A pipeline to analyse time-course gene expression data

immediate

Abstract The phenotypic diversity of cells is governed by a complex equilibrium between their genetic identity and their environmental interactions: Understanding the dynamics of gene expression is a fundamental question of biology.However, analysing time-course transcriptomic data raises unique challenging statistical and computational questions, requiring the development of novel methods and software. Using as case study time-course transcriptomics data from mice exposed to different strains of influenza, this workflow provides a step-by-step tutorial of the methodology used to analyse time-course data: (1) normalization of the micro-array data set; (2) differential expression analysis using functional data analysis; (3) clustering of time-course data; (4) interpreting clusters with GO term and KEGG pathway enrichment analysis.

Keywords

time-course gene expression data, clustering, differential expression, workflow

The manuscript was last compile on 2019-12-04 09:22:37.

#<U+00A0>TODO list

- Make clear that it applies to RNASeq and how to call the different functions to RNASeq
- Check that clustering can deal with count data.

EAP: What is your plan for the installation instructions Nelle? Shouldn't it be moved where we introduce the package, and give instructions about how to install it there.

Introduction

Gene expression studies provide simultaneous quantification of the level of mRNA from all genes in a sample. High-throughput studies of gene expression have a long history, starting with microarray technologies in the 1990s through to single-cell technologies. While many expression studies are designed to compare the gene expression between distinct groups, there is also a long history of time-course expression studies. Such studies compare the gene expression across time by measuring mRNA levels from samples collected at different timepoints [1]. Such time-course studies can vary from measuring a few distinct time points, to sampling ten to twenty time points. These longer time series are particularly interested in investigating development over time. More recently, a new variety of time course studies have come from single-cell sequencing experiments [1, 2, 3] which can sequence single cells at different stages of development; in this case, the time point is the stage of the cell in the process of development – a value that is not know but estimated from the data as its "pseudo-time."

While there are many methods that have been proposed for discrete aspects of time course data, the entire workflow for analysis of such data remains difficult, particularly for long, developmental time series. Most methods proposed for time course data are concerned with detecting genes that are changing over time (differential expression analysis), examples being `edge` [4], functional component analysis based models [5], time-course permutation tests [6], and multiple testing strategies to combine single time point differential expression analysis [7]. However, with long time course data sets, particularly in developmental systems, a massive number of genes will show *some* change. For example, in a study of mice lung tissues infected with influenza that we consider in this workflow, over 50% of genes are shown to be changing over time. The task in these settings is often not to detect changes in genes, but to categorize them into biologically interpretable patterns.

We present here a workflow for such an analysis that consists of 4 main parts (Figure 1):

- Quality control and normalization;
- Identification of genes that are differentially expressed;
- Clustering of genes into distinct temporal patterns;
- Biological interpretation of the clusters.

This workflow represents an integration of both novel implementations of previously established methods and new methodologies for the settings of developmental time series. It relies on several standard packages for analysing gene expression data, some specific for time-course data, others broadly used by the community. We provide the various steps of the workflow as functions in a R package called `moanin`.

Analysis of the dynamical response of mouse lung tissue to influenza

This workflow is illustrated using data from a micro-array time-course experiment, exposing mice to three different strains of influenza, and collecting lung tissue during 14 time-points after infection (0, 3, 6, 9, 12, 18, 24, 30, 36, 48, 60 hours, then 3, 5, and 7 days later) [8]. The three strains of influenza used in the study are (1) a low pathogenicity seasonal H1N1 influenza virus (A/Kawasaki/UTK4/2009 [H1N1]), a mildly pathogenic virus from the 2009 pandemic season (A/California/04/2009 [H1N1]), and a highly pathogenic H5N1 avian influenza virus (A/Vietnam/1203/2004 [H5N1]. Mice were injected with 10^5 PFU of each virus. An adidtional 42 mice were injected with a lower dose of the Vietnam avian influenza virus (10^3 PFU).

By combining gene expression time-course data with virus growth data, the authors show that the inflammatory response of lung tissue is gated until a threshold of the virus concentration is exceeded in the lung. Once this threshold is exceeded, a strong inflammatory and cytokin production occurs. This results provides evidence that the pathology response is non-linearly regulated by virus concentration.

While we showcase this pipeline on micro-array data, [9] leverages a similar set of steps to analyse RNA-seq data of the lifetime transcriptomic response of the crop *S. bicolor* to drought.

[1] Because the collection of the mRNA is often destructive, samples at different time points are generally from different biological samples; longitudinal studies, for example tracking the same subject over time, are certainly possible, but not directly considered here.

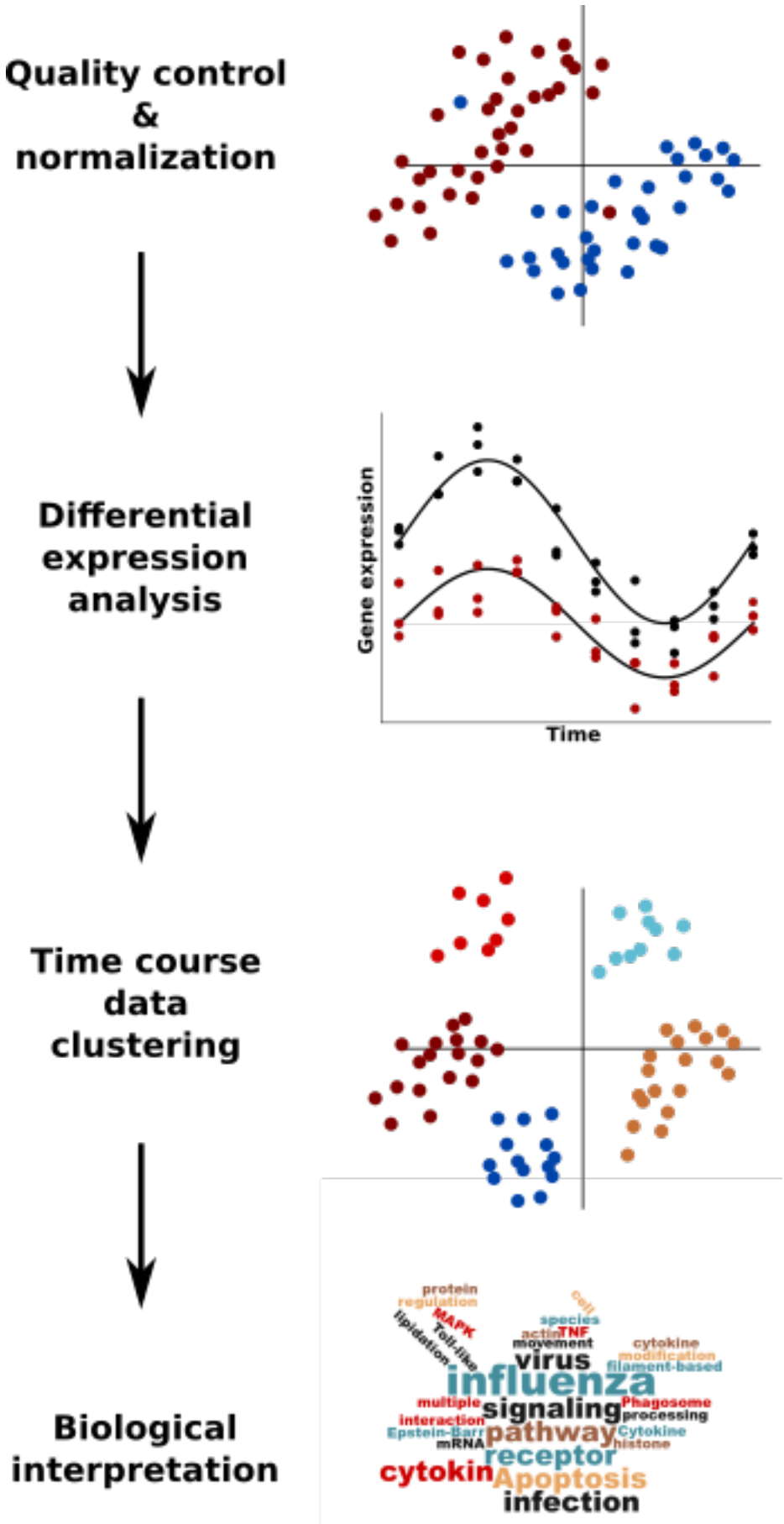


Figure 1. **Workflow for analyzing time-course datasets.**

Package versions

The following packages are needed for this workflow.

```
# From CRAN
library(NMF)
library(ggfortify)

# From Bioconductor
library(topGO)
library(biomaRt)

# From GitHub
library(moanin)
```

To install `moanin` directly from GitHub, use the devtools' `install_github` function:

```
library(devtools)
install_github("NelleV/moanin")

library(moanin)
```

Quality control and normalization

The first steps of analysis of gene expression data is always to do normalization and quality control checks of the data. In what follows, we show an example of this for the influenza data using generic methods; these steps are not in specific to time course data, but could be done for any gene expression analysis.

First let's load the data. The package `moanin` contains the normalized data and metadata of [8].

```
# Now load in the metadata
data(shoemaker2015)
meta = shoemaker2015$meta
data = shoemaker2015$data
```

The meta data contains information about the treatment group, the replicate, and the timepoint for each observation:

		Group	Replicate	Timepoint
GSM1557140	0	K	1	0
GSM1557141	1	K	2	0
GSM1557142	2	K	3	0
GSM1557143	3	K	1	12
GSM1557144	4	K	2	12
GSM1557145	5	K	3	12

Before we dive into the exploratory analysis and quality control, let us define color schemes for our data that we will use across the whole analysis. We define color schemes for groups and time points as named vectors. We also define a series of markers (or plotting symbols) to distinguish replicate samples in scatter plots.

```
group_colors = c(
    "M"="dodgerblue4",
    "K"="gold",
    "C"="orange",
    "VH"="red4",
    "VL"="red2")
time_colors = grDevices::rainbow(15)[1:14]
names(time_colors) = c(0, 3, 6, 9, 12, 18, 24, 30, 36, 48, 60, 72, 120, 168)

# Combine all color schemes into one named lists.
ann_colors = list(
    Timepoint=time_colors,
    Group=group_colors
    )
```

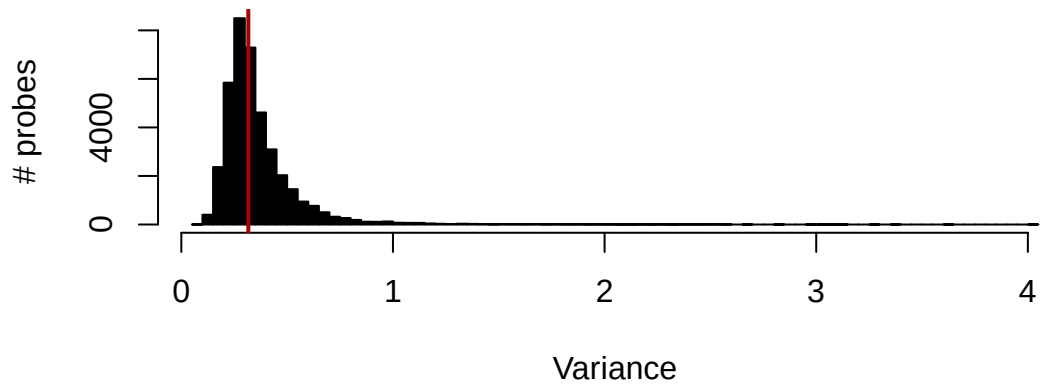
```
replicate_markers = c(15, 17, 19)
names(replicate_markers) = c(1, 2, 3)
ann_markers = list(
    Replicate=replicate_markers)
```

Exploratory analysis and quality control

Typically, two quality control and exploratory analysis steps are also performed before and after normalization: (1) low dimensionality embedding of the samples; (2) correlation plots between each samples. In both cases, we expect a strong biological signal, while replicate samples should be strongly clustered or correlated with one another.

Before performing any additional exploratory analysis, let us only keep highly variable genes: for this step, we keep only the top 50% most variable genes.

```
variance_cutoff = 0.5
# Filter genes by median absolute deviation (mad)
variance_per_genes = apply(data, 1, mad)
min_variance = quantile(variance_per_genes, c(variance_cutoff))
variance_filtered_data = data[variance_per_genes > min_variance,]
```



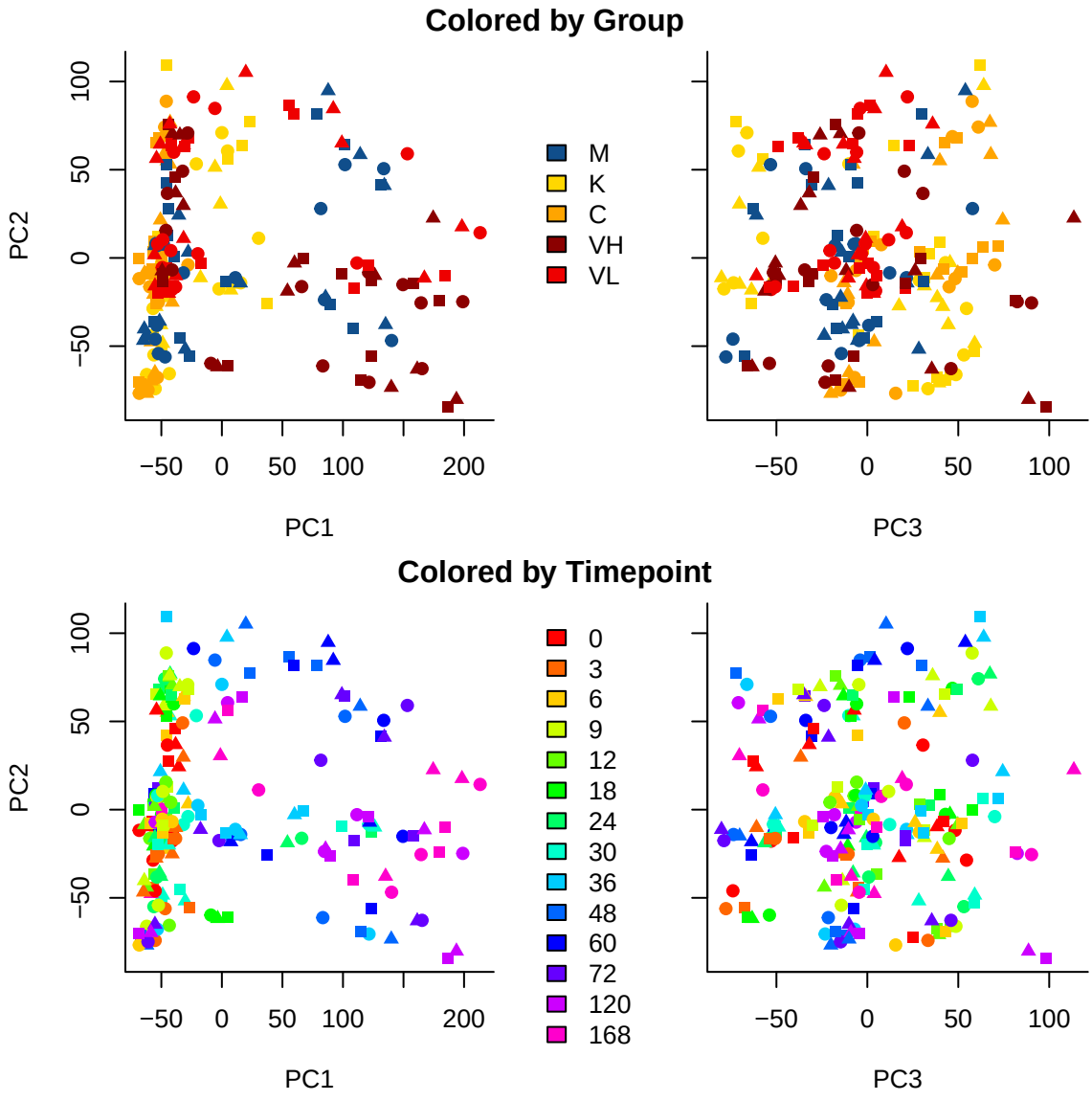
Let us first perform the PCA analysis. Here, we perform a PCA of rank 3 of the centered and scaled gene expression data.

```
pca_data = prcomp(t(variance_filtered_data), rank=3, center=TRUE, scale=TRUE)
percent_var = round(100 * attr(pca_data, "percentVar"))
```

We then plot the two first PC components, and color each sample by (1) its condition; (2) its sampling time. We use different plotting symbols for each replicate. We also plot the second and third components in the second column.

```
par(mfrow=c(2, 2))
par(mar=c(4.5, 4.5, 1.5, 1.5))
plot(
    pca_data$x[, c("PC1","PC2")],
    col=ann_colors$Group[meta$Group],
    pch=ann_markers$Replicate[as.factor(meta$Replicate)], bty="l")
mtext(at=275, text="Colored by Group", line=0.5, font=2, adj=0.5)
legend(x=250, y=25, legend=names(ann_colors$Group),
       fill=ann_colors$Group, bty="n", xpd=NA, yjust=0.5)
par(mar=c(4.5, 5.5, 1.5, .1))
```

