The Iterative Signature Algorithm

Gábor Csárdi

June 11, 2012

Contents

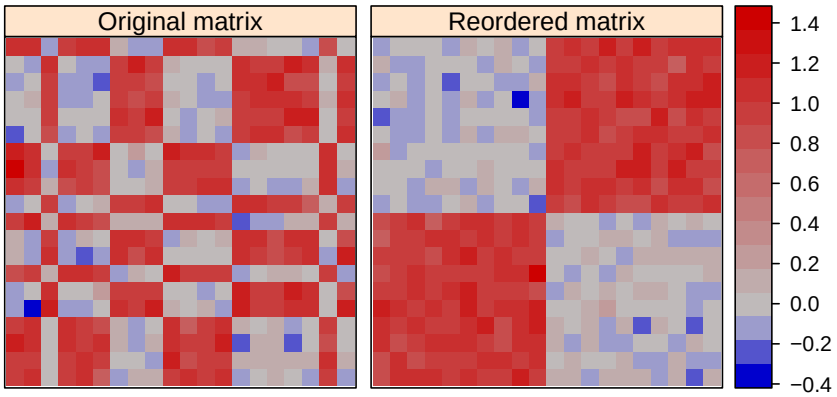

Figure 1: An artificial data matrix, on the left. On the right the reordered data matrix with two blocks.

1 For the impatient

To run the typical ISA work flow with default parameters on your data matrix, just load the `isa2` package and call the `isa()` function with your matrix as the single argument. The return value of `isa()` is a named list, the members `rows` and `columns` contain the biclusters ISA have found. Every bicluster is given by a pair of columns from `rows` and `columns` (i.e. the first columns define the first bicluster, etc.) and the elements of the biclusters are the non-zero elements in the columns of `rows` and `columns`.

Please continue reading for a less dense tutorial.

2 Introduction

The Iterative Signature Algorithm (ISA) [Ihmels, 2002, Bergmann et al., 2003, Ihmels, 2004] is biclustering method. Its input is a matrix and its output is a set of biclusters: blocks of the potentially reordered input matrix, that fulfill some predefined criteria. A biclustering algorithm typically tries to find blocks that are different from the rest of the matrix, e.g. the values covered by the bicluster are all above or below the background.

The ISA is developed to find biclusters (or modules as most of the ISA papers call them) that have correlated rows and columns. More precisely, the rows in the bicluster need to be only correlated across the columns of the bicluster and vice versa.

Fig. 1 shows possibly the simplest example of a (rather artificial) data matrix with very strong modular structure. It is a 20×20 matrix and—after reordering its rows and columns—it has two correlated blocks, each of size 10×10.

3 How ISA works

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

3.1 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 rows in E. First, 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 (direction="*up*"), or keeping the ones that are significantly lower than the mean (direction="*down*"); or keeping both (direction="*updown*").

The thresholded vector c_0 is the (column) *signature* of r_0. Then the (row) 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.

The `isa.iterate()` function performs the ISA iteration from a given set of seed vectors.

It is very possible that the ISA finds the same module more than once; two or more seed vectors 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 one that was already found from a different seed.

The `isa()` function performs the whole ISA workflow, this includes running `isa.iterate()` and `isa.unique()`.

It might be also apparent, that the ISA biclusters are soft, i.e. they might have an overlap in their rows, columns, or both. It is also possible that some rows and/or columns of the input matrix are not found to be part of any 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.

3.2 Parameters

The two main parameters of ISA are the two thresholds (one for the rows and one for the columns). They basically define the stringency of the modules. If the row threshold is high, then the modules will have very similar rows. If it is

mild, then modules will be bigger, with less similar rows than in the first case. The same applies to the column threshold and the columns of the modules.

3.3 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 rows/columns, 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.

3.4 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, then centering and scaling its rows (see the `scale()` R function). E_c is calculated by centering and scaling the rows of E. E_r is used to calculate the (column) signature of rows and E_c is used to calculate the (row) signature of the columns.

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 rows, `Ec` will be used for the signature of the columns. If you want to use the same matrix in both steps, then supply it twice, the first one transposed.

3.5 Row and column scores

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

3.6 Robustness

As ISA is an unsupervised algorithm, it may very well find some modules, even if you feed in noise as the input matrix. To avoid these spurious modules

we defined a robustness measure, a single number for a module that measures how well the rows and the columns are correlated. The robustness score is a generalization of the first singular value of the matrix. If there would be no thresholding during the ISA iteration, then the ISA would (almost always) converge to the leading SVD vector and the robustness score would be the corresponding singular value.

It is recommended that the user uses `isa.filter.robust()` to run ISA on the scrambled input matrix with the same threshold parameters and then drop every module, which has a robustness score lower than the highest robustness score among modules found in the scrambled data.

4 A simple work flow

The simplest way to use ISA on your data is by calling the `isa()` function with your input matrix as the single argument. This function uses random seeding and has three optional arguments:

thr.row A numeric vector, the row thresholds to use. By default this is 1, 1.5, 2, 2.5, 3.

thr.col A numeric vector, the column thresholds to use. By default this is 1, 1.5, 2, 2.5, 3.

no.seeds The number of random seed vectors to generate. By default 100 seeds are generated.

