GOVERNMENT COLLEGE OF ENGINEERING, ERODE – 638 316

RECORD NOTE BOOK

Register number:

Certified that this is the bonafide record of work done by Selvan / Selvi _______________________________________ of SIXTH Semester of B.E Electrical and Electronics Engineering branch during the Academic Year 2023 – 2024 in the OCS351 – Artificial Intelligence and Machine Learning Fundamentals.

Staff In-Charge **Head of the Department**

Submitted for the Anna University practical examination on __________ at Government College of Engineering, Erode – 638 316.

Date: ________________

Internal Examiner **External Examiner**

LIST OF EXPERIMENTS

EX. NO	DATE	EXPRIMENT NAME	PAGE NO	MARKS	SIGN
1		Breadth First Search.	3		
2		Depth First Search.	4		
3		Breadth first and depth first search in terms of time and space.	5		
4		Greedy and A* algorithms.	9		
5		Non-parametric locally weight regression algorithm in order to fit data points.	12		
6		Decision tree-based algorithm.	14		
7		Build an artificial neural network by implementing the back propagation algorithm and test the same using appropriate data sets.	17		
8		Write a program to implement the naïve Bayesian classifier.	19		
9		Neural network using self-organizing maps.	23		

Ex. No: 1

Date:

Breadth First Search

Aim:

To write a python to implement Breadth First Search.

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
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 python program to implement Breadth first search is successfully executed.

Ex. No: 2

Date:

Depth First Search

Aim:

To write a python to implement Depth First Search

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 python program to implement Depth first search is successfully executed.

Ex. No: 3

Date:

Breadth First and Depth First Search in terms of Time and Space

Aim:

To write a python program to analysis Breadth First and Depth First Search in terms of time and space

Program:

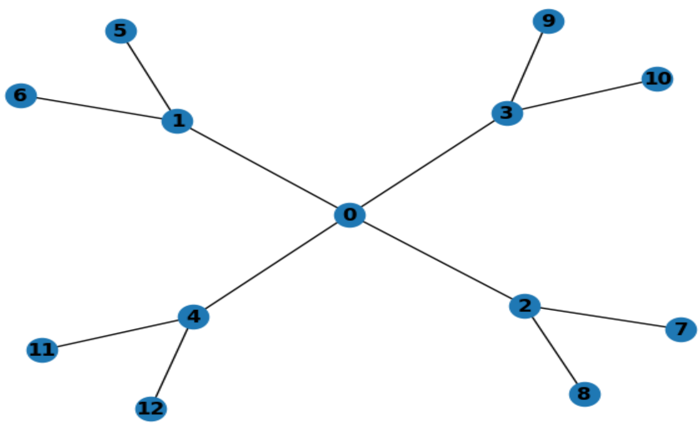
```
import networkx as nx
import matplotlib.pyplot as plt
import timeit
import sys
from collections import defaultdict

class Graph:
    def __init__(self):
        self.graph = defaultdict(list)

    def add_edge(self, u, v):
        self.graph[u].append(v)

def bfs(graph, start):
    visited = set()
    queue = [start]
    visited.add(start)

    while queue:
        node = queue.pop(0)
        for neighbor in graph[node]:
            if neighbor not in visited:
                queue.append(neighbor)
```

