1 Introduction

1.1 Why Use R for Econometrics?

- R/S platform is a de facto standard among professional statisticians.
- R is available for the Windows, Mac, and Linux OS.
- In addition to providing statistical operations, R is a general-purpose programming language.
- R incorporates features of object-oriented and functional programming languages.
- R is open source software. It is free and easy to get help from the user community.
- R is compatible with other programming languages, such as Python and C++.

Here I show a code that simulates random variables from a bivariate normal distribution. Using microbenchmark reveals that the code written in C++ is roughly 60 times faster than the code written in R. However, with the help of Rcpp, you can easily speed up your code by utilizing little knowledge of C++.

```r
> require(Rcpp)
> require(inline)
> cppFunction(code='
+ List bvnorm_cpp(double mu1, double mu2,
+ double sig1, double sig2, double rho, int sim){
+ NumericVector x(sim);
+ NumericVector y(sim);
+ x[0]=0;
+ y[0]=0;
+ for(int i=1; i<sim; ++i){
+ x[i]=mu2+rho*(sig2/sig1)*(y[i-1]-mu1)
+ +sqrt(pow(sig2,2)*(1-pow(rho,2)))*rnorm(1)[0];
+ y[i]=mu1+rho*(sig1/sig2)*(x[i-1]-mu2)
+ +sqrt(pow(sig1,2)*(1-pow(rho,2)))*rnorm(1)[0];
+ }
')
```
• It will be easier for you to find a job in the future.

\[ \text{Job Trends from Indeed.com} \]

- R statistics
- SAS statistics
- SPSS statistics
- MATLAB statistics
- STATA statistics

Figure 1: Demand for Statistical Language (2013/08/21)
1.2 Good Programming Habits (Suggestions from Ben Skrainka\textsuperscript{[3]})

A good code is

- Easy to maintain
- Easy to extend
- Easy to understand ... even after a six month break!
- Straight-forward and direct ... no side-effects or surprises!
- Reads like English (or some other human language)

Use a coding convention: A good coding convention makes your code read like a good story and makes your intent clear:

- Naming of functions, variables, and filenames: Function names should begin or end with a verb. Separate each word by CamelCase, ‘_’, or ‘.’. e.g. CalcValueFunc.
- Grouping and layout of code such as braces
- Modification history
- Comments: one variable definition per line and comments. Write why you did something rather than what you did.
- Respect the local coding convention when working on code

1.3 Getting Help

Use \texttt{help()} function.

To get information on the \texttt{rnorm()} function:

\begin{verbatim}
> help(rnorm)
\end{verbatim}

\texttt{example(rnorm)} will runs the example shown in the help entries.

You can also use a google-style search:

\begin{verbatim}
> help.search("GHK")
\end{verbatim}

Or learn about a package:

\begin{verbatim}
> help(package=MASS)
\end{verbatim}

1.4 Packages

An advantage of using R is that R has thousands of user-written packages available. You can install a package using the command \texttt{install.packages()} and load a packages by \texttt{library()} or \texttt{require()}.

\begin{verbatim}
> install.packages("MASS")
> library(MASS)
> require(MASS)
\end{verbatim}
2 Some Basics of R

2.1 Basic R Operators

Table 1: Basic R Operators

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>x + y</td>
<td>Addition</td>
</tr>
<tr>
<td>x - y</td>
<td>Subtraction</td>
</tr>
<tr>
<td>x * y</td>
<td>Multiplication</td>
</tr>
<tr>
<td>x / y</td>
<td>Division</td>
</tr>
<tr>
<td>x ^ y</td>
<td>Exponentiation</td>
</tr>
<tr>
<td>x %% y</td>
<td>Modular arithmetic</td>
</tr>
<tr>
<td>x %/% y</td>
<td>Integer division</td>
</tr>
<tr>
<td>x == y</td>
<td>Test for equality</td>
</tr>
<tr>
<td>x &lt;= y</td>
<td>Test for less than or equal to</td>
</tr>
<tr>
<td>x &gt;= y</td>
<td>Test for greater or equal to</td>
</tr>
<tr>
<td>x &amp;&amp; y</td>
<td>Boolean AND for scalars</td>
</tr>
<tr>
<td>x</td>
<td></td>
</tr>
<tr>
<td>x &amp; y</td>
<td>Boolean AND for vectors</td>
</tr>
<tr>
<td>x</td>
<td>y</td>
</tr>
<tr>
<td>!x</td>
<td>Boolean negation</td>
</tr>
</tbody>
</table>

