Manual for the R gaga package

David Rossell
Department of Bioinformatics & Biostatistics
IRB Barcelona
Barcelona, Spain.
rosselldavid@gmail.com

Here we illustrate several uses of the package gaga, including simulation, differential expression analysis, class prediction and sample size calculations. In Section ?? we review the GaGa and MiGaGa models. In Section ?? we simulate gene expression data, which we use to fit the GaGa model in Section ??. Diagnostics for model goodness-of-fit are presented in Section ??. Section ?? shows how to find differentially expressed genes, Section ?? how to obtain fold change estimates and Section ?? how to classify samples into groups. Finally, in Section ?? we perform fixed and sequential sample size calculations.

1 Introduction

? and ? introduced the Gamma-Gamma model to analyze microarray data, an elegant and parsimonious hierachical model that allows for the borrowing of information between genes. ? showed that the assumptions of this model are too simplistic, which resulted in a rather poor fit to several real datasets, and developed two extensions of the model: GaGa and MiGaGa. The gaga library implements the GaGa and MiGaGa generalizations, which can be used both to find differentially expressed genes and to predict the class of a future sample (*e.g.* given the mRNA measurements for a new patient, predict whether the patient has cancer or is healthy).

We now briefly outline the GaGa and MiGaGa models. Let x_{ij} be the expression measurement for gene i in array j, and let z_j indicate the group to which array belongs to (*e.g.* $z_j = 0$ for normal cells and $z_j = 1$ for cancer cells). The GaGa models envisions the observations as arising from a gamma distribution, *i.e.* $x_{ij} \sim \text{Ga}(\alpha_{i,z_j}, \alpha_{i,z_j}/\lambda_{i,z_j})$ (λ_{i,z_j} is the mean), where α_{i,z_j} and λ_{i,z_j} arise from a gamma and an inverse gamma distribution, respectively:

$$\begin{aligned}
\lambda_{i,k}|\delta_i, \alpha_0, \nu &\sim \text{IGa}(\alpha_0, \alpha_0/\nu), \text{ indep. for } i = 1 \dots n \\
\alpha_{i,k}|\delta_i, \beta, \mu &\sim \text{Ga}(\beta, \beta/\mu), \text{ indep. for } i = 1 \dots n \\
\delta_i|\boldsymbol{\pi} &\sim \text{Mult}(1, \boldsymbol{\pi}), \text{ indep. for } i = 1 \dots n. \qquad (1)
\end{aligned}$$

$\delta_1 \dots \delta_n$ are latent variables indicating what expression pattern each gene follows (see Section **??** for more details). For example, if there are only two groups δ_i indicates whether gene i is differentially expressed or not.

In principle, both the shape and mean parameters are allowed to vary between groups, and δ_i compares both parameters between groups (*i.e.* the GaGa model allows to compare not only mean expression levels but also the shape of the distribution between groups). However, the `gaga` library also implements a particular version of the model which assumes that the shape parameter is constant across groups, *i.e.* $\alpha_{i,k} = \alpha_i$ for all k.

The coefficient of variation in the Gamma distribution is equal to the inverse square root of the shape parameter, and hence assuming constant $\alpha_{i,k}$ is equivalent to assuming constant CV across groups.

In most routines the user can choose the constant CV model with the option `equalcv=TRUE` (the default), and the varying CV model with the option `equalcv=FALSE`.

The Bayesian model is completed by specifying priors on the hyper-parameters that govern the hierarchy:

$$\begin{aligned}
\alpha_0 \sim \text{Ga}(a_{\alpha_0}, b_{\alpha_0}); \nu &\sim \text{IGa}(a_\nu, b_\nu) \\
\beta \sim \text{Ga}(a_\beta, b_\beta); \mu &\sim \text{IGa}(a_\mu, b_\mu) \\
\boldsymbol{\pi} &\sim \text{Dirichlet}(\mathbf{p}). \qquad (2)
\end{aligned}$$

The `gaga` library provides some default values for the prior parameters that are a reasonable choice when the data has been normalized via the function `just.rma` from the R library `affy` or `just.gcrma` from the R library `just.gcrma`. The MiGaGa model extends GaGa by specifying a mixture of inverse gammas for ν.

Both models are fit using the routine `fitGG`: the argument `nclust` indicates the number of components in the mixture (`nclust=1` corresponds to the GaGa model).

