
Department of Electronic & Telecommunication Engineering,
University of Moratuwa, Sri Lanka.

# Assignment 1
# Learning from Data and Related Challenges and Linear Models for Regression

Wickramasinghe S.D.
220700T

Submitted in partial fulfillment of the requirements for the module
EN3150 Pattern Recognition

8/19/2025

# Contents

# 1 Introduction

This report investigates the impact of outliers on linear regression models, explores the use of robust estimators to mitigate their influence, and compares common loss functions for different types of prediction tasks. Additionally, it examines feature scaling methods for effective data preprocessing, ensuring that the structure and properties of features are preserved. Through practical examples and calculations, the report demonstrates how proper model selection, loss function choice, and data preparation contribute to building accurate and reliable machine learning models.

# 2 Linear Regression Impact on Outliers

## 2.1 Finding a Linear Regression Model

Using the given set of data points related to the independent variable (x) and the dependent variable (y) in the table, a linear regression model was found to approximate them. A scatter plot of x and y, along with the linear regression model, was plotted. The code for that is as follows.

```
# Define the feature variable
x = np.array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9],dtype=float).reshape(-1, 1)
# Define the target variable
y = np.array([20.26, 5.61, 3.14, -30.00, -40.00, -8.13, -11.73, -16.08, -19.95, -24.03], dtype=float)

# Create the model
model = linear_model.LinearRegression()

# Train the model
model.fit(x, y)

# Plot the results
plt.scatter(x, y, color='blue', label='Actual data')
plt.plot(x, model.predict(x), color='red', linewidth=2, label='Fitted line')
plt.xlabel("x")
plt.ylabel("y")
plt.legend()
plt.show()
```

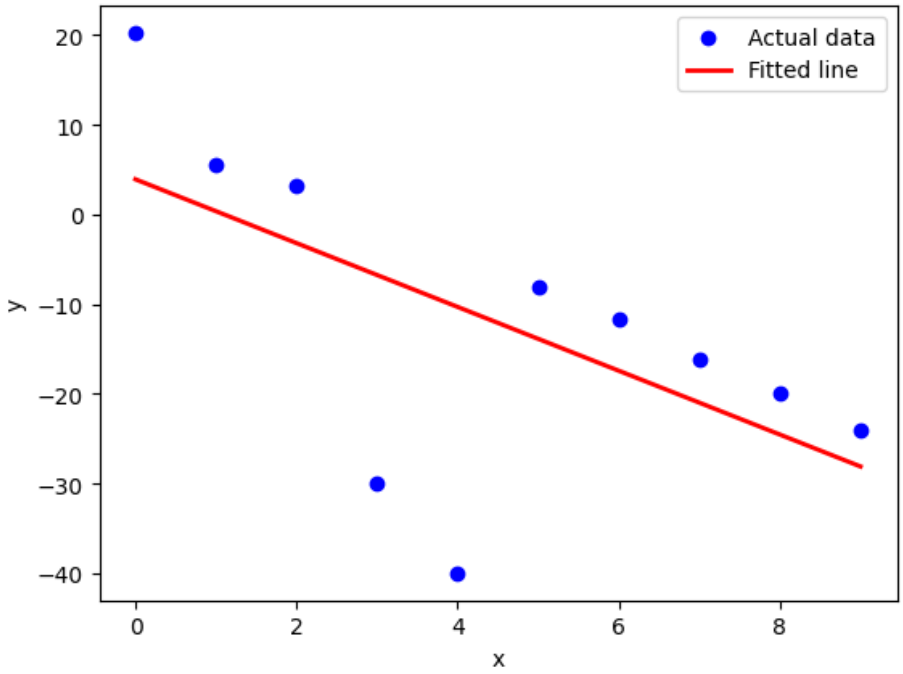


Figure 1: Scatter plot of x and y and the linear regression model

## 2.2 Calculating Loss Function Values

Two linear models were given to model the above points as follows.

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

A robust estimator was introduced to reduce the impact of outliers. The robust estimator finds model parameters that minimize the following loss function,

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

Here, $\theta$ represents the model parameters, $\beta$ is a hyperparameter, and the number of data samples is $N = 10$. Note that $y_i$ and $\hat{y}_i$ are the true and predicted $i$th data sample, respectively.

