Department of Electrical Engineering

University of Moratuwa

EN 3150 – Pattern Recognition

Learning from data and related challenges and linear models for regression

Kularathna A. K. D. D. D. – 220332P

This is submitted as a partial fulfilment for the module
EN 3150 Pattern Recognition
Department of Electronics and Telecommunication Engineering
University of Moratuwa

1. Linear regression impact on outliers

1.

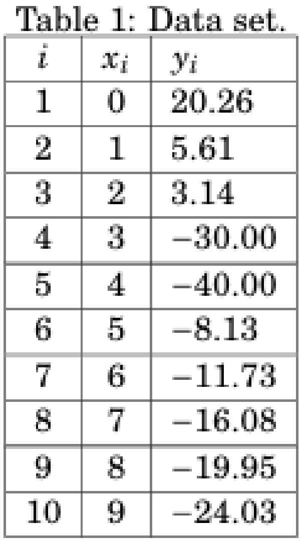


Table 1: Data set.

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. Finding the Linear regression model and Plotting x, y as scatter and the linear regression model.

- Regression Line Equation: - $\boldsymbol{y = 3.917 + -3.557x}$

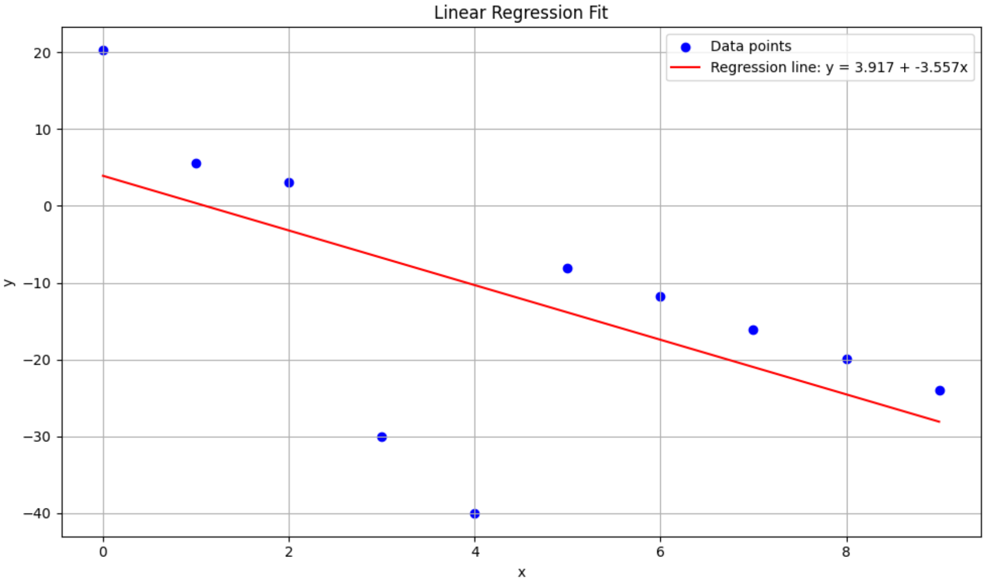


Figure 1.1

Code for Getting regression line and Plotting

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

# Data points (Table 1: Data Set)
x_i = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
y_i = [20.26, 5.61, 3.14, -30.00, -40.00, -8.13, -11.73, -16.08, -19.95, -
24.03]

# Create design matrix X (with a column of 1s for intercept) and target
vector Y
X = np.array([[1, x] for x in x_i])
Y = np.array([[y] for y in y_i])

# print("X =", X)
# print("Y =", Y)

# Apply OLS formula: w = (X^T X)^(-1) X^T Y
X_T = X.T
w_OLS = np.linalg.inv(X_T @ X) @ X_T @ Y
# Extract coefficients
intercept = w_OLS[0, 0]
slope = w_OLS[1, 0]

# Predict y values using the regression model
x_range = np.linspace(min(x_i), max(x_i), 100)
y_predicted = intercept + slope * x_range

