BFS

```
import queue as Q
from RMP import dict_gn

start='Arad'
goal='Bucharest'
result=''

def BFS(city,cityq,visitedq):
	global result
	if city == start:
		result=result+' '+city
	for eachcity in dict_gn[city].keys():
		if eachcity == goal:
			result=result+' '+eachcity
			return
		if eachcity not in cityq.queue and eachcity not in
visitedq.queue:
			cityq.put(eachcity)
			result=result+' '+eachcity
	visitedq.put(city)
	BFS(cityq.get(),cityq,visitedq)

def main():
	cityq=Q.Queue()
	visitedq=Q.Queue()
	BFS(start,cityq,visitedq)
	print('BFS Traversal from',start,'to',goal,'is:')
	print(result)

main()
```

IDFS

```
import queue as Q
from RMP import dict_gn

start='Arad'
goal='Bucharest'
result=''

def DLS(city,visitedstack,startlimit,endlimit):
	global result
	found=0
	result = result + city + ' '
	visitedstack.append(city)
	if city == goal:
		return 1
	if startlimit==endlimit:
		return 0
	for eachcity in dict_gn[city].keys():
```

```
            if eachcity not in visitedstack:

        found=DLS(eachcity,visitedstack,startlimit+1,endlimit)
                    if found:
                         return found

def IDDFS(city,visitedstack,endlist):
     global result
     for i in range(0,endlimit):
          print('Searching at limit: ',i)
          found=DLS(city,visitedstack,0,i)
          if found:
               print('Found')
               break
          else:
               print('Not Found')
               print(result)
               print('-----')
               result=''
               visitedstack=[]

def main():
     visitedstack=[]
     IDDFS(start,visitedstack,9)
     print('IDDFS Traversal from',start,'to',goal,'is:')
     print(result)

main()
```

A* Search

```
from RMP import dict_gn
from RMP import dict_hm
import queue as Q

start = 'Bucharest'
goal = 'Bucharest'
result = ' '

def get_fn(citystr):
     cities=citystr.split(',')
     hm=gn=0
     for ctr in range(0,len(cities)-1):
          gn=gn+dict_gn[cities[ctr]][cities[ctr+1]]
     hm=dict_hm[cities[len(cities)-1]]
     return (hm+gn)

def expand(cityq):
```

```
     global result
     tot,citystr,thiscity=cityq.get()
     if thiscity==goal:
          result=citystr+'::'+str(tot)
          return
     for cty in dict_gn[thiscity]:

     cityq.put((get_fn(citystr+","+cty),citystr+","+cty,cty))
     expand(cityq)

def main():
     cityq=Q.PriorityQueue()
     thiscity=start
     cityq.put((get_fn(start),start,thiscity))
     expand(cityq)
     print("The A* path with the total is: ")
     print(result)

main()
```

RBFS

```
from RMP import dict_gn
from RMP import dict_hm
import queue as Q

start = 'Arad'
goal = 'Sibiu'
result = ' '

def get_fn(citystr):
     cities=citystr.split(',')
     hm=gn=0
     for ctr in range(0,len(cities)-1):
          gn=gn+dict_gn[cities[ctr]][cities[ctr+1]]
     hm=dict_hm[cities[len(cities)-1]]
     return (hm+gn)

def printout(cityq):
     for i in range(0,cityq.qsize()):
          print(cityq.queue[i])

def expand(cityq):
     global result
     tot,citystr,thiscity = cityq.get()
     print('---',tot,citystr,thiscity,'---')
     nexttot = 999
     if not cityq.empty():
```

```
          nexttot,nextcitystr,nextthiscity=cityq.queue[0]
          if thiscity==goal and tot<nexttot:
               result = citystr + '::' + str(tot)
               return
          print('Expanded city------------------',thiscity)
          print('Second best f(n)-----------------',nexttot)
          tempq=Q.PriorityQueue()
          for cty in dict_gn[thiscity]:
               tempq.put((get_fn(citystr + "," + cty), citystr +
"," + cty,cty))
          for ctr in range(1,3):
               ctrtot,ctrcitystr,ctrthiscity = tempq.get()
               if ctrtot < nexttot:
                    cityq.put((ctrtot,ctrcitystr,ctrthiscity))
               else:
                    cityq.put((ctrtot,citystr,thiscity))
                    break
          printout(cityq)
          expand(cityq)

def main():
     cityq=Q.PriorityQueue()
     thiscity=start
     cityq.put((999,'NA','NA'))
     cityq.put((get_fn(start),start,thiscity))
     expand(cityq)
     print('This RBFS path with the total is: ')
     print(result)

main()
```

