MATH5743M: Statistical Learning

Dr Seppo Virtanen, School of Mathematics, University of Leeds

Semester 2: 2022

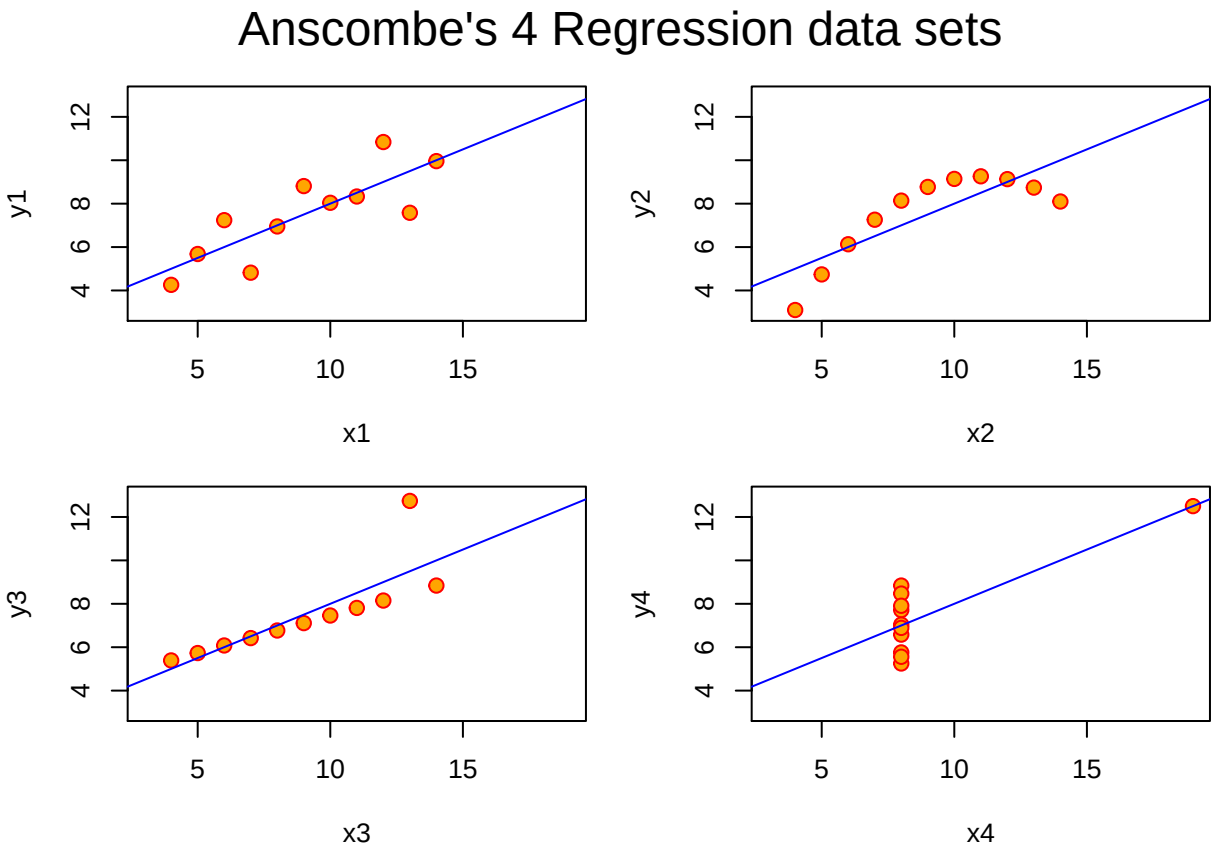
Week 5: Model Selection

So far we have considered situations how to estimate the parameters of a model that we have chosen. But how do we choose the right model for our data? Firstly we need to consider whether we are dealing with a regression problem or a classification problem. If we are dealing with a regression problem we might choose to use a linear regression model. If instead we are dealing with a classification problem we might choose to use a logistic regression model. A large part of model selection is down to us, the statistical analyst to decide. That (hopefully!) is why you are taking this course, to give you the training necessary to make that decision. In statistics nothing can ever be done completely automatically: the analyses we perform and the answers that we get from the data always depend on the assumptions that we make and on the existing knowledge that we bring to the problem. This is always worth remembering: it is impossible to do statistics without making assumptions. Statistics is always about quantifying rigorously what is implied by both the data and our own prior knowledge.

