Orchestrating Single-Cell Analysis with Bioconductor

Robert A. Amezquita
Stephanie C. Hicks

2019-04-12

Contents

Welcome

This is the website for **"Orchestrating Single-Cell Analysis with Bioconductor"**, a book that teaches users some common workflows for the analysis of single-cell RNA-seq data (scRNA-seq). This book will teach you how to make use of cutting-edge Bioconductor tools to process, analyze, visualize, and explore scRNA-seq data. Additionally, it serves as an online companion for the manuscript **"Orchestrating Single-Cell Analysis with Bioconductor"**.

While we focus here on scRNA-seq data, a newer technology that profiles transcriptomes at the single-cell level, many of the tools, conventions, and analysis strategies utilized throughout this book are broadly applicable to other types of assays. By learning the grammar of Bioconductor workflows, we hope to provide you a starting point for the exploration of your own data, whether it be scRNA-seq or otherwise.

This book is organized into two parts. In the *Preamble*, we introduce the book and dive into resources for learning R and Bioconductor (both at a beginner and developer level). Part I ends with a tutorial for a key data infrastructure, the *SingleCellExperiment* class, that is used throughout Bioconductor for single-cell analysis and in the subsequent section. This section can be safely skipped by readers already familiar with R.

The second part, *Workflows*, provides templates for performing single-cell RNA-seq analyses across various objectives. In these templates, we take various datasets from raw data through to preprocessing and finally to the objective at hand, using packages that are referred to in the main manuscript.

The book is written in RMarkdown with bookdown. OSCA is a collaborative effort, supported by various folks from the Bioconductor team who have contributed workflows, fixes, and improvements.

Part I

Preamble

Chapter 1

Introduction

Bioconductor is an open source, open development software project to provide tools for the analysis and comprehension of high-throughput genomic data. It is based primarily on the R programming language.

1.1 What you will learn

The goal of this book is to provide a solid foundation in the usage of Bioconductor tools for single-cell RNA-seq analysis by walking through various steps of typical workflows using example datasets. We strive to tackle key concepts covered in the manuscript, **"Orchestrating Single-Cell Analysis with Bioconductor"**, with each workflow covering these in varying detail, as well as essential preliminaries that are important for following along with the workflows on your own.

1.1.1 Preliminaries

For those unfamiliar with R (and those looking to learn more), we recommend reading the *Learning R and More* chapter, which first and foremost covers how to get started with R. We point to many great online resources for learning R, as well as related tools that are nice to know for bioinformatic analysis. For advanced users, we also point to some extra resources that go beyond the basics. While we provide an extensive list of learning resources for the interested audience in this chapter, we only ask for some familiarity with R before going to the next section.

We then briefly cover getting started with *Using R and Bioconductor*. Bioconductor, being its own repository, has a unique set of tools, documentation, resources, and practices that benefit from some extra explanation.

Data Infrastructure merits a separate chapter. The reason for this is that common data containers are an essential part of Bioconductor workflows because they enable interoperability across packages, allowing for "plug and play" usage of cutting-edge tools. Specifically, here we cover the *SingleCellExperiment* class in depth, as it has become the working standard for Bioconductor based single-cell analysis packages.

Finally, before diving into the various workflows, armed with knowledge about the *SingleCellExperiment* class, we briefly discuss the datasets that will be used throughout the book in *About the Data*.

1.1.2 Workflows

All workflows begin with data import and subsequent *quality control and normalization*, going from a raw (count) expression matrix to a clean one. This includes adjusting for experimental factors and possibly even latent factors. Using the clean expression matrix, *feature selection* strategies can be applied to select the

features (genes) driving heterogeneity. Furthermore, these features can then be used to perform *dimensionality reduction*, which enables downstream analysis that would not otherwise be possible and visualization on 2- or 3-dimensions.

From there, the workflows largely focus on differing downstream analyses. *Clustering* details how to segment a scRNA-seq dataset, and *differential expression* provides a means to determine what drives the differences between different groups of cells. *Integrating datasets* walks through merging scRNA-seq datasets, an area of need as the number of scRNA-seq datasets continues to grow and comparisons between datasets must be done. Finally, we touch upon how to work with *large scale data*, specifically where it becomes impractical or impossible to work with data solely in-memory.

As an added bonus, we dedicate a chapter to *interactive visualization*, which focuses on using the *iSEE* package to enable active exploration of a single cell experiment's data.

1.2 What you won't learn

The field of bioinformatic analysis is large and filled with many potential trajectories depending on the biological system being studied and technology being deployed. Here, we only briefly survey some of the many tools available for the analysis of scRNA-seq, focusing on Bioconductor packages. It is impossible to thoroughly review the plethora of tools available through R and Bioconductor for biological analysis in one book, but we hope to provide the means for further exploration on your own.

Thus, it goes without saying that you may not learn the optimal workflow for your own data from our examples - while we strive to provide high quality templates, they should be treated as just that - a template from which to extend upon for your own analyses.

1.3 Who we wrote this for

We've written this book with the interested experimental biologist in mind, and do our best to make few assumptions on previous programming or statistical experience. Likewise, we also welcome more seasoned bioinformaticians who are looking for a starting point from which to dive into single-cell RNA-seq analysis. As such, we welcome any and all feedback for improving this book to help increase accessibility and refine technical details.

1.4 Why we wrote this

This book was conceived in the fall of 2018, as single-cell RNA-seq analysis continued its rise in prominence in the field of biology. With its rapid growth, and the ongoing developments within Bioconductor tailored specifically for scRNA-seq, it became apparent that an update to the Orchestrating high-throughput genomic analysis with Bioconductor paper was necessary for the age of single-cell studies.

We strive to highlight the fantastic software by people who call Bioconductor home for their tools, and in the process hope to showcase the Bioconductor community at large in continually pushing forward the field of biological analysis.

1.5 Acknowledgements

We would like to thank all Bioconductor contributors for their efforts in creating the definitive leading-edge repository of software for biological analysis. It is truly extraordinary to chart the growth of Bioconductor

over the years. We are thankful for the wonderful community of scientists and developers alike that together make the Bioconductor community special.

We would first and foremost like to thank the Bioconductor core team and the emerging targets subcommittee for commissioning this work, Raphael Gottardo for his continuous mentorship of the writing herein, and all our contributors to the companion manuscript of this book.

We'd also like to thank Garret Grolemund and Hadley Wickham for their book, R for Data Science, from which we drew stylistic and teaching inspiration.

Chapter 2

Learning R and Bioconductor

In this section, we outline various resources for learning R and Bioconductor. We provide a brief set of instructions for installing R on your own machine, and then cover how to get help for functions, packages, and Bioconductor-specific resources for learning more.

2.1 The Benefits of R and Bioconductor

R is a high-level programming language that was initially designed for statistical applications. While there is much to be said about R as a programming language, one of the key advantages of using R is that it is highly extensible through *packages*. Packages are collections of functions, data, and documentation that extend the capabilities of base R. The ease of development and distribution of packages for R has made it a rich environment for many fields of study and application.

One of the primary ways in which packages are distributed is through centralized repositories. The first R repository a user typically runs into is the Comprehensive R Archive Network (CRAN), which hosts over 13,000 packages to date, and is home to many of the most popular R packages.

Similar to CRAN, Bioconductor is a repository of R packages as well. However, whereas CRAN is a general purpose repository, Bioconductor focuses on software tailored for genomic analysis. Furthermore, Bioconductor has stricter requirements for a package to be accepted into the repository. Of particular interest to us is the inclusion of high quality documentation and the use of common data infrastructure to promote package interoperability.

In order to use these packages from CRAN and Bioconductor, and start programming with R to follow along in these workflows, some knowledge of R is helpful. Here we outline resources to guide you through learning the basics.

2.2 Learning R Online

To learn more about programming with R, we highly recommend checking out the online courses offered by Datacamp, which includes both introductory and advanced courses within the R track. Datacamp is all online with many free courses, with videos and a code editor/console that promotes an interactive learning experience. What we like about Datacamp is that it is more focused on topics and programming paradigms that center around data science, which is especially helpful for getting started with R.

Beyond just Datacamp, a mainstay resource for learning R is the R for Data Science book. This book illustrates R programming through the exploration of various data science concepts - transformation, visualization, exploration, and more.

2.3 Running R Locally

While learning R through online resources is a great way to start with R, as it requires minimal knowledge to start up, at some point, it will be desirable to have a local installation - on your own hardware - of R. This will allow you to install and maintain your own software and code, and furthermore allow you to create a personalized workspace.

2.3.1 Installing R

Prior to getting started with this book, some prior programming experience with R is helpful. Check out the *Learning R and More* chapter for a list of resources to get started with R and other useful tools for bioinformatic analysis.

To follow along with the analysis workflows in this book on your personal computer, it is first necessary to install the R programming language. Additionally, we recommend a graphical user interface such as RStudio for programming in R and visualization. RStudio features many helpful tools, such as code completion and an interactive data viewer to name but two. For more details, please see the online book *R for Data Science* prerequisites section for more information about installing R and using RStudio.

2.3.1.1 For MacOS/Linux Users

A special note for MacOS/Linux users: we highly recommend using a package manager to manage your R installation. This differs across different Linux distributions, but for MacOS we highly recommend the Homebrew package manager. Follow the website directions to install homebrew, and install R via the commandline with `brew install R`, and it will automatically configure your installation for you. Upgrading to new R versions can be done by running `brew upgrade`.

2.3.2 Installing R & Bioconductor Packages

After installing R, the next step is to install R packages. In the R console, you can install packages from CRAN via the `install.packages()` function. In order to install Bioconductor packages, we will first need the *BiocManager* package which is hosted on CRAN. This can be done by running:

```
install.packages("BiocManager")
```

The *BiocManager* package makes it easy to install packages from the Bioconductor repository. For example, to install the *SingleCellExperiment* package, we run:

```
## the command below is a one-line shortcut for:
## library(BiocManager)
## install("SingleCellExperiment")
BiocManager::install("SingleCellExperiment")
```

Throughout the book, we can load packages via the `library()` function, which by convention usually comes at the top of scripts to alert readers as to what packages are required. For example, to load the *SingleCellExperiment* package, we run:

```
library(SingleCellExperiment)
```

Many packages will be referenced throughout the book within the workflows, and similar to the above, can be installed using the `BiocManager::install()` function.

2.4 Getting Help In (and Out) of R

One of the most helpful parts of R is being able to get help *inside* of R. For example, to get the manual associated with a function, class, dataset, or package, you can prepend the code of interest with a `?` to retrieve the relevant help page. For example, to get information about the `data.frame()` function, the *SingleCellExperiment* class, the in-built *iris* dataset, or for the *BiocManager* package, you can type:

```
?data.frame
?SingleCellExperiment
?iris
?BiocManager
```

Beyond the R console, there are myriad online resources to get help. The R for Data Science book has a great section dedicated to looking for help outside of R. In particular, Stackoverflow's R tag is a helpful resource for asking and exploring general R programming questions.

2.5 Bioconductor Help

One of the key tenets of Bioconductor software that makes it stand out from CRAN is the required documentation of packages and workflows. In addition, Bioconductor hosts a Bioconductor-specific support site that has grown into a valuable resource of its own, thanks to the work of dedicated volunteers.

2.5.1 Bioconductor Packages

Each package hosted on Bioconductor has a dedicated page with various resources. For an example, looking at the `scater` package page on Bioconductor, we see that it contains:

- a brief description of the package at the top, in addition to the authors, maintainer, and an associated citation
- installation instructions that can be cut and paste into your R console
- documentation - vignettes, reference manual, news

Here, the most important information comes from the documentation section. Every package in Bioconductor is *required* to be submitted with a *vignette* - a document showcasing basic functionality of the package. Typically, these vignettes have a descriptive title that summarizes the main objective of the vignette. These vignettes are a great resource for learning how to operate the essential functionality of the package.

The *reference manual* contains a comprehensive listing of all the functions available in the package. This is a compilation of each function's *manual*, aka help pages, which can be accessed programmatically in the R console via `?<function>`.

Finally, the *NEWS* file contains notes from the authors which highlight changes across different versions of the package. This is a great way of tracking changes, especially functions that are added, removed, or deprecated, in order to keep your scripts current with new versions of dependent packages.

Below this, the *Details* section covers finer nuances of the package, mostly relating to its relationship to other packages:

- upstream dependencies (*Depends*, *Imports*, *Suggests* fields): packages that are imported upon loading the given package
- downstream dependencies (*Depends On Me*, *Imports Me*, *Suggests Me*): packages that import the given package when loaded

For example, we can see that an entry called *simpleSingle* in the *Depends On Me* field on the `scater` page takes us to a step-by-step workflow for low-level analysis of single-cell RNA-seq data.

One additional *Details* entry, the *biocViews*, is helpful for looking at how the authors annotate their package. For example, for the `scater` package, we see that it is associated with `DataImport`, `DimensionReduction`, `GeneExpression`, `RNASeq`, and `SingleCell`, to name but some of its many annotations. We cover *biocViews* in more detail.

2.5.2 biocViews

To find packages via the Bioconductor website, one useful resource is the BiocViews page, which provides a hierarchically organized view of annotations associated with Bioconductor packages.

Under the "Software" label for example (which is comprised of most of the Bioconductor packages), there exist many different views to explore packages. For example, we can inspect based on the associated "Technology", and explore "Sequencing" associated packages, and furthermore subset based on "RNASeq".

Another area of particular interest is the "Workflow" view, which provides Bioconductor packages that illustrate an analytical workflow. For example, the "SingleCellWorkflow" contains the aforementioned tutorial, encapsulated in the *simpleSingleCell* package.

2.5.3 Bioconductor Forums

The Bioconductor support site contains a Stackoverflow-style question and answer support site that is actively contributed to from both users and package developers. Thanks to the work of dedicated volunteers, there are ample questions to explore to learn more about Bioconductor specific workflows.

Another way to connect with the Bioconductor community is through Slack, which hosts various channels dedicated to packages and workflows. The Bioc-community Slack is a great way to stay in the loop on the latest developments happening across Bioconductor, and we recommend exploring the "Channels" section to find topics of interest.

Chapter 3

Beyond R Basics

Here we briefly outline resources for taking your R programming to the next level, including resources for learning about package development. We also outline some companions to R that are good to know not only for package development, but also for running your own bioinformatic pipelines, enabling you to use a broader array of tools to go from raw data to preprocessed data before working in R.

3.1 Becoming an R Expert

For a deeper dive into the finer details of the R programming language, the Advanced R. While targeted at more experienced R users and programmers, this book represents a comprehensive compendium of more advanced concepts, and touches on some of the paradigms used extensively by developers throughout Bioconductor, specifically programming with S4.

Eventually, you'll reach the point where you have your own collection of functions, datasets, and reach the point where you will be writing your own packages. Luckily, there's a guide for just that, with the book R Packages. Packages are great even if just for personal use, and of course, with some polishing, can eventually become a package available on CRAN or Bioconductor. Furthermore, they are also a great way of putting together code associated with a manuscript, promoting reproducible, accessible computing practices, something we all strive for in our work.

For many of the little details that are oft forgotten learning about R, the aptly named What They Forgot to Teach You About R is a great read for learning about the little things such as file naming, maintaining an R installation, and reproducible analysis habits.

Finally, we save the most intriguing resource for last - another book for those on the road to becoming an R expert is R Inferno, which dives into many of the unique quirks of R. *Warning: this book goes* very *deep into the painstaking details of R.*

3.2 Becoming an R/Bioconductor Developer

While learning to use Bioconductor tools is a very welcoming experience, unfortunately there is no central resource for navigating the plethora of gotchas and paradigms associated with developing for Bioconductor. Based on conversations with folks involved in developing for Bioconductor, much of this knowledge is hard won and fairly spread out. This however is beginning to change with more recent efforts led by the Bioconductor team, and while this book represents an earnest effort towards addressing the user perspective, it is currently out of scope to include a deep dive about the developer side.

For those looking to get started with developing packages for Bioconductor, it is important to first become acquainted with developing standalone R packages. To this end, the R Packages book provides a deep dive into the details of constructing your own package, as well as details regarding submission of a package to CRAN. For programming practices,

With that, some resources that are worth looking into to get started are the BiocWorkshops repository under the Bioconductor Github provides a book composed of workshops that have been hosted by Bioconductor team members and contributors. These workshops center around learning, using, and developing for Bioconductor. A host of topics are also available via the Learn module on the Bioconductor website as well. Finally, the Bioconductor developers portal contains a bevy of individual resources and guides for experienced R developers.

3.3 Nice Companions for R

While not essential for our purposes, many bioinformatic tools for processing raw sequencing data require knowledge beyond just R to install, run, and import their results into R for further analysis. The most important of which are basic knowledge of the Shell/Bash utilities, for working with bioinformatic pipelines and troubleshooting (R package) installation issues.

Additionally, for working with packages or software that are still in development and not hosted on an official repository like CRAN or Bioconductor, knowledge of Git - a version control system - and the popular Github repository is helpful. This enables you to not only work with other people's code, but also better manage your own code to keep track of changes.

3.3.1 Shell/Bash

Datacamp and other interactive online resources such as Codecademy are great places to learn some of these extra skills. We highly recommend learning Shell/Bash, as it is the starting point for most bioinformatic processing pipelines.

3.3.2 Git

We would recommend learning Git next, a system for code versioning control which underlies the popular Github repository, where many of the most popular open source tools are hosted. Learning Git is essential for not only keeping track of your own code, but also for using, managing, and contributing to open source software projects.

For a more R centric look at using Git (and Github), we highly recommend checking out Happy Git and Github for the useR.

3.3.3 Other Languages

A frequent question that comes up is "What else should I learn besides R?" Firstly, we believe that honing your R skills is first and foremost, and beyond just R, learning Shell/Bash and Git covered in the *Nice Friends for R* section are already a great start. For those just getting started, these skills should become comfortable in practice before moving on.

