1

R Bioconductor introduction

Robert Gentleman, Florian Hahne, Seth Falcon, Martin Morgan and Paul Murrell

Abstract

In this lab we will cover some basic uses of R and also begin working with some of the Bioconductor data sets and tools. Topics covered include basic R programming, R graphics, and working with environments as hash tables Gentleman et al. (2005). We introduce the primary data structures used to hold data from many different high-throughput experiments and demonstrate different manipulations of them. In addition, we will explore some visualization techniques for gene expression data to get a feeling for R's extended graphical capabilities.

1.1 Introduction

In this tutorial you will learn some of the fundamental concepts in order to use R for the analyis of genomic data. You will see how to manipulate different data sets and use R and Bioconductor to explore and model your data. You should be familiar with the basics of R programming like its fundamental data structures, assignments, indexing and also have a basic understanding of R's very own object system.

1.2 Working with packages

The basic design of R and Bioconductor is modular, and a lot of the functionality is provided by additional pieces of software called packages. There are now hundreds of packages available for R and over 150 for Bioconductor. Before we begin working with real biological data, it is important that you learn how to find, download and install different packages. There are a number of different methods that can be used and over time we expect

them to become more standard. R packages are stored in libraries, you can have multiple libraries on your computer, although most people have only one on their personal machine. Packages must be downloaded and installed. You need to do this only once. After that, each time you want to use the package you must load it. You do this using either the `library` function or the function `require`. Downloading packages can be done using the menu on a distribution of R that has a GUI (this is either Windows or OS X). On these platforms you simply select the packages you want and they are downloaded and installed, but they are **not** loaded into your R session, you must do that. By default this mechanism will download the appropriate *binary* packages. You can use the function `install.packages` to download a specified list of packages. One of the arguments to `install.packages` controls whether package dependencies should also be downloaded and for Bioconductor packages we strongly recommend setting this to `TRUE`. To make the installation of Bioconductor packages as easy as possible, we provide a web-accessible script called `biocLite` that you can use to install any Bioconductor package along with its dependencies. You can also use `biocLite` to install packages hosted on CRAN. Here is a sample session illustrating how to use `biocLite` to install the **graph** and **xtable** packages.

```
> source("http://bioconductor.org/biocLite.R")
> biocLite(c("graph", "xtable"))
```

Exercise 1
What is the output of function `sessionInfo`*?*

Solutions:

```
> sessionInfo()
```

```
R version 2.5.0 RC (2007-04-22 r41275)
i386-apple-darwin8.9.1

locale:
C

attached base packages:
[1] "tools"     "stats"     "graphics"  "grDevices" "utils"     "datasets"
[7] "methods"   "base"

other attached packages:
   weaver codetools    digest
  "1.2.0"   "0.1-1"   "0.3.0"
```

1.3 Some Basic R

Before we begin, let's make sure you are familiar with the basic data structures in R and the fundamental operations that are necessary for both application

of existing software and for writing you own short scripts. If you don't have problems answering the following five questions you are ready to proceed with this chapter and learn about the great stuff you can do with your genomic data. If not, it might be a good idea to go back to the excellent 'Introduction to R' which you can find on the R foundation homepage at `http://cran.r-project.org/manuals/R-intro.html`.

Exercise 2

a *The simplest data structure in R is a vector. Can you create the following vectors?*

- `x` *with elements* `0.1`, `1.1`, `2.5` *and* `10`
- *integer vector* `y` *with elements 1 to 100*
- *a logical vector* `z` *indicating the elements of* `y` *that are below 10*
- *a named charactor vector* `pets` *with elements* `dog`, `cat` *and* `bird`. *You can chose whatever names you like for you new virtual pets.*

b *What happens to vectors in arithmetic operations? What is the result of the following expression?*

```
> 2 * x + c(1, 2)
```

c *Index vectors can be used to select subsets of elements of a vector. What are the three different types of index vectors? How do we index a matrix or an array?*

d *How can we select elements of a list? How do we create a list?*

e *What is the difference between a data.frame and a matrix?*

Solutions:

a
```
> x <- c(0.1, 1.1, 2.5, 10)
> y <- 1:100
> z <- y < 10
> pets <- c(Rex = "dog", Garfield = "cat",
+     Tweety = "bird")
```

b Arithmetic expressions in R are vectorized. The operations are performed element by element. If two vectors of unequal lenght are used in the same expression, R tries to recycle the shorter of the two vectors.

```
> 2 * x + c(1, 2)
```

```
[1]  1.2  4.2  6.0 22.0
```

c Index vectors can be of type *logical*, *integer* and *character* (for the special case of named vectors).

```
> y[z]
```

```
[1] 1 2 3 4 5 6 7 8 9
```

```
> y[1:4]
```

```
[1] 1 2 3 4
```

```
> y[-(1:95)]
```

```
[1]  96  97  98  99 100
```

```
> pets["Garfield"]
```

```
Garfield
   "cat"
```

Matrices and arrays can be indexed similar to vectors. Each dimension is separated by a comma in the square brackets.

```
> m <- matrix(1:12, ncol = 4)
> m[1, 3]
```

```
[1] 7
```

d List items are selected using the `$` operator or the `[` operator. The latter accepts all three types of index vectors, the former can be used with named lists only. Note that `[` returns a list even if only one element is selected. You can use the`[[` operator to get to the content of a single list element. List are created using the `list` function.

```
> l <- list(name = "Paul", sex = factor("male"),
+     age = 35)
> l$name
```

```
[1] "Paul"
```

```
> l[[3]]
```

```
[1] 35
```

e A *matrix* consists of elements of equal type. In a *data.frame*, each column may contain different types of elements (essentially it is a list with dimension attributes).