The `isa()` function runs the ISA algorithm for all threshold combinations of the supplied row and column thresholds. For every single run the same set of random seeds are used.

Let us see an example. We load the `isa2` package [Csárdi, 2009], and generate some in silico data with the `isa.in.silico()` function first:

```
> set.seed(10)
> library(isa2)
> data <- isa.in.silico(200,200,2,50,50)[[1]]
```

The function call above generates a data matrix similar to the one in Fig. 1, but bigger. The first two arguments define the size of the data matrix (200×200), the third the number of modules (two) and the fourth and fifth give the size of the modules, both are 50×50 submatrices in this case. By default `isa.in.silico()` creates non-overlapping modules with some background noise.

`isa.in.silico()` returns a list with three elements, we are most interested in the first now, that is the actual artificial data matrix. The second and third entries in the list give the correct module memberships of the elements, see the manual page of `isa.in.silico()` for the details. Let's take a look at the input data, see Fig. 2.

```
> images(list(data), xlab="", ylab="", strip=FALSE)
```

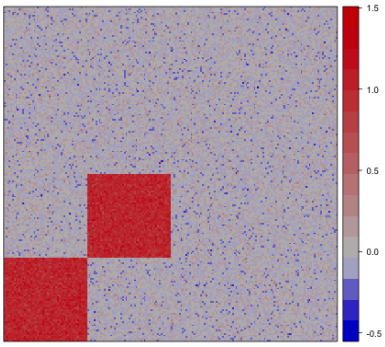

Figure 2: Another artificial data matrix with two modules and a lot of noisy elements.

All we have to do now is calling the `isa()` function on the data matrix.

```
> modules <- isa(data)
> names(modules)
```

```
[1] "rows"     "columns"  "seeddata" "rundata"
```

The `isa()` function returns a list with four elements. '`rows`' and '`columns`' define the modules. The non-zero elements in the first column of both define the first module, the second columns define the second module, etc. The number of columns in '`rows`' (and '`columns`') correspond to the number of modules.

```
> ncol(modules$rows)
```

```
[1] 3
```

ISA has found 3 modules in the artificial data. This seems a bit surprising, as we expected only two, so let us take a closer look. We simply plot the row scores of the individual modules (Fig. 3). The first two modules corresponds to the two blocks in the input matrix. The third module is the union of them, with opposite score signs. The `isa()` function also finds modules containing anticorrelated rows/columns, as long as the anti-correlation is coordinated. I.e. in the third module, the first 50 rows *always* behave in an opposite way compared to the second 50 rows. This is why they are collected into a single module. The following code creates Fig. 3.

```
> layout(cbind(seq(ncol(modules$rows))))
> sapply(seq(ncol(modules$rows)), function(x) {
   par(mar=c(3,5,1,1))
   plot(modules$rows[,x], ylim=c(-1,1),
        ylab="scores", xlab=NA,
        cex.lab=2, cex.axis=1.5)
 })
```

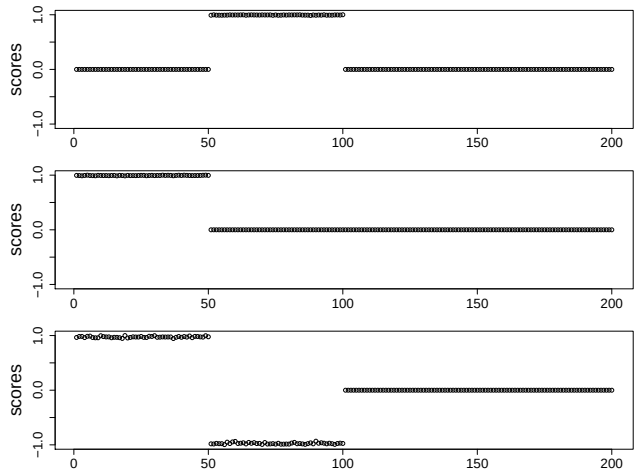

Figure 3: The row scores of the three modules found in the two-block artificial data matrix. A score of zero means that the row is not included in the module. Notice that in the third module the first 50 and the second 50 rows have opposite score signs, i.e. they are anti-correlated.

The '`rundata`' entry of the return value of `isa()` contains information about the ISA run, all the parameters (with the default values in our case) are included here. It has the following members:

```
> names(modules$rundata)
```

```
 [1] "direction"    "eps"          "cor.limit"
 [4] "maxiter"      "N"            "convergence"
 [7] "prenormalize" "hasNA"        "corx"
[10] "unique"       "oscillation"  "rob.perms"
```

See the documentation of `isa()` for more about these.

