PRATHYUSHA ENGINEERING COLLEGE

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

LAB MANUAL

for

CS3491-ARTIFICIAL INTELLIGENCE AND MACHINE LEARNING LABORATORY

(Regulation 2021, IV Semester)

(Even Semester)

ACADEMIC YEAR: 2023 – 2024

PREPARED BY

MEENA,

Assistant Professor / CSE

COURSE OBJECTIVES:

1. Study about uninformed and Heuristic search techniques.
2. Learn techniques for reasoning under uncertainty
3. Introduce Machine Learning and supervised learning algorithms
4. Study about ensembling and unsupervised learning algorithms
5. Learn the basics of deep learning using neural networks

EXPERIMENTS LIST

1. Implementation of Uninformed search algorithms (BFS, DFS)
2. Implementation of Informed search algorithms (A*, memory-bounded A*)
3. Implement naïve Bayes models
4. Implement Bayesian Networks
5. Build Regression models
6. Build decision trees and random forests
7. Build SVM models
8. Implement ensembling techniques
9. Implement clustering algorithms
10. Implement EM for Bayesian networks
11. Build simple NN models
12. Build deep learning NN models

COURSE OUTCOMES:

On completion of the course, students will be able to:

CO1: Use appropriate search algorithms for problem solving

CO2: Apply reasoning under uncertainty

CO3: Build supervised learning models

CO4: Build ensembling and unsupervised modelsCO5:

Build deep learning neural network models

TEXT BOOKS:

1. Stuart Russell and Peter Norvig, "Artificial Intelligence – A Modern Approach", Fourth Edition, Pearson Education, 2021.

2. Ethem Alpaydin, "Introduction to Machine Learning", MIT Press, Fourth Edition, 2020.

REFERENCES:

1. Dan W. Patterson, "Introduction to Artificial Intelligence and Expert Systems", Pearson Education,2007
2. Kevin Night, Elaine Rich, and Nair B., "Artificial Intelligence", McGraw Hill, 2008
3. Patrick H. Winston, "Artificial Intelligence", Third Edition, Pearson Education, 2006
4. Deepak Khemani, "Artificial Intelligence", Tata McGraw Hill Education, 2013
(http://nptel.ac.in/)
5. Christopher M. Bishop, "Pattern Recognition and Machine Learning", Springer, 2006.
6. Tom Mitchell, "Machine Learning", McGraw Hill, 3rd Edition,1997.
7. Charu C. Aggarwal, "Data Classification Algorithms and Applications", CRC Press, 2014
8. Mehryar Mohri, Afshin Rostamizadeh, Ameet Talwalkar, "Foundations of Machine Learning", MIT Press, 2012.
9. Ian Goodfellow, Yoshua Bengio, Aaron Courville, "Deep Learning", MIT Press

INDEX

1. Implementation of Uninformed search algorithms (BFS, DFS)

Aim:

To implement uninformed search algorithms such as BFS and DFS.

Algorithm:

Step 1:= Initialize an empty list called 'visited' to keep track of the nodes visited during the traversal.
Step 2:= Initialize an empty queue called 'queue' to keep track of the nodes to be traversed in the future.
Step 3:= Add the starting node to the 'visited' list and the 'queue'.
Step 4:= While the 'queue' is not empty, do the following:
  a. Dequeue the first node from the 'queue' and store it in a variable called 'current'.
  b. Print 'current'.
  c. For each of the neighbours of 'current' that have not been visited yet, do the following:
    i. Mark the neighbour as visited and add it to the 'queue'.
Step 5:= When all the nodes reachable from the starting node have been visited, terminate the algorithm.

