFormat	Create a wide DataFrame, 1 row per subject	DataFrame
Combining		
c	Concatenate an experiment to an existing MultiAssayExperiment	MultiAssayExperiment

Note. *assay* refers to a procedure for measuring the biochemical or immunological activity of a sample, e.g. RNA-seq, segmented copy number, and somatic mutation calls would be considered three different assays. *experiment* refers to the application of an assay to a set of samples. In general it is assumed that each experiment uses a different assay type, although an assay type may of course be repeated in different experiments. *mae* refers to a MultiAssayExperiment object. *subject* refers to patient, cell line, or other biological unit. *observation* refers to results of an assay, e.g. gene expression, somatic mutations, etc. *features* refer to measurements returned by the assays, labeled by row names or genomic ranges.

* "duplicated" was deprecated in Bioconductor 3.7 and replaced by "replicated"

Figure 10.4: The MultiAssayExperiment API for construction, access, subsetting, management, and reshaping to formats for application of R/Bioconductor graphics and analysis packages.

- 3. **RPPAArray**: protein abundance by Reverse Phase Protein Array
- 4. **Mutations**: non-silent somatic mutations by gene
- 5. **miRNASeqGene**: microRNA abundance by microRNA-seq.

```
data(miniACC)
miniACC
#> A MultiAssayExperiment object of 5 listed
#> experiments with user-defined names and respective classes.
#> Containing an ExperimentList class object of length 5:
#> [1] RNASeq2GeneNorm: SummarizedExperiment with 198 rows and 79 columns
#> [2] gistic: SummarizedExperiment with 198 rows and 90 columns
#> [3] RPPAArray: SummarizedExperiment with 33 rows and 46 columns
#> [4] Mutations: matrix with 97 rows and 90 columns
#> [5] miRNASeqGene: SummarizedExperiment with 471 rows and 80 columns
#> Features:
#> experiments() - obtain the ExperimentList instance
#> colData() - the primary/phenotype DataFrame
#> sampleMap() - the sample availability DataFrame
#> `$, `[, `[[` - extract colData columns, subset, or experiment
#> *Format() - convert into a long or wide DataFrame
#> assays() - convert ExperimentList to a SimpleList of matrices
```

10.7.2 colData - information biological units

This slot is a **DataFrame** describing the characteristics of biological units, for example clinical data for patients. In the prepared datasets from [The Cancer Genome Atlas][], each row is one patient and each column is a clinical, pathological, subtype, or other variable. The **\$** function provides a shortcut for accessing or setting **colData** columns.

```
colData(miniACC)[1:4, 1:4]
#> DataFrame with 4 rows and 4 columns
#>
#>             patientID years_to_birth vital_status days_to_death
#>             <character>     <integer>      <integer>      <integer>
#> TCGA-OR-A5J1  TCGA-OR-A5J1        58           1         1355
#> TCGA-OR-A5J2  TCGA-OR-A5J2        44           1         1677
#> TCGA-OR-A5J3  TCGA-OR-A5J3        23           0           NA
#> TCGA-OR-A5J4  TCGA-OR-A5J4        23           1          423
table(miniACC$race)
#>
#>             asian black or african american
#>             2                 1
#>             white
#>             78
```

Key points about the colData:

- Each row maps to zero or more observations in each experiment in the **ExperimentList**, below.
- One row per biological unit
 - **MultiAssayExperiment** supports both missing observations and replicate observations, ie one row of **colData** can map to 0, 1, or more columns of any of the experimental data matrices.
 - therefore you could treat replicate observations as one or multiple rows of **colData**, and this will result in different behaviors of functions you will learn later like subsetting, **duplicated()**, and **wideFormat()**.
 - multiple time points, or distinct biological replicates, should probably be separate rows of the **colData**.

10.7.3 ExperimentList - experiment data

A base `list` or `ExperimentList` object containing the experimental datasets for the set of samples collected. This gets converted into a class `ExperimentList` during construction.

```
experiments(miniACC)
#> ExperimentList class object of length 5:
#> [1] RNASeq2GeneNorm: SummarizedExperiment with 198 rows and 79 columns
#> [2] gistic: SummarizedExperiment with 198 rows and 90 columns
#> [3] RPPAArray: SummarizedExperiment with 33 rows and 46 columns
#> [4] Mutations: matrix with 97 rows and 90 columns
#> [5] miRNASeqGene: SummarizedExperiment with 471 rows and 80 columns
```

Key points:

- One matrix-like dataset per list element (although they do not even need to be matrix-like, see for example the `RaggedExperiment` package)
- One matrix column per assayed specimen. Each matrix column must correspond to exactly one row of `colData`: in other words, you must know which patient or cell line the observation came from. However, multiple columns can come from the same patient, or there can be no data for that patient.
- Matrix rows correspond to variables, e.g. genes or genomic ranges
- `ExperimentList` elements can be genomic range-based (e.g. `SummarizedExperiment`::`RangedSummarizedExperiment-class` or `RaggedExperiment`::`RaggedExperiment-class`) or ID-based data (e.g. `SummarizedExperiment`::`SummarizedExperiment`, `Biobase`::`eSet-class`, `base`::`matrix-class`, `DelayedArray`::`DelayedArray-class`, and derived classes)
- Any data class can be included in the `ExperimentList`, as long as it supports: single-bracket subsetting (`[]`), `dimnames`, and `dim`. Most data classes defined in Bioconductor meet these requirements.

10.7.4 sampleMap - relationship graph

`sampleMap` is a graph representation of the relationship between biological units and experimental results. In simple cases where the column names of `ExperimentList` data matrices match the row names of `colData`, the user won't need to specify or think about a sample map, it can be created automatically by the `MultiAssayExperiment` constructor. `sampleMap` is a simple three-column `DataFrame`:

1. `assay` column: the name of the assay, and found in the names of `ExperimentList` list names
 2. `primary` column: identifiers of patients or biological units, and found in the row names of `colData`
 3. `colname` column: identifiers of assay results, and found in the column names of `ExperimentList`
- Helper functions are available for creating a map from a list. See `?listToMap`

```
sampleMap(miniACC)
#> DataFrame with 385 rows and 3 columns
#>       assay      primary           colname
#>       <factor>  <character>        <character>
#> 1  RNASeq2GeneNorm TCGA-OR-A5J1 TCGA-OR-A5J1-01A-11R-A29S-07
#> 2  RNASeq2GeneNorm TCGA-OR-A5J2 TCGA-OR-A5J2-01A-11R-A29S-07
#> 3  RNASeq2GeneNorm TCGA-OR-A5J3 TCGA-OR-A5J3-01A-11R-A29S-07
#> 4  RNASeq2GeneNorm TCGA-OR-A5J5 TCGA-OR-A5J5-01A-11R-A29S-07
#> 5  RNASeq2GeneNorm TCGA-OR-A5J6 TCGA-OR-A5J6-01A-31R-A29S-07
#> ...
#> 381    ...          ...
#> 382    ...          ...
#> 383    ...          ...
#> 384    ...          ...
#> 385    ...          ...
```

Key points:

- relates experimental observations (`colnames`) to `colData`
- permits experiment-specific sample naming, missing, and replicate observations

[back to top](#)

10.7.5 metadata

Metadata can be used to keep additional information about patients, assays performed on individuals or on the entire cohort, or features such as genes, proteins, and genomic ranges. There are many options available for storing metadata. First, `MultiAssayExperiment` has its own metadata for describing the entire experiment:

```
metadata(miniACC)
#> $title
#> [1] "Comprehensive Pan-Genomic Characterization of Adrenocortical Carcinoma"
#>
#> $PMID
#> [1] "27165744"
#>
#> $sourceURL
#> [1] "http://s3.amazonaws.com/multiassayexperiments/accMAEO.rds"
#>
#> $RPPAfeatureDataURL
#> [1] "http://genomeportal.stanford.edu/pan-tcga/show_target_selection_file?filename=Allprotein.txt"
#>
#> $colDataExtrasURL
#> [1] "http://www.cell.com/cms/attachment/2062093088/2063584534/mmc3.xlsx"
```

Additionally, the `DataFrame` class used by `sampleMap` and `colData`, as well as the `ExperimentList` class, similarly support metadata. Finally, many experimental data objects that can be used in the `ExperimentList` support metadata. These provide flexible options to users and to developers of derived classes.

10.8 MultiAssayExperiment Subsetting

10.8.1 Single bracket [

In pseudo code below, the subsetting operations work on the rows of the following indices: 1. i experimental data rows 2. j the primary names or the column names (entered as a `list` or `List`) 3. k assay

```
multiassayexperiment[i = rownames, j = primary or colnames, k = assay]
```

Subsetting operations always return another `MultiAssayExperiment`. For example, the following will return any rows named “MAPK14” or “IGFBP2”, and remove any assays where no rows match:

```
miniACC[c("MAPK14", "IGFBP2"), , ]
```

The following will keep only patients of pathological stage iv, and all their associated assays:

```
miniACC[, miniACC$pathologic_stage == "stage iv", ]
#> harmonizing input:
#> removing 311 sampleMap rows with 'colname' not in colnames of experiments
#> removing 74 colData rownames not in sampleMap 'primary'
```

And the following will keep only the RNA-seq dataset, and only patients for which this assay is available:

```
miniACC[, , "RNASeq2GeneNorm"]
#> harmonizing input:
#>   removing 13 colData rownames not in sampleMap 'primary'
```

10.8.2 Subsetting by genomic ranges

If any ExperimentList objects have features represented by genomic ranges (e.g. `RangedSummarizedExperiment`, `RaggedExperiment`), then a `GRanges` object in the first subsetting position will subset these objects as in `GenomicRanges::findOverlaps()`. Any non-ranged `ExperimentList` element will be subset to zero rows.

10.8.3 Double bracket [[

The “double bracket” method (`[[]]`) is a convenience function for extracting a single element of the `MultiAssayExperiment` `ExperimentList`. It avoids the use of `experiments(mae)[[1L]]`. For example, both of the following extract the `ExpressionSet` object containing RNA-seq data:

```
miniACC[[1L]]  #or equivalently, miniACC[["RNASeq2GeneNorm"]]
#> class: SummarizedExperiment
#> dim: 198 79
#> metadata(3): experimentData annotation protocolData
#> assays(1): exprs
#> rownames(198): DIRAS3 MAPK14 ... SQSTM1 KCNJ13
#> rowData names(0):
#> colnames(79): TCGA-OR-A5J1-01A-11R-A29S-07
#>   TCGA-OR-A5J2-01A-11R-A29S-07 ... TCGA-PK-A5HA-01A-11R-A29S-07
#>   TCGA-PK-A5HB-01A-11R-A29S-07
#> colData names(0):
```

10.9 Complete cases

`complete.cases()` shows which patients have complete data for all assays:

```
summary(complete.cases(miniACC))
#>   Mode      FALSE      TRUE
#> logical        49        43
```

The above logical vector could be used for patient subsetting. More simply, `intersectColumns()` will select complete cases and rearrange each `ExperimentList` element so its columns correspond exactly to rows of `colData` in the same order:

```
accmatched = intersectColumns(miniACC)
#> harmonizing input:
#>   removing 170 sampleMap rows with 'colname' not in colnames of experiments
#>   removing 49 colData rownames not in sampleMap 'primary'
```

Note, the column names of the assays in `accmatched` are not the same because of assay-specific identifiers, but they have been automatically re-arranged to correspond to the same patients. In these TCGA assays, the first three - delimited positions correspond to patient, ie the first patient is `TCGA-OR-A5J2`:

```
colnames(accmatched)
#> CharacterList of length 5
#> [[["RNASeq2GeneNorm"]]] TCGA-OR-A5J2-01A-11R-A29S-07 ...
```

```
#> [[{"gistic"}]] TCGA-OR-A5J2-01A-11D-A29H-01 ...
#> [[{"RPPAArray"}]] TCGA-OR-A5J2-01A-21-A39K-20 ...
#> [[{"Mutations"}]] TCGA-OR-A5J2-01A-11D-A29I-10 ...
#> [[{"miRNASeqGene"}]] TCGA-OR-A5J2-01A-11R-A29W-13 ...
```

10.10 Row names that are common across assays

`intersectRows()` keeps only rows that are common to each assay, and aligns them in identical order. For example, to keep only genes where data are available for RNA-seq, GISTIC copy number, and somatic mutations:

```
accmatched2 <- intersectRows(miniACC[, , c("RNASeq2GeneNorm", "gistic", "Mutations")])
rownames(accmatched2)
#> CharacterList of length 3
#> [[{"RNASeq2GeneNorm"}]] DIRAS3 G6PD KDR ERBB3 ... RET CDKN2A MACC1 CHGA
#> [[{"gistic"}]] DIRAS3 G6PD KDR ERBB3 AKT1S1 ... PREX1 RET CDKN2A MACC1 CHGA
#> [[{"Mutations"}]] DIRAS3 G6PD KDR ERBB3 AKT1S1 ... RET CDKN2A MACC1 CHGA
```

[back to top](#)

10.11 Extraction

10.11.1 assay and assays

The `assay` and `assays` methods follow `SummarizedExperiment` convention. The `assay` (singular) method will extract the first element of the `ExperimentList` and will return a `matrix`.

```
class(assay(miniACC))
#> [1] "matrix"
```

The `assays` (plural) method will return a `SimpleList` of the data with each element being a `matrix`.

```
assays(miniACC)
#> List of length 5
#> names(5): RNASeq2GeneNorm gistic RPPAArray Mutations miRNASeqGene
```

Key point:

- Whereas the `[[` returned an assay as its original class, `assay()` and `assays()` convert the assay data to matrix form.

[back to top](#)

10.12 Summary of slots and accessors

Slot in the `MultiAssayExperiment` can be accessed or set using their accessor functions:

Slot	Accessor
ExperimentList	<code>experiments()</code>
colData	<code>colData()</code> and <code>\$ *</code>
sampleMap	<code>sampleMap()</code>

Slot	Accessor
metadata	metadata()

—*— The \$ operator on a MultiAssayExperiment returns a single column of the colData.

10.13 Transformation / reshaping

The longFormat or wideFormat functions will “reshape” and combine experiments with each other and with colData into one DataFrame. These functions provide compatibility with most of the common R/Bioconductor functions for regression, machine learning, and visualization.

10.13.1 longFormat

In *long* format a single column provides all assay results, with additional optional colData columns whose values are repeated as necessary. Here *assay* is the name of the ExperimentList element, *primary* is the patient identifier (rowname of colData), *rowname* is the assay rowname (in this case genes), *colname* is the assay-specific identifier (column name), *value* is the numeric measurement (gene expression, copy number, presence of a non-silent mutation, etc), and following these are the *vital_status* and *days_to_death* colData columns that have been added:

```
longFormat(miniACC[c("TP53", "CTNNB1"), , ],
            colDataCols = c("vital_status", "days_to_death"))
#> DataFrame with 518 rows and 7 columns
#>           assay      primary     rowname          colname
#> 1       <character> <character> <character> <character>
#> 1   RNASeq2GeneNorm TCGA-OR-A5J1        TP53 TCGA-OR-A5J1-01A-11R-A29S-07
#> 2   RNASeq2GeneNorm TCGA-OR-A5J1        CTNNB1 TCGA-OR-A5J1-01A-11R-A29S-07
#> 3   RNASeq2GeneNorm TCGA-OR-A5J2        TP53 TCGA-OR-A5J2-01A-11R-A29S-07
#> 4   RNASeq2GeneNorm TCGA-OR-A5J2        CTNNB1 TCGA-OR-A5J2-01A-11R-A29S-07
#> 5   RNASeq2GeneNorm TCGA-OR-A5J3        TP53 TCGA-OR-A5J3-01A-11R-A29S-07
#> ...
#> 514    ...          ...          ...          ...
#> 514    Mutations TCGA-PK-A5HA        CTNNB1 TCGA-PK-A5HA-01A-11D-A29I-10
#> 515    Mutations TCGA-PK-A5HB        TP53 TCGA-PK-A5HB-01A-11D-A29I-10
#> 516    Mutations TCGA-PK-A5HB        CTNNB1 TCGA-PK-A5HB-01A-11D-A29I-10
#> 517    Mutations TCGA-PK-A5HC        TP53 TCGA-PK-A5HC-01A-11D-A30A-10
#> 518    Mutations TCGA-PK-A5HC        CTNNB1 TCGA-PK-A5HC-01A-11D-A30A-10
#>           value vital_status days_to_death
#> 1      563.4006      1        1355
#> 2      5634.4669      1        1355
#> 3      165.4811      1        1677
#> 4      62658.3913      1        1677
#> 5      956.3028      0        NA
#> ...
#> 514    ...          ...          ...
#> 514    0            0            NA
#> 515    0            0            NA
#> 516    0            0            NA
#> 517    0            0            NA
#> 518    0            0            NA
```

10.13.2 wideFormat

In *wide* format, each feature from each assay goes in a separate column, with one row per primary identifier (patient). Here, each variable becomes a new column:

```
wideFormat(miniACC[c("TP53", "CTNNB1"), , ],
            colDataCols = c("vital_status", "days_to_death"))
#> DataFrame with 92 rows and 9 columns
#>
#>   primary vital_status days_to_death RNASEq2GeneNorm_CTNNB1
#>   <character>    <integer>     <integer>      <numeric>
#> 1 TCGA-OR-A5J1        1       1355      5634.4669
#> 2 TCGA-OR-A5J2        1       1677      62658.3913
#> 3 TCGA-OR-A5J3        0        NA      6337.4256
#> 4 TCGA-OR-A5J4        1       423        NA
#> 5 TCGA-OR-A5J5        1       365      5979.055
#> ...
#> 88 TCGA-PK-A5H9       0        NA      5258.9863
#> 89 TCGA-PK-A5HA       0        NA      8120.1654
#> 90 TCGA-PK-A5HB       0        NA      5257.8148
#> 91 TCGA-PK-A5HC       0        NA        NA
#> 92 TCGA-P6-A50G       1       383      6390.0997
#>   RNASEq2GeneNorm_TP53 gistict_CTNNB1 gistict_TP53 Mutations_CTNNB1
#>   <numeric>    <numeric>    <numeric>      <numeric>
#> 1      563.4006        0        0        0
#> 2      165.4811        1        0        1
#> 3      956.3028        0        0        0
#> 4        NA            0        1        0
#> 5     1169.6359        0        0        0
#> ...
#> 88     890.8663        0        0        0
#> 89     683.5722        0       -1        0
#> 90     237.3697       -1       -1        0
#> 91        NA            1        1        0
#> 92     815.3446        1       -1       NA
#>   Mutations_TP53
#>   <numeric>
#> 1        0
#> 2        1
#> 3        0
#> 4        0
#> 5        0
#> ...
#> 88        0
#> 89        0
#> 90        0
#> 91        0
#> 92       NA
```

10.14 MultiAssayExperiment class construction and concatenation

10.14.1 MultiAssayExperiment constructor function

The `MultiAssayExperiment` constructor function can take three arguments:

1. `experiments` - An `ExperimentList` or list of data
2. `colData` - A `DataFrame` describing the patients (or cell lines, or other biological units)
3. `sampleMap` - A `DataFrame` of assay, primary, and `colname` identifiers

The `miniACC` object can be reconstructed as follows:

```
MultiAssayExperiment(experiments=experiments(miniACC),
                      colData=colData(miniACC),
                      sampleMap=sampleMap(miniACC),
                      metadata=metadata(miniACC))

#> A MultiAssayExperiment object of 5 listed
#> experiments with user-defined names and respective classes.
#> Containing an ExperimentList class object of length 5:
#> [1] RNASeq2GeneNorm: SummarizedExperiment with 198 rows and 79 columns
#> [2] gistic: SummarizedExperiment with 198 rows and 90 columns
#> [3] RPPAArray: SummarizedExperiment with 33 rows and 46 columns
#> [4] Mutations: matrix with 97 rows and 90 columns
#> [5] miRNASeqGene: SummarizedExperiment with 471 rows and 80 columns
#> Features:
#> experiments() - obtain the ExperimentList instance
#> colData() - the primary/phenotype DataFrame
#> sampleMap() - the sample availability DataFrame
#> `$, `[, `[[` - extract colData columns, subset, or experiment
#> *Format() - convert into a long or wide DataFrame
#> assays() - convert ExperimentList to a SimpleList of matrices
```

10.14.2 prepMultiAssay - Constructor function helper

The `prepMultiAssay` function allows the user to diagnose typical problems when creating a `MultiAssayExperiment` object. See `?prepMultiAssay` for more details.

10.14.3 c - concatenate to MultiAssayExperiment

The `c` function allows the user to concatenate an additional experiment to an existing `MultiAssayExperiment`. The optional `sampleMap` argument allows concatenating an assay whose column names do not match the row names of `colData`. For convenience, the `mapFrom` argument allows the user to map from a particular experiment **provided** that the `order` of the colnames is in the **same**. A warning will be issued to make the user aware of this assumption. For example, to concatenate a matrix of log2-transformed RNA-seq results:

```
miniACC2 <- c(miniACC, log2rnaseq = log2(assays(miniACC)$RNASeq2GeneNorm), mapFrom=1L)
#> Warning in .local(x, ...): Assuming column order in the data provided
#> matches the order in 'mapFrom' experiment(s) colnames
experiments(miniACC2)
#> ExperimentList class object of length 6:
#> [1] RNASeq2GeneNorm: SummarizedExperiment with 198 rows and 79 columns
#> [2] gistic: SummarizedExperiment with 198 rows and 90 columns
```

```
#> [3] RPPAArray: SummarizedExperiment with 33 rows and 46 columns
#> [4] Mutations: matrix with 97 rows and 90 columns
#> [5] miRNASeqGene: SummarizedExperiment with 471 rows and 80 columns
#> [6] log2rnaseq: matrix with 198 rows and 79 columns
```

[back to top](#)

10.14.4 Building a MultiAssayExperiment from scratch

To start from scratch building your own MultiAssayExperiment, see the package Coordinating Analysis of Multi-Assay Experiments vignette. The package cheat sheet is also helpful.

If anything is unclear, please ask a question at <https://support.bioconductor.org/> or create an issue on the MultiAssayExperiment issue tracker.

10.15 The Cancer Genome Atlas (TCGA) as MultiAssayExperiment objects

Most unrestricted TCGA data are available as MultiAssayExperiment objects from the curatedTCGAData package. This represents a lot of harmonization!

```
library(curatedTCGAData)
curatedTCGAData("ACC")
#> ACC_CNASNP
#> "ACC_CNASNP-20160128.rda"
#> ACC_CNVSNP
#> "ACC_CNVSNP-20160128.rda"
#> ACC_GISTIC_AllByGene
#> "ACC_GISTIC_AllByGene-20160128.rda"
#> ACC_GISTIC_ThresholdedByGene
#> "ACC_GISTIC_ThresholdedByGene-20160128.rda"
#> ACC_Methylation
#> "ACC_Methylation-20160128.rda"
#> ACC_miRNASeqGene
#> "ACC_miRNASeqGene-20160128.rda"
#> ACC_Mutation
#> "ACC_Mutation-20160128.rda"
#> ACC_RNASeq2GeneNorm
#> "ACC_RNASeq2GeneNorm-20160128.rda"
#> ACC_RPPAArray
#> "ACC_RPPAArray-20160128.rda"
suppressMessages({
  acc <- curatedTCGAData("ACC",
    assays = c("miRNASeqGene", "RPPAArray", "Mutation", "RNASeq2GeneNorm", "CNVSNP"),
    dry.run = FALSE)
})
acc
#> A MultiAssayExperiment object of 5 listed
#> experiments with user-defined names and respective classes.
#> Containing an ExperimentList class object of length 5:
#> [1] ACC_CNVSNP-20160128: RaggedExperiment with 21052 rows and 180 columns
#> [2] ACC_miRNASeqGene-20160128: SummarizedExperiment with 1046 rows and 80 columns
```

```
#> [3] ACC_Mutation-20160128: RaggedExperiment with 20166 rows and 90 columns
#> [4] ACC_RNASeq2GeneNorm-20160128: SummarizedExperiment with 20501 rows and 79 columns
#> [5] ACC_RPPAArray-20160128: SummarizedExperiment with 192 rows and 46 columns
#> Features:
#> experiments() - obtain the ExperimentList instance
#> colData() - the primary/phenotype DataFrame
#> sampleMap() - the sample availability DataFrame
#> `$, `[, `[[` - extract colData columns, subset, or experiment
#> *Format() - convert into a long or wide DataFrame
#> assays() - convert ExperimentList to a SimpleList of matrices
```

These objects contain most unrestricted TCGA assay and clinical / pathological data, as well as material curated from the supplements of published TCGA primary papers at the end of the colData columns:

```
dim(colData(acc))
#> [1] 92 822
tail(colnames(colData(acc)), 10)
#> [1] "MethyLevel"           "miRNA.cluster"      "SCNA.cluster"
#> [4] "protein.cluster"     "CDC"              "OncoSign"
#> [7] "purity"               "ploidy"            "genome_doublings"
#> [10] "ADS"
```

The `TCGAutils` package provides additional helper functions, see below.

10.16 Utilities for TCGA

Aside from the available reshaping functions already included in the `MultiAssayExperiment` package, the `TCGAutils` package provides additional helper functions for working with TCGA data.

10.16.1 “Simplification” of `curatedTCGAData` objects

A number of helper functions are available for managing datasets from `curatedTCGAData`. These include:

- Conversions of `SummarizedExperiment` to `RangedSummarizedExperiment` based on `TxDb.Hsapiens.UCSC.hg19.knownGene` for:
 - `mirToRanges`: microRNA
 - `symbolsToRanges`: gene symbols
- `qreduceTCGA`: convert `RaggedExperiment` objects to `RangedSummarizedExperiment` with one row per gene symbol, for:
 - segmented copy number datasets (“CNVSNP” and “CNASNP”)
 - somatic mutation datasets (“Mutation”), with a value of 1 for any non-silent mutation and a value of 0 for no mutation or silent mutation

The `simplifyTCGA` function combines all of the above operations to create a more manageable `MultiAssayExperiment` object and using `RangedSummarizedExperiment` assays where possible.

```
(simpa <- TCGAutils::simplifyTCGA(acc))
#>
#> 'select()' returned 1:1 mapping between keys and columns
#> 'select()' returned 1:many mapping between keys and columns
#> 'select()' returned 1:1 mapping between keys and columns
#> A MultiAssayExperiment object of 7 listed
#> experiments with user-defined names and respective classes.
```

```
#> Containing an ExperimentList class object of length 7:
#> [1] ACC_RPPAArray-20160128: SummarizedExperiment with 192 rows and 46 columns
#> [2] ACC_Mutation-20160128_simplified: RangedSummarizedExperiment with 22945 rows and 90 columns
#> [3] ACC_CNVSNP-20160128_simplified: RangedSummarizedExperiment with 22945 rows and 180 columns
#> [4] ACC_miRNASeqGene-20160128_ranged: RangedSummarizedExperiment with 1002 rows and 80 columns
#> [5] ACC_miRNASeqGene-20160128_unranged: SummarizedExperiment with 44 rows and 80 columns
#> [6] ACC_RNASeq2GeneNorm-20160128_ranged: RangedSummarizedExperiment with 17594 rows and 79 columns
#> [7] ACC_RNASeq2GeneNorm-20160128_unranged: SummarizedExperiment with 2907 rows and 79 columns
#> Features:
#> experiments() - obtain the ExperimentList instance
#> colData() - the primary/phenotype DataFrame
#> sampleMap() - the sample availability DataFrame
#> `$, `[, `[[` - extract colData columns, subset, or experiment
#> *Format() - convert into a long or wide DataFrame
#> assays() - convert ExperimentList to a SimpleList of matrices
```

10.16.2 What types of samples are in the data?

Solution

The `sampleTables` function gives you an overview of samples in each assay:

```
sampleTables(acc)
#> $`ACC_CNVSNP-20160128`
#>
#> 01 10 11
#> 90 85 5
#>
#> $`ACC_miRNASeqGene-20160128`
#>
#> 01
#> 80
#>
#> $`ACC_Mutation-20160128`
#>
#> 01
#> 90
#>
#> $`ACC_RNASeq2GeneNorm-20160128`
#>
#> 01
#> 79
#>
#> $`ACC_RPPAArray-20160128`
#>
#> 01
#> 46

head(sampleTypes)
#>   Code          Definition Short.Letter.Code
#> 1    01          Primary Solid Tumor           TP
#> 2    02          Recurrent Solid Tumor          TR
#> 3    03 Primary Blood Derived Cancer - Peripheral Blood TB
```

```
#> 4 04 Recurrent Blood Derived Cancer - Bone Marrow
#> 5 05 Additional - New Primary
#> 6 06 Metastatic
```

<i>TRBM</i>
<i>TAP</i>
<i>TM</i>

10.16.3 Curated molecular subtypes

Is there subtype data available in the `MultiAssayExperiment` obtained from `curatedTCGAData`?

Solution

The `getSubtypeMap` function will show actual variable names found in `colData` that contain subtype information. This can only be obtained from `MultiAssayExperiment` objects provided by `curatedTCGAData`.

```
getSubtypeMap(acc)
#>           ACC_annotations      ACC_subtype
#> 1          Patient_ID          SAMPLE
#> 2 histological_subtypes      Histology
#> 3      mrna_subtypes          C1A/C1B
#> 4      mrna_subtypes          mRNA_K4
#> 5          cimp          MethyLevel
#> 6 microrna_subtypes      miRNA cluster
#> 7      scna_subtypes          SCNA cluster
#> 8 protein_subtypes      protein cluster
#> 9 integrative_subtypes      COC
#> 10 mutation_subtypes      OncoSign
head(colData(acc)$Histology)
#> [1] "Usual Type" "Usual Type" "Usual Type" "Usual Type" "Usual Type"
#> [6] "Usual Type"
```

10.16.4 Converting TCGA UUIDs to barcodes and back

`TCGAutils` provides a number of ID translation functions. These allow the user to translate from either file or case UUIDs to TCGA barcodes and back. These functions work by querying the Genomic Data Commons API via the `GenomicDataCommons` package (thanks to Sean Davis). These include:

- `UUIDtoBarcode()`
- `barcodeToUUID()`
- `UUIDtoUUID()`
- `filenameToBarcode()`

See the `TCGAutils` help pages for details.

10.16.5 Other TCGA data types

Helper functions to add TCGA exon files (legacy archive), copy number and GISTIC copy number calls to `MultiAssayExperiment` objects are also available in `TCGAutils`.

10.17 Plotting, correlation, and other analyses

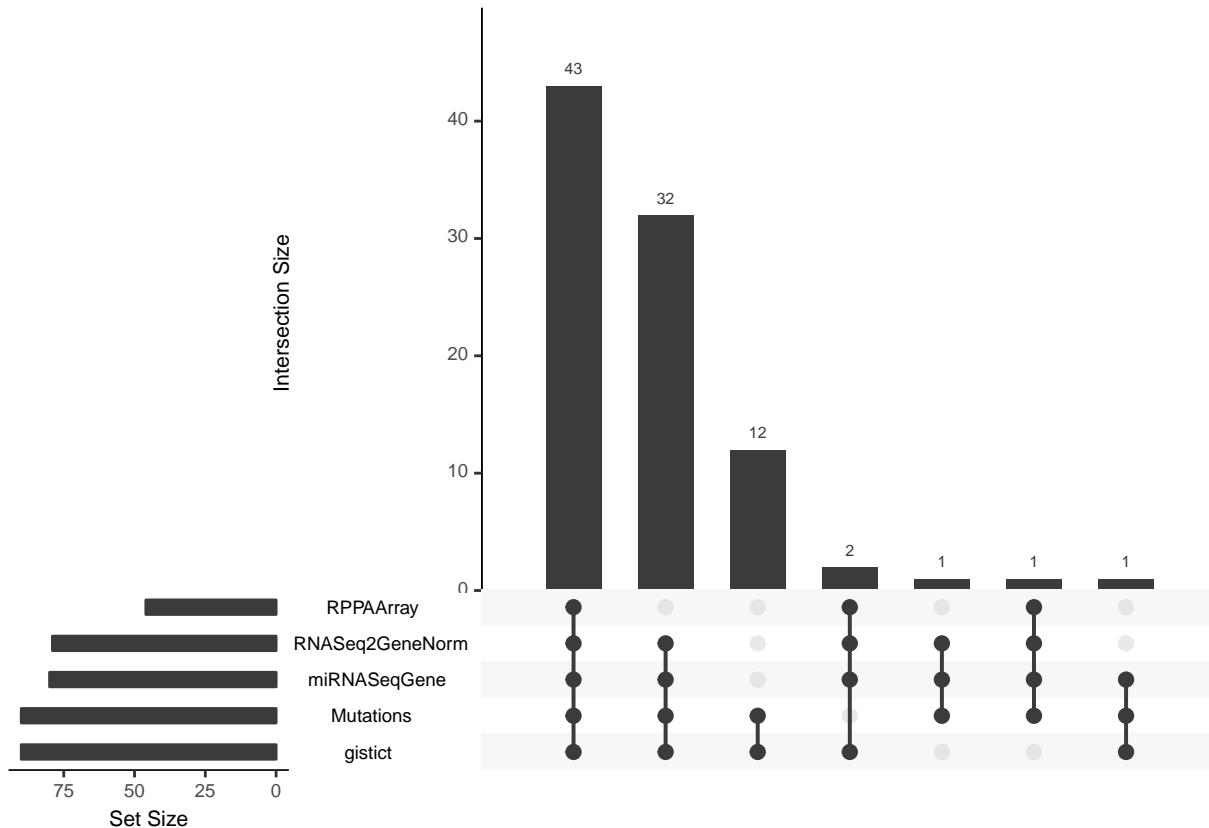
These provide exercises and solutions using the `miniACC` dataset.

10.17.1 How many miniACC samples have data for each combination of assays?

Solution

The built-in `upsetSamples` creates an “upset” Venn diagram to answer this question:

```
upsetSamples(miniACC)
```



In this dataset only 43 samples have all 5 assays, 32 are missing reverse-phase protein (RPPAArray), 2 are missing Mutations, 1 is missing gistic, 12 have only mutations and gistic, etc.

10.17.2 Kaplan-meier plot stratified by pathology_N_stage

Create a Kaplan-meier plot, using pathology_N_stage as a stratifying variable.

Solution

The `colData` provides clinical data for things like a Kaplan-Meier plot for overall survival stratified by nodal stage.

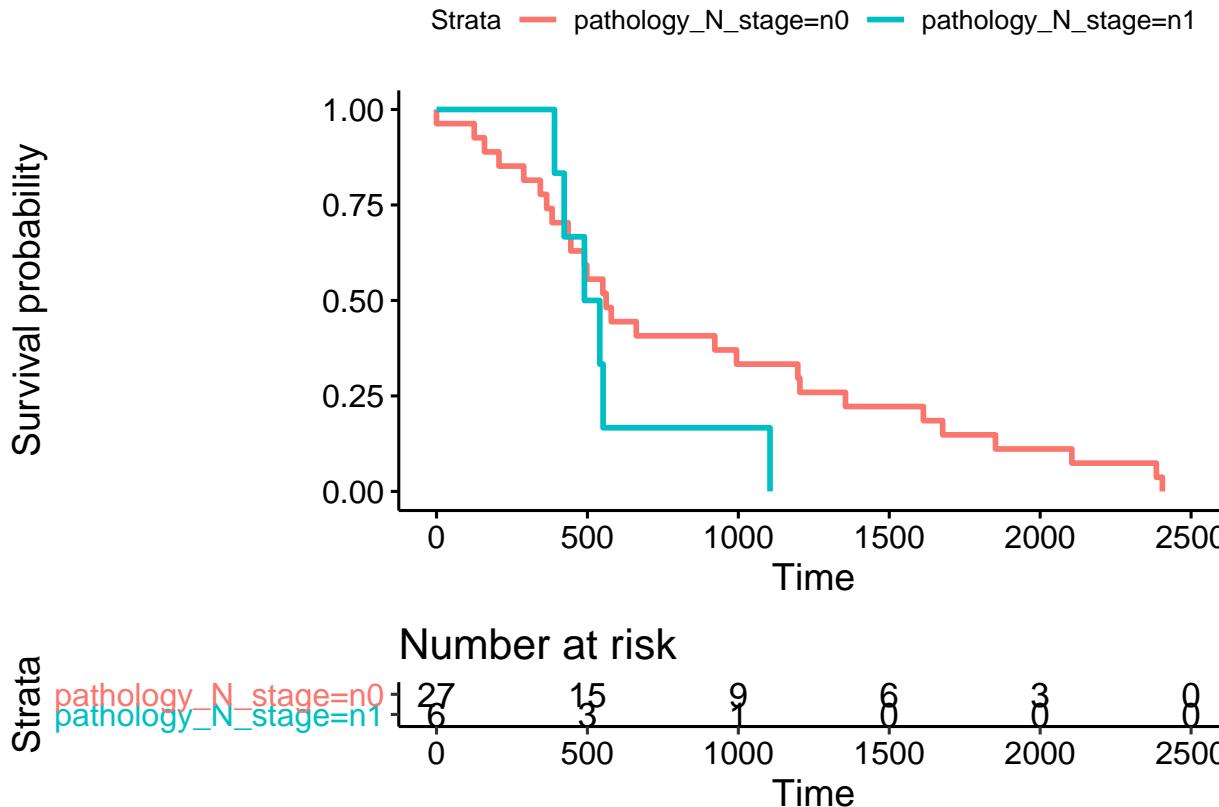
```
Surv(miniACC$days_to_death, miniACC$vital_status)
#> [1] 1355 1677 NA+ 423 365 NA+ 490 579 NA+ 922 551
#> [12] 1750 NA+ 2105 NA+ 541 NA+ NA+ 490 NA+ NA+ 562
#> [23] NA+ NA+ NA+ NA+ NA+ NA+ 289 NA+ NA+ NA+ 552
#> [34] NA+ NA+ NA+ 994 NA+ NA+ 498 NA+ NA+ 344 NA+
#> [45] NA+ NA+ NA+ NA+ NA+ NA+ NA+ NA+ 391 125
#> [56] NA+ 1852 NA+ NA+ NA+ NA+ NA+ NA+ 1204 159
#> [67] 1197 662 445 NA+ 2385 436 1105 NA+ 1613 NA+ NA+
```

```
#> [78] 2405    NA+    NA+    NA+    NA+    207     0      NA+    NA+    NA+
#> [89]    NA+    NA+    NA+    383
```

And remove any patients missing overall survival information:

```
miniACCsurv <- miniACC[, complete.cases(miniACC$days_to_death, miniACC$vital_status), ]
#> harmonizing input:
#> removing 248 sampleMap rows with 'colname' not in colnames of experiments
#> removing 58 colData rownames not in sampleMap 'primary'

fit <- survfit(Surv(days_to_death, vital_status) ~ pathology_N_stage, data = colData(miniACCsurv))
ggsurvplot(fit, data = colData(miniACCsurv), risk.table = TRUE)
```



10.17.3 Multivariate Cox regression including RNA-seq, copy number, and pathology

Choose the *EZH2* gene for demonstration. This subsetting will drop assays with no row named *EZH2*:

```
wideacc = wideFormat(miniACC["EZH2", , ],
  colDataCols=c("vital_status", "days_to_death", "pathology_N_stage"))
wideacc$y = Surv(wideacc$days_to_death, wideacc$vital_status)
head(wideacc)

#> DataFrame with 6 rows and 7 columns
#>           primary vital_status days_to_death pathology_N_stage
#> 1      TCGA-OR-A5J1          1        1355             n0
#> 2      TCGA-OR-A5J2          1        1677             n0
#> 3      TCGA-OR-A5J3          0         NA             n0
```

```
#> 4 TCGA-OR-A5J4      1      423      n1
#> 5 TCGA-OR-A5J5      1      365      n0
#> 6 TCGA-OR-A5J6      0      NA       n0
#>   RNASeq2GeneNorm_EZH2  gistiict_EZH2      y
#>           <numeric>    <numeric> <Surv>
#> 1      75.8886      0 1355:1
#> 2      326.5332      1 1677:1
#> 3      190.194       1  NA:0
#> 4      NA            -2 423:1
#> 5      366.3826      1 365:1
#> 6      30.7371      1  NA:0
```

Perform a multivariate Cox regression with *EZH2* copy number (*gistiict*), log2-transformed *EZH2* expression (RNASeq2GeneNorm), and nodal status (*pathology_N_stage*) as predictors:

```
coxph(Surv(days_to_death, vital_status) ~ gistiict_EZH2 + log2(RNASeq2GeneNorm_EZH2) + pathology_N_stage
      data=wideacc)
#> Call:
#> coxph(formula = Surv(days_to_death, vital_status) ~ gistiict_EZH2 +
#>   log2(RNASeq2GeneNorm_EZH2) + pathology_N_stage, data = wideacc)
#>
#>           coef  exp(coef) se(coef)     z      p
#> gistiict_EZH2 -0.0372    0.9635   0.2821 -0.13 0.89499
#> log2(RNASeq2GeneNorm_EZH2) 0.9773    2.6573   0.2811  3.48 0.00051
#> pathology_N_stagen1  0.3775    1.4586   0.5699  0.66 0.50774
#>
#> Likelihood ratio test=16.28 on 3 df, p=0.001
#> n= 26, number of events= 26
#>   (66 observations deleted due to missingness)
```

We see that *EZH2* expression is significantly associated with overall survival ($p < 0.001$), but *EZH2* copy number and nodal status are not. This analysis could easily be extended to the whole genome for discovery of prognostic features by repeated univariate regressions over columns, penalized multivariate regression, etc.

For further detail, see the main MultiAssayExperiment vignette.

[back to top](#)

10.17.4 Correlation between RNA-seq and copy number

Part 1

For all genes where there is both recurrent copy number (*gistiict* assay) and RNA-seq, calculate the correlation between $\log_2(\text{RNAseq} + 1)$ and copy number. Create a histogram of these correlations. Compare this with the histogram of correlations between all *unmatched* gene - copy number pairs.

Solution

First, narrow down `miniACC` to only the assays needed:

```
subacc <- miniACC[, , c("RNASeq2GeneNorm", "gistiict")]
```

Align the rows and columns, keeping only samples with both assays available:

```
subacc <- intersectColumns(subacc)
#> harmonizing input:
#>   removing 15 sampleMap rows with 'colname' not in colnames of experiments
```

```
#>   removing 15 colData rownames not in sampleMap 'primary'
subacc <- intersectRows(subacc)
```

Create a list of numeric matrices:

```
subacc.list <- assays(subacc)
```

Log-transform the RNA-seq assay:

```
subacc.list[[1]] <- log2(subacc.list[[1]] + 1)
```

Transpose both, so genes are in columns:

```
subacc.list <- lapply(subacc.list, t)
```

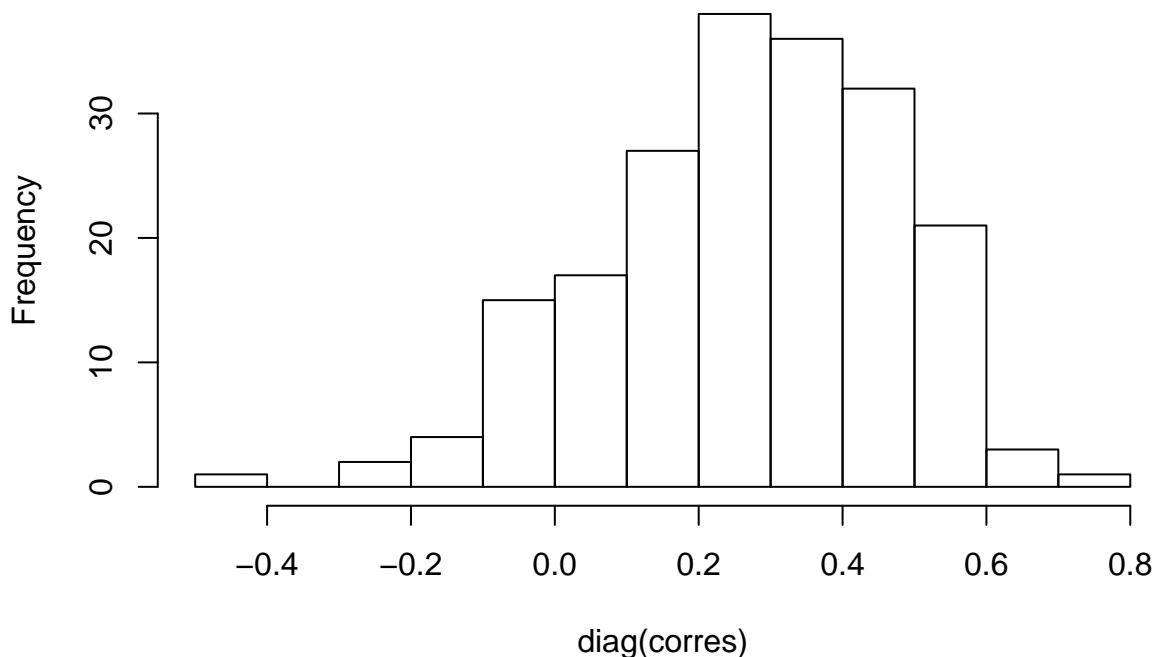
Calculate the correlation between columns in the first matrix and columns in the second matrix:

```
corres <- cor(subacc.list[[1]], subacc.list[[2]])
```

And finally, create the histograms:

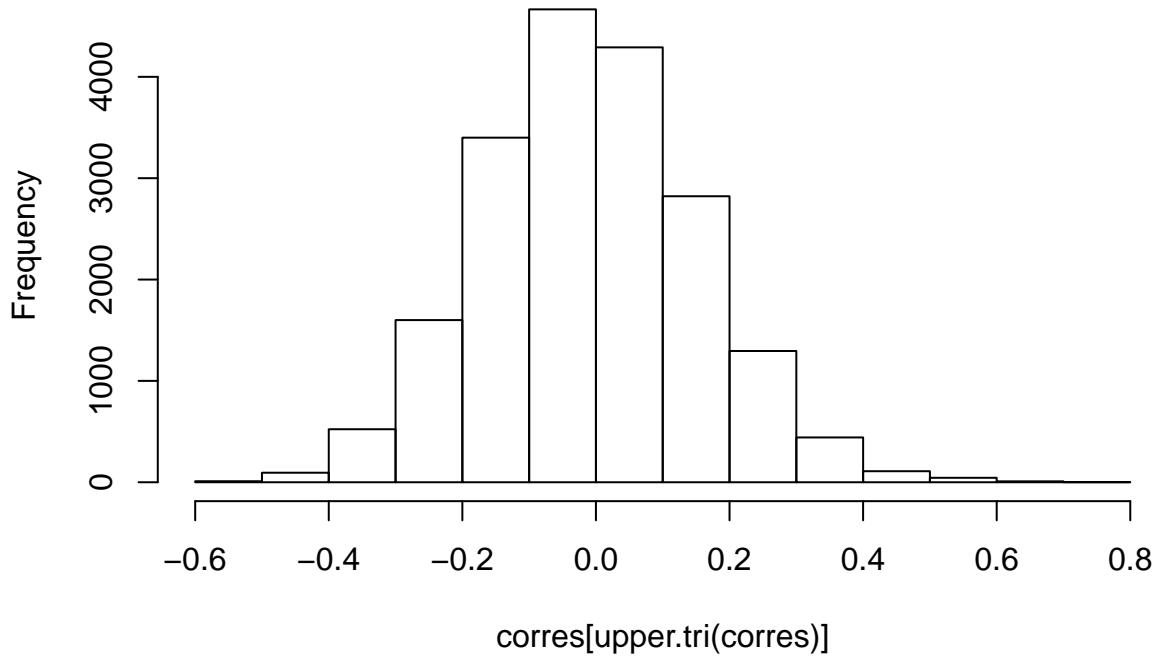
```
hist(diag(corres))
```

Histogram of diag(corres)



```
hist(corres[upper.tri(corres)])
```

Histogram of corre[upper.tri(corre)]



Part 2

For the gene with highest correlation to copy number, make a box plot of log2 expression against copy number.

Solution

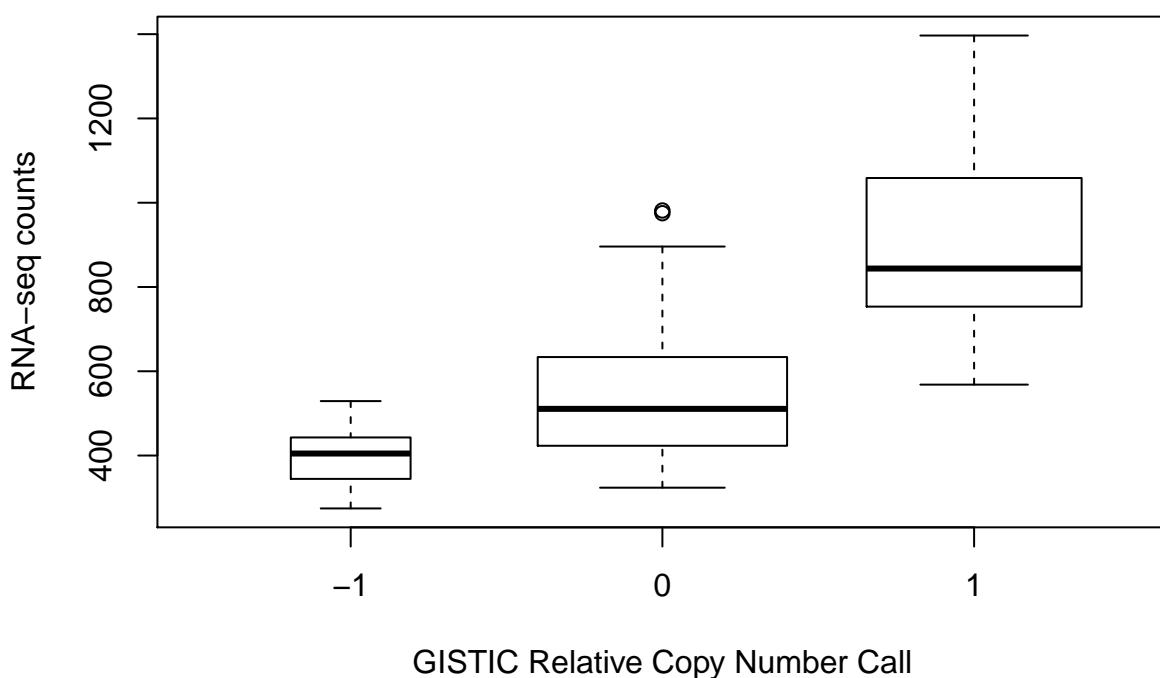
First, identify the gene with highest correlation between expression and copy number:

```
which.max(diag(corre))
#> EIF4E
#> 91
```

You could now make the plot by taking the EIF4E columns from each element of the list subacc.list *list* that was extracted from the subacc *MultiAssayExperiment*, but let's do it by subsetting and extracting from the *MultiAssayExperiment*:

```
df <- wideFormat(subacc["EIF4E", , ])
head(df)
#> DataFrame with 6 rows and 3 columns
#>      primary.RNASeq2GeneNorm_EIF4E gistic_EIF4E
#>      <character>             <numeric>     <numeric>
#> 1 TCGA-OR-A5J1          460.6148       0
#> 2 TCGA-OR-A5J2          371.2252       0
#> 3 TCGA-OR-A5J3          516.0717       0
#> 4 TCGA-OR-A5J5         1129.3571       1
#> 5 TCGA-OR-A5J6          822.0782       0
#> 6 TCGA-OR-A5J7          344.5648      -1

boxplot(RNASeq2GeneNorm_EIF4E ~ gistic_EIF4E,
        data=df, varwidth=TRUE,
        xlab="GISTIC Relative Copy Number Call",
        ylab="RNA-seq counts")
```



[back to top](#)

10.17.5 Identifying correlated principal components

Perform Principal Components Analysis of each of the five assays, using samples available on each assay, log-transforming RNA-seq data first. Using the first 10 components, calculate Pearson correlation between all scores and plot these correlations as a heatmap to identify correlated components across assays.

