Preface

Recent developments in genomics and molecular biology finally carry the promise of understanding the functions of complex biological systems on a whole genome level. These developments have led to enormous amounts of data generated in highthroughput technologies, most prominently in gene expression microarrays. Within the Bioconductor project, an increasing number of researchers are trying to establish solutions for the analysis of such data, combining knowledge from such diverse disciplines as statistics, computer science, bioinformatics, and molecular biology.

With microarrays becoming a standard technology in many molecular biology labs, there is increased demand for comprehensive yet easy to follow instructions to the complex data analysis process. After many years of teaching introductory Bioconductor courses we can identify the main topics of interest, the common misunderstandings and pitfalls, and have learned to better understand key problems with which beginners to the analysis tasks are often challenged. In this book, we try to guide the readers through each step of the data analysis process, beginning from import and data processing to the generation of lists of differentially expressed genes and finally the modeling and interpretation of these lists in downstream analyses. Every chapter focuses on real data use cases that illustrate the problem, and we present both executable code and detailed background information for each step. A companion Webpage to this book can be found at `http://www.bioconductor.org/pub/docs/BioconductorCaseStudies`

Acknowledgments

We would like to thank Stefano Iacus for organizing the annual Bioconductor courses in Bressanone, Italy, which are the basis for this book and May Alipao, who has helped to organize the Bioconductor courses at the Hutchinson Center. We also thank the many students who attended these and other courses and whose countless questions, helpful remarks, and enthusiasm provided a valuable source of inspiration during the genesis of this book, immensely shaping the outcome you hold in your hands right now. We thank all the co-authors of the individual chapters. Their

expert knowledge was highly appreciated and helped to clarify the key concepts and to point out critical steps. We would also like to thank the following individuals who contributed in one or another way to this book: James W. MacDonald, Marc Carlson, Nolwenn LeMeur, Brig Mecham, Joern Toedling, Steffen Durinck, Anna Freni Sterrantino, Deepayan Sarkar, Rafael Irizarry, Jean (Zhijin) Wu, and Li Long.

RG expresses his thanks and appreciation to Tanja and Sophie, for their encouragement and understanding during the long hours spent working on this and other projects.

Florian Hahne
Wolfgang Huber
Robert Gentleman
Seth Falcon

2

R and Bioconductor Introduction

R. Gentleman, F. Hahne, S. Falcon, and M. Morgan

Abstract

In this chapter we cover basic uses of R and begin working with Bioconductor datasets and tools. Topics covered include simple R programming, R graphics, and working with *environment*s as hash tables. We introduce the *ExpressionSet* class as an example for a basic Bioconductor structure used for holding genomic data, in this case expression microarray data. And we explore some visualization techniques for gene expression data to get a feeling for R's extensive graphical capabilities.

2.1 Finding help in R

To get started with R and Bioconductor it is important to know where you can find help for the numerous functions, classes, and concepts you are about to come across. The `?` operator is the most immediate source of information about R objects. Preceding the name of a function with `?` quickly gets you to the manual page of this function. Possible arguments and return values should be introduced there, and you will find basic information about the purpose and application of the function. A special flavor of `?` exists for classes. `class ? foo` will get you to the manual page of class `foo` where you will often also find information about available methods for this class.

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.

