Assignment 1
MY 459 / MY 559

Benjamin Lauderdale
Methodology Department
London School of Economics

This document contains a long R tutorial and a short homework assignment. The exercise combines a review of material from MY452 and material covered in Lectures 1 and 2.

R Tutorial

When you open up R for the first time, it will just present you with a command line waiting for you to tell it to do something. In these assignment notes, when I want you to try running a command, I will include it in the document like this:

```
print("Hello R?")
```

As you should be following along, I will only sometimes also show the output that will result if you run the given command in R. When I provide the output, it will look like this:

```
print("Hello R?")


## [1] "Hello R?"
```

Note that in R itself, there will be a command prompt > that will indicate when R is ready to take a command. If you instead see a +, that means that a previous command is incomplete (usually due to an open parenthesis, bracket or brace) and R is waiting for you to complete it.

While you can type commands directly into the command line prompt and this is useful for testing things out, this is not how you should do your actual work/assignments. You should open up a new document in R (a *script*), and enter all your commands into that file. You can save that script, but you can also run either the entire script at any time or subsets of it. If you select several lines of commands, you can run them by typing Ctrl-R in Windows or Cmd-Return on a Mac. You should make it so that your script file is a complete record of all the commands needed to do the assignment in the order they should be executed, so that you can replicate your entire analysis anytime with the script file plus any data files you are using.

Example 1: Anscombe

To get started, copy the following into a new R script document, and then run the set of commands:

```r
x <- c(10,8,13,9,11,14,6,4,12,7,5)
y <- c(8.04,6.95,7.58,8.81,8.33,9.96,7.24,4.26,10.84,4.82,5.68)

# Print some summary statistics:
mean(x)
sd(x)
mean(y)
sd(y)

# Run a linear regression for the outcome y on the
# explanatory variable x:
savedregression <- lm(y~x)
summary(savedregression)

# Plot the data:
plot(x,y)

# Add the regression line to the data:
abline(savedregression)
```

A few things to notice here.

First, R will simply ignore anything that appears after the character #. These are comments, which you should use to remind yourself (and explain to others) what you are doing.

Second, the characters <- are the assignment operator in R.[1] That is, if you want to save something for later use, you put the name you want this object to have on the left side of a "<-" and the stuff you want to put in on the right. If you use the same name as an existing object, R will overwrite it without any warning. Using descriptive names is highly recommended!

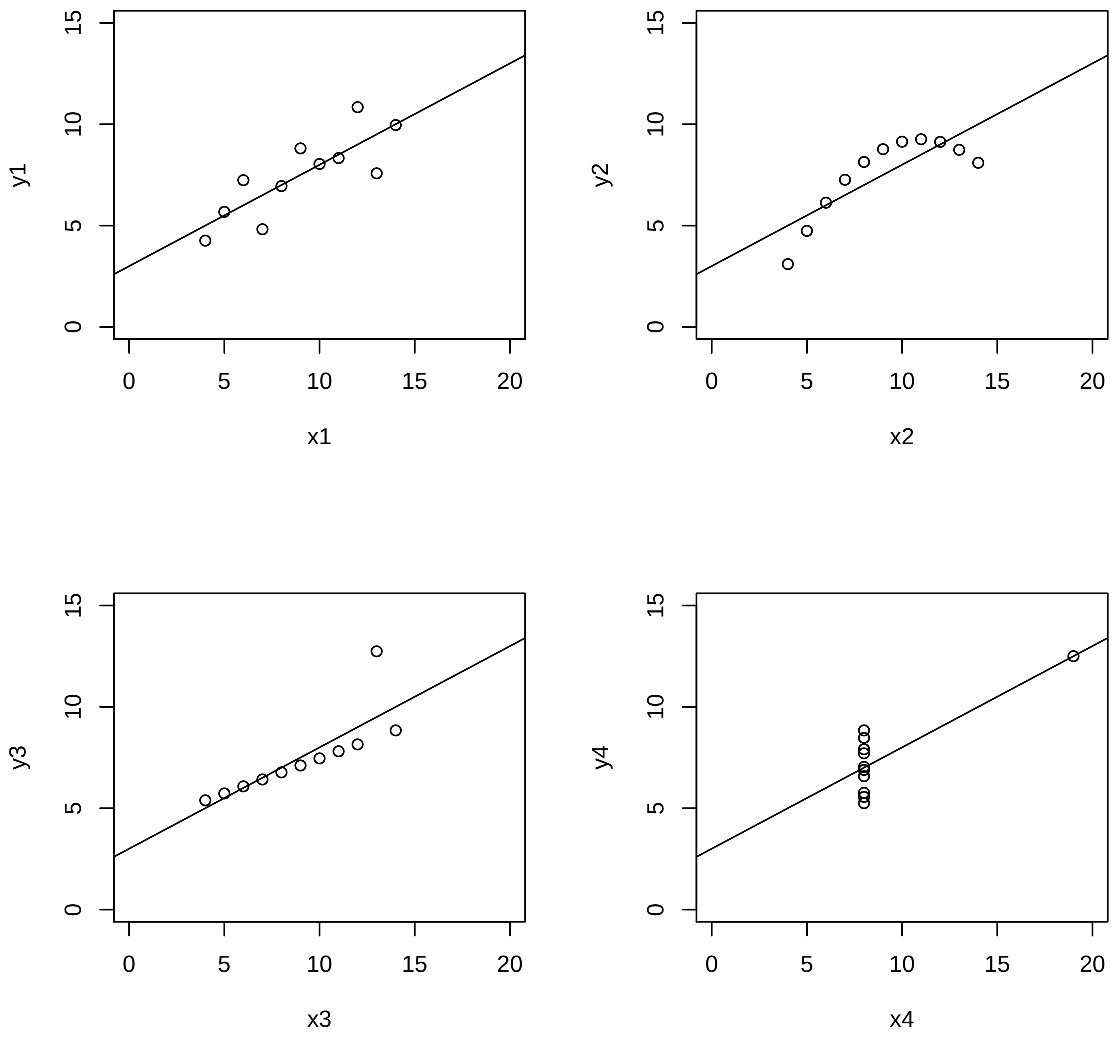
Third, a basic data type in R is a "vector", which can be manually created using the "`c()`" command (c is for concatenate). That is what we did in the first two lines above: we created a variable called "x" and a variable called "y", each of which has 11 data points in the specified order.