Solution

Here's a function to simplify doing the PCAs:

```
getLoadings <- function(x, ncomp=10, dolog=FALSE, center=TRUE, scale.=TRUE){
  if(dolog){
    x <- log2(x + 1)
  }
  pc = prcomp(x, center=center, scale.=scale.)
  return(t(pc$rotation[, 1:10]))
}
```

Although it would be possible to do the following with a loop, the different data types do require different options for PCA (e.g. mutations are a 0/1 matrix with 1 meaning there is a somatic mutation, and gistic varies between -2 for homozygous loss and 2 for a genome doubling, so neither make sense to scale and center). So it is just as well to do the following one by one, concatenating each PCA results to the MultiAssayExperiment:

```
miniACC2 <- intersectColumns(miniACC)
#> harmonizing input:
#>   removing 170 sampleMap rows with 'colname' not in colnames of experiments
#>   removing 49 colData rownames not in sampleMap 'primary'
miniACC2 <- c(miniACC2, rnaseqPCA=getLoadings(assays(miniACC2)[[1]], dolog=TRUE), mapFrom=1L)
#> Warning in .local(x, ...): Assuming column order in the data provided
#>   matches the order in 'mapFrom' experiment(s) colnames
```

```
miniACC2 <- c(miniACC2, gisticaPCA=getLoadings(assays(miniACC2)[[2]], center=FALSE, scale.=FALSE), mapFrom)
#> Warning in .local(x, ...): Assuming column order in the data provided
#> matches the order in 'mapFrom' experiment(s) colnames
miniACC2 <- c(miniACC2, proteinPCA=getLoadings(assays(miniACC2)[[3]]), mapFrom=3L)
#> Warning in .local(x, ...): Assuming column order in the data provided
#> matches the order in 'mapFrom' experiment(s) colnames
miniACC2 <- c(miniACC2, mutationsPCA=getLoadings(assays(miniACC2)[[4]], center=FALSE, scale.=FALSE), mapFrom)
#> Warning in .local(x, ...): Assuming column order in the data provided
#> matches the order in 'mapFrom' experiment(s) colnames
miniACC2 <- c(miniACC2, miRNAPCA=getLoadings(assays(miniACC2)[[5]]), mapFrom=5L)
#> Warning in .local(x, ...): Assuming column order in the data provided
#> matches the order in 'mapFrom' experiment(s) colnames
```

Now subset to keep *only* the PCA results:

```
miniACC2 <- miniACC2[, , 6:10]
experiments(miniACC2)
#> ExperimentList class object of length 5:
#> [1] rnaseqPCA: matrix with 10 rows and 43 columns
#> [2] gisticaPCA: matrix with 10 rows and 43 columns
#> [3] proteinPCA: matrix with 10 rows and 43 columns
#> [4] mutationsPCA: matrix with 10 rows and 43 columns
#> [5] miRNAPCA: matrix with 10 rows and 43 columns
```

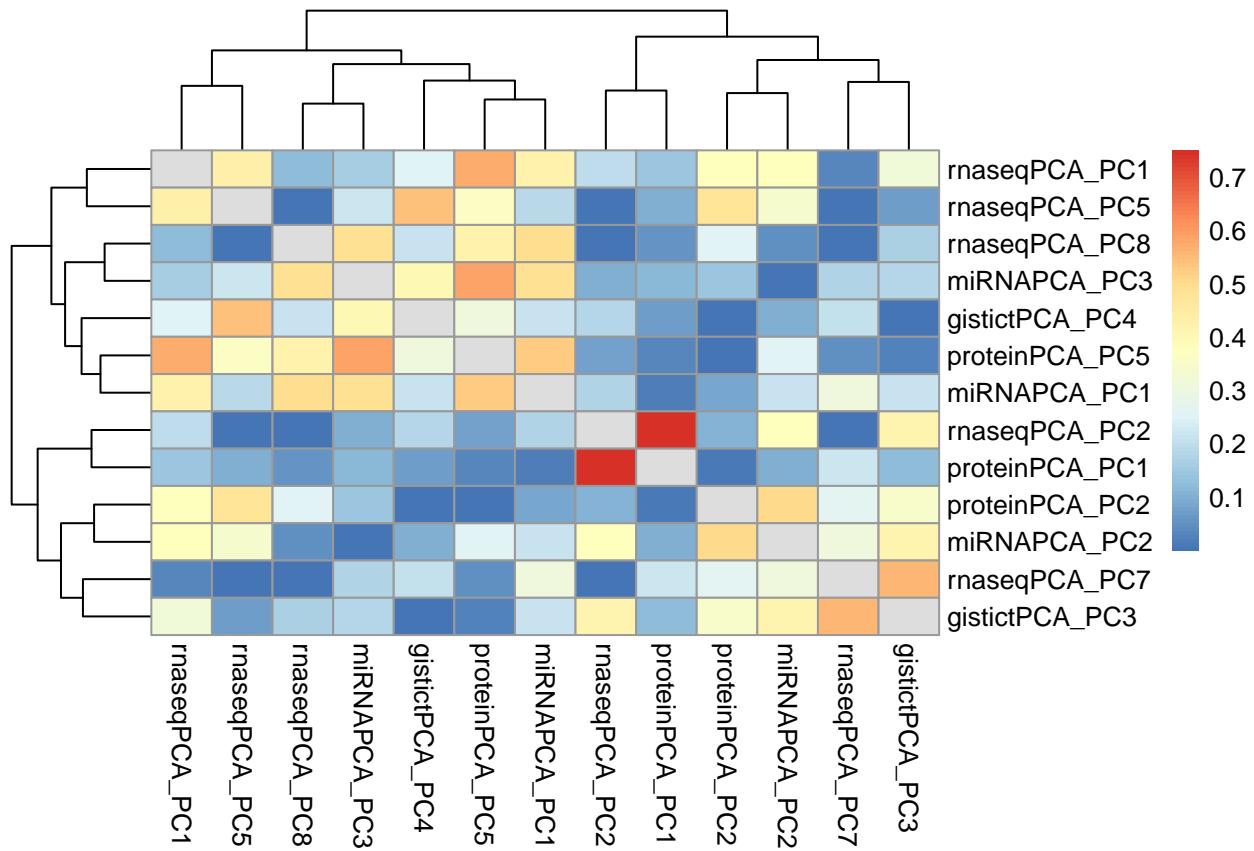
Note, it would be equally easy (and maybe better) to do PCA on all samples available for each assay, then do intersectColumns at this point instead.

Now, steps for calculating the correlations and plotting a heatmap: * Use *wideFormat* to paste these together, which has the nice property of adding assay names to the column names. * The first column always contains the sample identifier, so remove it. * Coerce to a matrix * Calculate the correlations, and take the absolute value (since signs of principal components are arbitrary) * Set the diagonals to NA (each variable has a correlation of 1 to itself).

```
df <- wideFormat(miniACC2)[, -1]
mycors <- cor(as.matrix(df))
mycors <- abs(mycors)
diag(mycors) <- NA
```

To simplify the heatmap, show only components that have at least one correlation greater than 0.5.

```
has.high.cor <- apply(mycors, 2, max, na.rm=TRUE) > 0.5
mycors <- mycors[has.high.cor, has.high.cor]
pheatmap::pheatmap(mycors)
```



The highest correlation present is between PC2 of the RNA-seq assay, and PC1 of the protein assay.

[back to top](#)

10.17.6 Annotating with ranges

This section doesn't use the `TCGAutils` shortcuts, and is more generally applicable.

Convert all the `ExperimentList` elements in `miniACC` to `RangedSummarizedExperiment` objects. Then use `rowRanges` to annotate these objects with genomic ranges. For the microRNA assay, annotate instead with the genomic coordinates of predicted targets.

Solution

The following shortcut function takes a list of human gene symbols and uses `AnnotationFilter` and `EnsDb.Hsapiens.v86` to look up the ranges, and return these as a `GRangesList` which can be used to replace the `rowRanges` of the `SummarizedExperiment` objects:

```
getrr <- function(identifiers, EnsDbFilterFunc=AnnotationFilter::SymbolFilter) {
  edb <- EnsDb.Hsapiens.v86::EnsDb.Hsapiens.v86
  afl <- AnnotationFilterList(
    EnsDbFilterFunc(identifiers),
    SeqNameFilter(c(1:21, "X", "Y")),
    TxBiotypeFilter("protein_coding"))
  gr <- genes(edb, filter=afl)
  grl <- split(gr, factor(identifiers))
  grl <- grl[match(identifiers, names(grl))]
  stopifnot(identical(names(grl), identifiers))
```

```

    return(gr1)
}

```

For example:

```

getrr(rownames(miniACC)[[1]])
#> GRangesList object of length 198:
#> $DIRAS3
#> GRanges object with 1 range and 7 metadata columns:
#>           seqnames      ranges strand |      gene_id
#>           <Rle>      <IRanges> <Rle> |      <character>
#> ENSG00000116288     1 7954291-7985505 + | ENSG00000116288
#>           gene_name   gene_biotype seq_coord_system      symbol
#>           <character>   <character>   <character> <character>
#> ENSG00000116288     PARK7 protein_coding      chromosome      PARK7
#>           entrezid    tx_biotype
#>           <list>      <character>
#> ENSG00000116288     11315 protein_coding
#>
#> $MAPK14
#> GRanges object with 1 range and 7 metadata columns:
#>           seqnames      ranges strand |      gene_id
#>           <character>   <character>   <character> <character>
#> ENSG00000116285     1 8004404-8026308 - | ENSG00000116285
#>           gene_name   gene_biotype seq_coord_system symbol
#>           <character>   <character>   <character> <character>
#> ENSG00000116285     ERRFI1 protein_coding      chromosome ERRFI1
#>           entrezid    tx_biotype
#> ENSG00000116285     54206 protein_coding
#>
#> $YAP1
#> GRanges object with 1 range and 7 metadata columns:
#>           seqnames      ranges strand |      gene_id
#>           <character>   <character>   <character> <character>
#> ENSG00000198793     1 11106535-11262507 - | ENSG00000198793
#>           gene_name   gene_biotype seq_coord_system symbol
#>           <character>   <character>   <character> <character>
#> ENSG00000198793     MTOR protein_coding      chromosome MTOR
#>           entrezid    tx_biotype
#> ENSG00000198793     2475 protein_coding
#>
#> ...
#> <195 more elements>
#> -----
#> seqinfo: 22 sequences from GRCh38 genome

```

Use this to set the rowRanges of experiments 1-4 with these GRangesList objects

```

rseACC <- miniACC
withRSE <- c(1:3, 5)
for (i in withRSE){
  rowRanges(rseACC[[i]]) <- getrr(rownames(rseACC[[i]]))
}

```

Note that the class of experiments 1-4 is now RangedSummarizedExperiment:

```

experiments(rseACC)
#> ExperimentList class object of length 5:
#> [1] RNASeq2GeneNorm: RangedSummarizedExperiment with 198 rows and 79 columns
#> [2] gistic: RangedSummarizedExperiment with 198 rows and 90 columns

```

```
#> [3] RPPAArray: RangedSummarizedExperiment with 33 rows and 46 columns
#> [4] Mutations: matrix with 97 rows and 90 columns
#> [5] miRNASeqGene: RangedSummarizedExperiment with 471 rows and 80 columns
```

With ranged objects in the MultiAssayExperiment, you can then do subsetting by ranges. For example, select all genes on chromosome 1 for the four *rangedSummarizedExperiment* objects:

```
rseACC[GRanges(seqnames="1:1-1e9"), , withRSE]
#> A MultiAssayExperiment object of 3 listed
#> experiments with user-defined names and respective classes.
#> Containing an ExperimentList class object of length 3:
#> [1] RNASeq2GeneNorm: RangedSummarizedExperiment with 22 rows and 79 columns
#> [2] gistic: RangedSummarizedExperiment with 22 rows and 90 columns
#> [3] RPPAArray: RangedSummarizedExperiment with 3 rows and 46 columns
#> Features:
#> experiments() - obtain the ExperimentList instance
#> colData() - the primary/phenotype DataFrame
#> sampleMap() - the sample availability DataFrame
#> `$, `[, `[[` - extract colData columns, subset, or experiment
#> *Format() - convert into a long or wide DataFrame
#> assays() - convert ExperimentList to a SimpleList of matrices
```

Note about microRNA: You can set ranges for the microRNA assay according to the genomic location of those microRNA, or the locations of their predicted targets, but we don't do it here. Assigning genomic ranges of microRNA targets would be an easy way to subset them according to their targets.

[back to top](#)

Chapter 11

230: Cytoscape automation in R using Rcy3

11.1 Overview

11.1.1 Instructor names and contact information

- Ruth Isserlin - ruth dot isserlin (at) utoronto (dot) ca
- Brendan Innes - brendan (dot) innes (at) mail (dot) utoronto (dot) ca
- Jeff Wong - jvwong (at) gmail (dot) com
- Gary Bader - gary (dot) bader (at) utoronto (dot) ca

11.1.2 Workshop Description

Cytoscape(www.cytoscape.org) is one of the most popular applications for network analysis and visualization. In this workshop, we will demonstrate new capabilities to integrate Cytoscape into programmatic workflows and pipelines using R. We will begin with an overview of network biology themes and concepts, and then we will translate these into Cytoscape terms for practical applications. The bulk of the workshop will be a hands-on demonstration of accessing and controlling Cytoscape from R to perform a network analysis of tumor expression data.

11.1.3 Workshop prerequisites:

- Basic knowledge of R syntax
- Basic knowledge of Cytoscape software
- Familiarity with network biology concepts

11.1.4 Background:

- “How to visually interpret biological data using networks.” Merico D, Gfeller D, Bader GD. Nature Biotechnology 2009 Oct 27, 921-924 - http://baderlab.org/Publications?action=AttachFile&do=view&target=2009_Merico_Primer_NatBiotech_Oct.pdf
- “CyREST: Turbocharging Cytoscape Access for External Tools via a RESTful API”. Keiichiro Ono, Tanja Muetze, Georgi Kolishovski, Paul Shannon, Barry Demchak. F1000Res. 2015 Aug 5;4:478. - <https://f1000research.com/articles/4-478/v1>

11.1.5 Workshop Participation

Participants are required to bring a laptop with Cytoscape, R, and RStudio installed. Installation instructions will be provided in the weeks preceding the workshop. The workshop will consist of a lecture and lab.

11.1.6 R / Bioconductor packages used

- RCy3
- gProfileR
- RCurl
- EnrichmentBrowser

11.1.7 Time outline

Activity	Time
Introduction	15m
Driving Cytoscape from R	15m
Creating, retrieving and manipulating networks	15m
Summary	10m

11.1.8 Workshop goals and objectives

Learning goals

- Know when and how to use Cytoscape in your research area
- Generalize network analysis methods to multiple problem domains
- Integrate Cytoscape into your bioinformatics pipelines

Learning objectives

- Programmatic control over Cytoscape from R
- Publish, share and export networks

11.2 Background

11.2.1 Cytoscape



- Cytoscape(Shannon et al. 2003) is a freely available open-source, cross platform network analysis software.
- Cytoscape(Shannon et al. 2003) can visualize complex networks and help integrate them with any other data type.

- Cytoscape(Shannon et al. 2003) has an active developer and user base with more than **300** community created apps available from the (Cytoscape App Store)[apps.cytoscape.org].
- Check out some of the tasks you can do with Cytoscape in our online tutorial guide - tutorials.cytoscape.org

11.2.2 Overview of network biology themes and concepts

11.2.2.1 Why Networks?

Networks are everywhere...

- Molecular Networks
- Cell-Cell communication Networks
- Computer networks
- Social Networks
- Internet

Networks are powerful tools...

- Reduce complexity
- More efficient than tables
- Great for data integration
- Intuitive visualization

Often data in our pipelines are represented as data.frames, tables, matrices, vectors or lists. Sometimes we represent this data as heatmaps, or plots in efforts to summarize the results visually. Network visualization offers an additional method that readily incorporates many different data types and variables into a single picture.

In order to translate your data into a network it is important to define the entities and their relationships. Entities and relationships can be anything. They can be user defined or they can be queried from a database.

Examples of Networks and their associated entities:

- **Protein - Protein interaction network** - is a directed or undirected network where nodes in the network are proteins or genes and edges represent how those proteins interact.
- **Gene - gene interaction network** - nodes in the network are genes and edges can represent synthetic lethality i.e. two genes have a connection if deleting both of them cause a decrease in fitness.
- **Coexpression network** - nodes in the network are genes or proteins and edges represent the degree of co-expression the two genes have. Often the edges are associated with a correlation score (i.e. pearson correlation) and edges are filtered by a defined threshold. If no threshold is specified all genes will be connected to all other genes creating a hairball.
- **Enrichment Map** - nodes in the networks are groups of genes from pathways or functions (i.e. genesets) and edges represent pathway crosstalk (genes in common).
- **Social network** - nodes in the network are individuals and edges are some sort of social interaction between two individuals, for example, friends on Facebook, linked in LinkedIn, ...
- **Copublication network** - a specialization of the social network for research purposes. Nodes in the network are individuals and edges between any two individuals for those who have co-authored a publication together.

11.2.3 Networks as Tools

Networks can be used for two main purposes but often go hand in hand.

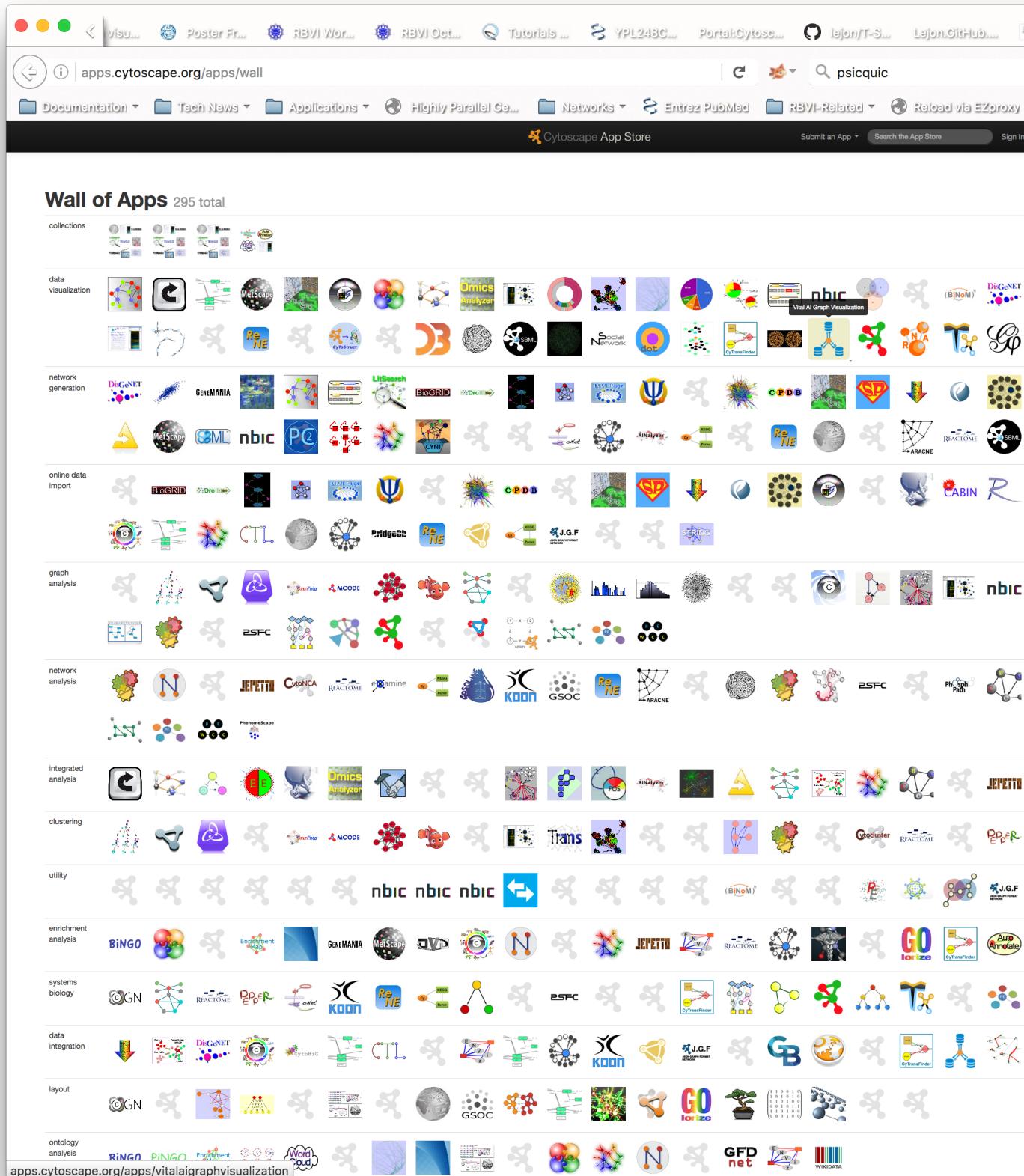
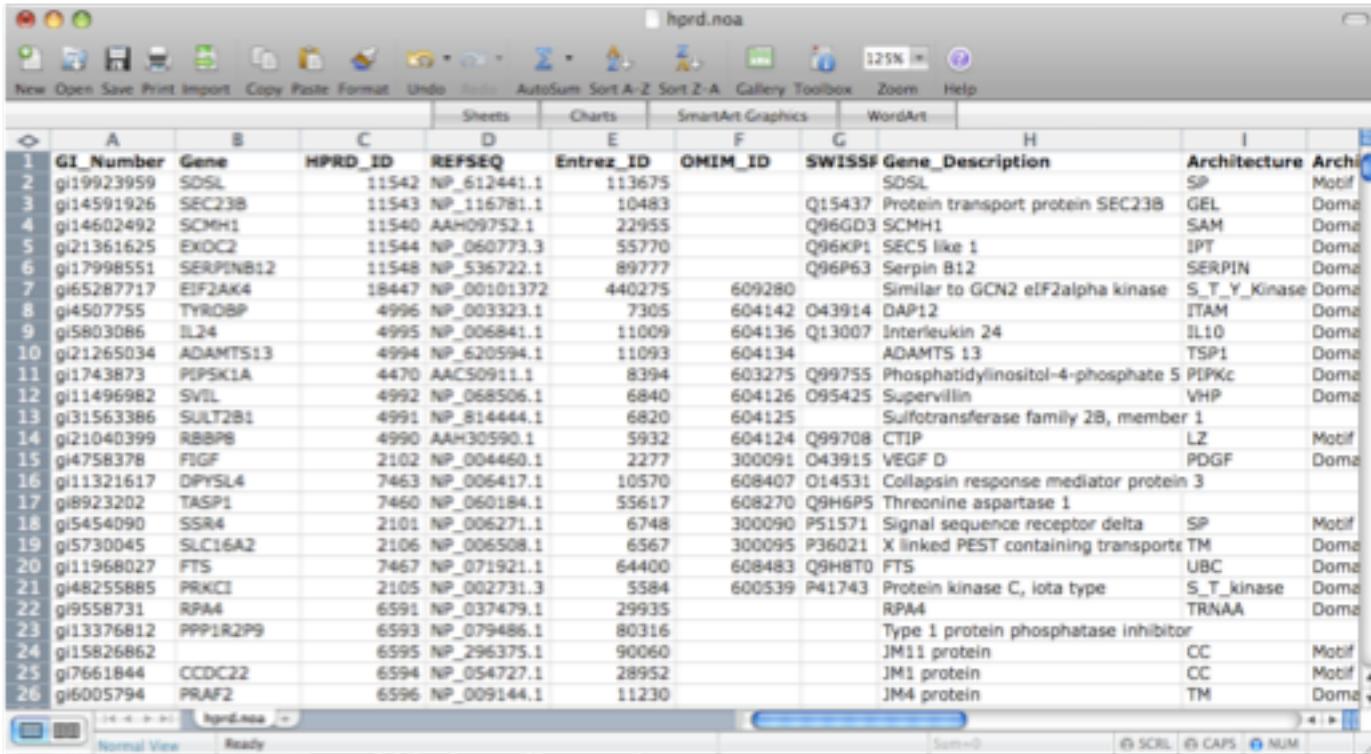


Figure 11.1: Available Cytoscape Apps



The screenshot shows a Microsoft Excel spreadsheet titled "hprd.noa". The table has 11 columns and 26 rows. The columns are labeled A through K, and the rows are numbered 1 through 26. The data includes various gene identifiers, their descriptions, and associated biological features like architecture and motifs.

	A	B	C	D	E	F	G	H	I	J
1	GI_Number	Gene	HPRD_ID	REFSEQ	Entrez_ID	OMIM_ID	SWISSP	Gene_Description	Architecture	Arch
2	gi19923959	SDSL	11542	NP_612441.1	113675		SDSL		SP	Motif
3	gi14591926	SEC23B	11543	NP_116781.1	10483		Q15437	Protein transport protein SEC23B	GEL	Doma
4	gi14602492	SCMH1	11540	AAH09752.1	22955		Q96GD3	SCMH1	SAM	Doma
5	gi21361625	EXOC2	11544	NP_060773.3	55770		Q96KP1	SEC5 like 1	IPT	Doma
6	gi17998551	SERPINB12	11548	NP_536722.1	89777		Q96P63	Serpin B12	SERPIN	Doma
7	gi65287717	EIF2AK4	18447	NP_00101372	440275	609280		Similar to GCN2 eIF2alpha kinase	S_T_Y_Kinase	Doma
8	gi4507755	TYROBP	4996	NP_003323.1	7305	604142	Q43914	DAP12	ITAM	Doma
9	gi5803086	IL24	4995	NP_006841.1	11009	604136	Q13007	Interleukin 24	IL10	Doma
10	gi21265034	ADAMTS13	4994	NP_620594.1	11093	604134	Q14531	ADAMTS 13	TSP1	Doma
11	gi1743873	PIP5K1A	4470	AAC50911.1	8394	603275	Q99755	Phosphatidylinositol-4-phosphate 5 PIPKc	VHP	Doma
12	gi11496982	SVIL	4992	NP_068506.1	6840	604126	Q95425	Supervillin		
13	gi31563386	SULT2B1	4991	NP_814444.1	6820	604125		Sulfotransferase family 2B, member 1		
14	gi21040399	RBBPB	4990	AAH30590.1	5932	604124	Q99708	CTIP	LZ	Motif
15	gi4758378	FIGF	2102	NP_004460.1	2277	300091	Q43915	VEGF D	PDGF	Doma
16	gi11321617	DPYSL4	7463	NP_006417.1	10570	608407	Q14531	Collapsin response mediator protein 3		
17	gi8923202	TASP1	7460	NP_060184.1	55617	608270	Q9H6P5	Threonine aspartase 1		
18	gi5454090	SSR4	2101	NP_006271.1	6748	300090	P51571	Signal sequence receptor delta	SP	Motif
19	gi5730045	SLC16A2	2106	NP_006508.1	6567	300095	P36021	X linked PEST containing transport	TM	Doma
20	gi11968027	FTS	7467	NP_071921.1	64400	608483	Q9H8T0	FTS	UBC	
21	gi48255885	PRKCI	2105	NP_002731.3	5584	600539	P41743	Protein kinase C, iota type	S_T_kinase	Doma
22	gi9558731	RPA4	6591	NP_037479.1	29935			RPA4	TRNAA	Doma
23	gi13376812	PPP1R2P9	6593	NP_079486.1	80316			Type 1 protein phosphatase inhibitor		
24	gi15826862		6595	NP_296375.1	90060			JM11 protein	CC	Motif
25	gi7661844	CCDC22	6594	NP_054727.1	28952			JM1 protein	CC	Motif
26	gi6005794	PRAF2	6596	NP_009144.1	11230			JM4 protein	TM	Doma

Tables

Figure 11.2: Translating data to networks

Analysis

- Topological properties - including number of nodes, number of edges, node degree, average clustering coefficients, shortest path lengths, density, and many more. Topological properties can help gain insights into the structure and organization of the resulting biological networks as well as help highlight specific node or regions of the network.
- Hubs and subnetworks - a hub is generally a highly connected node in a scale-free network. Removal of hubs cause rapid breakdown of the underlying network. Subnetworks are interconnected regions of the network and can be defined by any user defined parameter.
- Cluster, classify, and diffuse
- Data integration

Visualization

- Data overlays
- Layouts and animation
- Exploratory analysis
- Context and interpretation

11.3 Translating biological data into Cytoscape using RCy3

Networks offer us a useful way to represent our biological data. But how do we seamlessly translate our data from R into Cytoscape?

There are multiple ways to communicate with Cytoscape programmatically. There are two main complementary portals, **cyRest** (Ono et al. 2015) and **Commands**, that form the foundation. cyRest transforms Cytoscape in to a REST (Representational State Transfer) enabled service where it essentially listens for events through a predefined port (by default port 1234). The cyRest functionality started as an app add in but has now been incorporated into the main release. Commands, on the other hand, offer a mechanism whereby app developers can expose their functionality to other apps or to user through the command interface. Prior to the implementation of cyRest many of the basic network functions were first available as commands so there is some overlap between the two different methods. 11.3 shows the different ways you can call Cytoscape.

11.4 Set up

In order to create networks in Cytoscape from R you need:

- **RCy3** - a bioconductor package

```
if (! "RCy3" %in% installed.packages()){
  install.packages("BiocManager")
  BiocManager::install("RCy3")
}
library(RCy3)
```

- **Cytoscape** - Download and install Cytoscape 3.6.1. or higher. Java 9 is not supported. Please make sure that Java 8 is installed.
- Install additional cytoscape apps that will be used in this workshop. If using cytoscape 3.6.1 or older the apps need to manually installed through the app manager in Cytoscape or through your web browser. (click on the method to see detailed instructions)

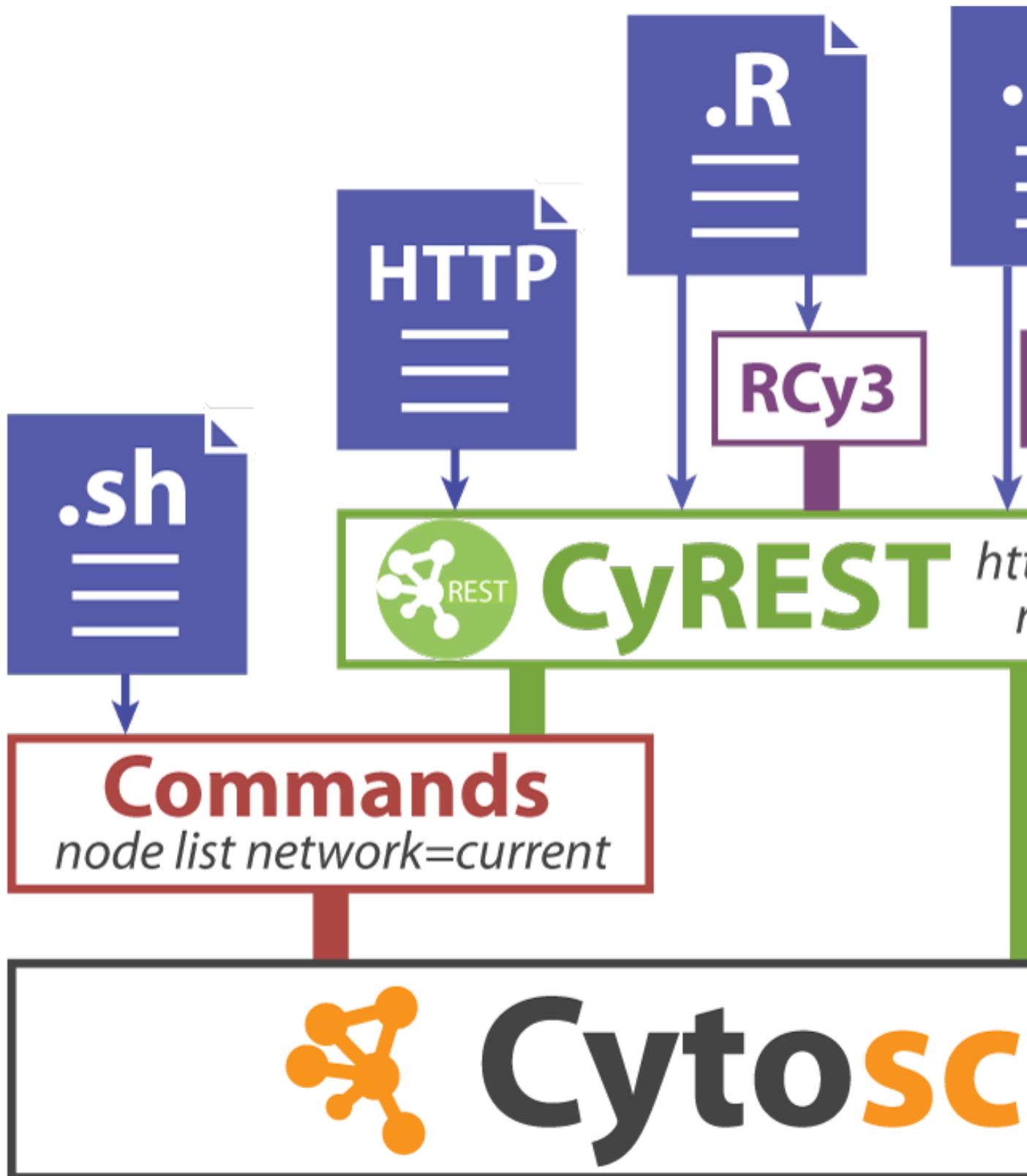


Figure 11.3: Different ways to communicate with Cytoscape

- Functional Enrichment Collection -a collection of apps to retrieve networks and pathways, integrate and explore the data, perform functional enrichment analysis, and interpret and display your results.
- EnrichmentMap Pipeline Collection - a collection of apps including EnrichmentMap, AutoAnnotate, WordCloud and clusterMaker2 used to visualize and analysis enrichment results.

If you are using Cytoscape 3.7 or higher (Cytoscape 3.7 will be released in August 2018) then apps can be installed directly from R.

```
#list of cytoscape apps to install
installation_responses <- c()

#list of app to install
cyto_app_toinstall <- c("clustermaker2", "enrichmentmap", "autoannotate", "wordcloud", "stringapp", "aMa

cytoscape_version <- unlist(strsplit( cytoscapeVersionInfo() ['cytoscapeVersion'],split = "\\\\" ))
if(length(cytoscape_version) == 3 && as.numeric(cytoscape_version[1]>=3)
  && as.numeric(cytoscape_version[2]>=7)){
  for(i in 1:length(cyto_app_toinstall)){
    #check to see if the app is installed. Only install it if it hasn't been installed
    if(!grep(commandsGET(paste("apps status app=\"", cyto_app_toinstall[i], "\"", sep="")), pattern = "status: Installed")){
      installation_response <- commandsGET(paste("apps install app=\"",
                                                    cyto_app_toinstall[i], "\"", sep=""))
      installation_responses <- c(installation_responses,installation_response)
    } else{
      installation_responses <- c(installation_responses,"already installed")
    }
  }
  installation_summary <- data.frame(name = cyto_app_toinstall,
                                       status = installation_responses)

  knitr::kable(list(installation_summary),
  booktabs = TRUE, caption = 'A Summary of automated app installation'
)
}
```

Make sure that Cytoscape is open

```
cytoscapePing ()
```

```
## [1] "You are connected to Cytoscape!"
```

11.4.1 Getting started

11.4.1.1 Confirm that Cytoscape is installed and opened

```
cytoscapeVersionInfo ()
```

```
##      apiVersion cytoscapeVersion
##              "v1"  "3.7.0-SNAPSHOT"
```

11.4.1.2 Browse available functions, commands and arguments

Depending on what apps you have installed there is different functionality available.

To see all the functions available in RCy3 package

```
help(package=RCy3)
```

Open swagger docs for live instances of CyREST API. The CyREST API list all the functions available in a base distribution of cytoscape. The below command will launch the swagger documentation in a web browser. Functions are clustered into categories. Expanding individual categories will show all the option available. Further expanding an individual command will show detailed documentation for the function, input, outputs and allow you to try and run the function. Running the function will show the url used for the query and all returned responses.

```
cyrestAPI() # CyREST API
```

As mentioned above, there are two ways to interact with Cytoscape, through the Cyrest API or commands. To see the available commands in swagger similar to the Cyrest API.

```
commandsAPI() # Commands API
```

To get information about an individual command from the R environment you can also use the commandsHelp function. Simply specify what command you would like to get information on by adding its name to the command. For example “commandsHelp(“help”)”

```
commandsHelp("help")
```

```
## [1] "Available namespaces:"
```

## [1] "apps"	"autoannotate"	"chemviz"
## [4] "cluster"	"clusterdimreduce"	"clusterrank"
## [7] "clusterviz"	"command"	"cybrowser"
## [10] "diffusion"	"edge"	"enrichmentmap"
## [13] "filter"	"group"	"idmapper"
## [16] "layout"	"network"	"node"
## [19] "session"	"string"	"table"
## [22] "view"	"vizmap"	"wordcloud"

Table 11.2: A table of example nodes and edges.

id	group	score	source	target	interaction	weight
node 0	A	20	node 0	node 1	inhibits	5.1
node 1	A	10	node 0	node 2	interacts	3.0
node 2	B	15	node 0	node 3	activates	5.2
node 3	B	5	node 2	node 3	interacts	9.9

11.5 Cytoscape Basics

Create a Cytoscape network from some basic R objects

```
nodes <- data.frame(id=c("node 0","node 1","node 2","node 3"),
                      group=c("A","A","B","B"), # categorical strings
                      score=as.integer(c(20,10,15,5)), # integers
                      stringsAsFactors=FALSE)
edges <- data.frame(source=c("node 0","node 0","node 0","node 2"),
                     target=c("node 1","node 2","node 3","node 3"),
                     interaction=c("inhibits","interacts","activates","interacts"), # optional
                     weight=c(5.1,3.0,5.2,9.9), # numeric
                     stringsAsFactors=FALSE)

createNetworkFromDataFrames(nodes,edges, title="my first network", collection="DataFrame Example")
```

```
## Loading data...
## Applying default style...
## Applying preferred layout...
```

```
## networkSUID
##          80
```

Remember. All networks we make are created in Cytoscape so get an image of the resulting network and include it in your current analysis if desired.

```
initial_network_png_file_name <- file.path(getwd(),"230_Isserlin_RCy3_intro", "images","initial_example")

if(file.exists(initial_network_png_file_name)){
  #cytoscape hangs waiting for user response if file already exists. Remove it first
  file.remove(initial_network_png_file_name)
}

#export the network
exportImage(initial_network_png_file_name, type = "png")
```

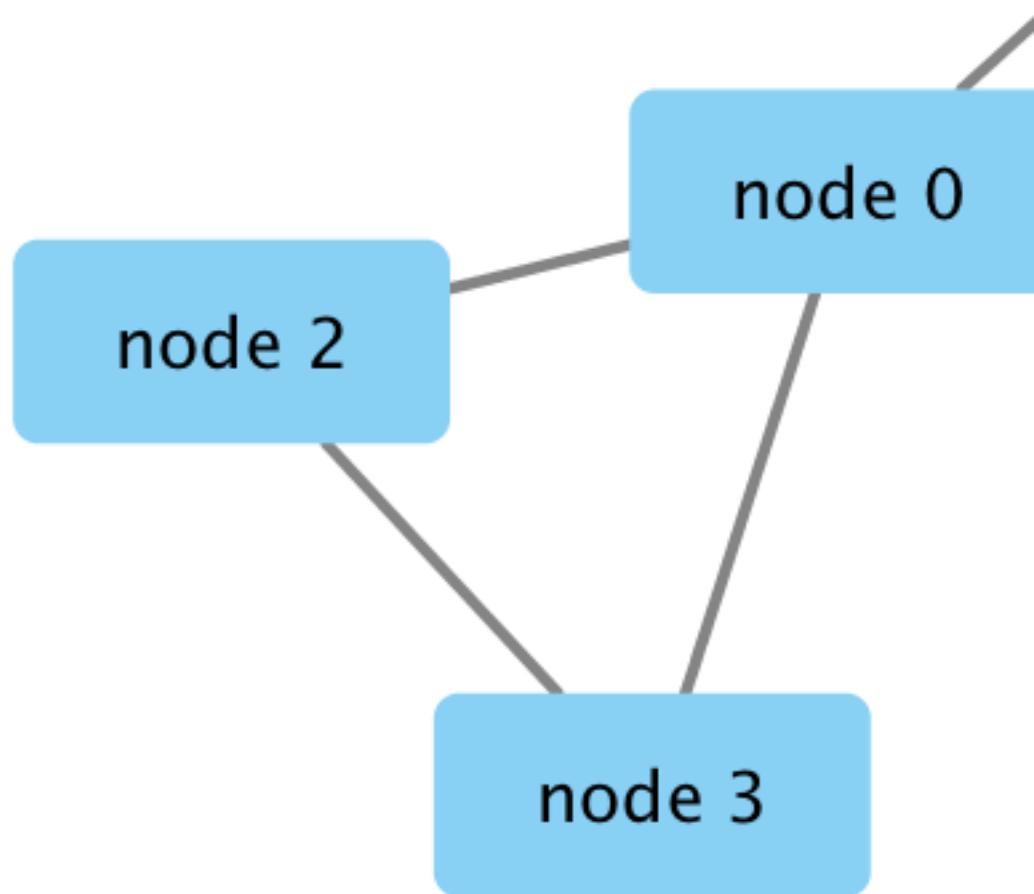


Figure 11.4: Example network created from dataframe

11.6 Example Data Set

We downloaded gene expression data from the Ovarian Serous Cystadenocarcinoma project of The Cancer Genome Atlas (TCGA)(International Cancer Genome et al. 2010), <http://cancergenome.nih.gov> via the Genomic Data Commons (GDC) portal(Grossman et al. 2016) on 2017-06-14 using TCGABiolinks R package(Colaprico et al. 2016). The data includes 300 samples available as RNA-seq data, with reads mapped to a reference genome using MapSplice(Wang et al. 2010) and read counts per transcript determined using the RSEM method(B. Li and Dewey 2011). RNA-seq data are labeled as ‘RNA-Seq V2’, see details at: <https://wiki.nci.nih.gov/display/TCGA/RNASeq+Version+2>). The RNA-SeqV2 data consists of raw counts similar to regular RNA-seq but RSEM (RNA-Seq by Expectation Maximization) data can be used with the edgeR method. The expression dataset of 300 tumours, with 79 classified as Immunoreactive, 72 classified as Mesenchymal, 69 classified as Differentiated, and 80 classified as Proliferative samples(class definitions were obtained from Verhaak et al.(Verhaak et al. 2013) Supplementary Table 1, third column). RNA-seq read counts were converted to CPM values and genes with CPM > 1 in at least 50 of the samples are retained for further study (50 is the minimal sample size in the classes). The data was normalized and differential expression was calculated for each cancer class relative to the rest of the samples.

There are two data files: 1. Expression matrix - containing the normalized expression for each gene across all 300 samples. 1. Gene ranks - containing the p-values, FDR and foldchange values for the 4 comparisons (mesenchymal vs rest, differential vs rest, proliferative vs rest and immunoreactive vs rest)

```
RNASeq_expression_matrix <- read.table(
  file.path(getwd(),"230_Isserlin_RCy3_intro","data","TCGA_OV_RNAseq_expression.txt"),
  header = TRUE, sep = "\t", quote="", stringsAsFactors = FALSE)

RNASeq_gene_scores <- read.table(
  file.path(getwd(),"230_Isserlin_RCy3_intro","data","TCGA_OV_RNAseq_All_edgeR_scores.txt"),
  header = TRUE, sep = "\t", quote="", stringsAsFactors = FALSE)
```

11.7 Finding Network Data

How do I represent *my* data as a network?

Unfortunately, there is not a simple answer. **It depends on your biological question!**

Example use cases:

1. Omics data - I have a *fill in the blank* (microarray, RNASeq, Proteomics, ATACseq, MicroRNA, GWAS ...) dataset. I have normalized and scored my data. How do I overlay my data on existing interaction data?
2. Coexpression data - I have a dataset that represents relationships. How do I represent it as a network.
3. Omics data - I have a *fill in the blank* (microarray, RNASeq, Proteomics, ATACseq, MicroRNA, GWAS ...) dataset. I have normalized and scored my data. I have run my data through a functional enrichment tool and now have a set of enriched terms associated with my dataset. How do I represent my functional enrichments as a network?

11.8 Use Case 1 - How are my top genes related?

Omics data - I have a *fill in the blank* (microarray, RNASeq, Proteomics, ATACseq, MicroRNA, GWAS ...) dataset. I have normalized and scored my data. How do I overlay my data on existing interaction data?

There are endless amounts of databases storing interaction data.



Figure 11.5: Info graphic of some of the available pathway databases

Thankfully we don't have to query each independent ally. In addition to many specialized (for example, for specific molecules, interaction type, or species) interaction databases there are also databases that collate these databases to create a broad resource that is easier to use. For example:

- StringApp - is a protein - protein and protein- chemical database that imports data from String(Szklarczyk et al. 2016) (which itself includes data from multiple species, coexpression, text-mining,existing databases, and genomic context), [STITCH] into a unified, queriable database.
- PSICQUIC(Aranda et al. 2011) - a REST-ful service that is the responsibility of the database provider to set up and maintain. PSICQUIC is an additional interface that allows users to search all available databases (that support this REST-ful service). The databases are required to represent their interaction data in Proteomic Standards Initiative - molecular interaction (PSI-MI) format. To see a list of all the available data source see here
- nDex(Pratt et al. 2017) - a network data exchange repository.
- GeneMANIA(Mostafavi et al. 2008) - contains multiple networks (shared domains, physical interactions, pathways, predicted, co-expression, genetic interactions and co-localized network). Given a set of genes GeneMANIA selects and weights networks that optimize the connectivity between the query genes. GeneMANIA will also return additional genes that are highly related to your query set.
- PathwayCommons - (access the data through the CyPath2App) is a pathway and interaction data source. Data is collated from a large set of resources (list here) and stored in the BioPAX(Demir et al. 2010) format. BioPAX is a data standard that allows for detailed representation of pathway mechanistic details as opposed to collapsing it to the set of interactions between molecules. BioPAX pathways from Pathway commons can also be loaded directly into R using the PaxToolsR(Luna et al. 2015) Bioconductor package.

Get a subset of genes of interest from our scored data:

```
top_mesenchymal_genes <- RNASeq_gene_scores[which(RNASeq_gene_scores$FDR.mesen < 0.05 & RNASeq_gene_scores$FDR.immuno < 0.05),]
head(top_mesenchymal_genes)
#>      Name geneid logFC.mesen logCPM.mesen LR.mesen PValue.mesen
#> 188 PRG4 10216    2.705305   2.6139056 95.58179  1.42e-22
#> 252 PROK1 84432    2.543381   1.3255202 68.31067  1.40e-16
#> 308 PRRX1 5396     2.077538   4.8570983 123.09925  1.33e-28
#> 434 PTGFR 5737     2.075707   -0.1960881 73.24646  1.14e-17
#> 438 PTGIS 5740     2.094198   5.8279714 165.11038  8.65e-38
#> 1214 BARX1 56033    3.267472   1.7427387 166.30064  4.76e-38
#>          FDR.mesen logFC.diff logCPM.diff LR.diff PValue.diff      FDR.diff
#> 188 7.34e-21 -1.2716521  2.6139056 14.107056 1.726950e-04 1.181751e-03
#> 252 4.77e-15  0.7455119  1.3255202 5.105528 2.384972e-02 6.549953e-02
#> 308 1.08e-26 -1.2367108  4.8570983 29.949104 4.440000e-08 1.250000e-06
#> 434 4.30e-16 -0.4233297 -0.1960881 2.318523 1.278413e-01 2.368387e-01
#> 438 1.20e-35 -0.4448761  5.8279714 5.696086 1.700278e-02 5.027140e-02
#> 1214 6.82e-36 -2.4411370  1.7427387 52.224346 4.950000e-13 6.970000e-11
#>          Row.names.y logFC.immuno logCPM.immuno LR.immuno PValue.immuno
#> 188 PRG4/10216   -0.4980017  2.6139056 2.651951  0.103422901
#> 252 PROK1/84432  -1.9692994  1.3255202 27.876348  0.000000129
#> 308 PRRX1/5396   -0.4914091  4.8570983 5.773502  0.016269586
#> 434 PTGFR/5737   -0.6737143 -0.1960881 6.289647  0.012144523
#> 438 PTGIS/5740   -0.6138074  5.8279714 11.708780  0.000622059
#> 1214 BARX1/56033 -0.6063633  1.7427387 4.577141  0.032401228
#>          FDR.immuno logFC.prolif logCPM.prolif LR.prolif PValue.prolif
#> 188 0.185548905 -0.9356510  2.6139056 8.9562066  0.0027652850
#> 252 0.000001650 -1.3195933  1.3255202 13.7675841  0.0002068750
#> 308 0.042062023 -0.3494185  4.8570983 2.9943819  0.0835537930
#> 434 0.033197211 -0.9786631 -0.1960881 12.9112727  0.0003266090
#> 438 0.002779895 -1.0355143  5.8279714 31.9162051  0.0000000161
```

```
#> 1214 0.073565295 -0.2199714 1.7427387 0.6315171 0.4267993850
#> FDR.prolif
#> 188 0.006372597
#> 252 0.000622057
#> 308 0.128528698
#> 434 0.000932503
#> 438 0.000000107
#> 1214 0.510856262
```

We are going to query the String Database to get all interactions found for our set of top Mesenchymal genes.

Reminder: to see the parameters required by the string function or to find the right string function you can use commandsHelp.

```
commandsHelp("help string")
```

```
## [1] "Available commands for 'string':"
```

```
## [1] "compound query"      "disease query"      "expand"
## [4] "filter enrichment"   "hide charts"       "hide images"
## [7] "list species"        "make string"       "protein query"
## [10] "pubmed query"        "retrieve enrichment" "settings"
## [13] "show charts"         "show enrichment"    "show images"
## [16] "version"
```

```
commandsHelp("help string protein query")
```

```
## [1] "Available arguments for 'string protein query':"
```

```
## [1] "cutoff"  "limit"   "query"   "species" "taxonID"
```

```
mesen_string_interaction_cmd <- paste('string protein query taxonID=9606 limit=150 cutoff=0.9 query=""', commandsGET(mesen_string_interaction_cmd))
```

```
## [1] "Loaded network 'String Network' with 262 nodes and 3628 edges"
```

Get a screenshot of the initial network

```
initial_string_network_png_file_name <- file.path(getwd(), "230_Isserlin_RCy3_intro", "images", "initial")
```

```

if(file.exists(initial_string_network_png_file_name)){
  #cytoscape hangs waiting for user response if file already exists. Remove it first
  response <- file.remove(initial_string_network_png_file_name)
}

response <- exportImage(initial_string_network_png_file_name, type = "png")

```

Layout the network

```
layoutNetwork('force-directed')
```

Check what other layout algorithms are available to try out

```
getLayoutNames()
```

## [1] "spherical"	"degree-circle"
## [3] "box"	"attributes-layout"
## [5] "kamada-kawai"	"force-directed"
## [7] "grid3D"	"cose"
## [9] "flatten"	"hierarchical"
## [11] "center3d"	"attribute-circle"
## [13] "fruchterman-rheingold-3D"	"stacked-node-layout"
## [15] "circular"	"genemania-force-directed"
## [17] "grid"	"fruchterman-rheingold"
## [19] "isom"	"force-directed-cl"

Get the parameters for a specific layout

```
getLayoutPropertyNames(layout.name='force-directed')
```

## [1] "numIterations"	"defaultSpringCoefficient"
## [3] "defaultSpringLength"	"defaultNodeMass"
## [5] "isDeterministic"	"singlePartition"

Re-layout the network using the force directed layout but specify some of the parameters

```
layoutNetwork('force-directed defaultSpringCoefficient=0.0000008 defaultSpringLength=70')
```

Get a screenshot of the re-laid out network

```

relayout_string_network_png_file_name <- file.path(getwd(),"230_Isserlin_RCy3_intro", "images","relayout")

if(file.exists(relayout_string_network_png_file_name)){
  #cytoscape hangs waiting for user response if file already exists. Remove it first
  response<- file.remove(relayout_string_network_png_file_name)
}

response <- exportImage(relayout_string_network_png_file_name, type = "png")

```



Figure 11.6: Initial network returned by String from our set of Mesenchymal query genes

Overlay our expression data on the String network.

To do this we will be using the loadTableData function from RCy3. It is important to make sure that your identifiers types match up. You can check what is used by String by pulling in the column names of the node attribute table.

```
getTableName('node')
```

[1] "SUID"	"shared name"
[4] "selected"	"canonical name"
[7] "full name"	"database identifier"
[10] "@id"	"namespace"
[13] "query term"	"sequence"
[16] "STRING style"	"enhancedLabel Passthrough"
[19] "compartment chloroplast"	"compartment cytoskeleton"
[22] "compartment endoplasmic reticulum"	"compartment endosome"
[25] "compartment golgi apparatus"	"compartment lysosome"
[28] "compartment nucleus"	"compartment peroxisome"
[31] "compartment vacuole"	"image"
[34] "target family"	"tissue adrenal gland"
[37] "tissue bone"	"tissue bone marrow"
[40] "tissue gall bladder"	"tissue heart"
[43] "tissue kidney"	"tissue liver"
[46] "tissue lymph node"	"tissue muscle"
[49] "tissue pancreas"	"tissue saliva"
[52] "tissue spleen"	"tissue stomach"
[55] "tissue urine"	

If you are unsure of what each column is and want to further verify the column to use you can also pull in the entire node attribute table.

```
node_attribute_table_topmesen <- getTableColumns(table="node")
head(node_attribute_table_topmesen[,3:7])
```



Figure 11.7: Initial network returned by String from our set of Mesenchymal query genes

```

##          name selected canonical name display name ful
## 134 9606.ENSP00000223095 FALSE      P05121 SERPINE1
## 135 9606.ENSP00000262487 FALSE      B1AKI9 ISM1
## 136 9606.ENSP00000337065 FALSE      O95715 CXCL14
## 137 9606.ENSP00000295137 FALSE      P63267 ACTG2
## 138 9606.ENSP00000334122 FALSE      P11487 FGF3
## 139 9606.ENSP00000267843 FALSE      P21781 FGF7

```

The column “display name” contains HGNC gene names which are also found in our Ovarian Cancer dataset.

To import our expression data we will match our dataset to the “display name” node attribute.

```
?loadTableData
```

```
loadTableData(RNASeq_gene_scores,table.key.column = "display name",data.key.column = "Name") #default
```

```
## [1] "Success: Data loaded in defaultnode table"
```

Modify the Visual Style Create your own visual style to visualize your expression data on the String network.