But as well as using our expertise and training, we can use data to help us decide in more detail what our statistical model should be. After we have chosen which type of model is most appropriate, we still need to make further choices about the model structure. Specifically how many and which inputs should we use? Every statistical model makes choices about which data to include or ignore. We previously learnt how we could use linear models with more than one input variable, but we also learnt that as we use more inputs we can run into difficulties. We can combine our own intuition with statistics to decide what our model should include.

Exploratory model selection

One important reason to do model selection is that simply applying one's favourite model to a new data set may give sensible looking answers, but these may in fact be misleading if the model is a bad representation of the data. Take the famous example of 'Anscombe's Quartet': four data sets that give almost identical results when using linear regression, despite looking extremely different to the human eye.

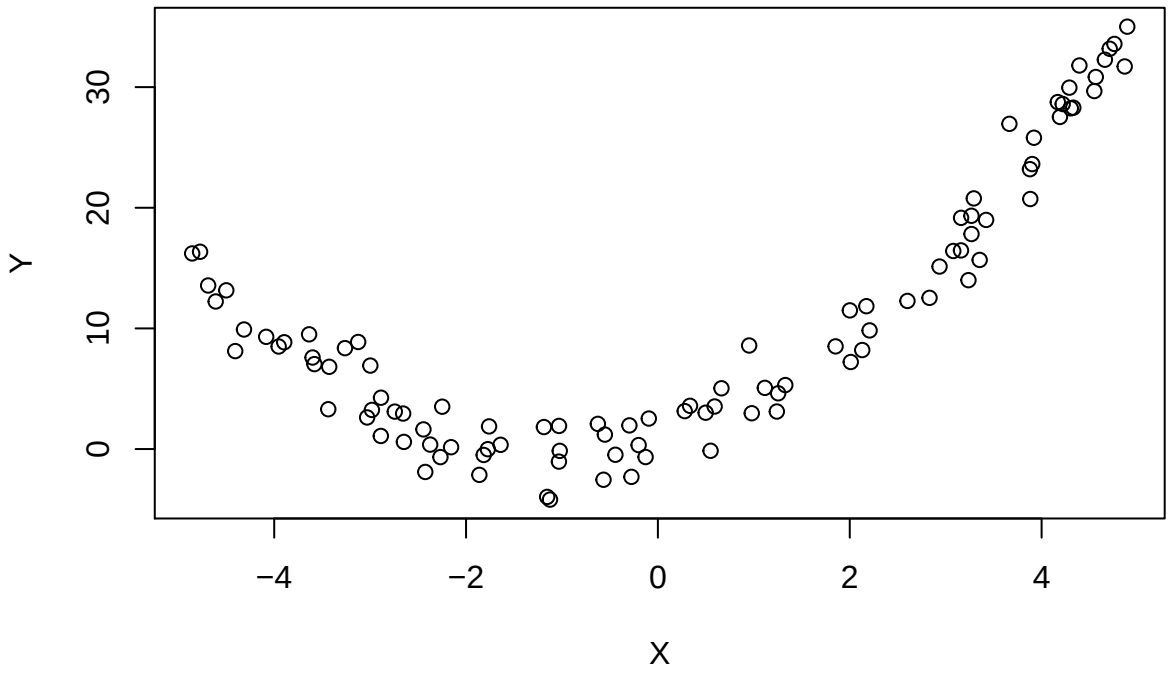


Feature selection

Deciding which inputs to use in a model is described as *feature selection*: we are selecting 'features' of the data that are important for predicting the output we are interested in. Feature selection is a highly active area of research. For prediction, stock market traders would love to be able to identify what factors are important for predicting tomorrow's stock prices. In science, researchers often use feature selection to determine whether one thing causes another from observational data (though we have already learnt to be careful when inferring causation!). In computer science research often focuses not just on selecting existing features from the data, but creating new ones! When we discussed General Linear Models we saw how we could use transformations of the original inputs as new inputs. Computer scientists are interested in what the best form of these transforms are, so that they can efficiently process data ready for analysis. We will learn more about this later in the course when we discuss neural networks. For now we will assume that we want to identify which of the inputs already in the data are useful for predicting the output.

