EN3150 – Pattern Recognition

Assignment 01

Learning from data and related challenges and linear models for regression

Sandeepa H.W.P. - 220569X

B.Sc. Engineering

Semester 05

Submission Date – 21/08/2025

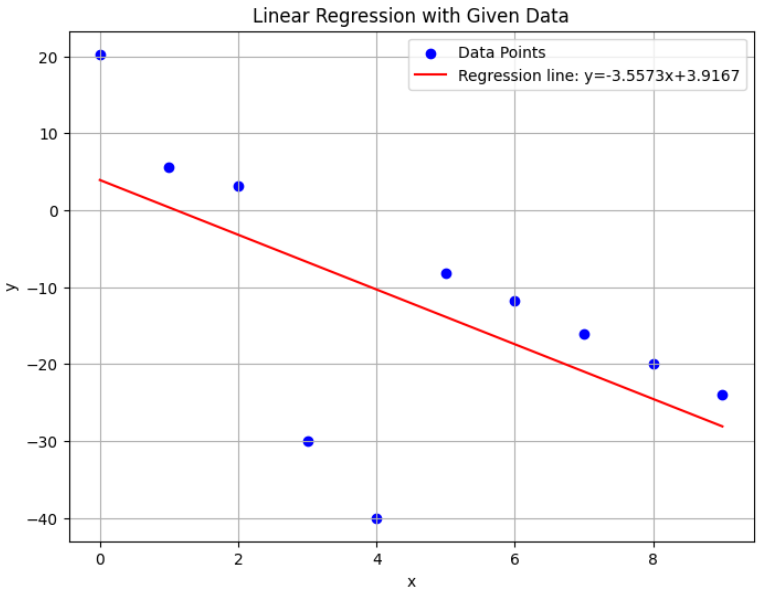
1) Linear regression impact on outliers.

1. Set of Data points

Table 1: Data set.

i	x_i	y_i
1	0	20.26
2	1	5.61
3	2	3.14
4	3	−30.00
5	4	−40.00
6	5	−8.13
7	6	−11.73
8	7	−16.08
9	8	−19.95
10	9	−24.03

2. Linear regression model finding and plotting



Linear Regression Equation: y = -3.5573x + 3.9167

Codes for calculating and plotting of Linear Regression Equation

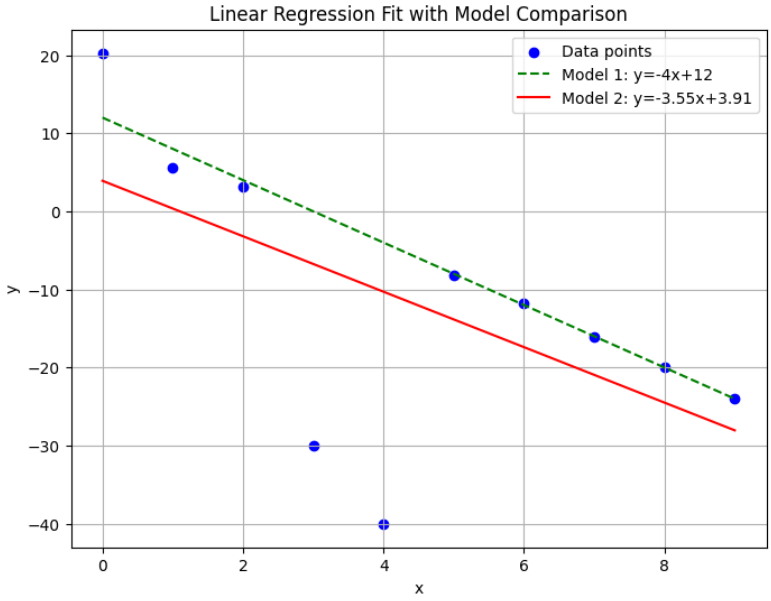
```
import numpy as np

import matplotlib.pyplot as plt

from sklearn.linear_model import LinearRegression

# Data from Table 1

x = np.array([0,1,2,3,4,5,6,7,8,9]).reshape(-1,1)

y = np.array([20.26, 5.61, 3.14, -30.00, -40.00, -8.13, -11.73, -16.08, -19.95, -
24.03])

# Fit linear regression model

model = LinearRegression()

model.fit(x, y)

# Predict values

y_pred = model.predict(x)

# Extract slope and intercept

slope = model.coef_[0]

intercept = model.intercept_

print(f"Linear Regression Equation: y = {slope:.4f}x + {intercept:.4f}")

# Plot data and regression line

plt.figure(figsize=(8,6))

plt.scatter(x, y, color="blue", label="Data Points")

plt.plot(x, y_pred, color="red", label=f"Regression line:
y={slope:.4f}x+{intercept:.4f}")

plt.xlabel("x")

plt.ylabel("y")

plt.title("Linear Regression with Given Data")

plt.legend()

plt.grid(True)

plt.show()
```

3.

