MACHINE LEARNING LABORATORY

[As per Choice Based Credit System (CBCS) scheme]

SEMESTER – VII **Subject Code 15CSL76**

Course objectives:

This course will enable students to

1. Make use of Data sets in implementing the machine learning algorithms
2. Implement the machine learning concepts and algorithms in any suitable language of choice

Description (If any):

1. The programs can be implemented in either JAVA or Python.
2. For Problems 1 to 6 and 10, programs are to be developed without using the built-in classes or APIs of Java/Python.
3. Data sets can be taken from standard repositories (https://archive.ics.uci.edu/ml/datasets.html) or constructed by the students.

Lab Experiments:

1. Implement and demonstrate the FIND-S algorithm for finding the most specific hypothesis based on a given set of training data samples. Read the training data from a .CSV file.
2. For a given set of training data examples stored in a .CSV file, implement and demonstrate the Candidate-Elimination algorithm to output a description of the set of all hypotheses consistent with the training examples.
3. Write a program to demonstrate the working of the decision tree based ID3 algorithm. Use an appropriate data set for building the decision tree and apply this knowledge to classify a new sample.
4. Build an Artificial Neural Network by implementing the Back propagation algorithm and test the same using appropriate data sets.
5. 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.
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.

7. Write a program to construct a Bayesian network considering medical data. Use this model to demonstrate the diagnosis of heart patients using standard Heart Disease Data Set. You can use Java/Python ML library classes/API.
8. Apply EM algorithm to cluster a set of data stored in a .CSV file. Use the same data set for clustering using k-Means algorithm. Compare the results of these two algorithms and comment on the quality of clustering. You can add Java/Python ML library classes/API in the program.
9. Write a program to implement k-Nearest Neighbour algorithm to classify the iris data set. Print both correct and wrong predictions. Java/Python ML library classes can be used for this problem.
10. Implement the non-parametric Locally Weighted Regression algorithm in order to fit data points. Select appropriate data set for your experiment and draw graphs.

Course outcomes:

The students should be able to:

1. Understand the implementation procedures for the machine learning algorithms.
2. Design Java/Python programs for various Learning algorithms.
3. Apply appropriate data sets to the Machine Learning algorithms.
4. Identify and apply Machine Learning algorithms to solve real world problems.

Conduction of Practical Examination:

- All laboratory experiments are to be included for practical examination.
- Students are allowed to pick one experiment from the lot.
- Strictly follow the instructions as printed on the cover page of answer script
- Marks distribution: Procedure + Conduction + Viva:20 + 50 +10 (80)

Program 1

Implement and demonstrate the FIND-S algorithm for finding the most specific hypothesis based on a given set of training data samples. Read the training data from a .CSV file.

Task: **Find-S algorithm is used to find a maximally specific hypothesis.**

Dataset: Enjoy Sports Training Examples:

Example	Sky	AirTemp	Humidity	Wind	Water	Forecast	EnjoySport
1	Sunny	Warm	Normal	Strong	Warm	Same	Yes
2	Sunny	Warm	High	Strong	Warm	Same	Yes
3	Rainy	Cold	High	Strong	Warm	Change	No
4	Sunny	Warm	High	Strong	Cool	Change	Yes

Given:

- **Instances X**: Possible days, each described by the attributes
  - **Sky** (with possible values Sunny, Cloudy, and Rainy),
  - **AirTemp** (with values Warm and Cold),
  - **Humidity** (with values Normal and High),
  - **Wind** (with values Strong and Weak),
  - **Water** (with values Warm and Cool), and
  - **Forecast** (with values Same and Change).
- **Hypotheses H**: Each hypothesis is described by a conjunction of constraints on the attributes Sky, AirTemp, Humidity, Wind, Water, and Forecast. The constraints may be "?" (any value is acceptable), "0 (no value is acceptable), or a specific value.
- **Target concept c**: EnjoySport : X □ (0,l)
- Training examples D: Positive and negative examples of the target function

Find-S Algorithm

Step 1: Initialize h to the most specific hypothesis in H.