# Step 4: Plot original data and regression line
plt.figure(figsize=(10, 6))
plt.scatter(x_i, y_i, color='blue', label='Data points')
plt.plot(x_range, y_predicted, color='red', label=f'Regression line: y =
{intercept:.3f} + {slope:.3f}x')
plt.xlabel('x')
plt.ylabel('y')
plt.title('Linear Regression Fit')
plt.legend()
plt.grid(True)
plt.tight_layout()

plt.show()

print(f"Regression Line Equation: y = {intercept:.3f} + {slope:.3f}x")
```

3.

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

4. Calculation Loss Function Values for given β values

$$\boldsymbol{Model\ 1\ \rightarrow y =\ -4x + 12}$$

$$\boldsymbol{Model\ 2\ \rightarrow\ -3.55x + 3.91}$$

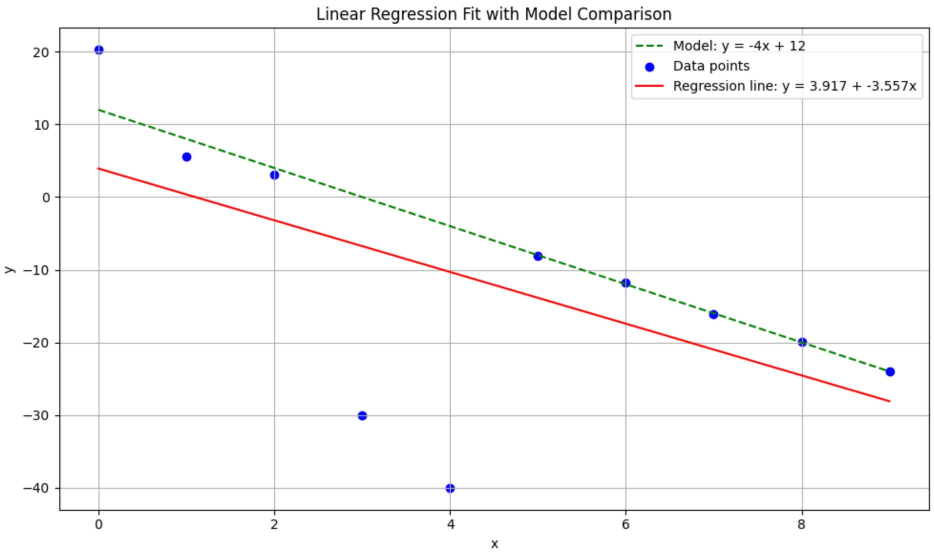


Figure 1.2

- **β = 1**

  Model 1 loss = 0.4354

  Model 2 loss = 0.9728

- **β = 10^{-6}**

  Model 1 loss = 1.0000

  Model 2 loss = 1.0000

- **β = 10^{3}**

  Model 1 loss = 0.0002

  Model 2 loss = 0.0002

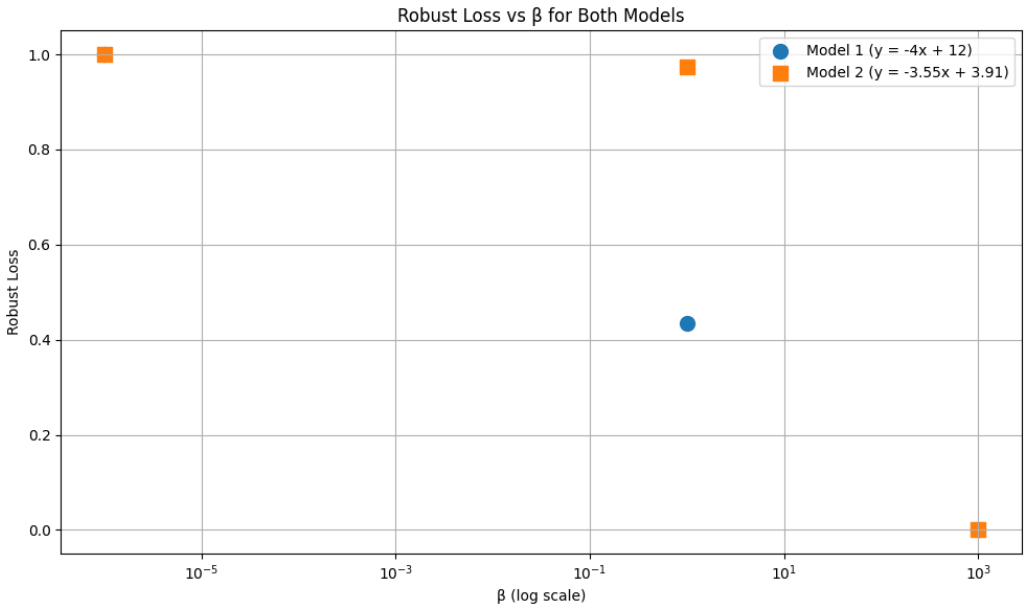


Figure 1.3

Code for Calculation of β values and plotting

