The Iterative Signature Algorithm for Gene Expression Data

Gábor Csárdi

October 18, 2010

Contents

1 Introduction

The Iterative Signature Algorithm (ISA) [Ihmels et al., 2002, Bergmann et al., 2003, Ihmels et al., 2004] is a biclustering method. The input of a biclustering method is a matrix and the output is a set of biclusters that fulfill some criteria. A bicluster is a block of the potentially reordered input matrix.
Most commonly, biclustering algorithms are used on microarray expression data, to find gene sets that are coexpressed across a subset of the original samples. In the ISA papers the biclusters are called transription modules (TM), we will often refer them under this name in the following.
This tutorial specifically deals with the modular analysis of gene expression data. Section 7 gives a short summary of how ISA works. If you need more information of the underlying math or want to apply it to other data, then please see the referenced papers, the vignette titled "The Iterative Signature Algorithm" in the `isa2` R package, or the ISA homepage at `http://www.unil.ch/cbg/homepage/software.html`.

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 Section 9 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:

```
> 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. The threshold parameters tune the size of the modules, less stringent (i.e. smaller) values result bigger, less correlated modules. The optimal values depend on your data and some experimentation is needed to determine them.
Since running ISA might take a couple of minutes and the results depend on the random number generator used, the ISA run is commented out from the next code block, and we just load a precomputed set of modules that is distributed with the `eisa` package.

```
> thr.gene <- 2.7
> thr.cond <- 1.4
> set.seed(1) # to get the same results, always
> # modules <- ISA(ALL, thr.gene=thr.gene, thr.cond=thr.cond)
> data(ALLModulesSmall)
> modules <- ALLModulesSmall
```

This first applies a non-specific filter to 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 (after the filtering) 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. `pData()` returns the phenotype data of the expression set as a data frame. The `getOrganism()` function returns the scientific name of the organism under study, `annotation()` the name of the chip. For the former the appropriate annotation package must be installed.

```
> getNoFeatures(modules)
```

```
[1] 38 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"))
> chr1features <- sapply(mget(featureNames(modules),
     chr), function(x) "1" %in% x)
> modules[chr1features, ]
```

```
An ISAModules instance.
  Number of modules: 8
  Number of features: 345
  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 probesets (more precisely, the feature names coming from the `ExpressionSet` object) in the modules. It returns a list, here we just print the first entry.

```
> 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"   "37497_at"   "37344_at"   "38833_at"
[13] "38096_f_at" "37039_at"   "41723_s_at" "39248_at"
[17] "172_at"     "2047_s_at"  "33238_at"   "32184_at"
[21] "38147_at"   "38051_at"   "38750_at"   "33039_at"
[25] "33705_at"   "32794_g_at" "33121_g_at" "33369_at"
[29] "39709_at"   "35839_at"   "32649_at"   "633_s_at"
[33] "37759_at"   "33514_at"   "38319_at"   "39226_at"
[37] "1096_g_at"  "35016_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"
```

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
     -0.91      -0.93      -0.81      -0.85      -0.82
```

```
   1292_at     529_at   40375_at   39715_at   36669_at
     -0.75      -0.79      -0.86       0.76      -0.85
   36711_at   39420_at   37187_at   280_g_at 32901_s_at
     -0.87      -0.76      -0.83      -0.84      -0.91
 35372_r_at   33146_at 39822_s_at     287_at   37623_at
     -0.83      -0.91      -0.86      -0.91      -0.83
 34304_s_at   36674_at   36979_at   40448_at   40790_at
     -0.92      -0.80      -1.00      -0.80      -0.87
    1237_at
     -1.00
```

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

```
[[1]]
03002 04007 04008 04016 08001 12007 12026 15001 24008 27004
-0.96 -0.67 -0.68 -1.00 -0.74 -0.71 -0.66 -0.84 -0.82  0.52
28003 28019 28021 28023 28035 28037 28044 28047 43012 19017
 0.42  0.58  0.56  0.47  0.49  0.38  0.45  0.44  0.38 -0.93
28008 64005
 0.49 -0.71
```

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. The probes/samples that are not included in a module will have a zero score by definition.

```
> 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          22           0     2.7     1.4    1  22
2          10           0     2.7     1.4    1  24
3          26           0     2.7     1.4    1  24
11          8           0     2.7     1.4    1  26
```

```
61          7           0     2.7     1.4    1  22
62         11           0     2.7     1.4    1  22
63         16           0     2.7     1.4    1  24
99         12           0     2.7     1.4    1  23
   rob.limit
1         22
2         22
3         22
11        22
61        22
62        22
63        22
99        22
```

The `runData()` function returns additional information about the ISA run, see 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] "corx"

$prenormalize
[1] FALSE

$hasNA
[1] FALSE

$corx
[1] 3

$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 gene sets corresponding to the ISA 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 `ISAGO()` function.

