COMP6235 -- Foundations of Data Science

This is a tutorial on some functions in R for COMP6235 Foundations of Data Science -- part III. This follows on from part II of the tutorial.

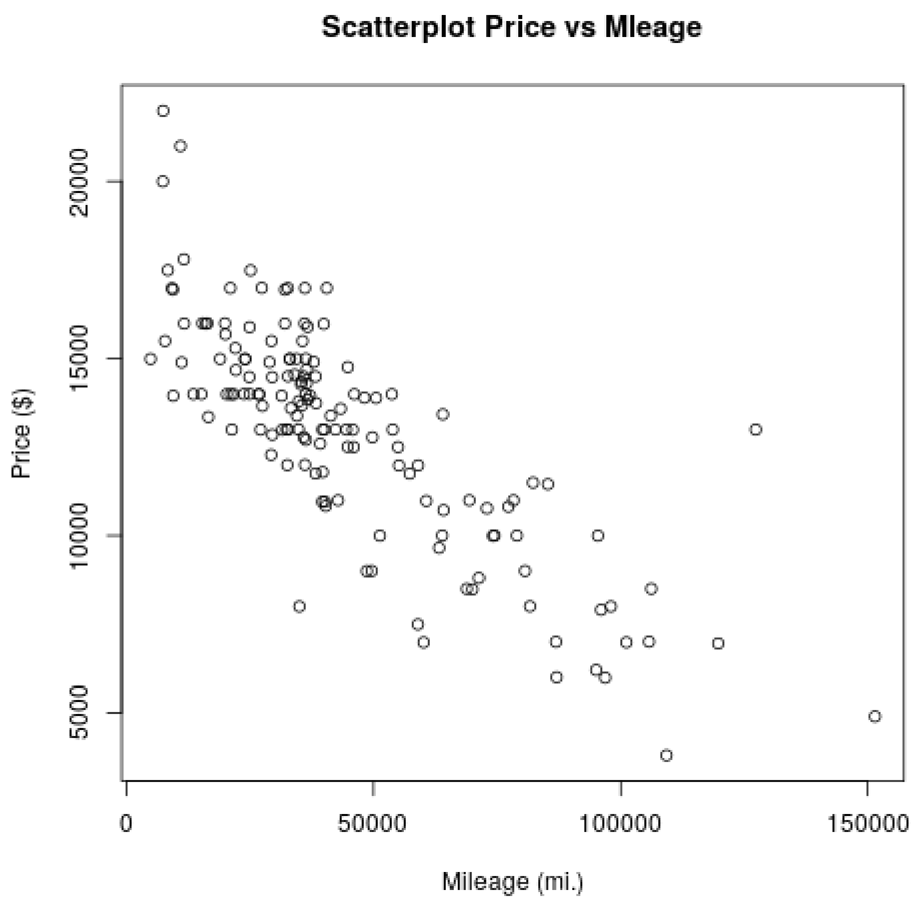
Fitting statistical models

In this section we are moving onto a more advanced statistical concept: the idea of fitting a model. Fitting a model generally means asking the question: "How well can I explain a given set of data with a simple model that relates the variable I am interested in to the other variables?". We'll start with simple models and make build on model complexity as we go.

Fitting a simple linear model

To illustrate the concept, let's return to the used cars data set we explored before. You might remember that we found a negative relationship/correlation between the price of the car and mileage.

```
> plot (x=usedcars$mileage, y=usedcars$price, main="Scatterplot Price vs Mleage", xlab="Mileage (mi.)", ylab="Price ($)")
```



which we had before approximated by a linear relationship (which I estimated by just looking at the cloud of points). The underlying model behind this reasoning was the hypothesis that on average the price of a car decreases by -0.10$ per mile it has been driven. Can we be more systematic about this? R makes it easy to look at the degree to which a theory like this is supported by the data. The relevant command is "lm" which means fitting a linear model to the data. The easiest way to understand the fitted linear model is to think about trying to draw a straight line on a graph that stays as close as possible to all of the data points of mileage/price of car points in our data set. In fact, R minimizes the square distance of the model from the data points or -- in other words -- carries out linear regression.