F. Hahne et al., *Bioconductor Case Studies*, DOI: 10.1007/978-0-387-77240-0_2,

```
> apropos("mean")
 [1] "colMeans"         "kmeans"
 [3] "mean"             "mean.Date"
 [5] "mean.POSIXct"     "mean.POSIXlt"
 [7] "mean.data.frame" "mean.default"
 [9] "mean.difftime"    "rowMeans"
[11] "weighted.mean"
> 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 on the Web: A very good introduction is R-Foundation (2007). 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`. More specialized sources 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 read the posting guides. Often questions on the mailing lists are not answered because major posting rules have been violated. It is also a good idea to search the online mailing archives before posting a question. A lot 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` and the R archives at `http://dir.gmane.org/index.php?prefix=gmane.comp.lang.r.`. All of these links can also be found on the Bioconductor and R-Project Web pages.

Most of the Bioconductor packages contain another valuable source of information through their package vignettes. Vignettes are supposed to describe more thoroughly the steps needed to perform one or several of the specific tasks for which the package was designed. Text and executable code are bundled together in one document, similar to the document you are reading right now, which makes it easy to reproduce individual steps on your own machine or to introduce modifications specific to your own task. The function `openVignette` in **Biobase** can be used to open PDF versions of the available vignettes. Vignettes become available once you load a package (see the next section for an introduction to the concept of packages).

Exercise 2.1

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

b. *Try to find out which function to use in order to perform a Mann–Whitney test.*

c. *Open the PDF version of the vignette "Bioconductor Overview" which is part of the* ***Biobase*** *package.*

2.2 Working with packages

The design of R and Bioconductor is modular. A lot of the functionality is provided by additional units of software called packages. There are many hundreds of packages available for R and around 260 for Bioconductor. Before we begin working with data, it is important that you learn how to find, download, and install packages.

Different methods can be used for this task, and over time we expect them to become more standardized. R packages are stored in libraries. You can have multiple libraries on your computer, although most people have only one on their personal machine. To add a package to your library, you need to download and install it. After that, each time you want to use the package, you need to 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 (Windows or Mac OS X). On these platforms you simply select the packages you want from a list, and they are downloaded and installed. Installing a package does not automatically load it 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"))
```

The command `update.packages` can be used to check for and install new versions of already installed packages. Note that you need to supply

`update.packages` with the URL to a Bioconductor repository in order to update Bioconductor packages as well. The recommended way of updating all your installed packages is:

```
> source("http://bioconductor.org/biocLite.R")
> update.packages(repos=biocinstallRepos(), ask=FALSE)
```

Exercise 2.2
What is the output of function `sessionInfo`*?*

2.3 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 your own short scripts. If you find it easy to answer 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 home page at `http://cran.r-project.org/manuals/R-intro.html` to acquire a more solid foundation of the nitty-gritty details of the language.

Exercise 2.3

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`
   – *An integer vector* `y` *with elements 1 to 100*
   – *A logical vector* `z` *indicating the elements of* `y` *that are below 10*
   – *A named character vector* `pets` *with elements* `dog`, `cat`, *and* `bird`.
   You can choose whatever names you like for your new virtual pets.

b. *What happens to vectors in arithmetic expressions? What is the result of*

```
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?*

2.3.1 Functions

Writing functions in R is easy. All functions take inputs and they return values. In R the value returned by a function is either specified explicitly 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. Although this can be useful for certain applications, it can also lead to unexpected results.

Exercise 2.4
In this exercise we want you to write a function that we use in the next section. It relies on the R function `paste`*, and you may want to read the function's manual page. The function should take a string as input and return that string with a caret prepended. Let's call it ppc; what we want is that* `ppc("xx")` *returns* `"^xx"`.

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

2.3.2 The apply family of 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*. We show more about how to create and how to work with environments in the next section.

To understand how the `apply` family of functions works, we use them to explore some of the metadata for the Affymetrix®HG-U95Av2 GeneChip®. Because these data are stored in *environment*s we make use of the `eapply` function.

The `hgu95av2MAP` *environment* contains the mappings between Affymetrix identifiers and chromosome band locations. For example, in the code below we find the chromosome band to which the gene, for probe `1001_at` (TIE1), maps.

```
> library("hgu95av2.db")
> 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] 190
```

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)
> identical(myPos, myPos2)
[1] TRUE
```

Exercise 2.5
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.*

2.3.3 Environments

In R, an *environment* is a set of symbol–value pairs. These are similar to lists, but there is no natural ordering of the values and so you cannot make use of numeric indices. Also unlike lists, partial matching of the symbols will not work. 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, we show how to work with your own *environment*s.