Step 2: For each positive training instance x.

- For each attribute constraint a_i in h
  - If the constraint a_i is satisfied by x Then
    do nothing
  - Else replace a_i in h by the next more general constraint that is satisfied by x.

Step 3: Output hypothesis h.

Find-S Program in Python:

```
import numpy as np
import pandas as pd
data = pd.read_csv('1finds.csv')
print(data)
concepts=data.iloc[:,0:-1].values
print("-----------------------------------------")
print (concepts)
print("-----------------------------------------")
target = data.iloc[:,-1].values
print (target)
def train(concepts,target):
    count=0
    specific_h = concepts[0]
    for i,h in enumerate(concepts):
        print(i)
        print(h)
        if target[i] == "Yes":
            for x in range(len(specific_h)):
                if h[x] == specific_h[x]:
                    pass
                else:
                    specific_h[x] = "?"
            count = count + 1
            print (f"Hypothesis after sample number:{count} processed: {specific_h}")
        else:
            pass
            count = count + 1
            print (f"Negative sample number:{count} Same Hypothesis:    {specific_h}")

    return specific_h


specific_h=train(concepts,target)
```

Input:

Finds.csv

Output:

```
[['Sunny' 'Warm' 'Normal' 'Strong' 'Warm' 'Same']
 ['Sunny' 'Warm' 'High' 'Strong' 'Warm' 'Same']
 ['Cloudy' 'Cold' 'High' 'Strong' 'Warm' 'Change']
 ['Sunny' 'Warm' 'High' 'Strong' 'Cool' 'Change']]

__________
['Yes' 'Yes' 'No' 'Yes']
0
['Sunny' 'Warm' 'Normal' 'Strong' 'Warm' 'Same']
Hypothesis after sample Number2 is ['Sunny' 'Warm' 'Normal' 'Strong' 'Warm'
'Same']
1
['Sunny' 'Warm' 'High' 'Strong' 'Warm' 'Same']
Hypothesis after sample Number3 is ['Sunny' 'Warm' '?' 'Strong' 'Warm' 'Sam
e']
2
['Cloudy' 'Cold' 'High' 'Strong' 'Warm' 'Change']
Negative Hypothesis after sample Number4 is ['Sunny' 'Warm' '?' 'Strong' 'W
arm' 'Same']
3
['Sunny' 'Warm' 'High' 'Strong' 'Cool' 'Change']
Hypothesis after sample Number5 is ['Sunny' 'Warm' '?' 'Strong' '?' '?']
```

Program 2

For a given set of training data examples stored in a .CSV file, implement and demonstrate the Candidate-Elimination algorithm to output a description of the set of all hypotheses consistent with the training examples.

Task: The CANDIDATE-ELIMINATION algorithm computes the version space containing all hypotheses from H that are consistent with an observed sequence of training examples.

Dataset: Enjoy Sports Training Examples:

Example	Sky	AirTemp	Humidity	Wind	Water	Forecast	EnjoySport
1	Sunny	Warm	Normal	Strong	Warm	Same	Yes
2	Sunny	Warm	High	Strong	Warm	Same	Yes
3	Rainy	Cold	High	Strong	Warm	Change	No
4	Sunny	Warm	High	Strong	Cool	Change	Yes

Candidate Elimination Algorithm:

Initialize G to the set of maximally general hypotheses in H
Initialize S to the set of maximally specific hypotheses in H
For each training example d, do

- If d is a positive example
  - Remove from G any hypothesis inconsistent with d
  - For each hypothesis s in S that is not consistent with d
    - Remove s from S
    - Add to S all minimal generalizations h of s such that
      - h is consistent with d, and some member of G is more general than h
    - Remove from S any hypothesis that is more general than another hypothesis in S
- If d is a negative example
  - Remove from S any hypothesis inconsistent with d
  - For each hypothesis g in G that is not consistent with d
    - Remove g from G
    - Add to G all minimal specializations h of g such that
      - h is consistent with d, and some member of S is more specific than h
    - Remove from G any hypothesis that is less general than another hypothesis in G

