A pipeline to analyse time-course gene expression data

true   true

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) quality control and normalization of the dataset; (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.

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 (Shalek et al. 2014; Habib et al. 2016; Trapnell et al. 2014) 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` (Storey et al. 2005), functional component analysis based models (Wu and Wu 2013), time-course permutation tests (Taesung Park et al. 2003), and multiple testing strategies to combine single time point differential expression analysis (Wenguang and Zhi 2011). However, with long time course datasets, 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.

---

[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.

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`.
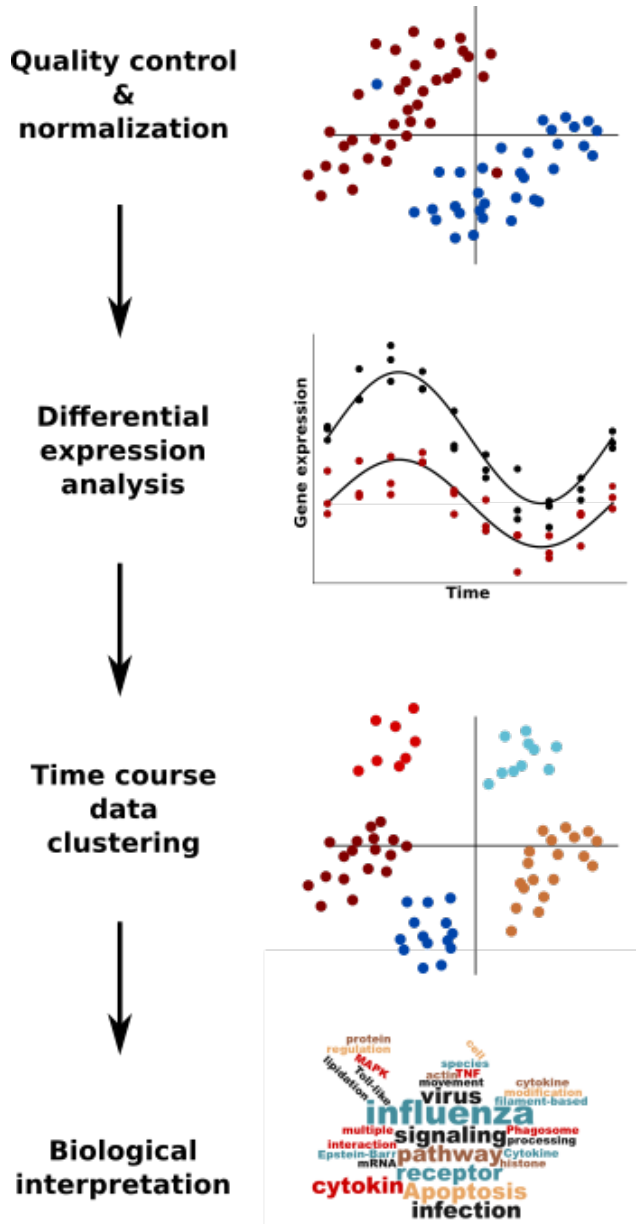

Figure 1: Workflow for analyzing time-course datasets.

Installation and setup

`moanin` and `timecoursedata` are available from bioconductor, and can be installed using the `install` function in the package `BiocManager`, along with the corresponding package that contains time course datasets we will use:

```
library(BiocManager)
# Need to use the development version for now.
```

```
BiocManager::install(version="devel")
BiocManager::valid()

BiocManager::install("timecoursedata")
BiocManager::install("moanin")
```

The following additional packages are needed for this workflow:

```
# From Github
library(moanin)
library(timecoursedata)

# From CRAN
library(NMF)
library(ggfortify)

# From Bioconductor
library(topGO)
library(biomaRt)
library(KEGGprofile)
library(BiocWorkflowTools) #Needed for compiling the .Rmd script
```

If Bioconductor is installed, the CRAN and Bioconductor packages above can be installed via

```
BiocManager::install(
    c("NMF", "ggfortify", "topGO", "biomaRt",
      "KEGGprofile", "BiocWorkflowTools", "timecoursedata", ))
```

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) (Shoemaker 2015). 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 additional 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 cytokine 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, (Varoquaux et al. 2019) leverages a similar set of steps to analyse RNA-seq data of the lifetime transcriptomic response of the crop *S. bicolor* to drought.

Overview of the data

First let's load the data. The package `moanin` contains the normalized data and metadata of (Shoemaker 2015).

```
# 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. We also reorder the factor `Group` which describes the treatments so that the treatments are ordered from low to high pathogeny (with Control being first).

```r
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)

# 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)])
```

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. However, in what follows, we do not show the steps for normalization, as these are specific to the platform (microarray); the code for the normalization can be found in as a supplemental file (Abrams et al. 2019; T. Park et al. 2003).

Instead, we focus on common steps for exploratory analysis of the data, including for the purpose of quality control. These steps are not specific to time course data, but could be applied for any gene expression analysis. For this reason, we will not print the detailed code that is needed for this part of the analysis; interested readers can examine the **rmarkdown** code that accompanies this workflow.

Exploratory analysis and quality control

Typically, two quality control and exploratory analysis steps are 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. In Figure 2, we plot the distribution of variance across all 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,]
```
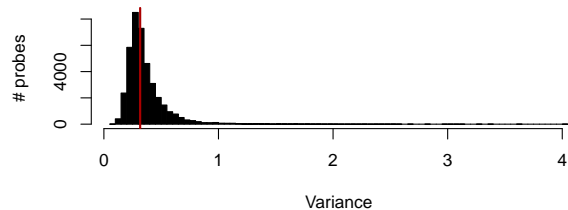

Figure 2: Distribution of variance

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

In Figure 3, we visualize the data by plotting the PC components, with samples colored by either its condition
(top row) or its sampling time (bottom row) and each replicate a different symbol. We can see a large
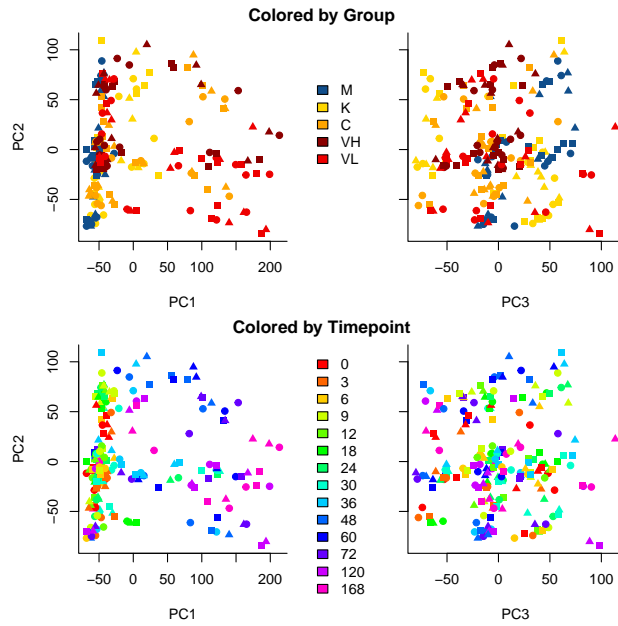difference in later time points.

Figure 3: Plot of PC components: PC2 vs PC1 (right) and PC2 vs PC3 (left). Samples are colored by condition (top row) and sampling time (bottom row).

We also plot in Figure 4 the pearson correlation between each sample as a heatmap diagram (using the function `aheatmap` in `NMF`). We order the samples by their Group (treatment) and Timepoint (the time of sampling).
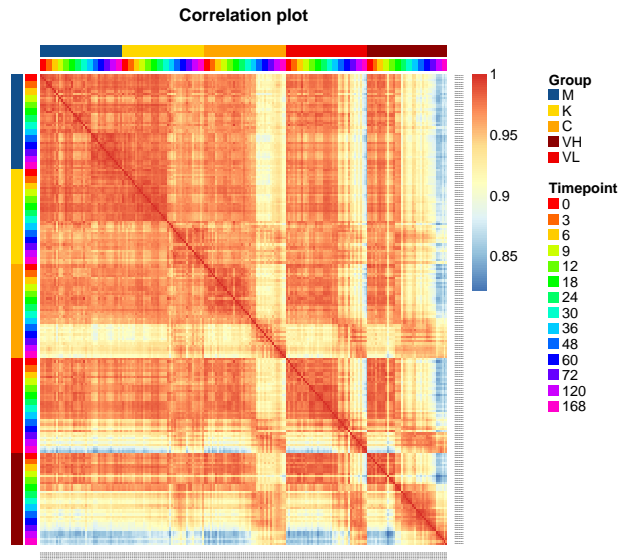

Figure 4: Heatmap of the correlation between samples

We can already see interesting patterns emerging from the correlation plot. First, the cross-correlation amongst samples taken 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 specific 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 (Storey et al. 2005), 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 (Ritchie et al. 2015; Robinson, McCarthy, and Smyth 2010; Love, Huber, and Anders 2014). 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 dataset we are considering), which will alter slightly the types of questions we are interested in.

