Department of Computer Science and Engineering

Lab Manual

for

Artificial Intelligence and Machine Learning

Regulation 2021

CS3491 – IV Semester

ADHIPARASAKTHI ENGINEERING COLLEGE

Melmaruvathur, Chengalpet District- 603319.

INDEX

Exp.no	Name of the Experiments	Page no
1A.	IMPLEMENTATION OF BREADTH FIRST SEARCH ALGORITHM	2
1B.	IMPLEMENTATION OF DEPTH FIRST SEARCH ALGORITHM	6
2A.	IMPLEMENTATION OF A* ALGORITHM	10
2B.	IMPLEMENTATION OF SIMPLIFIED MEMORY BOUNDED A* ALGORITHNM	15
3.	IMPLEMENT NAÏVE BAYES MODELS	20
4.	IMPLEMENT BAYESIAN NETWORKS	24
5.	BUILD REGRESSION MODELS	27
6.	BUILD DECISION TREES AND RANDOM FORESTS	31
7.	BUILD SVM MODELS	34
8.	IMPLEMENT ENSEMBLING TECHNIQUES	37
9.	IMPLEMENT CLUSTERING ALGORITHMS	41
10.	IMPLEMENT EM FOR BAYESIAN NETWORKS	46
11.	BUILD SIMPLE NN MODELS	53
12.	BUILD DEEP LEARNING NN MODELS	57

EX. NO: 1A	IMPLEMENTATION OF BREADTH FIRST SEARCH ALGORITHM

Aim:

To write a Python program to perform Breadth First Search (BFS) algorithm.

Theory:

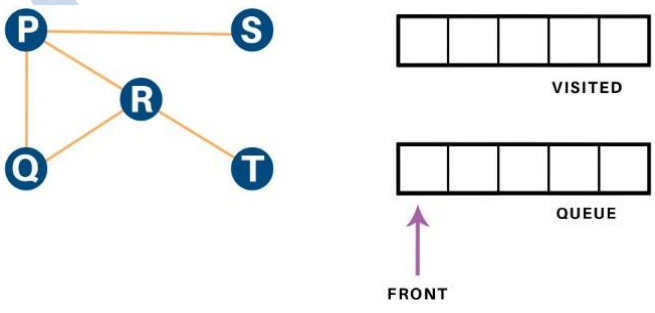
Breadth-First Search (BFS) is an algorithm used for traversing graphs or trees. Traverse means visiting each node of the graph. Breadth-First Search is a recursive algorithm to search all the vertices of a graph or a tree. BFS in Python can be implemented by using data structures like a dictionary and lists. Breadth-First Search in tree and graph is almost the same. The only difference is that the graph may contain cycles, so we may traverse to the same node again.

Algorithm:

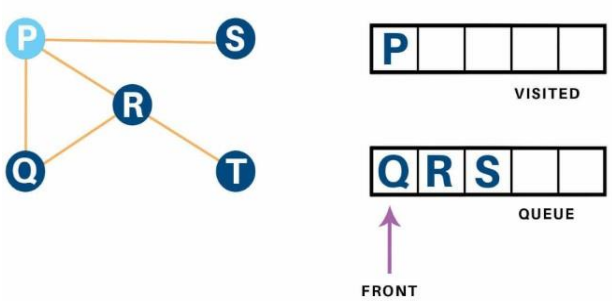
1. Pick any node, visit the adjacent unvisited vertex, mark it as visited, display it, and insert itin a queue.
2. If there are no remaining adjacent vertices left, remove the first vertex from the queue.
3. Repeat step 1 and step 2 until the queue is empty or the desired node is found.

Example:

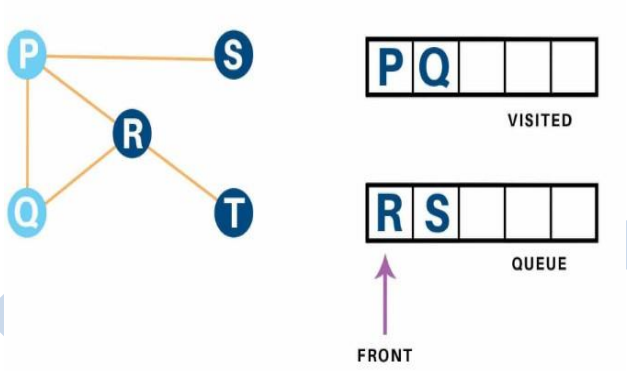
Let us use an undirected graph with 5 vertices.



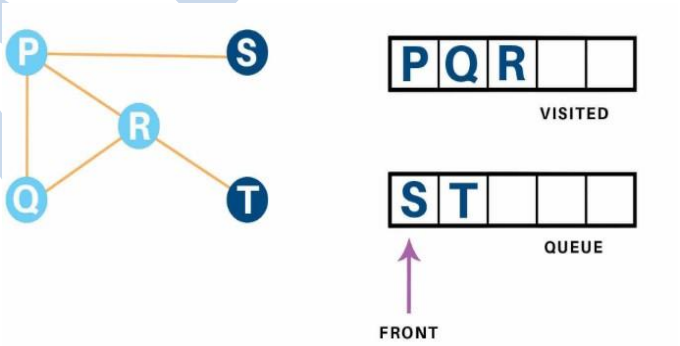
From the vertex P, the BFS algorithmic program starts by putting it within the Visited list and puts all its adjacent vertices within the stack.



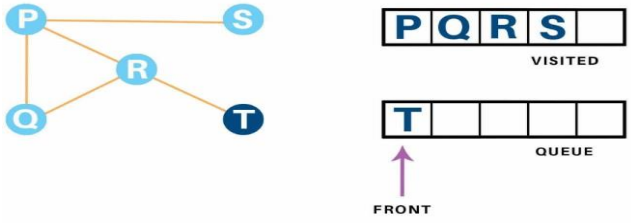
Next, we have a tendency to visit the part at the front of the queue i.e. Q and visit its adjacent nodes. Since P has already been visited, we have a tendency to visit R instead.



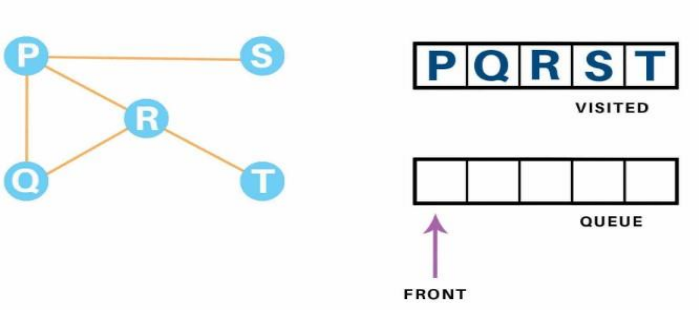
Vertex R has an unvisited adjacent vertex in T, thus we have a tendency to add that to the rear ofthe queue and visit S, which is at the front of the queue.



Now, only T remains within the queue since the only adjacent node of S i.e. P is already visited.We have a tendency to visit it.



Now, only T remains within the queue since the only adjacent node of S i.e. P is already visited.We have a tendency to visit it.



Since the queue is empty, we've completed the Traversal of the graph.

Program:

```
graph = {
   'A': ['B', 'C'],
   'B': ['D', 'E'],
   'C': ['F'],
   'D': [],
   'E': ['F'],
   'F': [],
   }
visited = []  # List to keep track of visited nodes.
queue = []  # Initialize a queue
def bfs(visited, graph, node):
   visited.append(node)
   queue.append(node)
   while queue:
      s = queue.pop(0)
      print (s)
      for neighbour in graph[s]:
         if neighbour not in visited:
            visited.append(neighbour)
            queue.append(neighbour)
bfs(visited, graph, 'A')  # Driver Code
```

Output:

A B C D E F

APEC

Result:

Thus the program to implement BFS algorithm using python is executed and verified successfully.

EX. NO: 1B	IMPLEMENTATION OF DEPTH FIRST SEARCH ALGORITHM

Aim:

To write a Python program to perform Depth First Search (DFS) algorithm.

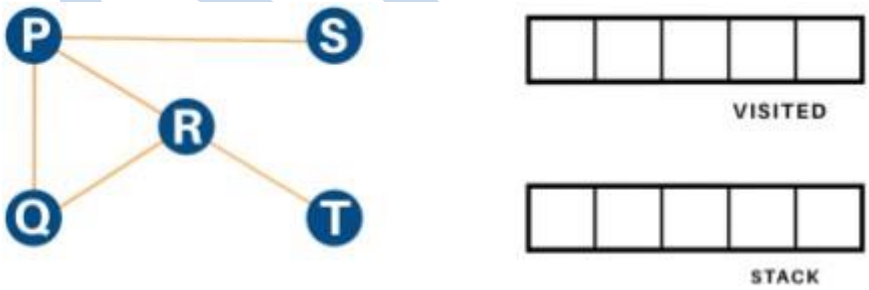
Theory:

The Depth-First Search is a recursive algorithm that uses the concept of backtracking. It involves thorough searches of all the nodes by going ahead if potential, else by backtracking. Here, the word backtrack means once you are moving forward and there are not any more nodes along the present path, you progress backward on an equivalent path to seek out nodes to traverse. All the nodes are progressing to be visited on the current path until all the unvisited nodes are traversed after which subsequent paths are going to be selected.

