DSA 6100: Statistical Methods for Data Science and Analytics

Winter 2019

Lab Session 1

1. Installation of R and RStudio

We first introduce some simple R commands. The best way to learn a new language is to try out the commands. R can be downloaded from http://cran.r-project.org/.

1.1. To Install R

a. Open an internet browser and go to www.r-project.org.

b. Click the "download R" link in the middle of the page under "Getting Started."

c. Select a CRAN location (a mirror site) and click the corresponding link.

d. Click on the "Download R for Windows" link at the top of the page.

e. Click on the "base" subdirectories containing the latest version of R

f. Save the .exe file, double-click it to open, and follow the installation instructions.

g. Now that R is installed, you need to download and install RStudio.

1.2. To Install RStudio

a. Go to www.rstudio.com and click on the "Download RStudio" button.

b. Click on "Download RStudio Desktop."

c. Click on the version recommended for your system, save the .exe file on your computer, double-click it to open to install.

2. Basic R functions

2.1. Vector creation and manipulation

In order to create a vector of numbers, we use the function c() (for *concatenate*). Any numbers inside the parentheses are joined together. The following command instructs R to join together the numbers 1, 3, 2, and 5, and to save them as a *vector* named x. When we type x, it gives us back the vector.

```
> x <- c(1,3,2,5)
> x
[1] 1 3 2 5
>y = c(1,4,3)
```

We can tell R to add two sets of numbers together. It will then add the first number from x to the first number from y, and so on. However, x and y should be the same length. We can check their length using the length() function.

```
> length (x)
[1] 3
> length (y)
[1] 3
> x+y
[1] 2 10 5
```

2.2 Random Vector

The rnorm() function generates a vector of random normal variables, with first argument n the sample size. Each time we call this function, we will get a different answer. Here we create two correlated sets of numbers, x and y, and use the cor() function to compute the correlation between them.

```
> x=rnorm (50)
> y=x+rnorm (50, mean=50, sd=.1)
> cor(x,y)
[1] 0.995
```

By default, rnorm() creates standard normal random variables with a mean of 0 and a standard deviation of 1. However, the mean and standard deviation can be altered using the mean and sd arguments, as illustrated above. Sometimes we want our code to reproduce the exact same set of random numbers; we can use the set.seed() function to do this. The set.seed() function takes an (arbitrary) integer argument.

```
> set.seed (1303)
> rnorm (50)
[1] -1.1440 1.3421 2.1854 0.5364 0.0632 0.5022 -0.0004
```

We use set.seed() throughout the labs whenever we perform calculations involving random quantities. In general, this should allow the user to reproduce our results. The mean() and var() functions can be used to compute the mean and variance of a vector of numbers. Applying sqrt() to the output of var() will give the standard deviation. Or we can simply use the sd() function.

sd()

```
> set.seed (3)
> y=rnorm (100)
> mean(y)
[1] 0.0110
> var(y)
[1] 0.7329
> sqrt(var(y))
[1] 0.8561
> sd(y)
[1] 0.8561
```

2.3 Matrix Creation and manipulation

To create matrix in R we can use matrix() function. The function takes a number of inputs, but for now we focus on the first three: the data (the entries in the matrix), the number of rows, and the number of columns. First, we create a simple matrix.

```
> x=matrix (data=c(1,2,3,4) , nrow=2, ncol =2)
> x
[,1] [,2]
[1,] 1 3
[2,] 2 4
```

Note that we could just as well omit typing data=, nrow=, and ncol= in the matrix() command above: that is, we could just type

```
> x=matrix (c(1,2,3,4) ,2,2)
```

and this would have the same effect. However, it can sometimes be useful to specify the names of the arguments passed in, as by default R creates matrices by successively filling in columns. Alternatively, the byrow=TRUE option can be used to populate the matrix in order of the rows.

```
> matrix (c(1,2,3,4) ,2,2,byrow =TRUE)
[,1] [,2]
[1,] 1 2
[2,] 3 4
```

