Experiment – 5

Data Preprocessing: Explore different strategies for dealing with missing data, such as imputation techniques (mean, median, mode), or advanced methods like k-nearest neighbors imputation. Convert categorical variables into numerical representations using techniques like one-hot encoding, label encoding, or target encoding. 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:

Explore strategies for handling missing data in a dataset.

Tasks:

1. Import the libraries.
2. Import the data set.
3. Identify missing values in the dataset.
4. Implement different imputation techniques (mean, median, mode).
5. Explore different encoding techniques (one-hot encoding/ label encoding/ target encoding).
6. Apply each encoding technique to categorical variables in the dataset.
7. Implement Min-Max scaling/ Z-score normalization/ robust scaling.
8. Apply each scaling method to relevant numerical features.

Python Program:

Importing the libraries

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

Note:

A. **import numpy as np**:
   a. **numpy** is a powerful library for numerical computing in Python.
   b. **import numpy as np** imports the **numpy** library and gives it an alias **np** for convenience.
   c. The **np** alias is commonly used to reference functions and objects from the **numpy** library. For example, **np.array()** creates a NumPy array.

B. **import matplotlib.pyplot as plt**:

a. **matplotlib** is a popular library for creating static, animated, and interactive visualizations in Python.
b. **import matplotlib.pyplot as plt** imports the **pyplot** module from **matplotlib** and gives it an alias **plt** for convenience.
c. The **plt** alias is commonly used to create plots and charts. For example, **plt.plot()** is used to create line plots.

C. **import pandas as pd**:
a. **pandas** is a data manipulation and analysis library for Python.
b. **import pandas as pd** imports the **pandas** library and gives it an alias **pd** for convenience.
c. The **pd** alias is commonly used to reference functions and objects from the **pandas** library. For example, **pd.DataFrame()** is used to create a DataFrame.

Importing the dataset

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

Note:
Here we have created two new entities: X is the matrix of features (also called independent variable), and Y is the dependent variable vector.

A. **X = dataset.iloc[:, :-1].values**:
a. **dataset**: This presumably represents a pandas DataFrame containing your dataset.
b. **iloc (i. e. integer-location based indexing)**: This is a method in pandas used for integer-location based indexing. It allows you to select data by row and column index.
c. **[:, :-1]**: This selects all rows (**:**) and all columns except the last one (**:-1**). In other words, it selects all features (independent variables) of the dataset. Remember -1 is the index of the last column of pandas DataFrame.
d. **.values**: This converts the selected data (features) into a NumPy array. Machine learning algorithms often work with NumPy arrays instead of pandas DataFrames.

So, **X** would contain the feature values of your dataset.

B. **Y = dataset.iloc[:, -1].values**:
a. **[:, -1]**: It selects all rows (**:**) and only the last column (**-1**). This is typically the target variable (dependent variable) in a machine learning problem.
b. **.values**: Converts the selected column into a NumPy array.

So, **Y** would contain the target variable values.

C. The **r** prefix before the string `(r'C:\Users\...\GHRCEM\Dataset\Data.csv')` makes it a raw string, so backslashes are treated as literal characters and won't cause the Unicode escape error.

```
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])
```

Note:

This code uses scikit-learn's **SimpleImputer** to fill in missing values in specific columns of the dataset with the mean value of each respective column. This is a common technique in data preprocessing when dealing with missing data before applying machine learning algorithms. Let's break down each line:

A. **from sklearn.impute import SimpleImputer as si**:
   a. Imports the **SimpleImputer** class from scikit-learn's **impute** module.
   b. The **as si** part gives the **SimpleImputer** class an alias **si** for convenience, making it shorter to reference in the code.

B. **imputer = si(missing_values = np.nan, strategy = 'mean')**:
   a. Creates an instance of the **SimpleImputer** class with specific parameters.
   b. **missing_values = np.nan**: Specifies that missing values in the dataset are represented by **np.nan** (NumPy's representation of missing or undefined values).
   c. **strategy = 'mean'**: Specifies the imputation strategy. In this case, it uses the mean value of the non-missing values in each column to fill in the missing values.

C. **imputer.fit(X[:, 1:3])**:
   a. Fits the imputer on the specified subset of the dataset.
   b. **X[:, 1:3]** selects all rows (**:**) and columns 1 to 2 (index 1 is the second column, and index 3 is exclusive), which typically means it's selecting specific features with missing values for imputation.
   c. The **fit** method computes the mean value for each selected column.

D. **X[:, 1:3] = imputer.transform(X[:, 1:3])**:
   a. Transforms the selected subset of the dataset by replacing missing values with the computed means.
   b. **transform** method applies the imputation strategy to fill in missing values in the specified columns.
   c. The updated values are then assigned back to the original dataset.

```
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
```

Note:

- **ColumnTransformer**: This class allows you to selectively apply transformers (such as scaling, one-hot encoding, etc.) to different columns or subsets of columns in your dataset.
- **OneHotEncoder**: This transformer is used for converting categorical variables into a form that can be provided to ML algorithms to do a better job in prediction.

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

Note:

- Here, a **ColumnTransformer** object **ct** is created. While creating **ct** object, **ColumnTransformer** class takes 2 parameters: **transformers** and **remainder.**
- **transformers** parameter: It's a list of tuples. Each tuple contains three elements:
  - Name: A name to identify the transformation (e.g., '**encoder**').

    - Transformer: The transformer to be applied. In this case, it's **OneHotEncoder()**.
    - Columns: The indices of the columns to which the transformation should be applied. **[0]** indicates the first column in the dataset (column indexing starts from 0).
- **remainder** parameter: It specifies what to do with the columns not explicitly transformed. '**passthrough**' means that those columns will be left untouched.

```
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
```

Note:
This line imports the **LabelEncoder** class from the **sklearn.preprocessing** module. This class is used for converting categorical labels into numerical labels. It assigns a unique integer to each category.

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

Note:

- Here, a **LabelEncoder** object **le** is created. This object will be used to perform the label encoding. **LabelEncoder** is used when dealing with categorical target variables (labels) in a supervised learning task. It converts categorical labels into numerical format, which is essential for many machine learning algorithms to work properly.
- **fit_transform(Y)**: This method fits the **LabelEncoder** to the target variable **Y** and then transforms it. It returns the transformed labels. **fit_transform()** method fits the **LabelEncoder** to the target variable **Y** and then transforms it. The fitting process learns the mapping of categories to integers, and the transformation process applies this mapping to convert the categorical labels into numerical ones.
- **Y**: It represents the target variable, typically the output or dependent variable in a machine learning problem. After the transformation, **Y** contains numerical labels instead of categorical ones.

```
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]]
```