Experiment – 6

Feature Scaling: Apply different feature scaling methods like Min-Max scaling, Z-score normalization, and robust scaling on model performance.

Date Set: *We will use the data in a csv file, namely* ***data.csv****, which is the data set on which we will implement procedure to handle missing data. This data set belongs to a retail company that collected some data from their customers, whether or not they purchased a certain product. Here, each of the rows correspond to different customers. For each of these customers the company gathered their country, their age, salary, and whether or not they purchased their product.*

Objective:

Perform different feature scaling methods on model performance.

Tasks:

1. Implement Min-Max scaling/ Z-score normalization/ robust scaling.
2. Apply each scaling method to relevant numerical features.
3. Assess the impact of feature scaling on model performance.

Feature scaling is a crucial step in data preprocessing, especially when dealing with machine learning algorithms that are sensitive to the scale of the input features. It involves transforming the numerical features of a dataset into a standardized range.
There are several common techniques for feature scaling:

1. **Min-Max Scaling (Normalization):** This method scales the features to a fixed range, typically between 0 and 1. It is calculated as:

$$X_{scaled} = \frac{X - X_{min}}{X_{max} - X_{min}}$$

where X is the original feature value, X_{min} is the minimum value of the feature, and X_{max} is the maximum value of the feature.

2. **Z-score Normalization (Standardization):** This method scales the features so that they have a mean of 0 and a standard deviation* of 1. It is calculated as:

$$X_{scaled} = \frac{X - \mu}{\delta}$$

where X is the original feature value, μ is the mean of the feature, and δ is the standard deviation of the feature.

3. **Robust Scaling:** This method scales the features using statistics that are robust to outliers. It is similar to Min-Max Scaling but uses the interquartile range (IQR) instead of the range of the data. It is calculated as:

4. $$X_{scaled} = \frac{X - Q_1}{Q_3 - Q_1}$$

where X is the original feature value, Q_1 is the first quartile, and Q_3 is the third quartile.

Feature scaling ensures that all features contribute equally to the analysis and prevents features with larger scales from dominating the model training process. The choice of scaling method depends on the distribution of the data and the requirements of the machine learning algorithm being used.

Standard deviation is a measure of the dispersion or spread of a set of values in a dataset. It quantifies how much the values of a dataset deviate from the mean (average). A low standard deviation indicates that the data points tend to be close to the mean, while a high standard deviation indicates that the data points are spread out over a wider range of values.

Mathematically, the standard deviation (δ) of a dataset is calculated as the square root of the variance. The variance (δ^2) is the average of the squared differences between each data point and the mean.
Here's the formula for calculating the standard deviation:

$$\delta = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i - \mu)^2}$$

Where:

- δ is the standard deviation,
- N is the number of data points,
- x_i is each individual data point in the dataset,
- μ is the mean of the dataset.

In simpler terms, the standard deviation measures how much variation or dispersion there is from the average. If the standard deviation is small, it means that most data points are close to the mean. If the standard deviation is large, it means that the data points are spread out over a wider range of values.

Python Program:

Importing the libraries

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

Importing the dataset

```
dataset = pd.read_csv(r'C:\Users\...\GHRCEM\Dataset\Data.csv')
X = dataset.iloc[:, :-1].values
Y = dataset.iloc[:, -1].values
```

```
print(dataset.values)
# prints the entire dataset
```

Output:

```
[['France' 44.0 72000.0 'No']
 ['Spain' 27.0 48000.0 'Yes']
 ['Germany' 30.0 54000.0 'No']
```

```
 ['Spain' 38.0 61000.0 'No']
 ['Germany' 40.0 nan 'Yes']
 ['France' 35.0 58000.0 'Yes']
 ['Spain' nan 52000.0 'No']
 ['France' 48.0 79000.0 'Yes']
 ['Germany' 50.0 83000.0 'No']
 ['France' 37.0 67000.0 'Yes']]
```

```
print(X)
```

Output:

```
[['France' 44.0 72000.0]
 ['Spain' 27.0 48000.0]
 ['Germany' 30.0 54000.0]
 ['Spain' 38.0 61000.0]
 ['Germany' 40.0 nan]
 ['France' 35.0 58000.0]
 ['Spain' nan 52000.0]
 ['France' 48.0 79000.0]
 ['Germany' 50.0 83000.0]
 ['France' 37.0 67000.0]]
```

```
print(Y)
```