```
> GO <- ISAGO(modules)
```

The `ISAGO()` function requires the annotation package of the chip, e.g. for the ALL data, the `hgu95av2.db` package is required.
The `GO` object is a list with three elements, these correspond to the GO ontologies, they are: biological function, cellular component and molecular function, in this order.

```
> GO

$BP
Gene to GO List BP  test for over-representation
5501 GO List BP ids tested (0-31 have p < 0.05)
Selected gene set sizes: 20-52
     Gene universe size: 3027
     Annotation package: hgu95av2

$CC
Gene to GO List CC  test for over-representation
829 GO List CC ids tested (0-20 have p < 0.05)
```

```
Selected gene set sizes: 20-56
     Gene universe size: 3144
     Annotation package: hgu95av2

$MF
Gene to GO List MF  test for over-representation
1207 GO List MF ids tested (0-11 have p < 0.05)
Selected gene set sizes: 19-54
     Gene universe size: 3106
     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. A *p*-value limit can be supplied to `summary()`. Note, that since `ISAGO()` 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$BP, p = 0.001)[[1]][, -6]
```

```
             Pvalue OddsRatio ExpCount Count Size
GO:0002376 6.6e-05       7.3      4.2    17  361
```

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 first 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$BP)[[1]][[3]]
```

```
[1] "360"   "3635"  "3932"  "4068"  "50852" "930"   "972"
```

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

```
> sigCategories(GO$BP)[[1]]
```

```
[1] "GO:0002376" "GO:0002504"
```

```
> library(GO.db)
> mget(na.omit(sigCategories(GO$BP)[[1]][1:3]),
     GOTERM)
```

```
$`GO:0002376`
GOID: GO:0002376
Term: immune system process
Ontology: BP
Definition: Any process involved in the development
    or functioning of the immune system, an
    organismal system for calibrated responses to
    potential internal or invasive threats.

$`GO:0002504`
GOID: GO:0002504
Term: antigen processing and presentation of peptide
    or polysaccharide antigen via MHC class II
Ontology: BP
Definition: The process by which an
    antigen-presenting cell expresses antigen
    (peptide or polysaccharide) on its cell surface
    in association with an MHC class II protein
    complex.
Synonym: peptide or polysaccharide antigen processing
    and presentation of via MHC class II
```

In addition, the following functions are implemented to work on the objects returned by `ISAGO()`: `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 usually return lists.

5.1.1 Multiple testing correction

By default, the `ISAGO()` 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 `ISAGO()` function for details, and also the `p.adjust()` function for the possible multiple testing correction schemes.

5.2 KEGG Pathway Database

Enrichment calculation against the KEGG pathway database goes essentially the same way as for the Gene Ontology, this time we use the `ISAKEGG()` function:

```
> KEGG <- ISAKEGG(modules)
> KEGG
```

```
Gene to   test for over-representation
262  ids tested (0-4 have p < 0.05)
Selected gene set sizes: 7-23
     Gene universe size: 1496
     Annotation package: hgu95av2
```

```
> summary(KEGG, p = 0.001)[[1]]
```

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

```
> library(KEGG.db)
> mget(sigCategories(KEGG, p = 0.001)[[1]], KEGGPATHID2NAME)
```

```
named list()
```

The functions mentioned in the Gene ontology enrichment section (`summary()`, `pvalues()`, `htmlReport()`, etc.) can be used for chromosome enrichment, 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 `ISACHR()` function:

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

```
   Pvalue OddsRatio ExpCount Count Size
19  0.011       5.7      1.6     7  192
```

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 (`summary()`, `pvalues()`, `htmlReport()`, etc.) can be used for chromosome enrichment, as well.

5.4 Predicted μRNA targets from the TargetScan database

μRNAs are short RNA molecules that regulate gene expression. TargetScan is a database of predicted target genes of μRNAs, for several organisms. There are two R packages that incorporate this data, one for human and another one for mouse, right now they can be downloaded from `http://www.unil.ch/cbg/homepage/software.html`. 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 `ISAmiRNA()` function should be used:

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

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, [Kaiser et al., 2009].

