`GeneSelector` package vignette

Martin Slawski[1,2] * Anne-Laure Boulesteix[1,2,3] †

[1] Sylvia Lawry Centre, Munich, Germany
[2] Institute for Medical Informatics, Biometry and Epidemiology, Ludwig-Maximilians-University Munich, Germany
[3] Department of Statistics, Ludwig-Maximilians-University Munich, Germany

Abstract

This is the vignette of the Bioconductor add-on package `GeneSelector` which contains methods to assess quantitatively the variability among multiple gene rankings, obtained by using altered datasets or several ranking procedures. The resulting multiplicity problem is addressed by functionality for rank aggregation.

1 Introduction

An important aspect of microarray data analysis is the detection of genes that are differentially expressed, e.g. in different experimental conditions or in individuals with different phenotypes. The results of microarray studies are usually the starting point for further more expensive and time-consuming experiments, which involve only a small number of candidate genes **?**. The set of candidate genes is typically determined by computing a two-sided test statistic for each gene $j = 1, \ldots, p$, and ordering them decreasingly according to the size of the absolute of the statistic. This yields an ordered list $\boldsymbol{l} = (l_m, \ \ m = 1, \ldots, p)$ and ranks $\boldsymbol{r} = (r_j, \quad j = 1, \ldots, p)$ defined by $r_j = m \ \Leftrightarrow \ \ l_m = j, \quad j, m = 1, \ldots, p$. The genes at the top of the list are displayed in almost all microarray-related biomedical publications, often considered as an unequivocal and definitive result. Critical voices have pointed out that this procedure might yield false research findings **?**, since it ignores the variability of the obtained ordered lists **?**, **?**. The `GeneSelector` package tries to quantify this variability by mimicking changed data situations via resampling- and related strategies, and then comparing the results to those obtained with the reference datasets. For this purpose, `GeneSelector` assembles several stability measures for rank data.
A second source of variability, which, to our knowledge, has not been addressed in the literature, is the multiplicity of test statistics (= ranking criteria) proposed for gene expression data with the aim to cope with the 'small n, large p' situation, with n denoting the number of replicates. `GeneSelector` implements a collection of fourteen such ranking criteria, displayed in Table **??**, and hence enables the user to explore this source of variability. Using several ranking criteria instead of only one may additionally be seen as sensitivity analysis, since most criteria rely on idealized, hard-to-check assumptions. Each ranking criteria

*ms@cs.uni-sb.de
†boulesteix@ibe.med.uni-muenchen.de

Method	Function name	Package	Reference
Foldchange	`RankingFC`		
t-statistic	`RankingTstat`		
Welch's t statistic	`RankingWelchT`		
Bayesian t-statistic (1)	`RankingBaldiLong`		?
Bayesian t-statistic (2)	`RankingFoxDimmic`		?
Shrinkage t-statistic	`RankingShrinkageT`		?
Soft-thresholded t-statistic	`RankingSoftthresholdT`		?
Parametric empirical Bayes	`RankingLimma`	`limma`	?
B-statistic	`RankingBstat`	`sma`	?
Nonparametric empirical Bayes	`RankingEbam`	`siggenes`	?
SAM	`RankingSam`	`samr`	?
Wilcoxon statistic	`RankingWilcoxon`		
Wilcoxon statistic, empirical Bayes	`RankingWilcEbam`	`siggenes`	?
Permutation test	`RankingPermutation`	`multtest`	

Table 1: Overview of the ranking procedures in `GeneSelector`. If the 'package' is not given, then the respecive procedure is *not* imported from a foreign package.

produces its own result, whereas the user may be confronted with the dilemma of finding exactly one result, which should unify all results as good as possible, hopefully giving rise to an improved and more stable ranking and in turn to a set of candidate genes with as least as possible false positives. In this spirit, our package offers a `GeneSelector` function as well as several methods for rank aggregation.

2 Illustration

2.1 Description of the data set

We demonstrate the functionalities of `GeneSelector` in the classical setting of two independent samples, each of size 10. We simulate a gene expression matrix `x` containing $2,000$ genes in the following manner.

- Gene expression intensities are drawn from a multivariate normal distribution with zero mean vector and covariance which itself has been drawn randomly from an inverse Wishart distribution.
- The first 40 genes are differentially expressed. The differences in the means between the two classes are simulated independently according to a normal distribution with variance 0.9.