1.4 Structures for genomic data

The data from many high-throughput genomic experiments, such as microarray experiments, can be summarized by a matrix of expression values. The matrix has F rows and S columns, where F is the number of features on the chip and S is the number of samples. In addition, one will have a data table that provides information on the samples (e.g., sex, age, and treatment status). The information describing the samples, or *phenotypes*, can be represented as an S by V table, where V is the number of covariates. In R, we use a *data.frame* to hold this phenotypic data, and we do acknowledge that it may contain information other than phenotype under some circumstances. Note that the columns of the expression matrix must align with the rows of the phenoData table. The *ExpressionSet* class provides a container for the expression matrix and phenoData and keeps the two properly aligned, e.g after subsetting operations.

In Bioconductor we have taken the approach that these data should be stored in a single data structure that can be used to easily obtain subsets of the samples, or of the features (often genes).

We begin by accessing a small data set that is provided with the **Biobase** package. First load the **Biobase** package and then the data set `sample.ExpressionSet`. We assign it to the object `exSet` in order to get rid of the bulky name. At this time we also load the `hgu95av2` package that contains metadata for the experiment.

```
> library("Biobase")
> library("hgu95av2")

> data(sample.ExpressionSet)
> exSet <- sample.ExpressionSet
> exSet

ExpressionSet (storageMode: lockedEnvironment)
assayData: 500 features, 26 samples
  element names: exprs, se.exprs
phenoData
  sampleNames: A, B, ..., Z (26 total)
  varLabels and varMetadata:
    sex: Female/Male
    type: Case/Control
    score: Testing Score
featureData
  featureNames: AFFX-MurIL2_at, AFFX-MurIL10_at, ..., 31739_at (500 total)
  varLabels and varMetadata: none
experimentData: use 'experimentData(object)'
Annotation [1] "hgu95av2"
```

The `exSet` object is an instance of the S4 *ExpressionSet* class. You can get help (a description of the class) by using the `?` operator; Try typing `class ? ExpressionSet`. Subsetting of `exSet` works similar to subsetting of *data.frames*.

```
> class(exSet)
```

```
[1] "ExpressionSet"
attr(,"package")
[1] "Biobase"
```

```
> slotNames(exSet)
```

```
[1] "assayData"
[2] "phenoData"
[3] "featureData"
[4] "experimentData"
[5] "annotation"
[6] ".__classVersion__"
```

```
> exSet$sex
```

```
 [1] Female Male   Male   Male   Female
 [6] Male   Male   Male   Female Male
[11] Male   Female Male   Male   Female
[16] Female Female Male   Male   Female
[21] Male   Female Male   Male   Female
[26] Female
Levels: Female Male
```

```
> exSet[1, ]
```

```
ExpressionSet (storageMode: lockedEnvironment)
assayData: 1 features, 26 samples
  element names: exprs, se.exprs
phenoData
  sampleNames: A, B, ..., Z (26 total)
  varLabels and varMetadata:
    sex: Female/Male
    type: Case/Control
    score: Testing Score
featureData
  rowNames: AFFX-MurIL2_at
  varLabels and varMetadata: none
experimentData: use 'experimentData(object)'
Annotation [1] "hgu95av2"
```

```
> exSet[, 1]
```

```
ExpressionSet (storageMode: lockedEnvironment)
assayData: 500 features, 1 samples
  element names: exprs, se.exprs
phenoData
  rowNames: A
  varLabels and varMetadata:
    sex: Female/Male
    type: Case/Control
    score: Testing Score
featureData
  featureNames: AFFX-MurIL2_at, AFFX-MurIL10_at, ..., 31739_at (500 total)
```

```
  varLabels and varMetadata: none
experimentData: use 'experimentData(object)'
Annotation [1] "hgu95av2"
```

You can extract the values in the slots using the `@` operator, however, if accessor functions are available, it is better to use them instead of the `@` operator. The names of the slots can be obtained using `slotNames`, as shown above. Extract the values for some of the named slots.

Exercise 3
This exercise shows you how to take subsets of ExpressionSets. This idea is important when writing your own code and often when analyzing data you will want to work only on a subset of a larger data set.

a *What happens when we subset* `exSet`*? What kind of an object do we get?*

b *What happened to the phenotypic data? What happened to the expression data?*

c *Imagine that your boss has asked you to work specifically on the female subjects. To do that the first step is to subset* `exSet` *by selecting all female samples.*

Solutions:

a Subsetting obects of class `ExpressionSet` will again result in objects of the same class:

```
> is(exSet[1, ], "ExpressionSet")
[1] TRUE
```

b Phenotypic as well as expression data is subset along with `exSet`:

```
> dim(pData(exSet[, 1]))
[1] 1 3
> dim(exprs(exSet[, 1]))
[1] 500   1
```

c
```
> exSet[, exSet$sex == "Female"]
ExpressionSet (storageMode: lockedEnvironment)
assayData: 500 features, 11 samples
  element names: exprs, se.exprs
phenoData
  rowNames: A, E, ..., Z (11 total)
  varLabels and varMetadata:
    sex: Female/Male
    type: Case/Control
    score: Testing Score
featureData
  featureNames: AFFX-MurIL2_at, AFFX-MurIL10_at, ..., 31739_at (500 total)
  varLabels and varMetadata: none
experimentData: use 'experimentData(object)'
Annotation [1] "hgu95av2"
```

1.4.1 Building an ExpressionSet From Scratch

Many readers will have their own data and want to create an instance of the *ExpressionSet* from it. In the next set of exercises, you will learn how to create a new *ExpressionSet* instance given a matrix of expression values and a *data.frame* containing the phenoData. Other tutorials will cover the creation of the expression matrix from raw CEL files for microarray data.

