CS3491 ARTIFICIAL INTELLIGENCE AND MACHINE LEARNING

1. Implementation of uniformed search algorithm (BFS ,DFS)
2. Implementation of Informed search algorithm (A*, Memory-bounded A*)
3. Implementation of naïve Bayes models
4. Implementation of Bayesian Networks
5. Build Regression Models
6. Build Decision Trees and Random Forests
7. Build SVM Models
8. Implementation Ensembling Techniques
9. Implementation Clustering Algorithms
10. Implementation EM for Bayesian networks
11. Build simple NN models
12. Build Deep learning NN models

1. A) BREADTH FIRST SEARCH

AIM:

To write a program for implementation of Breadth first search

ALGORITHM:

1. Start by putting any one of the group's vertices at the back of the queue.
2. Now take the front item of the queue and add it to the visited list.
3. Create a list of that vertex's adjacent nodes. Add those which are not within the visited list to the rear of the queue.
4. Keep continuing steps two and three till the queue is empty.
5. Stop the program

PROGRAM

```
graph = {
'5' : ['3','7'],
'3' : ['2', '4'],
'7' : ['8'],
'2' : [],
'4' : ['8'],
'8' : []
}

visited = [] # List for visited nodes.
queue =[]   #Initialize a queue

def bfs(visited, graph, node): #function for BFS
visited.append(node)
queue.append(node)

while queue:          # Creating loop to visit each node
    m =queue.pop(0)
print (m, end =" ")

for  neighbour in graph[m]:
if  neighbour not in visited:
visited.append(neighbour)
queue.append(neighbour)
# Driver Code
```

```python
print("Following is the Breadth-First Search")
bfs(visited, graph, '5')    # function calling
```

Output:

Following is the Breadth-First Search

5   3 7 2 4 8

RESULT

Thus the program for implementation of breadth first search has been executed successfully.

1. B) DEPTH FIRST SEARCH

AIM:

To write a program for implementation of Depth first search

ALGORITHM:

1. Start by putting any one of the group's vertices at the back of the stack.
2. Take the top item of the stack and add it to the visited list.
3. Create a list of that vertex's adjacent nodes. Add the ones which aren't in the visited list to the top of the stack.
4. Keep repeating steps 2 and 3 until the stack is empty.
5. Stop the program

PROGRAM

```
graph = {
'5' : ['3','7'],
'3' : ['2', '4'],
'7' : ['8'],
'2' : [],
'4' : ['8'],
'8' : []
}
visited = set() # Set to keep track of visited nodes of graph.
def dfs(visited, graph, node): #function for dfs
 if node not in visited:
  print(node)
  visited.add(node)
  for neighbour in graph[node]:
    dfs(visited, graph, neighbour)
```

```
# Driver Code
print("Following is the Depth-First Search")
dfs(visited, graph, '5')
```

Output :

```
Following is the Depth-First Search
5
3
2
4
8
7
```

RESULT

Thus the program for implementation of depth first search has been executed successfully.

2. A) A* SEARCH

AIM:

To write a program for implementation of A* Search.

ALGORITHM:

1. Start the program.
2. A set of all states we might end up in all.
3. A finish check (a way to check if we're at the finished state)
4. A set of possible action(in this case, different directions of movement)
5. A traversal function (a function that will tell us where we'll end up if we go a certain direction)
6. A set of movement costs from state-to-state(which correspond to edges in the graph)
7. Stop the program

PROGRAM

```
def aStarAlgo(start_node, stop_node):
open_set = set(start_node)
closed_set = set()
    g = {}              #store distance from starting node
    parents = {}        # parents contains an adjacency map of all nodes
    #distance of starting node from itself is zero
    g[start_node] = 0
    #start_node is root node i.e it has no parent nodes
    #so start_node is set to its own parent node
    parents[start_node] = start_node
    while len(open_set) > 0:
        n = None
        #node with lowest f() is found
        for v in open_set:
            if n == None or g[v] + heuristic(v) < g[n] + heuristic(n):
                n = v
        if n == stop_node or Graph_nodes[n] == None:
            pass
        else:
            for (m, weight) in get_neighbors(n):
                #nodes 'm' not in first and last set are added to first
```

```
            #n is set its parent
            if m not in open_set and m not in closed_set:
open_set.add(m)
               parents[m] = n
               g[m] = g[n] + weight
            #for each node m,compare its distance from start i.e g(m) to the
            #from start through n node
            else:
               if g[m] > g[n] + weight:
                  #update g(m)
                  g[m] = g[n] + weight
                  #change parent of m to n
                  parents[m] = n
                  #if m in closed set,remove and add to open
                  if m in closed_set:
closed_set.remove(m)
open_set.add(m)
      if n == None:
print('Path does not exist!')
         return None
      # if the current node is the stop_node
      # then we begin reconstructin the path from it to the start_node
      if n == stop_node:
         path = []
         while parents[n] != n:
path.append(n)
            n = parents[n]
path.append(start_node)
path.reverse()
print('Path found: {}'.format(path))
         return path
      # remove n from the open_list, and add it to closed_list
      # because all of his neighbors were inspected
open_set.remove(n)
closed_set.add(n)
print('Path does not exist!')
   return None
#define fuction to return neighbor and its distance
#from the passed node
def get_neighbors(v):
```

```
    if v in Graph_nodes:
        return Graph_nodes[v]
    else:
        return None
def heuristic(n):
H_dist = {
        'A': 11,
        'B': 6,
        'C': 5,
        'D': 7,
        'E': 3,
        'F': 6,
        'G': 5,
        'H': 3,
        'I': 1,
        'J': 0
    }
    return H_dist[n]
#Describe your graph here
Graph_nodes = {
    'A': [('B', 6), ('F', 3)],
    'B': [('A', 6), ('C', 3), ('D', 2)],
    'C': [('B', 3), ('D', 1), ('E', 5)],
    'D': [('B', 2), ('C', 1), ('E', 8)],
    'E': [('C', 5), ('D', 8), ('I', 5), ('J', 5)],
    'F': [('A', 3), ('G', 1), ('H', 7)],
    'G': [('F', 1), ('I', 3)],
    'H': [('F', 7), ('I', 2)],
    'I': [('E', 5), ('G', 3), ('H', 2), ('J', 3)],
}
aStarAlgo('A', 'J')
```

