University of Moratuwa

Department of Electronic and Telecommunication Engineering

EN3150– Pattern Recognition

Learning from data and related challenges and linear models

Submitted by:
Balasooriya B.A.P.I. 220054N

Jupyter Notebook

The full implementation and results of this assignment can be found in the above Jupyter Notebook.

Date: August 21, 2025

Contents

1 Linear regression impact on outliers

1.1 Dataset

The given dataset, that contains 10 data points with independent variable x and dependent variable y is loaded in to a numpy array and then converted in to a pandas dataframe.

```
import numpy as np
import pandas as pd

x = np.array([0,1,2,3,4,5,6,7,8,9])
y = np.array([20.26,5.61,3.14,-30.0,-40.0,-8.13,-11.73,-16.08,-19.95,-24.03])

df = pd.DataFrame({'x':x, 'y':y})
```

Listing 1: Loading data in to pandas dataframe

1.2 Linear Regression Model

Using ordinary least squares regression on the complete dataset, the linear regression model is fitted to determine the relationship between x and y.

```
from sklearn.linear_model import LinearRegression

model = LinearRegression()
model.fit(x.reshape(-1,1), y)

slope = model.coef_[0]
intercept = model.intercept_

plt.scatter(x,y,label="Data")
plt.plot(x, model.predict(x.reshape(-1,1)), color="red", label="Linear Regression")
plt.legend()
plt.show()

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

Listing 2: Linear Regression Implementation and Ploting

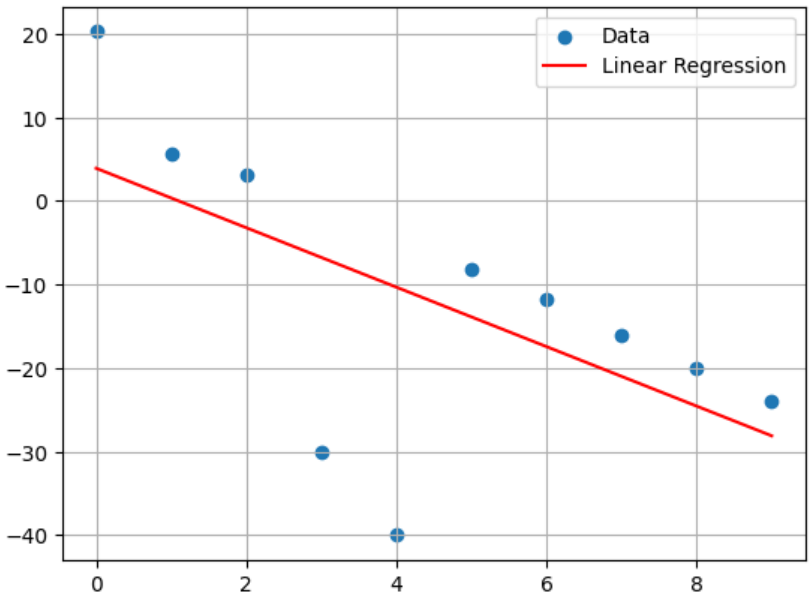


Figure 1: Scatter plot of data points with fitted linear regression line

The fitted linear regression model is,

$$\hat{y} = -3.557x + 3.917 \tag{1}$$

1.3 Robust Estimator

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

Listing 3: Robust Loss Function Implementation

1.4 Model Comparison

Two models are evaluated using the robust estimator:

$$\text{Model 1:} \quad y = -4x + 12 \tag{2}$$

$$\text{Model 2:} \quad y = -3.55x + 3.91 \tag{3}$$

```python
def model1(x): return -4*x + 12
def model2(x): return -3.55*x + 3.91
```

Listing 4: Linear models

