PROGRAM – 1
Implement A* Search algorithm.

```
def aStarAlgo(start_node, stop_node):
    open_set = set(start_node)
    closed_set = set()
    g = {}
    parents = {}
    g[start_node] = 0
    parents[start_node] = start_node
    while len(open_set) > 0:
        n = None
        for v in open_set:
            if n == None or g[v] + heuristic(v) < g[n] + heuristic(n):
                n = v
        if n == stop_node or Graph_nodes[n] == None:
            pass
        else:
            for (m, weight) in get_neighbors(n):
                if m not in open_set and m not in closed_set:
                    open_set.add(m)
                    parents[m] = n
                    g[m] = g[n] + weight
                else:
                    if g[m] > g[n] + weight:
                        g[m] = g[n] + weight
                        parents[m] = n
                        if m in closed_set:
                            closed_set.remove(m)
                            open_set.add(m)
        if n == None:
            print('Path does not exist!')
            return None
        if n == stop_node:
            path = []
            while parents[n] != n:
                path.append(n)
                n = parents[n]
            path.append(start_node)
            path.reverse()
            print('Path found: {}'.format(path))
            return path
        open_set.remove(n)
        closed_set.add(n)
    print('Path does not exist!')
    return None


def get_neighbors(v):
    if v in Graph_nodes:
```

```
        return Graph_nodes[v]
    else:
        return None


def heuristic(n):
    H_dist = {
        'A': 11,
        'B': 6,
        'C': 99,
        'D': 1,
        'E': 7,
        'G': 0,
    }
    return H_dist[n]


Graph_nodes = {
    'A': [('B', 2), ('E', 3)],
    'B': [('C', 1), ('G', 9)],
    'C': None,
    'E': [('D', 6)],
    'D': [('G', 1)],
}
aStarAlgo('A', 'G')
```

PROGRAM – 2
Implement AO* Search algorithm.

```
class Graph:
    def __init__(self, graph, heuristicNodeList, startNode):
        self.graph = graph
        self.H = heuristicNodeList
        self.start = startNode
        self.parent = {}
        self.status = {}
        self.solutionGraph = {}

    def applyAOStar(self):
        self.aoStar(self.start, False)

    def getNeighbors(self, v):
        return self.graph.get(v, '')

    def getStatus(self, v):
        return self.status.get(v, 0)

    def setStatus(self, v, val):
        self.status[v] = val
```

```
    def getHeuristicNodeValue(self, n):
        return self.H.get(n, 0)

    def setHeuristicNodeValue(self, n, value):
        self.H[n] = value

    def printSolution(self):
        print("FOR GRAPH SOLUTION, TRAVERSE THE GRAPH FROM THE
STARTNODE:", self.start)
        print("------------------------------------------------------------")
        print(self.solutionGraph)
        print("------------------------------------------------------------")

    def computeMinimumCostChildNodes(self, v):
        minimumCost = 0
        costToChildNodeListDict = {}
        costToChildNodeListDict[minimumCost] = []
        flag = True
        for nodeInfoTupleList in self.getNeighbors(v):
            cost = 0
            nodeList = []
            for c, weight in nodeInfoTupleList:
                cost = cost + self.getHeuristicNodeValue(c) + weight
                nodeList.append(c)

            if flag == True:
                minimumCost = cost
                costToChildNodeListDict[minimumCost] = nodeList
                flag = False
            else:
                if minimumCost > cost:
                    minimumCost = cost
                    costToChildNodeListDict[minimumCost] = nodeList

        return minimumCost, costToChildNodeListDict[minimumCost]

    def aoStar(self, v, backTracking):

        print("HEURISTIC VALUES :", self.H)
        print("SOLUTION GRAPH :", self.solutionGraph)
        print("PROCESSING NODE :", v)

        print("-----------------------------------------------------------------------------------------")

        if self.getStatus(v) >= 0:
            minimumCost, childNodeList = self.computeMinimumCostChildNodes(v)
            self.setHeuristicNodeValue(v, minimumCost)
            self.setStatus(v, len(childNodeList))
```

```
            solved = True

            for childNode in childNodeList:
                self.parent[childNode] = v
                if self.getStatus(childNode) != -1:
                    solved = solved & False

            if solved == True:
                self.setStatus(v, -1)
                self.solutionGraph[v] = childNodeList

            if v != self.start:
                self.aoStar(self.parent[v], True)

            if backTracking == False:
                for childNode in childNodeList:
                    self.setStatus(childNode, 0)
                    self.aoStar(childNode, False)


h1 = {'A': 1, 'B': 6, 'C': 2, 'D': 12, 'E': 2, 'F': 1, 'G': 5, 'H': 7, 'I': 7, 'J': 1}
graph1 = {
    'A': [[('B', 1), ('C', 1)], [('D', 1)]],
    'B': [[('G', 1)], [('H', 1)]],
    'C': [[('J', 1)]],
    'D': [[('E', 1), ('F', 1)]],
    'G': [[('I', 1)]]
}
G1 = Graph(graph1, h1, 'A')
G1.applyAOStar()
G1.printSolution()
```

PROGRAM – 3

For a given set of training data examples stored in a .CSV file, implement and demonstrate the Candidate-Elimination algorithm to output a description of the set of all hypotheses consistent with the training examples.

