ISA internals

Gábor Csárdi

October 18, 2011

Contents

1 Introduction

The Iterative Signature Algorithm (ISA) is a biclustering method, it finds consistent blocks (modules, biclusters) in tabular data, e.g. gene expression measurements across a number of samples. Please see the introductory tutorials at the ISA homepage, `http://www.unil.ch/cbg/ISA`, and also [Bergmann et al., 2003] for details.

In this document we specifically deal with the implementation of the ISA in the `isa2` and `eisa` R packages.

2 Why two packages?

We implemented the ISA in two R packages, `isa2` and `eisa`. ISA is a very general algorithm, that can be used for any tabular data to find correlated blocks. Examples for such tabular data are gene expression [Ihmels and Bergmann, 2004],

response of different cell lines to a number of drugs [Kutalik et al., 2008]. It can also be used as a general classifier, for biological or other data.

It is also true, however, that the ISA is frequently used for gene expression analysis.

Thus, we decided to provide two different user interfaces to the ISA. One interface, provided by `isa2` package, is very general, the input of the `isa2` functions is a numeric matrix, and the output contains two matrices, defining the ISA modules. The `isa2` package can be used for the modular analysis of tabular data of any kind, from any source.

The `eisa` package (the leading 'e' stands for expression) provides a second interface, for people that specifically deal with gene expression data. This package builds on the infrastructure created by the BioConductor project [Gentleman et al., 2004]. The input of the `eisa` functions is an `ExpressionSet` object, the standard BioConductor data structure for storing gene expression data. BioConductor provides functions to create such an `ExpressionSet` object from raw data and to download data from public repositories, such as the Gene Expression Omnibus [Davis and Meltzer, 2007, Barrett et al., 2009] and transform it into an `ExpressionSet`. The output of the `eisa` functions is an object that contains the ISA modules, the annotation of the genes and samples in the data set and possibly also further experimental meta data. The `eisa` package provides functions for calculating enrichment statistics against various databases for the ISA modules. The `eisa` package uses already existing BioConductor annotation packages, so it works for any organism that is supported by BioConductor.

Having two ISA packages, however, does not mean two implementations of the ISA. The `eisa` package is fully built on top of the services of the `isa2` package, only the latter one contains the implementation of the ISA iteration.

The two packages allow ease of installation and use: users dealing with gene expression data install the `eisa` package, and this automatically installs the `isa2` package as well. Users analyzing other data install the `isa2` package only, this does not need any BioConductor packages.

The `isa2` package is part of the standard R package repository (CRAN), the `eisa` package has been accepted as an official BioConductor package and is included in BioConductor from the 2.6 release, due in April, 2010.

3 Speeding up the ISA iteration

ISA is an unsupervised, iterative, randomized algorithm. It starts with a seed vector. r_0. This vector is an initial guess for the rows of the input matrix that form a single module. This guess is then refined, by iterating itself and another vector, c_i, that defines the columns of the module.

During the iteration, the ISA uses two matrices, E_r and E_c, derived from the input matrix, by standardizing it row-wise and column-wise, respectively. One step of the ISA iteration involves (1) multiplying E_r by r_n and then (2) thresholding the result to keep elements that are further away from its mean

than a prescribed value, Θ_c, in standard deviation units. This gives c_{n+1}. Next, (3) E_c is multiplied by c_{n+1} and (4) the result is thresholded with Θ_r. This gives r_{n+1}.

The iteration is finished if r_{n+1} is close to r_n and c_{n+1} is close to c_n.

Considerable speedup can be achived, if the ISA iteration is performed in batches of seed vectors, instead of handling them individually. The reason for this is the availability of the highly optimized linear algebra libraries that perform matrix-matrix multiplication much faster than all the corresponding matrix-vector multiplications individually. The seed vectors can be merged into a seed matrix. This can be done, even if the ISA thresholds are different for the different seeds, the `isa2` and `eisa` packages support this.

We considered using sparse matrices during the ISA iteration, because the matrix of seed vectors is sparse, but according to our measurements, sparse matrices are only marginally faster, and only in some cases. The reason for this is, that the product of the two matrices is always dense, and gets sparse only after the thresholding; the dense-sparse matrix multiplication, plus the conversion to a sparse matrix again, takes about the same time as the dense-dense matrix multiplication.

