MATH5743M: Statistical Learning

Dr Seppo Virtanen, School of Mathematics, University of Leeds

Semester 2: 2022

Week 4: Multiple Linear Regression

Last week we looked at linear regression models in one dimension. We started by defining the likelihood, the probability of observing a given set of outputs, conditioned on known input values and a particular choice of model parameters: the intercept and gradient values. We used optimisation techniques to infer the maximum-likelihood estimates for the model parameters. We then learnt about using the **glm** command in R to efficiently perform model fits and predictions.

This week we will extend our knowledge of linear regression to encompass multiple linear regression, where there may be more than one input value used to predict the output. We will also see how this can be used to model non-linear patterns in the data, by using transformations of the inputs known as *basis functions* as new inputs.

The linear model

In multiple linear regression the output, y, is assumed to be determined by a weighted, linear sum of several inputs, $x_1, x_2, \ldots, x_n$, and a normally-distributed random residual, ϵ

$$y_i = \beta_0 + \beta_1 x_{i1} + \beta_2 x_{i2} + \ldots + \beta_n x_{in} + \epsilon_i, \ \epsilon_i \sim \mathcal{N}(0, \sigma) \tag{1}$$

The likelihood

Assuming that we have some collected data to analyse, in the form of some inputs and outputs, we can write down the log-likelihood for the model parameters (the coefficient values) in a manner analogous to the one-dimensional case. I'll use both the summation form and the matrix form:

$$\begin{aligned} \log \mathcal{L}(\beta_0, \beta_1, \ldots, \beta_n, \sigma) &= \log P(\text{Data} \mid \beta_0, \beta_1, \ldots, \beta_n \sigma) \\ &= \sum_{i=1}^{N} \log \mathcal{N}(y_i; \beta_0 + \sum_{j=1}^{n} \beta_j x_{ij}, \sigma^2) \end{aligned} \tag{2}$$

Multiple linear regression in R

Performing multiple linear regression in R uses exactly the same process as in the one-dimensional case. We continue using the **glm** function. The only difference is that our data frame must contain all the (multiple)

inputs that we want to use, and we need to specify the names of all of these inputs in the formula that we provide. Lets illustrate with an example. First, I will create some simulated data where an output y is a linear function of three different inputs, x_1, x_2, x_3, with some random normally distributed residuals. The values of the inputs are drawn randomly between zero and one

```
x1 = runif(100)
x2 = runif(100)
x3 = runif(100)
y = 1*x1 + 2*x2 + 3*x3 + rnorm(100, 0, 0.1)
```

Now I make a data frame with these variables as columns

```
my_multiple_data = data.frame(y, x1, x2,x3)
```

Now use **glm** to perform the regression, specifying in the formula that I want to use all three inputs

```
my_multiple_model = glm(y ~ x1 + x2 + x3, data=my_multiple_data)
```

Finally we print a summary of the fitted model

```
summary(my_multiple_model)
```

```
##
## Call:
## glm(formula = y ~ x1 + x2 + x3, data = my_multiple_data)
##
## Deviance Residuals:
##       Min         1Q     Median         3Q        Max
## -0.210606  -0.058321  -0.006042   0.068937   0.248191
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) -0.03443    0.03598  -0.957    0.341
## x1           1.03981    0.03854  26.982   <2e-16 ***
## x2           2.02769    0.03666  55.309   <2e-16 ***
## x3           3.02346    0.03511  86.113   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for gaussian family taken to be 0.009303453)
##
##     Null deviance: 85.75851  on 99  degrees of freedom
## Residual deviance:  0.89313  on 96  degrees of freedom
## AIC: -178.03
##
## Number of Fisher Scoring iterations: 2
```

As you can see, the output of the model summary looks nearly identical to the one-dimensional cases we looked at last week. The only differences are that the 'Call' element records that we used a formula with several inputs, and we now have estimates and standard errors for an intercept value and for coefficients of each of the inputs - these closely match the actual values I used to generate the data, as you would expect.

