An Introduction to A Basic Limma Pipeline for Differential Expression

Rebekah Frye

2023-12-01

Overview

In class, we briefly talked about using the `lm()` function to fit a linear model. Linear regression is a powerful analysis tool that forms the basis of many more complicated analyses. One application of this is the use of linear regression to determine differential expression of genes under different conditions. Today, we'll be using the **Li**near **M**odels for **Mi**croarray Data, or **Limma** package to determine differential expression through linear regression of a simple one-factor experiment. The main goal of this tutorial is to introduce the very basics of the Limma package as well as some useful functions for handling and visualizing your data within this context.

The advantage of Limma is that it offers standardization in analyses rather than writing our own code for each comparison. The package also performs calculations that take into account information from other probes in the data (using *empirical Bayes* methods) during the course of our analysis. This ultimately gives us a slightly higher power than analyzing the a gene's expression in isolation, which can be helpful for researches who, like the real-life example we will look at in our GEO dataset, only have a handful of samples.

A detailed walk-through is provided in this document, which is entirely self- contained through R-markdown. A brief overview can be seen in the video and slides posted within the same OSF components page (see the wiki for more details of file locations and names).

A Note on Microarray Data Preprocessing

Please note that we are not covering processing raw expression data, which involves taking the images from microarrays and processing them into expression data. This is complicated and involves various different methods for different types of microarray, depending on the types of control your microarray uses. It's important to understand the specifics of your array for pre-processing. We are starting with a fully realized expression set, which we would normally need to build before we begin if we were generating our own data, but this is a good starting point for the basics. For the purposes of teaching only the basics of the Limma model and a few handy tools, we will also assume the data we are working with have already been pre-processed and normalized.

Required Packages and Libraries

You will need to first install the following packages if you do not already have them and then load the libraries:

```
### Install the biocLite package, which houses the pathway to Limma and then
# install Limma
# Code from (Ritchie, et al, 2015) available from 10.18129/B9.bioc.limma (R1)
```

```r
# Remove comments if you need to install the packages

# if (!require("BiocManager", quietly = TRUE))
#     install.packages("BiocManager")

## Installs Limma via Bioconductor
# BiocManager::install("limma")

## Installs GEOquery, which will allow us to load in our real world dataset
# later from the GEO database on NCBI
# BiocManager::install("GEOquery")


### Load the packages
library(limma)
library(Biobase)
library(BiocGenerics)
library(GEOquery)


### Load the data
# This the dataset we'll be using for our step-by-step tutorial. This is a
# data set of mock expression data of 500 probes across 26 samples of "Control"
# and "Case"

data(sample.ExpressionSet)

## Our second dataset will be loaded a different way to display a neat tool
# from the `Geoquery` package to retrieve published data from the NCBI's GEO
# database
```

Experimental Deisgn and Data

For this tutorial, we're going to use a **single factor design** for simplicity. That means that only one factor should be responsible for differences in gene expression (ex. a drug treatment, a disease condition, etc.). Keep in mind that Limma is based on the idea of linear regression, so we are essentially still working with lines. The generic equation for a line is:

$y = mx + b$

Where

- y = Output (dependent variable)
- m = Slope of the line
- X = Input/Predictor (independent variable)
- b = Intercept

Obviously, our design will be a little more complicated than this. Keep in mind that we'll be doing this for up to thousands of probes on our microarray. For our example, we're going to use the the `sample.ExpressionSet` data, which is already cleaned up for us in the form of an expression set. This is mock data that's been generated specifically to try out some of our functions on. For this dataset, we should have a total of 26 samples that are divided between a generic "Case" and "Control".

What we will end up looking at is the difference of the **expression level** of a gene between case and control. So our regression line for expression level needs to take into account the regression line for our control (or which group we wish to compare against) and the offset from that the comparison group (in this case, for our mock data, our `Case`s) for our other group of samples. You might imagine it's important, then, for the code to be able to determine which samples *do* have an offset (i.e. which are `Case` samples) and which do not (i.e. are `Control` samples). Let's take a look at our line:

Expression $\text{Level}_{\text{gene}}$ = mean $\text{expression}_{\text{control}}$ + (X * mean $\text{expression}_{\text{difference}}$) + error/randomness

This will be applied across all probes. Mathematically, "turning on/off" the offset term (i.e. X * mean $\text{expression}_{\text{difference}}$) is easy! The X variable will only ever be 0 or 1 in our model, and it will be used to either 'turn off' the offset term if a sample is a control sample (i.e. there is no offset because n*0 = 0, where n is any number, and therefore doesn't affect the expression level overall) or it will be used to 'turn on' the offset (i.e. B = 1, allowing that term to influence output, as a case sample potentially has an offset from control).