- Model 1: $y = -4x + 12$
- Model 2: $y = -3.55x + 3.91$

$$L(\theta, \beta) = \frac{1}{N} \sum_{i=1}^{N} \left(\frac{(y_i - \hat{y}_i)^2}{(y_i - \hat{y}_i)^2 + \beta^2} \right).$$

4. Calculation of the loss function L(θ, β) Values.



<u>Loss values</u>

For β=1:

Model 1 loss=0.4354

Model 2 loss=0.9728

For $\beta=10^{-6}$:

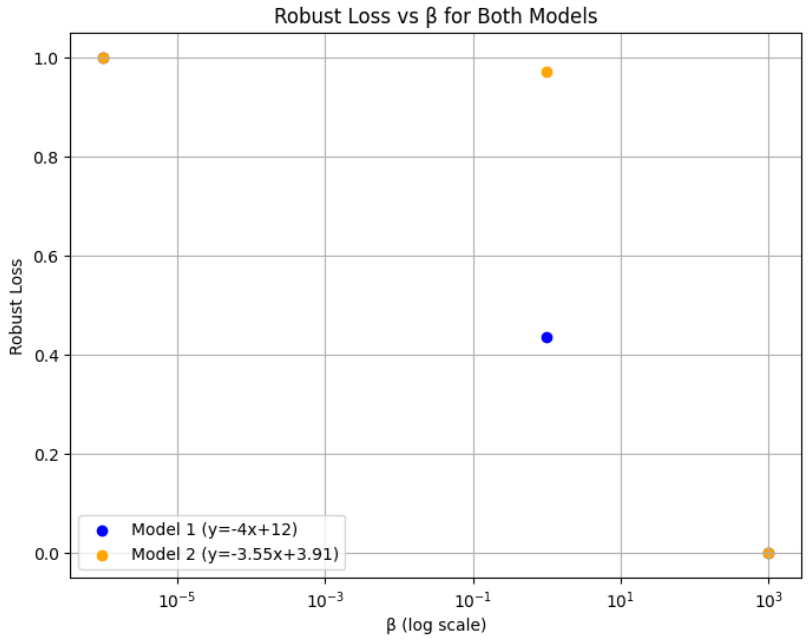
Model 1 loss=1.0000

Model 2 loss=1.0000

For $\beta=10^{3}$:

Model 1 loss=0.0002

Model 2 loss=0.0002



Code for calculation of the loss function L(θ,β) Values and Plotting.

```
import numpy as np

import matplotlib.pyplot as plt


# Data from Table 1

x = np.array([0,1,2,3,4,5,6,7,8,9])

y = np.array([20.26, 5.61, 3.14, -30.00, -40.00, -8.13, -11.73, -16.08, -19.95, -
24.03])


# Define the two models

def model1(x): return -4*x + 12

def model2(x): return -3.55*x + 3.91


y_pred1 = model1(x)

y_pred2 = model2(x)
```

```
# Figure 1.2: Regression Fit with Models
plt.figure(figsize=(8,6))
plt.scatter(x, y, color="blue", label="Data points")
plt.plot(x, y_pred1, "g--", label="Model 1: y=-4x+12")
plt.plot(x, y_pred2, "r-", label="Model 2: y=-3.55x+3.91")
plt.xlabel("x")
plt.ylabel("y")
plt.title("Linear Regression Fit with Model Comparison")
plt.legend()
plt.grid(True)
plt.show()

# Robust Loss Function
def robust_loss(y_true, y_pred, beta):
    errors = (y_true - y_pred)**2
    return np.mean(errors / (errors + beta**2))

betas = [1, 1e-6, 1e3]
losses_model1 = [robust_loss(y, y_pred1, b) for b in betas]
losses_model2 = [robust_loss(y, y_pred2, b) for b in betas]

print("Loss values:")
for b, l1, l2 in zip(betas, losses_model1, losses_model2):
    print(f"β={b}: Model 1 loss={l1:.4f}, Model 2 loss={l2:.4f}")

# Figure 1.3: Robust Loss vs β
plt.figure(figsize=(8,6))
plt.scatter(betas, losses_model1, color="blue", label="Model 1 (y=-4x+12)")
```

```
plt.scatter(betas, losses_model2, color="orange", label="Model 2 (y=-
3.55x+3.91)")

plt.xscale("log")

plt.xlabel("β (log scale)")

plt.ylabel("Robust Loss")

plt.title("Robust Loss vs β for Both Models")

plt.legend()

plt.grid(True, which="both")

plt.show()
```

5. Selecting Suitable β Value to Mitigate the Impact of Outliers

The most suitable value of β is 1.