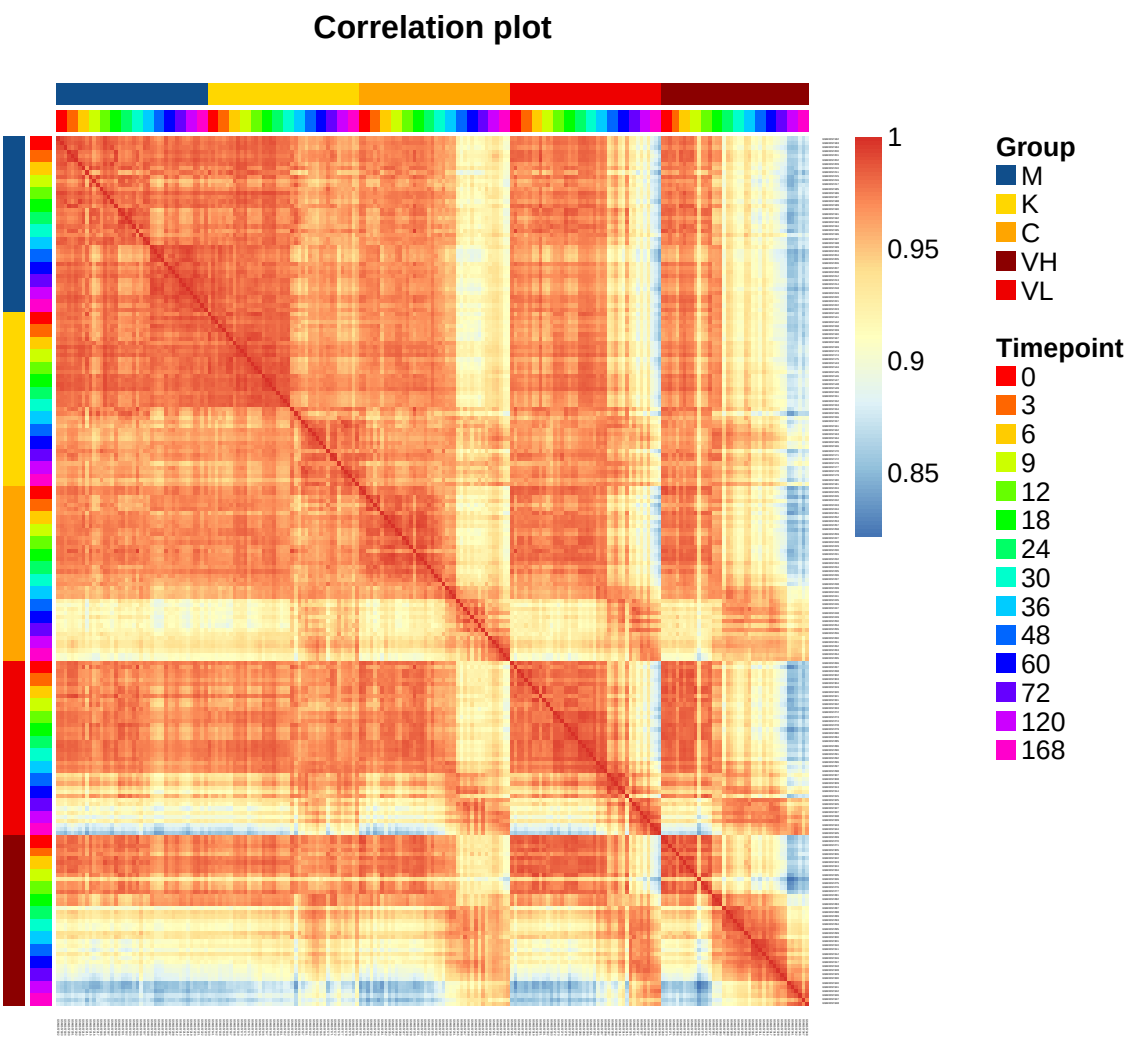
```
plot(
    pca_data$x[, c("PC3","PC2")],
    col=ann_colors$Group[meta$Group],
    pch=ann_markers$Replicate[as.factor(meta$Replicate)],
    ylab="", bty="l", yaxt="n")
axis(2, labels=FALSE)
par(mar=c(4.5, 4.5, 1.5, 1.5))
plot(
    pca_data$x[, c("PC1","PC2")],
    col=ann_colors$Timepoint[as.factor(meta$Timepoint)],
    pch=ann_markers$Replicate[as.factor(meta$Replicate)], bty="l")
mtext(at=275, text="Colored by Timepoint", line=0.5, font=2, adj=0.5)
legend(x=250, y=-15, legend=names(ann_colors$Timepoint),
       fill=ann_colors$Timepoint, bty="n", xpd=NA, yjust=0.5)
par(mar=c(4.5, 5.5, 1.5, .1))
plot(
    pca_data$x[, c("PC3","PC2")],
    col=ann_colors$Timepoint[as.factor(meta$Timepoint)],
    pch=ann_markers$Replicate[as.factor(meta$Replicate)],
    ylab="", bty="l", yaxt="n")
axis(2, labels=FALSE)
```



Then, we plot the pearson correlation across each samples. We order the samples by their Group (treatment)

and Timepoint (the time of sampling). Additionally, in this example, we order each treatment by strength of the pathogenicity of the treatment: Control, Kawasaki, California, low dose of Vietnam, then high dose of Vietnam.

```
# Reorder the conditions such that:
#   - Control is before any influenza treatment
#   - Each treatment is ordered from low to high pathogeny
meta$Group = factor(meta$Group, levels(meta$Group)[c(3, 2, 1, 5, 4)])

# Reorder genes on condition, time, and replicate
ord = order(
  meta$Group,
  meta$Timepoint,
  meta$Replicate)

variance_filtered_data = variance_filtered_data[, ord]
data_corr = cor(variance_filtered_data, method="pearson")
data_corr_meta = meta[ord, ]
data_corr_meta$Timepoint = as.factor(data_corr_meta$Timepoint)
# use aheatmap from the NMF package for a heatmap
aheatmap(
    data_corr,
    Colv=NA, Rowv=NA,
    annCol=data_corr_meta[, c("Group", "Timepoint")],
    annRow=data_corr_meta[, c("Group", "Timepoint")],
    annLegend=TRUE,
    annColors=ann_colors,
    main="Correlation plot")
```



We can already see interesting patterns emerging from the correlation plot. First, the cross-correlation amongst samples taking from the control mice is higher than the cross correlation amongst the rest of the treatments. Second, the influenza-infected mice mildly react until time point 36. Third, the less pathogenic the strain is, the closer the samples are to the control condition. Fourth, the Vietnam samples at time point 120 and 168 are the one that are the most different from control samples.

Differential expression analysis of time-course data

Approaches to DE analysis in time-course data

The next step in a gene expression analysis is typically to run a differential expression analysis, generally to find genes different between different conditions. For time-course data, there are two different approaches for determining differentially expressed genes,

1) Per-time point analysis, where we consider each time point a different condition and determine what genes are changing between time points, or between conditions at a single time-point.

2) Global analysis, where we consider the expression pattern globally over time, and consider what genes have either different patterns between conditions or a changing pattern (i.e. non-constant) over time. A common approach first step is to fit a spline model to each gene [4], and then use that spline model to test for different kinds of differential expression across time.

The per-time point analysis is using classical differential expression approaches, and is often the approach advocated when dealing with small time-course datasets, where there are only a few time points [10, 11, 12]

. For long time-course datasets, however, a separate test for each time-point results in creating many different tests, for example one for every time point, the results of which are difficult to integrate. We find in practice that the global analysis simplifies analysis and interpretation of longer time courses data, with per-time point analysis reserved for particularly interesting comparisons of individual time-points.

Time course data can either be on a single condition (to identify genes changing over time) or on multiple conditions (such as the influenza data set we are considering), which will alter slightly the types of questions we are interested in.

Use with `moanin` `moanin` provides functionality for performing both of these types of approaches, though our focus is on the global approach. In both situations, we first need to set up a object (a `moanin_model` object) to hold the meta data, as well as information for fitting the spline model (formula, number of degrees of freedom of the splines, <80>)

We start by creating the `moanin_model` object using the `create_moanin_model` function. We need to provide two things to the function: a `data.frame` with the metadata, and the number of degrees of freedom of the splines used in the functional modeling. The metadata `data.frame` object should contain at least two columns: one named `Group`, containing the treatement effect, and a second one named `Timepoint` containing the timepoint information.

		Group	Replicate	Timepoint
GSM1557140	0	K	1	0
GSM1557141	1	K	2	0
GSM1557142	2	K	3	0
GSM1557143	3	K	1	12
GSM1557144	4	K	2	12
GSM1557145	5	K	3	12

We create a `moanin_model` for our data:

```
moanin_model = create_moanin_model(meta, degrees_of_freedom=6)
```

The `moanin_model` object contains a number of metadata and options used throughout this analysis: the condition and timepoints of each samples, the formula object used, the basis matrix, and the degrees of freedom of the model.

If no formula is provided, it will default to the following, which provides for a different spline fit for every group (as defined by the `Group` variable in the `meta` data.frame): `formula = ~Group:ns(Timepoint, df=degrees_of_fredoom) + Group + 0`.

The basis matrix contains the value of all of the spline basis function for each sample of the moment. As such, replicate samples will have the same values.

```
kable(head(moanin_model$basis), digits=2) %>%
  kable_styling(font_size = 7)
```

	GroupM	GroupK	GroupC	GroupVL	GroupVH	GroupM:splines::ns(Timepoint, df = degrees_of_freedom)1	GroupK:splines::ns(Time
GSM1557140	0	1	0	0	0	0	
GSM1557141	0	1	0	0	0	0	
GSM1557142	0	1	0	0	0	0	
GSM1557143	0	1	0	0	0	0	
GSM1557144	0	1	0	0	0	0	
GSM1557145	0	1	0	0	0	0	

EAP: General comment: you need a few more print() and head() of some objects to demonstrate what the output looks like

Weekly differential expression analysis

`moanin` provides a simple interface to perform a timepoint by timepoint differential expression analysis. Comparison between groups is traditionally done by defining the group comparisons (called `contrasts` in linear models) as a linear combination of the coefficients of the model. Comparing groups within each timepoint can create many contrasts, and thus `moanin` provides functionality to create these contrasts in an automatic way, and then calls `limma` [10] on the set of contrasts provided. By default, `moanin` expects RNA-Seq contact counts, and will estimate voom weights.

Here, we show an example where we define our contrasts to be the difference between the control mouse ("M") and the mouse infected with the high dose of the influenza strain A/Vietnam/1203/04 (H5N1) ("VL") for each time point, but the function works with any form contrasts [10].

First, create the contrasts for all timepoints between the two groups of interest:

```
# Define contrasts
contrasts = create_timepoints_contrasts("M", "VL", moanin_model)
```

This creates a vector of contrasts to be tested, one for each timepoint, in the format required by `limma`:

Contrasts
M.0-VL.0
M.3-VL.3
M.6-VL.6
M.9-VL.9
M.12-VL.12
M.18-VL.18
M.24-VL.24
M.30-VL.30
M.36-VL.36
M.48-VL.48
M.60-VL.60
M.72-VL.72
M.120-VL.120
M.168-VL.168

Then `moanin` will run the differential expression analysis on all of those timepoints jointly using the function `DE_timepoints`.

```
weekly_de_analysis = DE_timepoints(
     data, moanin_model, contrasts,
     use_voom_weights=FALSE)
```

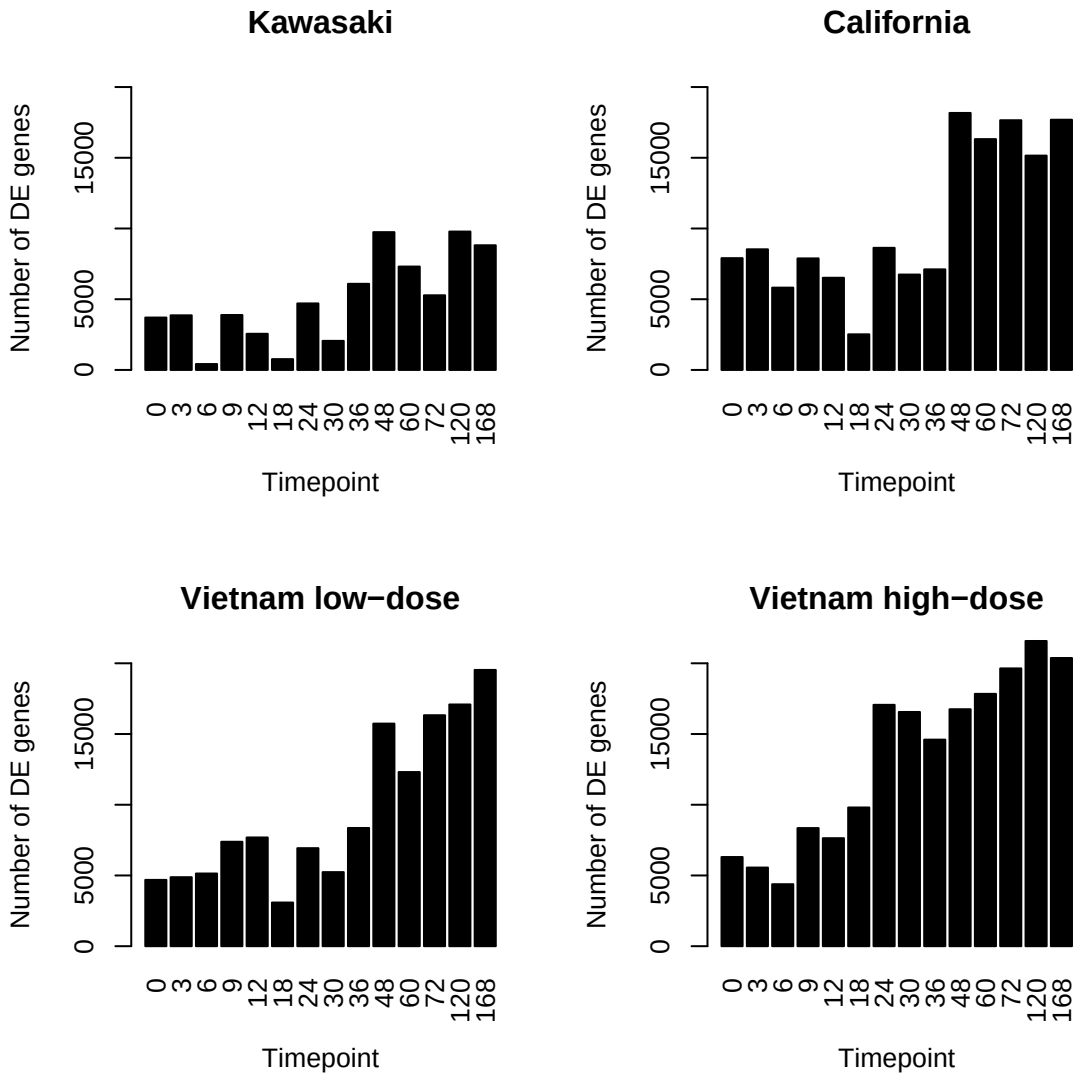
The output is a table of results, where each row corresponds to a gene and the columns correspond to the p-value (`pval`), log-fold change (`lfc`) and adjusted p-value (`qval`) of the sets of contrasts; the order of the genes in the table is the same as the input `data`. Here we show the results for the first timepoint (i.e. first three columns of the output) and the first ten genes:

	M.0-VL.0_pval	M.0-VL.0_qval	M.0-VL.0_lfc	M.3-VL.3_pval	M.3-VL.3_qval	M.3-VL.3_lfc
NM_009912	0.273	0.475	2.202	0.767	0.876	1.158
NM_008725	0.851	0.924	2.028	0.139	0.302	1.457
NM_007473	0.088	0.219	0.678	0.917	0.959	0.764
ENSMUST00000094955	0.008	0.037	1.186	0.935	0.969	-0.633
NM_001042489	0.419	0.621	1.212	0.761	0.872	0.599
NM_008159	0.123	0.278	1.102	0.117	0.268	0.737
NM_001013813	0.057	0.159	0.630	0.171	0.348	0.975
AK039774	0.015	0.059	0.637	0.062	0.171	0.971
NM_013782	0.875	0.938	0.594	0.000	0.001	1.180
NM_028622	0.767	0.876	1.648	0.946	0.974	-1.865

Additional timepoints are in the additional columns of the output.

We will repeat this, comparing each of the remaining three treatments to the control ("M") (code not printed here, as it is a replicate of the above, see accompanying Rmarkdown document).

Let's look at the distribution of genes found differentially expressed per week between control and each of the influenza strains.

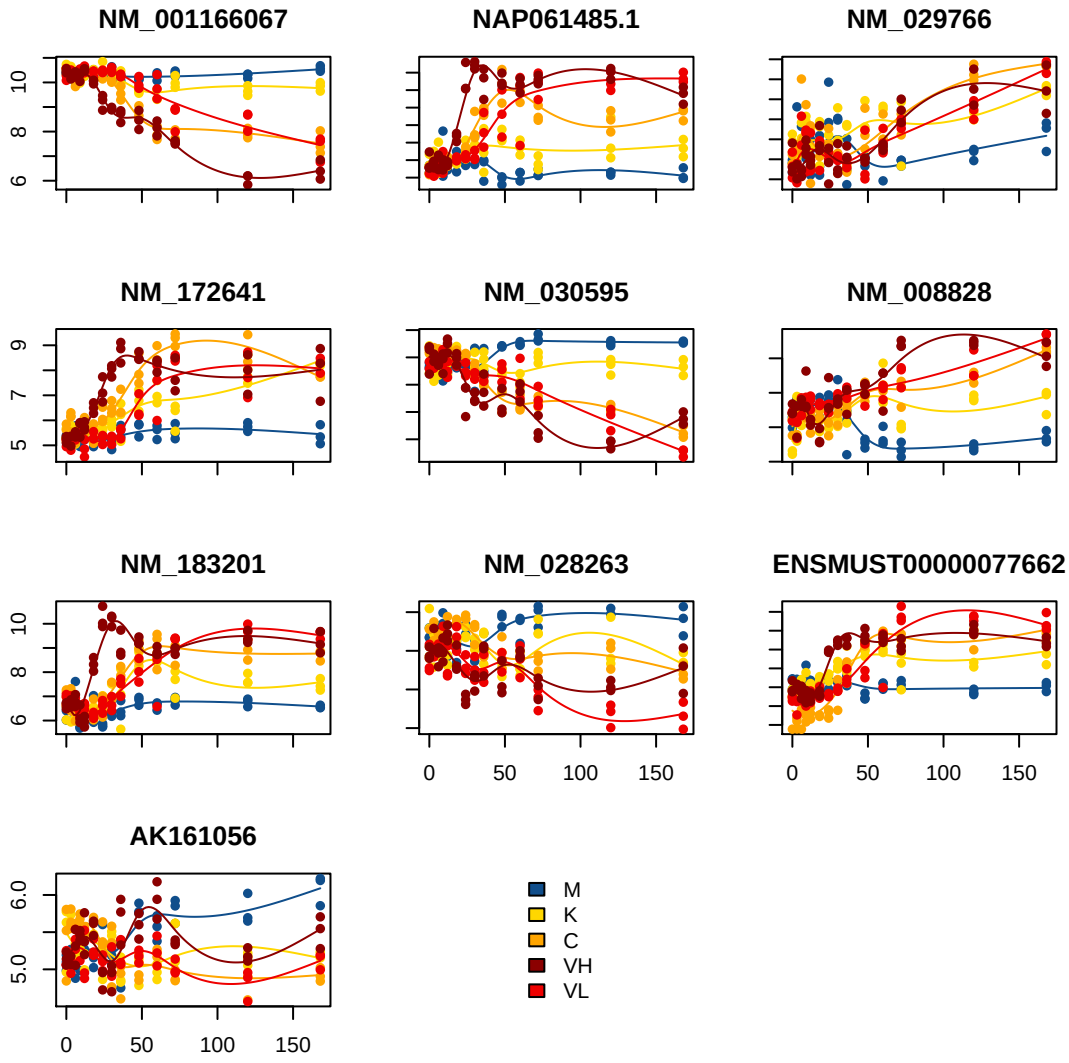


Such an analysis can demonstrate some general trends, with clearly more genes being differentially expressed at later time points, and the Vietnam high-dose showing perhaps an earlier onset than the low-dose.