The problem of overfitting

Imagine you are presented with the data shown in the plot below:



Clearly Y varies in a structured way as X changes, and the relationship between the two is non-linear. The points look like they lie along a fairly smooth curve, so you might consider fitting a General Linear Model, using polynomial transformations of X as extra inputs in a linear regression. The curve looks quite like a quadratic, but it could also be cubic or even quartic; how should you decide which to use?

In fact I generated this data using the following commands

```
X = runif(100)*10 -5
Y = 1 + 2*X + X^2 + rnorm(100, 0, 2)
```

So the true relationship is quadratic. Lets see what happens when we try to fit a simple linear regression, a quadratic model and a cubic model.

```
mydata = data.frame(Y = Y, X1 = X, X2=X^2, X3 = X^3)
model1 = glm(Y ~ X, data = mydata)
model2 = glm(Y ~ X + X2, data = mydata)
model3 = glm(Y ~ X + X2 + X3, data = mydata)
```

We can compare the likelihood of each model evaluated at the maximum-likelihood estimate parameters:

```
logL1 = logLik(model1)
logL2 = logLik(model2)
logL3 = logLik(model3)

print(paste("log-likelihood for linear model: ", logL1, collapse = ""))
```

```
## [1] "log-likelihood for linear model:  -346.615003359587"
```

```
print(paste("log-likelihood for quadratic model: ", logL2, collapse = ""))
```

```
## [1] "log-likelihood for quadratic model:  -206.925949487068"
```

```
print(paste("log-likelihood for cubic model: ", logL3, collapse = ""))
```

```
## [1] "log-likelihood for cubic model:  -206.615069914414"
```

So we can see that although the data is actually generated from a quadratic model, a cubic model actually produces a closer fit to the data (albeit only a by a small amount). This is a very common issue in statistical learning and is called 'over-fitting': more complicated models typically produce closer fits to the data you use to learn the model parameters than less complex ones. This seemingly goes against our expectations about the properties of a good model. We usually want our models to be as simple as possible, while still capturing the important features of the data. The philosophical principle of Occam's Razor states that we should favour simpler models because these tend to be better descriptions of reality than highly complicated models.

The reason why complicated models produce better fits even when the reality is simpler is quite easy to grasp in the polynomial model fit above. Consider first the quadratic model:

$$y = \beta_0 + \beta_1 x + \beta_2 x^2 + \epsilon \tag{1}$$

Compare this to the cubic model. We'll use the term α for the coefficients of this model to show they are different to the quadratic model,

$$y = \alpha_0 + \alpha_1 x + \alpha_2 x^2 + \alpha_3 x^3 + \epsilon \tag{2}$$

If we were to set $\alpha_3 = 0$ in the cubic model, these two models would be identical. Assume we know the coefficients $\hat{\beta}_0, \hat{\beta}_1, \hat{\beta}_2$ that maximise the likelihood for the quadratic model. We can always get at least the same likelihood value for the cubic model by setting $\alpha_0 = \hat{\beta}_0$, $\alpha_1 = \hat{\beta}_1$, $\alpha_2 = \hat{\beta}_2$, $\alpha_3 = 0$. Therefore the maximum likelihood of the cubic model must be at least as large of the maximum likelihood for the quadratic model, and will usually be greater, i.e.

$$\max \mathcal{L}_{\text{cubic}} \geq \max \mathcal{L}_{\text{quadratic}} \tag{3}$$

Therefore we cannot use the model likelihood alone to select which inputs to use in our model, since the likelihood will always increase if we add more inputs.