DE Analysis in `moanin` Our package `moanin` provides functionality for performing both of these types of approaches, though our focus is on the global approach, specifically by fitting spline models to the genes.

In both situations, we first need to set up an object (a `Moanin` object) to hold the meta data, as well as information for fitting the spline model (formula, number of degrees of freedom of the splines, . . .). The `Moanin` object will contain information used throughout this analysis, in particular the condition and timepoints of each sample and the basis matrix for fitting splines models.

We start by creating the `Moanin` 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 to be used in the functional modeling. The metadata `data.frame` object should contain at least two columns: one named `Group`, containing the treatment 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` object for our data:

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

The main operation of the `create_moanin_model` function, in addition to holding the necessary meta data of the samples, is to create a basis matrix for the splines fit. This matrix gives the evaluation of all of the spline basis functions for each sample (as such, replicate samples will have the same values). By default, `create_moanin_model` will create the spline basis functions which will lead to a different spline fit for every group (as defined by the `Group` variable in the `meta` data.frame). This is done by the following `R` formula syntax:

```
formula = ~Group:ns(Timepoint, df=degrees_of_fredoom) + Group + 0.
```

Alternatively, the user can provide a formula of their own, or simply provide the basis matrix themselves.

We can see this information when we print the object:

```
show(moanin_model)
```

```
## Moanin object on 209 samples containing the following information:
## Group variable given by 'Group' with the following levels:
##  M  K  C VL VH
## 42 42 42 42 41
## Time variable given by 'Timepoint'
## Basis matrix with 35 basis_matrix functions
## Basis matrix was constructed with the following spline_formula
## ~Group + Group:splines::ns(Timepoint, df = 6) + 0
##
## Information about the data (a SummarizedExperiment object):
## class: SummarizedExperiment
## dim: 39544 209
## metadata(0):
## assays(1): ''
## rownames(39544): NM_009912 NM_008725 ... NM_010201.1 AK078781
## rowData names(0):
## colnames(209): GSM1557140 GSM1557141 ... GSM1557347 GSM1557348
## colData names(5): '' Group Replicate Timepoint WeeklyGroup
```

The `Moanin` class extends the `SummarizedExperiment` class of Bioconductor, so that the data as well as the meta data and our spline information are all held in one object. This means that the object can be subsetted just like the data matrix, and the corresponding meta data and basis function evaluations will be similarly subsetted:

```
dim(moanin_model)
```

```
## [1] 39544   209
```

```
moanin_model[1:10,1:3]
```

```
## Moanin object on 3 samples containing the following information:
## Group variable given by 'Group' with the following levels:
##  M  K  C VL VH
##  0  3  0  0  0
## Time variable given by 'Timepoint'
## Basis matrix with 35 basis_matrix functions
## Basis matrix was constructed with the following spline_formula
## ~Group + Group:splines::ns(Timepoint, df = 6) + 0
##
## Information about the data (a SummarizedExperiment object):
## class: SummarizedExperiment
## dim: 10 3
## metadata(0):
```

```
## assays(1): ''
## rownames(10): NM_009912 NM_008725 ... NM_013782 NM_028622
## rowData names(0):
## colnames(3): GSM1557140 GSM1557141 GSM1557142
## colData names(5): '' Group Replicate Timepoint WeeklyGroup
```

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` (Ritchie et al. 2015) on the set of contrasts provided. By default, `moanin` expects RNA-Seq gene counts, and will estimate voom weights.

Here, we show an example where we create contrasts that will be 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 (Ritchie et al. 2015)).

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

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

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

```
## [1] "M.0-VL.0"   "M.3-VL.3"   "M.6-VL.6"   "M.9-VL.9"   "M.12-VL.12"
```

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

```r
weekly_de_analysis = DE_timepoints( 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_stat
NM_009912	0.2728	0.4746	-1.101
NM_008725	0.8509	0.9243	0.1884
NM_007473	0.08804	0.2193	-1.718
ENSMUST00000094955	0.008188	0.0368	-2.682
NM_001042489	0.4192	0.6208	-0.8102
NM_008159	0.1232	0.2778	-1.551
NM_001013813	0.05653	0.1594	1.923
AK039774	0.01504	0.05902	2.461
NM_013782	0.8754	0.9377	-0.1571
NM_028622	0.7668	0.8759	-0.2971

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).

In Figure 5, we show 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.
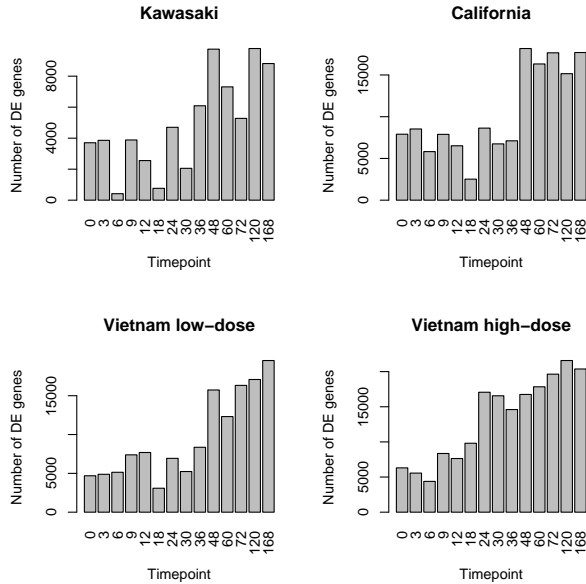

Figure 5: Number of genes found DE in each timepoint when comparing the influenza strains to control.

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 are some technical or biological artifacts about the samples for 6H that are 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. In Figure 6 we show the plots of the first 10 such genes. 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 that results in a lack of significance in 72H.
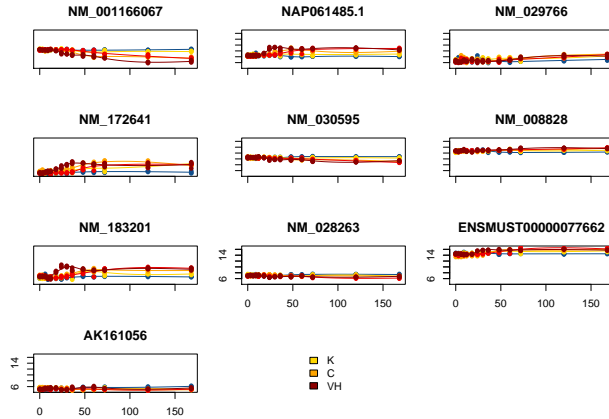
Figure 6: The expression values over time for 10 genes found significant DE in timepoints 48,60,120,168H, but not in the 72H timepoint.

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 (Storey et al. 2005) 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 not only fitting a splines function for each gene, but also 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 fit spline functions for each gene and test the entire mean function (and unlike `DE_timepoints`, therefore does not require the extra 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
# to the time-course information, the latter corresponding to the
# treatment).
DE_results = DE_timecourse( moanin_model, timecourse_contrasts,
    use_voom_weights=FALSE)
```

The output from `DE_timecourse` is a matrix of (raw) p-values and Benjamini-Hochberg corrected q-values for each comparison.

```
names(DE_results)
```

```
##  [1] "M-K_stat"  "M-K_pval"  "M-K_qval"  "M-C_stat"  "M-C_pval"  "M-C_qval"
##  [7] "M-VL_stat" "M-VL_pval" "M-VL_qval" "M-VH_stat" "M-VH_pval" "M-VH_qval"
```

For convenience we will separate these into two matrices.

```
pval_columns = colnames(DE_results)[
    grepl("pval", colnames(DE_results))]
qval_columns = colnames(DE_results)[
```