However, the distribution of the number of genes found differentially expressed by considering each time-point independently highlights the challenges of such approach. We can see that some timepoints have many less genes found significantly differentially expressed (e.g. timepoint 6H and 18H of the Kawasaki strain). While there may be biological differences at those time points for some genes, it seems unlikely that the large majority of genes differentially expressed at timepoint 3H stop being differentially expressed at 6H and then jump back to being differentially expressed at 9H. A more likely explanation is that there is are some technical or biological artifact about the samples for 6H that is creating higher variation and thus less ability to detect significance.

Another difficulty with such an approach is making sense of the general temporal structure for any particular gene, as different genes will have different combinations of timepoints DE. For the comparison of the Kawasaki strain to the control, for example, there are 26534 genes found DE in some timepoints, and there are 1590 different combinations of timepoints for which they are DE. Some of these make sense, such as DE in timepoints 48H-168H (509 genes), but many are very fragmentary. For example there are 330 genes which are DE in timepoints 48,60,120,168H, but *not* in the 72H. Many of these genes are likely to have not made the cutoff for significance in 72H, but don't show real differences in the overall trend between 48H-168H.

Here are plots of the first 10 such genes, where we can see that some might show some meaningful changes between 60 and 72H, but others clearly just have a single replicate that is different or increased variability at 72H.



As a summary, classic differential expression methods are appropriate for unordered treatments, but fail to make use of the temporal structure of the data.

Time-course differential expression analysis between two groups

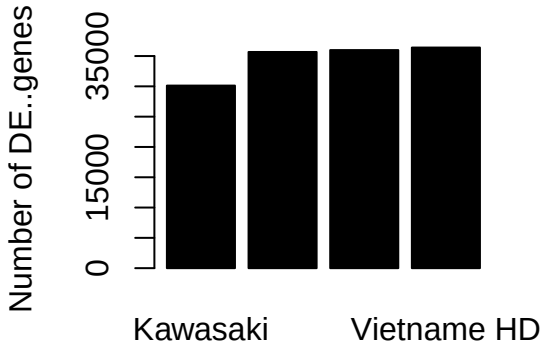
To leverage this temporal structure, Storey et al [4] proposed to model each gene in time-course micro-array with a splines function, and to use a log-ratio likelihood test to detect differentially expressed genes.

`moanin` extends this idea by providing functionality to compare time course data between different treatment conditions, using a similar mechanism of contrasts – only now the contrasts are differences between the estimated mean functions. This is done with the function `DE_timecourse`, which takes as similar input that of `DE_timepoints`, only now will test the entire mean function (and therefore does not require a step of expanding the contrasts into contrasts for individual timepoints).

```
# Differential expression analysis
timecourse_contrasts = c("M-K", "M-C", "M-VL", "M-VH")

# The function takes the data (data.frame or named matrix), the meta data
# (data.frame containing a timepoint and group column, the first corresponding
#<c2><a0>to the time-course information, the latter corresponding to the
# treatment).
DE_results = DE_timecourse(
    data, moanin_model, timecourse_contrasts,
    use_voom_weights=FALSE)
pval_columns = colnames(DE_results)[
    grepl("pval", colnames(DE_results))]
qval_columns = colnames(DE_results)[
```

```
    grepl("qval", colnames(DE_results))]
pvalues = DE_results[, pval_columns]
qvalues = DE_results[, qval_columns]
```

The number of genes found differentially expressed ranges from around 12000 to 29000 depending on the strain and dosage of influenza virus given to the mice. This corresponds to between 30% to 70% of the genes found differentially expressed in this time-course experiment.



The next step in a classical differential expression analysis is typically to assess the effect of the treatment by looking at the log fold change. Computing the log fold change on a time-course experiment is not trivial: one can be interested in the average log-fold change across time, or the cumulative log-fold change. Sometimes a gene can be over-expressed at the beginning of the time-course data, and then over-expressed at the end of the experiment. As a result, `moanin` provides a number of possible ways to compute the log fold change across the whole time-course.

First, `moanin` provides as simple interface to compute the log-fold change for each individual timepoints.

```
log_fold_change_timepoints = estimate_log_fold_change(
    data, moanin_model, timecourse_contrasts,  method="timely")
```

	M-K:1	M-K:7	M-K:11	M-K:14	M-K:2
NM_009912	-0.0061424	0.0224948	-0.0515947	-0.1143973	0.0614619
NM_008725	2.6547855	0.2734387	-0.6882925	0.4718807	-1.7463319
NM_007473	-0.1603623	0.1017655	0.5079573	0.0595773	0.4250185
ENSMUST00000094955	0.3505920	0.6463011	0.2564796	0.3599102	0.4177116
NM_001042489	0.0889750	0.2321381	-0.1891501	0.1786113	0.3320644
NM_008159	-0.3863770	0.0298395	-0.0862755	-0.3266351	-0.1893062

This matrix can then be used to visualize the log-fold change for each contrast per timepoint.

Sometimes, a single value per gene and per contrast is more useful. Here is a table of the possible ways to compute log-fold change values.

Name	Formula	
timely	$\text{lfc}(t)$	Function of time
sum	$\sum_t \text{lfc}(t)$	Sum of log fold change.
abs_sum	$\sum_t \lvert\text{lfc}(t)\rvert$	Always positive
max	$\max_t \lvert\text{lfc}(t)\rvert$	Always positive
min	$\min_t \lvert\text{lfc}(t)\rvert$	Always positive
timecourse	See details below	Captures overall strength and direction

To capture the overall strength and direction of the response, we leverage the timepoint by timepoint log-fold change lfc(t), and apply the following formula:

$$\text{sign}\left(\frac{1}{T}\sum_{t=1}^{T}\text{lfc}(t)\right) \times \left(\frac{1}{T}\sum_{t=1}^{T}|\text{lfc}(t)|\right)$$

Note that when a gene is not consistently up or down-regulated the estimation of the direction will not represent accurately the changes observed.

To compute the log-fold change, `moanin` provides a function taking as argument the data, the `moanin_model` object, the contrasts to evaluate, as well as the method to use to estimate the log-fold change.

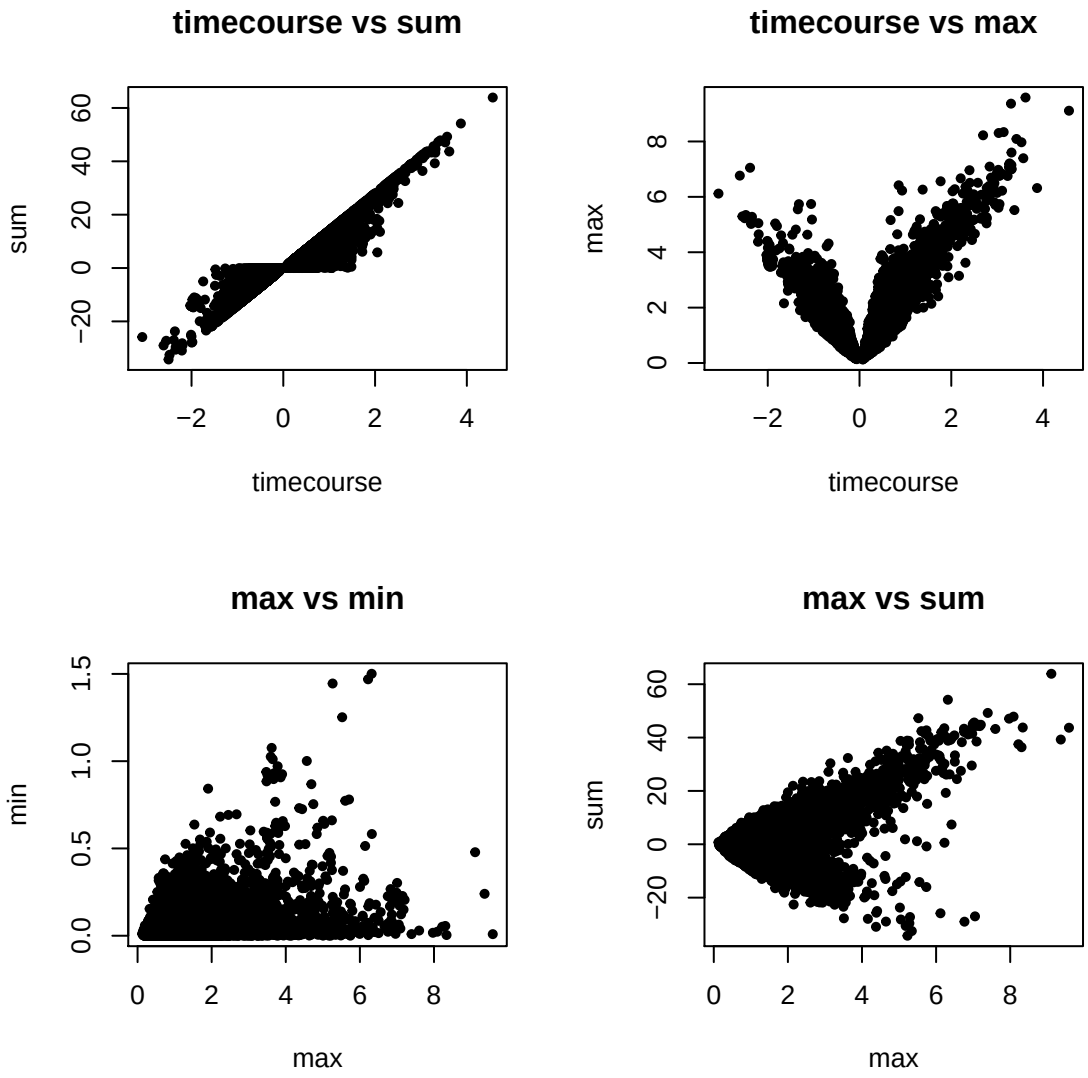
```
log_fold_change_timecourse = estimate_log_fold_change(
    data, moanin_model, timecourse_contrasts,  method="timecourse")

log_fold_change_sum = estimate_log_fold_change(
    data, moanin_model, timecourse_contrasts,  method="sum")

log_fold_change_max = estimate_log_fold_change(
    data, moanin_model, timecourse_contrasts, method="max")

log_fold_change_min = estimate_log_fold_change(
    data, moanin_model, timecourse_contrasts, method="min")
```

The returning object is a matrix, where each row corresponds to a gene, each column to a contrasts, and each entry to the log-fold change for this pair of contrast and gene.

```
kable(log_fold_change_timecourse[1:5, ], digits=3, booktabs=TRUE)
```

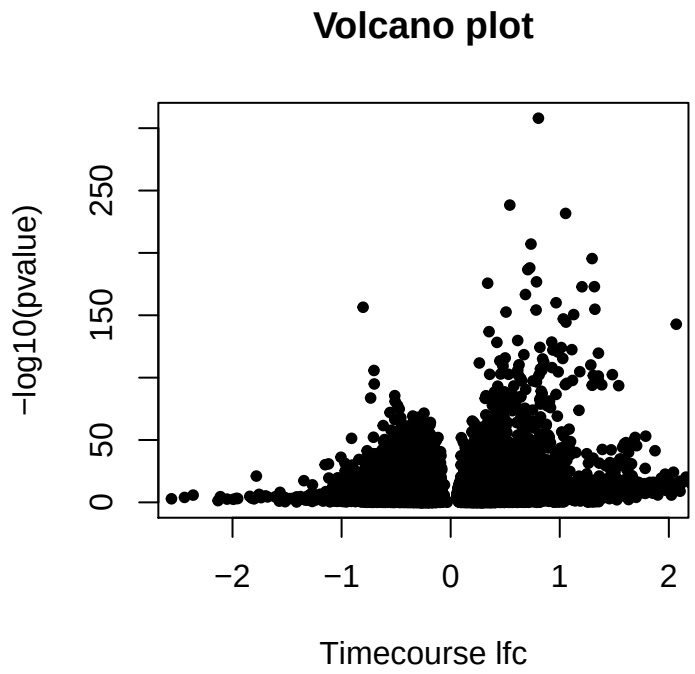
	M-K	M-C	M-VL	M-VH
NM_009912	0.432	0.753	-0.836	-0.488
NM_008725	-1.257	-2.008	1.924	1.729
NM_007473	0.385	0.335	0.516	0.263
ENSMUST00000094955	0.355	0.376	-0.316	0.256
NM_001042489	-0.266	0.410	0.415	0.374

Each methods of computing the log-fold change captures different elements of the time-course data: overall change, largest change, ...



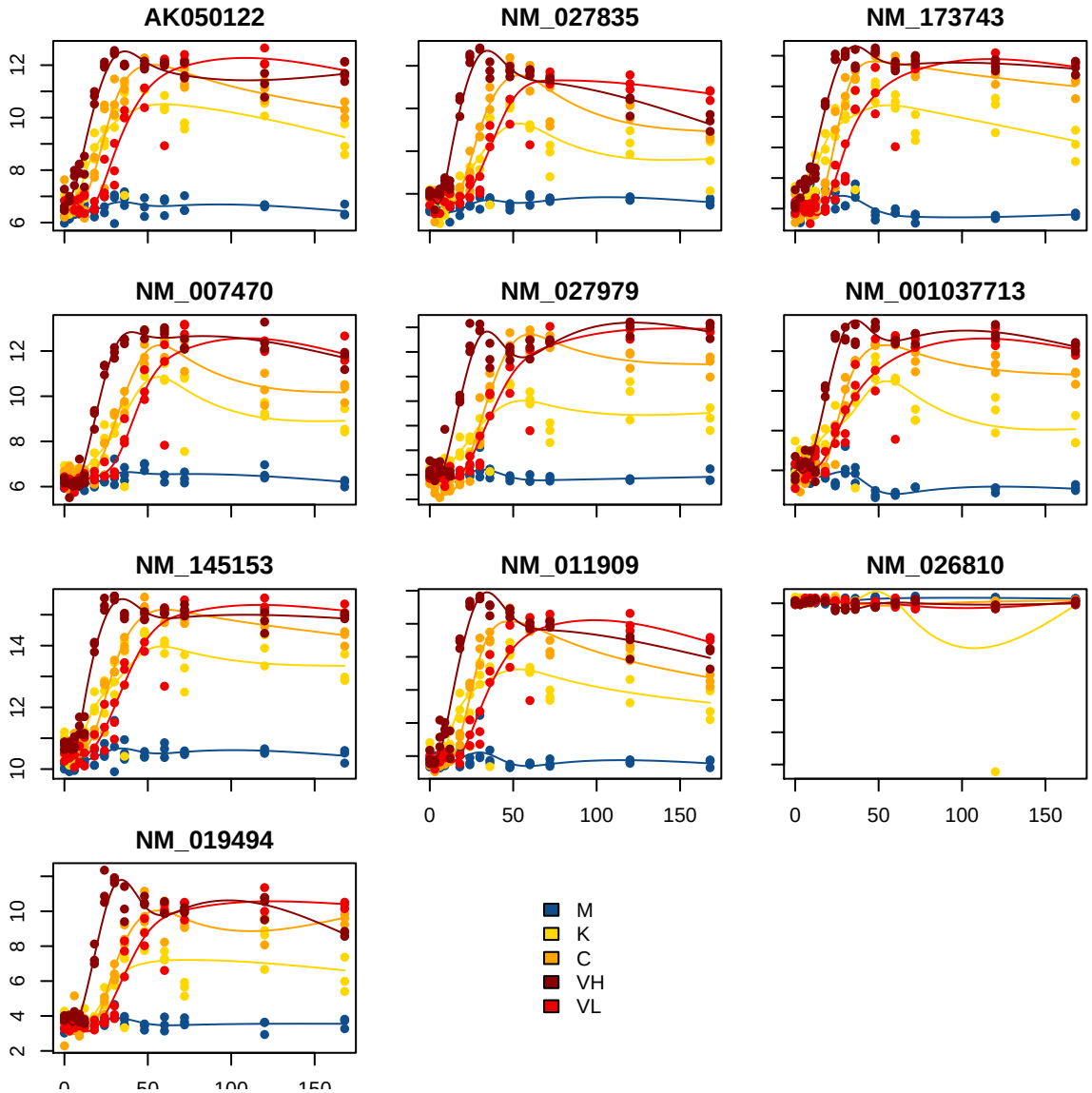
From the single measures of log-fold change and the p-value, we can now look at the volcano plot. Here is an example of a volcano plot for the comparaison of the control to the Kawasaki strain, using the timecourse log fold change computation.

```
pvalue = DE_results[, "M-K_pval"]
names(pvalue) = row.names(DE_results)
lfc_timecourse = log_fold_change_timecourse[, "M-K"]
names(lfc_timecourse) = row.names(log_fold_change_timecourse)

plot(lfc_timecourse, -log10(pvalue), pch=20, main="Volcano plot",
     xlim=c(-2.5, 2), xlab="Timecourse lfc")
```