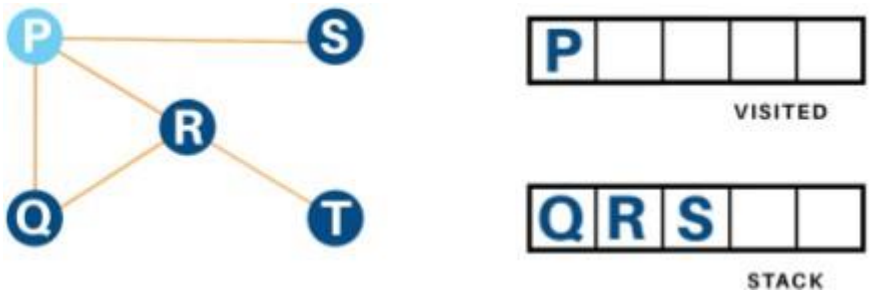
Algorithm:

1. Pick any node. If it is unvisited, mark it as visited and recur on all its adjacent nodes.
2. Repeat until all the nodes are visited, or the node to be searched is found.

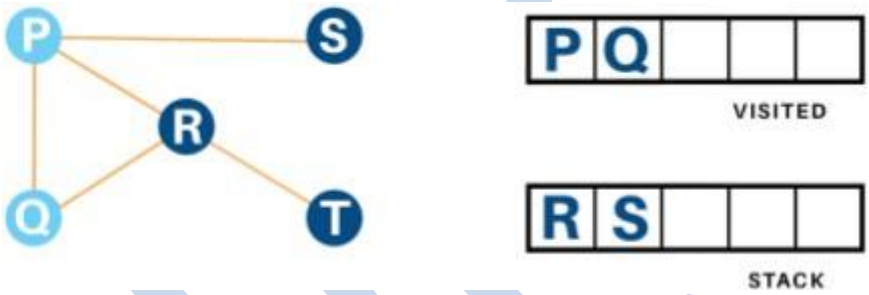
Example:



From the vertex P, the DFS rule starts by putting it within the Visited list and putting all its adjacent vertices within the stack



Next, we tend to visit the part at the highest of the stack i.e. Q, and head to its adjacent nodes. Since P has already been visited, we tend to visit R instead.



Vertex R has the unvisited adjacent vertex in T, therefore we will be adding that to the highest ofthe stack and visit it.



At last, we will visit the last component S, it does not have any unvisited adjacent nodes, thus we've completed the Depth First Traversal of the graph.



Program:

```
graph = {
   'A': ['B', 'C'],
   'B': ['D', 'E'],
   'C': ['F'],
   'D': [],
   'E': ['F'],
   'F': [],
   }
visited = set()  # Set to keep track of visited nodes.

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

dfs(visited, graph, 'A')  # Driver Code
```

APEC

Output:

A

B

D

E

F

C

APEC

Result:

Thus the program to implement Depth First Search algorithm using python is executed andverified successfully.

EX. NO: 2A	IMPLEMENTATION OF A* ALGORITHM

Aim:

To implement an A* algorithm using python.

Algorithm:

1. Add start node to list
2. For all the neighbouring nodes, find the least cost F node
3. Switch to the closed list
   - For 8 nodes adjacent to the current node
   - If the node is not reachable, ignore it. Else
   - If the node is not on the open list, move it to the open list and calculate f, g,h.
   - If the node is on the open list, check if the path it offers is less than the current pathand change to it if it does so.
4. Stop working when
   - You find the destination
   - You cannot find the destination going through all possible points.

Program:

```
def aStarAlgo(start_node, stop_node):
    open_set = set(start_node)  # {A}, len{open_set}=1
    closed_set = set()
    g = {}  # store the distance from starting node
    parents = {}
    g[start_node] = 0
    parents[start_node] = start_node  # parents['A']='A"
    while len(open_set) > 0:
        n = None
        for v in open_set:  # v='B'/'F'
            if n == None or g[v] + heuristic(v) < g[n] + heuristic(n):
                n = v  # n='A'
```

```
if n == stop_node or Graph_nodes[n] == None:
    pass
else:
    for (m, weight) in get_neighbors(n):
    # nodes 'm' not in first and last set are added to first
    # n is set its parent
        if m not in open_set and m not in closed_set:
            open_set.add(m)  # m=B weight=6 {'F','B','A'} len{open_set}=2
            parents[m] = n  # parents={'A':A,'B':A} len{parent}=2
            g[m] = g[n] + weight  # g={'A':0,'B':6, 'F':3} len{g}=2
        else:
            # for each node m,compare its distance from start i.e g(m) to the
            # from start through n node
            if g[m] > g[n] + weight:
                # update g(m)
                g[m] = g[n] + weight
                # change parent of m to n
                parents[m] = n
        # if m in closed set,remove and add to open
            if m in closed_set:
                closed_set.remove(m)
                open_set.add(m)
if n == None:
    print ('Path does not exist!')
    return None
# if the current node is the stop_node
# then we begin reconstruct in the path from it to the start_node
if n == stop_node:
    path = []
    while parents[n] != n:
        path.append(n)
        n = parents[n]
```

```
            path.append(start_node)
            path.reverse()
            print ('Path found: {}'.format(path))
            return path
        # remove n from the open_list, and add it to closed_list
        # because all of his neighbors were inspected
        open_set.remove(n)  # {'F','B'} len=2
        closed_set.add(n)  # {A} len=1
    print ('Path does not exist!')
    return None

# define fuction to return neighbor and its distance #from the passed node
def get_neighbors(v):
    if v in Graph_nodes:
        return Graph_nodes[v]
    else:
        return None

# for simplicity we ll consider heuristic distances given
# and this function returns heuristic distance for all nodes
def heuristic(n):
    H_dist = {
        'A': 10,
        'B': 8,
        'C': 5,
        'D': 7,
        'E': 3,
        'F': 6,
        'G': 5,
        'H': 3,
        'I': 1,
        'J': 0,
```

```
        }
    return H_dist[n]



# Describe your graph here
Graph_nodes = {
    'A': [('B', 6), ('F', 3)],
    'B': [('C', 3), ('D', 2)],
    'C': [('D', 1), ('E', 5)],
    'D': [('C', 1), ('E', 8)],
    'E': [('I', 5), ('J', 5)],
    'F': [('G', 1), ('H', 7)],
    'G': [('I', 3)],
    'H': [('I', 2)],
    'I': [('E', 5), ('J', 3)],
    }

aStarAlgo('A', 'J')
```

APEC

Output:

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

APEC

Result:

Thus the program to implement A* algorithm using python is executed and verifiedsuccessfully.

EX. NO: 2B	IMPLEMENTATION OF SIMPLIFIED MEMORY BOUNDED A* ALGORITHNM

Aim:

To implement an SMA A* algorithm using python.

Algorithm:

1. Initialize the data structures:

   f_scores: a dictionary mapping states to their f-scores (f = g + h)

   g_scores: a dictionary mapping states to their g-scores (g = cost to reach the state)

   parents: a dictionary mapping states to their parent states (used to construct the final path)

   closed_set: a set of states that have been expanded

   open_set: a priority queue (heap) of states that are currently being considered for expansion,sorted by their f-scores

   expansions: a counter that keeps track of the number of expansions that have been performed Initialize **f_scores[start_state] = heuristic(start_state), g_scores[start_state] = 0, parents[start_state] = None** and **open_set** with **(f_scores[start_state], start_state).**

2. While **open_set** is not empty and **expansions** is less than **max_expansions**, do the following:
   a. Pop the state with the lowest f-score from **open_set**. This will be the current state.
   b. Check if the goal has been reached by calling **goal_test(current_state)**. If so, return the finalpath by calling **construct_path(parents, current_state).**
   c. Add the current state to **closed_set** and increment **expansions.**
   d. Expand the current state by generating its successor states using the **successors(current_state)** function. Each successor state is represented as a tuple **(successor_state, successor_cost)**, where **successor_state** is the new state and **successor_cost** is the cost to get there from the current state.
   e. For each successor state:
      I. Calculate the tentative g-score by adding the successor cost to the current state's g-score: tentative_g_score = g_scores[current_state] + successor_cost**.**

II. Check if the successor state is already in **closed_set.** If so, skip this successor state and moveon to the next one.

