**EXP NO: 1(A)**

**DATE:**

**UNINFORMED SEARCH ALGORITHMS**

**BFS**

**AIM:**

To write and implement the python program for breadth first search using uniformed searching techniques.

**DESCRIPTION:**

Breadth First Search is simple strategy in which the root node is expanded first, then all the successors of the root nodes are expanded next, then their successors, and so on. In general, all the nodes are expanded at a given depth in the search tree before any nodes at the next level are expanded.

Breadth First Search is an instance of the general graph-search algorithm in which the shallowest unexpanded node is chosen for expansion.

**ALGORITHM:**

1. **Initialize:** Choose a starting node, mark it as visited, and enqueue it into a queue.

2. **While the queue is not empty:**

•Dequeue a node from the queue.

•Visit the dequeued node.

•Enqueue all adjacent nodes of the dequeued node that have not been visited yet, and mark them as visited.

3. Repeat step 2 until the queue is empty.

**PROGRAM:**

def bfs (G, S):

visited = []

queue = []

queue. append (S)

visited. append (S)

bfstree = []

while queue:

t = queue.pop (0)

bfstree. append (t)

for n in G[t]:

if n not in visited:

visited. append(n)

queue. append(n)

return bfstree

graph = {}

n=int (input ("Enter number of vertices in graph:"))

for i in range (n):

neighbour = []

node = input ("Enter name of node: ")

neighbour = input ("Enter name of neighbour node "). split ()

graph [node]=neighbour

S=input ("Enter starting node name:")

print (graph)

print ("BFS traversal")

print (bfs (graph, S))

**OUTPUT:**

Enter number of vertices in graph:6

Enter name of node: A

Enter name of neighbour node B C

Enter name of node: B

Enter name of neighbour node A D F

Enter name of node: C

Enter name of neighbour node A D

Enter name of node: D

Enter name of neighbour node B C E F

Enter name of node: E

Enter name of neighbour node D

Enter name of node: F

Enter name of neighbour node B D

Enter starting node name:A

{'A': ['B', 'C'], 'B': ['A', 'D', 'F'], 'C': ['A', 'D'], 'D': ['B', 'C', 'E', 'F'], 'E': ['D'], 'F': ['B', 'D']}

BFS traversal

['A', 'B', 'C', 'D', 'F', 'E']

**RESULT:**

Thus, the program to demonstrate breadth first search was written and executed successfully.

**EXP NO: 1(B)**

**DATE:**

**UNIFORMED SEARCH TECHNIQUES**

**DFS**

**AIM:**

To write and implement the python program for breadth first search using uniformed searching techniques.

**DESCRIPTION:**

Depth First Search always expands the deepest node in the current frontier of the search tree. The search proceeds immediately to the deepest level of the search tree, where the nodes have no successors. As those nodes are expanded, they are dropped from the frontier, so then the search “back up” to the next deepest node that still has unexplored successors.

**ALGORITHM:**

1. **Start with a node:** Choose a starting node and mark it as visited.
2. **Explore neighbors recursively:** Explore the unvisited neighbors of the current node recursively.
3. **Backtrack:** If all neighbors are visited or there are no neighbors, backtrack to the previous node and continue exploration from there.

**PROGRAM:**

def dfs (G, S, dfstree):

for n in G[S]:

if n not in dfstree:

dfstree[n]=S

dfs (G, n, dfstree)

return dfstree

graph= {}

n=int (input ('Enter number of vertices in graph: '))

for i in range (n):

neighbour= []

node=input ('Enter name of node: ')

neighbour=input ('Enter name of neighbour node:'). split ()

graph [node]=neighbour

S=input ('Enter starting node name:')

print ('DFS traversal')

dfstree= {S: None}

print (dfs (graph, S, dfstree))

**OUTPUT:**

Enter number of vertices in graph: 6

Enter name of node: A

Enter name of neighbour node:B C

Enter name of node: B

Enter name of neighbour node:A D F

Enter name of node: C

Enter name of neighbour node:A D

Enter name of node: D

Enter name of neighbour node:B C E F

Enter name of node: E

Enter name of neighbour node:D

Enter name of node: F

Enter name of neighbour node:B D

Enter starting node name:A

DFS traversal

{'A': None, 'B': 'A', 'D': 'B', 'C': 'D', 'E': 'D', 'F': 'D'}

**RESULT:**

Thus, the program to demonstrate depth first search was written and executed successfully.

**EXP NO: 2(A)**

**DATE:**

**A\* ALGORITHM**

**AIM:**

To find the most cost-effective path to reach the final state from initial state using A\* Algorithm.

**DESCRIPTION:**

A\* is the most commonly known form of Best-First Search. It uses heuristic function h(n), and cost to reach the node n from the start state g(n). A\* Algorithm finds the shortest path through the search space using the heuristic function. This search algorithm expands less search tree and provides optimal result faster.

A\* Algorithm uses heuristic as well as the cost to reach the node. Hence combining both costs and this sum is called as a fitness number.

f(n) = g(n) + h(n)

where,

f(n) – Estimated cost of the cheapest solution.

g(n) – Cost to reach node n from start node.

h(n) – Cost to reach from node n to goal node.

**ALGORITHM:**

1. Place the starting node in the OPEN list
2. C heck if the OPEN list is empty or not, if the list is empty then return failure and stop.
3. Select the node from the OPEN list which has the smallest value of evaluation function (g+h), if node n is goal node then return successors and stop, otherwise
4. Expand n and generate all of its successors, and put n into CLOSED list. For equal successors n’, check whether n’ is already in the OPEN or CLOSED list, if not then compute evaluation function for n’ and place into OPEN list.
5. Else if node n’ is already in OPEN and CLOSED then it should be attached to the back pointer which reflects the lowest g(n’) value.
6. Return to step 2.

**PROGRAM:**

def aStarAlgo (start\_node, stop\_node):

open\_list = set(start\_node)

closed\_list = set ()

g = {}

parents = {}

g[start\_node] = 0

parents[start\_node] = start\_node

while(len(open\_list) > 0):

n = None

for v in open\_list:

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):

if (m not in open\_list and m not in closed\_list):

open\_list.add(m)

parents[m] = n

g[m] = g[n] + weight

else:

if (g[m] > g[n] + weight):

g[m] = g[n] + weight

parents[m] = n

if (m in closed\_list):

closed\_list.remove(m)

open\_list.add(m)

if (n == None):

print ("Path does not exist!")

return None

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

open\_list.remove(n)

closed\_list.add(n)

print ("Path does not exist!")

return None

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]

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)],'J':[('E',5),('I',3)]}

S=str (input ("Enter the starting node :"))

G=str (input ("Enter the goal node :"))

aStarAlgo (S, G)

**OUTPUT:**

Enter the starting node :A

Enter the goal node :E

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

**RESULT:**

Thus, the program to demonstrate A\* algorithm was written and executed successfully.

**EXP NO: 2(B)**

**DATE:**

**MEMORY BOUNDED A\* ALGORITHM**

**AIM:**

To find the most cost effective path to reach the final state from initial state using Memory Bounded A\* Algorithm.

**DESCRIPTION:**

Memory Bounded A\* is used to find the shortest path. The main advantage is that it uses a bounded memory while A\* Algorithm might need exponential memory. All other characteristics of MBA\* are inherited from A\*.

**ALGORITHM:**

1. Initialize open a cloud set.
2. Initialize ‘g’ and ‘parents’ dictionary.
3. Iterate over search until the open set is empty or until the maximum depth is reached.
4. If goal node is found within the depth limit, Return the reconstructed path.

Pop node ‘n;.

If n is goal node, reconstruct path.