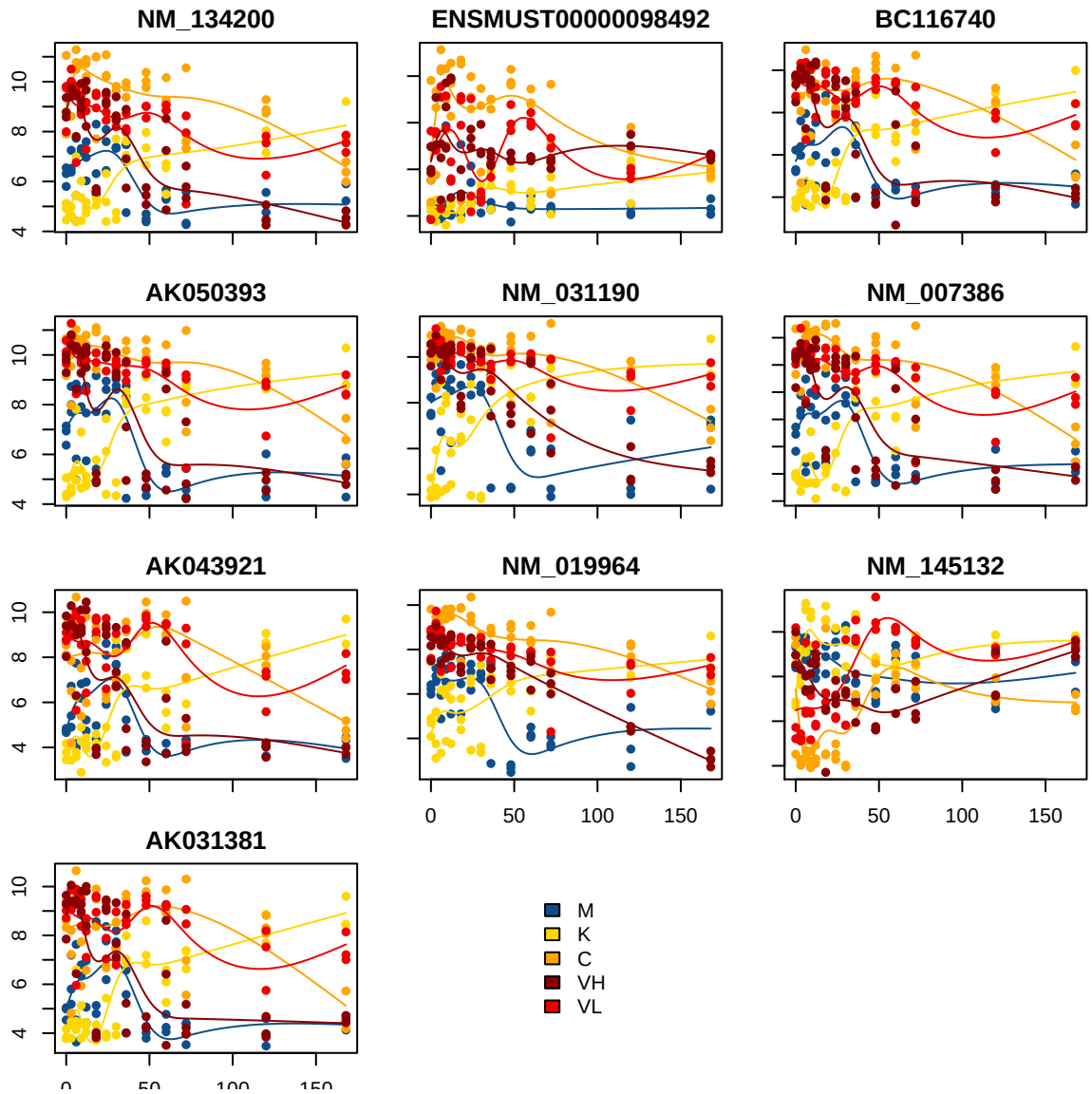
As another sanity check, `moanin` provides a simple utility function to visualize gene time-course data in different conditions. Here, we plot the 10 genes with the smallest p-values.

```
top_DE_genes_pval = names(sort(pvalue)[1:10])
plot_splines_data(data[top_DE_genes_pval, ], moanin_model,
    colors=ann_colors$Group, smooth=TRUE,
    mar=c(1.5,2.5,2,0.1))
```



And here we visualize the genes with the largest absolute timecourse log-fold change.

```
top_DE_genes_lfc = names(
    sort(abs(lfc_timecourse),
     decreasing=TRUE)[1:10])
plot_splines_data(data[top_DE_genes_lfc, ], moanin_model,
    colors=ann_colors$Group, smooth=TRUE,
    mar=c(1.5,2.5,2,0.1))
```



Thanks to those visualization, we can see that genes often follow similar patterns of expression, although on a different scale for each gene. We can leverage this observation to cluster the genes into groups of similar patterns of transcriptomic response.

Clustering of time-course data

The very large number of genes found differentially expressed impair any interpretation one would attempt: with 70% of the genome found differentially expressed, all pathways are affected by the treatment. Hence the next step of the workflow to cluster gene expression according to their dynamical response to the treatment. Before clustering the genes, we first reduce the set of genes of interest to genes that are (1) found significantly differentially expressed; (2) a large-fold change between conditions. To do this, we first aggregate all p-values obtained during the time-course differential expresison step in a single p-value using Fisher's method [13]. Then we select all the genes which have a Fisher adjusted p-value below 0.05 and a log fold change of at least two between at least one condition and one time-point. Reducing the set of genes on which to perform the clustering allows to estimate the centroids with more stability.

```
# Then rank by fisher's p-value and take max the number of genes of interest
# Filter out q-values for the pvalues table
fishers_pval = pvalues_fisher_method(pvalues)
qvalues = apply(pvalues, 2, p.adjust)
fishers_qval = p.adjust(fishers_pval)

genes_to_keep = row.names(
    log_fold_change_max[
```

```
    (rowSums(log_fold_change_max > 2) > 0) &
    (fishers_qval < 0.05), ])
# Keep the data corresponding to the genes of interest in another variable.
y = as.matrix(data[genes_to_keep, ])
```

After filtering, we are left with 3950 genes. We can then apply a clustering. As observed by looking at genes found differentially expressed, many genes share a similar gene expression pattern, but on different scale. We thus propose the following adaptation of k-means:

- **Splines estimation**: for each gene, fit the splines function with the basis of your choice (provided in the `moanin_model` object).
- **Rescaling splines**: for each gene, rescale the estimated splines function such that the values are bounded between 0 and 1 and thus comparable between genes.
- **K-means**: apply k-means on the rescaled fitted values of the splines to estimate the cluster centroids.
- **Assign scores and labels to all genes**: Our `splines_kmeans_score_and_label` function assigns a score to all gene-cluster pairs based on a goodness-of-fit measure of the gene to the cluster centroid. We then "assign" each gene to a cluster based on its score.

The first three steps are performed internally by the `splines_kmeans` function. For now, we will set the number of clusters to be 8, though we will return to the question of picking the best number of clusters below.

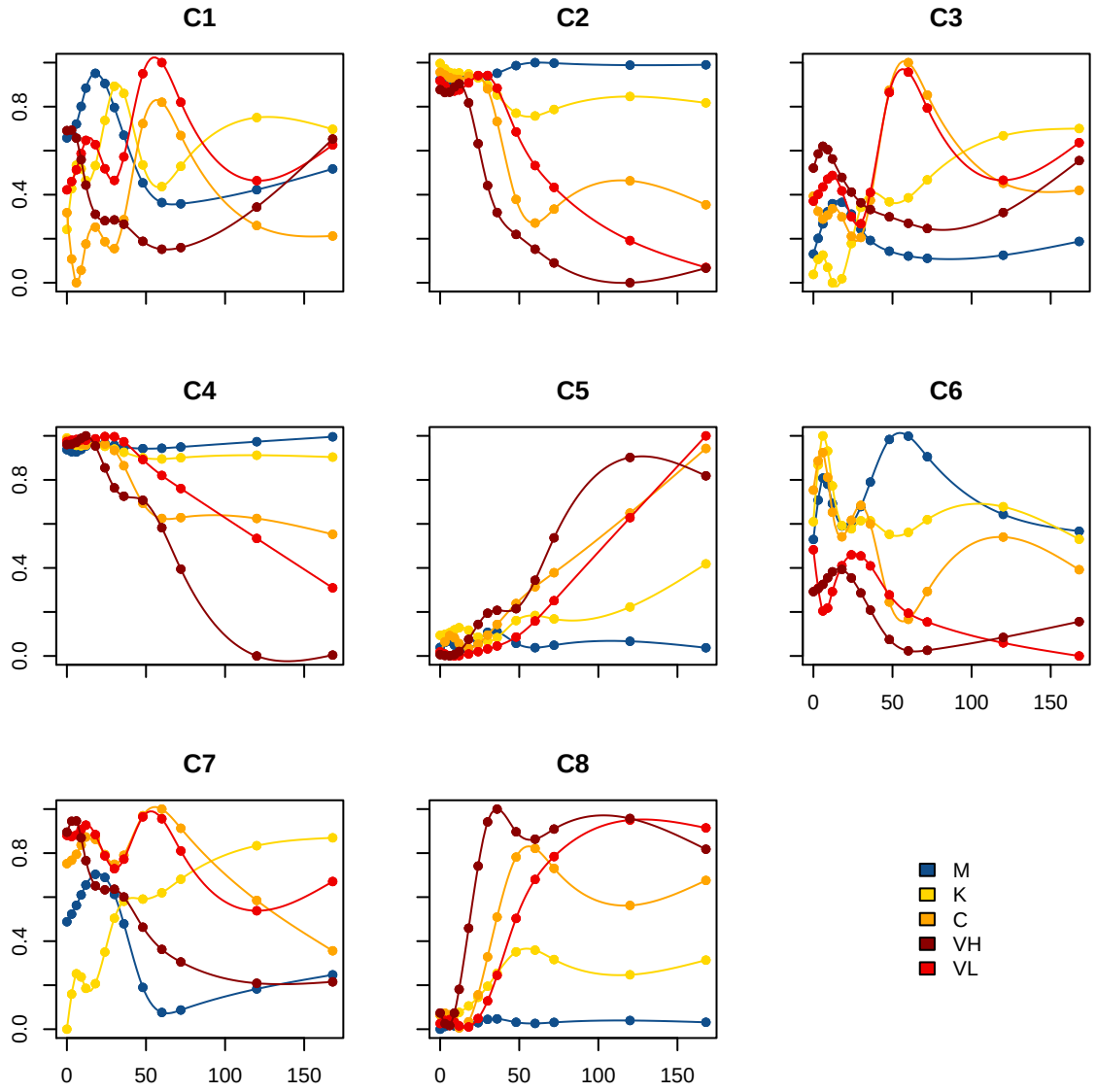
```
# First fit the kmeans clusters
kmeans_clusters = splines_kmeans(
    y, moanin_model, n_clusters=8,
    random_seed=42,
    n_init=20)
```

The `splines_kmeans` function returns a named list with:

- `centroids`: a matrix containing the cluster centroids. The matrix is of shape (`n_centroid`, `n_samples`).
- `clusters`: a vector of size `n_genes`, containing the

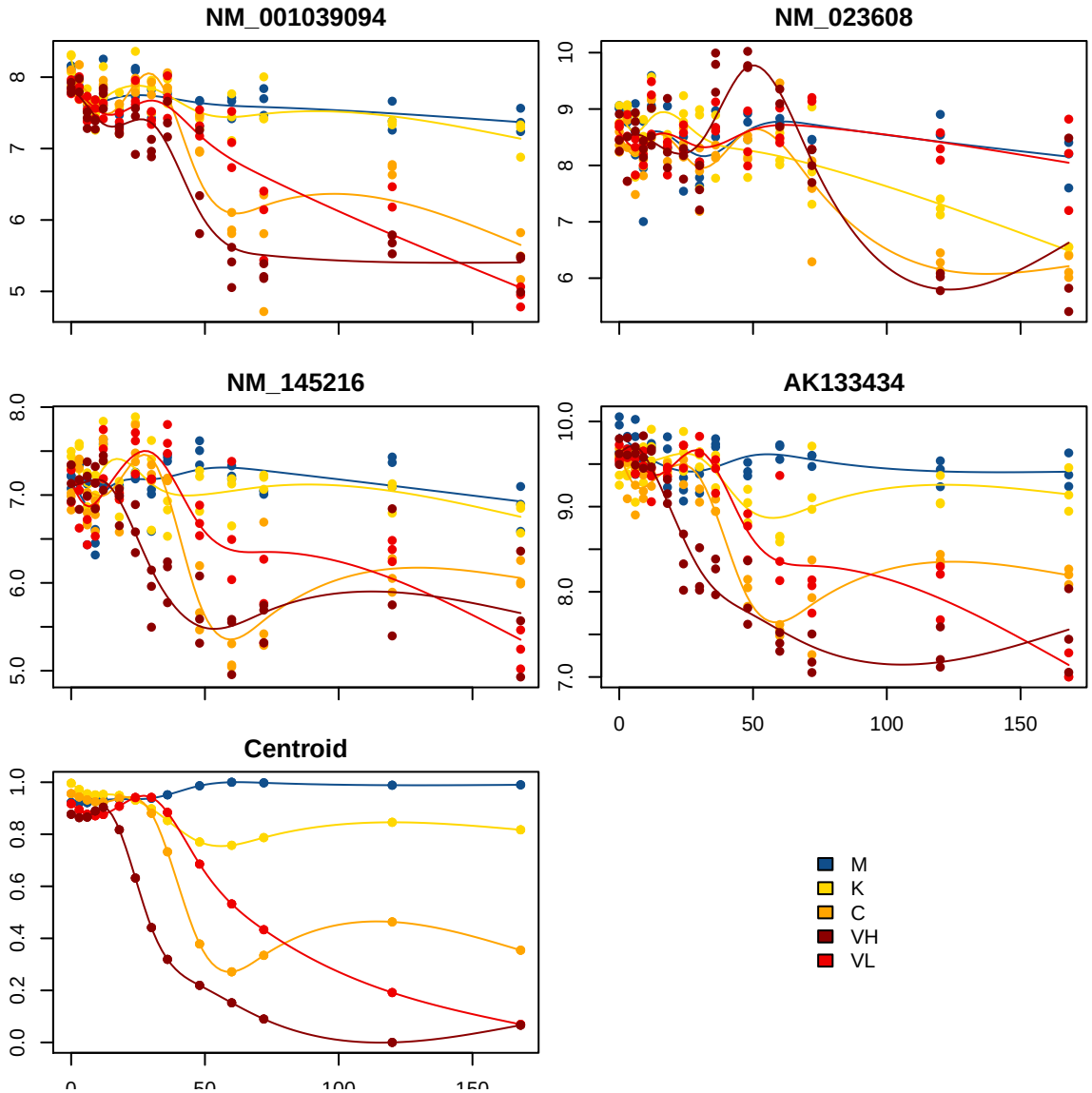
We then use the `plot_splines_data` function to visualize the centroids of each cluster obtained with the splines k-means model.

```
plot_splines_data(
    kmeans_clusters$centroids, moanin_model,
    colors=ann_colors$Group,
    smooth=TRUE)
```



These centroids are on a 0-1 scale, because of our rescaling of the spline fits, and do not represent the actual gene expression level. We can plot a few of the actual genes in a cluster to see the difference:

```
cluster_of_interest = 2
cluster2Genes = names(
    kmeans_clusters$clusters[kmeans_clusters$clusters==cluster_of_interest])
dataToPlot=rbind(y[cluster2Genes[3:6], ],"Centroid"=kmeans_clusters$centroids[cluster_of_interest,])
plot_splines_data(dataToPlot, moanin_model,
    colors=ann_colors$Group, smooth=TRUE,simpleY =FALSE,
    mar=c(1.5,2.5,2,0.1))
```



The spline plots are fitted per gene in the above plot, and illustrate that the genes assigned to a cluster have differing degrees to which their data fits that of the cluster centroid, and indeed fits that of their individual spline fits.

Assigning genes to clusters

Since we arbitrarily chose a subset of genes based on our filter, we would like all genes to have a chance of assignment to a cluster, as well as a score for whether it is a good assignment. The scoring and label step allows us to assign genes to clusters that were removed during the filtering step, yet are good matches to the centroids found during the clustering.The score corresponds to a goodness-of-fit between each gene and each cluster, computed as follows:

$$S(y_i, \mu_k) = \frac{1}{S_0(k)} \times \min_{a_{iG_j}, b_{iG_j}} \sum_j \left(b_{iG_j} y_{ij} + a_{iG_j} - \mu_k(t_j) \right)^2,$$

where μ_k is the centroid of cluster k and y_i the gene of interest. The scoring function thus returns a value between 0 and 1, 0 being the best score possible and 1 the worst score possible (no correlation between the gene and the cluster centroid).

The scoring and labeling is done via the `splines_kmeans_score_and_label` function. This function calculates the goodness of fit of the gene to the cluster centroid. By default, the function labels only the 50% of genes with the best scores, with the remaining genes getting `NA` as their assignment.

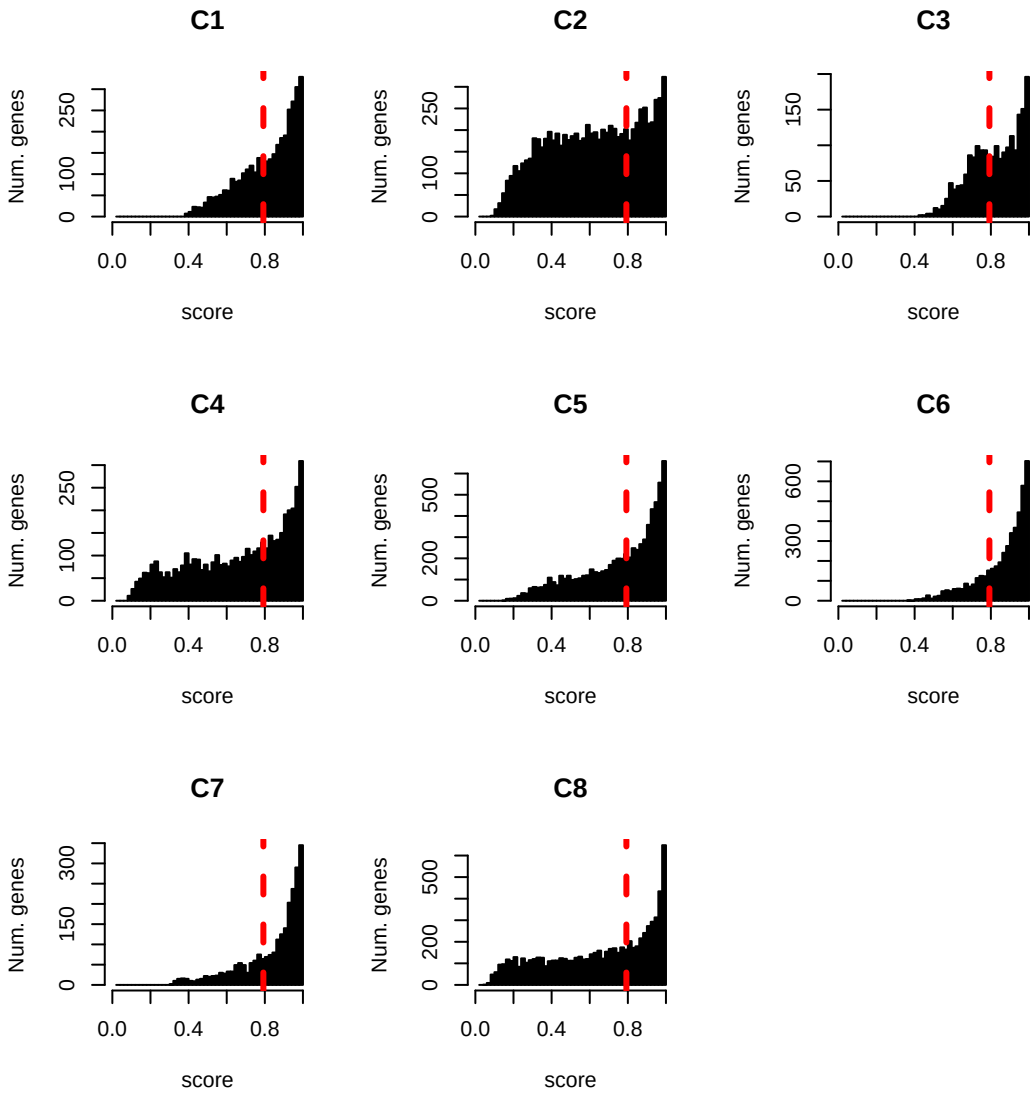
```
#<c2><a0>Then assign scores and labels to all the data, using a goodness-of-fit
# scoring function.
scores_and_labels = splines_kmeans_score_and_label(
    data, kmeans_clusters)
```