Output:

```
['No' 'Yes' 'No' 'No' 'Yes' 'Yes' 'No' 'Yes' 'No' 'Yes']
```

Taking care of missing data

```
from sklearn.impute import SimpleImputer as si
imputer = si(missing_values = np.nan, strategy = 'mean')
imputer.fit(X[:, 1:3])
X[:, 1:3] = imputer.transform(X[:, 1:3])
```

```
print(X)
```

Output:

```
[['France' 44.0 72000.0]
 ['Spain' 27.0 48000.0]
 ['Germany' 30.0 54000.0]
 ['Spain' 38.0 61000.0]
 ['Germany' 40.0 63777.77777777778]
 ['France' 35.0 58000.0]
 ['Spain' 38.77777777777778 52000.0]
 ['France' 48.0 79000.0]
 ['Germany' 50.0 83000.0]
 ['France' 37.0 67000.0]]
```

Encoding Categorical Data

Encoding the Independent Variable

```
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import OneHotEncoder
```

```
ct = ColumnTransformer(transformers = [('encoder', OneHotEncoder(), [0])], remainder = 'passthrough')
X = np.array(ct.fit_transform(X))
```

```
print(X)
```

Output:

```
[[1.0 0.0 0.0 44.0 72000.0]
 [0.0 0.0 1.0 27.0 48000.0]
 [0.0 1.0 0.0 30.0 54000.0]
 [0.0 0.0 1.0 38.0 61000.0]
 [0.0 1.0 0.0 40.0 63777.77777777778]
 [1.0 0.0 0.0 35.0 58000.0]
 [0.0 0.0 1.0 38.77777777777778 52000.0]
 [1.0 0.0 0.0 48.0 79000.0]
 [0.0 1.0 0.0 50.0 83000.0]
 [1.0 0.0 0.0 37.0 67000.0]]
```

Encoding the Dependent Variable

```
from sklearn.preprocessing import LabelEncoder
```

```
le = LabelEncoder()
Y = le.fit_transform(Y)
```

```
print(Y)
```

Output:

```
[0 1 0 0 1 1 0 1 0 1]
```

Splitting the dataset into the Training set and Test set

```
from sklearn.model_selection import train_test_split
X_train, X_test, Y_train, Y_test = train_test_split (X, Y, test_size = 0.2, random_state = 1)
```

Note:

This code snippet is using the **train_test_split** function from the **sklearn.model_selection** module, which is commonly used to split a dataset into training and testing sets for machine learning tasks.

Here's a detailed explanation of each component of the code:

1. **from sklearn.model_selection import train_test_split**: This line imports the **train_test_split** function from the **sklearn.model_selection** module. **train_test_split** is a utility function in scikit-learn that splits arrays or matrices into random train and test subsets.
2. **X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.2, random_state=1)**: This line actually performs the train-test split operation.
    - **X** is the feature matrix (or input data) of the dataset.
    - **Y** is the target variable (or output data) of the dataset.

- **test_size=0.2** specifies that 20% of the data should be used for testing, and the remaining 80% will be used for training. This parameter determines the proportion of the dataset that will be allocated to the testing set.
- **random_state=1** sets the random seed for reproducibility. When **random_state** is set to a specific value (in this case, 1), the data will be split the same way every time the code is run. This ensures consistent results across different runs of the code. [**P. S.** The **"random seed for reproducibility"** ensures that the random splitting of the data into training and testing sets is done in a consistent way each time the code is run. This means that if you use the same random seed value, you'll get the same split of data into training and testing sets every time you run the code. It helps to ensure that your results are consistent and reproducible across different runs of the code.]

3. **X_train, X_test, Y_train, Y_test**: These are the variables where the training and testing data will be stored after the split operation.
    - **X_train** contains the features of the training set.
    - **X_test** contains the features of the testing set.
    - **Y_train** contains the target values (labels) corresponding to the training set.
    - **Y_test** contains the target values (labels) corresponding to the testing set.

