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

Assignment 01: Learning from data and related challenges and linear models for regression

By:

220097X De Alwis W.M.R.

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

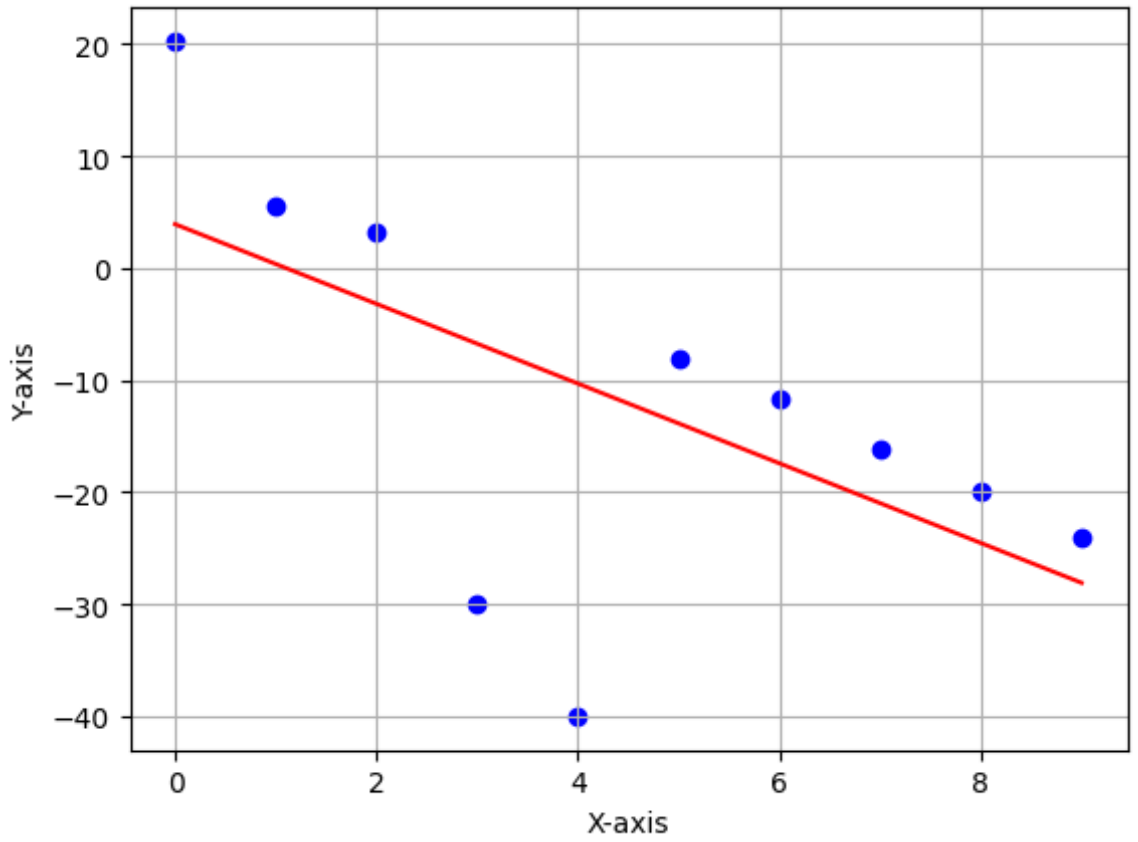
Date
21/08/2025

1 Linear Regression impact on outliers

1.1 Linear Regression Model

The Linear Regression model is plotted for the given dataset uding the code given below.

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression

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])


model = LinearRegression()
model.fit(x, y)
y_pred = model.predict(x)


plt.scatter(x, y, color='blue', label='Data Points')
plt.plot(x, y_pred, color='red', label='Linear Regression Line')
plt.xlabel('X-axis')
plt.ylabel('Y-axis')
plt.grid(True)
```



Task 4

The Loss function for different beta values is calculated using the following code.