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")
```

out put

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


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

Program 3

Write a program to demonstrate the working of the decision tree based ID3 algorithm. Use an appropriate data set for building the decision tree and apply this knowledge to classify a new sample.

Task: ID3 determines the information gain for each candidate attribute, then selects the one with highest information gain as the root node of the tree. The information gain values for all four attributes are calculated using the following formula:

$$\text{Entropy}(S)=\sum - P(I).\log_2 P(I)$$

$$\text{Gain}(S,A)=\text{Entropy}(S)-\sum[P(S/A).\text{Entropy}(S/A)]$$

Dataset: pima-indians-diabetes.csv

ID3 Algorithm:

ID3(*Examples, Target_attribute, Attributes*)

Examples are the training examples. Target_attribute is the attribute whose value is to be predicted by the tree. Attributes is a list of other attributes that may be tested by the learned decision tree. Returns a decision tree that correctly classifies the given Examples.

- Create a *Root* node for the tree
- If all *Examples* are positive, Return the single-node tree *Root*, with label = +
- If all *Examples* are negative, Return the single-node tree *Root*, with label = -
- If *Attributes* is empty, Return the single-node tree *Root*, with label = most common value of *Target_attribute* in *Examples*
- Otherwise Begin

❖ A ← the attribute from *Attributes* that best* classifies *Examples*

❖ The decision attribute for *Root* ←A

❖ For each possible value, υ_i, of A,

- Add a new tree branch below *Root*, corresponding to the test A = υ_i
- Let *Examples*$_{\upsilon i}$,be the subset of *Examples* that have value υ_i for A
- If *Examples*$_{\upsilon i}$, is empty
  - ❖ Then below this new branch add a leaf node with label=most common value of *Target_attribute* in *Examples*
  - ❖ Else below this new branch add the subtree

ID3(*Examples*$_{\upsilon i}$,*Target_attribute, Attributes–{A}*))

End

Return Root

ID3 Program:

```
import csv
import math
import random

# Majority Function which tells which class has more entries in given data-set
def majorClass(attributes, data, target):
    freq = {}
    index = attributes.index(target)
    for tuple in data:
        if tuple[index] in freq:
            freq[tuple[index]] += 1
        else:
            freq[tuple[index]] = 1
    max = 0
    major = ""
    for key in freq.keys():
        if freq[key]>max:
            max = freq[key]
            major = key
    return major

# Calculates the entropy of the data given the target attribute
def entropy(attributes, data, targetAttr):
    freq = {}
    dataEntropy = 0.0
    i = 0
    for entry in attributes:
        if (targetAttr == entry):
            break
        i = i + 1
    i = i - 1
    for entry in data:
        if entry[i] in freq:
            freq[entry[i]] += 1.0
        else:
            freq[entry[i]]  = 1.0
    for freq in freq.values():
        dataEntropy += (-freq/len(data)) * math.log(freq/len(data), 2)
    return dataEntropy

# Calculates the information gain (reduction in entropy) in the data when a particular attribute
```

```
is chosen for splitting the data.
def info_gain(attributes, data, attr, targetAttr):
    freq = {}
    subsetEntropy = 0.0
    i = attributes.index(attr)
    for entry in data:
        if entry[i] in freq:
            freq[entry[i]] += 1.0
        else:
            freq[entry[i]]  = 1.0
    for val in freq.keys():
        valProb        = freq[val] / sum(freq.values())
        dataSubset     = [entry for entry in data if entry[i] == val]
        subsetEntropy += valProb * entropy(attributes, dataSubset, targetAttr)
    return (entropy(attributes, data, targetAttr) - subsetEntropy)