```python
betas = [1, 1e-6, 1e3]

for beta in betas:
    L1 = robust_loss(y, model1(x), beta)
    L2 = robust_loss(y, model2(x), beta)
    print(f"β={beta}: Model1={L1:.8f}, Model2={L2:.8f}")
```

Listing 5: Robust Loss Function Implementation

β Value	Model 1 Loss	Model 2 Loss
10^{-6}	1.00000000	1.00000000
1	0.43541626	0.97284705
10^{3}	0.00022683	0.00018825

Table 1: Robust Loss Function Values for Different β Parameters

1.5 Optimal β Selection

Analysis and Justification:

The selection of an appropriate β value is crucial for balancing model accuracy and outlier robustness.

- **Small β ($\beta \approx 10^{-6}$):** The loss values for both models are 1.0, indicating that the loss function behaves almost identically for all residuals, making it highly sensitive to outliers. Thus, this setting does not help in distinguishing robust models.

- **Medium β ($\beta = 1$):** The loss for Model 1 drops to 0.435 while Model 2 remains high at 0.973. This shows that the robust loss can effectively downweight the influence of outliers and favor the model that better represents the majority of the data. Hence, $\beta = 1$ provides a meaningful balance between fitting the main trend and resisting outliers.

- **Large β ($\beta \approx 10^3$):** Both models have extremely small losses ($\sim 10^{-4}$), indicating that the influence of all residuals is almost equal. This diminishes the model discrimination and may lead to underfitting.

Based on the robust loss results, $\beta = 1$ is selected as the optimal value. It effectively distinguishes between the better-fitting model (Model 1) and a model influenced by outliers (Model 2), providing a good trade-off between accuracy and outlier robustness.

1.6 Model Selection Using Robust Estimator

Using the optimal $\beta = 1$,

Model Selection Results:

- Model 1 Loss: 0.43541626
- Model 2 Loss: 0.97284705
- Selected Model: Model 1

Justification:

Using the robust loss function with $\beta = 1$, Model 1 has a significantly lower loss (0.435) compared to Model 2 (0.973). This indicates that Model 1 fits the majority of the data points more accurately while reducing the influence of extreme outliers present in the dataset. In contrast, Model 2 is more affected by the outliers, resulting in a higher robust loss. Therefore, Model 1 is selected as the preferred model as it provides a better balance between capturing the underlying trend and maintaining robustness against outliers.

1.7 How Robust Estimator Reduces Outlier Impact

The robust estimator reduces the influence of outliers primarily through **weighting** of residuals: Each data point is assigned a weight based on the magnitude of its residual

$$w_i = \frac{1}{(y_i - \hat{y}_i)^2 + \beta^2} \tag{4}$$

Large residuals, corresponding to outliers, result in smaller w_i, reducing their contribution to the overall loss. Conversely, residuals near the trend line retain higher weights, preserving the influence of normal data points. This continuous weighting automatically adjusts the influence of each point based on its deviation from the model, allowing the estimator to fit the main trend without being dominated by extreme values.

1.8 Alternative for Robust Loss Function

An alternative loss function for robust estimation is the **Huber Loss**:

$$L_{\text{Huber}}(r) = \begin{cases} \frac{1}{2}r^2 & \text{if } |r| \leq \delta \\ \delta|r| - \frac{1}{2}\delta^2 & \text{if } |r| > \delta \end{cases} \tag{5}$$

where $r = y_i - \hat{y}_i$ and δ is a threshold parameter.

2 Loss Function

2.1 Loss Function Computation and Comparison

For a true label $y = 1$, the loss functions are computed for various prediction values.

```
def mse(y, yhat):
    return (y - yhat)**2

def bce(y, yhat):
    return -(y*np.log(yhat) + (1-y)*np.log(1-yhat))

preds = [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]
results = []

for yhat in preds:
    results.append([1, yhat, mse(1,yhat), bce(1,yhat)])

