MATH5743M: Statistical Learning

Dr Seppo Virtanen, School of Mathematics, University of Leeds

Semester 2: 2022

Week 6: Logistic Regression and Generalised Linear Models

In the previous two weeks we have been studying the use of linear models for performing regression analyses. In this week's work we will move on from regression and consider the other half of supervised learning problems: classification. We will introduce ourselves to a new statistical model, logistic regression, and then take a look at the whole family of statistical methods known as the Generalised Linear Model, which gives the tool **glm** its name.

Logistic regression is a technique for performing binary classification – tasks where the output can only take two possible values. The terminology can be somewhat confusing here, and stems from differences in usage for *regression* and *classification* in statistics and machine-learning. For our purposes, despite the name, logistic regression is a classification method, for modelling the probability that an output belongs to one of two possible classes.

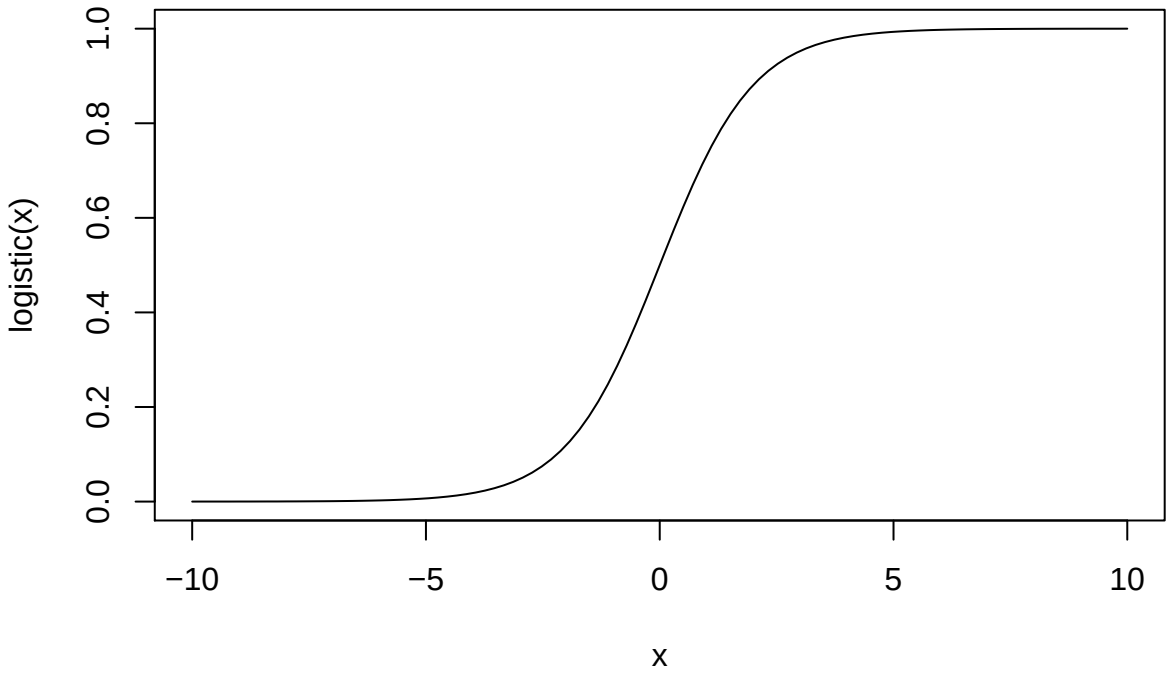
The logistic function

The logistic function, $\phi(x)$, is a *sigmoidal* function $\phi : \mathbb{R} \to [0, 1]$. That is, it takes in a real number and maps it to a position on the interval from 0 to 1. This is useful when modelling the probability of an output to belong to a particular class, since that probability must always be between zero and one. The functional form of ϕ is defined as:

$$\phi(x) = \frac{1}{1 + \exp(-x)} \tag{1}$$

Lets look at some of the properties of this function. When x is a large negative number, $\exp(-x)$ will be very large. So in this case $\phi(x)$ will be very small, tending towards zero. Conversely, when x is a large positive number, $\exp(-x)$ will be very small, and $\phi(x)$ will be very close to one. If $x = 0$ then $\exp(-x) = \exp(0) = 1$, so $\phi(0) = 1/2$. So, starting from a large negative value for x and increasing, we expect $\phi(x)$ to start near zero, to increase as x increases and pass through $1/2$ when $x = 0$, tending towards one as x becomes large. Lets look at the function plotted in R to verify this:

```
phi = function(x) 1/(1+exp(-x))
X = seq(-10, 10, length.out = 100)
Y = phi(X)
plot(X, Y, xlab="x", ylab="logistic(x)", type='l')
```



Now we can see why the logistic function is described as a *sigmoidal* function. The function has a distinctive S-shape; the term comes from the letter sigma, the Greek letter S.

The logistic-linear model in one-dimension

Given binary outputs Y and inputs X, the probability that $Y = 1$ is

$$P(Y = 1 \mid X = x) = \phi(\beta_0 + \beta_1 x) \tag{2}$$

and for $Y = 0$ we have

$$P(Y = 0 \mid X = x) = 1 - P(Y = 1 \mid X = x) = 1 - \phi(\beta_0 + \beta_1 x), \tag{3}$$

because Y can only take two values. We assume outputs follow the Bernoulli distribution.

The likelihood

Just as for linear regression, we will estimate the parameters of the logistic regression model by maximising the likelihood, that is finding the parameters that maximise the probability of generating the data we have seen. And just like with linear regression, we will work with the log-likelihood, which is easier to maximise because a product of independent probabilities is transformed into a sum of log-probabilities.

The log-likelihood is the sum of the log-probabilities for every data point. Each data point has an output value which is either 1 or 0. When an output is 1, we need to add $\log \phi(\beta_0 + \beta_1 x)$ to this sum. When an output is 0 we need to add $\log (1 - \phi(\beta_0 + \beta_1 x))$. The log-likelihood is written as

$$\log \mathcal{L}(\beta_0, \beta_1) = \sum_{i=1}^{n} Y_i \log \phi(\beta_0 + \beta_1 X_i) + (1 - Y_i) \log \left(1 - \phi(\beta_0 + \beta_1 X_i)\right). \tag{4}$$

Example

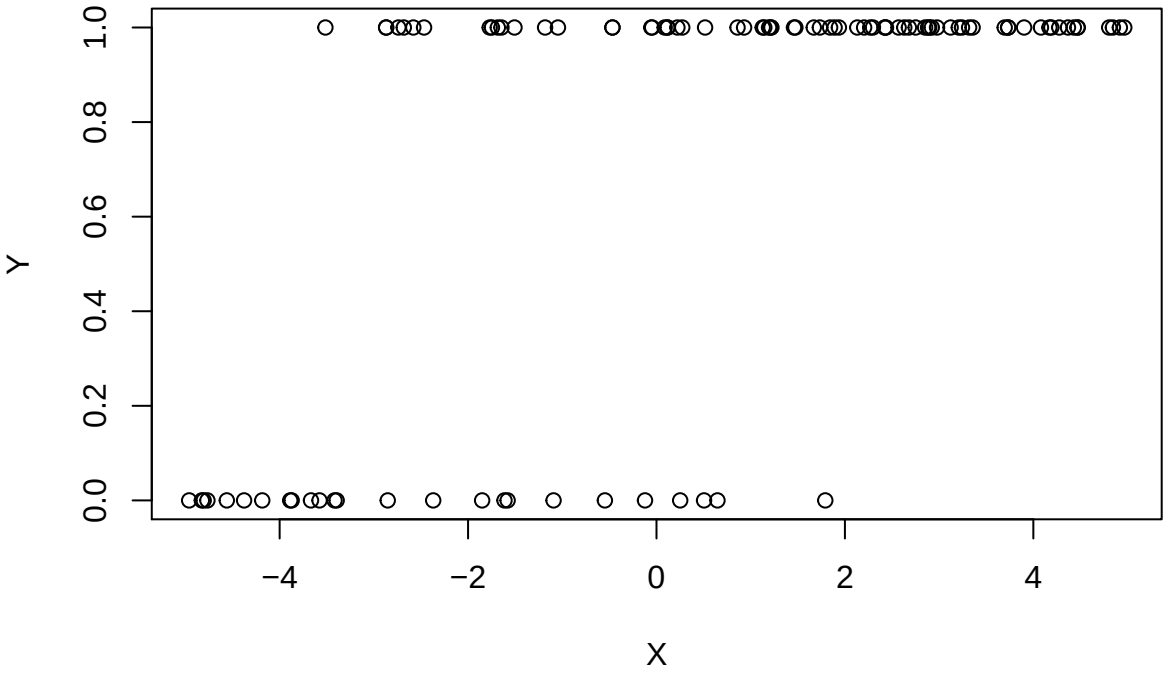
Unlike linear regression, logistic regression does not have a general analytic solution, so we will need to learn the appropriate parameters numerically. In the same way as we did for linear regression, we'll start by plotting what the log-likelihood looks like over a range of possible parameters, before using the numerical optimisation algorithms in R to get a more accurate solution.

