The Iterative Signature Algorithm for Gene Expression Data

Gábor Csárdi

October 27, 2020

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 6 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 8 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)
> 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 9 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: 332
  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
-0.9053170 -0.9327898 -0.8120785 -0.8507266 -0.8195864
   1292_at     529_at   40375_at   39715_at   36669_at
-0.7520874 -0.7887937 -0.8565162  0.7589608 -0.8510981
  36711_at   39420_at   37187_at   280_g_at 32901_s_at
-0.8675340 -0.7641193 -0.8338932 -0.8395013 -0.9076414
35372_r_at   33146_at 39822_s_at     287_at   37623_at
-0.8283072 -0.9081089 -0.8603682 -0.9134679 -0.8346154
34304_s_at   36674_at   36979_at   40448_at   40790_at
-0.9245084 -0.8015006 -1.0000000 -0.8029958 -0.8685983
   1237_at
-0.9964699

> getSampleScores(modules, 3)

[[1]]
     03002      04007      04008      04016      08001
-0.9606187 -0.6681491 -0.6835771 -1.0000000 -0.7404822
     12007      12026      15001      24008      27004
-0.7130576 -0.6552245 -0.8367460 -0.8245874  0.5244274
     28003      28019      28021      28023      28035
```

```
 0.4237176  0.5766343  0.5617935  0.4674956  0.4881987
     28037      28044      28047      43012      19017
 0.3756713  0.4524610  0.4401485  0.3822326 -0.9345939
     28008      64005
 0.4862616 -0.7058733
```

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 21.98199
2          10           0     2.7     1.4    1 24.31987
3          26           0     2.7     1.4    1 23.77689
11          8           0     2.7     1.4    1 26.23544
61          7           0     2.7     1.4    1 22.47315
62         11           0     2.7     1.4    1 22.05568
63         16           0     2.7     1.4    1 23.97600
99         12           0     2.7     1.4    1 22.84282
   rob.limit
1   21.98116
2   21.98116
3   21.98116
11  21.98116
61  21.98116
62  21.98116
63  21.98116
99  21.98116
```

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
11630 GO List BP ids tested (0-69 have p < 0.05)
Selected gene set sizes: 21-56
     Gene universe size: 3205
     Annotation package: hgu95av2

$CC
Gene to GO List CC  test for over-representation
1546 GO List CC ids tested (0-27 have p < 0.05)
Selected gene set sizes: 21-57
     Gene universe size: 3240
     Annotation package: hgu95av2

$MF
Gene to GO List MF  test for over-representation
1747 GO List MF ids tested (0-16 have p < 0.05)
Selected gene set sizes: 20-56
     Gene universe size: 3210
     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:0050852 1.208332e-07 17.149055 1.269891    13  110
GO:0050851 1.254153e-07 15.197434 1.570047    14  136
GO:0002250 3.017653e-07 14.111517 1.673947    14  145
GO:0002429 3.649288e-07 12.651515 2.043370    15  177
GO:0002757 3.649288e-07 12.651515 2.043370    15  177
GO:0002768 1.086852e-06 11.590909 2.204992    15  191
GO:0002764 1.260179e-06 11.453013 2.228081    15  193
GO:0002253 1.355604e-06 11.385221 2.239626    15  194
GO:0050778 6.466513e-06  9.379752 2.932293    16  254
GO:0002682 7.035958e-05  6.703146 5.691420    20  493
GO:0002684 1.553704e-04  6.955217 4.179095    17  362
GO:0050776 2.857099e-04  6.925083 3.809672    16  330
```

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] "27040" "28639" "3108"  "3113"  "3115"  "3122"  "3635"
 [8] "4068"  "50852" "814"   "915"   "917"   "930"   "972"
```

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

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

```
 [1] "GO:0050852" "GO:0050851" "GO:0002250" "GO:0002429"
 [5] "GO:0002757" "GO:0002768" "GO:0002764" "GO:0002253"
 [9] "GO:0050778" "GO:0002682" "GO:0002684" "GO:0050776"
[13] "GO:0006955" "GO:0002376" "GO:0046649" "GO:0048584"
[17] "GO:0007166" "GO:0050865" "GO:0051249" "GO:0045321"
```

