oblems

Fit a random forest to the Virco data to determine which protease mu-ons are most highly associated with SQV fold resistance as measured by `fold`.

Apply logic regression to the Virco data to characterize, with a logic cture, the association between protease mutations and SQV fold resistance easured by `SQV.fold`.

Apply Monte Carlo logic regression to the FAMuSS data to characterize mportance of SNPs in the `actn3` and `resistin` genes in predicting change on-dominant arm muscle strength as measured by `NDRM.CH`.

Apply a multivariable adaptive regression spline to address the question oblem 7.3. Compare and contrast your findings.

Appendix

R Basics

The purpose of this appendix is to introduce the fundamental concepts of programming in R that will provide sufficient fluency for the reader to understand the examples presented throughout this text. Specifically, we offer an overview of importing and manipulating data, installing and using R packages, and writing and applying basic functions. The novice reader is encouraged to refer to a number of comprehensive texts for a more thorough introduction to working and programming in R. See for example Gentleman (2008), Venables and Smith (2008), Spector (2008) and Dalgaard (2002).

A.1 Getting started

R is a free, open-source statistical computing language and environment available for Windows, Mac OS X, Linux and Unix and licensed under the terms of the GNU (GNU is Not Unix) General Public License (Free Software Foundation, 2007). R is available on the Comprehensive R Archive Network (CRAN) website (http://cran.r-project.org/), and installation is straightforward. We recommend downloading the precompiled binary distribution that is suitable to your operating system and then following the corresponding instructions.

Java GUI for R (JGR –pronounced "jaguar") is a graphical user interface (GUI) that is also freely available for download. It is distributed in binary form for Windows and Mac OS X 10.4.4 (and above) users at http://jgr.markushelbig.org/Download.html and can be compiled from source code for other platforms. JGR is not necessary to get started with using R but may be preferable to users more familiar with Windows-based applications. JGR has the primary advantage over other available GUIs of being platform independent. More information on the features and unique attributes of JGR can be found in Helbig *et al.* (2005).

Command line

Upon opening R (or JGR), you will see an R console that lists the version number and other pertinent information, including the current working directory, as shown below:

```
R version 2.7.1 (2008-06-23)
Copyright (C) 2008 The R Foundation for Statistical Computing
ISBN 3-900051-07-0

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

  Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

[Workspace restored from /Users/foulkes/.RData]

>
```

Commands can be typed directly at the prompt, indicated by the > symbol, or saved in an ascii file using any text editor. Since R is an interpreter language, each line of code is processed as it is entered. This feature is similar to alternative statistical programming languages such as STATA (http://www.stata.com/) and distinguishes R from compiler languages such as C/C++ and Fortran. Notably, this allows us to transition seamlessly in R between sourcing text files and typing commands directly at the prompt.

As a user, you have the choice of writing commands directly at the prompt or storing them in text files that can be sourced. In general, we strongly encourage the reader to maintain good programming practices by writing and saving code in organized, well-documented files. This is essential for generating reproducible findings, an ethical imperative in medical research studies involving human subjects. Consider, for example, the canonical "Hello World" program that simply prints the words "hello world". One approach in R is to simply type the command

```
> print("hello world")
```

directly at the prompt, which leads to the following output:

```
[1] "hello world"
```


Alternatively, we can create a text file entitled `hello_world.r`, within which each line represents a command to be executed. For example, this program might contain the text `print("hello world")`. Any lines beginning with the # symbol within this file are treated as comments and ignored in the R execution. Suppose this file is saved in the directory `~/Projects/ASG/`. Using the `source()` function, we call up this file at the R prompt by submitting

```
> source("~/Projects/ASG/hello_world.r")
```

which results in the same output as before:

```
[1] "hello world"
```

It is also straightforward to run this program at a Unix or DOS prompt using the following command line within the directory in which `hello_world.r` is stored:

```
$ R CMD BATCH hello_world.r hello_world.out
```

This results in the creation of a new file entitled `hello_world.out` that contains the R output.