III. If the successor state is not in **g_scores**, or if the tentative g-score is less than the current g-score forthe successor state, update its **g_scores, f_scores**, and **parents** as follows:

-`g_scores[successor_state] = tentative_g_score`

-`f_scores[successor_state] = g_scores[successor_state] + suboptimality * heuristic(successor_state)`

-`parents[successor_state] = current_state`

If the successor state is not already in `open_set`, add it to `open_set` with `(f_scores[successor_state], successor_state)`.

3. If the loop terminates without finding a goal state, return **None.**
4. To construct the final path, start with the goal state and repeatedly follow its parent pointersuntil the start state is reached. Return the path as a list of states in order from start to goal.

Program:

```
import heapq
def sma_star(  # Initialize data structures
    start_state,
    goal_test,
    successors,
    heuristic,
    suboptimality,
    max_expansions,
    ):
    f_scores = {start_state: heuristic(start_state)}
    g_scores = {start_state: 0}
    parents = {start_state: None}
    closed_set = set()
    open_set = []
```

```
    heapq.heappush(open_set, (f_scores[start_state], start_state))
    expansions = 0
    while open_set and expansions < max_expansions:
        # Pop the state with the lowest f-score from the open set
        current_state = heapq.heappop(open_set)[1]
        # Check if the goal has been reached
        if goal_test(current_state):
            return construct_path(parents, current_state)
        closed_set.add(current_state)
        expansions += 1
        # Expand the current state
        for successor in successors(current_state):
            (successor_state, successor_cost) = successor
            # Calculate tentative g-score
            tentative_g_score = g_scores[current_state] + successor_cost
            # Check if the successor is already closed
        if successor_state in closed_set:
            continue
        # Add the successor to the open set if it's new
        if successor_state not in g_scores or tentative_g_score \
            < g_scores[successor_state]:
            parents[successor_state] = current_state
            g_scores[successor_state] = tentative_g_score
            f_scores[successor_state] = g_scores[successor_state] \
                + suboptimality * heuristic(successor_state)
            heapq.heappush(open_set, (f_scores[successor_state],
                        successor_state))  # No path found
    return None

def construct_path(parents, goal_state):
    path = [goal_state]
    while parents[path[0]] is not None:
```

```
        path.insert(0, parents[path[0]])
    return path

# Example usage
def goal_test(state):
    return state == 5

def successors(state):
    if state == 1:
        return [(2, 1), (3, 2)]
    elif state == 2:
        return [(4, 3), (5, 4)]
    elif state == 3:
        return [(4, 1), (5, 2)]
    else:
        return []

def heuristic(state):
    return abs(state - 5)

start_state = 1
suboptimality = 2
max_expansions = 10
path = sma_star(
    start_state,
    goal_test,
    successors,
    heuristic,
    suboptimality,
    max_expansions,
    )
print (path)
```

Output:

[1, 3, 5]

APEC

Result:

Thus the program to implement SMA A* algorithm using python is executed and verifiedsuccessfully.

EX. NO: 3	IMPLEMENTATION NAÏVE BAYES MODELS

Aim:

To implement the Naïve Bayes models.

Algorithm:

1. Load the dataset: Load the dataset that you want to classify using Naïve Bayes algorithm. Thedataset should have labelled data points with attributes and their corresponding classes.
2. Split the dataset: Split the dataset into training and testing sets. Use the training set to train theNaïve Bayes classifier and the testing set to evaluate its performance.
3. Pre-process the data: Pre-process the data by removing any irrelevant or noisy attributes, cleaningthe text data (if applicable), and converting the data into numerical form.
4. Compute the prior probabilities: Calculate the prior probabilities of each class by dividing thenumber of training instances in each class by the total number of training instances.
5. Compute the likelihood probabilities: Compute the likelihood probabilities of each attribute given each class by calculating the conditional probabilities of each attribute given each class.
6. Apply Bayes theorem: Apply Bayes theorem to compute the posterior probabilities of each class given the attributes of a test instance. Choose the class with the highest posterior probability as thepredicted class for the test instance.
7. Evaluate the model: Evaluate the performance of the Naïve Bayes classifier on the testing set using metrics such as accuracy, precision, recall, and F1 score.
8. Tune the hyper-parameters: Tune the hyper-parameters of the Naive Bayes classifier to improve its performance on the testing set.
9. Deploy the model: Deploy the Naive Bayes classifier in a real-world application to classify new instances.

Program:

```
import numpy as np

class NaiveBayes:

    def init(self):
        self.prior = None
        self.likelihood = None
        self.classes = None

    def fit(self, X, y):
        self.classes = np.unique(y)
        n_features = X.shape[1]
        n_classes = len(self.classes)
        self.prior = np.zeros(n_classes)
        self.likelihood = np.zeros((n_classes, n_features))
        for (i, c) in enumerate(self.classes):
            X_c = X[y == c]
        self.prior[i] = X_c.shape[0] / X.shape[0]
        self.likelihood[i, :] = (X_c.sum(axis=0) / X_c.sum()).flatten()

    def predict(self, X):
        y_pred = []
        for x in X:
            posterior = []
        for (i, c) in enumerate(self.classes):
            likelihood = np.prod(self.likelihood[i, :] ** x)
            posterior.append(self.prior[i] * likelihood)
        y_pred.append(self.classes[np.argmax(posterior)])
        return y_pred
```

```
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
# Load iris dataset iris = load_iris()
# Split data into train and test sets
(X_train, X_test, y_train, y_test) = train_test_split(iris.data, iris.target, test_size=0.2,
random_state=42)
# Fit Naive Bayes classifier on train data
nb = NaiveBayes()
nb.fit(X_train, y_train)
# Make predictions on test data
y_pred = nb.predict(X_test)
# Evaluate model performance
accuracy = accuracy_score(y_test, y_pred)
print ('Accuracy:', accuracy)
```

APEC

Output:

('Accuracy:', 0.3333333333333333)

APEC

Result:

Thus the program to implement Naïve Bayes models using python is executed and verified successfully.

EX. NO: 4	IMPLEMENTATION OF BAYESIAN NETWORKS

Aim:

To implement the Bayesian Networks.

Algorithm:

1. Import the necessary libraries and classes from pgmpy, which is a Python library for workingwith probabilistic graphical models.
2. Define the structure of our Bayesian network by creating a new BayesianNetwork object and specifying the edges between nodes using a list of tuples. In this case, we have two nodes B andE that each point to a third node A.
3. Create conditional probability tables (CPDs) for each node using the TabularCPD class. The first argument is the name of the node, the second argument is the number of states it can be in (inthis case, 2 for binary variables), and the third argument is the actual table of probabilities.
4. Add the CPDs to the model using the add_cpds() method.
5. Check if the model is valid using the check_model() method.
6. Print the CPDs using the get_cpds() method.
7. Use the Variable elimination class from pgmpy.inference to infer the posterior probabilitydistribution of A given evidence of B=1 and E=0.
8. Print the result.

Program:

```
!pip install pgmpy  # Install pgmpy if not already installed

from pgmpy.models import BayesianNetwork
from pgmpy.factors.discrete import TabularCPD
from pgmpy.inference import VariableElimination

# Define the model structure
model = BayesianNetwork([('B', 'A'), ('E', 'A')])
```

```
# Define the conditional probability distributions
cpd_b = TabularCPD('B', 2, [[0.999], [0.001]])
cpd_e = TabularCPD('E', 2, [[0.998], [0.002]])
cpd_a = TabularCPD('A', 2, [[0.999, 0.71, 0.06, 0.05],
                            [0.001, 0.29, 0.94, 0.95]],
                   evidence=['B', 'E'], evidence_card=[2, 2])

# Add the CPDs to the model
model.add_cpds(cpd_b, cpd_e, cpd_a)

# Check if the model is valid
print("Model is valid?", model.check_model())

# Print the CPDs
print("CPDs:")
print(model.get_cpds())

# Infer the posterior probability distribution of A given evidence
inference = VariableElimination(model)
posterior_a = inference.query(['A'], evidence={'B': 1, 'E': 0})
print("Posterior distribution of A given evidence B=1, E=0:")
print(posterior_a)
```

Output:

[<TabularCPD representing P(B:2) at 0x4d88bb0>, <TabularCPD representing P(E:2) at 0x4d88dc0>, <TabularCPD representing P(A:2 | B:2, E:2) at 0x4d88b50>]

```
+------+----------+
| A    | phi(A) |
+======+==========+
| A(0) |  0.0600 |
+------+----------+
| A(1) |  0.9400 |
+------+----------+
```

APEC

Result:

Thus the program to implement Naïve Bayes models using python is executed and verifiedsuccessfully.

