# Ass2 - FYS-2021

Lars Moen Storvik

September 2024

## Problem 1

### 1a

The most common type of loss function used for linear regression is Mean Squared Error (MSE):

$$L_{reg}(y, \hat{y}) = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

where our linear regression prediction for the i-th data point is:

$$\hat{y}_i = w_1 x_i + w_o$$

This Loss function takes the squared error between the actual data point and our predicted value for the data point. Minimizing this error is what helps our function to find the best fitting line.
To minimize the error, we use gradient descent to fit the parameters w_1 and w_o. This is done by computing the derivation of the gradient with respect to each weight parameter:
The individual square error for each data point is:

$$(y_i - \hat{y}_i) = (y_i - (w_1 x_i + w_o)$$

Derivation with respect to w_1 gives:

$$\frac{d}{dw_1}L_{reg} = \frac{1}{n}\sum_{i=1}^{n}\frac{d}{dw_1}((y_i - (w_1 x_i + w_o))^2$$

Using the chain rule on $\frac{d}{dw_1}((y_i - (w_1 x_i + w_o))^2$:

$$\frac{d}{dw_1}((y_i - (w_1 x_i + w_o))^2 = 2(y_i - (w_1 x_i + w_o))\frac{d}{dw_1}(y_i - (w_1 x_i + w_o))$$

Knowing $\frac{d}{dw_1}(y_i - (w_1 x_i + w_o)) = x_i$ we get:

$$\frac{d}{dw_1}L_{reg} = \underline{\frac{2}{n}\sum_{i=1}^{n}(y_i - (w_1 x_i + w_o)) * x_i}$$

Computing $\frac{d}{dw_o}L_{reg}$ is very similar, only difference is that $\frac{d}{dw_0}(y_i - (w_1 x_i + w_o)) = 1$:

$$\frac{d}{dw_0}L_{reg} = \underline{\frac{2}{n}\sum_{i=1}^{n}(y_i - (w_1 x_i + w_o))}$$

## 1b

Smooth loss functions are preferred in linear regression because they have a faster convergence rate with more consistent parameter updates. The reason for their fast convergence is their error magnification mechanism, like MSE is squaring the error.
When looking at MAE I will first calculate the gradient:

$$L_{MAE} = \frac{1}{n}\sum_{i=1}^{n} |y_i - \hat{y}_i|$$

$$\frac{d}{dw_1}L_{MAE} = \frac{1}{n}\sum_{i=1}^{n} \frac{d}{dw_1}|y_i - (w_1 x_i + w_0)|$$

$$\frac{d}{dw_1}|y_i - \hat{y}_i| = sign(y_i - \hat{y}_i)x_i$$

where

$$\text{sign(a)} = \begin{cases} 1, & \text{if } a > 0 \\ -1, & \text{if } a < 0 \\ 0, & \text{if } a = 0 \end{cases}$$

so

$$\frac{d}{dw_1}L_{MAE} = \frac{1}{n}\sum_{i=1}^{n} sign(y_i - \hat{y}_i)x_i.$$

We know that

$$\frac{d}{dw_0}|y_i - \hat{y}_i| = sign(y_i - \hat{y}_i)$$

so

$$\frac{d}{dw_0}L_{MAE} = \frac{1}{n}\sum_{i=1}^{n} sign(y_i - \hat{y}_i)$$

Looking at the gradients, we can see that the change in the gradient when doing gradient descent depends on the sign of the error, making the gradient descent less consistent and slower than a smooth loss function. Also, when the error $|y_i - \hat{y}_i|$ is zero the gradient is undefined, maybe leading the optimizer to be unpredictable, especially when the models gets closer to the true values. However, MAE is more robust to outliers.

## 1c

In the optimization of weights in gradient descent we start by **initializing the weights**. The start value isn't important because the weights will change during gradient descent, but small random values are normally used.
**Forward propagation**
Forward propagation is the first step in gradient descent. It takes one or more samples $x\_i$ in and makes a prediction $\hat{y}$. The prediction is used to calculate the loss $L(y, \hat{y})$ by comparing it with the expected value $y$.
**Backward propagation**
Back ward propagation is calculating the gradient of the loss function with respect to each weight $w_j$: $\frac{dL}{dw_j}$
**Adjust the weights**
When adjusting the weights we use a learning rate $\alpha$ to control the step size. The weights are adjusted by subtracting the gradient multiplied by the learning rate:

$$w_j = w_j - a\frac{dL}{dw_j}$$

The three steps above are repeated until convergence. The convergence can happen in several ways, two of those being that the algorithm runs a fixed number of steps or the change in loss function between iterations has reached a threshold (e.g. $10^8$).