11

```
    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 (Figure **??**). This corresponds to between 30% to 70% of the genes found differentially expressed in this time-course experiment.
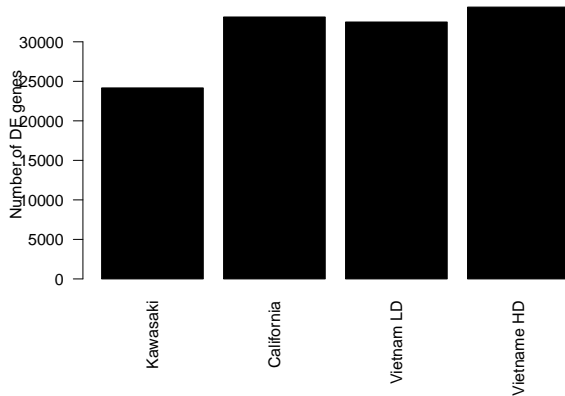

Figure 7: Distribution of DE genes per condition based on global splines analysis

Log-fold Change for Time Course Data

The next step in a classical differential expression analysis is typically to assess the effect of the treatment by calculating for each gene the log fold change in the gene expression between the treatment and control.

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 under-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. This is done via the function `estimate_log_fold_change` which takes as arguments the data, the `Moanin` object, the contrasts to evaluate, and the method to use to estimate the log-fold change.

Individual Timepoints The first method ("timely") gives a simple interface to compute the log-fold change for each individual timepoints.

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

	M-K:0	M-K:3	M-K:6	M-K:9	M-K:12
NM_009912	-0.5662	-0.2455	-0.4097	-0.332	-0.05026
NM_008725	0.9947	-3.461	0.6561	-1.211	-0.1999
NM_007473	-0.5177	-0.3147	0.2328	-0.3446	0.04762
ENSMUST00000094955	-0.2202	0.2567	-0.03584	-0.04697	-0.07075
NM_001042489	-0.2476	-0.2044	-0.3428	-0.3931	-0.4822
NM_008159	-0.7466	-0.06789	-0.3072	-0.7067	-0.4967

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

Cumulative Effect Sometimes, a single value per gene for each contrast is more useful, and

`estimate_log_fold_change` used above provides several options for this as well. Here is a table of the possible ways `estimate_log_fold_change` can compute log-fold change values (including `timely` discussed above).

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

The method "timecourse" tries to capture the overall strength and direction of the response in the following way: we leverage the timepoint by timepoint log-fold change $\text{lfc}(t)$, 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, however, that when a gene is not consistently up- or down-regulated the estimation of the direction will not accurately represent the changes observed.

We demonstrate several of these methods on our data.

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

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

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

log_fold_change_min = estimate_log_fold_change(
    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.

	M-K	M-C	M-VL	M-VH
NM_009912	-0.4091	-0.7525	-1.025	-1.42
NM_008725	1.575	2.008	2.543	2.618
NM_007473	-0.4158	-0.3346	-0.2127	0.3643
ENSMUST00000094955	-0.1126	-0.3763	-0.1558	-0.3091
NM_001042489	-0.2341	-0.4104	0.3085	0.4301

In Figure 8 we plot the log-fold change summary of each of these methods against each other. We see that each method captures different elements of the time-course data, for example, overall change versus the largest change.
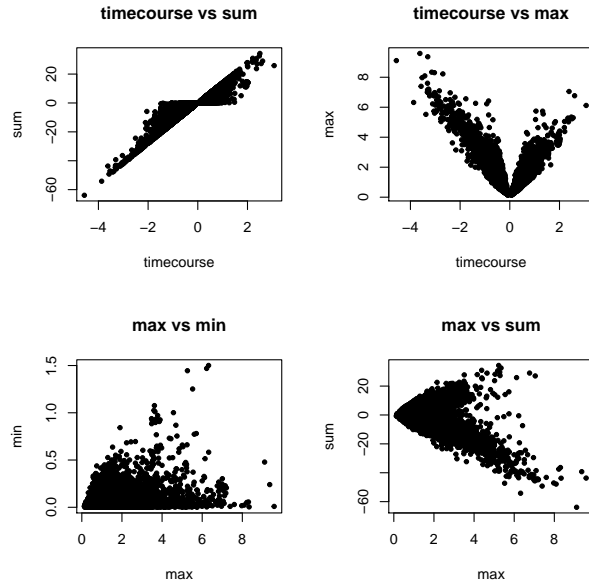
Figure 8: Comparison of the different methods of calculating log-fold change per gene

With a single measure of log-fold change and the p-value, we can now look at the traditional volcano plot. In Figure 9, we show the example of a volcano plot for the comparison of the control to the Kawasaki strain, using the "timecourse" method of calculating log fold change.

```
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")
```
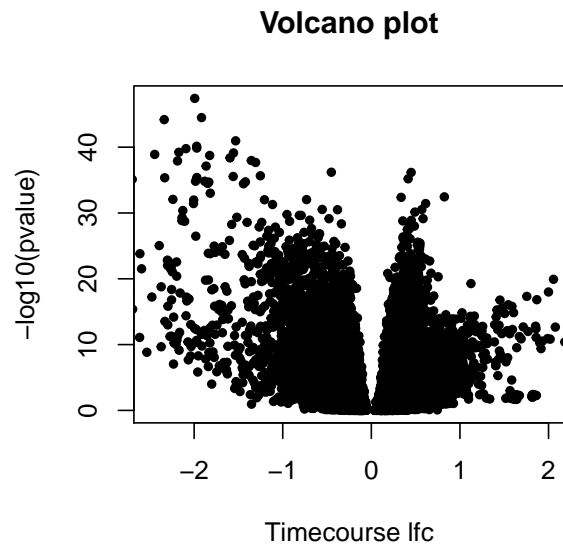
Volcano plot

Figure 9: Volcano plot for the comparison of the Kawasaki strain to control, where fold-change is calculated with the timecourse method

Visualizing Genes of Interest

The package moanin also provides a simple utility function (`plot_splines_data`) to visualize gene time-course data from different conditions. In Figure 10, we plot the 10 genes with the smallest p-values.

```
top_DE_genes_pval = names(sort(pvalue)[1:10])
plot_splines_data(moanin_model,subset_data=top_DE_genes_pval,
    colors=ann_colors$Group, smooth=TRUE,
    mar=c(1.5,2.5,2,0.1))
```
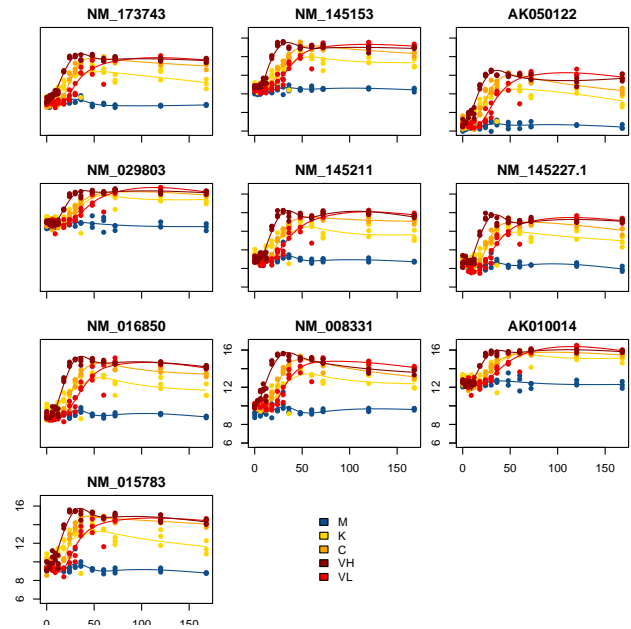
Figure 10: Top 10 genes with the smallest p-values

For each gene, the individual data points are plotted against time and color coded by their condition. Further, a fitted spline function for each group is plotted to aid in comparing global trends across conditions.

Figure 11, 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(moanin_model, subset_data=top_DE_genes_lfc,
    colors=ann_colors$Group, smooth=TRUE,
    mar=c(1.5,2.5,2,0.1))
```
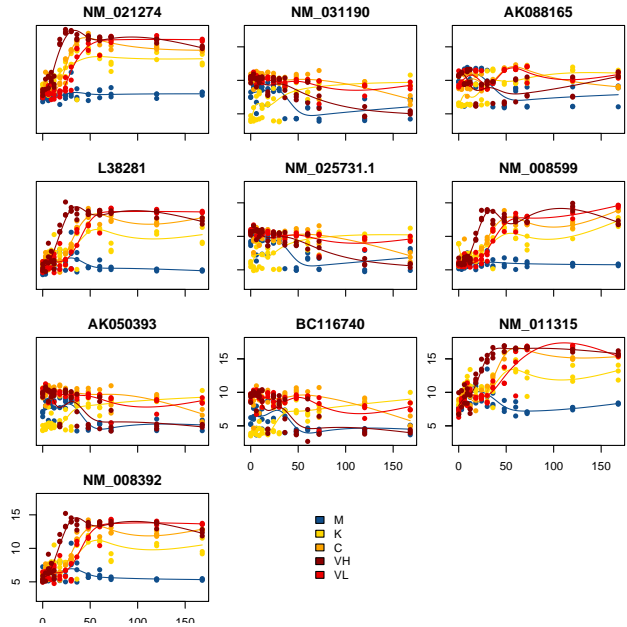
Figure 11: Top 10 genes with largest absolute timecourse log-fold change

In examining these visualizations, 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 impairs 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.