Obviously you will not want to manually type in all your data! There are many ways to get data into R. Most frequently, you will want to load datasets from saved files. Later in this assignment, we will load a data set directly from the internet. R also has some built-in data sets. In fact, the x and y data above are from a (famous) built-in dataset called anscombe after the statistician who made it up. Try running the following block of code:

```r
attach(anscombe)
par(mfrow=c(2,2))
plot(x1,y1,xlim=c(0,20),ylim=c(0,15))
abline(lm(y1~x1))
plot(x2,y2,xlim=c(0,20),ylim=c(0,15))
abline(lm(y2~x2))
plot(x3,y3,xlim=c(0,20),ylim=c(0,15))
abline(lm(y3~x3))
```

[1]Really this is just one of the assignment operators in R. All of the following do the same thing: `a <- b`; `b -> a`; `a = b`; `assign('a',b)`. By convention, most R programmers use `<-` as the assignment command and `=` only in

```
plot(x4,y4,xlim=c(0,20),ylim=c(0,15))
abline(lm(y4~x4))
```



Wait, what was all of that? Let's try again with some comments! Add these commands to your script file, then run them, some of the later commands in this exercise will depend on you having already run these commands.

```
# Take all of the variables in the "data frame" anscombe,
# and make them available for use. They are called:
#  x1, x2, x3, x4, y1, y2, y3, and y4.
attach(anscombe)

# Switch the plotting mode to create a grid of 2 by 2 plots
# (the default is a single plot)
par(mfrow=c(2,2))
```

arguments of functions. You will see examples of the latter usage below.

```
# Create a plot in which...
# the x variable is the vector "x1",
# the y variable is the vector "y1",
# the range of x values shown in the plot is from 0 to 20,
# and the range of y values shown in the plot is from 0 to 15.
plot(x1,y1,xlim=c(0,20),ylim=c(0,15))

# Fit the linear regression line to the data
# and add to the plot in one step.
abline(lm(y1~x1))

# Repeat for the 3 remaining x and y variables:
plot(x2,y2,xlim=c(0,20),ylim=c(0,15))
abline(lm(y2~x2))
plot(x3,y3,xlim=c(0,20),ylim=c(0,15))
abline(lm(y3~x3))
plot(x4,y4,xlim=c(0,20),ylim=c(0,15))
abline(lm(y4~x4))
```

If this has all worked properly, you should end up with a grid of four plots, all on the same scales, in which the data look very different, but the regression lines look suspiciously similar. If you run the following commands, you will see that the regression coefficients are in fact almost exactly the same.

```
lm(y1~x1)
lm(y2~x2)
lm(y3~x3)
lm(y4~x4)
```

These four pairs of x and y values were made up by Francis Anscombe (1973) to emphasise the point that one should actually plot one's data rather than just running regression models blindly!

You may be wondering how you can get more information when you run a regression. When you just type `lm(y1~x1)`, you will only get the coefficients. If you want more, try:

```
summary(lm(y1~x1))
```

