all-programs

November 7, 2023

[1]:
```
# Program1

def aStarAlgo(start_node, stop_node):

        open_set = set(start_node)
        print(open_set)
        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':10,
            'B':8,
            'C':5,
            'D':7,
            'E':3,
            'F':6,
            'G':5,
            'H':3,
            'I':1,
            'J':0
        }

        return H_dist[n]

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')
```

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

[1]:
```
['A', 'F', 'G', 'I', 'J']
```

[3]:
```
# Program 2
```

[4]:
```
def recAOStar(n):
    global finalPath
    print("Expanding Node:",n)
    and_nodes = []
    or_nodes =[]
    if(n in allNodes):
        if 'AND' in allNodes[n]:
            and_nodes = allNodes[n]['AND']
        if 'OR' in allNodes[n]:
            or_nodes = allNodes[n]['OR']
```

```
    if len(and_nodes)==0 and len(or_nodes)==0:
        return

    solvable = False
    marked ={}

    while not solvable:
        if len(marked)==len(and_nodes)+len(or_nodes):
            min_cost_least,min_cost_group_least =␣
 ↪least_cost_group(and_nodes,or_nodes,{})
            solvable = True
            change_heuristic(n,min_cost_least)
            optimal_child_group[n] = min_cost_group_least
            continue
        min_cost,min_cost_group = least_cost_group(and_nodes,or_nodes,marked)
        is_expanded = False
        if len(min_cost_group)>1:
            if(min_cost_group[0] in allNodes):
                is_expanded = True
                recAOStar(min_cost_group[0])
            if(min_cost_group[1] in allNodes):
                is_expanded = True
                recAOStar(min_cost_group[1])
        else:
            if(min_cost_group in allNodes):
                is_expanded = True
                recAOStar(min_cost_group)
        if is_expanded:
            min_cost_verify, min_cost_group_verify =␣
 ↪least_cost_group(and_nodes, or_nodes, {})
            if min_cost_group == min_cost_group_verify:
                solvable = True
                change_heuristic(n, min_cost_verify)
                optimal_child_group[n] = min_cost_group
        else:
            solvable = True
            change_heuristic(n, min_cost)
            optimal_child_group[n] = min_cost_group
        marked[min_cost_group]=1
    return heuristic(n)

def least_cost_group(and_nodes, or_nodes, marked):
    node_wise_cost = {}
    for node_pair in and_nodes:
        if not node_pair[0] + node_pair[1] in marked:
            cost = 0
            cost = cost + heuristic(node_pair[0]) + heuristic(node_pair[1]) + 2
```

```
            node_wise_cost[node_pair[0] + node_pair[1]] = cost
    for node in or_nodes:
        if not node in marked:
            cost = 0
            cost = cost + heuristic(node) + 1
            node_wise_cost[node] = cost
    min_cost = 999999
    min_cost_group = None
    for costKey in node_wise_cost:
        if node_wise_cost[costKey] < min_cost:
            min_cost = node_wise_cost[costKey]
            min_cost_group = costKey
    return [min_cost, min_cost_group]

def heuristic(n):
    return H_dist[n]

def change_heuristic(n, cost):
    H_dist[n] = cost
    return

def print_path(node):
    print(optimal_child_group[node], end="")
    node = optimal_child_group[node]
    if len(node) > 1:
        if node[0] in optimal_child_group:
            print("->", end="")
            print_path(node[0])
        if node[1] in optimal_child_group:
            print("->", end="")
            print_path(node[1])
    else:
        if node in optimal_child_group:
            print("->", end="")
            print_path(node)
H_dist = {
 'A': -1,
 'B': 4,
 'C': 2,
 'D': 3,
 'E': 6,
 'F': 8,
 'G': 2,
 'H': 0,
 'I': 0,
 'J': 0
}
```

```
allNodes = {
 'A': {'AND': [('C', 'D')], 'OR': ['B']},
 'B': {'OR': ['E', 'F']},
 'C': {'OR': ['G'], 'AND': [('H', 'I')]},
 'D': {'OR': ['J']}
}
optimal_child_group = {}
optimal_cost = recAOStar('A')
print('Nodes which gives optimal cost are')
print_path('A')
print('\nOptimal Cost is :: ', optimal_cost)
```

```
Expanding Node: A
Expanding Node: B
Expanding Node: C
Expanding Node: D
Nodes which gives optimal cost are
CD->HI->J
Optimal Cost is ::  5
```

