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 workflows provides a step-by-step tutorial of the methodology used to analyse time-course data: (1) normalization of the micro-array dataset; (2) differential expression analysis using functional data analysis; (3) clustering fo time-course data; (4) interpreting clusters with GO term and KEGG pathway enrichment analysis.

Keywords

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

#TODO list

- Add legends to the plots that don't have.
- Currently have two functions "plot_genes", "plot_centroids", that do exactly the same thing. Find a better name for both of them
- switch to camel case function names
- Remove the namespace call
- Make clear that it applies to RNASeq and how to call the different functions toRNASeq
- Check that clustering can deal with count data.

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 in distinct groups, there is also a long history of time-course expression studies, where the the gene expression is compared across time by measuring mRNA levels from different samples across time [1]. Such time course studies can vary from measuring a few distinct time points, to sampling ten to twenty time points. Many longer time series are particularly interested in investigating development over time. More recently, single-cell studies track single cells through their development, and a single cell is measured at a particular moment in its developmental progression – 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` [1], functional component analysis based models [2], time-course permutation tests [3], and multiple testing strategies to combine singe time point differential expression analysis [4]. However, with long time course datasets, particularly in developmental systems, a massive number of genes will show *some* change. For example, in the mice lung tissues infected with influenza, 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 into biologically interpretable patterns the vast number of changes discovered.

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

- 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) [5]. 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.

[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 in certain settings, but not directly considered here.

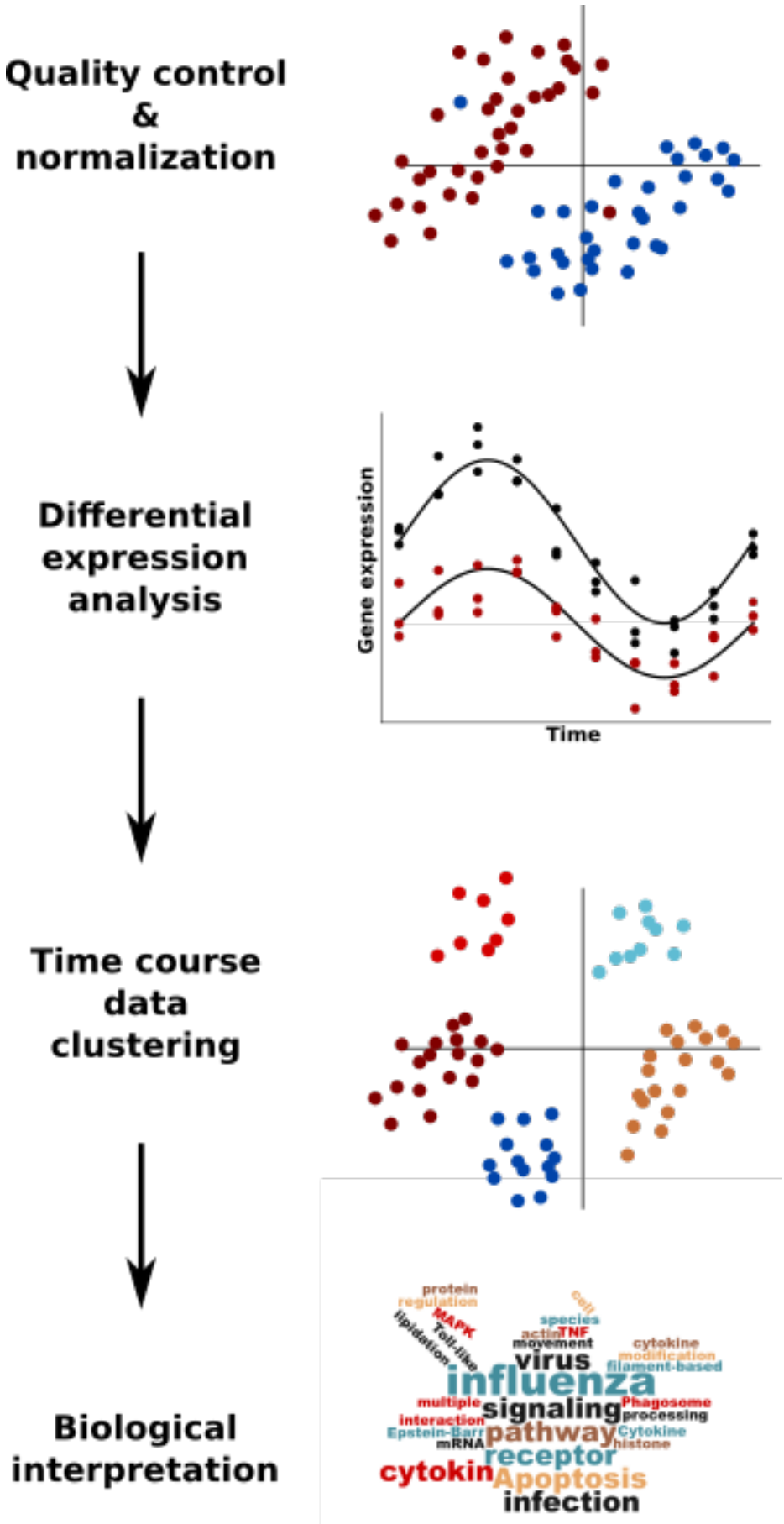


Figure 1. Workflow for analyzing time-course datasets.

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, but could be done for any gene expression analysis.

First let's load the data. The package `moanin` contains the normalized data and meta of [5].

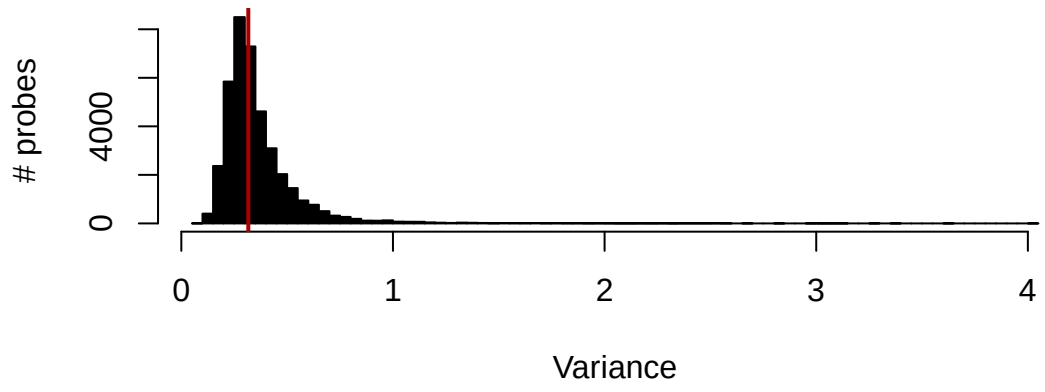
```
# Now load in the metadata
data(shoemaker2015)
meta = shoemaker2015$meta
data = shoemaker2015$data
```

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: we keep for this step only the top 50% most variable genes.

```
variance_cutoff = 0.5
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,]
```