First, we have to tell the model which instances have a 0 (control) or a 1 (case) during calculation. For this, we have to build a **design matrix**, which keeps track of where to apply a 0 or 1 with the order kept with respect to our sample list's order.

Building a Design Matrix

For this single factor analysis, we want to design a matrix such that our `Control` samples are given a 0 (for no offset term) in our offset term and our `Case` samples are given a 1 (offset on). To do this, we'll use the function `model.matrix()`. If the data are cleanly formatted, this will be relatively simple, as the function will automatically assign samples with the same label to the same group and samples of a different label into another group. Let's first call our data.

So let's figure out where that list is hiding by taking a look at our data, which is in the form of an **expression set**. This is what Limma is typically used to look at.

```
sample.ExpressionSet
```

```
## ExpressionSet (storageMode: lockedEnvironment)
## assayData: 500 features, 26 samples
##   element names: exprs, se.exprs
## protocolData: none
## phenoData
##   sampleNames: A B ... Z (26 total)
##   varLabels: sex type score
##   varMetadata: labelDescription
## featureData: none
## experimentData: use 'experimentData(object)'
## Annotation: hgu95av2
```

These data are in an expression set with 26 samples and 500 probes. This is good! This allows us neat access to what we wish to know, and the structure of this expression set is fairly standard. You can actually click on the expression set in your environment to be able to maximize the drop down menus and look more extensively at the data listed. Here, however, you can see things like `phenoData`, which will contain phenotype information about our samples; we will find the subheading containing the `Case`/`Control` status of each of our samples here under `sample.ExpressionSet$type`. The `featureData` will contain information about the microarray chip design, including gene probes and annotations we can later apply to our results. The `assayData` will have the expression information we'll compare. These expression sets can potentially contain a *lot* of information, so they're worth combing through.

Let's look at our `Case`/`Control` samples:

```
## Calls the status of samples as case or control in order

sample.ExpressionSet$type
```

```
##  [1] Control Case    Control Case    Case    Control Case    Case    Case
## [10] Control Case    Control Case    Case    Case    Control Case    Control
## [19] Case    Case    Control Control Control Control Case    Case
## Levels: Case Control
```

```
## Note: the data used to identify the comparison you want to make may not
# be under the same heading in every expression set: explore your data
# to find what you're looking for, which is easy to do if you click the
# expression set in your environment
```

So we can see that we have our data labeled as `Case` and `Control` in the phenotypic metadata. This is good! We can use this to make our design matrix, as the `model.matrix()` package will automatically sort the cases and controls into their respective groups in the matrix (**ASSUMING** all entries are in the same syntax and format–i.e. not "Case" and "cases"–so practice uniform data entry skills!!). However, there *is* sometimes a slight issue with the default as we'll see...

But first, let's go ahead and try to form our matrix with the default function. We'll use the function `model.matrix()` and specify that we want to make our matrix with the `sample.ExpressionSet$type` data. The argument is based on a `~` to specify that this is the relevant groups we are interested in comparing and then the location containing that information. Because we need this design matrix to run the model, we will need to save it eventually as a different object to make clearer code, but I've already tipped your off that there's an issue, so we'll just run the function for now...