Confidence intervals

When someone says 'these closely match', as I have done above, you should ask yourself what 'closely match' means. By now you should be thinking like statisticians. Are the true values that I used to generate the data consistent with the estimated values? To answer this question we should look at our uncertainty in those estimates, as specified by the confidence intervals. Last week we learnt how to construct a 95% confidence interval for a regression parameter value in the one-dimensional case. To do so for multiple dimensions we follow a very similar procedure. However, we must take to obtain the right number of degrees of freedom. Recall that for one-dimensional regression with n observations we had $n-2$ degrees of freedom, because that data had been used to estimate two parameters before the standard error. In the case of multiple linear regression we have fewer degrees of freedom, one fewer for each input. So if we have a model with 3 inputs as above, we will have $n-4$ degrees of freedom (the extra one is for the intercept). As an example, lets construct the 95% confidence interval for the x1 coefficient. We can extract the summary table of coefficients and standard errors like this:

```
summary_table = summary(my_multiple_model)$coefficients
print(summary_table)
```

```
##                Estimate Std. Error    t value     Pr(>|t|)
## (Intercept) -0.03442773 0.03597593 -0.9569656 3.409884e-01
## x1           1.03981225 0.03853705 26.9821417 1.317015e-46
## x2           2.02768640 0.03666097 55.3091385 1.296069e-74
## x3           3.02345947 0.03511054 86.1125812 1.063875e-92
```

We use R to find the appropriate critical t value via the **qt** function (which gives quantiles of the t distribution). Here we had 100 observations, so $n-4=96$ is the number of degree of freedom. The quantile we need is 0.975 (i.e. 2.5% lies beyond this)

```
t_critical = qt(0.975, 96)
```

The estimate of the coefficient and the standard error are extracted from the table:

```
estimate = summary_table[2, 1]
sterr = summary_table[2,2]
```

We now construct the interval as: estimate $\pm\ t_{2.5\%,96}\times$ standard error

```
interval_min = estimate - t_critical*sterr
interval_max = estimate + t_critical*sterr
print(paste(c('Min: ', interval_min), collapse=""))
```

```
## [1] "Min: 0.963316806794049"
```

```
print(paste(c('Max: ', interval_max), collapse=""))
```

```
## [1] "Max: 1.11630770237561"
```

You can see that this interval contains the true value of 1, thus showing that the estimate is consistent.

Basis functions for non-linear regression

After spending the last two weeks on various forms of linear regression, you might be wondering why we assume that everything in the world is linear. Surely not every real relationship is so simple. What do we do when the world is... non-linear?

One reason we learn about and use linear regression is that the process of fitting this model is extremely well understood, and the results are readily interpretable. We also do so because we can use the framework of linear regression to build more complex models. As we shall see over the coming weeks, a great many common statistical models can be seen as variations on the theme of linear regression.

One way that we can adapt linear regression is by the use of *basis functions*. These are transformations of the original inputs through some non-linear functions, to create new inputs for use in a (multiple) linear regression problem. Lets look at an example to get a better idea.

In nuclear physics, radioactive elements split into new elements or transform into new isotopes of the same element. This is called radioactive decay. Each radioactive atom will decay at random, with a fixed probability per unit time. This means that *on average* the number of radioactive atoms in any sample will decay exponentially, with a characteristic *half life* τ – the length of time it takes for half of the remaining atoms to decay. This half life is now very well known for most materials such as uranium-235 (used for nuclear energy) or carbon-14 (used for carbon dating). Decaying atoms produce radiation. We can estimate the remaining number of radioactive atoms by measuring the radiation produced by the material; this should also half on average over the time τ. Of course, there will typically be some error in our measurement of this radiation. We could therefore model the amount of radiation, r, that we record as a function of time, t, like this:

$$r = r_0 \exp(-t/\tau) + \epsilon,\ \epsilon \sim \mathcal{N}(0, \sigma) \tag{3}$$

Assume we are dealing with a sample of carbon-11, the half life of which is 1.22×10^3 seconds. We don't know exactly how much radioactive material we have now, but we can make measurements of the radiation. and we'd like to estimate what the level of radiation will be in 3 hours, so we can know whether the sample will be safe. We can solve this as a regression problem, using a new basis function. Define our basis function as:

$$f(t) = \exp(-t/1220) \tag{4}$$

where t is measured in seconds. Our model above now becomes

$$r = r_0 f(t) + \epsilon,\ \epsilon \sim \mathcal{N}(0, \sigma) \tag{5}$$

Now lets pretend that we take some measurements of the radiation every 10 minutes over the next hour and see the following results:

```
## [1] "radiation_data"
```

```
##           r    t
## 1 99.522692    0
## 2 61.100674  600
## 3 39.196337 1200
## 4 25.727056 1800
## 5 14.103496 2400
## 6  7.454127 3000
## 7  6.866051 3600
```

Now lets form a new input variable using the basis function we chose, and add this to the data frame:

```
radiation_data$f = exp(-radiation_data$t/1220)
```