We first create an *environment* and carry out some simple tasks, such as storing things in it, removing 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 2.6

a. *Create an environment and put in the chromosomal locations of all genes on chromosome 18 using your function from the last exercise.*

b. *Put into the environment a second function that takes the strings of chromosomal locations and strips the "18" from each string. The function* `gsub` *can help you with that.*

c. *Now write a function,* `myExtract`*, that takes an environment as an argument and returns a vector of stripped chromosomal locations (i.e., apply the stripping function in the environment to the vector of chromosomal locations in the same environment).*

2.4 Structures for genomic data

Genomic data can be very complex, usually consisting of a number of different bits and pieces. In Bioconductor we have taken the approach that these pieces should be stored in a single structure to easily manage the data. The package **Biobase** contains standardized data structures to represent genomic data. The *ExpressionSet* class is designed to combine several different sources of information into a single convenient structure. An *ExpressionSet* can be manipulated (e.g., subsetted, copied), and is the input to or output of many Bioconductor functions.

The data in an *ExpressionSet* consist of

- `assayData`: Expression data from microarray experiments (`assayData` is used to hint at the methods used to access different data components, as we show below).
- metadata: A description of the samples in the experiment (`phenoData`), metadata about the features on the chip or technology used for the experiment (`featureData`), and further annotations for the features, for example gene annotations from biomedical databases (`annotation`).
- `experimentData`: A flexible structure to describe the experiment.

The *ExpressionSet* class coordinates all of these data, so that you do not usually have to worry about the details. However, an *ExpressionSet* needs to be created in the first place, because it will be the starting point for many of the analyses using Bioconductor software.

In this section we learn how to create and manipulate *ExpressionSet* objects, and by doing that we again practice some basic R skills.

2.4.1 Building an ExpressionSet from .CEL and other files

Many users have access to .CEL or other files produced by microarray chip manufacturer hardware. Usually the strategy is to use a Bioconductor package such as **affyPLM**, **affy**, **oligo**, **limma**, or **arrayMagic** to read these files. These Bioconductor packages have functions (e.g., `ReadAffy`, `expresso`, or `justRMA` in **affy**) to read CEL files and perform preliminary preprocessing, and to represent the resulting data as an *ExpressionSet* or other type of object. Suppose the result from reading and preprocessing CEL or other files is named `object`, and `object` is different from *ExpressionSet*; a good bet is to try, for example,

```
> library(convert)
> as(object, "ExpressionSet")
```

It might be the case that no converter is available. The path then is to extract relevant data from `object` and use this to create an *ExpressionSet* using the instructions below.

2.4.2 Building an ExpressionSet from scratch

As mentioned before, the data from many high-throughput genomic experiments, such as microarray experiments, usually consist of several conceptually distinct parts: assay data, sample annotations, feature annotations, and an overall description of the experiment. We construct each of these components, and then assemble them into an *ExpressionSet*.

Assay data

One important part of the experiment is a matrix of "expression" values. The values are usually derived from microarrays of one sort or another, perhaps after initial processing by manufacturer software or Bioconductor packages. 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.

A likely scenario is that your assay data are in a "tab-delimited" text file (as exported from a spreadsheet, for instance) with rows corresponding to features and columns to samples. The strategy is to read this file into R using the `read.table` command, converting the result to a *matrix*. A typical command to read a tab-delimited file that includes column headers is

```
> dataDirectory = system.file("extdata", package="Biobase")
> exprsFile = file.path(dataDirectory, "exprsData.txt")
> exprs = as.matrix(read.table(exprsFile, header=TRUE,
      sep="\t", row.names=1, as.is=TRUE))
```

The first two lines create a file path pointing to where the assay data are stored; replace these with a character string pointing to your own file, for example,

```
> exprsFile = "c:/path/to/exprsData.txt"
```

(Windows users: note the use of / rather than \; this is because R treats the \ character as an "escape" sequence to change the meaning of the subsequent character.) See the help pages for `read.table` for more detail. A common variant is that the character separating columns is a comma ("comma-separated values", or "csv" files), in which case the `sep` argument might be `sep=","`.

