Data Expeditions

Exploring the genetic basis of complex colony morphology

Liana T. Burghardt and Colin S. Maxwell

Fall 2014

Contents

1 Day 1

1.1 Using this workbook

Learning a programming language is much like learning to speak a natural language: success comes through practice. A big part of learning to program is actually making mistakes and learning to figure out what you did wrong. One thing that often trips up new students of a programming language is the syntax. Typing out 'grammatically correct' lines of code means learning to see where commas and parentheses and the like can be placed and conversely the consequences of misplaced marks. To start to learn how to speak a programming language, it is important to learn how to type it. Therefore, please avoid copying and pasting the code below. It will be more helpful to you in the long-run if you take the time to write the code out by hand.

The workbook will contain blocks of code like this one:

```
print("Hello world")
```

Below some of these blocks will be the results of the code:

```
[1] "Hello world"
```

This will allow you to compare the results of typing in your code to what we expect you to get when you type in the code.

1.2 Getting started and oriented

1.2.1 Opening R

Open up R by clicking on the icon. You should see one window appear that is called the R Console. This is the interface where you enter R commands and recieve text output from R. The greater than sign indicates that R is ready to recieve a command. R is a very versatile language that allows you to do everything from evaluate basic math to compute statistics to create complex graphics.

1.2.2 Arithmetic in R

At its very simplest R can be used as a calculator. Let's try this out first. Type `2+2` in the console and press enter:

```
2+2
```

R adds the numbers and returns the answer:

```
[1] 4
```

There are lots of basic arithmatic symbols that R understands:

```
2-2  # subtraction
2*4  # multiplication
4/2  # division
4^2  # exponentiation
```

```
[1] 0
[1] 8
[1] 2
[1] 16
```

Notice that R didn't interpret the words "subtraction", "multiplication", etc. That's because these words were preceded by the # sign. The # sign is a way of telling R that these words are not code. They are often referred to as 'comments' since they can be used to mark what a particular line of code does.

If you omit the #, R will tell you that you have written grammatically incorrect code by generating an error message.

```
2-2 subtraction
```

```
Error: unexpected symbol in "2-2 subtraction"
```

You can also combine the standard operators together according to the standard orders of operation. Parentheses can also be used to group operations that should happen first. Here the 2's are added together first, multipled by 5 and then 8 is subtracted.

```
(2+2)*5-8
```

```
[1] 12
```

1.2.3 Assigning Variables

You can assign the results of some calculation (or any other line of code) to a variable by using the <- sign.

```
answer <- 2+2
answer #type 'answer' into the console
```

```
[1] 4
```

Now that you have assigned something to this variable, you can access it any time you want. However, if you've not assigned something to a variable R will raise an error if you try to access it:

```
answer3
```

```
Error: object 'answer3' not found
```

You can also assign variables using the = sign.

```
answer2 = 2+4
answer2
```

```
[1] 6
```

1.2.4 Functions

Functions perform a defined operation on an object. Functions are called by invoking the function name followed by parentheses containing zero or more 'arguments' to the function. For instance, there is a function that will sum a vector. Vectors are simply a string of elements (e.g. numbers or characters). The simplest way to create a vector at the interactive prompt is to use the `c()` function, which is short hand for 'combine' or 'concatenate'.

```
numberVector <- c(2,4,6,8)
characterVector <- c('joe','bob','fred')
```

Now that you've made the vectors, you can look at them any time you want by typing their names into the R console.

```
numberVector
characterVector
```

```
[1] 2 4 6 8
[1] "joe"  "bob"  "fred"
```

Now, if you want to sum the numbers in the number vector, you can use the `sum()` function:

```
sum( numberVector )
```

```
[1] 20
```

However, you can't add "joe" to "bob", so R will make an error if you try to sum the character vector:

```
sum( characterVector )
```

```
Error in sum(characterVector) : invalid 'type' (character) of argument
```

You can get help for any function by typing a questionmark followed by the function name (try typing in `?sum` to the R console)

1.2.5 Graphics in R

Other functions can create graphics. For instance we can plot a scatter plot:

```
my.x<-c(1,5,3,2,5,6,7,8,9,7) #define a vector of values for x
my.y<- 2 ^ my.x #Define a vector y
# the plot function expects the first two arguments to be vectors
plot(my.x,my.y)
```

You can make your plot look nicer by adding optional arguments. For example, you can add axis titles by specifying the `xlab` and/or `ylab` arguments (try leaving one of these out to see what happens).

```
plot(my.x,my.y, xlab="Underwear acquired", ylab="Profit")
```

You can add a title with the argument `main`:

```
plot(my.x, my.y, xlab="Underwear acquired", ylab="Profit", main="Step 3")
```

You can also change the point type with the `pch` argument and the color of the points with the `col` argument. Notice that it doesn't matter what order the arguments are specified in.

```
plot(my.x,my.y, main="Step 3",xlab="Underwear acquired",
     ylab="Profit", pch=19, col="red")
```

You can also add a sub-title using `sub`. For functions that have lots of arguments, it can be helpful to indent them so that each argument gets its own line. You can hit enter after each comma, and R will wait to evaluate the line of code until you get to the final finishing parentheses.

```
plot(my.x,
     my.y,
     main="Step 3",
     xlab="Underwear acquired",
     ylab="Profit",
     pch=19,
     col="red",
     sub="(TM) GnomeCorp")
```

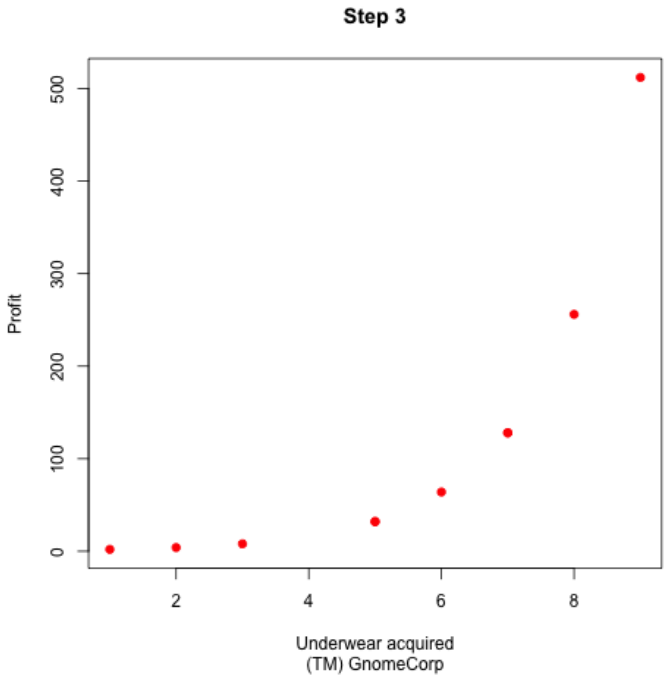

Figure 1.1: A plot demonstrating some of the optional arguments that can be supplied

For a list of all the optional arguments you can pass to `plot()`, see the help page by typing in `?plot`.