We will also call the `sample.ExpressionSet$type` data as well to double check our desgin matrix properly assigned 0 to `Control` samples and 1 to `Case` samples:

```
## This creates our design matrix based on the similarities AND LEVEL of the
# values listed in sample.ExpressionSet$type
model.matrix(~sample.ExpressionSet$type)
```

```
##    (Intercept) sample.ExpressionSet$typeControl
## 1            1                                1
## 2            1                                0
## 3            1                                1
## 4            1                                0
## 5            1                                0
## 6            1                                1
## 7            1                                0
## 8            1                                0
## 9            1                                0
## 10           1                                1
## 11           1                                0
## 12           1                                1
## 13           1                                0
## 14           1                                0
## 15           1                                0
## 16           1                                1
## 17           1                                0
## 18           1                                1
```

```
## 19           1                                    0
## 20           1                                    0
## 21           1                                    1
## 22           1                                    1
## 23           1                                    1
## 24           1                                    1
## 25           1                                    0
## 26           1                                    0
## attr(,"assign")
## [1] 0 1
## attr(,"contrasts")
## attr(,"contrasts")$`sample.ExpressionSet$type`
## [1] "contr.treatment"
```

```
## We'll just call the original data to make sure our defaults set the controls
# to 0 and cases to 1...
sample.ExpressionSet$type
```

```
##  [1] Control Case    Control Case    Case    Control Case    Case    Case
## [10] Control Case    Control Case    Case    Case    Control Case    Control
## [19] Case    Case    Control Control Control Control Case    Case
## Levels: Case Control
```

Uh oh! That's the *opposite* of what we wanted! *Why did this happen?* If you look under the `Levels` output, you'll see why: the default for the 'model.matrix()`function places`Case`first and`Control`second. This means the default will prioritize the highest level first and assign that group the 0 value. So let's just fix that using our first nifty function called`relevel()`when we make our next matrix. For this function, we'll specify that we want to take our`sample.ExpressionSet$type`and place`"Control"` as the higher level (recall that "Control" is a character and needs to be in quotes).

```
## We'll insert our relevel() inside the model matrix argument and save it
# as a new object, design.sample
design.sample <- model.matrix(~
               relevel(sample.ExpressionSet$type, "Control")
)

## We'll call our matrix now to make sure we're in the clear this time

design.sample
```

```
##    (Intercept) relevel(sample.ExpressionSet$type, "Control")Case
## 1            1                                                0
## 2            1                                                1
## 3            1                                                0
## 4            1                                                1
## 5            1                                                1
## 6            1                                                0
## 7            1                                                1
## 8            1                                                1
## 9            1                                                1
## 10           1                                                0
```

```
## 11           1                                                    1
## 12           1                                                    0
## 13           1                                                    1
## 14           1                                                    1
## 15           1                                                    1
## 16           1                                                    0
## 17           1                                                    1
## 18           1                                                    0
## 19           1                                                    1
## 20           1                                                    1
## 21           1                                                    0
## 22           1                                                    0
## 23           1                                                    0
## 24           1                                                    0
## 25           1                                                    1
## 26           1                                                    1
## attr(,"assign")
## [1] 0 1
## attr(,"contrasts")
## attr(,"contrasts")$`relevel(sample.ExpressionSet$type, "Control")`
## [1] "contr.treatment"
```

As you can see in our comparison, we've done what we set out to do! We can now use `design.sample` as our design matrix to fit our model. If you ever need to locate the predictor status, call the expression set and check under the phenoData. Ensure your data are in the correctly leveled. Now let's look a little closer.

As you can see, the intercept is always 1. This essentially is because we are using the control line to compare offset against, so it should always be turned on in the linear regression, regardless of case/control status. However, if we look at the case/control status, this is where our linear model will find that X term we mentioned at the start: for `Control` samples, the model will plug in a 0, and it will plug in a 1 for `Case`. This allows the offset of `Case` to be valuable information. We can see that the `Case`/`Control` status in our matrix, at least for the first 5 samples, matches, so we did successfully generate our matrix!

Fit the Linear Model using lmFit()

Now that we have assigned appropriate X terms to our samples, let's actually fit the linear regression model. We'll use the `lmFit()` function through Limma, or the Linear Model for Series of Arrays function. That may sound very fancy, but the `lmFit()` function is extremely simple in a single factor design like this, where we've yet to take into account any covariates. The generic argument is as follows:

```
lmFit(ExpressionSetDataToCompare, DesignMatrix)
```

This this case, we will use our expression set, `sample.ExpressionSet` as our data to compare and `sample.designmatrix` as our design matrix.

```
## Fitting the linear model, saved as the new object fit
fit <- lmFit(sample.ExpressionSet, design.sample)
```

Now we're going to use the `eBayes()` function to calculate the t-statistic for our data, which will give us our differential expression. To do this, all we need in the argument is our fitted model, and the function will take care of the rest! Please recall that this method takes advantage of information from the fact that all of our probes were on the same chip, so we will need to use the **adjusted** p-values for any determination of significance, as we must correct for multiple testing.

```
## Caluclates out t value

fit.Bayes <- eBayes(fit)
```