The loss function values for he above two models are calculated for $\beta = 1$, $\beta = 10^{-6}$, and $\beta = 10^{3}$. The computer program used for that is as follows.

```
# Define models
def model1(x):
  return -4 * x + 12

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

# Robust loss function
def robust_loss(y_true, y_pred, beta):
    errors_sq = (y_true - y_pred)**2
    loss_per_sample = errors_sq / (errors_sq + beta**2)
    return np.mean(loss_per_sample)  # average over N

# Beta values
```

```python
betas = [1, 1e-6, 1000]

# Calculate losses
for beta in betas:
    loss1 = robust_loss(y, model1(x), beta)
    loss2 = robust_loss(y, model2(x), beta)
    print(f" = {beta}")
    print(f"Model 1 Loss: {loss1:.6f}")
    print(f"Model 2 Loss: {loss2:.6f}")
    print()
```

The output of the above program is as follows.

| $\beta$ | **Model 1 Loss** | **Model 2 Loss** |
|---|---|---|
| 1 | 0.929811 | 0.949055 |
| $10^{-6}$ | 1.000000 | 1.000000 |
| 1000 | 0.000461 | 0.000396 |

Table 1: Loss values for Model 1 and Model 2 under different $\beta$ values.

## 2.3 Choosing a Suitable $\beta$ to Mitigate Outliers

If $\beta = 1$:

$$|e| \ll \beta \quad \Rightarrow \quad \text{loss} \approx \frac{e^2}{\beta^2}$$

This provides good sensitivity for inliers.

$$|e| \gg \beta \quad \Rightarrow \quad \text{loss} \approx 1$$

This reduces the effect of large residuals, i.e., outliers.

If $\beta = 1000$:

$$\text{loss} \approx \frac{e^2}{\beta^2} \quad \text{for all points}$$

Since the mean squared error (MSE) of outliers is high, the effect of outliers remains large here.

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

$$\text{loss} \approx 1 \quad \text{for every non-zero point}$$

Here, all points contribute equally (1), which loses magnitude information and useful gradients. The real structure of the data is ignored.

Therefore, $\beta = 1$ is the most suitable value to mitigate the impact of outliers, as it ensures that inliers fit well while outliers are saturated and their influence is reduced.

## 2.4 Selecting the Best Model

As Model 1 yielded the lower loss value for $\beta = 1$, it is the most suitable model of the two.

## 2.5 How the Robust Estimator Reduces the Impact of Outliers

The robust estimator modifies the standard squared error loss by dividing it by $(e^2 + \beta^2)$, where $e$ is the residual.

- For small residuals (inliers), $e^2 \ll \beta^2$, the loss behaves similarly to the Mean Squared Error (MSE), giving these points full influence in determining the model parameters.

- For large residuals (outliers), $e^2 \gg \beta^2$, the loss value approaches 1 and no longer grows with the residual magnitude.

This saturation effect means that very large errors contribute only a limited amount to the total loss, preventing them from dominating the fitting process. As a result, the model focuses more on fitting the majority of the data (inliers) rather than being heavily influenced by extreme points (outliers).

### 2.6 Another Loss Function that can be Used for the Robust Estimator

Huber Loss Function

$$L_\delta(e) = \begin{cases} \frac{1}{2}e^2 & \text{if } |e| \le \delta \quad \text{(quadratic region)} \\ \delta\left(|e| - \frac{1}{2}\delta\right) & \text{if } |e| > \delta \quad \text{(linear region)} \end{cases} \tag{2}$$

This combines the best parts of Mean Squared Error (MSE) and Mean Absolute Error (MAE). For small errors ($|e| \le \delta$), it uses MSE, which is quadratic. So, it is sensitive to small deviations and fits inliers well. For large errors ($|e| > \delta$), it uses MAE, which is linear. So, it reduces the effect of outliers and avoids the large penalty that MSE would give.

## 3 Loss Function

### 3.1 Calculating and Plotting MSE and BCE

We were given two applications, Application 1 and Application 2.

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

It was pallned to train

- A Linear Regression model for Application 1.
- A Logistic Regression model for Application 2.