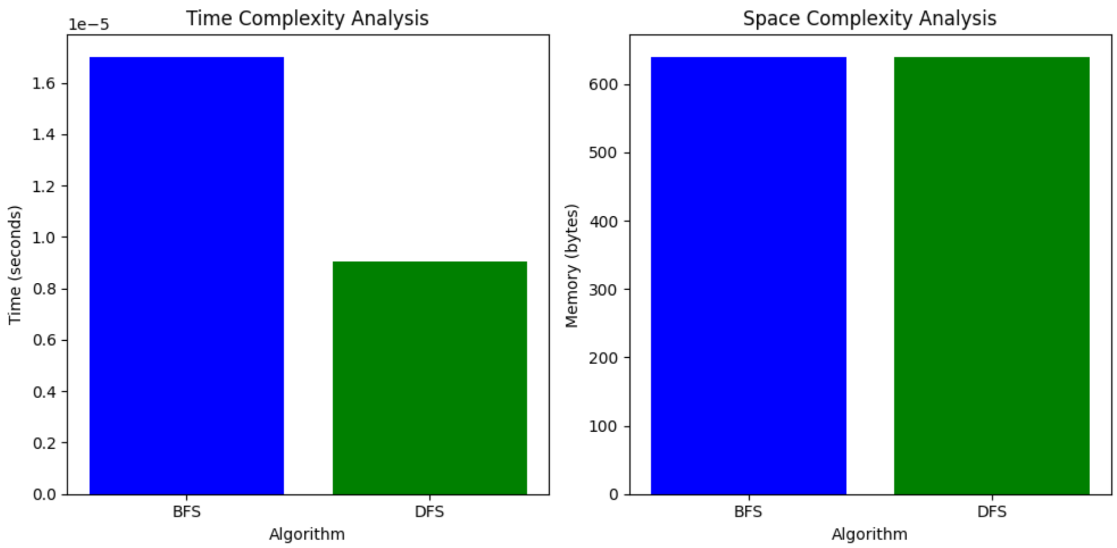
```
            visited.add(neighbor)

def dfs(graph, start, visited):
    visited.add(start)
    for neighbor in graph[start]:
        if neighbor not in visited:
            dfs(graph, neighbor, visited)
def measure_time_and_space(func, *args):
    start_time = timeit.default_timer()
    func(*args)
    end_time = timeit.default_timer()
    execution_time = end_time - start_time
    memory_usage = sys.getsizeof(args[0])
    return execution_time, memory_usage
def visualize_graph(graph):
    G = nx.Graph(graph)
    pos = nx.spring_layout(G)  # You can use other layout algorithms as well
    nx.draw(G, pos, with_labels=True, font_weight='bold')
    plt.show()

def generate_analysis_graphs(bfs_time, dfs_time, bfs_memory, dfs_memory):
    labels = ['BFS', 'DFS']
    time_values = [bfs_time, dfs_time]
    memory_values = [bfs_memory, dfs_memory]

    # Plotting time complexity
    plt.figure(figsize=(10, 5))
    plt.subplot(1, 2, 1)
    plt.bar(labels, time_values, color=['blue', 'green'])
    plt.title('Time Complexity Analysis')
    plt.xlabel('Algorithm')
    plt.ylabel('Time (seconds)')
```

```
    # Plotting space complexity
    plt.subplot(1, 2, 2)
    plt.bar(labels, memory_values, color=['blue', 'green'])
    plt.title('Space Complexity Analysis')
    plt.xlabel('Algorithm')
    plt.ylabel('Memory (bytes)')

    plt.tight_layout()
    plt.show()
if __name__ == "__main__":
    # Define a larger graph
    larger_graph = {
        0: [1, 2, 3, 4],
        1: [5, 6],
        2: [7, 8],
        3: [9, 10],
        4: [11, 12],
        5: [],
        6: [],
        7: [],
        8: [],
        9: [],
        10: [],
        11: [],
        12: []
    }

    start_node = 0
    # Perform BFS and DFS on the larger graph
    bfs_time, bfs_memory = measure_time_and_space(bfs, larger_graph, start_node)
```

```
visited = set()
dfs_time, dfs_memory = measure_time_and_space(dfs, larger_graph, start_node, visited)


print(f"BFS Time: {bfs_time:.6f} seconds, Memory: {bfs_memory} bytes")
print(f"DFS Time: {dfs_time:.6f} seconds, Memory: {dfs_memory} bytes")
visualize_graph(larger_graph)
generate_analysis_graphs(bfs_time, dfs_time, bfs_memory, dfs_memory)
```

Output:





Result:

Thus, the python program to analysis Breadth First and Depth First Search in terms of time and space is successfully executed.

Ex. No: 4

Date:

Greedy and A* algorithms.

Aim:

To write a python program to implement and compare Greedy and A* algorithms

Program:

```
import heapq
class PriorityQueue:
    def __init__(self):
        self.elements = []
    def empty(self):
        return len(self.elements) == 0
    def put(self, item, priority):
        heapq.heappush(self.elements, (priority, item))
    def get(self):
        return heapq.heappop(self.elements)[1]
def greedy_best_first_search(graph, start, goal):
    frontier = PriorityQueue()
    frontier.put(start, 0)
    explored = set()
    came_from = {}
    while not frontier.empty():
        current_node = frontier.get()
        if current_node == goal:
            return construct_path(start, goal, came_from)
        explored.add(current_node)
        for neighbor in graph[current_node]:
            if neighbor not in explored:
                came_from[neighbor] = current_node
                frontier.put(neighbor, heuristic(neighbor, goal))
    return None
```

```
def a_star_search(graph, start, goal):
    frontier = PriorityQueue()
    frontier.put(start, 0)
    came_from = {}
    cost_so_far = {}
    came_from[start] = None
    cost_so_far[start] = 0

    while not frontier.empty():
        current_node = frontier.get()

        if current_node == goal:
            return construct_path(start, goal, came_from)

        for next_node in graph[current_node]:
            new_cost = cost_so_far[current_node] + graph[current_node][next_node]
            if next_node not in cost_so_far or new_cost < cost_so_far[next_node]:
                cost_so_far[next_node] = new_cost
                priority = new_cost + heuristic(next_node, goal)
                frontier.put(next_node, priority)
                came_from[next_node] = current_node
    return None

# Example heuristic function (Euclidean distance)
def heuristic(node, goal):
    return ((node[0] - goal[0]) ** 2 + (node[1] - goal[1]) ** 2) ** 0.5
def construct_path(start, goal, came_from):
    current_node = goal
    path = []
    while current_node != start:
        path.append(current_node)
        current_node = came_from[current_node]
    path.append(start)
    return list(reversed(path))
```

```
# Example graph representation (dictionary of dictionaries)
graph = {
    (0, 0): {(1, 0): 1, (0, 1): 1},
    (1, 0): {(0, 0): 1, (1, 1): 1},
    (0, 1): {(0, 0): 1, (1, 1): 1},
    (1, 1): {(0, 1): 1, (1, 0): 1, (2, 1): 1},
    (2, 1): {(1, 1): 1}
}
start_node = (0, 0)
goal_node = (2, 1)

# Greedy Best-First Search
greedy_path = greedy_best_first_search(graph, start_node, goal_node)
if greedy_path:
    print("Greedy Best-First Search:")
    print("Path: ", greedy_path)
else:
    print("Goal is not reachable using Greedy Best-First Search.")
# A* Search
a_star_path = a_star_search(graph, start_node, goal_node)
if a_star_path:
    print("\nA* Search:")
    print("Path: ", a_star_path)
else:
    print("Goal is not reachable using A* Search.")
```

