**Homework 1**

**Varun Hoskere**

import pandas as pd

import matplotlib.pyplot as plt

data = pd.read\_csv('Breast\_Cancer.csv')

values = []

***(a)***

def a():

for variable in list(data.columns):

for i in data[variable]:

if i not in values:

values.append(i)

plt.hist(data[variable], bins=len(values), color='blue', edgecolor='black')

plt.title(f'{variable} distribution')

plt.xlabel(f'{variable}')

plt.ylabel('Number of individuals')

plt.savefig(f'{variable} plot.png')

# plot.show()

return

A diagram of a stage distribution

Description automatically generatedA diagram of a stage distribution

Description automatically generatedA graph of age distribution

Description automatically generatedA graph with a bar and text

Description automatically generated with medium confidenceA graph with blue lines

Description automatically generatedA graph with blue lines

Description automatically generatedA graph with blue bars

Description automatically generatedA diagram with blue lines

Description automatically generatedA graph with a bar and text

Description automatically generated with medium confidenceA graph with blue lines

Description automatically generatedA graph of a distribution of a region

Description automatically generated with medium confidenceA graph of a distribution of a region

Description automatically generated with medium confidenceA white rectangular graph with blue lines

Description automatically generatedA graph of a survival

Description automatically generatedA diagram of a stage distribution

Description automatically generatedA graph of a number of blue columns

Description automatically generated

***(b)***

def b():

# Age, Regional node examined, regional nodde +ve, survival months, tumour size are the continuous variables

# variables = ['Age', 'Regional Node Examined', 'Regional Node Positive', 'Tumor Size']

# for var in variables:

# plot.scatter(data[var], data['Survival Months'])

# plot.savefig(f'{var} vs Surival Months')

plt.scatter(data['Age'], data['Survival Months'], color='blue')

plt.xlabel('Age')

plt.ylabel('Survivial Months')

# plot.show()

plt.savefig('Age vs Survival Months')

plt.scatter(data['Regional Node Examined'], data['Survival Months'], color='blue')

plt.xlabel('Regional Node Examined')

plt.ylabel('Survivial Months')

# plt.show()

plt.savefig('Regional Node Examined vs Survival Months')

plt.scatter(data['Tumor Size'], data['Survival Months'], color='blue')

plt.xlabel('Tumor Size')

plt.ylabel('Survivial Months')

# plt.show()

plt.savefig('Tumor Size vs Survival Months')

plt.scatter(data['Regional Node Positive'], data['Survival Months'], color='blue')

plt.xlabel('Regional Node Positive')

plt.ylabel('Survivial Months')

plt.savefig('Regional Node Positive vs Survival Months')

# plt.show()

return

def bPearsons():

print("===============")

print("Pearsons coefficient against Survival Months:")

print("Age: ", data['Age'].corr(data['Survival Months']))

print("Regional Node Examined", data['Regional Node Examined'].corr(data['Survival Months']))

print("Regional Node Positive", data['Regional Node Positive'].corr(data['Survival Months']))

print("Tumor Size", data['Tumor Size'].corr(data['Survival Months']))

print("===============")

print()

return

A computer screen shot of a computer

Description automatically generated

A blue dotted chart with numbers

Description automatically generated

A blue dot diagram with white text

Description automatically generatedA blue square with white text

Description automatically generatedA blue square with white text

Description automatically generated

***(c)***

def variable\_type(data, column\_name, threshold=2):

# print(data[column\_name])

unique\_values = []

for point in data[column\_name]:

if point not in unique\_values:

unique\_values.append(point)

if len(unique\_values) > 10:

return 0

return 1

def c():

status = ['Alive', 'Dead']

vars\_x = []

for variable in list(data.columns):

if variable\_type(data, variable):

vars\_x.append(variable)

categories = {}

for var in vars\_x:

categories[var] = []

for i in data[var]:

if i not in categories[var]:

categories[var].append(i)

for category in categories.keys():

miniDF = data.groupby([f'{category}', 'Status']).size().reset\_index(name='count')

finalDF = miniDF.pivot(index=f'{category}', columns='Status', values='count')

finalDF.plot(kind='bar', stacked=False)

plt.xlabel(f'{category}')

plt.ylabel('Status')

plt.xticks(rotation=45, ha='right')

plt.savefig(f'{category} vs Status.png')

plt.show()

