STAT2008/6038 Regression Modelling Worksheet 3

1. *Randomness.* Use *R* commands to do the following:

   (a) Generate 1000 normal variates from the normal distribution with mean 3 and variance 16, and put the result into a vector `normals`.

   (b) Construct a histogram from the simulated vector `normals` using histogram command.

   (c) What are the smallest and largest value in `normals`?

   (d) The histogram command can be forced to use certain break points rather than *R's* default breakpoints. For example, the command

   ```
   hist(x,breaks=seq(floor(min(x)),floor(max(x)+1),1))
   ```

   will create a histogram of data values in `x`, but it will use bins of width one spanning the range between the smallest and largest integers bracketing the range of `x` values respectively. By adjusting the bin width (that is, the third argument in the seq() function above), you can achieve a smoother or rougher looking histogram than that produced by *R's* default binwidth. Experiment with various bin-widths by varying the line given above and see what bin-width yields the "smoothest-looking" histograms. What do you notice about the histograms as the bin-width increases? decreases?

   (e) When we talk about "smoothness" of a histogram, as we are doing above, we are referring to its closeness to a smooth curve. What curve is the histogram approximating? Since the variable `normals` ostensibly comes from a N(3,16) distribution, it makes sense to compare it with that normal distribution. To see what this distribution looks like, take a vector that spans the range of the values in `normals`, for example, the vector

   ```
   grid <- seq(floor(min(normals)),floor(max(normals)+1),1)
   ```

   is such a vector. Find the values of a N(3,16) density corresponding to the values in `grid`, and store these in a vector called `really.normal`. Now, use the `lines` command to superimpose the plot of `grid` versus `really.normal` on one of the histograms constructed in (d). What do you notice? If you said "not much", you are right, since

the normal density we plotted has a completely different scale to the histogram. This is because the histogram uses raw frequencies rather than relative frequencies. To account for this, we will want to "blow up" the normal density we plotted by a factor of 1000 (the total number of simulated values) multiplied by the bin width (in this case 1). So now superimpose on your histogram the plot of `grid` versus `1000` × bin-width × `really.normal`. You may like to compare a plotted density against each of the histograms plotted in (d) to see which bin-width corresponds most closely to the true curve. Do you think *R* has made the right choice of bin-width to accurately portray the data? Do you believe the generate pseudo-random values are normal?

(f) Let's test the last idea that the pseudo-random values are credibly normal. For this task, we want to make a normal q-q plot. The idea of a qqplot is basically to match up the ordered data values with what we would *expect* to be the ordered values from a normal distribution with the appropriate mean and variance. These latter values are called normal scores. Construct a qqplot of the simulated data in `normals`. Do you think the simulated data might reasonably have come from a normal distribution?

(g) Now, let's look at a sample smaller than 1000. Generate a vector `expo` of 15 variates from an exponential distribution with mean 1. What is the mean of your sample? the variance? Plot a normal q-q plot for your new data. What do you notice? How does this data differ from normal data?

2. *Functions.* In this part of the worksheet, we are going to write our own *R* function. This function is going to investigate how the slope and intercept of a least-squares regression line change when several user-chosen data points are excluded, and we shall call it `betachng()`. Such a function will be useful for identifying potential outliers and influential points in a data set.

   Writing this *R* function will introduce you to several *R* programming constructions, including conditional branching and functions themselves. In general, functions have the form

   ```
   function(argument1, argument2, ...){
   <text of function goes here>
   }
   ```

   The value of the last line of the contents of the function will be its returned value. As a simple example, recall the following function which finds the square of its argument:

   ```
   square <- function(x){
   ```

```
x^2
}
```

The last (and only) line of the function is the value returned by the function. Alternatively, we could have written this function as:

```
square <- function(x){
y <- x*x
y
}
```

Since the first line of this function is an assignment and therefore has no value, we must include the second line so that the function will return the desired result.

To start our function `betachng()`, we must first think of what our arguments need to be. The function must take as input the original response and predictor variable data to calculate the least-squares regression and also a vector of numbers indicating which points to exclude. Now we are ready to begin the function

```
betachng <- function(resp,pred,excl){
```

Now, our first step will be to check that there are no repeats in the `excl` argument. So the first line of our function will be:

```
exc <- unique(excl)
```

since the `unique()` function takes a vector input and returns a vector with any repeat values removed.

We now need to check whether any values in the `excl` argument are invalid, i.e. whether they are negative or larger than the number of data points in the given data vectors. If the values are invalid, then we need to print a warning message and otherwise, we need to run our functions calculation. To do this, we will use the `if()`, `else if()` and `else()` functions as follows:

```
if(min(excl)<1) {
print("Invalid Point to be Excluded - Index too small") }
else if(max(excl)>length(pred)) {
print("Invalid Point to be Excluded - Index too large") }
else {
```

Now, once we have established that the `excl` argument is valid, we can proceed to calculate the slope and intercept for the least-squares regression using the entire data set

```
beta <- lsfit(pred,resp)$coef
```

and the slope and intercept for the least-squares regression using the reduced data set

```
beta.red <- lsfit(pred[-exc],resp[-exc])$coef
```

Notice that we make use of an important aspect of the "square bracket" notation for vectors; namely, that if negative values are given, then *R* returns a vector with the specified elements removed. In other words, typing `x[1]` will return the first element of the vector `x`, while typing `x[-1]` will return a vector which contains all of the elements of the vector `x` except for the first. Lastly, we must calculate the difference in the slopes and intercepts and return this value as the value of our function:

```
beta - beta.red }
}
```

So, the entire function would look as follows:

```
betachng <- function(resp,pred,excl){
  exc <- unique(excl)
  if(min(excl)<1) {
    print("Invalid Point to be Excluded - Index too small") }
  else if(max(excl)>length(pred)) {
    print("Invalid Point to be Excluded - Index too large") }
  else {
    beta <- lsfit(pred,resp)$coef
    beta.red <- lsfit(pred[-exc],resp[-exc])$coef
    beta - beta.red }
}
```

Recall the data in the file *worksheet2_women*. This data should still be in your workspace (see Worksheet 2). Use this data to try out the new function `betachng()`. Can you find any points which have a large effect on the values of the slope and intercept of the least-squares regression, either singly or in groups?