The `scores_and_labels` list contains two elements:

```
- ‘scores_and_labels$scores‘: the matrix of shape ‘n_cluster‘ x ‘n_genes‘,
  containing for each gene and each cluster, the goodness of fit as
  described above.
- ‘scores_and_labels$labels‘: the labels for all of the genes with a
  good-enough goodness-of-fit score.
```

First, let us visualize the distribution of goodness-of-fit scores for each cluster. Here, we display the whole set of scores assigned to each cluster, prior to filtering poor-quality fits.

```
# Get the best score and best label for all of the genes
scores = rowMin(scores_and_labels$scores)
labels = apply(scores_and_labels$scores, 1, which.min)
max_score = quantile(scores, 0.5)
par(mfrow=c(3, 3))
n_clusters = dim(kmeans_clusters$centroids)[1]
for(cluster_id in 1:n_clusters){
    hist(scores[labels == cluster_id],
     breaks=(1:50/50), xlim=c(0, 1),
     col="black", main=paste("C", cluster_id, sep=""),
     xlab="score", ylab="Num. genes")
    abline(v=max_score, col="red", lwd=3, lty=2)
}
```

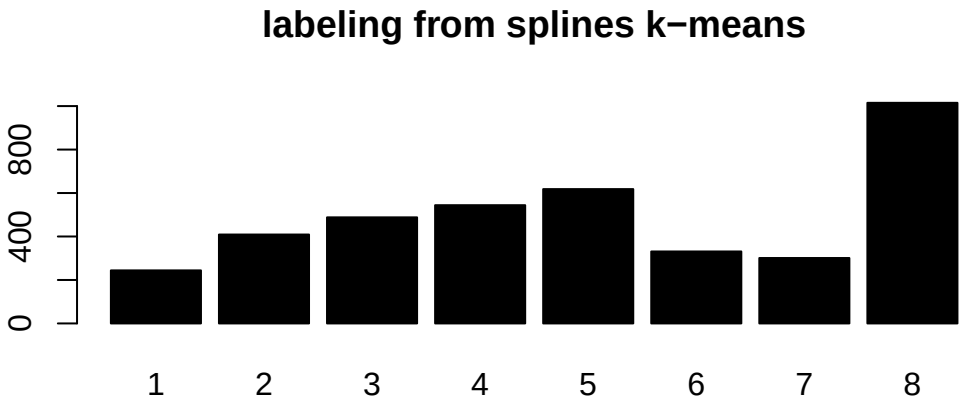


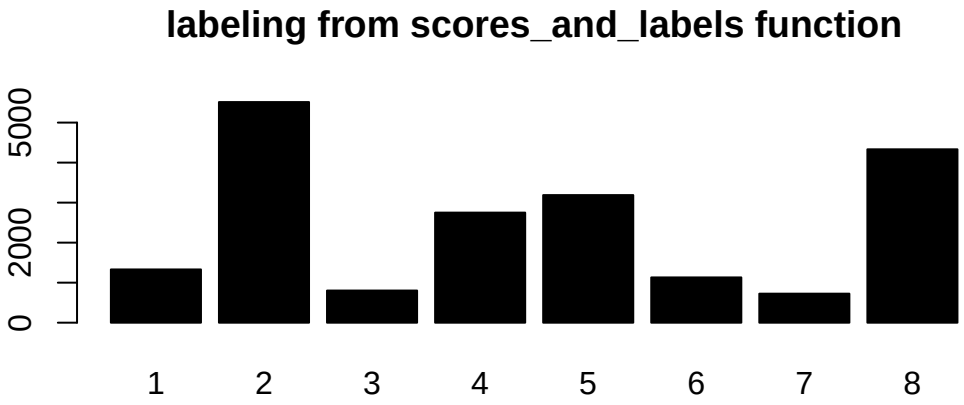
Note that for some of the genes, the best scores is 1. A score of 1 indicates that there is no correlation between the gene and the cluster: those genes fit poorly to all clusters and the assignment to one cluster is done randomly.. This underlines the importance of filtering genes that fit poorly to clusters.

```
labels = scores_and_labels$labels

# Let's keep only the list of genes that have a label.
labels = unlist(labels[!is.na(labels)])

# And also keep track of all the genes in cluster 2.
genes_in_cluster2 = names(labels[labels==cluster_of_interest])
```

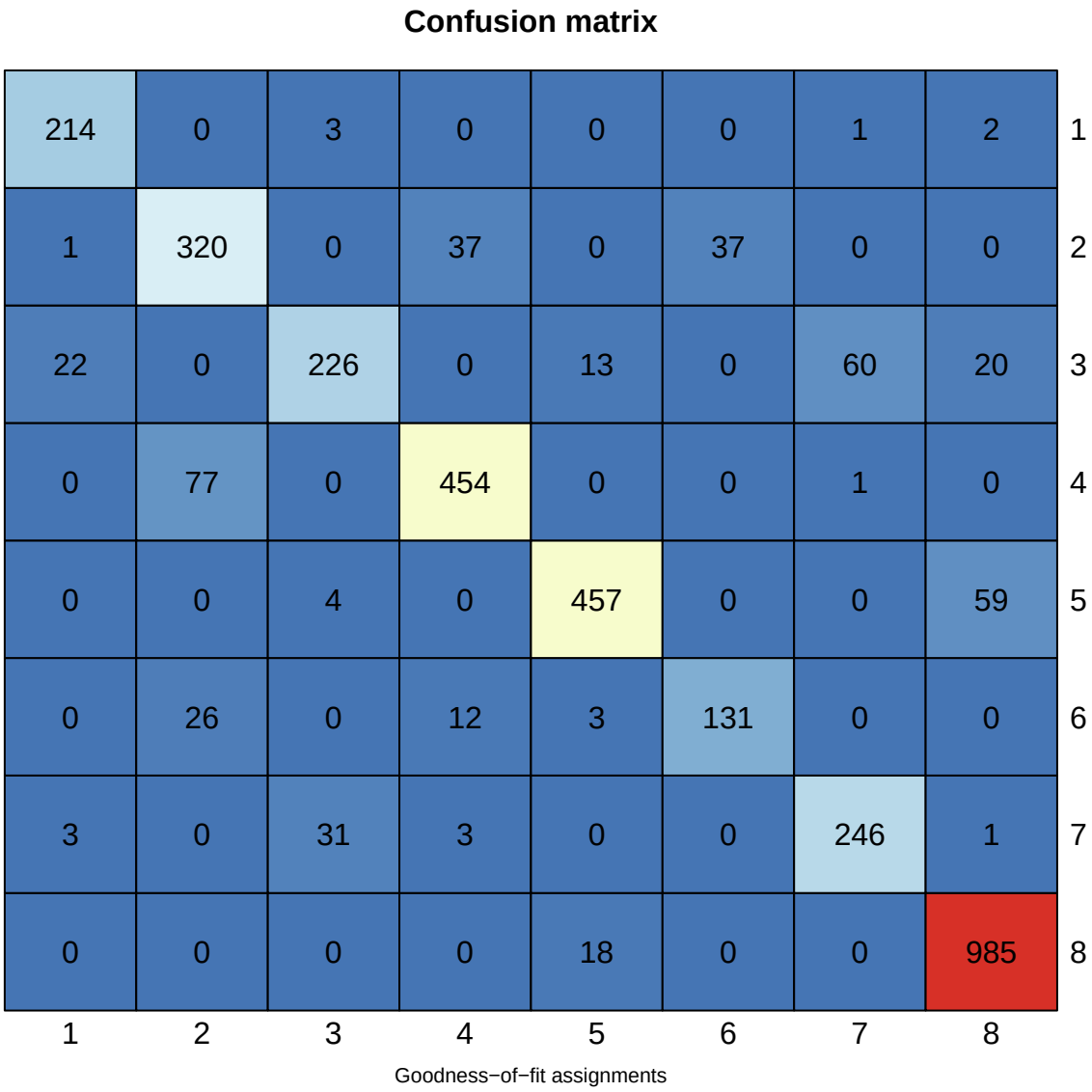
After running `scores_and_labels` we now have 19772 genes that are assigned to a cluster.

Before performing the next steps, let us investigate in more detail the differences between the labels provided by the splines k-means model and the scoring and labeling step.



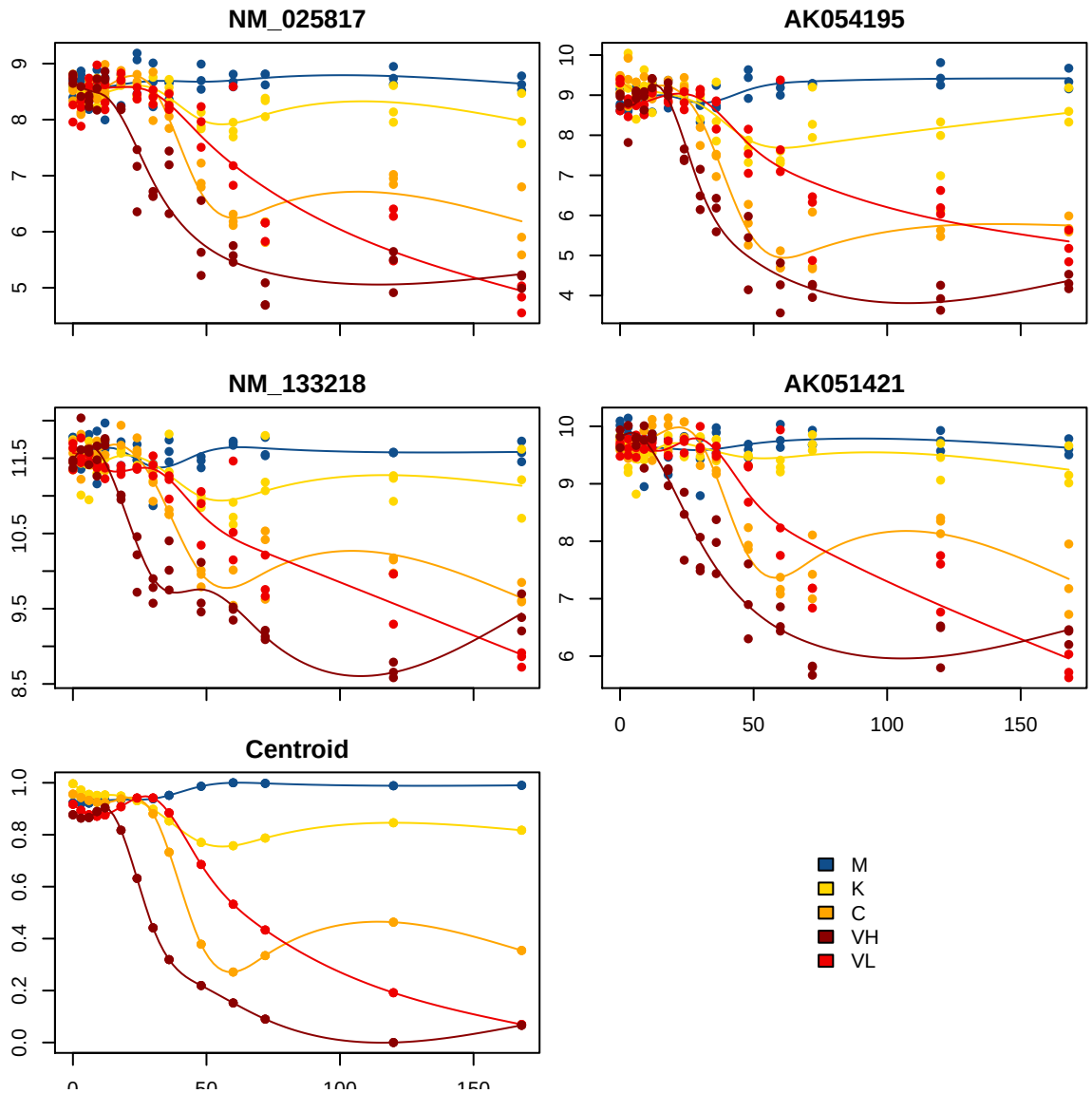


Of the original 3950 genes used in the clustering, 3467 are still assigned to a cluster based on their goodness of fit score. We can compare whether they are still assigned to the same clusters based on a confusion matrix shown below. The kmeans cluster assignments are shown on the rows, and the goodness-of-fit assignments in the columns, with the number in each cell indicating the number of genes in the intersection of the two clusters.



Of the four genes we plotted before, 3 of them remained assigned to cluster 2. We can again look at a few genes in cluster 2, only now pick the best scoring genes

```
# order them by their score
cluster2Score = genes_in_cluster2[order(scores_and_labels$scores[genes_in_cluster2, cluster_of_inter
dataToPlot = rbind(
    data[cluster2Score[1:4], ],
    "Centroid"=kmeans_clusters$centroids[cluster_of_interest, ])
plot_splines_data(dataToPlot, moanin_model,
    colors=ann_colors$Group, smooth=TRUE,simpleY =FALSE,
    mar=c(1.5,2.5,2,0.1))
```



Looking at specific clusters in details.

Now, let us look more in detail some specific clusters. Cluster 8 seems particularly interesting: it captures genes with strong differences between the different influenza treatments and the control, while the control remains relatively flat.

We've already shown how we can plot a few example genes, however, it can be hard to make sense of individual genes given the amount of noise, as well as being hard to draw conclusions based on a few genes.

Heatmaps are useful to investigate the range of expression patterns for specific genes. Here, we are going to plot heatmaps of the normalized gene expression patterns and the rescaled gene expression patterns side by side.

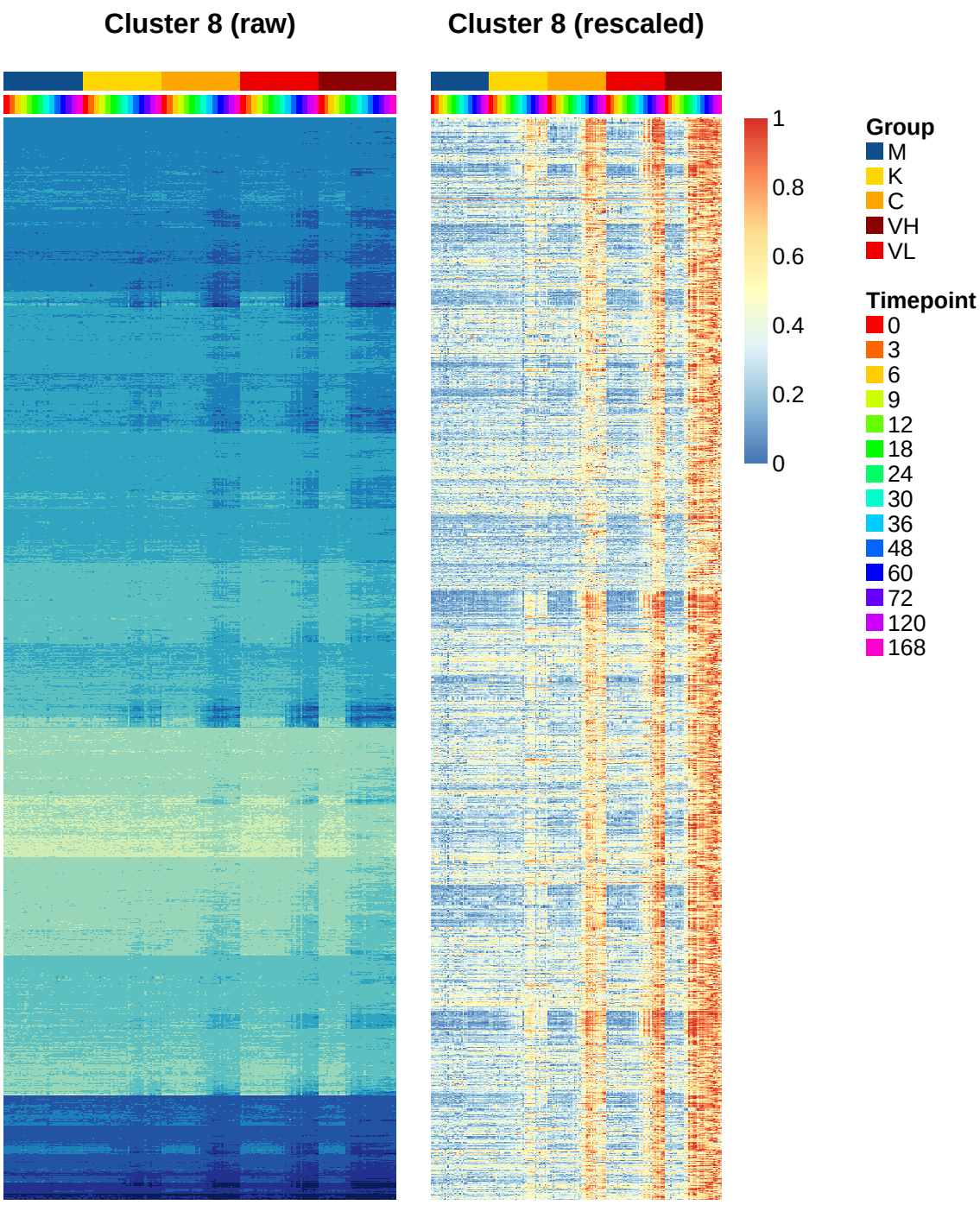
First, we select the genes of interest, namely those in cluster 8, based on our goodness-of-fit assignment.

```
cluster_to_plot = 8
genes_to_plot =  names(labels[labels == cluster_to_plot])
```

Now we will create the heatmaps of these genes (4332 genes).

```
layout(matrix(c(1,2),nrow=1), widths=c(1.5, 2))

data_to_plot = data[genes_to_plot, ]
submeta = meta
ord = order(
```

```
  submeta$Group,
  submeta$Timepoint,
  submeta$Replicate)
submeta$Timepoint = as.factor(submeta$Timepoint)

data_to_plot = data_to_plot[, ord]
submeta = submeta[ord, ]

res = aheatmap(
    data_to_plot,
    Colv=NA,
    color="YlGnBu",
    annCol=submeta[,
                c("Group", "Timepoint")],
    annLegend=FALSE,
    annColors=ann_colors,
    main=paste("Cluster", cluster_to_plot, "(raw)"),
    treeheight=0, legend=FALSE)

# Now use the results of the previous call to aheatmap to reorder the genes.
aheatmap(
    moanin:::rescale_values(data_to_plot)[res$rowInd,],
    Colv=NA,
    Rowv=NA,
    annCol=submeta[,
                c("Group", "Timepoint")],
    annLegend=TRUE,
    annColors=ann_colors,
    main=paste("Cluster", cluster_to_plot, "(rescaled)"),
    treeheight=0)
```



Those two heatmaps demonstrate that the clustering method successfully cluster genes that are on different scales, and yet share the same dynamical response to the treatments.