1.2.6 Packages in R

R is an open source language so you have access to and can edit all the functions that other people create and use. This makes R an incredibly powerful language. If R gave you access to all the functions that people have ever written, you would not have any space left on your computer! This is what packages are for. They are essentially conglomerations of code designed to perform certain tasks. Once you install and load a package you have access to all the functions it contains...

To install a package you use the `install.packages()` command. We have already installed all the necessary packages for this module on your computer so all you will have to do is to load packages into the working memory of your R session so you can have access to the functions in those packages.

For example, the package `RColorBrewer` has some functions in it to help you add color to plots. Inside the package there is a function called `display.brewer.all()`. However, if we haven't loaded the packages, we can't see the function.

```
display.brewer.all()
```

```
Error: could not find function "display.brewer.all"
```

```
library(RColorBrewer)
display.brewer.all()
```

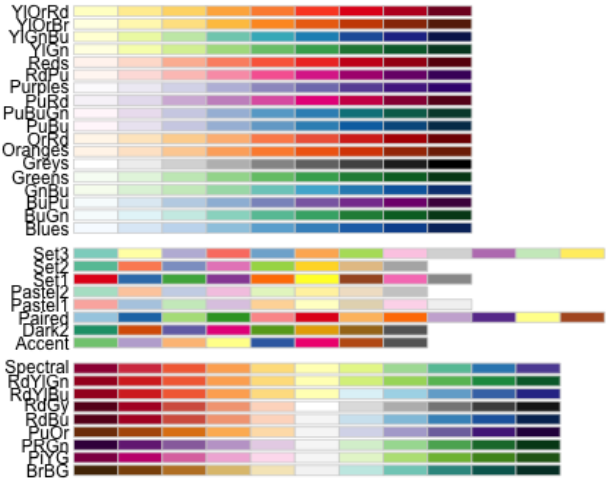

Figure 1.2: The plot coming from the `display.brewer.all()` function.

A second way to bring in new functions into R is to use the source command. We will see how to use the source command when we are writing our script below.

1.3 Creating an R script

An R script serves as an editable template of the code you are developing so you can open it up again later. Basically you copy and paste text from this script into your R console to execute the code you have written. From now on in this teaching module you will be creating a R script that will serve as a record of all the coding we do in this class!

First, open the drive called "Bio 201-202" that is located on the Desktop of the computer. Drag the folder called "$\text{data}_{\text{expeditions}}$" onto the Desktop. This puts the data we'll be working with somewhere that R can access it.

Click on the file menu in R and highlight "New Document". I new window will appear called "untitled". Click on the file menu again and highlight "Save as". Name your file "ComplexColonyCode-initials.R". Substitute in your and your partners initials for the words "initials" in the code name above.

1.3.1 Set working directory

If we want to do any data analysis in R we need to load that data into the program. The first step for doing this is to tell R where your data actually is. You do this by defining your working directory. Type the following command as the first line of your R script:

```
1 | getwd()
```

Then copy and paste this text into the R console. When you press return it should show you where R is currently looking for your files. Now you need to tell R where the data files are that you want to work on for this script is. We have put it in a folder called "CCAnalysis" on the desktop. Type the following command as the second line of your R script:

```
1 | setwd("~/Desktop/data_expeditions/")
```

As before, copy and paste this text into the R console. This function sets the working directory to that indicated within the quotes. You can type `getwd()` into your console again to check that the working directory has changed.

1.3.2 Importing packages and functions

Next, we'll import the libraries and packages we'll need for our analysis. First, we will import some functions that have will make our analysis easier.

```
1 | source("ccm_functions.R")
```

1.3.3 Importing Data into R

Now we need to load two data files into R. First lets look at the data in microsoft excel where you are likely to be more familar with exploring data. The first dataset is called "microarray.csv". I has 36 columns that reflect 36 different yeast genotypes. It has 1,141 rows that each reflect the relative expression level of a different gene. The genes have names like 'FOX2' and 'COA1'. Note the general values of the expression level. Now go back into your R script and type the following command and copy and paste it into your console:

```
microarray <- read.csv("microarray.csv", row.names=1)
```

The first argument to `read.csv()` is just the name of the file you want to read in. The second argument tells R that the rownames of the data are in the first column.

We can now explore the dataset you just imported into R by typing the following commands in your script and running them in R. The data is in the form of a big matrix with the rownames of the matrix corresponding to the genes and the column names corresponding to the genotypes. You can look at the dimensions of the data in the form (rows, columns):

```
dim(microarray)
```

You can look at the highest and the lowest expression values:

```
range(microarray)
```

You can look at the first 5 rows of the dataset (the expression level of the first 5 genes across all the offspring).

```
head(microarray)
```

You can look at the last 5 rows of the dataset

```
tail(microarray)
```

The second dataset is called 'phenotypes.csv'. This dataset describes whether each of the 36 yeast genotypes makes a 'simple' or a 'complex' colony.

```
phenotypes<-read.csv("phenotypes.csv", header=TRUE)
```

1.4 Subsetting datasets

The 'microarray' data is very large. Often, to do an analysis, you will want to focus on a smaller part of the data. To do this, you can use the function `get.subset()`. You can use `get.subset()` to select either the rows and or columns of a matrix that you want. For example, to select the genes 'FOX2' and 'COA1', you would write:

```
twoGenes=get.subset(microarray, rows=c("FOX2", "COA1"))
twoGenes
```

Let's choose five samples and six genes to focus on. We'll revisit this data in the next class period. First, we'll create two vectors that list the genes and the samples that we'll be looking at.

```
samplesToSubset = c("num1", "num2", "num34", "num31", "num55")
genesToSubset =c( "ALD3", "AMD1", "ISR1", "ASN2", "RMT2", "PHO89")
```

Look at the phenotypes data. Notice that `samplesToSubset` contains genotypes that make both complex and simple colonies.

