# PRACTICAL-1

**AIM:**

Perform the following using Python Pandas and Matplotlib library on given dataset:

1. Deal with missing values in the data either by deleting records or using mean/median/mode imputation.
2. Detect if Outliers exist and plot the data distribution using Box Plots, Scatter Plots and Histograms of matplotlib library

iii) Create and display the correlation matrix of all features of the data. Record and Analyse Observations.

**CODE:**

#IMPORTING LIBRARIES

import pandas as pd

import matplotlib.pyplot as plt

import sklearn

import matplotlib.pyplot as plt

import seaborn as sns

#GETTING THE DATA-SET

df = pd.read\_csv('cwurData.csv')

#BOX PLOT USING SEABORN

sns.boxplot(df['broad\_impact'])

#SCATTERPLOT USING MATPLOTLIB

fig, plot = plt.subplots(figsize = (15,15))

plot.scatter(df['world\_rank'], df['quality\_of\_education'])

#X-LABEL

plot.set\_xlabel('WORLD RANKS')

#Y-LABEL

plot.set\_ylabel('QUALITY OF EDUCATION')

plt.show()

plt.boxplot(df['score'])

plt.figure(figsize=(12,5))

#HISTOGRAM USING MATPLOTLIB

plt.hist(df['country'])

plt.xticks(rotation = 90)

plt.show()

#CORRELATION

df.corr()

corrmat = df.corr()

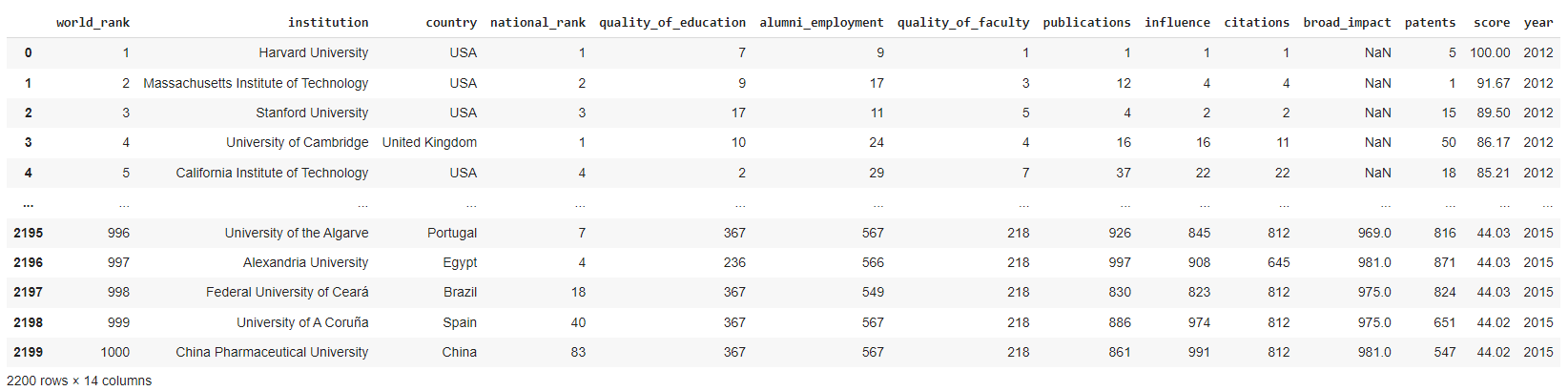
#HEATMAP

sns.heatmap(corrmat,annot=True)

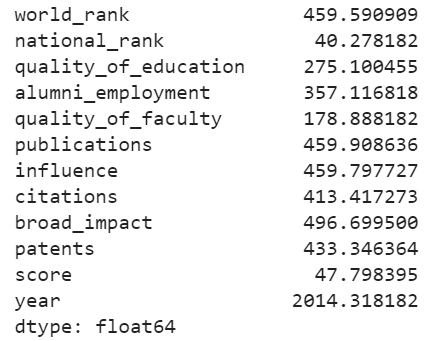
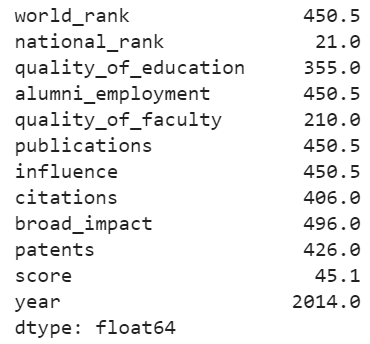
sns.heatmap(corrmat,annot=False)

sns.pairplot(df)

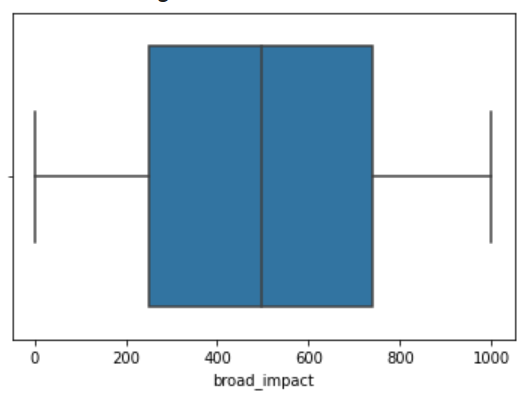
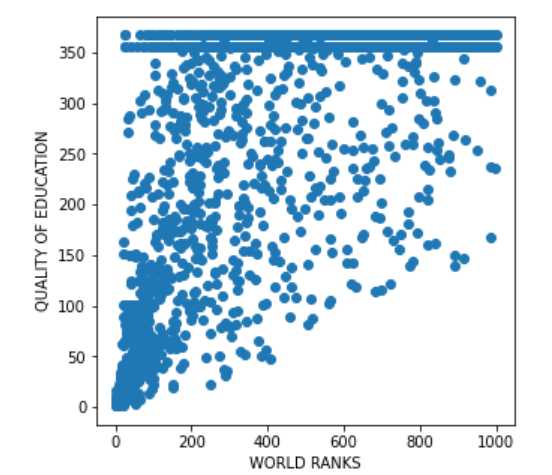
**OUTPUT:**