Case 1: $\beta = 10^{-6}$

- When the value of β is extremely small, the denominator $(e^2+\beta^2)$ is almost the same as e^2 for any nonzero error. This makes each term in the loss function simplify to roughly $\frac{e^2}{e^2} \approx 1$.. In practice, this means that every data point contributes almost equally to the loss, regardless of whether it is a normal point or an outlier. As a result, the outliers are not reduced in influence, and in our calculations, both models ended up with a loss close to 1.0, showing no meaningful distinction between them.

Case 2: $\beta = 10^3$

- When β is very large, the denominator $(e^2+\beta^2)$ is dominated by β^2, so each fraction becomes extremely small, roughly $\frac{e^2}{\beta^2} \approx 0$. This has the effect of almost completely ignoring the errors, since their contribution to the loss is negligible. In this situation, the loss values for both models drop close to zero, and because they are nearly identical, it becomes impossible to distinguish between the models in any meaningful way.

Case 3: $\beta = 1$

- With β set to 1, the loss function strikes a good balance between the numerator and the denominator. This means that large residuals (caused by outliers) are naturally down-weighted, while smaller residuals from normal points continue to contribute meaningfully. In our results, Model 1 produced a loss of 0.4354, while Model 2 gave a loss of 0.9728. This clear difference shows that the robust estimator is able to separate the

two models effectively, while at the same time reducing the overwhelming influence of outliers on the overall loss.

6. Determining the most suitable model from the models

From the analysis, the most appropriate value of β to reduce the influence of outliers is β = 1. At this setting, the robust loss function is able to lessen the impact of extreme errors while still giving proper weight to the normal data points. This balance makes β = 1 the best choice for handling outliers without losing important information from the majority of the dataset.

Robust Loss Values (β = 1)

- Model 1 (y = –4x + 12): Loss = 0.4354
- Model 2 (y = –3.55x + 3.91): Loss = 0.9728

Comparison of Models

- The lower the robust loss, the better the model fits the dataset under the robust estimator.
- With β = 1, Model 1 achieves a significantly lower loss than Model 2.
- This indicates that Model 1 explains the main data trend better while being less affected by outliers.

Using the robust estimator with β = 1, the most suitable model for the given dataset is Model 1 (y = –4x + 12).

7. How does the robust estimator reduce the impact of the outliers

Standard Loss (MSE) and Its Limitation

- In Mean Squared Error (MSE), the loss for each data point is:

$$\text{MSE} = \frac{1}{N}\sum(y_i - \hat{y}_i)^2$$

- Because the error is squared, data points with very large errors (outliers) dominate the loss function.
- This can cause the regression line to shift significantly away from the majority of the data just to minimize the influence of a few extreme points.

Robust Estimator Loss

The robust estimator modifies the error contribution as:

$$L(\theta, \beta) = \frac{1}{N} \sum_{i=1}^{N} \frac{(y_i - \hat{y}_i)^2}{(y_i - \hat{y}_i)^2 + \beta^2}$$

- When the squared error $(y_i - \hat{y}_i)^2$ is small (normal points):
  $\frac{e^2}{e^2+\beta^2}$ is close to $\frac{e^2}{\beta^2}$, meaning normal data points contribute significantly to the loss.
- When the squared error $(y_i - \hat{y}_i)^2$ is very large (outliers):
  The denominator grows much faster than the numerator, so

$$\frac{e^2}{e^2 + \beta^2} \to 1$$

Unlike in MSE, where outliers can cause the loss to explode, the robust estimator caps their contribution so they cannot dominate the optimization process

Effect on Model Training

- This "capping" ensures that outliers only have limited influence on the final regression line.
- This means the estimator is able to focus more on capturing the overall trend of the majority of the data, rather than being distorted by just a few extreme values.

In summary, the robust estimator lessens the impact of outliers by down-weighting large errors and ensuring they do not dominate the loss function. This makes the regression model better reflect the true pattern of the overall dataset, instead of being skewed by a few extreme points.

.

8. Another Loss Function for the Robust Estimator

Alternative: Huber Loss

One widely used alternative to the given robust loss is the Huber loss function, defined as:

$$L_\delta(e) = \begin{cases} \frac{1}{2}e^2 & \text{for } |e| \le \delta \\ \delta \cdot (|e| - \frac{1}{2}\delta) & \text{for } |e| > \delta \end{cases}$$

where e=$(y_i - \hat{y}_i)$ is the error and δ is a tuning parameter.

Why Huber Loss?

