# The Iterative Signature Algorithm for Gene Expression data

Gábor Csárdi

August 8, 2009

## Contents

# 1 Introduction

The Iterative Signature Algorithm (ISA) is biclustering method. The input of a biclustering method a matrix and its output is a set of biclusters that fulfill some criteria; a bicluster is a block of the potentially reordered input matrix. Most commonly, this algorithm is used on microarray expression data, to find gene sets that are coexpressed across a subset of the original samples. In the original ISA paper the biclusters are called transription modules (TM), we will often refer them under this name in the following.

# 2 Preparing the data

## 2.1 Loading the data

First, we load the required packages and the data to analyze. ISA is implemented in the `eisa` and `isa2` packages, see Sectionsec:isapackages for a more elaborated summary about the two packages. It is enough to load the `eisa` package, `isa2` and other required packages are loaded automatically. `eisa`

```
> library(eisa)
```

In this tutorial we will use the data in the `ALL` package.

```
> library(ALL)
> library(hgu95av2.db)
> library(affy)
> data(ALL)
```

This is a data set from a clinical trial in acute lymphoblastic leukemia and it contains 128 samples altogether.

# 3 Simple ISA runs

The simplest way to run ISA is to choose the two threshold parameters and then call the `isa` function on the `ExpressionSet` object:

```
> thr.gene <- 2.7
> thr.cond <- 1.4
> set.seed(1)
> modules <- ISA(ALL, thr.gene = thr.gene, thr.cond = thr.cond)
```

This first applies a non-specific filter to the data set filters the data set and then runs ISA from 100 random seeds (the default). See Section 10 if the default parameters are not appropriate for you and need more control.

# 4 Inspect the result

The `isa` function returns an `ISAModules` object. By typing in its name we can get a brief summary of the results:

```
> modules
```

```
An ISAModules instance.
  Number of modules: 8
  Number of features: 3522
  Number of samples: 128
  Gene threshold(s): 2.7
  Conditions threshold(s): 1.4
```

There are various other `ISAModules` methods that help to access the modules themselves and the ISA parameters that were used for the run.
Calling `length` on `modules` returns the number of ISA modules in the set, `dim` gives the dimension of the input expression matrix: the number of features and the number of samples.

```
> length(modules)
```

```
[1] 8
```

```
> dim(modules)
```

```
[1] 3522  128
```

Functions `featureNames` and `sampleNames` return the names of the features and samples, just like the functions with the same name for an `ExpressionSet`

```
> featureNames(modules)[1:5]

[1] "907_at"   "35430_at" "374_f_at" "33886_at" "34332_at"

> sampleNames(modules)[1:5]

[1] "01005" "01010" "03002" "04006" "04007"
```

The `getNoFeatures` function returns a numeric vector, the number of features (probesets in our case) in each module. Similarly, `getNoSamples` returns a numeric vector, the number of samples in each module. Similarly, `pData` returns the phenotype data of the expression set as a data frame. The `getOrganism` function returns the scientific name of the organism, `annotation` the name of the chip. For the former the appropriate annotation must be installed.

```
> getNoFeatures(modules)

[1] 40 30 26 63 23 24 45 36

> getNoSamples(modules)

[1] 21 18 22 11 21 20 13 22

> colnames(pData(modules))

 [1] "cod"            "diagnosis"      "sex"
 [4] "age"            "BT"             "remission"
 [7] "CR"             "date.cr"        "t(4;11)"
[10] "t(9;22)"        "cyto.normal"    "citog"
[13] "mol.biol"       "fusion protein" "mdr"
[16] "kinet"          "ccr"            "relapse"
[19] "transplant"     "f.u"            "date last seen"

> getOrganism(modules)

[1] "Homo sapiens"

> annotation(modules)

[1] "hgu95av2"
```

The double bracket indexing operator (‘`[[`’) can be used to select some modules from the complete set, the result is another, smaller `ISAModules` object. The following selects the first five modules.

```
> modules[[1:5]]

An ISAModules instance.
  Number of modules: 5
  Number of features: 3522
  Number of samples: 128
  Gene threshold(s): 2.7
  Conditions threshold(s): 1.4
```

The single bracket indexing operator can be used to restrict an ISAModules object to a subset of features and/or samples. E.g. selecting all features that map to a gene on chromosome 1 can be done with

```
> chr <- get(paste(annotation(modules), sep = "",
+     "CHR"))
> entrez <- sapply(mget(featureNames(modules), chr),
+     function(x) "1" %in% x)
> modules[entrez, ]
```

```
An ISAModules instance.
  Number of modules: 8
  Number of features: 356
  Number of samples: 128
  Gene threshold(s): 2.7
  Conditions threshold(s): 1.4
```

Similarly, selecting all B-cell samples can be performed with

```
> modules[, grep("^B", pData(modules)$BT)]
```

```
An ISAModules instance.
  Number of modules: 8
  Number of features: 3522
  Number of samples: 95
  Gene threshold(s): 2.7
  Conditions threshold(s): 1.4
```

getFeatureNames lists the probes (more precisely, the feature names coming from the ExpressionSet object) in the modules. It returns a list, here we just print the first entry in the list.

```
> getFeatureNames(modules)[[1]]
```

```
 [1] "34332_at"   "39829_at"   "41348_at"   "40147_at"
 [5] "34033_s_at" "39930_at"   "38067_at"   "41819_at"
 [9] "40688_at"   "38242_at"   "37497_at"   "37344_at"
[13] "38833_at"   "38096_f_at" "37039_at"   "41723_s_at"
[17] "39248_at"   "172_at"     "2047_s_at"  "33238_at"
[21] "32184_at"   "38147_at"   "38051_at"   "38750_at"
[25] "33039_at"   "33705_at"   "32794_g_at" "33121_g_at"
[29] "33369_at"   "39709_at"   "35839_at"   "32649_at"
[33] "633_s_at"   "37759_at"   "33514_at"   "38319_at"
[37] "39226_at"   "1096_g_at"  "35016_at"   "37988_at"
```

The getSampleNames function does the same for the samples. Again, the sample names are taken from the ExpressionSet object that was passed to isa:

```
> getSampleNames(modules)[[1]]
```

```
 [1] "01003" "01007" "04018" "09002" "12008" "15006" "16002"
 [8] "16007" "19002" "19017" "24006" "26009" "28008" "28009"
[15] "37001" "43006" "44001" "49004" "56007" "64005" "65003"
```

