Analyzing single-cell RNA-seq data containing read counts

Aaron T. L. Lun[1]***, Davis J. McCarthy***[2,3] ***and John C. Marioni***[1,2,4]

[1]Cancer Research UK Cambridge Institute, Li Ka Shing Centre, Robinson Way, Cambridge CB2 0RE, United Kingdom
[2]EMBL European Bioinformatics Institute, Wellcome Genome Campus, Hinxton, Cambridge CB10 1SD, United Kingdom
[3]St Vincent's Institute of Medical Research, 41 Victoria Parade, Fitzroy, Victoria 3065, Australia
[4]Wellcome Trust Sanger Institute, Wellcome Genome Campus, Hinxton, Cambridge CB10 1SA, United Kingdom

2019-01-09

1 Overview

In this workflow, we use a relatively simple dataset (Lun et al. 2017) to introduce most of the concepts of scRNA-seq data analysis. This dataset contains two plates of 416B cells (an immortalized mouse myeloid progenitor cell line), processed using the Smart-seq2 protocol (Picelli et al. 2014). A constant amount of spike-in RNA from the External RNA Controls Consortium (ERCC) was also added to each cell's lysate prior to library preparation. High-throughput sequencing was performed and the expression of each gene was quantified by counting the total number of reads mapped to its exonic regions. Similarly, the quantity of each spike-in transcript was measured by counting the number of reads mapped to the spike-in reference sequences.

Counts for all genes/transcripts in each cell are available from ArrayExpress using the accession number E-MTAB-5522 (https://www.ebi.ac.uk/arrayexpress/experiments/E-MTAB-5522). We download both the count tables (in the "processed files") as well as the metadata file using the *BiocFileCache (https://bioconductor.org/packages/3.8/BiocFileCache)* package. This saves the files to a local cache (`raw_data`) and avoids re-downloading them if they are already present.

```
library(BiocFileCache)
bfc <- BiocFileCache("raw_data", ask = FALSE)
lun.zip <- bfcrpath(bfc,
    file.path("https://www.ebi.ac.uk/arrayexpress/files",
        "E-MTAB-5522/E-MTAB-5522.processed.1.zip"))
lun.sdrf <- bfcrpath(bfc,
    file.path("https://www.ebi.ac.uk/arrayexpress/files",
        "E-MTAB-5522/E-MTAB-5522.sdrf.txt"))
unzip(lun.zip)
```

2 Setting up the data

2.1 Loading in the count matrix

Our first task is to load the count matrices into memory. One matrix was generated for each plate of cells used in the study. In each matrix, each row represents an endogenous gene or a spike-in transcript, and each column represents a cell. Subsequently, the count in each entry of the matrix represents the number of reads mapped to a particular gene/transcript in a particular cell.

```
plate1 <- read.delim("counts_Calero_20160113.tsv",
    header=TRUE, row.names=1, check.names=FALSE)
plate2 <- read.delim("counts_Calero_20160325.tsv",
    header=TRUE, row.names=1, check.names=FALSE)

gene.lengths <- plate1$Length # First column is the gene len
gth.
plate1 <- as.matrix(plate1[,-1]) # Discarding gene length (a
s it is not a cell).
plate2 <- as.matrix(plate2[,-1])
rbind(Plate1=dim(plate1), Plate2=dim(plate2))
```

```
##         [,1] [,2]
## Plate1 46703   96
## Plate2 46703   96
```

We combine the two matrices into a single object for further processing. This is done after verifying that the genes are in the same order between the two matrices.

```
stopifnot(identical(rownames(plate1), rownames(plate2)))
all.counts <- cbind(plate1, plate2)
```

For convenience, the count matrix is stored in a `SingleCellExperiment` object from the *SingleCellExperiment (https://bioconductor.org/packages/3.8/SingleCellExperiment)* package. This allows different types of row- and column-level metadata to be stored alongside the counts for synchronized manipulation throughout the workflow.

```
library(SingleCellExperiment)
sce <- SingleCellExperiment(list(counts=all.counts))
rowData(sce)$GeneLength <- gene.lengths
sce
```

```
## class: SingleCellExperiment
## dim: 46703 192
## metadata(0):
## assays(1): counts
## rownames(46703): ENSMUSG00000102693 ENSMUSG00000064842
... SIRV7
##   CBFB-MYH11-mcherry
## rowData names(1): GeneLength
## colnames(192): SLX-9555.N701_S502.C89V9ANXX.s_1.r_1
##   SLX-9555.N701_S503.C89V9ANXX.s_1.r_1 ...
##   SLX-11312.N712_S508.H5H5YBBXX.s_8.r_1
##   SLX-11312.N712_S517.H5H5YBBXX.s_8.r_1
## colData names(0):
## reducedDimNames(0):
## spikeNames(0):
```

We identify the rows corresponding to ERCC spike-in transcripts

from the row names. We store this information in the `SingleCellExperiment` object for future use. This is necessary as spike-ins require special treatment in downstream steps such as normalization.

```
isSpike(sce, "ERCC") <- grepl("^ERCC", rownames(sce))
summary(isSpike(sce, "ERCC"))
```

```
##    Mode   FALSE    TRUE
## logical   46611      92
```

This dataset is slightly unusual in that it contains information from another set of spike-in transcripts, the Spike-In RNA Variants (SIRV) set. For simplicity, we will only use the ERCC spike-ins in this analysis. Thus, we must remove the rows corresponding to the SIRV transcripts prior to further analysis, which can be done simply by subsetting the `SingleCellExperiment` object.

```
is.sirv <- grepl("^SIRV", rownames(sce))
sce <- sce[!is.sirv,]
summary(is.sirv)
```

```
##    Mode   FALSE    TRUE
## logical   46696       7
```

Comments from Aaron:

- Some feature-counting tools will report mapping statistics in the count matrix (e.g., the number of unaligned or unassigned reads). While these values can be useful for quality control, they would be misleading if treated as gene expression values. Thus, they should be removed (or at least moved to the `colData`) prior to further analyses.
- Be aware of using the `^ERCC` regular expression for human data where the row names of the count matrix are gene symbols. An ERCC gene family actually exists in human annotation, so this would result in incorrect identification of genes as spike-in transcripts. This problem can be avoided by publishing count matrices with standard identifiers (e.g., Ensembl, Entrez).

2.2 Incorporating cell-based annotation

We load in the metadata for each library/cell from the `sdrf.txt` file. It is important to check that the rows of the metadata table are in the same order as the columns of the count matrix. Otherwise, incorrect metadata will be assigned to each cell.

```
metadata <- read.delim(lun.sdrf, check.names=FALSE, header=T
RUE)
m <- match(colnames(sce), metadata[["Source Name"]]) # Enfor
cing identical order.
stopifnot(all(!is.na(m))) # Checking that nothing's missing.
metadata <- metadata[m,]
head(colnames(metadata))
```

```
## [1] "Source Name"                "Comment[ENA_SAMPLE]"
## [3] "Comment[BioSD_SAMPLE]"      "Characteristics[organis
m]"
## [5] "Characteristics[cell line]" "Characteristics[cell ty
pe]"
```

We only retain relevant metadata fields to avoid storing unnecessary information in the `colData` of the `SingleCellExperiment` object. In particular, we keep the plate of origin (i.e., `block`) and phenotype of each cell. The second field is relevant as all of the cells contain a *CBFB-MYH11* oncogene, but the expression of this oncogene is only induced in a subset of the cells.

```
colData(sce)$Plate <- factor(metadata[["Factor Value[bloc
k]"]])
pheno <- metadata[["Factor Value[phenotype]"]]
levels(pheno) <- c("induced", "control")
colData(sce)$Oncogene <- pheno
table(colData(sce)$Oncogene, colData(sce)$Plate)
```

```
##
##           20160113 20160325
##   induced       48       48
##   control       48       48
```

2.3 Incorporating gene-based annotation

Feature-counting tools typically report genes in terms of standard identifiers from Ensembl or Entrez. These identifiers are used as they are unambiguous and highly stable. However, they are difficult to interpret compared to the gene symbols which are more commonly used in the literature. Given the Ensembl identifiers, we obtain the corresponding gene symbols using annotation packages like *org.Mm.eg.db (https://bioconductor.org/packages/3.8/org.Mm.eg.db)*.

```
library(org.Mm.eg.db)
symb <- mapIds(org.Mm.eg.db, keys=rownames(sce), keytype="EN
SEMBL", column="SYMBOL")
rowData(sce)$ENSEMBL <- rownames(sce)
rowData(sce)$SYMBOL <- symb
head(rowData(sce))
```

```
## DataFrame with 6 rows and 3 columns
##                     GeneLength            ENSEMBL      SYM
BOL
##                      <integer>        <character> <charact
er>
## ENSMUSG00000102693       1070 ENSMUSG00000102693
NA
## ENSMUSG00000064842        110 ENSMUSG00000064842
NA
## ENSMUSG00000051951       6094 ENSMUSG00000051951        X
kr4
## ENSMUSG00000102851        480 ENSMUSG00000102851
NA
## ENSMUSG00000103377       2819 ENSMUSG00000103377
NA
## ENSMUSG00000104017       2233 ENSMUSG00000104017
NA
```

It is often desirable to rename the row names of `sce` to the gene symbols, as these are easier to interpret. However, this requires some work to account for missing and duplicate symbols. The code below will replace missing symbols with the Ensembl identifier and concatenate duplicated symbols with the (unique) Ensembl identifiers.

```
new.names <- rowData(sce)$SYMBOL
missing.name <- is.na(new.names)
new.names[missing.name] <- rowData(sce)$ENSEMBL[missing.nam
e]
dup.name <- new.names %in% new.names[duplicated(new.names)]
new.names[dup.name] <- paste0(new.names, "_", rowData(sce)$E
NSEMBL)[dup.name]
rownames(sce) <- new.names
head(rownames(sce))
```

```
## [1] "ENSMUSG00000102693" "ENSMUSG00000064842" "Xkr4"
## [4] "ENSMUSG00000102851" "ENSMUSG00000103377" "ENSMUSG000
00104017"
```

We also determine the chromosomal location for each gene using the *TxDb.Mmusculus.UCSC.mm10.ensGene (https://bioconductor.org/packages/3.8/TxDb.Mmusculus.UCSC.mm10.ensGene)* package. This will be useful later as several quality control metrics will be computed from rows corresponding to mitochondrial genes.

```
library(TxDb.Mmusculus.UCSC.mm10.ensGene)
location <- mapIds(TxDb.Mmusculus.UCSC.mm10.ensGene, keys=ro
wData(sce)$ENSEMBL,
    column="CDSCHROM", keytype="GENEID")
rowData(sce)$CHR <- location
summary(location=="chrM")
```

```
##    Mode   FALSE    TRUE    NA's
## logical   22428      13   24255
```

Alternatively, annotation from BioMart resources can be directly added to the object using the `getBMFeatureAnnos` function from *scater (https://bioconductor.org/packages/3.8/scater)*. This may be more convenient than the approach shown above, but depends on an available internet connection to the BioMart databases.

3 Quality control on the cells

3.1 Defining the quality control metrics

Low-quality cells need to be removed to ensure that technical effects do not distort downstream analysis results. We use several quality control (QC) metrics:

- The library size is defined as the total sum of counts across all features, i.e., genes and spike-in transcripts. Cells with small library sizes are of low quality as the RNA has not been efficiently captured (i.e., converted into cDNA and amplified) during library preparation.
- The number of expressed features in each cell is defined as the number of features with non-zero counts for that cell. Any cell with very few expressed genes is likely to be of poor quality as the diverse transcript population has not been successfully captured.
- The proportion of reads mapped to spike-in transcripts is calculated relative to the library size for each cell. High proportions are indicative of poor-quality cells, where endogenous RNA has been lost during processing (e.g., due to cell lysis or RNA degradation). The same amount of spike-in RNA to each cell, so an enrichment in spike-in counts is symptomatic of loss of endogenous RNA.
- In the absence of spike-in transcripts, the proportion of reads mapped to genes in the mitochondrial genome can also be used. High proportions are indicative of poor-quality cells (Islam et al. 2014; Ilicic et al. 2016), possibly because of loss of cytoplasmic RNA from perforated cells. The reasoning is that mitochondria are larger than individual transcript molecules and less likely to escape through tears in the cell

membrane.

