Inference Tutorial 2

This sheet covers the basics of linear modelling in R, as well as bootstrapping, and the frequentist notion of a confidence interval. When working in R, always create a file containing your R code, which you can paste into R to produce the answer to the question (or work in Rstudio if you prefer).

1. R has built in functions for least squares estimation of linear models, along with other inferential tasks based on the theory covered in lectures. Since we are mostly interested in linear models to illustrate some principles of statistical inference we will only cover some aspects of the use of these functions, rather than being comprehensive. As a first introduction let's fit the `cars` model from section 3.1.1 of the notes.

   (a) Type `head(cars)` at the R command prompt. This will print the first few lines of the `cars` data frame, to give you an idea of what is there. A *data frame* is a set of named columns of data, which can be of mixed type. For example you could have some columns of numbers, some columns representing factor variables, and even some columns of characters.

   (b) The `lm` function is used to estimate linear models. For example, the model

   $$\texttt{dist}_i = \beta_1 \texttt{speed}_i + \beta_2 \texttt{speed}_i^2 + \epsilon_i$$

   from the notes, can be estimated using

   ```
   cars.mod <- lm(dist ~ speed + I(speed^2) -1,data=cars)
   ```

   `<-` is the assignment operator in R. Here it is used to assign the *fitted model object*, returned by `lm` to a newly created object `cars.mod`. The arguments of `lm` are inside the round brackets on the r.h.s. The first argument is the *model formula* `dist ~ speed + I(speed^2)`. Model formulae provide a simple means for specifying models structures in R. Whatever is to the left of `~` specifies the response variable, while what is to the right specifies how the expected response depends on predictors. Here we have specified that the expected response should depend (linearly) on `speed` and $\texttt{speed}^2$. The `I()` in the model formula is to ensure that `^2` has its usual arithmetic meaning, and not the special meaning it would otherwise have in a model formula. Usually R would add an intercept term to the model by default: '`-1`' tells it not to. `data=cars` tells `lm` that the required variables are to be found in the `cars` data frame.

   Try it, and then type `cars.mod` in R, to get a short summary of the fitted model. Identify $\hat{\beta}_1$ and $\hat{\beta}_2$.

   (c) `cars.mod` is actually an R object of *class* `"lm"`. When you typed `cars.mod` at the R prompt, R tried to print it. Because of its class, printing got passed to the R function `print.lm` which resulted in the brief output you saw. There is alot more that can be extracted from `cars.mod`. For example type

   ```
   summary(cars.mod)
   ```

   From the result, identify $\hat{\beta}_1$, $\hat{\beta}_2$ and $\hat{\sigma}_{\hat{\beta}_1}$ and $\hat{\sigma}_{\hat{\beta}_2}$. Now identify what tests are being performed to arrive at the p-values reported in the `Coefficients` table. Find $\hat{\sigma}$.

   (d) Use `model.matrix(cars.mod)` to examine the model matrix (**X** matrix) of this linear model, confirming it is as you expected.

   (e) Now do a bit more model checking. There are 2 aspects - are the model assumptions of independence and constant variance of the residuals met, and was the model structure adequate? To address both, fit the model

   $$\texttt{dist}_i = \beta_0 + \beta_1 \texttt{speed}_i + \beta_2 \texttt{speed}_i^2 + \beta_3 \texttt{speed}_i^3 + \epsilon_i$$

   Supposing the resulting fitted model is `cm1`. First type `plot(cm1)` to check some residual plots for problems with the assumptions. The first plot shows $\hat{\epsilon}_i$ against $\hat{\mu}_i$ (the red line is a running average of the residuals) - is there any pattern in the mean or variability of the residuals w.r.t. the fitted values, $\hat{\mu}_i$? The next plot is a quantile-quantile plot (ordered residuals against quantiles of a standard normal distribution), which should be close to a straight line if the normality assumtion holds for the residuals. The third plot is like the first, but with transformed residuals, making it a bit easier to judge constant variance (a trend in the running average curve now indicates non constant variance). We won't cover the 4th plot here.