Model selection by penalised likelihood

One way to avoid the problem of overfitting is to change the criteria by which we judge how good a model is. So far we have focused on the maximum likelihood, a measure of how well the model is able to fit to the available data, when the parameters are chosen to make this fit as good as possible. The problem, as shown above, is that with more parameters we can almost always tweak the values to make the likelihood just a little higher. In the example above, the fit for the cubic model is only a little better, in terms of likelihood, than the true quadratic model. We might, qualitatively, think 'that little bit of extra likelihood isn't worth having an extra parameter'. How do we make that idea quantitative? One way is to use a *penalised likelihood* that explicitly 'punishes' the extra parameters. Usually this takes the form of an extra term, in the log-likelihood, proportional to the number of parameters k, with a penalisation strength λ:

$$\log L_{\text{PENALISED}} = \log L^* - \lambda k, \tag{4}$$

where $\log L^*$ is the usual maximum log-likelihood. Unfortunately there is no absolute consensus on how to choose λ. It may depend on your personal experience or needs regarding model complexity. However, probably the most commonly used penalised likelihood is known as the Akaike Information Criterion (AIC). This is defined as:

$$\text{AIC} = 2k - 2\log L^* \tag{5}$$

where the logarithm is the natural logarithm base e. This effectively sets λ to be equal to one, and the model that *minimises* the AIC is deemed to be the best.

Other similar examples of penalised likelihoods include the Bayesian Information Criterion (also known as the Schwarz Criterion), which sets $\lambda = \log(n)/2$, n being the number of data points. You may also encounter Wilk's theorem, which is used to decide if the likelihood difference between two models is statistically significant, and which only applies if the models are strictly nested (i.e.if one contains all of the parameters and inputs of the other). For our purposes we will stick to the Akaike Information Criterion, but it is useful to be aware that other methods are in use.

The AIC for a model fitted using **glm** is conveniently provided as part of the **summary** command.

An aside on AIC and other criteria

Where does the AIC come from? How does it differ from the other penalised likelihoods? This goes beyond the scope of this module, but to briefly summarise for those who are interested: Hirotugu Akaike derived the AIC by considering which of a set of different models would make the best predictions about future data. Importantly, he assumed that the 'real' model that generated the data was not necessarily among those being evaluated, i.e. all the models were approximations of the truth. He found that the AIC would, making some further assumptions about the data, on average pick the model which would make the most accurate predictions. A different criterion, the Bayesian Information Criterion (BIC) is different in that it tries to find the model that is *most likely to be the truth*, and thus assumes that one of the models being tested is true. Finally, of those listed above, Wilk's theorem asks how likely it is that improvement in fit between a simpler model and a more complex model would be more than a given value, assuming that the extra factors in the more complex model actually had no effect in reality. This is a reminder that in statistics it is always very important not only to remember the details of how to perform an analysis, but also to be very precise and clear about what question you are trying to ask, and what analysis you are choosing to answer that question, and why.

Cross-validation

An alternative method for avoiding problems of overfitting is to test how well a model can predict *unseen* data, i.e. data that has not been used to determine the model parameters. This is called *cross-validation*: we use the unseen data to validate the model that has been fitted. This is an intuitively sensible methodology, since the goal of statistical learning is often to develop a model that is useful for prediction. For this reason it is the gold standard method for testing models developed in machine-learning and computer science. If you read an academic paper in these fields that develops a new statistical model, you will generally find at some point a large table detailing the predictive accuracy of the new model against other state-of-the-art models that have been previously developed.

Cross-validation follows a simple process:

- Identify or create two subsets of the data. Label one the 'training' data and the other the 'test' data.
- Use the training data to fit the statistical model you want to test, i.e. to learn the parameters of that model
- Use the fitted model to predict the outputs of the test data, based on the inputs of the test data
- Evaluate the accuracy of those predictions

- Potentially repeat the process with two new data subsets

Selecting the training and test data

When formally testing models for academic publication it is common to use standard, accepted data sets, which are already split into training data and test data. We can also create these different data sets ourselves by splitting an existing data set in various ways.