Filtering 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. Reducing the set of genes on which to perform the clustering allows to estimate the centroids of the clusters with more stability.

To do this, we first aggregate all p-values obtained during the time-course differential expression step in a single p-value using Fisher's method (Fisher 1925) (`pvalues_fisher_method`). Next we select all the genes which have a Fisher-adjusted p-value below 0.05 and a log-fold change of at least two for at least one condition and one time-point.

```
# 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.
# by subsetting the `moanin_model`, which contains the data.
de_moanin_model = moanin_model[genes_to_keep,]
```

Clustering Based on Spline Fits

After filtering, we are left with 5388 genes. We can then apply a clustering routine. As observed by looking at genes found differentially expressed, many genes share a similar gene expression pattern, but on different scales.

We thus propose the following adaptation of k-means:

- **Splines estimation**: for each gene, fit the splines function with the basis of your choice (as contained in the `Moanin` 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.

The rescaling of the splines is similar to the rescaling genes by their standard deviation, which is common for gene expression studies without a time component.

These clustering steps are performed by the `splines_kmeans` function in `moanin`. 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(de_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 cluster assignments given by the `kmeans` step of each gene

We then use the `plot_splines_data` function, only now applied to the centroids, to visualize the centroids of each cluster obtained with the splines k-means model (Figure 12).

```
plot_splines_data(de_moanin_model,
    data=kmeans_clusters$centroids,
    colors=ann_colors$Group,
    smooth=TRUE)
```
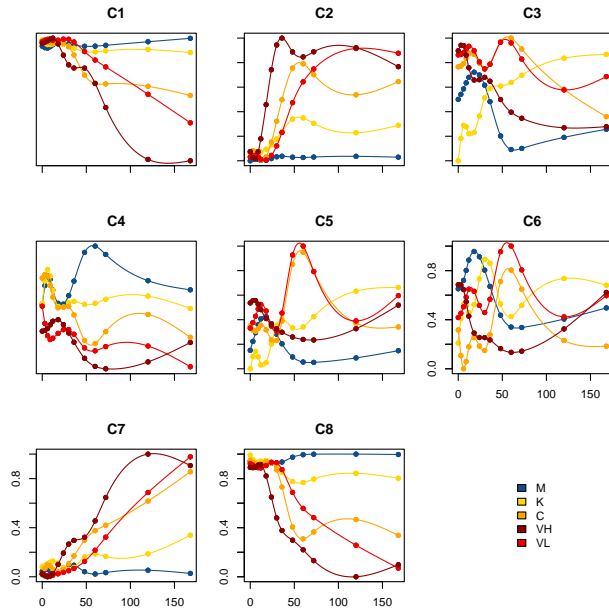
18

Figure 12: K-means centroids

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. In Figure 13, we plot a few of the actual genes assigned to cluster 2 with the estimated centroid overlaid:

```
cluster_of_interest = 2
cluster2Genes = names(
    kmeans_clusters$clusters[kmeans_clusters$clusters==cluster_of_interest])
plot_splines_data(de_moanin_model,
    centroid=kmeans_clusters$centroids[cluster_of_interest,],
    colors=ann_colors$Group, smooth=TRUE,simpleY =FALSE,
    subset_data=cluster2Genes[3:6],
    mar=c(1.5,2.5,2,0.1))
```
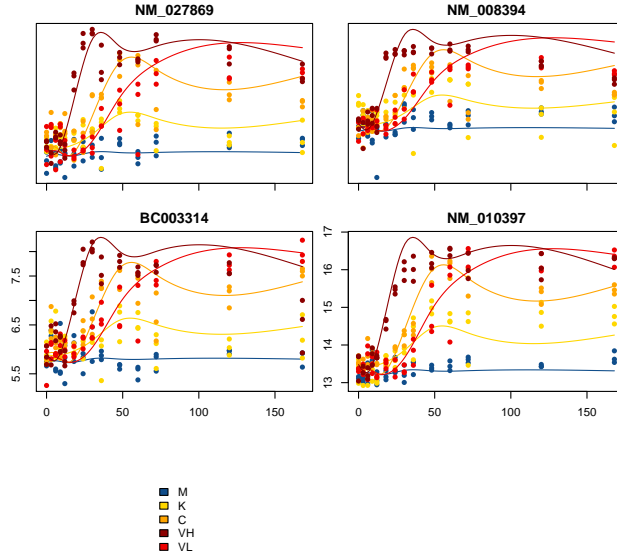
Figure 13: Genes from cluster 2

Assigning genes to clusters

As we can see, while these genes have some similarity with the pattern of the cluster centroids, these particular genes are not the best examples of the cluster, in the sense of matching the centroid estimates. Because of this variability in how well the genes fit a cluster, we would like to be able to score how well a gene fits a cluster.

Furthermore, we arbitrarily chose a subset of genes based on our filter, and we would like to have a mechanism to assign all genes to a cluster.

Thus, the next step in the clustering portion of the workflow is a scoring and label step. Each gene is given a score that 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 score then allows us to assign all genes to a cluster (i.e. a label) based on the cluster for which they have the best score, regardless of whether the gene was used in the clustering procedure.

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 and gives a cluster label to the gene if they have a sufficiently high score, as we explain above.

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

The `scores_and_labels` list contains three elements:

- `scores`: the matrix of shape `n_cluster` x `n_genes`, containing for each gene and each cluster, the goodness of fit as described above.
- `labels`: the labels for all of the genes with a sufficiently good goodness-of-fit score.
- `score_cutoff`: the cutoff used on `scores` to determine whether to assign a label

Assigning cluster labels: We could just assign each gene to a cluster based on which cluster gave the minimum score. By default, `splines_kmeans_score_and_label` does not do that, but rather requires a sufficiently low enough goodness-of-fit score. The criteria for being "sufficiently low" is based on looking at the distribution of the scores of all genes on all clusters (i.e. the entire `scores` matrix returned by `splines_kmeans_score_and_label`). A gene is then assigned to a cluster only if their best score is above the 50% percentile of that distribution, with the remaining genes getting `NA` as their assignment (this choice can be changed by `proportion_genes_to_label`). Note that because the distribution of scores of genes across *all* clusters is used, this is not equivalent to assigning 50% of genes to a cluster – it is possible that all genes are assigned to a cluster.

```
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` on our data, we now have 19772 genes that are assigned to a cluster.

We can visualize the impact of this filtering process by considering the distribution of goodness-of-fit scores for each cluster if we did not have the filtering cutoff of `splines_kmeans_score_and_label`, i.e. if we simply assign every gene to a cluster based on which gives them their minimum score. We display for each cluster, the scores of the genes assigned to that cluster (Figure 14), as compared to the cutoff value for what that score must be under our filtering procedure. We can do this by rerunning `splines_kmeans_score_and_label` and setting the filter `percentage_genes_to_label=1` (and we can speed up the calculation by giving our previously calculated score via the argument `previous_scores`)

