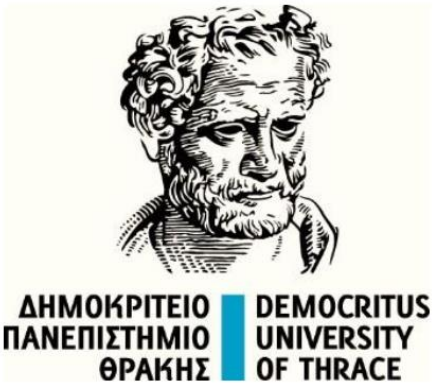


4Η ΕΡΓΑΣΙΑ

Στο μάθημα:
«Αναγνώριση Προτύπων»
Καθηγητής: Νικόλαος Μητιανούδης

Στυλιανός Μούσλεχ
ΑΜ:57382
25/11/2020 ,Ξάνθη

ΕΙΣΑΓΩΓΗ

Επειδή σε αυτή την εργασία υπάρχει αρκετός κώδικας, για ευκολότερη ανάγνωση/διόρθωση του κώδικα εισαγωγικά θα αναφέρω λίγο την δομή και την οργάνωση των αρχείων μου.

Στο αρχείο “Ex4_mFunctions.py” έχω γράψει όλες τις συναρτήσεις που χρησιμοποιώ για τις ασκήσεις 4.1 και 4.2.

Πιο συγκεκριμένα για την 4.1 :

def parzen_gauss_kernel(x,h,low,high)

def knn_density_estimate(x,knn,low,high,step)

Και για την 4.2:

def createOnevariablelassData(numberOfClasses,datasize,p,listofm,listofs)

def k_nn_classifier(numberOfClasses,trainDataSet,trainDataSetLabels,knn,testDataset)

def errorRateCalculation(classPredicted,actualClass):

def findBestKforTest(trainData,trainDataLabels,testData,testDataLabels,lowest_k, highest_k,maxTolerence)

def classifyFunc(d,x,m,s,pw)

def bayesianClassifier(trainData,trainDataLabels,testData,apriori)

def parzenWindowClassifier(trainData,trainDataLabels,testData,h,apriori):

def findBesthForParzen(trainData,trainDataLabels,testData,testDataLabels,p, lowest_h,highest_h,step,maxTolerence):

Η επεξήγηση για το τι κάνει η κάθε συνάρτηση γίνεται στην επεξήγηση της κάθε άσκησης παρακάτω.

Τέλος οι λύσεις και τα αποτελέσματα που βγάζουμε για την 4.1 και 4.2 υπάρχουν στα αρχεία “ Ex4_1.py” “ Ex4_2.py”

ΆΣΚΗΣΗ 4.1

Η σ.π.π που μας δίνεται είναι η : $p(x) = \begin{cases} \frac{1}{2} \ για \ 0 < x < 2 \\ 0 \ διαφορετικά \end{cases}$

Αυτή καταλαβαίνουμε ότι είναι από μια ομοιόμορφη κατανομή με άκρα 0,2 (uniform distribution).Οπότε θα πρέπει με την προσέγγιση της σ.π.π να προσεγγίζουμε γραφικά μια εικόνα ενός τετραγωνικού παλμού από 0 έως 2 με ύψος 0.5.

Α) Αρχικά θα δημιουργήσουμε τα δεδομένα με χρήση της εντολής της NumPy:

```
samples = np.random.uniform(0, 2, size=...)
```

Με μέγεθος κάθε φορά αυτό που θέλουμε.

Για τον υπολογισμό της προσέγγισης της σ.π.π έγραψα την συνάρτηση βασιζόμενη στην αντίστοιχη των διαλέξεων στο MATLAB με μερικές αλλαγές λόγω python

```
ef parzen_gauss_kernel(x,h,low,high):
  dims=x.shape
  if len(dims)==1:
    l=1
    n=dims[0]
  else:
    l=dims[0]
    n=dims[1]

  if (high-low)/h>1:
    px=np.zeros(math.floor((high-low)/h))
  else:
    px=np.zeros(math.ceil((high-low)/h))

  k=0
  for i in np.arange(low,high,h):
    for z in range(n):
      if l==1:
        xi=x[z]
      else:
        xi=x[:,z]
      c=i-xi
      dotProd=np.dot(c.T,c)
      px[k]=px[k] + math.exp(-dotProd/(2*h**2))
    px[k]=px[k]*(1/n)*(1/(((2*math.pi)**(l/2))*(h**l)))
    k=k+1

  return px
```