Start with a default style

```

style.name = "MesenchymalStyle"
defaults.list <- list(NODE_SHAPE="ellipse",
                      NODE_SIZE=60,
                      NODE_FILL_COLOR="#AAAAAA",
                      EDGE_TRANSPARENCY=120)
node.label.map <- mapVisualProperty('node label','display name','p') # p for passthrough; nothing else
createVisualStyle(style.name, defaults.list, list(node.label.map))
setVisualStyle(style.name=style.name)

```

```

##          message
## "Visual Style applied."
```

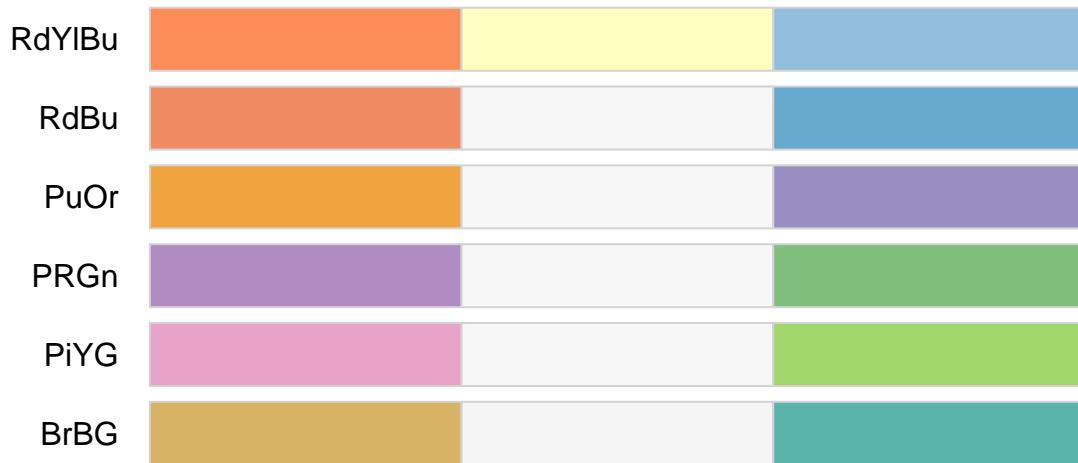
Update your created style with a mapping for the Mesenchymal logFC expression. The first step is to grab the column data from Cytoscape (we can reuse the node_attribute table concept from above but we have to call the function again as we have since added our expression data) and pull out the min and max to define our data mapping range of values.

Note: you could define the min and max based on the entire dataset or just the subset that is represented in Cytoscape currently. The two methods will give you different results. If you intend on comparing different networks created with the same dataset then it is best to calculate the min and max from the entire dataset as opposed to a subset.

```
min.mesen.logfc = min(RNASeq_gene_scores$logFC.mesen, na.rm=TRUE)
max.mesen.logfc = max(RNASeq_gene_scores$logFC.mesen, na.rm=TRUE)
data.values = c(min.mesen.logfc, 0, max.mesen.logfc)
```

Next, we use the RColorBrewer package to help us pick good colors to pair with our data values.

```
library(RColorBrewer)
display.brewer.all(length(data.values), colorblindFriendly=TRUE, type="div") # div, qual, seq, all
```



```
node.colors <- c(rev(brewer.pal(length(data.values), "RdBu")))
```

Map the colors to our data value and update our visual style.

```
setNodeColorMapping("logFC.mesen", data.values, node.colors, style.name=style.name)
```

Remember, String includes your query proteins as well as other proteins that associate with your query proteins (including the strongest connection first). Not all of the proteins in this network are your top hits. How can we visualize which proteins are our top Mesenchymal hits?

Add a different border color or change the node shape for our top hits.

```
getNodeShapes()
```

```
#select the Nodes of interest
#selectNode(nodes = top_mesenchymal_genes$name, by.col="display name")
setNodeShapeBypass(node.names = top_mesenchymal_genes$name, new.shapes = "TRIANGLE")
```

## [1] "ROUND_RECTANGLE" "ELLIPSE"	"OCTAGON"	"DIAMOND"
## [5] "PARALLELOGRAM" "HEXAGON"	"TRIANGLE"	"RECTANGLE"
## [9] "VEE"		

Change the size of the node to be correlated with the Mesenchymal p-value.

```
setNodeSizeMapping(table.column = 'LR.mesen',
                   table.column.values = c(min(RNASeq_gene_scores$LR.mesen),
                                           mean(RNASeq_gene_scores$LR.mesen),
                                           max(RNASeq_gene_scores$LR.mesen)),
                   sizes = c(30, 60, 150), mapping.type = "c", style.name = style.name)
```

Get a screenshot of the resulting network

```
mesen_string_network_png_file_name <- file.path(getwd(),"230_Isserlin_RCy3_intro", "images","mesen_string_network.png")

if(file.exists(mesen_string_network_png_file_name)){
  #cytoscape hangs waiting for user response if file already exists. Remove it first
  response<- file.remove(mesen_string_network_png_file_name)
}
response <- exportImage(mesen_string_network_png_file_name, type = "png")
```

11.9 Use Case 2 - Which genes have similar expression.

Instead of querying existing resources look for correlations in your own dataset to find out which genes have similar expression. There are many tools that can analyze your data for correlation. A popular tool is Weighted Gene Correlation Network Analysis (WGCNA)(Langfelder and Horvath 2008) which takes expression data and calculates functional modules. As a simple example we can transform our expression dataset into a correlation matrix.

Using the Cytoscape App, aMatReader(Settle et al. 2018), we transform our adjacency matrix into an interaction network. First we filter the correlation matrix to contain only the strongest connections (for example, only correlations greater than 0.9).

```
RNASeq_expression <- RNASeq_expression_matrix[,3:ncol(RNASeq_expression_matrix)]

rownames(RNASeq_expression) <- RNASeq_expression_matrix$Name
RNAseq_correlation_matrix <- cor(t(RNASeq_expression), method="pearson")

#set the diagonal of matrix to zero - eliminate self-correlation
RNAseq_correlation_matrix[
  row(RNAseq_correlation_matrix) == col(RNAseq_correlation_matrix) ] <- 0

# set all correlations that are less than 0.9 to zero
RNAseq_correlation_matrix[which(RNAseq_correlation_matrix<0.90)] <- 0

#get rid of rows and columns that have no correlations with the above thresholds
RNAseq_correlation_matrix <- RNAseq_correlation_matrix[which(rowSums(RNAseq_correlation_matrix) != 0),
                                                       which(colSums(RNAseq_correlation_matrix) !=0)] 

#write out the correlation file
correlation_filename <- file.path(getwd(), "230_Isserlin_RCy3_intro", "data", "TCGA_OV_RNAseq_expression")
write.table( RNAseq_correlation_matrix, file = correlation_filename, col.names = TRUE, row.names = FALSE)
```

Use the CyRest call to access the aMatReader functionality.

```
amat_url <- "aMatReader/v1/import"
amat_params = list(files = list(correlation_filename),
                   delimiter = "TAB",
                   undirected = FALSE,
                   ignoreZeros = TRUE,
                   interactionName = "correlated with",
                   rowNames = FALSE
                  )

response <- cyrestPOST(operation = amat_url, body = amat_params, base.url = "http://localhost:1234")
```

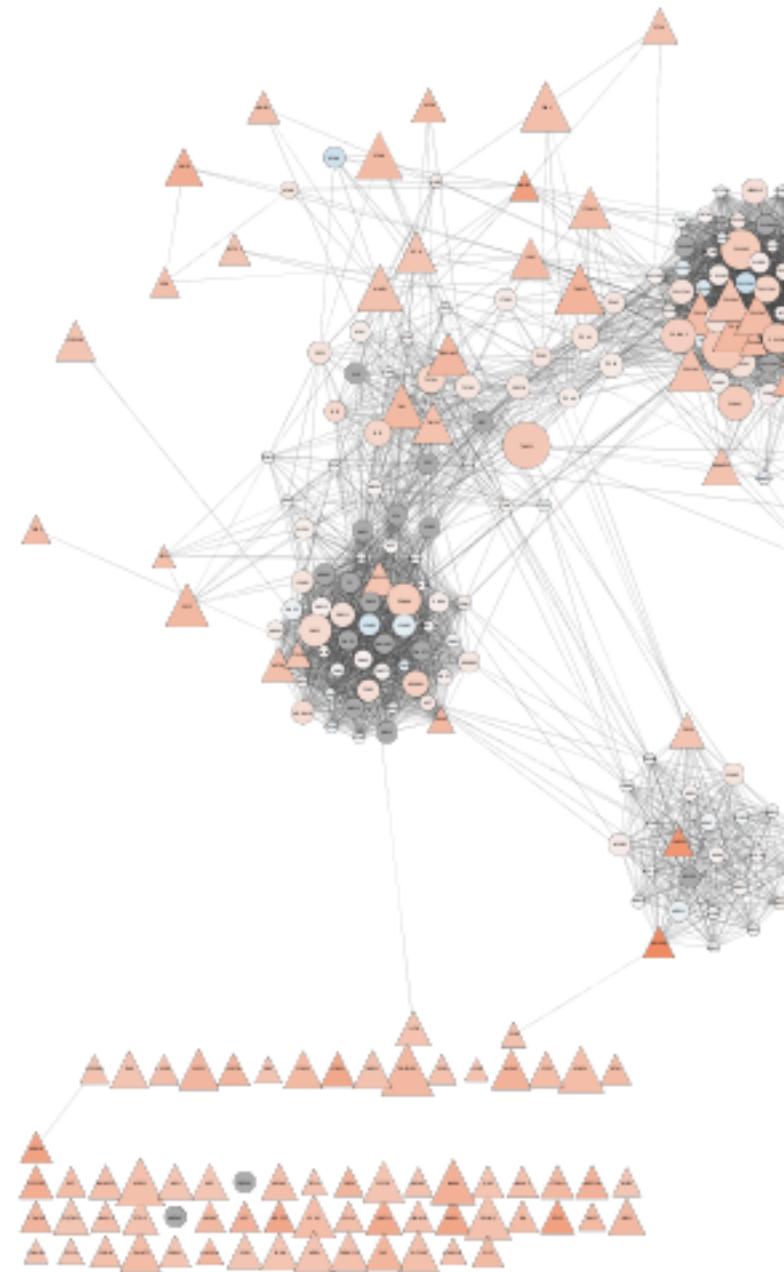


Figure 11.8: Formatted String network from our set of Mesenchymal query genes. Annotated with our expressin data

```
current_network_id <- response$data["suid"]

#relayout network
layoutNetwork('cose',
              network = as.numeric(current_network_id))

renameNetwork(title = "Coexpression_network_pear0_95",
              network = as.numeric(current_network_id))
```

Modify the visualization to see where each genes is predominantly expressed. Look at the 4 different p-values associated with each gene and color the nodes with the type associated with the lowest FDR.

Load in the scoring data. Specify the cancer type where the genes has the lowest FDR value.

```
nodes_in_network <- rownames(RNAseq_correlation_matrix)

#add an additional column to the gene scores table to indicate in which samples
# the gene is significant
node_class <- vector(length = length(nodes_in_network), mode = "character")
for(i in 1:length(nodes_in_network)){
  current_row <- which(RNASeq_gene_scores>Name == nodes_in_network[i])
  min_pvalue <- min(RNASeq_gene_scores[current_row,
                                         grep(colnames(RNASeq_gene_scores), pattern = "FDR")])
  if(RNASeq_gene_scores$FDR.mesen[current_row] <=min_pvalue){
    node_class[i] <- paste(node_class[i],"mesen",sep = " ")
  }
  if(RNASeq_gene_scores$FDR.diff[current_row] <=min_pvalue){
    node_class[i] <- paste(node_class[i],"diff",sep = " ")
  }
  if(RNASeq_gene_scores$FDR.prolif[current_row] <=min_pvalue){
    node_class[i] <- paste(node_class[i],"prolif",sep = " ")
  }
  if(RNASeq_gene_scores$FDR.immuno[current_row] <=min_pvalue){
    node_class[i] <- paste(node_class[i],"immuno",sep = " ")
  }
}
node_class <- trimws(node_class)
node_class_df <- data.frame(name=nodes_in_network, node_class, stringsAsFactors = FALSE)

head(node_class_df)
#>      name node_class
#> 1  ABCA6      mesen
#> 2  ABCA8      mesen
#> 3   ABI3      immuno
#> 4   ACAN      prolif
#> 5  ACAP1      immuno
#> 6 ADAM12      mesen
```

Map the new node attribute and the all the gene scores to the network.

```
loadTableData(RNASeq_gene_scores,table.key.column = "name",data.key.column = "Name") #default data.frame

loadTableData(node_class_df,table.key.column = "name",data.key.column = "name") #default data.frame key
```

Create a color mapping for the different cancer types.

```

# create a new mapping with the different types
unique_types <- sort(unique(node_class))

coul = brewer.pal(4, "Set1")

# I can add more tones to this palette :
coul = colorRampPalette(coul)(length(unique_types))

setNodeColorMapping(table.column = "node_class",table.column.values = unique_types,
                    colors = coul,mapping.type = "d")

correlation_network_png_file_name <- file.path(getwd(),"230_Isserlin_RCy3_intro", "images","correlation"

if(file.exists(correlation_network_png_file_name)){
  #cytoscape hangs waiting for user response if file already exists. Remove it first
  file.remove(correlation_network_png_file_name)
}

#export the network
exportImage(correlation_network_png_file_name, type = "png")

```

cluster the Network

```

#make sure it is set to the right network
setCurrentNetwork(network = getNetworkName(suid=as.numeric(current_network_id)))

#cluster the network
clustermaker_url <- paste("cluster mcl network=SUID:",current_network_id, sep="")
commandsGET(clustermaker_url)

#get the clustering results
default_node_table <- getTableColumns(table= "node",network = as.numeric(current_network_id))

head(default_node_table)

```

Perform pathway Enrichment on one of the clusters using g:Profiler (Reimand et al. 2016). g:Profiler is an online functional enrichment web service that will take your gene list and return the set of enriched pathways. For automated analysis g:Profiler has created an R library to interact with it directly from R instead of using the web page.

Create a function to call g:Profiler and convert the returned results into a generic enrichment map input file.

```

tryCatch(expr = { library("gProfileR")},
        error = function(e) { install.packages("gProfileR")}, finally = library("gProfileR"))

#function to run gprofiler using the gprofiler library
#
# The function takes the returned gprofiler results and formats it to the generic EM input file
#
# function returns a data frame in the generic EM file format.
runGprofiler <- function(genes,current_organism = "hsapiens",
                           significant_only = F, set_size_max = 200,
                           set_size_min = 3, filter_gs_size_min = 5 , exclude_iea = F){

  gprofiler_results <- gprofiler(genes ,
                                 significant=significant_only,ordered_query = F,

```

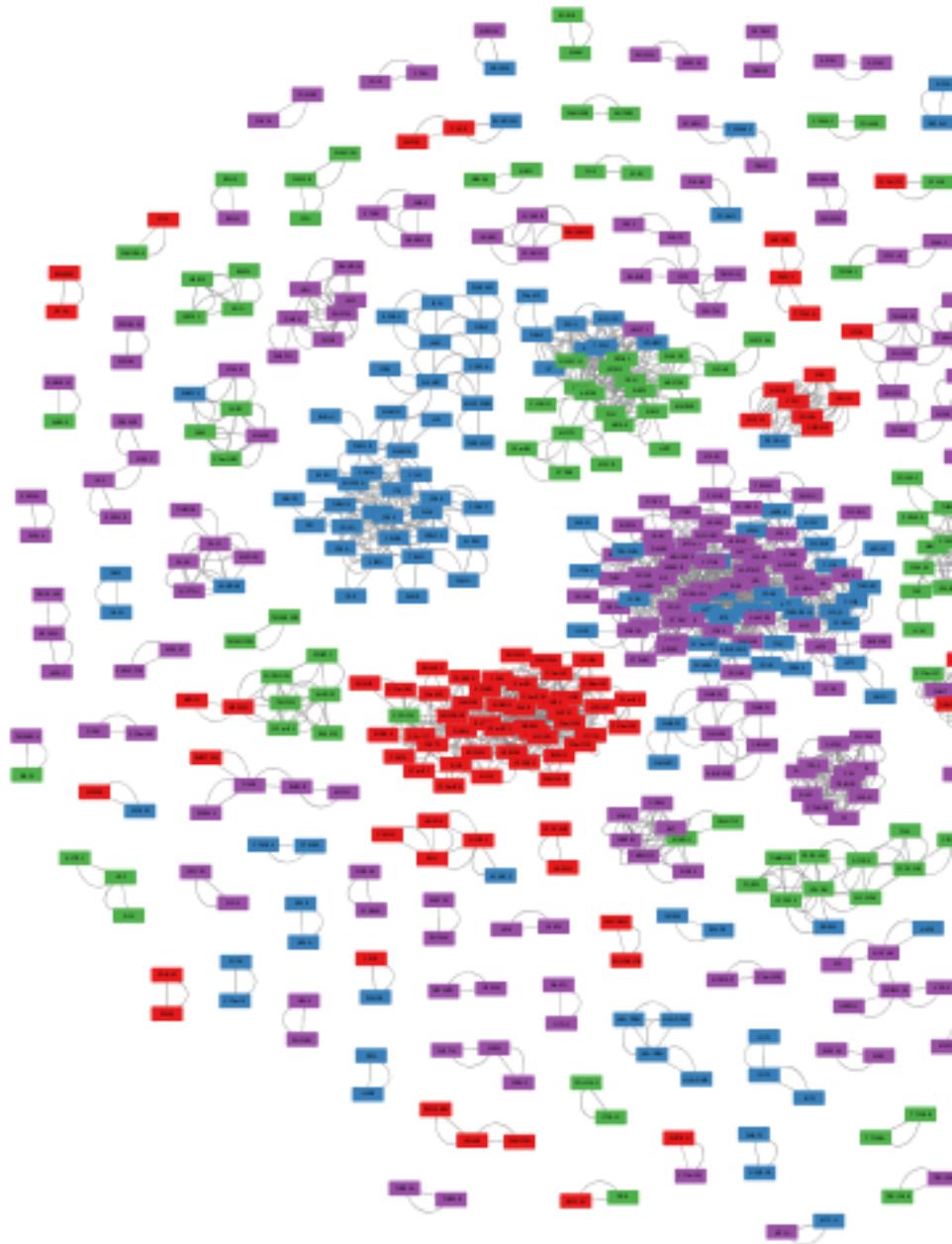


Figure 11.9: Example correlation network created using aMatReader

```

        exclude_iea=exclude_iea,max_set_size = set_size_max,
        min_set_size = set_size_min,
        correction_method = "fdr",
        organism = current_organism,
        src_filter = c("GO:BP","REAC"))

#filter results
gprofiler_results <- gprofiler_results[which(gprofiler_results[, 'term.size'] >= 3
                                              & gprofiler_results[, 'overlap.size'] >= filter_gs_size_min ),]

# gProfileR returns corrected p-values only. Set p-value to corrected p-value
if(dim(gprofiler_results)[1] > 0){
  em_results <- cbind(gprofiler_results[,,
                                         c("term.id","term.name","p.value","p.value")], 1,
                       gprofiler_results[, "intersection"])
  colnames(em_results) <- c("Name", "Description", "pvalue", "qvalue", "phenotype", "genes")

  return(em_results)
} else {
  return("no gprofiler results for supplied query")
}
}
}

```

Run g:Profiler. g:Profiler will return a set of pathways and functions that are found to be enriched in our query set of genes.

```

current_cluster <- "1"
#select all the nodes in cluster 1
selectednodes <- selectNodes(current_cluster, by.col="__mclCluster")

#create a subnetwork with cluster 1
subnetwork_suid <- createSubnetwork(nodes="selected")

renameNetwork("Cluster1_Subnetwork", network=as.numeric(subnetwork_suid))

subnetwork_node_table <- getTableColumns(table= "node", network = as.numeric(subnetwork_suid))

em_results <- runGprofiler(subnetwork_node_table$name)

#write out the g:Profiler results
em_results_filename <- file.path(getwd(),
                                  "230_Isserlin_RCy3_intro", "data", paste("gprofiler_cluster", current_cluster

write.table(em_results, em_results_filename, col.names=TRUE, sep="\t", row.names=FALSE, quote=FALSE)

head(em_results)

```

	Name <chr>	Description <chr>
29	GO:0038094	Fc-gamma receptor signaling pathway
30	GO:0002431	Fc receptor mediated stimulatory signaling pathway
99	GO:0097529	myeloid leukocyte migration
100	GO:0097530	granulocyte migration
101	GO:1990266	neutrophil migration
102	GO:0071621	granulocyte chemotaxis

6 rows | 1–6 of 6 columns

Create an enrichment map with the returned g:Profiler results. An enrichment map is a different sort of network. Instead of nodes representing genes, nodes represent pathways or functions. Edges between these pathways or functions represent shared genes or pathway crosstalk. An enrichment map is a way to visualize your enrichment results to help reduce redundancy and uncover main themes. Pathways can also be explored in detail using the features available through the App in Cytoscape.

```
em_command = paste('enrichmentmap build analysisType="generic" ',
  'pvalue=', "0.05", 'qvalue=', "0.05",
  'similaritycutoff=', "0.25",
  'coefficients=', "JACCARD",
  'enrichmentsDataset1=' , em_results_filename ,
  sep=" ")
```

#enrichment map command will return the uid of newly created network.

```
em_network_suid <- commandsRun(em_command)
```

```
renameNetwork("Cluster1_enrichmentmap", network=as.numeric(em_network_suid))
```

Export image of resulting Enrichment map.

```
cluster1em_png_file_name <- file.path(getwd(),"230_Isserlin_RCy3_intro", "images","cluster1em.png")
```

```
if(file.exists(cluster1em_png_file_name)){
  #cytoscape hangs waiting for user response if file already exists. Remove it first
  file.remove(cluster1em_png_file_name)
}
```

```
#export the network
exportImage(cluster1em_png_file_name, type = "png")
```

Annotate the Enrichment map to get the general themes that are found in the enrichment results of cluster 1

```
#get the column from the nodetable and node table
nodetable_colnames <- getTableColumnNames(table="node", network = as.numeric(em_network_suid))
```

```
descr_attrib <- nodetable_colnames[grep(nodetable_colnames, pattern = "_GS_DESCR")]
```



Figure 11.10: Example Enrichment Map created when running an enrichment analysis using g:Profiler with the genes that are part of cluster 1

```
#Autoannotate the network
autoannotate_url <- paste("autoannotate annotate-clusterBoosted labelColumn=", descr_attrib, " maxWord")
current_name <- commandsGET(autoannotate_url)
```

Export image of resulting Annotated Enrichment map.

```
cluster1em_annot_png_file_name <- file.path(getwd(),"230_Isserlin_RCy3_intro", "images","cluster1em_annot.png")

if(file.exists(cluster1em_annot_png_file_name)){
  #cytoscape hangs waiting for user response if file already exists. Remove it first
  file.remove(cluster1em_annot_png_file_name)
}

#export the network
exportImage(cluster1em_annot_png_file_name, type = "png")
```

Dense networks small or large never look like network figures we so often see in journals. A lot of manual tweaking, reorganization and optimization is involved in getting that perfect figure ready network. The above network is what the network starts as. The below figure is what it can look like after a few minutes of manual reorganization. (individual clusters were selected from the auto annotate panel and separated from other clusters)

11.10 Use Case 3 - Functional Enrichment of Omics set.

Reducing our large OMICs expression set to a simple list of genes eliminates the majority of the signals present in the data. Thresholding will only highlight the strong signals neglecting the often more interesting subtle signals. In order to capitalize on the wealth of data present in the data we need to perform pathway enrichment analysis on the entire expression set. There are many tools in R or as standalone apps that perform this type of analysis.

To demonstrate how you can use Cytoscape and RCy3 in your enrichment analysis pipeline we will use EnrichmentBrowser package (as demonstrated in detail in the workshop Functional enrichment analysis of high-throughput omics data) to perform pathway analysis.

```
if(!"EnrichmentBrowser" %in% installed.packages()){
  install.packages("BiocManager")
  BiocManager::install("EnrichmentBrowser")
}

suppressPackageStartupMessages(library(EnrichmentBrowser))
```

Download the latest pathway definition file from the Baderlab download site. Baderlab genesets are updated on a monthly basis. Detailed information about the sources can be found here. Only Human, Mouse and Rat gene set files are currently available on the Baderlab downloads site. If you are working with a species other than human (and it is either rat or mouse) change the gmt_url below to correct species.

```
tryCatch(expr = { suppressPackageStartupMessages(library("RCurl"))},
        error = function(e) { install.packages("RCurl")},
        finally = library("RCurl"))

gmt_url = "http://download.baderlab.org/EM_Genesets/current_release/Human/symbol/"

#list all the files on the server
```

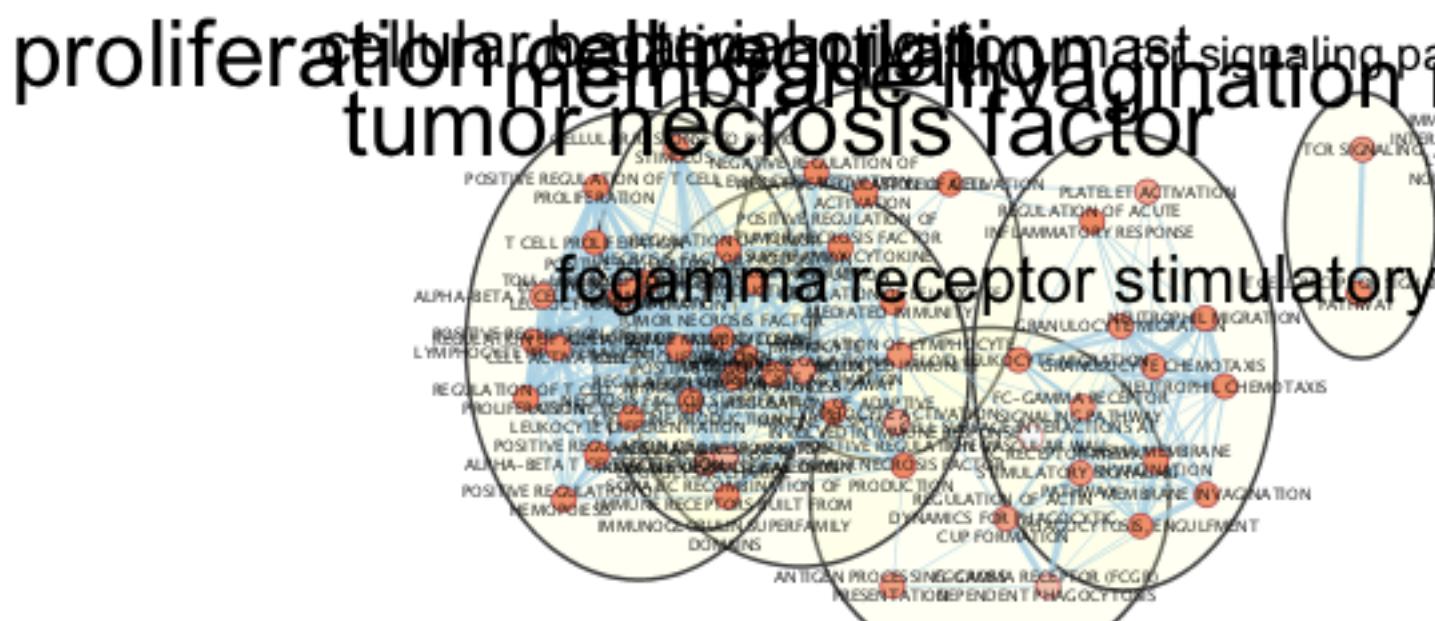


Figure 11.11: Example Annotated Enrichment Map created when running an enrichment analysis using g:Profiler with the genes that are part of cluster 1

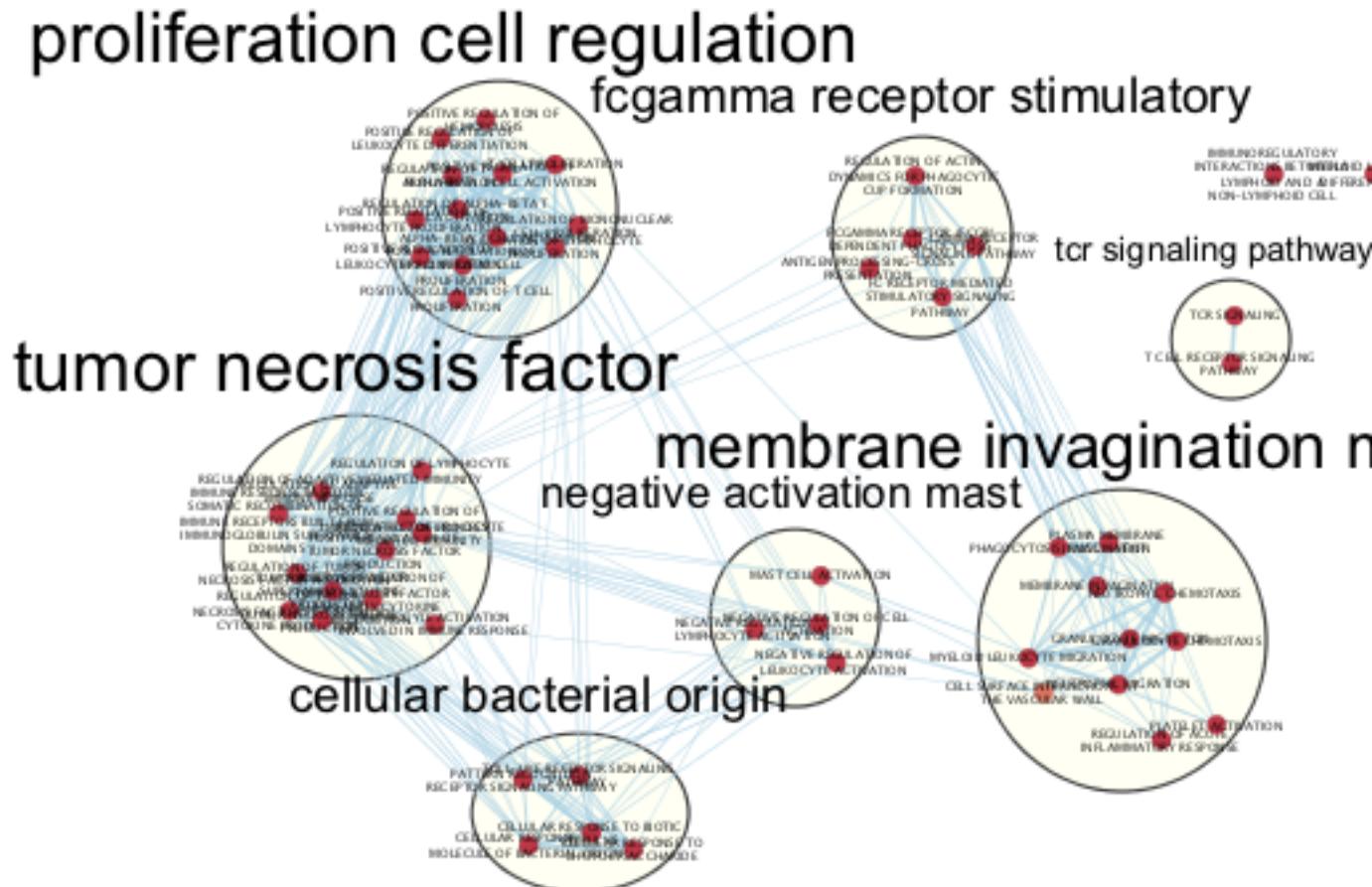


Figure 11.12: Example Annotated Enrichment Map created when running an enrichment analysis using g:Profiler with the genes that are part of cluster 1 after manual adjusting to generate a cleaner figure

```

filenames = getURL(gmt_url)
tc = textConnection(filenames)
contents = suppressWarnings(readLines(tc))
close(tc)

#get the gmt that has all the pathways and does not include terms inferred from electronic annotations.
#start with gmt file that has pathways only
rx = gregexpr("(?=<<a href=\"(.*.GOBP_AllPathways_no_GO_iea.*).gmt)(?=\\>)",
  contents, perl = TRUE)
gmt_file = unlist(regmatches(contents, rx))

dest_gmt_file <- file.path(getwd(), "230_Isserlin_RCy3_intro", "data", gmt_file)

download.file(
  paste(gmt_url,gmt_file,sep=""),
  destfile=dest_gmt_file
)

```

Load in the gmt file

```

baderlab.gs <- getGenesets(dest_gmt_file)
baderlab.gs[1:2]
#> $`THIO-MOLYBDENUM COFACTOR BIOSYNTHESIS%HUMANCYC%PWF-5963`
#> [1] "MOCOS"
#>
#> $`PROLINE BIOSYNTHESIS I%HUMANCYC%PROSYN-PWF`
#> [1] "PYCR2"      "ALDH18A1"   "PYCR1"      "PYCRL"

```

Create the dataset required by EnrichmentBrowser tools

```

#create the expression file - A tab separated text file containing expression values. Columns = samples
expr <- RNASeq_expression

sumexpr_filename <- file.path(getwd(), "230_Isserlin_RCy3_intro", "data", "SummarizeExperiment_expression")
write.table( expr ,  file = sumexpr_filename , col.names = FALSE, row.names = FALSE, sep = "\t", quote = FALSE)

rowData <- RNASeq_gene_scores[,grep(colnames(RNASeq_gene_scores), pattern="mesen")]
rowData <- cbind(RNASeq_gene_scores$Name, rowData)
colnames(rowData)[2] <- "FC"
colnames(rowData)[6] <- "ADJ.PVAL"

sumexpr_rdat_filename <- file.path(getwd(), "230_Isserlin_RCy3_intro", "data", "SummarizeExperiment_rdat")
write.table( rowData[,1] ,  file = sumexpr_rdat_filename , col.names = FALSE, row.names = FALSE, sep = "\t", quote = FALSE)

#load in the data classification data
# A tab separated text file containing annotation information for the samples in either *two or three*
classDefinitions_RNASeq <- read.table(
  file.path(getwd(), "230_Isserlin_RCy3_intro", "data", "RNASeq_classdefinitions.txt"), header = TRUE, sep = "\t")

colData <- data.frame(Sample = colnames(RNASeq_expression),
                      GROUP = classDefinitions_RNASeq$SUBTYPE,
                      stringsAsFactors = FALSE)
rownames(colData) <- colnames(RNASeq_expression)
colData$GROUP[which(colData$GROUP != "Mesenchymal")] <- 0

```

```

colData$GROUP[which(colData$GROUP == "Mesenchymal")] <- 1

sumexpr_cdat_filename <- file.path(getwd(), "230_Isserlin_RCy3_intro","data","SummarizeExperiment_cdat.r")
write.table( colData ,  file = sumexpr_cdat_filename , col.names = FALSE, row.names = FALSE, sep = "\t")

#create the Summarize Experiment object
se_OV <- EnrichmentBrowser::readSE(assay.file = sumexpr_filename , cdat.file = sumexpr_cdat_filename, r)

```

Put our precomputed p-values and fold change values into the Summarized Experiment object so we can use our rankings for the analysis

```

#set the Summarized Experiment to our computed p-values and FC
rowData(se_OV) <- rowData

```

Run basic Over representation analysis (ORA) using our ranked genes and our gene set file downloaded from the Baderlab genesets.

```

ora.all <- sbea(method="ora", se=se_OV, gs=baderlab.gs, perm=0, alpha=0.05)
gsRanking(ora.all)

#> DataFrame with 967 rows and 4 columns
#>
#> GENE SET
#> 1 EXTRACELLULAR MATRIX ORGANIZATION%GOBP%GO:0030198
#> 2 HALLMARK_EPITHELIAL_MESENCHYMAL_TRANSITION%MSIGDB_C2%HALLMARK_EPITHELIAL_MESENCHYMAL_TRANSITION
#> 3 EXTRACELLULAR STRUCTURE ORGANIZATION%GOBP%GO:0043062
#> 4 EXTRACELLULAR MATRIX ORGANIZATION%REACTOME%R-HSA-1474244.2
#> 5 NABA_CORE_MATRISOME%MSIGDB_C2%NABA_CORE_MATRISOME
#> ...
#> 963 OVULATION CYCLE%GOBP%GO:0042698
#> 964 NEGATIVE REGULATION OF TRANSPORT%GOBP%GO:0051051
#> 965 PEPTIDYL-TYROSINE MODIFICATION%GOBP%GO:0018212
#> 966 NEUTROPHIL DEGRANULATION%GOBP%GO:0043312
#> 967 PROCESSING OF CAPPED INTRON-CONTAINING PRE-MRNA%REACTOME DATABASE ID RELEASE 65%72203
#> NR.GENES NR.SIG.GENES P.VALUE
#> <numeric> <numeric> <numeric>
#> 1 195 147 6.42e-21
#> 2 193 145 2.07e-20
#> 3 223 160 5.43e-19
#> 4 239 163 6.1e-16
#> 5 215 145 1.03e-13
#> ...
#> 963 ... ...
#> 964 13 9 0.0487
#> 965 246 118 0.0488
#> 966 76 40 0.049
#> 967 416 194 0.0496
#> 968 213 103 0.0499

```

DataFrame with 967 rows and 4 columns

1			EXTRACELLULAR MATRIX ORGANIZATION
2	HALLMARK_EPITHELIAL_MESENCHYMAL_TRANSITION	%SIGDB_C2%	HALLMARK_EPITHELIAL_MESENCHYMAL_TRANSITION
3			EXTRACELLULAR STRUCTURE ORGANIZATION
4			EXTRACELLULAR MATRIX ORGANIZATION
5			NABA_CORE_MATRISOME
...			%SIGDB_C2%
963			OVULATION
964			NEGATIVE REGULATION OF
965			PEPTIDYL-TYROSINE MODIFICATION
966			NEUTROPHIL DEGRADATION
967	PROCESSING_OF_CAPPED_INTRON-CONTAINING_PRE-MRNA	%REACTOME	DATUM

Take the enrichment results and create a generic enrichment map input file so we can create an Enrichment map. Description of format of the generic input file can be found here and example generic enrichment map files can be found here

```
#manually adjust p-values
ora.all$res.tbl <- cbind(ora.all$res.tbl, p.adjust(ora.all$res.tbl$P.VALUE, "BH"))
colnames(ora.all$res.tbl)[ncol(ora.all$res.tbl)] <- "Q.VALUE"

#create a generic enrichment map file
em_results_mesen <- data.frame(name = ora.all$res.tbl$GENE.SET, descr = ora.all$res.tbl$GENE.SET,
                                 pvalue=ora.all$res.tbl$P.VALUE, qvalue=ora.all$res.tbl$Q.VALUE, stringsAsFactors=FALSE)

#write out the ora results
em_results_mesen_filename <- file.path(getwd(),
                                         "230_Isserlin_RCy3_intro","data","mesen_ora_enr_results.txt")

write.table(em_results_mesen, em_results_mesen_filename, col.names=TRUE, sep="\t", row.names=FALSE, quote=FALSE)
```

Create an enrichment map with the returned ORA results.

```
em_command = paste('enrichmentmap build analysisType="generic" ', "gmtFile=", dest_gmt_file,
                   'pvalue=',"0.05", 'qvalue=',"0.05",
                   'similaritycutoff=',"0.25",
                   'coefficients=',"JACCARD",
                   'enrichmentsDataset1=' ,em_results_mesen_filename ,
                   sep=" ")  
  

#enrichment map command will return the suid of newly created network.
em_mesen_network_suid <- commandsRun(em_command)
```

```
renameNetwork("Mesenchymal_ORA_enrichmentmap", network=as.numeric(em_mesen_network_suid))
```

Export image of resulting Enrichment map.

```
mesenem_png_file_name <- file.path(getwd(), "230_Isserlin_RCy3_intro", "images", "mesenem.png")

if(file.exists(mesenem_png_file_name)){
  #cytoscape hangs waiting for user response if file already exists. Remove it first
  file.remove(mesenem_png_file_name)
}

#export the network
exportImage(mesenem_png_file_name, type = "png")
```

Annotate the Enrichment map to get the general themes that are found in the enrichment results. Often for very busy networks annotating the networks doesn't help to reduce the complexity but instead adds to it. To get rid of some of the pathway redundancy and density in the network create a summary of the underlying network. The summary network collapses each cluster to a summary node. Each summary node is annotated with a word tag (the top 3 words associated with the nodes of the cluster) that is computed using the Wordcloud app.

```
#get the column from the nodetable and node table
nodetable_colnames <- getTableColumnNames(table="node", network = as.numeric(em_mesen_network_suid))

descr_attrib <- nodetable_colnames[grep(nodetable_colnames, pattern = "_GS_DESCR")]

#Autoannotate the network
autoannotate_url <- paste("autoannotate annotate-clusterBoosted labelColumn=", descr_attrib, " maxWordCount=3")
current_name <- commandsGET(autoannotate_url)

#create a summary network
commandsGET("autoannotate summary network='current'")

#change the network name
summary_network_suid <- getNetworkSuid()

renameNetwork(title = "Mesen_ORA_summary_netowrk",
              network = as.numeric(summary_network_suid))

#get the summary node names
summary_nodes <- getTableColumns(table="node", columns=c("name"))

#clear bypass style the summary network has
clearNodePropertyBypass(node.names = summary_nodes$name, visual.property = "NODE_SIZE")

#relayout network
layoutNetwork('cose',
              network = as.numeric(summary_network_suid))
```

Export image of resulting Summarized Annotated Enrichment map.

```
mesenem_summary_png_file_name <- file.path(getwd(), "230_Isserlin_RCy3_intro", "images", "mesenem_summary.png")

if(file.exists(mesenem_summary_png_file_name)){
  #cytoscape hangs waiting for user response if file already exists. Remove it first
}
```

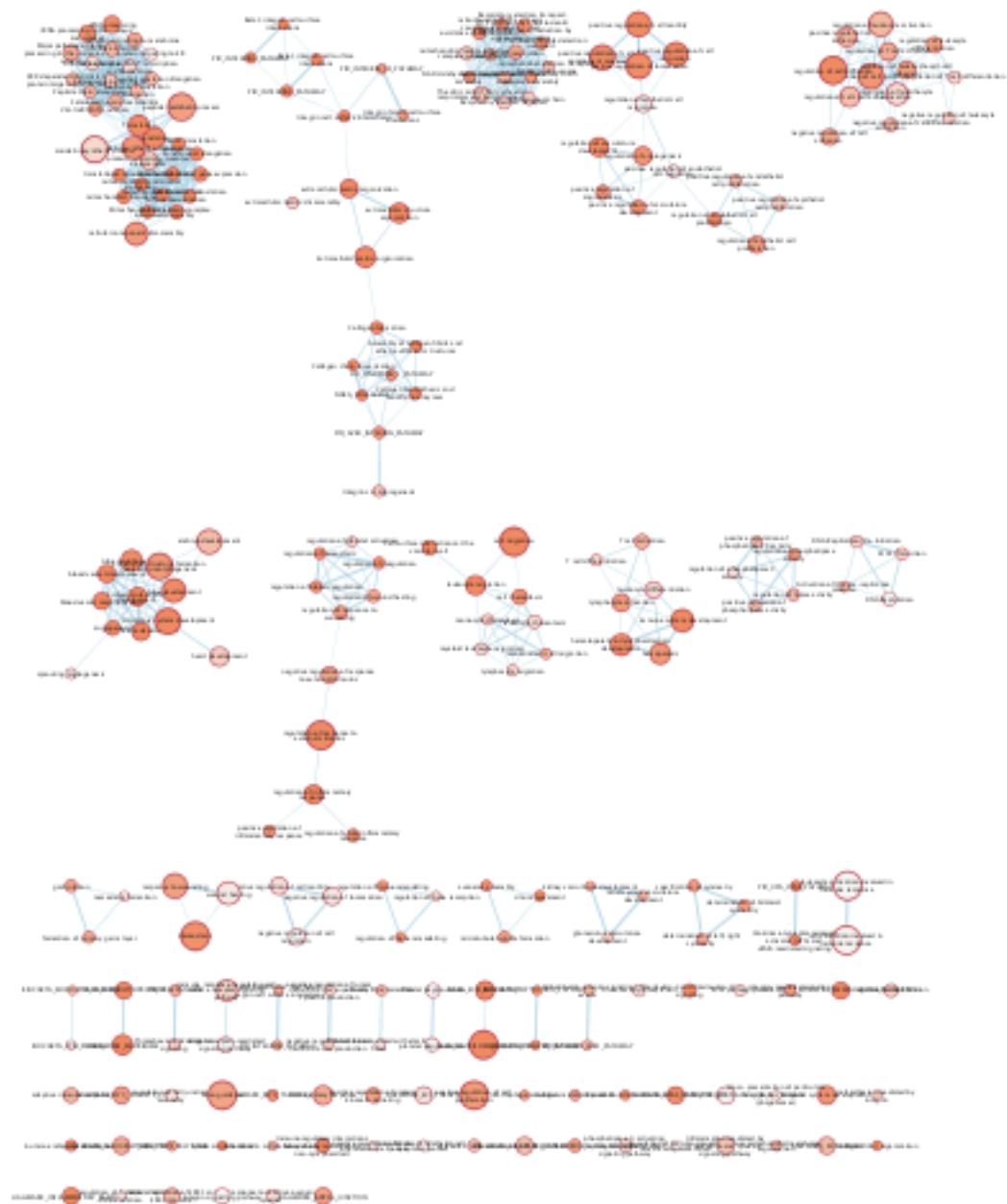


Figure 11.13: Example Enrichment Map created when running an enrichment analysis using Enrichment-Browser ORA with the genes differential in Mesenchymal OV

```
file.remove(mesenem_summary_png_file_name)
}

#export the network
exportImage(mesenem_summary_png_file_name, type = "png")
```



Figure 11.14: Example Annotated Enrichment Map created when running an enrichment analysis using EnrichmentBrowser ORA with the genes that differential in mesenchymal OV

Chapter 12

240: Fluent genomic data analysis with plyranges

12.1 Instructor names and contact information

- Stuart Lee (lee.s@wehi.edu.au)
- Michael Lawrence (michafla@gmail.com)

12.2 Workshop Description

In this workshop, we will give an overview of how to perform low-level analyses of genomic data using the grammar of genomic data transformation defined in the `plyranges` package. We will cover:

- introduction to `GRanges`
- overview of the core verbs for arithmetic, restriction, and aggregation of `GRanges` objects
- performing joins between `GRanges` objects
- designing pipelines to quickly explore data from `AnnotationHub`
- reading BAM and other file types as `GRanges` objects

The workshop will be a computer lab, in which the participants will be able to ask questions and interact with the instructors.

12.2.1 Pre-requisites

This workshop is self-contained however familiarity with the following would be useful:

- `plyranges` vignette
- the `GenomicRanges` and `IRanges` packages
- `tidyverse` approaches to data analysis

12.2.2 Workshop Participation

Students will work through an Rmarkdown document while the instructors respond to any questions they have.

12.2.3 *R / Bioconductor* packages used

- plyranges
- airway
- AnnotationHub
- GenomicRanges
- IRanges
- S4Vectors

12.2.4 Time outline

Activity	Time
Overview of GRanges	5m
The plyranges grammar	20m
I/O and data pipelines	20m

12.3 Workshop goals and objectives

12.3.1 Learning goals

- Understand that GRanges follows tidy data principles
- Apply the plyranges grammar to genomic data analysis

12.3.2 Learning objectives

- Use AnnotationHub to find and summarise data
- Read files into R as GRanges objects
- Perform coverage analysis
- Build data pipelines for analysis based on GRanges

12.4 Workshop

12.5 Introduction

12.5.1 What is plyranges?

The plyranges package is a domain specific language (DSL) built on top of the IRanges and GenomicRanges packages (Lee, Cook, and Lawrence (2018); Lawrence et al. (2013)). It is designed to quickly and coherently analyse genomic data in the form of GRanges objects (more on those later!) and from a wide variety of genomic data file types. For users who are familiar with the tidyverse, the grammar that plyranges implements will look familiar but with a few modifications for genomic specific tasks.

12.5.2 Why use plyranges?

The grammar that plyranges develops is helpful for reasoning about genomics data analysis, and provides a way of developing short readable analysis pipelines. We have tried to emphasise consistency and code

readability by following the design principles outlined by Green and Petre (1996).

One of the goals of plyranges is to provide an alternative entry point to analysing genomics data with Bioconductor, especially for R beginners and R users who are more familiar with the tidyverse approach to data analysis. As a result, we have de-emphasised the use of more complicated data structures provided by core Bioconductor packages that are useful for programming with.

12.5.3 Who is this workshop for?

This workshop is intended for new users of Bioconductor, users who are interested to learn about grammar based approaches for data analysis, and users who are interested in learning how to use R to perform analyses like those available in the command line packages BEDTools (Quinlan and Hall (2010)).

If that's you, let's begin!

12.6 Setup

To participate in this workshop you'll need to have R ≥ 3.5 and install the plyranges, AnnotationHub, and airway Bioconductor 3.7 packages (Morgan (2018); Love (2018)). You can achieve this by installing the BiocManager package from CRAN, loading it then running the install command:

```
install.packages("BiocManager")
library(BiocManager)
install(c("plyranges", "AnnotationHub", "airway"))
```

12.7 What are GRanges objects?

The plyranges package is built on the core Bioconductor data structure GRanges. It is very similar to the base R data.frame but with appropriate semantics for a genomics experiment: it has fixed columns for the chromosome, start and end coordinates, and the strand, along with an arbitrary set of additional columns, consisting of measurements or metadata specific to the data type or experiment (figure 12.1).

GRanges balances flexibility with formal constraints, so that it is applicable to virtually any genomic workflow, while also being semantically rich enough to support high-level operations on genomic ranges. As a core data structure, GRanges enables compatibility between plyranges and the rest of Bioconductor.

Since a GRanges object is similar to a data.frame, we can use plyranges to construct a GRanges object from a data.frame. We'll start by supposing we have a data.frame of genes from the yeast genome:

```
library(plyranges, quietly = TRUE)
genes <- data.frame(seqnames = "VI",
                     start = c(3322, 3030, 1437, 5066, 6426, 836),
                     end = c(3846, 3338, 2615, 5521, 7565, 1363),
                     strand = c("-", "-", "-", "+", "+", "+"),
                     gene_id=c("YFL064C", "YFL065C", "YFL066C",
                             "YFL063W", "YFL062W", "YFL067W"),
                     stringsAsFactors = FALSE)
gr <- as_granges(genes)
gr
#> GRanges object with 6 ranges and 1 metadata column:
#>      seqnames      ranges strand /      gene_id
#>      <Rle> <IRanges> <Rle> / <character>
#> [1]      VI 3322-3846      - /      YFL064C
```

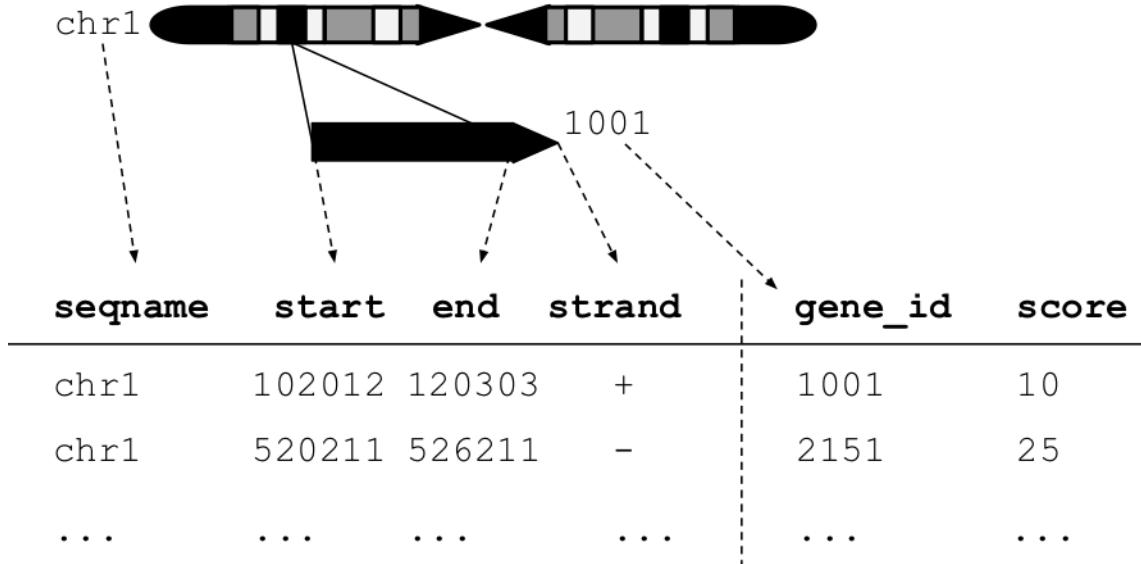


Figure 12.1: An illustration of a GRanges data object for a single sample from an RNA-seq experiment. The core components of the object include a seqname column (representing the chromosome), a ranges column which consists of start and end coordinates for a genomic region, and a strand identifier (either positive, negative, or unstranded). Metadata are included as columns to the right of the dotted line as annotations (gene-id) or range level covariates (score).

```
#> [2] VI 3030-3338      - /   YFL065C
#> [3] VI 1437-2615      - /   YFL066C
#> [4] VI 5066-5521      + /   YFL063W
#> [5] VI 6426-7565      + /   YFL062W
#> [6] VI 836-1363       + /   YFL067W
#> -----
#> seqinfo: 1 sequence from an unspecified genome; no seqlengths
```

The `as_granges` method takes a `data.frame` and allows you to quickly convert it to a `GRanges` object (and you can also specify which columns in the `data.frame` correspond to the columns in the `GRanges`).

