R/EBcoexpress: An Empirical Bayesian Framework for Discovering Differential Co-expression

John A. Dawson, Shuyun Ye and Christina Kendziorski

October 13, 2014

Contents

1 Introduction

A common goal of microarray and related high-throughput genomic experiments is to identify genes that vary across biological condition. Most often this is accomplished by identifying genes with changes in mean expression level, so called differentially expressed (DE) genes, and a number of effective methods for identifying DE genes have been developed. Although useful, these approaches do not accommodate other types of differential regulation. An important example concerns differential co-expression (DC). Investigations of this class of genes are hampered by the large cardinality of the space to be interrogated as well as by influential outliers. As a result, existing DC approaches are often underpowered, exceedingly prone to false discoveries, and/or computationally intractable for even a moderately large number of pairs.

This package implements an empirical Bayesian approach for identifying DC gene pairs. The approach provides a false discovery rate (FDR) controlled list of significant DC gene pairs without sacrificing power. It is applicable within a single study as well as across multiple studies. Computations are greatly facilitated by a modification to the EM algorithm and a procedural heuristic, as discussed below. Details of the approach are given in Dawson and Kendziorski (2011).

2 Single-study setup

We assume a user has normalized expression levels profiled from m genes in n subjects, where the subjects are partitioned into K conditions. Of primary interest in this analysis is identifying gene pairs for which the true underlying, or latent, correlation within condition varies across conditions. We denote such a latent correlation in condition k by λ^k. When $K = 2$ conditions are being considered, we say a gene pair may be equivalently co-expressed (EC; $\lambda^1 = \lambda^2$) or differentially co-expressed (DC; $\lambda^1 \neq \lambda^2$). When $K = 3$, there are 4 ways in which DC may arise: $\lambda^1 \neq \lambda^2 = \lambda^3$; $\lambda^2 \neq \lambda^1 = \lambda^3$; $\lambda^3 \neq \lambda^1 = \lambda^2$; and a case where λ^1, λ^2 and λ^3 are all distinct. We refer to these cases as DC classes. The methods developed in Dawson and Kendziorski (2011) evaluate the posterior probability of each EC/DC class for each of $m * (m-1)/2$ gene pairs. The posterior probabilities may be thresholded to provide a false discovery rate (FDR) controlled list of DC gene pairs (section 3.6). In section

3.7, we discuss limitations on m. Roughly speaking, analysis proceeds most smoothly if m does not exceed 10,000 genes (about 50M gene pairs).

3 Single-study analysis

3.1 Required inputs

A single study analysis requires three inputs:

X An m-by-n matrix of expression values, where m is the number of genes (or probes) under consideration and n is the total number of microarrays over all conditions. These values should be normalized in some manner; we suggest background normalization but not quantile normalization of the microarrays, as the latter destroys correlation structure, and instead suggest median correction so all arrays have the same median expression. While not required, expression is generally on the log_2 scale.

The conditions array An array of length n. The members of this array should take values in $1, \ldots, K$ where K is the total number of conditions (usually $K = 2$). All microarrays in condition 1 should have value 1 in the array, and so on; these should be in the same order as the n columns of X.

The pattern object An ebarraysPatterns object resulting from a call to ebPatterns(), found in the R/EBarrays package. This is used to define the EC/DC classes and is based on the unique values in the conditions array. The example below should suffice for $K = 2$; for $K > 2$, see the documentation for ebPatterns().

Our demo uses a small example data set, which can be accessed using the code given below. There are fifty genes, in two sets of twenty-five. Genes in the first set are DC among themselves, genes in the second set are also DC among themselves, but the two groups are uncorrelated in both conditions, so that all pairs involving a gene from each set are EC. The first condition has 100 samples while the expression values in the second condition were generated using only 25 samples. We note that this is a somewhat weakly powered setup, but the difference will be useful when we illustrate the use of the diagnostic function.

```
> library(EBcoexpress)
> data(fiftyGenes) # A 50-by-125 expression matrix
> tinyCond <- c(rep(1,100),rep(2,25))
> tinyPat <- ebPatterns(c("1,1","1,2"))
```