1. If no path is found within the depth limit or goal node is unreachable, Return ‘None’.

**PROGRAM:**

import heapq

def mem\_aStarAlgo(start\_node, stop\_node, max\_depth):

open\_set = [(heuristic(start\_node), start\_node, 0)] # Priority queue sorted by heuristic + depth

closed\_set = set()

g = {start\_node: 0} # Store the cost to reach each node from the start node

parents = {start\_node: start\_node} # Parents contain an adjacency map of all nodes

while open\_set:

z, n, depth = heapq.heappop(open\_set)

if n == stop\_node:

path = []

while parents[n] != n:

path.append(n)

n = parents[n]

path.append(start\_node)

path.reverse()

print("Path found:", path)

return path

if n in closed\_set or depth >= max\_depth:

continue

for (m, weight) in get\_neighbors(n):

if m not in closed\_set:

g\_cost = g[n] + weight

if m not in g or g\_cost < g[m]:

g[m] = g\_cost

heapq.heappush(open\_set, (g\_cost + heuristic(m), m, depth + 1))

parents[m] = n

closed\_set.add(n)

print("Path does not exist within the depth limit!")

return None

def get\_neighbors(v):

if v in Graph\_nodes:

return Graph\_nodes[v]

else:

return []

def heuristic(n):

Hdist = {'A': 11, 'B': 6, 'C': 99, 'D': 1, 'E': 7, 'G': 0}

return Hdist[n]

Graph\_nodes = {

'A': [('B', 2), ('E', 3)],

'B': [('C', 1), ('G', 9)],

'C': [],

'E': [('D', 6)],

'D': [('G', 1)]

}

mem\_aStarAlgo('A', 'G', 10)

**OUTPUT:**

Path found: ['A', 'E', 'D', 'G']

**RESULT:**

Thus, the program to demonstrate Memory Bounded A\* Algorithm was written and executed successfully.

**EXPT NO: 3(A)**

**DATE:**

**REGRESSION MODELS**

**LINEAR REGRESSION**

**AIM:**

To write a python program to implement the linear regression.

**DESCRIPTION:**

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

Linear Regression Algorithm shows a linear relationship between a dependent (y) and one or more independent (Y), variables, hence called as Linear Regression. Since it shows the linear relationship, which means it finds how the value of dependent variable is changing according to the value of the independent variable. Mathematically, Linear Regression can be expressed as:

y = a0 + a1x + e

where,

y – Independent variable.

a0 – Intercept of the line.

a1 – Linear Regression Coefficient.

x – Independent variable.

e – Random error.

**ALGORITHM:**

1. Initialize the parameter.
2. Predict the value of a dependent variable by given an independent variable.
3. Calculate the error in prediction for all data points.
4. Calculate partial derivation with respect to a0 and a1.
5. Calculate the cost for each number and add them.
6. Update the values of a0 and a1.

**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)

y\_pred = b[0] + b[1]\*x

plt.plot(x, y\_pred, color = "g")

plt.xlabel('x')

plt.ylabel('y')

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)

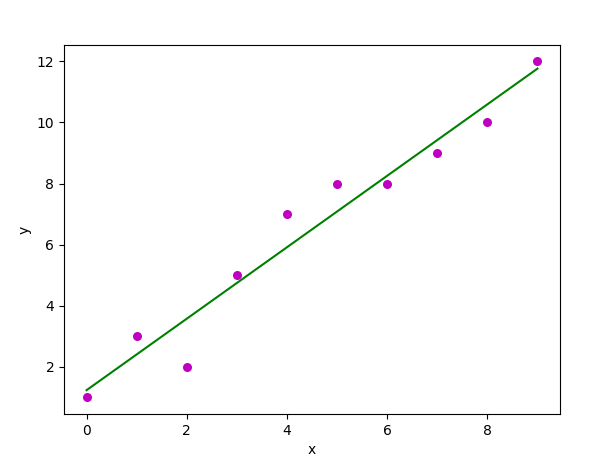
if \_\_name\_\_ == "\_\_main\_\_":

main ()

**OUTPUT:**

Estimated coefficients:

b\_0 = 1.2363636363636363 \nb\_1 = 1.1696969696969697



**RESULT:**

Thus, the program to implement the linear regression was written and executed successfully.

**EXP NO: 3(B)**

**DATE:**

**REGRESSION MODELS**

**MULTIPLE VARIABLE LINEAR REGRESSION**

**AIM:**

To write a python program to demonstrate multiple variable linear regression.

**DESCRIPTION:**

Multiple Variable Linear Regression is a statistical technique that uses multiple linear regression to model more complex relationships between two or more independent variables and one dependent variable. It is used when there are two or more variables.

**ALGORITHM:**

1. Data Preprocessing.
2. Importing the Libraries.
3. Importing the Data set.
4. Encoding the Categorical Data.
5. Avoiding the Dummy Variable Trap.
6. Splitting the Data set into Training set and Test set.
7. Fitting Multiple Linear Regression to the Training Set.
8. Predict the Test Set results.

**PROGRAM:**

import numpy as np

import pandas as pd

import statistics

import math

from matplotlib import pyplot as plt

import statsmodels.formula.api as smf

import requests # Module to process http/https requests

import warnings

# Suppress PyArrow deprecation warning

warnings.filterwarnings("ignore", category=DeprecationWarning)

remote\_url = "http://54.243.252.9/engr-1330-webroot/8-Labs/Lab29/heart.data.csv"

rget = requests.get(remote\_url, allow\_redirects=True)

open('heart.data.csv', 'wb').write(rget.content)

heartattack = pd.read\_csv('heart.data.csv')

data = heartattack.rename(columns={"biking": "Bike", "smoking": "Smoke", "heart.disease": "Disease"})

print(data.head(3))

# Initialise and fit linear regression model using statsmodels

model = smf.ols('Disease ~ Bike + Smoke', data=data)

model = model.fit()

print(model.summary())

dir(model)

# Activate to find attributes

intercept = model.params.iloc[0]

slope = model.params.iloc[1]

rsquare = model.rsquared

RMSE = math.sqrt(model.mse\_total)

# Predict values

heartfail = model.predict()

titleline = 'Disease Index versus Lifestyle Variables \n' + 'R squared = ' + str(round(rsquare, 3)) + ' \n RMSE = ' + str(round(RMSE, 2))

# Plot regression against actual data - Biking

plt.figure(figsize=(12, 6))

plt.plot(data['Bike'], data['Disease'], 'o') # Corrected variable name

plt.plot(data['Bike'], heartfail, marker='s', color='r', linewidth=0) # regression line

plt.xlabel('Biking (miles/week)')

plt.ylabel('Disease Index (Admissions/100,000 as per MMWR)')

plt.legend(['Observations', 'Model Prediction'])

plt.title(titleline)

plt.show()

# Plot regression against actual data - Smoking

plt.figure(figsize=(12, 6))

plt.plot(data['Smoke'], data['Disease'], 'o') # Corrected variable name

plt.plot(data['Smoke'], heartfail, marker='s', color='r', linewidth=0) # regression line

plt.xlabel('Smoking (packs/week)')

plt.ylabel('Disease Index (Admissions/100,000 as per MMWR)')

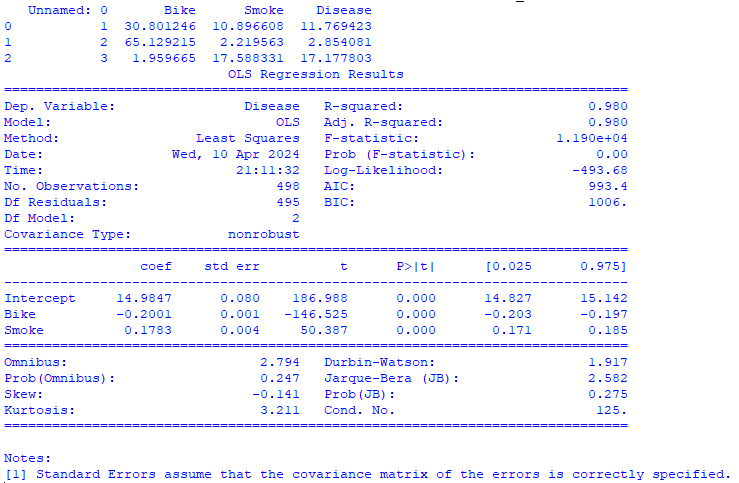
plt.legend(['Observations', 'Model Prediction'])