Finally, the '`seeddata`' entry is a data frame, it contains various information about the individual modules and the seeds that were used to find them. There is one row for each module. It has the following columns:

```
> colnames(modules$seeddata)
```

```
[1] "iterations"  "oscillation" "thr.row"     "thr.col"
[5] "freq"        "rob"         "rob.limit"
```

The columns '`thr.row`' and '`thr.col`' contain the row and columns thresholds that were used to find the module, while the '`rob`' column contains its robustness score:

```
> modules$seeddata
```

```
    iterations oscillation thr.row thr.col freq       rob
171          7           0     1.5     1.5   63  84.18955
191          7           0     1.5     1.5   52  84.16510
1            7           0     1.0     1.0   98 112.19460
    rob.limit
171  17.35840
191  17.35840
1    22.34714
```

See the documentation of the `isa()` function for more information about these and other fields.

Finally, let us show how to transform the result of the `isa()` function to a list of biclusters. Each entry of the list will have two sublists, the first – named '`rows`' – will contain the rows indices of the module, the second – named '`columns`' – the column indices:

```
> mymodules <- lapply(seq(ncol(modules$rows)), function(x) {
   list(rows=which(modules$rows[,x] != 0),
        columns=which(modules$columns[,x] != 0))
 })
> length(mymodules)

[1] 3

> mymodules[[1]]$rows

 [1]  51  52  53  54  55  56  57  58  59  60  61  62  63  64
[15]  65  66  67  68  69  70  71  72  73  74  75  76  77  78
[29]  79  80  81  82  83  84  85  86  87  88  89  90  91  92
[43]  93  94  95  96  97  98  99 100

> mymodules[[1]]$columns

 [1]  51  52  53  54  55  56  57  58  59  60  61  62  63  64
[15]  65  66  67  68  69  70  71  72  73  74  75  76  77  78
[29]  79  80  81  82  83  84  85  86  87  88  89  90  91  92
[43]  93  94  95  96  97  98  99 100
```

5 A detailed work flow

In this section we will perform each step of the ISA analysis individually. This makes sense only if the user wants to adjust the parameters of some steps. Otherwise a simple call to the `isa()` function would do.

Let us create some in-silico data to be analyzed:

```
> data <- isa.in.silico(200, 100, 10)
```

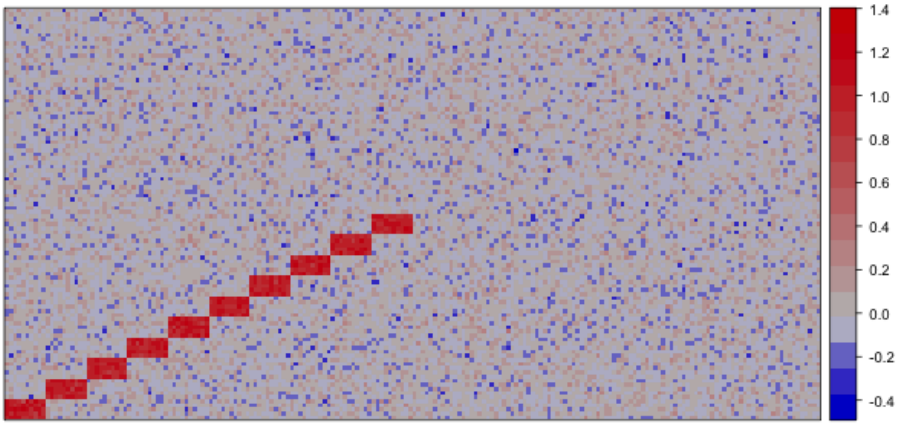

Figure 4: In-silico data with ten non-overlapping modules.

This will be a 200×100 matrix, with 10 modules. By default "half" of the matrix is filled with noise, so each module will be a 10×5 submatrix. See the input matrix on Fig. 4.

```
> images(list(data[[1]]), strip=FALSE, xlab="", ylab="")
```

5.1 Preparing the data

One ISA iteration consists of two thresholded matrix multiplications. The data preparation step means producing these two matrices, E_r and E_c from the input matrix, E. The default behavior (i.e. what the `isa()` function does) is to row-wise scale and center the transposed of E to get E_r; and to row-wise scale and center E to get E_c. The `isa.normalize()` function performs the appropriate scaling and returns E_r and E_c in a list:

```
> normed.data <- isa.normalize(data[[1]])
> names(normed.data)

[1] "Er" "Ec"

> dim(normed.data$Er)

[1] 100 200
```

```
> dim(normed.data$Ec)

[1] 200 100
```

Let us do a quick check that the rows of the matrices are indeed centered and scaled.

```
> max(abs(rowSums(normed.data$Er)))

[1] 5.096618e-15

> max(abs(rowSums(normed.data$Ec)))

[1] 4.108693e-15

> max(abs(apply(normed.data$Er, 1, sd)-1))

[1] 1.110223e-16

> max(abs(apply(normed.data$Ec, 1, sd)-1))

[1] 1.110223e-16
```

5.2 Running the ISA

To run the ISA, we need to create some starting seeds. The `isa()` function uses sparse random seeds, produced by `generate.seeds()`. We also have the choice of using non-random seeds, e.g. if you have a matrix with gene expression data measured on many samples you can use "gene" seeds that correspond to biological pathways; in this case ISA does a biased search to find modules related to the input pathways. Or, in a case/control experiment one can use seed vectors that correspond to cases and look for modules that are different in the control and the case samples.