- For small errors ($|e| \leq \delta$), the Huber loss acts just like MSE, providing smooth gradients and encouraging the model to fit the normal data points as accurately as possible.
- For large errors ($|e| > \delta$), the Huber loss behaves like the Mean Absolute Error (MAE), increasing linearly rather than quadratically. This reduces the impact of outliers by preventing their errors from dominating the training process.

Benefits

- Combines the advantages of both MSE (sensitivity to small errors) and MAE (robustness to outliers).
- Commonly used in robust regression and is available in machine learning libraries such as Scikit-learn and TensorFlow.

The Huber loss is a great alternative when we want a regression model that can handle outliers without losing accuracy on the rest of the data. It softens the effect of extreme points while still paying close attention to the normal data, making it a very practical choice for building models that are both reliable and robust.

2) Loss Function

- **Application 1:** The dependent variable is continuous.
- **Application 2:** The dependent variable is discrete and binary (only takes values 0 or 1 i.e., $y \in \{0, 1\}$).

- **Mean Squared Error (MSE):**

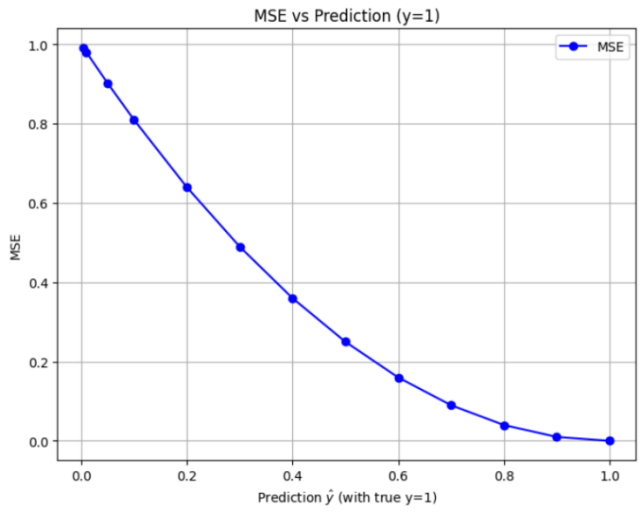
$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

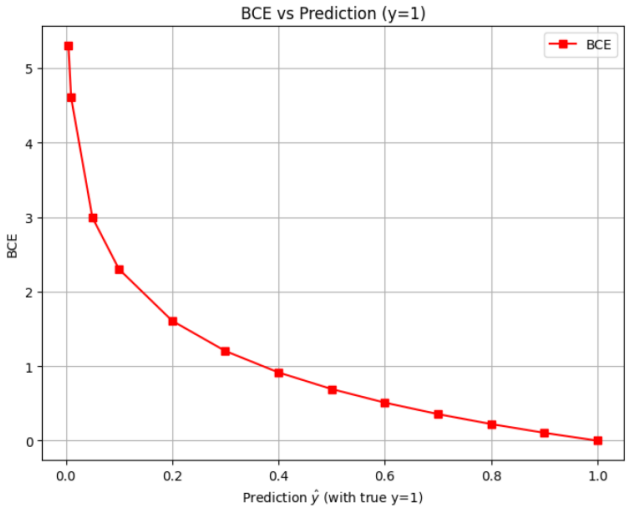
- **Binary Cross Entropy (BCE):**

$$\text{BCE} = -\frac{1}{n}\sum_{i=1}^{n}[y_i \log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i)]$$

Filled Table and Plotting Both Loss Functions

True y	Prediction $\hat{y}$	MSE (correct)	BCE
1	0.005	0.990025	5.298317
1	0.01	0.9801	4.60517
1	0.05	0.9025	2.995732
1	0.1	0.81	2.302585
1	0.2	0.64	1.609438
1	0.3	0.49	1.203973
1	0.4	0.36	0.916291
1	0.5	0.25	0.693147
1	0.6	0.16	0.510826
1	0.7	0.09	0.356675
1	0.8	0.04	0.223144
1	0.9	0.01	0.105361
1	1	0	0





Code for Calculation of MSC and BCE values and plotting

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

# Data setup
y_true = 1.0
yhat = np.array([0.005, 0.01, 0.05, 0.1, 0.2, 0.3, 0.4, 0.5,
                 0.6, 0.7, 0.8, 0.9, 1.0])

# Loss functions
def mse(y, y_pred):
    return (y - y_pred)**2   # Correct MSE (no 1/2 factor)

def bce(y, y_pred):
    eps = 1e-12  # to avoid log(0)
    y_pred = np.clip(y_pred, eps, 1 - eps)
    return -(y * np.log(y_pred) + (1 - y) * np.log(1 - y_pred))

mse_vals = mse(y_true, yhat)
bce_vals = bce(y_true, yhat)