The sqrt() function returns the square root of each element of a vector or matrix. The command x^2 raises each element of x to the power 2; any powers are possible, including fractional or negative powers.

```
> sqrt(x)
[,1] [,2]
[1,] 1.00 1.73
[2,] 1.41 2.00
> x^2
[,1] [,2]
[1,] 1 9
[2,] 4 16
```

2.4 Indexing Data

We often wish to examine part of a set of data. Suppose that our data is stored in the matrix A.

```
> A=matrix (1:16 ,4 ,4)
> A
[,1] [,2] [,3] [,4]
[1,] 1 5 9 13
[2,] 2 6 10 14
[3,] 3 7 11 15
[4,] 4 8 12 16
```

Then, typing

```
> A[2,3]
[1] 10
```

will select the element corresponding to the second row and the third column. The first number after the open-bracket symbol [always refers to the row, and the second number always refers to the column. We can also select multiple rows and columns at a time, by providing vectors as the indices.

```
> A[c(1,3) ,c(2,4) ]
[,1] [,2]
[1,] 5 13
[2,] 7 15
> A[1:3 ,2:4]
[,1] [,2] [,3]
[1,] 5 9 13
[2,] 6 10 14
[3,] 7 11 15
> A[1:2 ,]
[,1] [,2] [,3] [,4]
[1,] 1 5 9 13
[2,] 2 6 10 14
> A[ ,1:2]
[,1] [,2]
[1,] 1 5
[2,] 2 6
 [3,] 3 7
[4,] 4 8
```

The last two examples include either no index for the columns or no index for the rows. These indicate that R should include all columns or all rows, respectively. R treats a single row or column of a matrix as a vector.

```
> A[1,]
[1] 1 5 9 13
```

The use of a negative sign - in the index tells R to keep all rows or columns except those indicated in the index.

```
> A[-c(1,3) ,]
[,1] [,2] [,3] [,4]
[1,] 2 6 10 14
[2,] 4 8 12 16
> A[-c(1,3) ,-c(1,3,4)]
[1] 6 8
```

The dim() function outputs the number of rows followed by the number of columns of a given matrix.

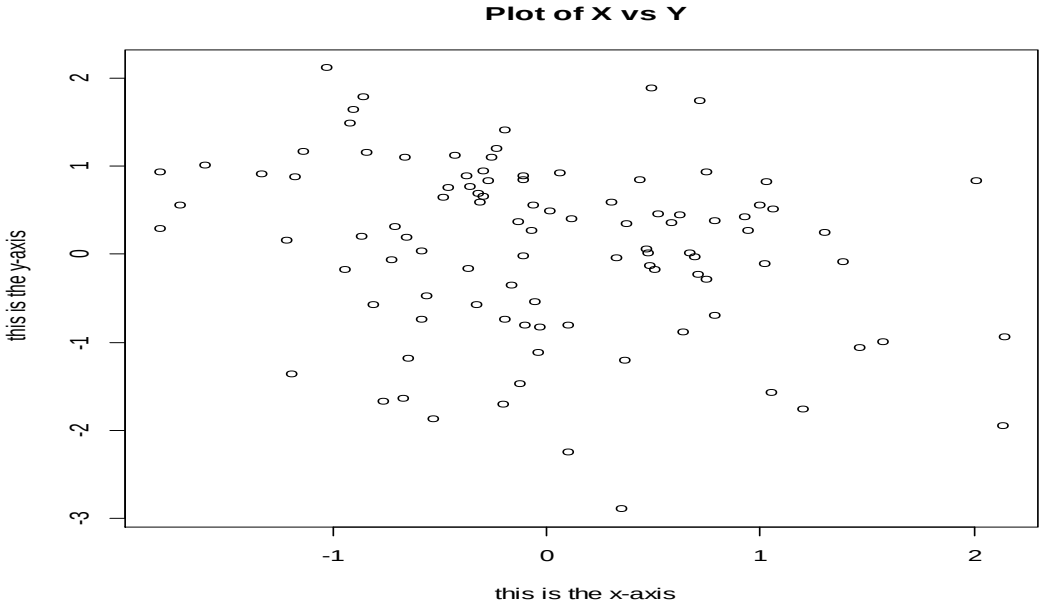
```
> dim(A)
[1] 4 4
```

2.5 Graphic

The plot() function is the primary way to plot data in R. For instance, plot(x,y) produces a scatterplot of the numbers in x versus the numbers in y. There are many additional options that can

be passed in to the plot() function. For example, passing in the argument xlab will result in a label on the *x*-axis. To find out more information about the plot() function, type ?plot.

```
> x=rnorm (100)
> y=rnorm (100)
> plot(x,y)
> plot(x,y,xlab=" this is the x-axis",ylab=" this is the y-axis", main=" Plot of X vs Y")
```



