FYS-2021
Mandatory Assignment 2

Tor-Ivar Hassfjord

UiT - Norges Arktiske Universitet

Summary

This report examines the optimization of linear regression using gradient descent and the application of probabilistic classification methods. In the first section, I analyze various loss functions, including Mean Squared Error (MSE), Mean Absolute Error (MAE), and Huber Loss. I derive their gradients, highlighting the importance of smooth loss functions for efficient convergence in gradient descent. The exploration of these loss functions illustrates the trade-offs in choosing the right one for regression tasks.

In the second section, I utilize maximum likelihood estimation to derive parameters for a Gamma distribution and a Gaussian distribution based on the provided dataset. Implementing a Bayes classifier, I achieve a classification accuracy of 91.81%, while the analysis of misclassifications reveals insights into the challenges posed by overlapping class distributions.

Contents

1 Problem 1: Linear Regression with Gradient Descent

1.1 a) Loss Functions Used for Linear Regression

In linear regression, the goal is to model the relationship between a dependent variable y and one or more independent variables x by fitting a linear equation to observed data. To find the best-fitting line, we minimize a loss function that measures the discrepancy between the predicted values $\hat{y}$ and the actual values y.

Three common loss functions for linear regression are:

- **Mean Squared Error (MSE)**
- **Mean Absolute Error (MAE)**
- **Huber Loss**

Mean Squared Error

The most commonly used loss function is the **Mean Squared Error (MSE)**. This loss function measures the average of the squared differences between the predicted values $\hat{y}$ and the actual values y.
the mathematical formula for the MSE loss function is given by:

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

where:

- N is the number of data points
- y_i is the actual value for the i-th data point.
- $\hat{y}_i$ is the predicted value for the i-th data point.

Derivation of the Gradient

To be able to minimize this loss function, we need to compute the gradient of L_{reg} with respect to the weight parameters ω_0 and ω_1.

For a linear model the prediction is given by:

$$\hat{y} = \omega_0 + \omega_1 x_i$$

Where:

- ω_0 is the intercept.
- ω_1 is the slope (or weight for the feature x_i).
- x_i is the input feature.

To be able to minimize the loss, we compute the gradient of L_{reg} with respect to ω_0 and ω_1.

Gradient with respect to ω_0:

$$\frac{\partial L_{reg}}{\partial \omega_0} = \frac{\partial}{\partial \omega_0}\left(\frac{1}{N}\sum_{i=1}^{N}(y_i - (\omega_0 + \omega_1 x_i))^2\right) \tag{1}$$

We can use the chain rule here, and note that $\omega_0 + \omega_1 x_i = \hat{y}_i$, giving us:

$$\frac{\partial L_{reg}}{\partial \omega_0} = \frac{\partial}{\partial \omega_0}\left(\frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2\right) \tag{2}$$

Applying the chain rule $(f(g(x)))' = f'(g(x))g'(x)$:

$$\frac{\partial}{\partial \omega_0}(y_i - \hat{y}_i)^2 = 2(y_i - \hat{y}_i) \cdot \frac{\partial}{\partial \omega_0}(-\hat{y}_i) \tag{3}$$

Since $\frac{\partial \hat{y}_i}{\partial \omega_0} = 1$ (because $\hat{y}_i = \omega_0 + \omega_1 x_i$), this simplifies to:

$$\frac{\partial L_{reg}}{\partial \omega_0} = \frac{2}{N}\sum_{i=1}^{N}(\hat{y}_i - y_i) \tag{4}$$

Gradient with respect to ω_1:

$$\frac{\partial L_{reg}}{\partial \omega_1} = \frac{\partial}{\partial \omega_1}\left(\frac{1}{N}\sum_{i=1}^{N}(y_i - (\omega_0 + \omega_1 x_i))^2\right) \tag{5}$$

chain rule:

$$\frac{\partial}{\partial w_1}(y_i - \hat{y}_i)^2 = 2(y_i - \hat{y}_i) \cdot \frac{\partial}{\partial w_1}(-\hat{y}_i) \tag{6}$$

Since $\frac{\partial}{\partial w_1}\hat{y}_i = \frac{\partial}{\partial w_1}(w_0 + w_1 x_i) = x_i$, we get:

$$\frac{\partial L_{reg}}{\partial w_1} = \frac{2}{N}\sum_{i=1}^{N}(\hat{y}_i - y_i)x_i \tag{7}$$

To conclude, the final gradients are:

$$\frac{\partial L_{reg}}{\partial w_0} = \frac{2}{N}\sum_{i=1}^{N}(\hat{y}_i - y_i) \tag{8}$$

$$\frac{\partial L_{reg}}{\partial w_1} = \frac{2}{N}\sum_{i=1}^{N}(\hat{y}_i - y_i)x_i \tag{9}$$

The gradients are positive, and the negative sign is explicitly included in the weight update rule in programming.

Mean Absolute Error (MAE)

The **Mean Absolute Error (MAE)** is a commonly used loss function. It measures the average absolute differences between the actual values y_i and the predicted values $\hat{y}_i$.

The formula for MAE is:

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

where:

- N is the number of data points.
- y_i is the actual value for the i-th data point.
- $\hat{y}_i$ is the predicted value for the i-th data point.

Understanding the Gradient of MAE

In linear regression, the predicted value is given by:

$$\hat{y}_i = \omega_0 + \omega_1 x_i$$

We want to find how the MAE changes with respect to the weights ω_0 and ω_1, which requires computing the gradient of the MAE loss function.

Gradient with respect to ω_0:

The absolute value function is not differentiable at zero, so we use a **piecewise function** to derive the gradient:

$$\frac{d}{dx}|x| = \begin{cases} 1, & \text{if } x > 0 \\ -1, & \text{if } x < 0 \end{cases}$$

The gradient of MAE with respect to ω_0 becomes:

$$\frac{\partial L_{MAE}}{\partial \omega_0} = -\frac{1}{N}\sum_{i=1}^{N}\text{sign}(y_i - \hat{y}_i)$$

Where:

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

This means if the prediction $\hat{y}_i$ is too low compared to y_i, we increase ω_0, and if it's too high, we decrease ω_0.

Gradient with respect to ω_1:

The process for ω_1 is similar:

$$\frac{\partial L_{MAE}}{\partial \omega_1} = \frac{1}{N}\sum_{i=1}^{N}\text{sign}(y_i - \hat{y}_i)\cdot(-x_i)$$

Here, we adjust ω_1 (the slope) based on both the sign of the error and the value of x_i.