In the following, we will give the fitted model a name (lm1) (for "linear model #1"). We build it as follows:

```
> lm1 <- lm (price ~ mileage)
> summary (lm1)
```

Our output should look similar to the following

```
Call:
lm(formula = price ~ mileage)

Residuals:
Min 1Q Median 3Q Max
-5830.9 -1113.7 40.5 992.6 7782.4

Coefficients:
Estimate Std. Error t value Pr(>|t|)
(Intercept) 1.709e+04 2.915e+02 58.63 <2e-16 ***
mileage -9.329e-02 5.629e-03 -16.57 <2e-16 ***
---
Signif. codes: 0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 1854 on 148 degrees of freedom
Multiple R-squared: 0.6499,Adjusted R-squared: 0.6475
F-statistic: 274.7 on 1 and 148 DF, p-value: < 2.2e-16
```

The output from the summary of this linear model might look confusing at first. We don't have the time to make this a full course in statistics, so we'll have to skip many of the details. Here are the elements we need to consider:

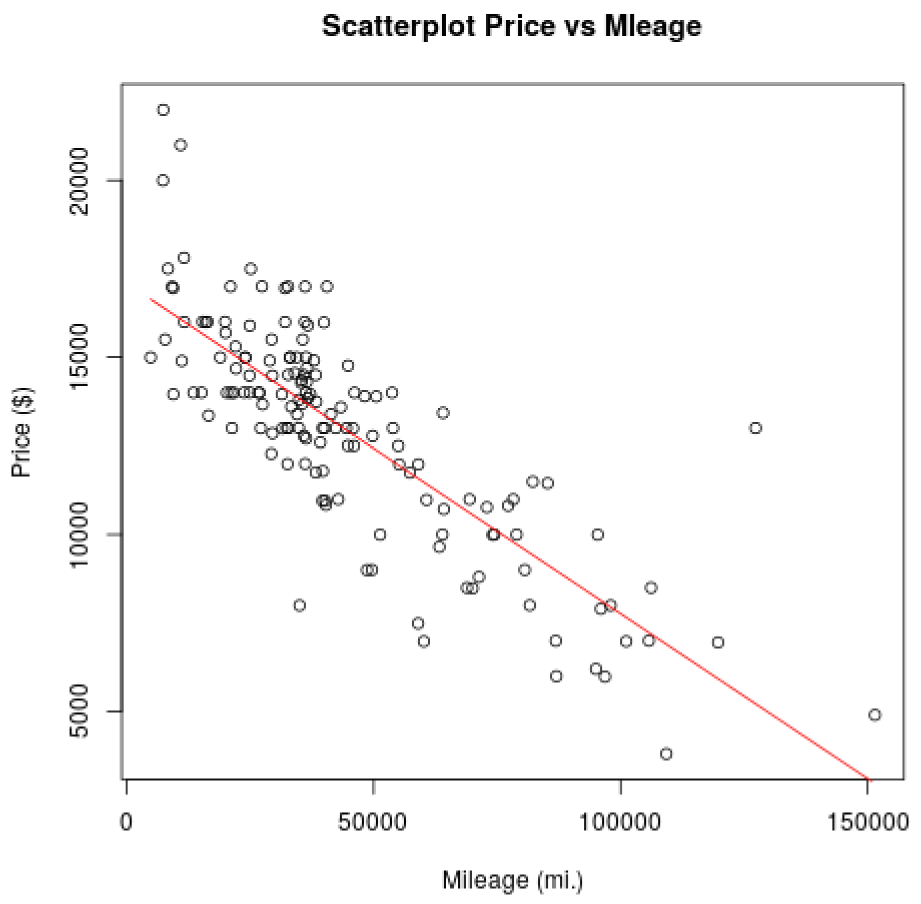
- The estimated intercept and mileage coefficients in the middle of the output. These numbers describe the equation for the regression line we have just fitted. The value of "1.709e+04" for the intercept can basically be interpreted as the average price of a new car (i.e. a car that has driven zero miles), roughly $17000. The mileage coefficient "-9.329e-02" gives the decrease in price of an average car per mile driven, i.e. roughly -0.10$ per mile, close to what I estimated by just looking at the data before.
- The "Adjusted R-squared" value on the right of the second last line. This number is always between 0 and 1 and gives the proportion of the variation in price that can be accounted for by mileage, i.e. around 65% in our case. To get some idea of the scale of these values, they are equivalent to taking a correlation coefficient and squaring it. So the value of 0.65 should be interpreted much like a correlation coefficient of 0.80: a fairly strong relationship.
- The "p=value" associated with the F-statistic, on the right side of the last line. P-values are probably the most misunderstood concept in statistics. They are best explained by considering that most statistical tests ask the question "If in fact the variables I am looking at are unrelated, how much of a fluke would it be to get results as weird as