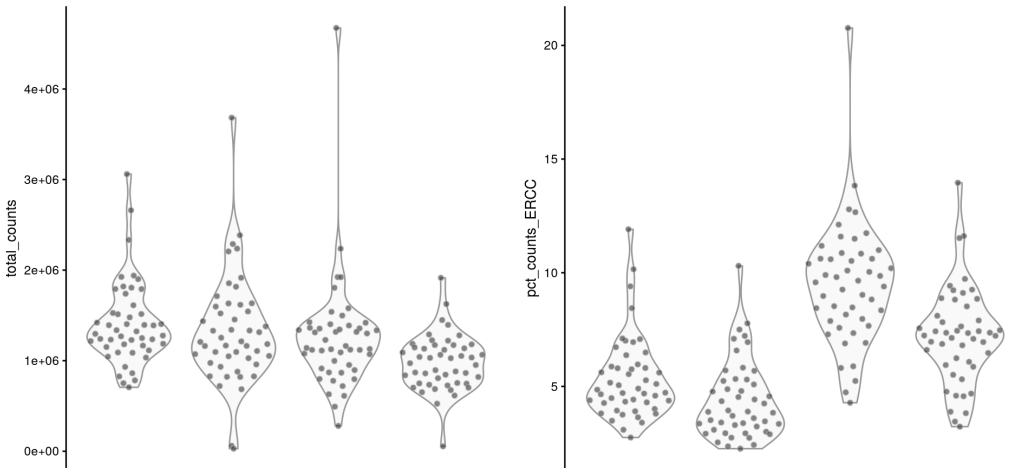
For each cell, we calculate these quality control metrics using the `calculateQCMetrics` function from the *scater (https://bioconductor.org/packages/3.8/scater)* package (McCarthy et al. 2017). These are stored in the row- and column-wise metadata of the `SingleCellExperiment` for future reference.

```
library(scater)
mito <- which(rowData(sce)$CHR=="chrM")
sce <- calculateQCMetrics(sce, feature_controls=list(Mt=mit
o))
head(colnames(colData(sce)), 10)
```

```
##  [1] "Plate"                           "Oncogene"
##  [3] "is_cell_control"                 "total_features_by_
counts"
##  [5] "log10_total_features_by_counts" "total_counts"
##  [7] "log10_total_counts"              "pct_counts_in_top_
50_features"
##  [9] "pct_counts_in_top_100_features" "pct_counts_in_top_
200_features"
```

The distributions of these metrics are shown in Figure 1, stratified by oncogene induction status and plate of origin. The aim is to remove putative low-quality cells that have low library sizes, low numbers of expressed features, and high spike-in (or mitochondrial) proportions. Such cells can interfere with downstream analyses, e.g., by forming distinct clusters that complicate interpretation of the results.

```
sce$PlateOnco <- paste0(sce$Oncogene, ".", sce$Plate)
multiplot(
    plotColData(sce, y="total_counts", x="PlateOnco"),
    plotColData(sce, y="total_features_by_counts", x="PlateO
nco"),
    plotColData(sce, y="pct_counts_ERCC", x="PlateOnco"),
    plotColData(sce, y="pct_counts_Mt", x="PlateOnco"),
    cols=2)
```



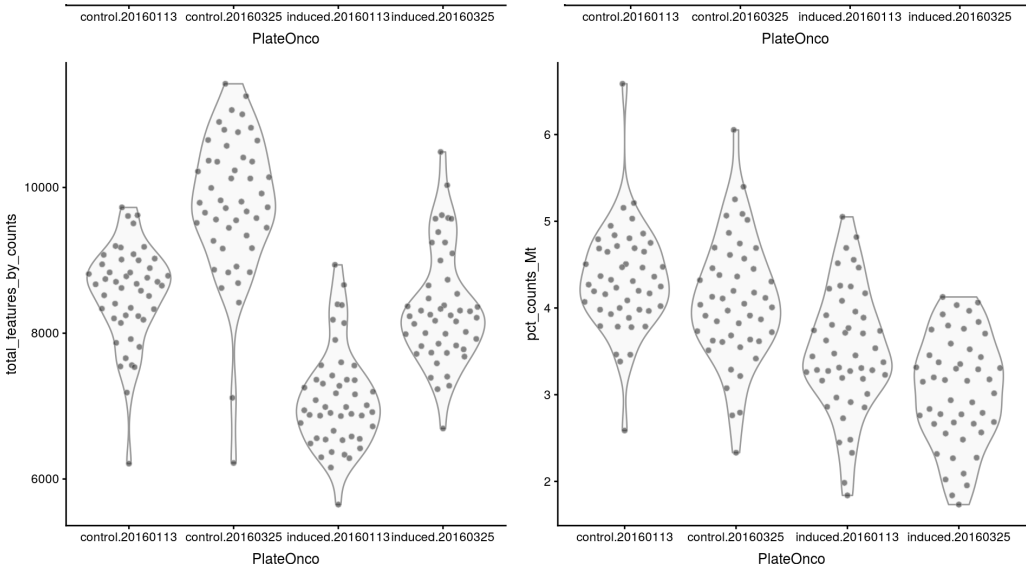


Figure 1: Distributions of various QC metrics for all cells in the 416B dataset. This includes the library sizes, number of expressed genes, and proportion of reads mapped to spike-in transcripts or mitochondrial genes.

It is also valuable to examine how the QC metrics behave with respect to each other (Figure 2). Generally, they will be in rough agreement, i.e., cells with low total counts will also have low numbers of expressed features and high ERCC/mitochondrial proportions. Clear discrepancies may correspond to technical differences between batches of cells (see below) or genuine biological differences in RNA content.

```
par(mfrow=c(1,3))
plot(sce$total_features_by_counts, sce$total_counts/1e6, xla
b="Number of expressed genes",
    ylab="Library size (millions)")
plot(sce$total_features_by_counts, sce$pct_counts_ERCC, xlab
="Number of expressed genes",
    ylab="ERCC proportion (%)")
plot(sce$total_features_by_counts, sce$pct_counts_Mt, xlab="
Number of expressed genes",
    ylab="Mitochondrial proportion (%)")
```

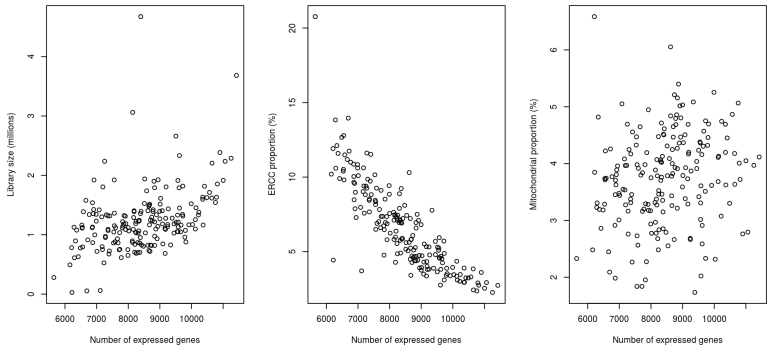


Figure 2: Behaviour of each QC metric compared to the total number of expressed features. Each point represents a cell in the 416B dataset.

3.2 Identifying outliers for each metric

Picking a threshold for these metrics is not straightforward as their absolute values depend on the experimental protocol. For example, sequencing to greater depth will lead to more reads and more expressed features, regardless of the quality of the cells. Similarly, using more spike-in RNA in the protocol will result in higher spike-in proportions. To obtain an adaptive threshold, we assume that most of the dataset consists of high-quality cells, and identify cells that are outliers for the various QC metrics.

Outliers are defined based on the median absolute deviation (MADs) from the median value of each metric across all cells. We remove cells with log-library sizes that are more than 3 MADs below the median log-library size. A log-transformation improves resolution at small values, especially when the MAD of the raw values is comparable to or greater than the median. We also remove cells where the log-transformed number of expressed genes is 3 MADs below the median value.

```
libsize.drop <- isOutlier(sce$total_counts, nmads=3, type="l
ower",
    log=TRUE, batch=sce$PlateOnco)
feature.drop <- isOutlier(sce$total_features_by_counts, nmad
s=3, type="lower",
    log=TRUE, batch=sce$PlateOnco)
```

The `batch=` argument ensures that outliers are identified *within* each level of the specified plate/oncogene factor. This allows `isOutlier()` to accommodate systematic differences in the QC metrics across plates (Figure 1), which can arise due to technical differences in processing (e.g., differences in sequencing depth) rather than any changes in quality. The same reasoning applies to the oncogene induction status, where induced cells may have naturally fewer expressed genes for biological reasons. Failing to account for these systematic differences would inflate the MAD estimate and compromise the removal of low-quality cells.

We identify outliers for the proportion-based metrics in a similar manner. Here, no transformation is required as we are identifying large outliers, for which the distinction should be fairly clear on the raw scale. We do not need to use the mitochondrial proportions as we already have the spike-in proportions (which serve a similar purpose) for this dataset. This avoids potential issues arising from genuine differences in mitochondrial content between cell types that may confound outlier identification.

```
spike.drop <- isOutlier(sce$pct_counts_ERCC, nmads=3, type="
higher",
    batch=sce$PlateOnco)
```

Subsetting by column will retain only the high-quality cells that

pass each filter described above. We examine the number of cells removed by each filter as well as the total number of retained cells. Removal of a substantial proportion of cells (> 10%) may be indicative of an overall issue with data quality.

```
keep <- !(libsize.drop | feature.drop | spike.drop)
data.frame(ByLibSize=sum(libsize.drop), ByFeature=sum(featur
e.drop),
    BySpike=sum(spike.drop), Remaining=sum(keep))
```

```
##   ByLibSize ByFeature BySpike Remaining
## 1         5         4       6       183
```

We then subset the `SingleCellExperiment` object to retain only the putative high-quality cells. We also save the original object to file for later use.

```
sce$PassQC <- keep
saveRDS(sce, file="416B_preQC.rds")
sce <- sce[,keep]
dim(sce)
```

```
## [1] 46696   183
```

Comments from Aaron:

- See this section (https://bioconductor.org/packages/3.8/simpleSingleCell/vignettes/xtra-1-qc.html#assumptions-of-outlier-identification) for a more detailed discussion of the assumptions underlying the outlier-based detection of low-quality cells.
- `isOutlier()` will also return the exact filter thresholds for each metric (within each batch, if `batch=` is specified). These may be useful for checking whether the automatically selected thresholds are appropriate.

  ```
  attr(libsize.drop, "thresholds")
  ```

  ```
  ##        control.20160113 control.20160325 induced.201
  60113 induced.20160325
  ## lower           5.82883         5.622679         5.6
  98215         5.664173
  ## higher              Inf              Inf
  Inf              Inf
  ```

  ```
  attr(spike.drop, "thresholds")
  ```

```
##        control.20160113 control.20160325 induced.201
60113 induced.20160325
## lower              -Inf             -Inf
-Inf             -Inf
## higher           8.99581         8.105749         15.
50477         12.71858
```

4 Classification of cell cycle phase

We use the prediction method described by Scialdone et al. (2015) to classify cells into cell cycle phases based on the gene expression data. Using a training dataset, the sign of the difference in expression between two genes was computed for each pair of genes. Pairs with changes in the sign across cell cycle phases were chosen as markers. Cells in a test dataset can then be classified into the appropriate phase, based on whether the observed sign for each marker pair is consistent with one phase or another.