Conclusion The gradients derived from the MAE loss function help determine whether to increase or decrease the weights ω_0 and ω_1 to reduce the error in future predictions.

Huber Loss

The **Huber Loss** combines the advantages of both the Mean Squared Error (MSE) and the Mean Absolute Error (MAE). It is less sensitive to outliers compared to MSE and differentiable, unlike MAE, making it useful in many regression tasks. The Huber Loss behaves like MSE for small errors and like MAE for larger errors.

The Huber Loss is defined as:

$$L_{Huber}(\hat{y}, y) = \begin{cases} \frac{1}{2}(y_i - \hat{y}_i)^2, & \text{for } |y_i - \hat{y}_i| \leq \delta \\ \delta \cdot (|y_i - \hat{y}_i| - \frac{1}{2}\delta), & \text{for } |y_i - \hat{y}_i| > \delta \end{cases}$$

Where:

- y_i is the actual value,
- $\hat{y}_i$ is the predicted value,
- δ is a threshold that controls the transition between MSE and MAE behavior.

Derivation of the Gradient for Huber Loss

As for other loss functions, the predicted value $\hat{y}_i$ is given by:

$$\hat{y}_i = \omega_0 + \omega_1 x_i$$

Gradient with respect to ω_0:
The gradient depends on whether $|y_i - \hat{y}_i|$ is smaller or larger than δ. For small errors (less than δ), the Huber Loss behaves like MSE, and for large errors (greater than δ), it behaves like MAE.

For the case where $|y_i - \hat{y}_i| \leq \delta$:

$$\frac{\partial L_{Huber}}{\partial \omega_0} = \frac{1}{N} \sum_{i=1}^{N} (\hat{y}_i - y_i)$$

For the case where $|y_i - \hat{y}_i| > \delta$:

$$\frac{\partial L_{Huber}}{\partial \omega_0} = \frac{1}{N} \sum_{i=1}^{N} \delta \cdot \text{sign}(y_i - \hat{y}_i)$$

Thus, the gradient with respect to ω_0 is piecewise depending on the error magnitude.

Gradient with respect to ω_1:
Similarly, for ω_1: For small errors:

$$\frac{\partial L_{Huber}}{\partial \omega_1} = \frac{1}{N} \sum_{i=1}^{N} (\hat{y}_i - y_i) x_i$$

For large errors:

$$\frac{\partial L_{Huber}}{\partial \omega_1} = \frac{1}{N} \sum_{i=1}^{N} \delta \cdot \text{sign}(y_i - \hat{y}_i) x_i$$

Summary of Loss Functions

Each loss function has its advantages and drawbacks:

- **MSE** is sensitive to outliers but provides smooth gradients, making optimization straightforward.
- **MAE** is robust to outliers but provides constant gradients, which can slow down convergence.
- **Huber Loss** offers a balance between MSE and MAE, being less sensitive to outliers while maintaining differentiability.

The choice of loss function depends on the specific requirements of the problem, such as the presence of outliers and the desired robustness.

1.2 b) Importance of Smooth Loss Functions in Gradient Descent

Smooth loss functions are preferred in gradient descent optimization because they are differentiable everywhere, providing well-defined gradients that guide the optimization process efficiently towards a global minimum.

Non-Differentiability Issues with MAE

The MAE loss function is not differentiable at $y_i = \hat{y}_i$ due to the absolute value function. This can lead to challenges:

- **Constant Gradient Magnitude:** The gradient magnitude does not change with the error magnitude, leading to slow convergence.
- **Optimization Stability:** Non-differentiability can cause instability in optimization algorithms that rely on gradient information.

Advantages of Smooth Loss Functions

Using a smooth loss function like MSE or Huber Loss (with appropriate δ):

- **Well-Behaved Gradients:** Differentiable loss functions provide gradients that change smoothly with the error magnitude, allowing for efficient updates.
- **Convergence Speed:** Smooth gradients enable faster convergence to the minimum.
- **Optimization Stability:** Gradient-based optimization algorithms perform better with continuous and differentiable loss surfaces.

1.3 c) Gradient Descent Optimization Process

Gradient Descent is an iterative optimization algorithm used to minimize a loss function by adjusting the model's parameters in the opposite direction of the gradient.

Algorithm Steps

1. **Initialize Weights:** Start with initial weights $w_0^{(0)}$, $w_1^{(0)}$.
2. **Compute Predictions:** For each data point, compute $\hat{y}_i^{(k)} = w_0^{(k)} + w_1^{(k)} x_i$.
3. **Compute Loss:** Calculate the loss $L^{(k)}$ using the chosen loss function.
4. **Compute Gradients:** Compute the gradients $\frac{\partial L}{\partial w_0}$ and $\frac{\partial L}{\partial w_1}$.
5. **Update Weights:** Update the weights using:

$$w_0^{(k+1)} = w_0^{(k)} - \eta \frac{\partial L}{\partial w_0}, \quad w_1^{(k+1)} = w_1^{(k)} - \eta \frac{\partial L}{\partial w_1}$$

   where η is the learning rate.

6. **Repeat:** Iterate steps 2-5 until convergence criteria are met (e.g., the change in loss is below a threshold or a maximum number of iterations is reached).

Learning Rate Considerations

The learning rate η controls the size of the weight updates:

- **Small η:** Leads to small updates, potentially slow convergence but more stable.
- **Large η:** Leads to large updates, faster convergence but risks overshooting the minimum or divergence.

Choosing an appropriate learning rate is critical for effective optimization.

1.4 d) Loss Surface Visualization

To understand the behavior of the optimization process, we visualize the loss surface and the trajectory of the weights during gradient descent.