2 Simulating the data

We start by loading the library and simulating mRNA expression levels for `n=100` genes and 2 groups, each with 6 samples. We set the seed for random

number generation so that you can reproduce the results presented here. We use the parameter estimates obtained from the Armstrong dataset (Armstrong, 2002) as described in (Rossell, 2009) As we shall see in the future sections, we use the first five samples from each group to fit the model. We will then use the model to predict the class for the sixth sample.

```
> library(gaga)
> set.seed(10)
> n <- 100; m <- c(6,6)
> a0 <- 25.5; nu <- 0.109
> balpha <- 1.183; nualpha <- 1683
> probpat <- c(.95,.05)
> xsim <- simGG(n,m=m,p.de=probpat[2],a0,nu,balpha,nualpha,equalcv=TRUE)
```

The object `xsim` is an `ExpressionSet`. The simulated expression values are accessible through `exprs(xsim)`, the parameters through `featureData(xsim)` and the group that each observation belongs through `pData(xsim)`. We save in `a` a matrix containing the gene-specific α parameters (`a[,1]` contains parameters for the first group, `a[,2]` for the second). Similarly, we save the gene-specific means λ in `l` and the expression values in `x`.

```
> xsim
```

```
ExpressionSet (storageMode: lockedEnvironment)
assayData: 100 features, 12 samples
  element names: exprs
protocolData: none
phenoData
  sampleNames: Array 1 Array 2 ... Array 12 (12 total)
  varLabels: group
  varMetadata: labelDescription
featureData
  featureNames: Gene 1 Gene 2 ... Gene 100 (100 total)
  fvarLabels: alpha.1 alpha.2 mean.1 mean.2
  fvarMetadata: labelDescription
experimentData: use 'experimentData(object)'
Annotation:
```

```
> featureData(xsim)
```

```
An object of class 'AnnotatedDataFrame'
  featureNames: Gene 1 Gene 2 ... Gene 100 (100 total)
  varLabels: alpha.1 alpha.2 mean.1 mean.2
  varMetadata: labelDescription
```

(a) (b)

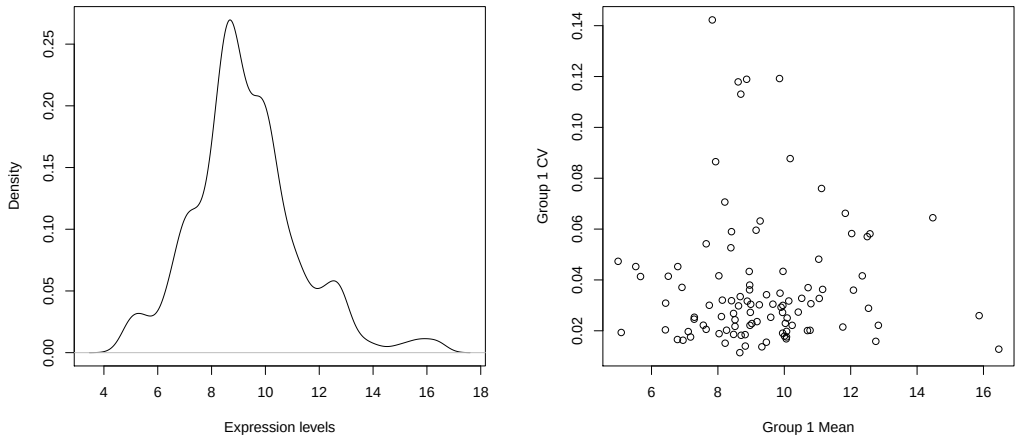

Figure 1: (a): marginal density of the simulated data; (b): plot of the simulated (α, λ) pairs

```
> phenoData(xsim)

An object of class 'AnnotatedDataFrame'
  sampleNames: Array 1 Array 2 ... Array 12 (12 total)
  varLabels: group
  varMetadata: labelDescription

> a <- fData(xsim)[,c("alpha.1","alpha.2")]
> l <- fData(xsim)[,c("mean.1","mean.2")]
> x <- exprs(xsim)
```

Figure **??**(a) shows the marginal distribution (kernel density estimate) of the simulated gene expression levels. Figure **??**(b) plots the simulated mean and coefficient of variation for group 1. The plots can be obtained with the following syntax:

```
> plot(density(x),xlab='Expression levels',main='')

> plot(l[,1],1/sqrt(a[,1]),xlab='Group 1 Mean',ylab='Group 1 CV')
```

3 Model fit

To fit the model we use the function `fitGG`. First, we define the vector `groups`, which indicates the group each sample belongs to. Second, we specify the gene expression patterns or hypotheses that we wish to entertain. In our example, since we have two groups there really are only two possible expression patterns:

Pattern 0 (null hypotheses): group 1 = group 2

Pattern 1 (alternative hypothesis): group 1 $\neq$ group 2.

