program 1

```
import csv
#!usr/bin/python
#list creatin
hypo=['%','%','%','%','%','%'];

with open('Training_examples.csv') as csv_file:
     readcsv = csv.reader(csv_file, delimiter=',')
     print(readcsv)
     data = []
     print("\nThe given training examples are:")
     for row in readcsv:
          print(row)
          if row[len(row)-1].upper() == "YES":
               data.append(row)

print("\nThe positive examples are:");
for x in data:
     print(x);
print("\n");

TotalExamples = len(data);
i=0;
j=0;
k=0;
print("The steps of the Find-s algorithm are\n",hypo);
list = [];
p=0;
d=len(data[p])-1;
for j in range(d):
     list.append(data[i][j]);
hypo=list;
i=1;
for i in range(TotalExamples):
     for k in range(d):
          if hypo[k]!=data[i][k]:
               hypo[k]='?';
               k=k+1;

          else:
               hypo[k];
     print(hypo);
i=i+1;


print("\nThe maximally specific Find-s hypothesis for the given
training examples is");
list=[];
for i in range(d):
     list.append(hypo[i]);
print(list);
```

program 2

```
import numpy as np
import pandas as pd
# Loading Data from a CSV File
data = pd.DataFrame(data=pd.read_csv('Training_examples.csv'))


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

# Isolating target into a separate DataFrame
#copying last column to target array
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()
    print("initialization of specific_h and general_h")
    print(specific_h)

    general_h = [["?" for i in range(len(specific_h))] for i in
range(len(specific_h))]
    print(general_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] = '?'
        print(" steps of Candidate Elimination Algorithm",i+1)
        print(specific_h)
        print(general_h)
    # 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 Specific_h:", s_final, sep="\n")
print("Final General_h:", g_final, sep="\n")
```

program 4

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

# Initialize a network
def initialize_network(n_inputs, n_hidden,
n_outputs):#inputs,hidden layer,outputs
     network = list()
     print("network",network)
     hidden_layer = [{'weights':[random() for i in
range(n_inputs + 1)]} for i in range(n_hidden)]
     network.append(hidden_layer)
     print("network",network)
     output_layer = [{'weights':[random() for i in
range(n_hidden + 1)]} for i in range(n_outputs)]
     network.append(output_layer)
     print(network)
     return network

# Calculate neuron activation for an input
def activate(weights, inputs):
     activation = weights[-1]
     for i in range(len(weights)-1):
```

```
          activation += weights[i] *
inputs[i]#activation=w1*i1+w2*i2
     return activation

# Transfer neuron activation
def transfer(activation):
     return 1.0 / (1.0 + exp(-activation))

# Forward propagate input to a network output
def forward_propagate(network, row):
     inputs = row
     for layer in network:
          new_inputs = []
          for neuron in layer:
               activation = activate(neuron['weights'], inputs)
               neuron['output'] = transfer(activation)
               new_inputs.append(neuron['output'])
          inputs = new_inputs
     return inputs

# Calculate the derivative of an neuron output
def transfer_derivative(output):
     return output * (1.0 - output)

# Backpropagate error and store in neurons
def backward_propagate_error(network, expected):# Error is
calculated between expected ouput and
     for i in reversed(range(len(network))):#outputs forward
propagated
          layer = network[i]
          errors = list()
          if i != len(network)-1:
               for j in range(len(layer)):
                    error = 0.0
                    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] *
transfer_derivative(neuron['output'])

# Update network weights with error
def update_weights(network, row, l_rate):
     for i in range(len(network)):
          inputs = row[:-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] += l_rate *
neuron['delta'] * inputs[j]
                neuron['weights'][-1] += l_rate * neuron['delta']

# Train a network for a fixed number of epochs
def train_network(network, train, l_rate, n_epoch, n_outputs):
     for epoch in range(n_epoch):
          sum_error = 0
          for row in train:
                outputs = forward_propagate(network, row)
                expected = [0 for i in range(n_outputs)]
                expected[row[-1]] = 1
                sum_error += sum([(expected[i]-outputs[i])**2 for
i in range(len(expected))])
                backward_propagate_error(network, expected)
                update_weights(network, row, l_rate)
          print('>epoch=%d, lrate=%.3f, error=%.3f' % (epoch,
l_rate, sum_error))