Visualization Code

```python
import numpy as np
import matplotlib.pyplot as plt

# Generate synthetic data
np.random.seed(42)
N = 100
x = 2 * np.random.rand(N)
y = 4 + 3 * x + np.random.randn(N)

# Define the loss function (MSE)
def mse_loss(w0, w1, x, y):
    y_pred = w0 + w1 * x
    return np.mean((y - y_pred) ** 2)

# Create a grid of weight values
w0_values = np.linspace(0, 8, 100)
w1_values = np.linspace(0, 6, 100)
W0, W1 = np.meshgrid(w0_values, w1_values)
Loss = np.array([[mse_loss(w0, w1, x, y) for w0 in w0_values] for w1 in w1_values])

# Plot the loss surface
plt.figure(figsize=(10, 6))
CS = plt.contour(W0, W1, Loss, levels=50, cmap='viridis')
plt.clabel(CS, inline=1, fontsize=8)
plt.xlabel('w0')
plt.ylabel('w1')
plt.title('Loss Surface (MSE)')
```

Listing 1: Visualizing the Loss Surface

Gradient Descent Trajectory

Instead of drawing on the given loss surface image, I tried to perform gradient descent and plot the trajectory of the weights on the loss surface instead.

```python
# Initialize weights
w0, w1 = 0.0, 0.0
learning_rate = 0.1
iterations = 50
w0_history = [w0]
w1_history = [w1]

# Gradient descent loop
for i in range(iterations):
    y_pred = w0 + w1 * x
    error = y_pred - y
    grad_w0 = (2 / N) * np.sum(error)
    grad_w1 = (2 / N) * np.sum(error * x)
    w0 -= learning_rate * grad_w0
    w1 -= learning_rate * grad_w1
    w0_history.append(w0)
    w1_history.append(w1)

# Plot the trajectory
plt.plot(w0_history, w1_history, 'ro-', label='Gradient Descent Path')
plt.legend()
plt.show()
```

Listing 2: Gradient Descent Trajectory

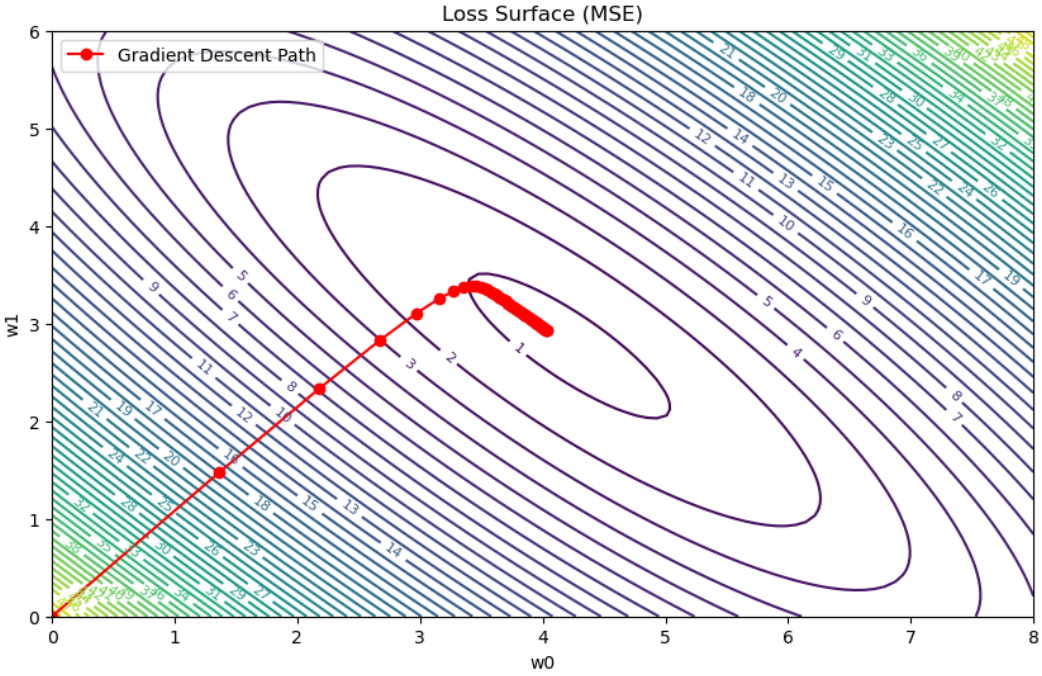

Figure 1: Loss surface visualization with gradient descent trajectory. The path shows the weights updating towards the global minimum.

1.5 e) Trajectory Derivation

The trajectory observed in the loss surface visualization results from the iterative updates of the weights guided by the gradients.

Mathematical Justification

At each iteration:

$$w_j^{(k+1)} = w_j^{(k)} - \eta \frac{\partial L}{\partial w_j}$$

For MSE loss, the gradients are:

$$\frac{\partial L_{\text{MSE}}}{\partial w_0} = -\frac{2}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i), \quad \frac{\partial L_{\text{MSE}}}{\partial w_1} = -\frac{2}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)x_i$$

The negative gradients point in the direction of steepest descent, and the learning rate η scales the step size. The iterative updates cause the weights to move along the path of steepest descent towards the minimum of the loss surface.

Connection to Visualization

In Figure 1, the red dots represent the sequence of weight values $(w_0^{(k)}, w_1^{(k)})$. The path shows how the weights are updated at each iteration, gradually approaching the point that minimizes the MSE loss.

1.6 f) Impact of Large Learning Rate on Gradient Descent

A large learning rate can adversely affect the convergence of gradient descent.

Experiment with Large Learning Rate

We set a large learning rate $\eta = 1.5$ and observe the behavior.

```
# Initialize weights
w0, w1 = 0.0, 0.0
learning_rate = 1.5  # Large learning rate
iterations = 10
w0_history = [w0]
w1_history = [w1]

# Gradient descent loop
for i in range(iterations):
    y_pred = w0 + w1 * x
    error = y_pred - y
    grad_w0 = (2 / N) * np.sum(error)
    grad_w1 = (2 / N) * np.sum(error * x)
    w0 -= learning_rate * grad_w0
    w1 -= learning_rate * grad_w1
    w0_history.append(w0)
    w1_history.append(w1)

# Plot the trajectory
plt.figure(figsize=(10, 6))
CS = plt.contour(W0, W1, Loss, levels=50, cmap='viridis')
plt.clabel(CS, inline=1, fontsize=8)
plt.plot(w0_history, w1_history, 'ro-', label='Gradient Descent Path with Large Learning Rate')
plt.xlabel('w0')
plt.ylabel('w1')
plt.title('Loss Surface (MSE) with Large Learning Rate')
plt.legend()
plt.show()
```