A screenshot of a graph

Description automatically generated

A screenshot of a computer

Description automatically generatedA screenshot of a graph

Description automatically generatedA screenshot of a graph

Description automatically generatedA screenshot of a graph

Description automatically generatedA screenshot of a graph

Description automatically generatedA screenshot of a graph

Description automatically generatedA screenshot of a graph

Description automatically generatedA screenshot of a graph

Description automatically generatedA screenshot of a graph

Description automatically generated

***(d)*** K N N Algorithm

import pandas as pd

import math

from collections import Counter

# import time as t

from tqdm import tqdm

import random

data = pd.read\_csv('Breast\_Cancer.csv')

def compute\_distances(point1, point2):

"""

Age, survival months, regional node positive, regional node examined and Tumor size

are the continuous variables. Euclidean distance is used for measuring similarity

between these variables.

Race, Marital Status, T Stage, N Stage, 6th Stage, Defferentiated, Grade, A Stage,

Estrogen Status, Progesterone Status are the categorical values. Hamming distance

is used for measuring the similarity across these variables.

"""

euclidean\_distance = math.sqrt(

(point1['Age'] - point2['Age'] ) \*\*2+

(point1['Tumor Size'] - point2['Tumor Size'] ) \*\*2+

(point1['Regional Node Examined'] - point2['Regional Node Examined']) \*\*2+

(point1['Regional Node Positive'] - point2['Regional Node Positive'] ) \*\*2+

(point1['Survival Months'] - point2['Survival Months'])\*\*2

)

# print("ed: ", euclidean\_distance)

hamming\_distance = (

(0 if point1['Race']==point2['Race'] else 1) +

(0 if point1['Marital Status']==point2['Marital Status'] else 1) +

(0 if point1['T Stage']==point2['T Stage'] else 1) +

(0 if point1['N Stage']==point2['N Stage'] else 1) +

(0 if point1['6th Stage']==point2['6th Stage'] else 1) +

(0 if point1['differentiate']==point2['differentiate'] else 1) +

(0 if point1['Grade']==point2['Grade'] else 1) +

(0 if point1['A Stage']==point2['A Stage'] else 1) +

(0 if point1['Estrogen Status']==point2['Estrogen Status'] else 1) +

(0 if point1['Progesterone Status']==point2['Progesterone Status'] else 1)

)

# print("hd: ", hamming\_distance)

return euclidean\_distance + hamming\_distance

def split\_dataset(data):

totalRows = data.shape[0] - 1

"""

split data into train, validation and testing sets : 75-15-15% each

find the total size of the dataset and \*0.75, .15, .15

"""

train\_boundary = math.floor(0.70\*totalRows)

val\_boundary = train\_boundary + math.ceil(0.15\*totalRows)

test\_boundary = val\_boundary + math.ceil(0.15\*totalRows)

train\_data = data.iloc[:train\_boundary]

val\_data = data.iloc[train\_boundary:val\_boundary]

test\_data = data.iloc[val\_boundary:test\_boundary]

train\_Y = train\_data['Status']

train\_X = train\_data.drop(['Status'], axis=1)

val\_Y = val\_data['Status']

val\_X = val\_data.drop(['Status'], axis=1)

test\_Y = test\_data['Status']

test\_X = test\_data.drop(['Status'], axis=1)

# print(train\_X.shape[0])

# print(val\_X.shape[0])

return train\_X, train\_Y, val\_X, val\_Y, test\_X, test\_Y

data['ID'] = data.index

train\_X, train\_Y, val\_X, val\_Y, test\_X, test\_Y = split\_dataset(data.sample(frac=1))

point\_distance\_map = {}

val\_point\_to\_sorted\_distances = {}

for val\_index in tqdm(range(val\_X.shape[0])):

for train\_index in range(train\_X.shape[0]):

distance = compute\_distances(val\_X.iloc[val\_index], train\_X.iloc[train\_index])

point\_distance\_map[train\_index] = distance

sorted\_distances = dict(sorted(point\_distance\_map.items(), key=lambda item: item[1]))

val\_point\_to\_sorted\_distances[val\_index] = sorted\_distances

def get\_k\_neighbours(k):

kNeighbours = {}

for val\_point in val\_point\_to\_sorted\_distances:

neighbours = val\_point\_to\_sorted\_distances[val\_point]

kNeighbours[val\_point] = list(neighbours.keys())[:k]

return kNeighbours

def predict\_for\_val(kNeighbours):

validation\_predictions = {}

for val\_point in kNeighbours.keys():

output = []

for neigh in kNeighbours[val\_point]:

# print(train\_X.loc[train\_X['ID'] == neigh])

output.append(train\_Y.iloc[neigh])

pred\_status, trash = Counter(output).most\_common()[0]

validation\_predictions[val\_point] = pred\_status

return validation\_predictions

tp = {}

fp = {}

tn = {}

fn = {}

def confusion\_matrix(predictions, k, actual\_Y):

global tp

global tn

global fn

global fp

tp.setdefault(k,0)

tn.setdefault(k,0)

fn.setdefault(k,0)

fp.setdefault(k,0)

for point in predictions:

if predictions[point] == actual\_Y.iloc[point]:

if actual\_Y.iloc[point] == 'Alive':

tp[k]+=1

else:

tn[k]+=1

else:

if actual\_Y.iloc[point] == 'Alive':

fn[k]+=1

else:

fp[k]+=1

def compute\_accuracies(tp, fp, tn, fn, k):

recall\_1 = tp[k]/(tp[k]+fn[k])

recall\_2 = tn[k]/(tn[k]+fp[k])

return ((tp[k]+tn[k])/(tp[k]+tn[k]+fp[k]+tn[k])) , (0.5\*(recall\_1 + recall\_2)) , (2\*tp[k] / (2\*tp[k] + fp[k] + fn[k]))

accuracies = []

bAccuracies = []

f1\_scores = []

for k in [1,3,5,7,9,11]:

kNeighbours = get\_k\_neighbours(k)

validation\_predictions = predict\_for\_val(kNeighbours=kNeighbours)

confusion\_matrix(validation\_predictions, k, val\_Y)

acc , bAcc , f1 = compute\_accuracies(tp, fp, tn, fn, k)

accuracies.append(acc)

bAccuracies.append(bAcc)

f1\_scores.append(f1)

print(f"For k = {k} : Accuracy = {acc}------Balanced Accuracy = {bAcc}------F1 score = {f1}")

print("==================================================================================================================")

A screenshot of a computer

Description automatically generated

import matplotlib.pyplot as plt

plt.scatter(accuracies, [1,3,5,7,9,11])

A graph with blue dots

Description automatically generated

plt.scatter(bAccuracies, [1,3,5,7,9,11])

A graph with numbers and points

Description automatically generated

plt.scatter(f1\_scores, [1,3,5,7,9,11])

A graph with blue dots

Description automatically generated

point\_distance\_map = {}

test\_point\_to\_sorted\_distances = {}

for test\_index in tqdm(range(test\_X.shape[0])):

for train\_index in range(train\_X.shape[0]):

distance = compute\_distances(test\_X.iloc[test\_index], train\_X.iloc[train\_index])

point\_distance\_map[train\_index] = distance

sorted\_distances = dict(sorted(point\_distance\_map.items(), key=lambda item: item[1]))

test\_point\_to\_sorted\_distances[test\_index] = sorted\_distances

def get\_k\_neighbours(k):

kNeighbours = {}

for test\_point in test\_point\_to\_sorted\_distances:

neighbours = test\_point\_to\_sorted\_distances[test\_point]

kNeighbours[test\_point] = list(neighbours.keys())[:k]

return kNeighbours

def predict\_for\_test(kNeighbours):

test\_predictions = {}

for test\_point in kNeighbours.keys():

output = []

for neigh in kNeighbours[test\_point]:

# print(train\_X.loc[train\_X['ID'] == neigh])

output.append(train\_Y.iloc[neigh])

pred\_status, trash = Counter(output).most\_common()[0]

test\_predictions[test\_point] = pred\_status

return test\_predictions

for k in [9,11]:

kNeighbours = get\_k\_neighbours(k)

