MATH5743M: Statistical Learning

Dr Seppo Virtanen, School of Mathematics, University of Leeds

Semester 2: 2022

Week 2: Optimisation

At the core of all statistical modelling is the concept of *optimisation*. Statistical learning is concerned with finding a model that is the best description of the available data – the clue is in the term *best*. Many models or descriptions of a given data set are possible. Our job is to find the best one both by using our training as statisticians to choose an appropriate type of model, and by employing methods of optimisation to make that model as good as possible.

Optimisation as a task can usually be phrased as attempting to maximise or minimise a specific function. We may wish to maximise the profit made by selling a product, which will be a function of the number of units sold and the price. Alternatively we may seek to maximise the number of patients we can treat at a hospital while minimising the cost of treatment. Consider as an example the problem of trying to optimise the firing range of a cannon, by manipulating the angle at which it fires. Clearly firing straight into the ground will not get the cannon ball very far, nor will shooting vertically into the air; there is some optimal angle in between these extremes that will maximise the distance the ball travels. We can determine this angle by various means, either *numerically* or *analytically*

Optimising numerically

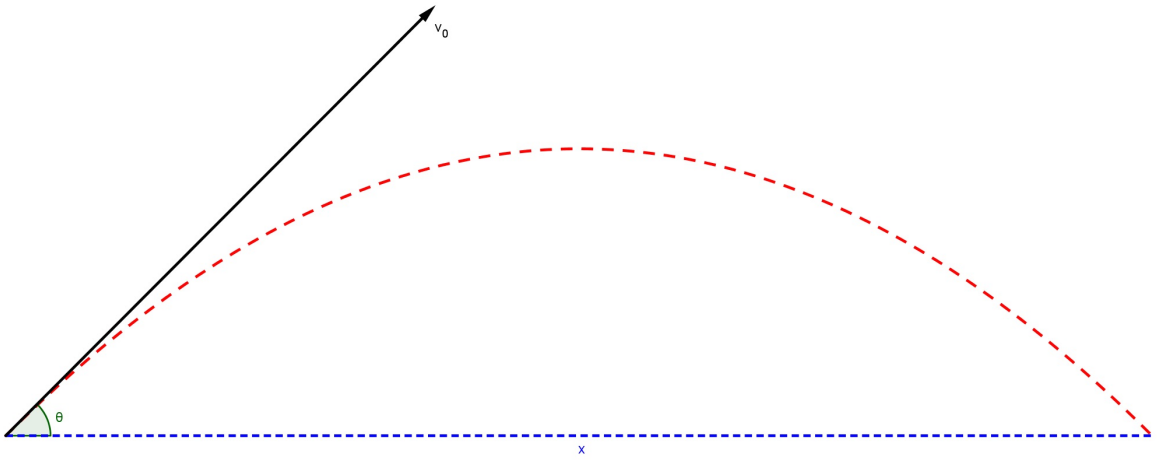


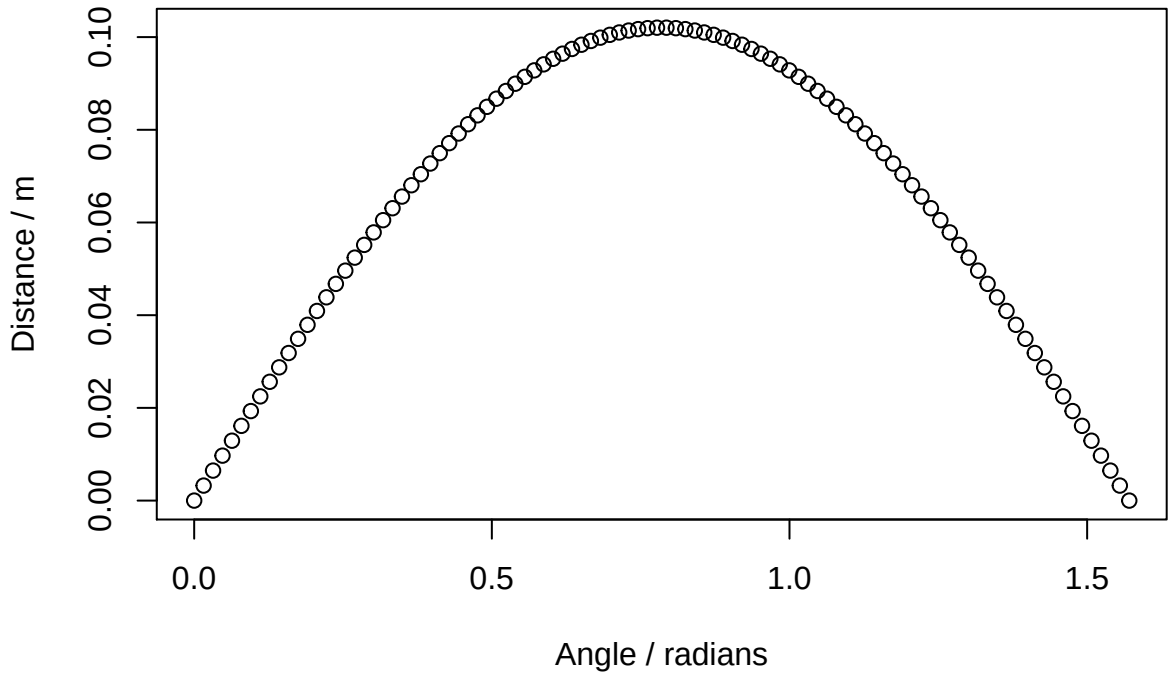
Figure 1: The path of a projectile fired from the ground depends on the initial velocity and angle.