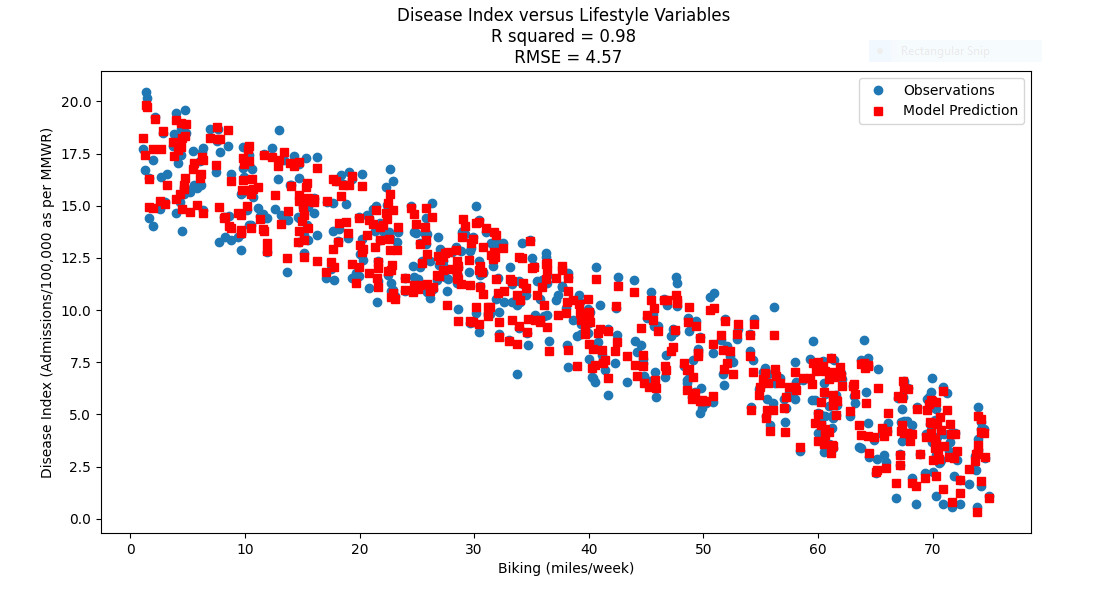
plt.title(titleline)

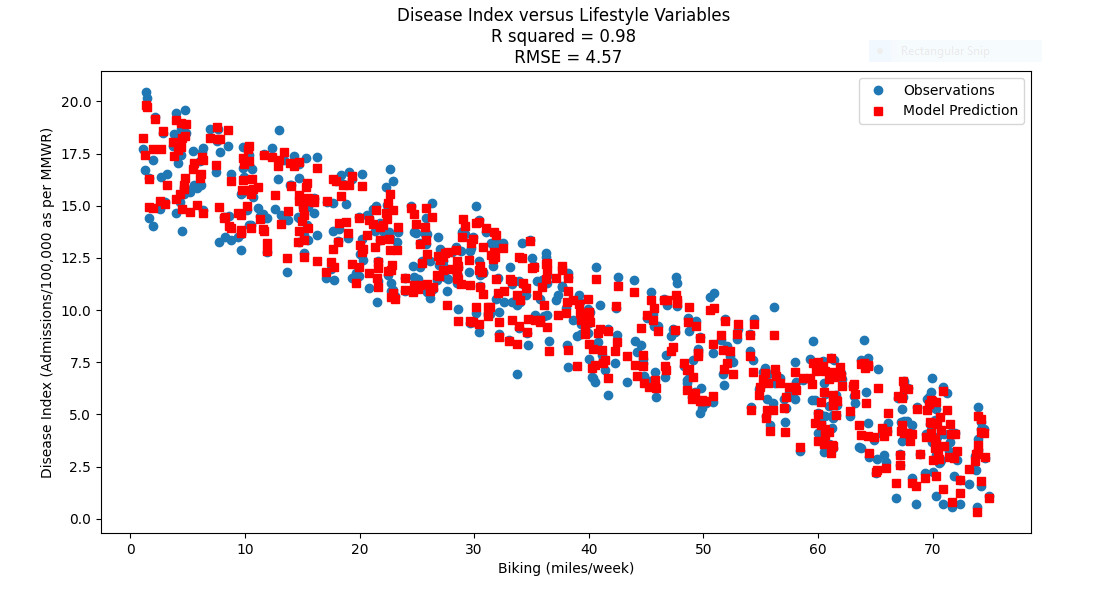
plt.show()

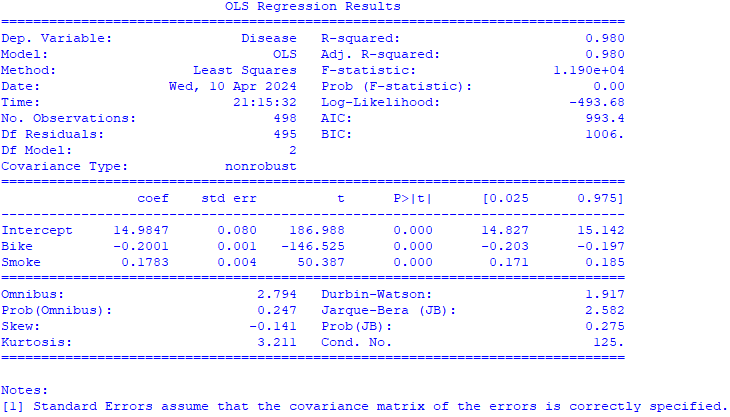
print(model.summary())

**OUTPUT:**

****







**RESULT:**

Thus, the program to demonstrate multiple variable linear regression was written and executed successfully.

**EXP NO: 4**

**DATE:**

**NAÏVE BAYES MODELS**

**AIM:**

To write a python program to demonstrate Naïve Bayes model using a given dataset.

**DESCRIPTION:**

The Naïve Bayes Classifier is a supervised machine learning algorithm, which is used for classification tasks like test classification. It is also a part of a family of generative learning algorithms, meaning, that it seeks to model the distribution of inputs of a given class or category.

**ALGORITHM:**

1. **Data Preprocessing:**

Clean the data set and split the data set in training and testing sets.

1. **Training:**

Estimate the prior probability of each class based on the training data.

1. **Prediction:**

For a new instance, estimate p(x) using law of total probability.

1. **Evaluation:**

Assess the performance of the classifier using metrics such as accuracy, prediction, recall and F1\_Score on the test set.

1. **Iterate:**

Refine the model by adjusting hyper rater meter or feature selection techniques based on evaluation results.

1. **Deployment:**

Use the trained model to make predictions on new, unseen data in production environments.

**PROGRAM:**

import math

import random

import csv

# The categorical class names are changed to numeric data

# e.g., yes and no encoded to 1 and 0

def encode\_class(mydata):

classes = []

for i in range(len(mydata)):

if mydata[i][-1] not in classes:

classes.append(mydata[i][-1])

for i in range(len(classes)):

for j in range(len(mydata)):

if mydata[j][-1] == classes[i]:

mydata[j][-1] = i

return mydata

# Splitting the data

def splitting(mydata, ratio):

train\_num = int(len(mydata) \* ratio)

train = []

# initially testset will have all the dataset

test = list(mydata)

while len(train) < train\_num:

# index generated randomly from range 0

# to length of testset

index = random.randrange(len(test))

# from testset, pop data rows and put it in train

train.append(test.pop(index))

return train, test

# Group the data rows under each class yes or

# no in dictionary eg: dict[yes] and dict[no]

def groupUnderClass(mydata):

group\_dict = {}

for i in range(len(mydata)):

if mydata[i][-1] not in group\_dict:

group\_dict[mydata[i][-1]] = []

group\_dict[mydata[i][-1]].append(mydata[i])

return group\_dict

# Calculating Mean

def mean(numbers):

return sum(numbers) / float(len(numbers))

# Calculating Standard Deviation

def std\_dev(numbers):

avg = mean(numbers)

variance = sum([pow(x - avg, 2) for x in numbers]) / float(len(numbers) - 1)

return math.sqrt(variance)

def MeanAndStdDev(mydata):

info = [(mean(attribute), std\_dev(attribute)) for attribute in zip(\*mydata)]

# Delete summaries of last class

del info[-1]

return info

# Find Mean and Standard Deviation under each class

def MeanAndStdDevForClass(mydata):

info = {}

group\_dict = groupUnderClass(mydata)

for classValue, instances in group\_dict.items():

info[classValue] = MeanAndStdDev(instances)

return info

# Calculate Gaussian Probability Density Function

def calculateGaussianProbability(x, mean, stdev):

expo = math.exp(-(math.pow(x - mean, 2) / (2 \* math.pow(stdev, 2))))

return (1 / (math.sqrt(2 \* math.pi) \* stdev)) \* expo

# Calculate Class Probabilities

def calculateClassProbabilities(info, test):

probabilities = {}

for classValue, classSummaries in info.items():

probabilities[classValue] = 1

for i in range(len(classSummaries)):

mean, std\_dev = classSummaries[i]