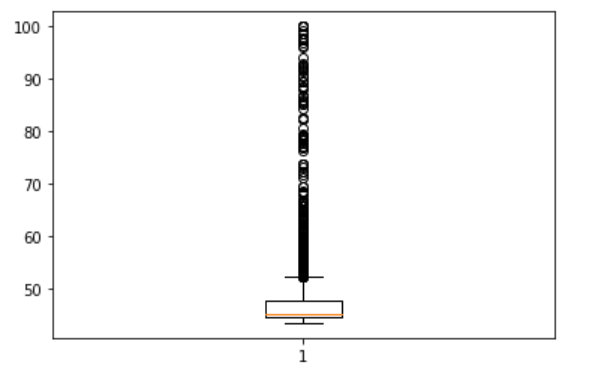
*GLIMPSES OF DATASET*

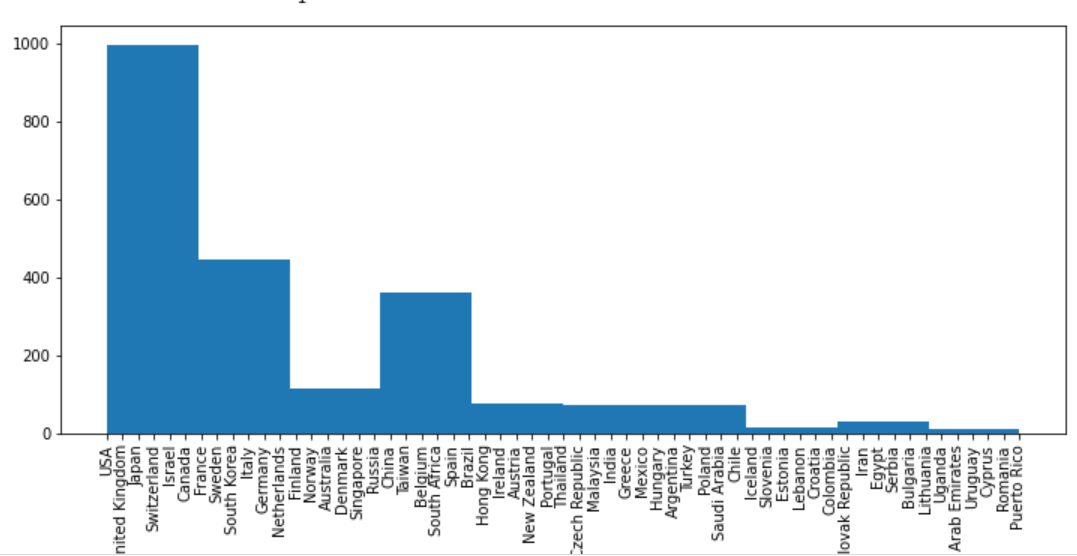
*Mean Median*



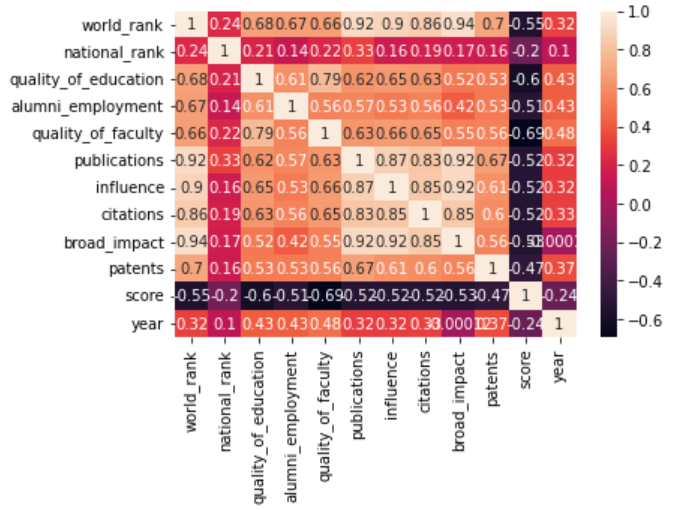
*Scatter plot Boxplot*



*Boxplot of score column*



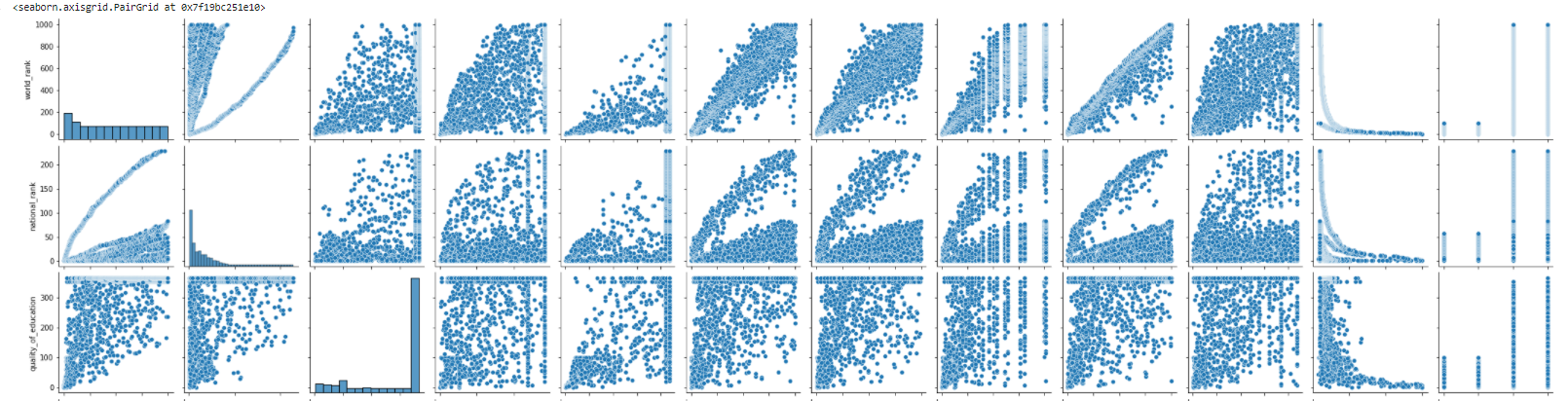
*Histogram*

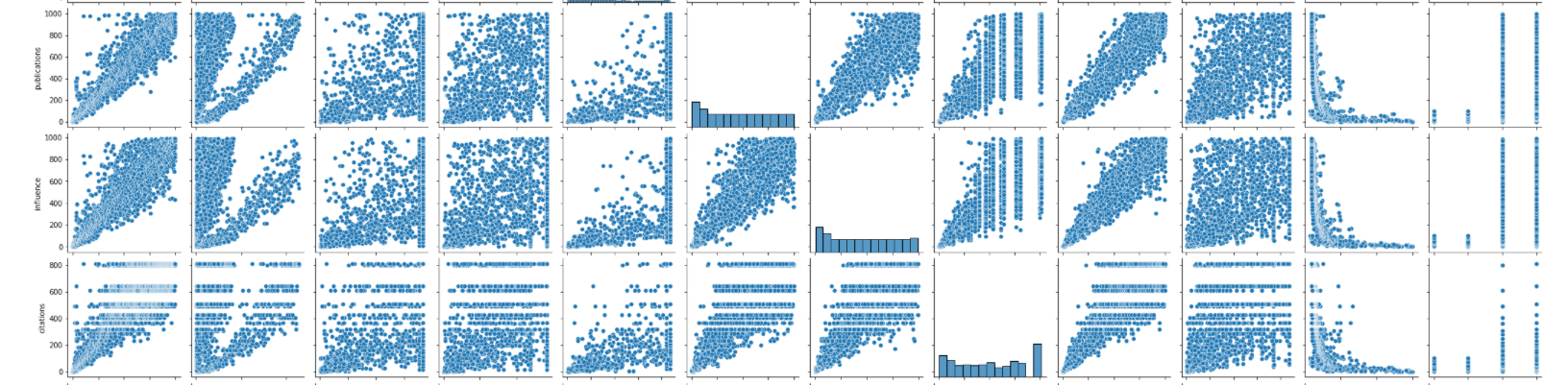


*Correlation heatmap*



*Correlation matrix*





*Pairplot*

# PRACTICAL-2

**AIM:**

For given Dataset (you may continue to use the same processed dataset from experiment 1 only for this experiment), perform the following using Python Pandas and scikit-learn library or by writing your own user-defined function:

1. Perform Data Standardization and Normalization
2. Select the 10 best features of the data using different statistical scoring methods. (Hint: Chi-Squared Statistical Test is a good scoring method)
3. Split the data into training and testing sets in a ratio of 80:20.

**CODE:**

#IMPORTING LIBRARIES

import pandas as pd

import matplotlib.pyplot as plt

from sklearn.preprocessing import StandardScaler

from sklearn.preprocessing import Normalizer

from sklearn.model\_selection import train\_test\_split

from sklearn.feature\_selection import SelectKBest

from sklearn.feature\_selection import chi2

from sklearn.preprocessing import MinMaxScaler

#GETTING THE DATASET

df = pd.read\_csv('winequality-red.csv',sep=';')

y=df['quality']

X = df.drop('quality',inplace=False,axis=1)

# SPLITTING THE DATASET INTO TESTING AND TRAINING SET

X\_train,X\_test,y\_train,y\_test= train\_test\_split(X,y,test\_size=0.2,random\_state=17)

sc\_X = StandardScaler()

sc\_X = sc\_X.fit\_transform(df)

keys = df.head(0).keys()

sc\_X = pd.DataFrame(data=sc\_X, columns=keys)

sc\_X

#STANDARDIZATION

scaler = StandardScaler().fit(X)

standardized\_X\_train = scaler.transform(X\_train)

standardized\_X\_test = scaler.transform(X\_test)

# Normalisation

scalerscaler = Normalizer()

Normalize\_x\_test = scalerscaler.transform(X\_train)

Normalize\_y\_test = scalerscaler.transform(X\_test)

print (Normalize\_x\_test)

print(df.isna().sum())

scaler = MinMaxScaler()

scaler.fit(df)

scaled\_features = scaler.transform(df)

df\_MinMax = pd.DataFrame(data=scaled\_features,columns=keys)

df\_MinMax

bestfeatures = SelectKBest(score\_func=chi2,k=3)

fit = bestfeatures.fit(X,y)

dfscores = pd.DataFrame(fit.scores\_)

dfcolumns = pd.DataFrame(X.columns)

featureScores = pd.concat([dfcolumns,dfscores],axis=1)

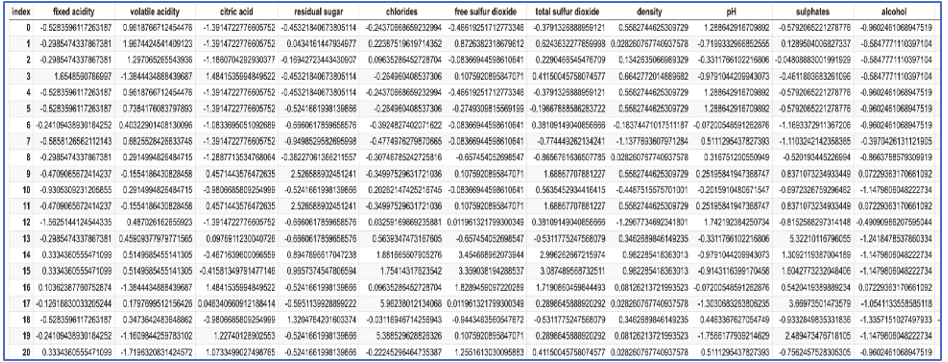
featureScores.columns = ['Column Names','Column Scores']

print(featureScores.nlargest(10,'Column Scores'))

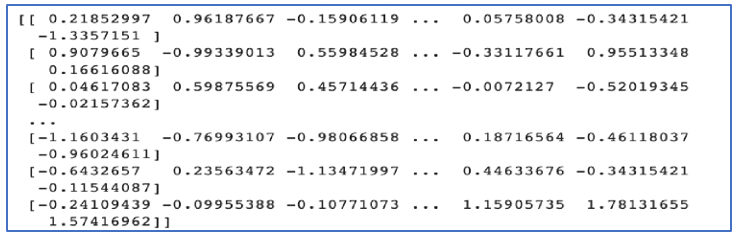
**OUTPUT:**



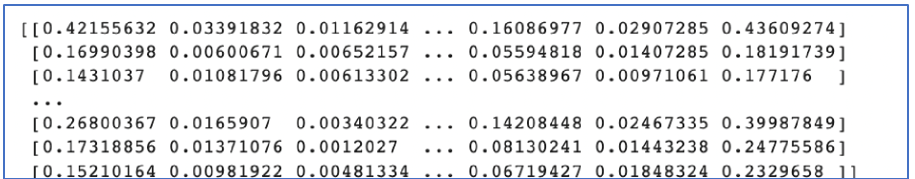
*Glimpses of Dataset*



*Data after min max scaler implementation*



*Standardisation*



*Normalisation*



*Best Feature Selection*

# PRACTICAL-3

**AIM:**

Implement the linear regression and calculate the different evaluation measure (MAE, RMSE etc.). for the same. Also implement gradient descent and observe the cost with linear regression using gradient descent. Do not use any Python library for linear regression. (Hint: Linear Regression Formula is Y= mX +b where Y is target

variable and X is independent variable)

**CODE:**

#PRACTICAL-3

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

#GETTING THE DATASET

df = pd.read\_csv('fifa\_players.csv')

df.head()

#Removing special character, "M" and "K" for the data column

df['Release Clause'] = df['Release Clause'].str.replace(r"€", "")

df['Release Clause'] = df['Release Clause'].str.replace(r"M", "")

df['Release Clause'] = df['Release Clause'].str.replace(r"K", "")

df['Release Clause'] = df['Release Clause'].astype(float)

x = df.Potential[:1000]

df['Release Clause'] = df['Release Clause'].fillna(0)

y = df['Release Clause'][:1000]

#SCATTER PLOT

plt.scatter(x,y, color='violet')

plt.show()

mean\_x = x.mean()  mean\_y = y.mean()

tempx = [(i - mean\_x)\*\*2 for i in x]

var\_x = sum(tempx, 0)/(len(tempx))

tempy = [(i - mean\_y)\*\*2 for i in y]

var\_y = sum(tempy, 0)/(len(tempy))

#PRINTING VARIANCE

print("Variance of x : ", var\_x)

print("Variance of y : ", var\_y)

temp1x = [(i - mean\_x) for i in x]

temp1y = [(i - mean\_y) for i in y]

total = 0

for i in range(len(temp1x)):

  for j in range(len(temp1y)):

    if i == j:

      total = total + temp1x[i]\*temp1y[j]

cov = total/(len(temp1x) - 1)

#PRINTING CO-VARIANCE

print("Covariance : ", cov)

r = cov/((var\_x\*var\_y)\*\*0.5)

slope = r\*((var\_y\*\*0.5)/(var\_x\*\*0.5))

c = mean\_y - slope \* mean\_x

print("Value of slope : ", slope)

print("Value of intercept : ", c)

predicted\_val = []

for i in range(len(x)):

  new\_y = slope\*x[i] + c

  predicted\_val.append(round(new\_y,2))

  print(round(new\_y, 2), y[i])

new\_meanY = sum(predicted\_val)

new\_diff = [(i - j)\*\*2 for i, j in zip(y, predicted\_val)]

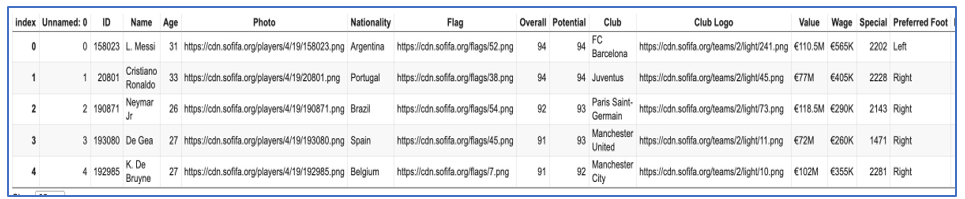
new\_sum = sum(new\_diff)

mse = new\_sum/len(new\_diff)

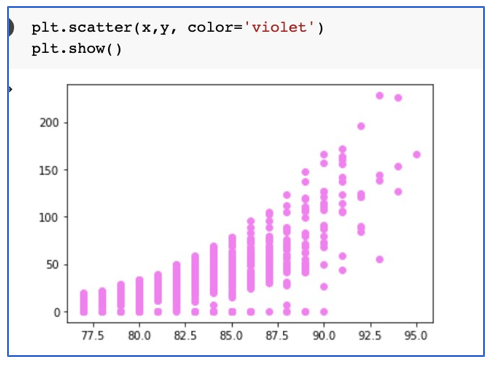
rmse = mse\*\*0.5 print("Mean squared error : ", mse)

print("Root mean squared error : ", rmse)

**OUTPUT:**



*Glimpses of Dataset*



*Scatter plot*



*Variance*



*Slope and Intercept*



*Covariance*



*MSE & RMSE*

**NON-LINEAR REGRESSION:**

**CODE:**

import numpy as np

import pandas as pd

df = pd.read\_csv("china\_gdp.csv")

df.head()

import matplotlib.pyplot as plt

plt.figure(figsize=(8,5))

x\_data, y\_data = (df["Year"].values, df["Value"].values)

plt.plot(x\_data, y\_data, 'ro')

plt.ylabel('GDP')

plt.xlabel('Year')

plt.show()

X = np.arange(-5.0, 5.0, 0.1)

Y = 1.0 / (1.0 + np.exp(-X))

plt.plot(X,Y)

plt.ylabel('Dependent Variable')

plt.xlabel('Indepdendent Variable')

plt.show()

def sigmoid(x, Beta\_1, Beta\_2):

     y = 1 / (1 + np.exp(-Beta\_1\*(x-Beta\_2)))

     return y

beta\_1 = 0.10

beta\_2 = 1990.0

#logistic function

Y\_pred = sigmoid(x\_data, beta\_1 , beta\_2)