```
def loss(y_i, y_ihat, beta):
    N = 10
    return (1/N) * np.sum(((y_i - y_ihat)**2) / (((y_i - y_ihat)**2)+ (beta)
        **2 ))

y_hat_model1 = -4 * x.flatten() + 12
y_hat_model2 = -3.55 * x.flatten() + 3.91

L1_m1 = loss(y, y_hat_model1, beta=1)
L1_m2 = loss(y, y_hat_model2, beta=1)

L2_m1= loss(y, y_hat_model1, beta=1e-6)
L2_m2 = loss(y, y_hat_model2, beta=1e-6)

L3_m1 = loss(y, y_hat_model1, beta=1e3)
L3_m2 = loss(y, y_hat_model2, beta=1e3)

# Create headers
print(f"{'   value':<10} {'Loss (Model 1)':<20} {'Loss (Model 2)':<20}")
print("-" * 50)

# Print rows
print(f"{'   = 1':<10} {L1_m1:<20} {L1_m2:<20}")
print(f"{'   = 1e-6':<10} {L2_m1:<20} {L2_m2:<20}")
print(f"{'   = 1e3':<10} {L3_m1:<20} {L3_m2:<20}")
```

Following results were obtained.

β value	Loss (Model 1)	Loss (Model 2)
$\beta = 1$	0.435416262490386	0.9728470518681676
$\beta = 1e^{-6}$	0.9999999998258207	0.9999999999999718
$\beta = 1e^{3}$	0.0002268287498440988	0.00018824684654645654

Table 1: Loss values for different β values across two models.

Task 5

The suitable β value to mitigate the impact of the outliers: $\beta = 1$

Justification:

The most appropriate choice is $\beta = 1$, as it achieves a good balance in handling outliers. When $\beta = 1e^{-6}$, the loss tends toward 1 for nearly all data points, making it impossible for the estimator to separate normal samples from outliers. On the other hand, with $\beta = 1e^{3}$, the loss collapses close to zero for all residuals, eliminating any meaningful penalization. In contrast, $\beta = 1$ preserves a quadratic penalty for small residuals (representing normal data), while capping the influence of large residuals at 1. This ensures that outliers are mitigated without disregarding the actual data distribution.

Task 6

Most suitable model when $\beta = 1$: Model 1

```python
if L1_m1 < L1_m2:
    print("Model 1 is better as it has a lower loss")

else:
    print("Model 2 is better as it has a lower loss")
```

Justification: Model 1 demonstrates a considerably lower robust loss value (0.435416) compared to Model 2, indicating a superior fit to the dataset when the effect of outliers is moderated with $\beta = 1$. This suggests that, although Model 2 may have been obtained using standard linear regression across the entire dataset, the parameters of Model 1 are more resilient to the presence of outliers. The robust evaluation highlights that Model 1 captures the underlying linear relationship more effectively, without being disproportionately influenced by extreme values, making it the more reliable choice for accurate predictions.

Task 7

How does this robust estimator reduce the impact of the outliers? The robust estimator mitigates the influence of outliers by employing a loss function that limits the contribution of individual data points to the total loss. For normal points, the squared residual $(y_i - \hat{y}_i)^2$ is small compared to the hyperparameter β^2, so the fraction $(y_i - \hat{y}_i)^2/((y_i - \hat{y}_i)^2 + \beta^2)$ behaves similarly to standard quadratic loss, allowing these points to influence the model normally. For outliers, where $(y_i - \hat{y}_i)^2$ is very large, the fraction approaches 1, effectively limiting the maximum contribution of outliers and preventing them from disproportionately skewing the estimated parameters. The hyperparameter β controls the sensitivity of this capping: a small β may cap even small residuals too early, reducing their influence, while a large β makes the estimator behave more like ordinary least squares, allowing outliers to dominate. By carefully choosing β, the estimator balances accurate fitting for normal data while mitigating the effect of outliers, ensuring that the model is robust without ignoring valid variations in the data.

Task 8

Alternative Loss Function: Huber Loss

The Huber loss is a robust loss function that combines the properties of Mean Squared Error (MSE) and Mean Absolute Error (MAE). For small residuals, it behaves quadratically, similar to MSE, which allows the model to remain sensitive to minor errors. For large residuals, it transitions to a linear behavior, like MAE, which reduces the influence of outliers on the overall loss. The parameter δ determines the threshold at which this transition occurs, controlling the balance between quadratic and linear regions. This characteristic makes the Huber loss smooth and suitable for optimization while providing robustness against outliers.