We can make two smaller versions of the data that is restricted to either the six genes with all the samples or all the genes with the five samples. Explore these datasets using the functions you learned above.

```
sampleSubset <- get.subset(microarray, cols=samplesToSubset)
geneSubset <- get.subset(microarray, rows=genesToSubset)
```

We can make an even smaller version of the data by taking a subset of our subset.

```
geneAndSampleSubset <- get.subset(sampleSubset, rows=genesToSubset)
```

1.5 Graphing the subset

What does our data look like? To try to find out, let's make a dotplot of the smallest subset of our data. We will use the function `gene.dotplot()`. Each dot shows the expression level of a gene for each of the five samples.

```
gene.dotplot(geneAndSampleSubset)
```

We can also plot all of the samples:

```
gene.dotplot(geneSubset)
```

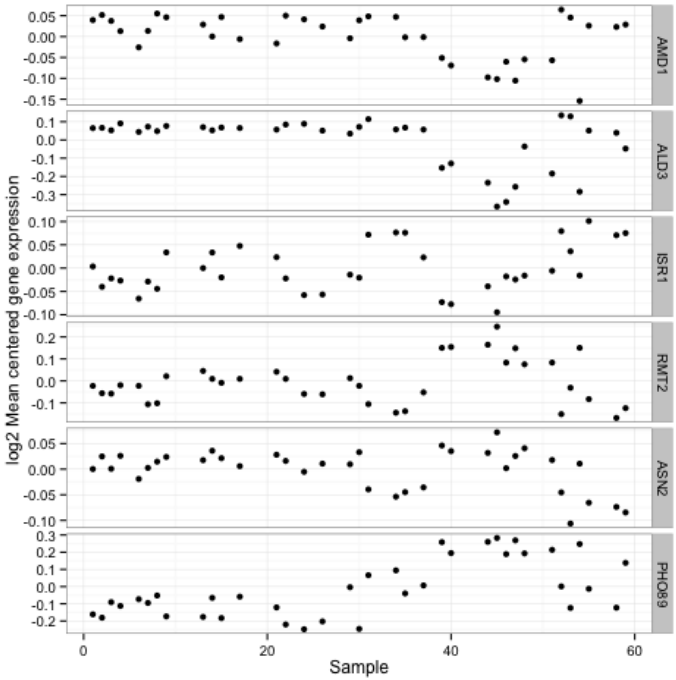

Figure 1.3: A plot of all samples for the selected 6 genes

Although there are clearly some trends in the data, these plots are hard to read. In the next workbook we'll learn a better way to represent these large datasets called a heatmap.

1.6 Saving your data

It's very important that after you're done with your data that you drag the script that you wrote into the drive called "Bio 201-202." Anything left on the Desktop (or anywhere else on the computer) will be deleted when the computer shuts down.

2 Day 2

2.1 Hierarchical clustering algorithm

For your reference, this is the hierarchical clustering algorithm we will be asking you to do:

1. Find the pair of items or clusters that has the closest distance to each other.
2. Merge these items or clusters into one cluster. To show this visually, draw a line connecting them on the dendrogram. The line should be draw so that it is even with the distance between the two clusters or items.
3. Re-compute the distances between the items or clusters. The distance between clusters is the average of the distances between each pair of the items.
4. Go to (1) until the items are all in clusters.

2.2 Setting up your workspace

Open up the script that you made in the last class period. It should look something like this:

```
setwd("~/Desktop/data_expeditions")

source("ccm_functions.R")

samplesToSubset = c("num2", "num1", "num55", "num31", "num34")
genesToSubset = c( "ALD3", "AMD1", "ISR1", "ASN2", "RMT2", "PHO89")

microarray <- read.csv("microarray.csv", row.names=1)
phenotypes<-read.csv("phenotypes.csv", header=TRUE)

sampleSubset <- get.subset(microarray, cols=samplesToSubset)
geneSubset <- get.subset(microarray, rows=genesToSubset)
geneAndSampleSubset <- get.subset(sampleSubset, rows=genesToSubset)
```

Feel free to delete all the extra lines of code that aren't shown above, since they aren't necessary for what we're going to do. Execute all the code in the script so that R is setup.

2.3 Correlation as distance

2.3.1 Clustering genes

We have seen that correlation is a number between -1 and 1 that expresses how linearly associated two variables are. Similarly, 1-correlation is a measure of how different two variables are. If two variables are perfectly anti-correlated with a correlation of -1, then 1-(-1) = 2. Conversely, if they are perfectly correlated, 1-1=0. This measure can be applied to both genotypes and to genes. We'll look at correlations between different genes first. Type the following into the R terminal:

```
plotPairs(geneSubset, c("ASN2", "ISR1"))
```

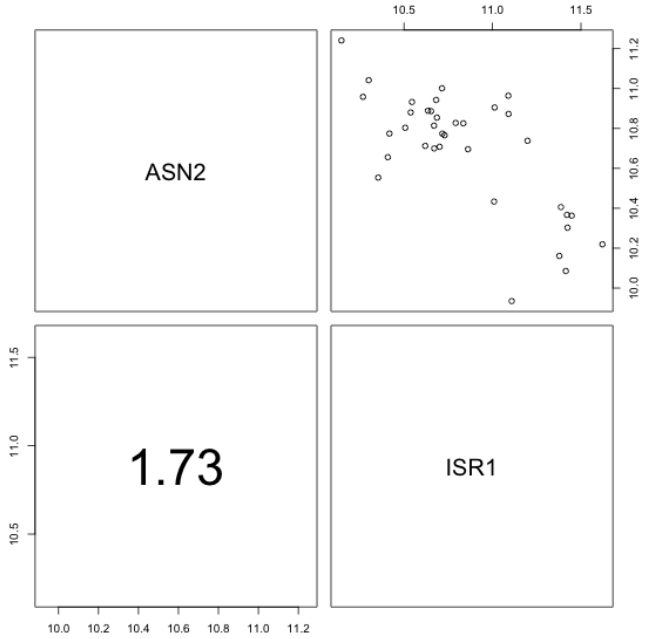

Figure 2.1: The correlation between the expression of ASN2 and ISR1 across all genotypes

The bottom panel shows the correlation distance between the two genes. Notice that the distance is close to 2, since the genes are very anti-correlated with each other. ASN2 expression is on the y-axis of the plot in the upper-left and ISR1 expression is on the x-axis. Each dot is a genotype.