However, there are indeed benefits to going beyond just R. At a basic level, learning other programming languages helps broaden one's perspective - similar to learning multiple spoken or written languages, learning about other programming languages (even if only in a cursory manner) helps one identify broader patterns that may be applicable across languages.

At an applied level, work within and outside of R has made it ever more friendly now than ever before with multi-lingual setups and teams, enabling the use of the best tool for the job at hand. For example, Python is another popular language used in both data science and a broader array of applications as well. R now supports a native Python interface via the *reticulate* package, enabling access to tools developed originally in Python such as the popular TensorFlow framework for machine learning applications. C++ is frequently used natively in R as well via Rcpp in packages to massively accelerate computations. Finally, multiple langauges are supported in code documents and reports through R Markdown.

Chapter 4

Data Infrastructure

One of the advantages of using Bioconductor packages is that they utilize common data infrastructures which makes analyses interoperable across various packages. Furthermore, much engineering effort is put into making this infrastructure robust and scalable. Here, we describe the *SingleCellExperiment* object (or `sce` in shorthand) in detail to describe how it is constructed, utilized in downstream analysis, and how it stores various types of primary data and metadata.

4.1 Prerequisites

The Bioconductor package `SingleCellExperiment` provides the `SingleCellExperiment` class for usage. While the package is implicitly installed and loaded when using any package that depends on the `sce` class, it can be explicitly installed (and loaded) as follows:

```
BiocManager::install('SingleCellExperiment')
```

Additionally, we use some functions from the `scater` and `scran` packages, as well as the CRAN package `uwot` (which conveniently can also be installed through `BiocManager`). These functions will be accessed through the `<package>::<function>` convention as needed.

```
BiocManager::install(c('scater', 'scran', 'uwot'))
```

For this session, all we will need loaded is the `SingleCellExperiment` package:

```
library(SingleCellExperiment)
```

4.2 The Essentials of `sce`

4.2.1 Primary Data: The `assays` Slot

The `SingleCellExperiment` (`sce`) object is the basis of single-cell analytical applications based in Bioconductor. The `sce` object is an S4 object, which in essence provides a more formalized approach towards construction and accession of data compared to other methods available in R. The utility of S4 comes from validity checks that ensure that safe data manipulation, and most important for our discussion, from its extensibility through *slots*.

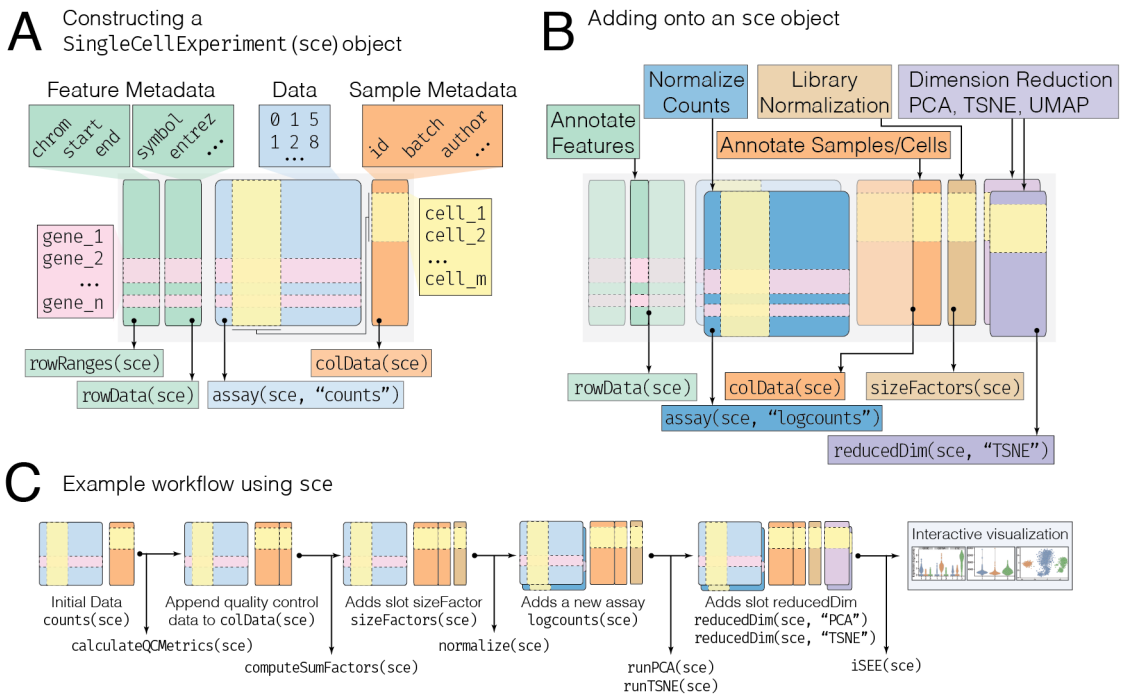


Figure 4.1: Figure 1. Overview of the SingleCellExperiment object

If we imagine the `sce` object to be a ship, the *slots* of `sce` can be thought of as individual cargo boxes - each exists as a separate entity *within* the `sce` object. Furthermore, each slot contains data that arrives in its own format. To extend the metaphor, we can imagine that different variations of cargo boxes are required for fruits versus bricks. In the case of `sce`, certain slots expect numeric matrices, whereas others may expect data frames.

To construct a rudimentary `sce` object, all we need is a single slot:

- `assays` slot: contains primary data such as counts in list, where each entry of the list is in a matrix format, where rows correspond to features (genes) and columns correspond to samples (cells) (*Figure 1A, blue box*)

Let's start simple by generating three cells worth of count data across ten genes.

```
counts_matrix <- data.frame(cell_1 = rpois(10, 10),
                            cell_2 = rpois(10, 10),
                            cell_3 = rpois(10, 30))
rownames(counts_matrix) <- paste0("gene_", 1:10)
counts_matrix <- as.matrix(counts_matrix) # must be a matrix object!
```

From this, we can now construct our first `SingleCellExperiment` object, using the defined *constructor*, `SingleCellExperiment()`. Note that we provide our data as a *named list*, and each entry of the list is a *matrix*. Here, we name the `counts_matrix` entry as simply `counts` within the list.

```
sce <- SingleCellExperiment(assays = list(counts = counts_matrix))
```

To inspect the object, we can simply type `sce` into the console to see some pertinent information, which will display an overview of the various slots available to us (which may or may not have any data).

```
sce
```

```
## class: SingleCellExperiment
## dim: 10 3
## metadata(0):
## assays(1): counts
## rownames(10): gene_1 gene_2 ... gene_9 gene_10
## rowData names(0):
## colnames(3): cell_1 cell_2 cell_3
## colData names(0):
## reducedDimNames(0):
## spikeNames(0):
```

To access the count data we just supplied, we can do any one of the following:

- `assay(sce, "counts")` - this is the most general method, where we can supply the name of the assay as the second argument.
- `counts(sce)` - this is the same as the above, but *only* works for assays with the special name `"counts"`.

```
counts(sce)
```

```
##         cell_1 cell_2 cell_3
## gene_1      11     10     25
## gene_2      18      9     32
## gene_3       7     16     31
## gene_4       6     16     33
## gene_5       9     11     38
## gene_6       6      7     28
## gene_7       7      7     22
## gene_8       8      8     33
## gene_9      10     12     39
## gene_10     10     12     25
```

```
## assay(sce, "counts") ## same as above in this special case
```

4.2.2 Extending the assays Slot

What makes the `assay` slot especially powerful is that it can hold *multiple* representations of the primary data. This is especially useful for storing a *normalized* version of the data. We can do just that as shown below, using the `scran` and `scater` packages to compute a log-count normalized representation of the initial primary data.

Note that here, we overwrite our previous `sce` upon reassigning the results to `sce` - this is because these functions *return* a `SingleCellExperiment` object. Some functions - especially those outside of single-cell oriented Bioconductor packages - do not, in which case you will need to append your results to the `sce` object (see below).

```
sce <- scran::computeSumFactors(sce)
sce <- scater::normalize(sce)
```

Viewing the object again, we see that these functions added some new entries:

```
sce
```

```
## class: SingleCellExperiment
## dim: 10 3
## metadata(1): log.exprs.offset
## assays(2): counts logcounts
## rownames(10): gene_1 gene_2 ... gene_9 gene_10
## rowData names(0):
## colnames(3): cell_1 cell_2 cell_3
## colData names(0):
## reducedDimNames(0):
## spikeNames(0):
```

Specifically, we see that the `assays` slot has grown to be comprised of two entries: `counts` (our initial data) and `logcounts` (the normalized data). Similar to `counts`, the `logcounts` name is a special name which lets us access it simply by typing `logcounts(sce)`, although the longhand version works just as well.

```
logcounts(sce)
```

```
##         cell_1 cell_2 cell_3
## gene_1    4.40   4.05   3.89
## gene_2    5.09   3.91   4.22
## gene_3    3.79   4.70   4.18
## gene_4    3.58   4.70   4.26
## gene_5    4.13   4.18   4.46
## gene_6    3.58   3.58   4.04
## gene_7    3.79   3.58   3.71
## gene_8    3.97   3.75   4.26
## gene_9    4.27   4.30   4.49
## gene_10   4.27   4.30   3.89
```

```
## assay(sce, "logcounts") ## same as above
```

Notice that the data before had a severe discrepancy in counts between cells 1/2 versus 3, and that normalization has ameliorated this difference.

To look at all the available assays within `sce`, we can type:

```
assays(sce)
```

```
## List of length 2
## names(2): counts logcounts
```

While the functions above demonstrate automatic addition of assays to our `sce` object, there may be cases where we want to perform our own calculations and save the result into the `assays` slot. In particular, this is important for using functions that do *not* return your `SingleCellExperiment` object.

Let's append a new version of the data that has been offset by `+100`.

```
counts_100 <- assay(sce, "counts") + 100
assay(sce, "counts_100") <- counts_100 # assign a new entry to assays slot
```

Then we can use the accessor `assays()` (notice this is plural!) to see all our entries into the `assay` slot that we have made so far. Note that to see the *names* of all the assays, we use the plural `assays()` accessor, and to retrieve a single assay entry (as a matrix) we use the singular `assay()` accessor, providing the name of the assay we wish to retrieve as above.

```
assays(sce)
```

```
## List of length 3
## names(3): counts logcounts counts_100
```

These entries are also seen on the default view of `sce`:

```
sce
```

```
## class: SingleCellExperiment
## dim: 10 3
## metadata(1): log.exprs.offset
## assays(3): counts logcounts counts_100
## rownames(10): gene_1 gene_2 ... gene_9 gene_10
## rowData names(0):
## colnames(3): cell_1 cell_2 cell_3
## colData names(0):
## reducedDimNames(0):
## spikeNames(0):
```

This sort of extension of the `assays` slot is represented graphically in *Figure 1B (dark blue box)*, showing the addition of the `logcounts` matrix into the `assays` slot.

In a similar manner, many of the slots of `sce` are extendable through assignment as shown above, thus allowing for myriad custom functionality as needed for interoperability with functions outside of single-cell oriented Bioconductor packages.

4.2.3 Column (Meta)Data: `colData` Slot

To further annotate our `sce` object, one of the first and most useful pieces of information is adding on metadata that describes the columns of our primary data, e.g. describing the samples or cells of our experiment. This data is entered into the `colData` slot:

- `colData` slot: metadata that describes that samples (cells) provided as a `data.frame` or (`DataFrame` if appending), where rows correspond to cells, and columns correspond to the sample (cells) metadata features (e.g. id, batch, author, etc.) (*Figure 1A, orange box*).

So, let's come up with some metadata for the cells, starting with a batch variable, where cells 1 and 2 are in batch 1, and cell 3 is from batch 2.

```
cell_metadata <- data.frame(batch = c(1, 1, 2))
rownames(cell_metadata) <- paste0("cell_", 1:3)
```

Now, we can take two approaches - either append the `cell_metadata` to our existing `sce`, or start from scratch via the `SingleCellExperiment()` constructor and provide it from the get go. We'll start from scratch for now, but will also show how to append the data as well:

```
## From scratch:
sce <- SingleCellExperiment(assays = list(counts = counts_matrix),
                            colData = cell_metadata)

## Appending to existing object (requires DataFrame() coercion)
## colData(sce) <- DataFrame(cell_metadata)
```

Similar to `assays`, we can see our colData is now populated from the default view of `sce`:

```
sce
```

```
## class: SingleCellExperiment
## dim: 10 3
## metadata(0):
## assays(1): counts
## rownames(10): gene_1 gene_2 ... gene_9 gene_10
## rowData names(0):
## colnames(3): cell_1 cell_2 cell_3
## colData names(1): batch
## reducedDimNames(0):
## spikeNames(0):
```

And furthermore access our column (meta)data with the accessor, `colData()`:

```
colData(sce)
```

```
## DataFrame with 3 rows and 1 column
##            batch
##        <numeric>
## cell_1         1
## cell_2         1
## cell_3         2
```

Finally, some packages automatically add to the `colData` slot, for example, the `scater` package features a function, `calculateQCMetrics()`, which appends a lot of quality control data. Here we show the first five columns of `colData(sce)` with the quality control metrics appended to it.

```
sce <- scater::calculateQCMetrics(sce)
colData(sce)[, 1:5]
```

```
## DataFrame with 3 rows and 5 columns
##            batch is_cell_control total_features_by_counts
##        <numeric>       <logical>                 <integer>
## cell_1         1           FALSE                        10
## cell_2         1           FALSE                        10
## cell_3         2           FALSE                        10
##        log10_total_features_by_counts total_counts
##                             <numeric>    <integer>
## cell_1               1.04139268515823           92
## cell_2               1.04139268515823          108
## cell_3               1.04139268515823          306
```

4.2.3.1 Using colData for Subsetting

A common operation with `colData` is its use in subsetting. One simple way to access `colData` is through the use of the `$` operator, which is a shortcut for accessing a variable within the `colData` slot:

```
sce$batch
```

```
## [1] 1 1 2
```

```
## colData(sce)$batch # same as above
```

If we only wanted cells within batch 1, we could subset our sce object as follows (remember, we subset on the *columns* in this case because we are filtering by cells/samples here).

```
sce[, sce$batch == 1]
```

```
## class: SingleCellExperiment
## dim: 10 2
## metadata(0):
## assays(1): counts
## rownames(10): gene_1 gene_2 ... gene_9 gene_10
## rowData names(7): is_feature_control mean_counts ... total_counts
##   log10_total_counts
## colnames(2): cell_1 cell_2
## colData names(10): batch is_cell_control ...
##   pct_counts_in_top_200_features pct_counts_in_top_500_features
## reducedDimNames(0):
## spikeNames(0):
```

4.2.4 Feature Metadata: rowData/rowRanges

Lastly, the rows also have their own metadata slot to store information that pertains to the features of the `sce` object:

- `rowData` slot: contains data in a `data.frame` (`DataFrame`) format that describes aspects of the data corresponding to the rows of the primary data (*Figure 1A, green box*).

Furthermore, there is a special slot which pertains to features with genomic coordinates:

- `rowRanges` slot: contains data in a `GRangesList` (where each entry is a `GenomicRanges` format) that describes the chromosome, start, and end coordinates of the features (genes, genomic regions).

Both of these can be accessed via their respective accessors, `rowRanges()` and `rowData()`. In our case, `rowRanges(sce)` produces an empty list:

```
rowRanges(sce) # empty
```

```
## GRangesList object of length 10:
## $gene_1
## GRanges object with 0 ranges and 0 metadata columns:
```

```
##    seqnames    ranges strand
##       <Rle> <IRanges>  <Rle>
##
## $gene_2
## GRanges object with 0 ranges and 0 metadata columns:
##      seqnames ranges strand
##
## $gene_3
## GRanges object with 0 ranges and 0 metadata columns:
##      seqnames ranges strand
##
## ...
## <7 more elements>
## -------
## seqinfo: no sequences
```

However, our call to `calculateQCMetrics(sce)` in the prior section filled in the `rowData` slot of our `sce` object, as we can see below (only the first three columns are shown for brevity):

```
rowData(sce)[, 1:3]
```

```
## DataFrame with 10 rows and 3 columns
##         is_feature_control      mean_counts log10_mean_counts
##                  <logical>        <numeric>         <numeric>
## gene_1               FALSE 15.3333333333333  1.21307482530885
## gene_2               FALSE 19.6666666666667  1.31527043477859
## gene_3               FALSE               18  1.27875360095283
## gene_4               FALSE 18.3333333333333  1.28630673884327
## gene_5               FALSE 19.3333333333333   1.3082085802911
## gene_6               FALSE 13.6666666666667  1.16633142176653
## gene_7               FALSE               12  1.11394335230684
## gene_8               FALSE 16.3333333333333  1.23888208891514
## gene_9               FALSE 20.3333333333333  1.32905871926422
## gene_10              FALSE 15.6666666666667  1.22184874961636
```

In a similar fashion to the `colData` slot, such feature metadata could be provided at the onset when creating the `SingleCellExperiment` object, which we leave up to the reader as an exercise.

4.2.4.1 Subsetting with on Rows

To subset an `sce` object down at the feature/gene level, we can do a row subsetting operation similar to other R objects, by supplying either numeric indices or a vector of names:

```
sce[c("gene_1", "gene_4"), ]
```

```
## class: SingleCellExperiment
## dim: 2 3
## metadata(0):
## assays(1): counts
## rownames(2): gene_1 gene_4
## rowData names(7): is_feature_control mean_counts ... total_counts
##   log10_total_counts
```

```
## colnames(3): cell_1 cell_2 cell_3
## colData names(10): batch is_cell_control ...
##   pct_counts_in_top_200_features pct_counts_in_top_500_features
## reducedDimNames(0):
## spikeNames(0):
```

```
## sce[c(1, 4), ] # same as above in this case
```

4.2.5 Size Factors Slot: `sizeFactors`

Briefly, we already encountered this via the `scran::computeSumFactors(sce)` call, which adds a `sizeFactors` slot:

- `sizeFactors` slot: contains information in a numeric vector regarding the sample/cell normalization factors used to produce a normalize data representation (*Figure 1B, brown box*)

```
sce <- scran::computeSumFactors(sce)
sce <- scater::normalize(sce)
sizeFactors(sce)
```

```
## [1] 0.545 0.640 1.814
```

4.3 A Brief Recap: From `se` to `sce`

So far, we have covered the `assays` (primary data), `colData` (sample metadata), `rowData`/`rowRanges` (feature metadata), and `sizeFactors` slots of `SingleCellExperiment`.