EX. NO: 5	BUILD REGRESSION MODELS

Aim:

To implement regression models using python.

Algorithm:

1. First, the required libraries numpy, pandas, matplotlib.pyplot, sklearn.linear_model.LinearRegression and sklearn.preprocessing.PolynomialFeatures are imported.
2. The program generates sample data for three features (feature_1, feature_2, and feature_3) and a target variable using numpy.random.rand() and adds some random noise to the target variable.
3. A pandas DataFrame is created using the generated sample data, and the DataFrame is saved toa CSV file using to_csv() function.
4. The CSV file is loaded into the program using pandas.read_csv().
5. The features and target variables are defined by extracting the appropriate columns from theDataFrame and assigning them to X and y, respectively.
6. A linear regression model is created using sklearn.linear_model.LinearRegression() and fitted tothe features and target variables using the fit() function.
7. A multiple regression model is created in the same way, but this time including all threefeatures.
8. A polynomial regression model is created using sklearn.preprocessing.PolynomialFeatures() with a degree of 2 to generate additional polynomial features from the original features. The resulting feature matrix is then fitted to the target variable using a linear regression model.
9. The data and three regression models are plotted using matplotlib.pyplot. The scatter plot shows the relationship between feature_1 and target, and the regression lines represent the predictions of the linear, multiple, and polynomial regression models. The plot also includes a legend to differentiate between the different regression models.

Program:

Simple Linear Regression:

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression

# Generate some random data
x = np.array([1, 2, 3, 4, 5]).reshape((-1, 1))
y = np.array([2, 3, 5, 6, 8])

# Create a linear regression object
model = LinearRegression()

# Fit the model to the data
model.fit(x, y)

# Predict the output for a new input
x_new = np.array([[6]])
y_new = model.predict(x_new)
print(y_new)  # Output: [9.4]

# Plot the data and the regression line
plt.scatter(x, y)
plt.plot(x, model.predict(x), color='red')
plt.show()
```

Multiple Linear Regression:

```
import numpy as np
from sklearn.linear_model import LinearRegression

# Generate some random data
```

```
x = np.array([[1, 2, 3], [2, 4, 6], [3, 6, 9], [4, 8, 12]])
y = np.array([6, 12, 18, 24])

# Create a linear regression object
model = LinearRegression()

# Fit the model to the data
model.fit(x, y)

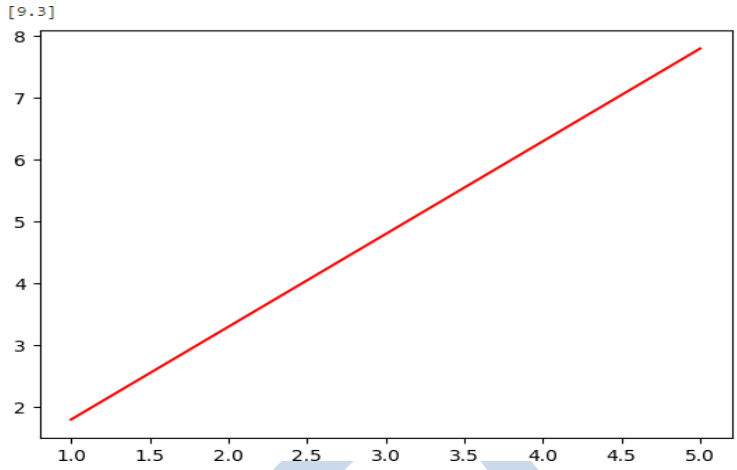
# Predict the output for new inputs
x_new = np.array([[5, 10, 15]])
y_new = model.predict(x_new)
print(y_new)  # Output: [30.]

# Print the coefficients and intercept
print('Coefficients:', model.coef_)
print('Intercept:', model.intercept_)
```

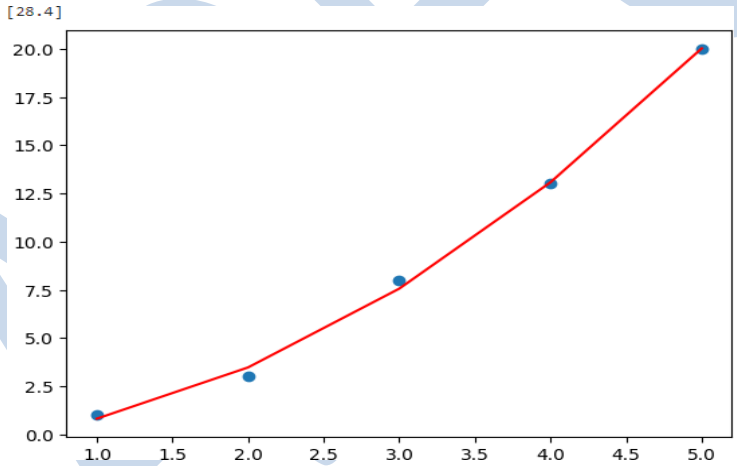
APEC

Output:

Simple Linear Regression



Multiple Linear Regression:



Result:

Thus the program to implement regression models using python is executed and verifiedsuccessfully.

EX. NO: 6	BUILD DECISION TREES AND RANDOM FORESTS

Aim:

To write a program for decision trees and random forests in Python.

Algorithm:

1. Import necessary libraries: pandas, load_iris from sklearn.datasets, train_test_split from sklearn.model_selection, DecisionTreeClassifier and RandomForestClassifier from sklearn.tree,accuracy_score from sklearn.metrics.
2. Load the Iris dataset using load_iris from sklearn.datasets.
3. Convert the dataset into a pandas dataframe and split into features (X) and target (y).
4. Split the data into training and test sets using train_test_split from sklearn.model_selection.
5. Create a decision tree classifier object with max_depth=3 and random_state=42 using DecisionTreeClassifier from sklearn.tree.
6. Fit the decision tree model to the training data using the fit() method.
7. Evaluate the decision tree model on the test data by making predictions using the predict()method, and then calculating the accuracy score using accuracy_score from sklearn.metrics.
8. Create a random forest classifier object with n_estimators=100, max_depth=3, and random_state=42 using RandomForestClassifier from sklearn.ensemble.
9. Fit the random forest model to the training data using the fit() method.
10. Evaluate the random forest model on the test data by making predictions using the predict()method, and then calculating the accuracy score using accuracy_score from sklearn.metrics.
11. Print the accuracy scores of both the decision tree and random forest models.

Program:

```
import pandas as pd
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
```

```
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score

# Load Iris dataset
iris = load_iris()
X = pd.DataFrame(iris.data, columns=iris.feature_names)
y = pd.Series(iris.target)

# Split data into train and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Build decision tree
tree = DecisionTreeClassifier(max_depth=3, random_state=42)
tree.fit(X_train, y_train)

# Evaluate decision tree
tree_pred = tree.predict(X_test)
tree_acc = accuracy_score(y_test, tree_pred)
print('Decision Tree Accuracy:', tree_acc)

# Build random forest
forest = RandomForestClassifier(n_estimators=100, max_depth=3, random_state=42)
forest.fit(X_train, y_train)

# Evaluate random forest
forest_pred = forest.predict(X_test)
forest_acc = accuracy_score(y_test, forest_pred)
print('Random Forest Accuracy:', forest_acc)
```

Output:

('Decision Tree Accuracy:', 1.0)

('Random Forest Accuracy:', 1.0)

APEC

Result:

Thus the program to implement decision trees and random forests models using python isexecuted and verified successfully.

EX. NO: 7	IMPLEMENTATION OF SVM MODEL

Aim:

To implement of SVM model.

Algorithms:

1. Import necessary libraries.
2. Load the Iris dataset.
3. Standardize the data.
4. Split the data into training and testing sets.
5. Perform hyper-parameter tuning using GridSearchCV.
6. Print the best hyper-parameters.
7. Create an SVM model with the best hyper-parameters.
8. Fit the SVM model to the training data.
9. Predict the classes of the test set.
10. Calculate the accuracy score and confusion matrix of the SVM model.

Program:

```
from sklearn import datasets
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.preprocessing import StandardScaler
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score, confusion_matrix

# Load Iris dataset
iris = datasets.load_iris()
X = iris.data
y = iris.target

# Standardize the data
scaler = StandardScaler()
X = scaler.fit_transform(X)
```

```
# Split data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Hyperparameter tuning with Grid Search
param_grid = {'C': [0.1, 1, 10, 100], 'gamma': [0.1, 1, 10, 100], 'kernel': ['linear', 'rbf',
'sigmoid']}
grid = GridSearchCV(SVC(), param_grid, refit=True, verbose=3)
grid.fit(X_train, y_train)

