1:-A* 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': [('A', 2), ('C', 1), ('G', 9)],
    'C': [('B', 1)],
    'D': [('E', 6), ('G', 1)],
    'E': [('A', 3), ('D', 6)],
    'G': [('B', 9), ('D', 1)]
}
aStarAlgo('A', 'G')
```

 # OUTPUT:

Path found: ['A', 'E', 'D', 'G']

2:- AO* ALGORITM

```python
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 START NODE:",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)
            print(minimumCost, childNodeList)
            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)
print ("Graph - 2")
```

```
h2 = {'A': 1, 'B': 6, 'C': 12, 'D': 10, 'E': 4, 'F': 4, 'G': 5, 'H': 7}
graph2 = {
    'A': [[('B', 1), ('C', 1)], [('D', 1)]],
    'B': [[('G', 1)], [('H', 1)]],
    'D': [[('E', 1), ('F', 1)]]
}
G2 = Graph(graph2, h2, 'A')
G2.applyAOStar()
G2.printSolution()
```

OUTPUT

```
Graph - 2
HEURISTIC VALUES : {'A': 1, 'B': 6, 'C': 12, 'D': 10, 'E': 4, 'F': 4, 'G': 5, 'H': 7}
SOLUTION GRAPH : {}
PROCESSING NODE : A
-----------------------------------------------------------------------------------------
11 ['D']
HEURISTIC VALUES : {'A': 11, 'B': 6, 'C': 12, 'D': 10, 'E': 4, 'F': 4, 'G': 5, 'H': 7}
SOLUTION GRAPH : {}
PROCESSING NODE : D
-----------------------------------------------------------------------------------------
10 ['E', 'F']
HEURISTIC VALUES : {'A': 11, 'B': 6, 'C': 12, 'D': 10, 'E': 4, 'F': 4, 'G': 5, 'H': 7}
SOLUTION GRAPH : {}
PROCESSING NODE : A
-----------------------------------------------------------------------------------------
11 ['D']
HEURISTIC VALUES : {'A': 11, 'B': 6, 'C': 12, 'D': 10, 'E': 4, 'F': 4, 'G': 5, 'H': 7}
SOLUTION GRAPH : {}
```

```
PROCESSING NODE : E
-----------------------------------------------------------------------------------------
0 []
HEURISTIC VALUES : {'A': 11, 'B': 6, 'C': 12, 'D': 10, 'E': 0, 'F': 4, 'G': 5, 'H': 7}
SOLUTION GRAPH : {'E': []}
PROCESSING NODE : D
-----------------------------------------------------------------------------------------
6 ['E', 'F']
HEURISTIC VALUES : {'A': 11, 'B': 6, 'C': 12, 'D': 6, 'E': 0, 'F': 4, 'G': 5, 'H': 7}
SOLUTION GRAPH : {'E': []}
PROCESSING NODE : A
-----------------------------------------------------------------------------------------
7 ['D']
HEURISTIC VALUES : {'A': 7, 'B': 6, 'C': 12, 'D': 6, 'E': 0, 'F': 4, 'G': 5, 'H': 7}
SOLUTION GRAPH : {'E': []}
PROCESSING NODE : F
-----------------------------------------------------------------------------------------
0 []
HEURISTIC VALUES : {'A': 7, 'B': 6, 'C': 12, 'D': 6, 'E': 0, 'F': 0, 'G': 5, 'H': 7}
SOLUTION GRAPH : {'E': [], 'F': []}
PROCESSING NODE : D
-----------------------------------------------------------------------------------------
2 ['E', 'F']
HEURISTIC VALUES : {'A': 7, 'B': 6, 'C': 12, 'D': 2, 'E': 0, 'F': 0, 'G': 5, 'H': 7}
SOLUTION GRAPH : {'E': [], 'F': [], 'D': ['E', 'F']}
PROCESSING NODE : A
-----------------------------------------------------------------------------------------
3 ['D']
FOR GRAPH SOLUTION, TRAVERSE THE GRAPH FROM THE START NODE: A
------------------------------------------------------------
{'E': [], 'F': [], 'D': ['E', 'F'], 'A': ['D']}
```

3:- CANDIDATE ELIMINATION

```
import csv
with open("CandidateElimination.csv") as f:
    csv_file = csv.reader(f)
    data = list(csv_file)
    s = data[1][:-1]
    g = [['?' for i in range(len(s))] for j in range(len(s))]
    for i in data:
        if i[-1] == "Yes":
            for j in range(len(s)):
                if i[j] != s[j]:
                    s[j] = '?'
                    g[j][j] = '?'
        elif i[-1] == "No":
            for j in range(len(s)):
                if i[j] != s[j]:
                    g[j][j] = s[j]
                else:
                    g[j][j] = "?"
        print("\nSteps of Candidate Elimination Algorithm", data.index(i) + 1)
        print(s)
        print(g)
    gh = []
    for i in g:
        for j in i:
            if j != '?':
```