A `GRanges` object follows tidy data principles: it is a rectangular table corresponding to a single biological context (Wickham (2014)). Each row contains a single observation and each column is a variable describing the observations. In the example above, each row corresponds to a single gene, and each column contains information about those genes. As `GRanges` are tidy, we have constructed `plyranges` to follow and extend the grammar in the R package `dplyr`.

12.8 The Grammar

Here we provide a quick overview of the functions available in `plyranges` and illustrate their use with some toy examples (see 12.11 for an overview). In the final section we provide two worked examples (with exercises) that show you can use `plyranges` to explore publicly available genomics data and perform coverage analysis of BAM files.

12.8.1 Core verbs

The plyranges grammar is simply a set of verbs that define actions to be performed on a GRanges (for a complete list see the appendix). Verbs can be composed together using the pipe operator, `%>%`, which can be read as ‘then’. Here’s a simple pipeline: first we will add two columns, one corresponding to the gene_type and another with the GC content (which we make up by drawing from a uniform distribution). Second we will remove genes if they have a width less than 400bp.

```
set.seed(2018-07-28)
gr2 <- gr %>%
  mutate(gene_type = "ORF",
        gc_content = runif(n())) %>%
  filter(width > 400)
gr2
#> GRanges object with 5 ranges and 3 metadata columns:
#>   seqnames      ranges strand |   gene_id   gene_type
#>   <Rle> <IRanges>  <Rle> / <character> <character>
#> [1]    VI 3322-3846     - /    YFL064C      ORF
#> [2]    VI 1437-2615     - /    YFL066C      ORF
#> [3]    VI 5066-5521     + /    YFL063W      ORF
#> [4]    VI 6426-7565     + /    YFL062W      ORF
#> [5]    VI 836-1363     + /    YFL067W      ORF
#> 
#>   gc_content
#>   <numeric>
#> [1] 0.49319754820317
#> [2] 0.216616344172508
#> [3] 0.747259315103292
#> [4] 0.907683959929273
#> [5] 0.221016310621053
#> -----
#> 
#> seqinfo: 1 sequence from an unspecified genome; no seqlengths
```

The `mutate()` function is used to add columns, here we’ve added one column called `gene_type` where all values are set to “ORF” (standing for open reading frame) and another called `gc_content` with random uniform values. The `n()` operator returns the number of ranges in GRanges object, but can only be evaluated inside of one of the plyranges verbs.

The `filter()` operation returns ranges if the expression evaluates to TRUE. Multiple expressions can be composed together and will be evaluated as &

```
gr2 %>%
  filter(strand == "+", gc_content > 0.5)
#> GRanges object with 2 ranges and 3 metadata columns:
#>   seqnames      ranges strand |   gene_id   gene_type
#>   <Rle> <IRanges>  <Rle> / <character> <character>
#> [1]    VI 5066-5521     + /    YFL063W      ORF
#> [2]    VI 6426-7565     + /    YFL062W      ORF
#> 
#>   gc_content
#>   <numeric>
#> [1] 0.747259315103292
#> [2] 0.907683959929273
#> -----
#> 
#> seqinfo: 1 sequence from an unspecified genome; no seqlengths
# is the same as using `&`
```

`gr2 %>%`

```

filter(strand == "+" & gc_content > 0.5)
#> GRanges object with 2 ranges and 3 metadata columns:
#>   seqnames      ranges strand |  gene_id  gene_type
#>   <Rle> <IRanges> <Rle> | <character> <character>
#> [1]    VI 5066-5521      + |    YFL063W      ORF
#> [2]    VI 6426-7565      + |    YFL062W      ORF
#>   gc_content
#>   <numeric>
#> [1] 0.747259315103292
#> [2] 0.907683959929273
#> -----
#> seqinfo: 1 sequence from an unspecified genome; no seqlengths
# but different from using or '|'
gr2 %>%
  filter(strand == "+" | gc_content > 0.5)
#> GRanges object with 3 ranges and 3 metadata columns:
#>   seqnames      ranges strand |  gene_id  gene_type
#>   <Rle> <IRanges> <Rle> | <character> <character>
#> [1]    VI 5066-5521      + |    YFL063W      ORF
#> [2]    VI 6426-7565      + |    YFL062W      ORF
#> [3]    VI 836-1363      + |    YFL067W      ORF
#>   gc_content
#>   <numeric>
#> [1] 0.747259315103292
#> [2] 0.907683959929273
#> [3] 0.221016310621053
#> -----
#> seqinfo: 1 sequence from an unspecified genome; no seqlengths

```

Now that we have some measurements over our genes, we are most likely interested in performing the favourite tasks of a biological data scientist: taking averages and counting. This is achieved with the `summarise()` verb which will return a `DataFrame` object (Why is this the case?).

```

gr2 %>%
  summarise(avg_gc = mean(gc_content),
            n = n())
#> DataFrame with 1 row and 2 columns
#>   avg_gc      n
#>   <numeric> <integer>
#> 1 0.517154695605859      5

```

which isn't very exciting when performed without `summarise()`'s best friend the `group_by()` operator:

```

gr2 %>%
  group_by(strand) %>%
  summarise(avg_gc = mean(gc_content),
            n = n())
#> DataFrame with 2 rows and 3 columns
#>   strand      avg_gc      n
#>   <Rle> <numeric> <integer>
#> 1     + 0.625319861884539      3
#> 2     - 0.354906946187839      2

```

The `group_by()` operator causes `plyranges` verbs to behave differently. Instead of acting on all the ranges in a `GRanges` object, the verbs act within each group of ranges defined by the values in the grouping column(s).

The `group_by()` operator does not change the appearance the of a GRanges object (well for the most part):

```
by_strand <- gr2 %>%
  group_by(strand)
by_strand
#> GRanges object with 5 ranges and 3 metadata columns:
#> Groups: strand [2]
#>   seqnames      ranges strand |   gene_id   gene_type
#>   <Rle> <IRanges> <Rle> | <character> <character>
#> [1]   VI 3322-3846     - /    YFL064C      ORF
#> [2]   VI 1437-2615     - /    YFL066C      ORF
#> [3]   VI 5066-5521     + /    YFL063W      ORF
#> [4]   VI 6426-7565     + /    YFL062W      ORF
#> [5]   VI 836-1363     + /    YFL067W      ORF
#>   gc_content
#>   <numeric>
#> [1] 0.49319754820317
#> [2] 0.216616344172508
#> [3] 0.747259315103292
#> [4] 0.907683959929273
#> [5] 0.221016310621053
#> -----
#> seqinfo: 1 sequence from an unspecified genome; no seqlengths
```

Now any verb we apply to our grouped GRanges, acts on each partition:

```
by_strand %>%
  filter(n() > 2)
#> GRanges object with 3 ranges and 3 metadata columns:
#> Groups: strand [1]
#>   seqnames      ranges strand |   gene_id   gene_type
#>   <Rle> <IRanges> <Rle> | <character> <character>
#> [1]   VI 5066-5521     + /    YFL063W      ORF
#> [2]   VI 6426-7565     + /    YFL062W      ORF
#> [3]   VI 836-1363     + /    YFL067W      ORF
#>   gc_content
#>   <numeric>
#> [1] 0.747259315103292
#> [2] 0.907683959929273
#> [3] 0.221016310621053
#> -----
#> seqinfo: 1 sequence from an unspecified genome; no seqlengths
by_strand %>%
  mutate(avg_gc_strand = mean(gc_content))
#> GRanges object with 5 ranges and 4 metadata columns:
#> Groups: strand [2]
#>   seqnames      ranges strand |   gene_id   gene_type
#>   <Rle> <IRanges> <Rle> | <character> <character>
#> [1]   VI 3322-3846     - /    YFL064C      ORF
#> [2]   VI 1437-2615     - /    YFL066C      ORF
#> [3]   VI 5066-5521     + /    YFL063W      ORF
#> [4]   VI 6426-7565     + /    YFL062W      ORF
#> [5]   VI 836-1363     + /    YFL067W      ORF
#>   gc_content      avg_gc_strand
#>   <numeric>       <numeric>
```

```
#> [1] 0.49319754820317 0.354906946187839
#> [2] 0.216616344172508 0.354906946187839
#> [3] 0.747259315103292 0.625319861884539
#> [4] 0.907683959929273 0.625319861884539
#> [5] 0.221016310621053 0.625319861884539
#> -----
#> seqinfo: 1 sequence from an unspecified genome; no seqlengths
```

To remove grouping use the `ungroup()` verb:

```
by_strand %>%
  ungroup()
#> GRanges object with 5 ranges and 3 metadata columns:
#>   seqnames      ranges strand |  gene_id  gene_type
#>   <Rle> <IRanges> <Rle> | <character> <character>
#> [1]    VI 3322-3846     - /    YFL064C      ORF
#> [2]    VI 1437-2615     - /    YFL066C      ORF
#> [3]    VI 5066-5521     + /    YFL063W      ORF
#> [4]    VI 6426-7565     + /    YFL062W      ORF
#> [5]    VI 836-1363     + /    YFL067W      ORF
#>   gc_content
#>   <numeric>
#> [1] 0.49319754820317
#> [2] 0.216616344172508
#> [3] 0.747259315103292
#> [4] 0.907683959929273
#> [5] 0.221016310621053
#> -----
#> seqinfo: 1 sequence from an unspecified genome; no seqlengths
```

Finally, metadata columns can be selected using the `select()` verb:

```
gr2 %>%
  select(gene_id, gene_type)
#> GRanges object with 5 ranges and 2 metadata columns:
#>   seqnames      ranges strand |  gene_id  gene_type
#>   <Rle> <IRanges> <Rle> | <character> <character>
#> [1]    VI 3322-3846     - /    YFL064C      ORF
#> [2]    VI 1437-2615     - /    YFL066C      ORF
#> [3]    VI 5066-5521     + /    YFL063W      ORF
#> [4]    VI 6426-7565     + /    YFL062W      ORF
#> [5]    VI 836-1363     + /    YFL067W      ORF
#> -----
#> seqinfo: 1 sequence from an unspecified genome; no seqlengths
# is the same as not selecting gc_content
gr2 %>%
  select(-gc_content)
#> GRanges object with 5 ranges and 2 metadata columns:
#>   seqnames      ranges strand |  gene_id  gene_type
#>   <Rle> <IRanges> <Rle> | <character> <character>
#> [1]    VI 3322-3846     - /    YFL064C      ORF
#> [2]    VI 1437-2615     - /    YFL066C      ORF
#> [3]    VI 5066-5521     + /    YFL063W      ORF
#> [4]    VI 6426-7565     + /    YFL062W      ORF
#> [5]    VI 836-1363     + /    YFL067W      ORF
```

```
#> -----
#> seqinfo: 1 sequence from an unspecified genome; no seqlengths
# you can also select by metadata column index
gr2 %>%
  select(1:2)
#> GRanges object with 5 ranges and 2 metadata columns:
#>   seqnames      ranges strand |  gene_id    gene_type
#>   <Rle> <IRanges> <Rle> | <character> <character>
#> [1]    VI 3322-3846     - /    YFL064C      ORF
#> [2]    VI 1437-2615     - /    YFL066C      ORF
#> [3]    VI 5066-5521     + /    YFL063W      ORF
#> [4]    VI 6426-7565     + /    YFL062W      ORF
#> [5]    VI 836-1363     + /    YFL067W      ORF
#> -----
#> seqinfo: 1 sequence from an unspecified genome; no seqlengths
```

12.8.2 Verbs specific to GRanges

We have seen how you can perform restriction and aggregation on GRanges, but what about specific actions for genomics data analysis, like arithmetic, nearest neighbours or finding overlaps?

12.8.2.1 Arithmetic

Arithmetic operations transform range coordinates, as defined by their *start*, *end* and *width* columns. These three variables are mutually dependent so we have to take care when modifying them. For example, changing the *width* column needs to change either the *start*, *end* or both to preserve integrity of the GRanges:

```
# by default setting width will fix the starting coordinate
gr %>%
  mutate(width = width + 1)
#> GRanges object with 6 ranges and 1 metadata column:
#>   seqnames      ranges strand |  gene_id
#>   <Rle> <IRanges> <Rle> | <character>
#> [1]    VI 3322-3847     - /    YFL064C
#> [2]    VI 3030-3339     - /    YFL065C
#> [3]    VI 1437-2616     - /    YFL066C
#> [4]    VI 5066-5522     + /    YFL063W
#> [5]    VI 6426-7566     + /    YFL062W
#> [6]    VI 836-1364     + /    YFL067W
#> -----
#> seqinfo: 1 sequence from an unspecified genome; no seqlengths
```

We introduce the `anchor_direction()` operator to clarify these modifications. Supported anchor points include the start `anchor_start()`, end `anchor_end()` and midpoint `anchor_center()`:

```
gr %>%
  anchor_end() %>%
  mutate(width = width * 2)
#> GRanges object with 6 ranges and 1 metadata column:
#>   seqnames      ranges strand |  gene_id
#>   <Rle> <IRanges> <Rle> | <character>
#> [1]    VI 2797-3846     - /    YFL064C
#> [2]    VI 2721-3338     - /    YFL065C
```

```
#> [3] VI 258-2615 - / YFL066C
#> [4] VI 4610-5521 + / YFL063W
#> [5] VI 5286-7565 + / YFL062W
#> [6] VI 308-1363 + / YFL067W
#> -----
#> seqinfo: 1 sequence from an unspecified genome; no seqlengths

gr %>%
  anchor_center() %>%
  mutate(width = width * 2)
#> GRanges object with 6 ranges and 1 metadata column:
#>   seqnames      ranges strand |   gene_id
#>   <Rle> <IRanges> <Rle> | <character>
#> [1] VI 3059-4108 - / YFL064C
#> [2] VI 2875-3492 - / YFL065C
#> [3] VI 847-3204 - / YFL066C
#> [4] VI 4838-5749 + / YFL063W
#> [5] VI 5856-8135 + / YFL062W
#> [6] VI 572-1627 + / YFL067W
#> -----
#> seqinfo: 1 sequence from an unspecified genome; no seqlengths
```

Note that the anchoring modifier introduces an important departure from the IRanges and GenomicRanges packages: by default we ignore the strandedness of a GRanges object, and instead we provide verbs that make stranded actions explicit. In this case of anchoring we provide the `anchor_3p()` and `anchor_5p()` to perform anchoring on the 3' and 5' ends of a range.

The table in the appendix provides a complete list of arithmetic options available in plyranges.

12.8.2.2 Genomic aggregation

There are two verbs that can be used to aggregate over nearest neighbours: `reduce_ranges()` and `disjoin_ranges()`. The reduce verb merges overlapping and neighbouring ranges:

```
gr %>% reduce_ranges()
#> GRanges object with 5 ranges and 0 metadata columns:
#>   seqnames      ranges strand
#>   <Rle> <IRanges> <Rle>
#> [1] VI 836-1363 *
#> [2] VI 1437-2615 *
#> [3] VI 3030-3846 *
#> [4] VI 5066-5521 *
#> [5] VI 6426-7565 *
#> -----
#> seqinfo: 1 sequence from an unspecified genome; no seqlengths
```

We could find out which genes are overlapping each other by aggregating over the `gene_id` column and storing the result in a List column:

```
gr %>%
  reduce_ranges(gene_id = List(gene_id))
#> GRanges object with 5 ranges and 1 metadata column:
#>   seqnames      ranges strand |   gene_id
#>   <Rle> <IRanges> <Rle> | <CharacterList>
#> [1] VI 836-1363 * / YFL067W
```

```
#> [2] VI 1437-2615 * / YFL066C
#> [3] VI 3030-3846 * / YFL065C, YFL064C
#> [4] VI 5066-5521 * / YFL063W
#> [5] VI 6426-7565 * / YFL062W
#> -----
#> seqinfo: 1 sequence from an unspecified genome; no seqlengths
```

The disjoin verb takes the union of end points over all ranges, and results in an expanded range

```
gr %>%
  disjoint_ranges(gene_id = List(gene_id))
#> GRanges object with 7 ranges and 1 metadata column:
#>   seqnames      ranges strand |      gene_id
#>   <Rle> <IRanges> <Rle> / <CharacterList>
#> [1] VI 836-1363    * / YFL067W
#> [2] VI 1437-2615   * / YFL066C
#> [3] VI 3030-3321   * / YFL065C
#> [4] VI 3322-3338   * / YFL064C, YFL065C
#> [5] VI 3339-3846   * / YFL064C
#> [6] VI 5066-5521   * / YFL063W
#> [7] VI 6426-7565   * / YFL062W
#> -----
#> seqinfo: 1 sequence from an unspecified genome; no seqlengths
```

You may have noticed that the resulting range are now unstranded, to take into account stranded features use the *directed* prefix.

12.8.2.3 Overlaps

Another important class of operations for genomics data analysis is finding overlaps or nearest neighbours. Here's where we will introduce the `plyranges` join operators and the overlap aggregation operators.

Now let's now suppose we have some additional measurements that we have obtained from a new experiment on yeast. These measurements are for three different replicates and represent single nucleotide or insertion deletion intensities from an array. Our collaborator has given us three different data.frames with the data but they all have names inconsistent with the GRanges data structure. Our goal is to unify these into a single GRanges object with column for each measurement and column for the sample identifier.

Here's our data:

```
set.seed(66+105+111+99+49+56)

pos <- sample(1:10000, size = 100)
size <- sample(1:3, size = 100, replace = TRUE)
rep1 <- data.frame(chr = "VI",
                     pos = pos,
                     size = size,
                     X = rnorm(100, mean = 2),
                     Y = rnorm(100, mean = 1))

rep2 <- data.frame(chrom = "VI",
                     st = pos,
                     width = size,
                     X = rnorm(100, mean = 0.5, sd = 3),
                     Y = rnorm(100, sd = 2))
```

```
rep3 <- data.frame(chromosome = "VI",
                     start = pos,
                     width = size,
                     X = rnorm(100, mean = 2, sd = 3),
                     Y = rnorm(100, mean = 4, sd = 0.5))
```

For each replicate we want to construct a GRanges object:

```
# we can tell as_granges which columns in the data.frame
# are the seqnames, and range coordinates
rep1 <- as_granges(rep1, seqnames = chr, start = pos, width = size)
rep2 <- as_granges(rep2, seqnames = chrom, start = st)
rep3 <- as_granges(rep3, seqnames = chromosome)
```

And to construct our final GRanges object we can bind all our replicates together:

```
intensities <- bind_ranges(rep1, rep2, rep3, .id = "replicate")
# sort by the starting coordinate
arrange(intensities, start)
#> GRanges object with 300 ranges and 3 metadata columns:
#>   seqnames      ranges strand |           X           Y
#>   <Rle> <IRanges> <Rle> | <numeric> <numeric>
#> [1]     VI  99-100    * /  2.18077108319727  1.15893283880961
#> [2]     VI  99-100    * / -1.14331853023759 -1.84545382593297
#> [3]     VI  99-100    * /  4.42535734042167  3.53884540635964
#> [4]     VI  110-111   * /  1.41581829875993 -0.262026041514519
#> [5]     VI  110-111   * /  0.0203313104969627 -1.18095384044377
#> ...
#> [296]    VI 9671-9673   * /  0.756423808063998 -0.24544579405238
#> [297]    VI 9671-9673   * /  0.715559817063897  4.6963376859667
#> [298]    VI 9838-9839   * /  1.83836043312615  0.267996156074214
#> [299]    VI 9838-9839   * / -4.62774336616852 -3.45271032367217
#> [300]    VI 9838-9839   * / -0.285141455604857  4.16118336728783
#>   replicate
#>   <character>
#> [1]     1
#> [2]     2
#> [3]     3
#> [4]     1
#> [5]     2
#> ...
#> [296]   2
#> [297]   3
#> [298]   1
#> [299]   2
#> [300]   3
#> -----
#>   seqinfo: 1 sequence from an unspecified genome; no seqlengths
```

Now we would like to filter our positions if they overlap one of the genes we have, one way we could achieve this is with the `filter_by_overlaps()` operator:

```
olap <- filter_by_overlaps(intensities, gr)
olap
#> GRanges object with 108 ranges and 3 metadata columns:
```

```

#>      seqnames    ranges strand |           X           Y
#>      <Rle> <IRanges> <Rle> |           <numeric>       <numeric>
#> [1]    VI 3300-3302    * /  2.26035109119553  1.06568454447289
#> [2]    VI 1045-1047    * /  2.21864775104592  1.08876098679654
#> [3]    VI 3791-3793    * /  2.64672161626967  1.49683387927811
#> [4]    VI     6503    * /  2.97614069143102 -0.842974371427135
#> [5]    VI 2613-2615    * /  0.829706619102562  1.00867596057383
#> ...
#> [104]   VI     2288    * /  4.51377823089056  3.98977444171162
#> [105]   VI     7191    * / -3.91180888573709  5.0451068476909
#> [106]   VI     5490    * /  6.8026659440345  4.71157047258809
#> [107]   VI 5268-5270    * / -1.40753324511308  4.48936193681021
#> [108]   VI     7333    * / -0.807795033496545  4.12171733927051
#>      replicate
#>      <character>
#> [1]    1
#> [2]    1
#> [3]    1
#> [4]    1
#> [5]    1
#> ...
#> [104]   3
#> [105]   3
#> [106]   3
#> [107]   3
#> [108]   3
#> -----
#> seqinfo: 1 sequence from an unspecified genome; no seqlengths

```

Another option would be to perform a join operation. A join acts on two GRanges objects, a query and a subject. The join operator retains the metadata from the query and subject ranges. All join operators generate a set of hits based on overlap or proximity of ranges and use those hits to merge the two datasets in different ways (figure 12.2, provides an overview of the overlap joins). We can further restrict the matching by whether the query is completely *within* the subject, and adding the *directed* suffix ensures that matching ranges have the same direction (strand).

Going back to our example, the overlap inner join will return all intensities that overlap a gene and propagate which gene_id a given intensity belongs to:

```
olap <- join_overlap_inner(intensities, gr)
```

If we wanted to return all intensities regardless of overlap we could use the left join, and a missing value will be propagated to the gene_id column if there isn't any overlap.

```

join_overlap_left(intensities, gr)
#> GRanges object with 300 ranges and 4 metadata columns:
#>      seqnames    ranges strand |           X           Y
#>      <Rle> <IRanges> <Rle> |           <numeric>       <numeric>
#> [1]    VI 3300-3302    * /  2.26035109119553  1.06568454447289
#> [2]    VI 1045-1047    * /  2.21864775104592  1.08876098679654
#> [3]    VI 4206-4207    * /  3.08931723239393  2.4388720191285
#> [4]    VI 3791-3793    * /  2.64672161626967  1.49683387927811
#> [5]    VI     6503    * /  2.97614069143102 -0.842974371427135
#> ...
#> [296]   VI 8881-8882    * /  4.15079125018189  4.47576275354234

```

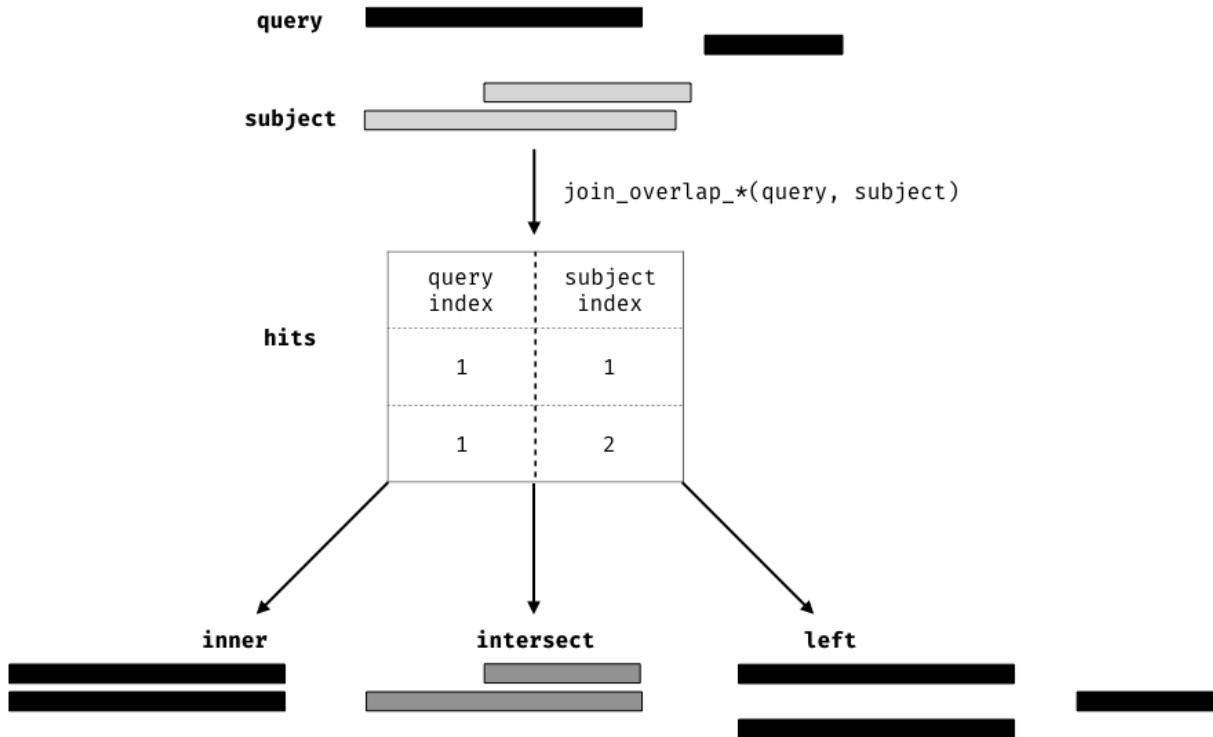


Figure 12.2: Illustration of the three overlap join operators. Each join takes query and subject range as input (black and light gray rectangles, respectively). An index for the join is computed, returning a `Hits` object, which contains the indices of where the subject overlaps the query range. This index is used to expand the query ranges by where it was ‘hit’ by the subject ranges. The join semantics alter what is returned: for an **inner** join the query range is returned for each match, for an **intersect** the intersection is taken between overlapping ranges, and for a **left** join all query ranges are returned even if the subject range does not overlap them. This principle is generally applied through the `plyranges` package for both overlaps and nearest neighbour operations.

```
#> [297] VI 7333 * / -0.807795033496545 4.12171733927051
#> [298] VI 715-716 * / -0.988759384492962 3.53496809806709
#> [299] VI 426-428 * / 10.5245928564545 3.52355010049711
#> [300] VI 9187-9188 * / 1.15204426681652 4.12867380190036
#> replicate gene_id
#> <character> <character>
#> [1] 1 YFL065C
#> [2] 1 YFL067W
#> [3] 1 <NA>
#> [4] 1 YFL064C
#> [5] 1 YFL062W
#> ... ...
#> [296] 3 <NA>
#> [297] 3 YFL062W
#> [298] 3 <NA>
#> [299] 3 <NA>
#> [300] 3 <NA>
#> -----
#> seqinfo: 2 sequences from an unspecified genome; no seqlengths
```

If we are interested in finding how much overlap there is we could use an intersect join. For example we could compute the fraction of overlap of the genes with themselves:

```
gr %>%
  mutate(gene_length = width) %>%
  join_overlap_intersect(gr, suffix = c(".query", ".subject")) %>%
  filter(gene_id.query != gene_id.subject) %>%
  mutate(folap = width / gene_length)
#> GRanges object with 2 ranges and 4 metadata columns:
#>   seqnames ranges strand | gene_id.query gene_length
#>   <Rle> <IRanges> <Rle> | <character> <integer>
#>   [1] VI 3322-3338 - / YFL064C 525
#>   [2] VI 3322-3338 - / YFL065C 309
#>   gene_id.subject folap
#>   <character> <numeric>
#>   [1] YFL065C 0.0323809523809524
#>   [2] YFL064C 0.0550161812297735
#> -----
#> seqinfo: 1 sequence from an unspecified genome; no seqlengths
```

12.8.3 Exercises

There are of course many more functions available in plyranges but hopefully the previous examples are sufficient to give you an idea of how to incorporate plyranges into your genomic workflows.

Here's a few exercises to help you familiar with the verbs:

1. Find the average intensity of the X and Y measurements for each each replicate over all positions in the intensities object
2. Add a new column to the intensities object that is the distance from each position to its closest gene (hint IRanges::distance)
3. Find flanking regions downstream of the genes in gr that have width of 8bp (hint: type: flank_ and see what comes up!)
4. Are any of the intensities positions within the flanking region? (use an overlap join to find out!)

12.9 Data import and creating pipelines

Here we provide some realistic examples of using plyranges to import genomic data and perform exploratory analyses.

12.9.1 Worked example: exploring BigWig files from AnnotationHub

In the workflow of ChIP-seq data analysis, we are often interested in finding peaks from islands of coverage over a chromosome. Here we will use plyranges to explore ChIP-seq data from the Human Epigenome Roadmap project Roadmap Epigenomics Consortium et al. (2015).

12.9.1.1 Extracting data from AnnotationHub

This data is available on Bioconductor's AnnotationHub. First we construct an AnnotationHub, and then `query()` for all bigWigFiles related to the project that correspond to the following conditions:

1. are from methylation marks (H3K4ME in the title)
2. correspond to primary T CD8+ memory cells from peripheral blood
3. correspond to unimputed log10 P-values

First we construct a hub that contains all references to the EpigenomeRoadMap data and extract the metadata as a data.frame:

```
library(AnnotationHub)
library(magrittr)
ah <- AnnotationHub()
#> snapshotDate(): 2018-06-27
roadmap_hub <- ah %>%
  query("EpigenomeRoadMap")

# extract2 is just a call to `[[` 
metadata <- ah %>%
  query("Metadata") %>%
  extract2(names(.))
#> downloading 0 resources
#> loading from cache
#> '/home/mramos//.AnnotationHub/47270'

head(metadata)
#>   EID    GROUP    COLOR      MNEMONIC
#> 1 E001    ESC #924965    ESC.I3
#> 2 E002    ESC #924965    ESC.WA7
#> 3 E003    ESC #924965    ESC.H1
#> 4 E004 ES-deriv #4178AE ESDR.H1.BMP4.MESO
#> 5 E005 ES-deriv #4178AE ESDR.H1.BMP4.TROP
#> 6 E006 ES-deriv #4178AE    ESDR.H1.MSC
#>                               STD_NAME
#> 1                           ES-I3 Cells
#> 2                           ES-WA7 Cells
#> 3                           H1 Cells
#> 4 H1 BMP4 Derived Mesendoderm Cultured Cells
#> 5 H1 BMP4 Derived Trophoblast Cultured Cells
#> 6     H1 Derived Mesenchymal Stem Cells
```

```
#>                                              EDACC_NAME      ANATOMY      TYPE
#> 1                      ES-I3_Cell_Line      ESC PrimaryCulture
#> 2                      ES-WA7_Cell_Line     ESC PrimaryCulture
#> 3                      H1_Cell_Line       ESC PrimaryCulture
#> 4 H1_BMP4_Derived_Mesendoderm_Cultured_Cells ESC_DERIVED    ESCDerived
#> 5 H1_BMP4_Derived_Trophoblast_Cultured_Cells ESC_DERIVED    ESCDerived
#> 6          H1_Derived_Mesenchymal_Stem_Cells ESC_DERIVED    ESCDerived
#> AGE SEX SOLID LIQUID ETHNICITY SINGLEDONOR COMPOSITE
#> 1 CL Female <NA> <NA> SD
#> 2 CL Female <NA> <NA> SD
#> 3 CL Male   <NA> <NA> SD
#> 4 CL Male   <NA> <NA> SD
#> 5 CL Male   <NA> <NA> SD
#> 6 CL Male   <NA> <NA> SD
```

To find out the name of the sample corresponding to primary memory T-cells we can filter the data.frame. We extract the sample ID corresponding to our filter.

```
primary_tcells <- metadata %>%
  filter(ANATOMY == "BLOOD") %>%
  filter(TYPE == "PrimaryCell") %>%
  filter(EDACC_NAME == "CD8_Memory_Primary_Cells") %>%
  extract2("EID") %>%
  as.character()
primary_tcells
#> [1] "E048"
```

Now we can take our roadmap hub and query it based on our other conditions:

```
methylation_files <- roadmap_hub %>%
  query("BigWig") %>%
  query(primary_tcells) %>%
  query("H3K4ME[1-3]") %>%
  query("pval.signal")

methylation_files
#> AnnotationHub with 5 records
#> # snapshotDate(): 2018-06-27
#> # $dataprovder: BroadInstitute
#> # $species: Homo sapiens
#> # $rdataclass: BigWigFile
#> # additional mcols(): taxonomyid, genome, description,
#> #   coordinate_1_based, maintainer, rdatadateadded, preparerclass,
#> #   tags, rdatapath, sourceurl, sourcetype
#> # retrieve records with, e.g., 'object[["AH33454"]]'
```

So we'll take the first two entries and download them as BigWigFiles:

```
bw_files <- lapply(c("AH33454", "AH33455"), function(id) ah[[id]])
#> downloading 0 resources
#> loading from cache
#>     '/home/mramos//.AnnotationHub/38894'
#> downloading 0 resources
#> loading from cache
#>     '/home/mramos//.AnnotationHub/38895'
names(bw_files) <- c("HK34ME1", "HK34ME3")
```

We have our desired BigWig files so now we can start analysing them.

12.9.1.2 Reading BigWig files

For this analysis, we will call peaks from islands of scores over chromosome 10.

First, we extract the genome information from the first BigWig file and filter to get the range for chromosome 10. This range will be used as a filter when reading the file.

```
chr10_ranges <- bw_files %>%
  extract2(1L) %>%
  get_genome_info() %>%
  filter(seqnames == "chr10")
```

Then we read the BigWig file only extracting scores if they overlap chromosome 10. We also add the genome build information to the resulting ranges. This book-keeping is good practice as it ensures the integrity of any downstream operations such as finding overlaps.

```
# a function to read our bigwig files
read_chr10_scores <- function(file) {
  read_bigwig(file, overlap_ranges = chr10_ranges) %>%
  set_genome_info(genome = "hg19")
}

# apply the function to each file
chr10_scores <- lapply(bw_files, read_chr10_scores)
# bind the ranges to a single GRanges object
# and add a column to identify the ranges by signal type
chr10_scores <- bind_ranges(chr10_scores, .id = "signal_type")
chr10_scores

#> GRanges object with 11268683 ranges and 2 metadata columns:
#>   seqnames      ranges strand |      score
#>   <Rle>      <IRanges> <Rle> |      <numeric>
#>   [1]    chr10      1-60612    * /  0.0394200012087822
#>   [2]    chr10      60613-60784   * /  0.154219999909401
#>   [3]    chr10      60785-60832   * /  0.354730010032654
#>   [4]    chr10      60833-61004   * /  0.154219999909401
#>   [5]    chr10      61005-61455   * /  0.0394200012087822
#>   ...
#>   [11268679]  chr10  135524731-135524739   * /  0.0704099982976913
#>   [11268680]  chr10  135524740-135524780   * /  0.132530003786087
#>   [11268681]  chr10  135524781-135524789   * /  0.245049998164177
#>   [11268682]  chr10  135524790-135524811   * /  0.450120002031326
#>   [11268683]  chr10  135524812-135524842   * /  0.619729995727539
#>   signal_type
#>   <character>
#>   [1]    HK34ME1
```

```
#> [2]    HK34ME1
#> [3]    HK34ME1
#> [4]    HK34ME1
#> [5]    HK34ME1
#> ...    ...
#> [11268679]    HK34ME3
#> [11268680]    HK34ME3
#> [11268681]    HK34ME3
#> [11268682]    HK34ME3
#> [11268683]    HK34ME3
#> -----
#> seqinfo: 25 sequences from hg19 genome
```

Since we want to call peaks over each signal type, we will create a grouped GRanges object:

```
chr10_scores_by_signal <- chr10_scores %>%
  group_by(signal_type)
```

We can then filter to find the coordinates of the peak containing the maximum score for each signal. We can then find a 5000 nt region centred around the maximum position by anchoring and modifying the width.

```
chr10_max_score_region <- chr10_scores_by_signal %>%
  filter(score == max(score)) %>%
  ungroup() %>%
  anchor_center() %>%
  mutate(width = 5000)
```

Finally, the overlap inner join is used to restrict the chromosome 10 coverage islands, to the islands that are contained in the 5000nt region that surrounds the max peak for each signal type.

```
peak_region <- chr10_scores %>%
  join_overlap_inner(chr10_max_score_region) %>%
  filter(signal_type.x == signal_type.y)
# the filter ensures overlapping correct methylation signal
```

12.9.1.3 Exercises

1. Use the `reduce_ranges()` function to find all peaks for each signal type.
2. How could you annotate the scores to find out which genes overlap each peak found in 1.?
3. Plot a 1000nt window centred around the maximum scores for each signal type using the `ggbio` or `Gviz` package.

12.9.2 Worked example: coverage analysis of BAM files

A common quality control check in a genomics workflow is to perform coverage analysis over features of interest or over the entire genome (again we see the bioinformatician's love of counting things). Here we use the airway package to compute coverage histograms and show how you can read BAM files into memory as GRanges objects with plyranges.

First let's gather all the BAM files available to use in airway (see `browseVignettes("airway")` for more information about the data and how it was prepared):

```
bfs <- system.file("extdata", package = "airway") %>%
  dir(pattern = ".bam",
      full.names = TRUE)
```

```
# get sample names (everything after the underscore)
names(bfs) <- bfs %>%
  basename() %>%
  sub("_[^_]+$", "", .)
```

To start let's look at a single BAM file, we can compute the coverage of the alignments over all contigs in the BAM as follows:

```
first_bam_cvg <- bfs %>%
  extract2(1) %>%
  compute_coverage()
first_bam_cvg
#> GRanges object with 11423 ranges and 1 metadata column:
#>   seqnames      ranges strand |  score
#>   <Rle>      <IRanges> <Rle> | <integer>
#> [1]     1  1-11053772    * /      0
#> [2]     1  11053773-11053835  * /      1
#> [3]     1  11053836-11053839  * /      0
#> [4]     1  11053840-11053902  * /      1
#> [5]     1  11053903-11067865  * /      0
#> ...
#> [11419] GL000210.1       1-27682    * /      0
#> [11420] GL000231.1       1-27386    * /      0
#> [11421] GL000229.1       1-19913    * /      0
#> [11422] GL000226.1       1-15008    * /      0
#> [11423] GL000207.1       1-4262     * /      0
#> -----
#> seqinfo: 84 sequences from an unspecified genome
```

Here the coverage is computed without the entire BAM file being read into memory. Notice that the score here is the count of the number of alignments that cover a given range. To compute a coverage histogram, that is the number of bases that have a given coverage score for each contig we can simply use `summarise()`:

```
first_bam_cvg %>%
  group_by(seqnames, score) %>%
  summarise(n_bases_covered = sum(width))
#> DataFrame with 277 rows and 3 columns
#>   seqnames      score n_bases_covered
#>   <Rle> <integer> <integer>
#> 1     1          0        249202844
#> 2     10         0        135534747
#> 3     11         0        135006516
#> 4     12         0        133851895
#> 5     13         0        115169878
#> ...
#> 273    1         189        3
#> 274    1         191        3
#> 275    1         192        3
#> 276    1         193        2
#> 277    1         194        1
```

For RNA-seq experiments we are often interested in splitting up alignments based on whether the alignment has skipped a region from the reference (that is, there is an "N" in the cigar string, indicating an intron).

In plyranges this can be achieved with the `chop_by_introns()`, that will split up an alignment if there's an

introns. This results in a grouped ranges object.

To begin we can read the BAM file using `read_bam()`, this does not read the entire BAM into memory but instead waits for further functions to be applied to it.

```
# no index present in directory so we use index = NULL
split_bam <- bfs %>%
  extract2(1) %>%
  read_bam(index = NULL)
# nothing has been read
split_bam
#> DeferredGenomicRanges object with 0 ranges and 0 metadata columns:
#>   seqnames      ranges strand
#>   <Rle> <IRanges> <Rle>
#>   -----
#>   seqinfo: no sequences
```

For example we can select elements from the BAM file to be included in the resulting GRanges to load alignments into memory:

```
split_bam %>%
  select(flag)

#> DeferredGenomicRanges object with 14282 ranges and 4 metadata columns:
#>   seqnames      ranges strand / flag      cigar
#>   <Rle> <IRanges> <Rle> / <integer> <character>
#>   [1] 1 11053773-11053835 + / 99      63M
#>   [2] 1 11053840-11053902 - / 147     63M
#>   [3] 1 11067866-11067928 + / 163     63M
#>   [4] 1 11067931-11067993 - / 83      63M
#>   [5] 1 11072708-11072770 + / 99      63M
#>   ...
#>   [14278] 1 11364733-11364795 - / 147     63M
#>   [14279] 1 11365866-11365928 + / 163     63M
#>   [14280] 1 11365987-11366049 - / 83      63M
#>   [14281] 1 11386063-11386123 + / 99      2S61M
#>   [14282] 1 11386132-11386194 - / 147     63M
#>   qwidth      njunc
#>   <integer> <integer>
#>   [1] 63      0
#>   [2] 63      0
#>   [3] 63      0
#>   [4] 63      0
#>   [5] 63      0
#>   ...
#>   [14278] 63      0
#>   [14279] 63      0
#>   [14280] 63      0
#>   [14281] 63      0
#>   [14282] 63      0
#>   -----
#>   seqinfo: 84 sequences from an unspecified genome
```

Finally, we can split our alignments using the `chop_by_introns()`:

```
split_bam <- split_bam %>%
  chop_by_introns()
```

```

split_bam
#> GRanges object with 18124 ranges and 4 metadata columns:
#> Groups: introns [14282]
#>
#>      seqnames      ranges strand |      cigar      qwidth
#>      <Rle>      <IRanges> <Rle> | <character> <integer>
#> [1]     1 11053773-11053835    + /      63M       63
#> [2]     1 11053840-11053902    - /      63M       63
#> [3]     1 11067866-11067928    + /      63M       63
#> [4]     1 11067931-11067993    - /      63M       63
#> [5]     1 11072708-11072770    + /      63M       63
#> ...
#> ...
#> [18120]   1 11364733-11364795    - /      63M       63
#> [18121]   1 11365866-11365928    + /      63M       63
#> [18122]   1 11365987-11366049    - /      63M       63
#> [18123]   1 11386063-11386123    + /      2S61M     63
#> [18124]   1 11386132-11386194    - /      63M       63
#>
#>      njunc      introns
#>      <integer> <integer>
#> [1]     0          1
#> [2]     0          2
#> [3]     0          3
#> [4]     0          4
#> [5]     0          5
#> ...
#> ...
#> [18120]   0          14278
#> [18121]   0          14279
#> [18122]   0          14280
#> [18123]   0          14281
#> [18124]   0          14282
#> -----
#> seqinfo: 84 sequences from an unspecified genome
# look at all the junction reads
split_bam %>%
  filter(n() >= 2)
#> GRanges object with 7675 ranges and 4 metadata columns:
#> Groups: introns [3833]
#>
#>      seqnames      ranges strand |      cigar      qwidth
#>      <Rle>      <IRanges> <Rle> | <character> <integer>
#> [1]     1 11072744-11072800    + /      57M972N6M     63
#> [2]     1 11073773-11073778    + /      57M972N6M     63
#> [3]     1 11072745-11072800    - /      56M972N7M     63
#> [4]     1 11073773-11073779    - /      56M972N7M     63
#> [5]     1 11072746-11072800    + /      55M972N8M     63
#> ...
#> ...
#> [7671]   1 11345701-11345716    + /      47M11583N16M   63
#> [7672]   1 11334089-11334117    - /      29M11583N34M   63
#> [7673]   1 11345701-11345734    - /      29M11583N34M   63
#> [7674]   1 11334109-11334117    - /      9M11583N54M    63
#> [7675]   1 11345701-11345754    - /      9M11583N54M    63
#>
#>      njunc      introns
#>      <integer> <integer>
#> [1]     1          11
#> [2]     1          11

```

```
#> [3] 1 12
#> [4] 1 12
#> [5] 1 13
#> ... ...
#> [7671] 1 13913
#> [7672] 1 13915
#> [7673] 1 13915
#> [7674] 1 13916
#> [7675] 1 13916
#> -----
#> seqinfo: 84 sequences from an unspecified genome
```

12.9.2.1 Exercises

1. Compute the total depth of coverage across all features.
2. How could you compute the proportion of bases covered over an entire genome? (hint: `get_genome_info` and `S4Vectors::merge`)
3. How could you compute the strand specific genome wide coverage?
4. Create a workflow for computing the strand specific coverage for all BAM files.
5. For each sample plot total breadth of coverage against the number of bases covered faceted by each sample name.

12.10 What's next?

The grammar based approach for analysing data provides a consistent approach for analysing genomics experiments with Bioconductor. We have explored how plyranges can enable you to perform common analysis tasks required of a bioinformatician. As plyranges is still a new and growing package, there is definite room for improvement (and likely bugs), if you have any problems using it or think things could be easier or clearer please file an issue on github. Contributions from the Bioconductor community are also more than welcome!

12.11 Appendix

Table 12.2: Overview of the plyranges grammar. The core verbs are briefly described and categorised into one of: aggregation, unary or binary arithmetic, merging, modifier, or restriction. A verb is given bold text if its origin is from the dplyr grammar.

Category	Verb	Description
Aggregation	<code>summarise()</code>	aggregate over column(s)
	<code>disjoin_ranges()</code>	aggregate column(s) over the union of end coordinates
	<code>reduce_ranges()</code>	aggregate column(s) by merging overlapping and neighbouring ranges
	<code>mutate()</code>	modifies any column
	<code>select()</code>	select columns
Arithmetic (Unary)	<code>arrange()</code>	sort by columns
	<code>stretch()</code>	extend range by fixed amount
	<code>shift_(direction)</code>	shift coordinates
	<code>flank_(direction)</code>	generate flanking regions
	<code>%intersection%</code>	row-wise intersection

Category	Verb	Description
Arithmetic (Binary)	<code>%union%</code>	row-wise union
	<code>compute_coverage</code>	coverage over all ranges
	<code>%setdiff%</code>	row-wise set difference
	<code>between()</code>	row-wise gap range
	<code>span()</code>	row-wise spanning range
	<code>join_overlap_()</code>	merge by overlapping ranges
Merging	<code>join_nearest</code>	merge by nearest neighbour ranges
	<code>join_follow</code>	merge by following ranges
	<code>join_precedes</code>	merge by preceding ranges
	<code>union_ranges</code>	range-wise union
	<code>intersect_ranges</code>	range-wise intersect
	<code>setdiff_ranges</code>	range-wise set difference
Modifier	<code>complement_ranges</code>	range-wise union
	<code>anchor_direction()</code>	fix coordinates at direction
	<code>group_by()</code>	partition by column(s)
Restriction	<code>group_by_overlaps()</code>	partition by overlaps
	<code>filter()</code>	subset rows
	<code>filter_by_overlaps()</code>	subset by overlap
	<code>filter_by_non_overlaps()</code>	subset by no overlap

Chapter 13

250: Working with genomic data in R with the DECIPHER package

Authors: Nicholas Cooley¹ Last Modified: 18 July, 2018

13.1 Overview

13.1.1 Workshop Description

In this workshop we will give an introduction to working with biological sequence data in R using the Biostrings and DECIPHER packages. We will cover:

- Importing, viewing, and manipulating genomes in R
- Construction of sequence databases for organizing genomes
- Mapping syntenic regions between genomes with the `FindSynteny` function
- Understanding, accessing, and viewing objects of class `Synteny`
- Using syntenic information to predict orthologous genes
- Alignment of sequences and genomes with DECIPHER
- Downstream analyses enabled by syntenic mapping

13.1.2 Pre-requisites

- Familiarity with Biostrings
- Familiarity with DECIPHER Databases²

13.1.3 Workshop Participation

This will be a lab where participants follow along on their computers.

13.1.4 *R / Bioconductor* packages used

- Biostrings

¹University of Pittsburgh

²Wright, E. S. The R Journal 2016, 8 (1), 352–359.

- DECIPHER
- stringr
- FindHomology

13.1.5 Time outline

Activity	Time
Packages/Introduction	5m
Basic commands for sequences	5m
Sequence databases	10m
Demonstration of synteny mapping	10m
Explanation of function arguments	10m
Dissecting Synteny objects	10m
Visualization of syntenic blocks	10m
Alignment of syntenic regions	10m
Ortholog prediction from Synteny	10m
Constructing phylogenetic trees	10m

13.2 Workshop goals and objectives

13.2.1 Learning goals

- Understand a simple workflow for analysis of sequences in R and DECIPHER
- Learn the basic use and appropriate application of functions within DECIPHER

13.2.2 Learning objectives

- Learn basic commands for working with sequences in R
- Import genomes from online repositories or local files
- Map synteny between genomes
- Analyze a synteny map among multiple genomes
- Develop an understanding of the data structures involved
- Predict orthologs from syntenic maps
- Select core and pan genomes from predicted orthologs
- Construct and interpret phylogenetic trees

13.2.2.1 Introduction

In this workshop we will be walking through a comparative genomics pipeline using functions within the R package DECIPHER, and within a package that is currently under construction under the name FindHomology.

13.2.2.2 Context

Lipid membranes are ubiquitous in living organisms, however the individual lipid molecules that compose membranes have a wide variety of structures. The biosynthesis of common lipid molecules is well understood, conversely the biosynthesis of more exotic lipid molecules is not. The genomes selected today are from the archaeal phylum Thaumarchaeota, which are known for being anammox archaea, and are important in the nitrogen cycle. In addition to that, these archaea all produce a series of unusual lipids. Since these lipids are



Figure 13.1: Grand Prismatic Spring in Yellowstone National Park (3)

shared across the entire phylum from which these genomes are pulled, it is possible that the biosynthetic genes responsible for lipid construction are shared across the selected genomes, and part of their ‘core’ genome.

Archaea are relatively newly characterized as organisms. First classified in the ’70s³, for their deviations from Prokaryotes, many Archaea are notable for being isolated from extreme environments. Despite the colorful beginnings of their classification, not all Archaea are extremophiles.

The aforementioned unusual lipids⁴ are striking in their differences from those common in both Eukaryotes and Prokaryotes.

Whereas Lipid construction in eukaryotes and prokaryotes relies on iterative incorporation of malonyl-CoA units, archaea utilize the mevalonate pathway to construct their lipids through iterative incorporation of dimethylallyl pyrophosphate. Which is responsible for the iterative methyl groups extant to the lipid chain in **B** above. Archaeal lipids are also notable for their lack of ester linkages between the alkyl chain, and glycerol. Even more interesting, and as yet uncharacterized, Archaea incorporate a range of ring structures into their lipids, and additionally appear to *fuse lipid tails together in the bilayer*, as shown above in **C**.

13.2.2.3 The above is interesting, but so what?

Our goal with this workshop is to show how functions within DECIPHER, in addition to those found within FindHomology can be used to predict homologs between genomes, as well as predict the core genome of a given set of genomes. As stated above, the chemistry and biology involved in some of the construction of archaeal lipids is unknown. Hypothetically, as this chemistry is universal in these organisms, the genes responsible could all be homologs, and be identified as part of the core genome. Further we can compare the core and pan genomes of this set.

To begin with, we will load the required packages: DECIPHER and FindHomology will provide tools to complete the phylogenetic analysis, while phytools will provide visualization tools, specifically for the visualization of tanglegrams, and stringr is used for a specific string operation.

³(1) Woese, C. R.; Fox, G. E. Proc. Natl. Acad. Sci. 1977, 74 (11), 5088–5090.

⁴(1) Caforio, A.; Driessens, A. J. M. Biochimica et Biophysica Acta - Molecular and Cell Biology of Lipids. Elsevier November 1, 2017, pp 1325–1339.

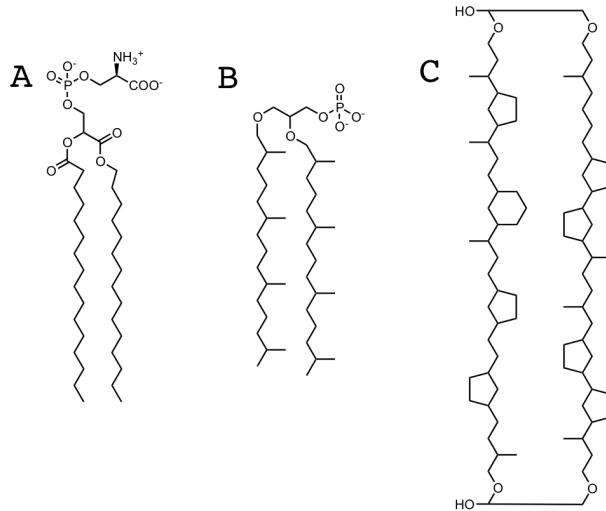


Figure 13.2: A. A generic lipid with phosphatidyl serine head group B. A generic Archaeal lipid C. A more exotic Archaeal lipid

13.2.2.4 Packages and Sequences

```
suppressMessages(library(DECIPHER))
library(FindHomology)
library(stringr)
```

13.2.2.5 Genomes, and DECIPHER databases

The first step in this process is selecting the genomes we want to work with. There are several ways to do this, especially when accessing publically available genomes through the NCBI Entrez Direct⁵ provides powerful unix command line tools, if that's your flavor, but the NCBI Genome List, and other search tools are easily accessible, and have a lower barrier to entry.