# Best hyperparameters
print("Best hyperparameters: ", grid.best_params_)

# Create SVM model with best hyperparameters
svm = SVC(C=grid.best_params_['C'], gamma=grid.best_params_['gamma'],
kernel=grid.best_params_['kernel'])

# Fit SVM model to training data
svm.fit(X_train, y_train)

# Predict classes of test set
y_pred = svm.predict(X_test)

# Calculate accuracy score and confusion matrix of SVM model
accuracy = accuracy_score(y_test, y_pred)
confusion = confusion_matrix(y_test, y_pred)

print("Accuracy: ", accuracy)
print("Confusion matrix:\n", confusion)
```

Output:

Fitting 3 folds for each of 48 candidates, totalling 144 fits

[CV] kernel=linear, C=0.1, gamma=0.1 .....

[CV] kernel=linear, C=0.1, gamma=0.1, score=0.975609756098, total=0.0s

[CV] kernel=linear, C=0.1, gamma=0.1.....

[CV] ..... kernel=linear, C=0.1, gamma=0.1, score=0.9, total=0.0s[CV] kernel=linear, C=0.1, gamma=0.1 .....

[CV] kernel=linear, C=0.1, gamma=0.1, score=0.974358974359, total=0.0s[CV] kernel=rbf, C=0.1, gamma=0.1 .....

[CV] kernel=rbf, C=100, gamma=100 .....

[CV] kernel=rbf, C=100, gamma=100, score=0.461538461538, total=0.0s

[CV] kernel=sigmoid, C=100, gamma=100 ....

[CV] kernel=sigmoid, C=100, gamma=100, score=0.536585365854, total=0.0s

[CV] kernel=sigmoid, C=100, gamma=100 .....

[CV] .... kernel=sigmoid, C=100, gamma=100, score=0.775, total=0.0s

[CV] kernel=sigmoid, C=100, gamma=100 .....

[CV] kernel=sigmoid, C=100, gamma=100, score=0.564102564103, total=0.0s ('Best hyperparameters: ', {'kernel': 'linear', 'C': 1, 'gamma': 0.1}) ('Accuracy: ', 0.9666666666666667)

('Confusion matrix:\n', array([[10, 0, 0],

[0, 8, 1],

[0, 0, 11]], dtype=int64))

[Parallel(n_jobs=1)]: Done 144 out of 144 | elapsed: 0.2s finished

Result:

Thus the program to implement SVM models using python is executed and verified successfully.

EX. NO: 8	IMPLEMENTATION OF ENSEMBLING TECHNIQUES

Aim:

To Implement Ensembling Techniques using Python.

Algorithms:

1. Import required modules.
2. Generate a random dataset.

   This generates a random dataset with 1000 samples, 10 features, and 2 classes, with a random seed of 42.
3. Split the data into training and testing sets.

   This splits the data into training and testing sets, with a testing set size of 20% and a random seed of 42.
4. Create a Random Forest model.

   This creates a Random Forest model with 100 trees and a random seed of 42, and fits it to the training data.
5. Create a Gradient Boosting model.

   This creates a Gradient Boosting model with 100 trees and a random seed of 42, and fits it to the training data.
6. Predict the classes of the test set using both models.

   This predicts the classes of the test set using both the Random Forest and Gradient Boosting models.
7. Combine the predictions using a majority vote.

   This combines the predictions of both models using a majority vote. For each sample in the test set, it creates a list of predictions from both models, and takes the prediction that occurs most frequently.
8. Calculate the accuracy score of the ensemble model.

   This calculates the accuracy score of the ensemble model by comparing the combined predictions to the true labels of the test set, and prints the result.

Program:

```
from sklearn.datasets import make_classification
from sklearn.ensemble import RandomForestClassifier, GradientBoostingClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

# Generate a random dataset
X, y = make_classification(n_samples=1000, n_features=10, n_classes=2, random_state=42)

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Create a Random Forest model
rfc = RandomForestClassifier(n_estimators=100, random_state=42)
rfc.fit(X_train, y_train)

# Create a Gradient Boosting model
gbc = GradientBoostingClassifier(n_estimators=100, random_state=42)
gbc.fit(X_train, y_train)

# Predict the classes of the test set using both models
y_pred_rfc = rfc.predict(X_test)
y_pred_gbc = gbc.predict(X_test)

# Combine the predictions using a majority vote ensemble
ensemble_preds = []
for i in range(len(X_test)):
    preds = [y_pred_rfc[i], y_pred_gbc[i]]
    ensemble_preds.append(max(set(preds), key=preds.count))

