University of Moratuwa

Department of Electronic and Telecommunication Engineering

EN3150 - Pattern Recognition

Jayaweera M.V.L.M - 220285X

Learning from Data and Related Challenges and Linear Models for Regression

August 20, 2025

Link to GitHub repository: Click here

1 Linear Regression Impact on Outliers

1.1 Find a regression model

The dataset provided in Table 1 consists of 10 samples of independent variable x and dependent variable y.

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

Table 1: Dataset used for regression.

Using all data points, a linear regression model was fitted. The estimated model is:

$$y = -3.56x + 3.92$$

which we will call **Model 2**.
Figure 1 shows the scatter plot of data points together with the regression line. I have used the **sklearn** library to get the linear regression model. This is the code snippet for it.

```python
from sklearn.linear_model import LinearRegression

# Reshape x into 2D for sklearn
X = df['x_i'].values.reshape(-1, 1)
y = df['y_i'].values

# Fit model
model = LinearRegression()
model.fit(X, y)

# Get coefficients
slope = model.coef_[0]
intercept = model.intercept_
print(f"Linear Regression Model: y = {slope:.2f}x + {intercept:.2f}")
```

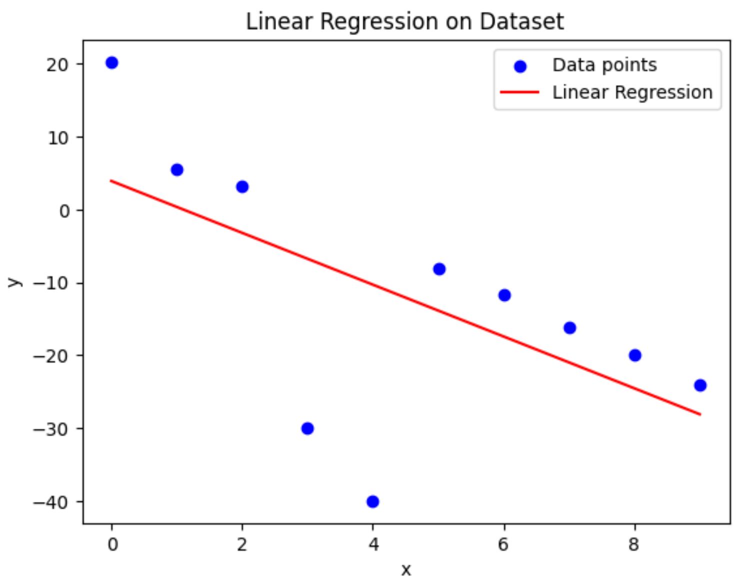

Figure 1: Scatter plot of data points with fitted regression line.

1.2 Given Models

We are provided with two linear models:

$$\text{Model 1: } y = -4x + 12$$

$$\text{Model 2: } y = -3.55x + 3.91$$

1.3 Robust Estimator Loss Function

The robust loss function is defined as:

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

where β is a hyperparameter controlling the sensitivity to outliers. I have used the following code snippet to calculate the loss.

```python
import numpy as np
import pandas as pd

x = df['x_i'].values
y = df['y_i'].values

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

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

```python

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

# Betas
betas = [1, 1e-6, 1e3]

# Calculate losses for both models
results = []
for beta in betas:
    loss_m1 = robust_loss(y, model1(x), beta)
    loss_m2 = robust_loss(y, model2(x), beta)
    results.append([beta, loss_m1, loss_m2])

