Program 1

```
def aStar(start, stop):
    open = set(start)
    close = set()
    dist = {}
    parents = {}
    dist[start] = 0
    parents[start] = start
    while len(open) > 0:
        u = None
        for v in open:
            if u == None or dist[v] + h[v] < dist[u] + h[u]:
                u = v
        if u != stop and neighbors(u) != None:
            for (v, d) in neighbors(u):
                if v in open or v in close:
                    if dist[u] + d < dist[v]:
                        dist[v] = dist[u] + d
                        parents[v] = u
                    if v in close:
                        close.remove(v)
                        open.add(v)
                else:
                    open.add(v)
                    dist[v] = dist[u] + d
                    parents[v] = u
        if u == stop:
            path = []
            while parents[u] != u:
                path.append(u)
                u = parents[u]
            path.append(start)
            path.reverse()
            return 'Path found: {}'.format(path)
        open.remove(u)
        close.add(u)
    return 'Path does not exist!'

def neighbors(u):
    if u in nodes:
        return nodes[u]
    else:
        return None

h = {'A': 11, 'B': 6, 'C': 5, 'D': 7, 'E': 3, 'F': 6, 'G': 5, 'H': 3, 'I': 1, 'J':
0}

nodes = {
    'A': [('B', 6)],
    'B': [('C', 3), ('D', 2)],
    'C': [('D', 1), ('E', 5)],
    'D': [('E', 8)],
    'E': [('I', 5)],
    'F': [('G', 1), ('H', 7)],
    'G': [('I', 3)],
    'H': [('I', 2)],
    'I': [('J', 3)],
}
```

```
print(aStar('A', 'J'))
```

Program 2

```
parent = {}
status = {}
solution = {}

def compute(v):
    min = 0
    child = []
    flag = True
    for i in nodes.get(v, ''):
        cost = 0
        temp = []
        for (c, d) in i:
            temp.append(c)
            cost += h.get(c, 0) + d
        if flag == True or flag == False and min > cost:
            min = cost
            child = temp
            flag = False
    return min, child

def aoStar(v, flag):
    print('Heuristic values :', h)
    print('Solution graph :', solution)
    print('Processing node :', v)
    print('')
    if status.get(v, 0) != -1:
        h[v], child = compute(v)
        status[v] = 0
        solved = True
        for i in child:
            parent[i] = v
            if status.get(i, 0) != -1:
                solved = False
        if solved == True:
            status[v] = -1
            solution[v] = child
        if v != start:
            aoStar(parent[v], True)
        if flag == False:
            for i in child:
                status[i] = 0
                aoStar(i, False)

h = {'A': 1, 'B': 6, 'C': 2, 'D': 12, 'E': 2, 'F': 1, 'G': 5, 'H': 7, 'I': 7, 'J':
1, 'T': 3}

nodes = {
    'A': [[('B', 1), ('C', 1)], [('D', 1)]],
    'B': [[('G', 1)], [('H', 1)]],
    'C': [[('J', 1)]],
    'D': [[('E', 1), ('F', 1)]],
    'G': [[('I', 1)]]
```

```
}

start = 'A'
aoStar(start, False)
print("Traversing from startnode:", start)
print(solution)
```

Program3

```
import pandas as pd
import numpy as np

data = pd.read_csv('lab3.csv')
concept = np.array(data)[:, :-1]
print('\nConcepts to be learned')
print(concept)
target = np.array(data)[:, -1]
print('\nLabels specific to the concepts')
print(target)
specific = concept[0].copy()
general = [['?' for i in range(len(specific))] for i in range(len(specific))]

for i, h in enumerate(concept):
    if target[i] == 'Yes':
        for j in range(len(specific)):
            if h[j] != specific[j]:
                specific[j] = '?'
                general[j][j] = '?'
    else:
        for j in range(len(specific)):
            if h[j] != specific[j]:
                general[j][j] = specific[j]
            else:
                general[j][j] = '?'
indices = [i for i, val in enumerate(general) if val == ['?' for i in
range(len(specific))]]

for i in indices:
    general.remove(['?' for i in range(len(specific))])

print("\nFinal S: ", specific)
print("\nFinal G: ", general)
```

