Trendy: segmented regression analysis of expression dynamics in high-throughput ordered profiling experiments

Rhonda Bacher, Ning Leng, Ron Stewart

October 28, 2020

Contents

1 Overview . . . 2
1.1 The model . . . 2
2 Installation . . . 3
2.1 Install via Bioconductor . . . 3
2.2 Install via GitHub . . . 3
2.3 Install locally . . . 3
2.4 Load the package . . . 4
3 Analysis . . . 4
3.1 Input . . . 4
3.1.1 Normalized Data . . . 4
3.1.2 Time Vector . . . 4
3.2 Run Trendy . . . 6
3.3 Visualize trends of the top dynamic genes . . . 7
3.4 Visualize individual genes . . . 9
3.5 Gene specific estimates . . . 14
3.6 Breakpoint distribution over the time course . . . 15
4 More advanced analysis . . . 16
4.1 Time course with non-uniform sampling . . . 16
4.2 Time-course with replicates available . . . 18
4.3 Extract genes with specific patterns . . . 20
4.4 Determining threshold for adjusted R^2 . . . 21
4.5 Further analysis of Trendy expression trends . . . 22
5 Trendy shiny app . . . 24

6 SessionInfo 27

1 Overview

Trendy is an R package for 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. Each breakpoint represents a significant change in the gene's expression across the time-course. The optimal model is chosen as the one with the lowest BIC.

The top dynamic genes are identified as those that are well profiled by their optimal gene-specific segmented regression model. Trendy also implements functions to: visualize dynamic genes and their trends, to order dynamic genes by their trends, and to compute the distribution of breakpoints across all genes and time-points.

To illustrate Trendy here we refer specifically to time-course gene expression data, however 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).

If you use Trendy in published research, please cite:

Bacher R, Leng N, Chu LF, Ni Z, Thomson JA, Kendziorski C, Stewart R. Trendy: segmented regression analysis of expression dynamics in high-throughput ordered profiling experiments. BMC Bioinformatics. 2018 Dec;19(1):380.

1.1 The model

Denote the normalized gene expression of gene g and sample/time t as $Y_{g,t}$ for a total of G genes and a total of N samples. For each gene, Trendy fits a set of segmented regression models having 0 to K breakpoints. 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 number of breakpoints, Trendy selects the optimal model by comparing the BIC.

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 `minNumInSeg` $= 5$.

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 a high adjusted R^2.

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 (slope and standard errors are estimated by the segmented package) has p-value greater than c_{pval}, the trend of this segment will be defined as no change. Otherwise the trend will be defined as up/down based on the slope coefficient. The default value of c_{pval} is `pvalCut` = 0.1, but may also be specified by the user.

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

Trendy also computes a breakpoint distribution of the number of breakpoints over all genes alonge the time-course. Time-points with a large number of breakpoints may represent global expression changes and be targetted for follow-up investigations.

2 Installation

2.1 Install via Bioconductor

The *Trendy* package can be installed from Bioconductor if you have R version ≥ 3.5:

```
if (!requireNamespace("BiocManager", quietly=TRUE))
    install.packages("BiocManager")
BiocManager::install("Trendy")
```

2.2 Install via GitHub

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

If you have R version ≥ 3.5:

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

For prior R versions you may use the following but note that this version is not being updated:

```
install.packages("devtools")
library(devtools)
install_github("rhondabacher/Trendy", ref="devel")
```

2.3 Install locally

Trendy may also be installed locally.

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

2.4 Load the package

To load the Trendy package:

```
library(Trendy)
```

3 Analysis

3.1 Input

3.1.1 Normalized Data

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 be expression data after normalization across samples. For example, for RNA-seq data, the raw counts may be normalized using Median Normalization (Anders and Huber, 2010) via the `MedianNorm` and `GetNormalizedMat` functions in the *EBSeq* package:

```
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

3.1.2 Time Vector

The time vector is important to specify as it contains information regarding replicates and the relative timing or spacing of each sample. The order of the time vector should match the order of the columns in the expression data. Below are a few examples on how to specify the time vector for a variety of situations:

Suppose all 40 samples are equally spaced time-points:

```
time.vector <- 1:40
time.vector
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24
## [25] 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40
```

Suppose there are 20 equally spaced time points, each with 2 replicates:

```
time.vector <- rep(1:20, each = 2)
time.vector
```

```
##  [1]  1  1  2  2  3  3  4  4  5  5  6  6  7  7  8  8  9  9 10 10 11 11 12 12
## [25] 13 13 14 14 15 15 16 16 17 17 18 18 19 19 20 20
```

Suppose there are 18 unequally spaced time points, most times have 2 replicates but a few times have 3:

```
time.vector <- c(rep(1, 3), rep(2:9, each = 2), rep(10:11, 3),
                 rep(12:17, each=2), rep(18, 3))
time.vector
```

```
##  [1]  1  1  1  2  2  3  3  4  4  5  5  6  6  7  7  8  8  9  9 10 11 10 11 10
## [25] 11 12 12 13 13 14 14 15 15 16 16 17 17 18 18 18
```

```
table(time.vector)
```

```
## time.vector
##  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18
##  3  2  2  2  2  2  2  2  2  3  3  2  2  2  2  2  2  3
```

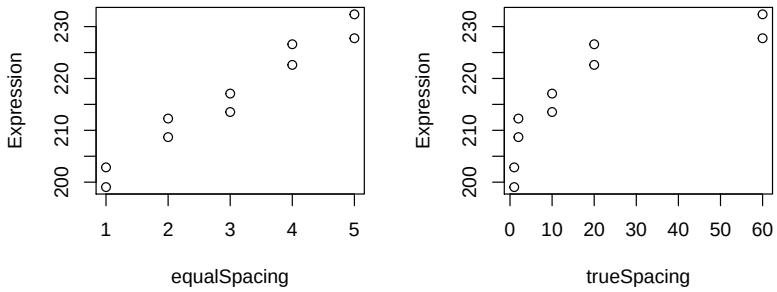
Remember, it is critical that this specification corresponds exactly to the order of samples (columns) in the normalized data matrix described in the previous section!

FAQ: Does it matter if I use the real time or equally spaced time?

Suppose you have an experiment that was sampled at minutes 1,2,10,20, and 60 with two replicates at each time. In the plot below we can clearly see that the interpretation would be quite different depending on the definition of the time vector. In the true time plot (right), the expression increases quickly initially then levels off, whereas in the equal spacing plot (left) the expression appears to increase at a constant rate.

In the majority of cases, we recommend using the true time to define the time vector. An example is given in Section 4.1 and 4.2.