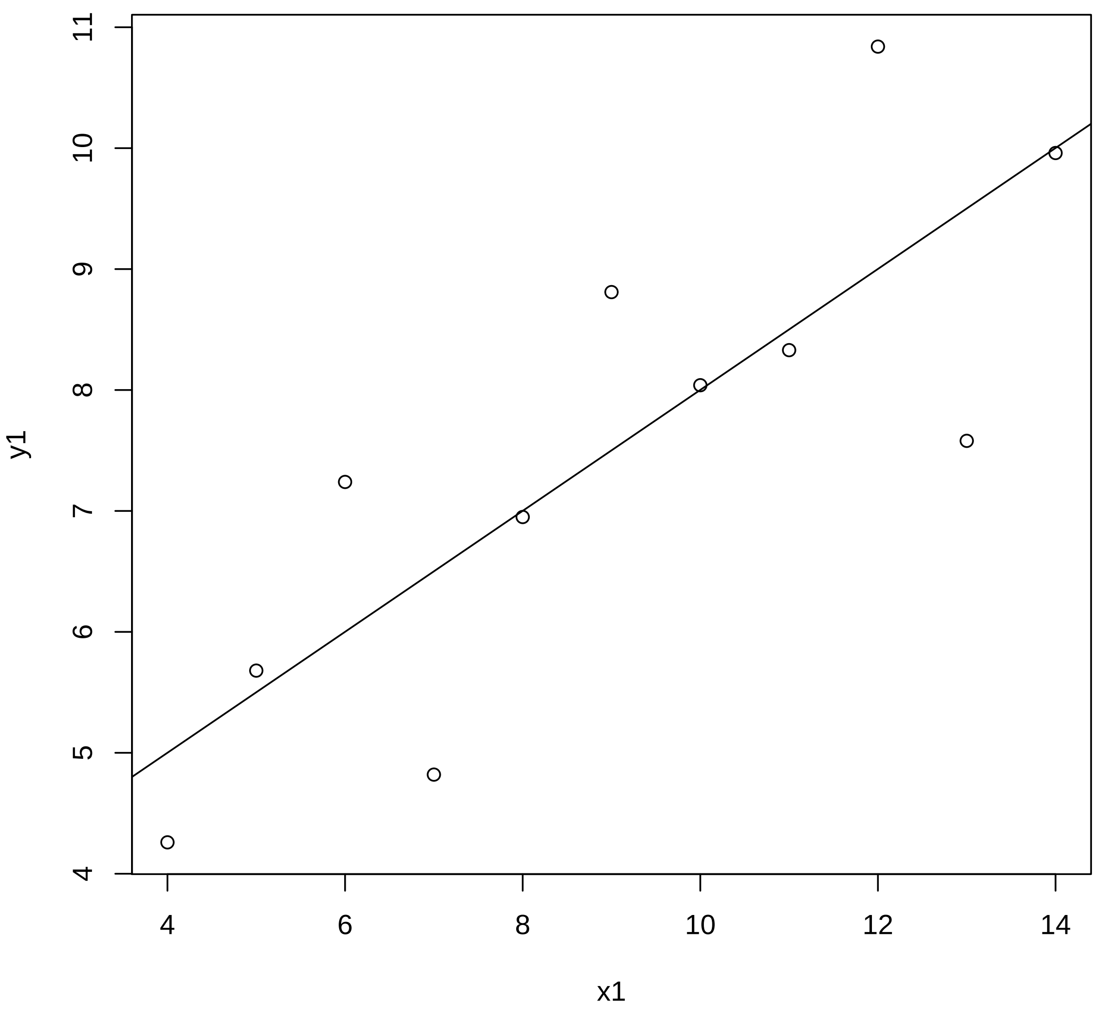
You can also do this in two steps, by first saving the regression results, and then asking R to give you a summary of the results.

```
mysavedregression <- lm(y1~x1)
summary(mysavedregression)


##
## Call:
## lm(formula = y1 ~ x1)
##
```

```
## Residuals:
##      Min       1Q   Median       3Q      Max 
## -1.92127 -0.45577 -0.04136  0.70941  1.83882 
## 
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)   
## (Intercept)   3.0001     1.1247   2.667  0.02573 * 
## x1            0.5001     0.1179   4.241  0.00217 **
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
## 
## Residual standard error: 1.237 on 9 degrees of freedom
## Multiple R-squared:  0.6665,Adjusted R-squared:  0.6295 
## F-statistic: 17.99 on 1 and 9 DF,  p-value: 0.00217
```

This will give you standard errors, 95% confidence intervals, etc. Once you have saved the regression results, you can also do lots of other stuff with the results. For example...

```
# To save and print the coefficients from the regression line:
mysavedcoefs <- coef(mysavedregression)
print(mysavedcoefs)

## (Intercept)          x1 
##   3.0000909   0.5000909

# To get differently sized confidence intervals:
confint(mysavedregression,level=0.9)

##                   5 %     95 %
## (Intercept) 0.9383030 5.061879
## x1          0.2839568 0.716225

confint(mysavedregression,level=0.99)

##                  0.5 %    99.5 %
## (Intercept) -0.6551512 6.6553330
## x1           0.1169174 0.8832644

# To plot the regression line:
plot(x1,y1)
abline(mysavedregression)
```



Example 2: Graduate School Admissions

Ok, let's try looking at a (somewhat) more interesting data set. The data set we are going to use is on PhD program admissions in the US. The included variables are Graduate Record Exam scores (gre), grade point average during the undergraduate (gap) and the prestige of the undergraduate institution (rank), and whether the applicant was admitted to the graduate program they applied to (admit). To begin, we are going to directly load the data off of a website on the internet.

```r
mydata <- read.csv("http://www.ats.ucla.edu/stat/data/binary.csv")
```

The data is in a widely used format called comma-seperated values (.csv). This is a widely useful format that can be read by almost any software and can be manually edited in any text editor. If you want to see what the data file looks like, try putting the web address from the command above into a web browser. It is just a text file, where each row corresponds to an observation, each column to a variable, and commas separating the columns. We do not need to open the file in a web-browser to look at the raw data, we have a variety of ways of doing so within R itself. Once you run the command above, R will have converted the data into an internal format called a "data frame". You can explore the contents of the data frame in many, many ways:

```r
# All of the data:
mydata
# The first few rows of the data:
head(mydata)
# The last few rows of the data:
tail(mydata)
# The fourth row (observation) of the data:
mydata[4,]
# The third column (variable) of the data:
mydata[,3]
# The value of the third variable for the
# fourth observation of the data:
mydata[4,3]
```

Since you probably do not want to try to remember which variable is in which column of the data frame, you can also find the names of the variables and address them directly. For example:

```r
# The names of the variables in the data frame:
names(mydata)
# One of the variables turns out to be called admit,
# we can directly access it by name with the command:
mydata$admit
```

When you are working with a single data set, it can be easier to skip writing `mydata$` before every variable name. You can use the `attach` command to make the variables in a data frame directly addressable. Using the attach command is a bit dangerous, because you can easily get confused as to whether you are changing the contents of the underlying data frame or not. We will use it here, but please be aware that you can get yourself into trouble this way!

```r
attach(mydata)
```

How many of the applicants in the data were admitted? There are many ways you can figure this out.

```r
sum(admit)
table(admit)
```

What fraction of the applicants in the data were admitted?

```r
mean(admit)
summary(admit)
prop.table(table(admit))
```

That last command is based on an idea that I have used several times already in these notes: giving the results of one function to another function. First, the function `table()` is called, and then the result is given as the *argument* for a second function `prop.table()`.