More precisely, under pattern 0 we have that $\alpha_{i1} = \alpha_{i2}$ and $\lambda_{i1} = \lambda_{i2}$, while under pattern 1 $\alpha_{i1} \neq \alpha_{i2}$ and $\lambda_{i2} \neq \lambda_{i2}$. We specify the patterns with a matrix with as many rows as patterns and as many columns as groups. For each row of the matrix (*i.e.* each hypothesis), we indicate that two groups are equal by assigning the same number to their corresponding columns. The column names of the matrix must match the group codes indicated in `groups`, otherwise the routine returns an error. For example, in our two hypothesis case we would specify:

```
> groups <- pData(xsim)$group[c(-6,-12)]
> groups
```

```
 [1] group 1 group 1 group 1 group 1 group 1 group 2 group 2 group 2 group 2
[10] group 2
Levels: group 1 group 2
```

```
> patterns <- matrix(c(0,0,0,1),2,2)
> colnames(patterns) <- c('group 1','group 2')
> patterns
```

```
     group 1 group 2
[1,]       0       0
[2,]       0       1
```

For illustration, suppose that instead we had 3 groups and 4 hypotheses, as follows:

Pattern 0: CONTROL = CANCER A = CANCER B

Pattern 1: CONTROL $\neq$ CANCER A = CANCER B

Pattern 2: CONTROL = CANCER A $\neq$ CANCER B

Pattern 3: CONTROL $\neq$ CANCER A $\neq$ CANCER B

In this case we would specify

```
> patterns <- matrix(c(0,0,0,0,1,1,0,0,1,0,1,2),ncol=3,byrow=TRUE)
> colnames(patterns) <- c('CONTROL','CANCER A','CANCER B')
> patterns
```

```
     CONTROL CANCER A CANCER B
[1,]       0        0        0
[2,]       0        1        1
[3,]       0        0        1
[4,]       0        1        2
```

That is, the second row indicates that under Pattern 1 cancers of type A and B are present the same expression levels, since they both have a `1` in their entries. The last row indicates that they are all different by specifying a different number in each entry.

Now, to fit the GaGa model to our simulated data we use `fitGG`, with `nclust=1` (to fit the MiGaGa model we would set `nclust` to the number of components that we want in the mixture). We remove columns 6 and 12 from the dataset, *i.e.* we do not use them for the fit so that we can evaluate the out-of-sample behavior of the classifier built in Section **??**. Here we use the option `trace=FALSE` to prevent iteration information from being printed. There are several available methods to fit the model. `method=='EM'` implements an Expectation-Maximization algorithm which seeks to maximize the expected likelihood. `method=='quickEM'` (the default) is a quicker version that uses only 1 maximization step. `quickEM` usually provides reasonably good hyper-parameter estimates at a low computational cost. In practice we have observed that the inference derived from the GaGa and MiGaGa models (*e.g.* lists of differentially expressed genes) is robust to slight hyper-parameter miss-specifications, so we believe `quickEM` to be a good default option for large datasets. `method=='SA'` implements a Simulated Annealing scheme which searches for a hyper-parameter value with high posterior density.

The three above-mentioned methods (`EM`, `quickEM`, `SA`) only provide point estimates. We can obtain credibility intervals with `method=='Gibbs'` or `method=='MH'`, which fit a fully Bayesian model via Gibbs or Metropolis-Hastings MCMC posterior sampling, respectively. Of course, obtaining credibility intervals comes at a higher computational cost. In our experience the five methods typically deliver similar results.

```
> patterns <- matrix(c(0,0,0,1),2,2)
> colnames(patterns) <- c('group 1','group 2')
> gg1 <- fitGG(x[,c(-6,-12)],groups,patterns=patterns,nclust=1,method='Gibbs',B
> gg2 <- fitGG(x[,c(-6,-12)],groups,patterns=patterns,method='EM',trace=FALSE)
> gg3 <- fitGG(x[,c(-6,-12)],groups,patterns=patterns,method='quickEM',trace=FA
```

We can obtain iteration plots to visually assess the convergence of the chain. The component `mcmc` of `gg1` contains an object of type `mcmc`, as defined in the library `coda`.

To obtain parameter estimates and the posterior probability that each gene is differentially expressed we use the function `parest`. We discard the first 100 MCMC iterations with `burnin=100`, and we ask for 95% posterior credibility intervals with `alpha=.05`. The slot `ci` of the returned object contains the credibility intervals (this option is only available for `method=='Gibbs'` and `method=='MH'`).

```
> gg1 <- parest(gg1,x=x[,c(-6,-12)],groups,burnin=100,alpha=.05)
> gg2 <- parest(gg2,x=x[,c(-6,-12)],groups,alpha=.05)
> gg3 <- parest(gg3,x=x[,c(-6,-12)],groups,alpha=.05)
> gg1

GaGa hierarchical model. Fit via Gibbs sampling (900 iterations kept)
Assumed constant CV across groups
  100 genes, 2 groups, 2 hypotheses (expression patterns)

The expression patterns are
  Pattern 0 (93.6% genes): group 1 = group 2
  Pattern 1 (6.4% genes): group 1 !=group 2

Hyper-parameter estimates

   a0 nu balpha nualpha
   21.699 0.113 1.326 1399.749

   probclus
   1

> gg1$ci

$a0
    2.5%    97.5%
16.40743 28.17880
```

```
$nu
     2.5%     97.5%
0.1086822 0.1174830

$balpha
     2.5%     97.5%
0.9729244 1.7644987

$nualpha
    2.5%    97.5%
1128.558 1711.677

$probclus
[1] 1 1

$probpat
      probpat.1  probpat.2
2.5%  0.8689520 0.01912461
97.5% 0.9808754 0.13104796
```