#plot initial prediction against datapoints

plt.plot(x\_data, Y\_pred\*15000000000000.)

plt.plot(x\_data, y\_data, 'ro')

# Lets normalize our data

xdata =x\_data/max(x\_data)

ydata =y\_data/max(y\_data)

from scipy.optimize import curve\_fit

popt, pcov = curve\_fit(sigmoid, xdata, ydata)

# Now we plot our resulting regression model.

x = np.linspace(1960, 2015, 55)

x = x/max(x)

plt.figure(figsize=(8,5))

y = sigmoid(x, \*popt)

plt.plot(xdata, ydata, 'ro', label='data')

plt.plot(x,y, linewidth=3.0, label='fit')

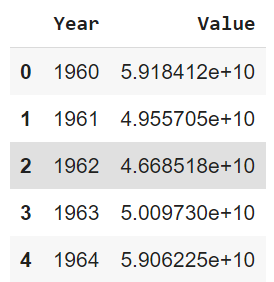
plt.legend(loc='best')

plt.ylabel('GDP')

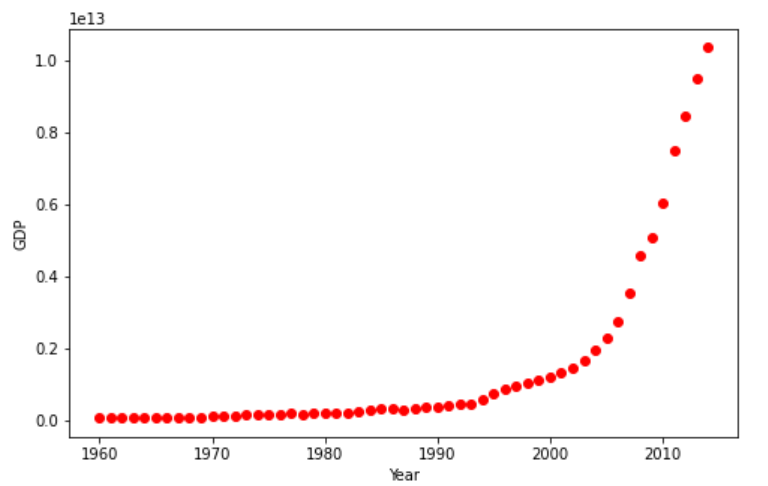
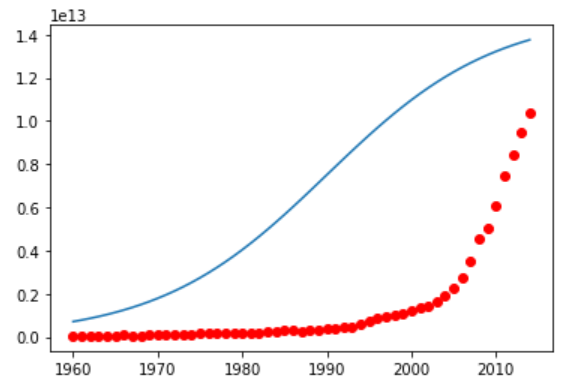
plt.xlabel('Year')

plt.show()

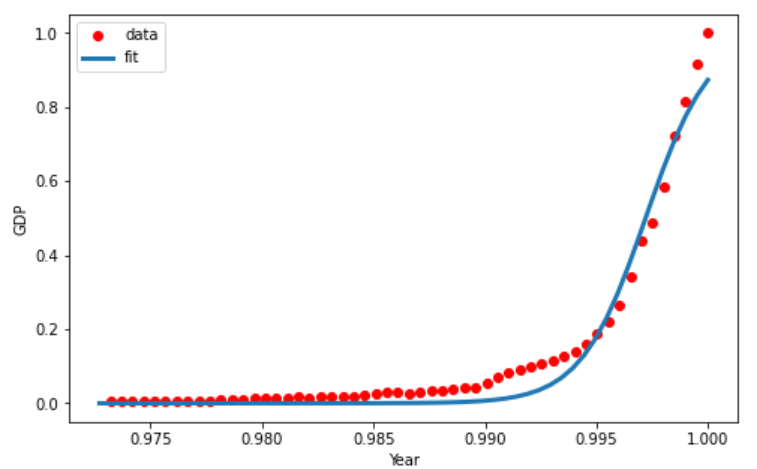
**OUTPUT:**



*Glimpses of Dataset*

*Plot of Dataset*  *Initial Prediction against data points*



*Graph after normalization*

# PRACTICAL-4

**AIM:**

Create Visual analysis for the given data set using MATLAB.

**BASIC THEORY:**

* The name MATLAB stands for **MATrix LABoratory.**
* It is written in C, C++, and Java.
* Matlab was initially released in 1984.
* MATLAB is a programming platform designed specifically for engineers and scientists to analyze and design systems and products that transform our world.
* Matlab helps to perform mathematical calculation, design, analysis and optimization (structural and mathematical), as well as gives speed, accuracy and precision to results.
* The basic data element of MATLAB as the name suggests is the Matrix or an array.

**Common applications of matlab:**

* Math and computation
* Algorithm development
* Modeling, simulation, and prototyping
* Data analysis, exploration, and visualization
* Scientific and engineering graphics
* Application development, including Graphical User Interface building

**CODE:**

X=[32,53,61,47,59,55,52,39,48,52,45,54,44,58,56,48,44,60,45,38]

Y=[31,68,62,71,87,78,79,59,75,71,55,82,62,75,81,60,82,97,48,56]

n=length(X)

denominator = ( ( n \* sum(X .\* X)) - sum(X) \* sum(X))

b = ( ( sum(Y) \* sum(X.\*X) ) - ( sum(X) .\* sum(X.\*Y) ) ) / denominator

m = ( n \* sum( X .\* Y ) - (sum(X) \* sum(Y) ) ) / denominator

yCalc1 = X.\*m + b

RGB = [255 0 0]/256

scatter(X,Y,[],RGB)

hold on

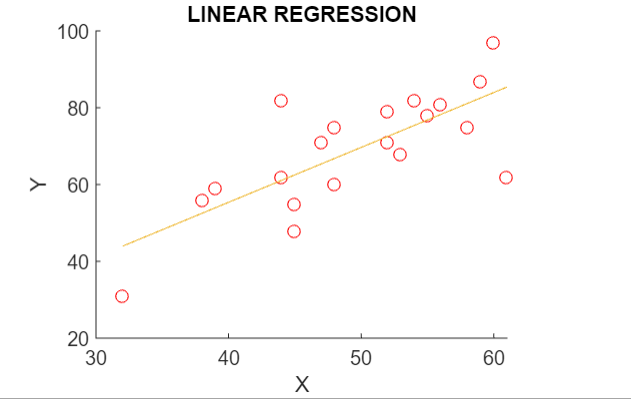
plot(X,yCalc1)

xlabel('X')

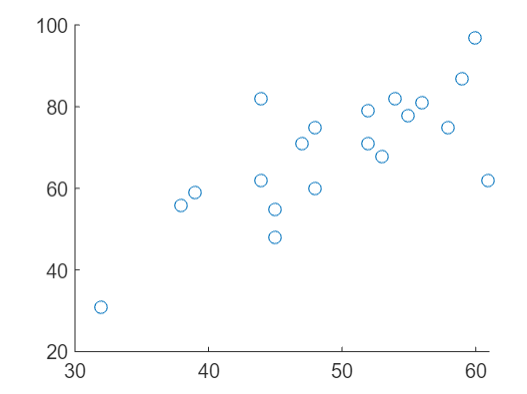
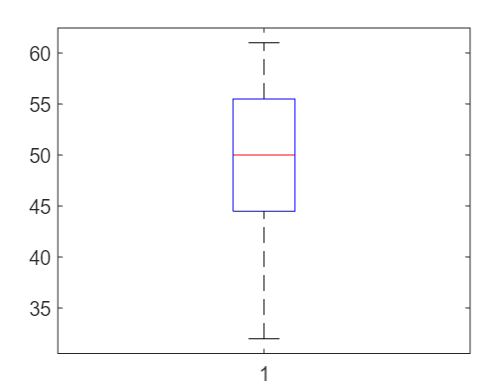
ylabel('Y')

title('LINEAR REGRESSION')

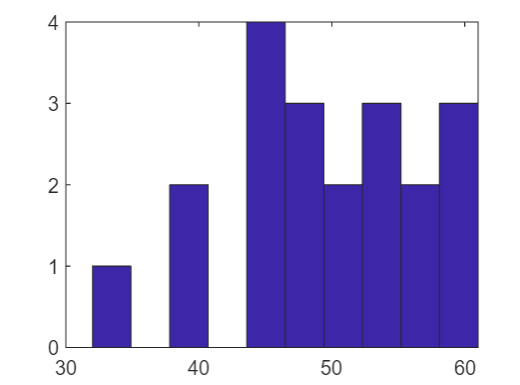
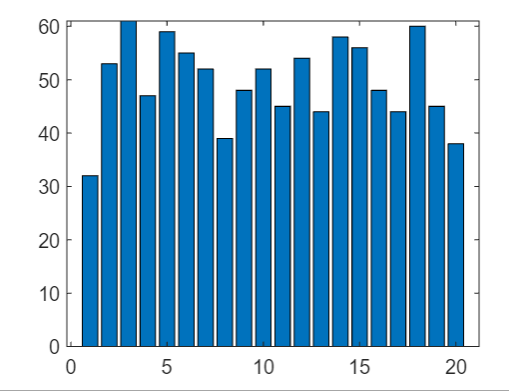
**OUTPUT:**



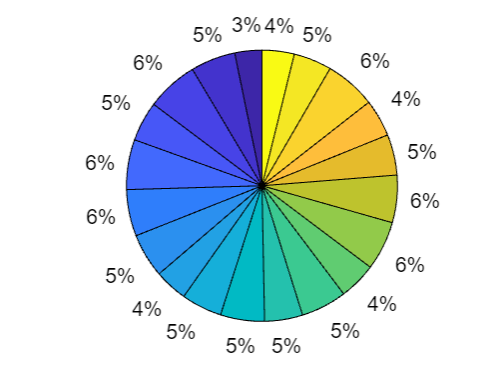
*Linear Regression Output*

*Scatter Plot Box Plot*

*Histogram Bar plot*



*Pie Chart*

# PRACTICAL-5

**AIM:**

Implement logistic regression and calculate the different evaluation measure (F-measures, Confusion Matrix etc.) for the same. Also implement gradient descent and observe the cost with logistic regression using gradient descent. (Hint: Confusion Matrix and Fmeasures involve use of True Negatives, True Positives, False Negatives and False Positives). Also implement Cross- Validation

**CODE:**

#PRACTICAL-5

import pandas as pd

from sklearn.linear\_model import LogisticRegression

from sklearn.preprocessing import Normalizer

from sklearn.preprocessing import StandardScaler

from sklearn.metrics import confusion\_matrix, ConfusionMatrixDisplay

from sklearn.metrics import precision\_recall\_fscore\_support

import matplotlib.pyplot as plt

#GETTING DATASET

df = pd.read\_excel('default of credit card clients.xls')

df1 = df.rename(columns=df.iloc[0]).drop(df.index[0])

df1

df1.drop('ID',inplace = True, axis = 1)

df1.head(2)

#DEFINING DEPENDENT AND INDEPENDENT VARIABLE

y = df1['default payment next month']

x = df1.drop('default payment next month',inplace =False,axis = 1)

#IMPLEMENTING LOGISTIC REGRESSION CLASSIFIER

y=y.astype('int')

classifier = LogisticRegression().fit(x, y)

classifier.score(x,y)

#IMPLEMENTING NORMALIZATION CLASSIFIER

transformer = Normalizer().fit(x)

x\_norm = transformer.transform(x)

x\_norm

classifier = LogisticRegression().fit(x\_norm,y)

classifier.score(x\_norm,y)

scaler = StandardScaler().fit(x)

std\_x=scaler.transform(x)

std\_x

cls=LogisticRegression().fit(std\_x,y)

cls.score(std\_x,y)

y\_pred=cls.predict(std\_x)

y\_pred

#BUILDING CONFUSION MATRIX

mat = confusion\_matrix(y,y\_pred)

print(mat)