The function table can take more than one argument. For example, to get a table of how many students were admitted and not admitted, depending on the , run the command:

```
table(rank,admit)
prop.table(table(rank,admit))
prop.table(table(rank,admit),margin=1)
```

Notice that by adding the `margin=1` option after a comma, we changed the behavior of the `prop.table()` function from giving proportions of all cells, to just proportions within rows. In general across the various functions you will use in R, margin 1 is rows, margin 2 is columns. For example, if you wanted to calculate the mean of all the variables in the mydata data frame, you could type:

```
apply(mydata,2,mean)
```

The `apply()` function is a very powerful function, but is not always easy to figure out how to use. In this case, we have told apply to take the object `mydata` and apply the function `mean` to the second margin (columns). This does what we want, because data frames are organized with each variable as a separate column.

Currently, the variable rank is just numeric values of 1, 2, 3, or 4, which means that if we put it into a model, it will be treated as interval-level. Since these are categories, this is not what we want. To have this variable treated as a categorical variable, we need to convert it to a factor. We can do this using the factor command:

```
mydata$rank <- factor(mydata$rank,
labels=c("Highest","High","Low","Lowest"))
```

The labels part of that command is optional, but without it the categories would just be labeled 1, 2, 3, and 4, and it would be easy to forget that "1" correspondents to high-ranked undergraduate institutions and "4" to low-ranked institutions. Now we can run some models:

```
myLinearRegression <- glm(admit ~ gre + gpa + rank,
data = mydata, family = "gaussian")
myLogisticRegression <- glm(admit ~ gre + gpa + rank,
data = mydata, family = "binomial")
```

This fits two models on the same data, first a linear regression and then a logistic regression. Note that both are GLMs, so both are fit using the `glm` function. The linear regression can also be fit with the simpler `lm` command. Type `help(lm)` to see information on how that command works.

```
summary(myLinearRegression)


##
## Call:
## glm(formula = admit ~ gre + gpa + rank, family = "gaussian",
##     data = mydata)
##
```

```
## Deviance Residuals:
##     Min       1Q   Median       3Q      Max
## -0.7022  -0.3288  -0.1922   0.4952   0.9093
##
## Coefficients:
##               Estimate Std. Error t value Pr(>|t|)
## (Intercept) -0.2589102  0.2159904  -1.199   0.2314
## gre          0.0004296  0.0002107   2.038   0.0422 *
## gpa          0.1555350  0.0639618   2.432   0.0155 *
## rankHigh    -0.1623653  0.0677145  -2.398   0.0170 *
## rankLow     -0.2905705  0.0702453  -4.137 4.31e-05 ***
## rankLowest  -0.3230264  0.0793164  -4.073 5.62e-05 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for gaussian family taken to be 0.1979062)
##
##     Null deviance: 86.677  on 399  degrees of freedom
## Residual deviance: 77.975  on 394  degrees of freedom
## AIC: 495.12
##
## Number of Fisher Scoring iterations: 2
```

```r
summary(myLogisticRegression)
```

```
##
## Call:
## glm(formula = admit ~ gre + gpa + rank, family = "binomial",
##     data = mydata)
##
## Deviance Residuals:
##     Min       1Q   Median       3Q      Max
## -1.6268  -0.8662  -0.6388   1.1490   2.0790
##
## Coefficients:
##              Estimate Std. Error z value Pr(>|z|)
## (Intercept) -3.989979   1.139951  -3.500 0.000465 ***
## gre          0.002264   0.001094   2.070 0.038465 *
## gpa          0.804038   0.331819   2.423 0.015388 *
## rankHigh    -0.675443   0.316490  -2.134 0.032829 *
## rankLow     -1.340204   0.345306  -3.881 0.000104 ***
## rankLowest  -1.551464   0.417832  -3.713 0.000205 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 499.98  on 399  degrees of freedom
## Residual deviance: 458.52  on 394  degrees of freedom
## AIC: 470.52
```

```
##
## Number of Fisher Scoring iterations: 4
```

If you want to turn the logistic regression coefficients and confidence intervals into odds ratios, and nicely format them at the same time, you can do something like the following. Below, a single line uses the `cbind` command to bind together the column of coefficients from the `coef` command, two columns for the limits of the confidence intervals, and then exponentiate all of them at the same time to turn the coefficient scale into the odds ratio scale. Try running the individual pieces of the command alone to see how they build up to the desired table.

```
exp(cbind(OR = coef(myLogisticRegression),
confint(myLogisticRegression)))

## Waiting for profiling to be done...

##                    OR       2.5 %    97.5 %
## (Intercept) 0.0185001 0.001889165 0.1665354
## gre         1.0022670 1.000137602 1.0044457
## gpa         2.2345448 1.173858216 4.3238349
## rankHigh    0.5089310 0.272289674 0.9448343
## rankLow     0.2617923 0.131641717 0.5115181
## rankLowest  0.2119375 0.090715546 0.4706961
```

Finally, how do we plot the lines/curves of the mean functions for the linear and logistic models? First, let's plot admission as a function of GRE. The first line returns the plot layout to a 1 by 1 grid of plots, from the 2 by 2 grid we used earlier.