2.2 Types of Objects

R stores the data structure in the following ways:

2.2.1 Vectors

A numeric vector can be assigned:

```r
> x<-c(1,1.5,0.75,0.875)
> x
[1] 1.000 1.500 0.750 0.875
```

You can pick an element from `x`:

```r
> x
[1] 1.000 1.500 0.750 0.875
> x[3]
[1] 0.75
```

Notice that R uses a recycle rule to do vector arithmetic:
> x<-c(1,1.5,0.75,0.875)
> y<-c(1,2)
> x+y+1
[1] 3.000 4.500 2.750 3.875

R only produces a warning when the longer object is not a multiple of short object length. However, it will still produce an output.

> z<-c(1,2,3)
> x+z+1
[1] 3.000 4.500 4.750 2.875
Warning message:
In x + z :
longer object length is not a multiple of shorter object length

Vector Indexing:
> length(x) # find the size of x
[1] 4
> x[c(1,3)] # extract element 1 and 3 of x
[1] 1.00 0.75
> x[-2] # remove element 2 from x
[1] 1.000 0.750 0.875

Generating a regular sequence:
> seq(from=0,to=12, by=3)
[1] 0 3 6 9 12
> seq(from=0,to=12, length=5)
[1] 0 3 6 9 12
> rep(0,3)
[1] 0 0 0
> 1:12
[1] 1 2 3 4 5 6 7 8 9 10 11 12

One of the easiest way to achieve speed in R is to use vectorized operations: a function applied to a vector is actually applied individually to each element.

> sqrt(1:12)
[1] 1.000000 1.414214 1.732051 2.000000 2.236068 2.449490 2.645751
> round(sqrt(1:12),digits=2)
[1] 1.00 1.41 1.73 2.00 2.24 2.45 2.65 2.83 3.00 3.16 3.32 3.46

You can also generate a logical vector:

> x
[1] 1.000 1.500 0.750 0.875
> x>1
[1] FALSE TRUE FALSE FALSE
> x[x>1] # filter out the element greater than 1
[1] 1.5
A vectorized “if-then-else” function:

```r
> ifelse(x>1,x,0)  # if x greater than 1, then show x; otherwise replace with 0
[1] 0.0 1.5 0.0 0.0
```

Finally, sometimes you would like to name the element of your vector:

```r
> names(x)=c("Amy","Martha","Donna","Rose")
> x
Amy Martha Donna Rose
1.000 1.500 0.750 0.875
```

### 2.2.2 Matrices and Arrays

There are three ways to generate a matrix:

```r
> x<-matrix(1:9,nrow=3,ncol=3)
> x
[,1] [,2] [,3]
[1,] 1 4 7
[2,] 2 5 8
[3,] 3 6 9
```

Or you can firstly create an empty matrix and then fill in its elements. This approach comes in handy when you would like to store the output of a loop in a matrix:

```r
> mat=matrix(ncol=3,nrow=3)
> mat[1,]=c(1,4,7)  # fill in the elements by row
> mat[2,]=c(2,5,8)
> mat[3,]=c(3,6,9)
> mat
[,1] [,2] [,3]
[1,] 1 4 7
[2,] 2 5 8
[3,] 3 6 9
```

```r
> dim(mat)  # the dimensionality of mat
[1] 3 3
```

Or you can create a matrix by combining the row vectors or column vectors:

```r
> x<-cbind(c(1,2,3),c(3,5,7))
> x
[,1] [,2]
[1,] 1 3
[2,] 2 5
[3,] 3 7
```

```r
> y<-rbind(c(1,2,3),c(3,5,7))
> y
[,1] [,2] [,3]
[1,] 1 2 3
[2,] 3 5 7
```
Matrix indexing:

> x[2,3]
[1] 8
> x[2,] # the second row of x
[1] 2 5 8
> x[,] # the second column of x
[1] 4 5 6

But notice that the dimension reduction may produce bugs.

> y<-x[,2]
> y
[1] 4 5 6
> is.matrix(y)
[1] FALSE
> is.vector(y)
[1] TRUE

When you reduce the dimension of a matrix, R stores it as a vector instead of a matrix.