2.6 Loading Data

For most analyses, the first step involves importing a data set into R. The read.table() function is one of the primary ways to do this. The help file read.table() contains details about how to use this function. We can use the function write.table() to export data. Before attempting to load a data set, we must make sure that R knows table() to search for the data in the proper directory. For example on a Windows system one could select the directory using the Change dir. option under the File menu. However, the details of how to do this depend on the operating system (e.g. Windows, Mac, Unix). We begin by loading in the Auto data set. This data is part of the ISLR library. The following command will load the Auto.data file (download from Canvas) into R and store it as an object called Auto, in a format referred to as a data frame. Once the data has been loaded, the fix() function can be used to view it in a spreadsheet like window. However, the window must be closed before further R commands can be entered.

```
> Auto=read.table ("Auto.data ")
> fix(Auto)
```

Note that Auto.data is simply a text file, which you could alternatively open on your computer using a standard text editor. It is often a good idea to view a data set using a text editor or other software such as Excel before loading it into R. This particular data set has not been loaded correctly, because R has assumed that the variable names are part of the data and so has included them in the first row. The data set also includes a number of missing observations, indicated by a question mark ?. Missing values are a common occurrence in real data sets. Using the option header=T (or header=TRUE) in the read.table() function tells R that the first line of the file contains the variable names, and using the option na.strings tells R that any time it sees a particular character

or set of characters (such as a question mark), it should be treated as a missing element of the data matrix.

```
> Auto=read.table ("Auto.data", header =T,na.strings ="?")
> fix(Auto)
```

Excel is a common-format data storage program. An easy way to load such data into R is to save it as a csv (comma separated value) file and then use the read.csv() function to load it in.

```
> Auto=read.csv (" Auto.csv", header =T,na.strings ="?")
> fix(Auto)
> dim(Auto)
[1] 397 9
> Auto [1:4 ,]
```

The dim() function tells us that the data has 397 observations, or rows, and nine variables, or columns. There are various ways to deal with the missing data. In this case, only five of the rows contain missing observations, and so we choose to use the na.omit() function to simply remove these rows.

```
> Auto=na.omit(Auto)
> dim(Auto)
[1] 392 9
```

Once the data are loaded correctly, we can use names() to check the variable names.

```
> names(Auto)
[1] "mpg " "cylinders " " displacement" "horsepower "
[5] "weight " " acceleration" "year" "origin "
[9] "name"
```

The pairs() function creates a scatterplot matrix i.e. a scatterplot for every pair of variables for any given data set. We can also produce scatterplots for just a subset of the variables.

```
> pairs(Auto)
> pairs(~ mpg + displacement + horsepower + weight + acceleration , Auto)
```

In conjunction with the plot() function, identify() provides a useful interactive method for identifying the value for a particular variable for points on a plot. We pass in three arguments to identify(): the x-axis variable, the y-axis variable, and the variable whose values we would like to see printed for each point. Then clicking on a given point in the plot will cause R to print the value of the variable of interest. Right-clicking on the plot will exit the identify() function (control-click on a Mac). The numbers printed under the identify() function correspond to the rows for the selected points.

```
> plot(Auto$horsepower ,Auto$mpg)
> identify (Auto$horsepower ,Auto$mpg ,Auto$name)
```

The summary() function produces a numerical summary of each variable in a particular data set.

```
> summary (Auto)
```

For qualitative variables such as name, R will list the number of observations that fall in each category. We can also produce a summary of just a single variable.