[5]:
```
## Program 3
```

[ ]:
```
pr3.csv

slno,Sky,AirTemp,Humidity,Wind,Water,forecast,EnjoySport
0,sunny,warm,normal,strong,warm,same,yes
1,sunny,warm,high,strong,warm,same,yes
2,rainy,cold,high,strong,warm,change,no
3,sunny,warm,high,strong,warm,change,yes
```

[6]:
```
import pandas as pd
df = pd.read_csv('/home/sahyadri/CS114/Prog3/pr3.csv')
df = df.drop(['slno'], axis=1)
concepts = df.values[:,:-1]
target = df.values[:,-1]
df.head()
def learn(concepts, target):
    specific_h = concepts[0].copy()
    general_h = [["?" for i in range(len(specific_h))] for i in␣
 ↪range(len(specific_h))]
    for i, h in enumerate(concepts):
        if target[i] == "yes":
            for x in range(len(specific_h)):
                if h[x] != specific_h[x]:
                    specific_h[x] = '?'
                    general_h[x][x] = '?'
        if target[i] == "no":
```

```
            for x in range(len(specific_h)):
                if h[x] != specific_h[x]:
                    general_h[x][x] = specific_h[x]
                else:
                    general_h[x][x] = '?'
    indices = [i for i, val in enumerate(general_h) if val == ['?','?','?','?
 ↪','?','?']]
    for i in indices:
        general_h.remove(['?','?','?','?','?','?'])
    return specific_h, general_h
s_final, g_final = learn(concepts, target)
print(f"Final S: {s_final}")
print(f"Final G: {g_final}")
```

```
Final S: ['sunny' 'warm' '?' 'strong' 'warm' '?']
Final G: [['sunny', '?', '?', '?', '?', '?'], ['?', 'warm', '?', '?', '?', '?']]
```

[7]:
```
#Program 4
```

[ ]:
```
Tennis.csv

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

[8]:
```
import pandas as pd
from pandas import DataFrame
from math import log
from collections import Counter
from pprint import pprint

df_tennis = pd.read_csv('/home/sahyadri/CS114/prog4/tennis.csv')
df_tennis.keys()[0]

def entropy(probs):
```

```
    return sum([-prob * log(prob, 2) for prob in probs])

def entropy_of_list(a_list):
    cnt = Counter(x for x in a_list)
    num_instances = len(a_list) * 1.0
    probs = [x / num_instances for x in cnt.values()]
    return entropy(probs)

def information_gain(df, split_attribute_name, target_attribute_name):
    df_split = df.groupby(split_attribute_name)
    nobs = len(df.index) * 1.0
    df_agg_ent = df_split.agg({target_attribute_name: [entropy_of_list, lambda␣
 ↪x: len(x)/nobs]})[target_attribute_name]
    df_agg_ent.columns = ['Entropy', 'PropObservations']
    new_entropy = sum(df_agg_ent['Entropy'] * df_agg_ent['PropObservations'])
    old_entropy = entropy_of_list(df[target_attribute_name])
    return old_entropy - new_entropy

def id3(df, target_attribute_name, attribute_names, default_class=None):
    cnt = Counter(x for x in df[target_attribute_name])
    if len(cnt) == 1:
        return next(iter(cnt))
    elif df.empty or (not attribute_names):
        return default_class
    else:
        default_class = max(cnt.keys())
        gainz = [information_gain(df, attr, target_attribute_name) for attr in␣
 ↪attribute_names]
        index_of_max = gainz.index(max(gainz))
        best_attr = attribute_names[index_of_max]
        tree = {best_attr:{}}
        remaining_attribute_names = [i for i in attribute_names if i !=␣
 ↪best_attr]

        for attr_val, data_subset in df.groupby(best_attr):
            subtree = id3(data_subset, target_attribute_name,␣
 ↪remaining_attribute_names, default_class)
            tree[best_attr][attr_val] = subtree
    return tree

attribute_names = list(df_tennis.columns)

attribute_names.remove('PlayTennis')

tree = id3(df_tennis, 'PlayTennis', attribute_names)

print("\n\nThe Resultant Decistion Tree is: \n")
```

```
pprint(tree)
```

```
The Resultant Decistion Tree is:

{'Outlook': {'Overcast': 'Yes',
             'Rain': {'Wind': {'Strong': 'No', 'Weak': 'Yes'}},
             'Sunny': {'Humidity': {'High': 'No', 'Normal': 'Yes'}}}}
```

[9]:
```
#Program 5
```

[10]:
```
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)
epoch = 1000
learning_rate = 0.6
inputlayer_neurons = 2
hiddenlayer_neurons = 3
output_neurons = 1
wh = np.random.uniform(size=(inputlayer_neurons, hiddenlayer_neurons))
bh = np.random.uniform(size=(1, hiddenlayer_neurons))
wo = np.random.uniform(size=(hiddenlayer_neurons, output_neurons))
bo = np.random.uniform(size=(1, output_neurons))
for i in range(epoch):
        #Forward Propogation
        net_h = np.dot(X, wh) + bh
        sigma_h = sigmoid(net_h)
        net_o = np.dot(sigma_h, wo) + bo
        output = sigmoid(net_o)
        #Backpropagation
        deltaK = (y - output) * derivatives_sigmoid(output)
        deltaH = deltaK.dot(wo.T) * derivatives_sigmoid(sigma_h)
        wo = wo + sigma_h.T.dot(deltaK) * learning_rate
        wh = wh + X.T.dot(deltaH) * learning_rate
print(f"Input: \n {X}")
print(f"Actual Output: \n{y}")
print(f"Predicted Output: \n{output}")
```

```
Input:
 [[0.66666667 1.        ]
```

```
 [0.33333333 0.55555556]
 [1.         0.66666667]]
Actual Output:
[[0.92]
 [0.86]
 [0.89]]
Predicted Output:
[[0.89632422]
 [0.8750899 ]
 [0.89799326]]
```

[11]:
```
#Program 6
```

[ ]:
```
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
```

[12]:
```
def probAttr(data, attr, val):
    Total = data.shape[0]
    cnt = len(data[data[attr] == val])
    return cnt, cnt / Total

def train(data, Attr, conceptVals, concept):
    conceptProbs = {}
    countConcept = {}
    for cVal in conceptVals:
        countConcept[cVal], conceptProbs[cVal] = probAttr(data, concept, cVal)
    AttrConcept = {}
    probability_list = {}
    for att in Attr:
        probability_list[att] = {}
        AttrConcept[att] = {}
        for val in Attr[att]:
            AttrConcept[att][val] = {}
            a, probability_list[att][val] = probAttr(data, att, val)
```

```
            for cVal in conceptVals:
                dataTemp = data[data[att] == val]
                AttrConcept[att][val][cVal] = len(dataTemp[dataTemp[concept]␣
 ↪==cVal]) / countConcept[cVal]
    print(f"P(A) : {conceptProbs}\n")
    print(f"P(X/A) : {AttrConcept}\n")
    print(f"P(X) : {probability_list}\n")
    return conceptProbs, AttrConcept, probability_list