```
mygene <- trendyExampleData[2,1:10]
equalSpacing <- rep(c(1:5), each=2)
trueSpacing <- c(1,1,2,2,10,10,20,20,60,60)
par(mfrow=c(1,2), mar=c(5,5,2,1))
plot(equalSpacing, mygene, ylab="Expression")
plot(trueSpacing, mygene, ylab="Expression")
```



3.2 Run Trendy

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

```
time.vector <- 1:40
res <- trendy(Data = trendyExampleData, tVectIn = time.vector, maxK = 2)

## Warning in BiocParallel::MulticoreParam(workers = NCores):  MulticoreParam() not supported
on Windows, use SnowParam()

## breakpoint estimate(s): 34.91237 39.07567

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 0.9751430 0.9739715 0.9729747 0.9711890 0.9710139
##       g10       g23       g14        g8        g5       g24        g9
## 0.9705118 0.9701402 0.9694164 0.9691341 0.9689555 0.9656732 0.9649686
##       g17       g12       g29       g16       g22       g18        g6
## 0.9648847 0.9644343 0.9632348 0.9630273 0.9627092 0.9626837 0.9619413
##       g25       g11       g30       g27       g26       g19        g7
## 0.9611528 0.9600736 0.9597989 0.9592516 0.9572072 0.9536620 0.9529077
##       g21       g13
## 0.9528865 0.9470935
```

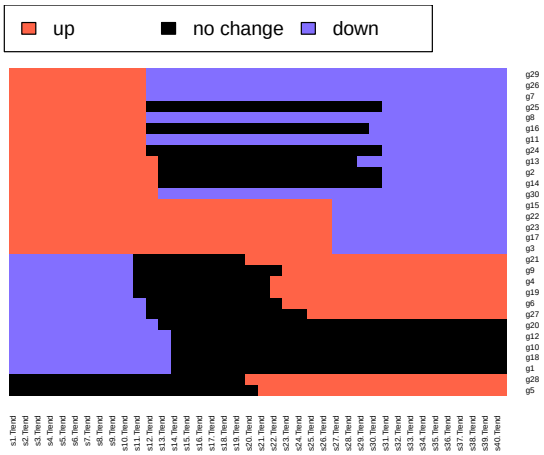
The `topTrendy` function may be used to extract top dynamic genes. By default, `topTrendy` will extract genes whose adjusted R^2, $\bar{R}^2$, is greater or equal to 0.5. To change this threshold, a user may specify the `adjR2Cut` parameter in the `topTrendy` function. The `topTrendy` function returns the Trendy output with genes sorted decreasingly 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 desired value using the parameter `meanCut`.

3.3 Visualize trends of the top dynamic genes

The object `res.top$Trend` contains the trend specification of the top genes. The function `trendHeatmap` can be used to display these trends. First, the `trendHeatmap` function classifies the top dynamic genes into three groups: those that start with 'up', start with 'down' and start with 'no change'. Within each group, genes are sorted by the position of the first breakpoint.

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

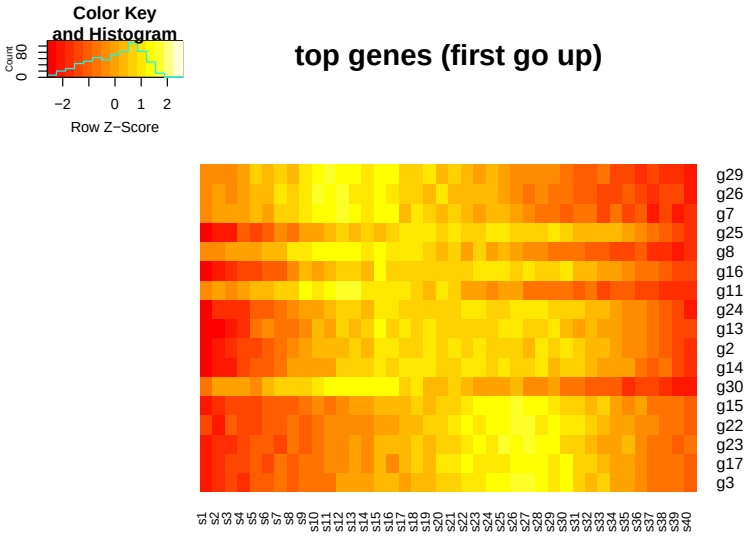