Different input seeds converge in different number of steps, in parctice many seeds tend to converge quickly, and very few seeds need a lot of steps. Because of this, it is essential to remove the seeds that have already converged, from the seed matrix, so that only a smaller seed matrix needs to be iterated for many steps.

As loops are typically slow in R, we implemented the thresholding operations in C.

4 Running time analysis

In this section we show an experimental running time analysis for our ISA implementation.

4.1 The hardware and software

The code in the following sections were run under Linux operating system, release '`2.6.18-238.12.1.el5`', version '`#1 SMP Tue May 31 13:22:04 EDT 2011`', on an '`x86_64`' machine with 12 processors of type '`Intel(R) Xeon(R) CPU X5650 @ 2.67GHz`' and 94.28 GiB memory.

4.2 Getting the data

We use subsets of the same data set for the test, this is a data set that is publicly available in the Gene Expression Omnibus (GEO) repository, its id is GSE18858. Note, that for the analysis of gene expression data, the `eisa` package is a better choice. It makes sense, however, if this tutorial can be run without any BioConductor packages, so we will use the `isa2` package.

The non-normalized data set of the experiment is available from GEO as a numeric matrix. To spare time and bandwidth, we only download it once, even if the code of this tutorial is run multiple times. We store the data file in the current local directory. If the file is already there, then there is nothing to do.

```
> GEO <- "GSE18858"
> GEOfile <- paste(sep = "", GEO, "_series_matrix.txt.gz")
> GEOurl <- paste(sep = "", "ftp://ftp.ncbi.nih.gov/pub/geo/DATA/SeriesMatrix/",
     GEO, "/", GEOfile)
> if (!file.exists(GEOfile)) {
     download.file(GEOurl, GEOfile)
 }
```

Next, we are read in the data. Lines that are part of the file header start with an exclamation mark in GEO files, so we skip them by setting the `comment.char` argument of `read.table()`. The table has a header, the first lines are the sample names; it also has the probeset names in the first column, we use these as row names.

```
> data <- read.table(gzfile(GEOfile), comment.char = "!",
     header = TRUE, row.names = 1)
> data <- as.matrix(data)
```

Let's check the size of the data matrix.

```
> dim(data)
```

```
[1] 45101   242
```

It has 45101 rows (=probesets) and 242 columns.

We load the `isa2` package, to perform the ISA. No BioConductor packages are needed.

```
> library(isa2)
```

4.3 Measuring running time

We define a simple function first, that runs the various steps of the ISA toolchain and measures the running time of each step. We use the `system.time()` function for this. The results are returned in table.

```
> mesISA <- function(E, thr.row, thr.col, no.seeds) {
     t1 <- system.time({
         NE <- isa.normalize(E)
     })
     t2 <- system.time({
         seeds <- generate.seeds(length = nrow(E),
             count = no.seeds)
```

```
    })
    t3 <- system.time({
        mods <- isa.iterate(NE, row.seeds = seeds,
            thr.row = thr.row, thr.col = thr.col)
    })
    t4 <- system.time({
        mods2 <- isa.unique(NE, mods)
    })
    cbind(normalization = t1, seeds.generation = t2,
        isa.iteration = t3, module.merge = t4,
        full = t1 + t2 + t3 + t4)
}
```

We quickly test this function, on a small subset of our data, with 1000 rows and 30 columns, chosen randomly.

```
> mydata <- data[sample(nrow(data), 1000), sample(ncol(data),
    30)]
> mesISA(mydata, 3, 3, 100)

           normalization seeds.generation isa.iteration
user.self          0.026            0.002         0.163
sys.self           0.000            0.000         0.000
elapsed            0.026            0.001         0.169
user.child         0.000            0.000         0.000
sys.child          0.000            0.000         0.000
           module.merge  full
user.self         0.008 0.199
sys.self          0.000 0.000
elapsed           0.008 0.204
user.child        0.000 0.000
sys.child         0.000 0.000
```

The table has one column for each step of the ISA toolchain: normalization, seed generation, performing the ISA iteration, merging the modules, and there is a column for the total time of these operations as well. The first row shows the processor time used by process itself, the second row the time spent in system calls. In the following we will use the total of both of them to measure the speed of the implementation.