You can save objects in your R session to a file using the `save` function. By default, this will create a file in R's internal binary format. The same binary file produced by a call to `save` can be used on Linux, OS X, and Windows. In most cases the saved file will have an extension `.rda`, and hence they are sometimes referred to as *rda files*. You can load the objects saved in an rda file using the `load` function.

In the code below, we will load a large *matrix* of expression values stored in the file `ALLmat.rda`. The example assumes that the file is in the current working directory. You can change the working directory using `setwd`. The file is supplied with the **BiocCaseStudies** package, and you will need to either specify the path to that package or set the current working directory to that package. If you are using the Windows GUI you can set the directory using a menu, but otherwise the code below will set the working directory appropriately.

```
> getwd()
```

```
[1] "/Users/robert/bioC/Docs/Books/useR-book/R_bioc_intro"
```

```
> ls()
```

```
 [1] "Rvers"
 [2] "basename"
 [3] "biocUrls"
 [4] "exSet"
 [5] "l"
 [6] "m"
 [7] "pets"
 [8] "repos"
 [9] "sample.ExpressionSet"
[10] "x"
[11] "y"
[12] "z"
```

```
> dataPath = system.file("extdata",
+     package = "BiocCaseStudies")
> oldLoc = setwd(dataPath)
> dir()
```

```
[1] "ALL-sample-info.txt"
[2] "ALL-varMeta.txt"
[3] "ALLmat.rda"
```

```
> load("ALLmat.rda")
> ls()
```

```
 [1] "ALLmat"
 [2] "Rvers"
```

```
 [3] "basename"
 [4] "biocUrls"
 [5] "dataPath"
 [6] "exSet"
 [7] "l"
 [8] "m"
 [9] "oldLoc"
[10] "pets"
[11] "repos"
[12] "sample.ExpressionSet"
[13] "x"
[14] "y"
[15] "z"
```

The `ls` function lists the R objects in your current working environment. You should see a new object named `ALLmat` appear after the call to `load`.

Exercise 4

a *Open and read the help page for* `load`.

b *Determine the class and dimension of the matrix.*

c *How would you set the working directory back to what it was, before we changed it to the* `extdata` *directory of the* ***BiocCaseStudies*** *package?*

Solutions:

a
```
> help(load)
```

b
```
> class(ALLmat)
[1] "matrix"
> dim(ALLmat)
[1] 12625   128
```

c
```
> setwd(oldLoc)
```

Covariates describing the samples in this experiment have been saved to a whitespace-delimited text file called `ALL-sample-info.txt`. Delimited text files are common and can be produced from Microsoft Excel by saving as a "csv" file (this stands for comma separated values, but the separator does not have to be a comma).

R's `read.table` function is a powerful tool for reading delimited text files. Below, you will use it to read in the phenoData.

```
> samples <- read.table(file.path(dataPath,
+     "ALL-sample-info.txt"), header = TRUE,
+     check.names = FALSE)
```

Exercise 5

a *What class does* `read.table` *return?*

b *Determine the column names of* `samples`. *Hint:* `apropos("name")`.

c *Examine the sex and age of the 15th and 30th samples. Do the same for the sample with* `cod` *matching 11005.*

Solutions:

a
```
> class(samples)
```
```
[1] "data.frame"
```

b
```
> names(samples)
```
```
 [1] "cod"            "diagnosis"
 [3] "sex"            "age"
 [5] "BT"             "remission"
 [7] "CR"             "date.cr"
 [9] "t(4;11)"        "t(9;22)"
[11] "cyto.normal"    "citog"
[13] "mol.biol"       "fusion protein"
[15] "mdr"            "kinet"
[17] "ccr"            "relapse"
[19] "transplant"     "f.u"
[21] "date last seen"
```

c
```
> samples[c(15, 30), c("sex", "age")]
```
```
      sex age
09008   M  41
20002   F  58
```
```
> samples[samples$cod == "11005",
+     c("sex", "age")]
```
```
      sex age
11005   M  27
```

To make the phenoData more self-documenting, we have a file named `ALL-varMeta.txt` that gives a description for each column of `samples`. You can use `read.table` to read this file into an R object.

```
> varInfo <- read.table(file.path(dataPath,
+     "ALL-varMeta.txt"), header = TRUE,
+     colClasses = "character")
> varInfo[c("sex", "cod", "mol.biol"),
+     , drop = FALSE]
```
```
              labelDescription
sex       Gender of the patient
cod                  Patient ID
mol.biol      molecular biology
```

Bioconductor's **Biobase** package provides a class called *AnnotatedDataFrame* that allows you to store the column descriptions with the data. Create an *AnnotatedDataFrame* instance for our phenoData by following the example below.

```
> pd <- new("AnnotatedDataFrame",
+     data = samples, varMetadata = varInfo)
```

Now that you have a *matrix* of expression values (`ALLmat`) and an *AnnotatedDataFrame* containing the phenotype information (`pd`), you are ready to put the pieces together and create an *ExpressionSet*.

```
> ALLSet <- new("ExpressionSet",
+     exprs = ALLmat, phenoData = pd,
+     annotation = "hgu95av2")
```

The `annotation` argument is intended to hold the name of the R package that provides annotation data for the chip used in the experiment. In this case, the appropriate annotation package is **hgu95av2**.