The code used to calculate and plot the MSE and BCE is as follows.

```
# True value
y_true = 1

# Predictions
y_predictions = 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])

# Calculate MSE
mse_values = (y_true - y_predictions)**2

# Calculate BCE
# Avoid log(0) by adding a small epsilon
epsilon = 1e-15
bce_values = - (y_true * np.log(y_predictions + epsilon) + (1 - y_true) * np.log(1 - y_predictions +
    ↪ epsilon))

# Print the table
print("Table 2: MSE and BCE loss values for different predictions when y=1.")
print("True y=1 | Prediction  | MSE        | BCE")
print("-" * 45)
for i in range(len(y_predictions)):
    print(f"1        | {y_predictions[i]:<12.3f} | {mse_values[i]:<10.6f} | {bce_values[i]:.6f}")

```

```
# Plot the loss functions
plt.figure(figsize=(10, 6))
plt.plot(y_predictions, mse_values, label='MSE', marker='o')
plt.plot(y_predictions, bce_values, label='BCE', marker='x')
plt.xlabel("Prediction ")
plt.ylabel("Loss")
plt.title("MSE and BCE Loss vs. Prediction (y=1)")
plt.legend()
plt.grid(True)
plt.show()
```

| True $y = 1$ | Prediction $\hat{y}$ | MSE | BCE |
|---|---|---|---|
| 1 | 0.005 | 0.990025 | 5.298317 |
| 1 | 0.010 | 0.980100 | 4.605170 |
| 1 | 0.050 | 0.902500 | 2.995732 |
| 1 | 0.100 | 0.810000 | 2.302585 |
| 1 | 0.200 | 0.640000 | 1.609438 |
| 1 | 0.300 | 0.490000 | 1.203973 |
| 1 | 0.400 | 0.360000 | 0.916291 |
| 1 | 0.500 | 0.250000 | 0.693147 |
| 1 | 0.600 | 0.160000 | 0.510826 |
| 1 | 0.700 | 0.090000 | 0.356675 |
| 1 | 0.800 | 0.040000 | 0.223144 |
| 1 | 0.900 | 0.010000 | 0.105361 |
| 1 | 1.000 | 0.000000 | -0.000000 |

Table 2: MSE and BCE loss values for different predictions when $y = 1$

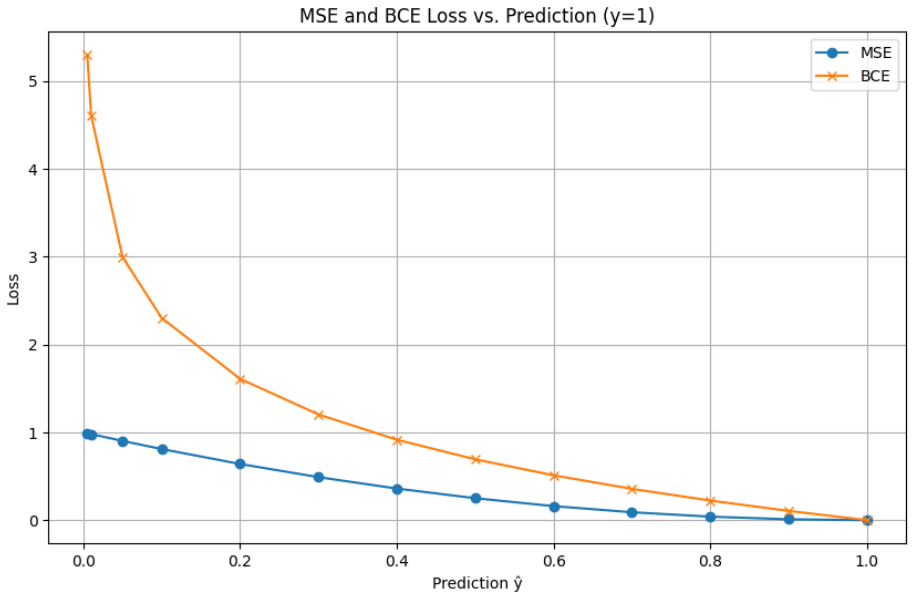


Figure 2: MSE and BCE

## 3.2 Selection of Loss Functions for each Application

**Application 1 - MSE**

MSE is derived from the assumption that the residual errors follow a Gaussian distribution with

constant variance. It penalizes large deviations more heavily because of the square term, which is desirable when we want to minimize significant prediction errors. Moreover, MSE produces smooth gradients, which makes optimization stable for continuous-value predictions.

**Application 2 - BCE**
BCE is derived from the negative log-likelihood of the Bernoulli distribution, making it theoretically grounded for probabilistic classification tasks. It measures the distance between the predicted probability distribution and the true distribution. BCE penalizes confident but wrong predictions very strongly (e.g., predicting 0.99 when the true label is 0), which encourages the model to produce well-calibrated probabilities.