We access the data using the lines:

```
> data(toydata)
> y <- as.numeric(toydata[1,])
> x <- as.matrix(toydata[-1,])
> dim(x)
```

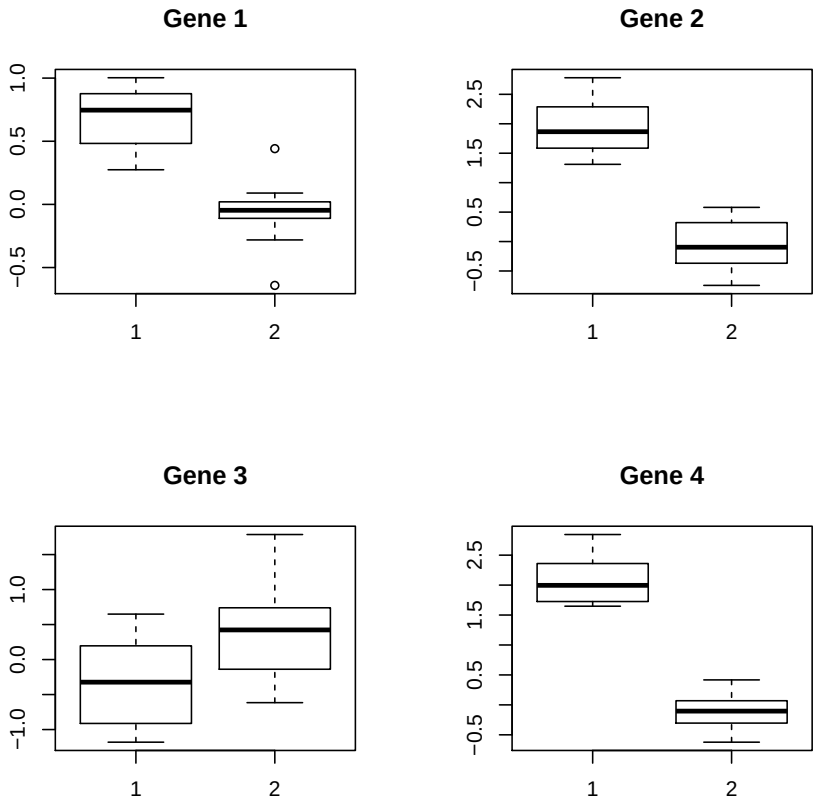
```
[1] 2000   20
```

```
> table(y)
```

```
y
 1  2
10 10
```

Knowing that the first genes are differentially expressed, we make boxplots of the gene expression intensities of the first four genes:

```
> par(mfrow=c(2,2))
> for(i in 1:4) boxplot(x[i,]~y, main=paste("Gene", i))
```



2.2 Rankings

We now perform a ranking using the ordinary t-statistic.

```
> ordT <- RankingTstat(x, y, type="unpaired")
```

The resulting objects are all instances of the class `GeneRanking`.
To get basic information, we use the commands:

```
> getSlots("GeneRanking")

          x           y   statistic     ranking        pval        type
   "matrix"    "factor"   "numeric"   "numeric"    "vector" "character"
     method
"character"

> str(ordT)

Formal class 'GeneRanking' [package "GeneSelector"] with 7 slots
  ..@ x        : num [1:2000, 1:20] 1 2.78 -1.18 2.79 -2.95 ...
  .. ..- attr(*, "dimnames")=List of 2
  .. .. ..$ : chr [1:2000] "1" "2" "3" "4" ...
  .. .. ..$ : chr [1:20] "arr1" "arr2" "arr3" "arr4" ...
  ..@ y        : Factor w/ 2 levels "1","2": 1 1 1 1 1 1 1 1 1 1 ...
  ..@ statistic: Named num [1:2000] 6.32 9.53 -2.29 13.09 -7.93 ...
  .. ..- attr(*, "names")= chr [1:2000] "1" "2" "3" "4" ...
  ..@ ranking  : Named int [1:2000] 11 3 185 1 5 17 13 451 6 375 ...
  .. ..- attr(*, "names")= chr [1:2000] "1" "2" "3" "4" ...
  ..@ pval     : Named num [1:2000] 5.83e-06 1.85e-08 3.44e-02 1.23e-10 2.79e-07 ...
  .. ..- attr(*, "names")= chr [1:2000] "1" "2" "3" "4" ...
  ..@ type     : chr "unpaired"
  ..@ method   : chr "ordinaryT"

> show(ordT)

Ranking by ordinaryT,
number of genes: 2000.

> toplist(ordT)

   index statistic         pval
1      4 13.087900 1.232978e-10
2     11 10.404717 4.833338e-09
3      2  9.533551 1.853162e-08
4     26 -8.378361 1.261238e-07
5      5 -7.927116 2.791221e-07
6      9 -7.744184 3.880996e-07
7     23  7.392767 7.402187e-07
8     38 -6.986973 1.592778e-06
9     28 -6.786421 2.345378e-06
10    40  6.421584 4.808461e-06

>
```

The last command yields the top-ranking genes according to the respective procedure.