```
            gh.append(i)
            break
    print("\nFinal specific hypothesis:\n", s)
    print("\nFinal general hypothesis:\n", gh)
```

OUTPUT

Steps of Candidate Elimination Algorithm 1 ['Sunny', 'Warm', 'Normal', 'Strong', 'Warm', 'Same'] [['?', '?', '?',

'?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?',

'?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?']]

Steps of Candidate Elimination Algorithm 2 ['Sunny', 'Warm', 'Normal', 'Strong', 'Warm', 'Same'] [['?', '?', '?',

'?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?',

'?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?']]

Steps of Candidate Elimination Algorithm 3 ['Sunny', 'Warm', '?', 'Strong', 'Warm', 'Same'] [['?', '?', '?', '?',

'?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?',

'?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?']]

Steps of Candidate Elimination Algorithm 4 ['Sunny', 'Warm', '?', 'Strong', 'Warm', 'Same'] [['Sunny', '?', '?', '?',

'?', '?'], ['?', 'Warm', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?',

'?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', 'Same']]

Steps of Candidate Elimination Algorithm 5 ['Sunny', 'Warm', '?', 'Strong', '?', '?'] [['Sunny', '?', '?', '?', '?',

'?'], ['?', 'Warm', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?',

'?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?']]

Final specific hypothesis:

['Sunny', 'Warm', '?', 'Strong', '?', '?']

Final general hypothesis:

[['Sunny', '?', '?', '?', '?', '?'], ['?', 'Warm', '?', '?', '?', '?']]

4:- ID3

```
from pprint import pprint
import pandas as pd


df_tennis = pd.read_csv('ID3.csv')

def entropy(probs):
    import math
    return sum([-prob * math.log(prob, 2) for prob in probs])


def entropy_of_list(a_list):
    from collections import Counter
    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)
total_entropy = entropy_of_list(df_tennis['PlayTennis'])
def information_gain(df, split_attribute_name, target_attribute_name,
trace=0):
    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):  #
Tally target attribute
    from collections import Counter
    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 Decision Tree is :\n")
pprint(tree)
```

OUTPUT

```
The Resultant Decision Tree is :
{'Outlook': {'overcast': 'yes',
            'rain': {'Wind': {'strong': 'no', 'weak': 'yes'}},
            'sunny': {'Humidity': {'high': 'no', 'normal': 'yes'}}}}
```

5:-BACKPOPOGATION

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

```
epoch=7000
lr=0.1
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))
wout=np.random.uniform(size=(hiddenlayer_neurons,output_neurons))
bout=np.random.uniform(size=(1,output_neurons))
for i in range(epoch):
   hinp1=np.dot(X,wh)
   hinp=hinp1 + bh
   hlayer_act = sigmoid(hinp)
   outinp1=np.dot(hlayer_act,wout)
   outinp= outinp1+ bout
   output = sigmoid(outinp)
   EO = y-output
   outgrad = derivatives_sigmoid(output)
   d_output = EO* outgrad
   EH = d_output.dot(wout.T)
   hiddengrad = derivatives_sigmoid(hlayer_act)
   d_hiddenlayer = EH * hiddengrad
   wout += hlayer_act.T.dot(d_output) *lr
   bout += np.sum(d_output, axis=0,keepdims=True) *lr
   wh += X.T.dot(d_hiddenlayer) *lr
print("Input: \n" + str(X))
```

```
print("Actual Output: \n" + str(y))
print("Predicted Output: \n" ,output)
```

OUTPUT:-

```
Input:
[[0.66666667 1.        ]
 [0.33333333 0.55555556]
 [1.         0.66666667]]
Actual Output:
[[92.]
 [86.]
 [89.]]
Predicted Output:
 [[0.9999991 ]
 [0.99999699]
 [0.99999915]]
```

6:-NBC

```
import pandas as pd
from sklearn.preprocessing import LabelEncoder
from sklearn.naive_bayes import GaussianNB

# Load Data from CSV
data = pd.read_csv('ID3.csv')
print("The first 5 Values of data is :\n", data.head())

# obtain train data and train output
X = data.iloc[:, :-1]
print("\nThe First 5 values of the train data is\n", X.head())
```

```
y = data.iloc[:, -1]
print("\nThe First 5 values of train output is\n", y.head())