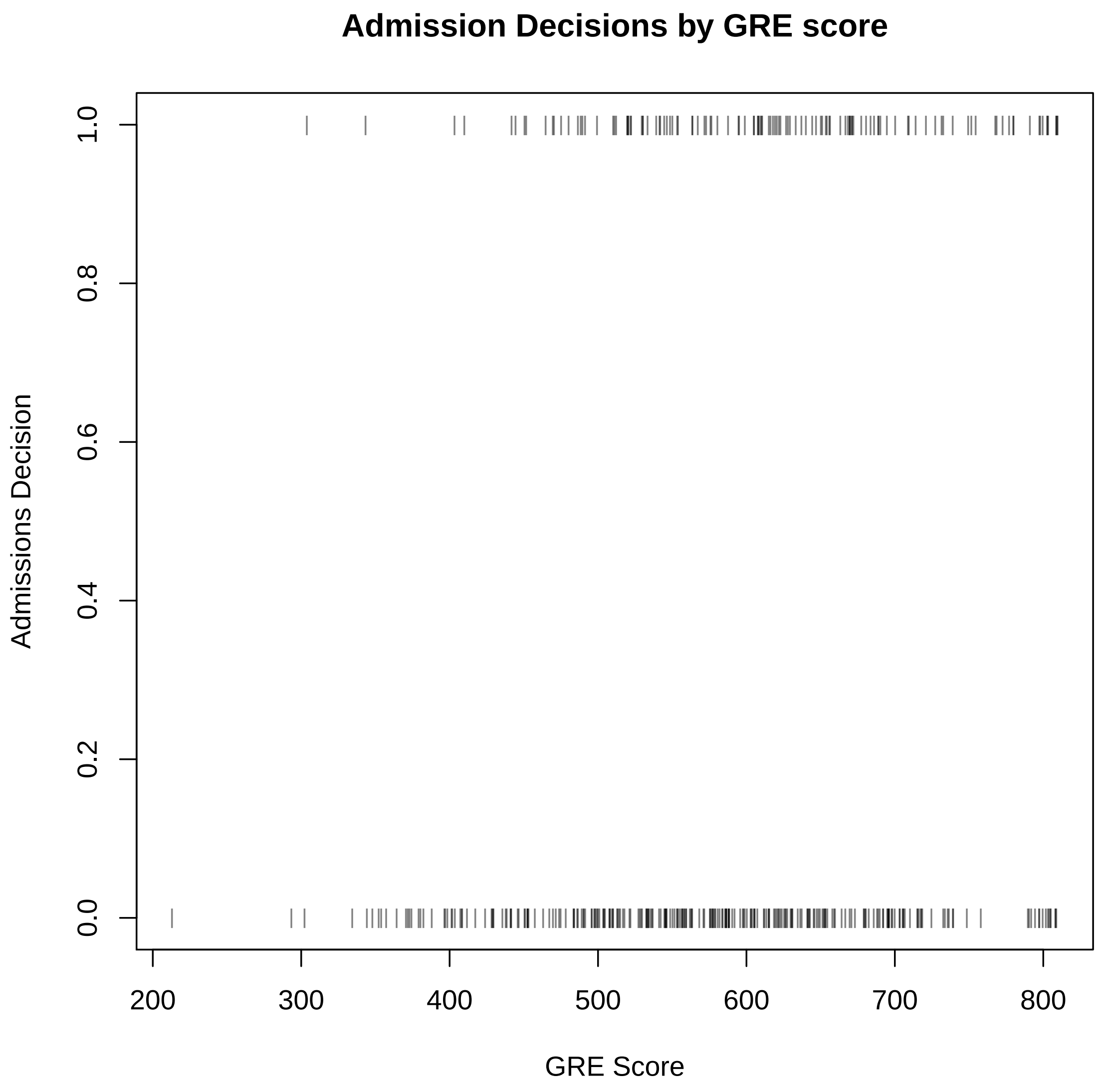
```
par(mfrow=c(1,1))
plot(gre,admit,main="Admission Decisions by GRE score")
```

Hmm, that is a bit difficult to read, what with all the data points being on top of one another. If you have this problem with a scatter plot, you could use a different kind of plot, but in this case we want this kind of plot so we can overlay the mean functions on the plot. A nice solution is the `jitter()` command, which adds a small amount of noise to a variable so that you can see all the data points. Try:

```
plot(gre,jitter(admit,factor=0.5),
main="Admission Decisions by GRE score")
```

You are not really a proper R user until you start obsessively tinkering around with the graphics settings. We can generate an even more stylish plot if we do the following:

```
plot(jitter(gre,amount=10),admit,
main="Admission Decisions by GRE score",
xlab="GRE Score",
ylab="Admissions Decision",
pch="|",
cex=0.75,
col=rgb(0,0,0,0.5))
```



There are many, many things you can do by playing with the plot options in R. In this case, using a vertical tick for the points works nicely to show where there are many observations and where there are few. Making the tick marks smaller (the `cex` option) makes the ticks a bit more distinct from one another. At the same time, adding a bit of noise to the GRE score helps with the visualization by eliminating the gaps that come from the fact that the scores are in intervals of 20 points. Adding ± 10 points through the `amount` option of the `jitter` command spreads out the observations evenly through those 20 point intervals. Making the colour of the points partially transparent (the `rgb(0,0,0,0.5)` argument) helps make the relative density of points easier to see as well.

