**Appendix**

**#The data is Having course Review and Labels . Label 1 is for a bad course and 5 is a good course**

import numpy as np

import pandas as pd

data=pd.read\_csv("reviews\_by\_course.csv") # getting the data into python program (Coursera courses with labels)

courses=[]

courses=data["CourseId"].unique() # saving the course names

courses[1:100]

len(courses) # getting the number of courses

data.describe()

from sklearn.utils import shuffle

df=pd.DataFrame(data)

df=shuffle(df) # to randomize the data and select just 10000 rows of 1 lakh rows

data.head()

df.head()

# import random

# random.shuffle(data)

# get only few rows for analysis

shortDf=df[:10000]

shortDf.isnull().sum() # check for null values

# shortDf.dropna() # to remove null rows

shortDf.insert(0, 'SNo', range(0,len(shortDf))) # adding a serial no column

courses=shortDf['CourseId'].unique() # getting name of all courses

print(len(courses))

courses

**#Removing Non English Reviews**

# from enchant.checker import SpellChecker # for removing non english text reviews

# def is\_in\_english(quote):

# d = SpellChecker("en\_US")

# d.set\_text(quote)

# errors = [err.word for err in d]

# return False if ((len(errors) > 4) or len(quote.split()) < 3) else True

import enchant

# dictionary =enchant.Dict("en\_US")

def is\_in\_english\_second\_approach(quote): # faster removal of non english reviews

dictionary =enchant.Dict("en\_US")

return dictionary.check(quote)

from nltk.tokenize import sent\_tokenize, word\_tokenize

# egwords=word\_tokenize(shortDf.iloc[0][2])[0]

# egwords

# word=word\_tokenize(shortDf.iloc[0][2])[0]

# word

print(shortDf.iloc[0][2])

# def remove\_rows(shortDf): # to remove review not in english

# row=shortDf.size

# for i in range(10000) :

# word=word\_tokenize(shortDf.iloc[i][2])[0] # checking first word of each review to see if its in english

# if word=="":

# shortDf.drop(shortDf.index[i])

# elif is\_in\_english\_second\_approach(word)==False:

# shortDf.drop(shortDf.index[i])

wordsforReference=[]

for i in range(10000):

wordsforReference.append(word\_tokenize(shortDf.iloc[i][2])[0])

wordsforReference # getting all first words of review to see if in english

indexes\_to\_drop=[] # getting the indexes where review are not in english

for i in range(10000):

if is\_in\_english\_second\_approach(wordsforReference[i])==False:

indexes\_to\_drop.append(i)

indexes\_to\_drop

indexes\_to\_keep = set(range(shortDf.shape[0])) - set(indexes\_to\_drop)

df\_sliced = shortDf.take(list(indexes\_to\_keep)) # it is much faster to create a new Dataframe than to drop

len(df\_sliced)

df\_sliced=df\_sliced.drop(columns=["SNo"]) # reindexing

df\_sliced.shape

df\_sliced.insert(0, 'SNo', range(0,df\_sliced.shape[0]))

df\_sliced.iloc[0]

courses[1:100]

**Making the feature sets**

from nltk.tokenize import sent\_tokenize, word\_tokenize

from nltk.corpus import wordnet

from nltk.corpus import stopwords

stops=set(stopwords.words(‘english'))

allCourseReview=[] # getting all the course reviews in a single array

def get\_all\_course\_review(df\_sliced):

for review in df\_sliced["Review"]:

allCourseReview.append(review)

get\_all\_course\_review(df\_sliced)

all\_words=[] # tokenizing each review

def get\_words\_from\_sentence(all\_words):

word=[]

for sentence in allCourseReview:

word=word\_tokenize(sentence)

all\_words.append(word)

get\_words\_from\_sentence(all\_words)

# documents = []

# for index in range(df\_sliced.shape[0]):

# label=df\_sliced.iloc[index][3]

# sentence=all\_words[index]

# for words in sentence:

# documents.append((words,label))

doc=[] # relating each review to its label

for index in range(df\_sliced.shape[0]):

label=df\_sliced.iloc[index][3]

currentReview=all\_words[index]

doc.append((currentReview,label))

wordArray=[] # getting individual words from reviews

for sentence in all\_words:

for word in sentence:

wordArray.append(word.lower())

wordArray

clean\_words=[] # removing unnecesary words

stops.update(".", "?", "(", ")", ",", "-", "'", '"',"!","...",";","course","'s")

for w in wordArray:

if w.lower() not in stops:

clean\_words.append(w.lower())

import nltk

clean\_words = nltk.FreqDist(clean\_words) # getting the most common words

clean\_words.most\_common(15)