For now, we will use thousand random seeds, with different sparseness:

```
> row.seeds <- generate.seeds(length=nrow(data[[1]]), count=1000,
                              sparsity=c(2,5,10,100))
```

Note, that vector giving the sparsity is recycled, thus we will have four kind of seeds, with 2, 5, 10 and 100 rows; 250 of each kind.

We are ready to run the ISA now. Let us assume that we only want to find modules that are "above" the background. (In this artificial case we actually know that there are no modules below the background.) The *direction* argument is thus set to `"up"`.

```
> modules <- isa.iterate(normed.data, row.seeds=row.seeds,
                         thr.row=2, thr.col=2, direction="up")
```

Unlike `isa()`, `isa.iterate()` does not merge or filter the modules the input seeds converge to. Consequently, the `modules` object contains 1000 modules, but these are not necessarily unique:

```
> ncol(modules$rows)

[1] 1000
```

It is also possible that some seeds converge to an all-zero vector, we have 44 such modules:

```
> sum(apply(modules$rows==0, 2, all))

[1] 44
```

The `isa.unique()` function finds duplicate or very similar modules in a list returned by `isa.iterate()` and keeps only a single one of them. It also eliminates all-zero modules and seeds that did not converge; `NA` columns in the '`rows`' and '`columns`' matrix correspond to non-convegent seeds:

```
> modules2 <- isa.unique(normed.data, modules, cor.limit=0.9)
```

The *cor.limit* argument specifies the correlation limit above which two modules are considered to be the same. Let us see how many modules are left:

```
> ncol(modules2$rows)

[1] 172
```

We still have 172 modules. Let us check whether each real module was find by the ISA. Remember that `isa.in.silico()` stores the correct modules in the second and third entry of its return value.

```
> found.rows <- cor(data[[2]], modules2$rows!=0)
> found.cols <- cor(data[[3]], modules2$columns!=0)
> found <- pmin(found.rows,found.cols)
> apply(found, 1, max)

 [1] 1 1 1 1 1 1 1 1 1 1
```

Out of the ten modules, 10 were correctly identified by the ISA.

5.3 Robustness of biclusters, filtering the results

There are two features that we expect from a good biclustering algorithm. The first is sensitivity: it is able to find all biclusters in the input data. The second is specificity: it should not generate spurious biclusters, or in other words biclusters that are correlated just by chance.

To address this problem, we have developed a robustness measure, a single scalar number that gives how correlated the rows and columns of a given module are. This measure is a simple generalization of the singular value decomposition (SVD) singular value, and can be calculated with the `robustness()` function:

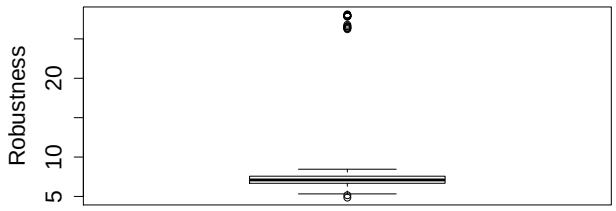

Figure 5: Box-plot for the robustness scores of the modules found by the ISA. Some modules have a much higher score than the rest.

```
> rob <- robustness(normed.data, modules2$rows, modules2$columns)
> summary(rob)

   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  4.852   6.672   7.094   9.092   7.558  28.120
```

The robustness of the 172 modules varies considerably, from 4.85 to 28.12. For more insight we create a boxplot for the robustness scores (Fig. 5).

```
> par(cex.lab=1.5, cex.axis=1.5)
> boxplot(rob, ylab="Robustness")
```

One can use the robustness measure to filter a list of modules, in the following way. We permute the input matrix and run the same module finding procedure that we did for the original matrix. Then we calculate the robustness scores for both the real and the scrambled modules and eliminate the "real" modules having a robustness score that is less than the score of at least one scrambled module. Note, that this filtering can be done only if both sets of modules were found using the same threshold parameters. The procedure is implemented in the `isa.filter.robust()` function:

```
> modules3 <- isa.filter.robust(data[[1]], normed.data, modules2,
                                perms=1, row.seeds=row.seeds)
```

We used the very same row seeds, as for the original matrix; this is not strictly required. The *perms* argument sets the number of permutations to perform.

Note that the ISA run on the scrambled matrix usually takes longer, than on the real data, because it takes more steps for the ISA to converge.

After the robustness filtering we have 18 modules left. This is still more than the ten correct modules, so let us take a closer look. First, let us create another plot with the robustness scores of the remaining modules. `isa.filter.robust()` places the robustness scores in the seed data, so we don't need to calculate them again. See Fig. 6.

