

EXP-5: Apply Data Pre-processing with the missing data handling, transformation of data and converting categorical data into label and one hot encoding. (Python/PowerBI)

Handling missing values


Encoding string values to integer values

Split data into a train-test-validation dataset

Feature scaling

Deletion of outliers

Let's discuss each term one by one along with their implementation using Python packages.""""

 <https://www.googleapis.com/download/storage/v1/b/kaggle-user-content/o/inbox%2F3328575%2Fd414b9295330421715f5176601c72732%2Fdataprep.jpeg?generation=1589614931397208&alt=media>


Data Preparation is one of the indispensable steps in any Machine Learning development life cycle. In today's world, the data is present in a structured as well as unstructured form. To deal with such data, data scientists spent almost 70-80% of their time in preparing data for further analysis which includes:

- Handling missing values
- Encoding string values to integer values
- Split data into a train-test-validation dataset
- Feature scaling
- Deletion of outliers

Let's discuss each term one by one along with their implementation using Python packages.'

```
#Importing libraries
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import sklearn
```


```
#Importing dataset
df = pd.read_csv("/content/Data (4).csv")
print(df)
```



	Country	Age	Salary	Purchased
0	France	44.0	72000.0	No
1	Spain	27.0	48000.0	Yes
2	Germany	30.0	54000.0	No
3	Spain	38.0	61000.0	No
4	Germany	40.0	NaN	Yes
5	France	35.0	58000.0	Yes
6	Spain	NaN	52000.0	No
7	France	48.0	79000.0	Yes
8	Germany	50.0	83000.0	No
9	France	37.0	67000.0	Yes

```
X = df.iloc[:, :-1].values
y = df.iloc[:, -1].values
```

```
#Let's separate the independent and the dependent variable before handling missing values.
print(X)
print(y)
```



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

Identifying and handling the missing values

We can handle these missing values by replacing the values with the following

aggregate values:

Mean

Mode (most frequent)

Median

Constant value

Delete the entire row/column with missing values

There is no rule of thumb to select a specific option, it depends on the data and the problem statement which is intended to solve. To select the best option, the knowledge of both data and the application are needed.

```
X = df.iloc[0:10,[0,1,2]].values# all rows and col-1,2 & 3
y = df.iloc[:,3].values# all rows col-1
print(X)
print(y)
```

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

```
df.isnull().sum()
```

```
0
Country    0
Age        1
Salary     1
Purchased  0
```

```
dtype: int64
```

```
#Solution 1 : Dropna
df1 = df.copy()
# summarize the shape of the raw data
print("Before:",df1.shape)

# drop rows with missing values
df1.dropna(inplace=True)

# summarize the shape of the data with missing rows removed
print("After:",df1.shape)
df1
```

```
Before: (10, 4)
After: (8, 4)
```

	Country	Age	Salary	Purchased
0	France	44.0	72000.0	No
1	Spain	27.0	48000.0	Yes
2	Germany	30.0	54000.0	No
3	Spain	38.0	61000.0	No
5	France	35.0	58000.0	Yes
7	France	48.0	79000.0	Yes
8	Germany	50.0	83000.0	No
9	France	37.0	67000.0	Yes

```
#Solution 2 : Fillna
df2 = df.copy()
df2[['Age', 'Salary']] = df2[['Age', 'Salary']].fillna(df2[['Age', 'Salary']].mean())
# count the number of NaN values in each column
print(df2.isnull().sum())
df2
```

```

Country      0
Age          0
Salary       0
Purchased    0
dtype: int64

```

	Country	Age	Salary	Purchased
0	France	44.000000	72000.000000	No
1	Spain	27.000000	48000.000000	Yes
2	Germany	30.000000	54000.000000	No
3	Spain	38.000000	61000.000000	No
4	Germany	40.000000	63777.777778	Yes
5	France	35.000000	58000.000000	Yes
6	Spain	38.777778	52000.000000	No
7	France	48.000000	79000.000000	Yes
8	Germany	50.000000	83000.000000	No
9	France	37.000000	67000.000000	Yes

We will use the SimpleImputer class from the sklearn.impute library to replace the missing values with the mean value of the corresponding columns.

Handling missing values

```

#Solution 3 : Scikit-Learn
from sklearn.impute import SimpleImputer
imputer = SimpleImputer(missing_values=np.nan, strategy="mean")
imputer.fit(X[:,1:3])
X[:,1:3] = imputer.transform(X[:,1:3])
print(X)

```

```

[[ '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]]

```

We replaced the missing values to 38.88 and 63777.77 respectively. Instead of mean, we can also replace values with their corresponding mode, median, or any constant values by changing the 'strategy' parameter.