# Table of values
df = pd.DataFrame({
    "True y": [1]*len(yhat),
    "Prediction y_hat": yhat,
    "MSE": mse_vals,
    "BCE": bce_vals
```

```
}).round(6)
print(df)

# Plot 1: MSE vs Prediction
plt.figure(figsize=(8,6))
plt.plot(yhat, mse_vals, marker='o', color="blue", label="MSE")
plt.xlabel("Prediction $\\hat{y}$ (with true y=1)")
plt.ylabel("MSE")
plt.title("MSE vs Prediction (y=1)")
plt.grid(True)
plt.legend()
plt.show()

# Plot 2: BCE vs Prediction
plt.figure(figsize=(8,6))
plt.plot(yhat, bce_vals, marker='s', color="red", label="BCE")
plt.xlabel("Prediction $\\hat{y}$ (with true y=1)")
plt.ylabel("BCE")
plt.title("BCE vs Prediction (y=1)")
plt.grid(True)
plt.legend()
plt.show()
```

Selection of Appropriate Loss Functions

Application 1: Continuous Dependent Variable

- In this application, the dependent variable yyy is continuous. Examples include predicting stock prices, temperature, or energy consumption, where the output can take on any real value.
- Suitable Loss Function: Mean Squared Error (MSE)

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

- MSE measures the average squared difference between predicted and actual values.
- It is especially sensitive to large errors, which means the model gives extra attention to minimizing big deviations and ensures that the predictions stay close to the true values.
- MSE is the most commonly used and mathematically convenient loss function for regression problems because it simplifies the optimization process, especially when working with linear regression.

Application 2: Binary Dependent Variable