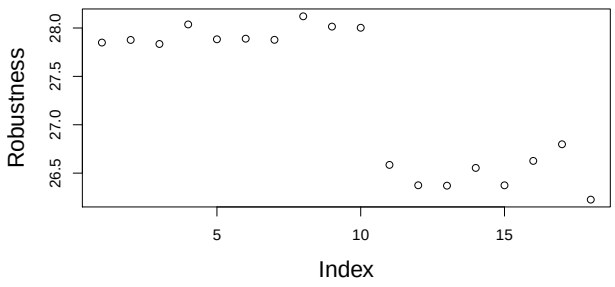

Figure 6: The robustness scores of ISA modules after filtering out some modules with low robustness. Ten modules have a higher score than the rest.

```
> plot(modules3$seeddata$rob, ylab="Robustness", cex.lab=1.5)
```

The scatterplot shows that ten modules have in fact higher robustness scores than the rest. Let us check that these are indeed the ten "real" modules of the data.

```
> bestmods <- order(modules3$seeddata$rob, decreasing=TRUE)[1:10]
> mod.cor <- pmin(cor(modules3$rows[,bestmods]!=0, data[[2]]),
                  cor(modules3$columns[,bestmods]!=0, data[[3]]))
> apply(mod.cor, 1, max)

 [1] 1 1 1 1 1 1 1 1 1 1

> apply(mod.cor, 2, max)

 [1] 1 1 1 1 1 1 1 1 1 1
```

Indeed, we have found the ten real modules, the ISA is both specific and sensitive. If one analyzes real data, then it is much more difficult to filter the modules found by the biclustering algorithm, but the robustness based filtering still helps getting rid of spurious modules.

Finally, let us take a look at some of the other modules, that were also kept after the robustness-based filtering. Here we plot five of them, together with the input matrix.

```
> othermods <- order(modules3$seeddata$rob)[1:5]
> print(plotModules(modules3, othermods, data=data[[1]]))
```

It turns out that these are double-modules, each is a union of two real modules. ISA finds these, since they also have correlated rows and columns, although the correlation is lower than for the single-modules. The double-modules are typically found at lower thresholds.

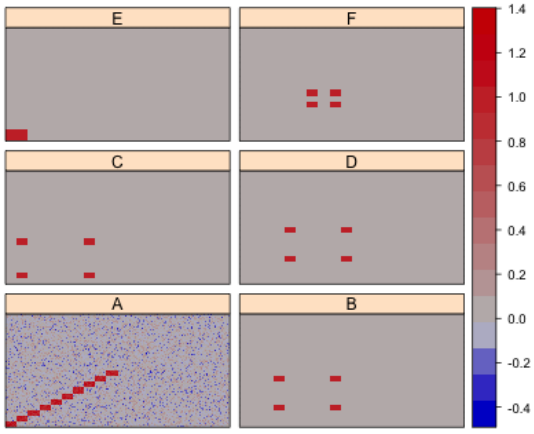

Figure 7: The input matrix and five modules with relatively low robustness scores. As it turns out, each of these modules contains two blocks of the input matrix.

5.4 Visualize 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, 2009].

5.4.1 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)
> five.mods <- isa.in.silico(200,200,5,20,20)
> modules <- isa(five.mods[[1]],thr.row=2, thr.col=2)
> Bc <- isa.biclust(modules)
> Bc

An object of class Biclust

call:
        NULL
```

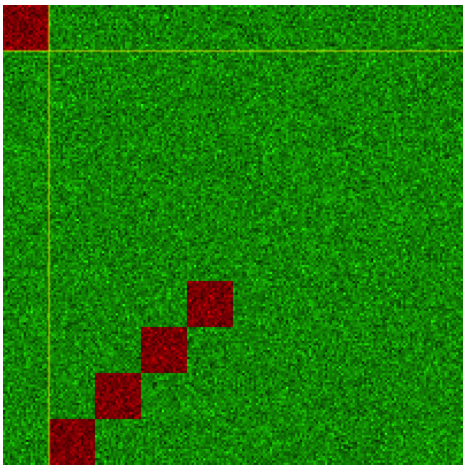

Figure 8: Heatmap plot of the reordered input matrix. The rows and columns corresponding to the first bicluster are moved to the top-left corner of the matrix.

```
Number of Clusters found:  5

First  5  Cluster sizes:
                  BC 1 BC 2 BC 3 BC 4 BC 5
Number of Rows:      20   20   20   20   20
Number of Columns:   20   20   20   20   20
```

5.4.2 Image plots

There are some examples in this document that show how to create image plots for the modules and potentially also the input data. See e.g. Figs 7 and 14. These plots use the `plotModules()` function, that directly takes the output of `isa()` (and other functions with the same output format: `isa.iterate()`, `isa.unique()`, etc.) In fact `plotModules()` calls the `images()` function to do its job. `images()` takes a list of matrices, see Fig. 11 for an example.

The `drawHeatmap()` function of the `biclust` package can be also used to draw an image plot. This visualizes a single bicluster on top of the input matrix. It reorders the rows and columns of the input matrix to move the block of the bicluster to the top left corner of the input matrix (Fig. 8).