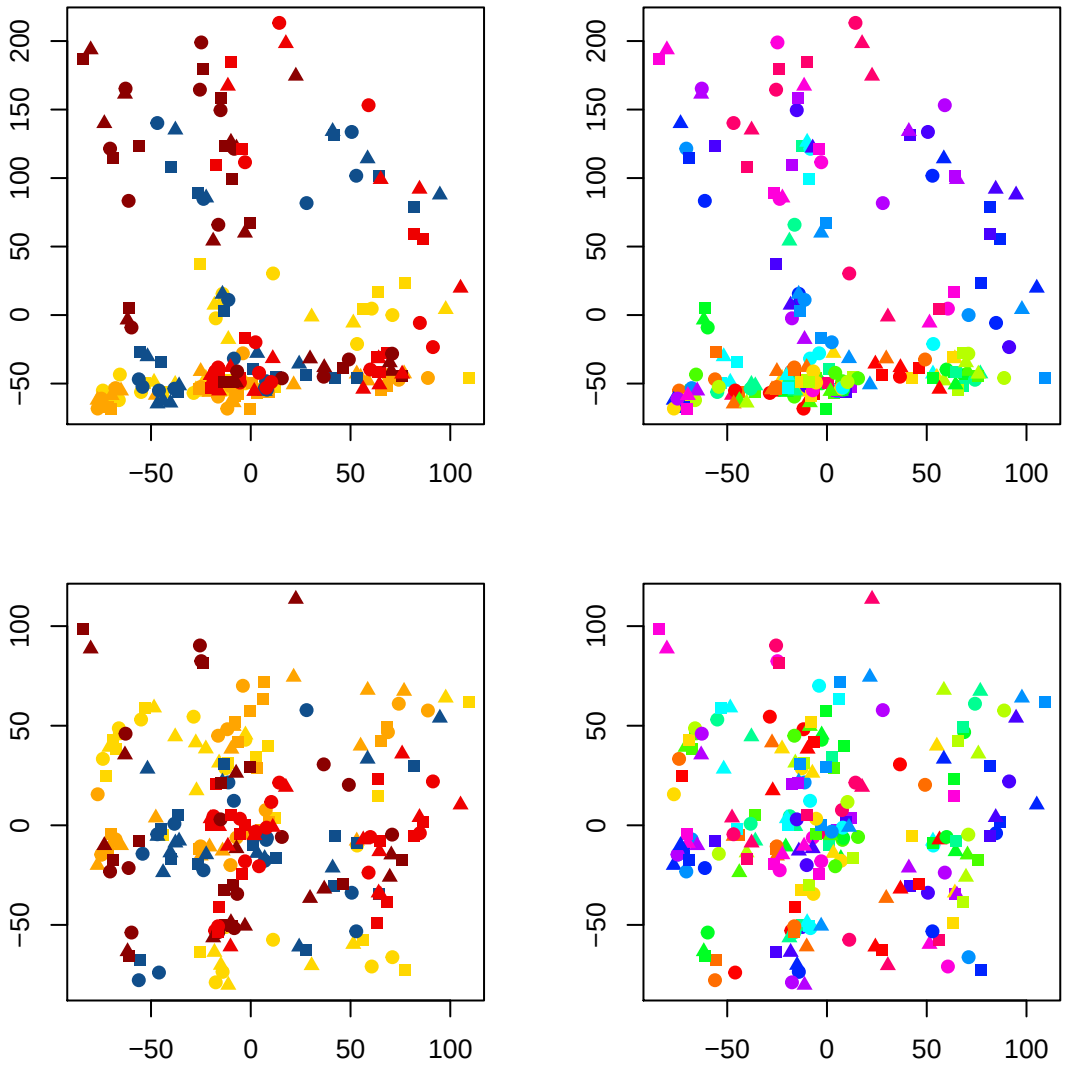


Let us first perform the PCA analysis. Here, we perform a PCA of rank 3 of the centered and scaled gene expression data.

```
# Reorder genes on condition, time, and replicate

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

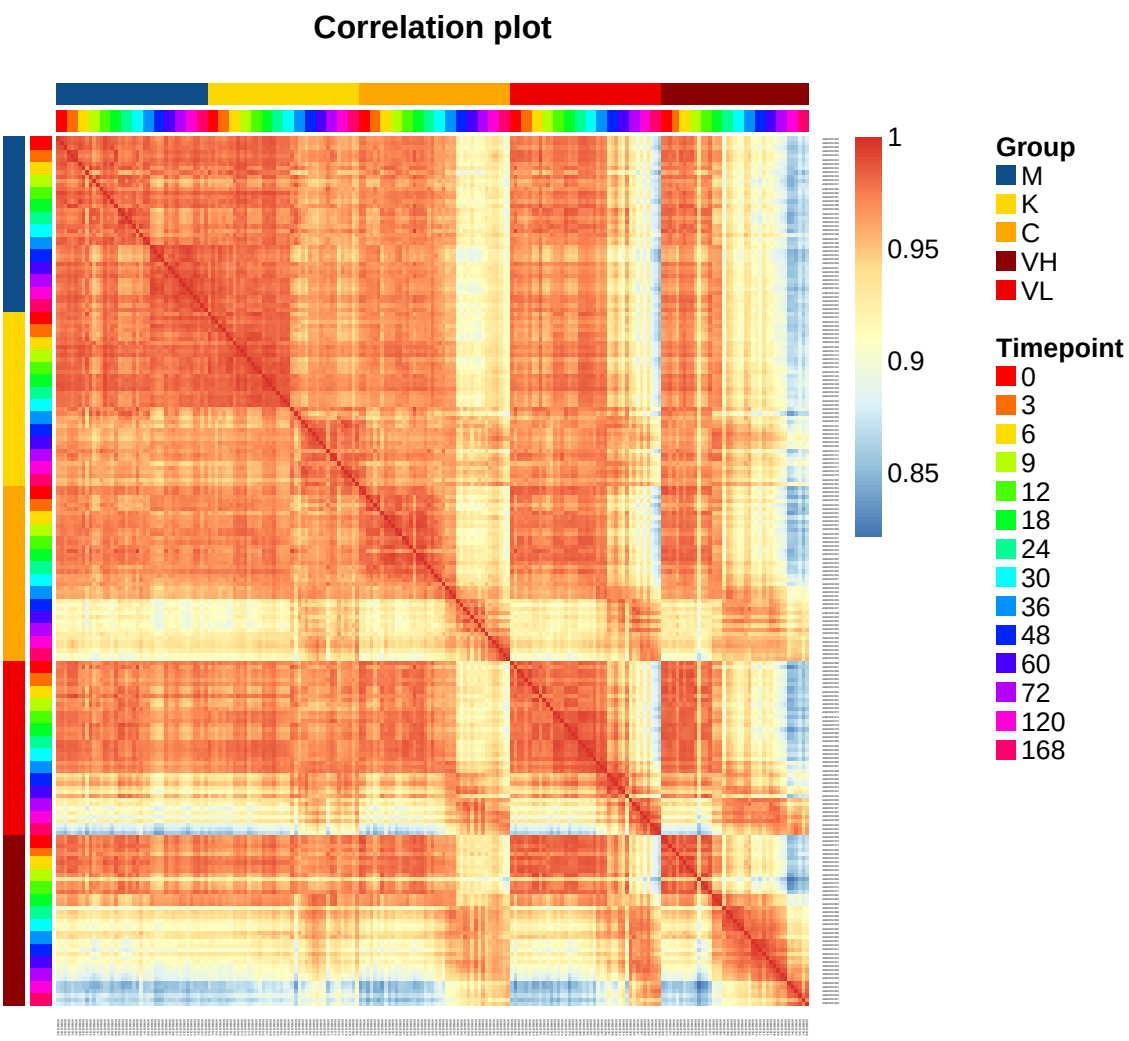
We then plot the two first components, and color each sample by (1) its condition; (2) its sampling time. We use different markers for each replicate. **EAP: Unclear what 'markers' refers to** We also plot the second and third components in the second row.

```
par(mfrow=c(2, 2), mar=c(2.5, 2.5, 2.5, 2.5))
plot(
    pca_data$x[, "PC2"], pca_data$x[, "PC1"],
    col=ann_colors$Group[meta$Group],
    pch=ann_markers$Replicate[as.factor(meta$Replicate)],
    xlab="PC2", ylab="PC1")
plot(
    pca_data$x[, "PC2"], pca_data$x[, "PC1"],
    col=ann_colors$Timepoint[as.factor(meta$Timepoint)],
    pch=ann_markers$Replicate[as.factor(meta$Replicate)],
    xlab="PC2", ylab="PC1")

plot(
    pca_data$x[, "PC2"], pca_data$x[, "PC3"],
    col=ann_colors$Group[meta$Group],
    pch=ann_markers$Replicate[as.factor(meta$Replicate)],
    xlab="PC2", ylab="PC3")
plot(
    pca_data$x[, "PC2"], pca_data$x[, "PC3"],
    col=ann_colors$Timepoint[as.factor(meta$Timepoint)],
    pch=ann_markers$Replicate[as.factor(meta$Replicate)],
    xlab="PC2", ylab="PC3")
```



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)

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 midly 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 [1], 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 [6, Robinson

et al. [7], Love et al. [8]] . 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 of longer time courses data, with per-time point analysis reserved for particularly interesting comparisons of individual time-points.

We note that we could have time course data on either a single condition or time course data on multiple conditions (such as the influenza dataset we are considering), which will alter slightly the types of questions we are interested in, but the two basic approaches remains the same. In what follows, we will focus on the situation where we have multiple conditions. **EAP: Any reason one and not the other. Maybe should change this sentence**

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 `splines_model` object) to hold the meta data, as well as information for fitting the spline model.

We start by creating the `splines_model` object using the `create_splines_model` function. The `splines_model` object contains a number of metadata and options used throughout this analysis: the condition and timepoints of each samples, the formula object used or the basis or the degrees of freedom of the model. 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

If no formula is provided, it will default to the following: `formula = ~Group:ns(Timepoint, df=degrees_of_fredoom) + Group + 0`

```
splines_model = create_splines_model(meta, degrees_of_freedom=6)
```

Here we have provided the meta data that defines... `create_splines_model` will .... **FIXME add information about what meta data is expected, describe what `create_splines_model` does (does it fit the splines? Just hold create an object? Create basis functions?)**.

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. This is traditionally done by the user defining the comparisons (called `contrasts` in linear models). Under the hood, simply calls `limma` [6] on the set of contrasts provided. By default, it expects RNA-Seq contact counts, and will estimate voom weights. We tern

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 [6].

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

```
contrasts = create_timepoints_contrasts("M", "VL", splines_model)
```

x
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

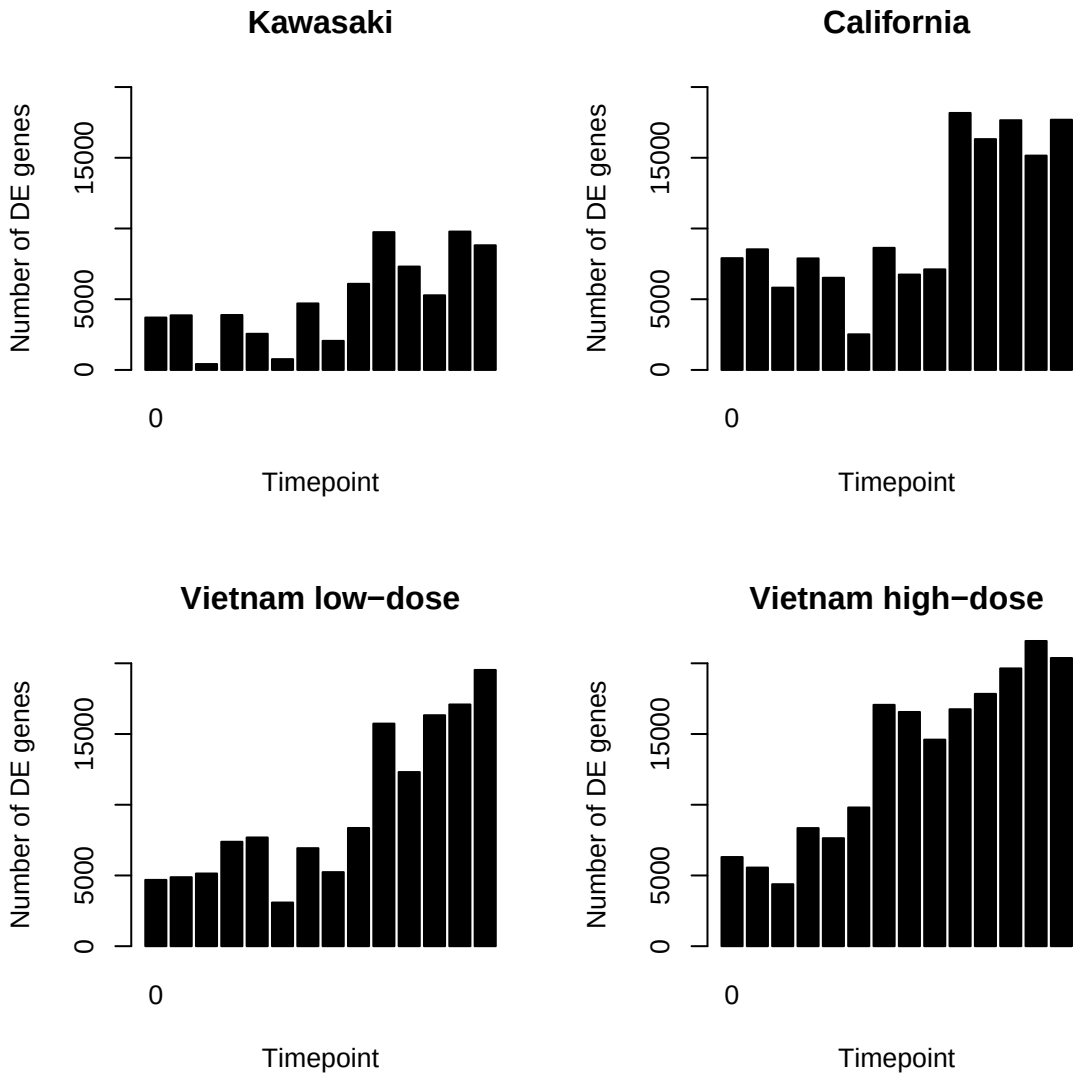
Then run the differential expression analysis on all of those timepoints jointly.