It is always important to verify that the data you have read match your expectations. At a minimum, check the class and dimensions of `geneData` and take a peek at the first several rows.

```
> class(exprs)
[1] "matrix"
> dim(exprs)
[1] 500  26
> colnames(exprs)
 [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L" "M"
[14] "N" "O" "P" "Q" "R" "S" "T" "U" "V" "W" "X" "Y" "Z"
> head(exprs)
                    A      B     C     D     E    F
AFFX-MurIL2_at  192.7  85.75 176.8 135.6 64.49 76.4
AFFX-MurIL10_at  97.1 126.20  77.9  93.4 24.40 85.5
AFFX-MurIL4_at   45.8   8.83  33.1  28.7  5.94 28.3
AFFX-MurFAS_at   22.5   3.60  14.7  12.3 36.87 11.3
AFFX-BioB-5_at   96.8  30.44  46.1  70.9 56.17 42.7
AFFX-BioB-M_at   89.1  25.85  57.2  70.0 49.58 26.1
                    G    H    I      J     K    L    M
AFFX-MurIL2_at  160.5 66.0 56.9 135.61 63.44 78.2 83.1
AFFX-MurIL10_at  98.9 81.7 97.8  90.48 70.57 94.5 75.3
AFFX-MurIL4_at   31.0 14.8 14.2  34.49 20.35 14.2 20.6
AFFX-MurFAS_at   23.0 16.2 12.0   4.55  8.52 27.3 10.2
AFFX-BioB-5_at   86.5 30.8 19.7  46.35 39.13 41.8 80.2
AFFX-BioB-M_at   75.0 42.3 41.1  91.53 39.91 49.8 63.5
```

```
                   N    O    P     Q     R      S    T
AFFX-MurIL2_at  89.3 91.1 95.9 179.8 152.5 180.83 85.4
AFFX-MurIL10_at 68.6 87.4 84.5  87.7 108.0 134.26 91.4
AFFX-MurIL4_at  15.9 20.2 27.8  32.8  33.5  19.82 20.4
AFFX-MurFAS_at  20.2 15.8 14.3  15.9  14.7  -7.92 12.9
AFFX-BioB-5_at  36.5 36.4 35.3  58.6 114.1  93.44 22.5
AFFX-BioB-M_at  24.7 47.5 47.4  58.1 104.1 115.83 58.1
                    U     V    W      X    Y      Z
AFFX-MurIL2_at  157.99 146.8 93.9 103.86 64.4 175.62
AFFX-MurIL10_at  -8.69  85.0 79.3  71.66 64.2  78.71
AFFX-MurIL4_at   26.87  31.1 22.3  19.01 12.2  17.38
AFFX-MurFAS_at   11.92  12.8 11.1   7.56 20.0   8.97
AFFX-BioB-5_at   48.65  90.2 42.0  57.57 44.8  61.70
AFFX-BioB-M_at   73.42  64.6 40.3  41.82 46.1  49.41
```

Sample annotation

The information about the samples (e.g., experimental conditions or parameters, or attributes of the subjects such as sex, age, and diagnosis) is often referred to as covariates. The information describing the samples can be represented as a table with S rows and V columns, where V is the number of covariates. An example of such a table can be input with

```
> pDataFile = file.path(dataDirectory, "pData.txt")
> pData = read.table(pDataFile,
      row.names=1, header=TRUE, sep="\t")
> dim(pData)
[1] 26  3
> rownames(pData)
 [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L" "M"
[14] "N" "O" "P" "Q" "R" "S" "T" "U" "V" "W" "X" "Y" "Z"
> summary(pData)
    gender        type        score
 Female:11   Case   :15   Min.   :0.100
 Male  :15   Control:11   1st Qu.:0.328
                          Median :0.415
                          Mean   :0.537
                          3rd Qu.:0.765
                          Max.   :0.980
```

There are three columns of data, and 26 rows. Note that the rows of the sample data table align with the columns of the expression data matrix:

```
> all(rownames(pData) == colnames(exprs))
[1] TRUE
```

This is an essential feature of the relationship between the assay and sample data; *ExpressionSet* will complain if these names do not match.

Sample data can take on a number of different forms. For instance, some covariates might reasonably be represented as numeric values. Other covariates (e.g., gender, tissue type, or cancer status) might be better represented as `factor` objects (see the help page for `factor` for more information). It is important that the sample covariates are encoded correctly; the `colClasses` argument to `read.table` can be helpful in correctly inputting (and ignoring, if desired) columns from the file.

