Amirkabir University of Technology
(Tehran Polytechnic)
Department of Computer Engineering

Statistical Pattern Recognition

Assignment III

Soroush Mahdi
99131050

supervised by
Dr. Mohammad Rahmati

January 19, 2021

Contents

1 Is Oxford/AstraZeneca Vaccine Safe?

1.1 a

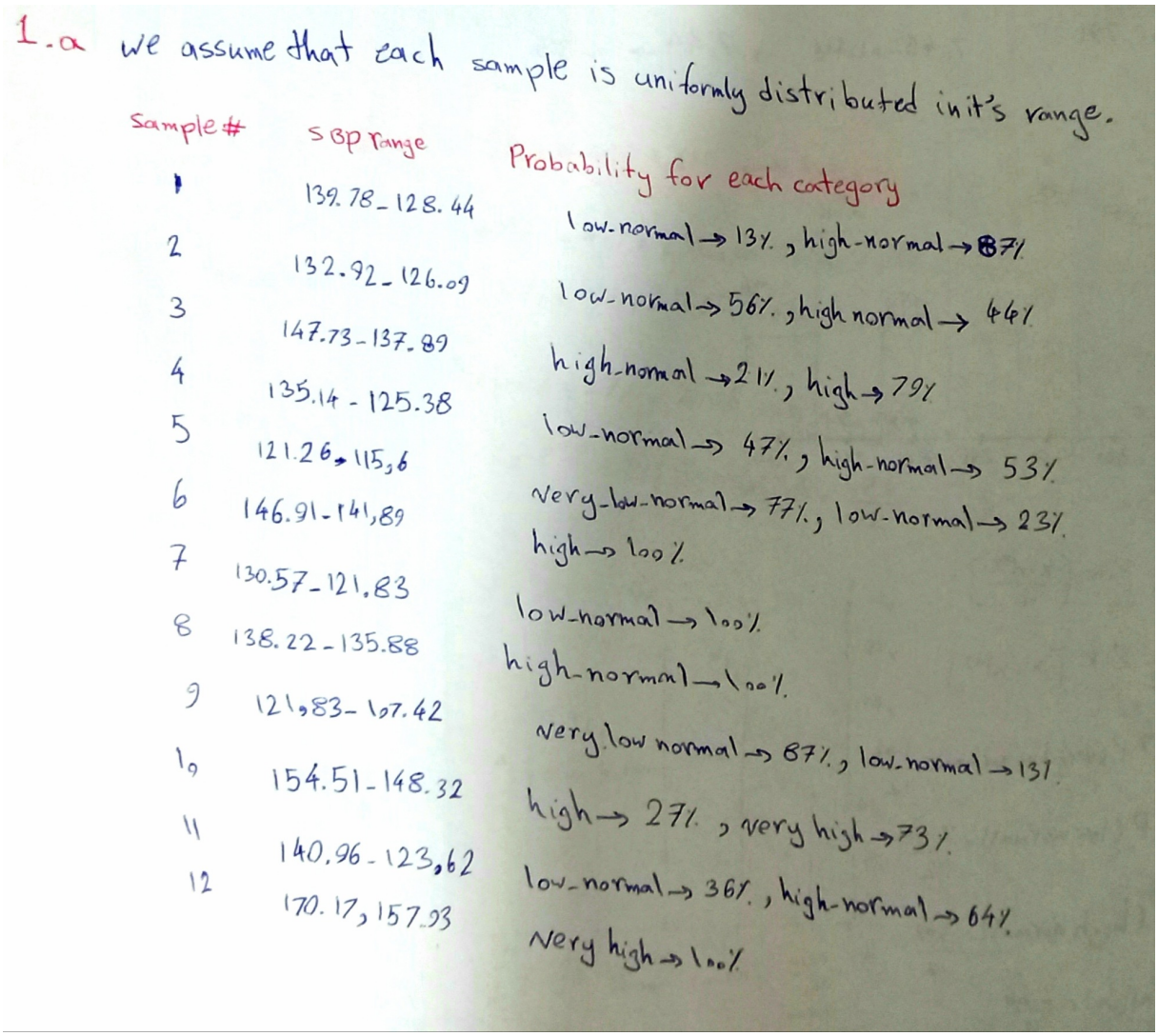

Figure 1: a1

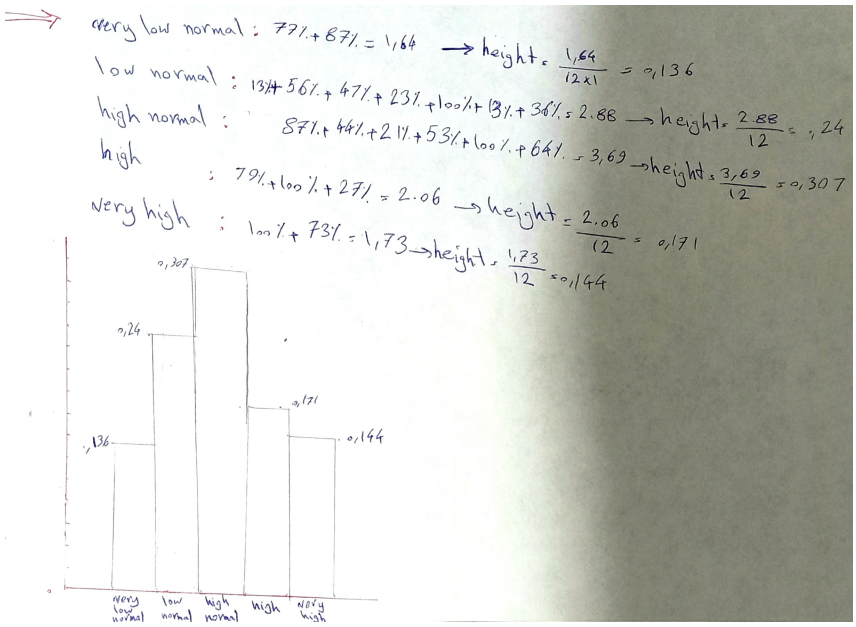

Figure 2: a2

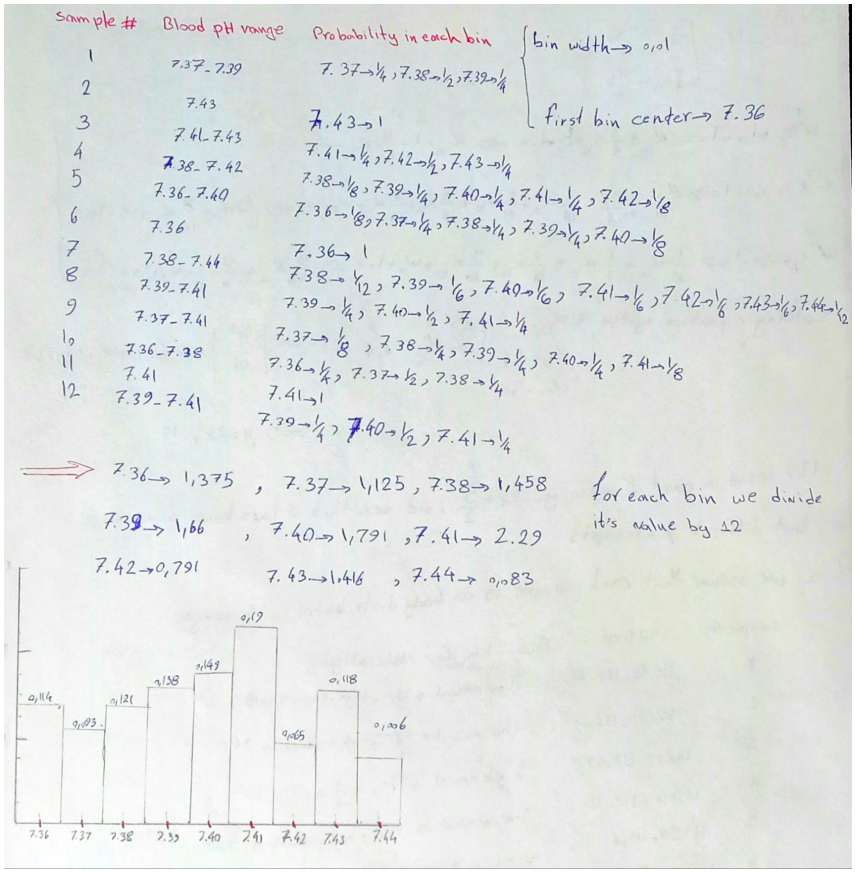

Figure 3: a3

1.2 b

in this section i used python. again i assumed that each sample is uniformly distributed in it's range.

In [19]:
```
SBP = [134.11 , 129.53 , 142.81 , 130.26 , 118.43, 144.4 , 126.2 , 137.05 , 114.63,151.42,132.29,164.05]
SBP_range = [5.67 , 3.44 , 4.92,4.88,2.83,2.51,4.37,1.17,7.21,3.1,8.67,6.12]
#this list is for keeping ranges and value of uniform distribution for each sample
l = []
for i in range(12):
    l.append([SBP[i]-SBP_range[i],SBP[i] + SBP_range[i] , 1./(2*SBP_range[i])])
```

now for phi function of parzan window where its equal to one i put the probabilty of each uniform distribution instead of one

In [23]:
```
parzan=[0 for i in range(105,173)]
for i in range(105,173):
    for j in l:
        #h = 1
        if i>=j[0]-1 and i<=j[1]+1:
            parzan[i-105]+=j[2]
```

In [24]:
```
from matplotlib import pyplot as plt
plt.plot(range(105,173) , parzan )
```

Out[24]:
```
[<matplotlib.lines.Line2D at 0x7f8c21a11100>]
```

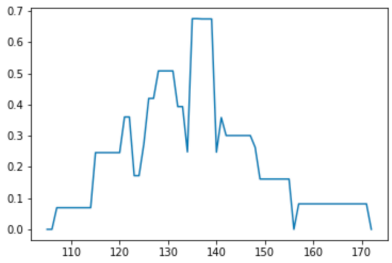

Figure 4: b1

In [28]:
```
parzan=[0 for i in range(105,173)]
for i in range(105,173):
    for j in l:
        #h = 3
        if i>=j[0]-3 and i<=j[1]+3:
            parzan[i-105]+=j[2]
```

