Program 1

Implement and demonstrate the FIND-S algorithm for finding the most specifichypothesis based on a given set of training data samples. Read the training data from a .CSVfile.

Find-S Program:

```
import numpy as np
import pandas as pd

# Loading Data from a CSV File
data = pd.DataFrame(data=pd.read_csv('finds.csv'))


# Separating concept features from Target
concepts = np.array(data.iloc[:,0:-1])

# Isolating target into a separate DataFrame
target = np.array(data.iloc[:,-1])
data

# target or concept can be seen for output
def learn(concepts, target):

 ''' learn() function implements the learning method of the Candidate elimination algorithm.

   Arguments:
   concepts - a data frame with all the features
   target - a data frame with corresponding output values'''


   # Initialise S0 with the first instance from concepts
   # .copy() makes sure a new list is created instead of just pointing to the same memory
location

specific_h = concepts[0].copy()

   # The learning iterations
   for i, h in enumerate(concepts):

# Checking if the hypothesis has a positive target
      if target[i] == "Yes":
        for x in range(len(specific_h)):
            if h[x] != specific_h[x]:
                # If the value in the corresponding index has changed, replace it with a ?
                specific_h[x] = "?"

# returning the final value
    return specific_h
specific_h = learn(concepts, target)
print(specific_h)
```

<u>Program 2</u>

Foragivensetoftrainingdataexamplesstoredina.CSVfile,implementanddemonstratethe Candidate-Eliminationalgorithmtooutputadescriptionofthesetofallhypothesesconsistent with the trainingexamples.

Candidate Elimination Program:

```
import numpy as np
import pandas as pd

# Loading Data from a CSV File
data = pd.DataFrame(data=pd.read_csv('finds.csv'))

# Separating concept features from Target
concepts = np.array(data.iloc[:,0:-1])

# Isolating target into a separate DataFrame
target = np.array(data.iloc[:,-1])

def learn(concepts, target):

#  learn() function implements the learning method of the Candidate elimination algorithm.
  #Arguments:
  #concepts - a data frame with all the features ,target - a data frame with corresponding
output values
  # Initialise S0 with the first instance from concepts
  # .copy() makes sure a new list is created instead of just pointing to the same memory
location

    specific_h = concepts[0].copy()

  general_h = [["?" for i in range(len(specific_h))] for i in range(len(specific_h))]

# The learning iterations
    for i, h in enumerate(concepts):

# Checking if the hypothesis has a positive target
        if target[i] == "Yes":
            for x in range(len(specific_h)):

# Change values in S & G only if values change
                if h[x] != specific_h[x]:
                    specific_h[x] = '?'
                    general_h[x][x] = '?'

        # Checking if the hypothesis has a positive target
        if target[i] == "No":
            for x in range(len(specific_h)):

# For negative hyposthesis change values only  in G
                if h[x] != specific_h[x]:
                    general_h[x][x] = specific_h[x]
                else:
                    general_h[x][x] = '?'

# find indices where we have empty rows, meaning those that are unchanged
    indices = [i for i,val in enumerate(general_h) if val == ['?', '?', '?', '?', '?', '?']]
    for i in indices:
# remove those rows from general_h
        general_h.remove(['?', '?', '?', '?', '?', '?'])

    # Return final values
    return specific_h, general_h
s_final, g_final = learn(concepts, target)
print("Final S:", s_final, sep="\n")
print("Final G:", g_final, sep="\n")
```

Input:

ce.csv

Output:

```
Final S:
['Sunny' 'Warm' '?' 'Strong' '?' '?']


Final G:
[['Sunny', '?', '?', '?', '?', '?'], ['?', 'Warm', '?', '?', '?', '?']]
```

<u>Program 3</u>

Writeaprogramtodemonstratetheworkingofthedecisiontreebased ID3algorithm.Usean appropriatedatasetforbuildingthedecisiontreeandapplythisknowledgetoclassifyanew sample.