For the purposes of this workshop, we will be using a set of preselected genomes that will be accessed via ftp addresses. These are loaded with the FindHomology Package and can be accessed easily.

```
data("GeneCallAdds")
data("GenomeAdds")
data("GenomeIDs")
```

We have now loaded in a character vector for ftp addresses for genomic fasta files `GenomeAdds`, a character vector of ftp addresses for genomic GFF files `GeneCallAdds` that we will be using to collect gene boundaries, strandedness, and annotations. And lastly a vector of abbreviated names for the genomes we are working with.

FindHomology contains a simple parser function for GFF files that allows us to grab gene boundaries and annotations.

```
GeneCalls <- FindHomology::GFFParser(GFFAddress = GeneCallAdds,
                                         Verbose = TRUE)
#> =====
#> Time difference of 7.119886 secs
```

⁵Entrez Direct



Figure 13.3: Databases are compressable and easily shareable

If you choose to mimic this work on your own, there are a few ways to get gene calls(the gene boundaries, and strandedness), and annotations, such as Prodigal⁶ (for gene calls) and Prokka⁷ (for annotations). Using the NCBI annotations is simple and conducive for this workshop. However if you are working on sequence data that you have generated or collected yourself, NCBI annotations will not be available. Additionally, a Gene Caller within the DECIPHER package is currently under construction.

13.2.2.6 DECIPHER and Databases

DECIPHER works with sequence data through SQL databases. There are several reasons for this.

Databases provide large improvements in file organization. Currently, a personalized analysis pipeline could include anywhere from one to tens of programs, all of which may take in similar data, or outputs of previous steps, or may output more files than a user needs. DECIPHER's use of databases allows for sequence data to be stored in an organized format that eliminates this kind of redundancy.

Use of databases also allows for a non-destructive workflow. Sequence data in the database can be accessed as needed, but cannot be accidentally manipulated. Databases are also shareable, allowing for accession by multiple users at once.

Additionally, constructing a database in DECIPHER is relatively easy.

In your own work, it can be inadvisable to use `tempfile()` to specify the location of your .sqlite file, especially if it will be a file you use repeatedly, or takes a significant amount of time to create, or that you will eventually need to share with collaborators. However for this workshop, it is fine.

We can access our databases through either a database connection, or simply a filepath to the database. In this code chunk below, we will use a database connection.

```
DBPath <- tempfile()

DBConn <- dbConnect(SQLite(),
                     DBPath)

for (i in seq_along(GenomeAdd$)) {
  Seqs2DB(seqs = GenomeAdd$[i],
          type = "FASTA",
          dbFile = DBConn,
          identifier = as.character(i),
```

⁶(1) Hyatt, D.; Chen, G.-L.; LoCascio, P. F.; Land, M. L.; Larimer, F. W.; Hauser, L. J. BMC Bioinformatics 2010, 11 (1), 119.

⁷(1) Seemann, T. Bioinformatics 2014, 30 (14), 2068–2069.

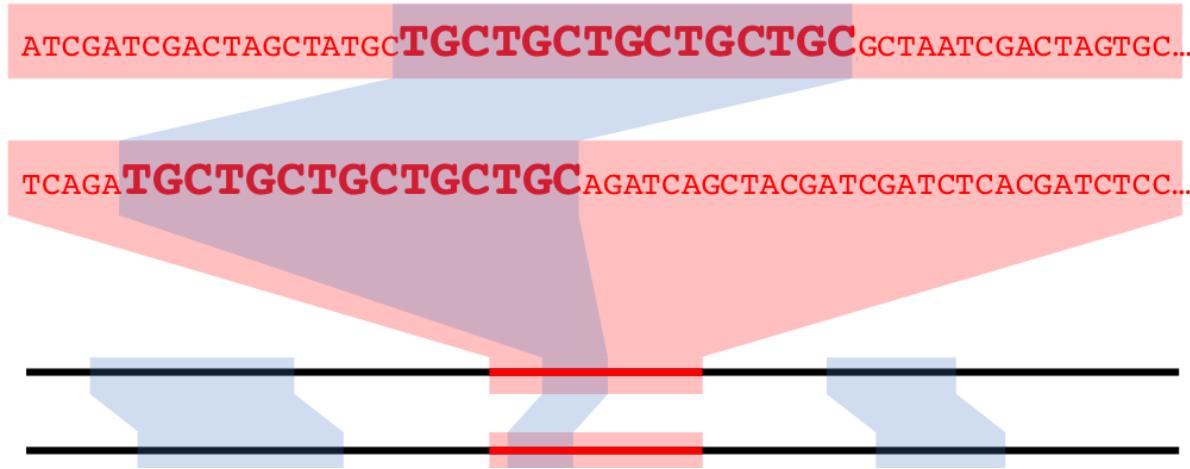


Figure 13.4: A syntenic hit in nucleotide space

```

tblName = "Seqs",
verbose = FALSE)
}

dbDisconnect(DBConn)

```

Identifying details of the database can be viewed in your default browser: `BrowseDB(DBPath)`

13.2.2.7 Comparison of genomes

Comparison of genomes will be accomplished with Synteny. Generically when regions in a genome are the same, they are syntenic matches. Often observing this phenomenon is accomplished through matching runs of reciprocal best blast hits. If in genome 1, genes A-B-C are reciprocal best blast hits to genes Z-Y-X in genome 2, the region encompassing those genes would be considered syntenic.

In DECIPHER this is accomplished using k-mer matching, runs of an exact match of length k are identified and recorded as a syntenic hit.

We can visualize syntenic hits as above, with co-linear representations of genomes with some graphical links to signify the locations and lengths of syntenic hits. Or we can employ a dotplot, which utilizes the cumulative nucleotide positions of two genomes as the x and y axes in a generic plot.

Syntenic Hits can be chained together, if they are close *enough*, to form larger syntenic blocks. The enough part, has to be measured in nucleotide distance between the hits in question, in both genomes. This also means that syntenic hits can be chained together with differing distances between them in each genome. Making syntenic blocks different lengths in corresponding genomes. Additionally, chaining of hits allows for utilizing smaller syntenic hits that may appear insignificant on their own, to be grouped in with other nearby hits to increase their significance.

DECIPHER's `FindSynteny` function allows users to control the degree to which hits and blocks are recorded through the argument `minScore`, or a minimum score required for a block, or hit, to be kept as a real syntenic match.

In this workflow, we will be determining Synteny between genomes before we consider the gene calls we have already collected. Which stands in contrast to how many current synteny finding programs work. In the code below `FindSynteny()` will perform pairways analysis of every set of genomes provided to it.

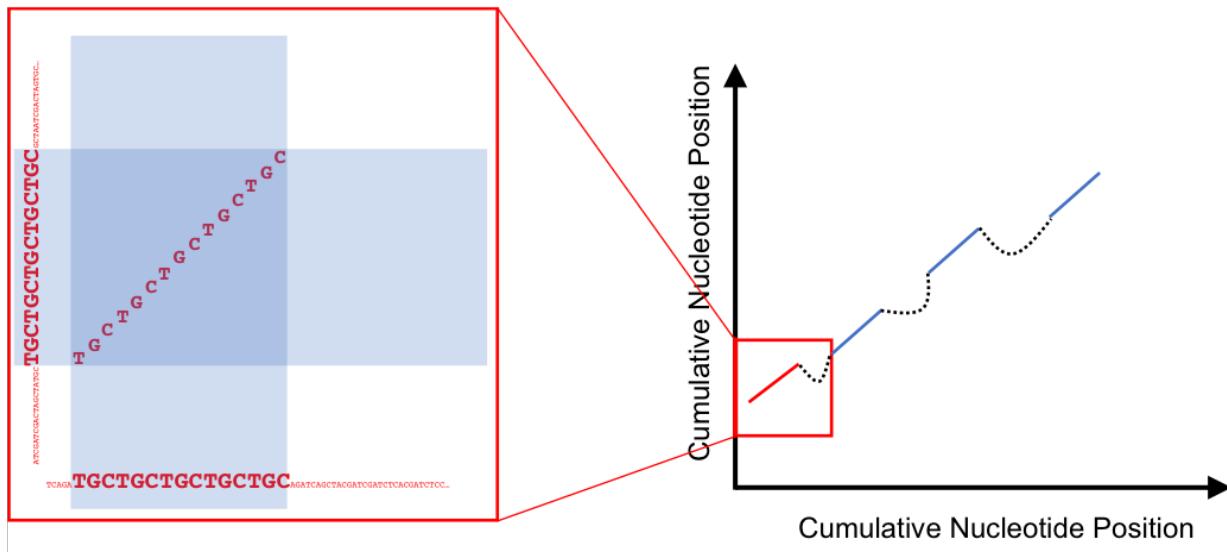


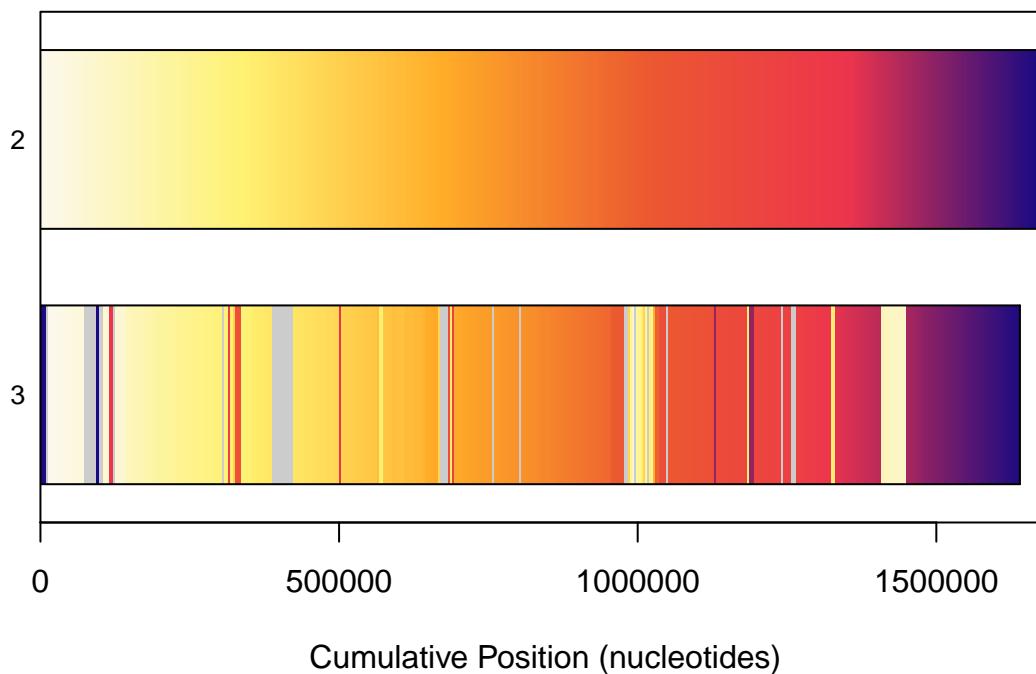
Figure 13.5: A synteny hit, and several hits being chained together into a synteny block

```
# Here we will use a file path instead of a dbconnection
SyntenyObject <- FindSynteny(dbFile = DBPath,
                                verbose = TRUE)
#> =====
#>
#> Time difference of 50.38 secs
```

There are multiple ways to view objects of class `synteny`. It has default view options for `plot()`. By default, colors are assigned as a ramp from the first nucleotide position, to the last nucleotide position of the *query* genome, or the first genome selected, and wherever synteny hits are observed, the bar representing the *subject* genome is marked with the section of the color ramp indicating the synteny hit in the *query*.

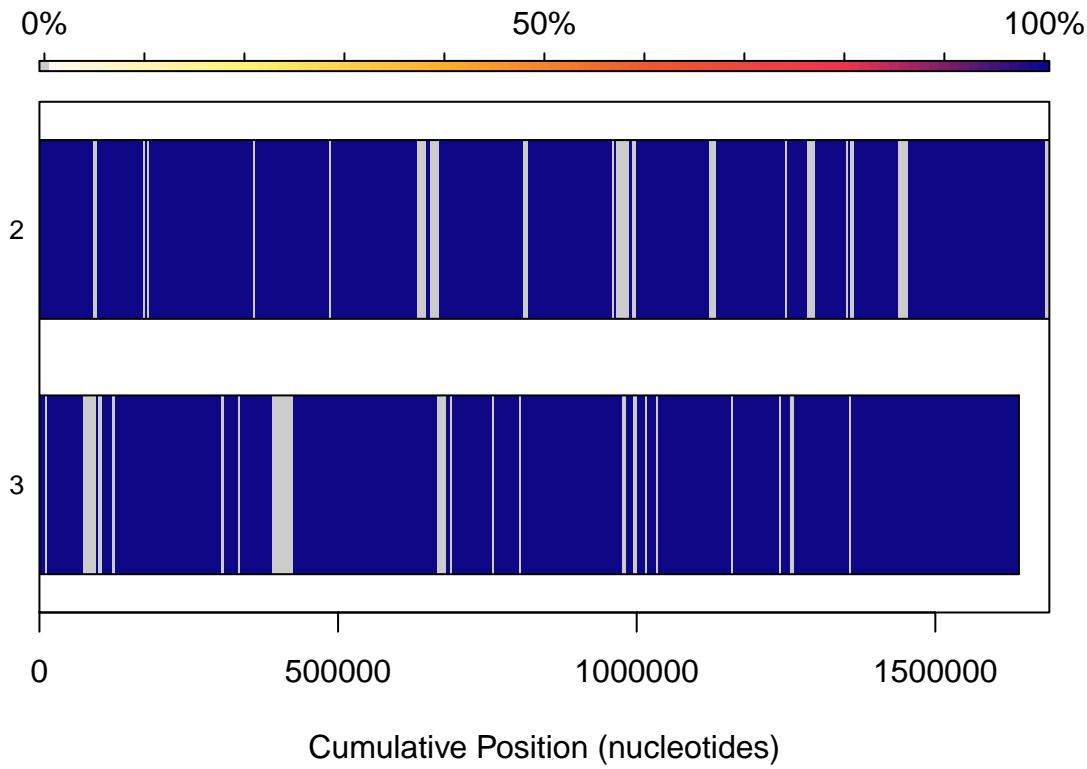
Plotting genomes 2 and 3, which are very synteny, provides a good example of this. Grey uncolored space in the subject genome indicates a lack of synteny blocks in that region. In some places, the ramp reverses, indicating an inversion, while in others, blocks are reordered from the default color bar, indicating possible rearrangements.

```
plot(SyntenyObject[2:3, 2:3])
```



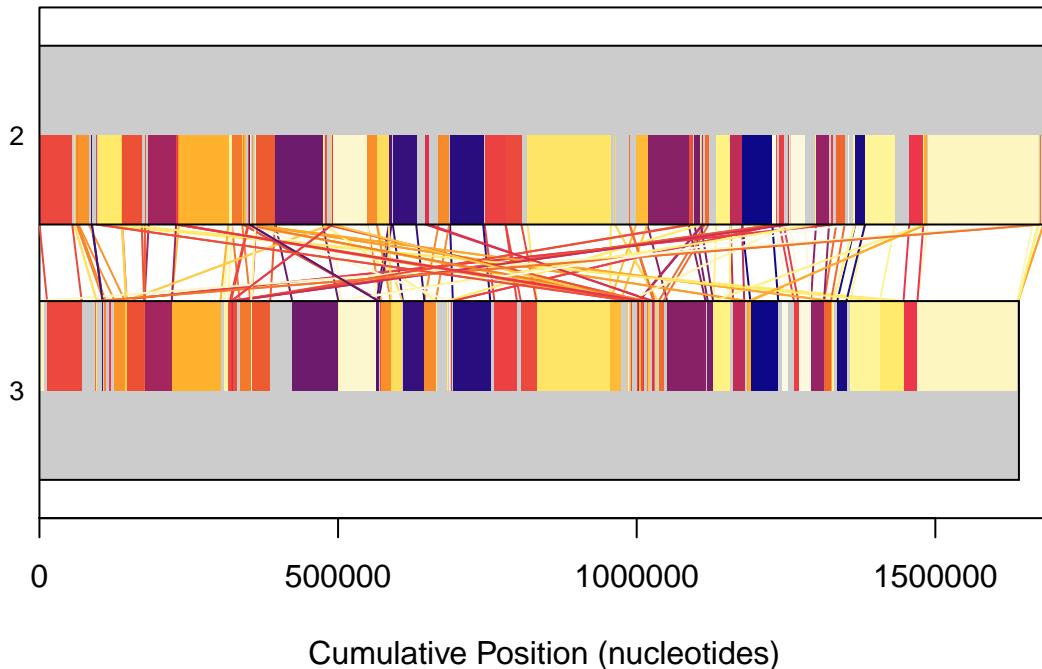
Because visualizing large numbers of syntenic hits in genome sized sequences can be non-trivial, additional commands can be passed to `plot` in order to modify coloring schemes. `frequency` causes bars representing genomes to be colored by **how syntenic** that particular region (as determined by cumulative nucleotide position) of the genome is, indiscriminate of where that hit is in a corresponding genome. Once again genomes 2 and 3 in our set are good examples of this. They are highly syntenic, which this option shows well, but the rearrangements present are no longer shown.

```
plot(SyntenyObject[2:3, 2:3],
      "frequency")
```



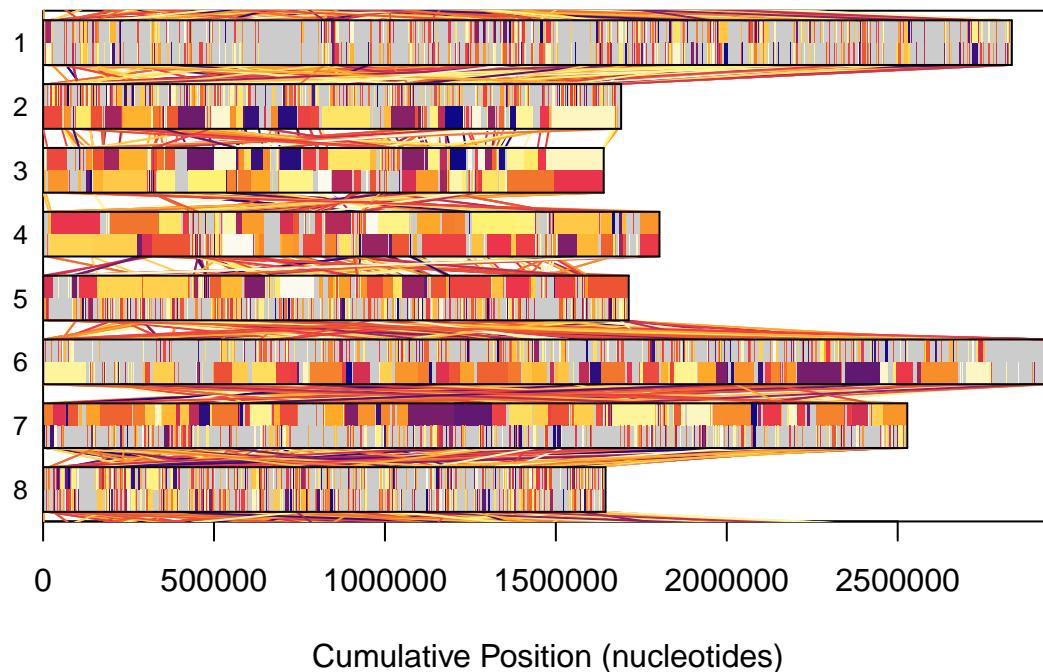
Because visualizing large numbers of syntenic hits in genome sized sequences can be non-trivial, additional commands can be passed to plot in order to modify coloring schemes. `neighbor` tries to provide each syntenic hit with a unique color, and additionally draws linear connections between them to facilitate better visualization. This can be a useful alternative to the default plotting, if many rearrangements are present in a way that you want to view.

```
plot(SyntenyObject[2:3, 2:3],
      "neighbor")
```



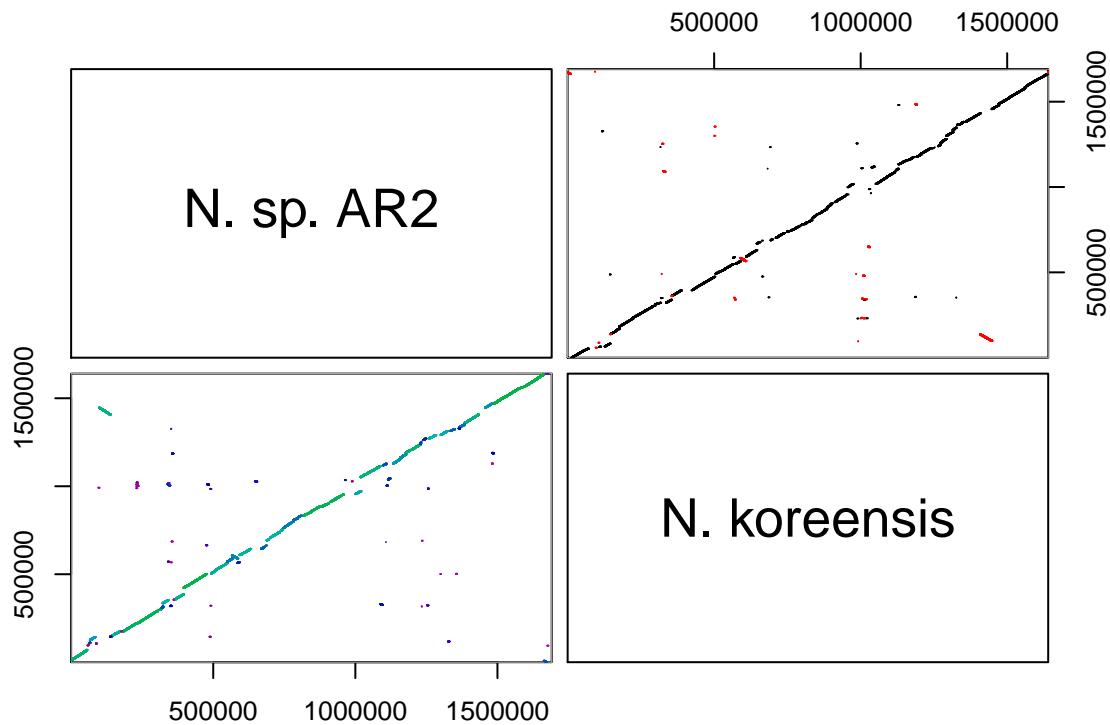
Plotting is a useful tool in analyzing sequence data, however sometimes plots of genome sized sequence data can be a bit overwhelming. Plotting our entire database, 8 sequences, though good at representing the complexity present here in this workshop, can be daunting to make sense of quickly if you are unused to looking at figures of this type frequently.

```
plot(SyntenyObject,
      "neighbor")
```



The most useful way to visualize Synteny data *in my opinion* is through a dot plot, which can be accessed via the `pairs()`. Dot plots, as explained above, utilize each genome being compared, as an axis in a normal x-y scatter plot. Where a *dot* - really a **very** short line, representing a run of nucleotides or amino acids - is shown, that x,y coordinate is a syntenic hit. Hits can also be chained together into blocks of syntenic hits.

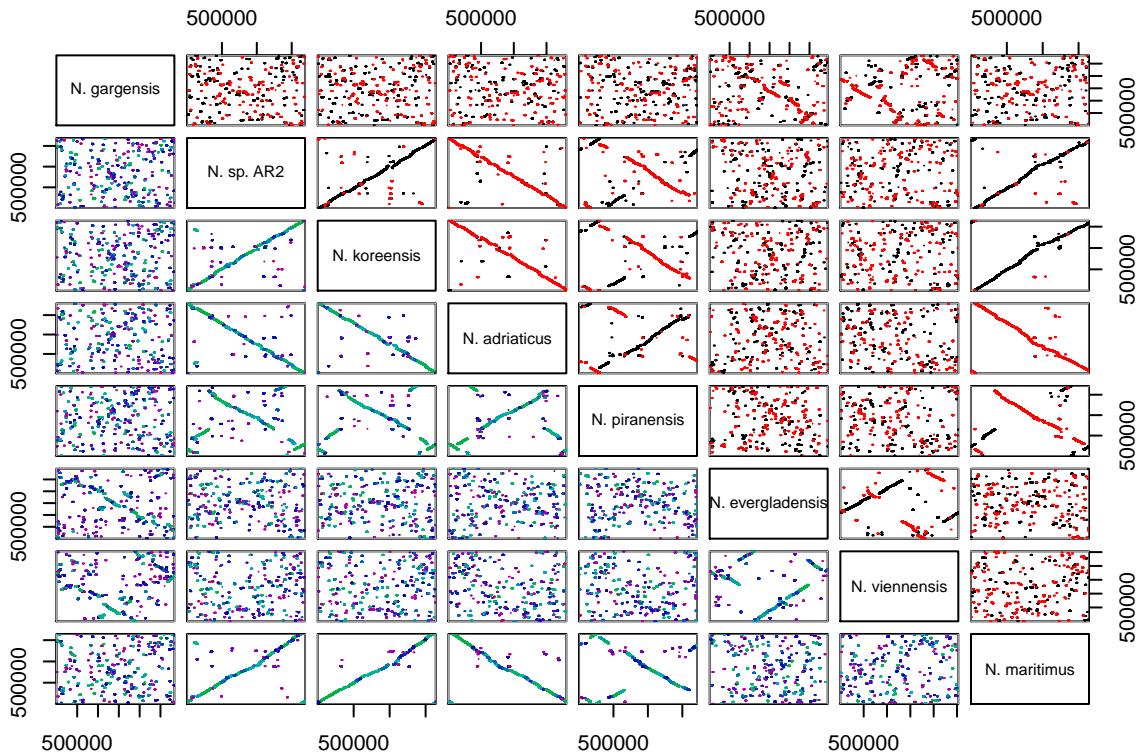
```
pairs(SyntenyObject[2:3, 2:3], labels = GenomeIDs[2:3])
```



Syntenic hits are shown in the upper left, while syntenic blocks are shown in the bottom left. In the upper triangle,hits in black are co-linear, while hits in red are inverted. In the bottom triangle, blocks are colored by score.

Pairs plots are *my* personal favorite way to visualize syntenic information, but, they can become overwhelming as the number of genomes compared increases. They provide very clear depictions of rearrangements and inversions.

```
pairs(SyntenyObject,
      labels = GenomeIDs)
```



13.2.2.8 Predict homology between genes that are linked by synteny hits

The FindHomology package contains 5 functions, the second of which is below. `NucleotideOverlap` takes in an object of class `Synteny`, and a list of gene calls. We previously generated the object `SyntenyObject`, using DECIPHER's function `FindSynteny`, and a list of gene calls and annotations using `GFFParser`. Here we will use these two objects to build what is functionally a list of pairs of genes that are linked by synteny hits. These linking of hits are used as predictions of homology. A single position in the object of class `Synteny` represents a pairwise comparison between two genomes. The corresponding position in `NucleotideOverlap`'s output object, is generated from that pairwise comparison, and the gene calls for each genome involved.

This is a fun object to scroll through, but is often very long, representing thousands of paired genes and is not represented well visually, a simple example would be:

Genome 1	Genome 2	# of Nucleotides
Gene A	Gene Z	25L
Gene B	Gene Y	1251L
Gene C	Gene X	145L
Gene ...	Gene

The reason we name this “MatrixObject” below is because the shape and dimensions of our data are dependent upon our synteny object. As in, if our Synteny object was built from 5 genomes, that object is a 5×5 matrix. `NucleotideOverlap` accesses the upper triangle of that object to build a 5×5 matrix where each position is built from data in the analogous position from the Synteny object. `MatrixObject[[1]][1, 2]` below was created from `SyntenyObject[[1]][1, 2]` and so on and so forth.

```
MatrixObject <- NucleotideOverlap(SyntenyObject = SyntenyObject,
                                  GeneCalls = GeneCalls,
                                  Verbose = TRUE)
```

```
#> =====
#> Time difference of 1.006026 mins
```

The function `Catalog` takes in the output of `NucleotideOverlap` and returns a list of matrices. These matrices represent agglomerated sets of pairs. The idea being, if gene **A** in genome **1** was paired by nucleotide overlap to genes **B** and **C** in genomes **2** and **3** respectively by `NucleotideOverLap`, and additionally gene **B** in genome **2** was similarly paired with gene **C** in genome **3**. With the assumption that these linkages indicate homology, gene **A** is predicted to be homologous to genes **B** and **C**, and gene **B** is predicted to be homologous to gene **C**.

Genome 1	Genome 2	Genome 3
Gene A	Gene B	NA
NA	Gene B	Gene C
Gene A	NA	Gene C

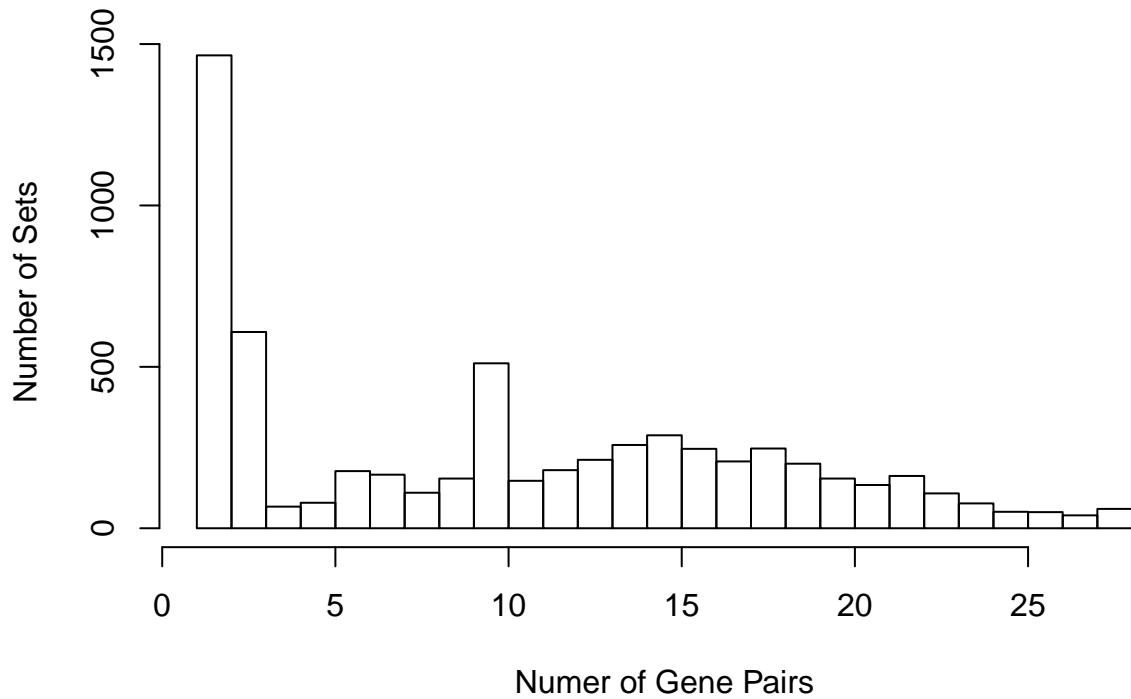
`Catalog` will return every agglomerated set of pairs, of every size possible for the given set of genomes. From a matrix with a single row, indicating a pair of genes that are predicted to have no other homologs anywhere else in the set of genomes, to a large matrix indicating that homologs are located in every genome, and are all predicted to be homologous to each other.

```
Homologs <- Catalog(MatrixObject,
                      Verbose = TRUE)
#> =====
#> Time difference of 1.886642 mins
```

We can visualize this object as a histogram of the size of these agglomerations, by the number of pairs included in each agglomerated group. Where **1** represents pairs of genes where neither gene is linked to any other gene, *from other genomes*. Where **28** *in this case* represents fully linked sets of pairs, where each gene in the set is predicted to be a homolog of *every other gene in the set*.

```
hist(sapply(Homologs,
            function(x) nrow(x)),
      main = "Size of Agglomerations",
      ylab = "Number of Sets",
      xlab = "Number of Gene Pairs",
      breaks = 27L)
```

Size of Agglomerations



We can collect these fully linked sets. For any N number of genomes, a set of genes where each genome contains a homolog, and those homologs in turn have agreeing homologs in every other genome, the number of rows of this matrix will be $N * (N-1) / 2$.

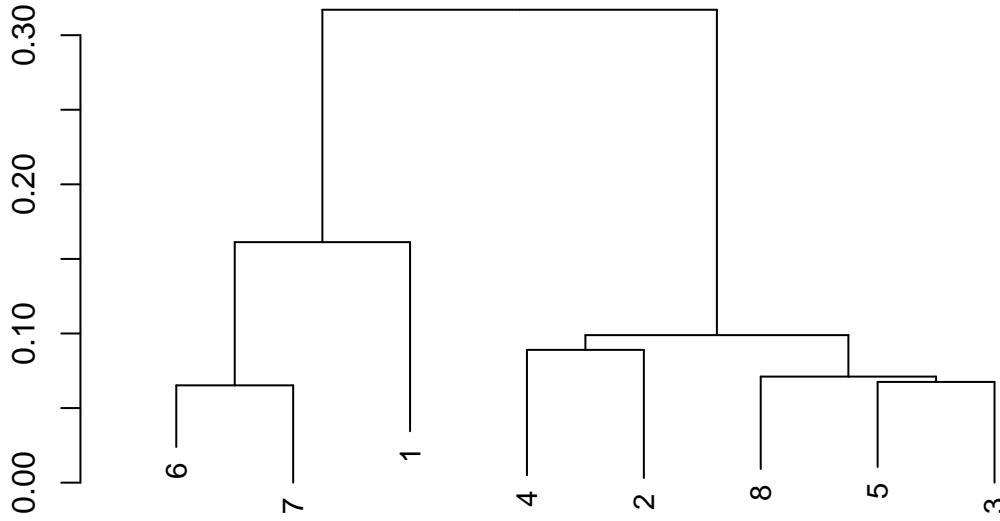
You could additionally query this list for any other presence absence pattern that you may be interested in, especially if it can be easily linked to the number of rows in each matrix.

```
MaxRows <- max(sapply(Homologs,
                       function(x) nrow(x)))
CoreSet <- which(sapply(Homologs,
                       function(x) nrow(x)) == MaxRows)
```

The second to last function in FindHomology, CoreAligner, collects the genes in these sets, from their respective genomes, aligns the respective sets, and then concatenates the alignments. Creating a concatenated core genome for this set of genomes. This function relies on a few DECIPHER functions internally, and utilizes DECIPHER's database functionality to perform that collection of genes. Additionally, we can use a few more DECIPHER functions to create a distance matrix from our concatenated core genome, and create a phylogenetic tree from that concatenated core.

```
CoreGenome <- CoreAligner(Homologs[CoreSet],
                           PATH = DBPath,
                           GeneCalls = GeneCalls,
                           Verbose = TRUE)
#> =====
#> Time difference of 33.55228 secs
CoreDist <- DistanceMatrix(myXStringSet = CoreGenome,
                           verbose = FALSE,
                           correction = "Jukes-Cantor")
CoreDend <- IdClusters(myDistMatrix = CoreDist,
                       myXStringSet = CoreGenome,
                       method = "NJ",
```

```
verbose = FALSE,
showPlot = TRUE,
type = "dendrogram")
```



We can now finally tidy up our workspace and unlink our temp file.

```
unlink(DBPath)
```

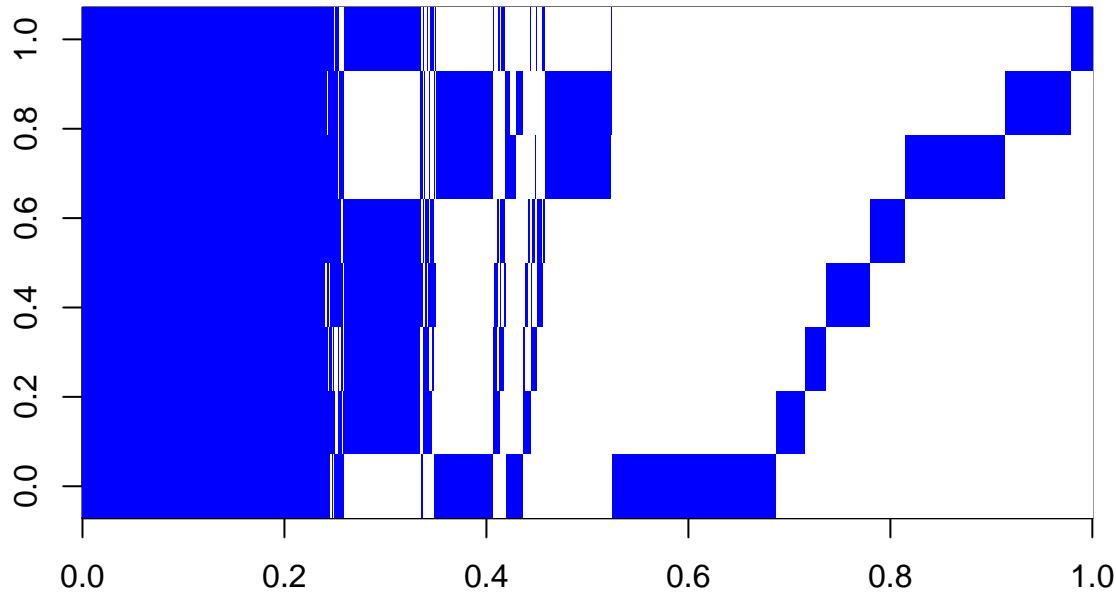
We are also interested in the pan genome, and FindHomolog's `LogicalPan` function allows us to utilize the Homologs object, and the list of GeneCall dataframes, to create a presence absence matrix of the pan genome.

```
PanGenomeMatrix <- LogicalPan(HomologList = Homologs,
                                GeneCalls = GeneCalls,
                                Verbose = TRUE,
                                Plot = FALSE)
#> =====
#> Time difference of 1.120135 secs
```

We can visualize this matrix if we so choose.

```
image(t(PanGenomeMatrix), col = c("white", "blue"),
      main = "Presence Absence")
```

Presence Absence

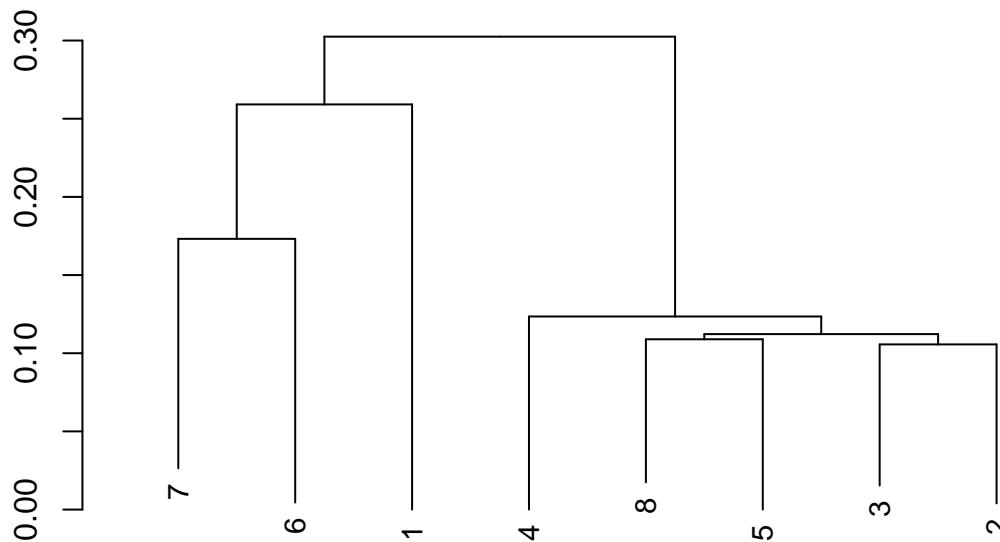


PanGenomeMatrix represents every homolog set, and every singleton gene in our genomes.

This matrix is constructed from the object Homologs which is a list of matrices. Each matrix represents of a set of predicted homologous pairs. Some of the sets are small, say a single pair. Some are large, encompassing genes in every genome in our set. However, the one thing this matrix is missing is singleton genes from each genome. Genes that have no predicted homologs by our method.

And we can create a dendrogram from this matrix.

```
PanGenome <- dist(PanGenomeMatrix,
                    method = "binary")
PanDend <- IdClusters(myDistMatrix = PanGenome,
                      method = "NJ",
                      type = "dendrogram",
                      showPlot = TRUE,
                      verbose = FALSE)
```



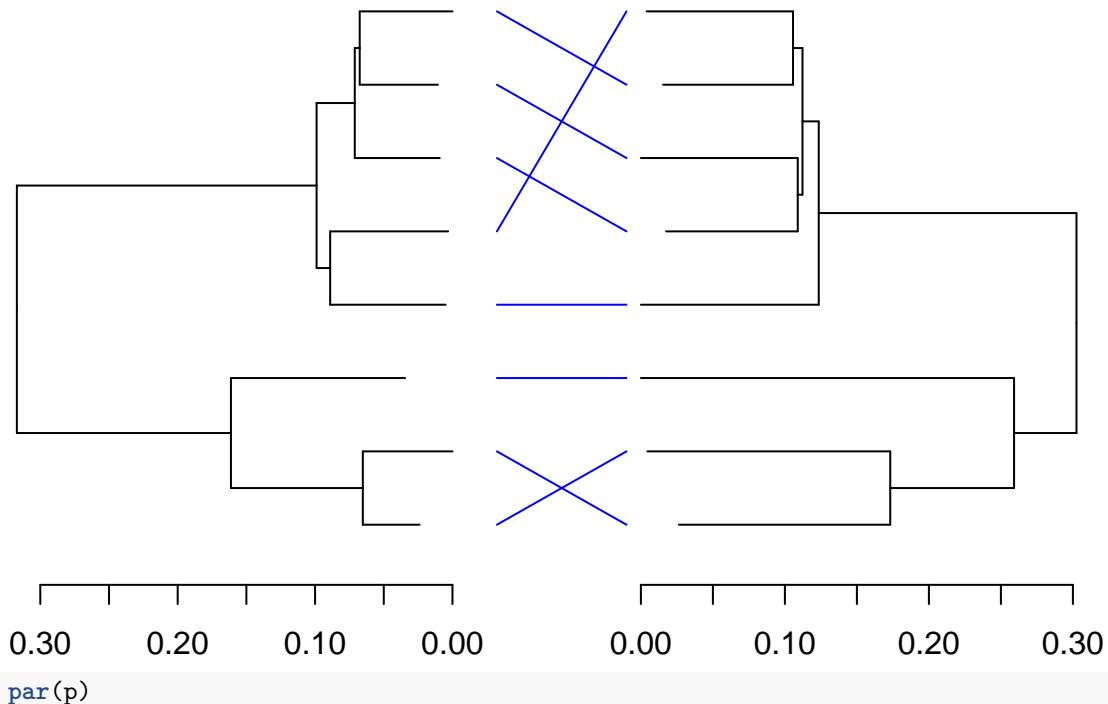
We can, additionally, create a simple tangleogram from these two phylogenetic trees. Allowing a comparison of the core, and pan genomes. The core is on the left, while the pan is on the right.

```
tf1 <- tempfile()
tf2 <- tempfile()

WriteDendrogram(x = PanDend,
                file = tf1)

WriteDendrogram(x = CoreDend,
                file = tf2)

unlink(tf1)
unlink(tf2)
layout(matrix(1:2, nrow = 1L))
p <- par(mar = c(5, 2, 1, 2))
plot(CoreDend, horiz = TRUE, leaflab = "none")
par(mar = c(5, 2, 1, 2))
plot(PanDend, horiz = TRUE, xlim = c(0, attr(PanDend, "height")), leaflab = "none")
C.Ord <- unlist(CoreDend)
P.Ord <- unlist(PanDend)
segments(-0.10, seq_along(C.Ord), -0.01, match(C.Ord, P.Ord), xpd = NA, col = "blue")
```



The proposition at the beginning of this workshop was that a specific set of genes, that may not normally be included in a core genome, could be included in the core genome, for this set of Archaea. Specifically, genes involved in some of the unusual chemistry that Archaea do with their lipids. If we wanted to really prove this, a significant amount of lab work would also be required. However we can collect the annotations for all of the sets of homologs that have been predicted, and see how much these annotations agree, and if any of these annotations provide evidence for our proposition.

```
CoreGenes <- matrix(data = NA_integer_,
                     ncol = length(GeneCalls),
```

```

        nrow = length(CoreSet))
CoreAnnotations <- matrix(data = NA_character_,
                           ncol = length(GeneCalls),
                           nrow = length(CoreSet))
for (i in seq_len(ncol(CoreGenes))) {
  for (j in seq_len(nrow(CoreGenes))) {
    CoreGenes[j, i] <- unique(Homologs[CoreSet[j]][[1]][, i])[!is.na(unique(Homologs[CoreSet[j]][[1]][,
    CoreAnnotations[j, i] <- GeneCalls[[i]][CoreGenes[j, i], "Annotation"]
  }
}

CoreAnnotations <- t(CoreAnnotations)
CoreAnnotations <- data.frame(CoreAnnotations,
                             stringsAsFactors = FALSE)

```

The annotations for each core set can now be viewed. Some provide near-universal agreement, while others indicate genes of similar family and function, but potentially not an exact match.

```

CoreAnnotations[, c(1L, 3L)]
#> X1          X3
#> 1 acetoacetyl-CoA thiolase DNA-directed RNA polymerase subunit H
#> 2 propanoyl-CoA C-acyltransferase RNA polymerase Rpb5
#> 3 propanoyl-CoA C-acyltransferase RNA polymerase Rpb5
#> 4 acetoacetyl-CoA thiolase DNA-directed RNA polymerase subunit H
#> 5 acetoacetyl-CoA thiolase DNA-directed RNA polymerase subunit H
#> 6 acetyl-CoA acetyltransferase DNA-directed RNA polymerase%2C subunit H
#> 7 acetoacetyl-CoA thiolase DNA-directed RNA polymerase subunit H
#> 8 Propanoyl-CoA C-acyltransferase RNA polymerase Rpb5

```

There are 60 of these!

```

CoreAnnotations[, 2L]
#> [1] "ribonucleoside-diphosphate reductase"
#> [2] "ribonucleoside-diphosphate reductase%2C adenosylcobalamin-dependent"
#> [3] "ribonucleoside-diphosphate reductase%2C adenosylcobalamin-dependent"
#> [4] "Ribonucleoside-diphosphate reductase"
#> [5] "Ribonucleoside-diphosphate reductase"
#> [6] "ribonucleoside-diphosphate reductase class II"
#> [7] "ribonucleoside-diphosphate reductase"
#> [8] "ribonucleoside-diphosphate reductase%2C adenosylcobalamin-dependent"

```

Now, a little insider baseball is that the unique chemistry we're interested in is often the chemistry of radicals, especially in biological systems, where the process is tightly controlled by enzymes. A quick perusal of all the sets of annotations provides two sets that support our original hypothesis.

```

CoreAnnotations[, c(17L, 31L)]
#> X17          X31
#> 1 radical SAM domain protein putative radical SAM domain protein
#> 2 radical SAM protein radical SAM protein
#> 3 radical SAM protein radical SAM protein
#> 4 Radical SAM domain protein Radical SAM protein
#> 5 hypothetical protein Radical SAM protein
#> 6 putative Fe-S oxidoreductase Fe-S oxidoreductase
#> 7 radical SAM domain protein putative radical SAM domain protein
#> 8 Radical SAM domain protein Radical SAM domain protein

```

This concludes the workshop, as any analysis after this would require work in a wet lab. Hopefully this was a useful showcase for the power of using DECIPHER to analyze sets of genomic data. The package FindHomology is currently located on github, and is still under construction. Upon completion, it will be available on Bioconductor.

Thank you for your time and attention! We hope you learned something, and if you come back to this at a later date and have any questions, concerns, or comments, feel free to email me, or you can tweet at me.

Chapter 14

260: Biomarker discovery from large pharmacogenomics datasets

14.1 Instructors:

- Zhaleh Safikhani (zhaleh.safikhani@utoront.ca)
- Petr Smirnov (petr.smirnov@mail.utoronto.ca)
- Benjamin Haibe-Kains (benjamin.haibe.kains@utoronto.ca)

14.2 Workshop Description

This workshop will focus on the challenges encountered when applying machine learning techniques in complex, high dimensional biological data. In particular, we will focus on biomarker discovery from pharmacogenomic data, which consists of developing predictors of response of cancer cell lines to chemical compounds based on their genomic features. From a methodological viewpoint, biomarker discovery is strongly linked to variable selection, through methods such as Supervised Learning with sparsity inducing norms (e.g., ElasticNet) or techniques accounting for the complex correlation structure of biological features (e.g., mRMR). Yet, the main focus of this talk will be on sound use of such methods in a pharmacogenomics context, their validation and correct interpretation of the produced results. We will discuss how to assess the quality of both the input and output data. We will illustrate the importance of unified analytical platforms, data and code sharing in bioinformatics and biomedical research, as the data generation process becomes increasingly complex and requires high level of replication to achieve robust results. This is particularly relevant as our portfolio of machine learning techniques is ever enlarging, with its set of hyperparameters that can be tuning in a multitude of ways, increasing the risk of overfitting when developing multivariate predictors of drug response.

14.2.1 Pre-requisites

- Basic knowledge of R syntax
- Familiarity with the machine learning concept and at least a few approaches

Following resources might be useful to read:

- <https://academic.oup.com/bioinformatics/article-lookup/doi/10.1093/bioinformatics/btv723>
- <https://academic.oup.com/nar/article/46/D1/D994/4372597>
- <https://web.stanford.edu/~hastie/Papers/ESLII.pdf>

14.2.2 Workshop Participation

Participants expected to have the following required packages installed on their machines to be able to run the commands along with the instructors.

- * PharmacoGx and Biobase from Bioconductor
- * xtable, Hmisc, foreach, devtools, mRMRe, caret, glmnet, randomForest from cran
- * bhklab/mci and bhklab/PharmacoGx-ML from github

14.2.3 *R / Bioconductor* packages used

- <https://bioconductor.org/packages/release/bioc/html/PharmacoGx.html>

14.2.4 Time outline

An example for a 45-minute workshop:

Activity	Time
Introduction	10m
Basic functionalities of PharmacoGx	15m
Consistency assessment between datasets	15m
Machine learning and biomarker discovery	20m

14.3 Workshop goals and objectives

14.3.1 Learning goals

- describe the pharmacogenomic datasets and their usefulness
- learn how to extract information from these datasets and to intersect them over their common features
- identify functionalities available in PharmacoGx package to work with the high dimensional pharmacogenomics data
- assess reproducibility and replication of pharmacogenomics studies
- understand how to handle the biomarker discovery as a pattern recognition problem in the domain of pharmacogenomics studies

14.3.2 Learning objectives

- list available standardized pharmacogenomic datasets and download them
- understand the structure of these datasets and how to access the features and response quantifications
- create drug-dose response plots
- Measure the consistency across multiple datasets and how to improve such measurements
- Assess whether known biomarkers are reproduced within these datasets
- Predict new biomarkers by applying different machine learning methods

14.4 Abstract

This course will focus on the challenges encountered when applying machine learning techniques in complex, high dimensional biological data. In particular, we will focus on biomarker discovery from pharmacogenomic data, which consists of developing predictors of response of cancer cell lines to chemical compounds based on

their genomic features. From a methodological viewpoint, biomarker discovery is strongly linked to variable selection, through methods such as Supervised Learning with sparsity inducing norms (e.g., ElasticNet) or techniques accounting for the complex correlation structure of biological features (e.g., mRMR). Yet, the main focus of this talk will be on sound use of such methods in a pharmacogenomics context, their validation and correct interpretation of the produced results. We will discuss how to assess the quality of both the input and output data. We will illustrate the importance of unified analytical platforms, data and code sharing in bioinformatics and biomedical research, as the data generation process becomes increasingly complex and requires high level of replication to achieve robust results. This is particularly relevant as our portfolio of machine learning techniques is ever enlarging, with its set of hyperparameters that can be tuning in a multitude of ways, increasing the risk of overfitting when developing multivariate predictors of drug response.

14.5 Introduction

Pharmacogenomics holds much potential to aid in discovering drug response biomarkers and developing novel targeted therapies, leading to development of precision medicine and working towards the goal of personalized therapy. Several large experiments have been conducted, both to molecularly characterize drug dose response across many cell lines, and to examine the molecular response to drug administration. However, the experiments lack a standardization of protocols and annotations, hindering meta-analysis across several experiments.

PharmacoGx was developed to address these challenges, by providing a unified framework for downloading and analyzing large pharmacogenomic datasets which are extensively curated to ensure maximum overlap and consistency.

PharmacoGx is based on a level of abstraction from the raw experimental data, and allows bioinformaticians and biologists to work with data at the level of genes, drugs and cell lines. This provides a more intuitive interface and, in combination with unified curation, simplifies analyses between multiple datasets.