# This function chooses the attribute among the remaining attributes which has the maximum
information gain.
def attr_choose(data, attributes, target):
    best = attributes[0]
    maxGain = 0;
    for attr in attributes:
        newGain = info_gain(attributes, data, attr, target)
        if newGain>maxGain:
            maxGain = newGain
            best = attr
    return best


# This function will get unique values for that particular attribute from the given data
def get_values(data, attributes, attr):
    index = attributes.index(attr)
    values = []
    for entry in data:
        if entry[index] not in values:
            values.append(entry[index])
    return values

# This function will get all the rows of the data where the chosen "best" attribute has a value
"val"
def get_data(data, attributes, best, val):
    new_data = [[]]
```

```
    index = attributes.index(best)
    for entry in data:
        if (entry[index] == val):
            newEntry = []
            for i in range(0,len(entry)):
                if(i != index):
                    newEntry.append(entry[i])
            new_data.append(newEntry)
    new_data.remove([])
    return new_data

# This function is used to build the decision tree using the given data, attributes and the target
attributes. It returns the decision tree in the end.
def build_tree(data, attributes, target):
    data = data[:]
    vals = [record[attributes.index(target)] for record in data]
    default = majorClass(attributes, data, target)
    if not data or (len(attributes) - 1) <= 0:
        return default
    elif vals.count(vals[0]) == len(vals):
        return vals[0]
    else:
        best = attr_choose(data, attributes, target)
        tree = {best:{}}

        for val in get_values(data, attributes, best):
            new_data = get_data(data, attributes, best, val)
            newAttr = attributes[:]
            newAttr.remove(best)
            subtree = build_tree(new_data, newAttr, target)
            tree[best][val] = subtree
        return tree


#Main function
def execute_decision_tree():
    data = []

    #load file
    with open("weather.csv") as tsv:
        for line in csv.reader(tsv):
            data.append(tuple(line))
```

```
        print("Number of records:",len(data))

        #set attributes
        attributes=['outlook','temperature','humidity','wind','play']
        target = attributes[-1]

        #set training data
        acc = []
        training_set = [x for i, x in enumerate(data)]
        tree = build_tree( training_set, attributes, target )
        print(tree)

        #execute algorithm on test data
        results = []
        test_set = [('rainy','mild','high','strong')]
        for entry in test_set:
            tempDict = tree.copy()
            result = ""
            while(isinstance(tempDict, dict)):
                child=[]
                nodeVal=next(iter(tempDict))
                child=tempDict[next(iter(tempDict))].keys()
                tempDict = tempDict[next(iter(tempDict))]
                index = attributes.index(nodeVal)
                value = entry[index]
                if(value in tempDict.keys()):
                    result = tempDict[value]
                    tempDict = tempDict[value]
                else:
                    result = "Null"
                    break
            if result != "Null":
                results.append(result == entry[-1])
        print(result)

if __name__ == "__main__":
    execute_decision_tree()
```

Input:

Weather.csv

Output:

Number of records: 15

{'wind': {'wind': 'play', 'weak': {'humidity': {'high': {'temperature': {'hot': {'outlook': {'sunny': 'no', ' overcast': 'yes'}}, 'mild': {'outlook': {'rainy': 'yes', 'sunny': 'no'}}}}, 'normal': 'yes'}}, 'strong': {'hu midity': {'high': {'outlook': {'sunny': 'no', 'overcast': 'yes', 'rainy': 'no'}}, 'normal': {'outlook': {'rainy ': 'no', 'overcast': 'yes', 'sunny': 'yes'}}}}}}

no

Program 4:

Build an Artificial Neural Network by implementing the Back propagation algorithm and test the same using appropriate data set

The stochastic gradient descent version of the BACKPROPAGATION algorithm for feed forward networks containing two layers of sigmoid units.

Step 1: begins by constructing a network with the desired number of hidden and output units and initializing all network weights to small random values. . For each training example, it applies the network to the example, calculates the error of the network output for this example, computes the gradient with respect to the error on this example, then updates all weights in the network. This gradient descent step is iterated (often thousands of times, using the same training examples multiple times) until the network performs acceptably well.

Step 2: The gradient descent weight-update rule is similar to the delta training rule The only difference is that the error (t - o) in the delta rule is replaced by a more complex error term aj.

Step 3: updates weights incrementally, following the Presentation of each training example. This corresponds to a stochastic approximation to gradient descent. To obtain the true gradient of E one would sum the Sj, xji values over all training examples before altering weight values.

Step 4: The weight-update loop in BACKPROPAGATION may be iterated thousands of times in a typical application. A variety of termination conditions can be used to halt the procedure.

One may choose to halt after a fixed number of iterations through the loop, or once the error on the training examples falls below some threshold.

Dataset:

ANN Program:

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

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