Output:

Greedy Best-First Search:
Path: [(0, 0), (1, 0), (1, 1), (2, 1)]

A* Search:
Path: [(0, 0), (1, 0), (1, 1), (2, 1)]

Result:

Thus, the python program to analysis Breadth First and Depth First Search in terms of time and space is successfully executed.

Ex. No: 5

Date:

Non-parametric locally weighted regression algorithm in order to fit data points. Select appropriate data set for your experiment and draw graphs

Aim:

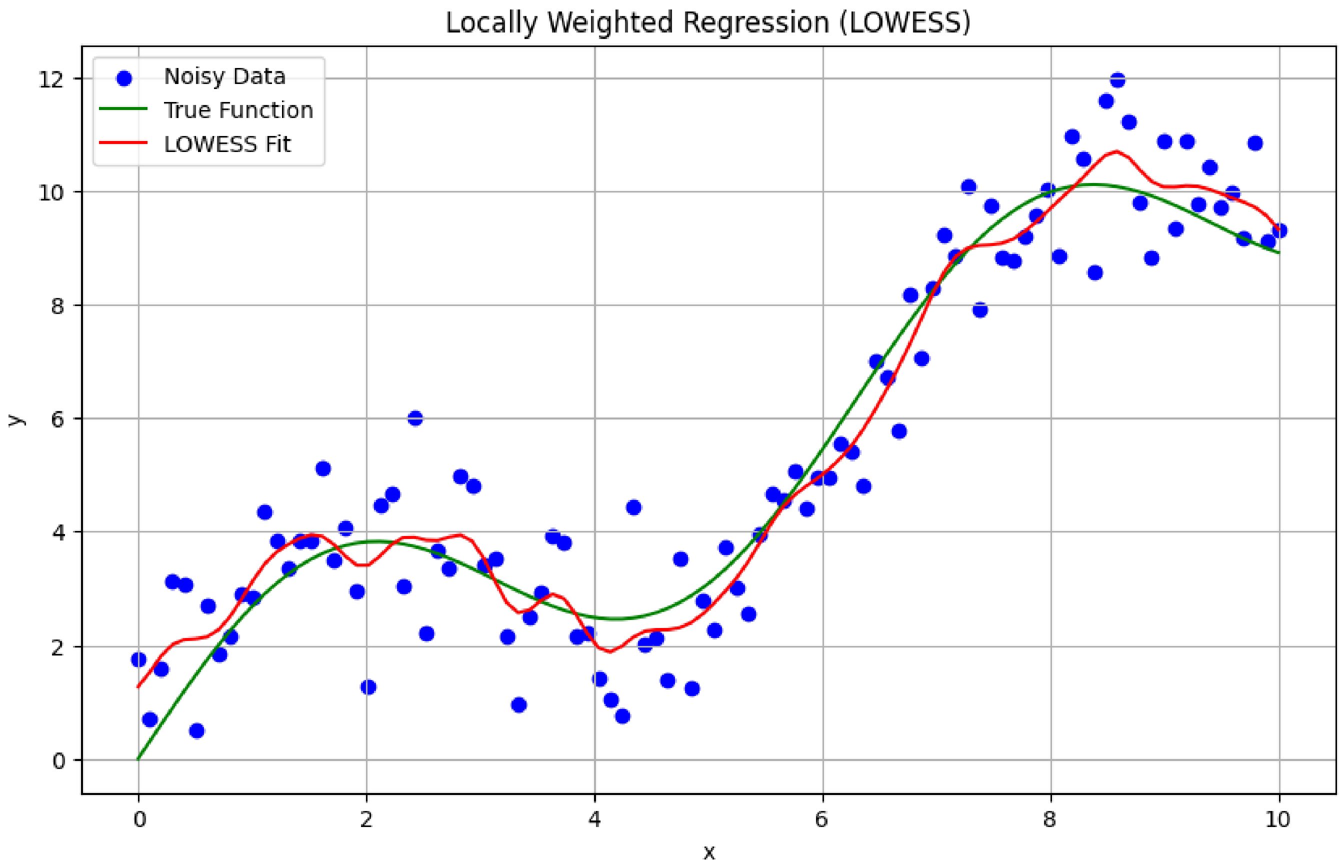
To write a python program to analysis non-parametric locally weighted regression algorithm in order to fit data points.