Breadth First Search :

Program :

```
graph = {
  '5' : ['3','7'],
  '3' : ['2', '4'],
  '7' : ['8'],
  '2' : [],
  '4' : ['8'],
  '8' : []
}
visited = []
queue = []
def bfs(visited, graph, node):
  visited.append(node)
  queue.append(node)
  while queue:
    m = queue.pop(0)
    print (m, end = " ")
    for neighbour in graph[m]:
      if neighbour not in visited:
        visited.append(neighbour)
        queue.append(neighbour)

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

Output:

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

Depth first Search:

Algorithm:
Step 1:= Initialize an empty set called 'visited' to keep track of the nodes visited during the traversal.
Step 2:= Define a DFS function that takes the current node, the graph, and the 'visited' set as input.
Step 3:= If the current node is not in the 'visited' set, do the following:
    a. Print the current node.
    b. Add the current node to the 'visited' set.
    c. For each of the neighbours of the current node, call the DFS function recursively with the neighbour as the current node.
Step 4:= When all the nodes reachable from the starting node have been visited, terminate the algorithm.

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

visited = set()
def dfs(visited, graph, node):
    if node not in visited:
        print (node)
        visited.add(node)
        for neighbour in graph[node]:
            dfs(visited, graph, neighbour)

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 uninformed search algorithms such as BFS and DFS have been executed successfully and the output got verified

2. Implementation of Informed search algorithm (A*)

Aim:

To implement the informed search algorithm A*.

Algorithm:

1. Initialize the distances dictionary with float('inf') for all vertices in the graph except for the start vertex which is set to 0.
2. Initialize the parent dictionary with None for all vertices in the graph.
3. Initialize an empty set for visited vertices.
4. Initialize a priority queue (pq) with a tuple containing the sum of the heuristic value and the distance from start to the current vertex, the distance from start to the current vertex, and the current vertex.
5. While pq is not empty, do the following:
   a. Dequeue the vertex with the smallest f-distance (sum of the heuristic value and the distance from start to the current vertex).
   b. If the current vertex is the destination vertex, return distances and parent.
   c. If the current vertex has not been visited, add it to the visited set.
   d. For each neighbor of the current vertex, do the following:
      i. Calculate the distance from start to the neighbor (g) as the sum of the distance from start to the current vertex and the edge weight between the current vertex and the neighbor.
   ii. Calculate the f-distance (f = g + h) for the neighbor.
   iii. If the f-distance for the neighbor is less than its current distance in the distances dictionary, update the distances dictionary with the new distance and the parent dictionary with the current vertex as the parent of the neighbor.
   iv. Enqueue the neighbor with its f-distance, distance from start to neighbor, and the neighbor itself into the priority queue.
6. Return distances and parent.

Program :

```
import heapq

def a_star(graph, start, dest, heuristic):
  distances = {vertex: float('inf') for vertex in graph}     distances[start] = 0

  parent = {vertex: None for vertex in graph}
  visited = set()

  pq = [( 0 + heuristic[start], 0 ,  start)] # E space

  while pq:
    curr_f, curr_dist, curr_vert = heapq.heappop(pq)


    if curr_vert not in visited:
      visited.add(curr_vert)

      for nbor, weight in graph[curr_vert].items():
        distance = curr_dist + weight  # distance from start (g)
        f_distance = distance + heuristic[nbor] # f = g + h
```

```
            # Only process new vert if it's f_distance is lower
            if f_distance < distances[nbor]:
                distances[nbor] = f_distance
                parent[nbor] = curr_vert


                if nbor == dest:
                    # we found a path based on heuristic
                    return distances, parent

                heapq.heappush(pq, (f_distance, distance, nbor)) #logE time

    return distances, parent
def generate_path_from_parents(parent, start, dest):
    path = []
    curr = dest
    while curr:
        path.append(curr)
        curr = parent[curr]

    return '->'.join(path[::-1])



graph = {
    'A': {'B':5, 'C':5},
    'B': {'A':5, 'C':4, 'D':3  },
    'C': {'A':5, 'B':4, 'D':7, 'E':7, 'H':8},
    'D': {'B':3, 'C':7, 'H':11, 'K':16, 'L':13, 'M':14},
    'E': {'C':7, 'F':4, 'H':5},
    'F': {'E':4, 'G':9},
    'G': {'F':9, 'N':12},
    'H': {'E':5, 'C':8, 'D':11, 'I':3 },
    'I': {'H':3, 'J':4},
    'J': {'I':4, 'N':3},
    'K': {'D':16, 'L':5, 'P':4, 'N':7},
    'L': {'D':13, 'M':9, 'O':4, 'K':5},
    'M': {'D':14, 'L':9, 'O':5},
    'N': {'G':12, 'J':3, 'P':7},
    'O': {'M':5, 'L':4},
    'P': {'K':4, 'J':8, 'N':7},
}


heuristic = {
    'A': 16,
    'B': 17,
    'C': 13,
    'D': 16,
    'E': 16,
    'F': 20,
    'G': 17,
    'H': 11,
    'I': 10,
```