3.2 Making the D matrix of correlations

From X and the conditions array, we need to calculate intra-group correlations for all $p = m * (m-1)/2$ gene pairs. This can be done many ways, but our implementation uses either the usual Pearson's correlation coefficient or biweight midcorrelation, which can be thought of as a robust version of Pearson's correlation coefficient. In our own analyses we have preferred biweight midcorrelation while noting that Pearson's correlation coefficient is much faster to compute for large m. Regardless, this step is accomplished through the function makeMyD():

```
> D <- makeMyD(fiftyGenes, tinyCond, useBWMC=TRUE)
```

Here biweight midcorrelation is used since the useBWMC= option is set to TRUE. The resulting D matrix of correlations is p-by-K, with gene pair names given in the form GENE1$\sim$GENE2; $\sim$ is the default separator but this can be changed. Note that the D matrix in the demo has pair names such as X1$\sim$X28.

3.3 Initializing hyperparamters

If you've taken a moment to read the methods section of Dawson and Kendziorski (2011), then you're aware that our model uses a mixture of Normals as a prior in its computations. We take an empirical Bayesian approach, so hyperparameters are estimated from the data. We'll do that presently, but first we need some initial values of these parameters as a starting point. The better our initial estimates, the faster the EM will converge. There are four sets of hyperparameters that need to be initialized:

The Component Number G The number of Normals in the mixture; in our model, $G = 1, 2$ or 3.

The MUS $\{\mu_g\}$ The means of the mixture components, ordered from smallest to largest. Since the model works on transformed as opposed to raw correlations, these are just real numbers.

The TAUS $\{\tau_g\}$ The standard deviations of the mixture components, which must be positive.

The WEIGHTS $\{\omega_g\}$ The weights of the mixture components, which sum to one.

While the user is free to do this any manner s/he deems fit, our suggested method of initialization uses the Mclust algorithm (Fraley and Raftery, 2002, 2006), accessed through our initializeHP() function:

```
> set.seed(3)
> initHP <- initializeHP(D, tinyCond)
```

The initializeHP() function initializes the hyperparameters by asking Mclust to find the 1-, 2- or 3- component Normal mixture model that best fits the correlations of D after transformation (see section 3.7), ignoring condition. Mclust directly returns estimates for G and the MUS; the TAUS are approximated using Mclust's point estimates and sample sizes. The WEIGHTS are estimated using the mixture component classifications Mclust provides; however, it is unclear whether those classifications should be weighted by how confident Mclust is of their accuracy. Thus, initializeHP() tries both approaches (i.e., the Weighted and Unweighted approaches), compares the two model fits through deviance approximations and returns the set of WEIGHTS that best fits the data.

In the event that the TAUS are estimated to be less than 0, a different estimation scheme for the TAUS is used, relying on division instead of subtraction to enforce positivity. This happens on occasion, but should not cause alarm; it simply means that the initial variance estimates fed into the EM will likely be somewhat inflated. As such, we do not recommend the use of ebCoexpressZeroStep() when this occurs (see section 3.4).

Let's take a look at initHP:

```
> print(initHP)

$G
[1] 3

$MUS
           2            1            3
-0.006557837  0.151097168  0.585087753

$TAUS
[1] 0.05314046 0.16175079 0.04716496

$WEIGHTS
[1] 0.5955102 0.2306122 0.1738776
```

In this example, Mclust has identified a three-component Normal mixture as best fitting the empirical distribution of correlations, with component means, standard deviations and weights as shown. These values will be used to initialize the EM algorithm.

Speed Up! When p is very large, it can take a while for Mclust to do its job. Since we are trying to estimate at most ten parameters from $p \times K >> 10$ observations, it is often useful to tell Mclust to only use some of the p pairs in its calculations. This is done by setting the subsize= option in initializeHP() to some number less than p (we suggest $p/1000$). You may also want to use the seed= option to make this process deterministic.

3.4 EM computations

For this task, one of the ebCoexpress series of functions should be used. Our example is small enough that we can run all three, but the full version will still take a few minutes. From the demo:

```
> zout <- ebCoexpressZeroStep(D, tinyCond, tinyPat, initHP)

Zero-Stepper Time: 0.223

> oout <- ebCoexpressOneStep(D, tinyCond, tinyPat, initHP)
```

```
Begin Phase I (Initial E-Step) ...
Begin Phase II (M2-Step) ...
Begin Phase III ([E M1] Cycle) ...
Iteration: 1
One-Stepper Time: 22.99
```

```
> fout <- ebCoexpressFullTCAECM(D, tinyCond, tinyPat, initHP)
```

