Bioconductor 2015 - FAQ Live!

James W. MacDonald*

July 17, 2015

Contents

*jmacdon@u.washington.edu

1 Overview

This workshop is intended to be a live version of the Bioconductor support site, where we can go over typical questions, but perhaps in more depth. Given the short time available, we will cover two of the most frequently asked question types; design and contrast matrix construction and interpretation, and error tracking/debugging.

2 Asking questions on the support site

Before getting to the main part of the course, we should first discuss how best to ask questions on the support site. We like to think of the support site as a repository of knowledge, where people with a question can look first, in order to answer their questions quickly and accurately. This works best if the questions posed on the support site are of high quality to begin with, which is where our end users come in.

If you are trying to do something using Bioconductor, and get stuck, the first place you should go is Google! A well crafted Google search will answer your question most of the time, especially since the support site is idexed by Google. But let's say you have tried Google, but haven't come up with a useful answer. You could search the support site directly, but I find that Google does a better job than the search function on the support site. So really, you need to ask a question. The best way to ask the question is to try to reverse your position, and ask yourself what you would need to know in order to answer the question!

This is actually easier to do than you would think. The first thing you need to do is be exact, but concise, about what you are doing, what you expect, and what tools you are using.

Here is a really bad question

I am using lmFit for my analysis, and it doesn't work. Can anybody help?

And here is a moderately improved question:

I am using the limma package to compare tumor versus control, and I get an error I don't understand. I have one tumor and one control sample, and when I run eBayes, it gives me an error. Here is the code I ran:

```
fit <- lmFit(eset, design)
fit2 <- eBayes(fit)
Error in ebayes(fit = fit, proportion = proportion, stdev.coef.lim = stdev.coef.lim,  :
  No residual degrees of freedom in linear model fits
```

Can anybody help me?

The first question obscures the real problem, and will require a few back-and-forth exchanges between the original poster and whomever decides to help. The second question is a bit better, because an experienced helper will know basically what the original poster is doing, what that error means, and will be able to explain it.

Ideally, the original poster will give the following bits of information:

- This is what I am trying to do
- This is the package (packages) I am using
- These are the data I am using (maybe R output using head(), or str())
- This is what I expect
- These are the results I am getting
- Here is the output from sessionInfo()

An improved way to ask the above question would be this:

I am trying to compare two samples (tumor and control) using limma, and I am getting an error I don't understand. My design matrix looks like this:

```
design
```

```
##   (Intercept) TreatTreated
## 1           1            1
## 2           1            0
## attr(,"assign")
## [1] 0 1
## attr(,"contrasts")
## attr(,"contrasts")$Treat
## [1] "contr.treatment"
```

My data look like this:

```
head(eset)
```

```
##             [,1]       [,2]
## [1,]  1.2862813  0.6391255
## [2,]  2.3013711  1.5288431
## [3,]  1.3769291 -0.3746192
## [4,] -0.6887273  1.7687910
## [5,] -0.1856149 -1.0475340
## [6,] -0.7912769 -1.0959626
```

and the error I get is:

```
fit <- lmFit(eset, design)
fit2 <- eBayes(fit)
Error in ebayes(fit = fit, proportion = proportion, stdev.coef.lim = stdev.coef.lim,  :
  No residual degrees of freedom in linear model fits
```

my sessionInfo() is:

[snip]

If you think you have found a bug, then you would give slightly different information. Ideally you would be able to give a small example that reproduces the error you are getting, so the package maintainer can track the bug down.

There is a trade off between saying too much and saying too little. Try to be as explicit as possible without adding in extraneous details, although if you are really stuck it may be hard to know what is important

and what is not. In that case, err on the side of too much information.

