EXPERIMENT -01

Aim : Data preprocessing is essential before its actual use. Data preprocessing is the concept of changing the raw data into a clean data set. In this experiment you have to perform data preprocessing on one of the popular dataset called Carseats which contains 400 observations and 11 variables or features

Description :

Pre-processing refers to the transformations applied to our data before feeding it to the algorithm. Data Preprocessing is a technique that is used to convert the raw data into a clean data set. In other words, whenever the data is gathered from different sources it is collected in raw format which is not feasible for the analysis.

Need of Data Preprocessing :

- For achieving better results from the applied model in Machine Learning projects the format of the data has to be in a proper manner. Some specified Machine Learning model needs information in a specified format, for example, Random Forest algorithm does not support null values, therefore to execute random forest algorithm null values have to be managed from the original raw data set.
- Another aspect is that the data set should be formatted in such a way that more than one Machine Learning and Deep Learning algorithm are executed in one data set, and best out of them is chosen.

Dataset Description :

The Carseats data set tracks sales information for car seats. It has 400 observations (each at a different store) and 11 variables:

- CompPrice: price charged by competitor at each location
- Sales : unit sales in thousands
- Income: community income level in 1000s of dollars
- Advertising: local ad budget at each location in 1000s of dollars
- Population: regional pop in thousands
- Price: price for car seats at each site
- ShelveLoc: Bad, Good or Medium indicates quality of shelving location
- Age: age level of the population
- Education: ed level at location
- Urban: Yes/No
- US: Yes/No

Procedure:

1. Importing the libraries
2. Importing the Dataset
3. Handling of Missing Data
4. Handling of Categorical Data
5. Splitting the dataset into training and testing datasets
6. Feature Scaling

Code Snippets :

```
import numpy as np
import pandas as pd
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import LabelEncoder, OneHotEncoder
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.preprocessing import MinMaxScaler

import matplotlib.pyplot as plt
dataset = pd.read_csv('Carseats.csv')
X = dataset.iloc[:, 1:11].values
y = dataset.iloc[:, 0].values

X
```

```
array([[138, 73, 11, ..., 17, 'Yes', 'Yes'],
       [111, 48, 16, ..., 10, 'Yes', 'Yes'],
       [113, 35, 10, ..., 12, 'Yes', 'Yes'],
       ...,
       [162, 26, 12, ..., 18, 'Yes', 'Yes'],
       [100, 79, 7, ..., 12, 'Yes', 'Yes'],
       [134, 37, 0, ..., 16, 'Yes', 'Yes']], dtype=object)
```

Handling of Missing Data : the dataset what you have may have missing values, invalid inputs , moise etc. to make the dataset ready for the model we nee to handle the missing or noisy data

Handling of Categorical Data As you might know by now, we can't have text in our data if we're going to run any kind of model on it. So before we can run a model, we need to make this data ready for the model. it can be two ways LabelEncoder and OneHotEncoder

```
le = LabelEncoder()
X[:, 0] = le.fit_transform(X[:, 0])
X
```

```
array([[49, 73, 11, ..., 17, 'Yes', 'Yes'],
       [22, 48, 16, ..., 10, 'Yes', 'Yes'],
       [24, 35, 10, ..., 12, 'Yes', 'Yes'],
       ...,
       [71, 26, 12, ..., 18, 'Yes', 'Yes'],
       [12, 79, 7, ..., 12, 'Yes', 'Yes'],
       [45, 37, 0, ..., 16, 'Yes', 'Yes']], dtype=object)
```

```
ohe = OneHotEncoder()
X = ohe.fit_transform(X).toarray()

from sklearn.compose import ColumnTransformer
ct = ColumnTransformer(transformers=[('encoder', OneHotEncoder(), [0])], remainder='passth
rough')
X = np.array(ct.fit_transform(X))

from sklearn.preprocessing import LabelEncoder
le = LabelEncoder()
y = le.fit_transform(y)

#Splitting dataset into training and testing datasets
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 =
0)
```

Feature Scaling The real-world dataset contains features that highly vary in magnitudes, units, and range. Normalization should be performed when the scale of a feature is irrelevant or misleading and not should Normalise when the scale is meaningful.

```
sc = StandardScaler()
X_train[:, 3:] = sc.fit_transform(X_train[:, 3:])
X_test[:, 3:] = sc.transform(X_test[:, 3:])
X_train.head()
```

	CompPrice	Income	Advertising	Population	Price	Age	Education Urba
336	0.885481	-1.189418	-0.089216	-1.418971	1.155793	-1.530974	1.574353
64	-1.614507	-0.031327	0.807613	-0.570074	-0.507142	-1.283981	0.799287
55	1.214427	0.475337	-0.238687	-1.418971	1.624826	0.506723	1.574353
106	-1.482929	-1.261799	-0.986044	-0.344158	0.985236	1.062458	1.574353
300	-0 561881	0 366766	-0 836573	-0 748069	-0 720339	-0 481252	-1 138378

The final Output is a preprocessed dataset that can be used for machine learning using a model.

EXPERIMENT -02

Aim : Most of the datasets available in internet for practice are having continuous values but to work with classification algorithms we need a dataset with categorical values. To use the same dataset for regression and classification we need to convert our continuous values to categorical values. In this Experiment your task is to perform a conversion of continuous to categorical values on sales feature of Carseats dataset.

Description :

Numerical data such as continuous, highly skewed data is frequently seen in data analysis. Sometimes analysis becomes effortless on conversion from continuous to discrete data. There are many ways in which conversion can be done, one such way is by using Pandas' integrated cut-function. Pandas' cut function is a distinguished way of converting numerical continuous data into categorical data. It has 3 major necessary parts:

1. First and foremost is the 1-D array/DataFrame required for input.
2. The other main part is bins. Bins that represent boundaries of separate bins for continuous data. The first number denotes the start point of the bin and the following number denotes the endpoint of the bin. Cut function permits more explicitness of the bins
3. The final main part is labels. The number of labels without exception will be one lower than the number of bins.

Dataset Description :

The Carseats data set tracks sales information for car seats. It has 400 observations (each at a different store) and 11 variables:

- CompPrice: price charged by competitor at each location
- Sales : unit sales in thousands
- Income: community income level in 1000s of dollars
- Advertising: local ad budget at each location in 1000s of dollars
- Population: regional pop in thousands
- Price: price for car seats at each site
- ShelveLoc: Bad, Good or Medium indicates quality of shelving location
- Age: age level of the population
- Education: ed level at location
- Urban: Yes/No
- US: Yes/No

Procedure :

Importing all the required packages using import statement.

numpy -> numerical python used mainly for working with arrays,linear algebra and matrices.

pandas -> an open source library which can be used to analyze data easily in Python. Pandas provide an easy way to create, manipulate, and wrangle the data.

scikit-learn -> sklearn package for selection of efficient tools like classification, regression, clustering and dimensionality reduction etc.

matplotlib ->for data visualization and graphical plotting.

Pandas cut function is used for converting numerical continuous data into categorical data. Here the values of 'y' dataset are in numerical form, we have converted these values into two categorical values 'Good','Bad' based upon the calculae average and maximum values of y.

- If the value of 'y' lies in the range between 0 and average value, then it is categorized as 'Bad'. Else if the value of 'y' lies in the range between average value and maximum value, then it is categorized as 'Good'.
- The bins parameter denotes the bin boundaries for segmentation.
- The right parameter denotes whether rightmost edge of bins should be included or not.It is of Boolean type and default value is True.
- The labels parameter defines labels for returned segmented bins.