Listing 3: Gradient Descent with Large Learning Rate

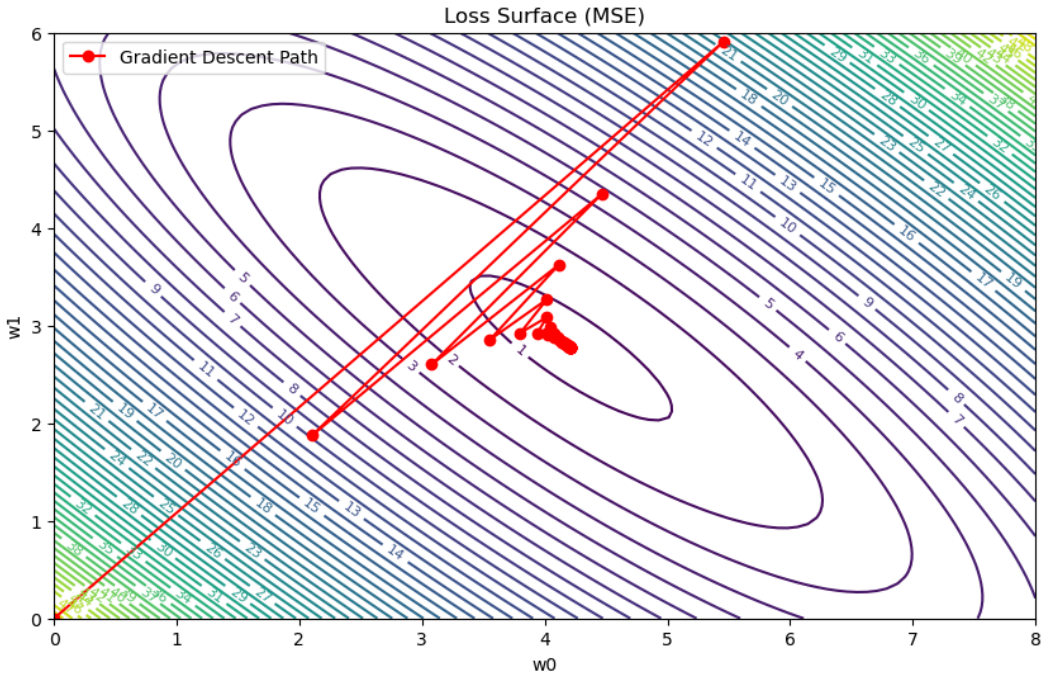

Figure 2: Gradient descent trajectory with a large learning rate. The path shows oscillations and divergence from the minimum.

Analysis

With a large learning rate, the weight updates become too large, causing the weights to overshoot the minimum and potentially diverge. The trajectory exhibits oscillations and fails to converge to the optimal weights in a reasonable manner.

If this was MAE, the constant gradient could potentially just oscillate around the minimum(global or local) for ever.

Conclusion

Choosing an appropriate learning rate is crucial. If the learning rate is too large, the optimization may not converge. One way of checking this is by plotting it as I have in this report.

1.7 g) Local Minima and Solutions

In some optimization problems, the weights can get stuck in local minima, represented by the orange star on the loss surface. This happens because the optimization algorithm follows the gradients, and in non-convex loss functions, it might settle in a local minimum rather than finding the global minimum.

Suggested Strategy: Momentum

One strategy to solve this problem is by using **momentum**. Momentum helps the optimization process avoid getting stuck by allowing it to "build speed" in directions where the gradient consistently points, making it easier to escape local minima. In simple terms, momentum gives the optimization a "push" based on its previous steps, which prevents it from settling in local dips on the loss surface.

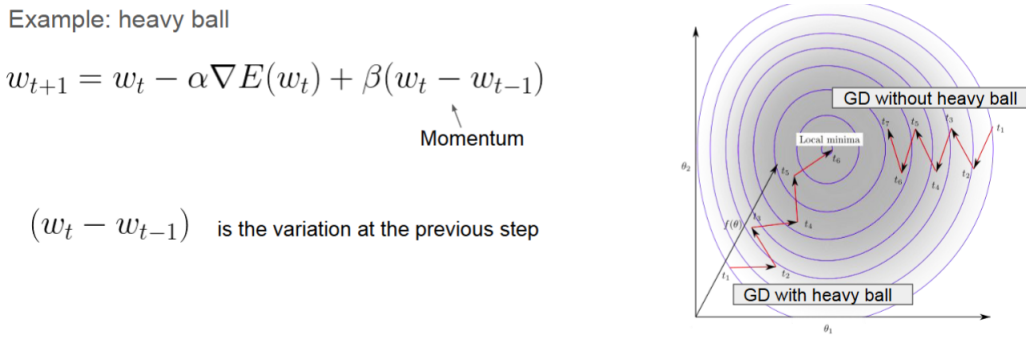

Figure 3: Example from lecture on Optimization and Gradient Descent (Benjamin Ricaud 2024)

Alternative Strategy: Adaptive Learning Rates

Another approach is to use **adaptive learning rates**. This technique adjusts the learning rate during the optimization process. For example, if the algorithm detects that it's moving slowly or getting stuck, it can increase the learning rate to make bigger steps, helping it escape local minima. On the other hand, it can reduce the learning rate when getting closer to the optimal solution to ensure more precise updates.

Summary

Both momentum and adaptive learning rates are effective ways to avoid the issue of local minima, ensuring that the optimization continues toward finding the global minimum.

1.8 Conclusion

In this problem, we explored linear regression using different loss functions and gradient descent optimization. We:

- Derived gradients for MSE, MAE, and Huber Loss.

- Discussed the importance of smooth loss functions.
- Demonstrated gradient descent optimization and the impact of learning rate.
- Showed understanding of strategies for circumventing local minima.

This analysis highlights the significance of choosing appropriate loss functions and hyperparameters to ensure efficient and effective optimization in linear regression tasks.

2 Problem 2: Classifying Data

2.1 a) Data Overview and Visualization

Data Description

The dataset consists of **3600 samples** with a single feature and a class label. The samples are divided into two classes:

- **Class 0 (C0):** Contains **1600 samples** generated from a *Gamma distribution* with a known shape parameter $\alpha = 2$ and an unknown scale parameter β.
- **Class 1 (C1):** Contains **2000 samples** generated from a *Gaussian distribution* with unknown mean μ and standard deviation σ.

The objective is to develop a Bayes classifier to distinguish between these two classes based on the feature values.