# 4 Data Pre-Processing

## 4.1 Generating Feature Values

Feature values for two features were generated using the following code.

```
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

signal_length = 100 # Total length of the signal
num_nonzero = 10 # Number of non-zero elements in the signal
your_index_no = 220700

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
  epsilon = np.random.normal(0, 15, signal_length )

#epsilon=epsilon[:, np.newaxis]
plt.figure(figsize=(15,10))
plt.subplot(2, 1, 1)
plt.xlim(0, signal_length)
plt.title("Feature 1", fontsize=18)
plt.xticks(fontsize=18) # Adjust x-axis tick label font size
plt.yticks(fontsize=18)
plt.stem(sparse_signal)
plt.subplot(2, 1, 2)
plt.xlim(0, signal_length)
plt.title("Feature 2", fontsize=18)
plt.stem(epsilon)
plt.xticks(fontsize=18) # Adjust x-axis tick label font size
plt.yticks(fontsize=18)
plt.show()
```

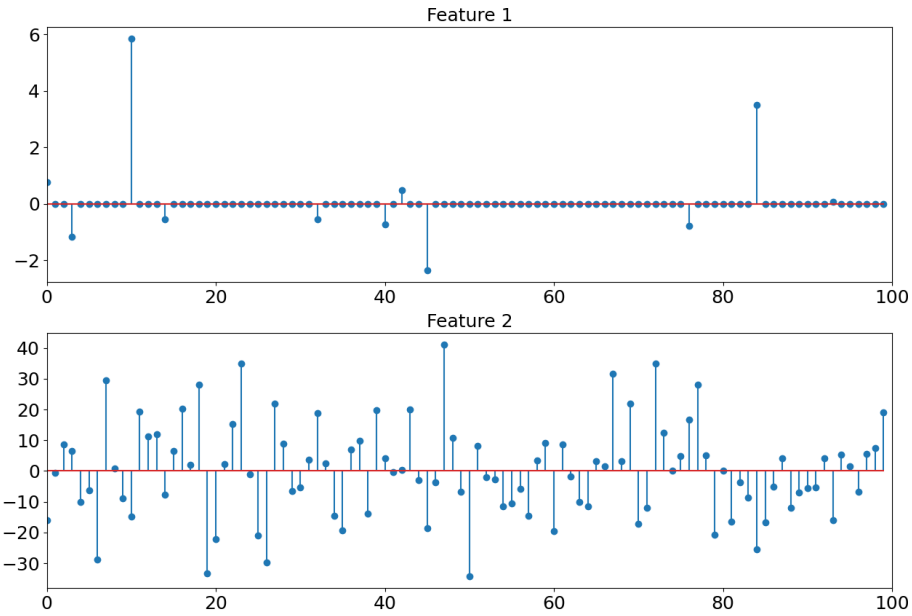


Figure 3: Feature 1 and Feature 1

## 4.2 Standard Scaling

The two features were standard-scaled using the following code.

```
from sklearn.preprocessing import StandardScaler

# Reshape the signals for StandardScaler
sparse_signal_reshaped = sparse_signal.reshape(-1, 1)
epsilon_reshaped = epsilon.reshape(-1, 1)

# Apply Standard Scaling
scaler_sparse = StandardScaler()
scaled_sparse_signal = scaler_sparse.fit_transform(sparse_signal_reshaped)

scaler_epsilon = StandardScaler()
scaled_epsilon = scaler_epsilon.fit_transform(epsilon_reshaped)

# Plot the scaled signals
plt.figure(figsize=(15, 10))
plt.subplot(2, 1, 1)
plt.xlim(0, signal_length)
plt.title("Standard Scaled Feature 1", fontsize=18)
plt.xticks(fontsize=18)
plt.yticks(fontsize=18)
plt.stem(scaled_sparse_signal)

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

plt.show()
```