def test(examples, Attr, concept_list, conceptProbs,␣
 ↪AttrConcept,probability_list):
    misclassification_count = 0
    Total = len(examples)
    for ex in examples:
        px = {}
        for a in Attr:
            for x in ex:
                for c in concept_list:
                    if x in AttrConcept[a]:
                        if c not in px:
                            px[c] = conceptProbs[c] * AttrConcept[a][x][c]/
 ↪probability_list[a][x]
                        else:
                            px[c] = px[c] * AttrConcept[a][x][c] /
 ↪probability_list[a][x]
        print(px)
        classification = max(px, key=px.get)
        print(f"Classification : {classification} Expected : {ex[-1]}")
        if (classification != ex[-1]):
            misclassification_count += 1
            misclassification_rate = misclassification_count * 100 / Total
            accuracy = 100 - misclassification_rate
    print(f"Misclassification Count={misclassification_count}")
    print(f"Misclassification Rate={misclassification_rate}%")
    print(f"Accuracy={accuracy}%")
import pandas as pd
df = pd.read_csv('/home/sahyadri/CS114/Prog6/playTennis.csv')
concept = str(list(df)[-1])
concept_list = set(df[concept])
Attr = {}
for a in df.columns[:-1]:
    Attr[a] = set(df[a])
    print(f"{a}: {Attr[a]}")
conceptProbs, AttrConcept, probability_list = train(df, Attr,␣
 ↪concept_list,concept)
examples = pd.read_csv('/home/sahyadri/CS114/Prog6/playTennis.csv')
```

```
test(examples.values, Attr, concept_list, conceptProbs,␣
 ↪AttrConcept,probability_list)
```

```
Outlook: {'overcast', 'rain', 'sunny'}
Temperature: {'mild', 'hot', 'cool'}
Humidity: {'high', 'normal'}
Wind: {'strong', 'weak'}
P(A) : {'no': 0.35714285714285715, 'yes': 0.6428571428571429}

P(X/A) : {'Outlook': {'overcast': {'no': 0.0, 'yes': 0.4444444444444444},
'rain': {'no': 0.4, 'yes': 0.3333333333333333}, 'sunny': {'no': 0.6, 'yes':
0.2222222222222222}}, 'Temperature': {'mild': {'no': 0.4, 'yes':
0.4444444444444444}, 'hot': {'no': 0.4, 'yes': 0.2222222222222222}, 'cool':
{'no': 0.2, 'yes': 0.3333333333333333}}, 'Humidity': {'high': {'no': 0.8, 'yes':
0.3333333333333333}, 'normal': {'no': 0.2, 'yes': 0.6666666666666666}}, 'Wind':
{'strong': {'no': 0.6, 'yes': 0.3333333333333333}, 'weak': {'no': 0.4, 'yes':
0.6666666666666666}}}

P(X) : {'Outlook': {'overcast': 0.2857142857142857, 'rain': 0.35714285714285715,
'sunny': 0.35714285714285715}, 'Temperature': {'mild': 0.42857142857142855,
'hot': 0.2857142857142857, 'cool': 0.2857142857142857}, 'Humidity': {'high':
0.5, 'normal': 0.5}, 'Wind': {'strong': 0.42857142857142855, 'weak':
0.5714285714285714}}

