Introduction to Bioconductor and R refresher

Mark Dunning

Last modified: May 20, 2015

Contents

1 Introduction

The purpose of this practical is to introduce some of the key object-types that are used in the Bioconductor project, and to refresh your memory on R basics. We will use a published dataset breast cancer study and go through the steps involved in determining whether a particular gene is associated with clinical variables of interest.

Please try and avoid copy-and-pasting the commands. You will learn R quicker if you go through the painful experience of making 'typos' and getting error messages. You can reduce the number of typos that you make by using the *tab-complete* facility in RStudio so you don't have to write the full name of a function or object.

1.1 The working directory

Like other software programs, R has a concept of a *working directory*. This is the location on your hard drive where R will save files that you create (e.g. graphics, scripts, tables), and look for input files.

There are two main ways of changing your working directory in RStudio. The first is to navigate the menu system. `Session` → `Set working directory` → `Choose Directory`.

Set your directory now to `nki`.

1.2 Reading data into R

We are going to read some expression data into R. In order to do this, we first need to locate the file on our hard drive and make a note of the path to the file.

A useful trick here is to use the `file.choose` function. This will show a dialogue box allowing you to select the location of the file that you want to read, and save the location as a *variable* (Recall that <- is the symbol for creating a variable in R).

Exercise: Use the `file.choose` function to locate the file `NKI295.exprs.txt` and save the location as a variable.

```
myfile <- file.choose()
myfile
```

The `read.` family of functions can be used to read tabular data into R. For instance, `read.delim` and `read.csv` are used to read *tab* and *comma* separated files respectively and are more general versions of the function `read.table`. Every aspect of reading a file can be customised; from the character used to separate the columns, whether to use column headings or not, or which character is used to specify a missing value. All these options (and many more) are detailed in the help page for `read.table`. To keep our example simple, we use the fact that our example file is tab-delimited. To read the file `NKI295.exprs.txt` into R, we just need to use the file location that we stored in the previous exercise as an argument to `read.delim`

Exercise: Read the breast cancer expression values into R. What type of object is created? How can you find out the dimensions of this object? How would you read the data differently if the file was comma-separated?

Hint: We can use the `file.exists` function to check that the path that we have specified is valid. If the function returns `FALSE` then we will not be able to use this path as an argument to a function that reads a table.

```
file.exists(myfile)
expression <- read.delim(myfile)
dim(expression)
head(expression)
```

The object we have created is a *data frame* representation of the data in the file with 24481 rows and 296 columns. We can now try the square bracket notation "[" to subset the rows and columns of the object.

Exercise: Print out i) the first five rows and first 10 columns ii) the first 10 rows and all columns

```
expression[1:5,1:10]
expression[1:10,]
```

Note that these statements just print a subset of the object to the screen, and the original object is unchanged. So if we query the dimensions of `expression`, we should get the same answer as before.

```
dim(expression)
```

Exercise: Create a new object that contains the first 10 rows

```
subset <- expression[1:10,]
subset
```

You will see that the first column of the data frame gives the probe IDs that were used on the microarray. If we proceed to do any analysis or computation on the object, then including this column could be problematic as it could be treated as a numeric quantity. Using the `[ ]` syntax, we can remove rows or column by preceeding the column index with a minus sign `-`. We can assign names to the *rows* of the data frame should we wish to keep track of what genes are being referred to in each row.

Exercise: Create a new data frame that i) contains just expression values and ii) has probe IDs as the rownames.

```
rownames(expression) <- expression[,1]
evals <- expression[,-1]
```

Each row in this new matrix contains the expression values for a particular (arbitrary) gene for each patient in the study. It is worth noting that we could have used the `row.names` argument to `read.delim` to pick a particular column in the original text file to be used as row names. This avoids having to rename the rownames after having read the file and creating a new copy of the data frame without the names.

```
evals <- read.delim(myfile,row.names=1,as.is=TRUE)
dim(evals)
head(evals)
evals[1:10,]
evals[1:5,1:10]
```

So far we have created subsets of our data matrix using consecutive rows via the `:` shortcut. We can in fact use a number of different methods to generate the rows indices.