these?". In other words, what is the probability of something that supports the model at least as much as the current data set does, if there is actually no relationship between the variables in the model? Very small p-values mean that the data would be unlikely to be observed in the case of no relationship, and thus they are generally taken as evidence in support of the model. Larger p-values (i.e. larger than about 0.01 or 0.05 in practice) are often taken as evidence against the model, as the data are not giving us strong reasons to doubt the no-relationship hypothesis.

In this case we find a p-value: < 2.2e-16, i.e. the linear model assumption works very well.

Because the fitted linear model has been stored in the R-object "lm1" we can use this object ot obtain a plot of the regression line and judge for ourselves what sort of fit has been achieved. The command to do this is as follows:

```
> lines (predict(lm1) ~ mileage, col="red")
```



Fitting polynomials

The fit achieved by the linear model of the price-mileage relationship is good, but does not explain all of the variation in the data. We could probably do much better if we were not limited to straight lines. R makes it simple to fit a polynomial model to the data, i.e. a model that employs a prabola (or higher order polynomial relationship) instead of the straight line. We can use the same "lm" command. (Of course, if this were real analysis, it would be strongly desirable that we have some theoretical reason for believing that a polynomial model might apply here before going ahead.) The command is:

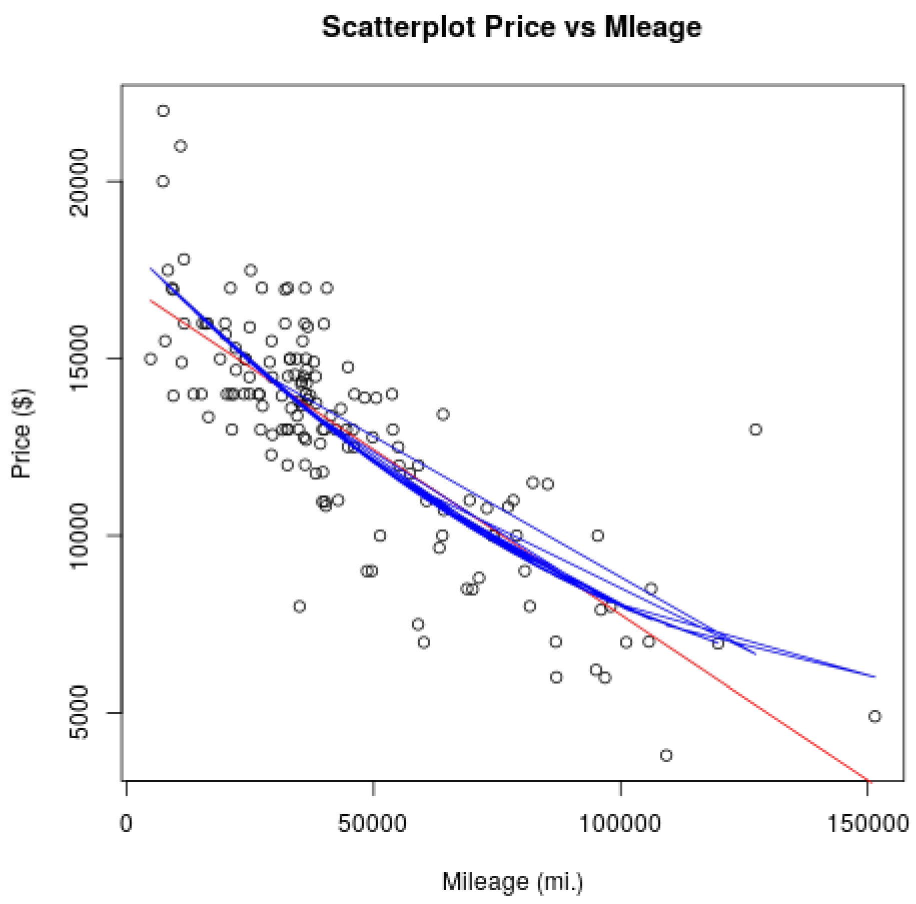
```
> lm2 <- lm ( price ~ poly (mileage, 2))
```

