**1.FindS**

import csv

def loadCsv(filename):

lines=csv.reader(open(filename,"r"))

dataset=list(lines)

headers=dataset.pop(0)

return dataset,headers

def print\_hypothesis(h):

print('<',end=' ')

for i in range(0,len(h)-1):

print(h[i],end=' ')

print('>')

def findS():

dataset,features=loadCsv('finds.csv')

rows=len(dataset)

col=len(dataset[0])

flag=0

for x in range(0,rows):

t=dataset[x]

if t[-1]=='1' and flag==0:

flag=1

h=dataset[x]

elif t[-1]=='1':

for y in range(col):

if h[y]!=t[y]:

h[y]='?'

print("The maximally specific hypothesis for a given training examples")

print\_hypothesis(h)

findS()

**2. Candidate elimination**

import random

import csv

def g\_o(n):

return ("?",)\*n

def s\_o(n):

return ("0",)\*n

def more\_general(h1,h2):

more\_general\_parts=[]

for x,y in zip(h1,h2):

mg=x=="?" or (x!="0" and (x==y or y=="0"))

more\_general\_parts.append(mg)

return all(more\_general\_parts)

def fulfills(example,hypothesis):

return more\_general(hypothesis,example)

def min\_generalization(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]!="0" else x[i]

return [tuple(h\_new)]

min\_generalization(h=('0','0','sunny'),x=('rainy','windy','cloudy'))

def min\_specialization(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]!="0":

h\_new=h[:i]+('0',)+h[i+1:]

results.append(h\_new)

return results

min\_specialization(h=('?','x'),domains=[['a','b','c'],['x','y']],x=('b','x'))

with open('candidate.csv') as csvFile:

examples=[tuple(line) for line in csv.reader(csvFile)]

def get\_domain(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\_domain(examples)

def candidate\_elimination(examples):

domains=get\_domain(examples)[:-1]

G=set([g\_o(len(domains))])

S=set([s\_o(len(domains))])

i=0

print("\nG[{0}]".format(i),G)

print("\nS[{0}]".format(i),S)

for xcx in examples:

i=i+1

x,cx=xcx[:-1],xcx[-1]

if cx=='Y':

G={g for g in G if fulfills(x,g) }

S=generalize\_S(x,G,S)

else:

S={s for s in S if not fulfills(x,s)}

G=specialize\_G(x,domains,G,S)

print("\nG[{0}]:".format(i),G)

print("\nS[{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\_generalization(s,x)

S.update([h for h in splus if any([more\_general(g,h) for g in G])])

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\_specialization(g,domains,x)

G.update([h for h in gminus if any([more\_general(h,s) for s in S])])

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)

**3. Decision Tree**

import math

import csv

def load\_csv(filename):

lines = csv.reader(open(filename, "r"));

dataset = list(lines)

headers = dataset.pop(0)

return dataset, headers

class Node:

def \_\_init\_\_(self, attribute):

self.attribute = attribute

self.children = []

self.answer = ""

def subtables(data, col, delete):

dic = {}

coldata = [ row[col] for row in data]

attr = list(set(coldata))

for k in attr:

dic[k] = []

for y in range(len(data)):

key = data[y][col]

if delete:

del data[y][col]

dic[key].append(data[y])

return attr, dic

def entropy(S):

attr = list(set(S))

if len(attr) == 1:

return 0

counts = [0,0]

for i in range(2):

counts[i] = sum( [1 for x in S if attr[i] == x] ) / (len(S) \* 1.0)

sums = 0

for cnt in counts:

sums += -1 \* cnt \* math.log(cnt,2)

return sums

def compute\_gain(data, col):

attValues, dic = subtables(data, col, delete=False)

total\_entropy = entropy([row[-1] for row in data])

for x in range(len(attValues)):

ratio = len(dic[attValues[x]]) / ( len(data) \* 1.0)

entro = entropy([row[-1] for row in dic[attValues[x]]])

total\_entropy -= ratio\*entro

return total\_entropy

def build\_tree(data, features):

lastcol = [row[-1] for row in data]

if (len(set(lastcol))) == 1:

node=Node("")

node.answer = lastcol[0]

return node

n = len(data[0])-1

gains = [compute\_gain(data, col) for col in range(n) ]

split = gains.index(max(gains))

node = Node(features[split])

fea = features[:split]+features[split+1:]

attr, dic = subtables(data, split, delete=True)

for x in range(len(attr)):

child = build\_tree(dic[attr[x]], fea)

node.children.append((attr[x], child))

return node

def print\_tree(node, level):

if node.answer != "":

print(" "\*level, node.answer)

return

print(" "\*level, node.attribute)

for value, n in node.children:

print(" "\*(level+1), value)

print\_tree(n, level + 2)

def classify(node,x\_test,features):

if node.answer != "":

print(node.answer)

return

pos = features.index(node.attribute)

for value, n in node.children:

if x\_test[pos]==value:

classify(n,x\_test,features)

dataset, features = load\_csv("decision.csv")

node = build\_tree(dataset, features)

print("The decision tree for the dataset using ID3 algorithm is ")

print\_tree(node, 0)

testdata, features = load\_csv("decisiontest.csv")

for xtest in testdata:

print("The test instance : ",xtest)

print("The predicted label : ", end="")

classify(node,xtest,features)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_­­­­­­­­­\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**4. ANN**

import numpy as np

x=np.array(([2,9],[1,5],[3,6]),dtype=float)

y=np.array(([92],[86],[89]),dtype=float)

x=x/np.amax(x,axis=0)

y=y/100

class Neural\_Network(object):

def \_\_init\_\_(self):

self.inputsize=2

self.outputsize=1

self.hiddensize=3

self.w1=np.random.randn(self.inputsize,self.hiddensize)

self.w2=np.random.randn(self.hiddensize,self.outputsize)

def forward(self,x):

self.z=np.dot(x,self.w1)

self.z2=self.sigmoid(self.z)

self.z3=np.dot(self.z2,self.w2)

o=self.sigmoid(self.z3)

return o

def sigmoid(self,s):

return 1/(1+np.exp(-s))

def sigmoidPrime(self,S):

return S\*(1-S)

def backward(self,x,y,o):

self.o\_error=y-0

self.o\_delta=self.o\_error\*self.sigmoidPrime(o)

self.z2\_error=self.o\_delta.dot(self.w2.T)

self.z2\_delta=self.z2\_error\*self.sigmoidPrime(self.z2)

self.w1+=x.T.dot(self.z2\_delta)

self.w2+=self.z2.T.dot(self.o\_delta)

def train(self,x,y):

o=self.forward(x)

self.backward(x,y,o)

NN=Neural\_Network()

for i in range(10):

print(i)

print("Input :\n"+str(x))

print("Actual output:\n" +str(y))

print("Predicted output:\n"+str(NN.forward(x)))

print("Loss:\n"+str(np.mean(np.square(y-NN.forward(x)))))

print("\n")

NN.train(x,y)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**5. NAÏVE BAYESIAN**

import csv

import random

import math

def loadCsv(filename):

lines = csv.reader(open(filename, "r"))

dataset = list(lines)

for i in range(len(dataset)):

dataset[i] = [float(x) for x in dataset[i]]

return dataset

def splitDataset(dataset, splitRatio):

trainSize = int(len(dataset) \* splitRatio)

trainSet = []

copy = list(dataset)

while len(trainSet) < trainSize:

index = random.randrange(len(copy))

trainSet.append(copy.pop(index))

return [trainSet, copy]

def separateByClass(dataset):

separated = {}

for i in range(len(dataset)):

vector = dataset[i]

if (vector[-1] not in separated):

separated[vector[-1]] = []

separated[vector[-1]].append(vector)

return separated

def mean(numbers):

return sum(numbers)/float(len(numbers))

def stdev(numbers):

avg = mean(numbers)

variance = sum([pow(x-avg,2) for x in numbers])/float(len(numbers)-1)

return math.sqrt(variance)

def summarize(dataset):

summaries = [(mean(attribute), stdev(attribute)) for attribute in zip(\*dataset)]

del summaries[-1]

return summaries

def summarizeByClass(dataset):

separated = separateByClass(dataset)

summaries = {}

for classValue, instances in separated.items():

summaries[classValue] = summarize(instances)

return summaries

def calculateProbability(x, mean, stdev):

exponent = math.exp(-(math.pow(x-mean,2)/(2\*math.pow(stdev,2))))

return (1 / (math.sqrt(2\*math.pi) \* stdev)) \* exponent

def calculateClassProbabilities(summaries, inputVector):

probabilities = {}

for classValue, classSummaries in summaries.items():

probabilities[classValue] = 1

for i in range(len(classSummaries)):

mean, stdev = classSummaries[i]

x = inputVector[i]

probabilities[classValue] \*= calculateProbability(x, mean, stdev)

return probabilities

def predict(summaries, inputVector):

probabilities = calculateClassProbabilities(summaries, inputVector)

bestLabel, bestProb = None, -1

for classValue, probability in probabilities.items():

if bestLabel is None or probability > bestProb:

bestProb = probability

bestLabel = classValue

return bestLabel

def getPredictions(summaries, testSet):

predictions = []

for i in range(len(testSet)):

result = predict(summaries, testSet[i])

predictions.append(result)

return predictions

def getAccuracy(testSet, predictions):

correct = 0

for i in range(len(testSet)):

if testSet[i][-1] == predictions[i]:

correct += 1

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

def main():

filename = 'bayesian.csv'

splitRatio = 0.67

dataset = loadCsv(filename)

trainingSet, testSet = splitDataset(dataset, splitRatio)

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 of the classifier is : {0}%'.format(accuracy))

main()

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**6. NAÏVE BAYESIAN BUILT IN**

import pandas as pd

msg=pd.read\_csv('naivebuiltin.csv',header=None,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

from sklearn.model\_selection import train\_test\_split

xtrain,xtest,ytrain,ytest=train\_test\_split(X,y,random\_state=1)

print('dimensions of train and test sets')

print(xtrain.shape)

print(xtest.shape)

print(ytrain.shape)

print(ytest.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)

# 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)

print('\nclassification results of testing samples are given below')

for doc,p in zip(xtest,predicted):

pred='pos' if p==1 else 'neg'

print('%s->%s' %(doc,pred))

#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))

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**7.HEART DISEASE**

import numpy as np

import pandas as pd

import csv

from pgmpy.estimators import MaximumLikelihoodEstimator

from pgmpy.models import BayesianModel

from pgmpy.inference import VariableElimination

lines = list(csv.reader(open('data7\_names.csv', 'r')));

attributes = lines[0]

heartDisease = pd.read\_csv('data7\_heart.csv', names = attributes)

heartDisease = heartDisease.replace('?', np.nan)

print('Few examples from the dataset are given below')

print(heartDisease.head())

print('\nAttributes and datatypes')

print(heartDisease.dtypes)

# Model Baysian Network

model = BayesianModel([('age', 'trestbps'), ('age', 'fbs'), ('sex', 'trestbps'), ('sex', 'trestbps'),

('exang', 'trestbps'),('trestbps','heartdisease'),('fbs','heartdisease'),

('heartdisease','restecg'),('heartdisease','thalach'),('heartdisease','chol')])

# Learning CPDs using Maximum Likelihood Estimators

print('\nLearning CPDs using Maximum Likelihood Estimators...');

model.fit(heartDisease, estimator=MaximumLikelihoodEstimator)

# Inferencing with Bayesian Network

print('\nInferencing with Bayesian Network:')

HeartDisease\_infer = VariableElimination(model)

# Computing the probability of bronc given smoke.

print('\n1.Probability of HeartDisease given Age=20')

q = HeartDisease\_infer.query(variables=['heartdisease'], evidence={'age': 57})

print(q)

print('\n2. Probability of HeartDisease given chol (Cholestoral) =100')

q = HeartDisease\_infer.query(variables=['heartdisease'], evidence={'chol': 203})

print(q)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**8. EM,K-means**

from sklearn.cluster import KMeans

#from sklearn import metrics

import matplotlib.pyplot as plt

from sklearn import datasets

from sklearn.cluster import KMeans

import pandas as pd

import numpy as np

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']

model=KMeans(n\_clusters=3)

model.fit(X)

plt.figure(figsize=(14,14))

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

plt.subplot(2,2,1)

plt.scatter(X.Petal\_Length,X.Petal\_Width,c=colormap[model.labels\_],s=40)

plt.title('K-Means Clustering')

plt.xlabel('Petal Length\n')

plt.ylabel('Petal Width\n')

from sklearn import preprocessing

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('\nGMM Clustering')

plt.xlabel('\nPetal Length')

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**9. KNN**

from sklearn.model\_selection import train\_test\_split

from sklearn.neighbors import KNeighborsClassifier

from sklearn import datasets

iris=datasets.load\_iris()

print("iris data set is loaded-----------------------")

x\_train,x\_test,y\_train,y\_test=train\_test\_split(iris.data,iris.target,test\_size=0.1)

print("dataset is spilt into training and testing:")

print("size of trsining data and its label:",x\_train.shape,y\_train.shape)

print("size of testing data and its label:",x\_test.shape,y\_test.shape)

for i in range(len(iris.target\_names)):

print("lable",i,"\_",str(iris.target\_names[i]))

classifier=KNeighborsClassifier(n\_neighbors=1)

classifier.fit(x\_train,y\_train)

y\_pred=classifier.predict(x\_test)

print("results of classfication using knnwith k=1")

for r in range(0,len(x\_test)):

print("sample:",str(x\_test[r]),"actual-label:",str(y\_test[r]),"predicted\_label:",str(y\_pred[r]))

print("classification accuracy:",classifier.score(x\_test,y\_test))

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**10.Locally Weighted Regression**

from numpy import \*

import operator

from os import listdir

import matplotlib

import matplotlib.pyplot as plt

import pandas as pd

import numpy.linalg

from scipy.stats.stats import pearsonr

def kernel(point,xmat, k):

m,n = shape(xmat)

weights = mat(eye((m)))

for j in range(m):

diff = point - X[j]

weights[j,j] = 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 = shape(xmat)

ypred = zeros(m)

for i in range(m):

ypred[i] = xmat[i]\*localWeight(xmat[i],xmat,ymat,k)

return ypred

# load data points

data = pd.read\_csv('tips.csv')

bill = array(data.total\_bill)

tip = array(data.tip)

#preparing and add 1 in bill

mbill = mat(bill)

mtip = mat(tip)

m= shape(mbill)[1]

one = mat(ones(m))

X= hstack((one.T,mbill.T))

#set k here

ypred = localWeightRegression(X,mtip,3)

SortIndex = X[:,1].argsort(0)

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();

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_