Load *PharmacoGx* into your current workspace:

```
suppressPackageStartupMessages({
  library(PharmacoGx, verbose=FALSE)
  library(mCI, verbose=FALSE)
  library(PharmacoGxML, verbose=FALSE)
  library(Biobase, verbose=FALSE)
})
#> Warning in fun(libname, pkgname): couldn't connect to display ":0"
```

14.5.1 Downloading PharmacoSet objects

We have made the PharmacoSet objects of the curated datasets available for download using functions provided in the package. A table of available PharmacoSet objects can be obtained by using the *availablePSets* function. Any of the PharmacoSets in the table can then be downloaded by calling *downloadPSet*, which saves the datasets into a directory of the users choice, and returns the data into the R session.

```
availablePSets(saveDir=file.path(".", "Safikhani_Pharmacogenomics"))
GDSC <- downloadPSet("GDSC", saveDir=file.path(".", "Safikhani_Pharmacogenomics"))
CCLE <- downloadPSet("CCLE", saveDir=file.path(".", "Safikhani_Pharmacogenomics"))
```

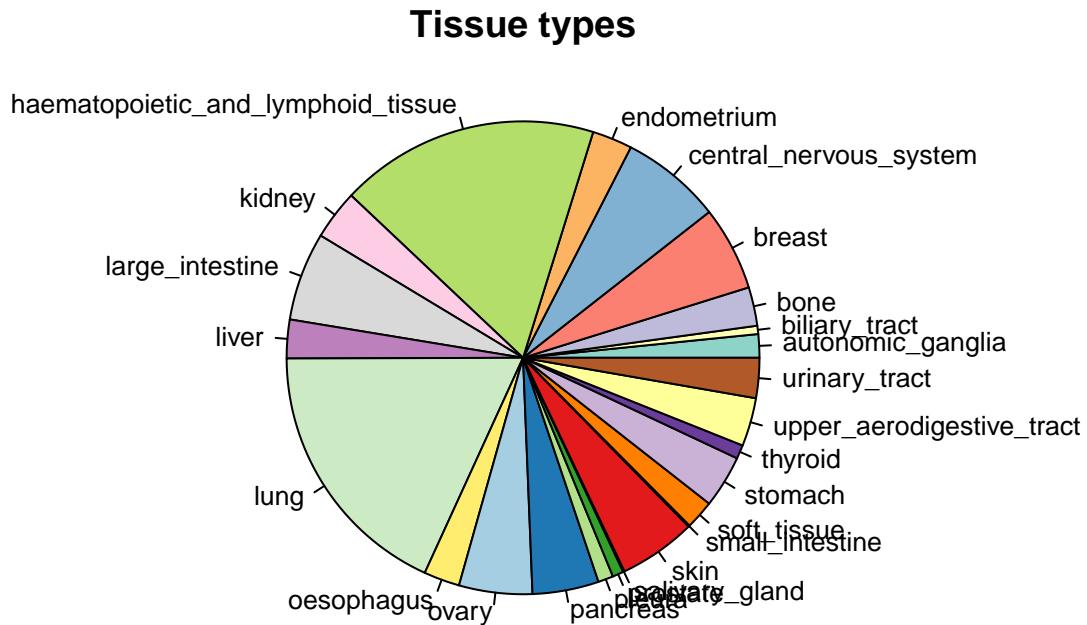


Figure 14.1: Tissue of origin of cell lines in CCLE study

14.6 Reproducibility

PharmacoGx can be used to process pharmacogenomic datasets. First we want to check the heterogeneity of cell lines in one of the available psets, CCLE.

```
mycol <- c("#8dd3c7", "#ffffb3", "#bebada", "#fb8072", "#80b1d3", "#fdb462",
          "#b3de69", "#fccde5", "#d9d9d9", "#bc80bd", "#ccebc5", "#ffed6f",
          "#a6cee3", "#1f78b4", "#b2df8a", "#33a02c", "#fb9a99", "#e31a1c",
          "#fdbf6f", "#ff7f00", "#cab2d6", "#6a3d9a", "#ffff99", "#b15928")
pie(table(CCLE@cell[,"tissueid"]),
    col=mycol,
    main="Tissue types",
    radius=1,
    cex=0.8)
```

14.6.1 Plotting Drug-Dose Response Data

Drug-Dose response data included in the *PharmacoSet* objects can be conveniently plotted using the *drugDoseResponseCurve* function. Given a list of *PharmacoSets*, a drug name and a cell name, it will plot the drug dose response curves for the given cell-drug combination in each dataset, allowing direct comparisons of data between datasets.

```
CCLE.auc <- summarizeSensitivityProfiles(
  pSet=CCLE,
  sensitivity.measure="auc_published",
  summary.stat="median",
  verbose=FALSE)

lapatinib.aac <- CCLE.auc["lapatinib",]
cells <- names(lapatinib.aac)[
```

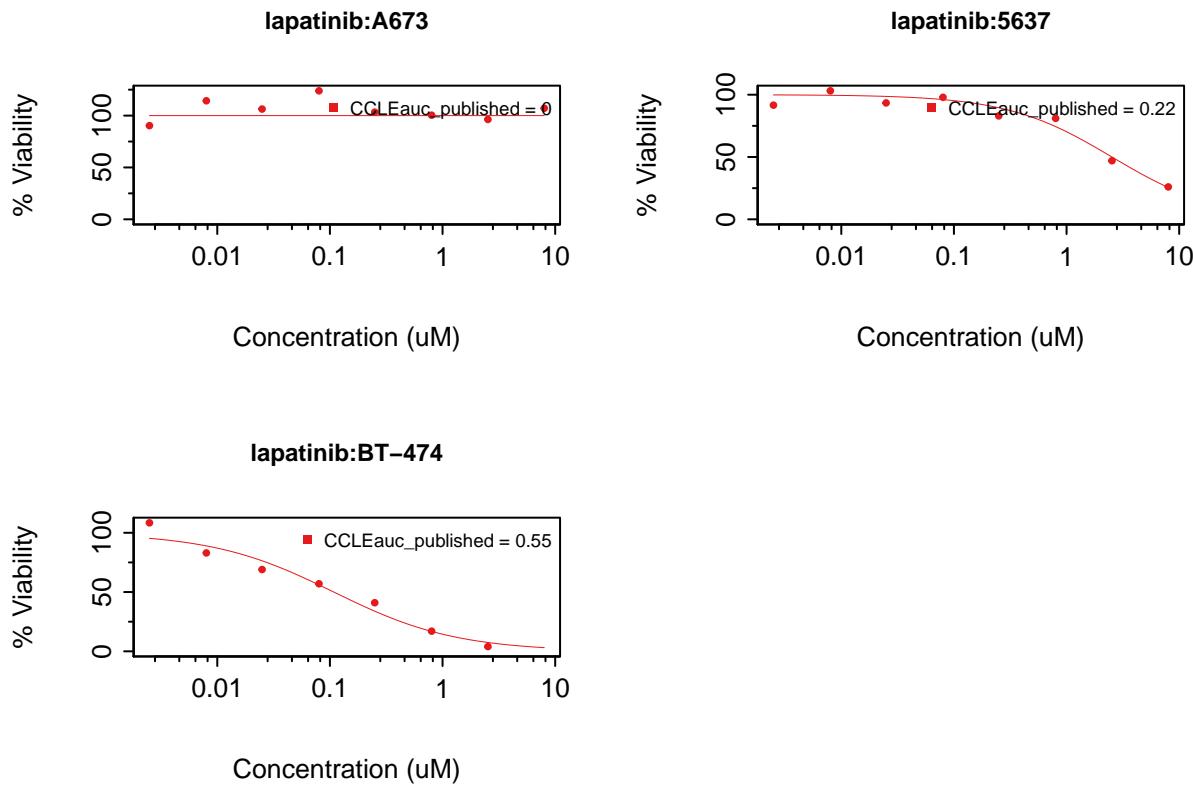


Figure 14.2: Cells response to lapatinib in CCLE

```

c(which.min(lapatinib.aac),
  which((lapatinib.aac > 0.2) & (lapatinib.aac < 0.4))[1],
  which.max(lapatinib.aac))

par(mfrow=c(2, 2))
drugDoseResponseCurve(drug="lapatinib", cellline=cells[1],
                      pSets=CCLE, plot.type="Fitted",
                      legends.label="auc_published")
drugDoseResponseCurve(drug="lapatinib", cellline=cells[2],
                      pSets=CCLE, plot.type="Fitted",
                      legends.label="auc_published")
drugDoseResponseCurve(drug="lapatinib", cellline=cells[3],
                      pSets=CCLE, plot.type="Fitted",
                      legends.label="auc_published")

```

14.6.2 Pharmacological profiles

In pharmacogenomic studies, cancer cell lines were also tested for their response to increasing concentrations of various compounds, and from this the IC₅₀ and AAC were computed. These pharmacological profiles are available for all the psets in *PharmacoGx*.

```

library(ggplot2, verbose=FALSE)
library(reshape2, verbose=FALSE)
melted_data <- melt(CCLE.auc)
NA_rows <- unique(which(is.na(melted_data), arr.ind=T)[,1])
melted_data <- melted_data[-NA_rows,]

```

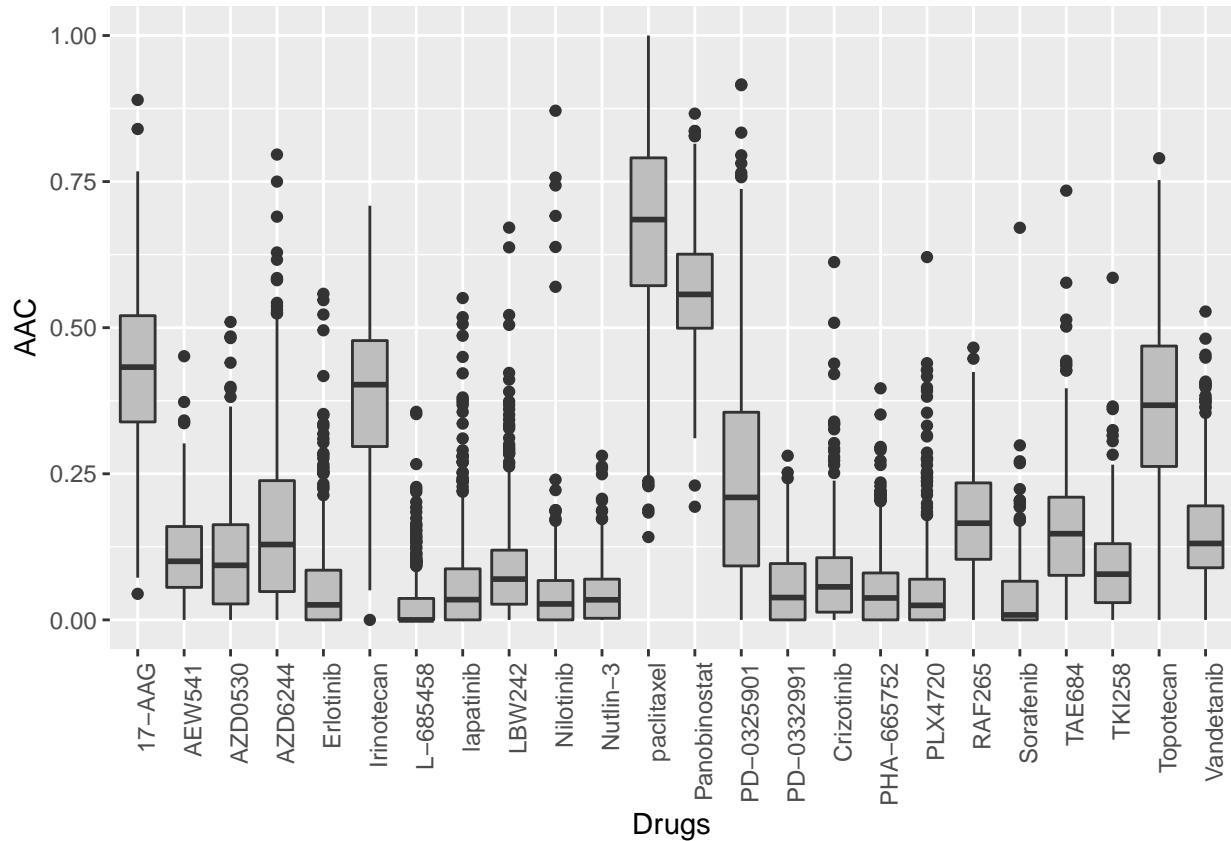


Figure 14.3: Cells response to drugs in CCLE

```
ggplot(melted_data, aes(x=Var1,y=value)) +
  geom_boxplot(fill="gray") +
  theme(axis.text.x=element_text(angle=90,hjust=1)) +
  xlab("Drugs") +
  ylab("AAC")

#hist(CCLE.auc["lapatinib"], xlab="Cells response to lapatinib(AAC)",
#      col="gray", main="")
```

14.7 Replication

In this section we will investigate the consistency between the GDSC and CCLE datasets. In both CCLE and GDSC, the transcriptome of cells was profiled using an Affymatrix microarray chip. Cells were also tested for their response to increasing concentrations of various compounds, and from this the IC₅₀ and AUC were computed. However, the cell and drugs names used between the two datasets were not consistent. Furthermore, two different microarray platforms were used. However, *PharmacoGx* allows us to overcome these differences to do a comparative study between these two datasets.

GDSC was profiled using the hgu133a platform, while CCLE was profiled with the expanded hgu133plus2 platform. While in this case the hgu133a is almost a strict subset of hgu133plus2 platform, the expression information in *PharmacoSet* objects is summarized by Ensemble Gene Ids, allowing datasets with different platforms to be directly compared. The probe to gene mapping is done using the BrainArray customCDF for

each platform [?].

To begin, you would load the datasets from disk or download them using the *downloadPSet* function above.

We want to investigate the consistency of the data between the two datasets. The common intersection between the datasets can then be found using *intersectPSet*. We create a summary of the gene expression and drug sensitivity measures for both datasets, so we are left with one gene expression profile and one sensitivity profile per cell line within each dataset. We can then compare the gene expression and sensitivity measures between the datasets using a standard correlation coefficient.

```
common <- intersectPSet(pSets = list("CCLE"=CCLE, "GDSC"=GDSC),
                         intersectOn = c("cell.lines", "drugs"),
                         strictIntersect = TRUE)
#> Intersecting large PSets may take a long time ...
drugs <- drugNames(common$CCLE)

##Example of concordant and discordant drug curves
cases <- rbind(
  c("CAL-85-1", "17-AAG"),
  c("HT-29", "PLX4720"),
  c("COLO-320-HSR", "AZD6244"),
  c("HT-1080", "PD-0332991"))

par(mfrow=c(2, 2))
for (i in seq_len(nrow(cases))) {
  drugDoseResponseCurve(pSets=common,
                        drug=cases[i,2],
                        cellline=cases[i,1],
                        legends.label="ic50_published",
                        plot.type="Fitted",
                        ylim=c(0,130))
}
```

14.7.1 Consistency of pharmacological profiles

```
##AAC scatter plot
GDSC.aac <- summarizeSensitivityProfiles(
  pSet=common$GDSC,
  sensitivity.measure='auc_recomputed',
  summary.stat="median",
  verbose=FALSE)
CCLE.aac <- summarizeSensitivityProfiles(
  pSet=common$CCLE,
  sensitivity.measure='auc_recomputed',
  summary.stat="median",
  verbose=FALSE)

GDSC.ic50 <- summarizeSensitivityProfiles(
  pSet=common$GDSC,
  sensitivity.measure='ic50_recomputed',
  summary.stat="median",
  verbose=FALSE)
CCLE.ic50 <- summarizeSensitivityProfiles(
```

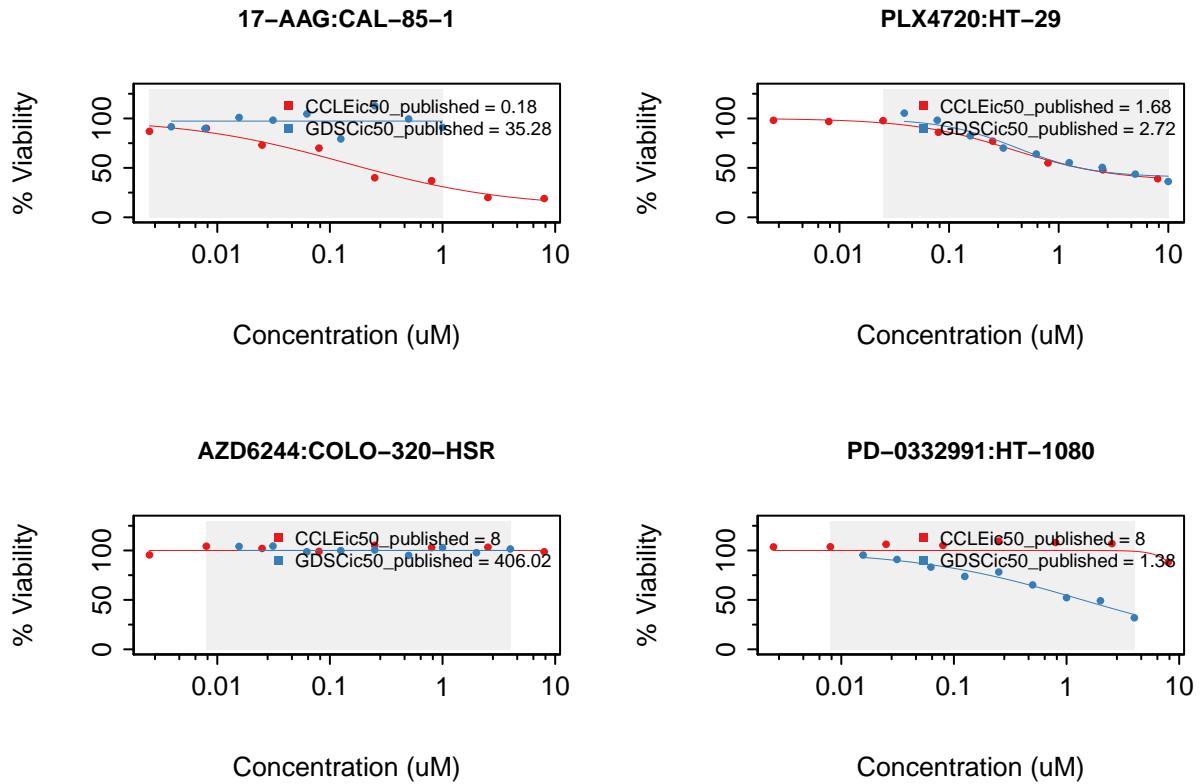


Figure 14.4: Consistency of drug response curves across studies

```

pSet=common$CCLE,
sensitivity.measure='ic50_recomputed',
summary.stat="median",
verbose=FALSE)

drug <- "lapatinib"
#par(mfrow=c(1, 2))
myScatterPlot(x=GDSC.aac[drug,],
              y=CCLE.aac[drug,],
              method=c("transparent"),
              transparency=0.8, pch=16, minp=50,
              xlim=c(0, max(max(GDSC.aac[drug,], na.rm=T), max(CCLE.aac[drug,], na.rm=T))),
              ylim=c(0, max(max(GDSC.aac[drug,], na.rm=T), max(CCLE.aac[drug,], na.rm=T))),
              main="cells response to lapatinib",
              cex.sub=0.7,
              xlab="AAC in GDSC",
              ylab="AAC in CCLE")
legend("topright",
       legend=sprintf("r=%s\nnrs=%s\nnCI=%s",
                     round(cor(GDSC.aac[drug,],
                               CCLE.aac[drug,],
                               method="pearson",
                               use="pairwise.complete.obs"),
                           digits=2),
                     round(cor(GDSC.aac[drug,],
                               CCLE.aac[drug,],
                               method="spearman",
                               use="pairwise.complete.obs"),
                           digits=2)))

```

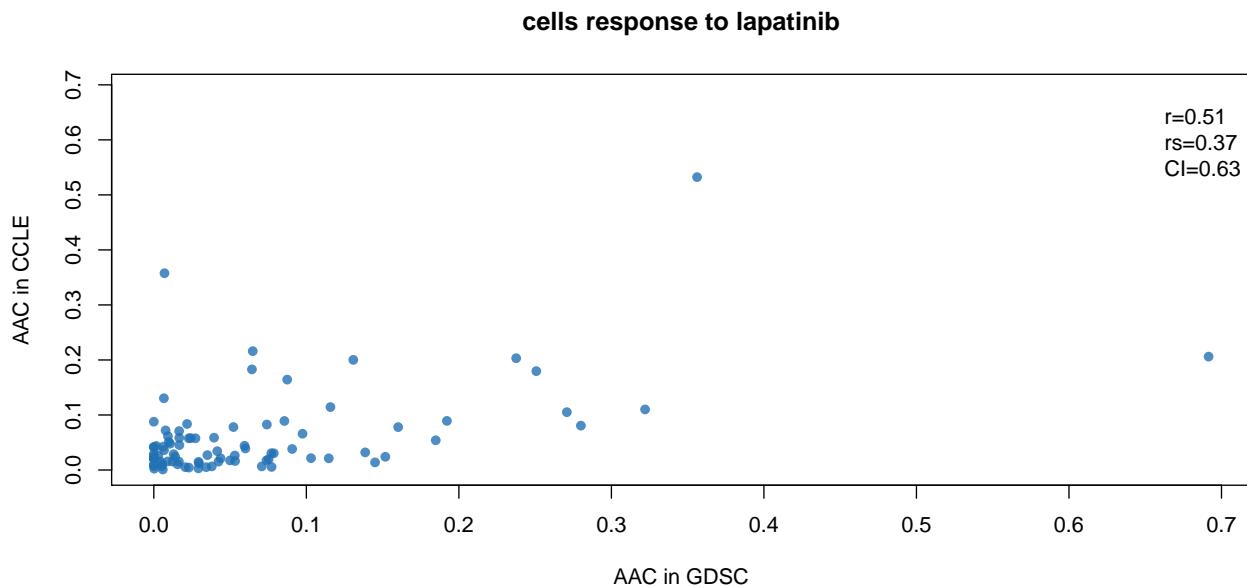


Figure 14.5: Concordance of AAC values

```

CCLE.aac[drug,],
method="spearman",
use="pairwise.complete.obs"),
digits=2),
round(paired.concordance.index(GDSC.aac[drug,],
                                   CCLE.aac[drug,],
                                   delta.pred=0,
                                   delta.obs=0)$cindex,
      digits=2)),
bty="n")

```

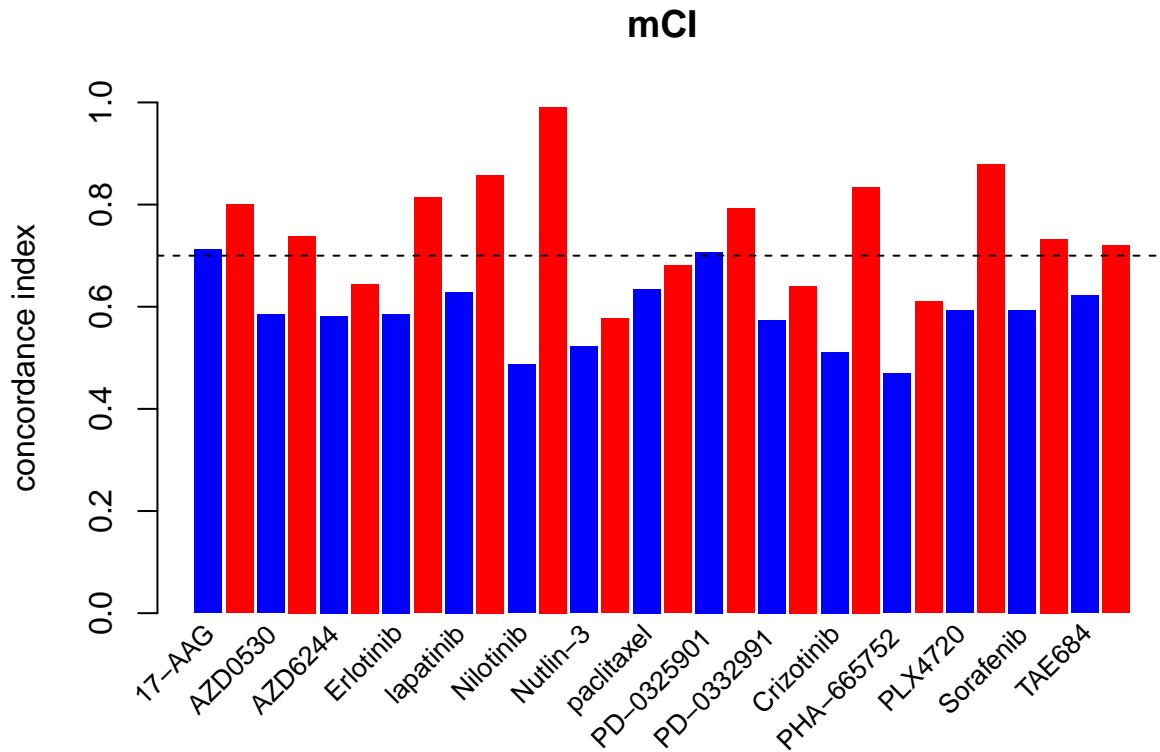
14.7.2 consistency assessment improved by Modified Concordance Index

To better assess the concordance of multiple pharmacogenomic studies we introduced the modified concordance index (mCI). Recognizing that the noise in the drug screening assays is high and may yield to inaccurate sensitive-based ranking of cell lines with close AAC values, the mCI only considers cell line pairs with drug sensitivity (AAC) difference greater than δ .

```

c_index <- mc_index <- NULL
for(drug in drugs){
  tt <- mCI::paired.concordance.index(GDSC.aac[drug,], CCLE.aac[drug,], delta.pred=0, delta.obs=0, alter...
  c_index <- c(c_index, tt$cindex)
  tt <- mCI::paired.concordance.index(GDSC.aac[drug,], CCLE.aac[drug,], delta.pred=0.2, delta.obs=0.2, ...
  mc_index <- c(mc_index, tt$cindex)
}
mp <- barplot(as.vector(rbind(c_index, mc_index)), beside=TRUE, col=c("blue", "red"), ylim=c(0, 1), yla...
text(mp, par("usr")[3], labels=as.vector(rbind(drugs, rep("", 15))), srt=45, adj=c(1.1,1.1), xpd=TRUE, ...
abline(h=.7, lty=2)

```



14.7.3 Known Biomarkers

The association between molecular features and response to a given drug is modelled using a linear regression model adjusted for tissue source:

$$Y = \beta_0 + \beta_i G_i + \beta_t T + \beta_b B$$

where Y denotes the drug sensitivity variable, G_i , T and B denote the expression of gene i , the tissue source and the experimental batch respectively, and β s are the regression coefficients. The strength of gene-drug association is quantified by β_i , above and beyond the relationship between drug sensitivity and tissue source. The variables Y and G are scaled (standard deviation equals to 1) to estimate standardized coefficients from the linear model. Significance of the gene-drug association is estimated by the statistical significance of β_i (two-sided t test). P-values are then corrected for multiple testing using the false discovery rate (FDR) approach.

As an example of the reproducibility of biomarker discovery across pharmacogenomic studies, we can model the significance of the association between two drugs and their known biomarkers in CCLE and GDSC. We examine the association between drug *17-AAG* and gene *NQO1*, as well as drug *PD-0325901* and gene *BRAF*:

```
features <- Pharmacogenomics::fNames(CCLE, "rna")[
  which(featureInfo(CCLE,
    "rna")$Symbol == "NQO1")]
ccle.sig.rna <- drugSensitivitySig(pSet=CCLE,
  mDataType="rna",
  drugs=c("17-AAG"),
  features=features,
  sensitivity.measure="auc_published",
  molecular.summary.stat="median",
  sensitivity.summary.stat="median",
  verbose=FALSE)
```

```

gdsc.sig.rna <- drugSensitivitySig(pSet=GDSC,
                                      mDataType="rna",
                                      drugs=c("17-AAG"),
                                      features=features,
                                      sensitivity.measure="auc_published",
                                      molecular.summary.stat="median",
                                      sensitivity.summary.stat="median",
                                      verbose=FALSE)
ccle.sig.mut <- drugSensitivitySig(pSet=CCLE,
                                      mDataType="mutation",
                                      drugs=c("PD-0325901"),
                                      features="BRAF",
                                      sensitivity.measure="auc_published",
                                      molecular.summary.stat="and",
                                      sensitivity.summary.stat="median",
                                      verbose=FALSE)
gdsc.sig.mut <- drugSensitivitySig(pSet=GDSC,
                                      mDataType="mutation",
                                      drugs=c("PD-0325901"),
                                      features="BRAF",
                                      sensitivity.measure="auc_published",
                                      molecular.summary.stat="and",
                                      sensitivity.summary.stat="median",
                                      verbose=FALSE)
ccle.sig <- rbind(ccle.sig.rna, ccle.sig.mut)
gdsc.sig <- rbind(gdsc.sig.rna, gdsc.sig.mut)
known.biomarkers <- cbind("GDSC effect size"=gdsc.sig[,1],
                           "GDSC pvalue"=gdsc.sig[,6],
                           "CCLE effect size"=ccle.sig[,1],
                           "CCLE pvalue"=ccle.sig[,6])
rownames(known.biomarkers) <- c("17-AAG + NQ01", "PD-0325901 + BRAF")
library(xtable, verbose=FALSE)
xtable(known.biomarkers, digits=c(0, 2, -1, 2, -1), caption='Concordance of biomarkers across stuud')
par(mfrow=c(2, 2))
CCLE_expr <- t(exprs(summarizeMolecularProfiles(CCLE, mDataType="rna", fill.missing=FALSE)))
CCLE_cells <- intersect(rownames(CCLE_expr), colnames(CCLE.aac))
plot(CCLE.aac["17-AAG", CCLE_cells], CCLE_expr[CCLE_cells, features],
     main="CCLE + 17-AAG + NQ01",
     cex.main=1, ylab="Predictions", xlab="drug sensitivity", pch=20, col="gray40")

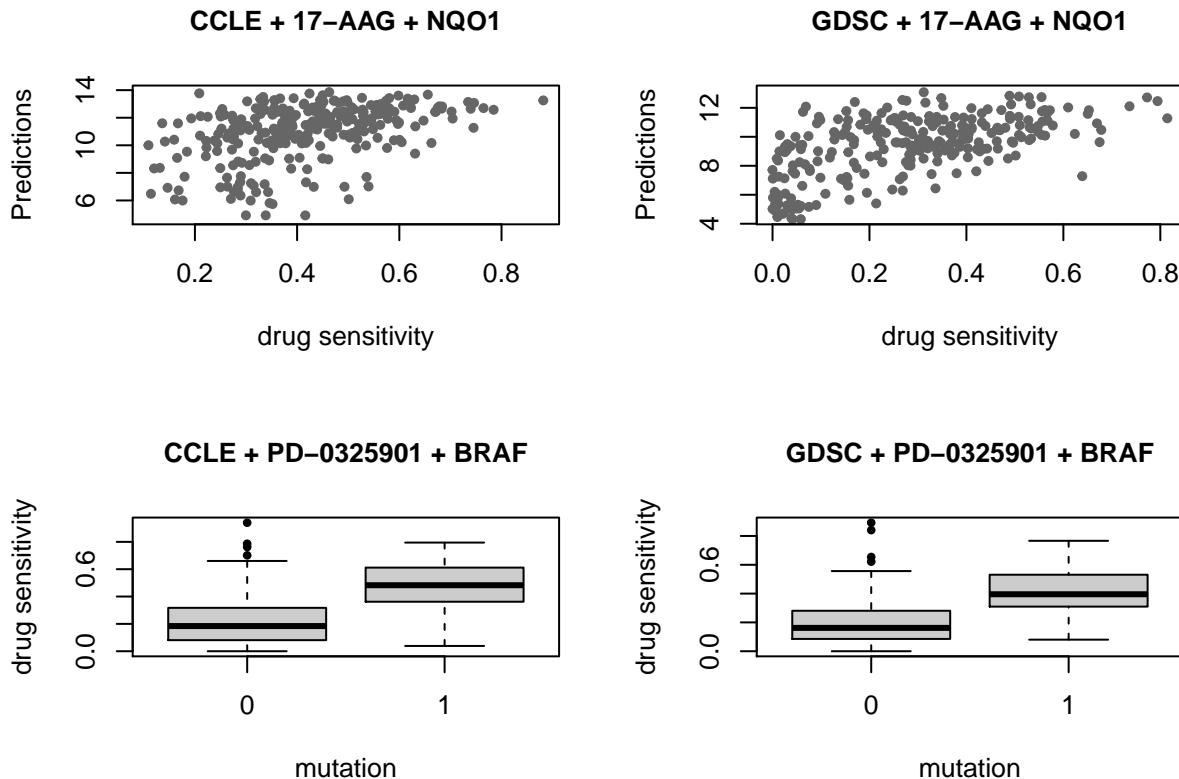
GDSC_expr <- t(exprs(summarizeMolecularProfiles(GDSC, mDataType="rna", fill.missing=FALSE)))
#> Summarizing rna molecular data for: GDSC
GDSC_cells <- intersect(rownames(GDSC_expr), colnames(GDSC.aac))
plot(GDSC.aac["17-AAG", GDSC_cells], GDSC_expr[GDSC_cells, features],
     main="GDSC + 17-AAG + NQ01",
     cex.main=1, ylab="Predictions", xlab="drug sensitivity", pch=20, col="gray40")

CCLE_mut <- t(exprs(summarizeMolecularProfiles(CCLE, mDataType="mutation", fill.missing=FALSE, summary=TRUE))
CCLE_cells <- intersect(rownames(CCLE_mut), colnames(CCLE.aac))
boxplot(CCLE.aac["PD-0325901", CCLE_cells] ~ CCLE_mut[CCLE_cells, "BRAF"], col="gray80", pch=20, main="CCLE + PD-0325901 + BRAF",
        cex.main=1, xlab="mutation", ylab="drug sensitivity")

GDSC_mut <- t(exprs(summarizeMolecularProfiles(GDSC, mDataType="mutation", fill.missing=FALSE, summary=TRUE))

```

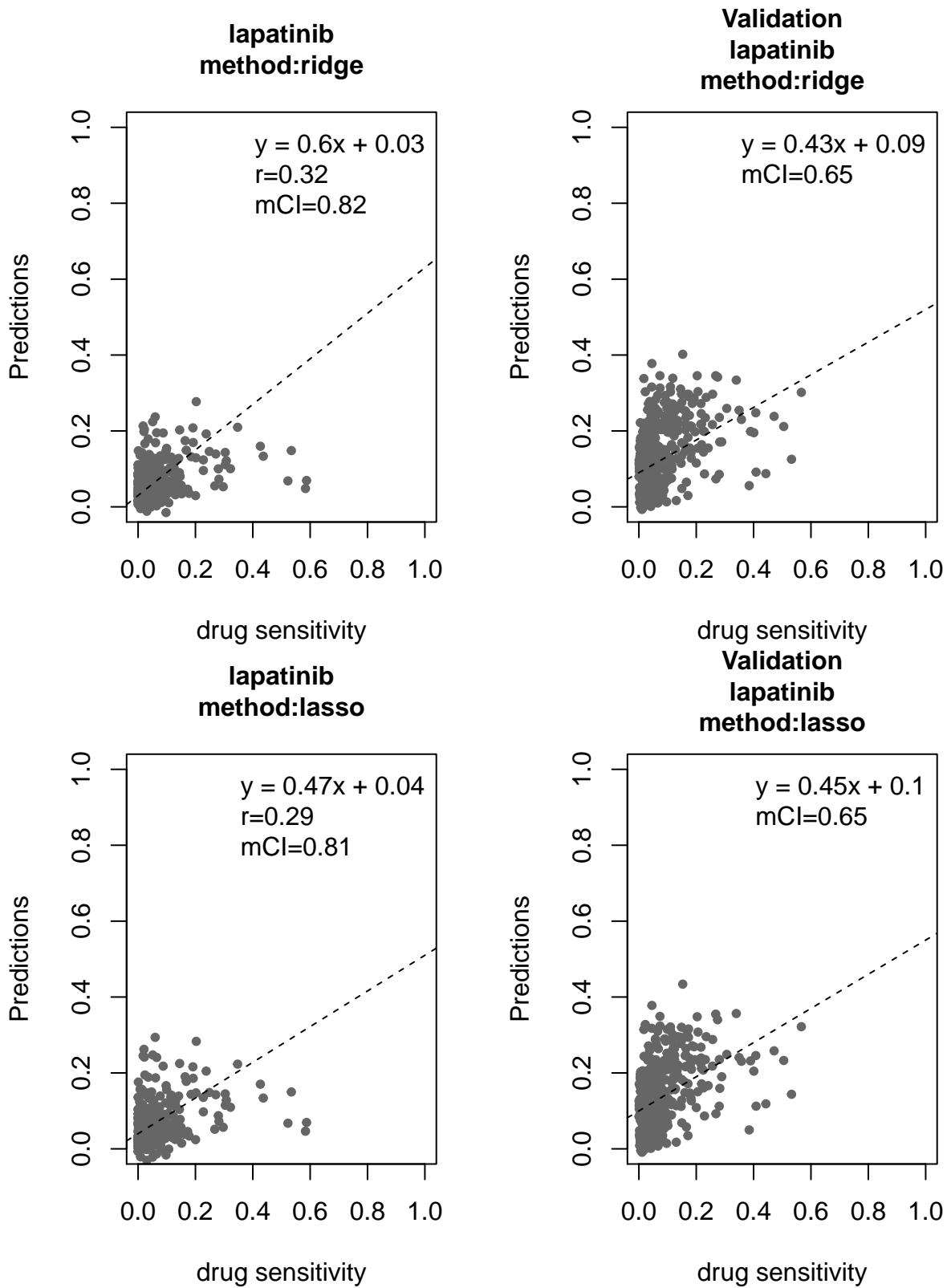
```
GDSC_cells <- intersect(rownames(GDSC_mut), colnames(GDSC.aac))
boxplot(GDSC.aac["PD-0325901", GDSC_cells] ~ GDSC_mut[GDSC_cells, "BRAF"], col="gray80", pch=20, main="GDSC + PD-0325901 + BRAF"
       cex.main=1, xlab="mutation", ylab="drug sensitivity")
```

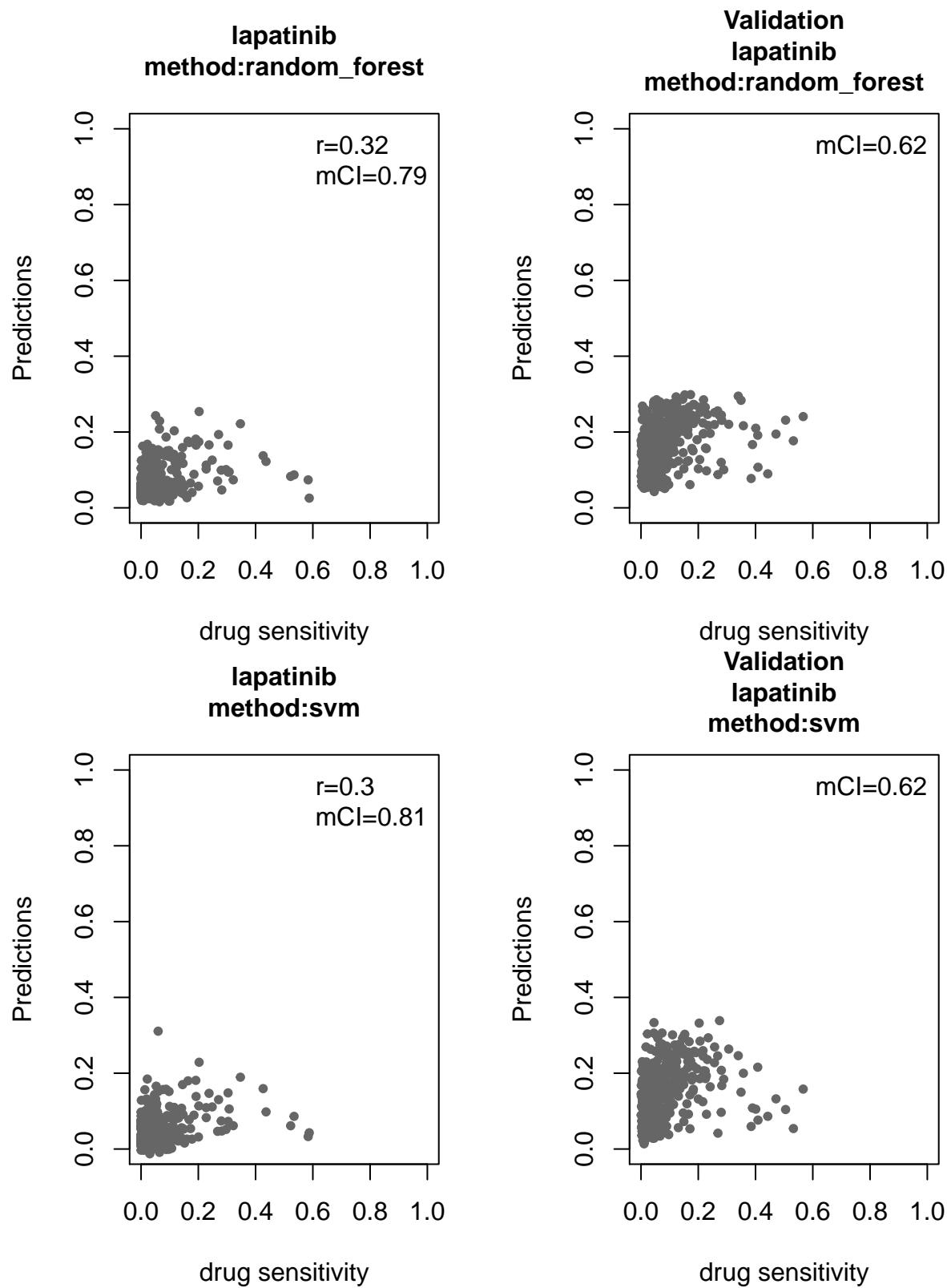


14.8 Machine Learning and Biomarker Discovery

Some of the widely used multivariate machine learning methods such as elastic net, Random Forest (RF) and Support Vector Machine (SVM) have been already implemented in the MLWorkshop. It optimizes hyperparameters of these methods in the training phase. To assess the performance of the predictive models, it implements m number of sampling with n -fold cross validations (CV). The performance will then be assessed by multiple metrics including pearson correlation coefficient, concordance index and modified concordance index.

```
suppressPackageStartupMessages({
  library(mRMRe, verbose=FALSE)
  library(Biobase, verbose=FALSE)
  library(Hmisc, verbose=FALSE)
  library(glmnet, verbose=FALSE)
  library(caret, verbose=FALSE)
  library(randomForest, verbose=FALSE)
})
##Preparing trainig dataset
train_expr <- t(exprs(summarizeMolecularProfiles(GDSC, mDataType="rna", fill.missing=FALSE, verbose=FALSE,
  aac <- summarizeSensitivityProfiles(GDSC, sensitivity.measure="auc_recomputed", drug="lapatinib", fill.missing=FALSE,
  cells <- intersect(rownames(train_expr), names(aac)))
  df <- as.matrix(cbind(train_expr[cells,], "lapatinib"=aac[cells]))
```



14.8.1 Bonus: Using the Connectivity Map for drug repurposing

We show here how to use *PharmacoGx* for linking drug perturbation signatures inferred from CMAP to independent signatures of HDAC inhibitors published in Glaser et al. (2003). We therefore sought to reproduce the HDAC analysis in Lamb et al. (2006) using the latest version of CMAP that can be downloaded using downloadPSet. The connectivityScore function enables the computation of the connectivity scores between the 14-gene HDAC signature from (Glaser et al., 2003) and over 1000 CMAP drugs. This analysis results in the four HDAC inhibitors in CMAP being ranked at the top of the drug list (Fig. 2), therefore concurring with the original CMAP analysis (Lamb et al., 2006).

```
## download and process the HDAC signature
mydir <- "1132939s"
downloader::download(paste("http://www.sciencemag.org/content/suppl/2006/09/29/313.5795.1929.DC1/", mydir))
unzip(paste(mydir, ".zip", sep=""))

library(hgu133a.db)
library(PharmacoGx)

HDAC_up <- gdata::read.xls(paste(mydir, paste(mydir, "sigS1.xls", sep="_"), sep="/"), sheet=1, header=FALSE)
HDAC_down <- gdata::read.xls(paste(mydir, paste(mydir, "sigS1.xls", sep="_"), sep="/"), sheet=2, header=FALSE)
HDAC <- as.data.frame(matrix(NA, nrow=nrow(HDAC_down)+nrow(HDAC_up), ncol=2))
annot <- AnnotationDbi::select(hgu133a.db, keys = c(HDAC_up[[1]], HDAC_down[[1]]), columns=c("ENSEMBL"))
gene_up <- unique(annot[match(HDAC_up[[1]], annot[,1]),2])
gene_down <- na.omit(unique(annot[match(HDAC_down[[1]], annot[,1]),2]))
HDAC_genes <- as.data.frame(matrix(NA, nrow=length(gene_down)+length(gene_up), ncol=2))

HDAC_genes[ , 2] <- c(rep(1, times=length(gene_up)), rep(-1, times=length(gene_down)))
HDAC_genes[ , 1] <- c(gene_up, gene_down)
rownames(HDAC_genes) <- HDAC_genes[ , 1]
HDAC <- HDAC_genes[ , 2]
names(HDAC) <- rownames(HDAC_genes)

drug.perturbation <- PharmacoGx::downloadPertSig("CMAP")
dimnames(drug.perturbation)[[1]] <- gsub("_at", "", dimnames(drug.perturbation)[[1]])

message("Be aware that computing sensitivity will take some time...")
cl <- parallel::makeCluster(2)
res <- parApply(drug.perturbation[ , , c("tstat", "fdr")], 2, function(x, HDAC){
  return(PharmacoGx::connectivityScore(x=x, y=HDAC, method="gsea", nperm=100))
}, cl=cl, HDAC=HDAC)
stopCluster(cl)
rownames(res) <- c("Connectivity", "P Value")
res <- t(res)

res <- apply(drug.perturbation[ , , c("tstat", "fdr")], 2, function(x, HDAC){
  return(PharmacoGx::connectivityScore(x=x, y=HDAC, method="gsea", nperm=100))
}, HDAC=HDAC)
rownames(res) <- c("Connectivity", "P Value")
res <- t(res)

HDAC_inhibitors <- c("vorinostat", "trichostatin A", "HC toxin", "valproic acid")
```

```
res <- res[order(res[, 1], decreasing=T), ]
HDAC_ranks <- which(rownames(res) %in% HDAC_inhibitors)
```

14.9 Session Info

This document was generated with the following R version and packages loaded:

```
sessionInfo()
#> R version 3.5.0 (2018-04-23)
#> Platform: x86_64-pc-linux-gnu (64-bit)
#> Running under: Debian GNU/Linux 9 (stretch)
#>
#> Matrix products: default
#> BLAS: /usr/lib/openblas-base/libblas.so.3
#> LAPACK: /usr/lib/libopenblas-r0.2.19.so
#>
#> locale:
#> [1] LC_CTYPE=en_US.UTF-8          LC_NUMERIC=C
#> [3] LC_TIME=en_US.UTF-8          LC_COLLATE=en_US.UTF-8
#> [5] LC_MONETARY=en_US.UTF-8       LC_MESSAGES=C
#> [7] LC_PAPER=en_US.UTF-8          LC_NAME=C
#> [9] LC_ADDRESS=C                  LC_TELEPHONE=C
#> [11] LC_MEASUREMENT=en_US.UTF-8    LC_IDENTIFICATION=C
#>
#> attached base packages:
#> [1] stats4      parallel    stats      graphics   grDevices  utils      datasets
#> [8] methods     base
#>
#> other attached packages:
#> [1] hgu133a.db_3.2.3   org.Hs.eg.db_3.6.0  AnnotationDbi_1.43.1
#> [4] IRanges_2.15.16    S4Vectors_0.19.19  randomForest_4.6-14
#> [7] caret_6.0-80      glmnet_2.0-16     foreach_1.4.4
#> [10] Matrix_1.2-14    Hmisc_4.1-1      Formula_1.2-3
#> [13] lattice_0.20-35   mRMRe_2.0.7     igraph_1.2.1
#> [16] survival_2.42-6   xtable_1.8-2     reshape2_1.4.3
#> [19] ggplot2_3.0.0     Biobase_2.41.2   BiocGenerics_0.27.1
#> [22] PharmacoGxML_0.1.0 mCI_0.1.0      Pharmacogenomics_1.11.0
#>
#> loaded via a namespace (and not attached):
#> [1] NISTunits_1.0.1    backports_1.1.2    fastmatch_1.1-0
#> [4] sm_2.2-5.5        plyr_1.8.4        lazyeval_0.2.1
#> [7] splines_3.5.0     BiocParallel_1.15.8 SnowballC_0.5.1
#> [10] digest_0.6.15    htmltools_0.3.6    gdata_2.18.0
#> [13] magrittr_1.5      checkmate_1.8.5    memoise_1.1.0
#> [16] cluster_2.0.7-1  sfsmisc_1.1-2     limma_3.37.3
#> [19] recipes_0.1.3     gower_0.1.2       celestial_1.4.1
#> [22] dimRed_0.1.0     piano_1.21.0     colorspace_1.3-2
#> [25] blob_1.1.1       xfun_0.3         dplyr_0.7.6
#> [28] tcltk_3.5.0      crayon_1.3.4     bindr_0.1.1
#> [31] iterators_1.0.10   glue_1.3.0       DRR_0.0.3
#> [34] gtable_0.2.0      ipred_0.9-6      kernlab_0.9-26
#> [37] ddalpha_1.3.4     DEoptimR_1.0-8    maps_3.3.0
```

```

#> [40] abind_1.4-5           scales_0.5.0          DBI_1.0.0
#> [43] relations_0.6-8      Rcpp_0.12.18         plotrix_3.7-2
#> [46] htmlTable_1.12       magic_1.5-8          foreign_0.8-71
#> [49] bit_1.1-14          mapproj_1.2.6        lava_1.6.2
#> [52] prodlim_2018.04.18   htmlwidgets_1.2     fgsea_1.7.1
#> [55] gplots_3.0.1         RColorBrewer_1.1-2   acepack_1.4.1
#> [58] pkgconfig_2.0.1       nnet_7.3-12          tidyselect_0.2.4
#> [61] labeling_0.3         rlang_0.2.1          munsell_0.5.0
#> [64] tools_3.5.0          downloader_0.4     RSQLite_2.1.1
#> [67] pls_2.6-0            broom_0.5.0          evaluate_0.11
#> [70] geometry_0.3-6       stringr_1.3.1       yaml_2.1.19
#> [73] ModelMetrics_1.1.0    knitr_1.20          bit64_0.9-7
#> [76] robustbase_0.93-1.1   caTools_1.17.1.1    purrrr_0.2.5
#> [79] RANN_2.6              bindrcpp_0.2.2       nlme_3.1-137
#> [82] slam_0.1-43          RcppRoll_0.3.0      pracma_2.1.4
#> [85] BiocStyle_2.9.3       compiler_3.5.0     rstudioapi_0.7
#> [88] marray_1.59.0         tibble_1.4.2         stringi_1.2.4
#> [91] highr_0.7             pillar_1.3.0        magicaxis_2.0.3
#> [94] data.table_1.11.4     bitops_1.0-6         R6_2.2.2
#> [97] latticeExtra_0.6-28   bookdown_0.7        KernSmooth_2.23-15
#> [100] gridExtra_2.3        lsa_0.73.1          codetools_0.2-15
#> [103] MASS_7.3-50          gtools_3.8.1        assertthat_0.2.0
#> [106] CVST_0.2-2          rprojroot_1.3-2    withr_2.1.2
#> [109] grid_3.5.0           rpart_4.1-13       timeDate_3043.102
#> [112] tidyrr_0.8.1          class_7.3-14       rmarkdown_1.10
#> [115] sets_1.0-18          lubridate_1.7.4     base64enc_0.1-3

```

Chapter 15

500: Effectively using the DelayedArray framework to support the analysis of large datasets

Authors: Peter Francis Hickey¹, Last modified: 19 June, 2018.

15.1 Overview

15.1.1 Description

This workshop will teach the fundamental concepts underlying the DelayedArray framework and related infrastructure. It is intended for package developers who want to learn how to use the DelayedArray framework to support the analysis of large datasets, particularly through the use of on-disk data storage.

The first part of the workshop will provide an overview of the DelayedArray infrastructure and introduce computing on DelayedArray objects using delayed operations and block-processing. The second part of the workshop will present strategies for adding support for DelayedArray to an existing package and extending the DelayedArray framework.

Students can expect a mixture of lecture and question-and-answer session to teach the fundamental concepts. There will be plenty of examples to illustrate common design patterns for writing performant code, although we will not be writing much code during the workshop.

15.1.2 Pre-requisites

- Solid understanding of R
- Familiarity with common operations on arrays (e.g., `colSums()` and those available in the **matrixStats** package)
- Familiarity with object oriented programming, particularly S4, will be useful but is not essential
- No familiarity required of technical details of particular data storage backends (e.g., HDF5, sparse matrices)
- No familiarity required of particular biological techniques (e.g., single-cell RNA-seq)

¹Department of Biostatistics, Johns Hopkins University

15.1.3 Participation

Questions and discussion are encouraged! This will be especially important to guide the second half of the workshop which focuses on integrating DelayedArray into an existing or new Bioconductor package. Students will be expected to be able to follow and reason about example R code.

15.1.4 R / Bioconductor packages used

- DelayedArray
- HDF5Array
- SummarizedExperiment
- DelayedMatrixStats
- beachmat

15.1.5 Time outline

Activity	Time
Introductory slides	15m
Part 1: Overview of DelayedArray framework	45m
Part 2: Incorporating DelayedArray into a package	45m
Questions and discussion	15m

15.1.6 Workshop goals and objectives

15.1.6.1 Learning goals

- Identify when it is useful to use a *DelayedArray* instead of an ordinary array or other array-like data structure.
- Become familiar with the fundamental concepts of delayed operations, block-processing, and realization.
- Learn of existing functions and packages for constructing and computing on *DelayedArray* objects, avoiding the need to re-invent the wheel.
- Learn common design patterns for writing performant code that operates on a *DelayedArray*.
- Evaluate whether an existing function that operates on an ordinary array can be readily adapted to work on a *DelayedArray*.
- Reason about potential bottlenecks in algorithms operating on *DelayedArray* objects.