Task: ID3 determines the information gain for each candidate attribute, then selects the one with highest information gain as the rootnodeofthetree.The informationgainvaluesforallfourattributesarecalculatedusingthe followingformula:

Entropy(S)=∑-P(I).log2P(I)

Gain(S,A)=Entropy(S)-∑[P(S/A).Entropy(S/A)]

Dataset: Weather.csv

<u>ID3 Program:</u>

```
import numpy as np
import pandas as pd
from pprint import pprint

data = pd.read_csv("playTennis.csv")
data_size= len(data)

treenodes = []
tree = {"ROOT": data}

def total_entropy(data, col):
    mydict = {}
    for elem in data[col]:
        if elem in mydict.keys():
            mydict[elem] += 1
        else:
            mydict[elem] = 1
    total = sum(mydict.values())
    E = 0
    for key in mydict.keys():
        E += entropy(mydict[key], total)
    return E

def entropy(num, denom):
    return -(num/denom) * np.log2(num/denom)

def get_sorted_data(data, column):
    sort = {}
    for column_name in get_attributes(data, column):
        sort[column_name] = data.loc[data[column]==column_name]
    return sort

def get_attributes(data, column):
    return data[column].unique().tolist()

def InfoGain(total_entropy, sorted_data, entropy_by_attribute):
    length = data_size
    total = 0
    for col, df in sorted_data.items():
        total += (len(df) / length) * entropy_by_attribute[col]
    return total_entropy - total

def get_entropy_by_attribute(sorted_data):
    entropies = {}
    for key, df in sorted_data.items():
        entropies[key] = total_entropy(df, 'Decision')
    return entropies

def drop_node(data, column):
    return data.drop(column, axis=1)

def id3(tree):
    for branch, data in tree.items():

        # Make sure it's a DataFrame
        if not isinstance(data, pd.DataFrame):
            continue

        #Fetch column names so you can use them to iterate later
        columns = data.columns

        # Calculate the Entropy for the entire dataset
        total_entropy_for_data = total_entropy(data.values, -1)

        # If only one column is left, it means we're done.
        if len(columns) == 1:
            break


        # Keep track of information gain to choose the attribute with maximum info gain.
        info_gain_list = []

        # Now iterate over each column to calculate information gain w.r.t o/p column
        for i in range(0, len(data.columns)-1):

            # Sort the rows w.r.t o/p
            sorted_rows = get_sorted_data(data, columns[i])

            # Calculate the entropy w.r.t to each attribute based on sorted columns
            entropy_by_attribute = get_entropy_by_attribute(sorted_rows)

            # get the info gain
            info_gain = InfoGain(total_entropy_for_data, sorted_rows, entropy_by_attribute)

            # save it
            info_gain_list.append(info_gain)

        # Find index of max info gain
        node = info_gain_list.index(max(info_gain_list))

        # sort the data into branches based on the new node
        branches = get_sorted_data(data, columns[node])

        # If we've reached the end of iterations, just assign the value, else drop the sorted
column
        for attr, df in branches.items():
            if (total_entropy(df, columns[-1]) == 0):
                branches[attr] = df.iloc[0,-1]
            else:
                branches[attr] = df.drop(columns[node], axis=1)

        # Keep track of nodes already done
        treenodes.append(columns[node])
        # add the new branches to the tree
        child = {columns[node]: {}}
        tree[branch] = child
        tree[branch][columns[node]] = branches

        # ID3
        id3(tree[branch][columns[node]])

x = id3(tree)

pprint(tree, depth=5)
```

Program4:

Build an Artificial Neural Network by implementing the Back propagation algorithmand test the same using appropriate dataset

Program:

```
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) # maximum of X array longitudinally
y = y/100

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

#Derivative of Sigmoid Function
def derivatives_sigmoid (x):
        return x*(1-x)

#Variable initialization
epoch=7000 #Setting training iterations
lr=0.1 #Setting learning rate
inputlayer_neurons = 2 #number of features in data set
hiddenlayer_neurons = 3 #number of hidden layers neurons
output_neurons = 1 #number of neurons at output layer

#weight and bias initialization
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))

#draws a random range of numbers uniformly of dim x*y
for i in range(epoch):

#Forward Propogation
        hinp1=np.dot(X,wh)
        hinp=hinp1 + bh
        hlayer_act = sigmoid(hinp)
        outinp1=np.dot(hlayer_act,wout)
        outinp= outinp1+ bout
        output = sigmoid(outinp)

#Backpropagation
        EO = y-output
        outgrad = derivatives_sigmoid(output)
        d_output = EO* outgrad
        EH = d_output.dot(wout.T)
        hiddengrad = derivatives_sigmoid(hlayer_act)#how much hidden layer wts contributed
to error
        d_hiddenlayer = EH * hiddengrad
        wout += hlayer_act.T.dot(d_output) *lr# dotproduct of nextlayererror and currentlayerop
        # bout += np.sum(d_output, axis=0,keepdims=True) *lr
        wh += X.T.dot(d_hiddenlayer) *lr
        #bh += np.sum(d_hiddenlayer, axis=0,keepdims=True) *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:
[[ 0.92]
[ 0.86]
[ 0.89]]

Predicted Output:
[[ 0.90224607]
[ 0.87341151]
[ 0.8925683 ]]
```

<u>Program 5</u>

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.