- Nature of the problem: The dependent variable $y \in \{0,1\}$ is binary.
- Suitable Loss Function: Binary Cross Entropy (BCE**)**

$$BCE = -\frac{1}{n}\sum_{i=1}^{n}\left[y_i \log(\hat{y}_i) + (1-y_i)\log(1-\hat{y}_i)\right]$$

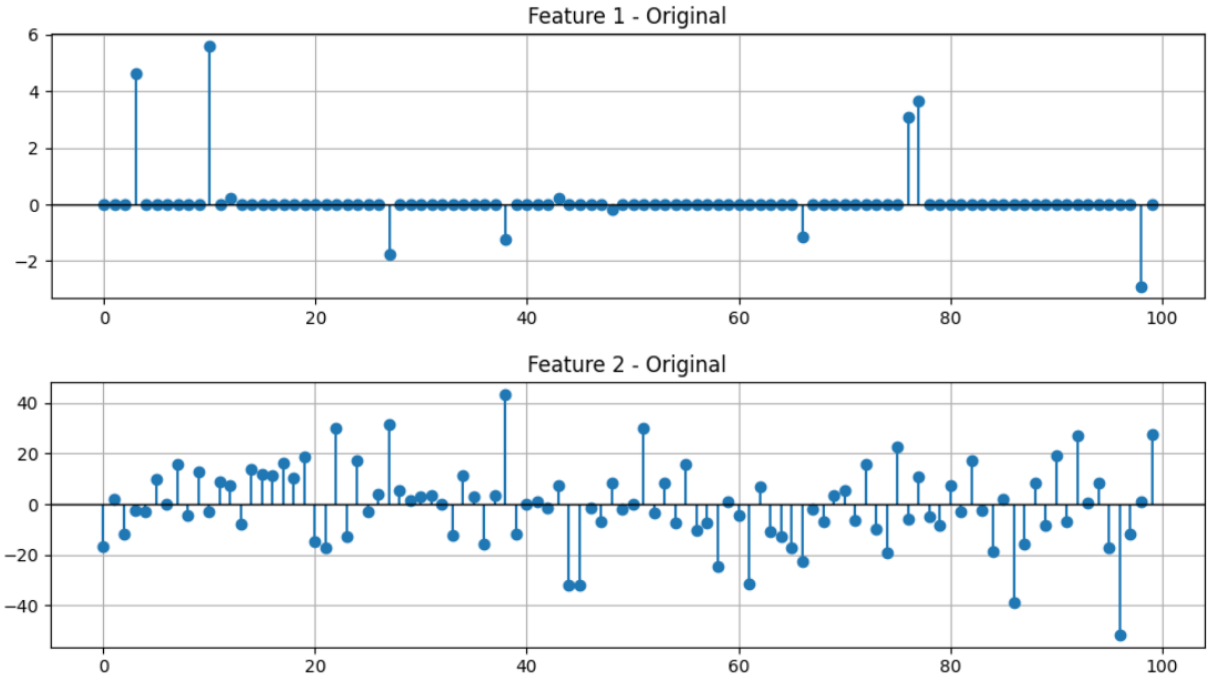
- BCE is specifically designed for binary classification tasks where the model outputs probabilities, making it well-suited for distinguishing between two possible outcomes.
- It strongly penalizes cases where the model is very confident but wrong, which pushes the predictions to stay closer to the actual class labels.
- It also fits naturally with logistic regression, since the outputs are interpreted as probabilities, making BCE a perfect match for this type of model.

Therefore,

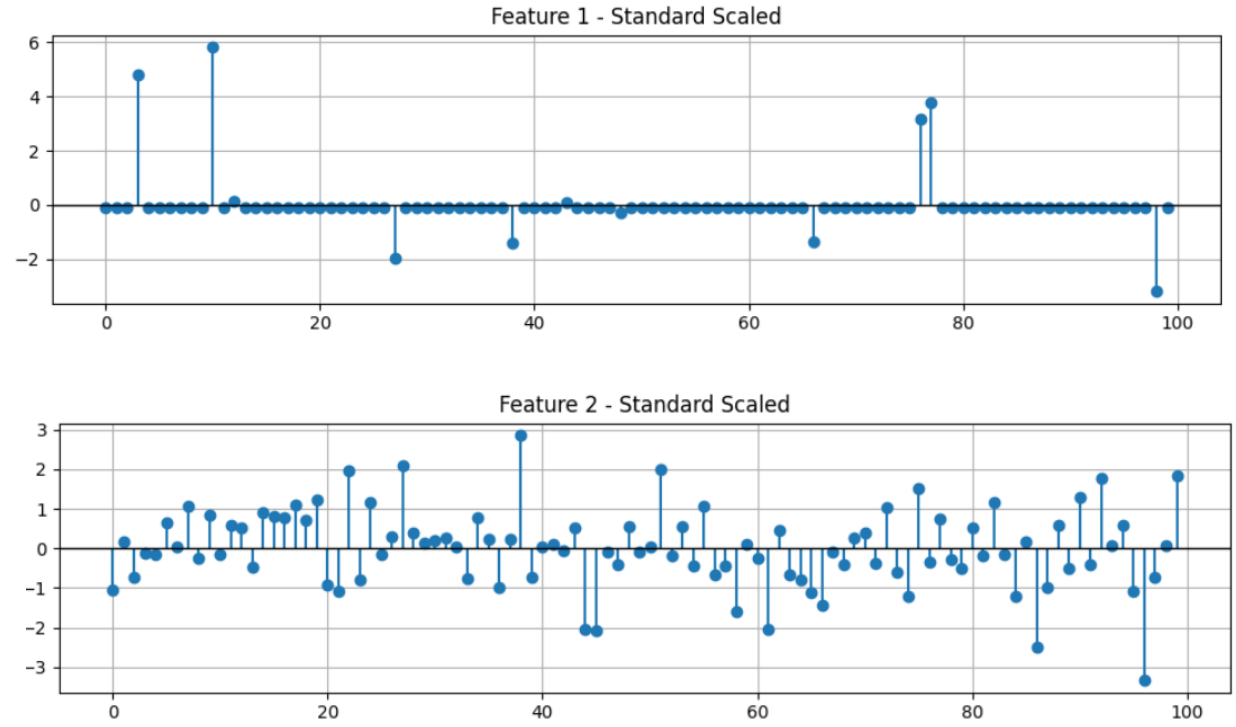
- Application 1 (continuous y): Use MSE, best suited for regression problems.
- Application 2 (binary y): Use BCE, best suited for binary classification problems.

3. Data pre-processing

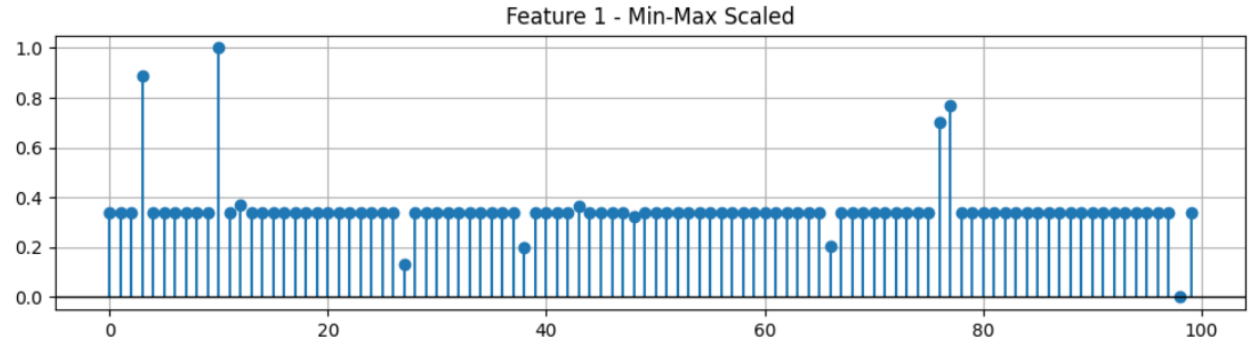
Generating the feature values of the two features

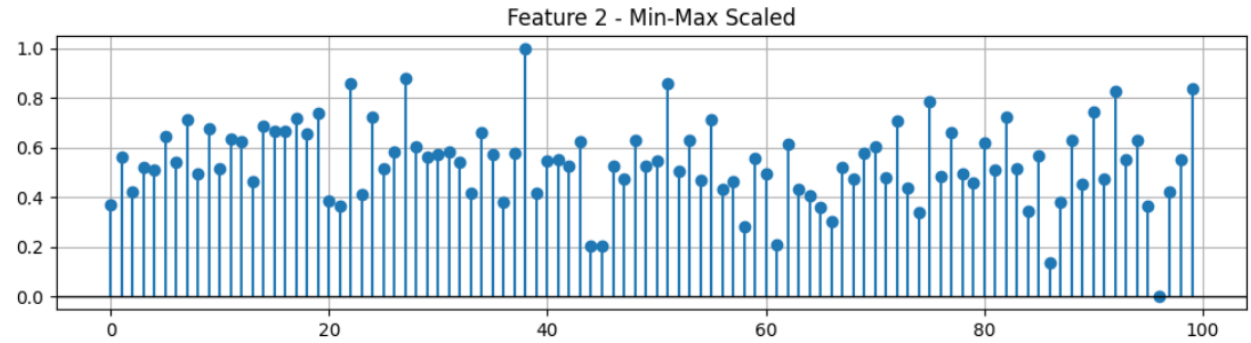


Considering Standard scaling method for feature 1 & 2

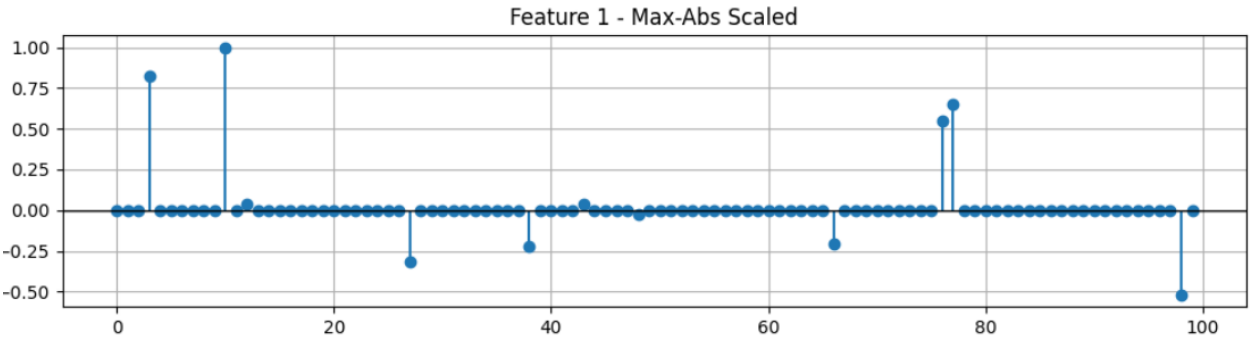


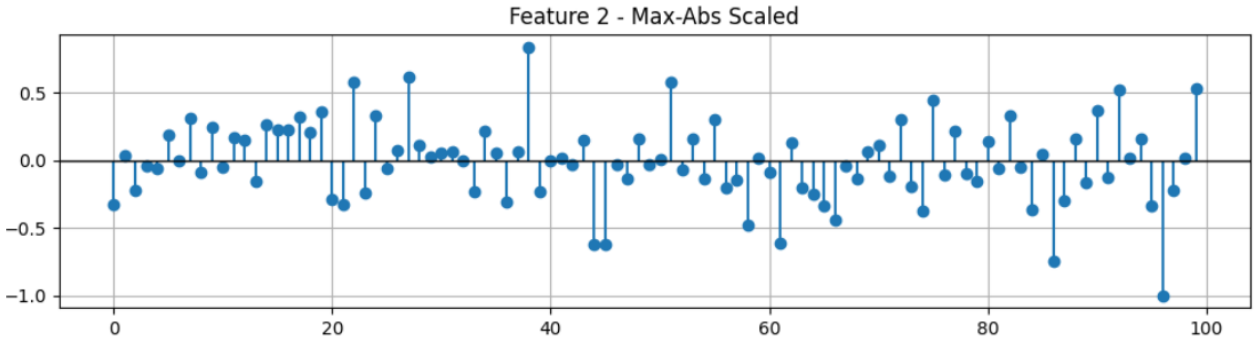
Considering Min-Max scaling method for feature 1 & 2





Considering Max-Abs scaling method for feature 1 & 2





Code for using scaling methos for features and plotting

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.preprocessing import StandardScaler, MinMaxScaler, MaxAbsScaler

# Function to generate sparse signal
def generate_signal(signal_length, num_nonzero):