```
str(res.trend)
```

```
## 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"
```

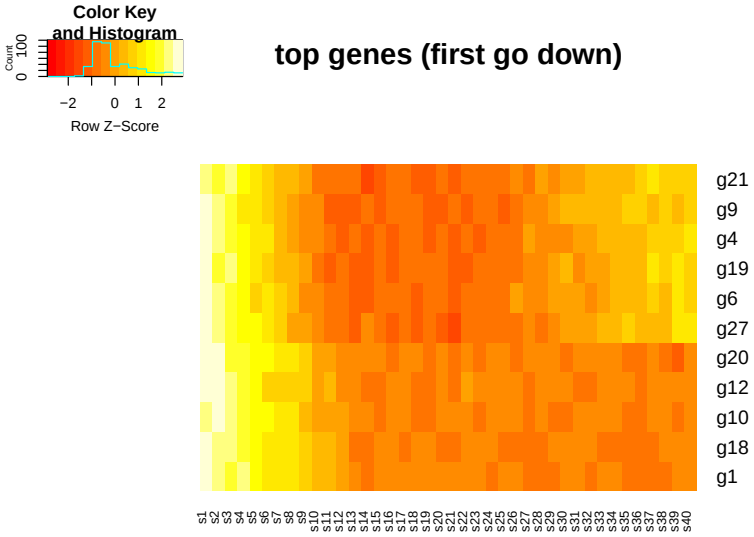
To generate an expression heatmap of the first group of genes (first go 'up'):

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



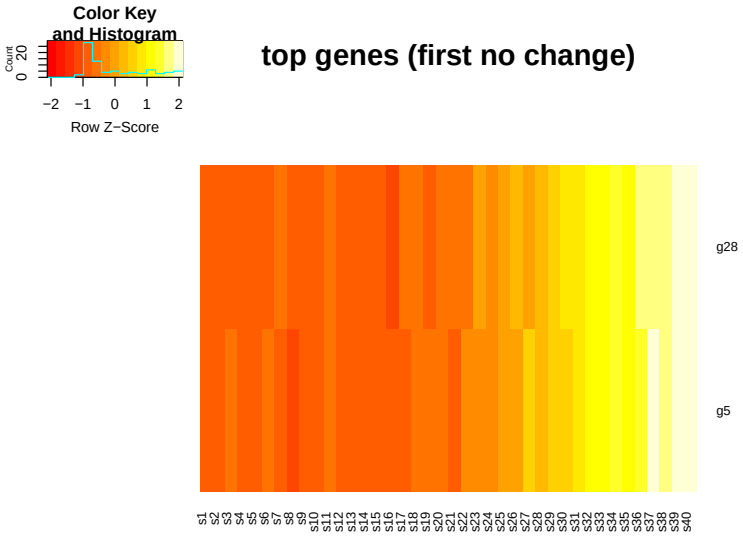
Similarly, to generate an expression heatmap of the second group of genes (first go down):

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

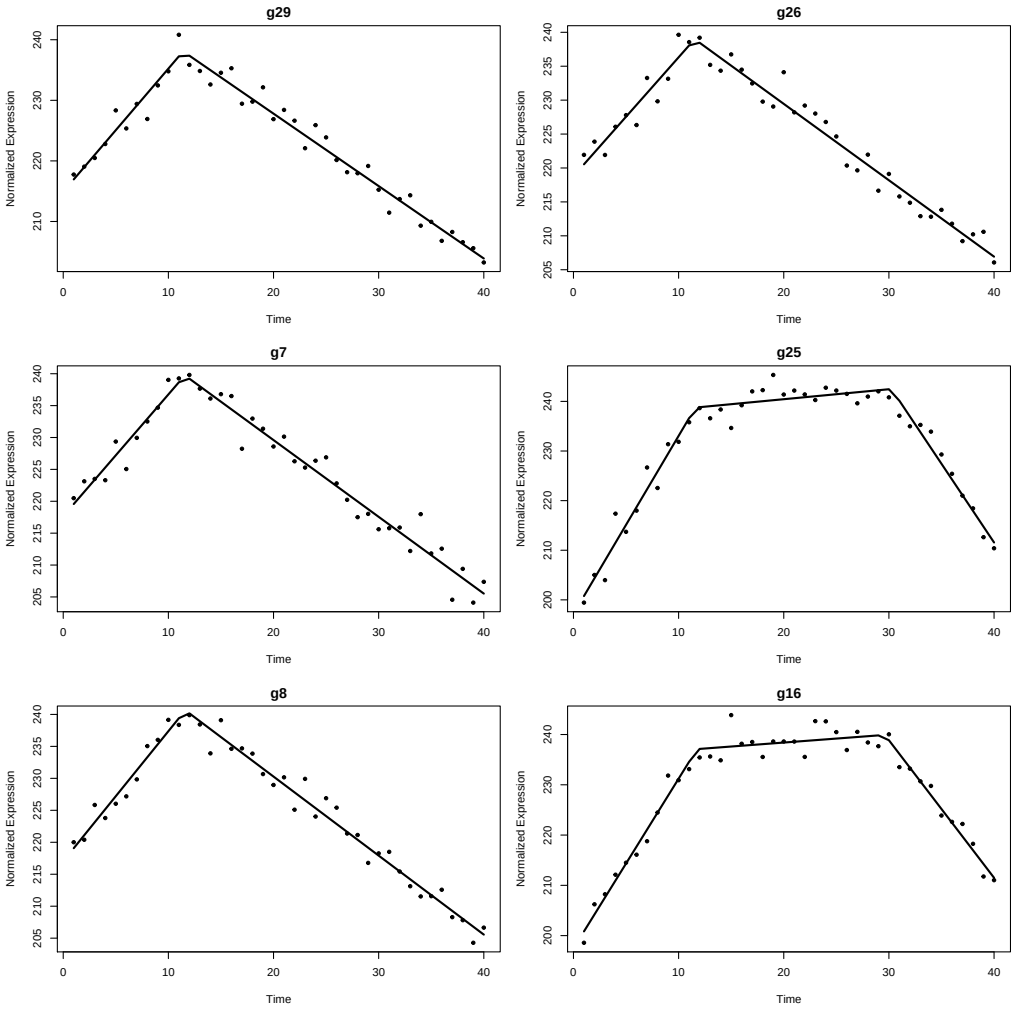


3.4 Visualize individual genes

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

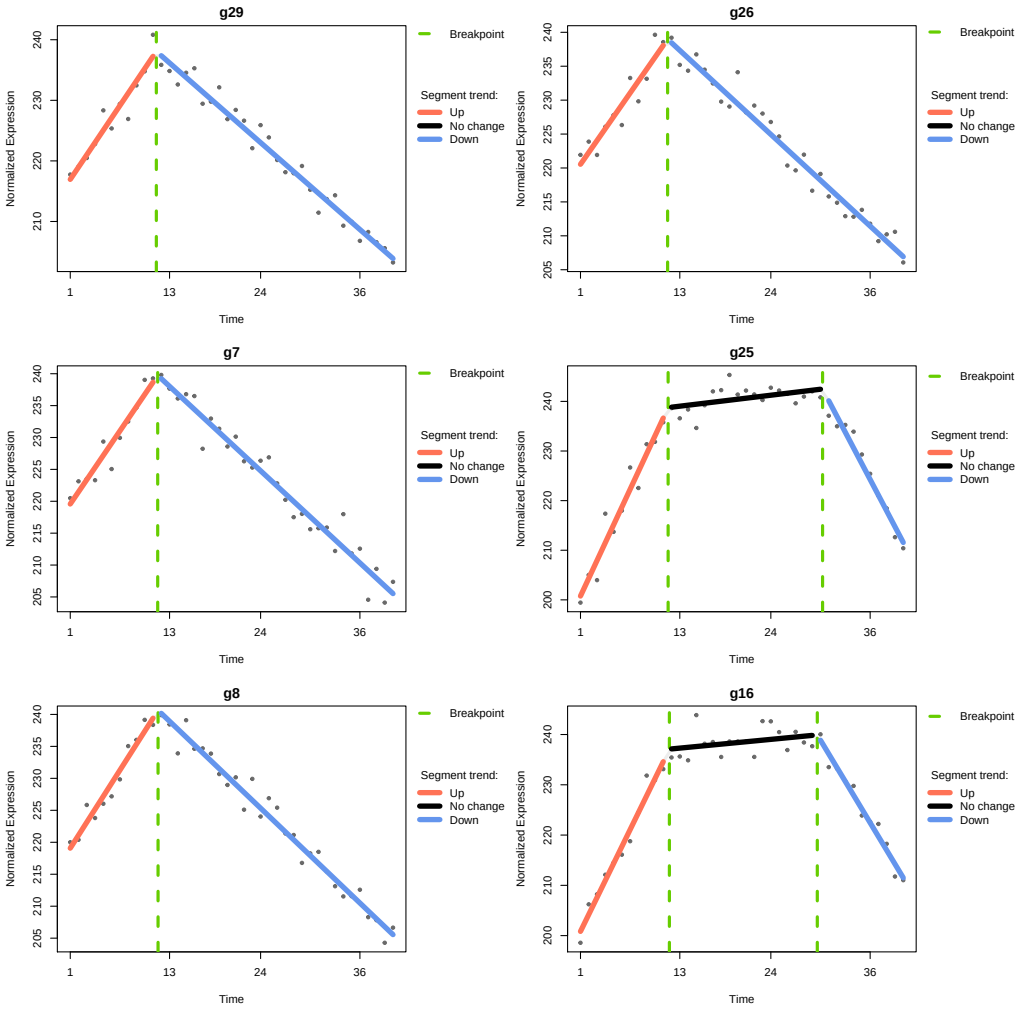
For example, to plot the top six genes in the first group of genes (first go up):