```
range(n_outputs)]
        network.append(output_layer)
        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]
        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):
        for i in reversed(range(len(network))):
                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],
        [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
n_outputs = len(set([row[-1] for row in dataset]))
network = initialize_network(n_inputs, 2, n_outputs)
print(network)
train_network(network, dataset, 0.5, 20, n_outputs)
for layer in network:
        print(layer)
```

Output

```
>epoch=0, lrate=0.500, error=6.350
>epoch=1, lrate=0.500, error=5.531
>epoch=2, lrate=0.500, error=5.22147
>epoch=3, lrate=0.500, error=4.951
>epoch=4, lrate=0.500, error=4.519
>epoch=5, lrate=0.500, error=4.173
>epoch=6, lrate=0.500, error=3.835
>epoch=7, lrate=0.500, error=3.506
>epoch=8, lrate=0.500, error=3.192
>epoch=9, lrate=0.500, error=2.898
>epoch=10, lrate=0.500, error=2.626
>epoch=11, lrate=0.500, error=2.377
>epoch=12, lrate=0.500, error=2.153
>epoch=13, lrate=0.500, error=1.953
>epoch=14, lrate=0.500, error=1.774
>epoch=15, lrate=0.500, error=1.614
>epoch=16, lrate=0.500, error=1.472
>epoch=17, lrate=0.500, error=1.346
>epoch=18, lrate=0.500, error=1.233
>epoch=19, lrate=0.500, error=1.132
[{'weights': [-1.4688375095432327, 1.850887325439514, 1.0858178629550297], 'output':
0.029980305604426185, 'delta': -0.0059546604162323625}, {'weights':
[0.37711098142462157, -0.0625909894552989, 0.2765123702642716], 'output':
0.9456229000211323, 'delta':
0.0026279652850863837}]
[{'weights': [2.515394649397849, -0.3391927502445985, -0.9671565426390275], 'output':
0.23648794202357587, 'delta': -0.04270059278364587}, {'weights': [-2.5584149848484263,
1.0036422106209202, 0.42383086467582715], 'output': 0.7790535202438367, 'delta':
0.03803132596437354}]
```

Program 5

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.

Task: It is a classification technique based on Bayes' Theorem with an assumption of independence among predictors. In simple terms, a Naive Bayes classifier assumes that the presence of a particular feature in a class is unrelated to the presence of any other feature. For example, a fruit may be considered to be an apple if it is red, round, and about 3 inches in diameter. Even if these features depend on each other or upon the existence of the other features, all of these properties independently contribute to the probability that this fruit is an apple and that is why it is known as 'Naive'.

Dataset : Pima-indians-diabetes.csv

It is a classification technique based on Bayes" Theorem with an assumption of independence among predictors. In simple terms, a Naive Bayes classifier assumes that the presence of a particular feature in a class is unrelated to the presence of any other feature. For example, a fruit may be considered to be an apple if it is red, round, and about 3 inches in diameter. Even if these features depend on each other or upon the existence of the other features, all of these properties independently contribute to the probability that this fruit is an apple and that is why it is known as „Naive".

Bayes theorem provides a way of calculating posterior probability P(c|x) from P(c), P(x) and P(x|c). Look at the equation below:

1) Handling Of Data:

- Load the data from the CSV file and split in to training and test data set.
- Training data set can be used to by Naïve Bayes to make predictions.
- And Test data set can be used to evaluate the accuracy of the model.

2) Summarize Data:

The summary of the training data collected involves the mean and the standard deviation for each attribute, by class value.

- These are required when making predictions to calculate the probability of specific attribute values belonging to each class value.
- Summary data can be break down into the following sub-tasks:

- **Separate Data By Class**: The first task is to separate the training dataset instances by class value so that we can calculate statistics for each class. We can do that by creating a map of each class value to a list of instances that belong to that class and sort the entire dataset of instances into the appropriate lists.
- **Calculate Mean**:We need to calculate the mean of each attribute for a class value.

The mean is the central middle or central tendency of the data, and we will use it as the middle of our gaussian distribution when calculating probabilities.

- **Calculate Standard Deviation**: We also need to calculate the standard deviation of each attribute for a class value. The standard deviation describes the variation of spread of the data, and we will use it to characterize the expected spread of each attribute in our Gaussian distribution when calculating probabilities.
- **Summarize Dataset**: For a given list of instances (for a class value) we can calculate the mean and the standard deviation for each attribute.
- The zip function groups the values for each attribute across our data instances into their own lists so that we can compute the mean and standard deviation values for the attribute.
- **Summarize Attributes By Class**: We can pull it all together by first separating our training dataset into instances grouped by class. Then calculate the summaries for each attribute.

3) Make Predictions:

- ❖ Making predictions involves calculating the probability that a given data instance belongs to each class,
- ❖ then selecting the class with the largest probability as the prediction.
- ❖ Finally, estimation of the accuracy of the model by making predictions for each data instance in the test dataset.

4) Evaluate Accuracy: The predictions can be compared to the class values in the test dataset and a classification\ accuracy can be calculated as an accuracy ratio between 0& and 100%.

Naïve Bayes Program:

```
import csv
import random
import math
def safe_div(x,y):
   if y == 0:
      return 0
   return x / y
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()
```

Input:

Pima-indians-diabetes.csv

OUTPUT:

```
Split 768 rows into train=576 and test=192 rows
Accuracy: 77.604 %
```

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.

LEARN_NAIVE_BAYES_TEXT($Examples$, V)

Examples is a set of text documents along with their target values. V is the set of all possible target values. This function learns the probability terms $P(w_k|v_j)$, describing the probability that a randomly drawn word from a document in class v_j will be the English word w_k. It also learns the class prior probabilities $P(v_j)$.

1. *collect all words, punctuation, and other tokens that occur in Examples*
   - $Vocabulary \leftarrow$ the set of all distinct words and other tokens occurring in any text document from $Examples$
2. *calculate the required $P(v_j)$ and $P(w_k|v_j)$ probability terms*
   - For each target value v_j in V do
     - $docs_j \leftarrow$ the subset of documents from $Examples$ for which the target value is v_j
     - $P(v_j) \leftarrow \frac{|docs_j|}{|Examples|}$
     - $Text_j \leftarrow$ a single document created by concatenating all members of $docs_j$
     - $n \leftarrow$ total number of distinct word positions in $Text_j$
     - for each word w_k in $Vocabulary$
       - $n_k \leftarrow$ number of times word w_k occurs in $Text_j$
       - $P(w_k|v_j) \leftarrow \frac{n_k+1}{n+|Vocabulary|}$

CLASSIFY_NAIVE_BAYES_TEXT(Doc)

Return the estimated target value for the document Doc. a_i denotes the word found in the ith position within Doc.

- $positions \leftarrow$ all word positions in Doc that contain tokens found in $Vocabulary$
- Return v_{NB}, where

$$v_{NB} = \underset{v_j \in V}{\text{argmax}} P(v_j) \prod_{i \in positions} P(a_i|v_j)$$

Dataset:

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))
```