    signal = np.zeros(signal_length)
    nonzero_indices = np.random.choice(signal_length, num_nonzero, replace=False)
    nonzero_values = 10 * np.random.randn(num_nonzero)
    signal[nonzero_indices] = nonzero_values
    return signal

# Parameters
signal_length = 100
num_nonzero = 10
your_index_no = 220569  # index without English letter

# Feature 1: Sparse signal
sparse_signal = generate_signal(signal_length, num_nonzero)
sparse_signal[10] = (your_index_no % 10) * 2 + 10
if your_index_no % 10 == 0:
    sparse_signal[10] = np.random.randn(1) + 30
sparse_signal = sparse_signal / 5

# Feature 2: Gaussian noise
epsilon = np.random.normal(0, 15, signal_length)
```

```
# Scaling methods to apply
scalers = {
    "Original": lambda x: x,
      "Standard Scaled": lambda x: StandardScaler().fit_transform(x.reshape(-1,1)).flatten(),
       "Min-Max Scaled": lambda x: MinMaxScaler().fit_transform(x.reshape(-1,1)).flatten(),
       "Max-Abs Scaled": lambda x: MaxAbsScaler().fit_transform(x.reshape(-1,1)).flatten()
}

# Plot results
plt.figure(figsize=(10,22))

plot_idx = 1
for method, func in scalers.items():
    # Feature 1
    plt.subplot(8,1,plot_idx)
    plt.stem(func(sparse_signal), basefmt=" ")
    plt.axhline(0, color='black', linewidth=1)
    plt.title(f"Feature 1 - {method}")
    plt.grid(True)
    plot_idx += 1

    # Feature 2
    plt.subplot(8,1,plot_idx)
    plt.stem(func(epsilon), basefmt=" ")
    plt.axhline(0, color='black', linewidth=1)
    plt.title(f"Feature 2 - {method}")
    plt.grid(True)
```

```
    plot_idx += 1


plt.tight_layout()
plt.show()
```