Exercise 6
What other tools are there for documenting an ExpressionSet instance? Can you add PubMed IDs for papers describing the data?

Solutions: You need to read the manual page for the *ExpressionSet* class. Yes.

Now, how can you extract information from the ExpressionSet you have created? A number of accessor functions are available to extract data from an *ExpressionSet* instance. You can access the columns of the phenotype data (an *AnnotatedDataFrame* instance) using `$`. It sometimes might be necessary to quote the column names if they contain white space or other special characters.

```
> ALLSet$sex[1:5] == "F"

[1] FALSE FALSE  TRUE FALSE FALSE

> ALLSet$"t(9;22)"[1:5]

[1]  TRUE FALSE    NA FALSE FALSE
```

You can retrieve the names of the features using `featureNames`. For many microarray datasets, the feature names are the probeset identifiers.

```
> featureNames(ALLSet)[1:5]

[1] "1000_at"   "1001_at"   "1002_f_at"
[4] "1003_s_at" "1004_at"
```

The unique identifiers of the samples in the data set are available via the `sampleNames` method. The `varLabels` method lists the column names of the phenotype data:

```
> sampleNames(ALLSet)[1:5]

[1] "01005" "01010" "03002" "04006"
[5] "04007"

> varLabels(ALLSet)

 [1] "cod"            "diagnosis"
 [3] "sex"            "age"
 [5] "BT"             "remission"
 [7] "CR"             "date.cr"
```

```
 [9] "t(4;11)"         "t(9;22)"
[11] "cyto.normal"     "citog"
[13] "mol.biol"        "fusion protein"
[15] "mdr"             "kinet"
[17] "ccr"             "relapse"
[19] "transplant"      "f.u"
[21] "date last seen"
```

You can extract the expression *matrix* and the *AnnotatedDataFrame* of sample information using `exprs` and `phenoData`, respectively:

```
> mat <- exprs(ALLSet)
> adf <- phenoData(ALLSet)
```

1.4.2 Functions

Writing functions in R is quite easy. All functions take inputs and return values. In R the value returned by a function is either the value that is explicitly returned by a call to the function `return` or it is simply the value of the last expression. So, the two functions below will return identical values.

```
> sq1 <- function(x) return(x * x)
> sq2 <- function(x) x * x
```

These functions are *vectorized*. This means you can pass a vector `x` to the function and each element of `x` will be squared. Note that if you use two vectors of unequal length for any vectorized operation R will try to recycle the shorter one. While this can be useful for certain applications it can also lead to unexpected results.

Exercise 7
In this exercise we want you to write a function that we will use in the next section. It relies on the R function `paste` *and you should read the manual page. It should take a string as input and return that string with a caret prepended. Let's call it ppc, what we want is* `ppc("xx")` *returns* `"^xx"`.

Solutions:

```
> ppc <- function(x) paste("^", x,
+     sep = "")
```

One of the places that user defined functions are often used is with the apply family of functions and in the next section we will see some examples.

1.4.3 The `apply` functions

In R a great deal of work is done by applying some function to all elements of a list, matrix or array. There are several functions available for you to use, `apply`, `lapply`, `sapply` are the most commonly used. The function `eapply` is also available for applying a function to each element of an environment. You will learn more about how to create and how to work with environments in the next

section. To understand how the apply functions work we will use them to explore some of the metadata for the HGU95Av2 chips. Since these data are stored in environments we will make use of the `eapply` function. The `hgu95av2MAP` contains the mappings between Affymetrix identifiers and chromosome band locations. For example, in the code below we find the chromosome band that the gene, for probe `1001_at` (TIE1) maps to.

```
> library("hgu95av2")
> hgu95av2MAP$"1001_at"
```

```
[1] "1p34-p33"
```

We can extract all of the map locations for a particular chromosome or part of a chromosome by using regular expressions and the apply family of functions. First let's be more explicit about the problem, say we want to find all genes that map to the p arm of chromosome 17. Then we know that their map positions will all start with the characters `17p`. This is a simple regular expression, `^17p`, where the caret, `^`, means that we should match the start of the word. We do this in two steps, first we use `eapply` and `grep` and ask for grep to return the value that matched.

```
> myPos = eapply(hgu95av2MAP, function(x) grep("^17p",
+     x, value = TRUE))
> myPos = unlist(myPos)
> length(myPos)
```

```
[1] 191
```

Here we used an anonymous function to process each element of the `hgu95av2MAP` environment. We could have named it and then used it.

```
> f17p = function(x) grep("^17p",
+     x, value = TRUE)
> myPos2 = eapply(hgu95av2MAP, f17p)
> myPos2 = unlist(myPos2)
> length(myPos2)
```

```
[1] 191
```

Exercise 8
Use the function `ppc` that you wrote in the previous exercise to create a new function that can find and return the probes that map to any chromosome (just prepend the caret to the chromosome number) or the chromosome number with a p or a q after it.

