SENTIMENT ANALYSIS FOR MARKETING

TEAM MEMBER: MONISH.R
REG NUMBER:510521205025

INTRODUCTION

Data preprocessing is the first machine learning step in which we transform raw data obtained from various sources into a usable format to implement accurate machine learning models. In this article, we cover all the steps involved in the data preprocessing phase.

Mounting our Drive to Google Colab

In this article, we shall carry out our data preprocessing experiments on Google Colab. Therefore, we need to ensure our Google Drive is accessible from Google Colab. To ensure this, first, let's download our data to our computer from here.

Since we've successfully downloaded our data, let's now upload it to Google Drive through google.drive.com. Our data is uploaded to Google Drive. It is saved in the root directory.

To use this data, we need to give Google Colab access to Google Drive. So let's type and run the code below in Google Colab.

```
from google.colab import drive
drive.mount("/content/drive/")
```

Upon executing our code, it leads us to a Google Authentication stage.

Click on the URL link on the Google Colab interface and proceed to allow permissions until we reach the verification code stage.

We copy and paste the obtained authorization code in the box available on the Colab interface and click **Ctrl + Enter**. After this step, one may access files in Drive. Next, let's proceed with importing the required libraries.

To import these libraries, let's type and run the code below.

Step 1: Importing the libraries

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

Step 2: Import the dataset

Let's type and run the following code:

```
Dataset = pd.read_csv("/content/drive/MyDrive/Dataset.csv")

# importing an array of features
x = Dataset.iloc[:, :-1].values
# importing an array of dependent variable
y = Dataset.iloc[:, -1].values
```

We specified two variables, x for the features and y for the dependent variable. The features set, as declared in the code Dataset.iloc[:, :-1] consists of all rows and columns of our dataset except the last column. Similarly, the dependent variable y consists of all rows but only the last column as declared in the code Dataset.iloc[:, -1].values.

Let's have a look at our data by executing the code:

```
print(x) # returns an array of features
```

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) # viewing an array of the dependent variable.

Output

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

Our data is imported successfully in the form of an array of features x and a dependent variable y.

Step 3: Taking care of the missing data

Missing data is a common problem that faces the data collected through a survey. This problem occurs when a dataset has no value for a feature in an observation.

There are many reasons why data might be missing in a dataset. For instance, data collected through a survey may have missing data due to participants' failure to respond to some questions, not knowing the correct response, or being unwilling to answer. It may also be missing due to the error made during the data entry process.

Most machine learning models require data with a value for all features in each observation. In such models, missing data may lead to bias in the estimation of the parameters and also compromise the accuracy of the machine learning models.

As a result, we may end up drawing wrong conclusions about data. Therefore, missing data is harmful to machine learning models and requires appropriate handling.

There are several techniques we use to handle the missing data. They include:

Deleting the observation with the missing value(s)

This technique works well on big datasets with few missing values. For instance, deleting a row from a dataset with hundreds of observations cannot affect the information quality of the dataset. However, this technique is not suitable for a dataset reporting many missing values. Deleting many rows from a dataset leads to the loss of information.

To ensure no risk of losing crucial information, we need to make use of more appropriate techniques. The following technique involves the imputation of the missing data. Imputation means replacing the missing data with an estimated value.

Mean Imputation

Under this technique, we replace the missing value with the average of the variable in which it occurs. The advantage of this technique is that it preserves the mean and the sample size. However, this technique has some serious disadvantages.

Mean imputation underestimates the standard error, and it does not preserve the correlation among variables. The relationship among variables is an essential aspect of analysis as the study's general objective is to understand it better.

Mean imputation is thus not an appropriate solution for missing data unless the data is missing completely at random (missing data is completely unrelated to both the missing data and observed values in the dataset).

Hot Deck Imputation

In this technique, we replace the missing value of the observation with a randomly selected value from all the observations in the sample that has similar values on other variables.

Thus, this technique ensures that the imputing value is only selected from the possible interval where the actual value could probably fall, and is randomly selected rather than being determined, which is an essential aspect for a correct standard error.

Cold Deck Imputation

We replace the missing data using a value chosen from other variables with similar observation values in this technique. The difference between this technique and the Hot Deck imputation is that the selecting process of the imputing value is not randomized.

Regression Imputation

Regression imputation involves fitting a regression model on a feature with missing data and then using this regression model's predictions to replace the missing values in this feature. This technique preserves the relationships between features, and this grants it a significant advantage over simple imputation techniques such as mean and mode imputation.

Regression imputation is of two categories:

Deterministic regression imputation

Deterministic regression imputation imputes the missing data with the exact value predicted from the regression model. This technique doesn't consider the random variation around the regression line. Since the imputed values are exact, the correlation between the features and the dependent variables is overestimated.

Stochastic regression imputation

In stochastic regression imputation, we add a random variation (error term) to the predicted value, therefore, reproducing the correlation of X and Y more appropriately.