# Put into DataFrame
df_results = pd.DataFrame(results, columns=["Beta", "Loss_Model1", "
    Loss_Model2"])
print(df_results)
```

Table 2 shows the calculated loss values for the two models with $\beta = 1, 10^{-6}, 10^3$.

β	Loss (Model 1)	Loss (Model 2)
1	0.4354	0.9728
10^{-6}	1.0000	1.0000
10^3	0.000227	0.000188

Table 2: Robust loss values for the two models under different β.

1.4 Selection of Suitable β

- **Very small** β **(10^{-6})**: denominator $\approx$ error^2, giving loss ≈ 1 for all points. Outliers still dominate.
- **Very large** β **(10^3)**: denominator $\approx \beta^2$, shrinking all contributions ≈ 0. Loss loses meaning.
- **Moderate** β **(1)**: balances normal samples and outliers, down-weighting outliers while preserving contribution from regular points.

Therefore, $\beta = 1$ is most suitable for mitigating the influence of outliers.

1.5 Most Suitable β value

The suitable β value is $\beta = 1$, because:
When β is too small, outliers dominate (loss $= 1$ for both models). When β is too large, all points lose their influence (loss ≈ 0). A moderate β (like 1) strikes a balance: normal points are considered properly, and outliers have reduced influence.

1.6 Most suitable model

The most suitable model is **Model 1**:

$$y = -4x + 12$$

because with the robust estimator at $\beta = 1$, it achieves a lower loss (0.4354) compared to Model 2 (0.9728). This shows Model 1 better captures the main trend of the data while mitigating the influence of outliers.

1.7 Effect of the Robust Estimator

The robust estimator reduces the influence of outliers by scaling their squared errors:

$$\frac{(y - \hat{y})^2}{(y - \hat{y})^2 + \beta^2}$$

Large errors (from outliers) get values close to 1, but since the denominator grows, their impact is reduced relative to smaller, more consistent errors. The ordinary regression model (Model 2) was pulled heavily by the large outliers

$$y = -30 \text{ at } x = 3 \quad \text{and} \quad y = -40 \text{ at } x = 4.$$

The robust estimator downweights these extreme points, giving more importance to the general trend of the other data points.
Under this criterion, **Model 1** achieves a significantly lower robust loss, making it the most suitable model for this dataset.

1.8 Alternative Loss Function

Another robust loss function is the **RANSAC consensus loss**, which evaluates models based on the proportion of inliers within a threshold τ. Formally:

$$L_{\text{RANSAC}} = 1 - \frac{\#\text{inliers}}{N}$$

with a trimmed MSE over inliers used for tie-breaking.
Table 3 shows example RANSAC losses for both models at different τ values.

τ	ConsensusLoss (M1)	ConsensusLoss (M2)	tMSE (M1)	tMSE (M2)
1.0	0.4	1.0	0.14	∞
2.0	0.4	1.0	0.14	∞
5.0	0.3	0.7	0.94	20.10
10.0	0.2	0.3	9.35	27.51

Table 3: Consensus loss and trimmed MSE for different thresholds τ.

2 Loss Function

Suppose we have two applications:

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

We plan to train:

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

Two common loss functions are:

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

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

2.1 Loss Table for $y = 1$

Following is my code snippet for fill the table.