```
par(mfrow=c(3,2))
plotFeature(Data = trendyExampleData, tVectIn = time.vector, simple = TRUE,
                    featureNames = names(res.trend$firstup)[1:6],
                    trendyOutData = res)
```

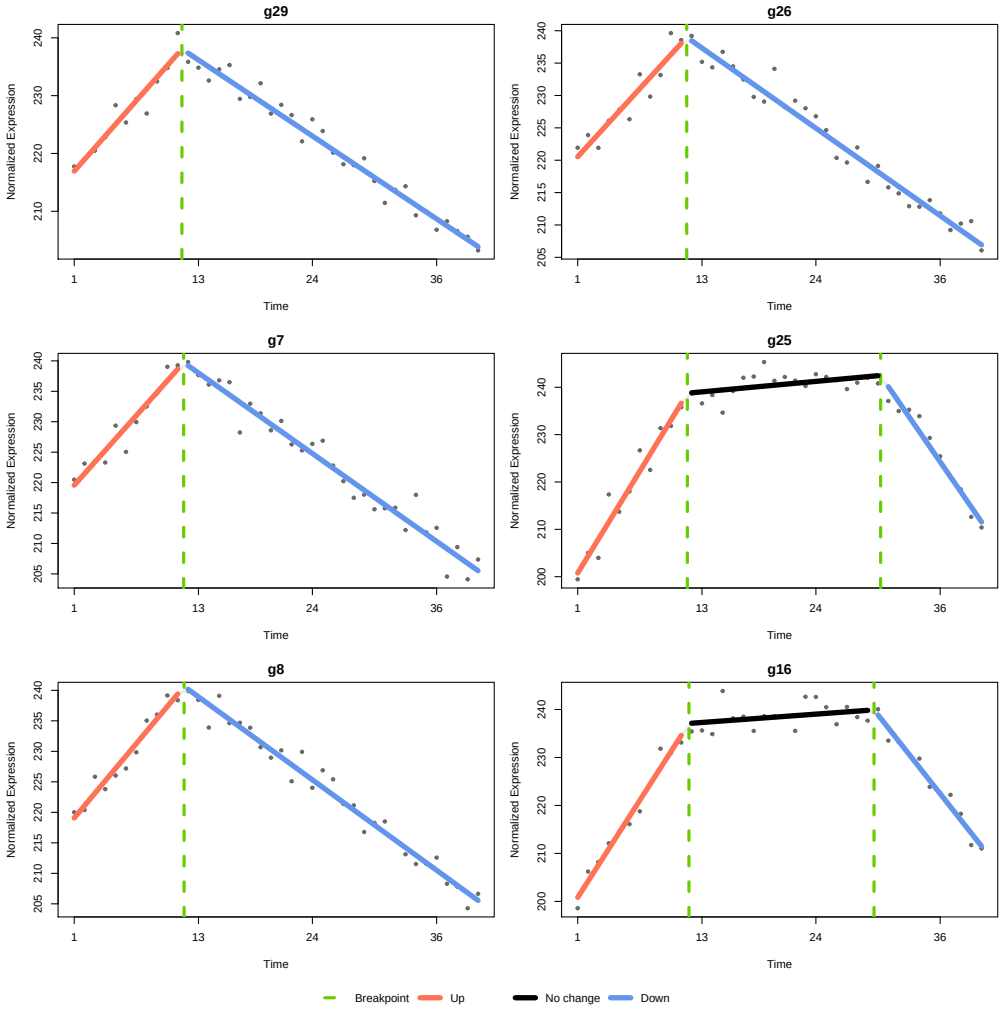


These can be plot together with segment trends colored and breakpoints highlighted by setting `simple=FALSE`. A legend can be placed by specifying `legendLocation = 'side'` or `legendLocation = 'bottom'`. The user may supress the legend by setting `showLegend = FALSE`. The size of the legend text can be adjusted using the parameter `legendCex`.

```
par(mfrow=c(3,2)) #specify the layout of multiple plots in a single panel
plotFeature(Data = trendyExampleData, tVectIn = time.vector, simple = FALSE,
                    showLegend = TRUE, legendLocation='side',cexLegend=1,
                    featureNames = names(res.trend$firstup)[1:6],
                    trendyOutData = res)
```



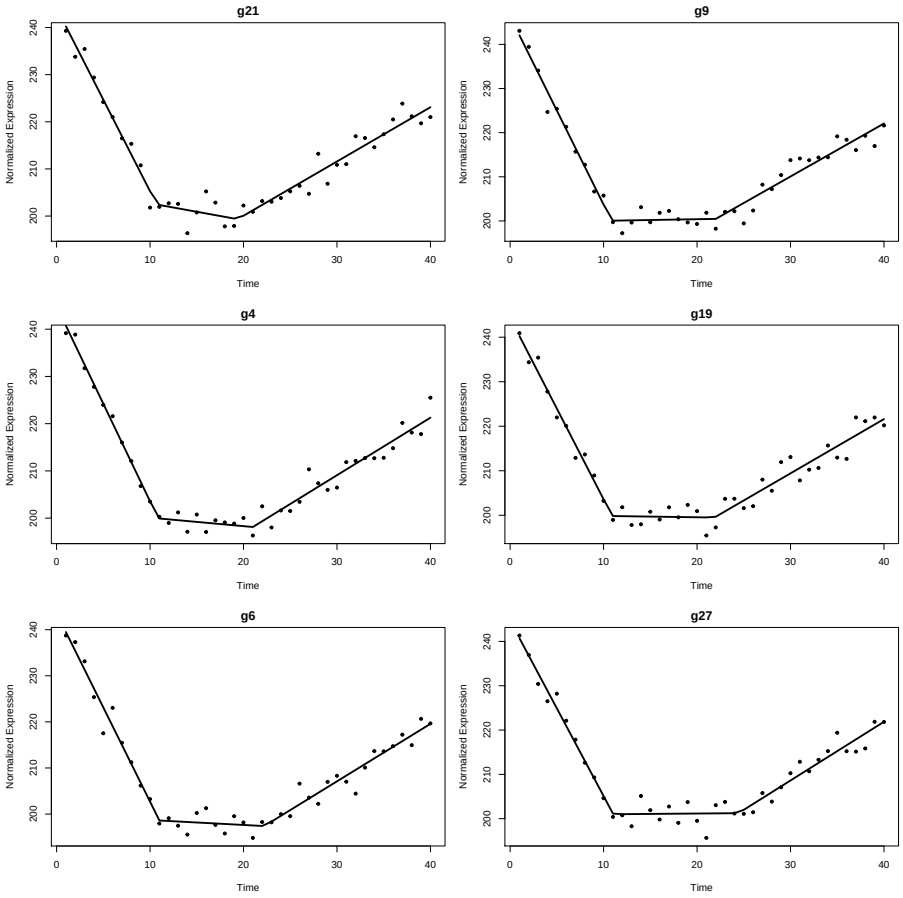

```
par(mfrow=c(3,2)) #specify the layout of multiple plots in a single panel
plotFeature(Data = trendyExampleData, tVectIn = time.vector, simple = FALSE,
                    showLegend = TRUE, legendLocation='bottom',cexLegend=1,
                    featureNames = names(res.trend$firstup)[1:6],
                    trendyOutData = res)
```



The input of function `plotFeature` requires the expression data and a list of genes of interest. The parameter `trendyOut` contains 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.

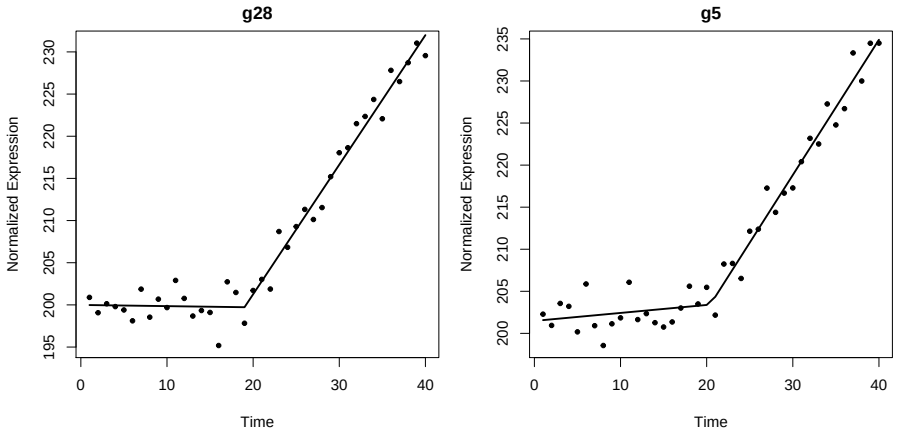
Similarly, to plot the top six genes in the second group of genes (first go down):