That's it! You've run Limma. Now let's actually look at our data. We'll use the function `topTable()` and specify our final model so that we're only looking at the top 10 differentially expressed genes:

```
topTable(fit.Bayes)
```

```
##                      logFC    AveExpr         t     P.Value adj.P.Val         B
## 31396_r_at     -462.473455 2504.39385 -3.427734 0.002125535 0.4056444 -4.005602
## 31573_at        832.300424 2034.80623  3.380489 0.002390602 0.4056444 -4.018612
## 31568_at        949.925697 2866.91769  3.264082 0.003187771 0.4056444 -4.050804
## 31440_at       -109.039248  659.05592 -3.175011 0.003965593 0.4056444 -4.075540
## 31341_at         85.221300  334.96943  3.061210 0.005227940 0.4056444 -4.107226
## 31721_at        -56.526530  190.89790 -3.040126 0.005500706 0.4056444 -4.113102
## 31461_at        -19.140850   41.53356 -3.026878 0.005679021 0.4056444 -4.116795
## AFFX-MurIL4_at   -9.901622   23.06363 -2.893427 0.007811270 0.4402824 -4.153983
## 31356_at        -54.481345  190.26500 -2.887323 0.007925083 0.4402824 -4.155683
## 31535_i_at      -60.629715  266.86950 -2.810016 0.009509630 0.4420851 -4.177186
```

This output is very sample. Here, for each probe, you can see the log(Fold change), which describes the amount of difference in expression between the groups being compared. A positive fold change denotes an increase in expression over control while a negative one denotes a decrease in expression over control. Additionally, you can see the t- and p-values, which indicate the significance of the hits. In our first case, none of these values are statistically significant when looking at the adjusted p-value, and all of the t-values are quite small.

Typically, this output might also include a wealth of information taken from the feature data in the file, including gene annotation, chromosomal location, gene ontology, and pathways that the top hits may be involved with. For this dataset, there was no metadata like this in our expression set, so it doesn't appear in our output. Again, note that the `topTable()` function **only** shows our top 10 hits. We can actually alter this by specifying in the argument with `number = #` how many hits we would like to see in place of the `#` as opposed to the default of 10. For example, let's try looking at the top 20 hits instead:

```
topTable(fit.Bayes, number = 20)
```

```
##                      logFC    AveExpr         t     P.Value adj.P.Val
## 31396_r_at     -462.473455 2504.39385 -3.427734 0.002125535 0.4056444
## 31573_at        832.300424 2034.80623  3.380489 0.002390602 0.4056444
## 31568_at        949.925697 2866.91769  3.264082 0.003187771 0.4056444
## 31440_at       -109.039248  659.05592 -3.175011 0.003965593 0.4056444
## 31341_at         85.221300  334.96943  3.061210 0.005227940 0.4056444
## 31721_at        -56.526530  190.89790 -3.040126 0.005500706 0.4056444
## 31461_at        -19.140850   41.53356 -3.026878 0.005679021 0.4056444
## AFFX-MurIL4_at   -9.901622   23.06363 -2.893427 0.007811270 0.4402824
## 31356_at        -54.481345  190.26500 -2.887323 0.007925083 0.4402824
## 31535_i_at      -60.629715  266.86950 -2.810016 0.009509630 0.4420851
## 31511_at        759.582545 3256.33615  2.743911 0.011097819 0.4420851
## 31583_at        842.628303 2819.34808  2.701214 0.012253105 0.4420851
## 31580_at        -11.813985   26.58297 -2.693953 0.012460491 0.4420851
## AFFX-TrpnX-M_at -20.286091   46.27918 -2.613243 0.014998300 0.4420851
```

```
## 31345_at         -13.159862   27.74779 -2.570169 0.016542970 0.4420851
## 31419_r_at      -231.946242 1447.44577 -2.569903 0.016552940 0.4420851
## 31683_at         -13.246698   51.58377 -2.485012 0.020041593 0.4420851
## 31447_at         -15.894345   55.46730 -2.482416 0.020158311 0.4420851
## 31631_f_at        15.732713  -32.66960  2.463896 0.021009367 0.4420851
## 31722_at         795.504061 3850.45192  2.442140 0.022051529 0.4420851
##                        B
## 31396_r_at      -4.005602
## 31573_at        -4.018612
## 31568_at        -4.050804
## 31440_at        -4.075540
## 31341_at        -4.107226
## 31721_at        -4.113102
## 31461_at        -4.116795
## AFFX-MurIL4_at  -4.153983
## 31356_at        -4.155683
## 31535_i_at      -4.177186
## 31511_at        -4.195529
## 31583_at        -4.207348
## 31580_at        -4.209355
## AFFX-TrpnX-M_at -4.231605
## 31345_at        -4.243427
## 31419_r_at      -4.243500
## 31683_at        -4.266669
## 31447_at        -4.267374
## 31631_f_at      -4.272402
## 31722_at        -4.278295
```