Exercise 2.7

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

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

c. *Use* `sapply` *to determine the classes of each column of* `pData`*. Hint: read the help page for* `sapply`*.*

d. *What is the sex and Case/Control status of the 15th and 20th samples? What is the status for the sample(s) with* `score` *greater than* 0.8*?*

Investigators often find that the meaning of simple column names does not provide enough information about the covariate. What is the cryptic name supposed to represent? In what units are the covariates measured? We can create a data frame containing such metadata (or read the information from a file using `read.table`) with

```
> metadata = data.frame(labelDescription=c("Patient gender",
      "Case/control status", "Tumor progress on XYZ scale"),
       row.names=c("gender", "type", "score"))
```

This creates a *data.frame* object with a single column called "labelDescription", and with row names identical to the column names of the *data.frame* containing the sample annotation data. The column `labelDescription` must be present; other columns are optional.

Bioconductor's **Biobase** package provides a class called *AnnotatedDataFrame* that conveniently stores and manipulates tabular data in a coordinated fashion. Create and view an *AnnotatedDataFrame* instance with:

```
> adf = new("AnnotatedDataFrame", data=pData,
      varMetadata=metadata)
> adf
An object of class "AnnotatedDataFrame"
  rowNames: A, B, ..., Z  (26 total)
```

```
  varLabels and varMetadata description:
    gender: Patient gender
    type: Case/control status
    score: Tumor progress on XYZ scale
```

Some useful operations on an *AnnotatedDataFrame* include `sampleNames`, `pData` (to extract the original `pData` *data.frame*), and `varMetadata`. In addition, *AnnotatedDataFrame* objects can be subset much as a *data.frame*:

```
> head(pData(adf))
  gender    type score
A Female Control  0.75
B   Male    Case  0.40
C   Male Control  0.73
D   Male    Case  0.42
E Female    Case  0.93
F   Male Control  0.22
> adf[c("A","Z"), "gender"]
An object of class "AnnotatedDataFrame"
  rowNames: A, Z
  varLabels and varMetadata description:
    gender: Patient gender
> pData(adf[adf$score > 0.8,])
  gender    type score
E Female    Case  0.93
G   Male    Case  0.96
X   Male Control  0.98
Y Female    Case  0.94
```

Annotations and feature data

Metadata on features are as important as metadata on samples, and can be very large and diverse. A single chip design (i.e., collection of features) is likely to be used in many different experiments, and it would be inefficient to repeatedly collect and coordinate the same metadata for each *ExpressionSet* instance. Instead, the idea is to construct specialized metadata packages for each type of chip or instrument. Many of these packages are available from the Bioconductor Web site. These packages contain information such as the gene name, symbol, and chromosomal location. There are other metadata packages that contain the information that is provided by other initiatives such as GO and KEGG. The *annotate* package provides basic data manipulation tools for the metadata packages.

The appropriate way to create annotation data for features is very straight forward: we provide a character string identifying the type of chip

used in the experiment. For instance, the data we are using are from the Affymetrix HG-U95Av2 GeneChip :

```
> annotation = "hgu95av2"
```

It is also possible to record information about features that are unique to the experiment (e.g., flagging particularly relevant features). This is done by creating or modifying an `AnnotatedDataFrame` like that for `adf` but with row names of the *AnnotatedDataFrame* matching rows of the assay data.

Experiment description

A basic description about the experiment (e.g., the investigator or lab where the experiment was done, an overall title, and other notes) can be recorded by creating a *MIAME* object. One way to create a *MIAME* object is to use the `new` function:

```
> experimentData = new("MIAME", name="Pierre Fermat",
      lab="Francis Galton Lab",
      contact="pfermat@lab.not.exist",
      title="Smoking-Cancer Experiment",
      abstract="An example ExpressionSet",
      url="www.lab.not.exist",
      other=list(notes="Created from text files"))
```

Usually, `new` takes as arguments the class name and pairs of names and values corresponding to different slots in the class; consult the help page for *MIAME* for details of available slots.

Assembling an *ExpressionSet*

An *ExpressionSet* object is created by assembling its component parts, and after all this work the final assembly is disappointingly easy:

```
> exampleSet = new("ExpressionSet", exprs=exprs,
      phenoData=adf, experimentData=experimentData,
      annotation="hgu95av2")
```

Note that the names on the right of each equal sign can refer to any object of appropriate class for the argument. See the help page for *ExpressionSet* for more information.