Data Loading and Exploration

```python
# Step 1: Load and Explore the Data
import numpy as np
import pandas as pd

data = pd.read_csv('data/data_problem2.csv', delimiter=',', header=None)
data = data.T.reset_index(drop=True)
data.columns = ['Feature', 'Class']

# General Information
print(data.info())
print(data.describe())
print(data.head())

# Class Counts
class_counts = data['Class'].value_counts()
print("\nClass Counts:")
print(class_counts)
```

Listing 4: Loading and Exploring the Data

```
[caption=Output]
    <class 'pandas.core.frame.DataFrame'>
RangeIndex: 3600 entries, 0 to 3599
Data columns (total 2 columns):
 #   Column   Non-Null Count  Dtype
---  ------   --------------  -----
 0   Feature  3600 non-null   float64
 1   Class    3600 non-null   float64
dtypes: float64(2)
memory usage: 56.4 KB
None
           Feature        Class
count  3600.000000  3600.000000
mean      9.118580     0.555556
std       5.747397     0.496973
min      -3.310259     0.000000
25%       3.710522     0.000000
50%       9.091724    1.000000
75%      13.729211    1.000000
max      25.673673    1.000000
     Feature  Class
0   8.903629    1.0
1   9.946774    1.0
2  14.458392    1.0
3   9.664572    1.0
4  14.412270    1.0
```

```
...                                     1.0     2000
Class                                   0.0     1600
```

We first load the data using `pandas` and assign appropriate column names. The dataset contains two classes, hence we get 2 columns: Feature and Class. By inspecting the data, we confirm that there are 1600 samples from Class 0 and 2000 samples from Class 1.

Feature and Label Separation

```python
# Features and Labels
X = data['Feature'].values
y = data['Class'].values
```

Listing 5: Separating Features and Labels

We separate the feature values and class labels for further processing.

Data Visualization

```python
# Step 2: Plot Histograms
import matplotlib.pyplot as plt
import seaborn as sns

plt.figure(figsize=(10, 6))
sns.histplot(data=data, x='Feature', hue='Class', bins=30, kde=True, element='step', stat='
    density')
plt.title('Histogram of Feature Values by Class')
plt.xlabel('Feature Value')
plt.ylabel('Density')
plt.show()
```

Listing 6: Plotting Histograms

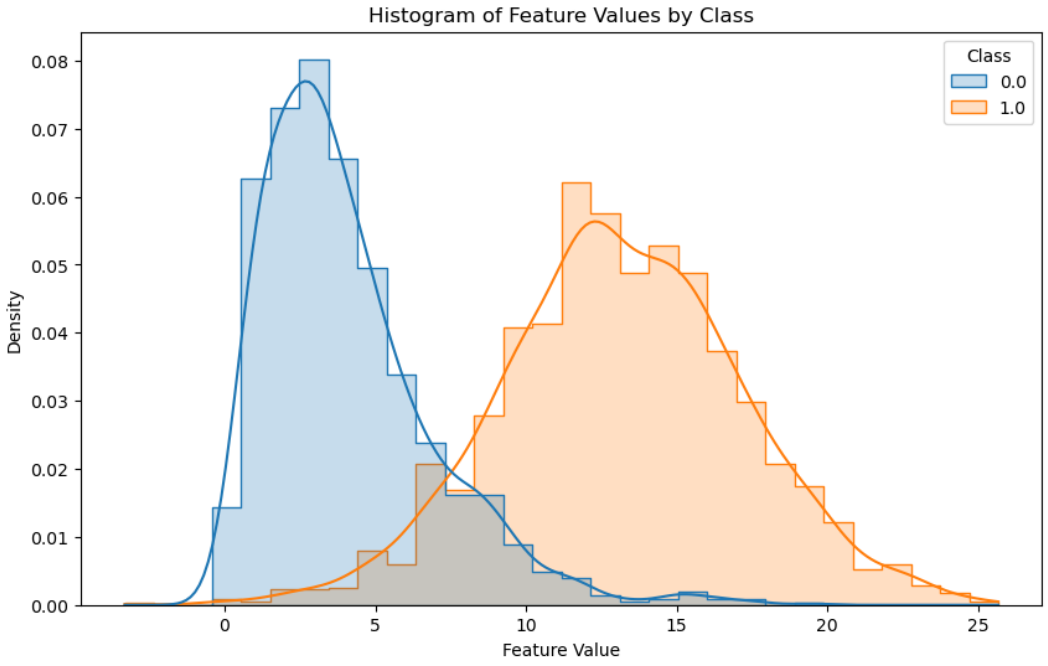

Figure 4: Histogram of feature values for Class 0 and Class 1.

The histogram in Figure 4 shows the distribution of feature values for both classes. Class 0 has a distribution skewed towards lower feature values, consistent with a Gamma distribution, while Class 1 has a distribution centered around a higher mean, consistent with a Gaussian distribution. There is some overlap between the two distributions, which may lead to misclassifications.

2.2 b) Maximum Likelihood Estimations

To develop the Bayes classifier, we need to estimate the parameters of the underlying distributions for each class using Maximum Likelihood Estimation (MLE).

We are tasked with deriving the maximum likelihood estimates (MLE) for the parameters of two distributions: β for the Gamma distribution (Class 0) and μ, σ^2 for the Gaussian distribution (Class 1).

Gamma Distribution (Class 0)

Given the probability density function (PDF) for the Gamma distribution with known $\alpha = 2$:

$$p(x|\beta) = \frac{1}{\beta^2 \Gamma(2)} x^{2-1} e^{-x/\beta}$$

Since $\Gamma(2) = (2-1)! = 1$, this simplifies to:

$$p(x|\beta) = \frac{1}{\beta^2} x e^{-x/\beta}$$

For n_0 independent samples $x_1, x_2, \ldots, x_{n_0}$, the log-likelihood function is:

$$\ell(\beta) = \sum_{j=1}^{n_0} \left(\ln \left(\frac{1}{\beta^2} \right) + \ln x_j - \frac{x_j}{\beta} \right)$$

Simplifying this expression:

$$\ell(\beta) = -2n_0 \ln \beta + \sum_{j=1}^{n_0} \ln x_j - \frac{1}{\beta} \sum_{j=1}^{n_0} x_j$$

Now, take the derivative of the log-likelihood with respect to β:

$$\frac{d\ell}{d\beta} = -\frac{2n_0}{\beta} + \frac{1}{\beta^2} \sum_{j=1}^{n_0} x_j$$

Setting this equal to zero to find the maximum:

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

Multiplying both sides by β^2:

$$0 = -2n_0\beta + \sum_{j=1}^{n_0} x_j$$

Solving for β:

$$\hat{\beta} = \frac{1}{2n_0} \sum_{j=1}^{n_0} x_j$$

Since $\alpha = 2$, we express this as:

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

Gaussian Distribution (Class 1)

The PDF for the Gaussian distribution is:

$$f(x;\mu,\sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right) \tag{10}$$

The likelihood function for n_1 independent samples $x_1^1, \ldots, x_1^{n_1}$ is:

$$L(\mu,\sigma^2) = \prod_{j=1}^{n_1} \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(x_1^j-\mu)^2}{2\sigma^2}\right) \tag{11}$$

Taking the log-likelihood:

$$\log L(\mu,\sigma^2) = -\frac{n_1}{2}\log(2\pi\sigma^2) - \frac{1}{2\sigma^2}\sum_{j=1}^{n_1}(x_1^j-\mu)^2 \tag{12}$$

Step 2: Deriving with Respect to μ
We now take the partial derivative of the log-likelihood with respect to μ:

$$\frac{\partial}{\partial\mu}\log L(\mu,\sigma^2) = \frac{1}{\sigma^2}\sum_{j=1}^{n_1}(x_1^j-\mu) = 0 \tag{13}$$

Solving for μ:

$$\hat{\mu} = \frac{1}{n_1}\sum_{j=1}^{n_1} x_1^j \tag{14}$$

Step 3: Deriving with Respect to σ^2
Next, we derive with respect to σ^2:

$$\frac{\partial}{\partial\sigma^2}\log L(\mu,\sigma^2) = -\frac{n_1}{2\sigma^2} + \frac{1}{2\sigma^4}\sum_{j=1}^{n_1}(x_1^j-\mu)^2 = 0 \tag{15}$$

Multiplying through by $2\sigma^4$:

$$-n_1\sigma^2 + \sum_{j=1}^{n_1}(x_1^j-\mu)^2 = 0 \tag{16}$$

Solving for σ^2:

$$\hat{\sigma}^2 = \frac{1}{n_1}\sum_{j=1}^{n_1}(x_1^j-\hat{\mu})^2 \tag{17}$$

Summary of Results

The MLE for the parameters are as follows:

$$\hat{\beta} = \frac{1}{n_0\alpha}\sum_{j=1}^{n_0} x_0^j, \tag{18}$$

$$\hat{\mu} = \frac{1}{n_1}\sum_{j=1}^{n_1} x_1^j, \tag{19}$$

$$\hat{\sigma}^2 = \frac{1}{n_1}\sum_{j=1}^{n_1}(x_1^j-\hat{\mu})^2 \tag{20}$$

Parameter Estimation Using Code

```
    # Step 4: Estimate Parameters

# Gamma distribution for Class 0
alpha = 2  # Given shape parameter
X_train_c0 = X_train[y_train == 0]
n0 = len(X_train_c0)
beta_hat = np.mean(X_train_c0) / alpha  # Estimate beta
print(f"Estimated beta for Class 0 (Gamma): {beta_hat}")

# Gaussian distribution for Class 1
X_train_c1 = X_train[y_train == 1]
n1 = len(X_train_c1)
mu_hat = np.mean(X_train_c1)  # Estimate mu
sigma_squared_hat = np.var(X_train_c1, ddof=0)  # Estimate sigma^2
sigma_hat = np.sqrt(sigma_squared_hat)  # Estimate sigma
print(f"Estimated mu for Class 1 (Gaussian): {mu_hat}")
print(f"Estimated sigma for Class 1 (Gaussian): {sigma_hat}");

# Step 5: Define PDFs
def gamma_pdf(x, alpha, beta):
    # Since alpha = 2, Gamma(alpha) = 1
    x = np.maximum(x, 1e-10)  # Ensure x > 0
    coef = 1 / (beta ** alpha)  # Simplified coefficient
    pdf = coef * x ** (alpha - 1) * np.exp(-x / beta)
    return pdf

def gaussian_pdf(x, mu, sigma):
    coef = 1 / (sigma * np.sqrt(2 * np.pi))
    exponent = -0.5 * ((x - mu) / sigma) ** 2
    pdf = coef * np.exp(exponent)
    return pdf


# Step 6: Implement Classifier
prior_c0 = np.mean(y_train == 0)
prior_c1 = np.mean(y_train == 1)
print(f"\nClass Prior for Class 0: {prior_c0}")
print(f"Class Prior for Class 1: {prior_c1}");
```

Listing 7: Estimating Parameters Using MLE

Results:

```
Estimated beta for Class 0 (Gamma): 2.0491
Estimated mu for Class 1 (Gaussian): 13.1380
Estimated sigma for Class 1 (Gaussian): 4.0571
```

2.3 c) Bayes Classifier Implementation

The Bayes classifier was implemented using maximum likelihood estimation and prior probabilities .

Defining the Probability Density Functions

Gamma PDF for Class 0

$$f_{C0}(x) = \frac{x^{\alpha-1}e^{-x/\beta}}{\beta^{\alpha}\Gamma(\alpha)}$$

Since $\alpha = 2$, we have $\Gamma(\alpha) = (\alpha - 1)! = 1! = 1$.

```
# Step 5: Define PDFs
def gamma_pdf(x, alpha, beta):
    gamma_alpha = 1  # Since (2 - 1)! = 1! = 1
    x = np.maximum(x, 1e-10)  # Avoid zero or negative values
    coef = (1 / (beta ** alpha * gamma_alpha))
    pdf = coef * x ** (alpha - 1) * np.exp(-x / beta)
    return pdf
```

Listing 8: Defining the Gamma PDF

Gaussian PDF for Class 1

$$f_{C1}(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

```
def gaussian_pdf(x, mu, sigma):
    coef = 1 / (sigma * np.sqrt(2 * np.pi))
    exponent = -0.5 * ((x - mu) / sigma) ** 2
    pdf = coef * np.exp(exponent)
    return pdf
```

Listing 9: Defining the Gaussian PDF

Calculating Class Priors

The class priors are estimated from the training data:

$$P(C0) = \frac{n_0}{n_0 + n_1}, \quad P(C1) = \frac{n_1}{n_0 + n_1}$$

```
# Step 6: Implement Classifier
prior_c0 = n0 / (n0 + n1)
prior_c1 = n1 / (n0 + n1)
print(f"\nClass Prior for Class 0: {prior_c0}")
print(f"Class Prior for Class 1: {prior_c1}")
```