Working directory

At the beginning of an R session, it is important to identify and indicate the directory in which you want your data to be stored. This will provide for easy access to these files during subsequent sessions and facilitate working on multiple projects. It is helpful to use different directories for each project since creating new objects of the same name will overwrite existing objects. Once a directory is specified, the data will be stored in a file entitled `.RData`. Suppose for example that we want our working directory to be `~/Projects/ASG/Examples/`. First we check the current working directory using the `getwd()` function as follows:

```
> getwd()

[1] "/Users/foulkes"
```

To change the current working directory, we apply the `setwd()` function:

```
> setwd("~/Projects/ASG/Examples/")
```

Any new objects (variables, dataframes, etc.) that you create will be saved in the workspace entitled `.RData` within the current working directory. The next time you begin an R session, this workspace can be loaded using the `load()` function as follows:

```
> load("~/Projects/ASG/Examples/.RData")
```

A.2 Types of data objects

There are several different types of objects that you may be creating or manipulating during an R session, including vectors, factors, matrices, lists and dataframes. Here we briefly describe the characteristics of each with simple demonstrations of how they can be created and viewed. In Section A.3, we describe approaches to reading in data from an existing file.

Assignment operator

A fundamental component of programming in R is the *assignment* operator. Suppose, for example, that we want to create a new variable named `NewVar` that is equal to 1. We do this using the `<-` symbol, as demonstrated in the following code:

```
> NewVar <- 1
```

Note that the variable name is on the left-hand side of the assignment arrow and the value that we want to assign to this variable name is on the right-hand side. This ordering is not necessary in R but is conventional and recommended. Alternatively, the = symbol can be used for assignment; however, this is highly discouraged. The resulting object can be printed using the `print()` function as

```
> print(NewVar)

[1] 1
```

or more simply

```
> NewVar

[1] 1
```

Object class

Every object has an associated *class* that defines how functions are applied to it. Two simple classes are `numeric` and `character`. For example, the variable `NewVar` that we just defined is a `numeric` variable, which we can see by applying the `class()` function:

```
> class(NewVar)

[1] "numeric"
```

A `character` variable is created using quotation marks, as shown below:

```
> CharVar <- "Gene1"
> class(CharVar)

[1] "character"
```


Other object classes include `factors`, `matrices` and `data.frames`, each of which is described below. Throughout this text, we create objects of many different classes, ranging from linear models generated using the `lm()` function in Chapter 2 to trees generated using the `rpart()` function in Chapter 6. It is important to keep in mind that the result of applying standard R functions, such as `print()` and `plot()`, will depend on the class associated with the argument to these functions. For example, in Example 6.2, we use the `plot()` function to plot the results of a classification tree, shown in Figure 6.2. We use the same `plot()` function in Example 7.5 to plot the results of applying logic regression, which yields the very different illustration in Figure 7.3.

Vectors

A *vector* is defined simply as a string of objects and can be generated using the `c()` function in R. For example, we can create the vector $\mathbf{y} = (1, 2, 3)$ using the command

```
> y <- c(1,2,3)
> y

[1] 1 2 3
```

The dimension of the resulting vector is unspecified, taking on the value 1×3 or 3×1 depending on the specific application, as demonstrated below. The `c()` function can take as its input both numeric and character objects, though if both are passed to it simultaneously, they will be coerced to be of the same type. For example, the command

```
> c(1,2,3,"gene")
```

yields

```
[1] "1"    "2"    "3"    "gene"
```

In the context of genetic association studies, the trait under investigation is often represented as a vector. For example, suppose we are interested in determining the genetic contributors to an abnormal lipid profile. In this case, the trait might be total cholesterol level and is represented by the $n \times 1$ vector $\mathbf{y}$, where n is the number of individuals in our sample. Each element of this vector is the cholesterol level for the corresponding individual in our sample.

Factors

A factor object is a type of vector with an associated attribute that describes the levels of the variable and can be created using the R function `factor()`. For example, suppose we have a vector of genotypes given by $\mathbf{x}_1^T$ =(AA, AT, TT, TT, AT, AT, AA, TT), with each element corresponding to the genotype of one individual in our sample of $n = 8$ observations. A detailed description of genotype data is given in Section 1.3. We can generate a factor object corresponding to these data using the following code:

```
> x1 <- factor(c("AA", "AT", "TT", "TT", "AT", "AT", "AA", "TT"))
> x1

[1] AA AT TT TT AT AT AA TT
Levels: AA AT TT
```

We see from the output that there are three levels to this variable, given by AA, AT and TT. Notably, use of factor objects as predictor variables in a regression setting must proceed with caution since interpretation of associated coefficients is not always straightforward. For example, we typically code gender as a numeric vector with elements equal to 1 for males and 0 for females. If gender is converted to a factor variable, then it is treated by the `lm()` function in R as taking on the values 0.5 and -0.5 in the regression equation. Calculating predicted responses for males and females based on the output of this function requires consideration of this alternative coding.