Code Snippets :

```
#importing essential libraries

import numpy as np
import pandas as pd

#importing the carseats dataset

dataset=pd.read_csv('/content/Carseats.csv')
dataset.head(5)

#checking for null values
dataset.isnull().sum()
```

```
Sales          0
CompPrice      0
Income         0
Advertising    0
Population     0
Price          0
ShelveLoc      0
Age            0
Education      0
Urban          0
US             0
dtype: int64
```

```
y=dataset.iloc[:,0]
y
```

```
y=pd.cut(y,bins=[np.min(y),np.mean(y),np.max(y)],labels=['bad','good'])
y
```

Output :

```
0      good
1      good
2      good
3       bad
4       bad
       ...
395    good
396     bad
397     bad
398     bad
399    good
Name: Sales, Length: 400, dtype: category
Categories (2, object): ['bad' < 'good']
```

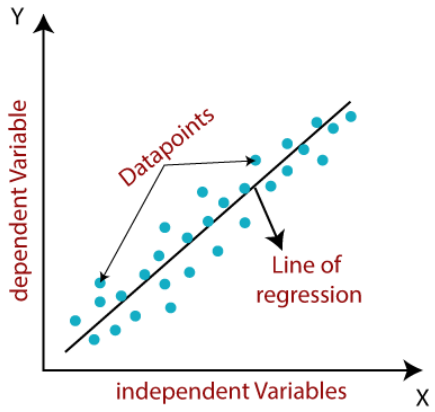
EXPERIMENT -03

Aim : Regression models a target prediction value based on independent variables. Linear-regression models have become a proven way to scientifically and reliably predict the outcome. In this experiment you need to perform a linear regression on one of the prominent Wine dataset.

Description :

Linear regression is one of the easiest and most popular Machine Learning algorithms. It is a statistical method that is used for predictive analysis. Linear regression makes predictions for continuous/real or numeric variables such as **sales, salary, age, product price,** etc.

A linear line showing the relationship between the dependent and independent variables is called a **regression line.**



Dataset Description :

The dataset is gathered from kaggle, which is named as Red Wine Quality dataset. The dataset has info regarding 1143 samples and has 12 attributes.

It Consists of the following Input Variables :

Input variables (based on physicochemical tests):
1 - fixed acidity
2 - volatile acidity
3 - citric acid
4 - residual sugar
5 - chlorides
6 - free sulfur dioxide
7 - total sulfur dioxide
8 - density
9 - pH
10 - sulphates
11 - alcohol

Output variable (based on sensory data):
12 - quality (score between 0 and 10)

Procedure :

Construct a Correlation Matrix to select a feature as independent variable which has highest correlation with the dependent variable in this case Density has the highest correlation with Quality.

Take 2 separate variables x for independent variable and y for Dependent variable. Using train_test_split from sklearn.model_selection module split the dataset into training and testing sets.

```
[19] df.corr()
```

	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	pH	sulphates	alcohol	quality	Id
fixed acidity	1.000000	-0.250728	0.673157	0.171831	0.107889	-0.164831	-0.110628	0.681501	-0.685163	0.174592	-0.075055	0.121970	-0.275826
volatile acidity	-0.250728	1.000000	-0.544187	-0.005751	0.056336	-0.001962	0.077748	0.016512	0.221492	-0.276079	-0.203909	-0.407394	-0.007892
citric acid	0.673157	-0.544187	1.000000	0.175815	0.245312	-0.057589	0.036871	0.375243	-0.546339	0.331232	0.106250	0.240821	-0.139011
residual sugar	0.171831	-0.005751	0.175815	1.000000	0.070863	0.165339	0.190790	0.380147	-0.116959	0.017475	0.058421	0.022002	-0.046344
chlorides	0.107889	0.056336	0.245312	0.070863	1.000000	0.015280	0.048163	0.208901	-0.277759	0.374784	-0.229917	-0.124085	-0.088099
free sulfur dioxide	-0.164831	-0.001962	-0.057589	0.165339	0.015280	1.000000	0.661093	-0.054150	0.072804	0.034445	-0.047095	-0.063260	0.095268
total sulfur dioxide	-0.110628	0.077748	0.036871	0.190790	0.048163	0.661093	1.000000	0.050175	-0.059126	0.026894	-0.188165	-0.183339	-0.107389
density	0.681501	0.016512	0.375243	0.380147	0.208901	-0.054150	0.050175	1.000000	-0.352775	0.143139	-0.494727	-0.175208	-0.363926
pH	-0.685163	0.221492	-0.546339	-0.116959	-0.277759	0.072804	-0.059126	-0.352775	1.000000	-0.185499	0.225322	-0.052453	0.132904
sulphates	0.174592	-0.276079	0.331232	0.017475	0.374784	0.034445	0.026894	0.143139	-0.185499	1.000000	0.094421	0.257710	-0.103954
alcohol	-0.075055	-0.203909	0.106250	0.058421	-0.229917	-0.047095	-0.188165	-0.494727	0.225322	0.094421	1.000000	0.484866	0.238087
quality	0.121970	-0.407394	0.240821	0.022002	-0.124085	-0.063260	-0.183339	-0.175208	-0.052453	0.257710	0.484866	1.000000	0.069708
Id	-0.275826	-0.007892	-0.139011	-0.046344	-0.088099	0.095268	-0.107389	-0.363926	0.132904	-0.103954	0.238087	0.069708	1.000000

Perform Standardization using Standardscaler available in sklearn.preprocessing. The purpose of standardization is to scale the data within a particular range to make the model more efficient in its training and predictions

```
#Standardising
from sklearn.preprocessing import StandardScaler
sc = StandardScaler()
x_train = sc.fit_transform(x_train)
x_test = sc.transform(x_test)
x_train[:15]
```

```
array([[ 1.61699432],
       [-0.49788539],
       [-0.95950887],
       [ 0.91919137],
       [ 0.32874272],
       [-0.15435163],
       [-0.26170593],
       [-0.54082711],
       [ 0.59712847],
       [-0.96487659],
       [-1.78613698],
       [-0.1436162 ],
       [-0.74480028],
       [-0.787742  ],
       [ 0.65080562]])
```

Code Snippets :

```
#importing essential libraries
import numpy as np
import pandas as pd

df = pd.read_csv("WineQT.csv")
df.head()

df.corr()

#Taking Dependent and Independent Variables
x = df[['density']]
y = df[['quality']]
y[:10]

#Splitting the train and test sets

from sklearn.model_selection import train_test_split

x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.33, random_state=42)

 #Standardising
 from sklearn.preprocessing import StandardScaler
 sc = StandardScaler()
 x_train = sc.fit_transform(x_train)
 x_test = sc.transform(x_test)

#Building the Model

from sklearn.linear_model import LinearRegression

regression = LinearRegression()
regression.fit(x_train, y_train)

#Predicting
y_pred = regression.predict(x_test)

from sklearn.metrics import mean_squared_error
mean_squared_error(y_test, y_pred)
```

OUTPUT :

```
from sklearn.metrics import mean_squared_error
mean_squared_error(y_test, y_pred)
```

```
0.6090265085067917
```

EXPERIMENT -04

Aim : In the real world, multiple linear regressions are used more frequently than simple linear regression. This is mostly the case because: Multiple linear regressions allow evaluating the relationship between two variables to evaluate the performance of the model. In this experiment you need to perform a multiple linear regression on one of the prominent Wine dataset.

Multiple Linear Regression:

Multiple Linear Regression is an extension of Simple Linear regression as it takes more than one predictor variable to predict the response variable. We can define it as:

> Multiple Linear Regression is one of the important regression algorithms which models the linear relationship between a single dependent continuous variable and more than one independent variable.

Some key points about MLR:

- For MLR, the dependent or target variable(Y) must be the continuous/real, but the predictor or independent variable may be of continuous or categorical form.
- Each feature variable must model the linear relationship with the dependent variable.
- MLR tries to fit a regression line through a multidimensional space of data-points.

Here : $Y = b0 + b1 * x1 + b2 * x2 + b3 * x3 + \ldots\ldots bn * xn$
Y = Dependent variable
x1, x2, x3, …… xn = multiple independent variables

Dataset Description :

The dataset is gathered from kaggle, which is named as Red Wine Quality dataset. The dataset has info regarding 1143 samples and has 12 attributes.