Remember, that the ISA biclustering is not binary, every feature (and similarly, every sample) has a score between -1 and 1; the further the score is from zero the stronger the association between the feature (or sample) and the module. If two features both have scores with the same sign, then they are correlated, if the sign of their scores are opposite, then they are anti-correlated. You can query the scores of the features with the `getFeatureScores` function, and similarly, the `getSampleScores` function queries the sample scores. You can supply the modules you want to query as an optional argument:

```
> getFeatureScores(modules, 3)
```

```
[[1]]
  41233_at   33849_at   40220_at   32833_at    1891_at
-0.9059774 -0.9333909 -0.8114228 -0.8504589 -0.8194976
   1292_at     529_at   40375_at   39715_at   36669_at
-0.7523284 -0.7877418 -0.8549541  0.7601123 -0.8510850
  36711_at   39420_at   37187_at   280_g_at 32901_s_at
-0.8677667 -0.7648151 -0.8347887 -0.8387569 -0.9068543
35372_r_at   33146_at 39822_s_at     287_at   37623_at
-0.8293233 -0.9089686 -0.8596327 -0.9134572 -0.8340685
34304_s_at   36674_at   36979_at   40448_at   40790_at
-0.9251146 -0.8023085 -1.0000000 -0.8027919 -0.8686802
   1237_at
-0.9964601
```

```
> getSampleScores(modules, 3)
```

```
[[1]]
     03002      04007      04008      04016      08001
-0.9609289 -0.6687051 -0.6816842 -1.0000000 -0.7413363
     12007      12026      15001      24008      27004
-0.7180883 -0.6604882 -0.8404454 -0.8221799  0.5298319
     28003      28019      28021      28023      28035
 0.4267097  0.5812084  0.5665996  0.4718876  0.4924001
     28037      28044      28047      43012      19017
 0.3790564  0.4565135  0.4460004  0.3864406 -0.9386168
     28008      64005
 0.4902438 -0.7124934
```

You can also query the scores in a matrix form, that is probably better if you need many or all of them at the same time. The `getFeatureMatrix` and `getSampleMatrix` functions are defined for this:

```
> dim(getFeatureMatrix(modules))
```

```
[1] 3522    8
```

```
> dim(getSampleMatrix(modules))
```

```
[1] 128   8
```

Objects from the `ISAModules` class store various information about the ISA run and the convergence of the seeds. Information associated with the individual seeds can be queried with the `seedData` function, it returns a data frame, with as many rows as the number of seeds and various seed-level information, e.g. the number of iterations required for the seed to converge. See the manual page of `isa` for details.

```
> seedData(modules)
```

```
   iterations oscillation thr.row thr.col freq      rob
1          18           0     2.7     1.4    1 22.95550
2           8           0     2.7     1.4    1 24.32006
3          24           0     2.7     1.4    1 23.77963
11          6           0     2.7     1.4    1 26.22564
61          5           0     2.7     1.4    1 22.47128
62          9           0     2.7     1.4    1 22.06247
63         14           0     2.7     1.4    1 24.01772
99         10           0     2.7     1.4    1 22.82995
   rob.limit
1   21.98084
2   21.98084
3   21.98084
11  21.98084
61  21.98084
62  21.98084
63  21.98084
99  21.98084
```

The `runData` function returns additional information about the ISA run, seed the `isa` manual page for details.

```
> runData(modules)
```

```
$direction
[1] "updown" "updown"

$eps
[1] 1e-04

$cor.limit
[1] 0.99
```

```
$maxiter
[1] 100

$N
[1] 100

$convergence
[1] "cor"

$prenormalize
[1] FALSE

$hasNA
[1] FALSE

$unique
[1] TRUE

$oscillation
[1] FALSE

$rob.perms
[1] 1

$annotation
[1] "hgu95av2"

$organism
[1] "Homo sapiens"
```

# 5 Enrichment calculations

The `eisa` package provides some functions to perform enrichment tests for the genes in the various modules against various databases. These tests are usually simplified and less sophisticated versions than the ones in the `Category`, `GOstats` or `topGO` packages, but they are much faster and this is important if we need to perform them for many modules.

## 5.1 Gene Ontology

To perform enrichment analysis against the Gene Ontology database, all you have to do is to supply your `ISAModules` object to the `ISA.GO` function.

```
> GO <- ISA.GO(modules)
```

```
 -- Extracting Entrez genes
 -- Extracting Entrez Universe
 -- Doing BP test
 -- Doing CC test
 -- Doing MF test
```

The `ISA.GO` requires an annotation package that maps the Entrez identifiers of the studied organism to GO terms. For human, the `org.Hs.eg.db` package is required, for the mouse the `org.Mm.eg.db` package, etc.
The `GO` object is a list with three elements, these correspond to the ontologies: biological function, cellular component and molecular function, in this order.

```
> GO

[[1]]
Gene to GO List BP  test for over-representation
3491 GO List BP ids tested (0-18 have p < 0.001)
Selected gene set sizes: 20-51
     Gene universe size: 2979
     Annotation package: hgu95av2

[[2]]
Gene to GO List CC  test for over-representation
711 GO List CC ids tested (0-11 have p < 0.001)
Selected gene set sizes: 21-57
     Gene universe size: 3111
     Annotation package: hgu95av2

[[3]]
Gene to GO List MF  test for over-representation
1063 GO List MF ids tested (0-6 have p < 0.001)
Selected gene set sizes: 19-56
     Gene universe size: 3080
     Annotation package: hgu95av2
```

We can see the number of categories tested, this is different for each ontology, as they have different number of terms. The gene universe size is also different, because it contains only genes that have at least one annotation in the given category.
For extracting the results themselves, the `summary` function can be used, this converts them to a simple data frame. The $p$-value limit can be supplied to `summary`. Note, that since `ISA.GO` calculates enrichment for many gene sets (i.e. for all biclusters), `summary` returns a list of data frames, one for each bicluster. A table for the first module:

```
> summary(GO[[1]], p = 0.001)[[2]][, -6]
```

```
[1] Pvalue    OddsRatio ExpCount  Count     Size
<0 rows> (or 0-length row.names)
```

We omitted the sixth column of the result, because it is very wide and would look bad in this vignette. This column is called `drive` and lists the Entrez IDs of the genes that are in the intersection of the bicluster and the GO category; or in other words, the genes that drive the enrichment. These genes can also be obtained with the `geneIdsByCategory` function. The following returns the genes in the second module and the third GO BP category. (The GO categories are ordered according to the enrichment $p$-values, just like in the output of `summary`.)

```
> geneIdsByCategory(GO[[1]])[[2]][[3]]

[1] "10589" "118"   "166"   "5518"  "598"   "7376"  "811"
```

You can use the `GO.db` package to obtain more information about the enriched GO categories.

```
> sigCategories(GO[[1]])[[2]]

character(0)

> library(GO.db)
> mget(sigCategories(GO[[1]])[[2]], GOTERM)

list()
```

In addition, the following functions are implemented to work on the objects returned by `ISA.GO`: `htmlReport`, `pvalues`, `geneCounts`, `oddsRatios`, `expectedCounts`, `universeCounts`, `universeMappedCount`, `geneMappedCount`, `geneIdUniverse`. These functions do essentially the same as they counterparts for `GOHyperGResult` objects, see the documentation of the `GOstats` package. The only difference is, that since here we are testing a list of gene sets (=biclusters), they calculate the results for all gene sets and return a list.