In [26]:
```
plt.plot(range(105,173) , parzan )
```

Out[26]:
```
[<matplotlib.lines.Line2D at 0x7f8c215cc160>]
```

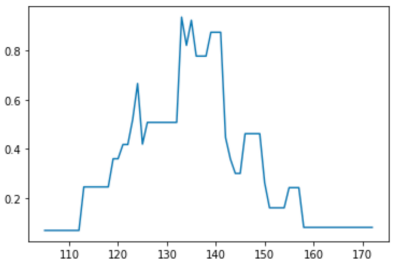

Figure 5: b2

2 Beyond Density Estimation: k-NN is Jack of All Trades!

2.1 a-b

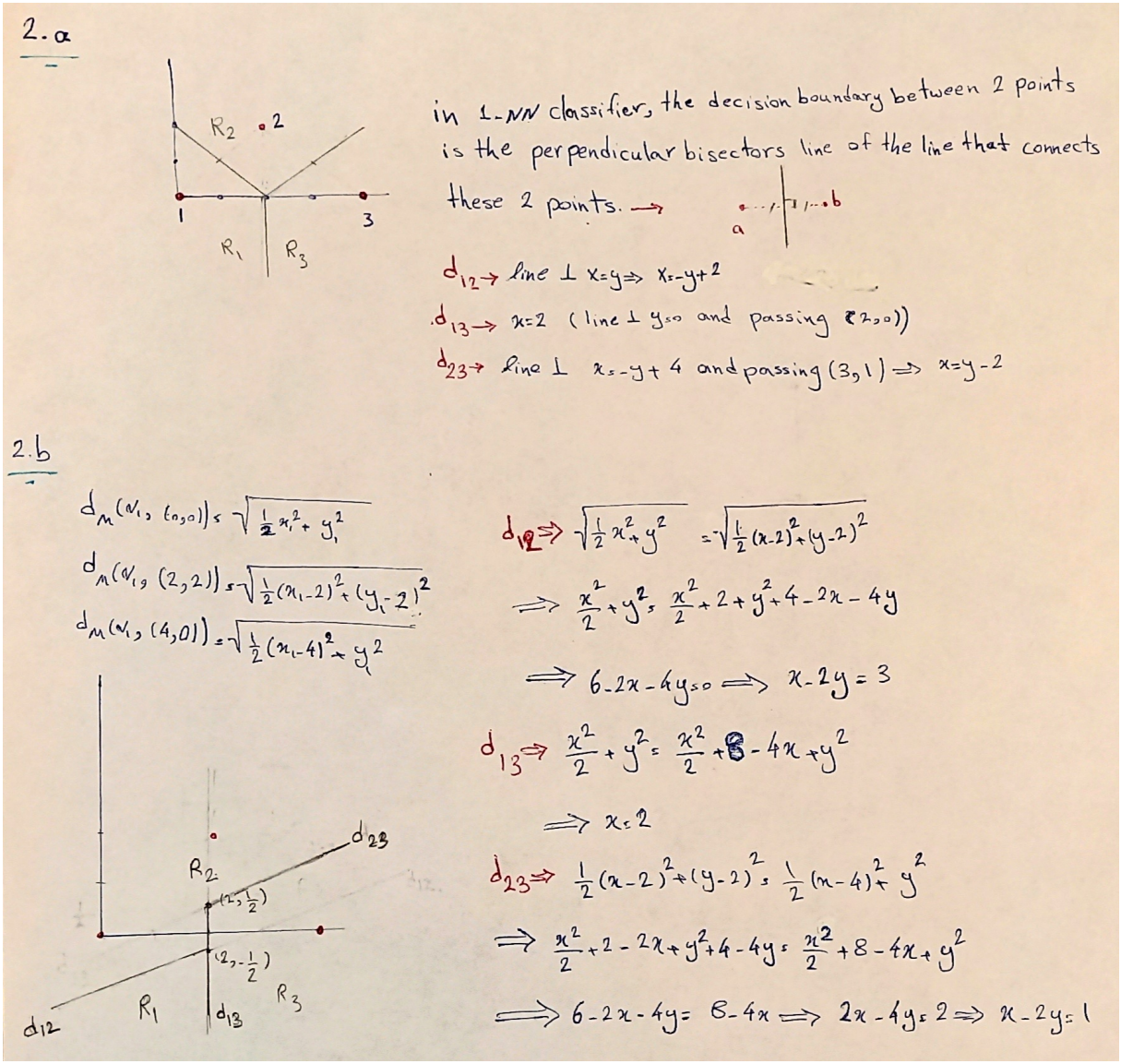

Figure 6: a-b

2.2 c-d

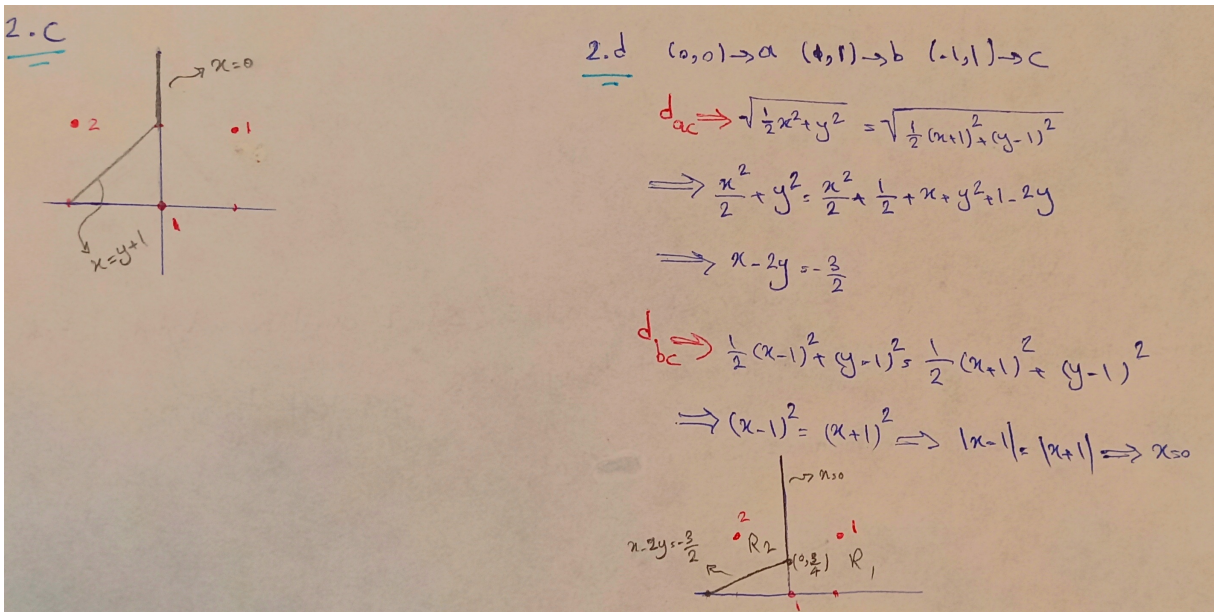

Figure 7: c-d

2.3 e-f

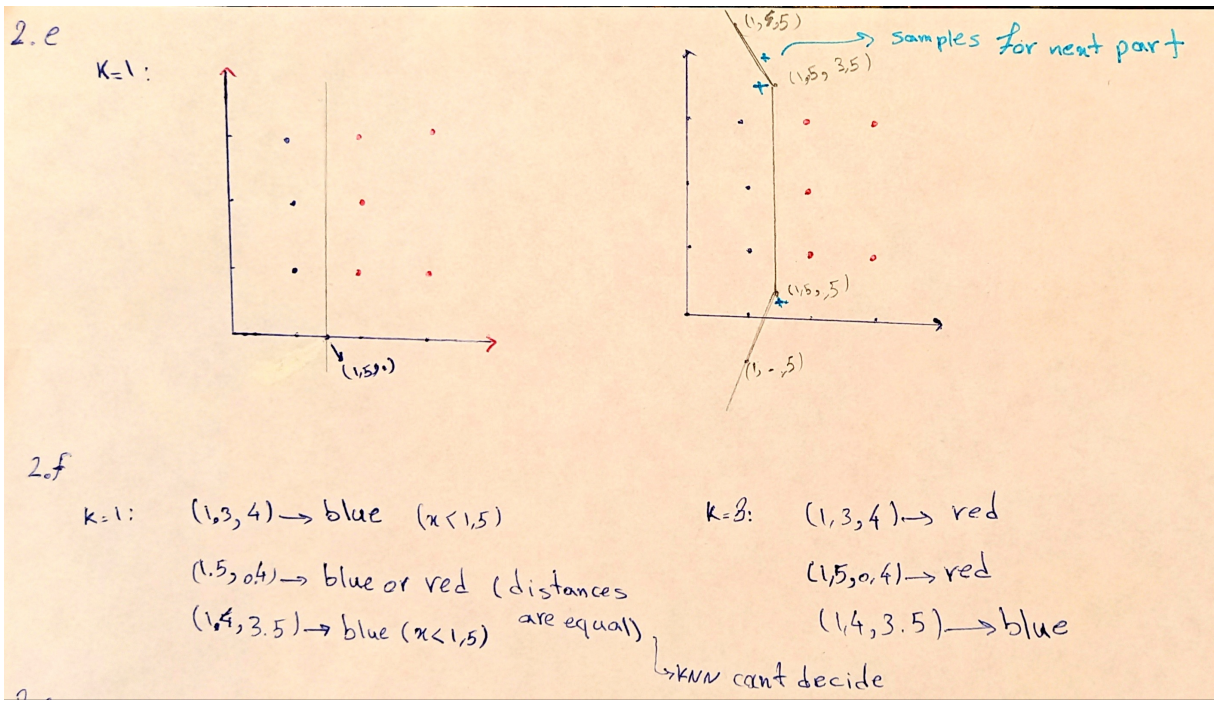

Figure 8: e-f

2.4 g-i

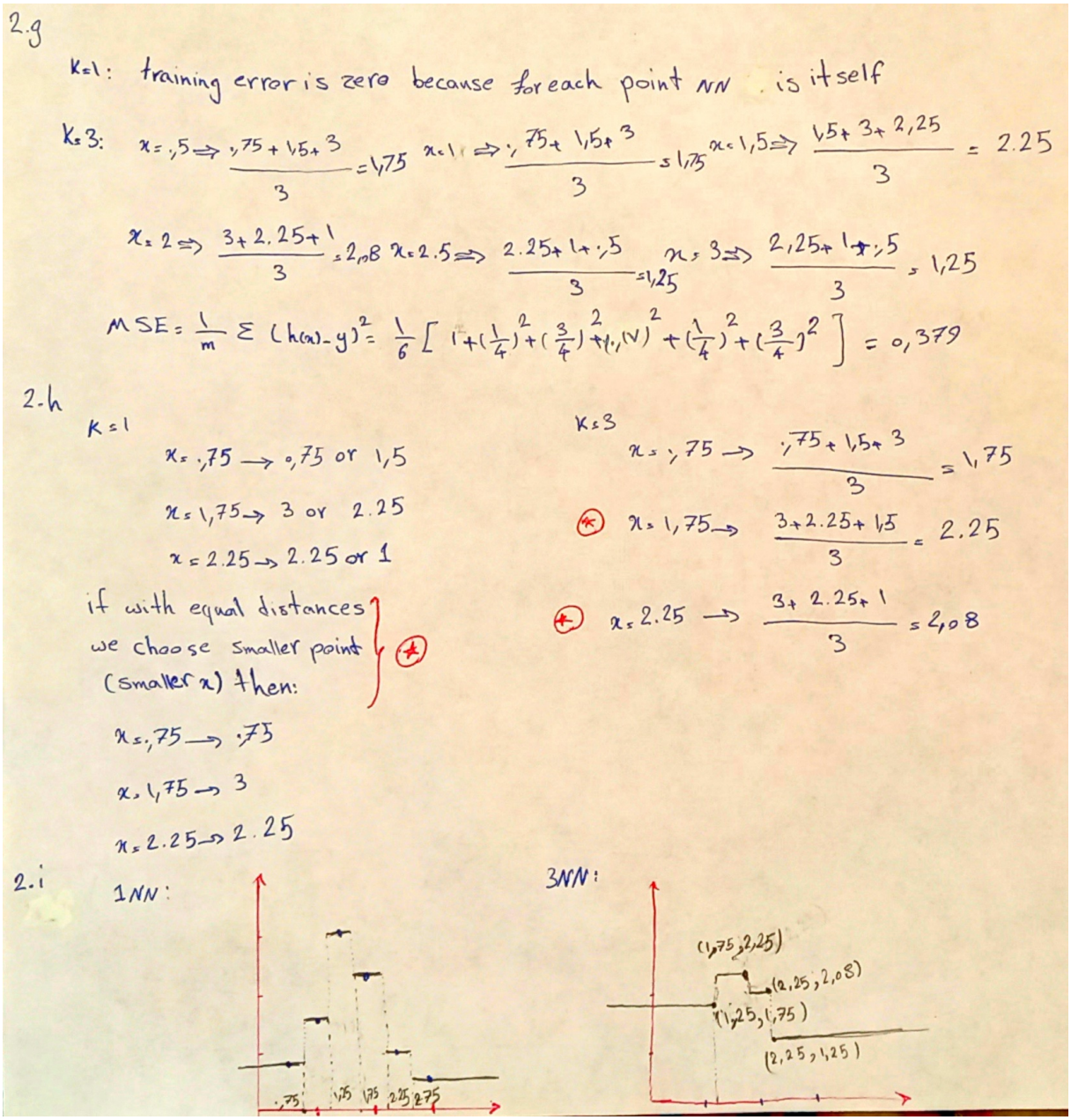

Figure 9: g-i

3 Never Get Fooled Again: Fake Bill Detection System

3.1 a

functions for executing KNN and computing accuracy:

```
def accuracy(y_true , y_predicted):
    """caculate num of correctly predicted samples / num of all samples

    class of samples should be 0 or 1
    Args:
        y_true ([numpy array n*1]): [true value of labels]
        y_predicted ([numpy array n*1]): [predicted value of labels]

```

```
    Returns:
        [float]: [accuracy of prediction]
    """

    difference = y_true - y_predicted
    return np.count_nonzero(difference == 0) / difference.shape[0]

def KNN( X_train , Y_train , X_test , K = 1 , dist_type = 'Euc' ):
    """this functions execute KNN algorithm

    Args:
        X_train ([numpy array m*n]): [features for training]
        Y_train ([numpy array m*1]): [targets for training]
        X_test ([numpy array z*n]): [features for predicting]
        K (int, optional): [num of neighbours]. Defaults to 1.
        dist_type (str, optional): [type of distance, it should be one of these: 'Euc','Mnhtn','Cosin']. Defaults to 'Euc'.

    Returns:
        [numpy array z*1]: [predicted values for X_test]
    """

    #for all types of distances the dists is a z*m matrix which z is num of test points and m is num of train points
    #and in the ith row of dists we have distances for ith test point from all of train points
    if dist_type == 'Euc':
        #calculating L2 norm for each test point from each train point , L2 norm is the Euclidean distance
        dists = np.linalg.norm(X_test[:,np.newaxis]-X_train ,axis = 2)

    if dist_type == 'Mnhtn':
        #calculating manhatan distance for each test point from each train point
        dists = np.sum(np.absolute(X_test[:,np.newaxis]-X_train) , axis = 2)

    if dist_type == 'Cosin':
        #calculating cosin distance for each test point from each train point
        norm_xtrain = np.linalg.norm(X_train,axis = 1 ).reshape(X_train.shape[0],1)
        norm_xtest = np.linalg.norm(X_test,axis = 1 ).reshape(X_test.shape[0],1)
        norms = norm_xtest @ norm_xtrain.T
        dists = 1 - ((X_test @ X_train.T) / norms)

    #choosing min distances
    min_dists_indices = np.argpartition(dists,K,axis = 1)[:,:K]
    #calculating prediction labels
    Y_sum_min_distances = np.sum(Y_train[min_dists_indices] , axis = 1)
    y_pred = Y_sum_min_distances > (K-1)/2

    return y_pred.astype('int64')
```

Listing 1: 3a