Program:

```
import numpy as np

import matplotlib.pyplot as plt

def lowess(x, y, tau=0.5, degree=1):

    n = len(x)

    y_pred = np.zeros(n)


    # Implementing LOWESS algorithm

    for i in range(n):

        weights = np.exp(-((x - x[i]) ** 2) / (2 * tau ** 2))

        W = np.diag(weights)

        X = np.column_stack((np.ones(n), x))

        theta = np.linalg.inv(X.T @ W @ X) @ X.T @ W @ y

        y_pred[i] = np.dot(np.array([1, x[i]]), theta)


    return y_pred


# Generating synthetic data

np.random.seed(0)

x = np.linspace(0, 10, 100)

y_true = 2 * np.sin(x) + x

noise = np.random.normal(0, 1, size=len(x))

y = y_true + noise

# Applying LOWESS regression
```

```
y_pred = lowess(x, y, tau=0.2)
# Plotting
plt.figure(figsize=(10, 6))
plt.scatter(x, y, color='blue', label='Noisy Data')
plt.plot(x, y_true, color='green', label='True Function')
plt.plot(x, y_pred, color='red', label='LOWESS Fit')
plt.title('Locally Weighted Regression (LOWESS)')
plt.xlabel('x')
plt.ylabel('y')
plt.legend()
plt.grid(True)
plt.show()
```

Output:



Result:

Thus, the python program to analysis non-parametric locally weighted regression algorithm in order to fit data points.

Ex. No: 6

Date:

Decision tree-based algorithm

Aim:

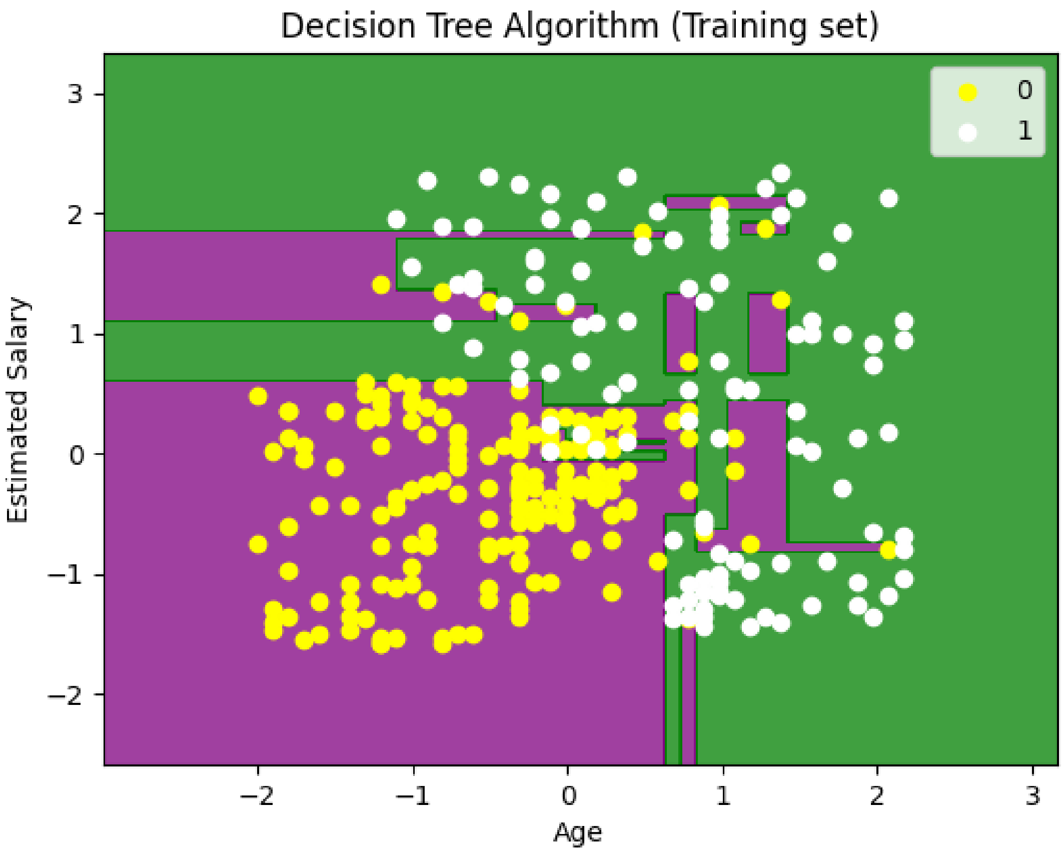
To write a python program to demonstrate the working of the decision tree-based algorithm.

