Trendy: Segmented regression analysis of expression dynamics for high-throughput ordered profiling experiments

Rhonda Bacher, Ning Leng, Ron Stewart

April 30, 2018

Contents

1 Overview

Trendy is an R package fo analyzing high throughput expression data (e.g RNA-seq or microarray) with ordered conditions (e.g. time-course, spatial-course).

For each gene (or other features), Trendy fits a set of segmented (or breakpoint) regression models. The optimal model is chosen as the one with the lowest BIC. Each breakpoint represents a significant changes in the gene's expression across the time-course.

The top dynamic genes are then identified as those that are well profiled by their gene-specific segmented regression model. Trendy also implements functions to visualize the dynamic genes and their trends, to order dynamic genes by their trends, and to compute breakpoint distribution at different time-points (e.g. detect time-points with a large number of expression changes).

1.1 The model

To illustrate Trendy, here we use time-course gene expression data as an example. Although, Trendy may also be applied to other types of features (e.g. isoform or exon expression) and/or other types of experiments with ordered conditions (e.g. spatial course).

Denote the normalized gene expression of gene g and sample/time t as $Y_{g,t}$. Denote the total number of genes as G and the total number of samples/times as N. For each gene, Trendy fits segmented regression models with varying numbers of breakpoints from 1 to K. K defaults to 3 but can also be specified by the user. The *segmented* R package is used to fit the segmented regression models.

For a given gene, among the models with varying k, Trendy selects the optimal number of breakpoints for this gene by comparing the BIC for each model.

To avoid overfitting, the optimal number of breakpoints will be set as $\tilde{k_g} = \tilde{k_g} - 1$ if at least one segment has less than c_{num} samples. The threshold c_{num} can be specified by the user; the default is 5.

Trendy reports the following for the optimal model:

- Gene specific adjusted R^2 (penalized for the chosen value of k)
- Segment slopes
- Segment trends (and associated p-values)
- Breakpoint estimates

Among all genes, the top dynamic genes are defined as those whose optimal model has high adjusted R^2s.

To compute the breakpoint distribution over the time-course, Trendy calculates the number of breakpoints for each time-point across all the genes.

The time-points with high D_t can be considered as those with global expression changes.

Trendy also summarizes the fitted trend or expression pattern of top genes. For samples between the i^{th} and $i + 1^{th}$ breakpoint for a given gene, if the t-statistic of the segment slope has p-value greater than c_{pval}, the trend of this segment will be defined as no change. Otherwise the trend of this segment will be defined as up/down based on the slope coefficient. The default value of c_{pval} is 0.1, but may also be specified by the user.

2 Installation

2.1 Install via GitHub

The Trendy package can be installed using functions in the devtools package.

To install, type the following code into R:

```
# install.packages("devtools")
# library(devtools)
# install_github("rhondabacher/Trendy")
```

2.2 Install locally

Install packages segmented, parallel, and gplots:

```
# install.packages(c("segmented","gplots"))
library("segmented")
library("parallel")
library("gplots")
```

Download the Trendy package from: https://github.com/rhonda/Trendy

And install the package locally.

2.3 Load the package

To load the Trendy package:

```
library(Trendy)
```

3 Analysis

3.1 Input

The input data should be a $G-by-N$ matrix containing the expression values for each gene and each sample, where G is the number of genes and N is the number of samples. The samples should be sorted following the time course order.

The object `trendyExampleData` is a simulated data matrix containing 50 rows of genes and 40 columns of samples.

```
data("trendyExampleData")
str(trendyExampleData)
```

```
##  num [1:50, 1:40] 240 199 198 239 202 ...
##  - attr(*, "dimnames")=List of 2
```

```
##   ..$ : chr [1:50] "g1" "g2" "g3" "g4" ...
##   ..$ : chr [1:40] "s1" "s2" "s3" "s4" ...
```

These values should exhibit expression data after normalization across samples. For example, for RNA-seq data, the raw counts may be normalized using MedianNorm and `GetNormalizedMat` function in *EBSeq*:

```
library(EBSeq)
Sizes <- MedianNorm(trendyExampleData)
normalizedData <- GetNormalizedMat(trendyExampleData, Sizes)
```

More details can be found in the *EBSeq* vignette:

http://www.bioconductor.org/packages/devel/bioc/vignettes/EBSeq/inst/doc/EBSeq_Vignette.pdf

If you are working with microarray expression data, an extensive overview for normalization can be found in the vignette of the *affy* package:

https://www.bioconductor.org/packages/release/bioc/html/affy.html

3.2 Run Trendy

The `trendy` function will fit multiple segmented regressions model for each gene (via the *segmented* R package) and select the the optimal model. Here we want to only consider a maximum of two breakpoints for each gene.

```
res <- trendy(Data = trendyExampleData, maxK = 2)
```

```
## Warning in trendy(Data = trendyExampleData, maxK = 2):  No values for parameter tVectIn
were given.
##        Trendy will assume data goes from 1:40
```

```
res <- results(res)
res.top <- topTrendy(res)
# default adjusted R square cutoff is 0.5
res.top$AdjustedR2
```

```
##        g3        g1       g28       g20       g15        g4        g2
## 0.9787382 0.9775005 0.9751380 0.9739715 0.9729747 0.9711890 0.9710139
##       g10       g23       g14        g8        g5       g24       g17
## 0.9705118 0.9701402 0.9694164 0.9691341 0.9689555 0.9656732 0.9652141
##        g9       g12       g29       g16       g22       g18        g6
## 0.9649424 0.9644343 0.9632348 0.9630272 0.9627092 0.9626837 0.9619387
##       g25       g11       g30       g27       g26       g19        g7
## 0.9611528 0.9600736 0.9597989 0.9592516 0.9572072 0.9536619 0.9529077
##       g21       g13
## 0.9528854 0.9470928
```

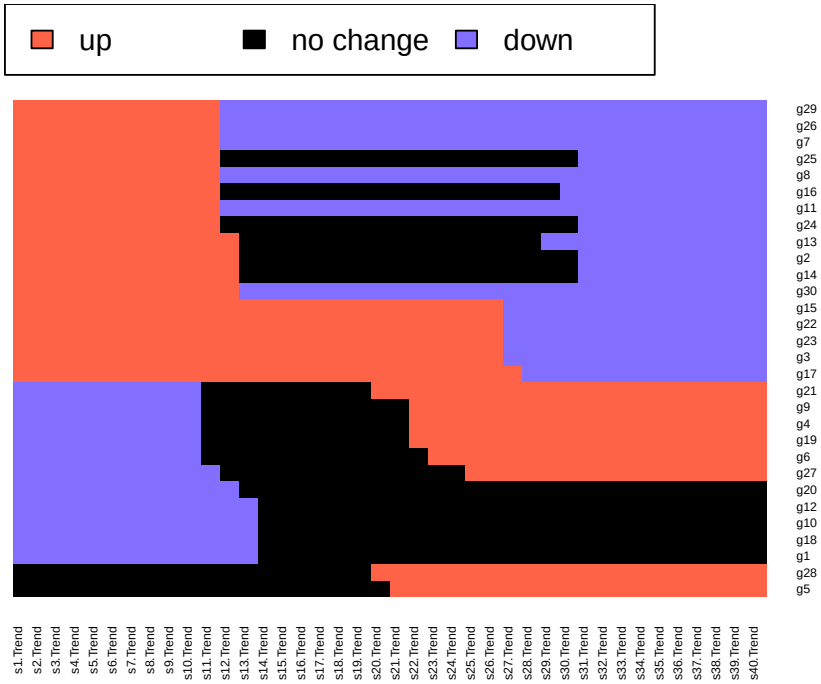
The `topTrendy` function may be used to extract top dynamic genes. By default, `topTrendy` will extract genes whose adjusted R^2, $\bar{R}^2$, is greater or equal to 0.5. To change this threshold, a user may specify the `AdjR2.Cut` parameter in the `topTrendy` function. The `topTrendy` function returns the Trendy output with genes sorted decereasingly by $\bar{R}^2$.