### 5.1.1 Multiple testing correction

By default, the `ISA.GO` function performs multiple testing correction using the Holm method, this can be changed via the `correction` and `correction.method` arguments. See the manual page of the `p.adjust` function for the possible multiple testing correction schemes.

## 5.2 KEGG Pathway Database

Enrichment calculation against the KEGG pathway goes essentially the same way as for the Gene Ontology, this time we use the `ISA.KEGG` function:

```
> KEGG <- ISA.KEGG(modules)
```

```
 -- Extracting Entrez genes
 -- Extracting Entrez Universe
 -- Doing test
```

```
> KEGG
```

```
Gene to KEGG List  test for over-representation
190 KEGG List ids tested (0-3 have p < 0.001)
Selected gene set sizes: 7-26
     Gene universe size: 1279
     Annotation package: hgu95av2
```

```
> summary(KEGG)[[4]]
```

```
            Pvalue OddsRatio  ExpCount Count Size
00860 2.524798e-06  80.44872 0.1548084     5   11
                       drive
00860 212;645;1371;3145;7389
```

```
> library(KEGG.db)
> mget(sigCategories(KEGG)[[4]], KEGGPATHID2NAME)
```

```
$`00860`
[1] "Porphyrin and chlorophyll metabolism"
```

The functions mentioned in the Gene ontology enrichment section can be used for KEGG, as well.

## 5.3 Chromosomes

The `eisa` includes a simple way to check whether the genes in a bicluster are associated with a chromosome. See the `ISA.CHR` function:

```
> CHR <- ISA.CHR(modules)
```

```
 -- Extracting Entrez genes
 -- Extracting Entrez Universe
 -- Doing test
```

```
> summary(CHR, p = 0.05)[[2]][, -6]
```

```
       Pvalue OddsRatio ExpCount Count Size
19 0.01520199   5.31562 1.670455     7  196
```

The second bicluster has 7 genes on chromosome 19. Here is a list of the genes:

```
> unlist(mget(geneIdsByCategory(CHR)[[2]][[1]],
+     org.Hs.egSYMBOL))
```

```
      166        811       1455       4713       5518
    "AES"     "CALR"  "CSNK1G2"   "NDUFB7"  "PPP2R1A"
     7376      79090
  "NR1H2" "TRAPPC6A"
```

The functions mentioned in the Gene ontology enrichment section can be used for chromosome enrichment, as well.

### 5.4 Predicted $\mu$RNA targets from the TargetScan database

$\mu$RNAs are short RNA molecules that regulate gene expression. TargetScan is a data based of predicted target genes of $\mu$RNAs, for several organisms. There are two R packages that incorporate this database, one for human and another one for mouse, right now they can be downloaded from `http://www2.unil.ch/cbg/index.php?title=Building_BioConductor_Annotation_Packages`. The `targetscan.Hs.eg.db` package is for human, the `targetscan.Mm.eg.db` package is for mouse.
The enrichment calculation itself is basically the same as for GO and KEGG, but the `ISA.miRNA` function should be used:

```
> if (require(targetscan.Hs.eg.db)) {
+     miRNA <- ISA.miRNA(modules)
+     summary(miRNA, p = 0.1)[[7]]
+ }

 -- Extracting Entrez genes
 -- Extracting Entrez Universe
 -- Doing test
              Pvalue OddsRatio  ExpCount Count Size
miR-196ab 0.06239421   8.04074 0.7811806     5   55
                              drive
miR-196ab 115;2581;3202;3205;10643
```

## 6 Visualizing the results

Visualizing overlapping biclusters is a challenging task. We show simple methods that usually visualize a single bicluster at a time. For some of these we will use the `biclust` R package.

### 6.1 The `biclust` package

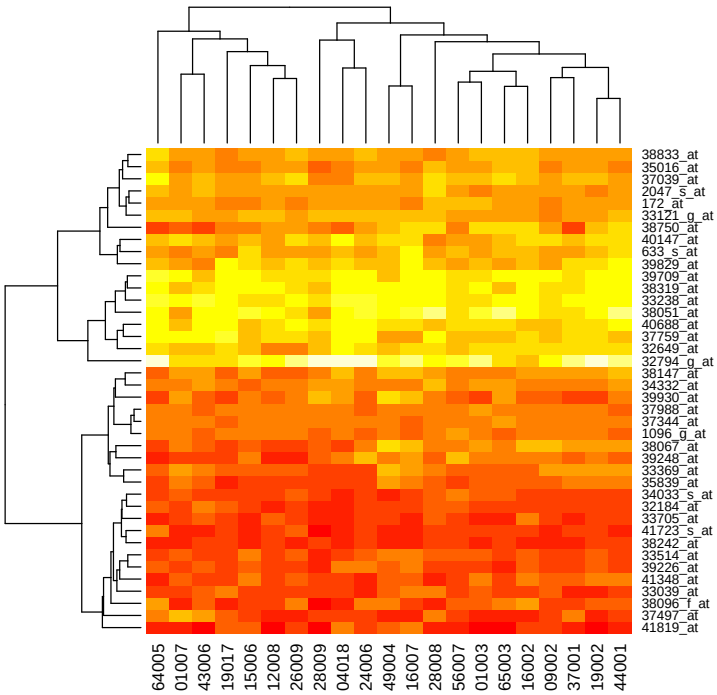
The `biclust` R package implements several biclustering algorithms in a unified framework. It uses the class `Biclust` to store a set of biclusters. The `ISA.biclust` function converts ISA modules to a `Biclust` object. This requires the binarization of the modules, i.e. the ISA scores are lost, they are converted to zeros and ones:

```
> library(biclust)
> Bc <- ISA.biclust(modules)
> Bc

An object of class Biclust

call:
        NULL

Number of Clusters found:  8

First  5  Cluster sizes:
                   BC 1 BC 2 BC 3 BC 4 BC 5
Number of Rows:    "40" "30" "26" "63" "23"
Number of Columns: "21" "18" "22" "11" "21"
```