This approach is implemented in the `cyclone` function from the *scran (https://bioconductor.org/packages/3.8/scran)* package. The package contains a pre-trained set of marker pairs for mouse data, which we can load in the the `readRDS` function. We use the Ensembl identifiers for each gene in our dataset to match up with the names in the pre-trained set of gene pairs.

```
set.seed(100)
library(scran)
mm.pairs <- readRDS(system.file("exdata", "mouse_cycle_marke
rs.rds",
    package="scran"))
assignments <- cyclone(sce, mm.pairs, gene.names=rowData(sc
e)$ENSEMBL)
```

The `cyclone` result for each cell in the HSC dataset is shown in Figure 3. Each cell is assigned a score for each phase, with a higher score corresponding to a higher probability that the cell is in that phase. We focus on the G1 and G2/M scores as these are the most informative for classification.

```
plot(assignments$score$G1, assignments$score$G2M,
    xlab="G1 score", ylab="G2/M score", pch=16)
```

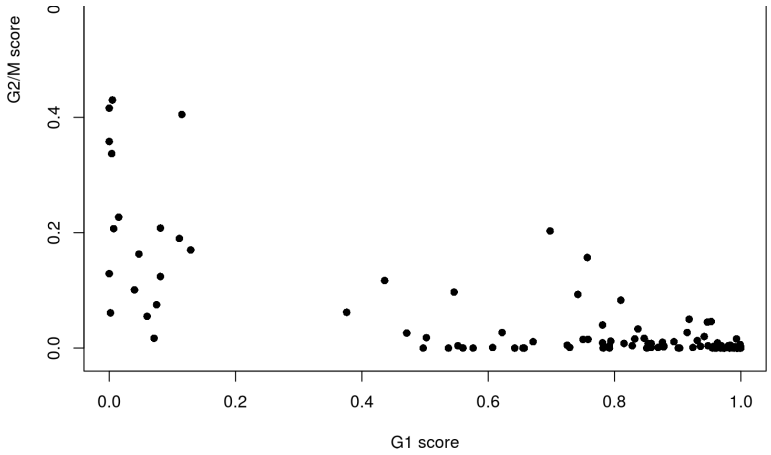


Figure 3: Cell cycle phase scores from applying the pair-based classifier on the 416B dataset. Each point represents a cell, plotted according to its scores for G1 and G2/M phases.

Cells are classified as being in G1 phase if the G1 score is above 0.5 and greater than the G2/M score; in G2/M phase if the G2/M score is above 0.5 and greater than the G1 score; and in S phase if neither score is above 0.5. Here, the vast majority of cells are classified as being in G1 phase. We save these assignments into the `SingleCellExperiment` object for later use.

```
sce$phases <- assignments$phases
table(sce$phases)
```

```
##
##  G1 G2M   S
##  99  62  22
```

Pre-trained classifiers are available in *scran (https://bioconductor.org/packages/3.8/scran)* for human and mouse data. While the mouse classifier used here was trained on data from embryonic stem cells, it is still accurate for other cell types (Scialdone et al. 2015). This may be due to the conservation of the transcriptional program associated with the cell cycle (Bertoli, Skotheim, and Bruin 2013; Conboy et al. 2007). The pair-based method is also a non-parametric procedure that is robust to most technical differences between datasets.

Comments from Aaron:

- To remove confounding effects due to cell cycle phase, we can filter the cells to only retain those in a particular phase (usually G1) for downstream analysis. Alternatively, if a non-negligible number of cells are in other phases, we can use the assigned phase as a blocking factor. This protects against cell cycle effects without discarding information, and will be discussed later in more detail.
- The classifier may not be accurate for data that are substantially different from those used in the training set,

e.g., due to the use of a different protocol. In such cases, users can construct a custom classifier from their own training data using the `sandbag` function. This will also be necessary for other model organisms where pre-trained classifiers are not available.

- Do not filter out low-abundance genes before applying `cyclone`. Even if a gene is not expressed in *any* cell, it may still be useful for classification if it is phase-specific. Its lack of expression relative to other genes will still yield informative pairs, and filtering them out would reduce power.

5 Examining gene-level expression metrics

5.1 Inspecting the most highly expressed genes

We examine the identities of the most highly expressed genes (Figure 4). This should generally be dominated by constitutively expressed transcripts, such as those for ribosomal or mitochondrial proteins. The presence of other classes of features may be cause for concern if they are not consistent with expected biology. For example, a top set containing many spike-in transcripts suggests that too much spike-in RNA was added during library preparation, while the absence of ribosomal proteins and/or the presence of their pseudogenes are indicative of suboptimal alignment.

```
fontsize <- theme(axis.text=element_text(size=12), axis.titl
e=element_text(size=16))
plotHighestExprs(sce, n=50) + fontsize
```

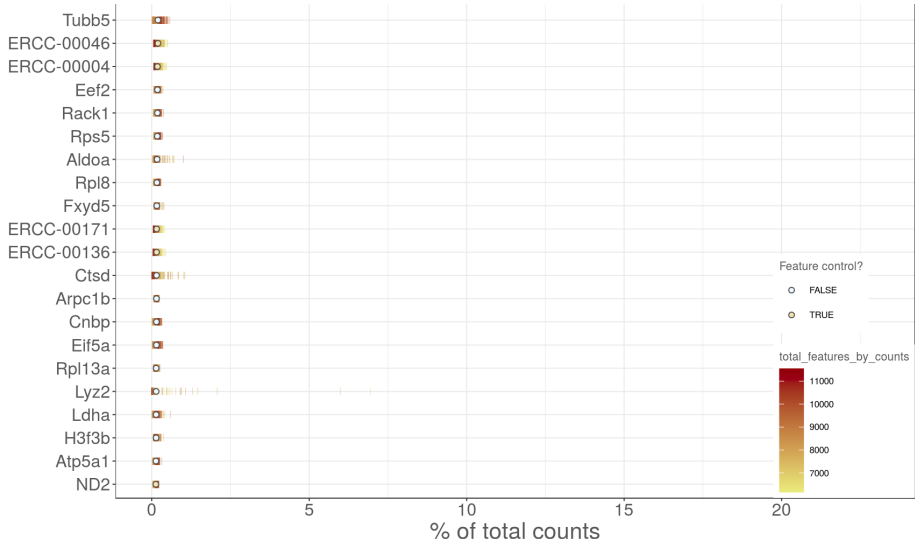


Figure 4: Percentage of total counts assigned to the top 50 most highly-abundant features in the 416B dataset. For each feature, each bar represents the percentage assigned to that feature for a single cell, while the circle represents the average across all cells. Bars are coloured by the total number of expressed features in each cell, while circles are coloured according to whether the feature is labelled as a control feature.

5.2 Filtering out low-abundance genes

Low-abundance genes are problematic as zero or near-zero counts do not contain much information for reliable statistical inference (Bourgon, Gentleman, and Huber 2010). These genes typically do not provide enough evidence to reject the null hypothesis during testing, yet they still increase the severity of the multiple testing correction. In addition, the discreteness of the counts may interfere with statistical procedures, e.g., by compromising the accuracy of continuous approximations. Thus, low-abundance genes are often removed in many RNA-seq analysis pipelines before the application of downstream methods.

The "optimal" choice of filtering strategy depends on the downstream application. A more aggressive filter is usually required to remove discreteness (e.g., for normalization) compared to that required for removing underpowered tests. For hypothesis testing, the filter statistic should also be independent of the test statistic under the null hypothesis. Thus, we (or the relevant function) will filter at each step as needed, rather than applying a single filter for the entire analysis.

Several metrics can be used to define low-abundance genes. The most obvious is the average count for each gene, computed across all cells in the dataset. We calculate this using the `calcAverage()` function, which also performs some adjustment for library size differences between cells. We typically observe a peak of moderately expressed genes following a plateau of lowly expressed genes (Figure 5).

```
ave.counts <- calcAverage(sce, use_size_factors=FALSE)
hist(log10(ave.counts), breaks=100, main="", col="grey80",
    xlab=expression(Log[10]~"average count"))
```

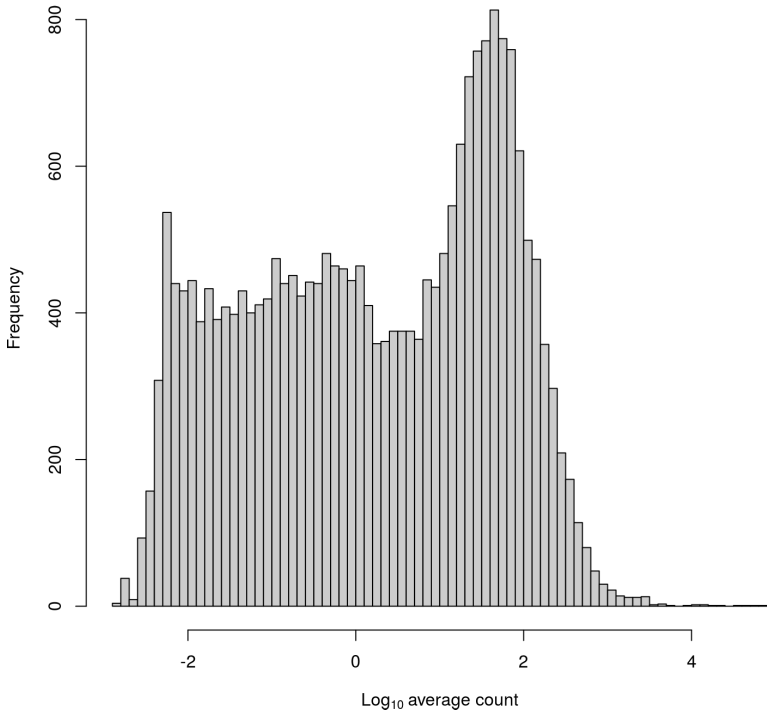


Figure 5: Histogram of log-average counts for all genes in the 416B dataset.

A minimum threshold can be applied to this value to filter out genes that are lowly expressed. The example below demonstrates how we *could* remove genes with average counts less than 1. The number of `TRUE` values in `demo.keep` corresponds to the number of retained rows/genes after filtering.

```
demo.keep <- ave.counts >= 1
filtered.sce <- sce[demo.keep,]
summary(demo.keep)
```

```
##    Mode   FALSE    TRUE
## logical   33490   13206
```

We also examine the number of cells that express each gene. This is closely related to the average count for most genes, as expression in many cells will result in a higher average (Figure 6). Genes expressed in very few cells are often uninteresting as they are driven by amplification artifacts (though they may also also arise from rare populations). We could then remove genes that are expressed in fewer than *n* cells.

```
num.cells <- nexprs(sce, byrow=TRUE)
smoothScatter(log10(ave.counts), num.cells, ylab="Number of
cells",
    xlab=expression(Log[10]~"average count"))
```

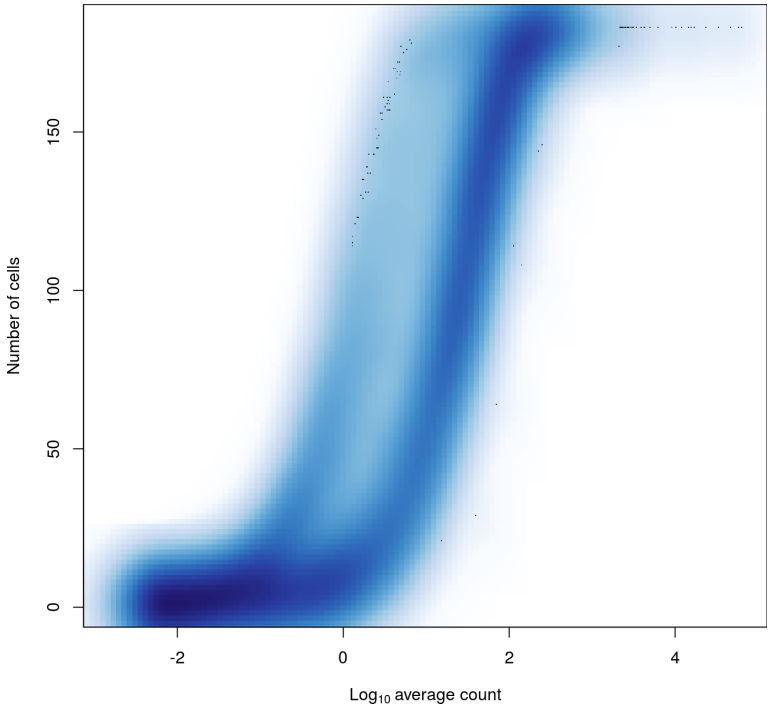


Figure 6: The number of cells expressing each gene in the 416B dataset, plotted against the log-average count. Intensity of colour corresponds to the number of genes at any given location.