These three functions represent different flavors of the modified EM approach we use, the TCA-ECM (for complete details, see Dawson and Kendziorski (2011)). Starting from the bottom up, the third function (the 'full' version) will run a complete TCA-ECM. The second function (the 'one-step' version) will perform a single iteration of the TCA-ECM and return the results. Lastly, the first function (the 'zero-step' version) does not perform any EM calculations and instead uses the initial estimates of the hyperparameters (e.g., those obtained from initializeHP() or some other method) to generate posterior probabilities of DC. Obviously the zero-step version is faster than the one-step version, which in turn is faster than the full TCA-ECM. We have found that the one-step version has about the same accuracy as the full approach but executes in a fraction of the time, while the performance of the zero-step version greatly depends on the quality of the initial hyperparameter estimates. Therefore, in general we suggest the one-step version be used.

The output from these functions is a structured list with two named elements, MODEL and POSTPROBS:

MODEL is a list containing an array MIX and a list HPS. MIX contains estimated mixing proportions for the EC/DC classes. HPS contains final estimates of the hyperparameters we introduced earlier: G, MUS, TAUS and WEIGHTS. The list of lists required by ebCoexpressMeta() (see section 4) is obtained by combining the separate analyses' HPS lists together in one big list.

POSTPROBS is a p-by-L matrix containing posterior probabilities of EC and DC over all L EC/DC classes. The EC posterior probabilities will always be in the first column (which should be fed into crit.fun() if using the soft threshold, see section 3.6). Total posterior probabilities of DC for each gene pair are found by summing over the other $L-1$ columns (or taking 1 minus the first column).

Let's pull off the POSTPROBS matrices, as we'll use them later:

```
> result0 <- zout$POSTPROBS
> result1 <- oout$POSTPROBS

> resultF <- fout$POSTPROBS
```

Speed Up! When p is very large, it can take a while for the one-step and full TCA-ECM versions to complete their computations. Since we are trying to estimate at most $L+10$ parameters from $p \times L >> L+10$ observations, it is often useful to tell the EM to only use some of the p pairs when estimating the hyperparameters. This is done by setting the subsize= option in the controlOptions list to some number less than p (we suggest $p/1000$); see the documentation for further details. Since the sample size far exceeds the number of parameters being estimated, and this particular portion of the EM is computationally expensive, taking this approach will have a miniscule effect on the accuracy of the estimates but a big effect on runtime.

3.5 Checking the prior

After the selected EM function has finished its computations, it is desirable to take a moment to check and see how well the model chosen by the EM fits the data. Compared to other analyses in other paradigms (e.g., DE analyses) we are somewhat limited in what diagnostics can be performed. For instance, we have only one observation for each pair's correlation in each condition, and 'n=1' does not lend itself to proper diagnostics.

We can, however, check the fit of the prior, using the prior predictive distribution. Given the hyperparameters estimated by the EM, there is a theoretical (but condition-dependent) form for this distribution and we can compare it to an empirical estimate in each condition. This is done visually using the priorDiagnostic() function:

```
> priorDiagnostic(D, tinyCond, zout, 1)

> priorDiagnostic(D, tinyCond, zout, 2)

> priorDiagnostic(D, tinyCond, oout, 1)

> priorDiagnostic(D, tinyCond, oout, 2)
```