**1d**

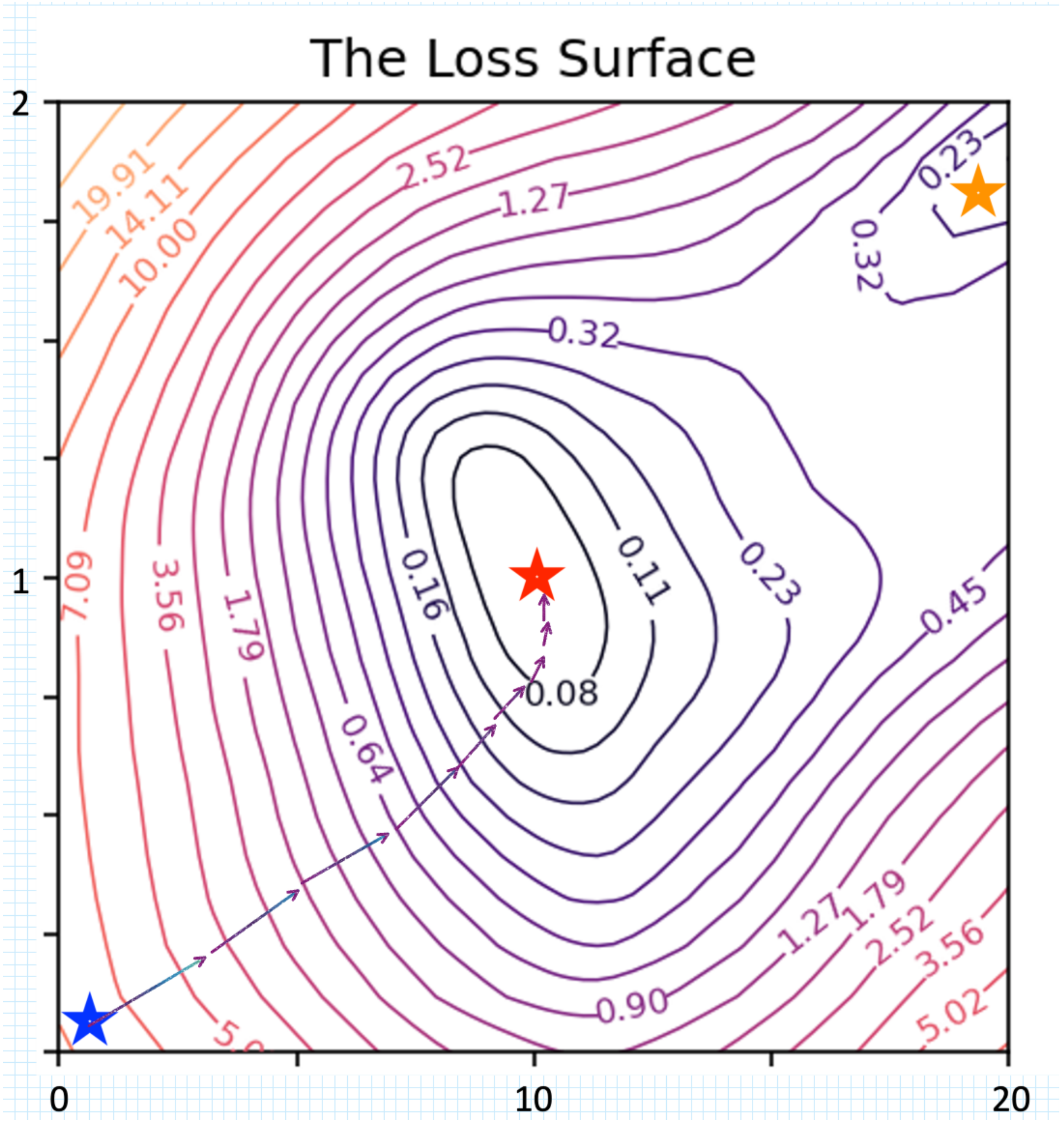

Figure 1: Gradient's direction illustrated by an arrow per step

Above in figure 1, I have drawn a simulated optimization using gradient descent using MSE. Each arrow direction represents the gradient direction and the length of the arrow represents the length of the step. The assignment text says to assume an efficiently small learning rate so in reality, the steps would be a bit smaller than shown above.