```
# Compare with the model y = -4x + 12
y_model = lambda x: -4 * x + 12

# Plot the model line
x_model = np.linspace(min(x_i), max(x_i), 100)
y_model_values = y_model(x_model)
# Plot the model line
plt.figure(figsize=(10, 6))
plt.plot(x_model, y_model_values, color='green', label='Model: y = -4x + 12', linestyle='--')
# Add the model line to the existing plot
plt.scatter(x_i, y_i, color='blue', label='Data points')
plt.plot(x_range, y_predicted, color='red', label=f'Regression line: y = {intercept:.3f} + {slope:.3f}x')
plt.xlabel('x')
plt.ylabel('y')
plt.title('Linear Regression Fit with Model Comparison')
plt.legend()
plt.grid(True)
plt.tight_layout()
```

```
def robust_loss(y_true, y_pred, beta):
    N = len(y_true)
    squared_errors = (y_true - y_pred) ** 2
    loss = np.mean(squared_errors / (squared_errors + beta**2))
    return loss

# Generate predictions for both models
y_model1 = -4 * np.array(x_i) + 12  # Model 1
y_model2 = -3.55 * np.array(x_i) + 3.91  # Model 2

# Test different beta values
beta_values = [1, 10**(-6), 10**(3)]
results = []

for beta in beta_values:
    loss_model1 = robust_loss(y_i, y_model1, beta)
    loss_model2 = robust_loss(y_i, y_model2, beta)
    results.append({
        'beta': beta,
        'loss_model1': loss_model1,
        'loss_model2': loss_model2
    })

# Create a table of results
for result in results:
    print(f"β = {result['beta']:.6f}:")
    print(f"  Model 1 (y = -4x + 12) loss: {result['loss_model1']:.4f}")
    print(f"  Model 2 (y = -3.55x + 3.91) loss:
{result['loss_model2']:.4f}")
    print()