Ok, now let's figure out how to add the mean function lines/curves. The basic logic is as follows. First, we are going to create a fake data set with the same variables we had in our real data set. The values of those variables are going to be set to the values of the explanatory variables we are interested in. For example, if we want to vary GRE over it's range from 200 to 800, while holding GPA fixed at its mean, and setting the rank of the school to it's second best value, we would use the following command:

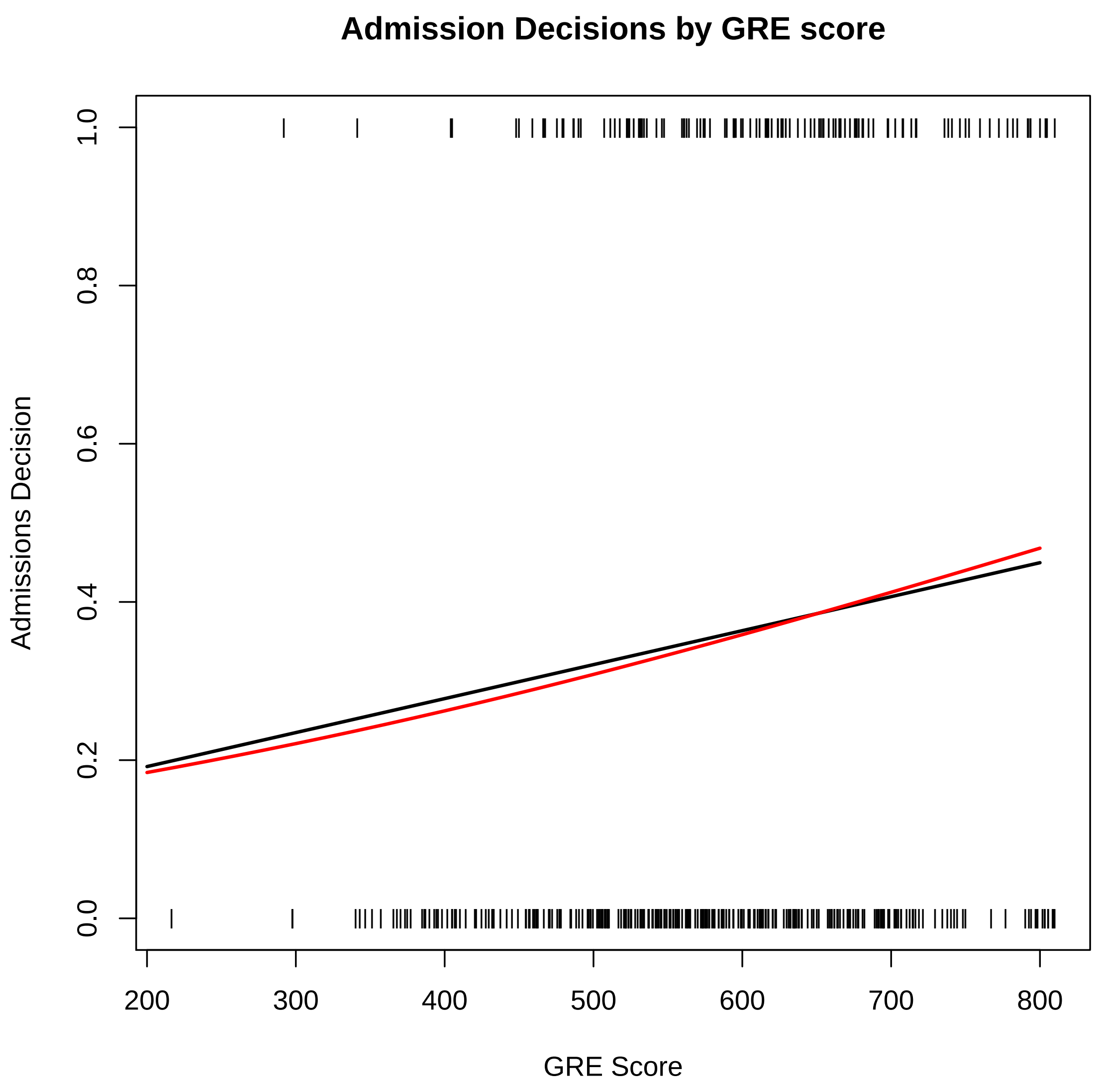
```
myfakedata <- with(mydata,
data.frame(gre = seq(200,800,20), gpa = mean(gpa), rank = "High"))
```

The with(mydata,) part of the command tells R that we are creating a new data frame that should look like the existing data frame called mydata. Then, the data.frame() part of the command tells R what to put in this new frame. The function seq(200,800,20) creates a sequence from 200 to 800 in equal intervals of 20 (try running just that command in R). For gpa and rank we have only provided a single value, but R will just repeat that value to generate a data set with the same length as the sequence of GRE scores. If you now enter myfakedata into R, it will show you your shiny new, fake data set. Now, all we have to do is run the following two commands to have R generate the model predictions for our fake data set, using the linear and logistic regression model estimates that we saved earlier.

```
LinearPredictions <- predict(myLinearRegression,
newdata = myfakedata, type = "response")
LogisticPredictions <- predict(myLogisticRegression,
newdata = myfakedata, type = "response")
```

Finally, we can plot these predictions. Remember, these predictions correspond to a sequence of points running from 200 to 800 in intervals of 20. We saved this sequence in myfakedata$gre. The lines command plots a series of points with lines connecting them, which is what we want here. First we plot the data as before, then we add the lines.

```
plot(jitter(gre,amount=10),admit,
main="Admission Decisions by GRE score",
xlab="GRE Score",
ylab="Admissions Decision",
pch="|",
cex=0.75)
lines(myfakedata$gre,LinearPredictions,lwd=2)
lines(myfakedata$gre,LogisticPredictions,lwd=2,col="red")
```



You will note that the linear and logistic predicted probabilities / fitted values are nearly identical. This tends to happen when the probability of a binary outcome is neither very close to zero nor very close to one, and the explanatory variable is not especially predictive of the outcome.