3.csv

```
sky,airtemp,humidity,wind,water,forcast,enjoysport
Sunny,Warm,Normal,Strong,Warm,Same,Yes
Sunny,Warm,High,Strong,Warm,Same,Yes
Rainy,Cold,High,Strong,Warm,Change,No
Sunny,Warm,High,Strong,Cool,Change,Yes
```

```
import csv
```

```
with open("3.csv") as f:
    csv_file = csv.reader(f)
    data = list(csv_file)

    specific = data[1][:-1]
    general = [['?' for i in range(len(specific))] for j in range(len(specific))]

    for i in data:
        if i[-1] == "Yes":
            for j in range(len(specific)):
                if i[j] != specific[j]:
                    specific[j] = "?"
                    general[j][j] = "?"

        elif i[-1] == "No":
            for j in range(len(specific)):
                if i[j] != specific[j]:
                    general[j][j] = specific[j]
                else:
                    general[j][j] = "?"

        print("\nStep " + str(data.index(i) + 1) + " of Candidate Elimination Algorithm")
        print(specific)
        print(general)

    gh = []
    for i in general:
        for j in i:
            if j != '?':
                gh.append(i)
                break
    print("\nFinal Specific hypothesis:\n", specific)
    print("\nFinal General hypothesis:\n", gh)
```

PROGRAM – 4

Write a program to demonstrate the working of the decision tree-based ID3 algorithm. Use an appropriate data set for building the decision tree and apply this knowledge to classify a new sample.

4.csv

```
Outlook,Temperature,Humidity,Wind,PlayTennis
Sunny,Hot,High,Weak,No
Sunny,Hot,High,Strong,No
Overcast,Hot,High,Weak,Yes
Rain,Mild,High,Weak,Yes
Rain,Cool,Normal,Weak,Yes
Rain,Cool,Normal,Strong,No
Overcast,Cool,Normal,Strong,Yes
```

```
Sunny,Mild,High,Weak,No
Sunny,Cool,Normal,Weak,Yes
Rain,Mild,Normal,Weak,Yes
Sunny,Mild,Normal,Strong,Yes
Overcast,Mild,High,Strong,Yes
Overcast,Hot,Normal,Weak,Yes
Rain,Mild,High,Strong,No
```

```python
import pandas as pd
import numpy as np

dataset = pd.read_csv('4.csv', names=['outlook', 'temperature', 'humidity', 'wind', 'class', ])


def entropy(target_col):
    elements, counts = np.unique(target_col, return_counts=True)
    entropy = np.sum([(-counts[i] / np.sum(counts)) * np.log2(counts[i] / np.sum(counts))
                      for i in range(len(elements))])
    return entropy


def InfoGain(data, split_attribute_name, target_name="class"):
    total_entropy = entropy(data[target_name])
    vals, counts = np.unique(data[split_attribute_name], return_counts=True)
    Weighted_Entropy = np.sum(
        [(counts[i] / np.sum(counts)) *
         entropy(data.where(data[split_attribute_name] == vals[i]).dropna()[target_name])
         for i in range(len(vals))])
    Information_Gain = total_entropy - Weighted_Entropy
    return Information_Gain


def ID3(data, originaldata, features, target_attribute_name="class",
parent_node_class=None):
    if len(np.unique(data[target_attribute_name])) <= 1:
        return np.unique(data[target_attribute_name])[0]
    elif len(data) == 0:
        return np.unique(originaldata[target_attribute_name])[
            np.argmax(np.unique(originaldata[target_attribute_name], return_counts=True)[1])]
    elif len(features) == 0:
        return parent_node_class
    else:
        parent_node_class = np.unique(data[target_attribute_name])[
            np.argmax(np.unique(data[target_attribute_name], return_counts=True)[1])]

    item_values = [InfoGain(data, feature, target_attribute_name) for feature in features]
    best_feature_index = np.argmax(item_values)
    best_feature = features[best_feature_index]
    tree = {best_feature: {}}
```

```
    features = [i for i in features if i != best_feature]
    for value in np.unique(data[best_feature]):
        value = value
        sub_data = data.where(data[best_feature] == value).dropna()
        subtree = ID3(sub_data, dataset, features, target_attribute_name, parent_node_class)
        tree[best_feature][value] = subtree
    return tree


def print_tree(tree, indent=""):
    if type(tree) == dict:
        for key, value in tree.items():
            print(indent + key)
            print_tree(value, indent + "  ")
    else:
        print(indent + "  class: " + str(tree))


tree = ID3(dataset, dataset, dataset.columns[:-1])

print('\nDisplay Tree:')
print_tree(tree)
```