Output

2257
1502
['alt.atheism', 'comp.graphics', 'sci.med', 'soc.religion.christian']
From: sd345@city.ac.uk (Michael Collier)
Subject: Converting images to HP LaserJet III?
Nntp-Posting-Host: hampton
Organization: The City University
Lines: 14
Does anyone know of a good way (standard PC application/PD utility) to
convert tif/img/tga files into LaserJet III format. We would also like to
do the same, converting to HPGL (HP plotter) files.
Please email any response.
Is this the correct group?
Thanks in advance. Michael.
--
Michael Collier (Programmer) The Computer Unit,
Email: M.P.Collier@uk.ac.city The City University,
Tel: 071 477-8000 x3769 London,
Fax: 071 477-8565 EC1V 0HB.
1
Accuracy: 0.8348868175765646
precision recall f1-score support
alt.atheism 0.97 0.60 0.74 319
comp.graphics 0.96 0.89 0.92 389
sci.med 0.97 0.81 0.88 396
soc.religion.christian 0.65 0.99 0.78 398
avg / total 0.88 0.83 0.84 1502
confusion matrix is
[[192 2 6 119]
[2 347 4 36]
[2 11 322 61]
[2 2 1 393]]

Program 7

Write a program to construct a Bayesian network considering medical data. Use this model to demonstrate the diagnosis of heart patients using standard Heart Disease Data Set. You can use Java/Python ML library classes/API.

Following steps are used to build the Bayesian network

Step 1: Identify the variables which is set of attributes specified in the dataset(ex Medical Dataset)
Step2: Determine the domain of each variable that is set of values a variable may take
Step3: Create a directed graph network of nodes where each node represents the attribute and edges
represent parent child relationship. Edge represents that the child variable is conditionally dependent on the parent.
Step4 : determine the prior and conditional probability for each attribute
Step5 : perform the inference on the model and determine the marginal probabilities

Dataset: heart_disease_data.csv
Bayesian Network Program

```
import bayespy
as bp import
numpy as np
import csv
from colorama import init
from colorama import Fore, Back,
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]]
,lifeStyl eEnum[x[4]],cholesterolEnum[x[5]],heartDiseaseEnum[x[6]]])

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

# Input data column assignment
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 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()

# Sample Test with hardcoded values
#print("Sample Probability")
#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']])
# Interactive Test 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))
   #print(Style.RESET_ALL)
   m = int(input("Enter for Continue:0, Exit :1  "))
```

Input:

heart_disease_data.csv

Output:

```
Enter Age: {'SuperSeniorCitizen': 0, 'SeniorCitizen': 1, 'MiddleAged': 2, 'Youth': 3,
'Teen':
4}1
Enter Gender: {'Male': 0, 'Female': 1}1

Enter FamilyHistory: {'Yes': 0, 'No': 1}1
Enter dietEnum: {'High': 0, 'Medium': 1, 'Low': 2}2
Enter LifeStyle: {'Athlete': 0, 'Active': 1, 'Moderate': 2, 'Sedetary': 3}2
Enter Cholesterol: {'High': 0, 'BorderLine': 1, 'Normal': 2}1
Probability(HeartDisease) = 0.5
Enter for Continue:0, Exit :1  0

Enter Age: {'SuperSeniorCitizen': 0, 'SeniorCitizen': 1, 'MiddleAged': 2, 'Youth': 3, 'Teen':
4}0
Enter Gender: {'Male': 0, 'Female': 1}0
Enter FamilyHistory: {'Yes': 0, 'No': 1}0
Enter dietEnum: {'High': 0, 'Medium': 1, 'Low': 2}0
Enter LifeStyle: {'Athlete': 0, 'Active': 1, 'Moderate': 2, 'Sedetary': 3}3
Enter Cholesterol: {'High': 0, 'BorderLine': 1, 'Normal': 2}0
Probability(HeartDisease) = 0.5
Enter for Continue:0, Exit :1
```

Program 8

Apply EM algorithm to cluster a set of data stored in a .CSV file. Use the same data set for clustering using k-Means algorithm. Compare the results of these two algorithms and comment on the quality of clustering. You can add Java/Python ML library classes/API in the program.