We can now use this new input just as if it was something we had measured. If you look at the model equation, you should be able to see that it looks exactly the same as a one-dimensional regression problem,

where f is the input, and r_0 is a regression coefficient. Thus if we perform linear regression we can estimate the value of r_0:

```
my_radiation_model = glm(r ~ f, data=radiation_data)
summary(my_radiation_model)
```

```
##
## Call:
## glm(formula = r ~ f, data = radiation_data)
##
## Deviance Residuals:
##       1        2        3        4        5        6        7
## -0.3635  -0.4190   1.1387   2.0170  -0.8327  -2.1167   0.5763
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)    1.125      0.848   1.326    0.242
## f             98.761      1.776  55.604 3.56e-08 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for gaussian family taken to be 2.23569)
##
##     Null deviance: 6923.482  on 6  degrees of freedom
## Residual deviance:   11.178  on 5  degrees of freedom
## AIC: 29.142
##
## Number of Fisher Scoring iterations: 2
```

You can see we obtain as usual intercepts and regression coefficients. But if you look again at the model equation, you will see there is no intercept term. We would like to be able to tell the regression this in advance; remember that a key fact in statistics is that the more we can tell the statistical model ourselves, the less it needs to learn from the data – this makes it more accurate in estimating the remaining quantities. In this case we can tell it that the intercept value should be zero (or that there should be no intercept term), which we do with a small trick in the way we specify the formula by adding '-1' at the end

```
my_radiation_model_no_intercept = glm(r ~ f - 1, data=radiation_data)
summary(my_radiation_model_no_intercept)
```

```
##
## Call:
## glm(formula = r ~ f - 1, data = radiation_data)
##
## Deviance Residuals:
##      Min        1Q    Median        3Q       Max
## -1.14207  -0.68176   0.04646   1.60797   2.74012
##
## Coefficients:
##   Estimate Std. Error t value Pr(>|t|)
## f  100.518      1.256   80.01 2.57e-10 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
```

```
## (Dispersion parameter for gaussian family taken to be 2.518667)
##
##     Null deviance: 16137.908  on 7  degrees of freedom
## Residual deviance:    15.112  on 6  degrees of freedom
## AIC: 29.252
##
## Number of Fisher Scoring iterations: 2
```

We can see now, by looking at the regression coefficient for f, that the regression fitting procedure has estimated r_0 to be close to, but not exactly, the value of r we measured at time $t = 0$. This is sensible, since this measurement should obviously be close to r_0, but corrupted by some amount of measurement noise.

Our goal was to predict the level of radiation in 3 hours (or 10800 seconds). As we have done before for predictions, we can use the **predict** function to do this immediately. We need to create a new data frame that contains the value of our basis function at this new time.

```
new_t = 10800
new_f = exp(-new_t/1220)
new_radiation_data = data.frame(f = new_f)
predicted_radiation = predict(my_radiation_model_no_intercept, newdata=new_radiation_data)
print(predicted_radiation)
```

```
##          1
## 0.01437704
```

Polynomial regression

The example given above is extremely simple, and models a physical system that is extremely well understood. But often we don't know exactly what what basis functions we should use in a non-linear regression problem. In these cases a common approach is to model data as being a *linear sum* of polynomial basis functions, i.e.:

$$y = \sum_{i=0}^{p} \beta_i x^i + \epsilon,\ \epsilon \sim \mathcal{N}(0, \sigma) \tag{6}$$

In general such a model will use all polynomial terms up to some maximum order, p. Using a linear sum allows us to use all the tools of linear regression to estimate the coefficients $\beta_0, \ldots, \beta_p$.

Difficulties and dangers

I hope that the last two weeks have taught you that linear regression is a powerful method for analysing data, easily expandable by the use of additional inputs and basis functions, and straightforwardly implementable in R through the **glm** function. In fact, one danger is that this method can be applied so easily once you are familiar with the basic procedures in R that you might easily go ahead with an analysis of a data set before thinking about whether such a statistical model is likely to give you sensible results. Although linear regression is a powerful tool, and the statistical model which you should become most familiar with, there are potential factors with can make it difficult, or even dangerous to apply.