How to choose the number of clusters.

A common question that arises when performing clustering is how to choose the number of clusters. A choice for the number of clusters K depends on the goal. In this particular case, the end goal is not the clustering, but to facilitate interpretation of the differential expression analysis step. As a result, the number of clusters should not exceed the number of gene sets the user wants to interpret. This allows to set a maximum number of clusters. Let us assume here that this number is 20 clusters.

Once the maximum number of clusters is set, several strategies allow to identify the number of clusters:

- **Elbow method**. First introduced in 1953 by Thorndike [14], the elbow methods looks at the total with cluster sum of squares as a function of the number of clusters (WCSS). When adding clusters doesn't decrease the WCSS by a sufficient amount, one can consider stopping. This method thus provides visual aid to the user to choose the number of clusters, but often the "elbow" is hard to see on real data, where the number of clusters is not clearly defined.
- **Silhouette method**. Similarly to the Elbow method, the Silhouette method refers to a method of validation of consistency within clusters, and provides visual aid to choose the number of clusters.
- **Stability methods** Stability methods are more computationally intensive than any other method, as they rely on assessing the stability of the clustering for every k to a small randomization of the data. The user is then invited to choose the number of cluster based on a number of similarity measures.

First, let us run the clustering for all possible clusters of interest. We will, for each clustering, conserve (1) with within cluster sum of squares; (2) the clustering assignement (or label) for each gene.

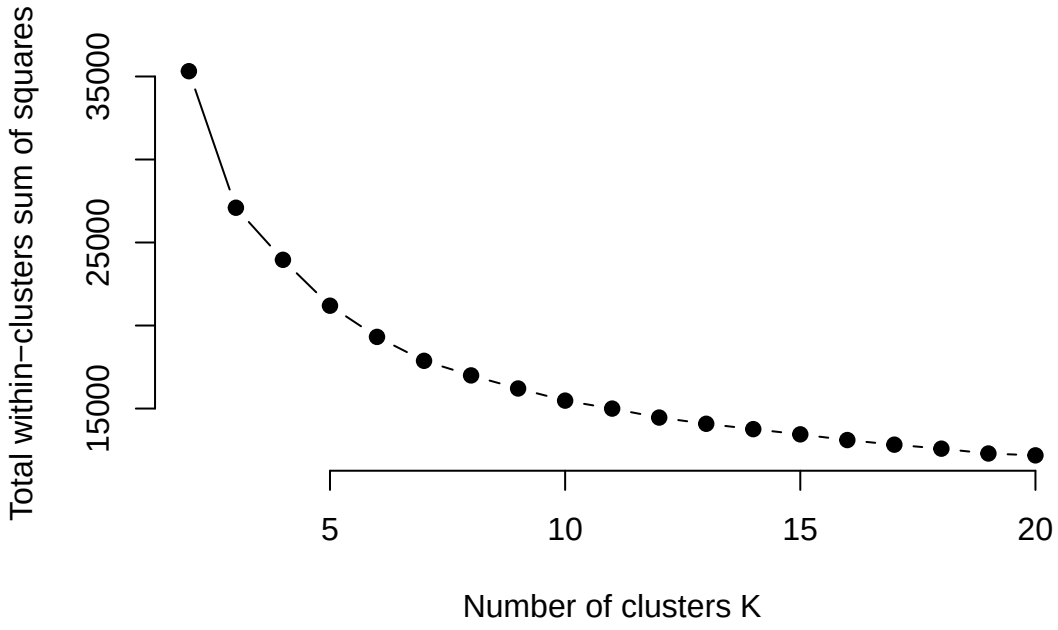
Below we run the clustering for k equal to $2-20$. The `splines_kmeans` function returns the WCSS for each cluster, which we sum to get the total WCSS.

```
all_possible_n_clusters = c(2:20)
all_clustering = list()
wss_values = list()

i = 1
for(n_cluster in all_possible_n_clusters){
    clustering_results = splines_kmeans(
    y, moanin_model,
    n_clusters=n_cluster, random_seed=42,
    n_init=10)
    wss_values[i] = sum(clustering_results$WCSS_per_cluster)
    all_clustering[[i]] = clustering_results$clusters
    i = i + 1
}
```

Elbow method The Elbow method to choose the number of clusters relies on visualization aid to choose the number of clusters. The method relies on plotting the within cluster sum of squares (WCSS) as a function of the number of clusters. At some point, the WCSS will start decreasing more slowly, giving an angle or "elbow" in the graph. The number of cluster is chosen at this "Elbow point."

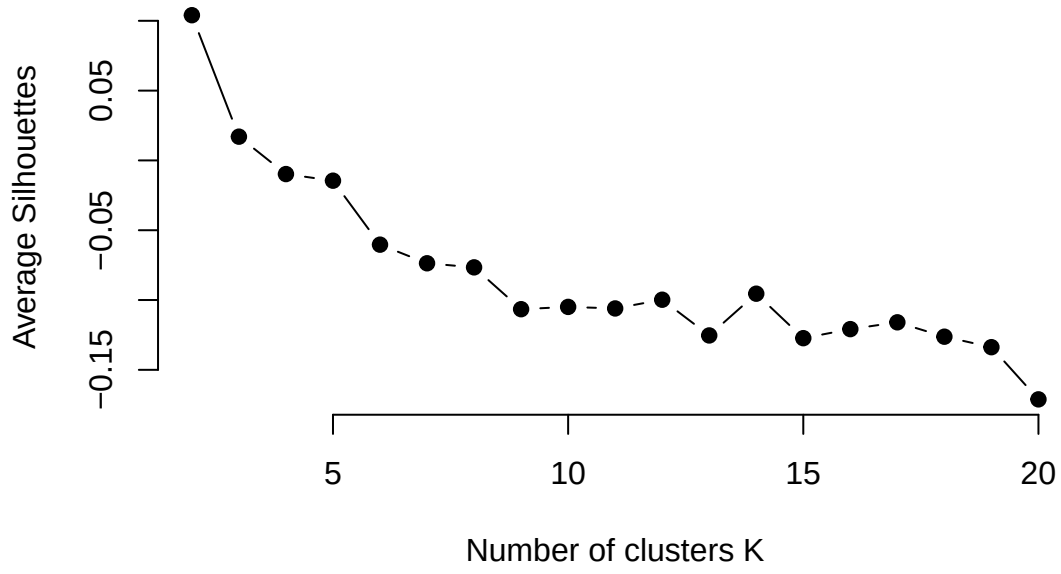
We plot the WCSS for $k = 2-20$ here. We see that as expected the WCSS continues to drop, but there is no clear drop in the decrease, except for very small values of k (3-4 clusters). However, 3-4 seems a very small number of gene clusters to find, given the complexity

```
plot(all_possible_n_clusters, wss_values,
     type="b", pch=19, frame=FALSE,
     xlab="Number of clusters K",
     ylab="Total within-clusters sum of squares")
```



Average silhouette method The silhouette value is a measure of how similar a data point is to its own cluster (cohesion) compared to other clusters (separation).

```
# function to compute average silhouette for k clusters
average_silhouette = function(labels, y) {
    silhouette_results = silhouette(unlist(labels[1]), dist(y))
    return(mean(silhouette_results[, 3]))
}

# extract the average silhouette
average_silhouette_values = list()
i = 1
for(i in 1:length(all_clustering)){
    clustering_results = all_clustering[i]
    average_silhouette_values[i] = average_silhouette(clustering_results, y)
    i = i + 1
}

plot(all_possible_n_clusters, average_silhouette_values,
     type="b", pch=19, frame=FALSE,
     xlab="Number of clusters K",
     ylab="Average Silhouettes")
```



Looking at the stability of the clustering On real data, the number of clusters is not only unknown but also ambiguous: it will depend on the desired clustering resolution of the user. Yet, in the case of biological data, stability and reproducibility of the results is necessary to ensure that the biological interpretation of the results hold when the data or the model is exposed to reasonable perturbations.

Methods that rely on the stability of the clustering results to choose k thus ensure that the biological interpretation of the clusters hold with perturbtation to the data. In addition, simulation where the data is generated with a well defined k show that the clustering is more stable for the correct of number of the clusters.

Most methods method to find the number of clusters with stability measures only provide visual aids to guide the user. The first element often visualized is the consensus matrix: the consensus matrix is an $n \times n$ matrix that stores the proportion of clustering in which two items are clustered together. A perfect consensus matrix ordered such as each elements that belong to the same cluster are adjacent to one another which show blocks along the diagonal close to 1.

To perform such analysis, the first step is run the clustering several times on a resampled dataset–either using bootstrap or subsampling.

Using the bootstrapping strategy, we sample with replacement a sample of the same size as our original clustering (i.e. 3950 genes):

```
n_genes = dim(y)[1]
indices = sample(1:dim(y)[1], n_genes, replace=TRUE)

bootstrapped_y = y[indices, ]
```

Using the subsampling strategy, we take a unique subset of the genes, keeping 80% of the genes:

```
subsample_proportion = 0.8
indices = sample(1:dim(y)[1], n_genes * subsample_proportion, replace=FALSE)
subsampled_y = y[indices, ]
```

We run bootstrap method on all genes differentially expressed and with a log-fold-change higher than 2 (computed with the `lfc_max` method), and do it $B = 20$ times for each of $k = 2 - 20$. We show the code below, but because of the time it takes we have evaluate these values separately and provided the results for users to explore more quickly.

```
pvalues = de_analysis[, pval_col_to_keep]

splines_model = moanin::create_moanin_model(meta, degrees_of_freedom=6)
fishers_pval = moanin:::pvalues_fisher_method(pvalues)
fishers_qval = stats::p.adjust(fishers_pval)

# Keep only the genes found differentially expressed after combining the
# p-values using Fisher's method and with a log fold change greater than 2
genes_to_keep = row.names(
    lfc_max[(rowSums(lfc_max > 2) > 0) & (fishers_qval < 0.05), ])
y = as.matrix(data[genes_to_keep, ])

# You may want to set the random seed of the experiment so that the results
# don't vary if you rerun the experiment.
# set.seed(random_seed)
n_genes = dim(y)[1] * sample_proportion
indices = sample(1:dim(y)[1], n_genes, replace=TRUE)

random_subsample_y = y[indices, ]
kmeans_clusters = moanin::splines_kmeans(
    random_subsample_y, splines_model, n_clusters=n_clusters,
    random_seed=random_seed,
    n_init=20)

# Perform prediction on the whole set of data.
kmeans_clusters = moanin:::splines_kmeans_prediction(
    y, kmeans_clusters)
```

For now, we bring in the results for $k = 5$ and $k = 20$.

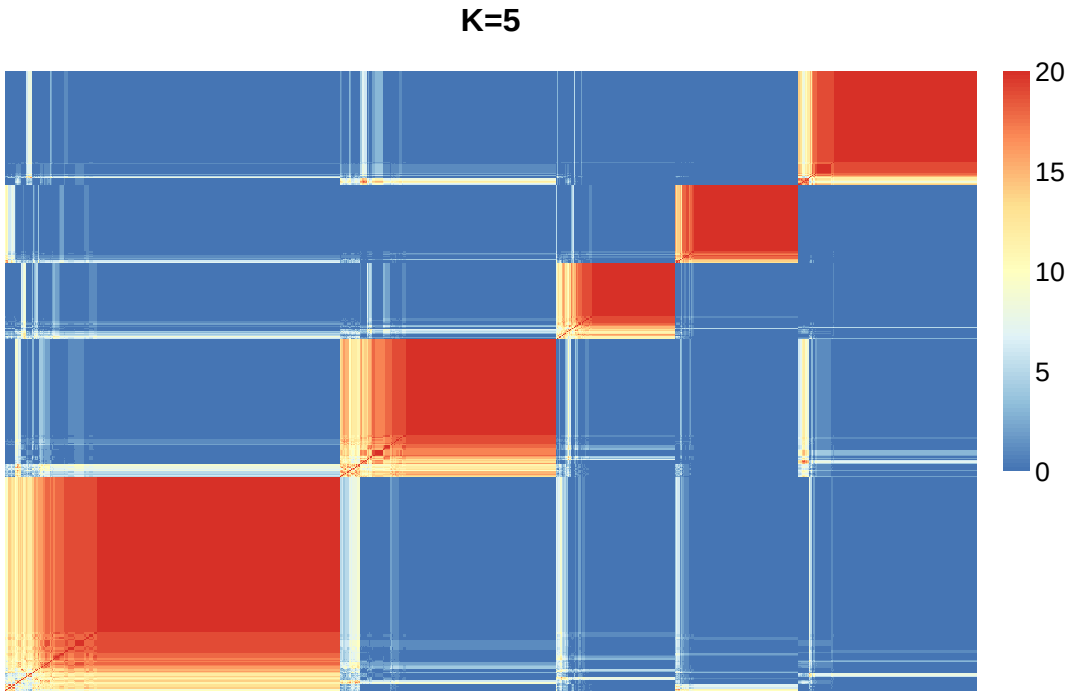
```
stability_5 = read.table("results/stability_5.tsv", sep="\t")
stability_20 = read.table("results/stability_20.tsv", sep="\t")
```

Each column correspond to a bootstrap sample, each row to a gene, and each entry to the label found for that particular clustering. Thus, for each gene, we have an assignment to a cluster over the 20 resampling runs. For example, for $k = 5$:

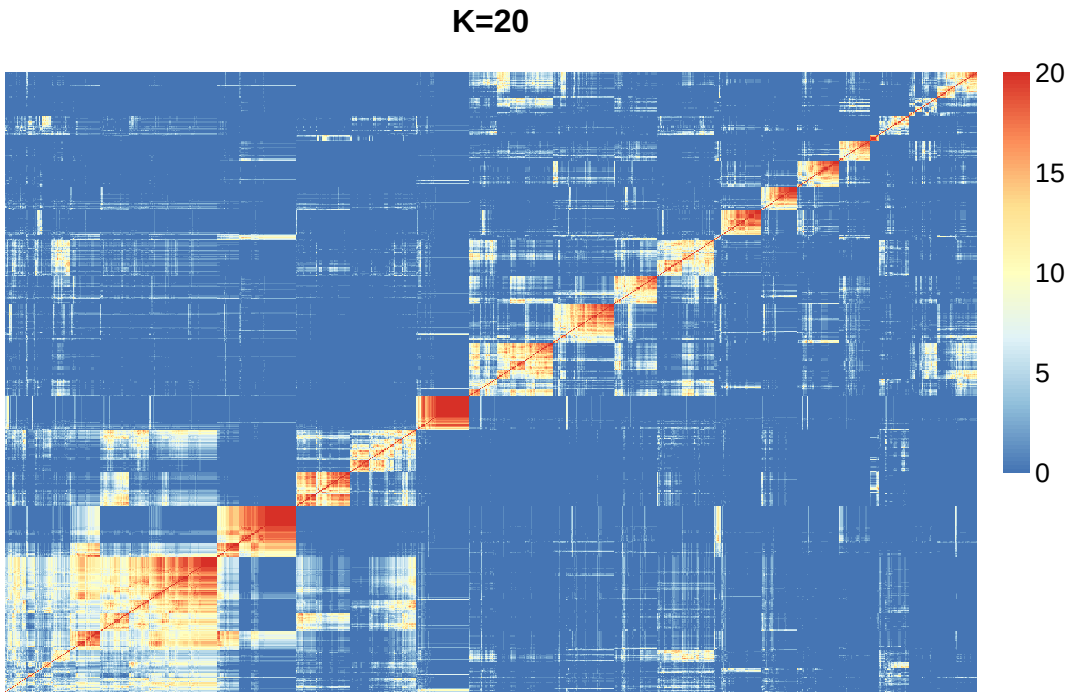
	B1	B2	B3	B4	B5	B6	B7	B8	B9	B10	B11	B12	B13	B14	B15	B16	B17	B
A_51_P114236	4	2	1	2	1	3	3	2	5	5	2	1	4	1	1	2	5	
A_51_P172409	1	5	5	5	5	5	1	3	2	1	3	4	3	3	5	4	2	
A_51_P188401	1	5	5	4	5	5	5	3	2	4	4	4	2	3	5	4	2	
A_51_P191274	4	2	1	2	1	3	3	2	5	5	2	1	4	1	1	2	5	
A_51_P226791	1	4	2	4	3	4	2	1	3	3	4	2	5	2	3	5	2	
A_51_P277048	1	1	4	5	5	2	5	5	2	4	3	4	2	5	4	3	4	

Now we will use the function `consensus_matrix` in the `moanin` package to calculate proportion of times each pair of samples was clustered together across the 20 resampling runs, and plot the heatmap of those consensus matrices for $k = 5$ and $k = 20$.

```
consensus_matrix_stability_5 = consensus_matrix(stability_5,
                          scale=FALSE)
aheatmap(consensus_matrix_stability_5[1:1000, 1:1000], Rowv=FALSE,
     Colv=FALSE,
     treeheight=0, main="K=5")
```




```
consensus_matrix_stability_20 = consensus_matrix(stability_20,
                            scale=FALSE)
aheatmap(consensus_matrix_stability_20[1:1000, 1:1000], Rowv=FALSE,
     Colv=FALSE,
     treeheight=0, main="K=20")
```



We can see that the choice of $K = 5$ seems much more stable across resampling runs than that of $K = 20$.