Matrices

A matrix is a two-dimensional array of objects and can be generated in R in multiple ways. Consider for example the matrix of genotype data given by $\mathbf{X} = [\mathbf{x}_1, \mathbf{x}_2]$, where $\mathbf{x}_1$ is defined above and $\mathbf{x}_2^T$ =(GG, GG, GC, CC, GG, CC, CC, GC). Each row of $\mathbf{X}$ represents an individual and each column represents a site on the genome. This matrix can be created in R using the `matrix()` function as follows:

```
> X <- matrix(c("AA", "AT", "TT", "TT", "AT", "AT", "AA", "TT",
    "GG", "GG", "GC", "CC", "GG", "CC", "CC", "GC"), nrow=8)
> X

     [,1] [,2]
[1,] "AA" "GG"
[2,] "AT" "GG"
[3,] "TT" "GC"
[4,] "TT" "CC"
[5,] "AT" "GG"
[6,] "AT" "CC"
[7,] "AA" "CC"
[8,] "TT" "GC"
```

As we see above, the input to the `matrix()` function is a vector containing the elements of the matrix, beginning in the top left corner, going down the first column and then beginning again at the top of the next column. We also specify `nrow=8` to indicate the number of rows in the matrix. Alternatively, we can generate the vectors $\mathbf{x}_1$ and $\mathbf{x}_2$ and use the `cbind()` function, as follows:

```
> x1 <- c("AA", "AT", "TT", "TT", "AT", "AT", "AA", "TT")
> x2 <- c("GG", "GG", "GC", "CC", "GG", "CC", "CC", "GC")
> X <- cbind(x1, x2)
> X
```



```
     x1   x2
[1,] "AA" "GG"
[2,] "AT" "GG"
[3,] "TT" "GC"
[4,] "TT" "CC"
[5,] "AT" "GG"
[6,] "AT" "CC"
[7,] "AA" "CC"
[8,] "TT" "GC"
```

The characteristics of our resulting matrix can be printed using the `attributes()` function. For example, for our genotype matrix $\mathbf{X}$, we have

```
> attributes(X)

$dim
[1] 8 2

$dimnames
$dimnames[[1]]
NULL

$dimnames[[2]]
[1] "x1" "x2"
```

This output tells us that the dimension of $\mathbf{X}$ is 8×2 (eight rows and two columns) and the names of the two columns are given by `x1` and `x2`, respectively. The names of the rows are not specified, and hence the `NULL` value is returned for this parameter. Each attribute can be printed separately by simply typing `attributes(X)$` followed by the name of the attribute. For example, the dimension of $\mathbf{X}$ is given by

```
> attributes(X)$dim

[1] 8 2
```

Lists

Another useful object type in R is a list. In fact, the `dimnames` attribute of $\mathbf{X}$ in the previous example is stored as a list. A list can contain multiple objects of different types, including vectors, matrices, and lists. For example, using the `list()` function, we can generate a list that contains the vector `y` and the matrix $\mathbf{X}$ that we just created above:

```
> list(trait=y, genotypes=X)

$trait
[1] 1 2 3

$genotypes
```

```
     X1   X2
[1,] "AA" "GG"
[2,] "AT" "GG"
[3,] "TT" "GC"
[4,] "TT" "CC"
[5,] "AT" "GG"
[6,] "AT" "CC"
[7,] "AA" "CC"
[8,] "TT" "GC"
```

Dataframes

Dataframes are similar to matrices, with the notable exception that they can have columns of different variable types, including numeric, character and factor variables. This characteristic makes dataframes extremely useful in the analysis of population-level data in which both continuous and categorical variables need to be stored for analysis. As a result, dataframes are used extensively throughout this text.

A.3 Importing data

All of the examples in this text are based on data that are stored in tab- or comma-delimited ascii text files. In general, data can be exported from other programs into this format and then read into R. For example, if your data are currently in a Microsoft Excel spreadsheet, simply open this file, go to the `File` menu and select `Save As`. Under the `Format` tab, choose `CSV (comma delimited)` or `Text (tab delimited)`. Click `Save`, and the resulting file can now be imported into R.