# convert them in numbers
le_outlook = LabelEncoder()
X.Outlook = le_outlook.fit_transform(X.Outlook)

le_Temperature = LabelEncoder()
X.Temperature = le_Temperature.fit_transform(X.Temperature)

le_Humidity = LabelEncoder()
X.Humidity = le_Humidity.fit_transform(X.Humidity)

le_Windy = LabelEncoder()
X.Windy = le_Windy.fit_transform(X.Windy)

print("\nNow the Train output is\n", X.head())

le_PlayTennis = LabelEncoder()
y = le_PlayTennis.fit_transform(y)
print("\nNow the Train output is\n",y)

from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X,y, test_size = 0.20)

classifier = GaussianNB()
classifier.fit(X_train, y_train)

from sklearn.metrics import accuracy_score
print("Accuracy is:", accuracy_score(classifier.predict(X_test), y_test))
```

OUTPUT

```
The first 5 Values of data is :
    Outlook Temperature Humidity   Windy PlayTennis
0    sunny        hot     high    weak        no
1    sunny        hot     high  strong        no
2  overcast       hot     high    weak       yes
3     rain       mild     high    weak       yes
4     rain       cool   normal    weak       yes

The First 5 values of the train data is
    Outlook Temperature Humidity   Windy
0    sunny        hot     high    weak
1    sunny        hot     high  strong
2  overcast       hot     high    weak
3     rain       mild     high    weak
4     rain       cool   normal    weak

The First 5 values of train output is
 0    no
1    no
2   yes
3   yes
4   yes
Name: PlayTennis, dtype: object

Now the Train output is
   Outlook  Temperature  Humidity  Windy
0       2           1        0     1
1       2           1        0     0
2       0           1        0     1
3       1           2        0     1
```

4 1 0 1 1

Now the Train output is

[0 0 1 1 1 0 1 0 1 1 1 1 1 0]

Accuracy is: 0.6666666666666666

7:-KMEANS

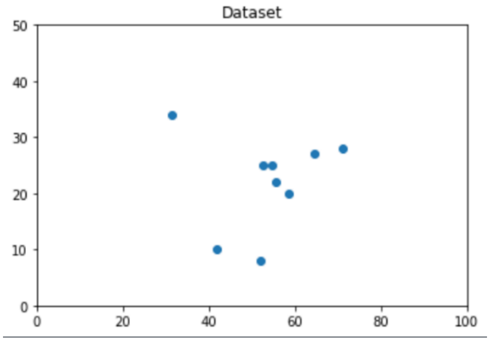
```
import numpy as np
from sklearn.cluster import KMeans
import matplotlib.pyplot as plt
from sklearn.mixture import GaussianMixture
import pandas as pd
X=pd.read_csv("kmeansdata.csv")
x1 = X['Distance_Feature'].values
x2 = X['Speeding_Feature'].values
X = np.array(list(zip(x1, x2))).reshape(len(x1), 2)
plt.plot()
plt.xlim([0, 100])
plt.ylim([0, 50])
plt.title('Dataset')
plt.scatter(x1, x2)
plt.show()
gmm = GaussianMixture(n_components=3)
gmm.fit(X)
em_predictions = gmm.predict(X)
```

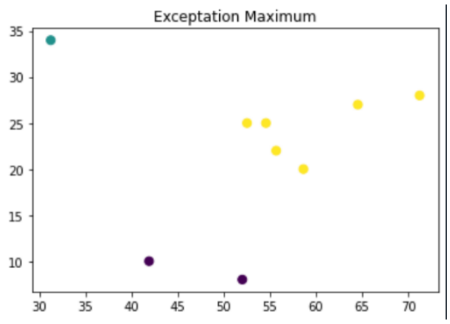
```
print("\nEM predictions")
print(em_predictions)
print("mean:\n",gmm.means_)
print('\n')
print("Covariances\n",gmm.covariances_)
print(X)
plt.title('Exceptation Maximum')
plt.scatter(X[:,0], X[:,1],c=em_predictions,s=50)
plt.show()