By default the `trendy` function only considers genes whose mean expression is greater than 10. To use another threshold, the user may specify the parameter `meanCut`.

3.3 Visualize trends of the top dynamic genes

`res.top$Trend` contains the trend specification of the top genes. The function `trendHeatmap` can be used to display these trends:

```
res.trend <- trendHeatmap(res.top)
```



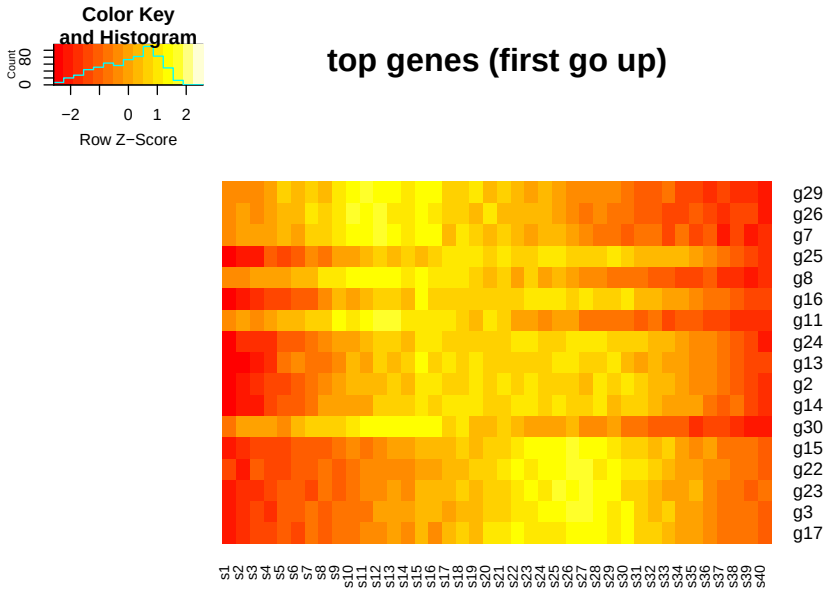

```
str(res.trend)
```

```
## List of 3
##  $ firstup      : Named num [1:17] 11.4 11.5 11.6 11.6 11.6 ...
##   ..- attr(*, "names")= chr [1:17] "g29" "g26" "g7" "g25" ...
##  $ firstdown    : Named num [1:11] 10.7 10.9 10.9 10.9 11 ...
##   ..- attr(*, "names")= chr [1:11] "g21" "g9" "g4" "g19" ...
##  $ firstnochange: Named num [1:2] 19 20.4
##   ..- attr(*, "names")= chr [1:2] "g28" "g5"
```

The `trendHeatmap` function classifies the top dynamic genes into three groups: start with up, start with down and start with no change. Within each group, genes are sorted by the position of the first breakpoint.

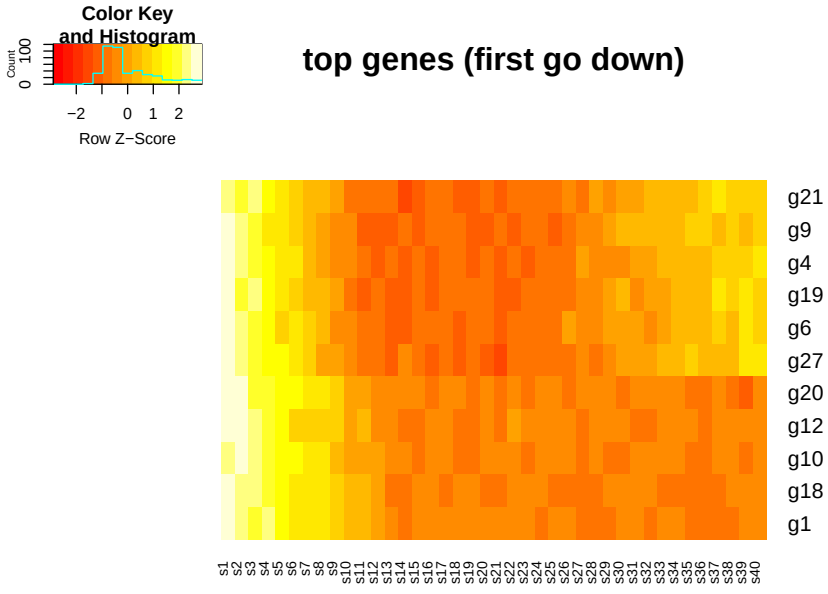
To generate an expression heatmap of the first group of genes (first go up):

```
library(gplots)
heatmap.2(trendyExampleData[names(res.trend$firstup),],
  trace="none", Rowv=FALSE,Colv=FALSE,
        scale="row", main="top genes (first go up)")
```



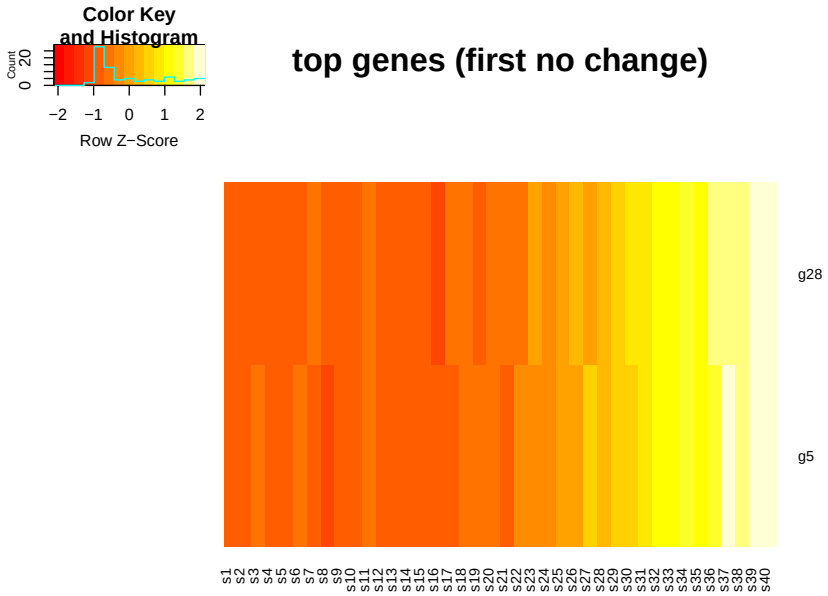
Similarly, to generate an expression heatmap of the second group of genes (first go down):

```
heatmap.2(trendyExampleData[names(res.trend$firstdown),],
  trace="none", Rowv=FALSE,Colv=FALSE,
        scale="row", main="top genes (first go down)")
```



To generate an expression heatmap of the second group of genes (first no change):

```
heatmap.2(trendyExampleData[names(res.trend$firstnochange),],
  trace="none", Rowv=FALSE,Colv=FALSE,
        scale="row", main="top genes (first no change)",
        cexRow=.8)
```