```
    'J': 8,
    'K': 4,
    'L': 7,
    'M': 10,
    'N': 7,
    'O': 5,
    'P': 0
}

start = 'A'
dest= 'P'
distances,parent = a_star(graph, start, dest, heuristic)
print('distances => ', distances)
print('parent => ', parent)
print('optimal path => ', generate_path_from_parents(parent,start,dest))
```

Output:

```
distances =>  {'A': 0, 'B': 22, 'C': 18, 'D': 24, 'E': 28, 'F': 36, 'G': inf, 'H': 24, 'I': 26, 'J': 28, 'K': 28, 'L': 28, 'M': 32, 'N': 30, 'O': 30, 'P': 28}
parent =>  {'A': None, 'B': 'A', 'C': 'A', 'D': 'B', 'E': 'C', 'F': 'E', 'G': None, 'H': 'C', 'I': 'H', 'J': 'I', 'K': 'D', 'L': 'D', 'M': 'D', 'N': 'J', 'O': 'L', 'P': 'K'}
optimal path =>  A->B->D->K->P
```


Result:

Thus the program to implement informed search algorithm have been executed successfully and output got verified.

3. Implement Naïve Bayes models.

Aim:

To diagnose the climate dataset withNaïve Bayes Classifier Algorithm.

Algorithm:

1. Import the required libraries: numpy, matplotlib.pyplot, pandas, seaborn.
2. Load the dataset from the given CSV file "NaiveBayes.csv" using the pandas "read_csv()" function.
3. Separate the input and output variables from the dataset by using "iloc" and "values" methods and assign them to "X" and "y" variables respectively.
4. Split the dataset into training and testing datasets using the "train_test_split()" function from the "sklearn.model_selection" module. Assign the split data to "X_train", "X_test", "y_train" and "y_test" variables.
5. Standardize the input data using the "StandardScaler()" function from the "sklearn.preprocessing" module. Scale the training data and testing data separately and assign them to "X_train" and "X_test" variables.
6. Create a Bernoulli Naive Bayes classifier object using the "BernoulliNB()" function from the "sklearn.naive_bayes" module and assign it to the "classifer" variable.
7. Train the Bernoulli Naive Bayes classifier using the "fit()" method of the "classifer" object by passing the "X_train" and "y_train" variables as arguments.
8. Predict the output values for the test dataset using the "predict()" method of the "classifer" object and assign them to "y_pred" variable.
9. Calculate the accuracy score of the model by passing the predicted output values "y_pred" and actual output values "y_test" to the "accuracy_score()" function from the "sklearn.metrics" module and print the result.
10. Create a Gaussian Naive Bayes classifier object using the "GaussianNB()" function from the "sklearn.naive_bayes" module and assign it to the "classifer1" variable.
11. Train the Gaussian Naive Bayes classifier using the "fit()" method of the "classifer1" object by passing the "X_train" and "y_train" variables as arguments.
12. Predict the output values for the test dataset using the "predict()" method of the "classifer1" object and assign them to "y_pred1" variable.
13. Calculate the accuracy score of the model by passing the predicted output values "y_pred1" and actual output values "y_test" to the "accuracy_score()" function from the "sklearn.metrics" module and print the result.

Program:

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import seaborn as sns
dataset = pd.read_csv('NaiveBayes.csv')
# split the data into inputs and outputs
X = dataset.iloc[:, [0,1]].values
y = dataset.iloc[:, 2].values
from sklearn.model_selection import train_test_split
# assign test data size 25%
X_train, X_test, y_train, y_test =train_test_split(X,y,test_size= 0.25, random_state=0)
from sklearn.preprocessing import StandardScaler
```

```
# scalling the input data
sc_X = StandardScaler()
X_train = sc_X.fit_transform(X_train)
X_test = sc_X.fit_transform(X_test)
from sklearn.naive_bayes import BernoulliNB
# initializaing the NB
classifer = BernoulliNB()
# training the model
classifer.fit(X_train, y_train)
# testing the model
y_pred = classifer.predict(X_test)
from sklearn.metrics import accuracy_score
# printing the accuracy of the model
print(accuracy_score(y_pred, y_test))
from sklearn.naive_bayes import GaussianNB
# create a Gaussian Classifier
classifer1 = GaussianNB()
# training the model
classifer1.fit(X_train, y_train)

# testing the model
y_pred1 = classifer1.predict(X_test)
from sklearn.metrics import accuracy_score
# printing the accuracy of the model
print(accuracy_score(y_test,y_pred1))
```