# Test training backprop algorithm
seed(1)
dataset = [[2.7810836,2.550537003,0],#network input
values,target network output values,learning rate.
     [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]]
n_inputs = len(dataset[0]) - 1
print("data set length",n_inputs)
n_outputs = len(set([row[-1] for row in dataset]))
print("outputs",n_outputs)
network = initialize_network(n_inputs, 2, n_outputs)
train_network(network, dataset, 0.5, 20, n_outputs)
for layer in network:
     print(layer)
```

program 5

```
print("\nNaive Bayes Classifier for concept learning problem")
import csv
import random
import math
import operator
def safe_div(x,y):
    if y == 0:
        return 0
    return x / y

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

def splitDataset(dataset, splitRatio):
     trainSize = int(len(dataset) * splitRatio)
     trainSet = []
     copy = list(dataset)
     i=0
     while len(trainSet) < trainSize:
          #index = random.randrange(len(copy))

          trainSet.append(copy.pop(i))
     #print("trainset",trainSet)
     #print("---------------------------------")
     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 safe_div(sum(numbers),float(len(numbers)))

def stdev(numbers):
     avg = mean(numbers)
     variance = safe_div(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)]
     #print("mean and stddev",summaries)
     del summaries[-1]
     return summaries

def summarizeByClass(dataset):
     separated = separateByClass(dataset)

     summaries = {}
     for classValue, instances in separated.items():
            #print("instance",instances)
         summaries[classValue] = summarize(instances)
     #print("summaries group",summaries)
     return summaries

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

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)
          #print("calprob",probabilities)
     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)):
          if testSet[i][-1] == predictions[i]:
               correct += 1
     accuracy = safe_div(correct,float(len(testSet))) * 100.0
     return accuracy

def main():
     filename = 'ConceptLearning.csv'
     splitRatio = 0.75
     dataset = loadCsv(filename)
     trainingSet, testSet = splitDataset(dataset, splitRatio)
     print('Split {0} rows into'.format(len(dataset)))
     print("-----------------------------------------------")
     print('Number of Training data: ' +
(repr(len(trainingSet))))
     print("-----------------------------------------------")
     print('Number of Test Data: ' + (repr(len(testSet))))
     print("-----------------------------------------------")
     print("\nThe values assumed for the concept learning
attributes are\n")
     print("OUTLOOK=> Sunny=1 Overcast=2 Rain=3\nTEMPERATURE=>
Hot=1 Mild=2 Cool=3\nHUMIDITY=> High=1 Normal=2\nWIND=> Weak=1
Strong=2")
     print("TARGET CONCEPT:PLAY TENNIS=> Yes=10 No=5")
     print("-----------------------------------------------")

     print("\nThe Training set are:")
     for x in trainingSet:
          print(x)
     #print("-----------------------------------")
     print("\nThe Test data set are:")
     for x in testSet:
          print(x)
     print("-----------------------------------")
     print("\n")
     # prepare model
     summaries = summarizeByClass(trainingSet)

     # test model
     predictions = getPredictions(summaries, testSet)
     #print("predictions",predictions)
     #print("-------------------------------------------------
-------")
     actual = []
     for i in range(len(testSet)):
          vector = testSet[i]
          actual.append(vector[-1])
```

```
    # Since there are five attribute values, each attribute
constitutes to 20% accuracy. So if all attributes match with
predictions then 100% accuracy
    print('Actual values: {0}%'.format(actual))
    print('Predictions: {0}%'.format(predictions))
    accuracy = getAccuracy(testSet, predictions)
    print('Accuracy: {0}%'.format(accuracy))