Program 4

```
import pandas as pd
import math

class Node:
    def __init__(self, l):
        self.label = l
        self.branch = {}

def decisionTree(data):
    root = Node("NULL")
```

```
    if(entropy(data) == 0):
        if(len(data.loc[data[data.columns[-1]] == "Yes"]) == len(data)):
            root.label = "Yes"
        else:
            root.label = "No"
        return root
    if(len(data.columns) > 1):
        #attribute = getAttribute(data)
        root.label = getAttribute(data)
        for i in set(data[root.label]):
            root.branch[i] = decisionTree(data.loc[data[root.label] ==
i].drop(root.label, axis = 1))
        return root

def entropy(data):
    total = len(data)
    pos = len(data.loc[data['PlayTennis'] == 'Yes'])
    neg = len(data.loc[data['PlayTennis'] == 'No'])
    en = 0
    if(pos > 0):
        en = -(pos/total) * (math.log(pos,2) - math.log(total, 2))
    if(neg > 0):
        en += -(neg/total) * (math.log(neg,2) - math.log(total, 2))
    return en

def getAttribute(data):
    attribute = ""
    max = 0
    for i in data.columns[:-1]:
        g = gain(data, i)
        if g > max:
            max = g
            attribute = i
    return attribute

def gain(data, attribute):
    en = entropy(data)
    for i in set(data[attribute]):
        en -= len(data.loc[data[attribute] == i])/len(data) *
entropy(data.loc[data[attribute] == i])
    return en

def getRules(root, rule, rules):
    if not root.branch:
        rules.append(rule[:-1] + "=>" + root.label)
        return rules
    for i in root.branch:
        getRules(root.branch[i], rule + root.label + "=" + str(i) + "^", rules)
    return rules

def test(tree, s):
    if not tree.branch:
        return tree.label
    return test(tree.branch[str(s[tree.label])], s)

data = pd.read_csv("lab4.csv")
tree = decisionTree(data)
rules = getRules(tree, '', [])
for i in rules:
```

```
    print(i)
s = {}
print("Enter the test case input: ")
for i in data.columns[:-1]:
    s[i] = input(i + ": ")
print(s)
print(test(tree, s))
```

Program 5

```
from math import exp
from random import seed
from random import random

def init():
    network = []
    network.append([{'weights':[random() for i in range(input)]} for i in
range(hidden)])
    network.append([{'weights':[random() for i in range(hidden + 1)]} for i in
range(output)])
    return network

def activate(weights, inputs):
    activation = weights[-1]
    for i in range(len(weights)-1):
        activation += weights[i] * inputs[i]
    return activation

def forward(inputs):
    for layer in network:
        temp = []
        for neuron in layer:
            activation = activate(neuron['weights'], inputs)
            neuron['output'] = 1 / (1 + exp(-activation))
            temp.append(neuron['output'])
        inputs = temp
    return inputs

def backward(expected):
    for i in reversed(range(len(network))):
        layer = network[i]
        errors = []
        if i < len(network)-1:
            for j in range(len(layer)):
                error = 0
                for neuron in network[i + 1]:
                    error += neuron['weights'][j] * neuron['delta']
                errors.append(error)
        else:
            for j in range(len(layer)):
                neuron = layer[j]
                errors.append(expected[j] - neuron['output'])
        for j in range(len(layer)):
            neuron = layer[j]
            neuron['delta'] = errors[j] * neuron['output'] * (1 - neuron['output'])

def update(inputs, lrate):
    for i in range(len(network)):
```

```
        inputs = inputs[:-1]
        if i != 0:
            inputs = [neuron['output'] for neuron in network[i - 1]]
        for neuron in network[i]:
            for j in range(len(inputs)):
                neuron['weights'][j] += lrate * neuron['delta'] * inputs[j]
            neuron['weights'][-1] += lrate * neuron['delta']

def train(epoch):
    lrate = 0.5
    for i in range(epoch):
        error = 0
        for inputs in data:
            outputs = forward(inputs)
            expected = [0 for i in range(output)]
            expected[inputs[-1]] = 1
            for j in range(len(expected)):
                error += (expected[j]-outputs[j])**2
            backward(expected)
            update(inputs, lrate)
        print('epoch = %d, lrate = %.3f, error = %.3f' % (i, lrate, error))

seed(1)
data = [[2.7810836,2.550537003,0],
           [1.465489372,2.362125076,0],
           [3.396561688,4.400293529,0],
           [1.38807019,1.850220317,0],
           [3.06407232,3.005305973,0],
           [7.627531214,2.759262235,1],
           [5.332441248,2.088626775,1],
           [6.922596716,1.77106367,1],
           [8.675418651,-0.242068655,1],
           [7.673756466,3.508563011,1]]

input = len(data[0])
hidden = 2
output = len(set([inputs[-1] for inputs in data]))
network = init()
train(20)
for layer in network:
    print(layer)
```

