1. Preprocessing

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

```
Mounted at /content/drive
```

Fetching the Dataset

```
import numpy as np
import pandas as pd

dataset=pd.read_csv('/content/drive/MyDrive/ML Lab/Data.csv')
dataset
```

	Country	Age	Salary	Purchased
0	France	44.0	72000.0	No
1	Spain	27.0	48000.0	Yes
2	Germany	NaN	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	NaN	No
9	France	37.0	67000.0	Yes

```
X=dataset.iloc[:,:-1].values    #---taking all the columns except the label column
y=dataset.iloc[:,-1].values #---taking the values of the label column
print(X)
print(y)
```

```
[['France' 44.0 72000.0]
 ['Spain' 27.0 48000.0]
 ['Germany' nan 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 nan]
 ['France' 37.0 67000.0]]
['No' 'Yes' 'No' 'No' 'Yes' 'Yes' 'No' 'Yes' 'No' 'Yes']
```

Handling missing data

```
from sklearn.impute import SimpleImputer

#---create an instance of the class SimpleImputer
imputer=SimpleImputer(missing_values=np.nan, strategy= 'mean') #---missing_values=np.nan, means all the empty values in the
#---strategy=mean, median, most_frequent, constant

#---connect the imputer object with the data
imputer.fit(X[:,1:3])  #---the column with numerical data is only fitted to imputer object

#----transform the data(replace the missing data with the mean)
X[:,1:3]=imputer.transform(X[:,1:3])

print(X) #----no missing values now in the X
```

```
[['France' 44.0 72000.0]
 ['Spain' 27.0 48000.0]
```

```
['Germany' 39.875 54000.0]
['Spain' 38.0 61000.0]
['Germany' 40.0 61375.0]
['France' 35.0 58000.0]
['Spain' 39.875 52000.0]
['France' 48.0 79000.0]
['Germany' 50.0 61375.0]
['France' 37.0 67000.0]]
```

----Arguments----

SimpleImputer(missing_values=nan, strategy='mean', fill_value=None, verbose=0, copy=True, add_indicator=False,)

copy-->If True, a copy of X will be created. If False, imputation will be done in-place whenever possible.

fill_value--> When strategy == "constant", fill_value is used to replace all occurrences of missing_values.default=None

Encoding Categorical data

Suppose the data is categorical in the dataset and we need to convert it into the numeric format

Here we are having "Purchased" & "Country" column as categorical data

'Country' has 3 values--> France, Spain, Germany, If we assign 0, 1 and 2 numbers respectively to each value then it will be considered as some kind of priority given to each value(0,1 & 2)

To do this, we can do one hot encoding(create different columns for each unique values)

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

#---Object of Column Transformer Class
ct= ColumnTransformer(transformers= [('encoder',OneHotEncoder(),[0])], remainder='passthrough')
#---3 things in 'transformers'= 1st--> kind of transformation/encoding
#---                           2nd--> Class of encoder
#---                           3rd--> index of the columns that we want to apply the OnHot encoding,here [0] for country co
#---remainder=passthrough --> that means we want to keep all other features/columns in feature matrix without transformation
#---{'drop','passthrough'}

print(type(X))
#---Connect the data with the 'ct' object,here we will use 'fit_transform' which will fit and transform together in 1 step
X=ct.fit_transform(X) #---update the encoded column to the same feature matrix X
X
```

```
<class 'numpy.ndarray'>
  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, 39.875, 54000.0],
         [0.0, 0.0, 1.0, 38.0, 61000.0],
         [0.0, 1.0, 0.0, 40.0, 61375.0],
         [1.0, 0.0, 0.0, 35.0, 58000.0],
         [0.0, 0.0, 1.0, 39.875, 52000.0],
         [1.0, 0.0, 0.0, 48.0, 79000.0],
         [0.0, 1.0, 0.0, 50.0, 61375.0],
         [1.0, 0.0, 0.0, 37.0, 67000.0]], dtype=object)
```

Encoding the dependent Variable

```
from sklearn.preprocessing import LabelEncoder

le=LabelEncoder()
y=le.fit_transform(y)  #---parameter as dependent variable
print(y)
```

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

Splitting Dataset

```
from sklearn.model_selection import train_test_split
#---returns the 4 matrices-2 for training and 2 for testing


X_train,X_test,y_train,y_test=train_test_split(X,y,test_size=0.2,random_state=1)  #--random_state=seed value for splitting o
#---X stands for feature matrix and y stands for target matrix
```

```
print(X_train)
```

```
[[0.0 0.0 1.0 39.875 52000.0]
 [0.0 1.0 0.0 40.0 61375.0]
 [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 61375.0]
 [1.0 0.0 0.0 35.0 58000.0]]
```

```
print(X_test)
```

```
[[0.0 1.0 0.0 39.875 54000.0]
 [1.0 0.0 0.0 37.0 67000.0]]
```

```
print(y_train)
```

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

```
print(y_test)
```

```
[0 1]
```

Feature Scaling

Feature scaling is done to prevent over dominate some feature values just based upon their magnitues even if they were not important

2 Methods of Feature Scaling

- Standardization (scales between -3 and +3)

  x(stand)=[x-mean(x)]/standard deviation(x)

- Normalization (scales between 0 and +1)

  x (norm)=[x-min(x)]/max(x)-min(x)

Normalization is recomended when you have a normal distribution in features

Standardization is used in general conditions

```
from sklearn.preprocessing import StandardScaler

sc=StandardScaler()
X_train[:,3:]=sc.fit_transform(X_train[:,3:])  #---specify the range, takes all rows and selected columns from 3 upto last c
#---fit method will only compute the mean and standard deviation of the feature,then transform method will apply the formula

#----we apply the same scaler to tes set, thats why we use 'transform' method only
X_test[:,3:]=sc.transform(X_test[:,3:])
```

```
print(X_train)
```

```
[[0.0 0.0 1.0 -0.05231070414657415 -1.0245464314521104]
 [0.0 1.0 0.0 -0.03411567661733096 -0.023360993551025323]
 [1.0 0.0 0.0 0.5481252043184508 1.1113158360702047]
 [0.0 0.0 1.0 -0.3252361170852219 -0.06340841106706872]
 [0.0 0.0 1.0 -1.9263985396586218 -1.4517188849565736]
 [1.0 0.0 0.0 1.1303660852542325 1.858867629703015]
 [0.0 1.0 0.0 1.4214865257221234 -0.023360993551025323]
 [1.0 0.0 0.0 -0.7619167777870582 -0.38378775119541597]]
```

```
1 print(X_test)
```

```
[[0.0 1.0 0.0 -0.05231070414657415 -0.8109602046998791]
 [1.0 0.0 0.0 -0.4707963373191673 0.5773502691896258]]
```

Note : We need to apply the same scaler of training data on the test data, otherwise it will compute the different values of mean and standard deviation for the test set and the results would be affected. So that we will get the same transformation on the test set

Do we have to apply the scaling to the dummy variables/ one hot encoders? No, as the scaling takes the values between -3 and +3, the valuesin dummy varibales lie between 0 and 1 which fall in this range.

Apply feature scaling to numerical values only, not to dummy variables.

Feature scaling should be done after splitting the data because the test set should be totally unseen, and feature scaling scales the features into one scale so that no fetaure dominates others just on the basis of magnitude

Feature scaling finds the mean or standard deviation of the feature in order to perform scaling

Doing feature scaling before, will do some information leakage and the test data will not totally remain unseen