If we want to look at our data further, let's use the function `decideTests()` which is included in the Limma package. This function generates a matrix, which we can see below if we input our model:

```
## Displays a matrix with a -1 for down-regulated genes, +1 for up-regulated
# genes, and 0 for no statistically significant difference in expression

decideTests(fit.Bayes)
```

```
## TestResults matrix
##                 (Intercept) relevel(sample.ExpressionSet$type, "Control")Case
## AFFX-MurIL2_at            1                                                 0
## AFFX-MurIL10_at           1                                                 0
## AFFX-MurIL4_at            1                                                 0
## AFFX-MurFAS_at            1                                                 0
## AFFX-BioB-5_at            1                                                 0
## 495 more rows ...
```

Here, we don't especially care about the results in this matrix for `Intercept`. What's interesting is the `sample.ExpressionSet$type`. A 0 in this matrix indicates the probe is not differentially expressed in either direction (i.e. it's not statistically significant). A -1 indicates that that gene is statistically significantly down-regulated. A +1 indicates that gene is statistically significantly unregulated. Let's do ourselves a favor and sum up all the values in each category by encapsulating our `decideTests()` function in the `summary()` function:

```
## Summarizes the number of up- and down- regulated genes and the number that
# are not statistically significant
summary(
  decideTests(fit.Bayes)
  )
```

```
##        (Intercept) relevel(sample.ExpressionSet$type, "Control")Case
## Down            12                                              0
## NotSig          46                                            500
## Up             442                                              0
```

Again, we'll ignore the `(Intercept)` column and focus on the `sample.ExpressionSet$type_edit`, where we can see all 500 genes represented in these data are not significantly different in expression from one another. This is good because this is the result we saw in out `topTable()` output: even the highest degree of differential regulation in our top 10 hits did not reach statistical significance.

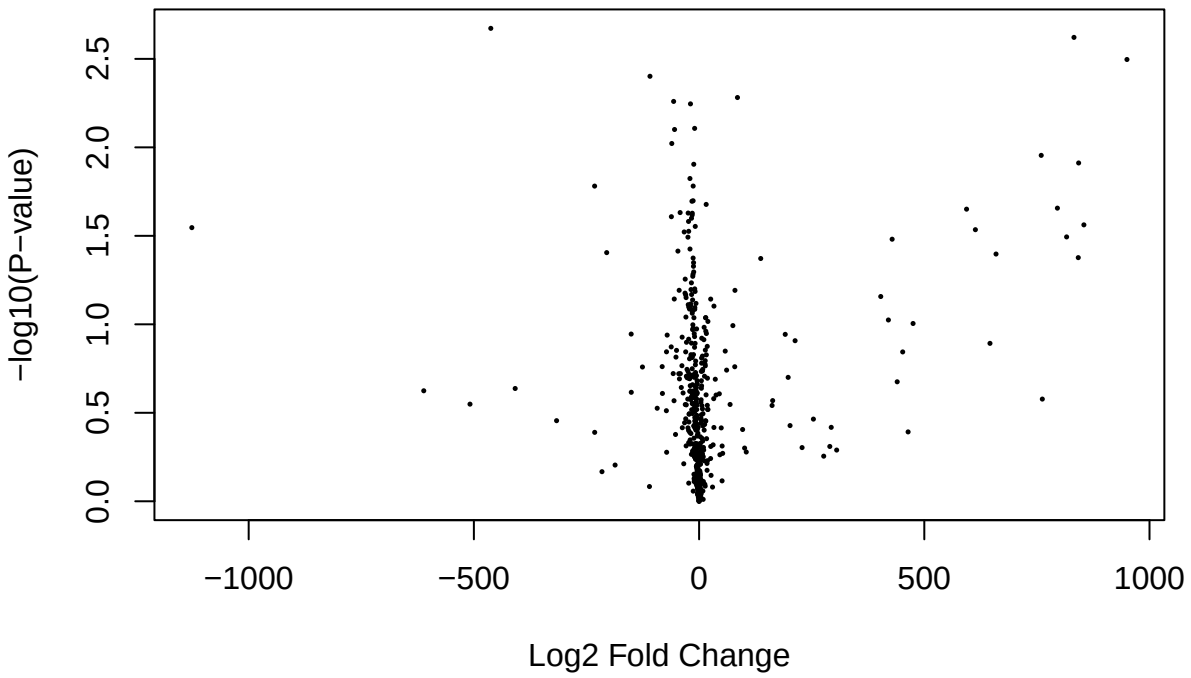
Visualizing our Data with `volcanoplot()`