> gg2

```
GaGa hierarchical model. Fit via Expectation-Maximization
Assumed constant CV across groups
  100 genes, 2 groups, 2 hypotheses (expression patterns)

The expression patterns are
  Pattern 0 (93.8% genes): group 1 = group 2
  Pattern 1 (6.2% genes): group 1 !=group 2

Hyper-parameter estimates

   alpha0 nu balpha nualpha
   21.659 0.113 1.332 1385.684

   probclus
   1
```

> gg3

```
GaGa hierarchical model.Fit via quick Expectation-Maximization
Assumed constant CV across groups
```

```
  100 genes, 2 groups, 2 hypotheses (expression patterns)

The expression patterns are
  Pattern 0 (93.8% genes): group 1 = group 2
  Pattern 1 (6.2% genes): group 1 !=group 2

Hyper-parameter estimates

   alpha0 nu balpha nualpha
   21.659 0.113 1.268 1394.422

   probclus
   1
```

Although the parameter estimates obtained from the four methods are similar to each other, some differences remain. This is due to some extent to our having a small dataset with only 100 genes. For the larger datasets encountered in practice the four methods typically deliver very similar results. In Section **??** we assess whether the lists of differentially expressed genes obtained with each method are actually the same. The slot `pp` in `gg1` and `gg2` contains a matrix with the posterior probability of each expression pattern for each gene. For example, to find probability that the first gene follows pattern 0 (*i.e.* is equally expressed) and pattern 1 (*i.e.* is differentially expressed) we do as follows.

```
> dim(gg1$pp)

[1] 100   2

> gg1$pp[1,]

[1] 0.994767031 0.005232969

> gg2$pp[1,]

[1] 0.994906189 0.005093811
```

4 Checking the goodness of fit

To graphically assess the goodness of fit of the model, one can used prior-predictive or posterior-predictive checks. The latter, implemented in the

function `checkfit`, are based on drawing parameter values from the posterior distribution for each gene, and possibly using then to generate data values, and then compare the simulated values to the observed data. The data generated from the posterior predictive is compared to the observed data in Figure **??**(a). Figure **??**(b)-(d) compares draws from the posterior of α and λ with their method of moments estimate, which is model-free. All plots indicate that the model has a reasonably good fit. The figures were generated with the following code:

```
> checkfit(gg1,x=x[,c(-6,-12)],groups,type='data',main='')

> checkfit(gg1,x=x[,c(-6,-12)],groups,type='shape',main='')

> checkfit(gg1,x=x[,c(-6,-12)],groups,type='mean',main='')

> checkfit(gg1,x=x[,c(-6,-12)],groups,type='shapemean',main='',xlab='Mean',ylab
```

It should be noted, however, that posterior-predictive plots can fail to detect departures from the model, since there is a double use of the data. Prior-predictive checks can be easily implemented using the function `simGG` and setting the hyper-parameters to their posterior mean.

5 Finding differentially expressed genes

The function `findgenes` finds differentially expressed genes, *i.e.* assigns each gene to an expression pattern. The problem is formalized as minizing the false negative rate, subject to an upper bound on the false discovery rate, say `fdrmax=0.05`. In a Bayesian sense, this is achieved by assigning to pattern 0 (null hypothesis) all genes for which the posterior probability of following pattern 0 is above a certain threshold (Mueller, 2004). The problem is then to find the optimal threshold, which can be done parametrically or non-parametrically through the use of permutations (for details see Rossell, 2009).Here we explore both options, specifying `B=1000` permutations for the non-parametric option.

```
> d1 <- findgenes(gg1,x[,c(-6,-12)],groups,fdrmax=.05,parametric=TRUE)
> d1.nonpar <- findgenes(gg1,x[,c(-6,-12)],groups,fdrmax=.05,parametric=FALSE,B

Finding clusters of z-scores for bootstrap... Done
Starting 1000 bootstrap iterations...
```

(a) (b)