{'no': 0.9408000000000002, 'yes': 0.2419753086419753}
Classification : no Expected : no
{'no': 1.8816000000000002, 'yes': 0.16131687242798354}
Classification : no Expected : no
{'no': 0.0, 'yes': 0.6049382716049383}
Classification : yes Expected : yes
{'no': 0.4181333333333335, 'yes': 0.4839506172839506}
Classification : yes Expected : yes
{'no': 0.07840000000000004, 'yes': 1.0888888888888888}
Classification : yes Expected : yes
{'no': 0.15680000000000005, 'yes': 0.7259259259259259}
Classification : yes Expected : no
{'no': 0.0, 'yes': 1.2098765432098766}
Classification : yes Expected : yes
{'no': 0.6272000000000001, 'yes': 0.3226337448559671}
Classification : no Expected : no
{'no': 0.11760000000000002, 'yes': 0.7259259259259256}
Classification : yes Expected : yes
{'no': 0.10453333333333338, 'yes': 0.9679012345679012}
Classification : yes Expected : yes
{'no': 0.31360000000000005, 'yes': 0.43017832647462273}
Classification : yes Expected : yes
{'no': 0.0, 'yes': 0.5377229080932785}
```

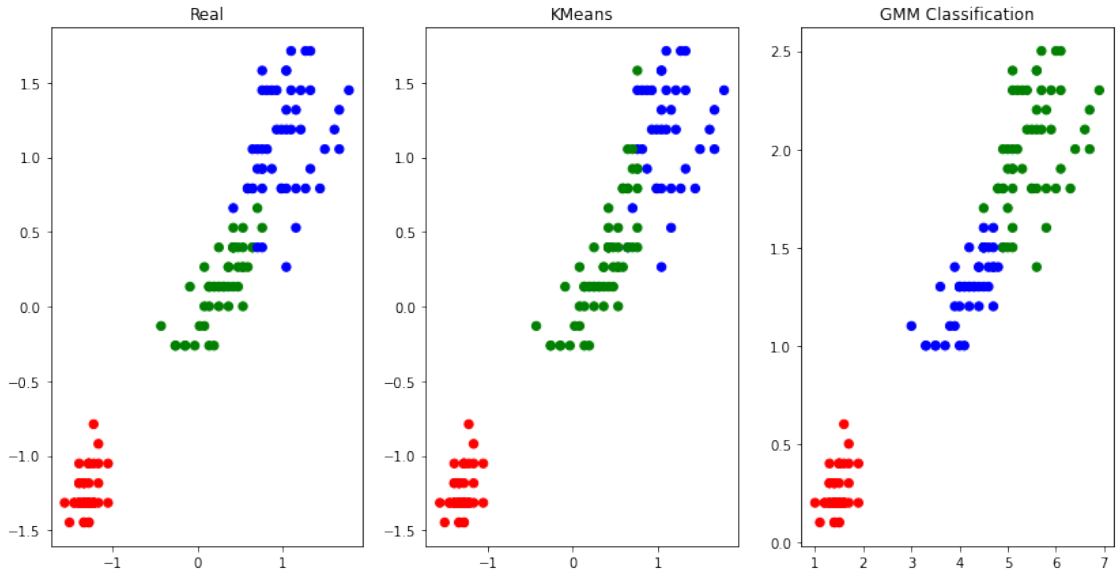
```
Classification : yes Expected : yes
{'no': 0.0, 'yes': 1.2098765432098766}
Classification : yes Expected : yes
{'no': 0.8362666666666669, 'yes': 0.3226337448559671}
Classification : no Expected : no
Misclassification Count=1
Misclassification Rate=7.142857142857143%
Accuracy=92.85714285714286%
```

[13]:
```
#Program 7
```

[14]:
```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
from sklearn import preprocessing
from sklearn.cluster import KMeans
from sklearn.datasets import load_iris
from sklearn.mixture import GaussianMixture

iris = load_iris()
df = pd.DataFrame(iris['data'], columns=iris['feature_names'])
df['target'] = iris['target']

X = df.iloc[:, :-1]
y = df['target']

from sklearn import preprocessing
scaler = preprocessing.StandardScaler()
scaler.fit(X)
X_scaled_array = scaler.transform(X)
X_scaled = pd.DataFrame(X_scaled_array, columns = X.columns)

plt.figure(figsize=(14, 7))
colormap = np.array(['red', 'green', 'blue'])
#REAL PLOT
plt.subplot(1, 3, 1)
plt.scatter(X_scaled['petal length (cm)'],X_scaled['petal width␣
 ↪(cm)'],c=colormap[y], s=40)
plt.title('Real')
#K-PLOT
plt.subplot(1, 3, 2)
model = KMeans(n_clusters=3, random_state=0)
pred_y = model.fit_predict(X)
pred_y = np.choose(pred_y, [1, 0, 2]).astype(np.int64)
plt.scatter(X_scaled['petal length (cm)'], X_scaled['petal width␣
 ↪(cm)'],c=colormap[pred_y], s=40)
plt.title('KMeans')
```