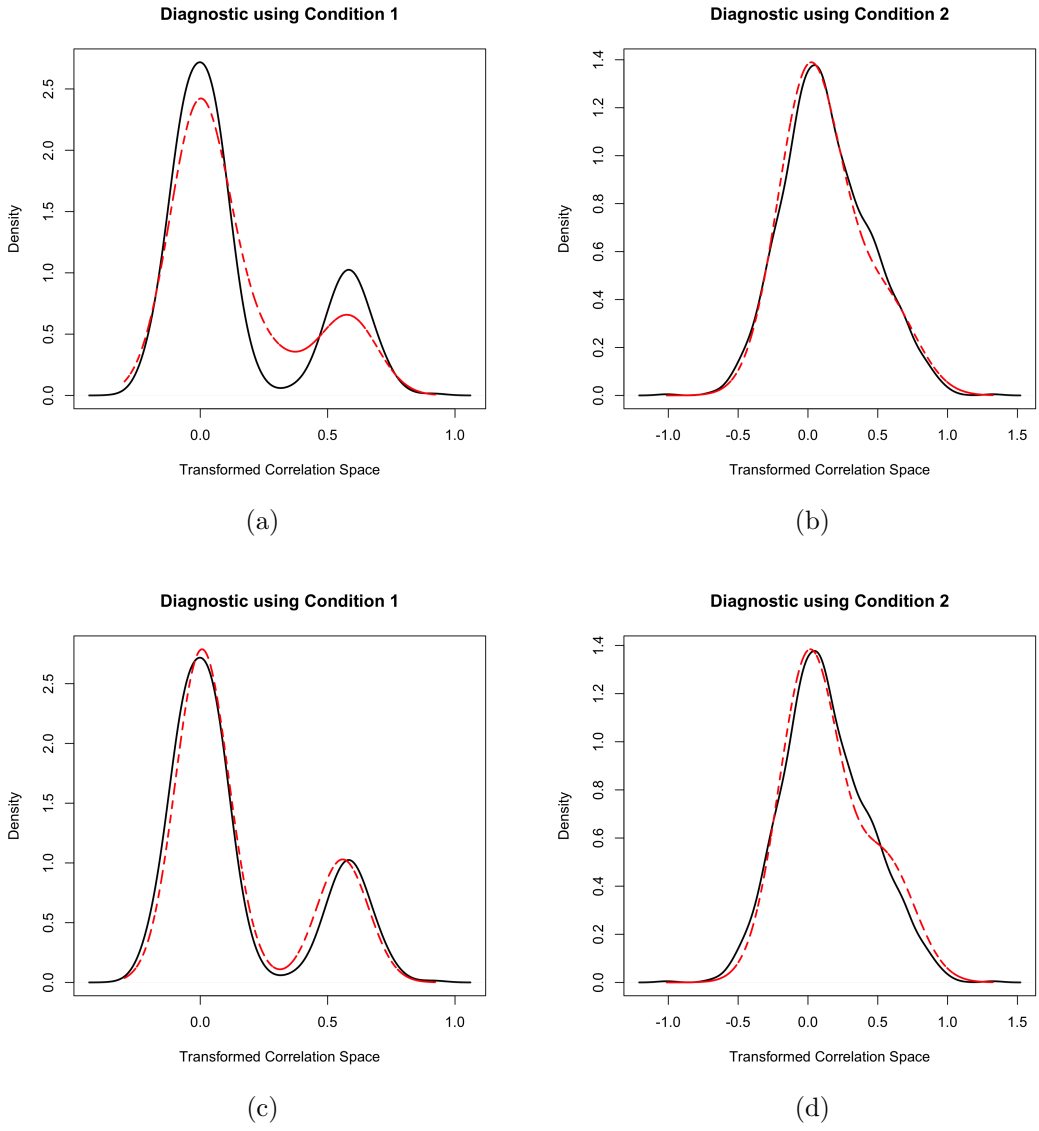


Figure 1: Four diagnostic plots using the prior predictive distribution. The top panels are for the inferior zero-step fit, which eschews running an EM, and the bottom panels are for the superior one-step fit. In both pairs of panels, condition 1 is on the left and condition 2 is on the right. The empirical and theoretical prior predictive distributions are in black and red, respectively.

In our example, both the zero-step and the one-step priors are decent fits, but the one-step is clearly superior. The full TCA-ECM's diagnostics are practically identical to the one-step and are not shown here, although there is code for them in the demo. Note that the form of the prior predictive distribution is condition-dependent, specifically changing along with the number of chips in each condition.

If the prior diagnostic indicates a poor fit and G is one or two, the initialization may be to blame. We will deal with this issue in section 3.7.