It Consists of the following Input Variables :

Input variables (based on physicochemical tests):
1 - fixed acidity
2 - volatile acidity
3 - citric acid
4 - residual sugar
5 - chlorides
6 - free sulfur dioxide
7 - total sulfur dioxide
8 - density
9 - pH
10 - sulphates
11 - alcohol

Output variable (based on sensory data):
12 - quality (score between 0 and 10)

Procedure :

The procedure is the same as in linear regression, here in multiple linear regression more than one input variables are used as independent variables and only one dependent variable.

Using train_test_split from sklearn.model_selection module split the dataset into training and testing sets.

LinearRegression model is built using libraries available in sklearn.

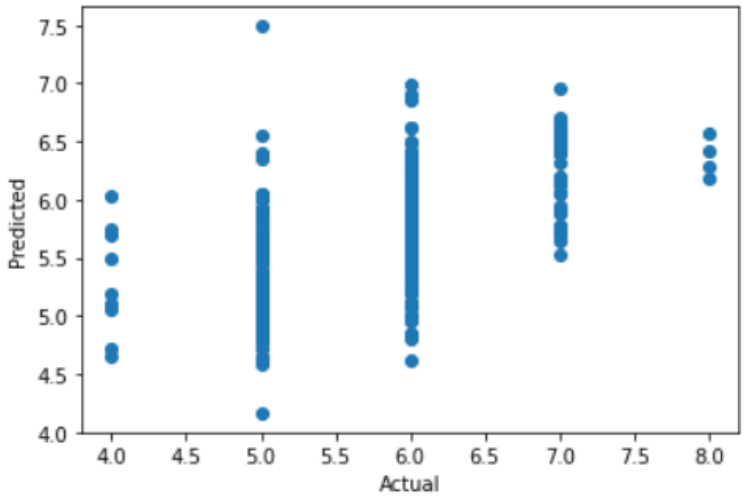
Code Snippets :

```
#importing essential libraries
import numpy as np
import pandas as pd

df = pd.read_csv("WineQT.csv")
df.head()

#Taking Dependent and Independent Variable

x = df.iloc[:,:-2]
y = df.iloc[:,-2]
Y

#Splitting the train and test sets

from sklearn.model_selection import train_test_split

x_train, x_test, y_train, y_test = train_test_split(x, y, test_size = 0.30, random_state=42)

#Building the Model

from sklearn.linear_model import LinearRegression
regression = LinearRegression()

reg = regression.fit(x_train, y_train)

#Predicting

y_pred = reg.predict(x_test)
y_pred[:10]

#Plotting the outputs vs test data

import matplotlib.pyplot as plt

plt.scatter(y_test, y_pred)
plt.xlabel('Actual')
plt.ylabel('Predicted')

#Calculating the error of the model
```

```
from sklearn.metrics import mean_squared_error
mean_squared_error(y_pred,y_test)
```

OUTPUT :

```
[ ]  import matplotlib.pyplot as plt

     plt.scatter(y_test, y_pred)
     plt.xlabel('Actual');
     plt.ylabel('Predicted');
```




```
[ ]  from sklearn.metrics import mean_squared_error
     mean_squared_error(y_pred,y_test)
```

```
0.38559178373841624
```

EXPERIMENT - 05

Aim : Decision tree is one of the supervised learning techniques which can be used for both regression and classification problems. One of the great ability of decision tree classifier is, it uses different subset of features and rules in different stages of classification. In this experiment you need to perform Decision tree classification on Pizza dataset to categorize the brand of the pizza.

Description :

Decision Trees are a type of Supervised Machine Learning (that is you explain what the input is and what the corresponding output is in the training data) where the data is continuously split according to a certain parameter. The tree can be explained by two entities, namely decision nodes and leaves.

- Decision tree algorithm falls under the category of supervised learning. They can be used to solve both regression and classification problems.
- Decision tree uses the tree representation to solve the problem in which each leaf node corresponds to a class label and attributes are represented on the internal node of the tree.
- We can represent any boolean function on discrete attributes using the decision tree.

Dataset Description :

The data set pizza. csv contains measurements that capture the kind of things that make a pizza tasty. Can you determine which pizza brand works best for you and explain why. The variables in the data set are:

brand -- Pizza brand (class label)
id -- Sample analysed
mois -- Amount of water per 100 grams in the sample
prot -- Amount of protein per 100 grams in the sample
fat -- Amount of fat per 100 grams in the sample
ash -- Amount of ash per 100 grams in the sample
sodium -- Amount of sodium per 100 grams in the sample
carb -- Amount of carbohydrates per 100 grams in the sample
cal -- Amount of calories per 100 grams in the sample

Importing all the required packages using import statement.

numpy -> numerical python used mainly for working with arrays,linear algebra and matrices. **pandas** -> an open source library which can be used to analyze data easily in Python. Pandas provide an easy way to create, manipulate, and wrangle the data.

scikit-learn -> sklearn package for selection of efficient tools like classification, regression, clustering and dimensionality reduction etc.

matplotlib ->for data visualization and graphical plotting.

Code Snippets :

```
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
```

```
from sklearn.metrics import accuracy_score
from sklearn.linear_model import LogisticRegression
from sklearn.preprocessing import LabelEncoder, OneHotEncoder, RobustScaler
import matplotlib.pyplot as plt
from sklearn.preprocessing import StandardScaler
from sklearn.preprocessing import MinMaxScaler
import seaborn as sb

df=pd.read_csv('Pizza.csv')
x=df.iloc[:,1:8]
y=df.iloc[:,0]

sc=StandardScaler()
X_train.iloc[:,:]=sc.fit_transform(X_train.iloc[:,:])
X_test.iloc[:,:]=sc.fit_transform(X_test.iloc[:,:])
print(X_train)

from sklearn.tree import DecisionTreeClassifier
d=DecisionTreeClassifier(criterion='entropy',random_state=0)

d.fit(X_train,y_train)
DecisionTreeClassifier(criterion='entropy', random_state=0)

y_predict=d.predict(X_test)
y_predict
```

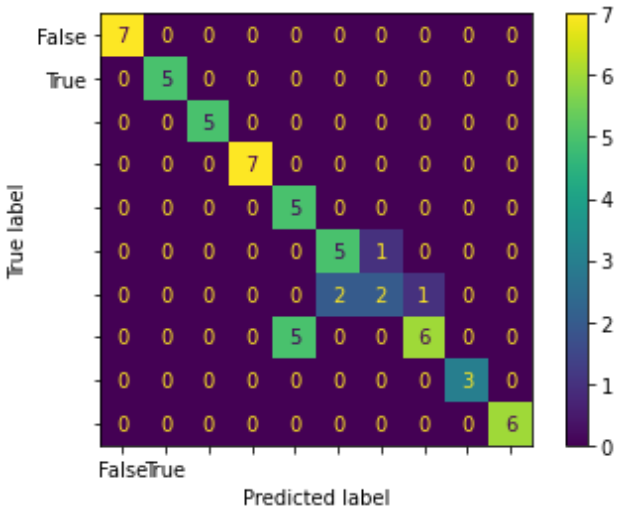
```
array(['E', 'G', 'A', 'H', 'I', 'E', 'E', 'H', 'B', 'D', 'J', 'A', 'G',
       'I', 'A', 'C', 'J', 'D', 'E', 'F', 'H', 'D', 'E', 'E', 'E', 'D',
       'F', 'J', 'H', 'J', 'D', 'C', 'A', 'E', 'H', 'F', 'G', 'F', 'J',
       'A', 'H', 'D', 'C', 'B', 'B', 'E', 'E', 'C', 'A', 'B', 'C', 'H',
       'F', 'D', 'B', 'I', 'F', 'A', 'J', 'F'], dtype=object)
```

```
from sklearn import metrics

m=metrics.accuracy_score(y_test,y_predict) print(m*100)

85.0
cm = metrics.confusion_matrix(y_test, y_predict)

print(cm)
```

```
[[7 0 0 0 0 0 0 0 0 0]
 [0 5 0 0 0 0 0 0 0 0]
 [0 0 5 0 0 0 0 0 0 0]
 [0 0 0 7 0 0 0 0 0 0]
 [0 0 0 0 5 0 0 0 0 0]
 [0 0 0 0 0 5 1 0 0 0]
 [0 0 0 0 0 2 2 1 0 0]
 [0 0 0 0 5 0 0 6 0 0]
 [0 0 0 0 0 0 0 0 3 0]
 [0 0 0 0 0 0 0 0 0 6]]
```

```
import matplotlib.pyplot as plt

cm_display=metrics.ConfusionMatrixDisplay(confusion_matrix=cm,display_labels=[False,True])
```

```
cm_display.plot()
plt.show()
```



EXPERIMENT - 06

Aim : Decision tree is one of the supervised learning techniques which can be used for both regression and classification problems. Decision tree regression supports for non linear when compared to linear regression models. In this experiment you are going to perform a decision tree regression on Dataset of your choice.