Highly correlated inputs

Consider the following toy problem. I know that ice cream sales tend to be higher on warm, sunny days in summer. I'm interested to find out whether people buy more ice creams because of the number of hours of sunlight, or because of the temperature. Of course, sunlight and temperature are related, so when there are many hours of sunlight it will also tend to be warmer. Lets pretend for the moment that people actually buy ice creams because of the temperature, which in turn is strongly predicted by the amount of sunlight. I'll create some simulated data from 20 days to illustrate

```
set.seed(55)
sunlight_hours = rnorm(20, 12, 3)
temperature = 10 + sunlight_hours + rnorm(20, 0, 1)
sales = 10 + temperature + rnorm(20, 0, 5)
icecream_data = data.frame(sunlight_hours, temperature, sales)
```

Now, pretending we don't know the true relationship, lets use a regression model to decide how much influence sunlight or temperature has on the sales.

```
icecream_model = glm(sales ~ sunlight_hours + temperature, data=icecream_data)
summary(icecream_model)
```

```
##
## Call:
## glm(formula = sales ~ sunlight_hours + temperature, data = icecream_data)
##
## Deviance Residuals:
##     Min       1Q   Median       3Q      Max
## -6.4564  -3.1500  -0.3761   3.2535   7.0245
##
## Coefficients:
##                Estimate Std. Error t value Pr(>|t|)
## (Intercept)    19.41075   12.16233   1.596    0.129
## sunlight_hours  1.10680    1.14160   0.970    0.346
## temperature    -0.08575    1.10523  -0.078    0.939
##
## (Dispersion parameter for gaussian family taken to be 20.45981)
##
##     Null deviance: 534.33  on 19  degrees of freedom
## Residual deviance: 347.82  on 17  degrees of freedom
## AIC: 121.88
##
## Number of Fisher Scoring iterations: 2
```

As you can see, in this case the regression model has wrongly attributed the effect on sales to the number of sunlight hours, rather than the temperature. But try repeating all of this process yourself several times, starting from the data simulation. You will probably find that the results you get are quite different each time, even though neither the simulation model, nor the regression model have changed.

This is what can happen when two inputs are strongly correlated to each other. Lets think about why. The regression model can only 'see' how probable the data is conditioned on possible relationships between the inputs and the outputs.

Overfitting

We've seen how easy it is to move from a one-dimensional linear regression to a multi-dimensional regression and/or a more complex non-linear regression. Additional inputs can be added extremely easily and the **glm** command will process them for us. Why then don't we just use lots of possible inputs in every model? Surely the more information we give the model, the better the predictions it will be able to make?

In fact, this is a dangerous mistake to make. Statistical models are vulnerable to what is called 'overfitting'. This is when a model has found all the useful information contained in some data and starts to fit not to the general patterns but to the noise that is always present in a real data set.

Consider a data set where there is no relationship between an input, x and an output y, and see what happens when you perform a linear regression with the goal of making predictions. First, create some data matching this situation

```
x = seq(0, 10, length.out=100)
y = rnorm(100, 0, 1)
mydata = data.frame(y, x)
```

Now perform the linear regression

```
mymodel = glm(y ~ x, data=mydata)
summary(mymodel)
```

```
##
## Call:
## glm(formula = y ~ x, data = mydata)
##
## Deviance Residuals:
##      Min        1Q    Median        3Q       Max
## -1.88014  -0.60693   0.01417   0.65553   2.25361
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) -0.42938    0.18465  -2.325   0.0221 *
## x            0.05857    0.03190   1.836   0.0694 .
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for gaussian family taken to be 0.8652099)
##
##     Null deviance: 87.707  on 99  degrees of freedom
## Residual deviance: 84.791  on 98  degrees of freedom
## AIC: 273.29
##
## Number of Fisher Scoring iterations: 2
```

As you can see, the model has estimated a non-zero regression coefficient for the effect of x and y. Now, we know that y is just normally distributed around zero, regardless of the value of x. But what happens if we try to use our fitted model to predict the value of y when $x = 1000$?

```
mynewdata = data.frame(x=100)
predicty = predict(mymodel, newdata=mynewdata)
print(predicty)
```

```
##        1
## 5.427169
```

The prediction is now around one standard deviation away from zero, which would be the actual best guess for y. This is because in any finite sample there will always be *some* non-zero correlation between the values of the inputs and outputs, and so the model will identify a best-fit coefficient which is slightly different from zero. Extrapolating this small coefficient away from the range of the data will then create large errors in prediction. The same process applies in the case of multiple inputs: if y really only depends on x_1, there will always be some small sample correlation of the residuals with another input x_2, so a model incorporating x_2 will attribute some small effect to this input, leading to errors in prediction.

We will learn more about overfitting and how to avoid it in Week 5 when we discuss Model Selection.