Output :

Path found: ['A', 'F', 'G', 'I', 'J']

RESULT

Thus the program for implementation of A* search has been executed successfully.

3. NAÏVE BAYES MODELS

AIM:

To write a program for implementation of Naïve Bayes Models.

ALGORITHM:

1. Start the program.
2. Import the numpy and matplotlib for plot the datasets.
3. create a dataframe from the encoded lists.
4. Visualize the datasets with the list of dataframe.
5. Stop the program

PROGRAM:

```
import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

Outlook ['sunny’ ,’sunny' ,'sunny’ ,’sunny'  , 'overcast', 'rainy, rainy', 'rainy', 'overcast, 'sunny', 'sunny', 'rainy', 'sunny', 'overcast', 'overcast', 'rainy"]

Temp = ['hot', 'hot', 'hot', 'mild', 'cool', 'cool', 'cool', 'mild', 'cool', 'mild', 'mild', 'mild', 'hot', 'mild"]

Humidity ['high', 'high', 'high', 'high', 'normal', 'normal', 'normal', 'high', 'normal', 'normal', 'normal', 'high', 'normal', 'high']

Windy = ['false', 'true', 'false', 'true',  'false' 'false', 'false',' ,'true' 'false', 'false', 'true', 'true', 'false' 'true', 'false', 'true' ]

Play ['no', 'no', 'yes', 'yes', 'yes', 'no', 'yes', 'no', 'yes', 'yes', 'yes', 'yes', 'yes', 'no']

weatherdata = pd.DataFrame({'Outlook': Outlook, ‘Temp’: Temp, 'Humidity': Humidity, 'Windy': Windy, 'Play': Play})

print (weatherdata.head())
```

OUTPUT:

```
    Outlook  Temp Humidity  Windy Play
0     sunny   hot     high  false   no
1     sunny   hot     high   true   no
2  overcast   hot     high  false  yes
3     rainy  mild     high  false  yes
4     rainy  cool   normal  false  yes
```

create a dataframe from the above lists.

Weatherdata= pd.DataFrame(('Outlook': Outlook, 'Temp' :Temp, 'Humidity' : Humidity,'windy' windy,'Play': Play})

print(Weatherdata)

fromsklearn import preprocessing

NaiveBayes_Weather.ipynb – Colaboratory

le= preprocessing.LabelEncoder()

outlook =le.fit_transform (Outlook)

temp = le.fit_transform(Temp)

humidity =le.fit_transform(Humidity)

windy =le.fit_transform(Windy)

play =le.fit_transform(Play)

create a dataframe from the encoded lists.

weatherFeatures = pd.DataFrame({'outlook': outlook, 'temp: temp, 'humidity': humidity, 'windy': windy,})

print (weatherFeatures.head())

print("Play = ", play)

OUTPUT:

```
print('Play = ', play)
      outlook  temp  humidity  windy
   0        2     1         0      0
   1        2     1         0      1
   2        0     1         0      0
   3        1     2         0      0
   4        1     0         1      0
   Play =  [0 0 1 1 1 0 1 0 1 1 1 1 1 0]
```

VISUALIZE THE DATASET

data2d weatherFeatures.loc[:, ['outlook', 'windy']]