Description :

Decision Trees are a type of Supervised Machine Learning (that is you explain what the input is and what the corresponding output is in the training data) where the data is continuously split according to a certain parameter. The tree can be explained by two entities, namely decision nodes and leaves.

- Decision tree algorithm falls under the category of supervised learning. They can be used to solve both regression and classification problems.
- Decision tree uses the tree representation to solve the problem in which each leaf node corresponds to a class label and attributes are represented on the internal node of the tree.
- We can represent any boolean function on discrete attributes using the decision tree.

Dataset Description :

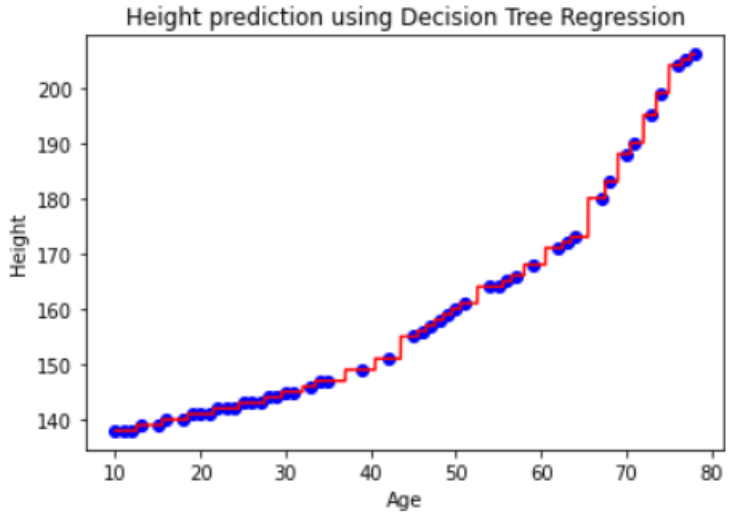
The data set 'Height_Age_Dataset.csv' contains the heights an ages of different people. The variables in the data set are:
Height – height of the person in cm
Age – age of the person

Code :

```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
from sklearn.linear_model import LogisticRegression
from sklearn.preprocessing import LabelEncoder, OneHotEncoder, RobustScaler

import matplotlib.pyplot as plt
from sklearn.preprocessing import StandardScaler
from sklearn.preprocessing import MinMaxScaler
import seaborn as sb

# Import the Height Weight Dataset

data = pd.read_csv('Height_Age_Dataset.csv')
data.head()
```

	Age	Height
0	10	138
1	11	138
2	12	138
3	13	139
4	14	139

```
#Store the data in the form of dependent and independent variables separately
X = data.iloc[:, 0:1].values
y = data.iloc[:, 1].values

#Split the Dataset into Training and Test Dataset
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=0)

#Import the Decision Tree Regressor
from sklearn.tree import DecisionTreeRegressor

#Create a decision tree regressor object from DecisionTreeRegressor class
DtReg = DecisionTreeRegressor(random_state = 0)

#Fit the decision tree regressor with training data represented by X_train and y_train
DtReg.fit(X_train, y_train)

DecisionTreeRegressor(random_state=0)

#Predicted Height from test dataset w.r.t Decision Tree Regression
y_predict_dtr = DtReg.predict((X_test))

#Model Evaluation using R-Square for Decision Tree Regression
from sklearn import metrics
r_square = metrics.r2_score(y_test, y_predict_dtr)
print('R-Square Error associated with Decision Tree Regression is:', r_square)


R-Square Error associated with Decision Tree Regression is: 0.9941828370498541

''' Visualise the Decision Tree Regression by creating range of values from min value of
X_tr having a difference of 0.01 between two consecutive values'''

X_val = np.arange(min(X_train), max(X_train), 0.01)

#Reshape the data into a len(X_val)*1 array in order to make a column out of the X_val
values X_val = X_val.reshape((len(X_val), 1))

#Define a scatter plot for training data
plt.scatter(X_train, y_train, color = 'blue')

#Plot the predicted data
plt.plot(X_val, DtReg.predict(X_val), color = 'red')

#Define the title
plt.title('Height prediction using Decision Tree Regression')

#Define X axis label plt.xlabel('Age')

#Define Y axis label plt.ylabel('Height')

#Set the size of the plot for better clarity plt.figure(figsize=(1,1))

#Draw the plot
plt.show()
```




```
#Import export_graphviz package
from sklearn.tree import export_graphviz

#Store the decision tree in a tree.dot file in order to visualize the plot.
#Visualize it on http://www.webgraphviz.com/ by copying and pasting related data from
dtregre
export_graphviz(DtReg, out_file ='dtregression.dot',
feature_names =['Age'])


# Predicting Height based on Age using Decision Tree Regression
height_pred = DtReg.predict([[41]])
print("Predicted Height: % d"% height_pred)

redicted Height:  15
```

EXPERIMENT - 07

Aim : Linear Discriminate Analysis, or LDA for short, is a predictive modeling algorithm for multi-class classification. It is also used for dimensionality reduction technique. In this experiment you have to perform LDA.

Description :

- Linear Discriminant Analysis is a linear classification machine learning algorithm.
- LDA is a dimensionality reduction mechanism which uses the information from both features to create a new axis and projects the data onto the new axis.
- The algorithm involves developing a probabilistic model per class based on the specific distribution of observations for each input variable.
- A new example is then classified by calculating the conditional probability of it belonging to each class and selecting the class with the highest probability

Dataset description :

We utilize the wine dataset which can be obtained from the UCI machine learning repository. The scikit-learn library provides a wrapper function for downloading the dataset.The dataset contains 178 rows of 13 columns each.