X = test[i]

probabilities[classValue] \*= calculateGaussianProbability(X, mean, std\_dev)

return probabilities

# Make prediction – highest probability is the prediction

def predict(info, test):

probabilities = calculateClassProbabilities(info, test)

bestLabel, bestProb = None, -1

for classValue, probability in probabilities.items():

if bestLabel is None or probability > bestProb:

bestProb = probability

bestLabel = classValue

return bestLabel

# Returns predictions for a set of examples

def getPredictions(info, test):

predictions = []

for i in range(len(test)):

result = predict(info, test[i])

predictions.append(result)

return predictions

# Accuracy score

def accuracy\_rate(test, predictions):

correct = 0

for i in range(len(test)):

if test[i][-1] == predictions[i]:

correct += 1

return (correct / float(len(test))) \* 100.0

# Driver code

# Add the data path in your system

filename = " " #In between the quotation Enter the path of the file

# Load the file and store it in mydata list

with open(filename, "r") as file:

reader = csv.reader(file)

headers = next(reader) # Skip the first row containing headers

mydata = list(reader)

mydata = encode\_class(mydata)

for i in range(len(mydata)):

mydata[i] = [float(x) for x in mydata[i]]

# Split ratio = 0.7

# 70% of data is training data and 30% is test data used for testing

ratio = 0.7

train\_data, test\_data = splitting(mydata, ratio)

print('Total number of data:', len(mydata))

print('Training data:', len(train\_data))

print('Test data:', len(test\_data))

# Prepare model

info = MeanAndStdDevForClass(train\_data)

# Test model

predictions = getPredictions(info, test\_data)

accuracy = accuracy\_rate(test\_data, predictions)

print("Accuracy of the model is:", accuracy)

**OUTPUT:**

Total number of data: 768

Training data: 537

Test data: 231

Accuracy of the model is: 72.2943722943723

**RESULT:**

Thus, the program to implement a Naïve Bates Model was written and executed successfully.

**EXP NO: 5**

**DATE:**

**BAYESIAN NETWORKS**

**AIM:**

To write a python program to demonstrate Bayesian network.

**DESCRIPTION:**

A Bayesian Network is a compact, flexible and interpretable representation of a joint probability distribution. It is also a useful tool in knowledge discovery as directed a cyclic graphics allow representing causal relationships between variables. Typically, a Bayesian network is learned from data.

**ALGORITHM:**

1. **Define the network structure:**

Specify the variables of interest and their relationships using a directed acyclic graph (DAG).

1. **Specify conditional probability distributions:**

For each node in the graph, specify the conditional probability distribution of that variable given its parents.

1. **Inference:**

Given observed evidence, infer the posterior distribution over the unobserved variables using methods like variable elimination, MCMC, or variational inference.

1. **Learning:**

Given data, learn the structure and/or parameters of the Bayesian network using techniques like maximum likelihood estimation, Bayesian estimation, or structure learning algorithms.

1. **Prediction and Decision-making:**

Use the learned Bayesian network for making predictions or decisions under uncertainty, considering observed evidence and utility functions.

1. **Validation and Evaluation:**

Evaluate the performance of the Bayesian network model using appropriate metrics and validate its effectiveness in solving the intended task.

**PROGRAM:**

import numpy as np

from pgmpy.models import BayesianNetwork

from pgmpy.factors.discrete import TabularCPD

# Define the model structure

model = BayesianNetwork([('Burglary', 'Alarm'), ('Earthquake', 'Alarm'), ('Alarm', 'JohnCalls'), ('Alarm', 'MaryCalls')])

# Define the conditional probability distributions

cpd\_burglary = TabularCPD(variable='Burglary', variable\_card=2, values=[[0.5], [0.5]])

cpd\_earthquake = TabularCPD(variable='Earthquake', variable\_card=2, values=[[0.998], [0.002]])

cpd\_alarm = TabularCPD(variable='Alarm', variable\_card=2, evidence=['Burglary', 'Earthquake'], evidence\_card=[2, 2],

values=[[0.999, 0.71, 0.06, 0.05], [0.001, 0.29, 0.94, 0.95]])

cpd\_john\_calls = TabularCPD(variable='JohnCalls', variable\_card=2, evidence=['Alarm'], evidence\_card=[2],

values=[[0.95, 0.1], [0.05, 0.9]])

cpd\_mary\_calls = TabularCPD(variable='MaryCalls', variable\_card=2, evidence=['Alarm'], evidence\_card=[2],

values=[[0.99, 0.3], [0.01, 0.7]])

# Add the conditional probability distributions to the model

model.add\_cpds(cpd\_burglary, cpd\_earthquake, cpd\_alarm, cpd\_john\_calls, cpd\_mary\_calls)

# Check if the model is valid

if model.check\_model():

print("Model is valid")

else:

print("Model is not valid")

print(model.local\_independencies('Burglary'))

print(model.edges())

print(model.nodes())

**OUTPUT:**

Model is valid

(Burglary ⟂ Earthquake)

[('Burglary', 'Alarm'), ('Alarm', 'JohnCalls'), ('Alarm', 'MaryCalls'), ('Earthquake', 'Alarm')]

['Burglary', 'Alarm', 'Earthquake', 'JohnCalls', 'MaryCalls']

**RESULT:**

Thus, the program to demonstrate Bayesian network was written and executed successfully.

**EXP NO: 6**

**DATE:**

**SVM MODELS**

**AIM:**

To write a python program to demonstrate support vector machine by finding a decision boundary.

**DESCRIPTION:**

A Support Vector Machine (SVM) is a type of Supervised Learning Algorithm used in Machine Learning to solve classification and regression tasks; SVMs are particularly good at solving binary classification problems, which requiring classifying the elements of a data set into two groups.

**ALGORITHM:**

1. **Data Preparation:**

Prepare the dataset with labelled samples for training and testing.

1. **Feature Selection/Extraction:**

Select relevant features or extract meaningful representations from the data.

1. **Model Training:**

Train the SVM model using the training data by choosing a kernel function.

1. **Model Evaluation:**

Evaluate the trained model's performance using validation data or cross-validation techniques.

Use metrics like accuracy, precision, recall, or F1-score to assess performance.

1. **Model Testing:**

Test the trained SVM model on unseen data to evaluate its generalization ability.

1. **Model Tuning (Optional):**

Fine-tune the model parameters based on performance evaluation results if necessary.

1. **Deployment:**

Deploy the trained SVM model for making predictions on new, unseen data.

**PROGRAM:**

# Basic packages

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

# Sklearn modules & classes

from sklearn.linear\_model import Perceptron, LogisticRegression

from sklearn.svm import SVC

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

from sklearn.preprocessing import StandardScaler

from sklearn import datasets

from sklearn import metrics

# Load the data set; In this example, the breast cancer dataset is loaded.

bc = datasets.load\_breast\_cancer()

X = bc.data

Y = bc.target

# Create training and test split

X\_train, X\_test, Y\_train, Y\_test = train\_test\_split(X, Y, test\_size=0.3, random\_state=1, stratify=Y)

# feature scaling

sc = StandardScaler()

sc.fit(X\_train)

X\_train\_std = sc.transform(X\_train)

X\_test\_std = sc.transform(X\_test)

#Instantiate the Support Vector Classifier (SVC)

svc = SVC (C=1.0, random\_state=1, kernel='linear')

# Fit the model

svc.fit(X\_train\_std, Y\_train)

#make the prediction

Y\_predict = svc.predict(X\_test\_std)

# Measure the performance

print ("Accuracy score %.3f" %metrics.accuracy\_score(Y\_test, Y\_predict))

**OUTPUT:**

Accuracy score 0.953

**RESULT:**

Thus, the program to demonstrate support vector machine by finding a decision boundary was written and executed successfully.

**EXP NO: 7(A)**

**DATE:**

**DECISION TREE**

**AIM:**

To write a python program to demonstrate decision tree to classify a given dataset.

**DESCRIPTION:**

A Decision Tree is a tree like nodes that acts as a decision support tool, visually displaying decision and their potential outcomes, consequences and costs. From these, the “branches” can easily be evaluated and compared in order to select the courses of action.