main()
```

program 6

```
from sklearn.datasets import fetch_20newsgroups
from sklearn.metrics import confusion_matrix
from sklearn.metrics import classification_report
import numpy as np
categories =
['alt.atheism','soc.religion.christian','comp.graphics',
'sci.med']
twenty_train
=fetch_20newsgroups(subset='train',categories=categories,shuffle
=True)
#print("twenty_train",twenty_train)
twenty_test =
fetch_20newsgroups(subset='test',categories=categories,shuffle=T
rue)
print(len(twenty_train.data))
print("------------------------------")
print(len(twenty_test.data))
print("------------------------------")
#print("target names",twenty_train.target_names)
#print("----------------------------------------")
print("\n ".join(twenty_train.data[0].split("\n")))
print("----------------------------------------")
print("tt",twenty_train.target[0])
print("------------------------------")
from sklearn.feature_extraction.text import CountVectorizer
count_vect = CountVectorizer()
#print("count_vect",count_vect)
#print("------------------------------")
X_train_tf = count_vect.fit_transform(twenty_train.data)
#print("X_train_tf",X_train_tf)
from sklearn.feature_extraction.text import TfidfTransformer
tfidf_transformer = TfidfTransformer()
print("tfidf_transformer",tfidf_transformer)
#print("------------------------------")
X_train_tfidf = tfidf_transformer.fit_transform(X_train_tf)
```

```
#print("X_train_tfidf",X_train_tfidf)
X_train_tfidf.shape
from sklearn.naive_bayes import MultinomialNB
from sklearn.metrics import accuracy_score
from sklearn import metrics
mod = MultinomialNB()
print("mod",mod)
mod.fit(X_train_tfidf, twenty_train.target)
X_test_tf = count_vect.transform(twenty_test.data)
X_test_tfidf = tfidf_transformer.transform(X_test_tf)
predicted = mod.predict(X_test_tfidf)
print("Accuracy:", accuracy_score(twenty_test.target,
predicted))
print("accu-------------------------------------------")
print(classification_report(twenty_test.target,predicted,target_
names=twenty_test.target_names))
print("class-----------------------------------------------------
--------")

print("confusion matrix is
\n",metrics.confusion_matrix(twenty_test.target, predicted))
```

program 7

```
import bayespy as bp#BayesPy provides tools for Bayesian
inference with Python.
                    #importing Dirichlet and Categorical
import numpy as np
import csv
from colorama import init
from colorama import Fore, Back, Style
init()

# Define Parameter Enum values
#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(Calorie Intake)
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 in dataset:

data.append([ageEnum[x[0]],genderEnum[x[1]],familyHistoryEnum[x[
2]],dietEnum[x[3]],lifeStyleEnum[x[4]],cholesterolEnum[x[5]],hea
rtDiseaseEnum[x[6]]])
    #print("data",data)
# Training data for machine learning todo: should import from
csv
data = np.array(data)
#print(data)
N = len(data)
print(N)

# Input data column assignment
p_age = bp.nodes.Dirichlet(1.0*np.ones(5))#used to classify text
in a document to a particular topic.
#print(p_age)
age = bp.nodes.Categorical(p_age, plates=(N,))#a sequence of
unique values and no missing values
#print(age)
age.observe(data[:,0])
#print("age",age)

p_gender = bp.nodes.Dirichlet(1.0*np.ones(2))
gender = bp.nodes.Categorical(p_gender, plates=(N,))
#print(gender)
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 establish edges
# np.ones(2) ->  HeartDisease has 2 options Yes/No
# plates(5, 2, 2, 3, 4, 3)  ->  corresponds to options present
for domain values
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()
m = 0
while m == 0:
    print("\n")
    res = bp.nodes.MultiMixture([int(input('Enter Age: ' +
str(ageEnum))), int(input('Enter Gender: ' + 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))
    m = int(input("Enter for Continue:0, Exit :1  "))
```

program 8

```
import matplotlib.pyplot as plt
from sklearn import datasets
from sklearn.cluster import KMeans
import sklearn.metrics as sm
import pandas as pd
import numpy as np
#%matplotlib inline

l1 = [0,1,2]

def rename(s):
     l2 = []
     for i in s:
          if i not in l2:
               l2.append(i)

     for i in range(len(s)):
          pos = l2.index(s[i])
          s[i] = l1[pos]