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 standard R `as()` function can be used to convert 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 <- as(modules, "Biclust")
> 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:    "38" "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, but `ISA2heatmap()` plots the raw values by default, optionally scaled. If you want to plot the normalized values, then you need supply the *norm* argument to `ISA2heatmap()`, see the manual for details.

```
> ISA2heatmap(modules, 1, ALL, norm = "sample",
      scale = "none", col = heat.colors(12))
```

We also create a color bar.

```
> sc <- function(x, ab) {
      x <- x - min(x)
      r <- range(x)
      x <- x/(r[2] - r[1])
      x * (ab[2] - ab[1]) + ab[1]
 }
> cb <- sc(1:12, range(exprs(ALL)[getFeatureNames(modules,
      1)[[1]], getSampleNames(modules, 2)[[1]]]))
> par(mar = c(3, 4, 5, 2) + 0.1)
> image(rbind(cb), axes = FALSE)
> axis(2, at = sc(1:12, 0:1), cb, label = format(cb,
      digits = 2))
```

The result is in Fig. 1.
`ISA2heatmap()` simply calls the `heatmap()` function, and passes additional arguments to it. See the manual of `heatmap()` for details.

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.

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

The `profilePlot()` 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.

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

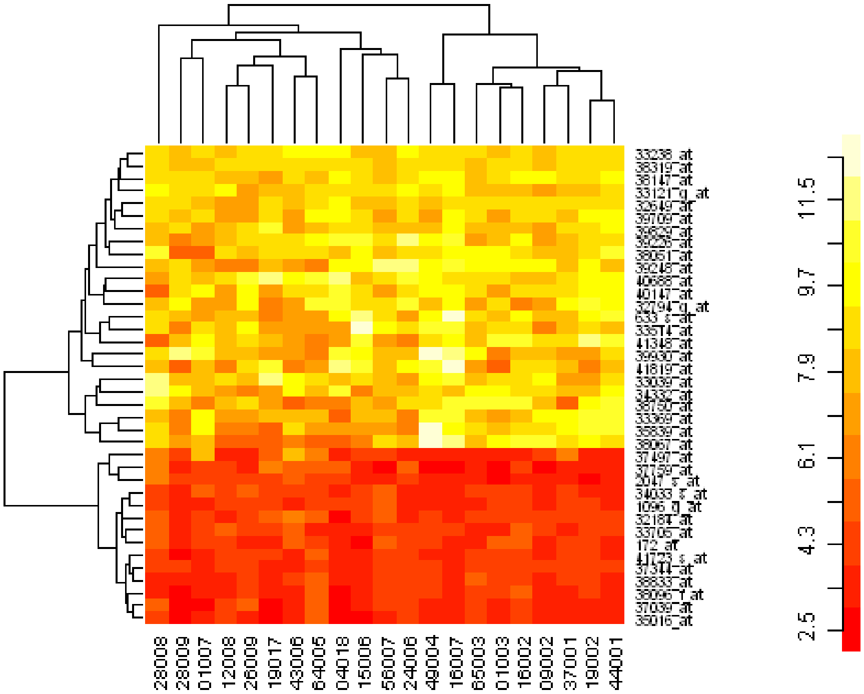


Figure 1: Heatmap of the first ISA module, plotted via the `ISA2heatmap()` function.

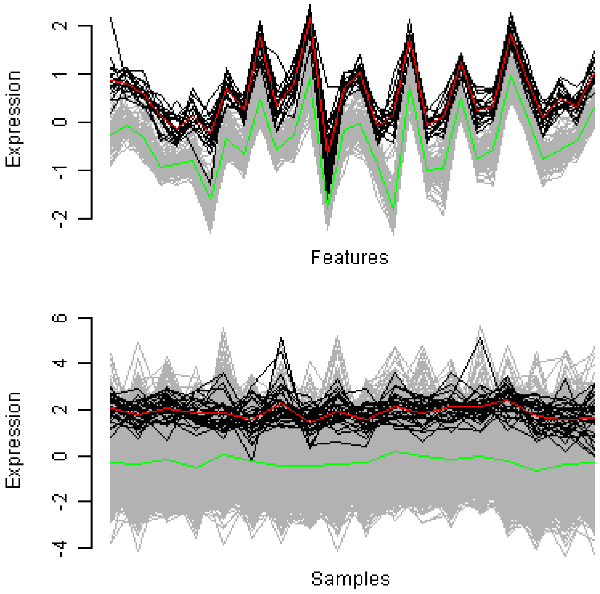


Figure 2: Profile plots for the second bicluster found by ISA.

a tree graph, 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 functions to plot parts of the GO graph that are 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$BP, p = 0.05)[[1]])
```

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

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