- Assign each data point to the training or test set with some probability by a coin toss
- Randomly select a fixed number of data points to be in the test set and place the others in the training set
- Select all data collected before some cut-off time to be in the training set and those after to be in the test set

An example

Consider a case where we have one output variable and three potential input variables, and we want to create a linear regression model based on a given data set that will allow us to make good predictions in the future. We don't know which of the input variables are actually important for predicting the output, so we'll try to learn this by doing cross-validation

To demonstrate I'll create some artificial data where we know for sure what the real answer is, so we can test whether cross-validation gives us the right answer. I'll create three inputs, X1, X2 and X3, and an output Y that is a linear sum of two of these plus some random residuals taken from a normal distribution

```
X1 = runif(200)-0.5
X2 = runif(200)-0.5
X3 = runif(200)-0.5
Y = 2*X1 + 1*X3 + rnorm(200, 0, 1)
mydata = data.frame(X1, X2, X3, Y)
```

Now I'll select 100 of the 200 data points at random to form my training set. The remainder will form the test set. I'll use an R convention where x[-idx] means 'select all points in X *except* idx'

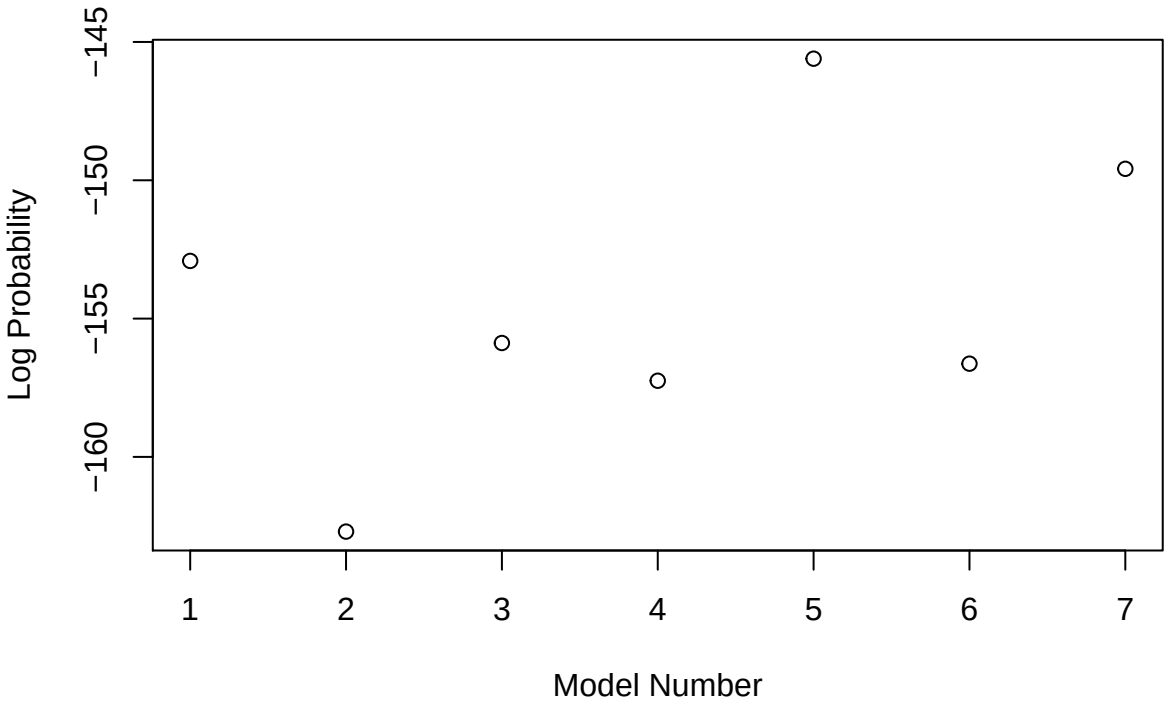
```
idx = sample(1:200, 100) #sample 100 points in 1...200 without replacement
train_data = mydata[idx, ]; test_data = mydata[-idx, ]
```

Now we need to try each possible combination of the inputs we want to consider. With three possible inputs there are 7 possible combinations of inputs (if we exclude the possibility of having no inputs):