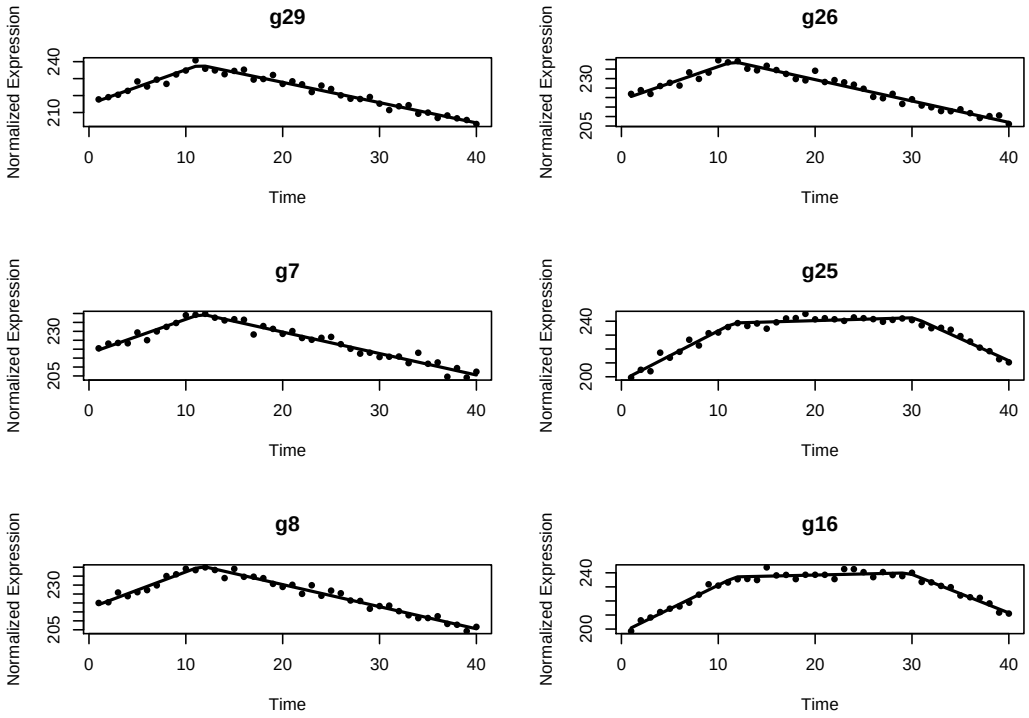


3.4 Visualize individual genes

The `plotFeature` function may be used to plot expression of individual features/genes and the fitted lines.

For example, to plot the top six genes in the first group of genes (first go up):

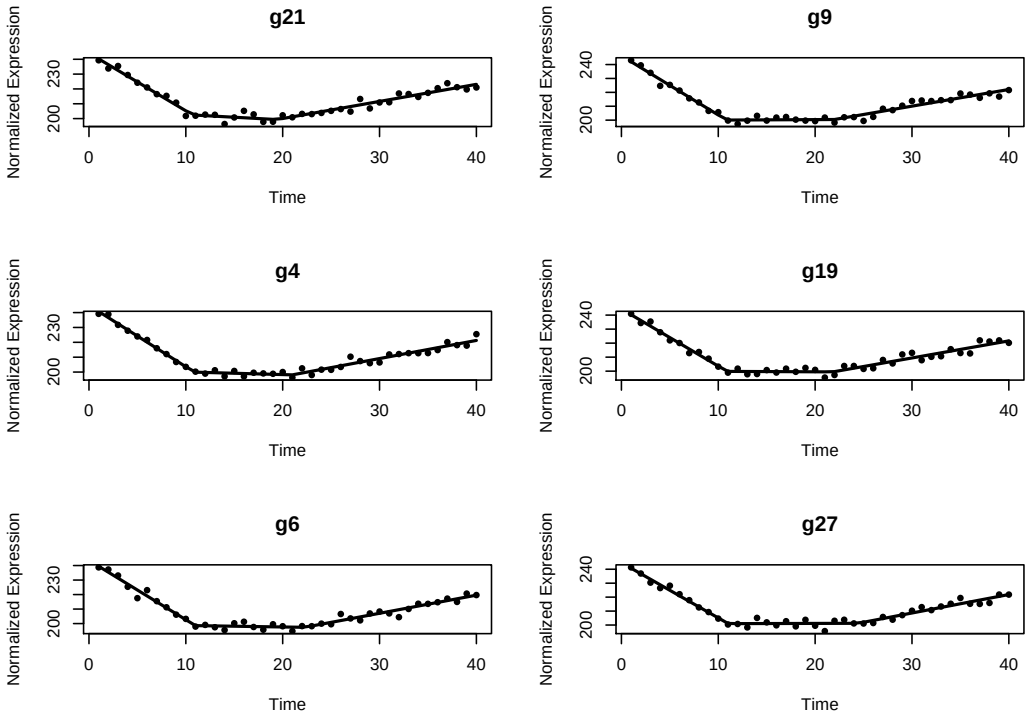
```
par(mfrow=c(3,2))
plot1 <- plotFeature(trendyExampleData,
                     featureNames = names(res.trend$firstup)[1:6],
                     trendyOut = res)
```



The input of function `plotFeature` requires the expression data and a list of genes of interest. The parameter `trendyOut` are the results from the `trendy` function. If it is not specified, then `plotFeature` will run `trendy` on the genes of interest before plotting. Specifying the output obtained from previous steps will save time by avoiding fitting the models again.

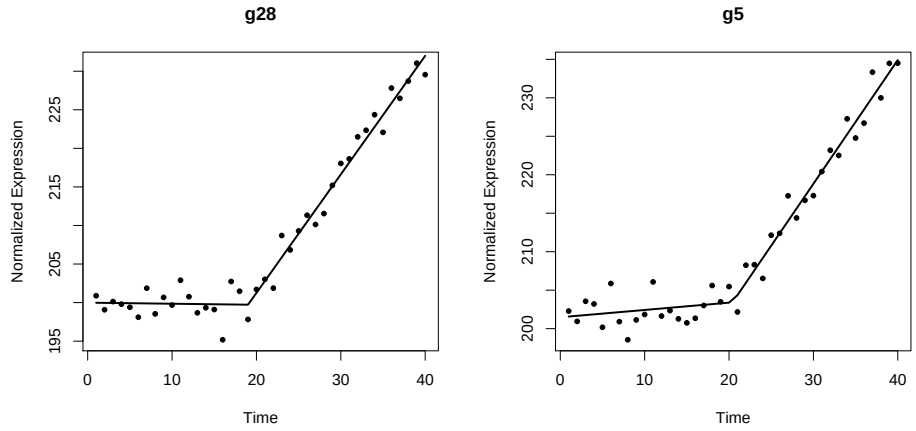
Similarly, to plot the top six genes in the second group of genes (first go down):

```
par(mfrow=c(3,2))
plot2 <- plotFeature(Data = trendyExampleData,
                     featureNames = names(res.trend$firstdown)[1:6],
                     trendyOut = res)
```



To plot the two genes in the third group of genes (first no change):

```
par(mfrow=c(1,2))
plot2 <- plotFeature(trendyExampleData,
                     featureNames = names(res.trend$firstnochange)[1:2],
                     trendyOut = res)
```