```python
import numpy as np
import pandas as pd

# True value
y_true = 1
# Predictions
predictions = [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]
# Initialize lists to store loss values
mse_values = []
bce_values = []
# Calculate MSE and BCE for each prediction
for y_pred in predictions:
    mse = (y_true - y_pred)**2
    # Avoid log(0) error for BCE
    y_pred_clipped = np.clip(y_pred, 1e-15, 1-1e-15)
    bce = - (y_true * np.log(y_pred_clipped) + (1 - y_true) * np.log(1 -
    y_pred_clipped))
    mse_values.append(round(mse, 6))  # rounded for readability
    bce_values.append(round(bce, 6))
# Create a DataFrame (table)
table = pd.DataFrame({
    'True y': [y_true]*len(predictions),
    'Prediction $\hat{y}$': predictions,
    'MSE': mse_values,
    'BCE': bce_values
})
# Print the table
print(table)
```

Table 4 shows the MSE and BCE loss values for different predictions when $y = 1$.

True y	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

Table 4: MSE and BCE values for different predictions when $y = 1$.

2.2 Loss Function Plots

Figure 2 shows the variation of MSE and BCE with respect to the predictions $\hat{y}$. BCE penalizes incorrect predictions more sharply compared to MSE.

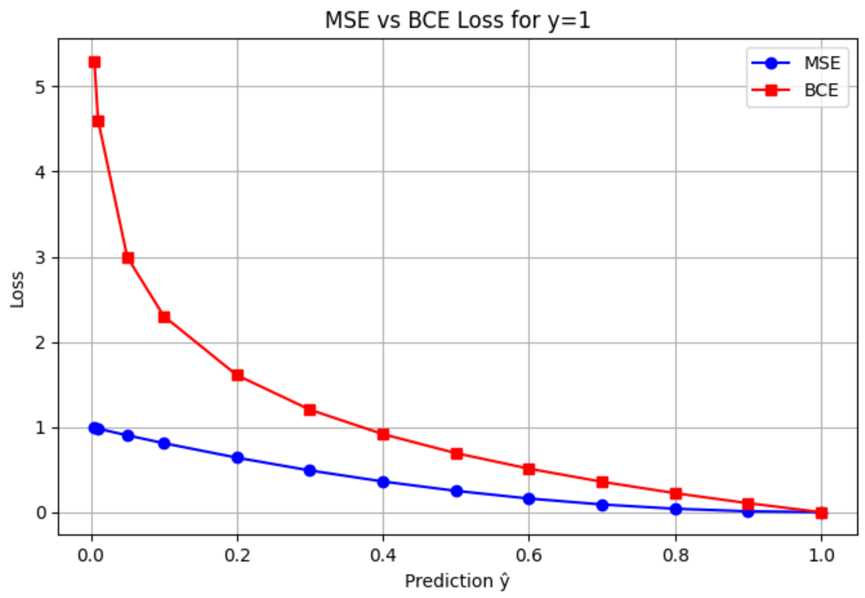

Figure 2: Comparison of MSE and BCE for $y = 1$.

2.3 Choice of Loss Function

Application 1: Linear Regression (Continuous y)

- Recommended Loss: **MSE**.
- **Justification:** The dependent variable is continuous, and MSE directly measures the squared difference between actual and predicted values. It is smooth, differentiable, penalizes large errors more strongly, and is standard for regression tasks.

Application 2: Logistic Regression (Binary $y \in \{0, 1\}$)

- Recommended Loss: **BCE**.
- **Justification:** Logistic regression outputs probabilities $\hat{y} \in [0, 1]$. BCE is the appropriate choice because it penalizes confident wrong predictions heavily (e.g., $\hat{y} \approx 0$ when $y = 1$). MSE is not ideal for classification as it treats errors linearly and leads to poor probability calibration, whereas BCE aligns with maximum likelihood estimation for binary data.

3 Data Pre-processing: Feature Scaling

3.1 Feature Generation

Two features were generated using the given code (Listing 1):

- **Feature 1:** Sparse signal with 10 non-zero elements and an index-specific adjustment.
- **Feature 2:** Random noise signal generated from a normal distribution.

Below shows the code snippet which i used to generate the feature values and do the scalling.

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

# Generate features
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
num_nonzero = 10
your_index_no = 285  #220285X
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)

# Plot original features
plt.figure(figsize=(15, 10))
plt.subplot(2,1,1)
plt.stem(sparse_signal)
plt.title("Feature 1: Sparse Signal", fontsize=18)
plt.xlim(0, signal_length)
plt.xticks(fontsize=14)
plt.yticks(fontsize=14)

plt.subplot(2,1,2)
plt.stem(epsilon)
plt.title("Feature 2: Noise Signal", fontsize=18)
plt.xlim(0, signal_length)
plt.xticks(fontsize=14)
plt.yticks(fontsize=14)
plt.show()

# Scaling
scalers = {
    'Standard': StandardScaler(),
    'Min-Max': MinMaxScaler(),
    'Max-Abs': MaxAbsScaler()
}
feature1_scaled = {}
feature2_scaled = {}
for name, scaler in scalers.items():
    feature1_scaled[name] = scaler.fit_transform(sparse_signal.reshape
    (-1,1)).flatten()
    feature2_scaled[name] = scaler.fit_transform(epsilon.reshape(-1,1)).
    flatten()

# Plot scaled features
plt.figure(figsize=(15,10))
for i, (name, scaled) in enumerate(feature1_scaled.items(), 1):
    plt.subplot(3,2,2*i-1)
    plt.stem(scaled)
    plt.title(f"Feature 1 - {name} Scaled", fontsize=14)
for i, (name, scaled) in enumerate(feature2_scaled.items(), 1):
    plt.subplot(3,2,2*i)
    plt.stem(scaled)
    plt.title(f"Feature 2 - {name} Scaled", fontsize=14)
plt.tight_layout()
plt.show()
```

Figure 3 shows the original feature values.