Program 6

```
import pandas as pd
data = pd.read_csv('lab6.csv')
total = len(data)
pos = len(data.loc[data['PlayTennis'] == 'Yes'])
neg = total - pos
train = data.sample(frac = 0.75, replace = False)
test = pd.concat([data, train]).drop_duplicates(keep = False)
print('Training Set : \n', train)
print('\nTesting Set : \n', test)

prob = {}

for i in train.columns[:-1]:
    prob[i] = {}
```

```
    for j in set(data[i]):
        temp = train.loc[train[i] == j]
        pe = len(temp.loc[temp['PlayTennis'] == 'Yes'])
        ne = len(temp) - pe
        prob[i][j] = [pe/pos, ne/neg]

pred = []
pospred = 0

for i in range(len(test)):
    row = test.iloc[i,:]
    posprob = pos/total
    negprob = neg/total
    for col in test.columns[:-1]:
        posprob *= prob[col][row[col]][0]
        negprob *= prob[col][row[col]][1]
    if posprob > negprob:
        pred.append('Yes')
    else:
        pred.append('No')
    if pred[-1] == row[-1]:
        pospred += 1

print('\nActual Values : ', list(test[test.columns[-1]]))
print('Predicted : ', pred)
print('Accuracy : ', pospred/len(test))
```

Program 71

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

iris = datasets.load_iris()

x = pd.DataFrame(iris.data)
x.columns = ['Sepal_Length','Sepal_Width','Petal_Length','Petal_Width']

y = pd.DataFrame(iris.target)
y.columns = ['Targets']

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

plt.subplot(1,2,1)
plt.scatter(x.Sepal_Length, x.Sepal_Width, c = color[y.Targets], s = 40)
plt.title('Sepal')

plt.subplot(1,2,2)
plt.scatter(x.Petal_Length, x.Petal_Width, c = color[y.Targets], s = 40)
plt.title('Petal')

model = KMeans(n_clusters = 3)
model.fit(x)
```

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

plt.subplot(1,2,1)
plt.scatter(x.Petal_Length, x.Petal_Width, c = color[y.Targets], s = 40)
plt.title('Real')

plt.subplot(1,2,2)
plt.scatter(x.Petal_Length, x.Petal_Width, c = color[model.labels_], s = 40)
plt.title('Kmeans')

print("accuracy_score", accuracy_score(y.Targets, model.labels_))
print("confusion_matrix\n", confusion_matrix(y.Targets,model.labels_))
```