Try modifying the code above to show admissions decisions as a function of GPA instead of GRE. This will involve making changes to the plot commands as well as the command where you create the fake data.

Finally, we can use the predict command on the actual data to generate fitted values for the models. Since this is the default for the `predict` command, we do not need to specify the data set.

```
LinearFittedValues <- predict(myLinearRegression,
type = "response")
LogisticFittedValues <- predict(myLogisticRegression,
type = "response")
```

Now that we have the fitted values, we can construct the root mean square error (RMSE) for both models. This involves taking the square root of the mean of the squares of differences between the observed values of the dependent variable `admit` and the fitted values under the two models:

```
sqrt(mean((admit - LinearFittedValues)^2))

## [1] 0.4415173

sqrt(mean((admit - LogisticFittedValues)^2))

## [1] 0.4412641
```

As the previous plot indicated, there is not much difference between the models in terms of fit to the sample. Rather than typing out the whole expression for the RMSE each time we want to calculate it, we can define a function to do it:

```
rmse <- function(fitted,observed) sqrt(mean((observed - fitted)^2))

rmse(LinearFittedValues,admit)

## [1] 0.4415173

rmse(LogisticFittedValues,admit)

## [1] 0.4412641
```

We have been using the saved regression models `myLinearRegression` and `myLogisticRegression` all of this time, but what is actually in those objects? The command `str()` will tell you about the internal structure of any R data structure. Here, let's look at what is created when we use the summary command on a linear model created by `lm()` (you can also run `str()` directly on the model object, but the resulting list of components is very large and I do not want to print it out here).

```
myLinearModel <- lm(admit ~ gre + gpa + rank, data = mydata)
str(summary(myLinearModel))

## List of 11
##  $ call         : language lm(formula = admit ~ gre + gpa + rank, data = mydata)
##  $ terms        :Classes 'terms', 'formula' length 3 admit ~ gre + gpa + rank
##   .. ..- attr(*, "variables")= language list(admit, gre, gpa, rank)
##   .. ..- attr(*, "factors")= int [1:4, 1:3] 0 1 0 0 0 0 1 0 0 0 ...
##   .. .. ..- attr(*, "dimnames")=List of 2
##   .. .. .. ..$ : chr [1:4] "admit" "gre" "gpa" "rank"
##   .. .. .. ..$ : chr [1:3] "gre" "gpa" "rank"
##   .. ..- attr(*, "term.labels")= chr [1:3] "gre" "gpa" "rank"
##   .. ..- attr(*, "order")= int [1:3] 1 1 1
##   .. ..- attr(*, "intercept")= int 1
##   .. ..- attr(*, "response")= int 1
##   .. ..- attr(*, ".Environment")=<environment: R_GlobalEnv>
##   .. ..- attr(*, "predvars")= language list(admit, gre, gpa, rank)
##  .. ..- attr(*, "dataClasses")= Named chr [1:4] "numeric" "numeric" "numeric" "factor"
##   .. .. ..- attr(*, "names")= chr [1:4] "admit" "gre" "gpa" "rank"
##  $ residuals    : Named num [1:400] -0.1752 0.6951 0.2931 0.8109 -0.0972 ...
##   ..- attr(*, "names")= chr [1:400] "1" "2" "3" "4" ...
##  $ coefficients : num [1:6, 1:4] -0.25891 0.00043 0.15554 -0.16237 -0.29057 ...
##   ..- attr(*, "dimnames")=List of 2
##   .. ..$ : chr [1:6] "(Intercept)" "gre" "gpa" "rankHigh" ...
##   .. ..$ : chr [1:4] "Estimate" "Std. Error" "t value" "Pr(>|t|)"
##  $ aliased      : Named logi [1:6] FALSE FALSE FALSE FALSE FALSE FALSE
##   ..- attr(*, "names")= chr [1:6] "(Intercept)" "gre" "gpa" "rankHigh" ...
##  $ sigma        : num 0.445
##  $ df           : int [1:3] 6 394 6
##  $ r.squared    : num 0.1
##  $ adj.r.squared: num 0.089
##  $ fstatistic   : Named num [1:3] 8.79 5 394
##   ..- attr(*, "names")= chr [1:3] "value" "numdf" "dendf"
##  $ cov.unscaled : num [1:6, 1:6] 2.36e-01 -4.64e-05 -5.53e-02 -2.22e-02 -1.92e-02 ...
##   ..- attr(*, "dimnames")=List of 2
##   .. ..$ : chr [1:6] "(Intercept)" "gre" "gpa" "rankHigh" ...
##   .. ..$ : chr [1:6] "(Intercept)" "gre" "gpa" "rankHigh" ...
##  - attr(*, "class")= chr "summary.lm"
```

You can see that all sorts of information is saved, in the form of a list: the model specification, the residuals, the coefficients, the standard deviation of the residuals σ, the R^2, etc. Because this object is a list, and we now know the names of its subcomponents, we can directly access them if we needed them for a calculation:

```
summary(myLinearModel)$coefficients
```

```
##                 Estimate   Std. Error   t value     Pr(>|t|)
## (Intercept) -0.258910210 0.2159903968 -1.198712 2.313606e-01
## gre          0.000429572 0.0002107347  2.038449 4.217224e-02
## gpa          0.155535025 0.0639618357  2.431685 1.547363e-02
## rankHigh    -0.162365349 0.0677144786 -2.397794 1.695874e-02
## rankLow     -0.290570480 0.0702453273 -4.136510 4.313004e-05
## rankLowest  -0.323026366 0.0793163565 -4.072632 5.621191e-05
```