What is important to note is that the `SingleCellExperiment` class *derives* from the `SummarizedExperiment` (`se`) class, its predecessor, and in particular inherits the aforementioned slots. As such, much of the `SummarizedExperiment` functionality is retained in `SingleCellExperiment`. This allows existing methods that work with `SummarizedExperiment` to work similarly on `SingleCellExperiment` objects.

So what's new about the `SingleCellExperiment` class then? For our discussion, the most important change is the addition of a new slot called `reducedDims`.

4.4 The `reducedDims` Slot

The `reducedDims` slot is a new addition which is specially designed to store the reduced dimensionality representations of primary data, such as PCA, tSNE, UMAP, and others.

- `reducedDims` slot: contains a list of numeric `matrix` entries which describe dimensionality reduced representations of the primary data, such that rows represent the columns of the primary data (aka the samples/cells), and columns represent the dimensions

Most importantly, just like the `assays` slot, the `reducedDims` slot can hold a list of many entries. So, it can hold a PCA, TSNE, and UMAP representation of a given dataset all within the `reducedDims` slot.

In our example, we can calculate a PCA representation of our data as follows using the `scater` package function `runPCA()`. We see that the `sce` now shows a new `reducedDim` and that the accessor `reducedDim()` produces the results of running PCA on the normalized data from `logcounts(sce)`.

```
sce <- scater::runPCA(sce)
reducedDim(sce, "PCA")
```

```
##           PC1    PC2
## cell_1 -2.160  1.486
## cell_2 -0.383 -2.388
## cell_3  2.543  0.902
## attr(,"percentVar")
## [1] 0.564 0.436
```

From this, we can also calculate a tSNE representation using the `scater` package function `runTSNE()`, and see that it can be seen both in the default view of `sce` and via accession:

```
sce <- scater::runTSNE(sce, perplexity = 0.1)
```

```
## Perplexity should be lower than K!
```

```
reducedDim(sce, "TSNE")
```

```
##         [,1]  [,2]
## cell_1  1991  5334
## cell_2  3619 -4404
## cell_3 -5610  -929
```

We can view the names of all our entries in the `reducedDims` slot via the accessor, `reducedDims()` (notice that this is plural, and thus not the same as `reducedDim()`:

```
reducedDims(sce)
```

```
## List of length 2
## names(2): PCA TSNE
```

Now, say we have a different dimensionality reduction approach which has not yet been implemented with `SingleCellExperiment` objects in mind. For example, let's say we want to try the `umap()` function as implemented in the `uwot` package (which is a much faster version of the default `umap` implementation currently in `scater`).

Similar to how we extended the `assays` slot with our own custom entry of `counts_100`, we can do similarly for the `reducedDims` slot:

```
u <- uwot::umap(t(logcounts(sce)), n_neighbors = 2)
reducedDim(sce, "UMAP_uwot") <- u

reducedDim(sce, "UMAP_uwot")
```

```
##           [,1]   [,2]
## cell_1  0.642  0.266
## cell_2 -0.381 -0.552
## cell_3 -0.260  0.286
## attr(,"scaled:center")
## [1]  4.8 14.7
```

And we can also see its entry when we look at the `reducedDims()` accessor output:

```
reducedDims(sce)
```

```
## List of length 3
## names(3): PCA TSNE UMAP_uwot
```

4.5 One More Thing: `metadata` Slot

Some analyses produce results that do not fit into the aforementioned slots. Thankfully, there is a slot just for this type of messy data, and in fact, can accommodate any type of data, so long as it is in a named list:

- `metadata` slot: a named list of entries, where each entry in the list can be anything you want it to be

For example, say we have some favorite genes, such as highly variable genes, we want to save inside of `sce` for use in our analysis at a later point. We can do this simply by appending to the metadata slot as follows:

```
my_genes <- c("gene_1", "gene_5")
metadata(sce) <- list(favorite_genes = my_genes)
metadata(sce)
```

```
## $favorite_genes
## [1] "gene_1" "gene_5"
```

Similarly, we can append more information via the `$` operator:

```
your_genes <- c("gene_4", "gene_8")
metadata(sce)$your_genes <- your_genes
metadata(sce)
```

```
## $favorite_genes
## [1] "gene_1" "gene_5"
##
## $your_genes
## [1] "gene_4" "gene_8"
```

4.6 About Spike-Ins

You might have noticed that the `sce` default view produces an entry with `spikeNames`. The `SingleCellExperiment` object contains some special considerations for experiments with spike-in (ERCC) controls. We leave this to the interested reader to learn more about in the `SingleCellExperiment` introductory vignette.

4.7 Working with *SingleCellExperiment*

Figure 1C shows an example workflow that uses the `SingleCellExperiment` object as its base, and similar to our walkthrough of the `sce` class above, continually appends new entries to save the results of the analysis. In the workflows that follow in this book, we will be similarly appending to an initial `sce` object many of our analytical results.

Part II

Workflows

Chapter 5

About the Data

5.1 10X Genomics PBMC Data

Bioconductor versions, PBMC4k/PBMC3k.

5.2 Human Cell Atlas

Bioconductor access.

```
d = c('a', 'b')
```

Chapter 6

Large-scale Data

author: Davide Risso, Robert A. Amezquita

This workflow illustrates the latest Bioconductor infrastructure to analyze large single-cell datasets that may not entirely fit in memory. We focus on the most common application in exploratory single-cell analysis, namely to find subpopulations of cells. The proposed workflow consists of the following steps:

1. Normalization
2. Dimensionality reduction
3. Clustering

We will exploit a number of Bioconductor packages able to interact with the HDF5 on-disk data representation, freeing us of the need to load the full dataset in memory.

This workflow was specially designed to handle large datasets such as the 1.3 million cell Human Cell Atlas in the `HCAData` scRNA-seq dataset. However, in our workflow, we will subsample the `HCAData` object down to a more manageable number solely for the sake of (compilation) time.

6.1 Package Requirements

These packages will be required for working through the vignette, and can be installed by running the code below:

```
data_pkg <- c("HCAData", "ExperimentHub")
calc_pkg <- c("scater", "scran", "mbkmeans", "BiocSingular", "uwot")
visl_pkg <- c("RColorBrewer", "pheatmap", "ggplot2")
infr_pkg <- c("DelayedMatrixStats", "pryr", "BiocParallel")

BiocManager::install(c(data_pkg, calc_pkg, visl_pkg, infr_pkg))
```

```
library(HCAData)
library(ExperimentHub)
library(scater)
library(scran)
library(uwot)
library(BiocSingular)
library(mbkmeans)
```

```
library(RColorBrewer)
library(pheatmap)
library(ggplot2)
library(DelayedMatrixStats)
library(pryr)
library(BiocParallel)
```

6.2 Interacting with HDF5 files

At a low-level, the main interface between HDF5 and Bioconductor is implemented in the packages `rhdf5`, which provides read/write functionalities, `Rhdf5lib`, which provides C and C++ HDF5 libraries, and `beachmat`, which provides a consistent C++ class interface for a variety of commonly used matrix types, including sparse and HDF5-backed matrices.

These packages are useful for developers that want to develop methods able to interact with HDF5 data sets. However, for most Bioconductor users interested in the analysis of single-cell data, the entry point is represented by the high-level class `SingleCellExperiment` (implemented in the `SingleCellExperiment` package) and the lower level classes `HDF5Matrix` and `DelayedMatrix`, which can be stored in the `assay` slot of a `SingleCellExperiment` object. Once the data are stored in a `SingleCellExperiment` object with `HDF5Matrix` or `DelayedMatrix` as its assay, the packages `scater`, `scran`, `BiocSingular` and `mbkmeans` can be seamlessly used. The package `DelayedMatrixStats` deserves a special mention: it implements the rich API of the CRAN package `matrixStats` for `HDF5Matrix` and `DelayedMatrix` objects.

We invite the reader to find more details on all the mentioned packages in their relative vignettes. In the remainder of this use case, we will use these methods to find cell sub-populations in a real datasets.

6.3 Loading the Data

Here, we use one of the Human Cell Atlas preview datasets available in the `HCAData` Bioconductor package, the `ica_bone_marrow` dataset.

```
eh <- ExperimentHub()
query(eh, "HCAData")
##change to brain
sce <- HCAData("ica_bone_marrow")
```

In order to make the compilation time more manageable, here we will subsample the data to a more reasonable size. For our use, we will go down to 5000 cells.

```
set.seed(1234)
subsample <- sample(ncol(sce), 5000) # super downsample temporary
sce <- sce[, subsample]
```

One small bit of housekeeping - the initial gene IDs used as `rownames` for `sce` are in Ensembl Gene ID format. To make the gene ids human-readable, we convert the rownames to gene symbol. In this case, the mapping between Ensembl Gene ID's and symbols is kept in the `rowData` slot of our `sce`, and so we overwrite the current rownames as follows:

```
rownames(sce) <- rowData(sce)$Symbol
```

We can inspect the resulting `sce` object's key characteristics:

```
sce[1:5, 1:5] # first 5x5 entries
```

```
## class: SingleCellExperiment
## dim: 5 5
## metadata(7): high_mito genes_keep ... wcss clusters_final
## assays(2): counts logcounts
## rownames(5): RP11-34P13.3 FAM138A OR4F5 RP11-34P13.7 RP11-34P13.8
## rowData names(3): ID Symbol scater_qc
## colnames(5): MantonBM1_HiSeq_8-AGGTCCGGTACCAGTT-1
##   MantonBM5_HiSeq_8-ATCTACTAGCTCCTTC-1
##   MantonBM5_HiSeq_7-CGATTGAGTCGAAAGC-1
##   MantonBM5_HiSeq_8-CAGAGAGAGTATTGGA-1
##   MantonBM7_HiSeq_8-CACAAACCAGCTGTTA-1
## colData names(4): Barcode scater_qc clusters_prenorm
##   clusters_final
## reducedDimNames(3): PCA TSNE UMAP
## spikeNames(0):
```

As well as the `assay` slot of `sce`, which we see below that the data is indeed stored in an object of the `DelayedMatrix` class, and that the Ensembl gene ID's have been replaced with gene symbols:

```
assay(sce)[1:5, 1:5] # first 5x5 entries
```

```
## <5 x 5> DelayedMatrix object of type "integer":
##              MantonBM1_HiSeq_8-AGGTCCGGTACCAGTT-1 ...
## RP11-34P13.3                                    0   .
## FAM138A                                         0   .
## OR4F5                                           0   .
## RP11-34P13.7                                    0   .
## RP11-34P13.8                                    0   .
##              MantonBM7_HiSeq_8-CACAAACCAGCTGTTA-1
## RP11-34P13.3                                    0
## FAM138A                                         0
## OR4F5                                           0
## RP11-34P13.7                                    0
## RP11-34P13.8                                    0
```

6.4 Preprocessing

First, we use the `scater` package to compute a set of QC measures and filter out the low-quality samples.

```
ctrl <- list(Mito = grep("^MT", rowData(sce)$Symbol))

sce <- calculateQCMetrics(sce,
                          feature_controls = ctrl,
                          compact = TRUE, # return as nested DF
                          BPPARAM = MulticoreParam(2))
```

6.4.1 Remove Damaged Cells

Here we calculate which cells have a high proportion of mitocondrial reads, using it as a proxy for cell damage, and save the result into the `metadata` slot.

```
high_mito <- isOutlier(colData(sce)$scater_qc$feature_control_Mito$pct_counts,
                       nmads = 3, type = "higher")
metadata(sce)$high_mito <- high_mito
```

The table below enumerates the cells which fail/pass this filter:

```
table(metadata(sce)$high_mito)
```

```
##
## FALSE  TRUE
##  4496   504
```

We can then filter cells in our `sce` object on this basis.

```
sce <- sce[, !metadata(sce)$high_mito]
```

6.4.2 Determine Lowly Expressed Genes

Before proceding with the data analysis, we remove the lowly expressed genes. Here, we keep only those genes that have at least 1 UMI in at least 5% of the data. These threshold are dataset-specific and may need to be taylored to specific applications.

```
num_reads <- 1
num_cells <- 0.05 * ncol(sce)
keep <- which(DelayedArray::rowSums(counts(sce) >= num_reads) >= num_cells)

metadata(sce)$genes_keep <- keep
```

We will use the `genes_keep` vector to subset our genespace in upcoming calculations.

6.5 Normalization

Normalization is a crucial step in the preprocessing of the results. Here, we use the `scran` package to compute size factors that we will use to compute the normalized log-expression values.

It has been shown that the `scran` method works best if the size factors are computed within roughly homogeneous cell populations; hence, it is beneficial to run a quick clustering on the raw data to compute better size factors. This ensures that we do not pool cells that are very different. Note that this is not the final clustering to identify cell sub-populations.

Here we use `mbkmeans` to perform an initial clustering based on the count data in the `sce` object. Note that given that this operates on counts rather than the principal components, and thus will take a longer amount of time to compute. A more thorough explanation of `mbkmeans` and its parameters will be covered in a subsequent section.

Note that we use the `set.seed()` function to ensure the reproducibility of the results.

```
set.seed(1234)

## Subset down based on kept genes; use only 1000 genes
## to speed up prenorm clustering
subsample <- sample(length(metadata(sce)$genes_keep), 1000)
genes_1k <- metadata(sce)$genes_keep[subsample]
counts_genes_1k <- counts(sce)[genes_1k, ]

## Cluster based on 1k genes x counts data
clusters_prenorm <- mbkmeans(counts_genes_1k,
                     max_iters = 1,   # reduced for faster clustering
                     clusters = 10,   # guesstimate
                     batch_size = 50)

## Save results into colData slot
colData(sce)$clusters_prenorm <- clusters_prenorm$Clusters

## Compute size factors w.r.t. prenorm clusters
sce <- computeSumFactors(sce,
                         min.mean = 0.1,
                         cluster = sce$clusters_prenorm,
                         BPPARAM = MulticoreParam(2))
```

Finally, we compute normalized log-expression values with the `normalize()` function from the `scater` package.

```
sce <- normalize(sce)
```

Note that the log-normalized data are stored in the `logcounts` assay of the object. Since the `counts` assay is a `DelayedMatrix` and we have only one set of size factors in the object, the normalized data are also stored as a `DelayedMatrix`.

```
## Verifying that our logcounts are also of DelayedMatrix class
logcounts(sce)
```

```
## <33694 x 4496> DelayedMatrix object of type "double":
##              MantonBM1_HiSeq_8-AGGTCCGGTACCAGTT-1 ...
## RP11-34P13.3                                    0   .
##      FAM138A                                    0   .
##        OR4F5                                    0   .
## RP11-34P13.7                                    0   .
## RP11-34P13.8                                    0   .
##          ...                                    .   .
##   AC233755.2                                    0   .
##   AC233755.1                                    0   .
##   AC240274.1                                    0   .
##   AC213203.1                                    0   .
##      FAM231B                                    0   .
##              MantonBM7_HiSeq_1-GTCTTCGTCTGTCCGT-1
## RP11-34P13.3                                    0
##      FAM138A                                    0
##        OR4F5                                    0
```

```
## RP11-34P13.7                                0
## RP11-34P13.8                                0
##          ...                                .
##   AC233755.2                                0
##   AC233755.1                                0
##   AC240274.1                                0
##   AC213203.1                                0
##      FAM231B                                0
```

This allows us to store in memory only the `colData` and `rowData`, resulting in a fairly small object. We can inspect the size of the object using the `pryr` package `object_size()` function. We leave it to the interested reader to compare the results to data that are fully loaded in-memory.

```
pryr::object_size(sce)
```

```
## 15.8 MB
```

6.6 Dimensionality reduction

Here, we perform dimensionality reduction by first identifying the top 1000 most variable genes and then running PCA on the normalized log counts.

```
## Find most variable genes based on logcounts
vars_genes <- DelayedMatrixStats::rowVars(logcounts(sce))
names(vars_genes) <- rownames(sce)
vars_genes <- sort(vars_genes, decreasing = TRUE)

## save our genes variances into the metadata slot for safekeeping
metadata(sce)$vars_genes <- vars_genes

## Specify the top 1000 most highly variable genes (hvg) by name
metadata(sce)$hvg_genes <- names(metadata(sce)$vars_genes)[1:1000]
```

```
## transpose and subset by hvg the logcounts
for_pca <- t(logcounts(sce)[metadata(sce)$hvg_genes, ])

## Run PCA with parallelization + random svd (via rsvd())
pca <- BiocSingular::runPCA(for_pca, rank = 20,
                            BSPARAM = RandomParam(deferred = FALSE),
                            BPPARAM = MulticoreParam(2))

## Save PCA result into the reducedDims slot
reducedDim(sce, "PCA") <- pca$x
```