## 6.2 Image plots

The easiest way to create a heatmap of a single module is to call the `ISA2heatmap` function. You need to specify which module you want to plot and also the `ExpressionSet` object that is being analyzed. Note that by default ISA normalizes the expression data before running the module detection; the raw, non-normalized values are plotted in the next example. If you want to plot the normalized values, then you need to do the normalization "by hand", before calling ISA, see Section 10.

```
> ISA2heatmap(modules, 1, ALL)
```




`ISA2heatmap` simply calls the `heatmap` function, and passes additional arguments to it. See the manual of `heatmap` for details.
You can also use the `biclust` package to create image plots, by calling the `drawHeatmap` function. The result is in Fig. 1.

```
> drawHeatmap(exprs(ALL[featureNames(modules), ]),
+     Bc, number = 1)
```

## 6.3 Profile plots

Profile plots visualize the difference between the genes (or samples) in the modules and the rest of the expression data. A profile plot contains a line plot for every single gene (or sample) and the genes that belong to the module have a different color, see Fig. 2.

```
> profile.plot(modules, 2, ALL, plot = "both")
```

The `profile.plot` function has several options to set the plot colors and styles, please see the manual for the details. This function was inspired by the `parallelCoordinates` function in the `biclust` package.

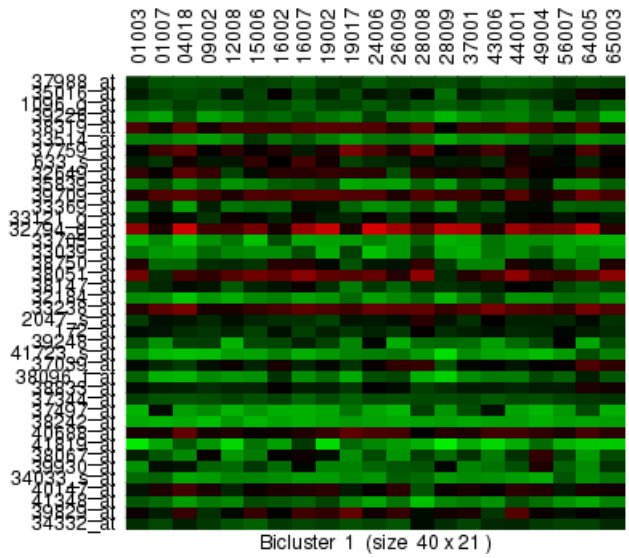



Figure 1: Heatmap of the first transcription module.

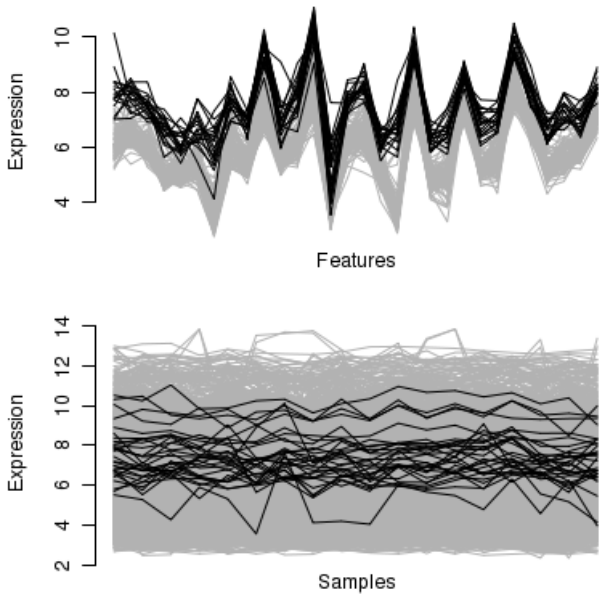



Figure 2: Profile plots for the second bicluster found by ISA.

## 6.4 Gene ontology tree plots

The GO database is organized in a hierarchical fashion, in a tree-like structure, where the broadest category sits in the root of the tree and broader categories are subdivided into more specific subcategories. But the GO is not exactly a tree, as the same category can be the subcategory of more than one broader categories: e.g. the "Golgi vesicle transport" category is part of both "vesicle-mediated transport" and "intracellular transport".
The `eisa` package provides some functions to plot parts of the GO graph that is related to a transcription module. The `gograph` function creates an object that is a representation of such a plot. Its input is a table with the GO categories to plot and their enrichment $p$-values. (Additional columns are silently ignored.) Here is how to use it on the previously calculated enrichment scores:

```
> goplot.2 <- gograph(summary(GO[[1]], p = 0.05)[[1]])
```

`goplot` uses the `igraph` package to create a graph with associated meta data:

```
> summary(goplot.2)
```

```
Vertices: 30
Edges: 29
Directed: TRUE
Graph attributes: width, height, layout.
Vertex attributes: color, name, plabel, label, desc, abbrv, definition, size, size2, shape,
Edge attributes: type, color, arrow.size.
```

```
> list.vertex.attributes(goplot.2)
```

```
 [1] "color"       "name"        "plabel"      "label"
 [5] "desc"        "abbrv"       "definition"  "size"
 [9] "size2"       "shape"       "label.color" "label.cex"
[13] "frame.color"
```

The `width` and `height` graph attributes contain the suggested width and height of the graph, if plotted to a bitmap device. (The graph attributes of an `igraph` graph can be queried with the '`$`' selector:

```
> goplot.2$width
```

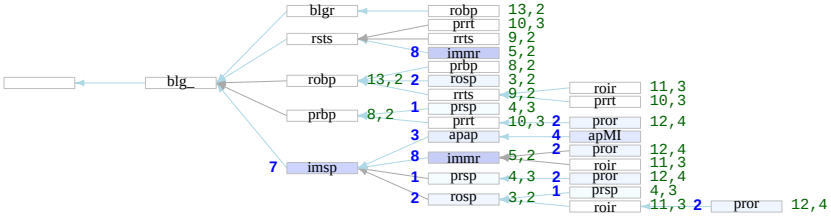
```
[1] 174.72
```

```
> goplot.2$height
```

```
[1] 83.03
```

Let's plot the graph, we can do this with the `gograph.plot` function.

```
> x11(width = 10, height = 10 * goplot.2$height/goplot.2$width)
> gograph.plot(goplot.2)
```




Because the GO is not really a tree, `gograph` "unfolds" it into a tree by including categories more than once, if needed. It also abbreviates the names of the GO categories to make them fit on the tree. The graph object contains the full name of the category as well. The full and abbreviated names can be listed by querying the appropriate vertex attributes of the graph. Here they are for the first five categories:

```
> V(goplot.2)$abbrv[1:5]

[1] "imsp" "apMI" "rosp" "prsp" "immr"

> lapply(V(goplot.2)$desc[1:5], strwrap)

[[1]]
[1] "immune system process"

[[2]]
[1] "antigen processing and presentation of peptide or"
[2] "polysaccharide antigen via MHC class II"

[[3]]
[1] "regulation of immune system process"

[[4]]
[1] "positive regulation of immune system process"

[[5]]
[1] "immune response"
```