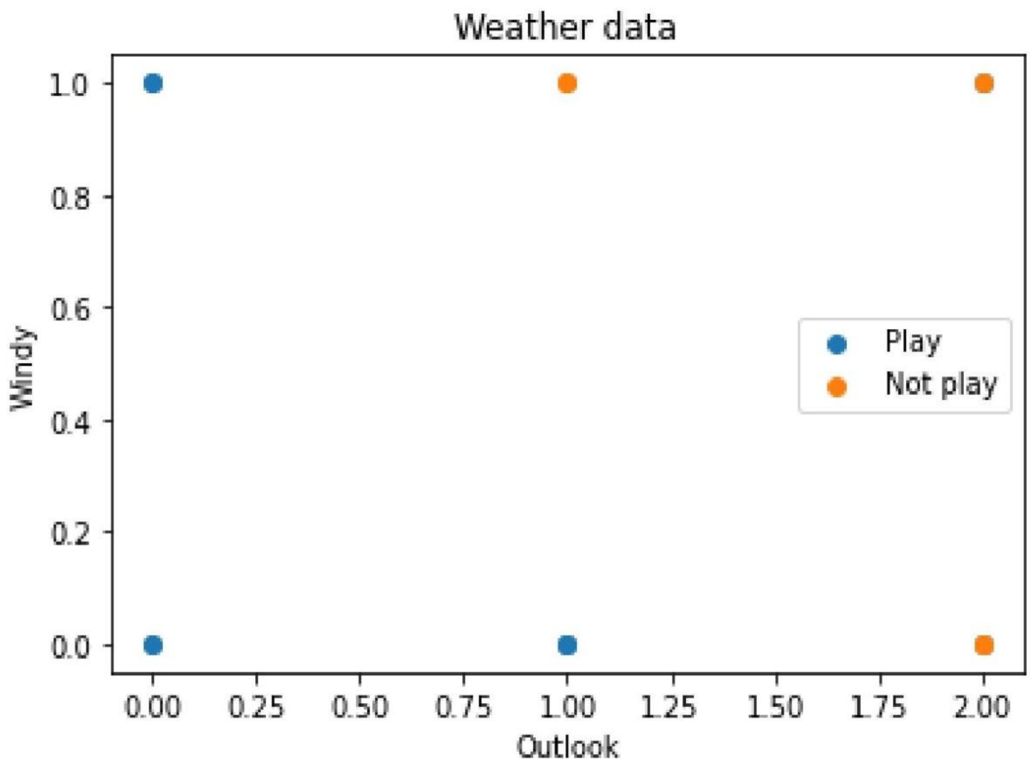
pos = data2d.loc[play == 1]

neg= data2d.loc[play == 0]

plt.scatter (pos.iloc[:, e], pos.iloc[:, 1], label-'Play')

```
plt.scatter (neg.iloc[:, 0], neg.iloc[:, 1], label-'Not play')

plt.xlabel('Outlook')

plt.ylabel('Windy')

plt.title('Weather data')

plt.legend()
```

OUTPUT:



RESULT

Thus the program for implementation of naïve bayes model has been executed successfully.

4. BAYESIAN NETWORK

AIM:

To write a program for implementation of Bayesian Network.

ALGORITHM:

1. Start the program.
2. Calculate the probability
3. Find probability with each attribute for each class.
4. Put these values in Bayes formula and calculate probability.
5. See which class has a higher probability, given the input belongs to the higher probability class.
6. Stop the program

PROGRAM:

```
import numpy as np

import pandas as pd

import csv

from pgmpy.estimators import MaximumLikelihoodEstimator

from pgmpy.models import BayesianModel

from pgmpy.inference import VariableElimination

heartDisease = pd.read_csv('heart.csv')

heartDisease = heartDisease.replace('?',np.nan)

print('Sample instances from the dataset are given below')

print(heartDisease.head())

print('\n Attributes and datatypes')

print(heartDisease.dtypes)
```

```
model=
BayesianModel([('age','heartdisease'),('sex','heartdisease'),('exang','heartdisease'),('cp',
'heartdisease'),('heartdisease','restecg'),('heartdisease','chol')])

print('\nLearning CPD using Maximum likelihood estimators')

model.fit(heartDisease,estimator=MaximumLikelihoodEstimator)

print('\n Inferencing with Bayesian Network:')

HeartDiseasetest_infer = VariableElimination(model)

print('\n 1. Probability of HeartDisease given evidence= restecg')

q1=HeartDiseasetest_infer.query(variables=['heartdisease'],evidence={'restecg':1})

print(q1)

print('\n 2. Probability of HeartDisease given evidence= cp ')

q2=HeartDiseasetest_infer.query(variables=['heartdisease'],evidence={'cp':2})

print(q2)
```

```
Learning CPD using Maximum likelihood estimators

 Inferencing with Bayesian Network:

 1. Probability of HeartDisease given evidence= restecg

   +------------------+----------------------+
   | heartdisease     |   phi(heartdisease)  |
   +==================+======================+
   | heartdisease(0)  |               0.1012 |
   +------------------+----------------------+
   | heartdisease(1)  |               0.0000 |
   +------------------+----------------------+
   | heartdisease(2)  |               0.2392 |
   +------------------+----------------------+
   | heartdisease(3)  |               0.2015 |
   +------------------+----------------------+
   | heartdisease(4)  |               0.4581 |
   +------------------+----------------------+
```

```
2. Probability of HeartDisease given evidence= cp
```

heartdisease	phi(heartdisease)
heartdisease(0)	0.3610
heartdisease(1)	0.2159
heartdisease(2)	0.1373
heartdisease(3)	0.1537
heartdisease(4)	0.1321

RESULT

Thus the program for implementation of Bayesian Network has been executed successfully.