<u>Naïve Bayes Program</u>:

```
import csv
import random
import math

def loadCsv(filename):
    lines = csv.reader(open(filename, "r"))
    dataset = list(lines)
    for i in range(len(dataset)):
        dataset[i] = [float(x) for x in dataset[i]]
    return dataset

def splitDataset(dataset, splitRatio):
    trainSize = int(len(dataset) * splitRatio)
    trainSet = []
    copy = list(dataset)
    while len(trainSet) < trainSize:
        index = random.randrange(len(copy))
        trainSet.append(copy.pop(index))
    return [trainSet, copy]

def separateByClass(dataset):
    separated = {}
    for i in range(len(dataset)):
        vector = dataset[i]
        if (vector[-1] not in separated):
            separated[vector[-1]] = []
        separated[vector[-1]].append(vector)

    return separated

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

def summarizeByClass(dataset):
    separated = separateByClass(dataset)
    summaries = {}
    for classValue, instances in separated.items():
        summaries[classValue] = summarize(instances)
    return summaries

def calculateProbability(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 calculateClassProbabilities(summaries, inputVector):
    probabilities = {}
    for classValue, classSummaries in summaries.items():
        probabilities[classValue] = 1
        for i in range(len(classSummaries)):
            mean, stdev = classSummaries[i]
            x = inputVector[i]
            probabilities[classValue] *= calculateProbability(x, mean, stdev)
    return probabilities

def predict(summaries, inputVector):
    probabilities = calculateClassProbabilities(summaries, inputVector)
    bestLabel, bestProb = None, -1
    for classValue, probability in probabilities.items():
        if bestLabel is None or probability > bestProb:
            bestProb = probability
            bestLabel = classValue
    return bestLabel

def getPredictions(summaries, testSet):
    predictions = []
    for i in range(len(testSet)):
        result = predict(summaries, testSet[i])
```

```
                predictions.append(result)

        return predictions

def getAccuracy(testSet, predictions):
    correct = 0
    for i in range(len(testSet)):
        #print(testSet[i][-1]," ",predictions[i])
        if testSet[i][-1] == predictions[i]:
            correct += 1

        return (correct/float(len(testSet))) * 100.0

def main():
        filename = 'pima-indians-diabetes.data.csv'
        splitRatio = 0.67
        dataset = loadCsv(filename)
        trainingSet,testSet=splitDataset(dataset, splitRatio) #dividing into training and test data
        #trainingSet = dataset #passing entire dataset as training data
        #testSet=[[8.0,183.0,64.0,0.0,0.0,23.3,0.672,32.0]]
        print('Split {0} rows into train={1} and test={2} rows'.format(len(dataset),
len(trainingSet), len(testSet)))
        # prepare model
        summaries = summarizeByClass(trainingSet)
        # test model
        predictions = getPredictions(summaries, testSet)
        accuracy = getAccuracy(testSet, predictions)
        print('Accuracy: {0}%'.format(accuracy))

main()
```

Program 6

Assuming a set of documents that need to be classified, use the naïve Bayesian Classifier model to perform this task. Built-in Java classes/API can be used to write the program. Calculate the accuracy, precision, and recall for your data set