```python
def print_huber_formula():
    formula = """
    Huber Loss L  (r) =
        {  0.5 * r^2                if |r|
        {     * (|r| - 0.5 *   )      if |r| >
    """
    print(formula)

print_huber_formula()
```

Huber Formula:

$$\rho_\delta(r_i) = \begin{cases} \frac{1}{2} r_i^2, & \text{if } |r_i| \leq \delta, \\ \delta |r_i| - \frac{1}{2}\delta^2, & \text{if } |r_i| > \delta, \end{cases}$$

where $r_i = y_i - \hat{y}_i$ is the residual, and δ is a threshold parameter.

2 Loss function

Below given code was used to calculate the values for MSE and BCE.

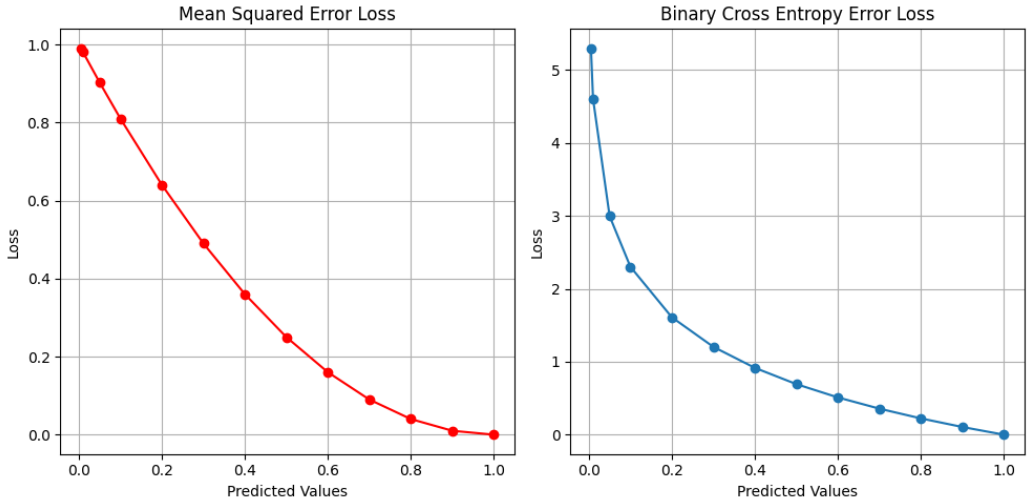
```
y = 1
y_hat = np.array([0.005, 0.01, 0.05, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8,
    0.9, 1.0])

mse = (y - y_hat) ** 2

y_pred_clipped = np.clip(y_hat, 1e-7, 1 - 1e-7)
bce = -1 * ((y * np.log(y_pred_clipped)) + ((1 - y) * np.log(1 -
   y_pred_clipped)))

print("True y=1    | Prediction    | MSE       | BCE")
for yp, m, b in zip(y_hat, mse, bce):
    print(f"   1   |     {yp:.3f}         | {m:.6f}   | {b:.6f}")
```

True y	Prediction	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

Task 2

For **Application 1**, where the dependent variable is continuous and a Linear Regression model is used, the Mean Squared Error (MSE) is the most suitable loss function. MSE directly measures the squared difference between the predicted and actual values, ensuring that the model minimizes large deviations and produces accurate predictions for continuous outcomes.

On the other hand, for **Application 2**, where the dependent variable is binary ($y \in \{0, 1\}$) and a Logistic Regression model is applied, the Binary Cross-Entropy (BCE) loss



is more appropriate. BCE evaluates the difference between the predicted probability and the actual binary class, heavily penalizing confident but incorrect predictions. This makes it highly effective for classification problems, as it aligns with the probabilistic nature of logistic regression outputs.

Thus, MSE is preferred for continuous prediction tasks, while BCE is best suited for binary classification.

3 Data Pre-processing

3.1 Feature Values Generation

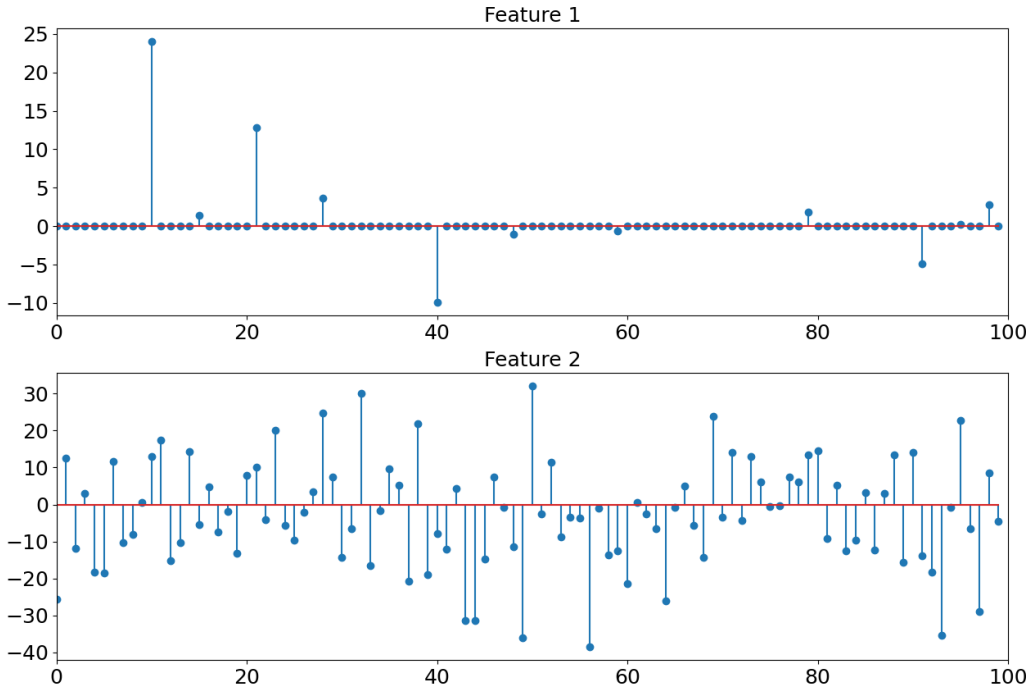
The feature values of two features were generated and plotted using the given code.

- **Feature 1:** Sparse signal with mostly zero values and few non-zero elements.
- **Feature 2:** Gaussian noise with mean 0 and standard deviation 15.

3.2 Choosing suitable scaling methods

Here I applied 3 different scaling methods and selected most appropriate one for each feature. The foolowing code was used to implement the 3 different scaling methods.