Some basic matrix algebra:

> x<-matrix(c(7,15,10,22),2,2)
> x+x # matrix addition

[,1] [,2]
[1,] 14 20
[2,] 30 44

> x%*%x # matrix multiplication

[,1] [,2]
[1,] 199 290
[2,] 435 634

> 2*x # matrix multiplication by a scalar

[,1] [,2]
[1,] 14 20
[2,] 30 44

> t(x) # matrix transpose

[,1] [,2]
[1,] 7 15
[2,] 10 22

> solve(x) # inverse of a matrix

[,1] [,2]
[1,] 5.50 -2.50
[2,] -3.75 1.75

apply function: apply function is a wrapper of the loop that allows you to apply a function to either row- or column- dimension of a matrix.

> mat<-matrix(1:9,3,3)
> mat
   [,1] [,2] [,3]
[1,]  1  4  7
[2,]  2  5  8
[3,]  3  6  9
> apply(mat,2,median)
[1] 2 5 8
> apply(mat,1,median)
[1] 4 5 6

Notice that if you would like to calculate row means or column means, it is more efficient to use base functions, like `rowMeans` and `colMeans`.

> x<-matrix(rnorm(1000^2),1000,1000) # create a 1000 by 1000 matrix
> system.time(apply(x,1,mean))
    user  system elapsed
   0.065  0.001   0.065
> system.time(rowMeans(x))
    user  system elapsed
   0.004  0.000   0.004
> system.time(apply(x,2,mean))
    user  system elapsed
   0.057  0.001   0.058
> system.time(colMeans(x))
    user  system elapsed
   0.007  0.000   0.008

2.2.3 Lists

In contrast to a vector, in which all elements must be of the same mode, the list structure can combine objects of different types.

> j<-list(name="TARDIS",price=1e15,inside.bigger=T)
> j

$name
[1] "TARDIS"

$price
[1] 1e+15

$inside.bigger
[1] TRUE

You can also create an empty list and then fill in its components:

> j<-vector(3,mode="list")
> j

[[1]]
> j[[1]]="TARDIS"
> j[[2]]=1e15
> j[[3]]=T
> j
[[1]]
[1] "TARDIS"

[[2]]
[1] 1e+15

[[3]]
[1] TRUE

> names(j)=c("name","price","inside.bigger") # name the components
> j
$name
[1] "TARDIS"

$price
[1] 1e+15

$inside.bigger
[1] TRUE

You can access a list component in several ways:

> j$name
[1] "TARDIS"
> j[["price"]]
[1] 1e+15
> j[[3]]
[1] TRUE

To apply a function to a list, you can use `lapply` or `sapply`:

> l=list(1:12,5:8,3:9)
> l
[[1]]
[1] 1 2 3 4 5 6 7 8 9 10 11 12
> lapply(l,mean)  
[[1]]  
[1] 6.5  
[[2]]  
[1] 6.5  
[[3]]  
[1] 6  

> sapply(l,sd)  
[1] 3.605551 1.290994 2.160247  

Notice that \texttt{lapply} returns a list and \texttt{sapply} returns a vector.

\subsection*{2.2.4 Dataframes}

A data frame is like a matrix. However, each column of a data frame can have different mode.

> kids<-c("Jack","Jill")  
> age<-c(12,10)  
> data<-data.frame(kids,age)  
> data  
   kids age  
1  Jack 12  
2  Jill 10  

Take a subset of the data:

> subset(data,age>10)  
   kids age  
1  Jack 12  
> subset(data,kids="Jill")  
   kids age  
2  Jill 10  