   (f) Now use the `summary` function on `cm1`. Notice how the p-values have all increased. Clearly this can not mean that all the $\beta_i = 0$, since when we had only 2 terms in the model then both appeared to be non-zero. What we have here is a loss of precision as a result of the columns of the model matrix being highly correlated, so that their effects can not be distinguished. To proceed it makes sense to try re-fitting the model sequentially dropping the least significant term (highest p-value), until all terms have p-values less than some threshold (e.g. 0.05). Try this. What model do you end up with? Any caveats?

(g) Obtain a 95% confidence interval for reaction time from your final fitted model (use `?cars` to get the units of distance and speed).

Solution

(a)
```
> head(cars)
  speed dist
1     4    2
2     4   10
```

(b)
```
> cars.mod <- lm(dist ~ speed + I(speed^2) -1,data=cars)
> cars.mod
...
Coefficients:
     speed  I(speed^2)
   1.23903     0.09014
```
So $\hat{\beta}_1 \simeq 1.24$ and $\hat{\beta}_2 \simeq 0.090$.

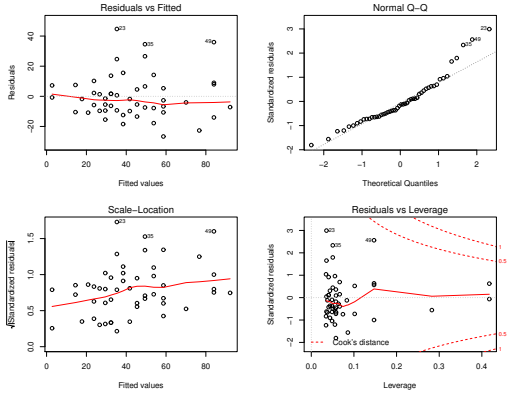
(c)
```
> summary(cars.mod)

Call:
lm(formula = dist ~ speed + I(speed^2) - 1, data = cars)
...
Coefficients:
           Estimate Std. Error t value Pr(>|t|)
speed       1.23903    0.55997   2.213  0.03171 *
I(speed^2)  0.09014    0.02939   3.067  0.00355 **
---

Residual standard error: 15.02 on 48 degrees of freedom
Multiple R-squared:  0.9133,Adjusted R-squared:  0.9097
F-statistic: 252.8 on 2 and 48 DF,  p-value: < 2.2e-16
```
Parameter estimates as above, $\hat{\sigma}_{\hat{\beta}_1} \simeq 0.56$ and $\hat{\sigma}_{\hat{\beta}_2} \simeq 0.029$. Test statistics and p-values for two tests are reported $H_0 : \beta_1 = 0$ and $H_0 : \beta_2 = 0$, the low p-values indicate that there is evidence against both, quite strong evidence in the second case. $\hat{\sigma} = 15.02$.

(d)
```
> model.matrix(cars.mod)
   speed I(speed^2)
1      4         16
2      4         16
3      7         49
.      .          .
```

(e)
```
> cm1 <- lm(dist ~ speed + I(speed^2) + I(speed^3),data=cars)
> par(mfrow=c(2,2)) ## 4 plots in one
> plot(cm1)
```



First and third plots suggest that constant variance is not quite right. QQ-plot reflects this (but is no where near bad enough to suggest a problem on its own). It might be slightly better to fix this, but for now we'll push on.