5. Build Regression model

AIM:

To write a program for implementation of Build Regression Models

ALGORITHM:

1. Start the program.
2. Import the packages numpy and the class Linear Regression.
3. Defining data to work. The input and output should be array or similar objects.
4. Create a linear regression model and fit using the existing data.
5. You can get the result to check whether the model works .
6. Stop the program

PROGRAM:

```
Import numpy as np
Import matplotlib. pyplot as plt

def estimate_coef(x, y):
    # number of observations/points
    n =np.size(x)

    # mean of x and y vector
    m_x =np.mean(x)
    m_y =np.mean(y)

    # calculating cross-deviation and deviation about x
    SS_xy =np.sum(y*x) -n*m_y*m_x
    SS_xx =np.sum(x*x) -n*m_x*m_x

    # calculating regression coefficients
    b_1 =SS_xy /SS_xx
    b_0 =m_y -b_1*m_x
    return(b_0, b_1)
def plot_regression_line(x, y, b):
    # plotting the actual points as scatter plot
    plt.scatter(x, y, color ="m",
              marker ="o", s =30)
    # predicted response vector
    y_pred =b[0] +b[1]*x
    # plotting the regression line
    plt.plot(x, y_pred, color ="g")
```

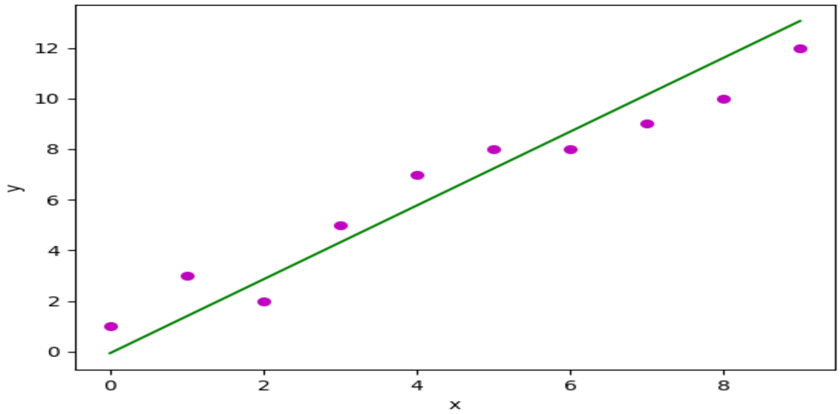
```
    # putting labels
    plt.xlabel('x')
    plt.ylabel('y')
    # function to show plot
    plt.show()
def main():
    # observations / data
    x =np.array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
    y =np.array([1, 3, 2, 5, 7, 8, 8, 9, 10, 12])

    # estimating coefficients
    b =estimate_coef(x, y)
    print("Estimated coefficients:\nb_0 ={}  \
        \nb_1 ={}".format(b[0], b[1]))
    # plotting regression line
    plot_regression_line(x, y, b)
if__name__ =="__main__":
    main()
```

Estimated coefficients:

b_0 = -0.0586206896552

b_1 = 1.45747126437

And graph obtained looks like this:



RESULT

Thus, the above program for implementation of Build Regression Models has been executed successfully.

6. Build Decision Trees and Random Forests

AIM:

To write a program for implementation of Build Decision Trees and Random Forests

ALGORITHM:

1. Start the program.
2. Select random sample from a given data or training set.
3. Construct a decision tree for each sample and considers all predicted output of those decision tree.
4. Construct the decision tree that you want to build.
5. Repeat step 2 &3
6. For a new data points, find the predictions of each decision tree, and assign the new data points category to complete the finalized report in matrix form.
7. Stop the program.

PROGRAM:

```
import pandas
from sklearn import tree
from sklearn.tree import DecisionTreeClassifier
import matplotlib.pyplot as plt

df = pandas.read_csv("data.csv")

d = {'UK': 0, 'USA': 1, 'N': 2}
df['Nationality'] = df['Nationality'].map(d)
d = {'YES': 1, 'NO': 0}
df['Go'] = df['Go'].map(d)

features = ['Age', 'Experience', 'Rank', 'Nationality']

X = df[features]
y = df['Go']

dtree = DecisionTreeClassifier()
dtree = dtree.fit(X, y)

tree.plot_tree(dtree, feature_names=features)
```

output:

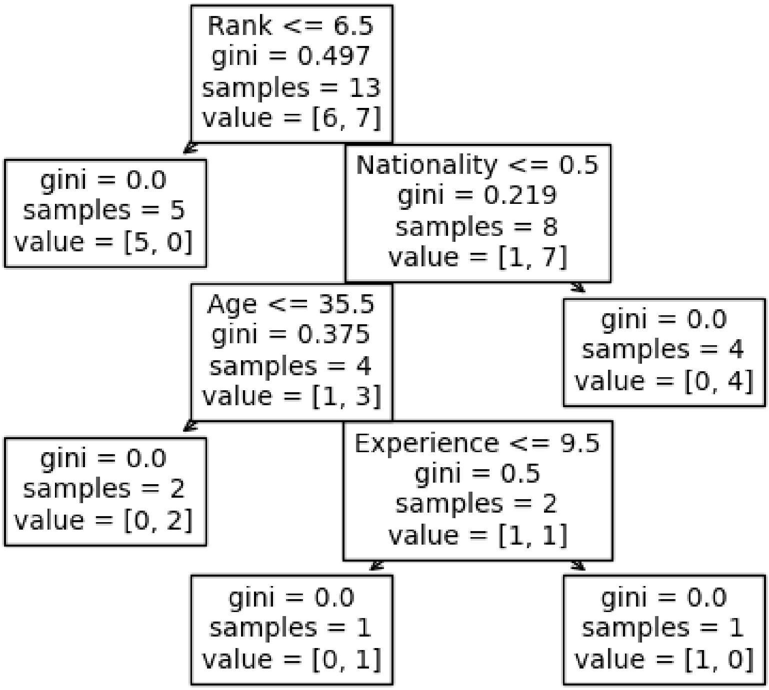
Random Forest

Random Forest

Random Forest

Random Forest

rando



Random Forest

```
from sklearn.ensemble import RandomForestClassifier

rf_clf = RandomForestClassifier(n_estimators=100)
rf_clf.fit(X_train, y_train)

print_score(rf_clf, X_train, y_train, X_test, y_test, train=True)
print_score(rf_clf, X_train, y_train, X_test, y_test, train=False)
```

