Machine Learning Lab Assessment

Tamlin Love - 1438243

1 Optimization: Direct Approach

1.1
One could differentiate the objective function and set it equal to 0. The parameters that give a function value of 0 denote a critical point of the function. Taking the second derivative and checking its sign (positive or negative) will ascertain whether the critical point is a minimum or maximum.

1.2
a)
The linear regression model is as follows:

$$\hat{y}_i = \theta_1 x_i + \theta_2 \tag{1}$$

The error function we wish to minimise is as follows:

$$E(\hat{y}_i) = \sum_{i=1}^{n} (\hat{y}_i - y_i)^2 \tag{2}$$

Substituting Equation (1):

$$E(\theta_1, \theta_2) = \sum_{i=1}^{n} (\theta_1 x_i + \theta_2 - y_i)^2 \tag{3}$$

which expands to

$$E(\theta_1, \theta_2) = \theta_1^2 \sum_{i=1}^{n} x_i^2 + 2\theta_1\theta_2 \sum_{i=1}^{n} x_i - 2\theta_1 \sum_{i=1}^{n} x_i y_i + n\theta_2^2 - 2\theta_2 \sum_{i=1}^{n} y_i + \sum_{i=1}^{n} y_i^2 \tag{4}$$

To make things simpler, we introduce the following constants:

$$\alpha = \sum_{i=1}^{n} x_i, \beta = \sum_{i=1}^{n} x_i^2, \gamma = \sum_{i=1}^{n} y_i, \delta = \sum_{i=1}^{n} y_i^2, \epsilon = \sum_{i=1}^{n} x_i y_i \tag{5}$$

Thus, Equation (4) becomes:

$$E(\theta_1, \theta_2) = \beta\theta_1^2 + 2\theta_1\theta_2\alpha - 2\theta_1\epsilon + n\theta_2^2 - 2\theta_2\gamma + \delta \tag{6}$$

To minimise the error, we set the partial derivates of E with respect to θ_1 and θ_2 to equal 0. Thus Equation (6) becomes:

$$\frac{\partial E}{\partial \theta_1} = 2\beta\theta_1 + 2\alpha\theta_2 - 2\epsilon = 0 \tag{7}$$

$$\frac{\partial E}{\partial \theta_2} = 2\alpha\theta_1 + 2n\theta_2 - 2\gamma = 0 \tag{8}$$

Rearranging the above equations, we obtain:

$$\theta_2 = \frac{\gamma - \theta_1\alpha}{n} \tag{9}$$

$$\theta_1 = \frac{n\epsilon - \alpha\gamma}{n\beta - \ \alpha^2} \tag{10}$$

We can leave Equation (9) as it is, as it is simple to implement θ_2 in terms of θ_1. For completion's sake, however, the uncoupled equation for θ_2 is given below:

$$\theta_2 = \frac{\gamma(n\beta - \alpha^2) - \alpha(n\epsilon - \alpha\gamma)}{n(n\beta - \alpha^2)} \tag{11}$$

b)
In the exponential model, the expression for $\hat{y}_i$ is as follows:

$$\hat{y}_i = \theta_1 e^{\theta_2 x_i} \tag{12}$$

Which is equivalent to:

$$\ln \hat{y}_i = \ln \theta_1 + \theta_2 x_i \tag{13}$$

We introduce the following substitutions:

$$Y = \ln \hat{y}_i, a = \ln \theta_1, b = \theta_2 \tag{14}$$

Thus, Equation (13) becomes:

$$Y = a + bx_i \tag{15}$$

which is simply a linear regression model. Thus, using the constants introduced in (5), we obtain:

$$\ln \theta_1 = a = \frac{\gamma - b\alpha}{n} = \frac{\gamma(n\beta - \alpha^2) - \alpha(n\epsilon - \alpha\gamma)}{n(n\beta - \alpha^2)} \tag{16}$$

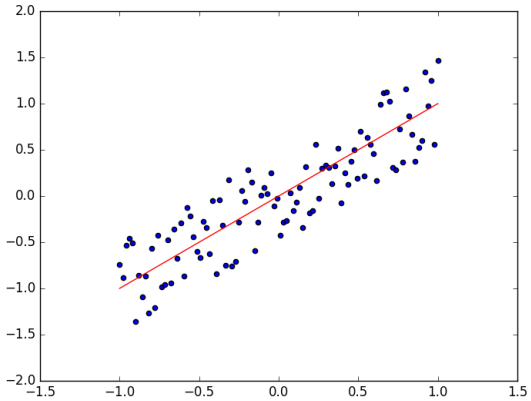
$$\theta_2 = b = \frac{n\epsilon - \alpha\gamma}{n\beta - \ \alpha^2} \tag{17}$$