## 1e

A gradient of a loss function indicates the steepest direction of ascent. Since we want the gradient to decrease, subtract the gradient, making the gradient the steepest direction of descent.
The contour lines above represent the loss function's value for different weights. Gradient descent will let the weights adjust to make the loss as small as possible, like a ball traveling down a slope.
With a small learning rate, the steps will be small, ensuring that the trajectory follows the contour lines closely and minimizes the loss by a small amount for each time.
When the weights converge, the gradient decreases and the steps become smaller and smaller as they get smaller to the global minimum.

## 1f

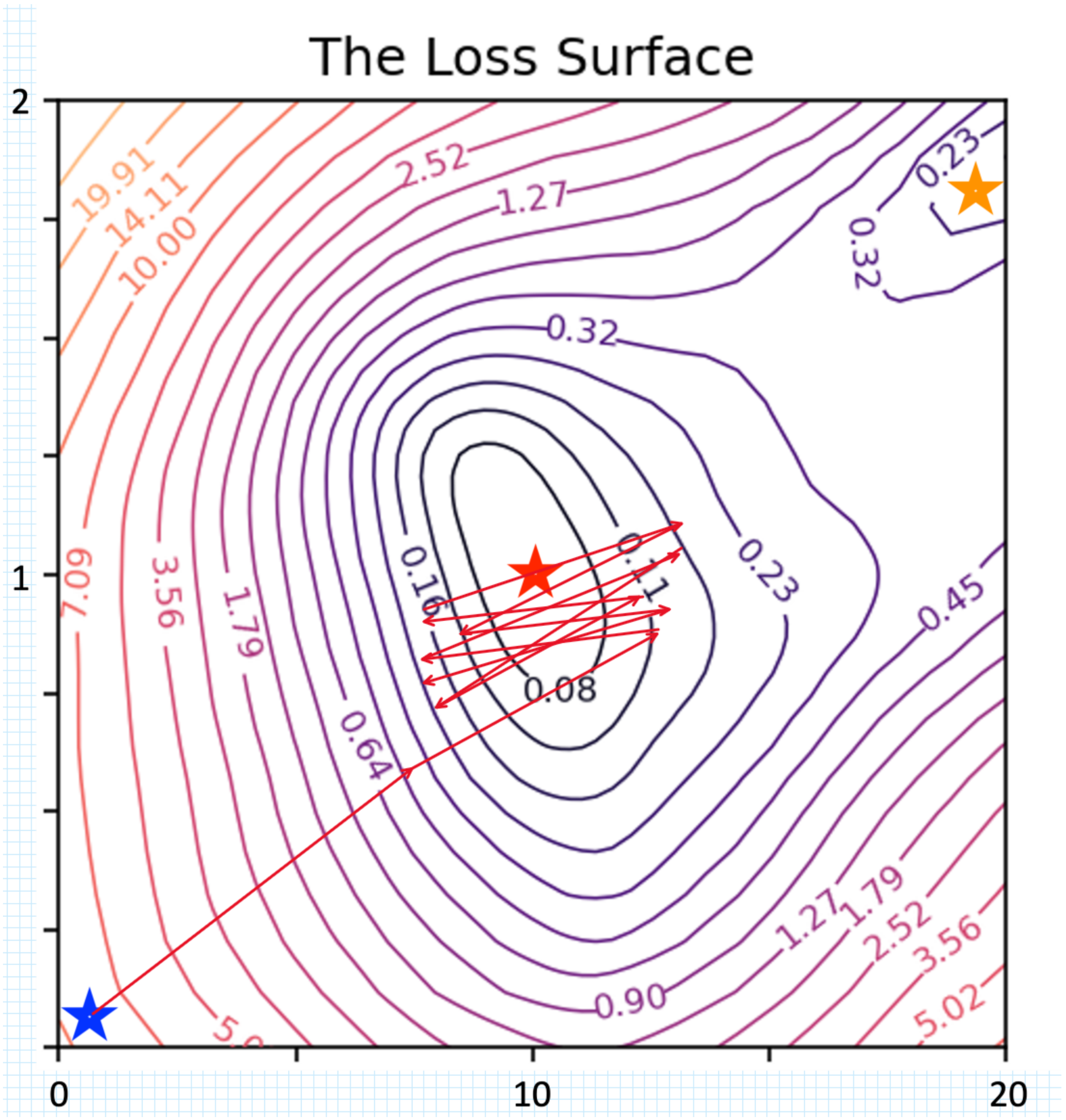

Figure 2: Gradient's direction illustrated by an arrow per step with a high learning rate

As seen in figure 2 the gradient descent will jump from one point to the next not being able to converge the weights. This is caused by a high learning rate.

## 1g

**Strategy 1: random start values**. If we use random start values for the weights, we can explore different local minimums, checking which of them is global.

**Strategy 2: Start with a high learning rate**. If we start with a high learning rate, we could potentially escape local minimums at the start and when the learning rate is decreased the gradient descent is settled in a trajectory towards the global minimum.

## Problem 2

### 2a

I used a simple code to derive the data from the csv file and divide it into training and test sets, with the training data in the two classes divided separately:

```python
# Read csv file
df = pd.read_csv("data_problem2.csv", header=None)

# Get array of samples and labels
samples = df.iloc[0, :].values
labels = df.iloc[1, :].values

# Ratio of trainingset to test set
ratio = int(len(samples)*0.8)