# Visualize how the loss changes with beta
plt.figure(figsize=(10, 6))
plt.scatter(beta_values, [r['loss_model1'] for r in results], marker='o',
s=100, label='Model 1 (y = -4x + 12)')
plt.scatter(beta_values, [r['loss_model2'] for r in results], marker='s',
s=100, label='Model 2 (y = -3.55x + 3.91)')
plt.xlabel('β (log scale)')
plt.ylabel('Robust Loss')
plt.title('Robust Loss vs β for Both Models')
plt.legend()
plt.grid(True)
plt.xscale('log')  # Ensure x-axis is in log scale
plt.tight_layout()
```

5. Suitable β value to mitigate the impact of the outliers

- When $\beta = 10^{-6}$
  - In this case for both models the loss function value becomes really close to 1 since the denominator becomes nearly equal (since β^2 term is nearly zero for small values). This makes the loss function **more sensitive to the outliers** in the dataset.
- When β = 1
  - In this case for both models the loss function gives balanced values. This time loss function is little less sensitive to the outliers but not as sensitive as earlier. This allows us to make the more **Robust to outliers** while being sensitive enough to differentiate between the performance of different models on the given data.
- When $\beta = 10^3$
  - In this case for both models the loss function value becomes really close to 0 since the denominator becomes nearly equal to β^2. This makes the model **less sensitive to the outliers** in the data set. The low loss values make it **harder to differentiate between the models**, as they both appear to perform well.

Due to these reasons β **= 1 is the best value to choose from**

6. Determining most suitable model from the models

- Using β = 1 Model 1 (y = -4x + 12) is the better choice because the loss = 0.4354 and the loss = 0.9728 for Model 2 (y = -3.55x + 3.91)
  - The robust loss at β=1 directly measures how well a model explains the data while reducing the effect of outliers. A lower robust loss indicates a better fit under this robust criterion.
  - Numerically Model 1's loss is substantially lower than Model 2's at β = 1. Therefore Model 1 better explains the bulk of the data while being less influenced by extreme points, which is exactly the goal when using a robust estimator.

Therefore, Choose Model 1

$$\boldsymbol{y = -4x + 12}$$

7. How Robust estimator reduce the impact of the outliers

The loss function is:

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

- **When the error $(y_i - \hat{y}_i)^2$ is small compared to β².**

  The fraction is small, so these points contribute only a little to the loss – like normal squared error.

- **When the error $(y_i - \hat{y}_i)^2$ is large (from an outlier).**

  The denominator grows almost as much as the numerator, and the fraction approaches 1. This means no matter how huge the error is, its contribution is capped at 1.

By doing this, the loss function prevents large errors from dominating the total loss, making the model less affected by outliers. The value of β controls when this “capping” effect starts.

8. Another loss function that can be used for this robust estimator

A common alternative is Huber **loss**. It is defined (for residual $\boldsymbol{r = y_i - \hat{y}_i}$) as:

$$l_\delta(r) = \begin{cases} \frac{1}{2}r^2 \leq \delta & if\ |r| \leq \delta \\ \delta\left(|r| - \frac{1}{2}\delta\right) & if\ |r| > \delta \end{cases}$$

- Huber is **quadratic for small residuals** (like MSE) and **linear for large residuals** (like absolute error), so it reduces the influence of large errors while staying differentiable and convex.
- The parameter δ plays a role like β: it sets the threshold between “squared” and “linear” regimes, enabling robustness tuning

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):

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

1. Filling the loss values and plotting functions

True $y = 1$	Prediction $\hat{y}$	MSE	BCE
1	0.005	0.990025	5.298317
1	0.01	0.980100	4.605170
1	0.05	0.902500	2.995732
1	0.1	0.810000	2.302585
1	0.2	0.640000	1.609438
1	0.3	0.490000	1.203973

1	0.4	0.360000	0.916291
1	0.5	0.250000	0.693147
1	0.6	0.160000	0.510826
1	0.7	0.090000	0.356675
1	0.8	0.040000	0.223144
1	0.9	0.010000	0.105361
1	1.0	0.000000	0.000000

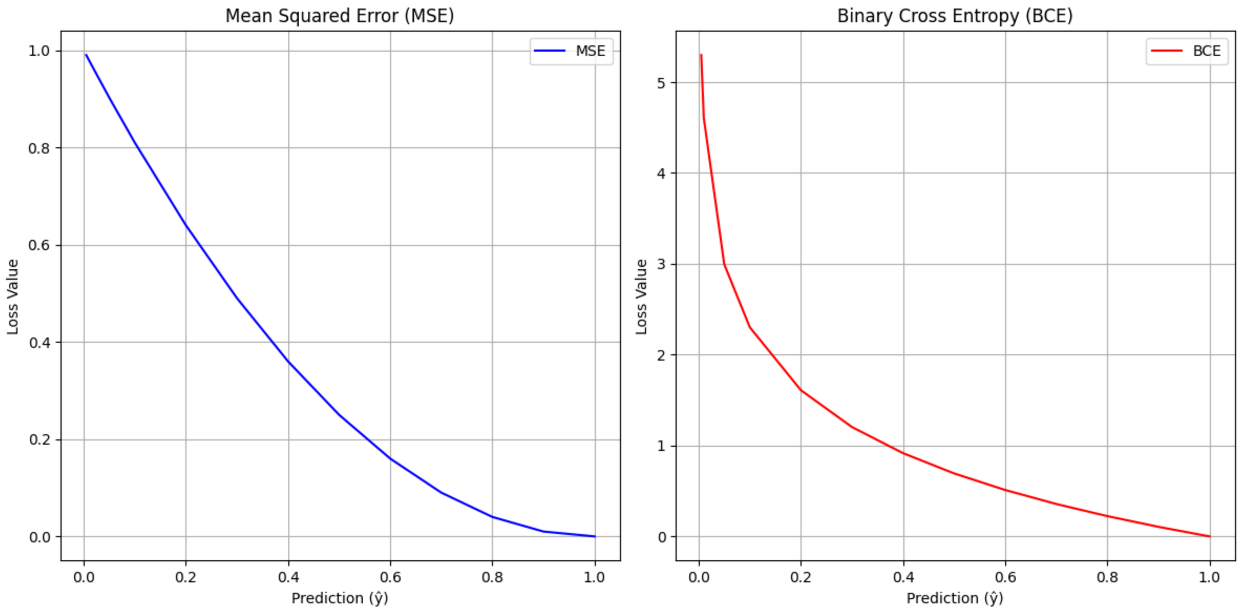


Figure 2. 1

Code for Calculation of MSC and BCE values and plotting

```
import math
import matplotlib.pyplot as plt

# Define the true labels
y_true = [1] * 11  # 11 samples, all with true value

# Define the predictions
y_pred = [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]

# Function to calculate Mean Squared Error (MSE)
def calculate_mse(y_true, y_pred):
    return (1 - y_pred) ** 2

# Function to calculate Binary Cross Entropy (BCE)
def calculate_bce(y_true, y_pred):
    epsilon = 1e-15  # Small value to prevent log(0)
    y_pred = max(min(y_pred, 1 - epsilon), epsilon)  # Clip prediction
    return -math.log(y_pred)

mse_values = []
bce_values = []

for pred in y_pred:
    mse_values.append(calculate_mse(y_true, pred))
    bce_values.append(calculate_bce(y_true, pred))