As mentioned above, we will apply these filters at each step rather than applying them globally by subsetting `sce`. This ensures that the most appropriate filter is used in each application. Nonetheless, we remove genes that are not expressed in any cell to reduce computational work in downstream steps. Such genes provide no information and would be removed by any filtering strategy.

```
to.keep <- num.cells > 0
sce <- sce[to.keep,]
summary(to.keep)
```

```
##    Mode   FALSE    TRUE
## logical   22833   23863
```

6 Normalization of cell-specific biases

6.1 Using the deconvolution method to deal with zero counts

Read counts are subject to differences in capture efficiency and sequencing depth between cells (Stegle, Teichmann, and Marioni 2015). Normalization is required to eliminate these cell-specific biases prior to downstream quantitative analyses. This is often done by assuming that most genes are not differentially expressed (DE) between cells. Any systematic difference in count size across the non-DE majority of genes between two cells is assumed to represent bias and is removed by scaling. More specifically, “size factors” are calculated that represent the extent to which counts should be scaled in each library.

Size factors can be computed with several different approaches, e.g., using the `estimateSizeFactorsFromMatrix` function in the *DESeq2 (https://bioconductor.org/packages/3.8/DESeq2)* package (Anders and Huber 2010; Love, Huber, and Anders 2014), or with the `calcNormFactors` function (Robinson and Oshlack 2010) in the *edgeR (https://bioconductor.org/packages/3.8/edgeR)* package. However, single-cell data can be problematic for these bulk data-based methods due to the dominance of low and zero counts. To overcome this, we pool counts from many cells to increase the count size for accurate size factor estimation (Lun, Bach, and Marioni 2016). Pool-based size factors are then “deconvolved” into cell-based factors for cell-specific normalization.

```
sce <- computeSumFactors(sce)
summary(sizeFactors(sce))
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##  0.3464  0.7117  0.9098  1.0000  1.1396  3.5695
```

The size factors are well-correlated with the library sizes for all cells (Figure 7). This suggests that most of the systematic differences between cells are driven by differences in capture efficiency or sequencing depth. Any DE between cells would yield a non-linear trend between the total count and size factor, and/or increased scatter around the trend. We observe some evidence of this after oncogene induction, where the size factors after induction are systematically lower. This is consistent with composition biases (Robinson and Oshlack 2010) introduced by upregulation of genes after induction.

```
plot(sce$total_counts/1e6, sizeFactors(sce), log="xy",
    xlab="Library size (millions)", ylab="Size factor",
    col=c("red", "black")[sce$Oncogene], pch=16)
legend("bottomright", col=c("red", "black"), pch=16, cex=1.
2,
    legend=levels(sce$Oncogene))
```

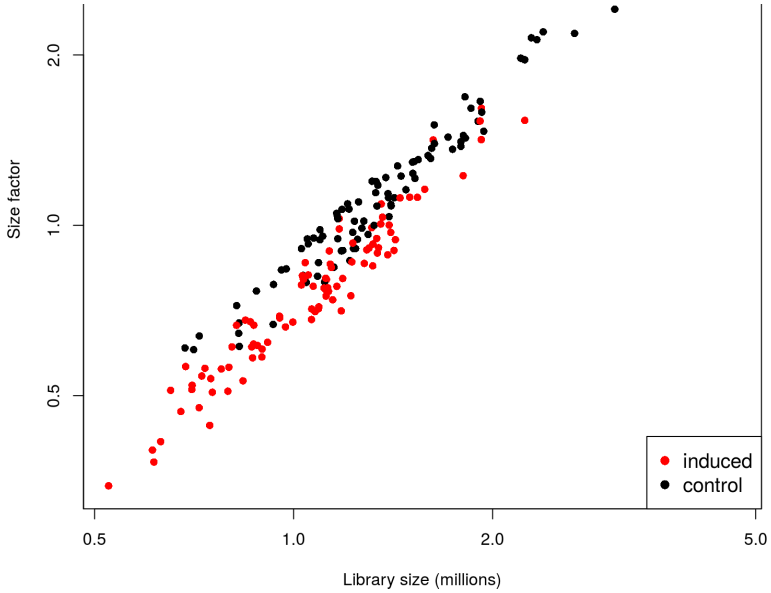


Figure 7: Size factors from deconvolution, plotted against library sizes for all cells in the 416B dataset. Axes are shown on a log-scale. Wild-type cells are shown in black and oncogene-induced cells are shown in red.

Comments from Aaron:

- While the deconvolution approach is robust to the high frequency of zeroes in scRNA-seq data, it will eventually fail if too many counts are zero. This manifests as negative size factors that are obviously nonsensical. To avoid this, the `computeSumFactors` function will automatically remove low-abundance genes prior to the calculation of size factors. Genes with a library size-adjusted average count below a specified threshold (`min.mean`) are ignored.
  - For read count data, the default value of 1 is usually satisfactory. For UMI data, counts are lower so a threshold of 0.1 is recommended.
  - While the library size-adjusted average is not entirely independent of the bias (Bourgon, Gentleman, and Huber 2010), this is a better filter statistic than the sample mean count. The latter would enrich for genes that are upregulated in cells with large library sizes, resulting in inflated size factor estimates for those cells.
- Cell-based QC should always be performed prior to normalization, to remove cells with very low numbers of expressed genes. Otherwise, `computeSumFactors()` may yield negative size factors for low-quality cells. This is because too many zeroes are present in these cells, reducing the effectiveness of pooling to eliminate zeroes. Indeed, for some low-quality cells, the effect of cell damage on the transcriptome may already violate the non-DE assumption.
- The `sizes` argument can be used to specify the number of pool sizes to use to compute the size factors. More `sizes` yields more precise estimates at the cost of some

computational time and memory. In general, all `sizes` should not above 20 cells to ensure that there are sufficient non-zero expression values in each pool. The total number of cells should also be at least 100 for effective pooling.

- For highly heterogeneous datasets, it is advisable to perform a rough clustering of the cells. This can be done with the `quickCluster` function and the results passed to `computeSumFactors` via the `cluster` argument. Cells in each cluster are normalized separately, and the size factors are rescaled to be comparable across clusters. This avoids the need to assume that most genes are non-DE across the entire population - only a non-DE majority is required between pairs of clusters. We demonstrate this approach later with a larger dataset in the next workflow (https://bioconductor.org/packages/3.8/simpleSingleCell/vignettes/work-2-umis.html#6_normalization_of_cell-specific_biases).

6.2 Computing separate size factors for spike-in transcripts

Size factors computed from the counts for endogenous genes are usually not appropriate for normalizing the counts for spike-in transcripts. Consider an experiment without library quantification, i.e., the amount of cDNA from each library is *not* equalized prior to pooling and multiplexed sequencing. Here, cells containing more RNA have greater counts for endogenous genes and thus larger size factors to scale down those counts. However, the same amount of spike-in RNA is added to each cell during library preparation. This means that the counts for spike-in transcripts are not subject to the effects of RNA content. Attempting to normalize the spike-in counts with the gene-based size factors will lead to over-normalization and incorrect quantification of expression. Similar reasoning applies in cases where library quantification is performed. For a constant total amount of cDNA, any increases in endogenous RNA content will suppress the coverage of spike-in transcripts. As a result, the bias in the spike-in counts will be opposite to that captured by the gene-based size factor.

To ensure normalization is performed correctly, we compute a separate set of size factors for the spike-in set. For each cell, the spike-in-specific size factor is defined as the total count across all transcripts in the spike-in set. This assumes that none of the spike-in transcripts are differentially expressed, which is reasonable given that the same amount and composition of spike-in RNA should have been added to each cell (Lun et al. 2017). (See below for a more detailed discussion on spike-in normalization.) These size factors are stored in a separate field of the `SingleCellExperiment` object by setting `general.use=FALSE`

in `computeSpikeFactors`. This ensures that they will only be used with the spike-in transcripts but not the endogenous genes.

```
sce <- computeSpikeFactors(sce, type="ERCC", general.use=FAL
SE)
```

6.3 Applying the size factors to normalize gene expression

The count data are used to compute normalized log-expression values for use in downstream analyses. Each value is defined as the $\log_2$-ratio of each count to the size factor for the corresponding cell, after adding a prior count of 1 to avoid undefined values at zero counts. Division by the size factor ensures that any cell-specific biases are removed. If spike-in-specific size factors are present in `sce`, they will be automatically applied to normalize the spike-in transcripts separately from the endogenous genes.

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

The log-transformation is useful as it means that any differences in the values directly represent $\log_2$-fold changes in expression between cells. This is usually more relevant than the absolute differences in coverage, which need to be interpreted in the context of the overall abundance. The log-transformation also provides some measure of variance stabilization (Law et al. 2014), so that high-abundance genes with large variances do not dominate downstream analyses. The computed values are stored as an `"logcounts"` matrix in addition to the other assay elements.

7 Modelling the technical noise in gene expression

Variability in the observed expression values across genes can be driven by genuine biological heterogeneity or uninteresting technical noise. To distinguish between these two possibiltiies, we need to model the technical component of the variance of the expression values for each gene. We do so using the set of spike-in transcripts, which were added in the same quantity to each cell. Thus, the spike-in transcripts should exhibit no biological variability, i.e., any variance in their counts should be technical in origin.

We use the `trendVar()` function to fit a mean-dependent trend to the variances of the log-expression values for the spike-in transcripts. We set `block=` to block on the plate of origin for each

cell, to ensure that technical differences between plates do not inflate the variances. Given the mean abundance of a gene, the fitted value of the trend is then used as an estimate of the technical component for that gene. The biological component of the variance is finally calculated by subtracting the technical component from the total variance of each gene with the `decomposeVar` function.

```
var.fit <- trendVar(sce, parametric=TRUE, block=sce$Plate,
    loess.args=list(span=0.3))
var.out <- decomposeVar(sce, var.fit)
head(var.out)
```

```
## DataFrame with 6 rows and 6 columns
##                                    mean                tota
l
##                               <numeric>           <numeric
>
## ENSMUSG00000103377 0.00807160215879455  0.011921865484604
6
## ENSMUSG00000103147  0.0346526071354525  0.072219615904211
1
## ENSMUSG00000103161 0.00519472220098498 0.0049385769482036
2
## ENSMUSG00000102331  0.0186660929969477  0.032923591717629
1
## ENSMUSG00000102948  0.0590569999301664  0.088137125286206
4
## Rp1                 0.0970243710886109   0.45233813512634
9
##                                   bio                tech
p.value
##                               <numeric>           <numeric>
<numeric>
## ENSMUSG00000103377 -0.0238255786081426 0.0357474440927472
1
## ENSMUSG00000103147 -0.0812680860365811  0.153487701940792
0.999999999992144
## ENSMUSG00000103161 -0.0180705438097231 0.0230091207579267
1
## ENSMUSG00000102331 -0.0497487335708696 0.0826723252884987
0.999999999998056
## ENSMUSG00000102948  -0.173441452289705  0.261578577575911
1
## Rp1                  0.022609672867438  0.429728462258911
0.0354980961109997
##                                   FDR
##                             <numeric>
## ENSMUSG00000103377                  1
## ENSMUSG00000103147                  1
## ENSMUSG00000103161                  1
## ENSMUSG00000102331                  1
## ENSMUSG00000102948                  1
## Rp1                 0.153727769600592
```

We visually inspect the trend to confirm that it corresponds to the spike-in variances (Figure 8)). The wave-like shape is typical of the mean-variance trend for log-expression values. A linear increase in the variance is observed as the mean increases from zero, as larger variances are possible when the counts increase. At very high abundances, the effect of sampling noise decreases due to the law of large numbers, resulting in a decrease in the variance.

```
plot(var.out$mean, var.out$total, pch=16, cex=0.6, xlab="Mea
n log-expression",
    ylab="Variance of log-expression")
curve(var.fit$trend(x), col="dodgerblue", lwd=2, add=TRUE)
cur.spike <- isSpike(sce)
points(var.out$mean[cur.spike], var.out$total[cur.spike], co
l="red", pch=16)
```