```
> drawHeatmap(five.mods[[1]], Bc, number=1,local=FALSE)
```

5.4.3 Profile plots

The `parallelCoordinates()` function of the `biclust` package plots the profile of the rows (or columns) that are included in a bicluster with a different color.

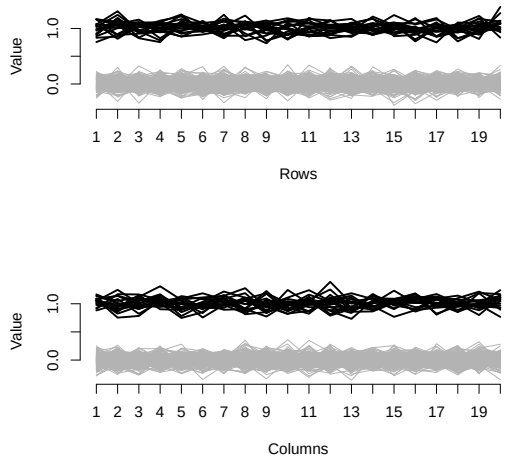

Figure 9: Profile plots for the first bicluster found by ISA. In this artificial data set there is a clear distinction between the background and the rows/columns of the module.

These plots visualize the difference between the bicluster and the rest of the matrix, see Fig. 9 for an example.

```
> parallelCoordinates(five.mods[[1]], Bc, number=1, plotBoth=TRUE)
```

5.4.4 Contrast bar plots

The `plotclust()` function of the `biclust` package creates barplots for one or more biclusters. A single bar is the mean of the columns of the bicluster for a given row of the bicluster; or the mean of the columns of the background (i.e. the rest of the input matrix). The bigger the difference between the bars of the two colors, the better the bicluster. The results of the following code are in Fig. 10.

```
> plotclust(Bc, five.mods[[1]], Titel="")
```

6 Features of ISA

In this section we show examples that highlight key features of the ISA: its resilience to noise and its ability to find overlapping modules.

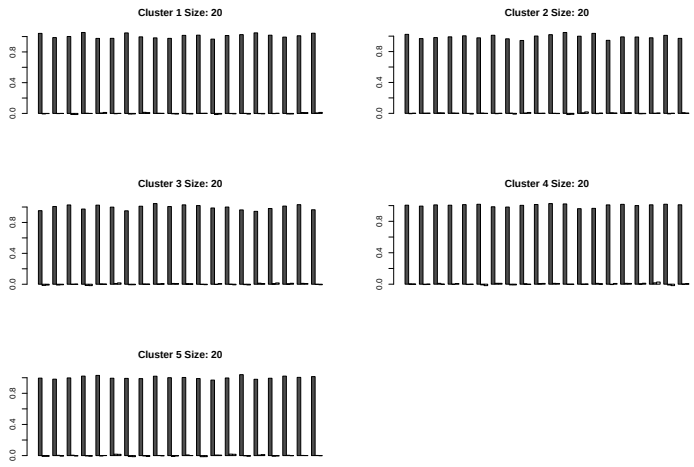

Figure 10: Bar plots show the difference between the biclusters and the rest of the input matrix, for the first five modules found by ISA.

6.1 Resilience to noise

To test the behavior of ISA with noisy inputs we generate a number of in-silico data sets with different noise levels (Fig. 11).

```
> noise <- seq(0.1,1,by=0.1)
> data <- lapply(noise, function(x) isa.in.silico(500,200,10,noise=x))

> images(lapply(data, "[[", 1), names=as.character(noise),
        xlab="", ylab="")
```

Next, we run ISA with the default parameters on all the data sets. This might take a while.

```
> modules <- lapply(data, function(x) isa(x[[1]]))
```

Let us check the sensitivity of ISA, for every real module, we pick one from the ones ISA has found, the one that is the most correlated to it.