PROGRAM – 5
Build an Artificial Neural Network by implementing the Backpropagation algorithm and test the same using appropriate data sets.

```
import numpy as np

X = np.array(([2, 9], [1, 5], [3, 6]), dtype=float)
y = np.array(([92], [86], [89]), dtype=float)
X = X / np.amax(X, axis=0)
y = y / 100


def sigmoid(x):
    return 1 / (1 + np.exp(-x))


def derivatives_sigmoid(x):
    return x * (1 - x)


num_epochs = 5
learning_rate = 0.1
inputlayer_neurons = 2
hiddenlayer_neurons = 3
```

```
output_neurons = 1

weights_hidden = np.random.uniform(size=(inputlayer_neurons, hiddenlayer_neurons))
biases_hidden = np.random.uniform(size=(1, hiddenlayer_neurons))
weights_output = np.random.uniform(size=(hiddenlayer_neurons, output_neurons))
biases_output = np.random.uniform(size=(1, output_neurons))

for epoch in range(num_epochs):
    hidden_layer_input = np.dot(X, weights_hidden) + biases_hidden
    hidden_layer_activation = sigmoid(hidden_layer_input)
    output_layer_input = np.dot(hidden_layer_activation, weights_output) + biases_output
    output = sigmoid(output_layer_input)
    output_error = y - output
    output_delta = output_error * derivatives_sigmoid(output)
    hidden_layer_error = output_delta.dot(weights_output.T)
    hidden_layer_delta = hidden_layer_error * derivatives_sigmoid(hidden_layer_activation)
    weights_output += hidden_layer_activation.T.dot(output_delta) * learning_rate
    weights_hidden += X.T.dot(hidden_layer_delta) * learning_rate
    print("Predicted Output: \n", output)

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

PROGRAM – 6

Write a program to implement the naïve Bayesian classifier for a sample training data set stored as a .CSV file. Compute the accuracy of the classifier, considering few test data sets.

```
import csv
import random
import math


def mean(numbers):
    return sum(numbers) / float(len(numbers))


def stdev(numbers):
    avg = mean(numbers)
    variance = sum([pow(x - avg, 2) for x in numbers]) / float(len(numbers) - 1)
    return math.sqrt(variance)


def split_dataset(dataset, split_ratio=0.67):
    train_size = int(len(dataset) * split_ratio)
    train_set = random.sample(dataset, train_size)
    test_set = [data for data in dataset if data not in train_set]
```

```
    return train_set, test_set


def separate_by_class(dataset):
    separated = {}
    for vector in dataset:
        class_value = vector[-1]
        if class_value not in separated:
            separated[class_value] = []
        separated[class_value].append(vector)
    return separated


def summarize_dataset(dataset):
    summaries = [(mean(attribute), stdev(attribute)) for attribute in zip(*dataset)]
    del summaries[-1]
    return summaries


def summarize_by_class(dataset):
    separated = separate_by_class(dataset)
    summaries = {}
    for class_value, instances in separated.items():
        summaries[class_value] = summarize_dataset(instances)
    return summaries


def calculate_probability(x, mean, stdev):
    exponent = math.exp(-(math.pow(x - mean, 2) / (2 * math.pow(stdev, 2))))
    return (1 / (math.sqrt(2 * math.pi) * stdev)) * exponent


def calculate_class_probabilities(summaries, input_vector):
    probabilities = {}
    for class_value, class_summaries in summaries.items():
        probabilities[class_value] = 1
        for i in range(len(class_summaries)):
            mean, stdev = class_summaries[i]
            x = input_vector[i]
            probabilities[class_value] *= calculate_probability(x, mean, stdev)
    return probabilities


def predict(summaries, input_vector):
    probabilities = calculate_class_probabilities(summaries, input_vector)
    best_label = max(probabilities, key=probabilities.get)
    return best_label


def get_predictions(summaries, test_set):
```

```
    predictions = [predict(summaries, data) for data in test_set]
    return predictions


def get_accuracy(test_set, predictions):
    correct = sum(1 for i in range(len(test_set)) if test_set[i][-1] == predictions[i])
    return (correct / float(len(test_set))) * 100.0


with open('6.csv') as f:
    data = [list(map(float, row)) for row in csv.reader(f)]
    training_set, test_set = split_dataset(data)
    print(f'Split {len(data)} rows into train={len(training_set)} and test={len(test_set)} rows')
    summaries = summarize_by_class(training_set)
    predictions = get_predictions(summaries, test_set)
    accuracy = get_accuracy(test_set, predictions)
    print(f'Accuracy of the classifier is: {accuracy}%')
```

PROGRAM – 7

Apply EM algorithm to cluster a set of data stored in a .CSV file. Use the same data set for clustering using the k-Means algorithm. Compare the results of these two algorithms and comment on the quality of clustering. You can add Java/Python ML library classes/API in the program.