```
#GMM PLOT
gmm = GaussianMixture(n_components=3, max_iter=200)
y_cluster_gmm = gmm.fit_predict(X_scaled)
y_cluster_gmm = np.choose(y_cluster_gmm, [2, 0, 1]).astype(np.int64)
plt.subplot(1, 3, 3)
plt.scatter(X['petal length (cm)'], X['petal width␣
 ↪(cm)'],c=colormap[y_cluster_gmm], s=40)
plt.title('GMM Classification')
```

[14]: Text(0.5, 1.0, 'GMM Classification')



[15]:
```
#Program 8
```

[16]:
```
from sklearn .model_selection import train_test_split
from sklearn .neighbors import KNeighborsClassifier
from sklearn import datasets
iris=datasets.load_iris()
print('iris Data set loaded,....')
x_train,x_test,y_train,y_test=train_test_split(iris.data,iris.
 ↪target,test_size=0.1)
for i in range(len(iris.target_names)):
    print("Label",i,"-",str(iris.target_names[i]))
classifier=KNeighborsClassifier(n_neighbors=2)
classifier.fit(x_train,y_train)
y_pred=classifier.predict(x_test)
print("Results of Classification using KNN  with k=1:")
for r in range(0,len(x_test)):
```

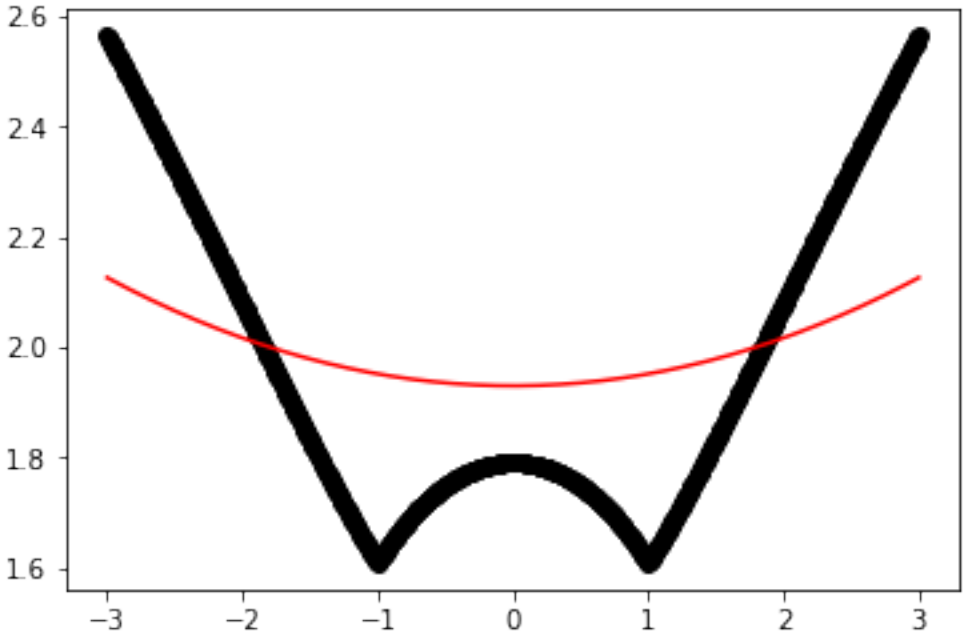
```