We created a rich data object to coordinate diverse sources of information. Less rich objects can be created by providing less information. A minimal expression set can be created with

```
> minimalSet = new("ExpressionSet", exprs=exprs)
```

Of course this object has no information about sample or feature data or about the chip used for the assay.

2.4.3 ExpressionSet basics

Now that you have an *ExpressionSet* instance, let's explore some of the basic operations. You can get an overview of the structure and available methods for *ExpressionSet* objects by reading the help page:

```
> help("ExpressionSet-class")
```

When you print an *ExpressionSet* object, a brief summary of the contents of the object is displayed (displaying the entire object would fill your screen with numbers):

```
> exampleSet
ExpressionSet (storageMode: lockedEnvironment)
assayData: 500 features, 26 samples
  element names: exprs
phenoData
  sampleNames: A, B, ..., Z  (26 total)
  varLabels and varMetadata description:
    gender: Patient gender
    type: Case/control status
    score: Tumor progress on XYZ scale
featureData
  featureNames: AFFX-MurIL2_at, AFFX-MurIL10_at, ..., 31
  739_at  (500 total)
  fvarLabels and fvarMetadata description: none
experimentData: use 'experimentData(object)'
Annotation: hgu95av2
```

Accessing data elements

A number of accessor functions are available to extract data from an *ExpressionSet* instance. You can access the columns of the sample data (an *AnnotatedDataFrame*) using `$`:

```
> exampleSet$gender[1:5]
[1] Female Male   Male   Male   Female
Levels: Female Male
> exampleSet$gender[1:5] == "Female"
[1]  TRUE FALSE FALSE FALSE  TRUE
```

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

```
> featureNames(exampleSet)[1:5]
[1] "AFFX-MurIL2_at"  "AFFX-MurIL10_at"
[3] "AFFX-MurIL4_at"  "AFFX-MurFAS_at"
[5] "AFFX-BioB-5_at"
```

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

```
> sampleNames(exampleSet)[1:5]
[1] "A" "B" "C" "D" "E"
> varLabels(exampleSet)
[1] "gender" "type"   "score"
```

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

```
> mat = exprs(exampleSet)
> dim(mat)
[1] 500  26
> adf = phenoData(exampleSet)
> adf
An object of class "AnnotatedDataFrame"
  sampleNames: A, B, ..., Z  (26 total)
  varLabels and varMetadata description:
    gender: Patient gender
    type: Case/control status
    score: Tumor progress on XYZ scale
```

Subsetting

Probably the most useful operation to perform on *ExpressionSet* objects is subsetting. Subsetting an *ExpressionSet* is very similar to subsetting the expression matrix that is contained within the *ExpressionSet*: the first argument subsets the features and the second argument subsets the samples. Here are some examples. Create a new *ExpressionSet* consisting of the five features and the first three samples:

```
> vv = exampleSet[1:5, 1:3]
> dim(vv)
Features  Samples
       5        3
> featureNames(vv)
```

```
[1] "AFFX-MurIL2_at"  "AFFX-MurIL10_at"
[3] "AFFX-MurIL4_at"  "AFFX-MurFAS_at"
[5] "AFFX-BioB-5_at"
> sampleNames(vv)
[1] "A" "B" "C"
```

Create a subset consisting of only the male samples:

```
> males = exampleSet[, exampleSet$gender == "Male"]
> males
ExpressionSet (storageMode: lockedEnvironment)
assayData: 500 features, 15 samples
  element names: exprs
phenoData
  sampleNames: B, C, ..., X  (15 total)
  varLabels and varMetadata description:
    gender: Patient gender
    type: Case/control status
    score: Tumor progress on XYZ scale
featureData
  featureNames: AFFX-MurIL2_at, AFFX-MurIL10_at, ..., 31
  739_at  (500 total)
  fvarLabels and fvarMetadata description: none
experimentData: use 'experimentData(object)'
Annotation: hgu95av2
```

2.5 Graphics

Graphics and visualization are important issues when dealing with complex data such as the ones typically found in biological science. In this section we work through some examples that allow us to create general plots in R. Both R and Bioconductor offer a range of functions that generate various graphical representations of our 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. (2005a) 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 dataset on a log–log scale.

