#1.1 Implementation of Standard scaler (normalize data with standard scaler)

import numpy as np

import pandas as pd

class StandardNorm:

  def scale(self,d):

    for i in d.columns:

      mean=d[i].mean()

      sd=d[i].std()

      d[i]=(d[i]-mean)/sd

    return d

data=pd.DataFrame([[45000,42],[32000,26],[58000,48],[37000,32]],columns=['Salary','Age'])

print("Original Data")

print(data)

s=StandardNorm()

df=s.scale(data)

print("\nScaled Data")

print(df)

#1.2 Implementation of min-max scaler

import pandas as pd

class MinMaxNorm:

  def scale(self,d):

    for i in d.columns:

      min=d[i].min()

      max=d[i].max()

      d[i]=(d[i]-min)/(max-min)

    return d

data=pd.DataFrame([[45000,42],[32000,26],[58000,48],[37000,32]],columns=['Salary','Age'])

print("Original Data")

print(data)

s=MinMaxNorm()

print("\nScaled Data")

df=s.scale(data)

print(df)

#Lab 2 : Data Cleaning

import pandas as pd

import numpy as np

data = pd.read\_csv('/content/drive/My Drive/Data/employees.csv')

print("Original Data")

print(data[0:25])

# Removing missing values

data=data.dropna(axis=0)

# Removing duplicate rows

data.drop\_duplicates(keep='first',inplace=True)

# Removing column Boonus %

del data['Bonus %']

# Correcting Inconsitencies among values

data['Team']=data['Team'].str.replace('Fin','Finance')

data['Team']=data['Team'].str.replace('Mkt','Marketing')

data['Team']=data['Team'].str.replace('Financeance','Finance')

print("Cleaned Data")

print(data[0:25])

#Lab 2 : Data Cleaning

import pandas as pd

import numpy as np

data = pd.read\_csv('/content/drive/My Drive/Data/employees.csv')

print("Original Data")

print(data[0:25])

# Filling missing values with mean

data['Salary']=data['Salary'].fillna(data['Salary'].mean())

print("Cleaned Data")

print(data[0:25])

data = pd.read\_csv('/content/drive/My Drive/Data/employees.csv')

print("Original Data")

print(data[0:25])

# Filling missing values withi

data['Salary']=data['Salary'].interpolate(method="linear")

print("Cleaned Data")

print(data[0:25])

#K-means Clustering

import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

from sklearn.cluster import KMeans

data=100\*np.random.rand(1000,2)

print(\*data)

km=KMeans(n\_clusters=3,init='random')

km.fit(data)

centers = km.cluster\_centers\_

labels = km.labels\_

print("Cluster Centers:",\*centers)

print("Cluster Labels:",\*labels)

colors = ["r","g","b"]

markers=["+","x","\*"]

for i in range(len(data)):

    plt.plot(data[i][0], data[i][1], color=colors[labels[i]], marker=markers[labels[i]])

plt.scatter(centers[:, 0],centers[:, 1], marker = "o", s=50, linewidths = 5)

plt.show()

#Mini-Batch K-means Clustering

import time

import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

from sklearn.cluster import MiniBatchKMeans

data=100\*np.random.rand(10000,2)

print(\*data)

mbk=MiniBatchKMeans(n\_clusters=5,init='random', batch\_size=3000)

t0= time.time()

mbk.fit(data)

t1= time.time()

tt=t1-t0

print("Total Time:",tt)

cents = mbk.cluster\_centers\_

labels = mbk.labels\_

print("Cluster Centers:",\*cents)

print("Labels:",\*labels)

colors = ["g","r","b",'y','m']

markers=["+","x","\*",'.','d']

for i in range(len(data)):

    plt.plot(data[i][0], data[i][1], color=colors[labels[i]], marker=markers[labels[i]])

plt.scatter(cents[:, 0],cents[:, 1], marker = "o", s=50, linewidths = 5)

plt.show()