15.1.6.2 Learning objectives

- Understand the differences between a *DelayedArray* instance and an instance of a subclass (e.g., *HDF5Array*, *RleArray*).
- Know what types of operations ‘degrade’ an instance of a *DelayedArray* subclass to a *DelayedArray*, as well as when and why this matters.
- Construct a *DelayedArray*:
 - From an in-memory array-like object.
 - From an on-disk data store (e.g., *HDF5*).
 - From scratch by writing data to a *RealizationSink*.
- Take a function that operates on rows or columns of a matrix and apply it to a *DelayedMatrix*.
- Use block-processing on a *DelayedArray* to compute:
 - A univariate (scalar) summary statistic (e.g., `max()`).

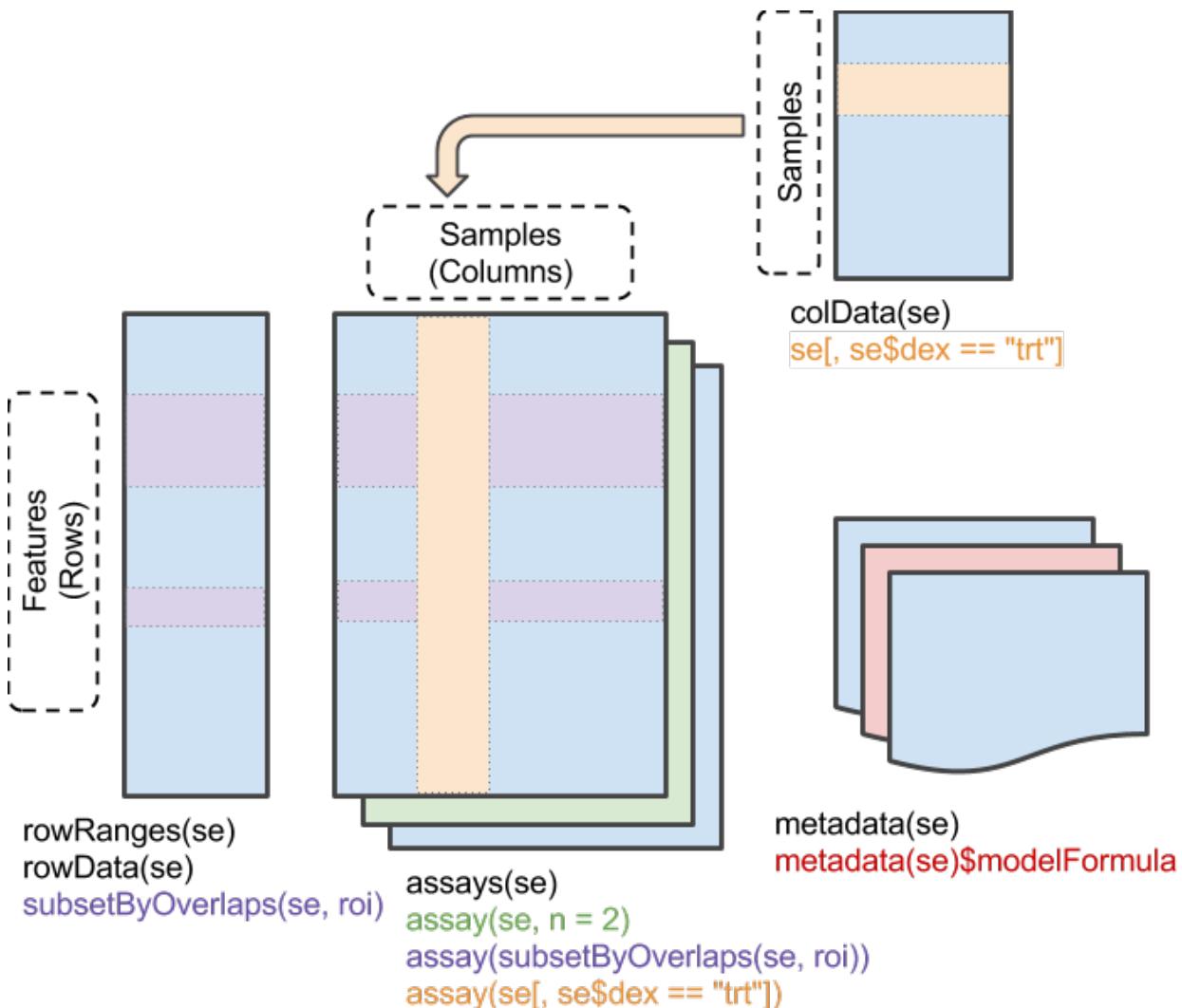


Figure 15.1: The `SummarizedExperiment` class is used to store rectangular arrays of experimental results (`assays`). Although each `assay` is here drawn as a matrix, higher-dimensional arrays are also supported.

- A multivariate (vector) summary statistic (e.g., `colSums()` or `rowMeans()`).
- A multivariate (array-like) summary statistic (e.g., `rowRanks()`).
- Design an algorithm that imports data into a `DelayedArray`.

15.2 Introductory material

Data from a high-throughput biological assay, such as single-cell RNA-seqencing (scRNA-seq), will often be summarised as a matrix of counts, where rows correspond to features and columns to samples². Within **Bioconductor**, the `SummarizedExperiment` class is the recommended container for such data, offering a rich interface that tightly links assay measurements to data on the features and the samples.

Traditionally, the assay data are stored in-memory as an ordinary `array` object³. Storing the data in-memory becomes a real pain with the ever-growing size of 'omics datasets; it is now not uncommon to collect

²Higher-dimensional arrays may be appropriate for some types of assays.

³In R, a `matrix` is just a 2-dimensional `array`.

10,000 – 100,000,000 measurements on 100 – 1,000,000 samples, which would occupy 10 – 1,000 gigabytes (GB) if stored in-memory as ordinary R arrays!

Let's take as an example some single-cell RNA-seq data on 1.3 million brain cells from embryonic mice, generated by 10X Genomics⁴.

```
library(TENxBrainData)
# NOTE: This will download the data and may take a little while on the first
#       run. The result will be cached, however, so subsequent runs are near
#       instantaneous.
tenx <- TENxBrainData()
# The data are stored in a SingleCellExperiment, an extension of the
# SummarizedExperiment class.
class(tenx)
#> [1] "SingleCellExperiment"
#> attr(package)
#> [1] "SingleCellExperiment"
dim(tenx)
#> [1] 27998 1306127

# How big much memory do the counts data use?
counts <- assay(tenx, "counts", withDimnames = FALSE)
print(object.size(counts))
#> 2144 bytes
```

The counts data only require a tiny amount of RAM despite it having 27998 rows and 1306127 columns. And the counts object still “feels” like an ordinary R matrix:

```
# Oooh, pretty-printing.
counts
#> <27998 x 1306127> HDF5Matrix object of type "integer":
#>      [,1]     [,2]     [,3]     [,4] ... [,1306124]
#> [1,] 0 0 0 0 .
#> [2,] 0 0 0 0 .
#> [3,] 0 0 0 0 .
#> [4,] 0 0 0 0 .
#> [5,] 0 0 0 0 .
#> ...
#> [27994,] 0 0 0 0 .
#> [27995,] 1 0 0 2 .
#> [27996,] 0 0 0 0 .
#> [27997,] 0 0 0 0 .
#> [27998,] 0 0 0 0 .
#>      [,1306125] [,1306126] [,1306127]
#> [1,] 0 0 0
#> [2,] 0 0 0
#> [3,] 0 0 0
#> [4,] 0 0 0
#> [5,] 0 0 0
#> ...
#> [27994,] 0 0 0
#> [27995,] 1 0 0
#> [27996,] 0 0 0
#> [27997,] 0 0 0
```

⁴These data are available in the **TENxBrainData** Bioconductor package

```
#> [27998,]          0          0          0

# Let's take a subset of the data
counts[1:10, 1:10]
#> <10 x 10> DelayedMatrix object of type "integer":
#>   [,1]  [,2]  [,3]  [,4] ...  [,7]  [,8]  [,9]  [,10]
#>  [1,]    0    0    0    0   .    0    0    0    0
#>  [2,]    0    0    0    0   .    0    0    0    0
#>  [3,]    0    0    0    0   .    0    0    0    0
#>  [4,]    0    0    0    0   .    0    0    0    0
#>  [5,]    0    0    0    0   .    0    0    0    0
#>  [6,]    0    0    0    0   .    0    0    0    0
#>  [7,]    0    0    0    0   .    0    0    0    0
#>  [8,]    0    0    0    0   .    2    0    1    1
#>  [9,]    0    0    1    0   .    0    0    0    0
#> [10,]    0    0    0    0   .    0    0    0    0

# Let's compute column sums (crude library sizes) for the first 100 samples.
# TODO: DelayedArray:: prefix shouldn't be necessary
DelayedArray::colSums(counts[, 1:100])
#>  [1] 4046 2087 4654 3193 8444 11178 2375 3672 3115 4592 7899
#> [12] 5474 2894 5887 8349 7310 3340 23410 7326 3446 4601 5921
#> [23] 4746 3063 3879 3582 2854 4053 6987 9155 3747 1534 6632
#> [34] 4099 2846 5025 6688 3742 2982 1998 1808 3121 10561 3874
#> [45] 4143 1500 2280 3060 4325 3161 2522 1979 6033 3721 2546
#> [56] 6317 2756 3896 4475 5580 1879 8746 4873 2202 4517 2815
#> [67] 3809 2580 4655 3523 4717 6436 2434 5704 2962 11654 4848
#> [78] 5288 6689 5761 11539 15745 2986 2736 3666 2476 2251 3052
#> [89] 5480 1721 4166 4451 1893 5606 2551 2810 1555 1840 2972
#> [100] 2404
```

The reason for the small memory footprint and matrix-like “feel” of the `counts` object is because the counts data are in fact stored on-disk in a Hierarchical Data Format (**HDF5**) file and we are interacting with the data via the `DelayedArray` framework.

15.2.1 Discussion

TODO: Make these ‘discussions/challenges’ into a boxes that visually break out

- Talk with your neighbour about the sorts of ‘big data’ you are analysing, the challenges you’ve faced, and strategies you’re using to tackle this.
- **TODO:** (keep this?) Play around with the `tenx` and `counts` data. Could use this to demonstrate the upcoming challenges of chunking (e.g., compute row-wise summary).

TABLE gives some examples of contemporary experiments and the memory requirements if these data are to be stored in-memory as ordinary R arrays.

Assay	Size	‘nrow’	‘ncol’	Number of assays	Type	Reference
WGBS (mCG)	67.992 GB	29,307,073	145	3	2 x dense integer, 1 x dense double	eGTEX
scRNA-seq	146.276 GB	27,998	1,306,127	1	sparse integer	https:///

There are various strategies for handling such large data in R. For example, we could use:

1. Sparse matrices for single-cell RNA-seq data.
2. Run length encoded vectors for ChIP-seq coverage profiles.

3. Storing data on disk, and only bringing into memory as required, for whole genome methylation data⁵.

Each of these approaches has its strengths, as well as weaknesses and idiosyncrasies. For example,

- **Matrix** doesn't support long vectors
- Sparsity is readily lost (e.g., normalization often destroys it)
- etc.

TODO: Lead discussion of limitations

15.2.2 Why learn DelayedArray?

The high-level goals of the DelayedArray framework are:

- Provide a common R interface to array-like data, where the data may be in-memory or on-disk.
- Support delayed operations, which avoid doing any computation until the result is required.
- Support block-processing of the data, which enables bounded-memory and parallel computations.

From the **DelayedArray** DESCRIPTION:

TODO: Use **desc** to extract Description field?

Wrapping an array-like object (typically an on-disk object) in a DelayedArray object allows one to perform common array operations on it without loading the object in memory. In order to reduce memory usage and optimize performance, operations on the object are either delayed or executed using a block processing mechanism. Note that this also works on in-memory array-like objects like DataFrame objects (typically with Rle columns), Matrix objects, and ordinary arrays and data frames. (<https://bioconductor.org/packages/release/bioc/html/DelayedArray.html>)

These goals are similar to that of the **tibble** and **dplyr** packages:

A **tibble**, or **tbl_df**, is a modern reimagining of the **data.frame**, keeping what time has proven to be effective, and throwing out what is not. Tibbles are **data.frames** that are lazy and surly (<http://tibble.tidyverse.org/#overview>)

dplyr is designed to abstract over how the data is stored. That means as well as working with local data frames, you can also work with remote database tables, using exactly the same R code. (<https://dplyr.tidyverse.org/#overview>)

An important feature of the DelayedArray framework is that it supports all these strategies, and more, with a common interface that aims to feel like the interface to ordinary arrays, which provides familiarity to R users.

15.2.2.1 Learning goal

- Identify when it is useful to use a *DelayedArray* instead of an ordinary array or other array-like data structure.

15.3 Overview of DelayedArray framework

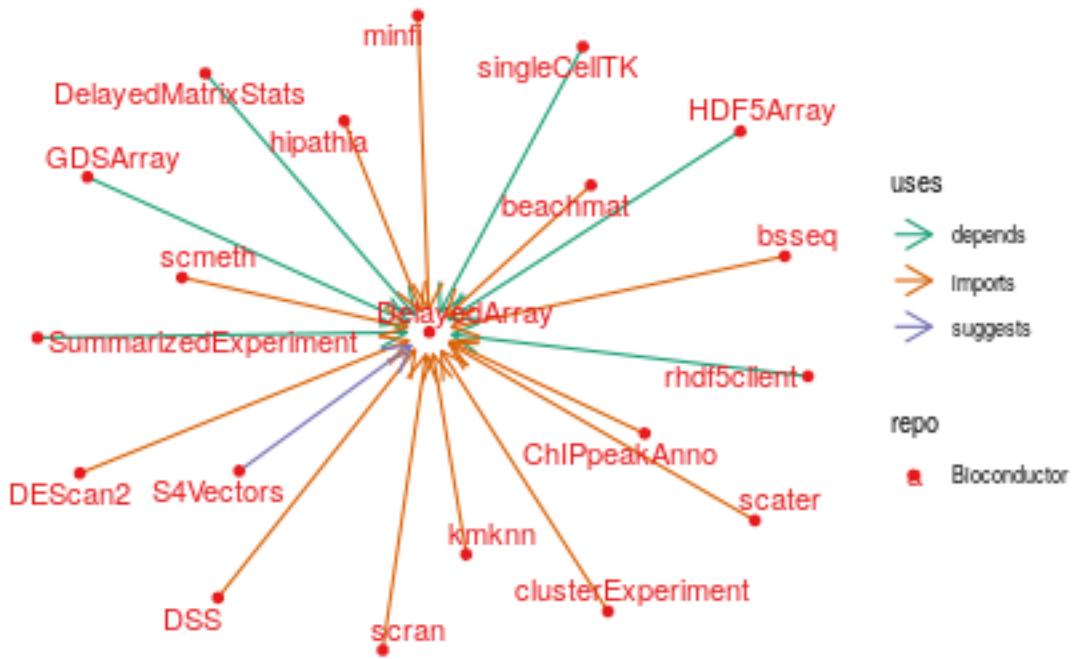
The core of DelayedArray framework is implemented in the **DelayedArray** Bioconductor package. Other packages extend the framework or simply use it as is to enable analyses of large datasets.

⁵This strategy can also be effectively applied to sparse and repetitive data by using on-disk compression of the data.

15.3.1 The DelayedArray package

The **DelayedArray** package defines the key classes, generics, and methods⁶, as well as miscellaneous helper functions, that implement the DelayedArray framework.

The reverse dependencies of **DelayedArray** are shown below:



The above figures includes packages that extend the DelayedArray framework in various ways, as well as those that simply use the DelayedArray framework to analyse specific types of 'omics data. We briefly discuss some of these:

15.3.1.1 Packages that extend DelayedArray

There are two ways a package may extend the DelayedArray framework. The first kind of package adds support for a new *realization backend* (**TODO Define?**). Examples of this are:

- The **HDF5Array** package adds the *HDF5Array* realization backend for accessing and creating data stored on-disk in an HDF5 file. This is typically used for on-disk representation of multidimensional (numeric) arrays.
- The **GDSArray** package adds the *GDSArray* backend for accessing and creating data stored on-disk in a GDS file. This is typically used for on-disk representation of genotyping or sequence data.
- The **rhdf5client** packages adds the *H5S_Array* realization backend for accessing and creating data stored on a HDF Server, a Python-based web service that can be used to send and receive HDF5 data using an HTTP-based REST interface. This is typically used for on-disk representation of multidimensional (numeric) arrays that need to be shared with multiple users from a central location.

⁶The **DelayedArray** package, like all core Bioconductor packages, uses the S4 object oriented programming system.

The second kind of package adds methods for computing on *DelayedArray* instances. Examples of this are:

- The **DelayedMatrixStats** package provides methods for computing commonly used row- and column-wise summaries of a 2-dimensional *DelayedArray*.
- The **beachmat** package provides a consistent C++ class interface for a variety of commonly used matrix types, including ordinary R arrays, sparse matrices, and *DelayedArray* with various backends.
- The **kmknn** package provides methods for performing k-means for k-nearest neighbours of data stored in a *DelayedArray*.
- Packages for performing matrix factorizations, (generalized) linear regression, , etc. (work in progress)

15.3.1.2 Packages that use *DelayedArray*

- The **bsseq** package uses the *DelayedArray* framework to support the analysis of large whole-genome bisulfite methylation sequencing experiments.
- **TODO:** minfi, scater, scran, others

15.3.2 The *DelayedArray* class

The *DelayedArray* class is the key data structure that end users of the *DelayedArray* framework will interact with. A *DelayedMatrix* is the same thing as a two-dimensional *DelayedArray*, just as a *matrix* is the same thing as a two-dimensional *array*.

The *DelayedArray* class has a single slot called the *seed*. This name is evocative, it is the core of the object.

A package developer may create a subclass of *DelayedArray*; we will make extensive use of the *HDF5Array* class, for example.

```
# TODO: Graphical representation of this network?
# TODO: Illustrate How classes depend on one another
#       a. with DelayedArray as the root
#       b. With DelayedArray as the leaf
showClass(getClass("DelayedArray", where = "DelayedArray"))
#> Class "DelayedArray" [package "DelayedArray"]
#>
#> Slots:
#>
#> Name: seed
#> Class: ANY
#>
#> Extends:
#> Class "DelayedUnaryOp", directly
#> Class "DelayedOp", by class "DelayedUnaryOp", distance 2
#> Class "Array", by class "DelayedUnaryOp", distance 3
#>
#> Known Subclasses:
#> Class "DelayedMatrix", directly, with explicit coerce
#> Class "DelayedArray1", directly
#> Class "RleArray", directly
#> Class "RleMatrix", by class "DelayedMatrix", distance 2
showClass(getClass("HDF5Array", where = "HDF5Array"))
#> Class "HDF5Array" [package "HDF5Array"]
#>
#> Slots:
#>
```

```
#> Name: seed
#> Class: ANY
#>
#> Extends:
#> Class "DelayedArray", directly
#> Class "DelayedUnaryOp", by class "DelayedArray", distance 2
#> Class "DelayedOp", by class "DelayedArray", distance 3
#> Class "Array", by class "DelayedArray", distance 4
#>
#> Known Subclasses:
#> Class "HDF5Matrix", directly, with explicit coerce
```

15.3.2.1 Learning objectives

In this section, we'll learn how an end user can construct a *DelayedArray* from:

- An in-memory array-like object.
- A local, on-disk data store, such as an HDF5 file.
- A remote data store, such as from a HDF Server.

We'll also learn:

- What defines the “seed contract”
- What types of operations ‘degrade’ an instance of a *DelayedArray* subclass to a *DelayedArray*, as well as when and why this matters (**TODO** Save the when and why it matters to later?).

15.3.2.2 The seed of a *DelayedArray*

From an end user's perspective, there are two broad categories of seed:

1. In-memory seeds
2. Out-of-memory seeds
 - a. Local, on-disk seeds
 - b. Remote (in-memory or on-disk) seeds

TODO: Box this out as an aside

For a developer's perspective, there's a third category of seed that is defined by the *DelayedOp* class. Instances of the *DelayedOp* class are not intended to be manipulated directly by the end user, but they are central to the concept of delayed operations, which we will learn about later (**TODO:** Link to section).

A seed must implement the “seed contract”⁷:

- `dim(x)`: Return the dimensions of the seed.
- `dimnames(x)`: Return the (possibly NULL) dimension names of the seed.
- `extract_array(x, index)`: Return a slice, specified by `index`, of the seed as an ordinary R *array*.

15.3.2.3 In-memory seeds

To begin, we'll consider a *DelayedArray* instance with the simplest *seed*, an in-memory array:

⁷For further details, see the vignette in the **DelayedArray** package (available via `vignette("02-Implementing_a_backend", "DelayedArray")`)

```
library(DelayedArray)

mat <- matrix(rep(1:20, 1:20), ncol = 2)
da_mat <- DelayedArray(seed = mat)
da_mat
#> <105 x 2> DelayedMatrix object of type "integer":
#>      [,1] [,2]
#> [1,]    1   15
#> [2,]    2   15
#> [3,]    2   15
#> [4,]    3   15
#> [5,]    3   15
#> ...
#> [101,]   14  20
#> [102,]   14  20
#> [103,]   14  20
#> [104,]   14  20
#> [105,]   14  20
```

We can use other, more complex, array-like objects as the *seed*, such as *Matrix* objects from the **Matrix** package:

```
library(Matrix)
#>
#> Attaching package: 'Matrix'
#> The following object is masked from 'package:S4Vectors':
#>
#>      expand
Mat <- Matrix(mat)
da_Mat <- DelayedArray(seed = Mat)
# NOTE: The type is "double" because of how the Matrix package stores the data.
da_Mat
#> <105 x 2> DelayedMatrix object of type "double":
#>      [,1] [,2]
#> [1,]    1   15
#> [2,]    2   15
#> [3,]    2   15
#> [4,]    3   15
#> [5,]    3   15
#> ...
#> [101,]   14  20
#> [102,]   14  20
#> [103,]   14  20
#> [104,]   14  20
#> [105,]   14  20
```

We can even use data frames as the *seed* of a two-dimensional *DelayedArray*.

```
df <- as.data.frame(mat)
da_df <- DelayedArray(seed = df)
# NOTE: This inherits the (default) column names of the data.frame.
da_df
#> <105 x 2> DelayedMatrix object of type "integer":
#>      V1 V2
#> [1,]  1 15
```

```

#> 2 2 15
#> 3 2 15
#> 4 3 15
#> 5 3 15
#> ... . .
#> 101 14 20
#> 102 14 20
#> 103 14 20
#> 104 14 20
#> 105 14 20

library(tibble)
tbl <- as_tibble(mat)
da_tbl <- DelayedArray(seed = tbl)
# NOTE: This inherits the (default) column names of the tibble.
da_tbl
#> <105 x 2> DelayedMatrix object of type "integer":
#>     V1 V2
#> 1 1 15
#> 2 2 15
#> 3 2 15
#> 4 3 15
#> 5 3 15
#> ... . .
#> 101 14 20
#> 102 14 20
#> 103 14 20
#> 104 14 20
#> 105 14 20

DF <- as(mat, "DataFrame")
da_DF <- DelayedArray(seed = DF)
# NOTE: This inherits the (default) column names of the DataFrame.
da_DF
#> <105 x 2> DelayedMatrix object of type "integer":
#>     V1 V2
#> [1,] 1 15
#> [2,] 2 15
#> [3,] 2 15
#> [4,] 3 15
#> [5,] 3 15
#> ...
#> [101,] 14 20
#> [102,] 14 20
#> [103,] 14 20
#> [104,] 14 20
#> [105,] 14 20

```

A package developer can also implement a novel in-memory seed. For example, the **DelayedArray** package defines the *RleArraySeed* class⁸. This can be used as the seed of an *RleArray*, a *DelayedArray* subclass for storing run-length encoded data.

⁸In fact, the *RleArraySeed* class is a virtual class, with concrete subclasses *SolidRleArraySeed* and *ChunkedRleArraySeed*

```
# NOTE: The DelayedArray package does not expose the RleArraySeed() constructor.
# Instead, we directly call the RleArray() constructor on the run-length
# encoded data.
da_Rle <- RleArray(rle = Rle(mat), dim = dim(mat))
da_Rle
#> <105 x 2> RleMatrix object of type "integer":
#>      [,1] [,2]
#> [1,]    1   15
#> [2,]    2   15
#> [3,]    2   15
#> [4,]    3   15
#> [5,]    3   15
#> ...
#> [101,]  14  20
#> [102,]  14  20
#> [103,]  14  20
#> [104,]  14  20
#> [105,]  14  20
```

The *RleArray* examples illustrates some important concepts in the *DelayedArray* class hierarchy that warrants reiteration and expansion.

15.3.2.4 Degrading *DelayedArray* subclasses

The `da_Rle` object is an *RleMatrix*, a direct subclass of *RleArray* and a direct subclass of *DelayedMatrix*. Both *RleArray* and *DelayedMatrix* are direct subclasses of a *DelayedArray*. As such, in accordance with properties of S4 class inheritance, **an *RleMatrix* is a *DelayedArray***.

```
da_Rle
#> <105 x 2> RleMatrix object of type "integer":
#>      [,1] [,2]
#> [1,]    1   15
#> [2,]    2   15
#> [3,]    2   15
#> [4,]    3   15
#> [5,]    3   15
#> ...
#> [101,]  14  20
#> [102,]  14  20
#> [103,]  14  20
#> [104,]  14  20
#> [105,]  14  20
is(da_Rle, "DelayedArray")
#> [1] TRUE
showClass(getClass("RleMatrix", where = "DelayedArray"))
#> Class "RleMatrix" [package "DelayedArray"]
#>
#> Slots:
#>
#> Name: seed
#> Class: ANY
#>
#> Extends:
```

```
#> Class "DelayedMatrix", directly
#> Class "RleArray", directly
#> Class "DelayedArray", by class "DelayedMatrix", distance 2
#> Class "DelayedUnaryOp", by class "DelayedMatrix", distance 2
#> Class "DelayedOp", by class "DelayedMatrix", distance 2
#> Class "Array", by class "DelayedMatrix", distance 2
#> Class "DataTable", by class "DelayedMatrix", distance 2
#> Class "DataTable_OR_NULL", by class "DelayedMatrix", distance 3
```

However, if we do (almost) anything to the *RleMatrix*, the result is ‘degraded’ to a *DelayedMatrix*. This ‘degradation’ isn’t an issue in and of itself, but it can be surprising and can complicate writing functions that expect a certain type of input (e.g., S4 methods).

```
# NOTE: Adding one to each element will 'degrade' the result.
da_Rle + 1L
#> <105 x 2> DelayedMatrix object of type "integer":
#>      [,1] [,2]
#> [1,]    2   16
#> [2,]    3   16
#> [3,]    3   16
#> [4,]    4   16
#> [5,]    4   16
#> ...
#> [101,] 15  21
#> [102,] 15  21
#> [103,] 15  21
#> [104,] 15  21
#> [105,] 15  21
is(da_Rle + 1L, "RleMatrix")
#> [1] FALSE

# NOTE: Subsetting will 'degrade' the result.
da_Rle[1:10, ]
#> <10 x 2> DelayedMatrix object of type "integer":
#>      [,1] [,2]
#> [1,]    1   15
#> [2,]    2   15
#> [3,]    2   15
#> [4,]    3   15
#> [5,]    3   15
#> [6,]    3   15
#> [7,]    4   15
#> [8,]    4   15
#> [9,]    4   15
#> [10,]   4   15
is(da_Rle[1:10, ], "RleMatrix")
#> [1] FALSE

# NOTE: Changing the dimnames will 'degrade' the result.
da_Rle_with_dimnames <- da_Rle
colnames(da_Rle_with_dimnames) <- c("A", "B")
da_Rle_with_dimnames
#> <105 x 2> DelayedMatrix object of type "integer":
#>      A   B
```

```

#> [1,] 1 15
#> [2,] 2 15
#> [3,] 2 15
#> [4,] 3 15
#> [5,] 3 15
#> ...
#> [101,] 14 20
#> [102,] 14 20
#> [103,] 14 20
#> [104,] 14 20
#> [105,] 14 20
is(da_Rle_with_dimnames, "RleMatrix")
#> [1] FALSE

# NOTE: Transposing will 'degrade' the result.
t(da_Rle)
#> <2 x 105> DelayedMatrix object of type "integer":
#>      [,1]   [,2]   [,3]   [,4] ... [,102] [,103] [,104] [,105]
#> [1,]     1     2     2     3   .    14     14     14     14
#> [2,]     15    15    15    15   .    20     20     20     20
is(t(da_Rle), "RleMatrix")
#> [1] FALSE

# NOTE: Even adding zero (conceptually a no-op) will 'degrade' the result.
da_Rle + 0L
#> <105 x 2> DelayedMatrix object of type "integer":
#>      [,1]   [,2]
#> [1,]     1     15
#> [2,]     2     15
#> [3,]     2     15
#> [4,]     3     15
#> [5,]     3     15
#> ...
#> [101,]   14    20
#> [102,]   14    20
#> [103,]   14    20
#> [104,]   14    20
#> [105,]   14    20

```

There are some exceptions to this rule, when the DelayedArray framework can recognise/guarantee that the operation is a no-op that will leave the object in its current state:

```

# NOTE: Subsetting to select the entire object does not 'degrade' the result.
da_Rle[seq_len(nrow(da_Rle)), seq_len(ncol(da_Rle))]
#> <105 x 2> RleMatrix object of type "integer":
#>      [,1]   [,2]
#> [1,]     1     15
#> [2,]     2     15
#> [3,]     2     15
#> [4,]     3     15
#> [5,]     3     15
#> ...
#> [101,]   14    20
#> [102,]   14    20

```

```
#> [103,] 14 20
#> [104,] 14 20
#> [105,] 14 20
is(da_Rle[seq_len(nrow(da_Rle)), seq_len(ncol(da_Rle))], "RleMatrix")
#> [1] TRUE

# NOTE: Transposing and transposing back does not 'degrade' the result.
t(t(da_Rle))
#> <105 x 2> RleMatrix object of type "integer":
#>      [,1] [,2]
#> [1,]    1   15
#> [2,]    2   15
#> [3,]    2   15
#> [4,]    3   15
#> [5,]    3   15
#> ...
#> [101,]   .   .
#> [102,]   14  20
#> [103,]   14  20
#> [104,]   14  20
#> [105,]   14  20
is(t(t(da_Rle)), "RleMatrix")
#> [1] TRUE
```

A particularly important example of this ‘degrading’ to be aware of is when accessing the assays of a *SummarizedExperiment* via the `assay()` and `assays()` getters. Each of these getters has an argument `withDimnames` with a default value of `TRUE`; this copies the dimnames of the *SummarizedExperiment* to the returned assay. Consequently, this may ‘degrade’ the object(s) returned by `assay()` and `assays()` to *DelayedArray*. To avoid this, use `withDimnames = FALSE` in the call to `assay()` and `assays()`.

```
library(SummarizedExperiment)
# Construct a SummarizedExperiment with column names 'A' and 'B' and da_Rle as
# the assay data.
se <- SummarizedExperiment(da_Rle, colData = DataFrame(row.names = c("A", "B")))
se
#> class: SummarizedExperiment
#> dim: 105 2
#> metadata(0):
#> assays(1): ''
#> rownames: NULL
#> rowData names(0):
#> colnames(2): A B
#> colData names(0):

# NOTE: dimnames are copied, so the result is 'degraded'.
assay(se)
#> <105 x 2> DelayedMatrix object of type "integer":
#>      A   B
#> [1,] 1 15
#> [2,] 2 15
#> [3,] 2 15
#> [4,] 3 15
#> [5,] 3 15
#> ...
#> ... . .
```

```

#> [101,] 14 20
#> [102,] 14 20
#> [103,] 14 20
#> [104,] 14 20
#> [105,] 14 20
is(assay(se), "RleMatrix")
#> [1] FALSE

# NOTE: dimnames are not copied, so the result is not 'degraded'.
assay(se, withDimnames = FALSE)
#> <105 x 2> RleMatrix object of type "integer":
#>     [,1] [,2]
#>     [1,]    1   15
#>     [2,]    2   15
#>     [3,]    2   15
#>     [4,]    3   15
#>     [5,]    3   15
#>     ...   .
#>     [101,] 14 20
#>     [102,] 14 20
#>     [103,] 14 20
#>     [104,] 14 20
#>     [105,] 14 20
is(assay(se, withDimnames = FALSE), "RleMatrix")
#> [1] TRUE

```

As noted up front, the degradation isn't in and of itself a problem. However, if you are passing an object to a function that expects an *RleMatrix*, for example, some care needs to be taken to ensure that the object isn't accidentally 'degraded' along the way to a *DelayedMatrix*.

To summarise, the lessons here are:

- Modifying an instance of a *DelayedArray* subclass will almost always 'degrade' it to a *DelayedArray*.
- It is very easy to accidentally trigger this 'degradation'.
- This 'degradation' can complicate method dispatch.

15.3.2.5 Out-of-memory seeds

The *DelayedArray* framework really shines when working with out-of-memory seeds. It can give the user the "feel" of interacting with an ordinary R array but allows for the data to be stored on a local disk or even on a remote server, thus reducing (local) memory usage.

15.3.2.5.1 Local on-disk seeds

The **HDF5Array** package defines the *HDF5Array* class, a *DelayedArray* subclass for data stored on disk in a HDF5 file. The *seed* of an *HDF5Array* is a *HDF5ArraySeed*. It is important to note that creating a *HDF5Array* does not read the data into memory! The data remain on disk until requested (we'll see how to do this later on in the workshop).

```

library(HDF5Array)

hdf5_file <- file.path(
  "500_Effectively_Using_the_DelayedArray_Framework",
  "hdf5_mat.h5")

```

```

# NOTE: We can use rhdf5::h5ls() to take a look what is in the HDF5 file.
#       This is very useful when working interactively!
rhdf5::h5ls(hdf5_file)
#>   group      name      otype dclass dim
#> 0    / hdf5_mat H5I_DATASET INTEGER  x 2

# We can create the HDF5Array by first creating a HDF5ArraySeed and then
# creating the HDF5Array.
hdf5_seed <- HDF5ArraySeed(filepath = hdf5_file, name = "hdf5_mat")
da_hdf5 <- DelayedArray(seed = hdf5_seed)
da_hdf5
#> <105 x 2> HDF5Matrix object of type "integer":
#>   [,1] [,2]
#>  [1,]    1   15
#>  [2,]    2   15
#>  [3,]    2   15
#>  [4,]    3   15
#>  [5,]    3   15
#>  ...   .
#> [101,]   14  20
#> [102,]   14  20
#> [103,]   14  20
#> [104,]   14  20
#> [105,]   14  20

# Alternatively, we can create this in one go using the HDF5Array() constructor.
da_hdf5 <- HDF5Array(filepath = hdf5_file, name = "hdf5_mat")
da_hdf5
#> <105 x 2> HDF5Matrix object of type "integer":
#>   [,1] [,2]
#>  [1,]    1   15
#>  [2,]    2   15
#>  [3,]    2   15
#>  [4,]    3   15
#>  [5,]    3   15
#>  ...   .
#> [101,]   14  20
#> [102,]   14  20
#> [103,]   14  20
#> [104,]   14  20
#> [105,]   14  20

```

Other on-disk seeds are possible such as `fst`, `bigmemory`, `ff`, or `matter`.

15.3.2.5.2 Remote seeds

The `rhdf5client` package defines the `H5S_Array` class, a `DelayedArray` subclass for data stored on a remote HDF Server. The `seed` of an `H5S_Array` is a `H5S_ArraySeed`. It is important to note that creating a `H5S_Array` does not read the data into memory! The data remain on the server until requested.

```

library(rhdf5client)
#>
#> Attaching package: 'rhdf5client'
#> The following object is masked from 'package:tidygraph':

```

```
#>
#>      groups
da_h5s <- HSDS_Matrix(URL_hsds(), "/home/stvjc/hdf5_mat.h5")
da_h5s
#> <105 x 2> H5S_Matrix object of type "double":
#>      [,1] [,2]
#>     [1,]    1   15
#>     [2,]    2   15
#>     [3,]    2   15
#>     [4,]    3   15
#>     [5,]    3   15
#>     ...   .
#>     [101,]   14  20
#>     [102,]   14  20
#>     [103,]   14  20
#>     [104,]   14  20
#>     [105,]   14  20
```

15.3.2.6 So what seed should I use?

Notably, `da_mat`, `da_Mat`, `da_tbl`, `da_df`, `da_Rle`, `da_hdf5`, and `da_h5s` all “look” and “feel” much the same. The `DelayedArray` is a very light wrapper around the `seed` that formalises this consistent “look” and “feel”.

If your data always fit in memory, and it’s still comfortable to work with it interactively, then you should probably stick with using ordinary `matrix` and `array` objects. If, however, your data sometimes don’t fit in memory or it becomes painful to work with them when they are in-memory, you should consider a disk-backed `DelayedArray`. If you need to share a large dataset with a number of uses, and most users only need slices of it, you may want to consider a remote server-backed `DelayedArray`.

Not surprisingly, working with in-memory seeds will typically be faster than disk-backed or remote seeds; you are trading off memory-usage and data access times to obtain faster performance.

Below are some further opinionated recommendations.

Although the `DelayedArray` is a light wrapper, it does introduce some overhead. For example, operations on a `DelayedArray` with an `array` or `Matrix` seed will be slower than operating on the `array` or `Matrix` directly. For some operations this is almost non-existent (e.g., seed-aware operations in the `DelayedMatrixStats` package), but for others this overhead may be noticeable. Conversely, a potential upside to wrapping a `Matrix` in a `DelayedArray` is that it can now leverage the block-processing strategy of the `DelayedArray` framework.

If you need a disk-backed `DelayedArray`, I would recommend using `HDF5Array` as the backend at this time. HDF5 is a well-established scientific data format and the `HDF5Array` package is developed by the core Bioconductor team. Furthermore, several bioinformatics tools natively export HDF5 files (e.g., `kallisto`, `CellRanger` from 10x Genomics). One potential downside of HDF5 is the recent split into “Community” and “Enterprise Support” editions. It’s not yet clear how this will affect the HDF5 library or its community. Other on-disk backends may not yet have well-established formats or libraries (e.g., `TileDB`). A topic of on-going research in the Bioconductor community is to compare the performance of HDF5 to other disk-backed data stores. Another topic is how to best choose the layout of the data on disk, often referred to as the “chunking” of the data. For example, if the data is stored in column-major order then it might be expected that row-wise data access suffers.

One final comment. If you are trying to support both in-memory and on-disk data, it may be worth considering using `DelayedArray` for everything rather than trying to support both `DelayedArray` and ordinary arrays. This is particularly true for internal, non-user facing code. Otherwise your software may need separate code paths for `array` and `DelayedArray` inputs, although the need for separate branches is reducing. For example, when I initially added support for the `DelayedArray` framework in `bsseq` (~2 years ago), I opted to remove support

for ordinary arrays. However, more recently I added support for the DelayedArray framework in **minfi**, and there I opted to maintain support for ordinary arrays. I've found that as the DelayedArray framework matures and expands, it is increasingly common that the same code "just works" for both *array* and *DelayedArray* objects. Maintaining support for ordinary arrays can also be critical for popular Bioconductor packages, although this can be handled by having functions that accept an *array* as input simply wrapping them in a *DelayedArray* for internal computations.

15.3.3 Operating on *DelayedArray* objects

Now that we know what a *DelayedArray* is, its various subclasses, and how to construct these, we will discuss how to operate on *DelayedArray* objects.

We'll cover three fundamental concepts:

1. Delayed operations
2. Block-processing
3. Realization

These three concepts work hand-in-hand. Delayed operations are exactly that, the operation is recorded but performing the operation is delayed until the result is required; *realization* is the process of executing the delayed operations; and *block-processing* is used to perform operations, including *realization*, on blocks of the *DelayedArray*.

15.3.3.1 Learning goals

- Become familiar with the fundamental concepts of delayed operations, block-processing, and realization.
- **TODO** Construct a *DelayedArray* from scratch by writing data to a *RealizationSink*.

15.3.3.2 Delayed operations

A delayed operation is one that is not actually performed until the result is required. Here's a simple example of a delayed operations: taking the negative of every element of a *HDF5Matrix*.

```
da_hdf5
#> <105 x 2> HDF5Matrix object of type "integer":
#>      [,1] [,2]
#> [1,]    1   15
#> [2,]    2   15
#> [3,]    2   15
#> [4,]    3   15
#> [5,]    3   15
#> ...
#> [101,]  14  20
#> [102,]  14  20
#> [103,]  14  20
#> [104,]  14  20
#> [105,]  14  20
-da_hdf5
#> <105 x 2> DelayedMatrix object of type "integer":
#>      [,1] [,2]
#> [1,]   -1  -15
#> [2,]   -2  -15
#> [3,]   -2  -15
#> [4,]   -3  -15
```

```
#> [5,] -3 -15
#> ...
#> [101,] -14 -20
#> [102,] -14 -20
#> [103,] -14 -20
#> [104,] -14 -20
#> [105,] -14 -20
```

It may look like running `-da_hdf5` has taken the negative of every element. However, we can see this is in fact not the case by using the `showtree()` function to inspect the internal state of these objects⁹:

```
showtree(da_hdf5)
#> 105x2 integer: HDF5Matrix object
#> └ 105x2 integer: [seed] HDF5ArraySeed object

# NOTE: The "Unary iso op" line is 'recording' the `-' as a delayed operation.
showtree(~da_hdf5)
#> 105x2 integer: DelayedMatrix object
#> └ 105x2 integer: Unary iso op
#>   └ 105x2 integer: [seed] HDF5ArraySeed object
```

To further illustrate the idea, let's perform some delayed operations on a large `HDF5Array` and compare them to performing the operations on the equivalent `array`.

```
library(h5vcData)

tally_file <- system.file("extdata", "example.tally.hfs5", package = "h5vcData")
x_h5 <- HDF5Array::HDF5Array(tally_file, "/ExampleStudy/16/Coverages")
x_h5
#> <6 x 2 x 90354753> HDF5Array object of type "integer":
#> , , 1
#>   [,1] [,2]
#> [1,] 0 0
#> [2,] 0 0
#> ...
#> [5,] 0 0
#> [6,] 0 0
#>
#> ...
#>
#> , , 90354753
#>   [,1] [,2]
#> [1,] 0 0
#> [2,] 0 0
#> ...
#> [5,] 0 0
#> [6,] 0 0
x <- as.array(x_h5)

# Delayed operations are fast compared to ordinary operations!
system.time(x_h5 + 100L)
#> user system elapsed
#> 0.004 0.000 0.005
```

⁹You may like to use the `str()` function for more detailed and verbose output

Block-processing

Problem: I need to traverse the array and performing some operation(s) but can only load **n** elements into memory.

The operation(s) could be element-wise or block-wise.

Side note: at the heart of realization.

Side note: **n** is controlled by

```
getOption("DelayedArray.block.size")
```

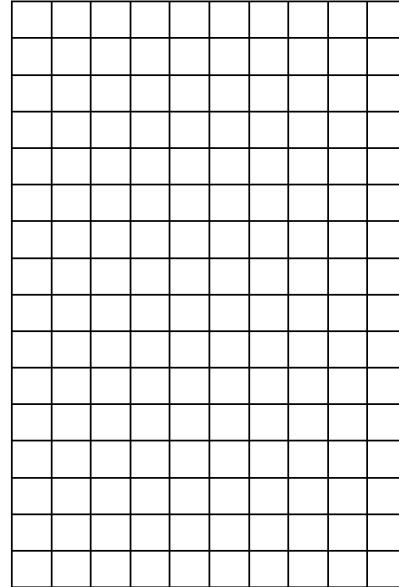


Figure 15.2:

```
system.time(x + 100L)
#>    user  system elapsed
#>  2.468   1.628   4.098

# Delayed operations can be chained
system.time(t(x_h5[1, , 1:100] + 100L))
#>    user  system elapsed
#>  0.016   0.000   0.017
```

Rather than modifying the data stored in the HDF5 file, which can be costly for large datasets, we've recorded the ‘idea’ of these operations as a tree of *DelayedOp* objects. Each node in the tree is represented by a *DelayedOp* object, of which there are 6 concrete subclasses:

Node type	Out-degree	Operation
<i>DelayedSubset</i>	1	Multi-dimensional single bracket subsetting
<i>DelayedAperm</i>	1	Extended <code>aperm()</code> (can drop dimensions)
<i>DelayedUnaryIsoOp</i>	1	Unary op that preserves the geometry (e.g., <code>-</code> , <code>log()</code>)
<i>DelayedDimnames</i>	1	Set dimnames
<i>DelayedNaryIsoOp</i>	N	N-ary op that preserves the geometry
<i>DelayedAbind</i>	N	<code>abind()</code>

15.3.3.3 Block-processing

Block-processing allows you to iterate over blocks of a *DelayedArray* in a manner that abstracts over the backend.

The following cartoon illustrates the basic idea of block-processing:

Each block is a row

E.g., `rowSums()`

```
RegularArrayGrid(  
  refdim = dim(x),  
  spacings = c(1L, ncol(x)))
```

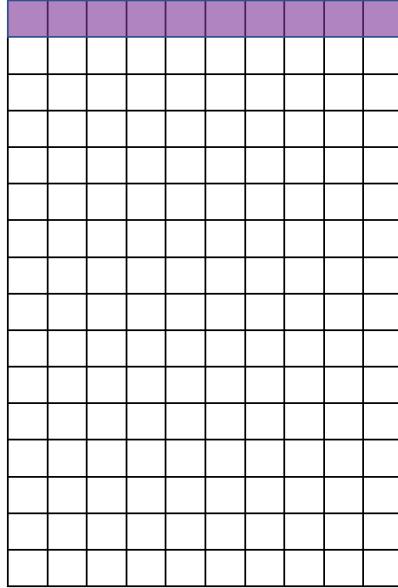


Figure 15.3:

Each block is a column

E.g., `colSums()`

```
RegularArrayGrid(  
  refdim = dim(x),  
  spacings = c(nrow(x), 1L))
```

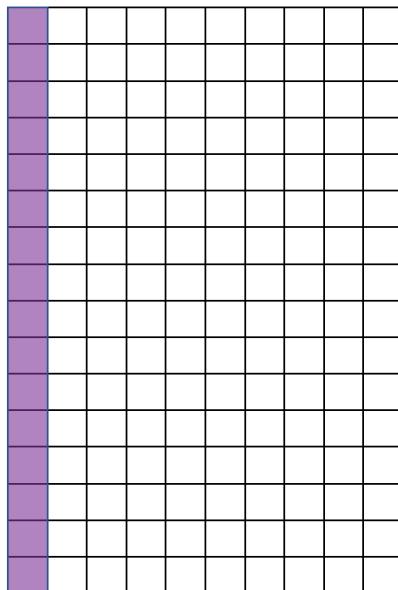


Figure 15.4:

Each block is a fixed number of columns

E.g., `colSums()`. More efficient if you can load > 1 columns' worth of data into memory.

```
RegularArrayGrid(  
  refdim = dim(x),  
  spacings = c(nrow(x), 5L))
```

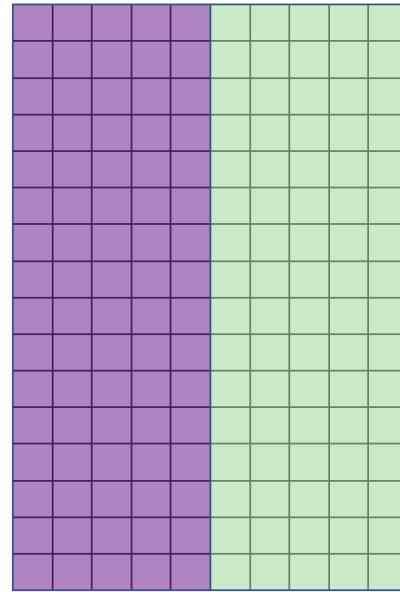


Figure 15.5:

Each block is a variable number of columns

E.g., `rowsum()`

```
ArbitraryArrayGrid(  
  tickmarks = list(  
    nrow(x),  
    c(4L, 7L, 9L, 10L)))
```

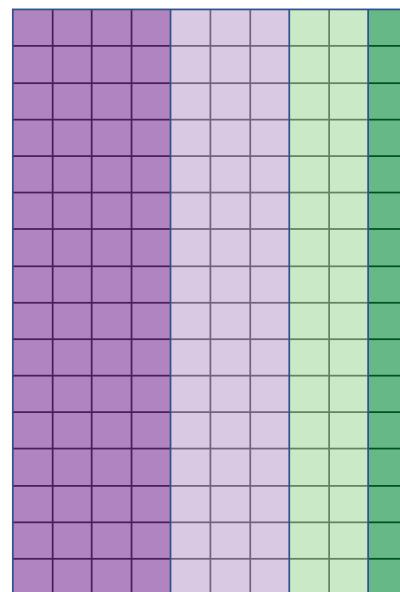


Figure 15.6:

Each block is the matrix

You probably don't want to do this!

```
RegularArrayGrid(
  refdim = dim(x),
  spacings = c(nrow(x), ncol(x))
```

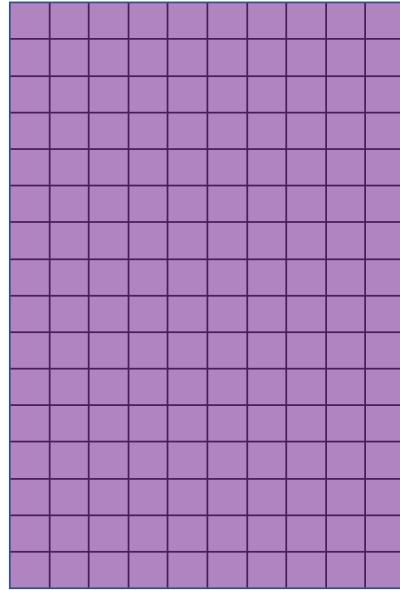


Figure 15.7:

Each block is “optimal”

E.g., when the data are chunked on disk in an HDF5 file.

```
blockGrid(
  x = x,
  block.shape = "hypercube")
```

block.shape can be one of:

- “hypercube”
- “scale”
- “first-dim-grows-first”
- “last-dim-grows-first”

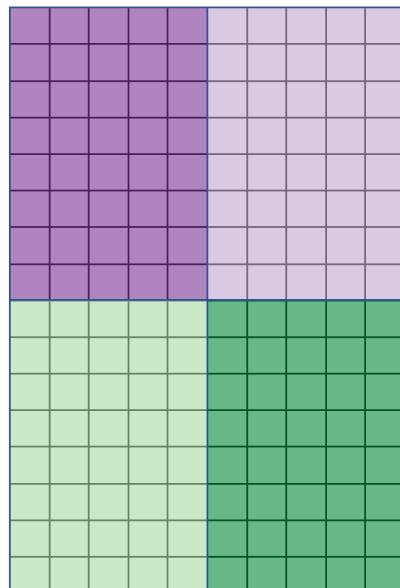


Figure 15.8:

To implement block-processing, we first construct an *ArrayGrid* over the *DelayedArray*. Each element of the *ArrayGrid* is called an *ArrayViewport*. We iterate over the *ArrayGrid*, extract the corresponding *ArrayViewport*, and load the data for that block into memory as an ordinary R *array* where it can be processed using ordinary R functions.

In pseudocode, we might implement block processing as follows:

```
# Construct an ArrayGrid over 'x'.
# NOTE: blockGrid() is explained below.
grid <- blockGrid(x)
for (b in seq_along(grid)) {
  # Construct an ArrayViewport using the b-th element of the ArrayGrid.
  viewport <- grid[[b]]
  # Read the block of data using the current 'viewport' applied to 'x'.
  block <- read_block(x, viewport)
  # Apply our function to 'block' (which is an ordinary R array).
  FUN(block)
}
```

In practice, when constructing the *ArrayGrid* for a particular operation, we need to consider a few issues:

1. Constraints on how much data we can bring into memory (the ‘maximum block length’).
2. The geometry of the *ArrayGrid*, which will control the order in which we access elements.
3. How the data are physically stored (‘chunking’ of the data on disk).

15.3.3.3.1 Maximum block length

There is a global option `getOption("DelayedArray.block.size")` for controlling how much data is brought into memory with default value corresponding to each block containing a maximum of 45 Mb of data. It’s good practice to respect this value, however, some algorithms may require this be ignored (e.g., if you require a full column’s worth of data for each block).

15.3.3.3.2 Chunking and data structure

Regardless of the backend, the data will be stored with a particular structure. For example, ordinary R arrays store the data in column-major order. The geometry of the data structure is especially relevant for HDF5 files.

The HDF5 format supports ‘chunking’ of data to maximize data access performance. From <https://support.hdfgroup.org/HDF5/doc/Advanced/Chunking/index.html>:

Datasets in HDF5 can represent arrays with any number of dimensions (up to 32). However, in the file this dataset must be stored as part of the 1-dimensional stream of data that is the low-level file. The way in which the multidimensional dataset is mapped to the serial file is called the layout. The most obvious way to accomplish this is to simply flatten the dataset in a way similar to how arrays are stored in memory, serializing the entire dataset into a monolithic block on disk, which maps directly to a memory buffer the size of the dataset. This is called a contiguous layout.