After executing this code, you will have four sets of data: **X_train** (features for training), **X_test** (features for testing), **Y_train** (target values for training), and **Y_test** (target values for testing). These datasets can then be used to train machine learning models on the training data and evaluate their performance on the testing data.

```
print(X_train)
```

Output:

```
[[0.0 0.0 1.0 38.77777777777778 52000.0]
 [0.0 1.0 0.0 40.0 63777.77777777778]
 [1.0 0.0 0.0 44.0 72000.0]
 [0.0 0.0 1.0 38.0 61000.0]
 [0.0 0.0 1.0 27.0 48000.0]
 [1.0 0.0 0.0 48.0 79000.0]
 [0.0 1.0 0.0 50.0 83000.0]
 [1.0 0.0 0.0 35.0 58000.0]]
```

```
print(X_test)
```

Output:

```
[[0.0 1.0 0.0 30.0 54000.0]
 [1.0 0.0 0.0 37.0 67000.0]]
```

```
print(Y_train)
```

Output:

```
[0 1 0 0 1 1 0 1]
```

```
print(Y_test)
```

Output:

```
[0 1]
```

Feature Scaling

Feature scaling should be performed after splitting the dataset into the training set and test set. Here's why:

1. **Prevent Data Leakage:** Performing feature scaling after splitting ensures that information from the test set does not leak into the training set. This helps maintain the integrity of the evaluation process by ensuring that the model only learns from the training data and is evaluated on truly unseen data.
2. **Replicate Real-world Scenarios:** In real-world scenarios, models are trained on historical data and applied to unseen future data. By scaling features after splitting, we replicate this scenario more accurately, as the model learns from past data and applies what it has learned to new, unseen data.
3. **Avoid Biased Performance Estimates:** Scaling features before splitting can lead to biased performance estimates because the statistics used for scaling (e.g., mean and standard deviation) may be influenced by the entire dataset, including the test set. This can result in overly optimistic performance estimates on the test set.

```
from sklearn.preprocessing import StandardScaler
sc = StandardScaler()
X_train[:, 3:] = sc.fit_transform(X_train[:, 3:])
X_test[:, 3:] = sc.transform(X_test[:, 3:])
```

Note:

1. **from sklearn.preprocessing import StandardScaler**: This line imports the **StandardScaler** class from the **sklearn.preprocessing** module. **StandardScaler** is a preprocessing transformer in scikit-learn that standardizes features by removing the mean and scaling them to unit variance.
2. **sc = StandardScaler()**: This line creates an instance of the **StandardScaler** class, which will be used to scale the features.
3. **X_train[:, 3:] = sc.fit_transform(X_train[:, 3:])**: This line scales the features in the training set using the **fit_transform** method of the **StandardScaler** instance.
    - **X_train[:, 3:]** selects all rows of the **X_train** array and all columns starting from index 3 (inclusive). This typically implies that we are selecting a subset of features to scale.
    - **sc.fit_transform(X_train[:, 3:])** applies the scaling transformation to the selected subset of features. The **fit_transform** method first computes the mean and standard deviation of each feature in the training set and then scales the features using these statistics. It returns the scaled features.
4. **X_test[:, 3:] = sc.transform(X_test[:, 3:])**: This line scales the features in the testing set using the **transform** method of the **StandardScaler** instance.
    - **X_test[:, 3:]** selects all rows of the **X_test** array and all columns starting from index 3 (inclusive), similar to the training set.
    - **sc.transform(X_test[:, 3:])** applies the same scaling transformation that was computed using the training set to the testing set. It uses the mean and standard deviation computed during the training set's **fit_transform** step to ensure consistency in scaling between the training and testing sets.

Overall, this code standardizes (scales) the features in both the training and testing sets using **StandardScaler**. It ensures that the features have a mean of 0 and a standard deviation of 1, which can be beneficial for many machine learning algorithms.

```
print(X_train)
```

Output:

```
[[0.0 0.0 1.0 -0.19159184384578545 -1.0781259408412425]
 [0.0 1.0 0.0 -0.014117293757057777 -0.07013167641635372]
 [1.0 0.0 0.0 0.566708506533324 0.633562432710455]
 [0.0 0.0 1.0 -0.30453019390224867 -0.30786617274297867]
 [0.0 0.0 1.0 -1.9018011447007988 -1.420463615551582]
 [1.0 0.0 0.0 1.1475343068237058 1.232653363453549]
 [0.0 1.0 0.0 1.4379472069688968 1.5749910381638885]
 [1.0 0.0 0.0 -0.7401495441200351 -0.5646194287757332]]
```

```
print(X_test)
```

Output:

```
[[0.0 1.0 0.0 -1.4661817944830124 -0.9069571034860727]
 [1.0 0.0 0.0 -0.44973664397484414 0.2056403393225306]]
```