Example 3: Calculating Fitted Values Manually

Above, we used some built-in commands to construct fitted values. But let's imagine that we did not know about the `predict()` command or even the much simpler `fitted()` command. We could still calculate the fitted values for the linear model, because there is a simple formula:

$$\hat{Y} = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \cdots \tag{1}$$

Note that if we are thinking about programming this calculation, to be applied to all the observations in our data, it is useful to keep track of the indices:

$$\hat{Y}_i = \beta_0 + \beta_1 X_{1i} + \beta_2 X_{2i} + \cdots \tag{2}$$

We need two things to do this calculation. We need the values of X for each observation and the need the coefficients β:

```
X <- model.matrix(myLinearModel)
print(head(X))

##   (Intercept) gre  gpa rankHigh rankLow rankLowest
## 1           1 380 3.61        0       1          0
## 2           1 660 3.67        0       1          0
## 3           1 800 4.00        0       0          0
## 4           1 640 3.19        0       0          1
## 5           1 520 2.93        0       0          1
## 6           1 760 3.00        1       0          0

beta <- coef(myLinearModel)
print(beta)

##  (Intercept)          gre          gpa     rankHigh      rankLow
## -0.258910210  0.000429572  0.155535025 -0.162365349 -0.290570480
##   rankLowest
## -0.323026366
```

Ok, so now we have X and β. There are several ways we can do the necessary calculation for each observation. Here is how we can do it with a for loop:

```
N <- nrow(X)     # how many observations are there?
fitted1 <- rep(NA,N)     # create empty vector to put fitted values in
for (i in 1:N){
        fitted1[i] <- sum(beta*X[i,]) # multiply coef by X for obs i
}
head(fitted1)

## [1] 0.17523809 0.30485034 0.70688745 0.18914620 0.09715846 0.37180420
```

Here is how we can do it with the `apply` command:

```
fitted2 <- apply(X,1,function(x) return(sum(x*beta)))
head(fitted2)

##          1          2          3          4          5          6
## 0.17523809 0.30485034 0.70688745 0.18914620 0.09715846 0.37180420
```

Here is how we can do it with matrix multiplication:

```
fitted3 <- X %*% beta
head(fitted3)

##         [,1]
## 1 0.17523809
## 2 0.30485034
## 3 0.70688745
## 4 0.18914620
## 5 0.09715846
## 6 0.37180420
```

Note that the last version returns an $N \times 1$ matrix rather than a length N vector, which is a difference that might be important depending on what you were doing next. But you can use `as.vector(fitted3)` or `matrix(fitted1,N,1)` to convert between the two types if you need to. Finally, let's just make sure those fitted values are the same as the ones we got the direct way!

```
tail(cbind(LinearFittedValues,fitted1,fitted2,fitted3))

##     LinearFittedValues   fitted1   fitted2
## 395          0.2687072 0.2687072 0.2687072 0.2687072
## 396          0.4671991 0.4671991 0.4671991 0.4671991
## 397          0.1639061 0.1639061 0.1639061 0.1639061
## 398          0.1853847 0.1853847 0.1853847 0.1853847
## 399          0.4471276 0.4471276 0.4471276 0.4471276
## 400          0.3132937 0.3132937 0.3132937 0.3132937
```

Homework

For this assignment, you will need to load a data set that is not in the standard set of R libraries, but is already installed on the LSE computers. If you are working on your own computer, you will need to first install the package onto your computer:

```r
install.packages("pscl",dependencies=TRUE)
```

Then, regardless of whether you are using an LSE computer or your own, you will need to load the package:

```r
library(pscl)
```

The data file for the assignment is `iraqVote`. Once you have loaded the `pscl` library, run the following command to directly access the variables in the data set:

```r
attach(iraqVote)

## The following objects are masked from package:datasets:
##
##    state.abb, state.name
```

The message about objects being masked means that there were already objects with the names `state.abb` and `state.name` loaded, and the versions from the data.frame are now the ones that will be accessed via those names. To figure out what all the variables in the data set are, use the help file that documents the data set: `help(iraqVote)`

1. Fit a linear regression model for a Senator's vote as a function of whether the senator was a Republican and the vote in that Senator's state in the preceding presidential election.
   (a) Write out the equation for the mean function in terms of the variables and the coefficient estimates.
   (b) Interpret the intercept of the model.
   (c) Calculate the root mean square error (RMSE) for the model.
   (d) Over what range of the explanatory variables does the linear model give a prediction greater than on or less than zero?
2. Fit a logistic regression model for vote as a function of party and vote.
   (a) Write out the equation for the mean function in terms of the variables and the coefficient estimates.
   (b) Calculate the RMSE for the model and compare it to the RMSE for the linear model.
3. Plot the predicted probabilities / fitted values for the logistic regression with a blue line for Democrats and a red line for Republicans.
   (a) How would the coefficient estimates need to be different in order to eliminate the gap between the two curves?
   (b) What is the substantive interpretation of this gap? What (if anything) does it tell us about which kinds of Senators voted to authorise the Iraq War in 2002? (I am not asking for anything subtle here.)

4. Manually calculate the fitted values for the logistic regression by extracting the model matrix and coefficients from the model object, multiplying as implied by the model, and then applying the logistic transformation. In case you have forgotten, the fitted values for the logistic model are given by:

$$\hat{\pi} = \frac{exp\left(\beta_0 + \beta_1 X_1 + \beta_2 X_2 + \cdots\right)}{1 + exp\left(\beta_0 + \beta_1 X_1 + \beta_2 X_2 + \cdots\right)} \tag{3}$$

   Check whether you get the same fitted values that you got with the predict command.