Solutions:

```
> myFindMap = function(mapEnv, which) {
+     myg = ppc(which)
+     a1 = eapply(mapEnv, function(x) grep(myg,
+         x, value = TRUE))
+     unlist(a1)
+ }
```

1.4.4 Environments

In R, an `environment` is a set of symbol-value pairs. These are very similar to lists, but there is no natural ordering of the values and so you cannot make use of numeric indices. Otherwise they behave the same way. In the previous section you have already used an environment that stored the mapping between Affymetrix identifiers and chromosome band locations. Here, you will learn how to work with your own environemnts. We first create an environment and carry out some simple tasks, such as storing things in it, remove things from it, and listing the contents.

```
> e1 <- new.env(hash = TRUE)
> e1$a <- rnorm(10)
> e1$b <- runif(20)
> ls(e1)
```

```
[1] "a" "b"
```

```
> xx <- as.list(e1)
> names(xx)
```

```
[1] "a" "b"
```

```
> rm(a, envir = e1)
```

Exercise 9
In this exercise you will learn how to create and environment and to place some objects into it.

a *Create an environment and put a copy of* `exSet` *into it.*

b *Fit a linear model to the data* `x=1:10, y=2 * x + rnorm(10, sd=0.25)`*, and also place this into your environment.*

c *Write a function,* `myExtract`*, that takes an environment as an argument and returns a list, one element is the variable* `score` *from* `exSet` *and the other is the vector of coefficients from the linear model.*

Solutions:

a
```
> theEnv <- new.env(hash = TRUE)
> theEnv$exSet <- exSet
```

b
```
> x <- 1:10
> y <- 2 * x + rnorm(10, sd = 0.25)
> model <- lm(y ~ x)
> theEnv$model <- model
```

c
```
> myExtract <- function(env) {
+     eset <- env$eset
+     model <- env$model
+     return(list(score = exSet$score,
+         coeff = model$coeff))
+ }
> myExtract(theEnv)
$score
 [1] 0.75 0.40 0.73 0.42 0.93 0.22 0.96
 [8] 0.79 0.37 0.63 0.26 0.36 0.41 0.80
[15] 0.10 0.41 0.16 0.72 0.17 0.74 0.35
[22] 0.77 0.27 0.98 0.94 0.32

$coeff
(Intercept)           x
 0.09544193  1.97579576
```

1.5 Something Harder

In one of the following tutorials will spend some time discussing machine learning (ML), but here we will just use one simple algorithm, k-nearest neighbors (knn) to make predictions. You should read the R manual page for a description of knn.

```
> library("class")
> apropos("knn")
[1] "knn"    "knn.cv" "knn1"
```

The knn algorithm predicts the class of a given observation (the test case) according to a majority vote of the k nearest neighbors in the training set. We will show how you can use this to predict the case/control status of sample 1, given the expression data on samples 2 through 26.

```
> exprsExSet <- exprs(exSet)
> classExSet <- exSet$type
> esub <- exSet[, -1]
> pred1 <- knn(t(exprs(esub)), exprs(exSet)[,
+     1], esub$type)
> classExSet[1]
```

```
[1] Control
Levels: Case Control
```

Exercise 10
In this exercise we want you to write a simple cross-validation function (note that `knn.cv` *does this already in a far more efficient way).*

a *Write a function, that takes an ExpressionSet and a character giving the name of the covariate as its input and carries out a leave-one-out set of predictions over all samples (i.e. leaving out the first sample, second sample, ...). Your function should return the vector of predicted values for the given covariate.*

b *Modify your function to allow the user to specify some of the parameters for knn, such as k.*

Solutions:

a
```
> predEset <- function(exSet, cov) {
+     nc <- ncol(exSet)
+     pred <- character(nc)
+     for (i in 1:nc) {
+         esub <- exSet[, -i]
+         pred[i] <- as.character(knn(t(exprs(esub)),
+             exprs(exSet)[, i],
+             esub[[cov]]))
+     }
+     return(pred)
+ }
> predEset(exSet, "type")
 [1] "Control" "Control" "Control"
 [4] "Case"    "Control" "Control"
 [7] "Control" "Control" "Case"
[10] "Case"    "Case"    "Case"
[13] "Case"    "Control" "Control"
[16] "Case"    "Control" "Case"
[19] "Case"    "Case"    "Case"
[22] "Case"    "Control" "Case"
[25] "Case"    "Case"
```

b Use the `...` argument in the definition of you function to pass all undefined arguments to `knn`.

```
> predEset <- function(exSet, cov,
+     ...) {
+     nc <- ncol(exSet)
+     pred <- character(nc)
+     pred <- character()
+     for (i in 1:nc) {
+         pred[i] <- as.character(knn(t(exprs(esub)),
+             exprs(exSet)[, i],
+             esub[[cov]], ...))
+     }
+     return(pred)
+ }
```

1.5.1 Gene Ontology

This next code chunk shows how to use apply-type functions to extract GO terms in the molecular function ontology, for each Affymetrix probe set. The environment `hgu95av2GO` provides mappings between Affymetrix IDs and GO terms via EntrezGene IDs. On first glance the code looks quite complicated but all of the elements have already been used in the previous sections. So at this point it should be easy for you to understand what is going on.

```
> library("GO")
> library("hgu95av2")
> affyGO <- as.list(hgu95av2GO)
> affyMF <- lapply(affyGO, function(x) {
+     onts <- sapply(x, function(z) z$Ontology)
+     if (is.null(unlist(onts)) ||
+         is.na(unlist(onts)))
+         NA
+     else unique(names(onts)[onts ==
+         "MF"])
+ })
```

Exercise 11

a *How are the GO terms stored? What information is available for each?*

b *What are the evidence codes and what do they mean?*

c *Turn this code into a function that would allow users to obtain either the MF, BP or CC data.*

d *Extend this function to allow the user to include only given evidence codes. (Or if you think it better - to exclude specific codes).*