Train Result:

Accuracy Score: 100.00%
CLASSIFICATION REPORT:

	0	1	accuracy	macro avg	weighted avg
precision	1.0	1.0	1.0	1.0	1.0
recall	1.0	1.0	1.0	1.0	1.0
f1-score	1.0	1.0	1.0	1.0	1.0
support	853.0	176.0	1.0	1029.0	1029.0

Confusion Matrix:
[[853   0]
[0 176]]

Test Result:
Accuracy Score: 86.85%

CLASSIFICATION REPORT:

	0	1	accuracy	macro avg	weighted avg
precision	0.874419	0.636364	0.868481	0.755391	0.841490
recall	0.989474	0.114754	0.868481	0.552114	0.868481
f1-score	0.928395	0.194444	0.868481	0.561420	0.826874
support	380.000000	61.000000	0.868481	441.000000	441.000000

Confusion Matrix:
[[376   4]
[54   7]]

RESULT

Thus the program for implementation of Build Decision Trees and Random Forest has been executed successfully.

7. Build SVM Models

AIM:

To write a program for implementation of Build SVM Models

ALGORITHM:

1. Start the program.
2. Import the required packages
3. Load files(excel/csv/text) into a dataframe.
4. Display the rows and describe the data set using built in method.
5. Visualize the data set using SVC model.
6. Stop the program

PROGRAM:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder
from sklearn.preprocessing import MinMaxScaler
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score

data = pd.read_csv(&#39;archive.zip&#39;)
data.head()
```

OUTPUT:

	mean_radius	mean_texture	mean_perimeter	mean_area	mean_smoothness	diagnosis
0	17.99	10.38	122.80	1001.0	0.11840	0
1	20.57	17.77	132.90	1326.0	0.08474	0
2	19.69	21.25	130.00	1203.0	0.10960	0
3	11.42	20.38	77.58	386.1	0.14250	0
4	20.29	14.34	135.10	1297.0	0.10030	0

PROGRAM

```
data.isna().sum()
```

OUTPUT

```
mean_radius        0
mean_texture       0
mean_perimeter     0
mean_area          0
mean_smoothness    0
diagnosis          0
dtype: int64
```

PROGRAM

```
data.describe()
```

OUTPUT

1 to 8 of 8 entries Filter

index	mean_radius	mean_texture	mean_perimeter	mean_area	mean_smoothness	diagnosis
count	569.0	569.0	569.0	569.0	569.0	569.0
mean	14.127291739894552	19.289648506151142	91.96903339191564	654.8891036906855	0.0963602811950791	0.6274165202108963
std	3.5240488262120775	4.301035768166949	24.298981038754906	351.914129181653	0.01406412813767362	0.48391795640316865
min	6.981	9.71	43.79	143.5	0.05263	0.0
25%	11.7	16.17	75.17	420.3	0.08637	0.0
50%	13.37	18.84	86.24	551.1	0.09587	1.0
75%	15.78	21.8	104.1	782.7	0.1053	1.0
max	28.11	39.28	188.5	2501.0	0.1634	1.0

Show 25 per page

PROGRAM

```
data.info()
```

OUTPUT

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 569 entries, 0 to 568
Data columns (total 6 columns):
 #   Column           Non-Null Count  Dtype
```

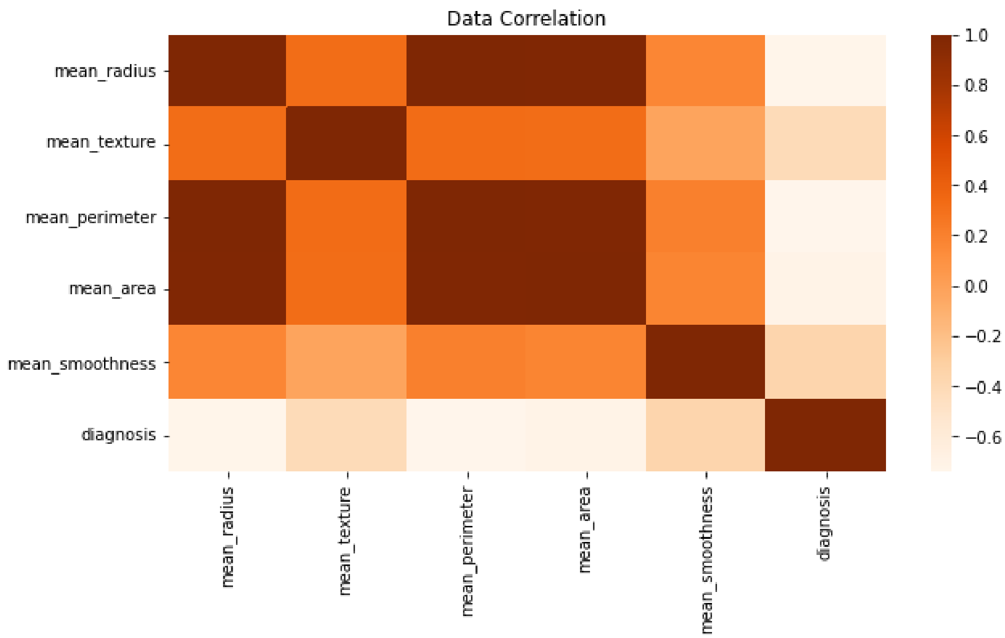
```
---  ------          --------------  -----
 0   mean_radius      569 non-null    float64
 1   mean_texture     569 non-null    float64
 2   mean_perimeter   569 non-null    float64
 3   mean_area        569 non-null    float64
 4   mean_smoothness  569 non-null    float64
 5   diagnosis        569 non-null    int64
dtypes: float64(5), int64(1)
memory usage: 26.8 KB
```