     #print("values",s[i])
```

```
    return s

# import some data to play with
iris = datasets.load_iris()

print("\n IRIS DATA :",iris.data);
print("\n IRIS FEATURES :\n",iris.feature_names)
print("\n IRIS TARGET  :\n",iris.target)
print("\n IRIS TARGET NAMES:\n",iris.target_names)


# Store the inputs as a Pandas Dataframe and set the column
names
X = pd.DataFrame(iris.data)

#print(X)
X.columns =
['Sepal_Length','Sepal_Width','Petal_Length','Petal_Width']

#print(X.columns) #print("X:",x)
#print("Y:",y)
y = pd.DataFrame(iris.target)
y.columns = ['Targets']

# Set the size of the plot
plt.figure(figsize=(14,7))

# Create a colormap
colormap = np.array(['red', 'lime', 'black'])

# Plot Sepal
plt.subplot(1, 2, 1)#rows,column,index
plt.scatter(X.Sepal_Length,X.Sepal_Width, c=colormap[y.Targets],
s=40)
plt.title('Sepal')

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

print("Actual Target is:\n", iris.target)

# K Means Cluster
model = KMeans(n_clusters=3)
model.fit(X)

# Set the size of the plot
plt.figure(figsize=(14,7))

# Create a colormap
colormap = np.array(['red', 'lime', 'black'])
```

```
# Plot the Original Classifications
plt.subplot(1, 2, 1)
plt.scatter(X.Petal_Length, X.Petal_Width,
c=colormap[y.Targets], s=40)
plt.title('Real Classification')

# Plot the Models Classifications
plt.subplot(1, 2, 2)
plt.scatter(X.Petal_Length, X.Petal_Width,
c=colormap[model.labels_], s=40)
plt.title('K Mean Classification')
plt.show()

km = rename(model.labels_)
print("\nWhat KMeans thought: \n", km)
print("Accuracy of KMeans is ",sm.accuracy_score(y, km))
print("Confusion Matrix for KMeans is \n",sm.confusion_matrix(y,
km))

from sklearn import preprocessing#several common utility
functions and transformer classes to change

scaler = preprocessing.StandardScaler()#raw feature vectors into
a representation that is more suitable for the
                                        #downstream estimators.
scaler.fit(X)
xsa = scaler.transform(X)#scale:provides a quick and easy way to
perform this operation on a single array-like dataset:
xs = pd.DataFrame(xsa, columns = X.columns)
print("\nsample",xs.sample(5))

from sklearn.mixture import GaussianMixture  #Gaussian mixture
model probability distribution.
gmm = GaussianMixture(n_components=3)
gmm.fit(xs)

y_cluster_gmm = gmm.predict(xs)

plt.subplot(1, 2, 1)
plt.scatter(X.Petal_Length, X.Petal_Width,
c=colormap[y_cluster_gmm], s=40)
plt.title('GMM Classification')
plt.show()

em = rename(y_cluster_gmm)
print("\nWhat EM thought: \n", em)
print("Accuracy of EM is ",sm.accuracy_score(y, em))
print("Confusion Matrix for EM is \n", sm.confusion_matrix(y,
em))
```

program 9

```
# Python program to demonstrate # KNN classification algorithm #
on IRISdataset
from sklearn.datasets import load_iris
from sklearn.neighbors import KNeighborsClassifier
import numpy as np
from sklearn.model_selection import train_test_split
iris_dataset=load_iris()
print("\n IRIS FEATURES \ TARGET NAMES: \n ",
iris_dataset.target_names)
for i in range(len(iris_dataset.target_names)):