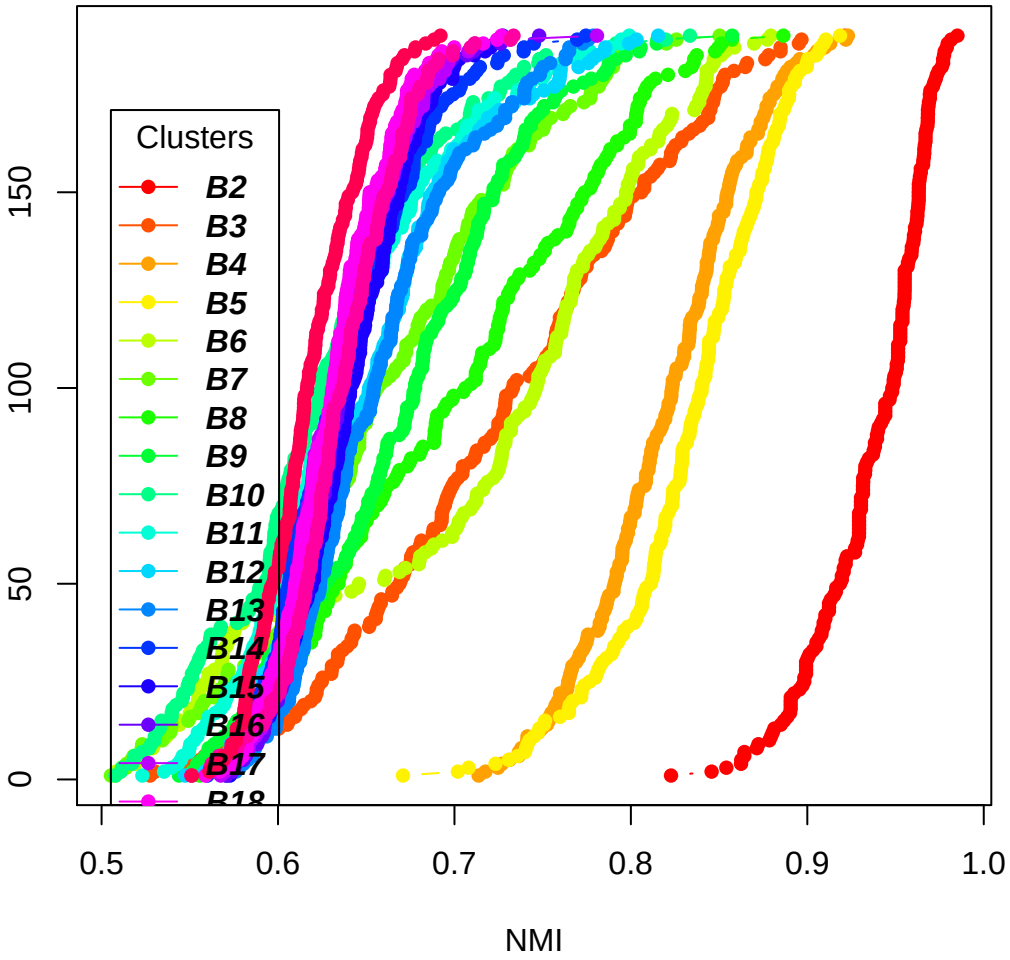
The model explorer strategy The model explorer algorithm [15] proposes to estimate the number of clusters exploiting the observation that if the number of clusters is correct, the clustering results are stable to bootstrap resampling, as described above. The distribution of similaries between bootstrapped results for each k can thus be compared for different values of k and guide the user in the choice of number of clusters.

The model explorer strategy works as follows. For a single choice of k, first perform n bootstrap experiments to estimate the cluster centroids, followed by a step of assigning a label to all data points. Then, choose a similarity measure between two partitions or clusters $S(B_1, B_2)$. Examples are the normalized mutual information or Fowlkes-Mallows. Finally, compute the pairwise similarity measure between all bootstrapped partitions (i.e. B choose 2 pairs). Repeat this procedure for different k and plot per k the cumulative density of the obtained scores.

We have already run the bootstrap resampling for values $k = 2-20$ and saved the results. We will read that data in, and use it to calculate the pairwise similarity scores.

The function `plot_model_explorer` takes in input a list of the bootstrapped results for all labels.

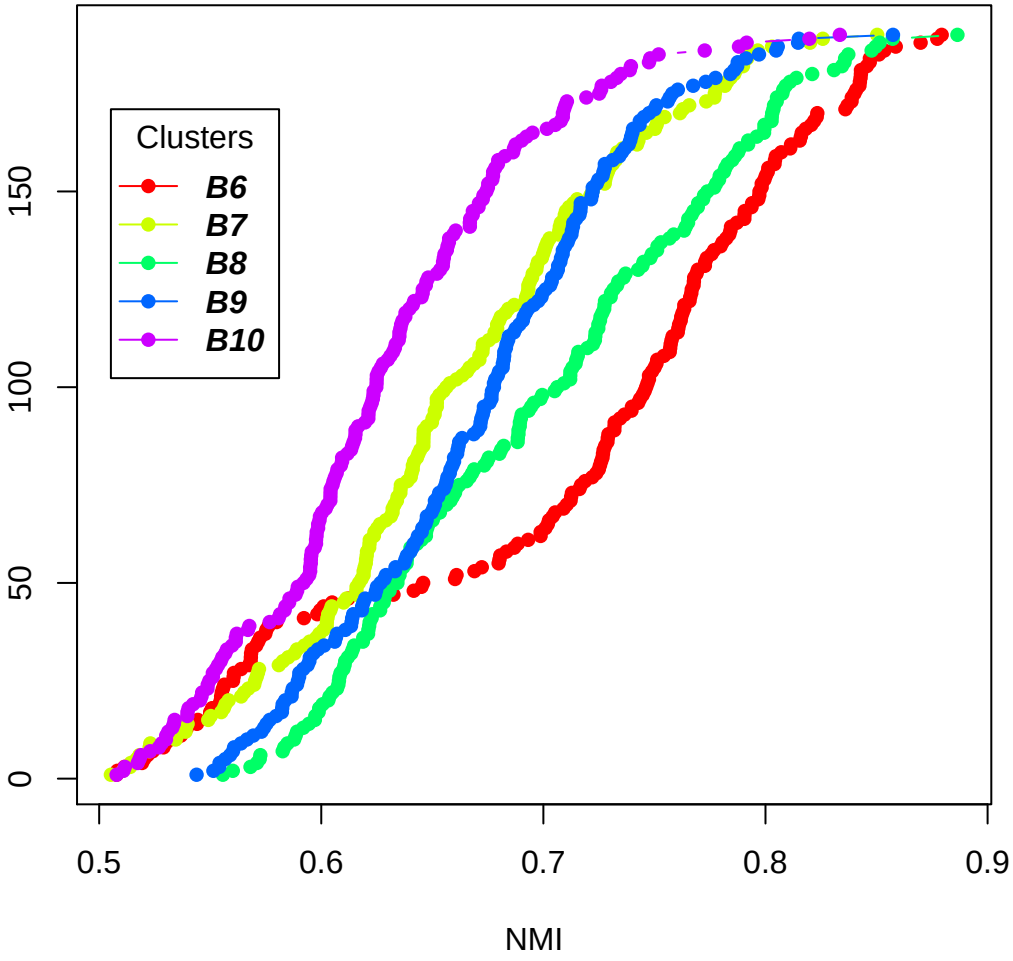
```
plot_model_explorer(all_labels)
```



From this plot, we can deduce that $k = 5$ is more stable than $k = 3$ and $k = 4$, but not as stable as $k = 2$. The model explorer strategy, in addition to visualizing the diversity of the centroids, can thus help assessing an adequate number of clusters.

Now, replot the same model explorer, but only for the clustering experiments $k = 6$, $k = 7$, $k = 8$, $k = 9$ and $k = 10$ so that we can see more clearly the stability measures in that range.

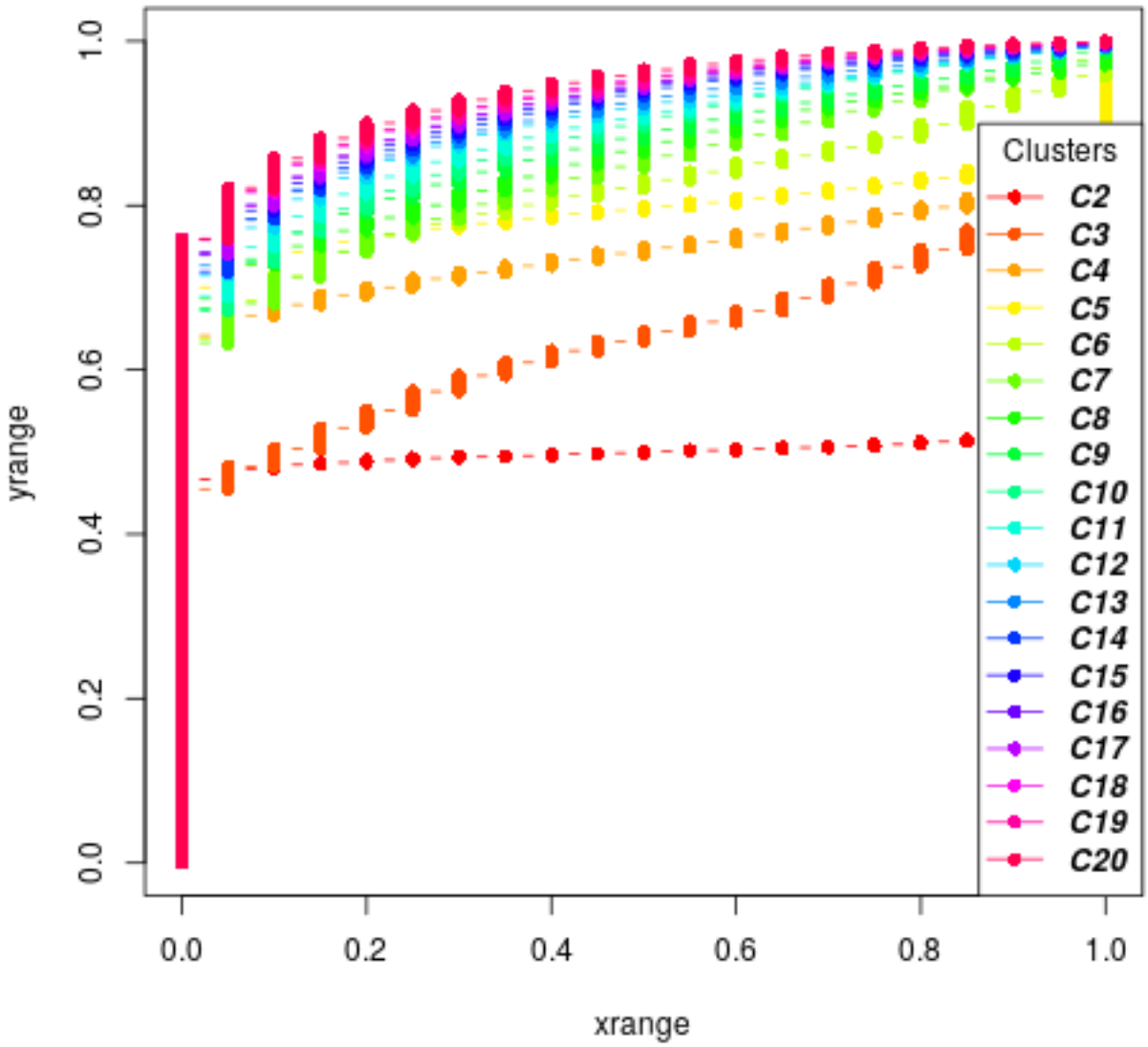
```
clusters = c("B6", "B7", "B8", "B9", "B10")
selected_labels = all_labels[clusters]
plot_model_explorer(selected_labels)
```



This analysis doesn't necessarily pick a particular k, but can help decide between k within a desired range, for example, or to avoid k that degrade the stability.

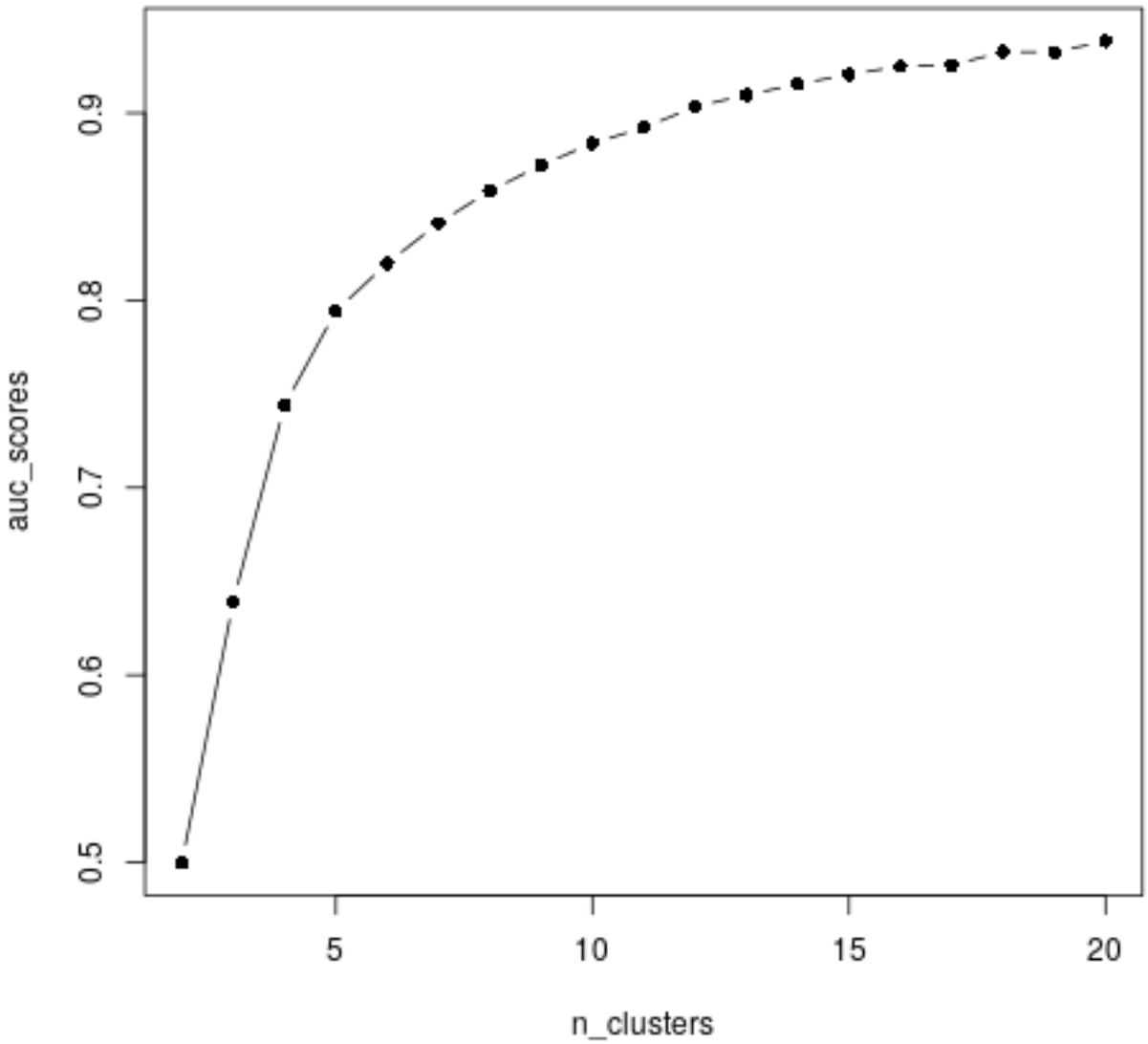
Consensus clustering as a way to find k The consensus clustering [16] relies on a similar idea but instead of looking at the cumulative density of similarity measures of bootstrapped clustering, the authors suggests plotting the cumulative density of elements of the consensus matrix. We provide a function `plot_cdf_consensus` in `moanin` to do this

```
plot_cdf_consensus(all_labels)
```

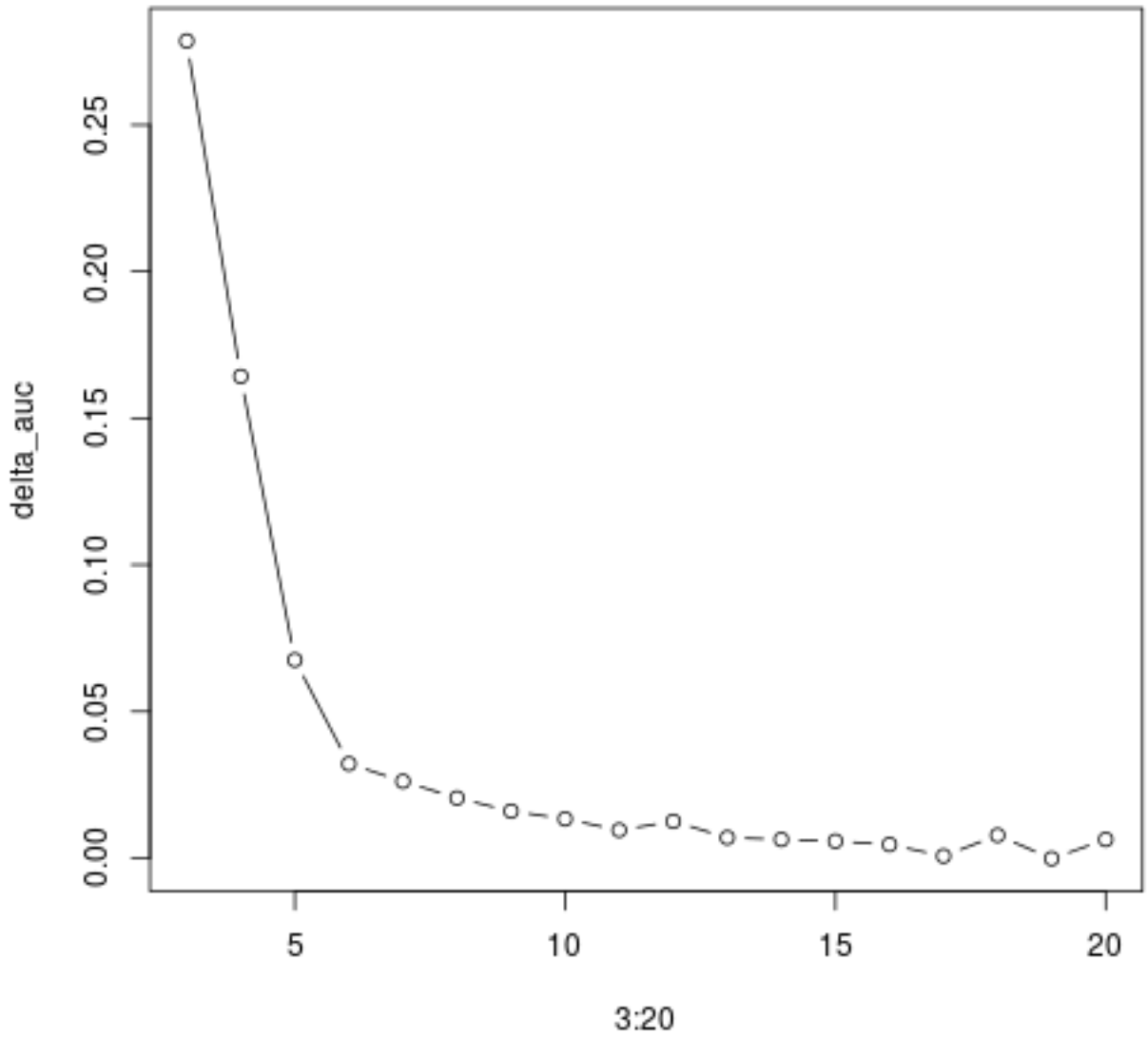


The stability of the clustering based on the consensus matrix can then be measured via a single number by looking at the area under the curve: the more stable the clustering, the closer to 0 or 1 will be the entries of consensus matrix. The consensus clustering strategy thus suggest at looking at either the AUC as a function of the number of the clusters or the "improvement" in the AUC as a function of the number of cluster.

```
auc_scores = moanin:::get_auc_similarity_scores(all_labels)
plot(n_clusters, auc_scores, col="black", type="b", pch=16)
```




```
delta_auc = diff(auc_scores)/auc_scores[1:(length(auc_scores)-1)]
plot(3:20, delta_auc, col="black", type="b")
```



The consensus clustering method suggest that the most stable is $k = 2$, which separates over-expressed genes from under-expressed genes. While it is indeed a very stable clustering, it does not capture the range of gene expression patterns present in the data. This shows the limitation of such method on real data, where the number of clusters is not clearly defined.