```
Vertices: 5
Edges: 4
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.
```

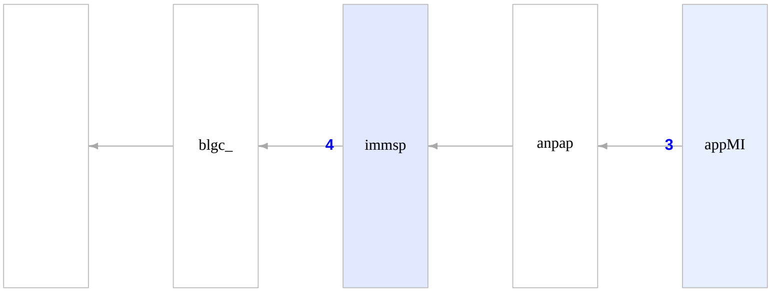


Figure 3: Part of the Gene ontology database, “Biological Function” ontology, that contains all GO terms enriched by a transcription module.

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

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

```
[1] 146
```

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

```
[1] 6.6
```

Let’s plot the graph, we can do this with the `gographPlot()` function, see Fig. 3.

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

Because the GO is not 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 plot. The graph object contains the full names of the categories 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] "immsp" "appMI" "blgc_" "anpap" NA
```

```
> 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] "biological_process"

[[4]]
[1] "antigen processing and presentation"

[[5]]
[1] "NA"
```

`gographPlot()` 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 `condPlot()` 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 (Fig. 4).

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

It is clear that the 30 genes included in this transcription module cannot separate B-cell and T-cell leukemia samples. Section 10.8 shows good separator modules for this data set.

6.6 Generating a HTML summary for the modules

The `ISAHTMLTable()` function creates a summary of a set of ISA modules, as an HTML table. The summary optionally includes the results of the enrichment calculations as well. Fig. 5 shows a screenshot of the table created for the 8 modules we have found earlier.

```
> htmldir <- tempdir()
> ISAHTMLTable(modules = modules, target.dir = htmldir,
     GO = GO, KEGG = KEGG, CHR = CHR)
```

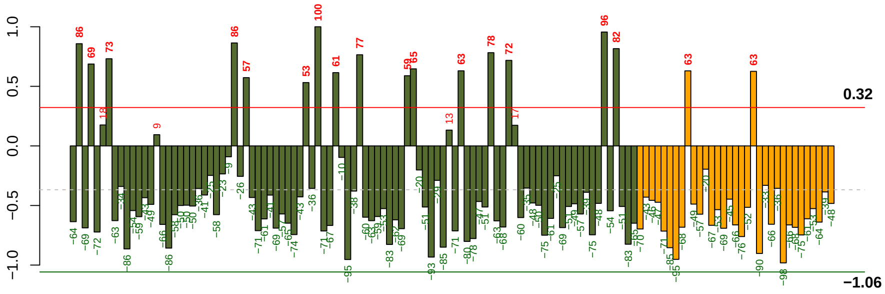
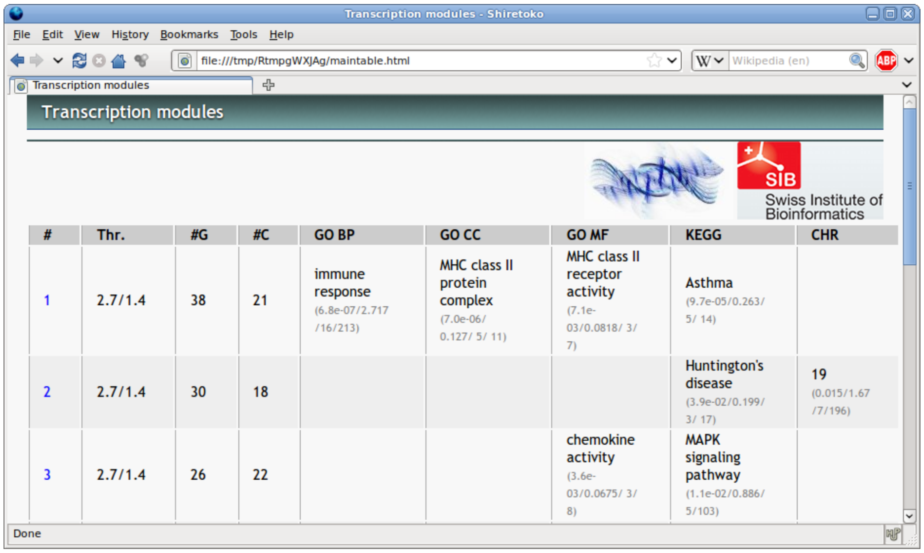