Note that we are fitting a polynomial model of degree 2, which just means that we are looking at both mileage and mileage^2 as predictors of price. You can try "summary (lm2)" to see the details of the fitted model and you will notice that we have slightly improved our adjusted R-squared value in this case (0.6649 for the polynomial model vs. 0.6475 for the linear model).

If the previous plot window is still open, we can add another relationship:

```
> lines (predict (lm2) ~ mileage, col="blue")
```

and obtain the following plot:



What went wrong here?

Actually, this is quite a pain and has something to do with the way how R plots the predicted line and how our data set is sorted. Namely, our data set is not sorted by mileage and so R just steps through the data points in the order in which they are in the data set, predicts, and connects the dots. This did not matter for a straight line (because the many lines effectively drawn by R all overlap), but it matters for any "curved" relationship. How to get around it?

Well, my best guess is to start from the beginning and build a new sorted data frame:

```
> df <- data.frame (x=sort(mileage), y=price[order(mileage)])
```

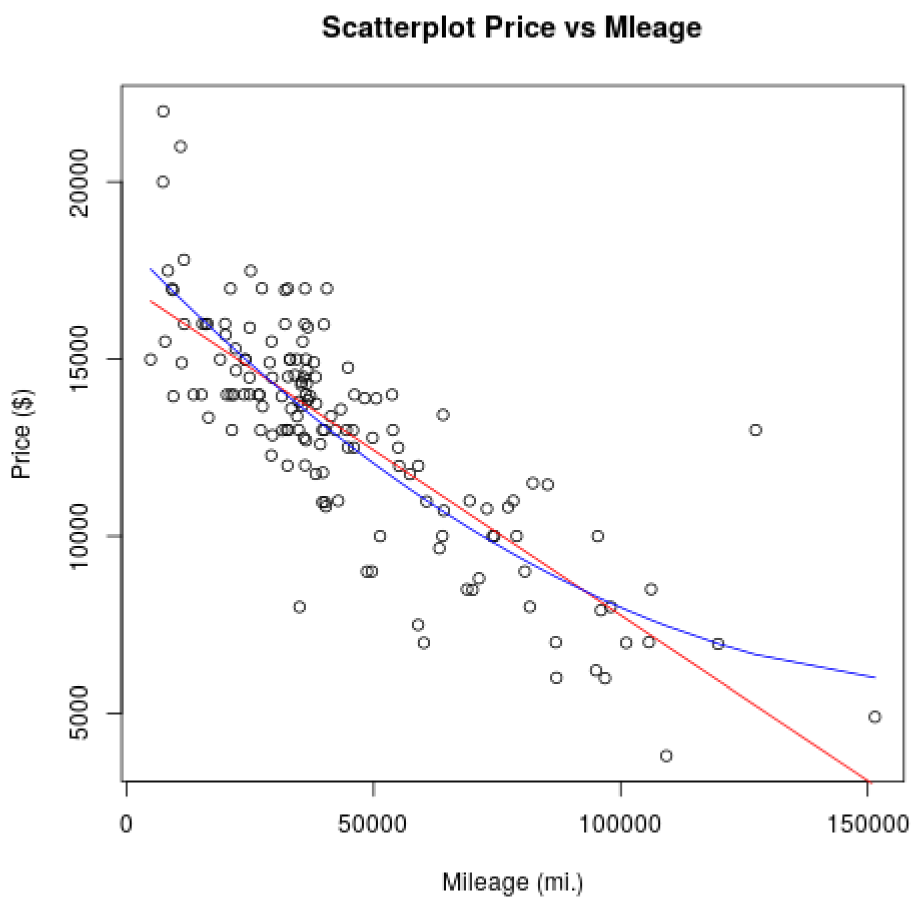
which now contains the sorted mileage variables and the corresponding prices. I then again build our linear models

```
> lm1 <- lm ( df$y ~ df$x)
> lm2 <- lm ( df$y ~ poly (df$x, 2))
```