data preprocessing:

```
#data preprocessing
data = pd.read_csv('data_banknote_authentication.txt', sep=",", header=None)
data.columns = ['f1' , 'f2' ,'f3'  , 'f4' ,'output']
#shuffling
data = data.sample(frac=1).reset_index(drop=True)
#splitting features and targets
X = data.drop('output' , axis = 1).to_numpy()
Y = data['output'].to_numpy()
#splitting train and test data
x_train = X[:500]
y_train = Y[:500]
x_test = X[500:]
y_test = Y[500:]
```

Listing 2: 3a

testing different features:

```
#testing different features
for i,j in [(1,2),(1,3),(1,4),(2,3),(2,4),(3,4)]:
    x_train_new = x_train[: , [i-1,j-1]]
    x_test_new = x_test[: , [i-1,j-1]]
    y_pre = KNN(x_train_new , y_train ,x_test_new , K=1)
    acc = accuracy(y_test,y_pre)
    print('Features: ' ,i , ' and ' , j , '   accuracy: ' , acc)
```

Listing 3: 3a

```
Features:  1  and  2    accuracy:  0.9208715596330275
Features:  1  and  3    accuracy:  0.8864678899082569
Features:  1  and  4    accuracy:  0.8910550458715596
Features:  2  and  3    accuracy:  0.8990825688073395
Features:  2  and  4    accuracy:  0.8658256880733946
Features:  3  and  4    accuracy:  0.7052752293577982
```

Figure 10: accuracy for different sets of features

we see that the best featuers for 1NN are feature 1 and 2

3.2 b

plotting decision boundaries for 1NN:

```
#seting up colors
cmap_light = ListedColormap(['#FFAAAA','#AAAAFF', '#AAFFAA'])

xx , yy = np.meshgrid(np.arange(-8, 8, .05),
                      np.arange(-15, 15, .05))

#executing 1NN
y_pred= KNN(x_train[: , [0,1]] , y_train ,np.c_[xx.ravel(), yy.ravel()] , K = 1)
y_pred=y_pred.reshape(xx.shape)

plt.figure(figsize=(10,8))
plt.pcolormesh(xx, yy, y_pred,cmap=cmap_light,shading='auto')
plt.savefig('3b.png')
```

Listing 4: 3b

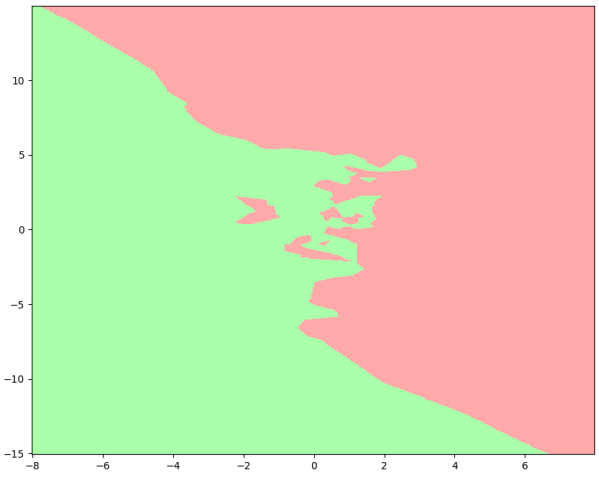

Figure 11: decision boundaries for 1NN

3.3 c

```
#testing different features
for i,j in [(1,2),(1,3),(1,4),(2,3),(2,4),(3,4)]:
    x_train_new = x_train[: , [i-1,j-1]]
    x_test_new = x_test[: , [i-1,j-1]]
    y_pre = KNN(x_train_new , y_train ,x_test_new , K=3)
    acc = accuracy(y_test,y_pre)
    print('Features: ' ,i , ' and ' , j , '   accuracy: ' , acc)
```

Listing 5: 3c

```
Features:  1  and  2    accuracy:  0.9369266055045872
Features:  1  and  3    accuracy:  0.8864678899082569
Features:  1  and  4    accuracy:  0.8876146788990825
Features:  2  and  3    accuracy:  0.8704128440366973
Features:  2  and  4    accuracy:  0.8761467889908257
Features:  3  and  4    accuracy:  0.6972477064220184
```

Figure 12: accuracy for different sets of features

we see that the best featuers for 3NN are feature 1 and 2

3.4 d

plotting decision boundaries for 3NN:

```
y_pred= KNN(x_train[: , [0,1]] , y_train ,np.c_[xx.ravel(), yy.ravel()] , K = 3)
y_pred=y_pred.reshape(xx.shape)
plt.figure(figsize=(10,8))
plt.pcolormesh(xx, yy, y_pred,cmap=cmap_light,shading='auto')
plt.savefig('3d.png')
```