Figure 4: Condition (or sample score) plot for the ALL data and the second module. Every sample is represented as a bar, and its ISA score is plotted. The scores are also printed at the top or bottom of the bars. The scores corresponding to the ISA thresholds are represented as horizontal lines. The dashed line is the mean of the condition scores. B-cell samples are dark green, T-cell samples are orange. The green and red horizontal lines show the sample thresholds. Samples above the red line, and samples below the green line are included in the module.

```
> if (interactive()) {
     browseURL(URLencode(paste("file://", htmldir,
         "/maintable.html", sep = "")))
 }
```

The `ISAHTMLModules()` function generates a HTML page for every single module with the following information on it:

- An expression plot of the genes and samples in the module, including the ISA scores. This is done by calling `expPlot()`.
- Overlap plot of all modules, see `overlapPlot()`.
- Gene Ontology tree plots for the enriched GO terms, separately for the three ontologies., These are produced by calling `gograph()` and `gographPlot()`.
- Tables for the enriched Gene Ontology terms, separately for the three ontologies.
- A table for the enriched KEGG pathways.
- A table for the enriched miRNA families. (This is optional.)
- The list of genes in the module.
- The list of samples in the module.



Transcription modules - Shiretoko
File Edit View History Bookmarks Tools Help
file:///tmp/RtmpgWXJAg/maintable.html
Wikipedia (en)
Transcription modules
Done

Transcription modules

#	Thr.	#G	#C	GO BP	GO CC	GO MF	KEGG	CHR
1	2.7/1.4	38	21	immune response (6.8e-07/2.717 /16/213)	MHC class II protein complex (7.0e-06/ 0.127/ 5/ 11)	MHC class II receptor activity (7.1e-03/0.0818/ 3/ 7)	Asthma (9.7e-05/0.263/ 5/ 14)	
2	2.7/1.4	30	18				Huntington's disease (3.9e-02/0.199/ 3/ 17)	19 (0.015/1.67 /7/196)
3	2.7/1.4	26	22			chemokine activity (3.6e-03/0.0675/ 3/ 8)	MAPK signaling pathway (1.1e-02/0.886/ 5/103)	

Figure 5: A HTML summary table showing information for the first couple of modules found in the ALL data set. The "Thr." column shows the ISA thresholds that were used to find the module; the next two columns are the number of features (genes) and the number of samples (conditions) in the modules. The other columns show the most significantly enriched GO categories, KEGG pathways and chromosomes. The small grey numbers are the enrichment p-value, the number of genes expected to be hit by chance, the number of hits and the size of the category (within the respective gene universe), in this order.

- A condition plot, see the `condPlot()` function.

```
> ISAHTMLModules(eset = ALL, modules = modules,
     target.dir = htmldir, GO = GO, KEGG = KEGG,
     CHR = CHR)
```

The rows of the summary table generated by `ISAHTMLTable()` link to the module pages generated by `ISAHTMLModules()`, if the same *target.dir* argument was used for both of them.

Both the summary and the module pages can be generated with the `ISAHTML()` function.

6.7 The ExpressionView package

The previously mentioned visualization techniques focus on a single module at at time. The `ExpressionView` package visualizes a set of possibly overlapping biclusters together.

The `ExpressionView` package tries to reorder the rows and columns of the expression matrix in such a way that the genes/samples that appear together in a module, are placed right beside/above each other. The package directly supports the `ISAModules` and `ExpressionSet` classes, see its documentation for details.

7 How ISA works

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 non-zero 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. Using the "direction" parameter, one can keep values that are (a) significantly higher ("up"); (b) lower ("down") than the mean; or (c) 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 `ISAUnique()`

eliminates every module from the result of `ISAIterate()` 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 `ISAIterate()` 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. We show an example of using smart seeds in Section 10.

7.4 Normalization

Using in silico data we observed that ISA has the best performance if the input matrix is normalized (see `ISANormalize()`). The normalization produces two matrices: E_r and E_c. E_r is calculated by transposing E and centering and scaling its expression values for each sample (see the `scale()` R function). E_c is calculated by centering 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 gene signature of the samples.
It is possible to use another normalization, or not to use normalization at all; the user has to construct an `ISAExpressionSet` object containing the three matrices corresponding to the raw data, the gene-wise normalized data and the sample-wise normalized data. This object can be passed to the `ISAIter-`

`ate()` function. The matrices are not required to be different, the user can supply the raw data matrix three times, if desired.