```
# Define contrast
weekly_de_analysis = DE_timepoints(
     data, splines_model, contrasts,
     use_voom_weights=FALSE)
```

	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	M.6
NM_009912	0.2728476	0.4745864	2.2018266	0.7670600	0.8760739	1.1576881	0
NM_008725	0.8508522	0.9242869	2.0281764	0.1390300	0.3020872	1.4568427	0
NM_007473	0.0880394	0.2192542	0.6779618	0.9165990	0.9592675	0.7639279	0
ENSMUST00000094955	0.0081879	0.0367991	1.1856925	0.9353016	0.9690575	-0.6333013	0
NM_001042489	0.4191775	0.6208106	1.2123141	0.7607212	0.8723405	0.5992596	0
NM_008159	0.1231754	0.2777536	1.1022344	0.1169243	0.2678638	0.7374376	0
NM_001013813	0.0565287	0.1593800	0.6299949	0.1710643	0.3480241	0.9746582	0
AK039774	0.0150416	0.0590150	0.6372239	0.0622016	0.1707939	0.9714819	0
NM_013782	0.8754233	0.9377109	0.5940058	0.0001107	0.0010876	1.1803029	0
NM_028622	0.7667946	0.8759265	1.6477918	0.9457494	0.9741115	-1.8648471	0

We can repeat this for each of the conditions

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



The distribution of number of genes found differentially expressed by considering each time-point independantly highlights the challenges of such approach. Timepoint 6H and 18H of the Kawasaki strain have fewer numbers of genes found differentially expressed than timepoints 3H and 9H: this is likely due to biological or technical variances for specific genes at specific timepoints. **EAP: We should add more to this**

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

Time-course differential expression analysis between two groups

To leverage this temporal structure, Storey et al [1] 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 means.

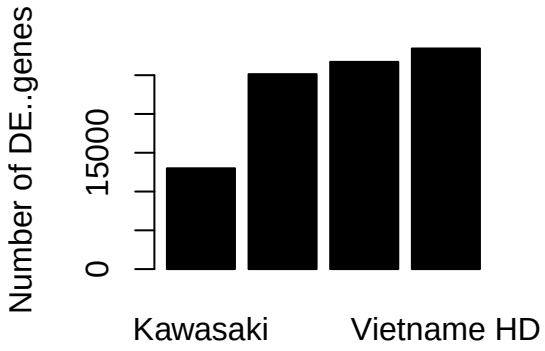
FIXME add in equations, like from the EPICON paper, to describe them mathematically

EAP: Do we need this call to limma by the user, or can these not be done internally by the `timecourse_differential_expressi` function? We can still preserve the option of user defining their own, but would be nicer to make this simpler for the user (e.g. an if clause that tests whether input is matrix or character vector)

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

```
contrasts = makeContrasts(
    contrasts=timecourse_contrasts,
    levels=levels(meta$Group))

# 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).
pvalues = DE_timecourse(
    data, splines_model, contrasts,
    use_voom_weights=FALSE)
qvalues = apply(pvalues, 2, p.adjust)
```

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, splines_model, 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

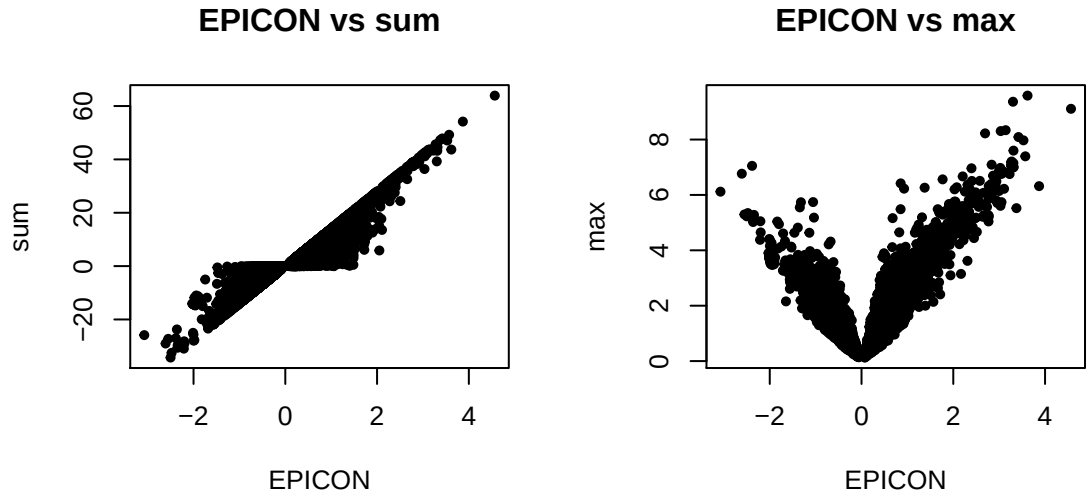
This matrix can then be used to visualize the log-fold change for each contrast per timepoint.

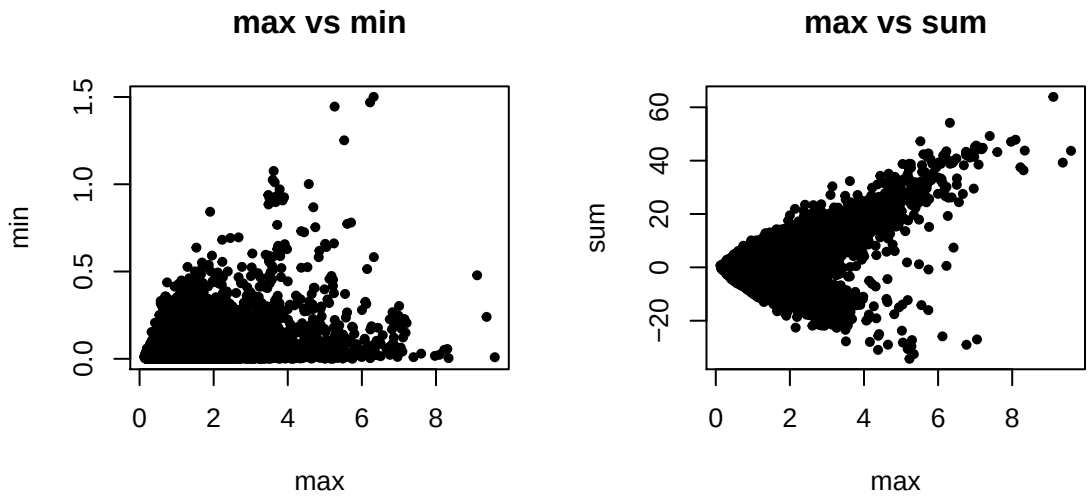
Sometimes, a single value per gene and per contrast is more useful. Here is a table of the possible ways to compute log-fold change values.

Name	Formula	
timely	$\text{lfc}(t)$	Function of time
sum	$\sum_t lfc(t)$	Sum of log fold change.
abs_sum	$\sum_t \|\|lfc(t)\|\|$	Always positive
max	$\max_t \|\|lfc(t)\|\|$	Always positive
min	$\min_t \|\|lfc(t)\|\|$	Always positive
epicon	$$	Captures overall strength of response and overall direction

EAP: `epicon` **is going to be weird for general use**

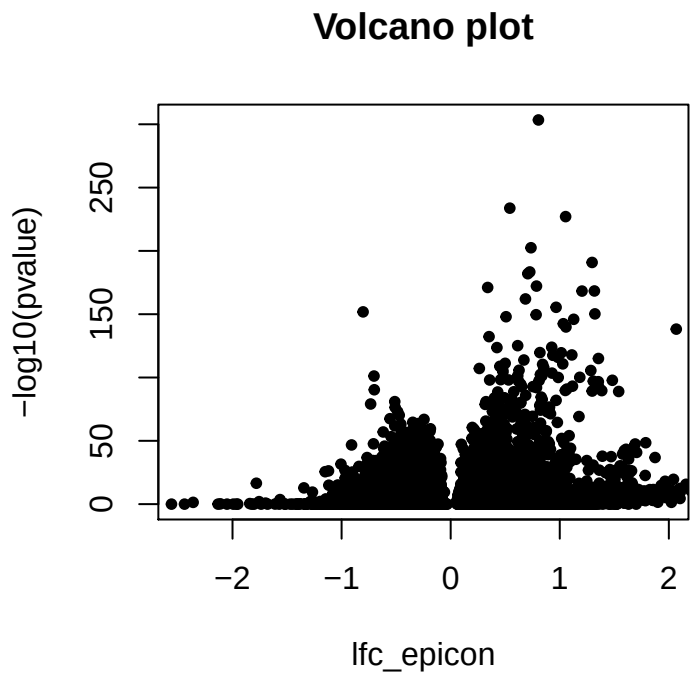
```
log_fold_change_epicon = estimate_log_fold_change(
    data, splines_model, contrasts,  method="epicon")

log_fold_change_sum = estimate_log_fold_change(
    data, splines_model, contrasts,  method="sum")

log_fold_change_max = estimate_log_fold_change(
    data, splines_model, contrasts, method="max")

log_fold_change_min = estimate_log_fold_change(
    data, splines_model, contrasts, method="min")
```





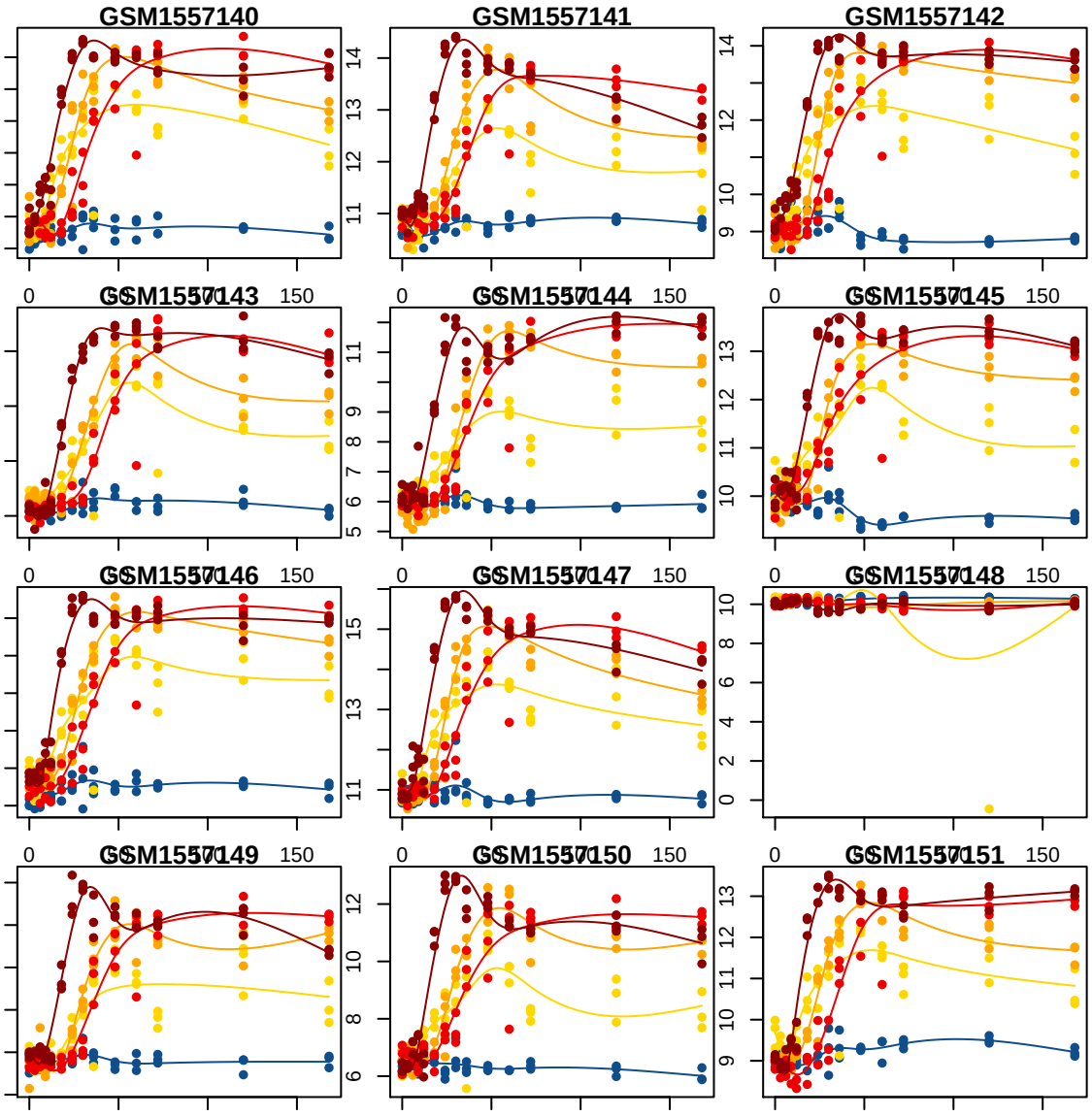
From the single measures of log-fold change and the p-value, we can now look at the volcano plot. Here is an example of a volcano plot for the comparaison of the control to the Kawasaki strain, using the EPICON log fold change computation.