# Calculate the accuracy score of the ensemble model
```

```
ensemble_accuracy = accuracy_score(y_test, ensemble_preds)
print("Ensemble model accuracy:", ensemble_accuracy)
```

APEC

Output:

('Ensemble model accuracy:', 0.89)

APEC

Result:

Thus the program to implement ensembling techniques using python is executed and verified successfully.

EX. NO: 9	IMPLEMENTATION CLUSTERING ALGORITHMS

Aim:

To implement clustering algorithms in Python.

Algorithms:

1. Import the required libraries
2. Generate random data using the make_blobs function from scikit-learn, with 500 samples, 4centers, a standard deviation of 1.0, and a random seed of 0
3. Standardize the data using the StandardScaler function from scikit-learn
4. Define a range of clusters to evaluate, from 2 to 9
5. Initialize empty lists to store the evaluation metrics
6. Loop through the range of clusters, and for each number of clusters
7. Initialize the K-Means model and fit the standardized data
8. Get the cluster labels and centroids from the fitted model
9. Calculate the sum of squared distances (SSE) and silhouette score for the current number ofclusters
10. Visualize the clustering results using a scatter plot of the data points with cluster assignments indicated by color, and the cluster centroids indicated by red crosses
11. After all iterations of the loop, plot the SSE and silhouette scores for each number of clustersusing subplots
12. Display the final plots to the user.

Program:

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans
from sklearn.preprocessing import StandardScaler
from sklearn.datasets import make_blobs
from sklearn.metrics import silhouette_score
```

```
# Generate random data
X, y = make_blobs(n_samples=500, centers=4, cluster_std=1.0, random_state=0)

# Standardize data
scaler = StandardScaler()
X_std = scaler.fit_transform(X)

# Define range of clusters to evaluate
range_clusters = range(2, 10)

# Initialize lists for evaluation metrics
sse = []
silhouette = []

# Evaluate different number of clusters
for k in range_clusters:
    # Initialize K-Means model and fit data
    kmeans = KMeans(n_clusters=k, random_state=0)
    kmeans.fit(X_std)

    # Get cluster labels and centroids
    labels = kmeans.labels_
    centroids = kmeans.cluster_centers_

    # Calculate SSE and Silhouette score
    sse.append(kmeans.inertia_)
    silhouette.append(silhouette_score(X_std, labels))

    # Visualize clustering results
    plt.scatter(X[:, 0], X[:, 1], c=labels)
    plt.scatter(centroids[:, 0], centroids[:, 1], marker='x', s=200, linewidths=3, color='r')
    plt.title("K-Means Clustering (k={})".format(k))
```

```
    plt.show()

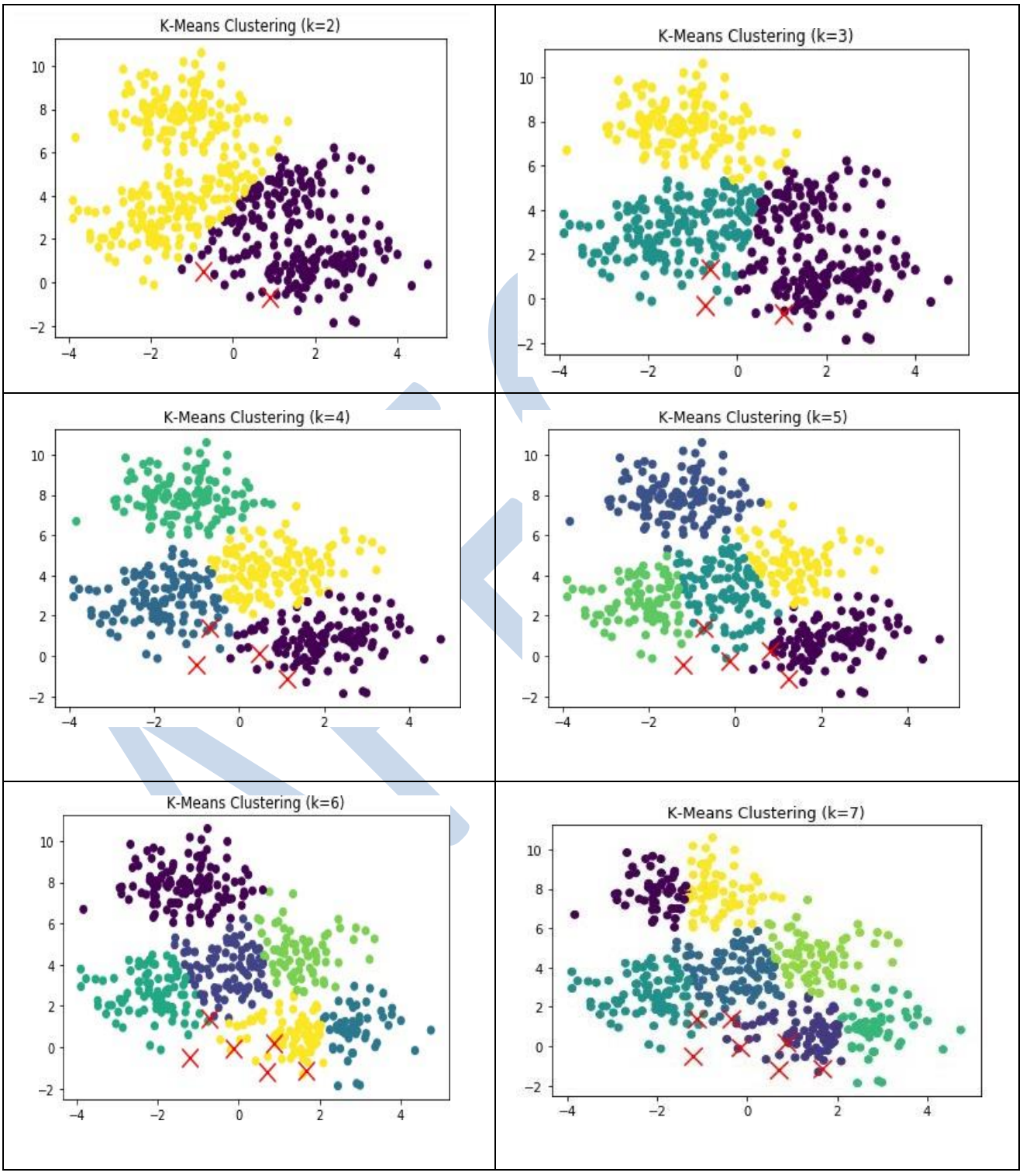
# Plot SSE and Silhouette scores
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 4))
ax1.plot(range_clusters, sse, 'bo-')
ax1.set_xlabel('Number of clusters')
ax1.set_ylabel('SSE')
ax1.set_title('Elbow Method')

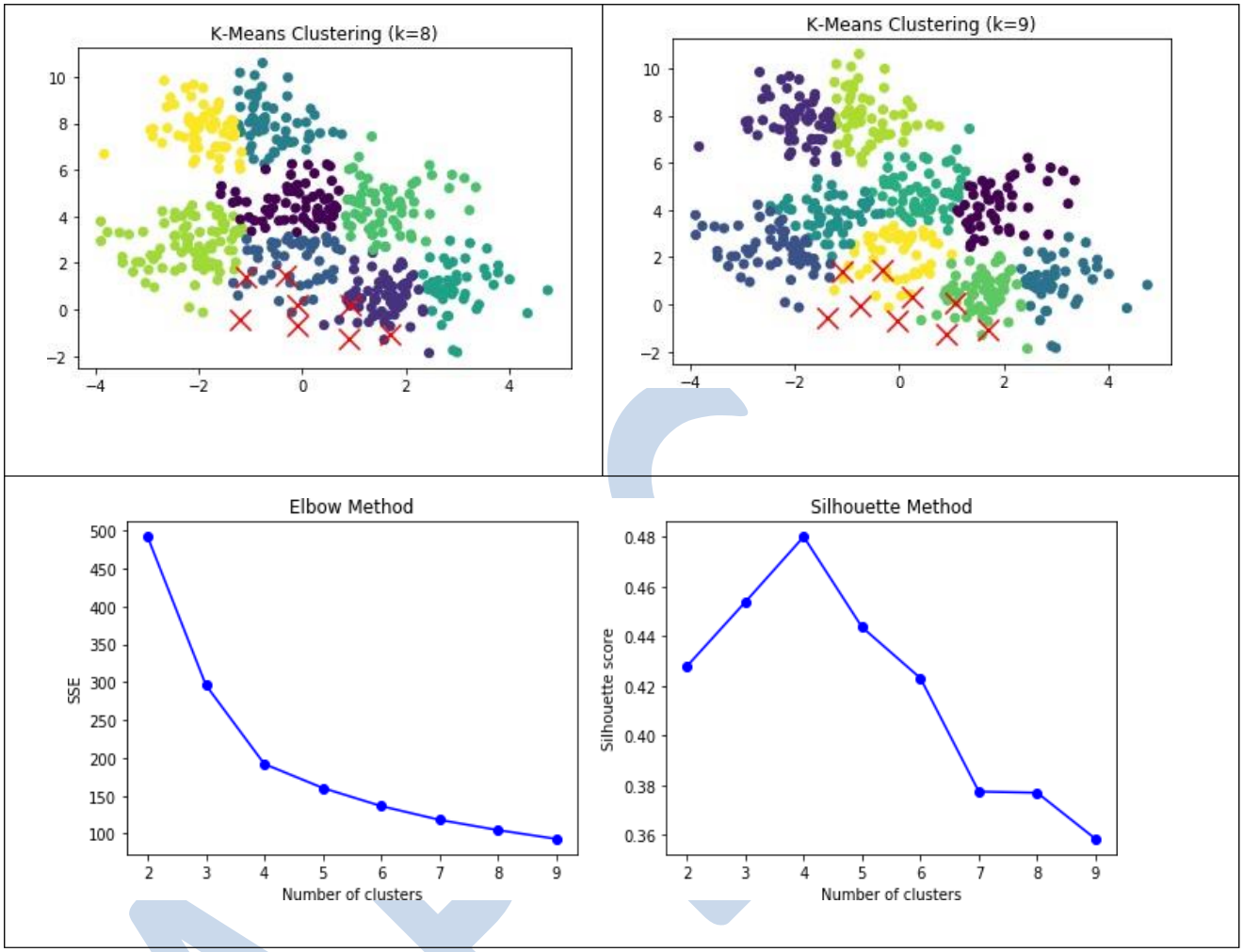
ax2.plot(range_clusters, silhouette, 'bo-')
ax2.set_xlabel('Number of clusters')
ax2.set_ylabel('Silhouette score')
ax2.set_title('Silhouette Method')

plt.tight_layout()
plt.show()
```

APEC

Output:


K-Means Clustering (k=2)
K-Means Clustering (k=3)
K-Means Clustering (k=4)
K-Means Clustering (k=5)
K-Means Clustering (k=6)
K-Means Clustering (k=7)



Result:

Thus the program to implement clustering using python is executed and verified successfully.

EX. NO: 10	IMPLEMENTATION OF EXPECTATION MAXIMIZATION (EM)

Aim:

To write a python code to implement expectation maximization (EM) algorithm.

Algorithm:

1. Import the required packages.
2. Generate and plot the cluster model.
3. Make an initial guess of parameter θ using random function.
4. Given the current estimates for θ, in the expectation step EM computes the cluster posterior probabilities P(Ci |xj) via the Bayes theorem.

$$P(C_i|\mathbf{x}_j) = \frac{f_i(\mathbf{x}_j) \cdot P(C_i)}{\sum_{a=1}^{k} f_a(\mathbf{x}_j) \cdot P(C_a)}$$

5. In the maximization step, using the weights P(Ci |xj) EM re-estimates θ, that is, it re-estimates the parameters for each cluster.
6. Repeat the step 4 & 5 until it converges.