DECISION TREE

```
import numpy as np
import pandas as pd
from sklearn import tree
from sklearn.preprocessing import LabelEncoder

# Loading the data
PlayTennis = pd.read_csv('PlayTennis.csv')

Le = LabelEncoder()

PlayTennis['outlook'] = Le.fit_transform(PlayTennis['outlook'])
PlayTennis['temp'] = Le.fit_transform(PlayTennis['temp'])
PlayTennis['humidity'] =
Le.fit_transform(PlayTennis['humidity'])
PlayTennis['windy'] = Le.fit_transform(PlayTennis['windy'])
```

```
PlayTennis['play'] = Le.fit_transform(PlayTennis['play'])

print(PlayTennis)
y = PlayTennis['play']
X = PlayTennis.drop(['play'], axis=1)

clf = tree.DecisionTreeClassifier(criterion='entropy')
clf = clf.fit(X, y)

tree.plot_tree(clf)

o = int(input("What is the outlook?: "))
t = int(input("What is the temperatur?: "))
h = int(input("What is the humidity?: "))
w = int(input("How windy it is?: "))

y_predict = clf.predict([[o,t,h,w]])
v = 'PLAY' if y_predict[0] == 1 else "not play"
print("As per situation one should",v)
```

FEED FORWARD BACKPROPAGATION NEURAL NETWORK

```
import numpy as np

X = np.array(([2, 9], [1, 5], [3,6]), dtype=float)
y= np.array(([92], [86], [89]), dtype=float)
X = X/np.amax(X,axis=0)
y = y/np.amax(y)

def sigmoid (x):
     return ((1/1 + np.exp(-x)))
def derivatives_sigmoid(x):
     return x * (1 - x)

epoch=7000
#epoch = 2
lr=0.1
inputlayer_neurons = 2
hiddenlayer_neurons = 3
output_neurons = 1

#wh = 2 x 3
wh=np.random.uniform(size=(inputlayer_neurons,
hiddenlayer_neurons))
#bh = 1 x 3
bh=np.random.uniform(size=(1,hiddenlayer_neurons))
#wout = 3 x 1
```

```
wout=np.random.uniform(size=(hiddenlayer_neurons,
output_neurons))
#bout = 1 x 1
bout=np.random.uniform(size=(1,output_neurons))


for i in range(epoch) :
     #between input layer and hidden layer
     #dot product of X and wh
     hinp1=np.dot(X,wh)
     hinp=hinp1 + bh
     hlayer_act = sigmoid(hinp)
     #between hidden layer and output
     #dot product of outputs of hidden layer and wout
     outinp1=np.dot(hlayer_act,wout)
     outinp= outinp1+ bout
     output = sigmoid(outinp)

     #Backpropogation of error
     EO = y-output
     if i == 0:
          print('Error',EO)
     outgrad = derivatives_sigmoid(output)
     d_output = EO* outgrad
     EH = d_output.dot(wout.T)
     hiddengrad = derivatives_sigmoid(hlayer_act)

     #weight adjustments
     d_hiddenlayer = EH * hiddengrad
     wout += hlayer_act.T.dot(d_output) *lr

     bout += np.sum(d_output, axis=0,keepdims=True) *lr
     wh += X.T.dot(d_hiddenlayer)*lr

print('Error' ,EO)
print("Input: \n" + str(X))
print("Actual Output: \n" + str(y))
print("Prdeicted Output: \n" ,output)
```

SVM

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.svm import SVC, LinearSVC
from sklearn.metrics import confusion_matrix
from sklearn.metrics import*# classification_report
```