Listing 10: Calculating Class Priors

Results:

```
Class Prior for Class 0: 0.4444
Class Prior for Class 1: 0.5556
```

Implementing the Bayes Classifier

The Bayes classifier assigns a sample x to the class with the highest posterior probability:

$$\text{Predict } C_i \text{ if } P(C_i|x) > P(C_j|x), \quad i \neq j$$

Using Bayes' theorem:

$$P(C_i|x) = \frac{f_{Ci}(x)P(C_i)}{f_{C0}(x)P(C0) + f_{C1}(x)P(C1)}$$

However, since the denominator is common, we can compare numerators directly.

```
def bayes_classifier(x, alpha, beta, mu, sigma, prior_c0, prior_c1):
    likelihood_c0 = gamma_pdf(x, alpha, beta)
    likelihood_c1 = gaussian_pdf(x, mu, sigma)
    posterior_c0 = likelihood_c0 * prior_c0
    posterior_c1 = likelihood_c1 * prior_c1
    predictions = np.where(posterior_c0 > posterior_c1, 0, 1)
    return predictions
```

Listing 11: Implementing the Bayes Classifier

Using Logarithms for Numerical Stability (Optional)

When dealing with very small probabilities, computing logarithms can improve numerical stability.

```
# Alternative implementation using logarithms (commented out)
def bayes_classifier_log(x, alpha, beta, mu, sigma, prior_c0, prior_c1):
    log_likelihood_c0 = np.log(gamma_pdf(x, alpha, beta) + 1e-300)
    log_likelihood_c1 = np.log(gaussian_pdf(x, mu, sigma) + 1e-300)
    log_posterior_c0 = log_likelihood_c0 + np.log(prior_c0)
    log_posterior_c1 = log_likelihood_c1 + np.log(prior_c1)
    predictions = np.where(log_posterior_c0 > log_posterior_c1, 0, 1)
    return predictions
```

Listing 12: Bayes Classifier with Logarithms

2.4 d) Making Predictions and Evaluating the Classifier

Making Predictions on the Test Set

```
# Step 7: Make Predictions
X_test_reshaped = X_test.reshape(-1, 1)
y_pred = bayes_classifier(X_test_reshaped.flatten(), alpha, beta_hat, mu_hat, sigma_hat, prior_c0
    , prior_c1)
```

Listing 13: Making Predictions

Evaluating the Classifier

```
# Step 8: Evaluate Classifier
accuracy = np.mean(y_pred == y_test)
print(f"\nTest Accuracy: {accuracy * 100:.2f}%")

from sklearn.metrics import confusion_matrix, classification_report

cm = confusion_matrix(y_test, y_pred)
print("\nConfusion Matrix:")
print(cm)

print("\nClassification Report:")
print(classification_report(y_test, y_pred))
```

Listing 14: Evaluating Classifier Performance

Results:

```
Test Accuracy: 91.81%

Confusion Matrix:
[[273  39]
 [ 20 388]]

Classification Report:
              precision    recall  f1-score   support

         0.0       0.93      0.88      0.90       312
         1.0       0.91      0.95      0.93       408

    accuracy                           0.92       720
   macro avg       0.92      0.91      0.92       720
weighted avg       0.92      0.92      0.92       720
```

2.5 Analyzing Misclassifications

Visualization of Misclassified Samples

```
# Step 9: Analyze Misclassifications
import seaborn as sns

misclassified_indices = np.where(y_pred != y_test)[0]
misclassified_samples = X_test[misclassified_indices]
correct_indices = np.where(y_pred == y_test)[0]
correct_samples = X_test[correct_indices]
correct_labels = y_test[correct_indices];

plt.figure(figsize=(12, 6))

sns.scatterplot(x=correct_samples, y=np.zeros_like(correct_samples), hue=correct_labels,
                palette={0: 'blue', 1: 'red'}, style=correct_labels, markers={0: 'o', 1: 's'},
                edgecolor='k', s=100, legend='full')

```

```
plt.scatter(misclassified_samples, np.zeros_like(misclassified_samples),
            facecolors='none', edgecolors='yellow', s=150, label='Misclassified')

plt.xlabel('Feature Value')
plt.yticks([])
plt.legend(title='Class', loc='upper right')
plt.title('Classification Results on Test Data')
plt.show()
```

Listing 15: Analyzing Misclassifications

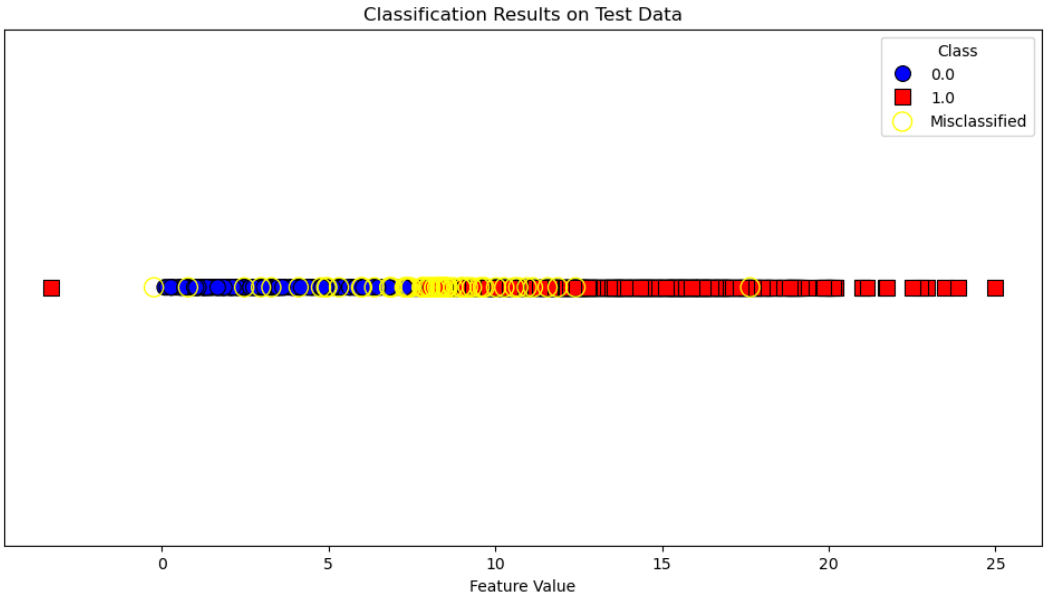

Figure 5: Visualization of correctly classified and misclassified samples on the test set.