Exercise: Make sure that you understand the following subset operations

```
expression[c(1,3,5,7),]
expression[seq(1, 10000,by=1000),]
expression[sample(1:24481,10),]
```

Hint: You can use the ? operator to bring up the help file for a particular R function

1.3 Dealing with numerical data

We will now explore some of the many ways that we can interrogate our gene expression values. Firstly, we can convert the data frame into a matrix, which will make things easier in the following code.

Exercise: Convert the data frame to a matrix and print the gene expression values for the gene in the first row

```
evals <- as.matrix(evals)
evals[1,]
```

As it was developed primarily as a statistical language, R includes many operations that can be performed on numerical data. It can produce numerical summaries of data that you should be familiar with (e.g. mean, median, standard deviation etc) as well as fit complex models to data. Initially we will just summarise our gene expression values using basic functions.

Exercise: Use the `mean` and `summary` functions on the expression values for the first gene

```
mean(evals[1,])
summary(evals[1,])
```

The functions we have just used will return the mean and overall distribution of a particular gene. We can also make a histogram of the values as follows.

```
hist(evals[1,])
```

1.4 A digression on plotting

If we try and make a scatter plot of the expression level of a particular gene across the samples, the result is quite ugly.

```
plot(evals[1,])
```

The `plot` function can be customised in many ways. R has many in-built colours, which can be selected from the `colors` function. This online pdf can be used as a guide to choosing colours

http://www.stat.columbia.edu/ tzheng/files/Rcolor.pdf

Exercise: Use the `colors` function to see what colours are available. Then plot the expression values for the first gene in the colour `steeblue`.

```
colors()

plot(evals[1,],col="steelblue")
```

Here we specify the same colour for each point. In fact, we could specify a unique color for each point by creating a vector the same length as the number of points to be plotted. In general, the colour vector will be *recycled*.

Exercise: Use the `rainbow` function to create a vector of length 295, and use this as an argument to change the colours in the scatter plot

```
rainbow(n=ncol(evals))
plot(evals[1,],col=rainbow(n=ncol(evals)))
```

Another modification is to use a different '**p**lotting **ch**aracter'.In total 25 different plotting characters are possible and each is assigned a unique numeric code. e.g. a popular choice is the number 16 , which defines filled circles. These are described in the documentation for the `points` function (`?points`)

Exercise: Repeat the previous plot, but use filled circles to plot the points

```
rainbow(n=ncol(evals))
plot(evals[1,],col=rainbow(n=ncol(evals)),pch=16)
```

As with specifying colours, we are not restricted to have the same plotting character for all points.

```
rainbow(n=ncol(evals))
plot(evals[1,],col=rainbow(n=ncol(evals)),pch=1:16)
```

We can also add more-meaningful labels to the axes and a title

```
plot(evals[1,],col="steelblue",pch=16,xlab="Patient",
     ylab="Expression value",main="Expression of gene X")
```

After a plot has been created, we can add extra lines, points and text to it. For example, we might want to highlight particular observations of interest. The `which` function can be used in conjunction with a logical test (i.e. one that will give `TRUE` or `FALSE`) to identify the indices of observations that satisfy particular criteria.

Exercise: Use a `which` statement to find out the indices of samples that have expression greater than 4 for the first gene.

```
which(evals[1,] > 4)
```

We can use the following lines of code to highlight samples with expression higher than 4, or lower than -4. `points` and `abline` are used to add extra details to the initial plot.

Exercise: Make sure that you can follow the lines of code

```
values <- evals[1,]
plot(values,col="steelblue",pch=16)
outl <- which(abs(values) > 4)
abline(h=c(-4,4))
points(outl,values[outl],col="red",pch=16)
```

A more robust version of the code would be to define the limit as a variable.

```
limit <- 4
values <- evals[1,]
plot(values,col="steelblue",pch=16)
outl <- which(abs(values) > limit)
abline(h=c(-limit,limit))
points(outl,values[outl],col="red",pch=16)
```

1.5 Importing the clincal data

We are dealing with a real-life dataset involving breast cancer patients, and patients in the study have various clinical or phenotypic characteristics that we may wish to incorporate into the analysis.