Result

Thus the program with Naïve Bayes Classifier Algorithm have been executed successfully and output got verified

3. Implement Bayesian Networks

Aim:

To construct a Bayesian network, to demonstrate the diagnosis of heart patientsusing standard Heart Disease Data Set.

Algorithm:

Step 1: Import required modules
Step 2: Define network structure
Step 3: Define the parameters using CPT
Step 4: Associate the parameters with the model structure
Step 5: Check if the cpds are valid for the model
Step 6: View nodes and edges of the model
Step 7: Check independencies of a node
Step 8: List all Independencies

Program:

```
from pgmpy.models import BayesianNetwork
from pgmpy.inference import VariableElimination
# Defining network structure

alarm_model = BayesianNetwork
(
   [
      ("Burglary", "Alarm"),
      ("Earthquake", "Alarm"),
      ("Alarm", "JohnCalls"),
      ("Alarm", "MaryCalls"),
   ]
)

# Defining the parameters using CPT
from pgmpy.factors.discrete import TabularCPD

cpd_burglary = TabularCPD(
   variable="Burglary", variable_card=2, values=[[0.999], [0.001]]
)
cpd_earthquake = TabularCPD(
   variable="Earthquake", variable_card=2, values=[[0.998], [0.002]]
)
cpd_alarm = TabularCPD(
   variable="Alarm",
   variable_card=2,
   values=[[0.999, 0.71, 0.06, 0.05], [0.001, 0.29, 0.94, 0.95]],
   evidence=["Burglary", "Earthquake"],
   evidence_card=[2, 2],
)
cpd_johncalls = TabularCPD(
   variable="JohnCalls",
   variable_card=2,
   values=[[0.95, 0.1], [0.05, 0.9]],
```

```
    evidence=["Alarm"],
    evidence_card=[2],
)
cpd_marycalls = TabularCPD(
    variable="MaryCalls",
    variable_card=2,
    values=[[0.1, 0.7], [0.9, 0.3]],
    evidence=["Alarm"],
    evidence_card=[2],
)

# Associating the parameters with the model structure
alarm_model.add_cpds(
    cpd_burglary, cpd_earthquake, cpd_alarm, cpd_johncalls, cpd_marycalls
)
# Checking if the cpds are valid for the model
alarm_model.check_model()

# Viewing nodes of the model
alarm_model.nodes()

# Viewing edges of the model
alarm_model.edges()

# Checking independcies of a node
alarm_model.local_independencies("Burglary")


# Listing all Independencies
alarm_model.get_independencies()
```