# Divide into training and test data
train_data = samples[:ratio]
test_data = samples[ratio:]
train_labels = labels[:ratio]
test_labels = labels[ratio:]


# Divide training data into two classes
train_data_0 = np.array([train_data[i] for i in range(len(train_data)) if (train_labels[i]
    == 0.0)])
train_data_1 = np.array([train_data[i] for i in range(len(train_data)) if (train_labels[i]
    == 1.0)])
```

I got some info from the training data with this simple function:

```python
def derive_info_data():
    """Function to derive simple information from the data"""
    print(f"Number of sampels is {len(samples)}")
    print(f"Number of training samples: {len(train_data)}")
    print(f"Number of train samples in class 0: {len(train_data_0)}")
    print(f"Number of train samples in class 1: {len(train_data_1)}")

    numbins1 = 50
    numbins2 = 80

    # Plot samples as histogram with improvements:
    plt.figure(figsize=(10, 6))  # Adjusts the size of the figure
    plt.hist(train_data_0, bins=numbins1, color='blue', alpha=0.7, label='Class 0',
    edgecolor='black')
    plt.hist(train_data_1, bins=numbins2, color='green', alpha=0.7, label='Class 1',
    edgecolor='black')

    # Add titles and labels
    plt.title('Histogram of Train Data', fontsize=16)
    plt.xlabel('Sample Values', fontsize=14)
    plt.ylabel('Frequency', fontsize=14)

    # Add a legend
    plt.legend(loc='upper right')

    # Show gridlines for better readability
    plt.grid(True, linestyle='--', alpha=0.6)
    plt.savefig("samples_histogram.png")
    # Display the plot
    plt.show()
```

The functions output shows that:
Number of samples is 3600
Number of training samples: 2880
Number of train samples in class 0: 1266
Number of train samples in class 1: 1614
As seen above, our training data consists of 2880 samples, where there are more samples of in class 1 than in class 0.
The plot of the sample data is here:

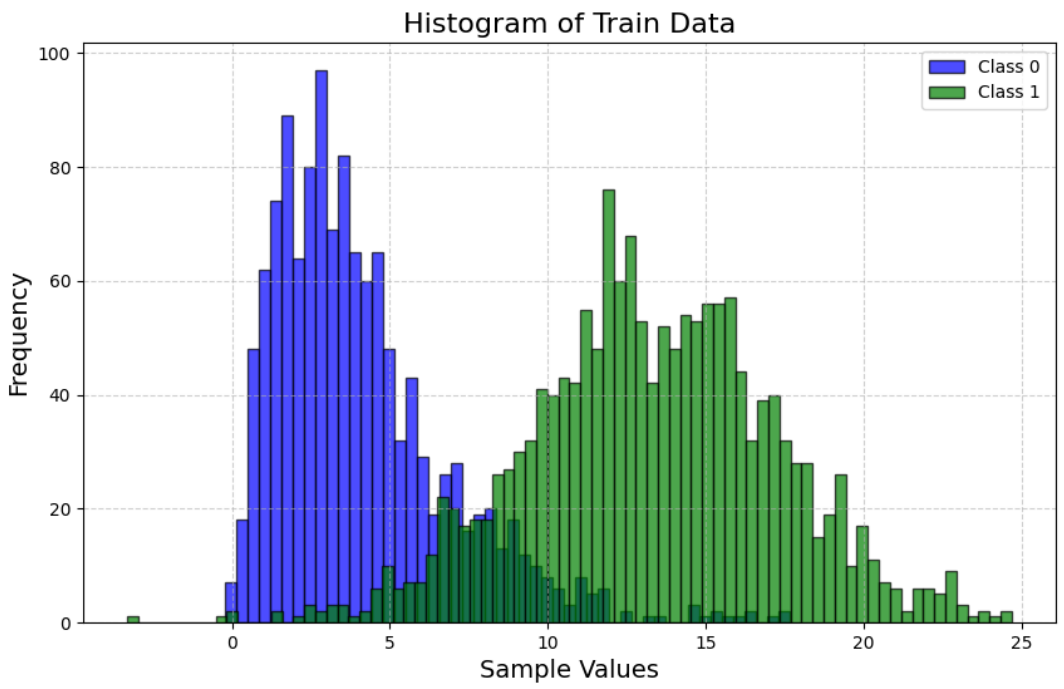

Figure 3: Plot of sample values

You can see in figure 3 that the distribution of class 0 is similar to a gamma distribution and the distribution of class 1 is similar to a Gaussian one. The data also have some overlap where the values are between 5 and 10 so it is likely many values will be misclassified in that interval.

## 2b

Using the formula for finding MLE for a parameter we can find beta, mean and sigma:

Finding the maximum likelihood estimation for $\beta$

$$\beta_{MLE} = \prod_{j=1}^{n_0} \frac{1}{\beta^\alpha \Gamma(\alpha)} x_0^{j^{\alpha-1}} e^{-\frac{x_0^j}{\beta}}$$

$$= (\frac{1}{\beta^\alpha \Gamma(\alpha)})^{n_0} \prod_{j=1}^{n_0} x_0^{j^{\alpha-1}} e^{-\frac{x_0^j}{\beta}}$$

$$ln(\beta_{MLE}) = -ln(\beta^{\alpha n_0}) - ln(\Gamma(\alpha)^{n_0} \ + \sum_{j=1}^{n_0} ln(x_0^{j^{\alpha-1}}) + \sum_{j=1}^{n_0} ln(e^{-\frac{x_0^j}{\beta}})$$

$$= -\alpha n_0 ln(\beta) - n_0 ln(\Gamma(\alpha) + (\alpha - 1)\sum_{j=1}^{n_0} ln(x_0^j) - \frac{1}{\beta}\sum_{j=1}^{n_0} x_0^j$$

$$\frac{d}{d\beta} ln(\beta_{MLE}) = -\frac{\alpha n_0}{\beta} + \frac{1}{\beta^2}\sum_{j=1}^{n_0} x_0^j$$

$$-\frac{\alpha n_0}{\beta} + \frac{1}{\beta^2}\sum_{j=1}^{n_0} x_0^j = 0$$

$$\frac{\alpha n_0}{\beta} = \frac{1}{\beta^2}\sum_{j=1}^{n_0} x_0^j$$

$$\beta = \frac{1}{\alpha n_0}\sum_{j=1}^{n_0} x_0^j$$

Estimate for $\beta$

$$\underline{\underline{\hat{\beta} = \frac{1}{\alpha n_0}\sum_{j=1}^{n_0} x_0^j}}$$