3. Linear regression

We first install, and load the MASS package, which is a very large collection of data sets and functions. We also install and load the ISLR package, which includes the data sets associated with this book.

```
> install.packages ("MASS")
> install.packages ("ISLR")
> library (MASS)
> library (ISLR)
```

The MASS library contains the Boston data set, which records medv (median house value) for 506 neighborhoods around Boston. We will seek to predict medv using 13 predictors such as rm (average number of rooms per house), age (average age of houses), and lstat (percent of households with low socioeconomic status).

```
> fix(Boston )
> names(Boston )
[1] "crim" "zn" "indus" "chas" "nox" "rm" "age"
[8] "dis" "rad" "tax" "ptratio " "black" "lstat" "medv"
```

To find out more about the data set, we can type ?Boston. We will start by using the lm() function to fit a simple linear regression model, with medv as the response and lstat as the predictor. The basic syntax is lm(y∼x,data), where y is the response, x is the predictor, and data is the data set in which these two variables are kept.

```
> lm.fit =lm(medv∼lstat ,data=Boston )
> attach (Boston )
> lm.fit =lm(medv∼lstat)
```

If we type lm.fit, some basic information about the model is output. For more detailed information, we use summary(lm.fit). This gives us p-values and standard errors for the coefficients, as well as the *R*2 statistic and F-statistic for the model.

```
> lm.fit
```

We can use the names() function in order to find out what other pieces of information are stored in lm.fit. Although we can extract these quantities by name—e.g. lm.fit$coefficients—it is safer to use the extractor functions like coef() to access them.

```
> names(lm.fit )
[1] " coefficients" "residuals " "effects "
[4] "rank" "fitted .values " "assign "
[7] "qr" "df.residual " "xlevels "
[10] "call" "terms" "model"
> coef(lm.fit)
(Intercept ) lstat
34.55 -0.95
```

In order to obtain a confidence interval for the coefficient estimates, we can use the confint() command.

```
> confint (lm.fit)
2.5 % 97.5 %
(Intercept ) 33.45 35.659
lstat -1.03 -0.874
```

The predict() function can be used to produce confidence intervals and prediction intervals for the prediction of medv for a given value of lstat.

```
> predict (lm.fit ,data.frame(lstat=c(5 ,10 ,15) ), interval =" confidence ")
fit lwr upr
1 29.80 29.01 30.60
2 25.05 24.47 25.63
3 20.30 19.73 20.87
```