```
pvalue = qvalues[, "M-K"]
lfc_epicon = log_fold_change_epicon[, "M-K"]
names(lfc_epicon) = row.names(log_fold_change_epicon)

plot(lfc_epicon, -log10(pvalue), pch=20, main="Volcano plot",
     xlim=c(-2.5, 2))
```



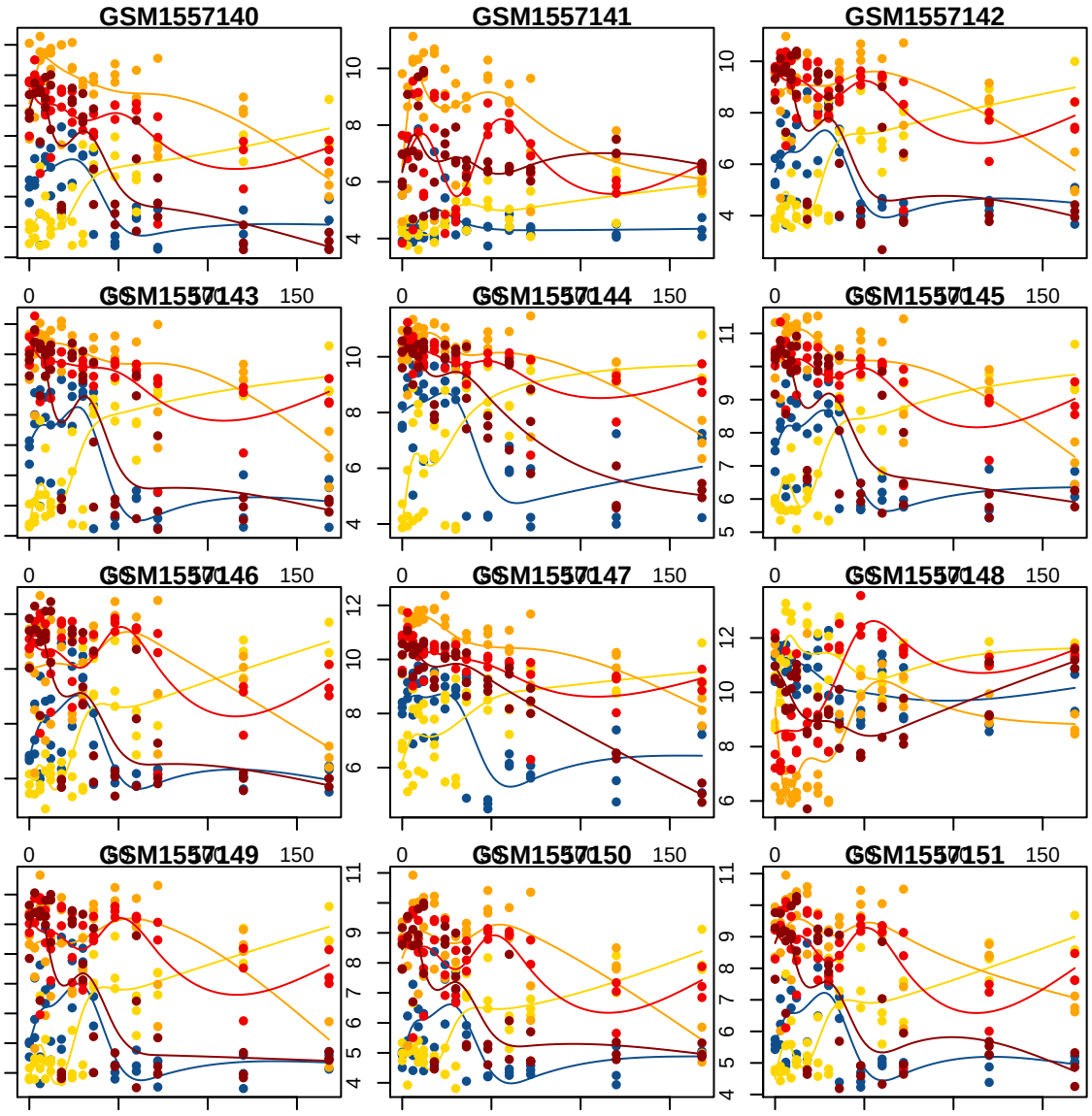
As another sanity check, `moanin` provides a simple utility function to visualize gene time-course data. Here, we plot the 12 genes with the smallest p-values.

```
top_DE_genes_pval = names(sort(pvalue)[1:12])
plot_genes(data[top_DE_genes_pval, ], splines_model,
           colors=ann_colors$Group, smooth=TRUE)
```



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