# Create a table of results
print("Table 2: MSE and BCE loss values for different predictions when y = 1")
print("True y=1 | Prediction ŷ |    MSE    |    BCE")
print("-" * 45)
for pred, mse, bce in zip(y_pred, mse_values, bce_values):
    print(f"    1    |    {pred:.3f}     | {mse:.6f} | {bce:.6f}")

# Plotting both loss functions
plt.figure(figsize=(12, 6))

# Plot MSE
plt.subplot(1, 2, 1)
plt.plot(y_pred, mse_values, 'b-', label='MSE')
plt.xlabel('Prediction (ŷ)')
plt.ylabel('Loss Value')
plt.title('Mean Squared Error (MSE)')
plt.grid(True)
plt.legend()

# Plot BCE
plt.subplot(1, 2, 2)
plt.plot(y_pred, bce_values, 'r-', label='BCE')
plt.xlabel('Prediction (ŷ)')
plt.ylabel('Loss Value')
plt.title('Binary Cross Entropy (BCE)')
plt.grid(True)
plt.legend()

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

2. Chosen Loss functions for each Application

- ***Application 1: - Linear Regression with Continuous Dependent Variable***
- **Selection: - Mean Square Error (MSE)**

Reasons for the choice

1. **MSE is ideal for continuous variables because:**
    - It directly measures the squared difference between predicted and actual values
    - It penalizes larger errors more heavily
    - It can handle both positive and negative prediction errors
    - The gradient of MSE is proportional to the error, making optimization straightforward.
2. **BCE is not suitable because:**
    - BCE needs probability between 0 and 1
    - It’s designed for binary classification, not continuous prediction
    - Using BCE for continuous values leads to undefined behavior

- ***Application 2: - Logistic Regression with Binary Dependent variable***
- **Selection: - Binary Cross Entropy (BCE)**

Reasons for the choice

1. **BCE is ideal for binary classification because:**
    - It's specifically designed for probabilities in range [0,1]
    - It measures the difference between probability distributions
    - It provides stronger gradients when predictions are confident but wrong
    - It naturally fits with the logistic function's output range
2. **MSE is not suitable because:**
    - It does not account for the probabilistic nature of binary classification
    - It can lead to slower convergence in binary problems
    - The gradients can become very small when predictions are far from targets.