The data set has the following attributes:

- Alcohol
- Malic acid
- Ash
- Alkalinity of ash
- Magnesium
- Total phenols
- Flavonoids
- Non Flavonoid phenols
- Proanthocyanidins
- Color intensity
- Hue
- OD280/OD315 of diluted wines
- Proline

Code:

Firstly, we import the required libraries

```
from sklearn.datasets import load_wine
import pandas as pd
import numpy as np
np.set_printoptions(precision=4)
from matplotlib import pyplot as plt
```

```
import seaborn as sns
sns.set()
from sklearn.preprocessing import LabelEncoder
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix
```

Now, we load the dataset and separate the dependent variable and independent variable in variables named as “dependentVariable” and “independentVariables” respectively.

```
wine = load_wine()
X = pd.DataFrame(wine.data, columns=wine.feature_names)
y = pd.Categorical.from_codes(wine.target, wine.target_names)
```

We create a DataFrame named "df" containing both the features and classes.

```
df = X.join(pd.Series(y, name='class'))
```

We create a vector containing the means of each feature for every class.

```
class_feature_means = pd.DataFrame(columns=wine.target_names)
for c, rows in df.groupby('class'):
    class_feature_means[c] = rows.mean()
class_feature_means
```

```
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:3:
FutureWarning: Dropping of nuisance columns in DataFrame reductions
(with 'numeric_only=None') is deprecated; in a future version this will
raise TypeError.  Select only valid columns before calling the
reduction.This is separate from the ipykernel package so we can avoid
doing imports until
```

	class_0	class_1	class_2
alcohol	13.744746	12.278732	13.153750
malic_acid	2.010678	1.932676	3.333750
ash	2.455593	2.244789	2.437083
alcalinity_of_ash	17.037288	20.238028	21.416667
magnesium	106.338983	94.549296	99.312500
total_phenols	2.840169	2.258873	1.678750
flavonoids	2.982373	2.080845	0.781458
nonflavanoid_phenols	0.290000	0.363662	0.447500
proanthocyanidins	1.899322	1.630282	1.153542
color_intensity	5.528305	3.086620	7.396250
hue	1.062034	1.056282	0.682708
od280/od315_of_diluted_wines	3.157797	2.785352	1.683542
proline	1115.711864	519.507042	629.895833

Then, we obtain the within the class scatter matrix and between the class scatter matrix.

```
within_class_scatter_matrix = np.zeros((13,13))
for c, rows in df.groupby('class'):
 rows = rows.drop(['class'], axis=1)
  s = np.zeros((13,13))
for index, row in rows.iterrows():
        x, mc = row.values.reshape(13,1),
class_feature_means[c].values.reshape(13,1)
        s += (x - mc).dot((x - mc).T)
within_class_scatter_matrix += s

feature_means = df.mean()
between_class_scatter_matrix = np.zeros((13,13))
for c in class_feature_means:
    n = len(df.loc[df['class'] == c].index)

    mc, m = class_feature_means[c].values.reshape(13,1),
feature_means.values.reshape(13,1)

    between_class_scatter_matrix += n * (mc - m).dot((mc - m).T)
between_class_scatter_matrix
```

```
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:10:
FutureWarning: Dropping of nuisance columns in DataFrame reductions
(with 'numeric_only=None') is deprecated; in a future version this will
raise TypeError.  Select only valid columns before calling the
reduction.
  # Remove the CWD from sys.path while we load stuff.
```

```
array([[ 7.0795e+01,  1.3723e+01,  1.0668e+01, -1.3186e+02,
5.5262e+02,
         2.1257e+01,  3.0029e+01, -2.6178e+00,  8.3076e+00,
1.3888e+02,
        -2.4933e+00,  8.2530e+00,  2.6987e+04],
       [ 1.3723e+01,  6.5578e+01,  5.1556e+00,  1.1793e+02,
1.5062e+00,
        -3.8943e+01, -7.9531e+01,  5.4292e+00, -2.7994e+01,
1.5941e+02,
        -1.7995e+01, -5.9906e+01, -6.1709e+03],
       [ 1.0668e+01,  5.1556e+00,  1.7592e+00, -1.2829e+01,
7.8095e+01,
         1.0900e+00,  3.3671e-01, -1.0316e-01, -2.0087e-01,
2.7430e+01,
        -1.2351e+00, -1.7747e+00,  3.5073e+03],
       [-1.3186e+02,  1.1793e+02, -1.2829e+01,  5.7283e+02, -
1.2702e+03,
        -1.3780e+02, -2.5058e+02,  1.8415e+01, -8.2987e+01,
4.3483e+01,
        -3.5293e+01, -1.5564e+02, -7.6268e+04],
       ...])
```

We simplify the generalized eigen value problem for obtaining the linear discriminants.

```
eigen_values, eigen_vectors =
```

```
np.linalg.eig(np.linalg.inv(within_class_scatter_matrix).dot(betwe
en_class_scatter_matrix))
```

We sort the eigenvalues from highest to lowest and select the first k eigenvectors. We place them in a temporary array to make sure that the eigenvalue maps to the same eigenvector after sorting .

```
pairs = [(np.abs(eigen_values[i]), eigen_vectors[:,i]) for i in
range(len(eigen_values))]
pairs = sorted(pairs, key=lambda x: x[0], reverse=True)
for pair in pairs:
    print(pair[0])
```

```
237.46123198302223
46.982859387586956
1.4228571271832464e-14
1.1116232351620357e-14
1.1116232351620357e-14
1.0911610291170709e-14
1.0911610291170709e-14
9.093864818295084e-15
7.105427357601002e-15
4.034740044409669e-15
4.034740044409669e-15
3.246780533526568e-15
1.875490237492733e-15
```

We express it as a percentage.

```
eigen_value_sums = sum(eigen_values)
print('Explained Variance')
for i, pair in enumerate(pairs):
    print('Eigenvector {}: {}'.format(i,
(pair[0]/eigen_value_sums).real))
```

```
Explained Variance
Eigenvector 0: 0.834825679938727
Eigenvector 1: 0.1651743200612729
Eigenvector 2: 5.002238296907953e-17
Eigenvector 3: 3.908055287088648e-17
Eigenvector 4: 3.908055287088648e-17
Eigenvector 5: 3.836117754667543e-17
Eigenvector 6: 3.836117754667543e-17
Eigenvector 7: 3.1970658186203857e-17
Eigenvector 8: 2.4980049061181467e-17
Eigenvector 9: 1.4184650575682753e-17
Eigenvector 10: 1.4184650575682753e-17
Eigenvector 11: 1.1414476981686563e-17
Eigenvector 12: 6.593528550568873e-18
```

We create a matrix W with the first two eigen vectors. After that, we save the dot product of X and W into a new matrix.

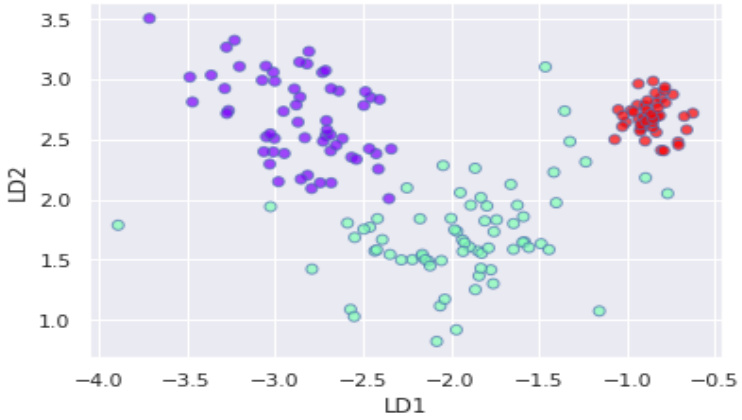
```
w_matrix = np.hstack((pairs[0][1].reshape(13,1),
]pairs[1][1].reshape(13,1))).real
```

```
X_lda = np.array(X.dot(w_matrix))
```

Since *matplotlib* cannot handle categorical variables directly, we encode every class as a number so that we can easily plot the class labels on the graph.

```
le = LabelEncoder()
y = le.fit_transform(df['class'])

plt.xlabel('LD1')
plt.ylabel('LD2')
plt.scatter(
    X_lda[:,0],
    X_lda[:,1],
    c=y,
    cmap='rainbow',
    alpha=0.7,
    edgecolors='b'
)
```

```
<matplotlib.collections.PathCollection at 0x7f8d8868a290>
```



To create the required model, we split the data into training and testing sets.

```
X_train, X_test, y_train, y_test = train_test_split(X_lda, y,
random_state=1)
```

We build and train the model using the Decision Tree algorithm. After predicting the category of every sample in the test set, we create a confusion matrix that will help evaluate the performance of the model.

```
dt = DecisionTreeClassifier()
dt.fit(X_train, y_train)
y_pred = dt.predict(X_test)
confusion_matrix(y_test, y_pred)
```

```
array([[18,  0,  0],
       [ 0, 17,  0],
       [ 0,  0, 10]])
```

We can conclude that our model produced accurate results and correctly classified all the samples present in the test dataset.