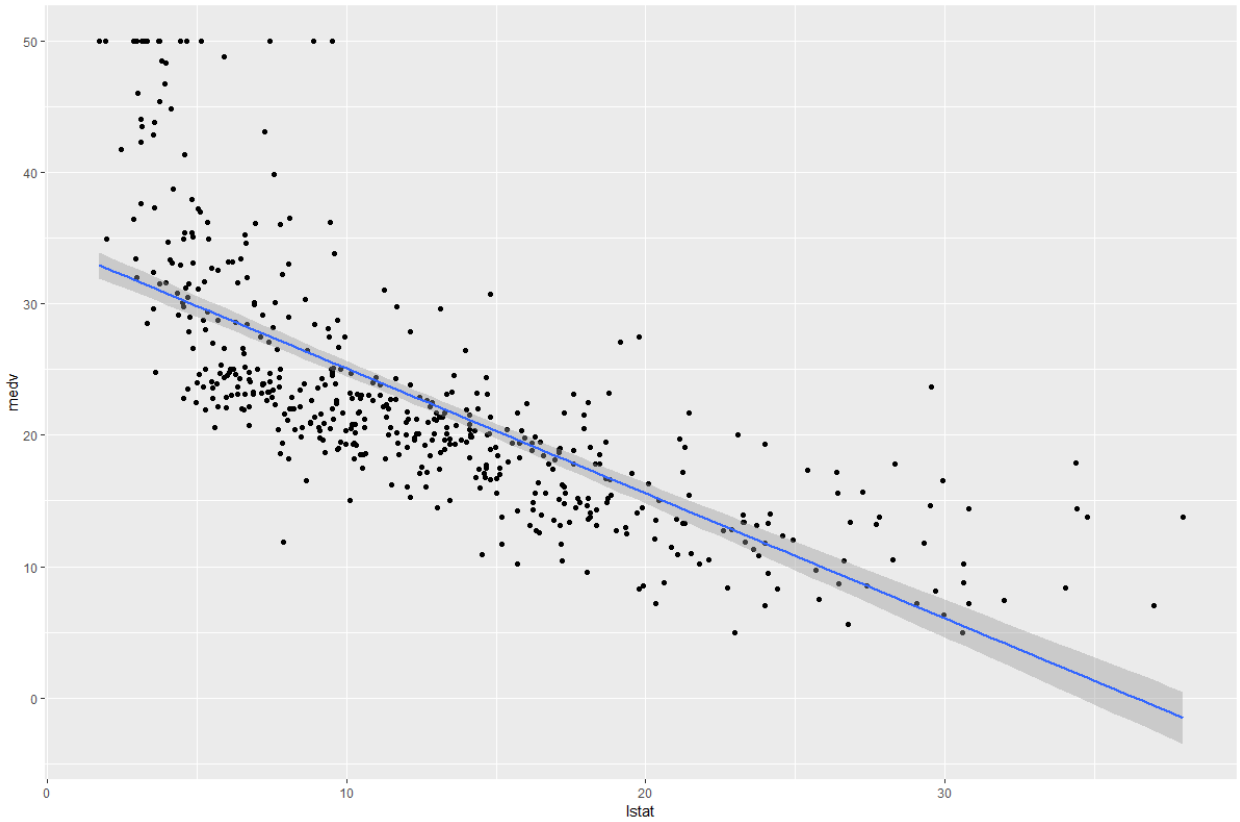
For instance, the 95% confidence interval associated with a lstat value of 10 is (24.47, 25.63), and the 95% prediction interval is (12.828, 37.28).

We will now plot medv and lstat along with the least squares regression line using the plot() and abline() functions.

```
> plot(lstat ,medv)
> abline (lm.fit)
```

To visualize the linear regression model, we can use geom_smooth():

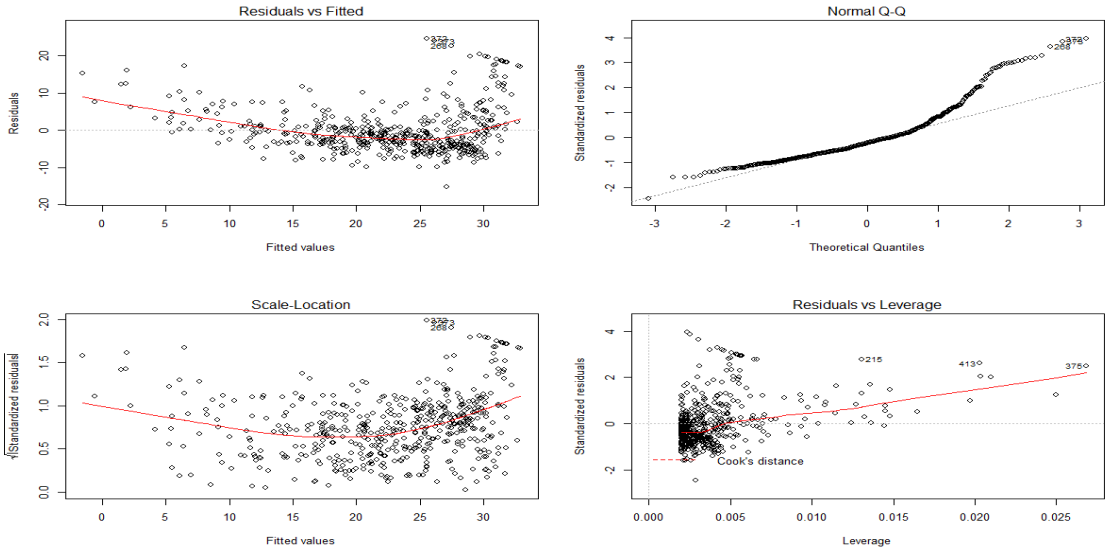
```
>ggplot(data = Boston, mapping = aes(x = lstat, y = medv)) + geom_point() +
geom_smooth(method = "lm")
```



