|  |
| --- |
| **Source Code of the Program 1:**  **import csv**  **#!usr/bin/python**  **#list creatin**  **hypo=['ɸ','ɸ','ɸ','ɸ','ɸ','ɸ'];**  **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);** |

|  |
| --- |
|  |
|  |
|  |
| **Program2:**Foragivensetoftrainingdataexamples storedina  .CSV file, implement and demonstrate the Candidate - elimination algorithm to output a description of the set ofall  hypotheses consistent with the training examples. |

|  |
| --- |
| Source Code :  import random  import csv  def g\_0(n):  return ("?",)\*n  def s\_0(n):  return ('ɸ',)\*n  def more\_general(h1, h2):  more\_general\_parts = []  for x, y in zip(h1, h2):  mg = x == "?" or (x != "ɸ" and (x == y or y == "ɸ"))  more\_general\_parts.append(mg)  return all(more\_general\_parts)  def fulfills(example, hypothesis):  ### the implementation is the same as for hypotheses:  return more\_general(hypothesis, example)  def min\_generalizations(h, x):  h\_new = list(h)  for i in range(len(h)):  if not fulfills(x[i:i+1], h[i:i+1]):  h\_new[i] = '?' if h[i] != 'ɸ' else x[i]  return [tuple(h\_new)]  def min\_specializations(h, domains, x):  results = []  for i in range(len(h)):  if h[i] == "?":  for val in domains[i]:  if x[i] != val:  h\_new = h[:i] + (val,) + h[i+1:]  results.append(h\_new)  elif h[i] != "ɸ":  h\_new = h[:i] + ('ɸ',) + h[i+1:]  results.append(h\_new)  return results  with open('trainingexamples.csv') as csvFile:  examples = [tuple(line) for line in csv.reader(csvFile)]  def get\_domains(examples):  d = [set() for i in examples[0]]  for x in examples:  for i, xi in enumerate(x):  d[i].add(xi)  return [list(sorted(x)) for x in d]  get\_domains(examples)  def candidate\_elimination(examples):  domains = get\_domains(examples)[:-1]  G = set([g\_0(len(domains))])  S = set([s\_0(len(domains))])  i = 0  print("\n G[{0}]:".format(i), G)  print("\n S[{0}]:".format(i), S)  for xcx in examples:  i = i + 1  x, cx = xcx[:-1], xcx[-1] # Splitting data into attributes and decisions  if cx == 'Y': # x is positive example  G = {g for g in G if fulfills(x, g)}  S = generalize\_S(x, G, S)  else: # x is negative example  S = {s for s in S if not fulfills(x, s)}  G = specialize\_G(x, domains, G, S)  print("\n G[{0}]:".format(i), G)  print("\n S[{0}]:".format(i), S)  return  def generalize\_S(x, G, S):  S\_prev = list(S)  for s in S\_prev:  if s not in S:  continue  if not fulfills(x, s):  S.remove(s)  Splus = min\_generalizations(s, x)  ## keep only generalizations that have a counterpart in G  S.update([h for h in Splus if any([more\_general(g,h)  for g in G])])  ## remove hypotheses less specific than any other in S  S.difference\_update([h for h in S if  any([more\_general(h, h1)  for h1 in S if h != h1])])  return S  def specialize\_G(x, domains, G, S):  G\_prev = list(G)  for g in G\_prev:  if g not in G:  continue  if fulfills(x, g):  G.remove(g)  Gminus = min\_specializations(g, domains, x)  ## keep only specializations that have a conuterpart in S  G.update([h for h in Gminus if any([more\_general(h, s)  for s in S])])  ## remove hypotheses less general than any other in G  G.difference\_update([h for h in G if  any([more\_general(g1, h)  for g1 in G if h != g1])])  return G  candidate\_elimination(examples) |
| **Program3:**Writeaprogramtodemonstratetheworkingofthe decisiontreebasedID3algorithm.Useanappropriatedataset forbuildingthedecisiontreeandapplythisknowledgetoclassify  a new sample. |

**Entropy of the Training Data Set**

|  |
| --- |
| **def**entropy(probs): *# Calulate the Entropy of givenprobability*  **import math**  **return** sum( [-prob\*math.log(prob, 2) **for** prob **in** probs] )  **def** entropy\_of\_list(a\_list): *# Entropy calculation of list of discrete val ues (YES/NO)*  **from collections import** Counter  cnt = Counter(x **for** x **in** a\_list) |

|  |
| --- |
| print("No and Yes Classes:",a\_list.name,cnt) num\_instances = len(a\_list)\*1.0  probs = [x / num\_instances **for** x **in** cnt.values()]  **return** entropy(probs) *# Call Entropy:*  *# The initial entropy of the YES/NO attribute for our dataset. #print(df\_tennis['PlayTennis'])*  total\_entropy = entropy\_of\_list(df\_tennis['PlayTennis'])  print("Entropy of given PlayTennis Data Set:",total\_entropy) |
|  |

**Output :**

No and Yes Classes: PlayTennis Counter({'Yes': 9, 'No': 5}) Entropy of given PlayTennis Data Set: 0.9402859586706309

# Information Gain ofAttributes

|  |
| --- |
| **def** information\_gain(df, split\_attribute\_name, target\_attribute\_name, trac e=0):  print("Information Gain Calculation of ",split\_attribute\_name)  *'''*  *Takes a DataFrame of attributes,and quantifies the entropy of a target attribute after performing a split along the values of another attribute.*  *'''*  *# Split Data by Possible Vals of Attribute:* df\_split = df.groupby(split\_attribute\_name) *#print(df\_split.groups)*  **for** name,group **in** df\_split: print(name) print(group)  *# Calculate Entropy for Target Attribute, as well as # Proportion of Obs in Each Data-Split*  nobs = len(df.index) \* 1.0  *#print("NOBS",nobs)*  df\_agg\_ent = df\_split.agg({target\_attribute\_name : [entropy\_of\_list, **l ambda** x: len(x)/nobs] })[target\_attribute\_name]  *#print("DFAGGENT",df\_agg\_ent)*  df\_agg\_ent.columns = ['Entropy', 'PropObservations']  *#if trace: # helps understand what fxn is doing: # print(df\_agg\_ent)*  *# Calculate Information Gain:*  new\_entropy = sum( df\_agg\_ent['Entropy'] \* df\_agg\_ent['PropObservation s'] )  old\_entropy = entropy\_of\_list(df[target\_attribute\_name]) |

|  |
| --- |
| **return** old\_entropy - new\_entropy  print('Info-gain for Outlook is :'+str( information\_gain(df\_tennis, 'Outlo ok', 'PlayTennis')),"**\n**")  print('**\n** Info-gain for Humidity is: ' + str( information\_gain(df\_tennis, 'Humidity', 'PlayTennis')),"**\n**")  print('**\n** Info-gain for Wind is:' + str( information\_gain(df\_tennis, 'Wind ', 'PlayTennis')),"**\n**")  print('**\n** Info-gain for Temperature is:' + str( information\_gain(df\_tennis  , 'Temperature','PlayTennis')),"**\n**") |