Encoding dataset features

It is obvious to have a string-based column in the dataset in the form of names, addresses, and so on. But no machine learning algorithm can train the model with string-based variables in it. Hence, we have to encode those variables into numeric-based variables before delving into any machine learning algorithm.

There are several ways by which we can handle such kind of data. Here, we will use 2 special Python libraries to convert the string-based into numeric-based variables.

Encoding the independent variables

```

#Solution 1 : ColumnTransformer
print(X)
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))
X

```

```

[[ '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]]
array([[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]], dtype=object)
```

```
#Solution 2 : Pd.get_dummies()
pd.get_dummies(df2)
```



	Age	Salary	Country_France	Country_Germany	Country_Spain	Purchased_No	Purchased_Yes
0	44.000000	72000.000000	True	False	False	True	False
1	27.000000	48000.000000	False	False	True	False	True
2	30.000000	54000.000000	False	True	False	True	False
3	38.000000	61000.000000	False	False	True	True	False
4	40.000000	63777.777778	False	True	False	False	True
5	35.000000	58000.000000	True	False	False	False	True
6	38.777778	52000.000000	False	False	True	True	False
7	48.000000	79000.000000	True	False	False	False	True
8	50.000000	83000.000000	False	True	False	True	False
9	37.000000	67000.000000	True	False	False	False	True

```
#Solution 3 : LabelEncoder
from sklearn.preprocessing import LabelEncoder
le = LabelEncoder()
y = le.fit_transform(y)
y
```



```
array([0, 1, 0, 0, 1, 1, 0, 1, 0, 1])
```

In Figure 6, you must be wondering how some extra columns appeared which were not present before. If you did, then you are going on a correct path.

The Machine Learning algorithm has nothing to do with the column names, instead, it tries to find the patterns within the data. As per Figure 7, we can easily infer that the value 2 is bigger than the value 1 and 0; it may also infer that there is a numerical order within the data. Hence, some misinterpretation could happen between independent variables and the dependent variables which could lead to the wrong correlations and future results.

To mitigate such issues, we will create 1 column for each value with 0 and 1; 0 being the value is absent and 1 being present.

Employee		Anjali Kanisha Parul			
0	Anjali	0	1	0	0
1	Parul	1	0	0	1
2	Kanisha	2	0	1	0
3	Parul	3	0	0	1
4	Kanisha	4	0	1	0
5	Anjali	5	1	0	0
6	Parul	6	0	0	1
7	Anjali	7	1	0	0
8	Kanisha	8	0	1	0
9	Anjali	9	1	0	0



If you noticed, the dependent variable is also string-based. But, the dependent column has only 2 unique values; in that case, we can skip the above process, and directly we can encode the values from ['No', 'Yes'] to [0,1] correspondingly. After encoding, you will notice that we have only

0 and 1 in the dependent column, and that is what we want. Let's quickly modify the dependent variable 'y' to a numerical-based variable.

Split data into a train-test-validation dataset In every model development process, we need to train the model with a subset of the original data, further, the model predicts the values for the new unseen data to evaluate its performance in terms of accuracy, ROC-AUC curve, and so on. We will not be going to discuss any evaluation parameters here as it is out of scope for this tutorial, but I cover those in my further tutorial for sure, until then stay tuned. :)

To achieve the above scenario, we need to split the original dataset into 2 or sometimes 3 splits namely training, validation, and testing dataset. Let's discuss all the 3 datasets and their significance in the machine learning life cycle.

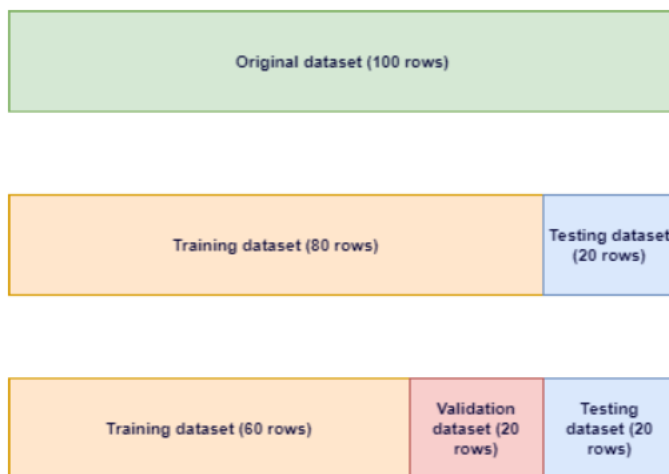
Training dataset: It consists of more than half of the original dataset, the sole purpose of this dataset is to train the model and update the weights of the model.