and plot everything:

```
> plot (df$y ~ df$x, main="Scatterplot Price vs Mleage", xlab="Mileage
(mi.)", ylab="Price ($)")
> lines (predict (lm2) ~ df$x, col="blue")
> lines (predict (lm1) ~ df$x, col="red")
```

with a slightly more pleasing result:



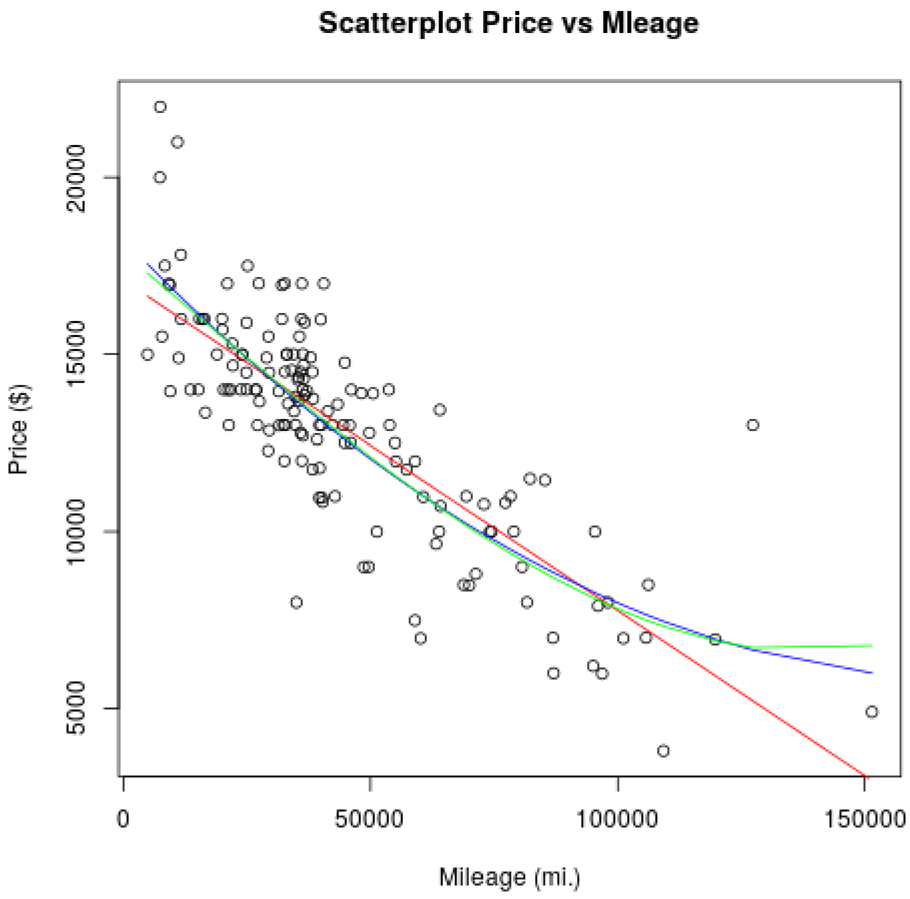
Looking hard we can see that the blue line on the graph gives a slightly better fit to the data than the red line. We should be sceptical, however, about the idea that we've found the "true story" behind these data. The blue line gives a parabola and would (if we extended the x-range for long enough) predict that the trend that car prices decrease as mileage increases reverts at some stage.