Downstream analysis of clusters.

Once good clusters are obtained, the next step is to leverage the clustering to ease interpretation. Classic enrichment analysis step can then be performed in the gene set defined in each cluster: KEGG pathway enrichment analysis, GO term enrichment analysis, motif enrichment analysis, etc.

First, let us clean up the genes we work with and only select the genes we are going to use in the enrichment analysis. One can either use the whole set of genes, only the set of differentially expressed genes in each cluster, or a subset of genes that fit well to a cluster (based on some criterion).

Finding enriched pathways using `biomaRt` and `KEGGprofile`

Let us first tackle the case of pathway enrichment analysis. We will leverage the packages `biomaRt` [17] and `KEGGprofile` [18] for this step. `KEGGprofile` is a package that easily allows to perform pathway enrichment analysis on a set of genes labeled with the ensembl annotation. We thus need to convert the gene

names into the appropriate format. This is where `biomaRt` comes in handy: it enables easy conversion from one gene annotation to another. Here, we will use it to convert the gene names from the Refseq annotation to the ensembl one.

```
ensembl = useMart("ensembl")
ensembl = useDataset("mmusculus_gene_ensembl", mart=ensembl)
```

```
cluster = 8
labels = unlist(labels)
gene_names = names(labels)
genes = gene_names[labels == cluster]

# convert gene names
genes = getBM(attributes=c("ensembl_gene_id", "entrezgene_id"),
        filters="refseq_mrna", values=genes,
        mart=ensembl)["entrezgene_id"]
genes = as.vector(unlist(genes))
pathways = find_enriched_pathway(
    genes, species="mmu",
    download_latest=FALSE)$stastic
```

	Pathway	Percentage	Adj. p-value
04060	Cytokine-cytokine receptor interaction	34	0
04620	Toll-like receptor signaling pathway	40	0
04630	Jak-STAT signaling pathway	33	0
03050	Proteasome	53	0
04621	NOD-like receptor signaling pathway	47	0
04380	Osteoclast differentiation	34	0
04623	Cytosolic DNA-sensing pathway	45	0
04210	Apoptosis	37	0
04622	RIG-I-like receptor signaling pathway	39	0
05160	Hepatitis C	29	0

Finding enriched GO terms

To find GO terms, we use `biomaRt` to find the mapping between GO terms and gene mapping. The GO enrichment library `topGO` [19] expects the GO term to gene mapping to be a list where each item is a mapping between a gene name and a GO term ID vector.

```
$NM_199153
[1] "GO:0016020" "GO:0016021" "GO:0007186" "GO:0004930" "GO:0007165"
[6] "GO:0050896" "GO:0050909"

$NM_201361
 [1] "GO:0016020" "GO:0016021" "GO:0003674" "GO:0008150" "GO:0005794"
 [6] "GO:0005829" "GO:0005737" "GO:0005856" "GO:0005874" "GO:0005739"
[11] "GO:0005819" "GO:0000922" "GO:0072686"
```

`biomaRt` queries results a matrix with two named columns of gene names and GO term ID.

```
genes = getBM(attributes=c("go_id", "refseq_mrna"),
          values=gene_names,
          filters="refseq_mrna",
          mart=ensembl)

# Create gene to GO id mapping
gene_id_go_mapping = create_go_term_mapping(genes)
```

Once the gene ID to GO mapping list is created, `moanin` provides a simple interface to `topGO` to fetch enriched GO terms. Here, we show an example of running a GOterm enrichment on the "Biological process" ontology (BP).

```
assignments = labels == cluster

go_terms_enriched = find_enriched_go_terms(
    assignments,
    gene_id_go_mapping, ontology="BP")
```

GO ID	Description	Annotated	Significant	Expected	P-value	Adj. p-value
GO:0007224	smoothened signaling pathway	74	69	57.82	0.00041	0.326
GO:0010970	transport along microtubule	82	75	64.07	0.00045	0.326
GO:0048562	embryonic organ morphogenesis	155	137	121.10	0.00067	0.378
GO:0060173	limb development	92	84	71.88	0.00067	0.378
GO:0006805	xenobiotic metabolic process	47	45	36.72	0.00089	0.451
GO:0060070	canonical Wnt signaling pathway	152	134	118.76	0.00099	0.456
GO:0019395	fatty acid oxidation	53	51	41.41	0.00133	0.562
GO:0055114	oxidation-reduction process	533	458	416.42	0.00152	0.593
GO:0090596	sensory organ morphogenesis	134	118	104.69	0.00216	0.783
GO:0010811	positive regulation of cell-substrate ad...	77	70	60.16	0.00260	0.879
GO:0060411	cardiac septum morphogenesis	33	32	25.78	0.00293	0.929
GO:0021510	spinal cord development	48	45	37.50	0.00339	1.000
GO:0016571	histone methylation	74	67	57.82	0.00418	1.000
GO:0042738	exogenous drug catabolic process	22	22	17.19	0.00435	1.000
GO:1901381	positive regulation of potassium ion tra...	22	22	17.19	0.00435	1.000

Conclusion

This workflow provides a tutoral for the analysis of time-course gene expression data in R, illustrated through the analysis of mice lung tissue exposed to different influenza strain. It covers four main steps; (1) quality control and normalization; (2) differential expression analysis; (3) clustering of time-course gene expression data; (4) downstream analysis of clusters.

Software and data availability

The source code for this workflow can be found at `https://github.com/NelleV/2019timecourse-rnaseq-pipeline`. Archived source code at the time of publication can be found at `XXX`.

All packages use in the workflow are available on GitHub, CRAN, or Bioconductor.

Data used in this workflow are available from NCBI GEO, accession GSE63786. Normalized data can be found in `moanin`. Normalization information is provided as supplementary information.

Last but not least, we use `sessionInfo()` to display all packages used in this pipeline and their version numbers.

```
sessionInfo()
```

```
## R version 3.6.0 (2019-04-26)
## Platform: x86_64-pc-linux-gnu (64-bit)
## Running under: Ubuntu 19.04
##
## Matrix products: default
## BLAS:   /usr/lib/x86_64-linux-gnu/blas/libblas.so.3.8.0
## LAPACK: /usr/lib/x86_64-linux-gnu/lapack/liblapack.so.3.8.0
##
## locale:
## [1] C
##
## attached base packages:
##  [1] stats4    parallel  splines   stats     graphics  grDevices utils
##  [8] datasets  methods   base
##
## other attached packages:
##  [1] RColorBrewer_1.1-2   moanin_0.0.0         ggfortify_0.4.6
##  [4] topGO_2.32.0         SparseM_1.77         GO.db_3.6.0
```

```
##  [7] AnnotationDbi_1.42.1 IRanges_2.14.12      S4Vectors_0.18.3
## [10] graph_1.58.2         viridis_0.5.1        viridisLite_0.3.0
## [13] KEGGprofile_1.22.0   RCurl_1.95-4.12      bitops_1.0-6
## [16] NMF_0.21.0           Biobase_2.40.0       BiocGenerics_0.26.0
## [19] cluster_2.0.9        rngtools_1.3.1.1     pkgmaker_0.27
## [22] registry_0.5-1       ggplot2_3.1.1        kableExtra_1.1.0
## [25] biomaRt_2.36.1       BiocStyle_2.8.2      knitr_1.22
## [28] limma_3.36.5         usethis_1.5.0        devtools_2.0.2
## [31] rmarkdown_1.12
##
## loaded via a namespace (and not attached):
##  [1] colorspace_1.4-1        rprojroot_1.3-2
##  [3] XVector_0.20.0          fs_1.3.1
##  [5] rstudioapi_0.10         remotes_2.0.4
##  [7] bit64_0.9-7             ClusterR_1.1.9
##  [9] KEGG.db_3.2.3           xml2_1.2.0
## [11] codetools_0.2-16        doParallel_1.0.14
## [13] pkgload_1.0.2           ade4_1.7-13
## [15] gridBase_0.4-7          png_0.1-7
## [17] FD_1.0-12               readr_1.3.1
## [19] compiler_3.6.0          httr_1.4.0
## [21] backports_1.1.4         Matrix_1.2-17
## [23] assertthat_0.2.1        lazyeval_0.2.2
## [25] cli_1.1.0               htmltools_0.3.6
## [27] prettyunits_1.0.2       tools_3.6.0
## [29] gmp_0.5-13.5            gtable_0.3.0
## [31] glue_1.3.1              reshape2_1.4.3
## [33] dplyr_0.8.0.1           BiocWorkflowTools_1.6.2
## [35] Rcpp_1.0.1              Biostrings_2.48.0
## [37] NMI_2.0                 ape_5.3
## [39] nlme_3.1-139            iterators_1.0.10
## [41] xfun_0.6                stringr_1.4.0
## [43] ps_1.3.0                testthat_2.1.1
## [45] rvest_0.3.3             gtools_3.8.1
## [47] XML_3.98-1.19           zlibbioc_1.26.0
## [49] MASS_7.3-51.4           scales_1.0.0
## [51] hms_0.4.2               curl_3.3
## [53] yaml_2.2.0              memoise_1.1.0
## [55] gridExtra_2.3           TeachingDemos_2.10
## [57] stringi_1.4.3           RSQLite_2.1.1
## [59] desc_1.2.0              foreach_1.4.4
## [61] permute_0.9-5           pkgbuild_1.0.3
## [63] bibtex_0.4.2            geometry_0.4.1
## [65] rlang_0.4.2             pkgconfig_2.0.2
## [67] matrixStats_0.54.0      evaluate_0.13
## [69] lattice_0.20-38         purrr_0.3.2
## [71] bit_1.1-14              processx_3.3.1
## [73] tidyselect_0.2.5        plyr_1.8.4
## [75] magrittr_1.5            bookdown_0.9
## [77] R6_2.4.0                DBI_1.0.0
## [79] mgcv_1.8-28             pillar_1.3.1
## [81] withr_2.1.2             abind_1.4-5
## [83] KEGGREST_1.20.2         tibble_2.1.1
## [85] crayon_1.3.4            progress_1.2.0
## [87] grid_3.6.0              blob_1.1.1
## [89] callr_3.2.0             git2r_0.25.2
## [91] vegan_2.5-4             digest_0.6.18
## [93] webshot_0.5.1           xtable_1.8-4
## [95] tidyr_0.8.3             munsell_0.5.0
## [97] magic_1.5-9             sessioninfo_1.1.1
```

Author contributions

NV and EP wrote the workflow.

Competing interests

The authors declare that they have no competing interests.

Grant information

This research was funded in part by a Department of Energy (DOE) grant (DE-SC0014081); by the Gordon and Betty Moore Foundation (Grant GBMF3834) and the Alfred P. Sloan Foundation (Grant 2013-10-27) to the University of California, Berkeley [N.V.]; by a ENS-CFM Data Science Chair [E.P.].

I confirm that the funders had no role in study design, data collection and analysis, decision to publish, or preparation of the manuscript.

Acknowledgments

The authors thank Karthik Ram and the Ropensci community for valuable feedback.

References

[1] A. K. Shalek, R. Satija, J. Shuga, J. J. Trombetta, D. Gennert, D. Lu, P. Chen, R. S. Gertner, J. T. Gaublomme, N. Yosef, S. Schwartz, B. Fowler, S. Weaver, J. Wang, X. Wang, R. Ding, R. Raychowdhury, N. Friedman, N. Hacohen, H. Park, A. P. May, and A. Regev. Single-cell RNA-seq reveals dynamic paracrine control of cellular variation. *Nature*, 510 (7505):363–369, Jun 2014.

[2] N. Habib, Y. Li, M. Heidenreich, L. Swiech, I. Avraham-Davidi, J. J. Trombetta, C. Hession, F. Zhang, and A. Regev. Div-Seq: Single-nucleus RNA-Seq reveals dynamics of rare adult newborn neurons. *Science*, 353(6302):925–928, 08 2016.

[3] C. Trapnell, D. Cacchiarelli, J. Grimsby, P. Pokharel, S. Li, M. Morse, N. J. Lennon, K. J. Livak, T. S. Mikkelsen, and J. L. Rinn. The dynamics and regulators of cell fate decisions are revealed by pseudotemporal ordering of single cells. *Nat. Biotechnol.*, 32(4):381–386, Apr 2014.

[4] J. D. Storey, W. Xiao, J. T. Leek, R. G. Tompkins, and R. W. Davis. Significance analysis of time course microarray experiments. *Proc. Natl. Acad. Sci. U.S.A.*, 102(36):12837–12842, Sep 2005.

[5] Shuang Wu and Hulin Wu. More powerful significant testing for time course gene expression data using functional principal component analysis approaches. *BMC Bioinformatics*, 14(1):6, Jan 2013. ISSN 1471-2105. doi: 10.1186/1471-2105-14-6. URL `https://doi.org/10.1186/1471-2105-14-6`.

[6] Taesung Park, Dong-Hyun Yoo, Jun-Ik Ahn, Seung Yeoun Lee, Seungmook Lee, Sung-Gon Yi, and Yong-Sung Lee. Statistical tests for identifying differentially expressed genes in time-course microarray experiments. *Bioinformatics*, 19(6):694–703, 04 2003. ISSN 1367-4803. doi: 10.1093/bioinformatics/btg068. URL `https://doi.org/10.1093/bioinformatics/btg068`.

[7] Sun Wenguang and Wei Zhi. Multiple testing for pattern identification, with applications to microarray time-course experiments. *Journal of the American Statistical Association*, 106(493):73–88, 2011. doi: 10.1198/jasa.2011.ap09587. URL `https://doi.org/10.1198/jasa.2011.ap09587`.

[8] J. E. Shoemaker, S. Fukuyama, A. J. Eisfeld, Dongming Zhao, Eiryo Kawakami, Saori Sakabe, Tadashi Maemura, Takeo Gorai, Hiroaki Katsura, Yukiko Muramoto, Shinji Watanabe, Tokiko Watanabe, Ken Fuji, Yukiko Matsuoka, Hiroaki Kitano, and Yoshihiro Kawaoka. An Ultrasensitive Mechanism Regulates Influenza Virus-Induced Inflammation. *PLoS Pathogens*, 11(6):1–25, 2015. ISSN 15537374.

[9] N. Varoquaux, B. Cole, C. Gao, G. Pierroz, C. Baker, D. Patel, M. Madera, T. Jeffers, J. Hollingsworth, J. Sievert, Y. Yoshinaga, J. Owiti, V. Singan, S. DeGraaf, L. Xu, J. M. Blow, M. Harrison, A. Visel, C. Jansson, K. K. Niyogi, R. Hutmacher, D. Coleman-Derr, R. O'Malley, J. Taylor, J. Dahlberg, J. P. Vogel, P. G. Lemaux, and E. Purdom. Lifecycle transcriptomics of field-droughted *Sorghum bicolor* reveals rapid biotic and metabolic changes. Under review.

[10] Matthiew E Ritchie, Belinda Phipson, Di Wu, Yifang Hu, Charity W Law, Wei Shi, and Gordon K Smyth. limma powers differential expression analyses for RNA-sequencing and microarray studies. *Nucleic Acids Research*, 43(7):e47, 2015.

[11] M. D. Robinson, D. J. McCarthy, and G. K. Smyth. edgeR: a Bioconductor package for differential expression analysis of digital gene expression data. *Bioinformatics*, 26(1):139–140, Jan 2010.

[12] Michael I. Love, Wolfgang Huber, and Simon Anders. Moderated estimation of fold change and dispersion for rna-seq data with deseq2. *Genome Biology*, 15:550, 2014. doi: 10.1186/s13059-014-0550-8.

[13] R.A. Fisher. *Statistical methods for research workers*. Edinburgh Oliver & Boyd, 1925.

[14] Robert L. Thorndike. Who belongs in the family? *Psychometrika*, 18(4):267–276, Dec 1953. ISSN 1860-0980. doi: 10.1007/BF02289263. URL `https://doi.org/10.1007/BF02289263`.

[15] Asa Ben-Hur, André Elisseeff, and Isabelle Guyon. A stability based method for discovering structure in clustered data. *Pacific Symposium on Biocomputing. Pacific Symposium on Biocomputing*, pages 6–17, 2001.

[16] Stefano Monti, Pablo Tamayo, Jill Mesirov, and Todd Golub. Consensus clustering: A resampling-based method for class discovery and visualization of gene expression microarray data. *Machine Learning*, 52(1):91–118, Jul 2003. ISSN 1573-0565. doi: 10.1023/A:1023949509487. URL `https://doi.org/10.1023/A:1023949509487`.

[17] S. Durinck, Y. Moreau, A. Kasprzyk, S. Davis, B. De Moor, A. Brazma, and W. Huber. BioMart and Bioconductor: a powerful link between biological databases and microarray data analysis. *Bioinformatics*, 21(16):3439–3440, Aug 2005.

[18] Shilin Zhao, Yan Guo, and Yu Shyr. *KEGGprofile: An annotation and visualization package for multi-types and multi-groups expression data in KEGG pathway*, 2017. R package version 1.22.0.

[19] Adrian Alexa and Jorg Rahnenfuhrer. *topGO: Enrichment Analysis for Gene Ontology*, 2016. R package version 2.32.0.