EXPERIMENT - 08

Aim : Support Vector Machine (SVM) models are a powerful tool to identify predictive models or classifiers, not only because they accommodate well sparse data but also because they can classify groups or create predictive rules for data that cannot be classified by linear decision functions. In this experiment you have to perform SVM on popular digits dataset

Description :

Support Vector Machines is considered to be a classification approach, it but can be employed in both types of classification and regression problems. It can easily handle multiple continuous and categorical variables. SVM constructs a hyperplane in multidimensional space to separate different classes. SVM generates optimal hyperplane in an iterative manner, which is used to minimize an error. The core idea of SVM is to find a maximum marginal hyperplane(MMH) that best divides the dataset into classes.

Data Set Description:

Database of handwritten digits, available from this page, has a training set of 60,000 examples, and a test set of 10,000 examples. The digits have been size-normalized and centered in a fixed-size image.
It consists of 785 columns
The pixels are given as the parameters and classification is performed on the pixels values of the images.

We'll divide the analysis into the following parts:
Data understanding and cleaning
Data preparation for model building
Building an SVM model - hyperparameter tuning, model evaluation

Code:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn import linear_model
from sklearn.model_selection import train_test_split
import gc
import cv2
```

Load the dataset

```
digits = pd.read_csv("../input/mnist-svm-m4/train.csv")
digits.head()
```

	label	pixel0	pixel1	pixel2	pixel3	pixel4	pixel5	pixel6	pixel7	pixel8
0	1	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0
2	1	0	0	0	0	0	0	0	0	0
3	4	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0	0

```
# visualise the array
print(four[5:-5, 5:-5])
```

```
[[  0 220 179   6   0   0   0   0   0   0   0   0   9  77   0   0   0   0]
 [  0  28 247  17   0   0   0   0   0   0   0   0  27 202   0   0   0   0]
 [  0   0 242 155   0   0   0   0   0   0   0   0  27 254  63   0   0   0]
 [  0   0 160 207   6   0   0   0   0   0   0   0  27 254  65   0   0   0]
 [  0   0 127 254  21   0   0   0   0   0   0   0  20 239  65   0   0   0]
 [  0   0  77 254  21   0   0   0   0   0   0   0   0 195  65   0   0   0]
 [  0   0  70 254  21   0   0   0   0   0   0   0   0 195 142   0   0   0]
 [  0   0  56 251  21   0   0   0   0   0   0   0   0 195 227   0   0   0]
 [  0   0   0 222 153   5   0   0   0   0   0   0   0 120 240  13   0   0]
```

```
# Summarise the counts of 'label' to see how many labels of each
digit are present
digits.label.astype('category').value_counts()
```

```
1    4684
7    4401
3    4351
9    4188
2    4177
6    4137
0    4132
4    4072
8    4063
5    3795
Name: label, dtype: int64
```

```
# Summarise count in terms of percentage
100*(round(digits.label.astype('category').value_counts()/len(digits
.index), 4))
```

```
1    11.15
7    10.48
3    10.36
9     9.97
2     9.95
6     9.85
0     9.84
4     9.70
8     9.67
5     9.04
Name: label, dtype: float64
```

Thus, each digit/label has an approximately 9%-11% fraction in the dataset and the **dataset is balanced**. This is an important factor in considering the choices of models to be used, especially SVM, since **SVMs rarely perform well on imbalanced data**

```
# missing values - there are none
digits.isnull().sum()
```

```
label       0
```

```
pixel0      0
pixel1      0
pixel2      0
pixel3      0
pixel4      0
pixel5      0
pixel6      0
pixel7      0
pixel8      0
pixel9      0
pixel10     0
pixel11     0
```

Also, let's look at the average values of each column, since we'll need to do some rescaling in case the ranges vary too much.

```
# average values/distributions of features
description = digits.describe()
description
```

	label	pixel0	pixel1	pixel2	pixel3	pixel4	pixel5
count	42000.000000	42000.0	42000.0	42000.0	42000.0	42000.0	42000
mean	4.456643	0.0	0.0	0.0	0.0	0.0	0.0
std	2.887730	0.0	0.0	0.0	0.0	0.0	0.0
min	0.000000	0.0	0.0	0.0	0.0	0.0	0.0
25%	2.000000	0.0	0.0	0.0	0.0	0.0	0.0
50%	4.000000	0.0	0.0	0.0	0.0	0.0	0.0
75%	7.000000	0.0	0.0	0.0	0.0	0.0	0.0
max	9.000000	0.0	0.0	0.0	0.0	0.0	0.0

Model building

```
# Creating training and test sets
# Splitting the data into train and test
X = digits.iloc[:, 1:]
Y = digits.iloc[:, 0]

# Rescaling the features
from sklearn.preprocessing import scale
X = scale(X)

# train test split with train_size=10% and test size=90%
x_train, x_test, y_train, y_test = train_test_split(X, Y,
train_size=0.10, random_state=101)
print(x_train.shape)
print(x_test.shape)
print(y_train.shape)
print(y_test.shape)
(4200, 784)
(37800, 784)
(4200,)
(37800,)
```

Linear SVM

```
from sklearn import svm
from sklearn import metrics

# an initial SVM model with linear kernel
```

```
svm_linear = svm.SVC(kernel='linear')

# fit
svm_linear.fit(x_train, y_train)
# predict
predictions = svm_linear.predict(x_test)
predictions[:10]
array([1, 3, 0, 0, 1, 9, 1, 5, 0, 6])

confusion = metrics.confusion_matrix(y_true = y_test, y_pred =
predictions)
confusion
metrics.accuracy_score(y_true=y_test, y_pred=predictions)
0.9042592592592592

# class-wise accuracy
class_wise = metrics.classification_report(y_true=y_test,
y_pred=predictions)
print(class_wise)
precision    recall  f1-score   support

           0       0.94      0.97      0.95      3715
           1       0.94      0.98      0.96      4185
           2       0.89      0.89      0.89      3790
           3       0.88      0.87      0.87      3900
           4       0.88      0.92      0.90      3702
```

Non linear svm

```
svm_rbf = svm.SVC(kernel='rbf')
svm_rbf.fit(x_train, y_train)
# predict
predictions = svm_rbf.predict(x_test)

# accuracy
print(metrics.accuracy_score(y_true=y_test, y_pred=predictions))
0.9255820105820106

from sklearn.model_selection import GridSearchCV

parameters = {'C':[1, 10, 100],
              'gamma': [1e-2, 1e-3, 1e-4]}

# instantiate a model
svc_grid_search = svm.SVC(kernel="rbf")

# create a classifier to perform grid search
clf = GridSearchCV(svc_grid_search, param_grid=parameters,
scoring='accuracy',return_train_score=True)