import matplotlib.pyplot as plt1
kmeans = KMeans(n_clusters=3)
kmeans.fit(X)
print(kmeans.cluster_centers_)
print(kmeans.labels_)
plt.title('KMEANS')
plt1.scatter(X[:,0], X[:,1], c=kmeans.labels_, cmap='rainbow')
plt1.scatter(kmeans.cluster_centers_[:,0] ,kmeans.cluster_centers_[:,1],
color='black')
```

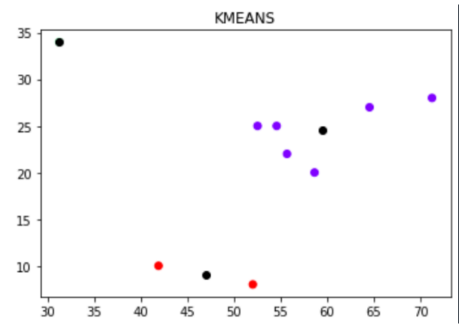
DATASET

Driver_id	Distance_Feature	Speeding_Feature
3.42E+09	71.24	28
3.42E+09	52.53	25
3.42E+09	64.54	27
3.42E+09	55.69	22
3.42E+09	54.58	25
3.42E+09	41.91	10
3.42E+09	58.64	20
3.42E+09	52.02	8
3.42E+09	31.25	34

OUTPUT:-







EM predictions
[2 2 2 2 2 0 2 0 1]
mean:
 [[46.965   9.   ]
 [31.25   34.   ]
 [59.53666666 24.5   ]]

Covariances
 [[[2.55530260e+01 -5.05500000e+00]
  [-5.05500000e+00 1.00000100e+00]]

 [[1.00000000e-06 5.55358077e-27]
  [5.55358077e-27 1.00000000e-06]]

 [[4.18773566e+01 1.01900001e+01]
  [1.01900001e+01 7.58333439e+00]]]
[[71.24 28.]
 [52.53 25.]
 [64.54 27.]
 [55.69 22.]
 [54.58 25.]
 [41.91 10.]
 [58.64 20.]
 [52.02 8.]
 [31.25 34.]]

```
[[59.53666667 24.5       ]
 [31.25       34.        ]
 [46.965       9.        ]]
[0 0 0 0 0 2 0 2 1]
```

8:-KNN

```
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import classification_report, confusion_matrix
from sklearn.metrics import accuracy_score
from sklearn import datasets
iris = datasets.load_iris()
iris_data = iris.data
iris_labels = iris.target
x_train, x_test, y_train, y_test = train_test_split(iris_data, iris_labels,
test_size=0.20)
classifier = KNeighborsClassifier(n_neighbors=5)
classifier.fit(x_train, y_train)
y_pred = classifier.predict(x_test)
print('Confusion matrix is as follows')
print(confusion_matrix(y_test, y_pred))
print('Accuracy Metrics')
print(classification_report(y_test, y_pred))
print("correct prediction",accuracy_score(y_test, y_pred))
print("wrong prediction",(1-accuracy_score(y_test, y_pred)))
```

OUTPUT:-

Confusion matrix is as follows

```
[[ 8  0  0]
 [ 0 13  0]
 [ 0  0  9]]
```

Accuracy Metrics

```
              precision    recall  f1-score   support

           0       1.00      1.00      1.00         8
           1       1.00      1.00      1.00        13
           2       1.00      1.00      1.00         9

    accuracy                           1.00        30
   macro avg       1.00      1.00      1.00        30
weighted avg       1.00      1.00      1.00        30
```

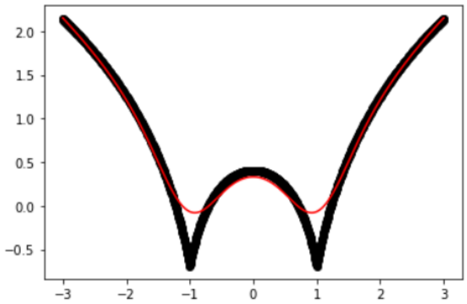
correct prediction 1.0

wrong prediction 0.0

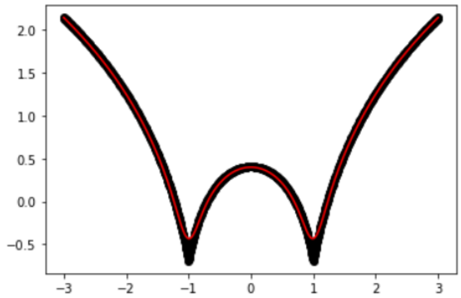
9:- REGRESSION

```
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.asarray(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)
```

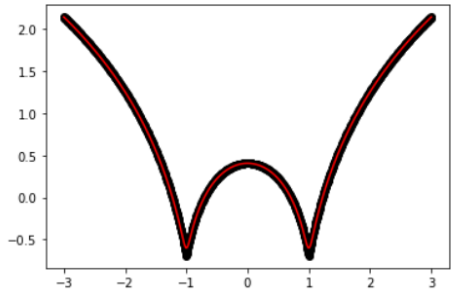
OUTPUT



2.0
1.5
1.0
0.5
0.0
−0.5
−3 −2 −1 0 1 2 3



2.0
1.5
1.0
0.5
0.0
−0.5
−3 −2 −1 0 1 2 3



2.0
1.5
1.0
0.5
0.0
−0.5
−3 −2 −1 0 1 2 3