Now let's look at all of the genes. Type this into your R terminal:

```
plotPairs(geneSubset)
```

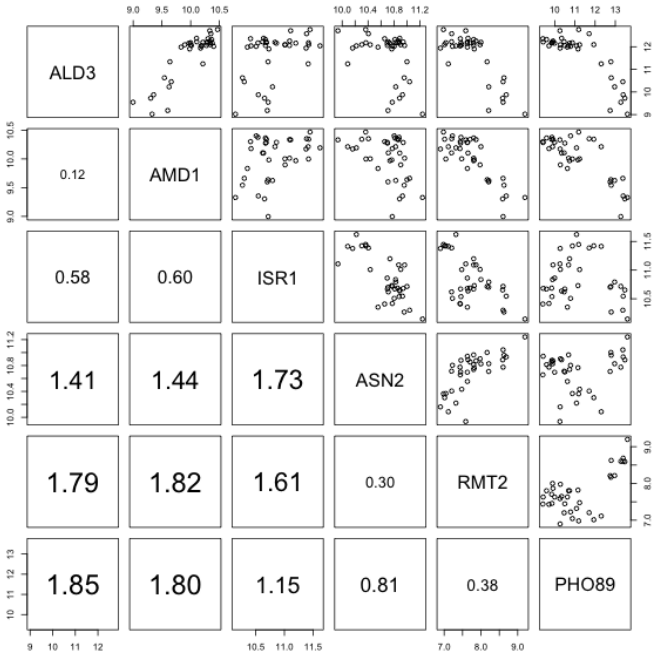

Figure 2.2: Correlation between pairs of genes across all samples

Each panel in the upper half of the plot shows a scatterplot of the expression level of two genes. The bottom half shows the correlation distance for those two genes. For example, the correlation distance between ALD3 and ASN2 is 1.40. Draw the dendrogram using the hierarchical clustering algorithm on paper. Make sure you label your y-axis so that the dendrogram is to scale. After you're done, type in the following:

```
plotHeatmap(geneSubset, genes="yes")
```

In this heatmap, yellow panels mean that a gene was more highly expressed in a particular genotype. Compare this plot to the dotplot above. It's easier to read because genes with similar expression patterns are grouped next to each other. The dendrogram also tells you that there are two obvious clusters in the data: ALD3/AMD1/ISR2 and ASN2/RMT2/PHO89.

2.3.2 Clustering genotypes

Instead of plotting the correlations between genes, we can plot the correlations between genotypes. Type the following into the R terminal:

```
1 plotPairs(sampleSubset, c("num1", "num2"), type="samples")
```

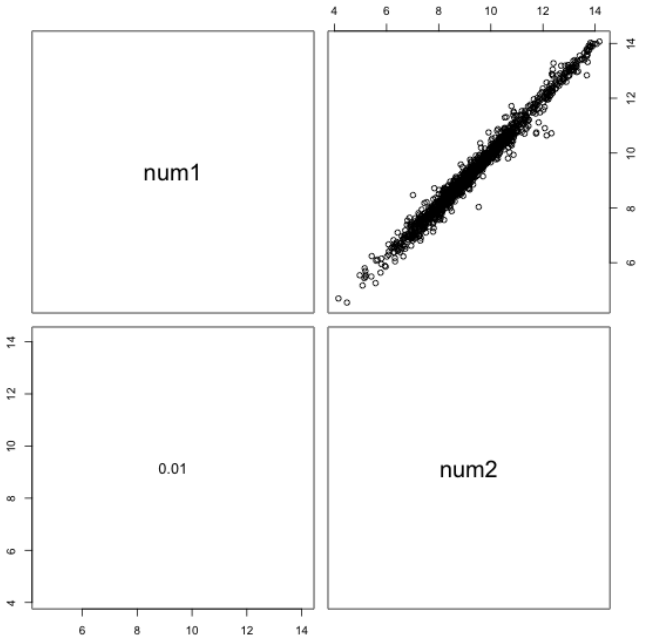

Figure 2.3: The correlation between the expression of genotypes 1 and 2 across all genes

Each of the dots is a gene, which is why there are so many more of them. You can see that genotypes 1 and 2 are pretty well correlated with each other, which is reflected in their having a small distance between each other. Similarly, we can plot all the genotypes. Type the following into your R terminal:

```
plotPairs(sampleSubset, type="samples")
```

Draw the dendrogram for the samples in the same way that you drew it for the genes. When you're done, type the following into the R terminal:

```
plotHeatmap(sampleSubset, samples="yes")
```

The dendrogram should be the same as what you got by hand. There are two obvious clusters: 1/2 and 55/31/34. Although the dendrogram tells you what samples are most similar to which, it's difficult to see the differences between the the gene expression in the different samples.

2.3.3 Clustering both genes and genotypes

To see these differences, more clearly, let's first cluster both the samples and the genes for the smallest subset of our data:

```
plotHeatmap(geneAndSampleSubset, samples="yes", genes="yes")
```

With such a small amount of data, it's easy to see the labels on the genes. Note that since we used a different amount of data, the clustering is somewhat different than what we saw above. When you're done examining the plot, plot the heatmap for the `sampleSubset` data clustered by both the genes and the samples. Does the sample dendrogram match what you saw above?

If we don't restrict ourselves to the five genotypes, we can see that the trend is preseved.

```
plotHeatmap(microarray, samples="yes", genes="yes")
```

Write down how many groups of genes and how many groups of samples you think there are in the data on your hypothesis sheet.

2.3.4 Final clustering

If we supply `plotHeatmap()` our dataset of phenotypes, we can see something interesting.

```
plotHeatmap(microarray, phenotypes, samples="yes", genes="yes")
```

When we supply the phenotypes, the function will put a black box underneath the genotypes that form complex colonies. Does this match your prediction above?

2.4 Saving your data

It's very important that after you're done with your data that you drag the script that you wrote into the drive called "Bio 201-202." Anything left on the Desktop (or anywhere else on the computer) will be deleted when the computer shuts down.

3 Day 3

3.1 Setting up your workspace

Type the following lines into R to get you ready for today:

```
setwd("~/Desktop/data_expeditions")

source("ccm_functions.R")

microarray <- read.csv("microarray.csv", row.names=1)
phenotypes<-read.csv("phenotypes.csv", header=TRUE)
```

3.2 Digging into the genetic basis of CCM

In day 2 we found that the complex colonies and the simple colonies had very different expression patterns and that within complex colonies, there were also large differences in expression pattern.

The strains that we have been looking at have also been (painstakingly) genotyped at each of four most prominent loci identified by the bulk segregant analysis. We're going to focus on three of those loci: *CYR1*, *HOT11* and *FLO11*. Type in the following line of code and examine the data as usual. You will see that each sample has an entry classifying the genotype of that strain as having either the 'complex' or the 'simple' allele of that gene. The 'complex' allele just means that complex colonies tended to have that allele.

```
alleles = read.csv("alleles.csv")
```

You can join the phenotypes and the genotype data using the `merge()` function. Examine the result and you can see that there is now a table that has both the phenotype as well as the genotype information in it for each of the samples we've been looking at.

```
phenotypes2 <- merge(alleles, phenotypes, by = "ID")
```

Now that we have the phenotypes and the genotypes in one table, we can plot them on the heatmap. For this heatmap, we'll go back to the smaller microarray dataset for the sake of speed. Type in the following code:

```
heatmap_object = plotHeatmap(microarray, phenotypes2,
            samples="yes", genes="yes")
```

Notice that this time, we're creating an object called `heatmap_object`. This object contains all the information needed to plot the heatmap, including the information about the clustering and the expression levels of all the genes and all the samples. Since this object isn't just a matrix of data, you can't see the 'head' or the 'tail' of it. You can plot a dendrogram of the samples by using the `plotDendrogram()` function:

```
plotDendrogram(heatmap_object)
```

You can display a line that would "cut" the dendrogram at a particular level of similarity like this:

```
plotDendrogram(heatmap_object, cut=0.06)
```

You can specify these 'cutpoints' in the `plotHeatmap()` function in order to highlight particular groups of genes or samples:

```
plotHeatmap(microarray, phenotypes2,
            samples="yes", genes="yes",
            cutsamples=0.06)
```

1 If you only have information about whether a strain makes simple or complex colonies, what level of gene expression similarity could you predict given two random strains?