3.6 Identifying DC pairs

Let's consider the full TCA-ECM results. There are two ways that one could threshold the total posterior probabilities of DC in order to identify DC gene pairs while targeting some desired false discovery rate (FDR). The simplest method is to use a hard threshold, EC ≤ 0.05 for 5% FDR control. In a two group analysis, this is equivalent to identifying those pairs for which the posterior probability of DC exceeds 0.95; but when more than two groups are considered, this is not the case, and initial thresholding should be done on the posterior probabilities of EC, not DC. Hard thresholds are often overly conservative and are technically not required to control FDR at a given level. The crit.fun() function can be used to provide a soft threshold and simulations suggest that the soft threshold is desirable when the model fits well. However, the soft threshold can produce empirically high FDR when this is not the case, and consequently it should be used with caution if the visual diagnostic presented in section 3.5 is less than stellar.

```
> ppbDC1 <- 1-result1[,1]
> crit_s <- crit.fun(result1[,1], 0.05)
> kept_s <- ppbDC1[ppbDC1 >= crit_s]
> kept_h <- ppbDC1[ppbDC1 >= 0.95]
> klabs_s <- names(kept_s)
>   # DC pair names, under soft thresholding
> klabs_h <- names(kept_h)
>   # DC pair names, under hard thresholding
```

Since this is a simulated data set, we know which gene pairs are truly DC; in the demo these are listed in the TP object. Thus, observed FDR and power

can be calculated and we do so below for the one-step TCA-ECM version; results for the full and zero-step versions are similar. Recall that, due to the low sample size in condition 2, this is an underpowered data set, as can be seen below. The FDR control, however, is excellent.

```
> nk_s <- length(kept_s) # No. taken as DC (soft)
> nk_h <- length(kept_h) # No. taken as DC (hard)
> nY_s <- sum(klabs_s %in% TP)
>   # Number of TP taken as DC under soft thresholding
> nY_h <- sum(klabs_h %in% TP)
>   # Number of TP taken as DC under soft thresholding
>
> (nk_s - nY_s)/nk_s           # Soft threshold Obs. FDR

[1] 0.004504505

> (nk_h - nY_h)/nk_h           # Hard threshold Obs. FDR

[1] 0

> nY_s/numDC                   # Soft threshold Obs. Power

[1] 0.3683333

> nY_h/numDC                   # Hard threshold Obs. Power

[1] 0.2133333
```

For those interested in knowing what genes are involved in the most number of genes deemed to be DC, we have a function called rankMyGenes(). It uses the EM output (specifically the POSTPROBS element) and a threshold to return a sorted, named list of gene counts within the DC genes. The threshold is by default a hard threshold of 5%, but the user may use another.

```
> hubs <- rankMyGenes(oout)
> print(hubs)

allNames
 X6 X17  X3  X5 X11 X12 X19 X21 X25  X4 X18 X22 X24  X1 X10
 14  10  10  10   9   9   9   9   9   9   8   8   8   7   7
X37  X8 X13  X2 X31 X33 X42 X45  X9 X14 X15 X20 X26 X28  X7
  7   7   6   6   6   6   6   6   6   5   5   5   5   5   5
X23 X36 X38 X46 X16 X34 X35 X43 X44 X49 X29 X32 X40 X48
  4   4   4   4   3   3   2   2   2   2   1   1   1   1
```

3.7 Some caveats

3.7.1 Restrictions on m

As mentioned earlier, the algorithm does not run as smoothly when m is large. First off, the algorithms are roughly $O(p) = O(m^2)$, so runtime increases quadratically with the number of genes. Furthermore, the multiplicative constants for those big-O order baselines also increase in a non-linear manner with m, due to issues involving memory. So, in general we suggest:

- No more than 10,000 genes be run at once; the algorithm is no worse than quadratic in m when the user stays below 5,000-8,000; and
- Use the subsize= options where available to cut down on needless computation and improve runtime

3.7.2 Extremely high correlations

The model assumed by our approach involves Normal mixtures with support on the real line, and as such performs Fisher's Z-transformation (Fisher, 1928) on the correlations:

$$f_Z(\rho) = 0.5 \times log\left(\frac{1+\rho}{1-\rho}\right)$$

This changes the working domain from [-1, 1] to the real line and has some other nice properties, as outlined in Dawson and Kendziorski (2011). One side-effect of this transformation is that it treats differences in correlation differently depending on where they fall within [-1, 1]. While the Z-transformation is roughly linear on [-0.5, 0.5], as a correlation ρ approaches either -1 or 1, $f_Z(\rho)$ approaches $-\infty$ or ∞. Thus, the transformed difference between 0.9 and 0.99 is roughly the same as that between 0.99 and 0.999; more significantly, it is the same as between 0 and 0.8.

This phenomenon can cause pairs of exceedingly correlated genes to be deemed DC, due to slight variation from condition to condition, especially when sample sizes are small. If this is a concern to the user, s/he may filter these gene pairs out a posteriori, using the D matrix as a guide.