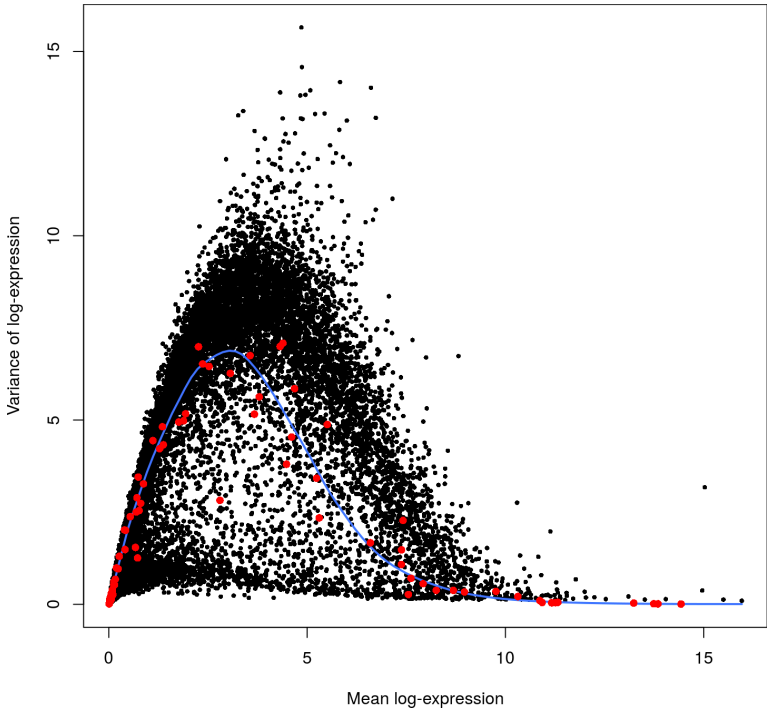
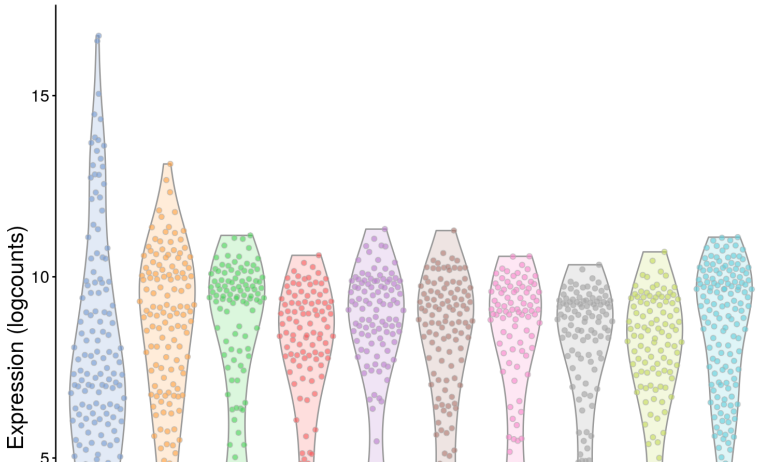


Figure 8: Variance of normalized log-expression values for each gene in the 416B dataset, plotted against the mean log-expression. The blue line represents the mean-dependent trend fitted to the variances of the spike-in transcripts (red).

We check the distribution of expression values for the genes with the largest biological components. This ensures that the variance estimate is not driven by one or two outlier cells (Figure 9).

```
chosen.genes <- order(var.out$bio, decreasing=TRUE)[1:10]
plotExpression(sce, features=rownames(var.out)[chosen.gene
s]) + fontsize
```



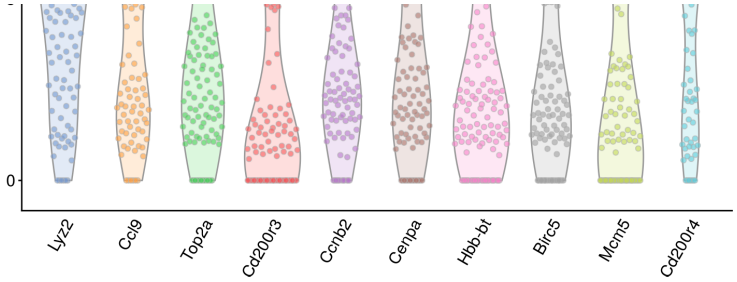


Figure 9: Violin plots of normalized log-expression values for the top 10 genes with the largest biological components in the 416B dataset. Each point represents the log-expression value in a single cell.

Comments from Aaron:

- In practice, trend fitting is complicated by the small number of spike-in transcripts and the uneven distribution of their abundances. See here (https://bioconductor.org/packages/3.8/simpleSingleCell/vignettes/xtra-3-var.html#details-of-trend-fitting-parameters) for more details on how to refine the fit.
- In the absence of spike-ins, users can set `use.spikes=FALSE` to fit a trend to the variances of the endogenous genes (see here (https://bioconductor.org/packages/3.8/simpleSingleCell/vignettes/xtra-3-var.html#when-spike-ins-are-unavailable)). Alternatively, we can create a trend based on the assumption of Poisson technical noise, as described here (https://bioconductor.org/packages/3.8/simpleSingleCell/vignettes/work-3-tenx.html#modelling-the-mean-variance-trend).
- Negative biological components are often obtained from `decomposeVar`. These are intuitively meaningless as it is impossible for a gene to have total variance below technical noise. Nonetheless, such values occur due to imprecise estimation of the total variance, especially for low numbers of cells.
- `decomposeVar` also yields *p*-values that can be used to define HVGs at a specific threshold for the false discovery rate (FDR). We will discuss this in more detail later, as formal detection of HVGs is not necessary for feature selection during data exploration.

8 Removing the batch effect

As previously mentioned, the data were collected on two plates. Small uncontrollable differences in processing between plates can result in a batch effect, i.e., systematic differences in expression between cells on different plates. Such differences are not interesting and can be removed by applying the `removeBatchEffect()` function from the *limma (https://bioconductor.org/packages/3.8/limma)* package (Ritchie et

al. 2015). This removes the effect of the plate of origin while accounting for the (interesting) effect of oncogene induction.

```
library(limma)
assay(sce, "corrected") <- removeBatchEffect(logcounts(sce),
    design=model.matrix(~sce$Oncogene), batch=sce$Plate)
assayNames(sce)
```

```
## [1] "counts"    "logcounts" "corrected"
```

Manual batch correction is necessary for downstream procedures that are not model-based, e.g., clustering and most forms of dimensionality reduction. However, if an analysis method can accept a design matrix, blocking on nuisance factors in the design matrix is preferable to using `removeBatchEffect()`. This is because the latter does not account for the loss of residual degrees of freedom, nor the uncertainty of estimation of the blocking factor terms.

Comments from Aaron:

- `removeBatchEffect()` performs a linear regression and sets the coefficients corresponding to the blocking factors to zero. This is effective provided that the population composition within each batch is known (and supplied as `design=`) or identical across batches. Such an assumption is reasonable for this dataset, involving a homogeneous cell line population on both plates. However, in most scRNA-seq applications, the factors of variation are not identical across batches and not known in advance. This motivates the use of more sophisticated batch correction methods such as `mnnCorrect()`.

9 Denoising expression values using PCA

Once the technical noise is modelled, we can use principal components analysis (PCA) to remove random technical noise. Consider that each cell represents a point in the high-dimensional expression space, where the spread of points represents the total variance. PCA identifies axes in this space that capture as much of this variance as possible. Each axis is a principal component (PC), where any early PC will explain more of the variance than a later PC.

We assume that biological processes involving co-regulated groups of genes will account for the most variance in the data. If this is the case, this process should be represented by one or more of the earlier PCs. In contrast, random technical noise affects each gene independently and will be represented by later PCs. The `denoisePCA()` function removes later PCs until the total

discarded variance is equal to the sum of technical components for all genes used in the PCA.

```
sce <- denoisePCA(sce, technical=var.out, assay.type="correc
ted")
dim(reducedDim(sce, "PCA"))
```

```
## [1] 183  24
```

The function returns a `SingleCellExperiment` object containing the PC scores for each cell in the `reducedDims` slot. The aim is to eliminate technical noise and enrich for biological signal in the retained PCs. This improves resolution of the underlying biology during downstream procedures such as clustering.

Comments from Aaron:

- `denoisePCA()` will only use genes that have positive biological components, i.e., variances greater than the fitted trend. This guarantees that the total technical variance to be discarded will not be greater than the total variance in the data.
- For the `technical=` argument, the function will also accept the trend function directly (i.e., `var.fit$trend`) or a vector of technical components per gene. Here, we supply the `DataFrame` from `decomposeVar()` to allow the function to adjust for the loss of residual degrees of freedom after batch correction. Specifically, the variance in the batch-corrected matrix is slightly understated, requiring some rescaling of the technical components to compensate.
- No filtering is performed on abundance here, which ensures that PCs corresponding to rare subpopulations can still be detected. Discreteness is less of an issue as low-abundance genes also have lower variance, thus reducing their contribution to the PCA.
- It is also possible to obtain a low-rank approximation of the original expression matrix, capturing the variance equivalent to the retained PCs. This is useful for denoising prior to downstream procedures that require gene-wise expression values.

```
sce2 <- denoisePCA(sce, technical=var.fit$trend,
    assay.type="corrected", value="lowrank")
assayNames(sce2)
```

```
## [1] "counts"    "logcounts" "corrected" "lowrank"
```

10 Visualizing data in low-dimensional space

10.1 With PCA

We visualize the relationships between cells by constructing pairwise PCA plots for the first three components (Figure 10). Cells with similar expression profiles should be located close together in the plot, while dissimilar cells should be far apart. In this case, we observe a clear separation of cells based on the oncogene induction status, consistent with the expected effects on the transcriptome.

```
plotReducedDim(sce, use_dimred="PCA", ncomponents=3,
    colour_by="Oncogene") + fontsize
```

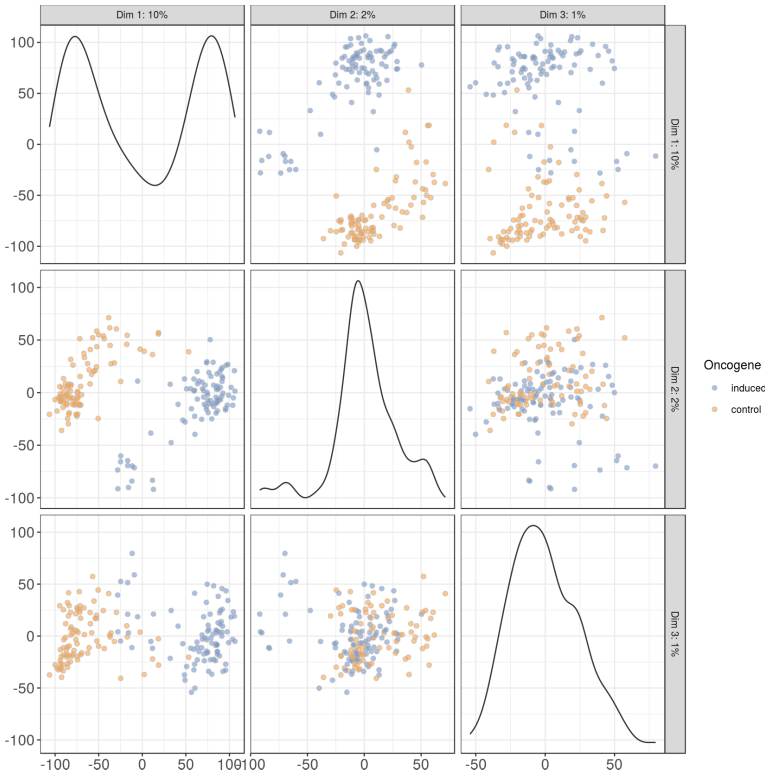
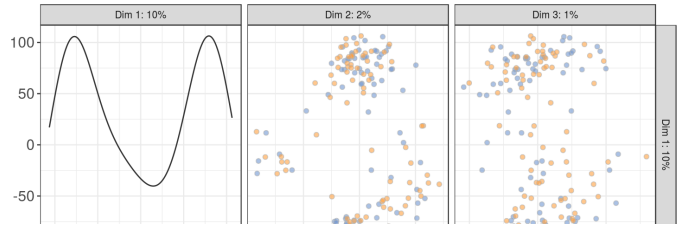


Figure 10: Pairwise PCA plots of the first three PCs in the 416B dataset, constructed from normalized log-expression values of genes with positive biological components. Each point represents a cell, coloured according to oncogene induction status.