word\_features = []

most\_common = clean\_words.most\_common()

for i in range(3000): # selecting 3000 most common words as features

word\_features.append(most\_common[i][0])

len(word\_features)

word\_features[1:10]

def find\_features(review): # it is a kind of heatmap to tell if a review contains any of the most common words

words = set(review)

features = {} # the dictionary is true for words which are present in the review

for w in word\_features:

features[w] = (w in words)

return features

features = find\_features(all\_words[11])# for a single document

featuresets = [(find\_features(review), label) for (review,label) in doc]# for multiple documents

training\_set = featuresets[:1500]

testing\_set = featuresets[1500:]

classifier = nltk.NaiveBayesClassifier.train(training\_set)

nltk.classify.accuracy(classifier, testing\_set)

classifier.show\_most\_informative\_features(15) # this shows if a word is 5 it means pretty good course and if a word is 1 it is very bad course

features=find\_features(all\_words[27])

**Saving The Classifier**

import pickle

save\_classifier = open("naivebayes.pickle", "wb") # Saving the current classifier

pickle.dump(classifier, save\_classifier)

save\_classifier.close()

classifier\_new = open("naivebayes.pickle", "rb")

classifier = pickle.load(classifier\_new)

classifier\_new.close()

**The classifier gives 1 to reviews with label 1 but it gives 5 to all other reviews as it is a binary class classifier .**

# #Dictionary of People rating for fruits

# choices={'John': {'Mango':4.5, 'Banana':3.5, 'Strawberry':4.0, 'Pineapple':4.0},

# 'Nick': {'Mango':4.0, 'Orange':4.5, 'Banana':3.0, 'Pineapple':4.5},

# 'Martha': {'Orange':5.0, 'Banana':2.5, 'Strawberry':4.5, 'Apple':3.5},

# 'Mathew': {'Mango':3.75, 'Strawberry':4.25, 'Apple':3.5, 'Pineapple':3.0}}

**pearson correlation**

from math import sqrt

#Finding Similarity among people using Eucledian Distance Formula

class testClass():

def pearson(self, cho, per1, per2):

#Will set the following dictionary if data is common for two persons

sample\_data={}

#Above mentioned varibale is an empty dictionary, that is length =0

for items in cho[per1]:

if items in cho[per2]:

sample\_data[items]=1

#Value is being set 1 for those items which are same for both persons

#Calculating length of sample\_data dictionary

length = len(sample\_data)

#If both person does not have any similarity or similar items return 0

if length==0: return 0

#Remember one thing we will calculate all the below values only for common items

# or the items which are being shared by both person1 and person2, that's why

# we will be using sample\_data dictionary in below loops

#Calculating Sum of all common elements for Person1 and Person2

sum1=sum([cho[per1][val] for val in sample\_data])

sum2=sum([cho[per2][val] for val in sample\_data])

#Calculating Sum of squares of all common elements for both

sumSq1=sum([pow(cho[per1][val],2) for val in sample\_data])

sumSq2=sum([pow(cho[per2][val],2) for val in sample\_data])

#Calculating Sum of Products of all common elements for both

sumPr=sum([cho[per1][val]\*cho[per2][val] for val in sample\_data])

#Calculating Person Correlation Score

x = sumPr-(sum1\*sum2/length)

y = sqrt((sumSq1-pow(sum1,2)/length)\*(sumSq2-pow(sum2,2)/length))

if y==0 : return 0

return(x/y)

#Value being returned above always lies between -1 and 1

#Value of 1 means maximum similarity

def main():

ob = testClass()

print(ob.pearson(choices, 'John', 'Nick'))

print(ob.pearson(choices, 'John', 'Martha'))

print(ob.pearson(choices, 'John', 'John'))

if \_\_name\_\_ == "\_\_main\_\_":

main()

datatoRead=pd.read\_excel("Untitled spreadsheet (1).xlsx")

**Trying out the clustering algorithm**

# import statements

from sklearn.datasets import make\_blobs

import numpy as np

import matplotlib.pyplot as plt

# create blobs

data = make\_blobs(n\_samples=200, n\_features=2, centers=4, cluster\_std=1.6, random\_state=50)

# create np array for data points

points = data[0]

# create scatter plot

plt.scatter(data[0][:,0], data[0][:,1], c=data[1], cmap='viridis')

plt.xlim(-15,15)

plt.ylim(-15,15)

# import KMeans

from sklearn.cluster import KMeans

# create kmeans object

kmeans = KMeans(n\_clusters=4)

# fit kmeans object to data

kmeans.fit(points)