No need to stop at degree 2 polynomials. With the following commands, we can fit an even more flexible curve and add it to the plot:

```
> lm3 <- lm ( df$y ~ poly (df$x, 3))
```

which we can add to our plot as a green line via:

```
> lines (predict (lm3) ~ df$x, col="green")
```

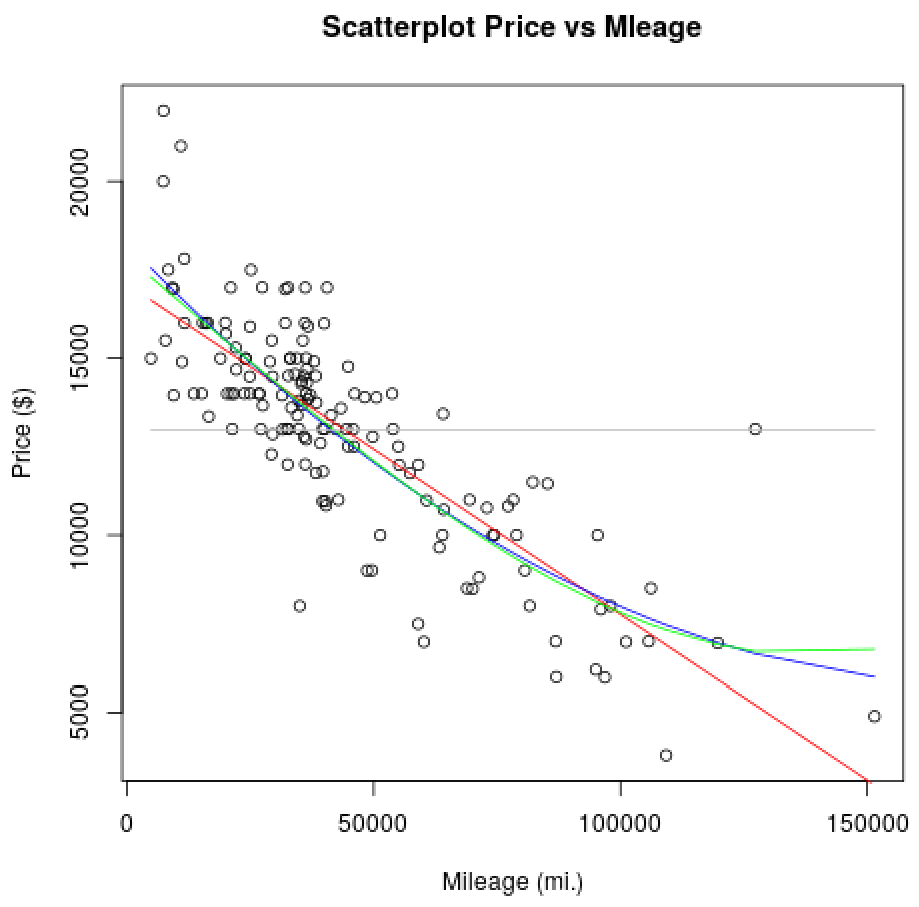


Notice that the green line doesn't seem to be much of an improvement over the blue line, however.