6.7 Clustering with Mini-batch k-means

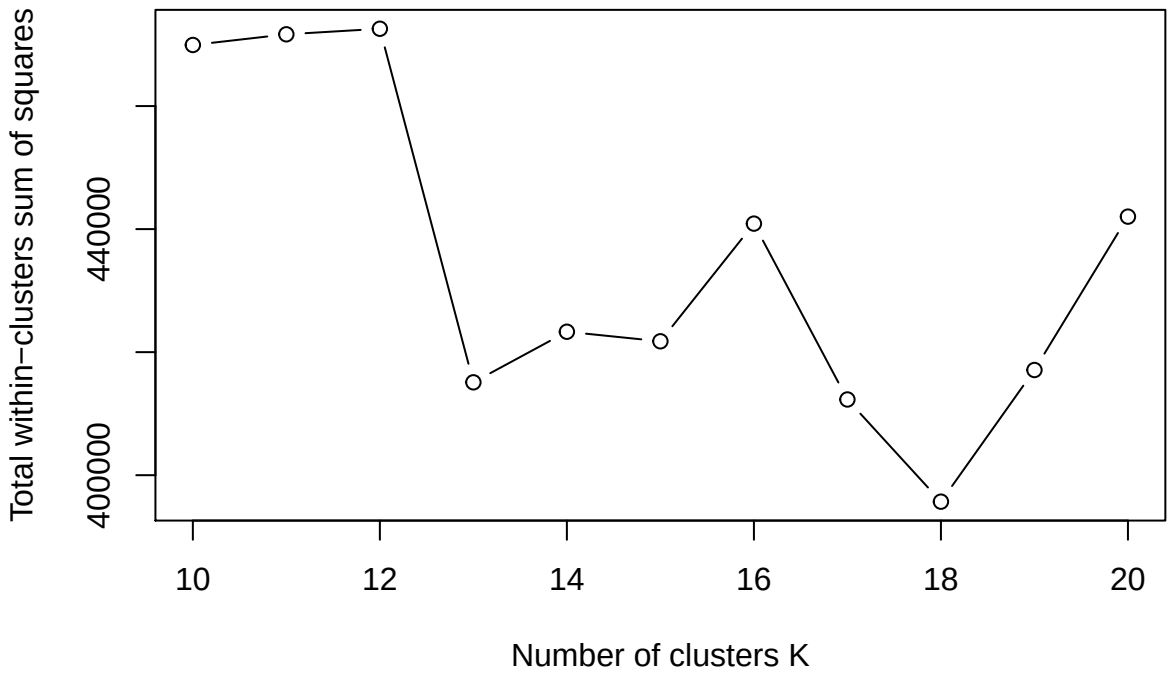
To perform an exploration of optimal clustering, we run `mbkmeans()` through multiple iterations of `k`. Here in particular, we've set the `calc_wcss` parameter to true to determine where our optimal number of `k` occurs. Further, we save the overall results once again into our `metadata` slot.

```
set.seed(1234)
wcss <- lapply(10:20, function(k) {
    cl <- mbkmeans(sce, reduceMethod = "PCA",
                   clusters = k,
                   batch_size = 50,
                   num_init = 10, max_iters = 100,
                   calc_wcss = TRUE)
})

metadata(sce)$wcss <- wcss
```

We can then plot the corresponding within-clusters sum of squares to determine our optimal `k`.

```
plot(10:20, sapply(metadata(sce)$wcss, function(x) sum(x$WCSS_per_cluster)),
     type = "b",
     xlab = "Number of clusters K",
     ylab = "Total within-clusters sum of squares")
```



Following that, we can once again perform clustering, but this time changing our parametrization to improve our results by increasing batch size, setting the desired number of clusters, and increasing the number of initializations and max iterations the algorithm goes through.

```
set.seed(1234)

## Perform final clustering on clustering with k=13
```

```
clusters_final <- mbkmeans(sce, reduceMethod = "PCA", clusters = 13,
                           batch_size = 200,
                           num_init = 10, max_iters = 100)

## Save full results into metadata
metadata(sce)$clusters_final <- clusters_final

## Save the clusters into colData as a factor for plotting
colData(sce)$clusters_final <- as.factor(clusters_final$Clusters)
## alternately: sce$cluster_final
```

6.8 Visualization

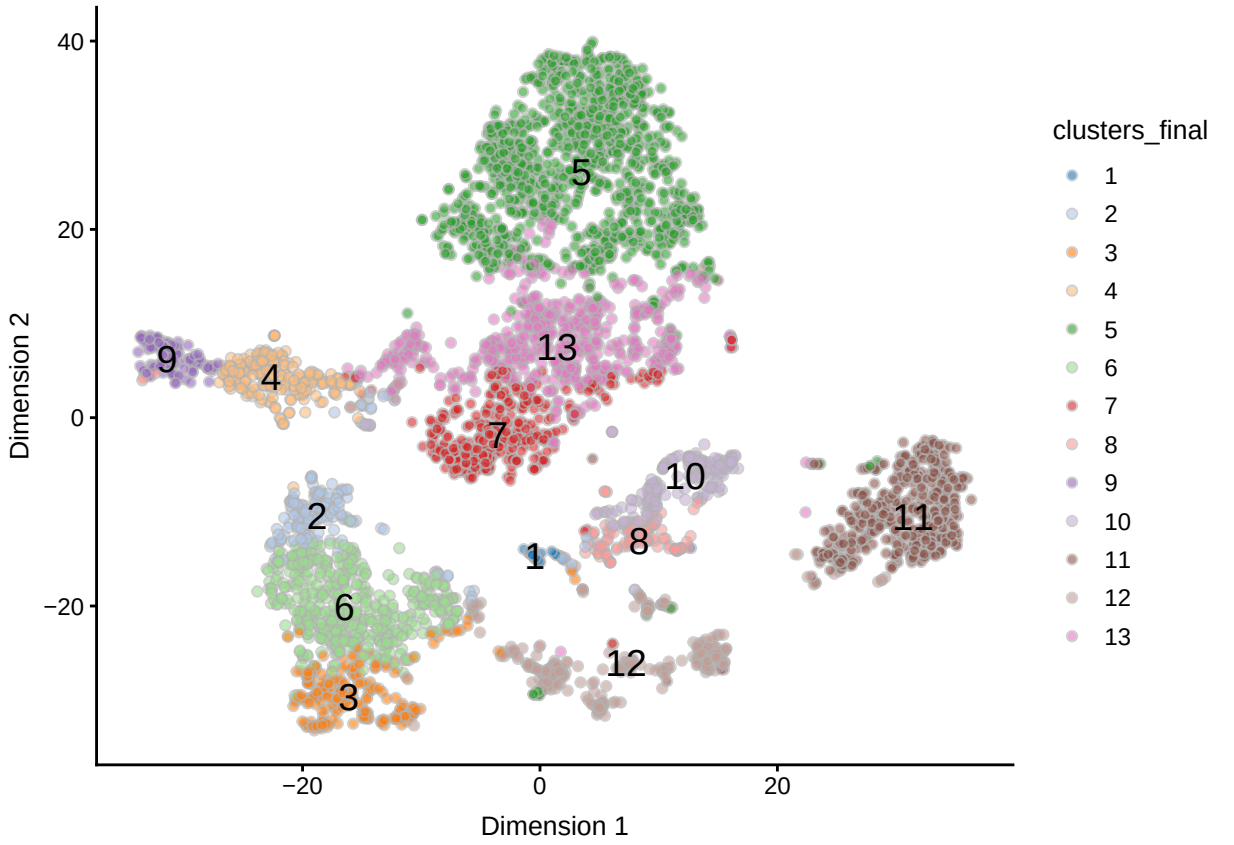
To visualize our final results, we can calculate the tSNE representation of our data, and then plot our final cluster designations. Note that here we use the `BNPARAM` argument of `runTSNE()` to supply a `BiocNeighbors` param, the `AnnoyParam()`

```
set.seed(1234)

## Calculate TSNE representation
sce <- runTSNE(sce,
               use_dimred = "PCA",
               external_neighbors = TRUE,
               BNPARAM = BiocNeighbors::AnnoyParam(),
               nthreads = 2,
               BPPARAM = BiocParallel::MulticoreParam(2))
```

We can then plot our resulting tSNE representation, here colouring by the final clustering assignment as well as adding a text overlay.

```
## Plot the TSNE
plotTSNE(sce,
         colour_by = "clusters_final",
         text_by = "clusters_final")
```



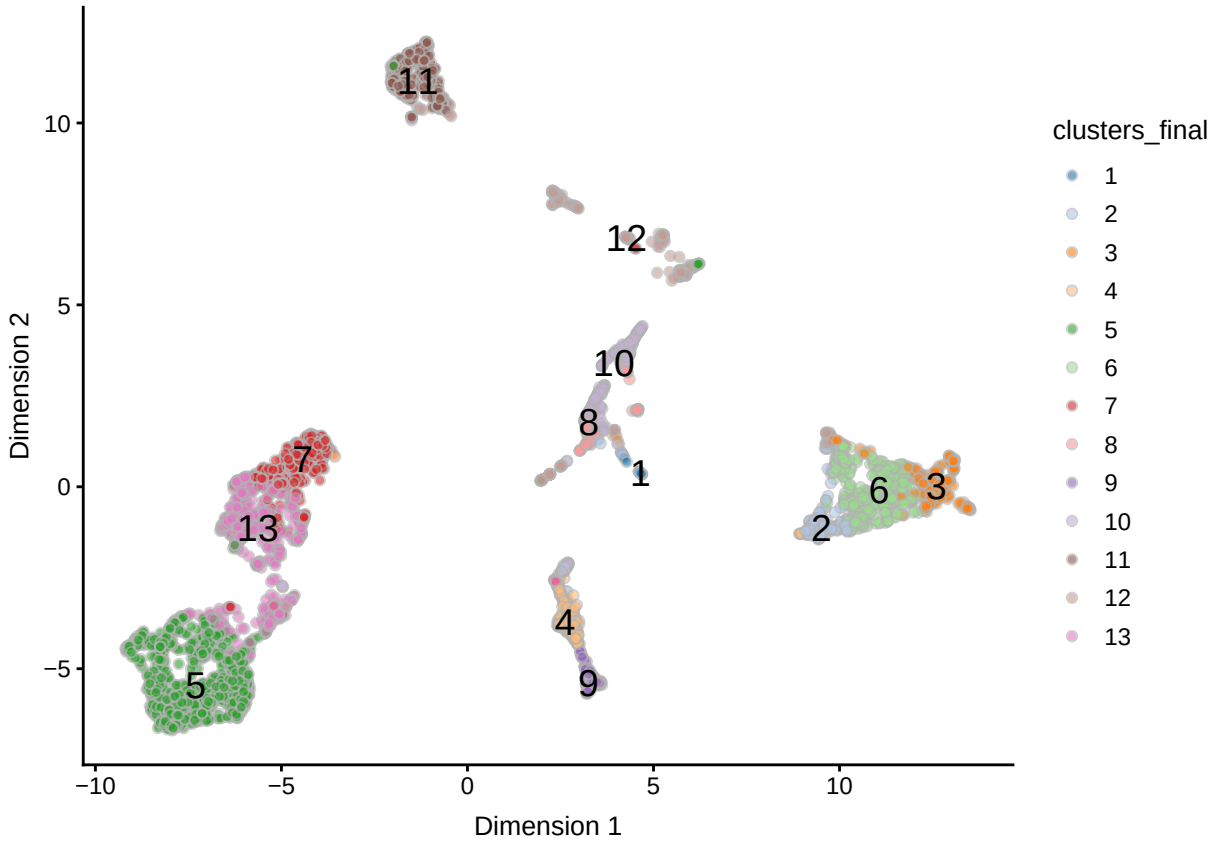
Furthermore, we can calculate the UMAP representation, using the very speedy `uwot` package implementation, manually assigning the results into the `reducedDims` slot as before.

```
set.seed(1234)

## Calculate umap representation and assign to reducedDims slot
um <- uwot::umap(reducedDim(sce, "PCA"), nn_method = "annoy",
                 approx_pow = TRUE, n_threads = 2)
reducedDim(sce, 'UMAP') <- um
```

And subsequently plot our UMAP representation as well:

```
plotReducedDim(sce, "UMAP",
               colour_by = "clusters_final",
               text_by = "clusters_final")
```



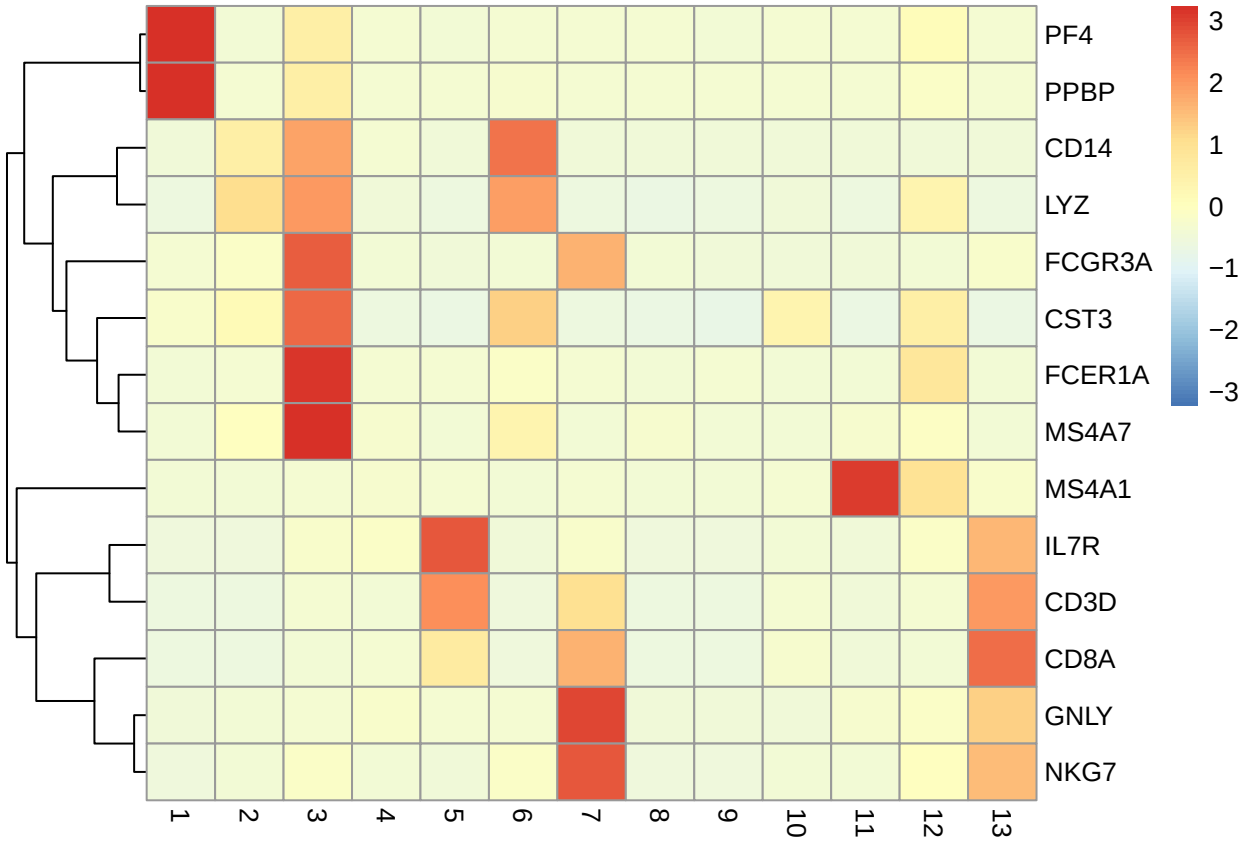
6.8.1 Marker genes

As a final exercise, we show some expression plots on key marker genes in the form of a heatmap and expression plots on the UMAP representation.

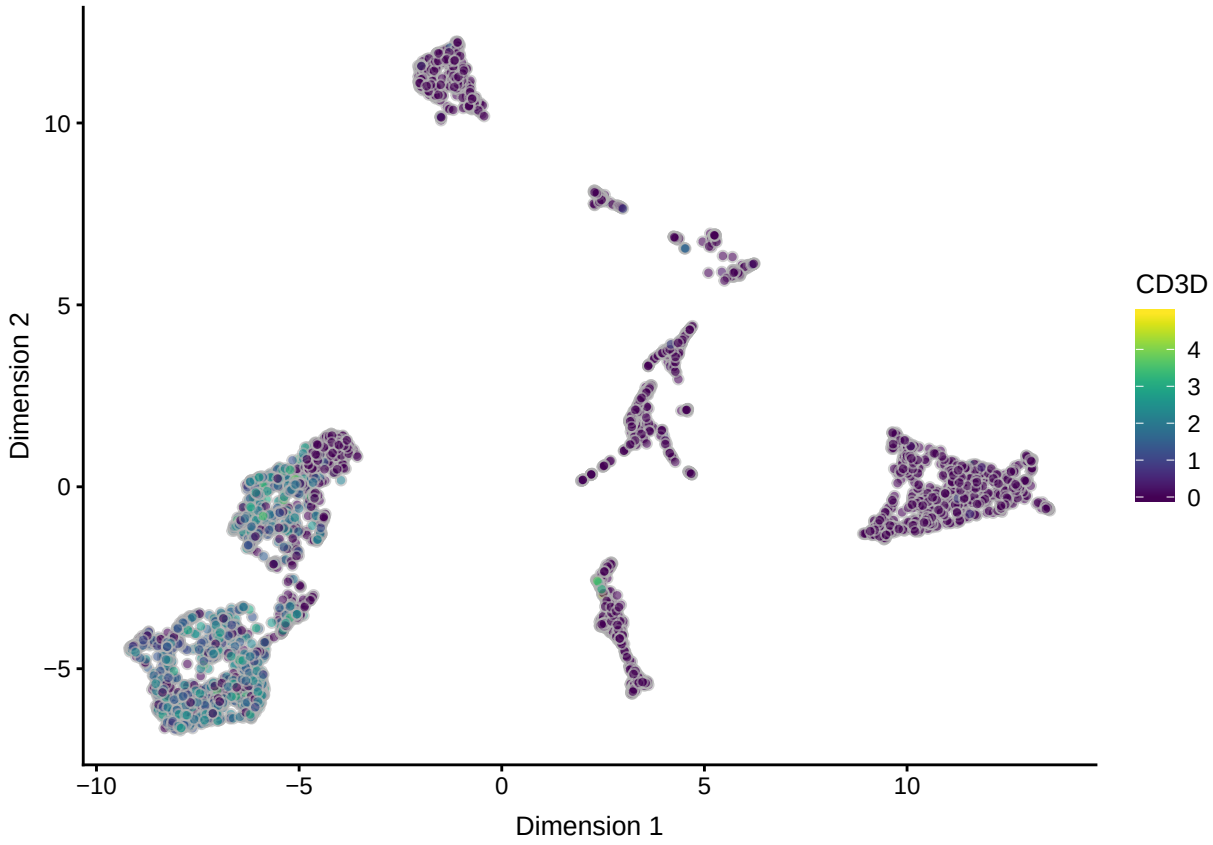
```
markers <- c("IL7R", #CD4
             "CD14",
             "LYZ", #CD14
             "MS4A1", #B cells
             "CD8A", #CD8
             "FCGR3A", #Monocytes
             "MS4A7",
             "GNLY",
             "NKG7", #NK cells
             "FCER1A", #Dendritic
             "CST3",
             "PPBP",
             "PF4", #megakaryocyte
             "CD3D"
             )

## Calculate the mean expression of each marker w.r.t. cluster
means <- apply(counts(sce[which(rowData(sce)$Symbol %in% markers),]), 1,
               tapply, sce$clusters_final, mean)
colnames(means) <- rowData(sce[colnames(means),])$Symbol
```