```
# Get the best score and best label for all of the genes
# This time without filtering labels
# We can give the previous calculated scores to `previous_scores` to save time
unfiltered_scores = splines_kmeans_score_and_label(
    moanin_model, kmeans_clusters,
    proportion_genes_to_label=1,
    previous_scores=scores_and_labels$scores)
best_score = rowMin(scores_and_labels$scores)
best_label = unfiltered_scores$labels

par(mfrow=c(3, 3))
n_clusters = dim(kmeans_clusters$centroids)[1]
for(cluster_id in 1:n_clusters){
    hist(best_score[best_label==cluster_id],
     breaks=(1:50/50), xlim=c(0, 1),
     col="black", main=paste("C", cluster_id, sep=""),
     xlab="score", ylab="Num. genes")
    abline(v=scores_and_labels$score_cutoff, col="red", lwd=3, lty=2)
}
```
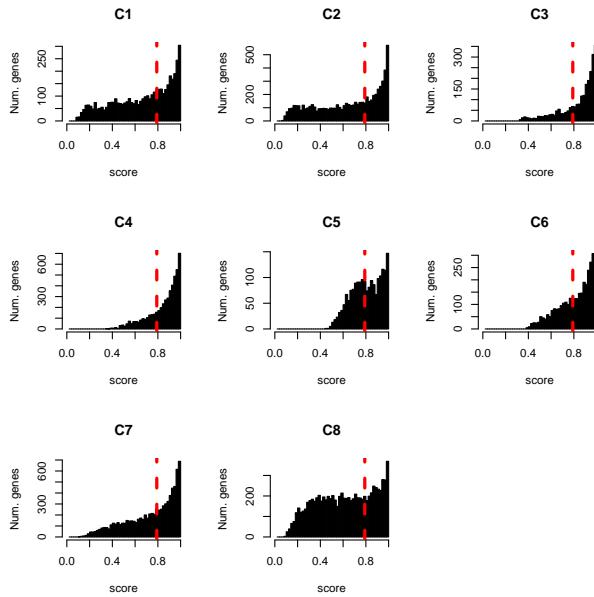
Figure 14: Distribution of goodness-of-fit scores for each cluster. The dashed red line indicates the scoring threshold: all genes with a score above this threshold will not be labeled.

Note that for some of the genes, their scores in all clusters is 1. Such genes fit poorly to all clusters and the assignment we did above to one cluster is done arbitrarily to the first matching cluster. This underlines the importance of filtering genes that fit poorly to clusters.

We can also investigate the differences between the labels provided by the splines k-means model and our scoring and labeling step, in terms of the number of genes assigned to each cluster (Figure 15).
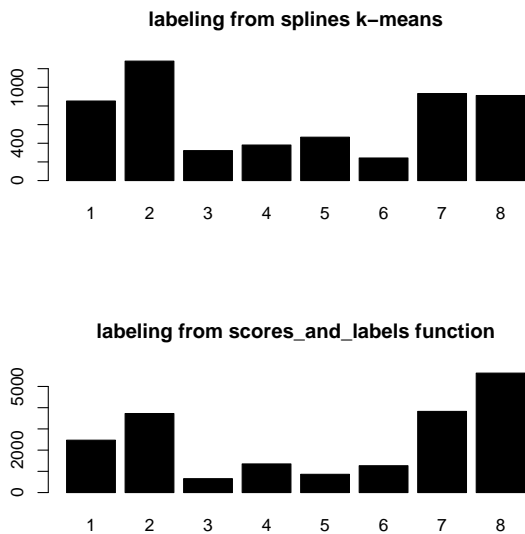

Figure 15: Number of genes assigned to each cluster based on kmeans criteria of nearest centroid (top) versus our goodness-of-fit filtering strategy (bottom)

Of the original 5388 genes used in the clustering, 4935 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 (Figure 16). 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.

Confusion matrix

714	0	4	0	0	0	0	124	1
0	1227	0	0	0	0	43	0	2
4	2	231	0	41	2	0	0	3
7	0	0	188	0	0	0	36	4
0	14	42	0	266	21	11	0	5
0	0	0	0	7	215	1	0	6
0	100	0	0	5	0	725	0	7
84	0	0	37	0	0	0	784	8
1	2	3	4	5	6	7	8	

Goodness–of–fit assignments

Figure 16: Confusion matrix between the two sets of labels

Of the four genes we plotted before, 4 of them remained assigned to cluster 2 after our scoring. We can again look at a few genes in cluster 2, only now pick the best scoring genes (Figure 17)

```
# order them by their score in cluster
ord = order(scores_and_labels$scores[genes_in_cluster2,
                    cluster_of_interest])
cluster2Score = genes_in_cluster2[ord]
plot_splines_data(
    moanin_model, subset_data=cluster2Score[1:4],
    centroid=kmeans_clusters$centroids[cluster_of_interest,],
    colors=ann_colors$Group, smooth=TRUE, simpleY=FALSE,
    mar=c(1.5, 2.5, 2, 0.1))
```
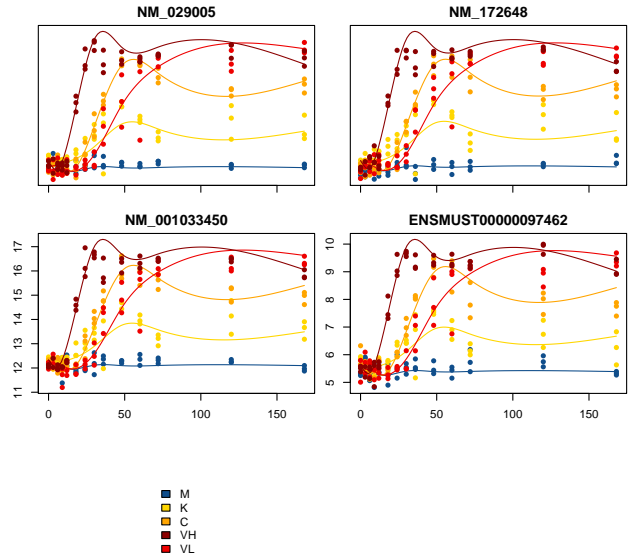
Figure 17: Top scoring genes in Cluster 2

We see that, as expected, these genes are a much better match to the cluster centroids.

Looking at specific clusters in detail.

Now, let us look more in detail at some specific clusters. Cluster 6 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 6, based on our goodness-of-fit assignment.

```
cluster_to_plot = 6
genes_to_plot =  names(labels[labels == cluster_to_plot])
```

Now we will create the heatmaps of these genes (1264 genes, Figure 18).

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

# order the observations by Group, then time, then replicate
orderedMeta<-data.frame(colData(moanin_model))
ord = order(
  orderedMeta$Group,
  orderedMeta$Timepoint,
  orderedMeta$Replicate)
orderedMeta$Timepoint = as.factor(orderedMeta$Timepoint)

orderedMeta = orderedMeta[ord, ]
```

```
res = aheatmap(
    as.matrix(assay(moanin_model[genes_to_plot,ord])),
    Colv=NA,
    color="YlGnBu",
    annCol=orderedMeta[,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(
    rescale_values(moanin_model[genes_to_plot,ord])[res$rowInd,],
    Colv=NA,
    Rowv=NA,
    annCol=orderedMeta[, c("Group", "Timepoint")],
    annLegend=TRUE,
    annColors=ann_colors,
    main=paste("Cluster", cluster_to_plot, "(rescaled)"),
    treeheight=0)
```
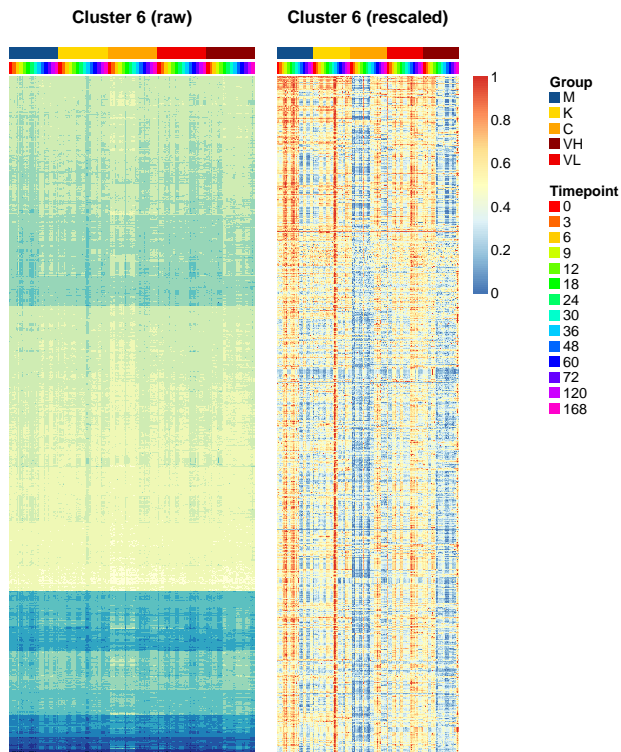

Figure 18: Heatmap of genes in Cluster 6

Those two heatmaps demonstrate that the clustering method successfully clusters 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 (Thorndike 1953), the elbow method looks at the total within 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 clusters based on a number of similarity measures.