# print location of clusters learned by kmeans object

print(kmeans.cluster\_centers\_)

# save new clusters for chart

y\_km = kmeans.fit\_predict(points)

plt.scatter(points[y\_km ==0,0], points[y\_km == 0,1], s=100, c='red')

plt.scatter(points[y\_km ==1,0], points[y\_km == 1,1], s=100, c='black')

plt.scatter(points[y\_km ==2,0], points[y\_km == 2,1], s=100, c='blue')

plt.scatter(points[y\_km ==3,0], points[y\_km == 3,1], s=100, c=‘cyan')

# import hierarchical clustering libraries

import scipy.cluster.hierarchy as sch

from sklearn.cluster import AgglomerativeClustering

# create dendrogram

dendrogram = sch.dendrogram(sch.linkage(points, method='ward'))

# create clusters

hc = AgglomerativeClustering(n\_clusters=4, affinity = 'euclidean', linkage = 'ward')

# save clusters for chart

y\_hc = hc.fit\_predict(points)

plt.scatter(points[y\_hc ==0,0], points[y\_hc == 0,1], s=100, c='red')

plt.scatter(points[y\_hc==1,0], points[y\_hc == 1,1], s=100, c='black')

plt.scatter(points[y\_hc ==2,0], points[y\_hc == 2,1], s=100, c='blue')

plt.scatter(points[y\_hc ==3,0], points[y\_hc == 3,1], s=100, c='cyan')

def getLevelColumn(str):

if(str=="beginner"):

return 1

elif(str=="intermediate"):

return 2

else :

return 3

numericalValueOfDifficultyLevel=datatoRead["Difficulty level of Course”].apply(getLevelColumn)

datatoRead[‘NumericalDifficultyLevelOfCourse']=numericalValueOfDifficultyLevel

datatoRead["NumericalValueOfCertificateNeeded"]=datatoRead['Certificate Needed’].apply(getNumericalValueofCertificate)

from sklearn import linear\_model

def getScore(year,DifficultyLevel,LengthOfCOurse,CertificateNeeded):

sumOfValues=(year/4)+(DifficultyLevel/3)+(LengthOfCOurse/5)+(CertificateNeeded)

return sumOfValues

scores=[]

for data in datatoRead:

scores.append(getScore(data[1],data[16],data[3],data[17]))

for data in datatoRead:

print(data)

dimension=datatoRead.shape

scores=[]

for i in range (dimension[0]):

sumOfVal=getScore(rows.iloc[i:i+1,1],rows.iloc[i:i+1,16],rows.iloc[i:i+1,3],rows.iloc[i:i+1,17])

scores.append(sumOfVal)

scoretoread=[]

for i in range(dimension[0]):

scoretoread.append(scores[i][i])

datatoRead[‘NewScores']=scoretoread

from sklearn.linear\_model import LinearRegression

from sklearn import cross\_validation

clf=LinearRegression()

x\_train, x\_test, y\_train, y\_test=cross\_validation.train\_test\_split(datatoRead, datatoRead['NewScores'], test\_size=0.2)

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

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

y\_pred

slicedData=datatoRead['Year Currently Studying in’]

slicedData['LengthOfCourse']=datatoRead['Length of Course(in months)’]

slicedData[‘NumericalDifficulty']=datatoRead['NumericalDifficultyLevelOfCourse']

sliceOfData=datatoRead.iloc[:,2]

newDataFiled=pd.DataFrame()

newDataFiled['Year']=datatoRead['Year Currently Studying in’]

newDataFiled['Length']=datatoRead["Length of Course(in months)”]

newDataFiled[‘Difficulty']=datatoRead['NumericalDifficultyLevelOfCourse']

newDataFiled[‘Certificate']=datatoRead['NumericalValueOfCertificateNeeded']

newDataFiled[‘score']=datatoRead['NewScores']

x\_train, x\_test, y\_train, y\_test=cross\_validation.train\_test\_split(newDataFiled.iloc[0:,0:4], newDataFiled['score'], test\_size=0.2)

def getDetails():

input("Enter name ")

year=input("enter year of study")

year=int(year)

length=input("Enter length of course")

length=int(length)

difficulty=input("enter level of difficulty")

difficulty=getLevelColumn(difficulty)

certificate=input("enter yes if certificate needed")

certificate=getNumericalValueofCertificate(certificate)

return (year,length,difficulty,certificate)

testcase=getDetails()

showval=clf.predict(testcase)

score=clf.predict(testcase)

def mainProgram():

test=getDetails()

score=clf.predict(test)

recommendation=getRecommendation(score[0])

return recommendation[1:6]