# fit
clf.fit(x_train, y_train)
# results
cv_results = pd.DataFrame(clf.cv_results_)
cv_results
```

	mean_fit_time	std_fit_time	mean_score_time	std_score_time	param_C	param_gamma	params	split0_test_score	split1_test_score	split2_test_score	mea
0	17.011370	0.100672	5.054957	0.047900	1	0.01	{'C': 1, 'gamma': 0.01}	0.723450	0.707857	0.712241	0.71
1	5.765255	0.058872	3.262784	0.039889	1	0.001	{'C': 1, 'gamma': 0.001}	0.919458	0.905714	0.914102	0.91
2	8.523973	0.065278	4.400614	0.059780	1	0.0001	{'C': 1, 'gamma': 0.0001}	0.867427	0.864286	0.872584	0.86
3	17.429509	0.012821	5.097013	0.057419	10	0.01	{'C': 10, 'gamma': 0.01}	0.742694	0.727143	0.732283	0.73
4	4.992204	0.052988	2.958831	0.019102	10	0.001	{'C': 10, 'gamma': 0.001}	0.937990	0.914286	0.916249	0.92
5	3.955956	0.070358	2.750300	0.041819	10	0.0001	{'C': 10, 'gamma': 0.0001}	0.920171	0.906429	0.913386	0.91
6	17.424671	0.019572	5.105521	0.054018	100	0.01	{'C': 100, 'gamma': 0.01}	0.742694	0.727143	0.732283	0.73
7	5.001782	0.065482	3.025350	0.038128	100	0.001	{'C': 100, 'gamma': 0.001}	0.937277	0.912857	0.916965	0.92
							{'C':				

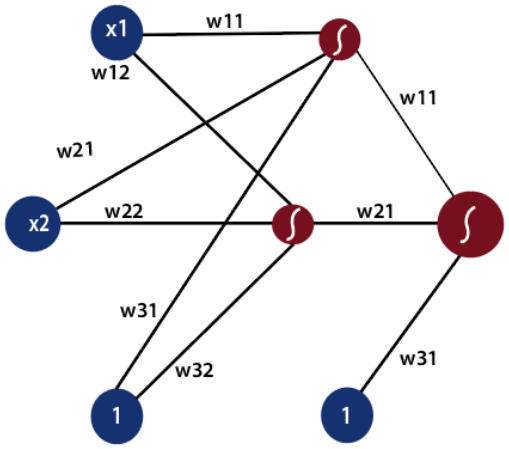
```
# converting C to numeric type for plotting on x-axis
cv_results['param_C'] = cv_results['param_C'].astype('int')

# optimal hyperparameters
best_C = 1
best_gamma = 0.001

# model
svm_final = svm.SVC(kernel='rbf', C=best_C, gamma=best_gamma)
# fit
svm_final.fit(x_train, y_train)
# predict
predictions = svm_final.predict(x_test)
# evaluation: CM
confusion = metrics.confusion_matrix(y_true = y_test, y_pred =
predictions)

# measure accuracy
test_accuracy = metrics.accuracy_score(y_true=y_test,
y_pred=predictions)
print(test_accuracy, "\n")
print(confusion)
0.924973544973545

[[3587    0   10   10    5   15   50   12   25    1]
 [   0 4108   14   16    5    3    6   18   10    5]
 [  24   23 3407   65   44    5   36  123   54    9]
 [   4   21   86 3502    5   89   11   73   76   33]
 [   3   11   36    7 3450   13   23   43    6  110]
 [  20   29   14  114   18 3020   79   53   36   35]
 [  31   12   11    1   14   34 3521   44   25    0]
 [   4   28   27    8   36    7    1 3739    7   97]
 [  14   59   32   80   22   97   25   44 3251   41]
 [  23   13   13   50   98    7    0  176   19 3379]]
```

The final accuracy on test data is approx. 92%. Note that this can be significantly increased by using the entire training data of 42,000 images (we have used just 10% of that!).

EXPERIMENT - 09

Aim : Neural networks are one of the important techniques of machine learning because it makes computer take intelligent decisions. Feature extraction also can be done automatically in neural networks. Two important phases of neural networks are feed forward and backward propagations. In this experiment you are going to write the step by step code for feed forward neural networks.

Description:



The process of receiving an input to produce some kind of output to make some kind of prediction is known as Feed Forward."

Feed Forward neural network is the core of many other important neural networks such as convolution neural network.

In the feed-forward neural network, there are not any feedback loops or connections in the network. Here is simply an input layer, a hidden layer, and an output layer.

There can be multiple hidden layers which depend on what kind of data you are dealing with. Input layer first provides the neural network with data and the output layer then make predictions on that data which is based on a series of functions. ReLU Function is the most commonly used activation function in the deep neural network.

Code for Forward feed in network:

```
# Initialize a network
def initialize_network(n_inputs, n_hidden, n_outputs):
      network = list()
      hidden_layer = [{'weights':[random() for i in range(n_inputs +
1)]} for i in range(n_hidden)]
      network.append(hidden_layer)
      output_layer = [{'weights':[random() for i in range(n_hidden +
1)]} for i in range(n_outputs)]
      network.append(output_layer)
      return network
```

```
from random import seed
from random import random

# Initialize a network
def initialize_network(n_inputs, n_hidden, n_outputs):
      network = list()
      hidden_layer = [{'weights':[random() for i in range(n_inputs +
1)]} for i in range(n_hidden)]
      network.append(hidden_layer)
      output_layer = [{'weights':[random() for i in range(n_hidden +
1)]} for i in range(n_outputs)]
      network.append(output_layer)
      return network

seed(1)
network = initialize_network(2, 1, 2)
for layer in network:
      print(layer)
[{'weights': [0.13436424411240122, 0.8474337369372327,
0.763774618976614]}]
[{'weights': [0.2550690257394217, 0.49543508709194095]}, {'weights':
[0.4494910647887381, 0.651592972722763]}]
```

Foward Propogate

We can calculate an output from a neural network by propagating an input signal through each layer until the output layer outputs its values. We can break forward propagation down into three parts:
1.Neuron Activation. 2.Neuron Transfer. 3.Forward Propagation.
Neuron Activation Neuron activation is calculated as the weighted sum of the inputs. Much like linear regression.
activation = sum(weight_i * input_i) + bias

```
# Calculate neuron activation for an input
def activate(weights, inputs):
      activation = weights[-1]
      for i in range(len(weights)-1):
            activation += weights[i] * inputs[i]
      return activation
```

Neuron Transfer Different transfer functions can be used. It is traditional to use the sigmoid activation function, but you can also use the tanh (hyperbolic tangent) function to transfer outputs. More recently, the rectifier transfer function has been popular with large deep learning networks.
Sigmoid Function :
output = 1 / (1 + e^(-activation))

```
def transfer(activation):
      return 1.0 / (1.0 + exp(-activation))
```

forward_propagate() that implements the forward propagation for a row of data from our dataset with our neural network. We collect the outputs for a layer in an array named new_inputs that becomes the array inputs and is used as inputs for the

following layer.
The function returns the outputs from the last layer also called the output layer.

```
# Forward propagate input to a network output
def forward_propagate(network, row):
	inputs = row
	for layer in network:
		new_inputs = []
		for neuron in layer:
			activation = activate(neuron['weights'], inputs)
			neuron['output'] = transfer(activation)
			new_inputs.append(neuron['output'])
		inputs = new_inputs
	return inputs
```

Test forward propogation

```
from math import exp

# Calculate neuron activation for an input
def activate(weights, inputs):
	activation = weights[-1]
	for i in range(len(weights)-1):
		activation += weights[i] * inputs[i]
	return activation


# Transfer neuron activation
def transfer(activation):
	return 1.0 / (1.0 + exp(-activation))


# Forward propagate input to a network output
def forward_propagate(network, row):
	inputs = row
	for layer in network:
		new_inputs = []
		for neuron in layer:
			activation = activate(neuron['weights'], inputs)
			neuron['output'] = transfer(activation)
			new_inputs.append(neuron['output'])
		inputs = new_inputs
	return inputs

# test forward propagation
network = [[{'weights': [0.13436424411240122, 0.8474337369372327,
0.763774618976614]}],
		[{'weights': [0.2550690257394217, 0.49543508709194095]},
{'weights': [0.4494910647887381, 0.651592972722763]}]]
row = [1, 0, None]
output = forward_propagate(network, row)

print(output)
[0.6629970129852887, 0.7253160725279748]
```

EXPERIMENT - 10

Aim : Neural networks are one of the important techniques of machine learning because it makes computer take intelligent decisions. Feature extraction also can be done automatically in neural networks. Two important phases of neural networks are feed forward and backward propagations. In this experiment you are going to write the step by step code for backward propagation of neural networks. This is one of the important features of machine learning to make computer adjust error automatically to predict output more accurate.

The Back propagation algorithm is a supervised learning method for multilayer feed-forward networks from the field of Artificial Neural Networks.
For a single training example, Backpropagation algorithm calculates the gradient of the error function.Backpropagation algorithms are a set of methods used to efficiently train artificial neural networks following a gradient descent approach which exploits the chain rule.

The principle of the backpropagation approach is to model a given function by modifying internal weightings of input signals to produce an expected output signal. The system is trained using a supervised learning method, where the error between the system's output and a known expected output is presented to the system and used to modify its internal state.

Dataset

The seeds dataset involves the prediction of species given measurements seeds from different varieties of wheat.

There are 201 records and 7 numerical input variables. It is a classification problem with 3 output classes. The scale for each numeric input value vary, so some data normalization may be required for use with algorithms that weight inputs like the backpropagation algorithm.

Below is a sample of the first 5 rows of the dataset.