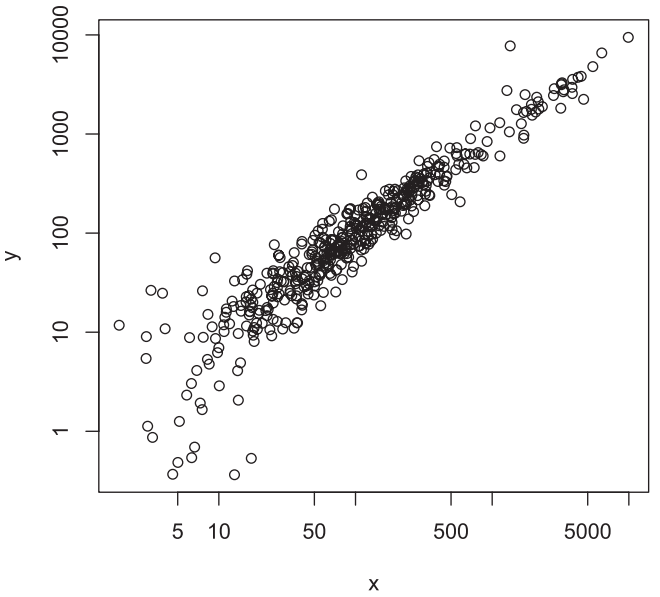

Figure 2.1. Scatterplot of expression intensities for two samples.

```
> x = exprs(exampleSet[, 1])
> y = exprs(exampleSet[, 3])
> plot(x=x, y=y, log="xy")
```

From the plot in Figure 2.1 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 degree diagonal.

Exercise 2.8
The axis annotation of the plot in Figure 2.1 is not very informative. Can you add more meaningful axis labels and a title to the plot? Can you change the plotting symbols? Add the 45 degrees diagonal to the plot. [Hint: use function `abline`*.]*

Proper visualization can help 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 Affymetrix HG-U95Av2 GeneChip. The `basecontent` function 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")
> bases = basecontent(hgu95av2probe$sequence)
```

We now need to match the probes via their position on the array to positions in the data matrix of the `CLLbatch` object. Because we have several samples in the set, we use the `rowMeans` function to compute, for each probe, the average of its expression values across arrays.

```
> iab = with(hgu95av2probe, xy2indices(x, y,
      cdf="hgu95av2cdf"))
> probedata = data.frame(
      int=rowMeans(log2(exprs(CLLbatch)[iab, ])),
      gc=bases[, "C"] + bases[, "G"])
```

Now we are ready to plot the $\log_2$-transformed intensity values of the probes grouped by their GC-content. An extraordinarily effective tool for the visualization of distributions is the boxplot. In the code below, we construct a set of boxplots, one box for each possible value of GC-content. The result is shown in Figure 2.2.