df2 = pd.DataFrame(results, columns=["y","y_hat","MSE","BCE"])
df2
```

Listing 6: Loss Function Calculation

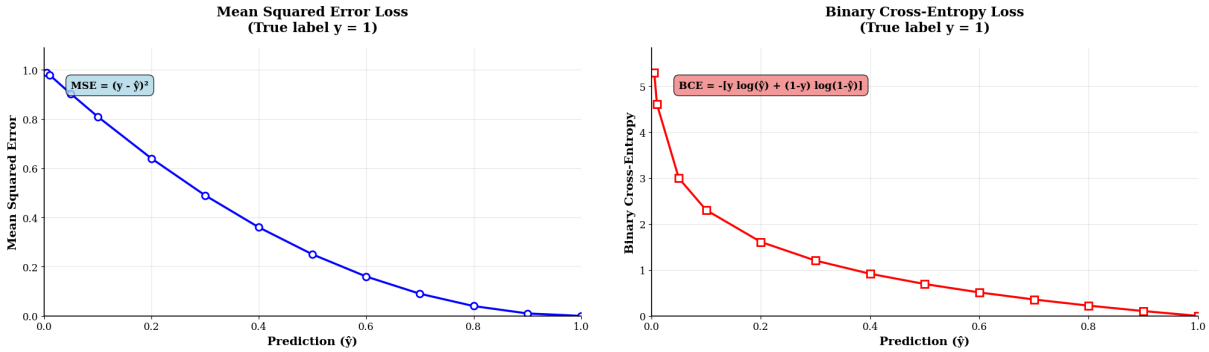
Table 2: MSE and BCE Loss Values for Different Predictions when $y = 1$

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	NaN

2.2 Loss Function Selection

2.2.1 Application 1: *Loss Function:* Mean Squared Error (MSE)

2.2.2 Application 2: *Loss Function:* Binary Cross Entropy (BCE)



(a) MSE Loss Function (b) BCE Loss Function

Figure 2: Individual loss function behaviors

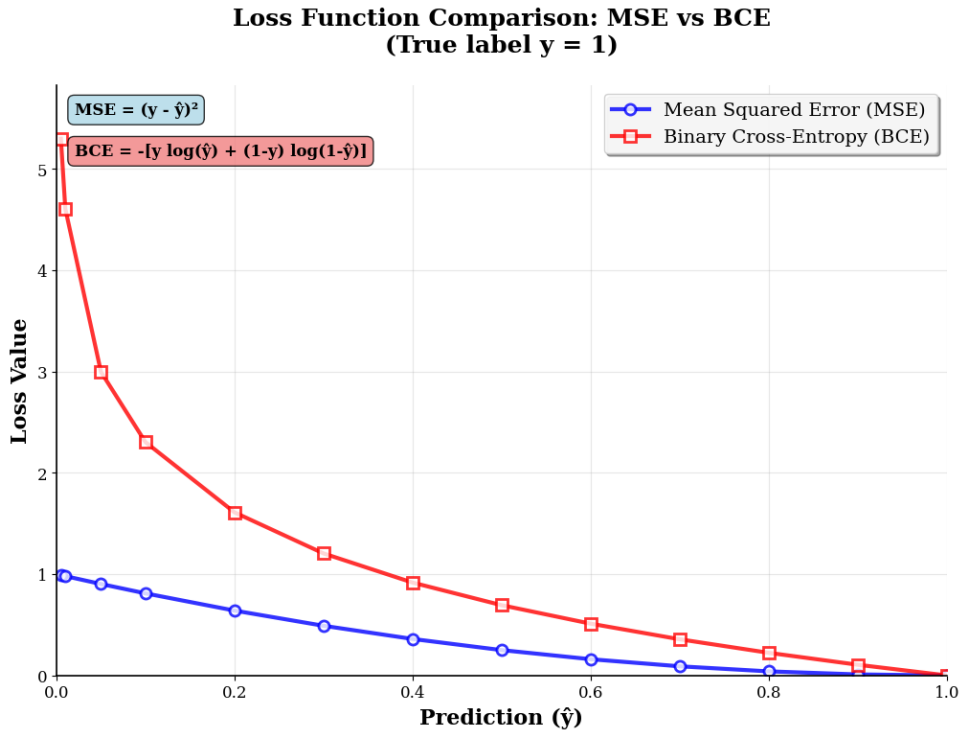


Figure 3: MSE vs BCE Loss Functions Comparison

Looking at the computed loss values for a true label $y = 1$, we can see the difference between MSE and BCE. MSE decreases gradually as the predicted value approaches 1, while BCE drops sharply, especially when predictions are close to the true label. Conversely, BCE assigns very large penalties for