2.3 Altered data sets

In order to inspect stability of the obtained ranking with respect to changes in the data, we use resampling techniques implemented in `GeneSelector`. The following command produces jackknif-ed data sets, i.e. datasets resulting from successively removing exactly one sample from the complete sample:

```
> loo <- GenerateFoldMatrix(y = y, k=1)
> show(loo)

number of removed samples per replicate: 1
number of replicates: 20
constraints: minimum classize for each class: 9
```

We plug this into the method `RepeatRanking`, which determines the ranking 20 times, i.e. for each removed observation, anew:

```
> loor_ordT <- RepeatRanking(ordT, loo)
>
```

The object `loo` may additionally be used in the following manner:

```
> ex1r_ordT <- RepeatRanking(ordT, loo, scheme = "labelexchange")
>
```

The argument `scheme = "labelexchange"` specifies that instead of leaving one observation out, it is assigned the opposite class label.
We may also use the bootstrap, e.g.

```
> boot <- GenerateBootMatrix(y = y, maxties=3, minclassize=5, repl=30)
> show(boot)

number of bootstrap replicates: 30
constraints: minimum classize for each class: 5
             maximum number of ties per observation: 3

> boot_ordT <- RepeatRanking(ordT, boot)
>
```

... or add a small amount of noise to the observed expression intensities:

```
> noise_ordT <- RepeatRanking(ordT, varlist=list(genewise=TRUE, factor=1/10))
```

To get a toplist that tabulates how top list positions are distributed over all repeated rankings, we use:

```
> toplist(loor_ordT, show=FALSE)

original dataset:

   index statistic        pvals
4      4 13.087900 1.232978e-10
11    11 10.404717 4.833338e-09
2      2  9.533551 1.853162e-08
26    26 -8.378361 1.261238e-07
5      5 -7.927116 2.791221e-07
9      9 -7.744184 3.880996e-07
23    23  7.392767 7.402187e-07
38    38 -6.986973 1.592778e-06
28    28 -6.786421 2.345378e-06
40    40  6.421584 4.808461e-06


In the following table, rownames correspond to gene indices.
The columns contain the absolute frequencies for the corresponding ranks
over all replications.
Genes are ordered according to the first column, then to the second, and so on.

   Rank 1 Rank 2 Rank 3 Rank 4 Rank 5 Rank 6 Rank 7 Rank 8 Rank 9 Rank 10
4      20      0      0      0      0      0      0      0      0       0
11      0     18      2      0      0      0      0      0      0       0
2       0      1     17      2      0      0      0      0      0       0
26      0      0      1     13      5      1      0      0      0       0
5       0      1      0      3      7      7      2      0      0       0
9       0      0      0      1      4     10      5      0      0       0
23      0      0      0      0      3      2     11      4      0       0
38      0      0      0      0      1      0      0     11      7       1
40      0      0      0      0      0      0      0      2      3       4
12      0      0      0      0      0      0      0      0      0       3
28      0      0      0      1      0      0      1      1      9       5
1       0      0      0      0      0      0      1      1      1       4
7       0      0      0      0      0      0      0      1      0       1
14      0      0      0      0      0      0      0      0      0       1
29      0      0      0      0      0      0      0      0      0       1
```

As an exploratory tool to examine the difference in rankings between original and perturbed data sets, a `plot` command is available.