2 If you combine the information about the *CYR1* allele with whether a strain makes complex colonies, what level of gene expression similarity could you predict given two random strains?

3 What does the heatmap tell us about the genetic basis of complex colonies? (Is there one way to get complex colonies? Two ways? More?)

3.3 Physiological consequences

3.3.1 Defining clusters of genes

Just like the sample dendrogram, you can cut the dendrogram of the genes:

```
plotDendrogram(heatmap_object, kind="Row", cut=0.5)
```

At a similarity level of 0.5, there are 5 clusters in the data.

You can display both gene and sample cutpoints on heatmaps. However, I'm just going to show you what that plot looks like because it takes too long to plot this due to technical reasons. Note: I've cropped this heatmap so you can see the labels better in your notebooks.

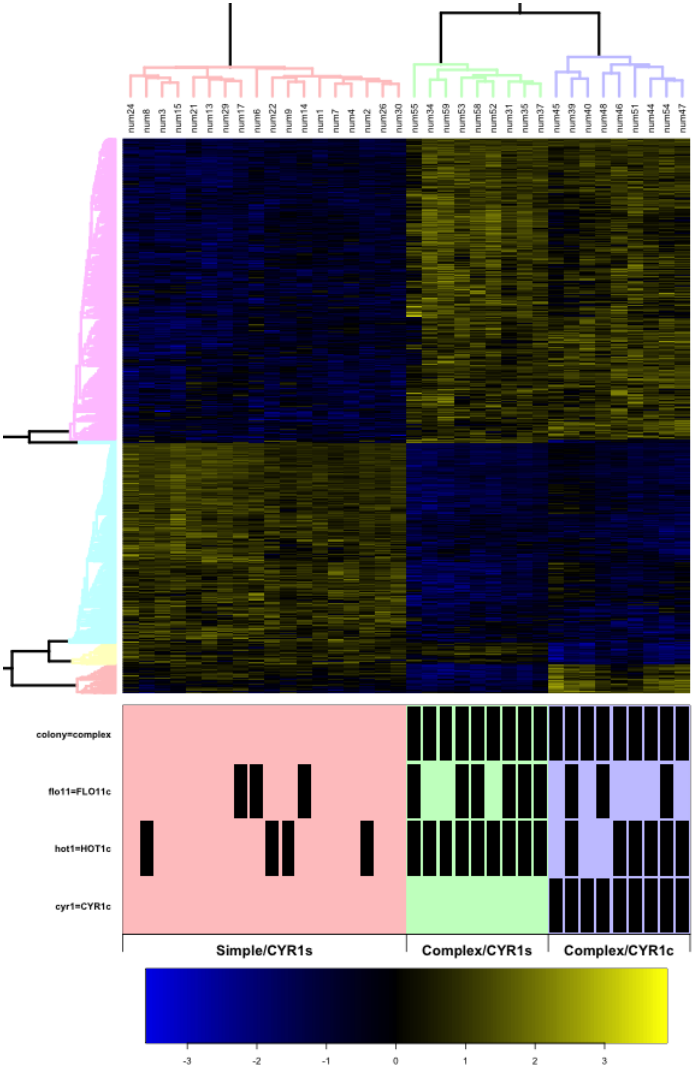

Figure 3.1: A heatmap grouping both samples and genes.

The result is five 'gene clusters' and three 'sample groups'. It is difficult to look at a line graph for the average expression of the different clusters in the different groups, but we can plot a heatmap of them. I'm not having you plot this summary heatmap because it's pretty complicated to produce and we're only going to do it once.

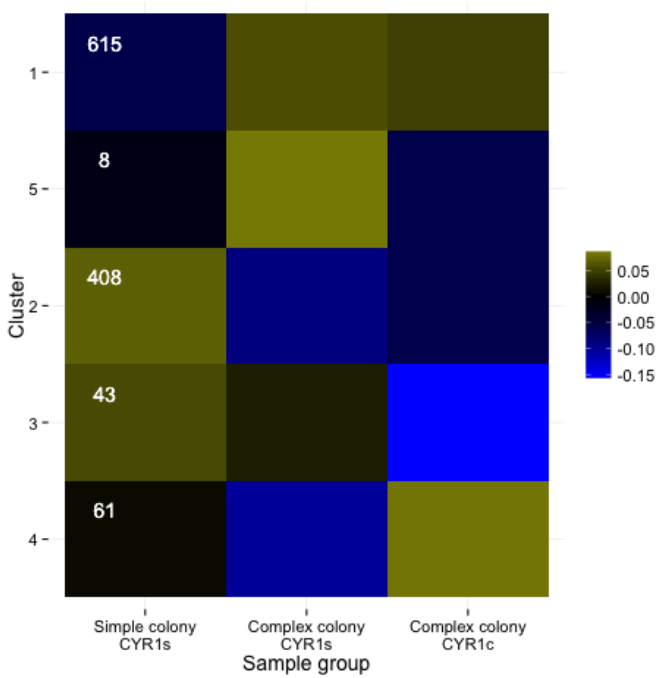

Figure 3.2: A summary of the average change in expresson of genes for a particular cluster of genes in the three different sample groups. The number of genes in a cluster is labelled in white numbers.

3.3.2 Examining the functional enrichment of the clusters

What are in these different clusters? For each of the four large clusters, I've written down some gene names.

Cluster	Name
1	SHO1
	MSB2
	PTP3
2	RSM25
	MRPL6
	MRPS35
3	ACO1
	SOL4
	GND2
4	MRT4
	RRB1
	NOP1

Look them up on the yeast genome database http://www.yeastgenome.org. You can also type them into Google if the yeast database is too slow (it may help to add "yeast" to your search to remove extraneous results). Your assignment is to prepare a short presentation that will answer:

1 What do the three genes have in common?

2 What does this suggest about the physiology and/or mechanism of complex colony morphology? Frame your conclusion as a hypothesis.