Program

```
import pandas as pd

msg=pd.read_csv('6pg.csv',names=['message','label'])
print('The dimensions of the dataset',msg.shape)
msg['labelnum']=msg.label.map({'pos':1,'neg':0})
X=msg.message
y=msg.labelnum
print(X)
print(y)

#splitting the dataset into train and test data
from sklearn.model_selection import train_test_split
xtrain,xtest,ytrain,ytest=train_test_split(X,y)
print(xtest.shape)
print(xtrain.shape)
print(ytest.shape)
print(ytrain.shape)

#output of count vectoriser is a sparse matrix
```

```
from sklearn.feature_extraction.text import CountVectorizer
count_vect = CountVectorizer()
xtrain_dtm = count_vect.fit_transform(xtrain)
xtest_dtm=count_vect.transform(xtest)
print(count_vect.get_feature_names())
df=pd.DataFrame(xtrain_dtm.toarray(),columns=count_vect.get_feature_names())
print(df)#tabular representation
print(xtrain_dtm) #sparse matrix representation

# Training Naive Bayes (NB) classifier on training data.
from sklearn.naive_bayes import MultinomialNB
clf = MultinomialNB().fit(xtrain_dtm,ytrain)
predicted = clf.predict(xtest_dtm)

#printing accuracy metrics

from sklearn import metrics
print('Accuracy metrics')
print('Accuracy of the classifer is',metrics.accuracy_score(ytest,predicted))
print('Confusion matrix')
print(metrics.confusion_matrix(ytest,predicted))
print('Recall and Precison ')
print(metrics.recall_score(ytest,predicted))
print(metrics.precision_score(ytest,predicted))
```

Program 7

WriteaprogramtoconstructaBayesiannetworkconsideringmedicaldata.Usethismodelto demonstratethediagnosisofheartpatientsusingstandardHeartDiseaseDataSet.Youcanuse Java/Python ML libraryclasses/API.

Bayesian Network Program

```
import bayespy
asbp import
numpy asnp
import csv
from colorama import init
from colorama import Fore, Back,
Style
 init()

# Define Parameter Enumvalues
#Age
ageEnum = {'SuperSeniorCitizen':0, 'SeniorCitizen':1, 'MiddleAged':2,
'Youth':3,'Teen':4}

# Gender
genderEnum = {'Male':0,'Female':1}

#FamilyHistory
familyHistoryEnum = {'Yes':0,'No':1}

# Diet(CalorieIntake)
dietEnum = {'High':0, 'Medium':1,'Low':2}


#LifeStyle
lifeStyleEnum = {'Athlete':0, 'Active':1, 'Moderate':2,'Sedetary':3}

#Cholesterol
cholesterolEnum = {'High':0, 'BorderLine':1,'Normal':2}
```

```
  #HeartDisease
  heartDiseaseEnum = {'Yes':0,'No':1}

  #heart_disease_data.csv
with open('heart_disease_data.csv') as  csvfile: lines =csv.reader(csvfile)
dataset =list(lines) data =[]
    for x indataset:

    data.append([ageEnum[x[0]],genderEnum[x[1]],familyHistoryEnum[x[2]],dietEnum[x[3]]
  ,lifeStyl eEnum[x[4]],cholesterolEnum[x[5]],heartDiseaseEnum[x[6]]])

  # Training data for machine learning todo: should import fromcsv data
  =np.array(data)
  N =len(data)

  # Input data columnassignment
  p_age =p.nodes.Dirichlet(1.0*np.ones(5))
  age = bp.nodes.Categorical(p_age,plates=(N,)) age.observe(data[:,0])

  p_gender =bp.nodes.Dirichlet(1.0*np.ones(2)) gender =
  bp.nodes.Categorical(p_gender,plates=(N,)) gender.observe(data[:,1])

  p_familyhistory =bp.nodes.Dirichlet(1.0*np.ones(2)) familyhistory =
  bp.nodes.Categorical(p_familyhistory,plates=(N,)) familyhistory.observe(data[:,2])

  p_diet =bp.nodes.Dirichlet(1.0*np.ones(3)) diet = bp.nodes.Categorical(p_diet,plates=(N,))
  diet.observe(data[:,3])

  p_lifestyle =bp.nodes.Dirichlet(1.0*np.ones(4)) lifestyle =
  bp.nodes.Categorical(p_lifestyle,plates=(N,)) lifestyle.observe(data[:,4])

  p_cholesterol =bp.nodes.Dirichlet(1.0*np.ones(3)) cholesterol =
  bp.nodes.Categorical(p_cholesterol,plates=(N,)) cholesterol.observe(data[:,5])

  # Prepare nodes and establishedges
  # np.ones(2) ->  HeartDisease has 2 optionsYes/No
  # plates(5, 2, 2, 3, 4, 3)  ->  corresponds to options present for domainvalues
  p_heartdisease = bp.nodes.Dirichlet(np.ones(2), plates=(5, 2, 2, 3, 4,3))
  heartdisease = bp.nodes.MultiMixture([age, gender, familyhistory, diet,
  lifestyle,cholesterol], bp.nodes.Categorical,p_heartdisease)
  heartdisease.observe(data[:,6])p_heartdisease.update()
```

```
# Sample Test with hardcodedvalues
#print("SampleProbability")
#print("Probability(HeartDisease|Age=SuperSeniorCitizen,
Gender=Female,FamilyHistory=Yes, DietIntake=Medium,
LifeStyle=Sedetary,Cholesterol=High)")
#print(bp.nodes.MultiMixture([ageEnum['SuperSeniorCitizen'],genderEnum['Female'],
familyHistoryEnum['Yes'], dietEnum['Medium'],lifeStyleEnum['Sedetary'],
cholesterolEnum['High']],bp.nodes.Categorical,
p_heartdisease).get_moments()[0][heartDiseaseEnum['Yes']])

# InteractiveTest m =0
while m ==0:
   print("\n")
   res=bp.nodes.MultiMixture([int(input('EnterAge:'+str(ageEnum))),int(input('EnterGe
   nder:'
+ str(genderEnum))), int(input('Enter FamilyHistory: ' +str(familyHistoryEnum))),
int(input('Enter dietEnum: ' + str(dietEnum))), int(input('Enter LifeStyle: '
+str(lifeStyleEnum))), int(input('Enter Cholesterol: ' +
str(cholesterolEnum)))],bp.nodes.Categorical,
p_heartdisease).get_moments()[0][heartDiseaseEnum['Yes']]
   print("Probability(HeartDisease) = " +str(res))
   #print(Style.RESET_ALL)
   m = int(input("Enter for Continue:0, Exit :1"))
```