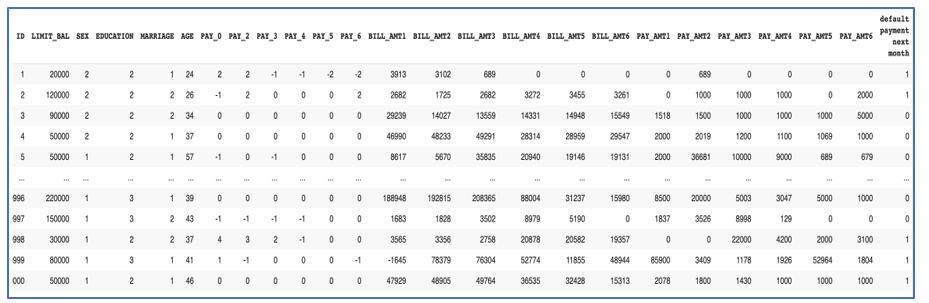
conf = ConfusionMatrixDisplay(mat)

conf.plot()

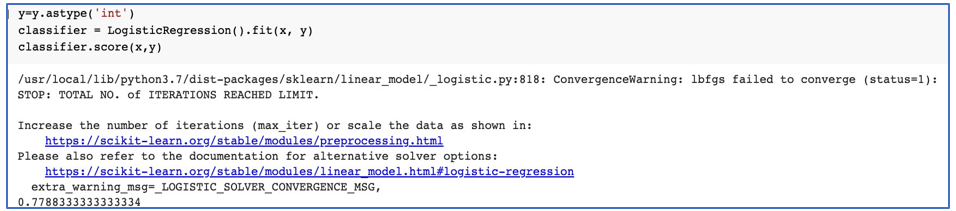
#FINDING OUT PRECISION,RECALL AND F1 SCORE

precision, recall, f1, support = precision\_recall\_fscore\_support(y,y\_pred)

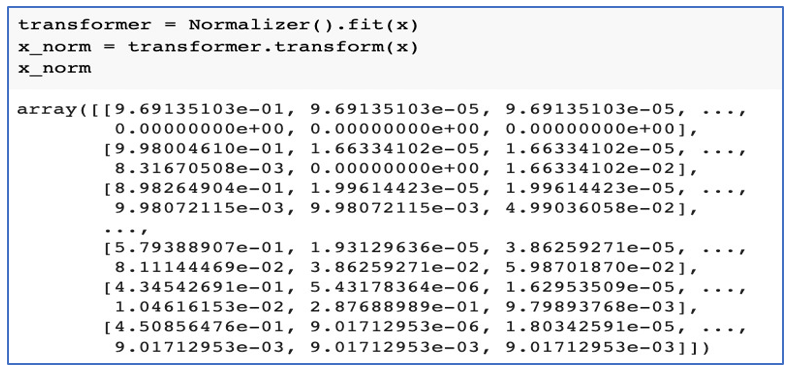
**OUTPUT:**



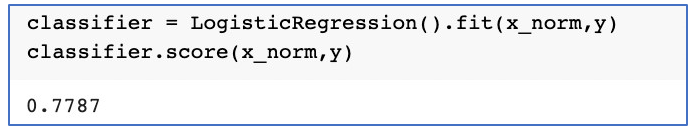
*Glimpses of Dataset*



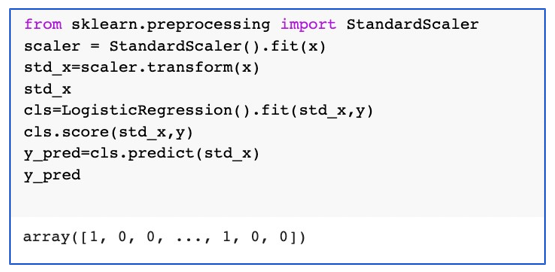
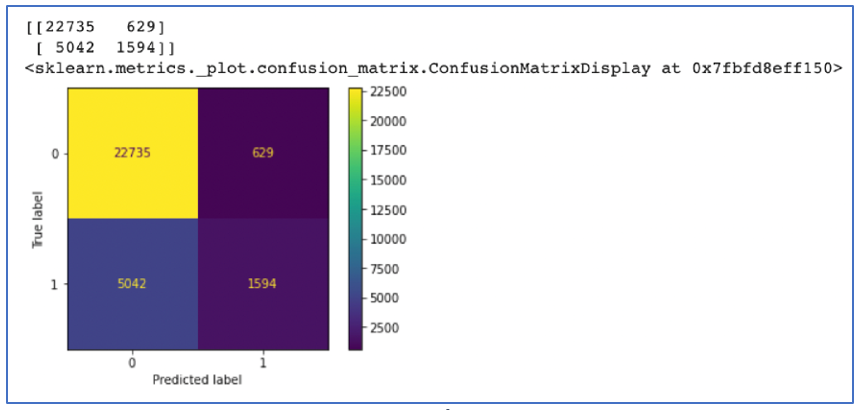
*Logistic Regression*



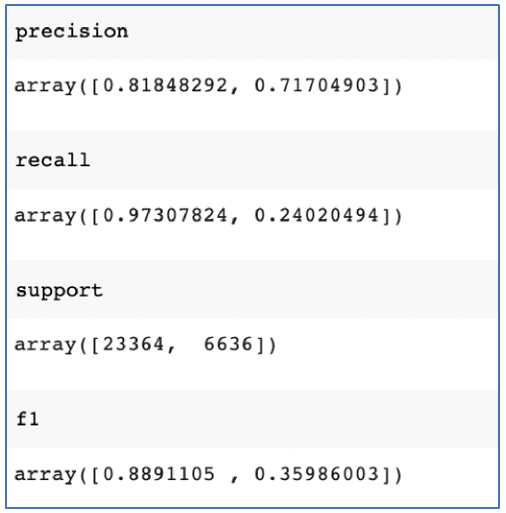
*Normalizer procedure*



*Logistic Regression*

*Standard scaler implementation Confusion Matrix*



*F1 score, precision, recall*

# PRACTICAL-6

**AIM:**

Implement K-Nearest Neighbours, Support Vector Machine (SVM) and Naïve Bayes Classifier with python’s Scikit-Learn on different datasets. Compare the classifiers based on their evaluation measures.

**CODE:**

#PRACTICAL-6

import pandas as pd

from sklearn.model\_selection import train\_test\_split

from sklearn.neighbors import KNeighborsClassifier

from sklearn.metrics import f1\_score

import matplotlib.pyplot as plt

from sklearn.svm import LinearSVC

from sklearn.naive\_bayes import GaussianNB

dft = pd.read\_excel('default of credit card clients.xls')

df=dft.rename(columns=dft.iloc[0]).drop(dft.index[0])

df.head()

df['default payment next month'].unique()

y = df['default payment next month']

y=y.astype('int')

y.head()

X=df.drop('default payment next month',axis=1,inplace=False)

X.head()

X\_train,X\_test,y\_train,y\_test = train\_test\_split(X,y)

X\_train.head()

f1=[]

n\_neighbours = [i for i in range(2,20)]

for i in range(2,20):

  neigh = KNeighborsClassifier(n\_neighbors=i,algorithm='auto')

  neigh.fit(X\_train, y\_train)

  y\_pred = neigh.predict(X\_test)

  f1.append(f1\_score(y\_test,y\_pred,average='weighted'))

  print(f"The fi score when neighbours is {i} is {f1\_score(y\_test,y\_pred,average='weighted')}")

plt.plot(n\_neighbours,f1)

svm\_clf = LinearSVC()

svm\_clf.fit(X\_train,y\_train)

y\_pred = svm\_clf.predict(X\_test)

svm\_clf\_f1 = f1\_score(y\_test,y\_pred,average='weighted') print(svm\_clf\_f1)

gnb = GaussianNB()

gnb.fit(X\_train,y\_train)

y\_pred = gnb.predict(X\_test)

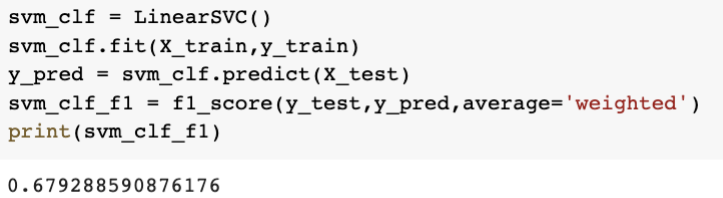
gnb\_f1 = f1\_score(y\_test,y\_pred,average='weighted')

print(gnb\_f1)

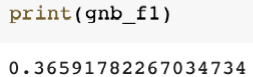
**OUTPUT:**

*K nearest neighbour output KNN graph plot*



*Linear SVC implementation*



*Gaussian Naïve bayes implementation*

# PRACTICAL-7

**AIM:**

Use K-Means Clustering and Hierarchical Clustering algorithm for following datasets

**CODE:**

#PRACTICAL-7

import pandas as pd

from sklearn.cluster import KMeans

from pandas.core.common import random\_state

import matplotlib.pyplot as plt

from sklearn.cluster import AgglomerativeClustering

import numpy as np

from scipy.cluster.hierarchy import dendrogram

df=pd.read\_csv('data.csv')

df.head()

df.drop('diagnosis', inplace=True, axis=1)

df.drop('Unnamed: 32', inplace=True, axis=1)

df.drop('id', inplace=True, axis=1)

kmeans = KMeans(n\_clusters=2, random\_state=0).fit(df)

kmeans.cluster\_centers\_

kmeans.predict([[1.93799237e+01, 2.16945802e+01, 1.28231298e+02, 1.18592977e+03,

 1.01294580e-01, 1.48612977e-01, 1.76939466e-01, 1.00698779e-01,

 1.91539695e-01, 6.06029008e-02, 7.42803817e-01, 1.22253817e+00,

 5.25058015e+00, 9.56781679e+01, 6.59868702e-03, 3.21766947e-02,

 4.24197710e-02, 1.56739847e-02, 2.03039695e-02, 3.95338931e-03,

 2.37094656e+01, 2.89126718e+01, 1.58496183e+02, 1.75302290e+03,

 1.40424733e-01, 3.57757710e-01, 4.49306107e-01, 1.92431069e-01,

 3.11881679e-01, 8.61654962e-02]])

WCSS=[]

for i in range(2,11):

 kmeans= KMeans(n\_clusters=i, random\_state=42, init='k-means++').fit(df)

 WCSS.append(kmeans.inertia\_);

WCSS

no\_of\_cluster = [i for i in range(2,11)]

plt.plot(no\_of\_cluster, WCSS)

clustering = AgglomerativeClustering(distance\_threshold=0, n\_clusters=None).fit(df)

clustering.labels\_

def plot\_dendrogram(model, \*\*kwargs):

 counts = np.zeros(model.children\_.shape[0])

 n\_samples = len(clustering.labels\_)

 for i, merge in enumerate(model.children\_):

current\_count = 0

 for child\_idx in merge:

 if child\_idx < n\_samples:

 current\_count += 1

 else:

 current\_count += counts[child\_idx - n\_samples]

 counts[i] = current\_count

 linkage\_matrix = np.column\_stack(

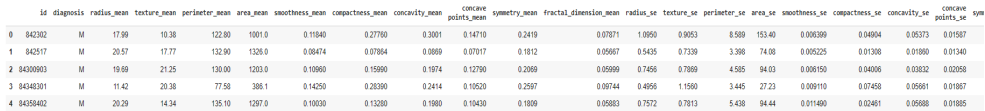
 [model.children\_, clustering.distances\_, counts]

 ).astype(float)