3 How could you test the prediction that you make in (2)?

For example, if all three genes function in pathways involved in making yeast fly and if the cluster of genes showed higher expression in complex colonies for both alleles of *CYR1*, you might say:

> "The cluster we examined contained genes involved in myco-aviation. Since the genes that we were looking at show higher expression in complex colonies, we hypothesize that the reason why the colonies are complex is because some yeast are flying into small piles on top of the colonies. Furthermore, myco-aviation is upregulated regardless of the *CYR1* allele. Therefore, we suspect that it is 'upstream' of whatever differences in physiology that the *CYR1* alleles cause. To test this, we propose making a movie of the colony growing under a microscope so that we can see whether flying yeast are causing the wrinkles."

Note: yeast can't fly.

3.3.3 Making a null distribution

You might be wondering how did we pick those three genes? For well-studied organisms like yeast, there exist databases that contain a bunch of 'functional terms' that summarize what is known about the function of the gene. Go to http://www.yeastgenome.org and search for ACO1. Next, click on the "Gene Ontology" tab. The "Gene Ontology" is the name of the database with the functional terms in it. If you scroll down a bit, you'll see a section called "Biological Process." This is the section that tells you what the gene "does." You can see that both "tricarboxylic acid cycle" and "mitochondrial genome maintenance" are listed as functional terms. The first term makes sense if you remember the TCA cycle from high-school. The second term comes from some research by Chen et al in 2005 that linked ACO1 to mitochondrial genome maintenance (which you can see as a link next to the term). To summarize, there is a database that associates names like "TCA cycle" with a variety of different genes.

That still doesn't answer the question of "how we decided that genes associated with mitochondrial respiration are associated with cluster 3?." The answer is that we did what's called a "functional enrichment analysis." Essentially we are looking for functional terms that are over represented in a particular set of genes. In this case we are looking for functional terms that are over represented in a cluster of genes that have a similar expression pattern across our samples. More precisely, we're going to ask whether the number of genes that are annotated with a particular functional category in a particular cluster is greater than what you'd expect by chance.

To illustrate this, imagine you are drawing M&Ms out of a bag. How many blue M&Ms would you expect to get in a draw of 10 M&Ms? How surprised should you be if you got 9/10 blue M&Ms? Could you conclude that you had gotten a bag with more blue M&Ms than usual? In order to get a feel for the factors that affect this, we are actually going to draw M&Ms out of a bag.

First you and your partner will each draw 8 M&M's out of your bag 25 times. Record these values in the appropriate column on the google document. Next draw 20 M&Ms out of your bag 25 times and record the number of blue M&Ms you got in the approriate column on the google document. The link is HERE.

3.3.4 Doing an enrichment analysis

This next section will set you up to answer the questions on your enrichment analysis worksheet. We started with 167 M&Ms and counted the different colors. For this example, we'll call the M&Ms "genes" and the colors "functional terms". First, let's save those numbers for future use:

```
# Number of genes (M&Ms) in each functional catagory
blue<-51
green<-17
red<-15
brown<-25
orange<-36
yellow<-23
```

When we sum the genes, we can see we ended up with 167:

```
# Total number of genes
totalGenes = sum(blue,green,red,brown,orange,yellow)
```

Let's say we drew 50 genes from this pool of 167 genes and got 21 blue genes. Is this likely to have happened by chance? We will use the function `rhyper()` to simulate drawing a sample genes out of the 167 genes. The function takes four arguments:

1. The first argument to `rhyper()` is the number of 'draws' you want in your null distribution. This is equivalent to the number of times you drew M&Ms out of a bag in the first excercise. Since we're humans we could only do a few, but the computer can do lots. Let's set it at 10,000 so that we get a good feeling for what the distribution looks like.

2. The second argument is the number of genes with that functonal term (aka color) in **all** the genes in the genome.

3. The third argument is the number of genes **without** the term in the genome.

4. The fourth argument is the number of genes in our cluster (the size of your draw). Let's say that there are 50 genes in our cluster. We can supply all those arguments to `rhyper()` and it will happily make our null distribution for us:

```
nullDist = rhyper( 10000, blue, totalGenes - blue, 50)
```

We can plot what it looks like using the hist function:

```
hist(nullDist)
```

If we observed 21 blue genes in our cluster, we now compute how likely we were to observe that just by chance using the `pval()` function:

```
pval(nullDist, 21)
```

You should get a p-value of around 0.01, which means that there's only a 1% chance that just drawing a random sample would give you that number of blue genes. A 1% chance is pretty unlikely, so there's some evidence that there are more blue genes in our hypothetical cluster than would be expected by chance.

Another statistic that gets used in this type of analysis is the 'fold enrichment.' You can also calculate that:

```
enrich(nullDist, 21)
```

This tells you that there is about half-again as many blue genes in your cluster as you would expect by chance. Having about 50% more genes could be interesting, but it's certainly not a startlingly high number. You might conclude from this that you observe more blue genes than you would expect by chance, that there is only a weak enrichment of blue genes in the cluster.

3.4 Exploring the co-variates of CCM

3.4.1 Description of the data

The next code will get you set up to do some open-ended exploration of the data. We'll read in a table that provides information about key measurements related to complex colony morphology physiology and genetics. You should explore what the data looks like using the functions we showed you on Day 1. The table below explains what each of the variables in the table are.

```
qPhenotypes <- read.csv("phenotypes2.csv", row.names=1)
```

Column	Description
`ID`	The ID of the strain
`cyr1`	Whether the strain has a simple or complex *CYR1* allele
`flo11`	Whether the strain has a simple or complex *FLO11* allele
`hot1`	Whether the strain has a simple or complex *HOT1* allele
`phenotype`	Whether the strain forms simple or complex colonies
`adhesion`	A measure of how well the strain sticks to plastic. Higher is stickier.
`CM`	The average colony morphology score of 3 replicates
`cyr1.expr`	qPCR data for the expression of the *CYR1* gene
`flo11.expr`	qPCR data for the expression of the *FLO11* gene
`cAMP`	The concentration of cAMP in the cells

3.4.2 Functions to explore the data with

To help you explore the data, you can use the `colonyDotPlot` function. This function at its most basic will let you plot a dot for the complex and simple colonies for whatever measure you want. For example, you can look at how sticky complex and simple colonies are:

```
colonyDotPlot(qPhenotypes, "adhesion")
```