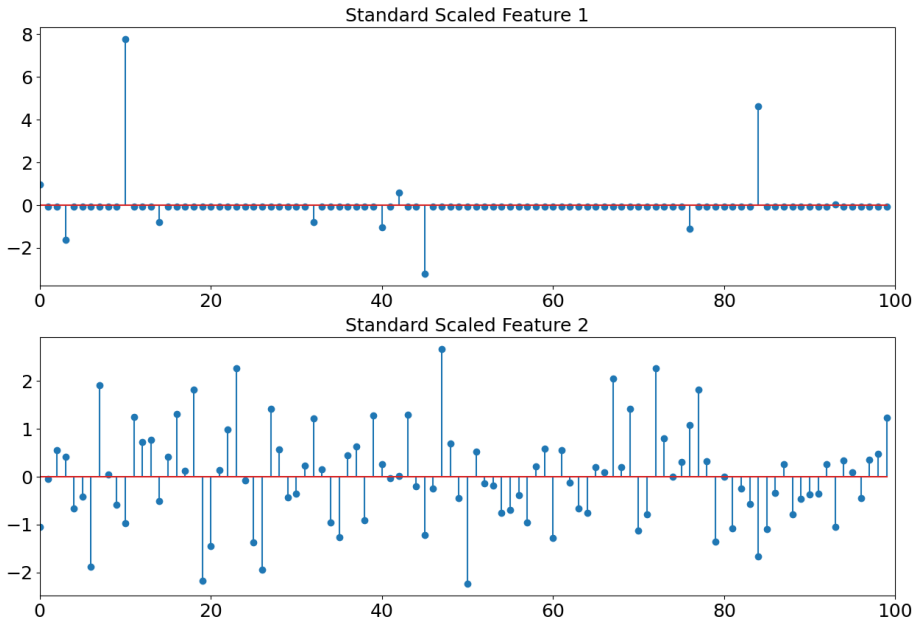


Figure 4: Standard-Scaled Features

## 4.3 Min-Max Scaling

The two features were min-max scaled using the following code.

```
from sklearn.preprocessing import MinMaxScaler

# Reshape the signals for MinMaxScaler
sparse_signal_reshaped = sparse_signal.reshape(-1, 1)
epsilon_reshaped = epsilon.reshape(-1, 1)

# Apply Min-Max Scaling
scaler_sparse_minmax = MinMaxScaler()
scaled_sparse_signal_minmax = scaler_sparse_minmax.fit_transform(sparse_signal_reshaped)

scaler_epsilon_minmax = MinMaxScaler()
scaled_epsilon_minmax = scaler_epsilon_minmax.fit_transform(epsilon_reshaped)

# Plot the scaled signals
plt.figure(figsize=(15, 10))
plt.subplot(2, 1, 1)
plt.xlim(0, signal_length)
plt.title("Min-Max Scaled Feature 1", fontsize=18)
plt.xticks(fontsize=18)
plt.yticks(fontsize=18)
plt.stem(scaled_sparse_signal_minmax)

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

plt.show()
```

## 4.4 Max-Abs Scaling

The two features were max abs scaled using the following code.