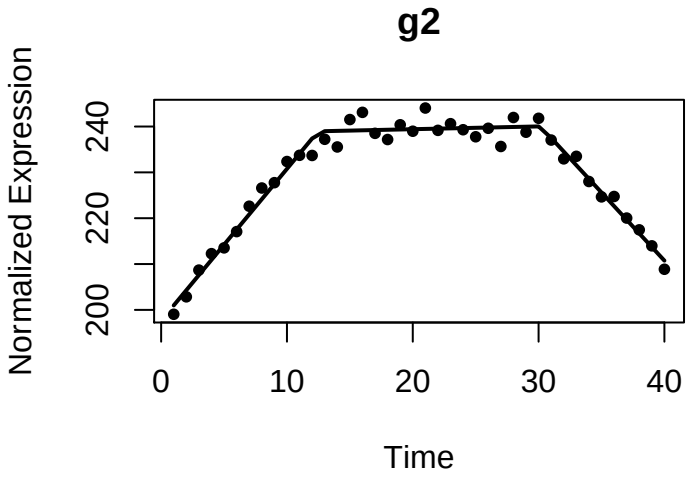
3.5 Gene specific estimates

For a given gene of interest, its estimated parameters can be obtained by (using g2 as an example):

```
par(mfrow=c(1,1))
plot2 <- plotFeature(trendyExampleData,
```

```
                    featureNames = "g2",
                    trendyOut = res)
```

```
## Warning in plotFeature(trendyExampleData, featureNames = "g2", trendyOut = res):  No values
for parameter tVectIn was given.
##        Trendy will assume data goes from 1:40
```




```
res.top$Breakpoints["g2",] # break points
```

```
## Breakpoint1 Breakpoint2
##    12.47356    30.14908
```

```
res.top$AdjustedR2["g2"] # adjusted r squared
```

```
##        g2
## 0.9710139
```

```
res.top$Segments["g2",] # fitted slopes of the segments
```

```
## NULL
```

```
res.top$Segment.Pvalues["g2",] # p value of each the segment
```

```
## Segment1.Pvalue Segment2.Pvalue Segment3.Pvalue
##      0.01669823      0.31816176      0.02445599
```

The above printouts show that for gene g2 the optimal number of breakpoints is 2. Two estimated breakpoints are around time-points s12 and s30. The fitted slopes for the 3 segments are 3.31, 0.06 and -2.97, which indicate the trend is up-same-down.

These estimates can also be automatically formatted using the function `formatResults` which can be saved as a .txt. or .csv file. The output currently includes the estimated slope, p-value, and trend of each segment, the estimated breakpoints, the trend for each sample, and the adjusted R^2.

```
trendy.summary <- formatResults(res.top)
trendy.summary[1:4,1:4]
```

```
##     Feature Segment1.Slope Segment2.Slope Segment3.Slope
## g3       g3       1.572400     -2.5483000             NA
## g1       g1      -3.145400      0.0015484             NA
## g28     g28      -0.013815      1.5369000             NA
## g20     g20      -3.381400     -0.0824630             NA
```

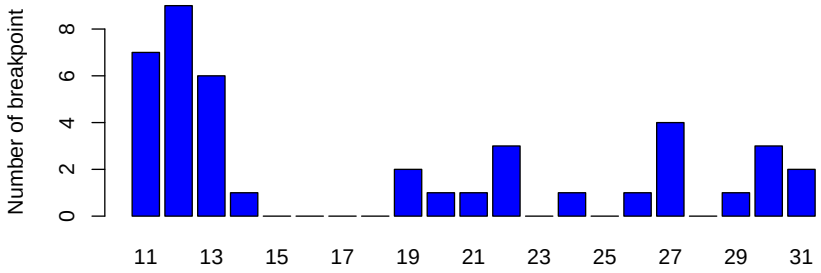
```
# write.table(trendy.summary, file="trendy_summary.txt")
```

The NA indicates that g3 does not have a segment 3 slope since it only has one breakpoint (i.e two segments).

3.6 Breakpoint distribution over the time course

To calculate number of breakpoints over the time course:

```
res.bp <- breakpointDist(res.top)
barplot(res.bp, ylab="Number of breakpoint", col="blue")
```



The bar plot indicates that a number of genes have breakpoints around s12 and s13.

4 More advanced analysis

4.1 Time course with non-uniform sampling

If the samples were collected with different time intervals then it is highly suggested to use the original time (instead of a vector of consecutive numbers). To do so, the user may specify the order/times via the tVectIn parameter in the `trendy` function.

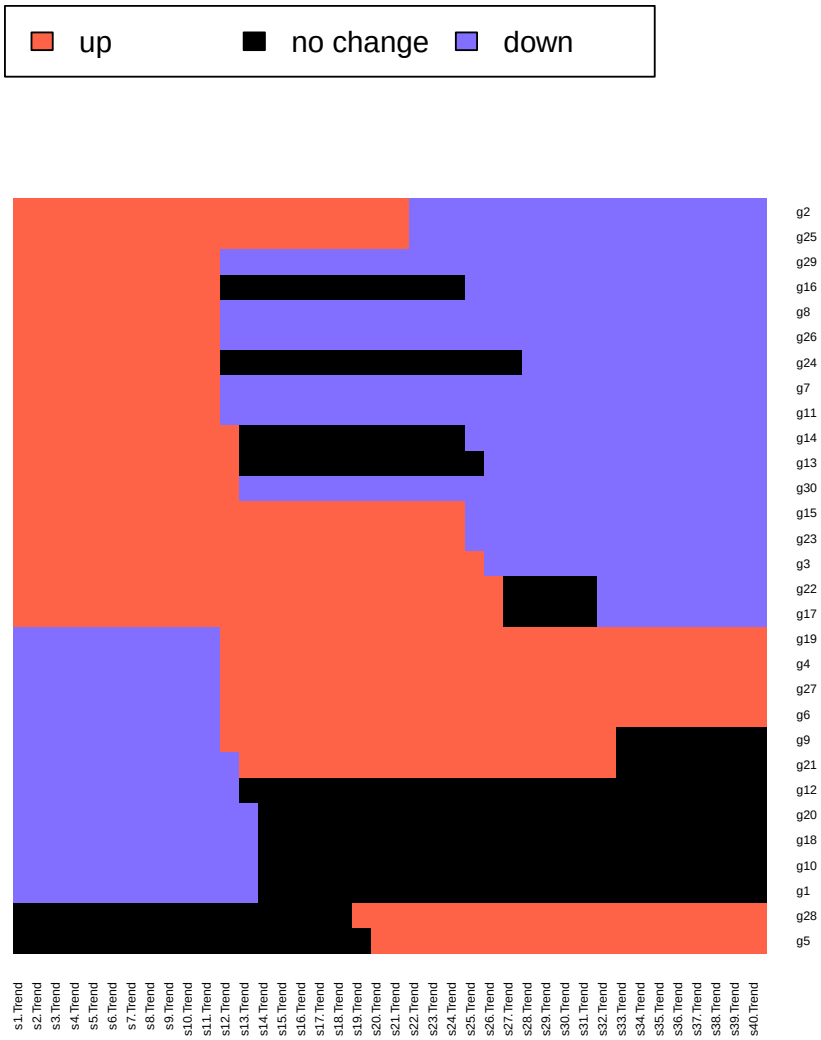
For example, suppose for the example data, the first 30 samples were collected every hour and the other 10 samples were collected every 5 hours. We may define the time vector as:

```
t.v <- c(1:30, seq(31, 80, 5))
names(t.v) <- colnames(trendyExampleData)
t.v
```

```
##  s1  s2  s3  s4  s5  s6  s7  s8  s9 s10 s11 s12 s13 s14 s15 s16 s17 s18 s19
##   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17  18  19
## s20 s21 s22 s23 s24 s25 s26 s27 s28 s29 s30 s31 s32 s33 s34 s35 s36 s37 s38
##  20  21  22  23  24  25  26  27  28  29  30  31  36  41  46  51  56  61  66
## s39 s40
##  71  76
```