3. Data pre-processing

1. Generating feature values of two features

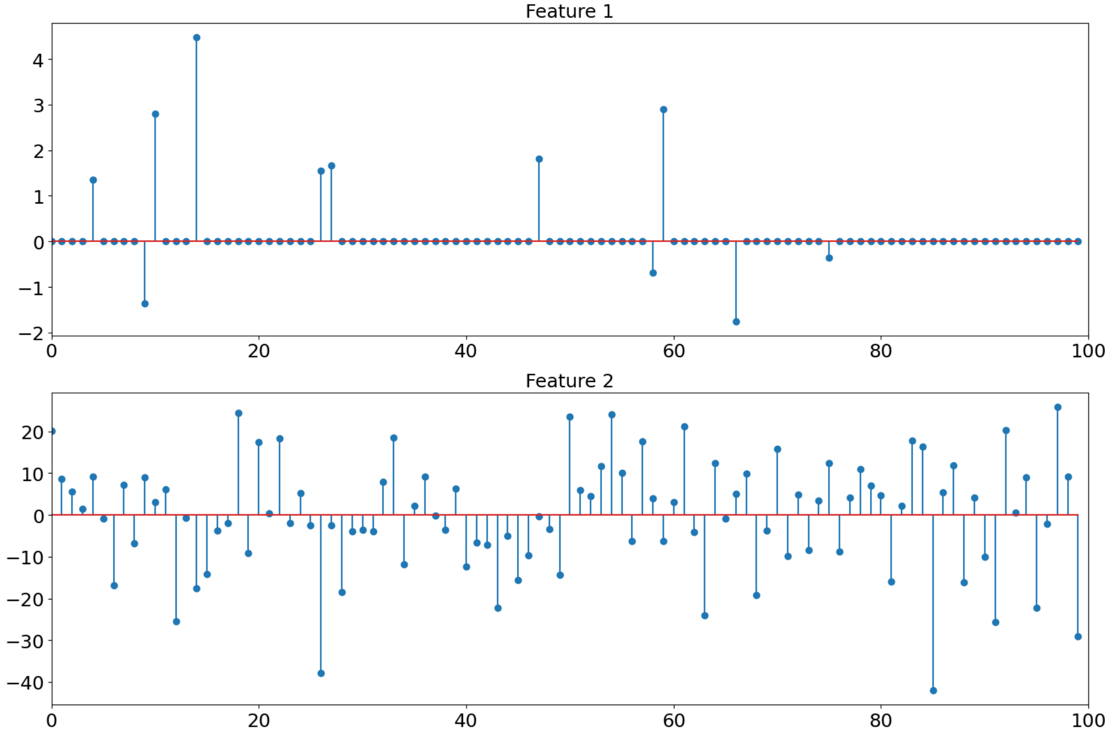


Figure 3. 1