Choosing between alternative models

Now that we have lm1, lm2, and lm3, a natural question arises as to which one is the best model. And in order to settle this question, we actually need to introduce one more model into the contest

```
> lm0 <- lm ( df$y ~ 1)
```



This final, rather boring, model is shown in gray in the graph. The call to lm asks R to build a model of price (aka df$y) with nothing but the integer one as a predictor. Thus the model fitting algorithm has nothing to go on, and all it can do is predict that the value of price will always be equal to its mean: that's why the gray line is flat. This sort of model is often called the null model, for obvious reasons.

There are various ways of determining which model is best, e.g. stepwise model reduction . A statistician named Akaike devised a very elegant way of determining which model should be preferred in cases like ours. He developed a measure, known as Akaike's Information Criterion , or AIC for short, that can be used to rank alternative models. AIC rewards a model for the amount of variation it explains, so that the blue and green curves in our case should score highly. However, AIC also penalizes a model for including too many variables, so that especially the green curve will suffer here, whereas the gray curve might do well. So you see that AIC embodies a trade-off between prediction and simplicity. You don't need to know how to calculate AIC, as R will do it for you. All you need to remember at this stage is that smaller is better. The R command to compare our four models is as follows:

> AIC (lm0, lm1, lm2, lm3) and we find that the lowest score is obtained for the model with polynomial degree 2 (i.e. 2680.587), hence the polynomial model is the most reasonable of the four models considered.

(Note: If you have done stats before, you might wonder why the more long-established criterion of analysis of variance was not used here. You can use this in R via the anova () command, however, the AIC method is more universal and does a better job of capturing scientific parsimony and is probably easier to learn to use.)

A note on linear models for categorical data

The lm command in R is rather powerful and can be used to fit models to categorical data as well. Let us return to the used cars data set which contains various categorical fields. For the moment, we might be interested in the field "model" which has the three levels "SE", "SES" and "SEL". We might be interested in figuring out whether the car model has an effect on price and would also want to consider an appropriate null model to check of our model fit is meaningful:

```
> flm0 <- lm ( price ~ 1)
> flm1 <- lm ( price ~ model)
> summary (flm1)
```

Amongst other stuff, we'll obtain the following output:

```
Coefficients:
Estimate Std. Error t value Pr(>|t|)
(Intercept) 12202.1 320.3 38.095 < 2e-16 ***
modelSEL 3907.1 671.2 5.821 3.53e-08 ***
modelSES 492.0 515.7 0.954 0.342
---
Signif. codes: 0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 2829 on 147 degrees of freedom
Multiple R-squared: 0.1902,Adjusted R-squared: 0.1792
F-statistic: 17.27 on 2 and 147 DF, p-value: 1.837e-07
```

Note, that the adjusted R-squared value is not very impressive, i.e. only a very small portion of the variation can be explained by car model type, but the p-value indicates that this model might be significant. Of primary interest in the above are the coefficients. The first coefficient corresponds to the reference level -- in this case R has chosen "SE" and gives the average price for a car of that type. The other coefficients are relative to the reference level, i.e. have to be added to the reference level. Thus, a "SEL" car has an average price of 12202+3907 and an SES car 12202+492 $. R also automatically carries out t-tests if the coefficients are significant and here we'd notice that the coefficient for SES is not significant -- hence we could not be sure is SES cars are indeed more expensive than SE cars. In contrast, our confidence that SEL cars are more expensive than SES cars would be high.

As a last step we might want to figure out if our model is sensible at all, by carrying out an AIC analysis. Using

```
> AIC (flm0, flm1)
```

we find that the AIC score for flm1 is indeed lower than the score for flm0, hence the preferred model is flm1.
We could also reverse the logic here and attempt to predict a categorical variable using a continuous(ordinal) variable as a predictor. In this case we end up in the territory of logistic regression (and related models). To learn more about regression models for categorical variables, you might also want to have a look at the corresponding R-command "glm ()".