To run Trendy model using the empirical collecting time instead of sample ID (1-40):

```
res2 <- trendy(Data = trendyExampleData, tVectIn = t.v, maxK=2)
res2 <- results(res2)
res.top2 <- topTrendy(res2)
res.trend2 <- trendHeatmap(res.top2)
```

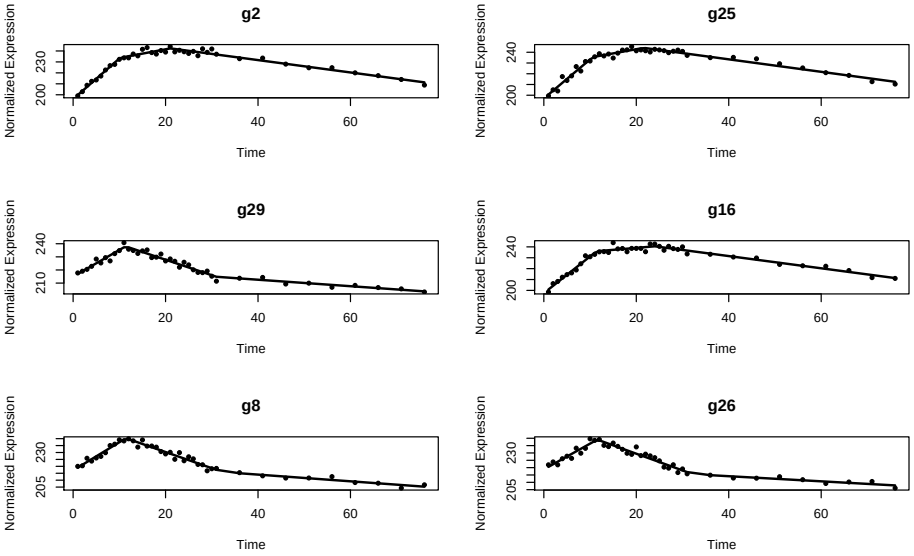



```
str(res.trend2)
```

```
## List of 3
##  $ firstup      : Named num [1:17] 10.2 10.6 11.4 11.4 11.4 ...
##   ..- attr(*, "names")= chr [1:17] "g2" "g25" "g29" "g16" ...
##  $ firstdown    : Named num [1:11] 11.2 11.4 11.4 11.5 11.7 ...
##   ..- attr(*, "names")= chr [1:11] "g19" "g4" "g27" "g6" ...
##  $ firstnochange: Named num [1:2] 19 19.5
##   ..- attr(*, "names")= chr [1:2] "g28" "g5"
```

To plot the first six genes that have up-regulated pattern at the beginning of the time course:

```
par(mfrow=c(3,2))
plot1.new <- plotFeature(trendyExampleData, tVectIn=t.v,
                         featureNames = names(res.trend2$firstup)[1:6],
                         trendyOut = res2)
```



4.2 Time-course with replicates available

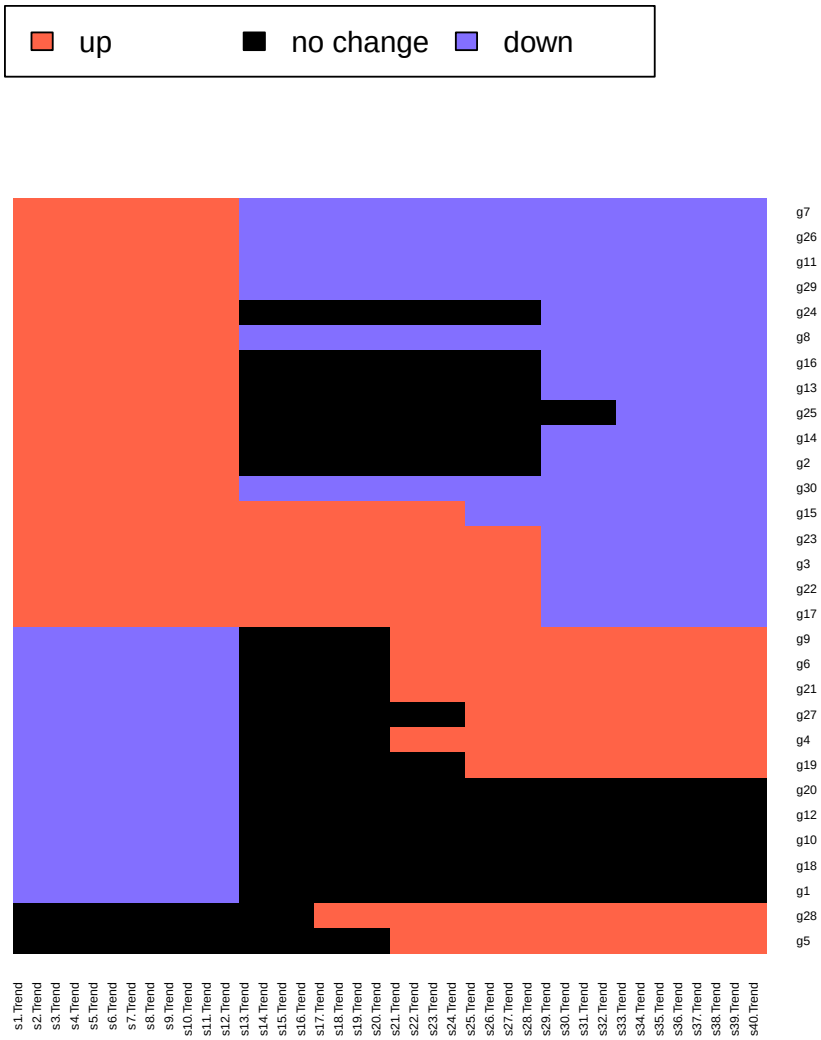
Trendy is able to make use of replicated time-points if available. To do so, the user can specify the replicated directly into the the tVectIn parameter in the `trendy` function.