Validation dataset: A small subset of original data is used to provide an unbiased evaluation of a model fit on the training dataset while tuning model hyperparameters. It is optional to have a validation dataset into the model.

Testing dataset: It ranges from 10–25% of the original data to evaluate the model performance based on various evaluation parameters discussed above.

Generally, it is recommended to have a split of 70–30 or 80–20 ratios of the train-test split, and 60–20–20 or 70–15–15 in case of the train-validation-split dataset.

Consider the below example with 100 rows in the original dataset. In the middle split, an 80–20 split ratio happened between training and testing dataset. The training dataset further split into a 75–25 split ratios between training and validation dataset in the last split.



```

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)
print(X_train)
print(X_test)
print(y_train)
print(y_test)

```

```

[[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]]
[[0.0 1.0 0.0 30.0 54000.0]
 [1.0 0.0 0.0 37.0 67000.0]]
[[0 1 0 0 1 1 0 1]
 [0 1]

```

As you see, it is very easy to split the dataset using `train_test_split` class from `sklearn.model_selection` library. Choose the 'test_size' parameter between 0 to 1, in our case we took 0.2 to get 20% testing data. It is recommended to set the seed parameter 'random_state' to achieve the reproducibility of the results. If we do not set the seed, every time the random split occurs between the datasets and hence results differ every time the model runs.



It is a technique to standardize the independent variables into a fixed range. It is a very crucial step in the process of data preparation because if we skip this step, the distance-based models cause variables with larger values to tend to dominate the variables with smaller values.

We can achieve this in various ways, but we will discuss here the 2 most popular feature scaling techniques i.e. Min-Max scaling and Standardization. Let's discuss it one by one.

$$X_{\text{new}} = \frac{X_i - \min(X)}{\max(x) - \min(X)}$$

Min-Max scaling: In this technique, the features/variables are re-scaled between 0 and 1.

Standardization: In this, the features got re-scaled to the values so that the distribution with mean=0 and standard deviation=1.

$$X_{\text{new}} = \frac{X_i - X_{\text{mean}}}{\text{Standard Deviation}}$$

In our case, we will apply the standardization technique to scale the independent features in the training and testing dataset using sklearn.preprocessing library.

Feature scaling of training dataset

```
from sklearn.preprocessing import StandardScaler
sc_X = StandardScaler()
X_train = sc_X.fit_transform(X_train)
X_test = sc_X.transform(X_test)
print(X_train)
print(X_test)
```

```
[[ -0.77459667 -0.57735027  1.29099445 -0.19159184 -1.07812594]
 [ -0.77459667  1.73205081 -0.77459667 -0.01411729 -0.07013168]
 [  1.29099445 -0.57735027 -0.77459667  0.56670851  0.63356243]
 [ -0.77459667 -0.57735027  1.29099445 -0.30453019 -0.30786617]
 [ -0.77459667 -0.57735027  1.29099445 -1.90180114 -1.42046362]
 [  1.29099445 -0.57735027 -0.77459667  1.14753431  1.23265336]
 [ -0.77459667  1.73205081 -0.77459667  1.43794721  1.57499104]
 [  1.29099445 -0.57735027 -0.77459667 -0.74014954 -0.56461943]]
[[ -0.77459667  1.73205081 -0.77459667 -1.46618179 -0.9069571 ]
 [  1.29099445 -0.57735027 -0.77459667 -0.44973664  0.20564034]]
```

```
from sklearn.preprocessing import MinMaxScaler
mm = MinMaxScaler()
X_train[:, 3:] = mm.fit_transform(X_train[:, 3:])
X_test[:, 3:] = mm.transform(X_test[:, 3:])
print(X_train[:, 3:])
```

```
[[0.51207729 0.11428571]
 [0.56521739 0.45079365]
 [0.73913043 0.68571429]
 [0.47826087 0.37142857]
 [0.         0.         ]
 [0.91304348 0.88571429]
 [1.         1.         ]
 [0.34782609 0.28571429]]
```