Output:

```
True
NodeView(('Burglary', 'Alarm', 'Earthquake', 'JohnCalls', 'MaryCalls'))
OutEdgeView([('Burglary', 'Alarm'), ('Alarm', 'JohnCalls'), ('Alarm', 'MaryCalls'),
('Earthquake', 'Alarm')])
(Burglary ⊥ Earthquake)
(MaryCalls ⊥ Earthquake, Burglary, JohnCalls | Alarm) (MaryCalls ⊥ Burglary,
JohnCalls | Earthquake, Alarm)
(MaryCalls ⊥ Earthquake, JohnCalls | Burglary, Alarm)
(MaryCalls ⊥ Earthquake, Burglary | JohnCalls, Alarm)
 (MaryCalls ⊥ JohnCalls | Earthquake, Burglary, Alarm)
(MaryCalls ⊥ Burglary | Earthquake, JohnCalls, Alarm)
 (MaryCalls ⊥ Earthquake | Burglary, JohnCalls, Alarm)
(JohnCalls ⊥ Earthquake, Burglary, MaryCalls | Alarm)
(JohnCalls ⊥ Burglary, MaryCalls | Earthquake, Alarm)
(JohnCalls ⊥ Earthquake, MaryCalls | Burglary, Alarm)
 (JohnCalls ⊥ Earthquake, Burglary | MaryCalls, Alarm)
(JohnCalls ⊥ MaryCalls | Earthquake, Burglary, Alarm)
```

(JohnCalls ⊥ Burglary | Earthquake, MaryCalls, Alarm)
(JohnCalls ⊥ Earthquake | Burglary, MaryCalls, Alarm)
(Earthquake ⊥ Burglary)
(Earthquake ⊥ MaryCalls, JohnCalls | Alarm)
(Earthquake ⊥ MaryCalls, JohnCalls | Burglary, Alarm)
(Earthquake ⊥ JohnCalls | MaryCalls, Alarm)
(Earthquake ⊥ MaryCalls | JohnCalls, Alarm)
(Earthquake ⊥ JohnCalls | Burglary, MaryCalls, Alarm)
(Earthquake ⊥ MaryCalls | Burglary, JohnCalls, Alarm)
(Burglary ⊥ Earthquake)
(Burglary ⊥ MaryCalls, JohnCalls | Alarm)
(Burglary ⊥ MaryCalls, JohnCalls | Earthquake, Alarm)
(Burglary ⊥ JohnCalls | MaryCalls, Alarm)
(Burglary ⊥ MaryCalls | JohnCalls, Alarm)
(Burglary ⊥ JohnCalls | Earthquake, MaryCalls, Alarm)
(Burglary ⊥ MaryCalls | Earthquake, JohnCalls, Alarm)

Result:

Thus the program to implement a bayesian networks have been executed successfully and the output got verified.

4. Build Regression models

Aim:

To build regression models such as locally weighted linear regression and plot the necessary graphs.

Algorithm:

1. Read the Given data Sample to X and the curve (linear or non-linear) to Y
2. Set the value for Smoothening parameter or Free parameter say τ
3. Set the bias /Point of interest set x0 which is a subset of X
4. Determine the weight matrix using :