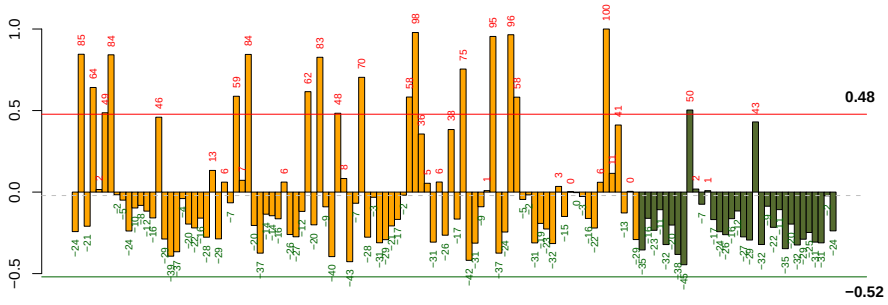
`gograph.plot` colors the categories according to the supplied enrichment $p$-values, the minus $\log_{10}$ $p$-value is also added to the plot, see the bold blue numbers.

## 6.5 Sample score plots

In many studies, especially the case-control ones, it is useful to plot the sample scores of a module. For example if the sample scores significantly differ for two groups of samples (e.g. cases versus controls), then the genes in the module can be used as discriminators between the two groups.

The `cond.plot` function can plot the sample scores, potentially including the scores before the ISA filtering. Let us plot the scores for the second module, the color code denotes B-cell vs. T-cell leukemia.

```
> col <- ifelse(grepl("^B", pData(modules)$BT),
+     "orange", "darkolivegreen")
> cond.plot(modules, 2, ALL, col = col)
```




It is clear that the 30 genes included in this transcription module can separate B-cell and T-cell leukemia samples.

### 6.6 Generating a HTML summary for the modules

The `autogen.table` function creates a summary

## 7 How ISA works

Before showing an actual ISA tool chain, a few words about how the algorithm works are in order.

### 7.1 ISA iteration

ISA works in an iterative way. For an $E(m \times n)$ input matrix it starts from a seed vector $r_0$, which is typically a sparse 0/1 vector of length $m$. The nonzero elements in the seed vector define a set of genes in $E$. Then the transposed of $E$, $E'$ is multiplied by $r_0$ and the result is thresholded.
The thresholding is an important step of the ISA, without thresholding ISA would be equivalent to a (not too effective) numerical singular value decomposition (SVD) algorithm. Currently thresholding is done by calculating the mean and standard deviation of the vector and keeping only elements that are further than a given number of standard deviations from the mean. Based on the *direction* parameter, this means keeping values that are significantly higher than the mean ("*up*"), or keeping the ones that are significantly lower than the mean ("*down*"); or keeping both ("*updown*").

The thresholded vector $c_0$ is the (sample) *signature* of $r_0$. Then the (gene) signature of $c_0$ is calculated, $E$ is multiplied by $c_0$ and then thresholded to get $r_1$.
This iteration is performed until it converges, i.e. $r_{i-1}$ and $r_i$ are *close*, and $c_{i-1}$ and $c_i$ are also close. The convergence criteria, i.e. what *close* means, is by default defined by high Pearson correlation.
It is very possible that the ISA finds the same module more than once; two or more seeds might converge to the same module. The function `ISA.unique` eliminates every module from the result of `ISA.iterate` that is very similar (in terms of Pearson correlation) to the one that was already found before.
It might be also apparent from the description of ISA, that the biclusters are soft, i.e. they might have an overlap in their genes, samples, or both. It is also possible that some genes and/or samples of the input matrix are not found to be part of any ISA biclusters. Depending on the stringency parameters in the thresholding (i.e. how far the values should be from the mean), it might even happen that ISA does not find any biclusters.

## 7.2 Parameters

The two main parameters of ISA are the two thresholds (one for the genes and one for the samples). They basically define the stringency of the modules. If the gene threshold is high, then the modules will have very similar genes. If it is mild, then modules will be bigger, with less similar genes than in the first case. The same applies to the sample threshold and the samples of the modules.

## 7.3 Random seeding and smart seeding

By default (i.e. if the `isa` function is used) the ISA is performed from random sparse starting seeds, generated by the `generate.seeds` function. This way the algorithm is completely unsupervised, but also stochastic: it might give different results for different runs.
It is possible to use non-random seeds as well. If you have some knowledge about the data or are interested in a particular subset of genes/samples, then you can feed in your seeds into the `ISA.iterate` function directly. In this case the algorithm is deterministic, for the same seed you will always get the same results. Using smart (i.e. non-random) seeds can be considered as a semi-supervised approach.

## 7.4 Normalization

Using in silico data we observed that ISA has the best performance if the input matrix is normalized (see `ISA.normalize`). The normalization produces two matrices: $E_r$ and $E_c$. $E_r$ is calculated by transposing $E$ and centering and scaling its genes (see the `scale` R function). $E_c$ is calculated by center-

ing and scaling the genes of $E$. $E_r$ is used to calculate the sample signature of genes and $E_c$ is used to calculate the signature of the samples.
It is possible to use another normalization. In this case the user is requested to supply the normalized input data in a named list, including the two matrices of appropriate dimensions to the `ISA.iterate` function.
The `Er` entry of the list will be used for calculating the signature of the genes, `Ec` will be used for the signature of the samples. If you want to use the same matrix in both steps, then supply it twice, the first one transposed.

### 7.5 Gene and sample scores

In addition to finding biclusters in the input matrix, the ISA also assigns scores to the genes and samples, separately for each module. The scores are between minus one and one and they are by definition zero for the genes/samples that are not included in the module. For the non-zero entries, the further the score of a gene/samples is from zero, the stronger the association between the gene/sample and the module. If the sign of two genes/samples are the same, then they are correlated, if they have opposite signs, then they are anti-correlated.