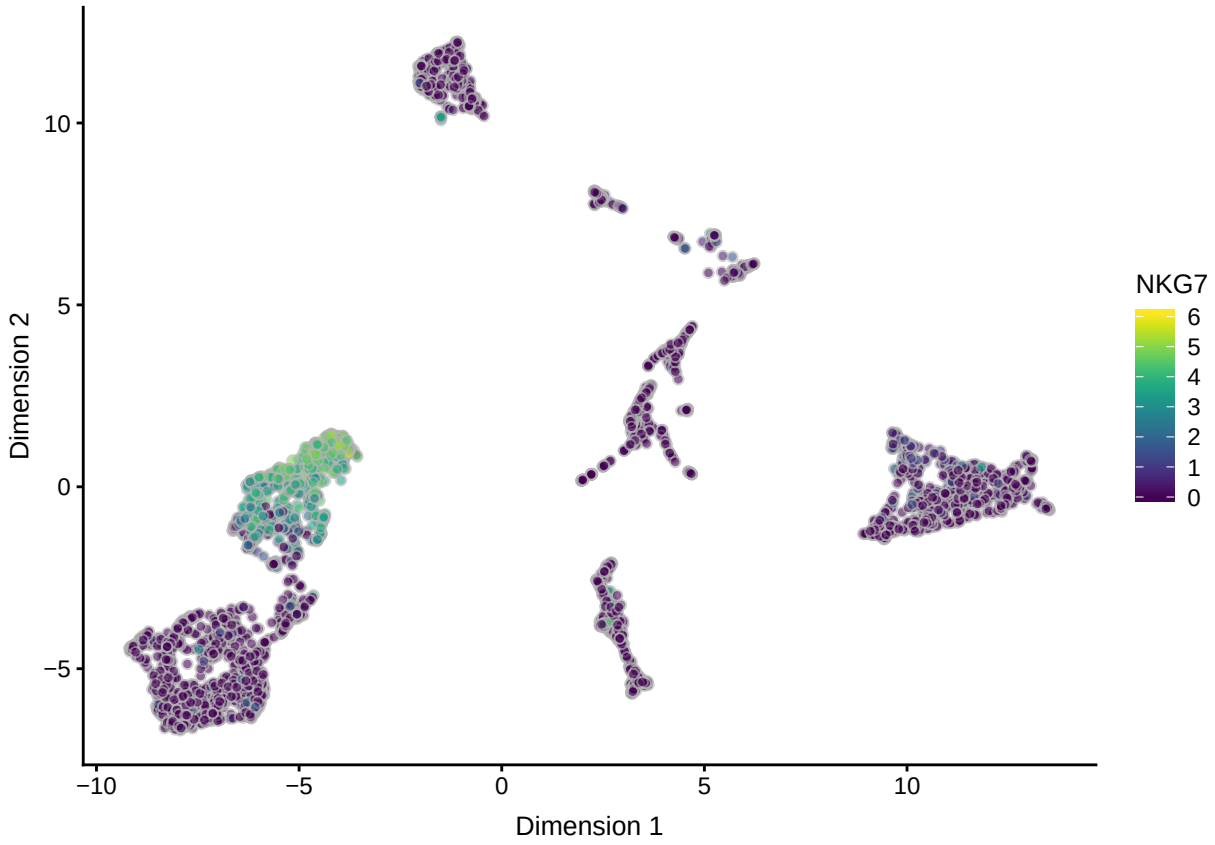
```
## Plot the heatmap of all marker genes
pheatmap(log2(t(means)+1), scale = "row", cluster_cols = FALSE)
```



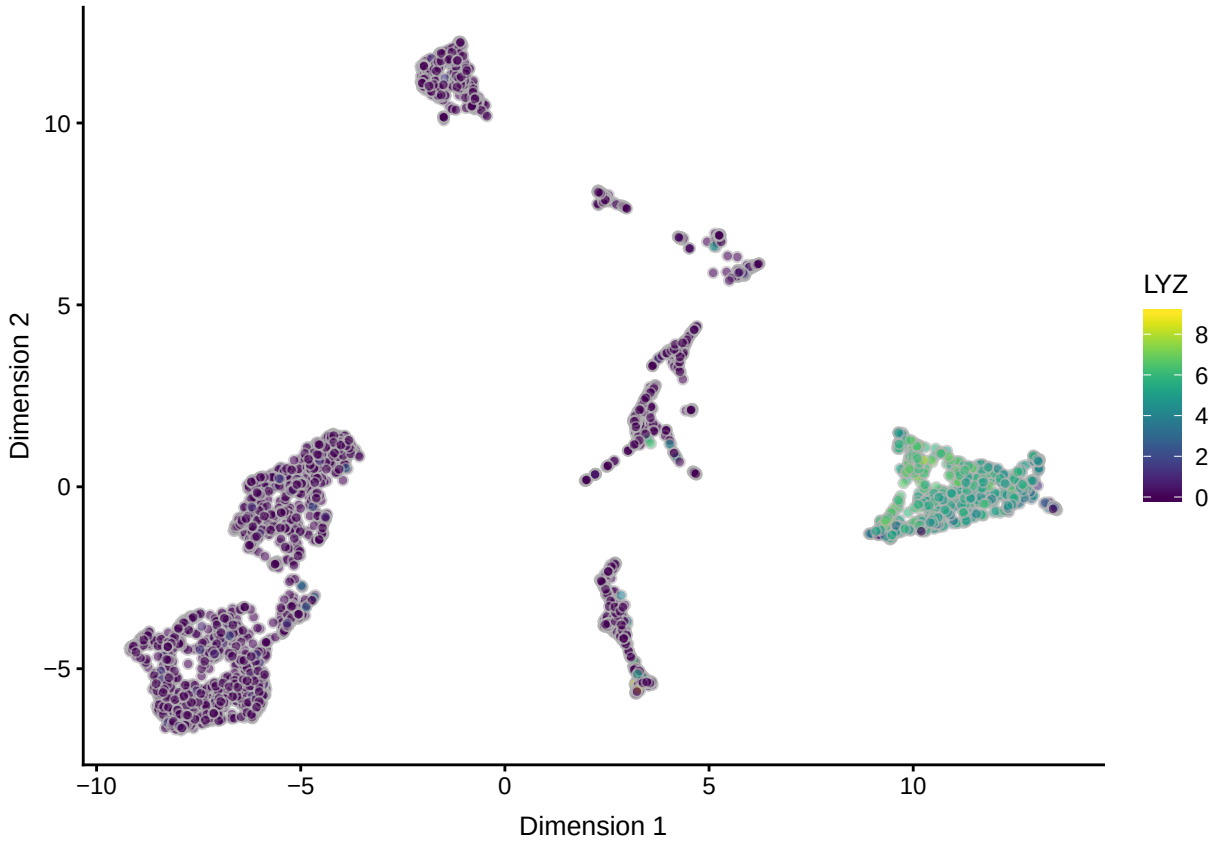

```
## Plot expression of select genes on UMAP
plotReducedDim(sce, "UMAP", colour_by = "CD3D")
```




```
plotReducedDim(sce, "UMAP", colour_by = "NKG7")
```




```
plotReducedDim(sce, "UMAP", colour_by = "LYZ")
```


Dimension 2
10
5
0
−5
−10
−5
0
5
10
Dimension 1
LYZ
8
6
4
2
0

Chapter 7

Integrating Datasets

authors: Stephanie C. Hicks, Robert A. Amezquita

The purpose of this case study is to demonstrate how to integrate multiple scRNA-seq datasets using R/Bioconductor packages. In this workflow, we go from preprocessing the data to integrating the data and visualizing it in a reduced dimensionality space to showcase the success of the integration approach using mutual nearest neighbors relative to a naive approach. This approach is helpful for ameliorating batch effects that can be introduced when combining data from different sequencing runs and/or platforms.

Here, we will be combining two datasets from 10X Genomics PBMC Data. One dataset is comprised of 3000 PBMCs from a healthy donor, and the other dataset is comprised of 4000 PBMCs from a different healthy donor. Our goal is to produce an integrated representation of this data to facilitate downstream analysis, such as clustering.

7.1 Package Requirements

These packages will be required for working through the vignette, and can be installed by running the code below. The data that we will be using comes from the `TENxPBMCData` package.

```
## required
BiocManager::install(c('scater', 'scran', 'limma', 'TENxPBMCData'))

## suggested
BiocManager::install(c('BiocParallel', 'BiocNeighbors'))
```

```
library(scater)
library(scran)
library(limma)
library(TENxPBMCData)
library(BiocParallel)
library(BiocNeighbors)
```

7.2 Loading the Data

Here we will be combining two different runs of scRNA-seq data - each from different healthy donors, and comprised of either 3000 cells (`pbmc3k`) or 4000 cells (`pbmc4k`). Note that these objects are already `SingleCellExperiment` objects.

```
pbmc3k <- TENxPBMCData('pbmc3k')
pbmc4k <- TENxPBMCData('pbmc4k')
```

7.3 Preprocessing

Here we walk through the steps required to produce a clean expression matrix, taking the raw count data through to a normalized representation.

7.3.1 Working with Common Genes

First, we find intersection of gene names and keep only the entries that are in common between the two datasets. We then reduce each of the individual datasets down to these matching entries (`keep_genes`) by subsetting.

```
keep_genes <- intersect(rownames(pbmc3k), rownames(pbmc4k))
pbmc3k <- pbmc3k[match(keep_genes, rownames(pbmc3k)), ]
pbmc4k <- pbmc4k[na.omit(match(keep_genes, rownames(pbmc4k))), ]
```

7.3.2 Cell and Gene Quality Control

First, for the combined data `sce` and the individual datasets `pbmc3k` and `pbmc4k`, we calculate essential quality control characteristics using the `scater` function `calculateQCMetrics()`. We then determine cells low quality cells by finding outliers with uncharacteristically low total counts or total number of features (genes) detected. We automate this into a function.

```
## For pbmc3k
pbmc3k <- calculateQCMetrics(pbmc3k)
low_lib_pbmc3k <- isOutlier(pbmc3k$log10_total_counts, type="lower", nmad=3)
low_genes_pbmc3k <- isOutlier(pbmc3k$log10_total_features_by_counts, type="lower", nmad=3)

## For pbmc4k
pbmc4k <- calculateQCMetrics(pbmc4k)
low_lib_pbmc4k <- isOutlier(pbmc4k$log10_total_counts, type="lower", nmad=3)
low_genes_pbmc4k <- isOutlier(pbmc4k$log10_total_features_by_counts, type="lower", nmad=3)
```

These results flag approximately 30 to 100 cells for removal from each of the datasets. We can then further subset our data to remove these cells by running the following:

```
pbmc3k <- pbmc3k[, !(low_lib_pbmc3k | low_genes_pbmc3k)]
pbmc4k <- pbmc4k[, !(low_lib_pbmc4k | low_genes_pbmc4k)]
```

7.3.3 Normalization

From here, we now compute the size factors using the `scran` package's `computeSumFactors()` function, and apply the size factors via the `scran` package's `normalize()` function to produce a new assay, `logcounts`, within each `SingleCellExperiment` object.

```
## compute the sizeFactors
pbmc3k <- computeSumFactors(pbmc3k)
pbmc4k <- computeSumFactors(pbmc4k)

## Normalize (using already calculated size factors)
pbmc3k <- normalize(pbmc3k)
pbmc4k <- normalize(pbmc4k)
```

7.4 Feature Selection

A key step across many data integration methods is the selection of informative features across the different experiments. This helps to speed up computation and possibly improve the resulting integration.

Once again, we rely on the `scran` package to identify the genes with the highest biological coefficient of variation, using the `trendVar()` and `decomposeVar()` functions to calculate the per gene variance and separate it into technical versus biological components. We perform this for each individual dataset:

```
fit_pbmc3k <- trendVar(pbmc3k, use.spikes=FALSE)
dec_pbmc3k <- decomposeVar(pbmc3k, fit_pbmc3k)
dec_pbmc3k$Symbol_TENx <- rowData(pbmc3k)$Symbol_TENx
dec_pbmc3k <- dec_pbmc3k[order(dec_pbmc3k$bio, decreasing = TRUE), ]

fit_pbmc4k <- trendVar(pbmc4k, use.spikes=FALSE)
dec_pbmc4k <- decomposeVar(pbmc4k, fit_pbmc4k)
dec_pbmc4k$Symbol_TENx <- rowData(pbmc4k)$Symbol_TENx
dec_pbmc4k <- dec_pbmc4k[order(dec_pbmc4k$bio, decreasing = TRUE), ]
```

Then select the most informative genes that are shared across *both* datasets:

```
universe <- intersect(rownames(dec_pbmc3k), rownames(dec_pbmc4k))
mean.bio <- (dec_pbmc3k[universe,"bio"] + dec_pbmc4k[universe,"bio"])/2
hvg_genes <- universe[mean.bio > 0]
```

7.5 Combining the Datasets

Finally, we combine the datasets into a unified `SingleCellExperiment` object for the downstream integration approaches, now that the data has been normalized (both within and between datasets) and the shared most informative features have been identified.

```
## total raw counts
counts_pbmc <- cbind(counts(pbmc3k), counts(pbmc4k))

## total normalized counts (with multibatch normalization)
logcounts_pbmc <- cbind(logcounts(pbmc3k), logcounts(pbmc4k))

sce <- SingleCellExperiment(
    assays = list(counts = counts_pbmc, logcounts = logcounts_pbmc),
    rowData = rowData(pbmc3k), # same as rowData(pbmc4k)
    colData = rbind(colData(pbmc3k), colData(pbmc4k))
)
```

For safekeeping, we will also store the `hvg_genes` from the prior section into the `sce` object's `metadata` slot via:

```
metadata(sce)$hvg_genes <- hvg_genes
```

7.6 Integrating Datasets

Here we will now be comparing the results of different approaches to integration.

7.6.1 Naive Approach

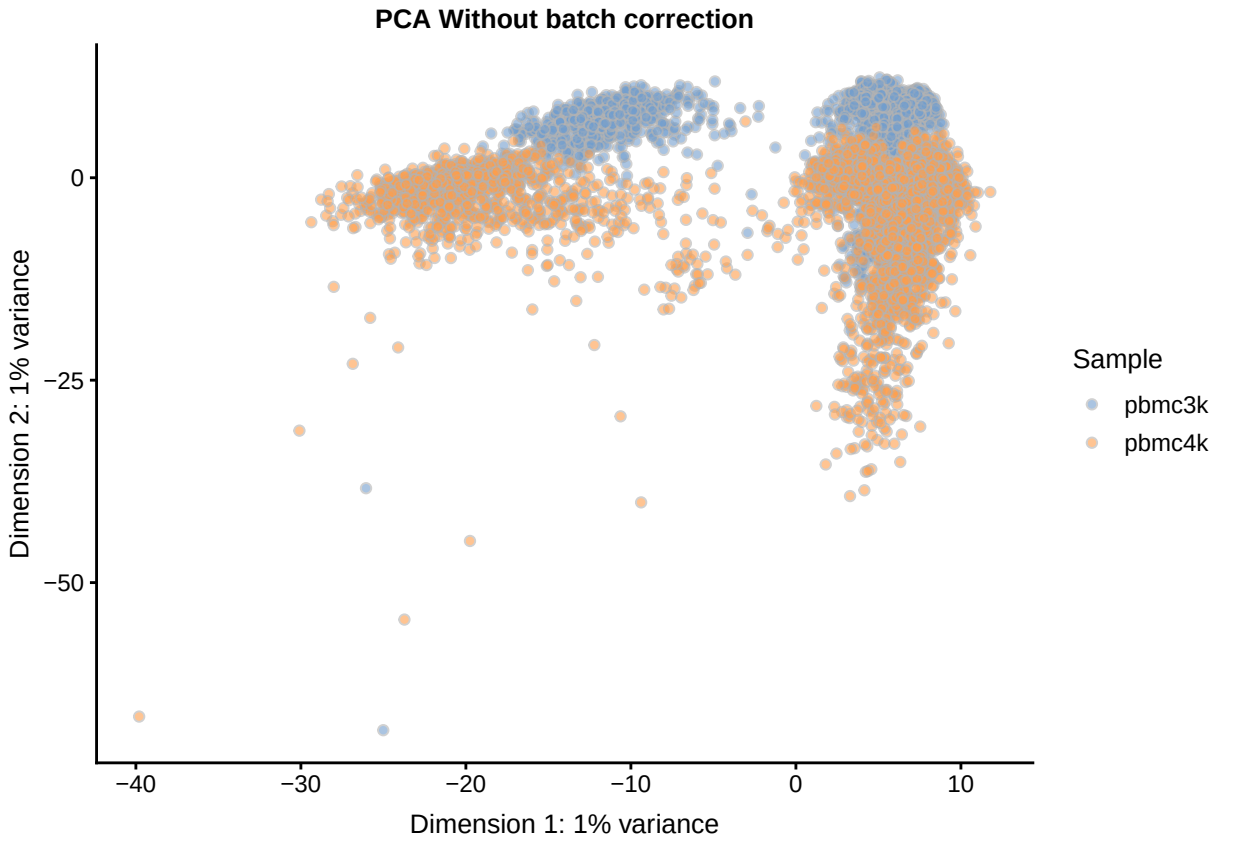
The naive approach simply entails visualizing the combined `sce` object post-normalization with no attempt at batch correction. Here we manually calculate the PCA represention on the normalized data (retrieved via `logcounts(sce)` or, similarly, via `assay(sce, "logcounts")` and then assign the result into the `reducedDim` slot of `sce`, naming it `"PCA_naive"`.

```
## Manual assignment of PCA to sce object
## px <- prcomp(t(logcounts(sce)[hvg_genes, ]))
## reducedDim(sce, "PCA_naive") <- px$x[, 1:20]

## Method for automating PCA calculation/saving into sce with additional
## parameters for speeding up calculation via Irlba/parallelization
sce <- runPCA(sce,
              ncomponents = 20,
              feature_set = hvg_genes,
              method = "irlba",
              BPPARAM = MulticoreParam(8))

names(reducedDims(sce)) <- "PCA_naive" # rename for clarity; prevent overwriting
```

```
plotReducedDim(sce, use_dimred = "PCA_naive",
               colour_by = "Sample") +
    ggtitle("PCA Without batch correction")
```