Listing 6: 3d

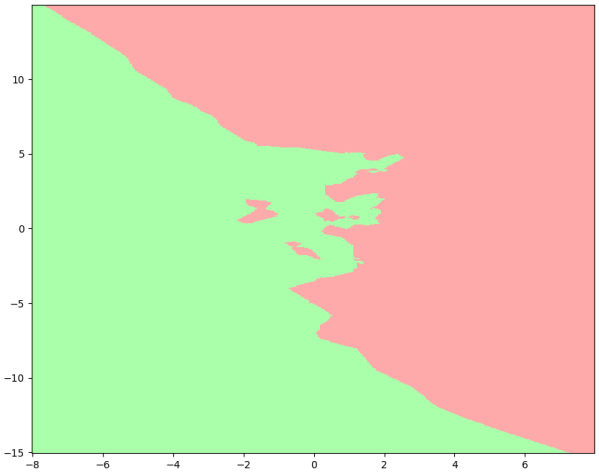

Figure 13: decision boundaries for 3NN

4 Help Trump Fight Against Fake News

4.1 a

preparing data:

```
def prepare_data():

    vectorizer = CountVectorizer()

    #concating fake and real datas and save them in one file
    data = data2 = ""
    # Reading data from file1
    with open('clean_fake.txt') as fp:
        data = fp.read()
    # Reading data from file2
    with open('clean_real.txt') as fp:
        data2 = fp.read()
    # Merging 2 files To add the data of file2 from next line
    data += data2
    #saving new file
    with open ('clean.txt', 'w') as fp:
        fp.write(data)

    corpus = open('clean.txt')
    #vectorizing corpus
    X = vectorizer.fit_transform(corpus)
    #changing x type to numpy array
    X = np.array(X.toarray())
    #fake : class 1 , real : class 0
    target = np.concatenate((np.ones((1298,1)) , np.zeros((1968,1))) , axis = 0)
    #concating targets to X for each row
    X = np.concatenate((X , target),axis = 1)
    #shuffling X
    np.random.shuffle(X)
    #splitting train , test and validation sets
    train = X[:int(len(X)*.7)]
    test = X[int(len(X)*.7) : int(len(X)*.85)]
    valid = X[int(len(X)*.85):]

    return train,test,valid
```

Listing 7: 4a

4.2 b

model selection:

```
def knn_model_selection(dist_type = 'minkowski'):
    train , test , valid = prepare_data()
    train_acc = []
    valid_acc =[]
    val_acc = 0
    best_k = 0
    for i in range(1,21):
        print('k: ',i)
        #initializing model
        neigh = KNeighborsClassifier(n_neighbors=i , metric=dist_type)
        #model training
        neigh.fit(train[: , :-1], train[: , -1])
        train_er = 1-neigh.score(train[: , :-1], train[: , -1])
        print('train error : ',train_er)
        train_acc.append(1-train_er)
        val_er = 1-neigh.score(valid[: , :-1], valid[: , -1])
        print('validation error : ',val_er)
        valid_acc.append(1-val_er)
        if (1-val_er) > val_acc:
            val_acc = 1-val_er
            best_k = i

```

```
23      print('Based on validation accuracy best k is : ' , best_k)
24      return train_acc , valid_acc , best_k
```

Listing 8: 4b-1

plotting accuracy based on K:

```
train_acc , valid_acc , best_k = knn_model_selection()
plt.plot(range(1,21),train_acc , label = 'train accuracy' )
plt.plot(range(1,21),valid_acc , label = 'validation accuracy')
plt.legend()
plt.xlabel('K')
plt.ylabel('accuracy')
plt.savefig('4b.png')
```

Listing 9: 4b-2

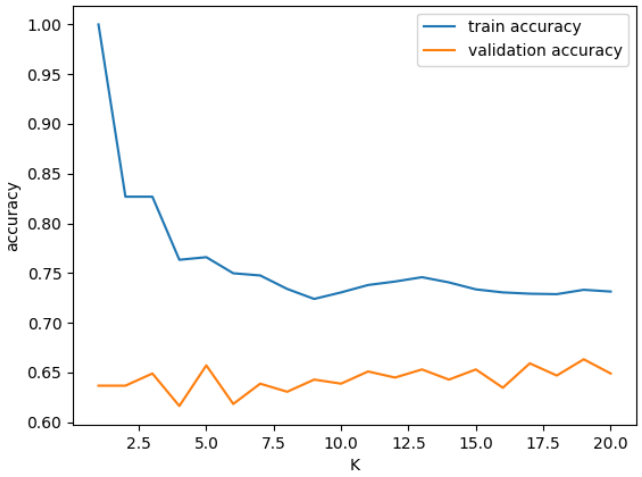

Figure 14: accuracy for different values of K

```
k:  1
train error :  0.0
validation error :  0.363265306122449
k:  2
train error :  0.17322834645669294
validation error :  0.363265306122449
k:  3
train error :  0.17322834645669294
validation error :  0.3510204081632653
k:  4
train error :  0.23665791776028
validation error :  0.38367346938775515
k:  5
train error :  0.23403324584426943
validation error :  0.34285714285714286
k:  6
train error :  0.2502187226596675
validation error :  0.3816326530612245
k:  7
train error :  0.25240594925634297
validation error :  0.36122448979591837
k:  8
train error :  0.26596675415573057
validation error :  0.3693877551020408
k:  9
train error :  0.2760279965004374
validation error :  0.3571428571428571
k:  10
train error :  0.2694663167104112
validation error :  0.36122448979591837
```

Figure 15: error for different values of K

```
k:  11
train error :  0.2620297462817148
validation error :  0.34897959183673466
k:  12
train error :  0.25853018372703407
validation error :  0.35510204081632657
k:  13
train error :  0.25415573053368334
validation error :  0.34693877551020413
k:  14
train error :  0.25940507436570426
validation error :  0.3571428571428571
k:  15
train error :  0.26640419947506566
validation error :  0.34693877551020413
k:  16
train error :  0.2694663167104112
validation error :  0.36530612244897964
k:  17
train error :  0.2707786526684165
validation error :  0.3408163265306122
k:  18
train error :  0.2712160979877515
validation error :  0.35306122448979593
k:  19
train error :  0.26684164479440065
validation error :  0.33673469387755106
k:  20
train error :  0.268591426071741
validation error :  0.3510204081632653
Based on validation accuracy best k is :  19
```

Figure 16: error for different values of K

4.3 c

KNeighborsClassifier() does not have cosine metric so I used KNN and accuracy functions that i implemented in part 3(Fake Bill Detection):

```
def knn_model_selection(dist_type):
    train , test , valid = prepare_data()
    train_acc = []
    valid_acc =[]
    val_acc = 0
    best_k = 0
    for i in range(1,21):
        print('k: ',i)
        y_pred = KNN(train[: , :-1], train[: , -1] ,train[: , :-1] , K = i , dist_type=dist_type )
        train_er = 1-accuracy(train[: , -1] , y_pred)
        print('train error : ',train_er)
        train_acc.append(1-train_er)
        y_pred = KNN(train[: , :-1], train[: , -1] ,valid[: , :-1] , K = i , dist_type=dist_type )
        val_er = 1-accuracy(valid[: , -1] , y_pred)
        print('validation error : ',val_er)
        valid_acc.append(1-val_er)
        if (1-val_er) > val_acc:
            val_acc = 1-val_er
            best_k = i

    print('Based on validation accuracy best k is : ' , best_k)
    return train_acc , valid_acc , best_k
```

Listing 10: 4c

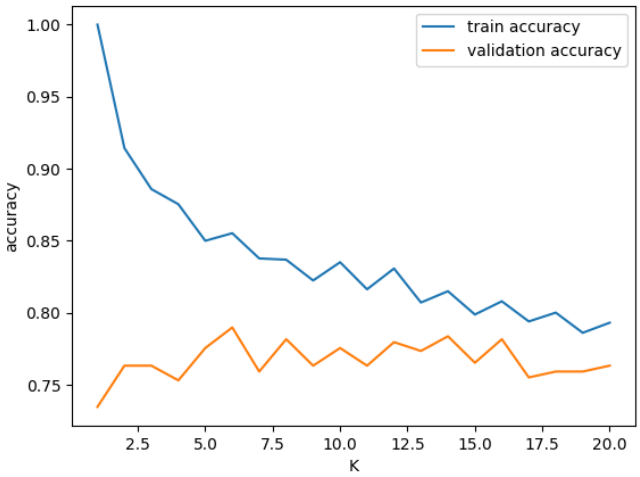

Figure 17: accuracy for different values of K

```
k:  1
train error :  0.0
validation error :  0.26530612244897955
k:  2
train error :  0.08573928258967634
validation error :  0.23673469387755097
k:  3
train error :  0.11417322834645671
validation error :  0.23673469387755097
k:  4
train error :  0.12467191601049865
validation error :  0.24693877551020404
k:  5
train error :  0.15004374453193348
validation error :  0.22448979591836737
k:  6
train error :  0.14479440069991256
validation error :  0.21020408163265303
k:  7
train error :  0.1622922134733158
validation error :  0.24081632653061225
k:  8
train error :  0.16316710411198598
validation error :  0.21836734693877546
k:  9
train error :  0.17760279965004377
validation error :  0.23673469387755097
k:  10
train error :  0.16491688538932636
validation error :  0.22448979591836737
```

Figure 18: error for different values of K

```
k:  11
train error :  0.18372703412073488
validation error :  0.23673469387755097
k:  12
train error :  0.1692913385826772
validation error :  0.2204081632653061
k:  13
train error :  0.19291338582677164
validation error :  0.226530612244898
k:  14
train error :  0.18503937007874016
validation error :  0.21632653061224494
k:  15
train error :  0.20122484689413822
validation error :  0.23469387755102045
k:  16
train error :  0.19203849518810145
validation error :  0.21836734693877546
k:  17
train error :  0.20603674540682415
validation error :  0.24489795918367352
k:  18
train error :  0.19991251093613294
validation error :  0.24081632653061225
k:  19
train error :  0.21391076115485563
validation error :  0.24081632653061225
k:  20
train error :  0.20691163604549434
validation error :  0.23673469387755097
Based on validation accuracy best k is :  6
```

Figure 19: error for different values of K

we see that cosine metric performs better than the Euclidean metric here and if we consider the Hint example we can see that Euclidean distance between dog and dogdogdog is 2 and between CR7 and dog is second root of 2 and in this metric dog NN is CR7 but for cosine distance the first distance is zero and the second distance is one and one is the max distance in cosine distance so NN for dog is dogdogdog.