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 signs 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[Madeira and Oliveira, 2004] 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. We use the `Bc` object that was created in Section 6.1. Here are the measures for the first bicluster:

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

```
[1] 4.0
```

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

```
[1] 2.1
```

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

```
[1] 0.39
```

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

```
[1] 2.6
```

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] 0.36
```

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 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 automatically 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 24 24 26 22 22 24 23
```

9 The `isa2` and `eisa` packages

ISA and its companion functions for visualization, functional enrichment calculation, etc. are distributed in two separate R 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.

If you analyze gene expression data, then we suggest using the interface provided in the `eisa` package. For other data, use the `isa2` package directly.

10 Finer control over ISA parameters

The `ISA()` function takes care of all steps performed in a modular study, and for each step it uses parameters, that 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.
If other filters are desired, then these can be implemented by using the functions of the `genefilter` package directly. 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 identifiers and remove the ones that don't map to any Entrez gene.

```
> 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 gene. The following code snipplet keeps the probeset with the highest variance.

```
> 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 `ISANormalize()` function can be used to calculate the normalized expression matrices; it returns an `ISAExpressionSet` object.

This is a subclass of `ExpressionSet`, and contains three expression matrices: the original raw expression, the row-wise (=gene-wise) normalized and the column-wise (=sample-wise) normalized expression matrix. The normalized expression matrices can be queried with the `featExprs()` and `sampExprs()` functions.

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

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

> dim(featExprs(ALL.normed))

[1] 1095  128

> dim(sampExprs(ALL.normed))

[1] 1095  128
```

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()` function 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.

```
> set.seed(3)
> 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. The two threshold parameters we use here were chosen after some experimentation; these result modules of the "right" size.

```
> modules1 <- ISAIterate(ALL.normed, feature.seeds = random.seeds,
     thr.feat = 1.5, thr.samp = 1.8, convergence = "cor")
> modules2 <- ISAIterate(ALL.normed, sample.seeds = smart.seeds,
     thr.feat = 1.5, thr.samp = 1.8, convergence = "cor")
```

10.6 Dropping non-unique modules

`ISAIterate()` returns the same number of modules as the number of input seeds; these are, however, not always meaningful, 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 `ISAUnique()` function eliminates the all-zero or non-convergent input seeds and keeps only one instance of the duplicated ones.

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

```
[1] 18
```

```
> length(modules2.unique)
```

```
[1] 3
```

18 modules were kept for the first set of seeds and 3 for the second set.

10.7 Dropping non-robust modules

The `ISAFilterRobust()` function filters a set of modules by running ISA with the same parameters on the 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 <- ISAFilterRobust(ALL.normed,
     modules1.unique)
> modules2.robust <- ISAFilterRobust(ALL.normed,
     modules2.unique)
> length(modules1.robust)

[1] 12

> length(modules2.robust)

[1] 3
```

We still have 12 modules for the first set of seeds and 3 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 is the weighted average of the normalized expression level of the modules genes. Thus, a good classifier module should have high sample scores for one type of ALL and low for the other. Here we perform a *t*-test to quantify this difference.
Let's first check, whether any of the modules can distinguish between T-cell and B-cell ALL samples. We perform *t*-tests for both sets of modules.

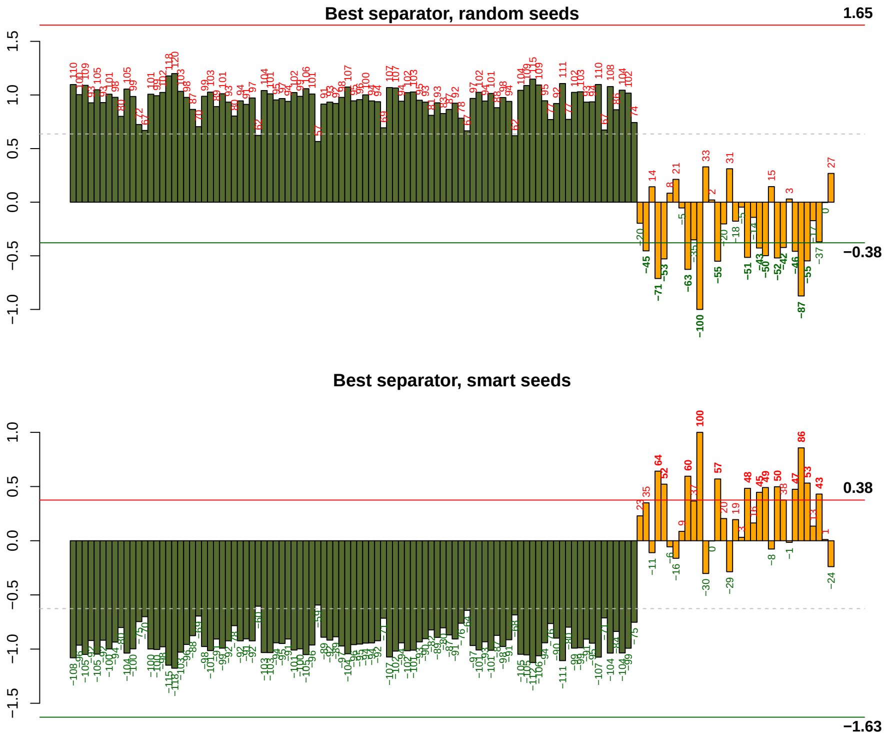
```
> scores1 <- getSampleMatrix(modules1.robust)
> tt1 <- colttests(scores1, as.factor(substr(type,
     1, 1)))
> scores2 <- getSampleMatrix(modules2.robust)
> tt2 <- colttests(scores2, as.factor(substr(type,
     1, 1)))
> sign1 <- which(p.adjust(tt1$p.value, "holm") <
     0.05)
> sign2 <- which(p.adjust(tt2$p.value,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