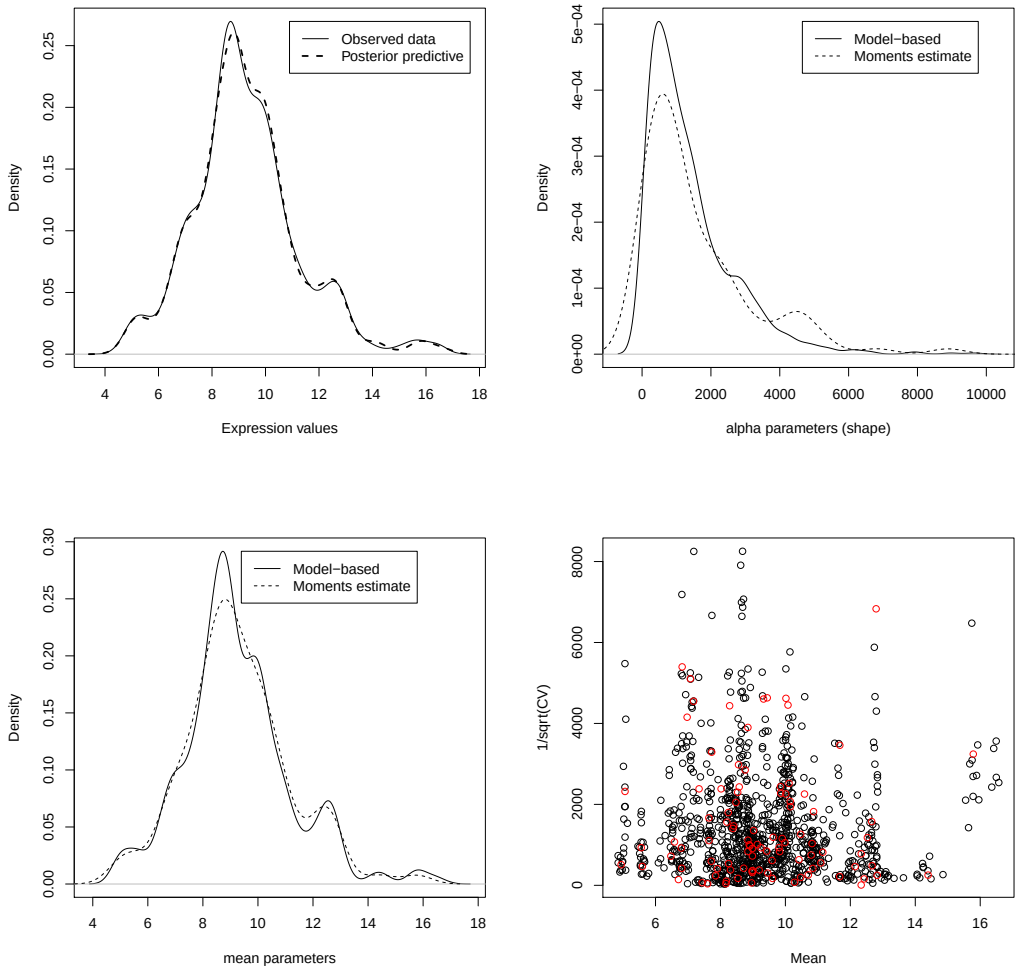

Figure 2: Assessing the goodness of fit. (a): compares samples from the posterior predictive to the observed data; (b): compares samples from the posterior of α to the method of moments estimate; (c): compares samples from the posterior of λ to the method of moments estimate; (d): as (b) and (c) but plots the pairs (α, λ) instead of the kernel density estimates

```
> dtrue <- (l[,1]!=l[,2])
> table(d1$d,dtrue)
```

```
   dtrue
    FALSE TRUE
  0    95    1
  1     0    4
```

```
> table(d1.nonpar$d,dtrue)
```

```
   dtrue
    FALSE TRUE
  0    95    1
  1     0    4
```

We set the variable `dtrue` to indicate which genes were actually differentially expressed (easily achieved by comparing the columns of `xsim$l`). Both the parametric and non-parametric versions declare 4 genes to be DE, all of them true positives. They both fail to find one of the DE genes. To obtain an estimated frequentist FDR for each Bayesian FDR one can plot `d1.nonpar$fdrest`. The result, shown in Figure **??**, reveals that setting the Bayesian FDR at a 0.05 level results in an estimated frequentist FDR around 0.015. That is, calling `findgenes` with the option `parametric=TRUE` results in a slightly conservative procedure from a frequentist point of view.

```
> plot(d1.nonpar$fdrest,type='l',xlab='Bayesian FDR',ylab='Estimated frequentis
```

Finally, we compare the list of differentially expressed genes with those obtained when using the other fitting criteria explained in Section **??**.

```
> d2 <- findgenes(gg2,x[,c(-6,-12)],groups,fdrmax=.05,parametric=TRUE)
> d3 <- findgenes(gg3,x[,c(-6,-12)],groups,fdrmax=.05,parametric=TRUE)
> table(d1$d,d2$d)
```

```
     0  1
  0 96  0
  1  0  4
```

```
> table(d1$d,d3$d)
```

```
     0  1
  0 96  0
  1  0  4
```

Despite the existence of small differences in the hyper-parameter estimates between methods, the final list of differentially expressed genes is the same for all of them. This suggests that the GaGa model is somewhat robust to the hyper-parameter specification.