3 Design matrices

3.1 Two-sample comparison

To explain how to generate and interpret design matrices, we will go over a set of examples that get progressively more complicated. The first one is simplest of all. Consider an experiment where you have run 10 samples using RNA-Seq, five each of tumor and control.

```
d.f <- data.frame(Samples = paste("Sample", 1:10),
                  Type = factor(rep(c("Tumor","Control"),
                  each = 5)))
d.f
```

```
##      Samples    Type
## 1   Sample 1   Tumor
## 2   Sample 2   Tumor
## 3   Sample 3   Tumor
## 4   Sample 4   Tumor
## 5   Sample 5   Tumor
## 6   Sample 6 Control
## 7   Sample 7 Control
## 8   Sample 8 Control
## 9   Sample 9 Control
## 10 Sample 10 Control
```

We can define the design matrices using the `model.matrix` function, either with or without an intercept. Please note that we will get identical results regardless - it just changes the interpretation of the model coefficients. First we can make a design matrix without an intercept.

```
design1 <- model.matrix(~0 + Type, d.f)
design1
```

```
##    TypeControl TypeTumor
## 1            0         1
## 2            0         1
## 3            0         1
## 4            0         1
## 5            0         1
## 6            1         0
## 7            1         0
## 8            1         0
## 9            1         0
## 10           1         0
## attr(,"assign")
## [1] 1 1
## attr(,"contrasts")
```

```
## attr(,"contrasts")$Type
## [1] "contr.treatment"
```

And we can make a design matrix with an intercept.

```
design2 <- model.matrix(~ Type, d.f)
design2
```

```
##    (Intercept) TypeTumor
## 1            1         1
## 2            1         1
## 3            1         1
## 4            1         1
## 5            1         1
## 6            1         0
## 7            1         0
## 8            1         0
## 9            1         0
## 10           1         0
## attr(,"assign")
## [1] 0 1
## attr(,"contrasts")
## attr(,"contrasts")$Type
## [1] "contr.treatment"
```

The only difference between these two design matrices is that the first one (without intercept) is computing the mean of each sample type, and the second design matrix has a baseline (intercept) term that is estimating the mean of the control samples, and a 'TypeTumor' term that is estimating the difference between Tumor and Control samples. How do we know this?

```
data.frame(Type = d.f$Type, design1)
```

```
##       Type TypeControl TypeTumor
## 1    Tumor           0         1
## 2    Tumor           0         1
## 3    Tumor           0         1
## 4    Tumor           0         1
## 5    Tumor           0         1
## 6  Control           1         0
## 7  Control           1         0
## 8  Control           1         0
## 9  Control           1         0
## 10 Control           1         0
```

The rows of a design matrix represent the samples, and the columns represent the model coefficients. The design matrix itself is an indicator matrix that shows if a given sample contributes information to a given coefficient (1 == yes, 0 == no). This leads to two conclusions. First, if a row is all zeros, except for one coefficient, then the coefficient is estimating the mean of the group that the sample belongs to. Second, we can figure out what that group is by looking to see which samples all have a single one in that column. So using that logic, we can say that TypeControl is estimating the mean of the Controls and TypeTumor is the

mean of the Tumor samples.

```
data.frame(Type = d.f$Type, design2)
```

```
##       Type X.Intercept. TypeTumor
## 1    Tumor            1         1
## 2    Tumor            1         1
## 3    Tumor            1         1
## 4    Tumor            1         1
## 5    Tumor            1         1
## 6  Control            1         0
## 7  Control            1         0
## 8  Control            1         0
## 9  Control            1         0
## 10 Control            1         0
```

The other design is more confusing, because the intercept is all 1s. But using the logic from above, we see that the rows with a single 1 are just the Control samples, so we infer that the Intercept is estimating the mean of the Controls. But what about the TypeTumor column?