    print("Sample:",str(x_test[r]),"actual-label:
  ↪",str(y_test[r]),"Predicted-Label:",str(y_pred[r]))
print("Classification accuracy:",classifier.score(x_test,y_test))
```

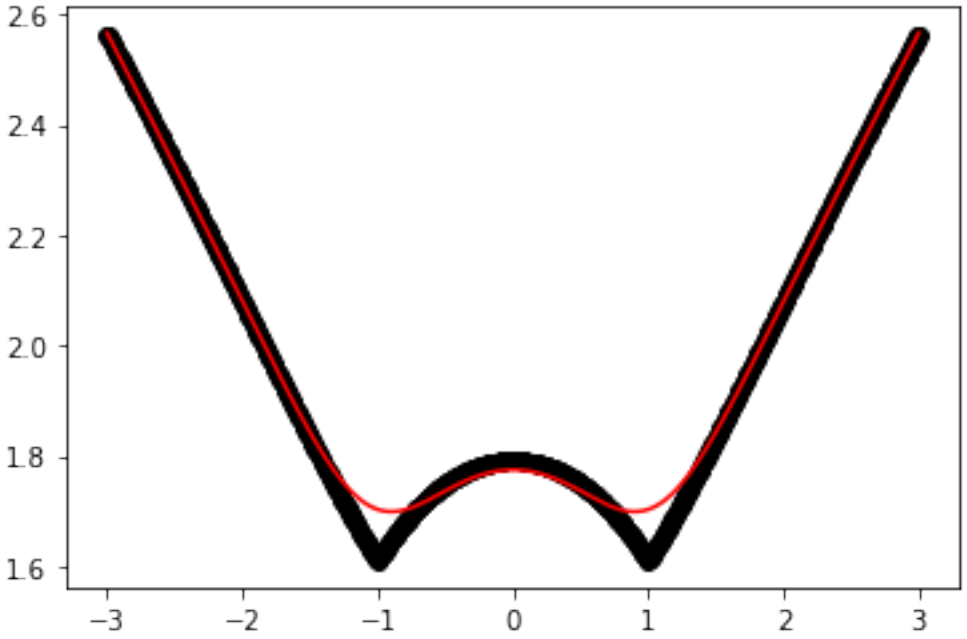
```
iris Data set loaded,…
Label 0 - setosa
Label 1 - versicolor
Label 2 - virginica
Results of Classification using KNN  with k=1:
Sample: [5.6 3.  4.5 1.5] actual-label: 1 Predicted-Label: 1
Sample: [6.  2.2 4.  1. ] actual-label: 1 Predicted-Label: 1
Sample: [6.  2.2 5.  1.5] actual-label: 2 Predicted-Label: 1
Sample: [5.8 2.7 3.9 1.2] actual-label: 1 Predicted-Label: 1
Sample: [7.7 3.8 6.7 2.2] actual-label: 2 Predicted-Label: 2
Sample: [5.  3.4 1.5 0.2] actual-label: 0 Predicted-Label: 0
Sample: [6.1 2.6 5.6 1.4] actual-label: 2 Predicted-Label: 1
Sample: [6.4 2.8 5.6 2.2] actual-label: 2 Predicted-Label: 2
Sample: [5.4 3.9 1.3 0.4] actual-label: 0 Predicted-Label: 0
Sample: [5.  3.3 1.4 0.2] actual-label: 0 Predicted-Label: 0
Sample: [6.4 3.1 5.5 1.8] actual-label: 2 Predicted-Label: 2