First lets create some data to work with using a logistic-linear model with known parameters:

$$P(Y = 1 \mid X = x) = \phi(\beta_0 + \beta_1 x), \ \beta_0 = 2, \beta_1 = 1 \tag{5}$$

In R we'll take random samples of X between -5 and 5, then calculate the probability that each corresponding value of Y will be 1. Then we choose Y to be 1 or 0 according to that probability. Note that to generate Y values as being 1 or 0 with a given probability, we generate uniform random numbers between 0 and 1, and test whether these are less than the desired probability. If I generate a uniform random number this way, there is a probability of p that it will be less than p. In R, we could also use the rbinom function to generate values from the Bernoulli distribution.

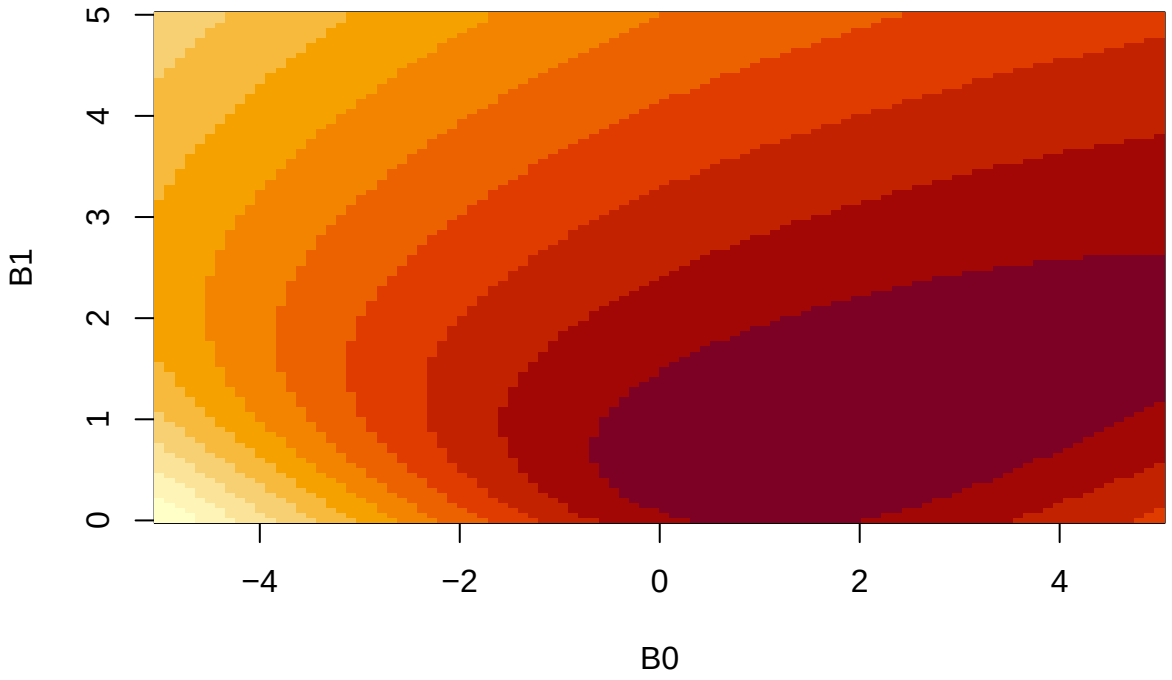
```
#Define number of data points
N = 100
#Define the logistic function phi
phi <- function(b0, b1, x) 1/(1+exp(-b0-b1*x))
#Generate random values of X between -5 and 5
X = runif(n=N, min=-5, max=5)
#Calculate the probability for each Y to be 1 based on X
P = phi(b0=2, b1= 1, x=X)
#Assign 1's or 0's to Y based on P
Y = as.numeric(runif(N) < P)
plot(X, Y)
```