PROGRAM

```
corr = data.corr()

fig = plt.figure(figsize=(10,5))

a = sns.heatmap(corr, cmap='Oranges')

a.set_title("Data Correlation")
```

OUTPUT

```
Text(0.5, 1.0, 'Data Correlation')
```



RESULT

Thus the program for implementation of Build SVM Models has been executed successfully.

8. Ensembling techniques

AIM:

To write a program for implementation of Ensembling techniques

ALGORITHM:

1. Start the program.
2. Split the train data set into n parts.
3. A base model is fitted on the whole data set. This model is used to predict the test data set.
4. The step 2 and 3 are repeated for another base model which result in another set of predictions for the train and test data set.
5. The prediction on train data set are used as a feature to build the new models.
6. The final model is used to make the prediction on test data set.
7. Stop the program

PROGRAM:

```
From
sklearn.datasets
import
load_breast_cancer
from sklearn.model_selection import train_test_split

from sklearn.tree import DecisionTreeClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.linear_model import LogisticRegression

from sklearn.ensemble import VotingClassifier
from sklearn.metrics import accuracy_score


class Ensemble:
    def __init__(self):
self.x_train = None
self.x_test = None
self.y_train = None
self.y_test = None

    def load_data(self):
        x, y = load_breast_cancer(return_X_y=True)
self.x_train, self.x_test, self.y_train, self.y_test =
train_test_split(x, y, test_size=0.25, random_state=23)
```

```
    @staticmethod
    def __Classifiers__(name=None):
        # See for reproducibility
random_state = 23

        if name == 'decision_tree':
            return
DecisionTreeClassifier(random_state=random_state)
        if name == 'kneighbors':
            return KNeighborsClassifier()
        if name == 'logistic_regression':
            return LogisticRegression(random_state=random_state,
solver='liblinear')

    def __DecisionTreeClassifier__(self):

        # Decision Tree Classifier
decision_tree =
Ensemble.__Classifiers__(name='decision_tree')

        # Train Decision Tree
decision_tree.fit(self.x_train, self.y_train)

    def __KNearestNeighborsClassifier__(self):

        # K-Nearest Neighbors Classifier
knn = Ensemble.__Classifiers__(name='kneighbors')

        # Train K-Nearest Neighbos
knn.fit(self.x_train, self.y_train)

    def __LogisticRegression__(self):

        # Decision Tree Classifier
logistic_regression =
Ensemble.__Classifiers__(name='logistic_regression')

        # Init Grid Search
logistic_regression.fit(self.x_train, self.y_train)

    def __VotingClassifier__(self):

        # Instantiate classifiers
decision_tree =
```

```
Ensemble.__Classifiers__(name='decision_tree')
knn = Ensemble.__Classifiers__(name='kneighbors')
logistic_regression =
Ensemble.__Classifiers__(name='logistic_regression')

        # Voting Classifier initialization
vc = VotingClassifier(estimators=[('decision_tree',
decision_tree),
                                   ('knn', knn), ('logistic_regression',
logistic_regression)], voting='soft')

        # Fitting the vc model
vc.fit(self.x_train, self.y_train)

        # Getting train and test accuracies from meta_model
y_pred_train = vc.predict(self.x_train)
y_pred = vc.predict(self.x_test)

        print(f"Train accuracy: {accuracy_score(self.y_train,
y_pred_train)}")
        print(f"Test accuracy: {accuracy_score(self.y_test,
y_pred)}")


if __name__ == "__main__":
    ensemble = Ensemble()
ensemble.load_data()
    ensemble.__VotingClassifier__()
```

Output:

```
        Train accuracy: 0.9882629107981221
Test accuracy: 0.965034965034965
```

RESULT

Thus, the program for implementation of Ensembling techniques has been executed successfully.

9. Clustering Algorithm

AIM:

To write a program for implementation of Clustering Algorithm.