Now, let's plot our data! We'll use a volcano plot using Limma's built-in `volcanoplot()` function. The argument, again, is quite simple:

`volcanoplot(model)`

We saw earlier that we have no statistically significant hits, so we won't add anything to this graph that might misleadingly highlight any of the genes as though they were differentially expressed. This will be different for our real data.

```
volcanoplot(fit.Bayes,
            coef = 2)
```



There we go! Obviously this dataset was made up specifically to practice these functions and doesn't represent real findings. Let's switch to something a little more interesting and see if we can find something of note as well as practice importing previously generated data from NCBI's GEO.

Real Data Example with `getGEO()`

Let's look at another simple experiment. We'll use a dataset from the National Institute for Bioinformatics (NCBI) Gene Expression Omnibus (GEO) that looks at gene expression differences between parental cell line (Hap1) and a mutated daughter cell (Baz2B-KO) that was derived from the Hap1 line with a knock-out for Baz2B, which is involved in chromatin remodeling. First, we'll have to add an additional step to get our data using the `GEOQuery` package. The `getGEO()` function will grab data for us from any GEO dataset so long as we have the GEO accession number of the submitted study of interest. In our case, that number is `GSE218705`. So we'll use the argument `GEO = AccessionNumber` to grab out expression set and save it as a new object, real.eset. To prove we grabbed the right thing, we'll call the data and have a look at it as well:

```
## This will grab our data from NCBI's GEO and save it as an object, real.est
real.eset <- getGEO(GEO="GSE218705")

## This calls out expression set to double check we pulled the right study
real.eset
```

```
## $GSE218705_series_matrix.txt.gz
## ExpressionSet (storageMode: lockedEnvironment)
## assayData: 53617 features, 4 samples
##   element names: exprs
```

```
## protocolData: none
## phenoData
##   sampleNames: GSM6755362 GSM6755363 GSM6755364 GSM6755365
##   varLabels: title geo_accession ... genotype:ch1 (35 total)
##   varMetadata: labelDescription
## featureData
##   featureNames: 16650001 16650003 ... 17127721 (53617 total)
##   fvarLabels: ID RANGE_STRAND ... RANGE_GB (8 total)
##   fvarMetadata: Column Description labelDescription
## experimentData: use 'experimentData(object)'
##   pubMedIds: 38000389
## Annotation: GPL16686
```

Great! If you search the GEO accession number, you can find the submission page on GEO. This usually has really good information on the study performed, including the type of experiment that generated the expression data, the array it was performed on, and details about the experiment. You can see there that we should have 4 samples total, two control and two knock-out, which we can confirm from our called data above. Now we need to find the codes for our phenotype data to determine which samples are control and which are knock-out.

