# Assignment 1

Lukas Jordbru

September 2024

## 1 Problem 1

### 1.1 a)

The formula fo linear regression is $\hat{y}_i = w_0 x_i + w_1$ and one of the most common loss function is the mean squared error given by equation 1

$$\mathcal{L}_{MSE}(\hat{y}, y) = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2 \tag{1}$$

To get the gradient descend for each of two weights, we need to take the partial derivative of the loss function with respect to each weight.

$$\frac{\partial \mathcal{L}_{MSE}}{\partial w_0} = \frac{1}{N}\sum_{i=1}^{N} -2w_0(y_i - \hat{y}_i) \tag{2}$$

$$\frac{\partial \mathcal{L}_{MSE}}{\partial w_1} = \frac{1}{N}\sum_{i=1}^{N} -2(y_i - \hat{y}_i) \tag{3}$$

### 1.2 b)

When using gradient descent it is preferred to have a smooth loss function, meaning that the loss function has differentiable and continuous gradients. This way, when updating the weights for the loss function will result in a stable convergence, and avoid that it oscillates because of sudden changes in the gradient. Taking the mean absolute error as an example, the formula is given by

$$\mathcal{L}_{MAE} = \frac{1}{N}\sum_{i=1}^{N}|y_i - \hat{y}_i| \tag{4}$$

If we calculate the gradient of this based on a weight $W_1$ we get

$$\frac{\partial \mathcal{L}_{MAE}}{\partial w_1} = \frac{1}{N}\sum_{i=1}^{N}\frac{\partial |y_i - \hat{y}_i|}{\partial \hat{y}_i}\cdot\frac{\partial \hat{y}_i}{\partial w_1} = \begin{cases} 1, & \text{if } \hat{y}_i < y_i \\ -1, & \text{if } \hat{y}_i > y_i \end{cases} \tag{5}$$

This means that it will update the parameter $w_1$ with the same value regardless of how far off the predictions are.

### 1.3 c)

Here I will describe a step by step guide on how to use gradient descend to improve the weights.

1. The first thing to do is to initialise the weights and set the learning rate. A simple approach to set the weights is to set them equal to zero, or choose a randomly small number. The learning rate might need some tuning to get the most effective and correct result, but should be set low. A lower learning rate means a more slow convergence and more stable, however a more slowly and in need of more computer power. A higher learning rate will converge more quickly, but could overshoot the minimum.

2. Chose a loss function that is well suited to your data set and end goal. Then compute the gradient of the loss function to each weight.

3. Iterative go through the training set and update the weight. In order to calculate the weight the gradient of the loss function must be used. The formula is

$$w_i = w_i - \alpha \frac{\partial \mathcal{L}}{\partial w_j} \tag{6}$$

   Here $\alpha$ is the learning rate. This way we update the weight in the correct direction, with a step of the learning rate. This is to be done until stop

4. Terminate the iterative process when finished. The process can be seen as finished when the convergence of the loss function is small, smaller than the threshold, or when the process has iterated over a certain amount of times, or when the gradient is close to zero. This means that the weight updates almost nothing at a time, and we have reached a local minimum

## 1.4 d)

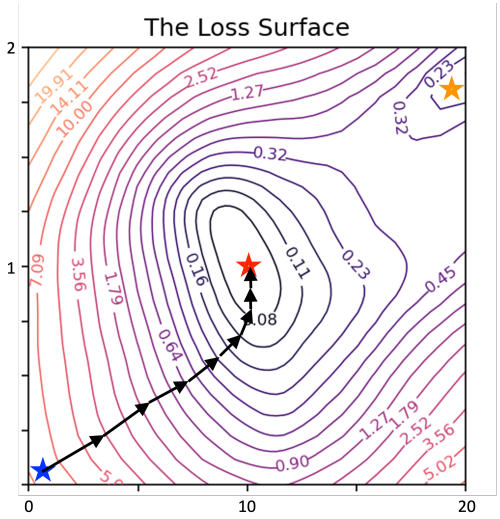


Figure 1: Gradient descend on the loss surface to the global minimum

## 1.5 e)

In the figure 1 you can see the path of the gradient descend drawn in by the black arrows as an iterative process. Here one arrow is one iterative process. In this picture you can see that when the lines becomes shorter and shorter, and that is because the values of each contour line becomes exponentially smaller and therefore the weight will be updated at a

lower smaller rate each time. The arrows are also pointing 90 degrees on to the contour lines at the arrows bottom. This is because it is going in the direction of where it is shortest to smaller value since the weights are updated based on the gradient