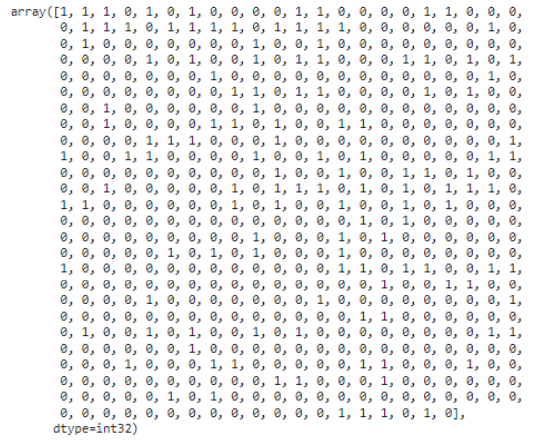
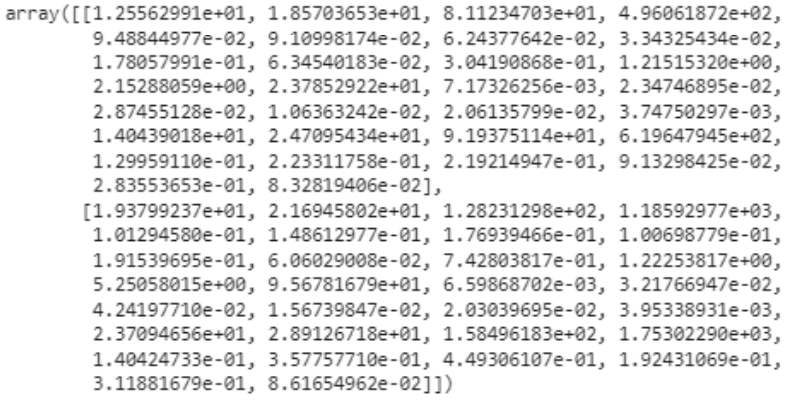
 dendrogram(linkage\_matrix, \*\*kwargs)

plot\_dendrogram(clustering, truncate\_mode="level",p=3)

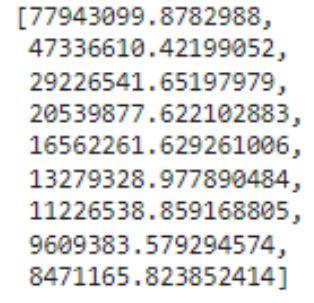
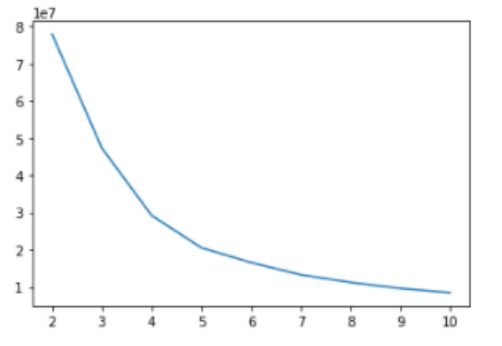
**OUTPUT:**



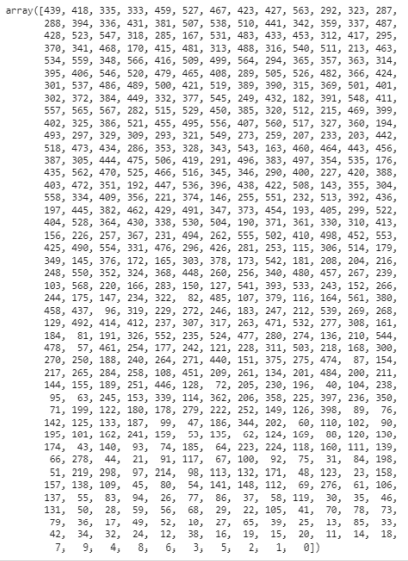
*Glimpses of Brest cancer dataset*

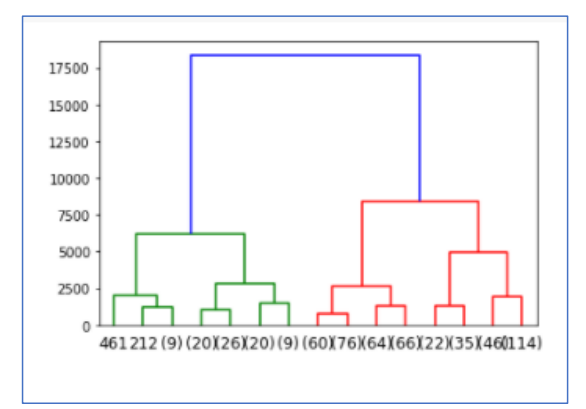
*K means labels cluster centers*

*WCSS sum of squared distance graph plot (elbow)*



*Hierachical cluster label*



*Dendrogram*

**PRACTICAL-8**

**AIM:**

Implement following using Tensorflow:

Constants, Variables, Placeholder, and operations, creating Graph and executing graph.

**CODE:**

import tensorflow as tf

print("TensorFlow version:", tf.\_version\_)

mnist = tf.keras.datasets.mnist

(x\_train, y\_train), (x\_test, y\_test) = mnist.load\_data()

x\_train, x\_test = x\_train / 255.0, x\_test / 255.0

model = tf.keras.models.Sequential([

tf.keras.layers.Flatten(input\_shape=(28, 28)),

tf.keras.layers.Dense(128, activation='relu'),

tf.keras.layers.Dropout(0.2),

tf.keras.layers.Dense(10)

])

loss\_fn = tf.keras.losses.SparseCategoricalCrossentropy(from\_logits=True)

model.compile(optimizer='adam',

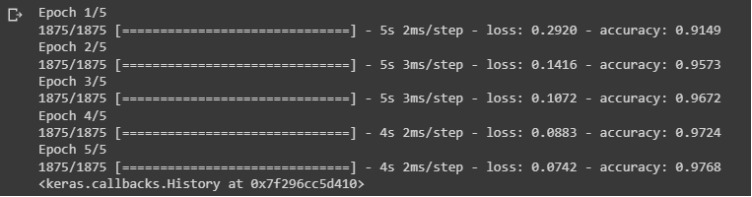
loss=loss\_fn,

metrics=['accuracy'])

model.fit(x\_train, y\_train, epochs=5)

model.evaluate(x\_test, y\_test, verbose=2)

**OUTPUT:**

******

******

**PRACTICAL-9**

**AIM:**

Implement the Multi-Layer Perceptron from scratch with at least 3 layers for a classification or a regression problem of your

choice, implement Backpropogation and observe Underfitting, Overfitting and Regularization.

**CODE:**

from google.colab import drive

drive.mount('/gdrive')

%cd /gdrive

%cd MyDrive/Colab\ Notebooks

%ls

%cd

%cd /gdrive/MyDrive/Datasets

%ls

import pandas as pd

df = pd.read\_csv('/content/wineQT.csv')

df.head(5)

df.drop('Id', axis=1, inplace=True)

df.head()

from sklearn.neural\_network import MLPClassifier

y = df['quality']

y.head()

X = df.drop('quality', axis=1, inplace=False)

X.head()

X.describe()

model = MLPClassifier(solver='sgd', hidden\_layer\_sizes=(16, 8), random\_state=1, learning\_rate\_init=0.005, learning\_rate='adaptive', verbose=True, validation\_fraction=0.1, early\_stopping=True)

model.fit(X,y)

model2 = MLPClassifier(solver='sgd', hidden\_layer\_sizes=(24, 8), random\_state=1, learning\_rate\_init=0.005, learning\_rate='adaptive', verbose=True, validation\_fraction=0.1, early\_stopping=True)

model2.fit(X,y)

import matplotlib.pyplot as plt

plt.plot(model.loss\_curve\_)

plt.plot(model2.loss\_curve\_)

model3 = MLPClassifier(solver='sgd', hidden\_layer\_sizes=(16, 12), random\_state=1, learning\_rate\_init=0.001, learning\_rate='invscaling', verbose=True, validation\_fraction=0.1, early\_stopping=True)

model3.fit(X,y)

plt.plot(model3.loss\_curve\_)

model3.score(X,y)

import pickle

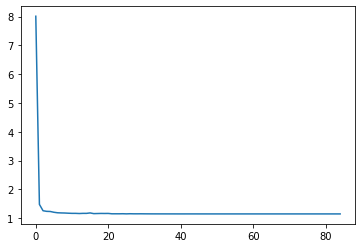
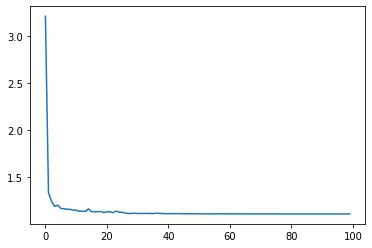
filename = 'finalized\_sklearn\_classification\_model.sav'

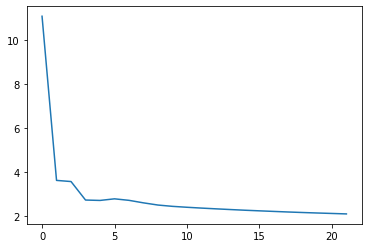
pickle.dump(model3, open(filename, 'wb'))

loaded\_model = pickle.load(open(filename, 'rb'))

loaded\_model.score(X,y)

**OUTPUT:**

******  ******

****

*The loss curves*

**PRACTICAL-10**

**AIM:**

Implement a Convolutional Neural Network (CNN) using Keras library for a face classification problem. Create dataset of faces of

your 5 friends. Also use data augmentation technique to increase dataset.

**CODE:**

import tensorflow as tf

from tensorflow.keras import datasets, layers, models

import matplotlib.pyplot as plt

(train\_images, train\_labels), (test\_images, test\_labels) = datasets.cifar10.load\_data()

# Normalize pixel values to be between 0 and 1

train\_images, test\_images = train\_images / 255.0, test\_images / 255.0

class\_names = ['airplane', 'automobile', 'bird', 'cat', 'deer',

'dog', 'frog', 'horse', 'ship', 'truck']

plt.figure(figsize=(10,10))

for i in range(25):

plt.subplot(5,5,i+1)

plt.xticks([])

plt.yticks([])

plt.grid(False)

plt.imshow(train\_images[i])

# The CIFAR labels happen to be arrays,

# which is why you need the extra index

plt.xlabel(class\_names[train\_labels[i][0]])

plt.show()

model = models.Sequential()

model.add(layers.Conv2D(32, (3, 3), activation='relu', input\_shape=(32, 32, 3)))

model.add(layers.MaxPooling2D((2, 2)))

model.add(layers.Conv2D(64, (3, 3), activation='relu'))

model.add(layers.MaxPooling2D((2, 2)))

model.add(layers.Conv2D(64, (3, 3), activation='relu'))

model.summary()

model.add(layers.Flatten())

model.add(layers.Dense(64, activation='relu'))

model.add(layers.Dense(10))

model.summary()

model.compile(optimizer='adam',

loss=tf.keras.losses.SparseCategoricalCrossentropy(from\_logits=True),

metrics=['accuracy'])

history = model.fit(train\_images, train\_labels, epochs=10,

validation\_data=(test\_images, test\_labels))

plt.plot(history.history['accuracy'], label='accuracy')

plt.plot(history.history['val\_accuracy'], label = 'val\_accuracy')

plt.xlabel('Epoch')

plt.ylabel('Accuracy')

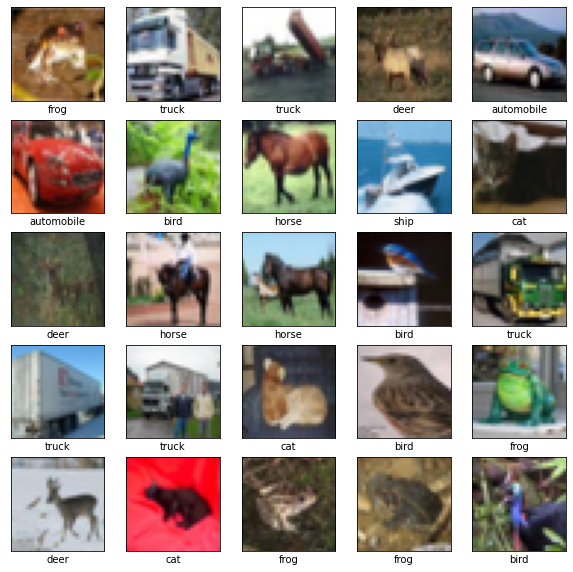
plt.ylim([0.5, 1])

plt.legend(loc='lower right')

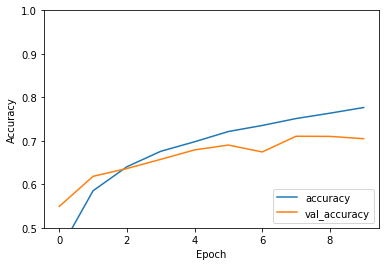
test\_loss, test\_acc = model.evaluate(test\_images, test\_labels, verbose=2)

print(test\_acc)

**OUTPUT:**

******

Glimpses of the images used

******

**PRACTICAL-11**

**AIM:**

Train a Reinforcement Learning Agent for the Multi-Armed Bandit Problem and visualize the results using matplotlib or seaborn

libraries in Python. Consider at least 15 arms (n=15).