First, let us run the clustering for all values of k of interest. We will, for each clustering, conserve (1) with within cluster sum of squares; (2) the clustering assignment (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(de_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 clusters is chosen at this "Elbow point."

We plot the WCSS for $k = 2 - 20$ here (Figure 19). 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")
```
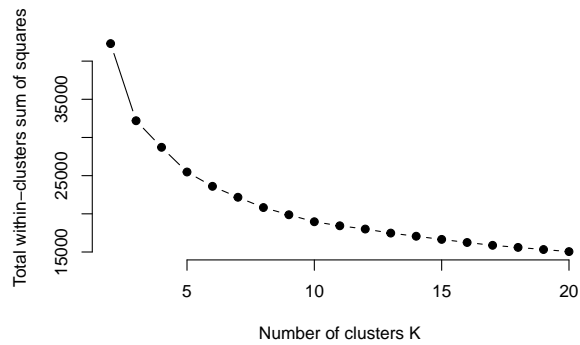
Figure 19: Plot of WCSS as a function of k

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), shown in Figure 20.

```r
# 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,
        assay(de_moanin_model))
    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")
```
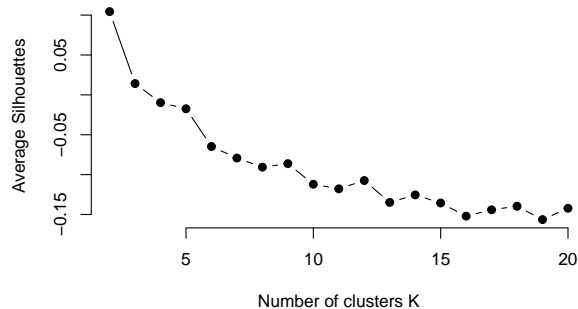
Figure 20: Plot of silhouette values as a function of k

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 perturbations 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 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 all 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. 5388 genes):

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

bootstrapped_y = de_moanin_model[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(de_moanin_model)[1],
        n_genes * subsample_proportion,
        replace=FALSE)

subsampled_y = de_moanin_model[indices, ]
```

We run the 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

28

below, but because of the time the computations take, we have evaluate these values separately and provided the results in separate files (one per k) for users to explore more quickly.

```r
# 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(de_moanin_model)[1] * subsample_proportion
indices = sample(1:dim(de_moanin_model)[1], n_genes, replace=TRUE)

kmeans_clusters = splines_kmeans(de_moanin_model[indices,],
    n_clusters=n_clusters,
    random_seed=42,
    n_init=20)

# Perform prediction on the whole set of data.
kmeans_clusters = splines_kmeans_predict(de_moanin_model, kmeans_clusters,
    method="distance")
```

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

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

Each column corresponds 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 25 resampling runs. For example, for the first 10 bootstrap samples and $k = 5$:

	B1	B2	B3	B4	B5	B6	B7	B8	B9	B10
AA204140	5	5	5	5	5	1	3	2	1	3
AB008911	5	5	1	4	1	2	3	3	2	3
AB010313	3	3	1	4	1	2	4	3	2	5
AB010342	5	3	1	5	1	1	3	3	1	3
AB011473	2	1	2	1	3	3	2	5	5	2
AB099518	2	1	2	1	3	3	2	5	5	2

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 25 resampling runs, and plot the heatmap of those consensus matrices for $k = 5$ and $k = 20$.

```r
par(mfrow=c(1, 2))
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")
```
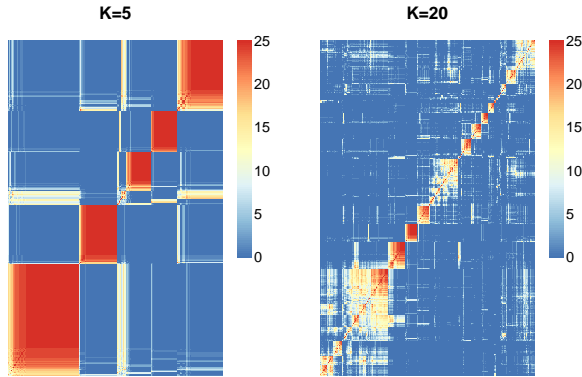
Figure 21: Consensus matrix of proportion of times samples were in the same cluster over bootstrap samples for k=5 and 20.

We can see (Figure 21) 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 (Ben-Hur, Elisseeff, and Guyon 2001) 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 similarities 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)
```
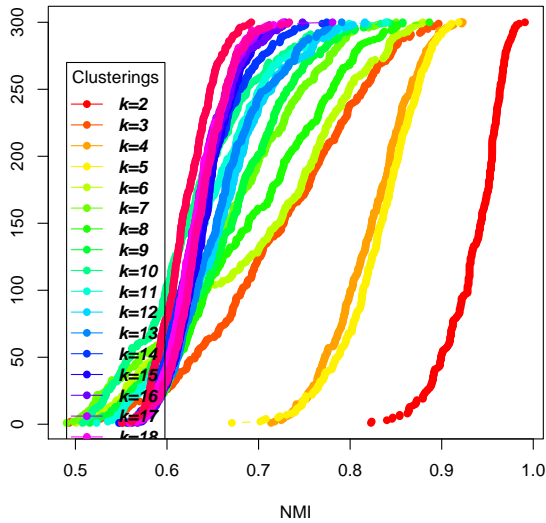
Figure 22: Plot of the cumulative density of similarity of clusterings of data for different k

From this plot (Figure 22), 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 assess 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 (Figure 23).

```
clusters = c("k=6", "k=7", "k=8", "k=9", "k=10")
selected_labels = all_labels[clusters]
plot_model_explorer(selected_labels)
```
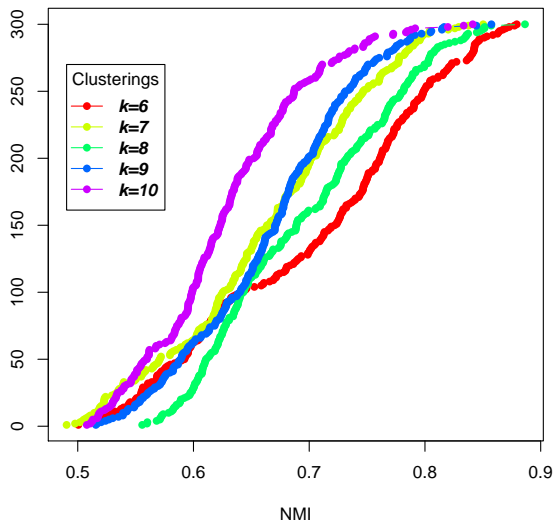
Figure 23: Plot of the cumulative density of similarity of clusterings of data for different k, restricted to k=6-10

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 (Monti et al. 2003) relies on a similar idea but instead of looking at the cumulative density of similarity measures of bootstrapped clustering, the authors suggest 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)
```
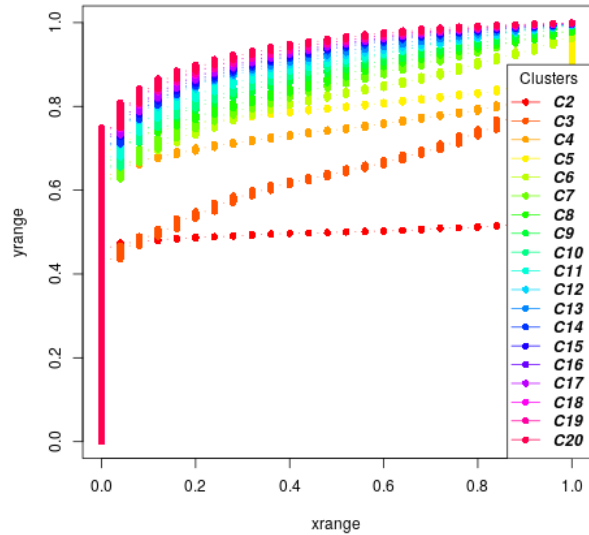
Figure 24: Plot of the Consensus Clustering

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 suggests 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 clusters.

Here we calculate the AUC (Figure 25).