**ALGORITHM:**

1. Begin the tree with Root node, says S which contains the complete data set.
2. Find the best attribute in the data set using Attribute Selection Measure (AIM).
3. Divide the S into subsets that contains possible values for the best attributes.
4. Generate the Decision Tree node, which contains the best attribute.
5. Recursively make new Decision Tree using the subsets of the dataset created in (3) continue this process until a stage is reached where the further classification of nodes is not possible and call the final ndoes as Leaf node.

**PROGRAM:**

# importing libraries

import numpy as np

import matplotlib.pyplot as plt

import pandas as pd

from sklearn.tree import DecisionTreeClassifier

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

from sklearn.preprocessing import StandardScaler

from matplotlib.colors import ListedColormap

# importing datasets

data\_set = pd.read\_csv('user\_data.csv')

# Extracting Independent and dependent Variable

X = data\_set.iloc[:, [2, 3]].values

Y = data\_set.iloc[:, 4].values

# Splitting the dataset into training and test set.

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, Y, test\_size=0.25, random\_state=0)

# feature Scaling

st\_x = StandardScaler()

X\_train = st\_x.fit\_transform(X\_train)

X\_test = st\_x.transform(X\_test)

classifier = DecisionTreeClassifier(criterion='entropy', random\_state=0)

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

# Predicting the test set result

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

from sklearn.metrics import confusion\_matrix

cm = confusion\_matrix(y\_test, y\_pred)

# Visualizing the training set result

X\_set, y\_set = X\_train, y\_train

X1, X2 = np.meshgrid(np.arange(start=X\_set[:, 0].min() - 1, stop=X\_set[:, 0].max() + 1, step=0.01),

np.arange(start=X\_set[:, 1].min() - 1, stop=X\_set[:, 1].max() + 1, step=0.01))

plt.contourf(X1, X2, classifier.predict(np.array([X1.ravel(), X2.ravel()]).T).reshape(X1.shape),

alpha=0.75, cmap=ListedColormap(['pink', 'yellow']))

plt.xlim(X1.min(), X1.max())

plt.ylim(X2.min(), X2.max())

for i, j in enumerate(np.unique(y\_set)):

plt.scatter(X\_set[y\_set == j, 0], X\_set[y\_set == j, 1],color=ListedColormap(('black', 'red'))(i), label=j)

plt.title('Decision Tree Algorithm (Training set)')

plt.xlabel('Age')

plt.ylabel('Estimated Salary')

plt.legend()

plt.show()

# Visualizing the test set result

X\_set, y\_set = X\_test, y\_test

X1, X2 = np.meshgrid(np.arange(start=X\_set[:, 0].min() - 1, stop=X\_set[:, 0].max() + 1, step=0.01),np.arange(start=X\_set[:, 1].min() - 1, stop=X\_set[:, 1].max() + 1, step=0.01))

plt.contourf(X1, X2, classifier.predict(np.array([X1.ravel(), X2.ravel()]).T).reshape(X1.shape),

alpha=0.75, cmap=ListedColormap(('pink', 'yellow')))

plt.xlim(X1.min(), X1.max())

plt.ylim(X2.min(), X2.max())

for i, j in enumerate(np.unique(y\_set)):

plt.scatter(X\_set[y\_set == j, 0], X\_set[y\_set == j, 1],

color=ListedColormap(('green', 'blue'))(i), label=j)

plt.title('Decision Tree Algorithm (Test set)')

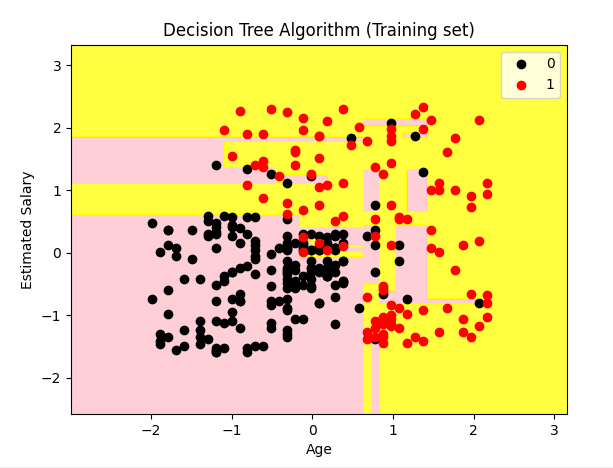
plt.xlabel('Age')

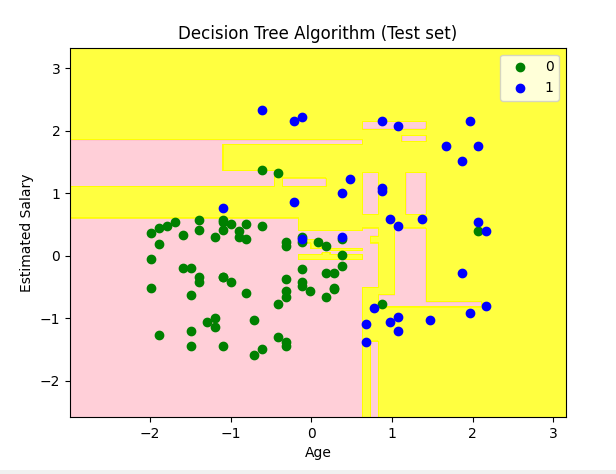
plt.ylabel('Estimated Salary')

plt.legend()

plt.show()

**OUTPUT:**

****

****

**RESULT:**

Thus, the program to demonstrate decision tree to classify a given dataset was written and executed successfully.

**EXP NO: 7(B)**

**DATE:**

**RANDOM FOREST**

**AIM:**

To write a python program to demonstrate a random forest.

**DESCRIPRION:**

Random Forests is a commonly used Machine Learning algorithm, that combines the output of multiple decision trees to reach a single result. Its ease of use and flexibility have fuelled its adoption, as it handles both classification and regression problems.

**ALGORITHM:**

1. Select random samples from given data or training set.
2. This algorithm will construct a Decision Tree for every training set.
3. Voting will take place by averaging the Decision Tree.
4. Finally, select the most voted prediction result as the final prediction result.

**PROGRAM:**

# importing libraries

import numpy as np

import matplotlib.pyplot as plt

import pandas as pd

from matplotlib.colors import ListedColormap

from sklearn.preprocessing import StandardScaler

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

from sklearn.ensemble import RandomForestClassifier

from sklearn.metrics import confusion\_matrix

# importing datasets

data\_set = pd.read\_csv('User\_data.csv')

# Extracting Independent and dependent Variable

X = data\_set.iloc[:, [2, 3]].values

Y = data\_set.iloc[:, 4].values

# Splitting the dataset into training and test set.

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, Y, test\_size=0.25, random\_state=0)

st\_x = StandardScaler()

X\_train = st\_x.fit\_transform(X\_train)

X\_test = st\_x.transform(X\_test)

# Fitting Decision Tree classifier to the training set

classifier = RandomForestClassifier(n\_estimators=10, criterion="entropy")

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

# Predicting the test set result

Y\_pred = classifier.predict(X\_test)

cm = confusion\_matrix(y\_test, Y\_pred)

# Visualizing the training set results

X\_set, y\_set = X\_train, y\_train

X1, X2 = np.meshgrid(np.arange(start=X\_set[:, 0].min() - 1, stop=X\_set[:, 0].max() + 1, step=0.01),

np.arange(start=X\_set[:, 1].min() - 1, stop=X\_set[:, 1].max() + 1, step=0.01))

plt.contourf(X1, X2, classifier.predict(np.array([X1.ravel(), X2.ravel()]).T).reshape(X1.shape),

alpha=0.75, cmap=ListedColormap(('pink', 'yellow')))

plt.xlim(X1.min(), X1.max())

plt.ylim(X2.min(), X2.max())

for i, j in enumerate(np.unique(y\_set)):

plt.scatter(X\_set[y\_set == j, 0], X\_set[y\_set == j, 1],

color=ListedColormap(('purple', 'green'))(i), label=j)

plt.title('Random Forest Algorithm (Training set)')

plt.xlabel('Age')

plt.ylabel('Estimated Salary')

plt.legend()

plt.show()