Furthermore, Four diagnostic plots are automatically produced by applying the plot() function directly to the output from lm(). In general, this command will produce one plot at a time, and hitting *Enter* will generate the next plot. However, it is often convenient to view all four plots together. We can achieve this by using the par() function, which tells R to split the display screen

into separate panels so that multiple plots can be viewed simultaneously. For example, par(mfrow=c(2,2)) divides the plotting region into a 2 × 2 grid of panels.

```
> par(mfrow =c(2,2))
> plot(lm.fit)
```



In order to fit a multiple linear regression model using least squares, we again use the lm() function. The syntax lm(y∼x1+x2+x3) is used to fit a model with three predictors, x1, x2, and x3. The summary() function now outputs the regression coefficients for all the predictors.

```
> lm.fit =lm(medv∼lstat+age ,data=Boston )
> summary (lm.fit)
```

The Boston data set contains 13 variables, and so it would be cumbersome to have to type all of these in order to perform a regression using all of the predictors. Instead, we can use the following short-hand:

```
> lm.fit =lm(medv∼.,data=Boston )
```

We can access the individual components of a summary object by name. summary(lm.fit)$r.sq gives us the *R*2, and summary(lm.fit)$sigma gives us the RSE. The vif() function, part of the car package, can be used to compute variance inflation factors. Most VIF's are low to moderate for this data. The car package is not part of the base R installation so it must be downloaded the first time you use it via the install.packages option in R.

```
> library (car)
> vif(lm.fit)
```

What if we would like to perform a regression using all of the variables but one? For example, in the above regression output, age has a high p-value. So, we may wish to run a regression excluding this predictor. The following syntax results in a regression using all predictors except age.

```
> lm.fit1=lm(medv∼.-age ,data=Boston )
```

It is easy to include interaction terms in a linear model using the lm() function. The syntax lstat:black tells R to include an interaction term between lstat and black. The syntax lstat*age simultaneously includes lstat, age, it is a shorthand for lstat+age+lstat:age.

4. Non-linear Transformations of the Predictors

The lm() function can also accommodate non-linear transformations of the predictors. For instance, given a predictor X, we can create a predictor X^2 using I(X^2). The function I() is needed since the ^ has a special meaning in a formula; wrapping as we do allows the standard usage in R, which is to raise X to the power 2. We now perform a regression of medv onto lstat and lstat^2.

```
> lm.fit2=lm(medv~lstat +I(lstat ^2))
```

We use the anova() function to further quantify the extent to which the quadratic fit is superior to the linear fit.

```
> lm.fit =lm(medv~lstat)
> anova(lm.fit ,lm.fit2)
```

In order to create a cubic fit, we can include a predictor of the form I(X^3). However, this approach can start to get cumbersome for higher order polynomials. A better approach involves using the poly() function to create the polynomial within lm(). For example, the following command produces a fifth-order polynomial fit:

```
> lm.fit5=lm(medv~poly(lstat ,5))
> summary (lm.fit5)
```

we can try a log transformation.

```
> summary (lm(medv~log(rm),data=Boston ))
```

5. Qualitative Predictors

We will now examine the Carseats data, which is part of the ISLR library. We will attempt to predict Sales (child car seat sales) in 400 locations based on a number of predictors.

```
> fix( Carseats )
> names(Carseats )
```

The Carseats data includes qualitative predictors such as Shelveloc, an indicator of the quality of the shelving location—that is, the space within a store in which the car seat is displayed—at each location. The predictor Shelveloc takes on three possible values, *Bad*, *Medium*, and *Good*. Given a qualitative variable such as Shelveloc, R generates dummy variables automatically. Below we fit a multiple regression model that includes some interaction terms.

```
> lm.fit =lm(Sales~.+ Income :Advertising +Price :Age ,data=Carseats )
> summary (lm.fit)
```

The contrasts() function returns the coding that R uses for the dummy variables.

```
> attach (Carseats )
> contrasts (ShelveLoc )
```