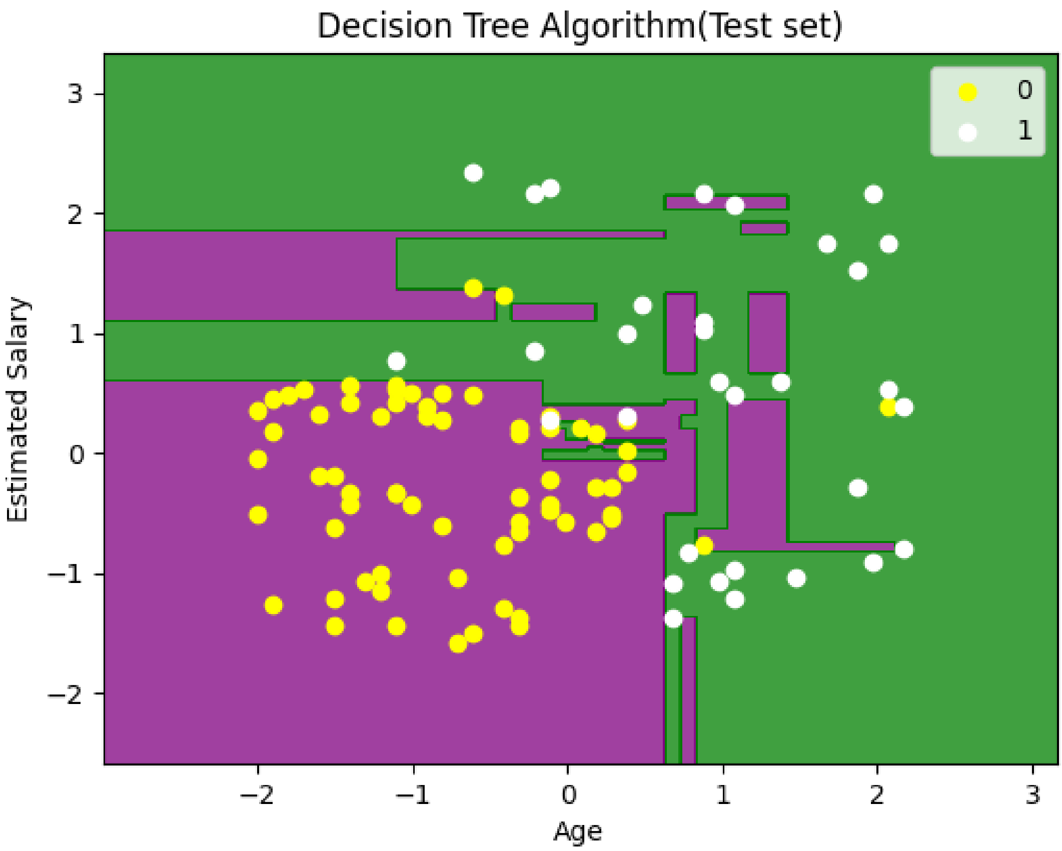
Program:

```
import numpy as nm
import matplotlib.pyplot as mtp
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('carve.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)
x_set, y_set = x_train, y_train
```

```
x1, x2 = nm.meshgrid(nm.arange(start = x_set[:, 0].min() - 1, stop = x_set[:, 0].max() + 1, step  =0.01),
nm.arange(start = x_set[:, 1].min() - 1, stop = x_set[:, 1].max() + 1, step = 0.01))
mtp.contourf(x1, x2, classifier.predict(nm.array([x1.ravel(), x2.ravel()]).T).reshape(x1.shape),
alpha = 0.75, cmap = ListedColormap(('purple','green' )))
mtp.xlim(x1.min(), x1.max())
mtp.ylim(x2.min(), x2.max())
for i, j in enumerate(nm.unique(y_set)):
        mtp.scatter(x_set[y_set == j, 0], x_set[y_set == j, 1], c = ListedColormap(('purple', 'green'))(i), label = j)
mtp.title('Decision Tree Algorithm (Training set)')
mtp.xlabel('Age')
mtp.ylabel('Estimated Salary')
mtp.legend()
mtp.show()
#Visulaizing the test set result
x_set, y_set = x_test, y_test
x1, x2 = nm.meshgrid(nm.arange(start = x_set[:, 0].min() - 1, stop = x_set[:, 0].max() + 1, step  =0.01),
nm.arange(start = x_set[:, 1].min() - 1, stop = x_set[:, 1].max() + 1, step = 0.01))
mtp.contourf(x1, x2, classifier.predict(nm.array([x1.ravel(), x2.ravel()]).T).reshape(x1.shape),
alpha = 0.75, cmap = ListedColormap(('purple','green' )))
mtp.xlim(x1.min(), x1.max())
mtp.ylim(x2.min(), x2.max())
for i, j in enumerate(nm.unique(y_set)):
        mtp.scatter(x_set[y_set == j, 0], x_set[y_set == j, 1], c = ListedColormap(('purple', 'green'))(i), label = j)
mtp.title('Decision Tree Algorithm(Test set)')
mtp.xlabel('Age')
mtp.ylabel('Estimated Salary')
mtp.legend()
mtp.show()
```