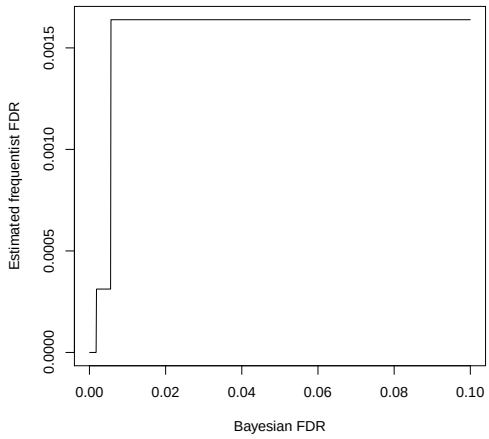

Figure 3: Estimated frequenstist FDR vs. Bayesian FDR

6 Obtaining fold change estimates

The GaGa and MiGaGa models can be used to obtain fold change estimates, by computing the posterior expected expression values for each group. As these posterior expectations are derived from a hierarchical model, they are obtained by borrowing information across genes. Therefore, in small sample situations they are preferrable to simply using the group sample means.

The function `posmeansGG` computes posterior expected values under any expression pattern. The expression pattern is indicated with the argument `underpattern`. In our example (as in most microarray experiments) pattern 0 corresponds to the null hypothesis that no genes are differentially expressed. Therefore, specifying `underpattern=0` would result in obtaining identical expression estimates for both groups. Instead, one is commonly interested in computing a different mean for each group, which in our case corresponds to pattern 1. As the expression measurements were simulated to be in log2 scale, the log-fold change can be computed by taking the difference between the two group means (if the data were in the original scale, we would divide instead). The code below computed posterior means and log-fold changes, and prints out the fold change for the first five genes.

```
> mpos <- posmeansGG(gg1,x=x[,c(-6,-12)],groups=groups,underpattern=1)

 Computing posterior means under expression pattern 1 ...
```

```
> fc <- mpos[,1]-mpos[,2]
> fc[1:5]

[1] -0.11041018 -0.01292394 -0.83741320  0.08419900 -0.05508204
```

7 Class prediction

We now use the fitted model to predict the class of the arrays number 6 and 12, neither of which were used to fit the model. We assume that the prior probability is 0.5 for each group, though in most settings this will not be realistical. For example, if `groups==2` indicates individuals with cancer, one would expect the prior probability to be well below 0.5, say around 0.1. But if the individual had a positive result in some test that was administered previously, this probability would have increased, say to 0.4.

Class prediction is implemented in the function `classpred`. The argument `xnew` contains the gene expression measurements for the new individuals, `x` is the data used to fit the model and `ngene` indicates the number of genes that should be used to build the classifier. It turns out that array 6 is correctly assigned to group 1 and array 12 is correctly assigned to group 2. `classpred` also returns the posterior probability that the sample belongs to each group. We see that for the dataset at hand the posterior probability of belonging to the wrong group is essentially zero. Similarly good results are obtained when using setting `ngene` to either 1 (the minimum value) or to 100 (the maximum value). The fact that adding more gene to the classifier does not change its performance is not surprising, since the classifier assigns little weight to genes with small probability of being DE. We have observed a similar behavior in many datasets. The fact that the classifier works so well with a single is typically not observed in real datasets, where it is rare to have a gene with such a high discrimination power.

```
> pred1 <- classpred(gg1,xnew=x[,6],x=x[,c(-6,-12)],groups,ngene=50,prgroups=c(
> pred2 <- classpred(gg1,xnew=x[,12],x=x[,c(-6,-12)],groups,ngene=50,prgroups=c
> pred1

$d
[1] 1

$posgroups
[1] 1.000000e+00 2.324522e-24

> pred2
```

```
$d
[1] 2

$posgroups
[1] 9.436972e-22 1.000000e+00
```