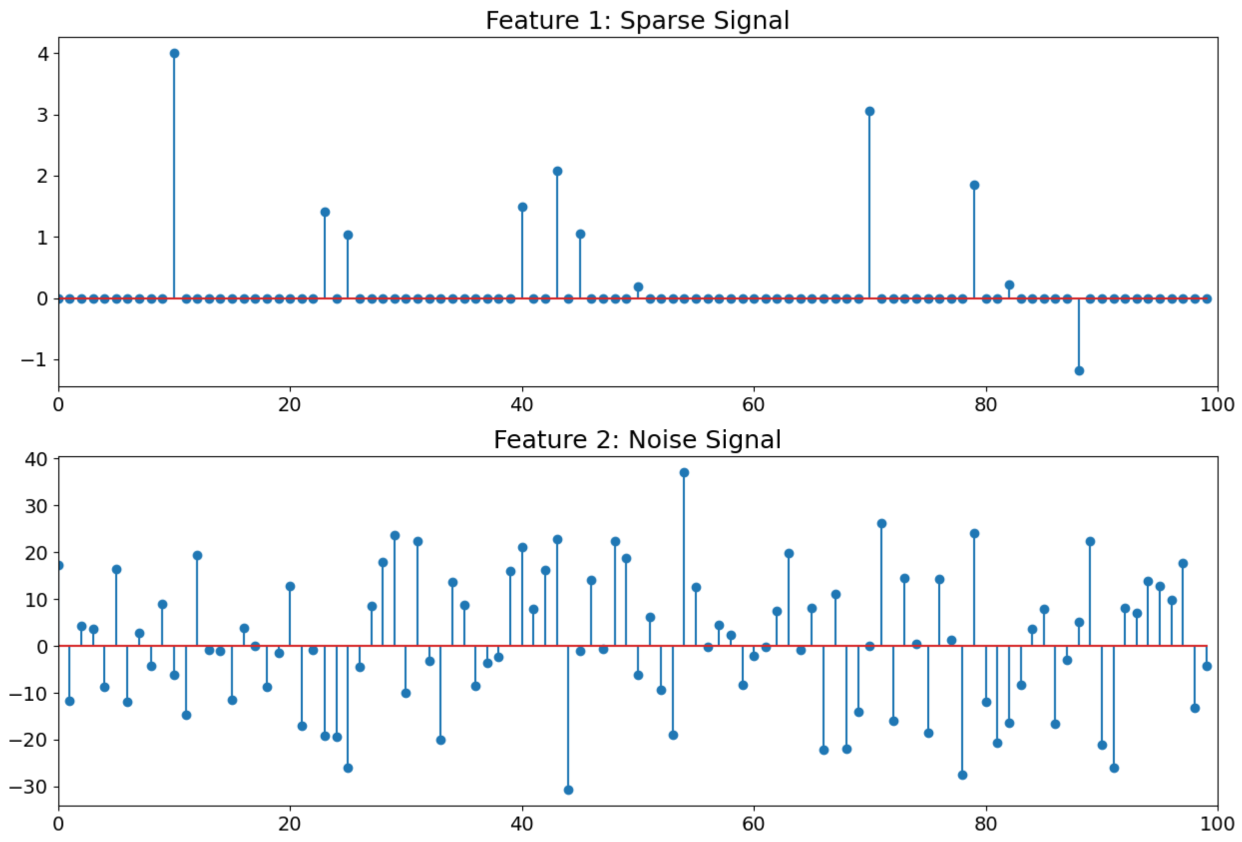

Figure 3: Original feature values: (a) Sparse signal (Feature 1), (b) Noise signal (Feature 2)

3.2 Scaling Methods

Three scaling methods were applied to both features:

1. **Standard Scaling:** Centers data to mean 0 with standard deviation 1.
2. **Min-Max Scaling:** Scales data to range $[0, 1]$.
3. **Max-Abs Scaling:** Scales data by the maximum absolute value, preserving sparsity and sign.