Output:





Result:

Thus, the python program to demonstrate the working of the decision tree-based algorithm is executed successfully.

Ex. No: 7

Date:

Build an artificial neural network by implementing the back propagation algorithm and test the same using appropriate data sets.

Aim:

To write a python program to build an artificial neural network by implementing the back propagation algorithm and test the same using appropriate data sets.

Program:

```
import numpy as np
class NeuralNetwork:
    def __init__(self, layers, learning_rate=0.1):
        self.layers = layers
        self.weights = [np.random.randn(layers[i], layers[i+1]) for i in range(len(layers)-1)]
        self.biases = [np.zeros((1, layers[i+1])) for i in range(len(layers)-1)]
        self.learning_rate = learning_rate
    def sigmoid(self, x):
        return 1 / (1 + np.exp(-x))
    def sigmoid_derivative(self, x):
        return x * (1 - x)
    def feedforward(self, X):
        activations = [X]
        for i in range(len(self.layers)-1):
            X = self.sigmoid(np.dot(X, self.weights[i]) + self.biases[i])
            activations.append(X)
        return activations

    def backpropagation(self, X, y, activations):
        deltas = [None] * (len(self.layers)-1)
        deltas[-1] = (activations[-1] - y) * self.sigmoid_derivative(activations[-1])

        for i in reversed(range(len(deltas)-1)):
            deltas[i] = np.dot(deltas[i+1], self.weights[i+1].T) * self.sigmoid_derivative(activations[i+1])

        for i in range(len(self.weights)):
            self.weights[i] -= self.learning_rate * np.dot(activations[i].T, deltas[i])
            self.biases[i] -= self.learning_rate * np.sum(deltas[i], axis=0)

    def train(self, X, y, epochs):
        for epoch in range(epochs):
            activations = self.feedforward(X)
            self.backpropagation(X, y, activations)

    def predict(self, X):
        return self.feedforward(X)[-1]

# Example usage:
```

```python
if __name__ == "__main__":
    # Example dataset
    X = np.array([[0,0],[0,1],[1,0],[1,1]])
    y = np.array([[0],[1],[1],[0]])

    # Define and train the neural network
    nn = NeuralNetwork([2, 3, 1])  # 2 input neurons, 3 hidden neurons, 1 output neuron
    nn.train(X, y, epochs=10000)

    # Make predictions
    predictions = nn.predict(X)
    print("Predictions:")
    for i in range(len(X)):
        print(f"Input: {X[i]}, Predicted output: {predictions[i]}")
```

Output:

```
Predictions:
Input: [0 0], Predicted output: [0.05691238]
Input: [0 1], Predicted output: [0.93468726]
Input: [1 0], Predicted output: [0.92721479]
Input: [1 1], Predicted output: [0.06697454]
```

Result:

Thus, the python program to build an artificial neural network by implementing the back propagation algorithm and test the same using appropriate data sets executed successfully.

Ex. No: 8

Date:

Write a program to implement the Naïve Bayesian classifier

Aim:

To write a python program to implement the Naïve Bayesian classifier in python program