Solutions:

a
```
> names(hgu95av2GO$"35889_at"[[1]])
[1] "GOID"     "Evidence" "Ontology"
```

b see help for **hgu95av2GO** (`? hgu95av2GO`)

c
```
> getGO <- function(ontology) {
+     affyGO = as.list(hgu95av2GO)
+     if (!ontology %in% c("MF",
+         "BP", "CC"))
+         stop("invalid ontology identifier")
+     affyOnts = lapply(affyGO, function(x) {
+         onts = sapply(x, function(z) z$Ontology)
+         if (is.null(unlist(onts)) ||
+             is.na(unlist(onts)))
+             NA
+         else unique(names(onts)[onts ==
+             ontology])
+     })
+     return(affyOnts)
+ }
> getGO("MF")[10:12]
$`226_at`
[1] "GO:0000166" "GO:0005515"
[3] "GO:0008603" "GO:0016301"
[5] "GO:0030552"

$`388_at`
[1] "GO:0005515" "GO:0016303"
[3] "GO:0035014"

$`33680_f_at`
[1] NA
```

d
```
> getGO2 <- function(ontology, evidence) {
+     affyGO = as.list(hgu95av2GO)
+     if (!ontology %in% c("MF",
+         "BP", "CC"))
+         stop("invalid ontology identifier")
+     affyOnts = lapply(affyGO, function(x) {
+         onts = sapply(x, function(z) z$Ontology)
+         evs = sapply(x, function(z) z$Evidence)
+         if (is.null(unlist(onts)) ||
+             is.na(unlist(onts)) ||
+             is.null(unlist(evs)) ||
+             is.na(unlist(evs)))
+             NA
+         else unique(names(onts)[onts ==
+             ontology & evs == evidence])
+     })
+     return(affyOnts)
+ }
> getGO2("MF", "IEA")[10:12]
$`226_at`
[1] "GO:0000166" "GO:0008603"
[3] "GO:0016301" "GO:0030552"

$`388_at`
[1] "GO:0035014"
```

1.6 Finding help in R

In Section 1 you have already learned about the `?` operator and how you can get information about a certain R function or object. In addition there are a lot of other sources for help in and out of R. Function `apropos` can be used to find objects in the search path partially matching the given character string. `find` also locates objects, yet in a more restrictive manner.

```
> apropos("mean")

 [1] "kmeans"          "weighted.mean"
 [3] "colMeans"        "mean"
 [5] "mean.Date"       "mean.POSIXct"
 [7] "mean.POSIXlt"    "mean.data.frame"
 [9] "mean.default"    "mean.difftime"
[11] "rowMeans"

> find("mean")

[1] "package:base"
```

If you want to get information about a certain topic or concept, try `help.search`. The function searches the help system for documentation matching a given character string in the (file) name, alias, title, concept or keyword entries. Names and titles of the matched help entries are displayed.

```
> help.search("mean")
```

Moreover there is a wealth of information just waiting for you out in the web: For many of the usual R-related questions you may most likely find an answer in the R-FAQ at `http://cran.r-project.org/faqs.html`. A more specialized source for help are the R and Bioconductor mailing lists (`http://www.r-project.org/mail.html`, `http://www.bioconductor.org/mailList.html`). You can subscribe to different sublists, regarding your interests and level of expertise and post your questions to the R society. Before doing so, you should by all means read the posting guides. Many questions on the mailing lists will most likely not be answered because major posting rules have been violated. It is also a good idea to search the online mailing archives before posting a question. Most of them have already been asked and answered by someone else. A searchable Bioconductor archive can be found at `http://dir.gmane.org/gmane.science.biology.informatics.conductor`, the R archives at `http://dir.gmane.org/index.php?prefix=gmane.comp.lang.r.`. All of these links can of course also be found on the Bioconductor and R-Project webpages.

Exercise 12

a *There are a number of different plotting functions available. Can you find them?*

b *Try to find out how to do a Mann-Whitney test.*

c *Take a look at the R posting guide and find out about the most common mistakes when posting a question.*

d *Use the searchable R mail archive and find out how to color tick marks in a plot with the* `segments` *function. You may need this information in one of the following exercises! [Hint: Try the keywords 'plot', 'tick' 'labels' and 'colour']*

Solutions:

a
```
> apropos("plot")
```
```
[1] ".__M__Makesense:geneplotter"
[2] ".__M__imageMap:geneplotter"
[3] ".__T__Makesense:geneplotter"
```

b
```
> help.search("mann-whitney")
```

1.7 Graphics

Graphics and visualization are important issues when dealing with complex data like the ones typicaly found in biological science. In this section you will work through some examples that allow you to create very general plots in R. Both R and Bioconductor offer a range of functions that generate various graphical representations of your data. For each function there are usually numerous parameters that enable the user to tailor the output to the specific needs. We only touch on some of the issues and tools. Interested readers should look at Chapter 10 of Gentleman et al. (2005) or for even more detail Murrell (2005).

The function `plot` can be used to produce dot plots. Read through its documentation (`? plot`) and also take a look into the documentation for `par`, which controls most of the parameters for R's base graphics. We now want to use the `plot` function to compare the gene expression intensities of two samples from our data set.

```
> x <- exprs(exSet[, 1])
> y <- exprs(exSet[, 3])
> plot(x, y)
```

From the plot in Figure 1.1 we can see that the measurements for each probe are highly correlated between the two samples. They form an almost perfect line along the 45 degrees diagonal.

Exercise 13
The axis annotation of the plot in Figure 1.1 is not very informative. Can you add more meanigful axis labels and a title to the plot? Can you change the plotting symbols? Add the 45 degrees diagonal to the plot.