1. X1
2. X2
3. X3
4. X1 & X2
5. X1 & X3
6. X2 & X3
7. X1, X2 & X3

What we'll do next is to create a code loop that implements the fit-predict procedure for cross-validation and store the quality of the prediction. Just as we use the (log-)likelihood when assessing the quality of fit, we'll use the predicted (log-)probabilities of the actual test outputs as our measure of prediction quality.

```
#List all possible models (you could do this automatically with
#a bit of string manipulation)
formulas = c("Y ~ X1", "Y ~ X2", "Y ~ X3",
             "Y ~ X1 + X2", "Y ~ X1 + X3", "Y ~ X2 + X3",
             "Y ~ X1 + X2 + X3")

predictive_log_likelihood = rep(NA, length(formulas))
for (i in 1:length(formulas)){
  #First fit a linear regression model with the training data
  current_model = glm(formula = formulas[i], data = train_data)

    #Extract the 'dispersion parameter' from the model - recall this is the unbiased estimate for the r

  sigma = sqrt(summary(current_model)$dispersion)

  #Now use this model to evaluate the probability of the test outputs
  #Get the predicted mean for each new data point

  ypredict_mean = predict(current_model, test_data)

    #Now calculate the predictive log probability by summing the
  #log probability of each output value in the test data
  predictive_log_likelihood[i] = sum(dnorm(test_data$Y, ypredict_mean, sigma, log=TRUE))

}
plot(1:length(formulas), predictive_log_likelihood,
     xlab="Model Number", ylab="Log Probability")
```



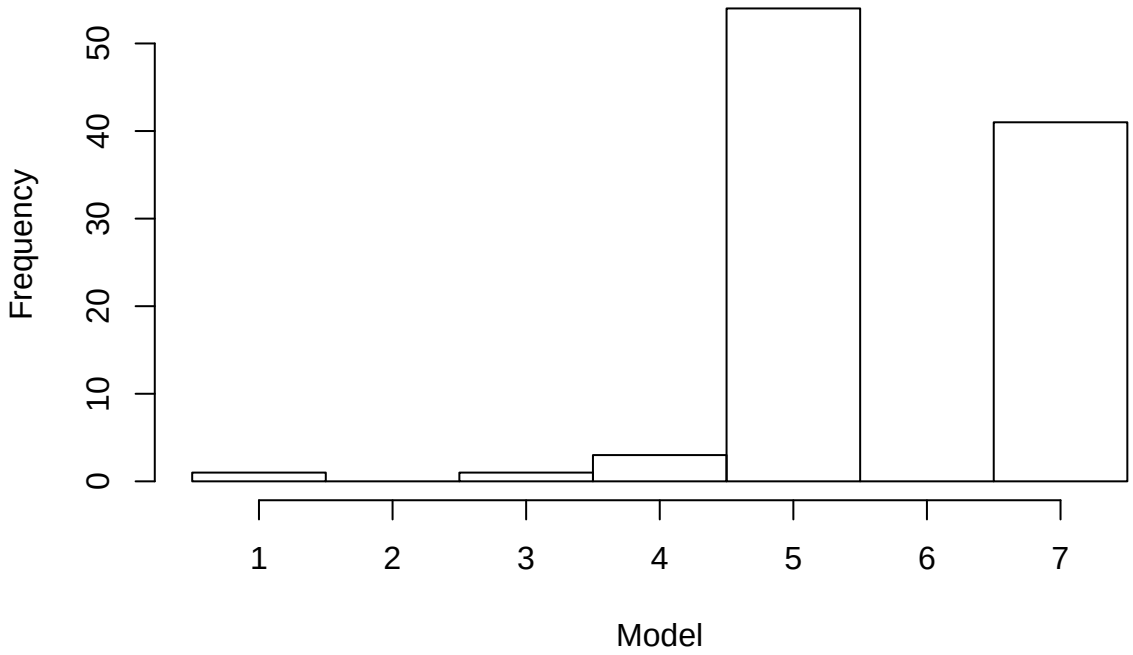
In this example we can see that model 5 has the highest log-probability for the test outputs. This is good because model 5 is 'Y ~ X1 + X3', which corresponds to the data we created. Note that not only does this model outperform models which don't include X1 and X3, it also outperforms model 7 which uses X1, X2 and X3. Thus we have avoided overfitting the model by including input variables that are not actually important.