Figure 4 shows all three scaled versions of Feature 1 and Feature 2.

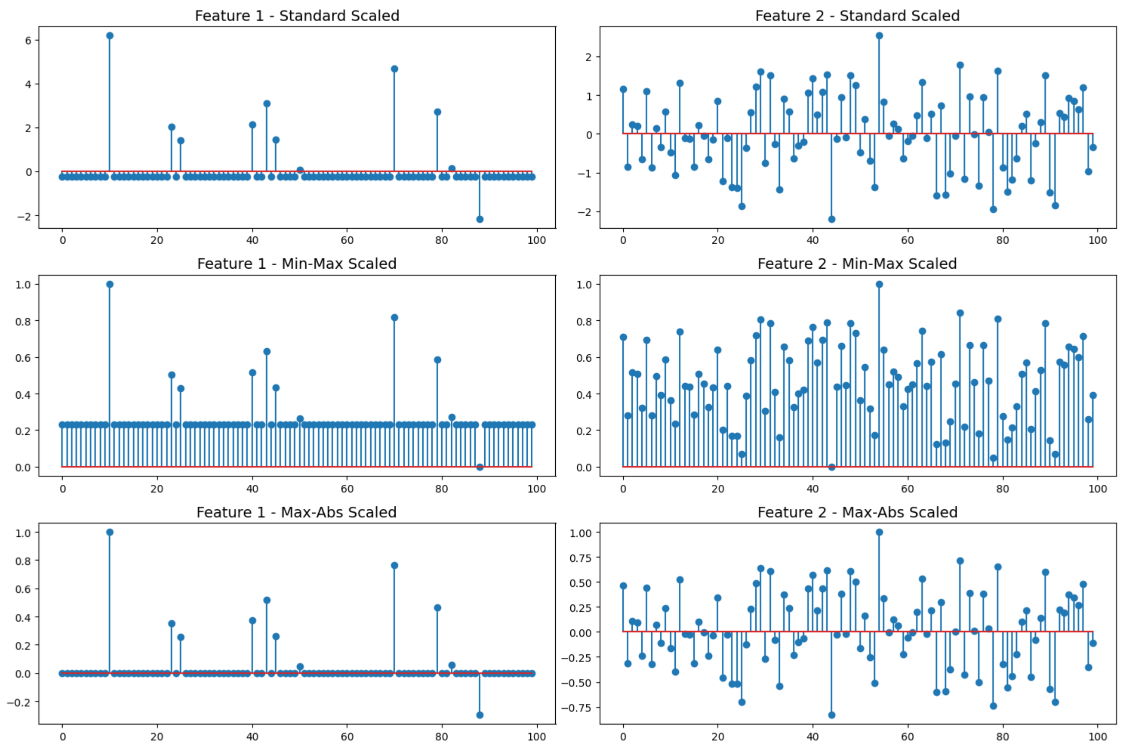

Figure 4: Scaled feature values: each feature scaled using Standard, Min-Max, and Max-Abs scaling.

3.3 Selected Scaling Method and Justification

- **Feature 1 (Sparse Signal):** *Max-Abs Scaling* was selected. **Justification:** This feature contains many zeros and a few extreme values. Max-Abs scaling preserves the sparsity (zeros remain zeros) and the sign of values, while scaling the non-zero values into the range $[-1, 1]$. Standard scaling or Min-Max scaling would distort the sparsity and relative magnitude of non-zero elements.

- **Feature 2 (Noise Signal):** *Standard Scaling* was selected. **Justification:** Feature 2 is approximately normally distributed. Standard scaling centers the data to mean 0 and unit variance, which preserves the distribution structure and is suitable for most algorithms that assume zero-centered input.

3.4 Summary

By applying appropriate scaling methods, the structure and properties of both features are preserved, ensuring that subsequent learning algorithms operate efficiently without distortion of the data distribution.