2. Considering Scaling methods for features

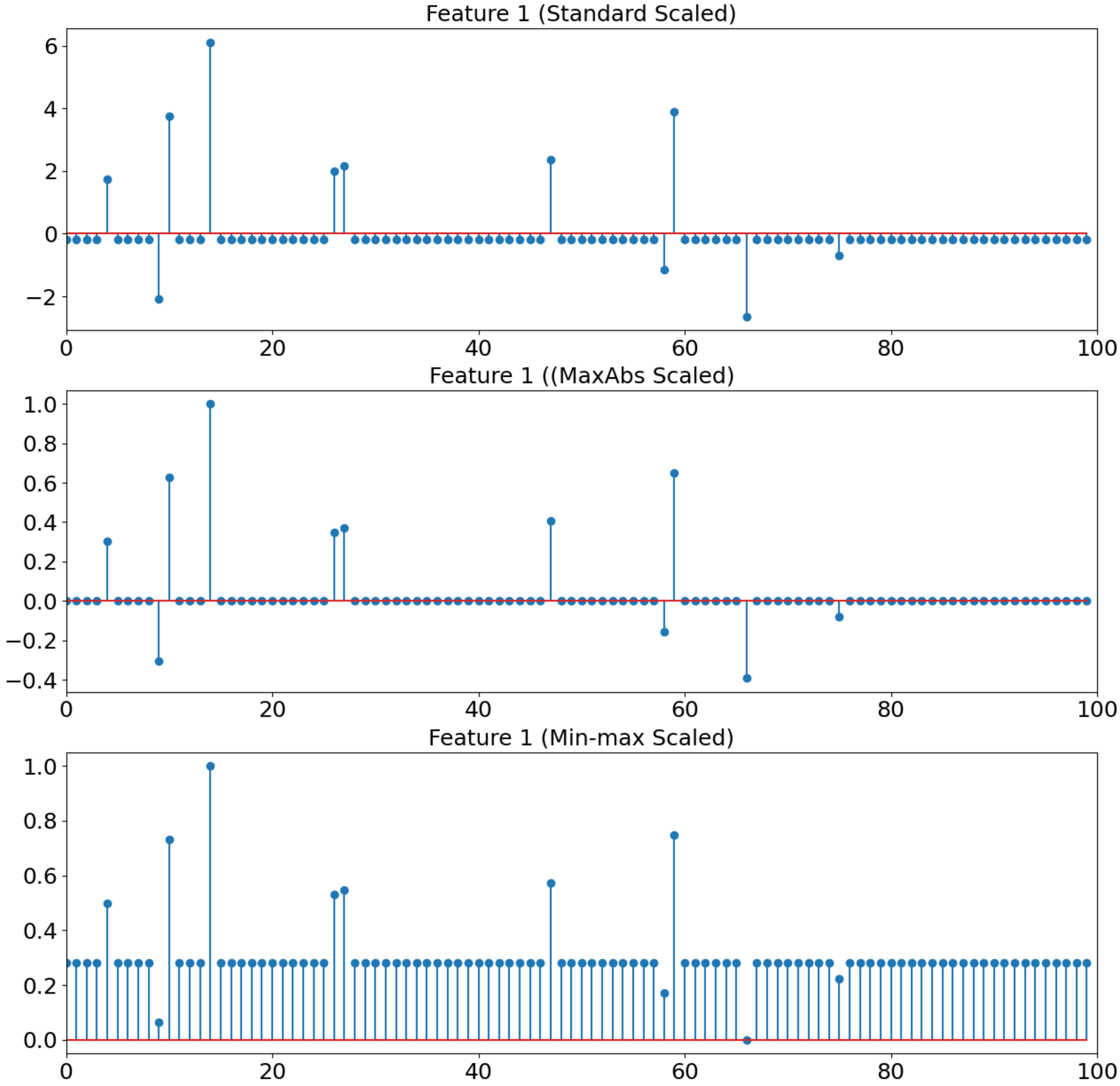
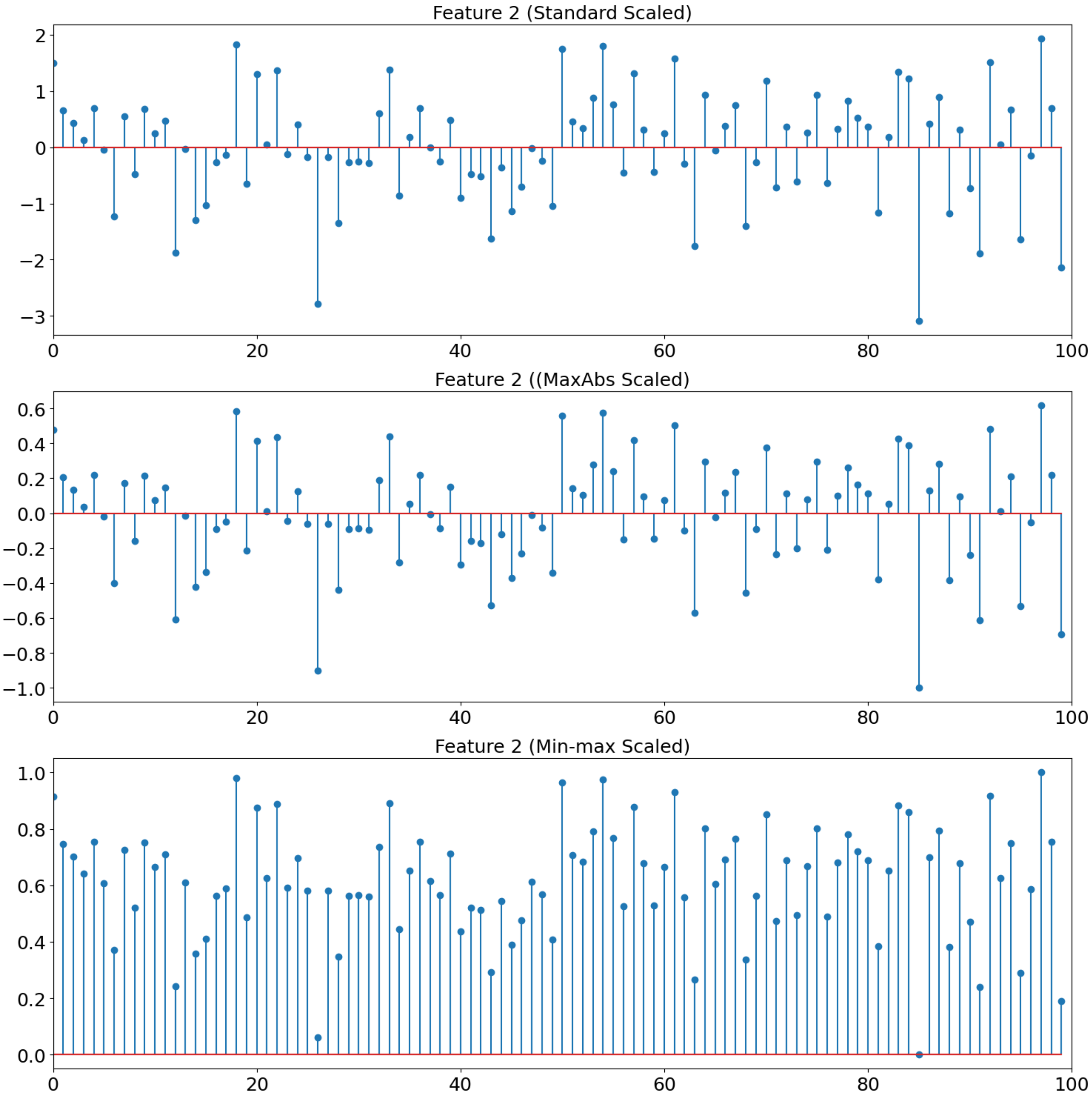