By comparison, we observe no clear separation of cells by batch (Figure 11). This indicates that our batch correction step using `removeBatchEffect()` was successful.

```
plotReducedDim(sce, use_dimred="PCA", ncomponents=3,
    colour_by="Plate") + fontsize
```



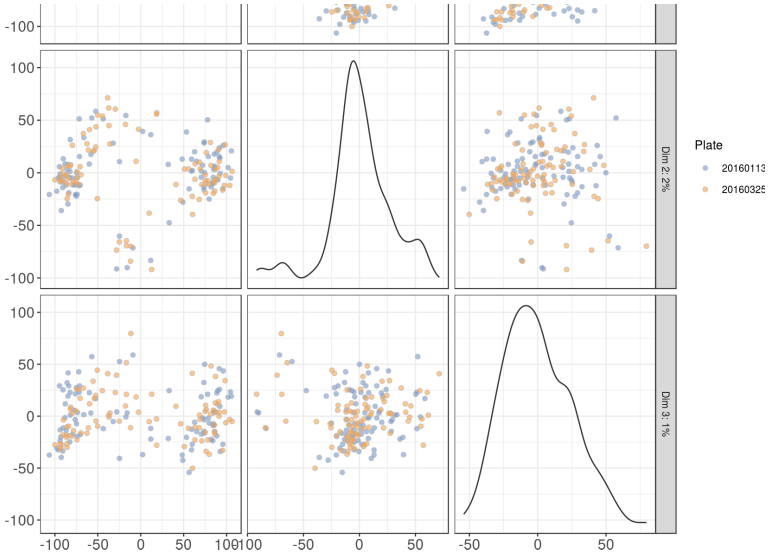


Figure 11: Pairwise PCA plots of the first three PCs in the 416B dataset, constructed from normalized log-expression values of genes with positive biological components. Each point represents a cell, coloured according to the plate of origin.

Note that `plotReducedDim()` will use the PCA results that were already stored in `sce` by `denoisePCA()`. This allows us to rapidly generate new plots with different aesthetics, without repeating the entire PCA computation. Similarly, `plotPCA()` will use existing results if they are available in the `SingleCellExperiment`, and will recalculate them otherwise. Users should set `rerun=TRUE` to forcibly recalculate the PCs in the presence of existing results.

Comments from Aaron:

- For each visualization method, additional cell-specific information can be incorporated into the size or shape of each point. This is done using the `size_by=` and `shape_by=` arguments in most plotting functions.
- More components can be shown but these are usually less informative as they explain less of the variance. They are also often more difficult to interpret as they are defined to be orthogonal to earlier PCs (and thus dependent on what is detected in those PCs).

10.2 With *t*-SNE

Another widely used approach for dimensionality reduction is the *t*-stochastic neighbour embedding (*t*-SNE) method (Van der Maaten and Hinton 2008). *t*-SNE tends to work better than PCA for separating cells in more diverse populations. This is because the former can directly capture non-linear relationships in high-dimensional space, whereas the latter must represent them on linear axes. However, this improvement comes at the cost of more computational effort and requires the user to consider parameters such as the random seed and perplexity (see comments).

We demonstrate the generation of *t*-SNE plots in Figure 12 using the `plotTSNE()` function. We set `use_dimred="PCA"` to perform the *t*-SNE on the low-rank approximation of the data, allowing the algorithm to take advantage of the previous denoising step.

```
set.seed(100)
out5 <- plotTSNE(sce, run_args=list(use_dimred="PCA", perple
xity=5),
    colour_by="Oncogene") + fontsize + ggtitle("Perplexity =
5")

set.seed(100)
out10 <- plotTSNE(sce, run_args=list(use_dimred="PCA", perpl
exity=10),
    colour_by="Oncogene") + fontsize + ggtitle("Perplexity =
10")

set.seed(100)
out20 <- plotTSNE(sce, run_args=list(use_dimred="PCA", perpl
exity=20),
    colour_by="Oncogene") + fontsize + ggtitle("Perplexity =
20")

multiplot(out5, out10, out20, cols=3)
```

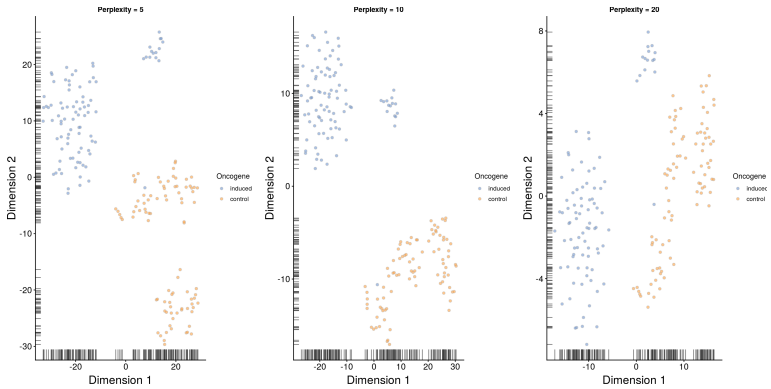


Figure 12: *t*-SNE plots constructed from the denoised PCs in the 416B dataset, using a range of perplexity values. Each point represents a cell, coloured according to its oncogene induction status. Bars represent the coordinates of the cells on each axis.

t-SNE is a stochastic method, so users should run the algorithm several times to ensure that the results are representative. Scripts should set a seed (via the `set.seed()` command) to ensure that the chosen results are reproducible. It is also advisable to test different settings of the "perplexity" parameter as this will affect the distribution of points in the low-dimensional space.

Here, we call `runTSNE()` with a perplexity of 20 to store the *t*-SNE results inside our `SingleCellExperiment` object. This avoids repeating the calculations whenever we want to create a new plot with `plotTSNE()`, as the stored results will be used instead. Again, users can set `rerun=TRUE` to force recalculation in the

presence of stored results.

```
set.seed(100)
sce <- runTSNE(sce, use_dimred="PCA", perplexity=20)
reducedDimNames(sce)
```

```
## [1] "PCA"  "TSNE"
```

There are many other dimensionality reduction techniques that we do not consider here but could also be used, e.g., multidimensional scaling, diffusion maps. These have their own advantages and disadvantages – for example, diffusion maps (see `plotDiffusionMap`) place cells along a continuous trajectory and are suited for visualizing graduated processes like differentiation (Angerer et al. 2016).

Comments from Aaron:

- A good guide on how to interpret *t*-SNE plots can be found at http://distill.pub/2016/misread-tsne/ (http://distill.pub/2016/misread-tsne/). This demonstrates how distances between clusters in the 2-dimensional embedding have little meaning, as does the apparent "size" (i.e., spread) of the clusters.

11 Clustering cells into putative subpopulations

11.1 Defining cell clusters from expression data

The denoised log-expression values are used to cluster cells into putative subpopulations. Specifically, we perform hierarchical clustering on the Euclidean distances between cells, using Ward's criterion to minimize the total variance within each cluster. This yields a dendrogram that groups together cells with similar expression patterns across the chosen genes.

```
pcs <- reducedDim(sce, "PCA")
my.dist <- dist(pcs)
my.tree <- hclust(my.dist, method="ward.D2")
```

Clusters are explicitly defined by applying a dynamic tree cut (Langfelder, Zhang, and Horvath 2008) to the dendrogram. This exploits the shape of the branches in the dendrogram to refine the cluster definitions, and is more appropriate than `cutree` for complex dendrograms. Greater control of the empirical clusters can be obtained by manually specifying `cutHeight` in `cutreeDynamic`. We also set `minClusterSize` to a lower value than the default of 20, to avoid spurious aggregation of distant

small clusters.

```
library(dynamicTreeCut)
my.clusters <- unname(cutreeDynamic(my.tree, distM=as.matrix
(my.dist),
    minClusterSize=10, verbose=0))
```

We examine the distribution of cells in each cluster with respect to known factors. Each cluster is comprised of cells from both batches, indicating that the clustering is not driven by a batch effect. Differences in the composition of each cluster are observed with respect to `Oncogene`, consistent with a biological effect of oncogene induction.

```
table(my.clusters, sce$Plate)
```

```
##
## my.clusters 20160113 20160325
##           1       41       39
##           2       19       20
##           3       16       11
##           4       10       14
##           5        5        8
```

```
table(my.clusters, sce$Oncogene)
```

```
##
## my.clusters induced control
##           1      80       0
##           2       0      39
##           3       0      27
##           4       0      24
##           5      13       0
```

We visualize the cluster assignments for all cells on the *t*-SNE plot in Figure 13. Adjacent cells are generally assigned to the same cluster, indicating that the clustering procedure was applied correctly.

```
sce$cluster <- factor(my.clusters)
plotTSNE(sce, colour_by="cluster") + fontsize
```

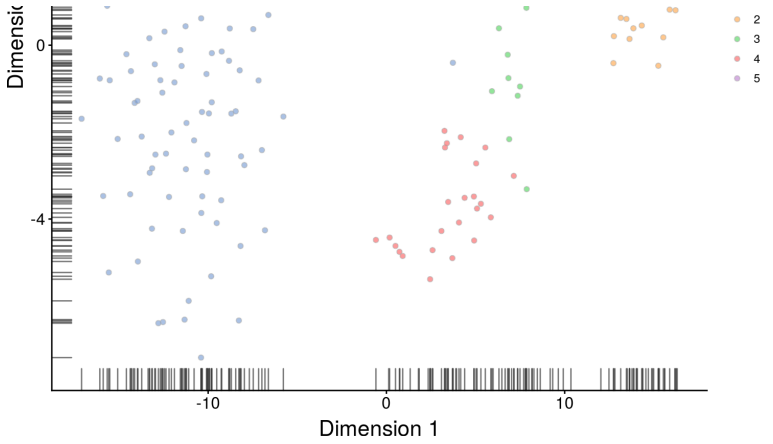
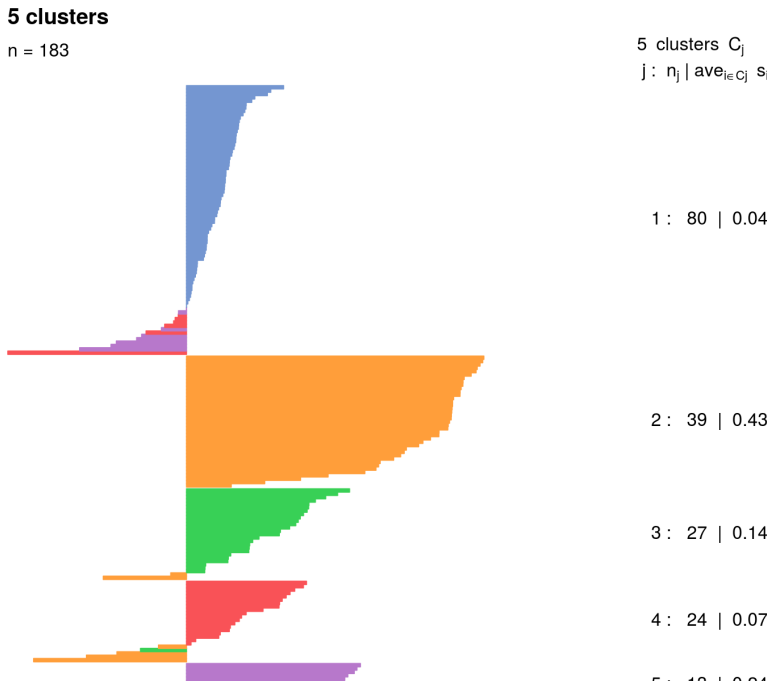
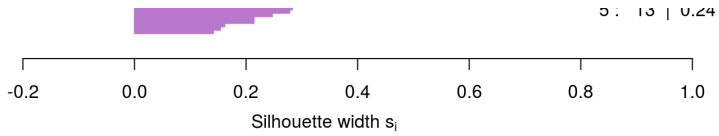


Figure 13: *t*-SNE plot of the denoised PCs of the 416B dataset. Each point represents a cell and is coloured according to the cluster identity to which it was assigned.