Program:

```
from csv import reader
from math import sqrt
from math import exp
from math import pi

# Load a CSV file
def load_csv(filename):
    dataset = list()
    with open(filename, 'r') as file:
        csv_reader = reader(file)
        for row in csv_reader:
            if not row:
                continue
            dataset.append(row)
    return dataset

# Convert string column to float
def str_column_to_float(dataset, column):
    for row in dataset:
        row[column] = float(row[column].strip())

# Convert string column to integer
def str_column_to_int(dataset, column):
    class_values = [row[column] for row in dataset]
    unique = set(class_values)
    lookup = dict()
    for i, value in enumerate(unique):
```

```
			lookup[value] = i
			print('[%s] => %d' % (value, i))
	for row in dataset:
			row[column] = lookup[row[column]]
	return lookup

# Split the dataset by class values, returns a dictionary
def separate_by_class(dataset):
	separated = dict()
	for i in range(len(dataset)):
			vector = dataset[i]
			class_value = vector[-1]
			if (class_value not in separated):
					separated[class_value] = list()
			separated[class_value].append(vector)
	return separated

# Calculate the mean of a list of numbers
def mean(numbers):
	return sum(numbers)/float(len(numbers))

# Calculate the standard deviation of a list of numbers
def stdev(numbers):
	avg = mean(numbers)
	variance = sum([(x-avg)**2 for x in numbers]) / float(len(numbers)-1)
	return sqrt(variance)

# Calculate the mean, stdev and count for each column in a dataset
def summarize_dataset(dataset):
	summaries = [(mean(column), stdev(column), len(column)) for column in zip(*dataset)]
	del(summaries[-1])
	return summaries

# Split dataset by class then calculate statistics for each row
```

```
def summarize_by_class(dataset):
	separated = separate_by_class(dataset)
	summaries = dict()
	for class_value, rows in separated.items():
		summaries[class_value] = summarize_dataset(rows)
	return summaries

# Calculate the Gaussian probability distribution function for x
def calculate_probability(x, mean, stdev):
	exponent = exp(-((x-mean)**2 / (2 * stdev**2 )))
	return (1 / (sqrt(2 * pi) * stdev)) * exponent

# Calculate the probabilities of predicting each class for a given row
def calculate_class_probabilities(summaries, row):
	total_rows = sum([summaries[label][0][2] for label in summaries])
	probabilities = dict()
	for class_value, class_summaries in summaries.items():
		probabilities[class_value] = summaries[class_value][0][2]/float(total_rows)
		for i in range(len(class_summaries)):
			mean, stdev, _ = class_summaries[i]
			probabilities[class_value] *= calculate_probability(row[i], mean, stdev)
	return probabilities

# Predict the class for a given row
def predict(summaries, row):
	probabilities = calculate_class_probabilities(summaries, row)
	best_label, best_prob = None, -1
	for class_value, probability in probabilities.items():
		if best_label is None or probability > best_prob:
			best_prob = probability
			best_label = class_value
	return best_label

# Make a prediction with Naive Bayes on Iris Dataset
```

```
filename = 'iris.csv'
dataset = load_csv(filename)
for i in range(len(dataset[0])-1):
	str_column_to_float(dataset, i)
# convert class column to integers
str_column_to_int(dataset, len(dataset[0])-1)
# fit model
model = summarize_by_class(dataset)
# define a new record
row = [5.7,2.9,4.2,1.3]
# predict the label
label = predict(model, row)
print('Data=%s, Predicted: %s' % (row, label))
```

Output:

```
[Iris-setosa] => 0
[Iris-versicolor] => 1
[Iris-virginica] => 2
Data=[5.7, 2.9, 4.2, 1.3], Predicted: 1
```

Result:

Thus, the python program to implement the Naïve Bayesian classifier is successfully executed.

Ex. No: 9

Date:

neural network using self-organizing maps

Aim:

To write a python program to implementing neural network using self-organizing maps

Program:

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


class SOM:
    def __init__(self, width, height, input_dim, learning_rate=0.5, radius=None, epochs=1000):
        self.width = width
        self.height = height
        self.input_dim = input_dim
        self.learning_rate = learning_rate
        self.radius = radius if radius is not None else max(width, height) / 2
        self.epochs = epochs
        self.weights = np.random.random((width, height, input_dim))
        self.time_constant = epochs / np.log(self.radius)

    def _neighborhood_function(self, distance, radius):
        return np.exp(-distance**2 / (2 * (radius**2)))

    def _find_bmu(self, input_vector):
        differences = self.weights - input_vector
        distances = np.linalg.norm(differences, axis=2)
        bmu_index = np.unravel_index(np.argmin(distances), (self.width, self.height))
        return bmu_index