```
# Generate signals
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 = 42  # Replace with your actual index number
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)

# Scaling methods
def standard_scaling(data):
    mean = np.mean(data)
    std = np.std(data)
    return (data - mean) / std

def min_max_scaling(data):
    min_val = np.min(data)
    max_val = np.max(data)
    return (data - min_val) / (max_val - min_val)

def max_abs_scaling(data):
    max_val = np.max(np.abs(data))
    return data / max_val

# Apply scaling to both features
feature1_standard = standard_scaling(sparse_signal)
feature1_minmax = min_max_scaling(sparse_signal)
feature1_maxabs = max_abs_scaling(sparse_signal)

feature2_standard = standard_scaling(epsilon)
feature2_minmax = min_max_scaling(epsilon)
feature2_maxabs = max_abs_scaling(epsilon)

# Plotting
plt.figure(figsize=(15, 20))

plt.subplot(3, 2, 1)
```

```
plt.title("Feature 1 - Standard Scaling", fontsize=18)
plt.stem(feature1_standard)
plt.xticks(fontsize=18)
plt.yticks(fontsize=18)

plt.subplot(3, 2, 2)
plt.title("Feature 2 - Standard Scaling", fontsize=18)
plt.stem(feature2_standard)
plt.xticks(fontsize=18)
plt.yticks(fontsize=18)

plt.subplot(3, 2, 3)
plt.title("Feature 1 - Min-Max Scaling", fontsize=18)
plt.stem(feature1_minmax)
plt.xticks(fontsize=18)
plt.yticks(fontsize=18)

plt.subplot(3, 2, 4)
plt.title("Feature 2 - Min-Max Scaling", fontsize=18)
plt.stem(feature2_minmax)
plt.xticks(fontsize=18)
plt.yticks(fontsize=18)

plt.subplot(3, 2, 5)
plt.title("Feature 1 - Max-Abs Scaling", fontsize=18)
plt.stem(feature1_maxabs)
plt.xticks(fontsize=18)
plt.yticks(fontsize=18)

plt.subplot(3, 2, 6)
plt.title("Feature 2 - Max-Abs Scaling", fontsize=18)
plt.stem(feature2_maxabs)
plt.xticks(fontsize=18)
plt.yticks(fontsize=18)