4.4 Number of rows and columns

First, we increase the number of rows in the data matrix gradually, and measure the running time of ISA. We do this for variuos (row) thresholds, to see if the trend is threshold-dependent. The column threshold will be fixed now.

```
> row.thresholds <- seq(1, 3, by = 0.5)
> col.thresholds <- 2
```

We create a function, `do.row.thr()`, that runs the ISA for fixed threshold parameters, and different number of rows in the data matrix. The function also does replications, 5 by default.

```
> no.rows <- seq(5000, min(40000, nrow(data)), by = 5000)
> do.row.thr <- function(thr, rep = 5) {
     res <- lapply(no.rows, function(x) {
         lapply(1:rep, function(xxx) {
             mydata <- data[sample(nrow(data),
                 x), ]
             mesISA(mydata, thr, col.thresholds,
                 100)
         })
     })
     res
 }
```

We are ready to do the running time measurement now; separately for each row threshold parameter. This takes about three hours to run, on the platform mentioned above.

```
> by.rows <- lapply(row.thresholds, do.row.thr)
```

Next, we define a function to plot the results, with error bars. Error bars are not supported by the builtin R plotting functions, so we put them together from line segments.

```
> myplot <- function(x, y, sd, xlim = range(x),
     ylim = c(min(y - sd), max(y + sd)), xlab = "",
     ylab = "", ...) {
     plot(NA, type = "n", xlim = xlim, ylim = ylim,
         xlab = xlab, ylab = ylab)
     xmin <- par("usr")[1]
     xr <- par("usr")[2] - xmin
     bw <- xr/200
     segments(x, y - sd, x, y + sd)
     segments(x - bw, y - sd, x + bw, y - sd)
     segments(x - bw, y + sd, x + bw, y + sd)
     points(x, y, ...)
 }
```

The following two functions calculate the mean and standard deviation of the running times in the result lists, we will use them later.

```
> get.mean <- function(xx) {
     sapply(xx, function(x) mean(sapply(x, function(y) sum(y[1:2,
         5]))))
 }
```

```
> get.sd <- function(xx) {
     sapply(xx, function(x) sd(sapply(x, function(y) sum(y[1:2,
         5]))))
 }
```

We are ready to create a plot, the running times in the function of the number of rows in the input matrix. The results can be seen in Fig. 1.

```
> layout(rbind(1:2, 3:4, 5:6))
> for (i in 1:length(row.thresholds)) {
     par(mar = c(5, 4, 1, 1) + 0.1)
     y <- get.mean(by.rows[[i]])
     s <- get.sd(by.rows[[i]])
     myplot(no.rows, y, s, type = "b", pch = 20,
         xlab = "# of rows", ylab = "running time [s]")
     rt <- row.thresholds[i]
     text(min(no.rows), max(y + s), substitute(Theta[r] ==
         rt, list(rt = rt)), adj = c(0, 1), cex = 1.3)
 }
```

In the following, we perform a similar analysis for the number of columns in the input matrix. ISA is a symmetric algorithm, rows are treated the same way as columns. The reason for discussing them separately here, is that gene expression matrices have usually much more rows than columns and this difference might affect the running time.

For these runs, the row threshold is fixed and the column threshold changes from 1 to 3.

```
> row.thresholds2 <- 2
> col.thresholds2 <- seq(1, 3, by = 0.5)
```

We create a function to perform all the runs for a given column threshold, with replications five by default.

```
> no.cols <- seq(30, ncol(data), by = 30)
> do.col.thr <- function(thr, rep = 5) {
     res <- lapply(no.cols, function(x) {
         lapply(1:rep, function(xxx) {
             mydata <- data[, sample(ncol(data),
                 x)]
             mesISA(mydata, row.thresholds2, thr,
                 100)
         })
     })
     res
 }
```

We are ready to do the measurements. On the above mentioned hardware configuration, this takes about 4 hours to run, depending on the load of the system.