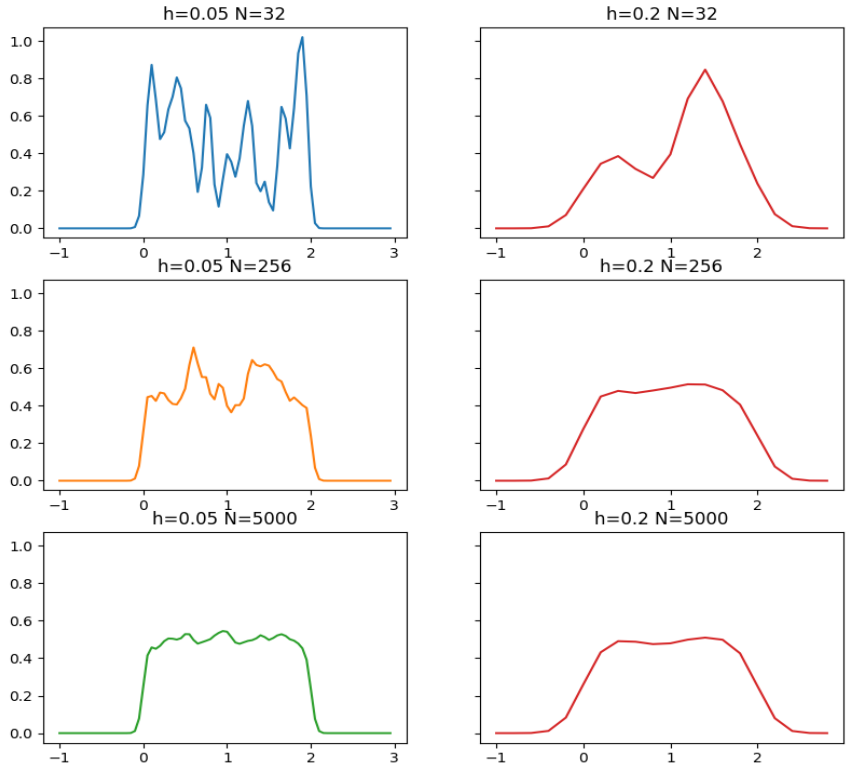
Αυτό που κάνει είναι για κάθε x μεταξύ του low και high με βήμα h να υπολογίζει το p(x) με βάση το παράθυρο parzen με τον εξής τρόπο (λόγω kernel gaussian N(0,1)

$$p(x) \approx \frac{1}{N}\sum_{i=1}^{N}\frac{1}{(2\pi)^{l/2}h^{l}}\exp\left(-\frac{(x-x_i)^T(x-x_i)}{2h^2}\right)$$

Οπότε για κάθε ζητούμενο h και N κάνουμε τον υπολογισμό και στη συνέχεια το plot με τη χρήση του παρακάτω κώδικα:

```
h = [0.05, 0.2]
sampleSize = [32, 256, 5000]
listofPlots=[]
for i in range(len(h)):
   for j in range(len(sampleSize)):
      samples = np.random.uniform(0, 2, size=sampleSize[j])
      px=ex4.parzen_gauss_kernel(samples,h[i],-1,3)
      x = np.arange(-1, 3, h[i])
      listofPlots.append((x,px))


fig, axs = plt.subplots(3,2,sharey=True)
axs[0, 0].plot(listofPlots[0][0], listofPlots[0][1])
axs[0, 0].set_title('h=0.05 N=32')
axs[1, 0].plot(listofPlots[1][0], listofPlots[1][1], 'tab:orange')
axs[1, 0].set_title('h=0.05 N=256')
axs[2, 0].plot(listofPlots[2][0], listofPlots[2][1], 'tab:green')
axs[2, 0].set_title('h=0.05 N=5000')
axs[0, 1].plot(listofPlots[3][0], listofPlots[3][1], 'tab:red')
axs[0, 1].set_title('h=0.2 N=32')
axs[1, 1].plot(listofPlots[4][0], listofPlots[4][1], 'tab:red')
axs[1, 1].set_title('h=0.2 N=256')
axs[2, 1].plot(listofPlots[5][0], listofPlots[5][1], 'tab:red')
axs[2, 1].set_title('h=0.2 N=5000')
```

και έχουμε το παρακάτω αποτέλεσμα:



Εδώ παρατηρούμε 2 πράγματα.