7.6.2 Limma Batch Correction

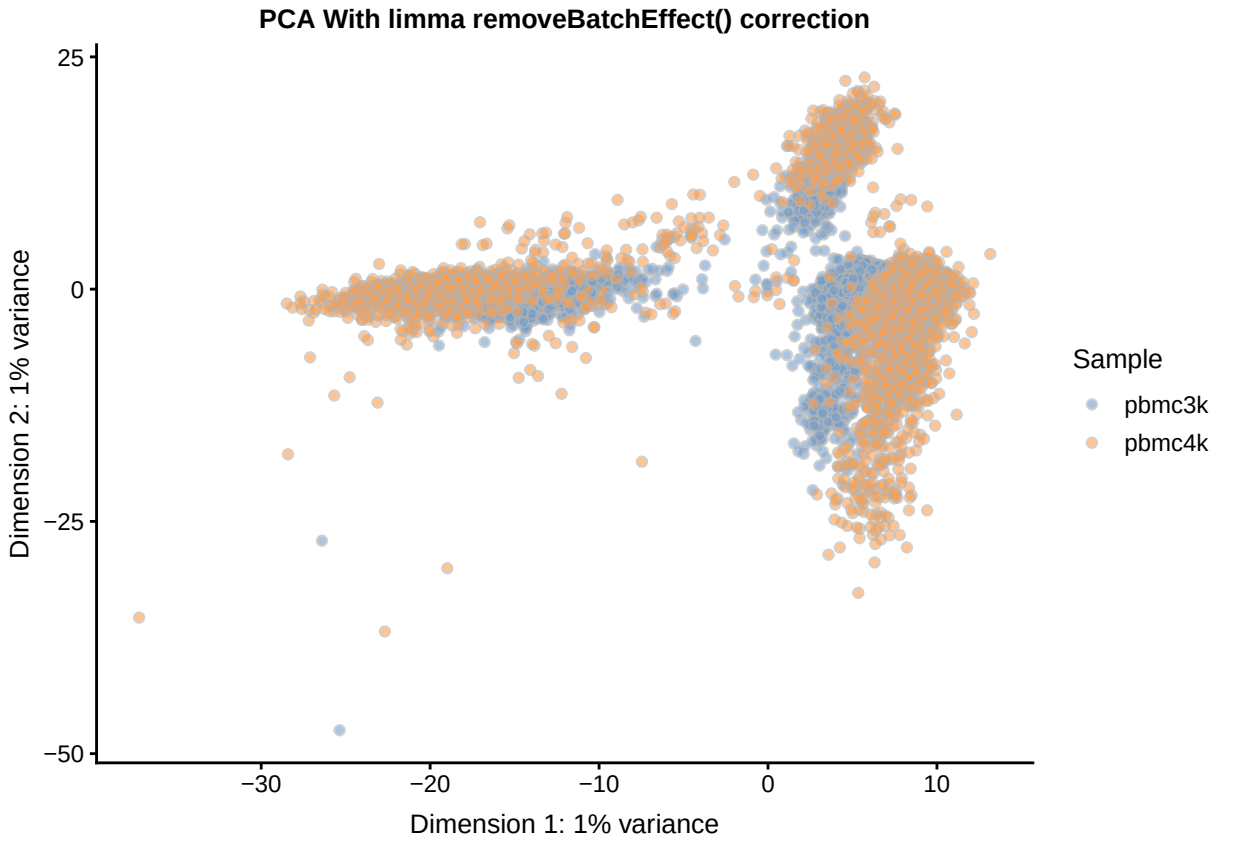
The `limma` package, a popular framework for the statistical analysis of RNA-seq, has a function `removeBatchEffect()` which will be used here to correct the normalized expression matrix `logcounts` across the two batches. The result will be assigned into the `assays` slot of the `sce` object as `limma_corrected`, and then used for PCA, saving the result in the `reducedDim` slot as `"PCA_limma"`.

```
limma_corrected <- limma::removeBatchEffect(logcounts(sce), batch = sce$Sample)
assay(sce, "logcounts_limma") <- limma_corrected ## add new assay

## Automated way of running PCA
sce <- runPCA(sce,
              ncomponents = 20,
              feature_set = hvg_genes,
              exprs_values = "logcounts_limma",
              method = "irlba",
              BPPARAM = MulticoreParam(8))

names(reducedDims(sce))[2] <- "PCA_limma"
```

```
plotReducedDim(sce, use_dimred = "PCA_limma",
               colour_by = "Sample") +
    ggtitle("PCA With limma removeBatchEffect() correction")
```



7.6.3 MNN Approach

The mutual nearest neighbors (MNN) approach within the `scran` package utilizes a novel approach to adjust for batch effects. The `fastMNN()` function returns a representation of the data with reduced dimensionality, which can be used in a similar fashion to other lower-dimensional representations such as PCA. In particular, this representation can be used for downstream methods such as clustering.

Where `fastMNN()` differs from other integration methods such as the limma approach above is that it does *not* produce a batch-corrected expression matrix. Thus, the result from `fastMNN()` should solely be treated as a reduced dimensionality representation, suitable for direct plotting, TSNE/UMAP, clustering, and trajectory analysis that relies on such results.

Prior to running `fastMNN()`, we rescale each batch to adjust for differences in sequencing depth between batches. The `multiBatchNorm()` function from the `scran` package (in the future, it will ported to be in the `batchelor` package) recomputes log-normalized expression values after adjusting the size factors for systematic differences in coverage between `SingleCellExperiment` objects. The previously computed size factors only remove biases between cells within a single batch. This improves the quality of the correction step by removing one aspect of the technical differences between batches.

```
rescaled <- multiBatchNorm(pbmc3k, pbmc4k)
pbmc3k_rescaled <- rescaled[[1]]
pbmc4k_rescaled <- rescaled[[2]]
```

A basic means of running the `fastMNN` method with mostly default parameters is shown below:

```
## Basic method - not run
mnn_out <- fastMNN(pbmc3k_rescaled,
                   pbmc4k_rescaled,
                   subset.row = metadata(sce)$hvg_genes,
                   k = 20, d = 50, approximate = TRUE)
```

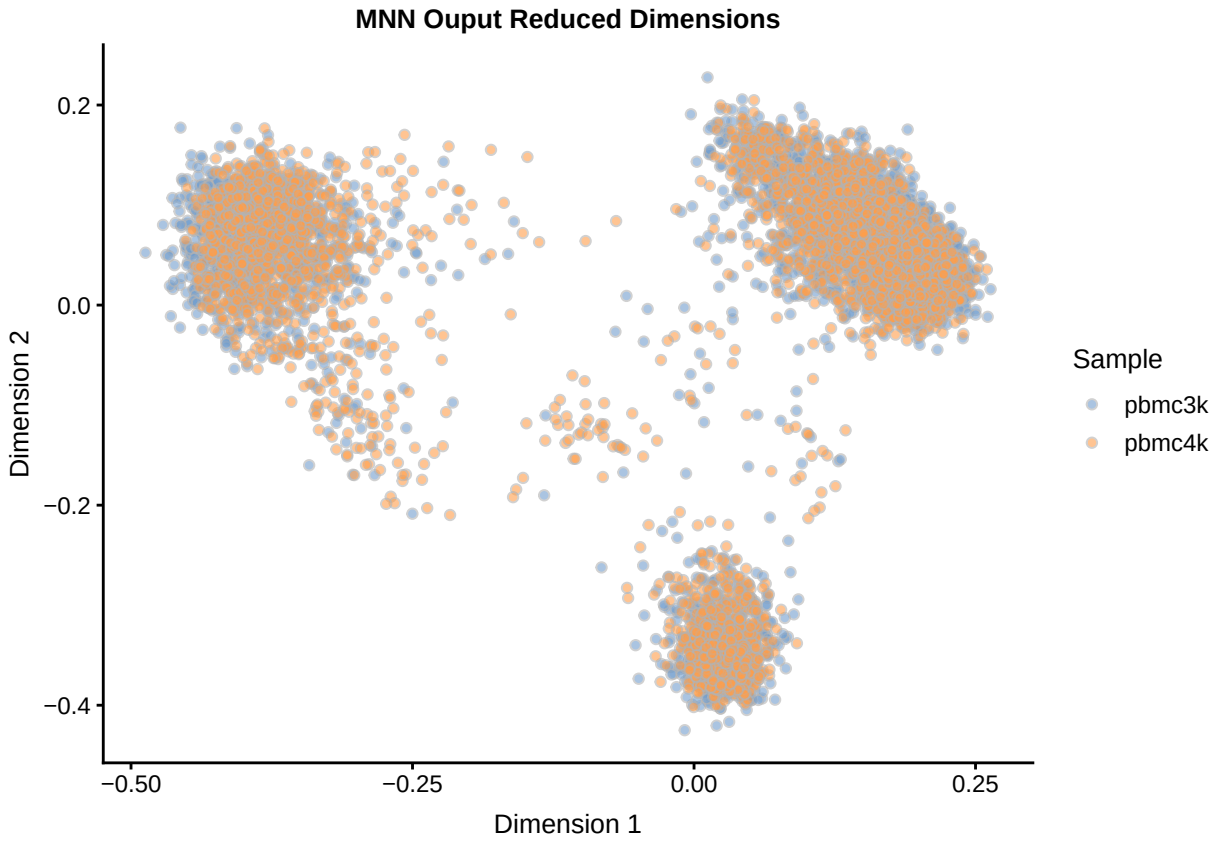
For finer-grained control of the algorithm, the `BNPARAM` can be specified to specify the specific nearest neighbors method to use from the `BiocNeighbors` package. Here we make use of the `Annoy` library via the `BiocNeighbors::AnnoyParam()` argument. Furthermore, we enable parallelization via the `BPPARAM` argument, enabling multicore processing. We save the reduced-dimension MNN representation into the `reducedDims` slot of our `sce` object just as before.

```
## Adding parallelization and Annoy method for approximate nearest neighbors
## this makes fastMNN faster on large data
mnn_out <- fastMNN(pbmc3k_rescaled, #[hvg_genes, ],
                   pbmc4k_rescaled, #[hvg_genes, ],
                   subset.row = metadata(sce)$hvg_genes,
                   k = 20, d = 50, approximate = TRUE,
                   BNPARAM = BiocNeighbors::AnnoyParam(),
                   BPPARAM = BiocParallel::MulticoreParam(8))

reducedDim(sce, "MNN") <- mnn_out$correct
```

Now we can plot the result:

```
plotReducedDim(sce, use_dimred = "MNN",
                    colour_by = "Sample") +
    ggtitle("MNN Ouput Reduced Dimensions")
```



Following MNN, the expression data can be supplied to other statistical frameworks that are better suited to handle batch effects. For example, in the case of differential expression, the batch effect could be regressed out within the statistical framework. Furthermore, cell-level characteristics such as cluster identification and trajectory analysis can be performed on the MNN reduced-dimension representation of the data.

Chapter 8

Clustering

author: Robert A. Amezquita

The purpose of this case study is to demonstrate various approaches to clustering scRNA-seq datasets using R/Bioconductor packages. In this workflow, we go from preprocessing the data to clustering the data. Furthermore, we highlight methods which are especially suitable for large datasets.

Here we will be working with a dataset from the CellBench scRNA-seq benchmarking dataset collection. Specifically, we will be working with the `sc_10x_5cl` dataset, which contains 5 sorted cell lines that were sequencing on the 10X Genomics platform. We will use this to showcase our different clustering strategies.

8.1 Package Requirements

These packages will be required for working through the vignette, and can be installed by running the code below:

```
BiocManager::install(c('scater', 'scran',
                       'SC3', 'clusterExperiment', 'BiocNeighbors',
                       'BiocParallel'))
```

```
library(scater)
library(scran)
library(SC3)
library(clusterExperiment)
library(BiocParallel)
```

8.2 Loading the Data

To follow along with the workflow, we use the CellBench_data Github repository `data` folder's `sincell_with_class_5cl.RData` workspace, which contains an object called `sc_10x_5cl_qc`. While this data will eventually be submitted to ExperimentHub, a central repository for datasets, it is currently as yet unavailable there.

For the time being, the data can be found in this book's Github repo in the `_rfiles/_data` folder. The data arrives as a `SingleCellExperiment` class object.

```
sce <- readRDS("_rfiles/_data/cellbench_sce_sc_10x_5cl_qc.rds")
```

8.3 Understanding the Data

Within the cell metadata stored in the `colData` slot of our `sce` object, we see that information regarding the cell line of origin for each cell is stored under the column `cell_line`, and furthermore can count the number of instances of each cell line present in the dataset as follows:

```
table(sce$cell_line)
```

```
##
##   A549  H1975  H2228   H838 HCC827
##   1244    515    751    840    568
```

We will be using this information to illustrate our dimensionality reduction by highlighting each of the cell lines, and furthermore using this information to verify our clustering performance. While ground truth is not often known in practice, such benchmarking datasets are essential for validation of methods. While simulations are another source of establishing ground truth, this dataset is particularly appealing as it is derived under realistic experimental conditions.

8.4 Preprocessing

This dataset has already undergone cell quality control, so in this case we will skip this step (for this, we refer to the Workflow on Integrating Datasets. Thus, we will begin with the feature selection step and go on from there.

8.5 Feature Selection

In order to improve performance in terms of both speed and quality of results, we will start with defining a set of highly variable genes based on the biological coefficient of variability (see the scran vignette for details).

```
## Calculating highly variable genes -------------------
fit <- trendVar(sce, parametric=TRUE, use.spikes = FALSE)
dec <- decomposeVar(sce, fit)
hvg_genes <- rownames(dec[dec$FDR < 0.00001, ]) # 1874 genes

## remove uninteresting/unknown genes (ribosomal, mitochondrial)
bad_patterns <- 'RP[0-9]|^MT|[0-9][0-9][0-9][0-9]|^RPL|^RPS'
hvg_genes <- hvg_genes[!grepl(bad_patterns, hvg_genes)] # 1713 genes

## save the decomposed variance table and hvg_genes into metadata for safekeeping
metadata(sce)$hvg_genes <- hvg_genes
metadata(sce)$dec_var <- dec
```

8.6 Dimensionality Reduction

Here we will use PCA to calculate the first 20 principal components with our highly variable geneset, using the `irlba` method for a faster approximate version of the calculation. These PCs will then form the basis of our subsequent tSNE calculation.

```
sce <- runPCA(sce, method = "irlba",
              ncomponents = 20,
              feature_set = metadata(sce)$hvg_genes)
sce <- runTSNE(sce,
               perplexity = 30,
               feature_set = metadata(sce)$hvg_genes)
sce <- runUMAP(sce,
               n_neighbors = 15,
               feature_set = metadata(sce)$hvg_genes)
```

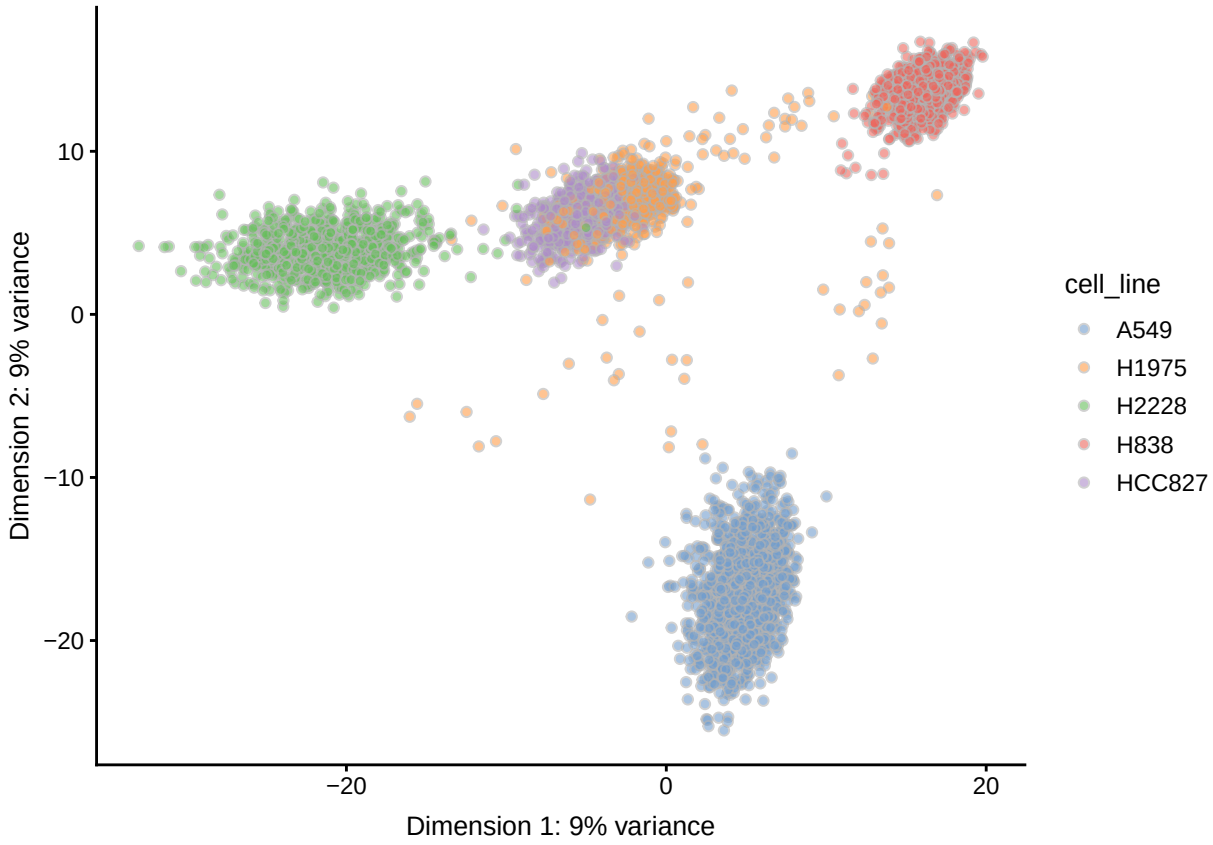
Here, we see that PCA nicely separates most of the cell lines, with the exception of HCC827 and H1975. On the other hand, tSNE nicely manages to separate the 5 cell lines, but interestingly produces two separate clusters for cell line H1975.