Finding the maximum likelihood estimation for $\mu$

$$\mu_{MLE} = \prod_{j=1}^{n_1} \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}(\frac{x_1^j - \mu}{\sigma})^2}$$

$$ln(\mu_{MLE}) = n_1 ln(\frac{1}{\sigma\sqrt{2\pi}}) + \sum_{j=1}^{n_1} ln(e^{-\frac{1}{2}(\frac{x_1^j - \mu}{\sigma})^2})$$

$$= -n_1 ln(\sigma) - n_1 ln(\sqrt{2\pi}) - \frac{1}{2}\sum_{j=1}^{n_1}(\frac{x_1^j - \mu}{\sigma})^2$$

$$= -n_1 ln(\sigma) - n_1 ln(\sqrt{2\pi}) - \frac{1}{2\sigma^2}\sum_{j=1}^{n_1}(x_1^{j^2} - 2x_1^j\mu + \mu^2)$$

$$\frac{d}{d\mu} ln(\mu_{MLE}) = -\frac{1}{2\sigma^2}\sum_{j=1}^{n_1}(-2x_1^j + 2\mu)$$

$$-\frac{1}{\sigma^2}\sum_{j=1}^{n_1}(-x_1^j + \mu) = 0$$

$$\frac{1}{\sigma^2}\sum_{j=1}^{n_1}(x_1^j) - \frac{n_1\mu}{\sigma^2} = 0$$

$$n_1\mu = \sum_{j=1}^{n_1}(x_1^j)$$

$$\mu = \frac{1}{n_1}\sum_{j=1}^{n_1}(x_1^j)$$

Estimate for $\mu$:

$$\underline{\underline{\hat{\mu} = \frac{1}{n_1}\sum_{j=1}^{n_1}(x_1^j)}}$$

Finding the maximum likelihood estimation for $\sigma^2$

$$L(\sigma) = \prod_{j=1}^{n_1} \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}(\frac{x_1^j-\mu}{\sigma})^2}$$

The same equation as when estimating $\hat{\mu}$ so end up at

$$ln(L(\sigma)) = -n_1 ln(\sigma) - n_1 ln(\sqrt{2\pi}) - \frac{1}{2\sigma^2}\sum_{j=1}^{n_1}(x_1^j-\mu)^2$$

$$\frac{d}{d\sigma} ln(L(\sigma)) = -\frac{n_1}{\sigma} + \frac{1}{\sigma^3}\sum_{j=1}^{n_1}(x_1^j-\mu)^2 = 0$$

$$n_1\sigma^2 = \sum_{j=1}^{n_1}(x_1^j-\mu)^2$$

$$\sigma^2 = \frac{1}{n_1}\sum_{j=1}^{n_1}(x_1^j-\mu)^2$$

Estimate for $\sigma^2$:

$$\hat{\sigma}^2 = \underline{\underline{\frac{1}{n_1}\sum_{j=1}^{n_1}(x_1^j-\mu)^2}}$$

## 2c

Using the estimations above I estimated beta, sigma and mean in the functions below:

```
# Calculate beta, mean, sigma
def calc_beta_hat(samples):
    alpha = 2
    sum_samp = np.sum(samples)
    return sum_samp/(alpha*len(samples))

def calc_mean_hat(samples):
    sum_samp = np.sum(samples)
    return sum_samp /(len(samples))

def calc_sigma_hat(samples, mean_hat):
    sumation = np.sum((samples-mean_hat)**2)
    return sumation/len(samples)
```

The input of the calc_beta_hat() function was the training data which contains samples of class 0. The other two function's input was samples from class 1.
My estimations were:

$$\hat{\beta} = 2.03$$

$$\hat{\mu} = 13.24$$

$$\hat{\sigma}^2 = 15.71$$

Using this plot:

```
def plot_both_distro(mean, sd, alpha, beta):
    x = np.linspace(-20, 100, 241)
    y1 = stats.norm.pdf(x, mean, sd) * p_C1
    y2 = stats.gamma.pdf(x, a=alpha, scale=(beta)) * p_C0

     # Create the plot
    plt.figure(figsize=(10, 6))  # Adjust figure size
    # Plot Gaussian distribution
    plt.plot(x, y1, label='Gaussian Distribution', color='blue', linestyle='-', linewidth=2)
    # Plot Gamma distribution
    plt.plot(x, y2, label='Gamma Distribution', color='green', linestyle='--', linewidth=2)
    # Add titles and labels
    plt.title('Comparison of Gaussian and Gamma Distributions', fontsize=16, fontweight='
    bold')
    plt.xlabel('x', fontsize=14)
    plt.ylabel('Probability Density', fontsize=14)
    # Add grid for better readability
    plt.grid(True, linestyle='--', alpha=0.6)
    # Add a legend
    plt.legend(fontsize=12)
    plt.savefig("both_distrobutions.png")
    # Show the plot
    plt.show()
```

I can show the estimated distributions:

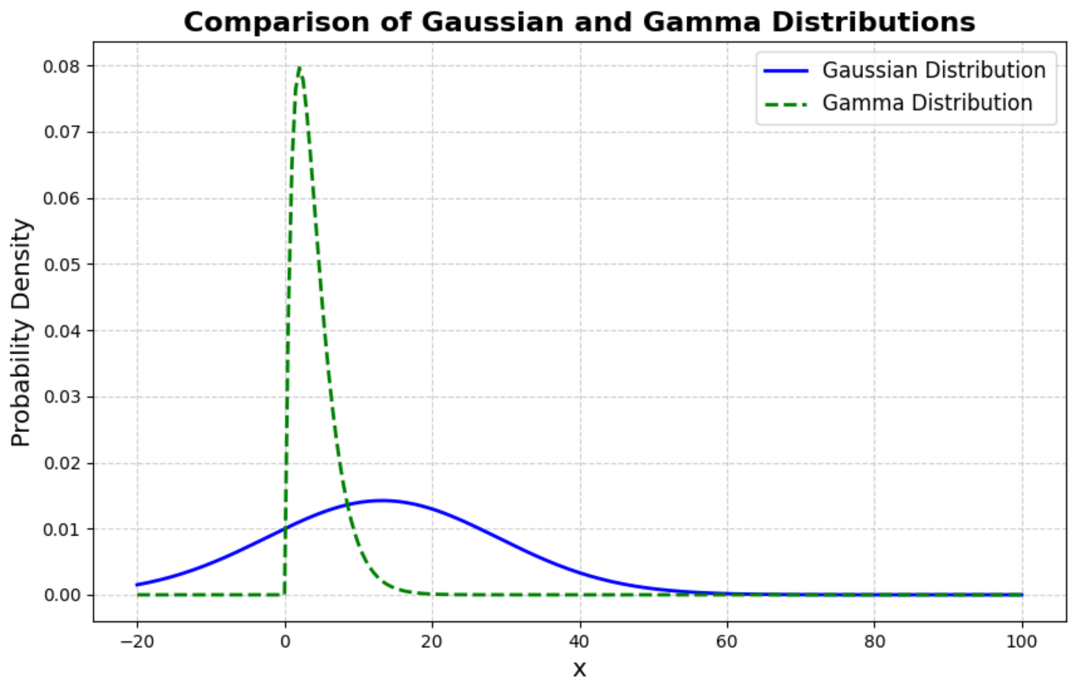

Figure 4: Estimated Distributions

The accuracy was estimated by these functions:

```

def test_one_sample(xi, mean, sd, alpha, beta):
    p0 = stats.gamma.pdf(xi, a=alpha, scale=beta)
    p1 = stats.norm.pdf(xi, mean, sd)
    if (p0*p_C0) > (p1*p_C1):
        return 0
    else:
        return 1

def test_samples(test_samples, test_labels, mean, sd, alpha, beta):
    classified_0 = 0
    classified_1 = 0
    class0_true = 0
    class0_false = 0
    class1_true = 0
    class1_false = 0

    for i in range(len(test_samples)):
        if(not test_one_sample(test_samples[i], mean, sd, alpha, beta)):
            classified_0+=1
            if (not test_labels[i]):
                class0_true+=1
                continue
            class0_false+=1
        else:
            classified_1+=1
            if (test_labels[i]):
                class1_true+=1
                continue
            class1_false+=1

    accuracy = (class0_true + class1_true)/len(test_samples)
    print(f"of {len(test_samples)} samples --- C0 got {classified_0} and C1 got {
    classified_1}")
    print(f"in C0, {class0_true} was true and {class0_false} was false")
    print(f"in C1, {class1_true} was true and {class1_false} was false")
    print(f"Accuracy was {accuracy}%")
```

And I found that the accuracy was around 89.3%

## 2d

Assuming the distribution probability is known, the Bayes' classifier will choose the class that has the highest possibility of a sample belonging to it. Another classifier would therefore not receive a lower probability of misclassification.