It is important to be aware that there is no guarantee that cross-validation will always identify the correct model. For one thing, in real life it is unlikely that the real situation perfectly corresponds to any of the models we might test. Also, there is always a chance that the wrong model may still predict better on a particular data set. It is only when the amount of data we have becomes very large that we can be sure we will end up with the right model. But we can make cross-validation more robust by reducing the sensitivity to the particular split we made between training and test data. A simple way to do this is to repeat the whole procedure many times, choosing a new random split in the data each time.

What I'll do now is to repeat the whole process above 100 times (I'm choosing 100 at random here!). Each time I'll record which model 'wins' - i.e. which model has the best predictive log-likelihood, as model 5 did above. I'll record for each model the number of times it 'wins' and plot that as a bar chart. This will give us a good idea of how robust the result above is.

```
#Note that all the formulas are still defined from above!

#Repeat previous cross-vaidation exercise 100 times...
winner = rep(NA, 100)
for (iteration in 1:100){

#Make a new random training data - test data split
idx = sample(1:200, 100) #sample 100 points in 1...200 without replacement
train_data = mydata[idx, ]
test_data = mydata[-idx, ]

predictive_log_likelihood = rep(NA, length(formulas))
```

```r
for (i in 1:length(formulas)){
  #First fit a linear regression model with the training data
  current_model = glm(formula = formulas[i], data = train_data)

  #Extract the 'dispersion parameter' from the model - recall this is the unbiased estimate for the res

  sigma = sqrt(summary(current_model)$dispersion)

  #Now use this model to evaulate the probability of the test outputs
  #Get the predicted mean for each new data point

  ypredict_mean = predict(current_model, test_data)

   #Now calculate the predictive log probability by summing the
  #log probability of each output value in the test data

  predictive_log_likelihood[i] = sum(dnorm(test_data$Y, ypredict_mean, sigma, log=TRUE))

}

#Find the winning model for this iteration

winner[iteration] = which.max(predictive_log_likelihood)
}

#Plot a histogram of how often each model wins
hist(winner, breaks = seq(0.5, 7.5, 1), xlab='Model', ylab='Frequency', main='')
```



Notice two things: (i) overall model 5 is favoured but (ii) there are many occasions on which model 7 wins. So in the first example I got lucky in finding the right model. Generally we need quite a few cross-validation

data splits before we can be confident to select the right model. How many? That question is too difficult to answer completely here (and indeed there is no perfect answer). But in general, the more splits you can do the better, so do as many as you can that time and the power of your computer will allow. You should also look at AIC as well. As with everything in statistics, the more different perspectives you can get giving the same answer, the more confident that you can be with that answer.

Extra: Validation data

So far we have considered splitting the data into two parts: the training data and the test data. We use the training data to fit our model, and we use the test data to decide which model generalises best, i.e. which model is best at predicting new data. We might try many different splits of the data to get a more robust decision about which model to select. Having selected your model, you might now want to report how good it is. As I noted earlier in this part of the course, this is how research in machine-learning is typically carried out: (i) Researchers develop a new type of model. (ii) They use training data to fit the parameters of their model. (iii) They use cross-validation with a test data set to decide which features they should include in the model, or if they have several possible models, which they should use. Now, for publication they typically want to report the performance of their model, for other researchers to see. In this case it is good practice to keep *another* portion of the data aside, called the *validation* data. Then they will (iv) perform a final prediction of the validation data using the model they have selected, and report the prediction accuracy.

Why can't we just report the accuracy of predictions on the test data? The reasons are quite subtle mathematically, but the essence of the problem is this: you have selected your model specifically to predict your test data as well as possible. Therefore it may be biased towards data sets that look like that test data. Just we can't use predictions of the training data to select a model because we used that data to pick parameters, we shouldn't use the test data to determine the reported predictive accuracy of the model, because we used that data to make our model selection.