3.7.3 Insufficient prior components

On occasion, the prior diagnostic may look bad in one or more conditions. If this appears to be due to the prior being too simple (e.g., the prior has only one mixture component and is hence a univariate Normal density, while the empirical prior predictive density appears to be quite bumpy) this may be remedied by rerunning the EM with a more complex prior. Since initializeHP() returned a less complex-model, it considered that model optimal and hence you'll have to make the new initialization yourself. This can be done by taking the output of initializeHP() and assigning new values, like this:

```
> newInitHP <- initHP
> newInitHP$G <- 3
> newInitHP$MUS <- c(-0.5, 0, 0.5)
> newInitHP$TAUS <- c(0.1, 0.2, 0.2)
> newInitHP$WEIGHTS <- c(0.25, 0.5, 0.25)
```

Since they're initial values, the specific numbers don't have to be perfect, just in line with what's eyeballed. Then run the one-step or full TCA-ECM (latter preferred in this situation) and the EM will adjust your guesses to match the data, just as it adjusted initializeHP()'s initializations before.

4 Meta-analysis

Meta-analysis in our framework assumes that that each study has its own study-specific parameters. Thus, those parameters should be estimated using a single-study DC function on each study, and then saving the resulting study-specific parameters. The meta-analysis itself is then run using

```
> ebCoexpressMeta(DList, conditionsList, pattern, hpEstsList)
```

It is perhaps easiest to explain the flow of execution through an example. Since it is not important that this example be realistic with respect to the data, and we want the example to run quickly, let us assume we have two studies that are each identical to the fiftyGenes data set, and so we can reuse the D matrix and the initialized values we computed earlier. The meta analysis would proceed thusly:

```
> D1 <- D
> D2 <- D
> DList <- list(D1, D2)
> cond1 <- tinyCond
> cond2 <- tinyCond
> conditionsList <- list(cond1, cond2)
> pattern <- ebPatterns(c("1,1","1,2"))
> initHP1 <- initHP
> initHP2 <- initHP
> out1 <- ebCoexpressZeroStep(D1, cond1, pattern, initHP1)

Zero-Stepper Time: 0.217

> out2 <- ebCoexpressZeroStep(D2, cond2, pattern, initHP2)

Zero-Stepper Time: 0.22

> hpEstsList <- list(out1$MODEL$HPS, out2$MODEL$HPS)
> metaResults <- ebCoexpressMeta(
+     DList, conditionsList, pattern, hpEstsList)

Running the [E M1] Cycle ...
Iteration: 1
Meta Analysis Time: 0.446999999999999
```

where the metaResults output is similar in format to that of the other, single-study analyses. Dawson and Kendziorski (2011) has a rather lengthy example of a meaningful meta-analysis involving three prostate cancer data sets; we will not deal with meta-analysis any further here.

5 Visualization

EBcoexpress contains two visualization functions that act as wrappers for plot() and plot.igraph(), allowing the user to use various output from EBcoexpress functions to look at co-expression at the gene-pair and network levels.

First, we'll use showNetwork() to build a co-expression network, with edges (pairs) colored according to the correlation between the gene (nodes) exhibited by the D matrix. showNetwork() is a wrapper for plot.igraph()

from the igraph package; as such it requires that package in order to work properly, along with the colorspace package. It takes as inputs an array of gene names (specifying the nodes), the D matrix (specifying the edges) and a condition to focus on. Additionally, the user may specify a layout recognized by igraph, such as random, circle or kamada.kawai (the default). Further options recognized by igraph, such as those used to specify node and edge characteristics like size or shape, may also be provided; see the documentation for more information.

It's more than a little cluttered to run showNetwork() on all fifty genes; so let's restrict our genes of interest to the first ten in each of the two sets:

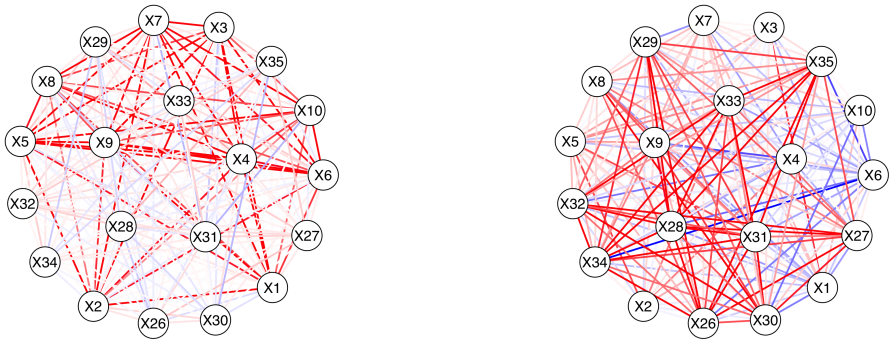
```
> twentyGeneNames <- dimnames(fiftyGenes)[[1]][c(1:10,26:35)]

> showNetwork(twentyGeneNames, D, condFocus = 1, gsep = "~",
+   layout = "kamada.kawai", seed = 5, vertex.shape="circle",
+   vertex.label.cex=1, vertex.color="white", edge.width=2,
+   vertex.frame.color="black", vertex.size=20,
+   vertex.label.color="black", vertex.label.family="sans")

> showNetwork(twentyGeneNames, D, condFocus = 2, gsep = "~",
+   layout = "kamada.kawai", seed = 5, vertex.shape="circle",
+   vertex.label.cex=1, vertex.color="white", edge.width=2,
+   vertex.frame.color="black", vertex.size=20,
+   vertex.label.color="black", vertex.label.family="sans")
```

The networks generated are shown in Figure 2. Nodes are genes and edges are gene pairs, with color indicating strength of correlation, ranging from blue (negative correlation) to white (uncorrelated) to red (positively correlated). Even with this smaller number of genes, things are still a little busy. We can remove edges corresponding to small magnitude correlations by using the hidingThreshold= option:

```
> showNetwork(twentyGeneNames, D, condFocus = 1, gsep = "~",
+   layout = "kamada.kawai", seed = 5, vertex.shape="circle",
+   vertex.label.cex=1, vertex.color="white", edge.width=2,
+   vertex.frame.color="black", vertex.size=20,
+   vertex.label.color="black", vertex.label.family="sans",
+   hidingThreshold=0.3)
```



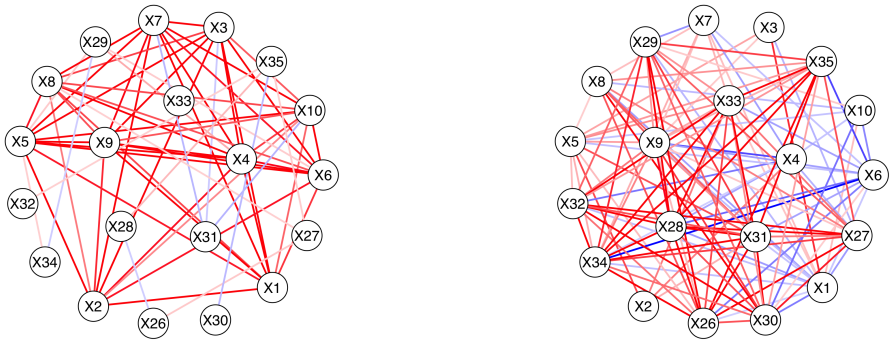
(a) (b)

Figure 2: A network of selected genes involved in DC pairs, created by showNetwork(); (a) and (b) correspond to conditions 1 and 2, respectively. Nodes are genes and edges are gene pairs, with color indicating strength of correlation, ranging from blue (negative correlation) to white (uncorrelated) to red (positively correlated). These networks include all pairs as thus are somewhat cluttered; a cleaned-up version may be found in Figure 3.

```
> showNetwork(twentyGeneNames, D, condFocus = 2, gsep = "~",
+   layout = "kamada.kawai", seed = 5, vertex.shape="circle",
+   vertex.label.cex=1, vertex.color="white", edge.width=2,
+   vertex.frame.color="black", vertex.size=20,
+   vertex.label.color="black", vertex.label.family="sans",
+   hidingThreshold=0.3)
```

We can look at the expression data for any given pair using showPair(), which plots the pair on a two-dimensional X-Y plane, colored by condition. By using the regLine= option (TRUE by default), the user can also have a line superimposed for each condition, indicating the trend; this line may be made 'robust' so that it is calculated using only those points used by biweight midcorrelation by setting useBWMC=TRUE. Other options specific to plot() may also be passed along. In Figure 4 we focus in on the co-expression between genes X1 and X2; see the demo for other examples.

```
> showPair("X1~X2", fiftyGenes, tinyCond, pch=20,
+   xlim=c(-4,4), ylim=c(-4,4))
```



(a) (b)

Figure 3: The network of selected genes involved in DC pairs from Figure 2, where (a) and (b) correspond to conditions 1 and 2, respectively. Nodes are genes and edges are gene pairs, with color indicating strength of correlation, ranging from blue (negative correlation) to white (uncorrelated) to red (positively correlated). Additionally, correlations less than 0.3 have been made transparent and hence removed from the network.