Thus, from Equation (16), we obtain:

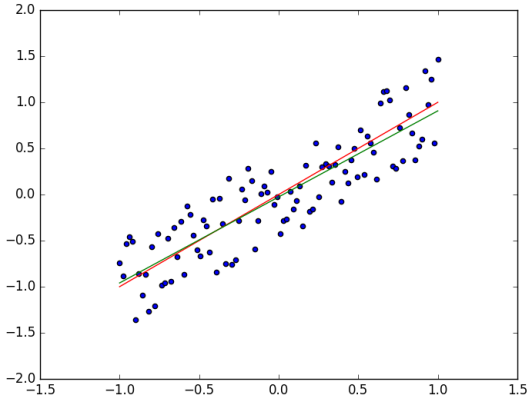
$$\theta_1 = e^{\frac{\gamma - \theta_2\alpha}{n}} = e^{\frac{\gamma(n\beta - \alpha^2) - \alpha(n\epsilon - \alpha\gamma)}{n(n\beta - \alpha^2)}} \tag{18}$$

1.3

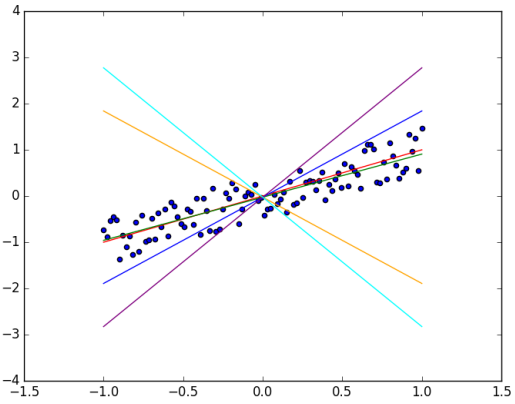
The following linear data set was produced using the line $y = x$ (shown in red), with added noise of uniform distribution.



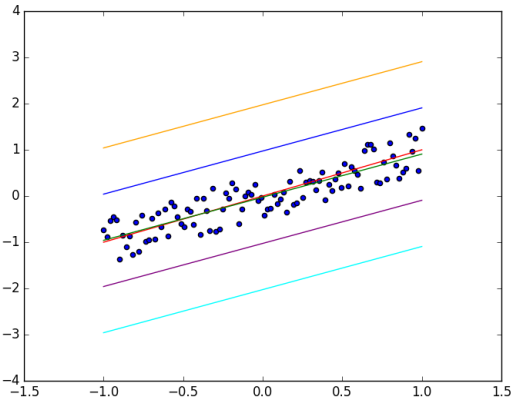
Using Equation (1) with the values of θ_1 and θ_2 defined in Equations (9) and (10), we obtain the following regression line (in green), which is reasonably close to the original line (in red).



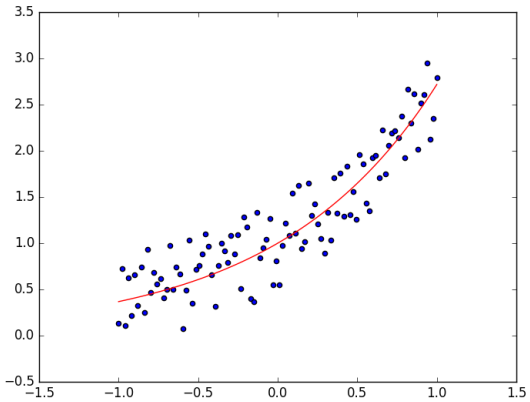
If we vary the value of θ_1, the slope of the regression line changes, as illustrated below.