We check the separatedness of the clusters using the silhouette width (Figure 14). Cells with large positive silhouette widths are closer to other cells in the *same* cluster than to cells in *different* clusters. Conversely, cells with negative widths are closer to other clusters than to other cells in the cluster to which it was assigned. Each cluster would ideally contain many cells with large positive widths, indicating that it is well-separated from other clusters.

```
library(cluster)
clust.col <- scater:::.get_palette("tableau10medium") # hidd
en scater colours
sil <- silhouette(my.clusters, dist = my.dist)
sil.cols <- clust.col[ifelse(sil[,3] > 0, sil[,1], sil[,2])]
sil.cols <- sil.cols[order(-sil[,1], sil[,3])]
plot(sil, main = paste(length(unique(my.clusters)), "cluster
s"),
    border=sil.cols, col=sil.cols, do.col.sort=FALSE)
```





Average silhouette width : 0.15

Figure 14: Barplot of silhouette widths for cells in each cluster. Each cluster is assigned a colour and cells with positive widths are coloured according to the colour of its assigned cluster. Any cell with a negative width is coloured according to the colour of the cluster that it is closest to. The average width for all cells in each cluster is shown, along with the average width for all cells in the dataset.

The silhouette width can be used to determine the parameter values that maximize the separation between clusters. For example, we could vary the cut height or splitting depth in `cutreeDynamic` to maximize the average silhouette width across all cells. This usually provides a satisfactory initial clustering for further examination. However, keep in mind that the granularity of clustering is much like the magnification on a microscope. Different views of the data can be obtained with different granularities, some of which may be suboptimal on measures of separation. Users should not fixate on the clustering with the greatest separation if it does not provide the desired granularity for a particular biological question.

Most cells have relatively small silhouette positive widths in Figure 14, indicating that the separation between clusters is weak. This may be symptomatic of over-clustering where clusters that are clearly defined on oncogene induction status are further split into subsets that are less well separated. Nonetheless, we will proceed with the current clustering scheme in `my.clusters`, as it provides reasonable partitions for further characterization of heterogeneity.

Comments from Aaron:

- An alternative clustering strategy is to use a matrix of distances derived from correlations (e.g., as in `quickCluster`). This is more robust to noise and normalization errors, but is also less sensitive to subtle changes in the expression profiles.
- Both Ward's criterion and complete linkage yield spherical, compact clusters. In particular, complete linkage favours the formation of clusters with the same diameter. This may be desirable in some cases but is less appropriate when subpopulations differ in their variance. Thus, we typically use Ward's criterion for our initial clustering. Of course, it is simple (and recommended) to try other approaches provided that some assessment is performed, e.g., using the silhouette width.

11.2 Detecting marker genes between clusters

Once putative subpopulations are identified by clustering, we can

identify marker genes for each cluster using the `findMarkers` function. This performs Welch t-tests on the log-expression values for every gene and between every pair of clusters (Soneson and Robinson 2018). The aim is to test for DE in each cluster compared to the others while blocking on uninteresting factors such as the plate of origin. The top DE genes are likely to be good candidate markers as they can effectively distinguish between cells in different clusters.

```
markers <- findMarkers(sce, my.clusters, block=sce$Plate)
```

For each cluster, the DE results of the relevant comparisons are consolidated into a single output table. This allows a set of marker genes to be easily defined by taking the top DE genes from each pairwise comparison between clusters. For example, to construct a marker set for cluster 1 from the top 10 genes of each comparison, one would filter `marker.set` to retain rows with `Top` less than or equal to 10. Other statistics are also reported for each gene, including the adjusted p-values (see below) and the log-fold changes relative to every other cluster.

```
marker.set <- markers[["1"]]
head(marker.set, 10)
```

```
## DataFrame with 10 rows and 7 columns
##              Top              p.value                  FD
R           logFC.2
##        <integer>            <numeric>            <numeric
>         <numeric>
## Aurkb          1  6.6586365293815e-75 1.58362352578282e-7
0 -7.37163348746989
## Tk1            1 6.41442400544287e-64 3.81385615303623e-6
0 -4.92754578116824
## Myh11          1 4.95086470978904e-49  9.8122012827427e-4
6  4.42159319409116
## Cdca8          1 2.22335011226212e-46   3.525195714662e-4
3 -6.84273524406495
## Ccna2          2 1.16841256912409e-68 1.38941780657393e-6
4 -7.30797562717896
## Rrm2           2 1.48873865795832e-56 5.05809592888897e-5
3 -5.52120320375744
## Cks1b          2 3.83636250452745e-39 2.40105814329412e-3
6 -6.67921179963203
## Pirb           2 1.83893780552265e-34 6.15992363785147e-3
2  5.25803750523071
## Pimreg         3 7.41004852233185e-68 5.87443946688721e-6
4 -7.30454209124205
## Pclaf          3 8.96517654145094e-51 2.13218793685327e-4
7 -5.60087983909796
##                  logFC.3            logFC.4            lo
gFC.5
##                <numeric>          <numeric>          <num
eric>
## Aurkb  -6.72799344278974  -1.95039440707189  -6.911288011
52809
## Tk1    -7.74065112846769  -3.53749564918697  -4.635162724
78573
## Myh11   4.30812919188871    4.4523571874051   1.041314956
10462
## Cdca8  -4.88595091616654  -2.43821401649457  -7.127914702
71152
## Ccna2  -6.96768519428654  -2.46589325539015  -7.126927208
39657
## Rrm2   -7.94685698699535  -3.19173143336724  -5.428780908
99579
## Cks1b  -5.92137180544953  -4.37146346065968  -6.214592461
47227
## Pirb    5.18195597944541   5.87631058664443 0.07049643744
72336
## Pimreg -5.91099761451465 -0.874660675194747  -7.017988525
50674
## Pclaf  -7.56997892341564  -2.36631043047157  -5.169569267
37242
```

We save the list of candidate marker genes for further examination.

```
write.table(marker.set, file="416B_marker_1.tsv", sep="\t",
    quote=FALSE, col.names=NA)
```

We visualize the expression profiles of the top candidates to verify that the DE signature is robust (Figure 15). Most of the top markers have strong and consistent up- or downregulation in cells of cluster 1 compared to some or all of the other clusters. A cursory examination of the heatmap indicates that cluster 1 contains oncogene-induced cells with strong downregulation of DNA replication and cell cycle genes. This is consistent with the potential induction of senescence as an anti-tumorigenic response (Wajapeyee et al. 2010). A more comprehensive investigation of the function of these markers can be performed with gene set enrichment analyses, e.g., using `kegga` or `goana` from *limma (https://bioconductor.org/packages/3.8/limma)*.

```
top.markers <- rownames(marker.set)[marker.set$Top <= 10]
plotHeatmap(sce, features=top.markers, columns=order(sce$clu
ster),
    colour_columns_by=c("cluster", "Plate", "Oncogene"),
    cluster_cols=FALSE, center=TRUE, symmetric=TRUE, zlim=c
(-5, 5))
```

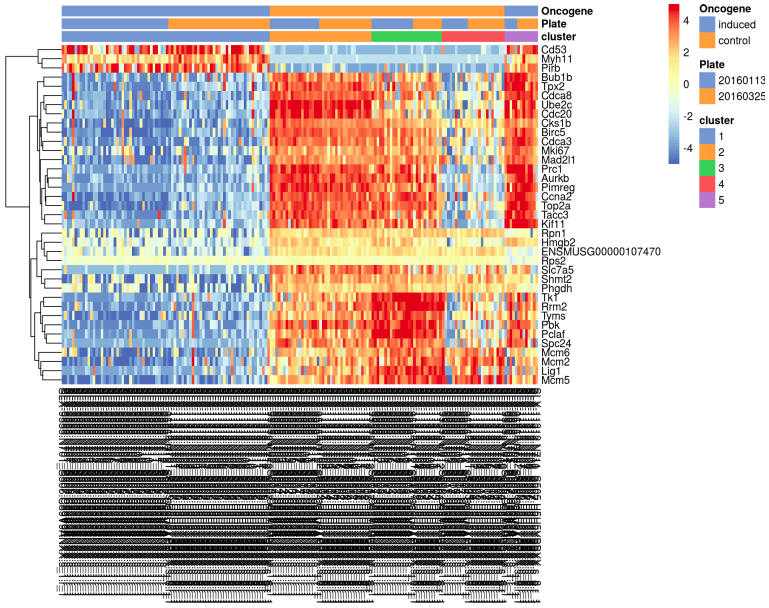


Figure 15: Heatmap of mean-centred and normalized log-expression values for the top set of markers for cluster 1 in the 416B dataset. Column colours represent the cluster to which each cell is assigned, the plate of origin or the oncogene induction status of each cell, as indicated by the legend.

Many of the markers in Figure 15 are not uniquely up- or downregulated in the chosen cluster. Testing for unique DE tends to be too stringent as it overlooks important genes that are expressed in two or more clusters. For example, in a mixed population of $CD4^+$-only, $CD8^+$-only, double-positive and double-negative T cells, neither *Cd4* or *Cd8* would be detected as

subpopulation-specific markers because each gene is expressed in two subpopulations. With our approach, both of these genes will be picked up as candidate markers as they will be DE between at least one pair of subpopulations. A combination of markers can then be chosen to characterize a subpopulation, which is more flexible than trying to find uniquely DE genes.

We strongly recommend selecting some markers for use in validation studies with an independent replicate population of cells. The aim is to identify a corresponding subset of cells that express the upregulated markers and do not express the downregulated markers. Ideally, a different technique for quantifying expression would also be used during validation, e.g., fluorescent *in situ* hybridisation or quantitative PCR. This confirms that the subpopulation genuinely exists and is not an artifact of scRNA-seq or the computational analysis.

Comments from Aaron:

- By setting `direction="up"`, `findMarkers` will only return genes that are upregulated in each cluster compared to the others. This is convenient in highly heterogeneous populations to focus on genes that can immediately identify each cluster. While lack of expression may also be informative, it is less useful for positive identification.
- By setting `block=`, `findMarkers()` will perform pairwise tests between clusters using only cells on the same plate. It will then combine p-values from different plates using Stouffer's Z method to obtain a single p-value per gene. An alternative approach is to use linear models via the `design=` argument - the differences between the two are discussed here (https://bioconductor.org/packages/3.8/simpleSingleCell/vignettes/work-5-mnn.html#using-the-corrected-values-in-downstream-analyses).
- `findMarkers` can also be directed to find genes that are DE between the chosen cluster and *all* other clusters. This should be done by setting `pval.type="all"`, which defines the p-value for each gene as the maximum value across all pairwise comparisons involving the chosen cluster. Combined with `direction="up"`, this can be used to identify unique markers for each cluster. However, this is sensitive to overclustering, as unique marker genes will no longer exist if a cluster is split into two smaller subclusters.
- It must be stressed that the (adjusted) *p*-values computed here cannot be properly interpreted as measures of significance. This is because the clusters have been empirically identified from the data. *limma (https://bioconductor.org/packages/3.8/limma)* does not account for the uncertainty of clustering, which means that the *p*-values are much lower than they should be. This is not a concern in other analyses where the groups are pre-

defined.

- The `overlapExprs` function may also be useful here, to prioritize candidates where there is clear separation between the distributions of expression values of different clusters. This uses the Wilcoxon rank sum test to detect uneven mixing of the distributions of expression values between clusters. By contrast, `findMarkers` uses t-tests and is primarily concerned with log-fold changes.