**ID3 Algorithm**

|  |
| --- |
| **def** id3(df, target\_attribute\_name, attribute\_names, default\_class=**None**):  *## Tally target attribute:*  **from collections import** Counter  cnt = Counter(x **for** x **in** df[target\_attribute\_name])*# class of YES /NO*  *## First check: Is this split of the dataset homogeneous?*  **if** len(cnt) == 1:  **return** next(iter(cnt))  *## Second check: Is this split of the dataset empty? # if yes, return a default value*  **elif** df.empty **or** (**not** attribute\_names): |

|  |
| --- |
| **return** default\_class  *## Otherwise: This dataset is ready to be divvied up!*  **else**:  *# Get Default Value for next recursive call of this function:*  default\_class = max(cnt.keys()) *#[index\_of\_max] # most common valu e of target attribute in dataset*  *# Choose Best Attribute to split on:*  gainz = [information\_gain(df, attr, target\_attribute\_name) **for** att r **in** attribute\_names]  index\_of\_max = gainz.index(max(gainz)) best\_attr = attribute\_names[index\_of\_max]  *# Create an empty tree, to be populated in a moment*  tree = {best\_attr:{}}  remaining\_attribute\_names = [i **for** i **in** attribute\_names **if** i != be  st\_attr]  *# Split dataset*  *# On each split, recursively call this algorithm. # populate the empty tree with subtrees, which*  *# are the result of the recursive call*  **for** attr\_val, data\_subset **in** df.groupby(best\_attr): subtree =id3(data\_subset,  target\_attribute\_name, remaining\_attribute\_names, default\_class)  tree[best\_attr][attr\_val] = subtree  **return** tree |

**Predicting Attributes**

|  |
| --- |
| *# Get Predictor Names (all but 'class')* attribute\_names = list(df\_tennis.columns) print("List of Attributes:", attribute\_names)  attribute\_names.remove('PlayTennis') *#Remove the class attribute*  print("Predicting Attributes:", attribute\_names) |

**Output :**

|  |
| --- |
| List of Attributes: ['PlayTennis', 'Outlook', 'Temperature', 'Humidity', ' Wind'] |
| Predicting Attributes: ['Outlook', 'Temperature', 'Humidity', 'Wind'] |

**Tree Construction**

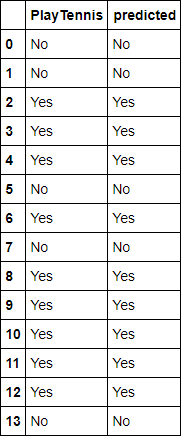
**Classification Accuracy**

|  |
| --- |
| **def** classify(instance, tree, default=**None**): attribute = next(iter(tree))*#tree.keys()[0]*  **if** instance[attribute] **in** tree[attribute].keys(): result = tree[attribute][instance[attribute]]  **if** isinstance(result, dict): *# this is a tree, delve deeper*  **return** classify(instance, result)  **else**:  **return** result *# this is a label*  **else**:  **return** default |

|  |
| --- |
| df\_tennis['predicted'] = df\_tennis.apply(classify, axis=1, args=(tree,'No'  ) )  *# classify func allows for a default arg: when tree doesn't have answe r for a particular*  *# combitation of attribute-values, we can use 'no' as the default gues*  *s*  print('Accuracy is:' + str( sum(df\_tennis['PlayTennis']==df\_tennis['predic ted'] ) / (1.0\*len(df\_tennis.index)) ))  df\_tennis[['PlayTennis', 'predicted']] |

Output :

Accuracy is:1.0



# Classification Accuracy: Training/Testing Set

|  |
| --- |
| training\_data = df\_tennis.iloc[1:-4] *# all but last thousand instances*  test\_data = df\_tennis.iloc[-4:] *# just the lastthousand*  train\_tree = id3(training\_data, 'PlayTennis', attribute\_names)  test\_data['predicted2'] = test\_data.apply(  *# <---- test\_data source*  classify, axis=1,  args=(train\_tree,'Yes') ) *# <---*  *- train\_data tree*  print ('**\n\n** Accuracy is : ' + str( sum(test\_data['PlayTennis']==test\_data ['predicted2'] ) / (1.0\*len(test\_data.index)) )) |

**Output :**

|  |
| --- |
| **Program4:** Build an Artificial Neural Network by implementing  the Backpropagation algorithm and test the same using appropriate datasets |

|  |
| --- |
| **import random**  **from math import** exp **from random import** seed *# Initialize a network*  **def** initialize\_network(n\_inputs, n\_hidden, n\_outputs): network = list()  hidden\_layer = [{'weights':[random.uniform(-0.5,0.5) **for** i **in** range(n\_ inputs + 1)]} **for** i **in** range(n\_hidden)]  network.append(hidden\_layer)  output\_layer = [{'weights':[random.uniform(-0.5,0.5) **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['outp  ut'])  *# 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(l en(expected))])  backward\_propagate\_error(network, expected) update\_weights(network, row, l\_rate)  print('>epoch=**%d**, lrate=**%.3f**, error=**%.3f**' % (epoch, l\_rate, sum\_er  ror))  *#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) train\_network(network, dataset, 0.5, 20, n\_outputs)  *#for layer in network: # print(layer)*  i= 1  **for** layer **in** network: j=1  **for** sub **in** layer:  print("**\n** Layer[**%d**] Node[**%d**]:**\n**" %(i,j),sub) j=j+1  i=i+1 |

|  |
| --- |
| **Output :** |

>epoch=0, lrate=0.500,error=4.763

>epoch=1, lrate=0.500,error=4.558

>epoch=2, lrate=0.500,error=4.316

>epoch=3, lrate=0.500,error=4.035

>epoch=4, lrate=0.500,error=3.733

>epoch=5, lrate=0.500,error=3.428

>epoch=6, lrate=0.500,error=3.132

>epoch=7, lrate=0.500,error=2.850

>epoch=8, lrate=0.500,error=2.588

>epoch=9, lrate=0.500,error=2.348

>epoch=10, lrate=0.500,error=2.128

>epoch=11, lrate=0.500,error=1.931

>epoch=12, lrate=0.500,error=1.753

>epoch=13, lrate=0.500,error=1.595

>epoch=14, lrate=0.500,error=1.454

>epoch=15, lrate=0.500,error=1.329

>epoch=16, lrate=0.500,error=1.218

>epoch=17, lrate=0.500,error=1.120

>epoch=18, lrate=0.500,error=1.033

>epoch=19, lrate=0.500,error=0.956

Layer[1] Node[1]:

{'weights': [-1.435239043819221, 1.8587338175173547, 0.7917644224148094],

'output': 0.029795197360175857, 'delta': -0.006018730117768358}

Layer[1] Node[2]:

{'weights': [-0.7704959899742789, 0.8257894037467045, 0.21154103288579731

], 'output': 0.06771641538441577, 'delta': -0.005025585510232048}

Layer[2] Node[1]:

{'weights': [2.223584933362892, 1.2428928053374768, -1.3519548925527454],

'output': 0.23499833662766154, 'delta': -0.042246618795029306}

Layer[2] Node[2]:

{'weights': [-2.509732251870173, -0.5925943219491905, 1.259965727484093],

'output': 0.7543931062537561, 'delta': 0.04550706392557862}

|  |
| --- |
| **from math import** exp  *# 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  *# Make a prediction with a network*  **def** predict(network, row):  outputs = forward\_propagate(network, row)  **return** outputs.index(max(outputs))  *# Test making predictions with the network*  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]]  network = [[{'weights': [-1.482313569067226, 1.8308790073202204,1.0783819  22048799]}, {'weights': [0.23244990332399884, 0.3621998343835864,0.402898  21191094327]}],  [{'weights': [2.5001872433501404, 0.7887233511355132,-1.1026649757805  829]}, {'weights': [-2.429350576245497, 0.8357651039198697,1.069921718128  0656]}]]  **for** row **in** dataset:  prediction = predict(network, row)  print('Expected=**%d**, Got=**%d**' % (row[-1], prediction)) |

Expected=0, Got=0 Expected=0, Got=0 Expected=0, Got=0 Expected=0, Got=0 Expected=0, Got=0 Expected=1, Got=1 Expected=1, Got=1 Expected=1, Got=1 Expected=1, Got=1 Expected=1, Got=1

|  |
| --- |
| **Program5:**Write a program to implement the **naïve Bayesian classifier**  forasampletrainingdatasetstoredasa.CSVfile.Computetheaccuracy of the classifier, considering few test datasets. |

### Bayesian Theorem:

**SourceCode:**

|  |
| --- |
| *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*  *# 1.Data Handling*  *# 1.1 Loading the Data from csv file of ConceptLearning dataset.*  *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]]*  *return dataset*  *#1.2 Splitting the Data set into Training Set*  *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))*  *return [trainSet, copy]*  *#2.Summarize Data*  *#The naive bayes model is comprised of a*  *#summary of the data in the training dataset.*  *#This summary is then used when making predictions.*  *#involves the mean and the standard deviation for each attribute, by class value*  *#2.1: Separate Data By Class*  *#Function to categorize the dataset in terms of classes*  *#The function assumes that the last attribute (-1) is the class value.*  *#The function returns a map of class values to lists of data instances.*  *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*  *#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*  *#2.2 : Calculate Mean*  *def mean(numbers):*  *return safe\_div(sum(numbers),float(len(numbers)))*  *#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.*  *#2.3 : Calculate Standard Deviation*  *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)*  *#2.4 : Summarize Dataset*  *#Summarize Data Set for a list of instances (for a class value)*  *#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.*  *def summarize(dataset):*  *summaries = [(mean(attribute), stdev(attribute)) for attribute in zip(\*dataset)]*  *del summaries[-1]*  *return summaries*  *#2.5 : 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.*  *def summarizeByClass(dataset):*  *separated = separateByClass(dataset)*  *summaries = {}*  *for classValue, instances in separated.items():*  *summaries[classValue] = summarize(instances)*  *return summaries*  *#3.Make Prediction*  *#3.1 Calculate Probaility Density Function*  *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*  *#3.2 Calculate Class Probabilities*  *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*  *#3.3 Prediction : look for the largest probability and return the associated class*  *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*  *#4.Make Predictions*  *# Function which return predictions for list of predictions*  *# For each instance*  *def getPredictions(summaries, testSet):*  *predictions = []*  *for i in range(len(testSet)):*  *result = predict(summaries, testSet[i])*  *predictions.append(result)*  *return predictions*  *#5. Computing Accuracy*  *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 = 0.90*  *dataset = loadCsv(filename)*  *trainingSet, testSet = splitDataset(dataset, splitRatio)*  *print('Split {0} rows into'.format(len(dataset)))*  *print('Number of Training data: ' + (repr(len(trainingSet))))*  *print('Number of Test Data: ' + (repr(len(testSet))))*  *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("\nThe Training set are:")*  *for x in trainingSet:*  *print(x)*  *print("\nThe Test data set are:")*  *for x in testSet:*  *print(x)*  *print("\n")*  *# prepare model*  *summaries = summarizeByClass(trainingSet)*  *# test model*  *predictions = getPredictions(summaries, testSet)*  *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()* |

Output:

Naive Bayes Classifier for concept learning problem

Split 16 rows into

Number of Training data: 14

Number of Test Data: 2

The values assumed for the concept learning attributes are

OUTLOOK=> Sunny=1 Overcast=2 Rain=3

TEMPERATURE=> Hot=1 Mild=2 Cool=3

HUMIDITY=> High=1 Normal=2

WIND=> Weak=1 Strong=2

TARGET CONCEPT:PLAY TENNIS=> Yes=10 No=5

The Training set are:

[1.0, 1.0, 1.0, 1.0, 5.0]

[1.0, 1.0, 1.0, 2.0, 5.0]