**CODE:**

!pip install tf-agents

import abc

import numpy as np

import tensorflow as tf

from tf\_agents.agents import tf\_agent

from tf\_agents.drivers import driver

from tf\_agents.environments import py\_environment

from tf\_agents.environments import tf\_environment

from tf\_agents.environments import tf\_py\_environment

from tf\_agents.policies import tf\_policy

from tf\_agents.specs import array\_spec

from tf\_agents.specs import tensor\_spec

from tf\_agents.trajectories import time\_step as ts

from tf\_agents.trajectories import trajectory

from tf\_agents.trajectories import policy\_step

nest = tf.nest

class BanditPyEnvironment(py\_environment.PyEnvironment):

def \_init\_(self, observation\_spec, action\_spec):

self.\_observation\_spec = observation\_spec

self.\_action\_spec = action\_spec

super(BanditPyEnvironment, self).\_init\_()

# Helper functions.

def action\_spec(self):

return self.\_action\_spec

def observation\_spec(self):

return self.\_observation\_spec

def \_empty\_observation(self):

return tf.nest.map\_structure(lambda x: np.zeros(x.shape, x.dtype),

self.observation\_spec())

# These two functions below should not be overridden by subclasses.

def \_reset(self):

"""Returns a time step containing an observation."""

return ts.restart(self.\_observe(), batch\_size=self.batch\_size)

def \_step(self, action):

"""Returns a time step containing the reward for the action taken."""

reward = self.\_apply\_action(action)

return ts.termination(self.\_observe(), reward)

# These two functions below are to be implemented in subclasses.

@abc.abstractmethod

def \_observe(self):

"""Returns an observation."""

@abc.abstractmethod

def \_apply\_action(self, action):

"""Applies `action` to the Environment and returns the corresponding reward.

"""

class SimplePyEnvironment(BanditPyEnvironment):

def \_init\_(self):

action\_spec = array\_spec.BoundedArraySpec(

shape=(), dtype=np.int32, minimum=0, maximum=2, name='action')

observation\_spec = array\_spec.BoundedArraySpec(

shape=(1,), dtype=np.int32, minimum=-2, maximum=2, name='observation')

super(SimplePyEnvironment, self).\_init\_(observation\_spec, action\_spec)

def \_observe(self):

self.\_observation = np.random.randint(-2, 3, (1,), dtype='int32')

return self.\_observation

def \_apply\_action(self, action):

return action \* self.\_observation

environment = SimplePyEnvironment()

observation = environment.reset().observation

print("observation: %d" % observation)

action = 2 #@param

print("action: %d" % action)

reward = environment.step(action).reward

print("reward: %f" % reward)

tf\_environment = tf\_py\_environment.TFPyEnvironment(environment)

class SignPolicy(tf\_policy.TFPolicy):

def \_init\_(self):

observation\_spec = tensor\_spec.BoundedTensorSpec(

shape=(1,), dtype=tf.int32, minimum=-2, maximum=2)

time\_step\_spec = ts.time\_step\_spec(observation\_spec)

action\_spec = tensor\_spec.BoundedTensorSpec(

shape=(), dtype=tf.int32, minimum=0, maximum=2)

super(SignPolicy, self).\_init\_(time\_step\_spec=time\_step\_spec,

action\_spec=action\_spec)

def \_distribution(self, time\_step):

pass

def \_variables(self):

return ()

def \_action(self, time\_step, policy\_state, seed):

observation\_sign = tf.cast(tf.sign(time\_step.observation[0]), dtype=tf.int32)

action = observation\_sign + 1

return policy\_step.PolicyStep(action, policy\_state)

sign\_policy = SignPolicy()

current\_time\_step = tf\_environment.reset()

print('Observation:')

print (current\_time\_step.observation)

action = sign\_policy.action(current\_time\_step).action

print('Action:')

print (action)

reward = tf\_environment.step(action).reward

print('Reward:')

print(reward)

step = tf\_environment.reset()

action = 1

next\_step = tf\_environment.step(action)

reward = next\_step.reward

next\_observation = next\_step.observation

print("Reward: ")

print(reward)

print("Next observation:")

print(next\_observation)

class TwoWayPyEnvironment(BanditPyEnvironment):

def \_init\_(self):

action\_spec = array\_spec.BoundedArraySpec(

shape=(), dtype=np.int32, minimum=0, maximum=2, name='action')

observation\_spec = array\_spec.BoundedArraySpec(

shape=(1,), dtype=np.int32, minimum=-2, maximum=2, name='observation')

# Flipping the sign with probability 1/2.

self.\_reward\_sign = 2 \* np.random.randint(2) - 1

print("reward sign:")

print(self.\_reward\_sign)

super(TwoWayPyEnvironment, self).\_init\_(observation\_spec, action\_spec)

def \_observe(self):

self.\_observation = np.random.randint(-2, 3, (1,), dtype='int32')

return self.\_observation

def \_apply\_action(self, action):

return self.\_reward\_sign \* action \* self.\_observation[0]

two\_way\_tf\_environment = tf\_py\_environment.TFPyEnvironment(TwoWayPyEnvironment())

class TwoWaySignPolicy(tf\_policy.TFPolicy):

def \_init\_(self, situation):

observation\_spec = tensor\_spec.BoundedTensorSpec(

shape=(1,), dtype=tf.int32, minimum=-2, maximum=2)

action\_spec = tensor\_spec.BoundedTensorSpec(

shape=(), dtype=tf.int32, minimum=0, maximum=2)

time\_step\_spec = ts.time\_step\_spec(observation\_spec)

self.\_situation = situation

super(TwoWaySignPolicy, self).\_init\_(time\_step\_spec=time\_step\_spec,

action\_spec=action\_spec)

def \_distribution(self, time\_step):

pass

def \_variables(self):

return [self.\_situation]

def \_action(self, time\_step, policy\_state, seed):

sign = tf.cast(tf.sign(time\_step.observation[0, 0]), dtype=tf.int32)

def case\_unknown\_fn():

# Choose 1 so that we get information on the sign.

return tf.constant(1, shape=(1,))

# Choose 0 or 2, depending on the situation and the sign of the observation.

def case\_normal\_fn():

return tf.constant(sign + 1, shape=(1,))

def case\_flipped\_fn():

return tf.constant(1 - sign, shape=(1,))

cases = [(tf.equal(self.\_situation, 0), case\_unknown\_fn),

(tf.equal(self.\_situation, 1), case\_normal\_fn),

(tf.equal(self.\_situation, 2), case\_flipped\_fn)]

action = tf.case(cases, exclusive=True)

return policy\_step.PolicyStep(action, policy\_state)

class SignAgent(tf\_agent.TFAgent):

def \_init\_(self):

self.\_situation = tf.Variable(0, dtype=tf.int32)

policy = TwoWaySignPolicy(self.\_situation)

time\_step\_spec = policy.time\_step\_spec

action\_spec = policy.action\_spec

super(SignAgent, self).\_init\_(time\_step\_spec=time\_step\_spec,

action\_spec=action\_spec,

policy=policy,

collect\_policy=policy,

train\_sequence\_length=None)

def \_initialize(self):

return tf.compat.v1.variables\_initializer(self.variables)

def \_train(self, experience, weights=None):

observation = experience.observation

action = experience.action

reward = experience.reward

# We only need to change the value of the situation variable if it is

# unknown (0) right now, and we can infer the situation only if the

# observation is not 0.

needs\_action = tf.logical\_and(tf.equal(self.\_situation, 0),

tf.not\_equal(reward, 0))

def new\_situation\_fn():

"""This returns either 1 or 2, depending on the signs."""

return (3 - tf.sign(tf.cast(observation[0, 0, 0], dtype=tf.int32) \*

tf.cast(action[0, 0], dtype=tf.int32) \*

tf.cast(reward[0, 0], dtype=tf.int32))) / 2

new\_situation = tf.cond(needs\_action,

new\_situation\_fn,

lambda: self.\_situation)

new\_situation = tf.cast(new\_situation, tf.int32)

tf.compat.v1.assign(self.\_situation, new\_situation)

return tf\_agent.LossInfo((), ())

sign\_agent = SignAgent()

# We need to add another dimension here because the agent expects the

# trajectory of shape [batch\_size, time, ...], but in this tutorial we assume

# that both batch size and time are 1. Hence all the expand\_dims.

def trajectory\_for\_bandit(initial\_step, action\_step, final\_step):

return trajectory.Trajectory(observation=tf.expand\_dims(initial\_step.observation, 0),

action=tf.expand\_dims(action\_step.action, 0),

policy\_info=action\_step.info,

reward=tf.expand\_dims(final\_step.reward, 0),

discount=tf.expand\_dims(final\_step.discount, 0),

step\_type=tf.expand\_dims(initial\_step.step\_type, 0),

next\_step\_type=tf.expand\_dims(final\_step.step\_type, 0))

step = two\_way\_tf\_environment.reset()

for \_ in range(10):

action\_step = sign\_agent.collect\_policy.action(step)

next\_step = two\_way\_tf\_environment.step(action\_step.action)

experience = trajectory\_for\_bandit(step, action\_step, next\_step)

print(experience)

sign\_agent.train(experience)

step = next\_step

# Imports for example.

from tf\_agents.bandits.agents import lin\_ucb\_agent

from tf\_agents.bandits.environments import stationary\_stochastic\_py\_environment as sspe

from tf\_agents.bandits.metrics import tf\_metrics

from tf\_agents.drivers import dynamic\_step\_driver

from tf\_agents.replay\_buffers import tf\_uniform\_replay\_buffer

import matplotlib.pyplot as plt

batch\_size = 2 # @param

arm0\_param = [-3, 0, 1, -2] # @param

arm1\_param = [1, -2, 3, 0] # @param

arm2\_param = [0, 0, 1, 1] # @param

def context\_sampling\_fn(batch\_size):

"""Contexts from [-10, 10]^4."""

def \_context\_sampling\_fn():

return np.random.randint(-10, 10, [batch\_size, 4]).astype(np.float32)

return \_context\_sampling\_fn

class LinearNormalReward(object):

"""A class that acts as linear reward function when called."""