Now we will try to use these generated data to rediscover the values of the parameters that we used to generate

it. We'll start by doing a search over lots of possible parameter values, just as we did for linear regression. But this time we will search for both the β_0 and β_1 parameters at the same time. That means we have to create a two-dimensional matrix of log-likelihood values, with each element of the matrix corresponding to a pair of parameter values:

```
#define the possible values of beta0 and beta1 to check.
#Note that it is a good idea to make the number of each different
B0 = seq(-5, 5, length.out=100)
B1 =seq(0, 5, length.out=101)
#define an empty matrix of log-likelihood values
LL = matrix(NA, 100,101)
#for every pair of B0 and B1, evalaute the log-likelihood
for (i in 1:100){
  for (j in 1:101){
    LL[i,j] = sum(Y*log(phi(B0[i],B1[j],X)) + (1-Y)*log(1-phi(B0[i], B1[j],X)))
  }
}
#show the results as an image
image(B0, B1, LL)
```




```
idx = which.max(LL)
#Trick to get the i and j values from the idx values
idx = arrayInd(idx, dim(LL))
print(paste("MLE estimate for beta 0 is", B0[idx[1]], collapse=""))
```

```
## [1] "MLE estimate for beta 0 is 1.56565656565657"
```

```
print(paste("MLE estimate for beta 1 is", B1[idx[2]], collapse=""))
```

```
## [1] "MLE estimate for beta 1 is 0.75"
```

Note that to test approximately 100 values of both β_0 and β_1 we had to try $100^2 = 10000$ different pairs of

parameters. As the number of parameters in a model grows this will clearly become impractical, and we will also find it harder to visualise any results. This is one reason why an exhaustive search over possible parameters in rarely used in real practical statistical modelling. But visualising the log-likelihood this way is a good pedagogical exercise as it gives us a feeling for how the parameters affect the fit of the model to the data.

We can also efficiently find the MLE parameter estimates using the **optim** function

```
#Define the negative log-likelihood function
negLL <- function(param) {-sum(Y*log(phi(param[1],param[2],X))
                               +(1-Y)*log(1-phi(param[1],param[2],X)))}
start_guess = c(0, 1)
optimResults = optim(par=start_guess, fn=negLL)
print(paste("MLE estimate for beta 0 is", optimResults$par[1], collapse=""))
```

```
## [1] "MLE estimate for beta 0 is 1.52479565928298"
```

```
print(paste("MLE estimate for beta 1 is", optimResults$par[2], collapse=""))
```

```
## [1] "MLE estimate for beta 1 is 0.727604793119088"
```

Most statistical packages include the capability to do logistic regression. The best way to do this in R, we first need to recognise that a logistic regression model is a particular case of something called a *Generalised Linear Model* (GLM). This is because although the model is non-linear, it includes a *latent* linear function: $\beta_0 + \beta_1 x$. GLMs are a wide class of models where a non-linear transformation (such as the logistic function) is applied to an underlying, latent linear function of the inputs in order to determine the probability of the outputs. We have already seen the simplest case: linear regression. Logistic regression is another type of GLM with a slightly more complicated set of features to understand.

GLMs can be efficiently used in R via the **glm** command, as we used for linear regression (you can do an awful lot of work in R with this one function!). Lets look again at the help file to see how to use this for logistic regression.

```
help(glm)
```

You can either see the relevant help file by running the above command on the R console or by checking out Figure 1.

As we've seen before, the first argument is a *formula* that specifies the relevant input and outputs in our model. In the GLM this formula specifies the latent linear function before it is transformed by the logistic function. The *family* argument is new: this specifies the type of GLM we will be using, and the default option of 'Gaussian' gives us standard linear regression. In the case of logistic regression this family should be *binomial*, because the likelihood for the logistic regression takes the form of a binomial probability: the outputs are either 1 or 0. Specifying this family will also automatically let **glm** know that we want to use the logistic function. Then, as with linear regression, we must specify the *data* in the form of a data frame with columns corresponding to the named inputs and outputs we are using. We can safely ignore the other arguments for now and leave them at their default values.

This is all best understood by an example. Lets place our previously used input and output values into a data frame and perform a logistic regression using **glm**.

```
#Make a data frame with the inputs and outputs
mydata = data.frame(X=X, Y=Y)
#Specify a glm
myglm = glm(Y ~ X, family=binomial, data=mydata)
summary(myglm)
```

Fitting Generalized Linear Models

Description

`glm` is used to fit generalized linear models, specified by giving a symbolic description of the linear predictor and a description of the error distribution.

Usage

```
glm(formula, family = gaussian, data, weights, subset,
    na.action, start = NULL, etastart, mustart, offset,
    control = list(...), model = TRUE, method = "glm.fit",
    x = FALSE, y = TRUE, contrasts = NULL, ...)

glm.fit(x, y, weights = rep(1, nobs),
        start = NULL, etastart = NULL, mustart = NULL,
        offset = rep(0, nobs), family = gaussian(),
        control = list(), intercept = TRUE)

## S3 method for class 'glm'
weights(object, type = c("prior", "working"), ...)
```

Arguments

`formula`	an object of class "`formula`" (or one that can be coerced to that class): a symbolic description of the model to be fitted. The details of model specification are given under 'Details'.
`family`	a description of the error distribution and link function to be used in the model. For `glm` this can be a character string naming a family function, a family function or the result of a call to a family function. For `glm.fit` only the third option is supported. (See `family` for details of family functions.)
`data`	an optional data frame, list or environment (or object coercible by `as.data.frame` to a data frame) containing the variables in the model. If not found in `data`, the variables are taken from `environment(formula)`, typically the environment from which `glm` is called.
`weights`	an optional vector of 'prior weights' to be used in the fitting process. Should be `NULL` or a numeric vector.
`subset`	an optional vector specifying a subset of observations to be used in the fitting process.
`na.action`	a function which indicates what should happen when the data contain `NA`s. The default is set by the `na.action` setting of `options`, and is `na.fail` if that is unset. The 'factory-fresh' default is `na.omit`. Another possible value is `NULL`, no action. Value `na.exclude` can be useful.
`start`	starting values for the parameters in the linear predictor.

Figure 1: Help summary for glm command

```
##
## Call:
## glm(formula = Y ~ X, family = binomial, data = mydata)
##
## Deviance Residuals:
##      Min        1Q    Median        3Q       Max
## -2.40250  -0.03991   0.23094   0.46089   1.63530
##
## Coefficients:
```

```
##             Estimate Std. Error z value Pr(>|z|)
## (Intercept)   1.5249     0.3650   4.178 2.94e-05 ***
## X             0.7277     0.1484   4.904 9.37e-07 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 112.467  on 99  degrees of freedom
## Residual deviance:  67.643  on 98  degrees of freedom
## AIC: 71.643
##
## Number of Fisher Scoring iterations: 6
```

Notice that the best-fit coefficients almost exactly match those we obtained by directly optimising the likelihood ourselves. This is a very good sanity check! The **glm** command also does a lot of extra work for us, providing estimates of the standard errors on the estimated coefficients so that we can calculate confidence intervals.

Confidence intervals in logistic regression

The model summary above shows that **glm** returns much of the same information to us for logistic regression as it does for linear regression. This includes both estimates for the model parameters (the coefficients) and also standard errors for these estimates. Recall from weeks 3-4 that we can use these standard errors to obtain confidence intervals for the parameters - intervals within which we can be confident (to a given percentage value) that the true parameter lies.

The process for obtaining confidence intervals in logistic regression is very similar to that for linear regression with one key difference. In linear regression the standard errors were calculated *exactly* (remember that we derived the analytical expressions for parameter estimation). Therefore we were able to be very precise in our calculation of the confidence interval, taking note of how many data points we had, and therefore how many degrees of freedom, and finding the required critical value from the t-distribution.

In logistic regression by contrast, the standard errors are approximated by assuming that the likelihood is normally distributed. This means that the confidence intervals themselves will be approximate, and we perform a lower precision calculation. Instead of defining the interval based on the number of degrees of freedom and the t-distribution, we assume that the data is *large* and use the appropriate value from the normal distribution (sometimes called the z-distribution) instead (this is the same as the t-distribution for $n \to \infty$). For example, we can construct a 95% CI as:

$$95\%\ \text{CI}: \ \text{Estimate} \pm \text{Standard Error} \times z_{2.5\%}$$

Where $z_{2.5\%} = 1.96$ (look this up on the normal distribution table)

To illustrate, in the example above we can obtain the 95% confidence interval for the X coefficient like so:

```
#Get critical value of z_2.5% (should be 1.96)
zc = qnorm(0.975)

#Extract estimate and standard error of coefficient from model summary (check above)
estimate = summary(myglm)$coefficients[2,1]
standard_error = summary(myglm)$coefficients[2,2]

#Calculate and print the lower and upper boundaries of the confidence interval
```

```
CI_min = estimate - zc*standard_error
CI_max = estimate + zc*standard_error
print(CI_min)
```

```
## [1] 0.4368703
```

```
print(CI_max)
```

```
## [1] 1.018472
```

If you look back up to when I first generated the data for this model you will see that I used a value of $\beta_1 = 1$ for the X coefficient. Therefore we can see that the confidence interval comfortably contains the real value used. Try generating data using different values and repeating the whole process. Do you always find the real parameter inside the confidence interval.

Remember, for linear regression we calculate the confidence interval *exactly* using the t-distribution (and we need to know the exact data size). For logistic regression we calculate the confidence interval *approximately* using the normal distribution, and we assume that the data set is *large* (i.e. we use the limit as $n \to \infty$).

Model selection in logistic regression

All the techniques we looked at last week for model selection apply equally to logistic regression as to linear regression. As a reminder:

1. Plot the data and use your judgement to decide whether a specific logistic regression model is appropriate. Is this a classification problem? Is the trend of the data monotonic (does the output always get more likely or less likely to be 1 as an input is increased?). Which inputs are likely to have an effect on the output?
2. Fit the models of interest and record the AIC values. Lower values of AIC indicate that the model will be more useful for prediction
3. Perform cross-validation to directly test how well each model predicts new data. Remember to repeat the data splitting many times to check your results. Compare these to the values you get from AIC.

Other Generalised Linear Models

By now you should be getting quite familiar with the **glm** help page! If you've investigated the *family* argument for this command, you may have noticed that there are several other options beyond 'Gaussian' or 'binomial'. This is because Generalised Linear Models cover a wide range of regression and classification models, beyond linear regression and logistic regression.

A GLM is any model where the probability of the OUTPUT is a function of some linear combination of the INPUTS. That is:

$$P(Y = y \mid X_1 = x_1, \ldots, X_n = x_n) = f(\beta_0 + \beta_1 x_1 + \ldots + \beta_n x_n) \tag{6}$$

For example, in linear regression, $f()$ was the normal distribution density function, and in logistic regression $f()$ was the logistic function $\phi()$.