Exercise: Now read the clinical annotations for the dataset, which are found in the file `NKI295.pdata.txt` What are the dimensions of the resulting object and how do they relate to the expression values?

```
clinical <- read.delim("NKI295.pdata.txt")
dim(clinical)
head(clinical)
```

You should find that the number of rows in the clinical data is the same as the number of columns in the expression data; 295. Moreover, they ordered in the same manner. This means we can easily associate the expression values for a particular sample with it's clinical parameters, and vice-versa. As we have created a data frame to store the clinical data, we can have both numerical and categorical variables in the same object.

In order to commence an analysis of the data, we also need to retrieve annotation for the rows in the expression matrix. i.e. the genes measured on the array.

Exercise: Read the feature annotation for the dataset, which can be found in the file `NKI295.fdata.txt`. Again, what are the dimensions are how to they correspond to the expression values?

```
features <- read.delim("NKI295.fdata.txt")
head(features)
```

You should find that the number of rows in the feature annotation is the same as the number of rows in the expression matrix.

2 Combining expression values, clinical data and feature information

2.1 Creating an ExpressionSet

The `ExpressionSet` is the standard Bioconductor way of storing microarray data and is ubiquitous throughout the software that is available through the Bioconductor project. The re-use of object-types

in this manner is encouraged and helps the user to begin to use new software.

Exercise: Load the Biobase package and create an ExpressionSet representation of the data using the `readExpressionSet` function. If you didn't have Biobase installed, how would you install it? How can you find out what `readExpressionSet` does and the arguments it accepts?

```
library(Biobase)
eset <- readExpressionSet("NKI295.exprs.txt", "NKI295.pdata.txt")
?readExpressionSet
```

When you type the name of a variable in R, it will print the contents of that variable to the screen. Unfortunately, this can be quite painful in the case of large data frames, like those that we were dealing with in the previous section. Moreover, more complicated classes like lists and environments can have a structure that is difficult for the user to interpret. To address this, Bioconductor have re-written the printing behaviour for the `ExpressionSet` class so that only a portion of the data is written to the screen. A summarised view of the object is displayed, showing how many features and samples are present, along with a snapshot of the metadata.

Exercise: Show a summary of the eset object and verify that it has the correct number of samples and features.

```
eset
```

2.2 Accessing data from the ExpressionSet

The `ExpressionSet` is a complex object type. Fortunately, you do not have to fully-understand how it is implemented in order to access and manipulate the data stored in such an object. Instead the authors provide a set of 'convenience functions', whose purpose is to access the data in the object in a painless manner.

Exercise: Use the `exprs` and `pData` function to extract the expression values and sample information from the example dataset.

```
dim(exprs(eset))
head(exprs(eset))
dim(pData(eset))
head(pData(eset))
```

Hint: pData is shorthand for phenotypic data

2.3 Subset operations

The `ExpressionSet` object has been cleverly-designed to make data manipulation consistent with other basic R object types. For example, creating a subset of an ExpressionsSet will subset the expression matrix, sample information and feature annotation (if available) simultaneously in an appropriate manner. The user does not need to know how the object is represented 'under-the-hood'. In effect, we can

treat the ExpressionSet as if it is a standard R data frame.

Exercise: Subset i) the first 1000 rows of the ExpressionSet and ii) first 10 columns. Notice how the output changes accordingly.

```
eset[1:1000,]
eset[,1:10]
```

As in the previous section, if we do not use an assignment operation the dimensions of the original object are not altered and a new object is not created.