From Figure **??**, it is obvious that variability increases for a higher list position. Moreover, the figure shows that variability depends on the method used to generate altered data sets. In this example, the bootstraped rankings are more scattered around the angle bisector than the jackknif-ed rankings.

```
> par(mfrow=c(2,2))
> plot(loor_ordT, col="blue", pch=".", cex=2.5, main = "jackknife")
> plot(ex1r_ordT, col="blue", pch=".", cex=2.5, main = "label exchange")
> plot(boot_ordT, col="blue", pch=".", cex=2.5, main = "bootstrap")
> plot(noise_ordT, frac=1/10, col="blue", pch=".", cex=2.5, main = "noise")
```

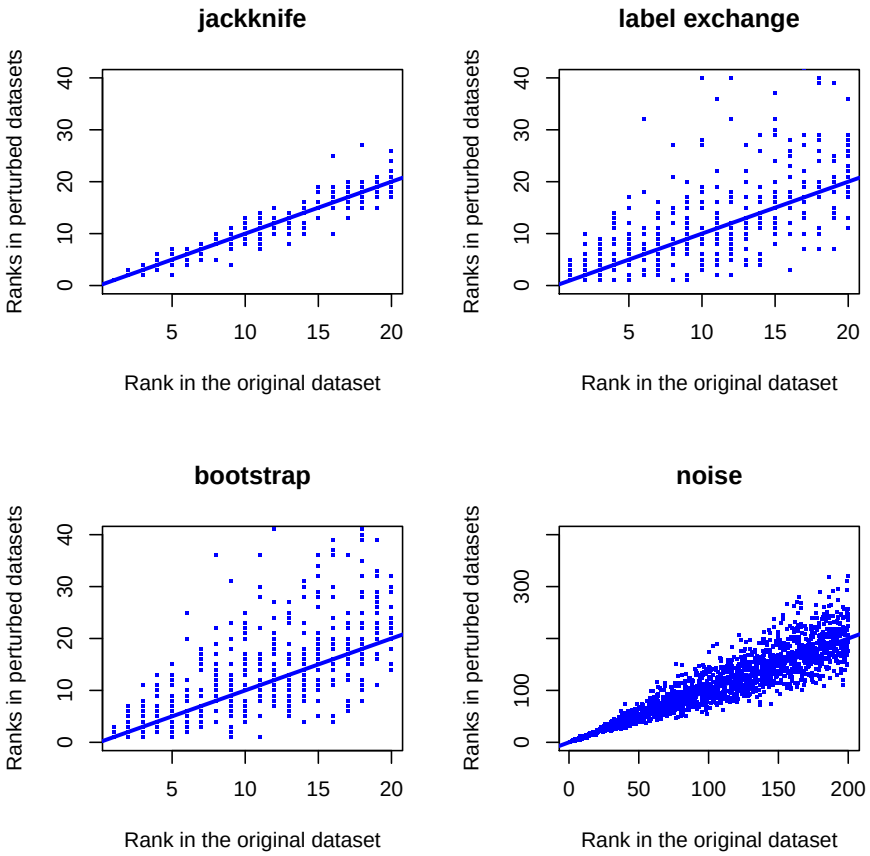


Figure 1: Scatterplots of rankings from altered datasets vs. rankings from the original dataset.

2.4 Stability measures

Alternative to visual methods, one can compute of the stability measures tabulated in Table **??**. Let $\boldsymbol{\sigma}, \boldsymbol{\sigma}'$ be either two rankings $\boldsymbol{r}, \boldsymbol{r}'$ or two lists $\boldsymbol{l}, \boldsymbol{l}'$. A function s is called *pairwise stability measure* if

(i) $s(\boldsymbol{\sigma}, \boldsymbol{\sigma}') = s(\boldsymbol{\sigma}', \boldsymbol{\sigma})$,

(ii) $s(\boldsymbol{\sigma}, \boldsymbol{\sigma}') \leq s(\boldsymbol{\sigma}, \boldsymbol{\sigma}) = s(\boldsymbol{\sigma}', \boldsymbol{\sigma}') = 1$.

In the current version of `GeneSelector`, there are two groups of pairwise stability measures: the first group is set-based, counting/summing up overlaps of lists, while the second one computes distances. Pairwise stability measures are particularly appropriate in the presence of a reference list/ranking. In the example given in the previous subsection, the