$$w(x, x_o) = e^{-\frac{(x-x_o)^2}{2\tau^2}}$$

5. Determine the value of model term parameter β using :

$$\hat{\beta}(x_o) = (X^T W X)^{-1} X^T W y$$

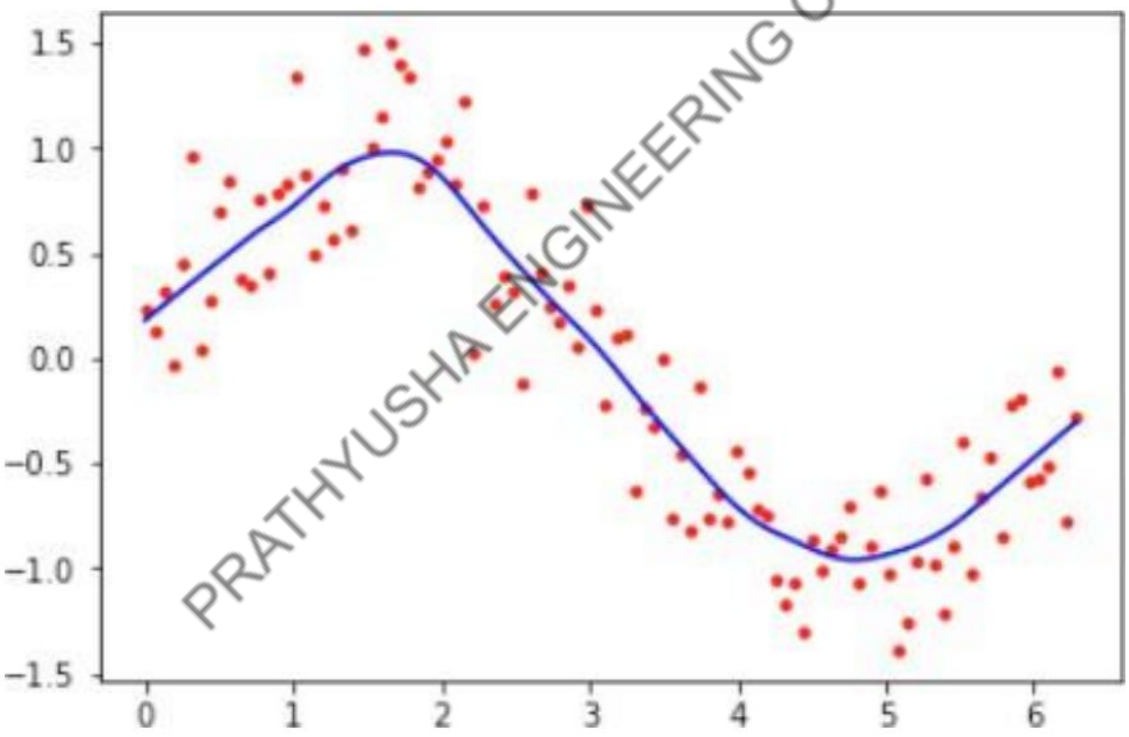
6. Prediction = x0*β.

Program:

```
from math import ceil
import numpy as np
from scipy import linalg

def lowess(x, y, f, iterations):
   n = len(x)
   r = int(ceil(f * n))
   h = [np.sort(np.abs(x - x[i]))[r] for i in range(n)]
   w = np.clip(np.abs((x[:, None] - x[None, :]) / h), 0.0, 1.0)
   w = (1 - w ** 3) ** 3
   yest = np.zeros(n)
   delta = np.ones(n)
   for iteration in range(iterations):
      for i in range(n):
         weights = delta * w[:, i]
         b = np.array([np.sum(weights * y), np.sum(weights * y * x)])
         A = np.array([[np.sum(weights), np.sum(weights * x)],[np.sum(weights * x),
np.sum(weights * x * x)]])
         beta = linalg.solve(A, b)
         yest[i] = beta[0] + beta[1] * x[i]

      residuals = y - yest
      s = np.median(np.abs(residuals))
```

```
        delta = np.clip(residuals / (6.0 * s), -1, 1)
        delta = (1 - delta ** 2) ** 2

    return yest

import math
n = 100
x = np.linspace(0, 2 * math.pi, n)
y = np.sin(x) + 0.3 * np.random.randn(n)
f =0.25
iterations=3
yest = lowess(x, y, f, iterations)

import matplotlib.pyplot as plt
plt.plot(x,y,"r.")
plt.plot(x,yest,"b-")
```

Output:



Result

Thus the program to implement non-parametric Locally Weighted Regression algorithm in order to fit data points with a graph visualization have been executed successfully.

5. Build decision trees and random forests.

Aim:

To implement the concept of decision trees with suitable dataset from real world problems using CART algorithm.

Algorithm:

Steps in CART algorithm:

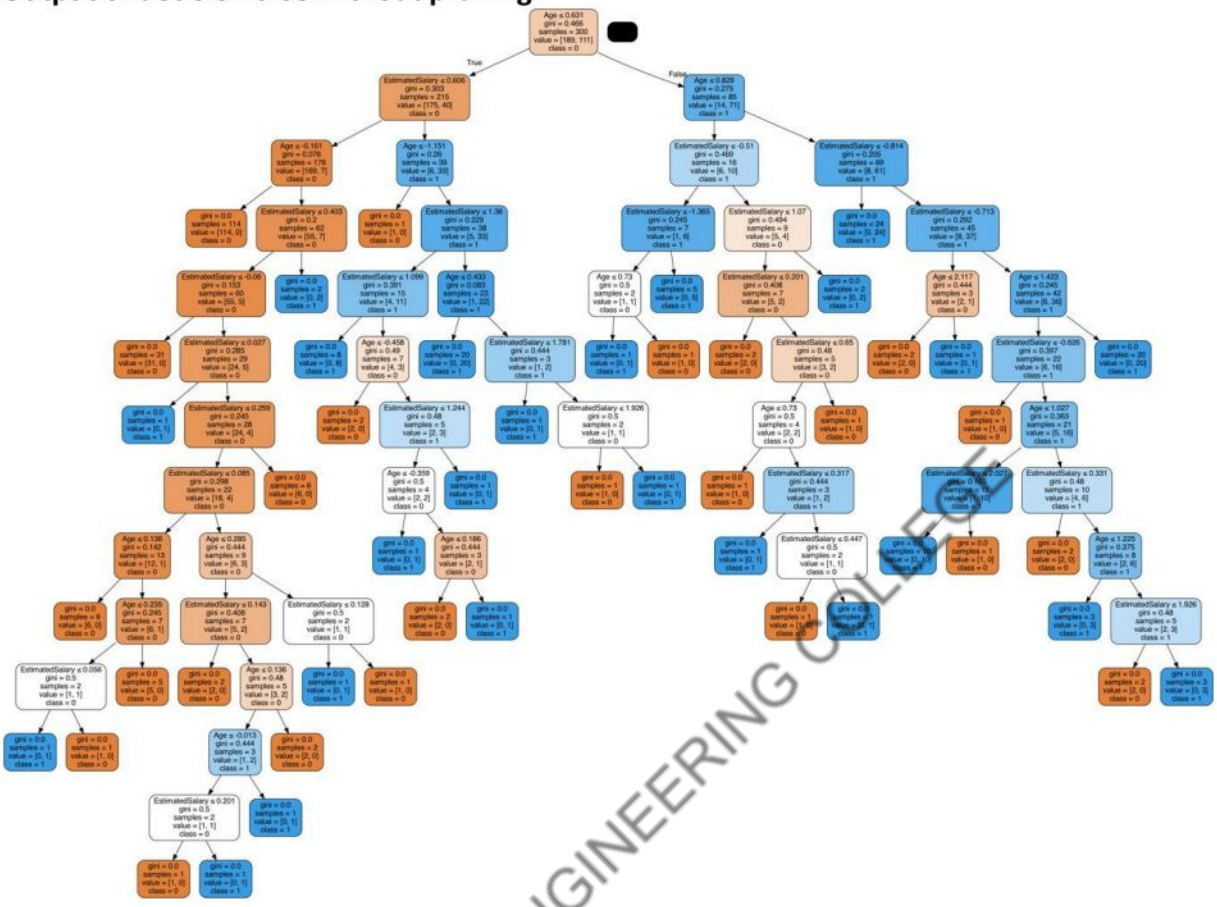
1. It begins with the original set S as the root node.
2. On each iteration of the algorithm, it iterates through the very unused attribute of the set S and calculates Gini index of this attribute.
3. Gini Index works with the categorical target variable "Success" or "Failure". It performs only Binary splits.
4. The set S is then split by the selected attribute to produce a subset of the data.
5. The algorithm continues to recur on each subset, considering only attributes never selected before.

Program:

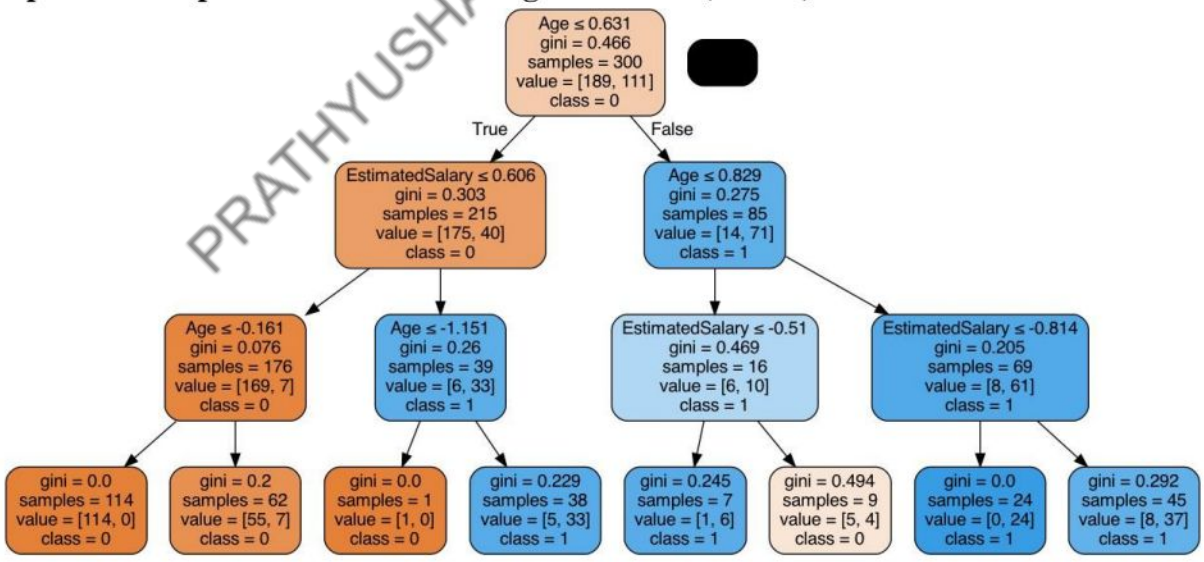
```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd

data = pd.read_csv('Social_Network_Ads.csv')
data.head()

feature_cols = ['Age', 'EstimatedSalary']
x = data.iloc[:, [2, 3]].values
y = data.iloc[:, 4].values

from sklearn.model_selection import train_test_split
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.25, random_state=0)

from sklearn.preprocessing import StandardScaler
sc_x = StandardScaler()
x_train = sc_x.fit_transform(x_train)
x_test = sc_x.transform(x_test)

from sklearn.tree import DecisionTreeClassifier
classifier = DecisionTreeClassifier()
classifier = classifier.fit(x_train, y_train)

y_pred = classifier.predict(x_test)
```

```
from sklearn import metrics
print('Accuracy Score:', metrics.accuracy_score(y_test, y_pred))

from sklearn.metrics import confusion_matrix
cm = confusion_matrix(y_test, y_pred)
print(cm)

from matplotlib.colors import ListedColormap
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(("red", "green")))
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], c=ListedColormap(("red", "green"))(i),
label=j)

plt.title("Decision Tree(Test set)")
plt.xlabel("Age")
plt.ylabel("Estimated Salary")
plt.legend()
plt.show()

from sklearn.tree import export_graphviz
from six import StringIO
from IPython.display import Image
import pydotplus

dot_data = StringIO()
export_graphviz(classifier, out_file=dot_data, filled=True, rounded=True,
special_characters=True, feature_names=feature_cols, class_names=['0', '1'])
graph = pydotplus.graph_from_dot_data(dot_data.getvalue())
Image(graph.write_png('decisiontree.png'))

classifier = DecisionTreeClassifier(criterion="gini", max_depth=3)
classifier = classifier.fit(x_train, y_train)
y_pred = classifier.predict(x_test)
print("Accuracy:", metrics.accuracy_score(y_test, y_pred))

dot_data = StringIO()
export_graphviz(classifier, out_file=dot_data, filled=True, rounded=True,
special_characters=True, feature_names=feature_cols, class_names=['0', '1'])
graph = pydotplus.graph_from_dot_data(dot_data.getvalue())
Image(graph.write_png('opt_decisiontree_gini.png'))
```

Output of decision tree without pruning:



Optimized output of decision tree using Gini Index (CART)



Result:

Thus the program to implement the concept of decision trees with suitable datasetfrom real world problems using CART algorithm have been executed successfully.