\subsection*{2.2.5 Factors}

Factor mode provides a handy way to handle categorical data in R.
```r
> x<-c(0,1,1,0)
> x
[1] 0 1 1 0
> y<-factor(x)
> y
[1] 0 1 1 0
Levels: 0 1
> y<-factor(x,levels=c(0,1),labels=c("female","male"))
> y
[1] female male male female
Levels: female male
Notice that the core of y is no longer (0,1) but rather (1,2):

> unclass(y)
[1] 1 2 2 1
attr("levels")
[1] "female" "male"

Common functions that are used with factors are tapply, split, and by.

2.3 Basic Functions for Statistical Distributions

The basic functions for distributions in R are named in the form: rxxx (random variable generator), dxxx (pdf), pxxx (cdf), and qxxx (inverse cdf). I provide a list of basic distributions:

<table>
<thead>
<tr>
<th>Distribution</th>
<th>Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>beta</td>
<td>dbeta</td>
</tr>
<tr>
<td>binomial</td>
<td>dbinom</td>
</tr>
<tr>
<td>Cauchy</td>
<td>dcauchy</td>
</tr>
<tr>
<td>chi-squared</td>
<td>dchisq</td>
</tr>
<tr>
<td>exponential</td>
<td>dexp</td>
</tr>
<tr>
<td>F</td>
<td>df</td>
</tr>
<tr>
<td>gamma</td>
<td>dgamma</td>
</tr>
<tr>
<td>geometric</td>
<td>dgeom</td>
</tr>
<tr>
<td>hypergeometric</td>
<td>dhyper</td>
</tr>
<tr>
<td>log-normal</td>
<td>dlnorm</td>
</tr>
<tr>
<td>multinomial</td>
<td>dmultinom</td>
</tr>
<tr>
<td>negative binomial</td>
<td>dnbinom</td>
</tr>
<tr>
<td>normal</td>
<td>dnorm</td>
</tr>
<tr>
<td>Poisson</td>
<td>dpois</td>
</tr>
<tr>
<td>Student’s t</td>
<td>dt</td>
</tr>
<tr>
<td>uniform</td>
<td>dunif</td>
</tr>
<tr>
<td>Weibull</td>
<td>dweibull</td>
</tr>
</tbody>
</table>
```

11
2.4 Basic Graphing in R

Personally, I only use \texttt{plot} and \texttt{hist} for basic graphs most of the time.

The \texttt{plot} function forms the foundation for much of R’s base graphing operations. \texttt{plot} function is a typical example of polymorphism of R. You can apply \texttt{plot()} to many different types of objects and get many different types of graphs for each. These generic functions greatly decrease the number of the functions to remember.

We can call \texttt{plot()} with an x-vector and an y-vector. You may use the options \texttt{xlab} and \texttt{ylab} to label the axes. \texttt{main} specifies the title of the graph. \texttt{xlim} and \texttt{ylim} specify the range of the axes.

\begin{verbatim}
> N=1000 # generate a fake dataset
> x=rnorm(N)
> u=rnorm(N)
> y=1+0.5*x+u
> plot(x,y,xlab="x",ylab="y")
\end{verbatim}

You can use \texttt{abline} function to add lines to the graph:

\begin{verbatim}
mylm=lm(y~x)
abline(mylm)
\end{verbatim}
You can also plot a density graph using `plot` and use `text(x-coordinate, y-coordinate, "text")` to put texts on the graphs.

```r
x1 = rbeta(100, 2, 2)
x2 = rbeta(100, 2, 3)
d1 = density(x1, from=0, to=1)
d2 = density(x2, from=0, to=1)
plot(d1, main="Beta Distributions", xlab="", ylim=c(0, 3), lty=2)
lines(d2)
text(0.2, 2, "beta(2,2)")
text(0.8, 1.5, "beta(2,3)")
```
I use `hist` to plot histograms. You can specify the bin size through `breaks` option and whether the y-axis is density or frequency through `freq`. The default setting is to use Sturges’ formula to calculate the bin size and report frequency on the y-axis.

```r
> x1 = rbeta(100, 2, 2)
> hist(x1)
```
As mentioned before, you can also apply `plot` to other classes of objects, such as `lm`. Doing so, you will obtain a sequence of plots that help you evaluate your linear model.
For more advanced graphs, you can use other packages. `lattice` package is good at creating three-dimensional plots. `ggplot2` produces a wide variety of beautiful graphs. With a single line like this, you will be able to visualize data in an elegant way.

```r
ggplot(data,aes(break_angle,start_speed))
  +geom_point(aes(alpha=as.factor(pitch_type),shape=as.factor(pitch_type)))
  +scale_shape_discrete(name="Pitch Types")
  +scale_alpha_discrete(name="Pitch Types")
  +facet_wrap(~pitcher,ncol=1)
  +theme_bw()
  +labs(title="Yu Darvish v.s. Max Sherzer (2013)", x="Break Angle", y="Start Speed")
```

Figure 2: Apply `plot()` to `lm()`
You can compare the quality of the default histograms produced by `plot` function and the `ggplot` package:
If you are interested in learning how to use ggplot2, you can find a detailed documentation at [http://docs.ggplot2.org/current/](http://docs.ggplot2.org/current/).