```
from sklearn.cluster import KMeans
from sklearn.mixture import GaussianMixture
import sklearn.metrics as metrics
import pandas as pd
from sklearn import datasets
import numpy as np
import matplotlib.pyplot as plt

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

plt.figure(figsize=(14, 7))
colormap = np.array(['red', 'lime', 'black'])

# REAL PLOT
plt.subplot(1, 3, 1)
plt.title('Real')
plt.scatter(X[:, 2], X[:, 3], c=colormap[y])

# K-PLOT
model = KMeans(n_clusters=3, random_state=0).fit(X)
plt.subplot(1, 3, 2)
```

```
plt.title('KMeans')
plt.scatter(X[:, 2], X[:, 3], c=colormap[model.labels_])

print('The accuracy score of K-Mean: ', metrics.accuracy_score(y, model.labels_))
print('The Confusion matrix of K-Mean:\n', metrics.confusion_matrix(y, model.labels_))

# GMM PLOT
gmm = GaussianMixture(n_components=3, random_state=0).fit(X)
y_cluster_gmm = gmm.predict(X)
plt.subplot(1, 3, 3)
plt.title('GMM Classification')
plt.scatter(X[:, 2], X[:, 3], c=colormap[y_cluster_gmm])
plt.show()

print('The accuracy score of EM: ', metrics.accuracy_score(y, y_cluster_gmm))
print('The Confusion matrix of EM:\n', metrics.confusion_matrix(y, y_cluster_gmm))
```

PROGRAM – 8
Write a program to implement the k-Nearest Neighbour algorithm to classify the iris data set. Print both correct and wrong predictions. Java/Python ML library classes can be used for this problem.

```
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier
from sklearn import datasets
from sklearn.metrics import classification_report, confusion_matrix


iris = datasets.load_iris()
print("Iris data set loaded...\n")


x_train, x_test, y_train, y_test = train_test_split(iris.data, iris.target, test_size=0.1)
print("data set split into training and testing...")
print("size of training data and its label", x_train.shape, y_train.shape, "\n")
print("size of training data and its label", x_test.shape, y_test.shape, "\n")


for i in range(len(iris.target_names)):
    print("label", i, "-", str(iris.target_names[i]))


classifier = KNeighborsClassifier(n_neighbors=1)


classifier.fit(x_train, y_train)
y_pred = classifier.predict(x_test)
```

```
print("results of classification using k-nn with k=1")
for r in range(0, len(x_test)):
    print("sample: ", str(x_test[r]), "actual-label:", str(y_test[r]), "predicted label:",
str(y_pred[r]), "\n")

print("classification accuracy:", classifier.score(x_test, y_test), "\n")

print("confusion matrix")
print(confusion_matrix(y_test, y_pred))
print("Accuracy matrix")
print(classification_report(y_test, y_pred))
```

PROGRAM – 9

Implement the non-parametric Locally Weighted Regression algorithm in order to fit data points. Select the appropriate data set for your experiment and draw graphs.

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


def kernel(point, xmat, k):
    m, n = np.shape(xmat)
    weights = np.mat(np.eye((m)))
    for j in range(m):
        diff = point - X[j]
        weights[j, j] = np.exp(diff * diff.T / (-2.0 * k ** 2))
    return weights


def localWeight(point, xmat, ymat, k):
    wei = kernel(point, xmat, k)
    W = (X.T * (wei * X)).I * (X.T * (wei * ymat.T))
    return W


def localWeightRegression(xmat, ymat, k):
    m, n = np.shape(xmat)
    ypred = np.zeros(m)
    for i in range(m):
        ypred[i] = xmat[i] * localWeight(xmat[i], xmat, ymat, k)
    return ypred
```

```
data = pd.read_csv('9.csv')
bill = np.array(data.total_bill)
tip = np.array(data.tip)

mbill = np.mat(bill)
mtip = np.mat(tip)
m = np.shape(mbill)[1]
one = np.mat(np.ones(m))
X = np.hstack((one.T, mbill.T))

ypred = localWeightRegression(X, mtip, 2)
SortIndex = X[:, 1].argsort(0)
xsort = X[SortIndex][:, 0]

fig = plt.figure()
ax = fig.add_subplot(1, 1, 1)
ax.scatter(bill, tip, color='blue')
ax.plot(xsort[:, 1], ypred[SortIndex], color='red', linewidth=1)
plt.xlabel('Total bill')
plt.ylabel('Tip')
plt.show()
```