```
top_DE_genes_lfc = names(
    sort(abs(lfc_epicon),
     decreasing=TRUE)[1:12])
plot_genes(data[top_DE_genes_lfc, ], splines_model,
           colors=ann_colors$Group, smooth=TRUE)
```



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) significantly found differentially expressed; (2) "highly" differentially expressed. 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 [9]. 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)
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:

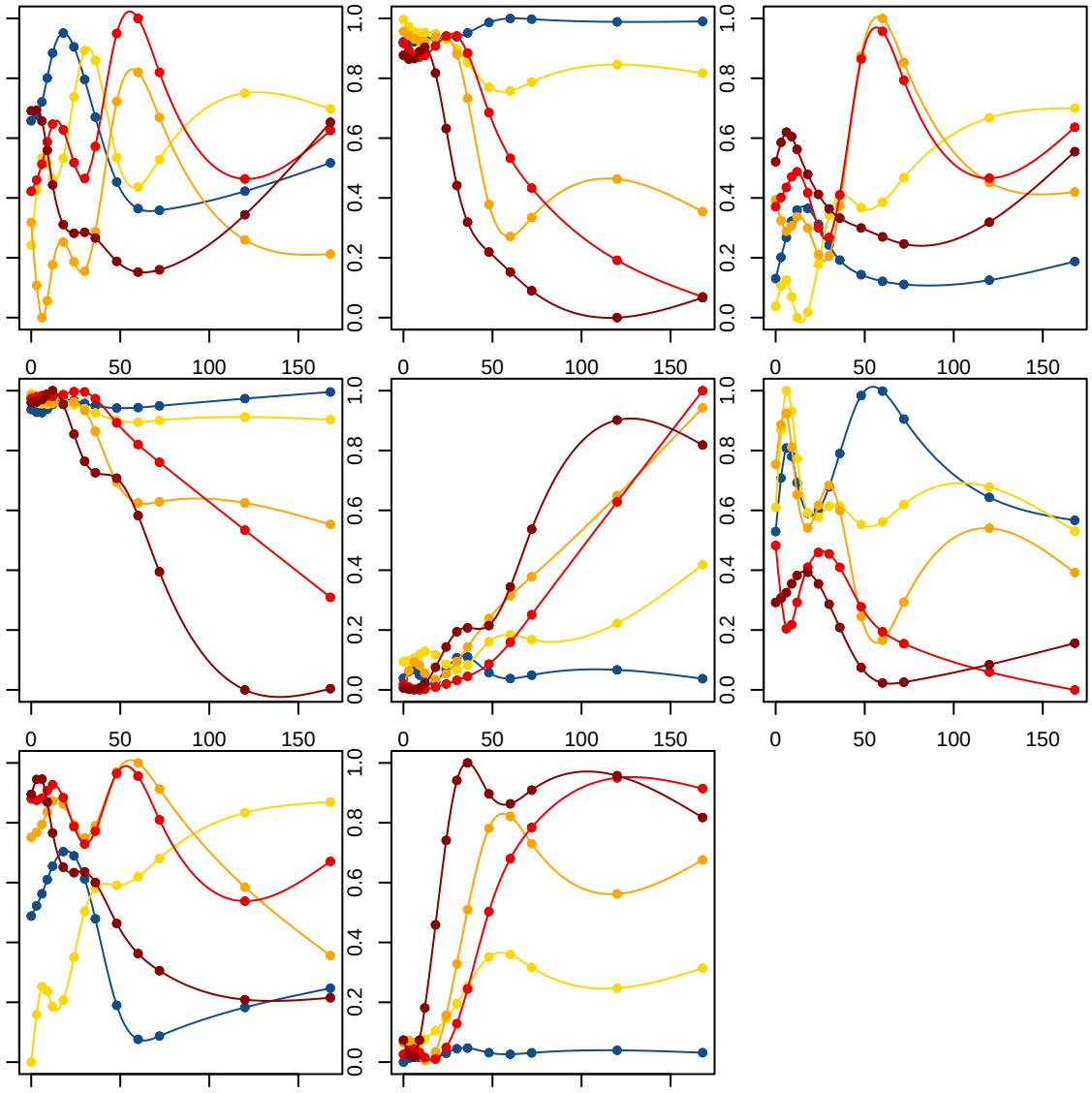
- **Splines estimation**: for each gene, fit the splines function with the basis of your choice.
- **Rescaling splines**: for each gene, rescale the estimated splines function such that the values are bounded between 0 and 1.
- **K-means**: apply k-means on the rescaled fitted splines to estimate the centroids.
- **Assign scores and labels to all genes**: then assign a score and a label to all gene based on a goodness-of-fit measure on the raw data. By default, the `splines_kmeans_score_and_label` function only labels the best 50% of genes.

The first three steps are performed jointly by the `splines_kmeans` function.

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

We then use the `plot_centroids` function to visualize the centroids obtained with the splines k-means model.

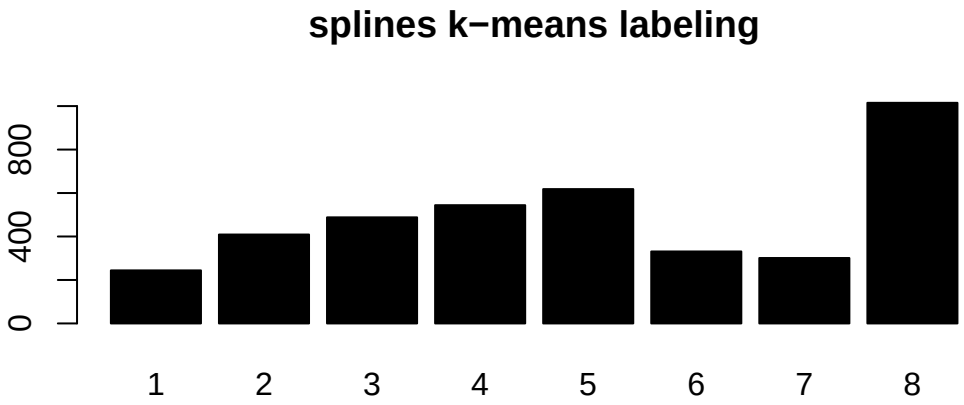
```
plot_centroids(kmeans_clusters$centroids, splines_model,
               colors=ann_colors$Group,
               smooth=TRUE)
```

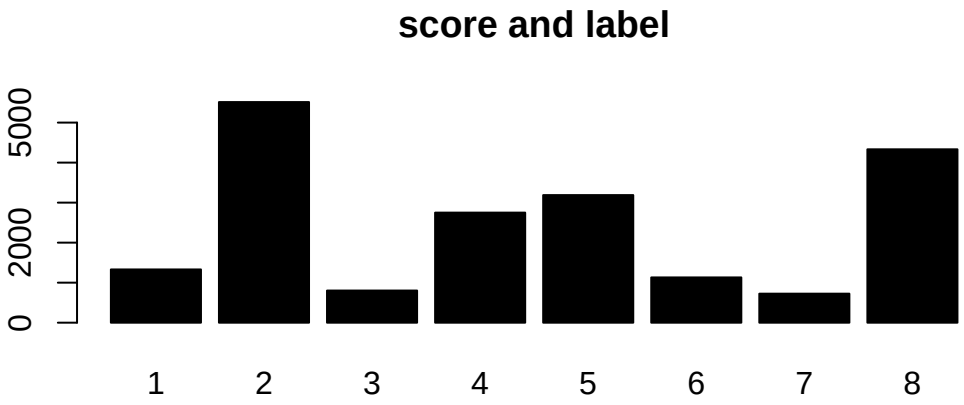


The scoring and labeling can be done via the `splines_kmeans_score_and_label` function.

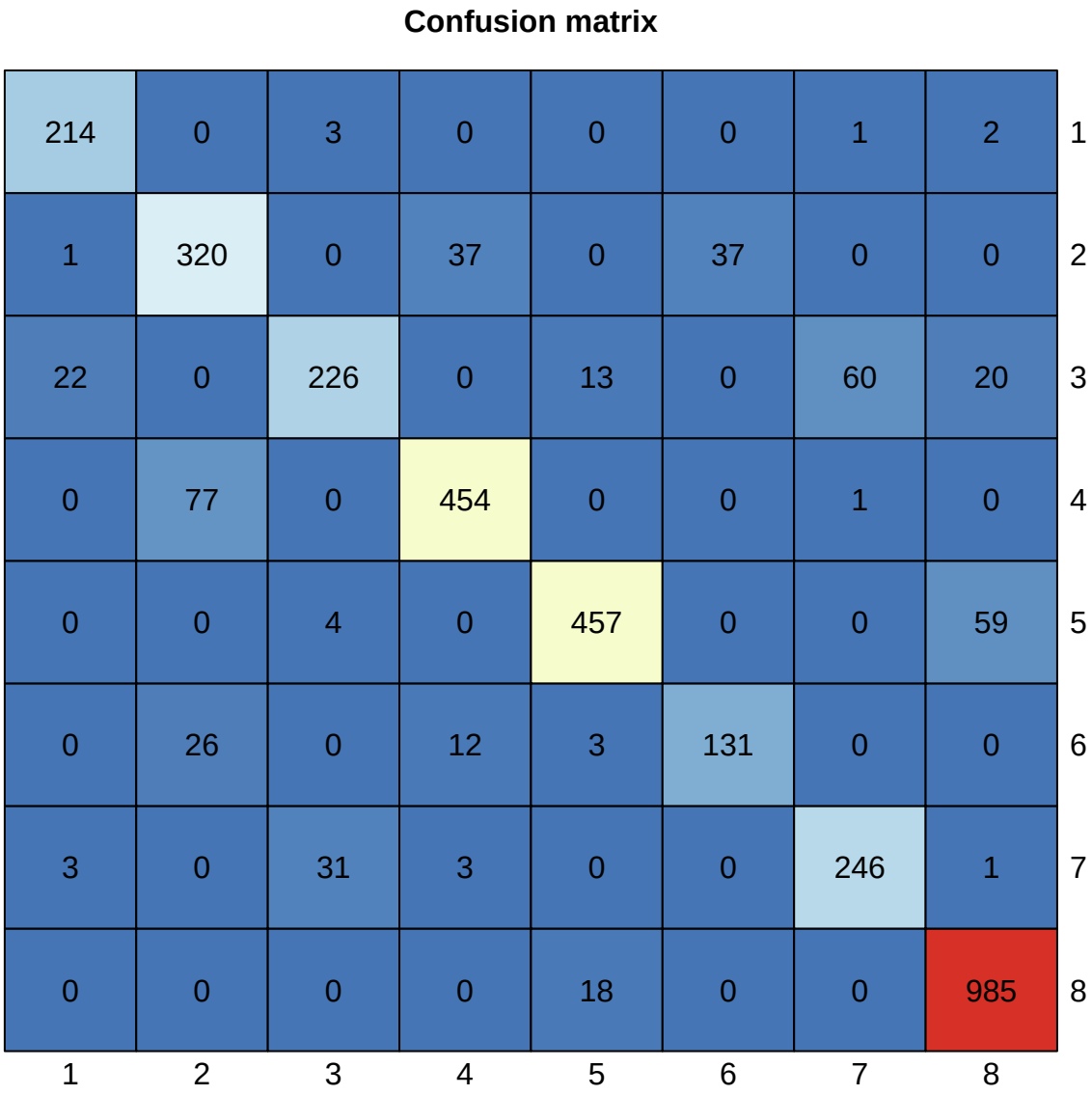
```
#<c2><a0>Then assign scores and labels to all the data, using a goodness-of-fit
# scoring function.
scores_and_labels = splines_kmeans_score_and_label(
    data, kmeans_clusters)
labels = scores_and_labels$labels

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

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 labelinig step.





The scoring and label step allows to label genes that were removed during the filtering step, yet are good matches to the centroids found during the clustering.



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 differant influenza treatments and the control, while the control remains relatively flat.

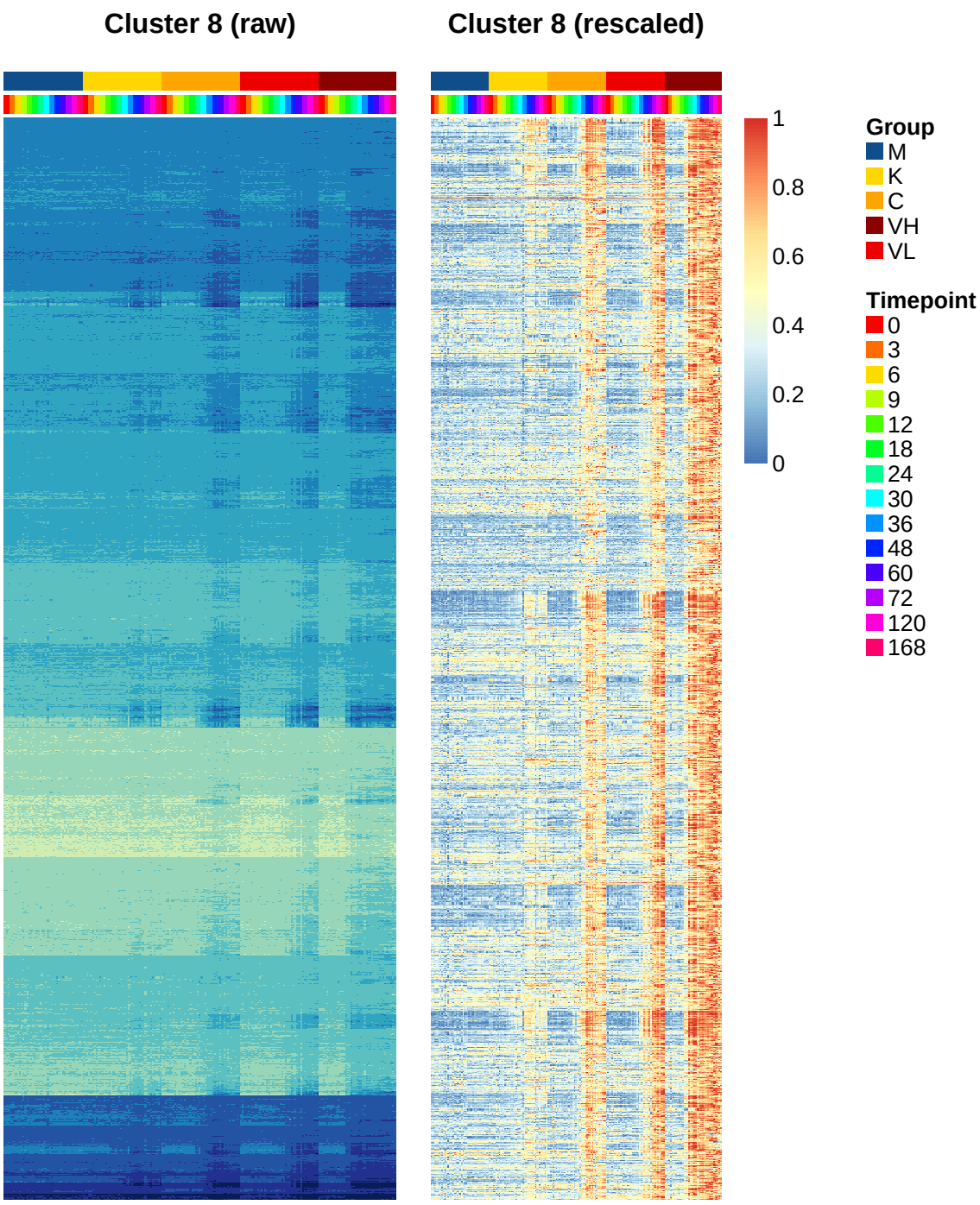
Heatmaps are useful to investigate the range of expression patterns for specific genes. Here, we are going to plot heatmaps of the normalized gene expression patterns and the rescaled gene expression patterns side by side.

First, select the genes of interest.