Name	**Definition**	**Reference**
Intersection count †	$s_\cap(\boldsymbol{l},\boldsymbol{l}') = s_\cap(\boldsymbol{l}_{[k]},\boldsymbol{l}'_{[k]}) = \frac{\sum_{1\le m,m'\le k} I(l_m=l'_{m'})}{k}, \quad k=1,\ldots,p.$	**?**
Overlap score †	$s_{O\cap}(\boldsymbol{l},\boldsymbol{l}') = \frac{\sum_{k=1}^p w_k s_\cap(\boldsymbol{l}_{[k]},\boldsymbol{l}'_{[k]})}{\sum_{k=1}^p w_k}$	**?**
ℓ^1 ‡	$s_{\ell^1}(\boldsymbol{r},\boldsymbol{r}') = 1 \; - \; \frac{\sum_{j=1}^p w_j \lvert r_j - r'_j\rvert}{\sum_{j=1}^p w_{(j)}\lvert j-(p-j+1)\rvert}$	**?**
ℓ^2 ‡	$s_{\ell^2}(\boldsymbol{r},\boldsymbol{r}') = 1 \; - \; \frac{\sum_{j=1}^p w_j (r_j - r'_j)^2}{\sum_{j=1}^p w_{(j)}\lvert j-(p-j+1)\rvert}$	**?**
Spearman's ρ ‡	$\frac{\sum_{j=1}^p w_j (r_j-(p+1)/2)(r'_j-(p+1)/2)}{\left(\sum_{j=1}^p r_j^2\right)^{1/2}\left(\sum_{j=1}^p r_j'^2\right)^{1/2}}$	**?**
Kendall's τ ‡	$s_\tau(\boldsymbol{r},\boldsymbol{r}') = 1 \; - \; \frac{\sum_{1\le j<m\le p} w_j\, w_m\, I([(r_j-r_m)(r'_j-r'_m)]<0)}{\sum_{1\le j<m\le p} w_j\, w_m}$	**?**
Union count Δ	$s_\cup(\boldsymbol{l}_{1\,[k]},\ldots,\boldsymbol{l}_{B\,[k]}) = 1 - \frac{\lvert U_{[k]}\rvert - k}{\min\{Bk,p\}-k}$	**?**
Union score Δ	$s_{O\cup}(\boldsymbol{l}_1,\ldots,\boldsymbol{l}_B) = \frac{\sum_{k=1}^p w_k s_\cup(\boldsymbol{l}_{1\,[k]},\ldots,\boldsymbol{l}_{B\,[k]})}{\sum_{k=1}^p w_k}$	

Table 2: Overview of the stability measures in `GeneSelector`. Notations: $\boldsymbol{l}_{[k]} = (l_m,\ 1 \le m \le k)$ denotes the top-k list of $\boldsymbol{l}$; the w_j's are (fixed) weights - the subscript in the brackets indicate ordering, i.e. $w_{(1)} \le \ldots \le w_{(p)}$; $|U_{[k]}|$ denotes the size of the union of all top k-lists to be compared. Legend: † - implemented in `GetStabilityOverlap`; ‡ - implemented in `GetStabilityDistance`; Δ - implemented in `GetStabilityUnion`.

natural choice for the reference is the ranking obtained with the original dataset. If one wants to compute a stability indicator for several lists without a reference, e.g. when comparing the output of different ranking criteria, we introduce the following notion. Let $\boldsymbol{\sigma}_b, \quad b = 1,\ldots,B$ be a sequence of rankings or lists. A function s is called *multi-input stability measure* if

(i) $s(\boldsymbol{\sigma}_1,\ldots,\boldsymbol{\sigma}_B) = s(\boldsymbol{\sigma}_{\pi_1},\ldots,\boldsymbol{\sigma}_{\pi_B})$ for any permutation $\boldsymbol{\pi}$ of $\{1,\ldots,B\}$,

(ii) $s(\boldsymbol{\sigma}_1,\ldots,\boldsymbol{\sigma}_B) \le s(\boldsymbol{\sigma}_1,\ldots,\boldsymbol{\sigma}_1) = \ldots = s(\boldsymbol{\sigma}_B,\ldots,\boldsymbol{\sigma}_B) = 1.$

As shown in Table **??**, an additional component of stability measures is a weighting scheme which penalizes variability at the top of list more severely than at the bottom, since only the top is of practical relevance.

As illustration, we apply `GetStabilityOverlap` to the rankings obtained after swapping class labels, which seems to perturb considerably the original ranking, as indicated by Figure **??**. Concerning the sequence of weights, we choose $w_m = 1/m, \quad m = 1,\ldots,p$, which is realized by using the option `decay = "linear"`.

```
> stab_ex1r_ordT <- GetStabilityOverlap(ex1r_ordT, scheme = "original",
```

```
+ decay="linear")
> show(stab_ex1r_ordT)

Stability measure: intersection count and overlap score,
scheme: original,
weighting: linear weight decay.

>
```