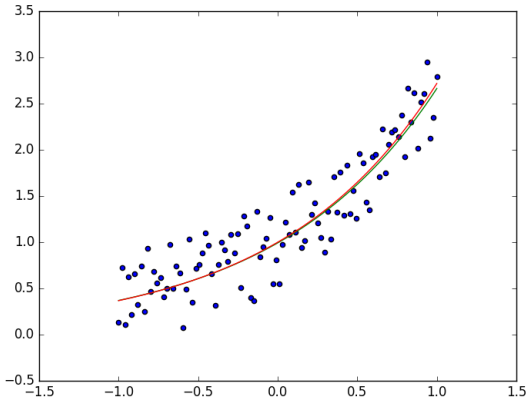
If we vary the value of θ_2, the regression line moves vertically in the x-y plane, as illustrated below.



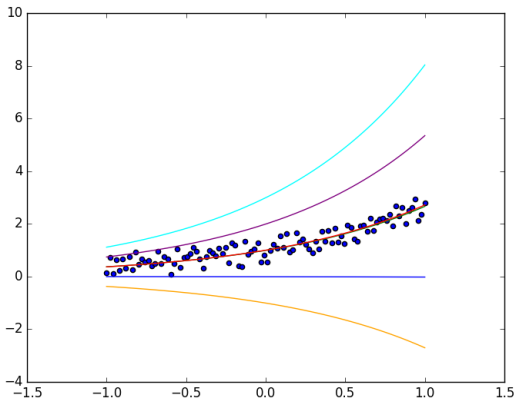
Now, we generate noisy data using the simple exponential function $y = e^x$ (shown in red).



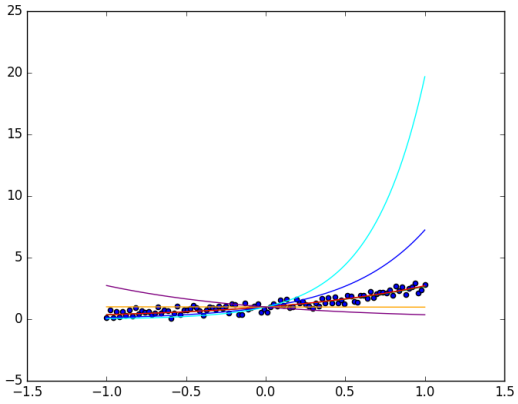
We must now change the functional form of the regression model from linear to exponential, as in Equation (12). Using the values of θ_1 and θ_2 from Equations (17) and (18), we obtain the following regression curve (shown in green)



Varying θ_1, we obtain the following series of curves.



Similarly, varying θ_2, we obtain the following:



2 Optimization: Iterative Approach

2.1
When a problem is impossible (or merely impractical) to optimize analytically, we can still optimize it numerically using iterative methods.

a)
For continuous functions, gradients provide a useful optimisation approach as they show either the direction of descent (in the case of univariate functions) or the direction of steepest descent (in the case of multivariate functions).

b)
We could evaluate sampled points across the domain of the function and then approximate the minimiser as the sampled point with the lowest function value. We could perhaps assign each sampled point a range to indicate the range around the point in which the minimiser could lie.

2.2

Let f be a function in $\mathbf{R}$, let $x \in \mathbf{R}$ such that $f(x) = 0$, and let x_0 be an approximation of x. The expansion of $f(x)$ as a Taylor series is

$$f(x) = f(x_0) + (x - x_0)f'(x_0) + O((x - x_0)^2) \tag{19}$$

Thus, $f(x)$ approximates to

$$f(x) = 0 \approx f(x_0) + (x - x_0)f'(x_0) \tag{20}$$

which can be rearranged to

$$x \approx x_0 - \frac{f(x_0)}{f'(x_0)} = x_1 \tag{21}$$

x_1 is a better approximation of x than x_0. Repeating the process iteratively, we arrive at the Newton-Raphson method, which uses the formula

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)} \tag{22}$$

a)
The Newton-Raphson can be used to optimise a function by finding the root of its derivative (x such that $f'(x) = 0$), which is necessary in order to minimise the error of the model.

b)
In a machine learning context, x_i can be interpreted as input data, $f(x_i)$ as the class of x_i, and $f'(x_i)$ as the rate at which the class changes with respect to the input data. The exact interpretations depend on the problem at hand.