[1] 3 4 5

> sign2

[1] 1 3
```

For the first (random) set of seeds 3 modules show significant difference for T-cell and B-cell samples, for the second (smart) set 2 of them.
Let's make some condition plots for the best separating modules from each set (Fig. 6).


Best separator, random seeds

Best separator, smart seeds

Figure 6: Condition plots for the best separator module found with random seeds and the one found with the smart seeds. They are essentially the same, with some minor differences.

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

As it turns out, the best separator module found using the smart seeds was also found with random seeds. This can be seen by calculating the correlation between the separator modules found in the two experiments.

```
> cor(getSampleMatrix(modules1.TB), getSampleMatrix(modules2.TB))
```

```
      [,1]   [,2]
[1,] -0.96  0.143
```

```
[2,]  0.37 -0.012
[3,]  0.21 -0.204
```

```
> cor(getFeatureMatrix(modules1.TB), getFeatureMatrix(modules2.TB))
```

```
      [,1]   [,2]
[1,] -0.96  0.020
[2,]  0.40 -0.011
[3,]  0.45 -0.076
```

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 <- ISAGO(modules1.TB)
> GO.dysreg2 <- ISAGO(modules2.TB)
> KEGG.dysreg1 <- ISAKEGG(modules1.TB)
> KEGG.dysreg2 <- ISAKEGG(modules2.TB)
```

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

```
> gocats <- unique(unlist(c(sigCategories(GO.dysreg1$BP),
     sigCategories(GO.dysreg1$CC), sigCategories(GO.dysreg1$MF),
     sigCategories(GO.dysreg2$BP), sigCategories(GO.dysreg2$CC),
     sigCategories(GO.dysreg2$MF))))
> keggp <- unique(unlist(c(sigCategories(KEGG.dysreg1),
     sigCategories(KEGG.dysreg2))))
> sapply(mget(gocats, GOTERM), Term)
```

```
                                  GO:0048015
     "phosphoinositide-mediated signaling"
                                  GO:0007051
                    "spindle organization"
                                  GO:0042101
                 "T cell receptor complex"
                                  GO:0044425
                           "membrane part"
                                  GO:0000777
        "condensed chromosome kinetochore"
                                  GO:0000779
"condensed chromosome, centromeric region"
                                  GO:0008380
                            "RNA splicing"
                                  GO:0000166
                      "nucleotide binding"
```

```
> mget(keggp, KEGGPATHID2NAME)

$`04623`
[1] "Cytosolic DNA-sensing pathway"

$`04920`
[1] "Adipocytokine 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] 1.3e-11

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

[1] 3.8e-12
```

Both module sets seem to have some good separators, let us make some condition plots (Fig. 7).

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

```
> 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] 1 1
```

Indeed, they are almost the same, their Pearson correlation is 1 for the sample scores and 0.996 for the feature (=gene) scores.