    def train(self, data):
        for epoch in range(self.epochs):
            for input_vector in data:
                bmu_index = self._find_bmu(input_vector)
```

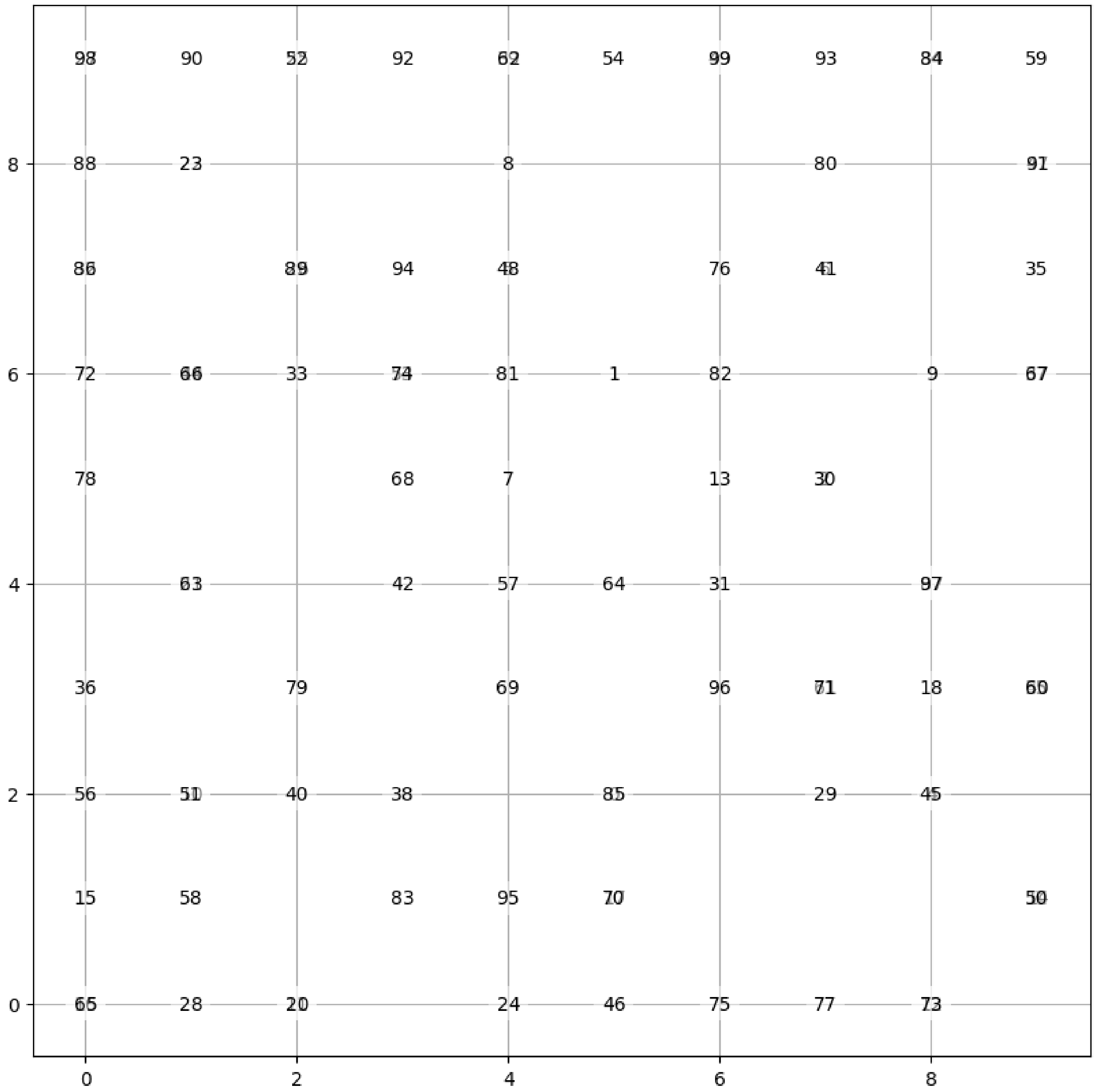
```
            bmu_x, bmu_y = bmu_index

            learning_rate = self.learning_rate * np.exp(-epoch / self.epochs)
            radius = self.radius * np.exp(-epoch / self.time_constant)

            for x in range(self.width):
                for y in range(self.height):
                    distance_to_bmu = np.linalg.norm(np.array([x, y]) - np.array([bmu_x, bmu_y]))
                    if distance_to_bmu <= radius:
                        influence = self._neighborhood_function(distance_to_bmu, radius)
                        self.weights[x, y, :] += influence * learning_rate * (input_vector - self.weights[x, y, :])

    def map_vects(self, data):
        mapped = np.array([self._find_bmu(vector) for vector in data])
        return mapped

    def plot(self, data, labels):
        mapped = self.map_vects(data)
        plt.figure(figsize=(10, 10))
        for i, m in enumerate(mapped):
            plt.text(m[0], m[1], labels[i], ha='center', va='center', bbox=dict(facecolor='white', alpha=0.5, lw=0))
        plt.xlim(-0.5, self.width-0.5)
        plt.ylim(-0.5, self.height-0.5)
        plt.grid()
        plt.show()

# Example usage:
if __name__ == "__main__":
    # Example dataset (2D points)
    data = np.random.random((100, 3))

    # Initialize and train the SOM
    som = SOM(10, 10, 3, learning_rate=0.5, epochs=100)
    som.train(data)
```

Plot the SOM with data points

labels = [str(i) for i in range(len(data))]

som.plot(data, labels)

Output:



Result:

Thus, the python program to implementing neural network using self-organizing maps is executed successfully.