12 Concluding remarks

Once the basic analysis is completed, it is often useful to save the `SingleCellExperiment` object to file with the `saveRDS` function. The object can then be easily restored into new R sessions using the `readRDS` function. This allows further work to be conducted without having to repeat all of the processing steps described above.

```
saveRDS(file="416B_data.rds", sce)
```

A variety of methods are available to perform more complex analyses on the processed expression data. For example, cells can be ordered in pseudotime (e.g., for progress along a differentiation pathway) with *monocle (https://bioconductor.org/packages/3.8/monocle)* (Trapnell et al. 2014) or *TSCAN (https://bioconductor.org/packages/3.8/TSCAN)* (Ji and Ji 2016); cell-state hierarchies can be characterized with the *sincell (https://bioconductor.org/packages/3.8/sincell)* package (Julia, Telenti, and Rausell 2015); and oscillatory behaviour can be identified using *Oscope (https://bioconductor.org/packages/3.8/Oscope)* (Leng et al. 2015). HVGs can be used in gene set enrichment analyses to identify biological pathways and processes with heterogeneous activity, using packages designed for bulk data like *topGO (https://bioconductor.org/packages/3.8/topGO)* or with dedicated single-cell methods like *scde (https://bioconductor.org/packages/3.8/scde)* (Fan et al. 2016). Full descriptions of these analyses are outside the scope of this workflow, so interested users are advised to consult the relevant documentation.

All software packages used in this workflow are publicly available from the Comprehensive R Archive Network (https://cran.r-project.org (https://cran.r-project.org)) or the Bioconductor project (http://bioconductor.org (http://bioconductor.org)). The specific version numbers of the packages used are shown below, along with the version of the R installation.

```
sessionInfo()
```

```
## R version 3.5.2 (2018-12-20)
## Platform: x86_64-pc-linux-gnu (64-bit)
## Running under: Ubuntu 16.04.5 LTS
##
## Matrix products: default
## BLAS: /home/biocbuild/bbs-3.8-bioc/R/lib/libRblas.so
## LAPACK: /home/biocbuild/bbs-3.8-bioc/R/lib/libRlapack.so
##
## locale:
##  [1] LC_CTYPE=en_US.UTF-8       LC_NUMERIC=C
##  [3] LC_TIME=en_US.UTF-8        LC_COLLATE=C
##  [5] LC_MONETARY=en_US.UTF-8    LC_MESSAGES=en_US.UTF-8
##  [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    stats     graphics  grDevices uti
ls     datasets
## [8] methods   base
##
## other attached packages:
##  [1] cluster_2.0.7-1
##  [2] dynamicTreeCut_1.63-1
##  [3] limma_3.38.3
##  [4] scran_1.10.2
##  [5] scater_1.10.1
##  [6] ggplot2_3.1.0
##  [7] TxDb.Mmusculus.UCSC.mm10.ensGene_3.4.0
##  [8] GenomicFeatures_1.34.1
##  [9] org.Mm.eg.db_3.7.0
## [10] AnnotationDbi_1.44.0
## [11] SingleCellExperiment_1.4.1
## [12] SummarizedExperiment_1.12.0
## [13] DelayedArray_0.8.0
## [14] BiocParallel_1.16.5
## [15] matrixStats_0.54.0
## [16] Biobase_2.42.0
## [17] GenomicRanges_1.34.0
## [18] GenomeInfoDb_1.18.1
## [19] IRanges_2.16.0
## [20] S4Vectors_0.20.1
## [21] BiocGenerics_0.28.0
## [22] bindrcpp_0.2.2
## [23] BiocFileCache_1.6.0
## [24] dbplyr_1.2.2
## [25] knitr_1.21
## [26] BiocStyle_2.10.0
##
## loaded via a namespace (and not attached):
##  [1] bitops_1.0-6             bit64_0.9-7
##  [3] RColorBrewer_1.1-2       progress_1.2.0
##  [5] httr_1.4.0               tools_3.5.2
##  [7] R6_2.3.0                 KernSmooth_2.23-15
```

```
##  [9] HDF5Array_1.10.1         vipor_0.4.5
## [11] DBI_1.0.0                lazyeval_0.2.1
## [13] colorspace_1.3-2         withr_2.1.2
## [15] tidyselect_0.2.5         gridExtra_2.3
## [17] prettyunits_1.0.2        bit_1.1-14
## [19] curl_3.2                 compiler_3.5.2
## [21] BiocNeighbors_1.0.0      rtracklayer_1.42.1
## [23] labeling_0.3             bookdown_0.9
## [25] scales_1.0.0             rappdirs_0.3.1
## [27] stringr_1.3.1            digest_0.6.18
## [29] Rsamtools_1.34.0         rmarkdown_1.11
## [31] XVector_0.22.0           pkgconfig_2.0.2
## [33] htmltools_0.3.6          highr_0.7
## [35] rlang_0.3.1              RSQLite_2.1.1
## [37] DelayedMatrixStats_1.4.0 bindr_0.1.1
## [39] dplyr_0.7.8              RCurl_1.95-4.11
## [41] magrittr_1.5             GenomeInfoDbData_1.2.0
## [43] Matrix_1.2-15            Rcpp_1.0.0
## [45] ggbeeswarm_0.6.0         munsell_0.5.0
## [47] Rhdf5lib_1.4.2           viridis_0.5.1
## [49] edgeR_3.24.3             stringi_1.2.4
## [51] yaml_2.2.0               zlibbioc_1.28.0
## [53] Rtsne_0.15               rhdf5_2.26.2
## [55] plyr_1.8.4               grid_3.5.2
## [57] blob_1.1.1               crayon_1.3.4
## [59] lattice_0.20-38          Biostrings_2.50.2
## [61] cowplot_0.9.4            hms_0.4.2
## [63] locfit_1.5-9.1           pillar_1.3.1
## [65] igraph_1.2.2             reshape2_1.4.3
## [67] biomaRt_2.38.0           XML_3.98-1.16
## [69] glue_1.3.0               evaluate_0.12
## [71] BiocManager_1.30.4       gtable_0.2.0
## [73] purrr_0.2.5              assertthat_0.2.0
## [75] xfun_0.4                 viridisLite_0.3.0
## [77] pheatmap_1.0.12          tibble_2.0.0
## [79] GenomicAlignments_1.18.1 beeswarm_0.2.3
## [81] memoise_1.1.0            statmod_1.4.30
```

References

Anders, S., and W. Huber. 2010. “Differential expression analysis for sequence count data.” *Genome Biol.* 11 (10):R106.

Angerer, P., L. Haghverdi, M. Buttner, F. J. Theis, C. Marr, and F. Buettner. 2016. “destiny: diffusion maps for large-scale single-cell data in R.” *Bioinformatics* 32 (8):1241–3.

Bertoli, C., J. M. Skotheim, and R. A. de Bruin. 2013. “Control of cell cycle transcription during G1 and S phases.” *Nat. Rev. Mol. Cell Biol.* 14 (8):518–28.

Bourgon, R., R. Gentleman, and W. Huber. 2010. “Independent filtering increases detection power for high-throughput

experiments.” *Proc. Natl. Acad. Sci. U.S.A.* 107 (21):9546–51.

Conboy, C. M., C. Spyrou, N. P. Thorne, E. J. Wade, N. L. Barbosa-Morais, M. D. Wilson, A. Bhattacharjee, et al. 2007. “Cell cycle genes are the evolutionarily conserved targets of the E2F4 transcription factor.” *PLoS ONE* 2 (10):e1061.

Fan, J., N. Salathia, R. Liu, G. E. Kaeser, Y. C. Yung, J. L. Herman, F. Kaper, et al. 2016. “Characterizing transcriptional heterogeneity through pathway and gene set overdispersion analysis.” *Nat. Methods* 13 (3):241–44.

Ilicic, T., J. K. Kim, A. A. Kołodziejczyk, F. O. Bagger, D. J. McCarthy, J. C. Marioni, and S. A. Teichmann. 2016. “Classification of low quality cells from single-cell RNA-seq data.” *Genome Biol.* 17 (1):29.

Islam, S., A. Zeisel, S. Joost, G. La Manno, P. Zajac, M. Kasper, P. Lonnerberg, and S. Linnarsson. 2014. “Quantitative single-cell RNA-seq with unique molecular identifiers.” *Nat. Methods* 11 (2):163–66.

Ji, Z., and H. Ji. 2016. “TSCAN: Pseudo-time reconstruction and evaluation in single-cell RNA-seq analysis.” *Nucleic Acids Res.* 44 (13):e117.

Julia, M., A. Telenti, and A. Rausell. 2015. “Sincell: an R/Bioconductor package for statistical assessment of cell-state hierarchies from single-cell RNA-seq.” *Bioinformatics* 31 (20):3380–2.

Langfelder, P., B. Zhang, and S. Horvath. 2008. “Defining clusters from a hierarchical cluster tree: the Dynamic Tree Cut package for R.” *Bioinformatics* 24 (5):719–20.

Law, C. W., Y. Chen, W. Shi, and G. K. Smyth. 2014. “voom: Precision weights unlock linear model analysis tools for RNA-seq read counts.” *Genome Biol.* 15 (2):R29.

Leng, N., L. F. Chu, C. Barry, Y. Li, J. Choi, X. Li, P. Jiang, R. M. Stewart, J. A. Thomson, and C. Kendziorski. 2015. “Oscope identifies oscillatory genes in unsynchronized single-cell RNA-seq experiments.” *Nat. Methods* 12 (10):947–50.

Love, M. I., W. Huber, and S. Anders. 2014. “Moderated estimation of fold change and dispersion for RNA-seq data with DESeq2.” *Genome Biol.* 15 (12):550.

Lun, A. T. L., F. J. Calero-Nieto, L. Haim-Vilmovsky, B. Gottgens, and J. C. Marioni. 2017. “Assessing the reliability of spike-in normalization for analyses of single-cell RNA sequencing data.” *Genome Res.* 27 (11):1795–1806.

Lun, A. T., K. Bach, and J. C. Marioni. 2016. “Pooling across cells to normalize single-cell RNA sequencing data with many zero

counts.” *Genome Biol.* 17 (April):75.

McCarthy, D. J., K. R. Campbell, A. T. Lun, and Q. F. Wills. 2017. “Scater: pre-processing, quality control, normalization and visualization of single-cell RNA-seq data in R.” *Bioinformatics* 33 (8):1179–86.

Picelli, S., O. R. Faridani, A. K. Bjorklund, G. Winberg, S. Sagasser, and R. Sandberg. 2014. “Full-length RNA-seq from single cells using Smart-seq2.” *Nat Protoc* 9 (1):171–81.

Ritchie, M. E., B. Phipson, D. Wu, Y. Hu, C. W. Law, W. Shi, and G. K. Smyth. 2015. “limma powers differential expression analyses for RNA-sequencing and microarray studies.” *Nucleic Acids Res.* 43 (7):e47.

Robinson, M. D., and A. Oshlack. 2010. “A scaling normalization method for differential expression analysis of RNA-seq data.” *Genome Biol.* 11 (3):R25.

Scialdone, A., K. N. Natarajan, L. R. Saraiva, V. Proserpio, S. A. Teichmann, O. Stegle, J. C. Marioni, and F. Buettner. 2015. “Computational assignment of cell-cycle stage from single-cell transcriptome data.” *Methods* 85 (September):54–61.

Soneson, C., and M. D. Robinson. 2018. “Bias, robustness and scalability in single-cell differential expression analysis.” *Nat. Methods* 15 (4):255–61.

Stegle, O., S. A. Teichmann, and J. C. Marioni. 2015. “Computational and analytical challenges in single-cell transcriptomics.” *Nat. Rev. Genet.* 16 (3):133–45.

Trapnell, C., D. Cacchiarelli, J. Grimsby, P. Pokharel, S. Li, M. Morse, N. J. Lennon, K. J. Livak, T. S. Mikkelsen, and J. L. Rinn. 2014. “The dynamics and regulators of cell fate decisions are revealed by pseudotemporal ordering of single cells.” *Nat. Biotechnol.* 32 (4):381–86.

Van der Maaten, L., and G. Hinton. 2008. “Visualizing Data Using T-SNE.” *J. Mach. Learn. Res.* 9 (2579-2605):85.

Wajapeyee, N., S. Z. Wang, R. W. Serra, P. D. Solomon, A. Nagarajan, X. Zhu, and M. R. Green. 2010. “Senescence induction in human fibroblasts and hematopoietic progenitors by leukemogenic fusion proteins.” *Blood* 115 (24):5057–60.