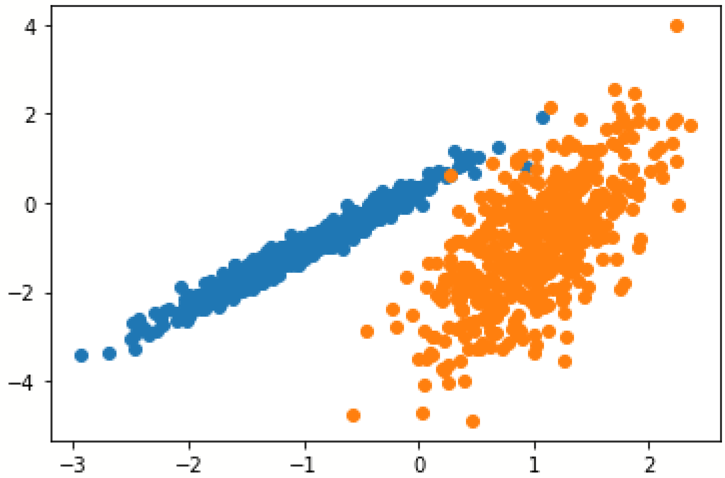
ALGORITHM:

1. Start the program.
2. Choose some values of K and run the clustering algorithm.
3. For each cluster, compute the within-cluster sum-of-square between the centroid and each data point.
4. Sum up for all cluster, plot on a graph.
5. Repeat for different values of K, Keep plotting on the graph.
6. Stop the program.

PROGRAM:

```
# synthetic classification dataset

from numpy import where

from sklearn.datasets import make_classification

from matplotlib import pyplot

# define dataset

X, y = make_classification(n_samples=1000, n_features=2, n_informative=2, n_redundant=0, n_clusters_per_class=1, random_state=4)

# create scatter plot for samples from each class

for class_value in range(2):

 # get row indexes for samples with this class

 row_ix = where(y == class_value)

 # create scatter of these samples

 pyplot.scatter(X[row_ix, 0], X[row_ix, 1])

# show the plot

pyplot.show()
```

OUTPUT:



RESULT

Thus, the program for implementation of Clustering Algorithm has been executed successfully.

10. EM Algorithm for Bayesian Network

Aim

To implement the Expectation–Maximization (EM) algorithm for parameter learning in a Bayesian Network with hidden variables.

Algorithm

Step 1: Initialize the probabilities:

- P(Rain)
- P(Traffic | Rain)

Step 2 (E-Step): For each observed data instance:

- Compute the probability of hidden variables using Bayes' rule
- Calculate expected counts

Step 3 (M-Step): Update probabilities using expected counts

- Maximize expected log-likelihood

Step 4:Repeat E-step and M-step until convergence or fixed iterations.

PROGRAM:

```
import random

# Observed data (Traffic only)

data = ["Heavy", "Light", "Heavy", "Heavy", "Light"]

# Initialize parameters randomly

P_rain = 0.5  # P(Rain = Yes)

P_traffic_given_rain = {

    "Yes": {"Heavy": 0.6, "Light": 0.4},

    "No": {"Heavy": 0.3, "Light": 0.7}

}

# EM Algorithm

iterations = 10

for it in range(iterations):
```

```
    # E-step: Expected counts
    expected_rain_yes = 0
    expected_rain_no = 0
    traffic_counts = {
        "Yes": {"Heavy": 0, "Light": 0},
        "No": {"Heavy": 0, "Light": 0}
    }
    for t in data:
        # Bayes rule
        prob_yes = P_rain * P_traffic_given_rain["Yes"][t]
        prob_no = (1 - P_rain) * P_traffic_given_rain["No"][t]
        total = prob_yes + prob_no
        prob_yes /= total
        prob_no /= total
        expected_rain_yes += prob_yes
        expected_rain_no += prob_no
        traffic_counts["Yes"][t] += prob_yes
        traffic_counts["No"][t] += prob_no
    # M-step: Update parameters
    P_rain = expected_rain_yes / len(data)
    for rain in ["Yes", "No"]:
        total = sum(traffic_counts[rain].values())
        for t in ["Heavy", "Light"]:
            P_traffic_given_rain[rain][t] = traffic_counts[rain][t] / total
print("Final Estimated Parameters:")
print("P(Rain=Yes):", round(P_rain, 3))
print("P(Traffic | Rain):")
print(P_traffic_given_rain)
```

OUTPUT:

Final Estimated Parameters:

P(Rain=Yes): 0.57

P(Traffic | Rain):

{'Yes': {'Heavy': 0.72, 'Light': 0.28},

'No': {'Heavy': 0.34, 'Light': 0.66}}

Result

The EM algorithm successfully estimated the parameters of the Bayesian Network with hidden variables.

11. Build simple NN models

AIM:

To write a program for implementation of Build simple NN models.