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

```
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:

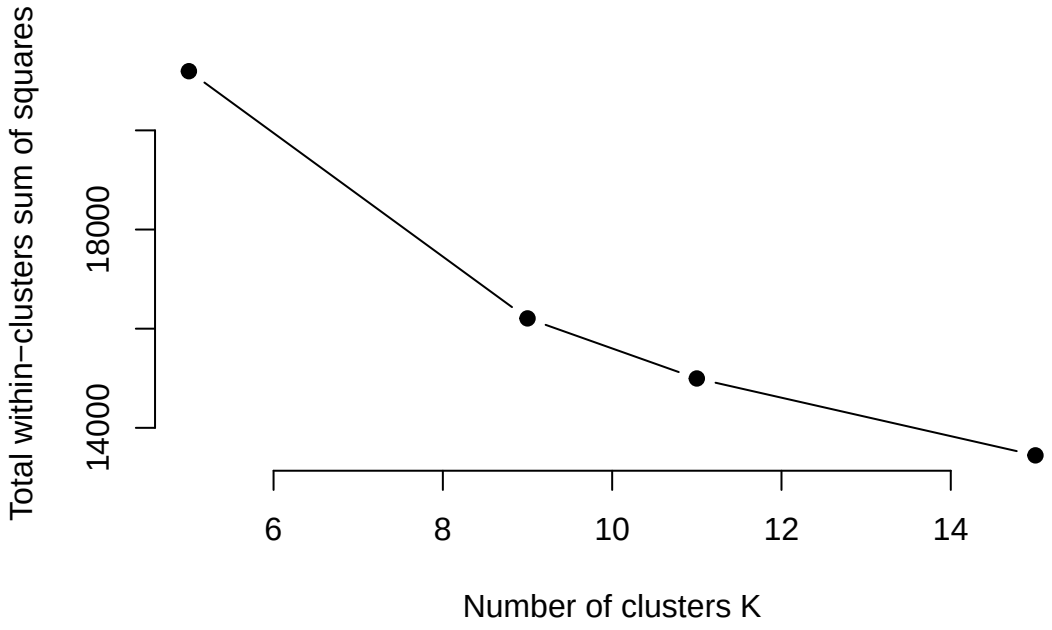
- **Elbow method**. First introduced in 1953 by Thorndike [10], the elbow methods looks at the total with cluster sum of squares as a function of the number of clusters (WCSS). When adding clusters doesn't decrease the WCSS by a sufficient amount, one can consider stopping. This method thus provides 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 experiment, conserve (1) with within cluster sum of squares; (2) the labels assigned to all genes.

```
all_possible_n_clusters = c(5, 9, 11, 15)
all_clustering = list()
wss_values = list()

i = 1
for(n_cluster in all_possible_n_clusters){
    clustering_results = splines_kmeans(
    y, splines_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
}
```

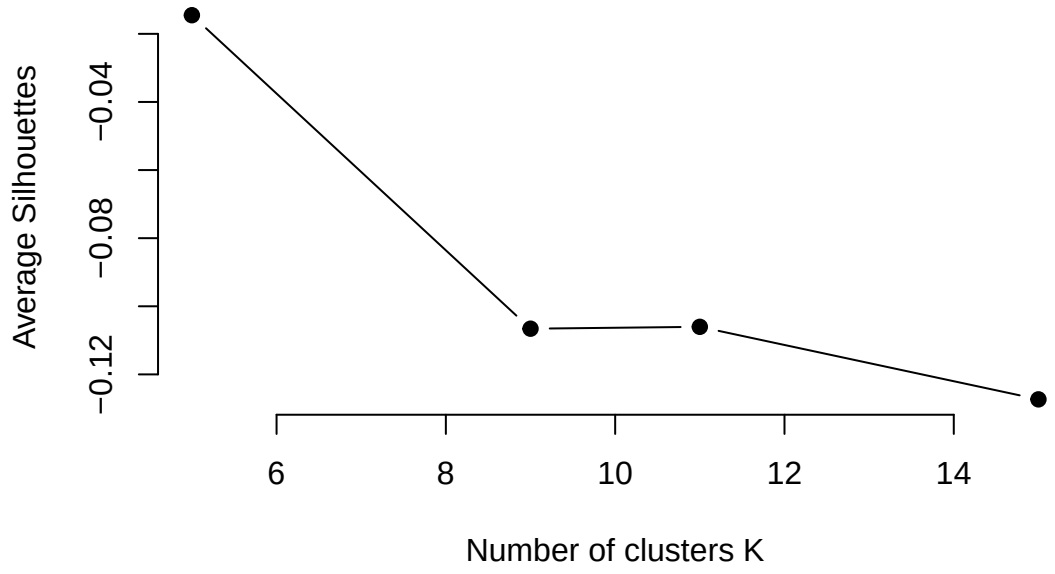
Elbow method The Elbow method to choose the number of clusters relies on visualization aid to choose the number of cluster. The method relies on plotting the within cluster sum of squares as a function of the number of clusters. At some point, the WCSS will stop dropping, giving an angle in the graph. The number of cluster is chosen at this "Elbow point."

```
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:

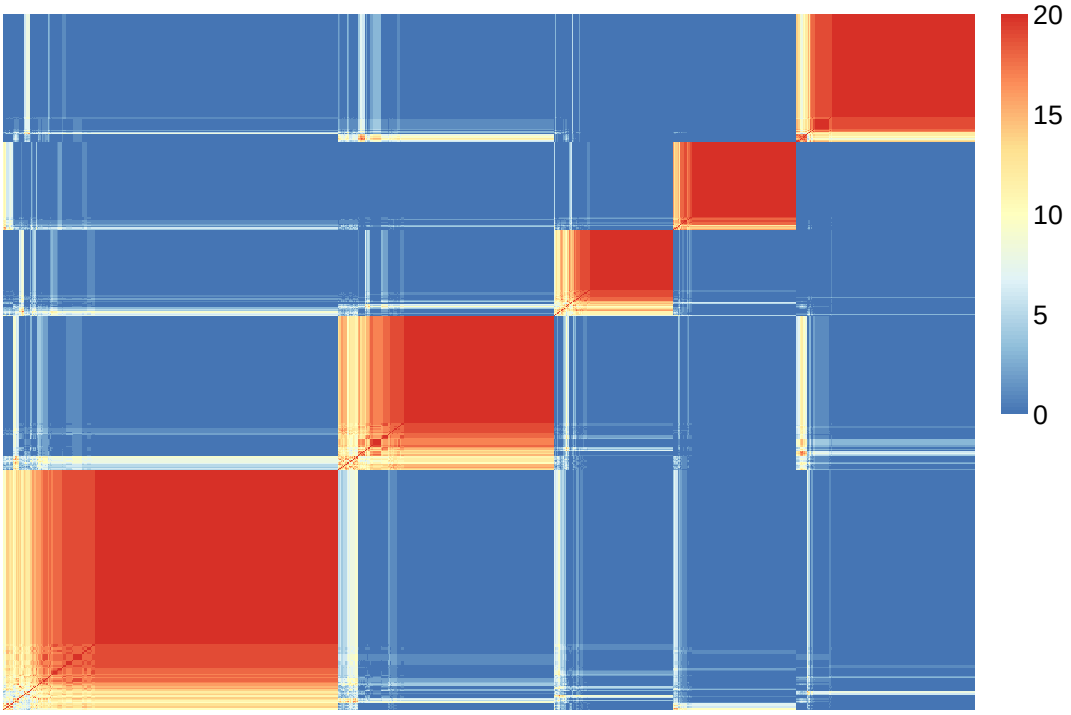
```
n_genes = dim(y)[1]
indices = sample(1:dim(y)[1], n_genes, replace=TRUE)

bootstrapped_y = y[indices, ]
```

Using the subsampling strategy, keeping 80% of the genes:

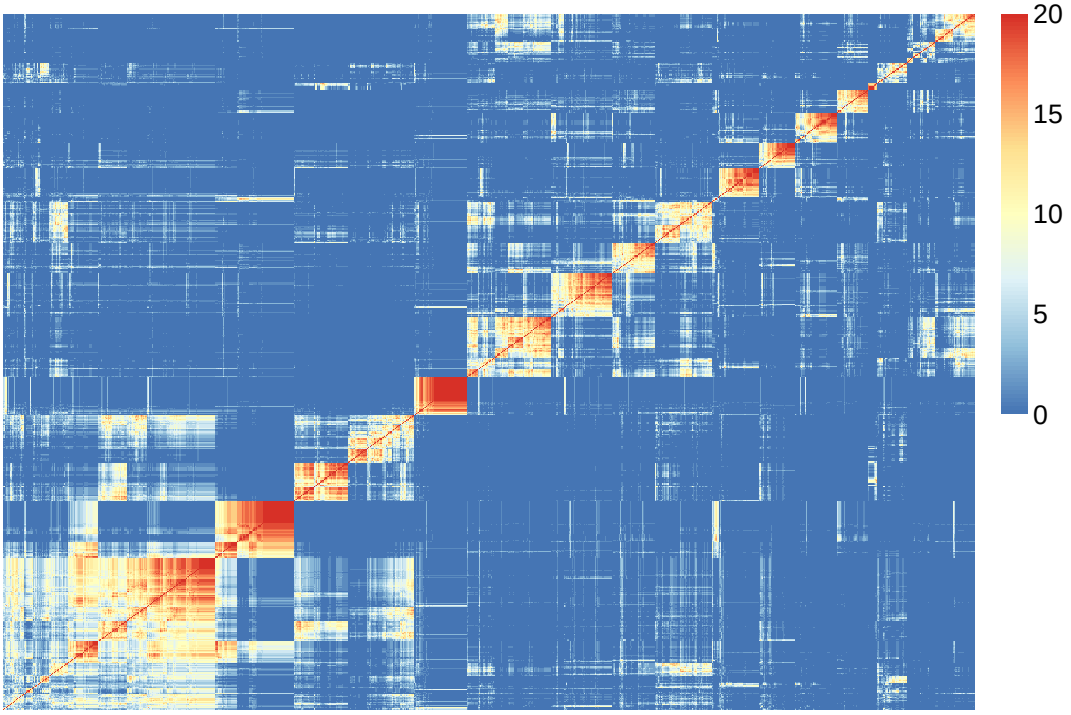
```
subsample_proportion = 1
indices = sample(1:dim(y)[1], n_genes * subsample_proportion, replace=FALSE)
subsampled_y = y[indices, ]
```

Here we plot to stability matrix of the top 1000 genes for $k = 5$ and $k = 20$.