Sample: [6.9 3.2 5.7 2.3] actual-label: 2 Predicted-Label: 2
Sample: [6.5 3.  5.2 2. ] actual-label: 2 Predicted-Label: 2
Sample: [5.2 3.4 1.4 0.2] actual-label: 0 Predicted-Label: 0
Sample: [4.7 3.2 1.6 0.2] actual-label: 0 Predicted-Label: 0
Classification accuracy: 0.8666666666666667
```

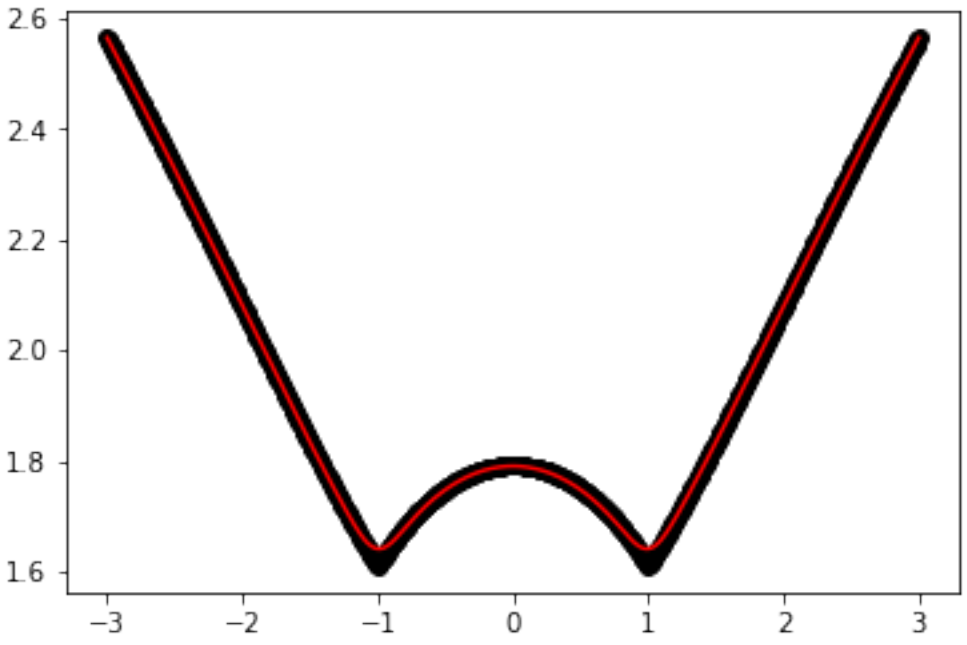
[17]:
```
#Program 9
```

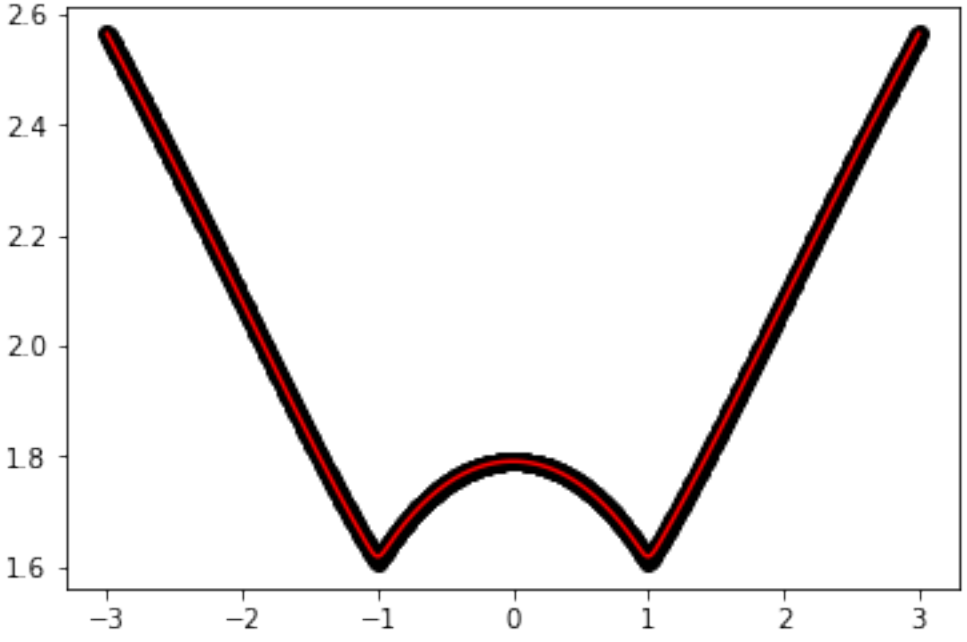
[18]:
```
import numpy as np
import matplotlib.pyplot as plt

def local_regression(x0,X,Y,tau):
    x0=[1,x0]
    X=[[1,i] for i in X]
    X=np.array(X)
    xw=(X.T)* np.exp(np.sum((X-x0)**2,axis=1)/(-2*tau))
    beta=np.linalg.pinv(xw @ X)@xw @Y @x0
    return beta
def draw (tau):
    prediction=[local_regression(x0,X,Y,tau) for x0 in domain]
    plt.plot(X,Y,'o',color='black')
    plt.plot(domain, prediction, color='red')
    plt.show()
X=np.linspace(-3,3,num=1000)
domain=X
Y=np.log(np.abs(X** 2-1)+5)
draw(10)
```

```
draw(0.1)
draw(0.01)
draw(0.001)
```






2.6
2.4
2.2
2.0
1.8
1.6
−3
−2
−1
0
1
2
3


2.6
2.4
2.2
2.0
1.8
1.6
−3
−2
−1
0
1
2
3

[ ]: