

# Analysis of single cell RNA-seq data

*Vladimir Kiselev (wikiselev), Tallulah Andrews, Jennifer Westoby (Jenni\_Westoby), Davis McCarthy (davisjmcc), Maren Büttner (marenbuettner) and Martin Hemberg (m\_hemberg)*

*2018-02-27*



# Contents

<b>1 About the course</b>	<b>7</b>
1.1 Video . . . . .	7
1.2 Registration . . . . .	7
1.3 GitHub . . . . .	7
1.4 Docker image (RStudio) . . . . .	8
1.5 Manual installation . . . . .	8
1.6 License . . . . .	8
1.7 Prerequisites . . . . .	8
1.8 Contact . . . . .	8
<b>2 Introduction to single-cell RNA-seq</b>	<b>9</b>
2.1 Bulk RNA-seq . . . . .	9
2.2 scRNA-seq . . . . .	9
2.3 Workflow . . . . .	9
2.4 Computational Analysis . . . . .	10
2.5 Challenges . . . . .	10
2.6 Experimental methods . . . . .	12
2.7 What platform to use for my experiment? . . . . .	14
<b>3 Processing Raw scRNA-seq Data</b>	<b>17</b>
3.1 FastQC . . . . .	17
3.2 Trimming Reads . . . . .	18
3.3 File formats . . . . .	19
3.4 Demultiplexing . . . . .	22
3.5 Using STAR to Align Reads . . . . .	28
3.6 Kallisto and Pseudo-Alignment . . . . .	29
<b>4 Construction of expression matrix</b>	<b>33</b>
4.1 Reads QC . . . . .	33
4.2 Reads alignment . . . . .	33
4.3 Alignment example . . . . .	34
4.4 Mapping QC . . . . .	35
4.5 Reads quantification . . . . .	35
4.6 Unique Molecular Identifiers (UMIs) . . . . .	36
<b>5 Introduction to R/Bioconductor</b>	<b>41</b>
5.1 Installing packages . . . . .	41
5.2 Installation instructions: . . . . .	42
5.3 Data-types/classes . . . . .	42
5.4 Basic data structures . . . . .	47
5.5 More information . . . . .	49
5.6 Data Types . . . . .	49

5.7 Bioconductor, <code>SingleCellExperiment</code> and <code>scater</code> . . . . .	53
5.8 An Introduction to <code>ggplot2</code> . . . . .	56
<b>6 Tabula Muris</b>	<b>65</b>
6.1 Introduction . . . . .	65
6.2 Downloading the data . . . . .	65
6.3 Reading the data (Smartseq2) . . . . .	66
6.4 Building a scater object . . . . .	68
6.5 Reading the data (10X) . . . . .	69
6.6 Building a scater object . . . . .	71
6.7 Advanced Exercise . . . . .	72
<b>7 Cleaning the Expression Matrix</b>	<b>73</b>
7.1 Expression QC (UMI) . . . . .	73
7.2 Expression QC (Reads) . . . . .	84
7.3 Data visualization . . . . .	93
7.4 Data visualization (Reads) . . . . .	100
7.5 Identifying confounding factors . . . . .	105
7.6 Identifying confounding factors (Reads) . . . . .	113
7.7 Normalization theory . . . . .	116
7.8 Normalization practice (UMI) . . . . .	119
7.9 Normalization practice (Reads) . . . . .	132
7.10 Dealing with confounders . . . . .	140
7.11 Dealing with confounders (Reads) . . . . .	168
<b>8 Biological Analysis</b>	<b>187</b>
8.1 Clustering Introduction . . . . .	187
8.2 Clustering example . . . . .	191
8.3 Feature Selection . . . . .	207
8.4 Pseudotime analysis . . . . .	217
8.5 Imputation . . . . .	253
8.6 Differential Expression (DE) analysis . . . . .	259
8.7 DE in a real dataset . . . . .	263
8.8 Comparing/Combining scRNASeq datasets . . . . .	274
8.9 Search scRNA-Seq data . . . . .	296
<b>9 Seurat</b>	<b>305</b>
9.1 <code>Seurat</code> object class . . . . .	305
9.2 Expression QC . . . . .	305
9.3 Normalization . . . . .	307
9.4 Highly variable genes . . . . .	307
9.5 Dealing with confounders . . . . .	308
9.6 Linear dimensionality reduction . . . . .	314
9.7 Significant PCs . . . . .	317
9.8 Clustering cells . . . . .	319
9.9 Marker genes . . . . .	321
9.10 <code>sessionInfo()</code> . . . . .	324
<b>10 “Ideal” scRNaseq pipeline (as of Oct 2017)</b>	<b>327</b>
10.1 Experimental Design . . . . .	327
10.2 Processing Reads . . . . .	327
10.3 Preparing Expression Matrix . . . . .	329
10.4 Biological Interpretation . . . . .	329
<b>11 Advanced exercises</b>	<b>331</b>

<b>12 Resources</b>	<b>333</b>
12.1 scRNA-seq protocols . . . . .	333
12.2 External RNA Control Consortium (ERCC) . . . . .	333
12.3 scRNA-seq analysis tools . . . . .	333
12.4 scRNA-seq public datasets . . . . .	333



# Chapter 1

## About the course

Today it is possible to obtain genome-wide transcriptome data from single cells using high-throughput sequencing (scRNA-seq). The main advantage of scRNA-seq is that the cellular resolution and the genome wide scope makes it possible to address issues that are intractable using other methods, e.g. bulk RNA-seq or single-cell RT-qPCR. However, to analyze scRNA-seq data, novel methods are required and some of the underlying assumptions for the methods developed for bulk RNA-seq experiments are no longer valid.

In this course we will discuss some of the questions that can be addressed using scRNA-seq as well as the available computational and statistical methods available. The course is taught through the University of Cambridge Bioinformatics training unit, but the material found on these pages is meant to be used for anyone interested in learning about computational analysis of scRNA-seq data. The course is taught twice per year and the material here is updated prior to each event.

The number of computational tools is increasing rapidly and we are doing our best to keep up to date with what is available. One of the main constraints for this course is that we would like to use tools that are implemented in R and that run reasonably fast. Moreover, we will also confess to being somewhat biased towards methods that have been developed either by us or by our friends and colleagues.

### 1.1 Video

This video was recorded in November 2017, at that time the course contained less chapters than the current version.

### 1.2 Registration

Please follow this link and register for the “**Analysis of single cell RNA-seq data**” course:  
<http://training.csx.cam.ac.uk/bioinformatics/search>

### 1.3 GitHub

<https://github.com/hemberg-lab/scRNA.seq.course>

## 1.4 Docker image (RStudio)

The course can be reproduced without any package installation by running the course docker RStudio image which contains all the required packages.

Make sure Docker is installed on your system. If not, please follow these instructions. To run the course RStudio docker image:

```
docker run -d -p 8787:8787 quay.io/hemberg-group/scrna-seq-course-rstudio
```

This download the docker image (may take some time) and start a new Rstudio session in a docker container with all packages installed and all data files available.

Then visit `localhost:8787` in your browser and log in with `username:password` as `rstudio:rstudio`. Now you are ready to go!

More details on how to run RStudio docker with different options can be found [here](#).

## 1.5 Manual installation

If you are not using a docker image of the course, then to be able to run all code chunks of the course you need to clone or download the course GitHub repository and start an R session in the cloned folder. You will also need to install all packages listed in the course docker files: Dockerfile1 and Dockerfile2.

Alternatively, you can just install packages listed in a chapter of interest.

## 1.6 License

All of the course material is licensed under GPL-3. Anyone is welcome to go through the material in order to learn about analysis of scRNA-seq data. If you plan to use the material for your own teaching, we would appreciate if you tell us about it in addition to providing a suitable citation.

## 1.7 Prerequisites

The course is intended for those who have basic familiarity with Unix and the R scripting language.

We will also assume that you are familiar with mapping and analysing bulk RNA-seq data as well as with the commonly available computational tools.

We recommend attending the Introduction to RNA-seq and ChIP-seq data analysis or the Analysis of high-throughput sequencing data with Bioconductor before attending this course.

## 1.8 Contact

If you have any **comments**, **questions** or **suggestions** about the material, please contact Vladimir Kiselev.

# Chapter 2

## Introduction to single-cell RNA-seq

### 2.1 Bulk RNA-seq

- A major breakthrough (replaced microarrays) in the late 00's and has been widely used since
- Measures the **average expression level** for each gene across a large population of input cells
- Useful for comparative transcriptomics, e.g. samples of the same tissue from different species
- Useful for quantifying expression signatures from ensembles, e.g. in disease studies
- **Insufficient** for studying heterogeneous systems, e.g. early development studies, complex tissues (brain)
- Does **not** provide insights into the stochastic nature of gene expression

### 2.2 scRNA-seq

- A **new** technology, first publication by (Tang et al., 2009)
- Did not gain widespread popularity until ~2014 when new protocols and lower sequencing costs made it more accessible
- Measures the **distribution of expression levels** for each gene across a population of cells
- Allows to study new biological questions in which **cell-specific changes in transcriptome are important**, e.g. cell type identification, heterogeneity of cell responses, stochasticity of gene expression, inference of gene regulatory networks across the cells.
- Datasets range **from  $10^2$  to  $10^6$  cells** and increase in size every year
- Currently there are several different protocols in use, e.g. SMART-seq2 (Picelli et al., 2013), CELL-seq (Hashimshony et al., 2012) and Drop-seq (Macosko et al., 2015)
- There are also commercial platforms available, including the Fluidigm C1, Wafergen ICELL8 and the 10X Genomics Chromium
- Several computational analysis methods from bulk RNA-seq **can** be used
- **In most cases** computational analysis requires adaptation of the existing methods or development of new ones

### 2.3 Workflow

Overall, experimental scRNA-seq protocols are similar to the methods used for bulk RNA-seq. We will be discussing some of the most common approaches in the next chapter.

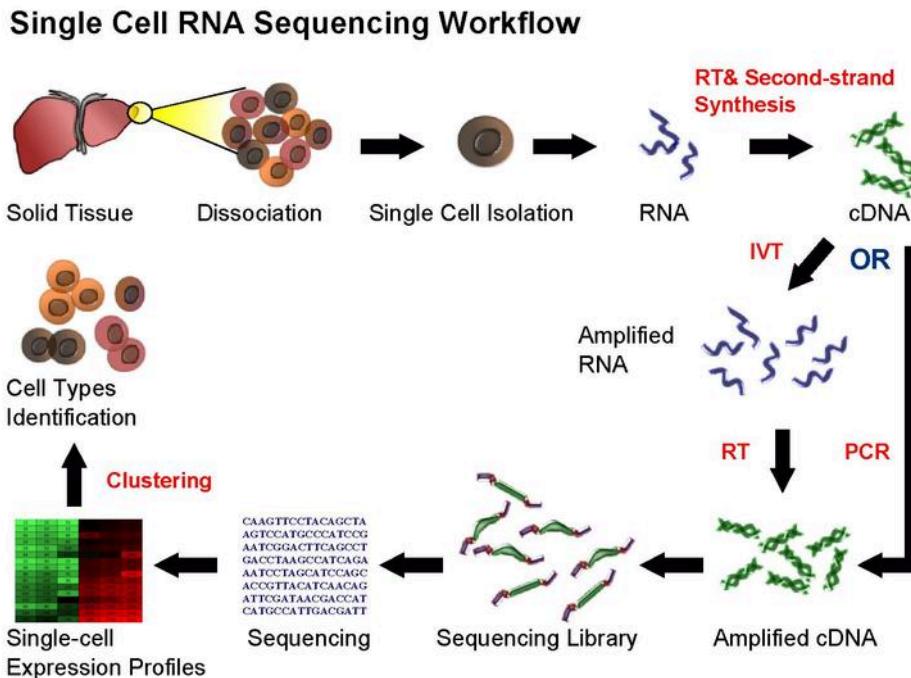


Figure 2.1: Single cell sequencing (taken from Wikipedia)

## 2.4 Computational Analysis

This course is concerned with the computational analysis of the data obtained from scRNA-seq experiments. The first steps (yellow) are general for any highthroughput sequencing data. Later steps (orange) require a mix of existing RNASeq analysis methods and novel methods to address the technical difference of scRNASeq. Finally the biological interpretation **should** be analyzed with methods specifically developed for scRNASeq.

There are several reviews of the scRNA-seq analysis available including (Stegle et al., 2015).

Today, there are also several different platforms available for carrying out one or more steps in the flowchart above. These include:

- Falco a single-cell RNA-seq processing framework on the cloud.
- SCONE (Single-Cell Overview of Normalized Expression), a package for single-cell RNA-seq data quality control and normalization.
- Seurat is an R package designed for QC, analysis, and exploration of single cell RNA-seq data.
- ASAP (Automated Single-cell Analysis Pipeline) is an interactive web-based platform for single-cell analysis.

## 2.5 Challenges

The main difference between bulk and single cell RNA-seq is that each sequencing library represents a single cell, instead of a population of cells. Therefore, significant attention has to be paid to comparison of the results from different cells (sequencing libraries). The main sources of discrepancy between the libraries are:

- **Amplification** (up to 1 million fold)
- **Gene ‘dropouts’** in which a gene is observed at a moderate expression level in one cell but is not detected in another cell (Kharchenko et al., 2014).

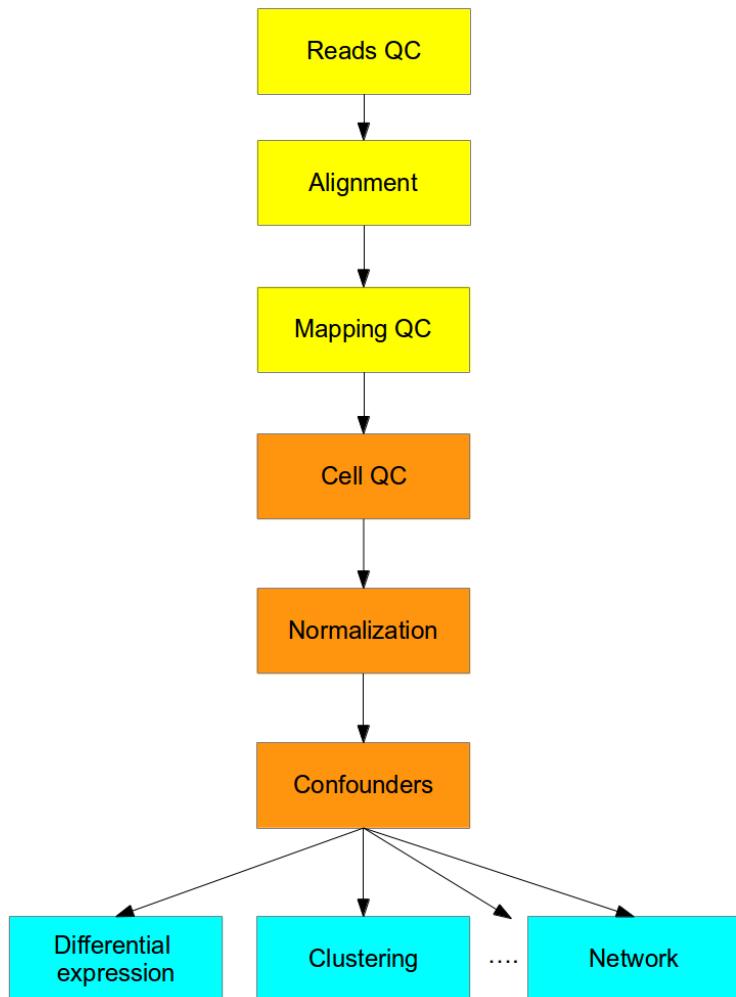


Figure 2.2: Flowchart of the scRNA-seq analysis

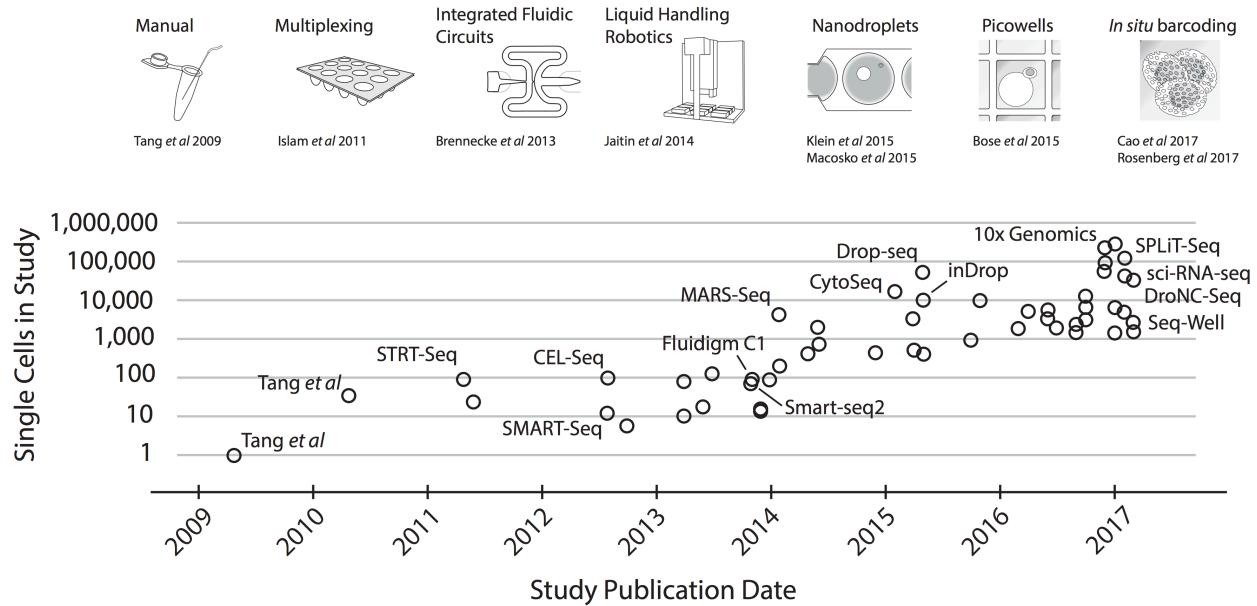


Figure 2.3: Moore's law in single cell transcriptomics (image taken from [Svensson et al](<https://arxiv.org/abs/1704.01379>))

In both cases the discrepancies are introduced due to low starting amounts of transcripts since the RNA comes from one cell only. Improving the transcript capture efficiency and reducing the amplification bias are currently active areas of research. However, as we shall see in this course, it is possible to alleviate some of these issues through proper normalization and corrections.

## 2.6 Experimental methods

Development of new methods and protocols for scRNA-seq is currently a very active area of research, and several protocols have been published over the last few years. A non-comprehensive list includes:

- CEL-seq (Hashimshony et al., 2012)
- CEL-seq2 (Hashimshony et al., 2016)
- Drop-seq (Macosko et al., 2015)
- InDrop-seq (Klein et al., 2015)
- MARS-seq (Jaitin et al., 2014)
- SCRB-seq (Soumillon et al., 2014)
- Seq-well (Gierahn et al., 2017)
- Smart-seq (Picelli et al., 2014)
- Smart-seq2 (Picelli et al., 2014)
- SMARTer
- STRT-seq (Islam et al., 2013)

The methods can be categorized in different ways, but the two most important aspects are **quantification** and **capture**.

For quantification, there are two types, **full-length** and **tag-based**. The former tries to achieve a uniform read coverage of each transcript. By contrast, tag-based protocols only capture either the 5'- or 3'-end of each RNA. The choice of quantification method has important implications for what types of analyses the data can be used for. In theory, full-length protocols should provide an even coverage of transcripts, but as we shall see, there are often biases in the coverage. The main advantage of tag-based protocol is that they

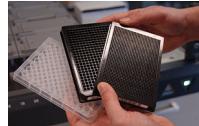


Figure 2.4: Image of microwell plates (image taken from Wikipedia)

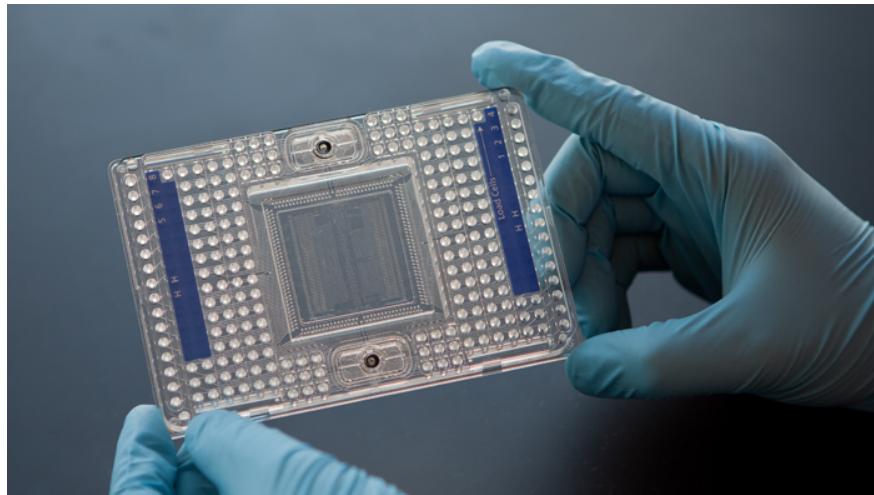


Figure 2.5: Image of a 96-well Fluidigm C1 chip (image taken from Fluidigm)

can be combined with unique molecular identifiers (UMIs) which can help improve the quantification (see chapter 4.6). On the other hand, being restricted to one end of the transcript may reduce the mappability and it also makes it harder to distinguish different isoforms (Archer et al., 2016).

The strategy used for capture determines throughput, how the cells can be selected as well as what kind of additional information besides the sequencing that can be obtained. The three most widely used options are **microwell-**, **microfluidic-** and **droplet-** based.

For well-based platforms, cells are isolated using for example pipette or laser capture and placed in microfluidic wells. One advantage of well-based methods is that they can be combined with fluorescent activated cell sorting (FACS), making it possible to select cells based on surface markers. This strategy is thus very useful for situations when one wants to isolate a specific subset of cells for sequencing. Another advantage is that one can take pictures of the cells. The image provides an additional modality and a particularly useful application is to identify wells containing damaged cells or doublets. The main drawback of these methods is that they are often low-throughput and the amount of work required per cell may be considerable.

Microfluidic platforms, such as Fluidigm's C1, provide a more integrated system for capturing cells and for carrying out the reactions necessary for the library preparations. Thus, they provide a higher throughput than microwell based platforms. Typically, only around 10% of cells are captured in a microfluidic platform and thus they are not appropriate if one is dealing with rare cell-types or very small amounts of input. Moreover, the chip is relatively expensive, but since reactions can be carried out in a smaller volume money can be saved on reagents.

The idea behind droplet based methods is to encapsulate each individual cell inside a nanoliter droplet together with a bead. The bead is loaded with the enzymes required to construct the library. In particular, each bead contains a unique barcode which is attached to all of the reads originating from that cell. Thus, all of the droplets can be pooled, sequenced together and the reads can subsequently be assigned to the cell of origin based on the barcodes. Droplet platforms typically have the highest throughput since the library preparation costs are on the order of .05 USD/cell. Instead, sequencing costs often become the limiting factor and a typical experiment the coverage is low with only a few thousand different transcripts detected

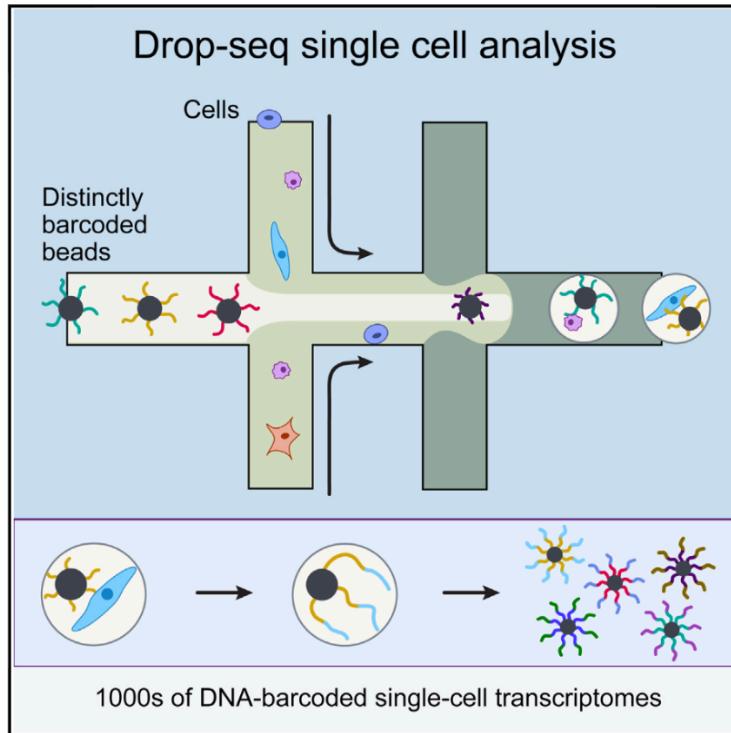


Figure 2.6: Schematic overview of the drop-seq method (Image taken from Macosko et al)

(Ziegenhain et al., 2017).

## 2.7 What platform to use for my experiment?

The most suitable platform depends on the biological question at hand. For example, if one is interested in characterizing the composition of a tissue, then a droplet-based method which will allow a very large number of cells to be captured is likely to be the most appropriate. On the other hand, if one is interesting in characterizing a rare cell-population for which there is a known surface marker, then it is probably best to enrich using FACS and then sequence a smaller number of cells.

Clearly, full-length transcript quantification will be more appropriate if one is interested in studying different isoforms since tagged protocols are much more limited. By contrast, UMIs can only be used with tagged protocols and they can facilitate gene-level quantification.

Two recent studies from the Enard group (Ziegenhain et al., 2017) and the Teichmann group (Svensson et al., 2017) have compared several different protocols. In their study, Ziegenhain et al compared five different protocols on the same sample of mouse embryonic stem cells (mESCs). By controlling for the number of cells as well as the sequencing depth, the authors were able to directly compare the sensitivity, noise-levels and costs of the different protocols. One example of their conclusions is illustrated in the figure below which shows the number of genes detected (for a given detection threshold) for the different methods. As you can see, there is almost a two-fold difference between drop-seq and Smart-seq2, suggesting that the choice of protocol can have a major impact on the study.

Svensson et al take a different approach by using synthetic transcripts (spike-ins, more about these later) with known concentrations to measure the accuracy and sensitivity of different protocols. Comparing a wide range of studies, they also reported substantial differences between the protocols.

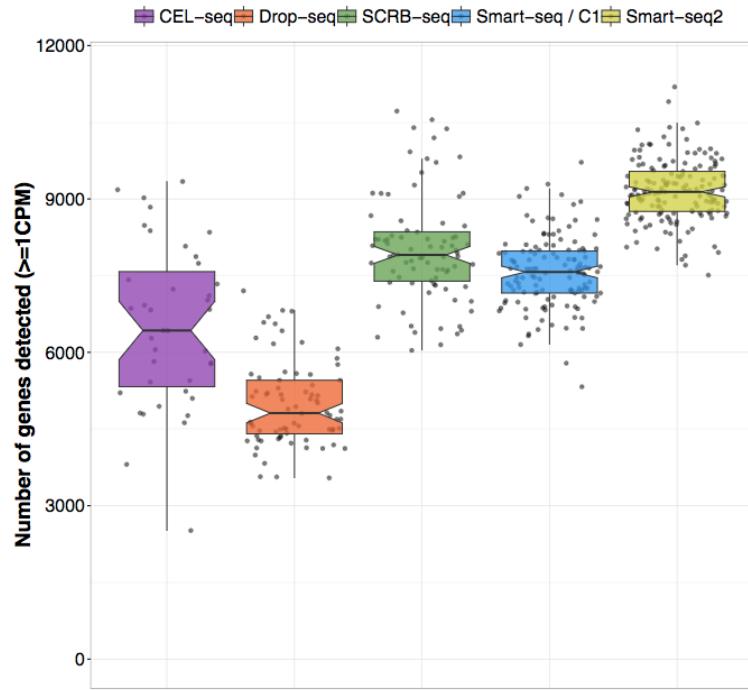


Figure 2.7: Enard group study

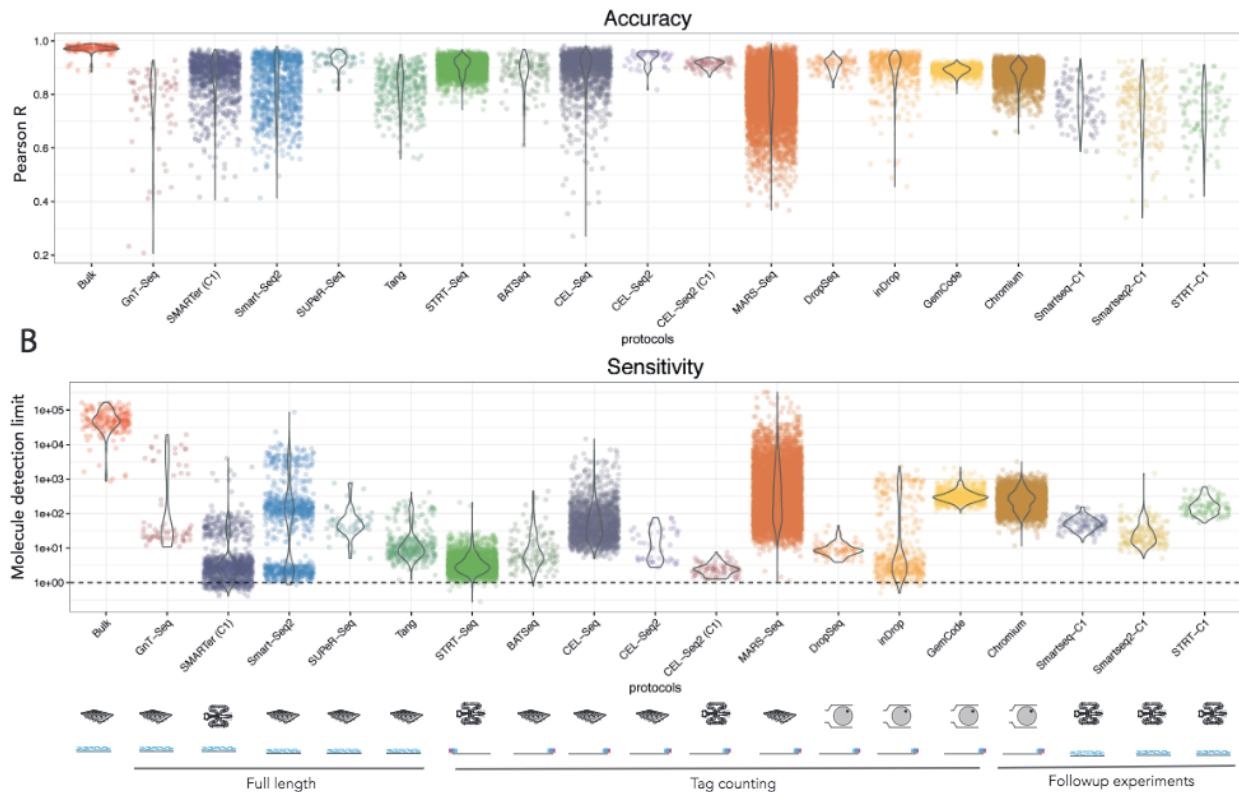


Figure 2.8: Teichmann group study

As protocols are developed and computational methods for quantifying the technical noise are improved, it is likely that future studies will help us gain further insights regarding the strengths of the different methods. These comparative studies are helpful not only for helping researchers decide which protocol to use, but also for developing new methods as the benchmarking makes it possible to determine what strategies are the most useful ones.

# Chapter 3

# Processing Raw scRNA-seq Data

## 3.1 FastQC

Once you've obtained your single-cell RNA-seq data, the first thing you need to do with it is check the quality of the reads you have sequenced. For this task, today we will be using a tool called FastQC. FastQC is a quality control tool for sequencing data, which can be used for both bulk and single-cell RNA-seq data. FastQC takes sequencing data as input and returns a report on read quality. Copy and paste this link into your browser to visit the FastQC website:

```
https://www.bioinformatics.babraham.ac.uk/projects/fastqc/
```

This website contains links to download and install FastQC and documentation on the reports produced. Fortunately we have already installed FastQC for you today, so instead we will take a look at the documentation. Scroll down the webpage to 'Example Reports' and click 'Good Illumina Data'. This gives an example of what an ideal report should look like for high quality Illumina reads data.

Now let's make a FastQC report ourselves.

Today we will be performing our analysis using a single cell from an mESC dataset produced by (Kolodziejczyk et al., 2015). The cells were sequenced using the SMART-seq2 library preparation protocol and the reads are paired end. The files are located in `Share`. Let's take a look at them:

```
less Share/ERR522959_1.fastq  
less Share/ERR522959_2.fastq
```

Task 1: Try to work out what command you should use to produce the FastQC report. Hint: Try executing `fastqc -h`

This command will tell you what options are available to pass to FastQC. Feel free to ask for help if you get stuck! If you are successful, you should generate a .zip and a .html file for both the forwards and the reverse reads files. Once you have been successful, feel free to have a go at the next section.

### 3.1.1 Solution and Downloading the Report

If you haven't done so already, generate the FastQC report using the commands below:

```
mkdir fastqc_results  
fastqc -o fastqc_results Share/ERR522959_1.fastq Share/ERR522959_2.fastq
```

Once the command has finished executing, you should have a total of four files - one zip file for each of the paired end reads, and one html file for each of the paired end reads. The report is in the html file. To view it, we will need to get it off AWS and onto your computer using either filezilla or scp. Ask an instructor if you are having difficulties.

Once the file is on your computer, click on it. Your FastQC report should open. Have a look through the file. Remember to look at both the forwards and the reverse end read reports! How good quality are the reads? Is there anything we should be concerned about? How might we address those concerns?

Feel free to chat to one of the instructors about your ideas.

## 3.2 Trimming Reads

Fortunately there is software available for read trimming. Today we will be using Trim Galore!. Trim Galore! is a wrapper for the reads trimming software cutadapt.

Read trimming software can be used to trim sequencing adapters and/or low quality reads from the ends of reads. Given we noticed there was some adaptor contamination in our FastQC report, it is a good idea to trim adaptors from our data.

Task 2: What type of adapters were used in our data? Hint: Look at the FastQC report ‘Adapter Content’ plot.

Now let’s try to use Trim Galore! to remove those problematic adapters. It’s a good idea to check read quality again after trimming, so after you have trimmed your reads you should use FastQC to produce another report.

Task 3: Work out the command you should use to trim the adapters from our data. Hint 1: You can use `trim_galore -h`

To find out what options you can pass to Trim Galore. Hint 2: Read through the output of the above command carefully. The adaptor used in this experiment is quite common. Do you need to know the actual sequence of the adaptor to remove it?

Task 3: Produce a FastQC report for your trimmed reads files. Is the adapter contamination gone?

Once you think you have successfully trimmed your reads and have confirmed this by checking the FastQC report, feel free to check your results using the next section.

### 3.2.1 Solution

You can use the command(s) below to trim the Nextera sequencing adapters:

```
mkdir fastqc_trimmed_results
trim_galore --nextera -o fastqc_trimmed_results Share/ERR522959_1.fastq Share/ERR522959_2.fastq
```

Remember to generate new FastQC reports for your trimmed reads files! FastQC should now show that your reads pass the ‘Adaptor Content’ plot. Feel free to ask one of the instructors if you have any questions.

Congratulations! You have now generated reads quality reports and performed adaptor trimming. In the next lab, we will use STAR and Kallisto to align our trimmed and quality-checked reads to a reference transcriptome.

### 3.3 File formats

#### 3.3.1 FastQ

FastQ is the most raw form of scRNASeq data you will encounter. All scRNASeq protocols are sequenced with paired-end sequencing. Barcode sequences may occur in one or both reads depending on the protocol employed. However, protocols using unique molecular identifiers (UMIs) will generally contain one read with the cell and UMI barcodes plus adapters but without any transcript sequence. Thus reads will be mapped as if they are single-end sequenced despite actually being paired end.

FastQ files have the format:

```
>ReadID
READ SEQUENCE
+
SEQUENCING QUALITY SCORES
```

#### 3.3.2 BAM

BAM file format stores mapped reads in a standard and efficient manner. The human-readable version is called a SAM file, while the BAM file is the highly compressed version. BAM/SAM files contain a header which typically includes

information on the sample preparation, sequencing and mapping; and a tab-separated row for each individual alignment of each read.

Alignment rows employ a standard format with the following columns:

- (1) QNAME : read name (generally will include UMI barcode if applicable)
- (2) FLAG : number tag indicating the “type” of alignment, link to explanation of all possible “types”
- (3) RNAME : reference sequence name (i.e. chromosome read is mapped to).
- (4) POS : leftmost mapping position
- (5) MAPQ : Mapping quality
- (6) CIGAR : string indicating the matching/mismatching parts of the read (may include soft-clipping).
- (7) RNEXT : reference name of the mate/next read
- (8) PNEXT : POS for mate/next read
- (9) TLEN : Template length (length of reference region the read is mapped to)
- (10) SEQ : read sequence
- (11) QUAL : read quality

BAM/SAM files can be converted to the other format using ‘samtools’:

```
 samtools view -S -b file.sam > file.bam
 samtools view -h file.bam > file.sam
```

Some sequencing facilities will automatically map your reads to the a standard genome and deliver either BAM or CRAM formatted files. Generally they will not have included ERCC sequences in the genome thus no ERCC reads will be mapped in the BAM/CRAM file. To quantify ERCCs (or any other genetic alterations) or if you just want to use a different alignment algorithm than whatever is in the generic pipeline (often outdated), then you will need to convert the BAM/CRAM files back to FastQs:

BAM files can be converted to FastQ using bedtools. To ensure a single copy for multi-mapping reads first sort by read name and remove secondary alignments using samtools. Picard also contains a method for converting BAM to FastQ files.

```
# sort reads by name
samtools sort -n original.bam -o sorted_by_name.bam
# remove secondary alignments
samtools view -b -F 256 sorted_by_name.bam -o primary_alignment_only.bam
# convert to fastq
bedtools bamtofastq -i primary_alignment_only.bam -fq read1.fq -fq2 read2.fq
```

### 3.3.3 CRAM

CRAM files are similar to BAM files only they contain information in the header to the reference genome used in the mapping in the header. This allow the bases in each read that are identical to the reference to be further compressed. CRAM also supports some lossy data compression approaches to further optimize storage compared to BAMs. CRAMs are mainly used by the Sanger/EBI sequencing facility.

CRAM and BAM files can be interchanged using the lastest version of samtools (>=v1.0). However, this conversion may require downloading the reference genome into cache. Alternatively, you may pre-download the correct reference either from metadata in the header of the CRAM file, or from talking to whomever generated the CRAM and specify that file using ‘-T’ Thus we recommend setting a specific cache location prior to doing this:

```
export REF_CACHE=/path_to/cache_directory_for_reference_genome
samtools view -b -h -T reference_genome.fasta file.cram -o file.bam
samtools view -C -h -T reference_genome.fasta file.bam -o file.cram
```

### 3.3.4 Manually Inspecting files

At times it may be useful to mannnual inspect files for example to check the metadata in headers that the files are from the correct sample. ‘less’ and ‘more’ can be used to inspect any text files from the command line. By “pipe-ing” the output of samtools view into these commands using ‘|’ we check each of these file types without having to save multiple copies of each file.

```
less file.txt
more file.txt
# counts the number of lines in file.txt
wc -l file.txt
samtools view -h file.[cram/bam] | more
# counts the number of lines in the samtools output
samtools view -h file.[cram/bam] | wc -l
```

### Exercises

You have been provided with a small cram file: EXAMPLE.cram

Task 1: How was this file aligned? What software was used? What was used as the genome? (Hint: check the header)

Task 2: How many reads are unmapped/mapped? How total reads are there? How many secondary alignments are present? (Hint: use the FLAG)

Task 3: Convert the CRAM into two Fastq files. Did you get exactly one copy of each read? (name these files “10cells\_read1.fastq” “10cells\_read2.fastq”)

If you get stuck help information for each piece of software can be displayed by entering running the command “naked” - e.g. ‘samtools view’, ‘bedtools’

### Answer

#### 3.3.5 Genome (FASTA, GTF)

To map your reads you will also need the reference genome and in many cases the genome annotation file (in either GTF or GFF format). These can be downloaded for model organisms from any of the main genomics databases: Ensembl, NCBI, or UCSC Genome Browser.

GTF files contain annotations of genes, transcripts, and exons. They must contain: (1) seqname : chromosome/scaffold (2) source : where this annotation came from (3) feature : what kind of feature is this? (e.g. gene, transcript, exon) (4) start : start position (bp) (5) end : end position (bp) (6) score : a number (7) strand : + (forward) or - (reverse) (8) frame : if CDS indicates which base is the first base of the first codon (0 = first base, 1 = second base, etc..) (9) attribute : semicolon-separated list of tag-value pairs of extra information (e.g. names/IDs, biotype)

Empty fields are marked with “.”

In our experience Ensembl is the easiest of these to use, and has the largest set of annotations. NCBI tends to be more strict in including only high confidence gene annotations. Whereas UCSC contains multiple geneset annotations that use different criteria.

If your experimental system includes non-standard sequences these must be added to both the genome fasta and gtf to quantify their expression. Most commonly this is done for the ERCC spike-ins, although the same must be done for CRISPR-related sequences or other overexpression/reporter constructs.

For maximum utility/flexibility we recommend creating complete and detailed entries for any non-standard sequences added.

There is no standardized way to do this. So below is our custom perl script for creating a gtf and fasta file for ERCCs which can be appended to the genome. You may also need to alter a gtf file to deal with repetitive elements in introns when/if you want to quantify intronic reads. Any scripting language or even ‘awk’ and/or some text editors can be used to do this relatively efficiently, but they are beyond the scope of this course.

```
# Converts the Annotation file from
# https://www.thermofisher.com/order/catalog/product/4456740 to
# gtf and fasta files that can be added to existing genome fasta & gtf files.

my @FASTALines = ();
my @GTFLines = ();
open (my $ifh, "ERCC_Controls_Annotation.txt") or die $!;
<$ifh>; #header
while (<$ifh>) {
    # Do all the important stuff
    chomp;
    my @record = split(/\t/);
    my $sequence = $record[4];
    $sequence =~ s/\s+//g; # get rid of any preceding/tailing white space
    $sequence = $sequence."NNNN";
    my $name = $record[0];
    my $genbank = $record[1];
    push(@FASTALines, ">$name\n$sequence\n");
# is GTF 1 indexed or 0 indexed? -> it is 1 indexed
# + or - strand?
```

```

push(@GTFlines, "$name\tERCC\tgene\t1\t".(length($sequence)-2)."t.\t+\t.\tgene_id \"$name-$genbank"
push(@GTFlines, "$name\tERCC\ttranscript\t1\t".(length($sequence)-2)."t.\t+\t.\tgene_id \"$name-$g
push(@GTFlines, "$name\tERCC\texon\t1\t".(length($sequence)-2)."t.\t+\t.\tgene_id \"$name-$genbank
} close($ifh);

# Write output
open(my $ofh, ">", "ERCC_Controls.fa") or die $!;
foreach my $line (@FASTALines) {
    print $ofh $line;
} close ($ofh);

open($ofh, ">", "ERCC_Controls.gtf") or die $!;
foreach my $line (@GTFlines) {
    print $ofh $line;
} close ($ofh);

```

## 3.4 Demultiplexing

Demultiplexing is done differently depending on the protocol used and the particular pipeline you are using a full pipeline. The most flexible demultiplexing pipeline we are aware of is zUMIs which can be used to demultiplex and map most UMI-based protocols. For Smartseq2 or other paired-end full transcript protocols the data will usually already be demultiplexed. Public repositories such as GEO or ArrayExpress require data small-scale/plate-based scRNASeq data to be demultiplexed prior to upload, and many sequencing facilities will automatically demultiplex data before returning it to you. If you aren't using a published pipeline and the data was not demultiplexed by the sequencing facility you will have to demultiplex it yourself. This usually requires writing a custom script since barcodes may be of different lengths and different locations in the reads depending on the protocols used.

For all data-type “demultiplexing” involves identifying and removing the cell-barcode sequence from one or both reads. If the expected cell-barcodes are known ahead of time, i.e. the data is from a PCR-plate-based protocol, all that is necessarily is to compare each cell-barcode to the expected barcodes and assign the associated reads to the closest cell-barcode (with maximum mismatches of 1 or 2 depending on the design of the cell-barcodes). These data are generally demultiplexed prior to mapping as an easy way of parallelizing the mapping step.

We have publicly available perl scripts capable of demultiplexing any scRNASeq data with a single cell-barcode with or without UMIs for plate-based protocols. These can be used as so:

```

perl 1_Flexible_UMI_Demultiplexing.pl 10cells_read1.fq 10cells_read2.fq "C12U8" 10cells_barcodes.txt 2

##
## Doesn't match any cell: 0
## Ambiguous: 0
## Exact Matches: 400
## Contain mismatches: 0
## Input Reads: 400
## Output Reads: 400
## Barcode Structure: 12 bp CellID followed by 8 bp UMI

perl 1_Flexible_FullTranscript_Demultiplexing.pl 10cells_read1.fq 10cells_read2.fq "start" 12 10cells_b

##
## Doesn't match any cell: 0
## Ambiguous: 0

```

```
## Exact Matches: 400
## Contain Mismatches: 0
## Input Reads: 400
## Output Reads: 400
```

For UMI containing data, demultiplexing includes attaching the UMI code to the read name of the gene-body containing read. If the data are from a droplet-based protocol or SeqWell where the number of expected barcodes is much higher than the expected number of cell, then usually the cell-barcode will also be attached to the read name to avoid generating a very large number of files. In these cases, demultiplexing will happen during the quantification step to facilitate the identification of cell-barcodes which correspond to intact cells rather than background noise.

### 3.4.1 Identifying cell-containing droplets/microwells

For droplet based methods only a fraction of droplets contain both beads and an intact cell. However, biology experiments are messy and some RNA will leak out of dead/damaged cells. So droplets without an intact cell are likely to capture a small amount of the ambient RNA which will end up in the sequencing library and contribute reads to the final sequencing output. The variation in droplet size, amplification efficiency, and sequencing will lead both “background” and real cells to have a wide range of library sizes. Various approaches have been used to try to distinguish those cell barcodes which correspond to real cells.

Most methods use the total molecules (could be applied to total reads) per barcode and try to find a “break point” between bigger libraries which are cells + some background and smaller libraries assumed to be purely background. Let’s load some example simulated data which contain both large and small cells:

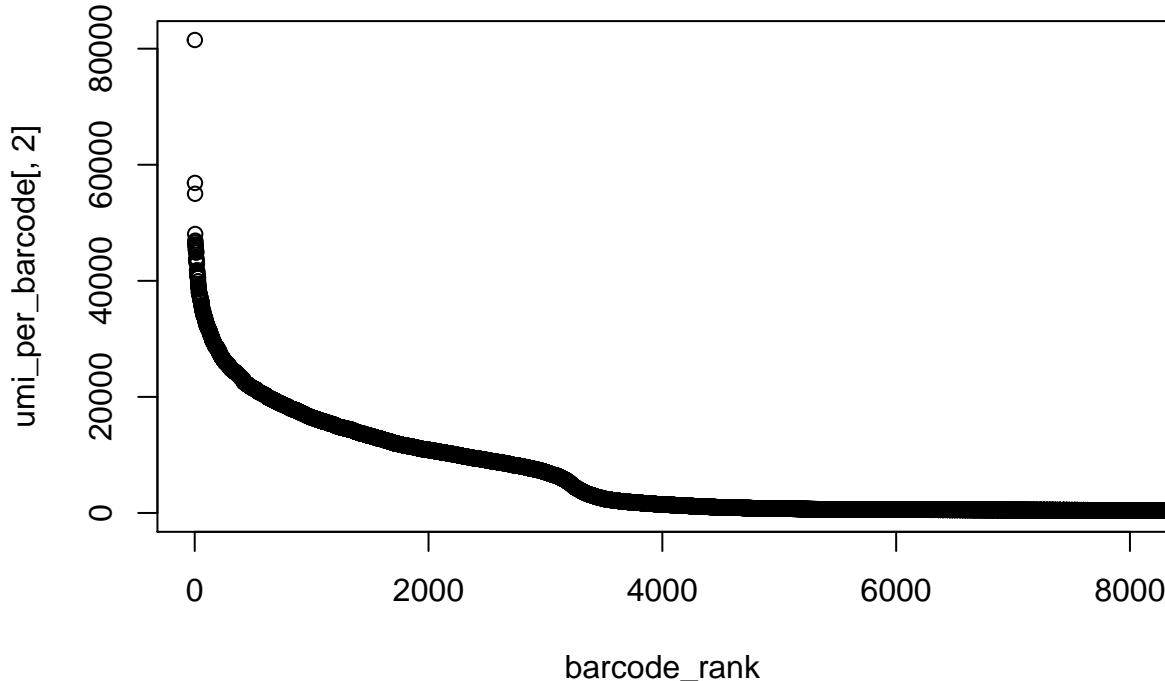
```
umi_per_barcode <- read.table("droplet_id_example_per_barcode.txt.gz")
truth <- read.delim("droplet_id_example_truth.gz", sep=",")
```

**Exercise** How many unique barcodes were detected? How many true cells are present in the data? To simplify calculations for this section exclude all barcodes with fewer than 10 total molecules.

#### Answer

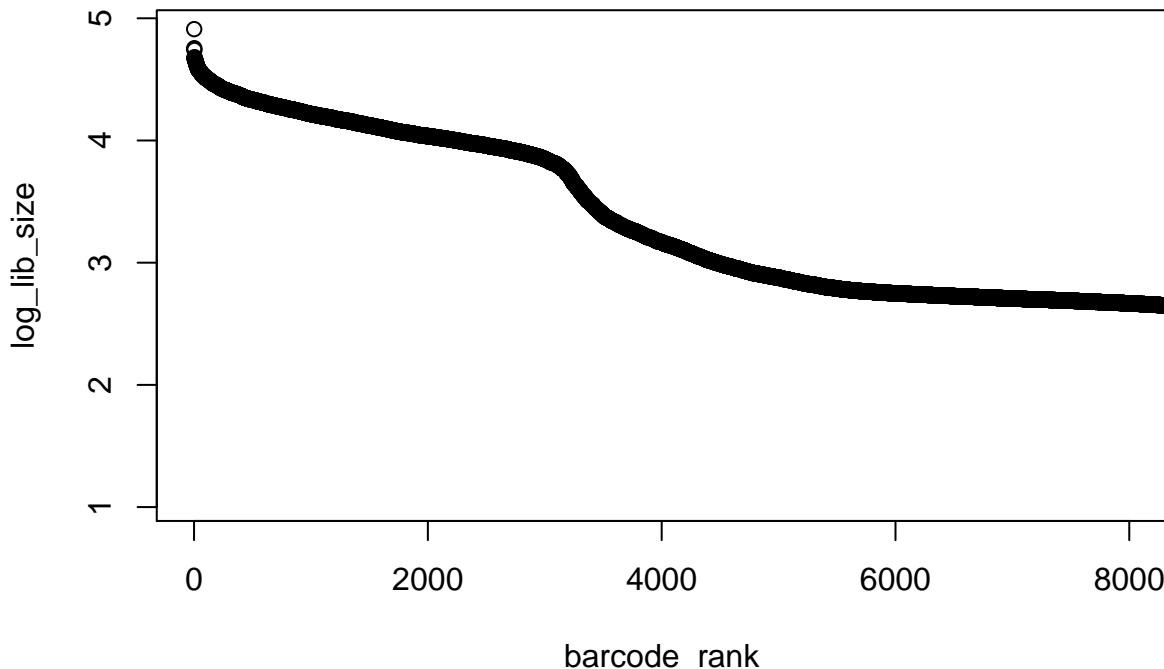
One approach is to look for the inflection point where the total molecules per barcode suddenly drops:

```
barcode_rank <- rank(-umi_per_barcode[,2])
plot(barcode_rank, umi_per_barcode[,2], xlim=c(1,8000))
```



Here we can see an roughly exponential curve of library sizes, so to make things simpler lets log-transform them.

```
log_lib_size <- log10(umi_per_barcode[, 2])
plot(barcode_rank, log_lib_size, xlim=c(1, 8000))
```



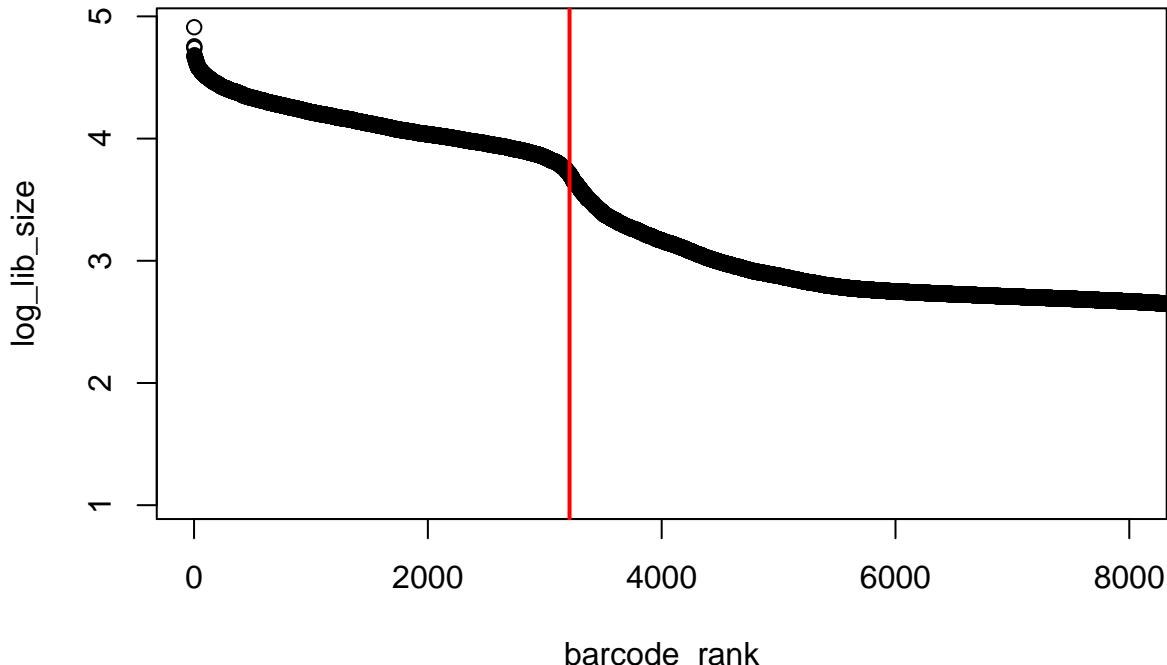
That's better, the “knee” in the distribution is much more pronounced. We could manually estimate where the “knee” is but it much more reproducible to algorithmically identify this point.

```
# inflection point
o <- order(barcode_rank)
log_lib_size <- log_lib_size[o]
```

```
barcode_rank <- barcode_rank[o]

rawdiff <- diff(log_lib_size)/diff(barcode_rank)
inflection <- which(rawdiff == min(rawdiff[100:length(rawdiff)]), na.rm=TRUE))

plot(barcode_rank, log_lib_size, xlim=c(1,8000))
abline(v=inflection, col="red", lwd=2)
```



```
threshold <- 10^log_lib_size[inflection]

cells <- umi_per_barcode[umi_per_barcode[,2] > threshold,1]
TPR <- sum(cells %in% truth[,1])/length(cells)
Recall <- sum(cells %in% truth[,1])/length(truth[,1])
c(TPR, Recall)

## [1] 1.0000000 0.7831707
```

Another is to fix a mixture model and find where the higher and lower distributions intersect. However, data may not fit the assumed distributions very well:

```
set.seed(-92497)
# mixture model
require("mixtools")

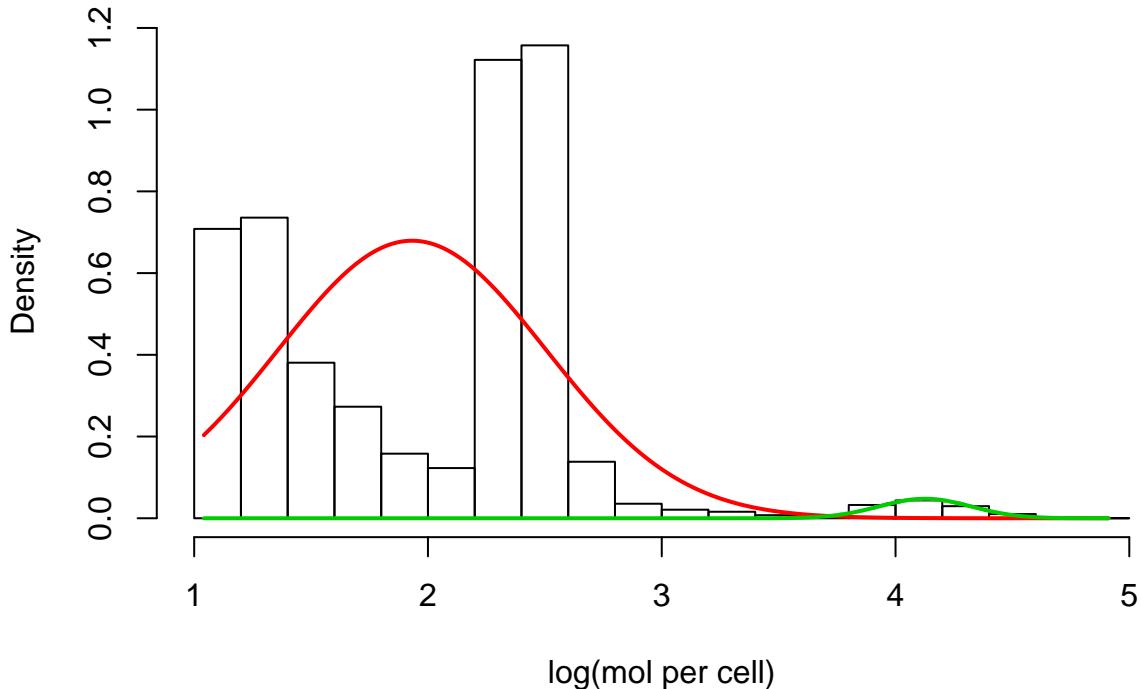
## Loading required package: mixtools

## mixtools package, version 1.1.0, Released 2017-03-10
## This package is based upon work supported by the National Science Foundation under Grant No. SES-0512546

mix <- normalmixEM(log_lib_size)

## number of iterations= 43
plot(mix, which=2, xlab2="log(mol per cell)")
```

## Density Curves



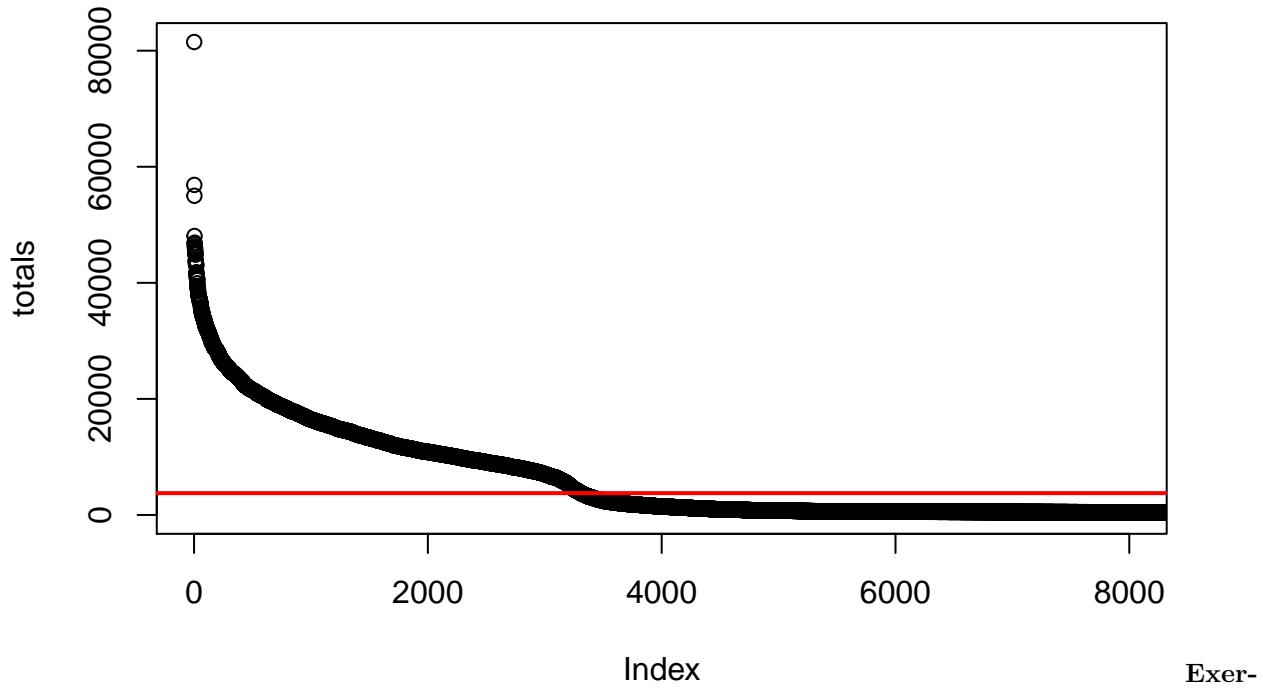
```
p1 <- dnorm(log_lib_size, mean=mix$mu[1], sd=mix$sigma[1])
p2 <- dnorm(log_lib_size, mean=mix$mu[2], sd=mix$sigma[2])
if (mix$mu[1] < mix$mu[2]) {
  split <- min(log_lib_size[p2 > p1])
} else {
  split <- min(log_lib_size[p1 > p2])
}
```

**Exercise** Identify cells using this split point and calculate the TPR and Recall.

### Answer

A third, used by CellRanger, assumes a ~10-fold range of library sizes for real cells and estimates this range using the expected number of cells.

```
n_cells <- length(truth[,1])
# CellRanger
totals <- umi_per_barcode[,2]
totals <- sort(totals, decreasing = TRUE)
# 99th percentile of top n_cells divided by 10
thresh = totals[round(0.01*n_cells)]/10
plot(totals, xlim=c(1,8000))
abline(h=thresh, col="red", lwd=2)
```



cise Identify cells using this threshodl and calculate the TPR and Recall.

### Answer

Finally (EmptyDrops)[<https://github.com/MarioniLab/DropletUtils>], which is currently in beta testing, uses the full genes x cells molecule count matrix for all droplets and estimates the profile of “background” RNA from those droplets with extremely low counts, then looks for cells with gene-expression profiles which differ from the background. This is combined with an inflection point method since background RNA often looks very similar to the expression profile of the largests cells in a population. As such EmptyDrops is the only method able to identify barcodes for very small cells in highly diverse samples.

Below we have provided code for how this method is currently run: (We will update this page when the method is officially released)

```

require("Matrix")
raw.counts <- readRDS("droplet_id_example.rds")

require("DropletUtils")
# emptyDrops
set.seed(100)
e.out <- emptyDrops(my.counts)
is.cell <- e.out$FDR <= 0.01
sum(is.cell, na.rm=TRUE)
plot(e.out$Total, -e.out$LogProb, col=ifelse(is.cell, "red", "black"),
     xlab="Total UMI count", ylab="-Log Probability")

cells <- colnames(raw.counts)[is.cell]

TPR <- sum(cells %in% truth[,1])/length(cells)
Recall <- sum(cells %in% truth[,1])/length(truth[,1])
c(TPR, Recall)

```

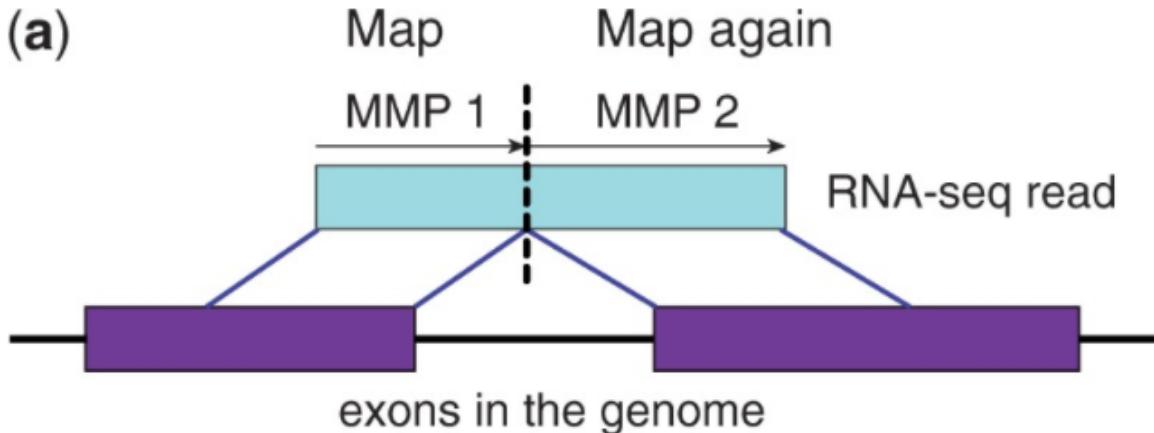
**Fig. 1.**

Figure 3.1: Figure 1: Diagram of how STAR performs alignments, taken from Dobin et al.

### 3.5 Using STAR to Align Reads

Now we have trimmed our reads and established that they are of good quality, we would like to map them to a reference genome. This process is known as alignment. Some form of alignment is generally required if we want to quantify gene expression or find genes which are differentially expressed between samples.

Many tools have been developed for read alignment, but today we will focus on two. The first tool we will consider is STAR (?). For each read in our reads data, STAR tries to find the longest possible sequence which matches one or more sequences in the reference genome. For example, in the figure below, we have a read (blue) which spans two exons and an alternative splicing junction (purple). STAR finds that the first part of the read is the same as the sequence of the first exon, whilst the second part of the read matches the sequence in the second exon. Because STAR is able to recognise splicing events in this way, it is described as a ‘splice aware’ aligner.

Usually STAR aligns reads to a reference genome, potentially allowing it to detect novel splicing events or chromosomal rearrangements. However, one issue with STAR is that it needs a lot of RAM, especially if your reference genome is large (eg. mouse and human). To speed up our analysis today, we will use STAR to align reads from to a reference transcriptome of 2000 transcripts. Note that this is NOT normal or recommended practice, we only do it here for reasons of time. We recommend that normally you should align to a reference genome.

Two steps are required to perform STAR alignment. In the first step, the user provides STAR with reference genome sequences (FASTA) and annotations (GTF), which STAR uses to create a genome index. In the second step, STAR maps the user’s reads data to the genome index.

Let’s create the index now. Remember, for reasons of time we are aligning to a transcriptome rather than a genome today, meaning we only need to provide STAR with the sequences of the transcripts we will be aligning reads to. You can obtain transcriptomes for many model organisms from Ensembl (<https://www.ensembl.org/info/data/ftp/index.html>).

Task 1: Execute the commands below to create the index:

```
mkdir indices
mkdir indices/STAR
STAR --runThreadN 4 --runMode genomeGenerate --genomeDir indices/STAR --genomeFastaFiles Share/2000_ref
```

Task 2: What does each of the options we used do? Hint: Use the STAR manual to help you (<https://github.com/alexdobin/STAR/blob/master/doc/STARmanual.pdf>)

Task 3: How would the command we used in Task 1 be different if we were aligning to the genome rather than the transcriptome?

Now that we have created the index, we can perform the mapping step.

Task 4: Try to work out what command you should use to map our trimmed reads (from ERR522959) to the index you created. Use the STAR manual to help you. Once you think you know the answer, check whether it matches the solution in the next section and execute the alignment.

Task 5: Try to understand the output of your alignment. Talk to one of the instructors if you need help!

### 3.5.1 Solution for STAR Alignment

You can use the following commands to perform the mapping step:

```
mkdir results
mkdir results/STAR
```

```
STAR --runThreadN 4 --genomeDir indices/STAR --readFilesIn Share/ERR522959_1.fastq Share/ERR522959_2.fas
```

## 3.6 Kallisto and Pseudo-Alignment

STAR is a reads aligner, whereas Kallisto is a pseudo-aligner (Bray et al., 2016). The main difference between aligners and pseudo-aligners is that whereas aligners map reads to a reference, pseudo-aligners map k-mers to a reference.

### 3.6.1 What is a k-mer?

A k-mer is a sequence of length k derived from a read. For example, imagine we have a read with the sequence ATCCCGGGTTAT and we want to make 7-mers from it. To do this, we would find the first 7-mer by counting the first seven bases of the read. We would find the second 7-mer by moving one base along, then counting the next seven bases. Figure 2 shows all the 7-mers that could be derived from our read:

### 3.6.2 Why map k-mers rather than reads?

There are two main reasons:

1. Pseudo-aligners use k-mers and a computational trick to make pseudo-alignment much faster than traditional aligners. If you are interested in how this is achieved, see (Bray et al., 2017) for details.
2. Under some circumstances, pseudo-aligners may be able to cope better with sequencing errors than traditional aligners. For example, imagine there was a sequencing error in the first base of the read above and the A was actually a T. This would impact on the pseudo-aligners ability to map the first 7-mer but none of the following 7-mers.

### 3.6.3 Kallisto's pseudo mode

Kallisto has a specially designed mode for pseudo-aligning reads from single-cell RNA-seq experiments. Unlike STAR, Kallisto pseudo-aligns to a reference transcriptome rather than a reference genome. This means

Read: ATCCCGGGTTAT

ATCCCGG

TCCCGGG

CCCGGGT

CCGGGTT

CGGGTTA

GGGTTAT

Figure 3.2: Figure 2: The 7-mers derived from an example read

Kallisto maps reads to splice isoforms rather than genes. Mapping reads to isoforms rather than genes is especially challenging for single-cell RNA-seq for the following reasons:

- Single-cell RNA-seq is lower coverage than bulk RNA-seq, meaning the total amount of information available from reads is reduced.
- Many single-cell RNA-seq protocols have 3' coverage bias, meaning if two isoforms differ only at their 5' end, it might not be possible to work out which isoform the read came from.
- Some single-cell RNA-seq protocols have short read lengths, which can also mean it is not possible to work out which isoform the read came from.

Kallisto's pseudo mode takes a slightly different approach to pseudo-alignment. Instead of aligning to isoforms, Kallisto aligns to equivalence classes. Essentially, this means if a read maps to multiple isoforms, Kallisto records the read as mapping to an equivalence class containing all the isoforms it maps to. Instead of using gene or isoform expression estimates in downstream analysis such as clustering, equivalence class counts can be used instead. Figure 3 shows a diagram which helps explain this.

Today we will just perform pseudo-alignment with one cell, but Kallisto is also capable of pseudo-aligning multiple cells simultaneously and using information from UMIs. See <https://pachterlab.github.io/kallisto/manual> for details.

As for STAR, you will need to produce an index for Kallisto before the pseudo-alignment step.

Task 6: Use the below command to produce the Kallisto index. Use the Kallisto manual (<https://pachterlab.github.io/kallisto/manual>) to work out what the options do in this command.

```
mkdir indices/Kallisto
kallisto index -i indices/Kallisto/transcripts.idx Share/2000_reference.transcripts.fa
```

Task 7: Use the Kallisto manual to work out what command to use to perform pseudo-alignment. Once you think you know the answer, check whether it matches the solution in the next section and execute the pseudo-alignment.

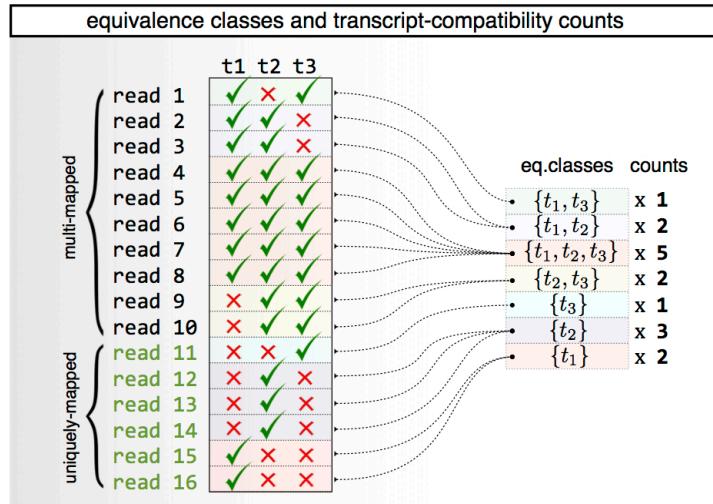


Figure 3.3: Figure 3: A diagram explaining Kallisto’s Equivalence Classes, taken from <https://pachterlab.github.io/kallisto/singlecell.html>.

### 3.6.4 Solution to Kallisto Pseudo-Alignment

Use the below command to perform pseudo-alignment

```
mkdir results/Kallisto
kallisto pseudo -i indices/Kallisto/transcripts.idx -o results/Kallisto -b batch.txt
```

See <https://pachterlab.github.io/kallisto/manual> for instructions on creating batch.txt, or ask an instructor if you get stuck.

### 3.6.5 Understanding the Output of Kallisto Pseudo-Alignment

The command above should produce 4 files - matrix.cells, matrix.ec, matrix.tsv and run\_info.json.

- matrix.cells contains a list of cell IDs. As we only used one cell, this file should just contain “ERR522959”
- matrix.ec contains information about the equivalence classes used. The first number in each row is the equivalence class ID. The second number(s) correspond to the transcript ID(s) in that equivalence class. For example “10 1,2,3” would mean that equivalence class 10 contains transcript IDs 1,2 and 3. The ID numbers correspond to the order that the transcripts appear in reference.transcripts.fa. Zero indexing is used, meaning transcript IDs 1,2 and 3 correspond to the second, third and fourth transcripts in 2000\_reference.transcripts.fa.
- matrix.tsv contains information about how many reads in each cell map to each equivalence class. The first number is the equivalence class ID, as defined in matrix.ec. The second number is the cell ID, where the cell ID corresponds to the order that the cell came in the matrix.cells file. The third number is the number of reads which fall into that equivalence class. For example, “5 1 3” means that 3 reads from cell 1 map to equivalence class 5. Note that zero indexing is used, so cell 1 corresponds to the second line of matrix.cells.
- run\_info.json contains information about how Kallisto was executed and can be ignored.



# Chapter 4

## Construction of expression matrix

Many analyses of scRNA-seq data take as their starting point an **expression matrix**. By convention, the each row of the expression matrix represents a gene and each column represents a cell (although some authors use the transpose). Each entry represents the expression level of a particular gene in a given cell. The units by which the expression is measured depends on the protocol and the normalization strategy used.

### 4.1 Reads QC

The output from a scRNA-seq experiment is a large collection of cDNA reads. The first step is to ensure that the reads are of high quality. The quality control can be performed by using standard tools, such as FastQC or Kraken.

Assuming that our reads are in experiment.bam, we run FastQC as

```
$<path_to_fastQC>/fastQC experiment.bam
```

Below is an example of the output from FastQC for a dataset of 125 bp reads. The plot reveals a technical error which resulted in a couple of bases failing to be read correctly in the centre of the read. However, since the rest of the read was of high quality this error will most likely have a negligible effect on mapping efficiency.

Additionally, it is often helpful to visualize the data using the Integrative Genomics Browser (IGV) or SeqMonk.

### 4.2 Reads alignment

After trimming low quality bases from the reads, the remaining sequences can be mapped to a reference genome. Again, there is no need for a special purpose method for this, so we can use the STAR or the TopHat aligner. For large full-transcript datasets from well annotated organisms (e.g. mouse, human) pseudo-alignment methods (e.g. Kallisto, Salmon) may out-perform conventional alignment. For drop-seq based datasets with tens- or hundreds of thousands of reads pseudoaligners become more appealing since their run-time can be several orders of magnitude less than traditional aligners.

An example of how to map reads.bam to using STAR is

```
$<path_to_STAR>/STAR --runThreadN 1 --runMode alignReads  
--readFilesIn reads1.fq.gz reads2.fq.gz --readFilesCommand zcat --genomeDir <path>  
--parametersFiles FileOfMoreParameters.txt --outFileNamePrefix <outpath>/output
```

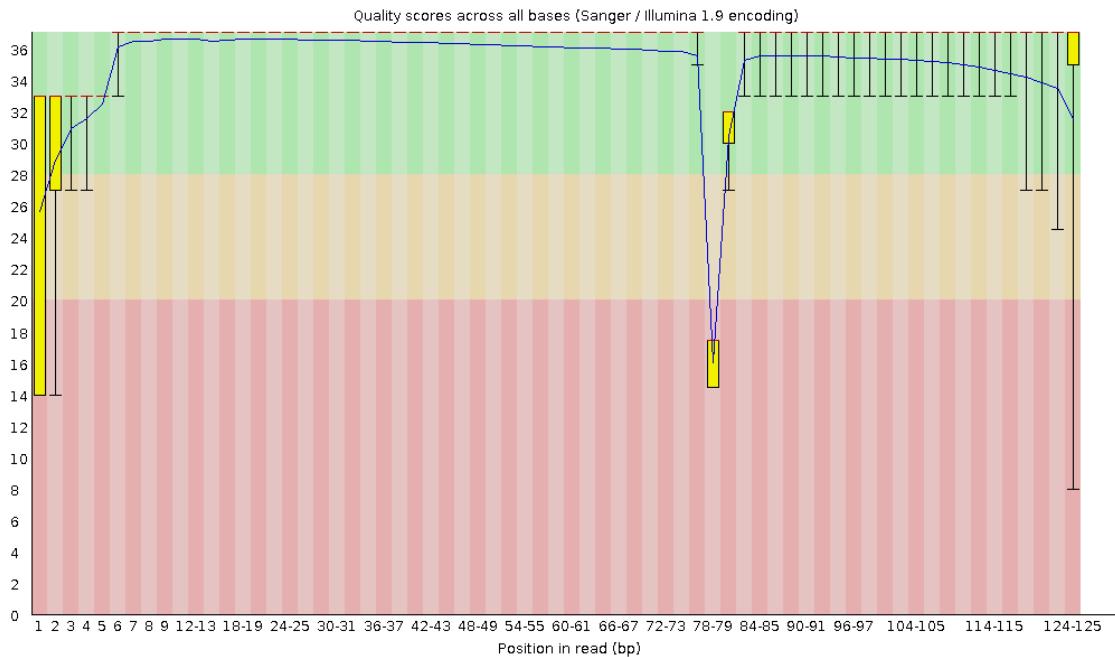


Figure 4.1: Example of FastQC output

**Note**, if the *spike-ins* are used, the reference sequence should be augmented with the DNA sequence of the *spike-in* molecules prior to mapping.

**Note**, when UMIs are used, their barcodes should be removed from the read sequence. A common practice is to add the barcode to the read name.

Once the reads for each cell have been mapped to the reference genome, we need to make sure that a sufficient number of reads from each cell could be mapped to the reference genome. In our experience, the fraction of mappable reads for mouse or human cells is 60-70%. However, this result may vary depending on protocol, read length and settings for the read alignment. As a general rule, we expect all cells to have a similar fraction of mapped reads, so any outliers should be inspected and possibly removed. A low proportion of mappable reads usually indicates contamination.

An example of how to quantify expression using Salmon is

```
$<path_to_Salmon>/salmon quant -i salmon_transcript_index -1 reads1.fq.gz -2 reads2.fq.gz -p #threads -1
```

**Note** Salmon produces estimated read counts and estimated transcripts per million (tpm) in our experience the latter over corrects the expression of long genes for scRNASeq, thus we recommend using read counts.

### 4.3 Alignment example

The histogram below shows the total number of reads mapped to each cell for an scRNA-seq experiment. Each bar represents one cell, and they have been sorted in ascending order by the total number of reads per cell. The three red arrows indicate cells that are outliers in terms of their coverage and they should be removed from further analysis. The two yellow arrows point to cells with a surprisingly large number of unmapped reads. In this example we kept the cells during the alignment QC step, but they were later removed during cell QC due to a high proportion of ribosomal RNA reads.

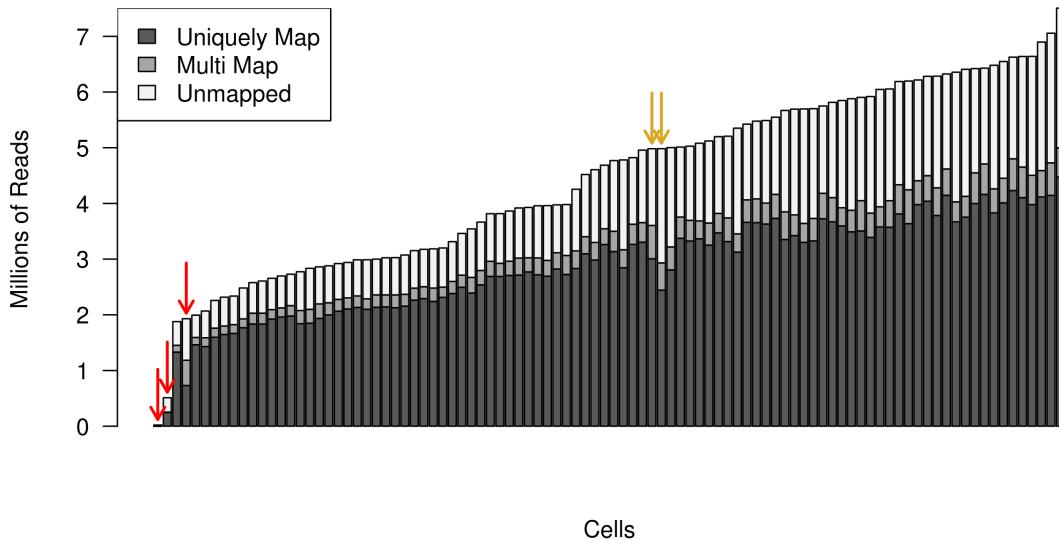


Figure 4.2: Example of the total number of reads mapped to each cell.

## 4.4 Mapping QC

After mapping the raw sequencing to the genome we need to evaluate the quality of the mapping. There are many ways to measure the mapping quality, including: amount of reads mapping to rRNA/tRNAs, proportion of uniquely mapping reads, reads mapping across splice junctions, read depth along the transcripts. Methods developed for bulk RNA-seq, such as RSeQC, are applicable to single-cell data:

```
python <RSeQCpath>/geneBody_coverage.py -i input.bam -r genome.bed -o output.txt
python <RSeQCpath>/bam_stat.py -i input.bam -r genome.bed -o output.txt
python <RSeQCpath>/split_bam.py -i input.bam -r rRNAmask.bed -o output.txt
```

However the expected results will depend on the experimental protocol, e.g. many scRNA-seq methods use poly-A selection to avoid sequencing rRNAs which results in a 3' bias in the read coverage across the genes (aka gene body coverage). The figure below shows this 3' bias as well as three cells which were outliers and removed from the dataset:

## 4.5 Reads quantification

The next step is to quantify the expression level of each gene for each cell. For mRNA data, we can use one of the tools which has been developed for bulk RNA-seq data, e.g. HT-seq or FeatureCounts

```
# include multimapping
<featureCounts_path>/featureCounts -O -M -Q 30 -p -a genome.gtf -o outputFile input.bam
# exclude multimapping
<featureCounts_path>/featureCounts -Q 30 -p -a genome.gtf -o outputFile input.bam
```

Unique molecular identifiers (UMIs) make it possible to count the absolute number of molecules and they have proven popular for scRNA-seq. We will discuss how UMIs can be processed in the next chapter.

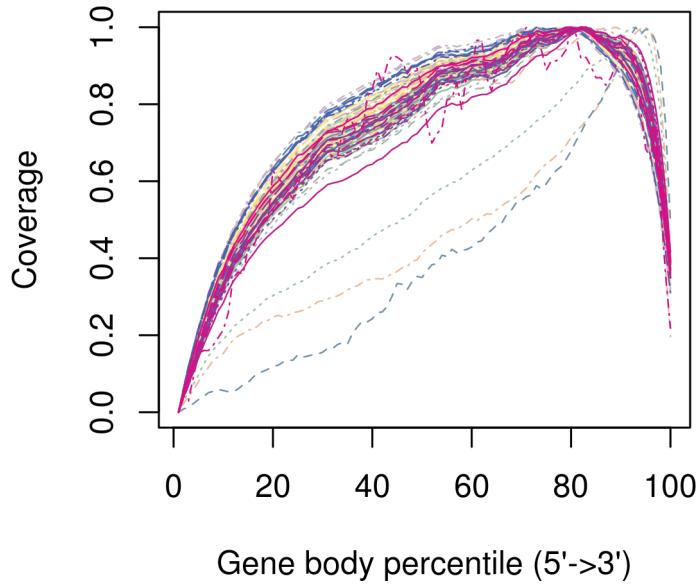


Figure 4.3: Example of the 3' bias in the read coverage.

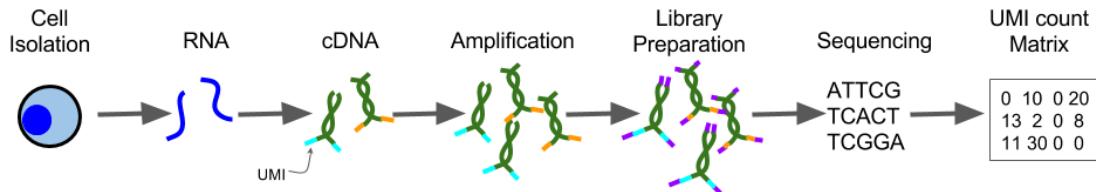


Figure 4.4: UMI sequencing protocol

## 4.6 Unique Molecular Identifiers (UMIs)

Thanks to Andreas Buness from EMBL Monterotondo for collaboration on this section.

### 4.6.1 Introduction

Unique Molecular Identifiers are short (4-10bp) random barcodes added to transcripts during reverse-transcription. They enable sequencing reads to be assigned to individual transcript molecules and thus the removal of amplification noise and biases from scRNASeq data.

When sequencing UMI containing data, techniques are used to specifically sequence only the end of the transcript containing the UMI (usually the 3' end).

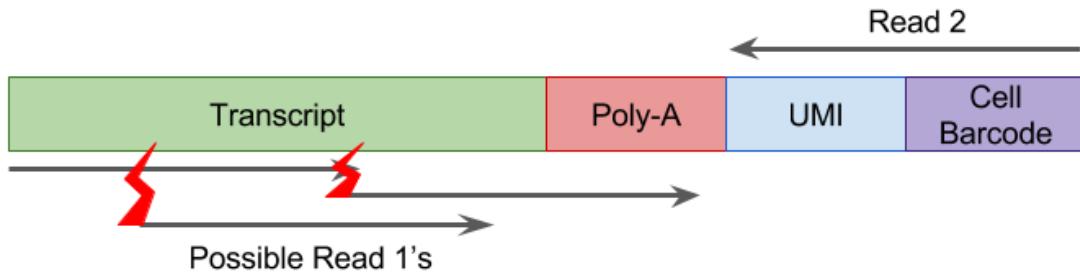


Figure 4.5: UMI sequencing reads, red lightning bolts represent different fragmentation locations

#### 4.6.2 Mapping Barcodes

Since the number of unique barcodes ( $4^N$ , where  $N$  is the length of UMI) is much smaller than the total number of molecules per cell ( $\sim 10^6$ ), each barcode will typically be assigned to multiple transcripts. Hence, to identify unique molecules both barcode and mapping location (transcript) must be used. The first step is to map UMI reads, for which we recommend using STAR since it is fast and outputs good quality BAM-alignments. Moreover, mapping locations can be useful for eg. identifying poorly-annotated 3' UTRs of transcripts.

UMI-sequencing typically consists of paired-end reads where one read from each pair captures the cell and UMI barcodes while the other read consists of exonic sequence from the transcript (Figure 4.5). Note that trimming and/or filtering to remove reads containing poly-A sequence is recommended to avoid errors due to these read mapping to genes/transcripts with internal poly-A/poly-T sequences.

After processing the reads from a UMI experiment, the following conventions are often used:

1. The UMI is added to the read name of the other paired read.
2. Reads are sorted into separate files by cell barcode
  - For extremely large, shallow datasets, the cell barcode may be added to the read name as well to reduce the number of files.

#### 4.6.3 Counting Barcodes

In theory, every unique UMI-transcript pair should represent all reads originating from a single RNA molecule. However, in practice this is frequently not the case and the most common reasons are:

1. **Different UMI does not necessarily mean different molecule**
  - Due to PCR or sequencing errors, base-pair substitution events can result in new UMI sequences. Longer UMIs give more opportunity for errors to arise and based on estimates from cell barcodes we expect 7-10% of 10bp UMIs to contain at least one error. If not corrected for, this type of error will result in an overestimate of the number of transcripts.
2. **Different transcript does not necessarily mean different molecule**
  - Mapping errors and/or multimapping reads may result in some UMIs being assigned to the wrong gene/transcript. This type of error will also result in an overestimate of the number of transcripts.
3. **Same UMI does not necessarily mean same molecule**
  - Biases in UMI frequency and short UMIs can result in the same UMI being attached to different mRNA molecules from the same gene. Thus, the number of transcripts may be underestimated.

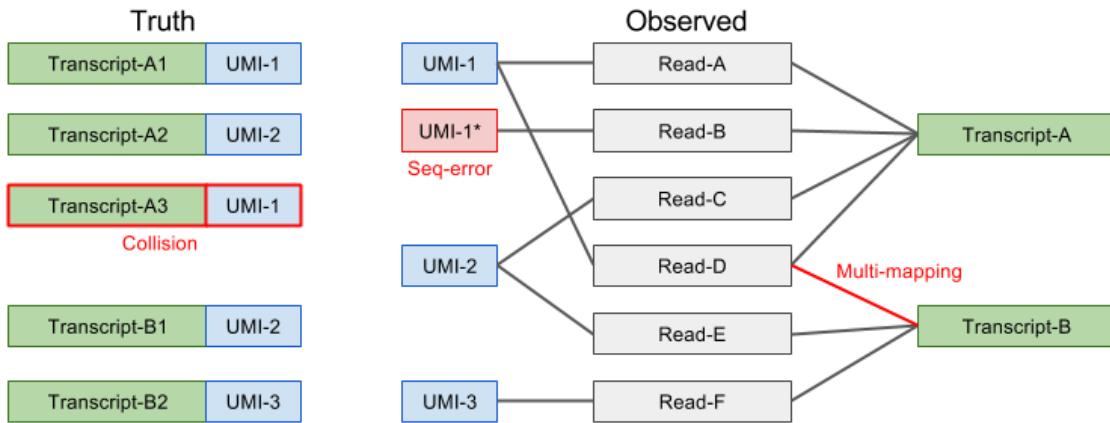


Figure 4.6: Potential Errors in UMIs

#### 4.6.4 Correcting for Errors

How to best account for errors in UMIs remains an active area of research. The best approaches that we are aware of for resolving the issues mentioned above are:

1. UMI-tools' directional-adjacency method implements a procedure which considers both the number of mismatches and the relative frequency of similar UMIs to identify likely PCR/sequencing errors.
2. Currently an open question. The problem may be mitigated by removing UMIs with few reads to support their association with a particular transcript, or by removing all multi-mapping reads.
3. Simple saturation (aka “collision probability”) correction proposed by Grun, Kester and van Oude-naarden (2014) to estimate the true number of molecules  $M$ :

$$M \approx -N * \log\left(1 - \frac{n}{N}\right)$$

where  $N$  = total number of unique UMI barcodes and  $n$  = number of observed barcodes.

An important caveat of this method is that it assumes that all UMIs are equally frequent. In most cases this is incorrect, since there is often a bias related to the GC content.

Determining how to best process and use UMIs is currently an active area of research in the bioinformatics community. We are aware of several methods that have recently been developed, including:

- UMI-tools
- PoissonUMIs
- zUMIs
- dropEst

#### 4.6.5 Downstream Analysis

Current UMI platforms (DropSeq, InDrop, ICell8) exhibit low and highly variable capture efficiency as shown in the figure below.

This variability can introduce strong biases and it needs to be considered in downstream analysis. Recent analyses often pool cells/genes together based on cell-type or biological pathway to increase the power. Robust statistical analyses of this data is still an open research question and it remains to be determined how to best adjust for biases.

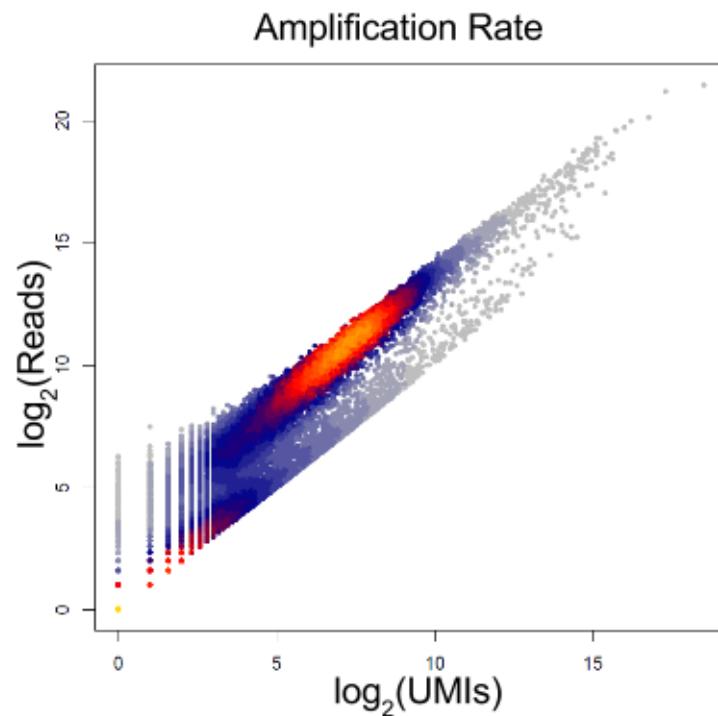


Figure 4.7: Per gene amplification rate

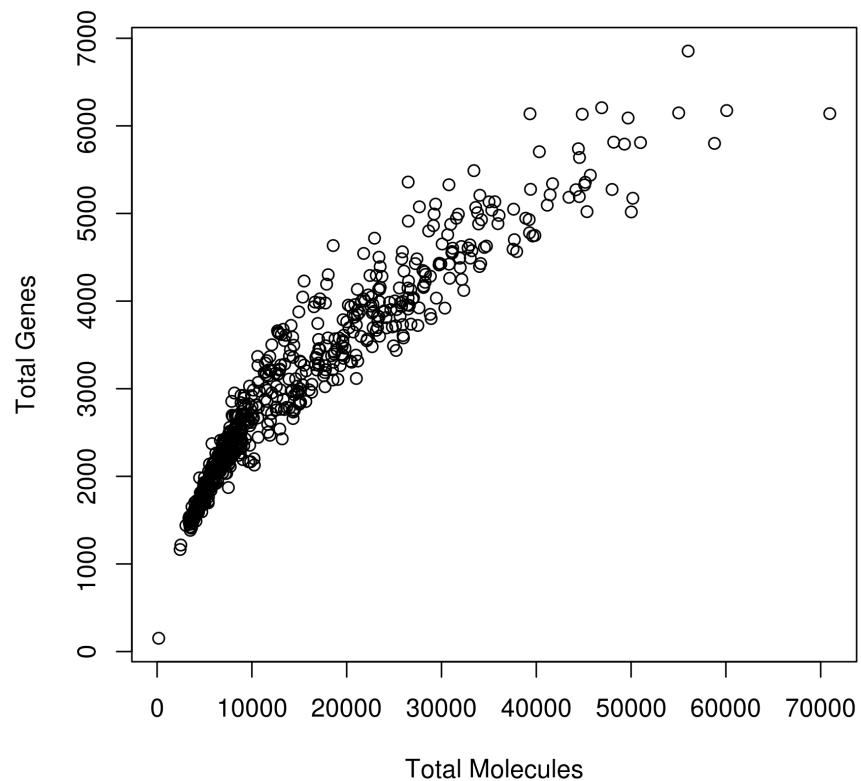


Figure 4.8: Variability in Capture Efficiency

**Exercise 1** We have provided you with UMI counts and read counts from induced pluripotent stem cells generated from three different individuals (Tung et al., 2017) (see: Chapter 7.1 for details of this dataset).

```
umi_counts <- read.table("tung/molecules.txt", sep = "\t")
read_counts <- read.table("tung/reads.txt", sep = "\t")
```

Using this data:

1. Plot the variability in capture efficiency
2. Determine the amplification rate: average number of reads per UMI.

# Chapter 5

## Introduction to R/Bioconductor

### 5.1 Installing packages

#### 5.1.1 CRAN

The Comprehensive R Archive Network CRAN is the biggest archive of R packages. There are few requirements for uploading packages besides building and installing successfully, hence documentation and support is often minimal and figuring how to use these packages can be a challenge in itself. CRAN is the default repository R will search to find packages to install:

```
install.packages("devtools")
require("devtools")
```

#### 5.1.2 Github

Github isn't specific to R, any code of any type in any state can be uploaded. There is no guarantee a package uploaded to github will even install, nevermind do what it claims to do. R packages can be downloaded and installed directly from github using the "devtools" package installed above.

```
devtools::install_github("tallulandrews/M3Drop")
```

Github is also a version control system which stores multiple versions of any package. By default the most recent "master" version of the package is installed. If you want an older version or the development branch this can be specified using the "ref" parameter:

```
# different branch
devtools::install_github("tallulandrews/M3D", ref="nbumi")
# previous commit
devtools::install_github("tallulandrews/M3Drop", ref="434d2da28254acc8de4940c1dc3907ac72973135")
```

Note: make sure you re-install the M3Drop master branch for later in the course.

#### 5.1.3 Bioconductor

Bioconductor is a repository of R-packages specifically for biological analyses. It has the strictest requirements for submission, including installation on every platform and full documentation with a tutorial (called a vignette) explaining how the package should be used. Bioconductor also encourages utilization of standard

data structures/classes and coding style/naming conventions, so that, in theory, packages and analyses can be combined into large pipelines or workflows.

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

Note: in some situations it is necessary to substitute “http://” for “https://” in the above depending on the security features of your internet connection/network.

Bioconductor also requires creators to support their packages and has a regular 6-month release schedule. Make sure you are using the most recent release of bioconductor before trying to install packages for the course.

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

#### 5.1.4 Source

The final way to install packages is directly from source. In this case you have to download a fully built source code file, usually packagename.tar.gz, or clone the github repository and rebuild the package yourself. Generally this will only be done if you want to edit a package yourself, or if for some reason the former methods have failed.

```
install.packages("M3Drop_3.05.00.tar.gz", type="source")
```

## 5.2 Installation instructions:

All the packages necessary for this course are available here. Starting from “RUN Rscript -e”`install.packages('devtools')`“, run each of the commands (minus“RUN“) on the command line or start an R session and run each of the commands within the quotation marks. Note the ordering of the installation is important in some cases, so make sure you run them in order from top to bottom.

## 5.3 Data-types/classes

R is a high level language so the underlying data-type is generally not important. The exception if you are accessing R data directly using another language such as C, but that is beyond the scope of this course. Instead we will consider the basic data classes: numeric, integer, logical, and character, and the higher level data class called “factor”. You can check what class your data is using the “`class()`” function.

Aside: R can also store data as “complex” for complex numbers but generally this isn’t relevant for biological analyses.

### 5.3.1 Numeric

The “numeric” class is the default class for storing any numeric data - integers, decimal numbers, numbers in scientific notation, etc...

```
x = 1.141
class(x)
```

```
## [1] "numeric"
```

```
y = 42
class(y)

## [1] "numeric"

z = 6.02e23
class(z)

## [1] "numeric"
```

Here we see that even though R has an “integer” class and 42 could be stored more efficiently as an integer the default is to store it as “numeric”. If we want 42 to be stored as an integer we must “coerce” it to that class:

```
y = as.integer(42)
class(y)
```

```
## [1] "integer"
```

Coercion will force R to store data as a particular class, if our data is incompatible with that class it will still do it but the data will be converted to NAs:

```
as.numeric("H")
```

```
## Warning: NAs introduced by coercion
```

```
## [1] NA
```

Above we tried to coerce “character” data, identified by the double quotation marks, into numeric data which doesn’t make sense, so we triggered (“threw”) an warning message. Since this is only a warning R would continue with any subsequent commands in a script/function, whereas an “error” would cause R to halt.

### 5.3.2 Character/String

The “character” class stores all kinds of text data. Programming convention calls data containing multiple letters a “string”, thus most R functions which act on character data will refer to the data as “strings” and will often have “str” or “string” in its name. Strings are identified by being flanked by double quotation marks, whereas variable/function names are not:

```
x = 5

a = "x" # character "x"
a

## [1] "x"

b = x # variable x
b

## [1] 5
```

In addition to standard alphanumeric characters, strings can also store various special characters. Special characters are identified using a backslash followed by a single character, the most relevant are the special character for tab : \t and new line : \n. To demonstrate the these special characters lets concatenate (cat) together two strings with these characters separating (sep) them:

```
cat("Hello", "World", sep= " ")
```

```
## Hello World
```

```
cat("Hello", "World", sep= "\t")
```

```
## Hello      World
cat("Hello", "World", sep= "\n")
```

```
## Hello
## World
```

Note that special characters work differently in different functions. For instance the `paste` function does the same thing as `cat` but does not recognize special characters.

```
paste("Hello", "World", sep= " ")
```

```
## [1] "Hello World"
paste("Hello", "World", sep= "\t")
```

```
## [1] "Hello\tWorld"
paste("Hello", "World", sep= "\n")
```

```
## [1] "Hello\nWorld"
```

Single or double backslash is also used as an `escape` character to turn off special characters or allow quotation marks to be included in strings:

```
cat("This \"string\" contains quotation marks.")
```

```
## This "string" contains quotation marks.
```

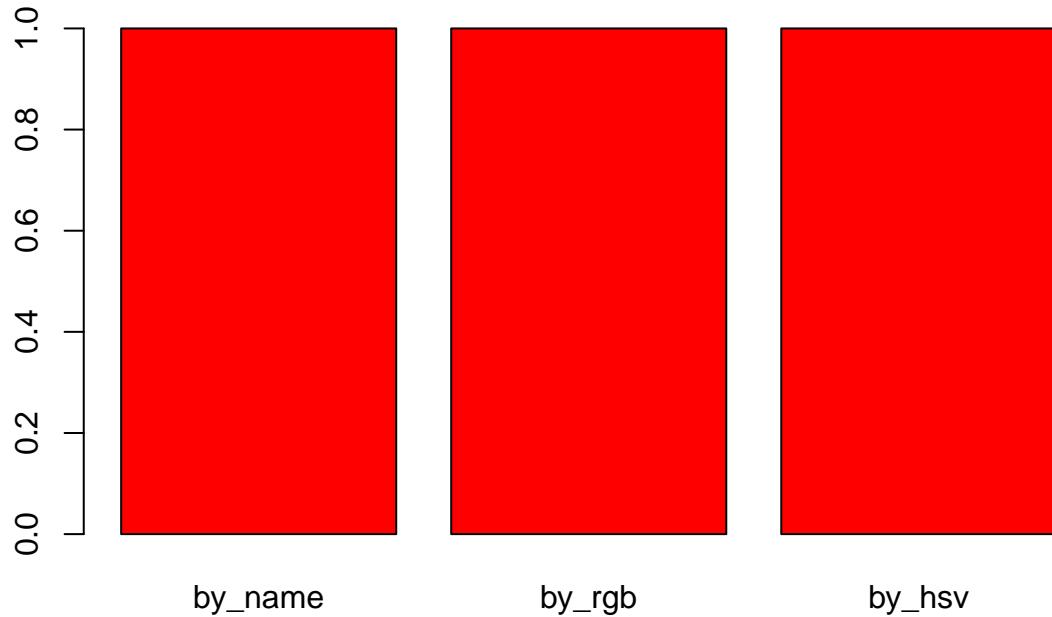
Special characters are generally only used in pattern matching, and reading/writing data to files. For instance this is how you would read a tab-separated file into R.

```
dat = read.delim("file.tsv", sep="\t")
```

Another special type of character data are colours. Colours can be specified in three main ways: by name from those available, by red, green, blue values using the `rgb` function, and by hue (colour), saturation (colour vs white) and value (colour/white vs black) using the `hsv` function. By default `rgb` and `hsv` expect three values in 0-1 with an optional fourth value for transparency. Alternatively, sets of predetermined colours with useful properties can be loaded from many different packages with `RColorBrewer` being one of the most popular.

```
reds = c("red", rgb(1,0,0), hsv(0, 1, 1))
reds
```

```
## [1] "red"      "#FF0000" "#FF0000"
barplot(c(1,1,1), col=reds, names=c("by_name", "by_rgb", "by_hsv"))
```



### 5.3.3 Logical

The `logical` class stores boolean truth values, i.e. `TRUE` and `FALSE`. It is used for storing the results of logical operations and conditional statements will be coerced to this class. Most other data-types can be coerced to boolean without triggering (or “throwing”) error messages, which may cause unexpected behaviour.

```
x = TRUE
class(x)

## [1] "logical"

y = "T"
as.logical(y)

## [1] TRUE

z = 5
as.logical(z)

## [1] TRUE

x = FALSE
class(x)

## [1] "logical"

y = "F"
as.logical(y)

## [1] FALSE

z = 0
as.logical(z)

## [1] FALSE
```

**Exercise 1** Experiment with other character and numeric values, which are coerced to TRUE or FALSE? which are coerced to neither? Do you ever throw a warning/error message?

### 5.3.4 Factors

String/Character data is very memory inefficient to store, each letter generally requires the same amount of memory as any integer. Thus when storing a vector of strings with repeated elements it is more efficient assign each element to an integer and store the vector as integers and an additional string-to-integer association table. Thus, by default R will read in text columns of a data table as factors.

```
str_vector = c("Apple", "Apple", "Banana", "Banana", "Banana", "Carrot", "Carrot", "Apple", "Banana")
factored_vector = factor(str_vector)
factored_vector

## [1] Apple Apple Banana Banana Banana Carrot Carrot Apple Banana
## Levels: Apple Banana Carrot
as.numeric(factored_vector)

## [1] 1 1 2 2 2 3 3 1 2
```

The double nature of factors can cause some unintuitive behaviour. E.g. joining two factors together will convert them to the numeric form and the original strings will be lost.

```
c(factored_vector, factored_vector)

## [1] 1 1 2 2 2 3 3 1 2 1 1 2 2 2 3 3 1 2
```

Likewise if due to formatting issues numeric data is mistakenly interpreted as strings, then you must convert the factor back to strings before coercing to numeric values:

```
x = c("20", "25", "23", "38", "20", "40", "25", "30")
x = factor(x)
as.numeric(x)

## [1] 1 3 2 5 1 6 3 4
as.numeric(as.character(x))

## [1] 20 25 23 38 20 40 25 30
```

To make R read text as character data instead of factors set the environment option `stringsAsFactors=FALSE`. This must be done at the start of each R session.

```
options(stringsAsFactors=FALSE)
```

**Exercise** How would you use factors to create a vector of colours for an arbitrarily long vector of fruits like `str_vector` above? **Answer**

### 5.3.5 Checking class/type

We recommend checking your data is of the correct class after reading from files:

```
x = 1.4
is.numeric(x)

## [1] TRUE
is.character(x)
```

```
## [1] FALSE
is.logical(x)

## [1] FALSE
is.factor(x)

## [1] FALSE
```

## 5.4 Basic data structures

So far we have only looked at single values and vectors. Vectors are the simplest data structure in R. They are a 1-dimensional array of data all of the same type. If the input when creating a vector is of different types it will be coerced to the data-type that is most consistent with the data.

```
x = c("Hello", 5, TRUE)
x

## [1] "Hello" 5      TRUE
class(x)

## [1] "character"
```

Here we tried to put character, numeric and logical data into a single vector so all the values were coerced to `character` data.

A `matrix` is the two dimensional version of a vector, it also requires all data to be of the same type. If we combine a character vector and a numeric vector into a matrix, all the data will be coerced to characters:

```
x = c("A", "B", "C")
y = c(1, 2, 3)
class(x)

## [1] "character"
class(y)

## [1] "numeric"
m = cbind(x, y)
m

##      x   y
## [1,] "A" "1"
## [2,] "B" "2"
## [3,] "C" "3"
```

The quotation marks indicate that the numeric vector has been coerced to characters. Alternatively, to store data with columns of different data-types we can use a `dataframe`.

```
z = data.frame(x, y)
z

##   x y
## 1 A 1
## 2 B 2
## 3 C 3
```

```
class(z[,1])
## [1] "character"
class(z[,2])
## [1] "numeric"
```

If you have set `stringsAsFactors=FALSE` as above you will find the first column remains characters, otherwise it will be automatically converted to a factor.

```
options(stringsAsFactors=TRUE)
z = data.frame(x, y)
class(z[,1])
```

```
## [1] "factor"
```

Another difference between matrices and dataframes is the ability to select columns using the `$` operator:

```
m$x # throws an error
z$x # ok
```

The final basic data structure is the `list`. Lists allow data of different types and different lengths to be stored in a single object. Each element of a list can be any other R object : data of any type, any data structure, even other lists or functions.

```
l = list(m, z)
l1 = list(sublist=l, a_matrix=m, numeric_value=42, this_string="Hello World", even_a_function=cbind)
l1

## $sublist
## $sublist[[1]]
##   x   y
## [1,] "A" "1"
## [2,] "B" "2"
## [3,] "C" "3"
##
## $sublist[[2]]
##   x y
## 1 A 1
## 2 B 2
## 3 C 3
##
## 
## $a_matrix
##   x   y
## [1,] "A" "1"
## [2,] "B" "2"
## [3,] "C" "3"
##
## $numeric_value
## [1] 42
##
## $this_string
## [1] "Hello World"
##
## $even_a_function
## function (... , deparse.level = 1)
```

```
## .Internal(cbind(deparse.level, ...))
## <bytecode: 0x55e4ded2f378>
## <environment: namespace:base>
```

Lists are most commonly used when returning a large number of results from a function that do not fit into any of the previous data structures.

## 5.5 More information

You can get more information about any R commands relevant to these datatypes using by typing `?function` in an interactive session.

## 5.6 Data Types

### 5.6.1 What is Tidy Data?

Tidy data is a concept largely defined by Hadley Wickham (Wickham, 2014). Tidy data has the following three characteristics:

1. Each variable has its own column.
2. Each observation has its own row.
3. Each value has its own cell.

Here is an example of some tidy data:

```
##   Students  Subject Years Score
## 1     Mark      Maths    1     5
## 2     Jane    Biology    2     6
## 3 Mohammed Physics    3     4
## 4     Tom      Maths    2     7
## 5   Celia Computing    3     9
```

Here is an example of some untidy data:

```
##   Students Sport Category Counts
## 1     Matt  Tennis      Wins      0
## 2     Matt  Tennis    Losses      1
## 3    Ellie  Rugby      Wins      3
## 4    Ellie  Rugby    Losses      2
## 5     Tim Football      Wins      1
## 6     Tim Football    Losses      4
## 7   Louise Swimming      Wins      2
## 8   Louise Swimming    Losses      2
## 9    Kelly  Running      Wins      5
## 10   Kelly  Running    Losses      1
```

Task 1: In what ways is the untidy data not tidy? How could we make the untidy data tidy?

Tidy data is generally easier to work with than untidy data, especially if you are working with packages such as ggplot. Fortunately, packages are available to make untidy data tidy. Today we will explore a few of the functions available in the `tidyverse` package which can be used to make untidy data tidy. If you are interested in finding out more about tidying data, we recommend reading “R for Data Science”, by Garrett Grolemund and Hadley Wickham. An electronic copy is available here: <http://r4ds.had.co.nz/>

The untidy data above is untidy because two variables (`Wins` and `Losses`) are stored in one column (`Category`). This is a common way in which data can be untidy. To tidy this data, we need to make `Wins` and `Losses` into columns, and store the values in `Counts` in these columns. Fortunately, there is a function from the tidyverse packages to perform this operation. The function is called `spread`, and it takes two arguments, `key` and `value`. You should pass the name of the column which contains multiple variables to `key`, and pass the name of the column which contains values from multiple variables to `value`. For example:

```
library(tidyverse)
sports<-data.frame(Students=c("Matt", "Matt", "Ellie", "Ellie", "Tim", "Tim", "Louise", "Louise", "Kelly", "Kelly"))
sports

##   Students   Sport Category Counts
## 1      Matt Tennis     Wins      0
## 2      Matt Tennis    Losses      1
## 3     Ellie Rugby     Wins      3
## 4     Ellie Rugby    Losses      2
## 5      Tim Football   Wins      1
## 6      Tim Football  Losses      4
## 7    Louise Swimming   Wins      2
## 8    Louise Swimming  Losses      2
## 9     Kelly Running    Wins      5
## 10    Kelly Running   Losses      1

spread(sports, key=Category, value=Counts)

##   Students   Sport Losses Wins
## 1     Ellie Rugby     2     3
## 2     Kelly Running   1     5
## 3    Louise Swimming  2     2
## 4      Matt Tennis    1     0
## 5      Tim Football   4     1
```

Task 2: The dataframe `foods` defined below is untidy. Work out why and use `spread()` to tidy it

```
foods<-data.frame(student=c("Antoinette","Antoinette","Taylor", "Taylor", "Alexa", "Alexa"), Category=c("Ramen", "Ramen", "Pasta", "Pasta", "Sushi", "Sushi"))
```

The other common way in which data can be untidy is if the columns are values instead of variables. For example, the dataframe below shows the percentages some students got in tests they did in May and June. The data is untidy because the columns `May` and `June` are values, not variables.

```
percentages<-data.frame(student=c("Alejandro", "Pietro", "Jane"), "May"=c(90,12,45), "June"=c(80,30,100))
```

Fortunately, there is a function in the tidyverse packages to deal with this problem too. `gather()` takes the names of the columns which are values, the `key` and the `value` as arguments. This time, the `key` is the name of the variable with values as column names, and the `value` is the name of the variable with values spread over multiple columns. Ie:

```
gather(percentages, "May", "June", key="Month", value = "Percentage")
```

```
##   student Month Percentage
## 1 Alejandro   May        90
## 2    Pietro   May        12
## 3      Jane   May        45
## 4 Alejandro  June        80
## 5    Pietro  June        30
## 6      Jane  June       100
```

These examples don't have much to do with single-cell RNA-seq analysis, but are designed to help illustrate the features of tidy and untidy data. You will find it much easier to analyse your single-cell RNA-seq data if

your data is stored in a tidy format. Fortunately, the data structures we commonly use to facilitate single-cell RNA-seq analysis usually encourage store your data in a tidy manner.

### 5.6.2 What is Rich Data?

If you google ‘rich data’, you will find lots of different definitions for this term. In this course, we will use ‘rich data’ to mean data which is generated by combining information from multiple sources. For example, you could make rich data by creating an object in R which contains a matrix of gene expression values across the cells in your single-cell RNA-seq experiment, but also information about how the experiment was performed. Objects of the `SingleCellExperiment` class, which we will discuss below, are an example of rich data.

### 5.6.3 What is Bioconductor?

From Wikipedia: Bioconductor is a free, open source and open development software project for the analysis and comprehension of genomic data generated by wet lab experiments in molecular biology. Bioconductor is based primarily on the statistical R programming language, but does contain contributions in other programming languages. It has two releases each year that follow the semiannual releases of R. At any one time there is a release version, which corresponds to the released version of R, and a development version, which corresponds to the development version of R. Most users will find the release version appropriate for their needs.

We strongly recommend all new comers and even experienced high-throughput data analysts to use well developed and maintained Bioconductor methods and classes.

### 5.6.4 SingleCellExperiment class

`SingleCellExperiment` (SCE) is a S4 class for storing data from single-cell experiments. This includes specialized methods to store and retrieve spike-in information, dimensionality reduction coordinates and size factors for each cell, along with the usual metadata for genes and libraries.

In practice, an object of this class can be created using its constructor:

```
library(SingleCellExperiment)
counts <- matrix(rpois(100, lambda = 10), ncol=10, nrow=10)
rownames(counts) <- paste("gene", 1:10, sep = "")
colnames(counts) <- paste("cell", 1:10, sep = "")
sce <- SingleCellExperiment(
  assays = list(counts = counts),
  rowData = data.frame(gene_names = paste("gene_name", 1:10, sep = "")),
  colData = data.frame(cell_names = paste("cell_name", 1:10, sep = ""))
)
sce

## class: SingleCellExperiment
## dim: 10 10
## metadata(0):
## assays(1): counts
## rownames(10): gene1 gene2 ... gene9 gene10
## rowData names(1): gene_names
## colnames(10): cell1 cell2 ... cell9 cell10
## colData names(1): cell_names
## reducedDimNames(0):
```

```
## spikeNames(0) :
```

In the `SingleCellExperiment`, users can assign arbitrary names to entries of assays. To assist interoperability between packages, some suggestions for what the names should be for particular types of data are provided by the authors:

- **counts**: Raw count data, e.g., number of reads or transcripts for a particular gene.
- **normcounts**: Normalized values on the same scale as the original counts. For example, counts divided by cell-specific size factors that are centred at unity.
- **logcounts**: Log-transformed counts or count-like values. In most cases, this will be defined as log-transformed normcounts, e.g., using log base 2 and a pseudo-count of 1.
- **cpm**: Counts-per-million. This is the read count for each gene in each cell, divided by the library size of each cell in millions.
- **tpm**: Transcripts-per-million. This is the number of transcripts for each gene in each cell, divided by the total number of transcripts in that cell (in millions).

Each of these suggested names has an appropriate getter/setter method for convenient manipulation of the `SingleCellExperiment`. For example, we can take the (very specifically named) `counts` slot, normalise it and assign it to `normcounts` instead:

```
normcounts(sce) <- log2(counts(sce) + 1)
sce
```

```
## class: SingleCellExperiment
## dim: 10 10
## metadata(0):
## assays(2): counts normcounts
## rownames(10): gene1 gene2 ... gene9 gene10
## rowData names(1): gene_names
## colnames(10): cell1 cell2 ... cell9 cell10
## colData names(1): cell_names
## reducedDimNames(0):
## spikeNames(0):
dim(normcounts(sce))

## [1] 10 10
head(normcounts(sce))

##          cell1    cell2    cell3    cell4    cell5    cell6    cell7
## gene1 3.169925 3.169925 2.000000 2.584963 2.584963 3.321928 3.584963
## gene2 3.459432 1.584963 3.584963 3.807355 3.700440 3.700440 3.000000
## gene3 3.000000 3.169925 3.807355 3.169925 3.321928 3.321928 3.321928
## gene4 3.584963 3.459432 3.000000 3.807355 3.700440 3.700440 3.700440
## gene5 3.906891 3.000000 3.169925 3.321928 3.584963 3.459432 3.807355
## gene6 3.700440 3.700440 3.584963 4.000000 3.169925 3.000000 3.459432
##          cell8    cell9    cell10
## gene1 3.321928 3.807355 2.807355
## gene2 3.807355 3.700440 4.000000
## gene3 2.584963 4.000000 3.700440
## gene4 3.169925 3.584963 3.700440
## gene5 3.807355 2.584963 3.584963
## gene6 3.321928 3.459432 4.000000
```

### 5.6.5 scater package

**scater** is a R package for single-cell RNA-seq analysis (McCarthy et al., 2017). The package contains several useful methods for quality control, visualisation and pre-processing of data prior to further downstream analysis.

**scater** features the following functionality:

- Automated computation of QC metrics
- Transcript quantification from read data with pseudo-alignment
- Data format standardisation
- Rich visualizations for exploratory analysis
- Seamless integration into the Bioconductor universe
- Simple normalisation methods

We highly recommend to use **scater** for all single-cell RNA-seq analyses and **scater** is the basis of the first part of the course.

As illustrated in the figure below, **scater** will help you with quality control, filtering and normalization of your expression matrix following mapping and alignment. Keep in mind that this figure represents the original version of **scater** where an **SCESet** class was used. In the newest version this figure is still correct, except that **SCESet** can be substituted with the **SingleCellExperiment** class.

## 5.7 Bioconductor, SingleCellExperiment and scater

### 5.7.1 Bioconductor

From Wikipedia: Bioconductor is a free, open source and open development software project for the analysis and comprehension of genomic data generated by wet lab experiments in molecular biology. Bioconductor is based primarily on the statistical R programming language, but does contain contributions in other programming languages. It has two releases each year that follow the semiannual releases of R. At any one time there is a release version, which corresponds to the released version of R, and a development version, which corresponds to the development version of R. Most users will find the release version appropriate for their needs.

We strongly recommend all new comers and even experienced high-throughput data analysts to use well developed and maintained Bioconductor methods and classes.

### 5.7.2 SingleCellExperiment class

**SingleCellExperiment** (SCE) is a S4 class for storing data from single-cell experiments. This includes specialized methods to store and retrieve spike-in information, dimensionality reduction coordinates and size factors for each cell, along with the usual metadata for genes and libraries.

In practice, an object of this class can be created using its constructor:

```
library(SingleCellExperiment)
counts <- matrix(rpois(100, lambda = 10), ncol=10, nrow=10)
rownames(counts) <- paste("gene", 1:10, sep = "")
colnames(counts) <- paste("cell", 1:10, sep = "")
sce <- SingleCellExperiment(
  assays = list(counts = counts),
  rowData = data.frame(gene_names = paste("gene_name", 1:10, sep = "")),
  colData = data.frame(cell_names = paste("cell_name", 1:10, sep = "")))
```

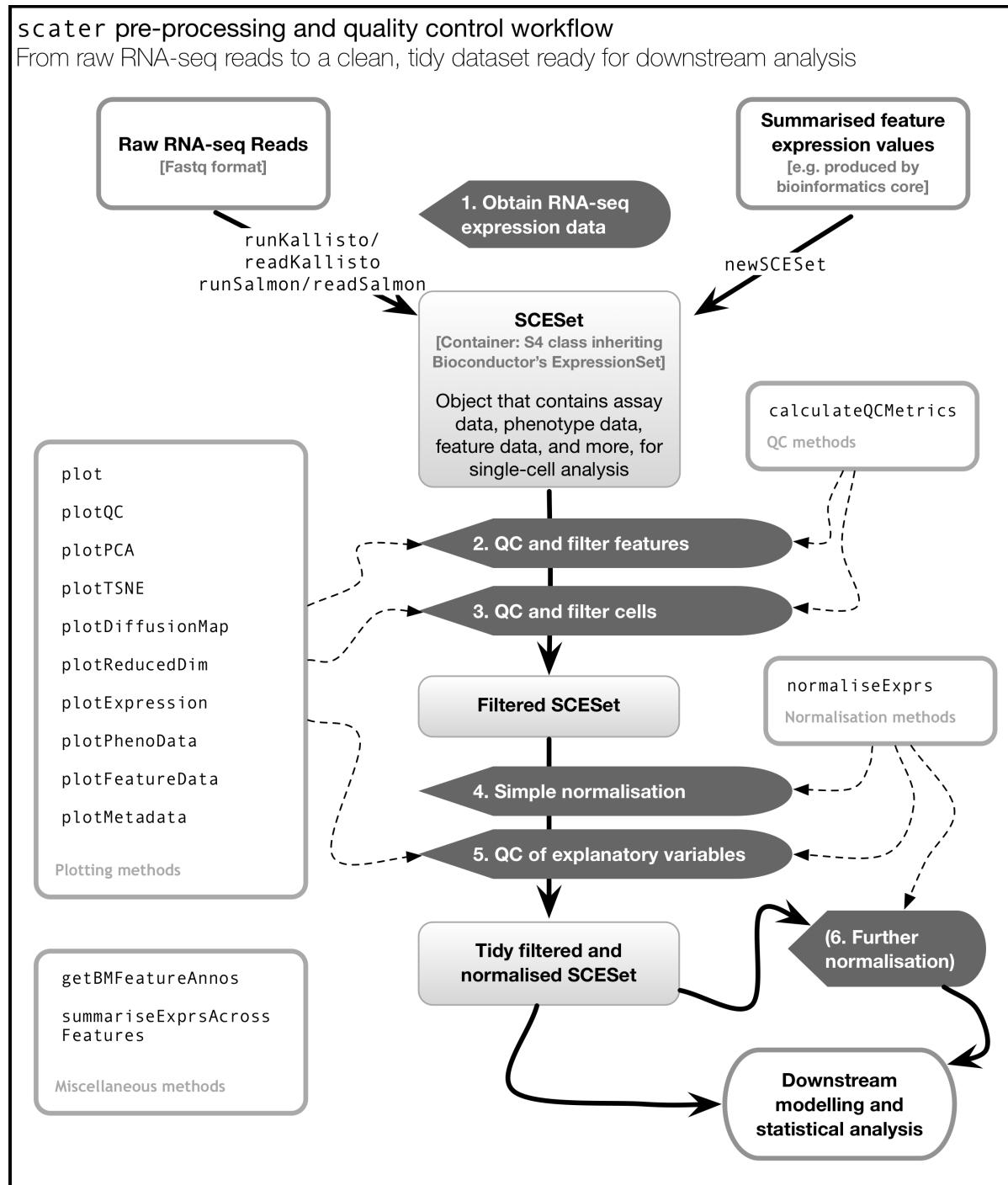


Figure 5.1:

```
)
sce

## class: SingleCellExperiment
## dim: 10 10
## metadata(0):
## assays(1): counts
## rownames(10): gene1 gene2 ... gene9 gene10
## rowData names(1): gene_names
## colnames(10): cell1 cell2 ... cell9 cell10
## colData names(1): cell_names
## reducedDimNames(0):
## spikeNames(0):
```

In the `SingleCellExperiment`, users can assign arbitrary names to entries of assays. To assist interoperability between packages, some suggestions for what the names should be for particular types of data are provided by the authors:

- **counts**: Raw count data, e.g., number of reads or transcripts for a particular gene.
- **normcounts**: Normalized values on the same scale as the original counts. For example, counts divided by cell-specific size factors that are centred at unity.
- **logcounts**: Log-transformed counts or count-like values. In most cases, this will be defined as log-transformed normcounts, e.g., using log base 2 and a pseudo-count of 1.
- **cpm**: Counts-per-million. This is the read count for each gene in each cell, divided by the library size of each cell in millions.
- **tpm**: Transcripts-per-million. This is the number of transcripts for each gene in each cell, divided by the total number of transcripts in that cell (in millions).

Each of these suggested names has an appropriate getter/setter method for convenient manipulation of the `SingleCellExperiment`. For example, we can take the (very specifically named) `counts` slot, normalise it and assign it to `normcounts` instead:

```
normcounts(sce) <- log2(counts(sce) + 1)
sce

## class: SingleCellExperiment
## dim: 10 10
## metadata(0):
## assays(2): counts normcounts
## rownames(10): gene1 gene2 ... gene9 gene10
## rowData names(1): gene_names
## colnames(10): cell1 cell2 ... cell9 cell10
## colData names(1): cell_names
## reducedDimNames(0):
## spikeNames(0):

dim(normcounts(sce))

## [1] 10 10
head(normcounts(sce))

##          cell1    cell2    cell3    cell4    cell5    cell6    cell7
## gene1 3.169925 3.169925 2.000000 2.584963 2.584963 3.321928 3.584963
## gene2 3.459432 1.584963 3.584963 3.807355 3.700440 3.700440 3.000000
## gene3 3.000000 3.169925 3.807355 3.169925 3.321928 3.321928 3.321928
## gene4 3.584963 3.459432 3.000000 3.807355 3.700440 3.700440 3.700440
## gene5 3.906891 3.000000 3.169925 3.321928 3.584963 3.459432 3.807355
```

```
## gene6 3.700440 3.700440 3.584963 4.000000 3.169925 3.000000 3.459432
##           cell18    cell19    cell110
## gene1 3.321928 3.807355 2.807355
## gene2 3.807355 3.700440 4.000000
## gene3 2.584963 4.000000 3.700440
## gene4 3.169925 3.584963 3.700440
## gene5 3.807355 2.584963 3.584963
## gene6 3.321928 3.459432 4.000000
```

### 5.7.3 scater package

`scater` is a R package for single-cell RNA-seq analysis (McCarthy et al., 2017). The package contains several useful methods for quality control, visualisation and pre-processing of data prior to further downstream analysis.

`scater` features the following functionality:

- Automated computation of QC metrics
- Transcript quantification from read data with pseudo-alignment
- Data format standardisation
- Rich visualizations for exploratory analysis
- Seamless integration into the Bioconductor universe
- Simple normalisation methods

We highly recommend to use `scater` for all single-cell RNA-seq analyses and `scater` is the basis of the first part of the course.

As illustrated in the figure below, `scater` will help you with quality control, filtering and normalization of your expression matrix following mapping and alignment. Keep in mind that this figure represents the original version of `scater` where an `SCESet` class was used. In the newest version this figure is still correct, except that `SCESet` can be substituted with the `SingleCellExperiment` class.

## 5.8 An Introduction to ggplot2

### 5.8.1 What is ggplot2?

`ggplot2` is an R package designed by Hadley Wickham which facilitates data plotting. In this lab, we will touch briefly on some of the features of the package. If you would like to learn more about how to use `ggplot2`, we would recommend reading “`ggplot2` Elegant graphics for data analysis”, by Hadley Wickham.

### 5.8.2 Principles of ggplot2

- Your data must be a data frame if you want to plot it using `ggplot2`.
- Use the `aes` mapping function to specify how variables in the data frame map to features on your plot
- Use geoms to specify how your data should be represented on your graph eg. as a scatterplot, a barplot, a boxplot etc.

### 5.8.3 Using the aes mapping function

The `aes` function specifies how variables in your data frame map to features on your plot. To understand how this works, let’s look at an example:

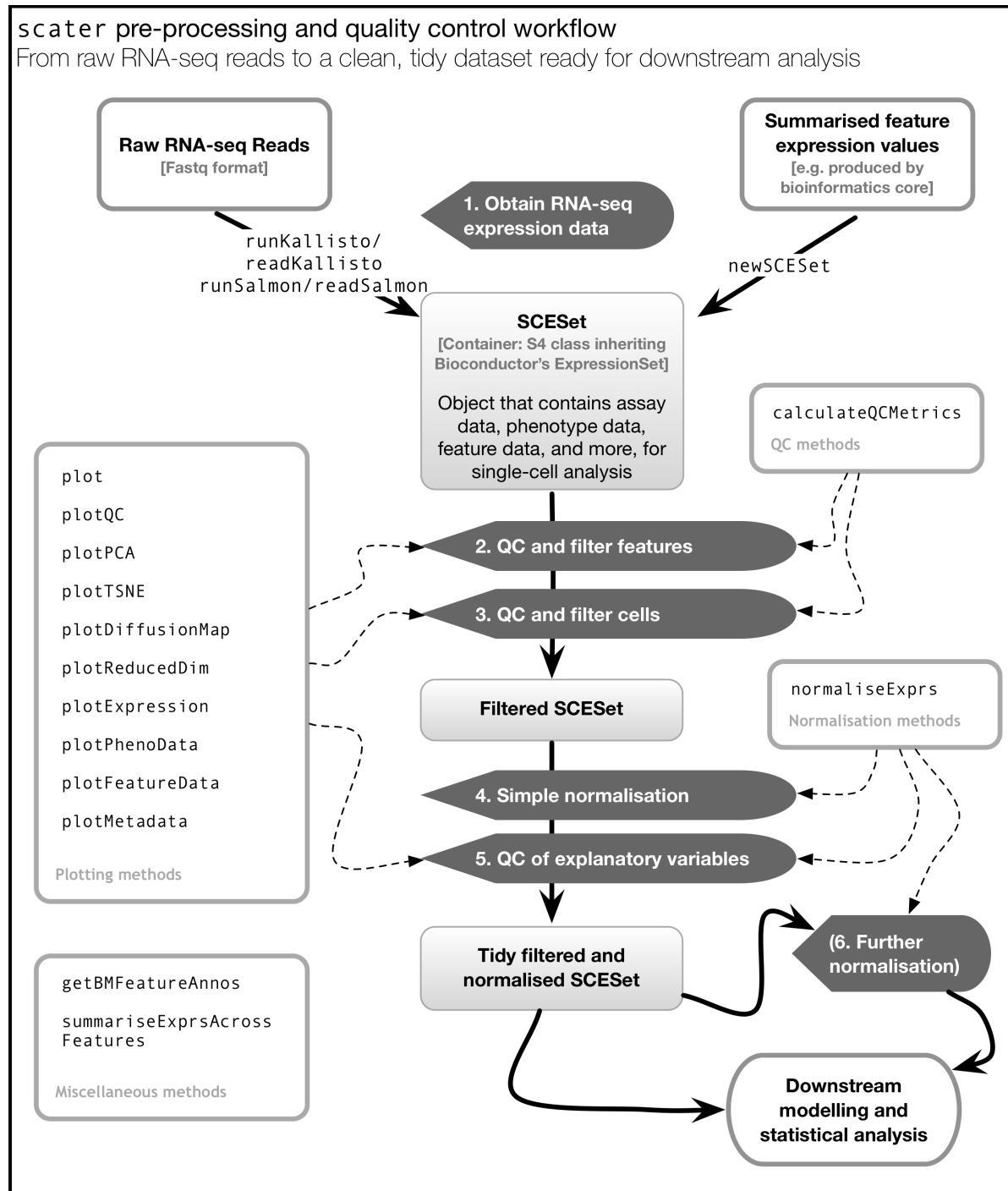


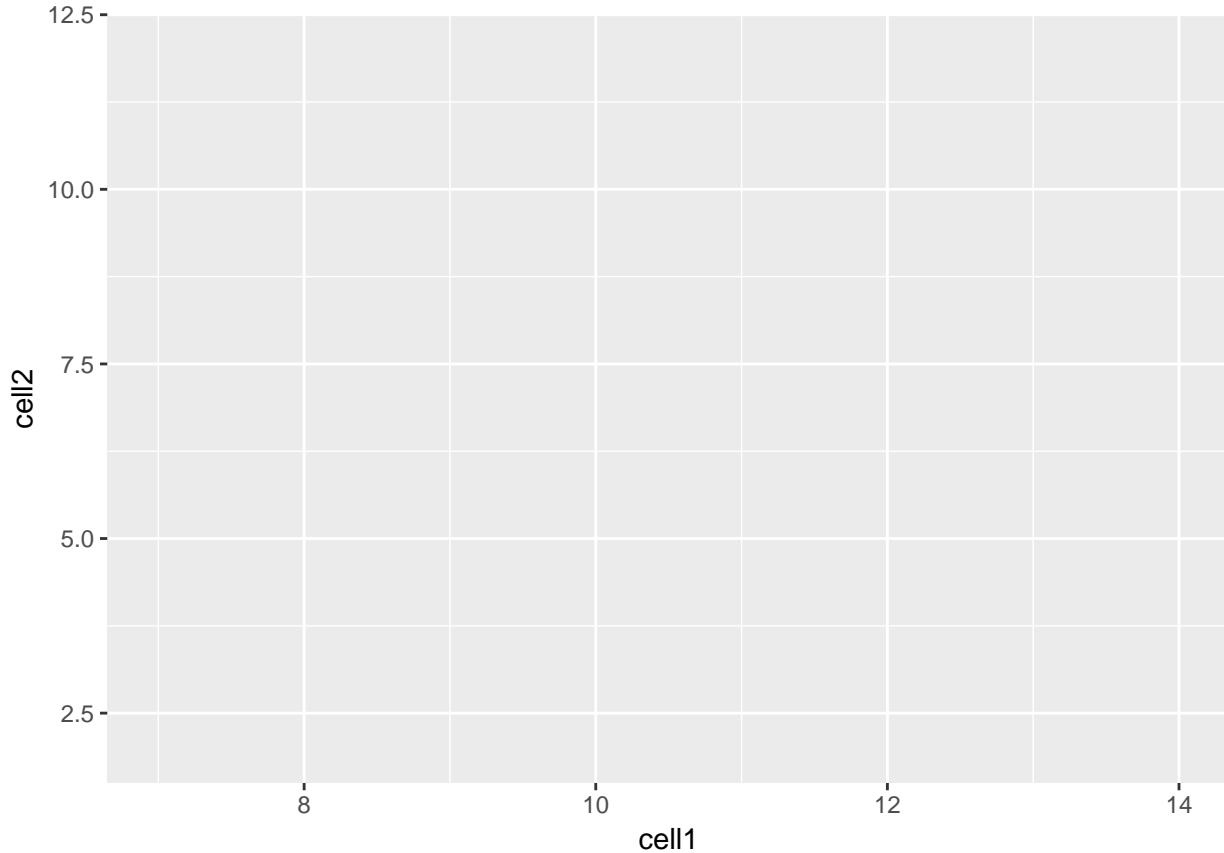
Figure 5.2:

```

library(ggplot2)
library(tidyverse)
set.seed(1)
counts <- as.data.frame(matrix(rpois(100, lambda = 10), ncol=10, nrow=10))
Gene_ids <- paste("gene", 1:10, sep = "")
colnames(counts) <- paste("cell", 1:10, sep = "")
counts<-data.frame(Gene_ids, counts)
counts

##      Gene_ids cell1  cell2  cell3  cell4  cell5  cell6  cell7  cell8  cell9  cell10
## 1    gene1     8      8      3      5      5      9     11      9     13      6
## 2    gene2    10      2     11     13     12     12      7     13     12     15
## 3    gene3     7      8     13      8      9      9      9      5     15     12
## 4    gene4    11     10      7     13     12     12     12      8     11     12
## 5    gene5    14      7      8      9     11     10     13     13      5     11
## 6    gene6    12     12     11     15      8      7     10      9     10     15
## 7    gene7    11     11     14     11     11      5      9     13     13      7
## 8    gene8     9     12      9      8      6     14      7     12     12     10
## 9    gene9    14     12     11      7     10     10      8     14      7     10
## 10   gene10   11     10      9      7     11     16      8      7      7      4
ggplot(data = counts, mapping = aes(x = cell1, y = cell2))

```



Let's take a closer look at the final command, `ggplot(data = counts, mapping = aes(x = cell1, y = cell2))`. `ggplot()` initialises a ggplot object and takes the arguments `data` and `mapping`. We pass our data frame of counts to `data` and use the `aes()` function to specify that we would like to use the variable `cell1` as our x variable and the variable `cell2` as our y variable.

Task 1: Modify the command above to initialise a ggplot object where cell10 is the x variable and cell8 is the y variable.

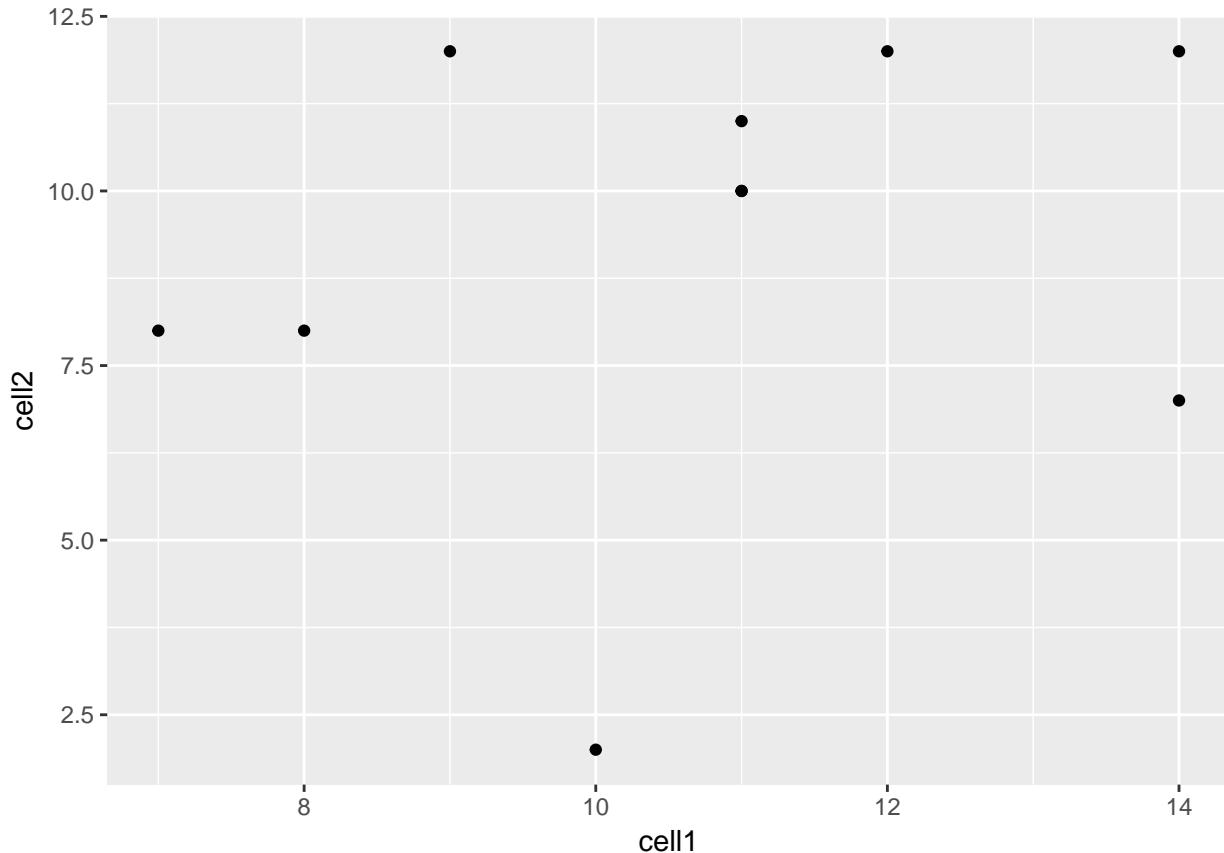
Clearly, the plots we have just created are not very informative because no data is displayed on them. To display data, we will need to use geoms.

#### 5.8.4 Geoms

We can use geoms to specify how we would like data to be displayed on our graphs. For example, our choice of geom could specify that we would like our data to be displayed as a scatterplot, a barplot or a boxplot.

Let's see how our graph would look as a scatterplot.

```
ggplot(data = counts, mapping = aes(x = cell1, y = cell2)) + geom_point()
```



Now we can see that there doesn't seem to be any correlation between gene expression in cell1 and cell2. Given we generated `counts` randomly, this isn't too surprising.

Task 2: Modify the command above to create a line plot. Hint: execute `?ggplot` and scroll down the help page. At the bottom is a link to the ggplot package index. Scroll through the index until you find the geom options.

#### 5.8.5 Plotting data from more than 2 cells

So far we've been considering the gene counts from 2 of the cells in our dataframe. But there are actually 10 cells in our dataframe and it would be nice to compare all of them. What if we wanted to plot data from all 10 cells at the same time?

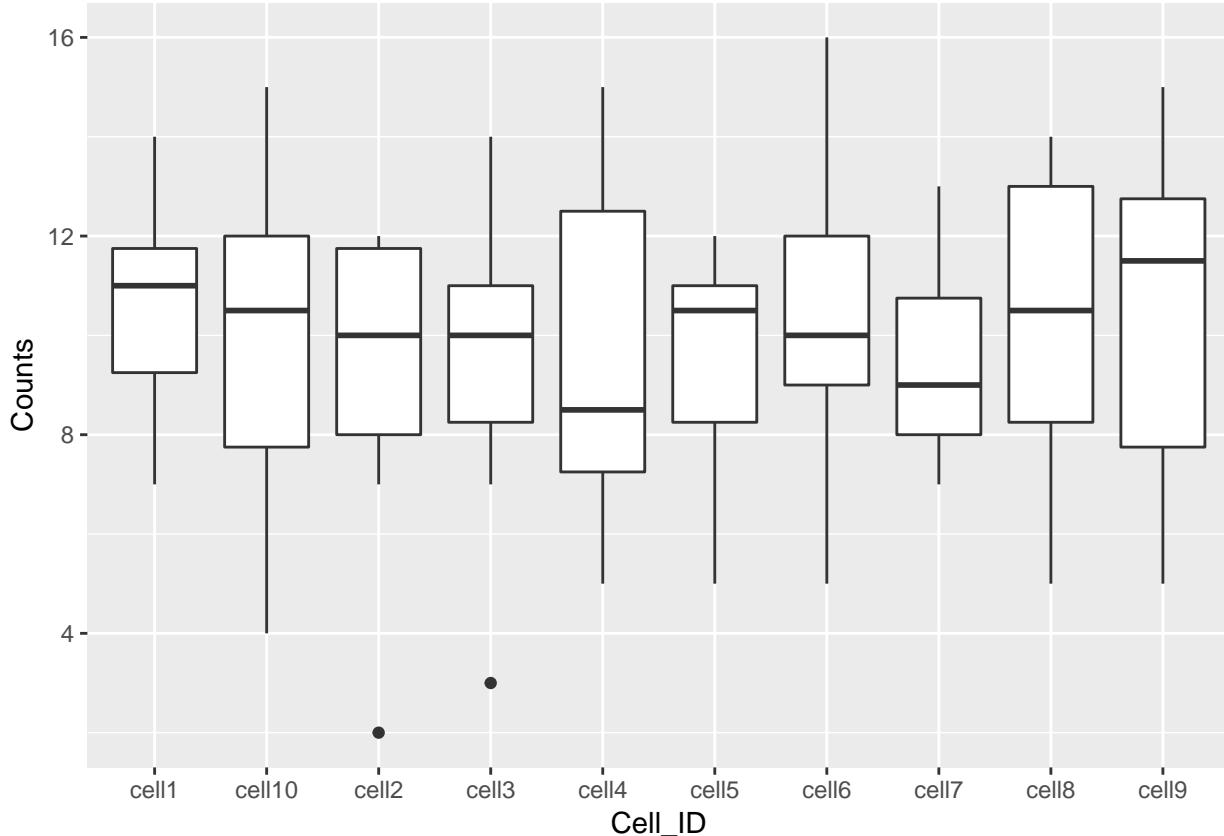
At the moment we can't do this because we are treating each individual cell as a variable and assigning that variable to either the x or the y axis. We could create a 10 dimensional graph to plot data from all 10 cells on, but this is a) not possible to do with ggplot and b) not very easy to interpret. What we could do instead is to tidy our data so that we had one variable representing cell ID and another variable representing gene counts, and plot those against each other. In code, this would look like:

```
counts<-gather(counts, colnames(counts)[2:11], key = 'Cell_ID', value='Counts')
head(counts)
```

```
##   Gene_ids Cell_ID Counts
## 1   gene1  cell1     8
## 2   gene2  cell1    10
## 3   gene3  cell1     7
## 4   gene4  cell1    11
## 5   gene5  cell1    14
## 6   gene6  cell1    12
```

Essentially, the problem before was that our data was not tidy because one variable (Cell\_ID) was spread over multiple columns. Now that we've fixed this problem, it is much easier for us to plot data from all 10 cells on one graph.

```
ggplot(counts,aes(x=Cell_ID, y=Counts)) + geom_boxplot()
```



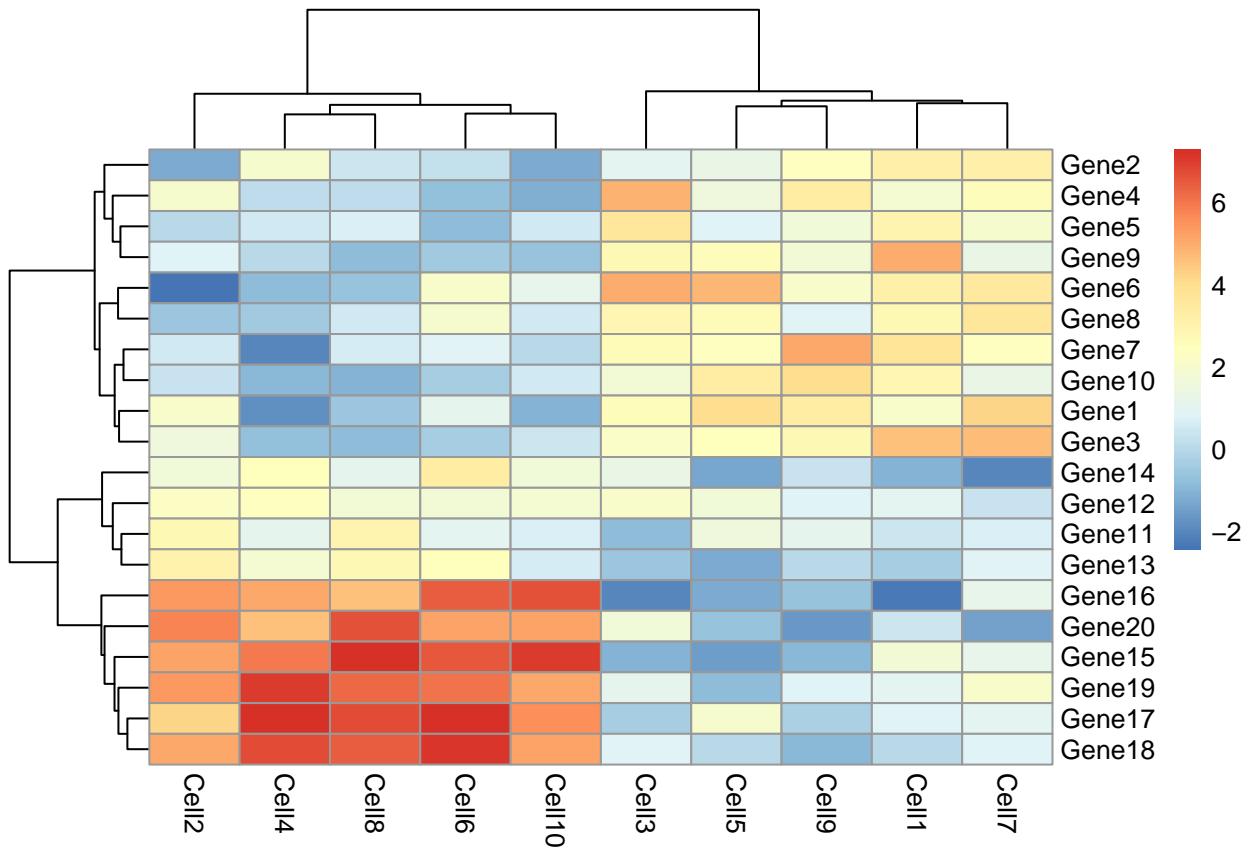
Task 3: Use the updated `counts` dataframe to plot a barplot with Cell\_ID as the x variable and Counts as the y variable. Hint: you may find it helpful to read `?geom_bar`.

Task 4: Use the updated `counts` dataframe to plot a scatterplot with Gene\_ids as the x variable and Counts as the y variable.

### 5.8.6 Plotting heatmaps

A common method for visualising gene expression data is with a heatmap. Here we will use the R package `pheatmap` to perform this analysis with some gene expression data we will name `test`.

```
library(pheatmap)
set.seed(2)
test = matrix(rnorm(200), 20, 10)
test[1:10, seq(1, 10, 2)] = test[1:10, seq(1, 10, 2)] + 3
test[11:20, seq(2, 10, 2)] = test[11:20, seq(2, 10, 2)] + 2
test[15:20, seq(2, 10, 2)] = test[15:20, seq(2, 10, 2)] + 4
colnames(test) = paste("Cell", 1:10, sep = "")
rownames(test) = paste("Gene", 1:20, sep = "")
pheatmap(test)
```



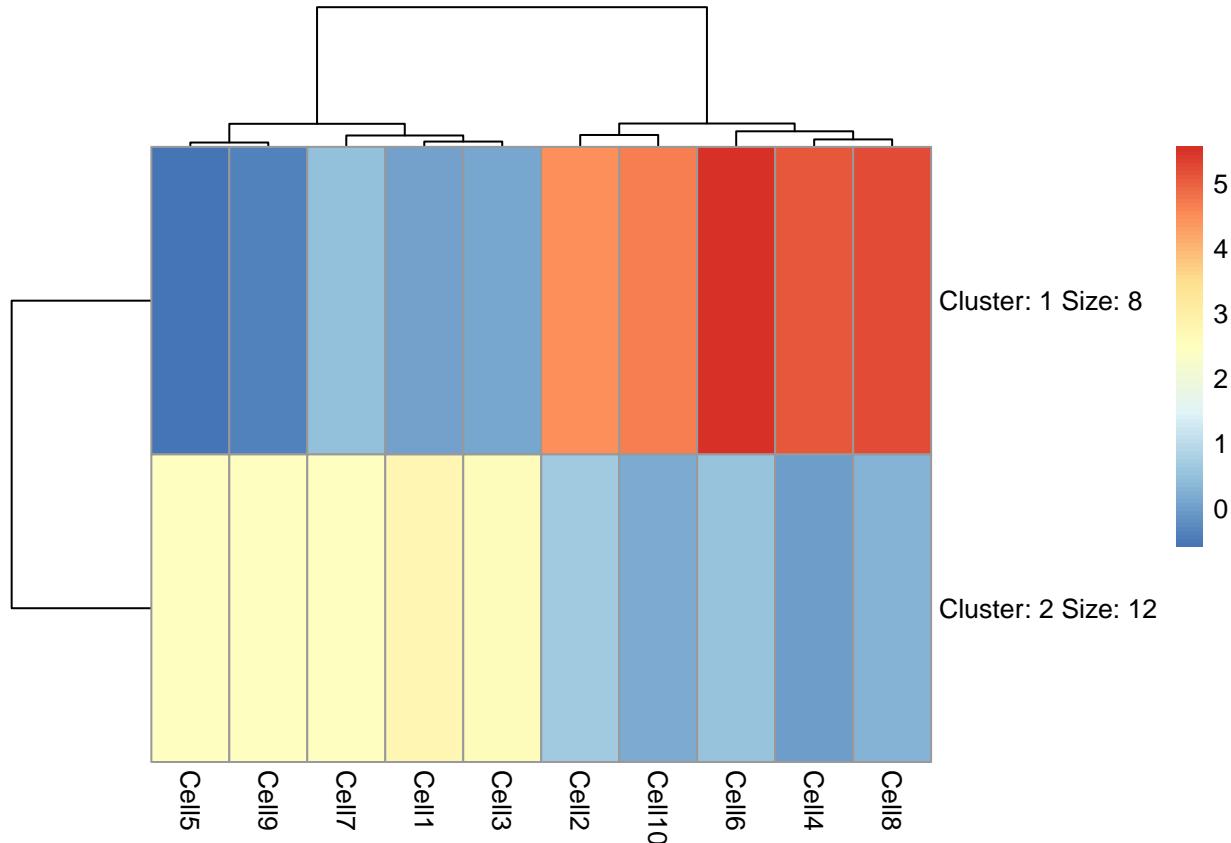
Let's take a moment to work out what this graphic is showing us. Each row represents a gene and each column represents a cell. How highly expressed each gene is in each cell is represented by the colour of the corresponding box. For example, we can tell from this plot that gene18 is highly expressed in cell10 but lowly expressed in cell1.

This plot also gives us information on the results of a clustering algorithm. In general, clustering algorithms aim to split datapoints (eg. cells) into groups whose members are more alike one another than they are alike the rest of the datapoints. The trees drawn on the top and left hand sides of the graph are the results of clustering algorithms and enable us to see, for example, that cells 4,8,2,6 and 10 are more alike one another than they are alike cells 7,3,5,1 and 9. The tree on the left hand side of the graph represents the results of a clustering algorithm applied to the genes in our dataset.

If we look closely at the trees, we can see that eventually they have the same number of branches as there are cells and genes. In other words, the total number of cell clusters is the same as the total number of

cells, and the total number of gene clusters is the same as the total number of genes. Clearly, this is not very informative, and will become impractical when we are looking at more than 10 cells and 20 genes. Fortunately, we can set the number of clusters we see on the plot. Let's try setting the number of gene clusters to 2:

```
pheatmap(test, kmeans_k = 2)
```



Now we can see that the genes fall into two clusters - a cluster of 8 genes which are upregulated in cells 2, 10, 6, 4 and 8 relative to the other cells and a cluster of 12 genes which are downregulated in cells 2, 10, 6, 4 and 8 relative to the other cells.

Task 5: Try setting the number of clusters to 3. Which number of clusters do you think is more informative?

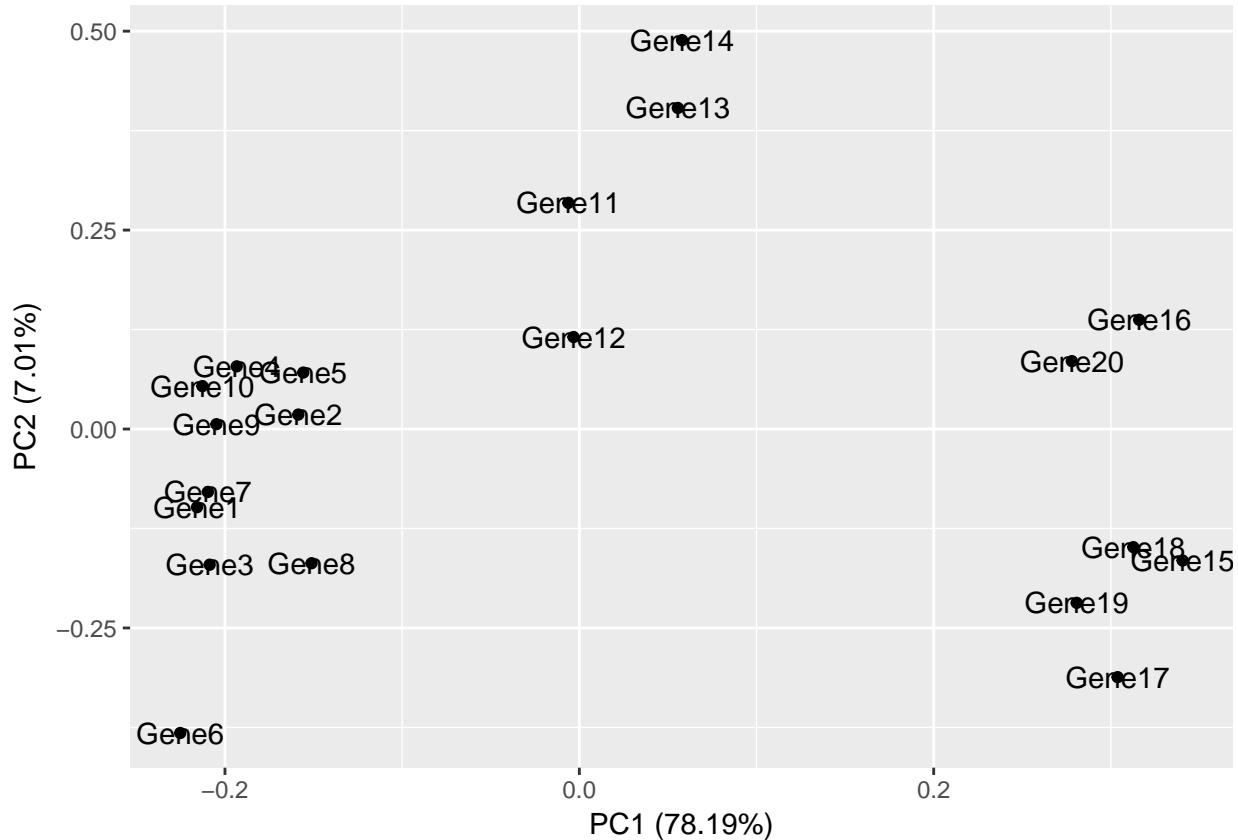
### 5.8.7 Principle Component Analysis

Principal component analysis (PCA) is a statistical procedure that uses a transformation to convert a set of observations into a set of values of linearly uncorrelated variables called principal components. The transformation is carried out so that the first principle component accounts for as much of the variability in the data as possible, and each following principle component accounts for the greatest amount of variance possible under the constraint that it must be orthogonal to the previous components.

PCA plots are a good way to get an overview of your data, and can sometimes help identify confounders which explain a high amount of the variability in your data. We will investigate how we can use PCA plots in single-cell RNA-seq analysis in more depth in a future lab, here the aim is to give you an overview of what PCA plots are and how they are generated.

Let's make a PCA plot for our `test` data. We can use the `ggfortify` package to let `ggplot` know how to interpret principle components.

```
library(ggfortify)
Principle_Components<-prcomp(test)
autoplot(Principle_Components, label=TRUE)
```



Task 6: Compare your clusters to the pheatmap clusters. Are they related? (Hint: have a look at the gene tree for the first pheatmap we plotted)

Task 7: Produce a heatmap and PCA plot for `counts` (below):

```
set.seed(1)
counts <- as.data.frame(matrix(rpois(100, lambda = 10), ncol=10, nrow=10))
rownames(counts) <- paste("gene", 1:10, sep = "")
colnames(counts) <- paste("cell", 1:10, sep = "")
```



# Chapter 6

## Tabula Muris

### 6.1 Introduction

To give you hands-on experience analyzing from start to finish a single-cell RNASeq dataset we will be using as an example, data from the Tabula Muris initial release. The Tabula Muris is an international collaboration with the aim to profile every cell-type in the mouse using a standardized method. They combine hightthroughput but low-coverage 10X data with lower throughput but high-coverage FACS-sorted cells + Smartseq2.

The initial release of the data (20 Dec 2017), contain almost 100,000 cells across 20 different tissues/organs. You have been assigned one of these tissues as an example to work on over this course, and on Friday each person will have 3 minutes to present the result for their tissue.

### 6.2 Downloading the data

Unlike most single-cell RNASeq data Tabula Muris has release their data through the figshare platform rather than uploading it to GEO or ArrayExpress. You can find the data by using the doi's in their paper : 10.6084/m9.figshare.5715040 for FACS/Smartseq2 and 10.6084/m9.figshare.5715025 for 10X data. The data can be downloaded manually by clinking the doi links or by using the command-line commands below:

Terminal-based download of FACS data:

```
wget https://ndownloader.figshare.com/files/10038307
unzip 10038307
wget https://ndownloader.figshare.com/files/10038310
mv 10038310 FACS_metadata.csv
wget https://ndownloader.figshare.com/files/10039267
mv 10039267 FACS_annotations.csv
```

Terminal-based download of 10X data:

```
wget https://ndownloader.figshare.com/files/10038325
unzip 10038325
wget https://ndownloader.figshare.com/files/10038328
mv 10038328 droplet_metadata.csv
wget https://ndownloader.figshare.com/files/10039264
mv 10039264 droplet_annotation.csv
```

Note if you download the data by hand you should unzip & rename the files as above before continuing.

You should now have two folders : “FACS” and “droplet” and one annotation and metadata file for each. To inspect these files you can use the `head` to see the top few lines of the text files (Press “q” to exit):

```
head -n 10 droplet_metadata.csv

## channel,mouse.id,tissue,subtissue,mouse.sex
## 10X_P4_0,3-M-8,Tongue,NA,M
## 10X_P4_1,3-M-9,Tongue,NA,M
## 10X_P4_2,3-M-8/9,Liver,hepatocytes,M
## 10X_P4_3,3-M-8,Bladder,NA,M
## 10X_P4_4,3-M-9,Bladder,NA,M
## 10X_P4_5,3-M-8,Kidney,NA,M
## 10X_P4_6,3-M-9,Kidney,NA,M
## 10X_P4_7,3-M-8,Spleen,NA,M
## 10X_P7_0,3-F-56,Liver,NA,F
```

You can also check the number of rows in each file using:

```
wc -l droplet_annotation.csv

## 54838 droplet_annotation.csv
```

**Exercise** How many cells do we have annotations for from FACS? from 10X?

**Answer** FACS : 54,838 cells Droplet : 42,193 cells

### 6.3 Reading the data (Smartseq2)

We can now read in the relevant count matrix from the comma-separated file. Then inspect the resulting dataframe:

```
dat = read.delim("FACS/Kidney-counts.csv", sep=",", header=TRUE)
dat[1:5,1:5]

##           X A14.MAA000545.3_8_M.1.1 E1.MAA000545.3_8_M.1.1
## 1 0610005C13Rik          0            0
## 2 0610007C21Rik          1            0
## 3 0610007L01Rik          0            0
## 4 0610007N19Rik          0            0
## 5 0610007P08Rik          0            0

##   M4.MAA000545.3_8_M.1.1 021.MAA000545.3_8_M.1.1
## 1                      0            0
## 2                      0            0
## 3                      0            0
## 4                      0            0
## 5                      0            0
```

We can see that the first column in the dataframe is the gene names, so first we move these to the rownames so we have a numeric matrix:

```
dim(dat)

## [1] 23433    866
rownames(dat) <- dat[,1]
dat <- dat[,-1]
```

Since this is a Smartseq2 dataset it may contain spike-ins so lets check:

```
rownames(dat)[grep("ERCC-", rownames(dat))]

## [1] "ERCC-00002" "ERCC-00003" "ERCC-00004" "ERCC-00009" "ERCC-00012"
## [6] "ERCC-00013" "ERCC-00014" "ERCC-00016" "ERCC-00017" "ERCC-00019"
## [11] "ERCC-00022" "ERCC-00024" "ERCC-00025" "ERCC-00028" "ERCC-00031"
## [16] "ERCC-00033" "ERCC-00034" "ERCC-00035" "ERCC-00039" "ERCC-00040"
## [21] "ERCC-00041" "ERCC-00042" "ERCC-00043" "ERCC-00044" "ERCC-00046"
## [26] "ERCC-00048" "ERCC-00051" "ERCC-00053" "ERCC-00054" "ERCC-00057"
## [31] "ERCC-00058" "ERCC-00059" "ERCC-00060" "ERCC-00061" "ERCC-00062"
## [36] "ERCC-00067" "ERCC-00069" "ERCC-00071" "ERCC-00073" "ERCC-00074"
## [41] "ERCC-00075" "ERCC-00076" "ERCC-00077" "ERCC-00078" "ERCC-00079"
## [46] "ERCC-00081" "ERCC-00083" "ERCC-00084" "ERCC-00085" "ERCC-00086"
## [51] "ERCC-00092" "ERCC-00095" "ERCC-00096" "ERCC-00097" "ERCC-00098"
## [56] "ERCC-00099" "ERCC-00104" "ERCC-00108" "ERCC-00109" "ERCC-00111"
## [61] "ERCC-00112" "ERCC-00113" "ERCC-00116" "ERCC-00117" "ERCC-00120"
## [66] "ERCC-00123" "ERCC-00126" "ERCC-00130" "ERCC-00131" "ERCC-00134"
## [71] "ERCC-00136" "ERCC-00137" "ERCC-00138" "ERCC-00142" "ERCC-00143"
## [76] "ERCC-00144" "ERCC-00145" "ERCC-00147" "ERCC-00148" "ERCC-00150"
## [81] "ERCC-00154" "ERCC-00156" "ERCC-00157" "ERCC-00158" "ERCC-00160"
## [86] "ERCC-00162" "ERCC-00163" "ERCC-00164" "ERCC-00165" "ERCC-00168"
## [91] "ERCC-00170" "ERCC-00171"
```

Now we can extract much of the metadata for this data from the column names:

```
cellIDs <- colnames(dat)
cell_info <- strsplit(cellIDs, "\\.")
Well <- lapply(cell_info, function(x){x[1]})
Well <- unlist(Well)
Plate <- unlist(lapply(cell_info, function(x){x[2]}))
Mouse <- unlist(lapply(cell_info, function(x){x[3]}))
```

We can check the distributions of each of these metadata classifications:

```
summary(factor(Mouse))
```

```
## 3_10_M 3_11_M 3_38_F 3_39_F 3_8_M 3_9_M
##     104     196     237     169      82      77
```

We can also check if any technical factors are confounded:

```
table(Mouse, Plate)
```

```
##          Plate
## Mouse      B001717 B002775 MAA000545 MAA000752 MAA000801 MAA000922
##   3_10_M      0      0      0     104      0      0
##   3_11_M      0      0      0      0     196      0
##   3_38_F     237      0      0      0      0      0
##   3_39_F      0    169      0      0      0      0
##   3_8_M       0      0     82      0      0      0
##   3_9_M       0      0      0      0      0     77
```

Lastly we will read the computationally inferred cell-type annotation and match them to the cell in our expression matrix:

```
ann <- read.table("FACS_annotations.csv", sep=",", header=TRUE)
ann <- ann[match(cellIDs, ann[,1]),]
celltype <- ann[,3]
```

## 6.4 Building a scater object

To create a SingleCellExperiment object we must put together all the cell annotations into a single dataframe, since the experimental batch (pcr plate) is completely confounded with donor mouse we will only keep one of them.

```
require("SingleCellExperiment")

## Loading required package: SingleCellExperiment
## Loading required package: SummarizedExperiment
## Loading required package: methods
## Loading required package: GenomicRanges
## Loading required package: stats4
## Loading required package: BiocGenerics
## Loading required package: parallel
##
## Attaching package: 'BiocGenerics'

## The following objects are masked from 'package:parallel':
## 
##     clusterApply, clusterApplyLB, clusterCall, clusterEvalQ,
##     clusterExport, clusterMap, parApply, parCapply, parLapply,
##     parLapplyLB, parRapply, parSapply, parSapplyLB

## The following objects are masked from 'package:stats':
## 
##     IQR, mad, sd, var, xtabs

## The following objects are masked from 'package:base':
## 
##     anyDuplicated, append, as.data.frame, cbind, colMeans,
##     colnames, colSums, do.call, duplicated, eval, evalq, Filter,
##     Find, get, grep, grepl, intersect, is.unsorted, lapply,
##     lengths, Map, mapply, match, mget, order, paste, pmax,
##     pmax.int, pmin, pmin.int, Position, rank, rbind, Reduce,
##     rowMeans, rownames, rowSums, sapply, setdiff, sort, table,
##     tapply, union, unique, unsplit, which, which.max, which.min

## Loading required package: S4Vectors

##
## Attaching package: 'S4Vectors'

## The following object is masked from 'package:base':
## 
##     expand.grid

## Loading required package: IRanges
## Loading required package: GenomeInfoDb
## Loading required package: Biobase
## Welcome to Bioconductor
##
##     Vignettes contain introductory material; view with
```

```

##      'browseVignettes()'. To cite Bioconductor, see
##      'citation("Biobase")', and for packages 'citation("pkgname")'.

## Loading required package: DelayedArray
## Loading required package: matrixStats
##
## Attaching package: 'matrixStats'
## The following objects are masked from 'package:Biobase':
##       anyMissing, rowMedians
##
## Attaching package: 'DelayedArray'
## The following objects are masked from 'package:matrixStats':
##       colMaxs, colMins, colRanges, rowMaxs, rowMins, rowRanges
## The following object is masked from 'package:base':
##       apply
require("scater")

## Loading required package: scater
## Loading required package: ggplot2
##
## Attaching package: 'scater'
## The following object is masked from 'package:S4Vectors':
##       rename
## The following object is masked from 'package:stats':
##       filter

cell_anns <- data.frame(mouse = Mouse, well=Well, type=celltype)
rownames(cell_anns) <- colnames(dat)
sceset <- SingleCellExperiment(assays = list(counts = as.matrix(dat)), colData=cell_anns)

```

Finally if the dataset contains spike-ins we a hidden variable in the SingleCellExperiment object to track them:

```
isSpike(sceset, "ERCC") <- grep("ERCC-", rownames(sceset))
```

## 6.5 Reading the data (10X)

Due to the large size and sparsity of 10X data (upto 90% of the expression matrix may be 0s) it is typically stored as a sparse matrix. The default output format for CellRanger is an .mtx file which stores this sparse matrix as a column of row coordinates, a column of column coordinates, and a column of expression values  $> 0$ . Note if you look at the .mtx file you will see two header lines followed by a line detailing the total number of rows, columns and counts for the full matrix. Since only the coordinates are stored in the .mtx file, the names of each row & column must be stored separately in the “genes.tsv” and “barcodes.tsv” files respectively.

We will be using the “Matrix” package to store matrices in sparse-matrix format in R.

```
require("Matrix")

## Loading required package: Matrix

##
## Attaching package: 'Matrix'

## The following object is masked from 'package:S4Vectors':
##
##      expand

cellbarcodes <- read.table("droplet/Kidney-10X_P4_5/barcodes.tsv")
genenames <- read.table("droplet/Kidney-10X_P4_5/genes.tsv")
molecules <- Matrix:::readMM("droplet/Kidney-10X_P4_5/matrix.mtx")
```

Now we will add the appropriate row and column names. However, if you inspect the read cellbarcodes you will see that they are just the barcode sequence associated with each cell. This is a problem since each batch of 10X data uses the same pool of barcodes so if we need to combine data from multiple 10X batches the cellbarcodes will not be unique. Hence we will attach the batch ID to each cell barcode:

```
head(cellbarcodes)

##           V1
## 1 AACCTGAGATGCCAG-1
## 2 AACCTGAGTGTCCAT-1
## 3 AACCTGCAAGGCTCC-1
## 4 AACCTGTCCTTGCCA-1
## 5 AACGGGAGCTAACG-1
## 6 AACGGGCAGGACCT-1

rownames(molecules) <- genenames[,1]
colnames(molecules) <- paste("10X_P4_5", cellbarcodes[,1], sep="_")
```

Now lets get the metadata and computational annotations for this data:

```
meta <- read.delim("droplet_metadata.csv", sep=",", header=TRUE)
head(meta)

##   channel mouse.id tissue subtissue mouse.sex
## 1 10X_P4_0   3-M-8  Tongue        <NA>       M
## 2 10X_P4_1   3-M-9  Tongue        <NA>       M
## 3 10X_P4_2  3-M-8/9 Liver hepatocytes       M
## 4 10X_P4_3   3-M-8 Bladder       <NA>       M
## 5 10X_P4_4   3-M-9 Bladder       <NA>       M
## 6 10X_P4_5   3-M-8  Kidney       <NA>       M
```

Here we can see that we need to use “10X\_P4\_5” to find the metadata for this batch, also note that the format of the mouse ID is different in this metadata table with hyphens instead of underscores and with the gender in the middle of the ID. From checking the methods section of the accompanying paper we know that the same 8 mice were used for both droplet and plate-based techniques. So we need to fix the mouse IDs to be consistent with those used in the FACS experiments.

```
meta[meta$channel == "10X_P4_5",]

##   channel mouse.id tissue subtissue mouse.sex
## 6 10X_P4_5   3-M-8  Kidney        <NA>       M
```

```
mouseID <- "3_8_M"
```

Note: depending on the tissue you have been assigned you may have 10X data from mixed samples : e.g. mouse id = 3-M-5/6. You should still reformat these to be consistent but they will not match mouse ids from the FACS data which may affect your downstream analysis. If the mice weren't from an inbred strain it would be possible to assign individual cells to a specific mouse using exonic-SNPs but that is beyond the scope of this course.

```
ann <- read.delim("droplet_annotation.csv", sep=",", header=TRUE)
head(ann)
```

```
##           cell  tissue cell_ontology_class
## 1 10X_P4_3_AAAGTAGAGATGCCAG Bladder    mesenchymal cell
## 2 10X_P4_3_AACCGCGTCCAACCAA Bladder    mesenchymal cell
## 3 10X_P4_3_AACTCCCGTCGGGTCT Bladder    mesenchymal cell
## 4 10X_P4_3_AACTCTTAGTTGCAGG Bladder    bladder cell
## 5 10X_P4_3_AACTCTTCATAACCG Bladder    mesenchymal cell
## 6 10X_P4_3_AAGACCTAGATCCGAG Bladder    endothelial cell
##           cell_ontology_term_iri cell_ontology_id
## 1 http://purl.obolibrary.org/obo/CL_0008019      CL:0008019
## 2 http://purl.obolibrary.org/obo/CL_0008019      CL:0008019
## 3 http://purl.obolibrary.org/obo/CL_0008019      CL:0008019
## 4 http://purl.obolibrary.org/obo/CL_1001319      CL:1001319
## 5 http://purl.obolibrary.org/obo/CL_0008019      CL:0008019
## 6 http://purl.obolibrary.org/obo/CL_0000115      CL:0000115
```

Again you will find a slight formating difference between the cellID in the annotation and the cellbarcodes which we will have to correct before matching them.

```
ann[,1] <- paste(ann[,1], "-1", sep="")
ann_subset <- ann[match(colnames(molecules), ann[,1]),]
celltype <- ann_subset[,3]
```

Now lets build the cell-metadata dataframe:

```
cell_anns <- data.frame(mouse = rep(mouseID, times=ncol(molecules)), type=celltype)
rownames(cell_anns) <- colnames(molecules);
```

**Exercise** Repeat the above for the other 10X batches for your tissue.

**Answer**

## 6.6 Building a scater object

Now that we have read the 10X data in multiple batches we need to combine them into a single SingleCellExperiment object. First we will check that the gene names are the same and in the same order across all batches:

```
identical(rownames(molecules1), rownames(molecules2))
```

```
## [1] TRUE
identical(rownames(molecules1), rownames(molecules3))
```

```
## [1] TRUE
```

Now we'll check that there aren't any repeated cellIDs:

```
sum(colnames(molecules1) %in% colnames(molecules2))
## [1] 0
sum(colnames(molecules1) %in% colnames(molecules3))
## [1] 0
sum(colnames(molecules2) %in% colnames(molecules3))
## [1] 0
```

Everything is ok, so we can go ahead and combine them:

```
all_molecules <- cbind(molecules1, molecules2, molecules3)
all_cell_anns <- as.data.frame(rbind(cell_anns1, cell_anns2, cell_anns3))
all_cell_anns$batch <- rep(c("10X_P4_5", "10X_P4_6", "10X_P7_5"), times = c(nrow(cell_anns1), nrow(cell_anns2), nrow(cell_anns3)))
```

**Exercise** How many cells are in the whole dataset?

**Answer**

Now build the SingleCellExperiment object. One of the advantages of the SingleCellExperiment class is that it is capable of storing data in normal matrix or sparse matrix format, as well as HDF5 format which allows large non-sparse matrices to be stored & accessed on disk in an efficient manner rather than loading the whole thing into RAM.

```
require("SingleCellExperiment")
require("scater")
all_molecules <- as.matrix(all_molecules)
sceset <- SingleCellExperiment(assays = list(counts = as.matrix(all_molecules)), colData=all_cell_anns)
```

Since this is 10X data it will not contain spike-ins, so we just save the data:

```
saveRDS(sceset, "kidney_droplet.rds")
```

## 6.7 Advanced Exercise

Write an R function/script which will fully automate this procedure for each data-type for any tissue.

# Chapter 7

## Cleaning the Expression Matrix

### 7.1 Expression QC (UMI)

#### 7.1.1 Introduction

Once gene expression has been quantified it is summarized as an **expression matrix** where each row corresponds to a gene (or transcript) and each column corresponds to a single cell. This matrix should be examined to remove poor quality cells which were not detected in either read QC or mapping QC steps. Failure to remove low quality cells at this stage may add technical noise which has the potential to obscure the biological signals of interest in the downstream analysis.

Since there is currently no standard method for performing scRNASeq the expected values for the various QC measures that will be presented here can vary substantially from experiment to experiment. Thus, to perform QC we will be looking for cells which are outliers with respect to the rest of the dataset rather than comparing to independent quality standards. Consequently, care should be taken when comparing quality metrics across datasets collected using different protocols.

#### 7.1.2 Tung dataset

To illustrate cell QC, we consider a dataset of induced pluripotent stem cells generated from three different individuals (Tung et al., 2017) in Yoav Gilad's lab at the University of Chicago. The experiments were carried out on the Fluidigm C1 platform and to facilitate the quantification both unique molecular identifiers (UMIs) and ERCC *spike-ins* were used. The data files are located in the `tung` folder in your working directory. These files are the copies of the original files made on the 15/03/16. We will use these copies for reproducibility purposes.

```
library(SingleCellExperiment)
library(scater)
options(stringsAsFactors = FALSE)
```

Load the data and annotations:

```
molecules <- read.table("tung/molecules.txt", sep = "\t")
anno <- read.table("tung/annotation.txt", sep = "\t", header = TRUE)
```

Inspect a small portion of the expression matrix

```
head(molecules[ , 1:3])
```

```

##          NA19098.r1.A01 NA19098.r1.A02 NA19098.r1.A03
## ENSG00000237683          0          0          0
## ENSG00000187634          0          0          0
## ENSG00000188976          3          6          1
## ENSG00000187961          0          0          0
## ENSG00000187583          0          0          0
## ENSG00000187642          0          0          0

head(anno)

```

```

##   individual replicate well      batch sample_id
## 1    NA19098         r1   A01 NA19098.r1 NA19098.r1.A01
## 2    NA19098         r1   A02 NA19098.r1 NA19098.r1.A02
## 3    NA19098         r1   A03 NA19098.r1 NA19098.r1.A03
## 4    NA19098         r1   A04 NA19098.r1 NA19098.r1.A04
## 5    NA19098         r1   A05 NA19098.r1 NA19098.r1.A05
## 6    NA19098         r1   A06 NA19098.r1 NA19098.r1.A06

```

The data consists of 3 individuals and 3 replicates and therefore has 9 batches in total.

We standardize the analysis by using both `SingleCellExperiment` (SCE) and `scater` packages. First, create the SCE object:

```

umi <- SingleCellExperiment(
  assays = list(counts = as.matrix(molecules)),
  colData = anno
)

```

Remove genes that are not expressed in any cell:

```

keep_feature <- rowSums(counts(umi) > 0) > 0
umi <- umi[keep_feature, ]

```

Define control features (genes) - ERCC spike-ins and mitochondrial genes (provided by the authors):

```

isSpike(umi, "ERCC") <- grepl("ERCC-", rownames(umi))
isSpike(umi, "MT") <- rownames(umi) %in%
  c("ENSG00000198899", "ENSG00000198727", "ENSG00000198888",
  "ENSG00000198886", "ENSG00000212907", "ENSG00000198786",
  "ENSG00000198695", "ENSG00000198712", "ENSG00000198804",
  "ENSG00000198763", "ENSG00000228253", "ENSG00000198938",
  "ENSG00000198840")

```

Calculate the quality metrics:

```

umi <- calculateQCMetrics(
  umi,
  feature_controls = list(
    ERCC = isSpike(umi, "ERCC"),
    MT = isSpike(umi, "MT")
  )
)

```

### Histogram of umi\$total\_counts

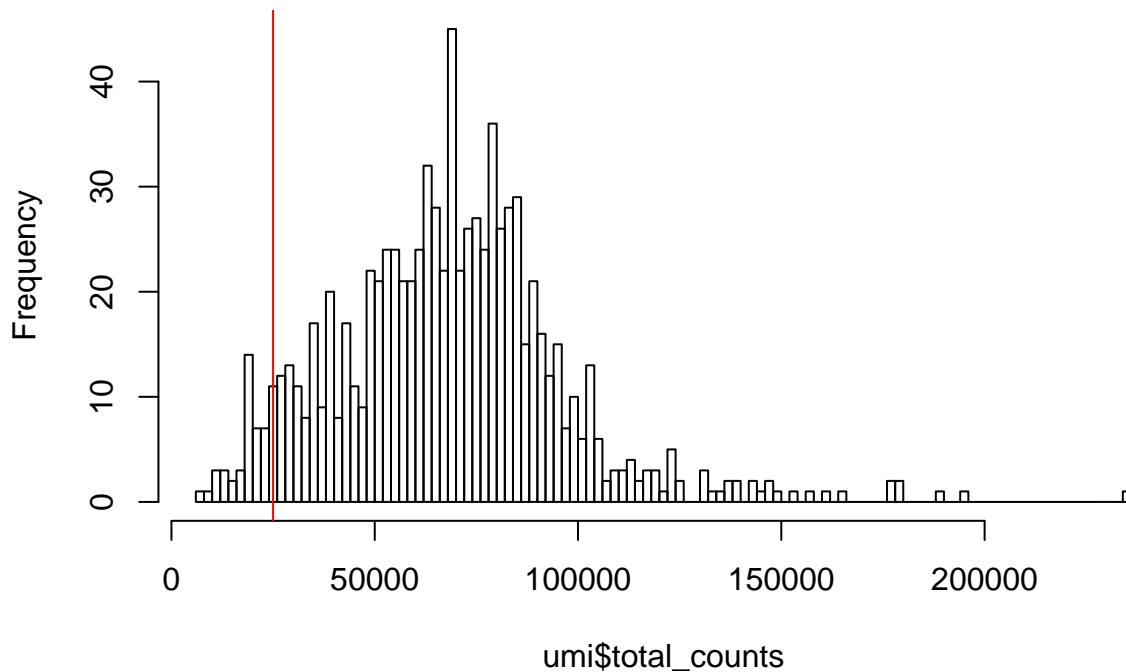


Figure 7.1: Histogram of library sizes for all cells

#### 7.1.3 Cell QC

##### 7.1.3.1 Library size

Next we consider the total number of RNA molecules detected per sample (if we were using read counts rather than UMI counts this would be the total number of reads). Wells with few reads/molecules are likely to have been broken or failed to capture a cell, and should thus be removed.

```
hist(
  umi$total_counts,
  breaks = 100
)
abline(v = 25000, col = "red")
```

##### Exercise 1

1. How many cells does our filter remove?
2. What distribution do you expect that the total number of molecules for each cell should follow?

##### Our answer

```
## filter_by_total_counts
## FALSE TRUE
##    46    818
```

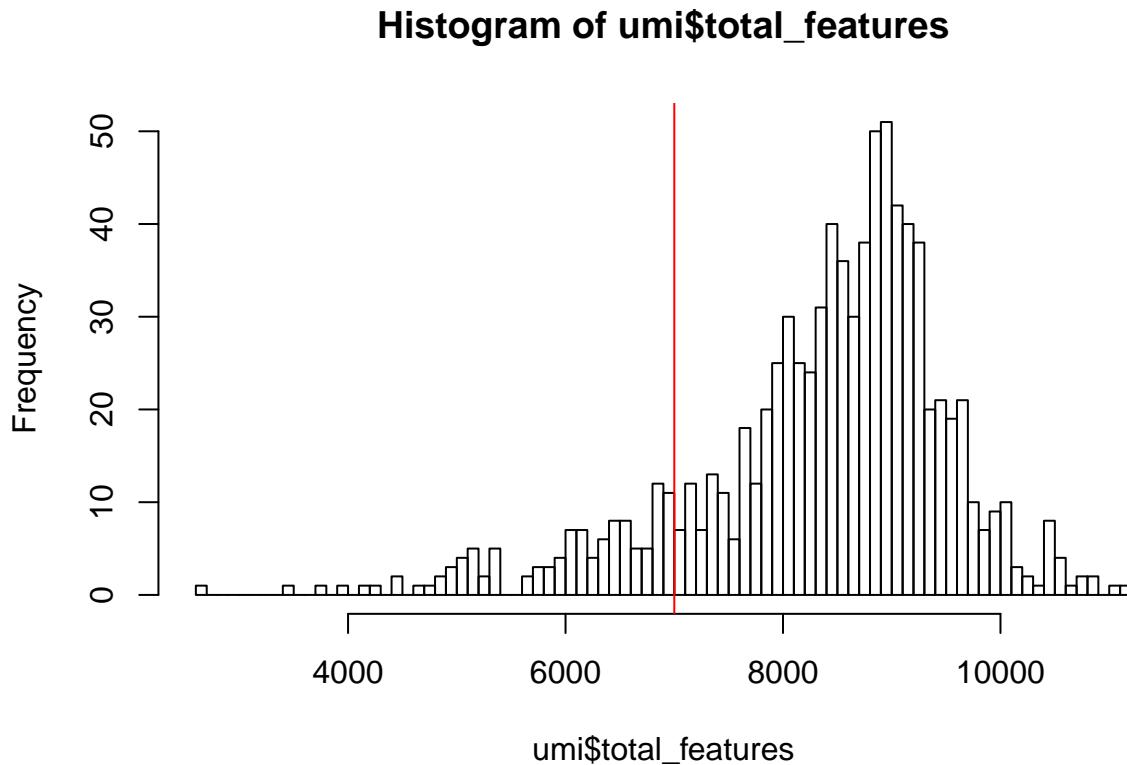


Figure 7.2: Histogram of the number of detected genes in all cells

### 7.1.3.2 Detected genes

In addition to ensuring sufficient sequencing depth for each sample, we also want to make sure that the reads are distributed across the transcriptome. Thus, we count the total number of unique genes detected in each sample.

```
hist(
  umi$total_features,
  breaks = 100
)
abline(v = 7000, col = "red")
```

From the plot we conclude that most cells have between 7,000-10,000 detected genes, which is normal for high-depth scRNA-seq. However, this varies by experimental protocol and sequencing depth. For example, droplet-based methods or samples with lower sequencing-depth typically detect fewer genes per cell. The most notable feature in the above plot is the “heavy tail” on the left hand side of the distribution. If detection rates were equal across the cells then the distribution should be approximately normal. Thus we remove those cells in the tail of the distribution (fewer than 7,000 detected genes).

#### Exercise 2

How many cells does our filter remove?

#### Our answer

```
## filter_by_expr_features
## FALSE TRUE
##    116    748
```

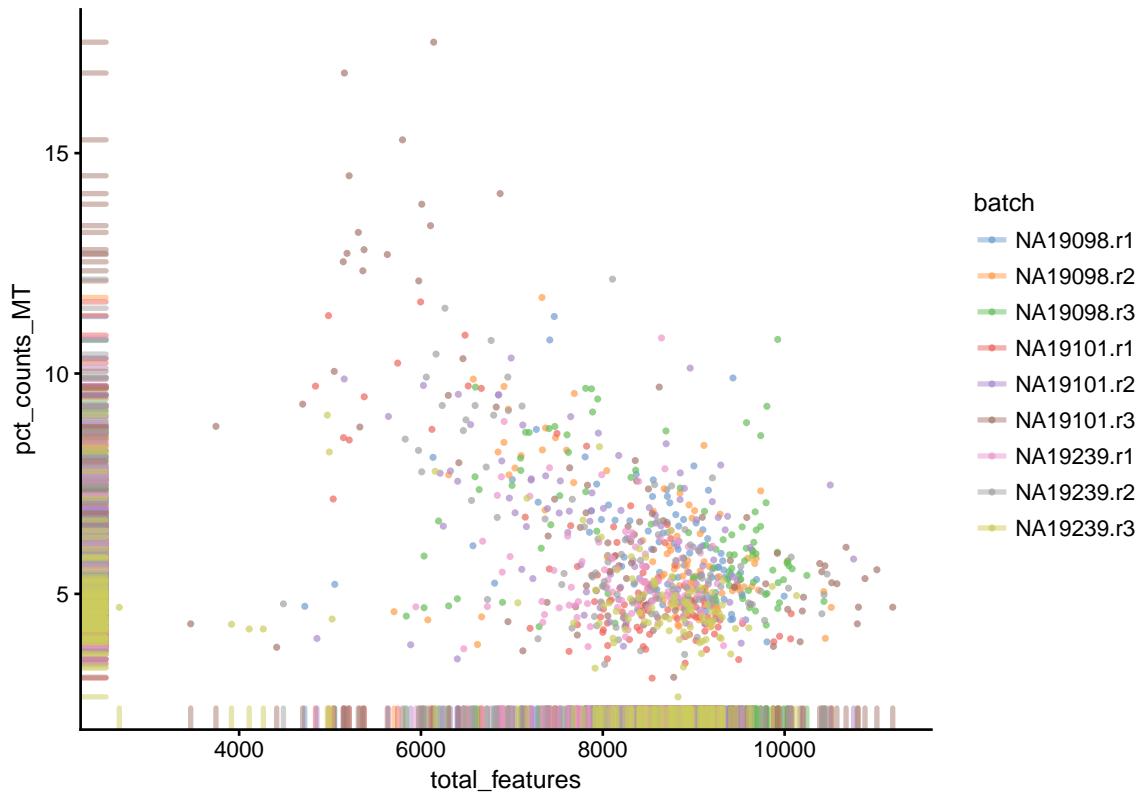


Figure 7.3: Percentage of counts in MT genes

#### 7.1.3.3 ERCCs and MTs

Another measure of cell quality is the ratio between ERCC *spike-in* RNAs and endogenous RNAs. This ratio can be used to estimate the total amount of RNA in the captured cells. Cells with a high level of *spike-in* RNAs had low starting amounts of RNA, likely due to the cell being dead or stressed which may result in the RNA being degraded.

```
plotPhenoData(
  umi,
  aes_string(
    x = "total_features",
    y = "pct_counts_MT",
    colour = "batch"
  )
)

plotPhenoData(
  umi,
  aes_string(
    x = "total_features",
    y = "pct_counts_ERCC",
    colour = "batch"
  )
)
```

The above analysis shows that majority of the cells from NA19098.r2 batch have a very high ERCC/Endo ratio. Indeed, it has been shown by the authors that this batch contains cells of smaller size.

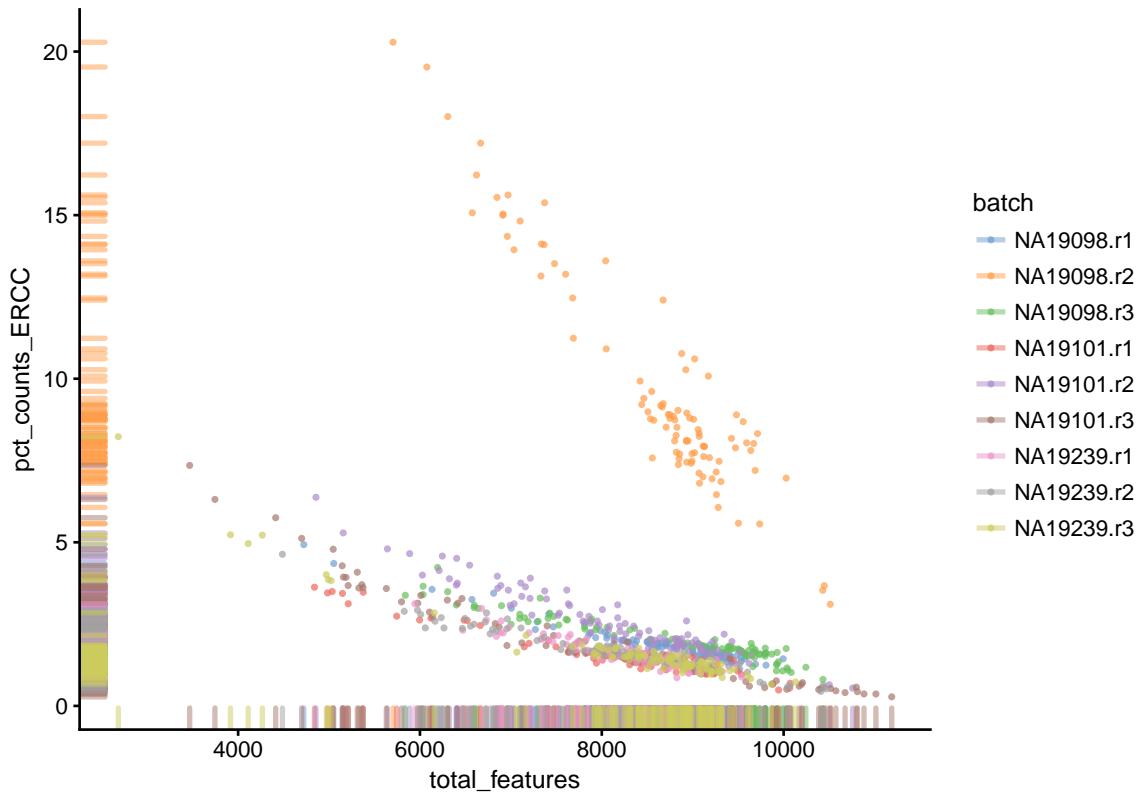


Figure 7.4: Percentage of counts in ERCCs

### Exercise 3

Create filters for removing batch NA19098.r2 and cells with high expression of mitochondrial genes (>10% of total counts in a cell).

### Our answer

```
## filter_by_ERCC
## FALSE TRUE
##    96   768

## filter_by_MT
## FALSE TRUE
##    31   833
```

### Exercise 4

What would you expect to see in the ERCC vs counts plot if you were examining a dataset containing cells of different sizes (eg. normal & senescent cells)?

### Answer

You would expect to see a group corresponding to the smaller cells (normal) with a higher fraction of ERCC reads than a separate group corresponding to the larger cells (senescent).

### 7.1.4 Cell filtering

#### 7.1.4.1 Manual

Now we can define a cell filter based on our previous analysis:

```
umi$use <- (
  # sufficient features (genes)
  filter_by_expr_features &
  # sufficient molecules counted
  filter_by_total_counts &
  # sufficient endogenous RNA
  filter_by_ERCC &
  # remove cells with unusual number of reads in MT genes
  filter_by_MT
)



```

#### 7.1.4.2 Automatic

Another option available in `scater` is to conduct PCA on a set of QC metrics and then use automatic outlier detection to identify potentially problematic cells.

By default, the following metrics are used for PCA-based outlier detection:

- `pct_counts_top_100_features`
- `total_features`
- `pct_counts_feature_controls`
- `n_detected_feature_controls`
- `log10_counts_endogenous_features`
- `log10_counts_feature_controls`

`scater` first creates a matrix where the rows represent cells and the columns represent the different QC metrics. Here, the PCA plot provides a 2D representation of cells ordered by their quality metrics. The outliers are then detected using methods from the `mvoutlier` package.

```
umi <- plotPCA(
  umi,
  size_by = "total_features",
  shape_by = "use",
  pca_data_input = "pdata",
  detect_outliers = TRUE,
  return_SCE = TRUE
)



```

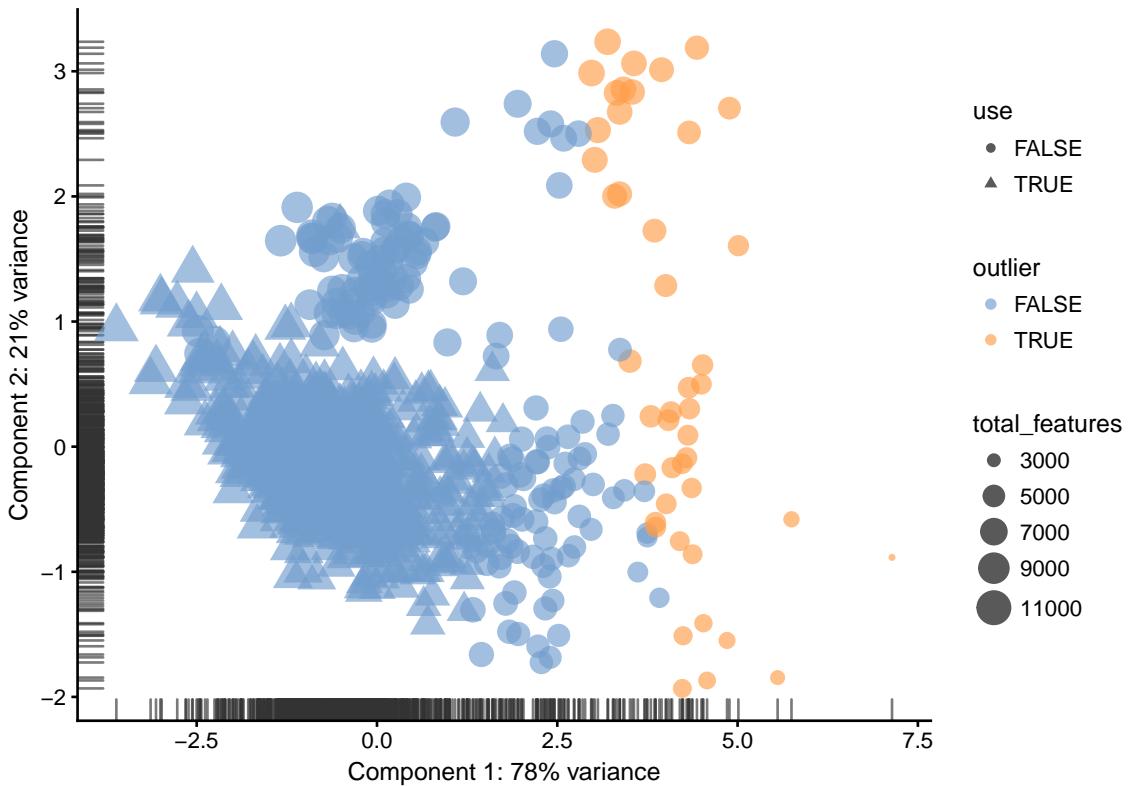


Figure 7.5: PCA plot used for automatic detection of cell outliers

### 7.1.5 Compare filterings

#### Exercise 5

Compare the default, automatic and manual cell filters. Plot a Venn diagram of the outlier cells from these filterings.

**Hint:** Use `vennCounts` and `vennDiagram` functions from the `limma` package to make a Venn diagram.

#### Answer

```
##  
## Attaching package: 'limma'  
  
## The following object is masked from 'package:scater':  
##  
##     plotMDS  
  
## The following object is masked from 'package:BiocGenerics':  
##  
##     plotMA
```

### 7.1.6 Gene analysis

#### 7.1.6.1 Gene expression

In addition to removing cells with poor quality, it is usually a good idea to exclude genes where we suspect that technical artefacts may have skewed the results. Moreover, inspection of the gene expression profiles

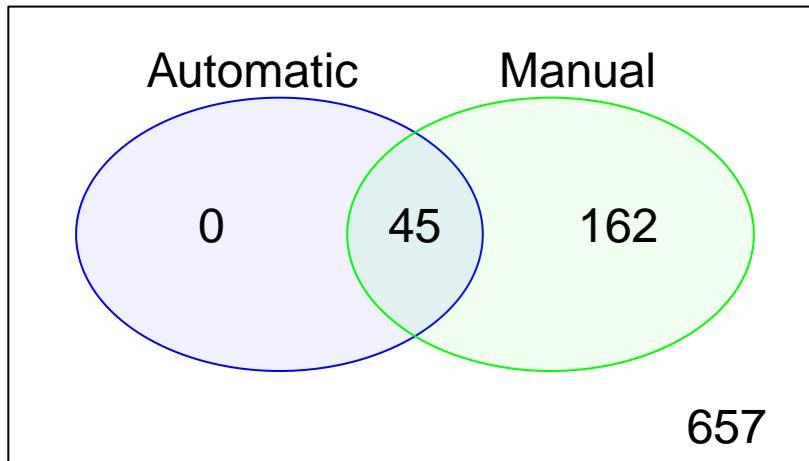


Figure 7.6: Comparison of the default, automatic and manual cell filters

may provide insights about how the experimental procedures could be improved.

It is often instructive to consider the number of reads consumed by the top 50 expressed genes.

```
plotQC(umi, type = "highest-expression")
```

The distributions are relatively flat indicating (but not guaranteeing!) good coverage of the full transcriptome of these cells. However, there are several spike-ins in the top 15 genes which suggests a greater dilution of the spike-ins may be preferable if the experiment is to be repeated.

#### 7.1.6.2 Gene filtering

It is typically a good idea to remove genes whose expression level is considered “**undetectable**”. We define a gene as detectable if at least two cells contain more than 1 transcript from the gene. If we were considering read counts rather than UMI counts a reasonable threshold is to require at least five reads in at least two cells. However, in both cases the threshold strongly depends on the sequencing depth. It is important to keep in mind that genes must be filtered after cell filtering since some genes may only be detected in poor quality cells (**note** `colData(umi)$use` filter applied to the `umi` dataset).

```
filter_genes <- apply(
  counts(umi[ , colData(umi)$use]),
  1,
  function(x) length(x[x > 1]) >= 2
)
rowData(umi)$use <- filter_genes

table(filter_genes)

## filter_genes
## FALSE  TRUE
## 4660 14066
```

Depending on the cell-type, protocol and sequencing depth, other cut-offs may be appropriate.

#### 7.1.7 Save the data

Dimensions of the QCed dataset (do not forget about the gene filter we defined above):

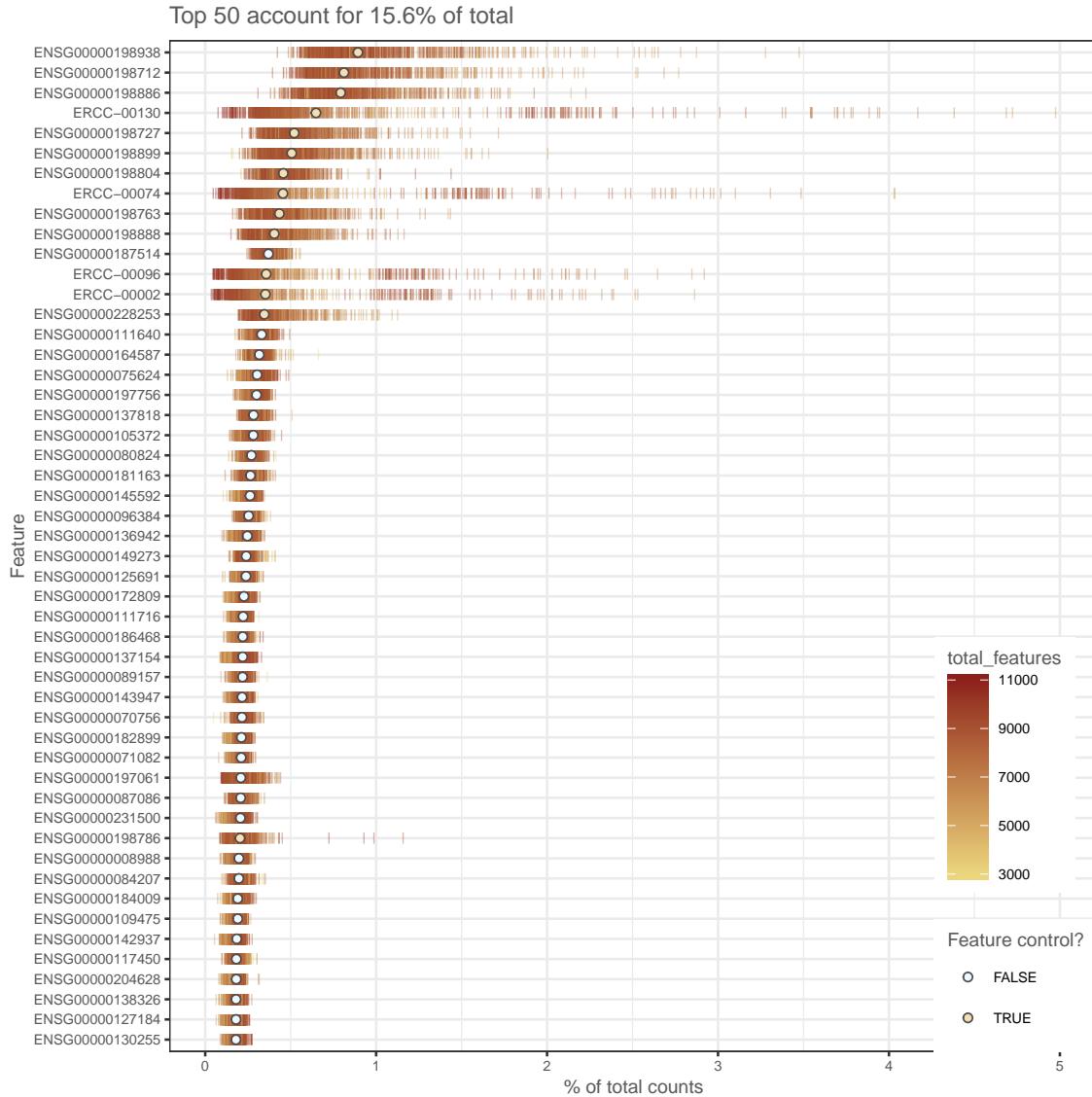


Figure 7.7: Number of total counts consumed by the top 50 expressed genes

```
dim(umi$rowData(umi)$use, colData(umi)$use])
## [1] 14066   657
```

Let's create an additional slot with log-transformed counts (we will need it in the next chapters) and remove saved PCA results from the `reducedDim` slot:

```
assay(umi, "logcounts_raw") <- log2(counts(umi) + 1)
reducedDim(umi) <- NULL
```

Save the data:

```
saveRDS(umi, file = "tung/umi.rds")
```

### 7.1.8 Big Exercise

Perform exactly the same QC analysis with read counts of the same Blischak data. Use `tung/reads.txt` file to load the reads. Once you have finished please compare your results to ours (next chapter).

### 7.1.9 sessionInfo()

```
## R version 3.4.3 (2017-11-30)
## Platform: x86_64-pc-linux-gnu (64-bit)
## Running under: Debian GNU/Linux 9 (stretch)
##
## Matrix products: default
## BLAS: /usr/lib/openblas-base/libblas.so.3
## LAPACK: /usr/lib/libopenblas-p0.2.19.so
##
## locale:
## [1] LC_CTYPE=en_US.UTF-8          LC_NUMERIC=C
## [3] LC_TIME=en_US.UTF-8          LC_COLLATE=en_US.UTF-8
## [5] LC_MONETARY=en_US.UTF-8      LC_MESSAGES=C
## [7] LC_PAPER=en_US.UTF-8         LC_NAME=C
## [9] LC_ADDRESS=C                 LC_TELEPHONE=C
## [11] LC_MEASUREMENT=en_US.UTF-8   LC_IDENTIFICATION=C
##
## attached base packages:
## [1] parallel    stats4     methods    stats      graphics   grDevices utils
## [8] datasets    base
##
## other attached packages:
## [1] limma_3.34.8           scater_1.6.2
## [3] ggplot2_2.2.1          SingleCellExperiment_1.0.0
## [5] SummarizedExperiment_1.8.1 DelayedArray_0.4.1
## [7] matrixStats_0.53.0       Biobase_2.38.0
## [9] GenomicRanges_1.30.1     GenomeInfoDb_1.14.0
## [11] IRanges_2.12.0          S4Vectors_0.16.0
## [13] BiocGenerics_0.24.0     knitr_1.19
##
## loaded via a namespace (and not attached):
## [1] ggbeeswarm_0.6.0        minqa_1.2.4        colorspace_1.3-2
## [4] mvoutlier_2.0.8          rjson_0.2.15       modeltools_0.2-21
## [7] class_7.3-14             mclust_5.4         rprojroot_1.3-2
```

```

## [10] XVector_0.18.0          pls_2.6-0           cvTools_0.3.2
## [13] MatrixModels_0.4-1       flexmix_2.3-14      bit64_0.9-7
## [16] AnnotationDbi_1.40.0    mvtnorm_1.0-7       sROC_0.1-2
## [19] splines_3.4.3          tximport_1.6.0      robustbase_0.92-8
## [22] nloptr_1.0.4          robCompositions_2.0.6 pbkrtest_0.4-7
## [25] kernlab_0.9-25        cluster_2.0.6      shinydashboard_0.6.1
## [28] shiny_1.0.5            rrcov_1.4-3        compiler_3.4.3
## [31] httr_1.3.1             backports_1.1.2    assertthat_0.2.0
## [34] Matrix_1.2-7.1         lazyeval_0.2.1     htmltools_0.3.6
## [37] quantreg_5.35          prettyunits_1.0.2   tools_3.4.3
## [40] bindrcpp_0.2            gtable_0.2.0       glue_1.2.0
## [43] GenomeInfoDbData_1.0.0  reshape2_1.4.3      dplyr_0.7.4
## [46] Rcpp_0.12.15            trimcluster_0.1-2  sgeostat_1.0-27
## [49] nlme_3.1-129           fpc_2.1-11        lmtest_0.9-35
## [52] laeken_0.4.6          xfun_0.1          stringr_1.2.0
## [55] lme4_1.1-15            mime_0.5          XML_3.98-1.9
## [58] edgeR_3.20.8           DEoptimR_1.0-8     zoo_1.8-1
## [61] zlibbioc_1.24.0         MASS_7.3-45       scales_0.5.0
## [64] VIM_4.7.0              SparseM_1.77      rhdf5_2.22.0
## [67] RColorBrewer_1.1-2     yaml_2.1.16       memoise_1.1.0
## [70] gridExtra_2.3           biomaRt_2.34.2    reshape_0.8.7
## [73] stringi_1.1.6          RSQLite_2.0       pcaPP_1.9-73
## [76] e1071_1.6-8            boot_1.3-18       prabclus_2.2-6
## [79] rlang_0.1.6             pkgconfig_2.0.1   bitops_1.0-6
## [82] evaluate_0.10.1         lattice_0.20-34  bindr_0.1
## [85] labeling_0.3            cowplot_0.9.2     bit_1.1-12
## [88] GGally_1.3.2           plyr_1.8.4        magrittr_1.5
## [91] bookdown_0.6            R6_2.2.2          DBI_0.7
## [94] pillar_1.1.0           mgcv_1.8-23      RCurl_1.95-4.10
## [97] sp_1.2-7               nnet_7.3-12       tibble_1.4.2
## [100] car_2.1-6             rmarkdown_1.8      viridis_0.5.0
## [103] progress_1.1.2         locfit_1.5-9.1    grid_3.4.3
## [106] data.table_1.10.4-3    blob_1.1.0        diptest_0.75-7
## [109] vcd_1.4-4              digest_0.6.15     xtable_1.8-2
## [112] httpuv_1.3.5          munsell_0.4.3     beeswarm_0.2.3
## [115] viridisLite_0.3.0      vipor_0.4.5

```

## 7.2 Expression QC (Reads)

```

library(SingleCellExperiment)
library(scater)
options(stringsAsFactors = FALSE)

reads <- read.table("tung/reads.txt", sep = "\t")
anno <- read.table("tung/annotation.txt", sep = "\t", header = TRUE)

head(reads[ , 1:3])

##                               NA19098.r1.A01 NA19098.r1.A02 NA19098.r1.A03
## ENSG00000237683                  0          0          0
## ENSG00000187634                  0          0          0
## ENSG00000188976                  57         140         1
## ENSG00000187961                  0          0          0

```

```

## ENSG00000187583          0          0          0
## ENSG00000187642          0          0          0
head(anno)

##   individual replicate well      batch sample_id
## 1     NA19098         r1    A01 NA19098.r1 NA19098.r1.A01
## 2     NA19098         r1    A02 NA19098.r1 NA19098.r1.A02
## 3     NA19098         r1    A03 NA19098.r1 NA19098.r1.A03
## 4     NA19098         r1    A04 NA19098.r1 NA19098.r1.A04
## 5     NA19098         r1    A05 NA19098.r1 NA19098.r1.A05
## 6     NA19098         r1    A06 NA19098.r1 NA19098.r1.A06

reads <- SingleCellExperiment(
  assays = list(counts = as.matrix(reads)),
  colData = anno
)

keep_feature <- rowSums(counts(reads) > 0) > 0
reads <- reads[keep_feature, ]

isSpike(reads, "ERCC") <- grepl("ERCC-", rownames(reads))
isSpike(reads, "MT") <- rownames(reads) %in%
  c("ENSG00000198899", "ENSG00000198727", "ENSG00000198888",
  "ENSG00000198886", "ENSG00000212907", "ENSG00000198786",
  "ENSG00000198695", "ENSG00000198712", "ENSG00000198804",
  "ENSG00000198763", "ENSG00000228253", "ENSG00000198938",
  "ENSG00000198840")

reads <- calculateQCMetrics(
  reads,
  feature_controls = list(
    ERCC = isSpike(reads, "ERCC"),
    MT = isSpike(reads, "MT")
  )
)

hist(
  reads$total_counts,
  breaks = 100
)
abline(v = 1.3e6, col = "red")

filter_by_total_counts <- (reads$total_counts > 1.3e6)

table(filter_by_total_counts)

## filter_by_total_counts
## FALSE  TRUE
##    180   684

hist(
  reads$total_features,
  breaks = 100
)
abline(v = 7000, col = "red")

```

### Histogram of reads\$total\_counts

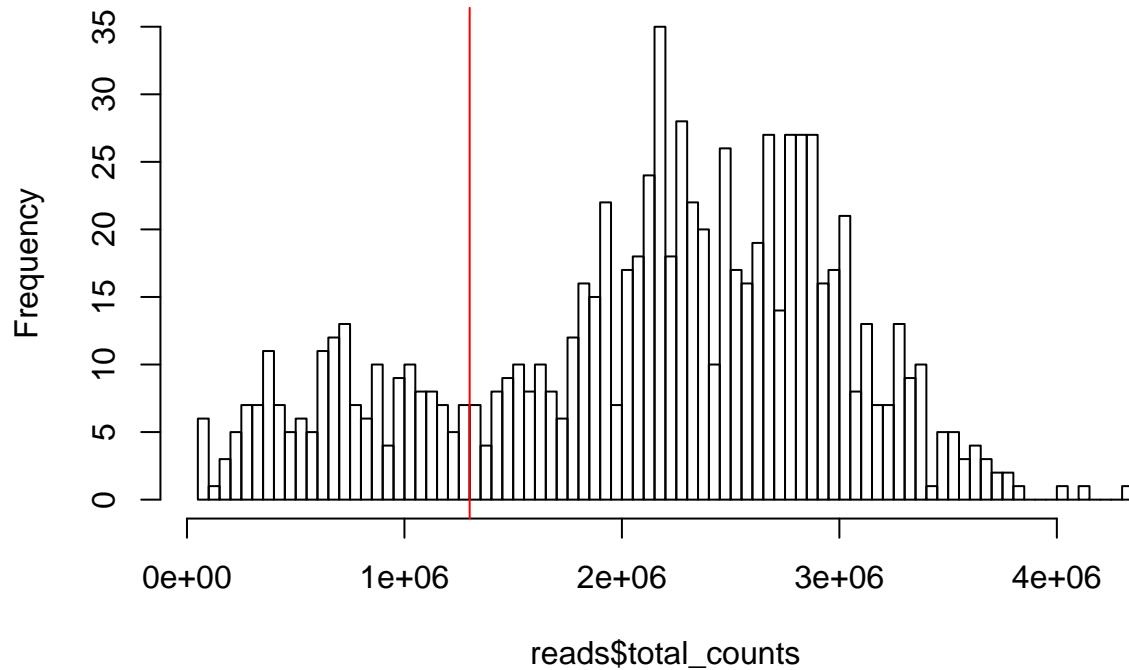


Figure 7.8: Histogram of library sizes for all cells

### Histogram of reads\$total\_features

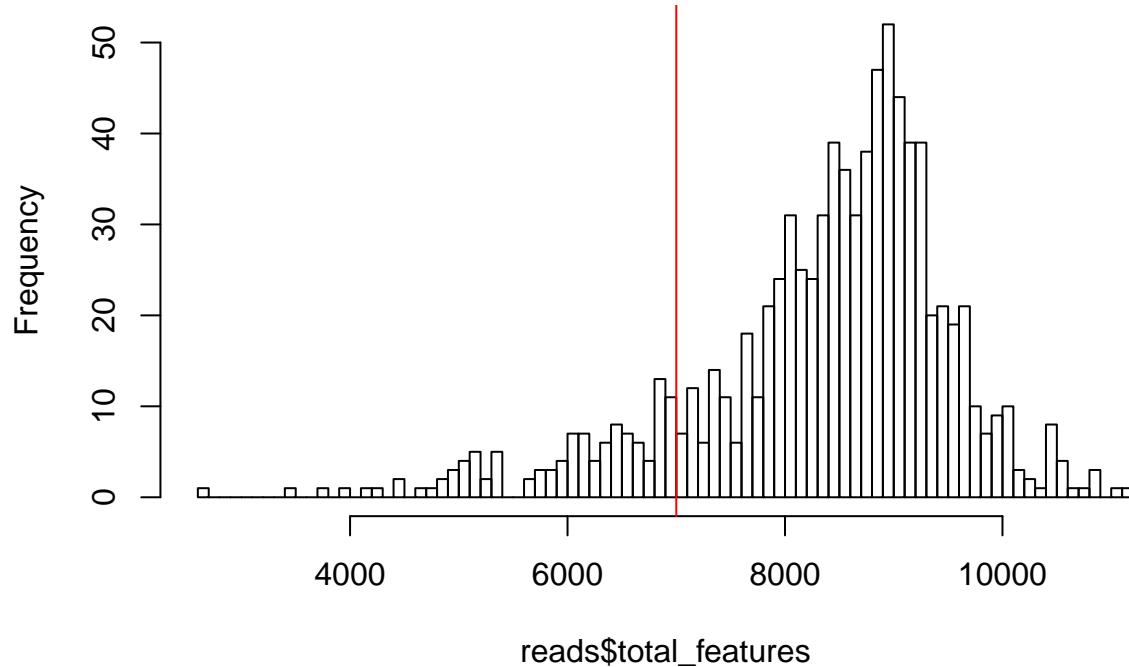


Figure 7.9: Histogram of the number of detected genes in all cells

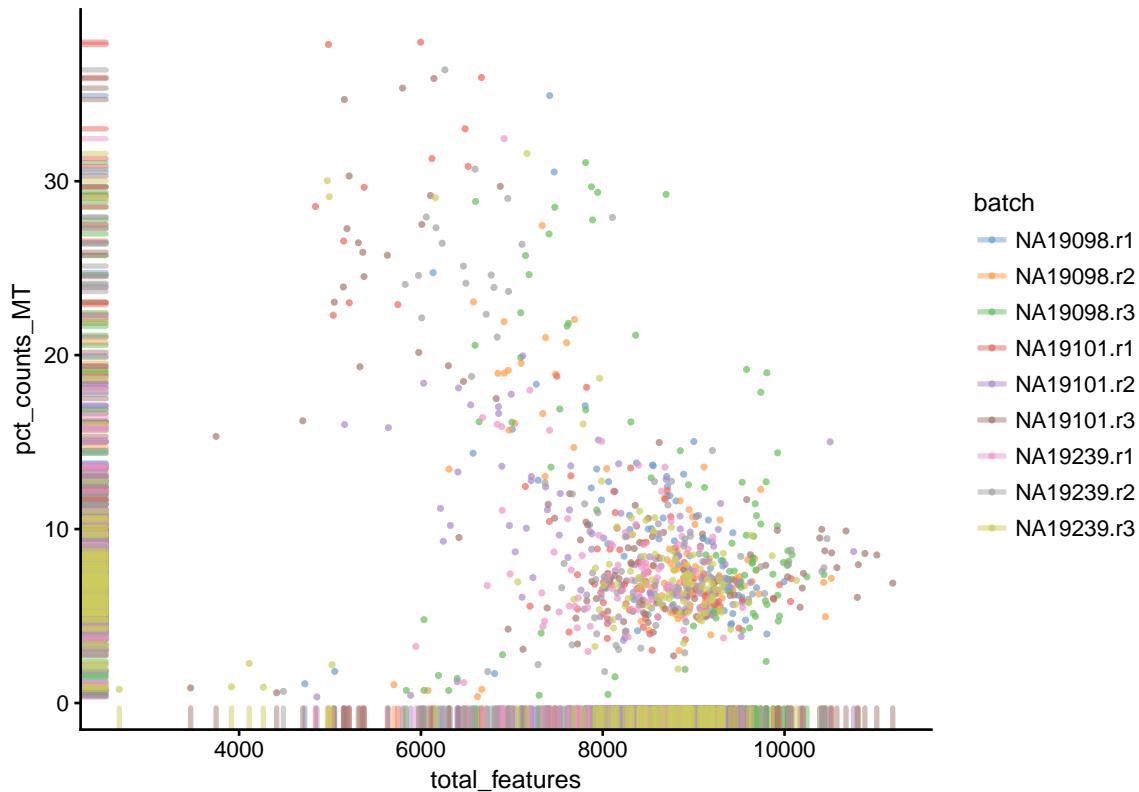


Figure 7.10: Percentage of counts in MT genes

```

filter_by_expr_features <- (reads$total_features > 7000)

table(filter_by_expr_features)

## filter_by_expr_features
## FALSE  TRUE
##    116   748

plotPhenoData(
  reads,
  aes_string(
    x = "total_features",
    y = "pct_counts_MT",
    colour = "batch"
  )
)

plotPhenoData(
  reads,
  aes_string(
    x = "total_features",
    y = "pct_counts_ERCC",
    colour = "batch"
  )
)

```

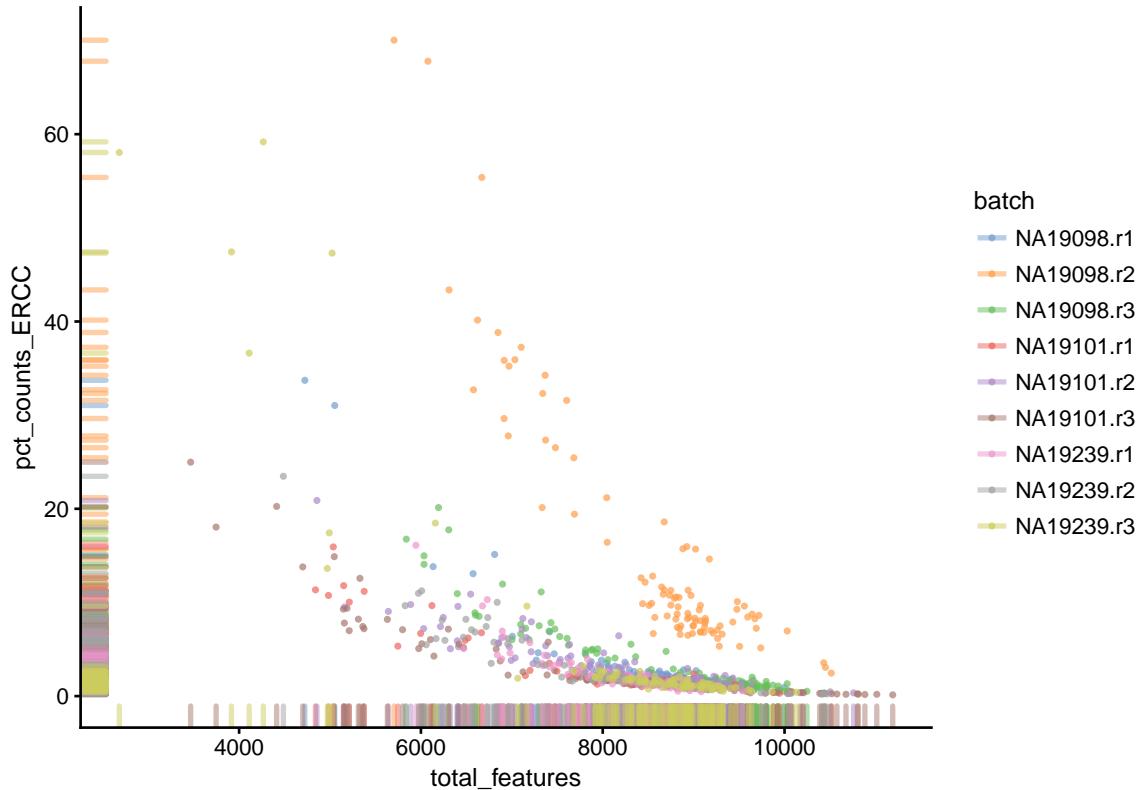


Figure 7.11: Percentage of counts in ERCCs

```

filter_by_ERCC <-
  reads$batch != "NA19098.r2" & reads$pct_counts_ERCC < 25
table(filter_by_ERCC)

## filter_by_ERCC
## FALSE  TRUE
##    103    761

filter_by_MT <- reads$pct_counts_MT < 30
table(filter_by_MT)

## filter_by_MT
## FALSE  TRUE
##    18    846

reads$use <- (
  # sufficient features (genes)
  filter_by_expr_features &
  # sufficient molecules counted
  filter_by_total_counts &
  # sufficient endogenous RNA
  filter_by_ERCC &
  # remove cells with unusual number of reads in MT genes
  filter_by_MT
)

```

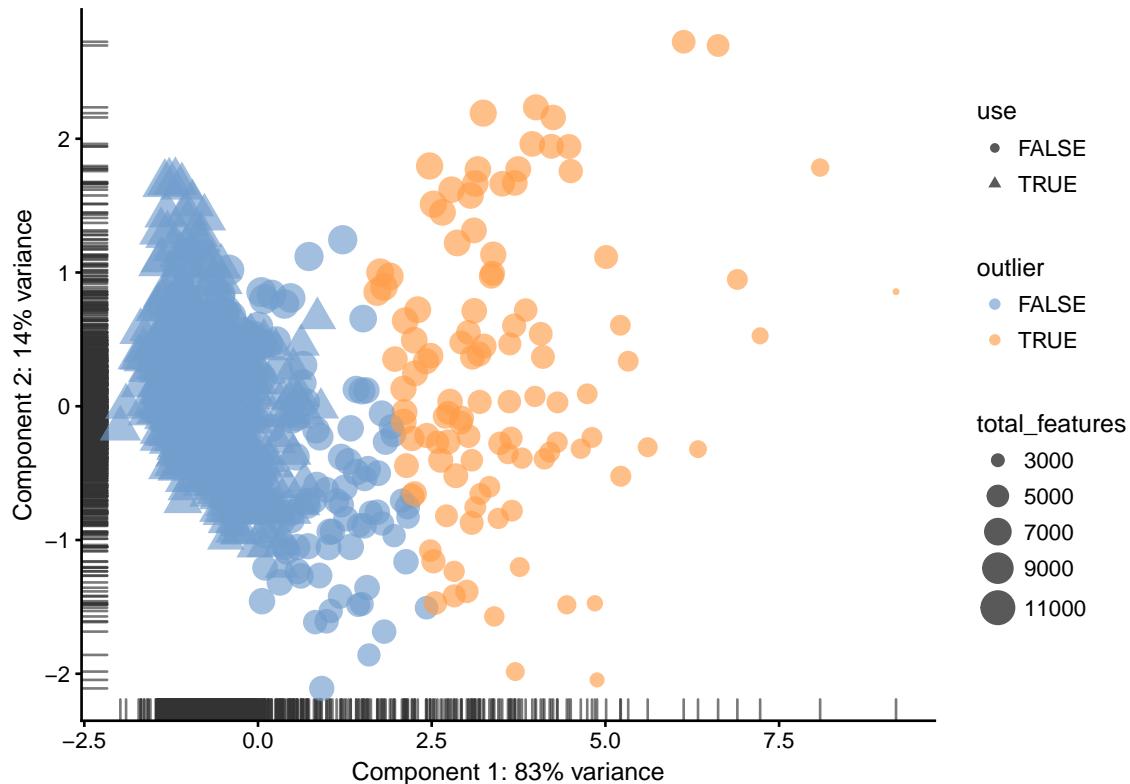


Figure 7.12: PCA plot used for automatic detection of cell outliers

```



```

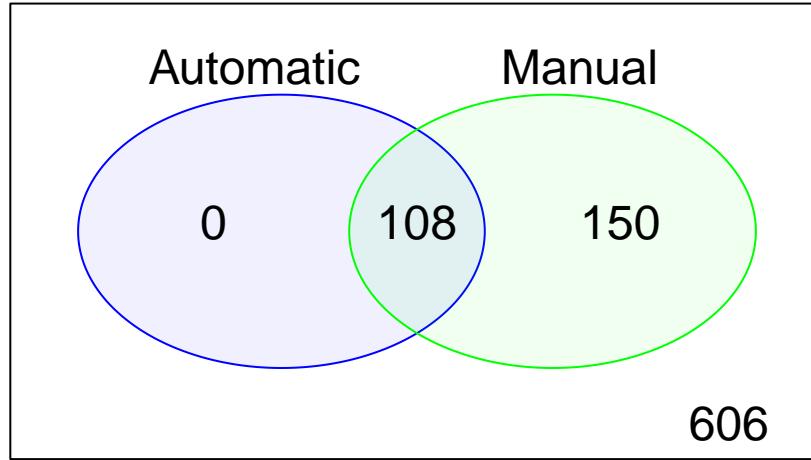


Figure 7.13: Comparison of the default, automatic and manual cell filters

```
##      plotMDS

## The following object is masked from 'package:BiocGenerics':
## 
##      plotMA

auto <- colnames(reads)[reads$outlier]
man <- colnames(reads)[!reads$use]
venn.diag <- vennCounts(
  cbind(colnames(reads) %in% auto,
        colnames(reads) %in% man)
)
vennDiagram(
  venn.diag,
  names = c("Automatic", "Manual"),
  circle.col = c("blue", "green")
)

plotQC(reads, type = "highest-expression")

filter_genes <- apply(
  counts(reads[, colData(reads)$use]),
  1,
  function(x) length(x[x > 1]) >= 2
)
rowData(reads)$use <- filter_genes

table(filter_genes)

## filter_genes
## FALSE  TRUE
## 2664 16062
dim(reads[rowData(reads)$use, colData(reads)$use])

## [1] 16062   606
assay(reads, "logcounts_raw") <- log2(counts(reads) + 1)
reducedDim(reads) <- NULL
```

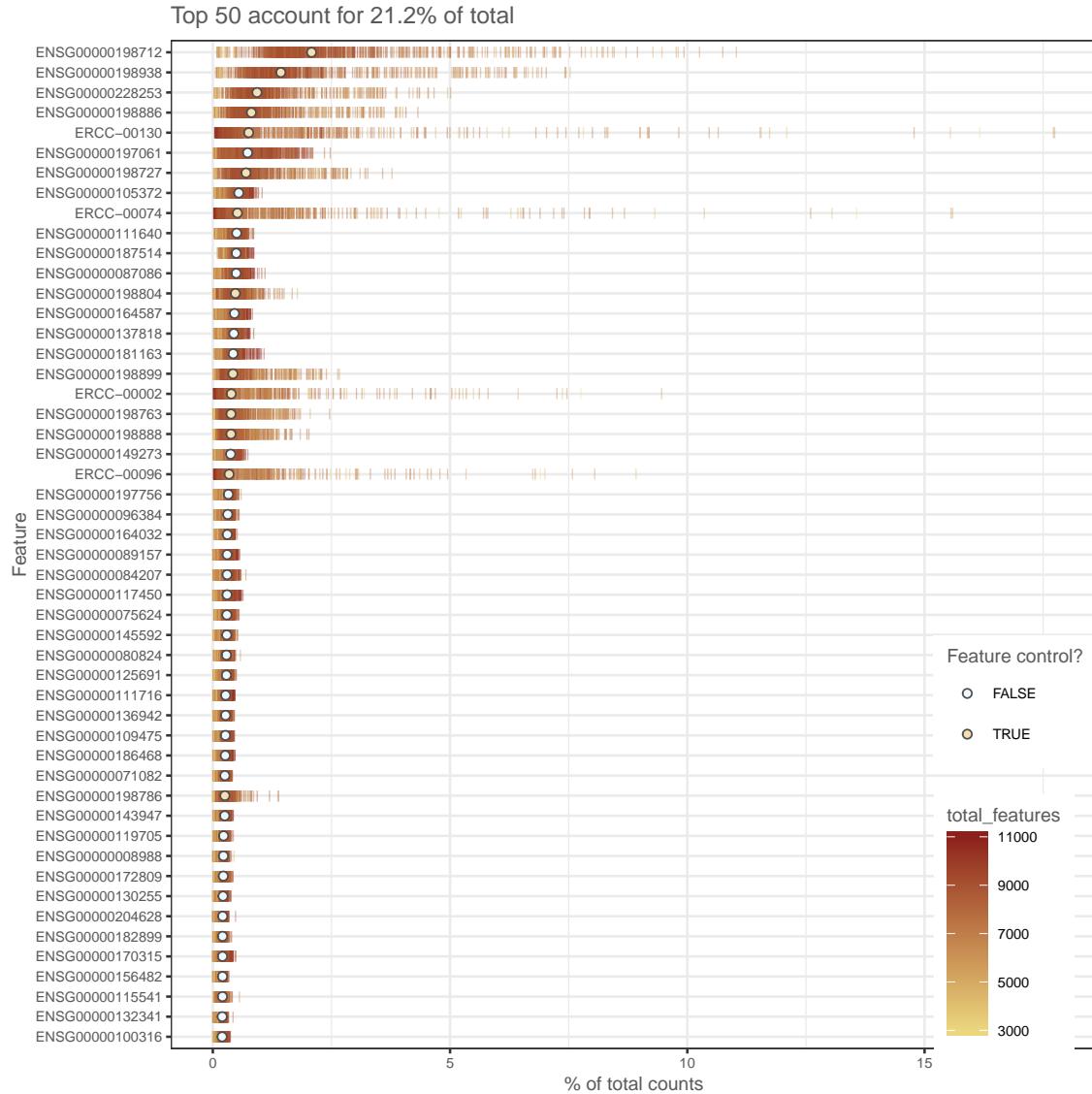


Figure 7.14: Number of total counts consumed by the top 50 expressed genes

```
saveRDS(reads, file = "tung/readss.rds")
```

By comparing Figure 7.6 and Figure 7.13, it is clear that the reads based filtering removed more cells than the UMI based analysis. If you go back and compare the results you should be able to conclude that the ERCC and MT filters are more strict for the reads-based analysis.

```
sessionInfo()
```

```
## R version 3.4.3 (2017-11-30)
## Platform: x86_64-pc-linux-gnu (64-bit)
## Running under: Debian GNU/Linux 9 (stretch)
##
## Matrix products: default
## BLAS: /usr/lib/openblas-base/libblas.so.3
## LAPACK: /usr/lib/libopenblas-p-r0.2.19.so
##
## locale:
## [1] LC_CTYPE=en_US.UTF-8      LC_NUMERIC=C
## [3] LC_TIME=en_US.UTF-8       LC_COLLATE=en_US.UTF-8
## [5] LC_MONETARY=en_US.UTF-8   LC_MESSAGES=C
## [7] LC_PAPER=en_US.UTF-8      LC_NAME=C
## [9] LC_ADDRESS=C              LC_TELEPHONE=C
## [11] LC_MEASUREMENT=en_US.UTF-8 LC_IDENTIFICATION=C
##
## attached base packages:
## [1] parallel    stats4     methods    stats      graphics   grDevices utils
## [8] datasets    base
##
## other attached packages:
## [1] limma_3.34.8           scater_1.6.2
## [3] ggplot2_2.2.1          SingleCellExperiment_1.0.0
## [5] SummarizedExperiment_1.8.1 DelayedArray_0.4.1
## [7] matrixStats_0.53.0      Biobase_2.38.0
## [9] GenomicRanges_1.30.1    GenomeInfoDb_1.14.0
## [11] IRanges_2.12.0         S4Vectors_0.16.0
## [13] BiocGenerics_0.24.0    knitr_1.19
##
## loaded via a namespace (and not attached):
## [1] ggbeeswarm_0.6.0        minqa_1.2.4      colorspace_1.3-2
## [4] mvoutlier_2.0.8         rjson_0.2.15    modeltools_0.2-21
## [7] class_7.3-14            mclust_5.4      rprojroot_1.3-2
## [10] XVector_0.18.0          pls_2.6-0       cvTools_0.3.2
## [13] MatrixModels_0.4-1     flexmix_2.3-14   bit64_0.9-7
## [16] AnnotationDbi_1.40.0   mvtnorm_1.0-7    sROC_0.1-2
## [19] splines_3.4.3          tximport_1.6.0   robustbase_0.92-8
## [22] nloptr_1.0.4           robCompositions_2.0.6 pbkrtest_0.4-7
## [25] kernlab_0.9-25         cluster_2.0.6   shinydashboard_0.6.1
## [28] shiny_1.0.5             rrcov_1.4-3     compiler_3.4.3
## [31] httr_1.3.1              backports_1.1.2  assertthat_0.2.0
## [34] Matrix_1.2-7.1          lazyeval_0.2.1   htmltools_0.3.6
## [37] quantreg_5.35            prettyunits_1.0.2 tools_3.4.3
## [40] bindrcpp_0.2              gtable_0.2.0     glue_1.2.0
## [43] GenomeInfoDbData_1.0.0  reshape2_1.4.3   dplyr_0.7.4
## [46] Rcpp_0.12.15             trimcluster_0.1-2 sgeostat_1.0-27
```

```

## [49] nlme_3.1-129          fpc_2.1-11           lmtest_0.9-35
## [52] laeken_0.4.6          xfun_0.1             stringr_1.2.0
## [55] lme4_1.1-15           mime_0.5            XML_3.98-1.9
## [58] edgeR_3.20.8          DEoptimR_1.0-8       zoo_1.8-1
## [61] zlibbioc_1.24.0        MASS_7.3-45          scales_0.5.0
## [64] VIM_4.7.0              SparseM_1.77        rhdf5_2.22.0
## [67] RColorBrewer_1.1-2     yaml_2.1.16          memoise_1.1.0
## [70] gridExtra_2.3          biomaRt_2.34.2      reshape_0.8.7
## [73] stringi_1.1.6          RSQLite_2.0          pcaPP_1.9-73
## [76] e1071_1.6-8            boot_1.3-18          prabclus_2.2-6
## [79] rlang_0.1.6             pkgconfig_2.0.1     bitops_1.0-6
## [82] evaluate_0.10.1         lattice_0.20-34    bindr_0.1
## [85] labeling_0.3            cowplot_0.9.2       bit_1.1-12
## [88] GGally_1.3.2           plyr_1.8.4           magrittr_1.5
## [91] bookdown_0.6            R6_2.2.2             DBI_0.7
## [94] pillar_1.1.0           mgcv_1.8-23          RCurl_1.95-4.10
## [97] sp_1.2-7                nnet_7.3-12          tibble_1.4.2
## [100] car_2.1-6              rmarkdown_1.8          viridis_0.5.0
## [103] progress_1.1.2         loo_1.5-9.1          grid_3.4.3
## [106] data.table_1.10.4-3     blob_1.1.0           diptest_0.75-7
## [109] vcd_1.4-4               digest_0.6.15        xtable_1.8-2
## [112] httpuv_1.3.5           munsell_0.4.3        beeswarm_0.2.3
## [115] viridisLite_0.3.0       vipor_0.4.5

```

## 7.3 Data visualization

### 7.3.1 Introduction

In this chapter we will continue to work with the filtered Tung dataset produced in the previous chapter. We will explore different ways of visualizing the data to allow you to asses what happened to the expression matrix after the quality control step. `scater` package provides several very useful functions to simplify visualisation.

One important aspect of single-cell RNA-seq is to control for batch effects. Batch effects are technical artefacts that are added to the samples during handling. For example, if two sets of samples were prepared in different labs or even on different days in the same lab, then we may observe greater similarities between the samples that were handled together. In the worst case scenario, batch effects may be mistaken for true biological variation. The Tung data allows us to explore these issues in a controlled manner since some of the salient aspects of how the samples were handled have been recorded. Ideally, we expect to see batches from the same individual grouping together and distinct groups corresponding to each individual.

```

library(SingleCellExperiment)
library(scater)
options(stringsAsFactors = FALSE)
umi <- readRDS("tung/umi.rds")
umi.qc <- umi$rowData(umi)$use, colData(umi)$use]
endog_genes <- !rowData(umi.qc)$is_feature_control

```

### 7.3.2 PCA plot

The easiest way to overview the data is by transforming it using the principal component analysis and then visualize the first two principal components.

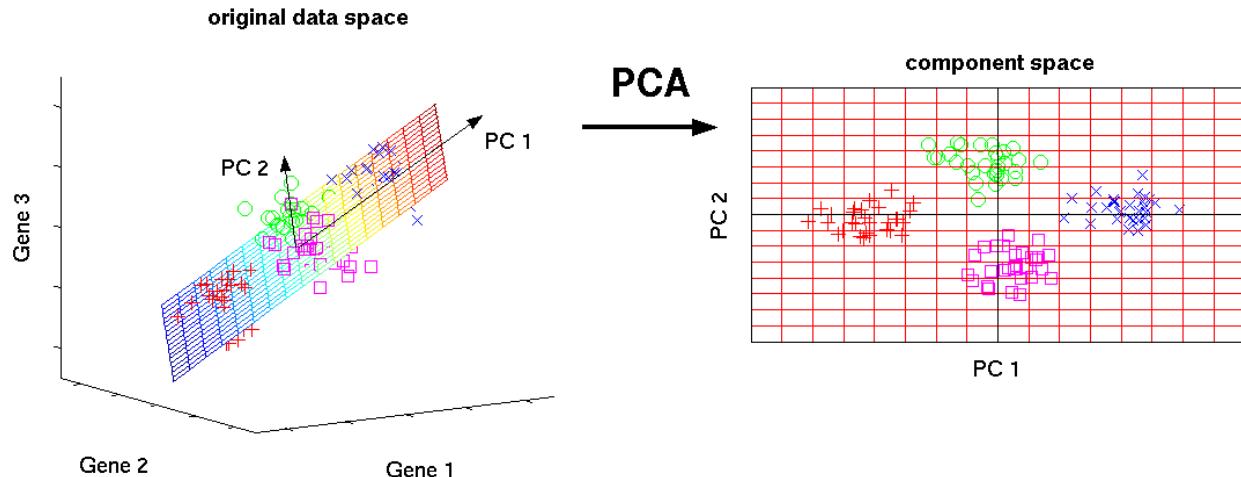


Figure 7.15: Schematic representation of PCA dimensionality reduction

Principal component analysis (PCA) is a statistical procedure that uses a transformation to convert a set of observations into a set of values of linearly uncorrelated variables called principal components (PCs). The number of principal components is less than or equal to the number of original variables.

Mathematically, the PCs correspond to the eigenvectors of the covariance matrix. The eigenvectors are sorted by eigenvalue so that the first principal component accounts for as much of the variability in the data as possible, and each succeeding component in turn has the highest variance possible under the constraint that it is orthogonal to the preceding components (the figure below is taken from here).

### 7.3.2.1 Before QC

Without log-transformation:

```
plotPCA(
  umi[endo_genes, ],
  exprs_values = "counts",
  colour_by = "batch",
  size_by = "total_features",
  shape_by = "individual"
)
```

With log-transformation:

```
plotPCA(
  umi[endo_genes, ],
  exprs_values = "logcounts_raw",
  colour_by = "batch",
  size_by = "total_features",
  shape_by = "individual"
)
```

Clearly log-transformation is beneficial for our data - it reduces the variance on the first principal component and already separates some biological effects. Moreover, it makes the distribution of the expression values more normal. In the following analysis and chapters we will be using log-transformed raw counts by default.

**However, note that just a log-transformation is not enough to account for different technical factors between the cells (e.g. sequencing depth). Therefore, please do not use logcounts\_raw**

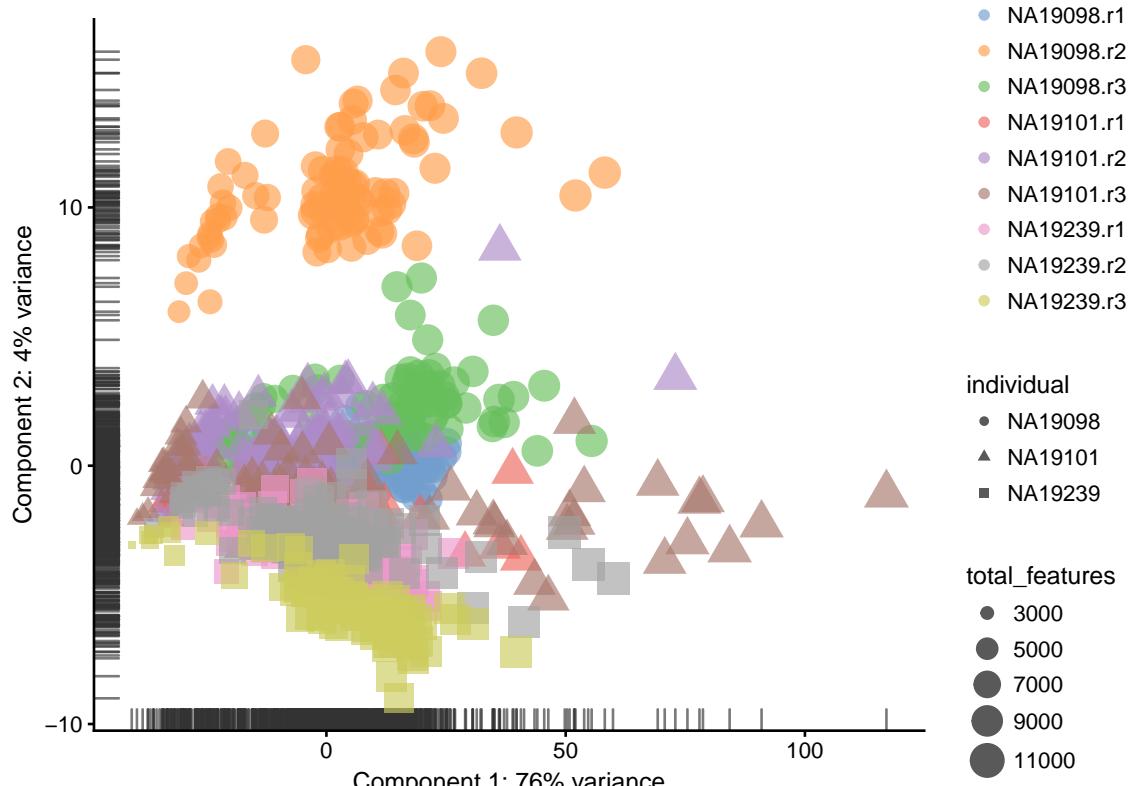


Figure 7.16: PCA plot of the tung data

for your downstream analysis, instead as a minimum suitable data use the `logcounts` slot of the `SingleCellExperiment` object, which not just log-transformed, but also normalised by library size (e.g. CPM normalisation). In the course we use `logcounts_raw` only for demonstration purposes!

### 7.3.2.2 After QC

```
plotPCA(
  umi.qc[enod_genes, ],
  exprs_values = "logcounts_raw",
  colour_by = "batch",
  size_by = "total_features",
  shape_by = "individual"
)
```

Comparing Figure 7.17 and Figure 7.18, it is clear that after quality control the NA19098.r2 cells no longer form a group of outliers.

By default only the top 500 most variable genes are used by scater to calculate the PCA. This can be adjusted by changing the `ntop` argument.

**Exercise 1** How do the PCA plots change if when all 14,214 genes are used? Or when only top 50 genes are used? Why does the fraction of variance accounted for by the first PC change so dramatically?

**Hint** Use `ntop` argument of the `plotPCA` function.

**Our answer**

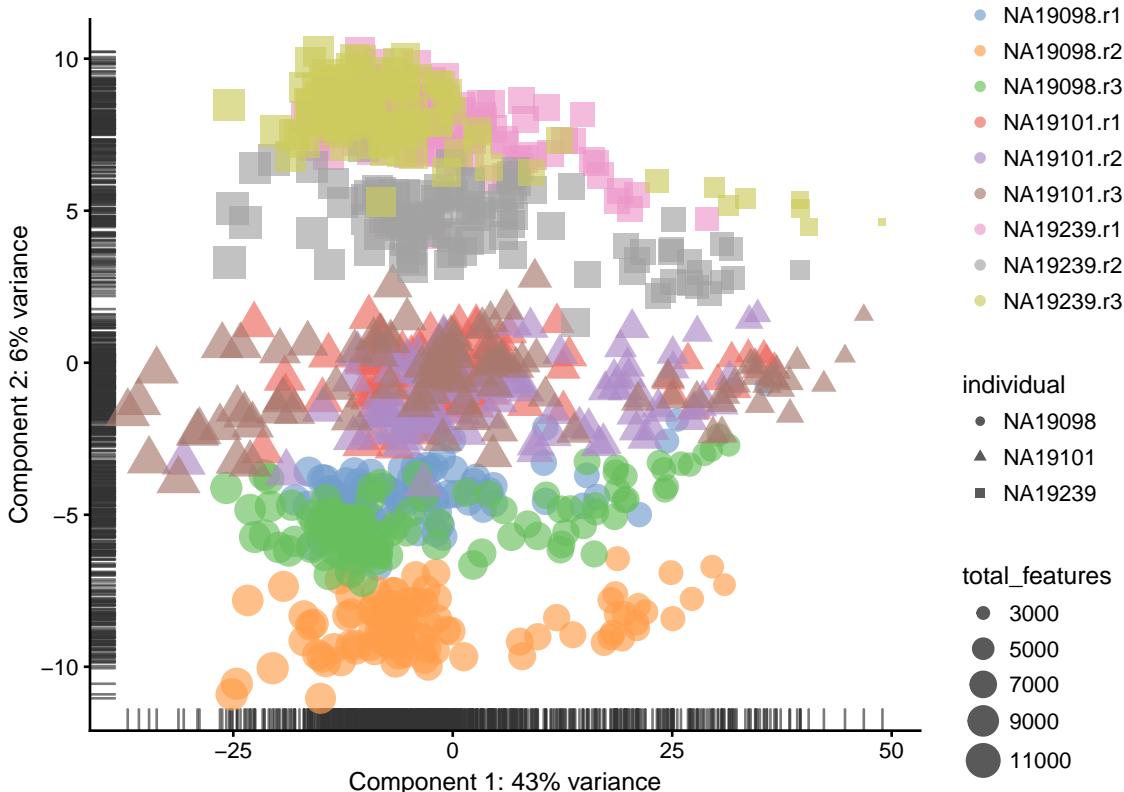


Figure 7.17: PCA plot of the tung data

If your answers are different please compare your code with ours (you need to search for this exercise in the opened file).

### 7.3.3 tSNE map

An alternative to PCA for visualizing scRNASeq data is a tSNE plot. tSNE (t-Distributed Stochastic Neighbor Embedding) combines dimensionality reduction (e.g. PCA) with random walks on the nearest-neighbour network to map high dimensional data (i.e. our 14,214 dimensional expression matrix) to a 2-dimensional space while preserving local distances between cells. In contrast with PCA, tSNE is a stochastic algorithm which means running the method multiple times on the same dataset will result in different plots. Due to the non-linear and stochastic nature of the algorithm, tSNE is more difficult to intuitively interpret tSNE. To ensure reproducibility, we fix the “seed” of the random-number generator in the code below so that we always get the same plot.

#### 7.3.3.1 Before QC

```
plotTSNE(
  umi[endo_genes, ],
  exprs_values = "logcounts_raw",
  perplexity = 130,
  colour_by = "batch",
  size_by = "total_features",
  shape_by = "individual",
```

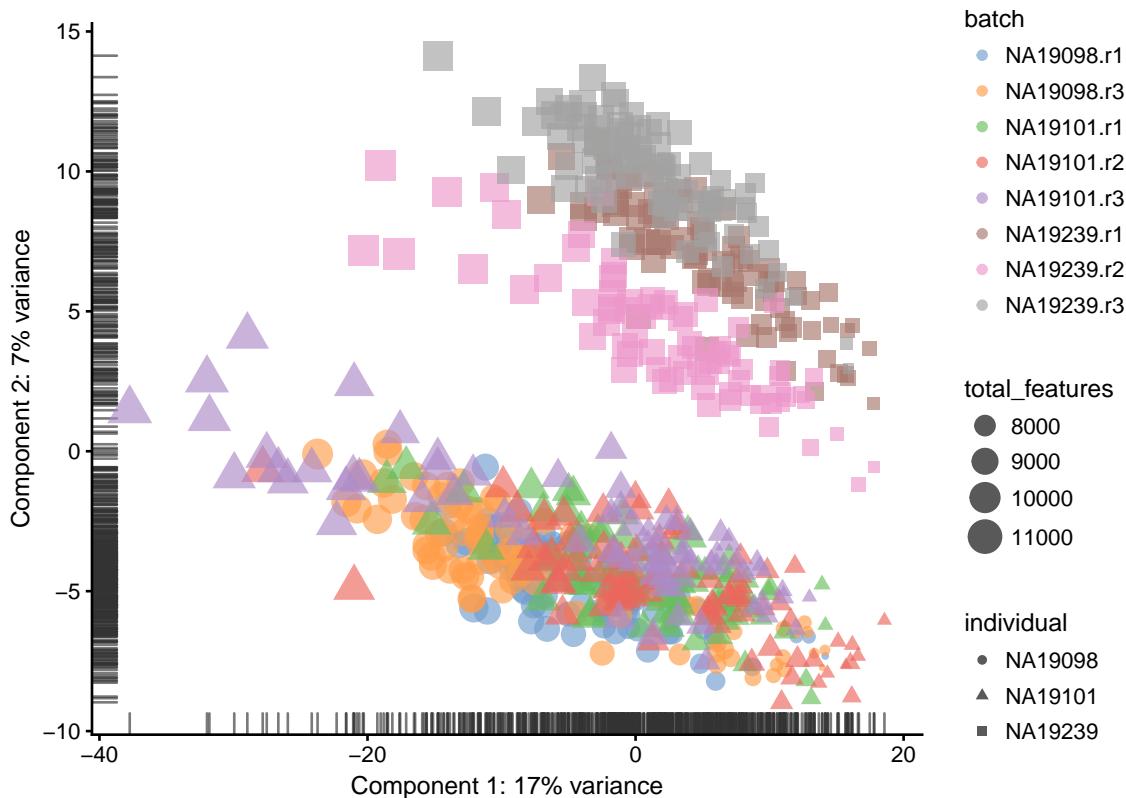


Figure 7.18: PCA plot of the tung data

```
    rand_seed = 123456
)
```

### 7.3.3.2 After QC

```
plotTSNE(
  umi.qc[!endog_genes, ],
  exprs_values = "logcounts_raw",
  perplexity = 130,
  colour_by = "batch",
  size_by = "total_features",
  shape_by = "individual",
  rand_seed = 123456
)
```

Interpreting PCA and tSNE plots is often challenging and due to their stochastic and non-linear nature, they are less intuitive. However, in this case it is clear that they provide a similar picture of the data. Comparing Figure 7.21 and 7.22, it is again clear that the samples from NA19098.r2 are no longer outliers after the QC filtering.

Furthermore tSNE requires you to provide a value of `perplexity` which reflects the number of neighbours used to build the nearest-neighbour network; a high value creates a dense network which clumps cells together while a low value makes the network more sparse allowing groups of cells to separate from each other. `scater` uses a default perplexity of the total number of cells divided by five (rounded down).

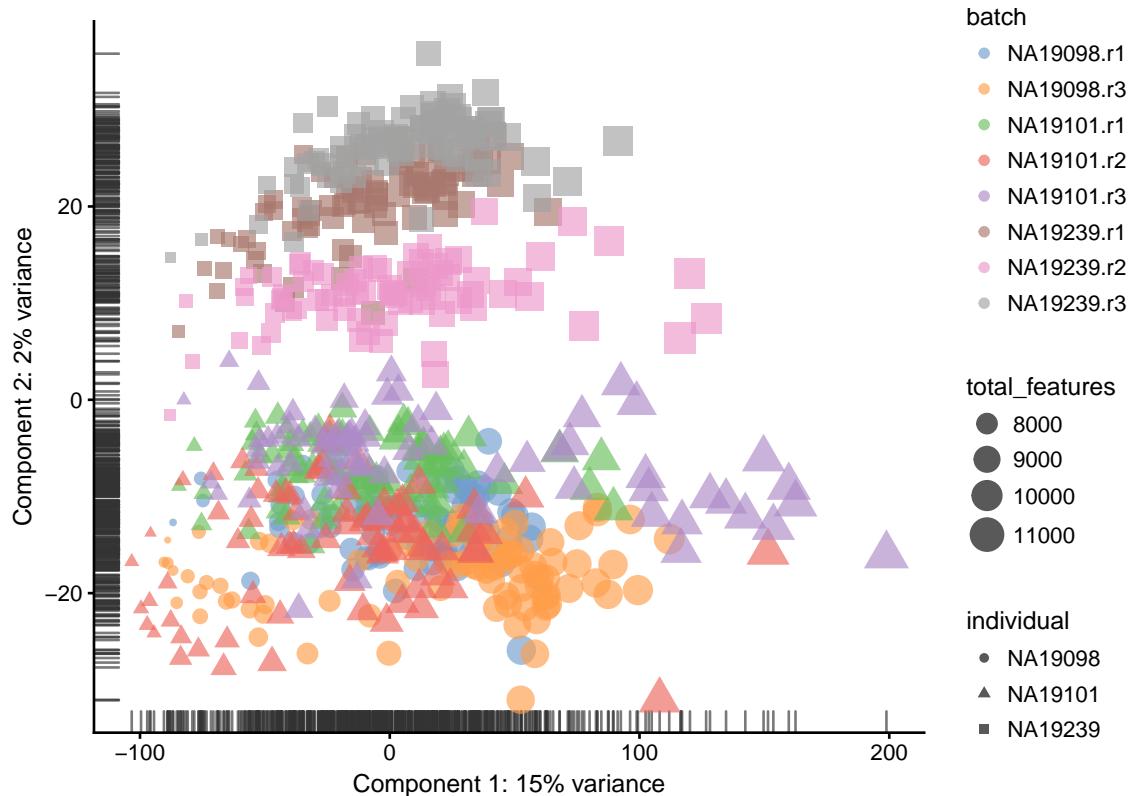


Figure 7.19: PCA plot of the tung data (14214 genes)

You can read more about the pitfalls of using tSNE here.

**Exercise 2** How do the tSNE plots change when a perplexity of 10 or 200 is used? How does the choice of perplexity affect the interpretation of the results?

**Our answer**

### 7.3.4 Big Exercise

Perform the same analysis with read counts of the Blischak data. Use `tung/reads.rds` file to load the reads SCE object. Once you have finished please compare your results to ours (next chapter).

### 7.3.5 sessionInfo()

```
## R version 3.4.3 (2017-11-30)
## Platform: x86_64-pc-linux-gnu (64-bit)
## Running under: Debian GNU/Linux 9 (stretch)
##
## Matrix products: default
## BLAS: /usr/lib/openblas-base/libblas.so.3
## LAPACK: /usr/lib/libopenblas-r0.2.19.so
##
## locale:
## [1] LC_CTYPE=en_US.UTF-8          LC_NUMERIC=C
## [3] LC_TIME=en_US.UTF-8          LC_COLLATE=en_US.UTF-8
```

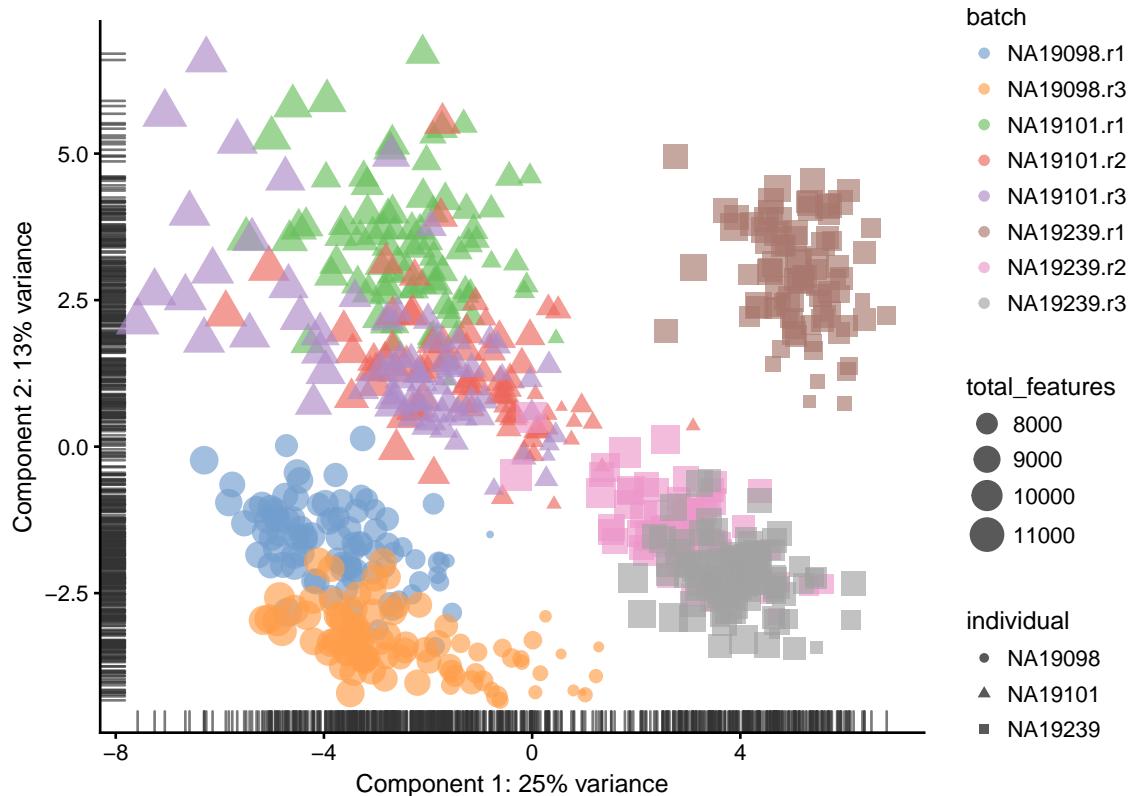


Figure 7.20: PCA plot of the tung data (50 genes)

```

## [5] LC_MONETARY=en_US.UTF-8      LC_MESSAGES=C
## [7] LC_PAPER=en_US.UTF-8        LC_NAME=C
## [9] LC_ADDRESS=C                LC_TELEPHONE=C
## [11] LC_MEASUREMENT=en_US.UTF-8 LC_IDENTIFICATION=C
##
## attached base packages:
## [1] parallel   stats4    methods   stats     graphics  grDevices utils
## [8] datasets   base
##
## other attached packages:
## [1] scater_1.6.2           ggplot2_2.2.1
## [3] SingleCellExperiment_1.0.0 SummarizedExperiment_1.8.1
## [5] DelayedArray_0.4.1       matrixStats_0.53.0
## [7] Biobase_2.38.0          GenomicRanges_1.30.1
## [9] GenomeInfoDb_1.14.0      IRanges_2.12.0
## [11] S4Vectors_0.16.0         BiocGenerics_0.24.0
## [13] knitr_1.19
##
## loaded via a namespace (and not attached):
## [1] viridis_0.5.0            httr_1.3.1          edgeR_3.20.8
## [4] bit64_0.9-7              viridisLite_0.3.0   shiny_1.0.5
## [7] assertthat_0.2.0          blob_1.1.0          viper_0.4.5
## [10] GenomeInfoDbData_1.0.0   yaml_2.1.16         progress_1.1.2
## [13] pillar_1.1.0             RSQLite_2.0          backports_1.1.2
## [16] lattice_0.20-34          glue_1.2.0          limma_3.34.8

```

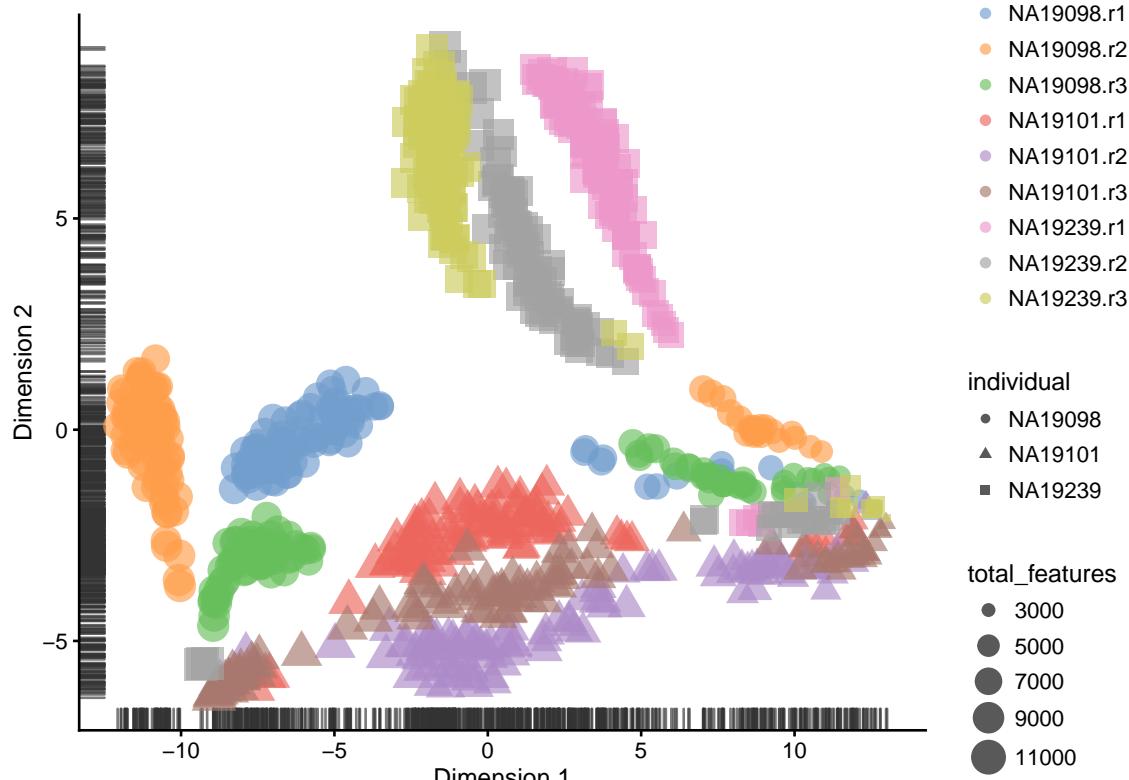


Figure 7.21: tSNE map of the tung data

```

## [19] digest_0.6.15          XVector_0.18.0           colorspace_1.3-2
## [22] cowplot_0.9.2         htmltools_0.3.6          httpuv_1.3.5
## [25] Matrix_1.2-7.1         plyr_1.8.4              XML_3.98-1.9
## [28] pkgconfig_2.0.1         biomaRt_2.34.2          bookdown_0.6
## [31] zlibbioc_1.24.0        xtable_1.8-2            scales_0.5.0
## [34] Rtsne_0.13              tibble_1.4.2             lazyeval_0.2.1
## [37] magrittr_1.5             mime_0.5                memoise_1.1.0
## [40] evaluate_0.10.1         beeswarm_0.2.3          shinydashboard_0.6.1
## [43] tools_3.4.3             data.table_1.10.4-3    prettyunits_1.0.2
## [46] stringr_1.2.0            munsell_0.4.3           locfit_1.5-9.1
## [49] AnnotationDbi_1.40.0    bindrcpp_0.2            compiler_3.4.3
## [52] rlang_0.1.6              rhdf5_2.22.0            grid_3.4.3
## [55] RCurl_1.95-4.10         tximport_1.6.0          rjson_0.2.15
## [58] labeling_0.3              bitops_1.0-6            rmarkdown_1.8
## [61] gtable_0.2.0             DBI_0.7                 reshape2_1.4.3
## [64] R6_2.2.2                 gridExtra_2.3           dplyr_0.7.4
## [67] bit_1.1-12               bindr_0.1                rprojroot_1.3-2
## [70] ggbeeswarm_0.6.0        stringi_1.1.6           Rcpp_0.12.15
## [73] xfun_0.1

```

## 7.4 Data visualization (Reads)

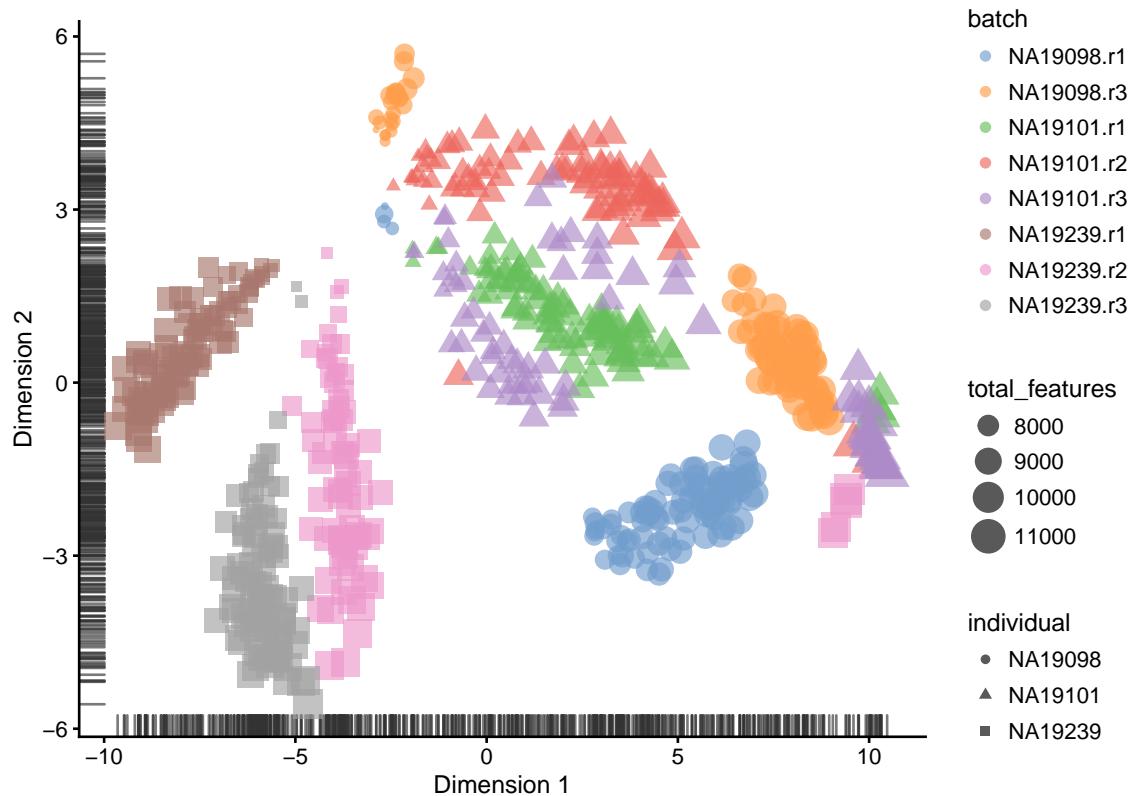


Figure 7.22: tSNE map of the tung data

```

library(scater)
options(stringsAsFactors = FALSE)
reads <- readRDS("tung/reads.rds")
reads.qc <- reads$rowData(reads)$use, colData(reads)$use]
endog_genes <- !rowData(reads.qc)$is_feature_control

plotPCA(
  reads[endog_genes, ],
  exprs_values = "counts",
  colour_by = "batch",
  size_by = "total_features",
  shape_by = "individual"
)

plotPCA(
  reads[endog_genes, ],
  exprs_values = "logcounts_raw",
  colour_by = "batch",
  size_by = "total_features",
  shape_by = "individual"
)

plotPCA(
  reads.qc[endog_genes, ],
  exprs_values = "logcounts_raw",
  colour_by = "batch",

```

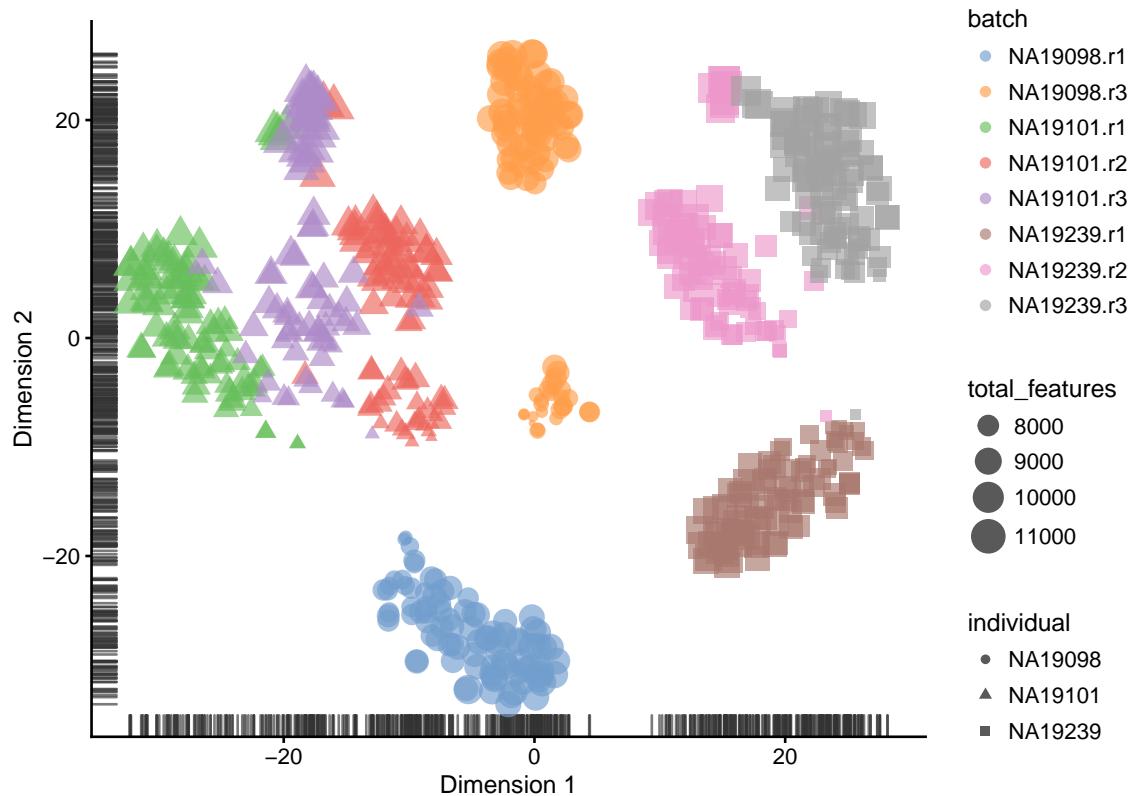


Figure 7.23: tSNE map of the tung data (perplexity = 10)

```

size_by = "total_features",
shape_by = "individual"
)

plotTSNE(
  reads[endog_genes, ],
  exprs_values = "logcounts_raw",
  perplexity = 130,
  colour_by = "batch",
  size_by = "total_features",
  shape_by = "individual",
  rand_seed = 123456
)

plotTSNE(
  reads.qc[endog_genes, ],
  exprs_values = "logcounts_raw",
  perplexity = 130,
  colour_by = "batch",
  size_by = "total_features",
  shape_by = "individual",
  rand_seed = 123456
)

sessionInfo()

```

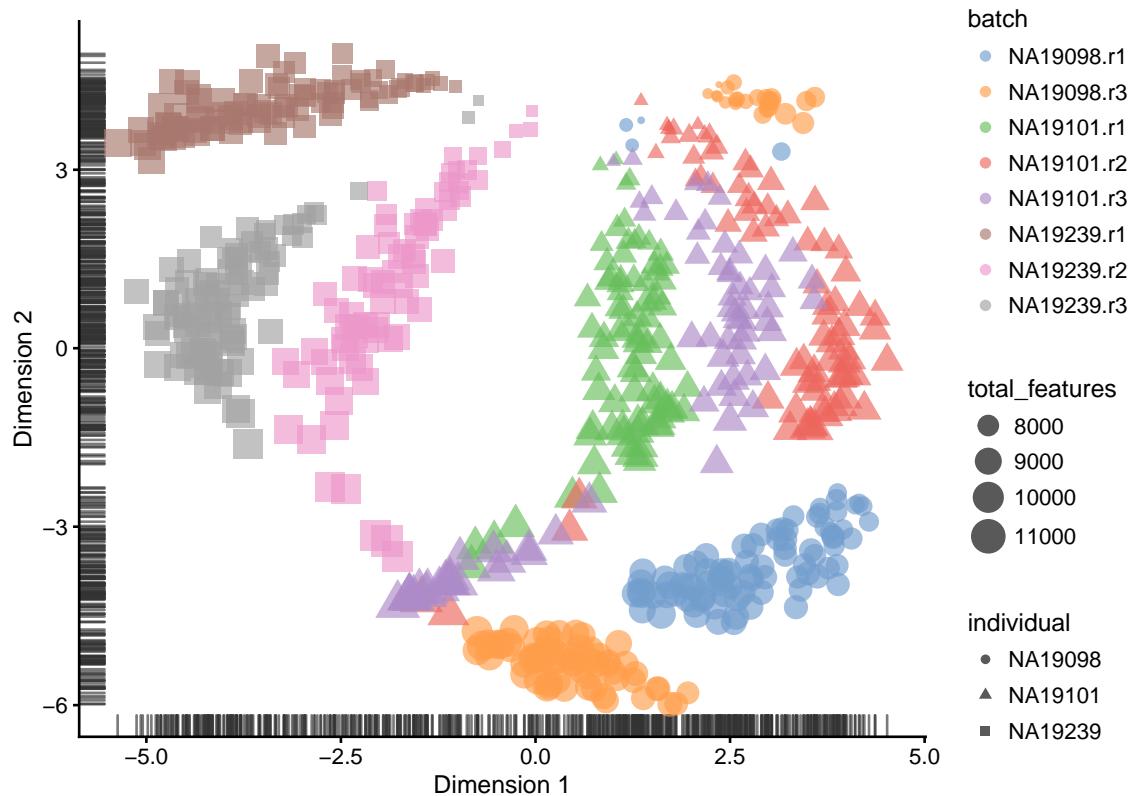


Figure 7.24: tSNE map of the tung data (perplexity = 200)

```
## R version 3.4.3 (2017-11-30)
## Platform: x86_64-pc-linux-gnu (64-bit)
## Running under: Debian GNU/Linux 9 (stretch)
##
## Matrix products: default
## BLAS: /usr/lib/openblas-base/libblas.so.3
## LAPACK: /usr/lib/libopenblas-p-r0.2.19.so
##
## locale:
## [1] LC_CTYPE=en_US.UTF-8          LC_NUMERIC=C
## [3] LC_TIME=en_US.UTF-8          LC_COLLATE=en_US.UTF-8
## [5] LC_MONETARY=en_US.UTF-8       LC_MESSAGES=C
## [7] LC_PAPER=en_US.UTF-8          LC_NAME=C
## [9] LC_ADDRESS=C                  LC_TELEPHONE=C
## [11] LC_MEASUREMENT=en_US.UTF-8   LC_IDENTIFICATION=C
##
## attached base packages:
## [1] stats4    parallel  methods   stats     graphics  grDevices utils
## [8] datasets  base
##
## other attached packages:
## [1] knitr_1.19           scater_1.6.2
## [3] SingleCellExperiment_1.0.0 SummarizedExperiment_1.8.1
## [5] DelayedArray_0.4.1    matrixStats_0.53.0
## [7] GenomicRanges_1.30.1   GenomeInfoDb_1.14.0
```

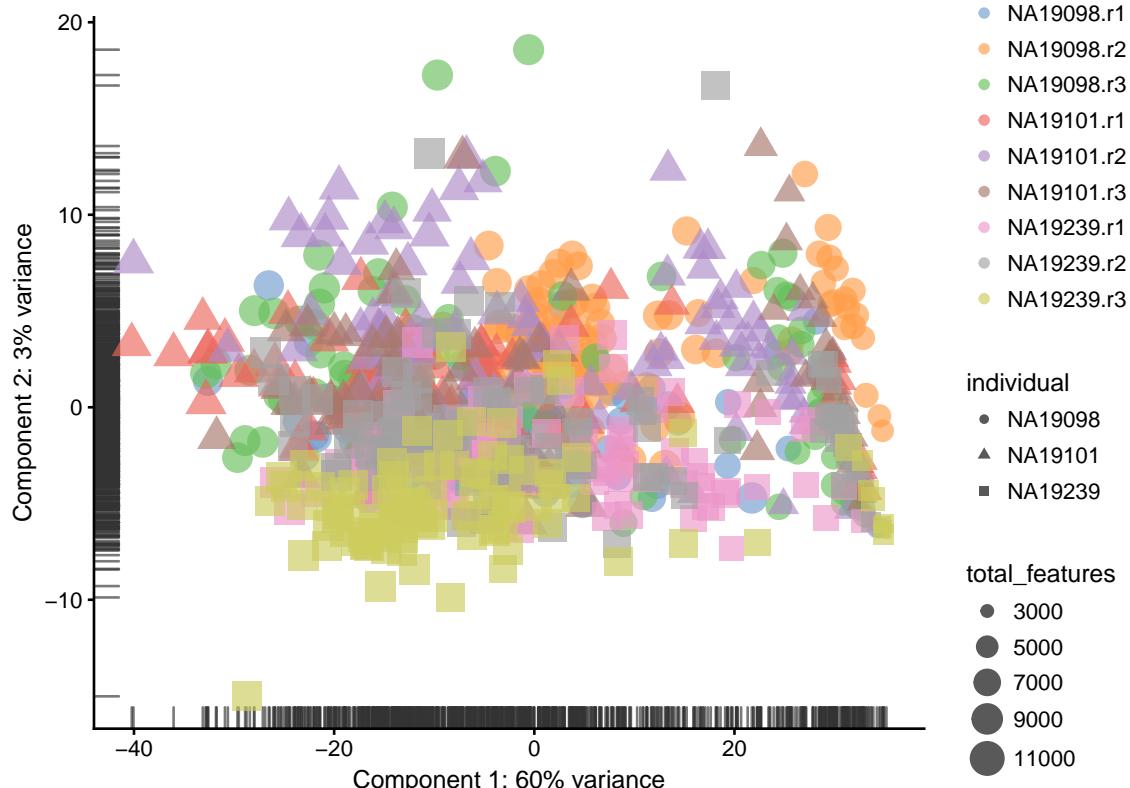


Figure 7.25: PCA plot of the tung data

```

## [ 9] IRanges_2.12.0           S4Vectors_0.16.0
## [11] ggplot2_2.2.1            Biobase_2.38.0
## [13] BiocGenerics_0.24.0
##
## loaded via a namespace (and not attached):
## [ 1] viridis_0.5.0          httr_1.3.1           edgeR_3.20.8
## [ 4] bit64_0.9-7            viridisLite_0.3.0    shiny_1.0.5
## [ 7] assertthat_0.2.0        blob_1.1.0          viper_0.4.5
## [10] GenomeInfoDbData_1.0.0   yaml_2.1.16         progress_1.1.2
## [13] pillar_1.1.0            RSQLite_2.0         backports_1.1.2
## [16] lattice_0.20-34         glue_1.2.0          limma_3.34.8
## [19] digest_0.6.15           XVector_0.18.0       colorspace_1.3-2
## [22] cowplot_0.9.2          htmltools_0.3.6     httpuv_1.3.5
## [25] Matrix_1.2-7.1          plyr_1.8.4          XML_3.98-1.9
## [28] pkgconfig_2.0.1          biomaRt_2.34.2      bookdown_0.6
## [31] zlibbioc_1.24.0         xtable_1.8-2        scales_0.5.0
## [34] Rtsne_0.13              tibble_1.4.2        lazyeval_0.2.1
## [37] magrittr_1.5             mime_0.5           memoise_1.1.0
## [40] evaluate_0.10.1         beeswarm_0.2.3      shinydashboard_0.6.1
## [43] tools_3.4.3             data.table_1.10.4-3 prettyunits_1.0.2
## [46] stringr_1.2.0           munsell_0.4.3       locfit_1.5-9.1
## [49] AnnotationDbi_1.40.0    bindrcpp_0.2        compiler_3.4.3
## [52] rlang_0.1.6              rhdf5_2.22.0        grid_3.4.3
## [55] RCurl_1.95-4.10         tximport_1.6.0      rjson_0.2.15
## [58] labeling_0.3             bitops_1.0-6        rmarkdown_1.8

```

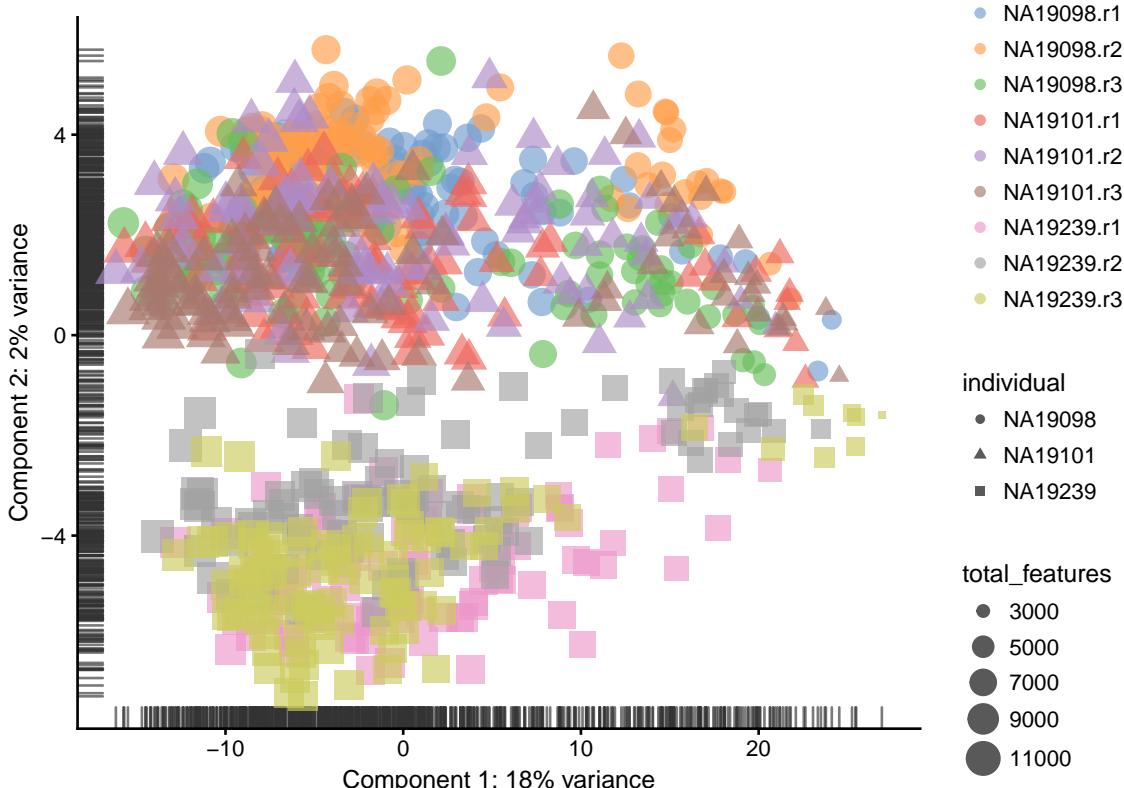


Figure 7.26: PCA plot of the tung data

```
## [61] gtable_0.2.0           DBI_0.7                reshape2_1.4.3
## [64] R6_2.2.2               gridExtra_2.3          dplyr_0.7.4
## [67] bit_1.1-12              bindr_0.1              rprojroot_1.3-2
## [70] ggbeeswarm_0.6.0        stringi_1.1.6         Rcpp_0.12.15
## [73] xfun_0.1
```

## 7.5 Identifying confounding factors

### 7.5.1 Introduction

There is a large number of potential confounders, artifacts and biases in sc-RNA-seq data. One of the main challenges in analyzing scRNA-seq data stems from the fact that it is difficult to carry out a true technical replicate (why?) to distinguish biological and technical variability. In the previous chapters we considered batch effects and in this chapter we will continue to explore how experimental artifacts can be identified and removed. We will continue using the `scater` package since it provides a set of methods specifically for quality control of experimental and explanatory variables. Moreover, we will continue to work with the Blischak data that was used in the previous chapter.

```
library(scater, quietly = TRUE)
options(stringsAsFactors = FALSE)
umi <- readRDS("tung/umi.rds")
umi.qc <- umi$rowData(umi)$use, colData(umi)$use]
endog_genes <- !rowData(umi.qc)$is_feature_control
```

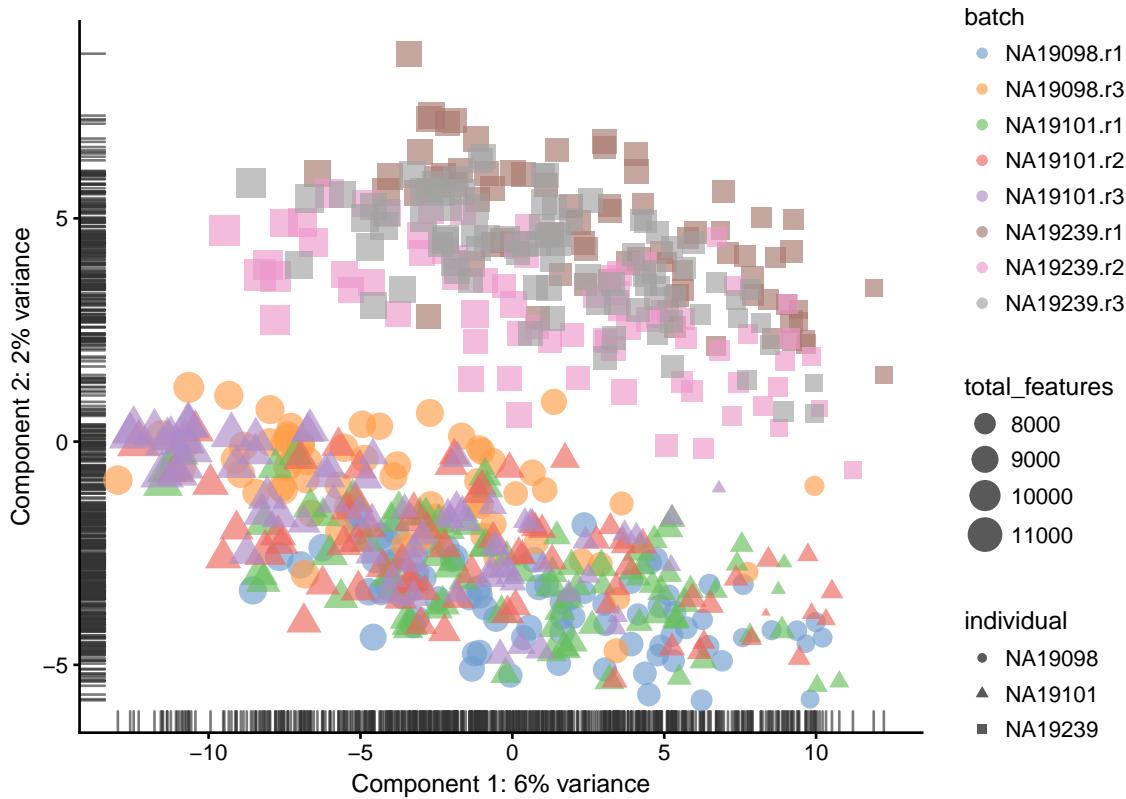


Figure 7.27: PCA plot of the tung data

The `umi.qc` dataset contains filtered cells and genes. Our next step is to explore technical drivers of variability in the data to inform data normalisation before downstream analysis.

### 7.5.2 Correlations with PCs

Let's first look again at the PCA plot of the QCed dataset:

```
plotPCA(
  umi.qc[enod_genes, ],
  exprs_values = "logcounts_raw",
  colour_by = "batch",
  size_by = "total_features"
)
```

`scater` allows one to identify principal components that correlate with experimental and QC variables of interest (it ranks principle components by  $R^2$  from a linear model regressing PC value against the variable of interest).

Let's test whether some of the variables correlate with any of the PCs.

#### 7.5.2.1 Detected genes

```
plotQC(
  umi.qc[enod_genes, ],
  type = "find-pcs",
```

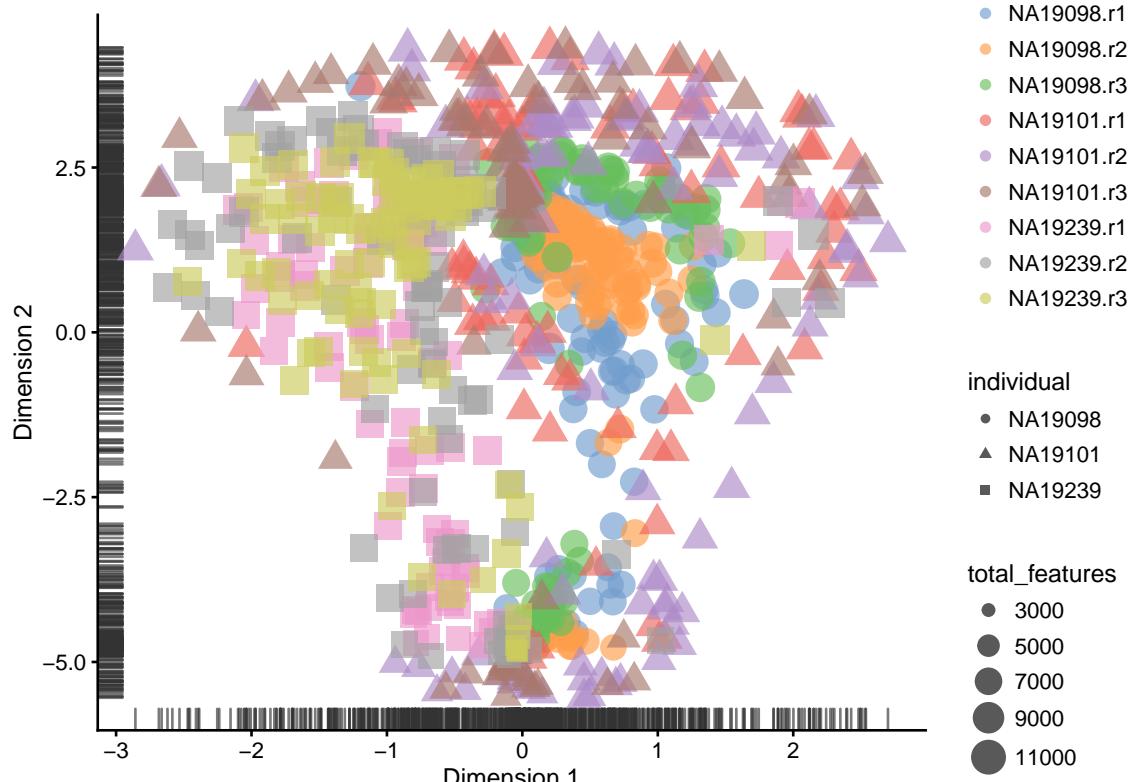


Figure 7.28: tSNE map of the tung data

```
exprs_values = "logcounts_raw",
variable = "total_features"
)
```

Indeed, we can see that PC1 can be almost completely explained by the number of detected genes. In fact, it was also visible on the PCA plot above. This is a well-known issue in scRNA-seq and was described here.

### 7.5.3 Explanatory variables

`scater` can also compute the marginal  $R^2$  for each variable when fitting a linear model regressing expression values for each gene against just that variable, and display a density plot of the gene-wise marginal  $R^2$  values for the variables.

```
plotQC(
  umi.qc[endo_genes, ],
  type = "expl",
  exprs_values = "logcounts_raw",
  variables = c(
    "total_features",
    "total_counts",
    "batch",
    "individual",
    "pct_counts_ERCC",
    "pct_counts_MT"
  )
)
```

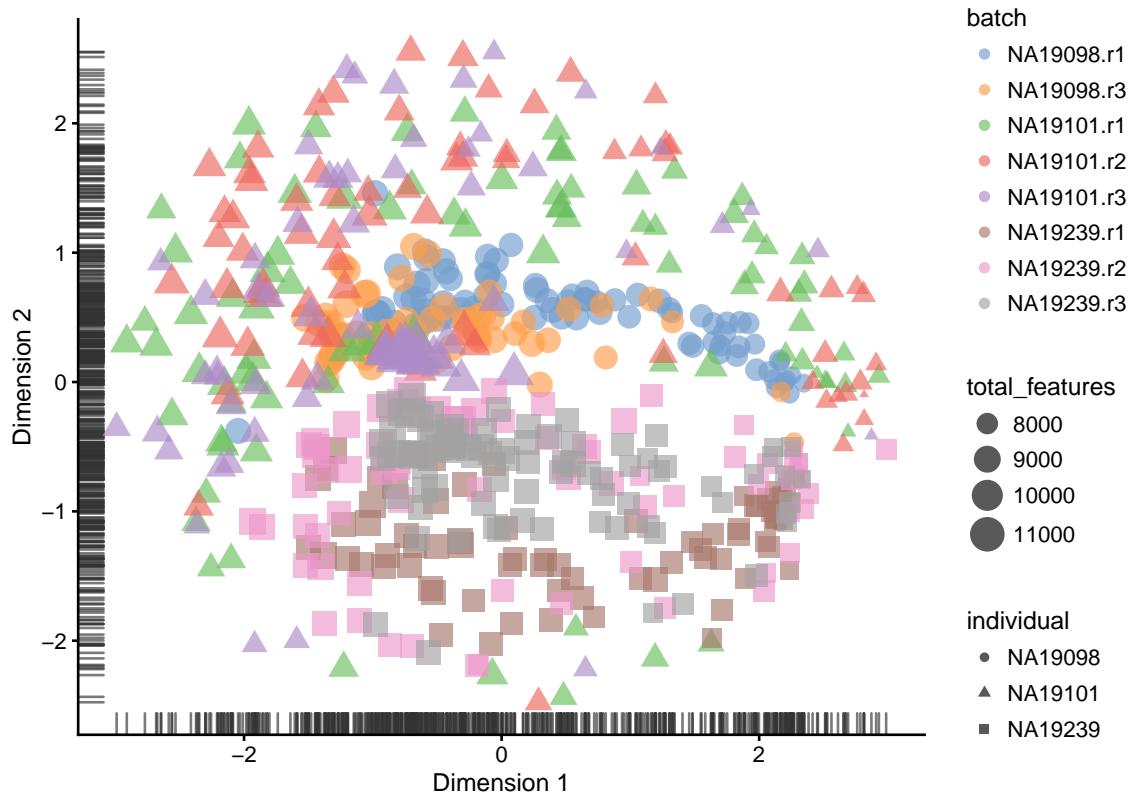


Figure 7.29: tSNE map of the tung data

)

This analysis indicates that the number of detected genes (again) and also the sequencing depth (number of counts) have substantial explanatory power for many genes, so these variables are good candidates for conditioning out in a normalisation step, or including in downstream statistical models. Expression of ERCCs also appears to be an important explanatory variable and one notable feature of the above plot is that batch explains more than individual. What does that tell us about the technical and biological variability of the data?

#### 7.5.4 Other confounders

In addition to correcting for batch, there are other factors that one may want to compensate for. As with batch correction, these adjustments require extrinsic information. One popular method is scLVM which allows you to identify and subtract the effect from processes such as cell-cycle or apoptosis.

In addition, protocols may differ in terms of their coverage of each transcript, their bias based on the average content of A/T nucleotides, or their ability to capture short transcripts. Ideally, we would like to compensate for all of these differences and biases.

#### 7.5.5 Exercise

Perform the same analysis with read counts of the Blischak data. Use `tung/reads.rds` file to load the reads SCESet object. Once you have finished please compare your results to ours (next chapter).

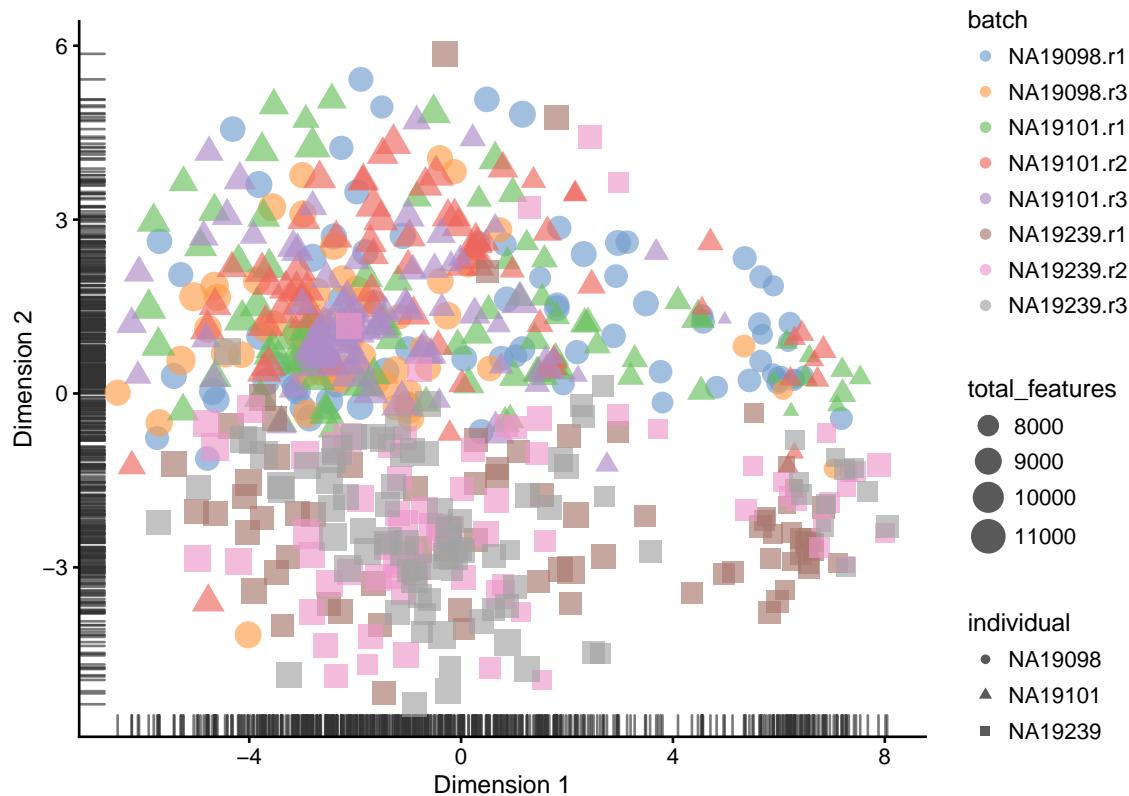


Figure 7.30: tSNE map of the tung data (perplexity = 10)

### 7.5.6 sessionInfo()

```
## R version 3.4.3 (2017-11-30)
## Platform: x86_64-pc-linux-gnu (64-bit)
## Running under: Debian GNU/Linux 9 (stretch)
##
## Matrix products: default
## BLAS: /usr/lib/openblas-base/libblas.so.3
## LAPACK: /usr/lib/libopenblas-p0.2.19.so
##
## locale:
## [1] LC_CTYPE=en_US.UTF-8          LC_NUMERIC=C
## [3] LC_TIME=en_US.UTF-8          LC_COLLATE=en_US.UTF-8
## [5] LC_MONETARY=en_US.UTF-8       LC_MESSAGES=C
## [7] LC_PAPER=en_US.UTF-8          LC_NAME=C
## [9] LC_ADDRESS=C                  LC_TELEPHONE=C
## [11] LC_MEASUREMENT=en_US.UTF-8    LC_IDENTIFICATION=C
##
## attached base packages:
## [1] stats4     parallel   methods   stats      graphics  grDevices utils
## [8] datasets  base
##
## other attached packages:
## [1] scater_1.6.2           SingleCellExperiment_1.0.0
## [3] SummarizedExperiment_1.8.1 DelayedArray_0.4.1
```

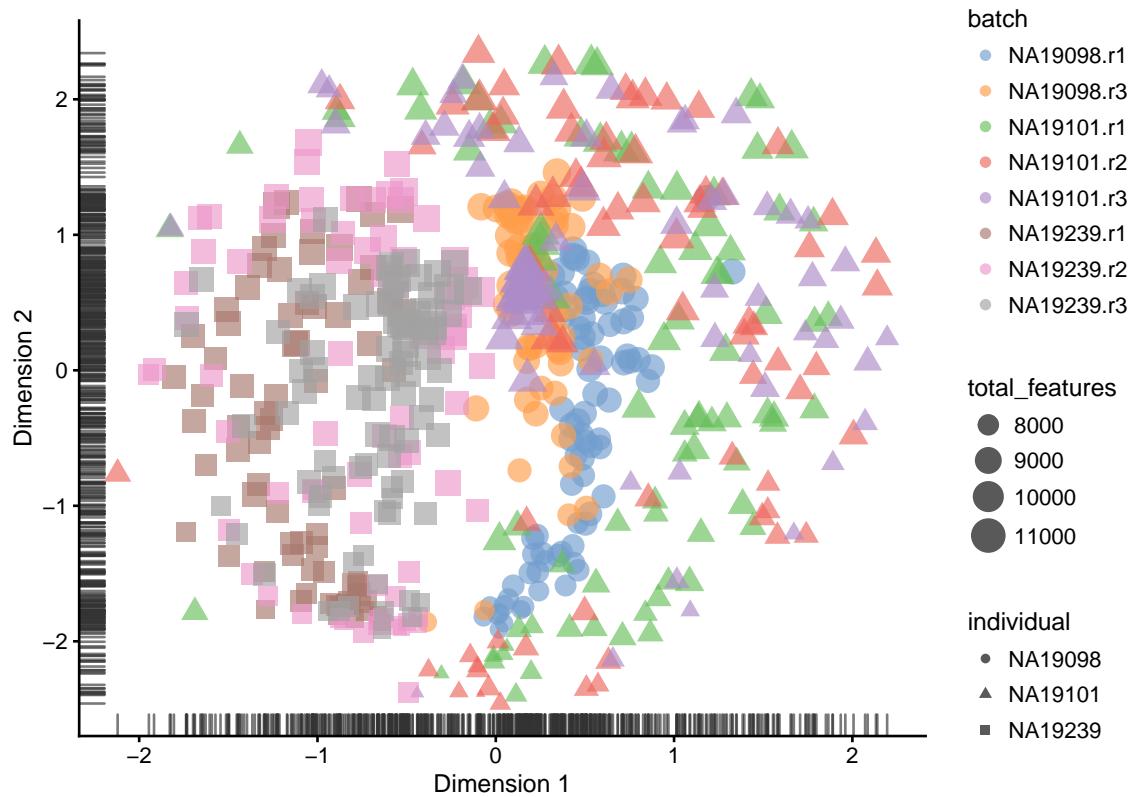


Figure 7.31: tSNE map of the tung data (perplexity = 200)

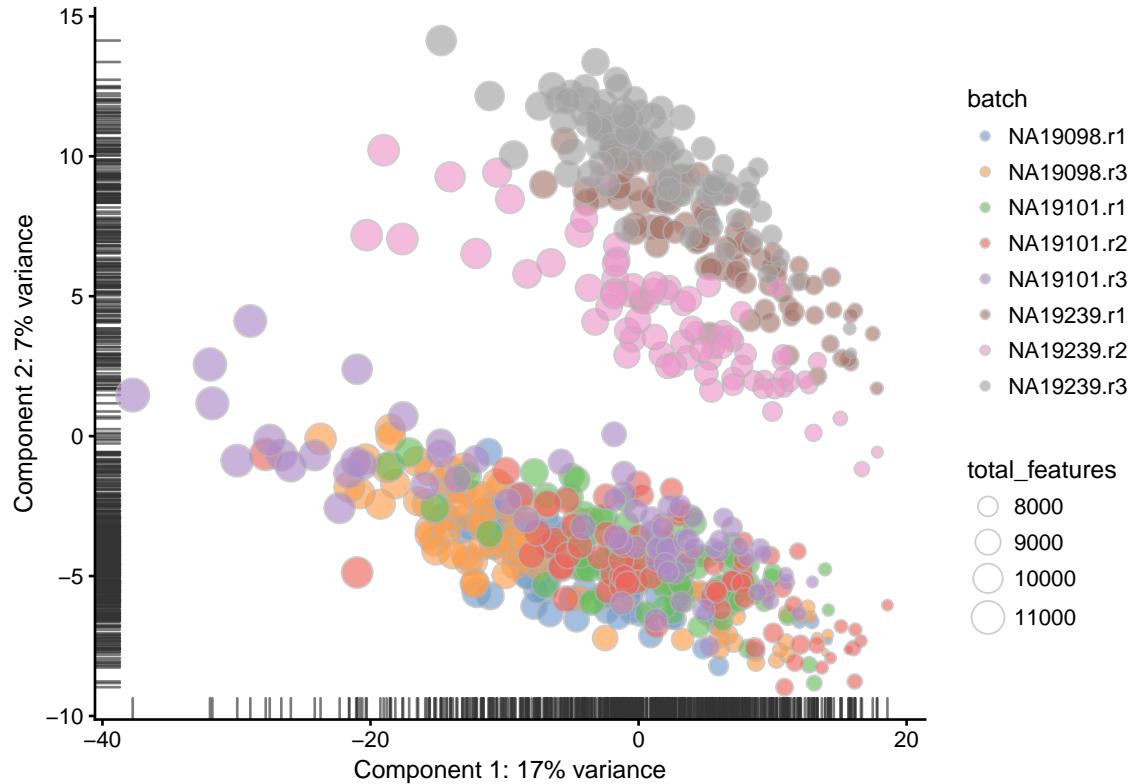


Figure 7.32: PCA plot of the tung data

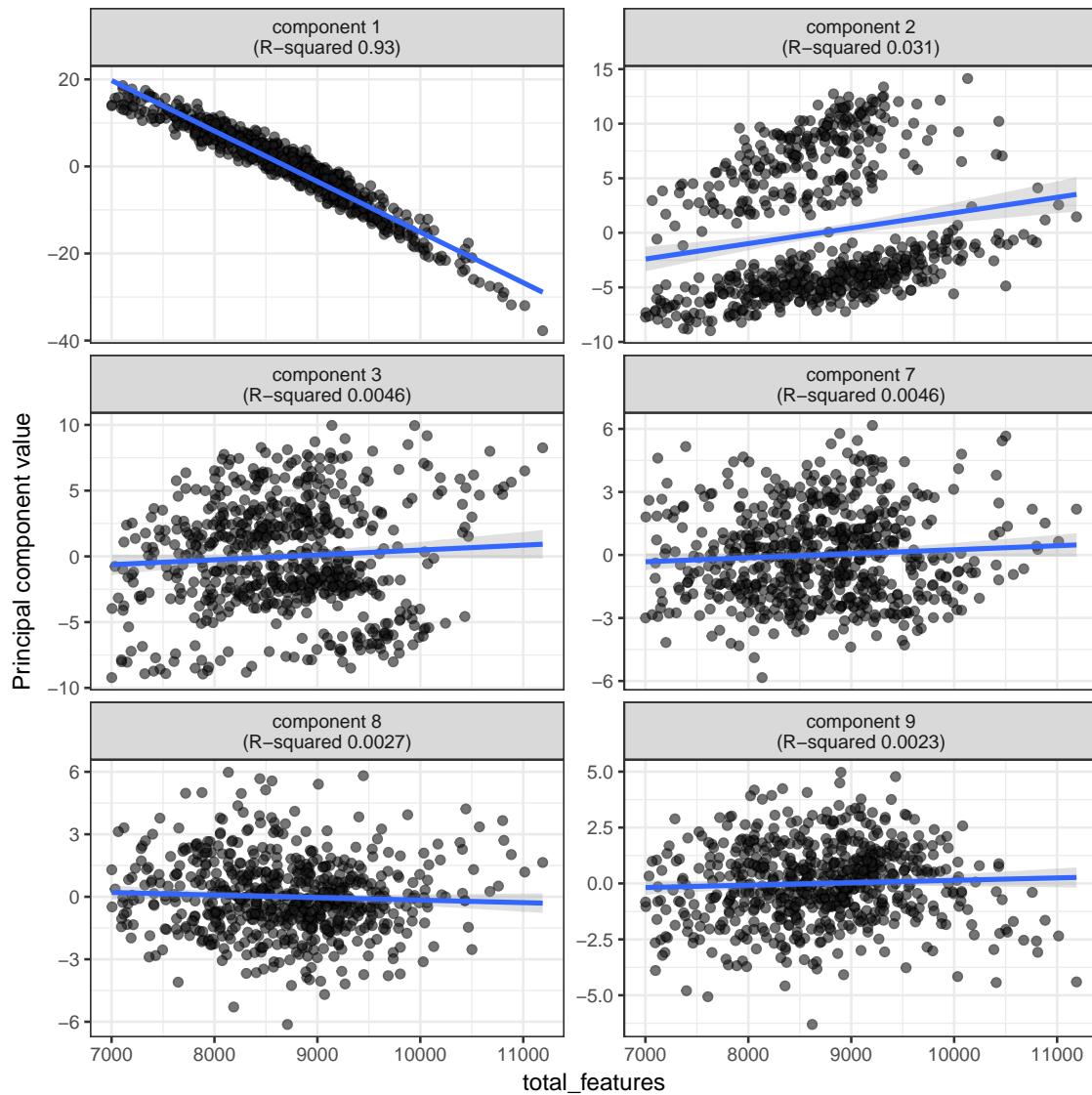


Figure 7.33: PC correlation with the number of detected genes

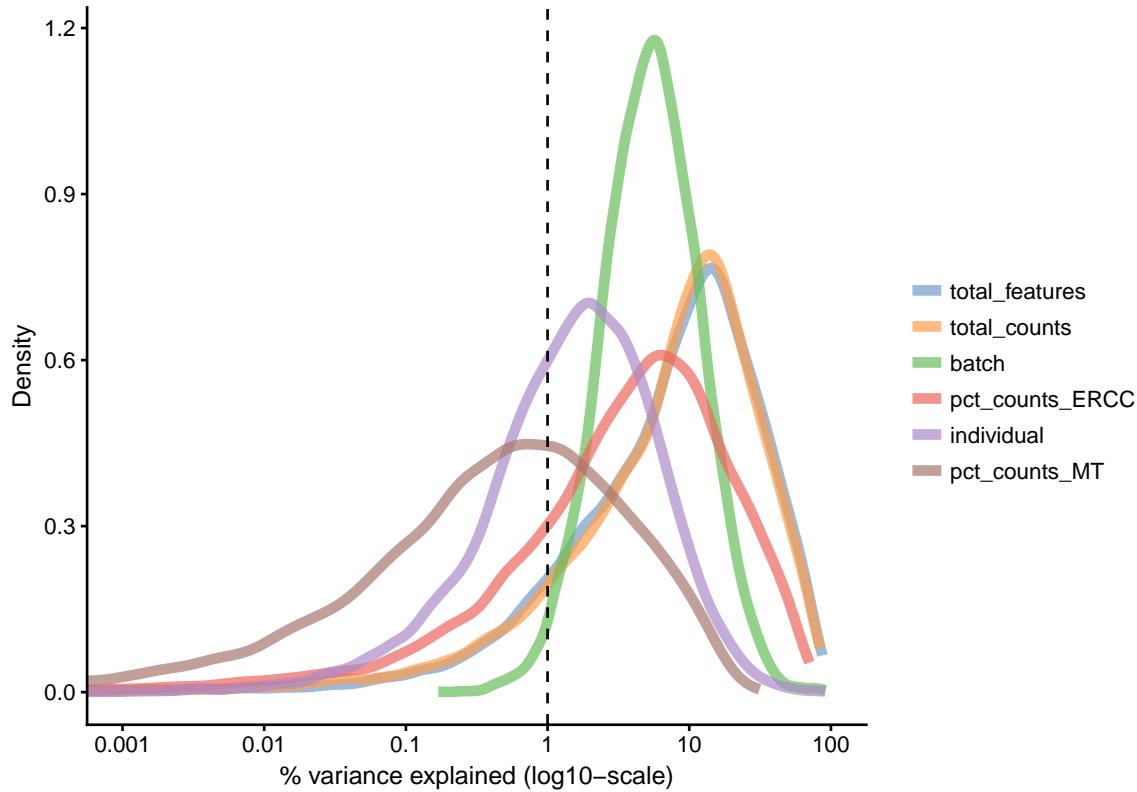


Figure 7.34: Explanatory variables

```

## [5] matrixStats_0.53.0           GenomicRanges_1.30.1
## [7] GenomeInfoDb_1.14.0          IRanges_2.12.0
## [9] S4Vectors_0.16.0            ggplot2_2.2.1
## [11] Biobase_2.38.0              BiocGenerics_0.24.0
## [13] knitr_1.19
##
## loaded via a namespace (and not attached):
## [1] viridis_0.5.0                 httr_1.3.1           edgeR_3.20.8
## [4] bit64_0.9-7                  viridisLite_0.3.0   shiny_1.0.5
## [7] assertthat_0.2.0              blob_1.1.0          viper_0.4.5
## [10] GenomeInfoDbData_1.0.0       yaml_2.1.16         progress_1.1.2
## [13] pillar_1.1.0                RSQLite_2.0          backports_1.1.2
## [16] lattice_0.20-34             glue_1.2.0          limma_3.34.8
## [19] digest_0.6.15               XVector_0.18.0      colorspace_1.3-2
## [22] cowplot_0.9.2              htmltools_0.3.6     httpuv_1.3.5
## [25] Matrix_1.2-7.1              plyr_1.8.4          XML_3.98-1.9
## [28] pkgconfig_2.0.1              biomaRt_2.34.2     bookdown_0.6
## [31] zlibbioc_1.24.0             xtable_1.8-2        scales_0.5.0
## [34] tibble_1.4.2                lazyeval_0.2.1      magrittr_1.5
## [37] mime_0.5                     memoise_1.1.0       evaluate_0.10.1
## [40] beeswarm_0.2.3              shinydashboard_0.6.1 tools_3.4.3
## [43] data.table_1.10.4-3         prettyunits_1.0.2   stringr_1.2.0
## [46] munsell_0.4.3               locfit_1.5-9.1      AnnotationDbi_1.40.0
## [49] bindrcpp_0.2                 compiler_3.4.3      rlang_0.1.6
## [52] rhdf5_2.22.0                grid_3.4.3          RCurl_1.95-4.10

```

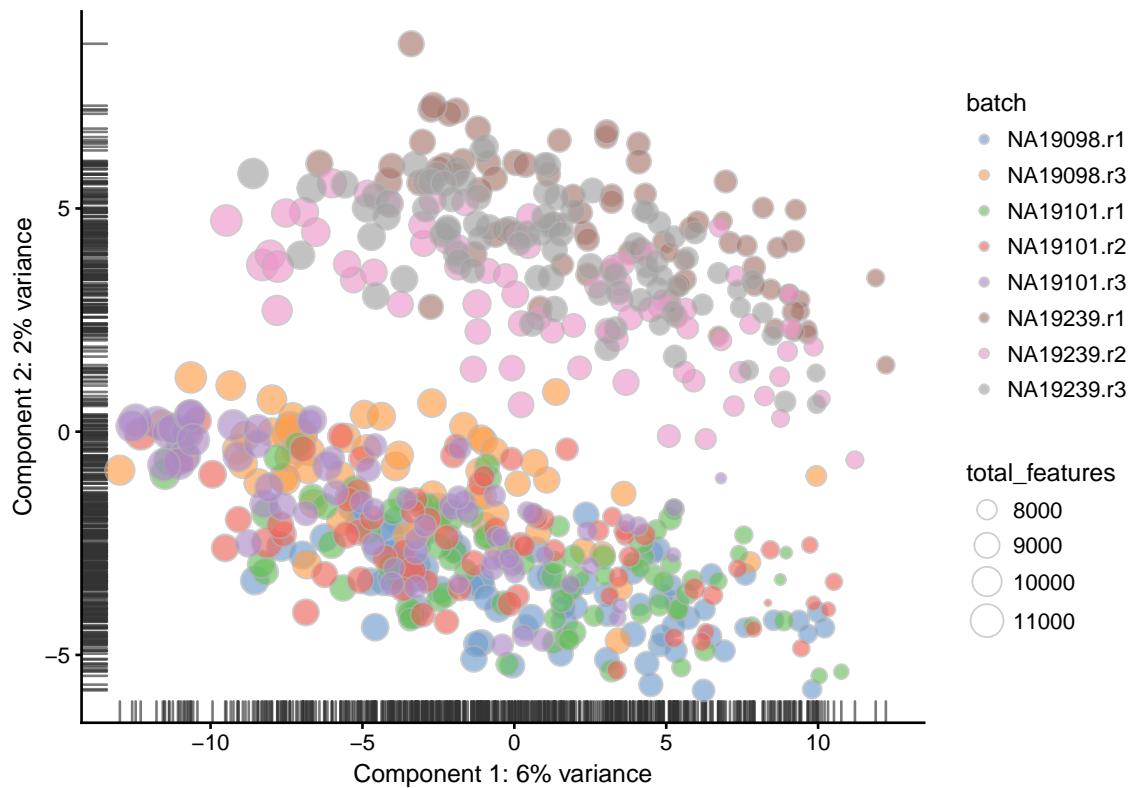


Figure 7.35: PCA plot of the tung data

```

## [55] tximport_1.6.0          rjson_0.2.15           labeling_0.3
## [58] bitops_1.0-6             rmarkdown_1.8          gtable_0.2.0
## [61] DBI_0.7                  reshape2_1.4.3         R6_2.2.2
## [64] gridExtra_2.3            dplyr_0.7.4            bit_1.1-12
## [67] bindr_0.1                 rprojroot_1.3-2       ggbeeswarm_0.6.0
## [70] stringi_1.1.6            Rcpp_0.12.15           xfun_0.1

```

## 7.6 Identifying confounding factors (Reads)

```

## R version 3.4.3 (2017-11-30)
## Platform: x86_64-pc-linux-gnu (64-bit)
## Running under: Debian GNU/Linux 9 (stretch)
##
## Matrix products: default
## BLAS: /usr/lib/openblas-base/libblas.so.3
## LAPACK: /usr/lib/libopenblas-p-r0.2.19.so
##
## locale:
##   [1] LC_CTYPE=en_US.UTF-8      LC_NUMERIC=C
##   [3] LC_TIME=en_US.UTF-8      LC_COLLATE=en_US.UTF-8
##   [5] LC_MONETARY=en_US.UTF-8    LC_MESSAGES=C
##   [7] LC_PAPER=en_US.UTF-8      LC_NAME=C
##   [9] LC_ADDRESS=C              LC_TELEPHONE=C
##  [11] LC_MEASUREMENT=en_US.UTF-8 LC_IDENTIFICATION=C

```

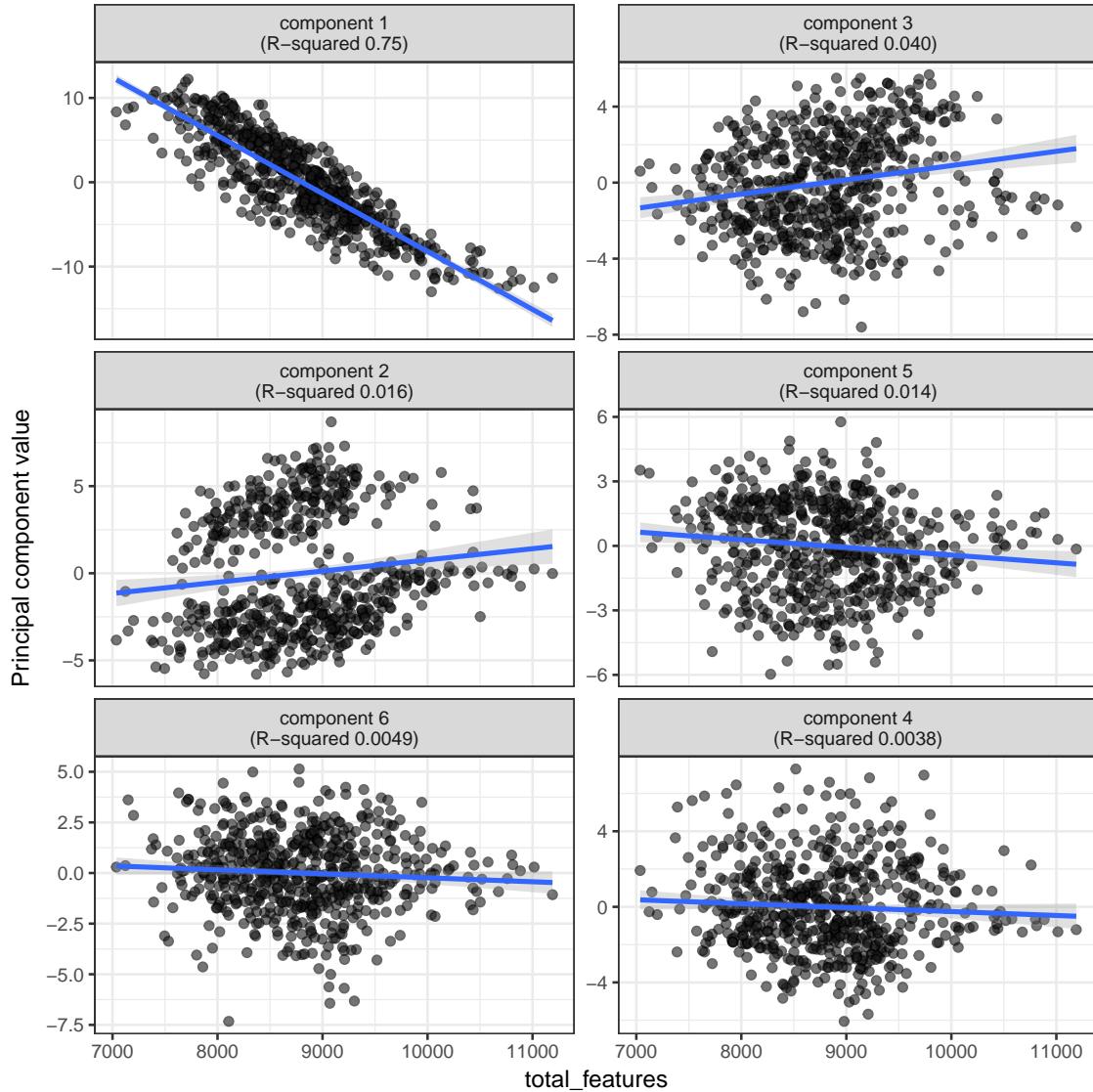


Figure 7.36: PC correlation with the number of detected genes

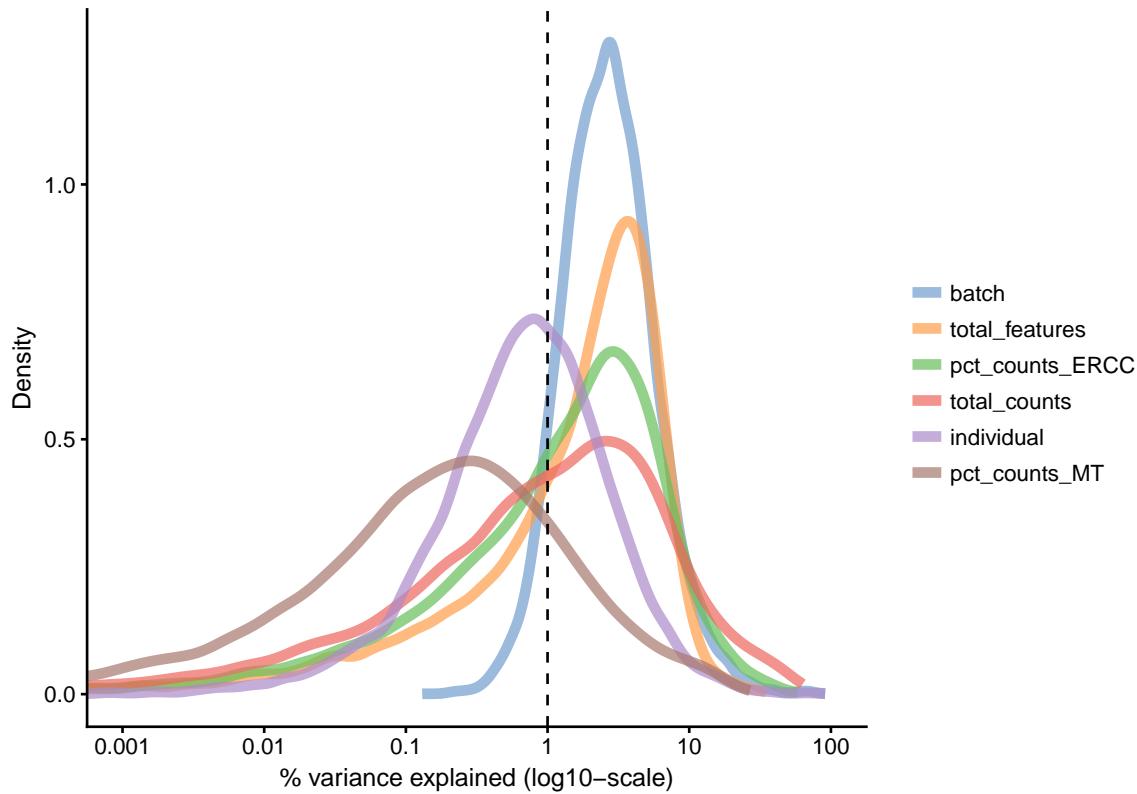


Figure 7.37: Explanatory variables

```
##
## attached base packages:
## [1] stats4      parallel   methods    stats       graphics   grDevices utils
## [8] datasets   base
##
## other attached packages:
## [1] knitr_1.19           scater_1.6.2
## [3] SingleCellExperiment_1.0.0 SummarizedExperiment_1.8.1
## [5] DelayedArray_0.4.1    matrixStats_0.53.0
## [7] GenomicRanges_1.30.1   GenomeInfoDb_1.14.0
## [9] IRanges_2.12.0        S4Vectors_0.16.0
## [11] ggplot2_2.2.1         Biobase_2.38.0
## [13] BiocGenerics_0.24.0
##
## loaded via a namespace (and not attached):
## [1] viridis_0.5.0          httr_1.3.1           edgeR_3.20.8
## [4] bit64_0.9-7            viridisLite_0.3.0  shiny_1.0.5
## [7] assertthat_0.2.0        blob_1.1.0          viper_0.4.5
## [10] GenomeInfoDbData_1.0.0 yaml_2.1.16        progress_1.1.2
## [13] pillar_1.1.0           RSQLite_2.0         backports_1.1.2
## [16] lattice_0.20-34        glue_1.2.0          limma_3.34.8
## [19] digest_0.6.15          XVector_0.18.0     colorspace_1.3-2
## [22] cowplot_0.9.2          htmltools_0.3.6    httpuv_1.3.5
## [25] Matrix_1.2-7.1         plyr_1.8.4          XML_3.98-1.9
## [28] pkgconfig_2.0.1         biomaRt_2.34.2    bookdown_0.6
```

```

## [31] zlibbioc_1.24.0      xtable_1.8-2          scales_0.5.0
## [34] tibble_1.4.2         lazyeval_0.2.1       magrittr_1.5
## [37] mime_0.5             memoise_1.1.0        evaluate_0.10.1
## [40] beeswarm_0.2.3       shinydashboard_0.6.1 tools_3.4.3
## [43] data.table_1.10.4-3  prettyunits_1.0.2   stringr_1.2.0
## [46] munsell_0.4.3        locfit_1.5-9.1      AnnotationDbi_1.40.0
## [49] bindrcpp_0.2          compiler_3.4.3     rlang_0.1.6
## [52] rhdf5_2.22.0          grid_3.4.3         RCurl_1.95-4.10
## [55] tximport_1.6.0        rjson_0.2.15       labeling_0.3
## [58] bitops_1.0-6          rmarkdown_1.8       gtable_0.2.0
## [61] DBI_0.7               reshape2_1.4.3     R6_2.2.2
## [64] gridExtra_2.3         dplyr_0.7.4        bit_1.1-12
## [67] bindr_0.1              rprojroot_1.3-2   ggbeeswarm_0.6.0
## [70] stringi_1.1.6         Rcpp_0.12.15      xfun_0.1

```

## 7.7 Normalization theory

### 7.7.1 Introduction

In the previous chapter we identified important confounding factors and explanatory variables. `scater` allows one to account for these variables in subsequent statistical models or to condition them out using `normaliseExprs()`, if so desired. This can be done by providing a design matrix to `normaliseExprs()`. We are not covering this topic here, but you can try to do it yourself as an exercise.

Instead we will explore how simple size-factor normalisations correcting for library size can remove the effects of some of the confounders and explanatory variables.

### 7.7.2 Library size

Library sizes vary because scRNA-seq data is often sequenced on highly multiplexed platforms the total reads which are derived from each cell may differ substantially. Some quantification methods (eg. `Cufflinks`, `RSEM`) incorporated library size when determining gene expression estimates thus do not require this normalization.

However, if another quantification method was used then library size must be corrected for by multiplying or dividing each column of the expression matrix by a “normalization factor” which is an estimate of the library size relative to the other cells. Many methods to correct for library size have been developed for bulk RNA-seq and can be equally applied to scRNA-seq (eg. **UQ**, **SF**, **CPM**, **RPKM**, **FPKM**, **TPM**).

### 7.7.3 Normalisations

#### 7.7.3.1 CPM

The simplest way to normalize this data is to convert it to counts per million (**CPM**) by dividing each column by its total then multiplying by 1,000,000. Note that spike-ins should be excluded from the calculation of total expression in order to correct for total cell RNA content, therefore we will only use endogenous genes. Example of a **CPM** function in R:

```

calc_cpm <-
function (expr_mat, spikes = NULL)
{
  norm_factor <- colSums(expr_mat[-spikes, ])
}

```

```

    return(t(t(expr_mat)/norm_factor)) * 10^6
}

```

One potential drawback of **CPM** is if your sample contains genes that are both very highly expressed and differentially expressed across the cells. In this case, the total molecules in the cell may depend of whether such genes are on/off in the cell and normalizing by total molecules may hide the differential expression of those genes and/or falsely create differential expression for the remaining genes.

**Note** **RPKM**, **FPKM** and **TPM** are variants on **CPM** which further adjust counts by the length of the respective gene/transcript.

To deal with this potentiality several other measures were devised.

### 7.7.3.2 RLE (SF)

The **size factor (SF)** was proposed and popularized by DESeq (Anders and Huber, 2010). First the geometric mean of each gene across all cells is calculated. The size factor for each cell is the median across genes of the ratio of the expression to the gene's geometric mean. A drawback to this method is that since it uses the geometric mean only genes with non-zero expression values across all cells can be used in its calculation, making it unadvisable for large low-depth scRNASeq experiments. **edgeR** & **scater** call this method **RLE** for “relative log expression”. Example of a **SF** function in R:

```

calc_sf <-
function (expr_mat, spikes = NULL)
{
  geomeans <- exp(rowMeans(log(expr_mat[-spikes, ])))
  SF <- function(cnts) {
    median((cnts/geomeans)[(is.finite(geomeans) & geomeans >
      0)])
  }
  norm_factor <- apply(expr_mat[-spikes, ], 2, SF)
  return(t(t(expr_mat)/norm_factor))
}

```

### 7.7.3.3 UQ

The **upperquartile (UQ)** was proposed by (Bullard et al., 2010). Here each column is divided by the 75% quantile of the counts for each library. Often the calculated quantile is scaled by the median across cells to keep the absolute level of expression relatively consistent. A drawback to this method is that for low-depth scRNASeq experiments the large number of undetected genes may result in the 75% quantile being zero (or close to it). This limitation can be overcome by generalizing the idea and using a higher quantile (eg. the 99% quantile is the default in scater) or by excluding zeros prior to calculating the 75% quantile. Example of a **UQ** function in R:

```

calc_uq <-
function (expr_mat, spikes = NULL)
{
  UQ <- function(x) {
    quantile(x[x > 0], 0.75)
  }
  uq <- unlist(apply(expr_mat[-spikes, ], 2, UQ))
  norm_factor <- uq/median(uq)
  return(t(t(expr_mat)/norm_factor))
}

```

### 7.7.3.4 TMM

Another method is called **TMM** is the weighted trimmed mean of M-values (to the reference) proposed by (Robinson and Oshlack, 2010). The M-values in question are the gene-wise log<sub>2</sub> fold changes between individual cells. One cell is used as the reference then the M-values for each other cell is calculated compared to this reference. These values are then trimmed by removing the top and bottom ~30%, and the average of the remaining values is calculated by weighting them to account for the effect of the log scale on variance. Each non-reference cell is multiplied by the calculated factor. Two potential issues with this method are insufficient non-zero genes left after trimming, and the assumption that most genes are not differentially expressed.

### 7.7.3.5 scran

**scran** package implements a variant on **CPM** specialized for single-cell data (L. Lun et al., 2016). Briefly this method deals with the problem of vary large numbers of zero values per cell by pooling cells together calculating a normalization factor (similar to **CPM**) for the sum of each pool. Since each cell is found in many different pools, cell-specific factors can be deconvoluted from the collection of pool-specific factors using linear algebra.

### 7.7.3.6 Downsampling

A final way to correct for library size is to downsample the expression matrix so that each cell has approximately the same total number of molecules. The benefit of this method is that zero values will be introduced by the down sampling thus eliminating any biases due to differing numbers of detected genes. However, the major drawback is that the process is not deterministic so each time the downsampling is run the resulting expression matrix is slightly different. Thus, often analyses must be run on multiple downsamplings to ensure results are robust. Example of a **downsampling** function in R:

```
Down_Sample_Matrix <-
function (expr_mat)
{
  min_lib_size <- min(colSums(expr_mat))
  down_sample <- function(x) {
    prob <- min_lib_size/sum(x)
    return(unlist(lapply(x, function(y) {
      rbinom(1, y, prob)
    })))
  }
  down_sampled_mat <- apply(expr_mat, 2, down_sample)
  return(down_sampled_mat)
}
```

### 7.7.4 Effectiveness

to compare the efficiency of different normalization methods we will use visual inspection of PCA plots and calculation of cell-wise *relative log expression* via **scater**'s **plotRLE()** function. Namely, cells with many (few) reads have higher (lower) than median expression for most genes resulting in a positive (negative) *RLE* across the cell, whereas normalized cells have an *RLE* close to zero. Example of a *RLE* function in R:

```
calc_cell_RLE <-
function (expr_mat, spikes = NULL)
{
  RLE_gene <- function(x) {
```

```

    if (median(unlist(x)) > 0) {
      log((x + 1)/(median(unlist(x)) + 1))/log(2)
    }
    else {
      rep(NA, times = length(x))
    }
  }
  if (!is.null(spikes)) {
    RLE_matrix <- t(apply(expr_mat[-spikes, ], 1, RLE_gene))
  }
  else {
    RLE_matrix <- t(apply(expr_mat, 1, RLE_gene))
  }
  cell_RLE <- apply(RLE_matrix, 2, median, na.rm = T)
  return(cell_RLE)
}

```

**Note** The **RLE**, **TMM**, and **UQ** size-factor methods were developed for bulk RNA-seq data and, depending on the experimental context, may not be appropriate for single-cell RNA-seq data, as their underlying assumptions may be problematically violated.

**Note** `scater` acts as a wrapper for the `calcNormFactors` function from `edgeR` which implements several library size normalization methods making it easy to apply any of these methods to our data.

**Note** `edgeR` makes extra adjustments to some of the normalization methods which may result in somewhat different results than if the original methods are followed exactly, e.g. `edgeR`'s and `scater`'s “RLE” method which is based on the “size factor” used by `DESeq` may give different results to the `estimateSizeFactorsForMatrix` method in the `DESeq`/`DESeq2` packages. In addition, some versions of `edgeR` will not calculate the normalization factors correctly unless `lib.size` is set at 1 for all cells.

**Note** For **CPM** normalisation we use `scater`'s `calculateCPM()` function. For **RLE**, **UQ** and **TMM** we use `scater`'s `normaliseExprs()` function. For `scran` we use `scran` package to calculate size factors (it also operates on `SingleCellExperiment` class) and `scater`'s `normalize()` to normalise the data. All these normalization functions save the results to the `logcounts` slot of the `SCE` object. For **downsampling** we use our own functions shown above.

## 7.8 Normalization practice (UMI)

We will continue to work with the `tung` data that was used in the previous chapter.

```

library(scRNA.seq.funcs)
library(scater)
library(scran)
options(stringsAsFactors = FALSE)
set.seed(1234567)
umi <- readRDS("tung/umi.rds")
umi.qc <- umi$rowData(umi)$use, colData(umi)$use]
endog_genes <- !rowData(umi.qc)$is_feature_control

```

### 7.8.1 Raw

```

plotPCA(
  umi.qc[endog_genes, ],

```

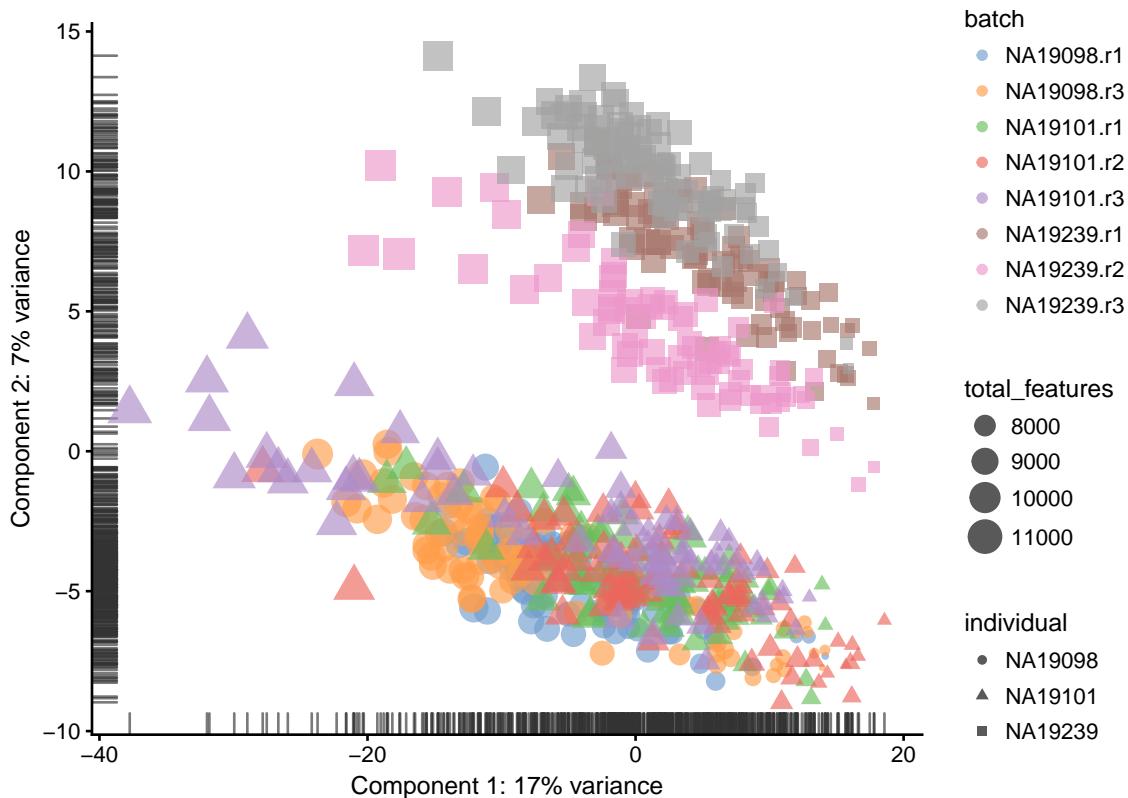


Figure 7.38: PCA plot of the tung data

```

exprs_values = "logcounts_raw",
colour_by = "batch",
size_by = "total_features",
shape_by = "individual"
)

```

### 7.8.2 CPM

```

logcounts(umi.qc) <- log2(calculateCPM(umi.qc, use.size.factors = FALSE) + 1)
plotPCA(
  umi.qc[enodg_genes, ],
  colour_by = "batch",
  size_by = "total_features",
  shape_by = "individual"
)

plotRLE(
  umi.qc[enodg_genes, ],
  exprs_mats = list(Raw = "logcounts_raw", CPM = "logcounts"),
  exprs_logged = c(TRUE, TRUE),
  colour_by = "batch"
)

```

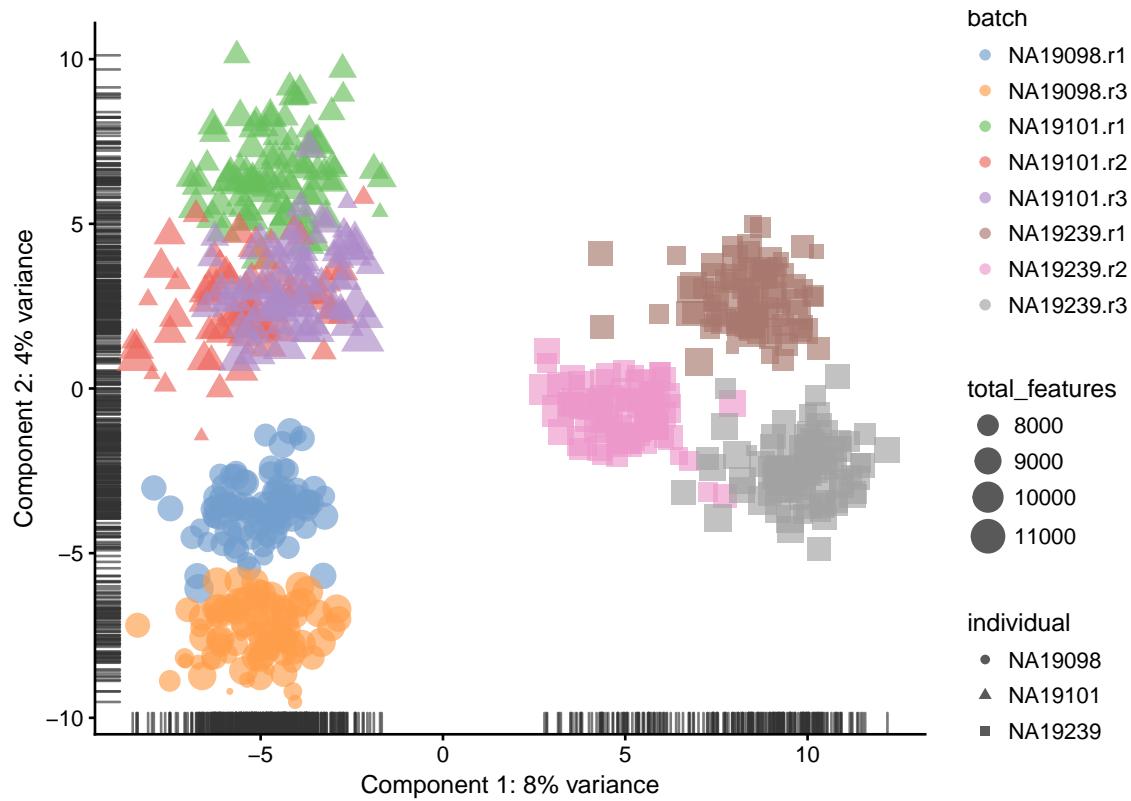


Figure 7.39: PCA plot of the tung data after CPM normalisation

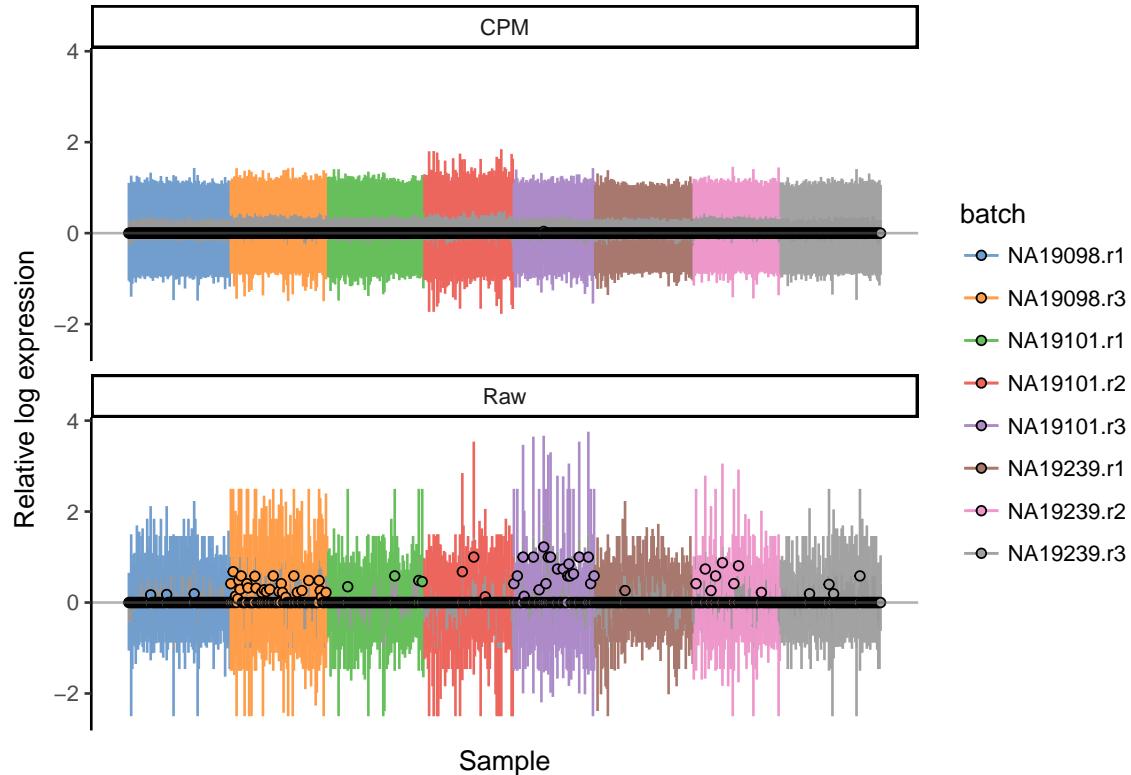


Figure 7.40: Cell-wise RLE of the tung data

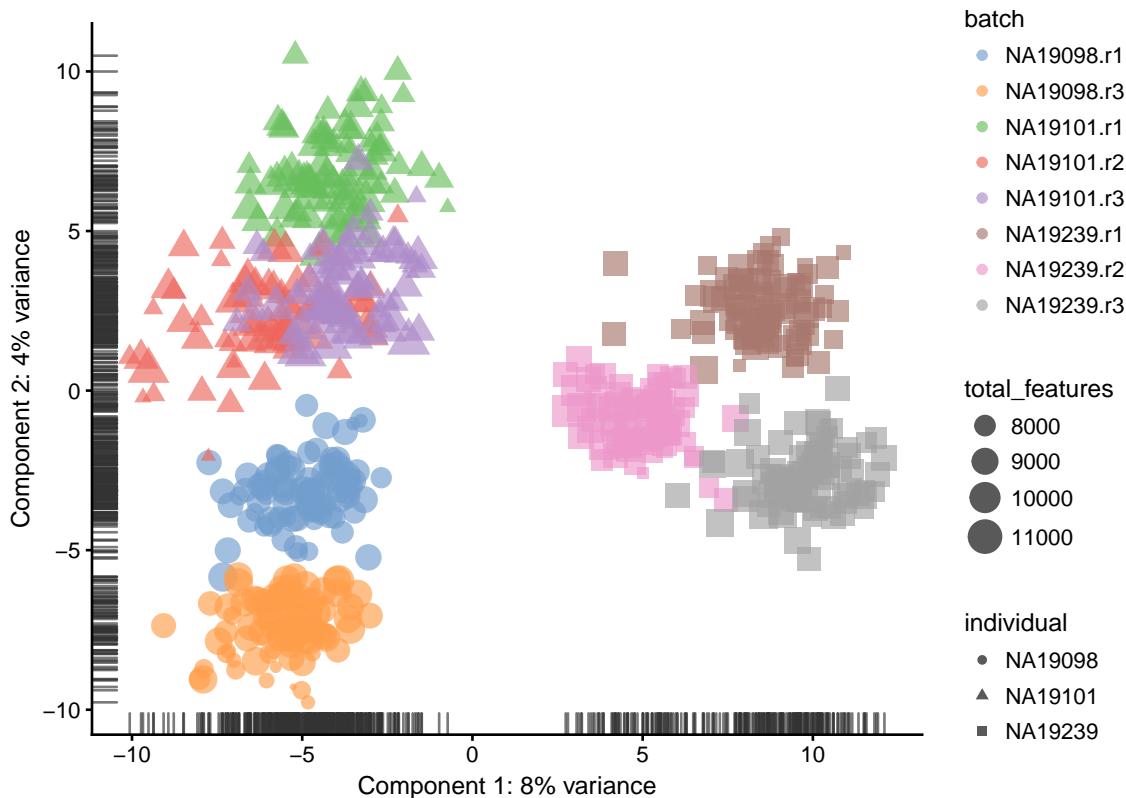


Figure 7.41: PCA plot of the tung data after RLE normalisation

### 7.8.3 Size-factor (RLE)

```

umi.qc <- normaliseExprs(
  umi.qc,
  method = "RLE",
  feature_set = endog_genes,
  return_log = TRUE,
  return_norm_as_exprs = TRUE
)

## Warning in normalizeSCE(object, exprs_values = exprs_values, return_log
## = return_log, : spike-in transcripts in 'ERCC' should have their own size
## factors

plotPCA(
  umi.qc[endog_genes, ],
  colour_by = "batch",
  size_by = "total_features",
  shape_by = "individual"
)

plotRLE(
  umi.qc[endog_genes, ],
  exprs_mats = list(Raw = "logcounts_raw", RLE = "logcounts"),
  exprs_logged = c(TRUE, TRUE),
  colour_by = "batch"
)

```

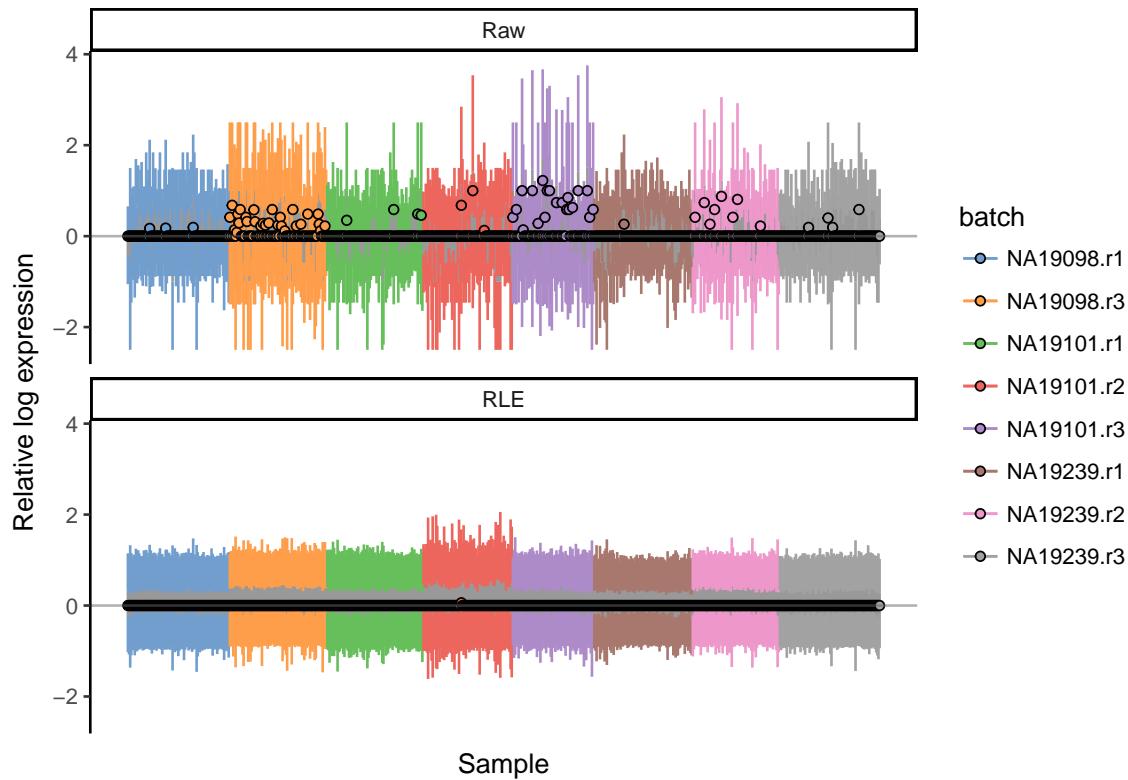


Figure 7.42: Cell-wise RLE of the tung data

)

#### 7.8.4 Upperquantile

```

umi.qc <- normaliseExprs(
  umi.qc,
  method = "upperquartile",
  feature_set = endog_genes,
  p = 0.99,
  return_log = TRUE,
  return_norm_as_exprs = TRUE
)

## Warning in normalizeSCE(object, exprs_values = exprs_values, return_log
## = return_log, : spike-in transcripts in 'ERCC' should have their own size
## factors

plotPCA(
  umi.qc[endog_genes, ],
  colour_by = "batch",
  size_by = "total_features",
  shape_by = "individual"
)

```

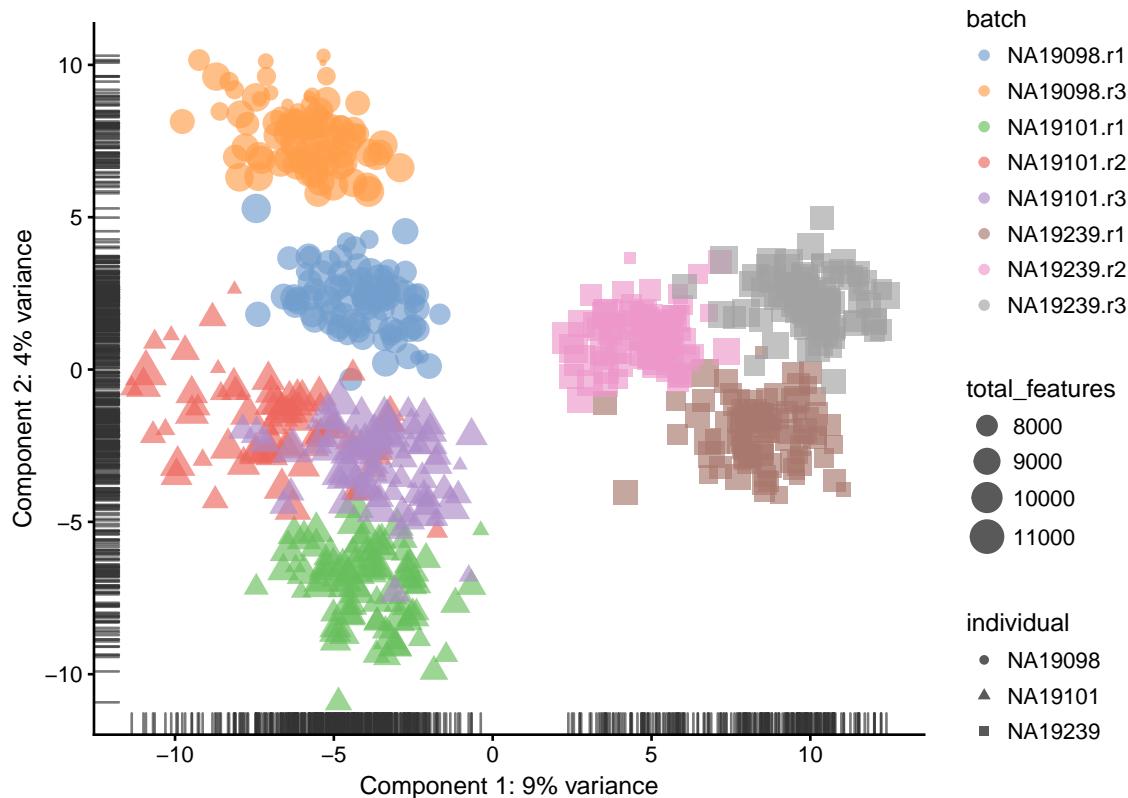


Figure 7.43: PCA plot of the tung data after UQ normalisation

```
plotRLE(
  umi.qc[endog_genes, ],
  exprs_mats = list(Raw = "logcounts_raw", UQ = "logcounts"),
  exprs_logged = c(TRUE, TRUE),
  colour_by = "batch"
)
```

### 7.8.5 TMM

```
umi.qc <- normaliseExprs(
  umi.qc,
  method = "TMM",
  feature_set = endog_genes,
  return_log = TRUE,
  return_norm_as_exprs = TRUE
)

## Warning in normalizeSCE(object, exprs_values = exprs_values, return_log
## = return_log, : spike-in transcripts in 'ERCC' should have their own size
## factors

plotPCA(
  umi.qc[endog_genes, ],
  colour_by = "batch",
```

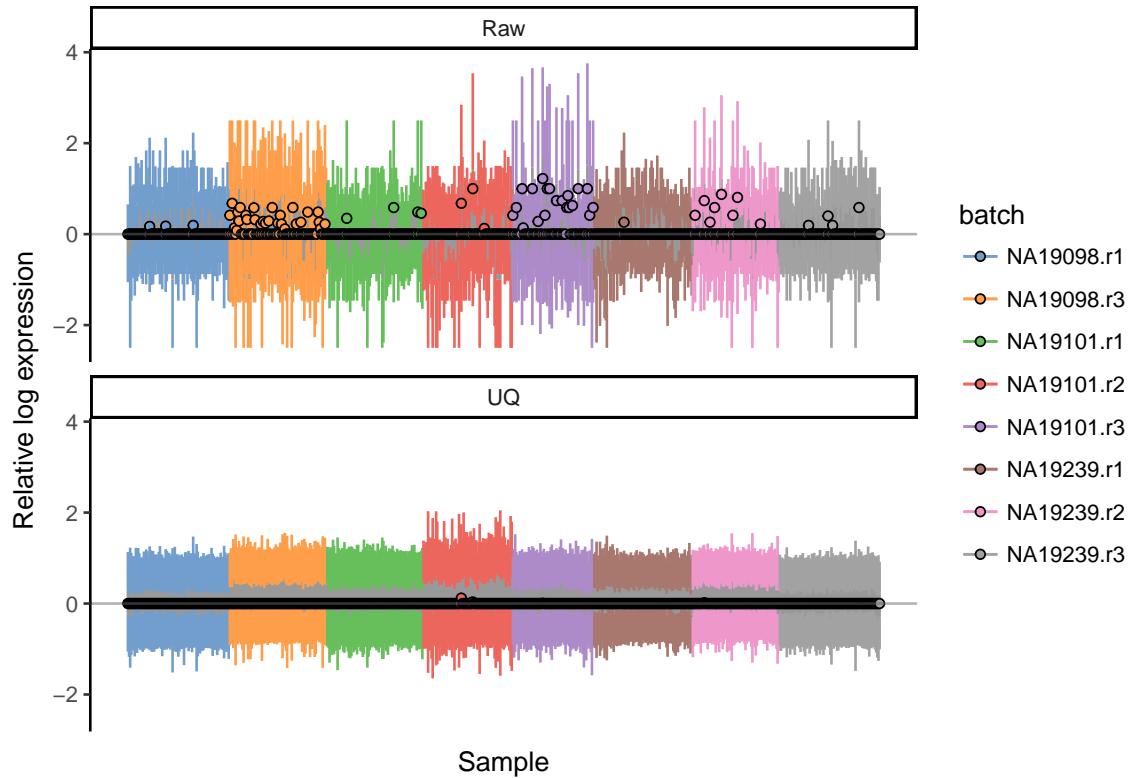


Figure 7.44: Cell-wise RLE of the tung data

```

    size_by = "total_features",
    shape_by = "individual"
)

plotRLE(
  umi.qc[enod_genes, ],
  exprs_mats = list(Raw = "logcounts_raw", TMM = "logcounts"),
  exprs_logged = c(TRUE, TRUE),
  colour_by = "batch"
)

```

### 7.8.6 scran

```

qclust <- quickCluster(umi.qc, min.size = 30)
umi.qc <- computeSumFactors(umi.qc, sizes = 15, clusters = qclust)
umi.qc <- normalize(umi.qc)

## Warning in .local(object, ...): spike-in transcripts in 'ERCC' should have
## their own size factors

plotPCA(
  umi.qc[enod_genes, ],
  colour_by = "batch",
  size_by = "total_features",
  shape_by = "individual"
)

```

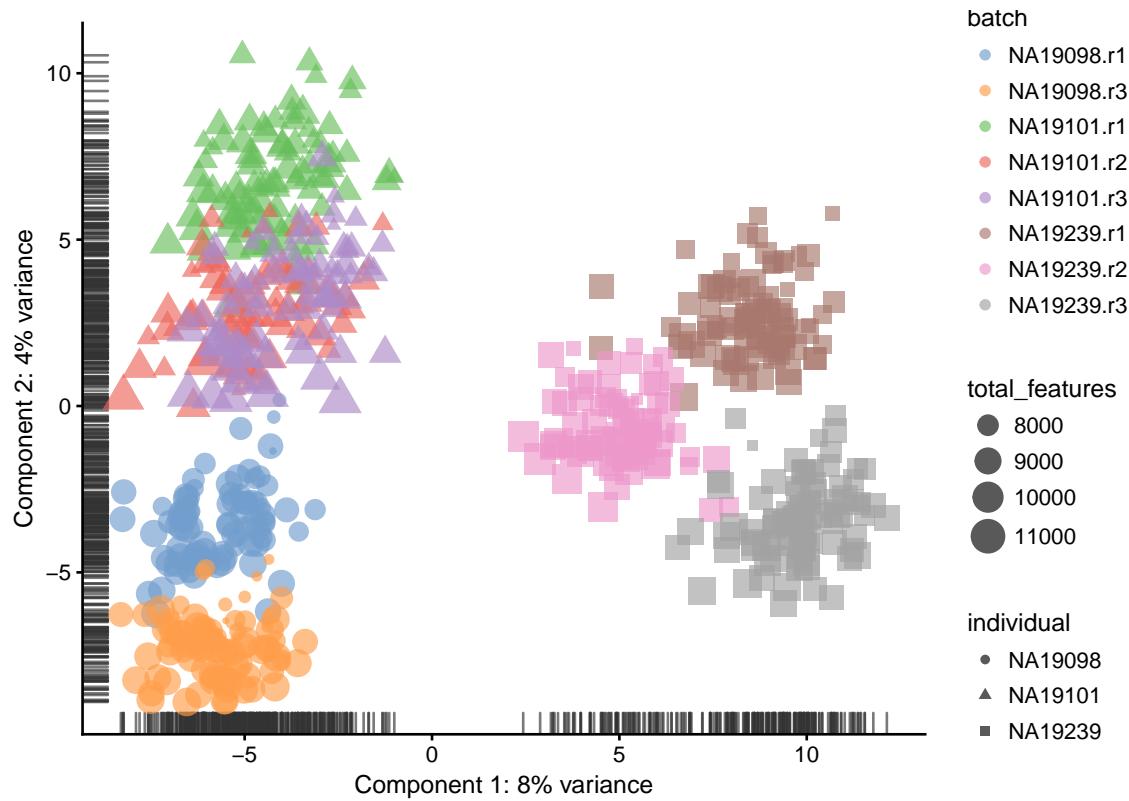


Figure 7.45: PCA plot of the tung data after TMM normalisation

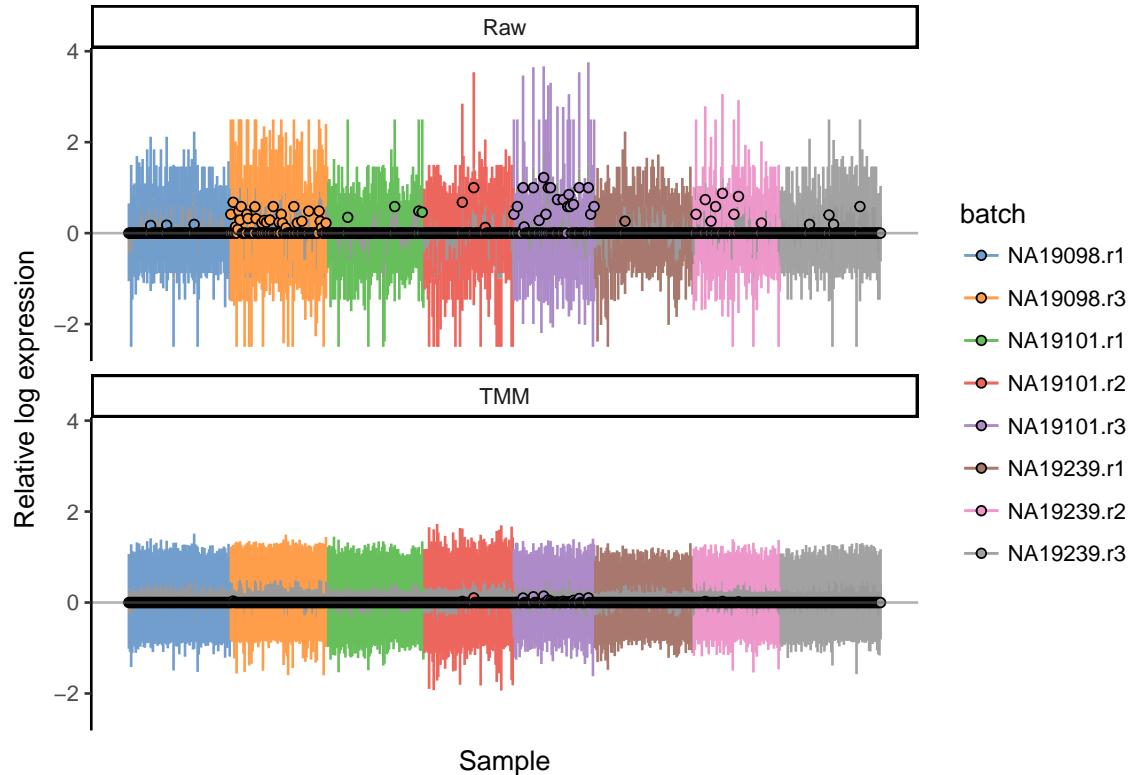


Figure 7.46: Cell-wise RLE of the tung data

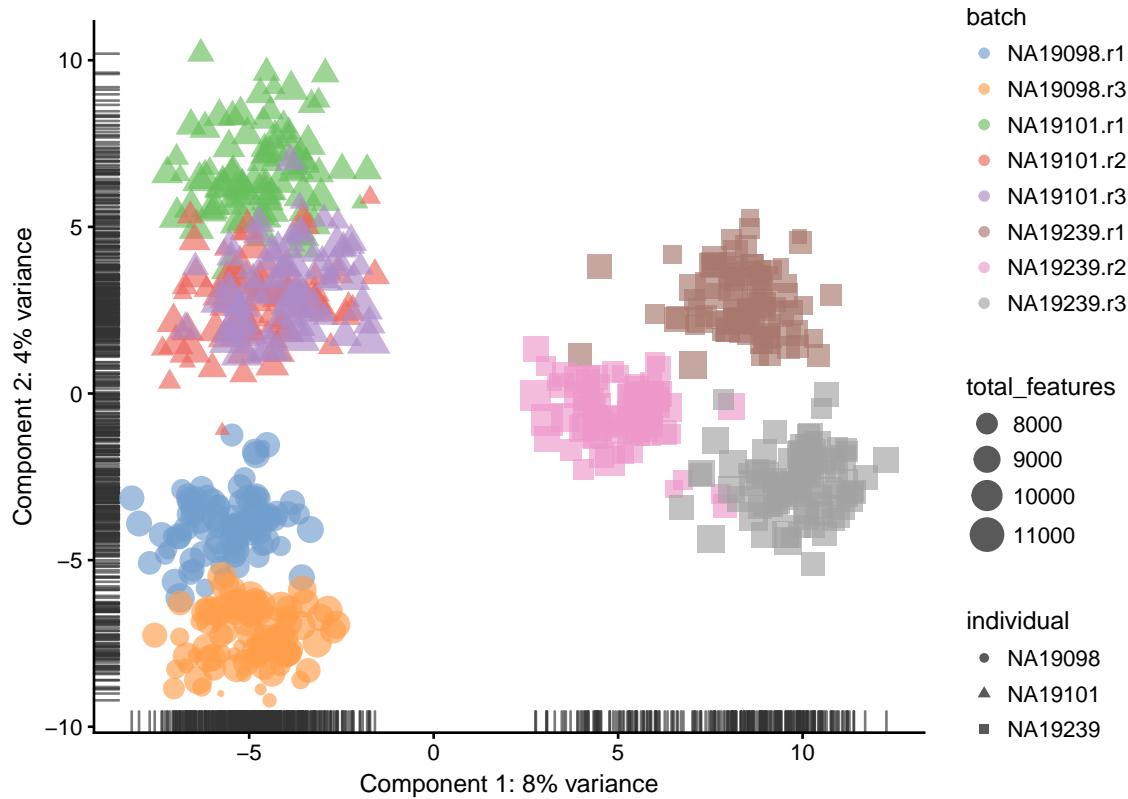


Figure 7.47: PCA plot of the tung data after LSF normalisation

```
)
plotRLE(
  umi.qc[enod_genes, ],
  exprs_mats = list(Raw = "logcounts_raw", scran = "logcounts"),
  exprs_logged = c(TRUE, TRUE),
  colour_by = "batch"
)
```

scran sometimes calculates negative or zero size factors. These will completely distort the normalized expression matrix. We can check the size factors scran has computed like so:

```
summary(sizeFactors(umi.qc))
```

```
##      Min. 1st Qu. Median     Mean 3rd Qu.    Max.
##  0.4646  0.7768  0.9562  1.0000  1.1444  3.4348
```

For this dataset all the size factors are reasonable so we are done. If you find scran has calculated negative size factors try increasing the cluster and pool sizes until they are all positive.

### 7.8.7 Downsampling

```
logcounts(umi.qc) <- log2(Down_Sample_Matrix(counts(umi.qc)) + 1)
plotPCA(
  umi.qc[enod_genes, ],
```

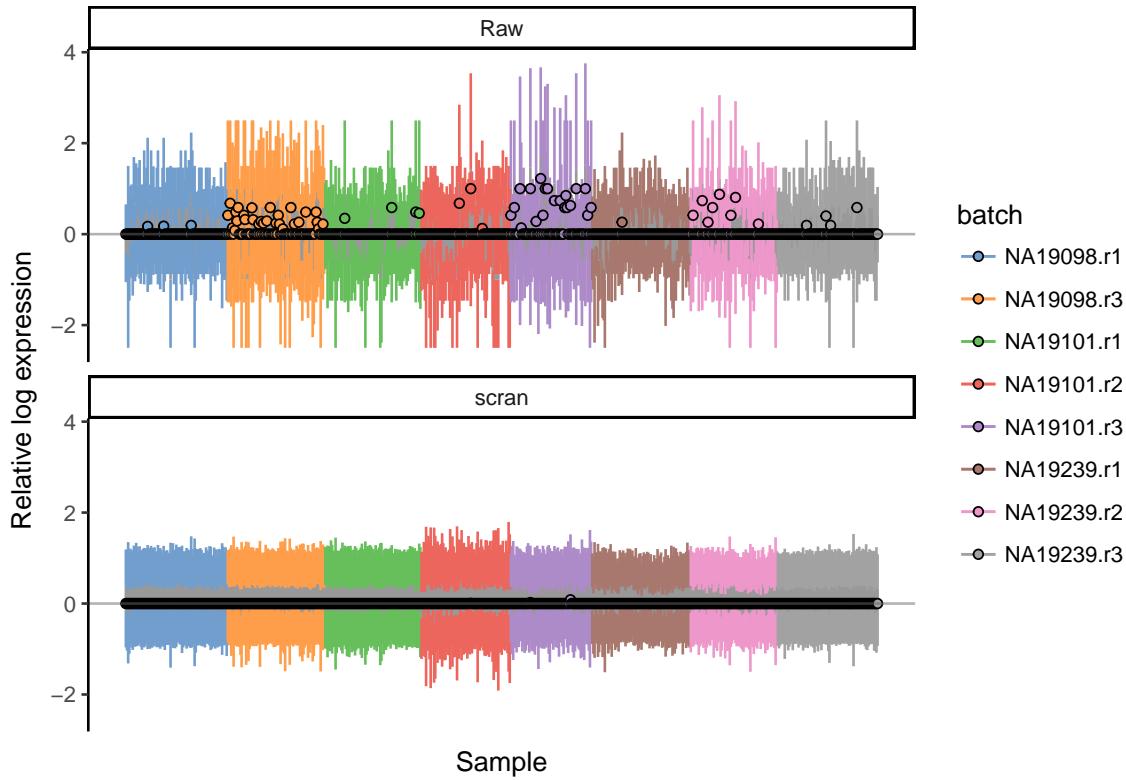


Figure 7.48: Cell-wise RLE of the tung data

```

colour_by = "batch",
size_by = "total_features",
shape_by = "individual"
)

plotRLE(
  umi.qc[enod_genes, ],
  exprs_mats = list(Raw = "logcounts_raw", DownSample = "logcounts"),
  exprs_logged = c(TRUE, TRUE),
  colour_by = "batch"
)

```

### 7.8.8 Normalisation for gene/transcript length

Some methods combine library size and fragment/gene length normalization such as:

- **RPKM** - Reads Per Kilobase Million (for single-end sequencing)
- **FPKM** - Fragments Per Kilobase Million (same as **RPKM** but for paired-end sequencing, makes sure that paired ends mapped to the same fragment are not counted twice)
- **TPM** - Transcripts Per Kilobase Million (same as **RPKM**, but the order of normalizations is reversed - length first and sequencing depth second)

These methods are not applicable to our dataset since the end of the transcript which contains the UMI was preferentially sequenced. Furthermore in general these should only be calculated using appropriate quantification software from aligned BAM files not from read counts since often only a portion of the entire gene/transcript is sequenced, not the entire length. If in doubt check for a relationship between

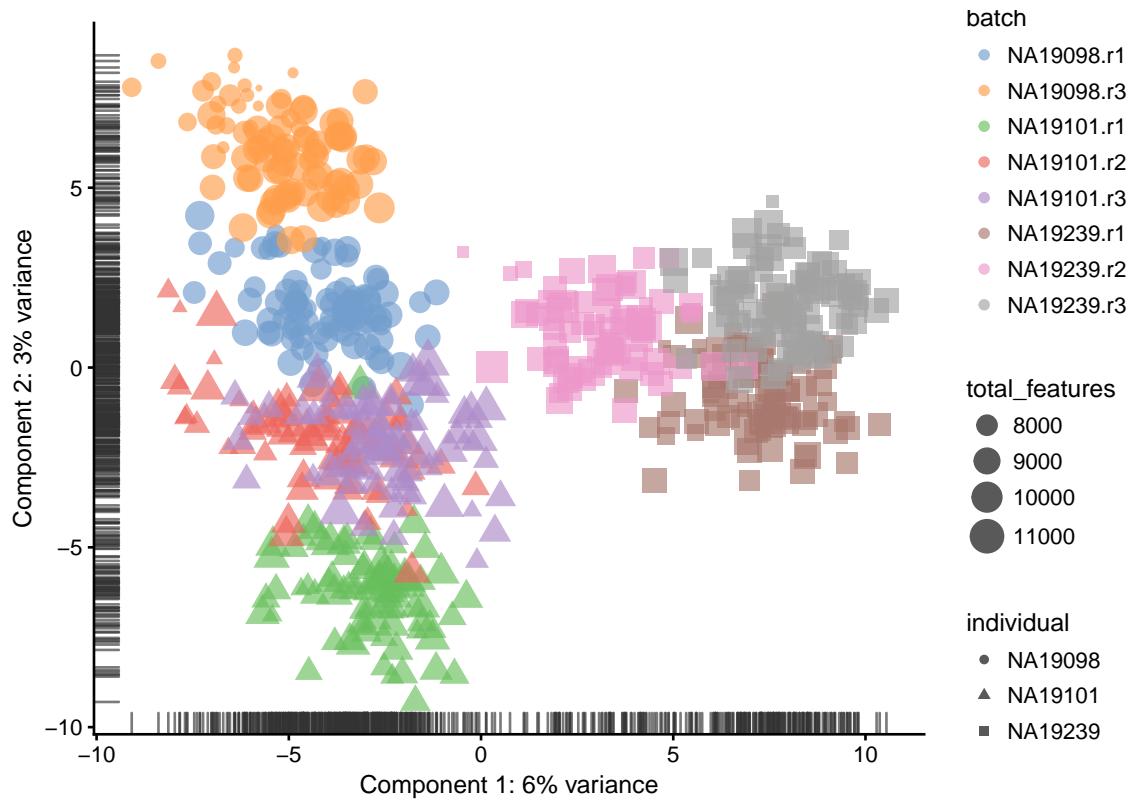


Figure 7.49: PCA plot of the tung data after downsampling

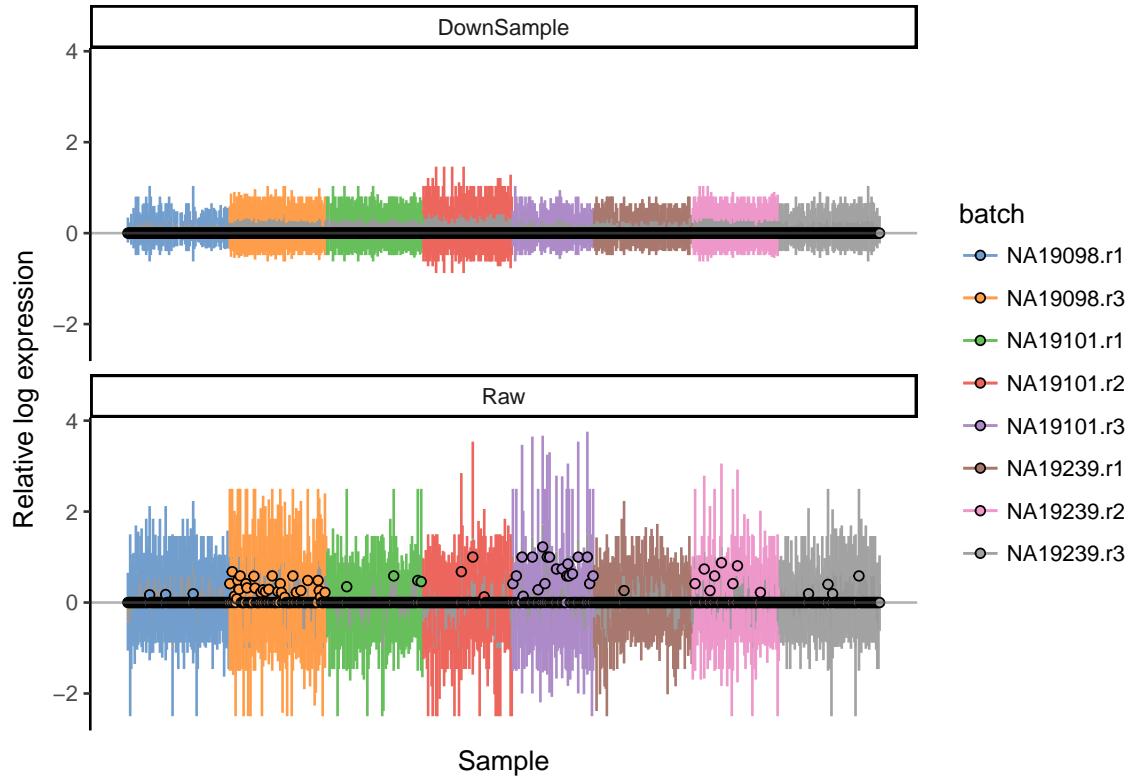


Figure 7.50: Cell-wise RLE of the tung data

gene/transcript length and expression level.

However, here we show how these normalisations can be calculated using `scater`. First, we need to find the effective transcript length in Kilobases. However, our dataset contains only gene IDs, therefore we will be using the gene lengths instead of transcripts. `scater` uses the `biomaRt` package, which allows one to annotate genes by other attributes:

```
umi.qc <- getBMFeatureAnnos(
  umi.qc,
  filters = "ensembl_gene_id",
  attributes = c(
    "ensembl_gene_id",
    "hgnc_symbol",
    "chromosome_name",
    "start_position",
    "end_position"
  ),
  feature_symbol = "hgnc_symbol",
  feature_id = "ensembl_gene_id",
  biomart = "ENSEMBL_MART_ENSEMBL",
  dataset = "hsapiens_gene_ensembl",
  host = "www.ensembl.org"
)

# If you have mouse data, change the arguments based on this example:
# getBMFeatureAnnos(
#   object,
#   filters = "ensembl_transcript_id",
#   attributes = c(
#     "ensembl_transcript_id",
#     "ensembl_gene_id",
#     "mgi_symbol",
#     "chromosome_name",
#     "transcript_biotype",
#     "transcript_start",
#     "transcript_end",
#     "transcript_count"
#   ),
#   feature_symbol = "mgi_symbol",
#   feature_id = "ensembl_gene_id",
#   biomart = "ENSEMBL_MART_ENSEMBL",
#   dataset = "mmusculus_gene_ensembl",
#   host = "www.ensembl.org"
# )
```

Some of the genes were not annotated, therefore we filter them out:

```
umi.qc.ann <- umi.qc[!is.na(rowData(umi.qc)$ensembl_gene_id), ]
```

Now we compute the total gene length in Kilobases by using the `end_position` and `start_position` fields:

```
eff_length <-
  abs(rowData(umi.qc.ann)$end_position - rowData(umi.qc.ann)$start_position) / 1000

plot(eff_length, rowMeans(counts(umi.qc.ann)))
```

There is no relationship between gene length and mean expression so \_\_\_\_FPKM\_\_\_\_s & \_\_\_\_TPM\_\_\_\_s are

inappropriate for this dataset. But we will demonstrate them anyway.

**Note** Here calculate the total gene length instead of the total exon length. Many genes will contain lots of introns so their `eff_length` will be very different from what we have calculated. Please consider our calculation as approximation. If you want to use the total exon lengths, please refer to this page.

Now we are ready to perform the normalisations:

```
tpm(umi.qc.ann) <- log2(calculateTPM(umi.qc.ann, eff_length) + 1)
```

Plot the results as a PCA plot:

```
plotPCA(
  umi.qc.ann,
  exprs_values = "tpm",
  colour_by = "batch",
  size_by = "total_features",
  shape_by = "individual"
)

tpm(umi.qc.ann) <- log2(calculateFPKM(umi.qc.ann, eff_length) + 1)

plotPCA(
  umi.qc.ann,
  exprs_values = "tpm",
  colour_by = "batch",
  size_by = "total_features",
  shape_by = "individual"
)
```

**Note** The PCA looks for differences between cells. Gene length is the same across cells for each gene thus **FPKM** is almost identical to the **CPM** plot (it is just rotated) since it performs **CPM** first then normalizes gene length. Whereas, **TPM** is different because it weights genes by their length before performing **CPM**.

### 7.8.9 Exercise

Perform the same analysis with read counts of the `tung` data. Use `tung/reads.rds` file to load the reads SCE object. Once you have finished please compare your results to ours (next chapter).

#### 7.8.10 sessionInfo()

```
## R version 3.4.3 (2017-11-30)
## Platform: x86_64-pc-linux-gnu (64-bit)
## Running under: Debian GNU/Linux 9 (stretch)
##
## Matrix products: default
## BLAS: /usr/lib/openblas-base/libblas.so.3
## LAPACK: /usr/lib/libopenblas-r0.2.19.so
##
## locale:
## [1] LC_CTYPE=en_US.UTF-8      LC_NUMERIC=C
## [3] LC_TIME=en_US.UTF-8       LC_COLLATE=en_US.UTF-8
## [5] LC_MONETARY=en_US.UTF-8   LC_MESSAGES=C
## [7] LC_PAPER=en_US.UTF-8      LC_NAME=C
## [9] LC_ADDRESS=C              LC_TELEPHONE=C
```

```

## [11] LC_MEASUREMENT=en_US.UTF-8 LC_IDENTIFICATION=C
##
## attached base packages:
## [1] stats4     parallel   methods    stats      graphics   grDevices utils
## [8] datasets   base
##
## other attached packages:
## [1] scran_1.6.7           BiocParallel_1.12.0
## [3] scater_1.6.2          SingleCellExperiment_1.0.0
## [5] SummarizedExperiment_1.8.1 DelayedArray_0.4.1
## [7] matrixStats_0.53.0     GenomicRanges_1.30.1
## [9] GenomeInfoDb_1.14.0    IRanges_2.12.0
## [11] S4Vectors_0.16.0      ggplot2_2.2.1
## [13] Biobase_2.38.0        BiocGenerics_0.24.0
## [15] knitr_1.19            scRNA.seq.funcs_0.1.0
##
## loaded via a namespace (and not attached):
## [1] bitops_1.0-6           bit64_0.9-7           progress_1.1.2
## [4] htr_1.3.1              rprojroot_1.3-2       dynamicTreeCut_1.63-1
## [7] tools_3.4.3             backports_1.1.2       DT_0.4
## [10] R6_2.2.2               hypergeo_1.2-13       vigor_0.4.5
## [13] DBI_0.7                lazyeval_0.2.1        colorspace_1.3-2
## [16] gridExtra_2.3           prettyunits_1.0.2     moments_0.14
## [19] bit_1.1-12              compiler_3.4.3       orthopolynom_1.0-5
## [22] labeling_0.3            bookdown_0.6          scales_0.5.0
## [25] stringr_1.2.0           digest_0.6.15        rmarkdown_1.8
## [28] XVector_0.18.0          pkgconfig_2.0.1      htmltools_0.3.6
## [31] limma_3.34.8            htmlwidgets_1.0       rlang_0.1.6
## [34] RSQLite_2.0              FNN_1.1                 shiny_1.0.5
## [37] bindr_0.1                zoo_1.8-1              dplyr_0.7.4
## [40] RCurl_1.95-4.10         magrittr_1.5          GenomeInfoDbData_1.0.0
## [43] Matrix_1.2-7.1          Rcpp_0.12.15          ggbeeswarm_0.6.0
## [46] munsell_0.4.3            viridis_0.5.0         stringi_1.1.6
## [49] yaml_2.1.16              edgeR_3.20.8          MASS_7.3-45
## [52] zlibbioc_1.24.0          rhdf5_2.22.0          Rtsne_0.13
## [55] plyr_1.8.4               grid_3.4.3            blob_1.1.0
## [58] shinydashboard_0.6.1     conffrac_1.1-11       lattice_0.20-34
## [61] cowplot_0.9.2            locfit_1.5-9.1        pillar_1.1.0
## [64] igraph_1.1.2              rjson_0.2.15          reshape2_1.4.3
## [67] biomaRt_2.34.2            XML_3.98-1.9          glue_1.2.0
## [70] evaluate_0.10.1           data.table_1.10.4-3  deSolve_1.20
## [73] httpuv_1.3.5              gtable_0.2.0          assertthat_0.2.0
## [76] xfun_0.1                  mime_0.5              xtable_1.8-2
## [79] viridisLite_0.3.0          tibble_1.4.2           elliptic_1.3-7
## [82] AnnotationDbi_1.40.0       beeswarm_0.2.3        memoise_1.1.0
## [85] tximport_1.6.0              bindrcpp_0.2          statmod_1.4.30

```

## 7.9 Normalization practice (Reads)

```

## Warning in normalizeSCE(object, exprs_values = exprs_values, return_log
## = return_log, : spike-in transcripts in 'ERCC' should have their own size
## factors

```

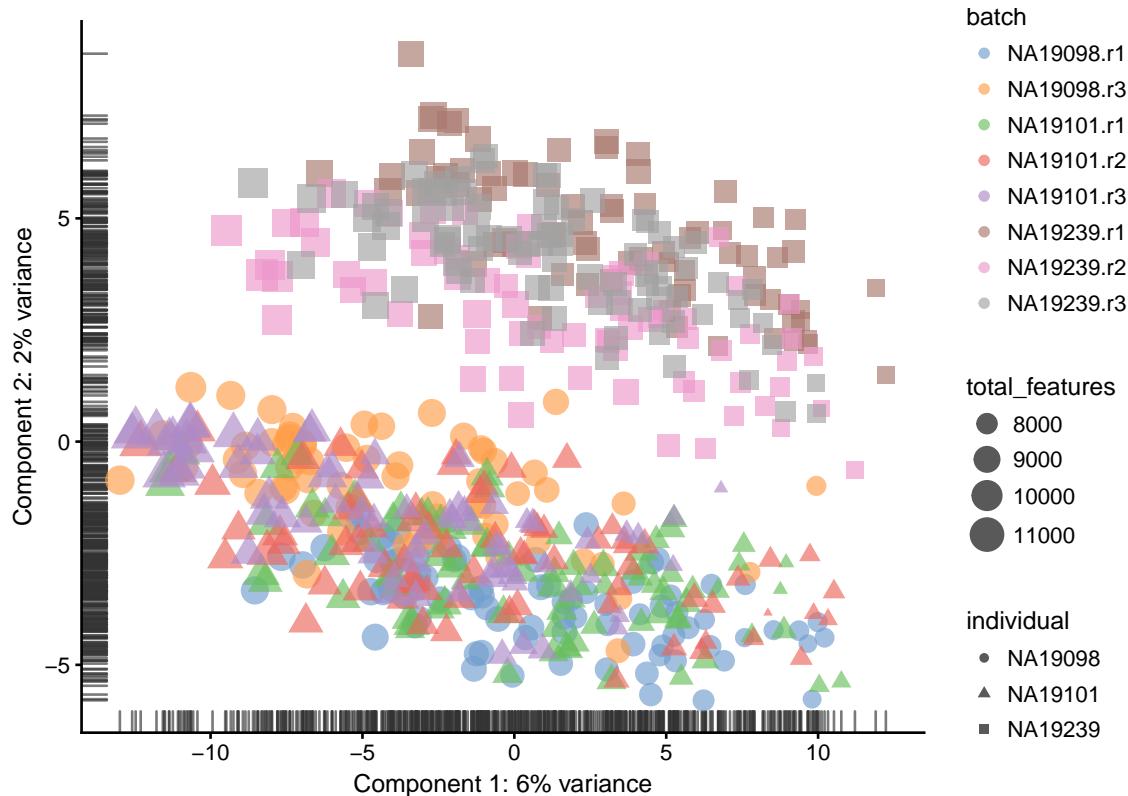


Figure 7.51: PCA plot of the tung data

```

## Warning in normalizeSCE(object, exprs_values = exprs_values, return_log
## = return_log, : spike-in transcripts in 'ERCC' should have their own size
## factors

## Warning in normalizeSCE(object, exprs_values = exprs_values, return_log
## = return_log, : spike-in transcripts in 'ERCC' should have their own size
## factors

## Warning in .local(object, ...): spike-in transcripts in 'ERCC' should have
## their own size factors

## R version 3.4.3 (2017-11-30)
## Platform: x86_64-pc-linux-gnu (64-bit)
## Running under: Debian GNU/Linux 9 (stretch)
##
## Matrix products: default
## BLAS: /usr/lib/openblas-base/libblas.so.3
## LAPACK: /usr/lib/libopenblas-p-r0.2.19.so
##
## locale:
## [1] LC_CTYPE=en_US.UTF-8          LC_NUMERIC=C
## [3] LC_TIME=en_US.UTF-8          LC_COLLATE=en_US.UTF-8
## [5] LC_MONETARY=en_US.UTF-8       LC_MESSAGES=C
## [7] LC_PAPER=en_US.UTF-8          LC_NAME=C
## [9] LC_ADDRESS=C                  LC_TELEPHONE=C
## [11] LC_MEASUREMENT=en_US.UTF-8    LC_IDENTIFICATION=C
##

```

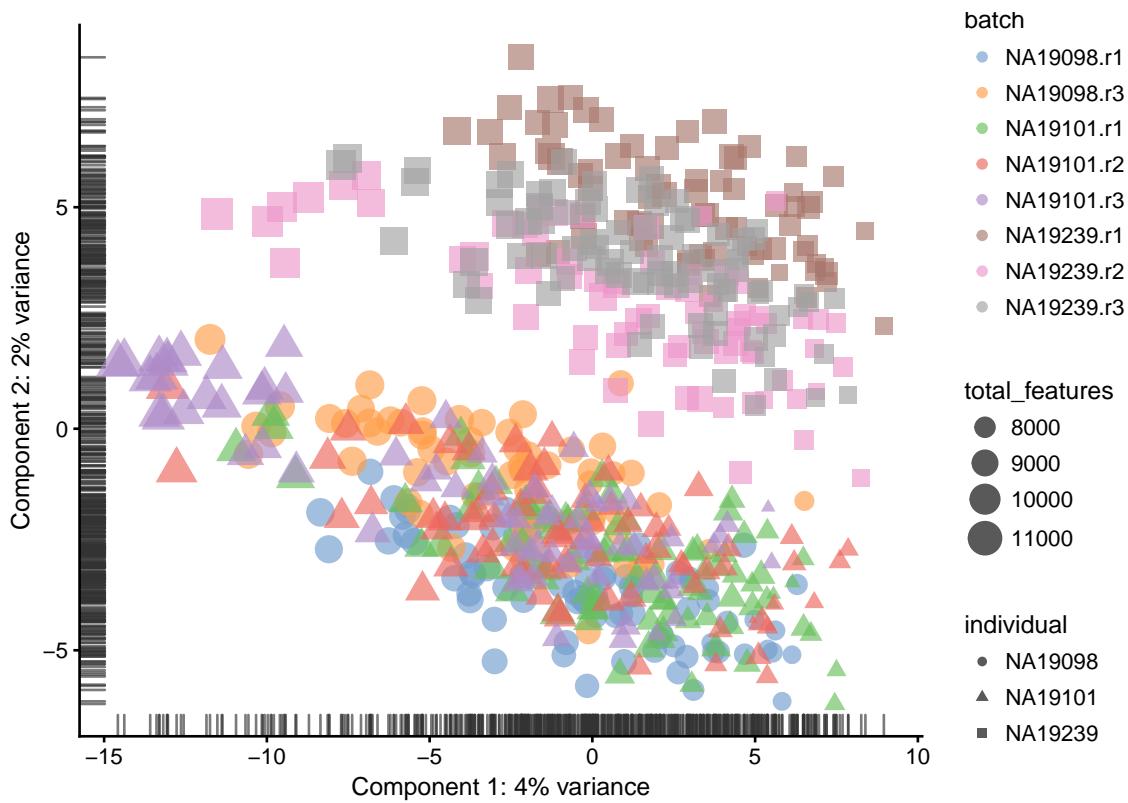


Figure 7.52: PCA plot of the tung data after CPM normalisation

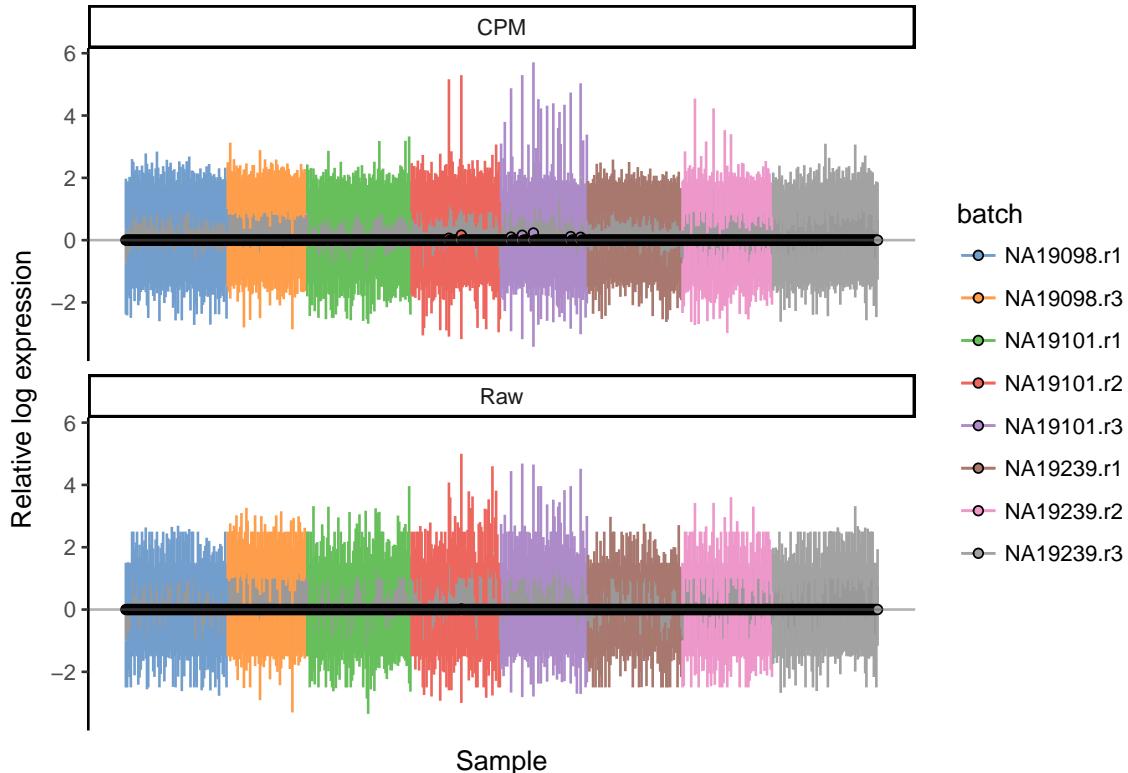


Figure 7.53: Cell-wise RLE of the tung data

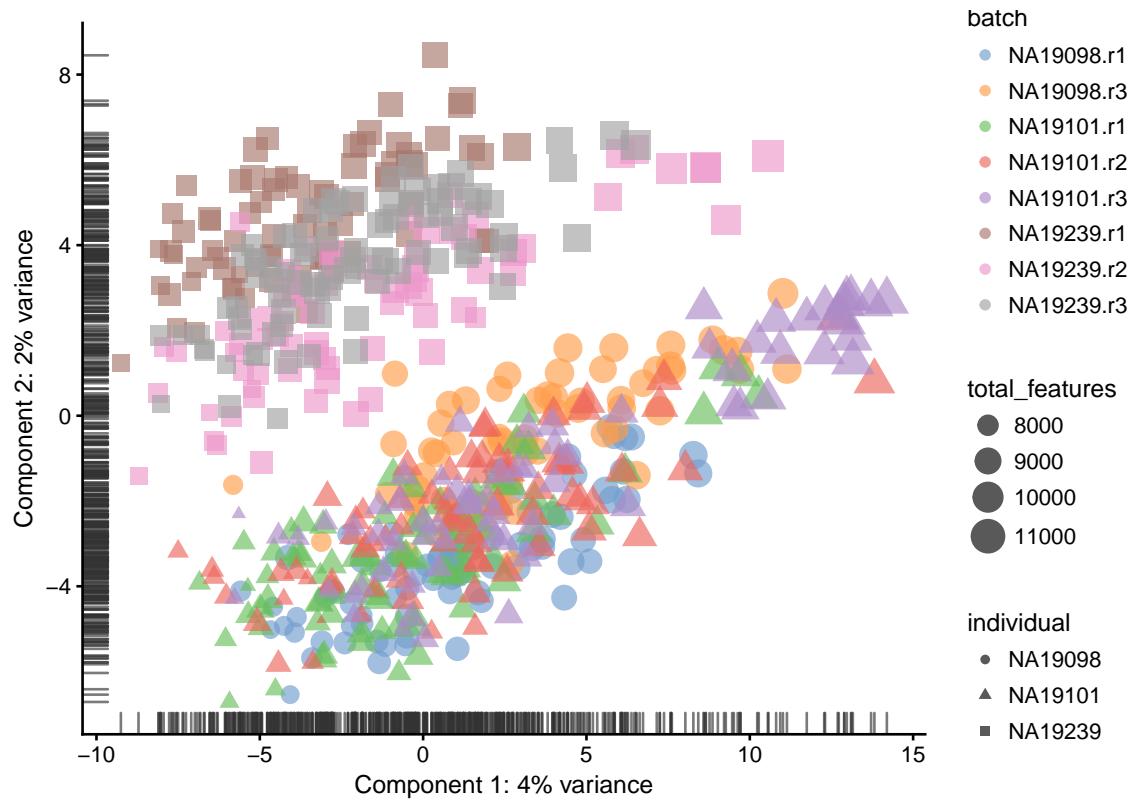


Figure 7.54: PCA plot of the tung data after RLE normalisation

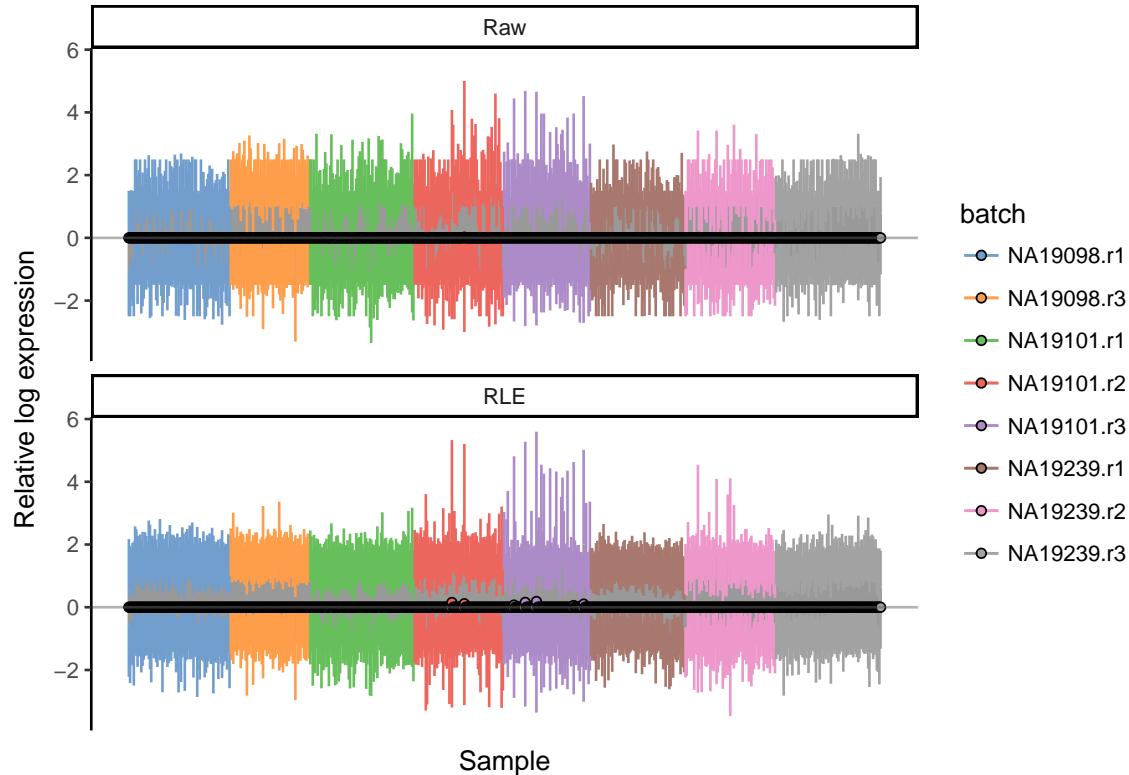


Figure 7.55: Cell-wise RLE of the tung data

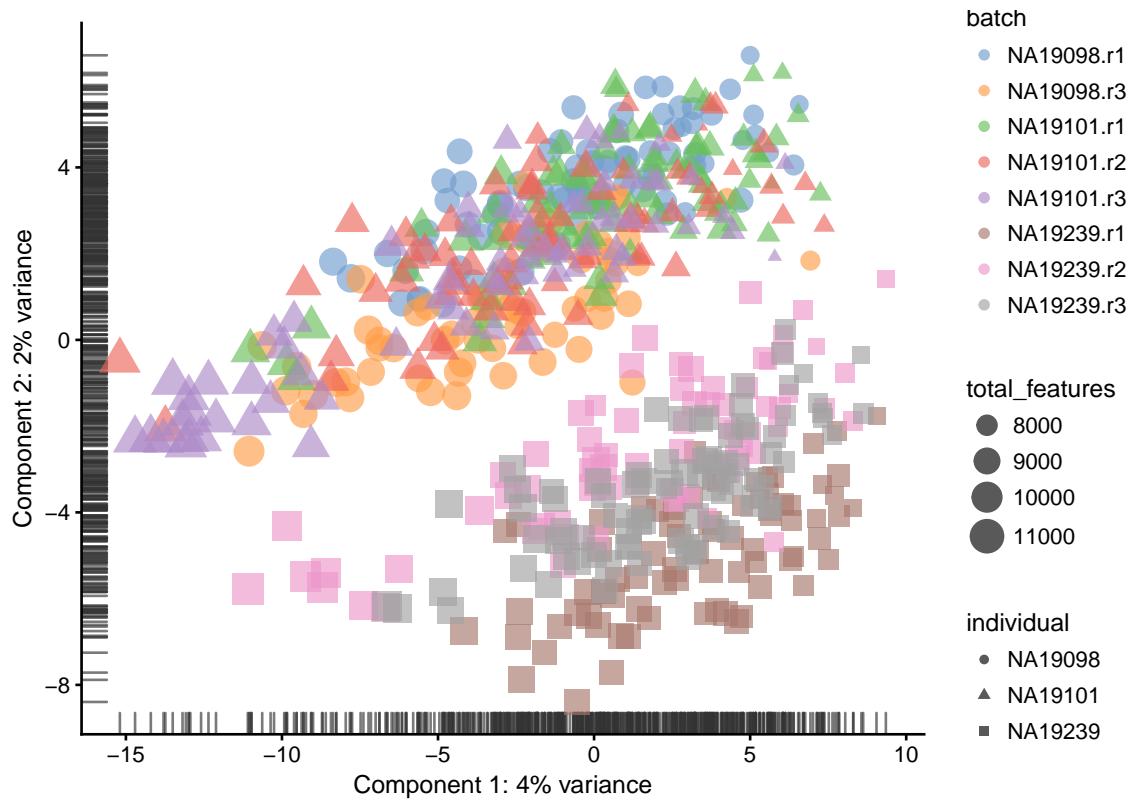


Figure 7.56: PCA plot of the tung data after UQ normalisation

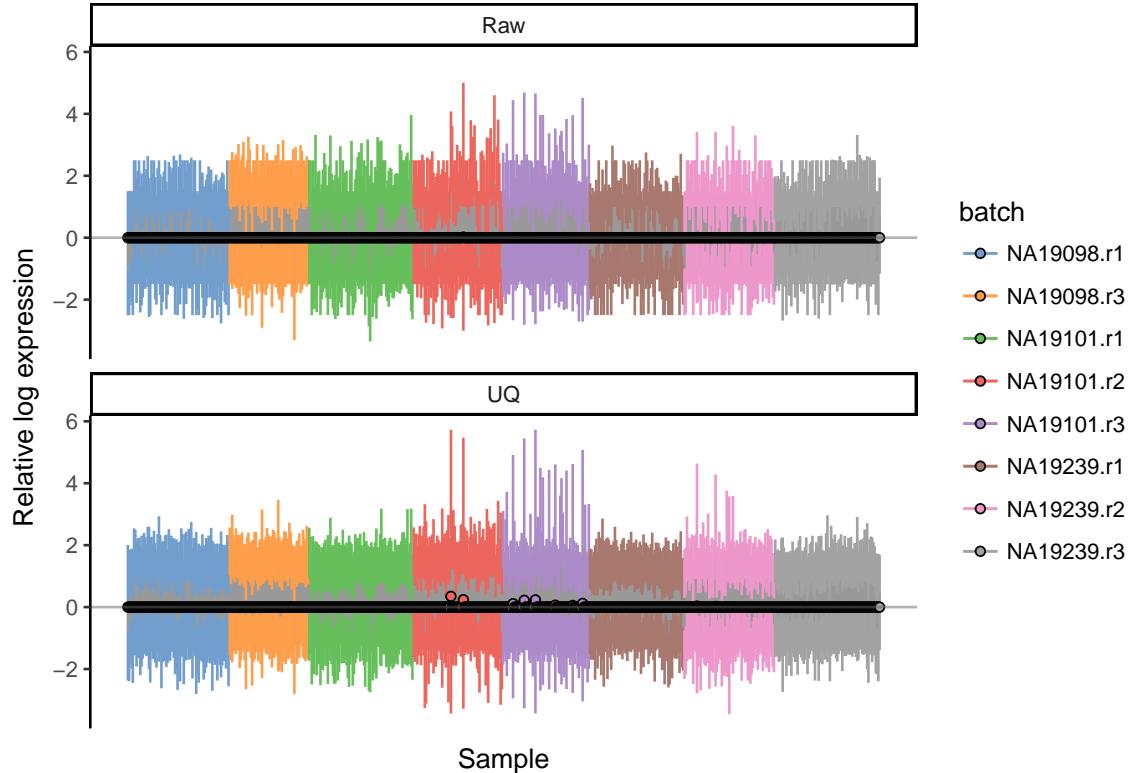


Figure 7.57: Cell-wise RLE of the tung data

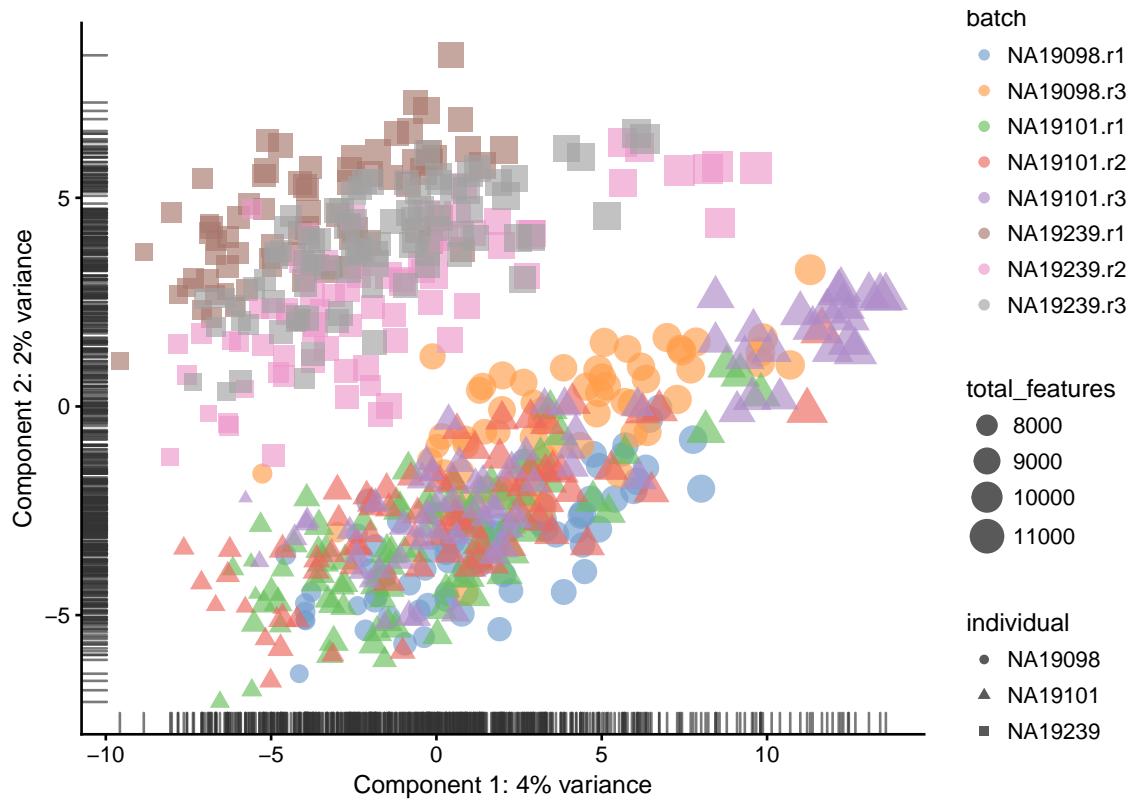


Figure 7.58: PCA plot of the tung data after TMM normalisation

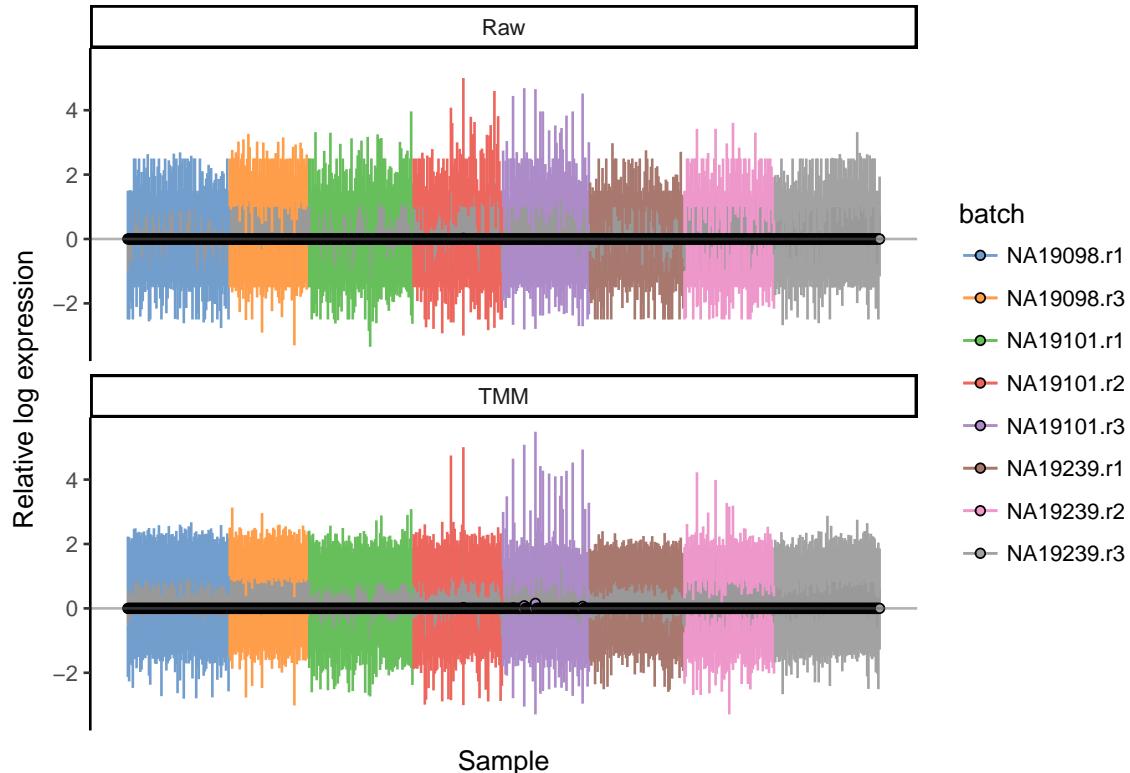


Figure 7.59: Cell-wise RLE of the tung data

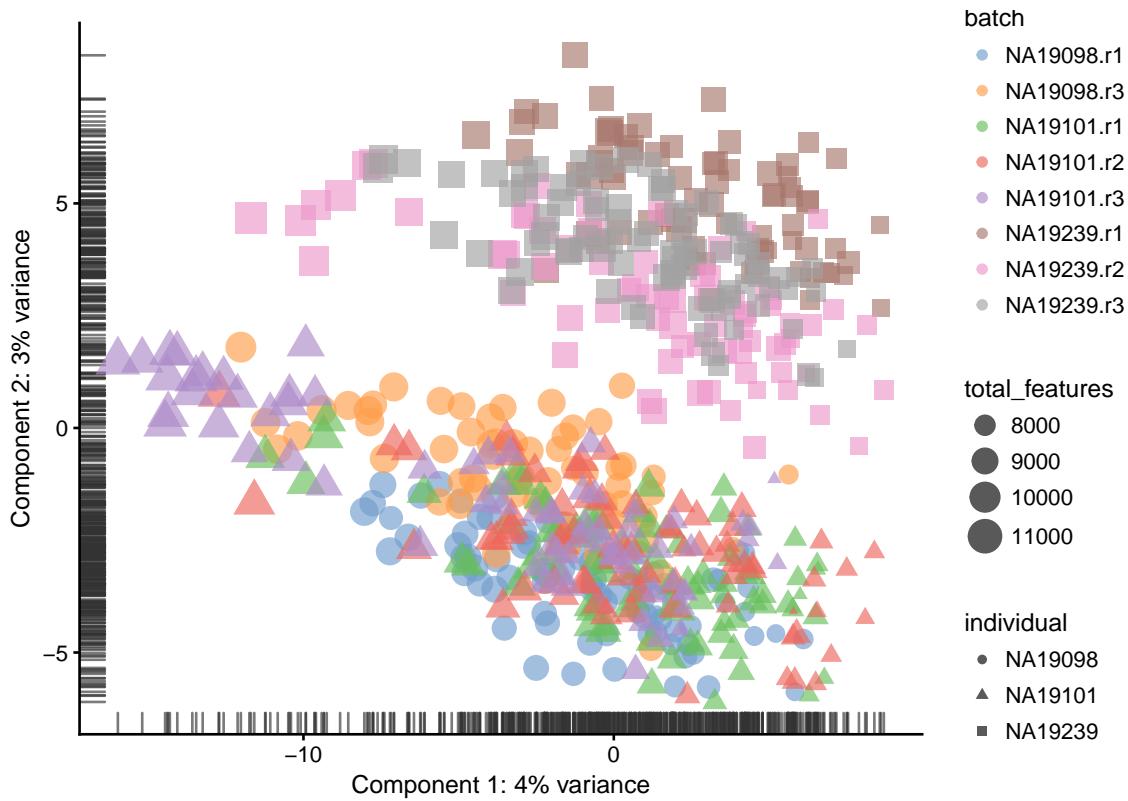


Figure 7.60: PCA plot of the tung data after LSF normalisation

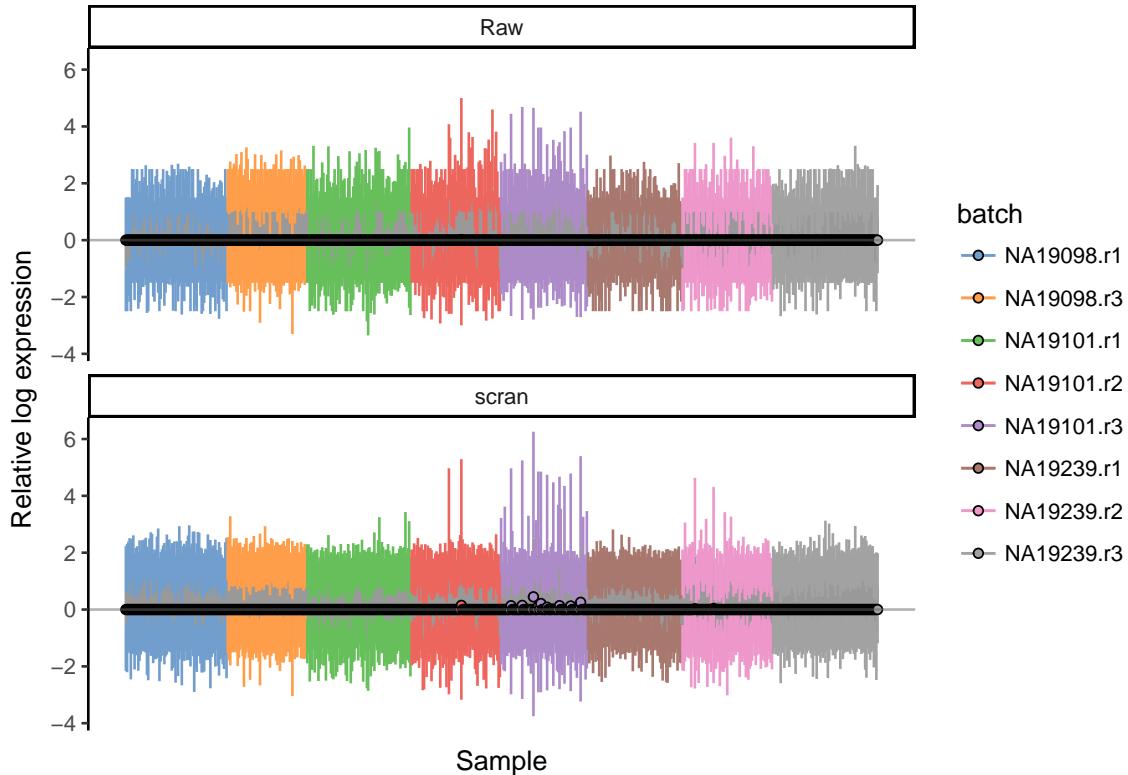


Figure 7.61: Cell-wise RLE of the tung data

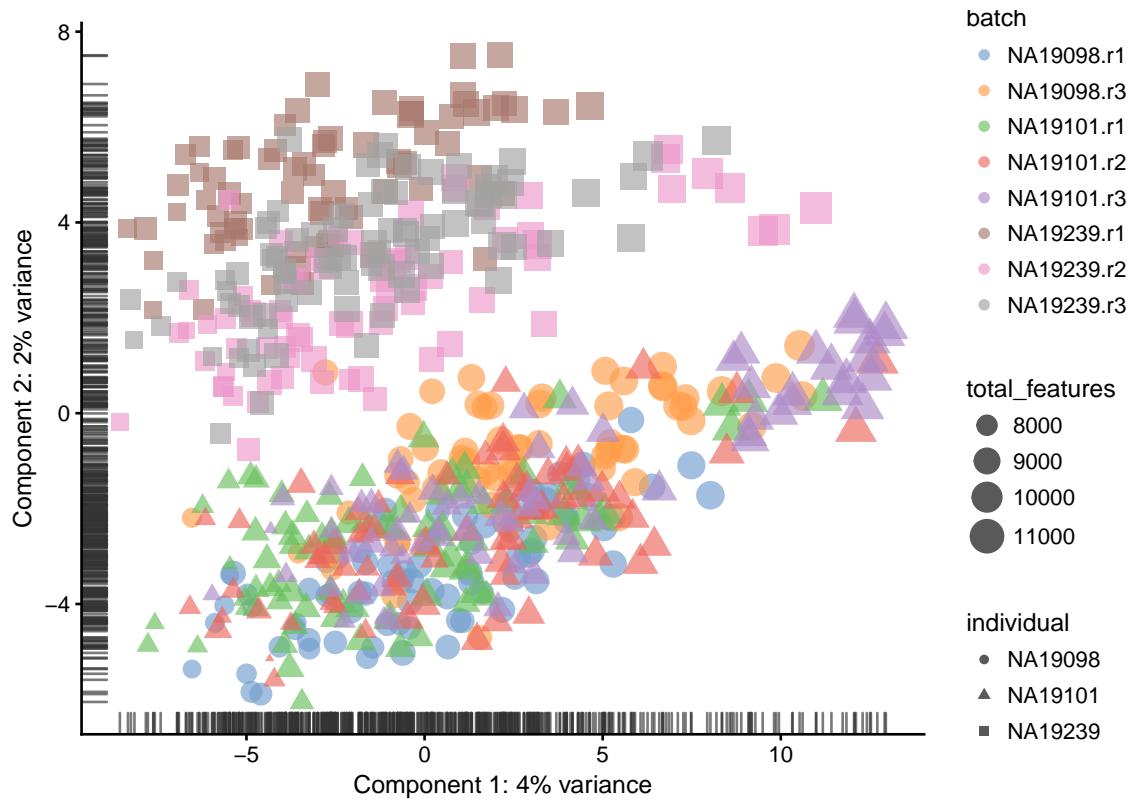


Figure 7.62: PCA plot of the tung data after downsampling

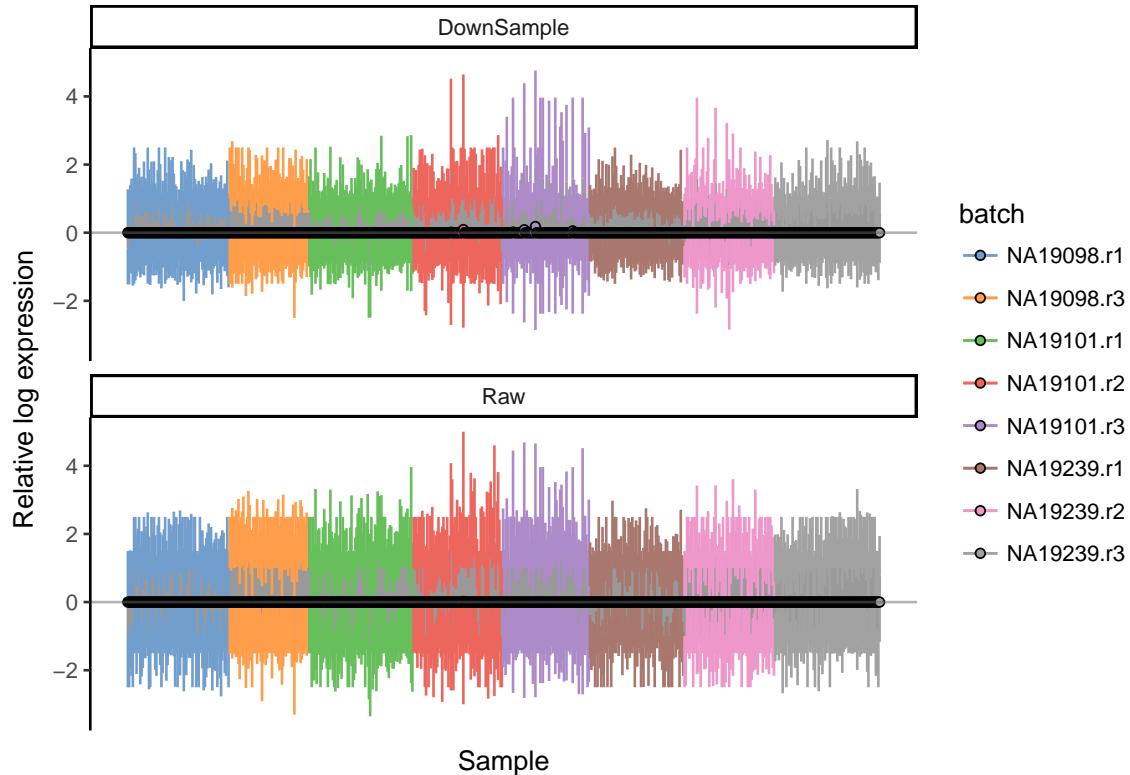


Figure 7.63: Cell-wise RLE of the tung data

```

## attached base packages:
## [1] stats4    parallel  methods   stats      graphics  grDevices utils
## [8] datasets  base

##
## other attached packages:
## [1] knitr_1.19           scran_1.6.7
## [3] BiocParallel_1.12.0  scater_1.6.2
## [5] SingleCellExperiment_1.0.0 SummarizedExperiment_1.8.1
## [7] DelayedArray_0.4.1   matrixStats_0.53.0
## [9] GenomicRanges_1.30.1  GenomeInfoDb_1.14.0
## [11] IRanges_2.12.0       S4Vectors_0.16.0
## [13] ggplot2_2.2.1        Biobase_2.38.0
## [15] BiocGenerics_0.24.0  scRNA.seq.funcs_0.1.0
##
## loaded via a namespace (and not attached):
## [1] bitops_1.0-6          bit64_0.9-7          progress_1.1.2
## [4] httr_1.3.1            rprojroot_1.3-2     dynamicTreeCut_1.63-1
## [7] tools_3.4.3           backports_1.1.2     DT_0.4
## [10] R6_2.2.2              hypergeo_1.2-13     viper_0.4.5
## [13] DBI_0.7               lazyeval_0.2.1      colorspace_1.3-2
## [16] gridExtra_2.3         prettyunits_1.0.2   moments_0.14
## [19] bit_1.1-12            compiler_3.4.3     orthopolynom_1.0-5
## [22] labeling_0.3          bookdown_0.6        scales_0.5.0
## [25] stringr_1.2.0         digest_0.6.15      rmarkdown_1.8
## [28] XVector_0.18.0        pkgconfig_2.0.1    htmltools_0.3.6
## [31] limma_3.34.8          htmlwidgets_1.0    rlang_0.1.6
## [34] RSQLite_2.0            FNN_1.1             shiny_1.0.5
## [37] bindr_0.1              zoo_1.8-1          dplyr_0.7.4
## [40] RCurl_1.95-4.10       magrittr_1.5       GenomeInfoDbData_1.0.0
## [43] Matrix_1.2-7.1        Rcpp_0.12.15       ggbeeswarm_0.6.0
## [46] munsell_0.4.3          viridis_0.5.0     stringi_1.1.6
## [49] yaml_2.1.16            edgeR_3.20.8      MASS_7.3-45
## [52] zlibbioc_1.24.0        rhdf5_2.22.0      Rtsne_0.13
## [55] plyr_1.8.4              grid_3.4.3         blob_1.1.0
## [58] shinydashboard_0.6.1   conffrac_1.1-11   lattice_0.20-34
## [61] cowplot_0.9.2          locfit_1.5-9.1    pillar_1.1.0
## [64] igraph_1.1.2            rjson_0.2.15      reshape2_1.4.3
## [67] biomaRt_2.34.2          XML_3.98-1.9      glue_1.2.0
## [70] evaluate_0.10.1        data.table_1.10.4-3 deSolve_1.20
## [73] httpuv_1.3.5            gtable_0.2.0      assertthat_0.2.0
## [76] xfun_0.1                mime_0.5           xtable_1.8-2
## [79] viridisLite_0.3.0        tibble_1.4.2      elliptic_1.3-7
## [82] AnnotationDbi_1.40.0    beeswarm_0.2.3   memoise_1.1.0
## [85] tximport_1.6.0           bindrcpp_0.2      statmod_1.4.30

```

## 7.10 Dealing with confounders

### 7.10.1 Introduction

In the previous chapter we normalized for library size, effectively removing it as a confounder. Now we will consider removing other less well defined confounders from our data. Technical confounders (aka batch effects) can arise from difference in reagents, isolation methods, the lab/experimenter who performed the

experiment, even which day/time the experiment was performed. Accounting for technical confounders, and batch effects particularly, is a large topic that also involves principles of experimental design. Here we address approaches that can be taken to account for confounders when the experimental design is appropriate.

Fundamentally, accounting for technical confounders involves identifying and, ideally, removing sources of variation in the expression data that are not related to (i.e. are confounding) the biological signal of interest. Various approaches exist, some of which use spike-in or housekeeping genes, and some of which use endogenous genes.

### 7.10.1.1 Advantages and disadvantages of using spike-ins to remove confounders

The use of spike-ins as control genes is appealing, since the same amount of ERCC (or other) spike-in was added to each cell in our experiment. In principle, all the variability we observe for these genes is due to technical noise; whereas endogenous genes are affected by both technical noise and biological variability. Technical noise can be removed by fitting a model to the spike-ins and “subtracting” this from the endogenous genes. There are several methods available based on this premise (eg. BASiCS, scLVM, RUVg); each using different noise models and different fitting procedures. Alternatively, one can identify genes which exhibit significant variation beyond technical noise (eg. Distance to median, Highly variable genes). However, there are issues with the use of spike-ins for normalisation (particularly ERCCs, derived from bacterial sequences), including that their variability can, for various reasons, actually be *higher* than that of endogenous genes.

Given the issues with using spike-ins, better results can often be obtained by using endogenous genes instead. Where we have a large number of endogenous genes that, on average, do not vary systematically between cells and where we expect technical effects to affect a large number of genes (a very common and reasonable assumption), then such methods (for example, the RUVs method) can perform well.

We explore both general approaches below.

```
library(scRNA.seq.funcs)
library(RUVSeq)
library(scater)
library(SingleCellExperiment)
library(scran)
library(kBET)
library(sva) # Combat
library(edgeR)
set.seed(1234567)
options(stringsAsFactors = FALSE)
umi <- readRDS("tung/umi.rds")
umi.qc <- umi$rowData(umi)$use, colData(umi)$use]
endog_genes <- !rowData(umi.qc)$is_feature_control
erccs <- rowData(umi.qc)$is_feature_control

qclust <- quickCluster(umi.qc, min.size = 30)
umi.qc <- computeSumFactors(umi.qc, sizes = 15, clusters = qclust)
umi.qc <- normalize(umi.qc)
```

### 7.10.2 Remove Unwanted Variation

Factors contributing to technical noise frequently appear as “batch effects” where cells processed on different days or by different technicians systematically vary from one another. Removing technical noise and correcting for batch effects can frequently be performed using the same tool or slight variants on it. We will be considering the Remove Unwanted Variation (RUVSeq). Briefly, RUVSeq works as follows. For  $n$  samples

and  $J$  genes, consider the following generalized linear model (GLM), where the RNA-Seq read counts are regressed on both the known covariates of interest and unknown factors of unwanted variation:

$$\log E[Y|W, X, O] = W\alpha + X\beta + O$$

Here,  $Y$  is the  $n \times J$  matrix of observed gene-level read counts,  $W$  is an  $n \times k$  matrix corresponding to the factors of “unwanted variation” and  $O$  is an  $n \times J$  matrix of offsets that can either be set to zero or estimated with some other normalization procedure (such as upper-quartile normalization). The simultaneous estimation of  $W$ ,  $\alpha$ ,  $\beta$ , and  $k$  is infeasible. For a given  $k$ , instead the following three approaches to estimate the factors of unwanted variation  $W$  are used:

- $RUVg$  uses negative control genes (e.g. ERCCs), assumed to have constant expression across samples;
- $RUVs$  uses centered (technical) replicate/negative control samples for which the covariates of interest are constant;
- $RUVr$  uses residuals, e.g., from a first-pass GLM regression of the counts on the covariates of interest.

We will concentrate on the first two approaches.

### 7.10.2.1 RUVg

```
ruvg <- RUVg(counts(umi.qc), erccs, k = 1)
assay(umi.qc, "ruvg1") <- log2(
  t(t(ruvg$normalizedCounts) / colSums(ruvg$normalizedCounts) * 1e6) + 1
)
ruvg <- RUVg(counts(umi.qc), erccs, k = 10)
assay(umi.qc, "ruvg10") <- log2(
  t(t(ruvg$normalizedCounts) / colSums(ruvg$normalizedCounts) * 1e6) + 1
)
```

### 7.10.2.2 RUVs

```
scIdx <- matrix(-1, ncol = max(table(umi.qc$individual)), nrow = 3)
tmp <- which(umi.qc$individual == "NA19098")
scIdx[1, 1:length(tmp)] <- tmp
tmp <- which(umi.qc$individual == "NA19101")
scIdx[2, 1:length(tmp)] <- tmp
tmp <- which(umi.qc$individual == "NA19239")
scIdx[3, 1:length(tmp)] <- tmp
cIdx <- rownames(umi.qc)
ruvs <- RUVs(counts(umi.qc), cIdx, k = 1, scIdx = scIdx, isLog = FALSE)
assay(umi.qc, "ruvs1") <- log2(
  t(t(ruvs$normalizedCounts) / colSums(ruvs$normalizedCounts) * 1e6) + 1
)
ruvs <- RUVs(counts(umi.qc), cIdx, k = 10, scIdx = scIdx, isLog = FALSE)
assay(umi.qc, "ruvs10") <- log2(
  t(t(ruvs$normalizedCounts) / colSums(ruvs$normalizedCounts) * 1e6) + 1
)
```

### 7.10.3 Combat

If you have an experiment with a balanced design, `Combat` can be used to eliminate batch effects while preserving biological effects by specifying the `mod` parameter. However the Tung

data contains multiple experimental replicates rather than a balanced design so using `mod1` to preserve biological variability will result in an error.

```
combat_data <- logcounts(umi.qc)
mod_data <- as.data.frame(t(combat_data))
# Basic batch removal
mod0 = model.matrix(~ 1, data = mod_data)
# Preserve biological variability
mod1 = model.matrix(~ umi.qc$individual, data = mod_data)
# adjust for total genes detected
mod2 = model.matrix(~ umi.qc$total_features, data = mod_data)
assay(umi.qc, "combat") <- ComBat(
  dat = t(mod_data),
  batch = factor(umi.qc$batch),
  mod = mod0,
  par.prior = TRUE,
  prior.plots = FALSE
)
```

```
## Standardizing Data across genes
```

### Exercise 1

Perform ComBat correction accounting for total features as a co-variate. Store the corrected matrix in the `combat_tf` slot.

```
## Standardizing Data across genes
```

#### 7.10.4 mnnCorrect

`mnnCorrect` (Haghverdi et al., 2017) assumes that each batch shares at least one biological condition with each other batch. Thus it works well for a variety of balanced experimental designs. However, the Tung data contains multiple replicates for each individual rather than balanced batches, thus we will normalize each individual separately. Note that this will remove batch effects between batches within the same individual but not the batch effects between batches in different individuals, due to the confounded experimental design.

Thus we will merge a replicate from each individual to form three batches.

```
do_mnn <- function(data.qc) {
  batch1 <- logcounts(data.qc[, data.qc$replicate == "r1"])
  batch2 <- logcounts(data.qc[, data.qc$replicate == "r2"])
  batch3 <- logcounts(data.qc[, data.qc$replicate == "r3"])

  if (ncol(batch2) > 0) {
    x = mnnCorrect(
      batch1, batch2, batch3,
      k = 20,
      sigma = 0.1,
      cos.norm.in = TRUE,
      svd.dim = 2
    )
    res1 <- x$corrected[[1]]
    res2 <- x$corrected[[2]]
    res3 <- x$corrected[[3]]
    dimnames(res1) <- dimnames(batch1)
    dimnames(res2) <- dimnames(batch2)
  }
}
```

```

    dimnames(res3) <- dimnames(batch3)
    return(cbind(res1, res2, res3))
} else {
  x = mnnCorrect(
    batch1, batch3,
    k = 20,
    sigma = 0.1,
    cos.norm.in = TRUE,
    svd.dim = 2
  )
  res1 <- x$corrected[[1]]
  res3 <- x$corrected[[2]]
  dimnames(res1) <- dimnames(batch1)
  dimnames(res3) <- dimnames(batch3)
  return(cbind(res1, res3))
}
}

indi1 <- do_mnn(umi.qc[, umi.qc$individual == "NA19098"])
indi2 <- do_mnn(umi.qc[, umi.qc$individual == "NA19101"])
indi3 <- do_mnn(umi.qc[, umi.qc$individual == "NA19239"])

assay(umi.qc, "mnn") <- cbind(indi1, indi2, indi3)

# For a balanced design:
#assay(umi.qc, "mnn") <- mnnCorrect(
#  list(B1 = logcounts(batch1), B2 = logcounts(batch2), B3 = logcounts(batch3)),
#  k = 20,
#  sigma = 0.1,
#  cos.norm = TRUE,
#  svd.dim = 2
#)

```

### 7.10.5 GLM

A general linear model is a simpler version of `Combat`. It can correct for batches while preserving biological effects if you have a balanced design. In a confounded/replicate design biological effects will not be fit/preserved. Similar to `mnnCorrect` we could remove batch effects from each individual separately in order to preserve biological (and technical) variance between individuals. For demonstration purposes we will naively correct all cofounded batch effects:

```

glm_fun <- function(g, batch, indi) {
  model <- glm(g ~ batch + indi)
  model$coef[1] <- 0 # replace intercept with 0 to preserve reference batch.
  return(model$coef)
}
effects <- apply(
  logcounts(umi.qc),
  1,
  glm_fun,
  batch = umi.qc$batch,
  indi = umi.qc$individual
)

```

```
corrected <- logcounts(umi.qc) - t(effects[as.numeric(factor(umi.qc$batch)), ])
assay(umi.qc, "glm") <- corrected
```

### Exercise 2

Perform GLM correction for each individual separately. Store the final corrected matrix in the `glm_indi` slot.

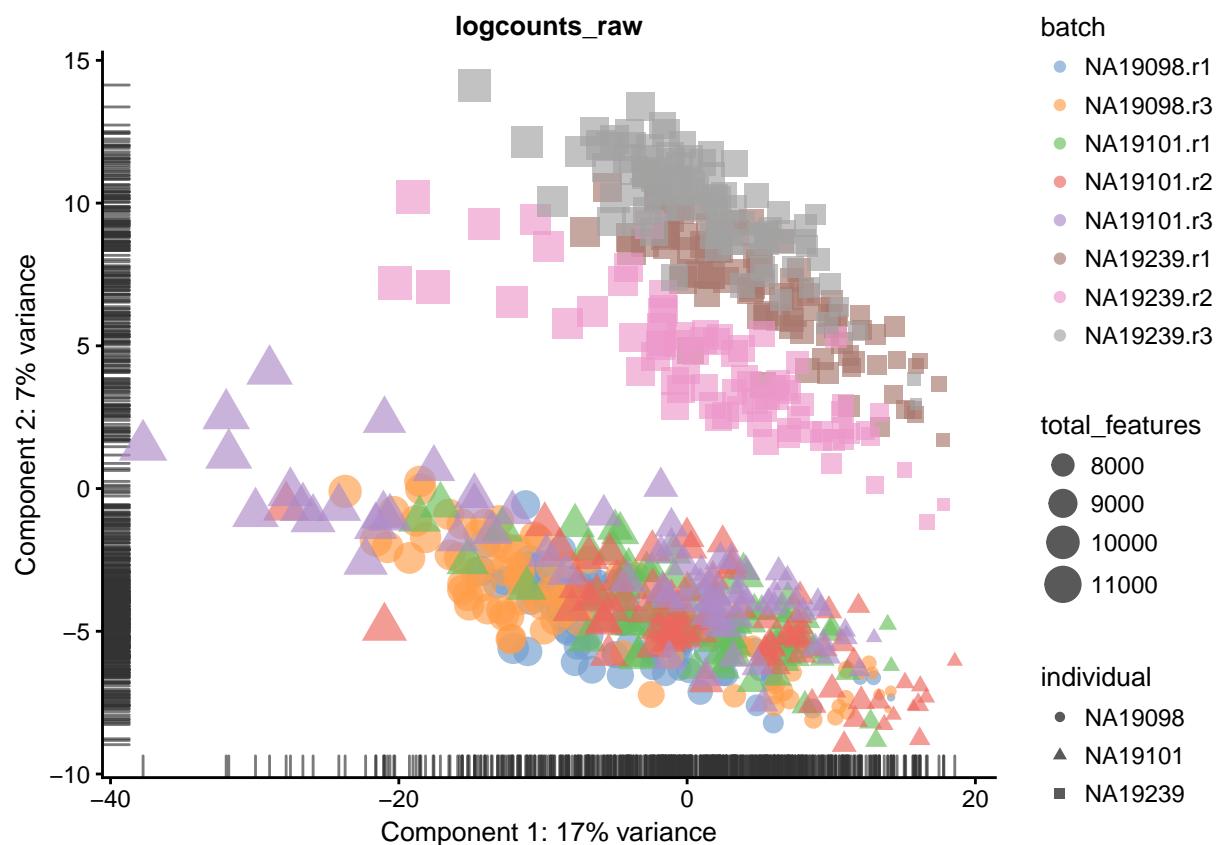
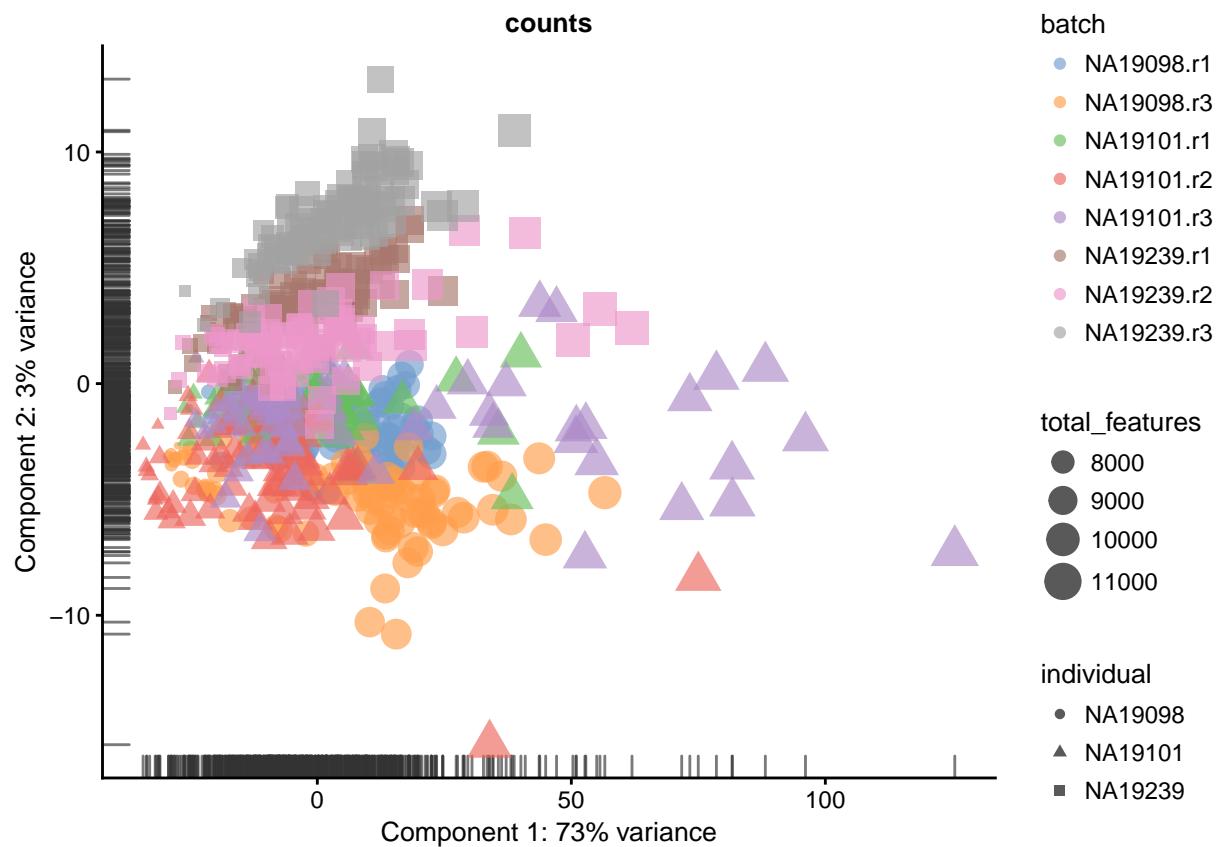
## 7.10.6 How to evaluate and compare confounder removal strategies

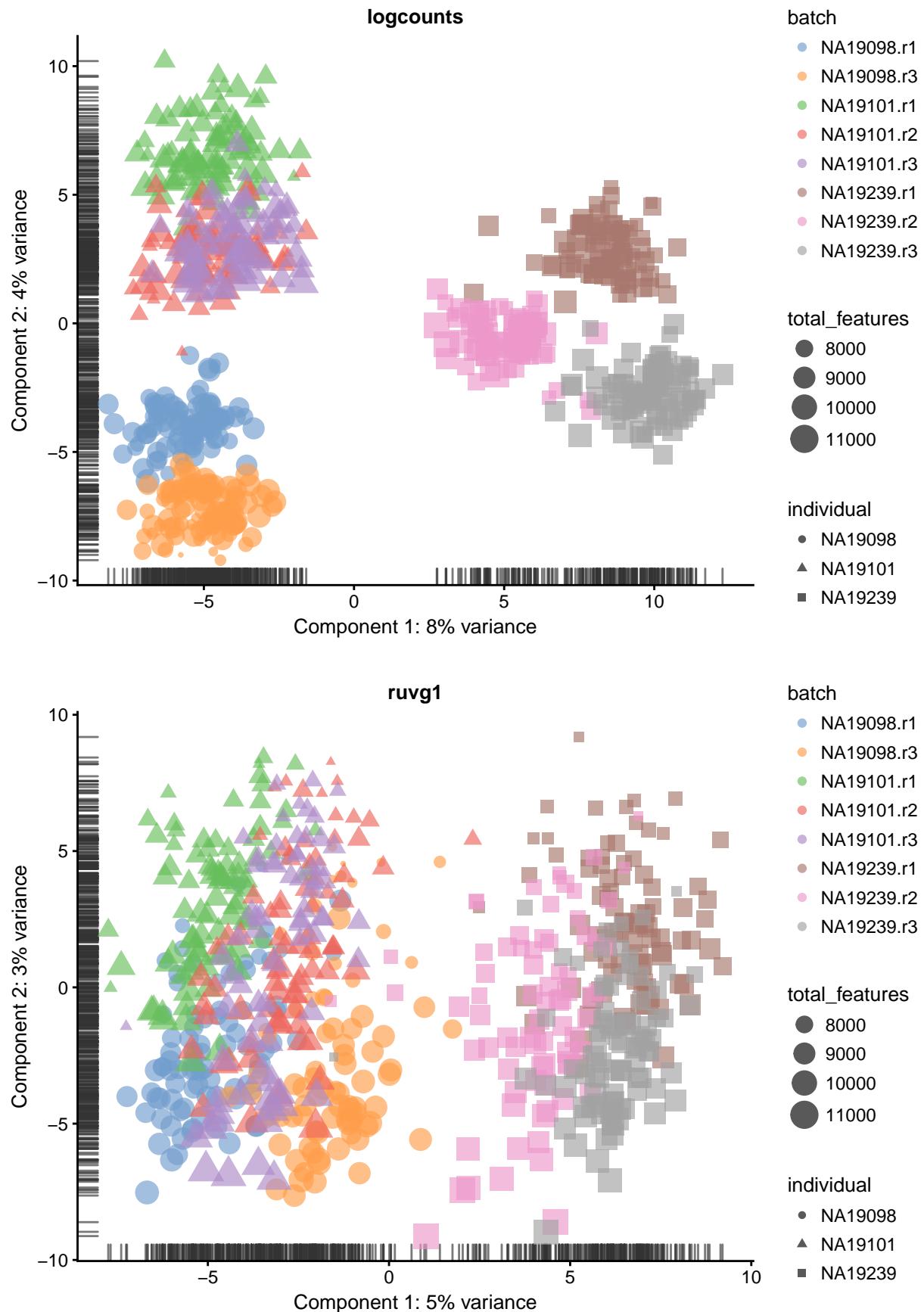
A key question when considering the different methods for removing confounders is how to quantitatively determine which one is the most effective. The main reason why comparisons are challenging is because it is often difficult to know what corresponds to technical confounders and what is interesting biological variability. Here, we consider three different metrics which are all reasonable based on our knowledge of the experimental design. Depending on the biological question that you wish to address, it is important to choose a metric that allows you to evaluate the confounders that are likely to be the biggest concern for the given situation.

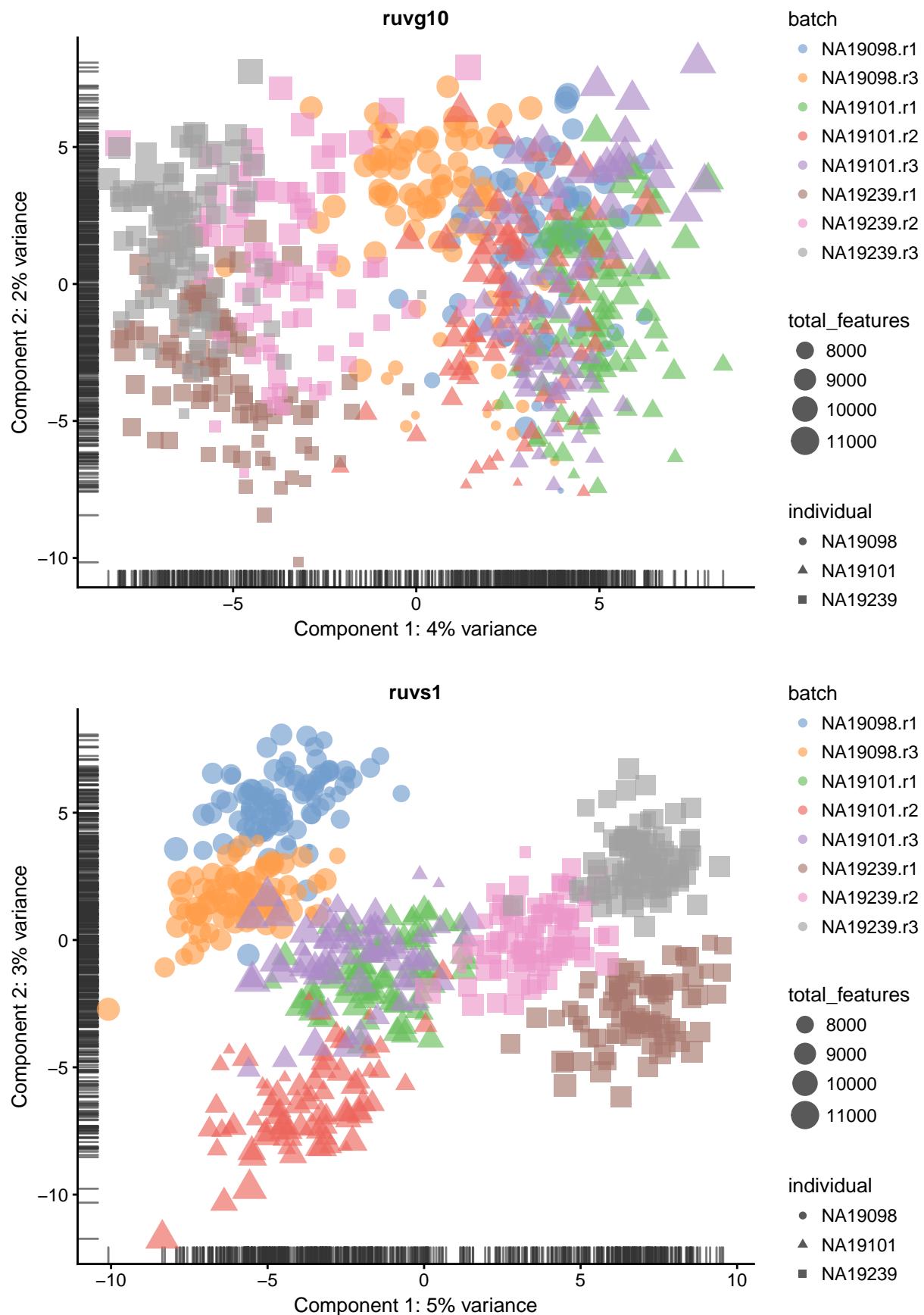
### 7.10.6.1 Effectiveness 1

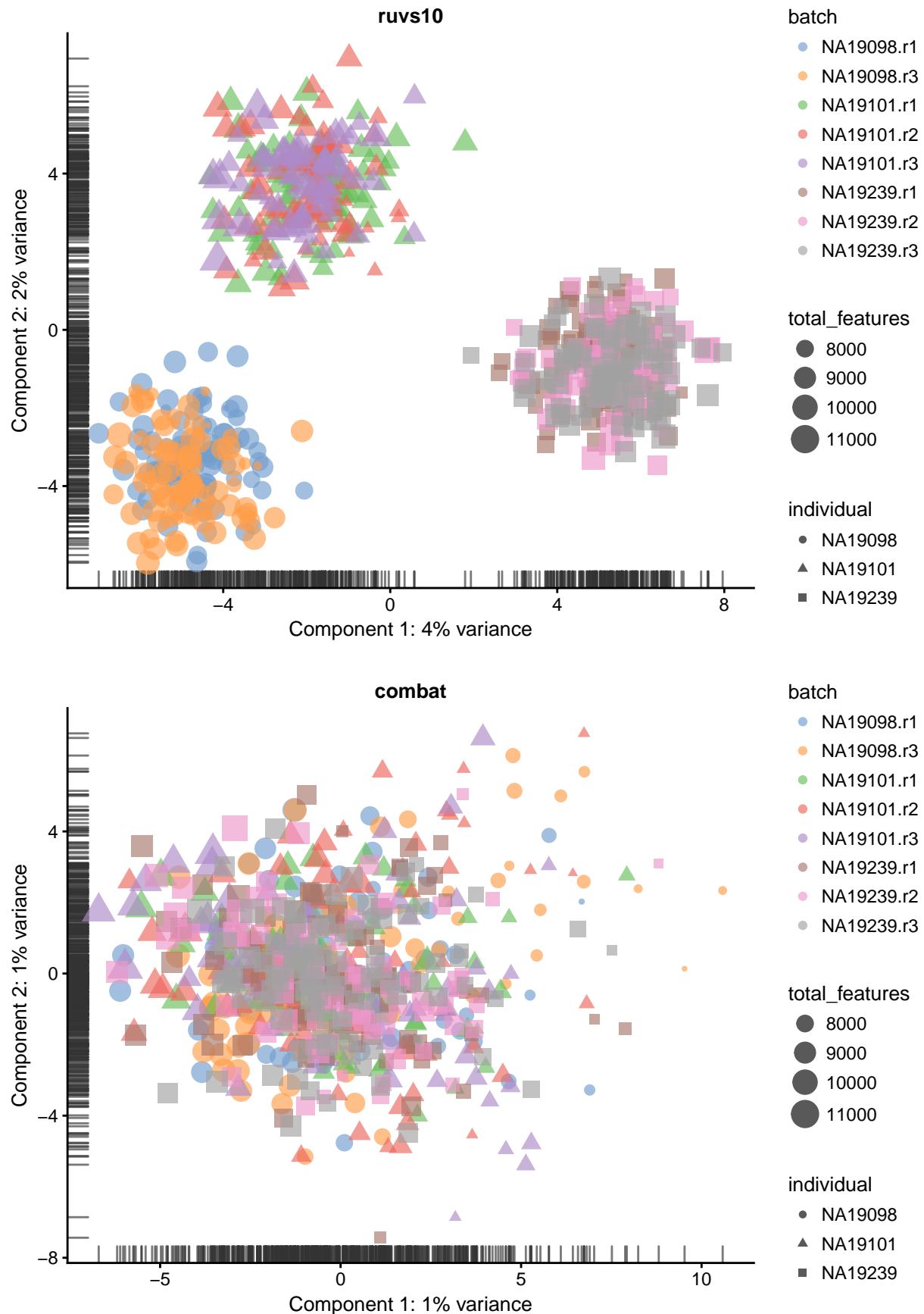
We evaluate the effectiveness of the normalization by inspecting the PCA plot where colour corresponds the technical replicates and shape corresponds to different biological samples (individuals). Separation of biological samples and interspersed batches indicates that technical variation has been removed. We always use log2-cpm normalized data to match the assumptions of PCA.

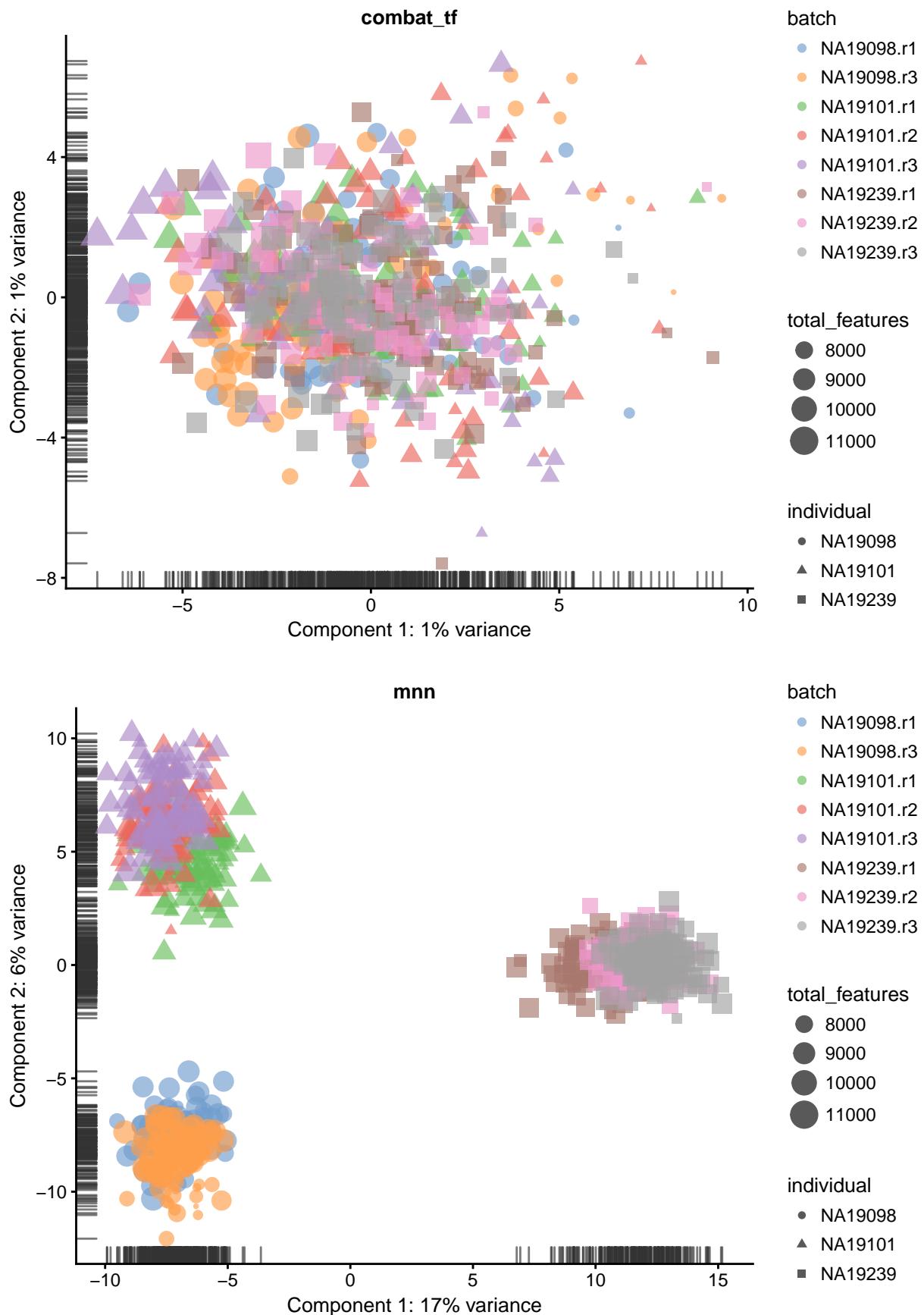
```
for(n in assayNames(umi.qc)) {
  print(
    plotPCA(
      umi.qc[endo_genes, ],
      colour_by = "batch",
      size_by = "total_features",
      shape_by = "individual",
      exprs_values = n
    ) +
    ggtitle(n)
  )
}
```

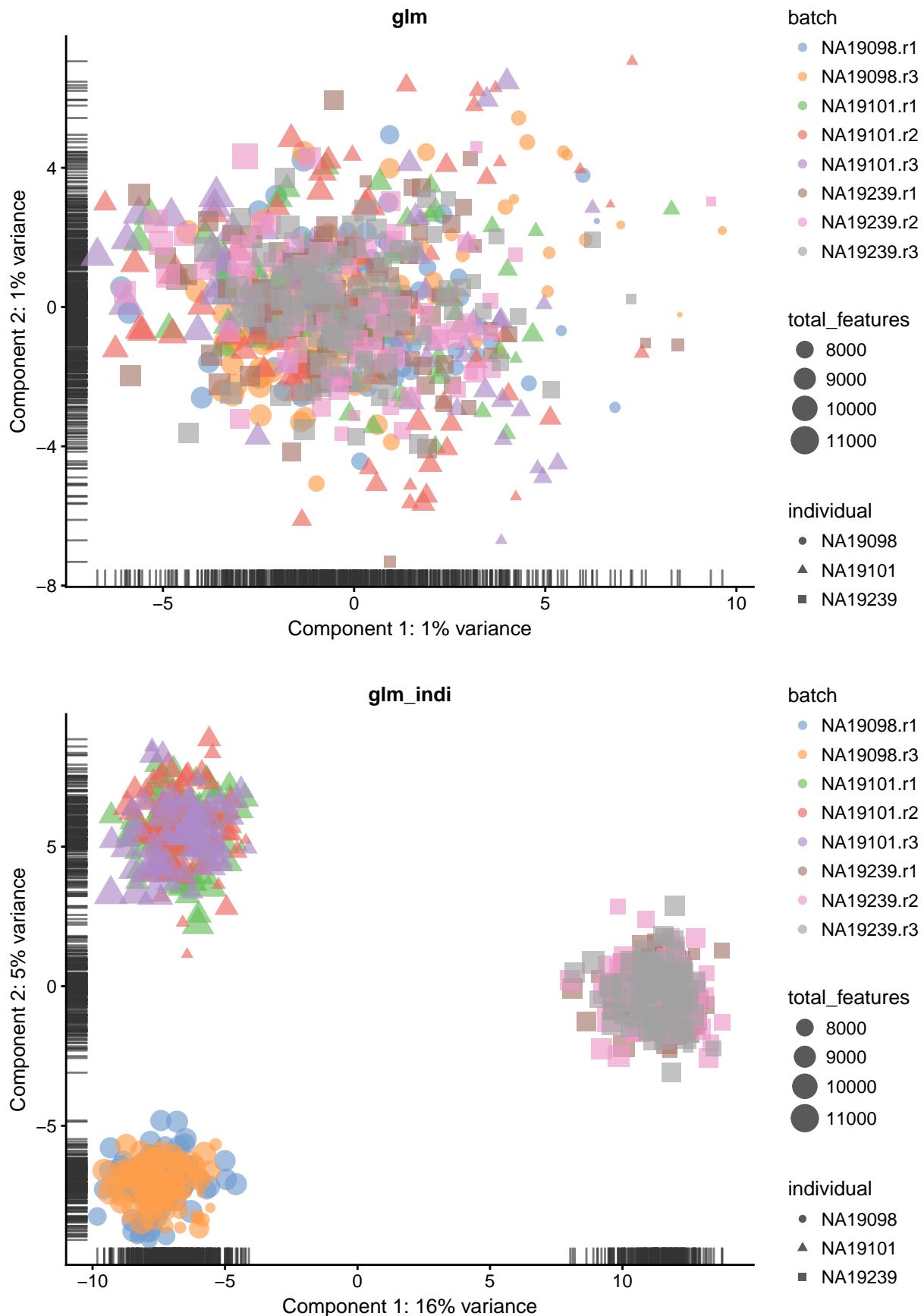












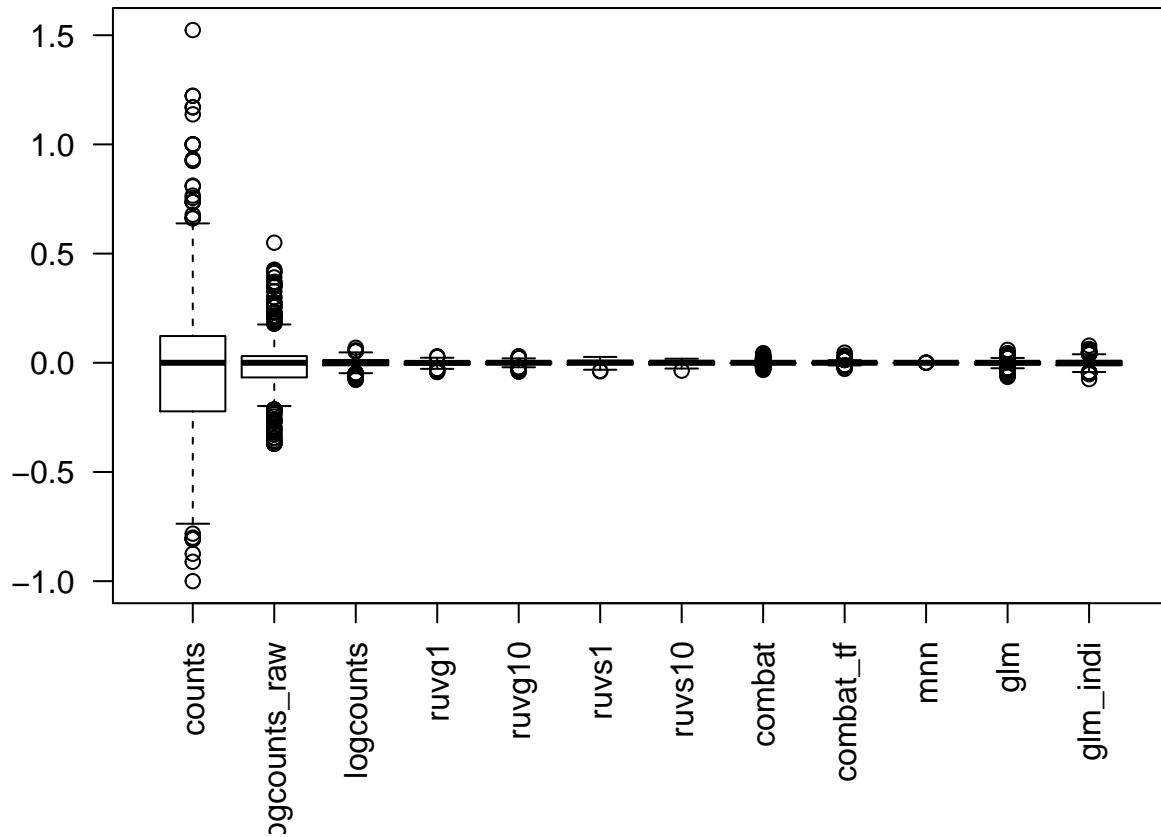
### Exercise 3

Consider different `ks` for RUV normalizations. Which gives the best results?

#### 7.10.6.2 Effectiveness 2

We can also examine the effectiveness of correction using the relative log expression (RLE) across cells to confirm technical noise has been removed from the dataset. Note RLE only evaluates whether the number of genes higher and lower than average are equal for each cell - i.e. systemic technical effects. Random technical noise between batches may not be detected by RLE.

```
res <- list()
for(n in assayNames(umi.qc)) {
  res[[n]] <- suppressWarnings(calc_cell_RLE(assay(umi.qc, n), erccs))
}
par(mar=c(6,4,1,1))
boxplot(res, las=2)
```



#### 7.10.6.3 Effectiveness 3

We can repeat the analysis from Chapter 12 to check whether batch effects have been removed.

```
for(n in assayNames(umi.qc)) {
  print(
    plotQC(
      umi.qc[endog_genes, ],
      type = "expl",
```

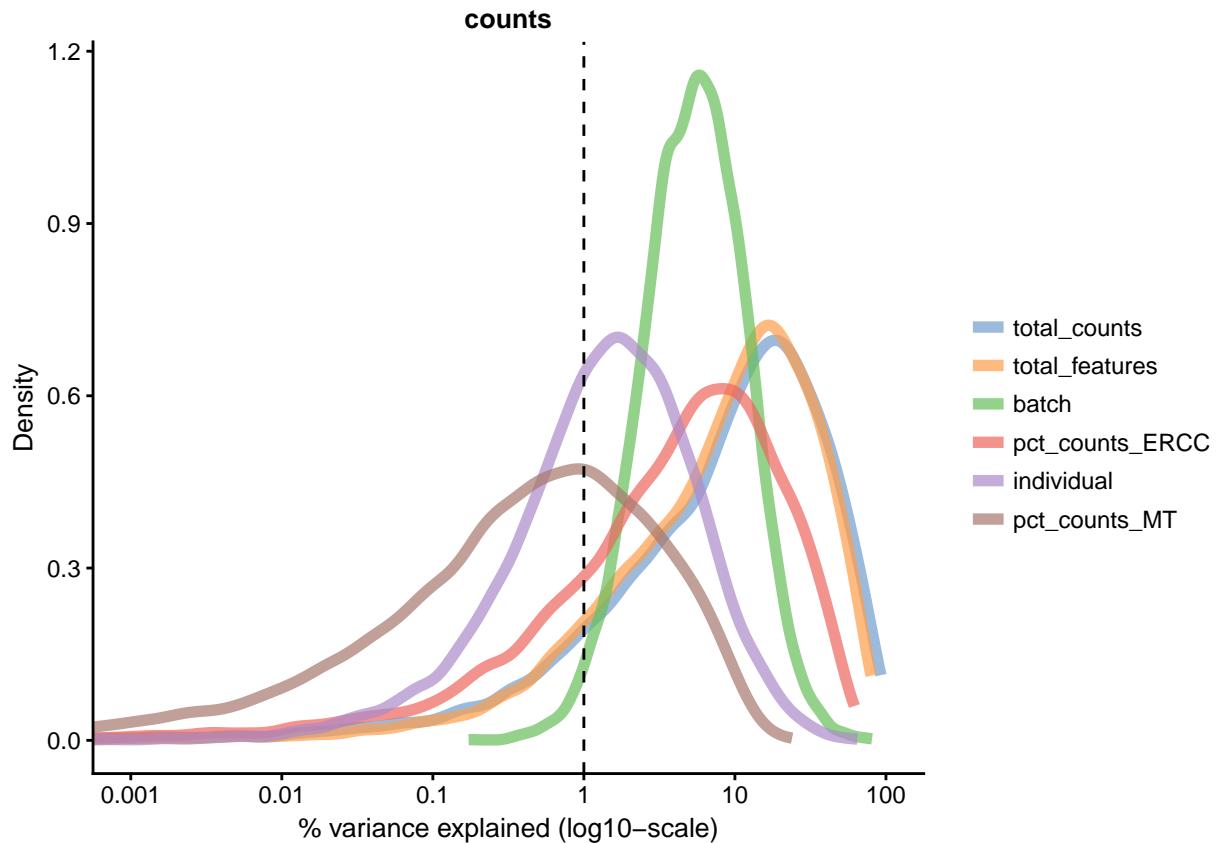


Figure 7.64: Explanatory variables (mnn)

```

exprs_values = n,
variables = c(
  "total_features",
  "total_counts",
  "batch",
  "individual",
  "pct_counts_ERCC",
  "pct_counts_MT"
)
) +
ggttitle(n)
}
}

```

**Exercise 4**

Perform the above analysis for each normalization/batch correction method. Which method(s) are most/least effective? Why is the variance accounted for by batch never lower than the variance accounted for by individual?

**7.10.6.4 Effectiveness 4**

Another method to check the efficacy of batch-effect correction is to consider the intermingling of points from different batches in local subsamples of the data. If there are no batch-effects then proportion of cells

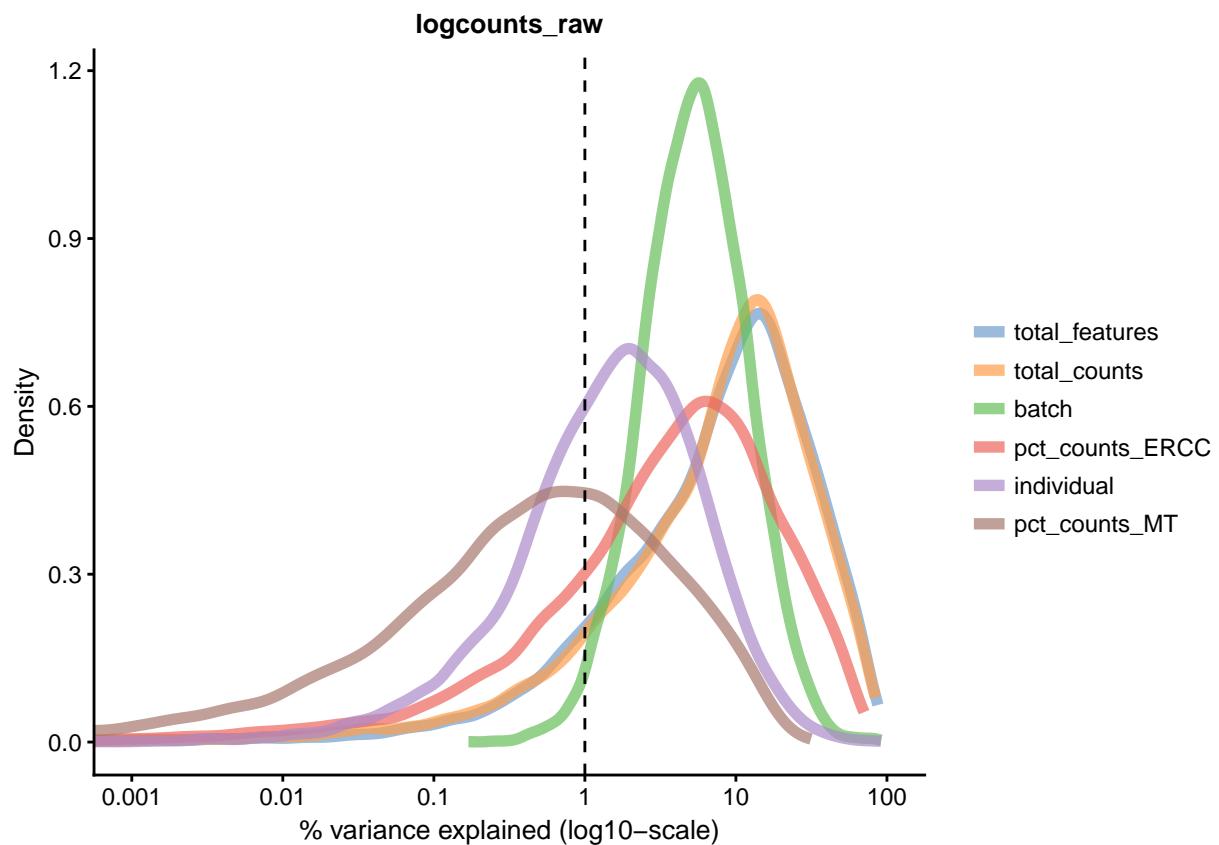


Figure 7.65: Explanatory variables (mnn)

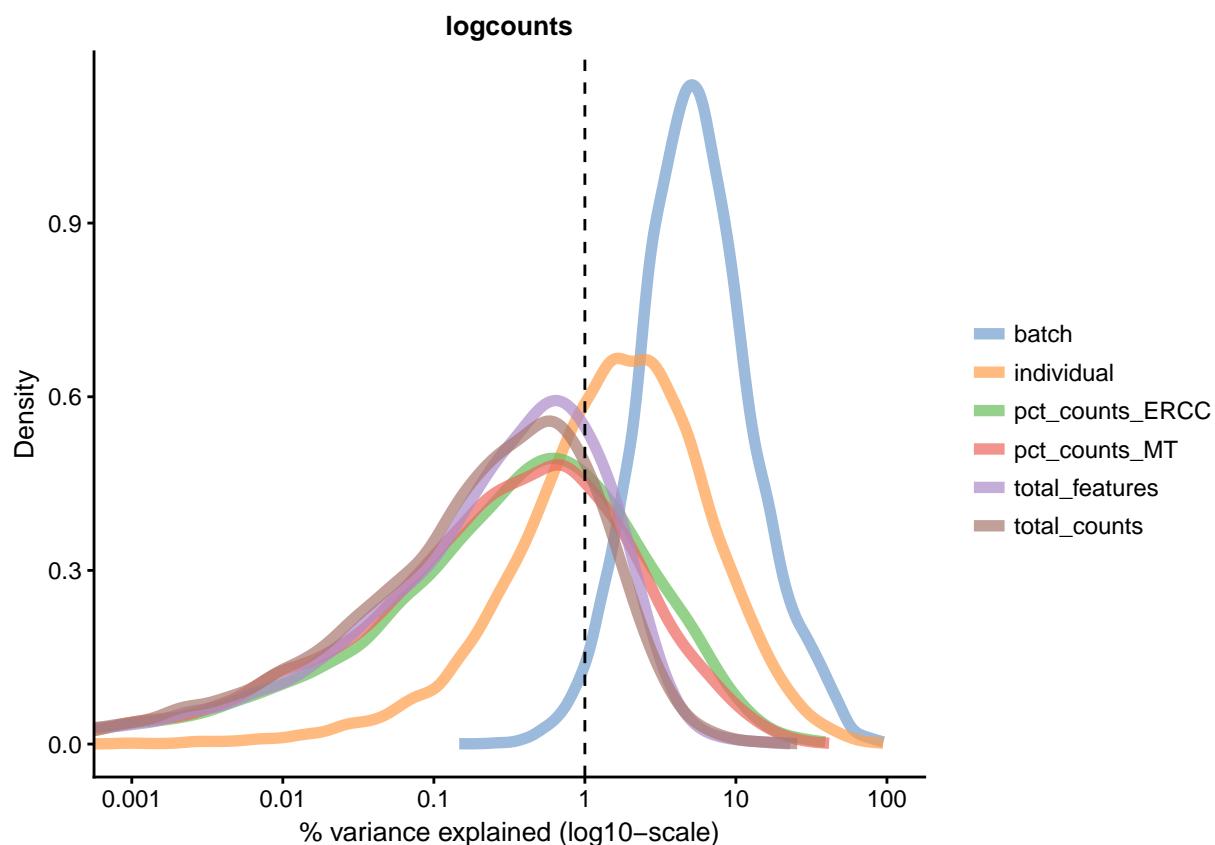


Figure 7.66: Explanatory variables (mnn)

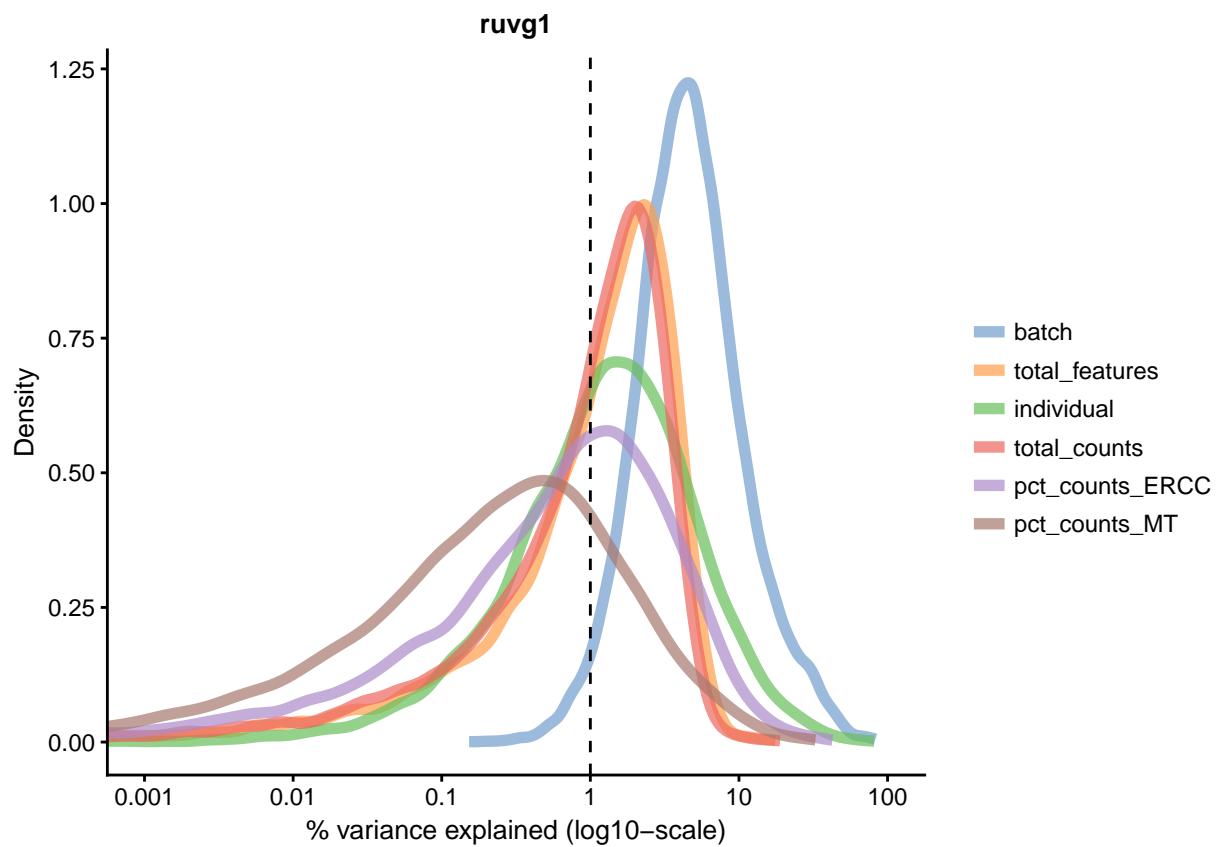


Figure 7.67: Explanatory variables (mnn)

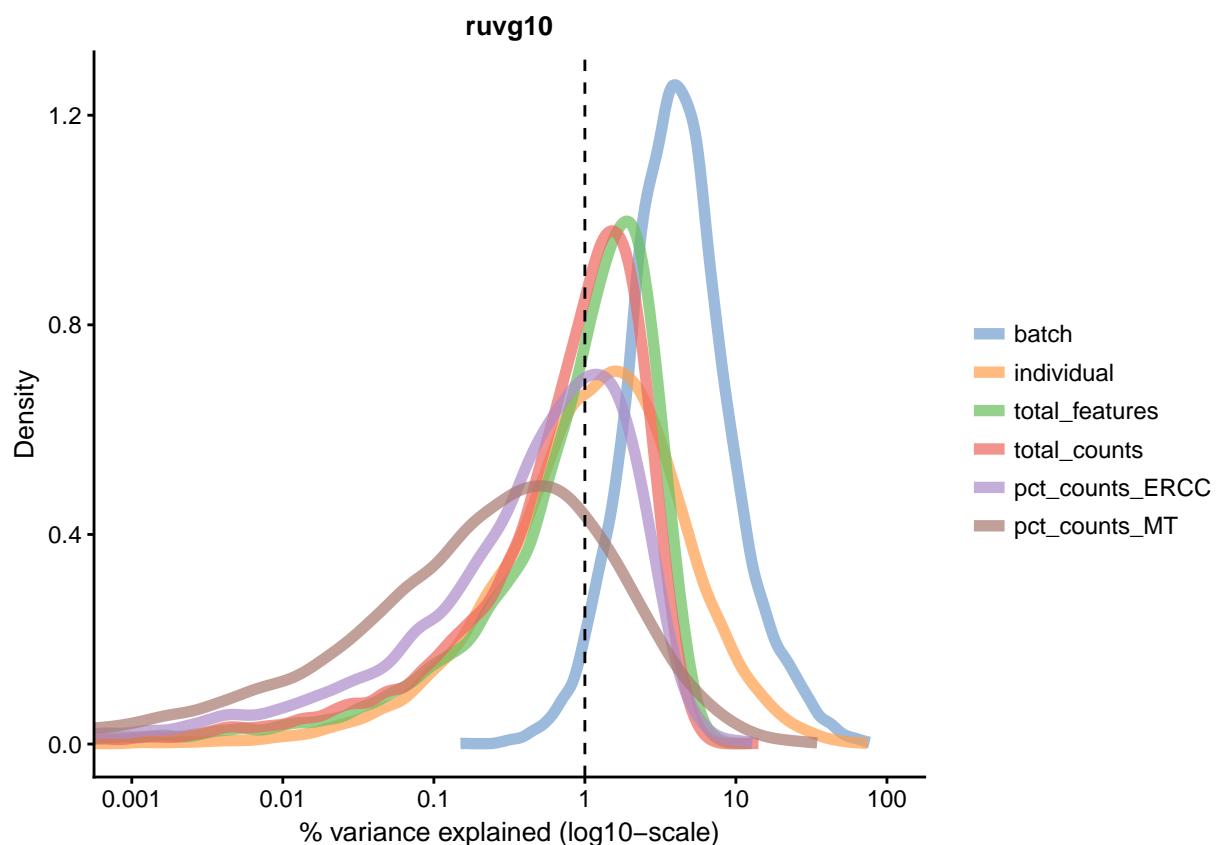


Figure 7.68: Explanatory variables (mnn)

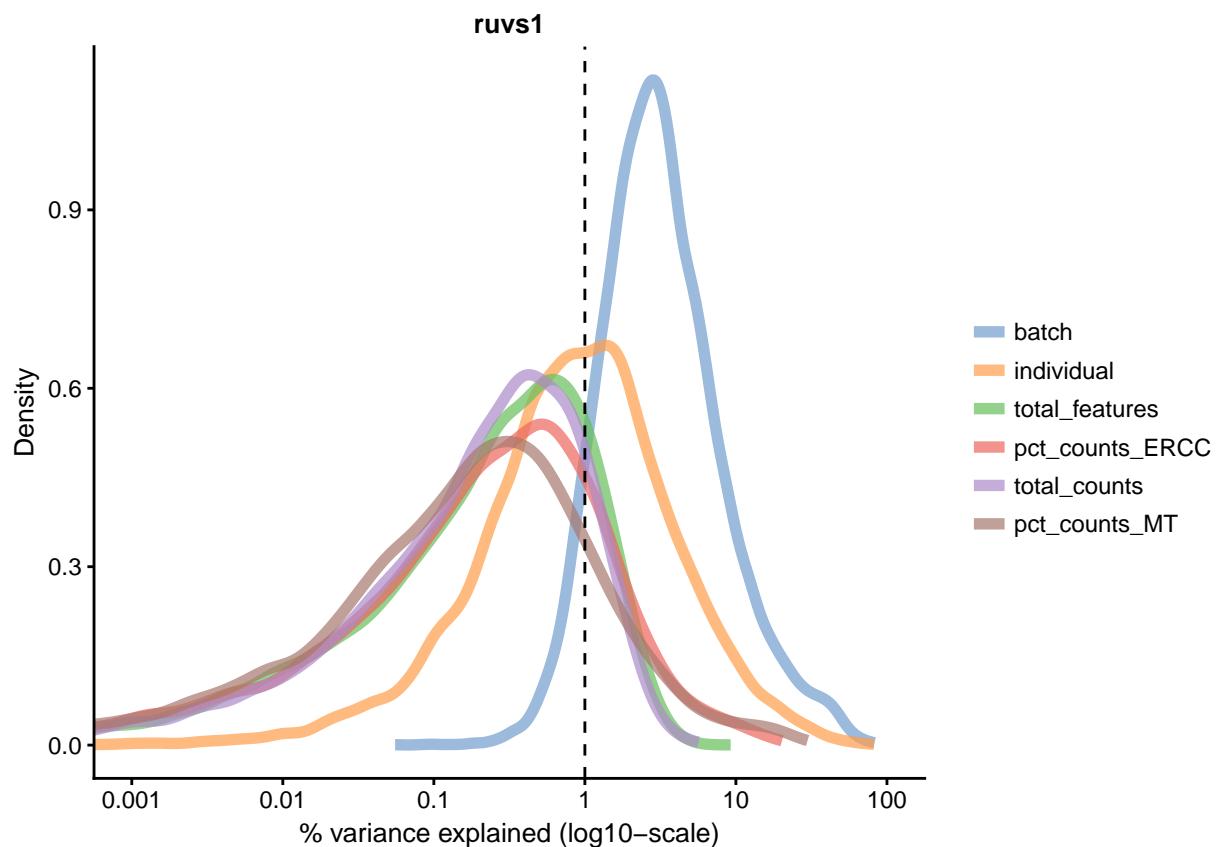


Figure 7.69: Explanatory variables (mnn)

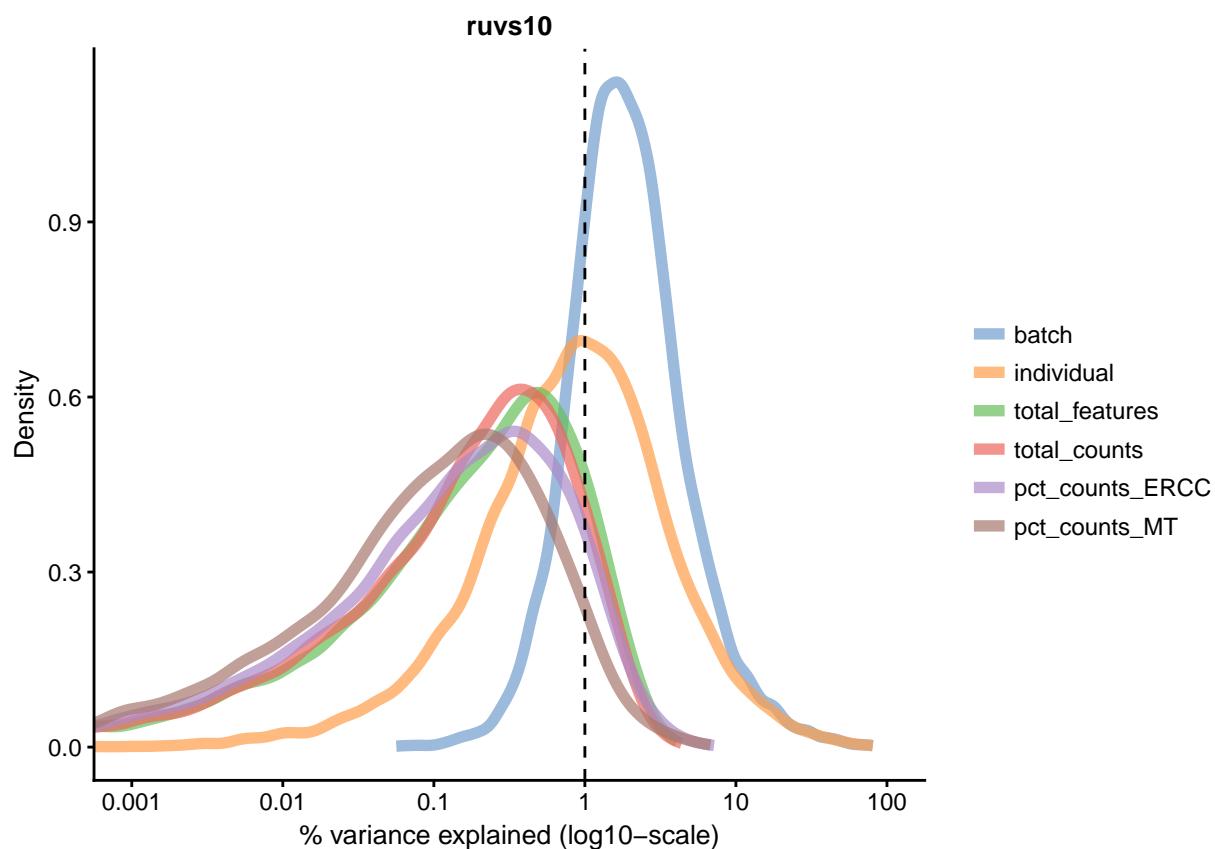


Figure 7.70: Explanatory variables (mn)

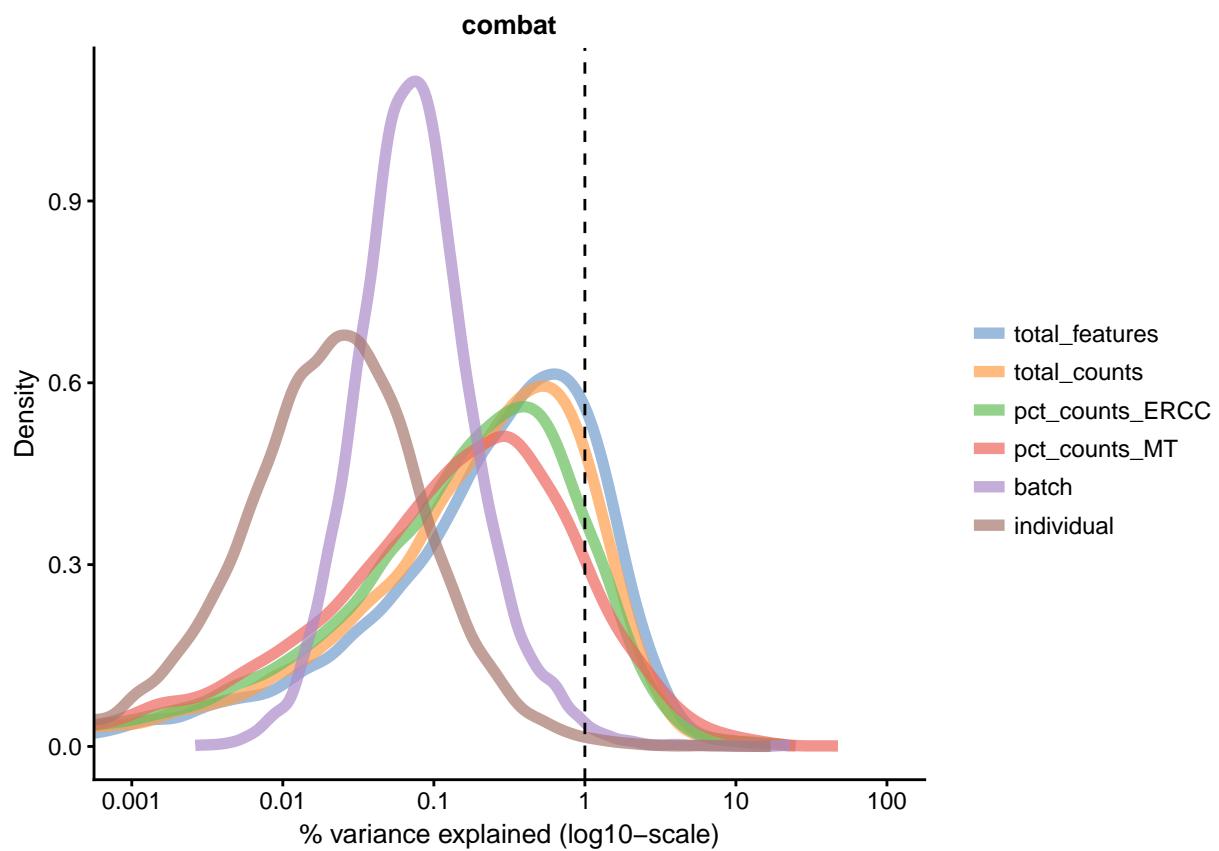


Figure 7.71: Explanatory variables (mnn)

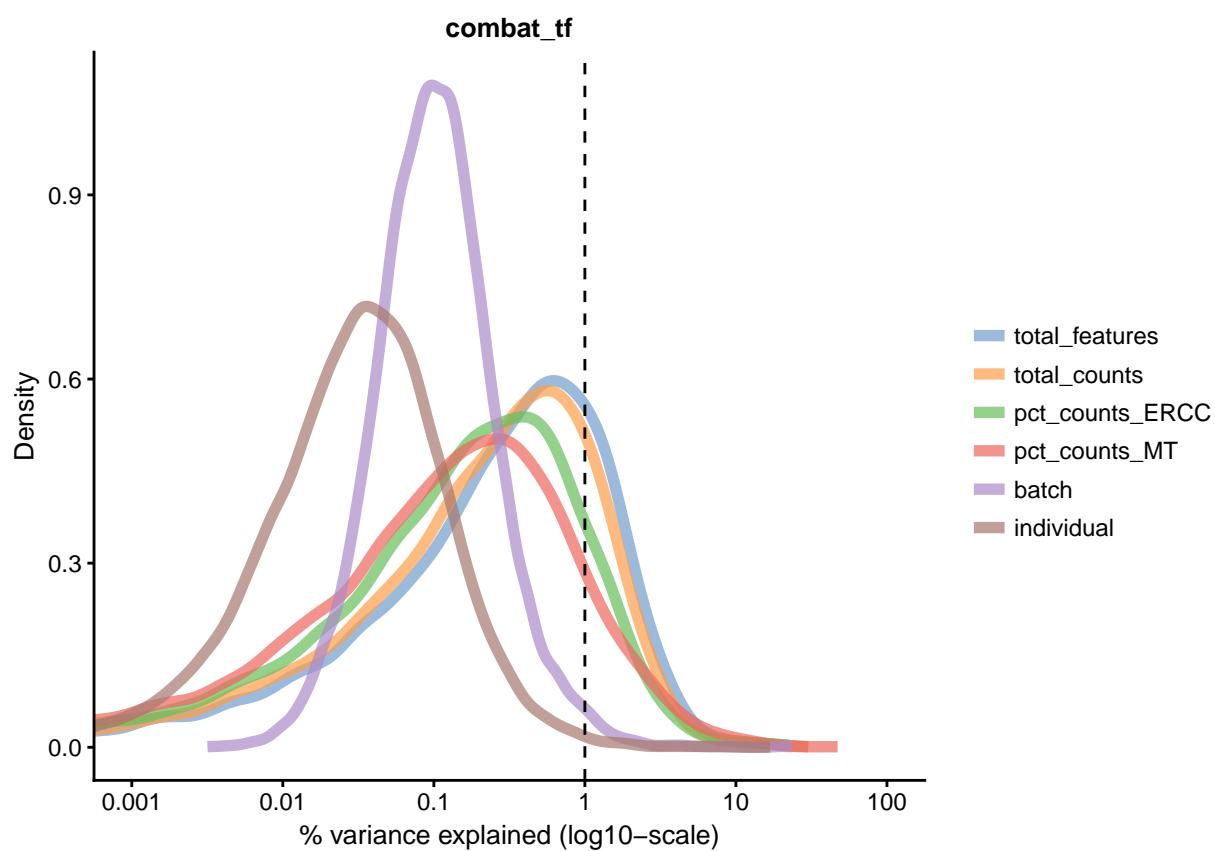


Figure 7.72: Explanatory variables (mnn)

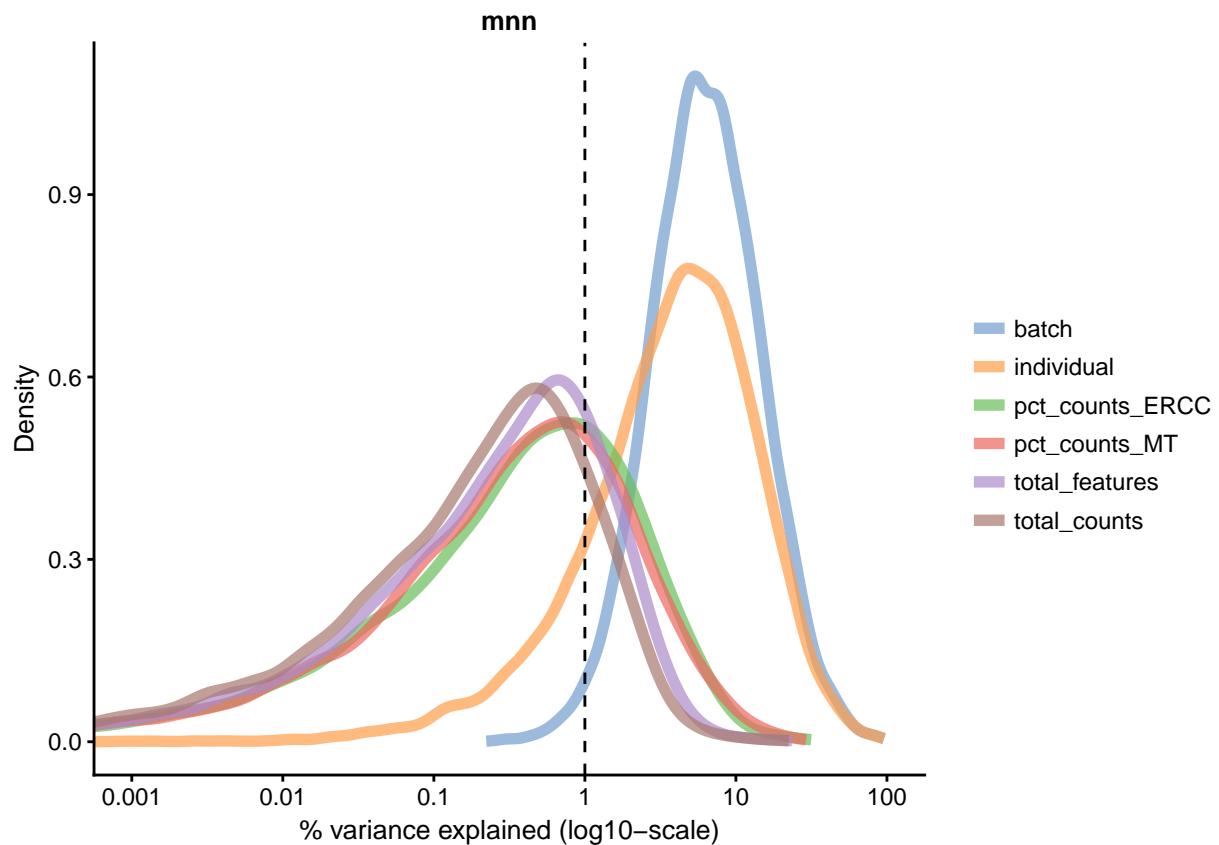


Figure 7.73: Explanatory variables (mnn)

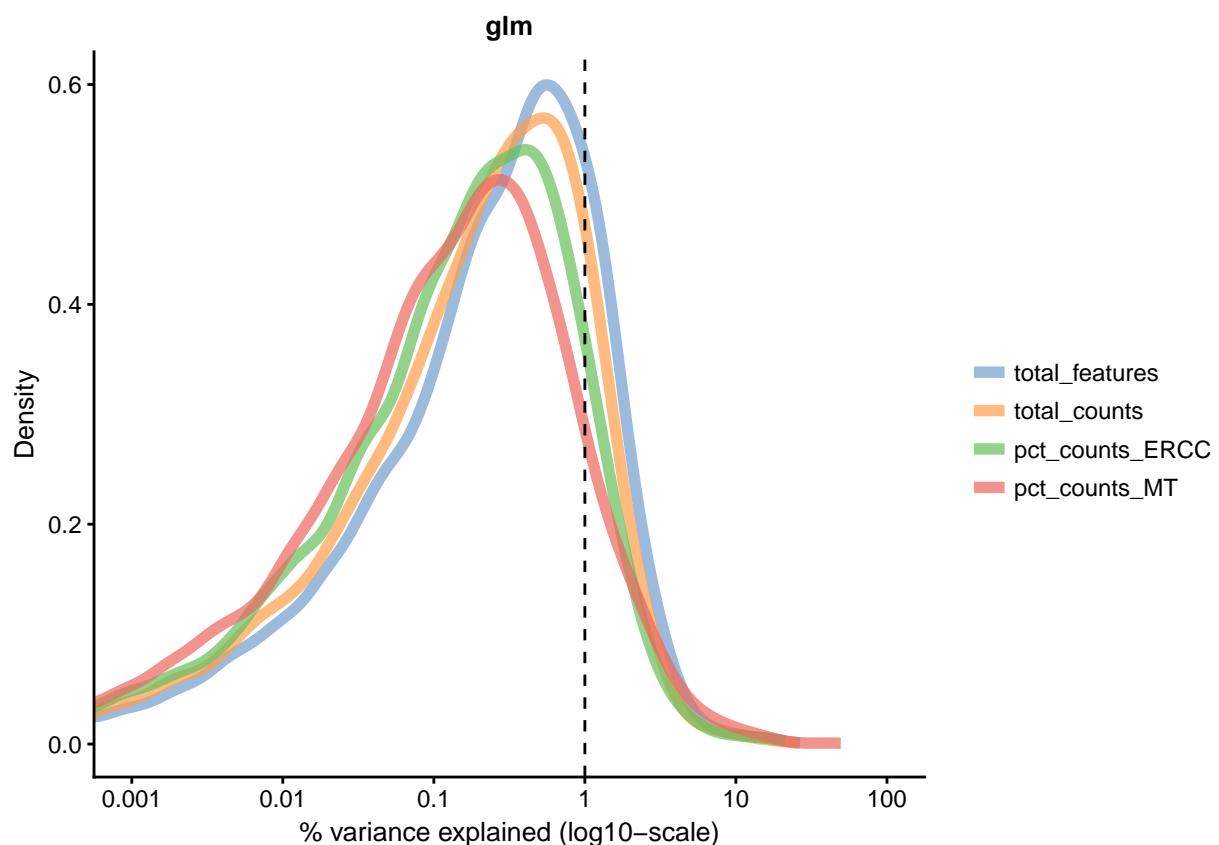


Figure 7.74: Explanatory variables (mnn)

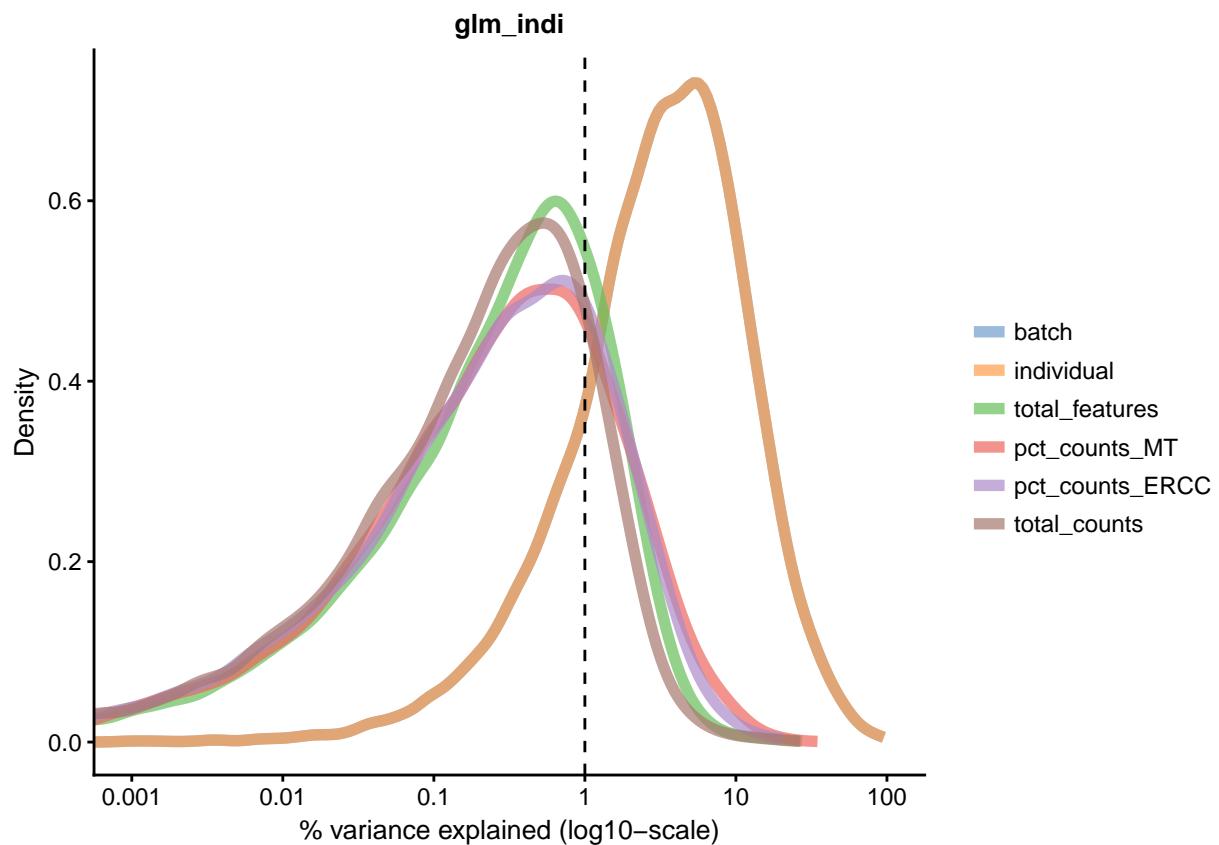


Figure 7.75: Explanatory variables (mnn)

from each batch in any local region should be equal to the global proportion of cells in each batch.

kBET (Buttner et al., 2017) takes kNN networks around random cells and tests the number of cells from each batch against a binomial distribution. The rejection rate of these tests indicates the severity of batch-effects still present in the data (high rejection rate = strong batch effects). kBET assumes each batch contains the same complement of biological groups, thus it can only be applied to the entire dataset if a perfectly balanced design has been used. However, kBET can also be applied to replicate-data if it is applied to each biological group separately. In the case of the Tung data, we will apply kBET to each individual independently to check for residual batch effects. However, this method will not identify residual batch-effects which are confounded with biological conditions. In addition, kBET does not determine if biological signal has been preserved.

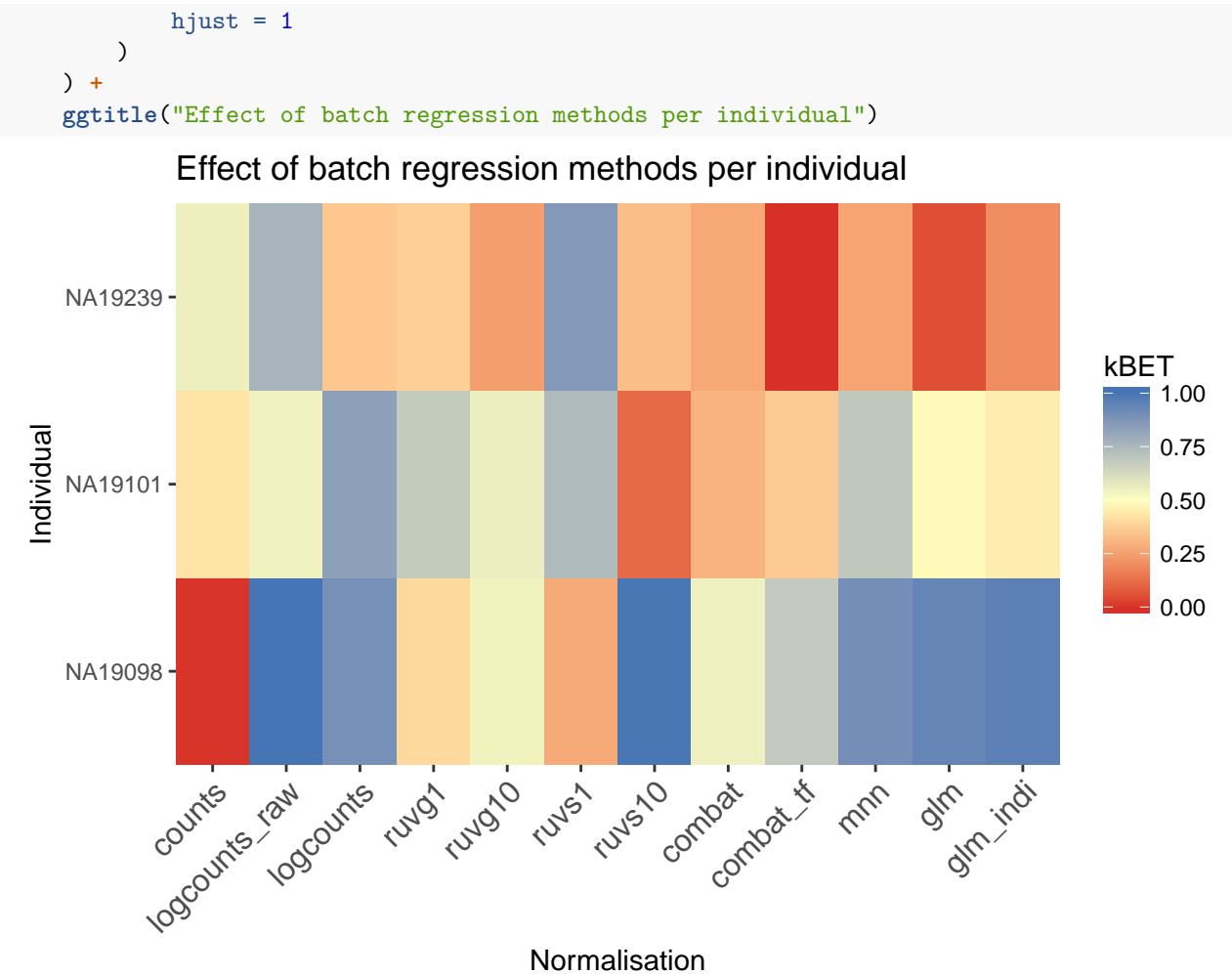
```
compare_kBET_results <- function(sce){
  indiv <- unique(sce$individual)
  norms <- assayNames(sce) # Get all normalizations
  results <- list()
  for (i in indiv){
    for (j in norms){
      tmp <- kBET(
        df = t(assay(sce[,sce$individual== i], j)),
        batch = sce$batch[sce$individual==i],
        heuristic = TRUE,
        verbose = FALSE,
        addTest = FALSE,
        plot = FALSE)
      results[[i]][[j]] <- tmp$summary$kBET.observed[1]
    }
  }
  return(as.data.frame(results))
}

eff_debatching <- compare_kBET_results(umi.qc)

require("reshape2")
require("RColorBrewer")
# Plot results
dod <- melt(as.matrix(eff_debatching), value.name = "kBET")
colnames(dod)[1:2] <- c("Normalisation", "Individual")

colorset <- c('gray', brewer.pal(n = 9, "RdYlBu"))

ggplot(dod, aes(Normalisation, Individual, fill=kBET)) +
  geom_tile() +
  scale_fill_gradient2(
    na.value = "gray",
    low = colorset[2],
    mid=colorset[6],
    high = colorset[10],
    midpoint = 0.5, limit = c(0,1)) +
  scale_x_discrete(expand = c(0, 0)) +
  scale_y_discrete(expand = c(0, 0)) +
  theme(
    axis.text.x = element_text(
      angle = 45,
      vjust = 1,
      size = 12,
```



### Exercise 5

Why do the raw counts appear to have little batch effects?

#### 7.10.7 Big Exercise

Perform the same analysis with read counts of the tung data. Use `tung/reads.rds` file to load the reads SCE object. Once you have finished please compare your results to ours (next chapter). Additionally, experiment with other combinations of normalizations and compare the results.

#### 7.10.8 sessionInfo()

```

## R version 3.4.3 (2017-11-30)
## Platform: x86_64-pc-linux-gnu (64-bit)
## Running under: Debian GNU/Linux 9 (stretch)
##
## Matrix products: default
## BLAS: /usr/lib/openblas-base/libblas.so.3
## LAPACK: /usr/lib/libopenblas-p-r0.2.19.so
##
## locale:

```

```

## [1] LC_CTYPE=en_US.UTF-8      LC_NUMERIC=C
## [3] LC_TIME=en_US.UTF-8       LC_COLLATE=en_US.UTF-8
## [5] LC_MONETARY=en_US.UTF-8   LC_MESSAGES=C
## [7] LC_PAPER=en_US.UTF-8     LC_NAME=C
## [9] LC_ADDRESS=C              LC_TELEPHONE=C
## [11] LC_MEASUREMENT=en_US.UTF-8 LC_IDENTIFICATION=C
##
## attached base packages:
## [1] stats4    parallel  methods   stats     graphics  grDevices utils
## [8] datasets  base
##
## other attached packages:
## [1] RColorBrewer_1.1-2        reshape2_1.4.3
## [3] sva_3.26.0                genefilter_1.60.0
## [5] mgcv_1.8-23               nlme_3.1-129
## [7] kBET_0.99.5               scran_1.6.7
## [9] scater_1.6.2              SingleCellExperiment_1.0.0
## [11] ggplot2_2.2.1             RUVSeq_1.12.0
## [13] edgeR_3.20.8              limma_3.34.8
## [15] EDASeq_2.12.0             ShortRead_1.36.0
## [17] GenomicAlignments_1.14.1  SummarizedExperiment_1.8.1
## [19] DelayedArray_0.4.1        matrixStats_0.53.0
## [21] Rsamtools_1.30.0          GenomicRanges_1.30.1
## [23] GenomeInfoDb_1.14.0       Biostrings_2.46.0
## [25] XVector_0.18.0            IRanges_2.12.0
## [27] S4Vectors_0.16.0          BiocParallel_1.12.0
## [29] Biobase_2.38.0            BiocGenerics_0.24.0
## [31] scRNA.seq.funcs_0.1.0     knitr_1.19
##
## loaded via a namespace (and not attached):
## [1] Rtsne_0.13                 ggbbeeswarm_0.6.0      colorspace_1.3-2
## [4] rjson_0.2.15               hwriter_1.3.2         dynamicTreeCut_1.63-1
## [7] rprojroot_1.3-2            DT_0.4                  bit64_0.9-7
## [10] AnnotationDbi_1.40.0      splines_3.4.3         R.methodsS3_1.7.1
## [13] tximport_1.6.0             DESeq_1.30.0           geneplotter_1.56.0
## [16] annotate_1.56.1            cluster_2.0.6         R.oo_1.21.0
## [19] shinydashboard_0.6.1      shiny_1.0.5            compiler_3.4.3
## [22] httr_1.3.1                backports_1.1.2       assertthat_0.2.0
## [25] Matrix_1.2-7.1            lazyeval_0.2.1        htmltools_0.3.6
## [28] prettyunits_1.0.2          tools_3.4.3            igraph_1.1.2
## [31] bindrcpp_0.2                gtable_0.2.0           glue_1.2.0
## [34] GenomeInfoDbData_1.0.0    dplyr_0.7.4            Rcpp_0.12.15
## [37] rtracklayer_1.38.3         xfun_0.1                stringr_1.2.0
## [40] mime_0.5                  hypergeo_1.2-13       statmod_1.4.30
## [43] XML_3.98-1.9              zoo_1.8-1               zlibbioc_1.24.0
## [46] MASS_7.3-45                scales_0.5.0           aroma.light_3.8.0
## [49] rhdf5_2.22.0              yaml_2.1.16            memoise_1.1.0
## [52] gridExtra_2.3              biomaRt_2.34.2         latticeExtra_0.6-28
## [55] stringi_1.1.6              RSQLite_2.0             RMySQL_0.10.13
## [58] orthopolynom_1.0-5        GenomicFeatures_1.30.3 conffrac_1.1-11
## [61] rlang_0.1.6                pkgconfig_2.0.1        moments_0.14
## [64] bitops_1.0-6                evaluate_0.10.1        lattice_0.20-34
## [67] bindr_0.1                  labeling_0.3           htmlwidgets_1.0
## [70] cowplot_0.9.2              bit_1.1-12             deSolve_1.20

```

```

## [ 73] plyr_1.8.4           magrittr_1.5          bookdown_0.6
## [ 76] R6_2.2.2             DBI_0.7              pillar_1.1.0
## [ 79] survival_2.40-1      RCurl_1.95-4.10    tibble_1.4.2
## [ 82] rmarkdown_1.8          viridis_0.5.0       progress_1.1.2
## [ 85] locfit_1.5-9.1       grid_3.4.3         data.table_1.10.4-3
## [ 88] FNN_1.1               blob_1.1.0        digest_0.6.15
## [ 91] xtable_1.8-2         httpuv_1.3.5      elliptic_1.3-7
## [ 94] R.utils_2.6.0         munsell_0.4.3     beeswarm_0.2.3
## [ 97] viridisLite_0.3.0    vipor_0.4.5

```

## 7.11 Dealing with confounders (Reads)

```

library(scRNA.seq.funcs)
library(RUVSeq)
library(scater)
library(SingleCellExperiment)
library(scran)
library(kBET)
library(sva) # Combat
library(edgeR)
set.seed(1234567)
options(stringsAsFactors = FALSE)
reads <- readRDS("tung/reads.rds")
reads.qc <- reads$rowData(reads)$use, colData(reads)$use]
endog_genes <- !rowData(reads.qc)$is_feature_control
erccs <- rowData(reads.qc)$is_feature_control

qclust <- quickCluster(reads.qc, min.size = 30)
reads.qc <- computeSumFactors(reads.qc, sizes = 15, clusters = qclust)
reads.qc <- normalize(reads.qc)

ruvg <- RUVg(counts(reads.qc), erccs, k = 1)
assay(reads.qc, "ruvg1") <- log2(
  t(t(ruvg$normalizedCounts) / colSums(ruvg$normalizedCounts) * 1e6) + 1
)
ruvg <- RUVg(counts(reads.qc), erccs, k = 10)
assay(reads.qc, "ruvg10") <- log2(
  t(t(ruvg$normalizedCounts) / colSums(ruvg$normalizedCounts) * 1e6) + 1
)

scIdx <- matrix(-1, ncol = max(table(reads.qc$individual)), nrow = 3)
tmp <- which(reads.qc$individual == "NA19098")
scIdx[1, 1:length(tmp)] <- tmp
tmp <- which(reads.qc$individual == "NA19101")
scIdx[2, 1:length(tmp)] <- tmp
tmp <- which(reads.qc$individual == "NA19239")
scIdx[3, 1:length(tmp)] <- tmp
cIdx <- rownames(reads.qc)
ruvs <- RUVs(counts(reads.qc), cIdx, k = 1, scIdx = scIdx, isLog = FALSE)
assay(reads.qc, "ruvs1") <- log2(
  t(t(ruvs$normalizedCounts) / colSums(ruvs$normalizedCounts) * 1e6) + 1
)

```

```

ruvs <- RUWs(counts(reads.qc), cIdx, k = 10, scIdx = scIdx, isLog = FALSE)
assay(reads.qc, "ruvs10") <- log2(
  t(t(ruv$normalizedCounts) / colSums(ruv$normalizedCounts) * 1e6) + 1
)

combat_data <- logcounts(reads.qc)
mod_data <- as.data.frame(t(combat_data))
# Basic batch removal
mod0 = model.matrix(~ 1, data = mod_data)
# Preserve biological variability
mod1 = model.matrix(~ reads.qc$individual, data = mod_data)
# adjust for total genes detected
mod2 = model.matrix(~ reads.qc$total_features, data = mod_data)
assay(reads.qc, "combat") <- ComBat(
  dat = t(mod_data),
  batch = factor(reads.qc$batch),
  mod = mod0,
  par.prior = TRUE,
  prior.plots = FALSE
)

```

## Standardizing Data across genes

### Exercise 1

```

## Standardizing Data across genes
do_mnn <- function(data.qc) {
  batch1 <- logcounts(data.qc[, data.qc$replicate == "r1"])
  batch2 <- logcounts(data.qc[, data.qc$replicate == "r2"])
  batch3 <- logcounts(data.qc[, data.qc$replicate == "r3"])

  if (ncol(batch2) > 0) {
    x = mnnCorrect(
      batch1, batch2, batch3,
      k = 20,
      sigma = 0.1,
      cos.norm.in = TRUE,
      svd.dim = 2
    )
    res1 <- x$corrected[[1]]
    res2 <- x$corrected[[2]]
    res3 <- x$corrected[[3]]
    dimnames(res1) <- dimnames(batch1)
    dimnames(res2) <- dimnames(batch2)
    dimnames(res3) <- dimnames(batch3)
    return(cbind(res1, res2, res3))
  } else {
    x = mnnCorrect(
      batch1, batch3,
      k = 20,
      sigma = 0.1,
      cos.norm.in = TRUE,
      svd.dim = 2
    )
  }
}

```

```

        res1 <- x$corrected[[1]]
        res3 <- x$corrected[[2]]
        dimnames(res1) <- dimnames(batch1)
        dimnames(res3) <- dimnames(batch3)
        return(cbind(res1, res3))
    }
}

indi1 <- do_mnn(reads.qc[, reads.qc$individual == "NA19098"])
indi2 <- do_mnn(reads.qc[, reads.qc$individual == "NA19101"])
indi3 <- do_mnn(reads.qc[, reads.qc$individual == "NA19239"])

assay(reads.qc, "mnn") <- cbind(indi1, indi2, indi3)

# For a balanced design:
#assay(reads.qc, "mnn") <- mnnCorrect(
#    list(B1 = logcounts(batch1), B2 = logcounts(batch2), B3 = logcounts(batch3)),
#    k = 20,
#    sigma = 0.1,
#    cos.norm = TRUE,
#    svd.dim = 2
#)

glm_fun <- function(g, batch, indi) {
    model <- glm(g ~ batch + indi)
    model$coef[1] <- 0 # replace intercept with 0 to preserve reference batch.
    return(model$coef)
}
effects <- apply(
    logcounts(reads.qc),
    1,
    glm_fun,
    batch = reads.qc$batch,
    indi = reads.qc$individual
)
corrected <- logcounts(reads.qc) - t(effects[as.numeric(factor(reads.qc$batch)), ])
assay(reads.qc, "glm") <- corrected

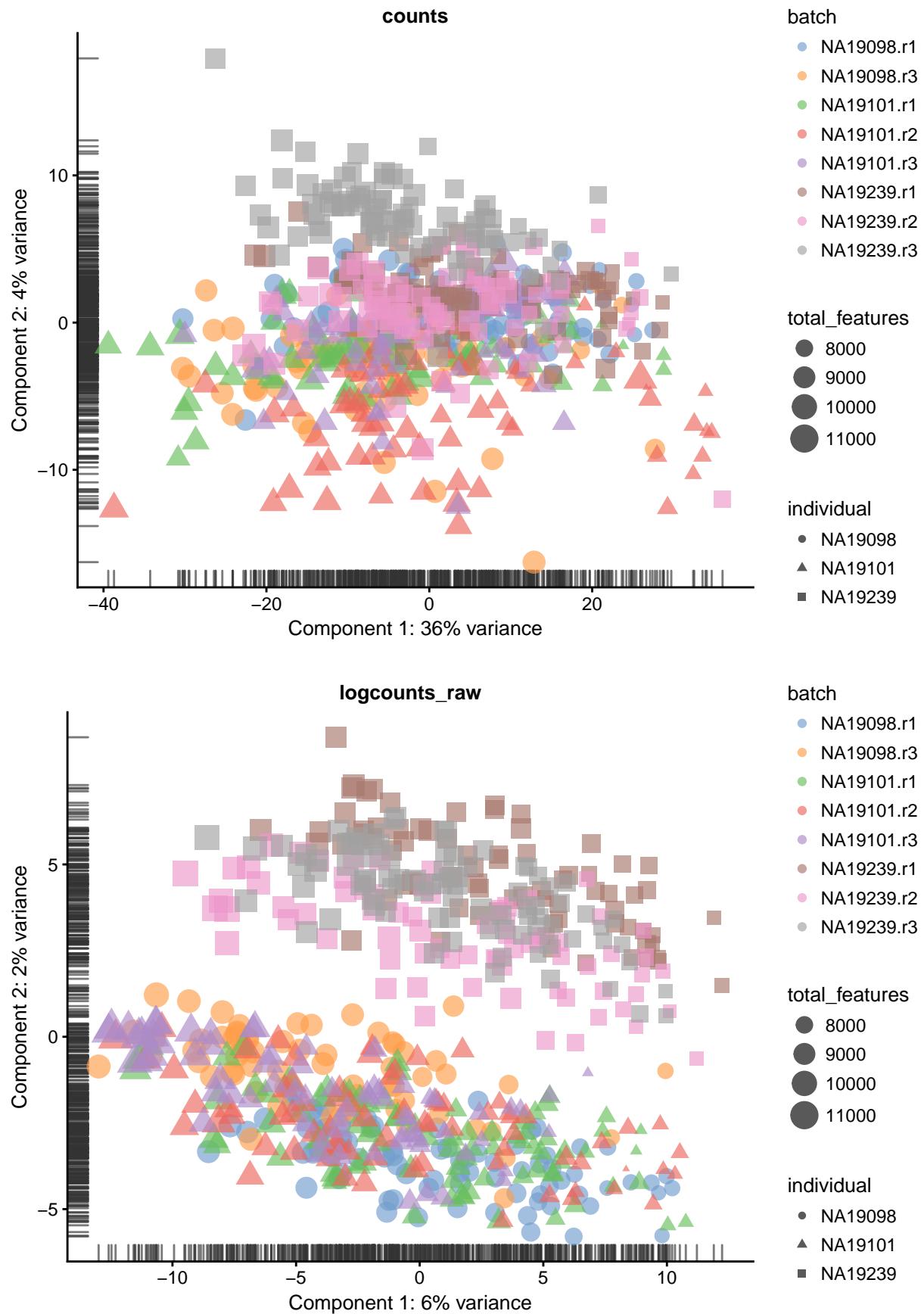
```

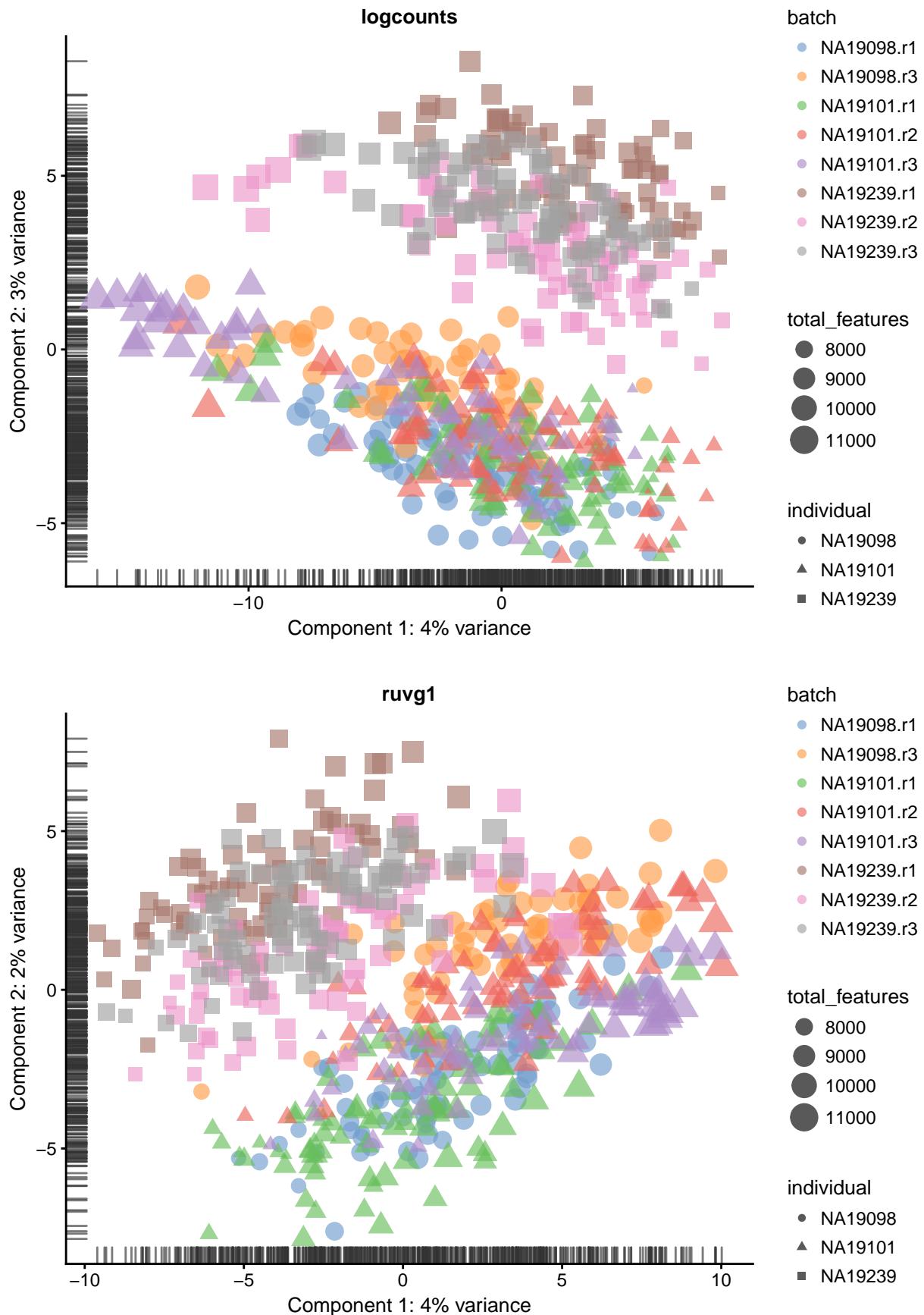
## Exercise 2

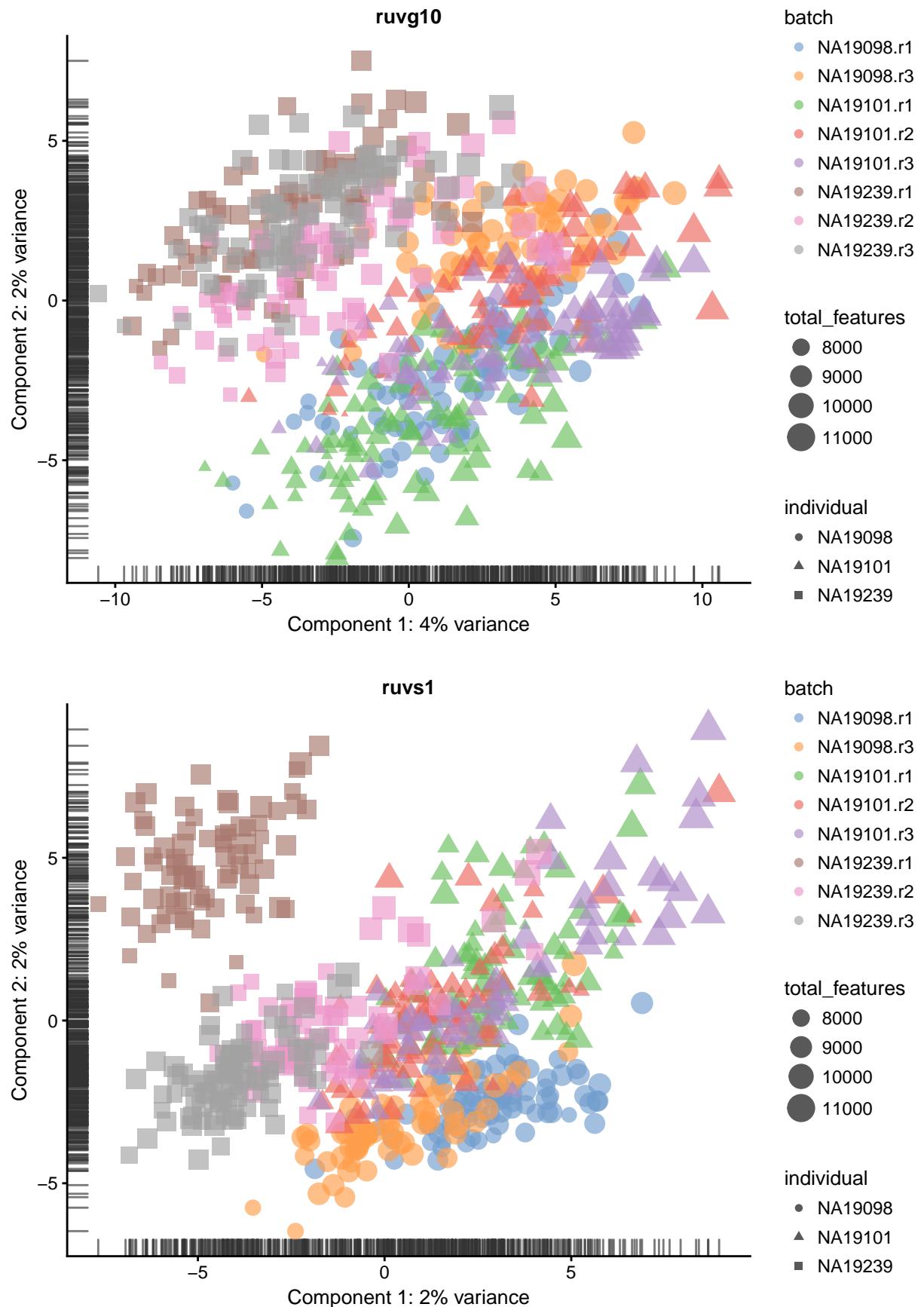
```

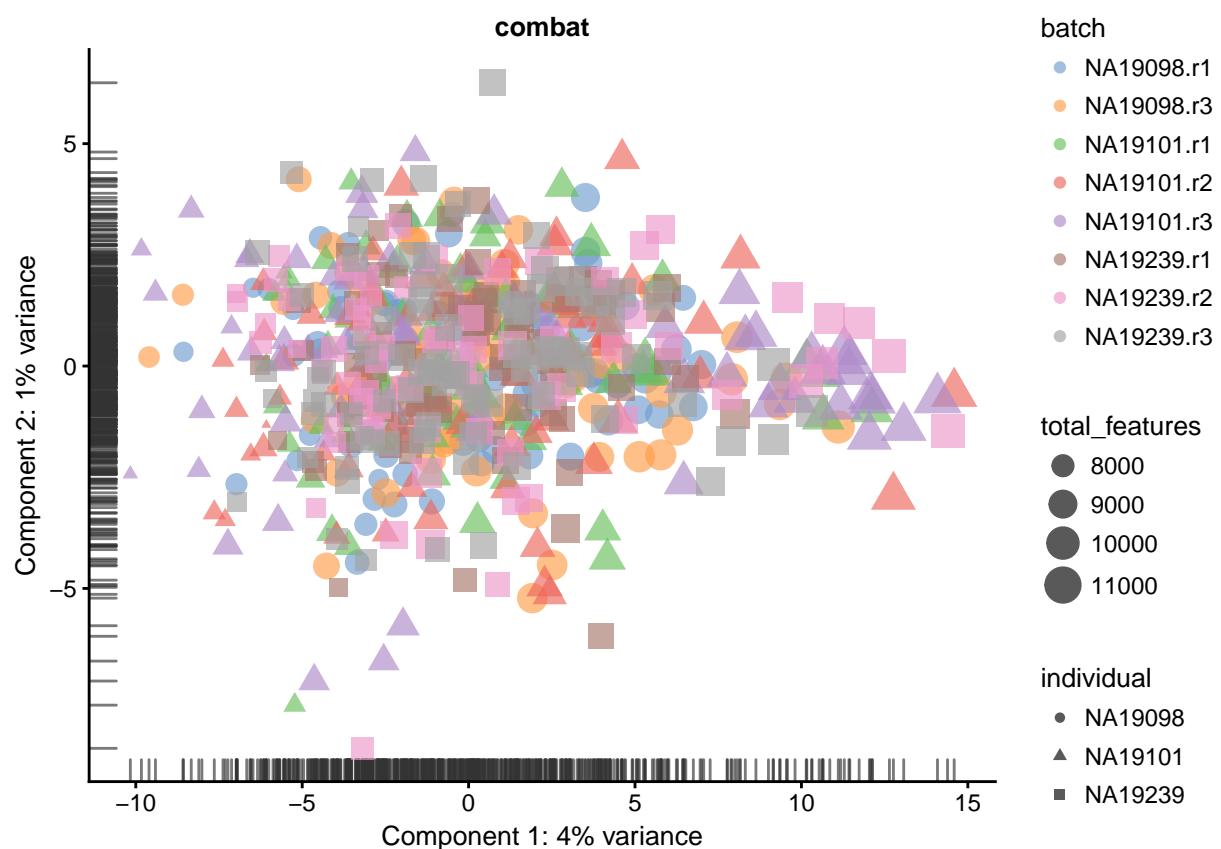
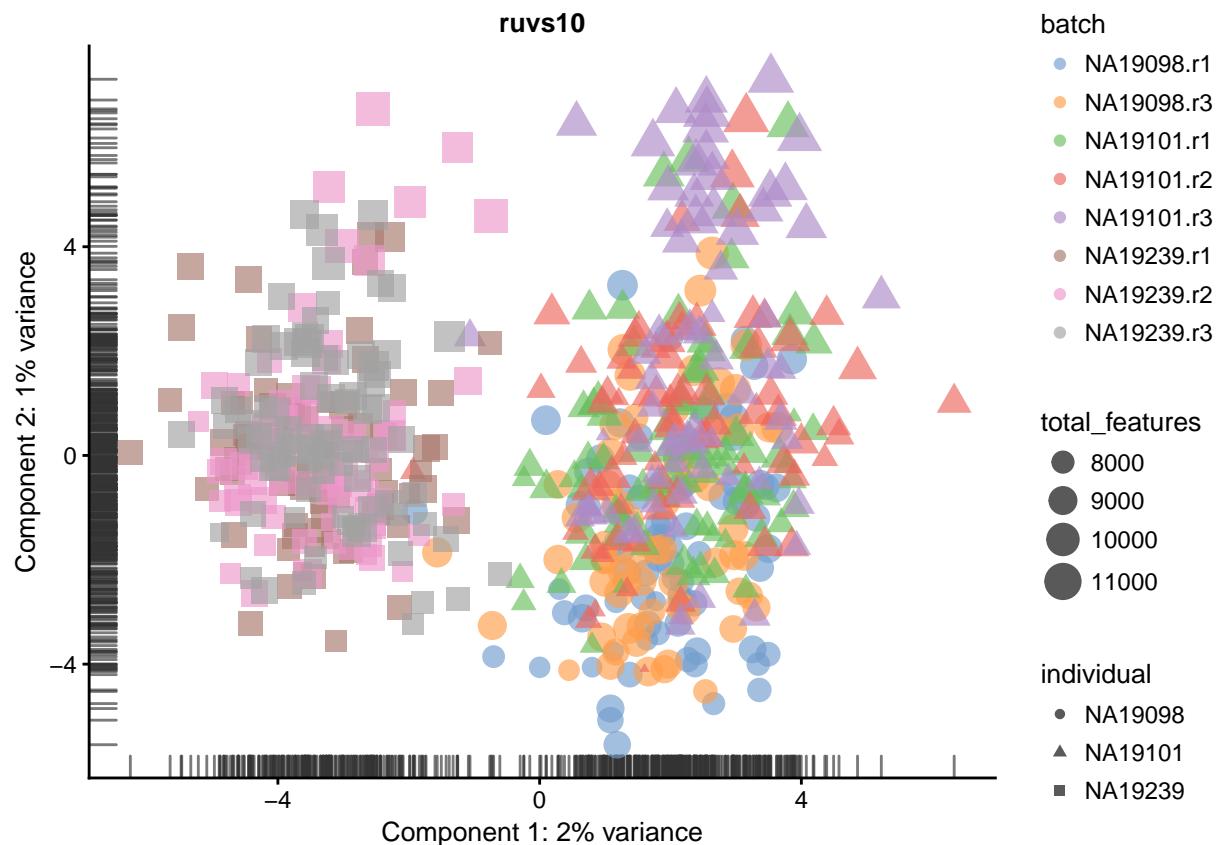
for(n in assayNames(reads.qc)) {
    print(
        plotPCA(
            reads.qc[!is.na(reads.qc), ],
            colour_by = "batch",
            size_by = "total_features",
            shape_by = "individual",
            exprs_values = n
        ) +
        ggtitle(n)
    )
}

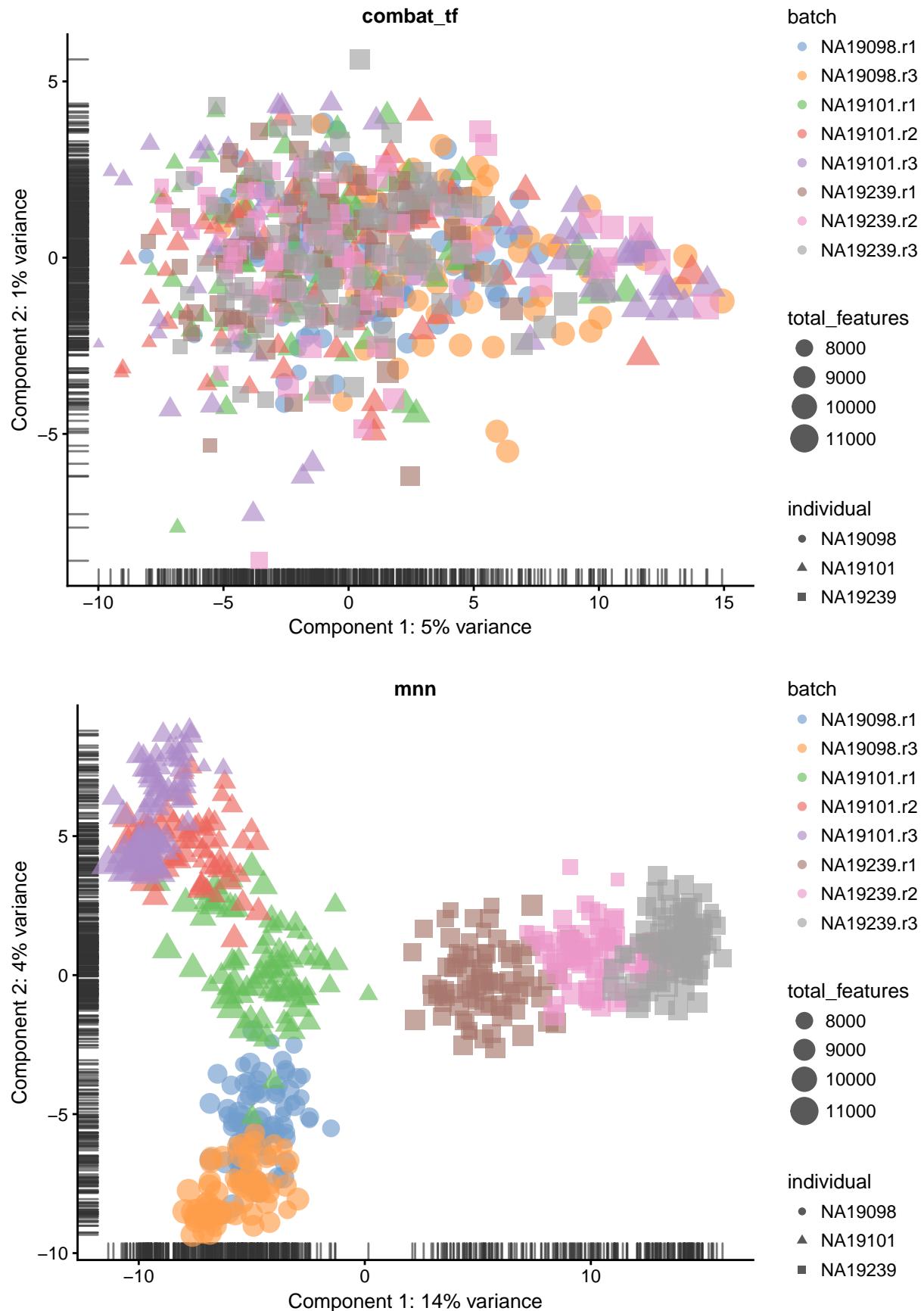
```

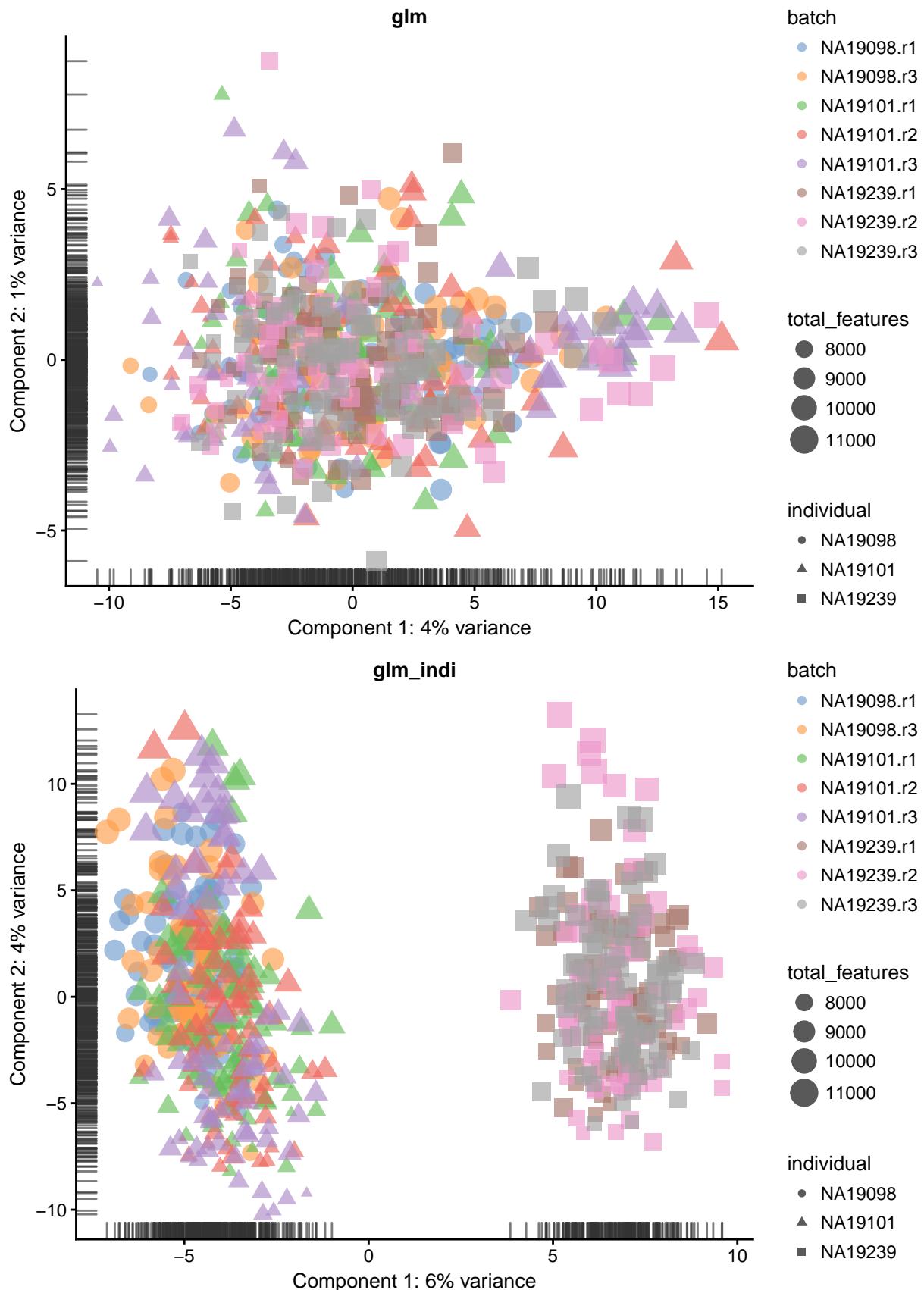








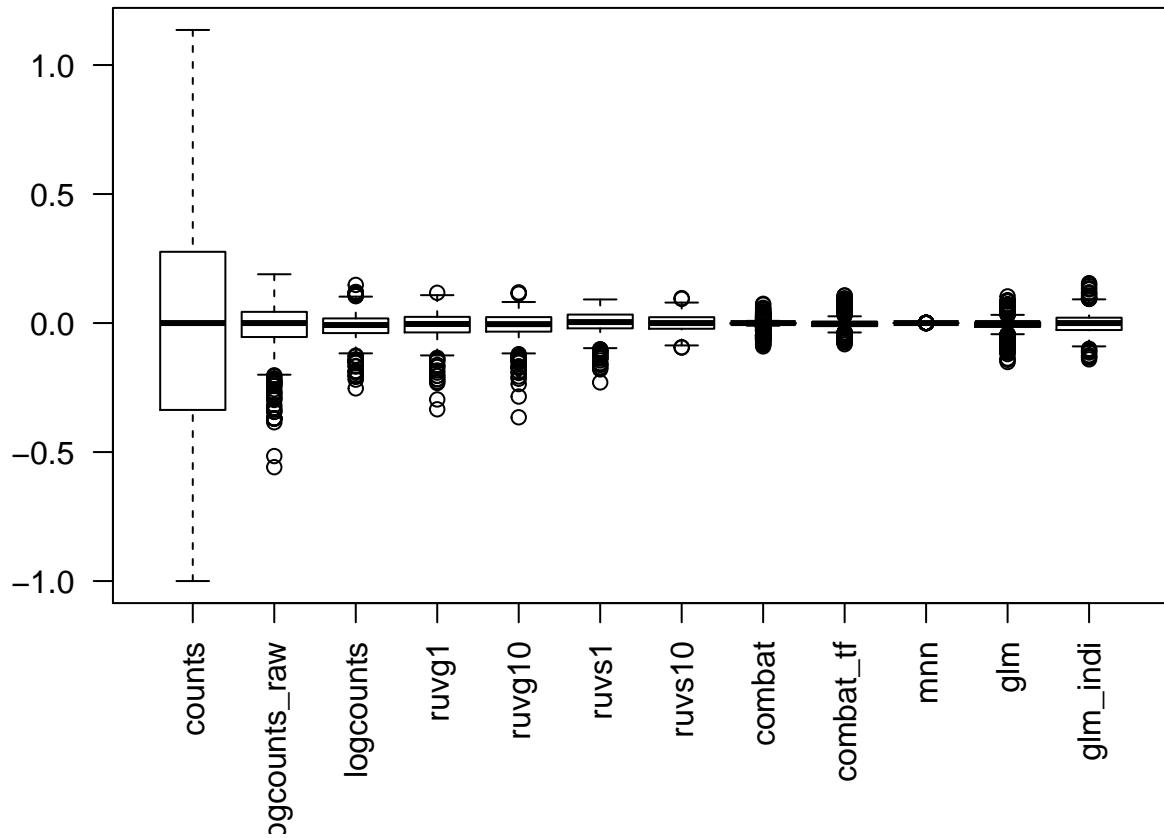




```

res <- list()
for(n in assayNames(reads.qc)) {
  res[[n]] <- suppressWarnings(calc_cell_RLE(assay(reads.qc, n), erccs))
}
par(mar=c(6,4,1,1))
boxplot(res, las=2)

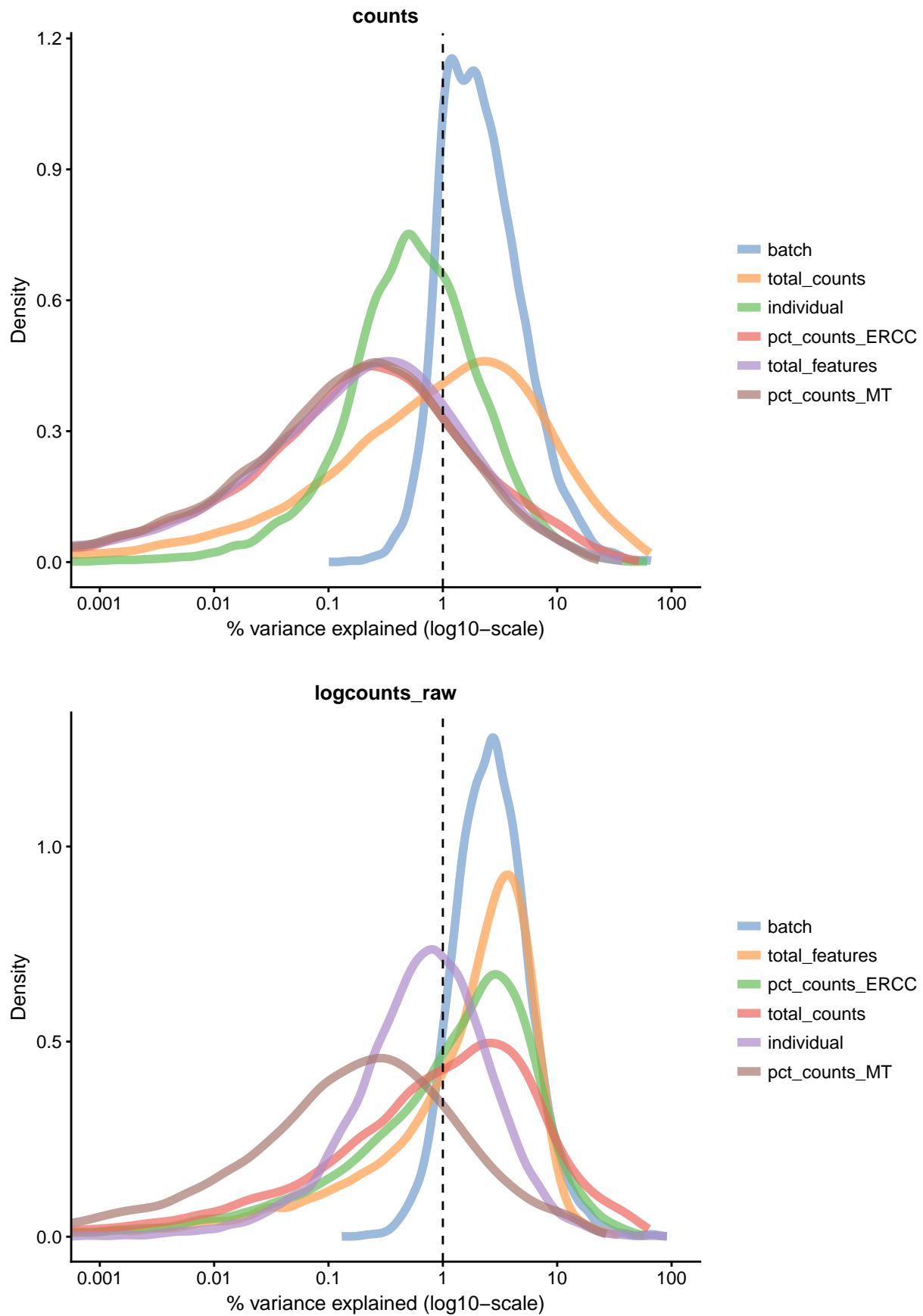
```

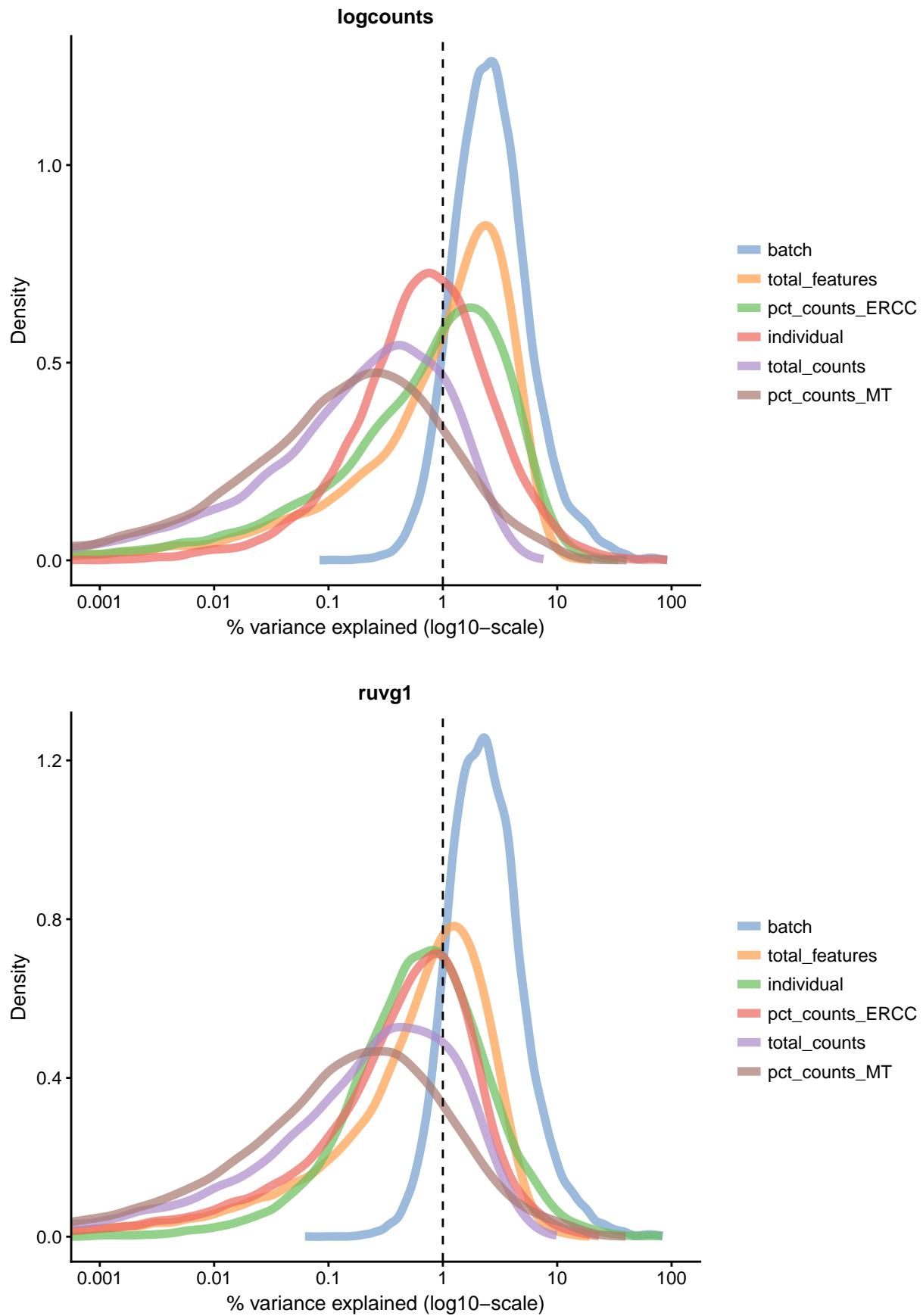


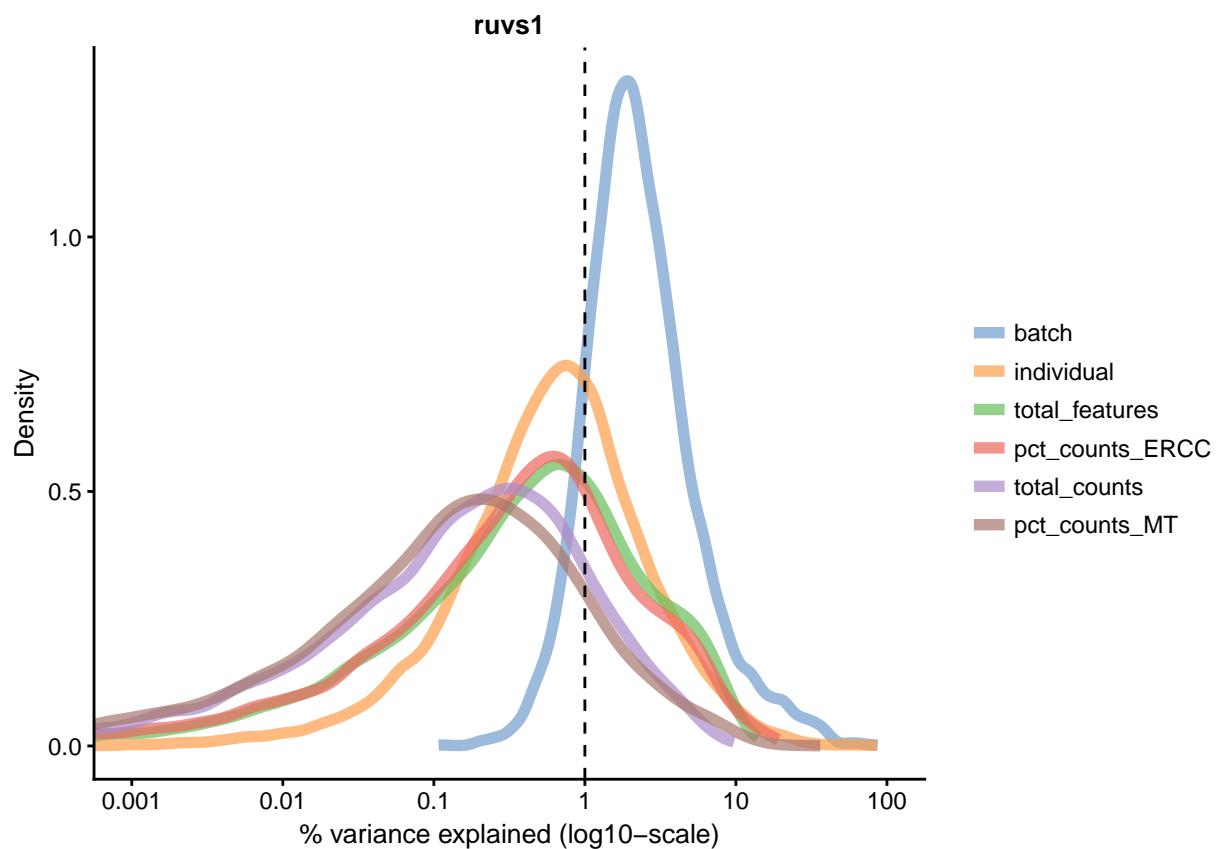
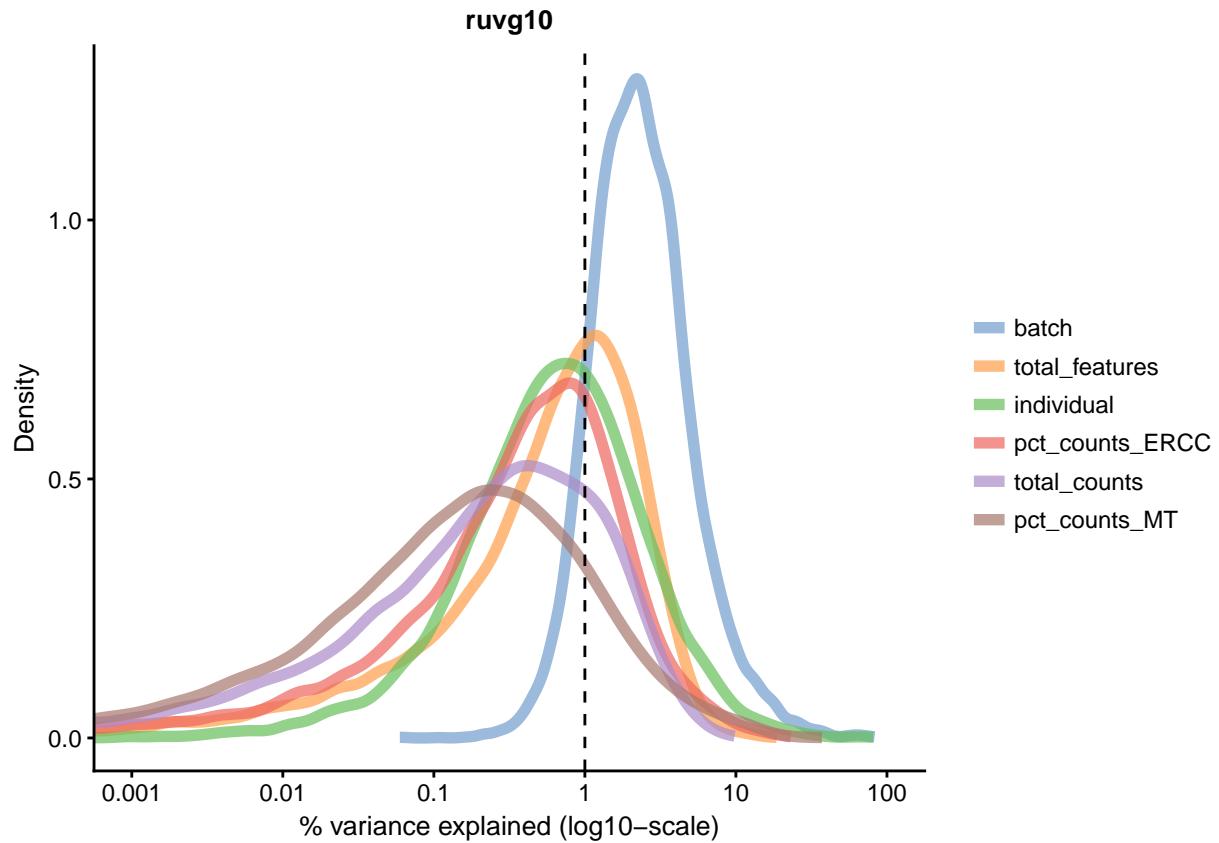
```

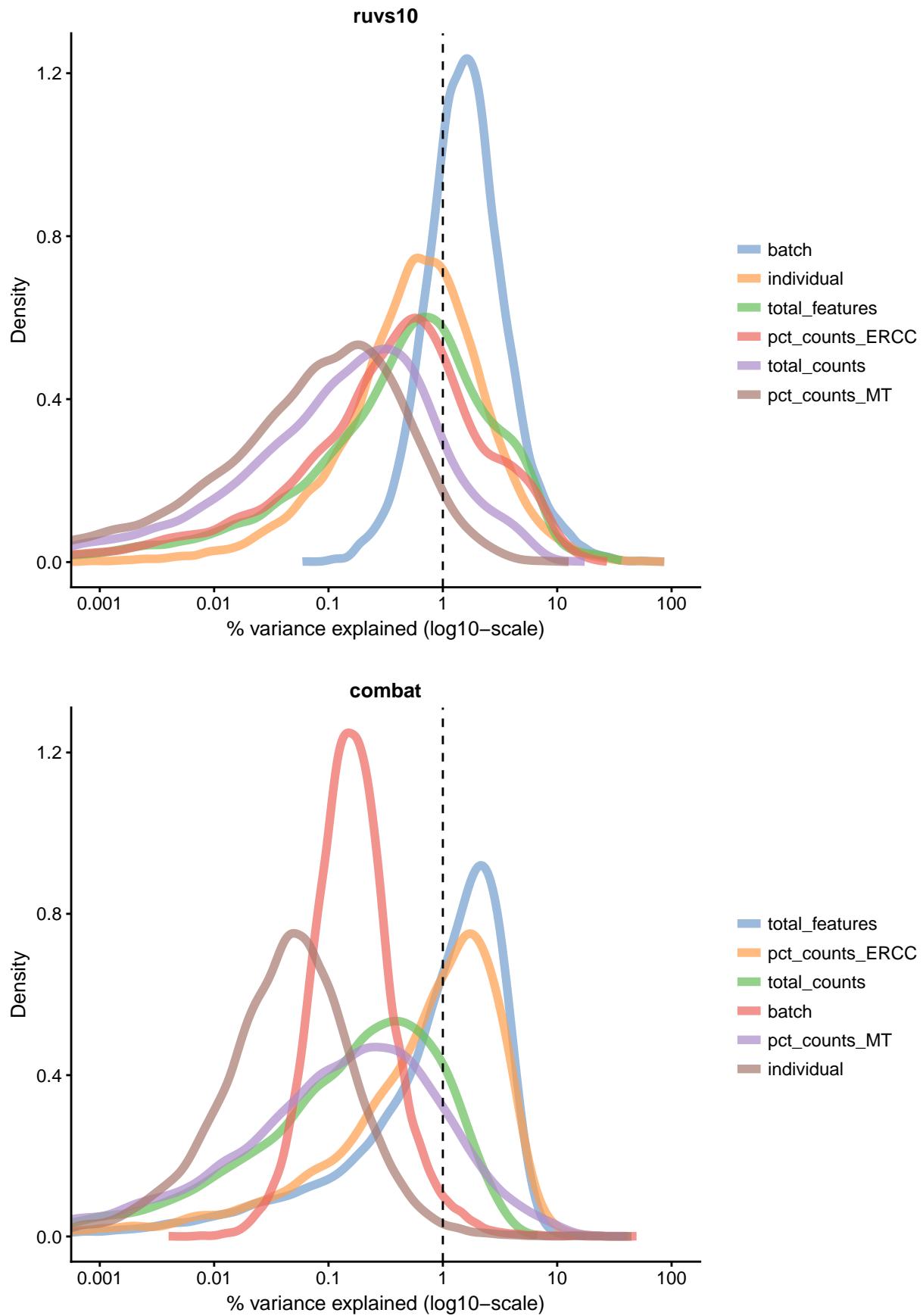
for(n in assayNames(reads.qc)) {
  print(
    plotQC(
      reads.qc[!endog_genes, ],
      type = "expl",
      exprs_values = n,
      variables = c(
        "total_features",
        "total_counts",
        "batch",
        "individual",
        "pct_counts_ERCC",
        "pct_counts_MT"
      )
    ) +
    ggttitle(n)
  )
}

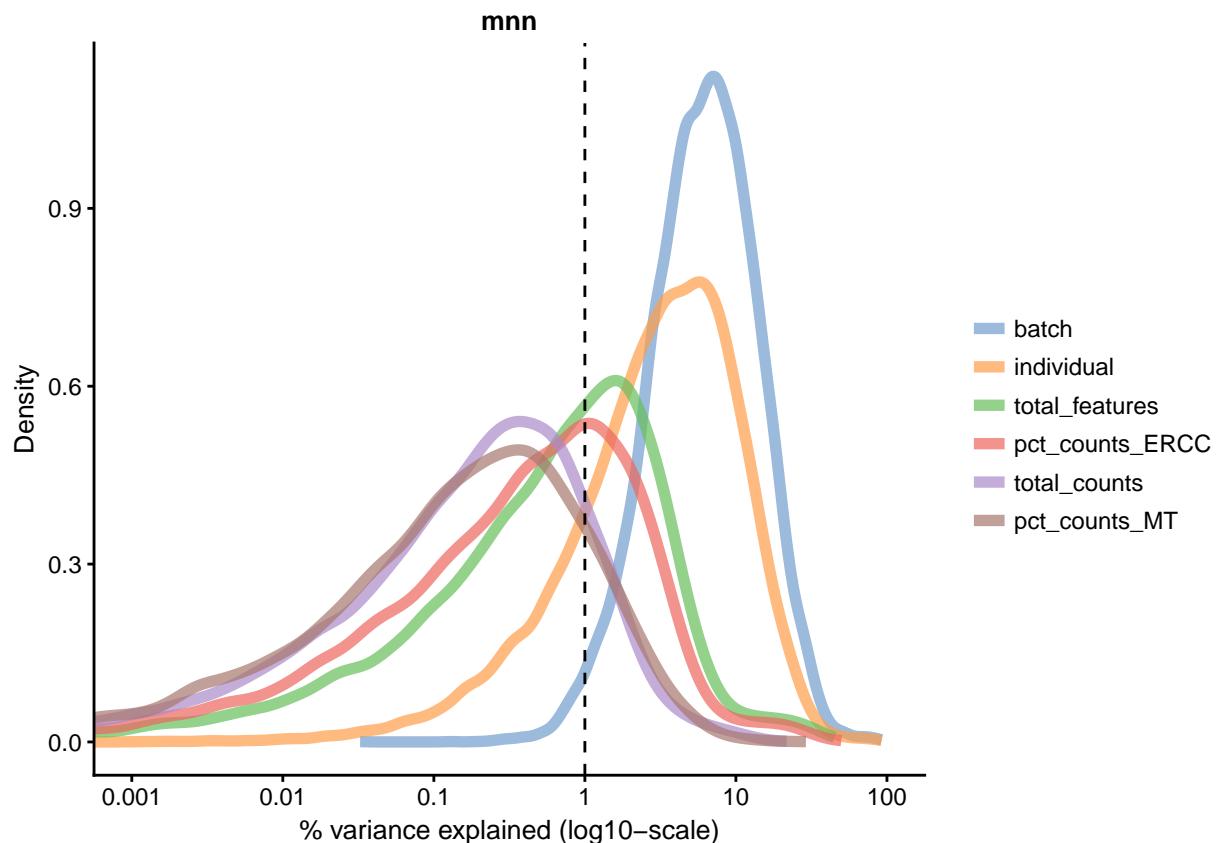
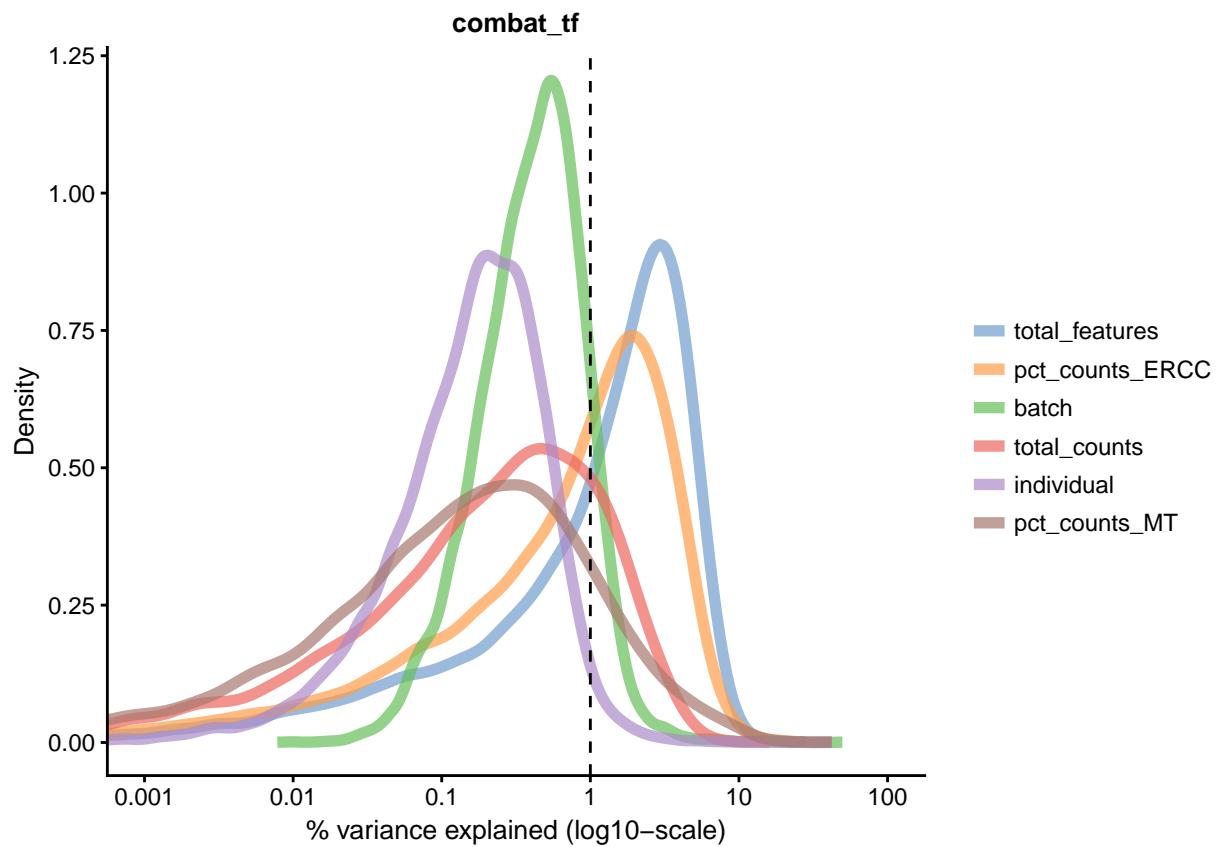
```

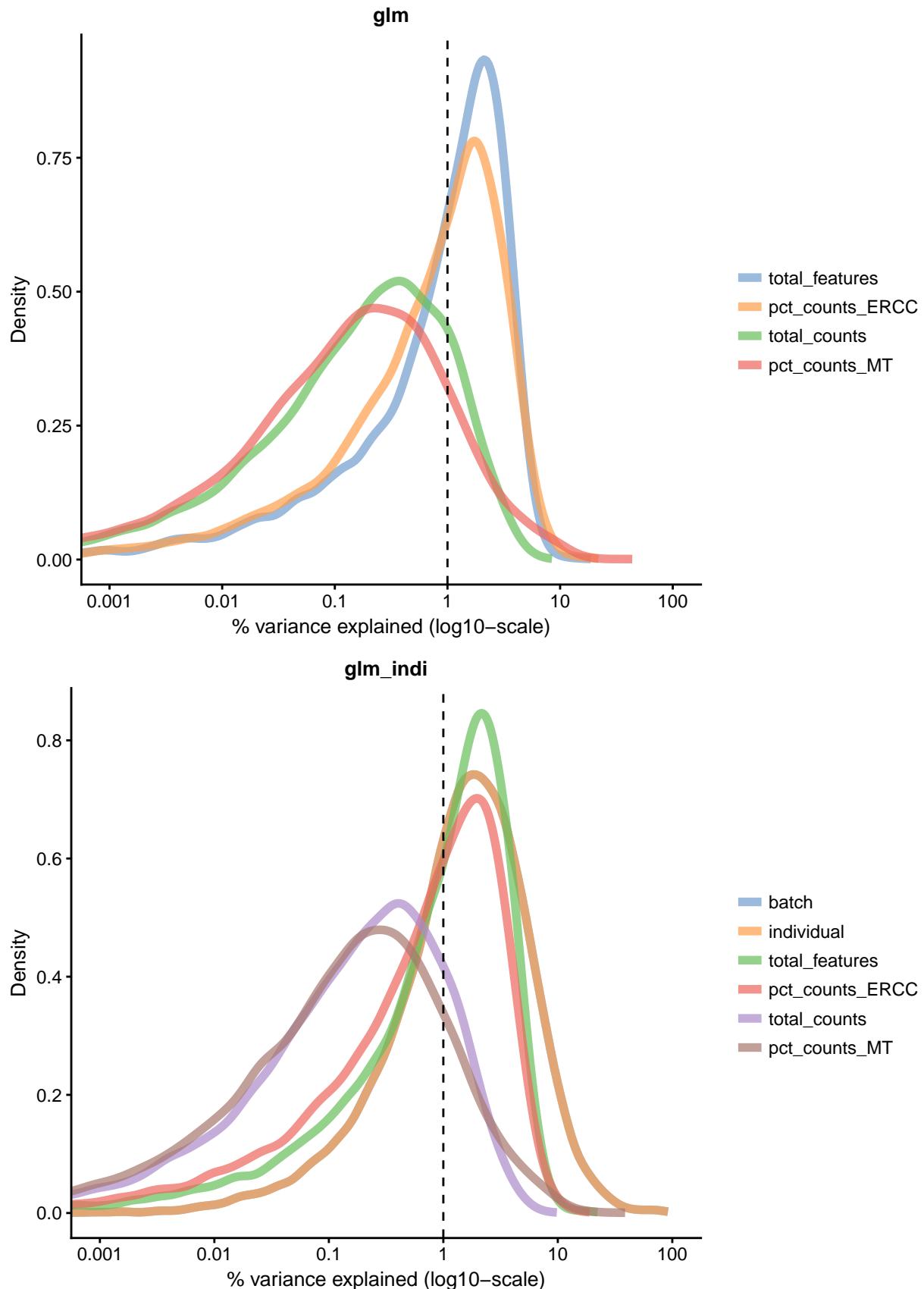












```

compare_kBET_results <- function(sce){
  indiv <- unique(sce$individual)
  norms <- assayNames(sce) # Get all normalizations
  results <- list()
  for (i in indiv){
    for (j in norms){
      tmp <- kBET(
        df = t(assay(sce[,sce$individual== i], j)),
        batch = sce$batch[sce$individual==i],
        heuristic = TRUE,
        verbose = FALSE,
        addTest = FALSE,
        plot = FALSE)
      results[[i]][[j]] <- tmp$summary$kBET.observed[1]
    }
  }
  return(as.data.frame(results))
}

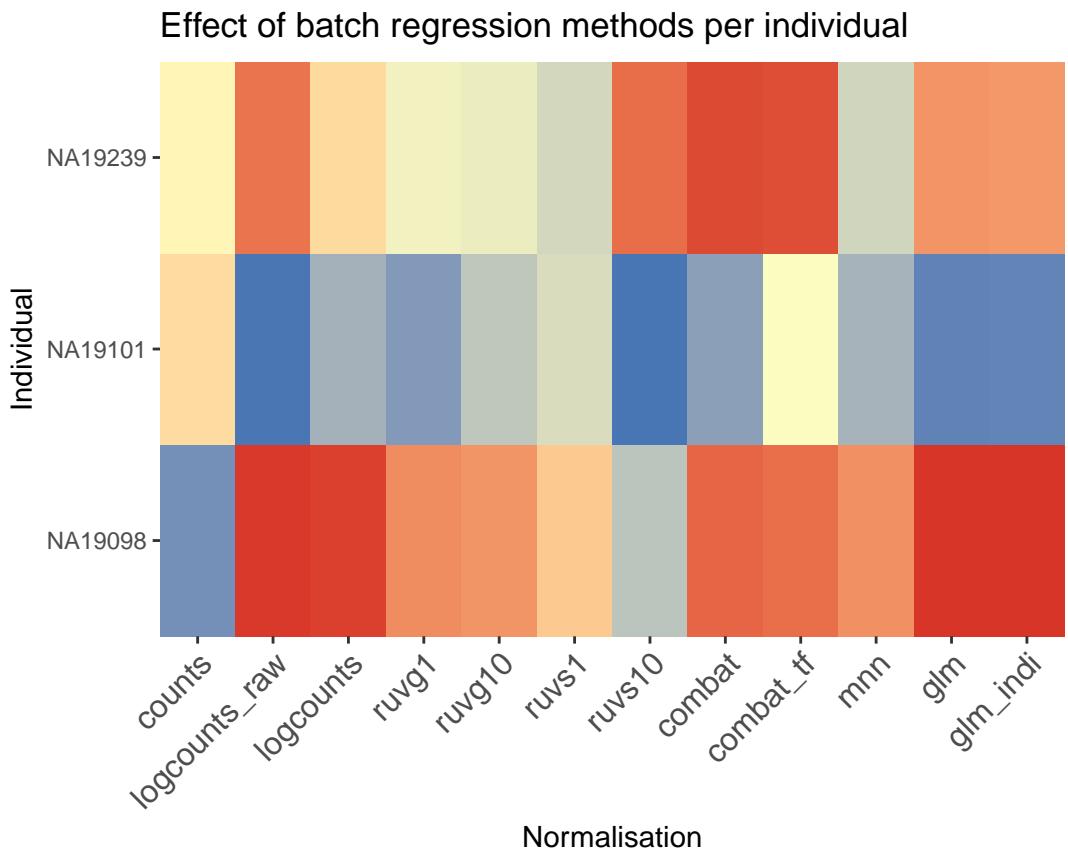
eff_debatching <- compare_kBET_results(reads.qc)

require("reshape2")
require("RColorBrewer")
# Plot results
dod <- melt(as.matrix(eff_debatching), value.name = "kBET")
colnames(dod)[1:2] <- c("Normalisation", "Individual")

colorset <- c('gray', brewer.pal(n = 9, "RdYlBu"))

ggplot(dod, aes(Normalisation, Individual, fill=kBET)) +
  geom_tile() +
  scale_fill_gradient2(
    na.value = "gray",
    low = colorset[2],
    mid=colorset[6],
    high = colorset[10],
    midpoint = 0.5, limit = c(0,1)) +
  scale_x_discrete(expand = c(0, 0)) +
  scale_y_discrete(expand = c(0, 0)) +
  theme(
    axis.text.x = element_text(
      angle = 45,
      vjust = 1,
      size = 12,
      hjust = 1
    )
  ) +
  ggtitle("Effect of batch regression methods per individual")

```



```
## R version 3.4.3 (2017-11-30)
## Platform: x86_64-pc-linux-gnu (64-bit)
## Running under: Debian GNU/Linux 9 (stretch)
##
## Matrix products: default
## BLAS: /usr/lib/openblas-base/libblas.so.3
## LAPACK: /usr/lib/libopenblas-p-r0.2.19.so
##
## locale:
## [1] LC_CTYPE=en_US.UTF-8          LC_NUMERIC=C
## [3] LC_TIME=en_US.UTF-8          LC_COLLATE=en_US.UTF-8
## [5] LC_MONETARY=en_US.UTF-8       LC_MESSAGES=C
## [7] LC_PAPER=en_US.UTF-8          LC_NAME=C
## [9] LC_ADDRESS=C                  LC_TELEPHONE=C
## [11] LC_MEASUREMENT=en_US.UTF-8    LC_IDENTIFICATION=C
##
## attached base packages:
## [1] stats4     parallel   methods   stats      graphics  grDevices utils
## [8] datasets   base
##
## other attached packages:
## [1] RColorBrewer_1.1-2      reshape2_1.4.3
## [3] sva_3.26.0              genefilter_1.60.0
## [5] mgcv_1.8-23             nlme_3.1-129
## [7] kBET_0.99.5              scran_1.6.7
## [9] scater_1.6.2             SingleCellExperiment_1.0.0
## [11] ggplot2_2.2.1            RUVSeq_1.12.0
```

```

## [13] edgeR_3.20.8           limma_3.34.8
## [15] EDASeq_2.12.0          ShortRead_1.36.0
## [17] GenomicAlignments_1.14.1 SummarizedExperiment_1.8.1
## [19] DelayedArray_0.4.1     matrixStats_0.53.0
## [21] Rsamtools_1.30.0       GenomicRanges_1.30.1
## [23] GenomeInfoDb_1.14.0    Biostrings_2.46.0
## [25] XVector_0.18.0        IRanges_2.12.0
## [27] S4Vectors_0.16.0      BiocParallel_1.12.0
## [29] Biobase_2.38.0        BiocGenerics_0.24.0
## [31] scRNA.seq.funcs_0.1.0 knitr_1.19
##
## loaded via a namespace (and not attached):
## [1] Rtsne_0.13              ggbbeeswarm_0.6.0   colorspace_1.3-2
## [4] rjson_0.2.15             hwriter_1.3.2     dynamicTreeCut_1.63-1
## [7] rprojroot_1.3-2          DT_0.4          bit64_0.9-7
## [10] AnnotationDbi_1.40.0    splines_3.4.3   R.methodsS3_1.7.1
## [13] tximport_1.6.0           DESeq_1.30.0    geneplotter_1.56.0
## [16] annotate_1.56.1          cluster_2.0.6   R.oo_1.21.0
## [19] shinydashboard_0.6.1    shiny_1.0.5     compiler_3.4.3
## [22] httr_1.3.1              backports_1.1.2 assertthat_0.2.0
## [25] Matrix_1.2-7.1          lazyeval_0.2.1  htmltools_0.3.6
## [28] prettyunits_1.0.2       tools_3.4.3    igraph_1.1.2
## [31] bindrcpp_0.2              gtable_0.2.0   glue_1.2.0
## [34] GenomeInfoDbData_1.0.0  dplyr_0.7.4    Rcpp_0.12.15
## [37] rtracklayer_1.38.3      xfun_0.1       stringr_1.2.0
## [40] mime_0.5                hypergeo_1.2-13 statmod_1.4.30
## [43] XML_3.98-1.9            zoo_1.8-1     zlibbioc_1.24.0
## [46] MASS_7.3-45              scales_0.5.0   aroma.light_3.8.0
## [49] rhdf5_2.22.0             yaml_2.1.16   memoise_1.1.0
## [52] gridExtra_2.3             biomaRt_2.34.2 latticeExtra_0.6-28
## [55] stringi_1.1.6            RSQLite_2.0    RMySQL_0.10.13
## [58] orthopolynom_1.0-5      GenomicFeatures_1.30.3 conffrac_1.1-11
## [61] rlang_0.1.6              pkgconfig_2.0.1 moments_0.14
## [64] bitops_1.0-6              evaluate_0.10.1 lattice_0.20-34
## [67] bindr_0.1                 labeling_0.3   htmlwidgets_1.0
## [70] cowplot_0.9.2            bit_1.1-12    deSolve_1.20
## [73] plyr_1.8.4               magrittr_1.5   bookdown_0.6
## [76] R6_2.2.2                 DBI_0.7       pillar_1.1.0
## [79] survival_2.40-1          RCurl_1.95-4.10 tibble_1.4.2
## [82] rmarkdown_1.8              viridis_0.5.0  progress_1.1.2
## [85] locfit_1.5-9.1            grid_3.4.3   data.table_1.10.4-3
## [88] FNN_1.1                  blob_1.1.0    digest_0.6.15
## [91] xtable_1.8-2              httpuv_1.3.5  elliptic_1.3-7
## [94] R.utils_2.6.0              munsell_0.4.3 beeswarm_0.2.3
## [97] viridisLite_0.3.0         viper_0.4.5

```

# Chapter 8

## Biological Analysis

### 8.1 Clustering Introduction

Once we have normalized the data and removed confounders we can carry out analyses that are relevant to the biological questions at hand. The exact nature of the analysis depends on the dataset. Nevertheless, there are a few aspects that are useful in a wide range of contexts and we will be discussing some of them in the next few chapters. We will start with the clustering of scRNA-seq data.

#### 8.1.1 Introduction

One of the most promising applications of scRNA-seq is *de novo* discovery and annotation of cell-types based on transcription profiles. Computationally, this is a hard problem as it amounts to **unsupervised clustering**. That is, we need to identify groups of cells based on the similarities of the transcriptomes without any prior knowledge of the labels. Moreover, in most situations we do not even know the number of clusters *a priori*. The problem is made even more challenging due to the high level of noise (both technical and biological) and the large number of dimensions (i.e. genes).

#### 8.1.2 Dimensionality reductions

When working with large datasets, it can often be beneficial to apply some sort of dimensionality reduction method. By projecting the data onto a lower-dimensional sub-space, one is often able to significantly reduce the amount of noise. An additional benefit is that it is typically much easier to visualize the data in a 2 or 3-dimensional subspace. We have already discussed PCA (chapter 7.3.2) and t-SNE (chapter 7.3.2).

#### 8.1.3 Clustering methods

**Unsupervised clustering** is useful in many different applications and it has been widely studied in machine learning. Some of the most popular approaches are **hierarchical clustering**, **k-means clustering** and **graph-based clustering**.

##### 8.1.3.1 Hierarchical clustering

In hierarchical clustering, one can use either a bottom-up or a top-down approach. In the former case, each cell is initially assigned to its own cluster and pairs of clusters are subsequently merged to create a hierarchy:

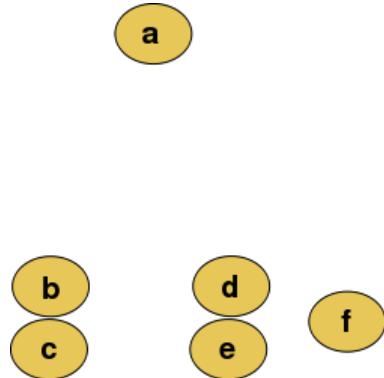


Figure 8.1: Raw data

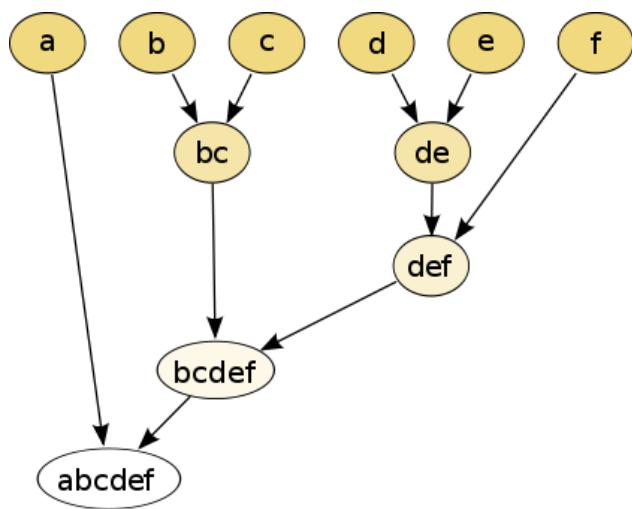


Figure 8.2: The hierarchical clustering dendrogram

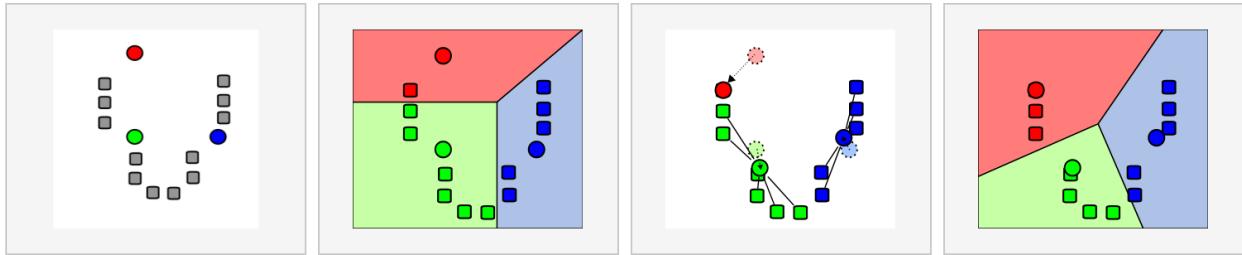


Figure 8.3: Schematic representation of the k-means clustering

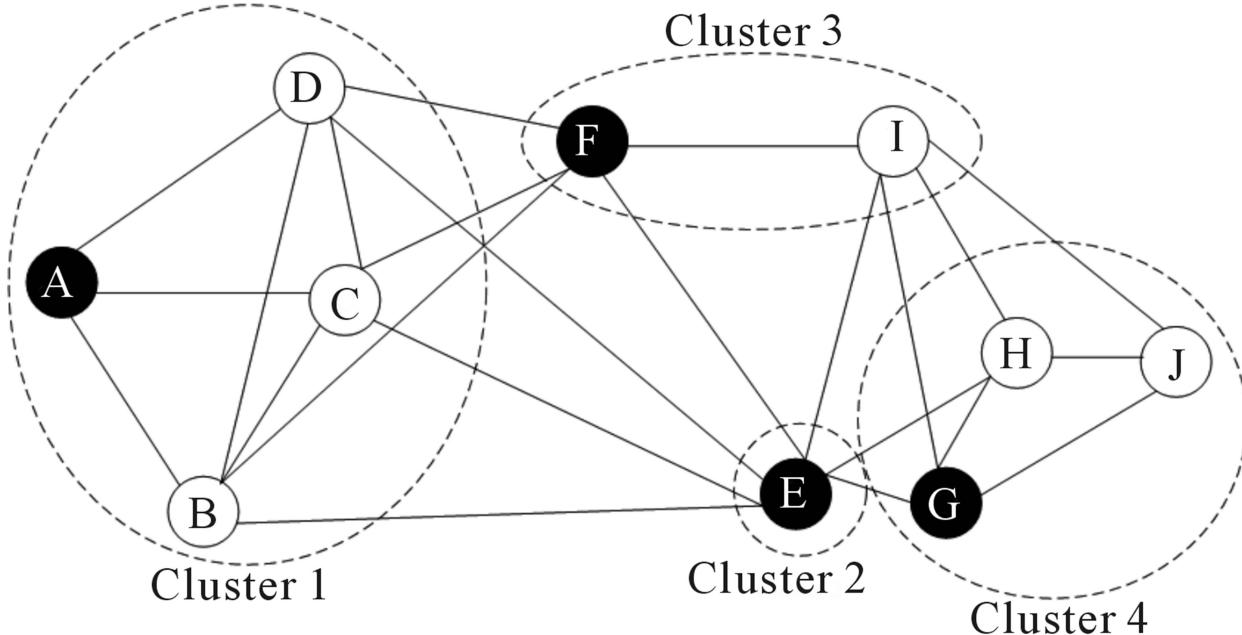


Figure 8.4: Schematic representation of the graph network

With a top-down strategy, one instead starts with all observations in one cluster and then recursively split each cluster to form a hierarchy. One of the advantages of this strategy is that the method is deterministic.

### 8.1.3.2 k-means

In  $k$ -means clustering, the goal is to partition  $N$  cells into  $k$  different clusters. In an iterative manner, cluster centers are assigned and each cell is assigned to its nearest cluster:

Most methods for scRNA-seq analysis includes a  $k$ -means step at some point.

### 8.1.3.3 Graph-based methods

Over the last two decades there has been a lot of interest in analyzing networks in various domains. One goal is to identify groups or modules of nodes in a network.

Some of these methods can be applied to scRNA-seq data by building a graph where each node represents a cell. Note that constructing the graph and assigning weights to the edges is not trivial. One advantage of graph-based methods is that some of them are very efficient and can be applied to networks containing millions of nodes.

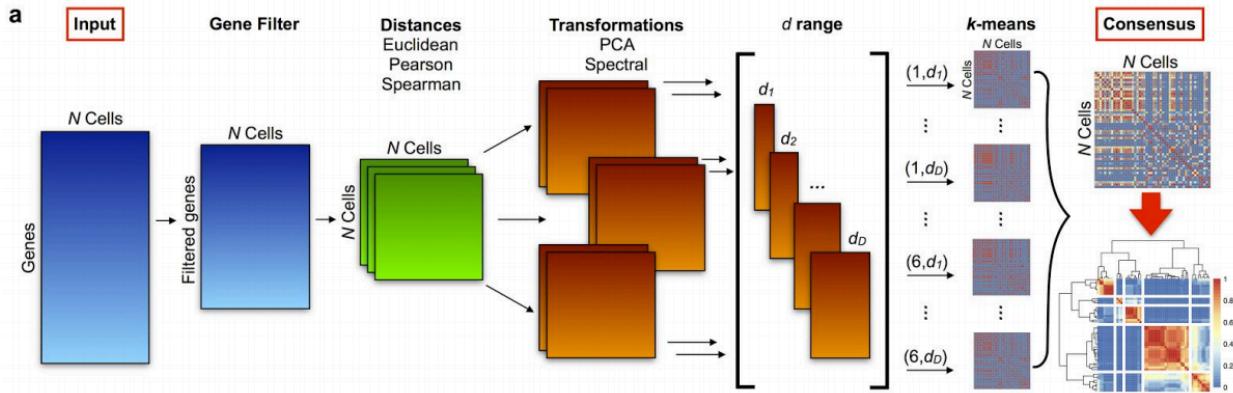


Figure 8.5: SC3 pipeline

### 8.1.4 Challenges in clustering

- What is the number of clusters  $k$ ?
- What is a cell type?
- **Scalability:** in the last few years the number of cells in scRNA-seq experiments has grown by several orders of magnitude from  $\sim 10^2$  to  $\sim 10^6$
- Tools are not user-friendly

### 8.1.5 Tools for scRNA-seq data

#### 8.1.5.1 SINCERA

- SINCERA (Guo et al., 2015) is based on hierarchical clustering
- Data is converted to  $z$ -scores before clustering
- Identify  $k$  by finding the first singleton cluster in the hierarchy

#### 8.1.5.2 pcaReduce

pcaReduce (žurauskienė and Yau, 2016) combines PCA,  $k$ -means and “iterative” hierarchical clustering. Starting from a large number of clusters pcaReduce iteratively merges similar clusters; after each merging event it removes the principle component explaining the least variance in the data.

#### 8.1.5.3 SC3

- SC3 (Kiselev et al., 2017) is based on PCA and spectral dimensionality reductions
- Utilises  $k$ -means
- Additionally performs the consensus clustering

#### 8.1.5.4 tSNE + k-means

- Based on **tSNE** maps
- Utilises  $k$ -means

### 8.1.5.5 SNN-Cliq

**SNN-Cliq** (Xu and Su, 2015) is a graph-based method. First the method identifies the k-nearest-neighbours of each cell according to the *distance* measure. This is used to calculate the number of Shared Nearest Neighbours (SNN) between each pair of cells. A graph is built by placing an edge between two cells If they have at least one SNN. Clusters are defined as groups of cells with many edges between them using a “clique” method. SNN-Cliq requires several parameters to be defined manually.

### 8.1.5.6 Seurat clustering

**Seurat** clustering is based on a *community detection* approach similar to **SNN-Cliq** and to one previously proposed for analyzing CyTOF data (Levine et al., 2015). Since **Seurat** has become more like an all-in-one tool for scRNA-seq data analysis we dedicate a separate chapter to discuss it in more details (chapter 9).

## 8.1.6 Comparing clustering

To compare two sets of clustering labels we can use adjusted Rand index. The index is a measure of the similarity between two data clusterings. Values of the adjusted Rand index lie in [0; 1] interval, where 1 means that two clusterings are identical and 0 means the level of similarity expected by chance.

## 8.2 Clustering example

```
library(pcaMethods)
library(pcaReduce)
library(SC3)
library(scater)
library(SingleCellExperiment)
library(pheatmap)
library(mclust)
set.seed(1234567)
```

To illustrate clustering of scRNA-seq data, we consider the **Deng** dataset of cells from developing mouse embryo (Deng et al., 2014). We have preprocessed the dataset and created a **SingleCellExperiment** object in advance. We have also annotated the cells with the cell types identified in the original publication (it is the **cell\_type2** column in the **colData** slot).

### 8.2.1 Deng dataset

Let’s load the data and look at it:

```
deng <- readRDS("deng/deng-reads.rds")
deng

## class: SingleCellExperiment
## dim: 22431 268
## metadata(0):
## assays(2): counts logcounts
## rownames(22431): Hvcn1 Gbp7 ... Sox5 Alg11
## rowData names(10): feature_symbol is_feature_control ...
##   total_counts log10_total_counts
```

```
## colnames(268): 16cell 16cell.1 ... zy.2 zy.3
## colData names(30): cell_type2 cell_type1 ... pct_counts_ERCC
##   is_cell_control
## reducedDimNames(0):
## spikeNames(1): ERCC
```

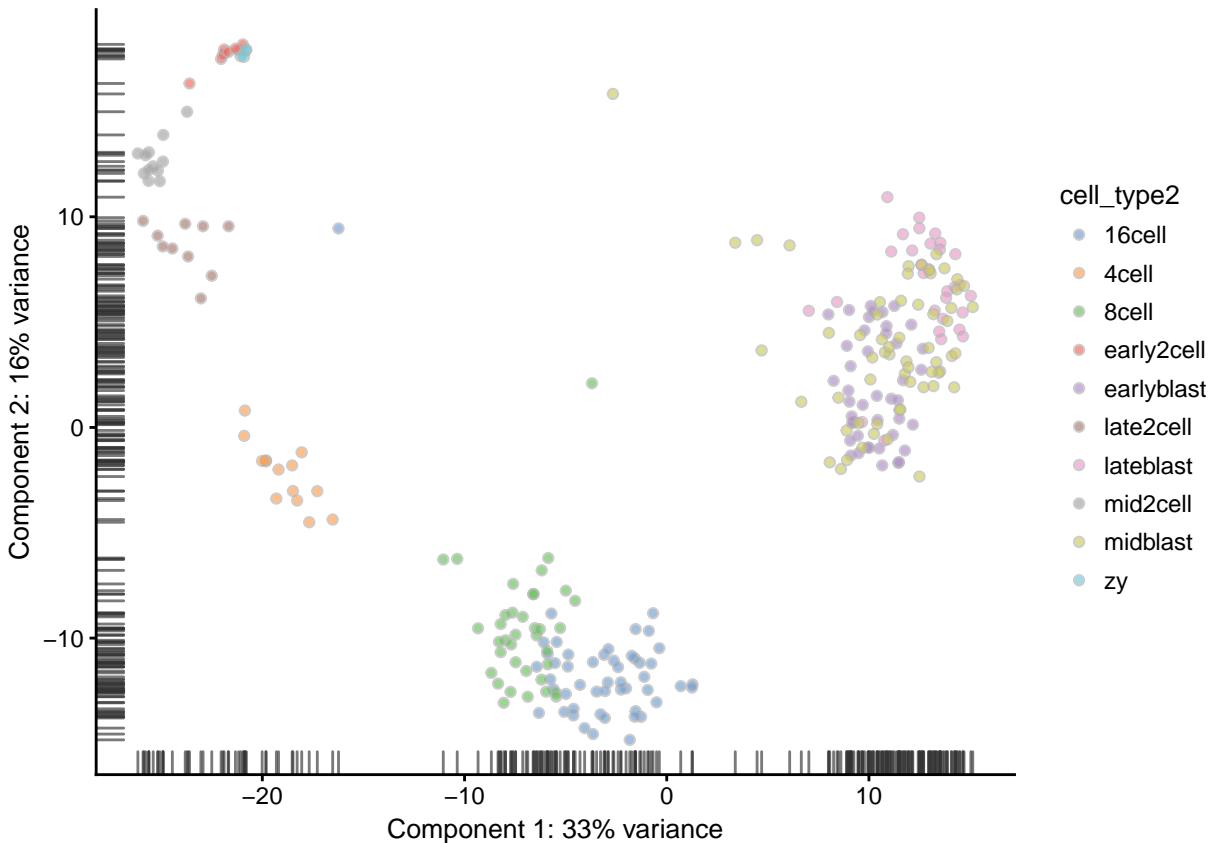
Let's look at the cell type annotation:

```
table(colData(deng)$cell_type2)
```

```
##
##      16cell      4cell      8cell early2cell earlyblast late2cell
##      50          14         37        8          43        10
## lateblast mid2cell midblast      zy
##      30          12         60        4
```

A simple PCA analysis already separates some strong cell types and provides some insights in the data structure:

```
plotPCA(deng, colour_by = "cell_type2")
```



As you can see, the early cell types separate quite well, but the three blastocyst timepoints are more difficult to distinguish.

### 8.2.2 SC3

Let's run SC3 clustering on the Deng data. The advantage of the SC3 is that it can directly ingest a SingleCellExperiment object.

Now let's image we do not know the number of clusters  $k$  (cell types). SC3 can estimate a number of clusters for you:

```
deng <- sc3_estimate_k(deng)
```

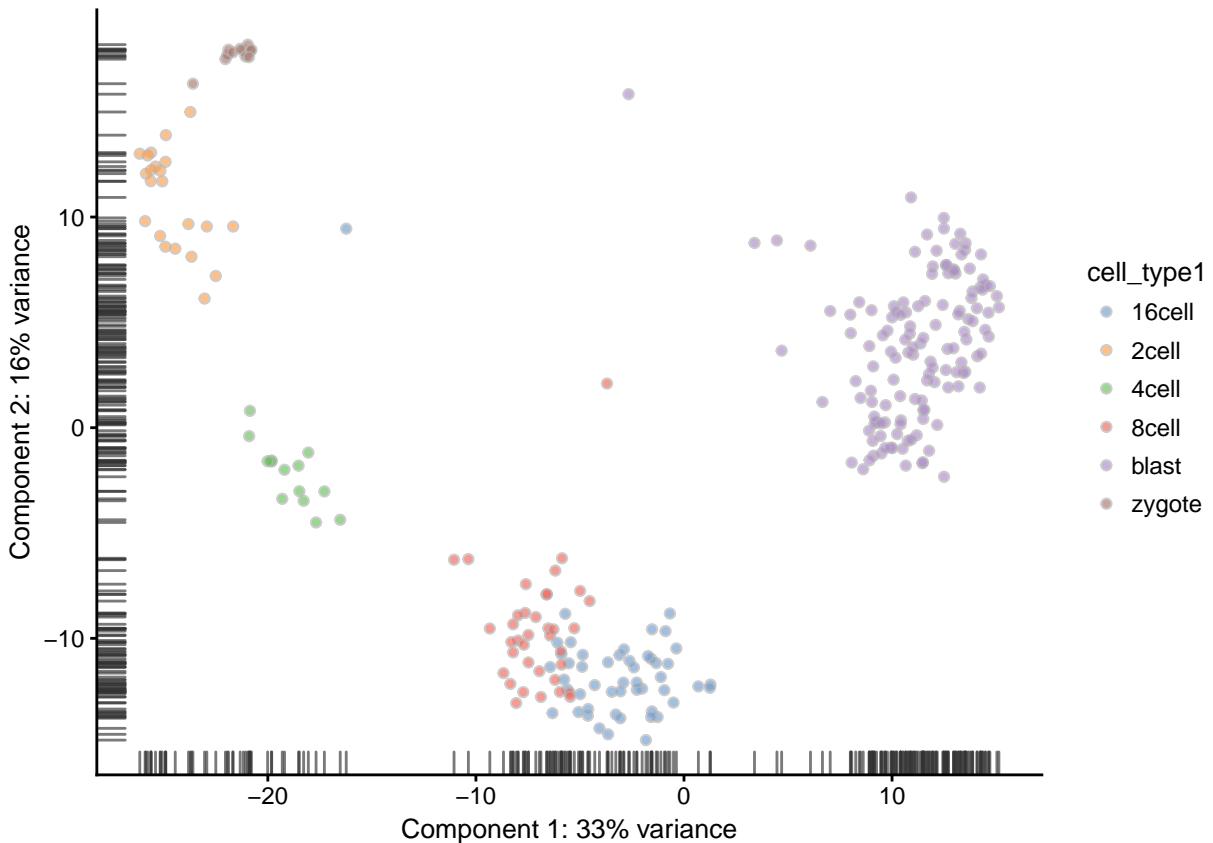
```
## Estimating k...
```

```
metadata(deng)$sc3$k_estimation
```

```
## [1] 6
```

Interestingly, the number of cell types predicted by SC3 is smaller than in the original data annotation. However, early, mid and late stages of different cell types together, we will have exactly 6 cell types. We store the merged cell types in `cell_type1` column of the `colData` slot:

```
plotPCA(deng, colour_by = "cell_type1")
```



Now we are ready to run SC3 (we also ask it to calculate biological properties of the clusters):

```
deng <- sc3(deng, ks = 10, biology = TRUE)
```

```
## Setting SC3 parameters...
```

```
## Calculating distances between the cells...
```

```
## Performing transformations and calculating eigenvectors...
```

```
## Performing k-means clustering...
```

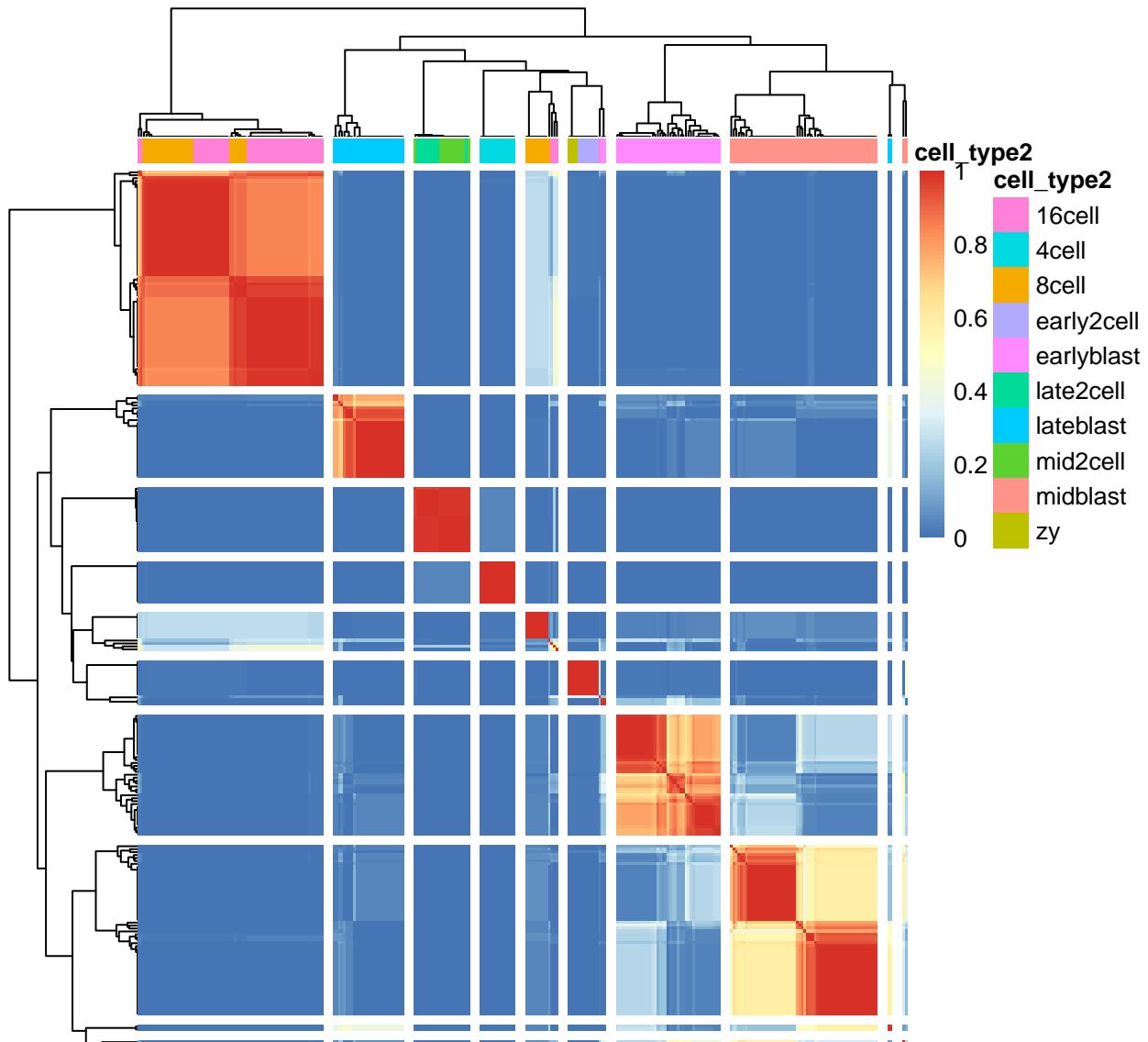
```
## Calculating consensus matrix...
```

```
## Calculating biology...
```

SC3 result consists of several different outputs (please look in (Kiselev et al., 2017) and SC3 vignette for more details). Here we show some of them:

Consensus matrix:

```
sc3_plot_consensus(deng, k = 10, show_pdata = "cell_type2")
```



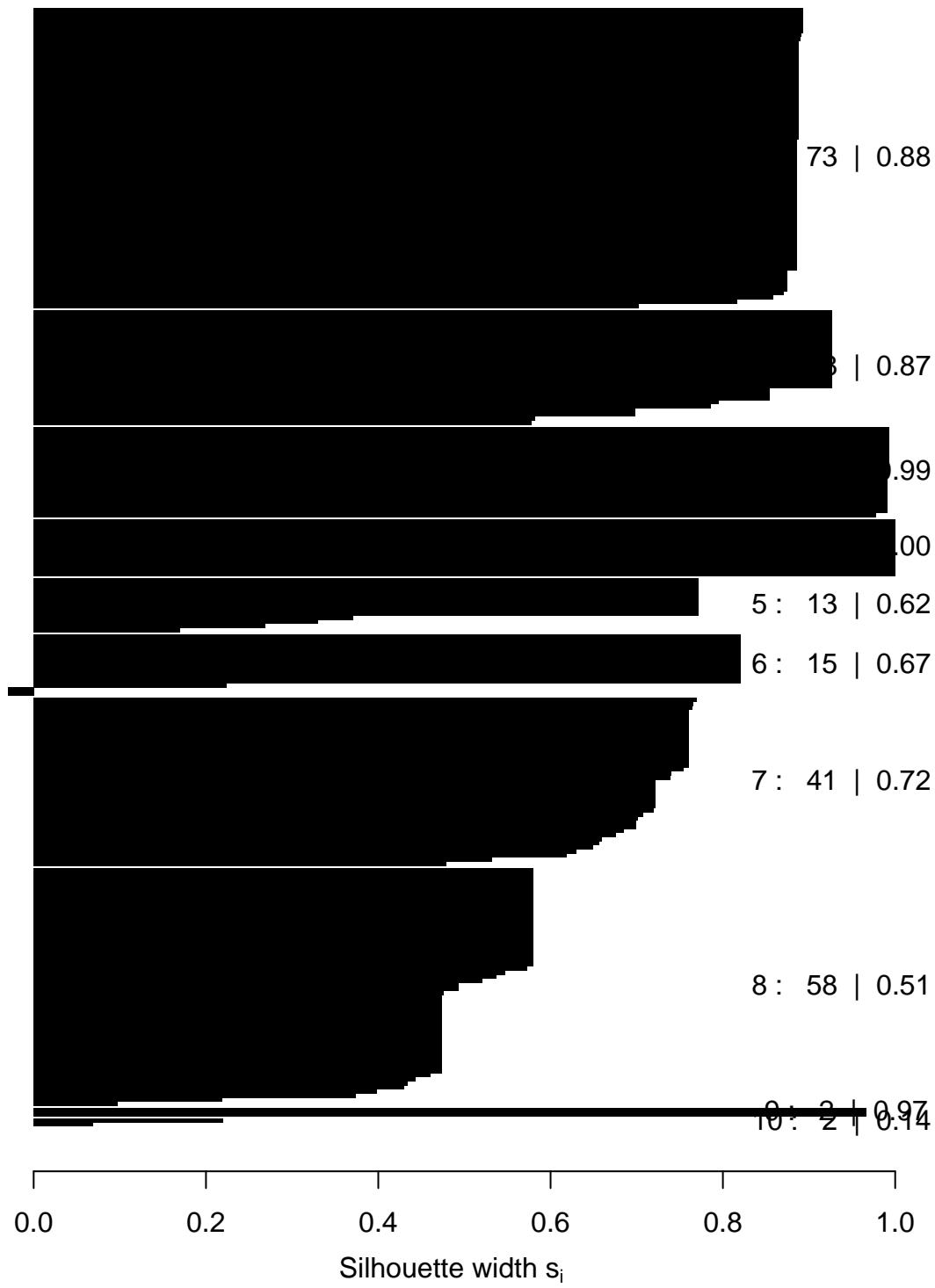
Silhouette plot:

```
sc3_plot_silhouette(deng, k = 10)
```

### Silhouette plot of (x = clusts, dist = diss)

$n = 268$

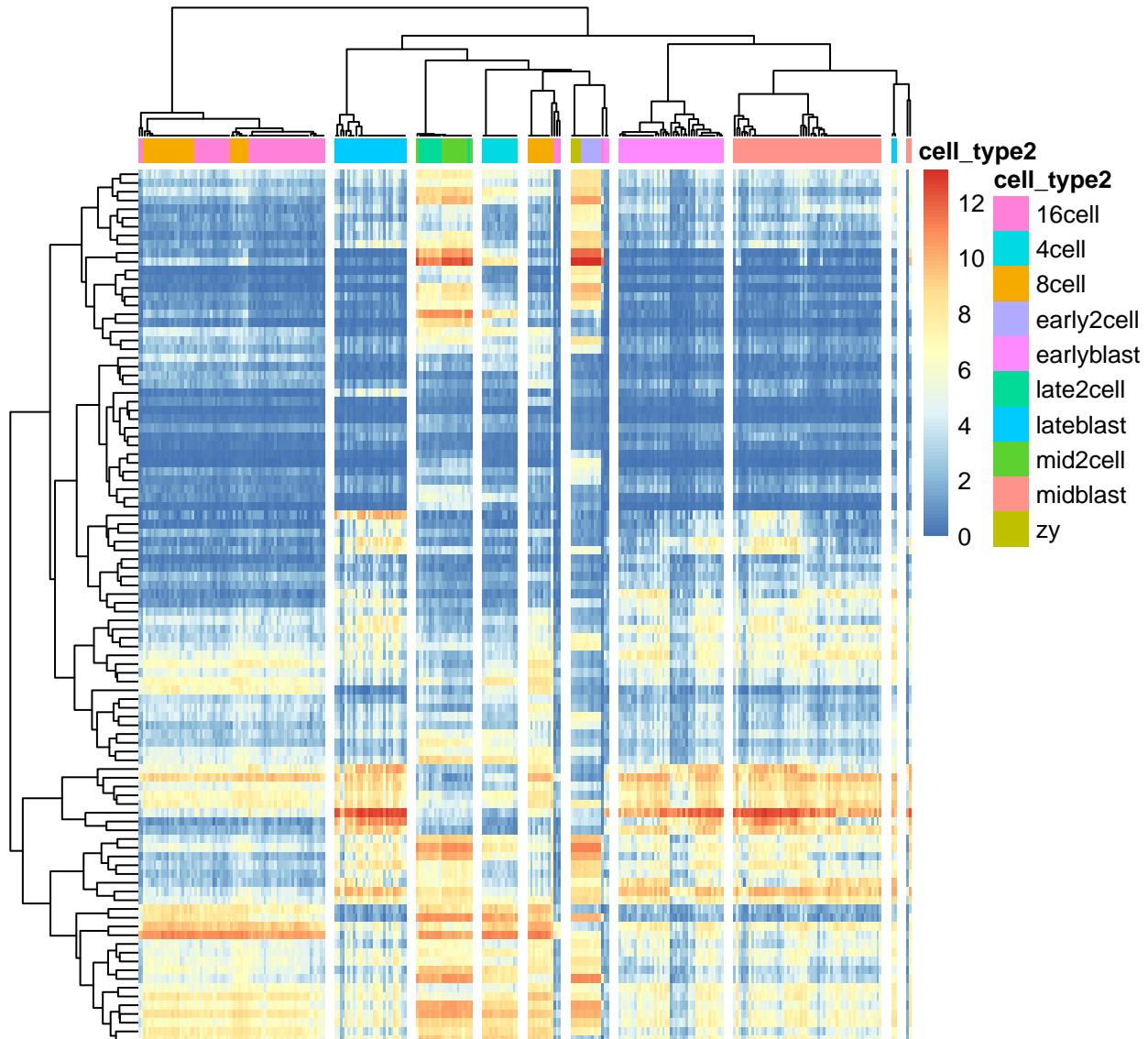
10 clusters  $C_j$   
 $j : n_j | \text{ave}_{i \in C_j} s_i$



Average silhouette width : 0.76

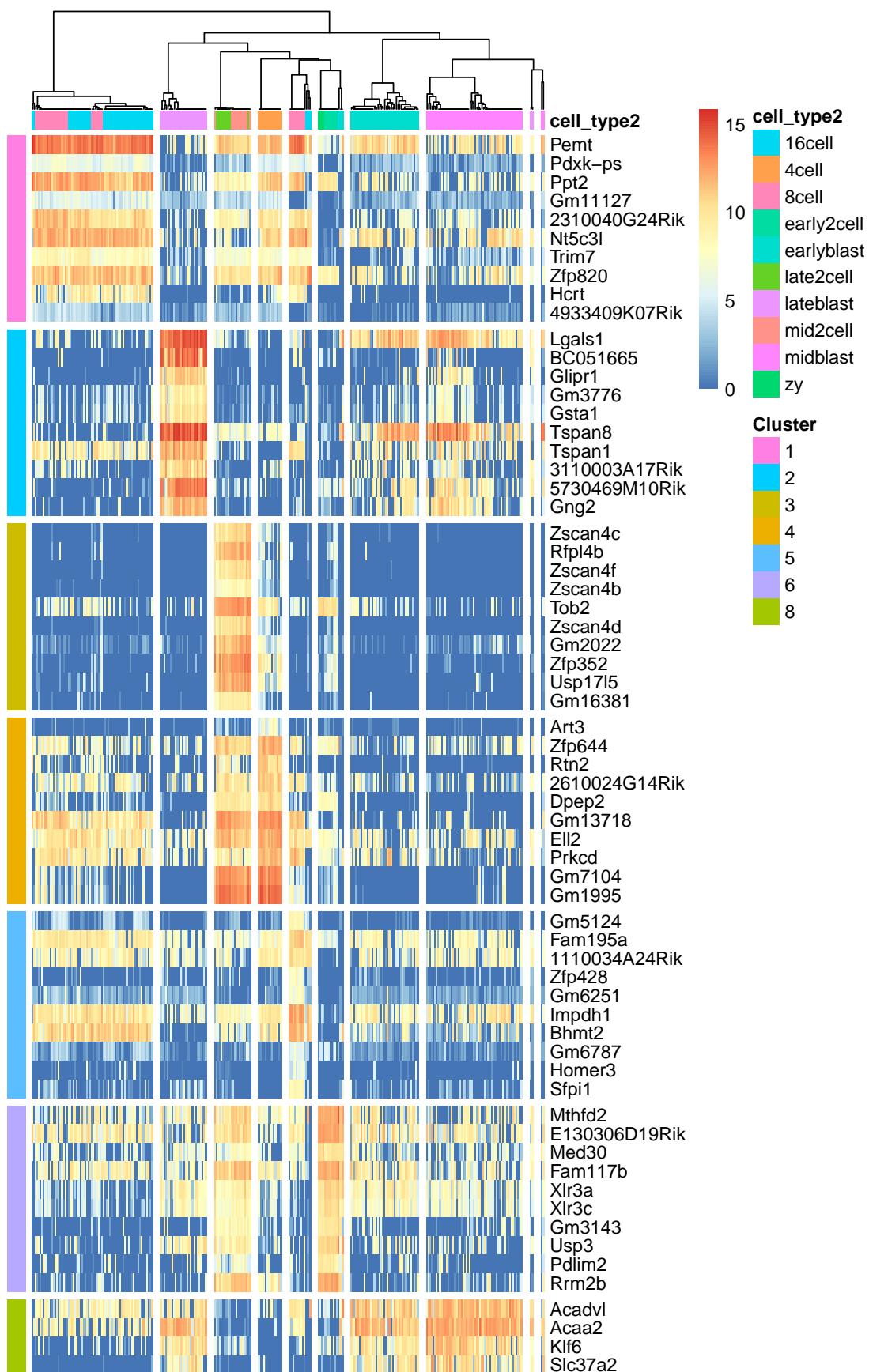
Heatmap of the expression matrix:

```
sc3_plot_expression(deng, k = 10, show_pdata = "cell_type2")
```



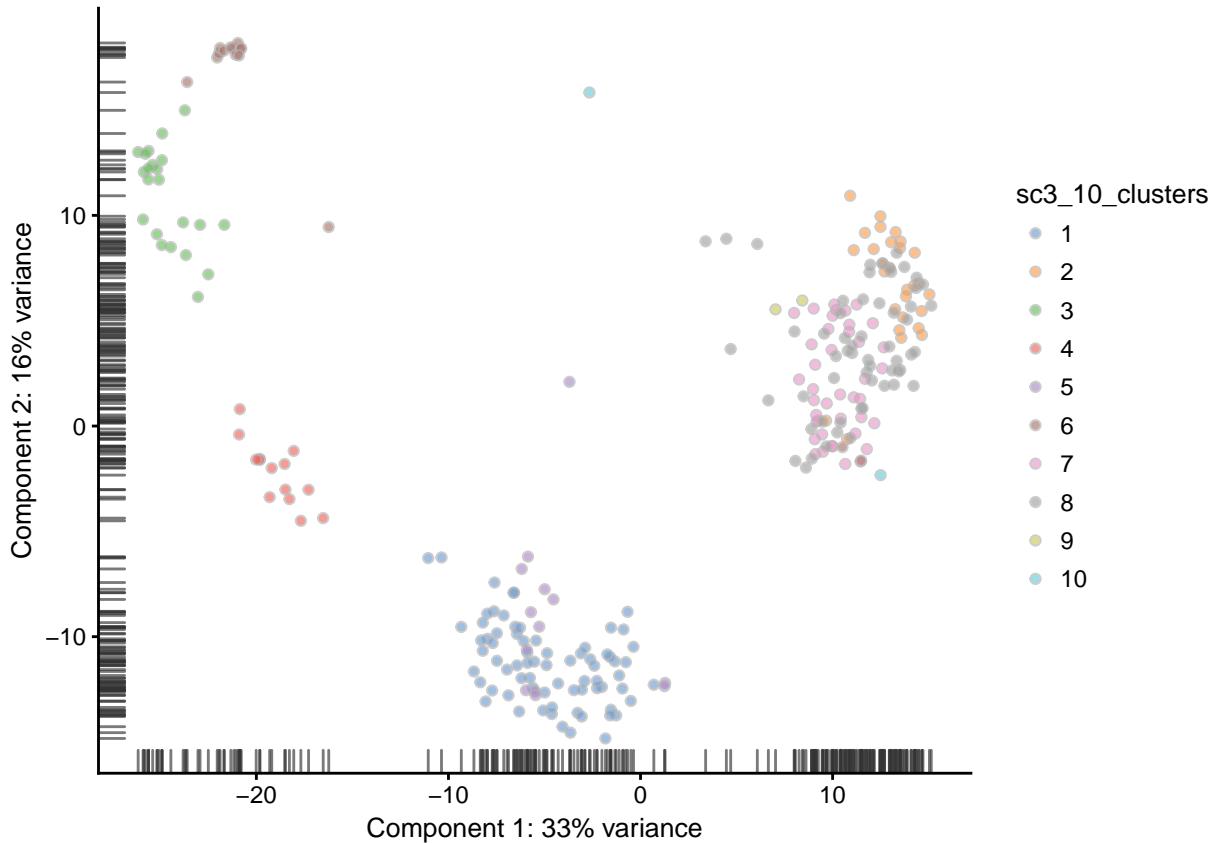
Identified marker genes:

```
sc3_plot_markers(deng, k = 10, show_pdata = "cell_type2")
```



PCA plot with highlighted SC3 clusters:

```
plotPCA(deng, colour_by = "sc3_10_clusters")
```



Compare the results of SC3 clustering with the original publication cell type labels:

```
adjustedRandIndex(colData(deng)$cell_type2, colData(deng)$sc3_10_clusters)
```

```
## [1] 0.7705208
```

**Note** SC3 can also be run in an interactive Shiny session:

```
sc3_interactive(deng)
```

This command will open SC3 in a web browser.

**Note** Due to direct calculation of distances SC3 becomes very slow when the number of cells is  $> 5000$ . For large datasets containing up to  $10^5$  cells we recommend using Seurat (see chapter 9).

- **Exercise 1:** Run SC3 for  $k$  from 8 to 12 and explore different clustering solutions in your web browser.
- **Exercise 2:** Which clusters are the most stable when  $k$  is changed from 8 to 12? (Look at the “Stability” tab)
- **Exercise 3:** Check out differentially expressed genes and marker genes for the obtained clusterings. Please use  $k = 10$ .
- **Exercise 4:** Change the marker genes threshold (the default is 0.85). Does SC3 find more marker genes?

### 8.2.3 pcaReduce

pcaReduce operates directly on the expression matrix. It is recommended to use a gene filter and log transformation before running pcaReduce. We will use the default SC3 gene filter (note that the `exprs` slot of a `scater` object is log-transformed by default).

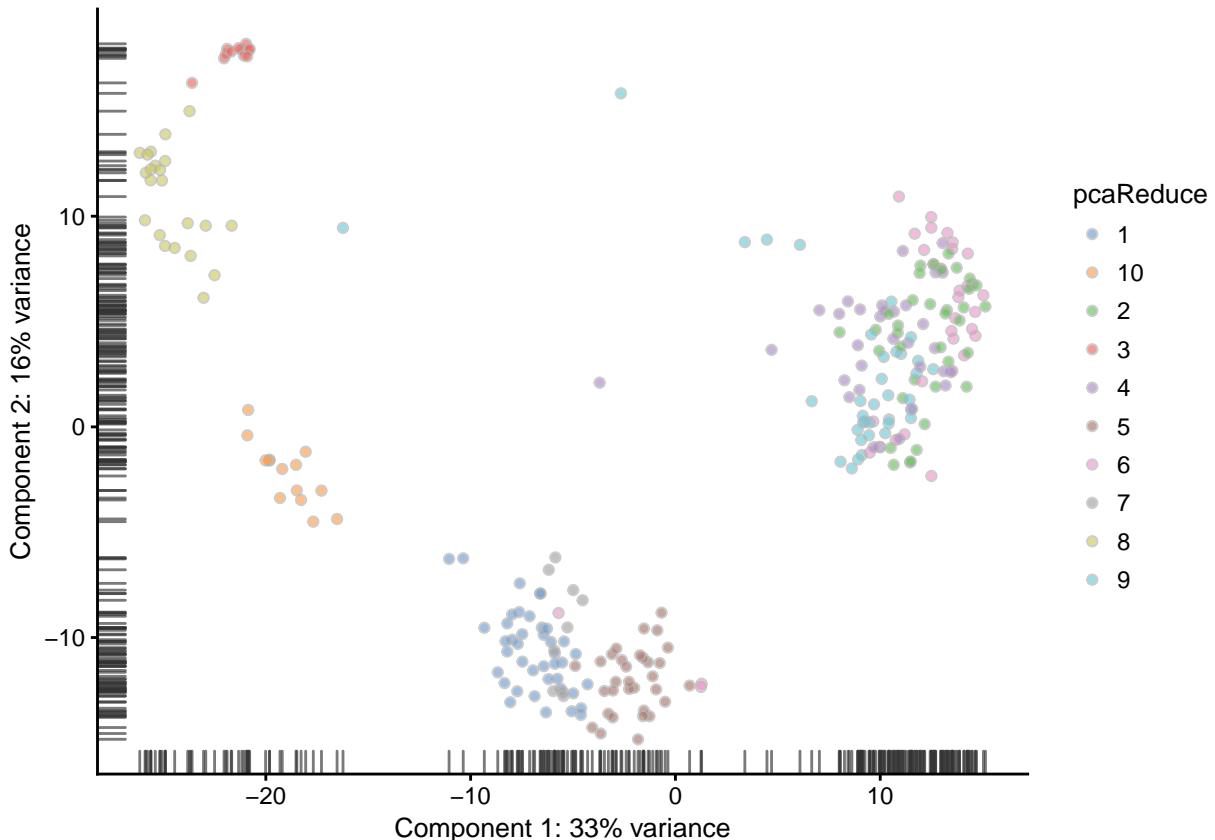
```
# use the same gene filter as in SC3
input <- logcounts(deng$rowData(deng)$sc3_gene_filter, ])
```

There are several parameters used by `pcaReduce`: \* `nbt` defines a number of `pcaReduce` runs (it is stochastic and may have different solutions after different runs) \* `q` defines number of dimensions to start clustering with. The output will contain partitions for all  $k$  from 2 to  $q+1$ . \* `method` defines a method used for clustering. S - to perform sampling based merging, M - to perform merging based on largest probability.

We will run `pcaReduce` 1 time:

```
# run pcaReduce 1 time creating hierarchies from 1 to 30 clusters
pca.red <- PCAreduce(t(input), nbt = 1, q = 30, method = 'S')[[1]]

colData(deng)$pcaReduce <- as.character(pca.red[, 32 - 10])
plotPCA(deng, colour_by = "pcaReduce")
```



**Exercise 5:** Run `pcaReduce` for  $k = 2$  and plot a similar PCA plot. Does it look good?

**Hint:** When running `pcaReduce` for different  $ks$  you do not need to rerun `PCAreduce` function, just use already calculated `pca.red` object.

**Our solution:**

**Exercise 6:** Compare the results between `pcaReduce` and the original publication cell types for  $k = 10$ .

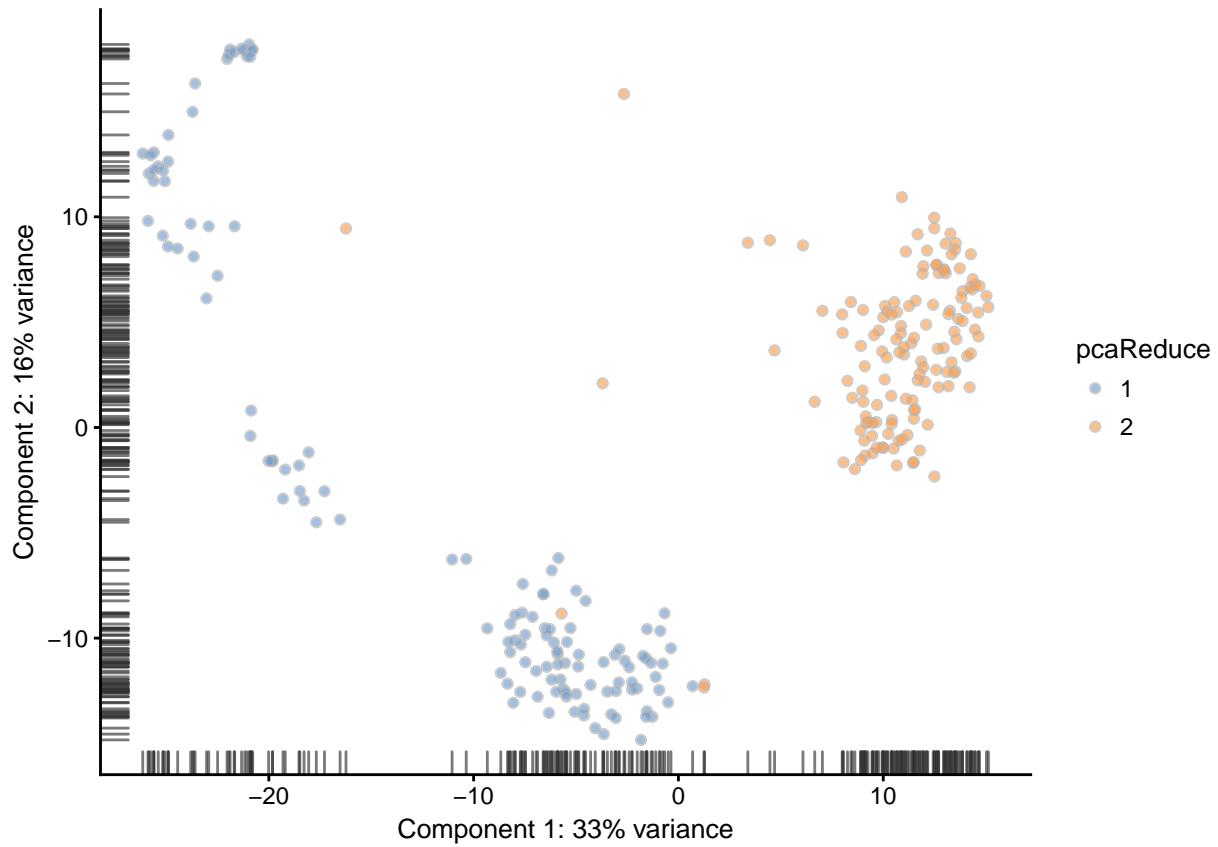


Figure 8.6: Clustering solutions of `pcaReduce` method for  $k = 2$ .

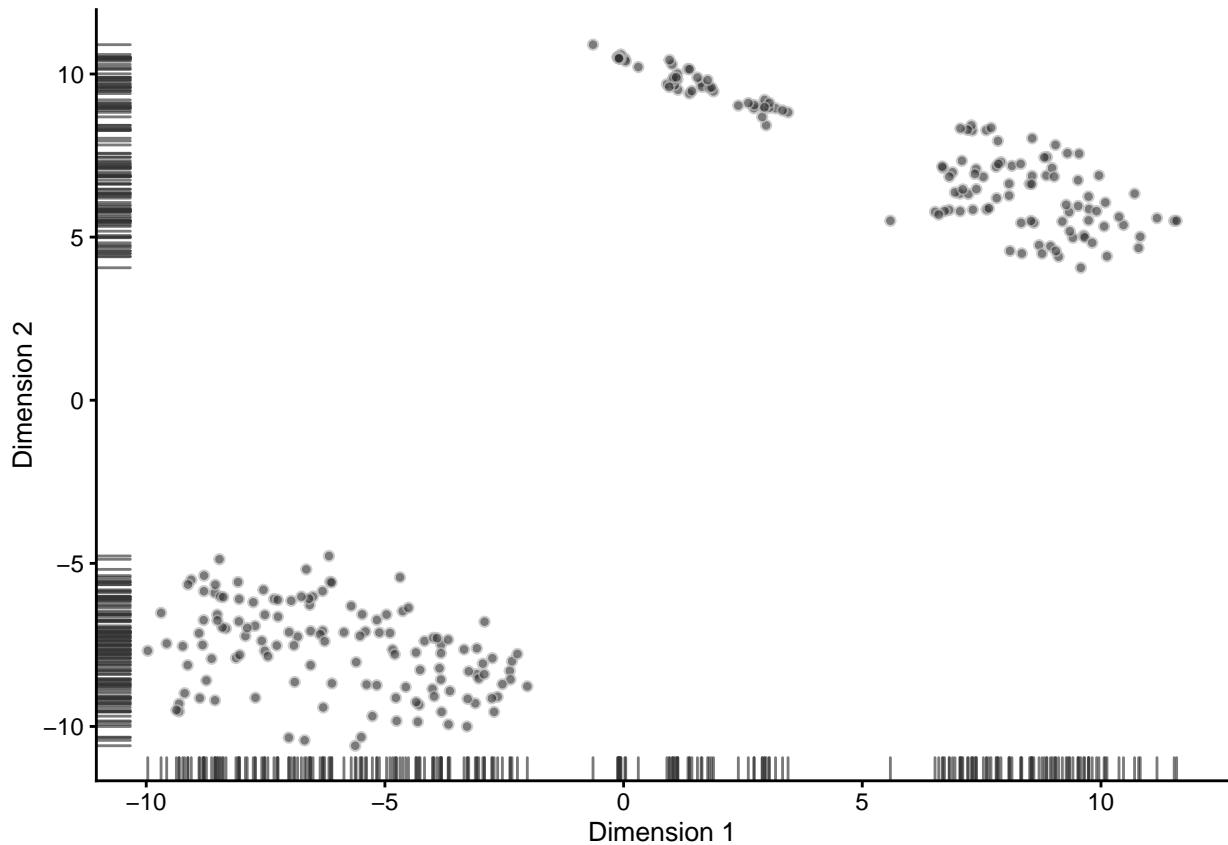


Figure 8.7: tSNE map of the patient data

**Our solution:**

```
## [1] 0.4216031
```

#### 8.2.4 tSNE + kmeans

tSNE plots that we saw before (7.3.3) when used the `scater` package are made by using the `Rtsne` and `ggplot2` packages. Here we will do the same:

```
deng <- plotTSNE(deng, rand_seed = 1, return_SCE = TRUE)
```

Note that all points on the plot above are black. This is different from what we saw before, when the cells were coloured based on the annotation. Here we do not have any annotation and all cells come from the same batch, therefore all dots are black.

Now we are going to apply  $k$ -means clustering algorithm to the cloud of points on the tSNE map. How many groups do you see in the cloud?

We will start with  $k = 8$ :

```
colData(deng)$tSNE_kmeans <- as.character(kmeans(deng@reducedDims$TSNE, centers = 8)$clust)
plotTSNE(deng, rand_seed = 1, colour_by = "tSNE_kmeans")
```

**Exercise 7:** Make the same plot for  $k = 10$ .

**Exercise 8:** Compare the results between `tSNE+kmeans` and the original publication cell types. Can the

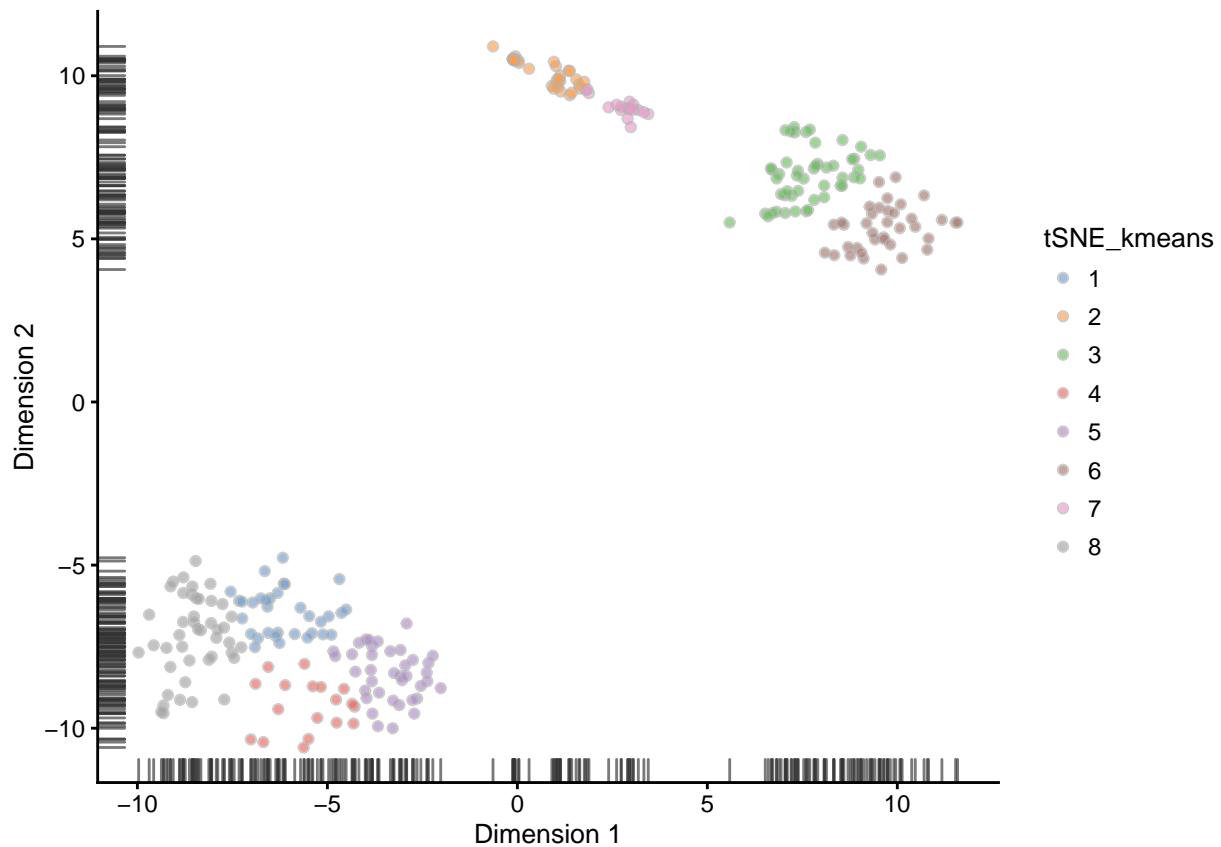


Figure 8.8: tSNE map of the patient data with 8 colored clusters, identified by the k-means clustering algorithm

results be improved by changing the `perplexity` parameter?

**Our solution:**

```
## [1] 0.3701639
```

As you may have noticed, both `pcaReduce` and `tSNE+kmeans` are stochastic and give different results every time they are run. To get a better overview of the solutions, we need to run the methods multiple times. `SC3` is also stochastic, but thanks to the consensus step, it is more robust and less likely to produce different outcomes.

### 8.2.5 SNN-Cliq

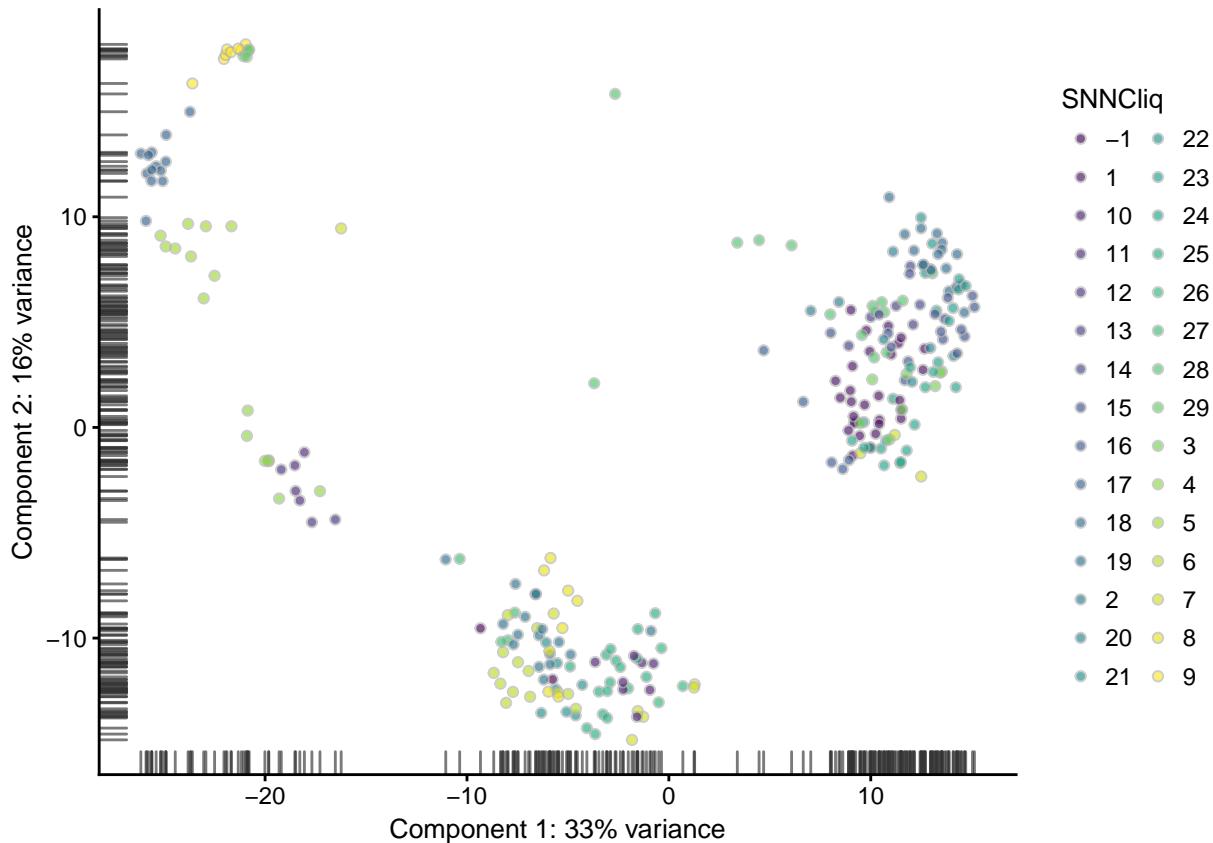
Here we run SNN-cliq with the default parameters provided in the author's example:

```
distan <- "euclidean"
par.k <- 3
par.r <- 0.7
par.m <- 0.5
# construct a graph
scRNA.seq.funcs::SNN(
  data = t(input),
  outfile = "snn-cliq.txt",
  k = par.k,
  distance = distan
)
# find clusters in the graph
snn.res <-
  system(
    paste0(
      "python utils/Cliq.py ",
      "-i snn-cliq.txt ",
      "-o res-snn-cliq.txt ",
      "-r ", par.r,
      " -m ", par.m
    ),
    intern = TRUE
  )
cat(paste(snn.res, collapse = "\n"))

## input file snn-cliq.txt
## find 66 quasi-cliques
## merged into 29 clusters
## unique assign done

snn.res <- read.table("res-snn-cliq.txt")
# remove files that were created during the analysis
system("rm snn-cliq.txt res-snn-cliq.txt")

colData(deng)$SNNCliq <- as.character(snn.res[,1])
plotPCA(deng, colour_by = "SNNCliq")
```



**Exercise 9:** Compare the results between SNN-Cliq and the original publication cell types.

**Our solution:**

```
## [1] 0.2629731
```

### 8.2.6 SINCERA

As mentioned in the previous chapter SINCERA is based on hierarchical clustering. One important thing to keep in mind is that it performs a gene-level z-score transformation before doing clustering:

```
# perform gene-by-gene per-sample z-score transformation
dat <- apply(input, 1, function(y) scRNA.seq.funcs::z.transform.helper(y))
# hierarchical clustering
dd <- as.dist((1 - cor(t(dat), method = "pearson"))/2)
hc <- hclust(dd, method = "average")
```

If the number of cluster is not known SINCERA can identify **k** as the minimum height of the hierarchical tree that generates no more than a specified number of singleton clusters (clusters containing only 1 cell)

```
num.singleton <- 0
kk <- 1
for (i in 2:dim(dat)[2]) {
  clusters <- cutree(hc, k = i)
  clustersizes <- as.data.frame(table(clusters))
  singleton.clusters <- which(clustersizes$Freq < 2)
  if (length(singleton.clusters) <= num.singleton) {
    kk <- i
```

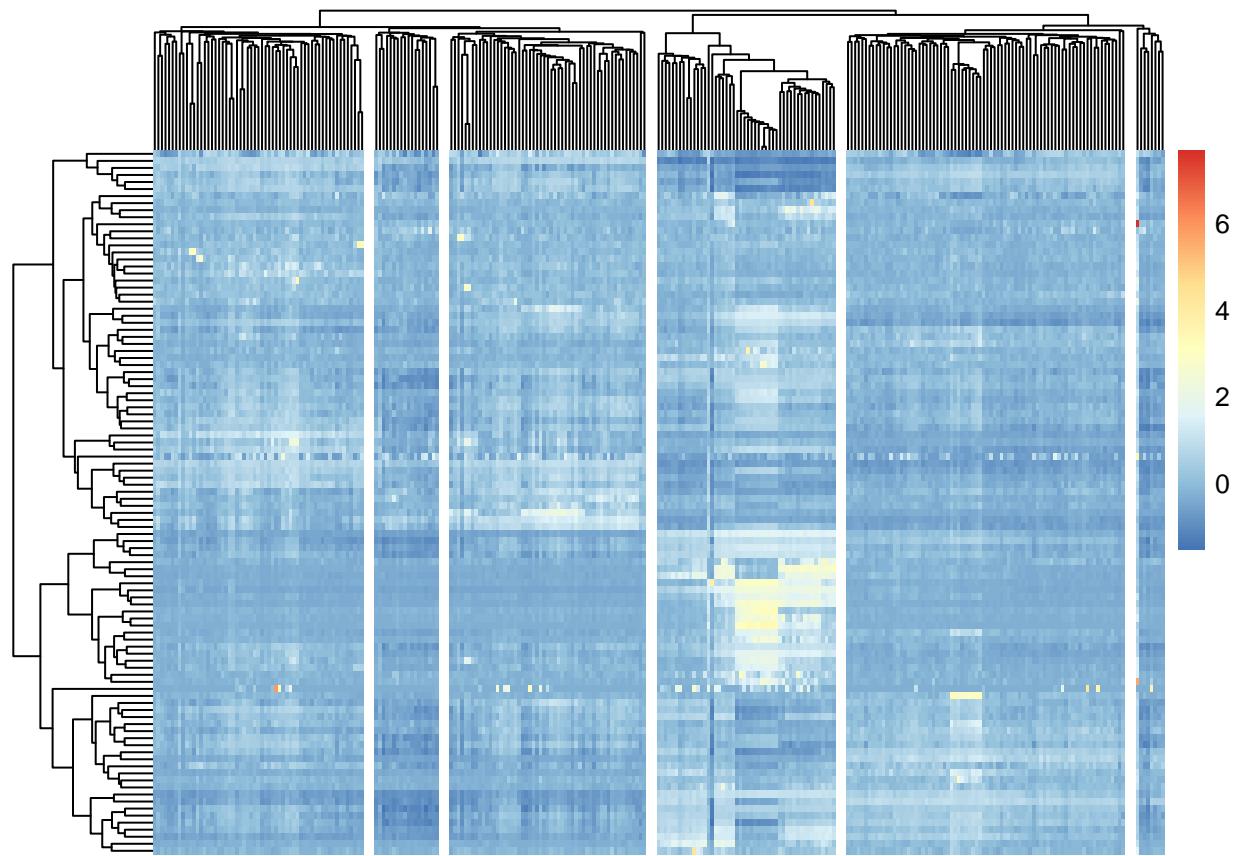


Figure 8.9: Clustering solutions of SINCERA method using found  $k$

```

    } else {
      break;
    }
}
cat(kk)

```

```
## 6
```

Let's now visualize the SINCERA results as a heatmap:

```

pheatmap(
  t(dat),
  cluster_cols = hc,
  cutree_cols = kk,
  kmeans_k = 100,
  show_rownames = FALSE
)

```

**Exercise 10:** Compare the results between SINCERA and the original publication cell types.

**Our solution:**

```
## [1] 0.3823537
```

**Exercise 11:** Is using the singleton cluster criteria for finding  $k$  a good idea?

### 8.2.7 sessionInfo()

```

## R version 3.4.3 (2017-11-30)
## Platform: x86_64-pc-linux-gnu (64-bit)
## Running under: Debian GNU/Linux 9 (stretch)
##
## Matrix products: default
## BLAS: /usr/lib/openblas-base/libblas.so.3
## LAPACK: /usr/lib/libopenblas-r0.2.19.so
##
## locale:
## [1] LC_CTYPE=en_US.UTF-8          LC_NUMERIC=C
## [3] LC_TIME=en_US.UTF-8          LC_COLLATE=en_US.UTF-8
## [5] LC_MONETARY=en_US.UTF-8      LC_MESSAGES=C
## [7] LC_PAPER=en_US.UTF-8         LC_NAME=C
## [9] LC_ADDRESS=C                 LC_TELEPHONE=C
## [11] LC_MEASUREMENT=en_US.UTF-8   LC_IDENTIFICATION=C
##
## attached base packages:
## [1] stats4     parallel    methods    stats      graphics   grDevices utils
## [8] datasets   base
##
## other attached packages:
## [1] pheatmap_1.0.8            scater_1.6.2
## [3] SingleCellExperiment_1.0.0 SummarizedExperiment_1.8.1
## [5] DelayedArray_0.4.1        matrixStats_0.53.0
## [7] GenomicRanges_1.30.1       GenomeInfoDb_1.14.0
## [9] IRanges_2.12.0             S4Vectors_0.16.0
## [11] ggpplot2_2.2.1            SC3_1.7.7
## [13] pcaReduce_1.0              mclust_5.4
## [15] mnormt_1.5-5             pcaMethods_1.70.0
## [17] Biobase_2.38.0            BiocGenerics_0.24.0
## [19] knitr_1.19
##
## loaded via a namespace (and not attached):
## [1] Rtsne_0.13           ggbeeswarm_0.6.0   colorspace_1.3-2
## [4] rjson_0.2.15          class_7.3-14       rprojroot_1.3-2
## [7] XVector_0.18.0         bit64_0.9-7        AnnotationDbi_1.40.0
## [10] mvtnorm_1.0-7          scRNA.seq.funcs_0.1.0 codetools_0.2-15
## [13] tximport_1.6.0          doParallel_1.0.11   robustbase_0.92-8
## [16] cluster_2.0.6          shinydashboard_0.6.1 shiny_1.0.5
## [19] rrcov_1.4-3            compiler_3.4.3     httr_1.3.1
## [22] backports_1.1.2        assertthat_0.2.0   Matrix_1.2-7.1
## [25] lazyeval_0.2.1          limma_3.34.8       htmltools_0.3.6
## [28] prettyunits_1.0.2        tools_3.4.3        bindrcpp_0.2
## [31] gtable_0.2.0            glue_1.2.0         GenomeInfoDbData_1.0.0
## [34] reshape2_1.4.3          dplyr_0.7.4       doRNG_1.6.6
## [37] Rcpp_0.12.15            gdata_2.18.0      iterators_1.0.9
## [40] xfun_0.1                stringr_1.2.0     mime_0.5
## [43] hypergeo_1.2-13         rngtools_1.2.4    gtools_3.5.0
## [46] WriteXLS_4.0.0          statmod_1.4.30   XML_3.98-1.9
## [49] edgeR_3.20.8            DEoptimR_1.0-8    MASS_7.3-45
## [52] zlibbioc_1.24.0          scales_0.5.0     rhdf5_2.22.0
## [55] RColorBrewer_1.1-2       yaml_2.1.16      memoise_1.1.0

```

```

## [58] gridExtra_2.3           pkgmaker_0.22          biomaRt_2.34.2
## [61] stringi_1.1.6            RSQLite_2.0             pcaPP_1.9-73
## [64] foreach_1.4.4            orthopolynom_1.0-5    e1071_1.6-8
## [67] contfrac_1.1-11          caTools_1.17.1          moments_0.14
## [70] rlang_0.1.6              pkgconfig_2.0.1         bitops_1.0-6
## [73] evaluate_0.10.1          lattice_0.20-34        ROCR_1.0-7
## [76] bindr_0.1                labeling_0.3            cowplot_0.9.2
## [79] bit_1.1-12               deSolve_1.20            plyr_1.8.4
## [82] magrittr_1.5              bookdown_0.6            R6_2.2.2
## [85] gplots_3.0.1              DBI_0.7                 pillar_1.1.0
## [88] RCurl_1.95-4.10          tibble_1.4.2            KernSmooth_2.23-15
## [91] rmarkdown_1.8               viridis_0.5.0           progress_1.1.2
## [94] locfit_1.5-9.1            grid_3.4.3              data.table_1.10.4-3
## [97] blob_1.1.0                digest_0.6.15           xtable_1.8-2
## [100] httpuv_1.3.5             elliptic_1.3-7          munsell_0.4.3
## [103] registry_0.5              beeswarm_0.2.3          viridisLite_0.3.0
## [106] viper_0.4.5

```

## 8.3 Feature Selection

```

library(scRNA.seq.funcs)
library(matrixStats)
library(M3Drop)
library(RColorBrewer)
library(SingleCellExperiment)
set.seed(1)

```

Single-cell RNASeq is capable of measuring the expression of many thousands of genes in every cell. However, in most situations only a portion of those will show a response to the biological condition of interest, e.g. differences in cell-type, drivers of differentiation, respond to an environmental stimulus. Most genes detected in a scRNASeq experiment will only be detected at different levels due to technical noise. One consequence of this is that technical noise and batch effects can obscure the biological signal of interest.

Thus, it is often advantageous to perform feature selection to remove those genes which only exhibit technical noise from downstream analysis. Not only does this generally increase the signal:noise ratio in the data; it also reduces the computational complexity of analyses, by reducing the total amount of data to be processed.

For scRNASeq data, we will be focusing on unsupervised methods of feature selection which don't require any a priori information, such as cell-type labels or biological group, since they are not available, or may be unreliable, for many experiments. In contrast, differential expression (chapter 8.6) can be considered a form of supervised feature selection since it uses the known biological label of each sample to identify features (i.e. genes) which are expressed at different levels across groups.

For this section we will continue working with the Deng data.

```

deng <- readRDS("deng/deng-reads.rds")
cellLabels <- colData(deng)$cell_type2

```

This data can be QCed and normalized for library size using M3Drop, which removes cells with few detected genes, removes undetected genes, and converts raw counts to CPM.

```

deng_list <- M3DropCleanData(
  counts(deng),
  labels = cellLabels,
  min_detected_genes = 100,

```

```

  is.counts = TRUE
)
expr_matrix <- deng_list$data # Normalized & filtered expression matrix
celltype_labs <- factor(deng_list$labels) # filtered cell-type labels
cell_colors <- brewer.pal(max(3,length(unique(celltype_labs))), "Set3")

```

**Exercise 1:** How many cells & genes have been removed by this filtering?

### 8.3.1 Identifying Genes vs a Null Model

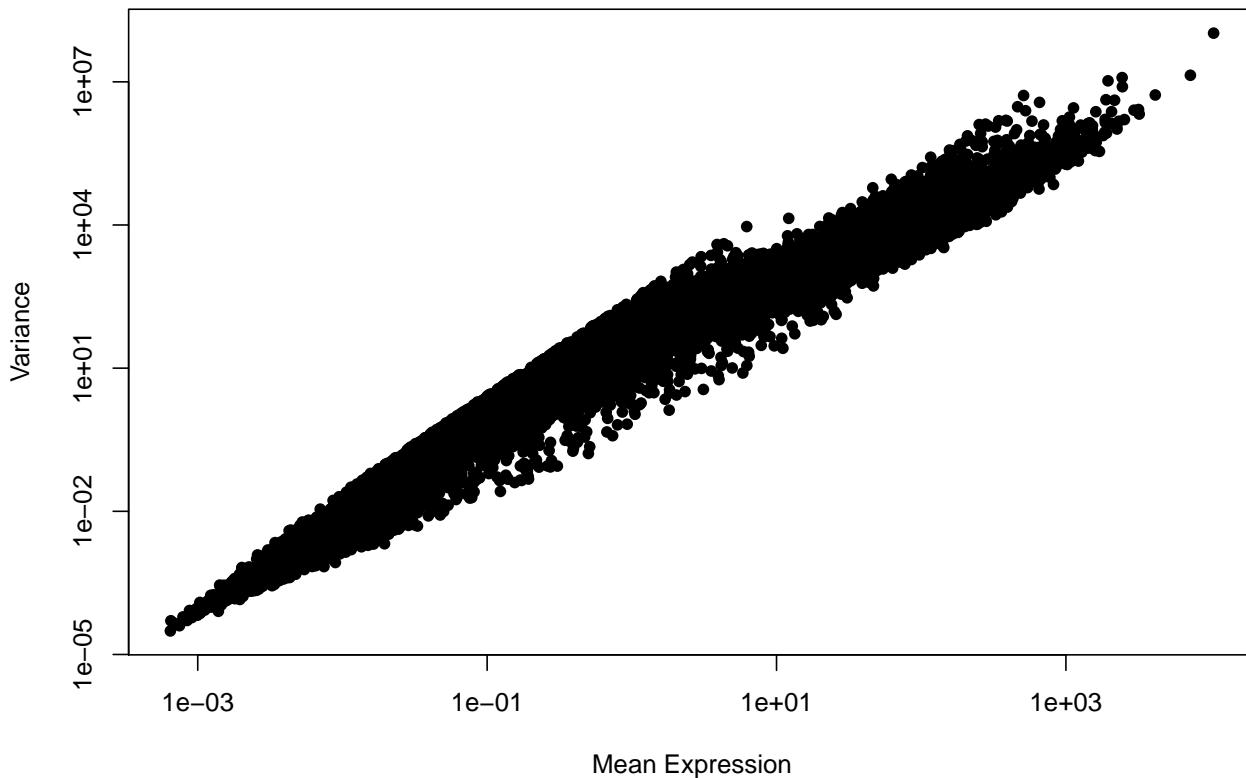
There are two main approaches to unsupervised feature selection. The first is to identify genes which behave differently from a null model describing just the technical noise expected in the dataset.

If the dataset contains spike-in RNAs they can be used to directly model technical noise. However, measurements of spike-ins may not experience the same technical noise as endogenous transcripts (Svensson et al., 2017). In addition, scRNASeq experiments often contain only a small number of spike-ins which reduces our confidence in fitted model parameters.

#### 8.3.1.1 Highly Variable Genes

The first method proposed to identify features in scRNASeq datasets was to identify highly variable genes (HVG). HVG assumes that if genes have large differences in expression across cells some of those differences are due to biological difference between the cells rather than technical noise. However, because of the nature of count data, there is a positive relationship between the mean expression of a gene and the variance in the read counts across cells. This relationship must be corrected for to properly identify HVGs.

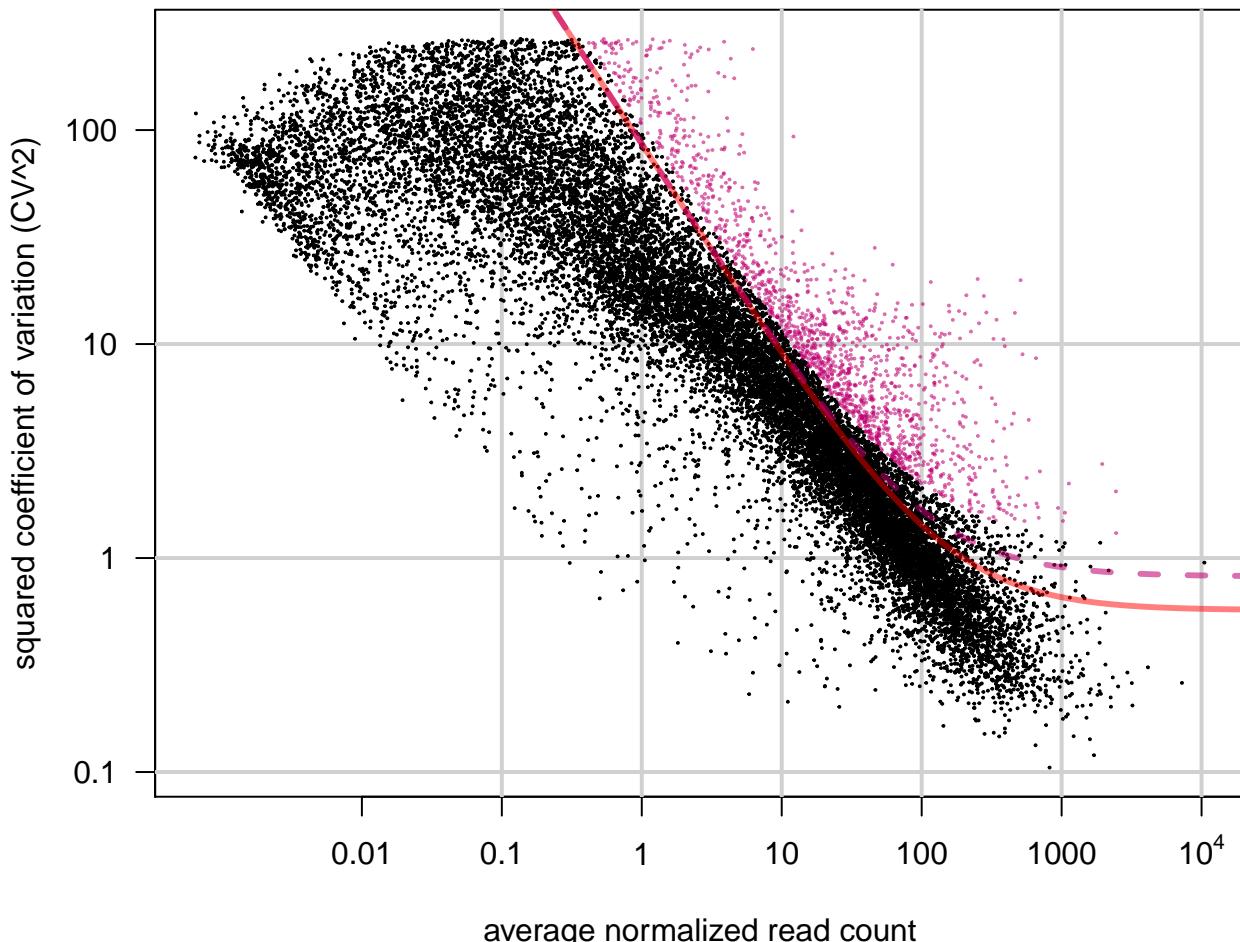
**Exercise 2** Using the functions `rowMeans` and `rowVars` to plot the relationship between mean expression and variance for all genes in this dataset. (Hint: use `log="xy"` to plot on a log-scale).



A popular method to correct for the relationship between variance and mean expression was proposed by Brennecke et al.. To use the Brennecke method, we first normalize for library size then calculate the mean and the square coefficient of variation (variation divided by the squared mean expression). A quadratic curve is fit to the relationship between these two variables for the ERCC spike-in, and then a chi-square test is used to find genes significantly above the curve. This method is included in the M3Drop package as the `Brennecke_getVariableGenes(counts, spikes)` function. However, this dataset does not contain spike-ins so we will use the entire dataset to estimate the technical noise.

In the figure below the red curve is the fitted technical noise model and the dashed line is the 95% CI. Pink dots are the genes with significant biological variability after multiple-testing correction.

```
Brennecke_HVG <- BrenneckeGetVariableGenes(
  expr_matrix,
  fdr = 0.01,
  minBiolDisp = 0.5
)
```



### 8.3.1.2 High Dropout Genes

An alternative to finding HVGs is to identify genes with unexpectedly high numbers of zeros. The frequency of zeros, known as the “dropout rate”, is very closely related to expression level in scRNASeq data. Zeros are the dominant feature of single-cell RNASeq data, typically accounting for over half of the entries in the final expression matrix. These zeros predominantly result from the failure of mRNAs failing to be reverse transcribed (Andrews and Hemberg, 2016). Reverse transcription is an enzyme reaction thus can be modelled using the Michaelis-Menten equation:

$$P_{\text{dropout}} = 1 - S/(K + S)$$

where  $S$  is the mRNA concentration in the cell (we will estimate this as average expression) and  $K$  is the Michaelis-Menten constant.

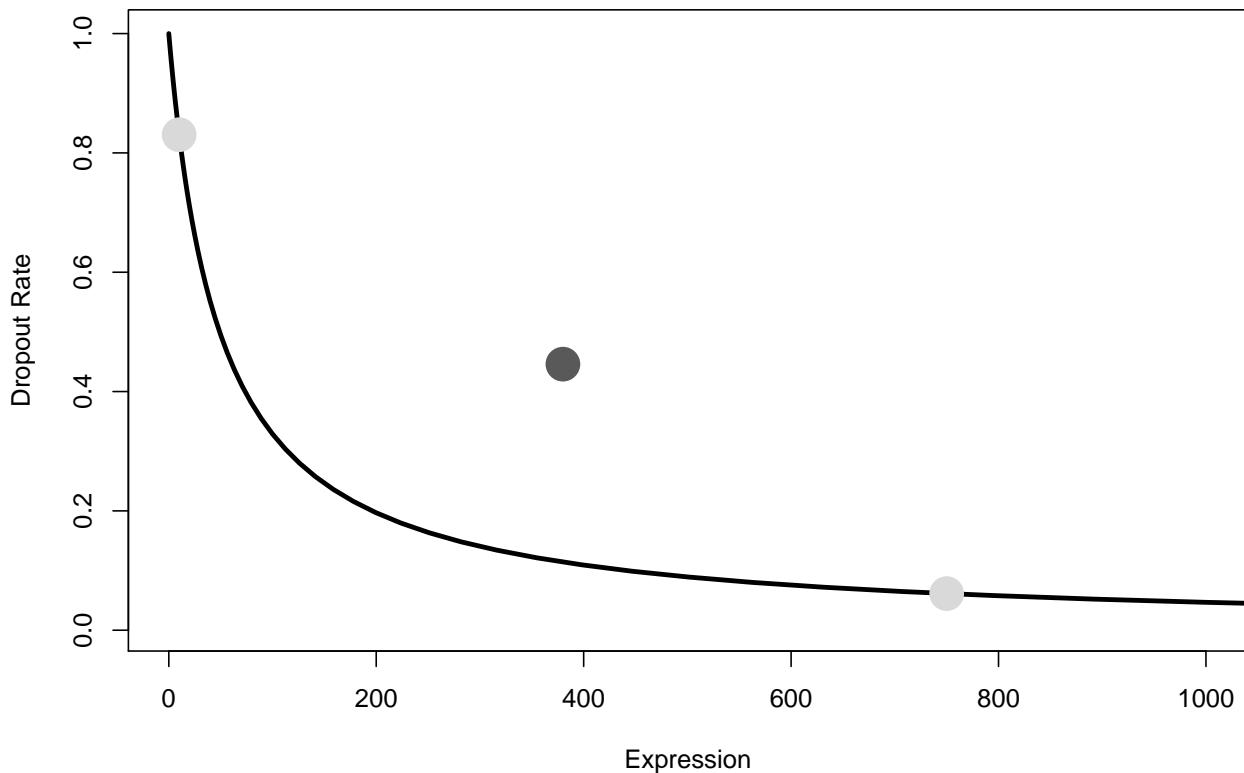
Because the Michaelis-Menten equation is a convex non-linear function, genes which are differentially expressed across two or more populations of cells in our dataset will be shifted up/right of the Michaelis-Menten model (see Figure below).

```
K <- 49
S_sim <- 10^seq(from = -3, to = 4, by = 0.05) # range of expression values
MM <- 1 - S_sim / (K + S_sim)
```

```

plot(
  S_sim,
  MM,
  type = "l",
  lwd = 3,
  xlab = "Expression",
  ylab = "Dropout Rate",
  xlim = c(1,1000)
)
S1 <- 10
P1 <- 1 - S1 / (K + S1) # Expression & dropouts for cells in condition 1
S2 <- 750
P2 <- 1 - S2 / (K + S2) # Expression & dropouts for cells in condition 2
points(
  c(S1, S2),
  c(P1, P2),
  pch = 16,
  col = "grey85",
  cex = 3
)
mix <- 0.5 # proportion of cells in condition 1
points(
  S1 * mix + S2 * (1 - mix),
  P1 * mix + P2 * (1 - mix),
  pch = 16,
  col = "grey35",
  cex = 3
)

```

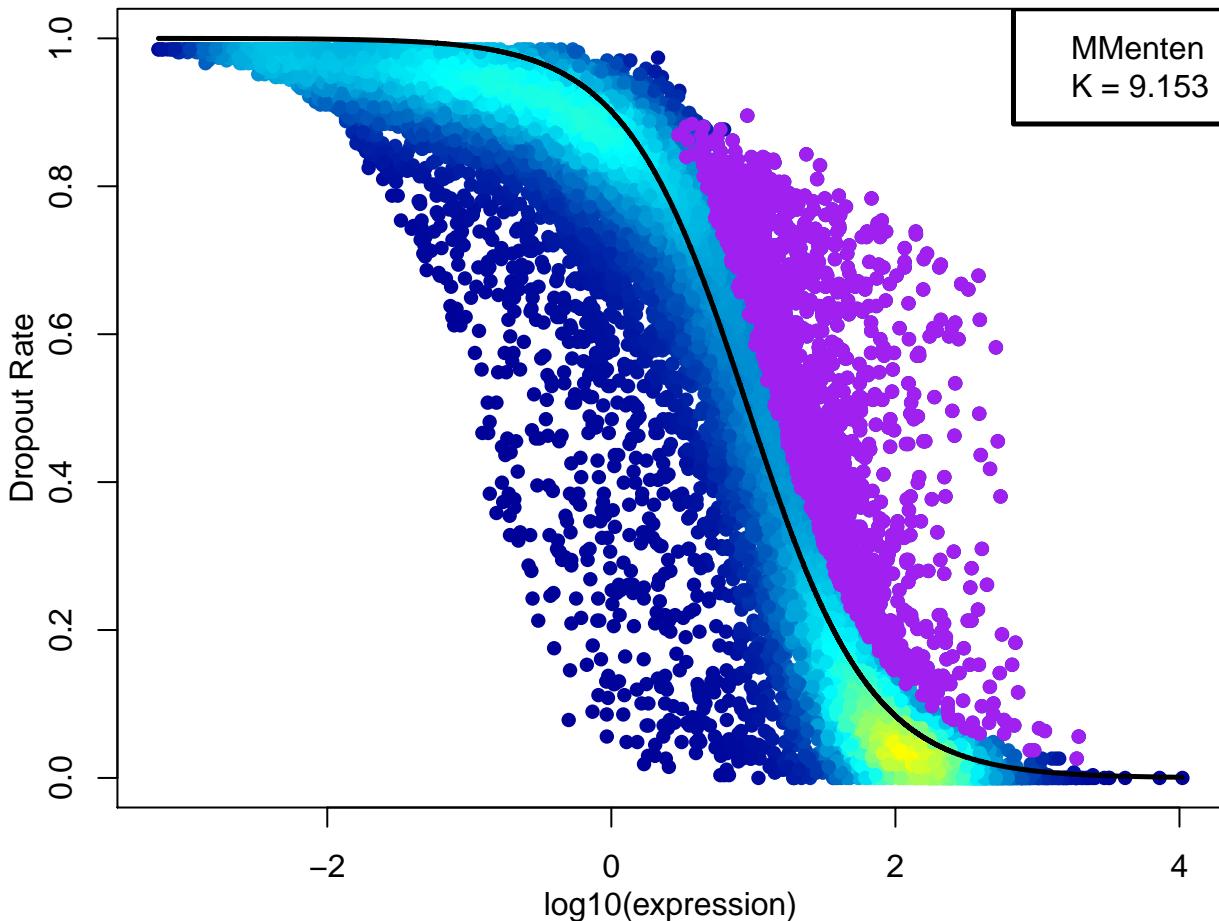


**Note:** add `log="x"` to the `plot` call above to see how this looks on the log scale, which is used in M3Drop figures.

**Exercise 3:** Produce the same plot as above with different expression levels (S1 & S2) and/or mixtures (mix).

We use M3Drop to identify significant outliers to the right of the MM curve. We also apply 1% FDR multiple testing correction:

```
M3Drop_genes <- M3DropFeatureSelection(
  expr_matrix,
  mt_method = "fdr",
  mt_threshold = 0.01
)
```



```
M3Drop_genes <- M3Drop_genes$Gene
```

An alternative method is contained in the M3Drop package that is tailored specifically for UMI-tagged data which generally contains many zeros resulting from low sequencing coverage in addition to those resulting from insufficient reverse-transcription. This model is the Depth-Adjusted Negative Binomial (DANB). This method describes each expression observation as a negative binomial model with a mean related to both the mean expression of the respective gene and the sequencing depth of the respective cell, and a variance related to the mean-expression of the gene.

Unlike the Michaelis-Menten and HVG methods there isn't a reliable statistical test for features selected by this model, so we will consider the top 1500 genes instead.

```
deng_int <- NBumiConvertToInteger(counts(deng))
DANB_fit <- NBumiFitModel(deng_int) # DANB is fit to the raw count matrix
# Perform DANB feature selection
DropFS <- NBumiFeatureSelectionCombinedDrop(DANB_fit)
DANB_genes <- names(DropFS[1:1500])
```

### 8.3.2 Correlated Expression

A completely different approach to feature selection is to use gene-gene correlations. This method is based on the idea that multiple genes will be differentially expressed between different cell-types or cell-states. Genes which are expressed in the same cell-population will be positively correlated with each other whereas genes expressed in different cell-populations will be negatively correlated with each other. Thus important genes can be identified by the magnitude of their correlation with other genes.

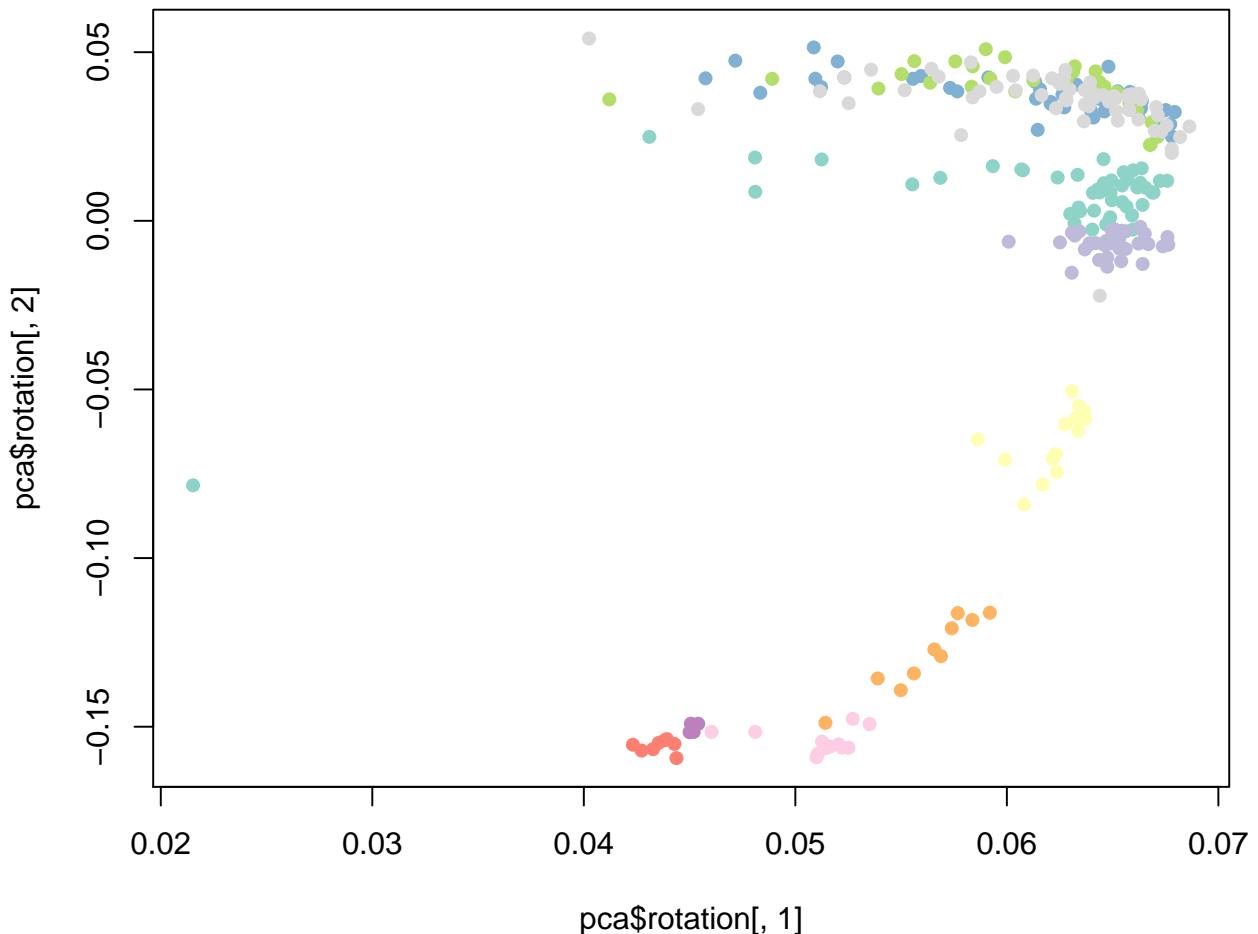
The limitation of this method is that it assumes technical noise is random and independent for each cell, thus shouldn't produce gene-gene correlations, but this assumption is violated by batch effects which are generally systematic between different experimental batches and will produce gene-gene correlations. As a result it is more appropriate to take the top few thousand genes as ranked by gene-gene correlation than consider the significance of the correlations.

```
cor_mat <- cor(t(expr_matrix), method = "spearman") # Gene-gene correlations
diag(cor_mat) <- rep(0, times = nrow(expr_matrix))
score <- apply(cor_mat, 1, function(x) {max(abs(x))}) #Correlation of highest magnitude
names(score) <- rownames(expr_matrix);
score <- score[order(-score)]
Cor_genes <- names(score[1:1500])
```

Lastly, another common method for feature selection in scRNASeq data is to use PCA loadings. Genes with high PCA loadings are likely to be highly variable and correlated with many other variable genes, thus may be relevant to the underlying biology. However, as with gene-gene correlations PCA loadings tend to be susceptible to detecting systematic variation due to batch effects; thus it is recommended to plot the PCA results to determine those components corresponding to the biological variation rather than batch effects.

```
# PCA is typically performed on log-transformed expression data
pca <- prcomp(log(expr_matrix + 1) / log(2))

# plot projection
plot(
  pca$rotation[,1],
  pca$rotation[,2],
  pch = 16,
  col = cell_colors[as.factor(celltype_labs)])
)
```



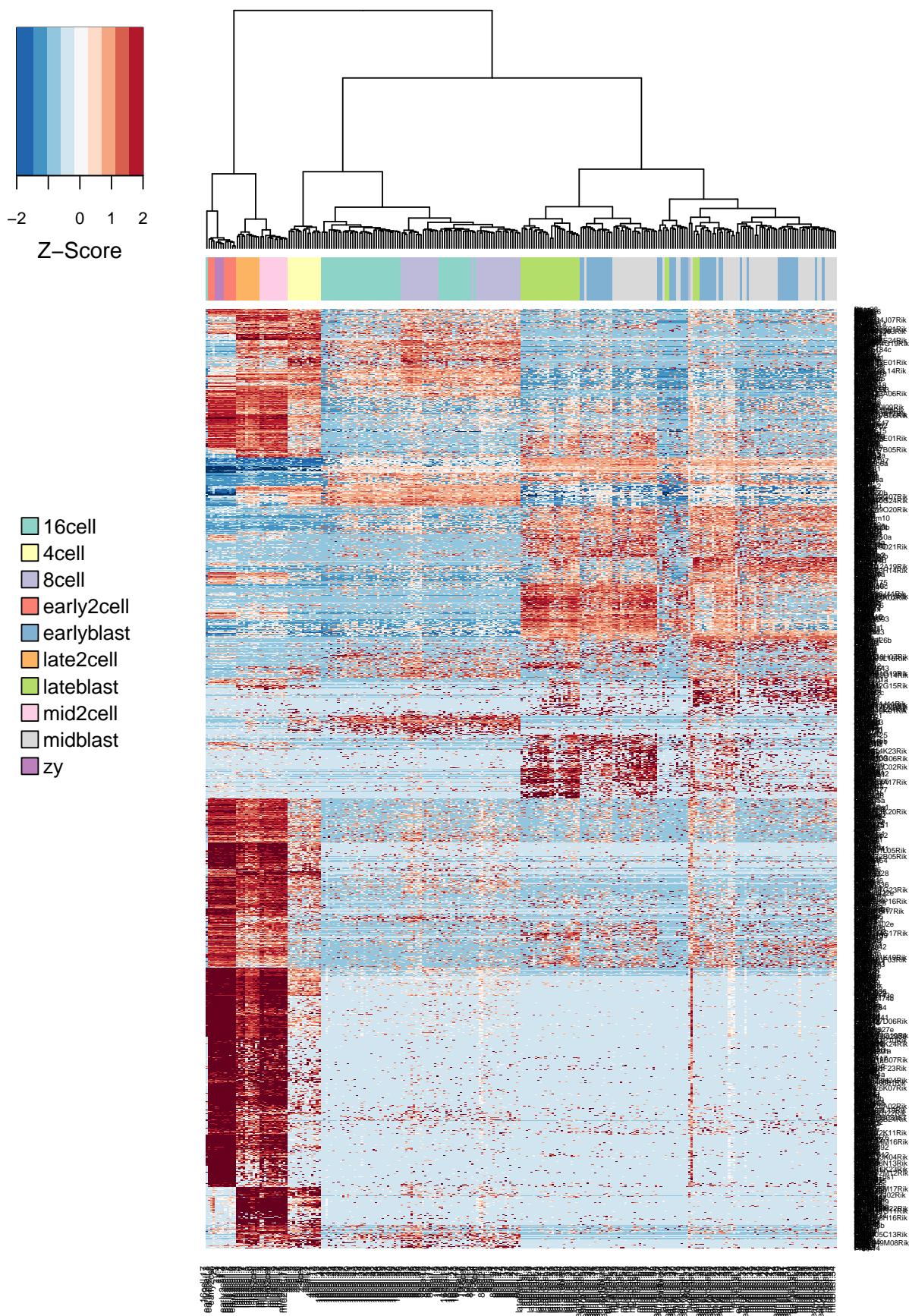
```
# calculate loadings for components 1 and 2
score <- rowSums(abs(pca$x[,c(1,2)]))
names(score) <- rownames(expr_matrix)
score <- score[order(-score)]
PCA_genes <- names(score[1:1500])
```

**Exercise 4** Consider the top 5 principal components. Which appear to be most biologically relevant? How does the top 1,500 features change if you consider the loadings for those components?

### 8.3.3 Comparing Methods

We can check whether the identified features really do represent genes differentially expressed between cell-types in this dataset.

```
M3DropExpressionHeatmap(
  M3Drop_genes,
  expr_matrix,
  cell_labels = celltype_labs
)
```



We can also consider how consistent each feature selection method is with the others using the Jaccard Index:

```
J <- sum(M3Drop_genes %in% HVG_genes) / length(unique(c(M3Drop_genes, HVG_genes)))
```

### Exercise 5

Plot the expression of the features for each of the other methods. Which appear to be differentially expressed? How consistent are the different methods for this dataset?

#### 8.3.4 sessionInfo()

```
## R version 3.4.3 (2017-11-30)
## Platform: x86_64-pc-linux-gnu (64-bit)
## Running under: Debian GNU/Linux 9 (stretch)
##
## Matrix products: default
## BLAS: /usr/lib/openblas-base/libblas.so.3
## LAPACK: /usr/lib/libopenblas-r0.2.19.so
##
## locale:
## [1] LC_CTYPE=en_US.UTF-8          LC_NUMERIC=C
## [3] LC_TIME=en_US.UTF-8          LC_COLLATE=en_US.UTF-8
## [5] LC_MONETARY=en_US.UTF-8      LC_MESSAGES=C
## [7] LC_PAPER=en_US.UTF-8         LC_NAME=C
## [9] LC_ADDRESS=C                 LC_TELEPHONE=C
## [11] LC_MEASUREMENT=en_US.UTF-8   LC_IDENTIFICATION=C
##
## attached base packages:
## [1] parallel    stats4     methods    stats      graphics   grDevices utils
## [8] datasets    base
##
## other attached packages:
## [1] SingleCellExperiment_1.0.0 SummarizedExperiment_1.8.1
## [3] DelayedArray_0.4.1        Biobase_2.38.0
## [5] GenomicRanges_1.30.1      GenomeInfoDb_1.14.0
## [7] IRanges_2.12.0            S4Vectors_0.16.0
## [9] BiocGenerics_0.24.0       RColorBrewer_1.1-2
## [11] M3Drop_3.05.00           numDeriv_2016.8-1
## [13] matrixStats_0.53.0       scRNA.seq.funcs_0.1.0
## [15] knitr_1.19
##
## loaded via a namespace (and not attached):
## [1] splines_3.4.3           elliptic_1.3-7      gtools_3.5.0
## [4] Formula_1.2-2           moments_0.14        statmod_1.4.30
## [7] latticeExtra_0.6-28     GenomeInfoDbData_1.0.0 yaml_2.1.16
## [10] pillar_1.1.0            backports_1.1.2     lattice_0.20-34
## [13] bbmle_1.0.20            digest_0.6.15       XVector_0.18.0
## [16] checkmate_1.8.5         colorspace_1.3-2    htmltools_0.3.6
## [19] Matrix_1.2-7.1          plyr_1.8.4          bookdown_0.6
## [22] zlibbioc_1.24.0         scales_0.5.0        gdata_2.18.0
## [25] Rtsne_0.13              htmlTable_1.11.2    tibble_1.4.2
## [28] mgcv_1.8-23             ggplot2_2.2.1       nnet_7.3-12
## [31] lazyeval_0.2.1           survival_2.40-1    magrittr_1.5
## [34] evaluate_0.10.1          nlme_3.1-129       MASS_7.3-45
```

```

## [37] gplots_3.0.1           foreign_0.8-67      reldist_1.6-6
## [40] tools_3.4.3            data.table_1.10.4-3 stringr_1.2.0
## [43] munsell_0.4.3          cluster_2.0.6       irlba_2.3.2
## [46] orthopolynom_1.0-5    compiler_3.4.3      caTools_1.17.1
## [49] contfrac_1.1-11        rlang_0.1.6        grid_3.4.3
## [52] RCurl_1.95-4.10       rstudioapi_0.7     htmlwidgets_1.0
## [55] bitops_1.0-6           base64enc_0.1-3    rmarkdown_1.8
## [58] hypergeo_1.2-13        gtable_0.2.0       deSolve_1.20
## [61] gridExtra_2.3           Hmisc_4.1-1        rprojroot_1.3-2
## [64] KernSmooth_2.23-15    stringi_1.1.6      Rcpp_0.12.15
## [67] rpart_4.1-10          acepack_1.4.1      xfun_0.1

```

## 8.4 Pseudotime analysis

```

library(SingleCellExperiment)
library(TSCAN)
library(M3Drop)
library(monocle)
library(destiny)
library(SLICER)
library(ouija)
library(scater)
library(ggplot2)
library(ggthemes)
library(ggbeeswarm)
library(corrplot)
set.seed(1)

```

In many situations, one is studying a process where cells change continuously. This includes, for example, many differentiation processes taking place during development: following a stimulus, cells will change from one cell-type to another. Ideally, we would like to monitor the expression levels of an individual cell over time. Unfortunately, such monitoring is not possible with scRNA-seq since the cell is lysed (destroyed) when the RNA is extracted.

Instead, we must sample at multiple time-points and obtain snapshots of the gene expression profiles. Since some of the cells will proceed faster along the differentiation than others, each snapshot may contain cells at varying points along the developmental progression. We use statistical methods to order the cells along one or more trajectories which represent the underlying developmental trajectories, this ordering is referred to as “pseudotime”.

In this chapter we will consider five different tools: Monocle, TSCAN, destiny, SLICER and ouija for ordering cells according to their pseudotime development. To illustrate the methods we will be using a dataset on mouse embryonic development (Deng et al., 2014). The dataset consists of 268 cells from 10 different time-points of early mouse development. In this case, there is no need for pseudotime alignment since the cell labels provide information about the development trajectory. Thus, the labels allow us to establish a ground truth so that we can evaluate and compare the different methods.

A recent review by Cannoodt et al provides a detailed summary of the various computational methods for trajectory inference from single-cell transcriptomics (Cannoodt et al., 2016). They discuss several tools, but unfortunately for our purposes many of these tools do not have complete or well-maintained implementations, and/or are not implemented in R.

Cannoodt et al cover:

- SCUBA - Matlab implementation

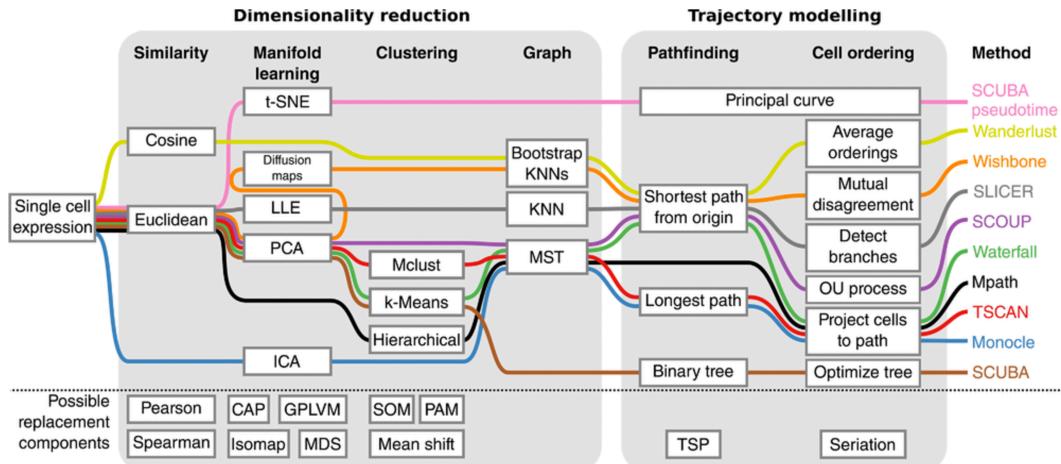


Figure 8.10: Descriptions of trajectory inference methods for single-cell transcriptomics data (Fig. 2 from Cannoodt et al, 2016).

- Wanderlust - Matlab (and requires registration to even download)
- Wishbone - Python
- SLICER - R, but package only available on Github
- SCOUPE - C++ command line tool
- Waterfall - R, but one R script in supplement
- Mpath - R pkg, but available as tar.gz on Github; function documentation but no vignette/workflow
- Monocle - Bioconductor package
- TSCAN - Bioconductor package

Unfortunately only two tools discussed (Monocle and TSCAN) meet the gold standard of open-source software hosted in a reputable repository.

The following figures from the paper summarise some of the features of the various tools.

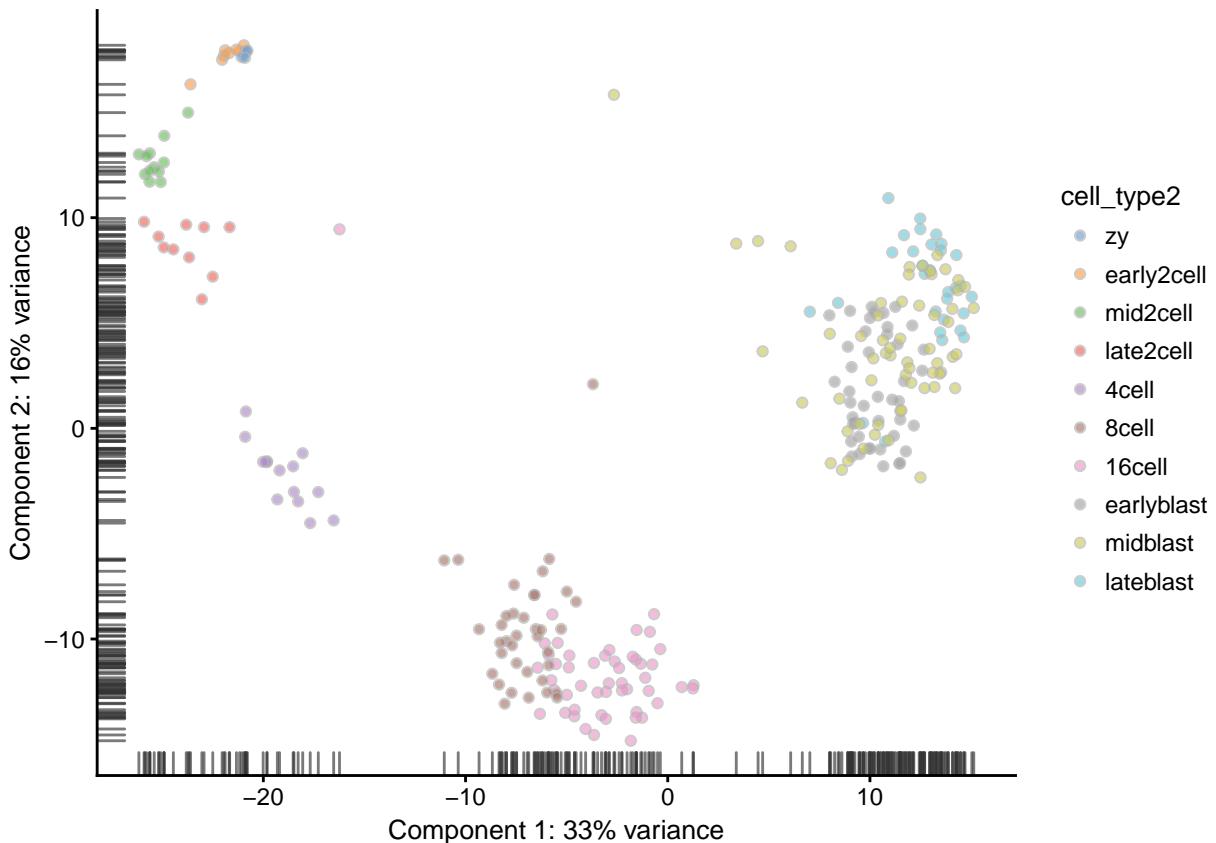
#### 8.4.1 First look at Deng data

Let us take a first look at the Deng data, without yet applying sophisticated pseudotime methods. As the plot below shows, simple PCA does a very good job of displaying the structure in these data. It is only once we reach the blast cell types (“earlyblast”, “midblast”, “lateblast”) that PCA struggles to separate the distinct cell types.

```
deng_SCE <- readRDS("deng/deng-reads.rds")
deng_SCE$cell_type2 <- factor(
  deng_SCE$cell_type2,
  levels = c("zy", "early2cell", "mid2cell", "late2cell",
            "4cell", "8cell", "16cell", "earlyblast",
            "midblast", "lateblast"))
)
cellLabels <- deng_SCE$cell_type2
deng <- counts(deng_SCE)
colnames(deng) <- cellLabels
deng_SCE <- plotPCA(deng_SCE, colour_by = "cell_type2",
                     return_SCE = TRUE)
```

Method	SCUBA pseudotime	Wanderlust	Wishbone	SLICER	SCOUP	Waterfall	Mpath	TSCAN	Monocle	SCUBA
Visual abstract										
Structure	Linear	Linear	Single bifurcation	Branching	Branching	Linear	Branching	Linear	Branching	Branching
Robustness strategy	Principal curves	Ensemble, starting cell	Ensemble, starting cell	Starting cell	Starting population	Clustering of cells	Clustering of cells using external labelling	Clustering of cells	Differential expression	Simple model
Extra input requirements	None	Starting cell	Starting cell	Starting cell	Starting population	None	Time points	None	Time points	Time points
Unbiased	+	±	±	±	±	+	-	+	-	-
Scalability w.r.t. cells	-	-	±	±	-	±	+	+	-	±
Scalability w.r.t. genes	+	+	+	+	-	+	±	±	±	+
Code and documentation	-	±	+	±	+	±	+	+	+	±
Parameter ease-of-use	+	+	+	+	-	±	-	+	+	+

Figure 8.11: Characterization of trajectory inference methods for single-cell transcriptomics data (Fig. 3 from Cannoodt et al, 2016).



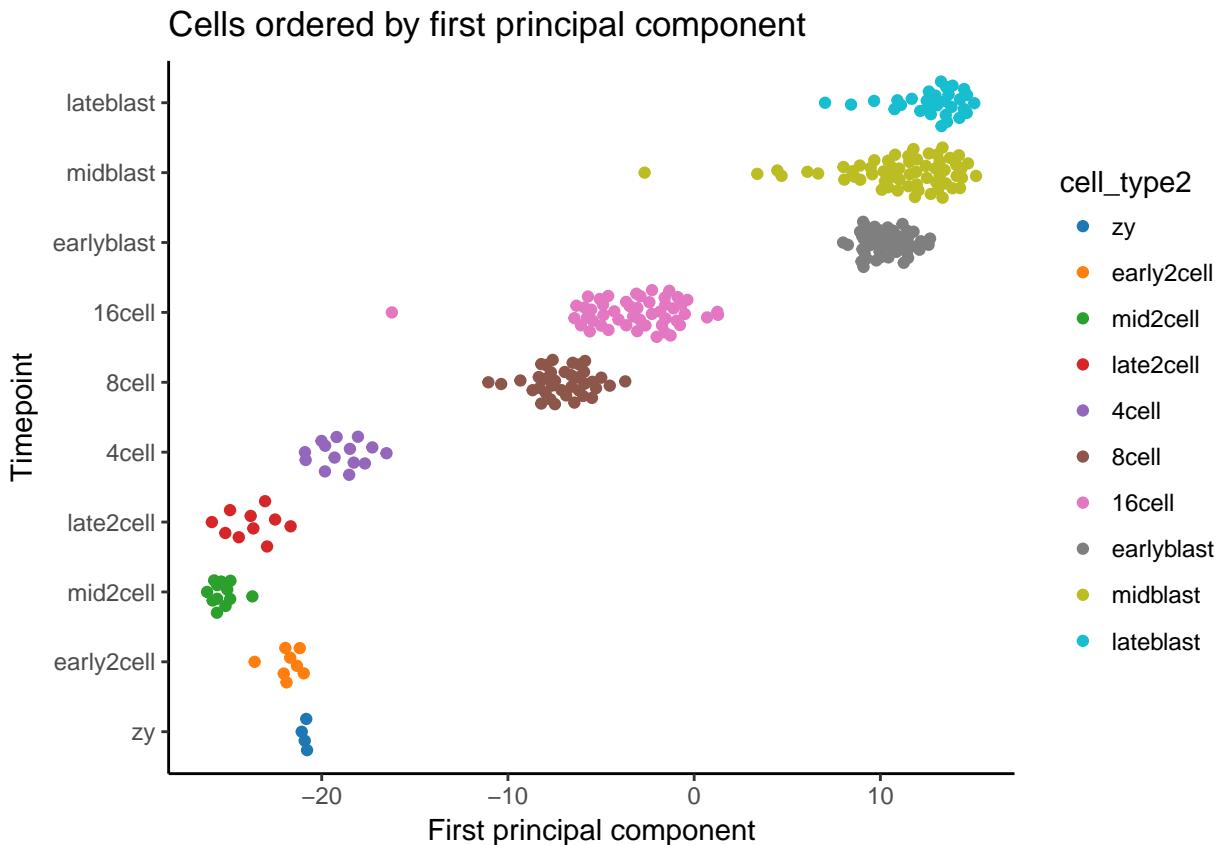
PCA, here, provides a useful baseline for assessing different pseudotime methods. For a very naive pseudotime we can just take the co-ordinates of the first principal component.

```
deng_SCE$PC1 <- reducedDim(deng_SCE, "PCA")[,1]
ggplot(as.data.frame(colData(deng_SCE)), aes(x = PC1, y = cell_type2,
```

```

        colour = cell_type2)) +
geom_quasirandom(groupOnX = FALSE) +
scale_color_tableau() + theme_classic() +
xlab("First principal component") + ylab("Timepoint") +
ggtitle("Cells ordered by first principal component")

```



As the plot above shows, PC1 struggles to correctly order cells early and late in the developmental timecourse, but overall does a relatively good job of ordering cells by developmental time.

Can bespoke pseudotime methods do better than naive application of PCA?

#### 8.4.2 TSCAN

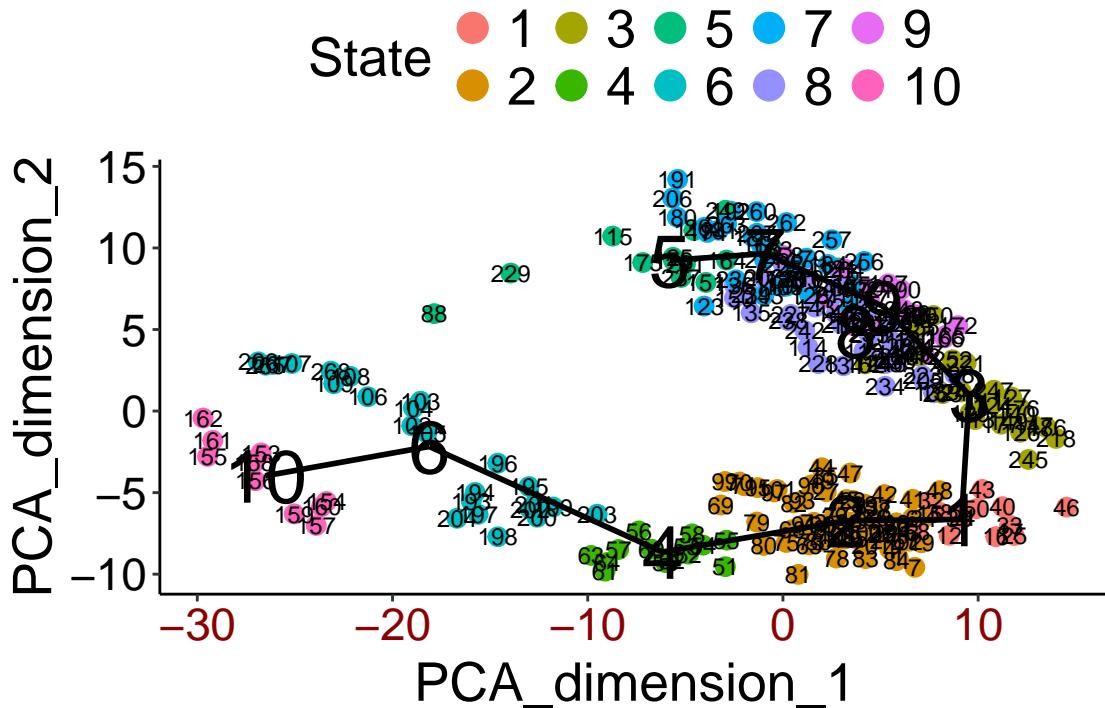
TSCAN combines clustering with pseudotime analysis. First it clusters the cells using `mclust`, which is based on a mixture of normal distributions. Then it builds a minimum spanning tree to connect the clusters. The branch of this tree that connects the largest number of clusters is the main branch which is used to determine pseudotime.

First we will try to use all genes to order the cells.

```

procdeng <- TSCAN::preprocess(deng)
colnames(procdeng) <- 1:ncol(deng)
dengclust <- TSCAN::exprmclust(procdeng, clusternum = 10)
TSCAN::plotmclust(dengclust)

```



```

dengorderTSCAN <- TSCAN::TSCANorder(dengclust, orderonly = FALSE)
pseudotime_order_tscan <- as.character(dengorderTSCAN$sample_name)
deng_SCE$pseudotime_order_tscan <- NA
deng_SCE$pseudotime_order_tscan[as.numeric(dengorderTSCAN$sample_name)] <-
  dengorderTSCAN$Pseudotime

```

Frustratingly, TSCAN only provides pseudotime values for 221 of 268 cells, silently returning missing values for non-assigned cells.

Again, we examine which timepoints have been assigned to each state:

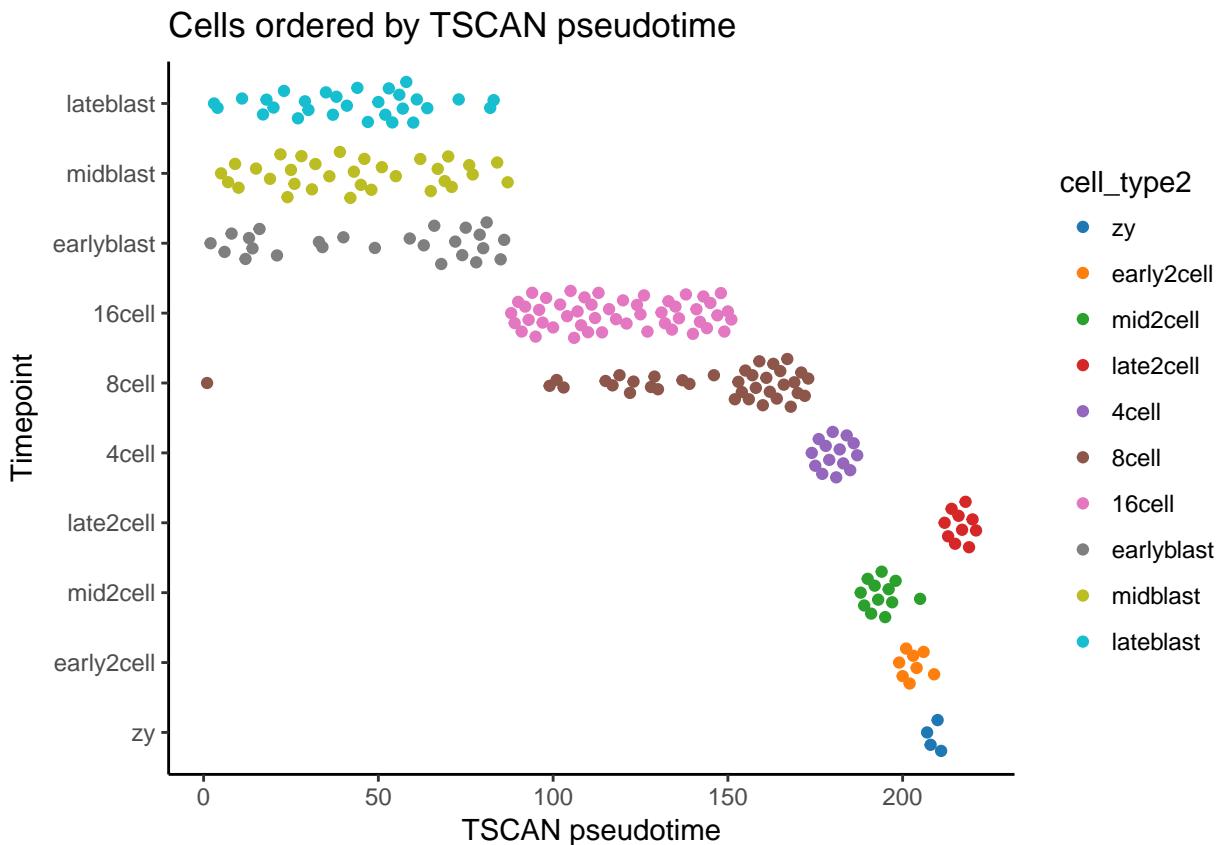
```
cellLabels[dengclust$clusterid == 10]
```

```

## [1] late2cell late2cell late2cell late2cell late2cell late2cell late2cell
## [8] late2cell late2cell late2cell
## 10 Levels: zy early2cell mid2cell late2cell 4cell 8cell ... lateblast
ggplot(as.data.frame(colData(deng_SCE)),
       aes(x = pseudotime_order_tscan,
            y = cell_type2, colour = cell_type2)) +
  geom_quasirandom(groupOnX = FALSE) +
  scale_color_tableau() + theme_classic() +
  xlab("TSCAN pseudotime") + ylab("Timepoint") +
  ggtitle("Cells ordered by TSCAN pseudotime")

```

## Warning: Removed 47 rows containing missing values (position quasirandom).



TSCAN gets the development trajectory the “wrong way around”, in the sense that later pseudotime values correspond to early timepoints and vice versa. This is not inherently a problem (it is easy enough to reverse the ordering to get the intuitive interpretation of pseudotime), but overall it would be a stretch to suggest that TSCAN performs better than PCA on this dataset. (As it is a PCA-based method, perhaps this is not entirely surprising.)

**Exercise 1** Compare results for different numbers of clusters (`clusternum`).

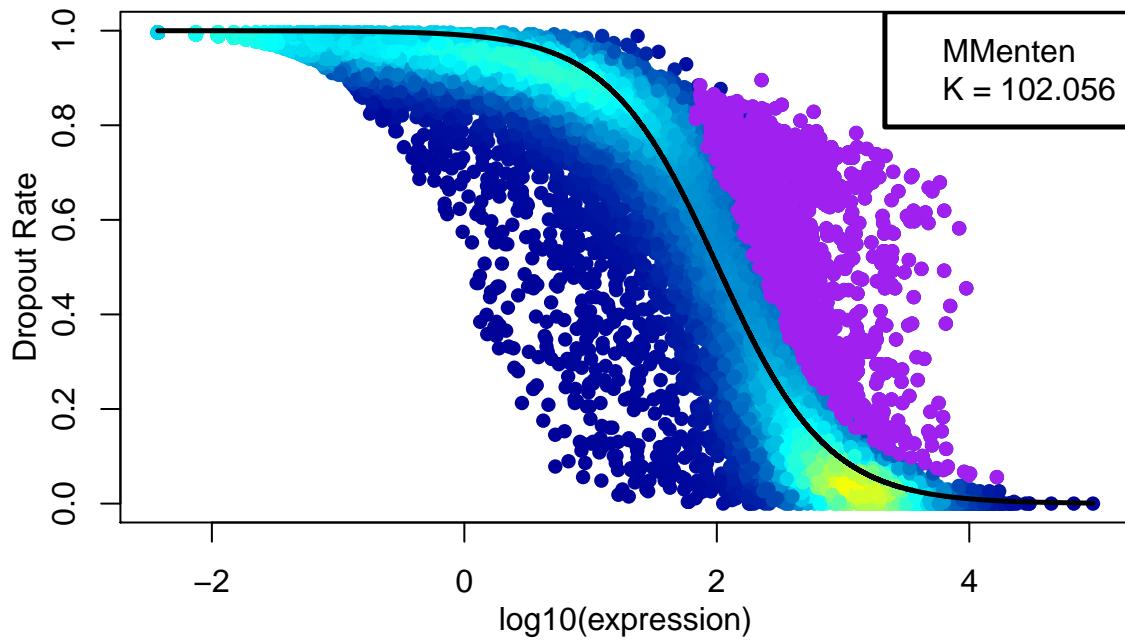
### 8.4.3 monocle

Monocle skips the clustering stage of TSCAN and directly builds a minimum spanning tree on a reduced dimension representation of the cells to connect all cells. Monocle then identifies the longest path in this tree to determine pseudotime. If the data contains diverging trajectories (i.e. one cell type differentiates into two different cell-types), monocle can identify these. Each of the resulting forked paths is defined as a separate cell state.

Unfortunately, Monocle does not work when all the genes are used, so we must carry out feature selection. First, we use M3Drop:

```
m3dGenes <- as.character(
  M3DropFeatureSelection(deng)$Gene
)
```

```
## Warning in bg__calc_variables(expr_mat): Warning: Removing 1134 undetected
## genes.
```

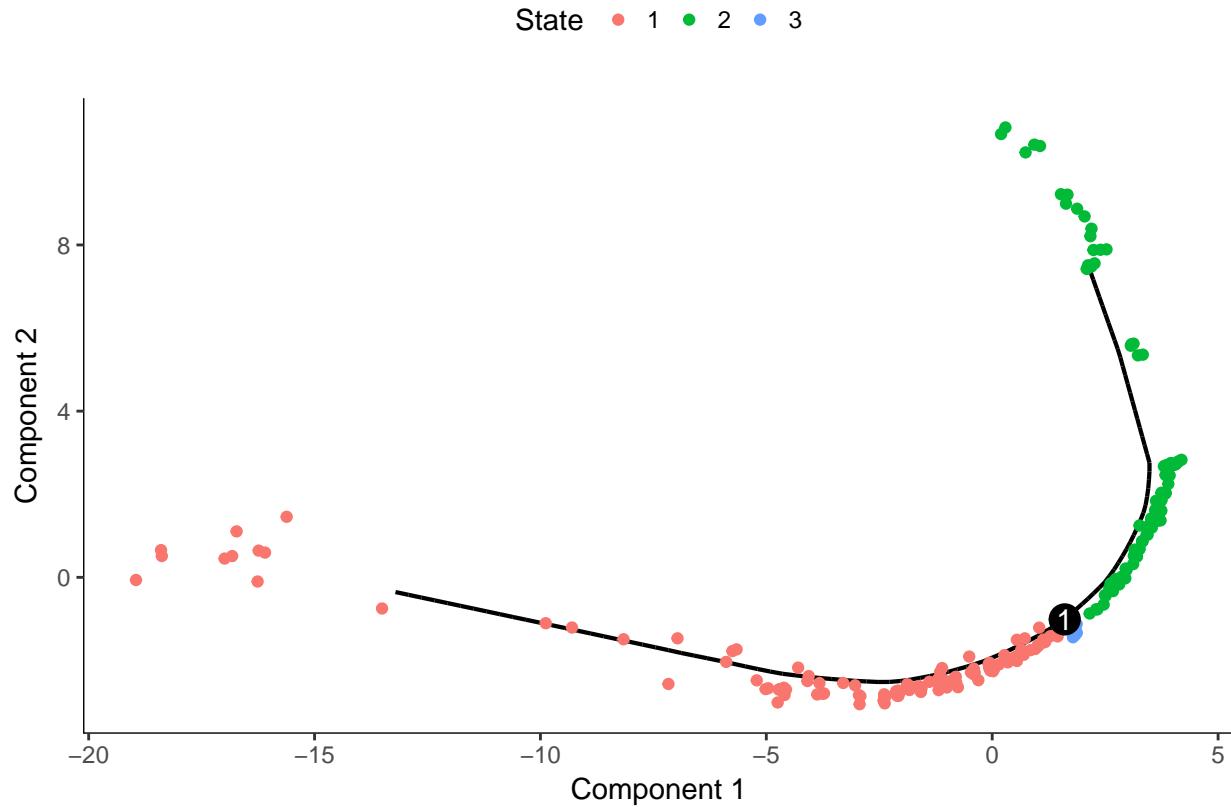


```
d <- deng[which(rownames(deng) %in% m3dGenes), ]
d <- d[!duplicated(rownames(d)), ]
```

Now run monocle:

```
colnames(d) <- 1:ncol(d)
geneNames <- rownames(d)
rownames(d) <- 1:nrow(d)
pd <- data.frame(timepoint = cellLabels)
pd <- new("AnnotatedDataFrame", data=pd)
fd <- data.frame(gene_short_name = geneNames)
fd <- new("AnnotatedDataFrame", data=fd)

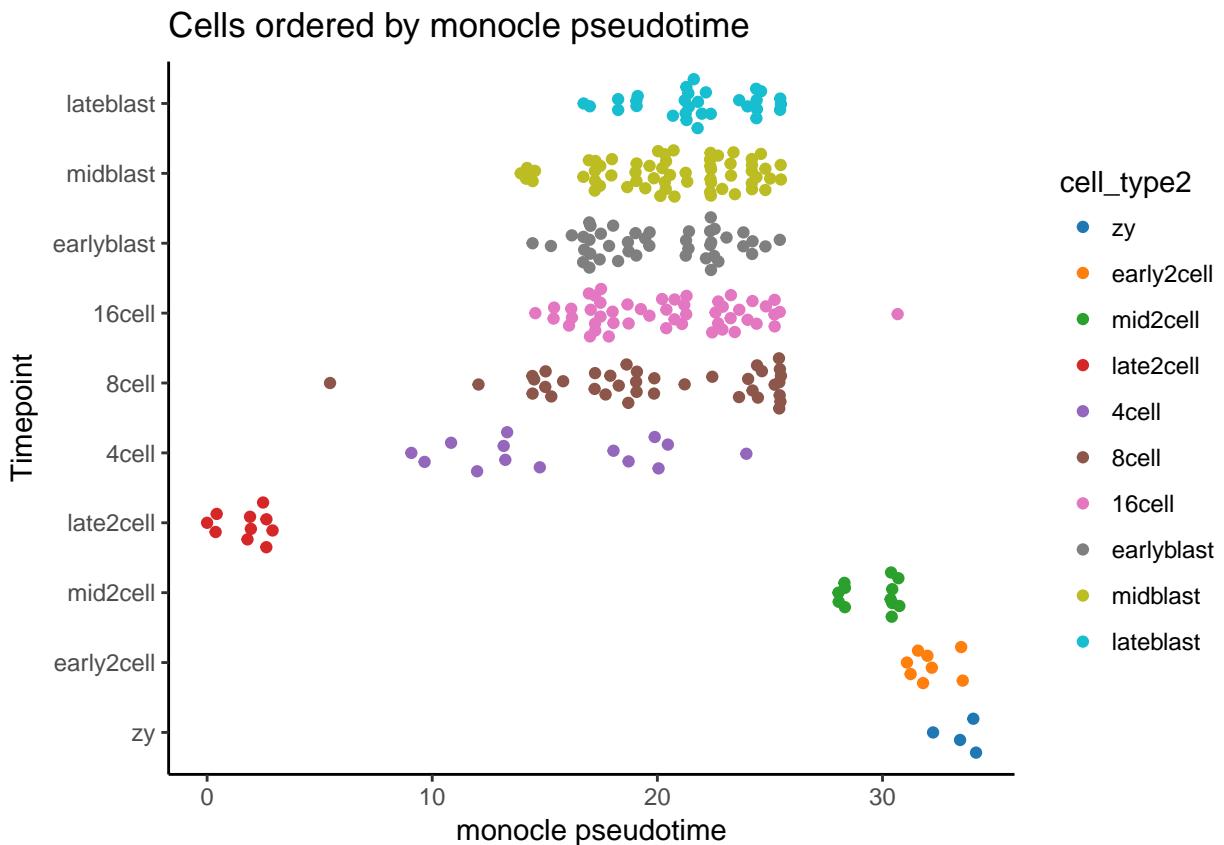
dCellData <- newCellDataSet(d, phenoData = pd, featureData = fd, expressionFamily = tobit())
dCellData <- setOrderingFilter(dCellData, which(geneNames %in% m3dGenes))
dCellData <- estimateSizeFactors(dCellData)
dCellDataSet <- reduceDimension(dCellData, pseudo_expr = 1)
dCellDataSet <- orderCells(dCellDataSet, reverse = FALSE)
plot_cell_trajectory(dCellDataSet)
```



```
# Store the ordering
pseudotime_monocle <-
  data.frame(
    Timepoint = phenoData(dCellDataSet)$timepoint,
    pseudotime = phenoData(dCellDataSet)$Pseudotime,
    State = phenoData(dCellDataSet)$State
  )
rownames(pseudotime_monocle) <- 1:ncol(d)
pseudotime_order_monocle <-
  rownames(pseudotime_monocle[order(pseudotime_monocle$pseudotime), ]])
```

We can again compare the inferred pseudotime to the known sampling timepoints.

```
deng_SCE$pseudotime_monocle <- pseudotime_monocle$pseudotime
ggplot(as.data.frame(colData(deng_SCE)),
  aes(x = pseudotime_monocle,
      y = cell_type2, colour = cell_type2)) +
  geom_quasirandom(groupOnX = FALSE) +
  scale_color_tableau() + theme_classic() +
  xlab("monocle pseudotime") + ylab("Timepoint") +
  ggtitle("Cells ordered by monocle pseudotime")
```



Monocle - at least with its default settings - performs poorly on these data. The “late2cell” group is completely separated from the “zy”, “early2cell” and “mid2cell” cells (though these are correctly ordered), and there is no separation at all of “4cell”, “8cell”, “16cell” or any blast cell groups.

#### 8.4.4 Diffusion maps

Diffusion maps were introduced by Ronald Coifman and Stephane Lafon, and the underlying idea is to assume that the data are samples from a diffusion process. The method infers the low-dimensional manifold by estimating the eigenvalues and eigenvectors for the diffusion operator related to the data.

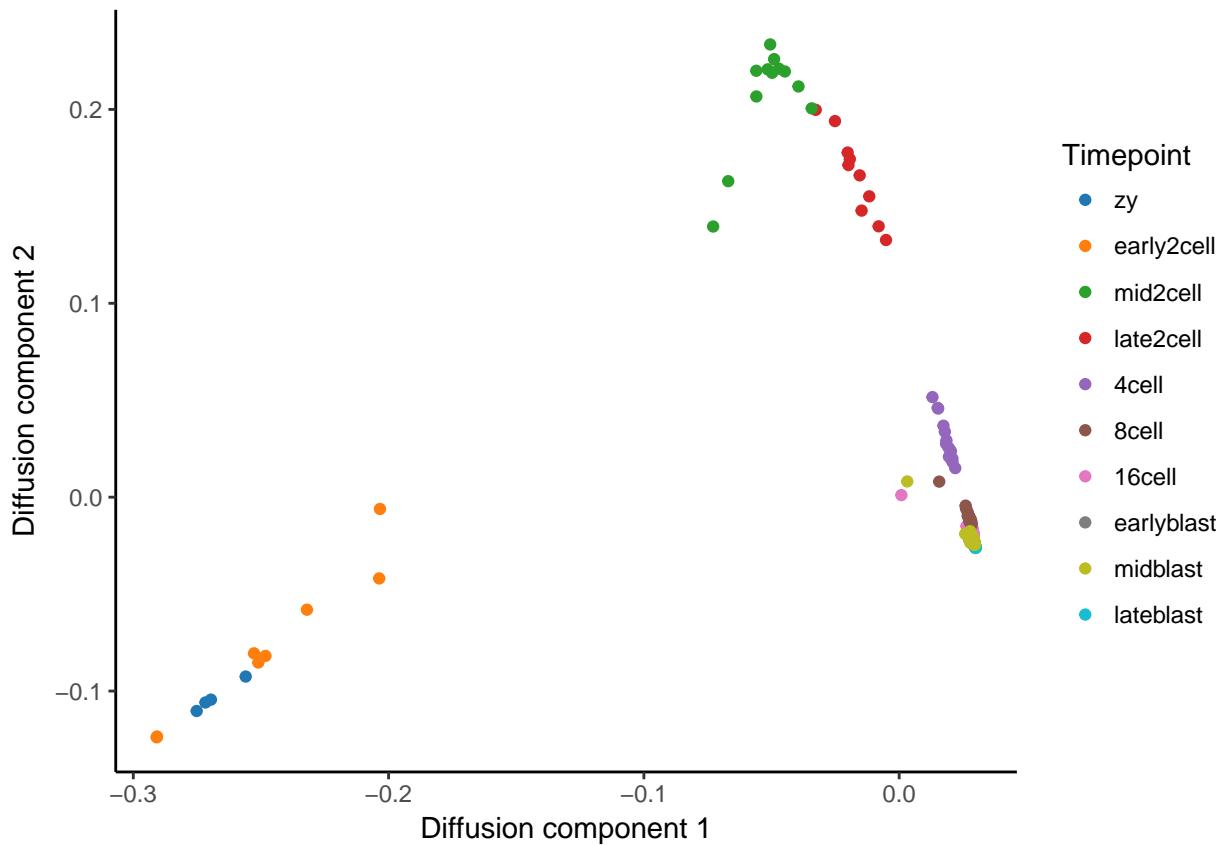
Haghverdi et al have applied the diffusion maps concept to the analysis of single-cell RNA-seq data to create an R package called *destiny*.

We will take the ranko prder of cells in the first diffusion map component as “diffusion map pseudotime” here.

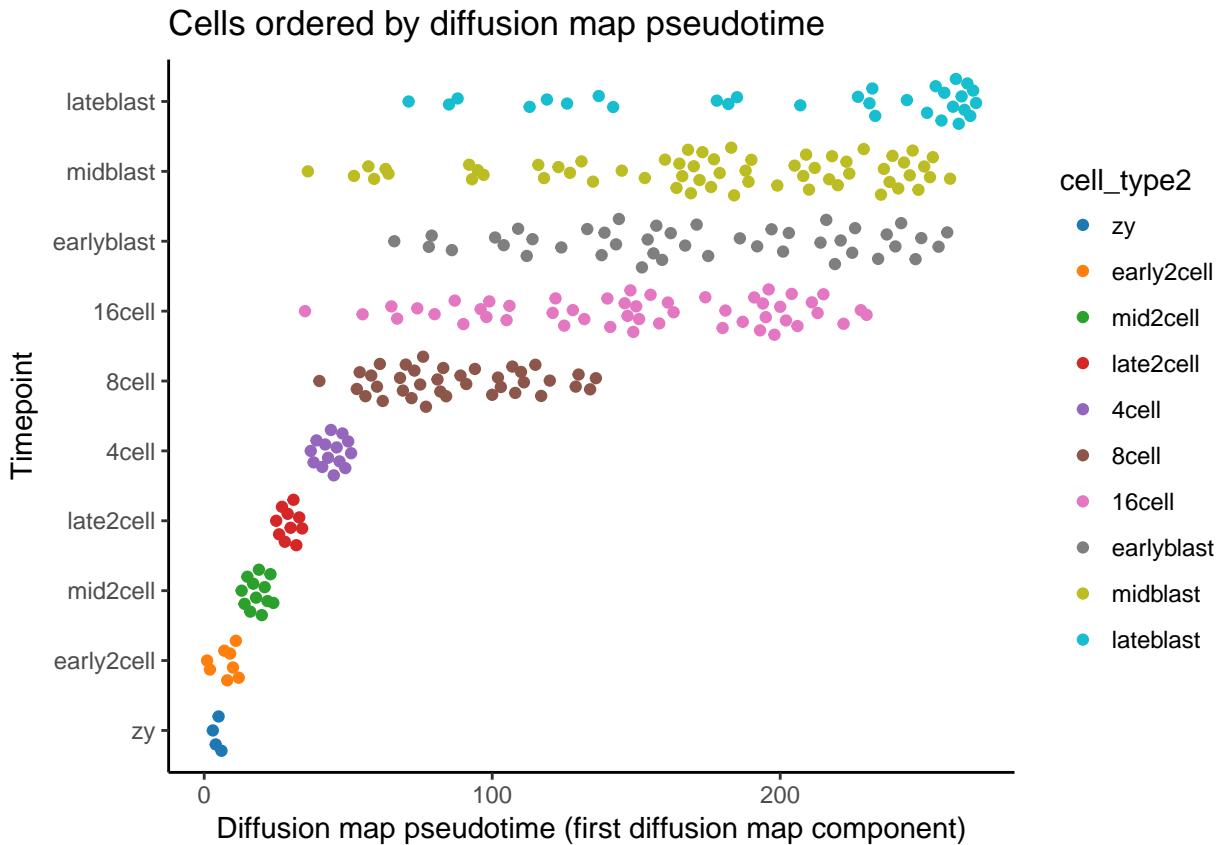
```
deng <- logcounts(deng_SCE)
colnames(deng) <- cellLabels
dm <- DiffusionMap(t(deng))

tmp <- data.frame(DC1 = eigenvectors(dm)[,1],
                   DC2 = eigenvectors(dm)[,2],
                   Timepoint = deng_SCE$cell_type2)
ggplot(tmp, aes(x = DC1, y = DC2, colour = Timepoint)) +
  geom_point() + scale_color_tableau() +
  xlab("Diffusion component 1") +
  ylab("Diffusion component 2") +
```

```
theme_classic()
```



```
deng_SCE$pseudotime_diffusionmap <- rank(eigenvalues(dm)[,1])
ggplot(as.data.frame(colData(deng_SCE)),
      aes(x = pseudotime_diffusionmap,
           y = cell_type2, colour = cell_type2)) +
  geom_quasirandom(groupOnX = FALSE) +
  scale_color_tableau() + theme_classic() +
  xlab("Diffusion map pseudotime (first diffusion map component)") +
  ylab("Timepoint") +
  ggtitle("Cells ordered by diffusion map pseudotime")
```



Like the other methods, using the first diffusion map component from destiny as pseudotime does a good job at ordering the early time-points (if we take high values as “earlier” in development), but it is unable to distinguish the later ones.

**Exercise 2** Do you get a better resolution between the later time points by considering additional eigenvectors?

**Exercise 3** How does the ordering change if you only use the genes identified by M3Drop?

#### 8.4.5 SLICER

The SLICER method is an algorithm for constructing trajectories that describe gene expression changes during a sequential biological process, just as Monocle and TSCAN are. SLICER is designed to capture highly nonlinear gene expression changes, automatically select genes related to the process, and detect multiple branch and loop features in the trajectory (Welch et al., 2016). The SLICER R package is available from its GitHub repository and can be installed from there using the `devtools` package.

We use the `select_genes` function in SLICER to automatically select the genes to use in building the cell trajectory. The function uses “neighbourhood variance” to identify genes that vary smoothly, rather than fluctuating randomly, across the set of cells. Following this, we determine which value of “*k*” (number of nearest neighbours) yields an embedding that most resembles a trajectory. Then we estimate the locally linear embedding of the cells.

```
library("l1e")
slicer_genes <- select_genes(t(deng))
k <- select_k(t(deng[slicer_genes,]), kmin = 30, kmax=60)

## finding neighbours
```

```
## calculating weights
## computing coordinates
## finding neighbours
## calculating weights
## computing coordinates
## finding neighbours
## calculating weights
## computing coordinates
## finding neighbours
## calculating weights
## computing coordinates
## finding neighbours
## calculating weights
## computing coordinates
## finding neighbours
## calculating weights
## computing coordinates
## finding neighbours
## calculating weights
## computing coordinates
## finding neighbours
## calculating weights
## computing coordinates

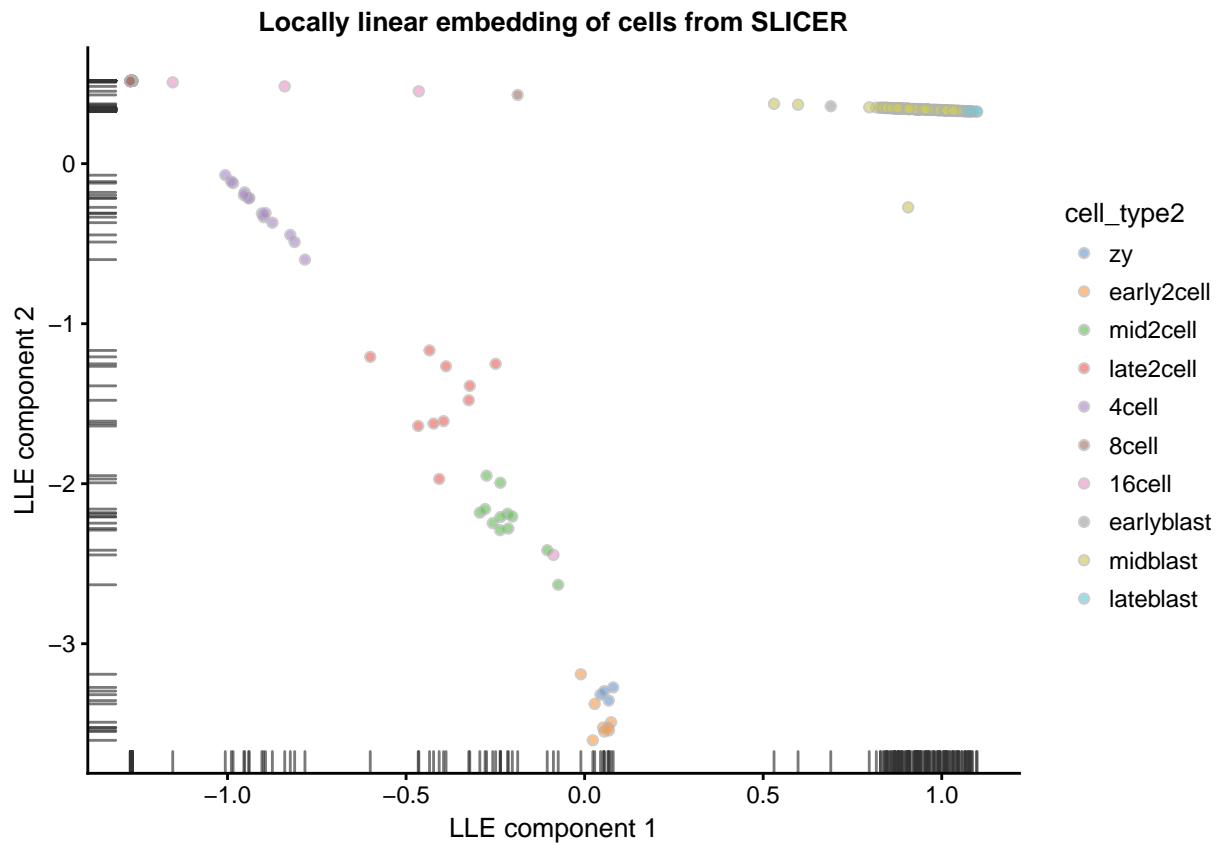
slicer_traj_llc <- lle(t(deng[slicer_genes,]), m = 2, k)$Y
```

```

## finding neighbours
## calculating weights
## computing coordinates

reducedDim(deng_SCE, "LLE") <- slicer_traj_lle
plotReducedDim(deng_SCE, use_dimred = "LLE", colour_by = "cell_type2") +
  xlab("LLE component 1") + ylab("LLE component 2") +
  ggtitle("Locally linear embedding of cells from SLICER")

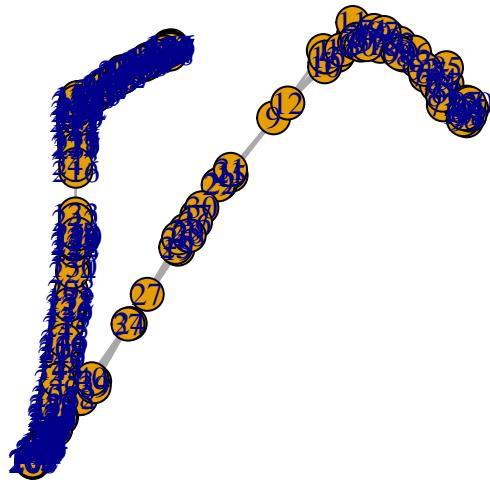
```



With the locally linear embedding computed we can construct a k-nearest neighbour graph that is fully connected. This plot displays a (yellow) circle for each cell, with the cell ID number overlaid in blue. Here we show the graph computed using 10 nearest neighbours. Here, SLICER appears to detect one major trajectory with one branch.

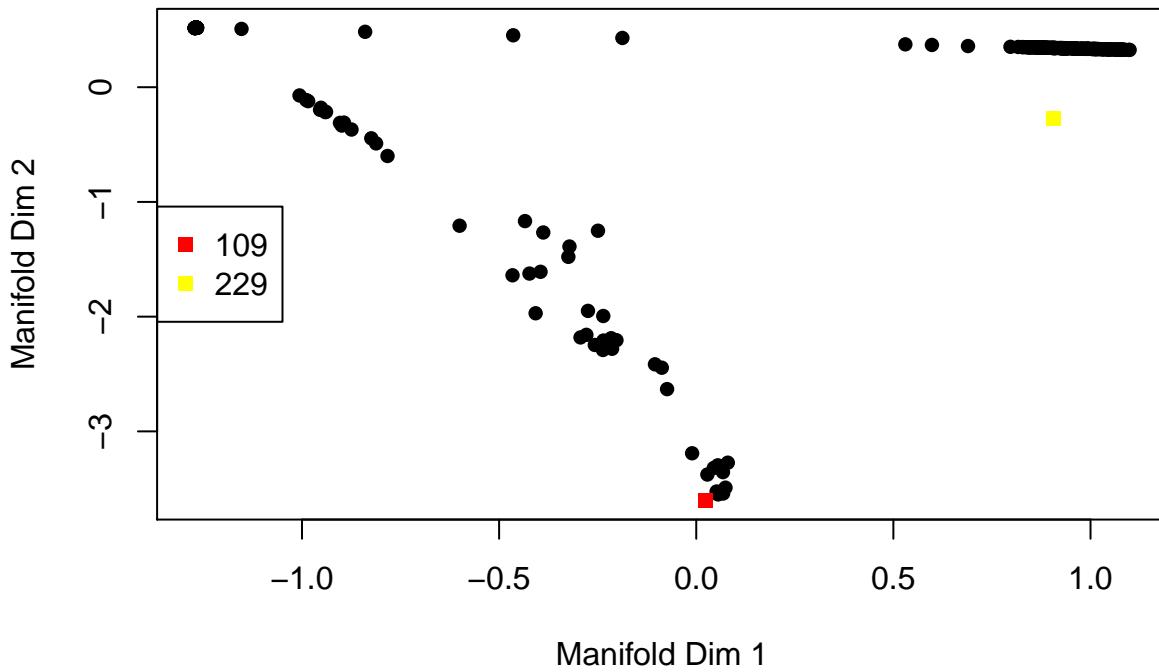
```
slicer_traj_graph <- conn_knn_graph(slicer_traj_lle, 10)
plot(slicer_traj_graph, main = "Fully connected kNN graph from SLICER")
```

**Fully connected kNN graph from SLICER**



From this graph we can identify “extreme” cells that are candidates for start/end cells in the trajectory.

```
ends <- find_extreme_cells(slicer_traj_graph, slicer_traj_lle)
```



```
start <- ends[1]
```

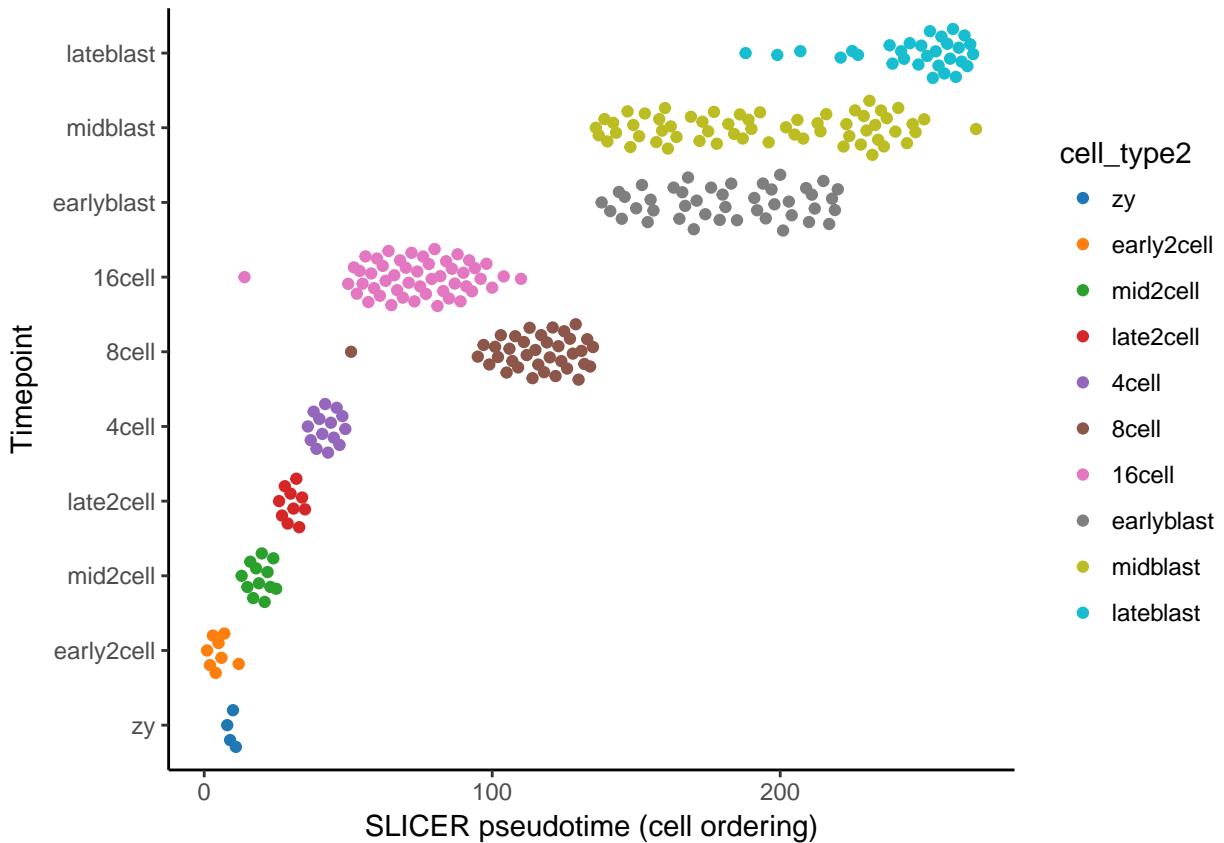
Having defined a start cell we can order the cells in the estimated pseudotime.

```
pseudotime_order_slicer <- cell_order(slicer_traj_graph, start)
branches <- assign_branches(slicer_traj_graph, start)

pseudotime_slicer <-
  data.frame(
    Timepoint = cellLabels,
    pseudotime = NA,
    State = branches
  )
pseudotime_slicer$pseudotime[pseudotime_order_slicer] <-
  1:length(pseudotime_order_slicer)
deng_SCE$pseudotime_slicer <- pseudotime_slicer$pseudotime
```

We can again compare the inferred pseudotime to the known sampling timepoints. SLICER does not provide a pseudotime value per se, just an ordering of cells.

```
ggplot(as.data.frame(colData(deng_SCE)),
  aes(x = pseudotime_slicer,
      y = cell_type2, colour = cell_type2)) +
  geom_quasirandom(groupOnX = FALSE) +
  scale_color_tableau() + theme_classic() +
  xlab("SLICER pseudotime (cell ordering)") +
  ylab("Timepoint") +
  theme_classic()
```



Like the previous method, SLICER here provides a good ordering for the early time points. It places “16cell” cells before “8cell” cells, but provides better ordering for blast cells than many of the earlier methods.

**Exercise 4** How do the results change for different k? (e.g. k = 5) What about changing the number of nearest neighbours in the call to `conn_knn_graph`?

**Exercise 5** How does the ordering change if you use a different set of genes from those chosen by SLICER (e.g. the genes identified by M3Drop)?

#### 8.4.6 Ouija

Ouija (<http://kieranrcampbell.github.io/ouija/>) takes a different approach from the pseudotime estimation methods we have looked at so far. Earlier methods have all been “unsupervised”, which is to say that apart from perhaps selecting informative genes we do not supply the method with any prior information about how we expect certain genes or the trajectory as a whole to behave.

Ouija, in contrast, is a probabilistic framework that allows for interpretable learning of single-cell pseudotimes using only small panels of marker genes. This method:

- infers pseudotimes from a small number of marker genes letting you understand why the pseudotimes have been learned in terms of those genes;
- provides parameter estimates (with uncertainty) for interpretable gene regulation behaviour (such as the peak time or the upregulation time);
- has a Bayesian hypothesis test to find genes regulated before others along the trajectory;
- identifies metastable states, ie discrete cell types along the continuous trajectory.

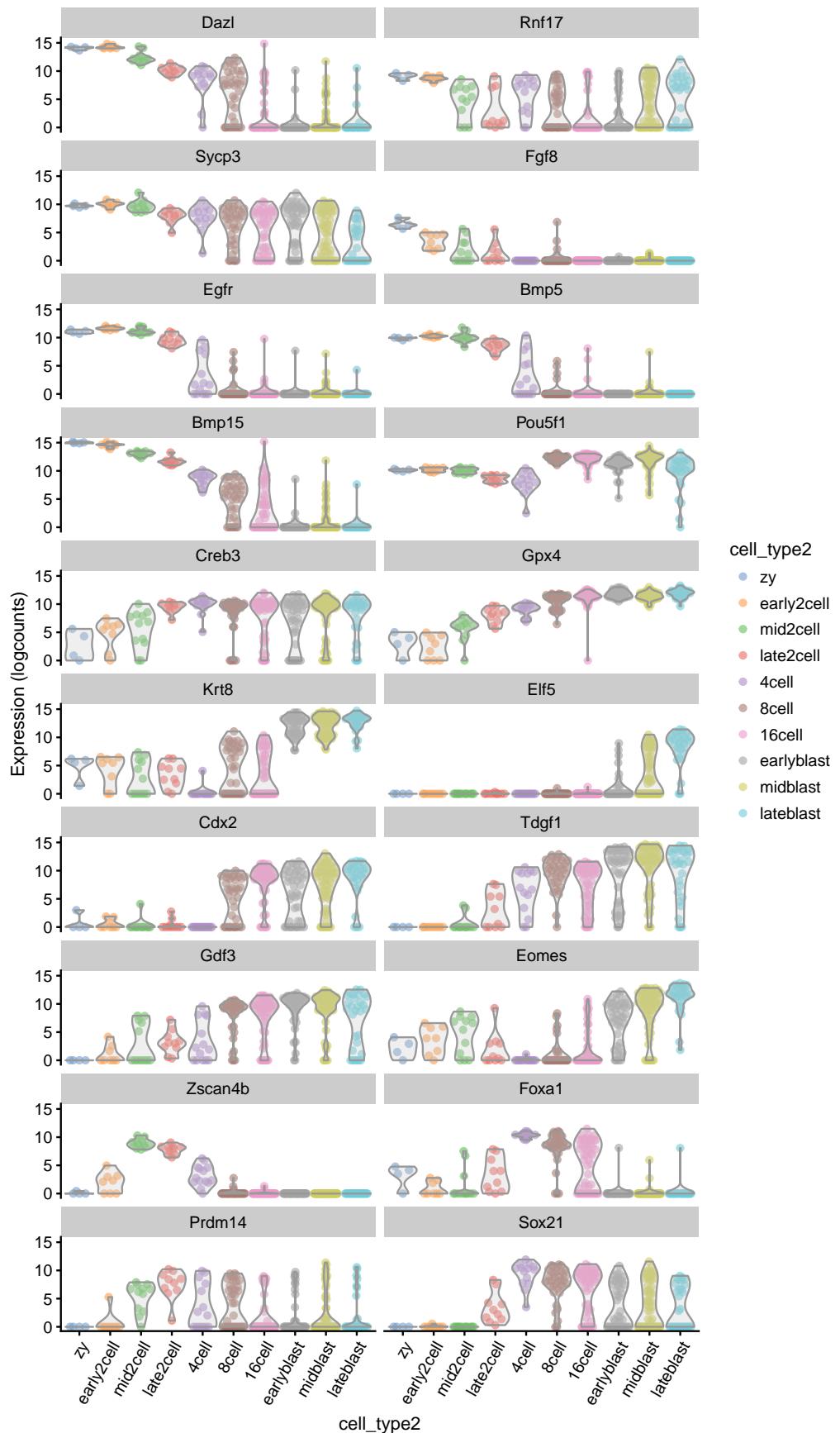
We will supply the following marker genes to Ouija (with timepoints where they are expected to be highly expressed):

- Early timepoints: Dazl, Rnf17, Sycp3, Nanog, Pou5f1, Fgf8, Egfr, Bmp5, Bmp15
- Mid timepoints: Zscan4b, Foxa1, Prdm14, Sox21
- Late timepoints: Creb3, Gpx4, Krt8, Elf5, Eomes, Cdx2, Tdgf1, Gdf3

With Ouija we can model genes as either exhibiting monotonic up or down regulation (known as switch-like behaviour), or transient behaviour where the gene briefly peaks. By default, Ouija assumes all genes exhibit switch-like behaviour (the authors assure us not to worry if we get it wrong - the noise model means incorrectly specifying a transient gene as switch-like has minimal effect).

Here we can “cheat” a little and check that our selected marker genes do actually identify different timepoints of the differentiation process.

```
ouija_markers_down <- c("Dazl", "Rnf17", "Sycp3", "Fgf8",
                         "Egfr", "Bmp5", "Bmp15", "Pou5f1")
ouija_markers_up <- c("Creb3", "Gpx4", "Krt8", "Elf5", "Cdx2",
                      "Tdgf1", "Gdf3", "Eomes")
ouija_markers_transient <- c("Zscan4b", "Foxa1", "Prdm14", "Sox21")
ouija_markers <- c(ouija_markers_down, ouija_markers_up,
                  ouija_markers_transient)
plotExpression(deng_SCE, ouija_markers, x = "cell_type2", colour_by = "cell_type2") +
  theme(axis.text.x = element_text(angle = 60, hjust = 1))
```



In order to fit the pseudotimes we simply call `ouija`, passing in the expected response types. Note that if no response types are provided then they are all assumed to be switch-like by default, which we will do here. The input to Ouija can be a cell-by-gene matrix of non-negative expression values, or an ExpressionSet object, or, happily, by selecting the `logcounts` values from a SingleCellExperiment object.

We can apply prior information about whether genes are up- or down-regulated across the differentiation process, and also provide prior information about when the switch in expression or a peak in expression is likely to occur.

We can fit the Ouija model using either:

- Hamiltonian Monte Carlo (HMC) - full MCMC inference where gradient information of the log-posterior is used to “guide” the random walk through the parameter space, or
- Automatic Differentiation Variational Bayes (ADVI or simply VI) - approximate inference where the KL divergence to an approximate distribution is minimised.

In general, HMC will provide more accurate inference with approximately correct posterior variance for all parameters. However, VB is orders of magnitude quicker than HMC and while it may underestimate posterior variance, the Ouija authors suggest that anecdotally it often performs as well as HMC for discovering posterior pseudotimes.

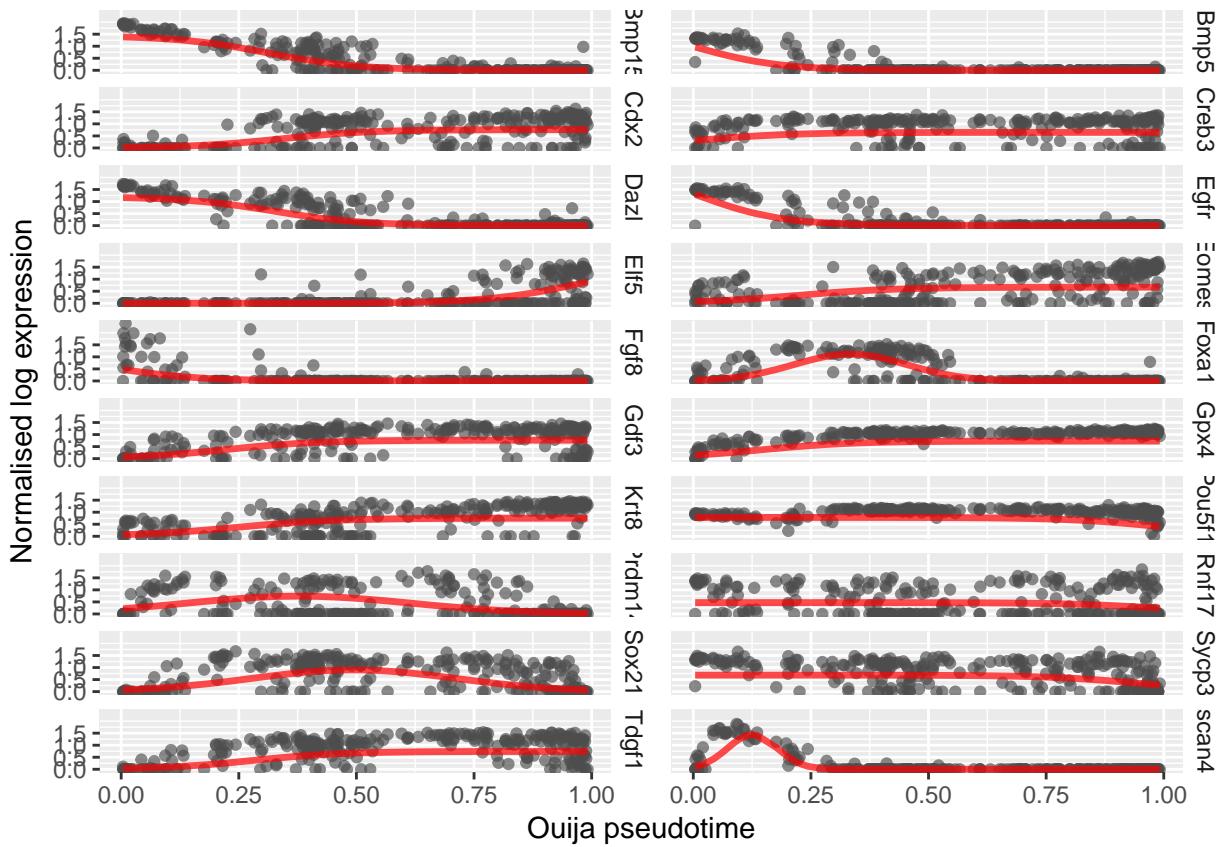
To help the Ouija model, we provide it with prior information about the strength of switches for up- and down-regulated genes. By setting switch strength to -10 for down-regulated genes and 10 for up-regulated genes with a prior strength standard deviation of 0.5 we are telling the model that we are confident about the expected behaviour of these genes across the differentiation process.

```
options(mc.cores = parallel::detectCores())
response_type <- c(rep("switch", length(ouija_markers_down) +
                      length(ouija_markers_up)),
                      rep("transient", length(ouija_markers_transient)))
switch_strengths <- c(rep(-10, length(ouija_markers_down)),
                      rep(10, length(ouija_markers_up)))
switch_strength_sd <- c(rep(0.5, length(ouija_markers_down)),
                         rep(0.5, length(ouija_markers_up)))
garbage <- capture.output(
  oui_vb <- ouija(deng_SCE[ouija_markers, ],
                  single_cell_experiment_assay = "logcounts",
                  response_type = response_type,
                  switch_strengths = switch_strengths,
                  switch_strength_sd = switch_strength_sd,
                  inference_type = "vb")
)
print(oui_vb)

## A Ouija fit with 268 cells and 20 marker genes
## Inference type: Variational Bayes
## (Gene behaviour) Switch/transient: 16 / 4
```

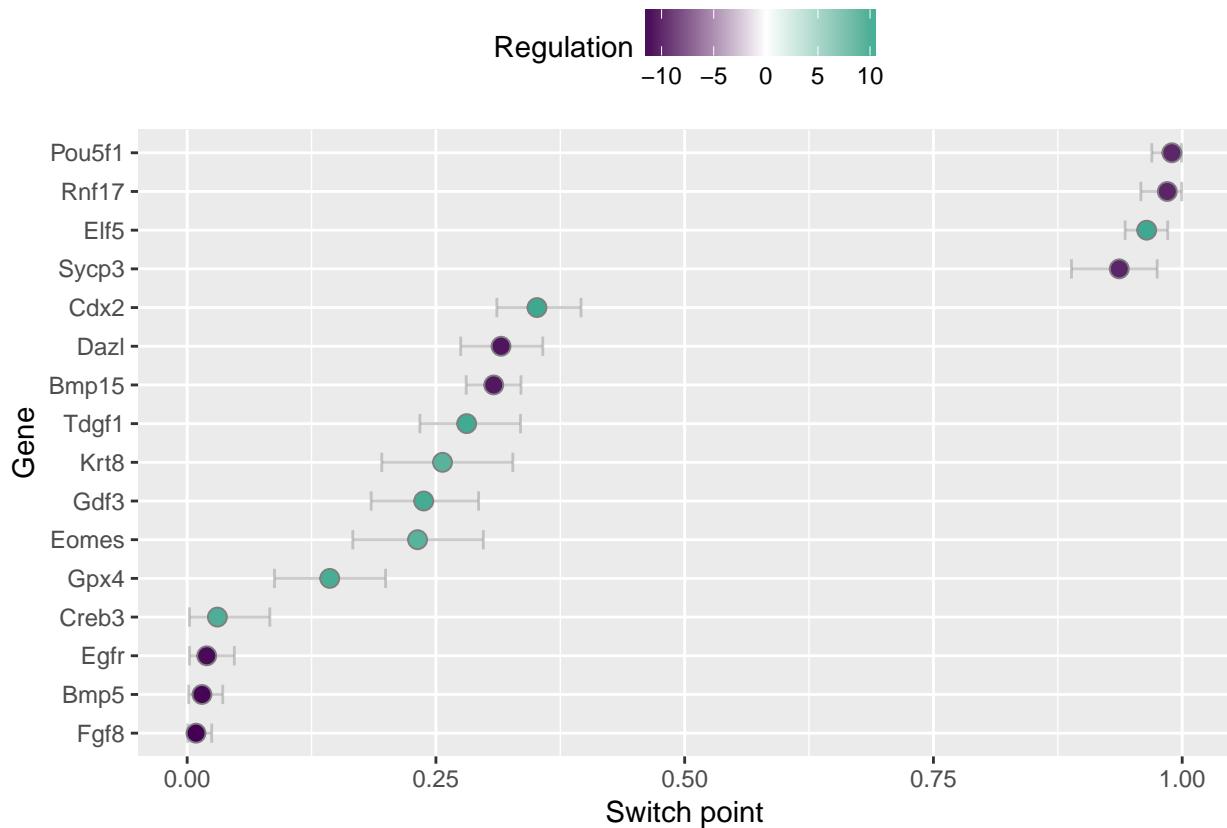
We can plot the gene expression over pseudotime along with the maximum a posteriori (MAP) estimates of the mean function (the sigmoid or Gaussian transient function) using the `plot_expression` function.

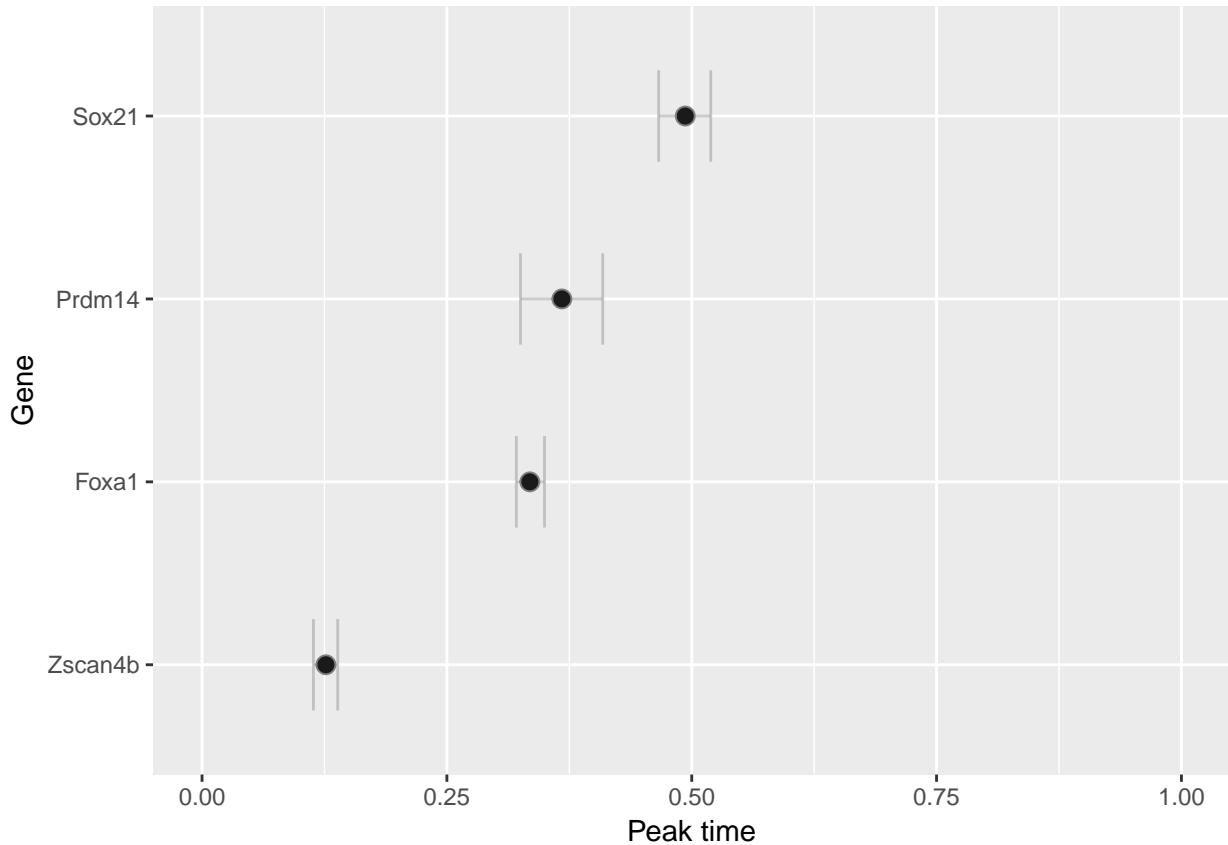
```
plot_expression(oui_vb)
```



We can also visualise when in the trajectory gene regulation behaviour occurs, either in the form of the switch time or the peak time (for switch-like or transient genes) using the `plot_switch_times` and `plot_transient_times` functions:

```
plot_switch_times(oui_vb)
```





Identify metastable states using consistency matrices.

```
cmo <- consistency_matrix(oui_vb)
plot_consistency(oui_vb)

## Warning in grid.Call(C_stringMetric, as.graphicsAnnot(x$label)): font
## metrics unknown for Unicode character U+2192

## Warning in grid.Call(C_stringMetric, as.graphicsAnnot(x$label)): font
## metrics unknown for Unicode character U+2192

## Warning in grid.Call(C_stringMetric, as.graphicsAnnot(x$label)): conversion
## failure on 'Pseudotime order →' in 'mbcsToSbcs': dot substituted for <e2>

## Warning in grid.Call(C_stringMetric, as.graphicsAnnot(x$label)): conversion
## failure on 'Pseudotime order →' in 'mbcsToSbcs': dot substituted for <86>

## Warning in grid.Call(C_stringMetric, as.graphicsAnnot(x$label)): conversion
## failure on 'Pseudotime order →' in 'mbcsToSbcs': dot substituted for <92>

## Warning in grid.Call(C_stringMetric, as.graphicsAnnot(x$label)): font
## metrics unknown for Unicode character U+2192

## Warning in grid.Call(C_stringMetric, as.graphicsAnnot(x$label)): font
## metrics unknown for Unicode character U+2192

## Warning in grid.Call(C_stringMetric, as.graphicsAnnot(x$label)): conversion
## failure on 'Pseudotime order →' in 'mbcsToSbcs': dot substituted for <e2>

## Warning in grid.Call(C_stringMetric, as.graphicsAnnot(x$label)): conversion
```

```
## failure on 'Pseudotime order →' in 'mbcsToSbcs': dot substituted for <86>
## Warning in grid.Call(C_stringMetric, as.graphicsAnnot(x$label)): conversion
## failure on 'Pseudotime order →' in 'mbcsToSbcs': dot substituted for <92>
## Warning in grid.Call(C_textBounds, as.graphicsAnnot(x$label), x$x, x$y, :
## conversion failure on 'Pseudotime order →' in 'mbcsToSbcs': dot substituted
## for <e2>
## Warning in grid.Call(C_textBounds, as.graphicsAnnot(x$label), x$x, x$y, :
## conversion failure on 'Pseudotime order →' in 'mbcsToSbcs': dot substituted
## for <86>
## Warning in grid.Call(C_textBounds, as.graphicsAnnot(x$label), x$x, x$y, :
## conversion failure on 'Pseudotime order →' in 'mbcsToSbcs': dot substituted
## for <92>
## Warning in grid.Call(C_textBounds, as.graphicsAnnot(x$label), x$x, x$y, :
## conversion failure on 'Pseudotime order →' in 'mbcsToSbcs': dot substituted
## for <e2>
## Warning in grid.Call(C_textBounds, as.graphicsAnnot(x$label), x$x, x$y, :
## conversion failure on 'Pseudotime order →' in 'mbcsToSbcs': dot substituted
## for <86>
## Warning in grid.Call(C_textBounds, as.graphicsAnnot(x$label), x$x, x$y, :
## conversion failure on 'Pseudotime order →' in 'mbcsToSbcs': dot substituted
## for <92>
## Warning in grid.Call(C_textBounds, as.graphicsAnnot(x$label), x$x, x$y, :
## conversion failure on 'Pseudotime order →' in 'mbcsToSbcs': dot substituted
## for <e2>
## Warning in grid.Call(C_textBounds, as.graphicsAnnot(x$label), x$x, x$y, :
## conversion failure on 'Pseudotime order →' in 'mbcsToSbcs': dot substituted
## for <86>
## Warning in grid.Call(C_textBounds, as.graphicsAnnot(x$label), x$x, x$y, :
## conversion failure on 'Pseudotime order →' in 'mbcsToSbcs': dot substituted
## for <92>
## Warning in grid.Call(C_textBounds, as.graphicsAnnot(x$label), x$x, x$y, :
## conversion failure on 'Pseudotime order →' in 'mbcsToSbcs': dot substituted
## for <e2>
## Warning in grid.Call(C_textBounds, as.graphicsAnnot(x$label), x$x, x$y, :
## conversion failure on 'Pseudotime order →' in 'mbcsToSbcs': dot substituted
## for <86>
## Warning in grid.Call(C_textBounds, as.graphicsAnnot(x$label), x$x, x$y, :
## conversion failure on 'Pseudotime order →' in 'mbcsToSbcs': dot substituted
## for <92>
## Warning in grid.Call(C_textBounds, as.graphicsAnnot(x$label), x$x, x$y, :
## conversion failure on 'Pseudotime order →' in 'mbcsToSbcs': dot substituted
## for <e2>
## Warning in grid.Call(C_textBounds, as.graphicsAnnot(x$label), x$x, x$y, :
## conversion failure on 'Pseudotime order →' in 'mbcsToSbcs': dot substituted
## for <86>
```



```
## conversion failure on 'Pseudotime order →' in 'mbcsToSbcs': dot substituted
## for <92>

## Warning in grid.Call(C_textBounds, as.graphicsAnnot(x$label), x$x, x$y, :
## conversion failure on 'Pseudotime order →' in 'mbcsToSbcs': dot substituted
## for <e2>

## Warning in grid.Call(C_textBounds, as.graphicsAnnot(x$label), x$x, x$y, :
## conversion failure on 'Pseudotime order →' in 'mbcsToSbcs': dot substituted
## for <86>

## Warning in grid.Call(C_textBounds, as.graphicsAnnot(x$label), x$x, x$y, :
## conversion failure on 'Pseudotime order →' in 'mbcsToSbcs': dot substituted
## for <92>

## Warning in grid.Call(C_textBounds, as.graphicsAnnot(x$label), x$x, x$y, :
## conversion failure on 'Pseudotime order →' in 'mbcsToSbcs': dot substituted
## for <e2>

## Warning in grid.Call(C_textBounds, as.graphicsAnnot(x$label), x$x, x$y, :
## conversion failure on 'Pseudotime order →' in 'mbcsToSbcs': dot substituted
## for <86>

## Warning in grid.Call(C_textBounds, as.graphicsAnnot(x$label), x$x, x$y, :
## conversion failure on 'Pseudotime order →' in 'mbcsToSbcs': dot substituted
## for <92>

## Warning in grid.Call(graphics(C_text, as.graphicsAnnot(x$label), x$x, x
## $y, : conversion failure on 'Pseudotime order →' in 'mbcsToSbcs': dot
## substituted for <e2>

## Warning in grid.Call(graphics(C_text, as.graphicsAnnot(x$label), x$x, x
## $y, : conversion failure on 'Pseudotime order →' in 'mbcsToSbcs': dot
## substituted for <86>

## Warning in grid.Call(graphics(C_text, as.graphicsAnnot(x$label), x$x, x
## $y, : conversion failure on 'Pseudotime order →' in 'mbcsToSbcs': dot
## substituted for <92>

## Warning in grid.Call(C_textBounds, as.graphicsAnnot(x$label), x$x, x$y, :
## conversion failure on 'Pseudotime order →' in 'mbcsToSbcs': dot substituted
## for <e2>

## Warning in grid.Call(C_textBounds, as.graphicsAnnot(x$label), x$x, x$y, :
## conversion failure on 'Pseudotime order →' in 'mbcsToSbcs': dot substituted
## for <86>

## Warning in grid.Call(C_textBounds, as.graphicsAnnot(x$label), x$x, x$y, :
## conversion failure on 'Pseudotime order →' in 'mbcsToSbcs': dot substituted
## for <92>

## Warning in grid.Call(C_textBounds, as.graphicsAnnot(x$label), x$x, x$y, :
## conversion failure on 'Pseudotime order →' in 'mbcsToSbcs': dot substituted
## for <e2>

## Warning in grid.Call(C_textBounds, as.graphicsAnnot(x$label), x$x, x$y, :
## conversion failure on 'Pseudotime order →' in 'mbcsToSbcs': dot substituted
## for <86>

## Warning in grid.Call(C_textBounds, as.graphicsAnnot(x$label), x$x, x$y, :
## conversion failure on 'Pseudotime order →' in 'mbcsToSbcs': dot substituted
```

```

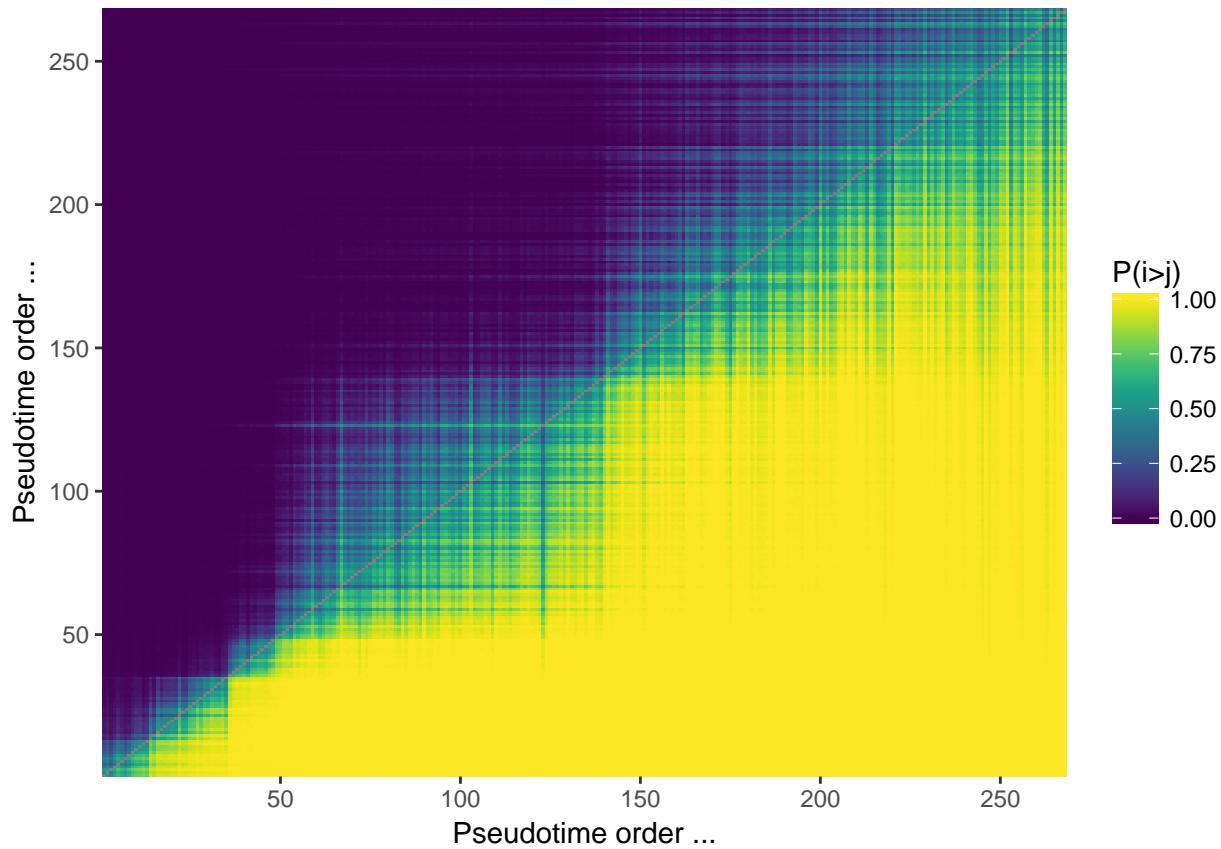
## for <92>

## Warning in grid.Call.graphics(C_text, as.graphicsAnnot(x$label), x$x, x
## $y, : conversion failure on 'Pseudotime order →' in 'mbcsToSbcs': dot
## substituted for <e2>

## Warning in grid.Call.graphics(C_text, as.graphicsAnnot(x$label), x$x, x
## $y, : conversion failure on 'Pseudotime order →' in 'mbcsToSbcs': dot
## substituted for <86>

## Warning in grid.Call.graphics(C_text, as.graphicsAnnot(x$label), x$x, x
## $y, : conversion failure on 'Pseudotime order →' in 'mbcsToSbcs': dot
## substituted for <92>

```



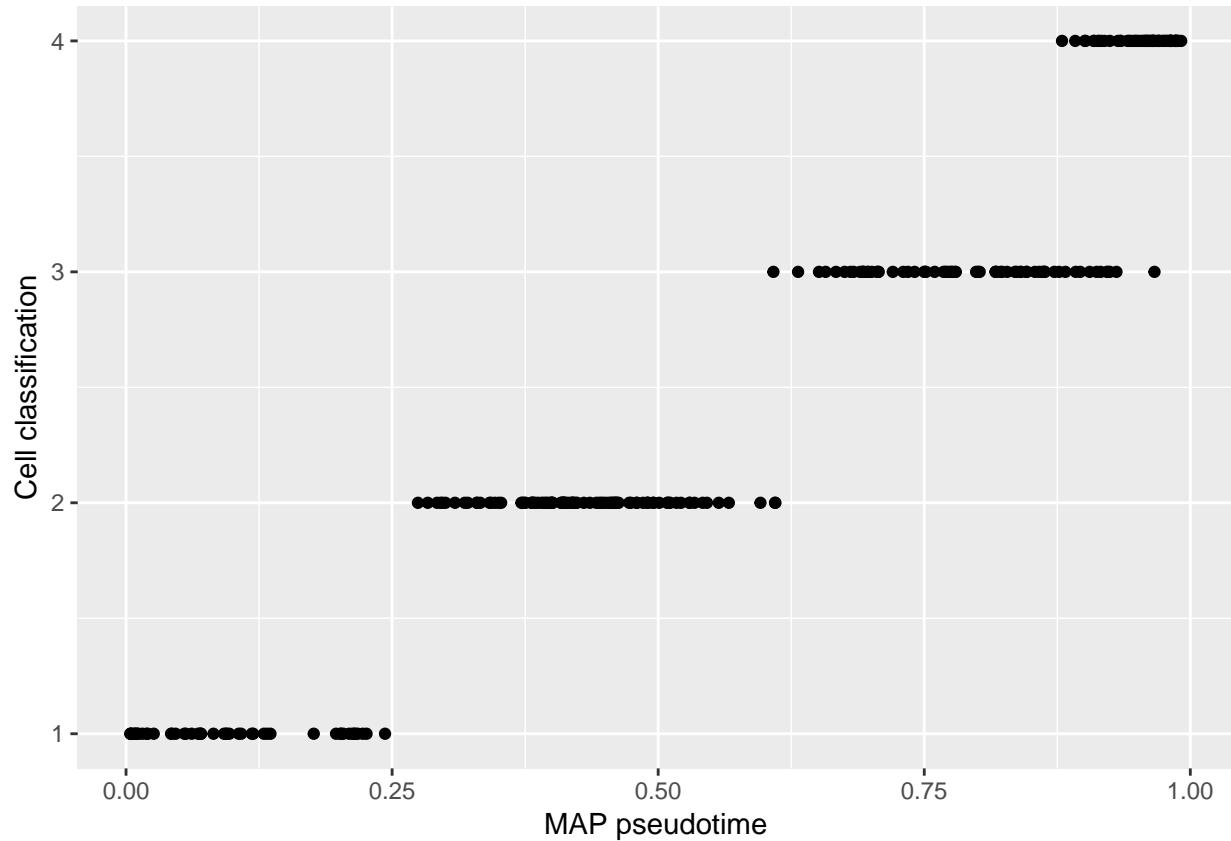
```

cell_classifications <- cluster_consistency(cmo)

map_pst <- map_pseudotime(oui_vb)
ouija_pseudotime <- data.frame(map_pst, cell_classifications)

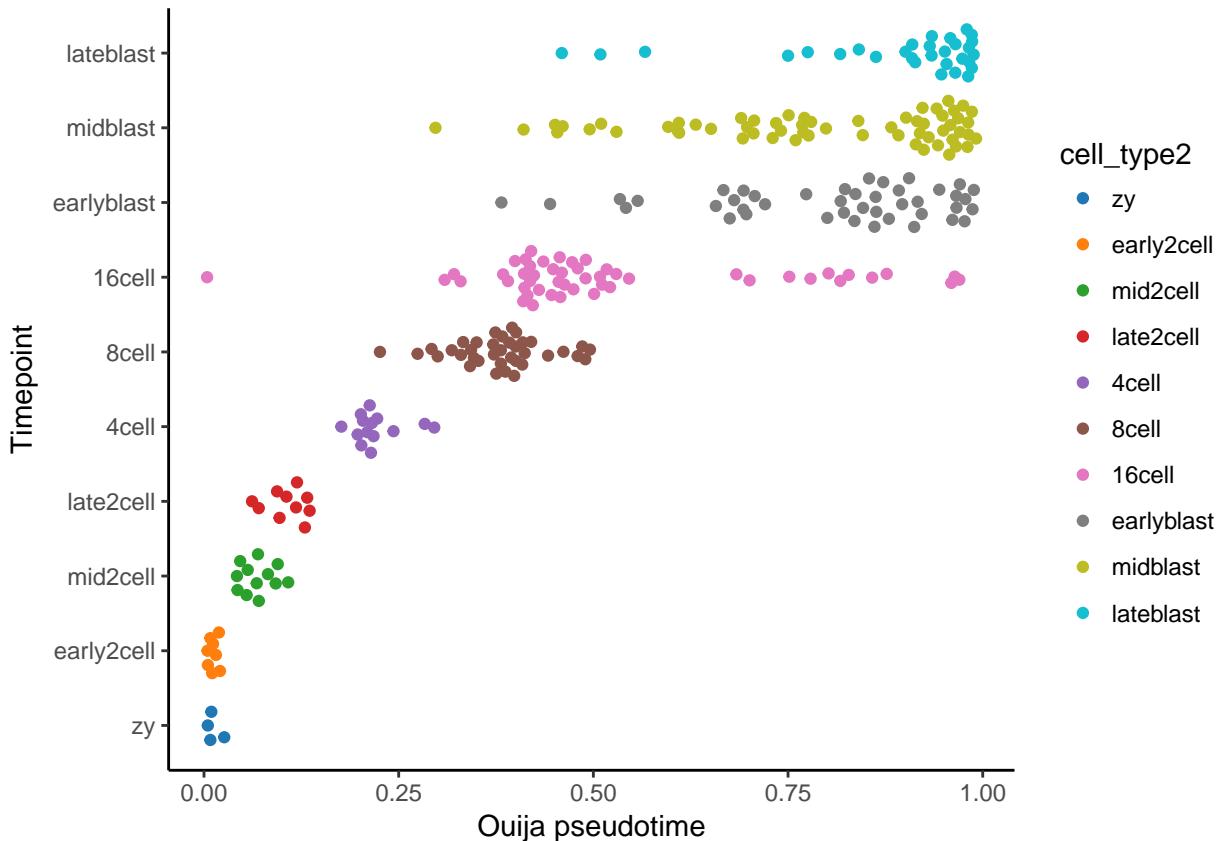
ggplot(ouija_pseudotime, aes(x = map_pst, y = cell_classifications)) +
  geom_point() +
  xlab("MAP pseudotime") +
  ylab("Cell classification")

```



```
deng_SCE$pseudotime_ouija <- ouija_pseudotime$map_pst
deng_SCE$ouija_cell_class <- ouija_pseudotime$cell_classifications

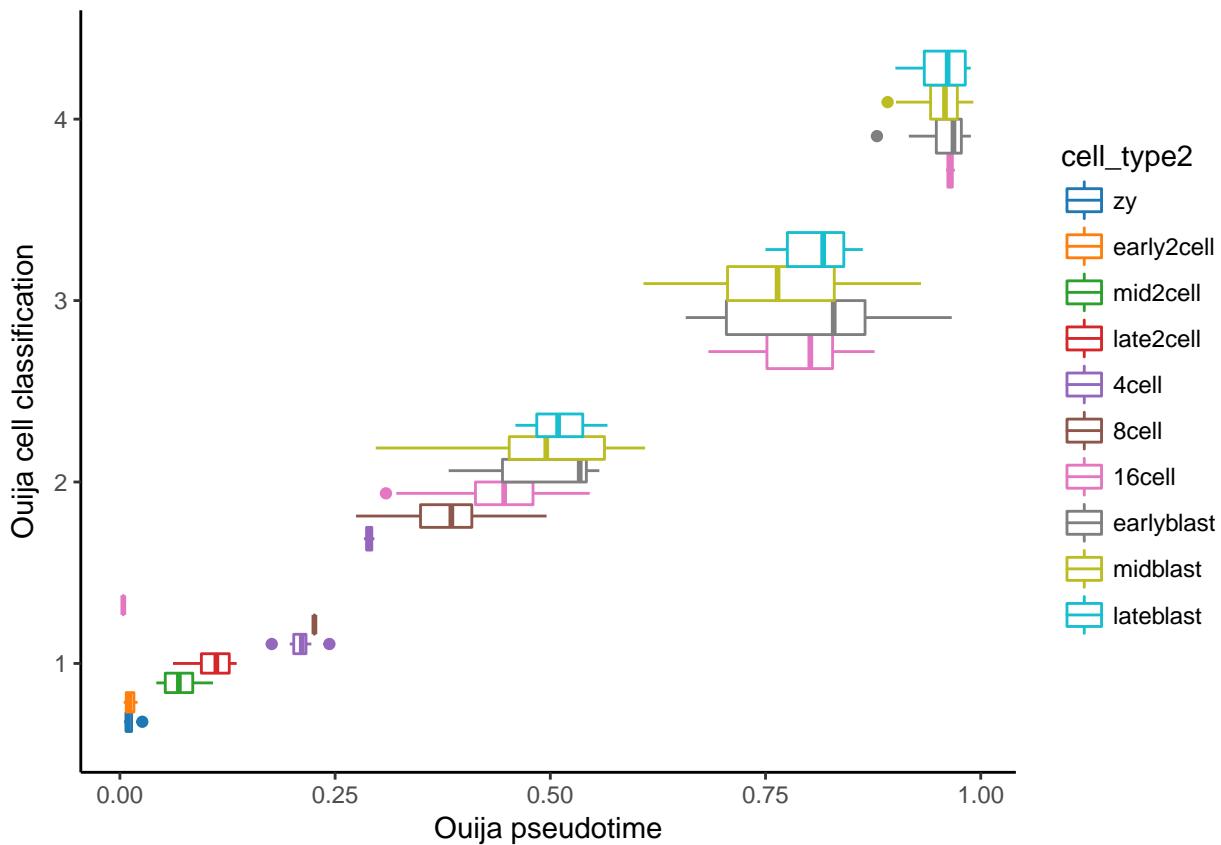
ggplot(as.data.frame(colData(deng_SCE)),
       aes(x = pseudotime_ouija,
            y = cell_type2, colour = cell_type2)) +
  geom_quasirandom(groupOnX = FALSE) +
  scale_color_tableau() + theme_classic() +
  xlab("Ouija pseudotime") +
  ylab("Timepoint") +
  theme_classic()
```



Ouja does quite well in the ordering of the cells here, although it can be sensitive to the choice of marker genes and prior information supplied. How do the results change if you select different marker genes or change the priors?

Ouja identifies four metastable states here, which we might annotate as “zygote/2cell”, “4/8/16 cell”, “blast1” and “blast2”.

```
ggplot(as.data.frame(colData(deng_SCE)),
       aes(x = as.factor(ouija_cell_class),
           y = pseudotime_ouija, colour = cell_type2)) +
  geom_boxplot() +
  coord_flip() +
  scale_color_tableau() + theme_classic() +
  xlab("Ouja cell classification") +
  ylab("Ouja pseudotime") +
  theme_classic()
```



A common analysis is to work out the regulation orderings of genes. For example, is gene A upregulated before gene B? Does gene C peak before the downregulation of gene D? Ouija answers these questions in terms of a Bayesian hypothesis test of whether the difference in regulation timing (either switch time or peak time) is significantly different to 0. This is collated using the gene\_regulation function.

```
gene_regs <- gene_regulation(oui_vb)
head(gene_regs)
```

```
## # A tibble: 6 x 7
## # Groups:   label, gene_A [6]
##   label      gene_A gene_B mean_difference lower_95 upper_95 significant
##   <chr>      <chr>  <chr>        <dbl>     <dbl>    <dbl>    <lgl>
## 1 Bmp15 - Cdx2 Bmp15  Cdx2       -0.0434  -0.0912   0.0110  F
## 2 Bmp15 - Cre~ Bmp15  Creb3      0.278    0.220    0.330   T
## 3 Bmp15 - Elf5 Bmp15  Elf5       -0.656   -0.688   -0.618   T
## 4 Bmp15 - Eom~ Bmp15  Eomes     0.0766   0.00433   0.153   T
## 5 Bmp15 - Fox~ Bmp15  Foxa1     -0.0266  -0.0611   0.00287 F
## 6 Bmp15 - Gdf3 Bmp15  Gdf3       0.0704   0.00963   0.134   T
```

What conclusions can you draw from the gene regulation output from Ouija?

If you have time, you might try the HMC inference method and see if that changes the Ouija results in any way.

#### 8.4.7 Comparison of the methods

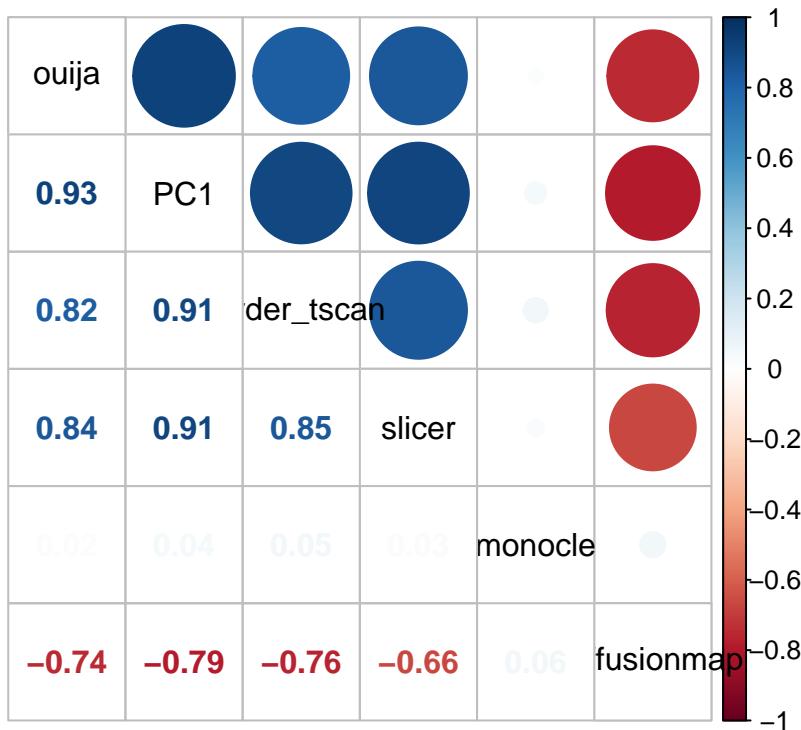
How do the trajectories inferred by TSCAN, Monocle, Diffusion Map, SLICER and Ouija compare?

TSCAN and Diffusion Map methods get the trajectory the “wrong way round”, so we’ll adjust that for these comparisons.

```
df_pseudotime <- as.data.frame(
  colData(deng_SCE) [, grep("pseudotime", colnames(colData(deng_SCE)))]
)
colnames(df_pseudotime) <- gsub("pseudotime_", "", 
                                colnames(df_pseudotime))
df_pseudotime$PC1 <- deng_SCE$PC1
df_pseudotime$order_tscan <- -df_pseudotime$order_tscan
df_pseudotime$diffusionmap <- -df_pseudotime$diffusionmap

corrplot.mixed(cor(df_pseudotime, use = "na.or.complete"),
               order = "hclust", tl.col = "black",
               main = "Correlation matrix for pseudotime results",
               mar = c(0, 0, 3.1, 0))
```

### Correlation matrix for pseudotime results



We see here that Ouija, TSCAN and SLICER all give trajectories that are similar and strongly correlated with PC1. Diffusion Map is less strongly correlated with these methods, and Monocle gives very different results.

**Exercise 6:** Compare destiny and SLICER to TSCAN, Monocle and Ouija in more depth. Where and how do they differ?

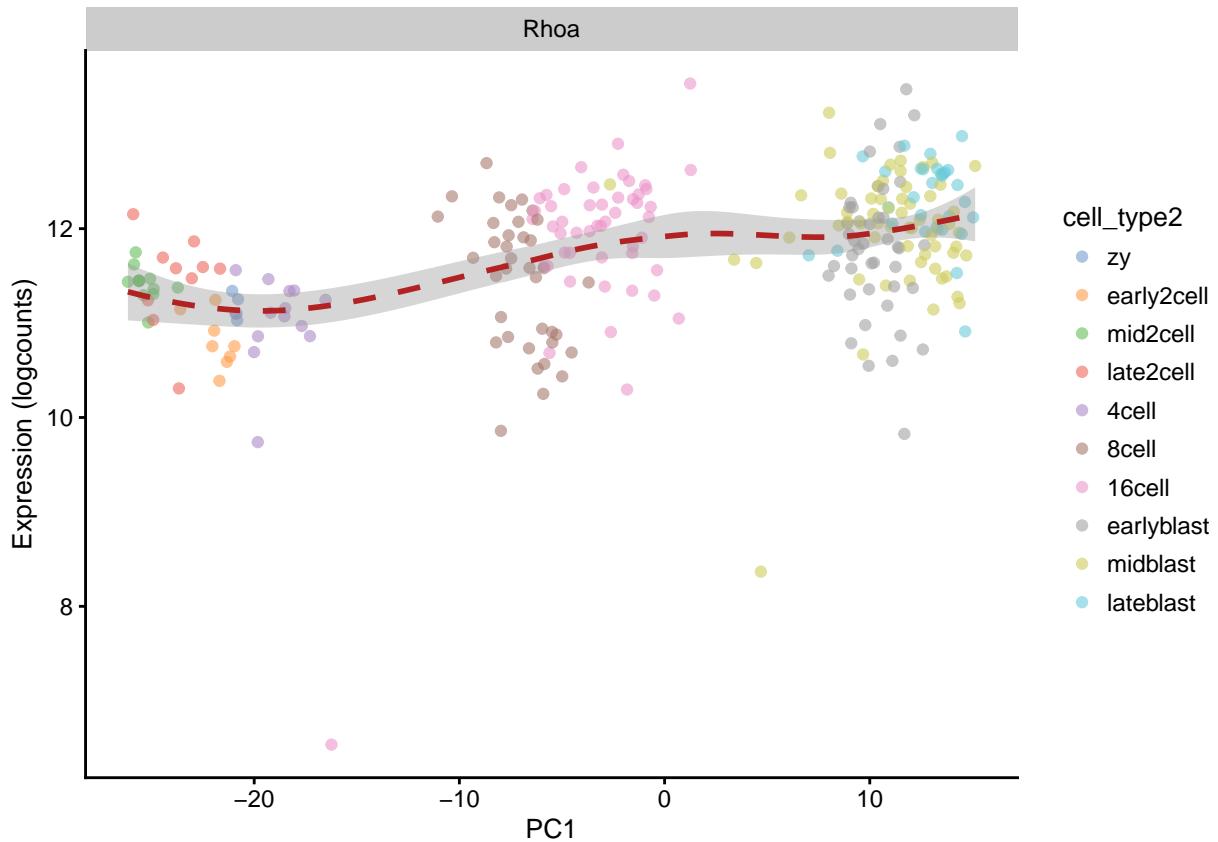
#### 8.4.8 Expression of genes through time

Each package also enables the visualization of expression through pseudotime. Following individual genes is very helpful for identifying genes that play an important role in the differentiation process. We illustrate the procedure using the Rhoa gene.

We have added the pseudotime values computed with all methods here to the `colData` slot of an `SCE` object. Having done that, the full plotting capabilities of the `scater` package can be used to investigate relationships between gene expression, cell populations and pseudotime. This is particularly useful for the packages such as SLICER that do not provide plotting functions.

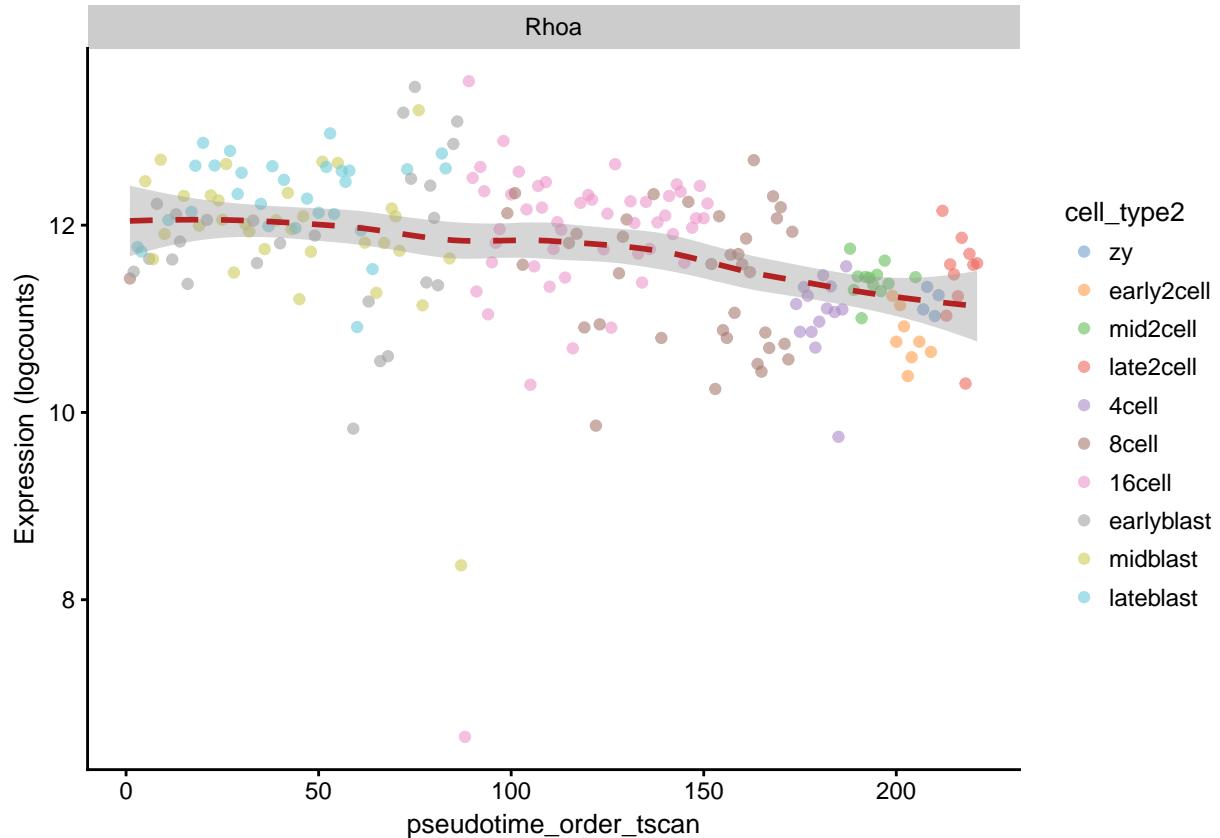
### Principal components

```
plotExpression(deng_SCE, "Rhoa", x = "PC1",
               colour_by = "cell_type2", show_violin = FALSE,
               show_smooth = TRUE)
```

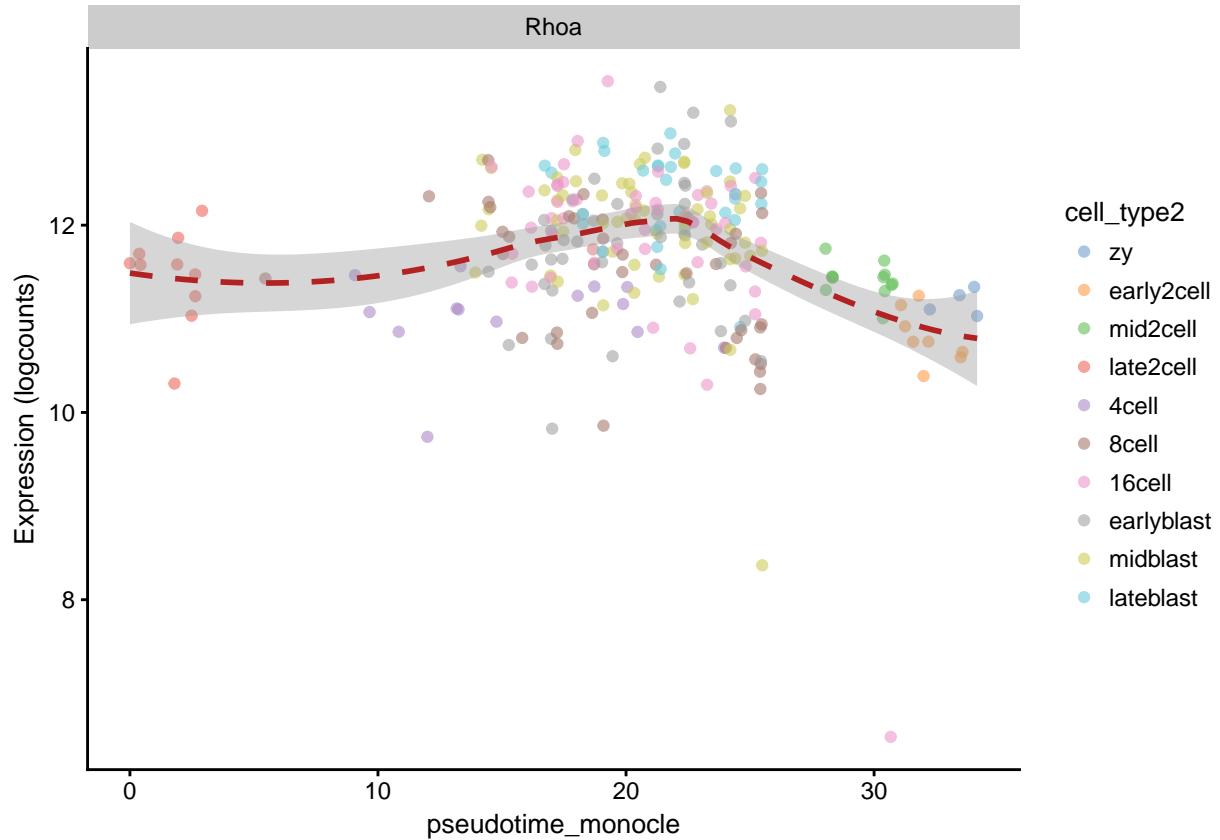


### TSCAN

```
plotExpression(deng_SCE, "Rhoa", x = "pseudotime_order_tscan",
               colour_by = "cell_type2", show_violin = FALSE,
               show_smooth = TRUE)
```

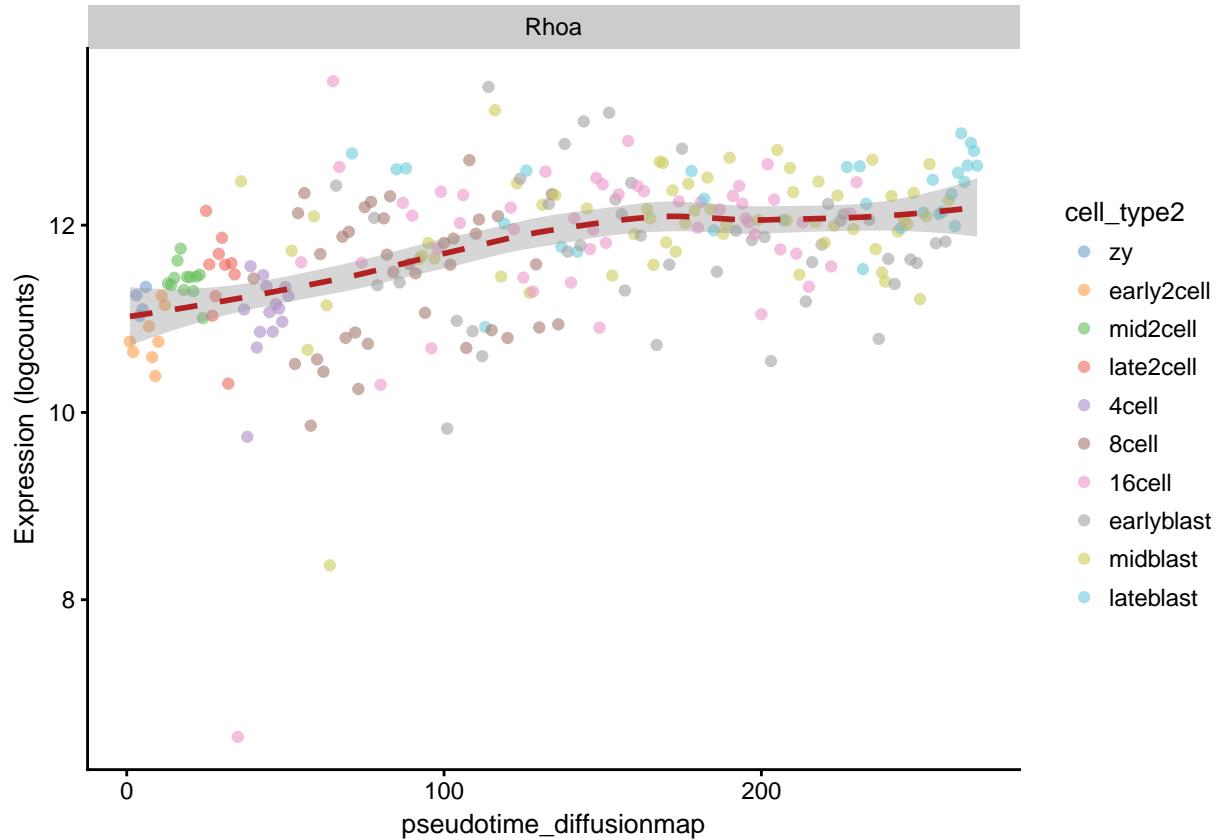
**Monocle**

```
plotExpression(deng_SCE, "Rhoa", x = "pseudotime_monocle",
               colour_by = "cell_type2", show_violin = FALSE,
               show_smooth = TRUE)
```



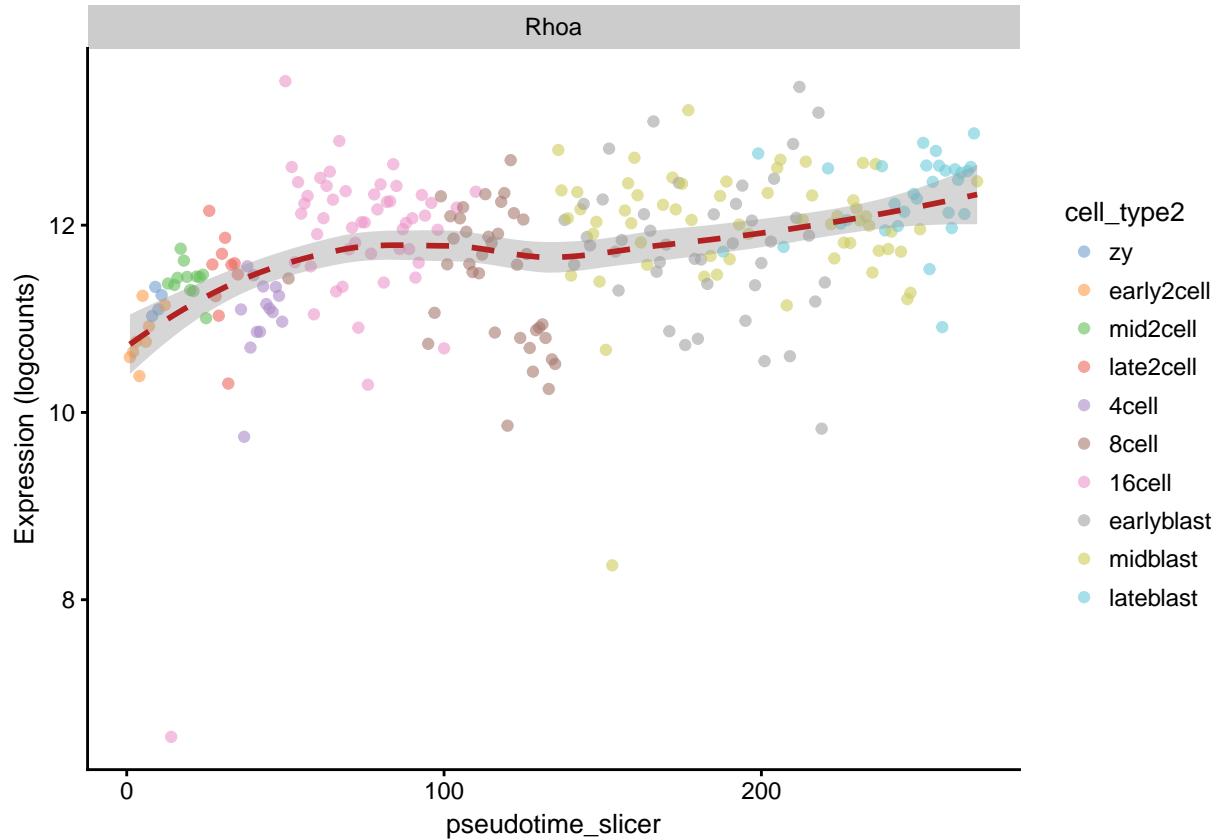
### Diffusion Map

```
plotExpression(deng_SCE, "Rho", x = "pseudotime_diffusionmap",
               colour_by = "cell_type2", show_violin = FALSE,
               show_smooth = TRUE)
```



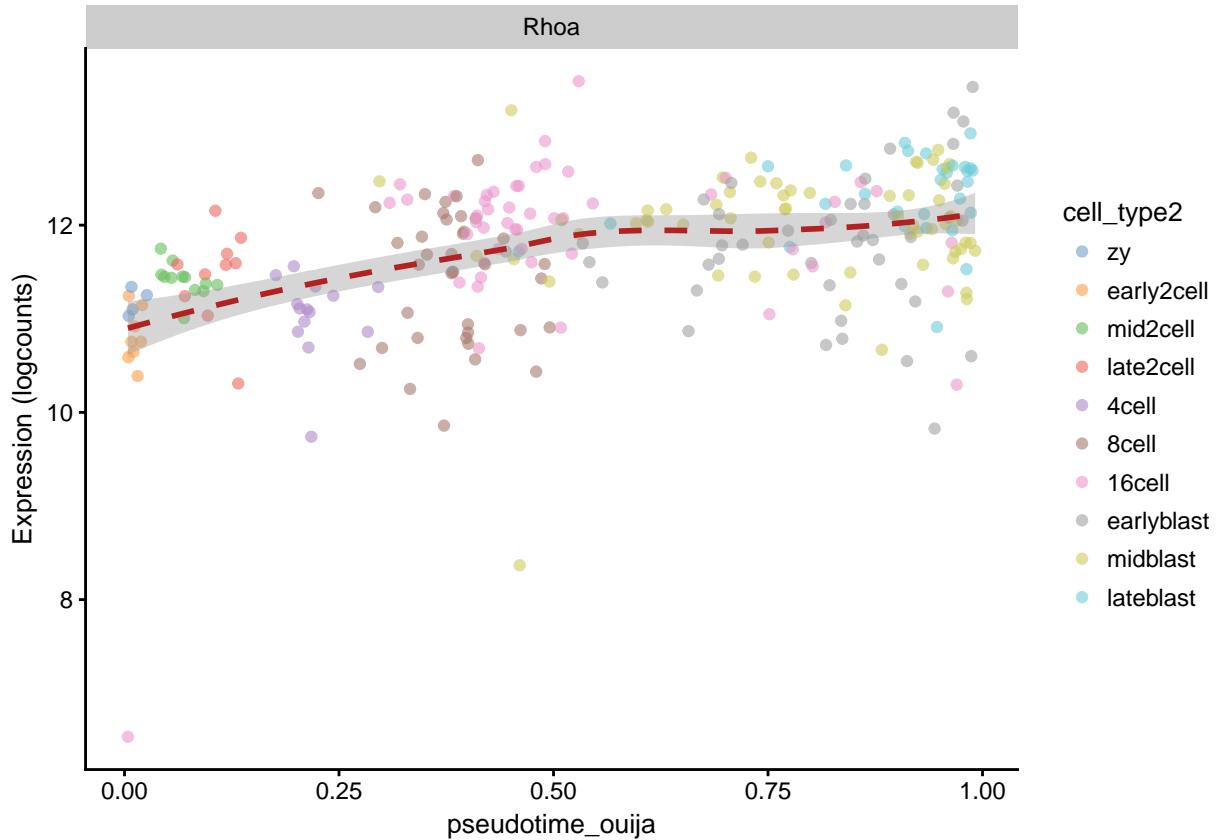
### SLICER

```
plotExpression(deng_SCE, "Rhoa", x = "pseudotime_slicer",
               colour_by = "cell_type2", show_violin = FALSE,
               show_smooth = TRUE)
```



### Ouija

```
plotExpression(deng_SCE, "Rhoa", x = "pseudotime_ouija",
               colour_by = "cell_type2", show_violin = FALSE,
               show_smooth = TRUE)
```



How many of these methods outperform the naive approach of using the first principal component to represent pseudotime for these data?

**Exercise 7:** Repeat the exercise using a subset of the genes, e.g. the set of highly variable genes that can be obtained using `Brennecke_getVariableGenes()`

#### 8.4.9 sessionInfo()

```
## R version 3.4.3 (2017-11-30)
## Platform: x86_64-pc-linux-gnu (64-bit)
## Running under: Debian GNU/Linux 9 (stretch)
##
## Matrix products: default
## BLAS: /usr/lib/openblas-base/libblas.so.3
## LAPACK: /usr/lib/libopenblas-r0.2.19.so
##
## locale:
## [1] LC_CTYPE=en_US.UTF-8          LC_NUMERIC=C
## [3] LC_TIME=en_US.UTF-8          LC_COLLATE=en_US.UTF-8
## [5] LC_MONETARY=en_US.UTF-8       LC_MESSAGES=C
## [7] LC_PAPER=en_US.UTF-8          LC_NAME=C
## [9] LC_ADDRESS=C                  LC_TELEPHONE=C
## [11] LC_MEASUREMENT=en_US.UTF-8    LC_IDENTIFICATION=C
##
## attached base packages:
## [1] splines     parallel   stats4     methods    stats      graphics   grDevices
## [8] utils       datasets   base
```

```

## 
## other attached packages:
## [1] bindrcpp_0.2                  rstan_2.17.3
## [3] StanHeaders_2.17.2            lle_1.1
## [5] snowfall_1.84-6.1             snow_0.4-2
## [7] MASS_7.3-45                  scatterplot3d_0.3-40
## [9] corrplot_0.84                 ggbeeswarm_0.6.0
## [11] gghthemes_3.4.0               scater_1.6.2
## [13] ouija_0.99.0                Rcpp_0.12.15
## [15] SLICER_0.2.0                 destiny_2.6.1
## [17] monocle_2.6.1               DDRTree_0.1.5
## [19] irlba_2.3.2                 VGAM_1.0-4
## [21] ggplot2_2.2.1                Matrix_1.2-7.1
## [23] M3Drop_3.05.00              numDeriv_2016.8-1
## [25] TSCAN_1.16.0                SingleCellExperiment_1.0.0
## [27] SummarizedExperiment_1.8.1   DelayedArray_0.4.1
## [29] matrixStats_0.53.0           Biobase_2.38.0
## [31] GenomicRanges_1.30.1         GenomeInfoDb_1.14.0
## [33] IRanges_2.12.0               S4Vectors_0.16.0
## [35] BiocGenerics_0.24.0          knitr_1.19
## 

## loaded via a namespace (and not attached):
## [1] utf8_1.1.3                   shinydashboard_0.6.1    R.utils_2.6.0
## [4] tidyselect_0.2.3              lme4_1.1-15              RSQLite_2.0
## [7] AnnotationDbi_1.40.0         htmlwidgets_1.0          grid_3.4.3
## [10] combinat_0.0-8               Rtsne_0.13               munsell_0.4.3
## [13] codetools_0.2-15            statmod_1.4.30          colorspace_1.3-2
## [16] fastICA_1.2-1               rstudioapi_0.7          robustbase_0.92-8
## [19] vcd_1.4-4                   tensor_1.5               VIM_4.7.0
## [22] TTR_0.23-3                 labeling_0.3              slam_0.1-42
## [25] splancs_2.01-40             tximport_1.6.0           bbmle_1.0.20
## [28] GenomeInfoDbData_1.0.0     polyclip_1.6-1           bit64_0.9-7
## [31] pheatmap_1.0.8              rhdf5_2.22.0             rprojroot_1.3-2
## [34] coda_0.19-1                 xfun_0.1                 R6_2.2.2
## [37] RcppEigen_0.3.3.3.1        locfit_1.5-9.1           bitops_1.0-6
## [40] spatstat.utils_1.8-0       assertthat_0.2.0         scales_0.5.0
## [43] nnet_7.3-12                 beeswarm_0.2.3          gtable_0.2.0
## [46] goftest_1.1-1               rlang_0.1.6              MatrixModels_0.4-1
## [49] lazyeval_0.2.1              acepack_1.4.1            checkmate_1.8.5
## [52] inline_0.3.14              yaml_2.1.16              reshape2_1.4.3
## [55] abind_1.4-5                backports_1.1.2          httpuv_1.3.5
## [58] Hmisc_4.1-1                 tensorA_0.36             tools_3.4.3
## [61] bookdown_0.6                 cubature_1.3-11          gplots_3.0.1
## [64] RColorBrewer_1.1-2          proxy_0.4-21             MCMCglmm_2.25
## [67] plyr_1.8.4                  progress_1.1.2            base64enc_0.1-3
## [70] zlibbioc_1.24.0             purrr_0.2.4              RCurl_1.95-4.10
## [73] densityClust_0.3            prettyunits_1.0.2         rpart_4.1-10
## [76] alphahull_2.1                deldir_0.1-14            reldist_1.6-6
## [79] viridis_0.5.0               cowplot_0.9.2            zoo_1.8-1
## [82] ggrepel_0.7.0                cluster_2.0.6            magrittr_1.5
## [85] data.table_1.10.4-3          SparseM_1.77             lmtest_0.9-35
## [88] RANN_2.5.1                  mime_0.5                 evaluate_0.10.1
## [91] xtable_1.8-2                 XML_3.98-1.9             smoother_1.1
## [94] pbkrtest_0.4-7              mclust_5.4               gridExtra_2.3

```

```

## [97] biomaRt_2.34.2          HSMMSSingleCell_0.112.0 compiler_3.4.3
## [100] tibble_1.4.2             crayon_1.3.4           KernSmooth_2.23-15
## [103] minqa_1.2.4              R.oo_1.21.0            htmltools_0.3.6
## [106] mgcv_1.8-23               corpcor_1.6.9          Formula_1.2-2
## [109] tidyverse_0.8.0            DBI_0.7                boot_1.3-18
## [112] car_2.1-6                 cli_1.0.0              sgeostat_1.0-27
## [115] R.methodsS3_1.7.1         gdata_2.18.0           bindr_0.1
## [118] igraph_1.1.2              pkgconfig_2.0.1        foreign_0.8-67
## [121] laeken_0.4.6              sp_1.2-7               viper_0.4.5
## [124] XVector_0.18.0            stringr_1.2.0          digest_0.6.15
## [127] spatstat.data_1.2-0       rmarkdown_1.8           htmlTable_1.11.2
## [130] edgeR_3.20.8              curl_3.1               shiny_1.0.5
## [133] gtools_3.5.0              quantreg_5.35          rjson_0.2.15
## [136] nlptr_1.0.4               nlme_3.1-129           viridisLite_0.3.0
## [139] limma_3.34.8              pillar_1.1.0           lattice_0.20-34
## [142] httr_1.3.1                DEoptimR_1.0-8          survival_2.40-1
## [145] glue_1.2.0                xts_0.10-1              qlcMatrix_0.9.5
## [148] FNN_1.1                  spatstat_1.55-0         bit_1.1-12
## [151] class_7.3-14              stringi_1.1.6          blob_1.1.0
## [154] memoise_1.1.0             latticeExtra_0.6-28    caTools_1.17.1
## [157] dplyr_0.7.4               e1071_1.6-8            ape_5.0
## [160] tripack_1.3-8

```

## 8.5 Imputation

```

library(scImpute)
library(SC3)
library(scater)
library(SingleCellExperiment)
library(mclust)
set.seed(1234567)

```

As discussed previously, one of the main challenges when analyzing scRNA-seq data is the presence of zeros, or dropouts. The dropouts are assumed to have arisen for three possible reasons:

- The gene was not expressed in the cell and hence there are no transcripts to sequence
- The gene was expressed, but for some reason the transcripts were lost somewhere prior to sequencing
- The gene was expressed and transcripts were captured and turned into cDNA, but the sequencing depth was not sufficient to produce any reads.

Thus, dropouts could be result of experimental shortcomings, and if this is the case then we would like to provide computational corrections. One possible solution is to impute the dropouts in the expression matrix. To be able to impute gene expression values, one must have an underlying model. However, since we do not know which dropout events are technical artefacts and which correspond to the transcript being truly absent, imputation is a difficult challenge.

To the best of our knowledge, there are currently two different imputation methods available: MAGIC (van Dijk et al., 2017) and scImpute (Li and Li, 2017). MAGIC is only available for Python or Matlab, but we will run it from within R.

### 8.5.1 scImpute

To test `scImpute`, we use the default parameters and we apply it to the Deng dataset that we have worked with before. `scImpute` takes a .csv or .txt file as an input:

```
deng <- readRDS("deng/deng-reads.rds")
write.csv(counts(deng), "deng.csv")
scimpute(
  count_path = "deng.csv",
  infile = "csv",
  outfile = "txt",
  out_dir = "./",
  Kcluster = 10,
  ncores = 2
)

## [1] "reading in raw count matrix ..."
## [1] "number of genes in raw count matrix 22431"
## [1] "number of cells in raw count matrix 268"
## [1] "inferring cell similarities ..."
## [1] "cluster sizes:"
## [1] 12 9 26 5 9 57 58 43 17 22
## [1] "estimating dropout probability for type 1 ..."
## [1] "imputing dropout values for type 1 ..."
## [1] "estimating dropout probability for type 2 ..."
## [1] "imputing dropout values for type 2 ..."
## [1] "estimating dropout probability for type 3 ..."
## [1] "imputing dropout values for type 3 ..."
## [1] "estimating dropout probability for type 4 ..."
## [1] "imputing dropout values for type 4 ..."
## [1] "estimating dropout probability for type 5 ..."
## [1] "imputing dropout values for type 5 ..."
## [1] "estimating dropout probability for type 6 ..."
## [1] "imputing dropout values for type 6 ..."
## [1] "estimating dropout probability for type 7 ..."
## [1] "imputing dropout values for type 7 ..."
## [1] "estimating dropout probability for type 8 ..."
## [1] "imputing dropout values for type 8 ..."
## [1] "estimating dropout probability for type 9 ..."
## [1] "imputing dropout values for type 9 ..."
## [1] "estimating dropout probability for type 10 ..."
## [1] "imputing dropout values for type 10 ..."
## [1] "writing imputed count matrix ..."

## [1] 17 18 88 111 126 177 186 229 244 247
```

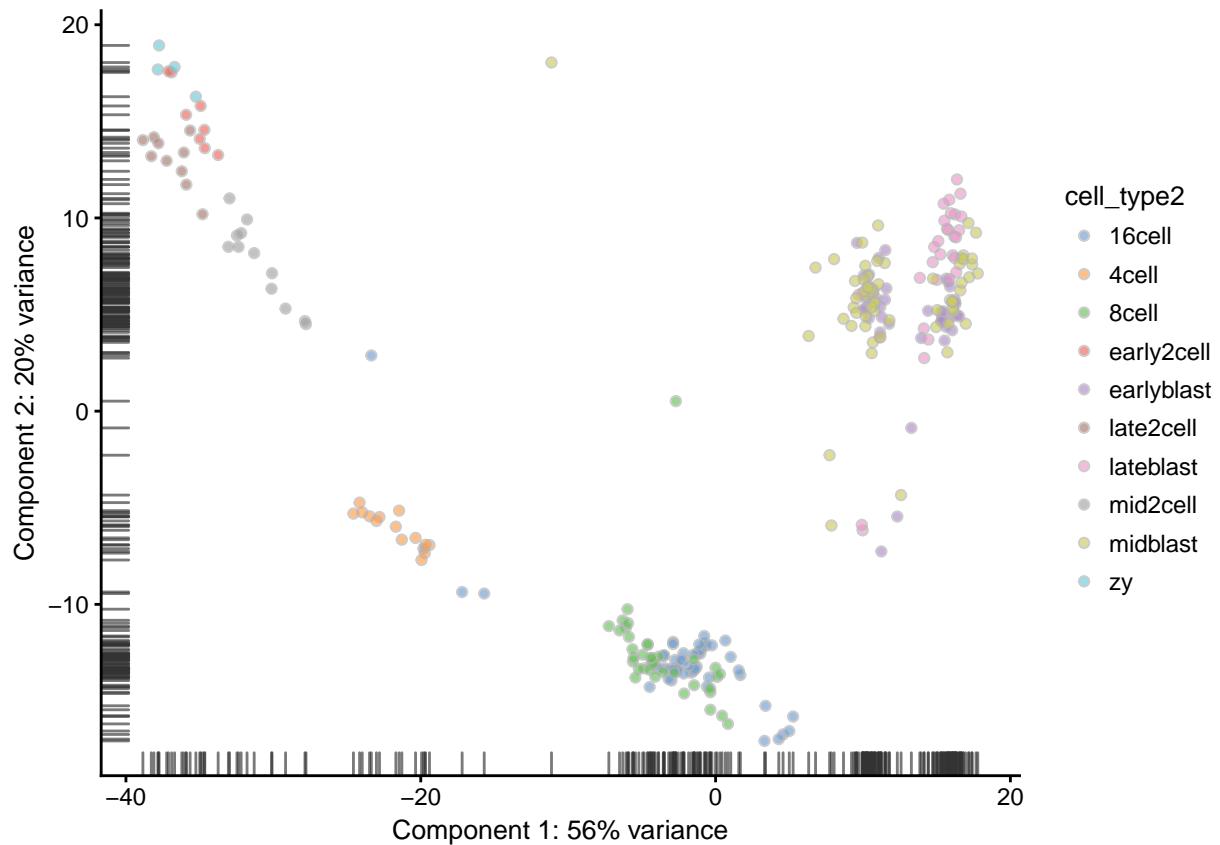
Now we can compare the results with original data by considering a PCA plot

```
res <- read.table("scimpute_count.txt")
colnames(res) <- NULL
res <- SingleCellExperiment(
  assays = list(logcounts = log2(as.matrix(res) + 1)),
  colData = colData(deng)
)
rowData(res)$feature_symbol <- rowData(deng)$feature_symbol
plotPCA(
```

```

    res,
    colour_by = "cell_type2"
)

```



Compare this result to the original data in Chapter 8.2. What are the most significant differences?

To evaluate the impact of the imputation, we use SC3 to cluster the imputed matrix

```

res <- sc3_estimate_k(res)
metadata(res)$sc3$k_estimation

```

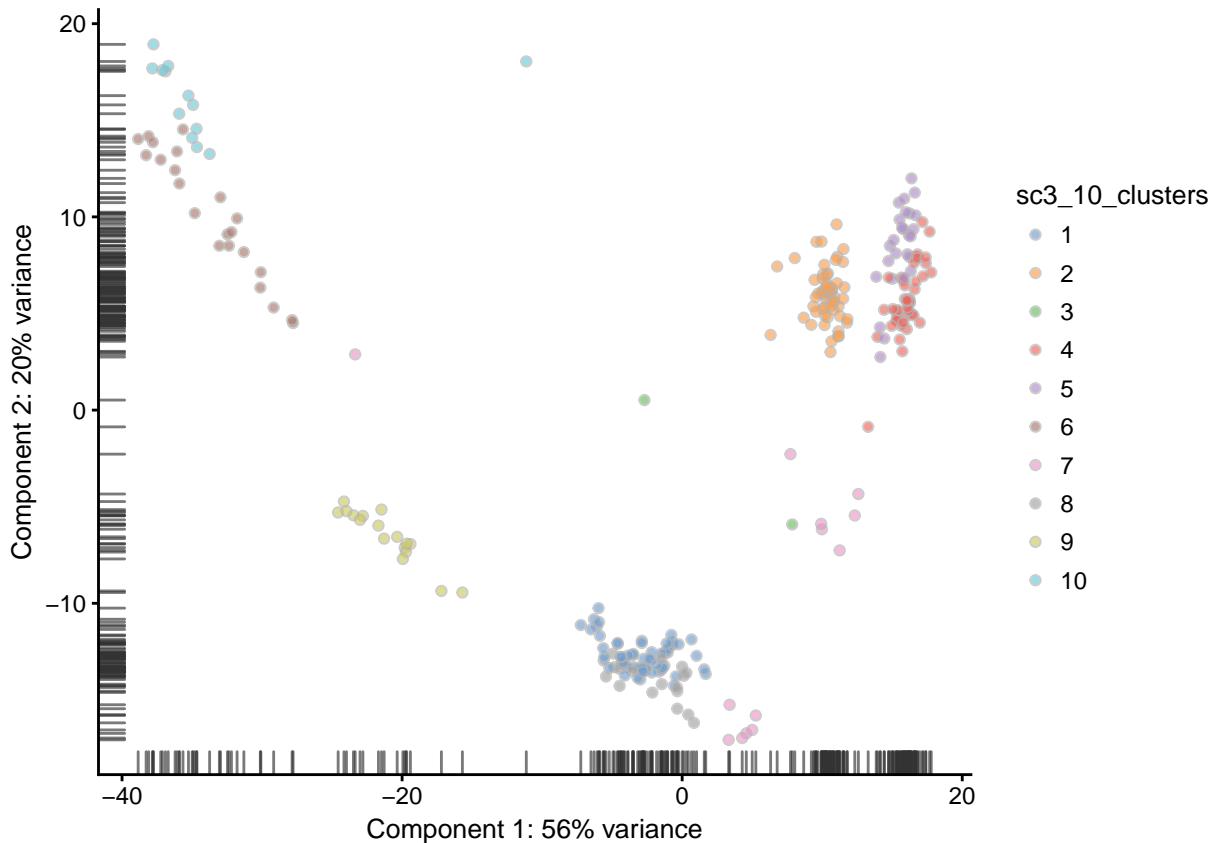
```

## [1] 6
res <- sc3(res, ks = 10, gene_filter = FALSE)

adjustedRandIndex(colData(deng)$cell_type2, colData(res)$sc3_10_clusters)

## [1] 0.4687332
plotPCA(
  res,
  colour_by = "sc3_10_clusters"
)

```

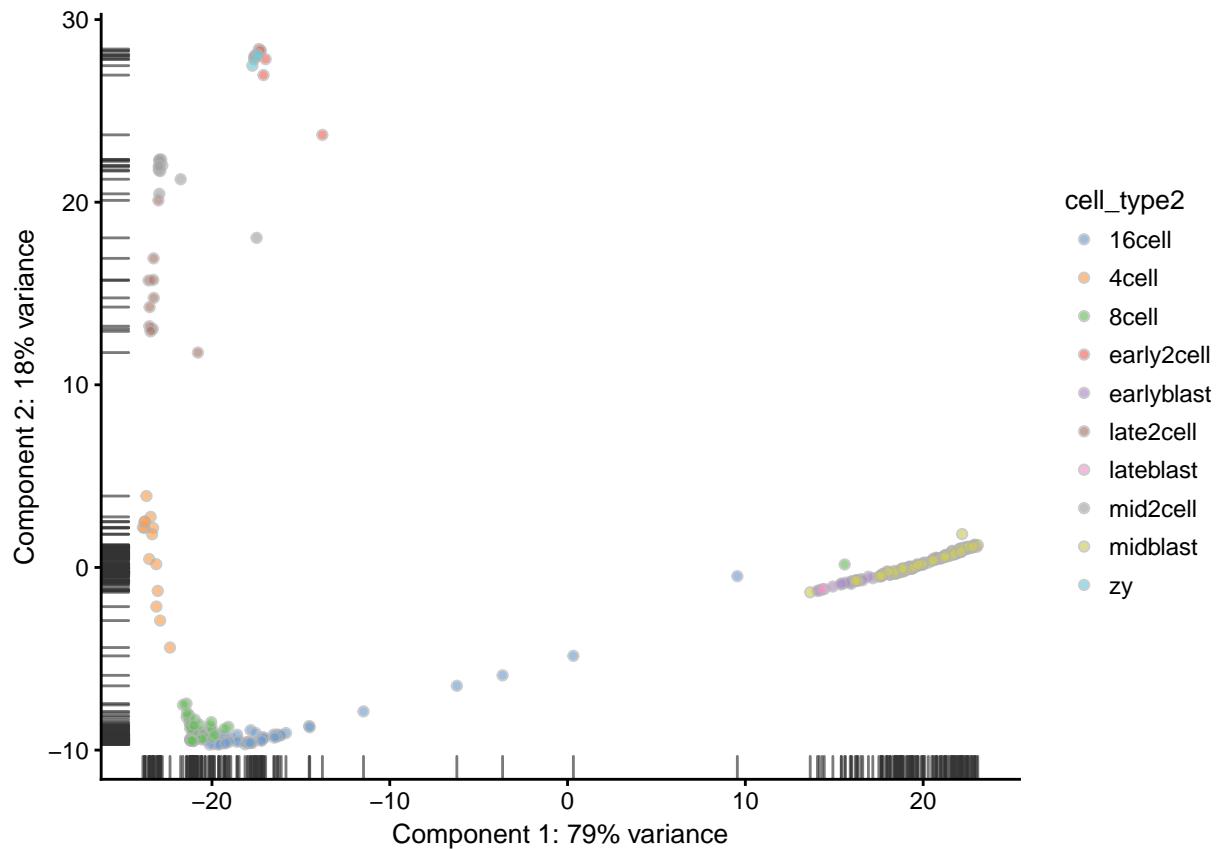


**Exercise:** Based on the PCA and the clustering results, do you think that imputation using `scImpute` is a good idea for the Deng dataset?

### 8.5.2 MAGIC

```
system("python3 utils/MAGIC.py -d deng.csv -o MAGIC_count.csv --cell-axis columns -l 1 --pca-non-random")

res <- read.csv("MAGIC_count.csv", header = TRUE)
rownames(res) <- res[,1]
res <- res[,-1]
res <- t(res)
res <- SingleCellExperiment(
  assays = list(logcounts = res),
  colData = colData(deng)
)
rowData(res)$feature_symbol <- rownames(res)
plotPCA(
  res,
  colour_by = "cell_type2"
)
```



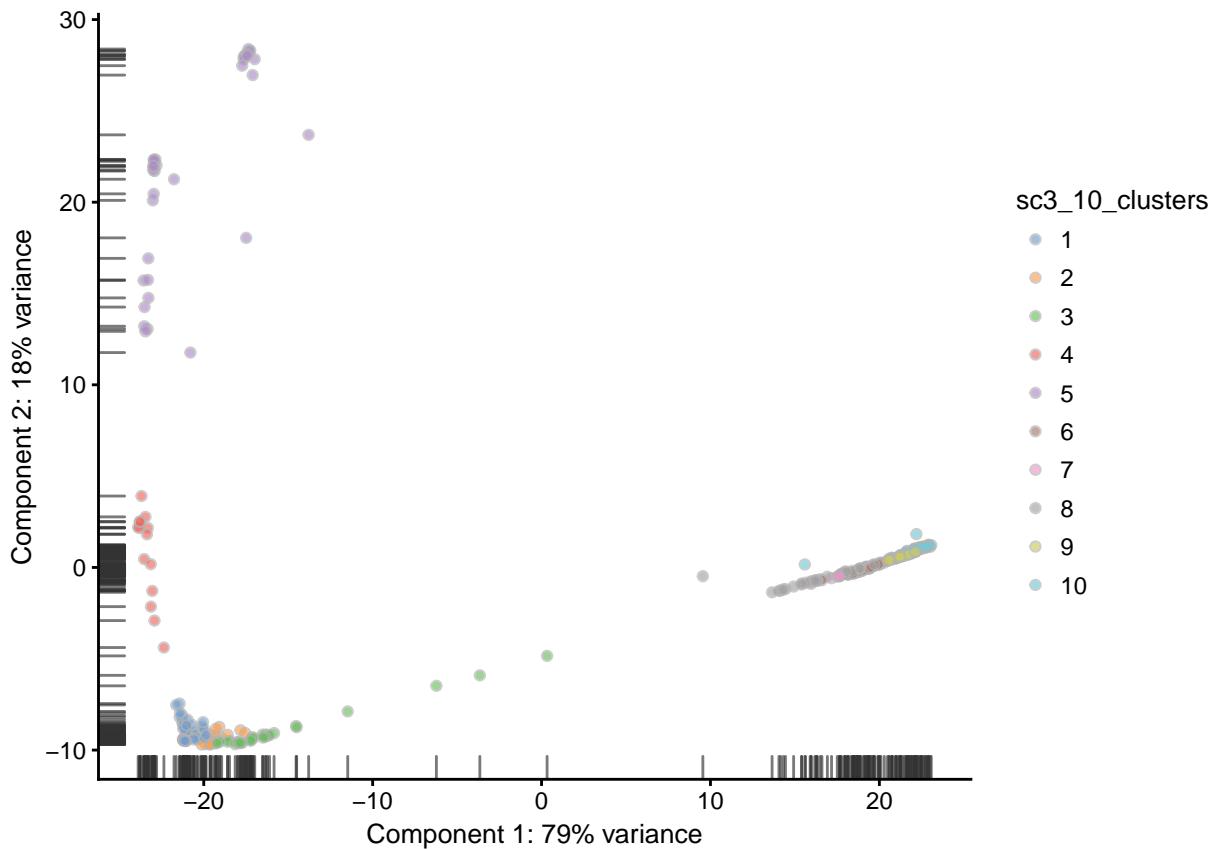
Compare this result to the original data in Chapter 8.2. What are the most significant differences?

To evaluate the impact of the imputation, we use SC3 to cluster the imputed matrix

```
res <- sc3_estimate_k(res)
metadata(res)$sc3$k_estimation
```

```
## [1] 4
res <- sc3(res, ks = 10, gene_filter = FALSE)
adjustedRandIndex(colData(deng)$cell_type2, colData(res)$sc3_10_clusters)
```

```
## [1] 0.3752866
plotPCA(
  res,
  colour_by = "sc3_10_clusters"
)
```



**Exercise:** What is the difference between scImpute and MAGIC based on the PCA and clustering analysis? Which one do you think is best to use?

### 8.5.3 sessionInfo()

```
## R version 3.4.3 (2017-11-30)
## Platform: x86_64-pc-linux-gnu (64-bit)
## Running under: Debian GNU/Linux 9 (stretch)
##
## Matrix products: default
## BLAS: /usr/lib/openblas-base/libblas.so.3
## LAPACK: /usr/lib/libopenblas-r0.2.19.so
##
## locale:
## [1] LC_CTYPE=en_US.UTF-8          LC_NUMERIC=C
## [3] LC_TIME=en_US.UTF-8          LC_COLLATE=en_US.UTF-8
## [5] LC_MONETARY=en_US.UTF-8       LC_MESSAGES=C
## [7] LC_PAPER=en_US.UTF-8          LC_NAME=C
## [9] LC_ADDRESS=C                  LC_TELEPHONE=C
## [11] LC_MEASUREMENT=en_US.UTF-8   LC_IDENTIFICATION=C
##
## attached base packages:
## [1] stats4     methods    parallel   stats      graphics   grDevices utils
## [8] datasets   base
##
## other attached packages:
```

```

## [1] mclust_5.4                  scater_1.6.2
## [3] SingleCellExperiment_1.0.0   SummarizedExperiment_1.8.1
## [5] DelayedArray_0.4.1          matrixStats_0.53.0
## [7] GenomicRanges_1.30.1        GenomeInfoDb_1.14.0
## [9] IRanges_2.12.0              S4Vectors_0.16.0
## [11] ggplot2_2.2.1              Biobase_2.38.0
## [13] BiocGenerics_0.24.0       SC3_1.7.7
## [15] scImpute_0.0.5             doParallel_1.0.11
## [17] iterators_1.0.9            foreach_1.4.4
## [19] penalized_0.9-50          survival_2.40-1
## [21] kernlab_0.9-25           knitr_1.19

##
## loaded via a namespace (and not attached):
## [1] ggbbeeswarm_0.6.0          colorspace_1.3-2      rjson_0.2.15
## [4] class_7.3-14                rprojroot_1.3-2      XVector_0.18.0
## [7] bit64_0.9-7                AnnotationDbi_1.40.0 mvtnorm_1.0-7
## [10] codetools_0.2-15           splines_3.4.3         tximport_1.6.0
## [13] robustbase_0.92-8          cluster_2.0.6         pheatmap_1.0.8
## [16] shinydashboard_0.6.1        shiny_1.0.5           rrcov_1.4-3
## [19] compiler_3.4.3             httr_1.3.1            backports_1.1.2
## [22] assertthat_0.2.0           Matrix_1.2-7.1       lazyeval_0.2.1
## [25] limma_3.34.8              htmltools_0.3.6       prettyunits_1.0.2
## [28] tools_3.4.3                bindr_0.2              gtable_0.2.0
## [31] glue_1.2.0                 GenomeInfoDbData_1.0.0 reshape2_1.4.3
## [34] dplyr_0.7.4                doRNG_1.6.6            Rcpp_0.12.15
## [37] gdata_2.18.0               xfun_0.1              stringr_1.2.0
## [40] mime_0.5                   rngtools_1.2.4         gtools_3.5.0
## [43] WriteXLS_4.0.0             XML_3.98-1.9          edgeR_3.20.8
## [46] DEoptimR_1.0-8              zlibbioc_1.24.0       scales_0.5.0
## [49] rhdf5_2.22.0               RColorBrewer_1.1-2    yaml_2.1.16
## [52] memoise_1.1.0              gridExtra_2.3          pkgmaker_0.22
## [55] biomaRt_2.34.2             stringi_1.1.6         RSQLite_2.0
## [58] pcaPP_1.9-73                e1071_1.6-8           caTools_1.17.1
## [61] rlang_0.1.6                 pkgconfig_2.0.1        bitops_1.0-6
## [64] evaluate_0.10.1             lattice_0.20-34       ROCR_1.0-7
## [67] bindr_0.1                   labeling_0.3           cowplot_0.9.2
## [70] bit_1.1-12                 plyr_1.8.4             magrittr_1.5
## [73] bookdown_0.6                R6_2.2.2              gplots_3.0.1
## [76] DBI_0.7                     pillar_1.1.0           RCurl_1.95-4.10
## [79] tibble_1.4.2                KernSmooth_2.23-15    rmarkdown_1.8
## [82] viridis_0.5.0                progress_1.1.2         locfit_1.5-9.1
## [85] grid_3.4.3                  data.table_1.10.4-3    blob_1.1.0
## [88] digest_0.6.15               xtable_1.8-2           httpuv_1.3.5
## [91] munsell_0.4.3               registry_0.5          beeswarm_0.2.3
## [94] viridisLite_0.3.0            vipor_0.4.5

```

## 8.6 Differential Expression (DE) analysis

### 8.6.1 Bulk RNA-seq

One of the most common types of analyses when working with bulk RNA-seq data is to identify differentially expressed genes. By comparing the genes that change between two conditions, e.g. mutant and wild-type or

stimulated and unstimulated, it is possible to characterize the molecular mechanisms underlying the change. Several different methods, e.g. DESeq2 and edgeR, have been developed for bulk RNA-seq. Moreover, there are also extensive datasets available where the RNA-seq data has been validated using RT-qPCR. These data can be used to benchmark DE finding algorithms and the available evidence suggests that the algorithms are performing quite well.

### 8.6.2 Single cell RNA-seq

In contrast to bulk RNA-seq, in scRNA-seq we usually do not have a defined set of experimental conditions. Instead, as was shown in a previous chapter (8.2) we can identify the cell groups by using an unsupervised clustering approach. Once the groups have been identified one can find differentially expressed genes either by comparing the differences in variance between the groups (like the Kruskal-Wallis test implemented in SC3), or by comparing gene expression between clusters in a pairwise manner. In the following chapter we will mainly consider tools developed for pairwise comparisons.

### 8.6.3 Differences in Distribution

Unlike bulk RNA-seq, we generally have a large number of samples (i.e. cells) for each group we are comparing in single-cell experiments. Thus we can take advantage of the whole distribution of expression values in each group to identify differences between groups rather than only comparing estimates of mean-expression as is standard for bulk RNASeq.

There are two main approaches to comparing distributions. Firstly, we can use existing statistical models/distributions and fit the same type of model to the expression in each group then test for differences in the parameters for each model, or test whether the model fits better if a particular parameter is allowed to be different according to group. For instance in Chapter 7.10 we used edgeR to test whether allowing mean expression to be different in different batches significantly improved the fit of a negative binomial model of the data.

Alternatively, we can use a non-parametric test which does not assume that expression values follow any particular distribution, e.g. the Kolmogorov-Smirnov test (KS-test). Non-parametric tests generally convert observed expression values to ranks and test whether the distribution of ranks for one group are significantly different from the distribution of ranks for the other group. However, some non-parametric methods fail in the presence of a large number of tied values, such as the case for dropouts (zeros) in single-cell RNA-seq expression data. Moreover, if the conditions for a parametric test hold, then it will typically be more powerful than a non-parametric test.

### 8.6.4 Models of single-cell RNASeq data

The most common model of RNASeq data is the negative binomial model:

```
set.seed(1)
hist(
  rnbinom(
    1000,
    mu = 10,
    size = 100),
  col = "grey50",
  xlab = "Read Counts",
  main = "Negative Binomial"
)
```

## Negative Binomial

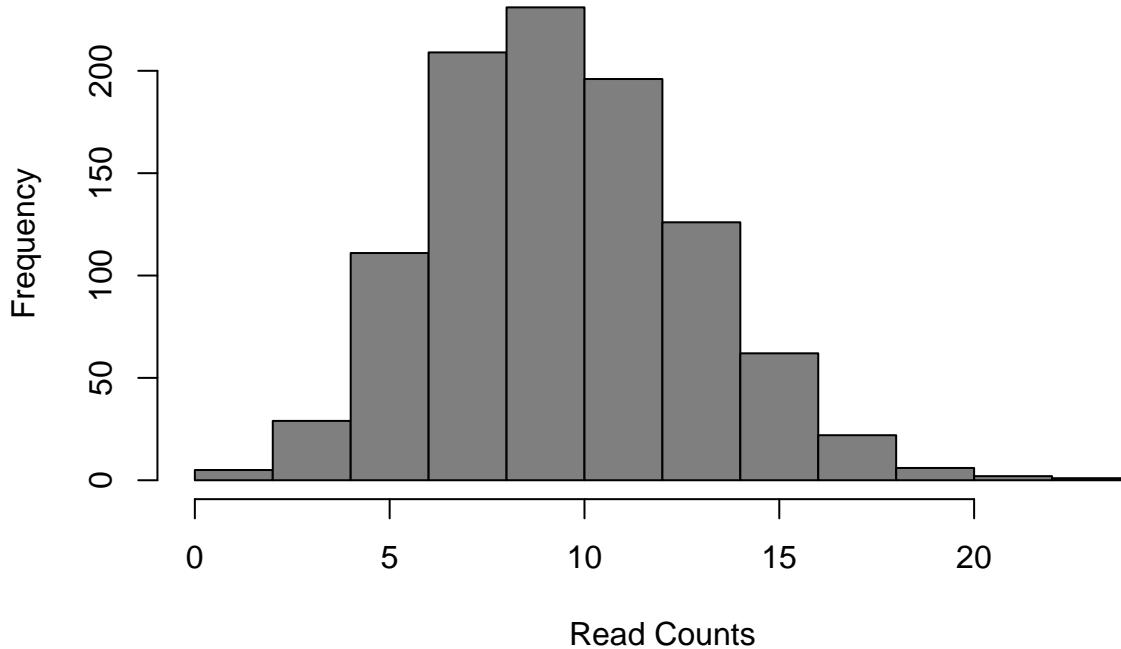


Figure 8.12: Negative Binomial distribution of read counts for a single gene across 1000 cells

Mean:  $\mu = \mu u$

Variance:  $\sigma^2 = \mu u + \mu u^2 / size$

It is parameterized by the mean expression ( $\mu u$ ) and the dispersion ( $size$ ), which is inversely related to the variance. The negative binomial model fits bulk RNA-seq data very well and it is used for most statistical methods designed for such data. In addition, it has been shown to fit the distribution of molecule counts obtained from data tagged by unique molecular identifiers (UMIs) quite well (Grun et al. 2014, Islam et al. 2011).

However, a raw negative binomial model does not fit full-length transcript data as well due to the high dropout rates relative to the non-zero read counts. For this type of data a variety of zero-inflated negative binomial models have been proposed (e.g. MAST, SCDE).

```
d <- 0.5;
counts <- rnbinom(
  1000,
  mu = 10,
  size = 100
)
counts[runif(1000) < d] <- 0
hist(
  counts,
  col = "grey50",
  xlab = "Read Counts",
  main = "Zero-inflated NB"
)
```

Mean:  $\mu = \mu u \cdot (1 - d)$

## Zero-inflated NB

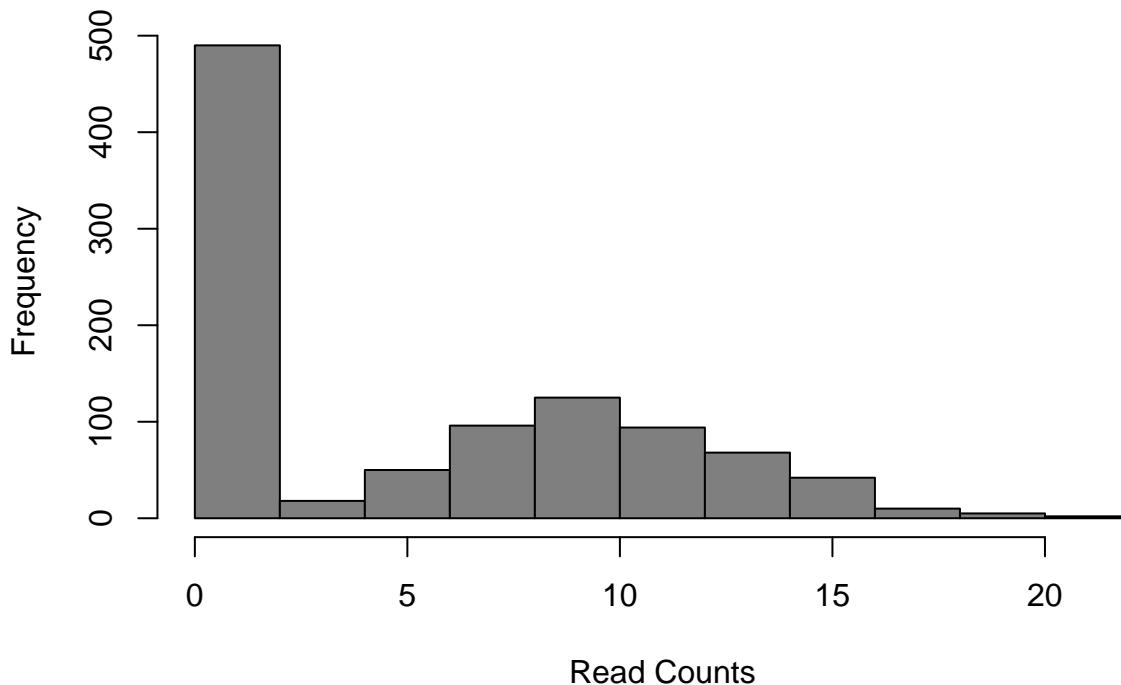


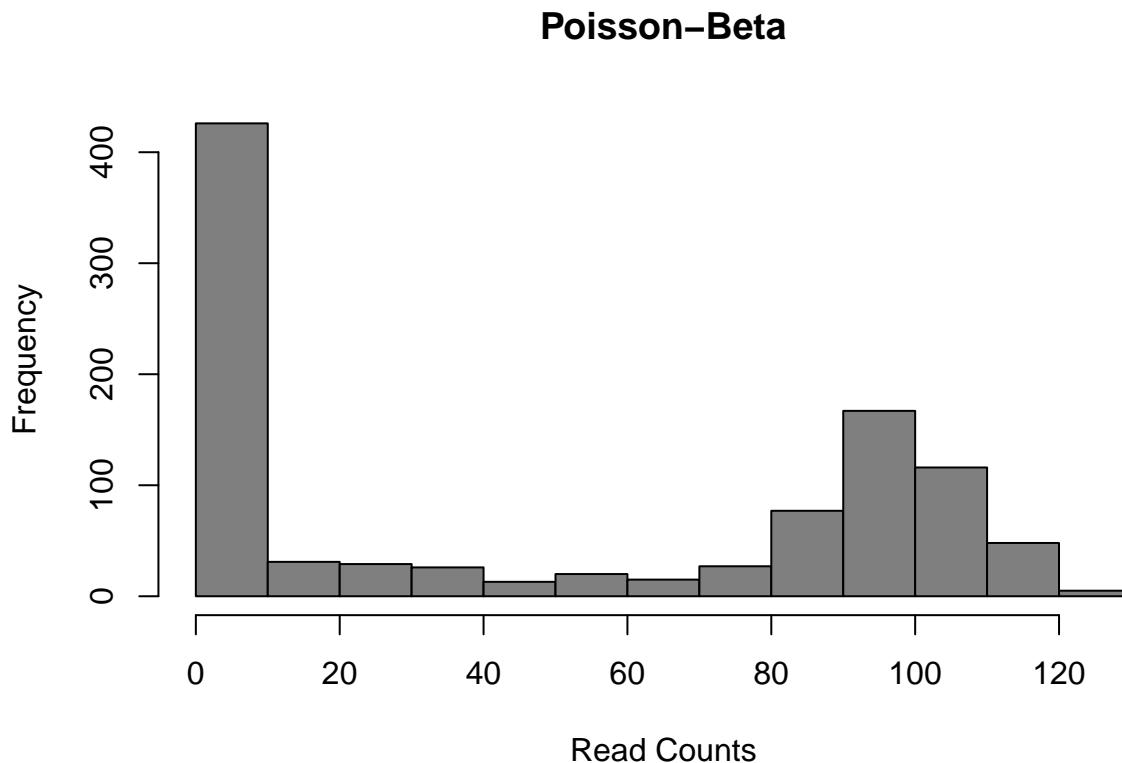
Figure 8.13: Zero-inflated Negative Binomial distribution

$$\text{Variance: } \sigma^2 = \mu \cdot (1 - d) \cdot (1 + d \cdot \mu + \mu/\text{size})$$

These models introduce a new parameter  $d$ , for the dropout rate, to the negative binomial model. As we saw in Chapter 19, the dropout rate of a gene is strongly correlated with the mean expression of the gene. Different zero-inflated negative binomial models use different relationships between mu and d and some may fit  $\mu$  and  $d$  to the expression of each gene independently.

Finally, several methods use a Poisson-Beta distribution which is based on a mechanistic model of transcriptional bursting. There is strong experimental support for this model (Kim and Marioni, 2013) and it provides a good fit to scRNA-seq data but it is less easy to use than the negative-binomial models and much less existing methods upon which to build than the negative binomial model.

```
a <- 0.1
b <- 0.1
g <- 100
lambda <- rbeta(1000, a, b)
counts <- sapply(g*lambda, function(l) {rpois(1, lambda = l)})
hist(
  counts,
  col = "grey50",
  xlab = "Read Counts",
  main = "Poisson-Beta"
)
```



Mean:  $\mu = g \cdot a / (a + b)$

Variance:  $\sigma^2 = g^2 \cdot a \cdot b / ((a + b + 1) \cdot (a + b)^2)$

This model uses three parameters:  $a$  the rate of activation of transcription;  $b$  the rate of inhibition of transcription; and  $g$  the rate of transcript production while transcription is active at the locus. Differential expression methods may test each of the parameters for differences across groups or only one (often  $g$ ).

All of these models may be further expanded to explicitly account for other sources of gene expression differences such as batch-effect or library depth depending on the particular DE algorithm.

**Exercise:** Vary the parameters of each distribution to explore how they affect the distribution of gene expression. How similar are the Poisson-Beta and Negative Binomial models?

## 8.7 DE in a real dataset

```
library(scRNA.seq.funcs)
library(edgeR)
library(monocle)
library(MAST)
library(ROCR)
set.seed(1)
```

### 8.7.1 Introduction

To test different single-cell differential expression methods we will be using the Blischak dataset from Chapters 7-17. For this experiment bulk RNA-seq data for each cell-line was generated in addition to single-cell data. We will use the differentially expressed genes identified using standard methods on the respective bulk data as

the ground truth for evaluating the accuracy of each single-cell method. To save time we have pre-computed these for you. You can run the commands below to load these data.

```
DE <- read.table("tung/TPs.txt")
notDE <- read.table("tung/TNs.txt")
GroundTruth <- list(
  DE = as.character(unlist(DE)),
  notDE = as.character(unlist(notDE))
)
```

This ground truth has been produced for the comparison of individual NA19101 to NA19239. Now load the respective single-cell data:

```
molecules <- read.table("tung/molecules.txt", sep = "\t")
anno <- read.table("tung/annotation.txt", sep = "\t", header = TRUE)
keep <- anno[,1] == "NA19101" | anno[,1] == "NA19239"
data <- molecules[,keep]
group <- anno[keep,1]
batch <- anno[keep,4]
# remove genes that aren't expressed in at least 6 cells
gkeep <- rowSums(data > 0) > 5;
counts <- data[gkeep,]
# Library size normalization
lib_size = colSums(counts)
norm <- t(t(counts)/lib_size * median(lib_size))
# Variant of CPM for datasets with library sizes of fewer than 1 mil molecules
```

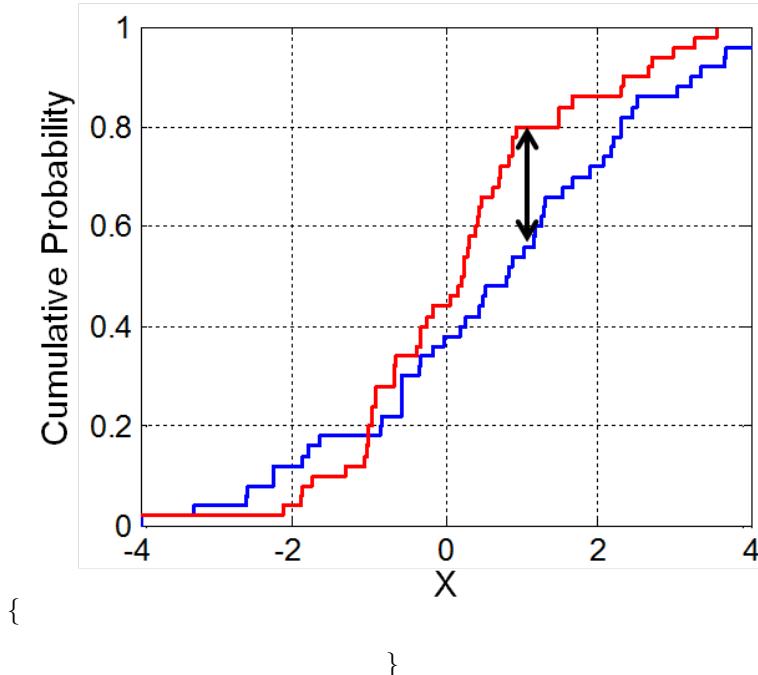
Now we will compare various single-cell DE methods. Note that we will only be running methods which are available as R-packages and run relatively quickly.

## 8.7.2 Kolmogorov-Smirnov test

The types of test that are easiest to work with are non-parametric ones. The most commonly used non-parametric test is the Kolmogorov-Smirnov test (KS-test) and we can use it to compare the distributions for each gene in the two individuals.

The KS-test quantifies the distance between the empirical cumulative distributions of the expression of each gene in each of the two populations. It is sensitive to changes in mean expression and changes in variability. However it assumes data is continuous and may perform poorly when data contains a large number of identical values (eg. zeros). Another issue with the KS-test is that it can be very sensitive for large sample sizes and thus it may end up as significant even though the magnitude of the difference is very small.

\begin{figure}



\caption{Illustration of the two-sample Kolmogorov–Smirnov statistic. Red and blue lines each correspond to an empirical distribution function, and the black arrow is the two-sample KS statistic. (taken from here)} \end{figure}

Now run the test:

```
pVals <- apply(
  norm, 1, function(x) {
    ks.test(
      x[group == "NA19101"],
      x[group == "NA19239"]
    )$p.value
  }
)
# multiple testing correction
pVals <- p.adjust(pVals, method = "fdr")
```

This code “applies” the function to each row (specified by 1) of the expression matrix, data. In the function we are returning just the p.value from the ks.test output. We can now consider how many of the ground truth positive and negative DE genes are detected by the KS-test:

### 8.7.2.1 Evaluating Accuracy

```
sigDE <- names(pVals)[pVals < 0.05]
length(sigDE)

## [1] 5095
# Number of KS-DE genes
sum(GroundTruth$DE %in% sigDE)

## [1] 792
```

```
# Number of KS-DE genes that are true DE genes
sum(GroundTruth$notDE %in% sigDE)
```

```
## [1] 3190
# Number of KS-DE genes that are truly not-DE
```

As you can see many more of our ground truth negative genes were identified as DE by the KS-test (false positives) than ground truth positive genes (true positives), however this may be due to the larger number of notDE genes thus we typically normalize these counts as the True positive rate (TPR),  $TP/(TP + FN)$ , and False positive rate (FPR),  $FP/(FP+TP)$ .

```
tp <- sum(GroundTruth$DE %in% sigDE)
fp <- sum(GroundTruth$notDE %in% sigDE)
tn <- sum(GroundTruth$notDE %in% names(pVals)[pVals >= 0.05])
fn <- sum(GroundTruth$DE %in% names(pVals)[pVals >= 0.05])
tpr <- tp/(tp + fn)
fpr <- fp/(fp + tn)
cat(c(tpr, fpr))
```

```
## 0.7346939 0.2944706
```

Now we can see the TPR is much higher than the FPR indicating the KS test is identifying DE genes.

So far we've only evaluated the performance at a single significance threshold. Often it is informative to vary the threshold and evaluate performance across a range of values. This is then plotted as a receiver-operating-characteristic curve (ROC) and a general accuracy statistic can be calculated as the area under this curve (AUC). We will use the ROCR package to facilitate this plotting.

```
# Only consider genes for which we know the ground truth
pVals <- pVals[names(pVals) %in% GroundTruth$DE |
               names(pVals) %in% GroundTruth$notDE]
truth <- rep(1, times = length(pVals));
truth[names(pVals) %in% GroundTruth$DE] = 0;
pred <- ROCR::prediction(pVals, truth)
perf <- ROCR::performance(pred, "tpr", "fpr")
ROCR::plot(perf)

aucObj <- ROCR::performance(pred, "auc")
aucObj@y.values[[1]] # AUC
```

```
## [1] 0.7954796
```

Finally to facilitate the comparisons of other DE methods let's put this code into a function so we don't need to repeat it:

```
DE_Quality_AUC <- function(pVals) {
  pVals <- pVals[names(pVals) %in% GroundTruth$DE |
                 names(pVals) %in% GroundTruth$notDE]
  truth <- rep(1, times = length(pVals));
  truth[names(pVals) %in% GroundTruth$DE] = 0;
  pred <- ROCR::prediction(pVals, truth)
  perf <- ROCR::performance(pred, "tpr", "fpr")
  ROCR::plot(perf)
  aucObj <- ROCR::performance(pred, "auc")
  return(aucObj@y.values[[1]])
}
```

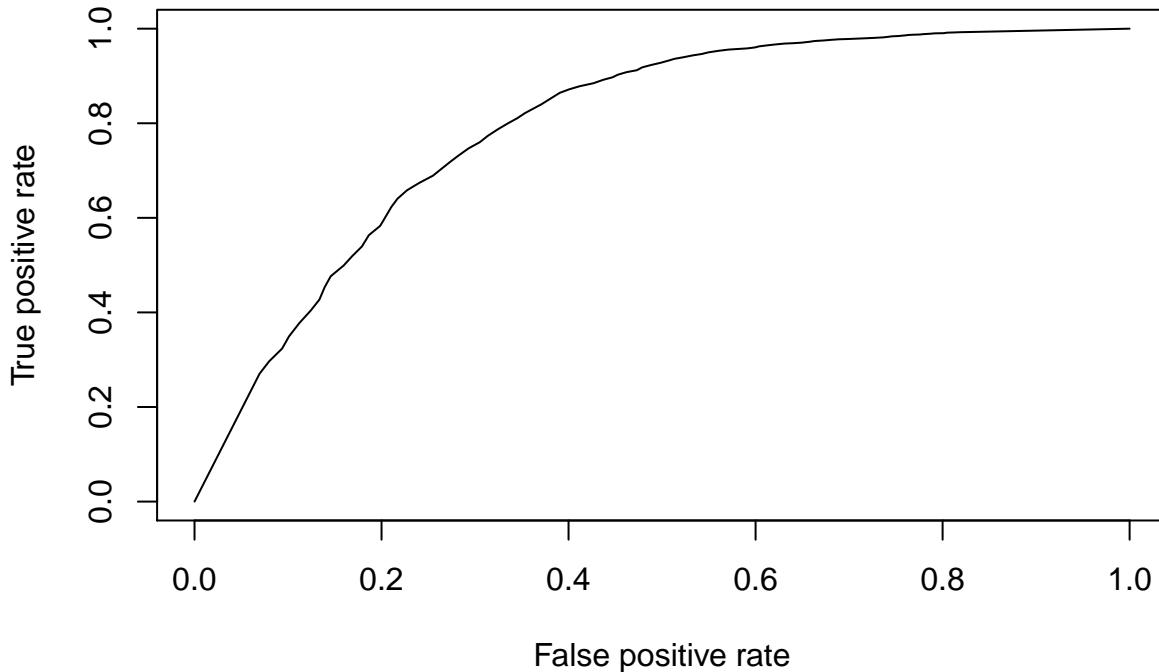


Figure 8.14: ROC curve for KS-test.

### 8.7.3 Wilcox/Mann-Whitney-U Test

The Wilcoxon-rank-sum test is another non-parametric test, but tests specifically if values in one group are greater/less than the values in the other group. Thus it is often considered a test for difference in median expression between two groups; whereas the KS-test is sensitive to any change in distribution of expression values.

```
pVals <- apply(
  norm, 1, function(x) {
    wilcox.test(
      x[group == "NA19101"],
      x[group == "NA19239"]
    )$p.value
  }
)
# multiple testing correction
pVals <- p.adjust(pVals, method = "fdr")
DE_Quality_AUC(pVals)

## [1] 0.8320326
```

### 8.7.4 edgeR

We've already used edgeR for differential expression in Chapter 7.10. edgeR is based on a negative binomial model of gene expression and uses a generalized linear model (GLM) framework, the enables us to include other factors such as batch to the model.

```
dge <- DGEList(
  counts = counts,
```

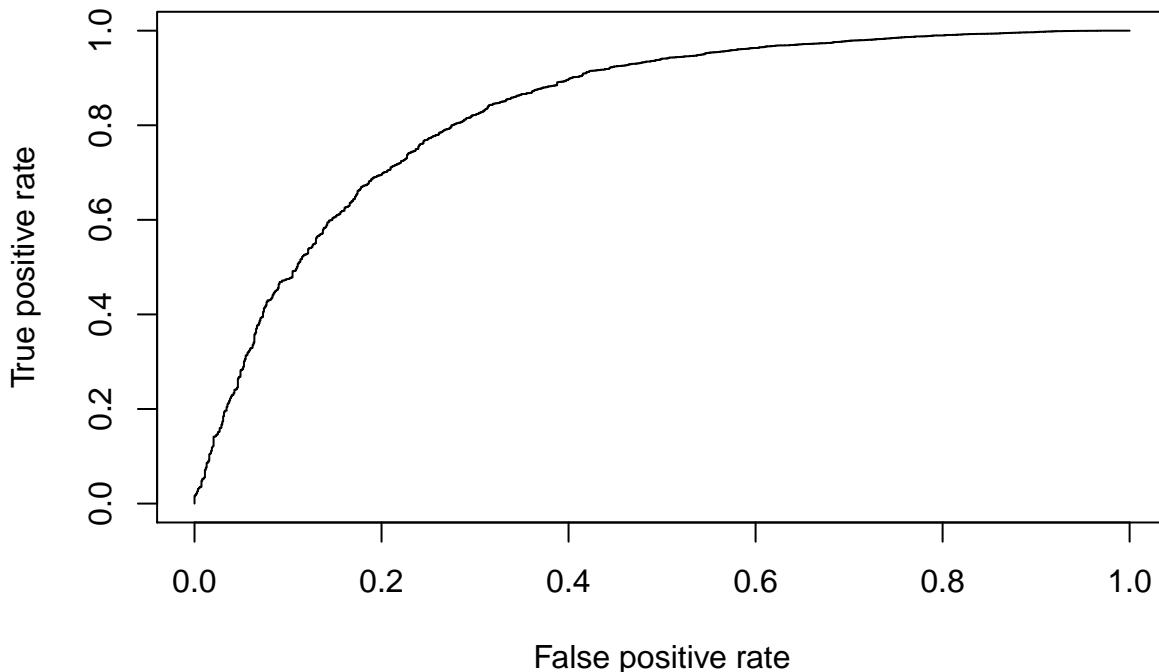


Figure 8.15: ROC curve for Wilcox test.

```

norm.factors = rep(1, length(counts[,1])),
group = group
)
group_edgeR <- factor(group)
design <- model.matrix(~ group_edgeR)
dge <- estimateDisp(dge, design = design, trend.method = "none")
fit <- glmFit(dge, design)
res <- glmlRT(fit)
pVals <- res$table[,4]
names(pVals) <- rownames(res$table)

pVals <- p.adjust(pVals, method = "fdr")
DE_Quality_AUC(pVals)

## [1] 0.8466764

```

### 8.7.5 Monocle

Monocle can use several different models for DE. For count data it recommends the Negative Binomial model (`negbinomial.size`). For normalized data it recommends log-transforming it then using a normal distribution (`gaussianff`). Similar to edgeR this method uses a GLM framework so in theory can account for batches, however in practice the model fails for this dataset if batches are included.

```

pd <- data.frame(group = group, batch = batch)
rownames(pd) <- colnames(counts)
pd <- new("AnnotatedDataFrame", data = pd)

Obj <- newCellDataSet(
  as.matrix(counts),

```

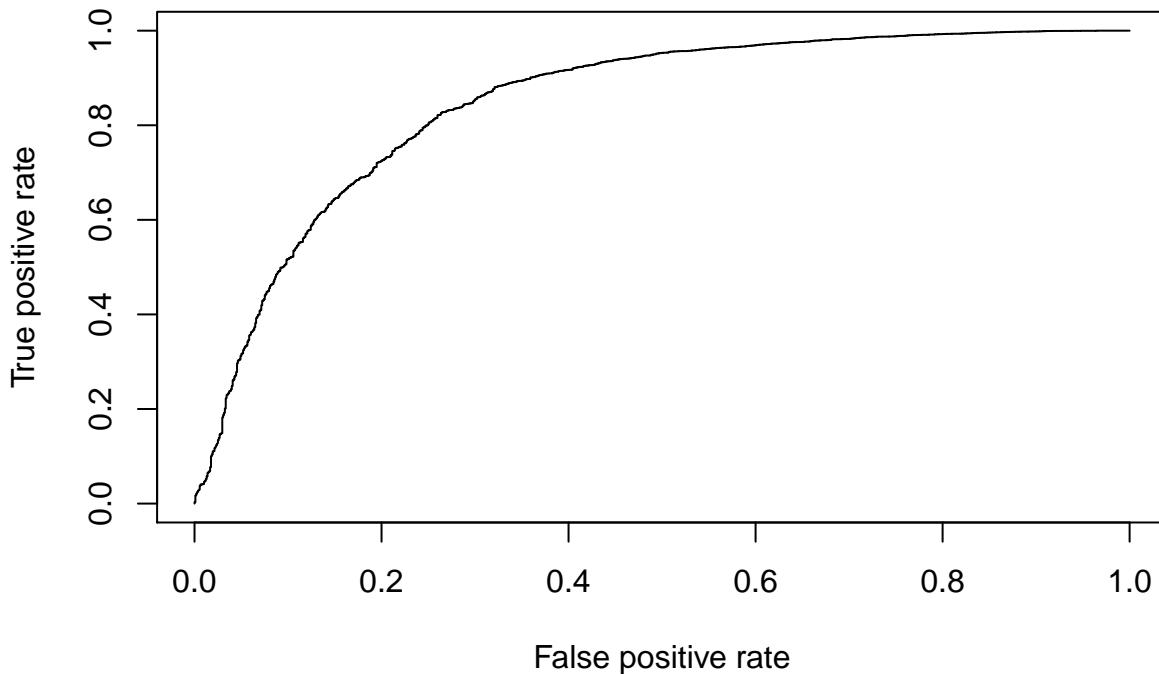


Figure 8.16: ROC curve for edgeR.

```

phenoData = pd,
expressionFamily = negbinomial.size()
)
Obj <- estimateSizeFactors(Obj)
Obj <- estimateDispersions(Obj)
res <- differentialGeneTest(Obj, fullModelFormulaStr = "~group")

pVals <- res[,3]
names(pVals) <- rownames(res)
pVals <- p.adjust(pVals, method = "fdr")
DE_Quality_AUC(pVals)

## [1] 0.8252662

```

**Exercise:** Compare the results using the negative binomial model on counts and those from using the normal/gaussian model (`gaussianff()`) on log-transformed normalized counts.

**Answer:**

```
## [1] 0.7357829
```

### 8.7.6 MAST

MAST is based on a zero-inflated negative binomial model. It tests for differential expression using a hurdle model to combine tests of discrete (0 vs not zero) and continuous (non-zero values) aspects of gene expression. Again this uses a linear modelling framework to enable complex models to be considered.

```

log_counts <- log(counts + 1) / log(2)
fData <- data.frame(names = rownames(log_counts))
rownames(fData) <- rownames(log_counts);

```

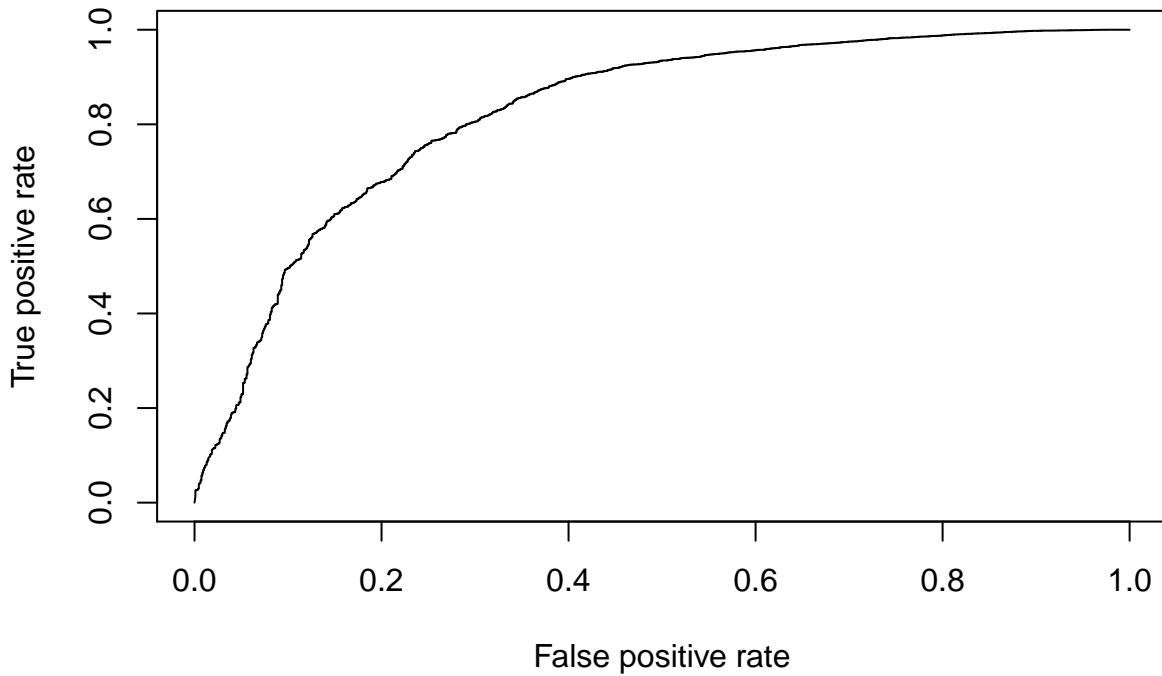


Figure 8.17: ROC curve for Monocle.

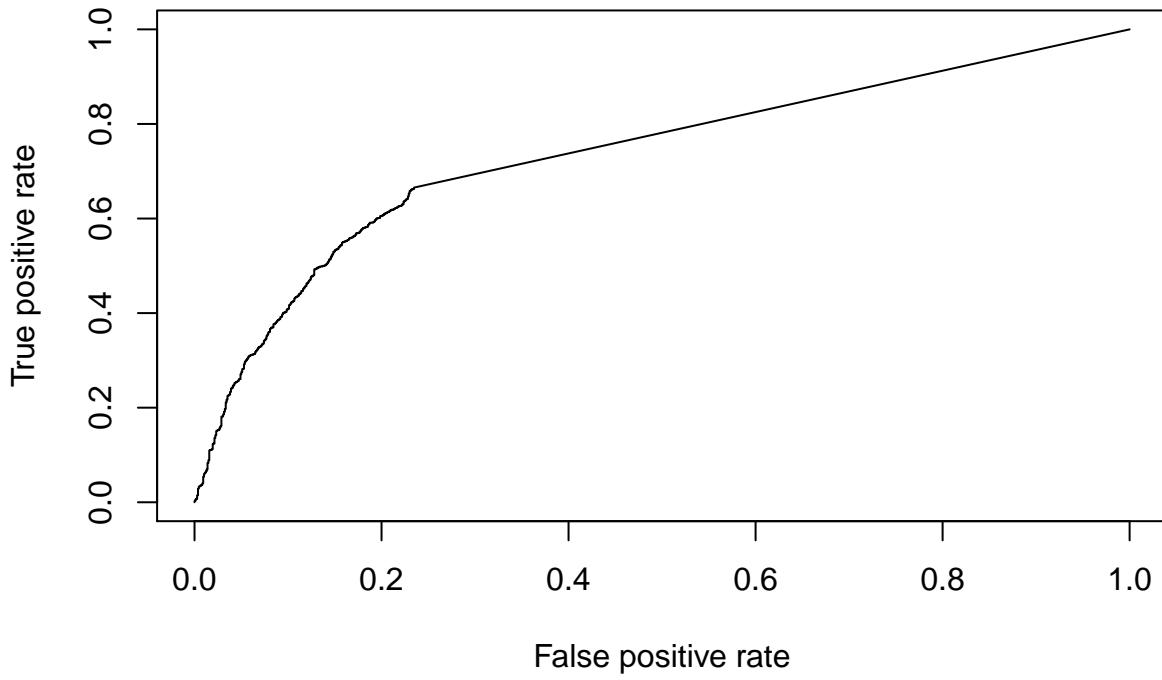


Figure 8.18: ROC curve for Monocle-gaussian.

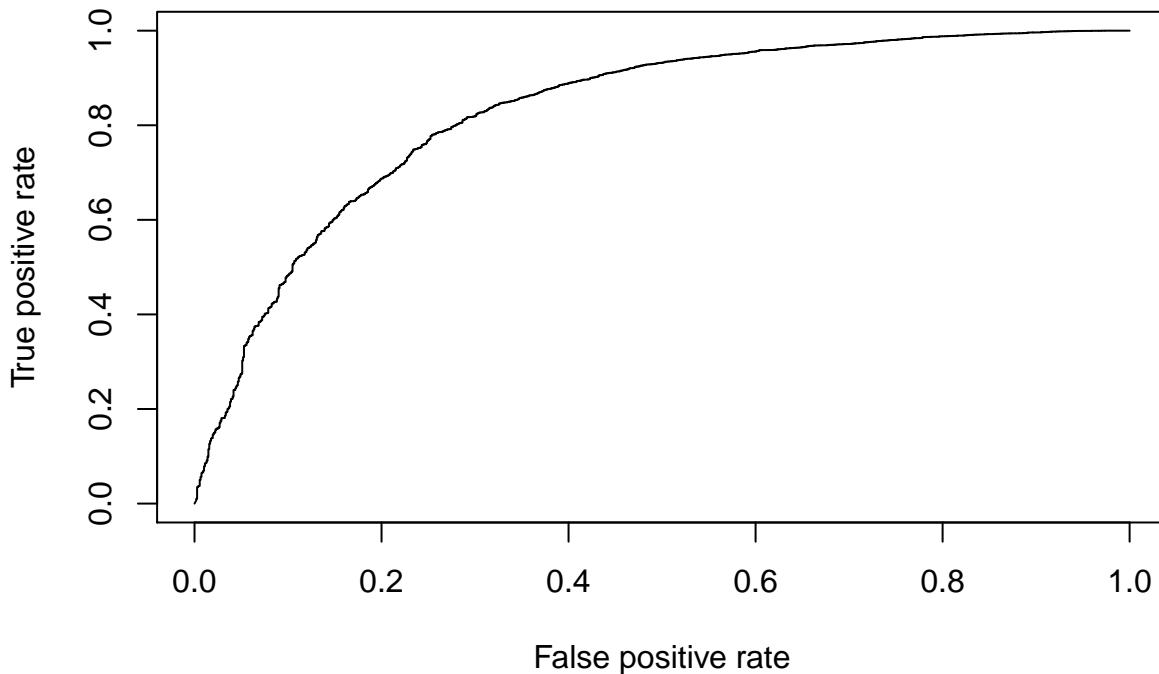


Figure 8.19: ROC curve for MAST.

```

cData <- data.frame(cond = group)
rownames(cData) <- colnames(log_counts)

obj <- FromMatrix(as.matrix(log_counts), cData, fData)
colData(obj)$cngeneson <- scale(colSums(assay(obj) > 0))
cond <- factor(colData(obj)$cond)

# Model expression as function of condition & number of detected genes
zlmCond <- zlm.SingleCellAssay(~ cond + cngeneson, obj)

## Warning: 'zlm.SingleCellAssay' is deprecated.
## Use 'zlm' instead.
## See help("Deprecated")

## Warning in .nextMethod(object = object, value = value): Coefficients
## condNA19239 are never estimable and will be dropped.
summaryCond <- summary(zlmCond, doLRT = "condNA19101")
summaryDt <- summaryCond$datatable

summaryDt <- as.data.frame(summaryDt)
pVals <- unlist(summaryDt[summaryDt$component == "H", 4]) # H = hurdle model
names(pVals) <- unlist(summaryDt[summaryDt$component == "H", 1])
pVals <- p.adjust(pVals, method = "fdr")
DE_Quality_AUC(pVals)

## [1] 0.8284046

```

### 8.7.7 Slow Methods (>1h to run)

These methods are too slow to run today but we encourage you to try them out on your own:

### 8.7.8 BPSC

BPSC uses the Poisson-Beta model of single-cell gene expression, which we discussed in the previous chapter, and combines it with generalized linear models which we've already encountered when using edgeR. BPSC performs comparisons of one or more groups to a reference group (“control”) and can include other factors such as batches in the model.

```
library(BPSC)
bpsc_data <- norm[,batch=="NA19101.r1" | batch=="NA19239.r1"]
bpsc_group = group[batch=="NA19101.r1" | batch=="NA19239.r1"]

control_cells <- which(bpsc_group == "NA19101")
design <- model.matrix(~bpsc_group)
coef=2 # group label
res=BPglm(data=bpsc_data, controlIds=control_cells, design=design, coef=coef,
           estIntPar=FALSE, useParallel = FALSE)
pVals = res$PVAL
pVals <- p.adjust(pVals, method = "fdr")
DE_Quality_AUC(pVals)
```

### 8.7.9 SCDE

SCDE is the first single-cell specific DE method. It fits a zero-inflated negative binomial model to expression data using Bayesian statistics. The usage below tests for differences in mean expression of individual genes across groups but recent versions include methods to test for differences in mean expression or dispersion of groups of genes, usually representing a pathway.

```
library(scde)
cnts <- apply(
  counts,
  2,
  function(x) {
    storage.mode(x) <- 'integer'
    return(x)
  }
)
names(group) <- 1:length(group)
colnames(cnts) <- 1:length(group)
o.ifm <- scde::scde.error.models(
  counts = cnts,
  groups = group,
  n.cores = 1,
  threshold.segmentation = TRUE,
  save.crossfit.plots = FALSE,
  save.model.plots = FALSE,
  verbose = 0,
  min.size.entries = 2
)
priors <- scde::scde.expression.prior(
```

```

models = o.ifm,
counts = cnts,
length.out = 400,
show.plot = FALSE
)
resSCDE <- scde::scde.expression.difference(
  o.ifm,
  cnts,
  priors,
  groups = group,
  n.randomizations = 100,
  n.cores = 1,
  verbose = 0
)
# Convert Z-scores into 2-tailed p-values
pVals <- pnorm(abs(resSCDE$cZ), lower.tail = FALSE) * 2
DE_Quality_AUC(pVals)

```

### 8.7.10 sessionInfo()

```

## R version 3.4.3 (2017-11-30)
## Platform: x86_64-pc-linux-gnu (64-bit)
## Running under: Debian GNU/Linux 9 (stretch)
##
## Matrix products: default
## BLAS: /usr/lib/openblas-base/libblas.so.3
## LAPACK: /usr/lib/libopenblas-p-r0.2.19.so
##
## locale:
## [1] LC_CTYPE=en_US.UTF-8          LC_NUMERIC=C
## [3] LC_TIME=en_US.UTF-8          LC_COLLATE=en_US.UTF-8
## [5] LC_MONETARY=en_US.UTF-8       LC_MESSAGES=C
## [7] LC_PAPER=en_US.UTF-8          LC_NAME=C
## [9] LC_ADDRESS=C                  LC_TELEPHONE=C
## [11] LC_MEASUREMENT=en_US.UTF-8   LC_IDENTIFICATION=C
##
## attached base packages:
## [1] splines      stats4      parallel    methods     stats       graphics   grDevices
## [8] utils        datasets    base
##
## other attached packages:
## [1] ROCOCR_1.0-7           gplots_3.0.1
## [3] MAST_1.4.1             SummarizedExperiment_1.8.1
## [5] DelayedArray_0.4.1      matrixStats_0.53.0
## [7] GenomicRanges_1.30.1    GenomeInfoDb_1.14.0
## [9] IRanges_2.12.0          S4Vectors_0.16.0
## [11] monocle_2.6.1          DDRTTree_0.1.5
## [13] irlba_2.3.2            VGAM_1.0-4
## [15] ggplot2_2.2.1           Biobase_2.38.0
## [17] BiocGenerics_0.24.0     Matrix_1.2-7.1
## [19] edgeR_3.20.8            limma_3.34.8
## [21] scRNA.seq.funcs_0.1.0    knitr_1.19

```

```
##
## loaded via a namespace (and not attached):
## [1] bitops_1.0-6           RColorBrewer_1.1-2    rprojroot_1.3-2
## [4] tools_3.4.3            backports_1.1.2      R6_2.2.2
## [7] KernSmooth_2.23-15    hypergeo_1.2-13      lazyeval_0.2.1
## [10] colorspace_1.3-2      gridExtra_2.3        moments_0.14
## [13] compiler_3.4.3        orthopolynom_1.0-5   bookdown_0.6
## [16] slam_0.1-42          caTools_1.17.1       scales_0.5.0
## [19] stringr_1.2.0         digest_0.6.15       rmarkdown_1.8
## [22] XVector_0.18.0       pkgconfig_2.0.0.1    htmltools_0.3.6
## [25] rlang_0.1.6           FNN_1.1              bindr_0.1
## [28] combinat_0.0-8        gtools_3.5.0         dplyr_0.7.4
## [31] RCurl_1.95-4.10       magrittr_1.5         GenomeInfoDbData_1.0.0
## [34] Rcpp_0.12.15          munsell_0.4.3       abind_1.4-5
## [37] viridis_0.5.0          stringi_1.1.6       yaml_2.1.16
## [40] MASS_7.3-45           zlibbioc_1.24.0     Rtsne_0.13
## [43] plyr_1.8.4             grid_3.4.3          gdata_2.18.0
## [46] ggrepel_0.7.0          conffrac_1.1-11     lattice_0.20-34
## [49] locfit_1.5-9.1         pillar_1.1.0         igraph_1.1.2
## [52] reshape2_1.4.3          glue_1.2.0          evaluate_0.10.1
## [55] data.table_1.10.4-3    deSolve_1.20        gtable_0.2.0
## [58] RANN_2.5.1              assertthat_0.2.0    xfun_0.1
## [61] qlcMatrix_0.9.5        HSMMSingleCell_0.112.0 viridisLite_0.3.0
## [64] tibble_1.4.2            pheatmap_1.0.8       elliptic_1.3-7
## [67] bindrcpp_0.2            cluster_2.0.6       fastICA_1.2-1
## [70] densityClust_0.3       statmod_1.4.30
```

## 8.8 Comparing/Combining scRNASeq datasets

```
library(scater)
library(SingleCellExperiment)
```

### 8.8.1 Introduction

As more and more scRNA-seq datasets become available, carrying merged\_seurat comparisons between them is key. There are two main approaches to comparing scRNASeq datasets. The first approach is “label-centric” which is focused on trying to identify equivalent cell-types/states across datasets by comparing individual cells or groups of cells. The other approach is “cross-dataset normalization” which attempts to computationally remove experiment-specific technical/biological effects so that data from multiple experiments can be combined and jointly analyzed.

The label-centric approach can be used with dataset with high-confidence cell-annotations, e.g. the Human Cell Atlas (HCA) (Regev et al., 2017) or the Tabula Muris (?) once they are completed, to project cells or clusters from a new sample onto this reference to consider tissue composition and/or identify cells with novel/unknown identity. Conceptually, such projections are similar to the popular BLAST method (Altschul et al., 1990), which makes it possible to quickly find the closest match in a database for a newly identified nucleotide or amino acid sequence. The label-centric approach can also be used to compare datasets of similar biological origin collected by different labs to ensure that the annotation and the analysis is consistent.

The cross-dataset normalization approach can also be used to compare datasets of similar biological origin, unlike the label-centric approach it enables the joint analysis of multiple datasets to facilitate the

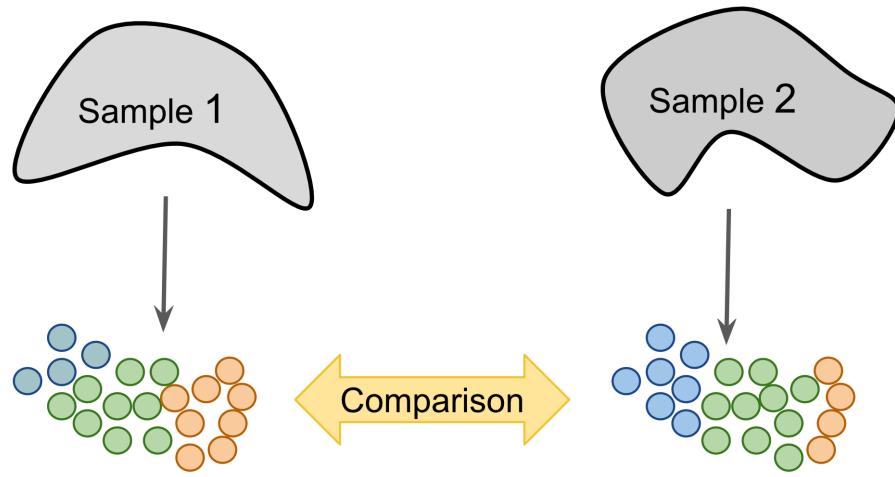


Figure 8.20: Label-centric dataset comparison can be used to compare the annotations of two different samples.

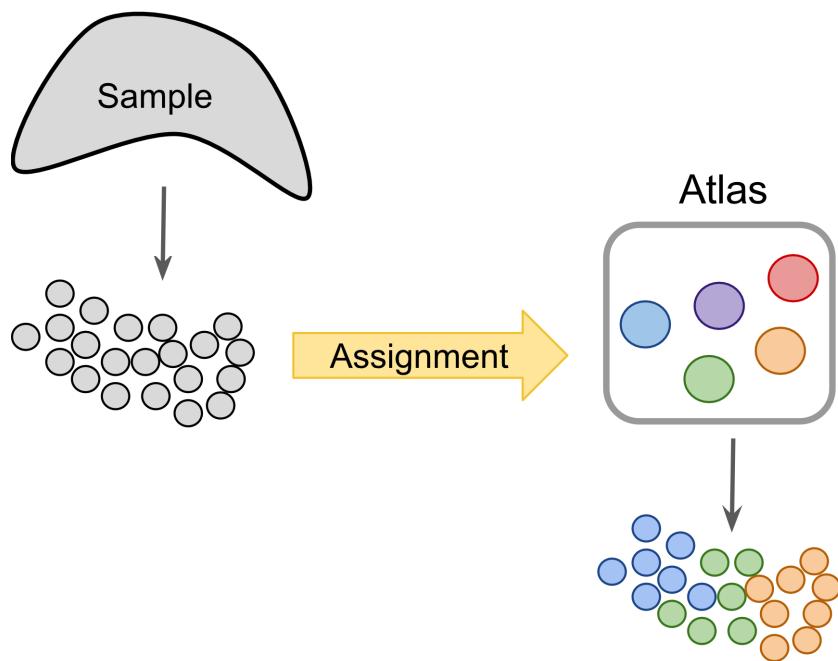


Figure 8.21: Label-centric dataset comparison can project cells from a new experiment onto an annotated reference.

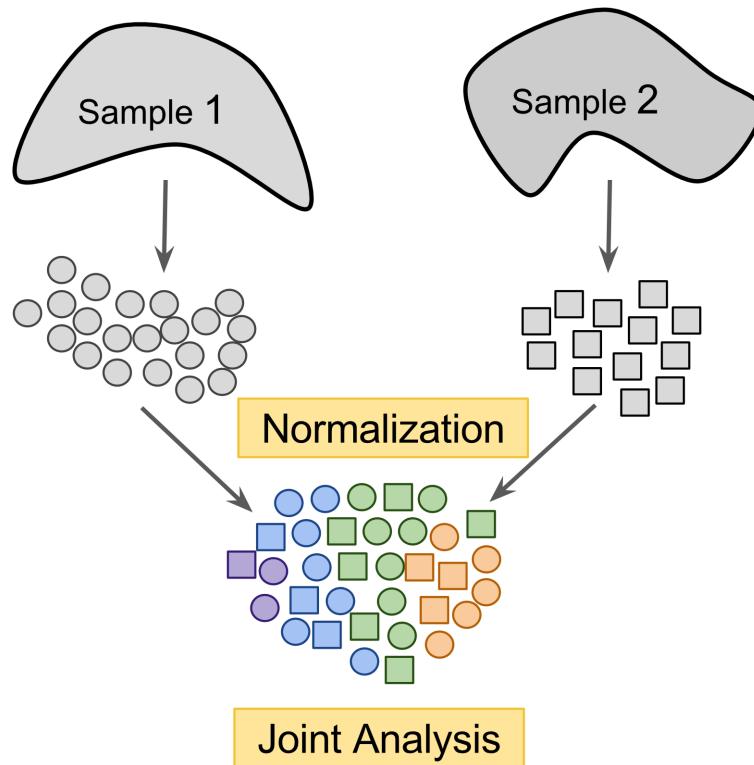


Figure 8.22: Cross-dataset normalization enables joint-analysis of 2+ scRNASeq datasets.

identification of rare cell-types which may be too sparsely sampled in each individual dataset to be reliably detected. However, cross-dataset normalization is not applicable to very large and diverse references since it assumes a significant portion of the biological variability in each of the datasets overlaps with others.

### 8.8.2 Datasets

We will run these methods on two human pancreas datasets: (Muraro et al., 2016) and (Segerstolpe et al., 2016). Since the pancreas has been widely studied, these datasets are well annotated.

```
muraro <- readRDS("pancreas/muraro.rds")
segerstolpe <- readRDS("pancreas/segerstolpe.rds")
```

This data has already been formatted for scmap. Cell type labels must be stored in the `cell_type1` column of the `colData` slots, and gene ids that are consistent across both datasets must be stored in the `feature_symbol` column of the `rowData` slots.

First, let's check our gene-ids match across both datasets:

```
sum(rowData(muraro)$feature_symbol %in% rowData(segerstolpe)$feature_symbol)/nrow(muraro)
## [1] 0.9599519
sum(rowData(segerstolpe)$feature_symbol %in% rowData(muraro)$feature_symbol)/nrow(segerstolpe)
## [1] 0.719334
```

Here we can see that 96% of the genes present in muraro match genes in segerstolpe and 72% of genes in segerstolpe are present in muraro. This is as expected because the segerstolpe dataset was more

deeply sequenced than the muraro dataset. However, it highlights some of the difficulties in comparing scRNASeq datasets.

We can confirm this by checking the overall size of these two datasets.

```
dim(muraro)

## [1] 19127 2126

dim(segerstolpe)

## [1] 25525 3514
```

In addition, we can check the cell-type annotations for each of these dataset using the command below:

```
summary(factor(colData(muraro)$cell_type1))

##      acinar      alpha      beta      delta      ductal      endothelial
##      219        812       448       193        245          21
##      epsilon     gamma mesenchymal    unclear
##      3           101        80         4

summary(factor(colData(segerstolpe)$cell_type1))

##      acinar      alpha      beta
##      185        886       270
##      co-expression      delta      ductal
##      39          114       386
##      endothelial      epsilon      gamma
##      16            7        197
##      mast      MHC class II    not applicable
##      7             5        1305
##      PSC      unclassified unclassified endocrine
##      54            2          41
```

Here we can see that even though both datasets considered the same biological tissue the two datasets, they have been annotated with slightly different sets of cell-types. If you are familiar with pancreas biology you might recognize that the pancreatic stellate cells (PSCs) in segerstolpe are a type of mesenchymal stem cell which would fall under the “mesenchymal” type in muraro. However, it isn’t clear whether these two annotations should be considered synonymous or not. We can use label-centric comparison methods to determine if these two cell-type annotations are indeed equivalent.

Alternatively, we might be interested in understanding the function of those cells that were “unclassified endocrine” or were deemed too poor quality (“not applicable”) for the original clustering in each dataset by leveraging information across datasets. Either we could attempt to infer which of the existing annotations they most likely belong to using label-centric approaches or we could try to uncover a novel cell-type among them (or a sub-type within the existing annotations) using cross-dataset normalization.

To simplify our demonstration analyses we will remove the small classes of unassigned cells, and the poor quality cells. We will retain the “unclassified endocrine” to see if any of these methods can elucidate what cell-type they belong to.

```
segerstolpe <- segerstolpe[, colData(segerstolpe)$cell_type1 != "unclassified"]
segerstolpe <- segerstolpe[, colData(segerstolpe)$cell_type1 != "not applicable",]
muraro <- muraro[, colData(muraro)$cell_type1 != "unclear"]
```

### 8.8.3 Projecting cells onto annotated cell-types (scmap)

```
library(scmap)
set.seed(1234567)
```

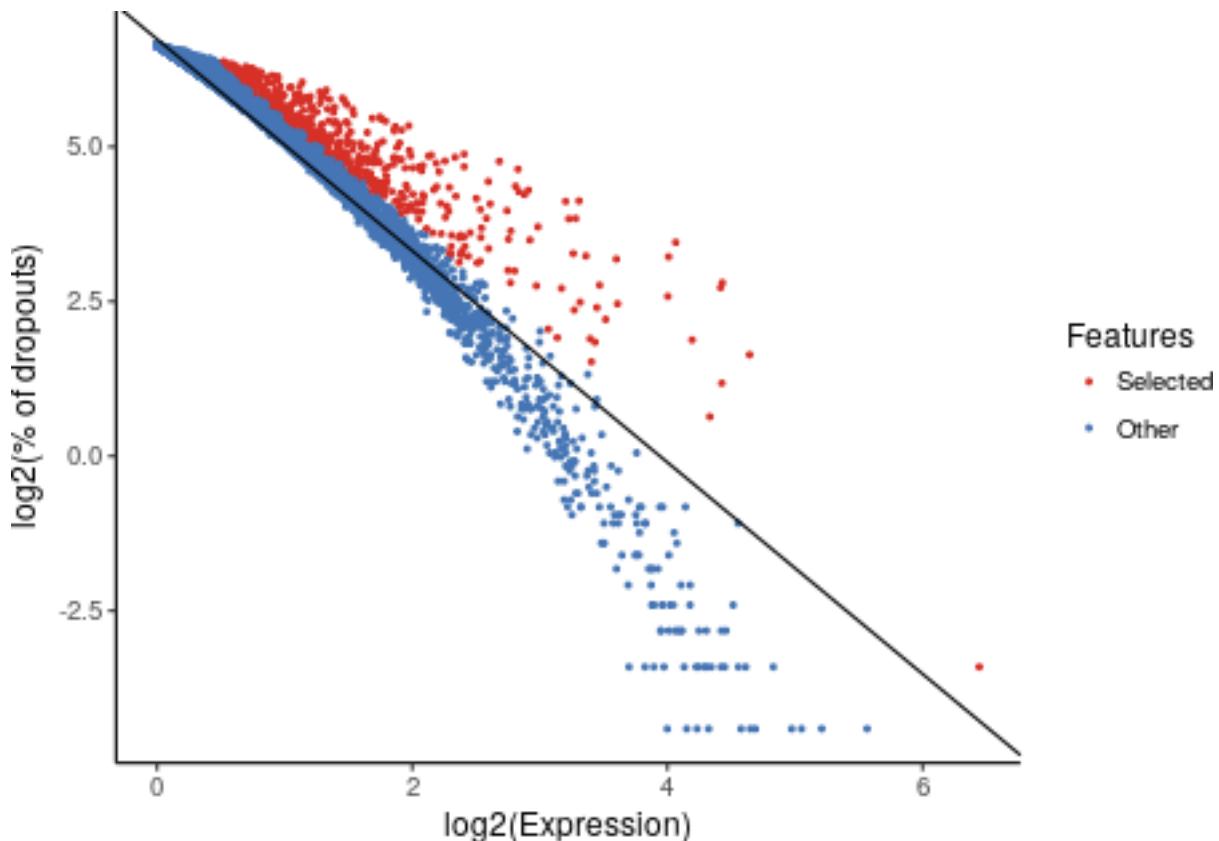
We recently developed **scmap** (Kiselev and Hemberg, 2017) - a method for projecting cells from a scRNA-seq experiment onto the cell-types identified in other experiments. Additionally, a cloud version of **scmap** can be run for free, with merged\_seurat restrictions, from <http://www.hemberg-lab.cloud/scmap>.

#### 8.8.3.1 Feature Selection

Once we have a **SingleCellExperiment** object we can run **scmap**. First we have to build the “index” of our reference clusters. Since we want to know whether PSCs and mesenchymal cells are synonymous we will project each dataset to the other so we will build an index for each dataset. This requires first selecting the most informative features for the reference dataset.

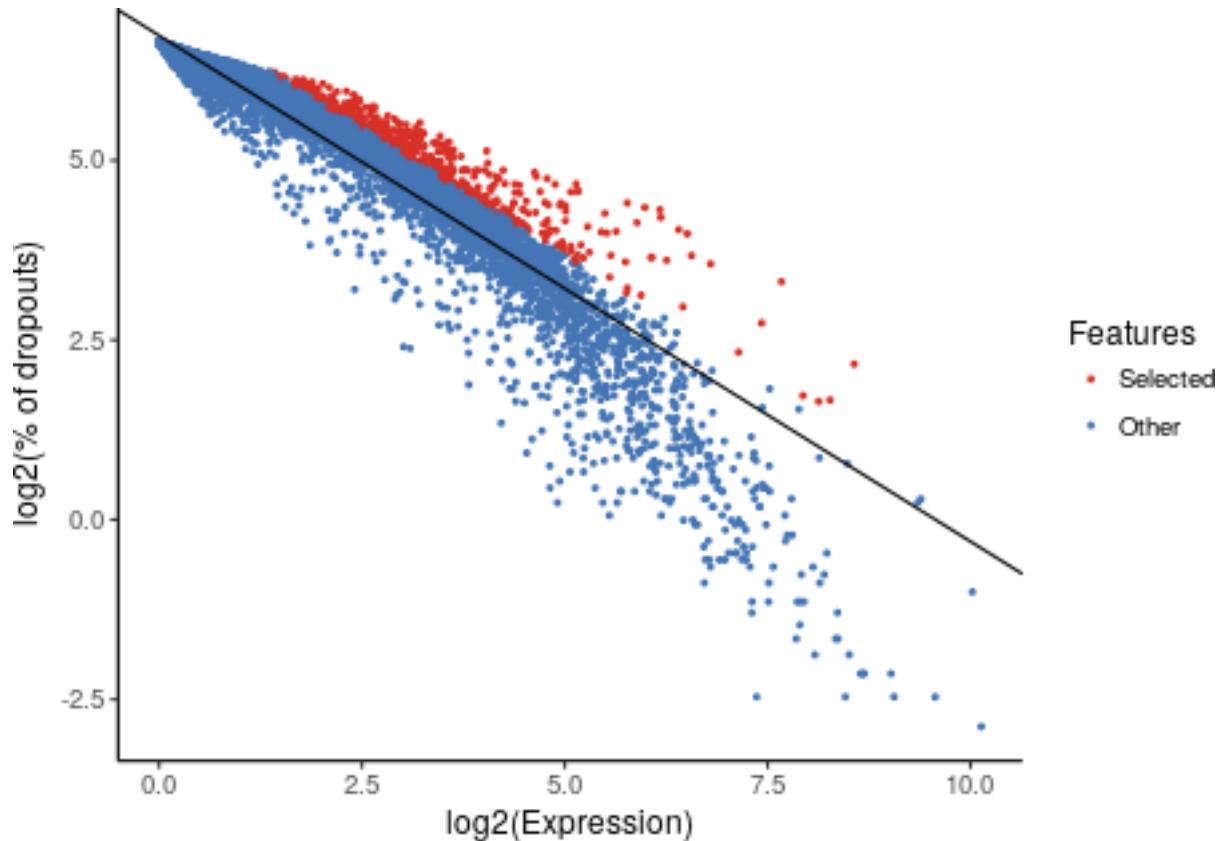
```
muraro <- selectFeatures(muraro, suppress_plot = FALSE)
```

```
## Warning in linearModel(object, n_features): Your object does not contain
## counts() slot. Dropouts were calculated using logcounts() slot...
```



Genes highlighted with the red colour will be used in the further analysis (projection).

```
segerstolpe <- selectFeatures(segerstolpe, suppress_plot = FALSE)
```



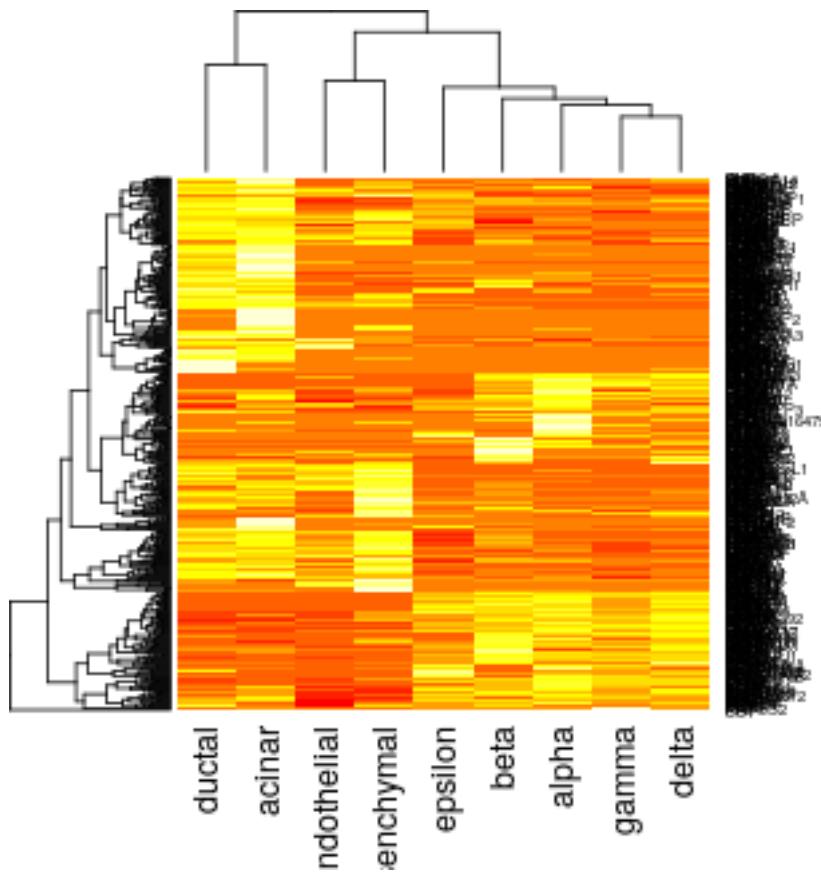
From the y-axis of these plots we can see that scmap uses a dropmerged\_seurat-based feature selection method.

Now calculate the cell-type index:

```
muraro <- indexCluster(muraro)
segerstolpe <- indexCluster(segerstolpe)
```

We can also visualize the index:

```
heatmap(as.matrix(metadata(muraro)$scmap_cluster_index))
```



You may want to adjust your features using the `setFeatures` function if features are too heavily concentrated in only a few cell-types. In this case the dropmerged\_seurat-based features look good so we will just them.

**Exercise** Using the `rowData` of each dataset how many genes were selected as features in both datasets?  
What does this tell you about merged\_seurat these datasets?

**Answer**

### 8.8.3.2 Projecting

`scmap` computes the distance from each cell to each cell-type in the reference index, then applies an empirically derived threshold to determine which cells are assigned to the closest reference cell-type and which are unassigned. To account for differences in sequencing depth distance is calculated using the spearman correlation and cosine distance and only cells with a consistent assignment with both distances are returned as assigned.

We will project the `segerstolpe` dataset to `muraro` dataset:

```
seger_to_muraro <- scmapCluster(
  projection = segerstolpe,
  index_list = list(
    muraro = metadata(muraro)$scmap_cluster_index
  )
)
```

and `muraro` onto `segerstolpe`

```

muraro_to_seger <- scmapCluster(
  projection = muraro,
  index_list = list(
    seger = metadata(segerstolpe)$scmap_cluster_index
  )
)

```

Note that in each case we are projecting to a single dataset but that this could be extended to any number of datasets for which we have computed indices.

Now lets compare the original cell-type labels with the projected labels:

```
table(colData(muraro)$cell_type1, muraro_to_seger$scmap_cluster_labs)
```

```

##
##          acinar alpha beta co-expression delta ductal endothelial
##  acinar      211     0     0           0     0     0       0
##  alpha        1   763     0           18     0     2       0
##  beta         2     1  397           7     2     2       0
##  delta        0     0     2           1   173     0       0
##  ductal       7     0     0           0     0   208       0
##  endothelial  0     0     0           0     0     0       15
##  epsilon       0     0     0           0     0     0       0
##  gamma        2     0     0           0     0     0       0
##  mesenchymal  0     0     0           0     0     1       0
##
##          epsilon gamma MHC class II PSC unassigned
##  acinar      0     0           0     0       8
##  alpha        0     2           0     0      26
##  beta         0     5           1     2      29
##  delta        0     0           0     0      17
##  ductal       0     0           5     3      22
##  endothelial  0     0           0     1       5
##  epsilon       3     0           0     0       0
##  gamma        0   95           0     0       4
##  mesenchymal  0     0           0   77       2

```

Here we can see that cell-types do map to their equivalents in segerstolpe, and importantly we see that all but one of the “mesenchymal” cells were assigned to the “PSC” class.

```
table(colData(segerstolpe)$cell_type1, seger_to_muraro$scmap_cluster_labs)
```

```

##
##          acinar alpha beta delta ductal endothelial
##  acinar      181     0     0     0     4       0
##  alpha        0   869     1     0     0       0
##  beta         0     0  260     0     0       0
##  co-expression 0     7   31     0     0       0
##  delta        0     0     1  111     0       0
##  ductal       0     0     0     0  383       0
##  endothelial  0     0     0     0     0       14
##  epsilon       0     0     0     0     0       0
##  gamma        0     2     0     0     0       0
##  mast          0     0     0     0     0       0
##  MHC class II 0     0     0     0     0       0
##  PSC          0     0     1     0     0       0

```

```

## unclassified endocrine      0   0   0   0   0   0
##
##                           epsilon gamma mesenchymal unassigned
## acinar                      0   0       0       0
## alpha                       0   0       0      16
## beta                        0   0       0      10
## co-expression                 0   0       0       1
## delta                       0   0       0       2
## ductal                      0   0       0       3
## endothelial                  0   0       0       2
## epsilon                      6   0       0       1
## gamma                       0 192       0       3
## mast                         0   0       0       7
## MHC class II                  0   0       0       5
## PSC                          0   0      53       0
## unclassified endocrine      0   0       0      41

```

Again we see cell-types match each other and that all but one of the “PSCs” match the “mesenchymal” cells providing strong evidence that these two annotations should be considered synonymous.

We can also visualize these tables using a Sankey diagram:

```
plot(getSankey(colData(muraro)$cell_type1, muraro_to_seger$scmap_cluster_labs[,1], plot_height=400))
```

**Exercise** How many of the previously unclassified cells would be able to assign to cell-types using scmap?

**Answer**

#### 8.8.4 Cell-to-Cell mapping

scmap can also project each cell in one dataset to its approximate closest neighbouring cell in the reference dataset. This uses a highly optimized search algorithm allowing it to be scaled to very large references (in theory 100,000-millions of cells). However, this process is stochastic so we must fix the random seed to ensure we can reproduce our results.

We have already performed feature selection for this dataset so we can go straight to building the index.

```

set.seed(193047)
segerstolpe <- indexCell(segerstolpe)

## Parameter M was not provided, will use M = n_features / 10 (if n_features <= 1000), where n_features
## Parameter k was not provided, will use k = sqrt(number_of_cells)
muraro <- indexCell(muraro)

## Parameter M was not provided, will use M = n_features / 10 (if n_features <= 1000), where n_features
## Parameter k was not provided, will use k = sqrt(number_of_cells)

```

In this case the index is a series of clusterings of each cell using different sets of features, parameters k and M are the number of clusters and the number of features used in each of these subclusterings. New cells are assigned to the nearest cluster in each subclustering to generate unique pattern of cluster assignments. We then find the cell in the reference dataset with the same or most similar pattern of cluster assignments.

We can examine the cluster assignment patterns for the reference datasets using:

```
metadata(muraro)$scmap_cell_index$subclusters[1:5,1:5]
```

```
##      D28.1_1 D28.1_13 D28.1_15 D28.1_17 D28.1_2
## [1,]      4      42      27      43      10
## [2,]      5       8       2      33      37
## [3,]     11      32      35      17      26
## [4,]      2       4      32       2      18
## [5,]     31      18      21      40       1
```

To project and find the w nearest neighbours we use a similar command as before:

```
muraro_to_seger <- scmapCell(
  projection = muraro,
  index_list = list(
    seger = metadata(segerstolpe)$scmap_cell_index
  ),
  w = 5
)
```

We can again look at the results:

```
muraro_to_seger$seger[[1]][,1:5]
```

```
##      D28.1_1 D28.1_13 D28.1_15 D28.1_17 D28.1_2
## [1,]   2201    1288    1117    1623    1078
## [2,]   1229    1724    2104    1448    1593
## [3,]   1793    1854    2201    2039    1553
## [4,]   1882    1737    1081    1202    1890
## [5,]   1731     976    1903    1834    1437
```

This shows the column number of the 5 nearest neighbours in segerstolpe to each of the cells in muraro. We could then calculate a pseudotime estimate, branch assignment, or other cell-level data by selecting the appropriate data from the colData of the segerstolpe data set. As a demonstration we will find the cell-type of the nearest neighbour of each cell.

```
cell_type_NN <- colData(segerstolpe)$cell_type1[muraro_to_seger$seger[[1]][1,]]
head(cell_type_NN)

## [1] "alpha"      "ductal"      "alpha"      "alpha"      "endothelial"
## [6] "endothelial"
```

### 8.8.5 Metaneighbour

Metaneighbour is specifically designed to ask whether cell-type labels are consistent across datasets. It comes in two versions. First is a fully supervised method which assumes cell-types are known in all datasets and calculates how “good” those cell-type labels are. (The precise meaning of “good” will be described below). Alternatively, metaneighbour can estimate how similar all cell-types are to each other both within and across datasets. We will only be using the unsupervised version as it has much more general applicability and is easier to interpret the results of.

Metaneighbour compares cell-types across datasets by building a cell-cell spearman correlation network. The method then tries to predict the label of each cell through weighted “votes” of its nearest-neighbours. Then scores the overall similarity between two clusters as the AUROC for assigning cells of typeA to typeB based on these weighted votes. AUROC of 1 would indicate all the cells of typeA were assigned to typeB before any other cells were, and an AUROC of 0.5 is what you would get if cells were being randomly assigned.

Metanighbour is just a couple of R functions not a complete package so we have to load them using `source`

```
source("2017-08-28-runMN-US.R")
```

### 8.8.5.1 Prepare Data

Metaneighbour requires all datasets to be combined into a single expression matrix prior to running:

```
is.common <- rowData(muraro)$feature_symbol %in% rowData(segerstolpe)$feature_symbol
muraro <- muraro[is.common,]
segerstolpe <- segerstolpe[match(rowData(muraro)$feature_symbol, rowData(segerstolpe)$feature_symbol),]
rownames(segerstolpe) <- rowData(segerstolpe)$feature_symbol
rownames(muraro) <- rowData(muraro)$feature_symbol
identical(rownames(segerstolpe), rownames(muraro))
```

```
## [1] TRUE
combined_logcounts <- cbind(logcounts(muraro), logcounts(segerstolpe))
dataset_labels <- rep(c("m", "s"), times=c(ncol(muraro), ncol(segerstolpe)))
cell_type_labels <- c(colData(muraro)$cell_type1, colData(segerstolpe)$cell_type1)

pheno <- data.frame(Sample_ID = colnames(combined_logcounts),
                     Study_ID=dataset_labels,
                     Celltype=paste(cell_type_labels, dataset_labels, sep="-"))
rownames(pheno) <- colnames(combined_logcounts)
```

Metaneighbor includes a feature selection method to identify highly variable genes.

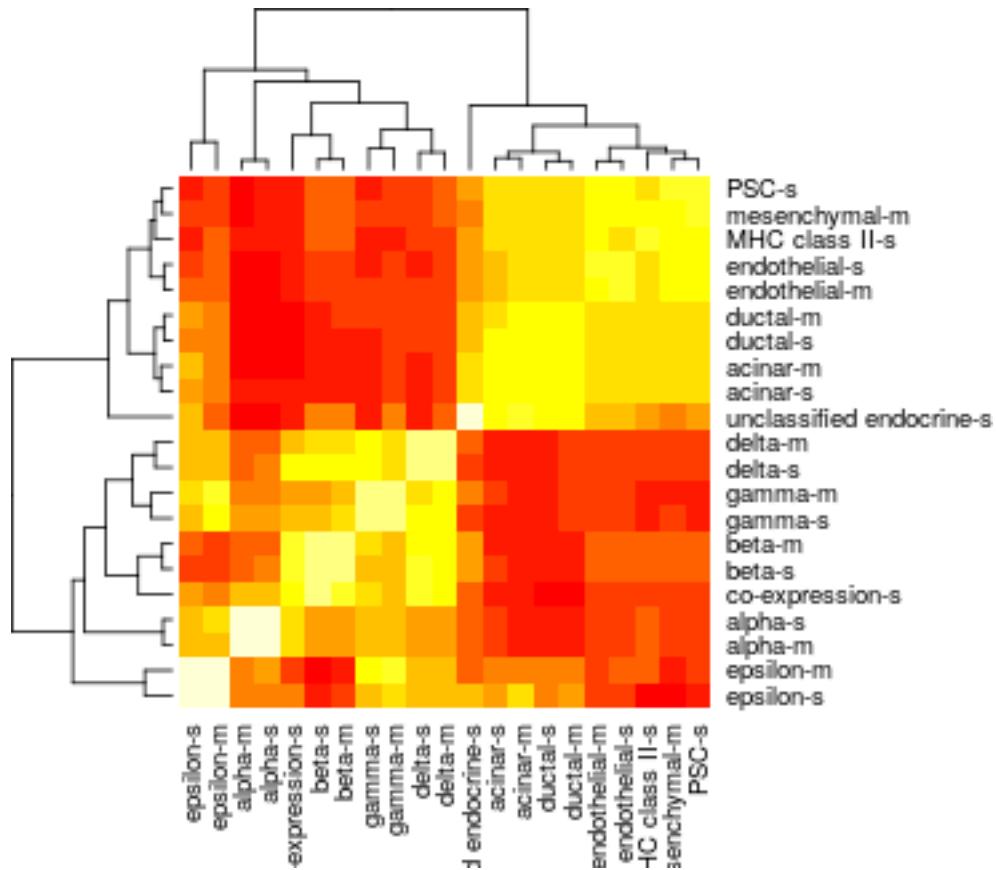
```
var.genes = get_variable_genes(combined_logcounts, pheno)
```

Since Metaneighbor is much slower than `scmap`, we will down sample these datasets.

```
subset <- sample(1:nrow(pheno), 2000)
combined_logcounts <- combined_logcounts[,subset]
pheno <- pheno[subset,]
cell_type_labels <- cell_type_labels[subset]
dataset_labels <- dataset_labels[subset]
```

Now we are ready to run Metaneighbor. First we will run the unsupervised version that will let us see which cell-types are most similar across the two datasets.

```
unsup <- run_MetaNeighbor_US(var.genes, combined_logcounts, unique(pheno$Celltype), pheno)
heatmap(unsup)
```



### 8.8.6 mnnCorrect

mnnCorrect corrects datasets to facilitate joint analysis. In order to account for differences in composition between two replicates or two different experiments it first matches individual cells across experiments to find the overlapping biological structure. Using that overlap it learns which dimensions of expression correspond to the biological state and which dimensions correspond to batch/experiment effect; mnnCorrect assumes these dimensions are orthogonal to each other in high dimensional expression space. Finally it removes the batch/experiment effects from the entire expression matrix to return the corrected matrix.

To match individual cells to each other across datasets, mnnCorrect uses the cosine distance to avoid library-size effect then identifies mutual nearest neighbours ( $k$  determines the neighbourhood size) across datasets. Only overlapping biological groups should have mutual nearest neighbours (see panel b below).

However, this assumes that  $k$  is set to approximately the size of the smallest biological group in the datasets, but a  $k$  that is too low will identify too few mutual nearest-neighbour pairs to get a good estimate of the batch effect we want to remove.

Learning the biological/technical effects is done with either singular value decomposition, similar to RUV we encounter in the batch-correction section, or with principal component analysis with the optimized `irlba` package, which should be faster than SVD. The parameter `svd.dim` specifies how many dimensions should be kept to summarize the biological structure of the data, we will set it to three as we found three major groups using Metaneighbor above. These estimates may be further adjusted by smoothing (`sigma`) and/or variance adjustment (`var.adj`).

mnnCorrect also assumes you've already subset your expression matrices so that they contain identical genes in the same order, fortunately we have already done this for our datasets when we set up our data for Metaneighbor.

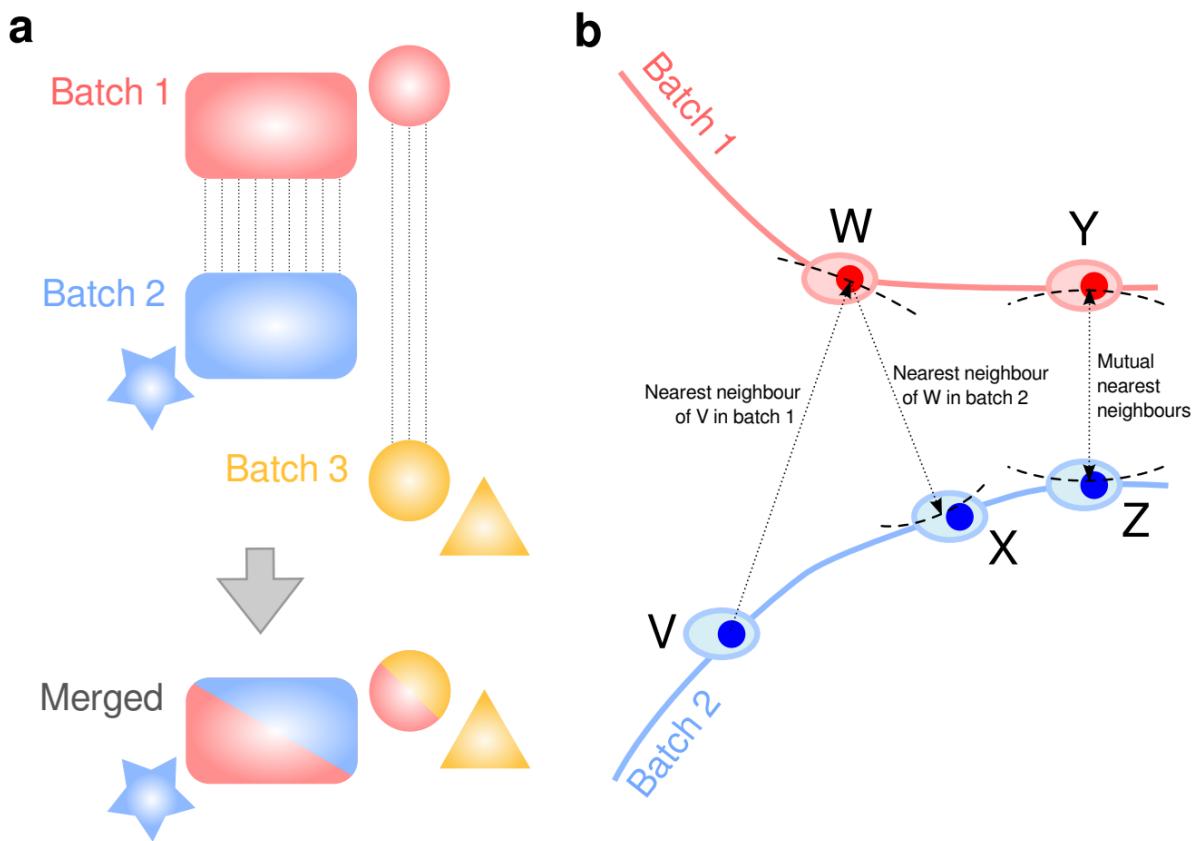


Figure 8.23: mnnCorrect batch/dataset effect correction. From Haghverdi et al. 2017

```

require("scran")

## Loading required package: scran
## Loading required package: BiocParallel
# mnnCorrect will take several minutes to run
corrected <- mnnCorrect(logcounts(muraro), logcounts(segerstolpe), k=20, sigma=1, pc.approx=TRUE, subse

```

First let's check that we found a sufficient number of mnn pairs, mnnCorrect returns a list of dataframe with the mnn pairs for each dataset.

```

dim(corrected$pairs[[1]]) # muraro -> others
## [1] 0 3
dim(corrected$pairs[[2]]) # seger -> others
## [1] 2533    3

```

The first and second columns contain the cell column IDs and the third column contains a number indicating which dataset/batch the column 2 cell belongs to. In our case, we are only comparing two datasets so all the mnn pairs have been assigned to the second table and the third column contains only ones

```

head(corrected$pairs[[2]])

## DataFrame with 6 rows and 3 columns
##   current.cell other.cell other.batch
##   <integer>     <Rle>     <Rle>
## 1      1553        5        1
## 2      1078        5        1
## 3      1437        5        1
## 4      1890        5        1
## 5      1569        5        1
## 6       373        5        1

total_pairs <- nrow(corrected$pairs[[2]])
n_unique_seger <- length(unique((corrected$pairs[[2]][,1])))
n_unique_muraro <- length(unique((corrected$pairs[[2]][,2])))

```

mnnCorrect found 2533 sets of mutual nearest-neighbours between `n_unique_seger` segerstolpe cells and `n_unique_muraro` muraro cells. This should be a sufficient number of pairs but the low number of unique cells in each dataset suggests we might not have captured the full biological signal in each dataset.

**Exercise** Which cell-types had mnns across these datasets? Should we increase/decrease k?

#### Answer

Now we could create a combined dataset to jointly analyse these data. However, the corrected data is no longer counts and usually will contain negative expression values thus some analysis tools may no longer be appropriate. For simplicity let's just plot a joint TSNE.

```

require("Rtsne")

## Loading required package: Rtsne
joint_expression_matrix <- cbind(corrected$corrected[[1]], corrected$corrected[[2]])

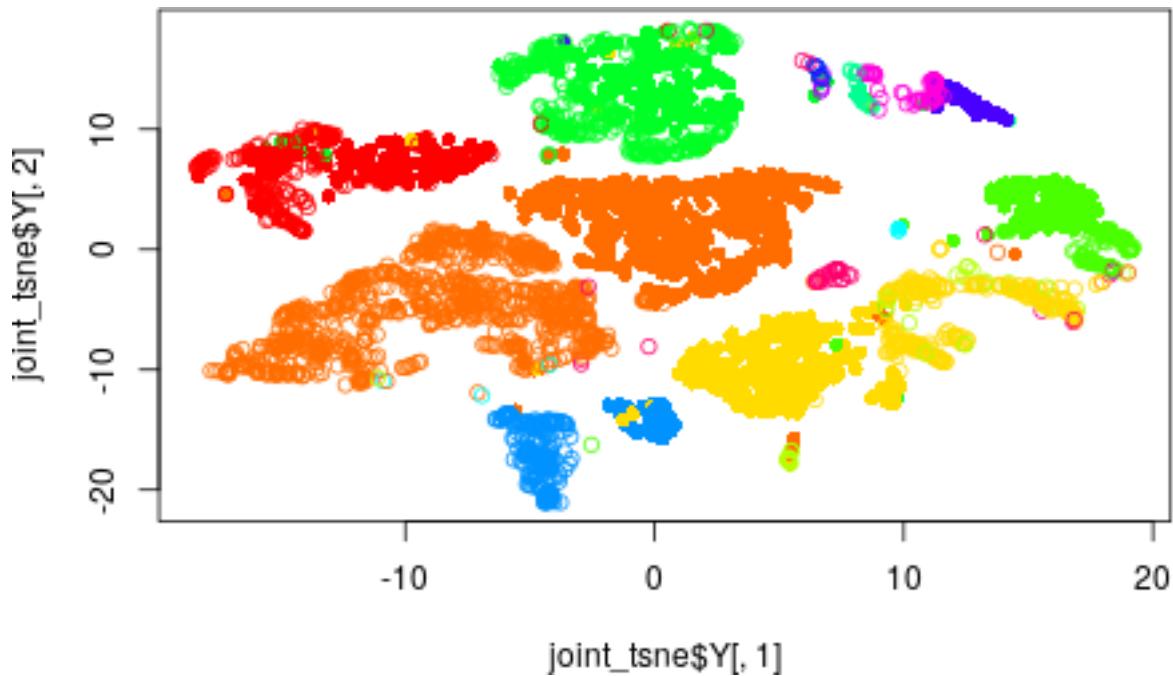
# Tsne will take some time to run on the full dataset
joint_tsne <- Rtsne(t(joint_expression_matrix[rownames(joint_expression_matrix) %in% var.genes,]), init

```

```

check_duplicates=FALSE, max_iter=200, stop_lying_iter=50, mom_switch_iter=50)
dataset_labels <- factor(rep(c("m", "s"), times=c(ncol(muraro), ncol(segerstolpe))))
cell_type_labels <- factor(c(colData(muraro)$cell_type1, colData(segerstolpe)$cell_type1))
plot(joint_tsne$Y[,1], joint_tsne$Y[,2], pch=c(16,1)[dataset_labels], col=rainbow(length(levels(cell_ty

```



### 8.8.7 Canonical Correlation Analysis (Seurat)

The Seurat package contains another correction method for combining multiple datasets, called CCA. However, unlike mnnCorrect it doesn't correct the expression matrix itself directly. Instead Seurat finds a lower dimensional subspace for each dataset then corrects these subspaces. Also different from mnnCorrect, Seurat only combines a single pair of datasets at a time.

Seurat uses gene-gene correlations to identify the biological structure in the dataset with a method called canonical correlation analysis (CCA). Seurat learns the shared structure to the gene-gene correlations and then evaluates how well each cell fits this structure. Cells which must better described by a data-specific dimensionality reduction method than by the shared correlation structure are assumed to represent dataset-specific cell-types/states and are discarded before aligning the two datasets. Finally the two datasets are aligned using ‘warping’ algorithms which normalize the low-dimensional representations of each dataset in a way that is robust to differences in population density.

Note because Seurat uses up a lot of library space you will have to restart your R-session to load it, and the plots/\_seuratput won't be automatically generated on this page.

Reload the data:

```

muraro <- readRDS("pancreas/muraro.rds")
segerstolpe <- readRDS("pancreas/segerstolpe.rds")
segerstolpe <- segerstolpe[, colData(segerstolpe)$cell_type1 != "unclassified"]
segerstolpe <- segerstolpe[, colData(segerstolpe)$cell_type1 != "not applicable",]
muraro <- muraro[, colData(muraro)$cell_type1 != "unclear"]
is.common <- rowData(muraro)$feature_symbol %in% rowData(segerstolpe)$feature_symbol
muraro <- muraro[is.common,]
segerstolpe <- segerstolpe[match(rowData(muraro)$feature_symbol, rowData(segerstolpe)$feature_symbol),]

```

```
rownames(segerstolpe) <- rowData(segerstolpe)$feature_symbol
rownames(muraro) <- rowData(muraro)$feature_symbol
identical(rownames(segerstolpe), rownames(muraro))
```

First we will reformat our data into Seurat objects:

```
require("Seurat")
set.seed(4719364)
muraro_seurat <- CreateSeuratObject(raw.data=assays(muraro)[["normcounts"]]) # raw counts aren't available in the package
muraro_seurat@meta.data[, "dataset"] <- 1
muraro_seurat@meta.data[, "celltype"] <- paste("m", colData(muraro)$cell_type1, sep="-")

seger_seurat <- CreateSeuratObject(raw.data=assays(segerstolpe)[["counts"]])
seger_seurat@meta.data[, "dataset"] <- 2
seger_seurat@meta.data[, "celltype"] <- paste("s", colData(segerstolpe)$cell_type1, sep="-")
```

Next we must normalize, scale and identify highly variable genes for each dataset:

```
muraro_seurat <- NormalizeData(object=muraro_seurat)
muraro_seurat <- ScaleData(object=muraro_seurat)
muraro_seurat <- FindVariableGenes(object=muraro_seurat, do.plot=TRUE)

seger_seurat <- NormalizeData(object=seger_seurat)
seger_seurat <- ScaleData(object=seger_seurat)
seger_seurat <- FindVariableGenes(object=seger_seurat, do.plot=TRUE)
```

Eventhough Seurat corrects for the relationship between dispersion and mean expression, it doesn't use the corrected value when ranking features. Compare the results of the command below with the results in the plots above:

```
head(muraro_seurat@hvg.info, 50)
head(seger_seurat@hvg.info, 50)
```

But we will follow their example and use the top 2000 most dispersed genes with merged\_seurat correcting for mean expression from each dataset anyway.

```
gene.use <- union(rownames(x = head(x = muraro_seurat@hvg.info, n = 2000)),
rownames(x = head(x = seger_seurat@hvg.info, n = 2000)))
```

**Exercise** Find the features we would use if we selected the top 2000 most dispersed after scaling by mean.  
(Hint: consider the `order` function)

### Answer

Now we will run CCA to find the shared correlation structure for these two datasets:

Note to speed up the calculations we will be using only the top 5 dimensions but ideally you would consider many more and then select the top most informative ones using `DimHeatmap`.

```
merged_seurat <- RunCCA(object=muraro_seurat, object2=seger_seurat, genes.use=genes.use, add.cell.id1="m",
DimPlot(object = merged_seurat, reduction.use = "cca", group.by = "dataset", pt.size = 0.5) # Before combining datasets
```

To identify dataset specific cell-types we compare how well cells are ‘explained’ by CCA vs dataset-specific principal component analysis.

```
merged_seurat <- CalcVarExpRatio(object = merged_seurat, reduction.type = "pca", grouping.var = "dataset")
merged.all <- merged_seurat
merged_seurat <- SubsetData(object=merged_seurat, subset.name="var.ratio.pca", accept.low = 0.5) # CCA vs PCA
merged.discard <- SubsetData(object=merged.all, subset.name="var.ratio.pca", accept.high = 0.5)
```

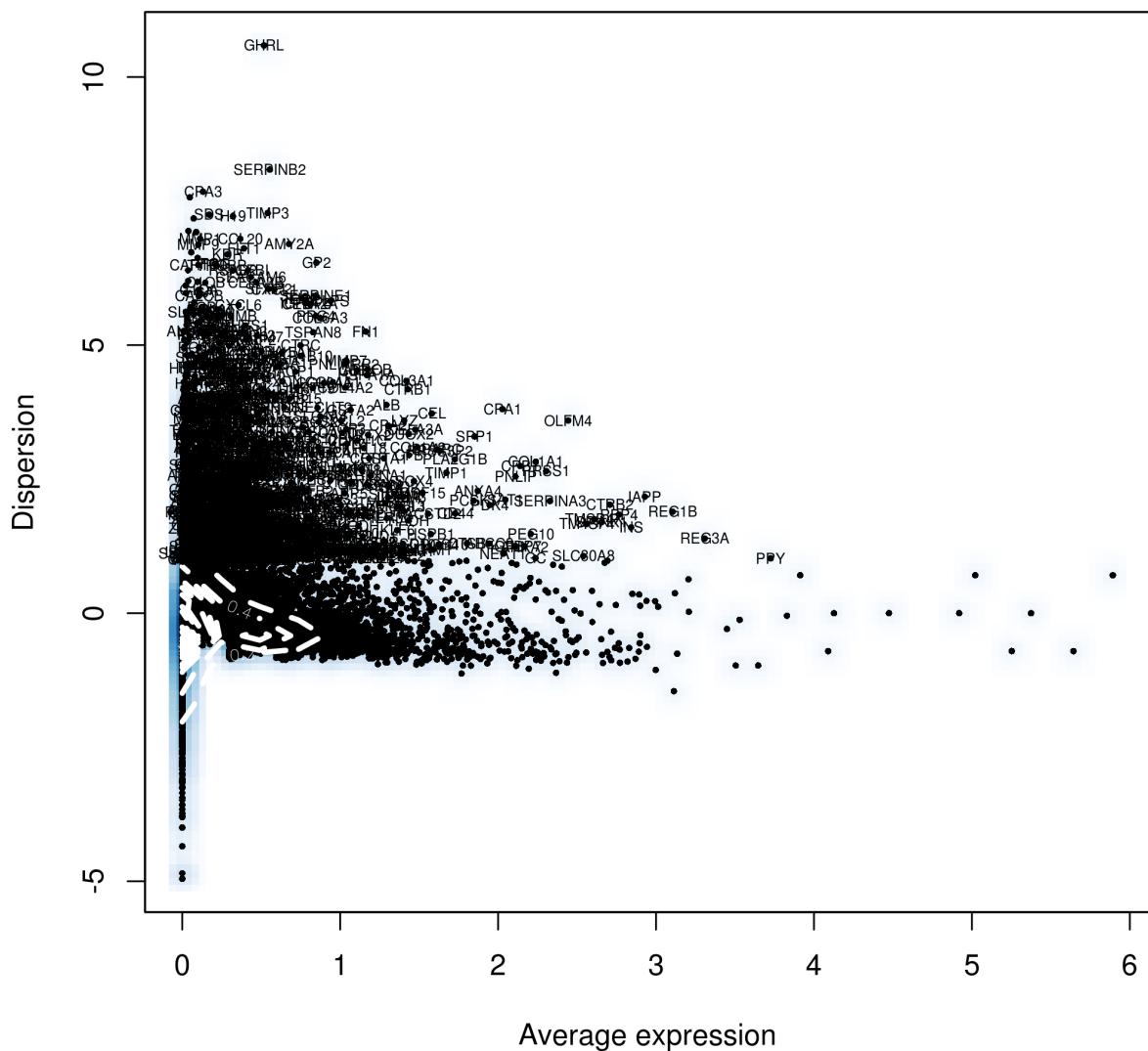


Figure 8.24: muraro variable genes

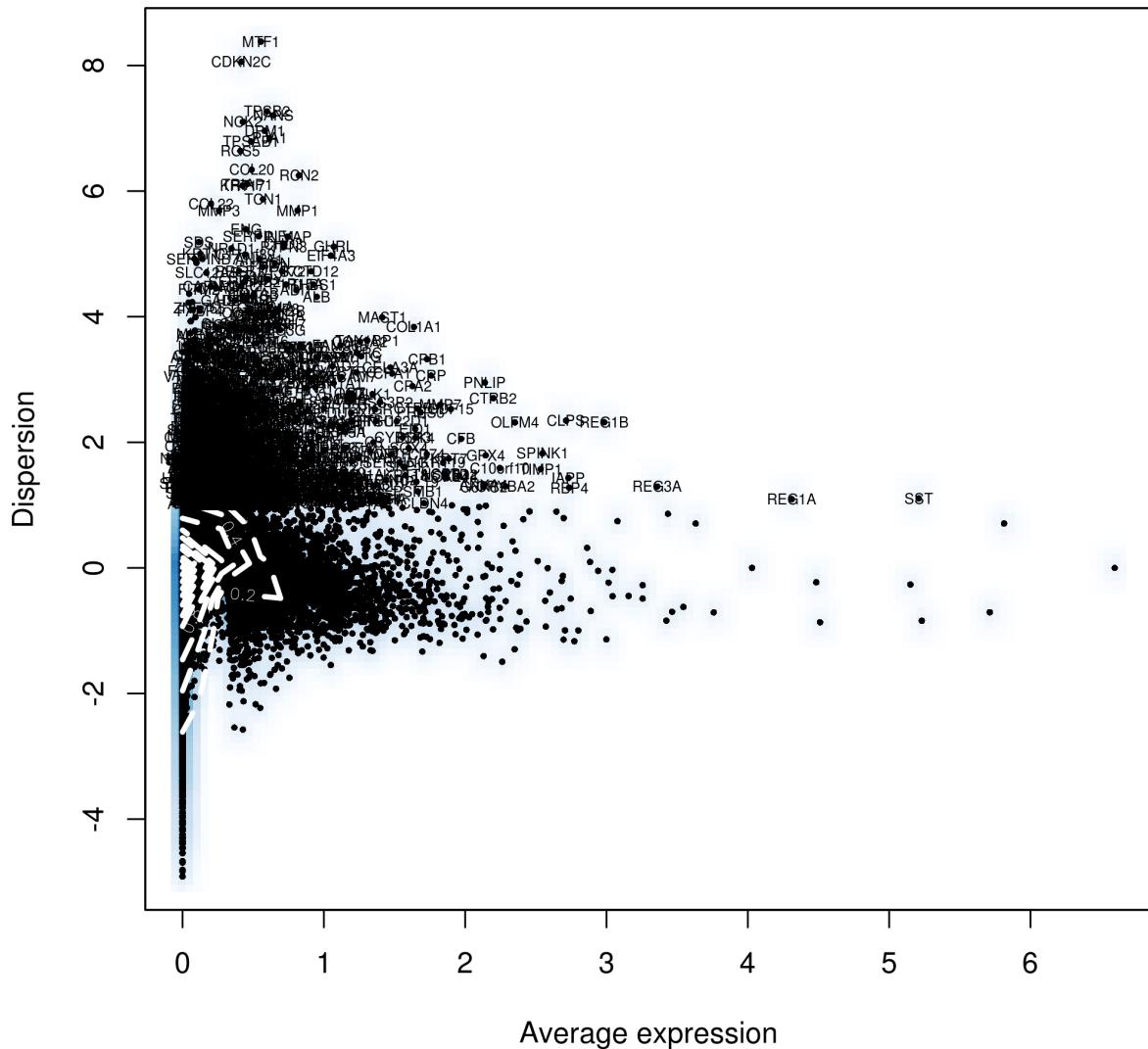


Figure 8.25: segerstolpe variable genes

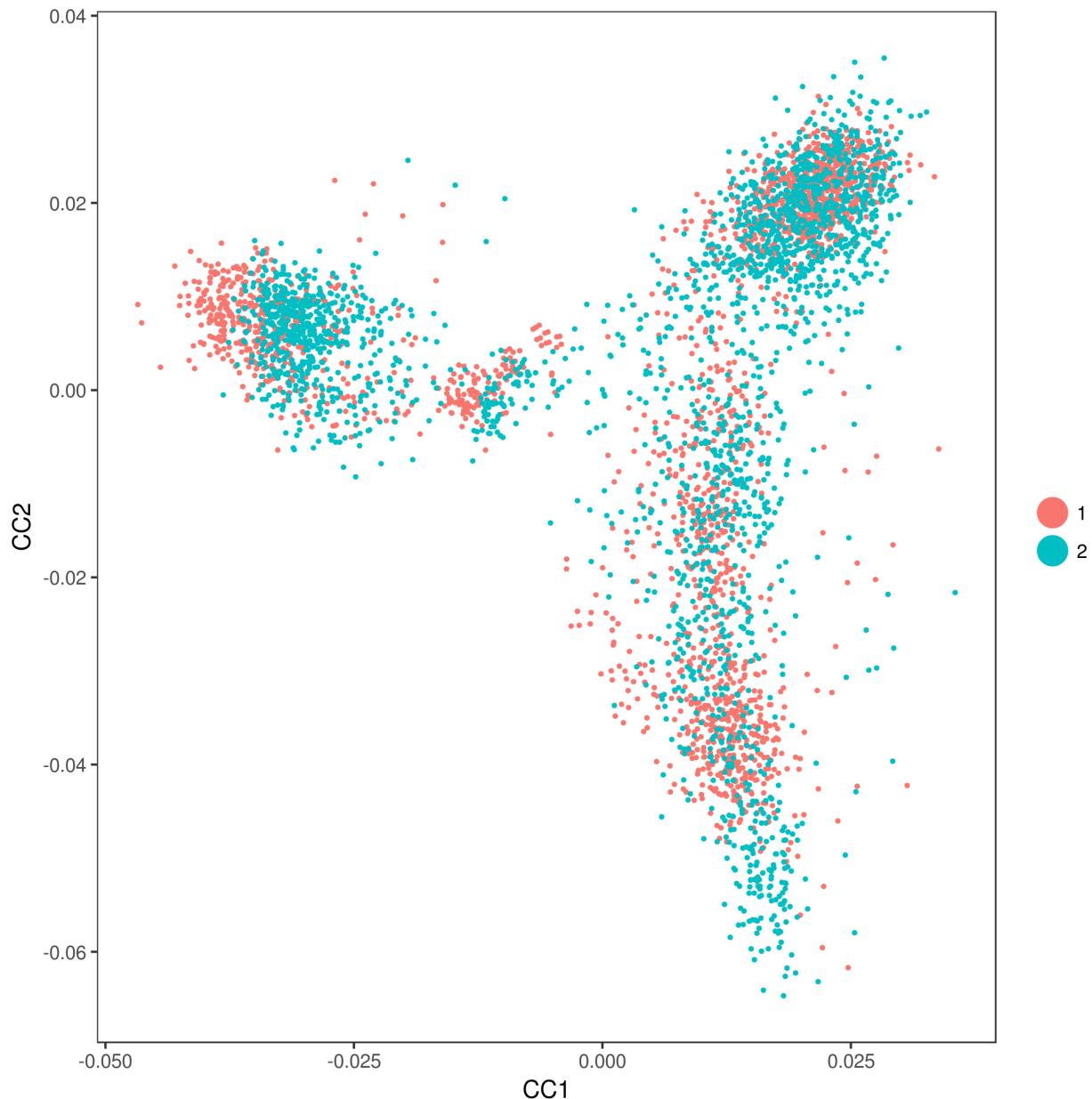


Figure 8.26: Before Aligning

```
summary(factor(merged.discard@meta.data$celltype)) # check the cell-type of the discarded cells.
```

Here we can see that despite both datasets containing endothelial cells, almost all of them have been discarded as “dataset-specific”. Now we can align the datasets:

```
merged_seurat <- AlignSubspace(object = merged_seurat, reduction.type = "cca", grouping.var = "dataset")
DimPlot(object = merged_seurat, reduction.use = "cca.aligned", group.by = "dataset", pt.size = 0.5) # A
```

**Exercise** Compare the results for if you use the features after scaling dispersions.

#### Answer

**Advanced Exercise** Use the clustering methods we previously covered on the combined datasets. Do you identify any novel cell-types?