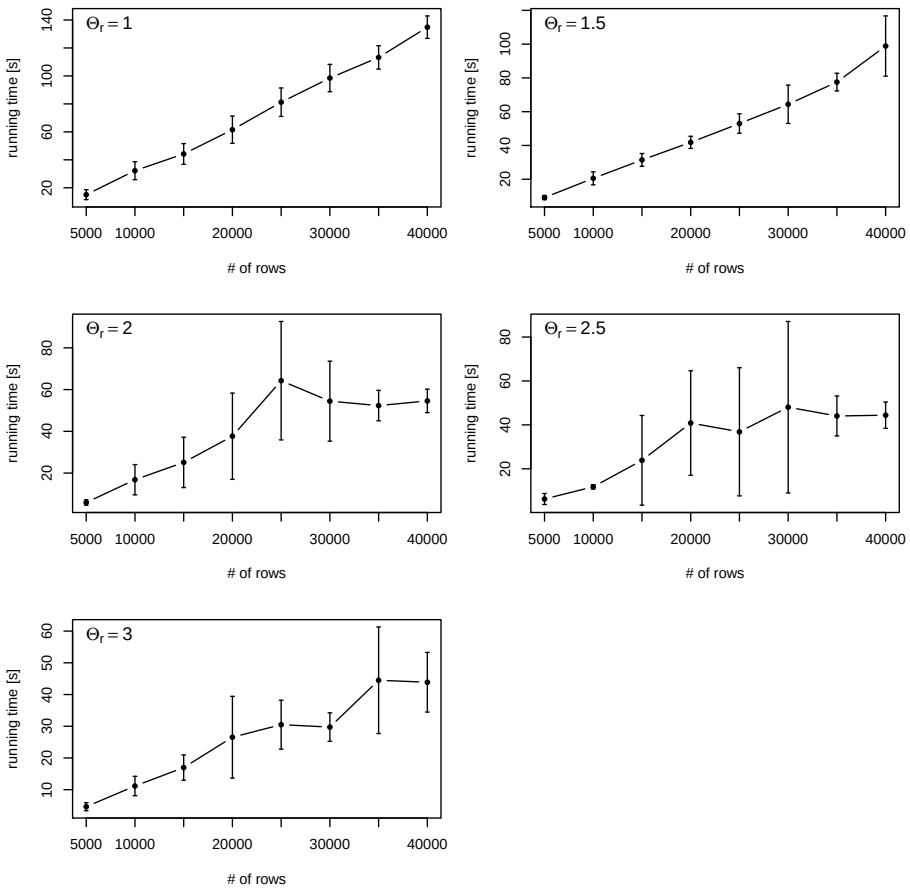

Figure 1: Running time of the ISA, in the function of the number of rows in the input matrix, for various row thresholds. Crearly, the running time increases linearly with the number of rows, for all row thresholds. It is also true, that for higher thresholds the running times tend to be smaller, this is because in these cases more seeds converge quickly to the null vector, and these don't need to be iterated further. Each data point is the mean of five runs.

```
> by.cols <- lapply(col.thresholds2, do.col.thr)
```

The mean running times are plotted, the results are shown in Fig. 2.

```
> layout(rbind(1:2, 3:4, 5:6))
> for (i in 1:length(col.thresholds2)) {
     par(mar = c(5, 4, 1, 1) + 0.1)
     y <- get.mean(by.cols[[i]])
     s <- get.sd(by.cols[[i]])
     myplot(no.cols, y, s, type = "b", pch = 20,
         xlab = "# of cols", ylab = "running time [s]")
     rt <- col.thresholds2[i]
     text(min(no.cols), max(y + s), substitute(Theta[c] ==
         rt, list(rt = rt)), adj = c(0, 1), cex = 1.3)
 }
```

4.5 Number of seeds

Finally, we also check the running time in the function of the number of starting seeds.

We define three threshold configurations to test. The first has intermediate thresholds for both the rows and the columns, the second has a low threshold for the rows and a high threshold for the columns, the third is the opposite of the second.

```
> thr.comb <- list(c(2, 2), c(1, 3), c(3, 1))
> no.seeds <- seq(50, 400, by = 50)
```

The following function does all the runs for a given threshold combination. The size of the data matrix is fixed here, 20000 times 100.

```
> do.no.seeds <- function(thr, rep = 5) {
     nr <- min(20000, nrow(data))
     nc <- min(100, ncol(data))
     res <- lapply(no.seeds, function(x) {
         lapply(1:rep, function(xxx) {
             mydata <- data[sample(nrow(data),
                 nr), sample(ncol(data), nc)]
             mesISA(mydata, thr[1], thr[2], x)
         })
     })
     res
 }
```

We are ready to do the measurement now.

```
> by.no.seeds <- lapply(thr.comb, do.no.seeds)
```

The results are plotted in Fig. 3.