We leave it as an exercise to the interested reader to modify the essential tuning parameters, `perplexity`, and `n_neighbors` for TSNE and UMAP respectively to see how the dimensionality reduction embeddings change. Here we have used the default parameters, but simply made them explicit. Information on these and other parameters can be seen by accessing the help for each function.

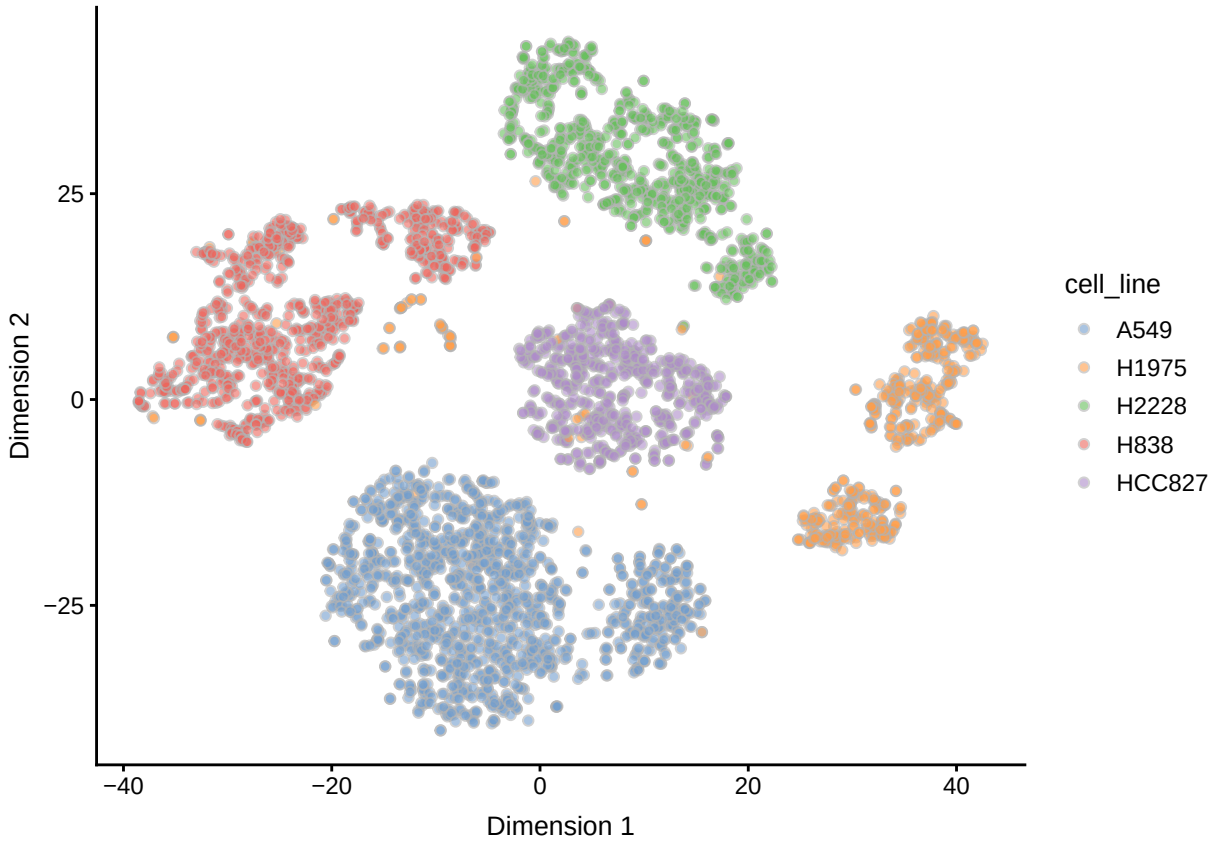
Note that clustering performance is *not* necessarily dependent on the dimensionality reduction results (to answer this, it will depend on the specific clustering technique at hand).

However, clustering performance indeed *may* be reflected in the dimensionality reduction results. For example, given the PCA result, we might expect some confusion between HCC827 and H1975, and based on the UMAP results, we might possibly even see two clusters within the H1975 cell line. Note however that these dimensionality reduction embeddings are also a function of the tuning parameters noted above.

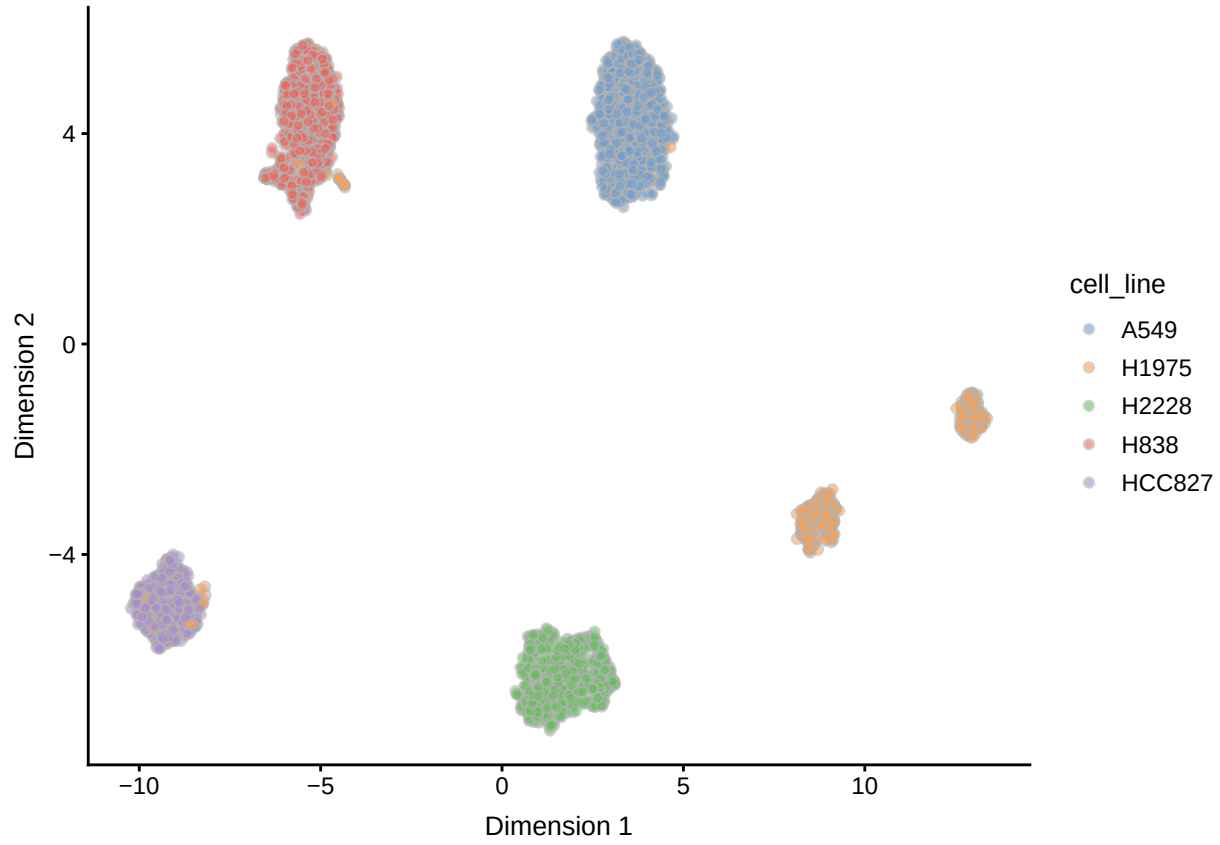
```
plotReducedDim(sce, "PCA", colour_by = "cell_line")
```




```
plotReducedDim(sce, "TSNE", colour_by = "cell_line")
```




```
plotReducedDim(sce, "UMAP", colour_by = "cell_line")
```



8.7 Clustering

Here we highlight different frameworks for clustering. The first two, using the `sc3` and `clusterExperiment` Bioconductor packages, are fuller implementations that can test across multiple parametrizations and furthermore inspect the quality of the clustering results. The `BiocNeighbors` package is briefly highlighted to show a minimal alternative to clustering that emphasizes speed.

8.7.1 SC3

The `sc3` package provides a simple framework that allows users to test for many k's, e.g. numbers of clusters, and subsequently compare the results from these differing k's in both quantitative and qualitative ways to pick an optimal k result. Below, we first set an essential `rowData` feature required by the package to run. Subsequently, we run the `sc3()` function in this example using a subset of the data (only the highly variable genes, `hvg_genes`, testing k's 3 through 6.

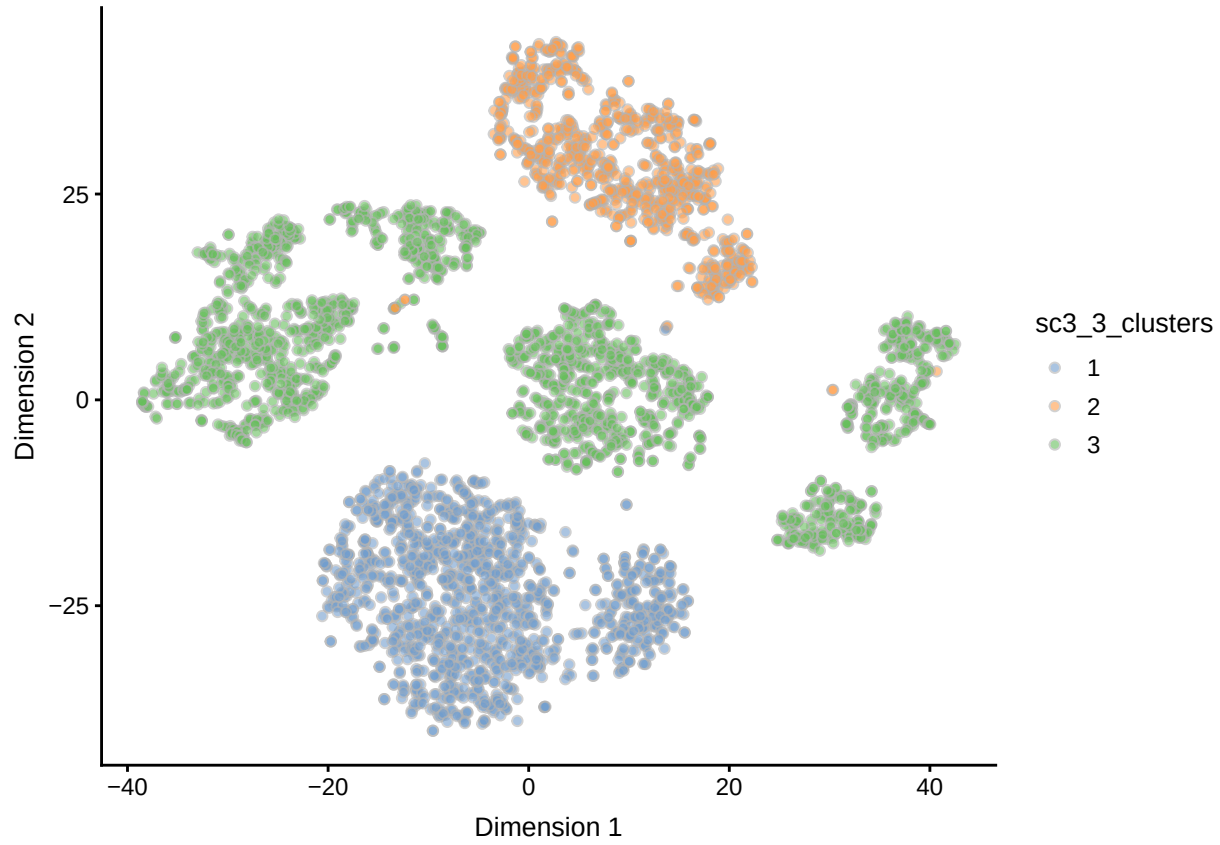
```
## SC3 requires this column to be appended
rowData(sce)$feature_symbol <- rownames(sce)

## SC3 will return an SCE object with appended "sc3_" columns
sce <- sc3(sce[metadata(sce)$hvg_genes, ],
           ks = 3:6,
           k_estimator = TRUE)
```

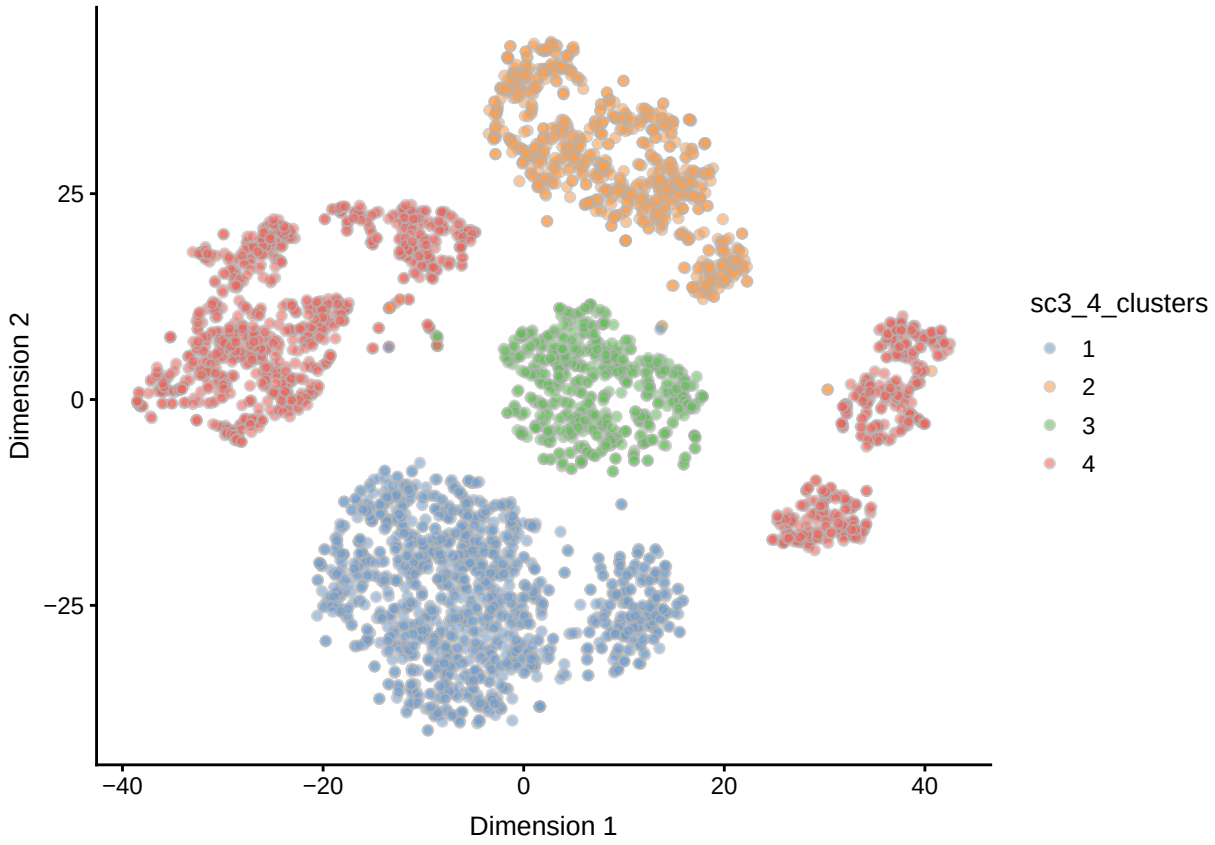
After using `sc3()`, the function returns the original `SingleCellExperiment` object, but with new columns in `colData(sce)` corresponding to the different `ks` supplied to the function, as well as a full representation of the analysis that is stored in `metadata(sce)$sc3`, which includes an estimate of the optimal `k` (as dictated by the `k_estimator = TRUE` argument above).

Below, we show the clustering results of the `ks` we supplied, 3 through 6, shown on the TSNE representation of the data. We use the `scater` package function `plotReducedDim()` to produce the individual plots, and for the sake of this vignette, wrap them together into a single plot using the `patchwork` package.