`GetStabilityOverlap` computes both normalized intersection counts and overlap scores when truncating at list position k, $k = 1, \ldots, p$. Evaluating these scores for position $k = 10$, we use the lines:

```
> summary(stab_ex1r_ordT, measure = "intersection", display = "all", position = 10)

intersection fractions (with respect to reference data set):
 iter.1  iter.2  iter.3  iter.4  iter.5  iter.6  iter.7  iter.8  iter.9 iter.10
    0.8     0.9     0.6     0.7     0.9     0.9     0.8     0.8     0.6     0.8
iter.11 iter.12 iter.13 iter.14 iter.15 iter.16 iter.17 iter.18 iter.19 iter.20
    0.8     0.7     0.9     0.8     0.7     0.7     0.8     0.9     0.9     0.8
expected score in the case of no-information:  0.005

> summary(stab_ex1r_ordT, measure = "overlapscore", display = "all", position = 10)

overlap scores (with respect to reference data set):
 iter.1  iter.2  iter.3  iter.4  iter.5  iter.6  iter.7  iter.8  iter.9 iter.10
  0.537   0.874   0.660   0.695   0.494   0.776   0.490   0.430   0.481   0.851
iter.11 iter.12 iter.13 iter.14 iter.15 iter.16 iter.17 iter.18 iter.19 iter.20
  0.769   0.606   0.716   0.597   0.647   0.763   0.528   0.703   0.608   0.672
expected score in the case of no-information:  0.001707086

>
```

The output shows that the overlap between reference- and alternative top-ten lists ranges from 60 to 90 percent. Overlap score and intersection count disagree visibly, which is due to the fact that the overlap score is computed with weights. Though **??** suggests some discrepancy between reference- and alternative lists, the output shows that the fraction of accordance is much larger than the expectation in the no-information case, i.e. in the case of mutually unrelated lists. To have a look at how the two scores vary with increasing list position (on average), we invoke the predefined `plot(...)` routine:

Next, let us investigate which sample is most influential in the sense that its removal perturbs the reference ranking most. For this purpose, we apply `GetStabilityDistance` with the option `measure = "spearman"` to the jackknif-ed rankings.

```
> stab_loo_ordT <- GetStabilityDistance(ex1r_ordT, scheme = "original", measure
+ = "spearman", decay="linear")
> show(stab_loo_ordT)
```

```
> plot(stab_ex1r_ordT, frac = 1, mode = "mean")
```