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



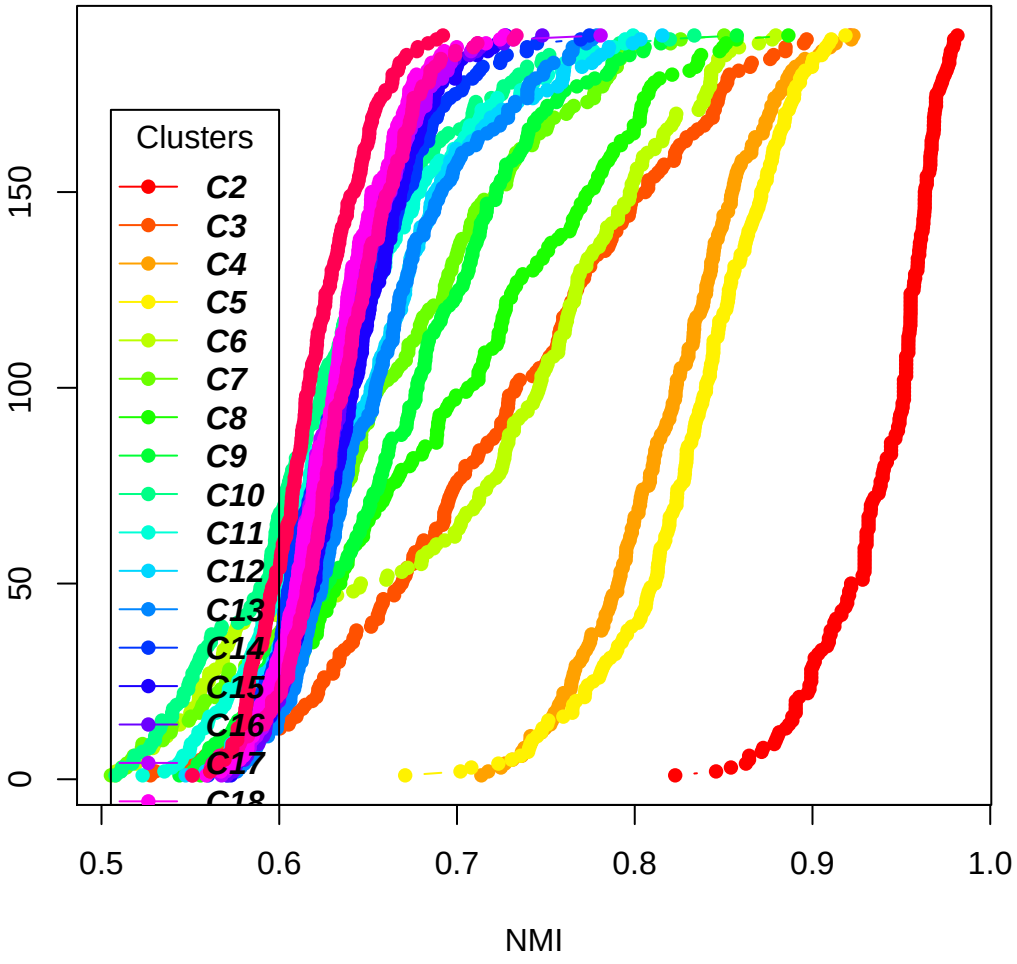

```
stability_20 = read.table("results/stability_20.tsv", sep="\t")
consensus_matrix_stability_20 = consensus_matrix(stability_20,
                            scale=FALSE)
aheatmap(consensus_matrix_stability_20[1:1000, 1:1000], Rowv=FALSE,
     Colv=FALSE,
     treeheight=0)
```



The model explorer strategy The model explorer algorithm [11] proposes to estimate the number of clusters exploiting the observation that if the number of clusters is correct, the clustering results are stable to bootstrapping. 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. First, choose a similarity measure between two partitions or clusters $S(C_1, C_2)$. Examples are the normalized mutual information or Fowlkes-Mallows. Then perform n bootstrap experiments to estimate the cluster centroids, followed by a step of assigning a label to all data points. Finally, compute the pairwise similarity measure between all bootstrapped partition, and plot the cumulative density of the obtained scores.

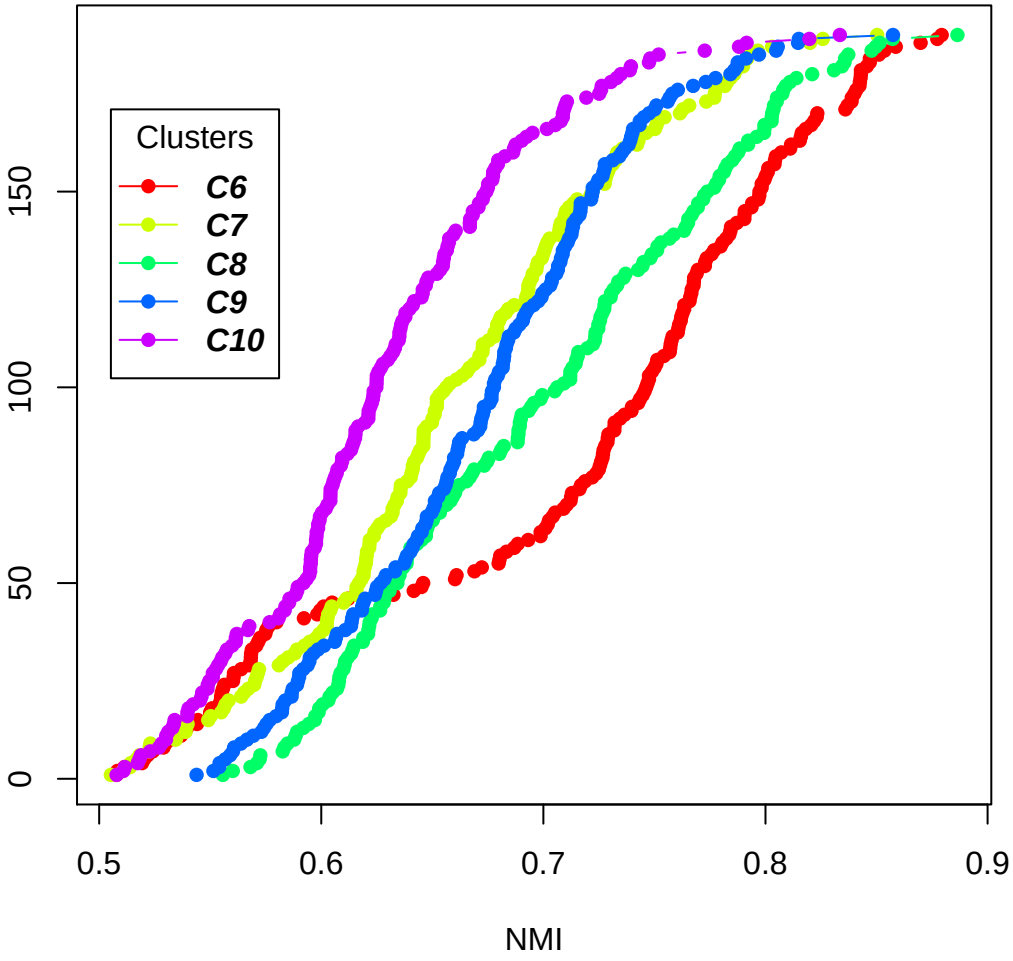
```
plot_model_explorer(all_labels)
```



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.

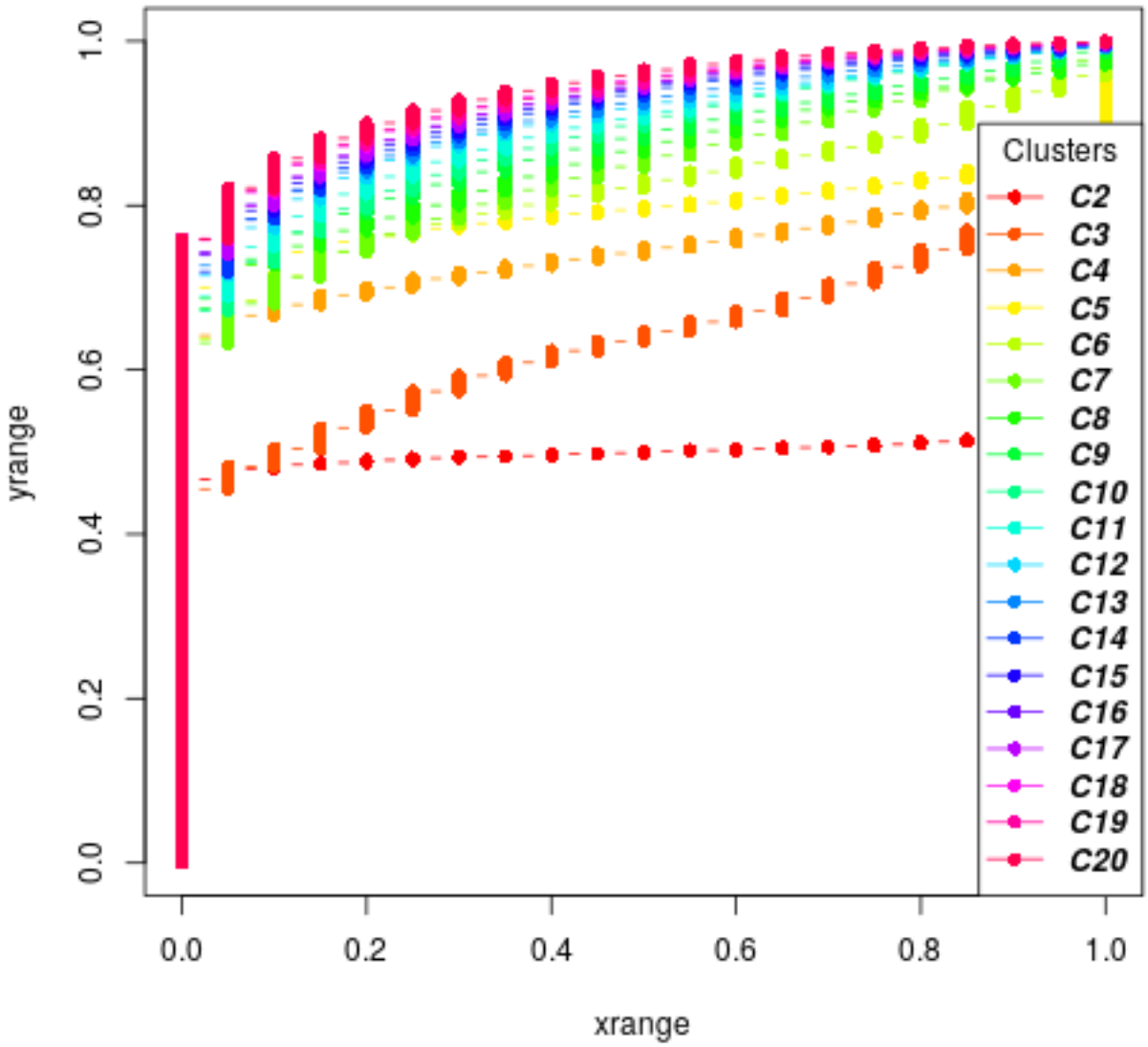
Now, replot the same model explorer, but only for the clustering experiments $k = 6$, $k = 7$, $k = 8$, $k = 9$ and $k = 10$ so that we can see more clearly the stability measures in that range.

```
clusters = c("C6", "C7", "C8", "C9", "C10")
selected_labels = all_labels[clusters]
plot_model_explorer(selected_labels)
```