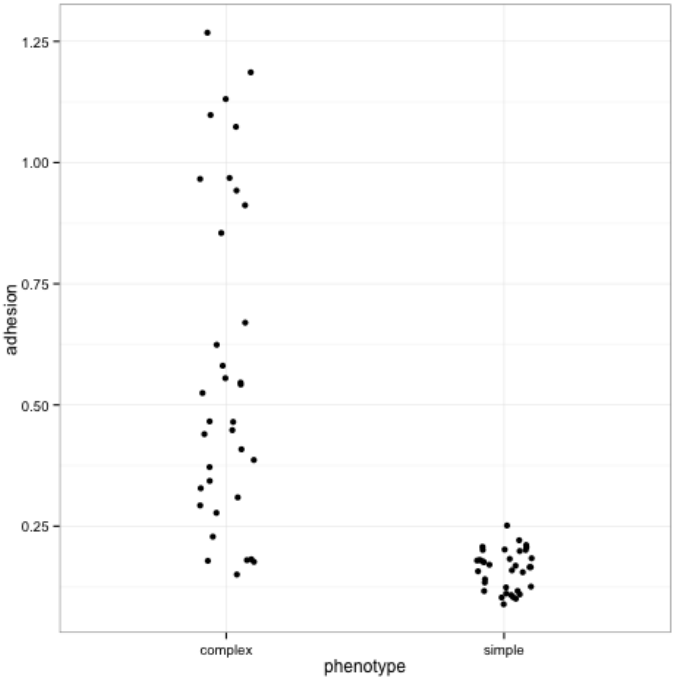

Figure 3.3: The adhesion of each sample for both simple and complex colonies.

Note that the points are jittered in the x direction to show you if they overlap. This plot tells us that complex colonies are stickier than simple ones. What we'd like to know, however, is whether there is more than one way to be a sticky colony. For example, there appears to be an extra group of points with very high adhesion (above 0.75). What could explain this? To find out, let's color the samples according to whether they have a complex or a simple *FLO11* allele:

```
colonyDotPlot(qPhenotypes, "adhesion", "flo11")
```

Sometimes, it's useful to plot the density of points along the Y axis. To do this, you can use the `doViolins` argument to add what's called a 'violin plot' to the points.

```
colonyDotPlot(qPhenotypes, "adhesion", "flo11", doViolins=TRUE)
```

A more traditional way to do the same thing is to use a boxplot:

```
colonyDotPlot(qPhenotypes, "adhesion", "flo11", doBoxplot=TRUE)
```

The boxplot plots make it clearer that while there isn't a difference between the adhesion within simple colonies between the simple and complex alleles of *FLO11*, within the complex colonies strains with the simple *FLO11* allele were less likely to be sticky. The plot actually looks cleaner if you suppress the points, which you can do with the `doPoints` argument:

```
colonyDotPlot(qPhenotypes, "adhesion", "flo11",
              doBoxplot=TRUE, doPoints=FALSE)
```

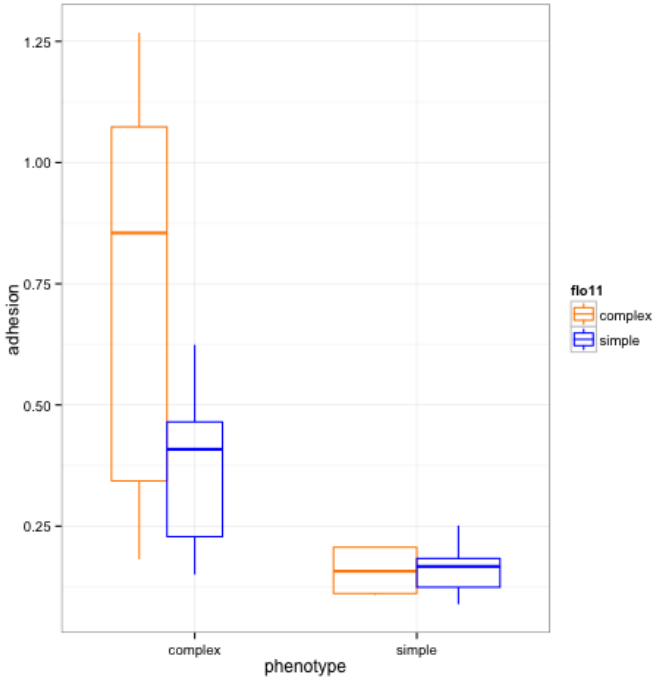

Figure 3.4: The adhesion of each strain for both simple and complex colonies broken up by what allele of *FLO11* the strain carries.

If you like, you can also plot the Y axis on a log10 scale.

```
colonyDotPlot(qPhenotypes, "adhesion", "flo11",
              doBoxplot=TRUE, doPoints=FALSE, log=TRUE)
```

You can also plot an XY plot in a similar way using the `colonyXYPlot()` function:

```
colonyXYPlot(qPhenotypes, "flo11.expr", "adhesion", "flo11")
```

You can also plot log scales with this function, but you have to use different arguments:

```
colonyXYPlot(qPhenotypes, "flo11.expr", "adhesion", "cyr1",
             log.x=TRUE, log.y=TRUE)
```

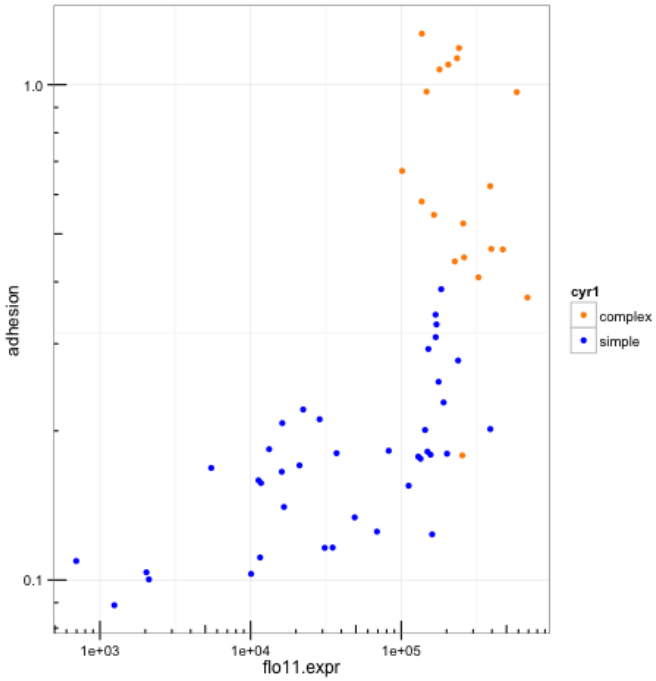

Figure 3.5: The relationship between adhesion and *FLO11* gene expression for strains that carry either a simple or complex *CYR1* allele.

3.5 An example exploration

For your final project in this class, we're going to ask you to answer a question about the relationship of allelic variation, gene expression, complex colony morphology, and/or a physiological measure associated with complex colonies. First, let's work through an example question:

> Strains that form complex colonies tend to stick to plastic more than strains that form simple colonies. An allele of *FLO11* is associated with complex colonies. How does *FLO11* expression and/or allelic variation relate to plastic adhesion?