## 8 Bicluster coherence and robustness measures

### 8.1 Coherence

Madeira and Oliviera[**?**] define various coherence scores for biclusters, these measure how well the rows and or columns are correlated. It is possible to use these measures for ISA as well, after converting the output of ISA to a `biclust` object. Here are the measures for the first bicluster:

```
> constantVariance(exprs(ALL), Bc, number = 1)

[1] 4.34281

> additiveVariance(exprs(ALL), Bc, number = 1)

[1] 2.319414

> multiplicativeVariance(exprs(ALL), Bc, number = 1)

[1] 0.4149014

> signVariance(exprs(ALL), Bc, number = 1)

[1] 2.687224
```

You can use `sapply` to perform the calculation for many or all modules, e.g. for this data set 'constant variance' and 'additive variance' are not the same:

```
> cv <- sapply(seq_len(Bc@Number), function(x) constantVariance(exprs(ALL),
+     Bc, number = x))
> av <- sapply(seq_len(Bc@Number), function(x) additiveVariance(exprs(ALL),
+     Bc, number = x))
> cor(av, cv)
```

```
[1] 0.5271451
```

Please see the manual pages of these functions and the paper cited above for more details.

### 8.2 Robustness

The `eisa` package uses a measure that is related to coherence; it is called robustness. Robustness is a generalization of the singular value of a matrix. If there were no thresholding during the ISA iteration, then ISA would be equivalent to a numerical method for singular value decomposition and robustness would be indeed the same the principal singular value of the input matrix.
If the `isa` function was used to find the transcription modules, then the robustness measure is used to filter the results. This is done by first scrambling the input matrix and then running ISA on it. As ISA is an unsupervised algorithm it usually finds some (although less and smaller) modules even in such a scrambled data set. Then the robustness scores are calculated for the proper and the scrambled modules and only (proper) modules that have a higher score than the highest scrambled module are kept. The robustness scores are stored in the seed data during this process, so you can check them later:

```
> seedData(modules)$rob
```

```
[1] 22.95550 24.32006 23.77963 26.22564 22.47128 22.06247
[7] 24.01772 22.82995
```

## 9 The `isa2` and `eisa` packages

ISA and its companion functions for visualization, functional enrichment calculation, etc. are distributed in two separate packages, `isa2` and `eisa`.
`isa2` contains the implementation of ISA itself, and `eisa` specifically deals with supplying expression data to `isa2` and visualizing the results.

## 10 Finer control over ISA parameters

The `ISA` function takes care of all steps performed during a modular study, and for each step it uses parameters work reasonably well. In some cases, however, one wants to access these steps individually, to use custom parameters instead of the defaults.
In this section, we will still use the acute lymphoblastic leukemia gene expression data from the `ALL` package.

## 10.1 Non-specific filtering

The first step of the analysis typically involves non-specific filtering of the probesets. The aim is to eliminate the probesets that do not show variation across the samples, as they only contribute noise to the data.
By default (i.e. if the `ISA` function is called) this is performed using the `genefilter` package, and the default filter is based on the inter-quantile ratio of the probesets' expression values, a robust measure of variance. Other possible filtering techniques include using the AffyMetrix present/absent calls produced by the `mas5calls` function of the `affy` package, but this requires the raw data, so in this vignette we use a simple method based on variance and minimum expression value: only probesets that have a variance of at least `varLimit` and that have at least `kLimit` samples with expression values over `ALimit` are kept.

```
> varLimit <- 0.5
> kLimit <- 4
> ALimit <- 5
> flist <- filterfun(function(x) var(x) > varLimit,
+     kOverA(kLimit, ALimit))
> ALL.filt <- ALL[genefilter(ALL, flist), ]
```

The original expression set had 12625 features, the filtered one has only 1313.

## 10.2 Entrez Id matching

In this step we match the probesets to Entrez Id and remove the ones that don't map to any Entrez genes.

```
> ann <- annotation(ALL.filt)
> library(paste(ann, sep = ".", "db"), character.only = TRUE)
> ENTREZ <- get(paste(ann, sep = "", "ENTREZID"))
> EntrezIds <- mget(featureNames(ALL.filt), ENTREZ)
> keep <- sapply(EntrezIds, function(x) length(x) >=
+     1 && !is.na(x))
> ALL.filt.2 <- ALL.filt[keep, ]
```

To reduce ambiguity in the interpretation of the results, we might also want to keep only single probeset for each Entrez genes.

```
> vari <- apply(exprs(ALL.filt.2), 1, var)
> larg <- findLargest(featureNames(ALL.filt.2),
+     vari, data = annotation(ALL.filt.2))
> ALL.filt.3 <- ALL.filt.2[larg, ]
```

## 10.3 Normalizing the data

The ISA works best, if the expression matrix is scaled and centered. In fact, the two sub-steps of an ISA step require expression matrices that are normalized differently. The `ISA.normalize` function can be used to calculate the

normalized expression matrices; it returns an `ExpressionSet` object including three expression matrices: the original raw expression, the row-wise normalized and the column-wise normalized expression matrix.

```
> ALL.normed <- ISA.normalize(ALL.filt.3)
> ls(assayData(ALL.normed))
```

```
[1] "ec.exprs" "er.exprs" "exprs"
```

## 10.4 Generating starting seeds for the ISA

The ISA is an iterative algorithm that starts with a set of input seeds. An input seed is basically a set of probesets and the ISA stepwise refines this set by 1) including other probesets in the set that are coexpressed with the input probesets and 2) removing probesets from it that are not coexpressed with the rest of the input set.
The `generate.seeds` generates a set of random seeds (i.e. a set of random gene sets). See its documentation if you need to change the sparsity of the seeds.

```
> random.seeds <- generate.seeds(length = nrow(ALL.normed),
+     count = 100)
```

In addition to random seeds, it is possible to start the ISA iteration from "educated" seeds, i.e. gene sets the user is interested in, or a set of samples that are supposed to have coexpressed genes. We create another set of starting seeds here, based on the type of acute lymphoblastic leukemia: "B", "B1", "B2", "B3", "B4" or "T", "T1", "T2", "T3" and "T4".

```
> type <- as.character(pData(ALL.normed)$BT)
> ss1 <- ifelse(grepl("^B", type), -1, 1)
> ss2 <- ifelse(grepl("^B1", type), 1, 0)
> ss3 <- ifelse(grepl("^B2", type), 1, 0)
> ss4 <- ifelse(grepl("^B3", type), 1, 0)
> ss5 <- ifelse(grepl("^B4", type), 1, 0)
> ss6 <- ifelse(grepl("^T1", type), 1, 0)
> ss7 <- ifelse(grepl("^T2", type), 1, 0)
> ss8 <- ifelse(grepl("^T3", type), 1, 0)
> ss9 <- ifelse(grepl("^T4", type), 1, 0)
> smart.seeds <- cbind(ss1, ss2, ss3, ss4, ss5,
+     ss6, ss7, ss8, ss9)
```

The `ss1` seed includes all samples, but their sign is opposite for B-cell leukemia samples and T-cell samples. This way ISA is looking for sets of genes that are differently regulated in these two groups of samples. `ss2` contains only B1 type samples, so here we look for genes that are specific to this variant of the disease. The other seeds are similar, for the other subtypes.