We already know that TypeControl is the mean of the Controls, so this is easy to figure out using algebra:

$$Tumor = Control + TypeTumor$$
$$Tumor - Control = TypeTumor$$

To show that we get identical results, regardless of the design matrix, let's try this using limma

```
library(limma)
set.seed(0xabeef)
y <- matrix(rnorm(10000), 1000)
## note that for design we need to use a contrasts matrix
contrast <- makeContrasts(TypeTumor - TypeControl, levels = design1)
fit <- lmFit(y, design1)
fit2 <- contrasts.fit(fit, contrast)
fit2 <- eBayes(fit2)

fit3 <- lmFit(y, design2)
fit3 <- eBayes(fit3)

 all.equal(topTable(fit2,1), topTable(fit3,2))
```

```
## [1] TRUE
```

3.2 Two-sample paired comparison

Let's assume that instead of having five tumor and five control subjects, we had five paired tumor/adjacent normal tissue samples and we wanted to know what genes are differentially expressed between the two tissue types. This is an inherently more powerful analysis, because we can control for the patient effect directly. In other words, one patient might have a higher expression of a given gene than the others in the study. This isn't of interest to us - instead we want to know how tumors differ from normal tissue - and the higher expression of a gene in a particular patient just introduces variability that makes it harder to detect differences. By using paired samples, we can estimate this patient-specific variability, and then ignore it when we make the comparisons we care about.

```
d.f2 <- data.frame(Patient = paste("Patient", rep(1:5, 2)),
                   Type = d.f$Type)

design3 <- model.matrix(~Type + Patient, d.f2)
colnames(design3) <- gsub("Type|^Patient", "", colnames(design3))
design3
```

```
##    (Intercept) Tumor Patient 2 Patient 3 Patient 4 Patient 5
## 1            1     1         0         0         0         0
## 2            1     1         1         0         0         0
## 3            1     1         0         1         0         0
## 4            1     1         0         0         1         0
## 5            1     1         0         0         0         1
## 6            1     0         0         0         0         0
## 7            1     0         1         0         0         0
## 8            1     0         0         1         0         0
## 9            1     0         0         0         1         0
## 10           1     0         0         0         0         1
## attr(,"assign")
## [1] 0 1 2 2 2 2
## attr(,"contrasts")
## attr(,"contrasts")$Type
## [1] "contr.treatment"
##
## attr(,"contrasts")$Patient
## [1] "contr.treatment"
```

Using what we learned in the last section, what does the Intercept estimate? How about Tumor? What about Patient 2 - Patient 5?

For a conventional paired analysis, we compute Tumor - Control first, and then test to see if the mean of those differences are equal to zero or not. If you compute the differences by hand (using the 'y' matrix we generated above), what would the design matrix look like? Do you get the same results that you get using design3?

3.3 Batch effects

Let's say your lab ran an experiment where you had eight mice, four of which were treated with a new drug, and four with a vehicle control. Later on, your PI decided it wouldn't be enough samples, and had you run three more mice per group. You analyze using microarrays, and as part of the QC do a PCA plot that indicates there is a batch effect, where the first group looks substantially different from the second group.

In this situation, we know there is some technical variability between the two batches, and like the patient-specific effect in the last section, it isn't interesting - it's just a nuisance that we have to deal with. We deal with them by estimating the difference between say batch A and batch B (A - B), and then using that difference to adjust the samples from batch A so they are comparable to batch B.

```
d.f3 <- data.frame(Treatment = rep(c("Treat","Cont","Treat","Cont"), c(4,4,3,3)),
                   Batch = rep(c("A","B"), c(8,6)))
design4 <- model.matrix(~Treatment + Batch, d.f3)
design4
```

```
##    (Intercept) TreatmentTreat BatchB
## 1            1              1      0
## 2            1              1      0
## 3            1              1      0
## 4            1              1      0
## 5            1              0      0
## 6            1              0      0
## 7            1              0      0
## 8            1              0      0
## 9            1              1      1
## 10           1              1      1
## 11           1              1      1
## 12           1              0      1
## 13           1              0      1
## 14           1              0      1
## attr(,"assign")
## [1] 0 1 2
## attr(,"contrasts")
## attr(,"contrasts")$Treatment
## [1] "contr.treatment"
##
## attr(,"contrasts")$Batch
## [1] "contr.treatment"
```

What do these three coefficients estimate?