### 1.6 f)

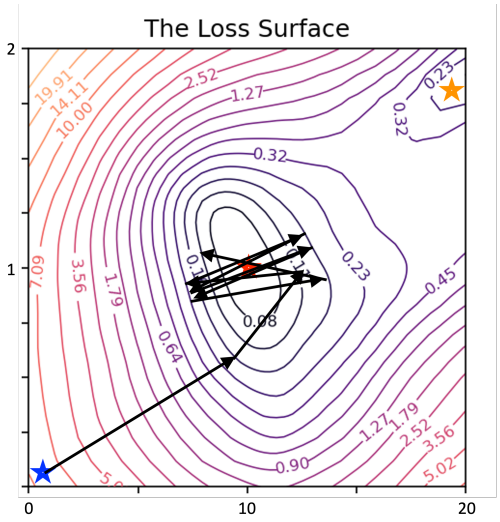


Figure 2: Gradient descent with a too large learning rate

### 1.7 g)

If we end up in a local minimum, it means we have reached a point where the gradient is, or is close to zero and therefore the gradient descend method will not unless specified go out of the local minimum and will terminate the process with a bad result. To fix this one can try to change the weight radically in the opposite way of the way we started to see if we can get out of the local minimum point and try to reach the globally minimum. This way we can start from a new point and go through the iterative process again, but get to a new minimum. When doing this it can be wise to store the new starting weight points. This is because if it fails to find the global minimum from there as well, one can to it again, but start from a new point away from both the previous starting points, and so on.

## 2 Problem 2

### 2.1 a)

The dataset has 3600 values, with features as one row, and the classification as the other row. Figure 3 shows the histogram of the data. From this we can see that there is some overlap between values 5 and 10 for the feature. This will likely cause some miss classification around this area. However, this is a small portion of the whole data, and therefore the accuracy should not bee too low, and should be able to separate the two classes.

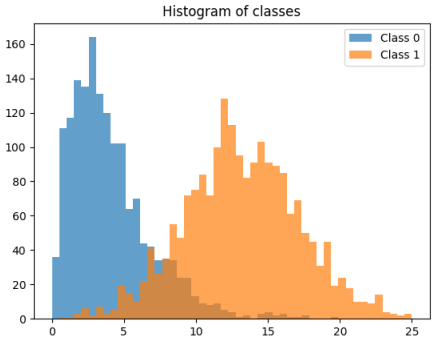


Figure 3: Histogram of classes

## 2.2 b)

The maximum likelihood estimate for $\beta$ can be calculated from the gamma distribution

$$p(x|C_0) = \frac{1}{\beta^\alpha \Gamma(\alpha)} x^{\alpha-1} e^{-\frac{x}{\beta}} \tag{7}$$

Here $\alpha = 2$, and since the gamma function for $n \in \{0, 1, 2, 3, ...\}$ can be calculated with $\Gamma(\alpha) = (\alpha - 1)! = 1$. The likelihood function $L(\beta)$ for data $x_1^0, x_2^0, ..., x_{n_0}^0$ is then

$$L(\beta) = \prod_{j=1}^{n_0} \frac{1}{\beta^2} x_j e^{-\frac{x_j}{\beta}} \tag{8}$$

The log-likelihood function then becomes

$$logL(\beta) = \sum_{j=1}^{n_0} \left( -2log(\beta) + log(x_j) - \frac{x_j}{\beta} \right) \tag{9}$$