Introduction to Expectation-Maximization (EM)

The EM algorithm tends to get stuck less than K-means algorithm. The idea is to assign data points partially to different clusters instead of assigning to only one cluster. To do this partial assignment, we model each cluster using a probabilistic distribution So a data point associates with a cluster with certain probability and it belongs to the cluster with the highest probability in the final assignment

Expectation-Maximization (EM) algorithm

Step 1: An initial guess is made for the model's parameters and a probability distribution is created. This is sometimes called the "E-Step" for the "Expected" distribution.

Step 2: Newly observed data is fed into the model.

Step 3: The probability distribution from the E-step is drawn to include the new data. This is sometimes called the "M-step."

Step 4: Steps 2 through 4 are repeated until stability.

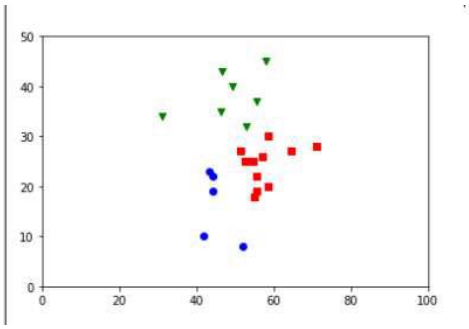
Dataset:

EM algorithm Programs:

```
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.')
```

Output



Program 9

Write a program to implement k-Nearest Neighbor algorithm to classify the iris data set. 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.

Dataset: iris.csv

ALGORITHM

Let m be the number of training data samples. Let p be an unknown point.

1. Store the training samples in an array of data points arr[]. This means each element of this array represents a tuple (x, y).
2. for i=0 to m:
   Calculate Euclidean distance d(arr[i], p).
3. Make set S of K smallest distances obtained. Each of these distances correspond to an already classified data point.
4. Return the majority label among S.

KNN Program

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
# ### Step 2 : Load the inbuilt data or the csv/excel file into pandas dataframe and clean the
data # In[66]:
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)))
```

Input:

Iris.csv

Output:

0.9666666666666667

Program 10

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

Locally Weighted Regression –

1. Nonparametric regression is a category of regression analysis in which the predictor does not take a predetermined form but is constructed according to information derived from the data(training examples).
2. Nonparametric regression requires larger sample sizes than regression based on parametric models. Because larger the data available ,accuracy will be high.

Locally Weighted Linear Regression –

- Locally weighted regression is called local because the function is approximated based a only on data near the query point, weighted because the contribution of each training example is weighted by its distance from the query point.
- Query point is nothing but the point nearer to the target function , which will help in finding the actual position of the target function.

Let us consider the case of locally weighted regression in which the target function f is approximated near x, using a linear function of the form

1. Minimize the squared error over just the k nearest neighbors:

$$E_1(x_q) \equiv \frac{1}{2} \sum_{x \in k \text{ nearest nbrs of } x_q} (f(x) - \hat{f}(x))^2$$

2. Minimize the squared error over the entire set D of training examples, while weighting the error of each training example by some decreasing function K of its distance from x_q:

$$E_2(x_q) \equiv \frac{1}{2} \sum_{x \in D} (f(x) - \hat{f}(x))^2 \; K(d(x_q, x))$$

3. Combine 1 and 2:

$$E_3(x_q) \equiv \frac{1}{2} \sum_{x \in k \text{ nearest nbrs of } x_q} (f(x) - \hat{f}(x))^2 \; K(d(x_q, x))$$

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