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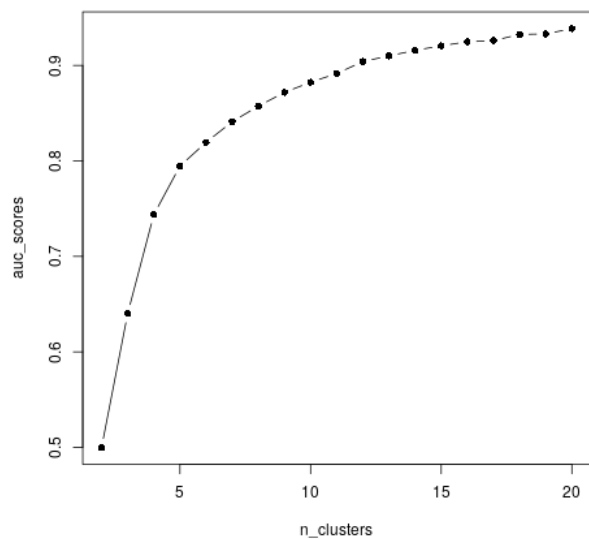

Figure 25: Plot of AUC vs number of clusters

While here we calculate the improvement in the AUC (Figure 26).

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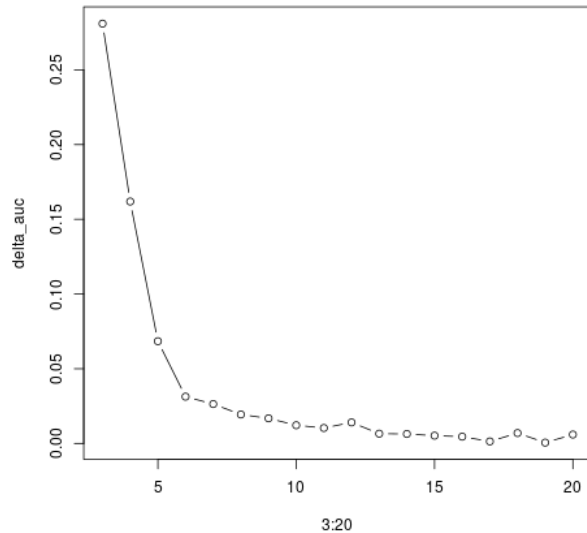

Figure 26: Plot of improvement in AUC vs number of clusters

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 methods 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 can then be performed on the gene set defined by each cluster. Examples include KEGG pathway enrichment analysis, GO term enrichment analysis, and motif enrichment analysis.

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 all genes assigned to the cluster, only the set of differentially expressed genes in each cluster, or the 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` (Durinck et al. 2005) and `KEGGprofile` (Zhao, Guo, and Shyr 2017) for this step. `KEGGprofile` is a package that facilitates enrichment analysis on a set of genes labeled with the Ensembl annotation (Yates et al. 2019) based on the set of biological pathways described in the KEGG database (Kanehisa et al. 2015).

We thus need to convert the gene names, which are given in the Refseq annotation, into the corresponding Ensembl name. This is where `biomaRt` comes in handy: it enables easy conversion from one gene annotation to another. Here, we will use the function `getBM` in `biomaRt` to convert the gene names from cluster 8.

```
## Set up right ensembl database
ensembl = useMart("ensembl")
```

34

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

## Get gene names of genes in Cluster 8
cluster = 6
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"]
```

Then we use the function `find_enriched_pathway` in the `KEGGprofile` package to determine whether any KEGG pathways are enriched in cluster 6, i.e. whether a higher percentage of genes from a single pathway are found in cluster 6 than we would expect by simply proportional assignment of genes to clusters.

```
genes = as.vector(unlist(genes))
pathways = find_enriched_pathway(
    genes, species="mmu",
    download_latest=FALSE)$stastic
```

	Pathway	Percentage	Adj. p-value
04740	Olfactory transduction	8	1.204e-17
04080	Neuroactive ligand-receptor interaction	9	1.104e-05

Finding enriched GO terms

The Gene Ontology (GO) database (Consortium 2018), also categorizes genes into meaningful biological ontologies and can be used for enrichment analysis via the package `topGO` (Alexa and Rahnenfuhrer 2016). We again use `biomaRt` to find the mapping between genes and the GO terms to which they match.

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

The `biomaRt` query results in a matrix with two columns: gene names and GO term ID. The package `topGO` (Alexa and Rahnenfuhrer 2016) 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, for example:

```
$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"
```

`moanin` provides a simple function (`create_go_term_mapping`) to make this conversion:

```
# 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 an interface to `topGO` to determine enriched GO terms. In particular, it performs a p-value correction and only returns the significant GO-term enrichment

in an easy to use `data.frame` object. Here, we show an example of running a GO term enrichment on the "Biological process" ontology (BP).

```
# Create logical vector of whether in cluster 6
assignments = labels == cluster

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

Table 9: Table continues below

	GO ID	Description	Annotated	Significant
2	GO:0019058	viral life cycle	135	135
3	GO:0048872	homeostasis of number of cells	174	173
4	GO:0019693	ribose phosphate metabolic process	213	212
5	GO:0043161	proteasome-mediated ubiquitin-dependent …	202	200
6	GO:0006163	purine nucleotide metabolic process	209	208
7	GO:0006091	generation of precursor metabolites and …	223	221
8	GO:0070925	organelle assembly	452	441
9	GO:0043603	cellular amide metabolic process	486	475
10	GO:0044271	cellular nitrogen compound biosynthetic …	2112	2023
11	GO:0006364	rRNA processing	113	113
12	GO:0070507	regulation of microtubule cytoskeleton o…	110	110
13	GO:0033554	cellular response to stress	952	920
14	GO:0022407	regulation of cell-cell adhesion	257	252
15	GO:0042113	B cell activation	148	147
16	GO:0006996	organelle organization	1902	1845

	Expected	P-value	Adj. p-value
2	127.1	0.00028	0.4093
3	163.8	0.00031	0.4093
4	200.6	0.00037	0.4093
5	190.2	0.00044	0.4093
6	196.8	0.00064	0.4093
7	210	0.00064	0.4093
8	425.6	0.00064	0.4093
9	457.6	0.00072	0.4093
10	1989	0.00104	0.4976
11	106.4	0.00107	0.4976
12	103.6	0.00128	0.5457
13	896.4	0.00155	0.5564
14	242	0.00161	0.5564
15	139.4	0.00171	0.5564
16	1791	0.00174	0.5564

Conclusion

This workflow provides a tutorial for the analysis of lengthy time-course gene expression data in R using the package `moanin`, which aids in implementing common timecourse analyses. We illustrate the workflow through the analysis of mice lung tissue exposed to different influenza strains and measured over time. The proposed workflow consists of three common analysis main steps generally performed after quality control and normalization: (1) differential expression analysis; (2) clustering of time-course gene expression data; (3) downstream analysis of clusters. We demonstrate how the use of the package `moanin` allows for easy implementation of these procedures in the setting of time-course data.

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 https://osf.io/2a5tw/.

All packages used 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 `timecoursedata`. 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 4.0.3 (2020-10-10)
## Platform: x86_64-pc-linux-gnu (64-bit)
## Running under: Ubuntu 20.04.1 LTS
##
## Matrix products: default
## BLAS:   /usr/local/lib/R/lib/libRblas.so
## LAPACK: /usr/local/lib/R/lib/libRlapack.so
##
## 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        BiocWorkflowTools_1.16.0
##  [3] ggfortify_0.4.11          timecoursedata_0.99.18
##  [5] moanin_0.99.17            SummarizedExperiment_1.20.0
##  [7] GenomicRanges_1.42.0      GenomeInfoDb_1.26.0
##  [9] MatrixGenerics_1.2.0      matrixStats_0.57.0
## [11] topGO_2.42.0              SparseM_1.78
## [13] GO.db_3.12.1              AnnotationDbi_1.52.0
## [15] IRanges_2.24.0            S4Vectors_0.28.0
## [17] graph_1.68.0              viridis_0.5.1
## [19] viridisLite_0.3.0         KEGGprofile_1.32.0
## [21] NMF_0.23.0                Biobase_2.50.0
## [23] BiocGenerics_0.36.0       cluster_2.1.0
## [25] rngtools_1.5              pkgmaker_0.32.2
## [27] registry_0.5-1            ggplot2_3.3.2
## [29] pander_0.6.3              biomaRt_2.46.0
```

```
## [31] BiocStyle_2.18.0            knitr_1.30
## [33] limma_3.46.0               devtools_2.3.2
## [35] usethis_1.6.3              rmarkdown_2.5
##
## loaded via a namespace (and not attached):
##  [1] colorspace_1.4-1      ellipsis_0.3.1         rprojroot_1.3-2
##  [4] XVector_0.30.0        fs_1.5.0               rstudioapi_0.11
##  [7] remotes_2.2.0         bit64_4.0.5            fansi_0.4.1
## [10] ClusterR_1.2.2        KEGG.db_3.2.4          xml2_1.3.2
## [13] codetools_0.2-16      doParallel_1.0.16      pkgload_1.1.0
## [16] gridBase_0.4-7        dbplyr_2.0.0           png_0.1-7
## [19] BiocManager_1.30.10   compiler_4.0.3         httr_1.4.2
## [22] backports_1.2.0       Matrix_1.2-18          assertthat_0.2.1
## [25] cli_2.1.0             htmltools_0.5.0        prettyunits_1.1.1
## [28] tools_4.0.3           gmp_0.6-1              gtable_0.3.0
## [31] glue_1.4.2            GenomeInfoDbData_1.2.4 reshape2_1.4.4
## [34] dplyr_1.0.2           rappdirs_0.3.1         Rcpp_1.0.5
## [37] vctrs_0.3.4           Biostrings_2.58.0      NMI_2.0
## [40] iterators_1.0.13      xfun_0.19              stringr_1.4.0
## [43] ps_1.4.0              testthat_3.0.0         lifecycle_0.2.0
## [46] gtools_3.8.2          XML_3.99-0.5           edgeR_3.32.0
## [49] MASS_7.3-53           zlibbioc_1.36.0        scales_1.1.1
## [52] hms_0.5.3             yaml_2.2.1             curl_4.3
## [55] memoise_1.1.0         gridExtra_2.3          TeachingDemos_2.12
## [58] stringi_1.5.3         RSQLite_2.2.1          desc_1.2.0
## [61] foreach_1.5.1         pkgbuild_1.1.0         rlang_0.4.8
## [64] pkgconfig_2.0.3       bitops_1.0-6           evaluate_0.14
## [67] lattice_0.20-41       purrr_0.3.4            bit_4.0.4
## [70] processx_3.4.4        tidyselect_1.1.0       bookdown_0.21
## [73] plyr_1.8.6            magrittr_1.5           R6_2.5.0
## [76] generics_0.1.0        DelayedArray_0.16.0    DBI_1.1.0
## [79] pillar_1.4.6          withr_2.3.0            KEGGREST_1.30.0
## [82] RCurl_1.98-1.2        tibble_3.0.4           crayon_1.3.4
## [85] BiocFileCache_1.14.0  progress_1.2.2         locfit_1.5-9.4
## [88] grid_4.0.3            git2r_0.27.1           blob_1.2.1
## [91] callr_3.5.1           digest_0.6.27          xtable_1.8-4
## [94] tidyr_1.1.2           openssl_1.4.3          munsell_0.5.0
## [97] sessioninfo_1.1.1     askpass_1.1
```