2.3

Let $\bar{f} : \mathbf{R}^n \to \mathbf{R}^n$ and let $\bar{x} \in \mathbf{R}^n$. The expansion of $\bar{f}(\bar{x})$ as a Taylor series is

$$\bar{f}_i(\bar{x} + h\bar{x}) = \bar{f}_i(\bar{x}) + \sum_{j=1}^{n} \frac{\partial \bar{f}_i(\bar{x})}{\partial x_j} h x_j + O(h\bar{x}^2) \approx \bar{f}_i(\bar{x}) + \sum_{j=1}^{n} \frac{\partial \bar{f}_i(\bar{x})}{\partial x_j} h x_j \quad (23)$$

for each $i = 1, ..., n$. The n equations can be combined into a single equation

$$\bar{f}_i(\bar{x} + h\bar{x}) \approx \begin{bmatrix} f_1(\bar{x}) \\ \vdots \\ f_n(\bar{x}) \end{bmatrix} + \begin{bmatrix} \frac{\partial f_1}{\partial x_1} \cdots \frac{\partial f_1}{\partial x_n} \\ \vdots \ddots \vdots \\ \frac{\partial f_n}{\partial x_1} \cdots \frac{\partial f_n}{\partial x_n} \end{bmatrix} \begin{bmatrix} h x_1 \\ \vdots \\ h x_n \end{bmatrix} \quad (24)$$

which is simply

$$\bar{f}_i(\bar{x} + h\bar{x}) \approx \bar{f}(\bar{x}) + J(\bar{x}) h\bar{x} \quad (25)$$

where J denotes the Jacobian of $\bar{f}$. Setting Equation (25) equal to 0, we can solve for $h\bar{x}$, obtaining

$$h\bar{x} = J(\bar{x})^{-1}[\bar{f}(\bar{x} + h\bar{x}) - \bar{f}(\bar{x})] = -J(\bar{x})^{-1}\bar{f}(\bar{x}) \quad (26)$$

Thus, we can approximate the root as

$$\bar{x} + h\bar{x} = \bar{x} - J(\bar{x})^{-1}\bar{f}(\bar{x}) \quad (27)$$

As this is only an approximation of the root, we can improve our approximation by repeating the process iteratively, providing us with the formula for the n-dimensional Newton-Raphson method,

$$\bar{x}_{k+1} = \bar{x}_k - J(\bar{x}_k)^{-1}\bar{f}(\bar{x}_k) \quad (28)$$

2.4

The following python script implements the multivariate Newton-Raphson method described in 2.3.

```
import numpy as np

def f(x):
    return np.array([[x.item(1) - x.item(0)**2],
                     [x.item(0)**2 + x.item(1)**2 - 1]
                     ], float)

def jacobian(x):
    return np.array([
                      [-2*x.item(0), 1],
                      [2*x.item(0), 2*x.item(1)]
                      ], float)

def newtonRaphson(x0,f,J,tol):
    x1 = x0 - np.matmul(np.linalg.inv(J(x0)), f(x0))
    while(np.all(x1-x0)>tol):
        x0 = x1
        x1 = x0 - np.matmul(np.linalg.inv(J(x0)), f(x0))
    return x1

x0 = np.array([[1],
               [1]
               ])
tol = 0.005
xroot = newtonRaphson(x0,f,jacobian,tol)
print(xroot)
```

The function *newtonRaphson* actually implements the method, while the functions f and *jacobian* define an example function and its Jacobian. In this case, these were

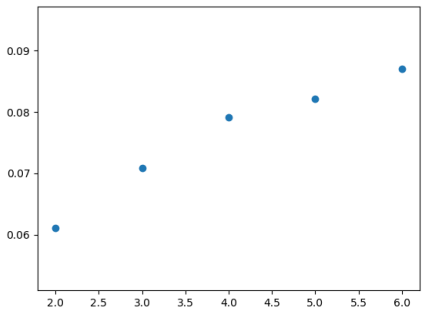
$$\bar{f}(\bar{x}) = \begin{bmatrix} x_2 - x_1^2 \\ x_1^2 + x_2^2 - 1 \end{bmatrix}, J(\bar{x}) = \begin{bmatrix} -2x_1 & 1 \\ 2x_1 & 2x_2 \end{bmatrix} \tag{29}$$

The inputs seen in in the code ($x_0 = [1, 1]$ and $tol = 0.005$) give a final answer of $[0.78615138, 0.61803399]$, which is correct within the tolerance.