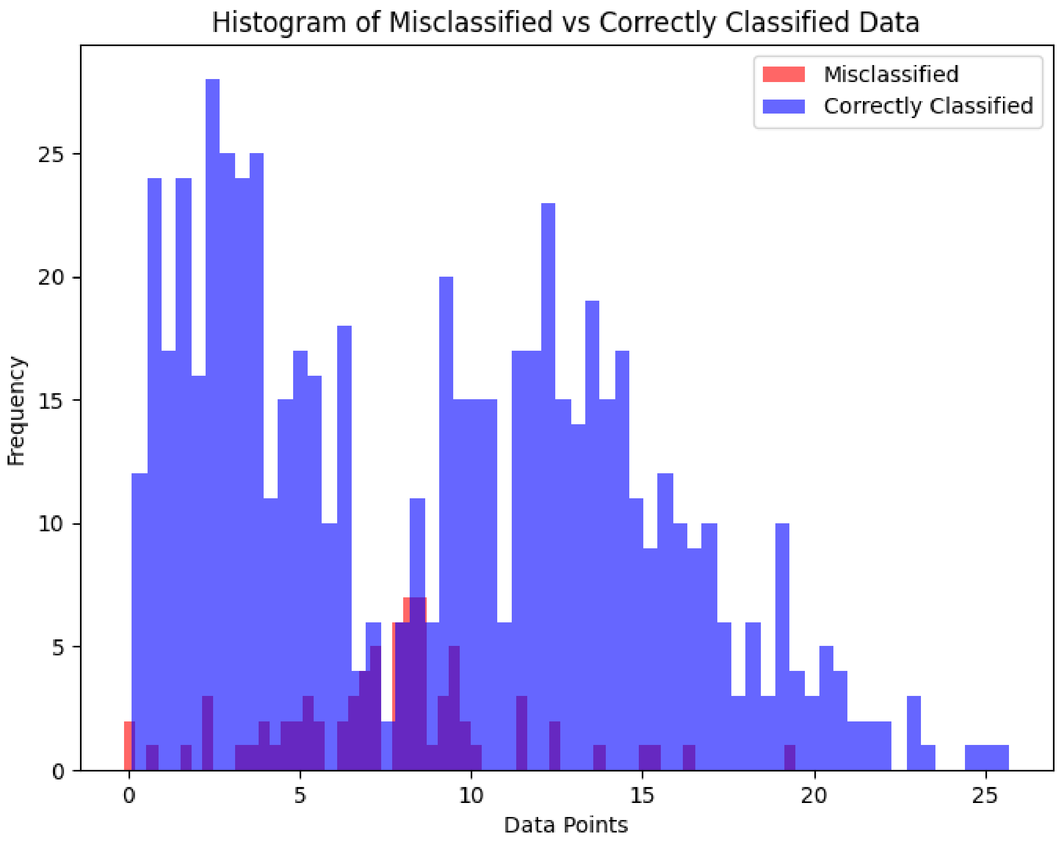

Figure 5: Misclassified samples vs correctly classified samples

In figure 5 you see a plot of all the misclassified data with the correctly classified data. The outcome was expected considering how Bayes' classification works. If $p_(x_i|C_o)p(C_0) > p_(x_i|C_1)p(C_1)$, the Bayes' classification says the sample $x_i$ most likely is in class $C_0$. Looking at figure 4 You can see that all values that is higher than $\approx 8.5$ and $< 0$ will be classified as class 1, and all between will be classified as class 0. This supports the claim in a) that most misclassifications will happen between 5 and 10, considering the probability of a misclassification is higher in this interval than othervise.