Now that we know the techniques to take care of the missing values, let's handle this problem in our dataset. We notice that our features set (x) has nan values in the Age and Salary columns.

We need to deal with this problem before we implement a machine learning model on our data. Since our dataset is small, we cannot eliminate a row reporting the missing value(s). Therefore, in our case, we shall make use of the mean imputation technique.

The code below solves this problem present in our dataset.

```
# Importing the class called SimpleImputer from impute model in sklearn
from sklearn.impute import SimpleImputer
# To replace the missing value we create below object of SimpleImputer class
imputa = SimpleImputer(missing_values = np.nan, strategy = 'mean')
''' Using the fit method, we apply the `imputa` object on the matrix of our feature x.
The `fit()` method identifies the missing values and computes the mean of such feature a missing value is present.
'''
imputa.fit(x[:, 1:3])
# Repalcing the missing value using transform method
x[:, 1:3] = imputa.transform(x[:, 1:3])
```

Upon executing the code, we obtain a matrix of features with the missing values replaced.

```
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]]
```

The missing values on the Age and Salary columns are replaced with their respective column means, i.e., 38.77777777777778 and 63777.77777777778, respectively.

Step 4: Encoding categorical data

During encoding, we transform text data into numeric data. Encoding categorical data involves changing data that fall into categories to numeric data.

The Country and the Purchased columns of our dataset contain data that fall into categories. Since machine learning models are based on a mathematical equation, which takes only numerical inputs, it is challenging to compute the correlation between the feature and the dependent variables. To ensure this does not happen, we need to convert the string entries in the dataset into numbers.

For our dataset, we shall encode France into 0, Spain into 1, and Germany into 2. However, our future machine learning model interprets the numerical order between 0 for France, 1 for Spain, and 2 for Germany do matter, which is not the case. To ensure this misinterpretation does not occur, we make use of one-hot encoding.

One-hot encoding converts our categorical Country column into three columns. It creates a unique binary vector for each country such that there is no numerical order between the country categories.

Let's see how One-hot encoding enables us to achieve this by executing the code below:

```
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))


# executing the cell we obtain:
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]]
```

From the output, the Country column has been transformed into 3 columns with each row representing only one encoded column where, France was encoded into a vector [1.0 0.0 0.0], Spain encoded into vector [0.0 0.0 1.0], and Germany encoded into vector [0.0 1.0 0.0] where they're all unique.

To encode our depended variable y, let's run the code below:

```
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]
```

Our dependent variable is encoded successfully into 0's and 1's.

Step 5: Splitting the dataset into the training and test sets

In machine learning, we split the dataset into a training set and a test set. The training set is the fraction of a dataset that we use to implement the model. On the other hand, the test set is the fraction of the dataset that we use to evaluate the performance our the model.

The test set is assumed to be unknown during the process of the model implementation.

We need to split our dataset into four subsets, x_train, x_test, y_train, and y_test. Let's look at the code to achieve this:

```
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)
```

Output

Let's print the output upon executing the code below.

```
print(x_train)
```

```
[[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)
```

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

```
print(y_train)
```

```
print(y_test)
```

```
[0 1]
```

Our dataset is successfully split. Our features set was divided into eight observations for the x_train and 2 for the x_test, which correspond (since we set our seed, random = 1) to the same splitting of the dependent variable y.

The selection is made randomly, and it's possible at any single execution to obtain different subsets other than the above output.

Step 6: Feature scaling

In most cases, we shall work with datasets whose features are not on the same scale. Some features often have tremendous values, and others have small values.

Suppose we implement our machine learning model on such datasets. In that case, features with tremendous values dominate those with small values, and the machine learning model treats those with small values as if they don't exist (their influence on the data is not be accounted for). To ensure this is not the case, we need to scale our features on the same range, i.e., within the interval of -3 and 3.

Therefore, we shall only scale the Age and Salary columns of our x_train and x_test into this interval. The code below enables us to achieve this.

```
from sklearn.preprocessing import StandardScaler
sc = StandardScaler()
# we only aply the feature scaling on the features other than dummy variables.
x_train[:, 3:] = sc.fit_transform(x_train[:, 3:])
x_test[:, 3:] = sc.fit_transform(x_test[:, 3:])
```

Output

print(x_train)

```
[[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)
```

```
[[0.0 1.0 0.0 -1.0 -1.0]
 [1.0 0.0 0.0 1.0 1.0]]
```

Notice that in the x_train and x_test, we only scaled the Age and Salary columns and not the dummy variables. This is because scaling the dummy variables may interfere with their intended interpretation even though they fall within the required range.

Conclusion

To this point, we have prepared our data wholly, and it is now ready to be fed into various machine learning models. At this point, our data is free from irregularities, and the models make analytical sense of the dataset. I hope you found this helpful.