[2.0, 1.0, 1.0, 2.0, 10.0]

[3.0, 2.0, 1.0, 1.0, 10.0]

[3.0, 3.0, 2.0, 1.0, 10.0]

[3.0, 3.0, 2.0, 2.0, 5.0]

[2.0, 3.0, 2.0, 2.0, 10.0]

[1.0, 2.0, 1.0, 1.0, 5.0]

[1.0, 3.0, 2.0, 1.0, 10.0]

[3.0, 2.0, 2.0, 2.0, 10.0]

[1.0, 2.0, 2.0, 2.0, 10.0]

[2.0, 2.0, 1.0, 2.0, 10.0]

[2.0, 1.0, 2.0, 1.0, 10.0]

[3.0, 2.0, 1.0, 2.0, 5.0]

The Test data set are:

[1.0, 2.0, 1.0, 2.0, 10.0]

[1.0, 2.0, 1.0, 2.0, 5.0]

Actual values: [10.0, 5.0]%

Predictions: [5.0, 5.0]%

Accuracy: 50.0%

|  |
| --- |
| Program6: 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. |

**Source Code:**

Loading the 20 newsgroups dataset : The dataset is called “Twenty Newsgroups”. Here is the official

description, quoted from the website[:http://qwone.com/~jason/20Newsgroups/](http://qwone.com/%7Ejason/20Newsgroups/)

The20Newsgroupsdatasetisacollectionofapproximately20,000newsgroupdocuments,partitioned (nearly)evenlyacross20differentnewsgroups.Tothebestofourknowledge,itwasoriginallycollected by Ken Lang, probably for his paper “Newsweeder: Learning to filter netnews,” though he does not explicitly mention this collection. The 20 newsgroups collection has become a popular data set for experimentsintextapplicationsofmachinelearningtechniques,suchastextclassificationandtext clustering.

|  |
| --- |
| Source Code:  from sklearn.datasets import fetch\_20newsgroups  from sklearn.metrics import confusion\_matrix  from sklearn.metrics import classification\_report  import numpy as np  twenty\_train = fetch\_20newsgroups(subset='train', shuffle=True)  x = len(twenty\_train.target\_names)  print("\n The number of categories:",x)  print("\n The %d Different Categories of 20Newsgroups\n" %x)  i=1  for cat in twenty\_train.target\_names:  print("Category[%d]:" %i,cat)  i=i+1  print("\n Length of train data is",len(twenty\_train.data))  print("\n Length of file names is ",len(twenty\_train.filenames))  #Considering only four Categories  categories = ['alt.atheism', 'soc.religion.christian','comp.graphics', 'sci.med']  twenty\_train = fetch\_20newsgroups(subset='train',categories=categories,shuffle=True)  twenty\_test = fetch\_20newsgroups(subset='test',categories=categories,shuffle=True)  print("Reduced length of train data",len(twenty\_train.data))  print("length of test data",len(twenty\_test.data))  print("Target Names",twenty\_train.target\_names)  #print("\n".join(twenty\_train.data[0].split("\n")))  #print(twenty\_train.target[0])  #Extracting features from text files  from sklearn.feature\_extraction.text import CountVectorizer  count\_vect = CountVectorizer()  #Term Frequencies(tf): Divide the number of occurrences of each word in a document by the total number of words in the document  X\_train\_tf = count\_vect.fit\_transform(twenty\_train.data)  X\_train\_tf.shape  print("tf train count",X\_train\_tf.shape)  #another refinement for tf is called tf–idf for “Term Frequency times Inverse Document Frequency”.  from sklearn.feature\_extraction.text import TfidfTransformer  tfidf\_transformer = TfidfTransformer()  X\_train\_tfidf = tfidf\_transformer.fit\_transform(X\_train\_tf)  X\_train\_tfidf.shape  print("tfidf train count",X\_train\_tfidf.shape)  from sklearn.naive\_bayes import MultinomialNB  from sklearn.metrics import accuracy\_score  from sklearn import metrics  mod = MultinomialNB()  mod.fit(X\_train\_tfidf, twenty\_train.target)  X\_test\_tf = count\_vect.transform(twenty\_test.data)  print("tf test count",X\_test\_tf.shape)  X\_test\_tfidf = tfidf\_transformer.transform(X\_test\_tf)  print("tfidf test count",X\_test\_tfidf.shape)  predicted = mod.predict(X\_test\_tfidf)  print("Accuracy:", accuracy\_score(twenty\_test.target, predicted))  print(classification\_report(twenty\_test.target,predicted,target\_names=twenty\_test.target\_names))  print("confusion matrix is \n",metrics.confusion\_matrix(twenty\_test.target, predicted)) |

Output:

|  |
| --- |
| The number of categories: 20  The 20 Different Categories of 20Newsgroups  Category[1]: alt.atheism  Category[2]: comp.graphics  Category[3]: comp.os.ms-windows.misc  Category[4]: comp.sys.ibm.pc.hardware  Category[5]: comp.sys.mac.hardware  Category[6]: comp.windows.x  Category[7]: misc.forsale  Category[8]: rec.autos  Category[9]: rec.motorcycles  Category[10]: rec.sport.baseball  Category[11]: rec.sport.hockey  Category[12]: sci.crypt  Category[13]: sci.electronics  Category[14]: sci.med  Category[15]: sci.space  Category[16]: soc.religion.christian  Category[17]: talk.politics.guns  Category[18]: talk.politics.mideast  Category[19]: talk.politics.misc  Category[20]: talk.religion.misc  Length of train data is 11314  Length of file names is 11314  Reduced length of train data 2257  length of test data 1502  Target Names ['alt.atheism', 'comp.graphics', 'sci.med', 'soc.religion.christian']  tf train count (2257, 35788)  tfidf train count (2257, 35788)  tf test count (1502, 35788)  tfidf test count (1502, 35788)  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]]  >>> |

|  |
| --- |
| **Program7:**Writeaprogramtoconstructa**Bayesiannetwork**considering medicaldata.Usethismodeltodemonstratethediagnosisofheartpatients usingstandardHeartDiseaseDataSet.YoucanuseJava/PythonML  library classes/API. |

**Source Code:**

* 1. **ConstructingaBayesianNetworkconsideringMedicalData**
     1. **DefiningaStructurewithnodesandedges**

|  |
| --- |
| *# Starting with defining the network structure*  **from pgmpy.models import** BayesianModel  cancer\_model = BayesianModel([('Pollution', 'Cancer'),  ('Smoker', 'Cancer'),  ('Cancer', 'Xray'),  ('Cancer', 'Dyspnoea')])  cancer\_model.nodes() |

cancer\_model.edges()

cancer\_model.get\_cpds()

### CreationofConditionalProbabilityTable

|  |
| --- |
| *# Now defining the parameters.*  **from pgmpy.factors.discrete import** TabularCPD  cpd\_poll = TabularCPD(variable='Pollution', variable\_card=2,  values=[[0.9], [0.1]])  cpd\_smoke = TabularCPD(variable='Smoker', variable\_card=2,  values=[[0.3], [0.7]])  cpd\_cancer = TabularCPD(variable='Cancer', variable\_card=2,  values=[[0.03, 0.05, 0.001, 0.02],  [0.97, 0.95, 0.999, 0.98]],  evidence=['Smoker', 'Pollution'], evidence\_card=[2, 2])  cpd\_xray = TabularCPD(variable='Xray', variable\_card=2,  values=[[0.9, 0.2], [0.1, 0.8]],  evidence=['Cancer'], evidence\_card=[2]) cpd\_dysp = TabularCPD(variable='Dyspnoea', variable\_card=2,  values=[[0.65, 0.3], [0.35, 0.7]],  evidence=['Cancer'], evidence\_card=[2]) |

* + 1. **AssociatingConditionalprobabilitieswiththeBayesianStructure**

|  |
| --- |
| *# Associating the parameters with the model structure.*  cancer\_model.add\_cpds(cpd\_poll, cpd\_smoke, cpd\_cancer, cpd\_xray, cpd\_dysp)  *# Checking if the cpds are valid for the model.*  cancer\_model.check\_model() |

|  |
| --- |
| *# Doing some simple queries on the network* cancer\_model.is\_active\_trail('Pollution', 'Smoker') cancer\_model.is\_active\_trail('Pollution', 'Smoker', observed=['Cancer']) cancer\_model.get\_cpds()  print(cancer\_model.get\_cpds('Pollution')) |

print(cancer\_model.get\_cpds('Smoker'))

|  |
| --- |
| print(cancer\_model.get\_cpds('Xray')) print(cancer\_model.get\_cpds('Dyspnoea'))  print(cancer\_model.get\_cpds('Cancer')) |

### Determining the Localindependencies

|  |
| --- |
| cancer\_model.local\_independencies('Xray') cancer\_model.local\_independencies('Pollution') cancer\_model.local\_independencies('Smoker') cancer\_model.local\_independencies('Dyspnoea') cancer\_model.local\_independencies('Cancer')  cancer\_model.get\_independencies() |

**7.1.5.Inferencing with BayesianNetwork**

|  |
| --- |
| *# Doing exact inference using Variable Elimination* **from pgmpy.inference import** VariableElimination cancer\_infer = VariableElimination(cancer\_model)  *# Computing the probability of bronc given smoke.*  q = cancer\_infer.query(variables=['Cancer'], evidence={'Smoker': 1}) print(q['Cancer'])  *# Computing the probability of bronc given smoke.*  q = cancer\_infer.query(variables=['Cancer'], evidence={'Smoker': 1}) print(q['Cancer'])  *# Computing the probability of bronc given smoke.*  q = cancer\_infer.query(variables=['Cancer'], evidence={'Smoker': 1,'Pollut ion': 1})  print(q['Cancer']) |

* 1. **DiagnosisofheartpatientsusingstandardHeartDiseaseDataSet**

|  |
| --- |
| **import numpy as np**  **from urllib.request import** urlopen  **import urllib**  **import matplotlib.pyplot as plt** *# Visuals*  **import seaborn as sns import sklearn as skl**  **import pandas as pd** |

* + 1. **ImportingHeartDiseaseDataSetandCustomizing**

|  |
| --- |
| Cleveland\_data\_URL = '<http://archive.ics.uci.edu/ml/machine-learning-datab>  ases/heart-disease/processed.hungarian.data' |

|  |
| --- |
| *#Hungarian\_data\_URL =* [*'http://archive.ic*](http://archive.ics.uci.edu/ml/machine-learning-data)*s*[*.uci.edu/ml/machine-learning-data*](http://archive.ics.uci.edu/ml/machine-learning-data) *bases/heart-disease/processed.hungarian.data'*  *#Switzerland\_data\_URL =* [*'http://archive.*](http://archive.ics.uci.edu/ml/machine-learning-da)*i*[*cs.uci.edu/ml/machine-learning-da*](http://archive.ics.uci.edu/ml/machine-learning-da) *tabases/heart-disease/processed.switzerland.data'* np.set\_printoptions(threshold=np.nan) *#see a whole array when we outputit*  names = ['age', 'sex', 'cp', 'trestbps', 'chol', 'fbs', 'restecg','thalac  h', 'exang', 'oldpeak', 'slope', 'ca', 'thal', 'heartdisease'] heartDisease = pd.read\_csv(urlopen(Cleveland\_data\_URL), names = names) *#ge ts Clevelanddata*  *#HungarianHeartDisease = pd.read\_csv(urlopen(Hungarian\_data\_URL), names = names) #gets Hungary data*  *#SwitzerlandHeartDisease = pd.read\_csv(urlopen(Switzerland\_data\_URL), name s = names) #gets Switzerland data*  *#datatemp = [ClevelandHeartDisease, HungarianHeartDisease, SwitzerlandHear tDisease] #combines all arrays into a list*  *#heartDisease = pd.concat(datatemp)#combines list into one array*  heartDisease.head()  **del** heartDisease['ca'] **del** heartDisease['slope'] **del**heartDisease['thal']  **del** heartDisease['oldpeak']  heartDisease = heartDisease.replace('?', np.nan) heartDisease.dtypes  heartDisease.columns |

* + 1. **Modeling Heart DiseaseData**

|  |
| --- |
| **from pgmpy.models import** BayesianModel  **from pgmpy.estimators import** MaximumLikelihoodEstimator, BayesianEstimator  model = BayesianModel([('age', 'trestbps'), ('age', 'fbs'), ('sex', 'trest bps'), ('sex', 'trestbps'),  ('exang', 'trestbps'),('trestbps','heartdisease'),(  'fbs','heartdisease'),  ('heartdisease','restecg'),('heartdisease','thalach'  ),('heartdisease','chol')])  *# Learing CPDs using Maximum Likelihood Estimators* model.fit(heartDisease, estimator=MaximumLikelihoodEstimator) *#for cpd in model.get\_cpds():*  *# print("CPD of {variable}:".format(variable=cpd.variable)) # print(cpd)*  print(model.get\_cpds('age')) |

print(model.get\_cpds('chol'))

|  |
| --- |
| print(model.get\_cpds('sex'))  model.get\_independencies() |

### Inferencing with BayesianNetwork

*# Doing exact inference using Variable Elimination* **from pgmpy.inference import** VariableElimination HeartDisease\_infer = VariableElimination(model)

*# Computing the probability of bronc given smoke.*

q = HeartDisease\_infer.query(variables=['heartdisease'], evidence={'age': 28}

)

print(q['heartdisease'])

╒════════════════╤═════════════════════╕

│heartdisease │ phi(heartdisease)│

╞════════════════╪═════════════════════╡

│heartdisease\_0│ 0.6333│

├────────────────┼─────────────────────┤

│heartdisease\_1│ 0.3667│

╘════════════════╧═════════════════════╛

In [35]:

q = HeartDisease\_infer.query(variables=['heartdisease'], evidence={'chol': 10 0})

print(q['heartdisease'])

╒════════════════╤═════════════════════╕

│heartdisease │ phi(heartdisease)│

╞════════════════╪═════════════════════╡

│heartdisease\_0│ 1.0000│

├────────────────┼─────────────────────┤

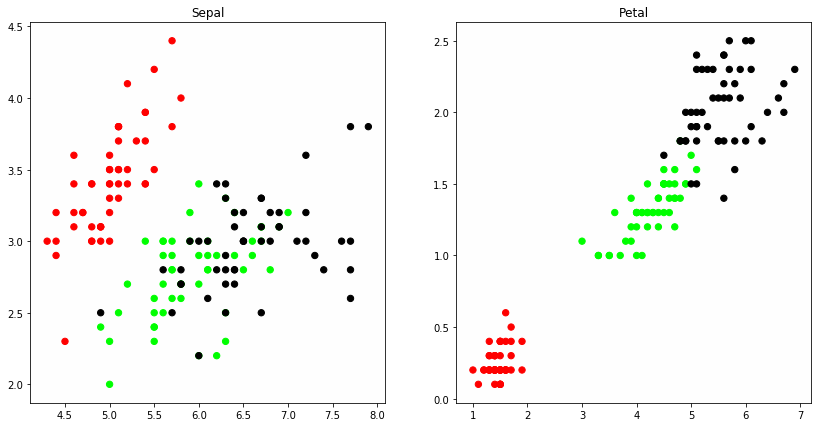
│heartdisease\_1│ 0.0000│

╘════════════════╧═════════════════════╛

|  |
| --- |
| **Program8:**Apply**EMalgorithm**toclusterasetofdatastoredina.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 ofclustering.YoucanaddJava/PythonMLlibraryclasses/APIinthe  program. |
| **import matplotlib.pyplot as plt from sklearn import** datasets  **from sklearn.cluster import** KMeans **import sklearn.metrics as sm import pandas aspd**  **import numpy asnp**  %**matplotlib**inline  *# 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 TARGETNAMES:\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 theplot*  plt.figure(figsize=(14,7))  *# Create a colormap*  colormap = np.array(['red', 'lime', 'black'])  *# Plot Sepal*  plt.subplot(1, 2, 1)  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') |

Text(0.5,1,'Petal')



### Build the KMeans Model

|  |
| --- |
| *# K Means Cluster*  model = KMeans(n\_clusters=3) model.fit(X)  *# This is what KMeans thought*  model.labels\_ |

array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

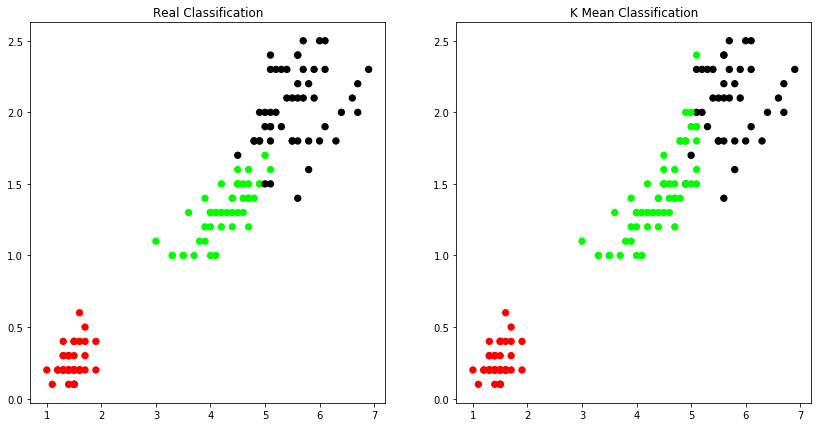
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0

,

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 0, | 0, | 0, | 0, | 1, | 1, | 2, | 1, | 1, | 1, | 1, | 1, | 1, | 1, | 1, | 1, | 1, | 1, | 1, | 1, | 1, | 1, | 1 |
| , |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | 1, | 1, | 1, | 1, | 1, | 1, | 1, | 1, | 2, | 1, | 1, | 1, | 1, | 1, | 1, | 1, | 1, | 1, | 1, | 1, | 1, | 1, | 1 |
| , |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | 1, | 1, | 1, | 1, | 1, | 1, | 1, | 1, | 2, | 1, | 2, | 2, | 2, | 2, | 1, | 2, | 2, | 2, | 2, | 2, | 2, | 1, | 1 |
| , |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | 2, | 2, | 2, | 2, | 1, | 2, | 1, | 2, | 1, | 2, | 2, | 1, | 1, | 2, | 2, | 2, | 2, | 2, | 1, | 2, | 2, | 2, | 2 |
| , |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | 1, | 2, | 2, | 2, | 1, | 2, | 2, | 2, | 1, | 2, | 2, | 1]) |  |  |  |  |  |  |  |  |  |  |  |

### Visualise the classifierresults

|  |
| --- |
| *# View the results*  *# Set the size of theplot*  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') |

Text(0.5,1,'K Mean Classification')

### The Fix

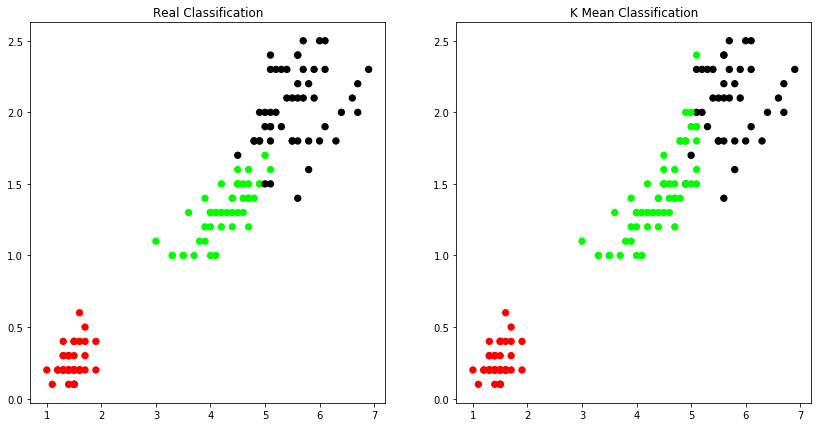
|  |
| --- |
| *# The fix, we convert all the 1s to 0s and 0s to 1s.*  predY = np.choose(model.labels\_, [0, 1, 2]).astype(np.int64)  print (predY) |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| [0 0 0 0 0 | 0 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 0 0 0 0 | 0 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 2 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 1 1 1 2 1 | 1 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 2 | 1 | 2 | 2 | 2 | 2 | 1 | 2 | 2 | 2 | 2 |
| 2 2 1 1 2 | 2 2 | 2 | 1 | 2 | 1 | 2 | 1 | 2 | 2 | 1 | 1 | 2 | 2 | 2 | 2 | 2 | 1 | 2 | 2 | 2 | 2 | 1 | 2 | 2 | 2 | 1 | 2 | 2 | 2 | 1 | 2 |
| 2 1] |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

**Re-plot**

|  |
| --- |
| *# View the results*  *# Set the size of theplot*  plt.figure(figsize=(14,7))  *# Create a colormap*  colormap = np.array(['red', 'lime', 'black'])  *# Plot Orginal*  plt.subplot(1, 2, 1)  plt.scatter(X.Petal\_Length, X.Petal\_Width, c=colormap[y.Targets], s=40) plt.title('Real Classification')  *# Plot Predicted with corrected values*  plt.subplot(1, 2, 2)  plt.scatter(X.Petal\_Length,X.Petal\_Width, c=colormap[predY], s=40)  plt.title('K Mean Classification') |

Text(0.5,1,'K Mean Classification')



### Performance Measures

**Accuracy**

sm.accuracy\_score(y,model.labels\_) 0.89333333333333331

# Confusion Matrix

|  |
| --- |
| *# Confusion Matrix*  sm.confusion\_matrix(y, model.labels\_) |

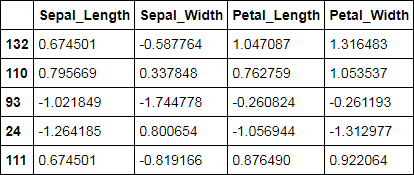
array([[50, 0, 0],

[0,48, 2],

[ 0, 14, 36]], dtype=int64)

# GMM

|  |
| --- |
| **from sklearn import** preprocessing scaler =preprocessing.StandardScaler()  scaler.fit(X)  xsa = scaler.transform(X)  xs = pd.DataFrame(xsa, columns = X.columns)  xs.sample(5) |



|  |
| --- |
| **from sklearn.mixture import** GaussianMixture gmm = GaussianMixture(n\_components=3)  gmm.fit(xs) |

GaussianMixture(covariance\_type='full', init\_params='kmeans', max\_iter=100

,

means\_init=None, n\_components=3, n\_init=1, precisions\_init=None, random\_state=None, reg\_covar=1e-06, tol=0.001, verbose=0, verbose\_interval=10, warm\_start=False, weights\_init=None)

|  |
| --- |
| y\_cluster\_gmm = gmm.predict(xs)  y\_cluster\_gmm |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| array([0, 0, | 0, | 0, | 0, | 0, | 0, | 0, | 0, | 0, | 0, | 0, | 0, | 0, | 0, | 0, | 0, | 0, | 0, | 0, | 0, | 0, | 0 |
| , |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 0, 0, | 0, | 0, | 0, | 0, | 0, | 0, | 0, | 0, | 0, | 0, | 0, | 0, | 0, | 0, | 0, | 0, | 0, | 0, | 0, | 0, | 0 |
| , |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 0, 0, | 0, | 0, | 1, | 1, | 1, | 1, | 1, | 1, | 1, | 1, | 1, | 1, | 1, | 1, | 1, | 1, | 1, | 1, | 1, | 1, | 2 |
| , |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 1, 2, | 1, | 2, | 1, | 1, | 1, | 1, | 2, | 1, | 1, | 1, | 1, | 1, | 2, | 1, | 1, | 1, | 1, | 1, | 1, | 1, | 1 |
| , |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 1, 1, | 1, | 1, | 1, | 1, | 1, | 1, | 2, | 2, | 2, | 2, | 2, | 2, | 2, | 2, | 2, | 2, | 2, | 2, | 2, | 2, | 2 |

,

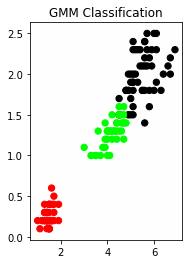
2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2

,

2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2], dtype=int32)

|  |
| --- |
| plt.subplot(1, 2, 1)  plt.scatter(X.Petal\_Length, X.Petal\_Width, c=colormap[y\_cluster\_gmm], s=40  )  plt.title('GMM Classification') |

ext(0.5,1,'GMM Classification')



sm.accuracy\_score(y,y\_cluster\_gmm) 0.96666666666666667

|  |
| --- |
| *# Confusion Matrix*  sm.confusion\_matrix(y, y\_cluster\_gmm) |

array([[50, 0, 0],

[0,45, 5],

[0, 0, 50]],dtype=int64)

|  |
| --- |
| *# so the GMM clustering matched the true labels more closely than the Kmea ns,*  *# as expected from the plots.* |

|  |
| --- |
| **Program9 :** Write a program to implement k-Nearest Neighbour algorithm to classify the iris data set. Print both correct andwrong  predictions.Java/PythonMLlibraryclassescanbeusedforthisproblem. |

Algorithm :

|  |
| --- |
| *# 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\_d ataset["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))  ) |

**Output :**

Actual : 2 virginica, Predicted :[2]['virginica'] Actual : 1 versicolor, Predicted :[1]['versicolor'] Actual : 0 setosa, Predicted:[0]['setosa']

Actual : 2 virginica, Predicted :[2]['virginica'] Actual : 0 setosa, Predicted:[0]['setosa']

--------

Actual : 1 versicolor, Predicted :[2]['virginica'] TEST SCORE[ACCURACY]: 0.97

|  |
| --- |
| **Program10 :** Implement the non-parametric Locally Weighted Regressionalgorithminordertofitdatapoints.Selectappropriatedata  set for your experiment and draw graphs. |

### Algorithm : Regression:

|  |
| --- |
| **import numpy as np**  **from bokeh.plotting import** figure, show, output\_notebook  **from bokeh.layouts import**gridplot  **from bokeh.io import**push\_notebook  output\_notebook() |

BokehJS 0.12.10 successfully loaded.

|  |
| --- |
| **import numpy as np** |

|  |
| --- |
| **def** local\_regression(x0, X, Y, tau):  *# add bias term*  x0 = np.r\_[1, x0] *# Add one to avoid the loss in information*  X = np.c\_[np.ones(len(X)), X]  *# fit model: normal equations with kernel*  xw = X.T \* radial\_kernel(x0,X,tau) *# XTranspose \*W*  beta = np.linalg.pinv(xw @ X) @ xw@Y *# @ Matrix Multiplication or DotProduct*  *# predict value*  **return** x0@beta *# @ Matrix Multiplication or Dot Product for predi ction*  **def** radial\_kernel(x0, X, tau):  **return** np.exp(np.sum((X - x0) \*\* 2, axis=1) / (-2 \* tau \* tau))  *# Weight or Radial Kernal Bias Function* |

|  |
| --- |
| n = 1000  *# generate dataset*  X = np.linspace(-3, 3, num=n)  print("The Data Set ( 10 Samples) X :**\n**",X[1:10]) Y = np.log(np.abs(X \*\* 2 - 1) + .5)  print("The Fitting Curve Data Set (10Samples)Y :**\n**",Y[1:10])  *# jitter X*  X += np.random.normal(scale=.1, size=n)  print("Normalised (10 Samples) X :**\n**",X[1:10]) |

The Data Set ( 10 Samples) X :

|  |  |  |
| --- | --- | --- |
| [-2.99399399 -2.98798799 -2.98198198 -2.97597598 | -2.96996997 | -2.96396396 |
| -2.95795796 -2.95195195 -2.94594595]  The Fitting Curve Data Set (10Samples)Y :  [2.13582188 2.13156806 2.12730467 2.12303166 | 2.11874898 | 2.11445659 |

2.11015444 2.10584249 2.10152068]

Normalised (10 Samples) X :

[-3.17013248 -2.87908581 -3.37488159 -2.90743352 -2.93640374 -2.97978828

-3.0549104 -3.0735006 -2.88552749]

|  |
| --- |
| domain = np.linspace(-3, 3, num=300)  print(" Xo DomainSpace(10Samples) :**\n**",domain[1:10])  **def** plot\_lwr(tau):  *# prediction through regression*  prediction = [local\_regression(x0, X, Y, tau) **for** x0 **in** domain] plot = figure(plot\_width=400, plot\_height=400) plot.title.text='tau=**%g**' % tau  plot.scatter(X, Y, alpha=.3)  plot.line(domain, prediction, line\_width=2, color='red')  **return** plot |

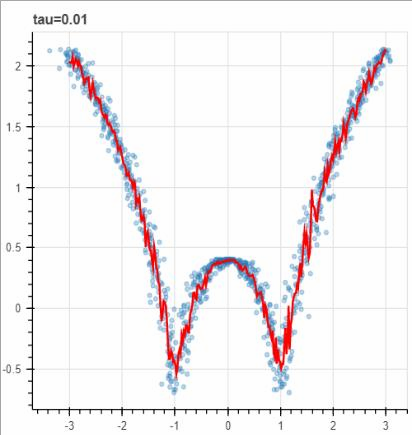
Xo DomainSpace(10Samples) :

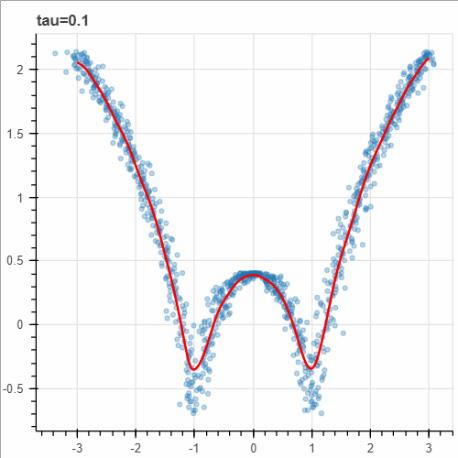
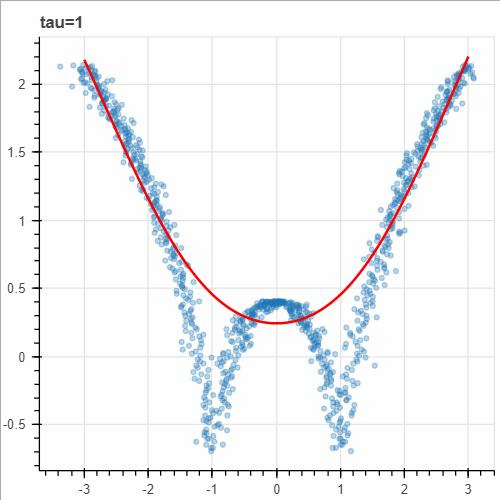
[-2.97993311 -2.95986622 -2.93979933 -2.91973244 -2.89966555 -2.87959866

-2.85953177 -2.83946488 -2.81939799]

|  |
| --- |
| *# Plotting the curves with different tau*  show(gridplot([  [plot\_lwr(10.), plot\_lwr(1.)], [plot\_lwr(0.1), plot\_lwr(0.01)]  ])) |

**Output :**





|  |
| --- |
|  |