8 Designing high-throughput experiments

The gaga package incorporates routines which can be used for fixed and sequential sample size calculation in high-throughput experiments. For details on the methodology see **?**. We start by simulating some data from a GaGa model. Since the computations can be intensive, here we simulate data for 20 genes only. The question is, given the observed data, how many more samples should we collect, if any?

```
> set.seed(1)
> x <- simGG(n=20,m=2,p.de=.5,a0=3,nu=.5,balpha=.5,nualpha=25)
> gg1 <- fitGG(x,groups=1:2,method='EM')

Initializing parameters... Done.
Refining initial estimates...
Starting EM algorithm...
alpha0[1]     nu[1]    balpha    nualpha probpat[1] probpat[2] logl
 3.892212  0.433702  0.189467 105.964470   0.650039   0.349961 -21.289791
 3.532750  0.456514  0.459891 21.641185   0.695806   0.304194 -17.570110
 3.596507  0.451893  0.459891 21.462008   0.734122   0.265878 -17.393875
 3.657439  0.447951  0.459891 21.260235   0.766362   0.233638 -17.260041
 3.706742  0.444576  0.459891 21.072151   0.793631   0.206369 -17.156884
 3.746888  0.441692  0.459891 20.902542   0.816838   0.183162 -17.076171
 3.779824  0.439215  0.459891 20.751736   0.836716   0.163284 -17.012084
 3.807127  0.437077  0.459891 20.618402   0.853855   0.146145 -16.960480
 3.807127  0.437077  0.459891 20.618402   0.868318   0.131682 -16.919115
 3.807127  0.437077  0.459891 20.618402   0.880643   0.119357 -16.886760
 3.807127  0.437077  0.459891 20.618402   0.891240   0.108760 -16.861047
 3.807127  0.437077  0.459891 20.618402   0.900427   0.099573 -16.840319
 3.807127  0.437077  0.459891 20.618402   0.908451   0.091549 -16.823389
 3.807127  0.437077  0.459891 20.618402   0.915508   0.084492 -16.809398
 3.807127  0.437077  0.459891 20.618402   0.921754   0.078246 -16.797712
 3.807127  0.437077  0.459891 20.618402   0.927312   0.072688 -16.787856
 3.807127  0.437077  0.459891 20.618402   0.932286   0.067714 -16.779470
```

```
> gg1 <- parest(gg1,x=x,groups=1:2)
```

The function `powfindgenes` evaluates the (posterior) expected number of new true gene discoveries if one were to obtain an additional batch of data with `batchSize` new samples per group. For our simulated data, we expect that obtaining 1 more sample per group would provide no new gene discoveries. For 2 and 3 more samples per group we still expect to discover less than one new gene (which seems reasonable for our simulated data with only 20 genes).

```
> pow1 <- powfindgenes(gg1, x=x, groups=1:2, batchSize=1, fdrmax=0.05, B=1000)
> pow2 <- powfindgenes(gg1, x=x, groups=1:2, batchSize=2, fdrmax=0.05, B=1000)
> pow3 <- powfindgenes(gg1, x=x, groups=1:2, batchSize=3, fdrmax=0.05, B=1000)
> pow1$m

[1] 0

> pow2$m

[1] 0.02612138

> pow3$m

[1] 0.1148547
```

As illustrated, calling `powfindgenes` for different values of `batchSize` can be used to determine the sample size. We refer to this approach as fixed sample size calculation, since the number of additional samples is fixed from now on, regardless of the evidence provided by new data. Function `forwsimDiffExpr` provides a sequential sample size alternative. The idea is that, every time that we observe new data, we can use `powfindgenes` to estimate the expected number of new gene discoveries for an additional data batch. As long as this quantity is large enough, we keep adding new samples. When this quantity drops below some threshold we stop experimentation. `forwsimDiffExpr` uses forward simulation to determine reasonable values for this threshold. Shortly, the function simulates `batchSize` new samples per group from the GaGa posterior predictive distribution, and for each of them evaluates the expected number of new discoveries via `powfindgenes` (estimated via `Bsummary` Monte Carlo samples). Then `batchSize` more samples are added, and `powfindgenes` is called again, up to a maximum number of batches `maxBatch`. The whole process is repeated `B` times. Notice that, although not illustrated here, parallel processing can be used to speed up computations (*e.g.* see `mcapply` from package `parallel`).

```
> fs1 <- forwsimDiffExpr(gg1, x=x, groups=1:2,
+ maxBatch=3,batchSize=1,fdrmax=0.05, B=100, Bsummary=100, randomSeed=1)

Starting forward simulation...
  0 iterations
  10 iterations
  20 iterations
  30 iterations
  40 iterations
  50 iterations
  60 iterations
  70 iterations
  80 iterations
  90 iterations

> head(fs1)

    simid time u fdr        fnr power summary
1       0    0 0   0 0.06324234     0       0
101     0    1 0   0 0.05726000     0       0
102     0    2 0   0 0.05671000     0       0
103     0    3 0   0 0.05430000     0      NA
2       1    0 0   0 0.06324234     0       0
104     1    1 0   0 0.07213000     0       0
```

`forwsimDiffExpr` returns a `data.frame` indicating, for each simulation and stopping time (number of batches), the posterior expectation of the number of true posities (`u`), false discovery and false negative rates (`fdr`, `fnr`), and power (*e.g.* number of detected DE genes at a Bayesian FDR `fdrmax` divided by overall number of DE genes). It also returns the (posterior predictive) expected new DE discoveries for one more data batch in `summary`. Since experimentation is always stopped at `time==maxBatch`, it is not necessary to evaluate `summary` at this time point and `NA` is returned.