Observations

From the confusion matrix and classification report, we observe:

- **Accuracy:** The classifier achieves an overall accuracy of **91.81%**.
- **Precision and Recall:**
  - **Class 0 (Gamma Distribution):**
    - Precision: 0.93
    - Recall: 0.88
    - F1-Score: 0.90
  - **Class 1 (Gaussian Distribution):**
    - Precision: 0.91
    - Recall: 0.95
    - F1-Score: 0.93

2.6 Discussion

Impact of Class Imbalance

The class priors reflect the imbalance in the training data:

$$P(C0) = 0.4444, \quad P(C1) = 0.5556$$

This imbalance can influence the classifier's predictions, potentially biasing it towards the majority class (Class 1).

Analysis of Misclassifications

Figure 5 shows that misclassifications primarily occur in regions where the feature distributions of the two classes overlap. Specifically:

- Some samples from **Class 0** with higher feature values are misclassified as **Class 1**.
- A few samples from **Class 1** with lower feature values are misclassified as **Class 0**.

Possible Improvements

To enhance the classifier's performance, especially for the minority class, we can consider:

Adjusting Class Priors

Assuming equal class priors could reduce bias towards the majority class:

$$P(C0) = 0.5, \quad P(C1) = 0.5$$

Applying Feature Transformation

Transforming the feature values (e.g., using logarithmic scaling) might improve class separability.

2.7 Visualizing Estimated PDFs

```
# Step 10: Additional Visualization
x_values = np.linspace(min(X), max(X), 1000)
gamma_pdf_values = gamma_pdf(x_values, alpha, beta_hat)
gaussian_pdf_values = gaussian_pdf(x_values, mu_hat, sigma_hat)

plt.figure(figsize=(12, 6))

sns.histplot(X_train_c0, bins=30, color='blue', alpha=0.5, stat='density', label='Class 0
    Histogram')
sns.histplot(X_train_c1, bins=30, color='red', alpha=0.5, stat='density', label='Class 1
    Histogram')

plt.plot(x_values, gamma_pdf_values, color='blue', label='Estimated Gamma PDF (Class 0)')
plt.plot(x_values, gaussian_pdf_values, color='red', label='Estimated Gaussian PDF (Class 1)')
% adding descision boundary
plt.axvline(x=(mu_hat+ beta_hat * alpha)/2, color='green', linestyle='--', label=f'Decision
    Boundary{(mu_hat+ beta_hat * alpha)/2:.2f}')

plt.xlabel('Feature Value')
plt.ylabel('Density')
plt.legend()
plt.title('Estimated PDFs and Histograms')
plt.show()
```

Listing 16: Plotting Estimated PDFs

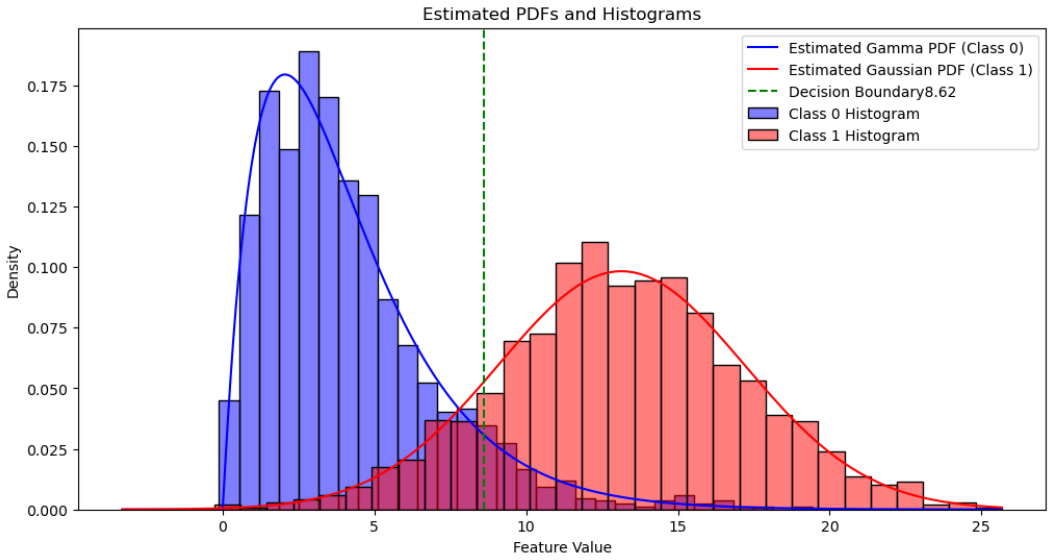

Figure 6: Estimated probability density functions overlaid on the histograms of the training data.

The plot in Figure 6 shows that the estimated PDFs fit the data distributions reasonably well. The overlap between the two distributions explains the misclassifications.

2.8 Conclusion

We developed a Bayes classifier to distinguish between two classes with different underlying distributions. By estimating the parameters using MLE, we implemented the classifier and achieved a high accuracy of **91.81%**. The analysis highlights the impact of class imbalance and overlapping distributions on classifier performance. Adjusting class priors and exploring other techniques can further improve the classifier.

All information were gathered from lecture notes and these sites and pdfs:
GeeksforGeeks 2024
Sign Function 2024
Statlect 2024
MIT 2024

References

Benjamin Ricaud, UiT (2024). *OptimizationAndGradientDescent*. Accessed: 2024-10-06. URL: `https://uit.instructure.com/courses/34698/files/3156825?module_item_id=1057845`.

GeeksforGeeks (2024). *Loss function for Linear regression in Machine Learning*. Accessed: 2024-07-29. URL: `https://www.geeksforgeeks.org/loss-function-for-linear-regression/#computing`.

MIT (2024). *Introduction to Gradient Descent*. Accessed: 2024-10-06. URL: `https://introml.mit.edu/_static/fall22/LectureNotes/chapter_Gradient_Descent.pdf`.

Sign Function, Wikipedia (2024). *sign function*. Accessed: 2024-10-06. URL: `https://en.wikipedia.org/wiki/Sign_function`.

Statlect (2024). *Maximum Likelihood Estimation*. Accessed: 2024-10-06. URL: `https://www.statlect.com/fundamentals-of-statistics/maximum-likelihood`.