Exercise: Create a new ExpressionSet that comprises 1000 random genes from the original object, and the first 50 samples. Use the `exprs` and `pData` function to check that you get the dimensions that you would expect

```
random.rows <- sample(1:nrow(eset),1000)
eset.sub <- eset[random.rows,1:50]

dim(exprs(eset.sub))

head(exprs(eset.sub))
pData(eset.sub)
```

2.4 Subsetting according to sample annotation

Exercise: Use the `pData` function to access the sample information for the ExpressionSet object. Check that the row names of the sample information correspond to the columns in the expression matrix.

```
sampleMat <- pData(eset)
head(sampleMat)
rownames(sampleMat)
colnames(exprs(eset))
all(rownames(sampleMat) == colnames(exprs(eset)))
```

The process of identifying clinically-relevant breast cancer subtypes has been improved dramatically using high-throughput genomics technologies. However, a good indicator of the severity of a breast cancer is still its 'Estrogen Receptor' (ER) status: with ER negative samples having much worse prognosis. Thus, the ER status of a patient is often critical in the analysis. In our case, the ER status is recorded in the `ER` column.

```
table(sampleMat$ER)
```

Hint: We are fortunate here to be using a well-curated dataset and there are just two values present in the ER status column. Other datasets might not be as tidy and may require curation. For example, R would treat 'Negative', 'negative', 'NEGATIVE' as different values when performing calculations.

Exercise: Create an ExpressionSet that contains just ER negative patients.

```
sampleMat$ER == "Negative"
erNegSet <- eset[,sampleMat$ER == "Negative"]
```

2.5 Subsetting according to gene annotation

If we want to locate particular genes in the data, a similar procedure can be used. We can first verify that the rows of the expression matrix are in the same order as our features matrix.

```
all(features[,1] == rownames(exprs(eset)))
```

Consequently, we can look up the rows in the annotation matrix that satisfy certain critera. The resulting indices can then be used to extract the expression values for those same genes. A common use-case is to find the rows corresponding to a given gene name. The look-up can be performed by the `match`, `grep`, `==`, or `%in%` functions as required.

Exercise: Use the features matrix to find the probe ID for the *ESR1* gene symbol. Extract the row from the expression matrix that corresponds to this probe.

```
head(features)
match("ESR1", features[,2])
features[match("ESR1", features[,2]),]

myrow <- match("ESR1", features[,2])
mygene <- exprs(eset)[myrow,]
mygene
plot(density(mygene))
```

Hint: However, note that the default behaviour of `match` is to return the first match only. For situations where you expect multiple matches use can use `grep`.

3 Basic data analysis

We will now bring together the expression values, sample annotation and gene annotation to perform a basic analysis. From the literature we know to expect a huge difference in ESR1 expression between ER negative and positive samples. So as a sanity check, it would be useful to visualise the gene expression levels of ESR1 between the two groups of interest. A boxplot is ideal for this task. To construct a boxplot, we either need to supply a data frame to the `boxplot` function, or specify a response and explanatory variable. In our case, we want the response to be the gene expression level of a particular gene and the explanatory variable to be the ER status of each sample. The 'tilde' notation can be used to create a formula in the form `y ~ x` as input to the `boxplot` function.

Exercise: Make a boxplot of ESR1 expression in ER positive and ER negative groups. Do you notice a trend? Use a t-test to test if the difference is significant.

```
head(pData(eset))

ERStatus <- pData(eset)$ER

boxplot(mygene ~ ERStatus)

t.test(mygene ~ ERStatus)
```

The genefilter package contains a convenient way to calculate t-statistics for many genes simultaneously. We are going to use these statistics to find out what genes show the biggest difference between ER positive and negative samples.

Exercise: Use the `rowttests` function to calculate t-statistics for each gene, and then order the output.

```
library(genefilter)
tstats <- rowttests(exprs(eset), as.factor(ERStatus))
head(tstats)

ordt <- order(abs(tstats),decreasing=TRUE)
```

Exercise: Find out the indices of top 10 genes. What genes do they correspond to?

```
topGenes <- ordt[1:10]
features[topGenes,]
```

Exercise: Use a `for` loop to make boxplots of the top 10 genes and save to a pdf file. Put the Symbol of each gene in the plot title.

```
pdf("TopGenes.pdf")

for(i in 1:10){
  rowind <- ordt[i]
  geneName <- features[rowind,2]
  boxplot(exprs(eset)[rowind,]~ERStatus,main=geneName)
}
dev.off()
```