```
> best <- lapply(seq_along(modules), function(i) {
   cc <- pmin(cor(modules[[i]]$rows!=0, data[[i]][[2]]),
              cor(modules[[i]]$columns!=0, data[[i]][[3]]))
   apply(cc, 2, max)
 })
> best.mean <- sapply(best, mean)
```

Let's create boxplots for the different noise levels. (Fig. 12.)

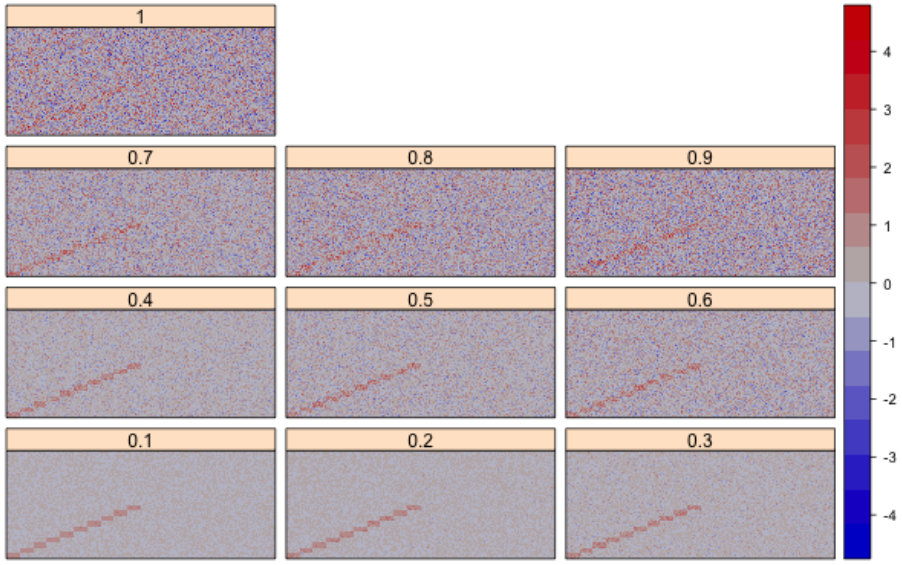

Figure 11: Modular data sets with different levels of noise. The data sets are generated by adding normally distributed background noise with given variance to the checkerboard matrix.

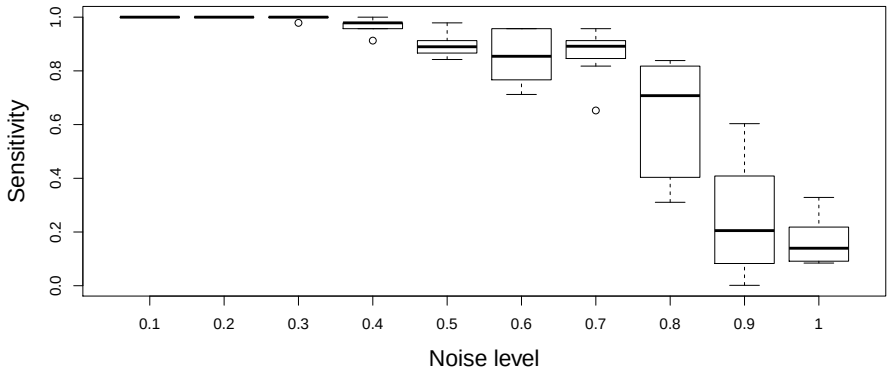

Figure 12: The sensitivity of the ISA in the function of noise. The sensitivity score is calculated by finding the best (in terms of Pearson correlation) ISA module for each block in the input matrix. Each boxplot is contains ten points, one for each module. ISA performs very well, even in the presence of relatively high noise levels.

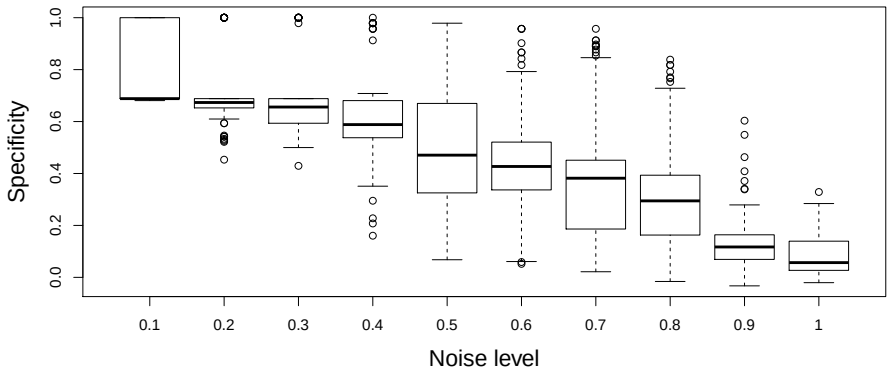

Figure 13: Specificity of the ISA modules, in the function of noise. Specificity was calculated by finding the best – again, by highest Pearson correlation – matching block in the input matrix for each ISA module. These correlation coefficients are plotted in a boxplot for each noise level. Specificity of ISA decreases approximately linearly with the increase of noise.

```
> boxplot(best, names=noise, xlab="Noise level", ylab="Sensitivity",
        cex.lab=1.5)
```

To test the specificity, we create boxplots for the correlation coefficients of all modules found by ISA and their closest real module. (Fig. 13.)

```
> spec <- lapply(seq_along(modules), function(i) {
   cc <- pmin(cor(modules[[i]]$rows != 0, data[[i]][[2]]),
              cor(modules[[i]]$columns != 0, data[[i]][[3]]))
   apply(cc, 1, max)
 })

> boxplot(spec, names=noise, xlab="Noise level", ylab="Specificity",
        cex.lab=1.5)
```

6.2 Finding overlapping biclusters

ISA biclusters might overlap in their rows, columns, or both. To illustrate this, we create an artificial data set with two overlapping blocks.

```
> set.seed(1)
> two.over <- isa.in.silico(100,100,2,40,40,overlap.row=10)
```

We run ISA on this input matrix, with two threshold parameters, the first thresholds are very mild, they are both zero. In other words, in each iteration

step all rows/columns are kept in each that have a higher score than the average. The second set of thresholds are stricter, the scores have to be at least one standard deviation away from the mean to keep them.