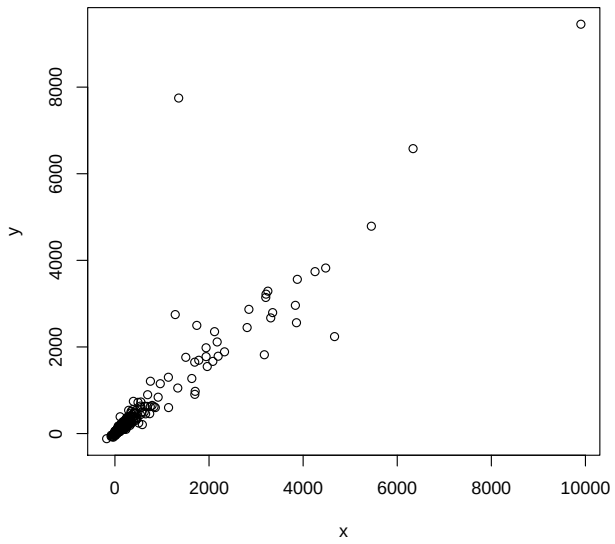

Figure 1.1. Scatter plot of expression intensities for two samples.

Solutions:

```
> plot(x, y, xlab = "gene expression sample #1",
+     ylab = "gene expression sample #3",
+     main = "scatterplot of expression intensities",
+     pch = 20)
> abline(a = 0, b = 1)
```

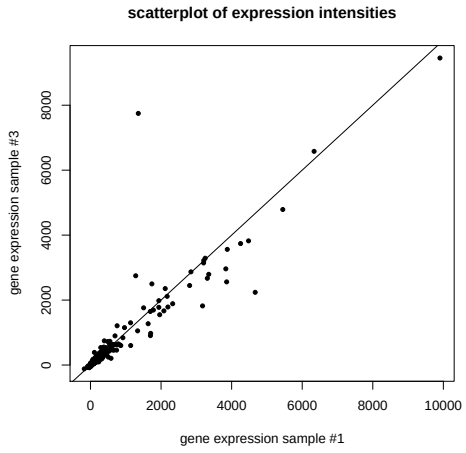

Figure 1.2. Scatter plot of expression intensities for two samples.

Visualization is an important aspect when trying to detect possible problems or

inconsistencies in the data. In the simplest case one can spot such problems by looking at distribution summaries. A good example for this is the dependency of the measurement intensity of a microarray probe on its GC-content. To demonstrate this, we need to load a more extended data set from the **CLL** package which includes the raw measurement values for each probe from an experiment using the Affy hguav2 chip. The `basecontent` from package **matchprobes** calculates the base frequencies for each probe based on a sequence vector.

```
> library("CLL")
> library("matchprobes")
> library("hgu95av2probe")
> library("hgu95av2cdf")
> library("RColorBrewer")
> data("CLLbatch")
> baseCt = basecontent(hgu95av2probe$sequence)
> gcCt = ordered(baseCt[, "C"] +
+     baseCt[, "G"])
```

We now need to match the probes via their position on the array to the expression values in the `CLLbatch` data set. Since we have several samples in the set, we will compute mean expression values for each probe.

```
> iab = get("xy2i", "package:hgu95av2cdf")(hgu95av2probe$x,
+     hgu95av2probe$y)
> meanlogint = rowMeans(log2(exprs(CLLbatch)[iab,
+     ]))
```

Finally, we can plot the log-transformed expression data of the probes grouped by their GC-content. Our data consists of thousands of points, thus looking at the whole bulk of data doesn't make much sense. Instead, we want to focus on the important features provided by distribution summaries. We can get the most comprehensive summary of the data by means of a box plot. Read the documentation of function `boxplot` if you don't know how to interpret such plots.

```
> mycol <- colorRampPalette(brewer.pal(9,
+     "GnBu"))(nlevels(gcCt))
> exlab <- expression(log[2] ~ intensity)
> boxplot(meanlogint ~ gcCt, col = mycol,
+     outline = FALSE, xlab = "Number of G and C",
+     ylab = exlab)
```

Density plots are less detailed than box plots, but sometimes you need a more concise representation of the shape of the distributions. We can plot multiple distribution densities using the function `multidensity` from package **geneplotter**. For this plot we will focus on the ten most popular groups.

```
> library("geneplotter")
> tab = table(gcCt)
> ord = sort(order(tab, decreasing = TRUE)[1:10])
> datsel = (as.character(gcCt) %in%
+     names(tab)[ord])
> gcCtsel = ordered(gcCt[datsel])
```