3.4 Designs with an interaction

Your lab has a new cancer drug that you want to test, and one of the hallmarks of the cancer it is intended to treat is that the BRCA1 gene often gets mutated (it gets a stop codon inserted, so in effect it gets knocked out, and no longer creates a protein). The goal is to see how the drug affects tumors as compared to normal tissue in wild type animals, as well as animals that don't express BRCA1 any longer. In addition, you want to

know if the drug affects BRCA1 knockouts differently than wild type. Figure 1 shows just one example of an interaction.

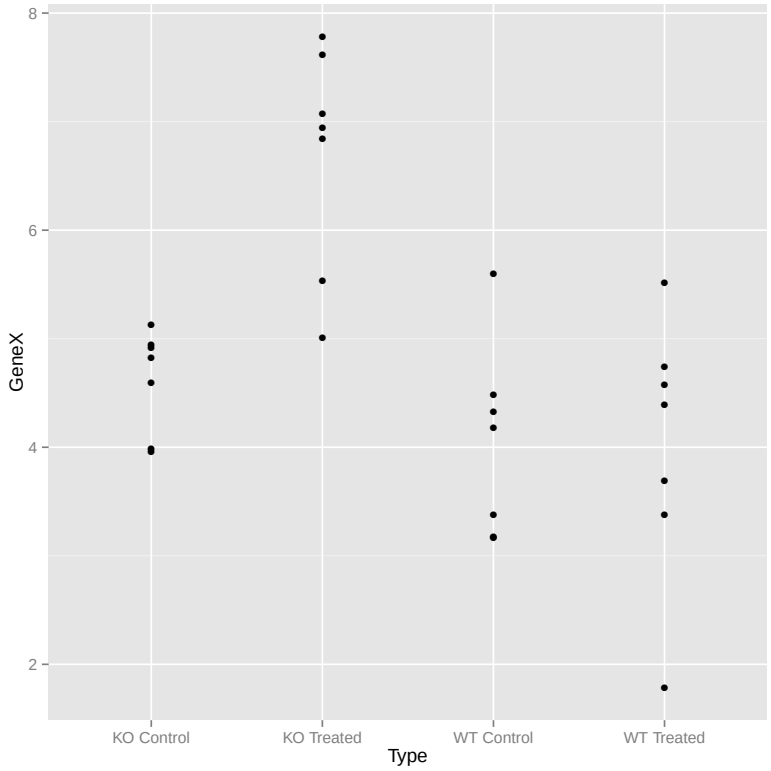

Figure 1: Interaction example

```
## let's assume that we have seven replicates per group
d.f4 <- data.frame(Treatment = rep(c("Control","Treated"), times = 2, each = 7),
                   Genotype = rep(c("WT","KO"), each = 14))

design5 <- model.matrix(~Treatment * Genotype, d.f4)
design5
```

```
##    (Intercept) TreatmentTreated GenotypeWT TreatmentTreated:GenotypeWT
## 1            1                 0          1                           0
## 2            1                 0          1                           0
## 3            1                 0          1                           0
## 4            1                 0          1                           0
## 5            1                 0          1                           0
## 6            1                 0          1                           0
## 7            1                 0          1                           0
## 8            1                 1          1                           1
## 9            1                 1          1                           1
## 10           1                 1          1                           1
## 11           1                 1          1                           1
## 12           1                 1          1                           1
## 13           1                 1          1                           1
```

```
## 14           1                 1          1                           1
## 15           1                 0          0                           0
## 16           1                 0          0                           0
## 17           1                 0          0                           0
## 18           1                 0          0                           0
## 19           1                 0          0                           0
## 20           1                 0          0                           0
## 21           1                 0          0                           0
## 22           1                 1          0                           0
## 23           1                 1          0                           0
## 24           1                 1          0                           0
## 25           1                 1          0                           0
## 26           1                 1          0                           0
## 27           1                 1          0                           0
## 28           1                 1          0                           0
## attr(,"assign")
## [1] 0 1 2 3
## attr(,"contrasts")
## attr(,"contrasts")$Treatment
## [1] "contr.treatment"
##
## attr(,"contrasts")$Genotype
## [1] "contr.treatment"
```

Note that we have used new notation here. Instead of Treatment + Genotype, we have used Treatment * Genotype. This is a short form for Treatment + Genotype + Treatment:Genotype, where we are telling R that we want to estimate the 'main effects' for Treatment and Genotype, as well as the interaction. What are all these coefficients estimating? Are these coefficients useful for answering the questions we have?