(f)
```
> summary(cm1)
...
Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept) -19.50505   28.40530  -0.687    0.496
speed         6.80111    6.80113   1.000    0.323
I(speed^2)   -0.34966    0.49988  -0.699    0.488
I(speed^3)    0.01025    0.01130   0.907    0.369
...
> cm2 <- lm(dist ~ speed + I(speed^2) + I(speed^3)-1,data=cars)
> summary(cm2)
...
Coefficients:
            Estimate Std. Error t value Pr(>|t|)
speed       2.299945   1.802672   1.276    0.208
I(speed^2) -0.038399   0.209557  -0.183    0.855
I(speed^3)  0.003638   0.005871   0.620    0.539
...
> cm3 <- lm(dist ~ speed + I(speed^3)-1,data=cars)
> summary(cm3)
...
Coefficients:
            Estimate Std. Error t value Pr(>|t|)
speed      1.9751471  0.3250123   6.077 1.91e-07 ***
I(speed^3) 0.0025727  0.0008204   3.136  0.00292 **
---
...
```

So it could be that the braking distance dependence on speed is even worse than the physics suggests — we have ended up selecting a cubic, rather than quadratic dependence of distance on speed.

(g) Assume we can still interpret β_1 as relating to 'thinking distance'. Distances are in feet, and speeds in mph. To see how to convert, first consider converting all the distance units to feet. To convert speed into feet per hour we would have to multiply by 5280 (feet per mile). $\hat{\beta}_1$ would consequently be divided by 5280 (otherwise the model predicted distances would change). The newly scaled $\hat{\beta}_1$ would now have units of hours. To convert to units of seconds we have to multiply the scaled $\hat{\beta}_1$ by 3600 (seconds per hour). So, to convert the original $\hat{\beta}_1$ to units of seconds, as desired, we have to multiply by 3600/5280. Hence the estimated reaction time is $1.975 \times 3600/5280 = 1.35$ seconds. Similarly to get the 95% CI compute the CI for $\hat{\beta}_1$ and apply the same unit conversion factor

```
> beta1 <- coef(cm3)[1]
> sig1 <- vcov(cm3)[1]^.5
> t.crit <- qt(0.975,df=48)
> c(beta1-t.crit*sig1,beta1+t.crit*sig1)*3600/5280
   speed    speed
0.901136 1.792246
```

– a caveat however. The interpretation of β_1`speed` as the thinking distance was based on the original physically based model. Having selected a slightly different model it is not clear that the same reasoning really holds. Perhaps the cubic dependence effects thinking distance as well, for example due to psychological factors we did not think of.

2. A 95% frequentist confidence interval is supposed to include the true value of a parameter with probability 0.95, where the probability is taken over an infinite series of replications of the data gathering and inference process. For a correct linear model, with p parameters and n data, a 95% interval for a parameter β_i is $\hat{\beta}_i \pm t_{n-p}(.975)\hat{\sigma}_{\hat{\beta}_i}$, where $\hat{\beta}_i$ is the parameter estimate, and $\hat{\sigma}_{\hat{\beta}_i}$ is its estimated standard error. $t_{n-p}(.975)$ denotes the value below which a t_{n-p} random variable lies with probability 0.975. This question examines the coverage probability of such intervals by simulation. That is we simulate data with known true parameter values, and then see how well our statistical methods do at making inferences about them.

(a) The following code simulates data from the model $y_i = 0.5 + x_i + 10x_i^2 + \epsilon_i$ where the x_i are uniformly distributed predictor variables, and $\epsilon_i \sim N(0, 0.3^2)$.

```
n <- 100 ## sample size
b.true <- c(.5,1,10) ## true parameter values
ct <- qt(.975,n-3) ## critical points for CIs
cp <- b.true*0 ## coverage probability array
n.rep <- 1000  ## number of replicates to run
for (i in 1:n.rep) {
  x <- runif(n) ## simulated covariate
  mu <- b.true[1] + b.true[2]*x + b.true[3]*x^2
  y <- mu + rnorm(n)*.3 ## simulated data
  m1 <- lm(y~x+I(x^2))  ## fit model to this replicate
  b <- coef(m1)  ## extract parameter estimates
  sig.b <- diag(vcov(m1))^.5 ## and standard errors
  ## accumulate count of how often intervals include
  ## true value...
  cp <- cp + as.numeric(b-ct*sig.b <= b.true & b+ct*sig.b >= b.true)
}
cp/n.rep ## observed coverage probability
```

Read through the code to make sure that you understand what each line is doing. Then run the code to see how close the observed coverage probability is to the nominal coverage of 0.95. You can cut and paste the code from the pdf version of this sheet.