# Visualizing the test set results

X\_set, y\_set = X\_test, y\_test

X1, X2 = np.meshgrid(np.arange(start=X\_set[:, 0].min() - 1, stop=X\_set[:, 0].max() + 1, step=0.01),

np.arange(start=X\_set[:, 1].min() - 1, stop=X\_set[:, 1].max() + 1, step=0.01))

plt.contourf(X1, X2, classifier.predict(np.array([X1.ravel(), X2.ravel()]).T).reshape(X1.shape),

alpha=0.75, cmap=ListedColormap(('yellow', 'pink')))

plt.xlim(X1.min(), X1.max())

plt.ylim(X2.min(), X2.max())

for i, j in enumerate(np.unique(y\_set)):

plt.scatter(X\_set[y\_set == j, 0], X\_set[y\_set == j, 1],

color=ListedColormap(('blue', 'black'))(i), label=j)

plt.title('Random Forest Algorithm(Test set)')

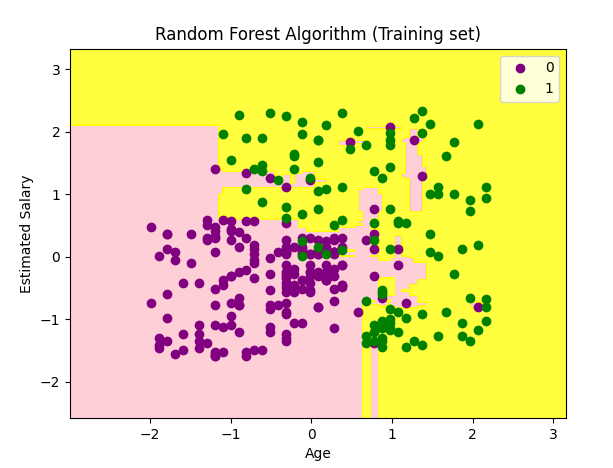
plt.xlabel('Age')

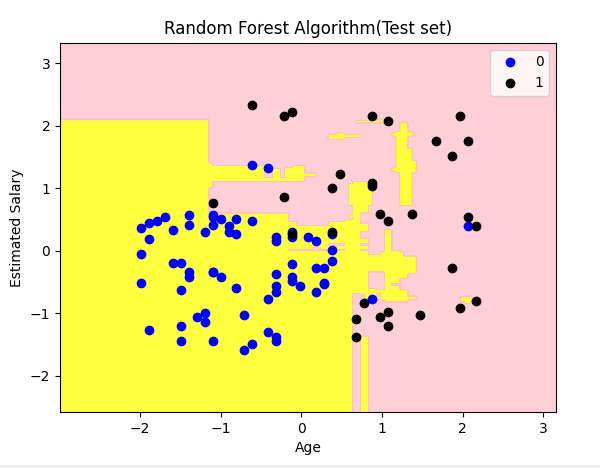
plt.ylabel('Estimated Salary')

plt.legend()

plt.show()

**OUTPUT:**





**RESULT:**

Thus, the program to demonstrate a random forest was written and executed successfully.

**EXP NO: 8**

**DATE:**

**ENSEMBLING TECHNIQUES**

**BAGGING USING DECISION TREE CLASSIFICATION**

**AIM:**

To write a python program to implement bagging using decision tree classification and evaluate its performance.

**DESCRIPTION:**

Bagging also known as Bootstrap Aggregation, is the Ensemble Learning method that is commonly used to reduce variance within a noisy data set. In Bagging, a random sample of data in a training selected with replacement – meaning that is the individual data points can be chosen more than one.

**ALGORITHM:**

1. **Initialization:**

Set the number of bootstrap samples to create (usually equal to the number of data points).

Choose the base learning algorithm (e.g., decision trees).

1. **Generate Bootstrap Samples:**

For each bootstrap sample:

Randomly select data points (with replacement) from the original dataset to create a new dataset of the same size.

1. **Train Base Models:**

Train a base model (e.g., decision tree) on each bootstrap sample.

1. **Aggregate Predictions:**

For regression: Take the average of predictions from all base models.

For classification: Take a majority vote (mode) of predictions from all base models.

1. **Final Prediction:**

Use the aggregated predictions as the final prediction.

**PROGRAM:**

from sklearn import datasets

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

from sklearn.metrics import accuracy\_score

from sklearn.tree import DecisionTreeClassifier

import matplotlib.pyplot as plt

from sklearn.ensemble import BaggingClassifier

from sklearn.tree import plot\_tree

# Load the wine dataset

data = datasets.load\_wine(as\_frame=True)

X = data.data

y = data.target

# Split the dataset into training and testing sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.25, random\_state=22)

# Train a decision tree classifier

dtree = DecisionTreeClassifier(random\_state=22)

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

# Predict on the test set

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

# Print accuracy scores

print("Train data accuracy:", accuracy\_score(y\_true=y\_train, y\_pred=dtree.predict(X\_train)))

print("Test data accuracy:", accuracy\_score(y\_true=y\_test, y\_pred=y\_pred))

# Performance evaluation of bagging

estimator\_range = [2, 4, 6, 8, 10, 12, 14, 16]

scores = []

for n\_estimators in estimator\_range:

# Create bagging classifier

clf = BaggingClassifier(n\_estimators=n\_estimators, random\_state=22)

# Fit the model

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

# Append the score to the scores list

scores.append(accuracy\_score(y\_true=y\_test, y\_pred=clf.predict(X\_test)))

# Generate the plot of scores against number of estimators

plt.figure(figsize=(9, 6))

plt.plot(estimator\_range, scores)

# Adjust labels and font

plt.xlabel("n\_estimators", fontsize=18)

plt.ylabel("score", fontsize=18)

plt.tick\_params(labelsize=16)

# Visualize plot

plt.show()

# Generate Decision Trees from Bagging Classifier

clf = BaggingClassifier(n\_estimators=12, oob\_score=True, random\_state=22)

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

plt.figure(figsize=(30, 20))

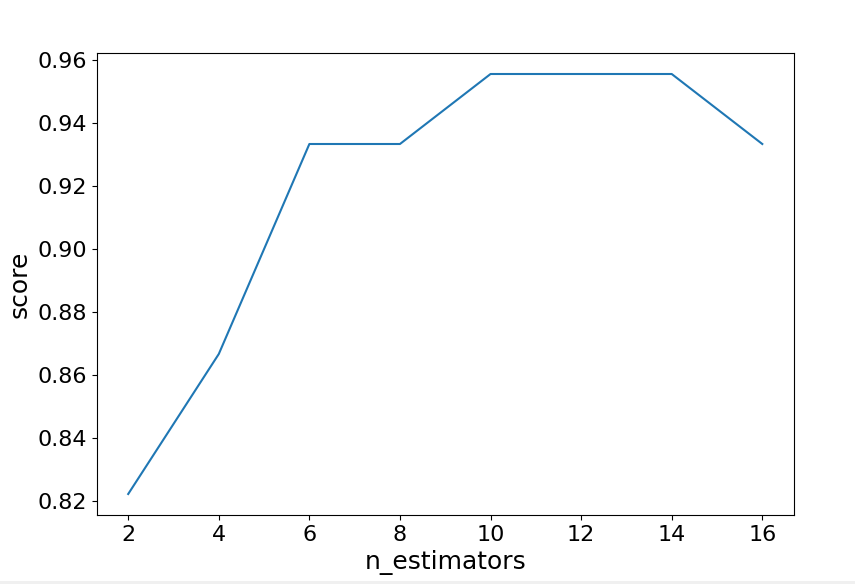
plot\_tree(clf.estimators\_[0], feature\_names=data.feature\_names)