Program:

```
import random
import numpy as np
import matplotlib.pyplot as plt
from numpy.linalg import inv
from scipy.stats import multivariate_normal

# Ground Truth data generation and plotting
m1 = [1, 1]
m2 = [7, 7]
cov1 = [[3, 2], [2, 3]]
cov2 = [[2, -1], [-1, 2]]

x = np.random.multivariate_normal(m1, cov1, size=(200,))
y = np.random.multivariate_normal(m2, cov2, size=(200,))
```

```
d = np.concatenate((x, y), axis=0)

plt.figure(figsize=(10, 10))
plt.scatter(d[:, 0], d[:, 1], marker='o')
plt.axis('equal')
plt.xlabel('X-Axis', fontsize=16)
plt.ylabel('Y-Axis', fontsize=16)
plt.title('Ground Truth', fontsize=22)
plt.grid()
plt.show()

# Initial State plotting
m1 = random.choice(d)
m2 = random.choice(d)
cov1 = np.cov(np.transpose(d))
cov2 = np.cov(np.transpose(d))

pi = 0.5
x1 = np.linspace(-4, 11, 200)
x2 = np.linspace(-4, 11, 200)
X, Y = np.meshgrid(x1, x2)
Z1 = multivariate_normal(m1, cov1)
Z2 = multivariate_normal(m2, cov2)

pos = np.empty(X.shape + (2,))
pos[:, :, 0] = X
pos[:, :, 1] = Y

plt.figure(figsize=(10, 10))
plt.scatter(d[:, 0], d[:, 1], marker='o')
plt.contour(X, Y, Z1.pdf(pos), colors="r", alpha=0.5)
plt.contour(X, Y, Z2.pdf(pos), colors="b", alpha=0.5)
```

```
plt.axis('equal')
plt.xlabel('X-Axis', fontsize=16)
plt.ylabel('Y-Axis', fontsize=16)
plt.title('Initial State', fontsize=22)
plt.grid()
plt.show()

# Expectation step function
def Estep(lis1):
    m1 = lis1[0]
    m2 = lis1[1]
    cov1 = lis1[2]
    cov2 = lis1[3]
    pi = lis1[4]

    pt2 = multivariate_normal.pdf(d, mean=m2, cov=cov2)
    pt1 = multivariate_normal.pdf(d, mean=m1, cov=cov1)
    w1 = pi * pt2
    w2 = (1 - pi) * pt1
    eval1 = w1 / (w1 + w2)

    return eval1

# Maximization step function
def Mstep(eval1):
    num_mu1, din_mu1, num_mu2, din_mu2 = 0, 0, 0, 0

    for i in range(len(d)):
        num_mu1 += (1 - eval1[i]) * d[i]
        din_mu1 += (1 - eval1[i])
        num_mu2 += eval1[i] * d[i]
        din_mu2 += eval1[i]
```

```
    mu1 = num_mu1 / din_mu1
    mu2 = num_mu2 / din_mu2

    num_s1, din_s1, num_s2, din_s2 = 0, 0, 0, 0
    for i in range(len(d)):
        q1 = np.matrix(d[i] - mu1)
        num_s1 += (1 - eval1[i]) * np.dot(q1.T, q1)
        din_s1 += (1 - eval1[i])

        q2 = np.matrix(d[i] - mu2)
        num_s2 += eval1[i] * np.dot(q2.T, q2)
        din_s2 += eval1[i]

    s1 = num_s1 / din_s1
    s2 = num_s2 / din_s2

    pi = sum(eval1) / len(d)

    lis2 = [mu1, mu2, s1, s2, pi]
    return lis2

# Plot function
def plot(lis1):
    mu1 = lis1[0]
    mu2 = lis1[1]
    s1 = lis1[2]
    s2 = lis1[3]

    Z1 = multivariate_normal(mu1, s1)
    Z2 = multivariate_normal(mu2, s2)
```

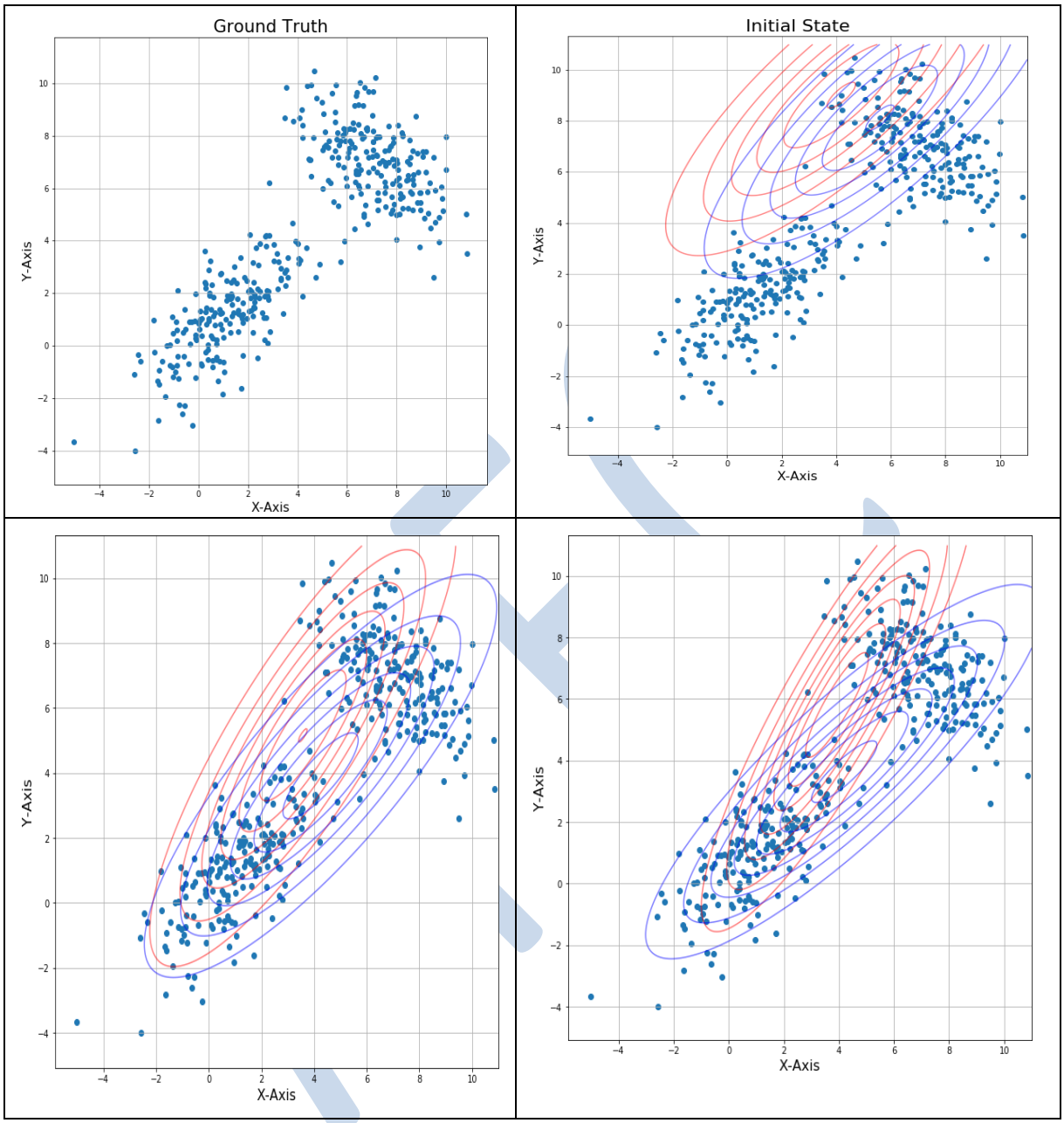
```
    pos = np.empty(X.shape + (2,))
    pos[:, :, 0] = X
    pos[:, :, 1] = Y

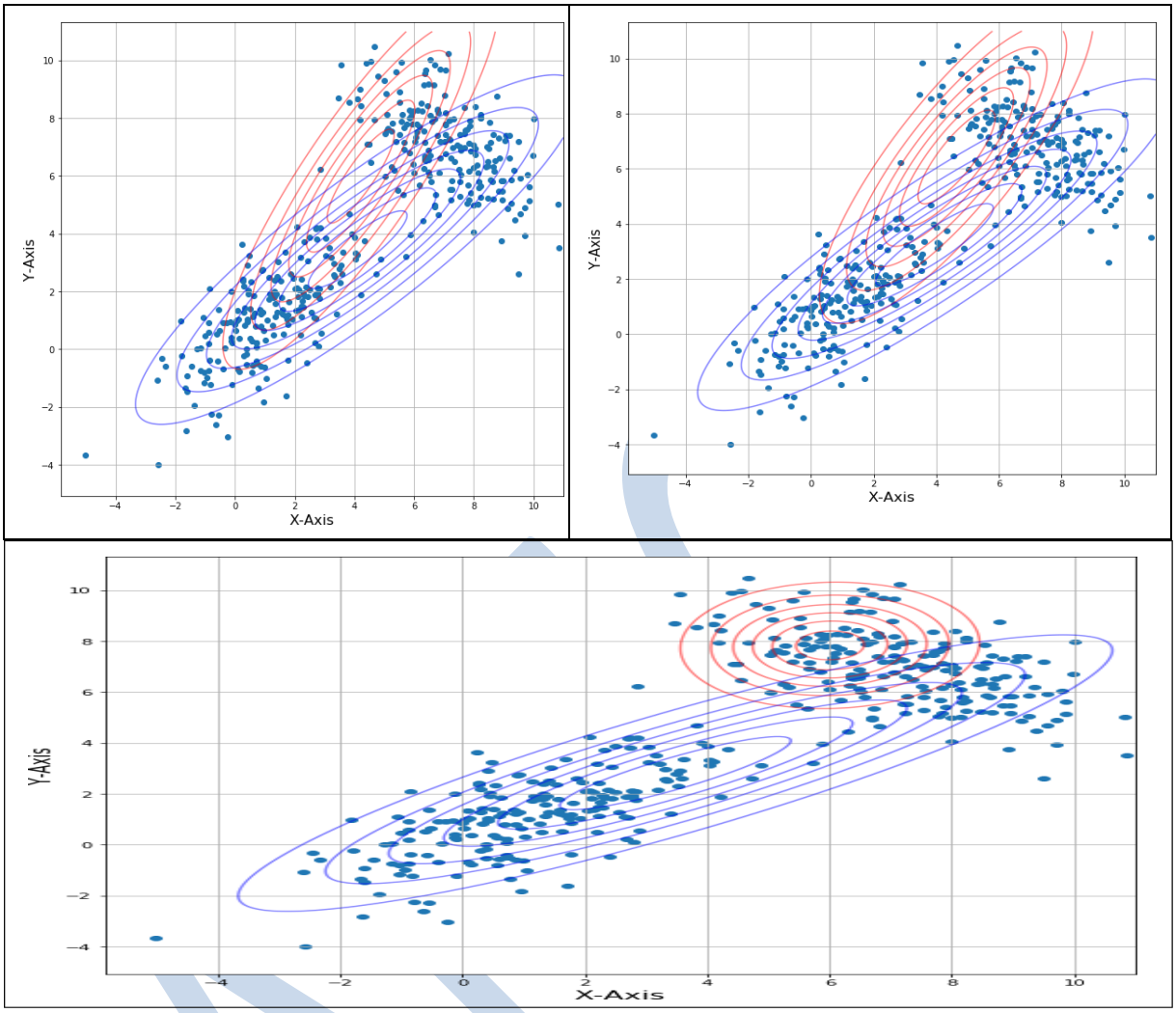
    plt.figure(figsize=(10, 10))
    plt.scatter(d[:, 0], d[:, 1], marker='o')
    plt.contour(X, Y, Z1.pdf(pos), colors="r", alpha=0.5)
    plt.contour(X, Y, Z2.pdf(pos), colors="b", alpha=0.5)
    plt.axis('equal')
    plt.xlabel('X-Axis', fontsize=16)
    plt.ylabel('Y-Axis', fontsize=16)
    plt.grid()
    plt.show()

# Iterative process
iterations = 20
lis1 = [m1, m2, cov1, cov2, pi]
for i in range(iterations):
    lis2 = Mstep(Estep(lis1))
    lis1 = lis2

    if i in [0, 4, 9, 14, 19]:
        plot(lis1)
```