(b) What happens if the y_i data have the same dependence of the expected response on x, but the y_i are Poisson deviates? That is we simulate y_i, not as above, but using `y <- rpois(n,mu)`. Modify your coverage probability loop so that the response is Poisson, but the confidence intervals and their coverage is computed as before. How does the observed coverage probability compare to the nominal 0.95 now? Why do you think this might be?

(c) Given the results from the previous part, we might try bootstrapping as an alternative for computing the intervals. The simplest way to do this is to resample the x and y data together as pairs. For example the following code snippet performs `nb` bootstrap resamples, fitting a linear model to each and storing the fitted coefficients in the rows of a matrix `B`.

```
for (j in 1:nb) {
  bi <- sample(1:n,n,replace=TRUE)
  yb <- y[bi]
  xb <- x[bi]
  m1 <- lm(yb~xb+I(xb^2))
  B[j,] <- coef(m1)
}
```

The `quantile` function can be applied to each column of `B` to compute the confidence intervals. Using this code, or otherwise, write code to find the observed coverage probabilities of bootstrap confidence intervals for the linear model with Poisson response, as in part (b). This will involve nested loops – an outer one simulating new x, y data, and an inner one doing the bootstrap re-sampling. Make sure that your code is doing something sensible with small values of `n.rep` and `nb` before setting it to run with `nb=400` and `n.rep=1000`. The bootstrap interval should give more reasonable coverage.

Solution

(a) On running the code I got

```
[1] 0.959 0.951 0.952
```

Of course random error (often termed *Monte Carlo* error) will mean that your answer may be a little different, but the key point is that the coverages are close to nominal.

(b)
```
n <- 100
b.true <- c(.1,1,10)
cp <- b.true*0
n.rep <- 1000
```

```
for (i in 1:n.rep) {
  x <- runif(n)
  mu <- b.true[1] + b.true[2]*x + b.true[3]*x^2
  y <- rpois(n,mu)
  m1 <- lm(y~x+I(x^2))
  b <- coef(m1)
  sig.b <- diag(vcov(m1))^.5
  cp <- cp + as.numeric(b-ct*sig.b < b.true & b+ct*sig.b > b.true)
}
cp/n.rep
[1] 0.997 0.972 0.930
```

So now the coverages are far from nominal (0.997 is really very far from 0.95, for example). This happens because a Poisson random variable does not have constant variance. The variance increases with the mean, which undermines the theoretical variance calculation for the model coefficients.

(c) The final part is a bit more involved (and takes 5-10 minutes to run). Here is one possible solution...

```
cp <- b.true*0
nb <- 400;n <- 100
B <- matrix(0,nb,3)
for (i in 1:n.rep) {
  x <- runif(n)
  mu <- b.true[1] + b.true[2]*x + b.true[3]*x^2
  y <- rpois(n,mu) ## replicate Poisson data
  for (j in 1:nb) { ## bootstrapping loop
    bi <- sample(1:n,n,replace=TRUE)
    yb <- y[bi]
    xb <- x[bi]
    m1 <- lm(yb~xb+I(xb^2)) ## model fit to bs rep
    B[j,] <- coef(m1) ## store resulting parameter estimates
  }
  for (j in 1:3) { ## get the 3 bs CIs for this rep
    ci <- quantile(B[,j],c(.025,.975))
    if (ci[1]<=b.true[j]&&ci[2]>=b.true[j]) cp[j] <- cp[j] + 1
  }
  if (i%%50==0) cat(".") ## just to let you know it's doing something
}
cp/n.rep
[1] 0.944 0.946 0.945
```

This is obviously a great improvement on computing intervals using theory for which the assumptions did not hold.