Let's take a different tactic.

```
newgrp <- factor(apply(d.f4, 1, paste, collapse = "_"))
design5.1 <- model.matrix(~ 0 + newgrp)
colnames(design5.1) <- gsub("newgrp", "", colnames(design5.1))
design5.1
```

```
##    Control_KO Control_WT Treated_KO Treated_WT
## 1           0          1          0          0
## 2           0          1          0          0
## 3           0          1          0          0
## 4           0          1          0          0
## 5           0          1          0          0
## 6           0          1          0          0
## 7           0          1          0          0
## 8           0          0          0          1
## 9           0          0          0          1
## 10          0          0          0          1
## 11          0          0          0          1
## 12          0          0          0          1
```

```
## 13          0          0          0          1
## 14          0          0          0          1
## 15          1          0          0          0
## 16          1          0          0          0
## 17          1          0          0          0
## 18          1          0          0          0
## 19          1          0          0          0
## 20          1          0          0          0
## 21          1          0          0          0
## 22          0          0          1          0
## 23          0          0          1          0
## 24          0          0          1          0
## 25          0          0          1          0
## 26          0          0          1          0
## 27          0          0          1          0
## 28          0          0          1          0
## attr(,"assign")
## [1] 1 1 1 1
## attr(,"contrasts")
## attr(,"contrasts")$newgrp
## [1] "contr.treatment"
```

What are these coefficients estimating? Would this be easier to interpret? How would you go about computing an interaction?

4 Debugging and dealing with errors

When writing your own code, and to a larger extent using other people's code, it is very useful to be able to debug when things go wrong. There is the obvious reason; if it's your own code, you want to be able to figure out where the problem is. But if it's somebody else's code, it is still useful to be able to debug, and it is actually more important in this regard because this is one of the best ways to improve your own R skills. Seeing how other people do things makes you a better coder because you learn things you might not have otherwise been exposed to, and you can see how those more adept than you solve problems.

But there is an even better reason to be able to debug. If you are trying to get something done, and you are getting errors, you can either Google around for an answer, or you can ask on the Bioconductor support site and wait (hope for) an answer, or you can take matters into your own hands and try to figure out if it is a bug or if you are doing something wrong. This of course assumes that you have already looked at the help page...

4.1 The `debug` function

The simplest way to debug functions is to use the `debug` function, which will put the function into a 'browser' and allow you to step through line by line. This works best for 'bare' functions, for example `lmFit` in the limma package.

```
library(limma)
debug(lmFit)
lmFit(y, design)
```

4.2 Namespaces

Some packages use highly modularized code, where a top-level function calls a bunch of lower-level functions that do much of the actual work. My affycoretools package is an example. These are a bit more difficult to debug, because you have to look through lots of small functions, and some of them may be buried in a namespace.

```
library("affycoretools")
vennPage
drawVenn
debug(drawVenn)
## access unexported functions with the ::: operator
affycoretools:::drawVenn
debug(affycoretools:::drawVenn)
```

Sometimes you hit an error in a package with lots of helper functions, and it's not clear where it comes from. In that case you can use `traceback`. An example is the error we used as an example above.

```
eBayes(lmFit(eset, design))
traceback()
```

Here we can see that the error occurred in the `ebayes` function, rather than in `eBayes`. We could then either use debug(ebayes) to step through that function, or just look at the function itself to see where we went wrong.