```
par(mfrow=c(3,2))
plotFeature(Data = trendyExampleData,tVectIn = time.vector, simple=TRUE,
                    featureNames = names(res.trend$firstdown)[1:6],
                    trendyOutData = res)
```



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

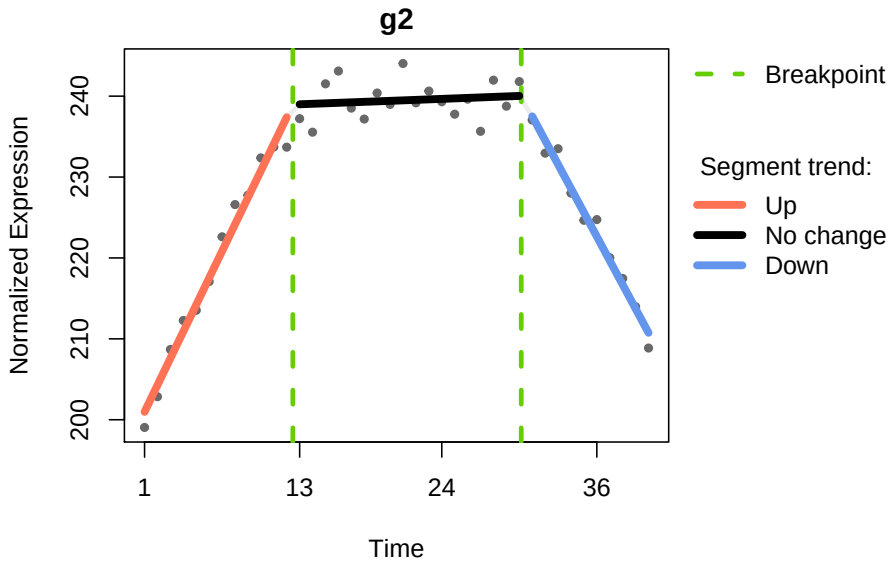
```
par(mfrow=c(1,2))

plotFeature(trendyExampleData,tVectIn = time.vector, simple=TRUE,
                    featureNames = names(res.trend$firstnochange)[1:2],
                    trendyOutData = res)
```



3.5 Gene specific estimates

For a given gene of interest, its estimated parameters can be obtained individually:

```
par(mfrow=c(1,1))
plot2 <- plotFeature(trendyExampleData,tVectIn = time.vector,
                     featureNames = "g2",
                     trendyOutData = res)
```




```
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 printout shows that for gene g2 the optimal number of breakpoints is two estimated at time-points 12 and 30. The fitted slopes for the 3 adjoining segments are 3.31, 0.06 and -2.97, which indicates the trend is 'up'-'no change'-'down.'

These estimates can be automatically formatted using the function `formatResults`, which can then 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.

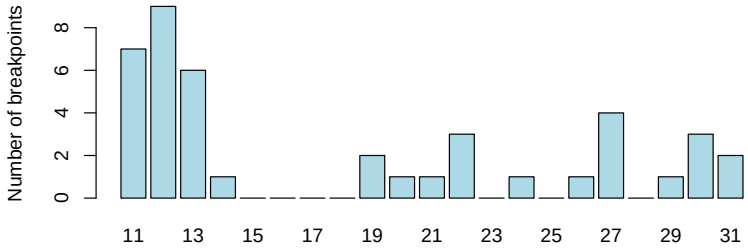
```
trendy.summary <- formatResults(res.top)
trendy.summary[1:4,1:8]

##     Feature Segment1.Slope Segment2.Slope Segment3.Slope Segment1.Trend
## g3       g3       1.572400     -2.5483000             NA              1
## g1       g1      -3.145400      0.0015484             NA             -1
## g28     g28      -0.014241      1.5366000             NA              0
## g20     g20      -3.381300     -0.0824620             NA             -1
##     Segment2.Trend Segment3.Trend Segment1.Pvalue
## g3              -1             NA     0.009070596
## g1               0             NA     0.013816451
## g28              1             NA     0.438468884
## g20              0             NA     0.015765320

# To save:
# 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 the number of breakpoints for all genes over the time course:

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



The bar plot indicates that a number of genes have breakpoints around times 11 - 13.

4 More advanced analysis

4.1 Time course with non-uniform sampling

If the samples were collected at different time intervals then it is highly suggested to denote the time vector by this scale (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.

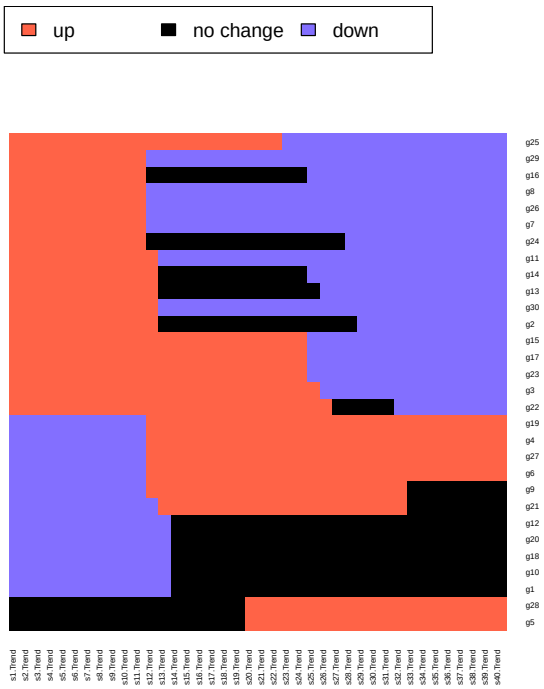
For example, suppose for the simulated data, the first 30 samples were collected every hour and the remaining 10 samples were collected every 5 hours. We may define the time vector as:

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

```
##  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 using the empirical collecting time instead of sample ID (1-40):

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

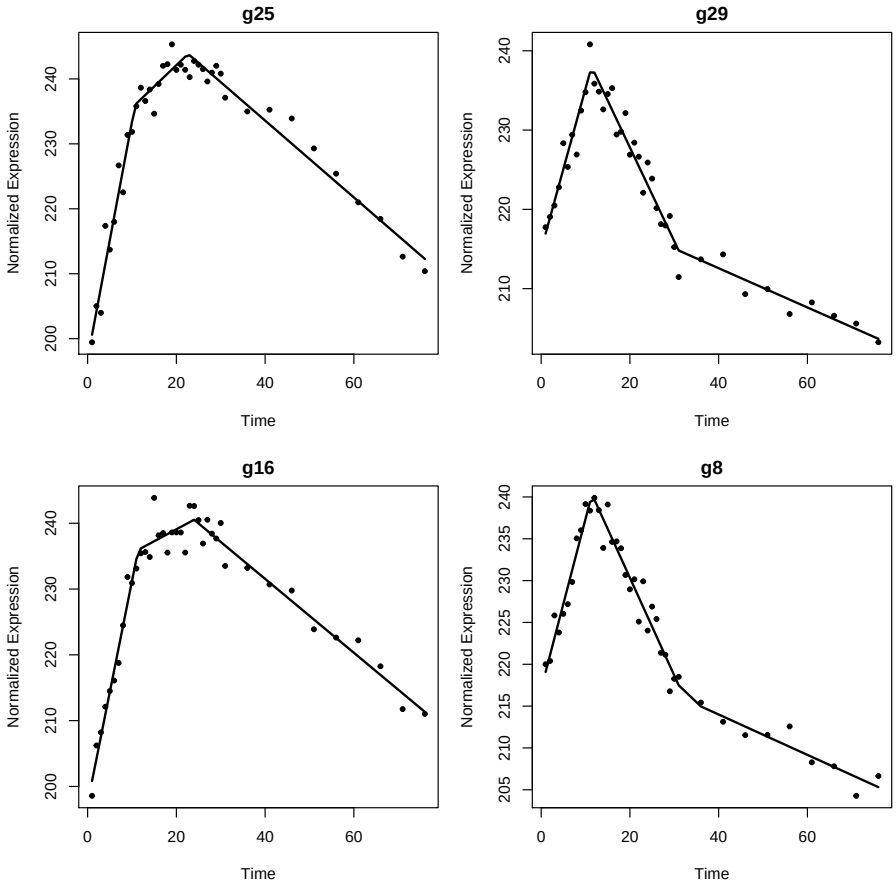



```
str(res.trend2)
```

```
## List of 3
##  $ firstup      : Named num [1:17] 10.7 11.4 11.4 11.4 11.6 ...
##   ..- attr(*, "names")= chr [1:17] "g25" "g29" "g16" "g8" ...
##  $ 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 four genes that have up-regulated pattern at the beginning of the time course:

```
par(mfrow=c(2,2))
plotFeature(trendyExampleData, tVectIn=time.vector, simple = TRUE,
                      featureNames = names(res.trend2$firstup)[1:4],
                      trendyOutData = 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 replicates directly in the the `tVectIn` parameter in the `trendy` function.