5 A Glance At the World of Linear Discriminant Analysis

5.1 a-c

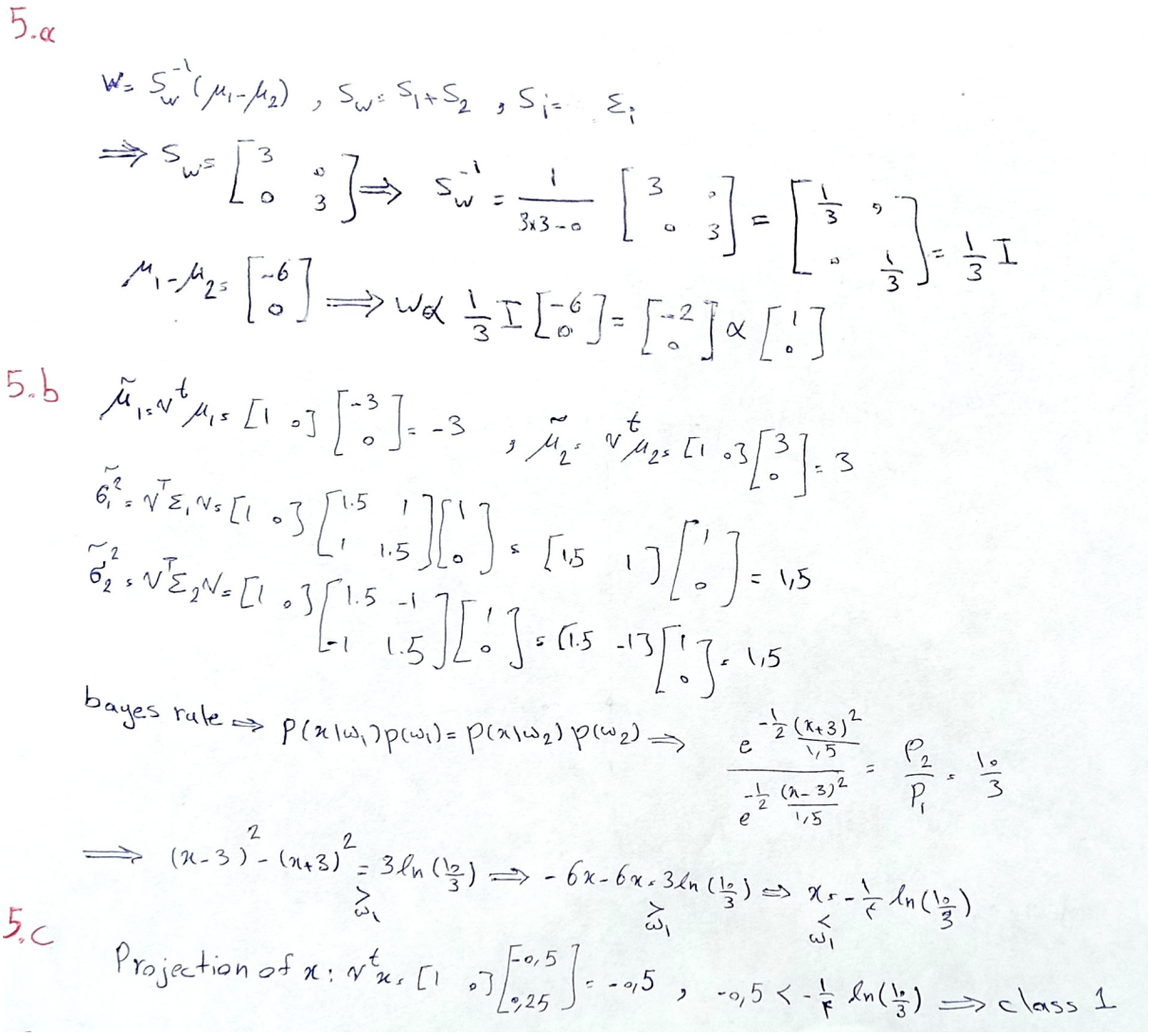

Figure 20: a-c

5.2 d

```
#setting mean vectors
m1 = np.array([-3,0])
m2 = np.array([3,0])
#setting covariance matrices
cov1 = np.array([[1.5,1],[1,1.5]])
cov2 = np.array([[1.5,-1],[-1,1.5]])
#generating samples
x1 = np.random.multivariate_normal(m1, cov1, 500)
x2 = np.random.multivariate_normal(m2, cov2, 500)
#plotting samples and seperating line
fig = plt.figure()
ax1 = fig.add_subplot(111)
ax1.scatter(x1[:,0],x1[:,1] ,label='class 1')
ax1.scatter(x2[:,0],x2[:,1] , label = 'class 2')
ax1.axline((-7,0),(7,0) , label = 'projection line' , color = 'r')
ax1.axline((-0.3,5),(-0.3,-5) , label = 'seperating line' , color = 'g',linestyle='--
    ')
ax1.legend()
fig.savefig('5d.png')
```