2.5 How to Write an R Function

A function in R is an object that has input arguments and returns output arguments:

```
> f <- function(input)
>   ...
>   return (output)
> }
```

A function that generate a Fibonacci sequence. The input argument is the length of the sequence you would like to generate. The output argument is the sequence.

```
> GenFiboSeq <- function(length)
+   out <- numeric(length) # define an empty vector to store the output
+   out[1] <- 0 # initialize x1
+   out[2] <- 1 # initialize x2
+   idx <- 3 # define a counter
+   while (idx <= length){
+     out[idx] <- out[idx-1] + out[idx-2] # generate the sequence
+     idx <- idx + 1
+   }
+   return(out)
> }
```
> GenFiboSeq(10)
[1]  0  1  1  2  3  5  8 13 21 34

2.6 How to Write a Loop

To write a loop, you can use one of the following structures:

> for (i in x){
+   
+ }

It means that there will be one iteration of the loop for each component of the vector x with i taking on the values of those components.

> x<-c(3, 5, 7)
> for (i in x){
+   print(i^2)
+ }
[1]  9
[1] 25
[1] 49

You can also use while and repeat.

> while(i<=7){
+   print(i^2)
+   i<-i+2
+ }
[1]  9
[1] 25
[1] 49

> repeat {
+   print(i^2)
+   i<-i+2
+   if(i>7){
+     break
+   }
+ }
[1]  9
[1] 25
[1] 49

2.7 Optimization Routines

Even though R has thousands of useful packages, you may inevitably encounter a situation where you have to write our model and estimate it using some optimization routine. For one-dimension problem, you can use optimize, which implements a combination of golden section search and successive parabolic interpolation.
Suppose we would like to maximize the function:

\[ f(x) = \frac{\log(1 + \log(x))}{\log(1 + x)} \]

Firstly, we define the function \( f \):

```r
f=function(x){
    val=log(1+log(x))/log(1+x)
    return(val)
}
```

Then we use `optimize` to search for the maximizer. Notice that (1) you have to specify the interval, in which `optimize` is applied to, and (2) the default is set to minimize the objective function so be sure you set the option `maximum` to `TRUE`.

```r
> optimize(f,c(4,8), maximum=T)
$maximum
[1] 4.00006
$objective
[1] 0.5404008
```

For higher-dimension optimization problem, you can use `optim`. It implements Nelder-Mead algorithm, quasi-Newton algorithm, conjugate gradient algorithm, box-constrained BFGS algorithm, and simulated annealing. For details, please read the documentation by typing `help(optima)` in R.

An application that you will definitely encounter is MLE. Suppose that \( x \) are generated from Gamma(\( a \), \( b \)) distribution. We would like to estimate the parameters \( a \) and \( b \). We know the pdf of gamma(a,b) is

\[ f(x) = \frac{b^a}{\Gamma(a)} x^{a-1} \exp(-bx) \]

Then we derive the log-likelihood function:

\[ L(a, b; x_1, \cdots, x_N) = N \cdot a \cdot \log(b) - N \cdot \log(\Gamma(b)) + (a - 1) \sum_{i=1}^{N} \log x_i - b \sum_{i=1}^{N} x_i \]

To estimate the MLE, we firstly define the objective function

```r
log.l=function(x,theta){
    a=theta[1]
    b=theta[2]
    N=length(x)
    val=N*a*log(b)-N*lgamma(a)+(a-1)*sum(log(x))-b*sum(x)
    return(-1*val)
}
```

Notice that we multiply `val` by -1 to turn the maximization problem into a minimization problem. Also, we define all the parameters in one sequence, `theta`.

Let the data generated process be Gamma(1,1)
set.seed(85719)
a=1
b=1
N=1000
x=rgamma(N,shape=1,rate=1)

Then we can estimate the model using optim

> optim(c(2,2), log.l, x=x)
$par
[1] 0.9713958 1.0333209

$value
[1] 937.8111

$counts
  function gradient     63     NA

$convergence
[1] 0

$message
NULL

An important option of optim is control. control is a list of parameters. In particular, in order to gain more precision, you may want to set the maximum number of iteration, maxit, to a larger number and the relative convergence tolerance, reltol to a smaller number.

2.8 How to Load Datasets

To load a dataset from .txt, .csv, or other delimited format, you can use read.table, read.csv, or read.delim. You can make adjustments by specifying the options: header=T to specify the data format has a header, sep="," to specify the data format is separated by commas, and col.names=c(...) or row.names=c(...) to specify the row names and column names.