Author contributions

NV and EP wrote the workflow.

Competing interests

The authors declare that they have no competing interests.

Grant information

References

Abrams, Z. B., T. S. Johnson, K. Huang, P. R. O. Payne, and K. Coombes. 2019. "A protocol to evaluate RNA sequencing normalization methods." *BMC Bioinformatics* 20 (Suppl 24): 679.

Alexa, Adrian, and Jorg Rahnenfuhrer. 2016. *TopGO: Enrichment Analysis for Gene Ontology.*

Ben-Hur, Asa, André Elisseeff, and Isabelle Guyon. 2001. "A Stability Based Method for Discovering Structure in Clustered Data." *Pacific Symposium on Biocomputing. Pacific Symposium on Biocomputing*, 6–17.

Consortium, The Gene Ontology. 2018. "The Gene Ontology Resource: 20 years and still GOing strong." *Nucleic Acids Research* 47 (D1): D330–D338. https://doi.org/10.1093/nar/gky1055.

Durinck, S., Y. Moreau, A. Kasprzyk, S. Davis, B. De Moor, A. Brazma, and W. Huber. 2005. "BioMart and Bioconductor: a powerful link between biological databases and microarray data analysis." *Bioinformatics* 21 (16): 3439–40.

Fisher, R.A. 1925. *Statistical Methods for Research Workers.* Edinburgh Oliver & Boyd.

Habib, N., Y. Li, M. Heidenreich, L. Swiech, I. Avraham-Davidi, J. J. Trombetta, C. Hession, F. Zhang, and A. Regev. 2016. "Div-Seq: Single-nucleus RNA-Seq reveals dynamics of rare adult newborn neurons." *Science* 353 (6302): 925–28.

Kanehisa, Minoru, Yoko Sato, Masayuki Kawashima, Miho Furumichi, and Mao Tanabe. 2015. "KEGG as a reference resource for gene and protein annotation." *Nucleic Acids Research* 44 (D1): D457–D462. https://doi.org/10.1093/nar/gkv1070.

Love, Michael I., Wolfgang Huber, and Simon Anders. 2014. "Moderated Estimation of Fold Change and Dispersion for Rna-Seq Data with Deseq2." *Genome Biology* 15 (12): 550. https://doi.org/10.1186/s13059-014-0550-8.

Monti, Stefano, Pablo Tamayo, Jill Mesirov, and Todd Golub. 2003. "Consensus Clustering: A Resampling-Based Method for Class Discovery and Visualization of Gene Expression Microarray Data." *Machine Learning* 52 (1): 91–118. https://doi.org/10.1023/A:1023949509487.

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

Park, T., S. G. Yi, S. H. Kang, S. Lee, Y. S. Lee, and R. Simon. 2003. "Evaluation of normalization methods for microarray data." *BMC Bioinformatics* 4 (September): 33.

Ritchie, Matthiew E, Belinda Phipson, Di Wu, Yifang Hu, Charity W Law, Wei Shi, and Gordon K Smyth. 2015. "limma Powers Differential Expression Analyses for RNA-Sequencing and Microarray Studies." *Nucleic Acids Research* 43 (7): e47.

Robinson, M. D., D. J. McCarthy, and G. K. Smyth. 2010. "edgeR: a Bioconductor package for differential expression analysis of digital gene expression data." *Bioinformatics* 26 (1): 139–40.

Shalek, A. K., R. Satija, J. Shuga, J. J. Trombetta, D. Gennert, D. Lu, P. Chen, et al. 2014. "Single-cell RNA-seq reveals dynamic paracrine control of cellular variation." *Nature* 510 (7505): 363–69.

Shoemaker, S. AND Eisfeld, J. E. AND Fukuyama. 2015. "An Ultrasensitive Mechanism Regulates Influenza Virus-Induced Inflammation." *PLoS Pathogens* 11 (6): 1–25.

Storey, J. D., W. Xiao, J. T. Leek, R. G. Tompkins, and R. W. Davis. 2005. "Significance analysis of time course microarray experiments." *Proc. Natl. Acad. Sci. U.S.A.* 102 (36): 12837–42.

Thorndike, Robert L. 1953. "Who Belongs in the Family?" *Psychometrika* 18 (4): 267–76. https://doi.org/10.1007/BF02289263.

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

Varoquaux, N., B. Cole, C. Gao, G. Pierroz, C. R. Baker, D. Patel, M. Madera, et al. 2019. "Transcriptomic Analysis of Field-Droughted Sorghum from Seedling to Maturity Reveals Biotic and Metabolic Responses." *Proceedings of the National Academy of Sciences.* https://doi.org/10.1073/pnas.1907500116.

Wenguang, Sun, and Wei Zhi. 2011. "Multiple Testing for Pattern Identification, with Applications to Microarray Time-Course Experiments." *Journal of the American Statistical Association* 106 (493): 73–88. https://doi.org/10.1198/jasa.2011.ap09587.

Wu, Shuang, and Hulin Wu. 2013. "More Powerful Significant Testing for Time Course Gene Expression Data Using Functional Principal Component Analysis Approaches." *BMC Bioinformatics* 14 (1): 6. https://doi.org/10.1186/1471-2105-14-6.

Yates, Andrew D, Premanand Achuthan, Wasiu Akanni, James Allen, Jamie Allen, Jorge Alvarez-Jarreta, M Ridwan Amode, et al. 2019. "Ensembl 2020." *Nucleic Acids Research* 48 (D1): D682–D688. https://doi.org/10.1093/nar/gkz966.

Zhao, Shilin, Yan Guo, and Yu Shyr. 2017. *KEGGprofile: An Annotation and Visualization Package for Multi-Types and Multi-Groups Expression Data in Kegg Pathway.*