#5.1 Clustering Iris data using KMedoids

from sklearn.datasets import load\_iris

import numpy as np

import matplotlib.pyplot as plt

from sklearn\_extra.cluster import KMedoids

from sklearn.preprocessing import StandardScaler

from sklearn import metrics

iris\_data=load\_iris()

x=iris\_data.data

y=iris\_data.target

print(\*x)

print("Actual Group:",\*y)

sc=StandardScaler()

sc.fit(x)

sx=sc.transform(x)

km=KMedoids(n\_clusters=3)

km.fit(sx)

py=km.fit\_predict(sx)

print("Predicted Group",\*py)

fig = plt.figure(figsize = (12, 8))

ax = fig.add\_subplot(111, projection='3d')

colors = ["g","r","b"]

markers=["+","x","\*"]

for i in range(len(sx)):

  ax.scatter(sx[i][0], sx[i][1], sx[i][2],color=colors[py[i]], marker=markers[py[i]])

plt.show()

hs=metrics.homogeneity\_score(y, py)  #Homoginity Score

print("Homogeniety Score:",hs)

sc=metrics.silhouette\_score(sx, py, metric='euclidean') #Silhouette Coefficient

print("Silhouette Coefficient:",sc)

#5.2 Clustering Iris data using Agglomerative

from sklearn.datasets import load\_iris

import numpy as np

import matplotlib.pyplot as plt

from sklearn.cluster import AgglomerativeClustering

from sklearn.preprocessing import StandardScaler

from sklearn import metrics

iris\_data=load\_iris()

x=iris\_data.data

y=iris\_data.target

sc=StandardScaler()

sc.fit(x)

sx=sc.transform(x)

ac=AgglomerativeClustering(n\_clusters=3)

ac.fit(sx)

py=ac.fit\_predict(sx)

fig = plt.figure(figsize = (12, 8))

ax = fig.add\_subplot(111, projection='3d')

colors = ["g","r","b"]

markers=["+","x","\*"]

for i in range(len(x)):

  ax.scatter(sx[i][0], sx[i][1], sx[i][2],color=colors[py[i]], marker=markers[py[i]])

plt.show()

hs=metrics.homogeneity\_score(y, py)  #Homoginity Score

print("Homogeniety Score:",hs)

sc=metrics.silhouette\_score(sx, py, metric='euclidean') #Silhouette Coefficient

print("Silhouette Coefficient:",sc)

#Diabetes Prediction Using Naive Bayes Classifier

import pandas as pd

from sklearn import metrics

from sklearn.naive\_bayes import GaussianNB

dataset = pd.read\_csv('/content/drive/My Drive/Data/Diabetes.csv')

split = int(len(dataset)\*0.7)

train, test = dataset[:split], dataset[split:]

p = train['Pragnency'].values

g = train['Glucose'].values

bp= train['Blod Pressure'].values

st= train['Skin Thikness'].values

ins= train['Insulin'].values

bmi= train['BMI'].values

dfp= train['DFP'].values

a= train['Age'].values

d= train['Diabetes'].values

trainfeatures=zip(p,g,bp,st,ins,bmi,dfp,a)

traininput=list(trainfeatures)

#print(traininput)

model = GaussianNB()

model.fit(traininput,d)

p = test['Pragnency'].values

g = test['Glucose'].values

bp= test['Blod Pressure'].values

st= test['Skin Thikness'].values

ins= test['Insulin'].values

bmi= test['BMI'].values

dpf= test['DFP'].values

a= test['Age'].values

d= test['Diabetes'].values

testfeatures=zip(p,g,bp,st,ins,bmi,dpf,a)

testinput=list(testfeatures)

predicted= model.predict(testinput)

print("Actual Class:   ", \*d)

print("Predicted Class:", \*predicted)

print("Confusion Matrix")

print(metrics.confusion\_matrix(d, predicted))

print("\*\*\*\*\*\*\*\*Classifiaction Measures\*\*\*\*\*\*\*\*\*")

print("Accuracy:",metrics.accuracy\_score(d,predicted))

print("Recall:",metrics.recall\_score(d,predicted))

print("Precision:",metrics.precision\_score(d,predicted))

print("F1-Score:",metrics.f1\_score(d,predicted))

import pandas as pd

from sklearn import metrics

from sklearn.tree import DecisionTreeClassifier

dataset = pd.read\_csv('/content/drive/My Drive/Data/Diabetes.csv')

split = int(len(dataset)\*0.7)

train, test = dataset[:split], dataset[split:]