In Section 1.3.3 of this text, we illustrate how to use the `read.delim()` and `read.csv()` functions in R to import tab- and comma-delimited files, respectively. For example, consider the tab-delimited text file entitled "FMS_data.txt". We import the data into R using the following commands:

```
> fmsURL <- "http://people.umass.edu/foulkes/asg/data/FMS_data.txt"
> fms <- read.delim(file=fmsURL, header=T, sep="\t")
```

The default settings for the `read.delim()` function are `header=T`, which specifies that the first row of the text file consists of variable names, and `sep="\t"`, which indicates the data are tab delimited. Alternatively, we can specify `header=F`, which assumes the first line of the data file is the first patient record. If the variables are separated by columns, we specify `sep=","`, and if they are separated by one or more spaces, we use `sep=""`. Alternatively, if the data are stored in a SAS export file, the `sasxport.get()` function of the `Hmisc` package can be used to read the data into R. A discussion of how to install R packages is given below. See documentation for the `Hmisc` package on the CRAN website for more details on using the `sasxport.get()` function and linking to associated macros.

A.4 Managing data

R is a powerful and versatile tool for data management. Here we introduce some useful commands that allow basic data manipulations. Consider first the following dataframe, consisting of identification numbers, genotypes, genders and disease statuses of five subjects:

```
> SampleDat <- data.frame(ID = c(1,2,3,4,5),
+   SNP=c("AA", "AT", "TT", "TT", "AA"),
+   Gender=c("Female", "Male", "Female", "Female", "Male"),
+   DiseaseStatus=c(1, 1, 0, 0, 0))
> SampleDat

  ID SNP Gender DiseaseStatus
1  1  AA Female             1
2  2  AT   Male             1
3  3  TT Female             0
4  4  TT Female             0
5  5  AA   Male             0
```

We can print data on each variable of the resulting dataframe by using the $ symbol as follows:

```
> SampleDat$ID

[1] 1 2 3 4 5

> SampleDat$SNP

[1] AA AT TT TT AA
Levels: AA AT TT
```

Alternatively, we can first apply the `attach()` function to our dataframe

```
> attach(SampleDat)
```

which allows us to use the shorthand notation

```
> ID

[1] 1 2 3 4 5

> SNP

[1] AA AT TT TT AA
Levels: AA AT TT
```

The names of all of the variables in our dataframe can be listed using the `names()` function:

```
> names(SampleDat)

[1] "ID"            "SNP"           "Gender"        "DiseaseStatus"
```

Now suppose we want to determine the number of individuals and the number of variables in our dataset. We can do this using the `dim()` function

```
> dim(SampleDat)

[1] 5 4
```

which tells us that we have five individuals (rows) and four variables (columns). Tabulating the number of males and females is also straightforward using the `table()` function:

```
> table(SampleDat$Gender)

Female   Male
     3      2
```

Here we see that the `table()` function results in a vector with an element for each level of the input variable. Specifically, this output tells us that there are three females and two males in the dataset. The frequency of males and females in our sample is calculated using any of the equivalent commands

```
> table(SampleDat$Gender)/5
> table(SampleDat$Gender)/dim(SampleDat)[1]
> table(SampleDat$Gender)/sum(table(SampleDat$Gender))
```

which all yield the following output:

```
Female   Male
   0.6    0.4
```

Several useful aspects of R were applied in the code above. First we note that we applied division to every element of the vector given by `table()` by simply dividing by a scalar, in this case 5. We also see that we can pull out an element of a vector using `[ ]` with a corresponding index. Recall that applying the `dim()` function yielded a vector with the first element equal to the number of rows and the second element equal to the number of columns. By specifying `[1]`, we are pulling out the first element of this vector, which is again the number 5. We revisit this below. Finally, by applying the function `sum()` to our table vector, we get the sum of the elements of this vector, given in this case by $3 + 2 = 5$.