Scripts in R

If you find yourself doing the same thing in R over and over you should switch from interactive mode to batch mode, i.e. you should write a script (effectively a short R program) in order to do the repetitive operations in R for you.Here is an example script that loads the usedcars data set and plots the relationship between price and mileage into a pdf called example.pdf. In order to run scripts in R you can do either of two things:

- The first is that you can run R from a command line environment (e.g. from a linux system) using:

  ```
  R --vanilla < script.txt
  ```

  The "vanilla" option is to specify that R should not save the workspace, etc. and the "<" symbol (in a linux environment) means that input to R should come from the named text file. You could use "R --vanilla < script.txt > output.txt" if you want the output of R to be redirected to a particular file (here output.txt) instead of being shown on the terminal.

- The other way of executin R scripts is within the R environment by using the R-command "source ()", i.e.:

  ```
  >source ("script.txt")
  ```

Programming in R

Functions

R has the flexibility to allow for user-defined functions. The typical syntax for defining a function is given below:

```
myfunction <- function(arg1, arg2, ... ){
statements
return(object)
}
```

where "myfunction" is the name of the function, "arg1, arg2, ..." is a list of inputs to the function which is enclosed in brackets. The body of the function starts with "{" and ends with "}". Then a list of statements or operations follows, ending by a return statement specifying which object is to be returned (Actually: return is not needed, R will return whichever variable is in the last line of the function.)

Let's say we want to build a function that returns the mean and the standard deviation of a sample in one go. We could do this as follows:

```
> meansd <- function (sample) {
+ df <- list (center=mean(sample), spread=sd (sample))
+ return (df)
+ }
```

Note that R continues lines with the "+" sign once you press enter after an opening brace "{" and will change back to standard input mode after the next closing brace "}". In our function we assign the name "meansd". You can later retrieve the source code of this function by typing the name of the function without "()", i.e. if you type "meansd" R will display the corresponding source code. Next, we define a data structure which will enclose all of the results to be returned. In our case, a list seems appropriate and we create two fields called "center" and "spread" within it. The last statement in the function's body returns our list of results. Let's test it:

```
> sample1 <- rnorm (100, 0, 1)
> sample1
```

i.e. I create 100 random variates from a normal distribution with mean 0 and standard deviation 1. Now I can use our convenient way of calculating mean and standard deviation in one go:

```
> meansd (sample1)
```

and obtain the following output

```
$center
[1] -0.01405578

$spread
[1] 0.9821303
```

i.e. not surprisingly fairly close to the parameters I used to generate the random variates.

Control structures

R allows for all the usual types of control structures typically present in programming languages. For example, you can use conditional expressions following the syntax:

```
if (cond) expr
if (cond) expr1 else expr2
```

Use "for loops" following

```
for (var in seq) expr
```

Use "while loops" following

```
while (cond) expr
```

and others like "switch" or "ifelse" etc. Check the R documentation for more detail.

Let's suppose we want to build a function which first checks the input data structure if it is a vector and then calculates the population version of the standard deviation of our sample. We could do it as follows:

```
mysd <- function(x) {
if (!is.vector(x)) {
```

```
warning("argument is not a vector!")
return (NA)
}
s=0.
s1=0.
for (i in 1:length(x)) {
s=s+x[i]
s1=s1+x[i]*x[i]
}
s=s/length(x)
s1=s1/length(x)
s=sqrt(s1-s*s)
return(s)
}
```

As a last step, let's test this function. For this purpose I generate 1000 normally distributed random variates from a normal distribution with mean zero and variance 1.

```
> x<- rnorm (1000,0,1)
> mysd(x)
```

for my particular random seed I get a value of 1.037462 which is not too bad.