p = train['Pragnency'].values

g = train['Glucose'].values

bp= train['Blod Pressure'].values

st= train['Skin Thikness'].values

ins= train['Insulin'].values

bmi= train['BMI'].values

dpf= train['DFP'].values

a= train['Age'].values

d= train['Diabetes'].values

trainfeatures=zip(p,g,bp,st,ins,bmi,dpf,a)

traininput=list(trainfeatures)

model = DecisionTreeClassifier(criterion = "entropy", max\_depth=8)

model.fit(traininput,d)

p = test['Pragnency'].values

g = test['Glucose'].values

bp= test['Blod Pressure'].values

st= test['Skin Thikness'].values

ins= test['Insulin'].values

bmi= test['BMI'].values

dpf= test['DFP'].values

a= test['Age'].values

d= test['Diabetes'].values

testfeatures=zip(p,g,bp,st,ins,bmi,dpf,a)

testinput=list(testfeatures)

predicted= model.predict(testinput)

print("Actual Class:   ",\*d)

print("Predicted Class:",\*predicted)

print("Confusion Matrix")

print(metrics.confusion\_matrix(d, predicted))

print("\*\*\*\*\*\*\*\*Classifiaction Measures\*\*\*\*\*\*\*\*\*")

acc=metrics.accuracy\_score(d,predicted)

f1=metrics.f1\_score(d, predicted)

rec=metrics.recall\_score(d, predicted)

pre= metrics.precision\_score(d, predicted)

print("Accuracy Score:",acc)

print("Recall Score:",rec)

print("Precision-Score:",pre)

print("F1-Score:",f1)

#Multi-class Classification Using MLP

import pandas as pd

import numpy as np

from sklearn import metrics

from sklearn.preprocessing import LabelEncoder

from keras.models import Sequential

from keras.layers import Dense

from keras.utils import np\_utils

from sklearn.preprocessing import StandardScaler

from sklearn.utils import shuffle

from sklearn.metrics import classification\_report

dataset = pd.read\_csv('/content/drive/My Drive/Data/iris.csv')

dataset = dataset.values

dataset=shuffle(dataset)

x = dataset[:,0:4].astype(float)

y = dataset[:,4]

# encode class values as integers

encoder = LabelEncoder()

encoder.fit(y)

ey = encoder.transform(y)

# convert integers to dummy variables (i.e. one hot encoded)

dy = np\_utils.to\_categorical(ey)

#print(\*y)

#print(\*ey)

#print(\*dy)

#Normalize input attributes

sc=StandardScaler().fit(x)

sx=sc.transform(x)

#Train/Test Split

split = int(len(x)\*0.7)

trainx, testx = sx[:split], sx[split:]

trainy, testy = dy[:split], dy[split:]

# define the keras model

model = Sequential()

model.add(Dense(64, input\_dim=4, activation='relu'))

model.add(Dense(32, activation='relu'))

model.add(Dense(16, activation='relu'))

model.add(Dense(units = 3, activation='softmax'))

# compile and fit the model

model.compile(optimizer = 'adam', loss = 'categorical\_crossentropy',  metrics = ['accuracy'])

model.fit(trainx, trainy, epochs=20, batch\_size=8,  verbose=1)

# make class predictions with the model

yp = model.predict(testx)

yp=np.argmax(yp, axis=-1)

yp=yp.ravel()

a=list()

for i in range(len(testy)):

  d=np.argmax(testy[i])

  a.append(d)

a=np.array(a)

al=encoder.inverse\_transform(a)

pl=encoder.inverse\_transform(yp)

print("Actual Class:  ",\*al)

print("Predicted Class:",\*pl)

print(classification\_report(al, pl))

# Sample code to do FP-Growth in Python

import pyfpgrowth

# Creating Sample Transactions

transactions = [

['Milk', 'Bread', 'Saffron'],

['Milk', 'Saffron'],

['Bread', 'Saffron','Wafer'],

['Bread','Wafer'],

]

#Finding the frequent patterns with min support threshold=0.5

FrequentPatterns=pyfpgrowth.find\_frequent\_patterns(transactions=transactions,support\_threshold=0.5)

print(FrequentPatterns)

# Generating rules with min confidence threshold=0.5

print("Generating rules with min confidence threshold=0.5")

Rules=pyfpgrowth.generate\_association\_rules(patterns=FrequentPatterns,confidence\_threshold=0.5)

print(Rules)