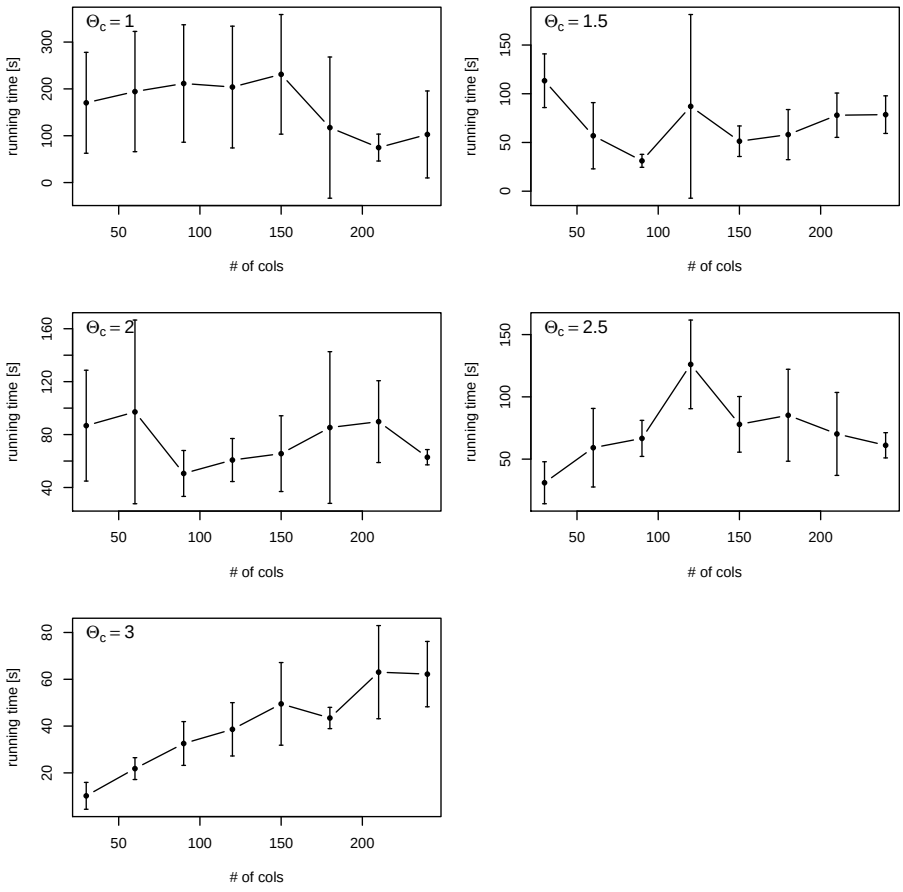

Figure 2: Running times, in the function of the number of columns in the input matrix, for various column thresholds. Interestingly, the running time can be considered as independent of the number of columns. This is simply because the row seed matrix is two orders of magnitude larger than the column seed matrix, and the former dominates the running time. Each data point is the mean of five runs.