If we look at the phenotype data, we can see that it's located under `source_name_ch1`, so let's have a look and see what form it takes:

```
## This will call the sample list to see which are hap1 or BazB2-KO samples
real.eset$GSE218705_series_matrix.txt.gz$source_name_ch1
```

```
## [1] "Hap1 cell line"          "Hap1 cell line"
## [3] "Hap1 cell line Baz2B-KO" "Hap1 cell line Baz2B-KO"
```

Here, we see the structure of the sample labels, `Hap1 cell line` for control and `Hap1 cell line Baz2B-KO` for KO samples. Let's go ahead and try out our model and see if the defaults settings will work this time.

```
## This makes our design matrix
real.design <- model.matrix(~real.eset$GSE218705_series_matrix.txt.gz$
                              source_name_ch1)

## This calls our design matrix to double check ourselves
real.design
```

```
##   (Intercept)
## 1           1
## 2           1
## 3           1
## 4           1
##   real.eset$GSE218705_series_matrix.txt.gz$source_name_ch1Hap1 cell line Baz2B-KO
## 1                                                                               0
## 2                                                                               0
## 3                                                                               1
## 4                                                                               1
## attr(,"assign")
## [1] 0 1
## attr(,"contrasts")
## attr(,"contrasts")$`real.eset$GSE218705_series_matrix.txt.gz$source_name_ch1`
## [1] "contr.treatment"
```

That's correct! Hooray for defaults! Now let's use that and fit our model:

```
## Fitting our model
fit.real <- lmFit(real.eset$GSE218705_series_matrix.txt.gz, real.design)
```

Let's calculate our t-statistics...

```
## Calculating our t statistics
fit.Bayes.real <- eBayes(fit.real)
```

Now let's view our top hits!

```
## Viewing our top hits! Remember that the default is 10
topTable(fit.Bayes.real)
```

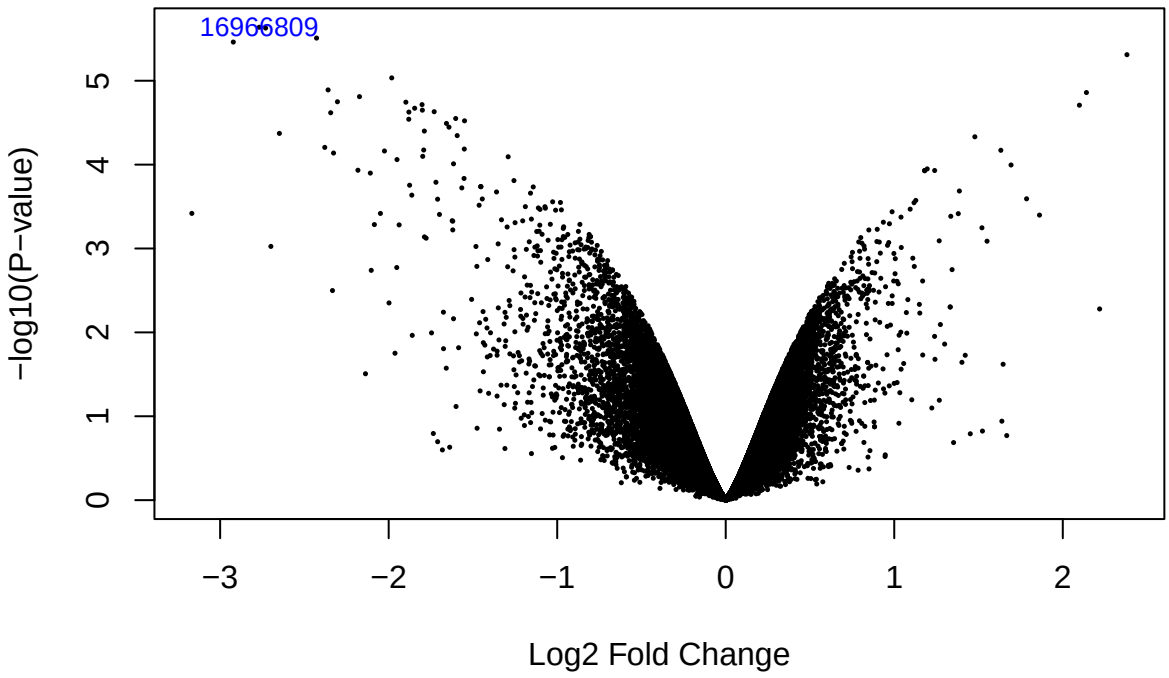
```
##                ID RANGE_STRAND RANGE_START RANGE_END total_probes    GB_ACC
## 16966809 16966809            +    55095264  55164414           30
## 17122488 17122488            +           1       417           17
## 16942958 16942958            +    89156674  89531284           30 NM_005233
## 16820437 16820437            +    68569418  68569561           21
## 16798479 16798479            +    27111510  27194357           26
## 16650883 16650883          ---           0         0            4
## 17048190 17048190            +    90095738  90839905           24 NM_012395
## 17105706 17105706            +   102611373 102613397           18 NM_016303
## 16992346 16992346                       NA        NA           NA
## 17070110 17070110            +    75736772  75767264           20 NM_015886
##                          SPOT_ID     RANGE_GB     logFC  AveExpr         t
## 16966809   chr4:55095264-55164414 NC_000004.11 -2.768259 5.703066 -23.43230
## 17122488               chrU:1-417         chrU -2.730162 5.815162 -23.33167
## 16942958   chr3:89156674-89531284 NC_000003.11 -2.427913 5.491787 -22.08707
## 16820437  chr16:68569418-68569561  NC_000016.9 -2.921847 2.925353 -21.61293
## 16798479  chr15:27111510-27194357  NC_000015.9  2.380759 6.857446  20.16928
## 16650883                --unknown              -1.981863 2.929512 -17.76037
## 17048190   chr7:90095738-90839905 NC_000007.13 -2.359597 4.925660 -16.62431
## 17105706 chrX:102611373-102613397 NC_000023.10  2.140324 6.193697  16.38628
## 16992346                --unknown              -2.173168 4.417316 -16.02153
## 17070110   chr8:75736772-75767264 NC_000008.10 -2.304033 6.180895 -15.57540
##               P.Value  adj.P.Val        B
## 16966809 2.303297e-06 0.04634603 2.395631
## 17122488 2.353691e-06 0.04634603 2.391975
## 16942958 3.100459e-06 0.04634603 2.343015
## 16820437 3.457562e-06 0.04634603 2.322372
## 16798479 4.891719e-06 0.05245586 2.251528
## 16650883 9.249376e-06 0.07170295 2.099149
## 17048190 1.286857e-05 0.07170295 2.007561
## 17105706 1.382869e-05 0.07170295 1.986375
## 16992346 1.547154e-05 0.07170295 1.952417
## 17070110 1.780916e-05 0.07170295 1.908256
```