```
> colorfunction = colorRampPalette(brewer.pal(9, "GnBu"))
> mycolors = colorfunction(length(unique(probedata$gc)))
> label = expression(log[2]~intensity)
> boxplot(int ~ gc, data=probedata, col=mycolors,
      outline=FALSE, xlab="Number of G and C",
      ylab=label, main="")
```

The first two lines of the above code chunk are concerned with creating a set of colors for the boxes. We use the function `brewer.pal` from the package **RColorBrewer** to obtain a set of basic colors and `colorRampPalette` to interpolate between them, defining a unique color for each possible value of `gc`.

An alternative visualization of univariate distributions is the density plot. We can plot multiple densities in one panel using the function `multidensity` from the package **geneplotter**. For this plot we focus on the ten biggest groups. First, we find out which of the GC-content values occur most often.

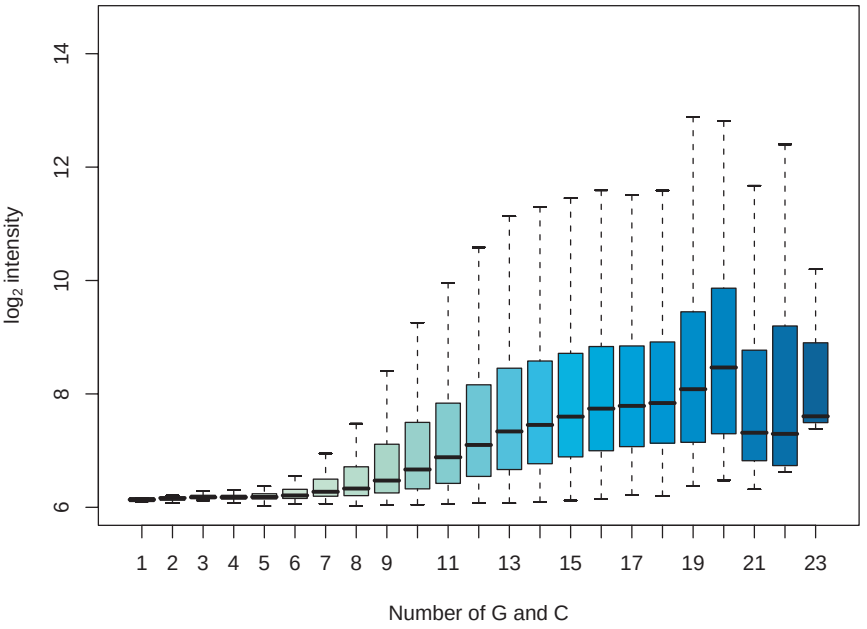

Figure 2.2. Boxplots of the distributions of $\log_2$-intensities from the CLL dataset grouped by GC-content. Please refer to the manual pages of the functions `boxplot.stats` and `boxplot` for details on the features shown in a boxplot.

```
> tab = table(probedata$gc)
> gcUse = as.integer(names(sort(tab, decreasing=TRUE)[1:10]))
> gcUse
 [1] 13 11 12 10 14  9 15  8 16 17
```

Then we can call the plot function.

```
> library("geneplotter")
> multidensity(int ~ gc, data=subset(probedata,
      gc %in% gcUse), xlim=c(6, 11),
      col=colorfunction(12)[-(1:2)],
      lwd=2, main="", xlab=label)
```

Exercise 2.9
Another useful distribution summary is plots of the empirical cumulative distribution function. Create a plot similar to the one in Figure 2.3 using the function `multiecdf`.

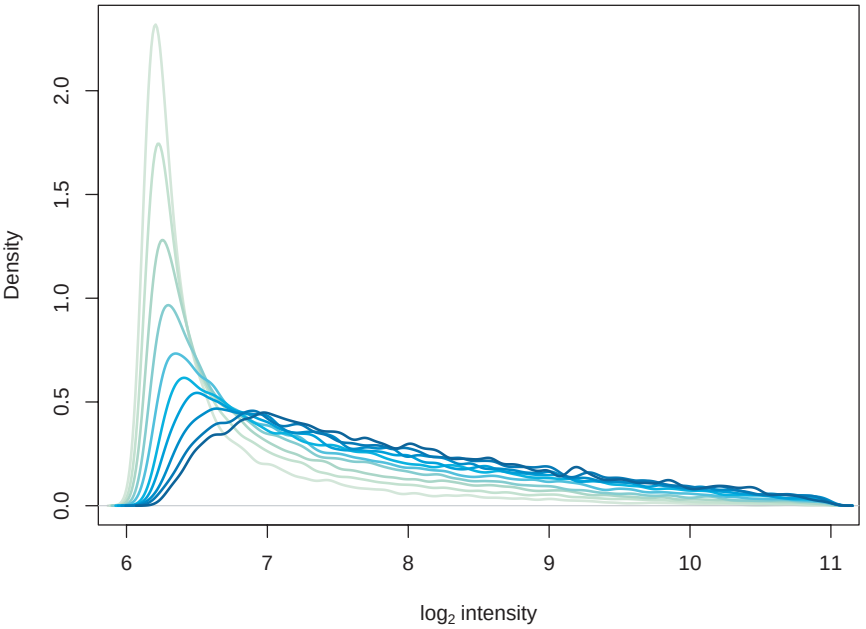

Figure 2.3. Density plot of distributions of $\log_2$-intensities from the CLL dataset grouped by GC content.