Output:





Result:

Thus the python program for expectation maximization on clustering is written and executed successfully.

EX. NO: 11	SIMPLE NEURAL NETWORK

Aim:

To write a python program to build simple neural networks.

Algorithm:

1. Import the libraries. For example: import numpy as np
2. Define/create input data. For example, use numpy to create a dataset and an array of datavalues.
3. Add weights and bias (if applicable) to input features. These are learnable parameters, meaning that they can be adjusted during training.
    - Weights = input parameters that influences output
    - Bias = an extra threshold value added to the output
4. Train the network against known, good data in order to find the correct values for the weights and biases.
5. Test the Network against a set of test data to see how it performs.

Program:

```
import numpy as np

class NeuralNetwork:
    def __init__(self):
        # seeding for random number generation
        np.random.seed(1)

        # converting weights to a 3 by 1 matrix with values from -1 to 1 and mean of 0
        self.synaptic_weights = 2 * np.random.random((3, 1)) - 1

    def sigmoid(self, x):
        # applying the sigmoid function
        return 1 / (1 + np.exp(-x))
```

```
    def sigmoid_derivative(self, x):
        # computing derivative of the Sigmoid function
        return x * (1 - x)

    def train(self, training_inputs, training_outputs, training_iterations):
        # training the model to make accurate predictions while adjusting weights continually
        for iteration in range(training_iterations):
            # siphon the training data via the neuron
            output = self.think(training_inputs)

            # computing error rate for back-propagation
            error = training_outputs - output

            # performing weight adjustments
            adjustments = np.dot(training_inputs.T, error * self.sigmoid_derivative(output))
            self.synaptic_weights += adjustments

    def think(self, inputs):
        # passing the inputs via the neuron to get output
        # converting values to floats
        inputs = inputs.astype(float)
        output = self.sigmoid(np.dot(inputs, self.synaptic_weights))
        return output

if __name__ == "__main__":
    # initializing the neural network class
    neural_network = NeuralNetwork()

    print("Beginning Randomly Generated Weights:")
    print(neural_network.synaptic_weights)

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

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

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

# training taking place
neural_network.train(training_inputs, training_outputs, 15000)

print("Ending Weights After Training:")
print(neural_network.synaptic_weights)

# User input to test the trained model
user_input_one = int(input("User Input One: "))
user_input_two = int(input("User Input Two: "))
user_input_three = int(input("User Input Three: "))

print("\nConsidering New Situation: ", user_input_one, user_input_two, user_input_three)
print("New Output data:")
print(neural_network.think(np.array([user_input_one, user_input_two, user_input_three])))
print("Wow, we did it!")
```

Output:

Beginning Randomly Generated Weights:

[[-0.16595599]

[0.44064899]

[-0.99977125]]

Ending Weights After Training:

[[10.08740896]

[-0.20695366]

[-4.83757835]]

User Input One: 1 User Input Two: 2 User Input Three: 3

('Considering New Situation: ', '1', '2', '3')New Output data:

[0.00785099]

Wow, we did it!

Result:

Thus the python program for building simple neural network model is executed successfully.

EX. NO: 12	DEEP LEARNING NEURAL NETWORK

Aim:

To write a python program to build deep learning neural network models.

Algorithm:

1. Import necessary libraries and load the data
    - Use the NumPy library to load your dataset and two classes from the Keras libraryto define the model
    - Use the Pima Indians onset of diabetes dataset
2. Create a Sequential model and add layers one at a time until satisfied with the architecture.The model expects rows of data with 8 variables (the *input_shape=(8,)* argument).
    - The first hidden layer has 12 nodes and uses the relu activation function.
    - The second hidden layer has 8 nodes and uses the relu activation function.
    - The output layer has one node and uses the sigmoid activation function.
3. Compile the model by specifying the loss function to use to evaluate a set of weights, the optimizer used to search through different weights for the network and metrics for reporting the training.
4. Train or fit the model on loaded data by calling the fit() function on the model.
5. Evaluate the model on the training dataset using the evaluate() function and pass it the same input and output used to train the model.

Program:

```
from numpy import loadtxt
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from google.colab import files

# Load the dataset
uploaded = files.upload()
```

```
dataset = loadtxt('diabetes.csv', delimiter=',', dtype=float, skiprows=1)
dataset[:, 0] = dataset[:, 0].astype(float)

# Split into input (X) and output (y) variables
X = dataset[:, 0:8]
y = dataset[:, 8]

# Define the keras model
model = Sequential()
model.add(Dense(12, input_shape=(8,), activation='relu'))
model.add(Dense(8, activation='relu'))
model.add(Dense(1, activation='sigmoid'))

# Compile the keras model
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])

# Fit the keras model on the dataset
model.fit(X, y, epochs=150, batch_size=10, verbose=0)

# Make class predictions with the model
predictions = (model.predict(X) > 0.5).astype(int)

# Summarize the first 5 cases
for i in range(5):
    print('%s => %d (expected %d)' % (X[i].tolist(), predictions[i], y[i]))
```

Output:

Saving diabetes.csv to diabetes.csv

24/24 [==============================] - 0s 1ms/step

[6.0, 148.0, 72.0, 35.0, 0.0, 33.6, 0.627, 50.0] => 1 (expected 1)

[1.0, 85.0, 66.0, 29.0, 0.0, 26.6, 0.351, 31.0] => 0 (expected 0)

[8.0, 183.0, 64.0, 0.0, 0.0, 23.3, 0.672, 32.0] => 1 (expected 1)

[1.0, 89.0, 66.0, 23.0, 94.0, 28.1, 0.167, 21.0] => 0 (expected 0)

[0.0, 137.0, 40.0, 35.0, 168.0, 43.1, 2.288, 33.0] => 1 (expected 1)

APEC

Result:

Thus the python program for building deep learning neural network model is executedsuccessfully.