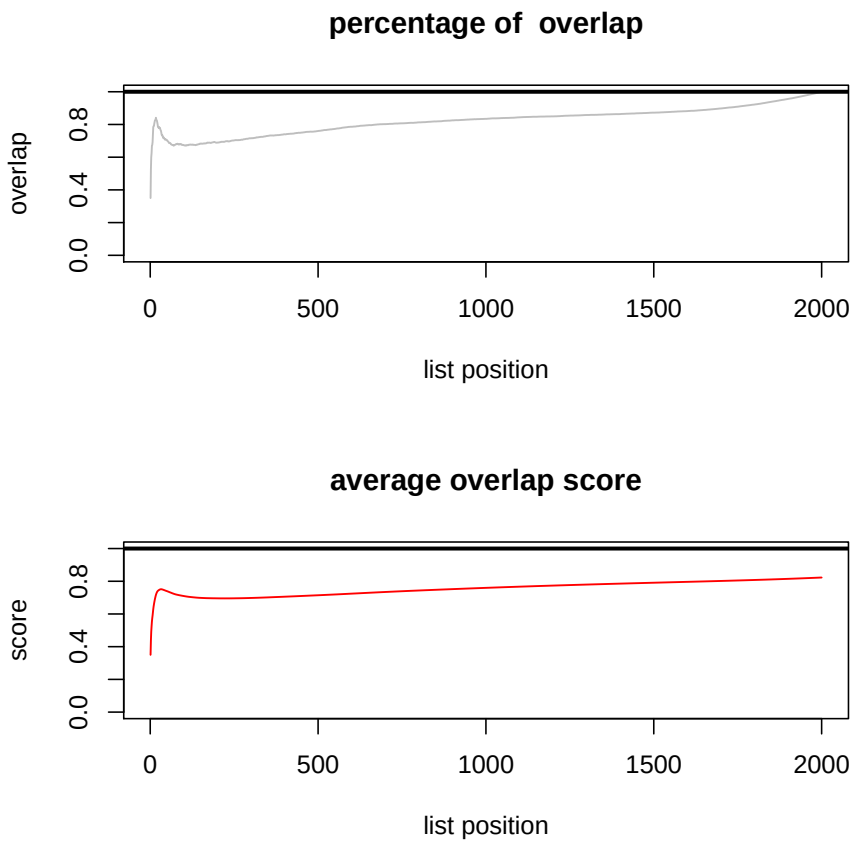


Figure 2: Visualization of intersection count and overlap score.

```
Stability measure: spearman's rank correlation,
scheme: original,
weighting: linear weight decay.

> summary(stab_loo_ordT, display = "all")

stability scores (with respect to reference data set):
 iter.1  iter.2  iter.3  iter.4  iter.5  iter.6  iter.7  iter.8  iter.9 iter.10
  0.351   0.749   0.745   0.568   0.157   0.489   0.585   0.420   0.393   0.683
iter.11 iter.12 iter.13 iter.14 iter.15 iter.16 iter.17 iter.18 iter.19 iter.20
  0.402   0.460   0.654   0.674   0.720   0.446   0.230   0.698   0.509   0.564
expected score in the case of no-information:  0

>
```

From the output we conclude that the fifth sample is by far the most influential one.

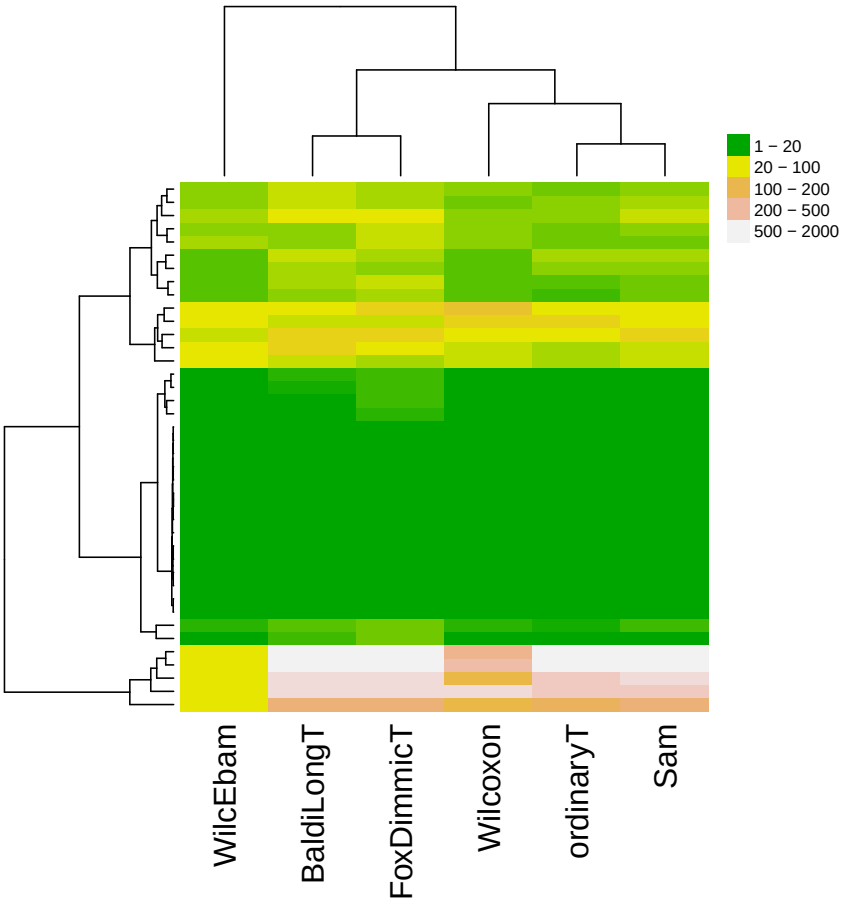
2.5 Aggregating multiple ranking criteria

In addition to the ordinary t-statistic, we compute five additional rankings (cf. Table **??**):

```
> BaldiLongT <- RankingBaldiLong(x, y, type="unpaired")
> FoxDimmicT <- RankingFoxDimmic(x, y, type="unpaired")
> sam <- RankingSam(x, y, type="unpaired")
> wilcox <- RankingWilcoxon(x, y, type="unpaired")
> wilcoxeb <- RankingWilcEbam(x, y, type="unpaired")
```

Again, we first assess variability visually. The method `HeatmapRankings` produces a heatmap from all obtained rankings, clustering genes and criteria simultaneously. We restrict to our attention to the first forty, differentially expressed genes (`ind = 1:40`).

```
> Merged <- MergeMethods(list(ordT, BaldiLongT, FoxDimmicT, sam, wilcox, wilcoxeb))
> HeatmapRankings(Merged, ind=1:40)
>
```