### 8.8.8 sessionInfo()

```
## R version 3.4.3 (2017-11-30)
## Platform: x86_64-pc-linux-gnu (64-bit)
## Running under: Debian GNU/Linux 9 (stretch)
##
## Matrix products: default
## BLAS: /usr/lib/openblas-base/libblas.so.3
## LAPACK: /usr/lib/libopenblas-r0.2.19.so
##
## locale:
## [1] LC_CTYPE=en_US.UTF-8          LC_NUMERIC=C
## [3] LC_TIME=en_US.UTF-8          LC_COLLATE=en_US.UTF-8
## [5] LC_MONETARY=en_US.UTF-8       LC_MESSAGES=C
## [7] LC_PAPER=en_US.UTF-8          LC_NAME=C
## [9] LC_ADDRESS=C                  LC_TELEPHONE=C
## [11] LC_MEASUREMENT=en_US.UTF-8    LC_IDENTIFICATION=C
##
## attached base packages:
## [1] stats4     parallel   methods   stats      graphics  grDevices utils
## [8] datasets   base
##
## other attached packages:
## [1] Rtsne_0.13            scran_1.6.7
## [3] BiocParallel_1.12.0    bindrcpp_0.2
## [5] scmap_1.1.5           scater_1.6.2
## [7] SingleCellExperiment_1.0.0 SummarizedExperiment_1.8.1
## [9] DelayedArray_0.4.1     matrixStats_0.53.0
## [11] GenomicRanges_1.30.1    GenomeInfoDb_1.14.0
## [13] IRanges_2.12.0          S4Vectors_0.16.0
## [15] ggplot2_2.2.1           Biobase_2.38.0
## [17] BiocGenerics_0.24.0     googleVis_0.6.2
## [19] knitr_1.19
##
## loaded via a namespace (and not attached):
## [1] bitops_1.0-6             bit64_0.9-7           progress_1.1.2
## [4] httr_1.3.1                rprojroot_1.3-2        dynamicTreeCut_1.63-1
## [7] tools_3.4.3               backports_1.1.2        irlba_2.3.2
## [10] DT_0.4                   R6_2.2.2              viper_0.4.5
```

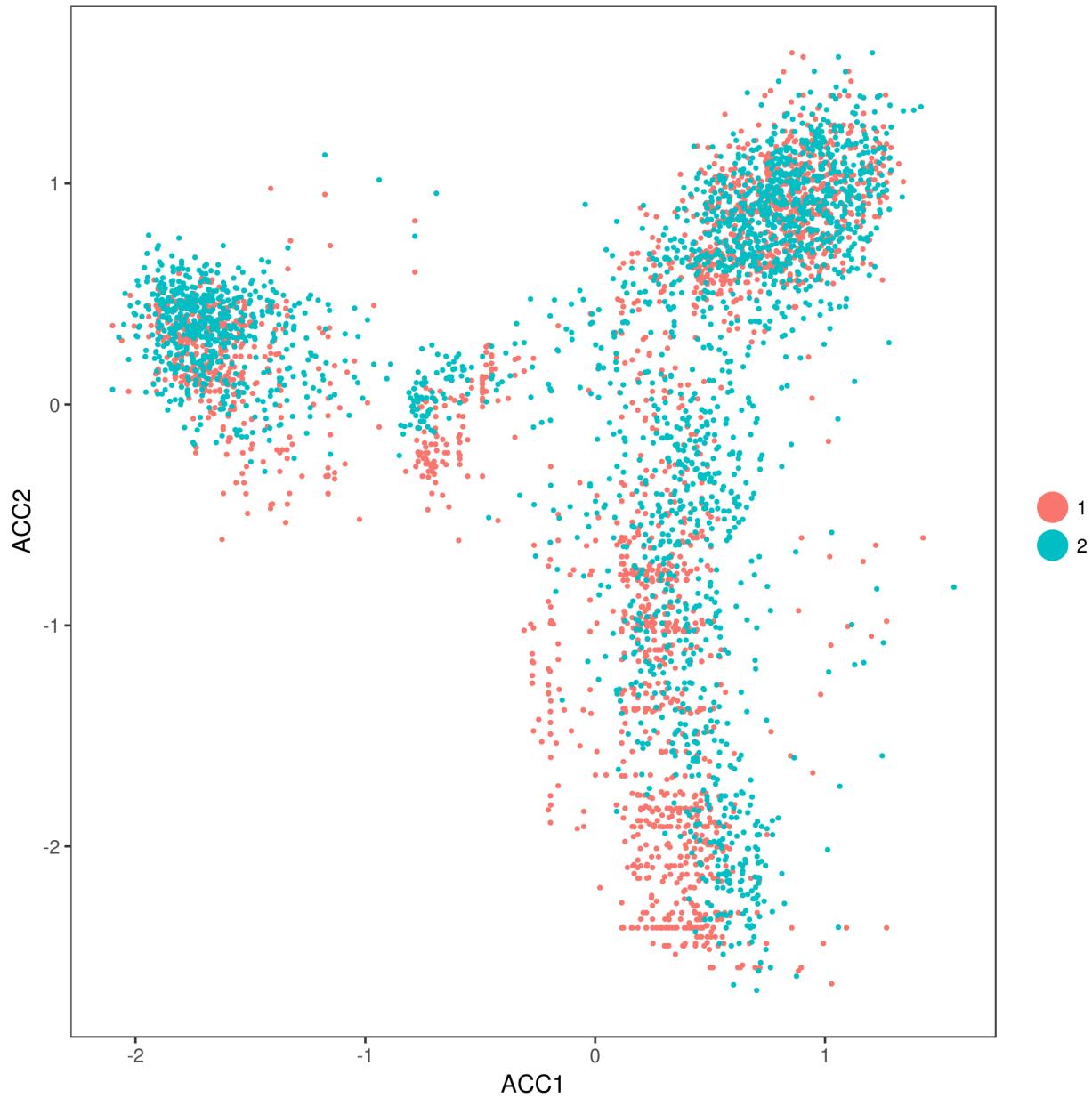


Figure 8.27: After Aligning

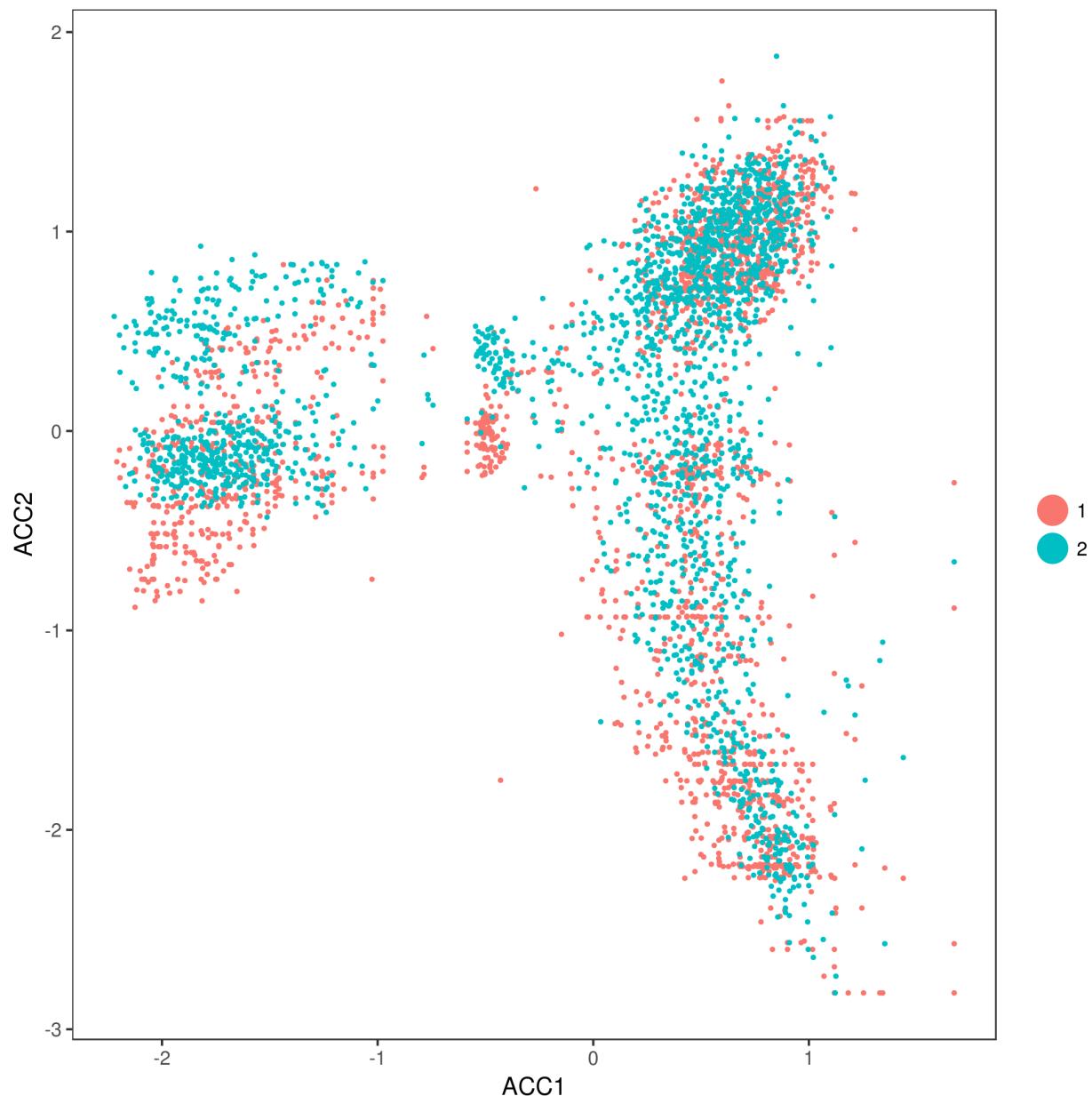


Figure 8.28: After Aligning

```

## [13] DBI_0.7           lazyeval_0.2.1      colorspace_1.3-2
## [16] gridExtra_2.3      prettyunits_1.0.2    bit_1.1-12
## [19] compiler_3.4.3     labeling_0.3       bookdown_0.6
## [22] scales_0.5.0      randomForest_4.6-12 proxy_0.4-21
## [25] stringr_1.2.0      digest_0.6.15     rmarkdown_1.8
## [28] XVector_0.18.0     pkgconfig_2.0.1    htmltools_0.3.6
## [31] limma_3.34.8       htmlwidgets_1.0   rlang_0.1.6
## [34] RSQLite_2.0         FNN_1.1          shiny_1.0.5
## [37] bindr_0.1           zoo_1.8-1        jsonlite_1.5
## [40] dplyr_0.7.4        RCurl_1.95-4.10 magrittr_1.5
## [43] GenomeInfoDbData_1.0.0 Matrix_1.2-7.1  Rcpp_0.12.15
## [46] ggbeeswarm_0.6.0    munsell_0.4.3    viridis_0.5.0
## [49] stringi_1.1.6      yaml_2.1.16      edgeR_3.20.8
## [52] zlibbioc_1.24.0     rhdf5_2.22.0    plyr_1.8.4
## [55] grid_3.4.3          blob_1.1.0      shinydashboard_0.6.1
## [58] lattice_0.20-34     locfit_1.5-9.1  pillar_1.1.0
## [61] igraph_1.1.2        rjson_0.2.15    reshape2_1.4.3
## [64] codetools_0.2-15   biomaRt_2.34.2  XML_3.98-1.9
## [67] glue_1.2.0          evaluate_0.10.1 data.table_1.10.4-3
## [70] httpuv_1.3.5       gtable_0.2.0    assertthat_0.2.0
## [73] xfun_0.1            mime_0.5        xtable_1.8-2
## [76] e1071_1.6-8         class_7.3-14   viridisLite_0.3.0
## [79] tibble_1.4.2        AnnotationDbi_1.40.0 beeswarm_0.2.3
## [82] memoise_1.1.0       tximport_1.6.0   statmod_1.4.30

```

## 8.9 Search scRNA-Seq data

```

library(scfind)
library(SingleCellExperiment)
set.seed(1234567)

```

### 8.9.1 About

`scfind` is a tool that allows one to search single cell RNA-Seq collections (Atlas) using lists of genes, e.g. searching for cells and cell-types where a specific set of genes are expressed. `scfind` is a Bioconductor package. Cloud implementation of `scfind` with a large collection of datasets is available on our website.

### 8.9.2 Dataset

We will run `scfind` on the same human pancreas dataset as in the previous chapter. `scfind` also operates on `SingleCellExperiment` class:

```

muraro <- readRDS("pancreas/muraro.rds")

```

### 8.9.3 Gene Index

Now we need to create a gene index using our dataset:

```

cellIndex <- buildCellIndex(muraro)

```

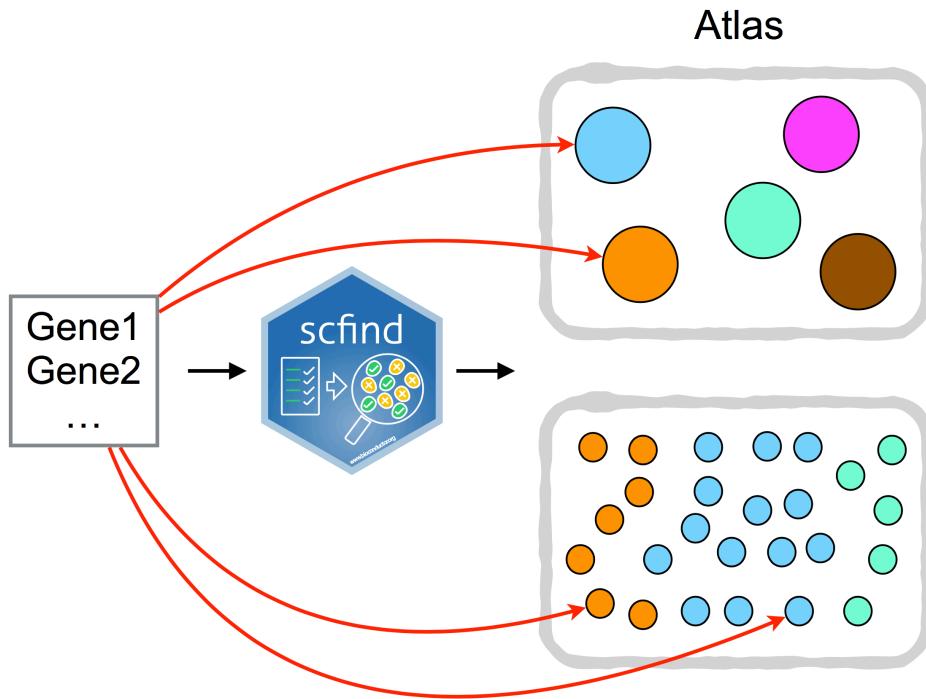


Figure 8.29: scfind can be used to search large collection of scRNA-seq data by a list of gene IDs.

The gene index contains for each gene indexes of the cells where it is expressed. This is similar to sparsification of the expression matrix. In addition to this the index is also compressed in a way that it can accessed very quickly. We estimated that one can achieve 5-10 compression factor with this method.

By default the `cell_type1` column of the `colData` slot of the `SingleCellExperiment` object is used to define cell types, however it can also be defined manually using the `cell_type_column` argument of the `buildCellTypeIndex` function (check `?buildCellTypeIndex`).

#### 8.9.4 Marker genes

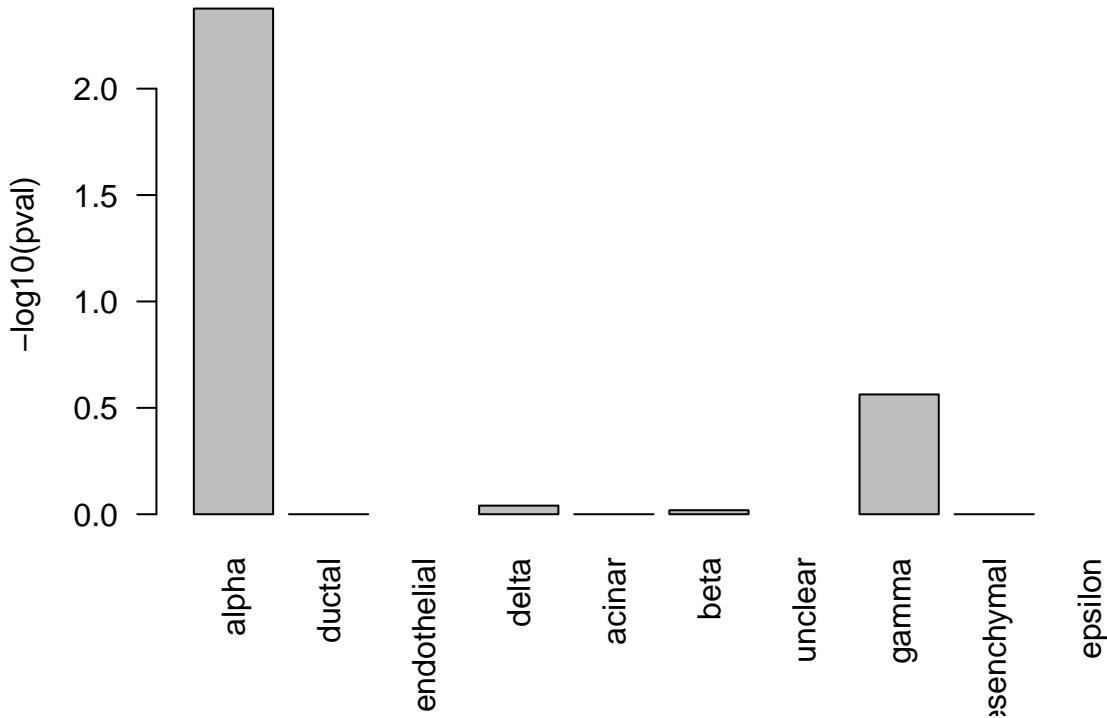
Now let's define lists of cell type specific marker genes. We will use the marker genes identified in the original publication, namely in Figure 1:

```
# these genes are taken from fig. 1
muraro_alpha <- c("GCG", "LOXL4", "PLCE1", "IRX2", "GC", "KLHL41",
                  "CRYBA2", "TTR", "TM4SF4", "RGS4")
muraro_beta <- c("INS", "IAPP", "MAFA", "NPTX2", "DLK1", "ADCYAP1",
                  "PFKFB2", "PDX1", "TGFBR3", "SYT13")
muraro_gamma <- c("PPY", "SERTM1", "CARTPT", "SLITRK6", "ETV1",
                  "THSD7A", "AQP3", "ENTPD2", "PTGFR", "CHN2")
muraro_delta <- c("SST", "PRG4", "LEPR", "RBP4", "BCHE", "HHEX",
                  "FRZB", "PCSK1", "RGS2", "GABRG2")
```

### 8.9.5 Search cells by a gene list

`findCell` function returns a list of p-values corresponding to all cell types in a given dataset. It also outputs a list of cells in which genes from the given gene list are co-expressed. We will run it on all lists of marker genes defined above:

```
res <- findCell(cellIndex, muraro_alpha)
barplot(-log10(res$p_values), ylab = "-log10(pval)", las = 2)
```

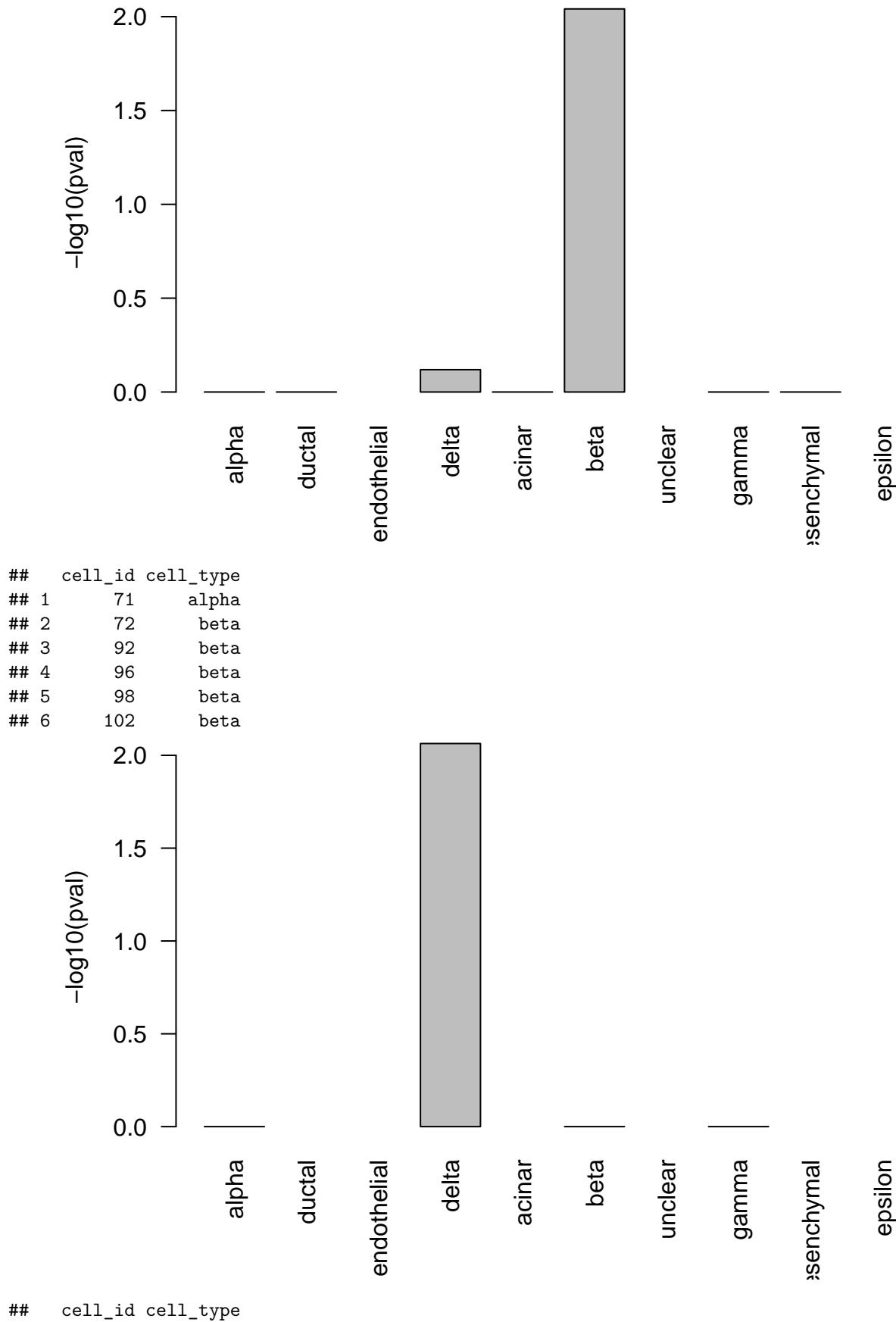


```
head(res$common_exprs_cells)
```

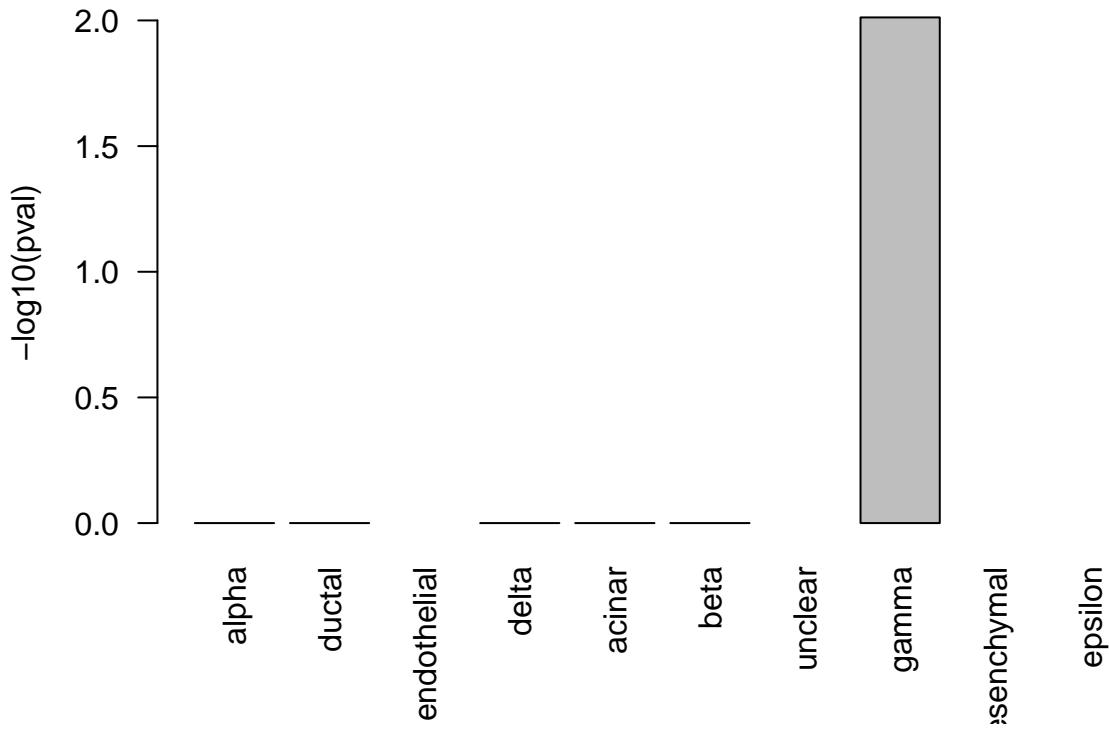
```
##   cell_id cell_type
## 1       1    alpha
## 2       3    alpha
## 3       7    alpha
## 4       9    alpha
## 5      15    alpha
## 6      20    alpha
```

### Exercise 1

Perform a search by *beta*, *delta* and *gamma* gene lists and explore the results.



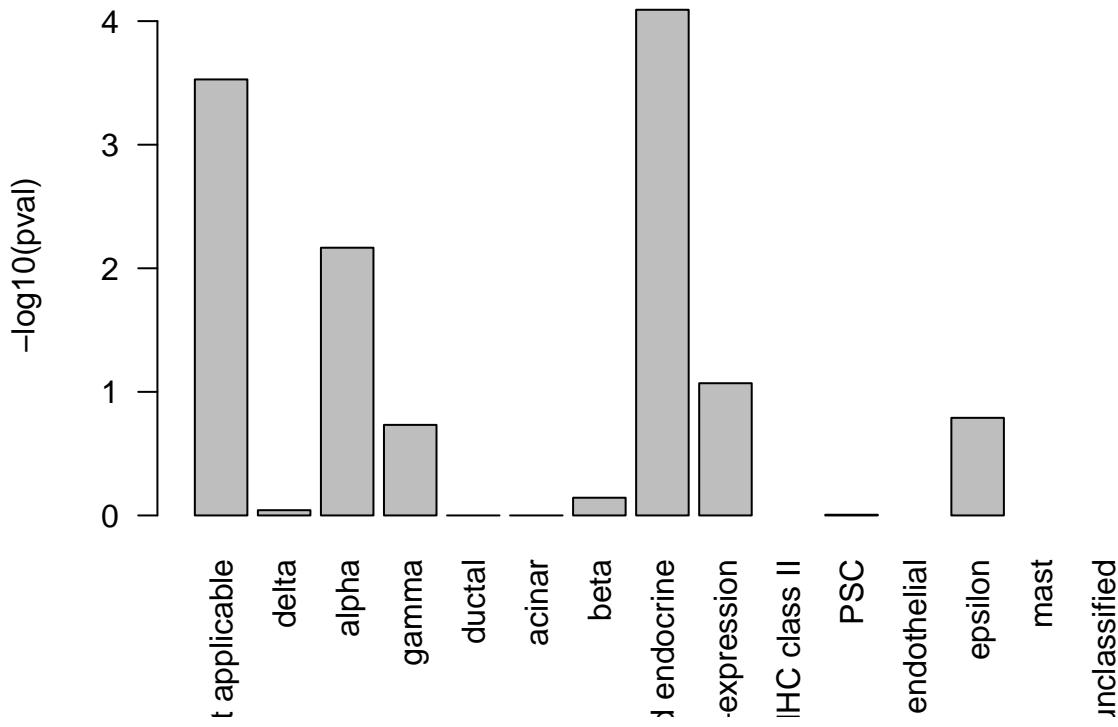
```
## 1      40    delta
## 2     212    delta
## 3     225    delta
## 4     253    delta
## 5     330    delta
## 6     400    delta
```



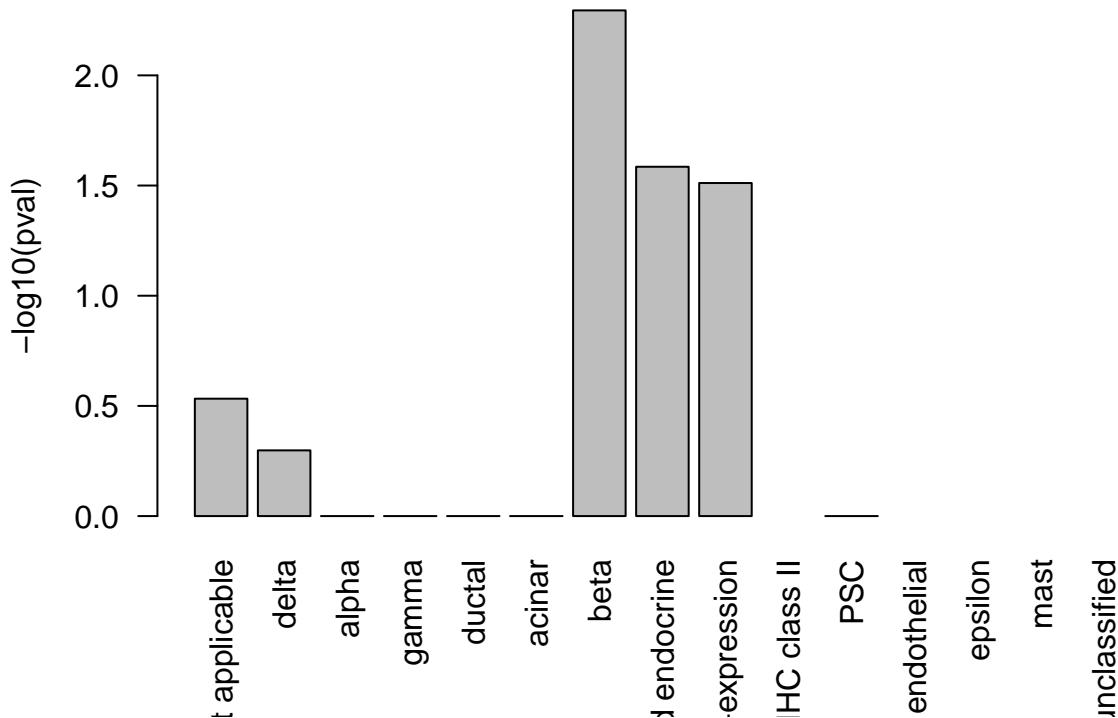
```
##   cell_id cell_type
## 1      53    alpha
## 2     102    beta
## 3     255    gamma
## 4     305    gamma
## 5     525    gamma
## 6     662    gamma
```

## Exercise 2

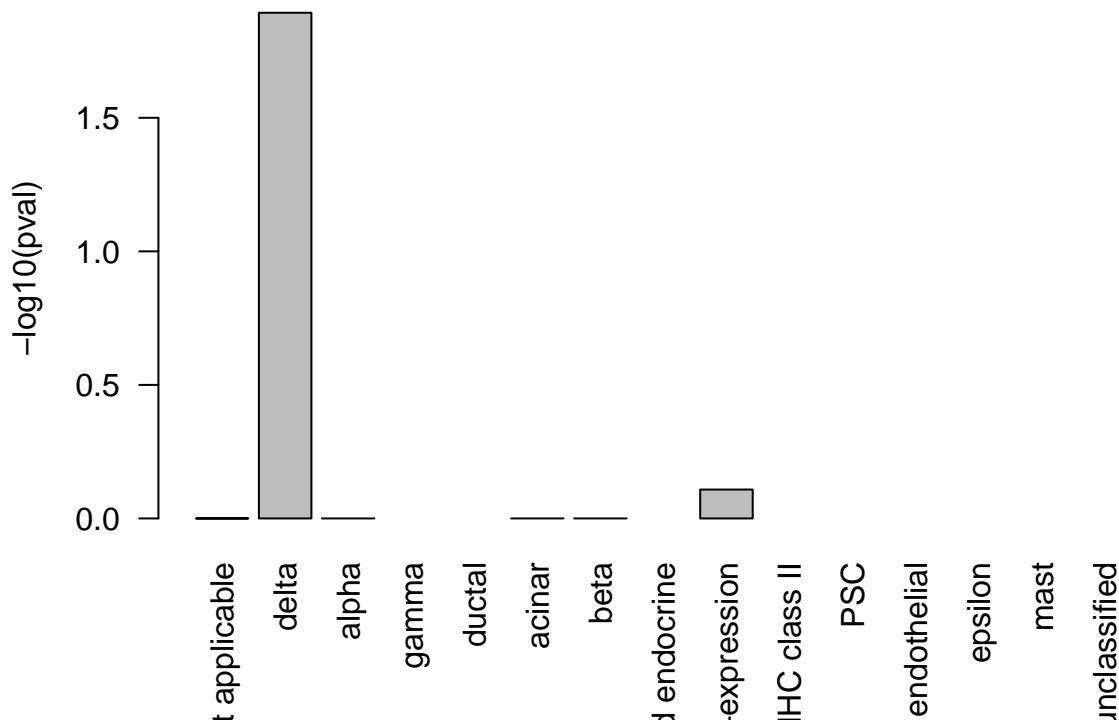
Load the `segerstolpe` and search it using *alpha*, *beta*, *delta* and *gamma* gene lists identified in `muraro` dataset.



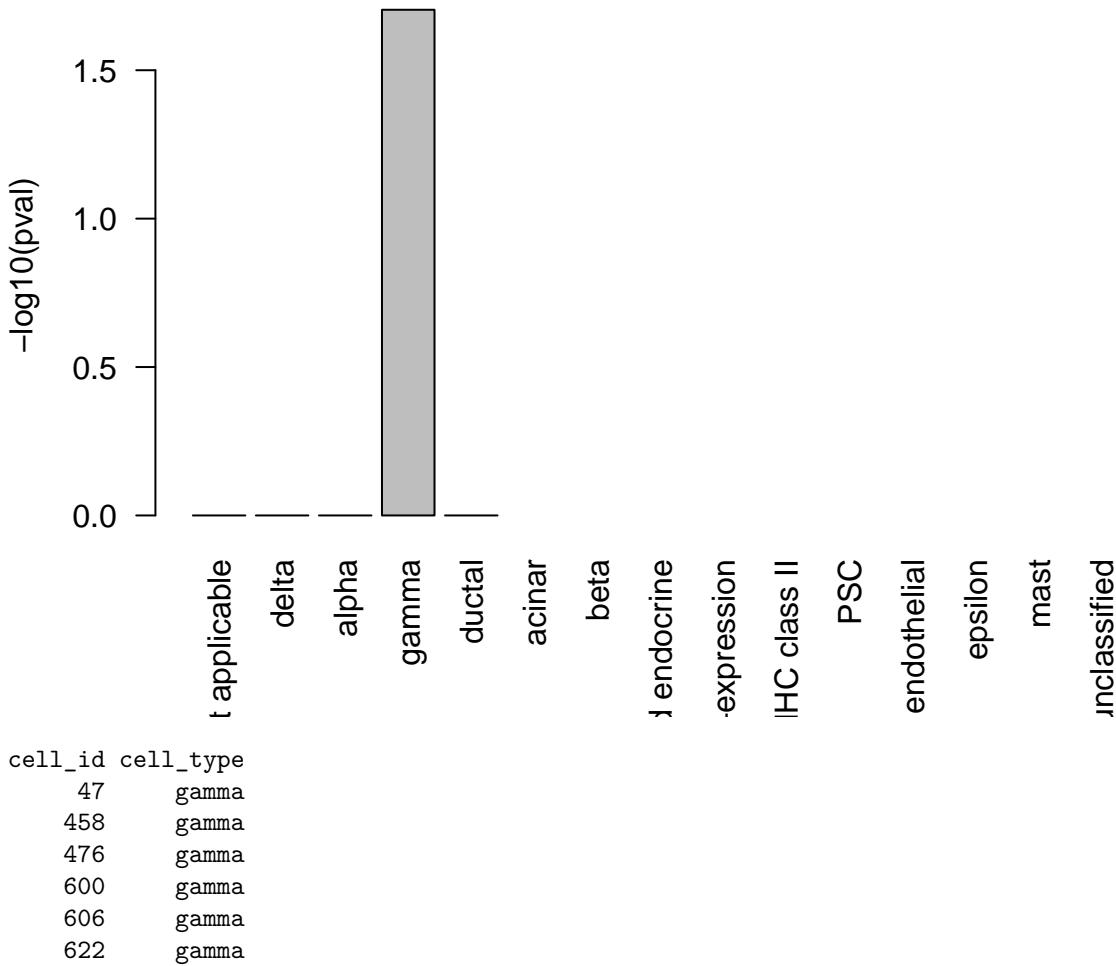
```
##   cell_id cell_type
## 1      18    alpha
## 2      20    alpha
## 3      24    alpha
## 4      32    alpha
## 5      43    alpha
## 6      48    alpha
```



```
##   cell_id      cell_type
## 1      15 co-expression
## 2      58      beta
## 3     300      beta
## 4     390 co-expression
## 5     504 co-expression
## 6     506      beta
```



```
##   cell_id      cell_type
## 1     170      delta
## 2     715      delta
## 3    1039 co-expression
## 4    1133      delta
## 5    1719      delta
## 6    1721      delta
```



### 8.9.6 sessionInfo()

```

sessionInfo()

## R version 3.4.3 (2017-11-30)
## Platform: x86_64-pc-linux-gnu (64-bit)
## Running under: Debian GNU/Linux 9 (stretch)
##
## Matrix products: default
## BLAS: /usr/lib/openblas-base/libblas.so.3
## LAPACK: /usr/lib/libopenblas-r0.2.19.so
##
## locale:
## [1] LC_CTYPE=en_US.UTF-8          LC_NUMERIC=C
## [3] LC_TIME=en_US.UTF-8          LC_COLLATE=en_US.UTF-8
## [5] LC_MONETARY=en_US.UTF-8       LC_MESSAGES=C
## [7] LC_PAPER=en_US.UTF-8          LC_NAME=C
## [9] LC_ADDRESS=C                  LC_TELEPHONE=C
## [11] LC_MEASUREMENT=en_US.UTF-8   LC_IDENTIFICATION=C
##
## attached base packages:
## [1] parallel stats4   methods   stats      graphics grDevices utils

```

```
## [8] datasets  base
##
## other attached packages:
## [1] SingleCellExperiment_1.0.0 SummarizedExperiment_1.8.1
## [3] DelayedArray_0.4.1      matrixStats_0.53.0
## [5] Biobase_2.38.0          GenomicRanges_1.30.1
## [7] GenomeInfoDb_1.14.0    IRanges_2.12.0
## [9] S4Vectors_0.16.0        BiocGenerics_0.24.0
## [11] scfind_1.0.0            knitr_1.19
##
## loaded via a namespace (and not attached):
## [1] Rcpp_0.12.15           plyr_1.8.4          pillar_1.1.0
## [4] compiler_3.4.3         XVector_0.18.0     bindr_0.1
## [7] bitops_1.0-6           tools_3.4.3         zlibbioc_1.24.0
## [10] digest_0.6.15          bit_1.1-12         tibble_1.4.2
## [13] evaluate_0.10.1       lattice_0.20-34    pkgconfig_2.0.1
## [16] rlang_0.1.6            Matrix_1.2-7.1     yaml_2.1.16
## [19] xfun_0.1               bindrcpp_0.2       GenomeInfoDbData_1.0.0
## [22] stringr_1.2.0          dplyr_0.7.4        rprojroot_1.3-2
## [25] grid_3.4.3             glue_1.2.0          R6_2.2.2
## [28] hash_2.2.6              rmarkdown_1.8       bookdown_0.6
## [31] reshape2_1.4.3          magrittr_1.5       backports_1.1.2
## [34] htmltools_0.3.6         assertthat_0.2.0    stringi_1.1.6
## [37] RCurl_1.95-4.10
set.seed(1234567)
```

# Chapter 9

## Seurat

Seurat was originally developed as a clustering tool for scRNA-seq data, however in the last few years the focus of the package has become less specific and at the moment **Seurat** is a popular R package that can perform QC, analysis, and exploration of scRNA-seq data, i.e. many of the tasks covered in this course. Although the authors provide several tutorials, here we provide a brief overview by following an example created by the authors of **Seurat** (2,800 Peripheral Blood Mononuclear Cells). We mostly use default values in various function calls, for more details please consult the documentation and the authors. For course purpose will use a small **Deng** dataset described in the previous chapters:

```
deng <- readRDS("deng/deng-reads.rds")
```

**Note** Thanks to *community detection* approach used in **Seurat** clustering, it allows one to work on datasets containing up to  $10^5$  cells. We recommend using **Seurat** for datasets with more than 5000 cells. For smaller dataset a good alternative will be **SC3**.

### 9.1 Seurat object class

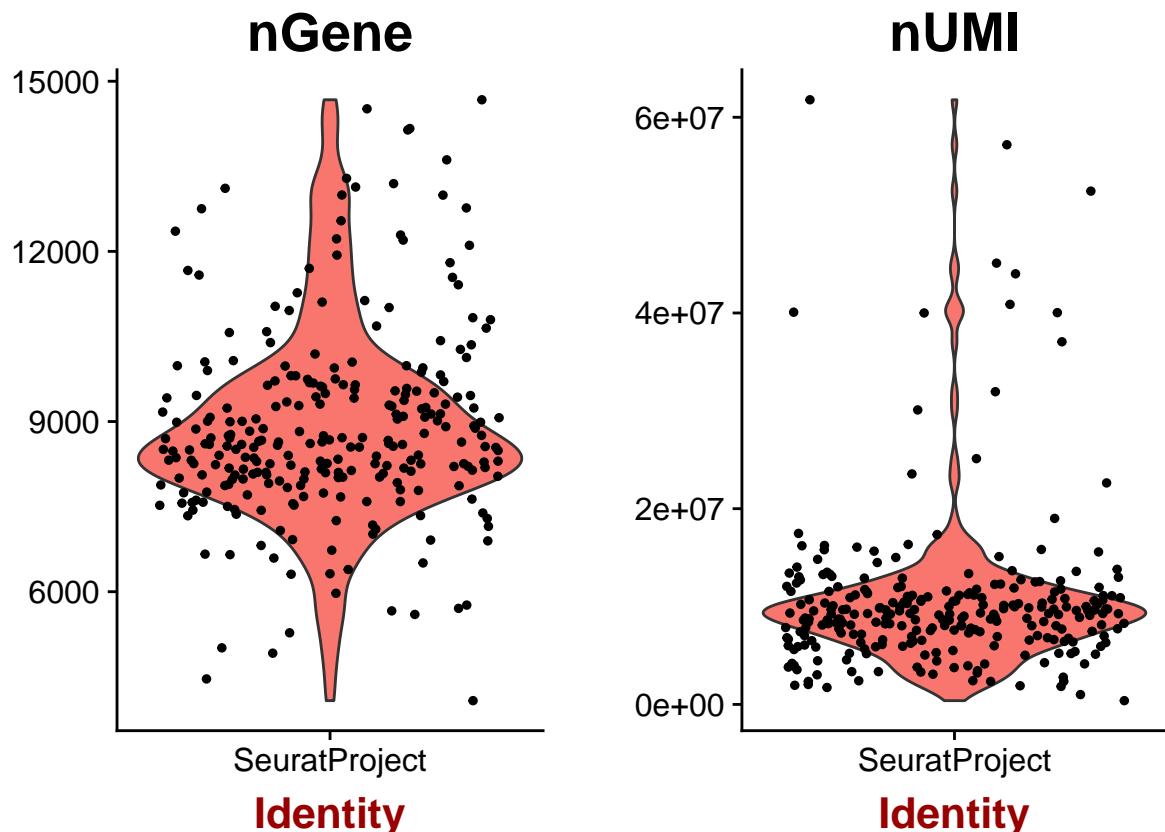
**Seurat** does not integrate **SingleCellExperiment** Bioconductor class described above, but instead introduces its own object class - **seurat**. All calculations in this chapter are performed on an object of this class. To begin the analysis we first need to initialize the object with the raw (non-normalized) data. We will keep all genes expressed in  $\geq 3$  cells and all cells with at least 200 detected genes:

```
library(SingleCellExperiment)
library(Seurat)
library(mclust)
library(dplyr)
seuset <- CreateSeuratObject(
  raw.data = counts(deng),
  min.cells = 3,
  min.genes = 200
)
```

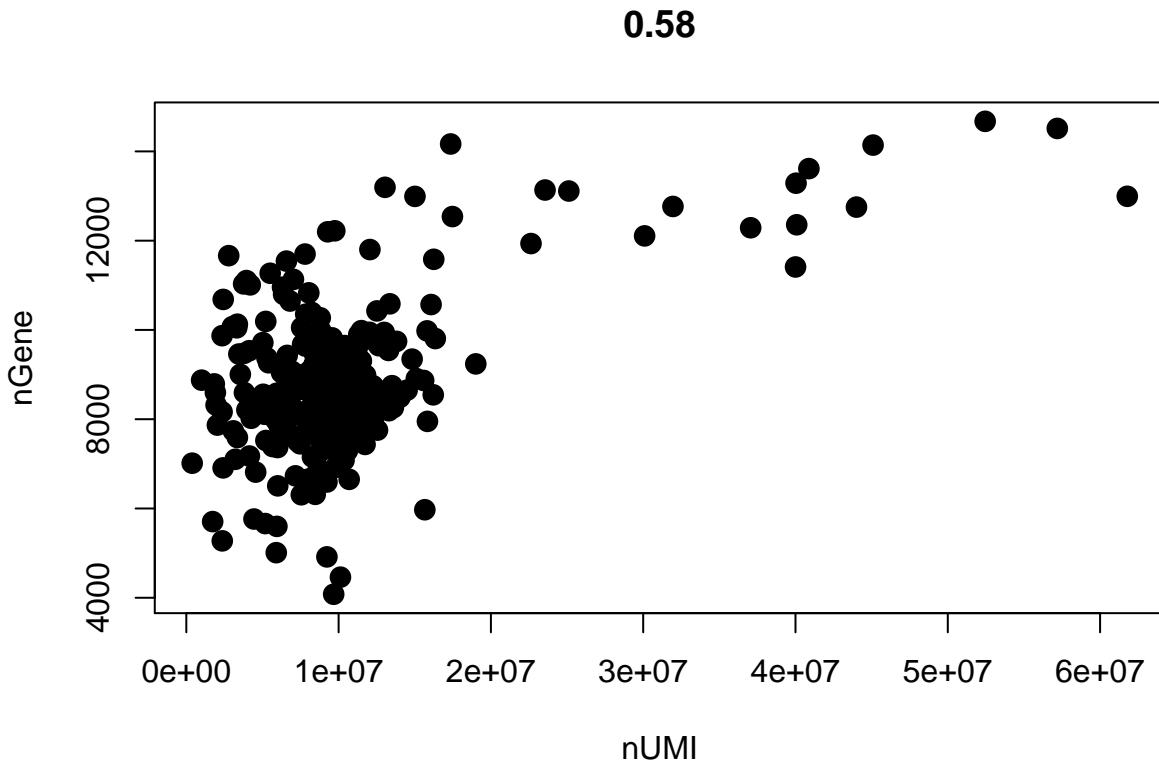
### 9.2 Expression QC

**Seurat** allows you to easily explore QC metrics and filter cells based on any user-defined criteria. We can visualize gene and molecule counts and plot their relationship:

```
VlnPlot(  
  object = seuset,  
  features.plot = c("nGene", "nUMI"),  
  nCol = 2  
)
```



```
GenePlot(  
  object = seuset,  
  gene1 = "nUMI",  
  gene2 = "nGene"  
)
```



Now we will exclude cells with a clear outlier number of read counts:

```
seuset <- FilterCells(
  object = seuset,
  subset.names = c("nUMI"),
  high.thresholds = c(2e7)
)
```

### 9.3 Normalization

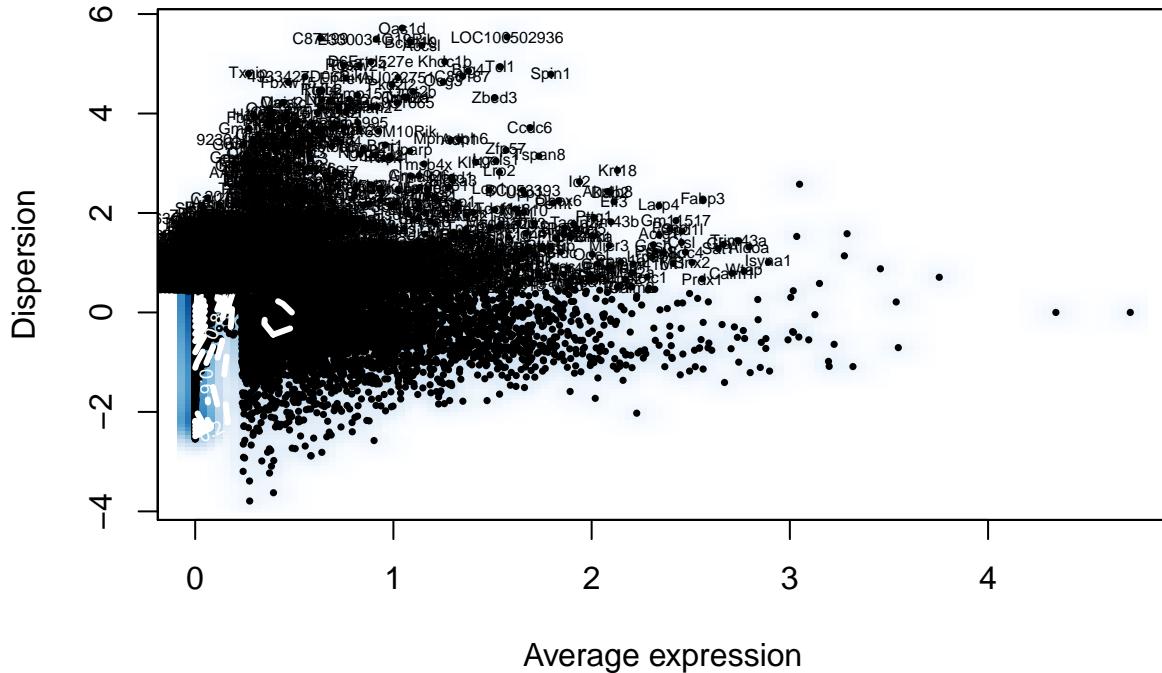
After removing unwanted cells from the dataset, the next step is to normalize the data. By default, we employ a global-scaling normalization method `LogNormalize` that normalizes the gene expression measurements for each cell by the total expression, multiplies this by a scale factor (10,000 by default), and log-transforms the result:

```
seuset <- NormalizeData(
  object = seuset,
  normalization.method = "LogNormalize",
  scale.factor = 10000
)
```

### 9.4 Highly variable genes

Seurat calculates highly variable genes and focuses on these for downstream analysis. `FindVariableGenes` calculates the average expression and dispersion for each gene, places these genes into bins, and then calculates a z-score for dispersion within each bin. This helps control for the relationship between variability and average expression:

```
seuset <- FindVariableGenes(
  object = seuset,
  mean.function = ExpMean,
  dispersion.function = LogVMR,
  x.low.cutoff = 0.0125,
  x.high.cutoff = 3,
  y.cutoff = 0.5
)
```



```
length(x = seuset@var.genes)
```

```
## [1] 6127
```

We are not entirely sure what is going on in the lower left hand corner of the plot above. A similar feature can be found in the Satija lab tutorial, so we do not believe that it is due to an error in how we used the method.

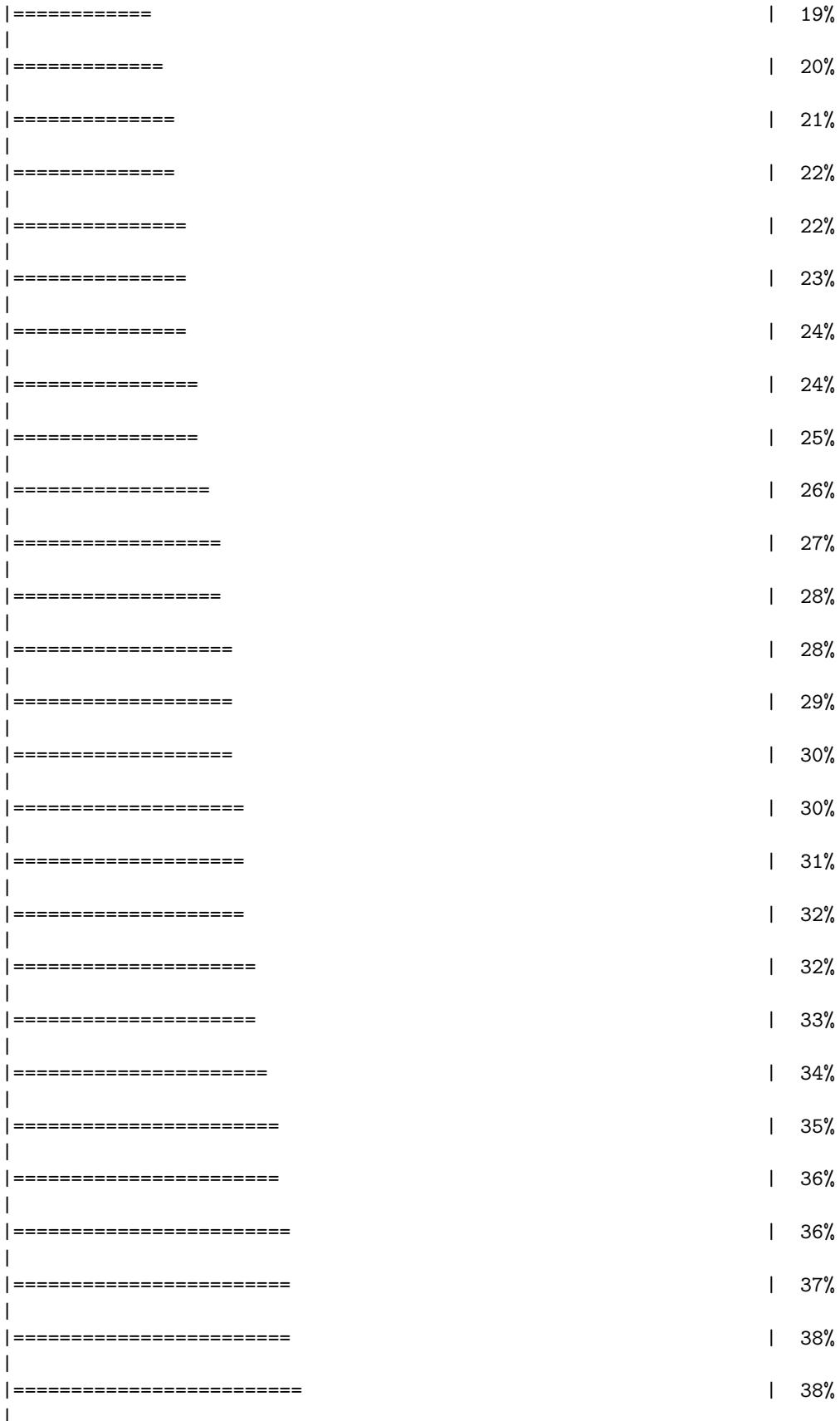
## 9.5 Dealing with confounders

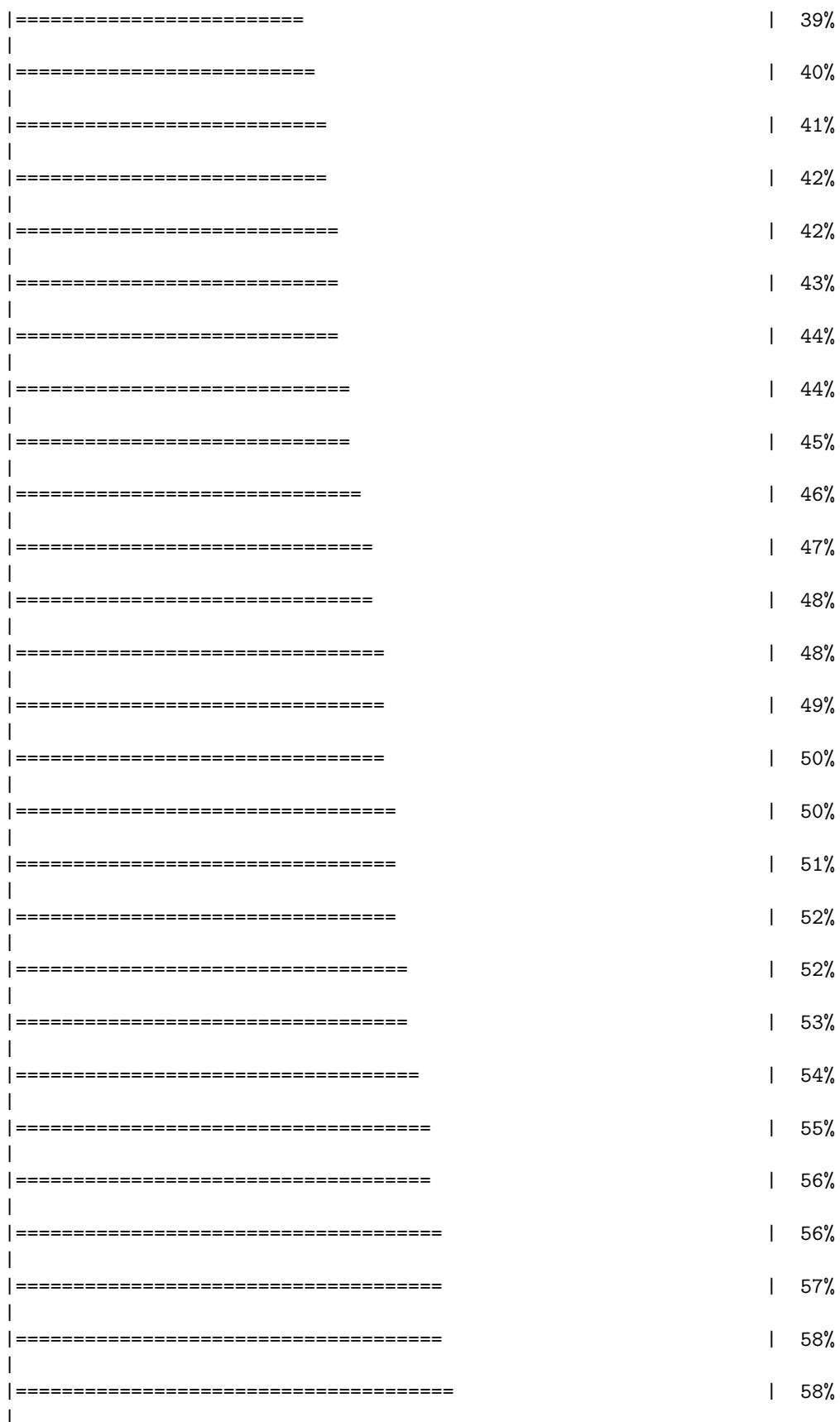
To mitigate the effect of confounding factors, **Seurat** constructs linear models to predict gene expression based on user-defined variables. The scaled z-scored residuals of these models are stored in the **scale.data** slot, and are used for dimensionality reduction and clustering.

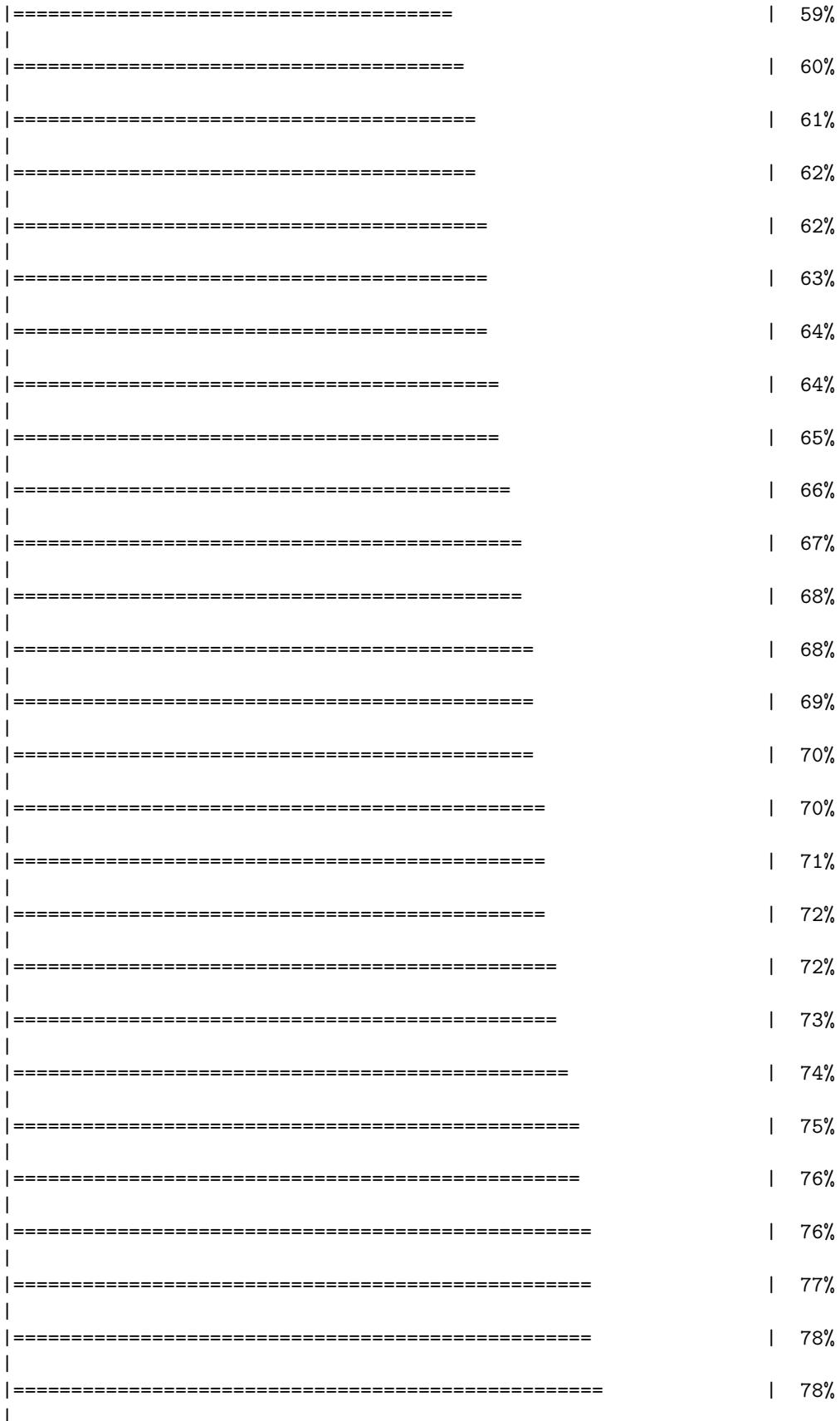
**Seurat** can regress out cell-cell variation in gene expression driven by batch, cell alignment rate (as provided by Drop-seq tools for Drop-seq data), the number of detected molecules, mitochondrial gene expression and cell cycle. Here we regress on the number of detected molecules per cell.

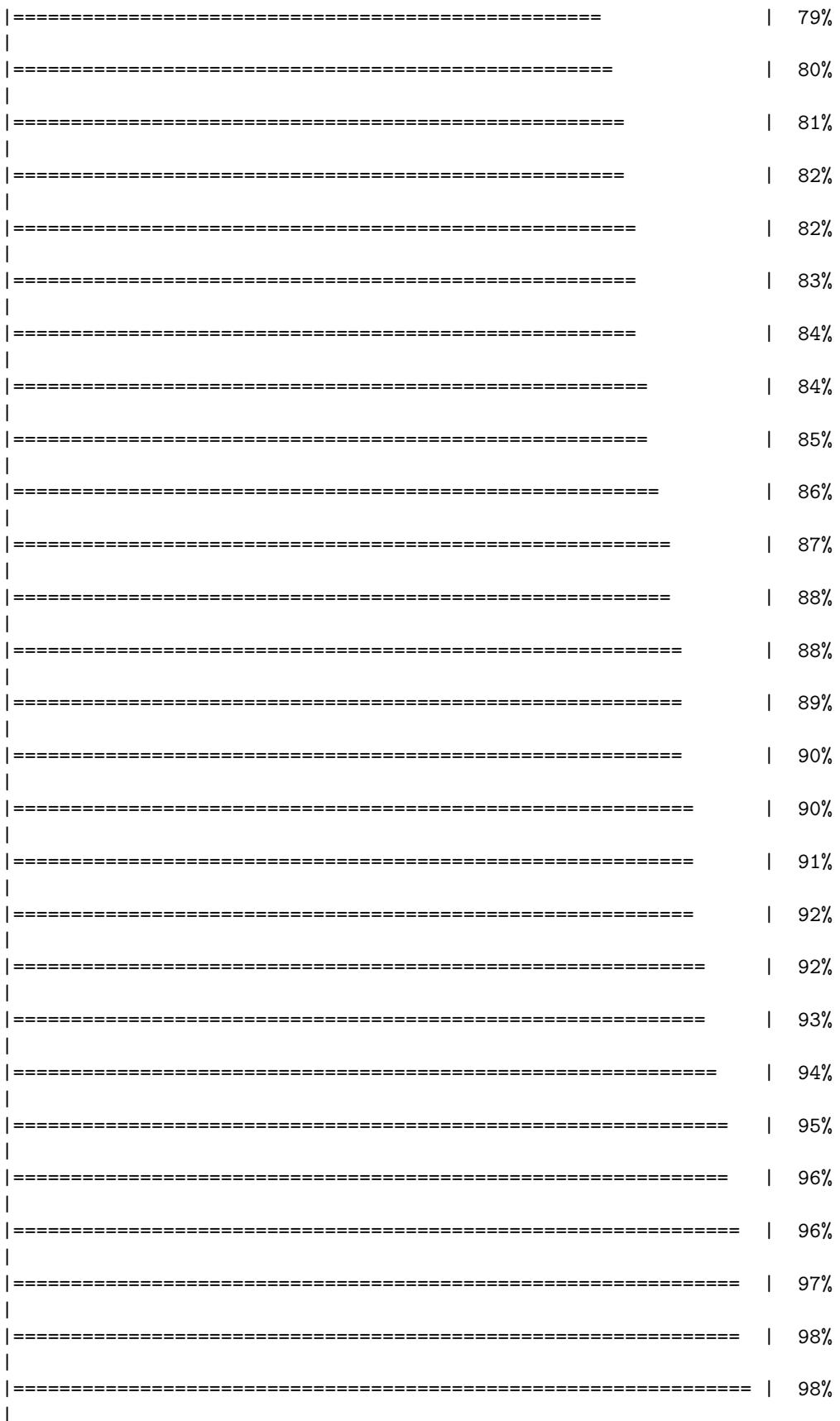
```
seuset <- ScaleData(
  object = seuset,
  vars.to.regress = c("nUMI")
)
## [1] "Regressing out nUMI"
```

```
##  
|  
|  
|  
|= | 0%  
|  
|= | 1%  
|  
|= | 2%  
|  
|==| 2%  
|  
|==| 3%  
|  
|==| 4%  
|  
|==| 4%  
|  
|==| 5%  
|  
|====| 6%  
|  
|=====| 7%  
|  
|=====| 8%  
|  
|=====| 8%  
|  
|=====| 9%  
|  
|=====| 10%  
|  
|=====| 10%  
|  
|=====| 11%  
|  
|=====| 12%  
|  
|=====| 12%  
|  
|=====| 13%  
|  
|=====| 14%  
|  
|=====| 15%  
|  
|=====| 16%  
|  
|=====| 16%  
|  
|=====| 17%  
|  
|=====| 18%  
|  
|=====| 18%
```









```

| ====== | 99%
|
| ====== | 100%
## [1] "Scaling data matrix"
##
| |
| | 0%
|
| ====== | 100%

```

## 9.6 Linear dimensionality reduction

Next we perform PCA on the scaled data. By default, the genes in `object@var.genes` are used as input, but can be alternatively defined using `pc.genes`. Running dimensionality reduction on highly variable genes can improve performance. However, with some types of data (UMI) - particularly after regressing out technical variables, PCA returns similar (albeit slower) results when run on much larger subsets of genes, including the whole transcriptome.

```

seuset <- RunPCA(
  object = seuset,
  pc.genes = seuset@var.genes,
  do.print = TRUE,
  pcs.print = 1:5,
  genes.print = 5
)

## [1] "PC1"
## [1] "Gm10436" "Zbed3"    "Gm13023" "Oog1"      "C86187"
## [1] ""
## [1] "Fbp2"     "Fam96a"   "Cstb"     "Lrpap1"   "Ctsd"
## [1] ""
## [1] ""
## [1] "PC2"
## [1] "Gsta4"    "Id2"      "Ptgr1"    "AA467197" "Myh9"
## [1] ""
## [1] "Gm11517"  "Obox6"    "Pdxk"     "Map1lc3a" "Cited1"
## [1] ""
## [1] ""
## [1] "PC3"
## [1] "Pssrc1"   "Ninj2"    "Gja4"     "Tdrd12"   "Wdr76"
## [1] ""
## [1] "Efnb2"     "Gm9125"   "Pabpn1"   "Mad2l1bp"
## [5] "1600025M17Rik"
## [1] ""
## [1] ""
## [1] "PC4"
## [1] "Upp1"     "Tdgf1"    "Baz2b"    "Rnd3"     "Col4a1"
## [1] ""
## [1] "Rragd"    "Ppfibp2"  "Smpdl3a"  "Cldn4"    "Amotl2"
## [1] ""
## [1] ""
## [1] "PC5"
## [1] "Snhg8"    "Trappc2"  "Acsm2"    "Angptl2"  "Nlgn1"

```

```

## [1] ""
## [1] "Akap1"   "Stub1"    "Apoe"     "Scand1"  "Hjurp"
## [1] ""
## [1] ""

```

`Seurat` provides several useful ways of visualizing both cells and genes that define the PCA:

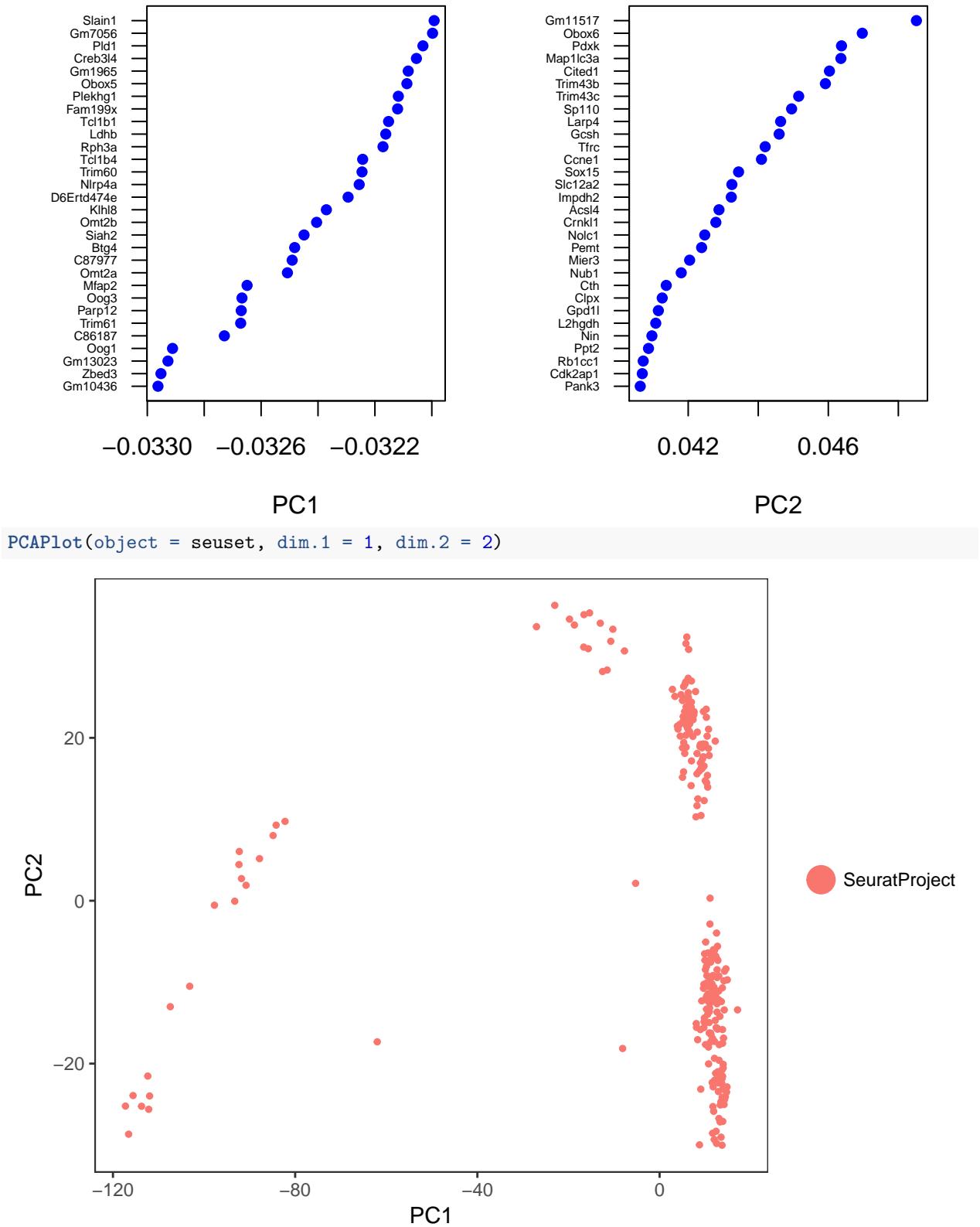
```
PrintPCA(object = seuset, pcs.print = 1:5, genes.print = 5, use.full = FALSE)
```

```

## [1] "PC1"
## [1] "Gm10436"  "Zbed3"    "Gm13023"  "Oog1"      "C86187"
## [1] ""
## [1] "Fbp2"      "Fam96a"   "Cstb"     "Lrpap1"   "Ctsd"
## [1] ""
## [1] ""
## [1] "PC2"
## [1] "Gsta4"     "Id2"       "Ptgr1"    "AA467197"  "Myh9"
## [1] ""
## [1] "Gm11517"   "Obox6"    "Pdxk"     "Map1lc3a"  "Cited1"
## [1] ""
## [1] ""
## [1] "PC3"
## [1] "Psrc1"     "Ninj2"    "Gja4"     "Tdrd12"   "Wdr76"
## [1] ""
## [1] "Efnb2"      "Gm9125"   "Pabpn1"   "Mad2l1bp"
## [5] "1600025M17Rik"
## [1] ""
## [1] ""
## [1] "PC4"
## [1] "Upp1"      "Tdgf1"    "Baz2b"    "Rnd3"     "Col4a1"
## [1] ""
## [1] "Rragd"     "Ppfibp2"  "Smpdl3a"  "Cldn4"    "Amotl2"
## [1] ""
## [1] ""
## [1] "PC5"
## [1] "Snhg8"     "Trappc2"  "Acsm2"    "Angptl2"  "Nlgn1"
## [1] ""
## [1] "Akap1"     "Stub1"    "Apoe"     "Scand1"  "Hjurp"
## [1] ""
## [1] ""

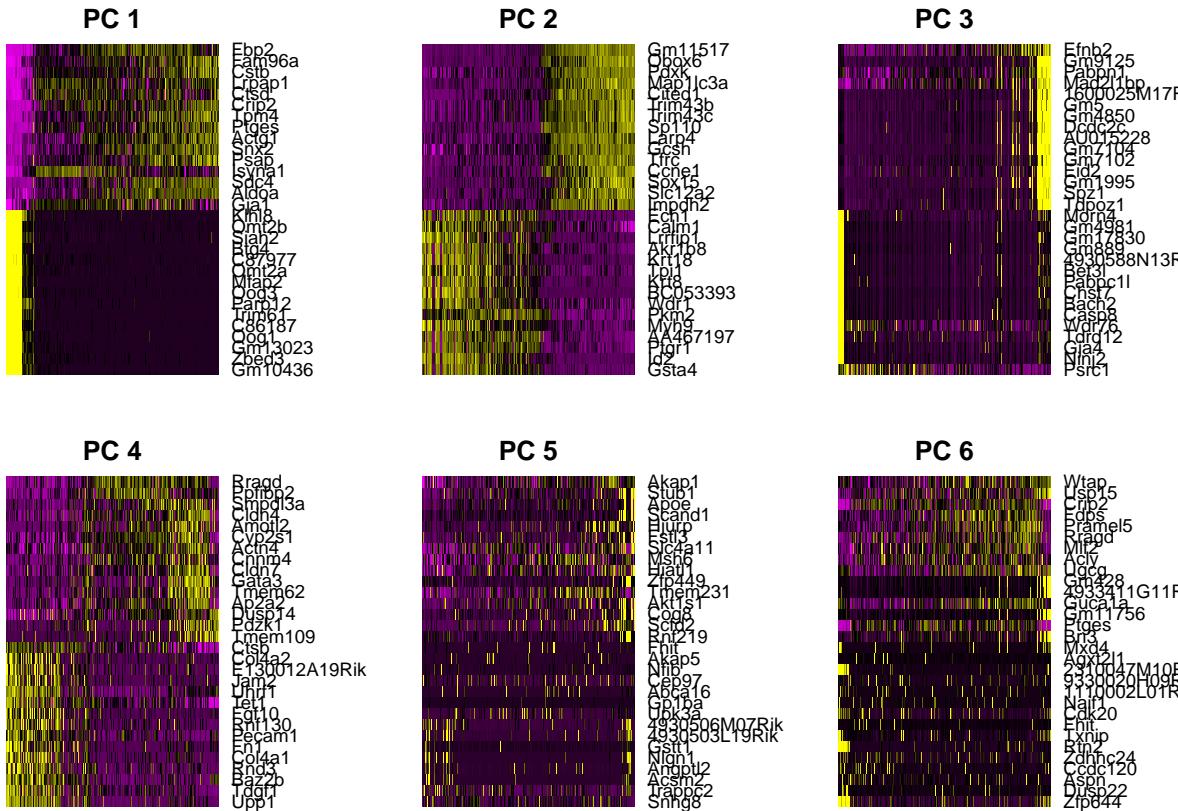
```

```
VizPCA(object = seuset, pcs.use = 1:2)
```



In particular, PCHeatmap allows for easy exploration of the primary sources of heterogeneity in a dataset, and can be useful when trying to decide which PCs to include for further downstream analyses. Both cells and genes are ordered according to their PCA scores. Setting `cells.use` to a number plots the *extreme* cells on both ends of the spectrum, which dramatically speeds plotting for large datasets:

```
PCHeatmap(
  object = seuset,
  pc.use = 1:6,
  cells.use = 500,
  do.balanced = TRUE,
  label.columns = FALSE,
  use.full = FALSE
)
```



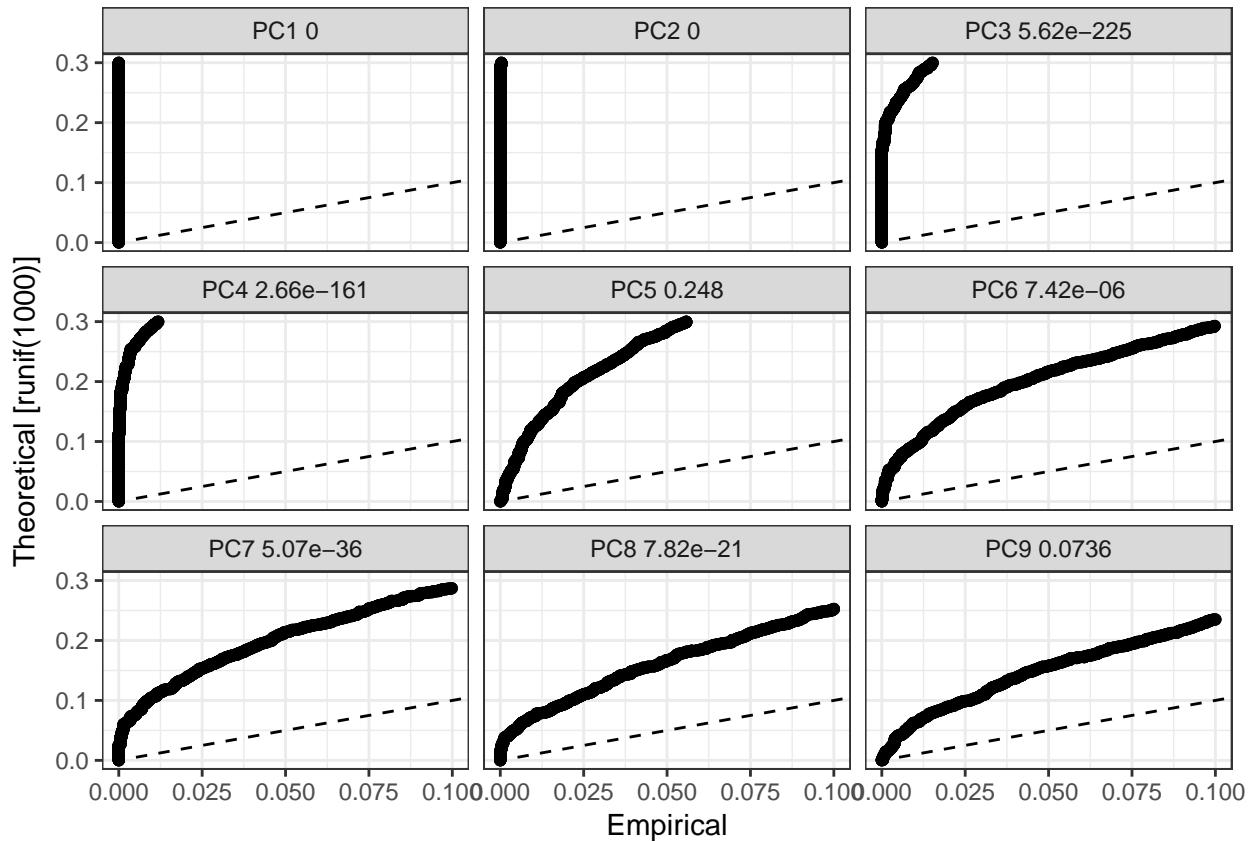
To overcome the extensive technical noise in any single gene for scRNA-seq data, **Seurat** clusters cells based on their PCA scores, with each PC essentially representing a *metagene* that combines information across a correlated gene set. Determining how many PCs to include downstream is therefore an important step. **Seurat** randomly permutes a subset of the data (1% by default) and reruns PCA, constructing a *null distribution* of gene scores by repeating this procedure. We identify *significant* PCs as those who have a strong enrichment of low p-value genes:

```
seuset <- JackStraw(
  object = seuset,
  num.replicate = 100,
  do.print = FALSE
)
```

The **JackStrawPlot** function provides a visualization tool for comparing the distribution of p-values for each PC with a uniform distribution (dashed line). *Significant* PCs will show a strong enrichment of genes with low p-values (solid curve above the dashed line). In this case it appears that PCs 1-8 are significant.

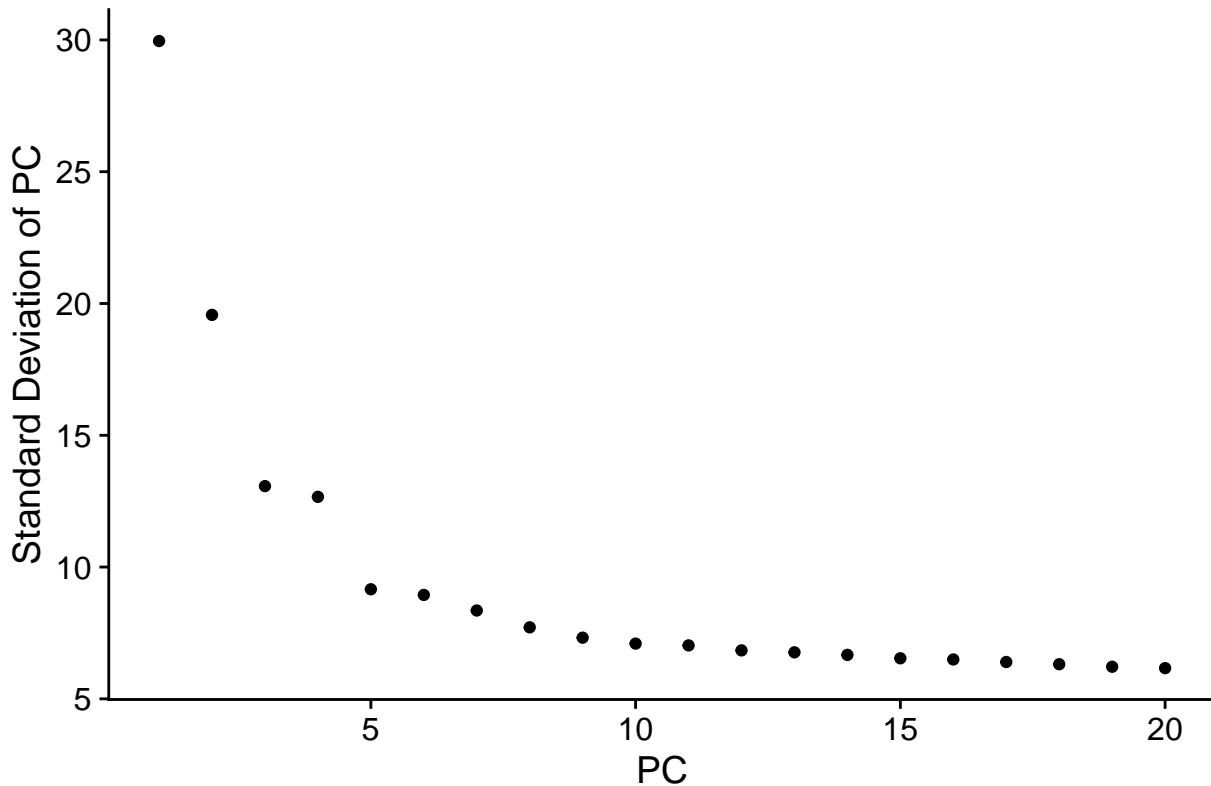
```
JackStrawPlot(object = seuset, PCs = 1:9)
```

```
## Warning: Removed 39410 rows containing missing values (geom_point).
```



A more ad hoc method for determining which PCs to use is to look at a plot of the standard deviations of the principle components and draw your cutoff where there is a clear elbow in the graph. This can be done with `PCElbowPlot`. In this example, it looks like the elbow would fall around PC 5.

```
PCElbowPlot(object = seuset)
```



## 9.8 Clustering cells

`Seurat` implements an graph-based clustering approach. Distances between the cells are calculated based on previously identified PCs. `Seurat` approach was heavily inspired by recent manuscripts which applied graph-based clustering approaches to scRNA-seq data - SNN-Cliq ((Xu and Su, 2015)) and CyTOF data - PhenoGraph ((Levine et al., 2015)). Briefly, these methods embed cells in a graph structure - for example a K-nearest neighbor (*KNN*) graph, with edges drawn between cells with similar gene expression patterns, and then attempt to partition this graph into highly interconnected *quasi-cliques* or *communities*. As in PhenoGraph, we first construct a *KNN* graph based on the euclidean distance in PCA space, and refine the edge weights between any two cells based on the shared overlap in their local neighborhoods (Jaccard distance). To cluster the cells, we apply modularity optimization techniques - SLM ((Blondel et al., 2008)), to iteratively group cells together, with the goal of optimizing the standard modularity function.

The `FindClusters` function implements the procedure, and contains a resolution parameter that sets the granularity of the downstream clustering, with increased values leading to a greater number of clusters. We find that setting this parameter between 0.6 – 1.2 typically returns good results for single cell datasets of around 3,000 cells. Optimal resolution often increases for larger datasets. The clusters are saved in the `object@ident` slot.

```
seuset <- FindClusters(
  object = seuset,
  reduction.type = "pca",
  dims.use = 1:8,
  resolution = 1.0,
  print.output = 0,
  save.SNN = TRUE
)
```

A useful feature in **Seurat** is the ability to recall the parameters that were used in the latest function calls for commonly used functions. For `FindClusters`, there is the function `PrintFindClustersParams` to print a nicely formatted summary of the parameters that were chosen:

```
PrintFindClustersParams(object = seuset)
```

```
## Parameters used in latest FindClusters calculation run on: 2018-02-27 21:15:36
## =====
## Resolution: 1
##
## -----
## Modularity Function      Algorithm          n.start      n.iter
##   1                      1                  100          10
## -----
## Reduction used           k.param          k.scale      prune.SNN
##   pca                     30                 25          0.0667
## -----
## Dims used in calculation
## =====
## 1 2 3 4 5 6 7 8
```

We can look at the clustering results and compare them to the original cell labels:

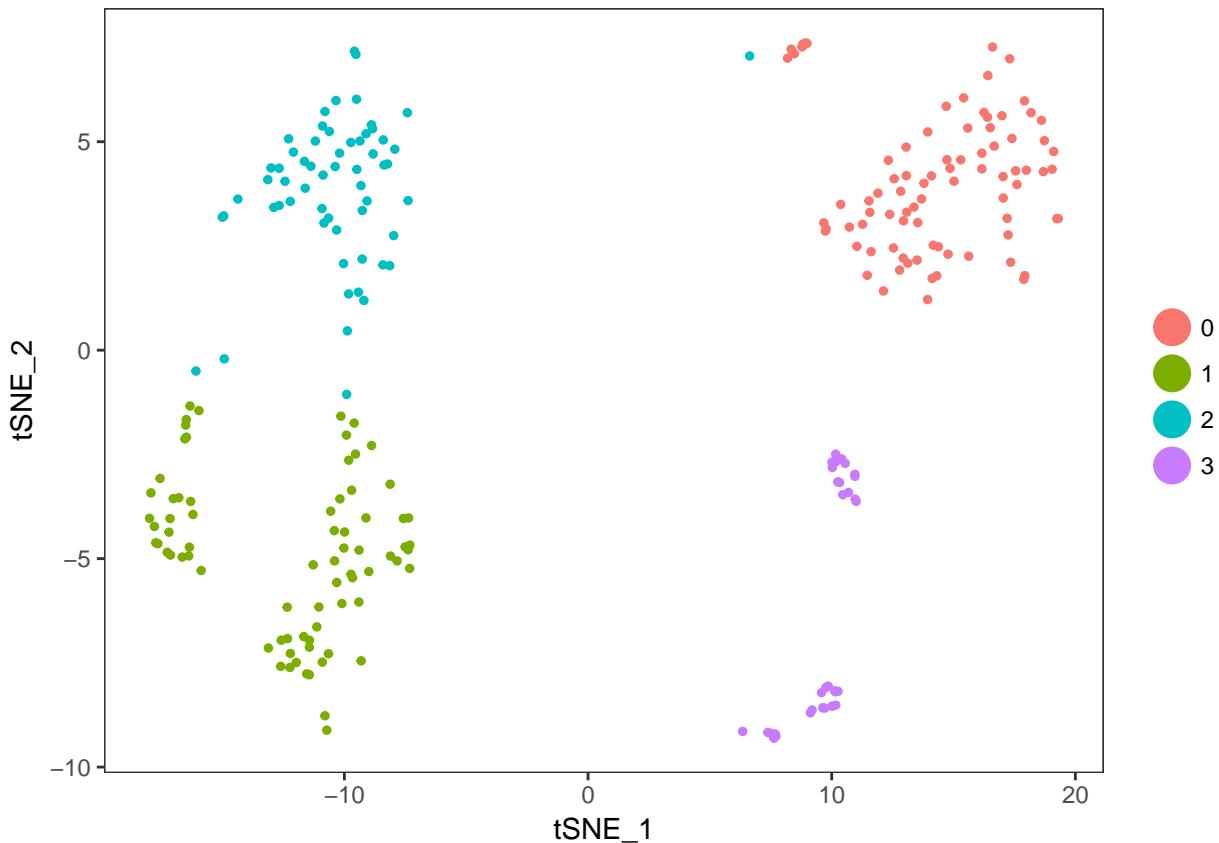
```
table(seuset@ident)
```

```
##
##    0    1    2    3
## 85  75  59  34
adjustedRandIndex(colData(deng)[seuset@cell.names, ]$cell_type2, seuset@ident)
```

```
## [1] 0.3981315
```

**Seurat** also utilises tSNE plot to visualise clustering results. As input to the tSNE, we suggest using the same PCs as input to the clustering analysis, although computing the tSNE based on scaled gene expression is also supported using the `genes.use` argument.

```
seuset <- RunTSNE(
  object = seuset,
  dims.use = 1:8,
  do.fast = TRUE
)
TSNEPlot(object = seuset)
```



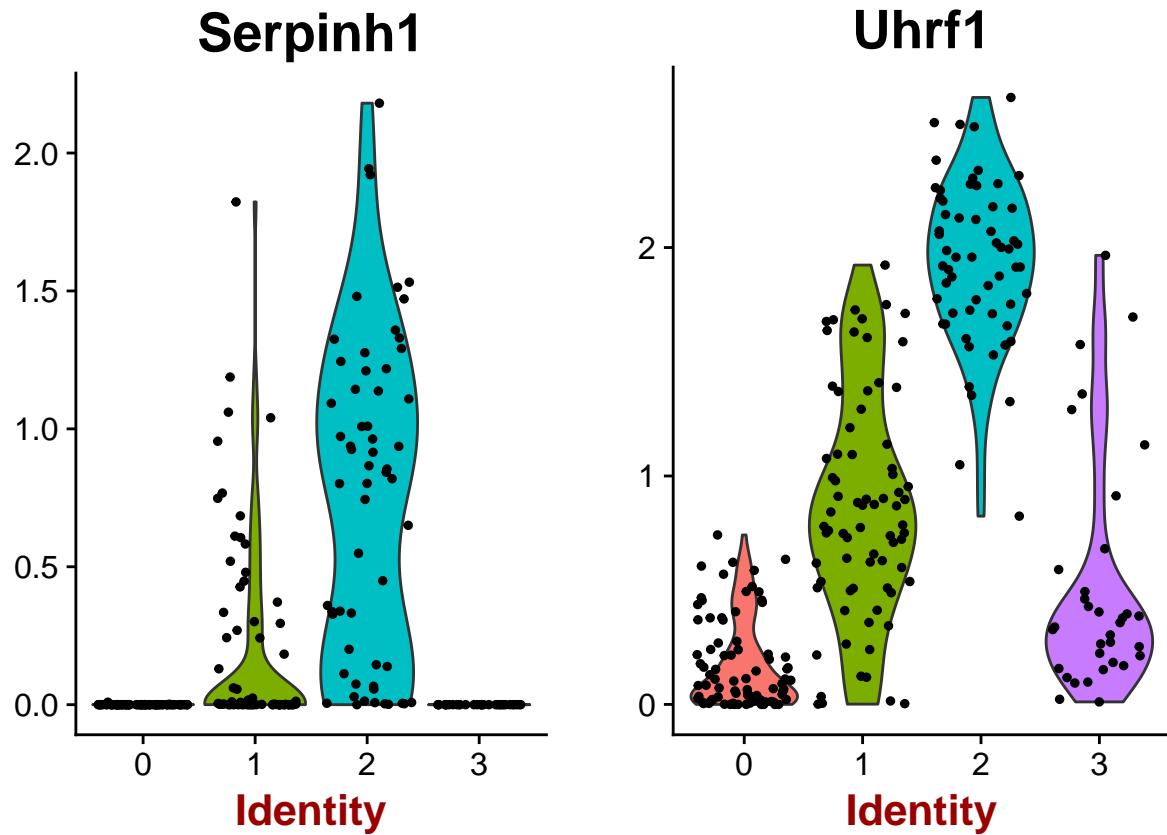
## 9.9 Marker genes

Seurat can help you find markers that define clusters via differential expression. By default, it identifies positive and negative markers of a single cluster, compared to all other cells. You can test groups of clusters vs. each other, or against all cells. For example, to find marker genes for cluster 2 we can run:

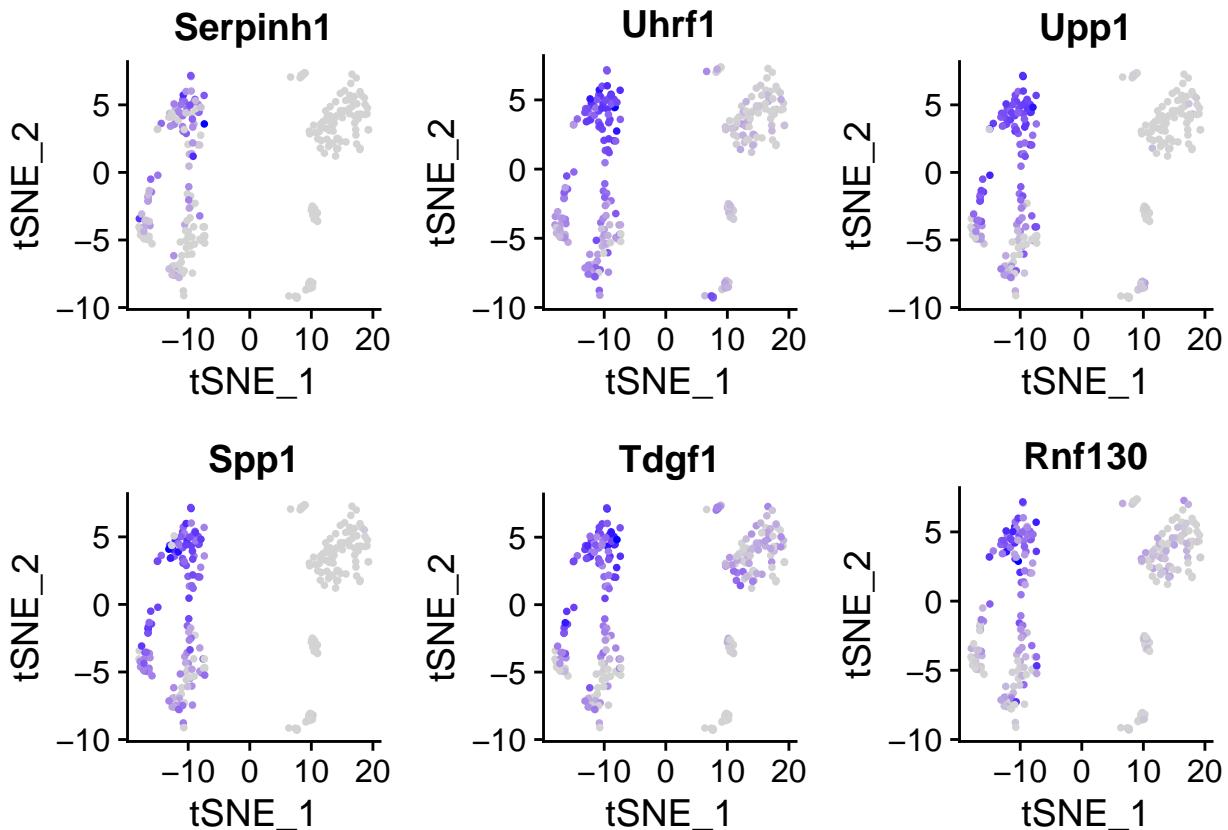
```
markers2 <- FindMarkers(seuset, 2)
```

Marker genes can then be visualised:

```
VlnPlot(object = seuset, features.plot = rownames(markers2)[1:2])
```



```
FeaturePlot(  
  seuset,  
  head(rownames(markers2)),  
  cols.use = c("lightgrey", "blue"),  
  nCol = 3  
)
```

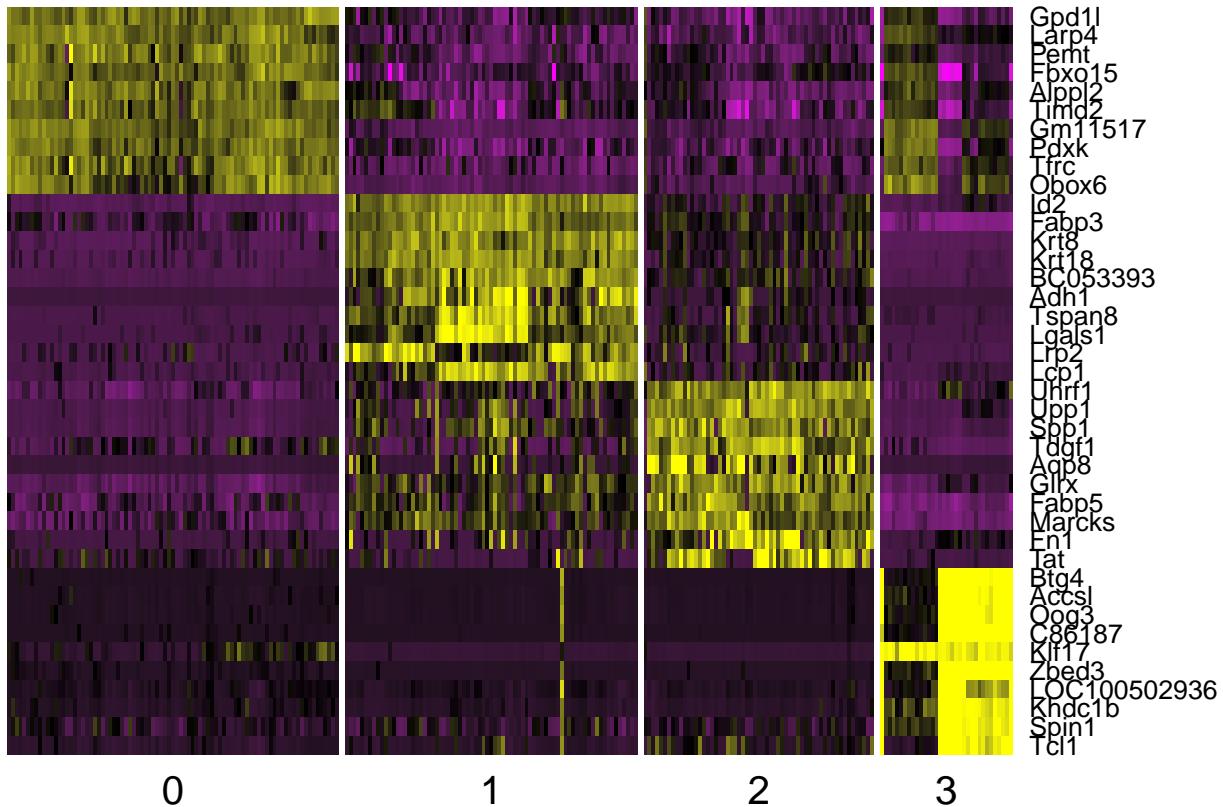


FindAllMarkers automates this process and find markers for all clusters:

```
markers <- FindAllMarkers(
  object = seuset,
  only.pos = TRUE,
  min.pct = 0.25,
  thresh.use = 0.25
)
```

DoHeatmap generates an expression heatmap for given cells and genes. In this case, we are plotting the top 10 markers (or all markers if less than 20) for each cluster:

```
top10 <- markers %>% group_by(cluster) %>% top_n(10, avg_logFC)
DoHeatmap(
  object = seuset,
  genes.use = top10$gene,
  slim.col.label = TRUE,
  remove.key = TRUE
)
```



**Exercise:** Compare marker genes provided by Seurat and SC3.

## 9.10 sessionInfo()

```
## R version 3.4.3 (2017-11-30)
## Platform: x86_64-pc-linux-gnu (64-bit)
## Running under: Debian GNU/Linux 9 (stretch)
##
## Matrix products: default
## BLAS: /usr/lib/openblas-base/libblas.so.3
## LAPACK: /usr/lib/libopenblas-r0.2.19.so
##
## locale:
##   [1] LC_CTYPE=en_US.UTF-8          LC_NUMERIC=C
##   [3] LC_TIME=en_US.UTF-8          LC_COLLATE=en_US.UTF-8
##   [5] LC_MONETARY=en_US.UTF-8      LC_MESSAGES=C
##   [7] LC_PAPER=en_US.UTF-8         LC_NAME=C
##   [9] LC_ADDRESS=C                 LC_TELEPHONE=C
##  [11] LC_MEASUREMENT=en_US.UTF-8    LC_IDENTIFICATION=C
##
## attached base packages:
## [1] parallel  stats4    methods   stats     graphics  grDevices utils
## [8] datasets  base
##
## other attached packages:
## [1] bindr_0.2.0            dplyr_0.7.4
## [3] mclust_5.4             Seurat_2.2.0
```

```

## [5] Matrix_1.2-7.1          cowplot_0.9.2
## [7] ggplot2_2.2.1           SingleCellExperiment_1.0.0
## [9] SummarizedExperiment_1.8.1 DelayedArray_0.4.1
## [11] matrixStats_0.53.0      Biobase_2.38.0
## [13] GenomicRanges_1.30.1    GenomeInfoDb_1.14.0
## [15] IRanges_2.12.0          S4Vectors_0.16.0
## [17] BiocGenerics_0.24.0    knitr_1.19
##
## loaded via a namespace (and not attached):
## [1] backports_1.1.2          Hmisc_4.1-1           VGAM_1.0-4
## [4] NMF_0.20.6                sn_1.5-1             plyr_1.8.4
## [7] igraph_1.1.2              lazyeval_0.2.1       splines_3.4.3
## [10] gridBase_0.4-7            digest_0.6.15       foreach_1.4.4
## [13] htmltools_0.3.6          lars_1.2             gdata_2.18.0
## [16] magrittr_1.5              checkmate_1.8.5     cluster_2.0.6
## [19] doParallel_1.0.11         mixtools_1.1.0      ROCR_1.0-7
## [22] sfsmisc_1.1-1            recipes_0.1.2       gower_0.1.2
## [25] dimRed_0.1.0              R.utils_2.6.0        colorspace_1.3-2
## [28] xfun_0.1                 RCurl_1.95-4.10    bindr_0.1
## [31] survival_2.40-1           iterators_1.0.9     ape_5.0
## [34] glue_1.2.0                DRR_0.0.3            registry_0.5
## [37] gtable_0.2.0              ipred_0.9-6         zlibbioc_1.24.0
## [40] XVector_0.18.0            kernlab_0.9-25     ddalpha_1.3.1.1
## [43] prabclus_2.2-6            DEoptimR_1.0-8      scales_0.5.0
## [46] mvtnorm_1.0-7              rngtools_1.2.4      Rcpp_0.12.15
## [49] dtw_1.18-1                xtable_1.8-2        htmlTable_1.11.2
## [52] tclust_1.3-1              foreign_0.8-67     proxy_0.4-21
## [55] SDMTools_1.1-221          Formula_1.2-2      tsne_0.1-3
## [58] lava_1.6                  prodlim_1.6.1       htmlwidgets_1.0
## [61] FNN_1.1                   gplots_3.0.1         RColorBrewer_1.1-2
## [64] fpc_2.1-11                acepack_1.4.1       modeltools_0.2-21
## [67] ica_1.0-1                 pkgconfig_2.0.1     R.methodsS3_1.7.1
## [70] flexmix_2.3-14             nnet_7.3-12         caret_6.0-78
## [73] labeling_0.3               tidyselect_0.2.3    rlang_0.1.6
## [76] reshape2_1.4.3              munsell_0.4.3       tools_3.4.3
## [79] ranger_0.9.0               broom_0.4.3         ggridges_0.4.1
## [82] evaluate_0.10.1             stringr_1.2.0       yaml_2.1.16
## [85] ModelMetrics_1.1.0          robustbase_0.92-8   caTools_1.17.1
## [88] purrrr_0.2.4               pbapply_1.3-4        nlme_3.1-129
## [91] R.oo_1.21.0                RcppRoll_0.2.2      compiler_3.4.3
## [94] rstudioapi_0.7              tibble_1.4.2         stringi_1.1.6
## [97] lattice_0.20-34             trimcluster_0.1-2   psych_1.7.8
## [100] diffusionMap_1.1-0        pillar_1.1.0         irlba_2.3.2
## [103] data.table_1.10.4-3        bitops_1.0-6         R6_2.2.2
## [106] latticeExtra_0.6-28        bookdown_0.6        KernSmooth_2.23-15
## [109] gridExtra_2.3              codetools_0.2-15    MASS_7.3-45
## [112] gtools_3.5.0               assertthat_0.2.0     CVST_0.2-1
## [115] pkgmaker_0.22              rprojroot_1.3-2      withr_2.1.1
## [118] mnormt_1.5-5              GenomeInfoDbData_1.0.0 diptest_0.75-7
## [121] grid_3.4.3                 rpart_4.1-10        timeDate_3042.101
## [124] tidyrr_0.8.0                class_7.3-14         rmarkdown_1.8
## [127] segmented_0.5-3.0           Rtsne_0.13          numDeriv_2016.8-1
## [130] scatterplot3d_0.3-40       lubridate_1.7.2      base64enc_0.1-3

```



# Chapter 10

## “Ideal” scRNAseq pipeline (as of Oct 2017)

### 10.1 Experimental Design

- Avoid confounding biological and batch effects (Figure 10.1)
  - Multiple conditions should be captured on the same chip if possible
  - Perform multiple replicates of each condition where replicates of different conditions should be performed together if possible
  - Statistics cannot correct a completely confounded experiment!
- Unique molecular identifiers
  - Greatly reduce noise in data
  - May reduce gene detection rates (unclear if it is UMIs or other protocol differences)
  - Lose splicing information
  - Use longer UMIs (~10bp)
  - Correct for sequencing errors in UMIs using UMI-tools
- Spike-ins
  - Useful for quality control
  - May be useful for normalizing read counts
  - Can be used to approximate cell-size/RNA content (if relevant to biological question)
  - Often exhibit higher noise than endogenous genes (pipetting errors, mixture quality)
  - Requires more sequencing to get enough endogenous reads per cell
- Cell number vs Read depth
  - Gene detection plateaus starting from 1 million reads per cell
  - Transcription factor detection (regulatory networks) require high read depth and most sensitive protocols (i.e. Fluidigm C1)
  - Cell clustering & cell-type identification benefits from large number of cells and doesn’t require as high sequencing depth (~100,000 reads per cell).

### 10.2 Processing Reads

- Read QC & Trimming
  - FASTQC, cutadapt
- Mapping
  - Small datasets or UMI datasets: align to genome/transcriptome using STAR
  - Large datasets: pseudo-alignment with Salmon or kallisto

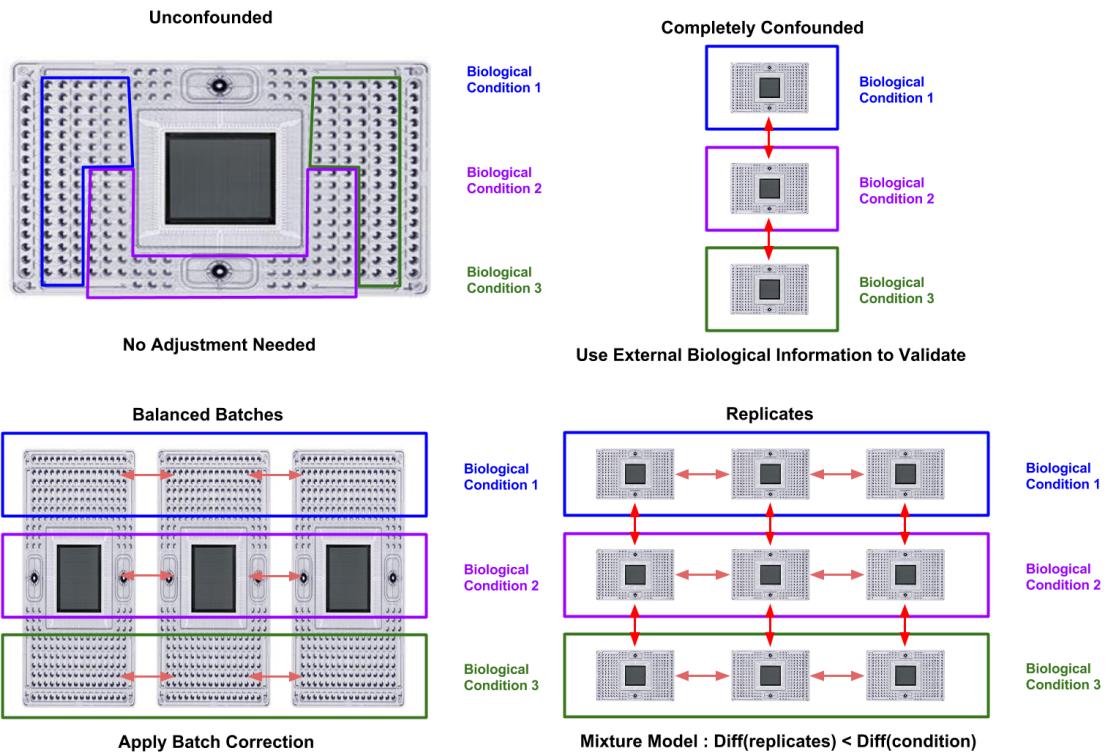


Figure 10.1: Appropriate approaches to batch effects in scRNASeq. Red arrows indicate batch effects which are (pale) or are not (vibrant) correctable through batch-correction.

- Quantification
  - Small dataset, no UMIs : featureCounts
  - Large datasets, no UMIs: Salmon, kallisto
  - UMI dataset : UMI-tools + featureCounts

## 10.3 Preparing Expression Matrix

- Cell QC
  - scater
  - consider: mtRNA, rRNA, spike-ins (if available), number of detected genes per cell, total reads/molecules per cell
- Library Size Normalization
  - scran
- Batch correction (if appropriate)
  - Replicates/Confounded RUVs
  - Unknown or unbalanced biological groups mnncorrect
  - Balanced design ComBat

## 10.4 Biological Interpretation

- Feature Selection
  - M3Drop
- Clustering and Marker Gene Identification
  - $\leq 5000$  cells : SC3
  - $> 5000$  cells : Seurat
- Pseudotime
  - distinct timepoints: TSCAN
  - small dataset/unknown number of branches: Monocle2
  - large continuous dataset: destiny
- Differential Expression
  - Small number of cells and few groups : scde
  - Replicates with batch effects : mixture/linear models
  - Balanced batches: edgeR or MAST
  - Large datasets: Kruskal-Wallis test (all groups at once), or Wilcox-test (compare 2-groups at a time).



# Chapter 11

## Advanced exercises

For the final part of the course we would like you to work on more open ended problems. The goal is to carry out the type of analyses that you would be doing for an actual research project.

Participants who have their own dataset that they are interested in should feel free to work with them.

For other participants we recommend downloading a dataset from the conquer resource (consistent quantification of external rna-seq data). conquer uses Salmon to quantify the transcript abundances in a given sample. For a given organism, the fasta files containing cDNA and ncRNA sequences from Ensembl are complemented with ERCC spike-in sequences, and a Salmon quasi-mapping index is built for the entire catalog. Then Salmon is run to estimate the abundance of each transcript. The abundances estimated by Salmon are summarised and provided to the user in the form of a `MultiAssayExperiment` object. This object can be downloaded via the buttons in the `MultiAssayExperiment` column. The provided `MultiAssayExperiment` object contains two “experiments”, corresponding to the gene-level and transcript-level values.

The gene-level experiment contains four “assays”:

- TPM
- count
- count\_lstpm (count-scale length-scaled TPMs)
- avetxlength (the average transcript length, which can be used as offsets in count models based on the count assay, see here).

The transcript-level experiment contains three “assays”:

- TPM
- count
- efflength (the effective length estimated by Salmon)

The `MultiAssayExperiment` also contains the phenotypic data (in the `colData` slot), as well as some metadata for the data set (the genome, the organism and the Salmon index that was used for the quantification).

Here we will show you how to create an `SCE` from a `MultiAssayExperiment` object. For example, if you download Shalek2013 dataset you will be able to create an `SCE` using the following code:

```
library(MultiAssayExperiment)
library(SummarizedExperiment)
library(scater)
d <- readRDS("~/Desktop/GSE41265.rds")
cts <- assays(experiments(d)[["gene"]])[["count_lstpm"]]
tpms <- assays(experiments(d)[["gene"]])[["TPM"]]
```

```
phn <- colData(d)
sce <- SingleCellExperiment(
  assays = list(
    countData = cts,
    tpmData = tpms
  ),
  colData = phn
)
```

You can also see that several different QC metrics have already been pre-calculated on the conquer website.

Here are some suggestions for questions that you can explore:

- There are two mESC datasets from different labs (i.e. Xue and Kumar). Can you merge them and remove the batch effects?
- Clustering and pseudotime analysis look for different patterns among cells. How might you tell which is more appropriate for your dataset?
- One of the main challenges in hard clustering is to identify the appropriate value for  $k$ . Can you use one or more of the clustering tools to explore the different hierarchies available? What are good mathematical and/or biological criteria for determining  $k$ ?
- The choice of normalization strategy matters, but how do you determine which is the best method? Explore the effect of different normalizations on downstream analyses.
- scRNA-seq datasets are high-dimensional and since most dimensions (ie genes) are not informative. Consequently, dimensionality reduction and feature selection are important when analyzing and visualizing the data. Consider the effect of different feature selection methods and dimensionality reduction on clustering and/or pseudotime inference.
- One of the main challenges after clustering cells is to interpret the biological relevance of the sub-populations. One approach is to identify gene ontology terms that are enriched for the set of marker genes. Identify marker genes (e.g. using SC3 or M3Drop) and explore the ontology terms using gProfiler, WebGestalt or DAVID.
- Similarly, when ordering cells according to pseudotime we would like to understand what underlying cellular processes are changing over time. Identify a set of changing genes from the aligned cells and use ontology terms to characterize them.

# Chapter 12

## Resources

### 12.1 scRNA-seq protocols

- SMART-seq2
- CELL-seq
- Drop-seq
- UMI
- STRT-Seq

### 12.2 External RNA Control Consortium (ERCC)

ERCCs

### 12.3 scRNA-seq analysis tools

Extensive list of software packages (and the people developing these methods) for single-cell data analysis:

- awesome-single-cell  
Tallulah Andrews' single cell processing scripts:
- scRNASeqPipeline

### 12.4 scRNA-seq public datasets

- Hemberg group's public datasets



# Bibliography

- Altschul, S. F., Gish, W., Miller, W., Myers, E. W., and Lipman, D. J. (1990). Basic local alignment search tool. *Journal of Molecular Biology*, 215(3):403–410.
- Anders, S. and Huber, W. (2010). Differential expression analysis for sequence count data. *Genome Biol*, 11(10):R106.
- Archer, N., Walsh, M. D., Shahrezaei, V., and Hebenstreit, D. (2016). Modeling enzyme processivity reveals that RNA-seq libraries are biased in characteristic and correctable ways. *Cell Systems*, 3(5):467–479.e12.
- Blondel, V. D., Guillaume, J.-L., Lambiotte, R., and Lefebvre, E. (2008). Fast unfolding of communities in large networks. *J. Stat. Mech.*, 2008(10):P10008.
- Bray, N. L., Pimentel, H., Melsted, P., and Pachter, L. (2016). Near-optimal probabilistic rna-seq quantification. *Nat Biotechnol*, 34(5):525–527.
- Bullard, J. H., Purdom, E., Hansen, K. D., and Dudoit, S. (2010). Evaluation of statistical methods for normalization and differential expression in mRNA-seq experiments. *BMC Bioinformatics*, 11(1):94.
- Buttner, M., Miao, Z., Wolf, A., Teichmann, S. A., and Theis, F. J. (2017). Assessment of batch-correction methods for scRNA-seq data with a new test metric. *bioRxiv*, page 200345.
- Cannoodt, R., Saelens, W., and Saeys, Y. (2016). Computational methods for trajectory inference from single-cell transcriptomics. *Eur. J. Immunol.*, 46(11):2496–2506.
- Deng, Q., Ramskold, D., Reinius, B., and Sandberg, R. (2014). Single-cell RNA-seq reveals dynamic, random monoallelic gene expression in mammalian cells. *Science*, 343(6167):193–196.
- Gierahn, T. M., Wadsworth, 2nd, M. H., Hughes, T. K., Bryson, B. D., Butler, A., Satija, R., Fortune, S., Love, J. C., and Shalek, A. K. (2017). Seq-Well: Portable, low-cost RNA sequencing of single cells at high throughput. *Nat. Methods*, 14(4):395–398.
- Guo, M., Wang, H., Potter, S. S., Whitsett, J. A., and Xu, Y. (2015). SINCERA: A pipeline for single-cell RNA-seq profiling analysis. *PLoS Comput Biol*, 11(11):e1004575.
- Haghverdi, L., Lun, A. T. L., Morgan, M. D., and Marioni, J. C. (2017). Correcting batch effects in single-cell RNA sequencing data by matching mutual nearest neighbours. *bioRxiv*, page 165118.
- Hashimshony, T., Senderovich, N., Avital, G., Klochendler, A., de Leeuw, Y., Anavy, L., Gennert, D., Li, S., Livak, K. J., Rozenblatt-Rosen, O., Dor, Y., Regev, A., and Yanai, I. (2016). CEL-seq2: Sensitive highly-multiplexed single-cell RNA-seq. *Genome Biol*, 17(1).
- Hashimshony, T., Wagner, F., Sher, N., and Yanai, I. (2012). CEL-seq: Single-cell RNA-seq by multiplexed linear amplification. *Cell Reports*, 2(3):666–673.
- Islam, S., Zeisel, A., Joost, S., La Manno, G., Zajac, P., Kasper, M., Lönnerberg, P., and Linnarsson, S. (2013). Quantitative single-cell RNA-seq with unique molecular identifiers. *Nat Meth*, 11(2):163–166.

- Jaitin, D. A., Kenigsberg, E., Keren-Shaul, H., Elefant, N., Paul, F., Zaretsky, I., Mildner, A., Cohen, N., Jung, S., Tanay, A., and Amit, I. (2014). Massively parallel single-cell RNA-seq for marker-free decomposition of tissues into cell types. *Science*, 343(6172):776–779.
- Kharchenko, P. V., Silberstein, L., and Scadden, D. T. (2014). Bayesian approach to single-cell differential expression analysis. *Nat Meth*, 11(7):740–742.
- Kiselev, V. Y. and Hemberg, M. (2017). scmap - a tool for unsupervised projection of single cell RNA-seq data. *bioRxiv*, page 150292.
- Kiselev, V. Y., Kirschner, K., Schaub, M. T., Andrews, T., Yiu, A., Chandra, T., Natarajan, K. N., Reik, W., Barahona, M., Green, A. R., and Hemberg, M. (2017). SC3: Consensus clustering of single-cell RNA-seq data. *Nat Meth*, 14(5):483–486.
- Klein, A., Mazutis, L., Akartuna, I., Tallapragada, N., Veres, A., Li, V., Peshkin, L., Weitz, D., and Kirschner, M. (2015). Droplet barcoding for single-cell transcriptomics applied to embryonic stem cells. *Cell*, 161(5):1187–1201.
- Kolodziejczyk, A., Kim, J. K., Svensson, V., Marioni, J., and Teichmann, S. (2015). The technology and biology of single-cell RNA sequencing. *Molecular Cell*, 58(4):610–620.
- L. Lun, A. T., Bach, K., and Marioni, J. C. (2016). Pooling across cells to normalize single-cell RNA sequencing data with many zero counts. *Genome Biol*, 17(1).
- Levine, J., Simonds, E., Bendall, S., Davis, K., Amir, E.-a., Tadmor, M., Litvin, O., Fienberg, H., Jager, A., Zunder, E., Finck, R., Gedman, A., Radtke, I., Downing, J., Pe'er, D., and Nolan, G. (2015). Data-driven phenotypic dissection of AML reveals progenitor-like cells that correlate with prognosis. *Cell*, 162(1):184–197.
- Li, W. V. and Li, J. J. (2017). scImpute: Accurate and robust imputation for single cell RNA-Seq data. *bioRxiv*, page 141598.
- Macosko, E., Basu, A., Satija, R., Nemesh, J., Shekhar, K., Goldman, M., Tirosh, I., Bialas, A., Kamitaki, N., Martersteck, E., Trombetta, J., Weitz, D., Sanes, J., Shalek, A., Regev, A., and McCarroll, S. (2015). Highly parallel genome-wide expression profiling of individual cells using nanoliter droplets. *Cell*, 161(5):1202–1214.
- McCarthy, D. J., Campbell, K. R., Lun, A. T. L., and Wills, Q. F. (2017). Scater: Pre-processing, quality control, normalization and visualization of single-cell RNA-seq data in r. *Bioinformatics*, page btw777.
- Muraro, M., Dharmadhikari, G., Grün, D., Groen, N., Dielen, T., Jansen, E., van Gurp, L., Engelse, M., Carlotti, F., de Koning, E., and van Oudenaarden, A. (2016). A single-cell transcriptome atlas of the human pancreas. *Cell Systems*, 3(4):385–394.e3.
- Picelli, S., Björklund, . K., Faridani, O. R., Sagasser, S., Winberg, G., and Sandberg, R. (2013). Smart-seq2 for sensitive full-length transcriptome profiling in single cells. *Nat Meth*, 10(11):1096–1098.
- Picelli, S., Faridani, O. R., Björklund, A. K., Winberg, G., Sagasser, S., and Sandberg, R. (2014). Full-length RNA-seq from single cells using smart-seq2. *Nat. Protoc.*, 9(1):171–181.
- Regev, A., Teichmann, S., Lander, E. S., Amit, I., Benoist, C., Birney, E., Bodenmiller, B., Campbell, P., Carninci, P., Clatworthy, M., Clevers, H., Deplancke, B., Dunham, I., Eberwine, J., Eils, R., Enard, W., Farmer, A., Fugger, L., Gottgens, B., Hacohen, N., Haniffa, M., Hemberg, M., Kim, S. K., Kleinerman, P., Kriegstein, A., Lein, E., Linnarsson, S., Lundeberg, J., Majumder, P., Marioni, J., Merad, M., Mhlanga, M., Nawijn, M., Netea, M., Nolan, G., Pe'er, D., Philipakis, A., Ponting, C. P., Quake, S. R., Reik, W., Rozenblatt-Rosen, O., Sanes, J. R., Satija, R., Shumacher, T., Shalek, A. K., Shapiro, E., Sharma, P., Shin, J., Stegle, O., Stratton, M., Stubbington, M. J. T., van Oudenaarden, A., Wagner, A., Watt, F. M., Weissman, J. S., Wold, B., Xavier, R. J., Yosef, N., and Human Cell Atlas (2017). The human cell atlas. *bioRxiv*, page 121202.

- Robinson, M. D. and Oshlack, A. (2010). A scaling normalization method for differential expression analysis of RNA-seq data. *Genome Biol.*, 11(3):R25.
- Segerstolpe, ., Palasantza, A., Eliasson, P., Andersson, E.-M., Andréasson, A.-C., Sun, X., Picelli, S., Sabirsh, A., Clausen, M., Bjursell, M. K., Smith, D., Kasper, M., Ämmälä, C., and Sandberg, R. (2016). Single-cell transcriptome profiling of human pancreatic islets in health and type 2 diabetes. *Cell Metabolism*, 24(4):593–607.
- Soumillon, M., Cacchiarelli, D., Semrau, S., van Oudenaarden, A., and Mikkelsen, T. S. (2014). Characterization of directed differentiation by high-throughput single-cell RNA-Seq. *bioRxiv*, page 003236.
- Stegle, O., Teichmann, S. A., and Marioni, J. C. (2015). Computational and analytical challenges in single-cell transcriptomics. *Nat Rev Genet*, 16(3):133–145.
- Svensson, V., Natarajan, K. N., Ly, L.-H., Miragaia, R. J., Labalette, C., Macaulay, I. C., Cvejic, A., and Teichmann, S. A. (2017). Power analysis of single-cell RNA-sequencing experiments. *Nat Meth*, 14(4):381–387.
- Tang, F., Barbacioru, C., Wang, Y., Nordman, E., Lee, C., Xu, N., Wang, X., Bodeau, J., Tuch, B. B., Siddiqui, A., Lao, K., and Surani, M. A. (2009). mRNA-seq whole-transcriptome analysis of a single cell. *Nat Meth*, 6(5):377–382.
- Tung, P.-Y., Blischak, J. D., Hsiao, C. J., Knowles, D. A., Burnett, J. E., Pritchard, J. K., and Gilad, Y. (2017). Batch effects and the effective design of single-cell gene expression studies. *Sci. Rep.*, 7:39921.
- van Dijk, D., Nainys, J., Sharma, R., Kathail, P., Carr, A. J., Moon, K. R., Mazutis, L., Wolf, G., Krishnaswamy, S., and Pe'er, D. (2017). MAGIC: A diffusion-based imputation method reveals gene-gene interactions in single-cell RNA-sequencing data. *bioRxiv*, page 111591.
- Welch, J. D., Hartemink, A. J., and Prins, J. F. (2016). SLICER: Inferring branched, nonlinear cellular trajectories from single cell RNA-seq data. *Genome Biol.*, 17(1).
- Wickham, H. (2014). Tidy data. *J Stat Softw*, 59(10).
- Xu, C. and Su, Z. (2015). Identification of cell types from single-cell transcriptomes using a novel clustering method. *Bioinformatics*, 31(12):1974–1980.
- Ziegenhain, C., Vieth, B., Parekh, S., Reinius, B., Guillaumet-Adkins, A., Smets, M., Leonhardt, H., Heyn, H., Hellmann, I., and Enard, W. (2017). Comparative analysis of single-cell RNA sequencing methods. *Molecular Cell*, 65(4):631–643.e4.
- žurauskienė, J. and Yau, C. (2016). pcaReduce: Hierarchical clustering of single cell transcriptional profiles. *BMC Bioinformatics*, 17(1).