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

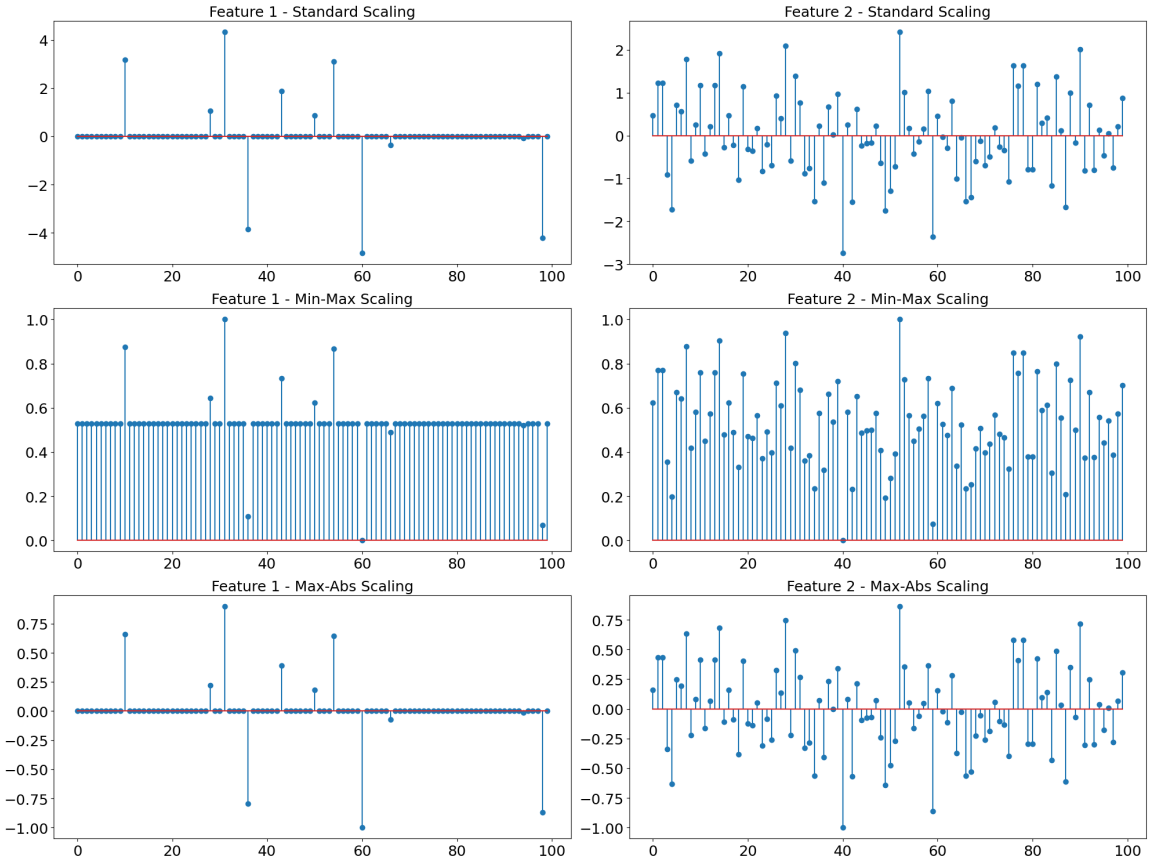


Figure 1: Comparison of different scaling methods for the 2 features

Selected Scaling Methods

Feature 1 – MaxAbsScaler

Feature 1 represents a sparse signal, where most values are exactly zero and only a few non-zero spikes occur. When scaling such data, two key requirements must be met:

1. Zeros should remain unchanged.
2. The magnitude and sign of the spikes must be preserved.

The `MaxAbsScaler` addresses these requirements by dividing each value by the maximum absolute value within the feature. This transformation ensures that all values lie within the range $[-1, 1]$, while keeping zeros intact.

Unlike the `StandardScaler`, it does not shift the mean, which would introduce small non-zero values and destroy sparsity. Similarly, unlike the `MinMaxScaler`, it does not restrict values to $[0, 1]$, thereby preserving the sign and avoiding distortion of the feature's structure.

Due to these properties, `MaxAbsScaler` is the most appropriate scaling method for Feature 1, as it normalizes the feature range without altering sparsity or disrupting the relative proportions of spike amplitudes.

Feature 2 – StandardScaler

Feature 2 consists of Gaussian-like random noise with a mean near zero and a relatively large standard deviation. For this feature, it is essential to preserve the overall shape of the distribution while normalizing its scale, ensuring that it does not overshadow other features in magnitude.

The `StandardScaler` standardizes the data by centering it at zero mean and scaling it to unit variance. This transformation maintains the Gaussian distribution and preserves the relative distances between points.

In contrast to the `MinMaxScaler`, it does not compress values into the $[0, 1]$ range, and unlike the `MaxAbsScaler`, it removes the mean offset. These characteristics make `StandardScaler` ideal for features with Gaussian-like distributions.

Justification: MaxAbs scaling is the most appropriate choice for sparse data because it maintains the original sparsity pattern. On the other hand, Standard scaling is well-suited for features with a Gaussian-like distribution, such as Feature 2. Applying alternative scaling methods could alter the intrinsic characteristics of each feature, for instance, MinMax scaling would shrink the dynamic range of the sparse signal, and Standard scaling would unnecessarily change the zero values in the sparse feature.