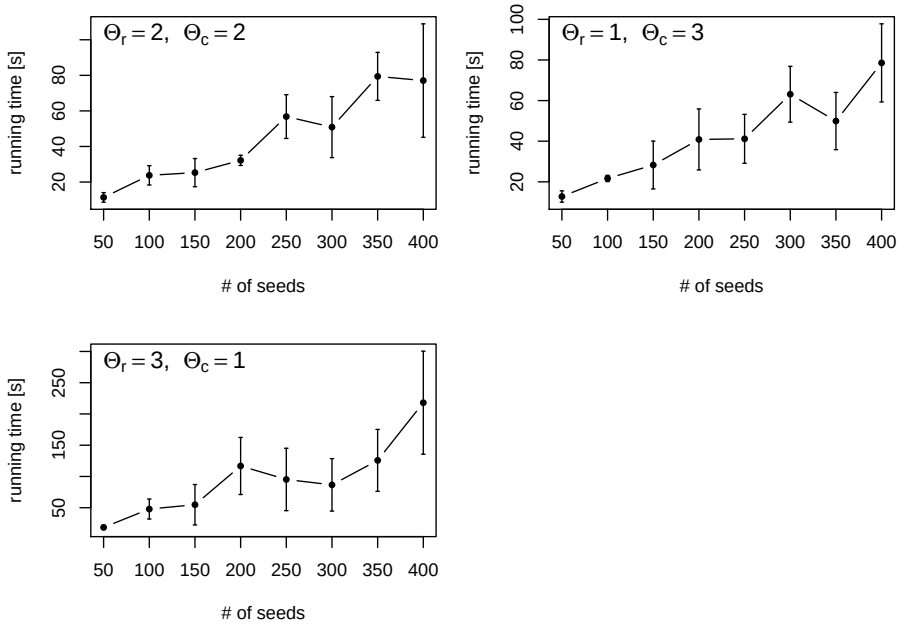

Figure 3: The running time, in the function of the number of starting seeds, for various row and column threshold combinations. Each point is the average of five ISA runs. The running time is increasing linearly with the number of seeds.

```
> layout(rbind(1:2, 3:4))
> for (i in 1:length(thr.comb)) {
     par(mar = c(5, 4, 1, 1) + 0.1)
     y <- get.mean(by.no.seeds[[i]])
     s <- get.sd(by.no.seeds[[i]])
     myplot(no.seeds, y, s, type = "b", pch = 20,
         xlab = "# of seeds", ylab = "running time [s]")
     th <- thr.comb[[i]]
     text(min(no.seeds), max(y + s), substitute(paste(Theta[r] ==
         r1, ",  ", Theta[c] == r2), list(r1 = th[1],
         r2 = th[2])), adj = c(0, 1), cex = 1.3)
 }
```

5 Running ISA in parallel

It is trivial to run an ISA analysis in parallel, on a multi-processor machine, or a computer cluster: one just runs different threshold-combinations and/or

different seeds on the different processors. The achived speedup is close to linear, since the ISA iteration step dominates in the toolchain. Please see more about this on the ISA homepage at `http://www.unil.ch/cbg/ISA`.

6 Session information

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

- R version 2.13.0 (2011-04-13), `x86_64-unknown-linux-gnu`
- Locale: `LC_CTYPE=en_US.UTF-8`, `LC_NUMERIC=C`, `LC_TIME=en_US.UTF-8`, `LC_COLLATE=en_US.UTF-8`, `LC_MONETARY=C`, `LC_MESSAGES=en_US.UTF-8`, `LC_PAPER=en_US.UTF-8`, `LC_NAME=C`, `LC_ADDRESS=C`, `LC_TELEPHONE=C`, `LC_MEASUREMENT=en_US.UTF-8`, `LC_IDENTIFICATION=C`
- Base packages: base, datasets, graphics, grDevices, methods, stats, utils
- Other packages: isa2 0.3.1

References

[Barrett et al., 2009] Barrett, T., Troup, D., Wilhite, S., Ledoux, P., Rudnev, D., Evangelista, C., Kim, I., Soboleva, A., Tomashevsky, M., Marshall, K., Phillippy, K., Sherman, P., Muertter, R., and R, E. (2009). NCBI GEO: archive for high-throughput functional genomic data. *Nucleic Acids Research*, 37:D885–90.

[Bergmann et al., 2003] Bergmann, S., Ihmels, J., and Barkai, N. (2003). Iterative signature algorithm for the analysis of large-scale gene expression data. *Physical Review E*, 67:031902.

[Davis and Meltzer, 2007] Davis, S. and Meltzer, P. (2007). GEOquery: a bridge between the Gene Expression Omnibus (GEO) and BioConductor. *Bioinformatics*, 14:1846–1847.

[Gentleman et al., 2004] Gentleman, R. C., Carey, V. J., Bates, D. M., et al. (2004). BioConductor: Open software development for computational biology and bioinformatics. *Genome Biology*, 5:R80.

[Ihmels and Bergmann, 2004] Ihmels, J. H. and Bergmann, S. (2004). Challenges and prospects in the analysis of large-scale gene expression data. *Brief Bioinform*, 5(4):313–327.

[Kutalik et al., 2008] Kutalik, Z., Beckmann, J. S., and Bergmann, S. (2008). A modular approach for integrative analysis of large-scale gene-expression and drug-response data. *Nat Biotechnol*, 26(5):531–539.