def \_init\_(self, theta, sigma):

self.theta = theta

self.sigma = sigma

def \_call\_(self, x):

mu = np.dot(x, self.theta)

return np.random.normal(mu, self.sigma)

arm0\_reward\_fn = LinearNormalReward(arm0\_param, 1)

arm1\_reward\_fn = LinearNormalReward(arm1\_param, 1)

arm2\_reward\_fn = LinearNormalReward(arm2\_param, 1)

environment = tf\_py\_environment.TFPyEnvironment(

sspe.StationaryStochasticPyEnvironment(

context\_sampling\_fn(batch\_size),

[arm0\_reward\_fn, arm1\_reward\_fn, arm2\_reward\_fn],

batch\_size=batch\_size))

observation\_spec = tensor\_spec.TensorSpec([4], tf.float32)

time\_step\_spec = ts.time\_step\_spec(observation\_spec)

action\_spec = tensor\_spec.BoundedTensorSpec(

dtype=tf.int32, shape=(), minimum=0, maximum=2)

agent = lin\_ucb\_agent.LinearUCBAgent(time\_step\_spec=time\_step\_spec,

action\_spec=action\_spec)

def compute\_optimal\_reward(observation):

expected\_reward\_for\_arms = [

tf.linalg.matvec(observation, tf.cast(arm0\_param, dtype=tf.float32)),

tf.linalg.matvec(observation, tf.cast(arm1\_param, dtype=tf.float32)),

tf.linalg.matvec(observation, tf.cast(arm2\_param, dtype=tf.float32))]

optimal\_action\_reward = tf.reduce\_max(expected\_reward\_for\_arms, axis=0)

return optimal\_action\_reward

regret\_metric = tf\_metrics.RegretMetric(compute\_optimal\_reward)

num\_iterations = 90 # @param

steps\_per\_loop = 1 # @param

replay\_buffer = tf\_uniform\_replay\_buffer.TFUniformReplayBuffer(

data\_spec=agent.policy.trajectory\_spec,

batch\_size=batch\_size,

max\_length=steps\_per\_loop)

observers = [replay\_buffer.add\_batch, regret\_metric]

driver = dynamic\_step\_driver.DynamicStepDriver(

env=environment,

policy=agent.collect\_policy,

num\_steps=steps\_per\_loop \* batch\_size,

observers=observers)

regret\_values = []

for \_ in range(num\_iterations):

driver.run()

loss\_info = agent.train(replay\_buffer.gather\_all())

replay\_buffer.clear()

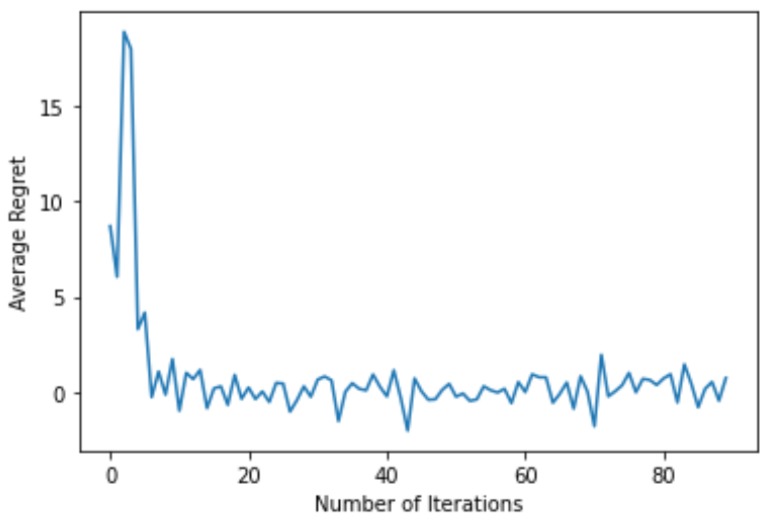
regret\_values.append(regret\_metric.result())

plt.plot(regret\_values)

plt.ylabel('Average Regret')

plt.xlabel('Number of Iterations')

**OUTPUT:**

******

**PRACTICAL-12**

**AIM:**

Implement a Deep Learning Algorithm/Method to Predict stock prices based on past price variation.

**CODE:**

pip install yahoo-fin

import tensorflow as tf

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import LSTM, Dense, Dropout, Bidirectional

from tensorflow.keras.callbacks import ModelCheckpoint, TensorBoard

from sklearn import preprocessing

from sklearn.model\_selection import train\_test\_split

from yahoo\_fin import stock\_info as si

from collections import deque

import os

import numpy as np

import pandas as pd

import random

# set seed, so we can get the same results after rerunning several times

np.random.seed(314)

tf.random.set\_seed(314)

random.seed(314)

import os

import time

from tensorflow.keras.layers import LSTM

# Window size or the sequence length

N\_STEPS = 50

# Lookup step, 1 is the next day

LOOKUP\_STEP = 15

# whether to scale feature columns & output price as well

SCALE = True

scale\_str = f"sc-{int(SCALE)}"

# whether to shuffle the dataset

SHUFFLE = True

shuffle\_str = f"sh-{int(SHUFFLE)}"

# whether to split the training/testing set by date

SPLIT\_BY\_DATE = False

split\_by\_date\_str = f"sbd-{int(SPLIT\_BY\_DATE)}"

# test ratio size, 0.2 is 20%

TEST\_SIZE = 0.2

# features to use

FEATURE\_COLUMNS = ["adjclose", "volume", "open", "high", "low"]

# date now

date\_now = time.strftime("%Y-%m-%d")

### model parameters

N\_LAYERS = 2

# LSTM cell

CELL = LSTM

# 256 LSTM neurons

UNITS = 256

# 40% dropout

DROPOUT = 0.4

# whether to use bidirectional RNNs

BIDIRECTIONAL = False

### training parameters

# mean absolute error loss

# LOSS = "mae"

# huber loss

LOSS = "huber\_loss"

OPTIMIZER = "adam"

BATCH\_SIZE = 64

EPOCHS = 500

# Amazon stock market

ticker = "AMZN"

ticker\_data\_filename = os.path.join("data", f"{ticker}\_{date\_now}.csv")

# model name to save, making it as unique as possible based on parameters

model\_name = f"{date\_now}\_{ticker}-{shuffle\_str}-{scale\_str}-{split\_by\_date\_str}-\

{LOSS}-{OPTIMIZER}-{CELL.\_\_name\_\_}-seq-{N\_STEPS}-step-{LOOKUP\_STEP}-layers-{N\_LAYERS}-units-{UNITS}"

if BIDIRECTIONAL:

model\_name += "-b"

def shuffle\_in\_unison(a, b):

# shuffle two arrays in the same way

state = np.random.get\_state()

np.random.shuffle(a)

np.random.set\_state(state)

np.random.shuffle(b)

def load\_data(ticker, n\_steps=50, scale=True, shuffle=True, lookup\_step=1, split\_by\_date=True,

test\_size=0.2, feature\_columns=['adjclose', 'volume', 'open', 'high', 'low']):

# see if ticker is already a loaded stock from yahoo finance

if isinstance(ticker, str):

# load it from yahoo\_fin library

df = si.get\_data(ticker)

elif isinstance(ticker, pd.DataFrame):

# already loaded, use it directly

df = ticker

else:

raise TypeError("ticker can be either a str or a `pd.DataFrame` instances")

# this will contain all the elements we want to return from this function

result = {}

# we will also return the original dataframe itself

result['df'] = df.copy()

# make sure that the passed feature\_columns exist in the dataframe

for col in feature\_columns:

assert col in df.columns, f"'{col}' does not exist in the dataframe."

# add date as a column

if "date" not in df.columns:

df["date"] = df.index

if scale:

column\_scaler = {}

# scale the data (prices) from 0 to 1

for column in feature\_columns:

scaler = preprocessing.MinMaxScaler()

df[column] = scaler.fit\_transform(np.expand\_dims(df[column].values, axis=1))

column\_scaler[column] = scaler

# add the MinMaxScaler instances to the result returned

result["column\_scaler"] = column\_scaler

# add the target column (label) by shifting by `lookup\_step`

df['future'] = df['adjclose'].shift(-lookup\_step)

# last `lookup\_step` columns contains NaN in future column

# get them before droping NaNs

last\_sequence = np.array(df[feature\_columns].tail(lookup\_step))

# drop NaNs

df.dropna(inplace=True)

sequence\_data = []

sequences = deque(maxlen=n\_steps)

for entry, target in zip(df[feature\_columns + ["date"]].values, df['future'].values):

sequences.append(entry)

if len(sequences) == n\_steps:

sequence\_data.append([np.array(sequences), target])

# get the last sequence by appending the last `n\_step` sequence with `lookup\_step` sequence

# for instance, if n\_steps=50 and lookup\_step=10, last\_sequence should be of 60 (that is 50+10) length

# this last\_sequence will be used to predict future stock prices that are not available in the dataset

last\_sequence = list([s[:len(feature\_columns)] for s in sequences]) + list(last\_sequence)

last\_sequence = np.array(last\_sequence).astype(np.float32)

# add to result

result['last\_sequence'] = last\_sequence

# construct the X's and y's

X, y = [], []

for seq, target in sequence\_data:

X.append(seq)

y.append(target)

# convert to numpy arrays

X = np.array(X)

y = np.array(y)

if split\_by\_date:

# split the dataset into training & testing sets by date (not randomly splitting)

train\_samples = int((1 - test\_size) \* len(X))

result["X\_train"] = X[:train\_samples]

result["y\_train"] = y[:train\_samples]

result["X\_test"] = X[train\_samples:]

result["y\_test"] = y[train\_samples:]

if shuffle:

# shuffle the datasets for training (if shuffle parameter is set)

shuffle\_in\_unison(result["X\_train"], result["y\_train"])

shuffle\_in\_unison(result["X\_test"], result["y\_test"])

else:

# split the dataset randomly

result["X\_train"], result["X\_test"], result["y\_train"], result["y\_test"] = train\_test\_split(X, y, test\_size=test\_size, shuffle=shuffle)

# get the list of test set dates

dates = result["X\_test"][:, -1, -1]

# retrieve test features from the original dataframe

result["test\_df"] = result["df"].loc[dates]

# remove duplicated dates in the testing dataframe

result["test\_df"] = result["test\_df"][~result["test\_df"].index.duplicated(keep='first')]

# remove dates from the training/testing sets & convert to float32

result["X\_train"] = result["X\_train"][:, :, :len(feature\_columns)].astype(np.float32)

result["X\_test"] = result["X\_test"][:, :, :len(feature\_columns)].astype(np.float32)

return result

def create\_model(sequence\_length, n\_features, units=256, cell=LSTM, n\_layers=2, dropout=0.3,

loss="mean\_absolute\_error", optimizer="rmsprop", bidirectional=False):

model = Sequential()

for i in range(n\_layers):

if i == 0:

# first layer

if bidirectional:

model.add(Bidirectional(cell(units, return\_sequences=True), batch\_input\_shape=(None, sequence\_length, n\_features)))

else:

model.add(cell(units, return\_sequences=True, batch\_input\_shape=(None, sequence\_length, n\_features)))

elif i == n\_layers - 1:

# last layer

if bidirectional:

model.add(Bidirectional(cell(units, return\_sequences=False)))

else:

model.add(cell(units, return\_sequences=False))

else:

# hidden layers

if bidirectional:

model.add(Bidirectional(cell(units, return\_sequences=True)))

else:

model.add(cell(units, return\_sequences=True))

# add dropout after each layer

model.add(Dropout(dropout))

model.add(Dense(1, activation="linear"))

model.compile(loss=loss, metrics=["mean\_absolute\_error"], optimizer=optimizer)

return model

# create these folders if they does not exist

if not os.path.isdir("results"):

os.mkdir("results")

if not os.path.isdir("logs"):

os.mkdir("logs")

if not os.path.isdir("data"):

os.mkdir("data")

# load the data

data = load\_data(ticker, N\_STEPS, scale=SCALE, split\_by\_date=SPLIT\_BY\_DATE,

shuffle=SHUFFLE, lookup\_step=LOOKUP\_STEP, test\_size=TEST\_SIZE,

feature\_columns=FEATURE\_COLUMNS)

# save the dataframe

data["df"].to\_csv(ticker\_data\_filename)

# construct the model

model = create\_model(N\_STEPS, len(FEATURE\_COLUMNS), loss=LOSS, units=UNITS, cell=CELL, n\_layers=N\_LAYERS,

dropout=DROPOUT, optimizer=OPTIMIZER, bidirectional=BIDIRECTIONAL)

# some tensorflow callbacks

checkpointer = ModelCheckpoint(os.path.join("results", model\_name + ".h5"), save\_weights\_only=True, save\_best\_only=True, verbose=1)

tensorboard = TensorBoard(log\_dir=os.path.join("logs", model\_name))

# train the model and save the weights whenever we see

# a new optimal model using ModelCheckpoint

history = model.fit(data["X\_train"], data["y\_train"],

batch\_size=BATCH\_SIZE,

epochs=EPOCHS,

validation\_data=(data["X\_test"], data["y\_test"]),

callbacks=[checkpointer, tensorboard],

verbose=1)

import matplotlib.pyplot as plt

def plot\_graph(test\_df):

"""