Listing 11: 5d

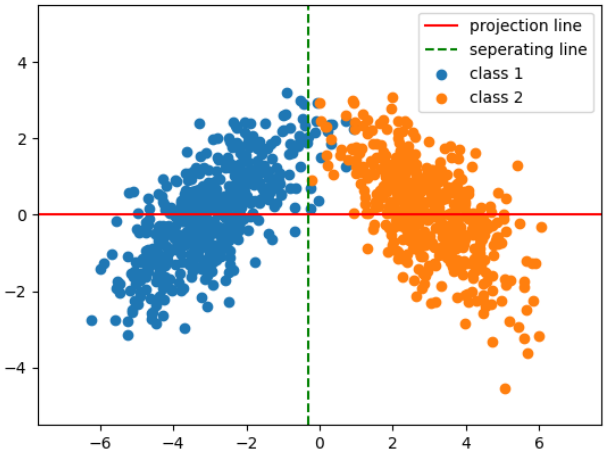

Figure 21: d

5.3 e

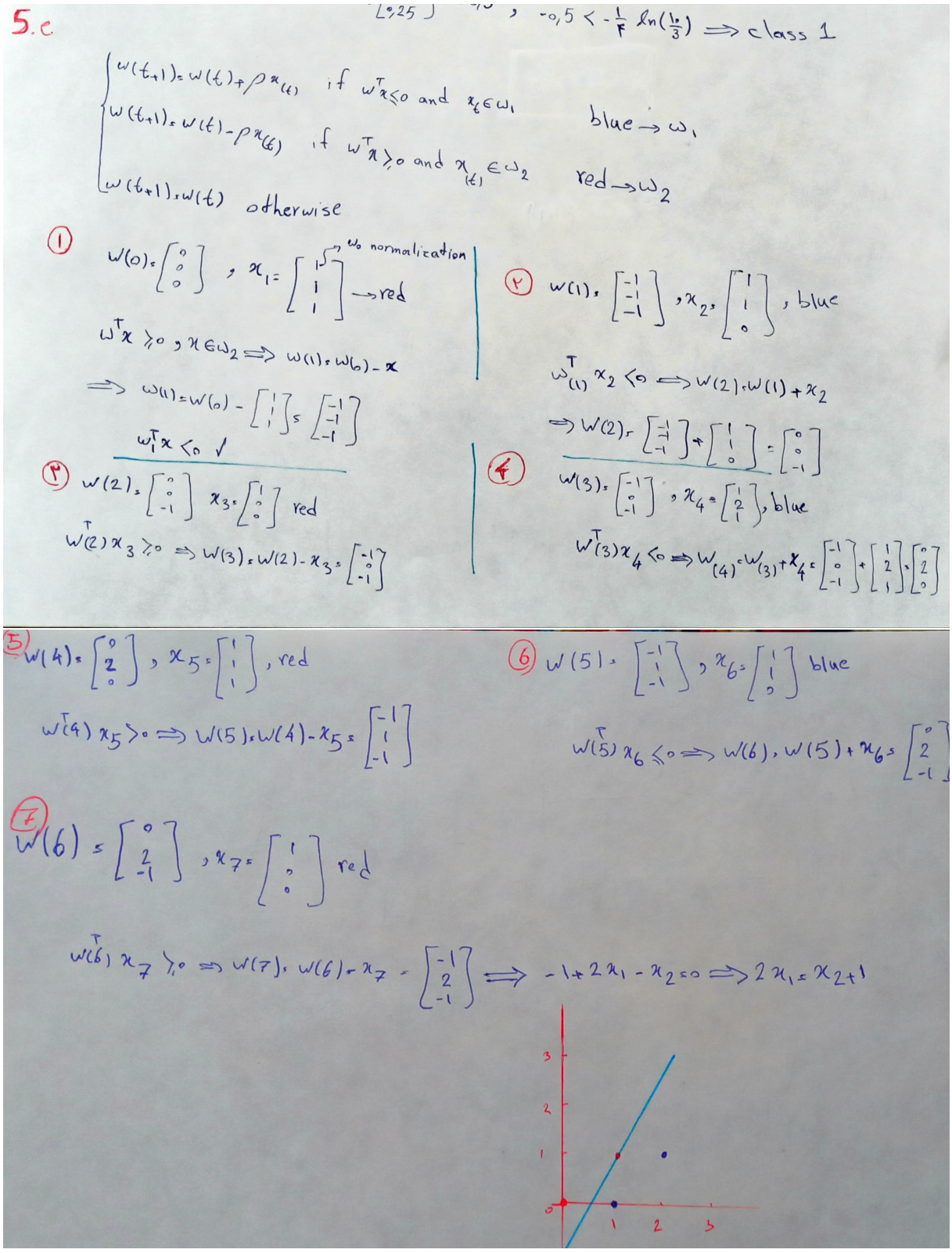

Figure 22: e

5.4 f-h

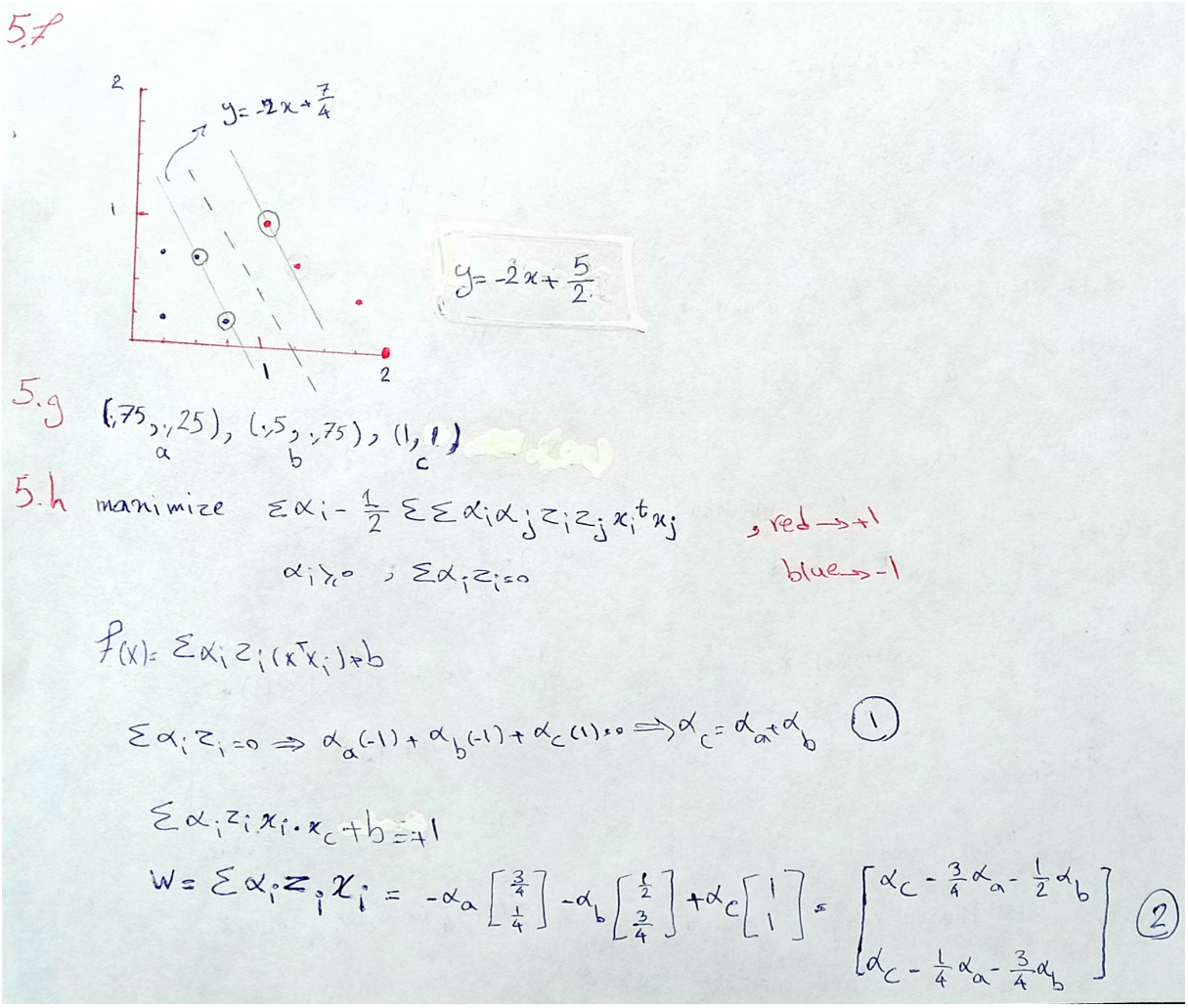

Figure 23: f-h

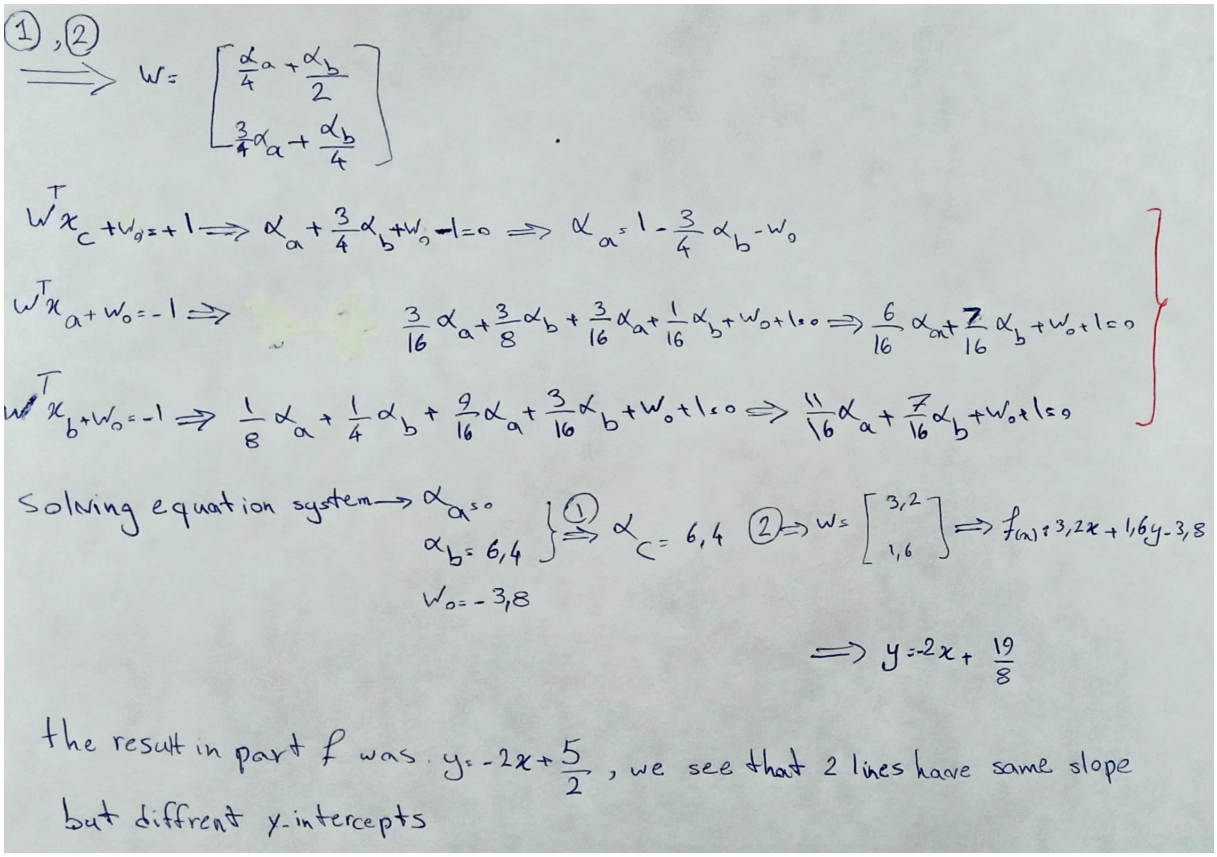

Figure 24: h

6 Categorizing Different Antarctic Penguin Species

6.1 a

data preprocessing:

```
#reading data
data = pd.read_csv('penguins.csv')
#selecting columns that we want
data = data [['species' , 'bill_length_mm','bill_depth_mm']]
#shuffling data
data = data.sample(frac=1).reset_index(drop=True)
#changing type of species column to category
data['species']= data['species'].astype('category')
#dropping null values
data = data.dropna()
#changing categorical variable to numerical
data['species']= data['species'].cat.codes
#Ade = 0   Gentoo = 2  Chin = 1
#splitting target and features
target = data['species']
features = data[['bill_length_mm','bill_depth_mm']]
#changing terget and faetures type to numpy array
target = target.to_numpy()
features = features.to_numpy()
#splitting train and test sets
y_train = target[:300]
y_test = target[300:]
x_train = features[:300]
x_test = features[300:]
#selecting 2 cetegories(Gentoo and adelie)
x_train_a = x_train[y_train != 1]
y_train_a = y_train[y_train != 1]
x_test_a = x_test[y_test != 1]
```

```
y_test_a = y_test[y_test != 1]
#normalization
#add a column of ones to features
x_train_a = np.append(np.ones((x_train_a.shape[0],1)) , x_train_a , axis = 1)
x_test_a = np.append(np.ones((x_test_a.shape[0],1)) , x_test_a , axis = 1)
#featuers of samples that are from class 2 * -1
x_train_a[y_train_a == 2] = x_train_a[y_train_a == 2]*-1
```

Listing 12: 6a

gradient decent and Newton's algorithm:

```
# b is vector of ones that se consider as target vector
b = np.ones((x_train_a.shape[0],1))
#gradian decent
#a is Weights matrix
a = np.zeros((3,1))
#MSE_GD is a list that store different values of MSE in differen iterations
MSE_GD = []
for i in range(400):
    MSE_GD.append(np.sum(np.power(x_train_a @ a - b , 2)))
    a = a - .000003 * (x_train_a.T @ (x_train_a @ a - b))

#newton
#calculating hessian matrix
#hessian matrix in this case is X^T * X and the result is like psudoinverse algorithm
H = x_train_a.T @ x_train_a
H_inv = np.linalg.inv(H)
#n is Weights matrix
n = np.zeros((3,1))
#MSE_N is a list that store different values of MSE in differen iterations
MSE_N = []
for i in range(5):
    MSE_N.append(np.sum(np.power(x_train_a @ n - b , 2)))
    n = n - H_inv @ (x_train_a.T @ (x_train_a @ n - b))
```

Listing 13: 6a

i used 0.000003 as learning rate for convergence because with learning rate = 0.1 gradient decent did not converge.
criterion function:

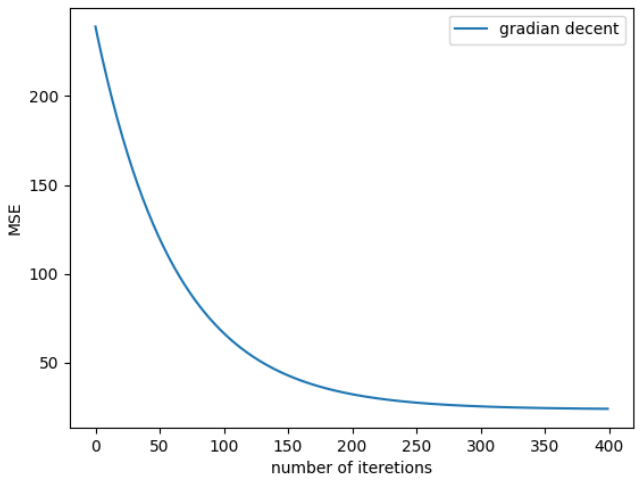

Figure 25: criterion function as function of the iteration number (gradient decent)