ALGORITHM:

1. Start the program.
2. Create a class with random number generation.
3. Convert the weight to 3 by 1 matrix with the values and mean.
4. Applying the sigmoid function.
5. Training the model to make accurate predictions while adjusting weights continually and performing weight adjustments.
6. Stop the program

PROGRAM:

```
import numpy as np

class NeuralNetwork():

  def __init__(self):

    # seeding for random number generation

    np.random.seed(1)

    #converting weights to a 3 by 1 matrix with values from -1 to 1 and mean of 0

    self.synaptic_weights = 2 * np.random.random((3, 1)) - 1

  def sigmoid(self, x):

    #applying the sigmoid function

    return 1 / (1 + np.exp(-x))

  def sigmoid_derivative(self, x):

    #computing derivative to the Sigmoid function

    return x * (1 - x)

  def train(self, training_inputs, training_outputs, training_iterations):

    #training the model to make accurate predictions while adjusting weights
continually

    for iteration in range(training_iterations):

      #siphon the training data via  the neuron
```

```
            output = self.think(training_inputs)

            #computing error rate for back-propagation

            error = training_outputs - output

            #performing weight adjustments

            adjustments = np.dot(training_inputs.T, error *
self.sigmoid_derivative(output))

            self.synaptic_weights += adjustments

    def think(self, inputs):

        #passing the inputs via the neuron to get output

        #converting values to floats

        inputs = inputs.astype(float)

        output = self.sigmoid(np.dot(inputs, self.synaptic_weights))

        return output

if __name__ == "__main__":

    #initializing the neuron class

    neural_network = NeuralNetwork()

    print("Beginning Randomly Generated Weights: ")

    print(neural_network.synaptic_weights)

    #training data consisting of 4 examples--3 input values and 1 output

    training_inputs = np.array([[0,0,1],

                                [1,1,1],

                                [1,0,1],

                                [0,1,1]])

    training_outputs = np.array([[0,1,1,0]]).T

    #training taking place

    neural_network.train(training_inputs, training_outputs, 15000)

    print("Ending Weights After Training: ")

    print(neural_network.synaptic_weights)
```

```
    user_input_one = str(input("User Input One: "))
    user_input_two = str(input("User Input Two: "))
    user_input_three = str(input("User Input Three: "))
    print("Considering New Situation: ", user_input_one, user_input_two,
user_input_three)
    print("New Output data: ")
    print(neural_network.think(np.array([user_input_one, user_input_two,
user_input_three])))
    print("Wow, we did it!")
```

OUT PUT

```
Beginning Randomly Generated Weights:
[[-0.16595599]
 [ 0.44064899]
 [-0.99977125]]
Ending Weights After Training:
[[10.08740896]
 [-0.20695366]
 [-4.83757835]]
User Input One: 49
User Input Two: 76
User Input Three: 1
Considering New Situation:  49 76 1
New Output data: [1.]
```

RESULT

Thus, the program for implementation of build simple NN models has been executed successfully.

12. Build deep learning NN models

AIM:

To write a program for implementation of Build deep learning NN models.

ALGORITHM:

1. Start the program.
2. Generate random number within a bound.
3. Set the number of neurons/nodes for each layer and initialize the weight matrices
4. Run simple_network for arrays, lists and tuples with shape and get result.
5. Stop the program

PROGRAM:

```
# Import python
n libraries required in this example:
import numpy as np
from scipy.special import expit as activation_function
from scipy.stats import truncnorm

# DEFINE THE NETWORK

# Generate random numbers within a truncated (bounded)
# normal distribution:
def truncated_normal(mean=0, sd=1, low=0, upp=10):
   return truncnorm(
      (low - mean) / sd, (upp - mean) / sd, loc=mean, scale=sd)

# Create the ‘Nnetwork’ class and define its arguments:
# Set the number of neurons/nodes for each layer
# and initialize the weight matrices:
class Nnetwork:
   def __init__(self,
            no_of_in_nodes,
            no_of_out_nodes,
            no_of_hidden_nodes,
            learning_rate):
      self.no_of_in_nodes = no_of_in_nodes
      self.no_of_out_nodes = no_of_out_nodes
      self.no_of_hidden_nodes = no_of_hidden_nodes
      self.learning_rate = learning_rate
      self.create_weight_matrices()
   def create_weight_matrices(self):
      """ A method to initialize the weight matrices of the neural network"""
```

```
        rad = 1 / np.sqrt(self.no_of_in_nodes)
        X = truncated_normal(mean=0, sd=1, low=-rad, upp=rad)
        self.weights_in_hidden = X.rvs((self.no_of_hidden_nodes,
                                        self.no_of_in_nodes))
        rad = 1 / np.sqrt(self.no_of_hidden_nodes)
        X = truncated_normal(mean=0, sd=1, low=-rad, upp=rad)
        self.weights_hidden_out = X.rvs((self.no_of_out_nodes,
                                         self.no_of_hidden_nodes))
    def train(self, input_vector, target_vector):
        pass # More work is needed to train the network
            def run(self, input_vector):
        """
        running the network with an input vector 'input_vector'.
        'input_vector' can be tuple, list or ndarray
        """
        # Turn the input vector into a column vector:
        input_vector = np.array(input_vector, ndmin=2).T
        # activation_function() implements the expit function,
        # which is an implementation of the sigmoid function:
        input_hidden = activation_function(self.weights_in_hidden @   input_vector)
        output_vector = activation_function(self.weights_hidden_out @ input_hidden)
        return output_vector
# RUN THE NETWORK AND GET A RESULT
# Initialize an instance of the class:
simple_network = Nnetwork(no_of_in_nodes=2,
                          no_of_out_nodes=2,
                          no_of_hidden_nodes=4,
                          learning_rate=0.6)
# Run simple_network for arrays, lists and tuples with shape (2):
# and get a result:
simple_network.run([(3, 4)])
```

OUTPUT:

```
array([[0.4724468 ],
 [0.59443007]])
```

RESULT

Thus, the program for implementation of build deep learning NN models has been executed successfully.