## 10.5 Performing the ISA iteration

We perform the ISA iterations for our two sets of seeds separately.

```
> modules1 <- ISA.iterate(ALL.normed, row.seeds = random.seeds,
+     thr.row = 2, thr.col = 2)
> modules2 <- ISA.iterate(ALL.normed, col.seeds = smart.seeds,
+     thr.row = 2, thr.col = 2)
```

## 10.6 Dropping non-unique modules

`ISA.iterate` returns the same number of "modules" as the number of input seeds; these, however, do not always correspond to meaningful modules, the input seeds can converge to an all-zero vector, or occasionally they may not converge at all. It is also possible that two or more input seeds converge to the same module.
The `ISA.unique` function eliminates the all-zero or non-convergent input seeds and keeps only one instance of the duplicated ones.

```
> modules1.unique <- ISA.unique(ALL.normed, modules1)
> modules2.unique <- ISA.unique(ALL.normed, modules2)
```

48 modules were kept for the first set of seeds and 9 for the second set.

## 10.7 Dropping non-robust modules

The `ISA.filter.robust` function filters a set of modules by running ISA with the same parameters on a scrambled data set and then calculating a robustness score, both for the real modules and the ones from the scrambled data. The highest robustness score obtained from the scrambled data is used as a threshold to filter the real modules.

```
> modules1.robust <- ISA.filter.robust(ALL.normed,
+     modules1.unique)
> modules2.robust <- ISA.filter.robust(ALL.normed,
+     modules2.unique)
```

We still have 44 modules for the first set of seeds and 9 for the second set.

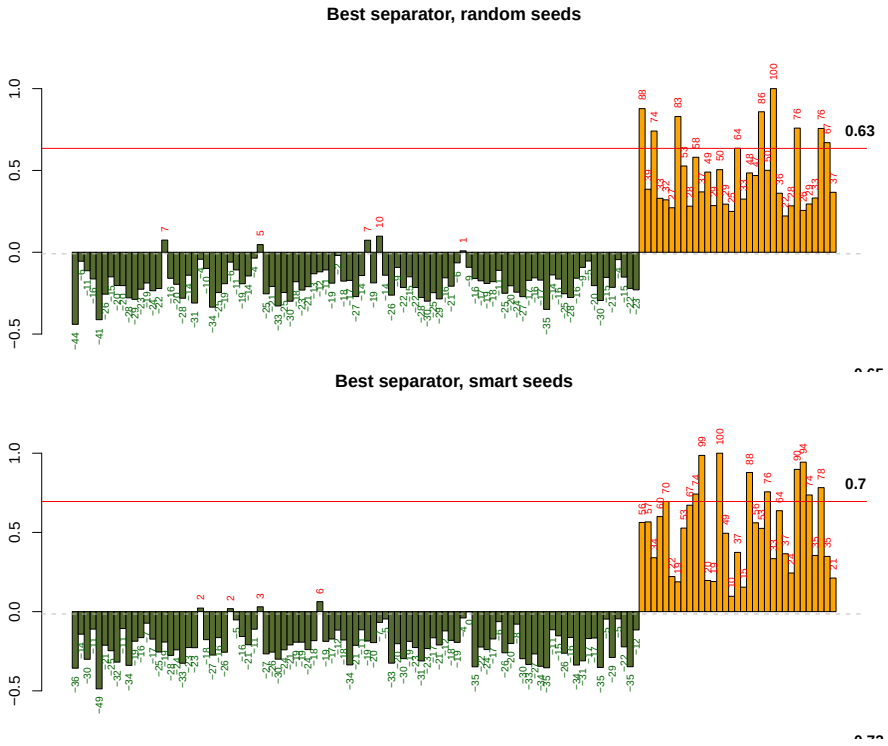
## 10.8 Differentially regulated modules

Now we check whether the ISA modules that we found can be used as classifiers for the different types of ALL. For this we use the sample scores of the modules, the score of a sample in a given module is the (weighted) average of the expression of the genes of the module in the sample in question.
Let's first check, whether any of the modules can distinguish between T-cell and B-cell ALL samples.

```
> scores1 <- getSampleMatrix(modules1.robust)
> tt1 <- colttests(scores1, as.factor(substr(type,
+     1, 1)))
> scores2 <- getSampleMatrix(modules2.robust)
> tt2 <- colttests(scores2, as.factor(substr(type,
+     1, 1)))
> sign1 <- which(p.adjust(tt1$p.value, "holm") <
+     0.05)
> sign2 <- which(p.adjust(tt2$p.value, "holm") <
+     0.05)
```

For the first set of samples 12 modules show significant difference for T-cell and B-cell samples, for the second (smart) set 5 of them.
Let's make some condition plots for the best separating modules from each set.

```
> color <- ifelse(grepl("T", type), "orange", "darkolivegreen")
> layout(cbind(1:2))
> cond.plot(modules1.robust, which.min(tt1$p.value),
+     ALL.normed, col = color, main = "Best separator, random seeds")
> cond.plot(modules2.robust, which.min(tt2$p.value),
+     ALL.normed, col = color, main = "Best separator, smart seeds")
```




Let's extract the modules that are good separators.

```
> modules1.TB <- modules1.robust[[sign1]]
> modules2.TB <- modules2.robust[[sign2]]
```

## 10.9 Enrichment calculations

We can check our separator modules against the Gene Ontology categories and the pathways in the KEGG database to find dysregulated GO categories and/or KEGG pathways.

```
> GO.dysreg1 <- ISA.GO(modules1.TB)

 -- Extracting Entrez genes
 -- Extracting Entrez Universe
 -- Doing BP test
 -- Doing CC test
 -- Doing MF test

> GO.dysreg2 <- ISA.GO(modules2.TB)

 -- Extracting Entrez genes
 -- Extracting Entrez Universe
 -- Doing BP test
 -- Doing CC test
 -- Doing MF test

> KEGG.dysreg1 <- ISA.KEGG(modules1.TB)

 -- Extracting Entrez genes
 -- Extracting Entrez Universe
 -- Doing test

> KEGG.dysreg2 <- ISA.KEGG(modules2.TB)

 -- Extracting Entrez genes
 -- Extracting Entrez Universe
 -- Doing test
```