Program 72

```
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np
from sklearn import datasets
from sklearn.metrics import accuracy_score
from sklearn.metrics import confusion_matrix
from sklearn import preprocessing
from sklearn.mixture import GaussianMixture

iris = datasets.load_iris()

x = pd.DataFrame(iris.data)
x.columns = ['Sepal_Length','Sepal_Width','Petal_Length','Petal_Width']

y = pd.DataFrame(iris.target)
y.columns = ['Targets']

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

plt.subplot(1,2,1)
plt.scatter(x.Sepal_Length, x.Sepal_Width, c = color[y.Targets], s = 40)
plt.title('Sepal')

plt.subplot(1,2,2)
plt.scatter(x.Petal_Length, x.Petal_Width, c = color[y.Targets], s = 40)
plt.title('Petal')

scaler = preprocessing.StandardScaler()
scaler.fit(x)
xs = pd.DataFrame(scaler.transform(x), columns = x.columns)
model = GaussianMixture(n_components = 3)
model.fit(xs)
model.labels_ = model.predict(xs)

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

plt.subplot(1,2,1)
plt.scatter(x.Petal_Length, x.Petal_Width, c = color[y.Targets], s = 40)
plt.title('Real')
```

```
plt.subplot(1,2,2)
plt.scatter(x.Petal_Length, x.Petal_Width, c = color[model.labels_], s = 40)
plt.title('EM')

print("accuracy_score", accuracy_score(y.Targets, model.labels_))
print("confusion_matrix\n", confusion_matrix(y.Targets, model.labels_))
```

Program 8

```
import csv
import random
import math
import operator

def init(file, split):
    data = list(csv.reader(open(file)))
    for i in range(len(data)-1):
        for j in range(4):
            data[i][j] = float(data[i][j])
        if random.random() < split:
            train.append(data[i])
        else:
            test.append(data[i])
    print('\n Number of Training data:', len(train))
    print(' Number of Test Data: ', len(test))

def getNeighbors(testinstance, k):
    distances = []
    for traininstance in train:
        euclid = 0
        for i in range(len(testinstance)-1):
            euclid += pow((testinstance[i] - traininstance[i]), 2)
        distances.append((traininstance, math.sqrt(euclid)))
    distances.sort(key = operator.itemgetter(1))
    neighbors = []
    for i in range(k):
        neighbors.append(distances[i][0])
    return neighbors

def getResponse(neighbors):
    classVotes = {}
    for neighbor in neighbors:
        response = neighbor[-1]
        if response not in classVotes:
            classVotes[response] = 1
        else:
            classVotes[response] += 1
    sortedVotes = sorted(classVotes.items(), key = operator.itemgetter(1))
    return sortedVotes[-1][0]

def accuracy():
    correct = 0
    for i in range(len(test)):
        if test[i][-1] == predicted[i]:
            correct += 1
    print('\n The Accuracy is: ', correct/len(test) * 100, '%')
```

```
train = []
test = []
predicted = []

split = 0.67
init('lab8.csv', split)
k = 3

print('\n The predictions are: ')
for testinstance in test:
    neighbors = getNeighbors(testinstance, k)
    result = getResponse(neighbors)
    predicted.append(result)
    print(' predicted =', result, ', actual =', testinstance[-1])
accuracy()
```

Program 9

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

def localWeightRegression(k):
    ypred = np.zeros(m)
    for i in range(m):
        w = np.mat(np.eye(m))
        for j in range(m):
            diff = x[i] - x[j]
            w[j, j] = np.exp(diff * diff.T/(-2 * k**2))
        ypred[i] = x[i] * (x.T * w * x).I * x.T * w * y.T
    return ypred

def graphPlot():
    i = x[:,1].argsort(0)
    plt.figure()
    plt.subplot(1, 1, 1)
    plt.scatter(bill, tip, color = 'green')
    plt.plot(x[i,1], ypred[i], color = 'red', linewidth = 2)
    plt.xlabel('Total bill')
    plt.ylabel('Tip')
    plt.show()

data = pd.read_csv('lab9.csv')

bill = np.array(data.bill)
tip = np.array(data.tip)

mbill = np.mat(bill)
mtip = np.mat(tip)

one = np.mat(np.ones(np.shape(mbill)[1]))

x = np.hstack((one.T, mbill.T))
y = mtip

m, n = np.shape(x)
ypred = localWeightRegression(1.5)
graphPlot()
```