An alternative to the contiguous layout is the chunked layout. Whereas contiguous datasets are stored in a single block in the file, chunked datasets are split into multiple chunks which are all stored separately in the file. The chunks can be stored in any order and any position within the HDF5 file. Chunks can then be read and written individually, improving performance when operating on a subset of the dataset.

Although this sounds similar to the blocks used in block-processing, the two are distinct.

Chunks are used to determine how the data are stored, blocks are used to determine how the data are accessed via the *ArrayGrid*. That is, you can use a different *ArrayGrid* geometry to the chunk

geometry, but performance may be suboptimal.

15.3.3.3.3 Geometry of the *ArrayGrid*

The *ArrayGrid* could be a regularly spaced grid (a *RegularArrayGrid*) or a grid with arbitrary spacing (an *ArbitraryArrayGrid*).

The **DelayedArray** package includes some helper functions for setting up *ArrayGrid* instances with particular geometries and other properties.

The `blockGrid()` function returns the “optimal” *ArrayGrid* for block-processing. The grid is “optimal” in the sense that:

- It’s “compatible” with the chunk grid (i.e. with `chunkGrid(x)` or with the chunk grid supplied via the `chunk.grid` argument). That is, the chunks are contained in the blocks. In other words, chunks never cross block boundaries.
- Its “resolution” is such that the blocks have a length that is as close as possible to (but does not exceed) `block maxlen`. An exception is when some chunks are already $\geq \text{block maxlen}$, in which case the returned grid is the same as the chunk grid.

15.3.3.3.4 Block-processing in practice

In practice, most block-processing algorithms can be implemented by constructing an *ArrayGrid* with a suitable geometry and then using `DelayedArray::blockApply()` or `DelayedArray::blockReduce()`.

As an example, let’s compute row and column medians of `da_hdf5`. We’ll start by constructing *ArrayGrid* instances over the rows and columns of `da_hdf5`.

```
# NOTE: Making block-processing verbose to show what's going on under the hood.
DelayedArray:::set_verbose_block_processing(TRUE)
#> [1] FALSE
system.time(
  row_medians <- blockApply(
    x = da_hdf5,
    FUN = median,
    grid = RegularArrayGrid(
      refdim = dim(da_hdf5),
      spacings = c(1L, ncol(da_hdf5))))
)
#>   user   system elapsed
#> 0.120   1.212   1.822
head(row_medians)
#> [[1]]
#> [1] 8
#>
#> [[2]]
#> [1] 8.5
#>
#> [[3]]
#> [1] 8.5
#>
#> [[4]]
#> [1] 9
#>
#> [[5]]
#> [1] 9
#>
```

```
#> [[6]]
#> [1] 9
system.time(
  col_medians <- blockApply(
    x = da_hdf5,
    FUN = median,
    grid = RegularArrayGrid(
      refdim = dim(da_hdf5),
      spacings = c(nrow(da_hdf5), 1L))))
#>   user  system elapsed
#> 0.064  0.112  0.257
head(col_medians)
#> [[1]]
#> [1] 10
#>
#> [[2]]
#> [1] 18
```

That works, but is somewhat slow for computing row maximums because we read from the HDF5 file `nrow(da_hdf5)` (105) times. We would be better off loading multiple rows of data per block. There are several ways to do this; here, we'll use `DelayedArray::blockGrid()` to construct an optimal `ArrayGrid` and `matrixStats::rowMedians()` and `matrixStats::colMedians()` to compute the row and column maximums.

```
system.time(
  row_medians <- blockApply(
    x = da_hdf5,
    FUN = matrixStats::rowMedians,
    grid = blockGrid(
      x = da_hdf5,
      block.shape = "first-dim-grows-first")))
#> Processing block 1/1 ... OK
#>   user  system elapsed
#> 0.02  0.00  0.02
head(row_medians)
#> [[1]]
#> [1] 8.0 8.5 8.5 9.0 9.0 9.0 9.5 9.5 9.5 9.5 10.0 10.0 10.0 10.0
#> [15] 10.0 11.0 11.0 11.0 11.0 11.0 11.0 11.5 11.5 11.5 11.5 11.5 11.5 11.5
#> [29] 12.0 12.0 12.0 12.5 12.5 12.5 12.5 13.0 13.0 13.0 13.0 13.0 13.0 13.0
#> [43] 13.0 13.0 13.0 13.5 13.5 13.5 14.0 14.0 14.0 14.0 14.0 14.0 14.0 14.5
#> [57] 14.5 14.5 14.5 14.5 14.5 14.5 14.5 14.5 14.5 14.5 14.5 14.5 15.5 15.5
#> [71] 15.5 15.5 15.5 15.5 15.5 15.5 15.5 16.0 16.0 16.0 16.0 16.0 16.0 16.0
#> [85] 16.0 16.5 16.5 16.5 16.5 16.5 17.0 17.0 17.0 17.0 17.0 17.0 17.0 17.0
#> [99] 17.0 17.0 17.0 17.0 17.0 17.0 17.0
system.time(
  col_medians <- blockApply(
    x = da_hdf5,
    FUN = matrixStats::colMedians,
    grid = blockGrid(
      x = da_hdf5,
      block.shape = "last-dim-grows-first")))
#> Processing block 1/1 ... OK
#>   user  system elapsed
#> 0.020  0.000  0.018
head(col_medians)
```

```
#> [[1]]
#> [1] 10 18
```

For larger problems, we can improve performance by processing blocks in parallel by passing an appropriate `BiocParallelParam` object via the `bpparam` argument of `blockApply()` and `blockReduce()`.

Although `blockApply()` and `blockReduce()` cover most of the block-processing tasks, sometimes you may need to implement the block-processing at a lower level. For example, you may need to iterate over multiple `DelayedArray` objects or your FUN returns an object equally large (or larger) than the `block`. The details of these abstractions are still being worked out, but some likely candidates include methods that conceptually do:

- `blockMapply()`
- `blockApplyWithRealization()`
- `blockMapplyWithRealization()`

15.3.3.4 Realization

Realization is the process of taking a `DelayedArray`, executing any delayed operations, and returning the result as a new `DelayedArray` with the appropriate backend.

Returning to our earlier example with data from the `h5vcData` package:

```
x_h5
#> <6 x 2 x 90354753> HDF5Array object of type "integer":
#> , , 1
#>      [,1] [,2]
#> [1,]    0    0
#> [2,]    0    0
#> ...   .   .
#> [5,]    0    0
#> [6,]    0    0
#>
#> ...
#>
#> , , 90354753
#>      [,1] [,2]
#> [1,]    0    0
#> [2,]    0    0
#> ...   .   .
#> [5,]    0    0
#> [6,]    0    0
showtree(x_h5)
#> 6x2x90354753 integer: HDF5Array object
#> └ 6x2x90354753 integer: [seed] HDF5ArraySeed object
y <- t(x_h5[1, , 1:10000000] + 100L)
showtree(y)
#> 10000000x2 integer: DelayedMatrix object
#> └ 10000000x2 integer: Unary iso op
#>   └ 10000000x2 integer: Aperm (perm=c(3,2))
#>     └ 1x2x10000000 integer: Subset
#>       └ 6x2x90354753 integer: [seed] HDF5ArraySeed object

# Realize the result in-memory
z <- realize(y, BACKEND = NULL)
```

```

# NOTE: 'z' does not carry any delayed operations.
showtree(z)
#> 10000000x2 integer: DelayedMatrix object
#> └ 10000000x2 integer: [seed] matrix object

# Realize the result on-disk in an autogenerated HDF5 file
z_h5 <- realize(y, BACKEND = "HDF5Array")
#> Processing block 1/2 ... OK
#> Processing block 2/2 ... OK
# NOTE: 'z_h5' does not carry any delayed operations.
showtree(z_h5)
#> 10000000x2 integer: HDF5Matrix object
#> └ 10000000x2 integer: [seed] HDF5ArraySeed object
path(z_h5)
#> [1] "/tmp/RtmpoJVoyV/HDF5Array_dump/auto00001.h5"

# NOTE: The show() method performs realization on the first few and last few
#       elements in order to preview the result
y
#> <10000000 x 2> DelayedMatrix object of type "integer":
#>      [,1] [,2]
#>     [1,] 100 100
#>     [2,] 100 100
#>     [3,] 100 100
#>     [4,] 100 100
#>     [5,] 100 100
#>     ...
#>     [9999996,] 100 100
#>     [9999997,] 100 100
#>     [9999998,] 100 100
#>     [9999999,] 100 100
#>     [10000000,] 100 100

```

Realization uses block-processing under the hood:

```

DelayedArray:::set_verbose_block_processing(TRUE)
#> [1] TRUE
realize(y, BACKEND = "HDF5Array")
#> Processing block 1/2 ... OK
#> Processing block 2/2 ... OK
#> <10000000 x 2> HDF5Matrix object of type "integer":
#>      [,1] [,2]
#>     [1,] 100 100
#>     [2,] 100 100
#>     [3,] 100 100
#>     [4,] 100 100
#>     [5,] 100 100
#>     ...
#>     [9999996,] 100 100
#>     [9999997,] 100 100
#>     [9999998,] 100 100
#>     [9999999,] 100 100
#>     [10000000,] 100 100
DelayedArray:::set_verbose_block_processing(FALSE)

```

```
#> [1] TRUE
```

15.4 What's out there already?

UP TO HERE

- DelayedMatrixStats
- beachmat

15.4.0.1 Learning goal

- Learn of existing functions and packages for constructing and computing on DelayedArray objects, avoiding the need to re-invent the wheel.

15.5 Incorporating DelayedArray into a package

15.5.1 Writing algorithms to process *DelayedArray* instances

15.5.1.1 Learning goals

- Learn common design patterns for writing performant code that operates on a DelayedArray.
- Evaluate whether an existing function that operates on an ordinary array can be readily adapted to work on a DelayedArray.
- Reason about potential bottlenecks in algorithms operating on DelayedArray objects.

15.5.1.2 Learning objectives

- Take a function that operates on rows or columns of a matrix and apply it to a DelayedMatrix.
- Use block-processing on a *DelayedArray* to compute:
 - A univariate (scalar) summary statistic (e.g., `max()`).
 - A multivariate (vector) summary statistic (e.g., `colSum()` or `rowMean()`).
 - A multivariate (array-like) summary statistic (e.g., `rowRanks()`).
 - Design an algorithm that imports data into a DelayedArray.

15.6 Questions and discussion

This section will be updated to address questions and to summarise the discussion from the presentation of this workshop at BioC2018.

15.7 TODOs

- Use `BiocStyle`?
- Show packages depend on one another, with HDF5Array as the root (i.e. explain the HDF5 stack)
- Use `suppressPackageStartupMessages()` or equivalent.
- Note that we'll be focusing on numerical array-like data, i.e. no real discussion of `GDSArray`.
- Remove `memuse` dependency

- Link to my useR talk and slides

Chapter 16

510: Maintaining your Bioconductor package

16.1 Instructor name and contact information

- Nitesh Turaga¹ (nitesh.turaga@roswellpark.org)

16.2 Workshop Description

Once an R package is accepted into Bioconductor, maintaining it is an active responsibility undertaken by the package developers and maintainers. In this short workshop, we will cover how to maintain a Bioconductor package using existing infrastructure. Bioconductor hosts a range of tools which maintainers can use such as daily build reports, landing page badges, RSS feeds, download stats, support site questions, and the bioc-devel mailing list. Some packages have their own continuous integration hook setup on their github pages which would be an additional tool maintainers can use to monitor their package. We will also highlight one particular git practice which need to be done while updating and maintaining your package on out git system.

16.2.1 Pre-requisites

Accepted Bioconductor package or plans to contribute a Bioconductor package in the future.

16.2.2 Participation

Students will be expected to follow along with their laptops if they choose to, although it is not needed.

16.2.3 Time outline: 50 mins short workshop

- Introduction: 10 mins
 - Why maintain your package
 - Infrastructure used for maintenance
- Outline how to use infrastructure: 35 mins

¹Roswell Park Comprehensive Cancer Center, Buffalo, NY

- Build report
- Version numbering
- Landing page badges
- Download statistics
- RSS feeds
- Support site, Bioc-devel mailing list
- Github
 - * Sync package with Github / Bioconductor before every change
 - * Github issues
- Round up of resources available: 5 mins
- Acknowledgements

16.2.4 Workshop goals and objectives

- Gain knowledge of how to track Bioconductor packages via download statistics, RSS feeds.
- Understand the importance of supporting their user community and how to do so using the support site and bioc-devel mailing list.
- Maintain their package and fix bugs using build reports, continuous integration tools.
- Update their package consistently on both Github and Bioconductor private git server.

16.3 Introduction

Maintainer - Who should people contact if they have a problem with this package? You must have someones name and email address here. You must make sure this is up to date and accurate.

Maintaining it is an active responsibility undertaken by the package developers and maintainers. Users contact developers for various reasons.

Keep your email up to date in the DESCRIPTION file!

Maintainer: Who to complain to <your_fault@fix.it>

or

```
Authors@R: c(
  person("Complain", "here", email="your_fault@fix.it", role=c("aut", "cre"),
  person("Don't", "Complain", role="aut"))
)
```

NOTE: There should be only 1 maintainer (one “cre”). But all authors get credit for the package.

Interchangeable terminology

Bioconductor maintainers <-> Bioconductor developers <-> Package developer <-> developer

16.3.1 Why maintain your package

1. Changes in underlying code (upstream dependencies of your package) might break your package.
2. Packages using web resources might fail because of change of location of these resources.
3. Bug fixes reported or discovered due to usage.
4. Feature requests from users.
5. Improvements in the science behind your package, which lead to updates.

Hostname OS	Arch (*)	Platform label (**)	R version	Installed pkgs
malbec1 Linux (Ubuntu 16.04.1 LTS)	x86_64	x86_64-linux-gnu	3.5.1 RC (2018-06-24 r74929) -- "Feather Spray"	3246
tokay1 Windows Server 2012 R2 Standard	x64	mingw32 / x86_64-w64-mingw32	3.5.1 RC (2018-06-24 r74929) -- "Feather Spray"	3087
merida1 OS X 10.11.6 El Capitan	x86_64	x86_64-apple-darwin15.6.0	3.5.1 Patched (2018-07-12 r74967) - - "Feather Spray"	3078

Click on any hostname to see more info about the system (e.g. compilers) (*) as reported by 'uname -p', except on Windows and Mac OS X (**) as reported by 'gcc -v'

Figure 16.1: Build machines on DEVEL

6. It leads to wide spread usage of your package.
7. It's **required**, also a good practice that users appreciate.
8. Poorly maintained packages get deprecated if they fail build and check on Bioconductor during release cycles. (Bioconductor is fairly serious about this)

16.3.2 Infrastructure used for maintenance

Build machines are dedicated towards building your package *daily*. As a maintainer it's useful to check your package build report once a week, both in the current RELEASE and DEVEL cycle.

<http://bioconductor.org/checkResults/>

Build machines cover three platforms (Linux, Windows and OS X) over RELEASE and DEVEL versions of Bioconductor.

Version control - Since last year, GIT has been the primary version control system.

16.4 Outline how to use infrastructure

Everything you need is on the Bioconductor website. As a developer the link with most important resources, <http://bioconductor.org/developers/>

16.4.1 Build report

Report of the build status of your package. It posts at around 11AM EST (can change sometimes) and is located on the time stamp at the top of the page.

Two versions of the build report.

- Bioconductor RELEASE version
<http://bioconductor.org/checkResults/release/bioc-LATEST/>
- Bioconductor development version
<http://bioconductor.org/checkResults-devel/bioc-LATEST/>

Six build machines in total at Bioconductor.

DEVEL (3 for devel)

1. Linux (malbec1)
2. Windows (tokay1)
3. OS X (merida1)

RELEASE (3 for release)

Hostname OS	Arch (*)	Platform label (**)	R version	Installed pkgs
malbec2	Linux (Ubuntu 16.04.1 LTS)	x86_64	x86_64-linux-gnu	3.5.1 RC (2018-06-24 r74929) -- "Feather Spray"
tokay2	Windows Server 2012 R2 Standard	x64	mingw32 / x86_64-w64-mingw32	3.5.1 RC (2018-06-24 r74929) -- "Feather Spray"
merida2	OS X 10.11.6 El Capitan	x86_64	x86_64-apple-darwin15.6.0	3.5.1 Patched (2018-07-12 r74967) - - "Feather Spray"

Click on any hostname to see more info about the system (e.g. compilers) () as reported by 'uname -p', except on Windows and Mac OS X (***) as reported by 'gcc -v'*

Figure 16.2: Build machines on RELEASE

INSTALL report for a4 on malbec2

This page was generated on 2018-07-17 10:34:04 -0400 (Tue, 17 Jul 2018).

Package 1/1560	Hostname OS / Arch	INSTALL	BUILD	CHECK	BUILD BIN
a4 1.28.0 Tobias Verbeke Snapshot Date: 2018-07-16 16:45:11 -0400 (Mon, 16 Jul 2018) URL: https://git.bioconductor.org/packages/a4 Branch: RELEASE_3_7 Last Commit: e91a8c1 Last Changed Date: 2018-04-30 10:35:18 -0400 (Mon, 30 Apr 2018)	malbec2 Linux (Ubuntu 16.04.1 LTS) / x86_64 tokay2 Windows Server 2012 R2 Standard / x64 merida2 OS X 10.11.6 El Capitan / x86_64	OK	OK	OK	OK

Summary

Package: a4
Version: 1.28.0
Command: /home/biocbuild/bbs-3.7-bioc/R/bin/R CMD INSTALL a4
StartedAt: 2018-07-16 18:37:00 -0400 (Mon, 16 Jul 2018)
EndedAt: 2018-07-16 18:37:15 -0400 (Mon, 16 Jul 2018)
ElapsedTime: 14.8 seconds
RetCode: 0
Status: OK

Command output

```
#####
#####
### Running command:
###   /home/biocbuild/bbs-3.7-bioc/R/bin/R CMD INSTALL a4
###
```

Figure 16.3: a4 R CMD INSTALL

1. Linux (malbec2)
2. Windows (tokay2)
3. OS X (merida2)

16.4.1.1 What is running on these machines?

16.4.1.1.1 INSTALL

Command:

```
/home/biocbuild/bbs-3.7-bioc/R/bin/R CMD INSTALL a4
```

<http://bioconductor.org/checkResults/release/bioc-LATEST/a4/malbec2-install.html>

16.4.1.1.2 BUILD

Command:

```
/home/biocbuild/bbs-3.7-bioc/R/bin/R CMD build a4
```

<http://bioconductor.org/checkResults/release/bioc-LATEST/a4/malbec2-buildsrc.html>

BUILD report for a4 on malbec2						
This page was generated on 2018-07-17 10:34:04 -0400 (Tue, 17 Jul 2018).						
Package 1/1560	Hostname OS / Arch	INSTALL	BUILD	CHECK	BUILD BIN	
a4 1.28.0 Tobias Verbeke Snapshot Date: 2018-07-16 16:45:11 -0400 (Mon, 16 Jul 2018) URL: https://git.bioconductor.org/packages/a4 Branch: RELEASE_3_7 Last Commit: e91a8c1 Last Changed Date: 2018-04-30 10:35:18 -0400 (Mon, 30 Apr 2018)	malbec2 Linux (Ubuntu 16.04.1 LTS) / x86_64 tokay2 Windows Server 2012 R2 Standard / x64 merida2 OS X 10.11.6 El Capitan / x86_64	OK	[OK]	OK	OK	OK
		OK	OK	OK	OK	OK
		OK	OK	OK	OK	OK

Summary						
Package: a4 Version: 1.28.0 Command: /home/biocbuild/bbs-3.7-bioc/R/bin/R CMD build --keep-empty-dirs --no-resave-data a4 StartedAt: 2018-07-16 19:21:15 -0400 (Mon, 16 Jul 2018) EndedAt: 2018-07-16 19:23:33 -0400 (Mon, 16 Jul 2018) ElapsedTime: 138.4 seconds RetCode: 0 Status: OK PackageFile: a4_1.28.0.tar.gz PackageFileSize: 950.4 KiB						

Command output						
##### ##### Running command: #### ### /home/biocbuild/bbs-3.7-bioc/R/bin/R CMD build --keep-empty-dirs --no-resave-data a4 ###						

Figure 16.4: a4 R CMD build

16.4.1.1.3 CHECK

Command:

```
/home/biocbuild/bbs-3.7-bioc/R/bin/R CMD check a4_1.28.0.tar.gz
```

<http://bioconductor.org/checkResults/release/bioc-LATEST/a4/malbec2-checksrc.html>

16.4.1.1.4 BUILD BIN

This makes the tar ball which is specific to the platform and propagates it to main repository.

The build machines display a status for each package. They indicate different things.

TIMEOUT - INSTALL, BUILD, CHECK or BUILD BIN of package took more than 40 minutes

ERROR - INSTALL, BUILD or BUILD BIN of package failed, or CHECK produced errors

WARNINGS - CHECK of package produced warnings

OK - INSTALL, BUILD, CHECK or BUILD BIN of package was OK

NotNeeded - INSTALL of package was not needed (click on glyph to see why)

skipped - CHECK or BUILD BIN of package was skipped because the BUILD step failed

NA- BUILD, CHECK or BUILD BIN result is not available because of an anomaly in the Build System

NOTE: The build machines run these commands a little differently and the maintainers need not do it the same way.

16.4.1.2 Exercise:

Question:

CHECK report for a4 on malbec2

This page was generated on 2018-07-17 10:34:04 -0400 (Tue, 17 Jul 2018).

Package 1/1560	Hostname OS / Arch	INSTALL	BUILD	CHECK	BUILD BIN
a4 1.28.0 Tobias Verbeke Snapshot Date: 2018-07-16 16:45:11 -0400 (Mon, 16 Jul 2018) URL: https://git.bioconductor.org/packages/a4 Branch: RELEASE_3_7 Last Commit: e91a8c1 Last Changed Date: 2018-04-30 10:35:18 -0400 (Mon, 30 Apr 2018)	malbec2 Linux (Ubuntu 16.04.1 LTS) / x86_64 tokay2 Windows Server 2012 R2 Standard / x64 merida2 OS X 10.11.6 El Capitan / x86_64	OK	OK	OK	OK

Summary

```
ge: a4
n: 1.28.0
and: /home/biocbuild/bbs-3.7-bioc/R/bin/R CMD check --install=check:a4.install-out.txt --library=/home/biocbuild/bbs-3.7-bioc/R/library --no-vignettes --timings a4_1.28.0.tar.gz
dAt: 2018-07-16 22:18:01 -0400 (Mon, 16 Jul 2018)
lAt: 2018-07-16 22:19:04 -0400 (Mon, 16 Jul 2018)
edTime: 63.2 seconds
de: 0
:: OK
Dir: a4.Rcheck
ngs: 0
```

Command output

```
#####
## Running command:
## /home/biocbuild/bbs-3.7-bioc/R/bin/R CMD check --install=check:a4.install-out.txt --library=/home/biocbuild/bbs-3.7-bioc/R/
###
```

Figure 16.5: a4 R CMD check

BUILD BIN report for a4 on tokay2

This page was generated on 2018-07-17 10:49:49 -0400 (Tue, 17 Jul 2018).

Package 1/1560	Hostname OS / Arch	INSTALL	BUILD	CHECK	BUILD BIN
a4 1.28.0 Tobias Verbeke Snapshot Date: 2018-07-16 16:45:11 -0400 (Mon, 16 Jul 2018) URL: https://git.bioconductor.org/packages/a4 Branch: RELEASE_3_7 Last Commit: e91a8c1 Last Changed Date: 2018-04-30 10:35:18 -0400 (Mon, 30 Apr 2018)	malbec2 Linux (Ubuntu 16.04.1 LTS) / x86_64 tokay2 Windows Server 2012 R2 Standard / x64 merida2 OS X 10.11.6 El Capitan / x86_64	OK	OK	OK	OK

Summary

```
Package: a4
Version: 1.28.0
Command: rm -rf a4.buildbin-libdir && mkdir a4.buildbin-libdir && C:\Users\biocbuild\bbs-3.7-bioc\R\bin\R.exe CMD INSTALL --merge-multiarch --build --library=a4.buildbin-libdir
StartedAt: 2018-07-17 06:00:36 -0400 (Tue, 17 Jul 2018)
EndedAt: 2018-07-17 06:01:01 -0400 (Tue, 17 Jul 2018)
ElapsedTime: 24.3 seconds
RetCode: 0
Status: OK
PackageFile: a4_1.28.0.zip
PackageFileSize: 858 KIB
```

Command output

```
#####
## Running command:
## /rm -rf a4.buildbin-libdir && mkdir a4.buildbin-libdir && C:\Users\biocbuild\bbs-3.7-bioc\R\bin\R.exe CMD INSTALL --merge-mu
###
```

Figure 16.6: a4 buildbin

Package propagation status is indicated by one of the following LEDs

- YES: Package was propagated because it didn't previously exist or version was bumped
- NO: Package was not propagated because of a problem (impossible dependencies, or version lower than what is already propagated)
- UNNEEDED: Package was not propagated because it is already in the repository with this version. A version bump is required in order to propagate it

Figure 16.7: Lights next to the build-bin

Filter packages on the DEVEL build machines which have a WARNING on their build report. List the top 5 packages from this filter.

Question:

What is the timestamp on DEVEL build report of the package BiocParallel?

16.4.2 Version Numbering

Version numbering is crucial to getting your package to build on the Bioconductor build machines. Every time you want to push a change, it's important that you make a version number update if you want to your package to propagate to your users. The package is not built unless there is a version number update.

<https://bioconductor.org/developers/how-to/version-numbering/>

16.4.2.1 Exercise:

Question:

A very common developer question is, "I made a commit and pushed to my package, but the new build was not triggered. Why wasn't the build triggered?"

The immediate response is to check if they issued a version bump. How would someone check this?

Question:

What is the version number next to the RELEASE and devel versions of your package? Do they align with Bioconductor policy as given on <https://bioconductor.org/developers/how-to/version-numbering/> ?

16.4.3 Landing page badges

What are badges?

BiocGenerics



Figure 16.8: BiocGenerics badges

Badges can indicate the **HEALTH** and **reliability** of a package. A reason users look at the badges is to get a sense of which packages to use in their analysis. Active maintainers, useful posts and how long the package is in Bioconductor sway users towards packages.

Badges indicate the following:

1. **platforms** availability of the package
2. **downloads** statistics
3. **posts** on support.bioconductor.org which are related to that package based on the **tag**.
4. **in Bioc** how long the package has been in Bioconductor.
5. **build** current build status.

Where can I find these badges?

- Landing page to go to **RELEASE**
bioconductor.org/packages/release/BiocGenerics
- Landing page to go to **devel**
bioconductor.org/packages/devel/BiocGenerics

NOTE: After a push is made it can take up to 48 hours for it to appear on the build report and landing pages - i.e. we do one pull, build, check, propagate in 24 hours - so if your commit was after the pull for the day it wont be until the following day

16.4.4 Download statistics

The number reported next to each package name is the download score, that is, the average number of distinct IPs that hit the package each month for the last 12 months (not counting the current month).

<http://bioconductor.org/packages/stats/>

Top 75 packages as of July 17th 3:15pm EST,

BiocGenerics Download statistics,

16.4.4.1 Exercise:

Question:

Top 75

- | | | |
|---|--|---|
| 1 BiocInstaller (31146) | 26 DESeq2 (7545) | 51 vsn (2711) |
| 2 BiocGenerics (24929) | 27 geneplotter (7349) | 52 Gviz (2640) |
| 3 IRanges (22034) | 28 affy (5980) | 53 GOSemSim (2638) |
| 4 S4Vectors (21872) | 29 BSgenome (5890) | 54 fgsea (2633) |
| 5 Biobase (20703) | 30 affyio (5657) | 55 DOSE (2576) |
| 6 AnnotationDbi (18649) | 31 rhd5 (5476) | 56 clusterProfiler (2525) |
| 7 zlibbioc (17062) | 32 RBGL (5285) | 57 AnnotationForge (2334) |
| 8 GenomicRanges (16350) | 33 multtest (5260) | 58 Category (2319) |
| 9 limma (15702) | 34 Rgraphviz (5141) | 59 pcaMethods (2303) |
| 10 XVector (15057) | 35 impute (4832) | 60 ComplexHeatmap (2243) |
| 11 GenomeInfoDb (14009) | 36 VariantAnnotation (4745) | 61 topGO (2202) |
| 12 Biostrings (13711) | 37 qvalue (4474) | 62 phyloseq (2116) |
| 13 BiocParallel (13345) | 38 GEOquery (4177) | 63 OrganismDbi (2114) |
| 14 SummarizedExperiment (13332) | 39 ShortRead (4055) | 64 GOstats (2062) |
| 15 annotate (11076) | 40 AnnotationHub (4010) | 65 EBImage (2053) |
| 16 GenomicAlignments (10425) | 41 ensemblDb (3825) | 66 BiocStyle (2039) |
| 17 biomaRt (10316) | 42 ProtGenerics (3660) | 67 pathview (2022) |
| 18 rtracklayer (10116) | 43 interactiveDisplayBase (3525) | 68 KEGGgraph (1976) |
| 19 DelayedArray (10060) | 44 GSEABase (3313) | 69 tximport (1970) |
| 20 genefilter (9878) | 45 DESeq (3198) | 70 aroma.light (1813) |
| 21 Rsamtools (9750) | 46 biovizBase (3082) | 71 illuminaio (1802) |
| 22 graph (8859) | 47 sva (2964) | 72 biomformat (1785) |
| 23 GenomicFeatures (8727) | 48 DNAcopy (2917) | 73 Rsubread (1738) |
| 24 edgeR (8388) | 49 AnnotationFilter (2805) | 74 ggbio (1716) |
| 25 preprocessCore (7657) | 50 KEGGREST (2718) | 75 gcrma (1654) |

Figure 16.9: Top 75

Download stats for software package BiocGenerics

This page was generated on 2018-07-13 07:00:02 -0400 (Fri, 13 Jul 2018).

BiocGenerics home page: [release version](#), [devel version](#).

Number of downloads for software package BiocGenerics, year by year, from 2018 back to 2009 (years with no downloads are omitted):

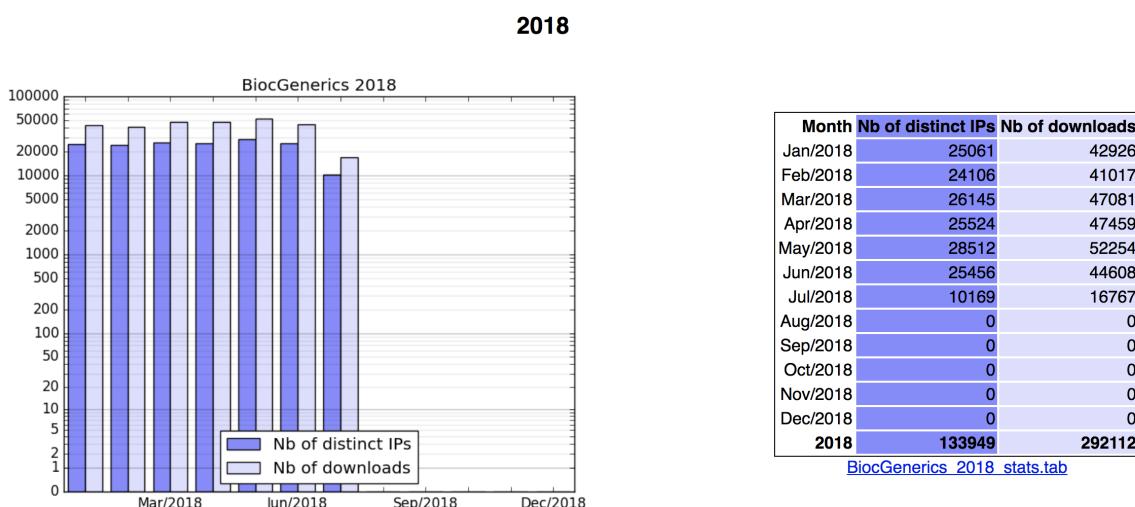


Figure 16.10: BiocGenerics Download Statistics

Get the download statistics to the package you maintain or the package you use the most. Next, get the month in 2017 with the most number of downloads.

16.4.5 RSS feeds

RSS feeds help developers keep track of the development going on in Bioconductor across all packages. There are multiple ways to use RSS feeds, and you can use the application of your choice to subscribe to the feeds.

Blog which shows the 12 best RSS reader apps across all platforms.

Links for RSS feeds,

<http://bioconductor.org/developers/rss-feeds/>

Git log published on the website. The git log hosts the last 500 commits to bioconductor across all packages in the development branch of Bioconductor.

<http://bioconductor.org/developers/gitlog/>

16.4.5.1 Exercise:

Question:

Take a package of your choice and subscribe to the RSS feed. Use a tool from the given link, or one of your own liking. (I use "feedly" the chrome extension)

16.4.6 Support site, Bioc-devel mailing list

Support infrastructure set up for Bioconductor **users** to be in touch with maintainers.

16.4.6.1 Mailing list

The mailing list **bioc-devel** is for all questions related package development, infrastructure, package support. If a question is misplaced, the author will be pointed to the correct location to ask the question.

<https://stat.ethz.ch/mailman/listinfo/bioc-devel>

16.4.6.2 Support site

Public support site, with easy to use interface to ask questions among the broader community. Any and all end-user questions are welcome here.

<http://support.bioconductor.org/>

16.4.7 GitHub

Optional, but recommended.

GIT Logs

This is a list of recent commits to git.bioconductor.org, the master(development) branch of the Bioconductor GIT repository.

This list is also available as an [RSS feed \(master branch\)](#), and [RSS feed \(release branch\)](#).

```
Package: scDD
Commit: e682bd3a8d3a3d896281fc994a841e82c6baf03b
Author: kdkorthauer <kdkorthauer@gmail.com>
Date: 2018-07-16 09:44:25 -0400
Commit message:
```

```
Merge remote-tracking branch 'upstream/master'
```

```
Package: scDD
Commit: 8f7cc2c137327d106c8f39ef455241434e6548f2
Author: kdkorthauer <kdkorthauer@gmail.com>
Date: 2018-07-16 09:42:08 -0400
Commit message:
```

```
Update installation instructions to use BiocManager
```

```
Package: dmrseq
Commit: a1ae095accd68e59476deddad424a65459de66fd
Author: kdkorthauer <kdkorthauer@gmail.com>
Date: 2018-07-16 09:42:38 -0400
Commit message:
```

```
Update installation instructions to use BiocManager
```

Figure 16.11: Git log on Jul 16th 2018 at 10:30 AM EST

16.4.7.1 Sync package with Github / Bioconductor before every change

Do this every time there is an update on your package from your end, or, before you start developing.

Help with Git (most canonical location),

<http://bioconductor.org/developers/how-to/git/>

Sync package from Bioconductor repository to Github repository

<http://bioconductor.org/developers/how-to/git/sync-existing-repositories/>

16.4.7.2 Github issues

Github issues are more developer centric than mailing list and support site questions. This needs to be posted directly on the development repositories of individual packages. Each package maintainer can choose to do their development on Github giving an interface for issues.

The Bioconductor core team manages issues on the repositories maintained under the Bioconductor Organization on Github.

The packages maintained by Bioconductor core follow the structure

https://github.com/Bioconductor/<PACKAGE_NAME>/issues

Example: <https://github.com/Bioconductor/GenomicRanges/issues>

16.5 Round up of resources available

Material covered in this workshop very briefly, highlighting

1. Build machines, what commands the build machines run and what information is displayed on the build reports.
2. How to check the HEALTH of your package with badges on the landing pages.
3. Download statistics of packages.
4. RSS feeds and subscribing to them, checking package development across bioconductor with GIT logs.
5. Support sites, mailing lists and where to get specific support as a maintainer.
6. Github and social coding with issues.

There is plenty of infrastructure not covered in this short workshop,

1. Docker containers.
2. AWS - Amazon Machine images(AMIs).
3. BiocCredentials Application for SSH key processing for developers.
4. Git + Bioconductor private git server.
5. Single package builder during package contribution.
6. Github and webhooks for continuous integration.

16.6 Acknowledgements

The Bioconductor core team.

Chapter 17

Bibliography

- Aranda, Bruno, Hagen Blankenburg, Samuel Kerrien, Fiona SL Brinkman, Arnaud Ceol, Emilie Chautard, Jose M Dana, et al. 2011. “PSICQUIC and Psiscore: Accessing and Scoring Molecular Interactions.” *Nature Methods* 8 (7). Nature Publishing Group: 528.
- Bray, Nicolas L, Harold Pimentel, Pál Melsted, and Lior Pachter. 2016. “Near-Optimal RNA-Seq Quantification.” *Nat. Biotechnol.*
- Colaprico, A., T. C. Silva, C. Olsen, L. Garofano, C. Cava, D. Carolini, T. S. Sabedot, et al. 2016. “TCGAbiolinks: An R/Bioconductor Package for Integrative Analysis of Tega Data.” Journal Article. *Nucleic Acids Res* 44 (8): e71. doi:10.1093/nar/gkv1507.
- Collado-Torres, Leonardo, Abhinav Nellore, Kai Kammers, Shannon E Ellis, Margaret A Taub, Kasper D Hansen, Andrew E Jaffe, Ben Langmead, and Jeffrey T Leek. 2017. “Reproducible RNA-seq Analysis Using Recount2.” *Nature Biotechnology* 35 (4): 319–21. doi:10.1038/nbt.3838.
- Davis, Sean R., and Paul S Meltzer. 2007. “GEOquery: A Bridge Between the Gene Expression Omnibus (GEO) and BioConductor.” *Bioinformatics* 23 (14): 1846–7. doi:10.1093/bioinformatics/btm254.
- Demir, Emek, Michael P Cary, Suzanne Paley, Ken Fukuda, Christian Lemer, Imre Vastrik, Guanming Wu, et al. 2010. “The Biopax Community Standard for Pathway Data Sharing.” *Nature Biotechnology* 28 (9). Nature Publishing Group: 935.
- Di Tommaso, Paolo, Maria Chatzou, Evan W Floden, Pablo Prieto Barja, Emilio Palumbo, and Cedric Notredame. 2017. “Nextflow Enables Reproducible Computational Workflows.” *Nature Biotechnology* 35 (4). Nature Research: 316–19.
- Dijk, David van, Juozas Nainys, Roshan Sharma, Pooja Kathail, Ambrose J Carr, Kevin R Moon, Linas Mazutis, Guy Wolf, Smita Krishnaswamy, and Dana Pe’er. 2017. “MAGIC: A diffusion-based imputation method reveals gene-gene interactions in single-cell RNA-sequencing data.” *bioRxiv*. doi:10.1101/111591.
- Fletcher, Russell B, Diya Das, Levi Gadye, Kelly N Street, Ariane Baudhuin, Allon Wagner, Michael B Cole, et al. 2017. “Deconstructing Olfactory Stem Cell Trajectories at Single-Cell Resolution.” *Cell Stem Cell* 20 (6). Elsevier: 817–830.e8. doi:10.1016/j.stem.2017.04.003.
- Ganzfried, Benjamin Frederick, Markus Riester, Benjamin Haibe-Kains, Thomas Risch, Svitlana Tyekucheva, Ina Jazic, Xin Victoria Wang, et al. 2013. “curatedOvarianData: Clinically Annotated Data for the Ovarian Cancer Transcriptome.” *Database: The Journal of Biological Databases and Curation* 2013 (April): bat013. doi:10.1093/database/bat013.
- Green, TRG, and M Petre. 1996. “Usability Analysis of Visual Programming Environments: A ‘Cognitive Dimensions’ Framework.” *Journal of Visual Languages & Computing* 7 (2): 131–74. doi:10.1006/jvlc.1996.0009.
- Grossman, R. L., A. P. Heath, V. Ferretti, H. E. Varmus, D. R. Lowy, W. A. Kibbe, and L. M. Staudt. 2016.

“Toward a Shared Vision for Cancer Genomic Data.” Journal Article. *N Engl J Med* 375 (12): 1109–12. doi:10.1056/NEJMp1607591.

Hardcastle, Thomas, and Krystyna Kelly. 2010. “baySeq: Empirical Bayesian methods for identifying differential expression in sequence count data.” *BMC Bioinformatics* 11 (1): 422+. doi:10.1186/1471-2105-11-422.

Himes, Blanca E., Xiaofeng Jiang, Peter Wagner, Ruoxi Hu, Qiyu Wang, Barbara Klanderman, Reid M. Whitaker, et al. 2014. “RNA-Seq transcriptome profiling identifies CRISPLD2 as a glucocorticoid responsive gene that modulates cytokine function in airway smooth muscle cells.” *PloS One* 9 (6). doi:10.1371/journal.pone.0099625.

International Cancer Genome Consortium, T. J. Hudson, W. Anderson, A. Artez, A. D. Barker, C. Bell, R. R. Bernabe, et al. 2010. “International Network of Cancer Genome Projects.” Journal Article. *Nature* 464 (7291): 993–8. doi:10.1038/nature08987.

Kannan, Lavanya, Marcel Ramos, Angela Re, Nehme El-Hachem, Zhaleh Safikhani, Deena M A Gendoo, Sean Davis, et al. 2016. “Public Data and Open Source Tools for Multi-Assay Genomic Investigation of Disease.” *Briefings in Bioinformatics* 17 (4): 603–15. doi:10.1093/bib/bbv080.

Köster, Johannes, and Sven Rahmann. 2012. “Snakemake—a Scalable Bioinformatics Workflow Engine.” *Bioinformatics* 28 (19). Oxford University Press: 2520–2.

Langfelder, Peter, and Steve Horvath. 2008. “WGCNA: An R Package for Weighted Correlation Network Analysis.” *BMC Bioinformatics* 9 (1). BioMed Central: 559.

Lappalainen, Tuuli, Michael Sammeth, Marc R. Friedländer, Peter A. C. ’t Hoen, Jean Monlong, Manuel A. Rivas, Mar González-Porta, et al. 2013. “Transcriptome and Genome Sequencing Uncovers Functional Variation in Humans.” *Nature*, no. 501: 506–11.

Law, Charity W., Yunshun Chen, Wei Shi, and Gordon K. Smyth. 2014. “Voom: precision weights unlock linear model analysis tools for RNA-seq read counts.” *Genome Biology* 15 (2). BioMed Central Ltd: R29+. doi:10.1186/gb-2014-15-2-r29.

Lawrence, Michael, Wolfgang Huber, Hervé Pagès, Patrick Aboyoun, Marc Carlson, Robert Gentleman, Martin Morgan, and Vincent Carey. 2013. “Software for Computing and Annotating Genomic Ranges.” *PLoS Comput. Biol.* 9. doi:10.1371/journal.pcbi.1003118.

Lee, Stuart, Dianne Cook, and Michael Lawrence. 2018. “Plyranges: A Grammar of Genomic Data Transformation.” *bioRxiv*. Cold Spring Harbor Laboratory. doi:10.1101/327841.

Leng, N., J. A. Dawson, J. A. Thomson, V. Ruotti, A. I. Rissman, B. M. G. Smits, J. D. Haag, M. N. Gould, R. M. Stewart, and C. Kendziorski. 2013. “EBSeq: an empirical Bayes hierarchical model for inference in RNA-seq experiments.” *Bioinformatics* 29 (8). Oxford University Press: 1035–43. doi:10.1093/bioinformatics/btt087.

Li, B., and C. N. Dewey. 2011. “RSEM: Accurate Transcript Quantification from Rna-Seq Data with or Without a Reference Genome.” Journal Article. *BMC Bioinformatics* 12: 323. doi:10.1186/1471-2105-12-323.

Li, Bo, and Colin N. Dewey. 2011. “RSEM: accurate transcript quantification from RNA-Seq data with or without a reference genome.” *BMC Bioinformatics* 12: 323+. doi:10.1186/1471-2105-12-3231.

Love, Michael. 2018. *Airway: RangedSummarizedExperiment for Rna-Seq in Airway Smooth Muscle Cells, by Himes et Al Plos One 2014*.

Love, Michael I., Wolfgang Huber, and Simon Anders. 2014. “Moderated estimation of fold change and dispersion for RNA-seq data with DESeq2.” *Genome Biology* 15 (12). BioMed Central Ltd: 550+. doi:10.1186/s13059-014-0550-8.

Luna, Augustin, Özgün Babur, Bülent Arman Aksoy, Emek Demir, and Chris Sander. 2015. “PaxtoolsR:

- Pathway Analysis in R Using Pathway Commons.” *Bioinformatics* 32 (8). Oxford University Press: 1262–4.
- Morgan, Martin. 2018. *AnnotationHub: Client to Access Annotationhub Resources*.
- Morgan, Xochitl C, and Curtis Huttenhower. 2012. “Chapter 12: Human Microbiome Analysis.” *PLoS Computational Biology* 8 (12): e1002808. doi:10.1371/journal.pcbi.1002808.
- Mostafavi, Sara, Debajyoti Ray, David Warde-Farley, Chris Grouios, and Quaid Morris. 2008. “GeneMANIA: A Real-Time Multiple Association Network Integration Algorithm for Predicting Gene Function.” *Genome Biology* 9 (1). BioMed Central: S4.
- Ono, Keiichiro, Tanja Muetze, Georgi Kolishovski, Paul Shannon, and Barry Demchak. 2015. “CyREST: Turbocharging Cytoscape Access for External Tools via a Restful API.” *F1000Research* 4. Faculty of 1000 Ltd.
- Pasolli, Edoardo, Lucas Schiffer, Paolo Manghi, Audrey Renson, Valerie Obenchain, Duy Tin Truong, Francesco Beghini, et al. 2017. “Accessible, Curated Metagenomic Data Through ExperimentHub.” *Nature Methods* 14 (11). Nature Research: 1023–4. doi:10.1038/nmeth.4468.
- Patro, Rob, Geet Duggal, Michael I Love, Rafael A Irizarry, and Carl Kingsford. 2017. “Salmon Provides Fast and Bias-Aware Quantification of Transcript Expression.” *Nat. Methods* 14 (6~mar): 417–19.
- Patro, Rob, Stephen M. Mount, and Carl Kingsford. 2014. “Sailfish enables alignment-free isoform quantification from RNA-seq reads using lightweight algorithms.” *Nature Biotechnology* 32: 462–64. doi:10.1038/nbt.2862.
- Pierson, Emma, and Christopher Yau. 2015. “ZIFA: Dimensionality reduction for zero-inflated single-cell gene expression analysis.” *Genome Biology* 16 (1): 241. doi:10.1186/s13059-015-0805-z.
- Pratt, Dexter, Jing Chen, Rudolf Pillich, Vladimir Rynkov, Aaron Gary, Barry Demchak, and Trey Ideker. 2017. “NDEX 2.0: A Clearinghouse for Research on Cancer Pathways.” *Cancer Research* 77 (21). AACR: e58–e61.
- Quinlan, Aaron R, and Ira M Hall. 2010. “BEDTools: A Flexible Suite of Utilities for Comparing Genomic Features.” *Bioinformatics* 26 (6): 841–42. doi:10.1093/bioinformatics/btq033.
- Ramos, Marcel, Lucas Schiffer, Angela Re, Rimsha Azhar, Azfar Basunia, Carmen Rodriguez, Tiffany Chan, et al. 2017. “Software for the Integration of Multiomics Experiments in Bioconductor.” *Cancer Research* 77 (21). American Association for Cancer Research: e39–e42. doi:10.1158/0008-5472.CAN-17-0344.
- Reimand, Jüri, Tambet Arak, Priit Adler, Liis Kolberg, Sulev Reisberg, Hedi Peterson, and Jaak Vilo. 2016. “G: Profiler—a Web Server for Functional Interpretation of Gene Lists (2016 Update).” *Nucleic Acids Research* 44 (W1). Oxford University Press: W83–W89.
- Risso, Davide, Fanny Perraudeau, Svetlana Gribkova, Sandrine Dudoit, and Jean-Philippe Vert. 2018a. “A General and Flexible Method for Signal Extraction from Single-Cell RNA-Seq Data.” *Nature Communications* 9 (1): 284. <https://doi.org/10.1038/s41467-017-02554-5>.
- . 2018b. “A General and Flexible Method for Signal Extraction from Single-Cell RNA-Seq Data.” *Nature Communications* 9 (1): 284.
- Roadmap Epigenomics Consortium, Anshul Kundaje, Wouter Meuleman, Jason Ernst, Misha Bilenky, Angela Yen, Alireza Heravi-Moussavi, et al. 2015. “Integrative Analysis of 111 Reference Human Epigenomes.” *Nature* 518 (7539): 317–30. doi:10.1038/nature14248.
- Robert, Christelle, and Mick Watson. 2015. “Errors in RNA-Seq quantification affect genes of relevance to human disease.” *Genome Biology*. doi:10.1186/s13059-015-0734-x.
- Robinson, M. D., D. J. McCarthy, and G. K. Smyth. 2009. “edgeR: a Bioconductor package for differential expression analysis of digital gene expression data.” *Bioinformatics* 26 (1). Oxford University Press: 139–40. doi:10.1093/bioinformatics/btp616.
- Schiffer, Lucas, Rimsha Azhar, Lori Shepherd, Marcel Ramos, Ludwig Geistlinger, Curtis Huttenhower,

- Jennifer B Dowd, Nicola Segata, and Levi Waldron. 2018. “HMP16SDData: Efficient Access to the Human Microbiome Project Through Bioconductor.” *bioRxiv*. doi:10.1101/299115.
- Settle, Brett, David Otasek, John H Morris, and Barry Demchak. 2018. “AMatReader: Importing Adjacency Matrices via Cytoscape Automation.” *F1000Research* 7.
- Shannon, Paul, Andrew Markiel, Owen Ozier, Nitin S Baliga, Jonathan T Wang, Daniel Ramage, Nada Amin, Benno Schwikowski, and Trey Ideker. 2003. “Cytoscape: A Software Environment for Integrated Models of Biomolecular Interaction Networks.” *Genome Research* 13 (11). Cold Spring Harbor Lab: 2498–2504.
- Soneson, Charlotte, Michael I. Love, and Mark Robinson. 2015. “Differential analyses for RNA-seq: transcript-level estimates improve gene-level inferences.” *F1000Research* 4 (1521). doi:10.12688/f1000research.7563.1.
- Stephens, Matthew. 2016. “False Discovery Rates: A New Deal.” *Biostatistics* 18 (2). <https://doi.org/10.1093/biostatistics/kxw041>.
- Street, Kelly, Davide Risso, Russell B Fletcher, Diya Das, John Ngai, Nir Yosef, Elizabeth Purdom, and Sandrine Dudoit. 2018. “Slingshot: Cell lineage and pseudotime inference for single-cell transcriptomics.” *BMC Genomics* 19 (1): 477.
- Szklarczyk, Damian, John H Morris, Helen Cook, Michael Kuhn, Stefan Wyder, Milan Simonovic, Alberto Santos, et al. 2016. “The String Database in 2017: Quality-Controlled Protein–protein Association Networks, Made Broadly Accessible.” *Nucleic Acids Research*. Oxford University Press, gkw937.
- Trapnell, Cole, David G Hendrickson, Martin Sauvageau, Loyal Goff, John L Rinn, and Lior Pachter. 2013. “Differential analysis of gene regulation at transcript resolution with RNA-seq.” *Nature Biotechnology*. doi:10.1038/nbt.2450.
- Tseng, George C., and Wing H. Wong. 2005. “Tight Clustering: A Resampling-Based Approach for Identifying Stable and Tight Patterns in Data.” *Biometrics* 61 (1): 10–16. doi:10.1111/j.0006-341X.2005.031032.x.
- Van den Berge, Koen, Fanny Perraudeau, Charlotte Soneson, Michael I. Love, Davide Risso, Jean-Philippe Vert, Mark D. Robinson, Sandrine Dudoit, and Lieven Clement. 2018. “Observation weights unlock bulk RNA-seq tools for zero inflation and single-cell applications.” *Genome Biology* 19 (1). <https://doi.org/10.1186/s13059-018-1406-4>.
- Verhaak, R. G., P. Tamayo, J. Y. Yang, D. Hubbard, H. Zhang, C. J. Creighton, S. Fereday, et al. 2013. “Prognostically Relevant Gene Signatures of High-Grade Serous Ovarian Carcinoma.” Journal Article. *J Clin Invest* 123 (1): 517–25. doi:10.1172/JCI65833.
- Wang, K., D. Singh, Z. Zeng, S. J. Coleman, Y. Huang, G. L. Savich, X. He, et al. 2010. “MapSplice: Accurate Mapping of Rna-Seq Reads for Splice Junction Discovery.” Journal Article. *Nucleic Acids Res* 38 (18): e178. doi:10.1093/nar/gkq622.
- Wickham, Hadley. 2014. “Tidy Data.” *Journal of Statistical Software, Articles* 59 (10): 1–23. doi:10.18637/jss.v059.i10.
- Wu, Hao, Chi Wang, and Zhijin Wu. 2013. “A new shrinkage estimator for dispersion improves differential expression detection in RNA-seq data.” *Biostatistics* 14 (2). Oxford University Press: 232–43. doi:10.1093/biostatistics/kxs033.
- Zappia, Luke, Belinda Phipson, and Alicia Oshlack. 2017. “Splatter: simulation of single-cell RNA sequencing data.” *Genome Biology* 18 (174). <https://doi.org/10.1186/s13059-017-1305-0>.
- Zhu, Anqi, Joseph G. Ibrahim, and Michael I. Love. 2018. “Heavy-Tailed Prior Distributions for Sequence Count Data: Removing the Noise and Preserving Large Differences.” *bioRxiv*. <https://doi.org/10.1101/303255>.