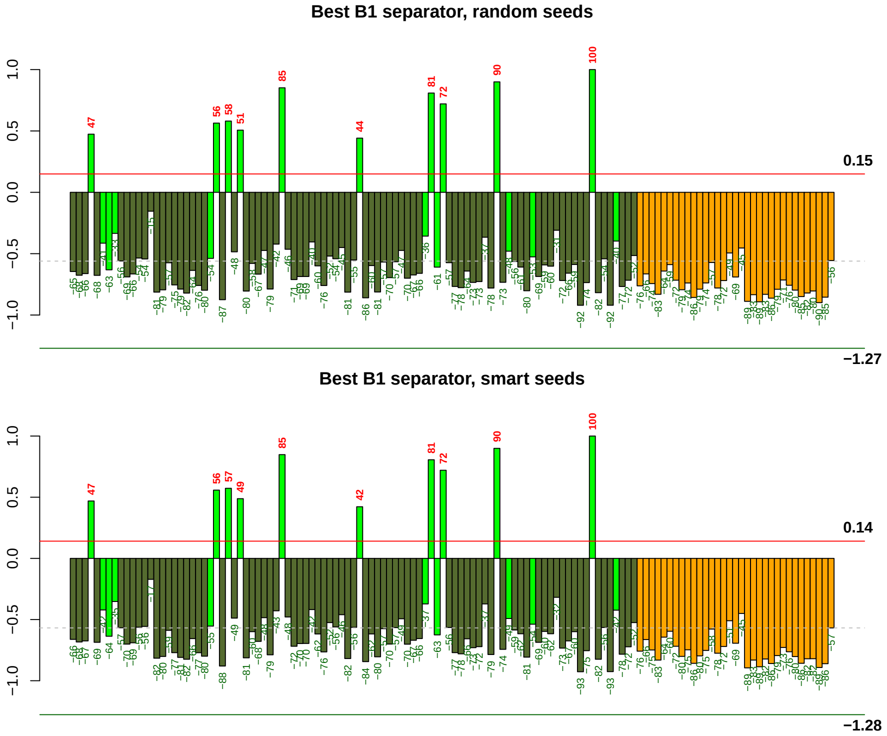


Figure 7: Condition plots for the modules that best separate the B1-type ALL samples from the other B-cell ALL samples. The two modules are essentially the same.

11 More information

For more information about the ISA, please see the references below. The ISA homepage at `http://www.unil.ch/cbg/homepage/software.html` has example data sets, and all ISA related tutorials and papers.

12 Session information

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

- R version 2.12.0 RC (2010-10-11 r53293), `i386-pc-mingw32`
- Locale: `LC_COLLATE=C`, `LC_CTYPE=English_United States.1252`, `LC_MONETARY=English_United States.1252`, `LC_NUMERIC=C`, `LC_TIME=English_United States.1252`
- Base packages: base, datasets, grDevices, graphics, grid, methods, stats, utils
- Other packages: ALL 1.4.7, AnnotationDbi 1.12.0, Biobase 2.10.0, Category 2.16.0, DBI 0.2-5, GO.db 2.4.5, KEGG.db 2.4.5, MASS 7.3-8, RSQLite 0.9-2, affy 1.28.0, biclust 0.9.1, colorspace 1.0-1, eisa 1.2.0, genefilter 1.32.0, hgu95av2.db 2.4.5, igraph 0.5.4-2, isa2 0.2.1, org.Hs.eg.db 2.4.6, vcd 1.2-9, xtable 1.5-6
- Loaded via a namespace (and not attached): GSEABase 1.12.0, RBGL 1.26.0, XML 3.2-0.1, affyio 1.18.0, annotate 1.28.0, graph 1.28.0, preprocessCore 1.12.0, splines 2.12.0, survival 2.35-8, tools 2.12.0

References

[Bergmann et al., 2003] Bergmann, S., Ihmels, J., and Barkai, N. (2003). Iterative signature algorithm for the analysis of large-scale gene expression data. *Phys Rev E Nonlin Soft Matter Phys*, page 031902.

[Ihmels et al., 2004] Ihmels, J., Bergmann, S., and Barkai, N. (2004). Defining transcription modules using large-scale gene expression data. *Bioinformatics*, pages 1993–2003.

[Ihmels et al., 2002] Ihmels, J., Friedlander, G., Bergmann, S., Sarig, O., Ziv, Y., and Barkai, N. (2002). Revealing modular organization in the yeast transcriptional network. *Nat Genet*, pages 370–377.

[Kaiser et al., 2009] Kaiser, S., Santamaria, R., Theron, R., Quintales, L., and Leisch, F. (2009). biclust: Bicluster algorithms. R package version 0.7.2.

[Madeira and Oliveira, 2004] Madeira, S. and Oliveira, A. (2004). Biclustering algorithms for biological data analysis: a survey. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 1:24–45.