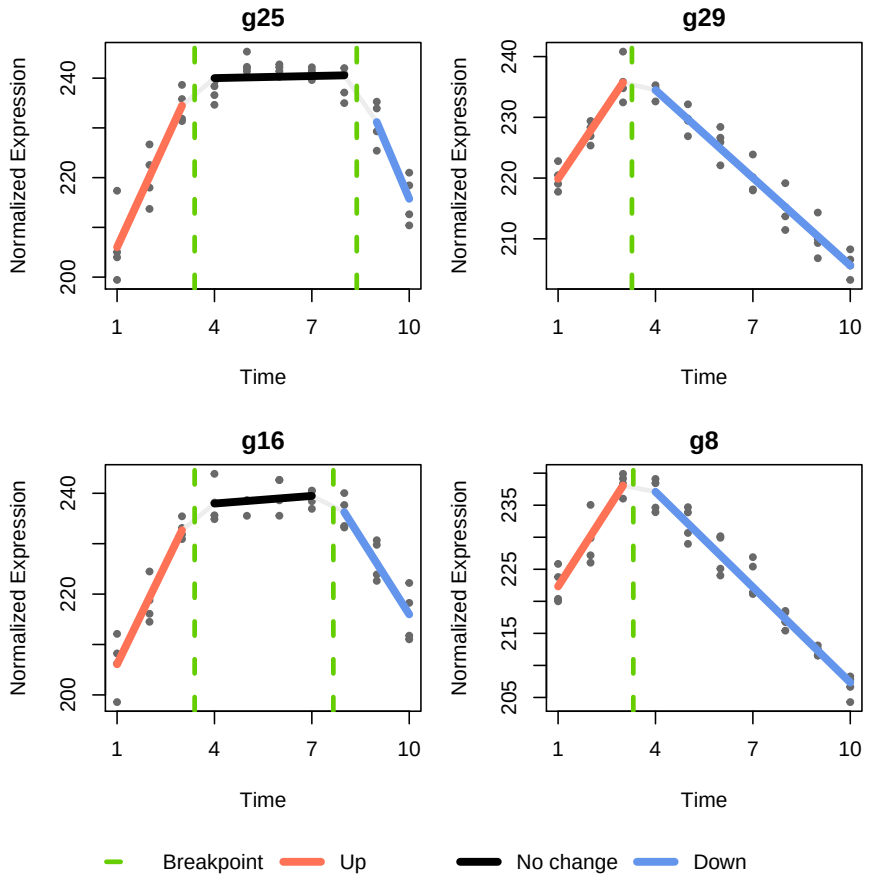
For example, suppose for the simulated data, 10 time points were observed 4 times each. We may define the time vector as:

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

```
##  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
```

```
res3 <- trendy(Data = trendyExampleData, tVectIn = time.vector, maxK=2)
res3 <- results(res3)
res.top3 <- topTrendy(res3)
res.trend3 <- trendHeatmap(res.top3)
```


```
par(mfrow=c(2,2))
plotFeature(trendyExampleData, tVectIn=time.vector, simple = FALSE,
                          legendLocation = 'bottom',
                          featureNames = names(res.trend2$firstup)[1:4],
                          trendyOutData = res3)
```



4.3 Extract genes with specific patterns

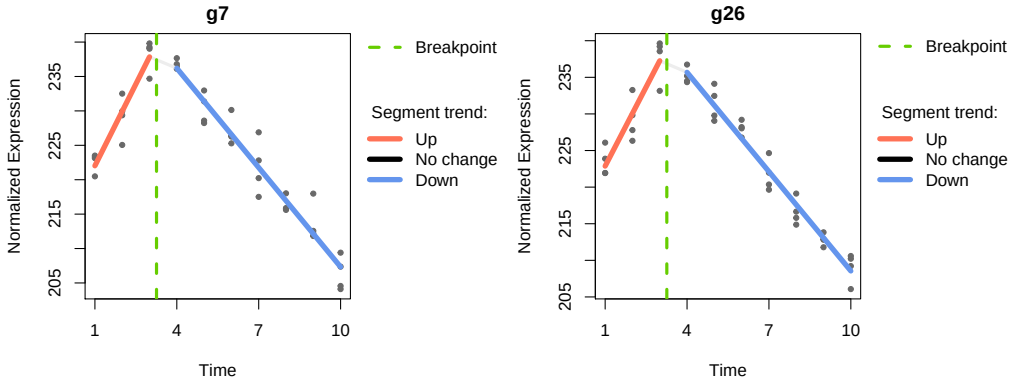
Users can search for genes with patterns of interest using the `extractPatterns` function in the Trendy package.

For example, genes that have a peak along the time-course will have fitted trend somewhere as "up-down":

```
# Genes that peak
pat1 <- extractPattern(res3, Pattern = c("up","down"))
head(pat1)
```

```
##    Gene BreakPoint1
## 3    g7    3.256781
## 1   g26    3.259182
## 9   g11    3.270083
## 2   g29    3.273990
## 5    g8    3.317005
## 11  g30    3.565860
```

```
par(mfrow=c(1,2))
plotPat1 <- plotFeature(trendyExampleData, tVectIn=time.vector,
                        featureNames = pat1$Gene[1:2],
                        trendyOutData = res3)
```



We may only want those where the peak has occured after some time-point. This can be specified using the `Delay` parameter:

```
# Genes that peak after some time
pat3 <- extractPattern(res3, Pattern = c("up","down"), Delay = 7)
head(pat3)
```

```
##    Gene BreakPoint1
## 4   g23    7.001090
## 7    g3    7.104316
## 10  g22    7.184048
```

To search for genes that have a 'no change' segment, the `extractPattern` function accepts both 'no change' and 'same'. For example, here we search for genes that are stable and then go up:

```
# Genes that are constant, none
extractPattern(res2, Pattern = c("no change", "up"))
```

```
##   Gene BreakPoint1
## 1  g28    19.00003
## 2   g5    19.46826
```

```
extractPattern(res2, Pattern = c("same", "up"))
```

```
##   Gene BreakPoint1
## 1  g28    19.00003
## 2   g5    19.46826
```

4.4 Determining threshold for adjusted R^2

Depending on the type of experiment (RNA-seq, microarray, scRNA-seq, etc.) and level of noise, different thresholds for the adjusted R^2 may be used.