To cope with the multiplicity problem, we exploit the functionalities for rank aggregation in the `GeneSelector` package. A simple approach would just take the average of all observed ranks, which is, among other things, implemented in the method `AggregateSimple`. As a more sophisticated approach, we use the Markov chain model propagated in **?** and

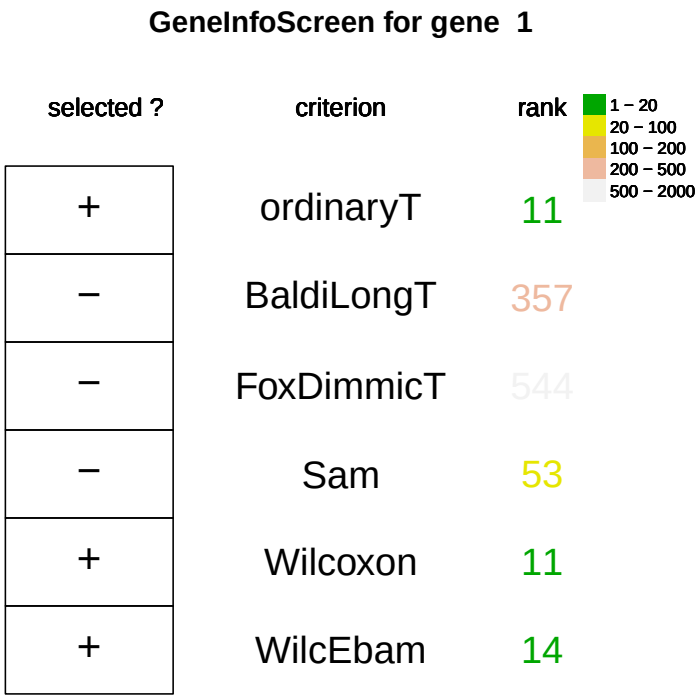
implemented in the method `AggregateMC`. Lastly, we use the `GeneSelector` function, which aims at finding genes falling consistently, i.e. for all ranking criteria, below a pre-specified threshold, here chosen as 50.

```
> AggMean <- AggregateSimple(Merged, measure = "mean")
> AggMC <- AggregateMC(Merged, type = "MCT", maxrank = 100)
> GeneSel <- GeneSelector(list(ordT, BaldiLongT, FoxDimmicT, sam, wilcox,
+ wilcoxeb), threshold="user", maxrank=50)
> show(GeneSel)

GeneSelector run with gene rankings from the following statistics:
ordinaryT
BaldiLongT
FoxDimmicT
Sam
Wilcoxon
WilcEbam
Number of genes below threshold rank 50 in all statistics:29

> sel <- sum(slot(GeneSel, "selected"))
> cbind(mean = toplist(AggMean, top = sel, show = F), MC = toplist(AggMC, top
+ = sel, show = F), GeneSelector = toplist(GeneSel, top = sel, show = F)[,1])

   index index GeneSelector
1      5     5            4
2      4     4           11
3     11    11            2
4      2     2           26
5      9     9            5
6     26    28            9
7     28    26           38
8     38    38           28
9      7     7           40
10    29    29           12
11    14    14            7
12    30    30           29
13   820    40           30
14    40   820           14
15    12    12           33
16    33    33         1799
17  1799  1799          820
18  1146    23         1146
19  1633  1146         1633
20  1677  1633         1551
21  1258  1258          724
22  1551     1         1258
23   937   937         1199
24  1715  1677          100
```

```
25   724  1551          1370
26  1370  1715           937
27   100   724          1715
28  1641     6          1267
29  1199  1370           476

>
```

Here, we have first determined the number of genes passing the `GeneSelector` filter. In total, 29 genes manage to fall below rank 50 in all six rankings. Although the `GeneSelector` attempts to minimize the number of false positives, one still ends up with 14 false positive genes among the 29 selected ones. In this regard, the Markov chain approach is superior, because it selects only 11 false positive ones. Simple averaging seems to perform slightly worse, putting the false positive gene 820 at position 13. In contrast, the first false positive gene of the `GeneSelector` occurs at position 16. A nice feature we want to present at the end is the `plot` routine for the class `GeneSelector`. It allows one to obtain a detailed gene-specific overview:

```
> plot(GeneSel, which = 1)
>
```



Interestingly, for the first, differentially expressed gene, simple approaches such as the ordinary t- and the Wilcoxon statistic perform well, while the more sophisticated statistics, which depend on hyperparameters, fail to detect differential expression.