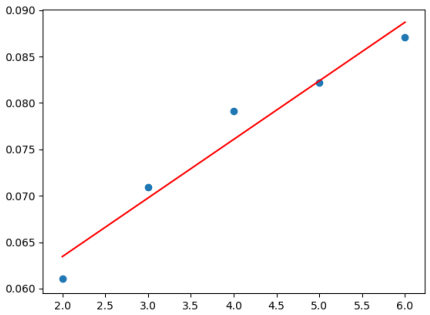
2.5

One aspect for which we can test is how much time the algorithm takes given a problem of dimension n.

To time the algorithm, a function and x_0 of dimension n and a Jacobian of dimension $n \times n$ were supplied. The method was timed for 100 runs, and then this was repeated 100 times to produce the mean time it takes for the method to run 100 times for dimension n. The results are shown below for $n = 2, 3, 4, 5, 6$.



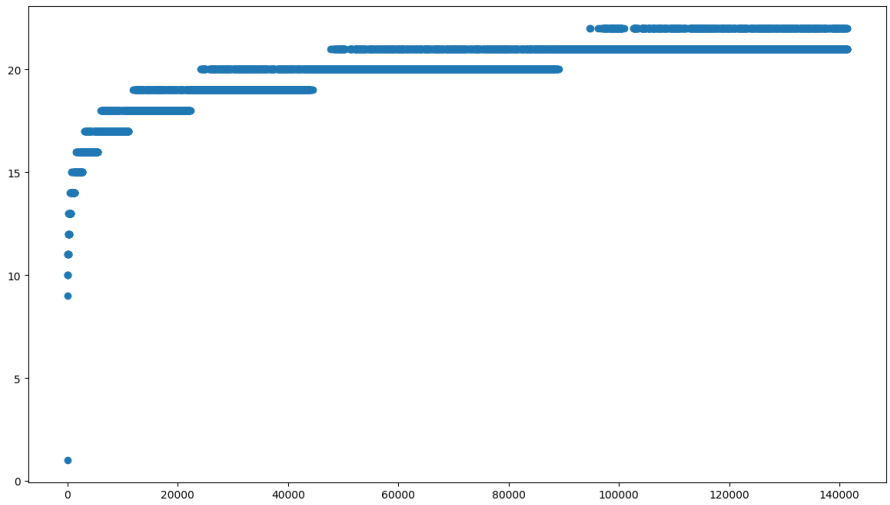
Applying a linear regression on the data, we arrive at the following graph



which displays the line $time = 0.0063128951n + 0.0508223819$ seconds. Since the regression line fits the data reasonably well, we can assume that the time the algorithm takes to run will increase linearly as n increases, which means we can expect the algorithm to run fairly well for large n.

Another aspect for which we can test is the number of iterations it takes to find the root from an initial point a given distance away from the root.

The method was run using initial points of increasingly greater distance from the actual root. In this case, this was repeated 10,000 times, with the initial point increasing by 10 each time. The graph below plots this distance against the number of iterations it took for the method to find the root.



The number of iterations plateaus as the distance increases, indicating that the method will find the root reasonably quickly from points a great distance away from it, provided there aren't other roots between the initial point and the desired root.

Overall, it seems the Newton-Raphson method is robust enough to cope with problems of a large dimension and with initial approximations of incredibly poor accuracy (provided that there isn't much happening between the desired root and the initial point).

2.6

a)

One could improve the Newton-Raphson method by changing it from the form given in Equation (28) to

$$\bar{x}_{k+1} = \bar{x}_k - \rho_k J(\bar{x}_k)^{-1} \bar{f}(\bar{x}_k) \tag{30}$$

This does introduce the problem of choosing a suitable ρ_k for each iteration that improves convergence rather than hindering it.

One downside of the Newton-Raphson method is that computing the inverse Jacobian each iteration can be prohibitively expensive and slow. We might be able to improve convergence by only updating the Jacobian every σ iterations. This does introduce the problem of how to set the value of σ.

b)

If the function has many roots and a poor initial point is chosen, the method may converge to an undesired root (for example, a root which falls outside the desired parameter range) or, worse, may not converge at all (for example, an x_k is chosen such that $x_{k+1} = x_k$). To overcome this, it is usually sufficient to choose an initial point close to the root.

c)

If the function is "very wrinkly", the method would likely converge to an undesired root (for example, a very "shallow" optimum). The method can converge to undesired roots that are close to or far away from the initial point, and thus this problem can only be overcome with very careful selection of the initial point.