For example, suppose for the example data, 10 time points were observed 4 times each. We may define the time vector as:

```
t.v <- rep(1:10, each=4)
names(t.v) <- colnames(trendyExampleData)
t.v
```

```
##  s1  s2  s3  s4  s5  s6  s7  s8  s9 s10 s11 s12 s13 s14 s15 s16 s17 s18 s19
##   1   1   1   1   2   2   2   2   3   3   3   3   4   4   4   4   5   5   5
## s20 s21 s22 s23 s24 s25 s26 s27 s28 s29 s30 s31 s32 s33 s34 s35 s36 s37 s38
##   5   6   6   6   6   7   7   7   7   8   8   8   8   9   9   9   9  10  10
## s39 s40
##  10  10
```

```
res2 <- trendy(Data = trendyExampleData, tVectIn = t.v, maxK=2)
res2 <- results(res2)
res.top2 <- topTrendy(res2)
res.trend2 <- trendHeatmap(res.top2)
```




```
str(res.trend2)
```

```
## List of 3
##  $ firstup      : Named num [1:17] 3.26 3.26 3.27 3.28 3.31 ...
##   ..- attr(*, "names")= chr [1:17] "g7" "g26" "g11" "g29" ...
##  $ firstdown    : Named num [1:11] 3.04 3.11 3.14 3.18 3.22 ...
##   ..- attr(*, "names")= chr [1:11] "g9" "g6" "g21" "g27" ...
##  $ firstnochange: Named num [1:2] 4.76 5.5
##   ..- attr(*, "names")= chr [1:2] "g28" "g5"
```

5 Extract genes with certain pattern

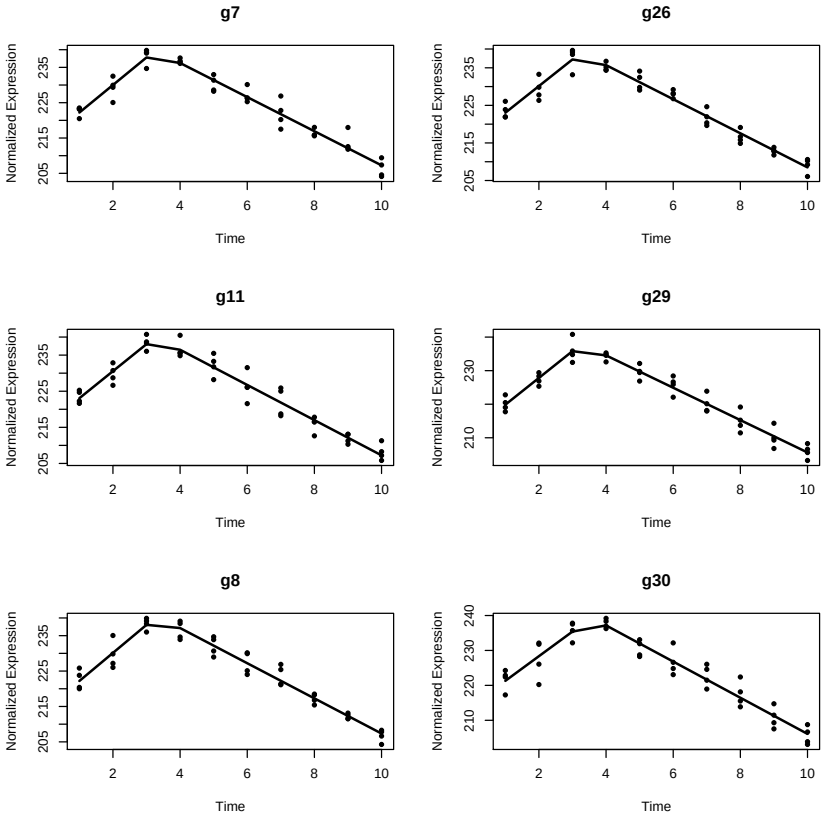
Genes that have a peak along the time-course will have fitted trend somewhere as "up-down". Genes that are oscillating may have the fitted trend "up-down". To extract a list of such genes we can use the `extractPattern`:

```
# Genes that peak
pat1 <- extractPattern(res2, Pattern = c("up","down"))
```

```
head(pat1)
```

```
##    Gene BreakPoint1
## 2    g7    3.258496
## 1   g26    3.260000
## 8   g11    3.267782
## 5   g29    3.278039
## 3    g8    3.315098
## 11  g30    3.566570
```

```
par(mfrow=c(3,2))
plotPat1 <- plotFeature(trendyExampleData, tVectIn=t.v,
                        featureNames = pat1$Gene[1:6],
                        trendyOut = res2)
```