- Αρχικά όσο μεγαλύτερο Ν έχουμε τόσο καλύτερη προσέγγιση έχουμε, και τόσο λιγότερο φαίνεται να επηρεάζει η διαφορά στο h
- Επιπλέον όσο μικρότερο είναι το h τόσο περισσότερα “spikes” εμφανίζει η σ.π.π μας. (κάτι που είναι λογικό καθώς με μικρό h τόσο πιο σημαντικός γίνεται ο «θόρυβος» του κάθε σημείου από τα οποία βασιζόμαστε με αποτέλεσμα να έχουμε ένα είδος σαν overfitting)

Β)Όμοια έκανα μια συνάρτηση για την εκτίμηση της σ.π.π με τη χρήση k nearest neighbors βασιζόμενη σε αυτή των διαφανειών (σε MATLAB)

(Έκανα μια μικρή προέκταση στην συνάρτηση για να δουλεύει και για 2D και 3D δεδομένα με βάση τον τύπο για το V που είδαμε στη θεωρία αν και εδώ εχουμε 1D δεδομένα)

```
def knn_density_estimate(x,knn,low,high,step):
  dims=x.shape
  if len(dims)==1:
    l=1
    n=dims[0]
  elif len(dims)>3:
    print("this function currently works up to 3d data :) ")
    return
  else:
    l=dims[0]
    n=dims[1]
  px=np.zeros(math.floor((high-low)/step))
  euclidianDistances=np.zeros(n)
  k=0
  for i in np.arange(low, high, step):
    for z in range(n):
      if l == 1:
        xi = x[z]
        euclidianDistances[z] = np.sqrt(np.sum((i - xi) ** 2))
      else:
        xi = x[:,z]
        euclidianDistances[z]=np.np.sqrt(np.sum((i-xi).T.dot(x-xi)))

    sortedDistances=np.sort(euclidianDistances)
    r=sortedDistances[knn-1] #indexing starts at 0 so first neighbor is [0]
    if l==1:  #1D
      v=2*r
    elif l==2: #2D
      v=2*math.pi*r**2
    else:     #3D
      v=(4/3)*math.pi*r**2
    px[k]=knn/(n*v)
    k=k+1

  return px
```

Η συνάρτηση αυτή ακολουθεί τον εξής αλγόριθμο που είδαμε στις διαφάνειες με χρήση της ευκλείδειας απόστασης (στο τετράγωνο για υπολογιστικούς λόγους). Δηλαδή για κάθε x στο εύρος low:high με κάποιο step για καλή «ευκρίνεια» κάνουμε:

1. Βρίσκουμε την απόσταση του x από όλα τα δεδομένα (Εδώ ευκλείδεια)

2. Βρίσκουμε τα πλησιέστερα k δεδομένα στο x

4. Υπολογίζουμε τον όγκο V(x) που περικλείει τα k σημεία

5. Προσεγγίζουμε την pdf στο χ με $p(\mathbf{x}) \approx \frac{k}{NV(\mathbf{x})}$

Εάν έχουμε επιλέξει την Ευκλείδεια απόσταση στο d-διαστατο χώρο και η απόσταση από το μακρινότερο