One way to decide an appropriate threshold is to perform a permutation procedure as follows:

```
library(Trendy)
res.r2 <- c()
for(i in 1:100) { # permute 100 times at least
  BiocParallel::register(BiocParallel::SerialParam())
  seg.shuffle <- trendy(trendyExampleData[sample(1:nrow(data.norm.scale), 100),], #sample genes each time
                        tVectIn = sample(time.vector), # shuffle the time vector
                        saveObject=FALSE, numTry = 5)
```

```
  res <- results(seg.shuffle)
  res.r2 <- c(res.r2, sapply(res, function(x) x$AdjustedR2))
}

# Histogram of all R^2
hist(res.r2, ylim=c(0,1000), xlim=c(0,1), xlab=expression(paste("Adjusted R"^"2")))

# Say you want to use the value such that less than 1% of permutations reach:
sort(res.r2, decreasing=T)[round(.01 * length(res.r2))]
# Say you want to use the value such that less than 5% of permutations reach:
sort(res.r2, decreasing=T)[round(.05 * length(res.r2))]
```

Note: For an experiment with replicates, you should shuffle the replicated timepoints together:

```
time.vector = c(1,1,2,2,10,10,20,20,60,60)
# How to shuffle the replicates -together-
set.seed(12)
shuf.temp=sample(unique(time.vector))
print(shuf.temp)
```

```
## [1]  2 60 10  1 20
```

```
setshuff=do.call(c,lapply(shuf.temp, function(x) which(!is.na(match(time.vector, x)))))
use.shuff <- time.vector[setshuff]
print(use.shuff)
```

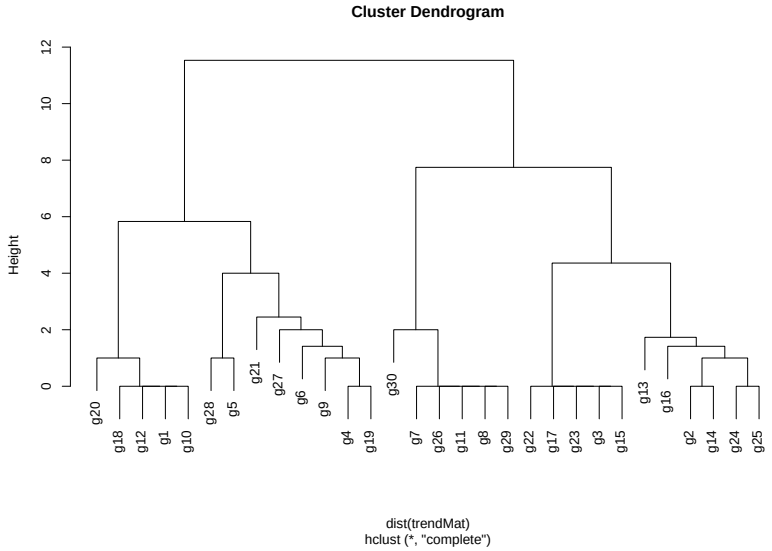
```
##  [1]  2  2 60 60 10 10  1  1 20 20
```

```
## Then in the permutation code you'll do:

for(i in 1:100) { # permute 100 times at least
  BiocParallel::register(BiocParallel::SerialParam())

  shuf.temp=sample(unique(time.vector))
  setshuff=do.call(c,lapply(shuf.temp, function(x) which(!is.na(match(time.vector, x)))))
  use.shuff <- time.vector[setshuff]
  seg.shuffle <- trendy(trendyExampleData[sample(1:nrow(data.norm.scale), 100),], #sample genes each time
                        tVectIn = use.shuff, # shuffle the time vector
                        saveObject=FALSE, numTry = 5)
  res <- results(seg.shuffle)
  res.r2 <- c(res.r2, sapply(res, function(x) x$AdjustedR2))
}
```

4.5 Further analysis of Trendy expression trends

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

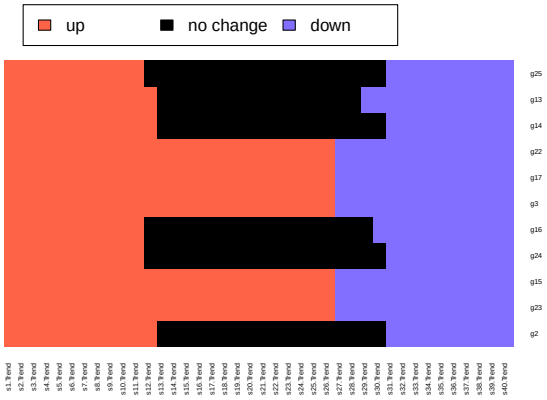
```
# Get trend matrix:
trendMat <- res.top$Trends
# Cluster genes using hierarchical clustering:
hc.results <- hclust(dist(trendMat))
plot(hc.results) #Decide how many clusters to choose
```



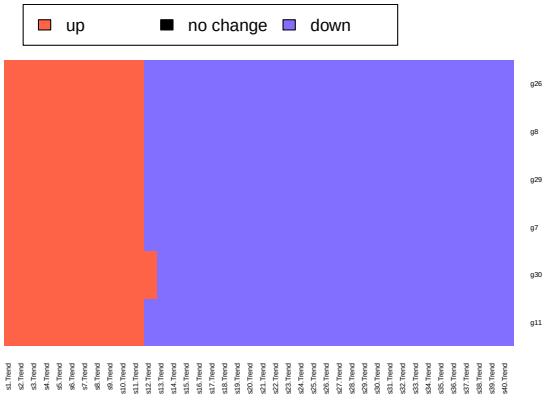

```
#Let's say there are 4 main clusters
hc.groups <- cutree(hc.results, k = 4)
```

Here are heatmaps of genes in Clusters 1 and 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 used as input for gene enrichment analysis. Two popular gene set enrichment tools include: enrichr (web-based, http://amp.pharm.mssm.edu/Enrichr/) or GSEA (via MSigDB: http://software.broadinstitute.org/gsea/msigdb/index.jsp).

5 Trendy shiny app

The Trendy shiny app requires the .RData object output from the `trendy` function, which can be obtained by setting `saveObject=TRUE` and specifying a name via the `fileName` parameter.

```
res <- trendy(trendyExampleData, tVectIn = 1:40, maxK=2, saveObject = TRUE, fileName="exampleObject")

## breakpoint estimate(s): 34.91237 39.07567

res <- results(res)
```

Then in R run:

```
trendyShiny()
```

Below are screenshots of the Shiny application:

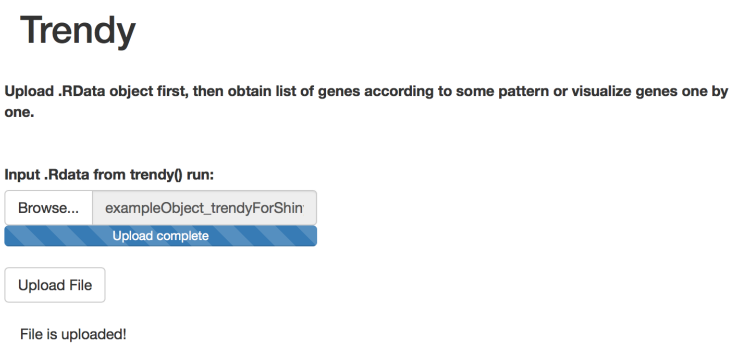


Figure 1: Upload shiny object

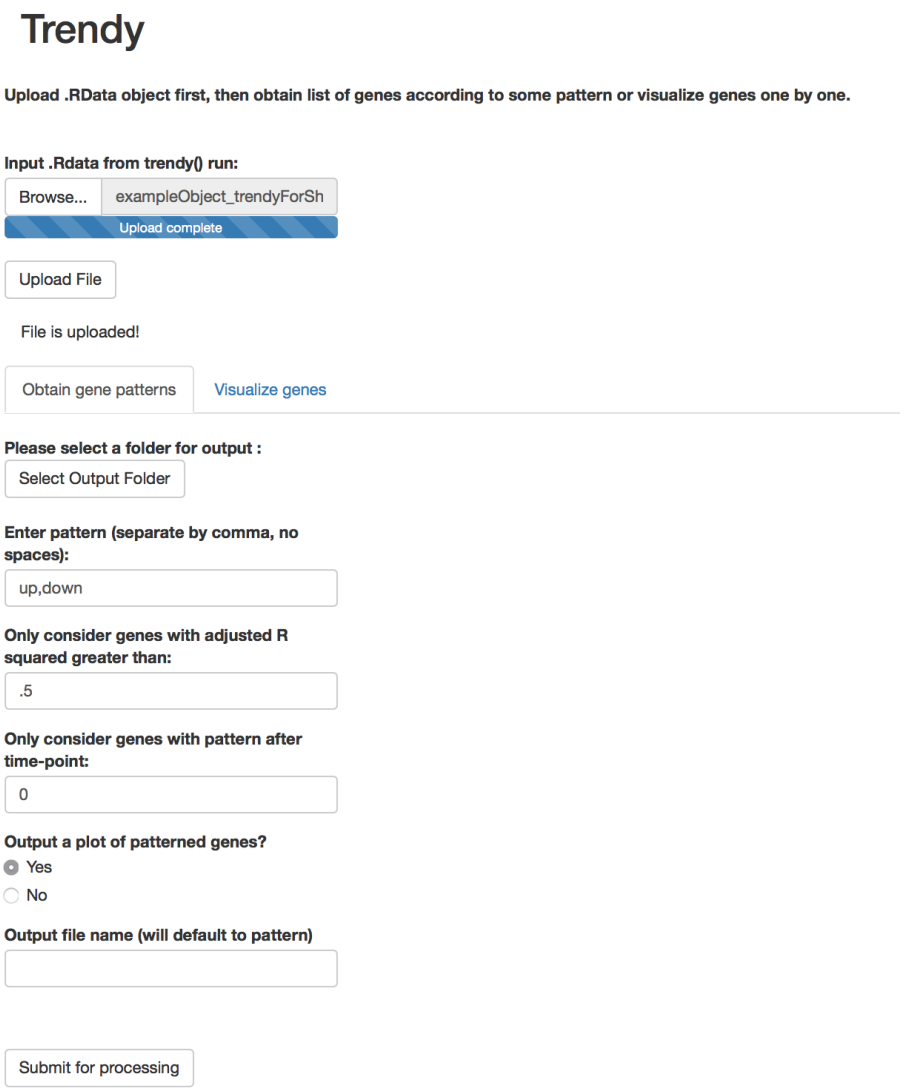


Figure 2: Find all genes with a given pattern

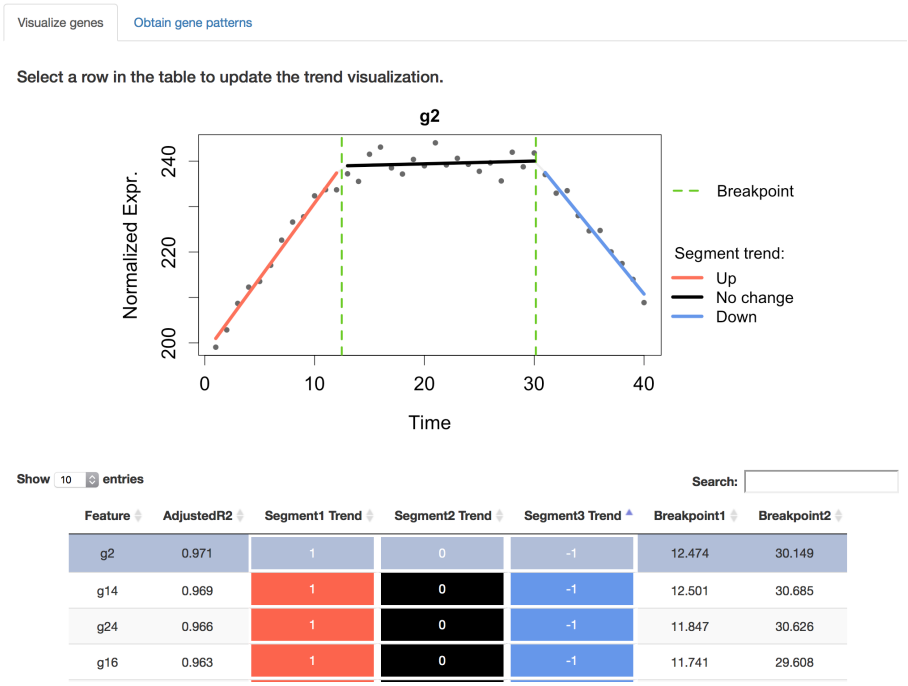


Figure 3: Search genes individually

6 SessionInfo

```
sessionInfo()
```

```
## R version 4.0.3 (2020-10-10)
## Platform: x86_64-w64-mingw32/x64 (64-bit)
## Running under: Windows Server 2012 R2 x64 (build 9600)
##
## Matrix products: default
##
## locale:
## [1] LC_COLLATE=C
## [2] LC_CTYPE=English_United States.1252
## [3] LC_MONETARY=English_United States.1252
## [4] LC_NUMERIC=C
## [5] LC_TIME=English_United States.1252
##
## attached base packages:
## [1] stats     graphics  grDevices utils     datasets  methods   base
##
## other attached packages:
## [1] gplots_3.1.0  Trendy_1.12.0
##
## loaded via a namespace (and not attached):
##  [1] SummarizedExperiment_1.20.0 gtools_3.8.2
##  [3] xfun_0.18                   splines_4.0.3
##  [5] lattice_0.20-41             vctrs_0.3.4
##  [7] htmltools_0.5.0             stats4_4.0.3
##  [9] yaml_2.2.1                  rlang_0.4.8
## [11] later_1.1.0.1               pillar_1.4.6
## [13] BiocParallel_1.24.0         BiocGenerics_0.36.0
## [15] segmented_1.3-0             matrixStats_0.57.0
## [17] GenomeInfoDbData_1.2.4      lifecycle_0.2.0
## [19] stringr_1.4.0               MatrixGenerics_1.2.0
## [21] zlibbioc_1.36.0             caTools_1.18.0
## [23] evaluate_0.14               Biobase_2.50.0
## [25] knitr_1.30                  IRanges_2.24.0
## [27] fastmap_1.0.1               httpuv_1.5.4
## [29] GenomeInfoDb_1.26.0         parallel_4.0.3
## [31] highr_0.8                   Rcpp_1.0.5
## [33] KernSmooth_2.23-17          xtable_1.8-4
## [35] formatR_1.7                 promises_1.1.1
## [37] BiocManager_1.30.10         DelayedArray_0.16.0
## [39] S4Vectors_0.28.0            jsonlite_1.7.1
## [41] XVector_0.30.0              mime_0.9
## [43] shinyFiles_0.8.0            fs_1.5.0
## [45] BiocStyle_2.18.0            digest_0.6.27
## [47] stringi_1.5.3               shiny_1.5.0
## [49] GenomicRanges_1.42.0        grid_4.0.3
## [51] tools_4.0.3                 bitops_1.0-6
## [53] magrittr_1.5                RCurl_1.98-1.2
## [55] tibble_3.0.4                crayon_1.3.4
## [57] pkgconfig_2.0.3             ellipsis_0.3.1
## [59] Matrix_1.2-18               rmarkdown_2.5
## [61] R6_2.5.0                    compiler_4.0.3
```