For instance,

read.csv(tf, fill = TRUE, header = FALSE,
        col.names = paste("V", seq_len(ncol), sep = ",")

To load datasets from other formats, you have to install foreign package. It provides functions that allow you to load STATA binary files (read.dta), SPSS data file (read.spss), SAS XPORT format library (read.xport), etc..

3 Linear Regression

To run an OLS regression in R, you can use the function lm:
lm(formula, data, subset, weights, na.action, ...) 

In formula, you specify the model. data is optional. You can specify which dataset in the environment you would like to use. subset provides you an option to use only a subset of observations in the regression, say households with two children. weights allows you to weight the regression. You use na.action to tell R how to handle missing values. There are more options in lm. Please see the documentation by typing help(lm).

An Example:

```r
> ## Annette Dobson (1990) "An Introduction to Generalized Linear Models".
> ## Page 9: Plant Weight Data.
> ctl <- c(4.17,5.58,5.18,6.11,4.50,4.61,5.17,4.53,5.14)
> trt <- c(4.81,4.17,4.41,3.59,5.87,3.83,6.03,4.89,4.32,4.69)
> group <- gl(2, 10, 20, labels = c("Ctl","Trt"))
> weight <- c(ctl, trt)
> lm.D9 <- lm(weight ~ group)
> lm.D90 <- lm(weight ~ group - 1) # omitting intercept
```

After you run the regression, you can use summary function to see the results:

```r
> summary(lm.D9)

Call:
  lm(formula = weight ~ group)

Residuals:
    Min     1Q Median     3Q    Max
-1.0710 -0.4938  0.0685  0.2462  1.3690

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept)  5.03200    0.22020  22.850  9.55e-15 ***
groupTrt   -0.37100    0.31140  -1.191   0.249

---
Signif. codes:  0 *** 0.001 ** 0.01 * 0.05 . 0.1 1

Residual standard error: 0.6964 on 18 degrees of freedom
Multiple R-squared:  0.0731, Adjusted R-squared:  0.02158
F-statistic: 1.419 on 1 and 18 DF,  p-value: 0.249
```

```r
> summary(lm.D90)

Call:
  lm(formula = weight ~ group - 1)

Residuals:
    Min     1Q Median     3Q    Max
     22
```

22
Coefficients:

|                | Estimate | Std. Error | t value | Pr(>|t|) |
|----------------|----------|------------|---------|----------|
| groupCtl       | 5.0320   | 0.2202     | 22.85   | 9.55e-15 *** |
| groupTrt       | 4.6610   | 0.2202     | 21.16   | 3.62e-14 *** |

---

Signif. codes: 0 *** 0.001 ** 0.01 * 0.05 . 0.1 1

Residual standard error: 0.6964 on 18 degrees of freedom
Multiple R-squared: 0.9818, Adjusted R-squared: 0.9798
F-statistic: 485.1 on 2 and 18 DF, p-value: < 2.2e-16

After running the regressions, you can use `predict` to calculate the fitted values or to extrapolate the model to new data points. You can also use `residuals` to calculate the residuals.

```r
> predict(lm.D9,type="response")
       1       2       3       4       5       6       7       8       9      10      11      12
5.032   5.032   5.032   5.032   5.032   5.032   5.032   5.032   5.032   5.032   4.661   4.661
      13      14      15      16      17      18      19      20
> residuals(lm.D9)
       1       2       3       4       5       6       7       8       9      10
-0.862   0.548   0.148   1.078  -0.532  -0.422   0.138  -0.502   0.298   0.108
     11     12     13     14     15     16     17     18     19     20
 0.149 -0.491 -0.251 -1.071  1.209 -0.831  1.369  0.229 -0.341  0.029
```

The `summary` function in R only provides standard errors under homoskedasticity. In order to obtain the White robust standard errors or the cluster standard errors, you can use the package `sandwich`. The main function you would like to use is `vcovHC` in the package. It calculates the heteroskedasticity-consistent estimation of the covariance matrix of the coefficient estimates in regression models. An input is the `lm` object. You can specify the type of variance-covariance matrix as an option. The function returns the variance-covariance matrix instead of the standard errors.