Figure 3. 2



Feature 2 (Standard Scaled)

Feature 2 ((MaxAbs Scaled)

Feature 2 (Min-max Scaled)

Figure 3. 3

Code for using scaling methos for features and plotting

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

def generate_signal(signal_length, num_nonzero):
    """
    Generate a sparse signal with specified number of non-zero elements

    Parameters:
    signal_length (int): Length of the signal to generate
    num_nonzero (int): Number of non-zero elements to include

    Returns:
    numpy.ndarray: Generated sparse signal
    """
    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

# Define parameters
signal_length = 100  # Total length of the signal
num_nonzero = 10    # Number of non-zero elements in the signal
your_index_no = 220332  # Enter your index no without english letters and
without leading zeros

# Generate sparse signal
sparse_signal = generate_signal(signal_length, num_nonzero)

# Modify signal based on index number
sparse_signal[10] = (your_index_no % 10) * 2 + 10
if your_index_no % 10 == 0:
    sparse_signal[10] = np.random.randn(1) + 30

# Scale the signal
sparse_signal = sparse_signal / 5

# Generate random noise for feature 2
epsilon = np.random.normal(0, 15, signal_length)

# Create visualization
plt.figure(figsize=(15, 10))
```

```
# Plot Feature 1
plt.subplot(2, 1, 1)
plt.xlim(0, signal_length)
plt.title("Feature 1", fontsize=18)
plt.xticks(fontsize=18)
plt.yticks(fontsize=18)
plt.stem(sparse_signal)

# Plot Feature 2
plt.subplot(2, 1, 2)
plt.xlim(0, signal_length)
plt.title("Feature 2", fontsize=18)
plt.stem(epsilon)
plt.xticks(fontsize=18)
plt.yticks(fontsize=18)

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

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

# Create scalers
maxabs_scaler = MaxAbsScaler()
standard_scaler = StandardScaler()
minmax_scaler = MinMaxScaler()

# Scale features
feature1_scaled_maxabs =
maxabs_scaler.fit_transform(sparse_signal.reshape(-1, 1)).ravel()
feature1_scaled_std =
standard_scaler.fit_transform(sparse_signal.reshape(-1, 1)).ravel()
feature1_scaled_minmax =
minmax_scaler.fit_transform(sparse_signal.reshape(-1, 1)).ravel()

feature2_scaled_maxabs = maxabs_scaler.fit_transform(epsilon.reshape(-1,
1)).ravel()
feature2_scaled_std = standard_scaler.fit_transform(epsilon.reshape(-1,
1)).ravel()
feature2_scaled_minmax = minmax_scaler.fit_transform(epsilon.reshape(-1,
1)).ravel()
```

```
# Plot scaled Feature 1
plt.figure(figsize=(15, 15))

plt.subplot(3, 1, 1)
plt.xlim(0, signal_length)
plt.title("Feature 1 (Standard Scaled)", fontsize=18)
plt.xticks(fontsize=18)
plt.yticks(fontsize=18)
plt.stem(feature1_scaled_std)

plt.subplot(3, 1, 2)
plt.xlim(0, signal_length)
plt.title("Feature 1 ((MaxAbs Scaled)", fontsize=18)
plt.xticks(fontsize=18)
plt.yticks(fontsize=18)
plt.stem(feature1_scaled_maxabs)

plt.subplot(3, 1, 3)
plt.xlim(0, signal_length)
plt.title("Feature 1 (Min-max Scaled)", fontsize=18)
plt.xticks(fontsize=18)
plt.yticks(fontsize=18)
plt.stem(feature1_scaled_minmax)


# Plot scaled Feature 2
plt.figure(figsize=(15, 15))

plt.subplot(3, 1, 1)
plt.xlim(0, signal_length)
plt.title("Feature 2 (Standard Scaled)", fontsize=18)
plt.xticks(fontsize=18)
plt.yticks(fontsize=18)
plt.stem(feature2_scaled_std)

plt.subplot(3, 1, 2)
plt.xlim(0, signal_length)
plt.title("Feature 2 ((MaxAbs Scaled)", fontsize=18)
plt.xticks(fontsize=18)
plt.yticks(fontsize=18)
plt.stem(feature2_scaled_maxabs)

plt.subplot(3, 1, 3)
plt.xlim(0, signal_length)
plt.title("Feature 2 (Min-max Scaled)", fontsize=18)
plt.xticks(fontsize=18)
plt.yticks(fontsize=18)
plt.stem(feature2_scaled_minmax)

# plt.subplots_adjust(hspace=0.4)
plt.tight_layout()
plt.show()
```

3. Choosing Scaling methods for both features

- **Feature 1: - Sparse Signal**
- **Scaling method: - MaxAbs Scaling**
- **Reason: -** This scaling method preserves sparsity and maintains sign values. Scaler scales data to [-1, 1] range without shifting the mean by preserving the signal structure.

- **Feature 2: - Random Noise**
- **Scaling method: - Standard Scaling**
- **Reason: -** This scaling is ideal for normally distributed data. Centers the data around zero and preserves the distribution's shape. The scaling makes the variance uniform. By these things it handles outliers better than other scalers.

References

1. *Scikit-learn preprocessing data*
2. Introduction to sparsity in signal processing
3. sklearn linear regression

Appendix

GitHub Repo Link for the Assignment 01 - *Link*