```
from sklearn.metrics import accuracy_score
#import seaborn as sns

df = pd.read_csv('diabetes.csv')
#df.head()

x = df.drop('Outcome', axis=1)
y = df['Outcome']

x_train = x.iloc[:600]
x_test = x.iloc[600:]
y_train = y[:600]
y_test = y[600:]

classifier = SVC(kernel="linear")
classifier.fit(x_train, y_train)

y_pred = classifier.predict(x_test)

print('Confusion Martix:')
print(confusion_matrix(y_test, y_pred))

print('Accuracy Score')
print(accuracy_score(y_test, y_pred))

print('Classification Report:')
print(classification_report(y_test,y_pred))
```

ADABOOST

```
import pandas as pd
import warnings
from sklearn.model_selection import train_test_split
from sklearn.ensemble import AdaBoostClassifier
from sklearn.metrics import confusion_matrix, accuracy_score,
classification_report

warnings.filterwarnings("ignore")

data = pd.read_csv("apples_and_oranges.csv")
print(data)

#test_size = 0.2 =>20% test, 80% training

training_set, test_set = train_test_split(data, test_size =0.2,
random_state = 1)
X_train = training_set.iloc[:,0:2].values
Y_train = training_set.iloc[:,2].values
```

```
X_test = test_set.iloc[:,0:2].values
Y_test = test_set.iloc[:,2].values

#base_estimator: it  is a week learner used to train the model.
It uses DecisionTreeClassifier as default weak learner for
training purpose. You can also specify different machine
learning algorithms.
#N_estimators: Number of weak learners to train iteratively.
#learning_rate: It contributes to the weights of weak learners.
It uses 1 as a default value.
Adaboost = AdaBoostClassifier(n_estimators=100, learning_rate =
1, random_state = 1)
adaboost.fit(X_train,Y_train)
Y_pred =adaboost.predict(X_test)
test_set["Predictions"] = Y_pred
print(Y_pred)
```

NAÏVE BAYES’

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.naive_bayes import MultinomialNB, GaussianNB
from sklearn.metrics import confusion_matrix
from sklearn.metrics import classification_report
from sklearn.metrics import accuracy_score
from sklearn.preprocessing import LabelEncoder

df = pd.read_csv('disease.csv')

le = LabelEncoder()
df['Sore Throat'] = le.fit_transform(df['Sore Throat'])
df['fever'] = le.fit_transform(df['fever'])
df['Swollen Glands'] = le.fit_transform(df['Swollen Glands'])
df['congestion'] = le.fit_transform(df['congestion'])
df['headache'] = le.fit_transform(df['headache'])
df['diagnosis'] = le.fit_transform(df['diagnosis'])

x = df.drop('diagnosis',axis=1)
y = df['diagnosis']

x_train, x_test, y_train, y_test =
train_test_split(x,y,test_size=0.2)

classifier = MultinomialNB()
classifier.fit(x_train,y_train)
```

```
y_pred = classifier.predict(x_test)

print('Confusion Matrix:')
print(confusion_matrix(y_test, y_pred))

print('classification Report:')
print(classification_report(y_test, y_pred))
```

KNN

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import confusion_matrix
from sklearn.metrics import classification_report

plt.style.use('ggplot')

df = pd.read_csv('diabetes.csv')
#df.head()
#df.shape
#df.dtypes
X = df.drop('Outcome',axis=1).values
y = df['Outcome'].values

x_train, x_test, y_train, y_test = train_test_split(X, y,
test_size=0.4, random_state=42, shuffle=True)

neighbors = np.arange(1, 9)
train_accuracy = np.empty(len(neighbors))
test_accuray = np.empty(len(neighbors))

for i, k in enumerate(neighbors):
    knn = KNeighborsClassifier(n_neighbors=k) #Setup a KNN
classifier with k neighbours
    knn.fit(x_train,y_train)
    train_accuracy[i] = knn.score(x_train,y_train)
    test_accuray[i] = knn.score(x_test,y_test)

knn = KNeighborsClassifier(n_neighbors=7)
knn.fit(x_train, y_train)
s = knn.score(x_test, y_test)
print(s)

y_pred = knn.predict(x_test)
confusion_matrix(y_test, y_pred)
```

```
print(classification_report(y_test, y_pred))

plt.title('KNN Vari')
plt.plot(neighbors, test_accuray, label='Testing Accuracy')
plt.plot(neighbors, train_accuracy, label='Training Accuracy')
plt.legend()
plt.xlabel('Number of neighbours')
plt.ylabel('Accuracy')
plt.show()
```

-BY LAXMAN