print("\n[{0}]:[{1}]".format(i,iris_dataset.target_names[i]))
print("\n IRIS DATA :\n",iris_dataset["data"])
X_train, X_test, y_train, y_test =
train_test_split(iris_dataset["data"], iris_dataset["target"],
random_state=0)
print("\n Target :\n",iris_dataset["target"])
print("\n X TRAIN \n", X_train)
print("\n X TEST \n", X_test)
print("\n Y TRAIN \n", y_train)
print("\n Y TEST \n", y_test)
kn = KNeighborsClassifier(n_neighbors=1)
kn.fit(X_train, y_train)
x_new = np.array([[5, 2.9, 1, 0.2]])
print("\n XNEW \n",x_new)
prediction = kn.predict(x_new)
print("\n Predicted target value: {}\n".format(prediction))
print("\n Predicted feature name:
{}\n".format(iris_dataset["target_names"][prediction]))
i=1
x= X_test[i]
x_new = np.array([x])
print("\n XNEW \n",x_new)
for i in range(len(X_test)): x = X_test[i]
x_new = np.array([x])
prediction = kn.predict(x_new)
print("\n Actual : {0} {1}, Predicted
:{2}{3}".format(y_test[i],iris_dataset["target_names"][y_test[i]
],prediction,iris_dataset["target_names"][ prediction]))

print("\n TEST SCORE[ACCURACY]:
{:.2f}\n".format(kn.score(X_test, y_test))
)
```

program 10

```
from math import ceil
import numpy as np
from scipy import linalg
def lowess(x, y, f=2./3., iter=3):
    n = len(x)
    r = int(ceil(f*n))
    h = [np.sort(np.abs(x - x[i]))[r] for i in range(n)]
    w = np.clip(np.abs((x[:,None] - x[None,:]) / h), 0.0, 1.0)
    w = (1 - w**3)**3
    yest = np.zeros(n)
    delta = np.ones(n)
    for iteration in range(iter):
        for i in range(n):
            weights = delta * w[:,i]
            b = np.array([np.sum(weights*y),
np.sum(weights*y*x)])
            A = np.array([[np.sum(weights), np.sum(weights*x)],
                   [np.sum(weights*x), np.sum(weights*x*x)]])
            beta = linalg.solve(A, b)
            yest[i] = beta[0] + beta[1]*x[i]
        residuals = y - yest
        s = np.median(np.abs(residuals))
        delta = np.clip(residuals / (6.0 * s), -1, 1)
        delta = (1 - delta**2)**2
    return yest
if __name__ == '__main__':
    import math
    n = 100
    x = np.linspace(0, 2 * math.pi, n)
    print("==========================values of
x====================")
    print(x)
    y = np.sin(x) + 0.3*np.random.randn(n)
    print("================================Values of
y==================")
    print(y)
    f = 0.25
    yest = lowess(x, y, f=f, iter=3)
    import pylab as pl
    pl.clf()
    pl.plot(x, y, label='y noisy')
    pl.plot(x, yest, label='y pred')
    pl.legend()
    pl.show()
from math import ceil
import numpy as np
from scipy import linalg
def lowess(x, y, f=2./3., iter=3):
    n = len(x)
```

```
    r = int(ceil(f*n))
    h = [np.sort(np.abs(x - x[i]))[r] for i in range(n)]
    w = np.clip(np.abs((x[:,None] - x[None,:]) / h), 0.0, 1.0)
    w = (1 - w**3)**3
    yest = np.zeros(n)
    delta = np.ones(n)
    for iteration in range(iter):
        for i in range(n):
            weights = delta * w[:,i]
            b = np.array([np.sum(weights*y),
np.sum(weights*y*x)])
            A = np.array([[np.sum(weights), np.sum(weights*x)],
                   [np.sum(weights*x), np.sum(weights*x*x)]])
            beta = linalg.solve(A, b)
            yest[i] = beta[0] + beta[1]*x[i]
        residuals = y - yest
        s = np.median(np.abs(residuals))
        delta = np.clip(residuals / (6.0 * s), -1, 1)
        delta = (1 - delta**2)**2
    return yest
if __name__ == '__main__':
    import math
    n = 100
    x = np.linspace(0, 2 * math.pi, n)
    print("==========================values of
x====================")
    print(x)
    y = np.sin(x) + 0.3*np.random.randn(n)
    print("================================Values of
y==================")
    print(y)
    f = 0.25
    yest = lowess(x, y, f=f, iter=3)
    import pylab as pl
    pl.clf()
    pl.plot(x, y, label='y noisy')
    pl.plot(x, yest, label='y pred')
    pl.legend()
    pl.show()
```