We can then take the derivative and set it equal to zero to solve it for $\beta$ and find a estimate. This becomes

$$\frac{d}{d\beta} logL(\beta) = \sum_{j=1}^{n_0} \left( -\frac{2}{\beta} + \frac{x_j}{\beta^2} \right) = 0 \tag{10}$$

$$\hat{\beta} = \frac{1}{n_0 \alpha} \sum_{j=1}^{n_0} x_j \tag{11}$$

The maximum likelihood estimates for $\sigma$ and $\mu$ can be calculated from the Gaussian distribution

$$p(x|C_1) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2} \tag{12}$$

The likelihood function $L(\mu, \sigma^2)$ for data $x_1^1, x_2^1, ..., x_{n_1}^1$ is

$$L(\mu, \sigma^2) = \prod_{j=1}^{n_1} \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x_j-\mu)^2}{2\sigma^2}} \tag{13}$$

The log-likelihood function is then

$$logL(\mu, \sigma^2) = -n_1 log(\sigma\sqrt{2\pi}) - \frac{1}{2\sigma^2} \sum_{j=1}^{n_1} (x_j - \mu)^2 \tag{14}$$

The fro calculating the $\mu$ estimate we can take the partial derivative with respect to $\mu$ and setting it equal to zero

$$\frac{\partial}{\partial \mu} logL(\mu, \sigma^2) = \frac{1}{\sigma^2} \sum_{j=1}^{n_1} (x_j - \mu) = 0 \tag{15}$$

Solve for $\mu$

$$\hat{\mu} = \frac{1}{n_1} \sum_{j=1}^{n_1} x_j \tag{16}$$

To find the $\sigma$ estimate we can take the partial derivative of the log-likelihood function with respect to $\sigma$ and setting it to zero

$$\frac{\partial}{\partial \sigma} logL(\mu, \sigma^2) = -\frac{n_1}{2\sigma^2} + \frac{1}{2\sigma^4} \sum_{j=1}^{n_1} (x_j - \mu)^2 = 0 \tag{17}$$

Solving it for $\sigma$ and we get

$$\hat{\sigma}^2 = \frac{1}{n_1} \sum_{j=1}^{n_1} (x_j - \mu)^2 \tag{18}$$

Thus we have shown the estimates for the unknown parameters

## 2.3 c)

Starting with splitting the data into training and test data sets wit the 80/20 rule as you can see below

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

#loading the data
df = pd.read_csv("assignment2/data_problem2.csv", header=None)

features = df.loc[0].to_numpy()
classification = df.loc[1].to_numpy()

#Splitting the data
split = int(len(features)*0.8)

training_features = features[:split]
training_classification = classification[:split]

test_features = features[split:]
test_classification = classification[split:]
```

Then I can define the estimates as function, and use them with the training sets to get the estimates. I can then use a classifier function to classify the test set. This can be done by using the estimates in the gaussian and the gamma function, and which ever has the highest probability is what it gets classified as.

```
#making a classifier function
def classifier(C0, C1):
    classific = np.zeros(len(C0))
    classific[C0<C1] = 1
    return classific


#Findin the accuracy
prob_classification = classifier(prob_class_C0, prob_class_C1)

accuracy = np.mean(test_classification == prob_classification)*100
```

I then get the accuracy of 85%

## 2.4 d)

When the probability distribution is known, we can calculate the probability of each data point belonging to each class. Then decide on which class that data point is belonging to based on which has the higher probability. For example a data point x has a certain probability of belonging to class 0, and another probability for belonging to class 1. We can then calculate each probability $p(x|C_0)$ and $p(x|C_1)$. If $p(x|C_0) > p(x|C_1)$ then x will be classified as class 0. This way there will be less miss classification since we calculate the probability for each class, and then choosing the highest probability.

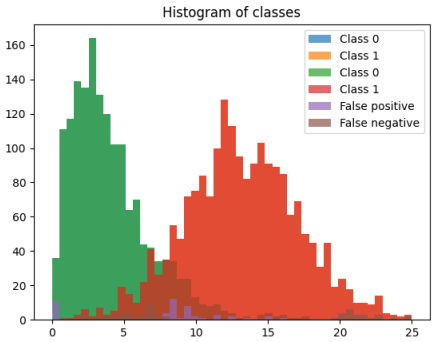


Figure 4: Histogram of data with the false classifications

In the figure 4 you can see the histogram with the false classifications added in. Here we can see that there are some false classification, but it is more spread out than what I believed to be according to problem a). I thought it would be more false classifications around feature value 5-10 since that is were there is the most overlap between the classes.