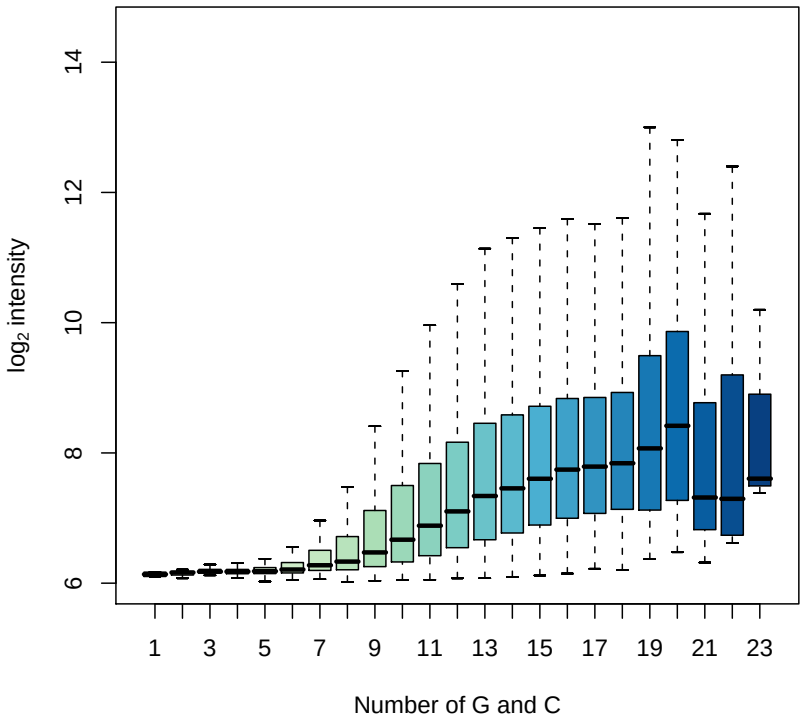

Figure 1.3. Box plot of distributions of log_2-intensities from the CLL datast grouped by GC content.

```
> mycolsel = mycol[sort(ord)]
> multidensity(meanlogint[datsel] ~
+     gcCtsel, xlim = c(6, 11), col = mycolsel,
+     lwd = 2)
```

Exercise 14
Another useful distribution summary are plots of the empirical cumulative distribution function. Create a plot similar to the one in Figure **??** *using the function* `multiecdf`.

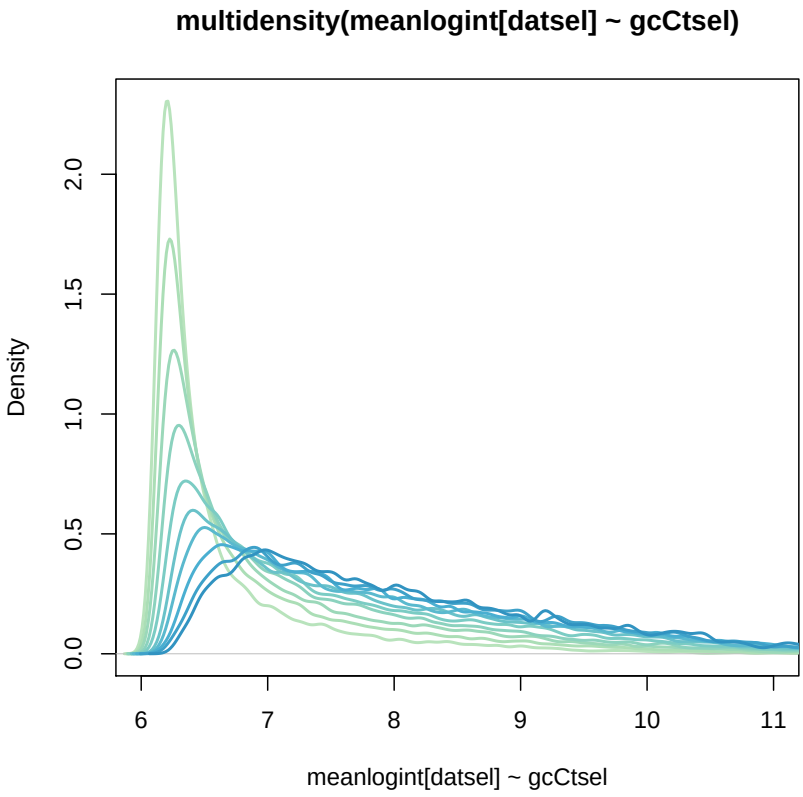

Figure 1.4. Density plot of distributions of log_2-intensities from the CLL datast grouped by GC content.

Solutions:

```
> multiecdf(meanlogint[datsel] ~
+     gcCtsel, xlim = c(6, 11), xlab = exlab,
+     ylab = "ECDF", main = "", col = mycolsel,
+     lwd = 2)
```

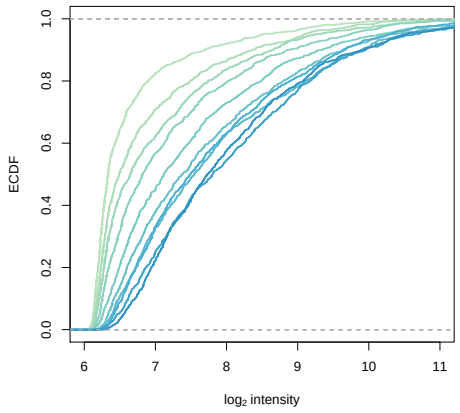

Figure 1.5. Scatter plot of expression intensities for two samples.

Testing crossreference: This should link to Lab R Bioconductor introduction. Let's see if it works.

The version number of R and the packages and their versions that were used to generate this document are listed below

```
R version 2.5.0 RC (2007-04-22 r41275)
i386-apple-darwin8.9.1

locale:
C

attached base packages:
[1] "tools"     "stats"     "graphics"
[4] "grDevices" "utils"     "datasets"
[7] "methods"   "base"

other attached packages:
    geneplotter         lattice
       "1.14.0"        "0.15-5"
       annotate     hgu95av2cdf
       "1.14.1"        "1.16.0"
  hgu95av2probe     matchprobes
       "1.16.2"         "1.8.1"
            CLL            affy
        "1.2.2"        "1.14.0"
         affyio BiocCaseStudies
        "1.4.0"         "1.0.5"
   RColorBrewer              GO
        "0.2-3"        "1.16.0"
          class        hgu95av2
       "7.2-33"        "1.16.0"
        Biobase          weaver
       "1.14.0"         "1.2.0"
      codetools          digest
        "0.1-1"         "0.3.0"
```

References

R. Gentleman, W. Huber, V. Carey, R. Irizarry, and S. Dudoit, editors. *Bioinformatics and Computational Biology Solutions Using R and Bioconductor*. Springer, 2005.

P. Murrell. *R Graphics*. Chapman and Hall, 2005.