4.3 Debugging object-oriented functions

R has two different types of object oriented programming, where the idea is to be able to have a single function, say 'print', that will print out a useful summary, depending on what sort of object you want to print. In base R, the `print` function is a 'S3' object oriented function.

```
print
```

```
## function (x, ...)
## UseMethod("print")
## <bytecode: 0x1b6eb78>
## <environment: namespace:base>
```

```
head(methods(print))
```

```
## [1] "print.AES"     "print.Anova"   "print.Arima"   "print.AsIs"    "print.Bibtex"
## [6] "print.DLLInfo"
```

There are almost 200 `print` functions, that do different things depending on what you are trying to print. So if you wanted to print results from fitting an ANOVA model, you would end up using print.aov.

```
mod <-  aov(yield ~ block + N * P + K, npk)
class(mod)
print(mod)
print.aov
## no idea where this function lives. Use getAnywhere()
getAnywhere(print.aov)$where
debug(stats:::print.aov)
## can now step through debugger by doing print(mod)
```

A more sophisticated version of object oriented programming is the S4 system, which is the predominant system for Bioconductor. For S3 methods you use `method` to find out what functions operate on a particular object, but in S4 you use `showMethods` instead.

```
library("Biobase")
show
showMethods(show)
## we can look at individual functions by saying what class we
## want and setting includeDefs to TRUE
showMethods(show, class = "AnnotatedDataFrame", includeDefs = TRUE)
```

This tells us that when we want to look at an *AnnotatedDataFrame* object, R first examines the object, determines that it is of class *AnnotatedDataFrame*, and then uses a function called `.showAnnotatedDataFrame` to process it. Let's see what that function looks like.

```
.showAnnotatedDataFrame
## Hmmm
getAnywhere(.showAnnotatedDataFrame)
## can we use debug()?

df <- data.frame(x=1:6,
                 y=rep(c("Low", "High"),3),
                 z=I(LETTERS[1:6]),
                 row.names=paste("Sample", 1:6, sep="_"))

ad <- AnnotatedDataFrame(data=df)
show(ad)
debug(Biobase:::.showAnnotatedDataFrame)
## show(ad) to run through the debugger
```

How about the `show` method for an `eSet`?

```
showMethods(show, class = "eSet", includeDefs = TRUE)
## just a function there. Can we debug() directly?
debug(show)
data(sample.ExpressionSet)
sample.ExpressionSet
```

That appeared to work, but we didn't actually step through the function. Instead we need to use the `trace` function, which is a bit more sophisticated.

```
trace(show, tracer = browser, signature = c(x = "eSet"), where = getNamespace("Biobase"))
sample.ExpressionSet
```

And now we are able to step through the function that processes eSet objects.

5 Session information

The version of R and packages loaded when creating this vignette were:

```
toLatex(sessionInfo())
```

- R version 3.2.1 (2015-06-18), x86_64-unknown-linux-gnu
- Locale: LC_CTYPE=en_US.UTF-8, LC_NUMERIC=C, LC_TIME=en_US.UTF-8, LC_COLLATE=C, LC_MONETARY=en_US.UTF-8, LC_MESSAGES=en_US.UTF-8, LC_PAPER=en_US.UTF-8, LC_NAME=C, LC_ADDRESS=C, LC_TELEPHONE=C, LC_MEASUREMENT=en_US.UTF-8, LC_IDENTIFICATION=C
- Base packages: base, datasets, grDevices, graphics, methods, stats, utils
- Other packages: ggplot2 1.0.1, limma 3.25.13
- Loaded via a namespace (and not attached): BiocStyle 1.7.4, MASS 7.3-43, Rcpp 0.11.6, colorspace 1.2-6, digest 0.6.8, evaluate 0.7, formatR 1.2, grid 3.2.1, gtable 0.1.2, highr 0.5, knitr 1.10.5, labeling 0.3, magrittr 1.5, munsell 0.4.2, plyr 1.8.3, proto 0.3-10, reshape2 1.4.1, scales 0.2.5, stringi 0.5-5, stringr 1.0.0, tools 3.2.1