Here, you can see we actually have some additional information. This includes some information about our top hits. Of note, you can see the `GB_Acc`, which is the GenBank Accession number so that you can look up your hits, along with other information that might be useful in looking deeper into location and function

of your hits. What information is displayed here depends on your methods and what's contained in the expression set's metadata. Some even contain gene ontology information and the like, so it's always nice to know what's included before you begin!

Now, we can see that we have 4 significant hits (if only barely), so we're going to add an additional term to our `volcanoplot()` to highlight those hits: this is the `highlight = #` argument, where the number of top hits you want to highlight is in place of the `#`. For now, we'll only add a single hit to highlight, but feel free to play around with it! :

```
## Generates a volcano plot with the top hit shown in blue and labeled
# with an identifier from your metadata - note that this identifier might
# be different depending on what you have available in your metadata.
volcanoplot(fit.Bayes.real,
            coef = 2,
            highlight = 1)
```



That's it! You're on your way! If you would like another practice, you can try to download the `breastcancerVDX` dataset and try to set up a model and test it on your own! This is a commonly used dataset for tutorials in Limma with a simple design, so it's very easy to check your work.

Note that, in our case, our genes of interest are down-regulated in our KO line as compared to control, which is a good logical check!

Things Get Complicated Quickly

Note that we only discussed a single factor design... without covariates. Normally, things like sex, age, batch number, etc. might be of interest and potentially have interactions with your expression data that

can be informative. Additionally, you may have more than 2 groups you wish to compare (ex. you want to compare differential expression across a mutant and wild-type cell line with two subsets for each–those that have received a treatment and those that have not). This requires a different set of arguments to add contrasts to our model to account for different descriptions, very much like the 0's and 1's we designated for our design matrix.

References

- 1 - Limma installation code
- 2 - Subscripts in RMD
- 3 - Generally trying to learn Limma
- 4 - sample.ExpressionSet data
- 5 - Simple Limma pipeline
- 6 - Code modified to replace case with 1 and control with 0
- 7 - Code modified to show the table of up and downregulated genes
- 8 - Code modified for use of GEOquery
- 9 - Code modified to change default number in TopTable
- 10 - Knock-out data use for second half of the tutorial
- 11 - Code for releveling the type data to help make the design matrix