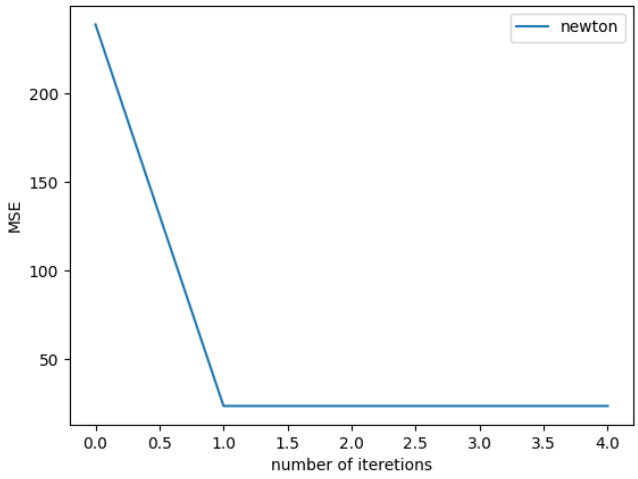

Figure 26: criterion function as function of the iteration number(newton's algorithm)

data distribution and decision boundaries:

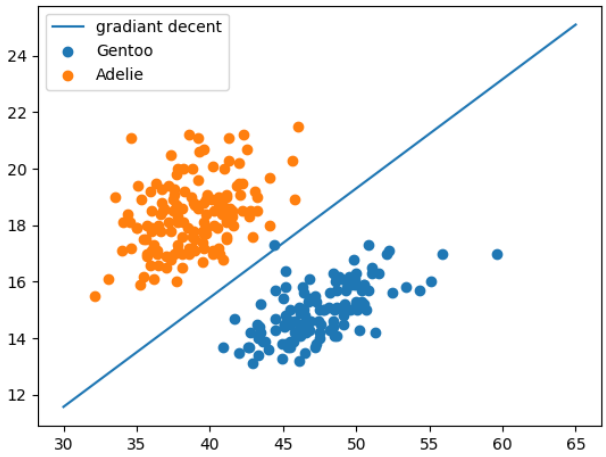

Figure 27: data distribution and decision boundaries(gradient decent)

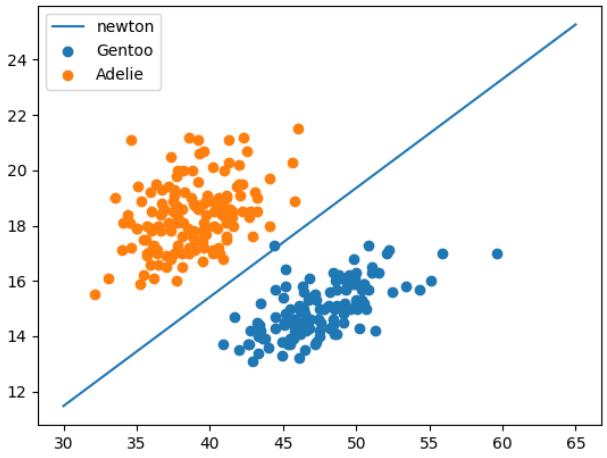

Figure 28: data distribution and decision boundaries (newton's algorithm)

accuracy of gradient: 1.0 , accuracy of newton: 1.0

6.2 b

in this section i normalized features between 0 and 1 for working with bigger learning rates.

```
b = np.ones((x_train_a.shape[0],1))
#gradian decent
#converge_time is a list that store differnt learning rates and their number of
    itration needed for convergence
converge_time = []
#we start with learning rate = 0.0001 and in each step we add 0.0001 to it
learning_rate = .0001
#first loop is for testing different learning rates
for j in range(80):
    a = np.zeros((3,1))
    #this loop is for executing gradient decent
    for i in range(2000):
        a = a - learning_rate * (x_train_a.T @ (x_train_a @ a - b))
        #condition for convergence
        if np.sum(np.power(x_train_a @ a - b , 2)) <27 :
            converge_time.append([learning_rate,i])
            break
    #checking if this learning rate didnt converge
    if np.sum(np.power(x_train_a @ a - b , 2)) > 27:
        min_fail = learning_rate
        break
    learning_rate+=.0001

#plotting learning rate vs number of iterations needed for convergence
print('minimum learning rate that fails to lead convergences' , min_fail)
plt.plot([x[0] for x in converge_time ] , [x[1] for  x in converge_time])
plt.xlabel('learning rate')
plt.ylabel('number of iteretions for convergence')
plt.savefig('6b.png')
```

Listing 14: 6b

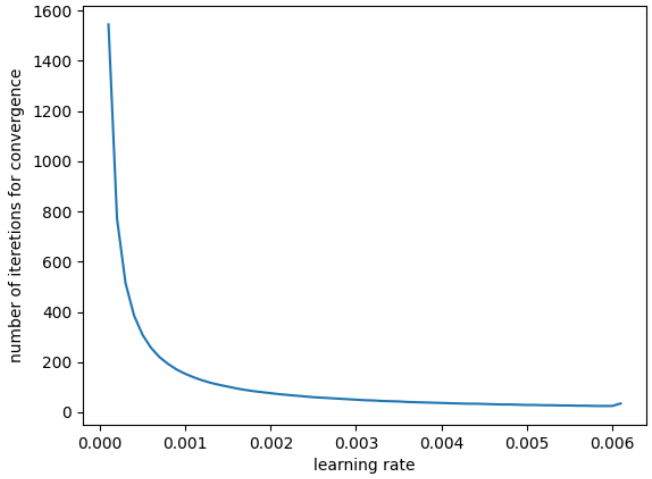

Figure 29: learning rate vs number of iterations needed for convergence

minimum learning rate that fails to lead convergences: 0.0062

6.3 c

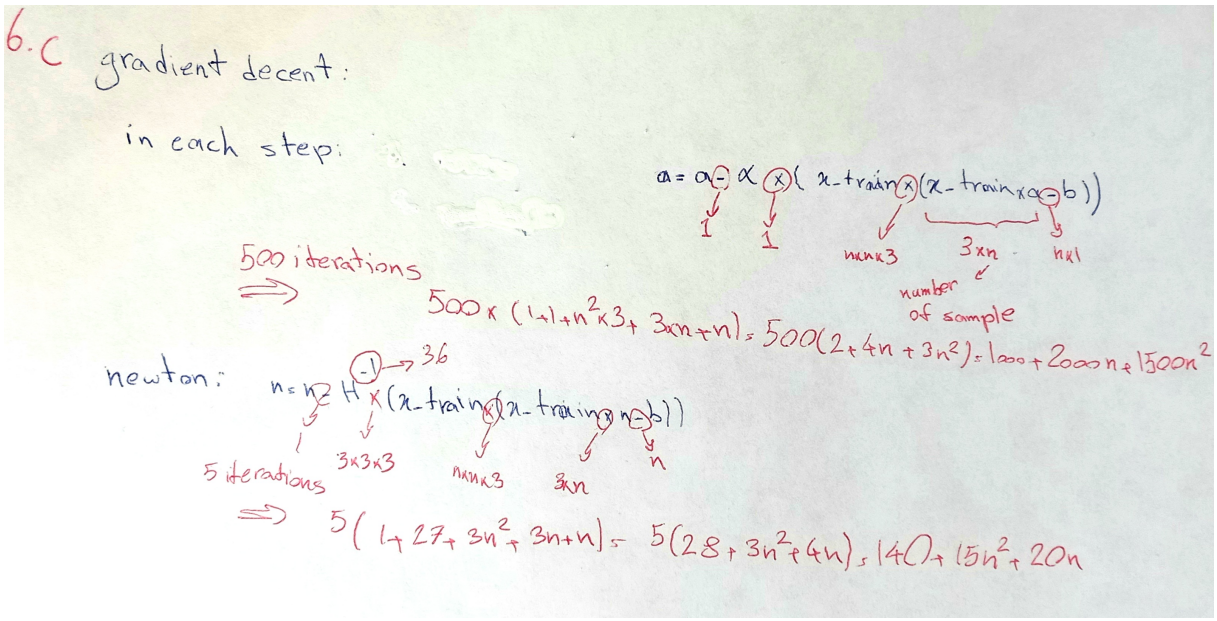

Figure 30: learning rate vs number of iterations needed for convergence

6.4 d

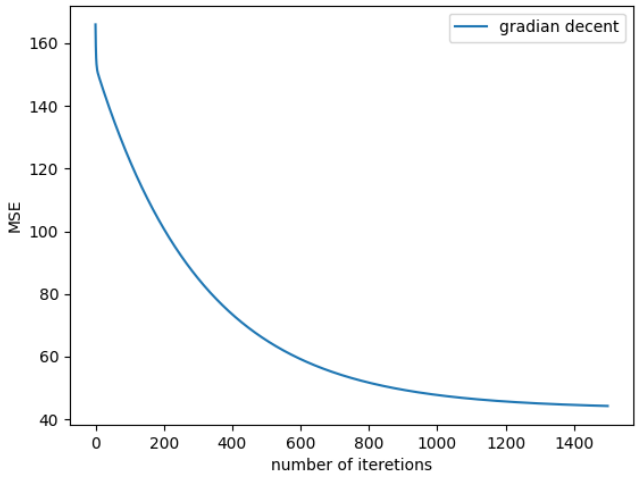

Figure 31: criterion function as function of the iteration number (gradient decent)

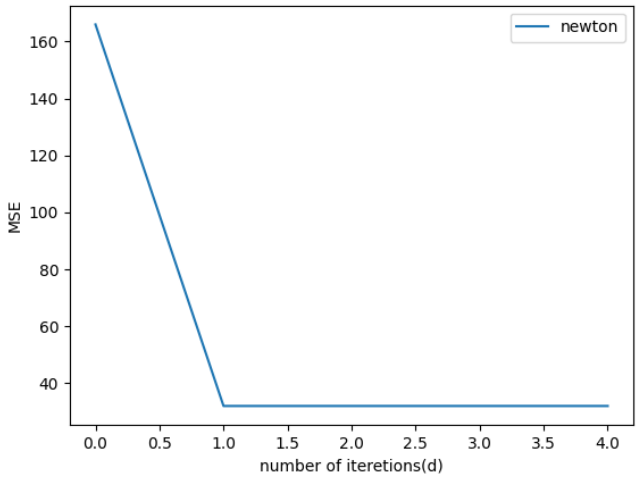

Figure 32: criterion function as function of the iteration number(newton's algorithm)

data distribution and decision boundaries:

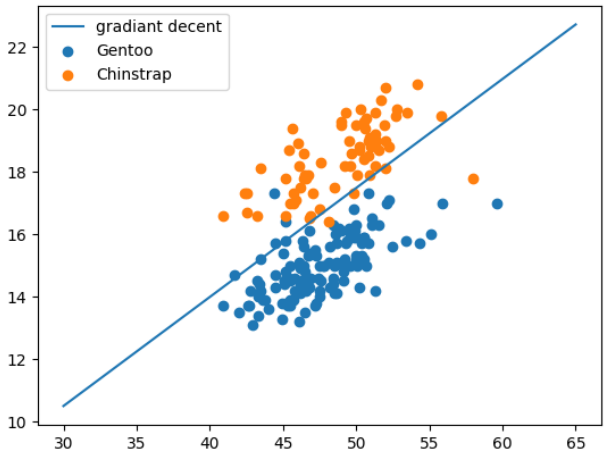

Figure 33: data distribution and decision boundaries(gradient decent)

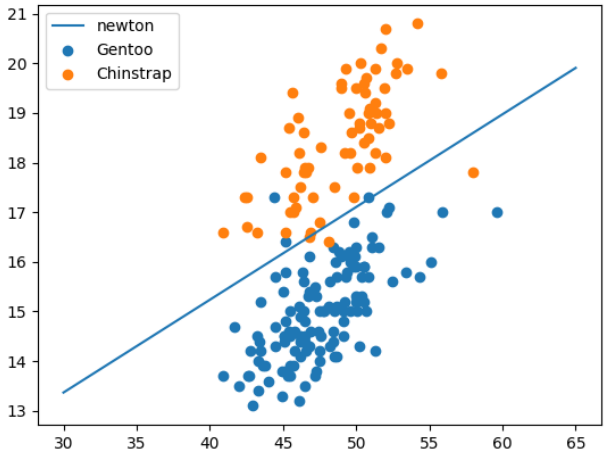

Figure 34: data distribution and decision boundaries (newton's algorithm)

accuracy of gradient: 0.96 , accuracy of newton: 0.96

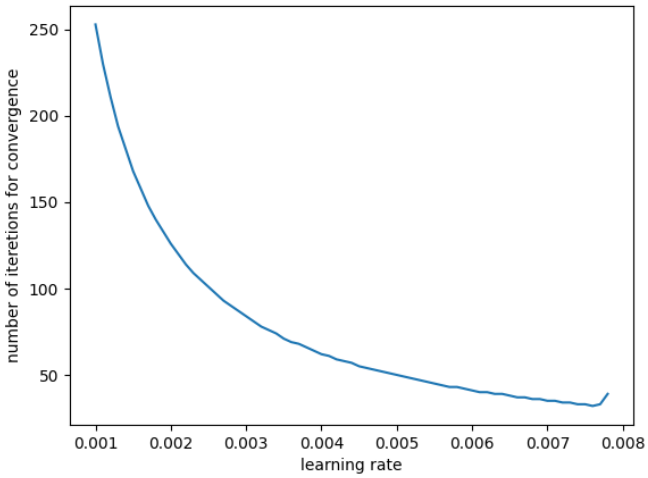

Figure 35: learning rate vs number of iterations needed for convergence

it took more iterations for gradient decent to converge because 2 classes are not linearly separable.

6.5 e

```
#a is Weights matrix
a = np.zeros((3,1))
#perceptron algorithm
for i in range(2500):
    temp =  x_train_a @ a
    #Y_m is misclassified samples with weights a
    Y_m = x_train_a[(temp <= 0).ravel() ]
    #updating weights
    a = a + .000004 * (np.sum(Y_m , axis = 0).reshape(3,1))
    #check for convergence
    if -(np.sum(temp[temp < 0]))<0.2 and i!=0:
        print('number of iterations needed for convergence: ' , i)
        break
```

Listing 15: 6e

number of iterations needed for convergence: 80 , accuracy of Perceptron: 1.0

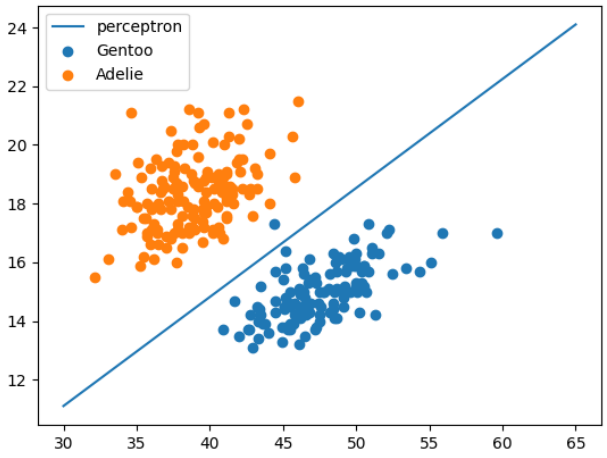

Figure 36: data distribution and decision boundary (perceptron algorithm)

6.6 f

accuracy of perceptron: 1.0 we can see it did not converge even with 1500 iterations

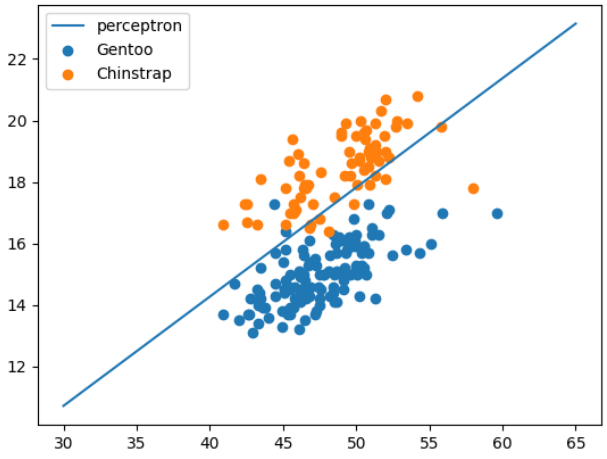

Figure 37: data distribution and decision boundary (perceptron algorithm)

6.7 g

in part f 2 classes are not linearly separable so perceptron can not converge even with 1500 iterations.

6.8 h

```
#a is Weights matrix
a = np.zeros((3,1))
#pocket weights
a_pocket = np.zeros((3,1))
misclassified_num = 10000
#perceptron algorithm
for i in range(500):
    temp =  x_train_a @ a
    #Y_m is misclassified samples with weights a
```

```
10      Y_m = x_train_a[(temp <= 0).ravel() ]
11      #updating weights
12      a = a + .000004 * (np.sum(Y_m , axis = 0).reshape(3,1))
13      temp = x_train_a @ a
14      #storing best weights
15      if len(np.nonzero(temp<=0)[0]) < misclassified_num:
16          misclassified_num = len(np.nonzero(temp<=0)[0])
17          a_pocket = a
```

Listing 16: pocket algorithm

pocket weights have changed in this iterations: 0, 13, 27, 78, 79, 80 ,86

6.9 i

pocket weights have changed in this iterations: 0, 1, 13, 84, 260

7 Some Explanatory Questions

7.1 a

The natural way for choosing the smoothing parameter is to plot out several curves and choose the estimate that is most in accordance with one's prior (subjective) ideas. but this method is not practical since we have usually highd imensional data. another method is Assume a standard density function and find the value of the bandwidth that minimizes the integral of the square error

7.2 b

yes consider that we have 2 linearly separable classes with similar distributions but different parameters then these 2 classifiers will have same decision boundaries.

7.3 c

No . consider this example:

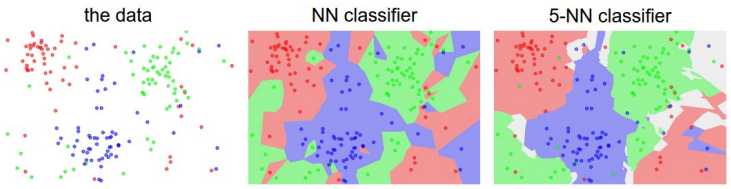

Figure 38: 7c

7.4 d

for greater values of K decision boundary is smoother.

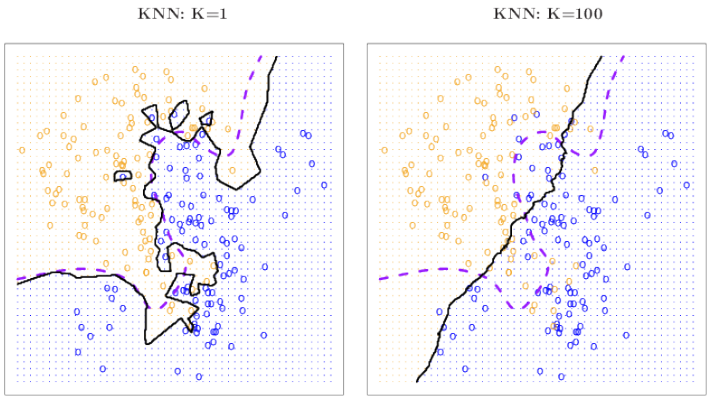

Figure 39: 7d

7.5 e

No let's assume that it's possible and all assigned labels are from class 1 and we have a point in training set that is from class 2. we can say that there is a point that it's nearest neighbor is this point from class 2 so our Hypothesis is not true.but we have this condition that we cannot have 2 points from 2 classes with the same position.

7.6 f

it's Sensitive to the scale of the data and gives more attention to the feature with greater values for this problem we can normalize features in the same range.

7.7 g

if multiple boundaries were required to separate the two classes, MLP could represent the better choice. This is because MLP, as opposed to SVM, can model XOR functions. SVMs are good for lots of features that are naturally on the same scale and fewer data points. If you have more than a few 10's of thousands of samples, SVMs take a really long time to train.

7.8 h

it has to be a DAG. because if not, for feedforward it can got stuck in a circle.