Features from the Code

1. **Feature 1 (Sparse Signal)**

   Feature 1 is mostly made up of zeros, with just a handful of large non-zero spikes. This gives it a sparse structure, where the data is dominated by zero values but also contains a few significant outliers.

2. **Feature 2 (Noise / Epsilon)**

   Generated from a normal distribution N $(0,15^2)$. Continuous values centered around 0, with a spread defined by variance.

Scaling Methods

1. **Standard Scaling (Z-score Normalization):**

$$x' = \frac{x - \mu}{\sigma}$$

   - Centers data to mean = 0, standard deviation = 1.

2. **Min-Max Scaling:**

$$x' = \frac{x - x_{\min}}{x_{\max} - x_{\min}}$$

   - Scales values into the range [0,1].
   - Useful when features must be bounded.

3. **Max-Abs Scaling:**

$$x' = \frac{x}{\max(|x|)}$$

   - Scales values to the range [-1,1].
   - Zero values remain zero
   - Best for sparse data.

Suitable Choices

- **Feature 1:**
  For this feature, the best choice is Max-Abs Scaling. This method keeps all the zero values exactly as they are (so the sparse structure is preserved) while scaling the non-zero spikes into a comparable range [-1,1].
- **Feature 2:**
  For this feature, the best choice is Standard Scaling. Since the data already follows a Gaussian (bell-curve) distribution, standardization simply shifts it to have a mean of 0 and a standard deviation of 1. This makes the noise easier to work with while keeping its natural distribution shape intact.

References

[1] Scikit-learn developers, "Preprocessing data — scikit-learn 1.5.1 documentation," *scikit-learn.org*. Available: https://scikit-learn.org/stable/modules/preprocessing.html. [Accessed: Aug. 21, 2025].

[2] "StandardScaler, MinMaxScaler and RobustScaler techniques | ML," *GeeksforGeeks*. Available: https://www.geeksforgeeks.org/machine-learning/standardscaler-minmaxscaler-and-robustscaler-techniques-ml/. [Accessed: Aug. 21, 2025].

[3] "Regression tutorial," *StatTrek*. Available: https://stattrek.com/tutorials/regression-tutorial. [Accessed: Aug. 21, 2025].

Appendix

Assignment 01 Github Repo Link : Link