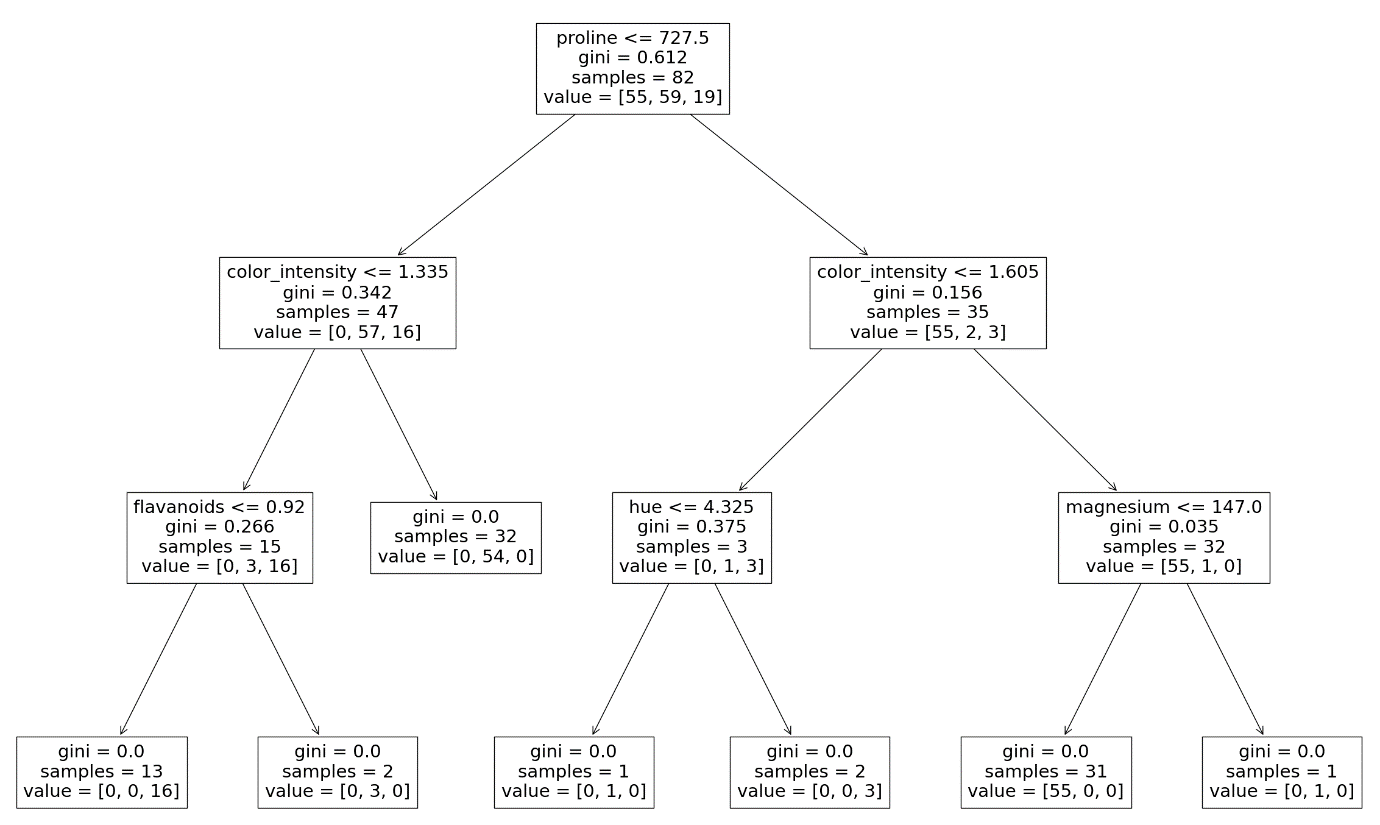
plt.show()

**OUTPUT:**

Train data accuracy: 1.0

Test data accuracy: 0.8222222222222222





**RESULT:**

Thus, the program to implement bagging using decision tree classification and evaluate its performance was written and executed successfully.

**EXP NO: 9**

**DATE:**

**K MEANS CLUSTERING**

**AIM:**

To write a python program to demonstrate k means clustering.

**DESCRIPTION:**

K Means Clustering is an iterative process of assigning each data point to groups and slowly data points gets clustered based o similar features. The objective is to minimize the sum of distances between the data points and the cluster centroid, to identify the correct group each data point should belong to.

**ALGORITHM:** for K-means clustering:

1. **Initialization:**

Choose the number of clusters (K).

Initialize K cluster centroids randomly or using a heuristic (e.g., randomly select K data points as centroids).

1. **Assign Data Points to Clusters:**

For each data point:

Calculate the distance to each centroid.

Assign the data point to the cluster with the nearest centroid.

1. **Update Cluster Centroids:**

For each cluster:

Calculate the mean of all data points assigned to that cluster.

Set the centroid of the cluster to the mean.

1. **Repeat Steps 2 and 3:**

Iterate Steps 2 and 3 until convergence:

Convergence occurs when the centroids no longer change significantly or after a fixed number of iterations.

1. **Final Step:**

Once convergence is reached, the clusters are formed, and each data point is assigned to one of the K clusters.

**PROGRAM:**

import numpy as np

import matplotlib.pyplot as plt

from sklearn.cluster import KMeans

np.random.seed(0)

X = np.random.randn(200, 2) + np.array([2, 2])

X = np.vstack((X, np.random.randn(200, 2) + np.array([-2, -2])))

X = np.vstack((X, np.random.randn(200, 2) + np.array([2, -2])))

X = np.vstack((X, np.random.randn(200, 2) + np.array([-2, 2])))

kmeans = KMeans(n\_clusters=4)

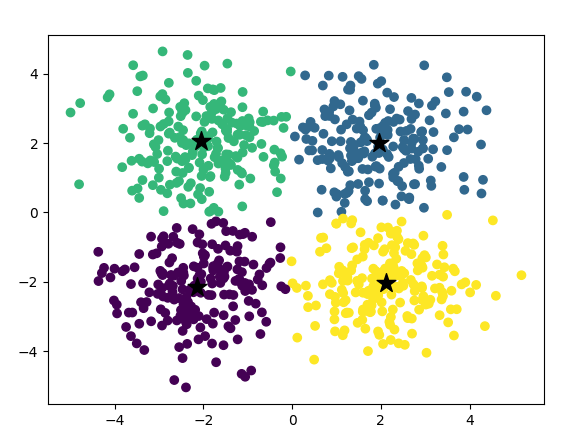
kmeans.fit(X)

plt.scatter(X[:,0], X[:,1], c=kmeans.labels\_, cmap='viridis')

plt.scatter(kmeans.cluster\_centers\_[:, 0], kmeans.cluster\_centers\_[:, 1], marker='\*', s=200, color='black')

plt.show()

**OUTPUT:**



**RESULT:**

Thus, the program to demonstrate k means clustering was written and executed successfully.

**EXP NO: 10**

**DATE:**

**EM FOR BAYESIAN NETWORKS**

**AIM:**

To write a python program to demonstrate and building EM for Bayesian network.

**DESCRIPTION:**

Bayesian Networks (BNs) are often used in these domains because of their geographical and causal interpretations. BN parameter learning from incomplete data is usually implemented with the EM (Expectation Maximization Algorithm), which computes the relevant sufficient statistics (“soft EM”) using belief propagation.

**ALGORITHM:**

1. **Initialization:** Start with initial parameter.
2. **E-Step:** Estimate the posterior distribution of latent variables given in observed data.
3. **M-Step:** Update parameters to maximize expected log-likelihood using E-Step results.
4. **Convergence Check:** Verify if parameters or log-likelihood change significantly.
5. **Repeat:** Iterate E-Step and M-Step until convergence.
6. **Output:** Return learned parameters.

**PROGRAM:**

import numpy as np

import pandas as pd

from pgmpy.estimators import MaximumLikelihoodEstimator

from pgmpy.models import BayesianNetwork # Adjusted import statement

from pgmpy.inference import VariableElimination

heartDise = pd.read\_csv(r" ") #In between quotation give the path of the file

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

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

print(heartDise.head())

print('\nAttributes and datatypes')

print(heartDise.dtypes)

# Define the edges of the Bayesian Network

edges = [('age', 'heartdisease'), ('gender', 'heartdisease'), ('exang', 'heartdisease'),

('cp', 'heartdisease'), ('heartdisease', 'restecg'), ('heartdisease', 'chol')]

# Create and fit the Bayesian Network

model = BayesianNetwork(edges)

model.fit(heartDise, estimator=MaximumLikelihoodEstimator)

print('\nInferencing with Bayesian Network:')

HeartDiseasetest\_infer = VariableElimination(model)

print('\n1. Probability of HeartDisease given evidence= restecg')

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

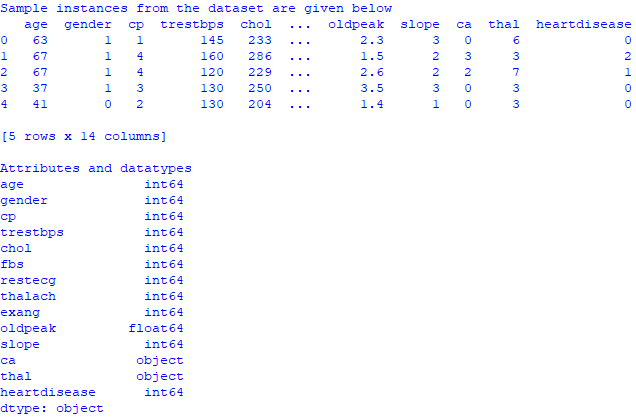
print(q1)

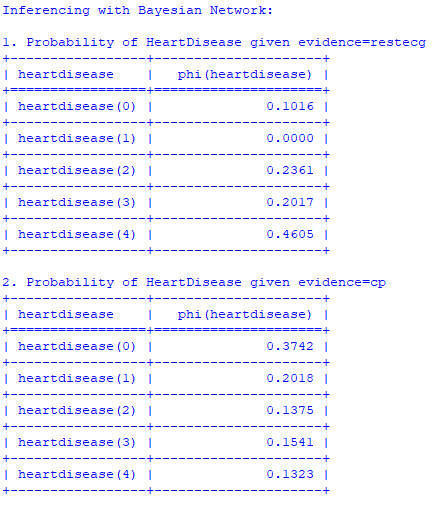
print('\n2. Probability of HeartDisease given evidence= cp')

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

print(q2)

**OUTPUT:**





**RESULT:**

Thus, the program to demonstrate EM for Bayesian network was written and executed successfully.