The simplest explanation would be that since the *FLO11c* allele is enriched in complex colonies and since complex colonies are more adhesive that the *FLO11c* allele causes increased adhesion to plastic. To test this, let's look at the relationship between *FLO11* gene expression, adhesion, and the allelic variation of *FLO11* and *CYR1*.

3.5.1 Plots

I made a variety of plots when I was researching this question. Some are useful, some not so much. I suggest making lots of them and saving them with informative titles.

```
1 colonyDotPlot(qPhenotypes, "adhesion", "cyr1")
```

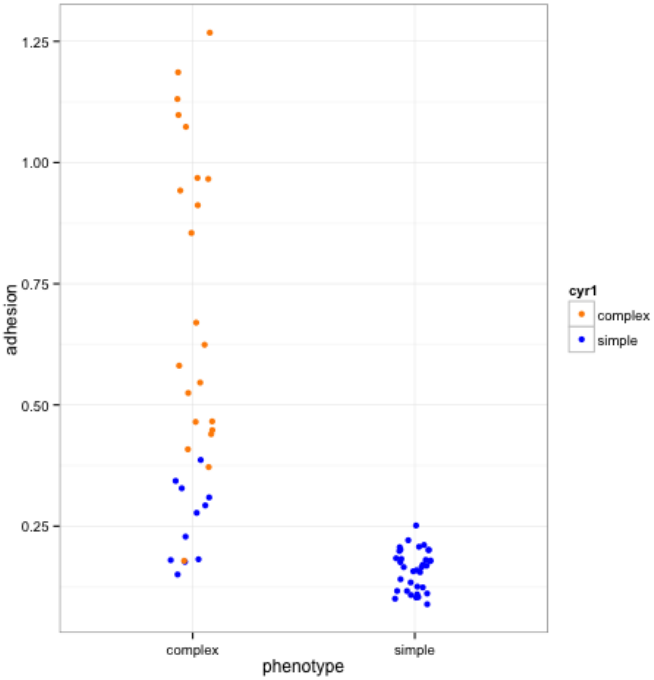

Figure 3.6: Adhesion in simple and complex colonies for strains that carry either a simple or complex *CYR1* allele.

```
colonyDotPlot(qPhenotypes, "flo11.expr", "flo11")
```

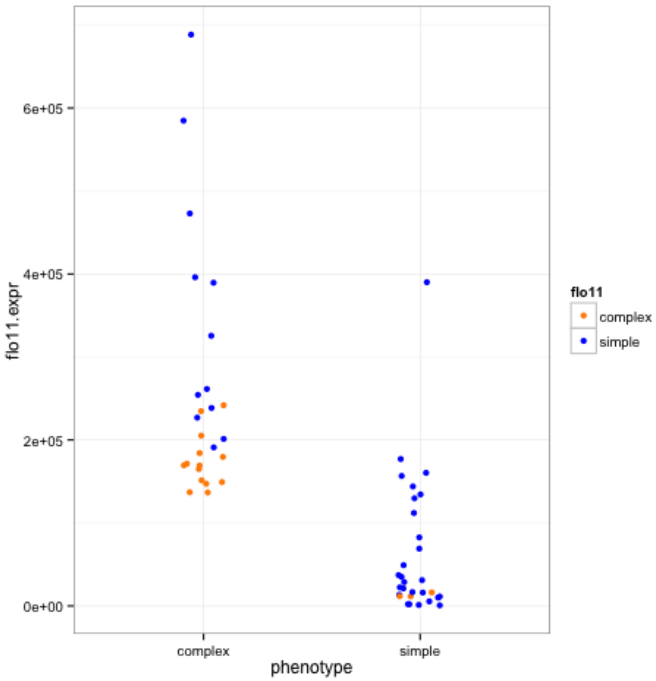

Figure 3.7: *FLO11* expression in simple and complex colonies for strains that carry either a simple or complex *FLO11* allele.

```
colonyXYPlot(qPhenotypes, "flo11.expr", "adhesion", "flo11",
             log.x=TRUE, log.y=TRUE)
```

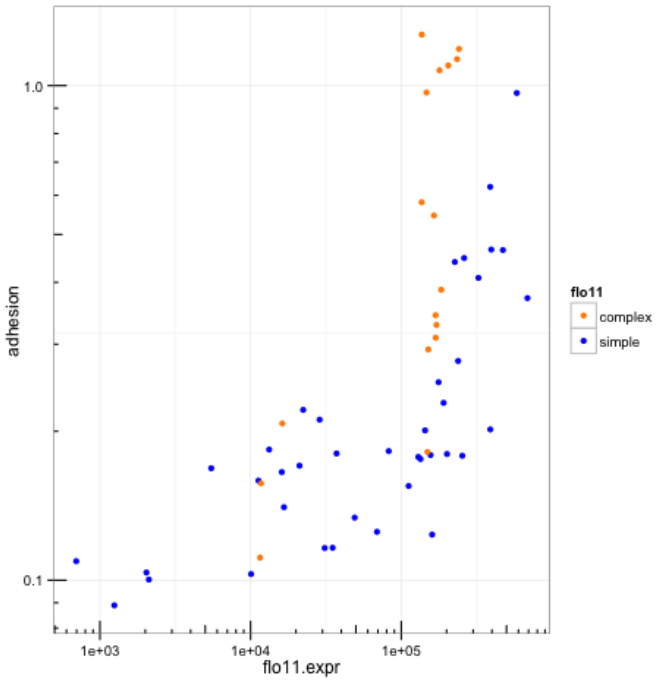

Figure 3.8: The relationship between adhesion and *FLO11* gene expression for strains that carry either a simple or complex *FLO11* allele.

3.6 The final exploration

cAMP is produced by the gene *CYR1*. The allele *CYR1c* is associated with complex colonies, and complex colonies have higher concentrations of cAMP. Is the *CYR1c* allele responsible for the increased concentration of cAMP between simple and complex colonies? What about within complex colonies? Is the expression of *CYR1* responsible for the increased concentration? How does this fit in with what we learned about the *CYR1c* cluster in the gene expression data? What does this tell us about the interaction between the lower level properties that may be needed to produce complex colonies?

4 Acknowledgements and gratitude

- Thanks to Josh Granek for collecting the gene expression data!
- Thanks to Debra Murray for collecting the high-resolution mapping data to determine the genotypes of the strains!
- Thanks to Paul Magwene for the LaTeX template and for some of the code introducting R.
- Thanks to Daniele Armaleo and Katherine Picard for working with us to help make this data expedition happen
- Thanks to the Data Expeditions program for funding and resources.