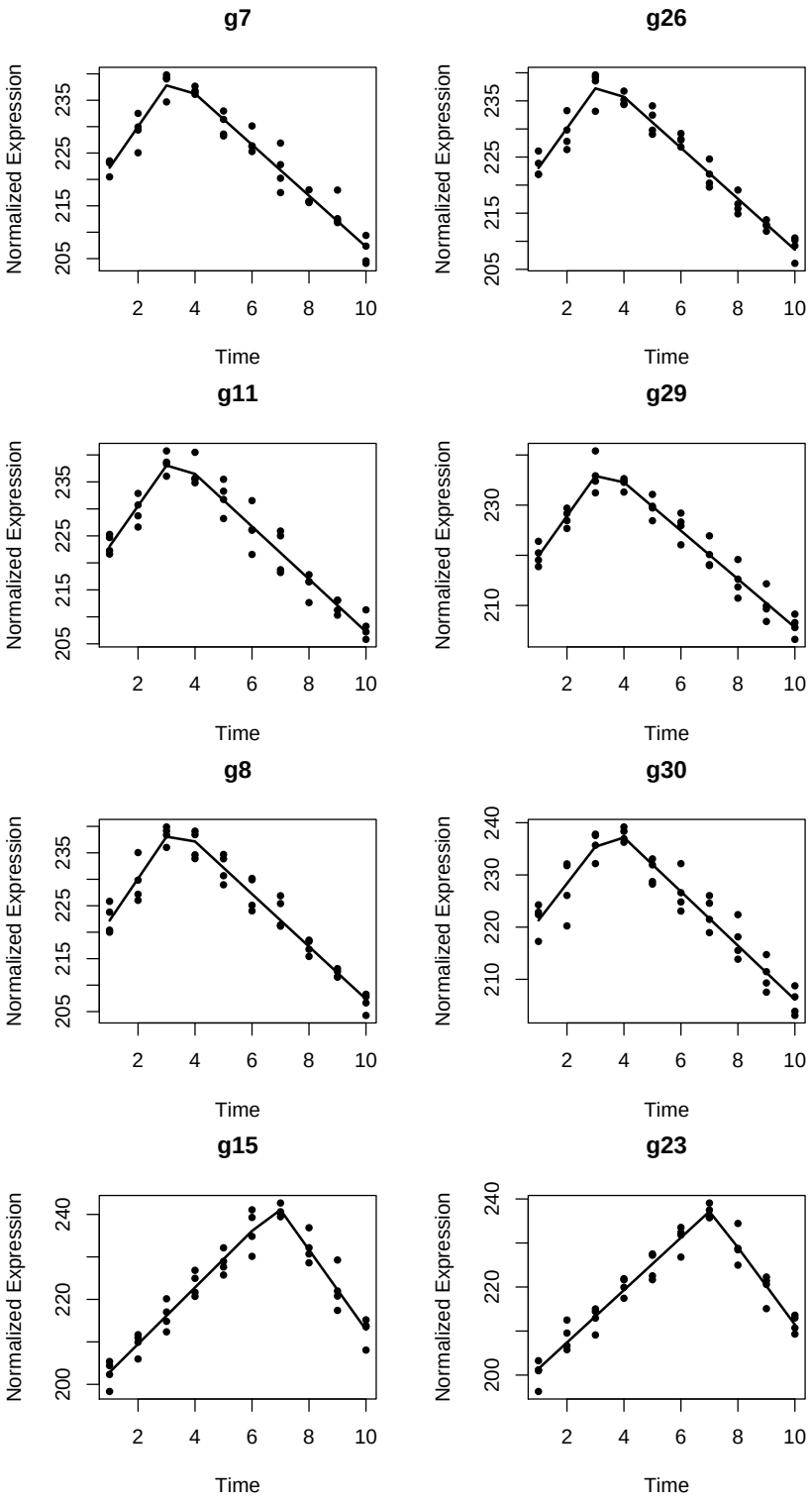


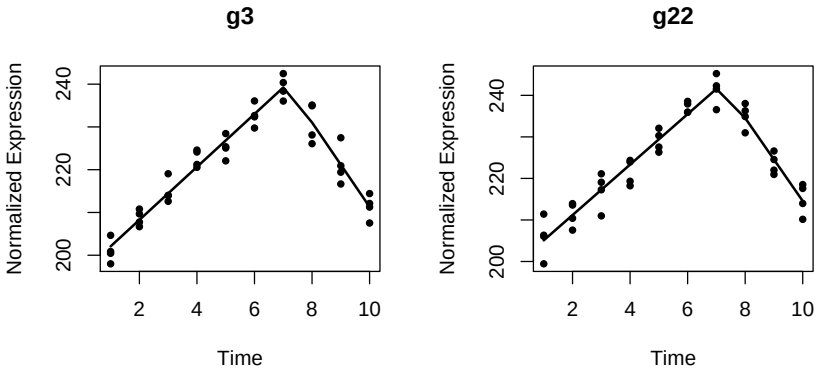

```
# Genes that peak after some time
pat3 <- extractPattern(res2, Pattern = c("up","down"), Delay = 25)
head(pat3)
```

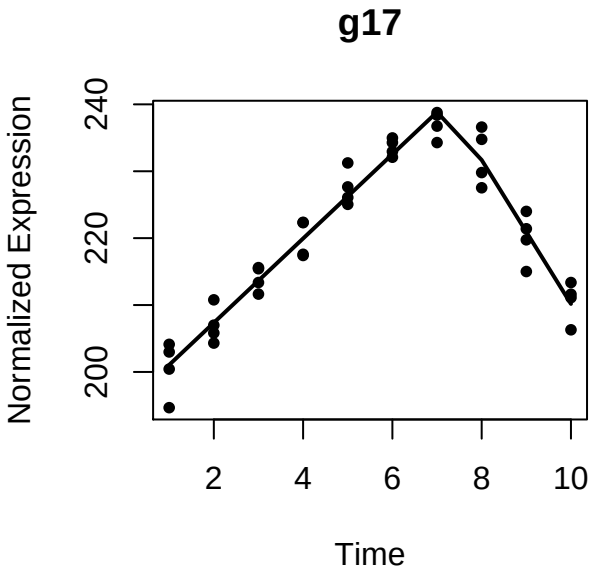
```
##    Gene BreakPoint1
## 2    g7    3.258496
## 1   g26    3.260000
## 8   g11    3.267782
## 5   g29    3.278039
## 3    g8    3.315098
## 11  g30    3.566570
```

```
par(mfrow=c(1,2))
plotPat3 <- plotFeature(trendyExampleData, tVectIn=t.v,
```

```
                    featureNames = pat3$Gene,
                    trendyOut = res2)
```





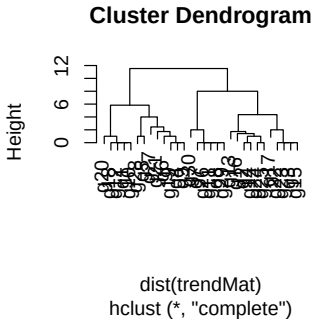



```
# Genes that are constant, none
pat4 <- extractPattern(res2, Pattern = c("same"))
```

6 Further analysis of Trendy expression trends

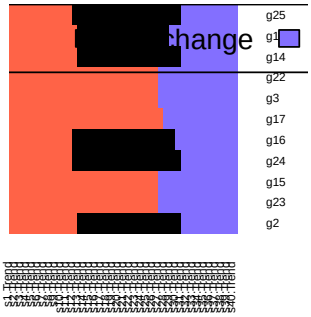
For each gene, the Trendy segments are assigned a direction as: "up", "down", or "same". These directions can be used to cluster genes having similar trends along the time-course. Here I will use a simple hierarchical clustering to demonstrate the clustering but other clustering methods may be applicable.

```
# Get trend matrix:
trendMat <- res.top$Trends
# Cluster genes using hierarchical clustering:
hc.results <- hclust(dist(trendMat))
plot(hc.results) #decide how many cluster to choose
```

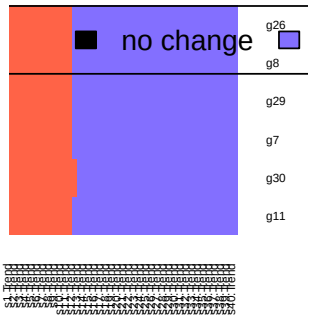



```
#Let's say there are 4 main clusters
hc.groups <- cutree(hc.results, k = 4)

cluster1.genes <- names(which(hc.groups == 1))
res.trend2 <- trendHeatmap(res.top, featureNames = cluster1.genes)
```




```
cluster4.genes <- names(which(hc.groups == 4))
res.trend2 <- trendHeatmap(res.top, featureNames = cluster4.genes)
```



The genes in each cluster can then be input to a gene enrichment analysis, using tools such as enrichr, allez, or GSEA.

7 Additional options

In the `trendy` function, the thresholds c_{num} and c_{pval} can be specified via parameters minNumInSeg and pvalCut, respectively.

8 Trendy shiny app

The Trendy shiny app requires an .RData object output from the `trendy` function, which can be obtained by setting `saveObject=TRUE`.

```
res <- trendy(trendyExampleData, maxK=2, saveObject = TRUE, fileName="exampleObject")
res <- results(res)
```