```
15.26,14.84,0.871,5.763,3.312,2.221,5.22,1
14.88,14.57,0.8811,5.554,3.333,1.018,4.956,1
14.29,14.09,0.905,5.291,3.337,2.699,4.825,1
13.84,13.94,0.8955,5.324,3.379,2.259,4.805,1
16.14,14.99,0.9034,5.658,3.562,1.355,5.175,1
```

Back propagation algorithm, first initializing a network, training it on the training dataset and then using the trained network to make predictions on a test dataset.

```
# Backprop on the Seeds Dataset
```

Import the packages and libraries required

```
from random import seed
from random import randrange
from random import random
from csv import reader
from math import exp
#Load a CSV file

def load_csv(filename):
	dataset = list()
	with open(filename, 'r') as file:
		csv_reader = reader(file)
		for row in csv_reader:
			if not row:
				continue
			dataset.append(row)
	return dataset

# Convert string column to float
def str_column_to_float(dataset, column):
	for row in dataset:
		row[column] = float(row[column].strip())

# Convert string column to integer
def str_column_to_int(dataset, column):
	class_values = [row[column] for row in dataset]
	unique = set(class_values)
	lookup = dict()
	for i, value in enumerate(unique):
		lookup[value] = i
	for row in dataset:
		row[column] = lookup[row[column]]
	return lookup
#Find the min and max values for each column

def dataset_minmax(dataset):
	minmax = list()
	stats = [[min(column), max(column)] for column in
zip(*dataset)]
	return stats

# Rescale dataset columns to the range 0-1
def normalize_dataset(dataset, minmax):
	for row in dataset:
		for i in range(len(row)-1):
			row[i] = (row[i] - minmax[i][0]) / (minmax[i][1] -
minmax[i][0])

#Split a dataset into k folds

def cross_validation_split(dataset, n_folds):
	dataset_split = list()
```

```
	dataset_copy = list(dataset)
	fold_size = int(len(dataset) / n_folds)
	for i in range(n_folds):
		fold = list()
		while len(fold) < fold_size:
			index = randrange(len(dataset_copy))
			fold.append(dataset_copy.pop(index))
		dataset_split.append(fold)
	return dataset_split

#calculate accuracy percentage
def accuracy_metric(actual, predicted):
	correct = 0
	for i in range(len(actual)):
		if actual[i] == predicted[i]:
			correct += 1
	return correct / float(len(actual)) * 100.0

#Evaluate an algorithm using a cross validation split

def evaluate_algorithm(dataset, algorithm, n_folds, *args):
	folds = cross_validation_split(dataset, n_folds)
	scores = list()
	for fold in folds:
		train_set = list(folds)
		train_set.remove(fold)
		train_set = sum(train_set, [])
		test_set = list()
		for row in fold:
			row_copy = list(row)
			test_set.append(row_copy)
			row_copy[-1] = None
		predicted = algorithm(train_set, test_set, *args)
		actual = [row[-1] for row in fold]
		accuracy = accuracy_metric(actual, predicted)
		scores.append(accuracy)
	return scores

#Calculate neuron activation for an input

def activate(weights, inputs):
	activation = weights[-1]
	for i in range(len(weights)-1):
		activation += weights[i] * inputs[i]
	return activation

# Transfer neuron activation
def transfer(activation):
	return 1.0 / (1.0 + exp(-activation))

#Forward Feed

# Forward propagate input to a network output
def forward_propagate(network, row):
```

```
	inputs = row
	for layer in network:
		new_inputs = []
		for neuron in layer:
			activation = activate(neuron['weights'], inputs)
			neuron['output'] = transfer(activation)
			new_inputs.append(neuron['output'])
		inputs = new_inputs
	return inputs
# Calculate the derivative of an neuron output
def transfer_derivative(output):
	return output * (1.0 - output)

# Backpropagate error and store in neurons
def backward_propagate_error(network, expected):
	for i in reversed(range(len(network))):
		layer = network[i]
		errors = list()
		if i != len(network)-1:
			for j in range(len(layer)):
				error = 0.0
				for neuron in network[i + 1]:
					error += (neuron['weights'][j] *
neuron['delta'])
				errors.append(error)
		else:
			for j in range(len(layer)):
				neuron = layer[j]
				errors.append(neuron['output'] -
expected[j])
		for j in range(len(layer)):
			neuron = layer[j]
			neuron['delta'] = errors[j] *
transfer_derivative(neuron['output'])

# Update network weights with error
def update_weights(network, row, l_rate):
	for i in range(len(network)):
		inputs = row[:-1]
		if i != 0:
			inputs = [neuron['output'] for neuron in network[i - 1]]
		for neuron in network[i]:
			for j in range(len(inputs)):
				neuron['weights'][j] -= l_rate * neuron['delta'] *
inputs[j]
			neuron['weights'][-1] -= l_rate * neuron['delta']
```

```
#Train the network
# Train a network for a fixed number of epochs
def train_network(network, train, l_rate, n_epoch, n_outputs):
	for epoch in range(n_epoch):
		for row in train:
			outputs = forward_propagate(network, row)
			expected = [0 for i in range(n_outputs)]
			expected[row[-1]] = 1
			backward_propagate_error(network, expected)
			update_weights(network, row, l_rate)

#Initialize a network

def initialize_network(n_inputs, n_hidden, n_outputs):
	network = list()
	hidden_layer = [{'weights':[random() for i in range(n_inputs +
1)]} for i in range(n_hidden)]
	network.append(hidden_layer)
	output_layer = [{'weights':[random() for i in range(n_hidden +
1)]} for i in range(n_outputs)]
	network.append(output_layer)
	return network
# Make a prediction with a network
def predict(network, row):
	outputs = forward_propagate(network, row)
	return outputs.index(max(outputs))

#Backpropagation Algorithm With Stochastic Gradient Descent

def back_propagation(train, test, l_rate, n_epoch, n_hidden):
	n_inputs = len(train[0]) - 1
	n_outputs = len(set([row[-1] for row in train]))
	network = initialize_network(n_inputs, n_hidden, n_outputs)
	train_network(network, train, l_rate, n_epoch, n_outputs)
	predictions = list()
	for row in test:
		prediction = predict(network, row)
		predictions.append(prediction)
	return(predictions)

#Test Backprop on Seeds dataset
seed(1)

#Load and prepare data
filename = 'wheat-seeds.csv'
dataset = load_csv(filename)
for i in range(len(dataset[0])-1):
	str_column_to_float(dataset, i)
# convert class column to integers
str_column_to_int(dataset, len(dataset[0])-1)
# normalize input variables
minmax = dataset_minmax(dataset)
```

```
normalize_dataset(dataset, minmax)
```
#Evaluate algorithm

```
n_folds = 5
l_rate = 0.3
n_epoch = 500
n_hidden = 5
scores = evaluate_algorithm(dataset, back_propagation, n_folds,
l_rate, n_epoch, n_hidden)
print('Scores: %s' % scores)
print('Mean Accuracy: %.3f%%' % (sum(scores)/float(len(scores))))
Scores: [92.85714285714286, 92.85714285714286, 97.61904761904762,
95.23809523809523, 88.09523809523809]
Mean Accuracy: 93.333%
```