<u>Program8</u>

ApplyEMalgorithmtoclusterasetofdatastoredina.CSVfile.Usethesamedatasetfor clusteringusingk-Meansalgorithm.Comparetheresultsofthesetwoalgorithmsandcomment on the quality of clustering. You can add Java/Python ML library classes/API in theprogram.

Introduction to Expectation-Maximization(EM)

Dataset:iris.csv

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

 # import some data to play with
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']

 # Build the K Means Model
model = KMeans(n_clusters=3)
model.fit(X)

# model.labels_ : Gives cluster no for which samples belongs to Visualise the clustering
results

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

 # Plot the Original Classifications using Petal features
plt.subplot(2, 2, 1)
plt.scatter(X.Petal_Length, X.Petal_Width, c=colormap[y.Targets], s=40)
plt.title('Real Clusters')
plt.xlabel('Petal Length')
plt.ylabel('Petal Width')

 # Plot the Models Classifications
```

```
plt.subplot(2, 2, 2)
plt.scatter(X.Petal_Length, X.Petal_Width, c=colormap[model.labels_], s=40)
plt.title('K-Means Clustering')
plt.xlabel('Petal Length')
plt.ylabel('Petal Width')

# General EM for GMM
from sklearn import preprocessing

# transform your data such that its distribution will have a
mean value 0 and standard deviation of 1.
scaler = preprocessing.StandardScaler()
scaler.fit(X)
xsa = scaler.transform(X)
xs = pd.DataFrame(xsa, columns = X.columns)

from sklearn.mixture import GaussianMixture
gmm = GaussianMixture(n_components=3)
gmm.fit(xs)
gmm_y = gmm.predict(xs)


plt.subplot(2, 2, 3)
plt.scatter(X.Petal_Length, X.Petal_Width, c=colormap[gmm_y], s=40)
plt.title('GMM Clustering')
plt.xlabel('Petal Length')
plt.ylabel('Petal Width')