```
> ov.normed <- isa.normalize(two.over[[1]])
> ov.seeds <- generate.seeds(count=100, length=100)
> ov.modules.1 <- isa.iterate(ov.normed, ov.seeds, convergence="cor",
                              thr.row=0, thr.col=0, direction="up")
> ov.modules.1 <- isa.unique(ov.normed, ov.modules.1)
> ov.modules.1 <- isa.filter.robust(two.over[[1]], ov.normed, ov.modules.1)
> ov.modules.2 <- isa.iterate(ov.normed, ov.seeds, convergence="cor",
                              thr.row=1, thr.col=1, direction="up")
> ov.modules.2 <- isa.unique(ov.normed, ov.modules.2)
> ov.modules.2 <- isa.filter.robust(two.over[[1]], ov.normed, ov.modules.2)
> ncol(ov.modules.1$rows)

[1] 3

> ncol(ov.modules.2$rows)

[1] 3
```

ISA found 3 modules with the mild thresholds and 3 modules with the strict ones. Let us plot the original data and the modules found by ISA (Fig. 14).

```
> no.modules <- ncol(ov.modules.1$rows) + ncol(ov.modules.2$rows)
> plotModules(data=two.over[[1]],
              list(rows=cbind(ov.modules.1$rows, ov.modules.2$rows),
                   columns=cbind(ov.modules.1$columns,
                     ov.modules.2$column)),
              names=c("Input matrix", paste("Module", seq_len(no.modules))))
```

As expected the ISA modules are in general bigger with the mild thresholds. ISA correctly identifies the two overlapping modules and also the union of them. With the stricter thresholds it finds the non-overlapping parts of the two modules, plus their overlap as a separate module.

7 More information

For more information about the ISA please see the references at the end of this paper. The ISA homepage is at `http://www.unil.ch/cbg/homepage/software.html`.

For analyzing gene expression data with ISA, we suggest using BioConductor [Gentleman, 2004] and the `eisa` [Csárdi, 2009] R package.

If you want to run ISA on a computer cluster or a multi-processor machine, then see the vignette titled "Running ISA in parallel with the `snow` pacakge", in the `isa2` package.

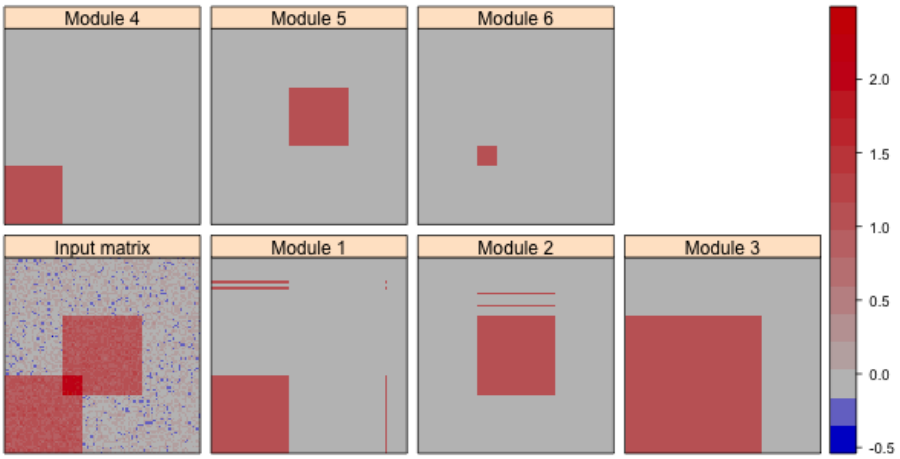

Figure 14: The input matrix with two overlapping blocks and the ISA modules for this data set. The first three modules were found at milder threshold parameters, that is why they are bigger. As you can see for Modules #1 and #2, at milder thresholds there is a higher probability that ISA will pick up some incorrect rows/columns.

8 Session information

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

- R version 2.15.0 (2012-03-30), `x86_64-apple-darwin9.8.0`
- Locale:
  `en_US.UTF-8/en_US.UTF-8/en_US.UTF-8/C/en_US.UTF-8/en_US.UTF-8`
- Base packages: base, datasets, graphics, grDevices, grid, methods, stats, utils
- Other packages: biclust 1.0.1, colorspace 1.1-1, isa2 0.3.1-1, lattice 0.20-6, MASS 7.3-18, Matrix 1.0-6
- Loaded via a namespace (and not attached): tools 2.15.0

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.

[Gentleman, 2004] R. Gentleman, V. J. Carey, D. M. Bates, B.Bolstad, M. Dettling, S. Dudoit, B. Ellis, L. Gautier, Y. Ge, and others (2004). Bioconductor: Open software development for computational biology and bioinformatics. *Genome Biology*, Vol. 5, R80

[Csárdi, 2009] Csárdi, G. (Apr 1, 2009). *isa2: The Iterative Signature Algorithm*. R package version 0.1.

[Csárdi, 2009] Csárdi, G. (Sep 15, 2009). *eisa: The Iterative Signature Algorithm for Gene Expression Data*. R package version 0.1.

[Ihmels, 2002] Ihmels, J., Friedlander, G., Bergmann, S., Sarig, O., Ziv, Y., Barkai, N. (2002). Revealing modular organization in the yeast transcriptional network. *Nat Genet* , page 370–7.

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

[Kaiser, 2009] Sebastian Kaiser, Rodrigo Santamaria, Roberto Theron, Luis Quintales and Friedrich Leisch. (2009). *biclust: BiCluster Algorithms*. R package version 0.7.2.