To obtain the standard errors, you take the diagonal terms of the variance-covariance matrix returned from `vcovHC` and take the squared root of it.

```r
> sqrt(diag(vcovHC(lm.D9,type="const"))) # Homoskedasticity
   (Intercept) groupTrt
0.2202177 0.3114349
> summary(lm.D9) # Check whether the output is the same as summary.lm

Call:
  lm(formula = weight ~ group)

Residuals:
       Min      1Q  Median      3Q     Max
-1.0710 -0.4938  0.0685  0.2462  1.3690
```
Coefficients:

| Estimate | Std. Error | t value | Pr(>|t|) |
|----------|------------|---------|----------|
| (Intercept) | 5.0320 | 0.2202 | 22.850 | 9.55e-15 *** |
| groupTrt | -0.3710 | 0.3114 | -1.191 | 0.249 |

---

Signif. codes: 0 *** 0.001 ** 0.01 * 0.05 . 0.1 1

Residual standard error: 0.6964 on 18 degrees of freedom
Multiple R-squared: 0.07308, Adjusted R-squared: 0.02158
F-statistic: 1.419 on 1 and 18 DF, p-value: 0.249

> sqrt(diag(vcovHC(lm.D9,type="HC"))) # White robust standard errors

<table>
<thead>
<tr>
<th>Intercept</th>
<th>groupTrt</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1749274</td>
<td>0.2954530</td>
</tr>
</tbody>
</table>

Aside, to estimate panel data models, it is convenient to use the package plm.

4 Binary Choice Models

To estimate Probit or Logit models, you can use the generalized linear model function glm. The function has a similar context to lm function. An important additional option is family. The option allows you to choose the underlying error distribution and the link function in the model. To estimate a Logit model, you specify `family = binomial(link = "logit")`. To estimate a Probit model, you specify `family = binomial(link = "probit")`. As a side note, you can also estimate a Poisson regression using `family = poisson()`.

> N=1000 # generate a fake dataset
> x1=rnorm(N)
> u=rnorm(N)
> y0=1+0.5*x1+u
> y=as.numeric(y0>0)

> myprobit=glm(y~x1,family=binomial(link="probit")) #ESTIMATE PROBIT
> summary(myprobit)

Call:
`glm(formula = y ~ x1, family = binomial(link = "probit"))`

Deviance Residuals:

<table>
<thead>
<tr>
<th>Min</th>
<th>1Q</th>
<th>Median</th>
<th>3Q</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>-2.7205</td>
<td>0.2792</td>
<td>0.4875</td>
<td>0.6798</td>
<td>1.3945</td>
</tr>
</tbody>
</table>

Coefficients:

| Estimate | Std. Error | z value | Pr(>|z|) |
|----------|------------|---------|----------|
| (Intercept) | 5.0320 | 0.2202 | 22.850 | 9.55e-15 *** |
| groupTrt | -0.3710 | 0.3114 | -1.191 | 0.249 |
(Intercept)  0.98961  0.05149  19.220  <2e-16  ***
x1         0.50113  0.05148   9.735  <2e-16  ***
---
Signif. codes:  0 ***  0.001 **  0.01 *  0.05 .  0.1  1

(Dispersion parameter for binomial family taken to be 1)

Null deviance: 972.45 on 999 degrees of freedom
Residual deviance: 864.45 on 998 degrees of freedom
AIC: 868.45

Number of Fisher Scoring iterations: 5

> mylogit=glm(y~x1,family=binomial(link="logit")) #ESTIMATE LOGIT
> summary(mylogit)

Call:
glm(formula = y ~ x1, family = binomial(link = "logit"))

Deviance Residuals:
     Min       1Q   Median       3Q      Max
-2.6205   0.3027   0.4848   0.6688   1.4425

Coefficients:
                      Estimate Std. Error   z value Pr(>|z|)
(Intercept)         1.68364    0.09655  17.4376   <2e-16 ***
x1                 0.88223    0.09338   9.4470   <2e-16 ***
---
Signif. codes:  0 ***  0.001 **  0.01 *  0.05 .  0.1  1

(Dispersion parameter for binomial family taken to be 1)

Null deviance: 972.45  on 999  degrees of freedom
Residual deviance: 864.89  on 998  degrees of freedom
AIC: 868.89

Number of Fisher Scoring iterations: 5

After estimating the models, you can use the generic functions predict and residuals to obtain the fitted values and residuals, respectively.

References