print('Observation: The GMM using EM algorithm based clustering matched the true labels
more closely than the Kmeans.')
```

Program9

Write a program to implement k-Nearest Neighbor algorithm to classify there is dataset. Print both correct and wrong predictions. Java/Python ML library classes can be used for this problem.

TASK: The task of this program is to classify the IRIS data set examples by using the k-Nearest Neighbour algorithm. The new instance has to be classified based on its k nearest neighbors.

1.

KNN Program

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

from sklearn.datasets import load_iris
data = load_iris()
df = pd.DataFrame(data.data, columns=data.feature_names)
df['Class'] = data.target_names[data.target]
df.head()
x = df.iloc[:, :-1].values
y = df.Class.values
print(x[:5])
print(y[:5])
from sklearn.model_selection import train_test_split
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size = 0.2)
from sklearn.neighbors import KNeighborsClassifier
knn_classifier = KNeighborsClassifier(n_neighbors=5)
knn_classifier.fit(x_train, y_train)
predictions = knn_classifier.predict(x_test)
print(predictions)
from sklearn.metrics import accuracy_score, confusion_matrix
print("Training accuracy Score is : ", accuracy_score(y_train, knn_classifier.predict(x_train)))
print("Testing accuracy Score is : ", accuracy_score(y_test, knn_classifier.predict(x_test)))
print("Training Confusion Matrix is : \n", confusion_matrix(y_train,
knn_classifier.predict(x_train)))
```

```
print("Testing Confusion Matrix is : \n", confusion_matrix(y_test,
knn_classifier.predict(x_test)))
```

Program10

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

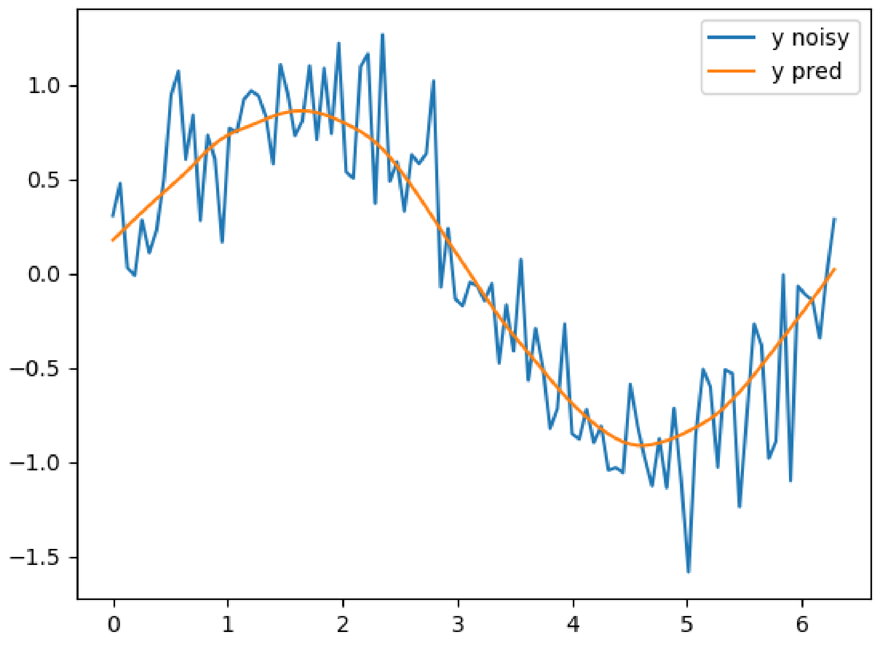
Locally Weighted Regression–

Locally Weighted Regression Program

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


def kernel(point,xmat, k):
    m,n = np.shape(xmat)
    weights = np.mat(np.eye((m))) # eye - identity matrix
    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

def graphPlot(X,ypred):
    sortindex = X[:,1].argsort(0) #argsort - index of the smallest
    xsort = X[sortindex][:,0]
    fig = plt.figure()
    ax = fig.add_subplot(1,1,1)
    ax.scatter(bill,tip, color='green')
    ax.plot(xsort[:,1],ypred[sortindex], color = 'red', linewidth=5)
    plt.xlabel('Total bill')
    plt.ylabel('Tip')
    plt.show();
# load data points
data = pd.read_csv('10data_tips.csv')
bill = np.array(data.total_bill) # We use only Bill amount and Tips data
tip = np.array(data.tip)
mbill = np.mat(bill) # .mat will convert nd array is converted in 2D array
mtip = np.mat(tip)
m= np.shape(mbill)[1]
one = np.mat(np.ones(m))
X = np.hstack((one.T,mbill.T)) # 244 rows, 2 cols
ypred = localWeightRegression(X,mtip,8) # increase k to get smooth curves
```

```
graphPlot(X,ypred)
```

Output


y noisy
y pred