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

```
$`GO:0050852`
GOID: GO:0050852
Term: T cell receptor signaling pathway
Ontology: BP
Definition: A series of molecular signals initiated
    by the cross-linking of an antigen receptor on a
    T cell.
Synonym: T lymphocyte receptor signaling pathway
Synonym: T lymphocyte receptor signalling pathway
Synonym: T-cell receptor signaling pathway
Synonym: T-cell receptor signalling pathway
Synonym: T-lymphocyte receptor signaling pathway
Synonym: T-lymphocyte receptor signalling pathway
Synonym: TCR signaling pathway

$`GO:0050851`
GOID: GO:0050851
Term: antigen receptor-mediated signaling pathway
Ontology: BP
Definition: A series of molecular signals initiated
    by the cross-linking of an antigen receptor on a
    B or T cell.
Synonym: antigen receptor-mediated signalling pathway

$`GO:0002250`
GOID: GO:0002250
Term: adaptive immune response
Ontology: BP
Definition: An immune response mediated by cells
    expressing specific receptors for antigen
    produced through a somatic diversification
    process, and allowing for an enhanced secondary
    response to subsequent exposures to the same
    antigen (immunological memory).
Synonym: acquired immune response
Synonym: immune memory response
```

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.

6 How ISA works

6.1 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. 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.

6.2 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.

6.3 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 9.

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

6.5 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.

7 Bicluster coherence and robustness measures

7.1 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 **??**. Here are the measures for the first bicluster:

```
> library(biclust)
> Bc <- as(modules, "Biclust")
> constantVariance(exprs(ALL), Bc, number=1)

[1] 4.041903

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

[1] 2.055509

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

[1] 0.3876909

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

[1] 2.611917
```

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] 0.3582246
```

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

7.2 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] 21.98199 24.31987 23.77689 26.23544 22.47315 22.05568
[7] 23.97600 22.84282
```

8 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.

9 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.

9.1 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.

```
> library(genefilter)
> 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.

9.2 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,]
```

9.3 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] 1080  128

> dim(sampExprs(ALL.normed))

[1] 1080  128
```

9.4 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.

9.5 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")
```

9.6 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] 25

> length(modules2.unique)

[1] 5
```

25 modules were kept for the first set of seeds and 5 for the second set.

9.7 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] 17

> length(modules2.robust)

[1] 5
```

We still have 17 modules for the first set of seeds and 5 for the second set.

10 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.

11 Session information

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

- R version 4.0.3 (2020-10-10), `x86_64-pc-linux-gnu`
- Locale: `LC_CTYPE=en_US.UTF-8`, `LC_NUMERIC=C`, `LC_TIME=en_US.UTF-8`, `LC_COLLATE=C`, `LC_MONETARY=en_US.UTF-8`, `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`
- Running under: `Ubuntu 18.04.5 LTS`
- Matrix products: default
- BLAS: `/home/biocbuild/bbs-3.12-bioc/R/lib/libRblas.so`
- LAPACK: `/home/biocbuild/bbs-3.12-bioc/R/lib/libRlapack.so`
- Base packages: base, datasets, grDevices, graphics, grid, methods, parallel, stats, stats4, utils
- Other packages: ALL 1.31.0, AnnotationDbi 1.52.0, Biobase 2.50.0, BiocGenerics 0.36.0, GO.db 3.12.0, IRanges 2.24.0, MASS 7.3-53, S4Vectors 0.28.0, biclust 2.0.2, colorspace 1.4-1, eisa 1.42.0, genefilter 1.72.0, hgu95av2.db 3.2.3, isa2 0.3.5, lattice 0.20-41, org.Hs.eg.db 3.12.0
- Loaded via a namespace (and not attached): Category 2.56.0, DBI 1.1.0, GSEABase 1.52.0, Matrix 1.2-18, R6 2.4.1, RBGL 1.66.0, RSQLite 2.2.1, Rcpp 1.0.5, XML 3.99-0.5, additivityTests 1.1-4, annotate 1.68.0, bit 4.0.4, bit64 4.0.5, blob 1.2.1, class 7.3-17, compiler 4.0.3, crayon 1.3.4, digest 0.6.27, dplyr 1.0.2, ellipsis 0.3.1, flexclust 1.4-0, generics 0.0.2, ggplot2 3.3.2, glue 1.4.2, graph 1.68.0, gtable 0.3.0, httr 1.4.2, lifecycle 0.2.0, magrittr 1.5, memoise 1.1.0, modeltools 0.2-23, munsell 0.5.0, pillar 1.4.6, pkgconfig 2.0.3, purrr 0.3.4, rlang 0.4.8, scales 1.1.1, splines 4.0.3, survival 3.2-7, tibble 3.0.4, tidyr 1.1.2, tidyselect 1.1.0, tools 4.0.3, vctrs 0.3.4, xtable 1.8-4

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.

[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.