**EXP NO: 11**

**DATE:**

**SIMPLE NN MODEL**

**AIM:**

To write a python program to demonstrate Neural Network Back Propagation.

**DESCRIPTION:**

Back propagation in Neural Network is a short form of “Backward Propagation of errors”. It is a standard method of training Artificial Neural Networks. This method helps to calculate the gradient of a loss of a function with respect to all the weights in the network.

**ALGORITHM:**

1. **Initialize weights:** randomly initialize weights and biases.
2. **Forward Pass:** Compute output using current weights and biases.
3. **Compute Loss:** Calculate error between predicted and actual outputs.
4. **Backward Pass:** Compute gradients using chain rule, update weights to minimize loss.
5. **Repeat:** Iterate steps 2 to 4 for a fixed number of epochs.
6. **Convergence:** Stop when loss or converges or after a certain number of epochs.
7. **Evaluation:** Asses the model performance on validation set, adjust hyper parameters as needed.

**PROGRAM:**

import numpy as np

# Define the sigmoid function

def sigmoid(x):

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

# Define the derivative of the sigmoid function

def derivatives\_sigmoid(x):

return x \* (1 - x)

# Input data

X = np.array(([2, 9], [1, 5], [3, 6]), dtype=int)

# Output data

y = np.array(([92], [86], [89]), dtype=float)

y = y/100

# Variable initialization

epoch = 10000 # Setting training iterations

lr = 0.1 # Setting learning rate

inputlayer\_neurons = 2 # number of features in data set

hiddenlayer\_neurons = 3 # number of hidden layers neurons

output\_neurons = 1 # number of neurons at output layer

# Weight and bias initialization

wh = np.random.uniform(size=(inputlayer\_neurons, hiddenlayer\_neurons))

bh = np.random.uniform(size=(1, hiddenlayer\_neurons))

wout = np.random.uniform(size=(hiddenlayer\_neurons, output\_neurons))

bout = np.random.uniform(size=(1, output\_neurons))

# Training

for i in range(epoch):

# Forward Propagation

hinp1 = np.dot(X, wh)

hinp = hinp1 + bh

hlayer\_act = sigmoid(hinp)

outinp1 = np.dot(hlayer\_act, wout)

outinp = outinp1 + bout

output = sigmoid(outinp)

# Backpropagation

EO = y - output

outgrad = derivatives\_sigmoid(output)

d\_output = EO \* outgrad

EH = d\_output.dot(wout.T)

hiddengrad = derivatives\_sigmoid(hlayer\_act)

d\_hiddenlayer = EH \* hiddengrad

# Update weights and biases

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

bh += np.sum(d\_hiddenlayer, axis=0, keepdims=True) \* lr

# Print results

print("Input:\n" + str(X))

print("Actual Output:\n" + str(y))

print("Predicted Output:\n", output)

**OUTPUT:**

Input:

[[2 9]

[1 5]

[3 6]]

Actual Output:

[[0.92]

[0.86]

[0.89]]

Predicted Output:

[[0.89419442]

[0.88559337]

[0.88896262]]

**RESULT:**

Thus, the program to demonstrate Neural Network Back Propagation was written and executed successfully.

**EXP NO: 12**

**DATE:**

**DEEP NN MODEL**

**AIM:**

To write a python program to demonstrate Deep Neural Network model using keras.

**DESCRIPTION:**

Deep Learning models are computer files that data scientists have trained to perform tasks using an algorithm or a predefined set of steps. Businesses use deep learning models to analyze data and make predictions in various applications.

**ALGORITHM:**

1. **Data Preparation:** Clean, pre-process and split the data
2. **Model Architecture Design:** Design the neural network structure.
3. **Initialization:** Initialize model parameters.
4. **Forward Propagation:** Pass data through the network for prediction.
5. **Loss Computation:** Calculate error between predicted and actual outputs.
6. **Back Propagation:** Compute gradients to update parameters.
7. **Parameter Update:** Update parameters to minimize loss.
8. **Validation:** Assess model performance on validation set.
9. **Testing:** Evaluate model performance on a separate test set.
10. **Deployment:** Integrate and deploy the trained model for real world use.

**PROGRAM:**

import numpy as np

import tensorflow as tf

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

from sklearn.preprocessing import StandardScaler

import warnings

warnings.filterwarnings("ignore", category=DeprecationWarning)

# Load the dataset

# Skip the header row

dataset = np.loadtxt(r" ", delimiter=',', skiprows=1) #Inside double quotation give the path of the file

# Split the dataset into features and labels

X = dataset[:, 0:8]

y = dataset[:, 8]

# Normalize the features

scaler = StandardScaler()

X\_scaled = scaler.fit\_transform(X)

accuracies = []

for i in range(5): # Perform 5 executions

# Split the dataset into training and testing sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X\_scaled, y, test\_size=0.2, random\_state=42)

# Define the neural network model

model = tf.keras.Sequential([

tf.keras.layers.Input(shape=(8,)), # Input layer with shape (8,)

tf.keras.layers.Dense(12, activation='relu'),

tf.keras.layers.Dense(8, activation='relu'),

tf.keras.layers.Dense(1, activation='sigmoid')

])

# Compile the model

model.compile(loss='binary\_crossentropy', optimizer='adam', metrics=['accuracy'])

# Train the model

model.fit(X\_train, y\_train, epochs=150, batch\_size=10, verbose=0)

# Evaluate the model

\_, accuracy = model.evaluate(X\_test, y\_test, verbose=0)

accuracies.append(accuracy \* 100)

print("Accuracy:", accuracy \* 100)

# Make predictions on some sample inputs

sample\_inputs = np.array([

[6.0, 148.0, 72.0, 35.0, 0.0, 33.6, 0.627, 50.0],

[1.0, 85.0, 66.0, 29.0, 0.0, 26.6, 0.351, 31.0],

[8.0, 183.0, 64.0, 0.0, 0.0, 23.3, 0.672, 32.0],

[1.0, 89.0, 66.0, 23.0, 94.0, 28.1, 0.167, 21.0],

[0.0, 137.0, 40.0, 35.0, 168.0, 43.1, 2.288, 33.0]

])

# Normalize the sample inputs

sample\_inputs\_scaled = scaler.transform(sample\_inputs)

# Predict and print results

predictions = model.predict(sample\_inputs\_scaled)

for i in range(len(predictions)):

print(f"{sample\_inputs[i]} => {int(round(predictions[i][0]))} (expected {int(y[i])})")

**OUTPUT:**

Accuracy: 74.02597665786743

Accuracy: 74.02597665786743

Accuracy: 75.97402334213257

Accuracy: 72.07792401313782

Accuracy: 74.67532753944397

[ 6. 148. 72. 35. 0. 33.6 0.627 50. ] => 1 (expected 1)

[ 1. 85. 66. 29. 0. 26.6 0.351 31. ] => 0 (expected 0)

[ 8. 183. 64. 0. 0. 23.3 0.672 32. ] => 1 (expected 1)

[ 1. 89. 66. 23. 94. 28.1 0.167 21. ] => 0 (expected 0)

[ 0. 137. 40. 35. 168. 43.1 2.288 33. ] => 1 (expected 1)

**RESULT:**

Thus, the program to demonstrate Deep Neural Network using Keras was written and executed successfully.