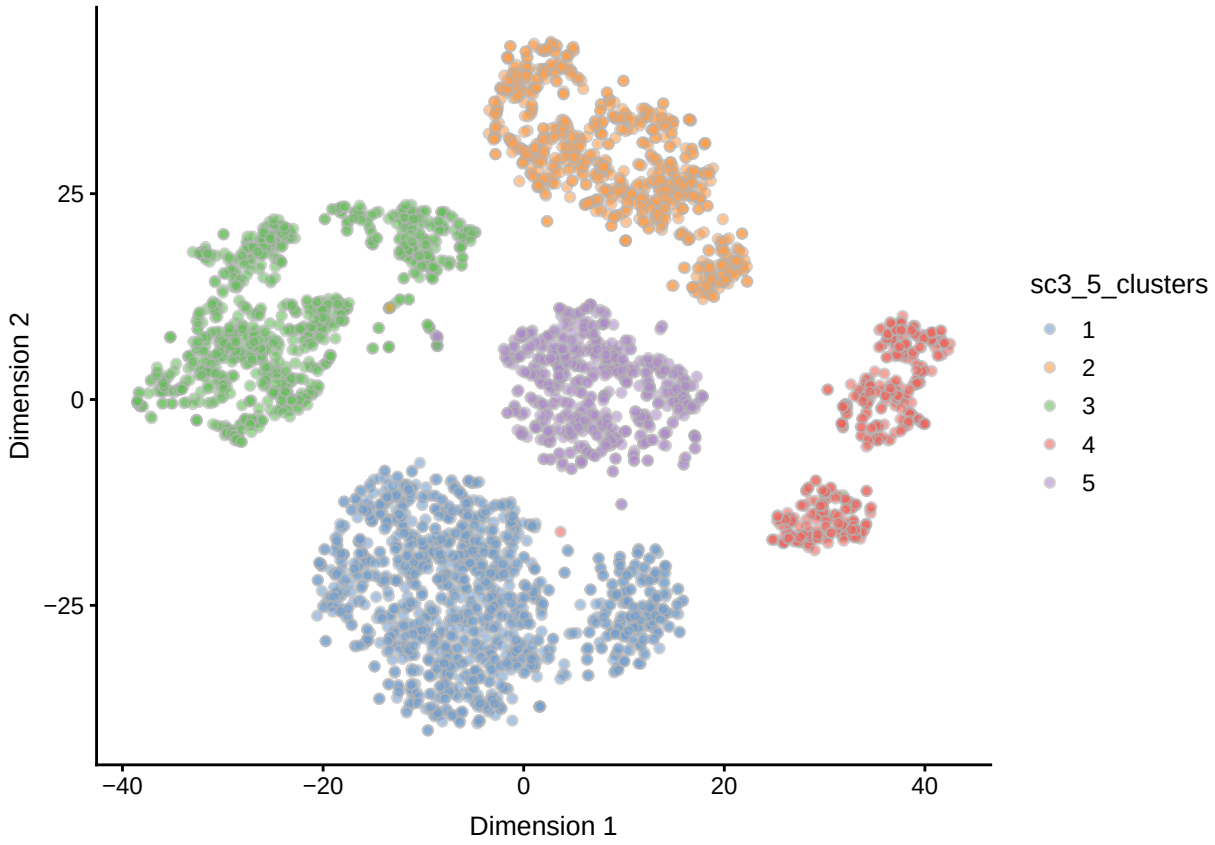
```
plotReducedDim(sce, use_dimred = "TSNE", colour_by = "sc3_3_clusters")
```



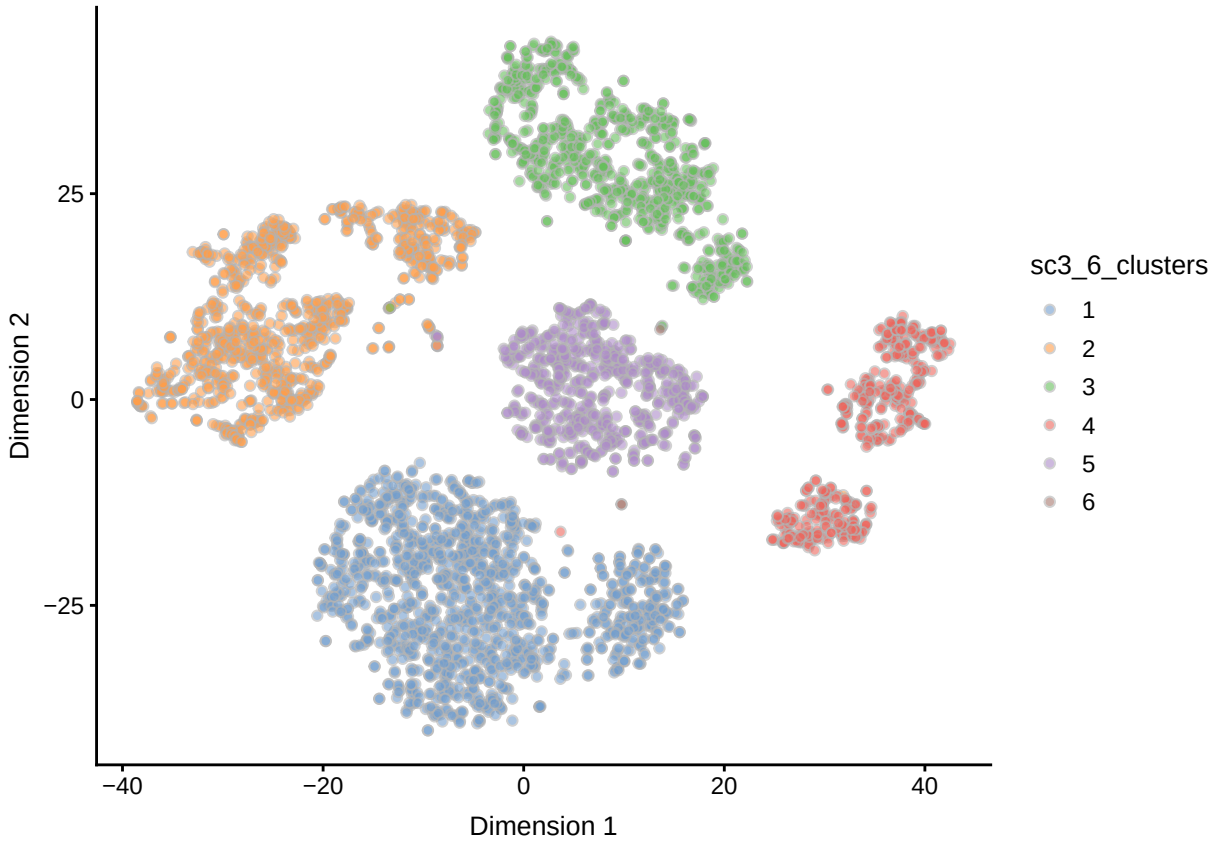

```
plotReducedDim(sce, use_dimred = "TSNE", colour_by = "sc3_4_clusters")
```




```
plotReducedDim(sce, use_dimred = "TSNE", colour_by = "sc3_5_clusters")
```




```
plotReducedDim(sce, use_dimred = "TSNE", colour_by = "sc3_6_clusters")
```



The `sc3` package contains many more utilities for exploring the stability of clustering and can even produce differential expression analysis results using the `biology = TRUE` argument within the `sc3()` function. We leave it to the interested reader to learn more about `sc3` via their vignette.

8.7.2 clusterExperiment

The `clusterExperiment` package uses the function `RSEC()` to calculate clusters across various parametrizations, and contains multiple parameters to do so. Here, we specify the parametrizations to iterate over using the `alphas` and `k0s` arguments below. We refer the reader to the help page for `?RSEC` to learn more about the different parameters. In the end, this exhaustive exercise is used to ultimately determine a consensus clustering that combines the information learned from the various parametrizations.

Note that `clusterExperiment` takes a long time to run, even with the reduced search space parametrized below. We recommend utilizing cluster resources for this job.

```
rsec <- RSEC(sce[metadata(sce)$hvg_genes, ],
             reduceMethod = "PCA",
             nReducedDims = 20,
             alphas = c(0.1, 0.3), k0s = 4:6,
             consensusMinSize = 50,
             ncores = 8)
```

The `clusterExperiment` package contains many visualization tools for assessing the cluster assignment. Here we show two plots with the cluster assignments across our various parametrizations and a dendrogram showing the relatedness of the final consensus clustering. Note that white denotes unassigned cells that were not assigned to a cluster.

```
plotClusters(rsec)
plotDendrogram(rsec)
```

Once again, we refer the interested reader to the vignette for `clusterExperiment` to learn more about extended functionality for visualizing the data and about the specifics of the clustering workflow.

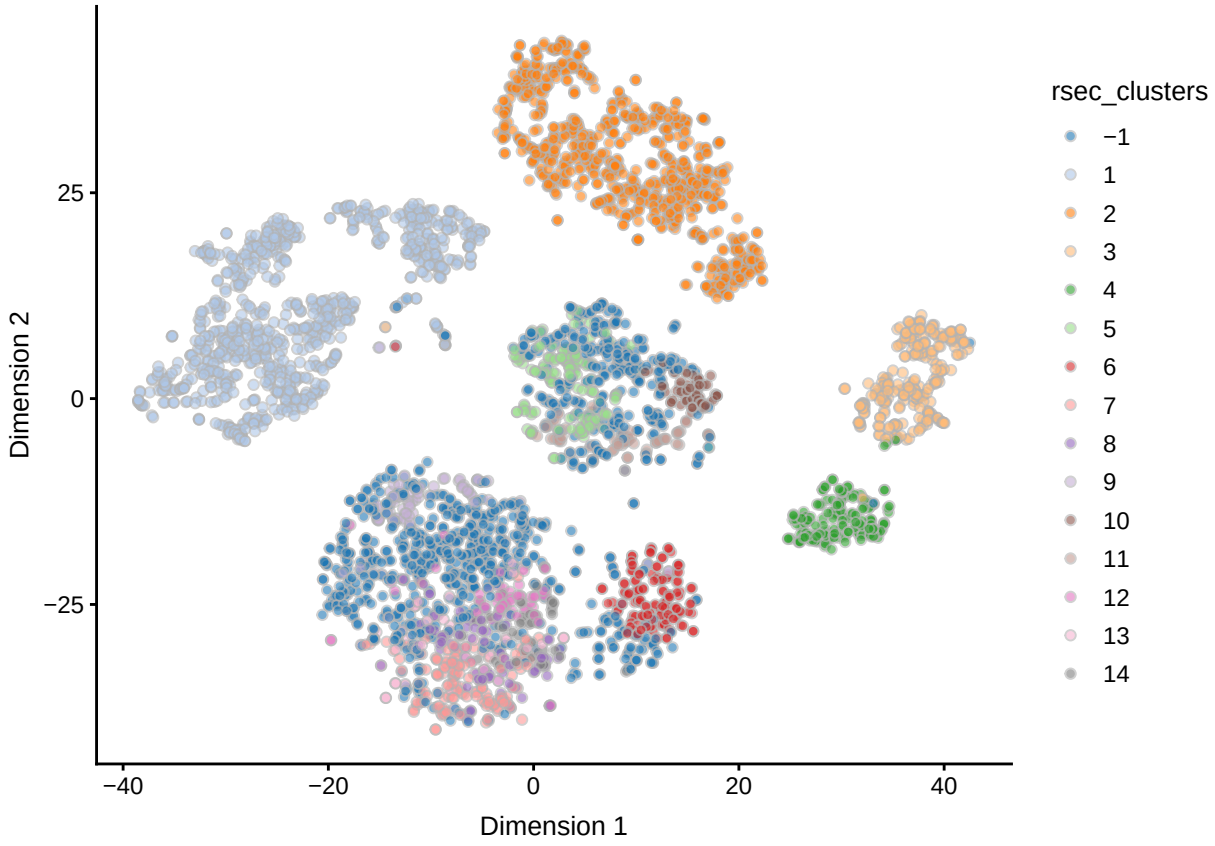
To get our final, consensus clustering assignments out (as shown by the `plotDendrogram()` function) we use the `primaryCluster()` function to retrieve the labels. To save this result into our `sce` object, we add it to the `colData` slot as shown below.

```
rsec_clusters <- primaryCluster(rsec)
rsec_clusters <- as.factor(rsec_clusters) # convert to a categorical variable

colData(sce)$rsec_clusters <- rsec_clusters
## sce$rsec_clusters <- rsec_clusters  # same as above
```

We can now use the `scater` function `plotReducedDim()` to visualize our results from `clusterExperiment` on a t-SNE plot. Note that the `-1` assignment denotes that a cell was not assigned a cluster.

```
plotReducedDim(sce, use_dimred = "TSNE", colour_by = "rsec_clusters")
```



8.7.3 Manual Clustering

Low-level clustering relies on building the shared- or k-nearest neighbors graphs manually, and then applying a graph-based clustering algorithm based on the resulting graph. One such wrapper to construct

the SNN/KNN graph comes from the `scran` package's `buildSNNGraph()` and `buildKNNGraph()` functions, respectively. The resulting `igraph` object from these functions can then be fed into any number of clustering algorithms provided by `igraph`. For example, louvain clustering is a popular algorithm that is implemented in the `cluster_louvain()` function.

Further, one additional parameter to note in the `buildSNNGraph()` function below is the `BNPARAM`, which provides even finer control over nearest-neighbors detection via the `BiocNeighbors` package. This parameter allows the user to specify an implementation from `BiocNeighbors` to use that has been designed for high-dimensional data. Here, we highlight the use of an approximate method via the Annoy algorithm by way of providing `AnnoyParam()`.

Given the limited scope of the calculations and optimizations provided herein via the use of already calculated dimension reduction results (via `use.dimred`), the `BNPARAM` as explained above, and parallelization via `BPPARAM`, this manual clustering approach is quite speedy. We use the louvain clustering algorithm to cluster on the resulting graph and save the result into our sce object's `colData` slot.

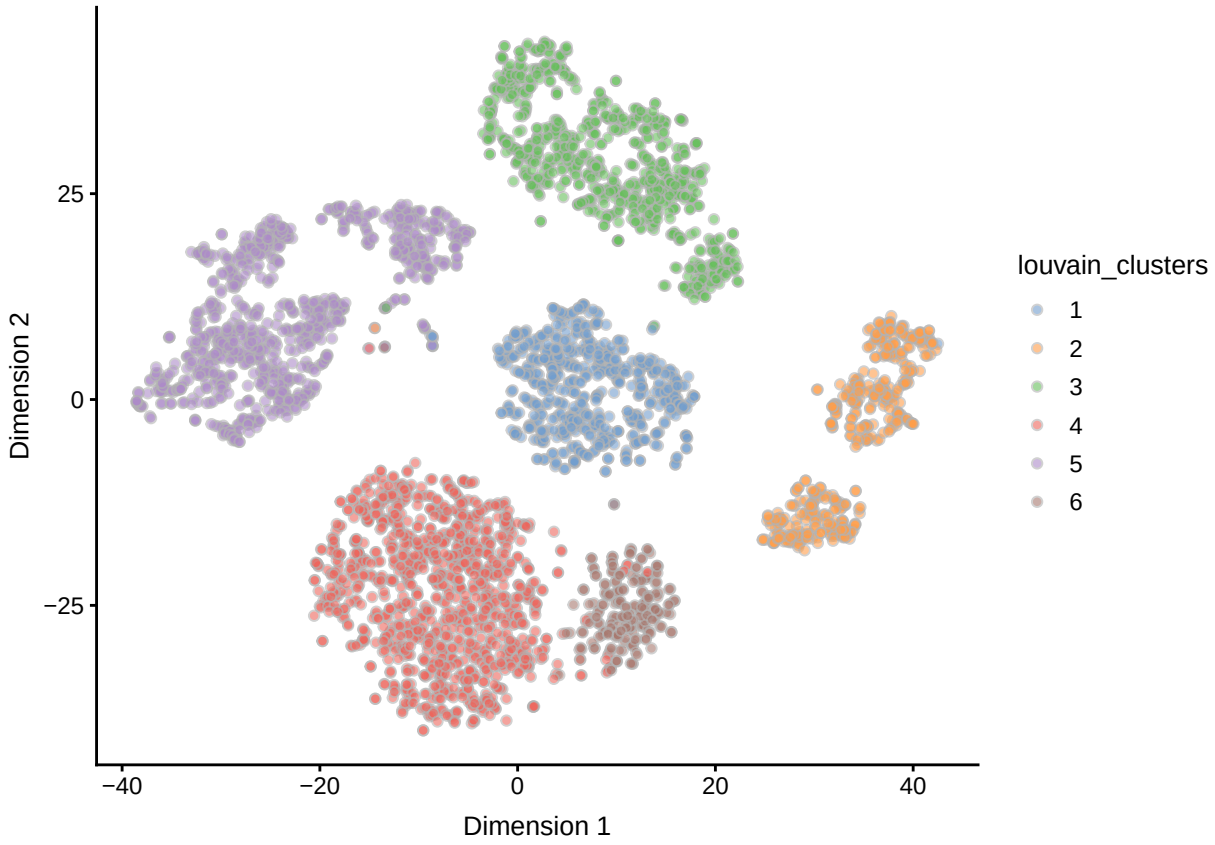
```
g <- buildSNNGraph(sce, k = 50,
                   use.dimred = 'PCA',
                   BNPARAM = AnnoyParam(),
                   BPPARAM = MulticoreParam(8))

louvain_clusters <- igraph::cluster_louvain(g)$membership

sce$louvain_clusters <- as.factor(louvain_clusters)
```

The results can then be plotted just as before:

```
plotReducedDim(sce, use_dimred = "TSNE", colour_by = "louvain_clusters")
```



The `BiocNeighbors` package is especially designed for developers to enable a unified interface for clustering algorithm specification, allowing for the algorithm to be easily switched within a Bioconductor package or workflow. For more information, check out the Bioconductor page for BiocNeighbors.