A projectile such as a cannon ball, fired from the ground, follows a parabolic curve and travels a distance that depends on the initial velocity and angle (see Figure 1). The equation for the distance x travelled by a cannon ball fired at angle θ and at speed v (assuming no air resistance) is:

$$x = \frac{v^2}{g} sin(2\theta). \tag{1}$$

The simplest way to estimate the best value for θ is simply to start trying some values. Once this would have required hours of tedious calculation, but today we can test hundreds of values in an instance with almost no effort. Lets try this in R. We'll set $g = 9.8ms^{-2}$ and use an initial velocity of $v = 1ms^{-1}$. We'll test angles from 0 radians (straight along the ground) to $\pi/2$ radians (straight into the air):

```
theta = seq(0, pi/2, length.out=100)
x = (1/9.8)*sin(2*theta)
plot(theta, x, xlab="Angle / radians", ylab="Distance / m")
```



This plot shows us quite clearly that projectile flies furthest when the angle is close to $pi/4$ radians (about 0.79). We can use R to interrogate the values of x to find the best angle among the 100 that we tried

```
idx = which.max(x)
print(theta[idx])
```

```
## [1] 0.7774648
```

This asks for the index of the largest value in x, and then prints the corresponding value of θ, which we see is close to $\pi/4$. We could get a better estimate simply by increasing the number of different values of θ that we tried.

The above example illustrates what optimising a function means, but in many cases we cannot get good results by simply trying many different values and picking the best one. This is the case when the function has more than just one or two inputs, or when we need a very high degree of accuracy. Thankfully there

are much better numerical algorithms for optimisation that give us higher accuracy with less computational effort.

Hill-climbing algorithms

There are many different optimisation algorithms, but most rely fundamentally on the concept of *gradient ascent*. These types of algorithm use an initial guess for the best value, and then 'climb up' the function until they reach a peak, just like walker climbing to the top of a mountain, as shown in Figure 2

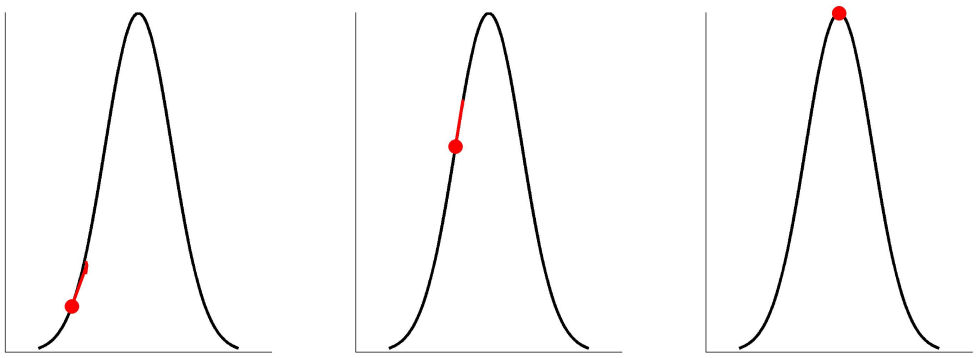


Figure 2: A gradient-ascent or 'hill climbing' algorithm improves an initial guess by following the gradient of the function, until it reaches a peak.

R has a general purpose optimisation function **optim**, which incorporates many of the best known and widely used optimisation algorithms. To use it you need to provide what is called an *objective function* – the function to minimise (for historical reasons optimisers are usually set to minimise rather than maximise) – and an initial guess at the solution. You can also help make the optimisation more efficient if you can provide a function that gives the gradient of the objective function. Lets see how it works for our cannon example.

```
#define an objective function to MINIMISE (notice the minus sign)
obj_fn <- function(param){-(1/9.8)*sin(2*param)}
#Start randomly between 0 and pi/2
start_guess = (pi/2)*runif(1)
#Run the optimiser and store the result
opt_result = optim(par=start_guess, fn=obj_fn)
#Print the optimised parameter value
print(paste("Best parameter value is: ", opt_result$par, collapse=""))
```

```
## [1] "Best parameter value is:  0.785423606907987"
```

Although the gradient-ascent algorithms utilised by the R **optim** function are powerful optimisation tools, they can give suboptimal results for complex optimisation problems due to a common problem, namely the existence of local maxima of the function being optimised. This is illustrated in Figure 3

Thankfully the two main statistical models that we will learn about in this course do not suffer from local maxima. This guarantees that when we use gradient-ascent to find the maximum of the likelihood function we can be sure we have found the global maximum, rather than a sub-optimal local maximum. Near the end of the course, when we take a look at more complex statistical models, we will need to remember to stay aware of the problems caused by local maxima.

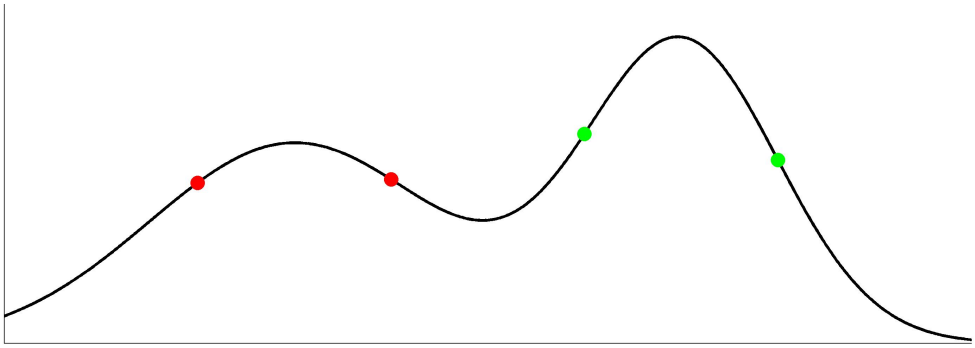


Figure 3: Local maxima can lead to optimsation answers that depend on the initial guess. Here the red initial guesses result in finding the left hand peak, while the green guesses result in finding the right hand peak.

Optimising analytically

Many of you will wonder why we went to the trouble of numerically estimating the best value of θ above, when the solution can be obtained exactly using basic calculus. The example is designed to illustrate the principles of optimisation, which can be applied to more complex problems where no analytic solution is possible. However, where an analytic solution can be found this gives us greater speed and precision.

To optimise a function analytically we use the property that the *gradient* of the function at its maximum is zero. This can be seen by considering that the gradient below the optimal value must be positive (i.e. the function is increasing), and the gradient above the optimal value must be negative (i.e. the function is decreasing). To go from a positive to a negative gradient we must pass through a point with zero gradient, at the optimal value.

In our example, this means that we can find the optimal value of θ by differentiating equation (1) and setting this to be zero at the optimal $\hat{\theta}$:

$$\begin{aligned} \left.\frac{dx}{d\theta}\right|_{\hat{\theta}} &= 0 = \frac{2v^2}{g}\cos(2\hat{\theta}) \\ &\Rightarrow \cos(2\hat{\theta}) = 0 \\ &\Rightarrow 2\hat{\theta} = \cos^{-1}(0) = \pi/2 \\ &\Rightarrow \hat{\theta} = \pi/4 = 0.7853982\ldots \end{aligned} \tag{2}$$

Maximum-likelihood estimation

Optimisation is important because it underlies all the methods we use to fit statistical models to data, and thus how we use statistical models to learn from data.

The technique we will use most often for fitting models is called *maximum-likelihood estimation* (MLE). Recall from the previous lecture that in supervised learning we try to determine the conditional probability of the outputs, based on the inputs:

$$P(\textsc{output} = y \mid \textsc{input} = x, \theta) = f(y, x, \theta). \tag{3}$$

The function $f()$ specifies the probability that the output value(s) will be y, given input value(s) of x, and the model parameters θ.

In the *learning* phase of statistical modelling, we first select an appropriate function $f()$, then we try to identify the best possible set of parameters θ so that we can make accurate predictions of the OUTPUT from the INPUT.

Maximum-likelihood estimation is a technique for choosing the best value of θ. It works by considering which value of θ would have made the data we have already seen most predictable. In other words, we look for the value of θ that maximises a quantity known as the likelihood $\mathcal{L}(\theta)$, which is the probability of all observed outputs, conditioned on all observed inputs and the value of θ

$$\mathcal{L}(\theta) = P(\text{OUTPUT} = \{y_1, \ldots, y_n\} \mid \text{INPUT} = \{x_1, \ldots, x_n\}, \theta) \tag{4}$$

Example

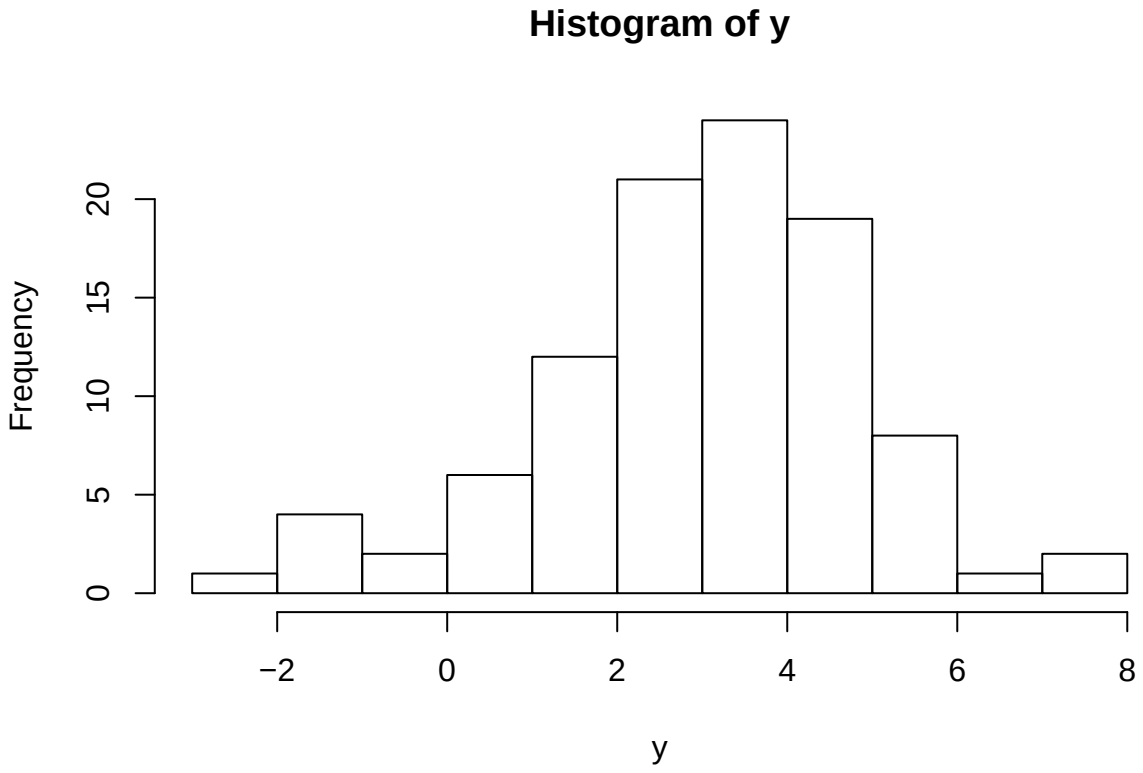
The normal or Gaussian distribution has a probability density function defined as:

$$p(X = x \mid \mu, \sigma) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(\frac{-(x-\mu)^2}{2\sigma^2}\right) \tag{5}$$

Lets generate 100 numbers from a normal distribution with a mean of 3 and a variance of 4 (st. deviation of 2), and see how well we can use these data to estimate the original parameters used, numerically or analytically.

First generate the data:

```
y = rnorm(100, m = 3, sd = 2)
hist(y)
```



The likelihood function for the model parameters μ and σ^2 is the probability of observing (or generating) these data from a normal distribution with those parameters. Since the samples are generated independently, this is the product of the probability for each data point

$$\mathcal{L}(\mu, \sigma) = \prod_{i=1}^{100} \frac{1}{\sqrt{2\pi}\sigma} \exp\left(\frac{-(y_i - \mu)^2}{2\sigma^2}\right) \tag{6}$$

When multiplying lots of probabilities together we can easily end up with very small numbers which can cause numerical errors on a computer. To avoid this we can use the *log-likelihood* instead; the maximum value of $f(x)$ and $\log(f(x))$ occur at the same value of x. This turns the product over probabilities into a sum over log-probabilities:

$$\log \mathcal{L}(\mu, \sigma^2) = -100 \log(\sqrt{2\pi}\sigma) - \sum_{i=1}^{100} \frac{(y_i - \mu)^2}{2\sigma^2} \tag{7}$$

In R we first define the negative log-likelihood as a function of parameter values, then we use the optim function to find the parameters that minimise this function.

```
lognorm <- function(param, Y){-100*log(sqrt(2*pi)*param[2]) -sum((Y-param[1])^2/(2*param[2]^2))}
neg_LL <- function(param) -lognorm(param, Y=y)
start_guess = c(0, 1) #first guess of mean of 0, stand.dev. of 1
mle_param = optim(par=start_guess, fn=neg_LL)
print(paste("MLE mean = ", mle_param$par[1], collapse=""))
```

```
## [1] "MLE mean =  3.00446055343074"
```

```
print(paste("MLE variance = ", mle_param$par[2]^2, collapse=""))
```

```
## [1] "MLE variance =  3.80165329838569"
```

Notice that this code prints its results in the form of easily human-read text – this is a good habit to get into when presenting your own work. Don't leave the reader searching for your key results in a jumble of numbers!

The MLE estimates for the mean and variance we obtain are close to the real values we used to generate the data. This is because this method of estimation specifically tries to replicate the generating process.

The analytical solution. In this case we didn't really need to use numerical approximation to obtain the MLE estimates for the mean and standard deviation. They can be inferred analytically using the calculus method described earlier. From before we have the following log-likelihood function:

$$\log \mathcal{L}(\mu, \sigma) = -100 \log(\sqrt{2\pi}\sigma) - \sum_{i=1}^{100} \frac{(y_i - \mu)^2}{2\sigma^2} \tag{8}$$

Since there are two parameters to estimate, we need the gradient to be zero with respect to both parameters at the MLE values, i.e.

$$\frac{\partial}{\partial \mu} \log \mathcal{L}\Big|_{\hat{\mu}, \hat{\sigma}} = 0, \tag{9}$$

and

$$\frac{\partial}{\partial \sigma} \log \mathcal{L} \Big|_{\hat{\mu},\hat{\sigma}} = 0. \tag{10}$$

Taking equation 9 first:

$$\begin{aligned} 0 &= \frac{\partial}{\partial \mu} \log \mathcal{L} \Big|_{\hat{\mu},\hat{\sigma}} \\ &= \sum_{i=1}^{100} \frac{(y_i - \hat{\mu})}{\sigma^2} \\ \Rightarrow \hat{\mu} &= \frac{1}{100} \sum_{i=1}^{100} y_i = \bar{y} \end{aligned} \tag{11}$$

The MLE estimate for the mean is just the sample mean, $\bar{y}$. Now taking the condition in equation 10:

$$\begin{aligned} 0 &= \frac{\partial}{\partial \sigma} \log \mathcal{L} \Big|_{\hat{\mu},\hat{\sigma}} \\ &= -\frac{100}{\hat{\sigma}} + \sum_{i=1}^{100} \frac{(y_i - \hat{\mu})^2}{\sigma^{*3}} \\ \Rightarrow \hat{\sigma}^2 &= \frac{1}{100} \sum_{i=1}^{100} (y_i - \hat{\mu})^2. \end{aligned} \tag{12}$$

The MLE estimate for the variance is simply the average square distance from the mean. These results therefore reflect our intuition for how to estimate the properties of the generating distribution from sample data (but see below for a note on the variance estimator). For the numbers we randomly generated before, these estimators give values of:

```
print(paste("Analytic MLE for mean = ", mean(y), collapse=""))
```

```
## [1] "Analytic MLE for mean =  3.00499541294645"
```

```
print(paste("Analytic MLE for variance = ", mean((y-mean(y))^2), collapse=""))
```

```
## [1] "Analytic MLE for variance =  3.8012932139099"
```

As you can see, these are very close to the results we obtained by numerical approximation. Try repeating this exercise yourself. Everytime you run the code, a new set of 100 numbers will be generated, so the estimates you get will change. Check that the numerical and analytic solutions remain close to each other.

- What happens if you increase or decrease the number of samples generated? Try 3, 10, 50, 1000, 10000.

A note on the MLE estimation of the variance

You may have noticed that the MLE estimate for the variance is different from the estimator you will have learned to use in other modules. As a reminder, the MLE estimate is:

$$\hat{\sigma}^2_{\text{MLE}} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{\mu})^2. \tag{13}$$

In previous modules you will have encountered the *unbiased* estimator:

$$\hat{\sigma}^2_{\text{unbiased}} = \frac{1}{n-1}\sum_{i=1}^{n}(y_i - \hat{\mu})^2. \tag{14}$$

Note the difference in the divisor. The unbiased estimator, as the name suggests, gives an unbiased estimate of the true variance. That means that the expected value taken over lots of different data sets is the same as the true variance. As you can probably tell then, the MLE estimator does not have this property. Therefore, when we want to estimate variances, we need to be careful to specify which estimator we are supposed to use. As a rule of thumb, it is usually better to use the unbiased estimate – that is, to use the $n-1$ denominator – unless you are specifically told otherwise

We will encounter this issue again next week in linear regression, so stay tuned and keep it in mind.