The output of `forwsimDiffExpr` can be used to estimate the expected number of new true discoveries for each sample size, as well as to estimate the expected utility by subtracting a sampling cost. As illustrated above these results can also be obtained with `powfindgenes`, which is much faster computationally.

```
> tp <- tapply(fs1$u,fs1$time,'mean')
> tp
```

```
        0         1         2         3
0.0000000 0.0000000 0.0485364 0.1376917
```

```
> samplingCost <- 0.01
> util <- tp - samplingCost*(0:3)
> util
```

```
         0          1          2          3
 0.0000000 -0.0100000  0.0285364  0.1076917
```

Again, for our simulated data we expect to find very few DE genes with 1, 2 or 3 additional data batches. For a sampling cost of 0.01, the optimal fixed sample design is to obtain 3 more data batches. Here we set a very small sampling cost for illustration purposes, although in most applications both the number of genes and the sampling cost will be much larger. For instance, `samplingCost=50` would indicate that the experimenter considers it worthwhile to obtain one more batch of samples as long as that allows him to find at least 50 new DE genes.

In order to find the optimal sequential design, we define a grid of intercept and slope values for the linear stopping boundaries. The function `seqBoundariesGrid` returns the expected utility for each intercept and slope in the grid in the element `grid`. The element `opt` contains the optimum and the expected utility, FDR, FNR, power and stopping time (*i.e.* the expected number of data batches).

```
> b0seq <- seq(0,20,length=200); b1seq <- seq(0,40,length=200)
> bopt <-seqBoundariesGrid(b0=b0seq,b1=b1seq,forwsim=fs1,samplingCost=samplingC
> names(bopt)
```

```
[1] "opt"  "grid"
```

```
> bopt$opt
```

```
       b0        b1         u       fdr       fnr     power      time
0.0000000 0.0000000 0.1154298 0.0023041 0.0568169 0.0608967 2.1800000
```

```
> head(bopt$grid)
```

```
  b0        b1          u       fdr       fnr     power time
1  0 0.0000000  0.1154298 0.0023041 0.0568169 0.0608967 2.18
2  0 0.2010050  0.0573792 0.0009914 0.0549463 0.0312966 1.14
3  0 0.4020101 -0.0100000 0.0000000 0.0577324 0.0000000 1.00
4  0 0.6030151 -0.0100000 0.0000000 0.0577324 0.0000000 1.00
5  0 0.8040201 -0.0100000 0.0000000 0.0577324 0.0000000 1.00
6  0 1.0050251 -0.0100000 0.0000000 0.0577324 0.0000000 1.00
```

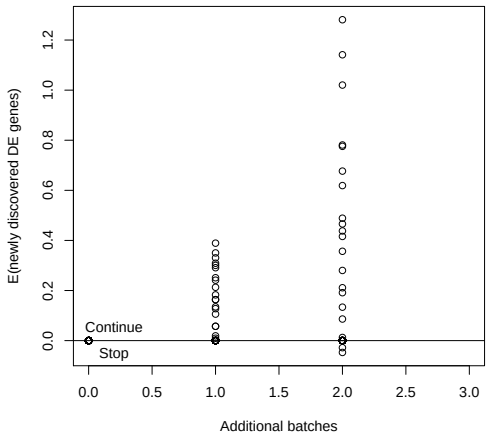

Figure 4: Forward simulations and optimal sequential rule

The expected utility for the optimal boundary is slightly larger than for a fixed sample size of 3 batches per group (see above), and perhaps more importantly it is achieved with a smaller average sample size. The optimal intercept and slope are equal to 0. Recall that experimentation at time $t = 1, \ldots,$`batchSize`-2 continues as long as `summary` is greater or equal than the stopping boundary. Therefore the optimal rule implies never stopping at $t = 1$. At time `batchSize`-1 ($t = 2$ in our example) the optimal decision is to continue whenever the one-step ahead expected new true discoveries (`summary`) is greater than `samplingCost`, regardless of the stopping boundary.

We produce a plot to visualize the results (Figure **??**). The plot shows the simulated trajectories for the summary statistic, and the optimal stopping boundary.

```
> plot(fs1$time,fs1$summary,xlab='Additional batches',ylab='E(newly discovered
> abline(bopt$opt['b0'],bopt$opt['b1'])
> text(.2,bopt$opt['b0'],'Continue',pos=3)
> text(.2,bopt$opt['b0'],'Stop',pos=1)
```