predictions near 0, which is desirable for classification tasks because it discourages confident wrong predictions. MSE, on the other hand, treats all deviations more uniformly, making it better suited for regression problems.

3 Data Preprocessing and Feature Scaling

3.1 Feature Generation

Using the provided code, two distinct features are generated:

My Index Number: 220054N

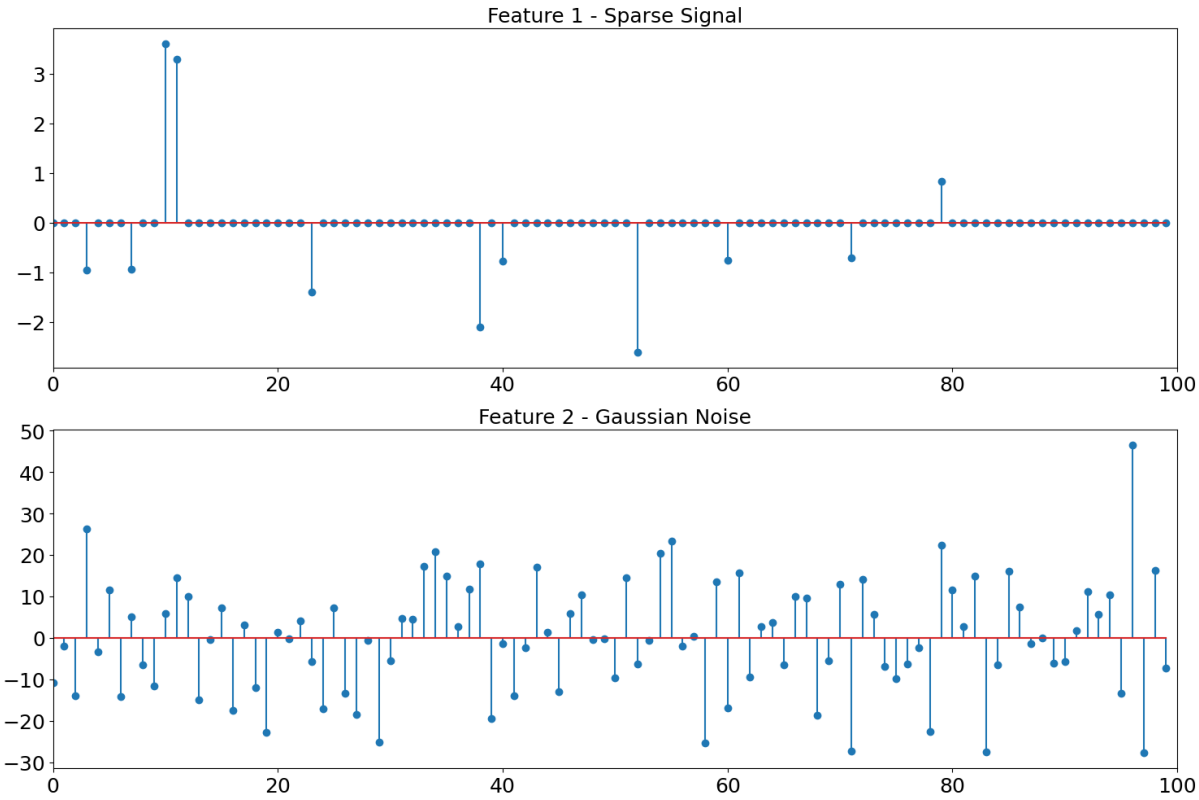


Figure 4: Feature 1: Sparse Signal

Table 3: Statistical Properties of Generated Features

Property	Feature 1 (Sparse)	Feature 2 (Gaussian)
Mean	-0.0254	-0.0100
Standard Deviation	0.6416	13.5547
Minimum	-2.6092	-27.7738
Maximum	3.6000	46.5349
Non-zero Elements	11/100	100/100

3.2 Scaling

1. **Standard Scaling:** $x_{scaled} = \frac{x-\mu}{\sigma}$
2. **Min-Max Scaling:** $x_{scaled} = \frac{x-x_{min}}{x_{max}-x_{min}}$
3. **Max-Absolute Scaling:** $x_{scaled} = \frac{x}{|x|_{max}}$

Implementation Code:

```python
f1 = sparse_signal.reshape(-1,1)
f2 = epsilon.reshape(-1,1)

scalers = {
    "Standard": StandardScaler(),
    "MinMax": MinMaxScaler(),
    "MaxAbs": MaxAbsScaler()
}

fig, axes = plt.subplots(2, 3, figsize=(15, 6))  # 2 rows, 3 columns

# Feature 1 (row 0)
for col, (name, scaler) in enumerate(scalers.items()):
    scaled_f1 = scaler.fit_transform(f1)
    axes[0, col].stem(scaled_f1)
    axes[0, col].set_title(f"Feature 1 - {name}")

# Feature 2 (row 1)
for col, (name, scaler) in enumerate(scalers.items()):
    scaled_f2 = scaler.fit_transform(f2)
    axes[1, col].stem(scaled_f2)
    axes[1, col].set_title(f"Feature 2 - {name}")

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

Listing 7: Scaling Methods Implementation

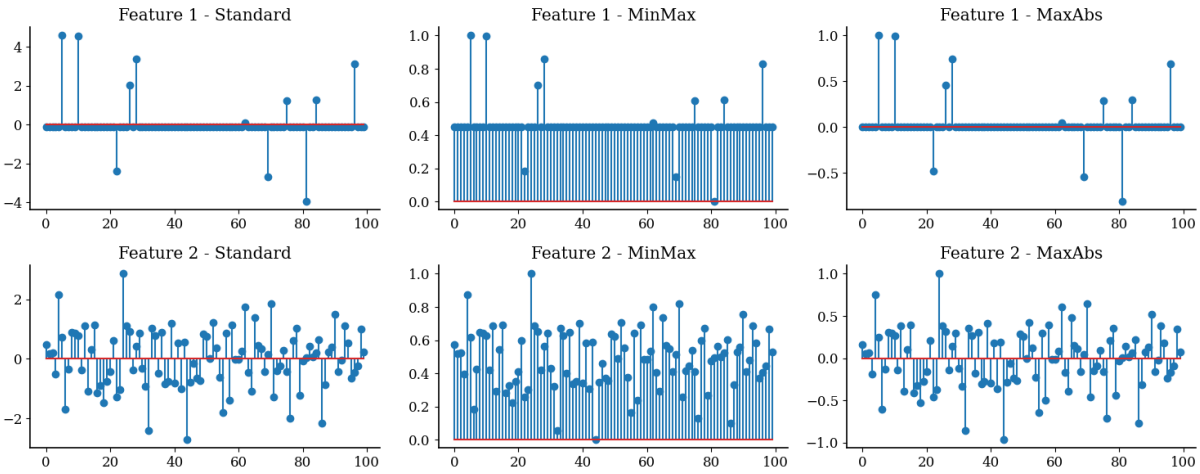


Figure 5: Comparison of different scaling methods on both features

Table 4: Statistical Properties After Scaling

2*Method	Feature 1 (Sparse)			Feature 2 (Gaussian)		
	Mean	Std	Sparsity	Mean	Std	Range
Original	-0.0254	0.6416	89%	-0.0100	13.5547	74.3087
Standard	0.0000	1.0000	0%	0.0000	1.0000	5.4822
Min-Max	0.4161	0.1033	0%	0.3736	0.1824	1.0000
Max-Abs	-0.0071	0.1782	89%	-0.0002	0.2913	1.5968

3.2.1 Selected Scaling Methods

Feature 1: Max-Absolute Scaling

- Preserves sparsity by keeping exact zeros unchanged.
- Keeps the original structure of the signal, since it only rescales values based on the maximum absolute value.
- Maps all values into the range $[-1, 1]$ without shifting the mean, which is important for sparse data.
- Simple and efficient, as it only requires dividing by the maximum absolute value.

Feature 2: Standard Scaling

- Normalizes the feature to have zero mean and unit variance, which matches the properties of Gaussian data.
- Ensures the distribution becomes standard normal, which is ideal for many statistical and machine learning models.