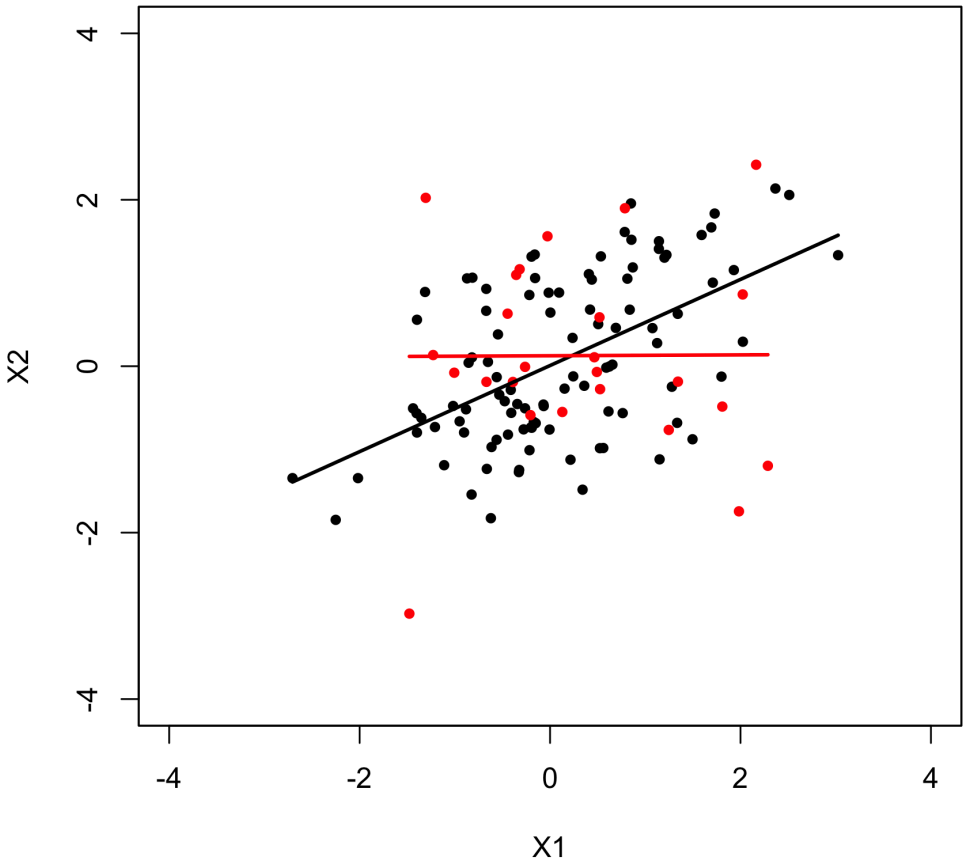


Figure 4: Expression data for a pair of genes, X1 and X2, from the fiftyGenes expression matrix. Colors represent condition (1 is black, 2 is red), and a robust line is added to indicate trend in each condition.

References

Dawson, J. A. and Kendziorski, C. (2011). An empirical Bayesian approach for identifying differential co-expression in high-throughput experiments. *Biometrics* E-publication before print: http://onlinelibrary.wiley.com/doi/10.1111/j.1541-0420.2011.01688.x/abstract.

Fisher, R. A. (1928). The general sampling distribution of the multiple correlation coefficient. *Journal of the Royal Statistical Society (Series A)* **121,** 654–673.

Fraley, C. and Raftery, A. E. (2002). Model-based clustering, discriminant analysis and density estimation. *Journal of the American Statistical Association* **97,** 611–631.

Fraley, C. and Raftery, A. E. (2006). MCLUST version 3 for R: Normal mixture modeling and model-based clustering. Technical Report 504, University of Washington, Department of Statistics. (revised 2009).