Let's collect the significantly enriched GO categories and KEGG pathways.

```
> gocats <- unique(unlist(lapply(c(GO.dysreg1, GO.dysreg2),
+     function(x) unique(unlist(sigCategories(x))))))
> keggp <- unique(unlist(lapply(list(KEGG.dysreg1,
+     KEGG.dysreg2), function(x) unique(unlist(sigCategories(x))))))
> library(GO.db)
> library(KEGG.db)
> sapply(mget(gocats, GOTERM), Term)
```

```
                           GO:0019882
"antigen processing and presentation"
                           GO:0002376
              "immune system process"
                           GO:0006955
                    "immune response"
                           GO:0044425
                     "membrane part"
                           GO:0044459
               "plasma membrane part"
                           GO:0016021
               "integral to membrane"
                           GO:0005886
                    "plasma membrane"
                           GO:0031224
              "intrinsic to membrane"
                           GO:0016020
                           "membrane"
                           GO:0042611
                "MHC protein complex"
```

```
> mget(keggp, KEGGPATHID2NAME)
```

```
$`04640`
[1] "Hematopoietic cell lineage"

$`04660`
[1] "T cell receptor signaling pathway"
```

## 10.10 More separator modules

Of course we can search for modules that can separate the samples of the ALL subtypes as well, e.g. let's try to find some that differentiate between type B1 and other B types.

```
> keep <- grepl("^B[1234]", type)
> type.B <- ifelse(type[keep] == "B1", "B1", "Bx")
> scores.B1.1 <- scores1[keep, ]
> scores.B1.2 <- scores2[keep, ]
> tt.B1.1 <- colttests(scores.B1.1, as.factor(type.B))
> tt.B1.2 <- colttests(scores.B1.2, as.factor(type.B))
> min(p.adjust(na.omit(tt.B1.1$p.value)))
```

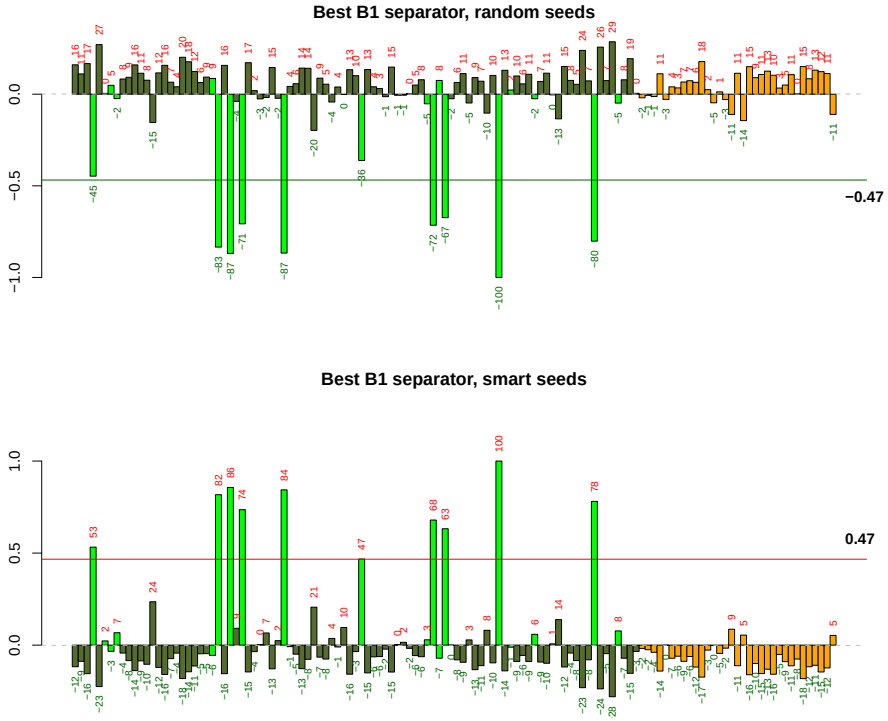
```
[1] 1.466285e-08
```

```
> min(p.adjust(na.omit(tt.B1.2$p.value)))
```

```
[1] 2.659643e-12
```

Both module sets seem to have some good separators, let us make some condition plots.

```
> color <- ifelse(type == "B1", "green", ifelse(grepl("^T",
+     type), "orange", "darkolivegreen"))
> layout(cbind(1:2))
> cond.plot(modules1.robust, which.min(tt.B1.1$p.value),
+     ALL.normed, col = color, main = "Best B1 separator, random seeds")
> cond.plot(modules2.robust, which.min(tt.B1.2$p.value),
+     ALL.normed, col = color, main = "Best B1 separator, smart seeds")
```




From the plot it seems that these two modules are essentially the same, one is the opposite of the others.

```
> B1.cor <- c(cor(scores1[, which.min(tt.B1.1$p.value)],
+     scores2[, which.min(tt.B1.2$p.value)]), cor(getFeatureMatrix(modules1.robust,
+     mods = which.min(tt.B1.1$p.value)), getFeatureMatrix(modules2.robust,
+     mods = which.min(tt.B1.2$p.value))))
> B1.cor

[1] -0.9506588 -0.9182716
```

Indeed, they are almost the same, their Pearson correlation is around -0.951 for the sample scores and -0.918 for the feature (=gene) scores.

# 11 Session information

The version number of R and packages loaded for generating this vignette were:

- R version 2.9.1 (2009-06-26), `i486-pc-linux-gnu`
- Locale: `LC_CTYPE=en_US.UTF-8;LC_NUMERIC=C;LC_TIME=en_US.UTF-8;LC_COLLATE=en_US.UTF-8;LC_MONETARY=C;LC_MESSAGES=en_US.UTF-8;LC_PAPER=en_US.UTF-8;LC_NAME=C;LC_ADDRESS=C;LC_TELEPHONE=C;LC_MEASUREMENT=en_US.UTF-8;LC_IDENTIFICATION=C`
- Base packages: base, datasets, graphics, grDevices, grid, methods, stats, utils
- Other packages: affy 1.22.1, ALL 1.4.4, annotate 1.22.0, AnnotationDbi 1.6.1, biclust 0.8.1, Biobase 2.4.1, Cairo 1.4-4, Category 2.10.0, colorspace 1.0-1, DBI 0.2-4, eisa 0.1, genefilter 1.24.2, GO.db 2.2.11, hgu95av2.db 2.2.12, igraph 0.6, isa2 0.1, KEGG.db 2.2.11, MASS 7.2-47, org.Hs.eg.db 2.2.11, RSQLite 0.7-1, targetscan.Hs.eg.db 5.0-1, vcd 1.2-4
- Loaded via a namespace (and not attached): affyio 1.12.0, graph 1.22.2, GSEABase 1.6.0, preprocessCore 1.6.0, RBGL 1.20.0, splines 2.9.1, survival 2.35-4, tools 2.9.1, XML 2.3-0, xtable 1.5-5