This function plots true close price along with predicted close price

with blue and red colors respectively

"""

plt.plot(test\_df[f'true\_adjclose\_{LOOKUP\_STEP}'], c='b')

plt.plot(test\_df[f'adjclose\_{LOOKUP\_STEP}'], c='r')

plt.xlabel("Days")

plt.ylabel("Price")

plt.legend(["Actual Price", "Predicted Price"])

plt.show()

def get\_final\_df(model, data):

"""

This function takes the `model` and `data` dict to

construct a final dataframe that includes the features along

with true and predicted prices of the testing dataset

"""

# if predicted future price is higher than the current,

# then calculate the true future price minus the current price, to get the buy profit

buy\_profit = lambda current, pred\_future, true\_future: true\_future - current if pred\_future > current else 0

# if the predicted future price is lower than the current price,

# then subtract the true future price from the current price

sell\_profit = lambda current, pred\_future, true\_future: current - true\_future if pred\_future < current else 0

X\_test = data["X\_test"]

y\_test = data["y\_test"]

# perform prediction and get prices

y\_pred = model.predict(X\_test)

if SCALE:

y\_test = np.squeeze(data["column\_scaler"]["adjclose"].inverse\_transform(np.expand\_dims(y\_test, axis=0)))

y\_pred = np.squeeze(data["column\_scaler"]["adjclose"].inverse\_transform(y\_pred))

test\_df = data["test\_df"]

# add predicted future prices to the dataframe

test\_df[f"adjclose\_{LOOKUP\_STEP}"] = y\_pred

# add true future prices to the dataframe

test\_df[f"true\_adjclose\_{LOOKUP\_STEP}"] = y\_test

# sort the dataframe by date

test\_df.sort\_index(inplace=True)

final\_df = test\_df

# add the buy profit column

final\_df["buy\_profit"] = list(map(buy\_profit,

final\_df["adjclose"],

final\_df[f"adjclose\_{LOOKUP\_STEP}"],

final\_df[f"true\_adjclose\_{LOOKUP\_STEP}"])

# since we don't have profit for last sequence, add 0's

)

# add the sell profit column

final\_df["sell\_profit"] = list(map(sell\_profit,

final\_df["adjclose"],

final\_df[f"adjclose\_{LOOKUP\_STEP}"],

final\_df[f"true\_adjclose\_{LOOKUP\_STEP}"])

# since we don't have profit for last sequence, add 0's

)

return final\_df

def predict(model, data):

# retrieve the last sequence from data

last\_sequence = data["last\_sequence"][-N\_STEPS:]

# expand dimension

last\_sequence = np.expand\_dims(last\_sequence, axis=0)

# get the prediction (scaled from 0 to 1)

prediction = model.predict(last\_sequence)

# get the price (by inverting the scaling)

if SCALE:

predicted\_price = data["column\_scaler"]["adjclose"].inverse\_transform(prediction)[0][0]

else:

predicted\_price = prediction[0][0]

return predicted\_price

# load optimal model weights from results folder

model\_path = os.path.join("results", model\_name) + ".h5"

model.load\_weights(model\_path)

# evaluate the model

loss, mae = model.evaluate(data["X\_test"], data["y\_test"], verbose=0)

# calculate the mean absolute error (inverse scaling)

if SCALE:

mean\_absolute\_error = data["column\_scaler"]["adjclose"].inverse\_transform([[mae]])[0][0]

else:

mean\_absolute\_error = mae

# get the final dataframe for the testing set

final\_df = get\_final\_df(model, data)

# predict the future price

future\_price = predict(model, data)

# we calculate the accuracy by counting the number of positive profits

accuracy\_score = (len(final\_df[final\_df['sell\_profit'] > 0]) + len(final\_df[final\_df['buy\_profit'] > 0])) / len(final\_df)

# calculating total buy & sell profit

total\_buy\_profit = final\_df["buy\_profit"].sum()

total\_sell\_profit = final\_df["sell\_profit"].sum()

# total profit by adding sell & buy together

total\_profit = total\_buy\_profit + total\_sell\_profit

# dividing total profit by number of testing samples (number of trades)

profit\_per\_trade = total\_profit / len(final\_df)

# printing metrics

print(f"Future price after {LOOKUP\_STEP} days is {future\_price:.2f}$")

print(f"{LOSS} loss:", loss)

print("Mean Absolute Error:", mean\_absolute\_error)

print("Accuracy score:", accuracy\_score)

print("Total buy profit:", total\_buy\_profit)

print("Total sell profit:", total\_sell\_profit)

print("Total profit:", total\_profit)

print("Profit per trade:", profit\_per\_trade)

# plot true/pred prices graph

plot\_graph(final\_df)

final\_df.head(20)

final\_df.tail(20)

# save the final dataframe to csv-results folder

csv\_results\_folder = "csv-results"

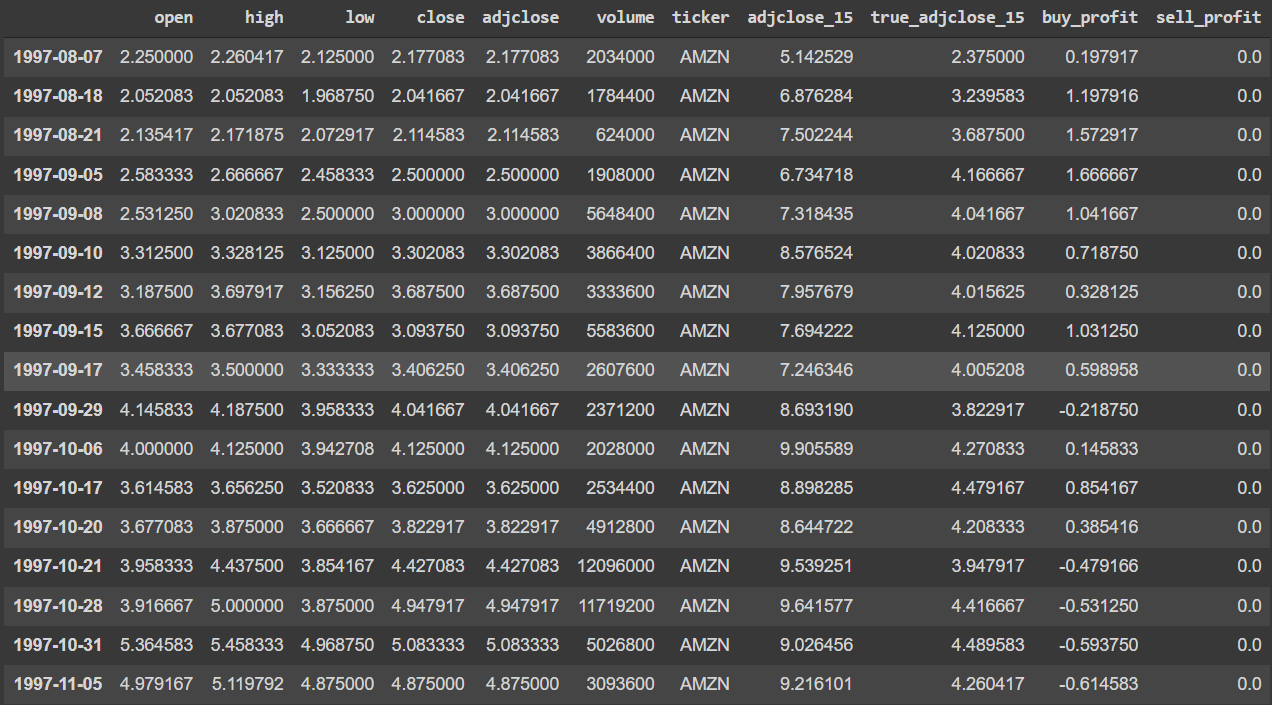
if not os.path.isdir(csv\_results\_folder):

os.mkdir(csv\_results\_folder)

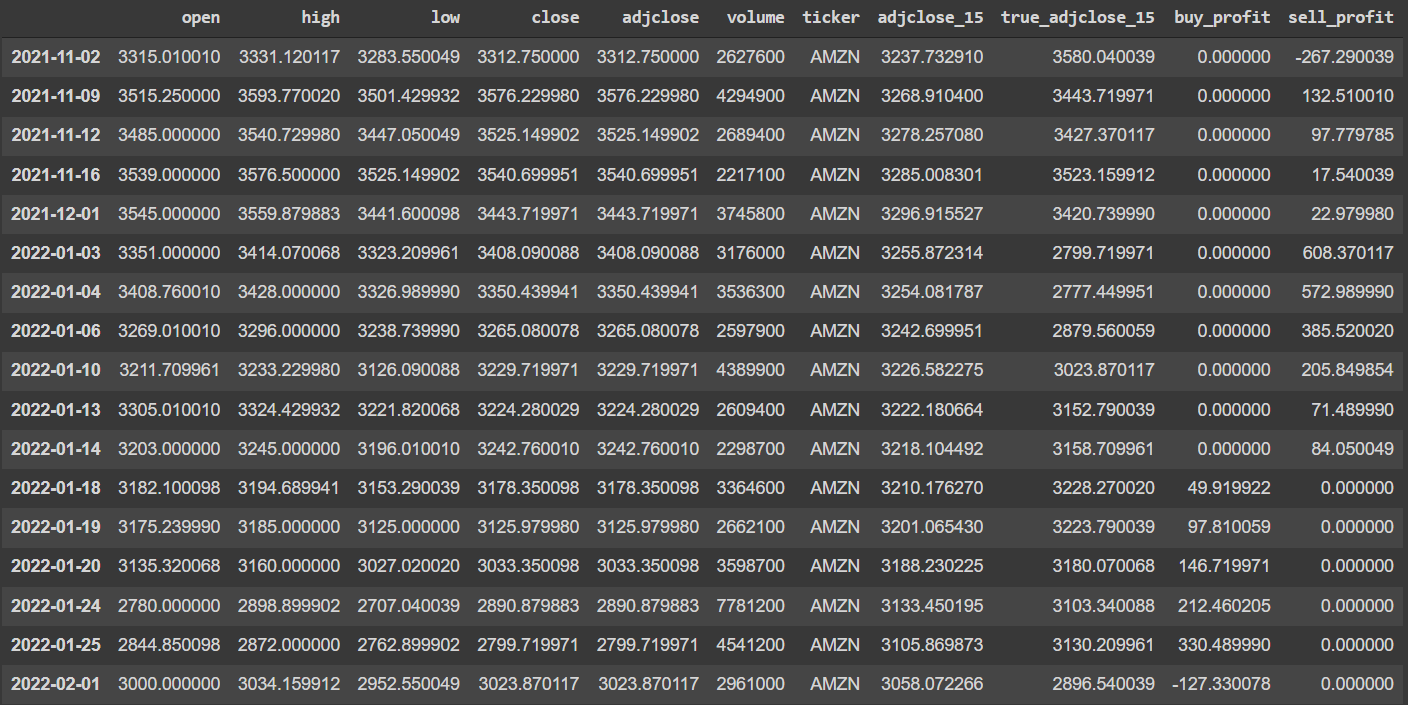
csv\_filename = os.path.join(csv\_results\_folder, model\_name + ".csv")

final\_df.to\_csv(csv\_filename)

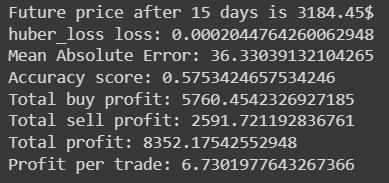
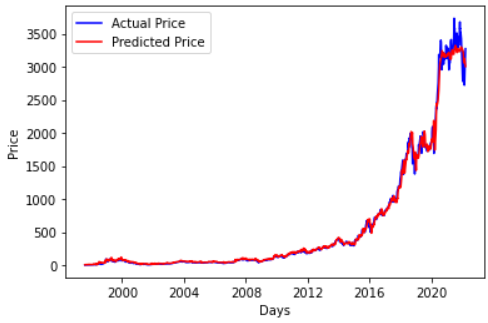
**OUTPUT:**



Output of Head()



Output of Tail()

Metrics Actual vs predicted price