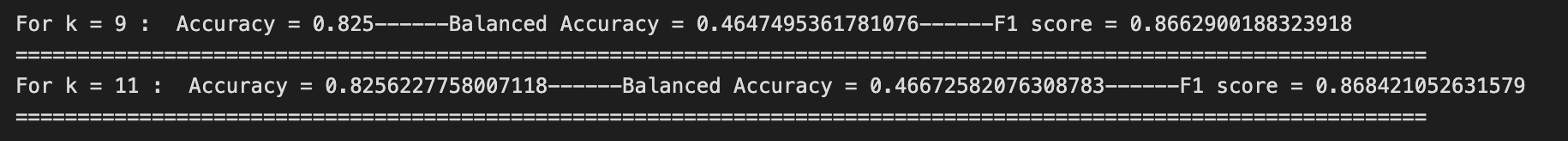
test\_predictions = predict\_for\_test(kNeighbours=kNeighbours)

confusion\_matrix(test\_predictions, k, test\_Y)

acc , bAcc , f1 = compute\_accuracies(tp, fp, tn, fn, k)

print(f"For k = {k} : Accuracy = {acc}------Balanced Accuracy = {bAcc}------F1 score = {f1}")

print("==================================================================================================================")



K\* = 9, K\*\*9 = , K+ = 11

***(e)*** I do not think that would be a good idea because the data on which the model is trained is dominated by people belonging to the race ‘White’. Any predictions made based on the given dataset will inevitably be biased towards the status of individuals belonging to the ‘White’ race.

(d)

tp\_W = 0

tn\_W = 0

fn\_W = 0

fp\_W = 0

tp\_B = 0

tn\_B = 0

fn\_B = 0

fp\_B = 0

tp\_O = 0

tn\_O = 0

fn\_O = 0

fp\_O = 0

def confusion\_matrix\_optimal(predictions, k, actual\_Y):

for point in predictions:

if test\_X.iloc[point]['Race'] == "White" :

if predictions[point] == actual\_Y.iloc[point]:

if actual\_Y.iloc[point] == 'Alive':

global tp\_W

tp\_W+=1

else:

global tn\_W

tn\_W+=1

else:

if actual\_Y.iloc[point] == 'Alive':

global fn\_W

fn\_W+=1

else:

global fp\_W

fp\_W+=1

elif test\_X.iloc[point]['Race'] == "Black":

if predictions[point] == actual\_Y.iloc[point]:

if actual\_Y.iloc[point] == 'Alive':

global tp\_B

tp\_B+=1

else:

global tn\_B

tn\_B+=1

else:

if actual\_Y.iloc[point] == 'Alive':

global fn\_B

fn\_B+=1

else:

global fp\_B

fp\_B+=1

elif test\_X.iloc[point]['Race'] == "Other":

if predictions[point] == actual\_Y.iloc[point]:

if actual\_Y.iloc[point] == 'Alive':

global tp\_O

tp\_O+=1

else:

global tn\_O

tn\_O+=1

else:

if actual\_Y.iloc[point] == 'Alive':

global fn\_O

fn\_O+=1

else:

global fp\_O

fp\_O+=1

kNeighbours = get\_k\_neighbours(11)

test\_predictions = predict\_for\_test(kNeighbours)

confusion\_matrix\_optimal(test\_predictions, 11, test\_Y)

def compute\_accuracies\_optimal(tp, fp, tn, fn, k):

recall\_1 = tp/(tp+fn)

recall\_2 = tn/(tn+fp)

return ((tp+tn)/(tp+tn+fp+tn)) , (0.5\*(recall\_1 + recall\_2)) , (2\*tp / (2\*tp + fp + fn))

f1\_W = compute\_accuracies\_optimal(tp=tp\_W, tn=tn\_W, fp=fp\_W, fn=fn\_W, k=11)

f1\_B = compute\_accuracies\_optimal(tp=tp\_B, tn=tn\_B, fp=fp\_B, fn=fn\_B, k=11)

f1\_O = compute\_accuracies\_optimal(tp=tp\_O, tn=tn\_O, fp=fp\_O, fn=fn\_O, k=11)

f1\_W[2], f1\_B[2], f1\_O[2]



F1 Scored for

1. White: 0.8706
2. Black: 0.8387
3. Other: 0.8641

Comments:   
  
The above code only implements a basic version of the KNN algorithm. There are multiple scopes for optimization, which I have not implemented due to lack of time. For example, the functions for computing distances between two validation and train can be generalized, and then would not be needed to be duplicated and modified for computing the same distance set between train and test sets. The distance computation has scope for performance optimization by using numpy arrays. However, for simplicity’s sake I stuck with straightforward simple approach.