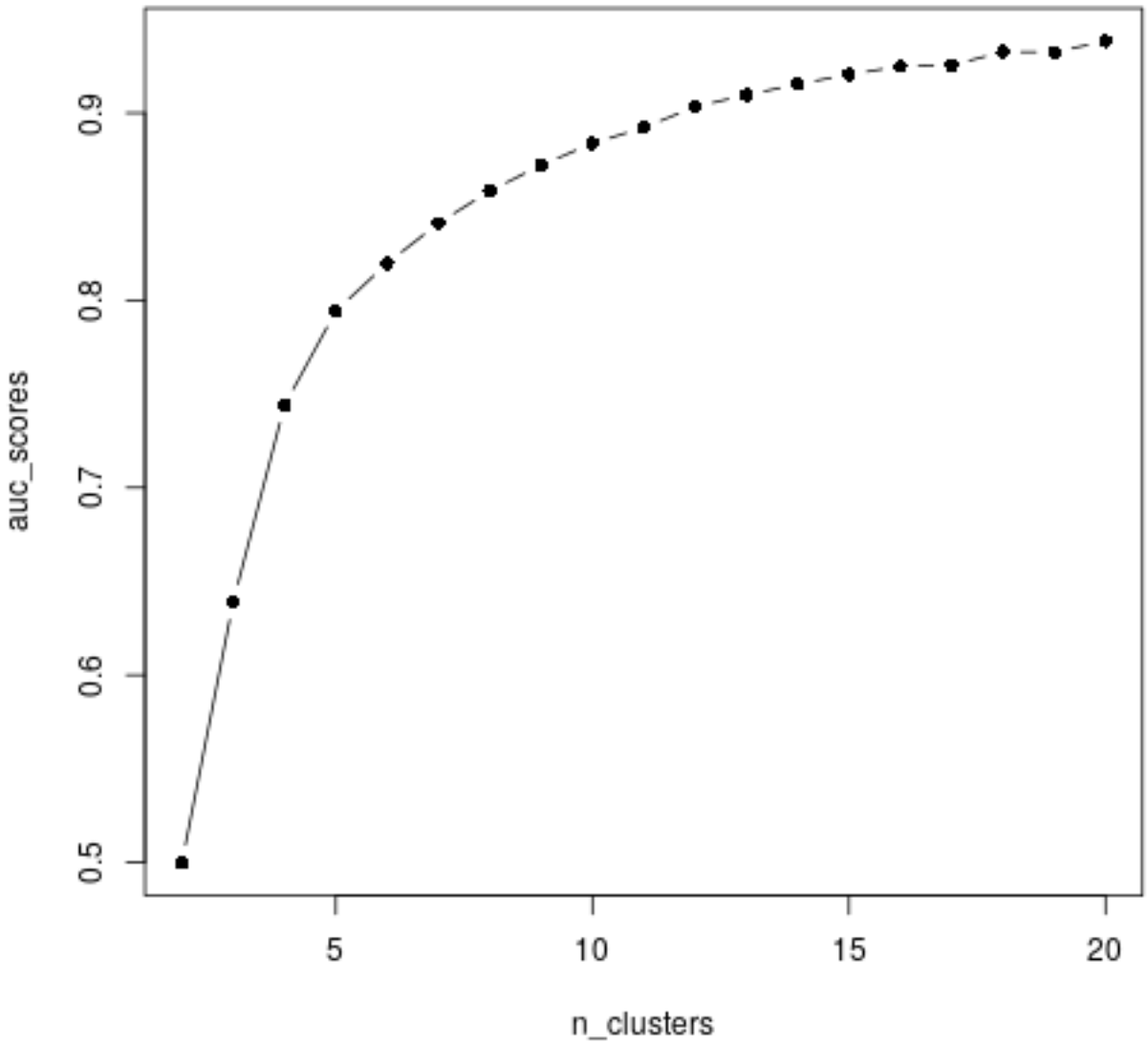


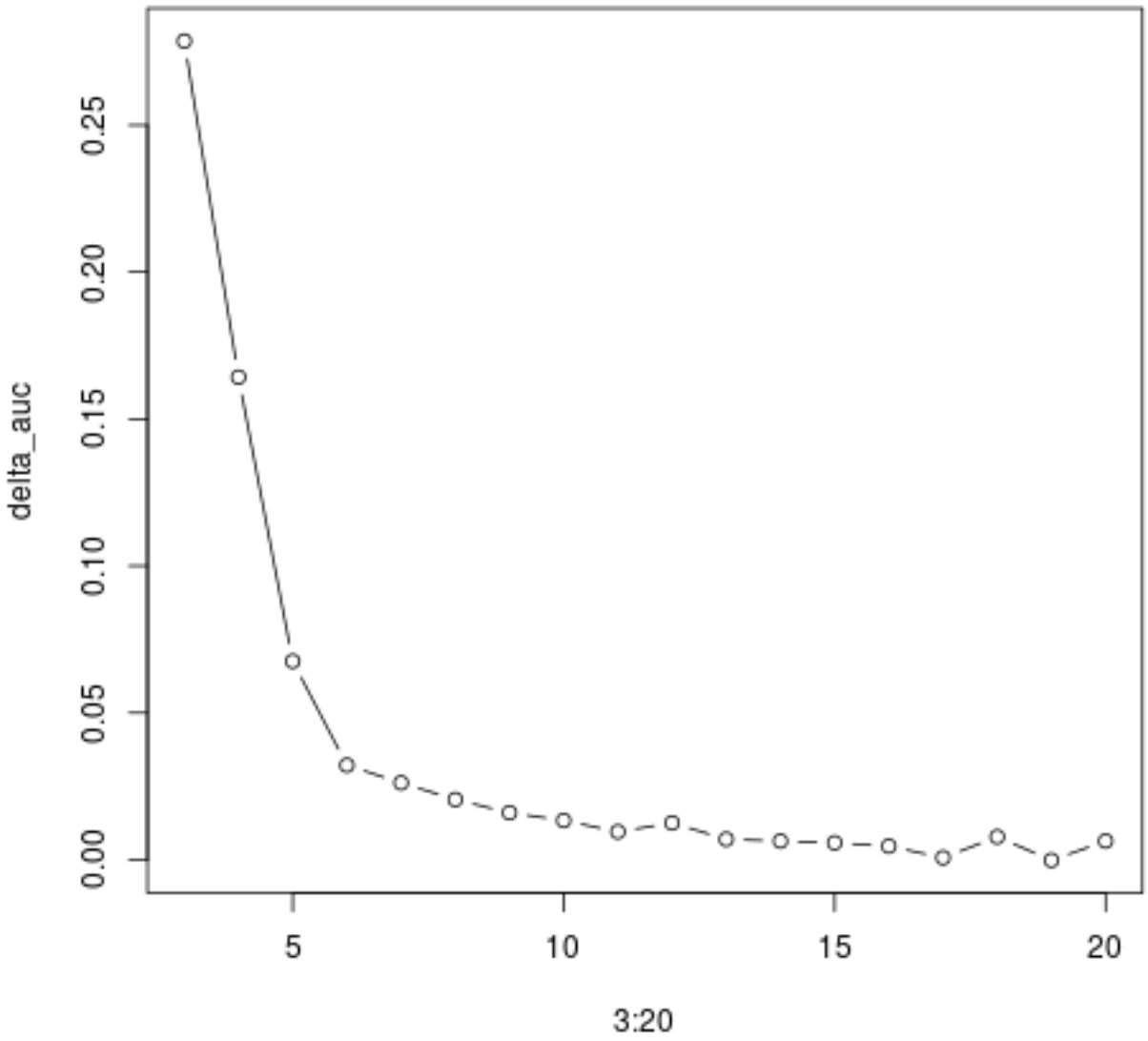
Consensus clustering as a way to find k The consensus clustering [12] relies on a similar idea but instead of looking at the cumulative density of similarity measures of bootstrapped clustering, the authors suggests plotting the cumulative density of elements of the consensus matrix.

```
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
0.5
0.6
0.7
0.8
0.9
5
10
15
20
n_clusters



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 gene obtained 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` [13] and `KEGGprofile` [14] 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"),
        filters="refseq_mrna", values=genes,
        mart=ensembl)["entrezgene"]
genes = as.vector(unlist(genes))
pathways = find_enriched_pathway(
    genes, species="mmu",
    download_latest=TRUE)$stastic
```

	Pathway	Percentage	Adj. p-value
05169	Epstein-Barr virus infection	37	0
04060	Cytokine-cytokine receptor interaction	31	0
05164	Influenza A	38	0
04668	TNF signaling pathway	45	0
04621	NOD-like receptor signaling pathway	34	0
05162	Measles	37	0
04630	JAK-STAT signaling pathway	33	0
05160	Hepatitis C	34	0
04620	Toll-like receptor signaling pathway	40	0
05167	Kaposi sarcoma-associated herpesvirus infection	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` [15] 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
5	GO:0003002	regionalization	172	153	134.46	0.00024	0.2416
6	GO:0048562	embryonic organ morphogenesis	154	137	120.39	0.00035	0.2936
7	GO:0010970	transport along microtubule	81	74	63.32	0.00057	0.4098
8	GO:0060173	limb development	92	84	71.92	0.00070	0.4404
9	GO:0060070	canonical Wnt signaling pathway	147	130	114.92	0.00090	0.5033
10	GO:0055114	oxidation-reduction process	531	457	415.10	0.00135	0.6223
11	GO:0019395	fatty acid oxidation	53	51	41.43	0.00136	0.6223
12	GO:0006805	xenobiotic metabolic process	44	42	34.40	0.00168	0.7046
13	GO:0090596	sensory organ morphogenesis	134	118	104.75	0.00224	0.8197
14	GO:0010811	positive regulation of cell-substrate ad...	78	71	60.98	0.00228	0.8197
15	GO:0060411	cardiac septum morphogenesis	33	32	25.80	0.00298	0.9999
16	GO:0051056	regulation of small GTPase mediated sign...	151	131	118.04	0.00349	1.0000
17	GO:0042738	exogenous drug catabolic process	22	22	17.20	0.00441	1.0000
18	GO:1901381	positive regulation of potassium ion tra...	22	22	17.20	0.00441	1.0000
19	GO:0007368	determination of left/right symmetry	67	61	52.38	0.00464	1.0000

Session information

```
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         topGO_2.32.0
##  [4] SparseM_1.77         GO.db_3.6.0          AnnotationDbi_1.42.1
##  [7] IRanges_2.14.12      S4Vectors_0.18.3     graph_1.58.2
## [10] viridis_0.5.1        viridisLite_0.3.0    KEGGprofile_1.22.0
## [13] RCurl_1.95-4.12      bitops_1.0-6         NMF_0.21.0
## [16] Biobase_2.40.0       BiocGenerics_0.26.0  cluster_2.0.9
## [19] rngtools_1.3.1.1     pkgmaker_0.27        registry_0.5-1
## [22] ggplot2_3.1.1        biomaRt_2.36.1       BiocStyle_2.8.2
## [25] kableExtra_1.1.0     knitr_1.22           limma_3.36.5
## [28] usethis_1.5.0        devtools_2.0.2       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_0.3.4             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] munsell_0.5.0           magic_1.5-9
## [97] sessioninfo_1.1.1
```

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`. Data used in this workflow are available from NCBI GEO, accession GSE95601.

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] 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.

[2] 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`.

[3] 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`.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

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

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

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