We see from the results above that the gender variable is coded here as a character variable. We may want to create a new variable for gender that is coded as a numeric variable, which we can do using the `as.numeric()` function:

```
> GenderNum <- as.numeric(SampleDat$Gender)
> GenderNum

[1] 1 2 1 1 2
```


If we want males to be coded as 1 and females to be coded as 0, we simply subtract 1 from this vector. Again, we can subtract a scalar from a vector and it will apply to all elements of the vector, as shown below:

```
> GenderNum-1

[1] 0 1 0 0 1
```

Subsetting a dataframe can be a useful tool and can be achieved similar to the example above of pulling out elements of a vector. For example, suppose we want to print out the first row of our data. We can do this by specifying

```
> SampleDat[1,]

  ID SNP Gender DiseaseStatus
1  1  AA Female             1
```

Note that the number before the comma within the brackets indicates the row number. A number after the comma indicates the column number. For example, we can print the third column using

```
> SampleDat[,3]

[1] Female Male   Female Female Male
Levels: Female Male
```

Multiple rows and columns can also be printed by replacing the scalar with a sequence of numbers. For example, we can print the second and fourth columns by using the following command:

```
> SampleDat[,c(2,4)]

  SNP DiseaseStatus
1  AA             1
2  AT             1
3  TT             0
4  TT             0
5  AA             0
```

Alternatively, we can indicate one or more rows to be printed by specifying the level of a variable. For example, suppose we want to print the records for females. This is achieved as follows:

```
> SampleDat[SampleDat$Gender=="Female",]

  ID SNP Gender DiseaseStatus
1  1  AA Female             1
3  3  TT Female             0
4  4  TT Female             0
```

Subsetting our data is especially useful if we aim to do a stratified analysis. For example, suppose we want to tabulate genotypes for those individuals with disease and those without disease. One approach is to tabulate the data for the two disease statuses separately. Using the `table()` function in R, we have

```
> table(SampleDat $SNP[SampleDat$DiseaseStatus==1])

AA AT TT
 1  1  0

> table(SampleDat $SNP[SampleDat$DiseaseStatus==0])

AA AT TT
 1  0  2
```

Alternatively, we can use the `tapply()` function to calculate the table means simultaneously:

```
> tapply(SampleDat$SNP, SampleDat$DiseaseStatus, table)

$`0`

AA AT TT
 1  0  2

$`1`

AA AT TT
 1  1  0
```

The result is a list with the number of elements equal to the number of levels of `DiseaseStatus` and each element representing the result of applying the `table()` function to the corresponding subset of the data.

A.5 Installing packages

Throughout this text, we use several R packages that contain useful functions for the analysis of data arising from genetic association studies. Many of these are not included in a standard download and need to be downloaded and installed, which can be done using the `install.package()` function. For example, suppose we want to use a function within the `genetics` package. We simply type

```
> install.packages("genetics")
```

Once a prompt appears, select a CRAN mirror in a location near you to download the package and the package will be installed automatically. To access the functions within the package, we need to use the `library()` function at the start of each new R session. For example, to load the `genetics` package, we write

```
> library(genetics)
```


Bioconductor is a development project that distributes several R packages targeted for the analysis of genomic data. Additional information on the aims and scope of this project as well as its practical applications can be found in Gentleman *et al.* (2004) and Gentleman *et al.* (2005). Initially this project focused on applications in DNA microarray studies, but it has recently expanded to include useful tools for the analysis of SNP data. To install Bioconductor packages, simply type

```
> source("http://bioconductor.org/biocLite.R")
> biocLite()
```

This installation can take several minutes. Additional packages can be installed by specifying the names of the packages within the `biocLite()` function

```
> source("http://bioconductor.org/biocLite.R")
> biocLite(c("pckg1","pckg2"))
```

where `pckg1` and `pckg2` are the names of packages. Additional information about Bioconductor can be found at http://www.bioconductor.org/.

A.6 Additional help

The CRAN website's FAQs are an excellent source of information for questions about R. In addition, you can get help on how to use existing commands using the `help()` function. For example, if you want more information on the `read.table()` function, you can enter

```
> help(read.table)
```

The associated help file includes a general description of the function and how to use it, details on the arguments to be entered in the function call and the value(s) returned by it. In addition, most documentation includes at least a simple example illustrating application of the function. Finally, if you do not know the name of a function, you can use the `help.search()` command to search the existing documentation for character strings that match your input. For example, typing

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

will yield a list of functions and associated packages that include the term variance in their documentation.