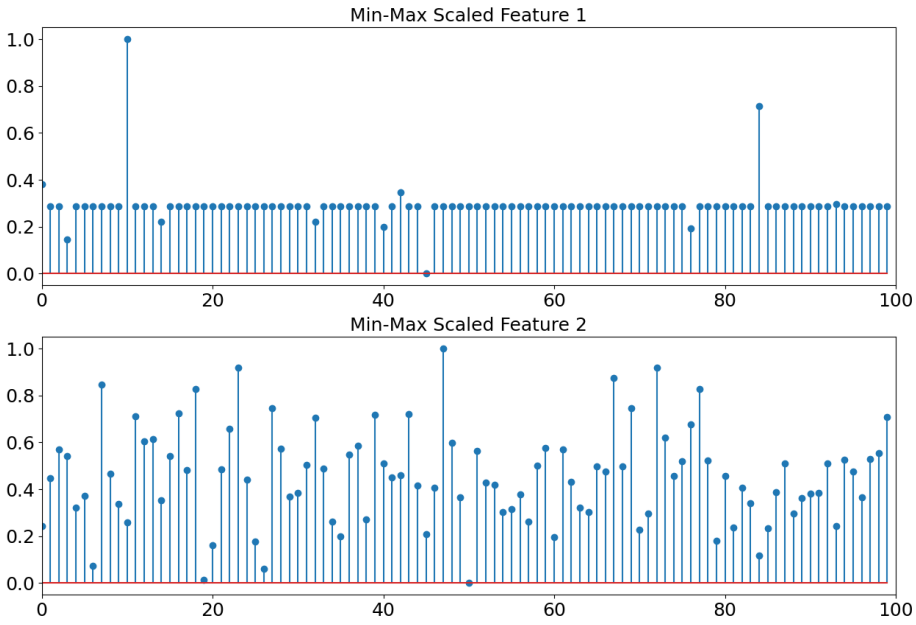


Figure 5: Min-Max Scaled Features

```python
from sklearn.preprocessing import MaxAbsScaler

# Reshape the signals for MaxAbsScaler
sparse_signal_reshaped = sparse_signal.reshape(-1, 1)
epsilon_reshaped = epsilon.reshape(-1, 1)

# Apply Max Abs Scaling
scaler_sparse_maxabs = MaxAbsScaler()
scaled_sparse_signal_maxabs = scaler_sparse_maxabs.fit_transform(sparse_signal_reshaped)

scaler_epsilon_maxabs = MaxAbsScaler()
scaled_epsilon_maxabs = scaler_epsilon_maxabs.fit_transform(epsilon_reshaped)

# Plot the scaled signals
plt.figure(figsize=(15, 10))
plt.subplot(2, 1, 1)
plt.xlim(0, signal_length)
plt.title("Max Abs Scaled Feature 1", fontsize=18)
plt.xticks(fontsize=18)
plt.yticks(fontsize=18)
plt.stem(scaled_sparse_signal_maxabs)

plt.subplot(2, 1, 2)
plt.xlim(0, signal_length)
plt.title("Max Abs Scaled Feature 2", fontsize=18)
plt.xticks(fontsize=18)
plt.yticks(fontsize=18)
plt.stem(scaled_epsilon_maxabs)

plt.show()
```

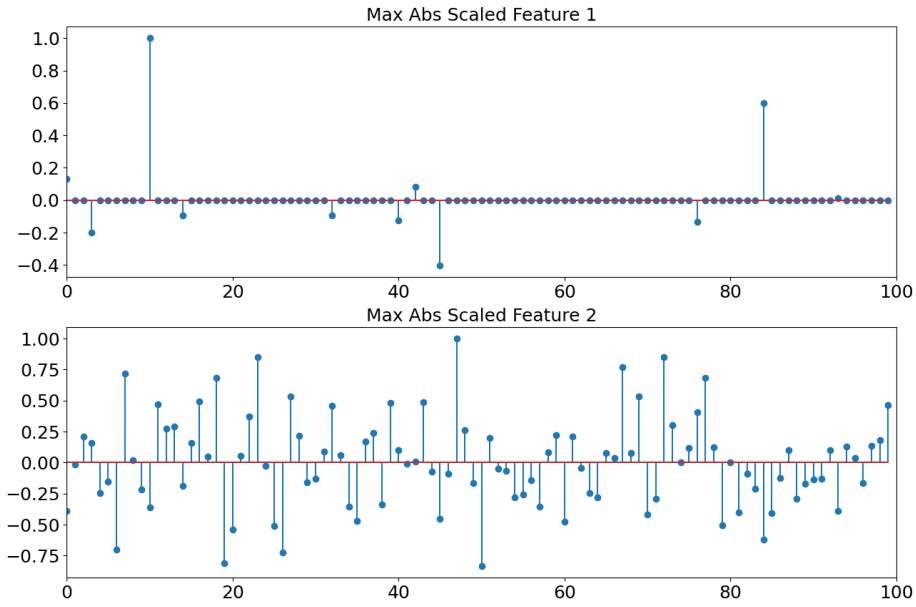


Figure 6: Max-Abs Scaled Features

#### 4.4.1 Selecting the Best Scaling Method

**Feature 1 - Max-Abs Scaling**
As feature 1 follows a sparse distribution, we want to keep zeros as zeros, keep the sign, and avoid shifting of the mean. As max-abs scaling divides the values by the absolute maximum value of the range, it does not change the zeros, but only scales the non-zero values. Also, dividing by the absolute value keeps positive and negative spikes proportional to their real value. As this scales magnitudes into [-1,1] without shifting the mean, this protects the sparse Nature.

**Feature 2 - Standard Scaling**
As feature 2 follows a Gaussian distribution, we need to normalize the mean and variance for stability. As standard scaling centers the distribution to zero mean and scales the variance to 1, this keeps the statistical assumptions of Gaussian features protected.

## 5 Conclusion

This report explored key aspects of machine learning, including linear regression, robust estimation, loss functions, and data preprocessing. Using the given dataset, we demonstrated how outliers can significantly affect linear regression models and how a robust estimator reduces their impact by limiting the influence of extreme residuals. Through the calculation of the robust loss for different $\beta$ values, an optimal parameter was identified for a more reliable model selection.

This also analyzed loss functions for two applications, using Mean Squared Error (MSE) for continuous regression tasks and Binary Cross-Entropy (BCE) for binary classification, highlighting the importance of choosing the right loss function for the target type. Finally, feature scaling methods, Standard Scaling, Min-Max Scaling, and Max-Abs Scaling, were applied to generated features, ensuring the data structure was preserved and optimizing model performance.