Then in R run:

```
trendyShiny()
```

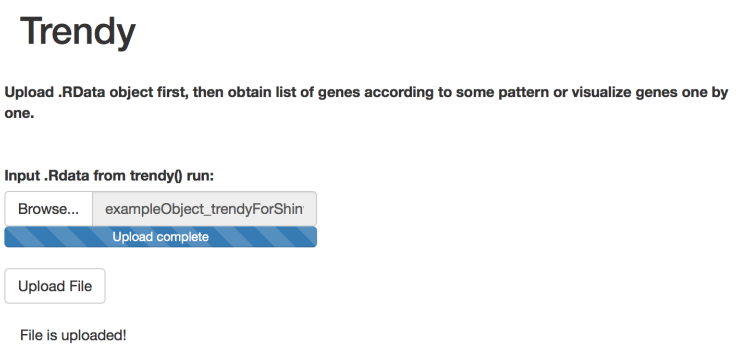


Figure 1: Upload shiny object

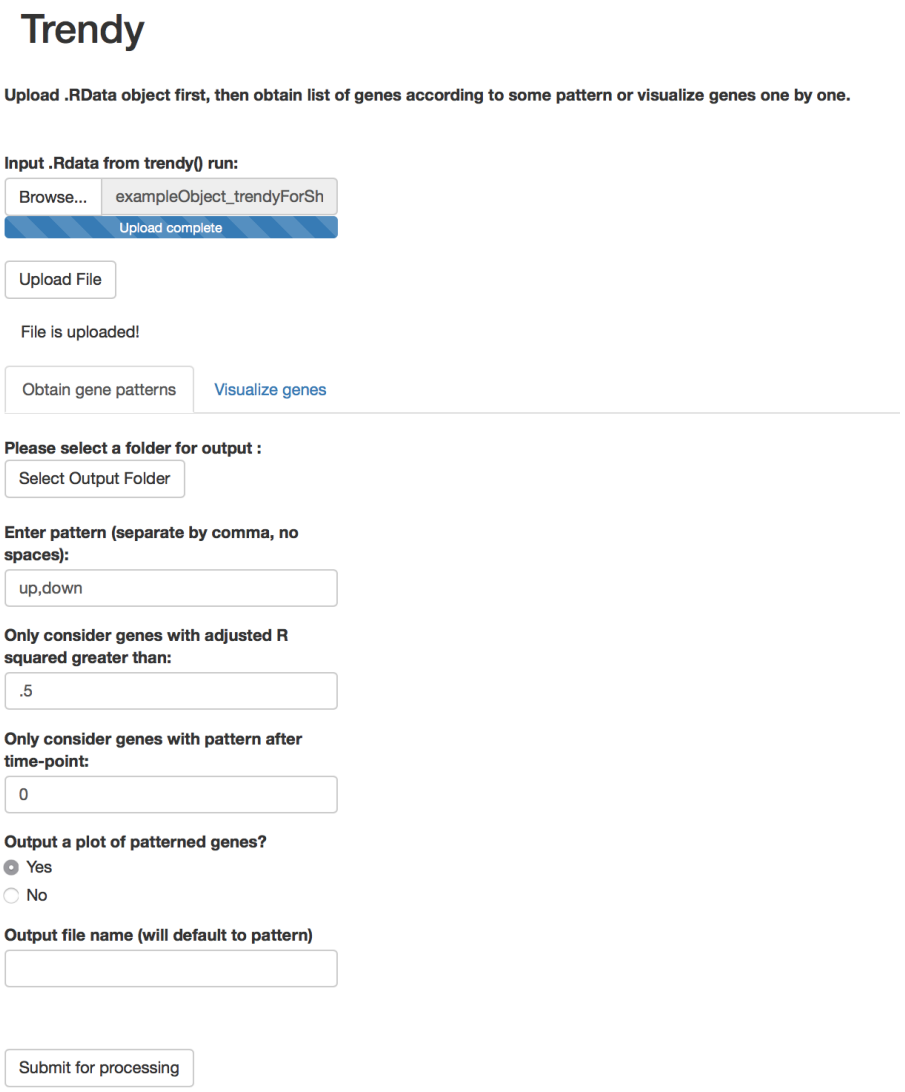


Figure 2: Find all genes with a given pattern

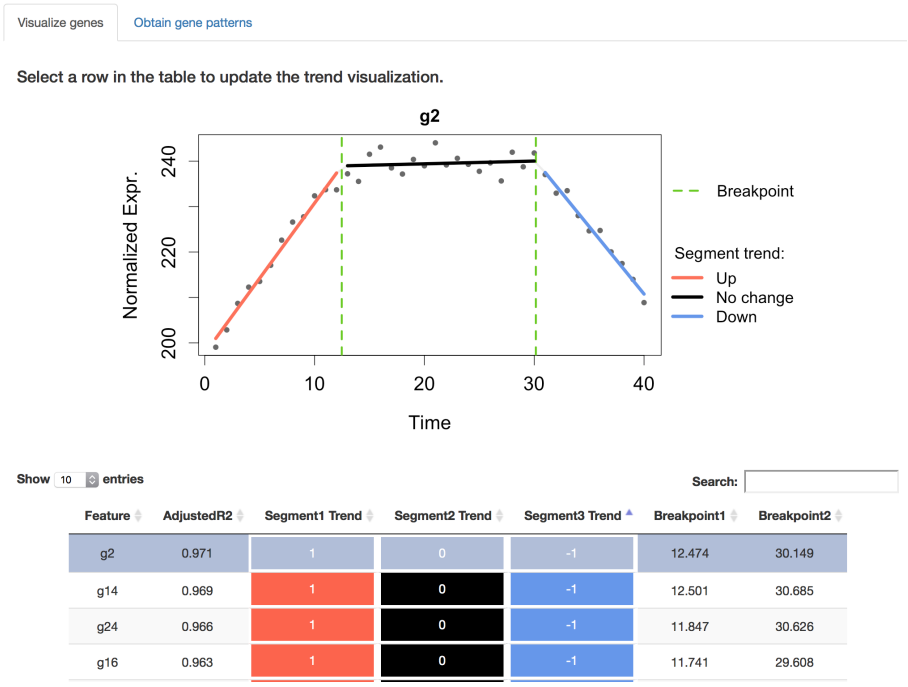


Figure 3: Search genes individually

9 SessionInfo

```
sessionInfo()
```

```
## R version 3.5.0 (2018-04-23)
## Platform: x86_64-pc-linux-gnu (64-bit)
## Running under: Ubuntu 16.04.4 LTS
##
## Matrix products: default
## BLAS: /home/biocbuild/bbs-3.7-bioc/R/lib/libRblas.so
## LAPACK: /home/biocbuild/bbs-3.7-bioc/R/lib/libRlapack.so
##
## locale:
##  [1] LC_CTYPE=en_US.UTF-8       LC_NUMERIC=C
##  [3] LC_TIME=en_US.UTF-8        LC_COLLATE=C
##  [5] LC_MONETARY=en_US.UTF-8    LC_MESSAGES=en_US.UTF-8
##  [7] LC_PAPER=en_US.UTF-8       LC_NAME=C
##  [9] LC_ADDRESS=C               LC_TELEPHONE=C
## [11] LC_MEASUREMENT=en_US.UTF-8 LC_IDENTIFICATION=C
##
## attached base packages:
## [1] stats     graphics  grDevices utils     datasets  methods   base
##
## other attached packages:
## [1] gplots_3.0.1 Trendy_1.2.0
##
## loaded via a namespace (and not attached):
##  [1] Rcpp_0.12.16                formatR_1.5
##  [3] compiler_3.5.0              later_0.7.1
##  [5] highr_0.6                   GenomeInfoDb_1.16.0
##  [7] XVector_0.20.0              zlibbioc_1.26.0
##  [9] bitops_1.0-6                tools_3.5.0
## [11] digest_0.6.15               lattice_0.20-35
## [13] evaluate_0.10.1             Matrix_1.2-14
## [15] DelayedArray_0.6.0          shiny_1.0.5
## [17] yaml_2.1.18                 parallel_3.5.0
## [19] GenomeInfoDbData_1.1.0      stringr_1.3.0
## [21] knitr_1.20                  caTools_1.17.1
## [23] gtools_3.5.0                S4Vectors_0.18.0
## [25] shinyFiles_0.6.2            IRanges_2.14.0
## [27] grid_3.5.0                  stats4_3.5.0
## [29] segmented_0.5-3.0           rprojroot_1.3-2
## [31] Biobase_2.40.0              R6_2.2.2
## [33] BiocParallel_1.14.0         rmarkdown_1.9
## [35] RJSONIO_1.3-0               gdata_2.18.0
## [37] magrittr_1.5                matrixStats_0.53.1
## [39] backports_1.1.2             promises_1.0.1
## [41] htmltools_0.3.6             BiocGenerics_0.26.0
## [43] GenomicRanges_1.32.0        SummarizedExperiment_1.10.0
## [45] BiocStyle_2.8.0             mime_0.5
## [47] xtable_1.8-2                httpuv_1.4.1
## [49] KernSmooth_2.23-15          stringi_1.1.7
## [51] RCurl_1.95-4.10
```