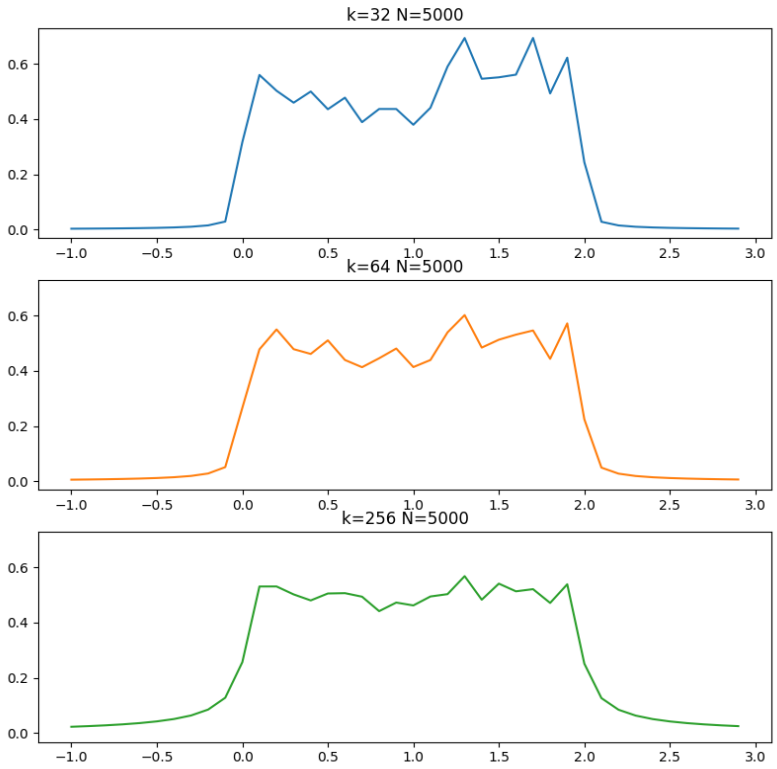
δείγμα είναι ρ τότε

V(x) = 2ρ για d=1, V(x) = πρ2 για d=2, V(x) = (4/3)πρ2 για d=3

Οπότε τρέχουμε το κώδικα για ζητούμενα k και κάνουμε plot το αποτέλεσμα:

```
k=[32,64,256]

listofPlots2=[]
for i in range(len(k)):
   px=ex4.knn_density_estimate(samples,k[i],-1,3,0.1)
   listofPlots2.append(px)

fig2, axs2 = plt.subplots(3,1,sharey=True)

x = np.arange(-1, 3, 0.1)
axs2[0].plot(x, listofPlots2[0])
axs2[0].set_title('k=32 N=5000')
axs2[1].plot(x, listofPlots2[1], 'tab:orange')
axs2[1].set_title('k=64 N=5000')
axs2[2].plot(x, listofPlots2[2], 'tab:green')
axs2[2].set_title('k=256 N=5000')

plt.show()
```

Και έχουμε:



Παρατηρούμε ότι όσο αυξάνουμε το k τόσο καλύτερη προσέγγιση έχουμε.

ΆΣΚΗΣΗ 4.2

Α) Αρχικά έφτιαξα μια παραμετροποιήσιμη συνάρτηση (στη λογική αυτής που είχα κάνει στην προηγούμενη άσκηση 3) για να δημιουργούμε δεδομένα μιας μεταβλητής που ακολουθούν την κανονική κατανομή με όποιον αριθμό κλάσεων θέλουμε δίνοντας τα μ τα σ και τις a a priori πιθανότητες και τον αριθμό των κλάσεων που θέλουμε. Η συνάρτηση επιστρέφει ένα NumPy array με όλα τα δεδομένα και ένα δεύτερο NumPy array με όλα τα labels για τα δεδομένα

```
def createOnevariablelassData(numberOfClasses,datasize,p,listofm,listofs):
    listOfData=[]
    labellist=[]
    numberOfSamplesCreated=0 #check how many we have created so last class rounds to
datasize!!!!
    for i in range(numberOfClasses-1):
        classSize=math.floor(datasize*p[i])
        listOfData.append(np.random.normal(listofm[i],listofs[i],size=classSize))
        numberOfSamplesCreated=numberOfSamplesCreated+classSize
        labellist.append(np.ones(classSize)*i)
    #for last class we need to take the remaining samples cause for sum of p we might not have
round number of data
    lastclassSize=datasize-numberOfSamplesCreated
    listOfData.append(np.random.normal(listofm[-1],listofs[-1],size=lastclassSize))
    labellist.append(np.ones(lastclassSize) * (i+1))

    dataVector=np.concatenate(listOfData)
    labelVector=np.concatenate(labellist)


    return dataVector,labelVector
```

Οπότε δημιουργούμε τα train data και test data με βάση την εκφώνηση:

```
listofm=[2,1,3]
listofstdDeviation=[math.sqrt(0.5),math.sqrt(0.1),math.sqrt(1.3)]

p=[0.5,0.3,0.2]

trainDataSize=100
testDataSize=1000

trainData,trainDataLabels=ex4.createOnevariablelassData(3,trainDataSize,p,listofm,listofstdDevi
ation)
testData,testDataLabels=ex4.createOnevariablelassData(3,testDataSize,p,listofm,listofstdDeviatio
n)
```

Β) Εδώ έγραψα μια συνάρτηση που υλοποιεί τον αλγόριθμο του knn classifier:

```
def k_nn_classifier(numberOfClasses,trainDataSet,trainDataSetLabels,knn,testDataset):
    dims_trainData=trainDataSet.shape
    dims_testData=testDataset.shape

    if len(dims_testData)!=len(dims_trainData):
        print("Train and test data variables must have the same number of dimensions! ")
        return
    else:
        if len(dims_trainData)==1: # we can also use testData dims because now we know they are
equal
            l=1
            n_train=dims_trainData[0]
            n_test=dims_testData[0]
        else:
            l = dims_trainData[0]  #l is the same we can use train or test to get it
            n_train=dims_trainData[1]
            n_test =dims_testData[1]

    testDataLabels=np.zeros(n_test)

    for i in range(n_test):
        #euclidian squared here
        if len(dims_trainData)>1: #in casee our data are not 1D
            distances=np.sum(np.power(testDataset[:,i] -trainDataSet, 2),axis=0)
        else:#in case data is 1D we get absolut distance as euclidian distance
            distances=np.abs(testDataset[i] -trainDataSet)
        pointsNearest=np.argsort(distances)
        labelCount=np.zeros(numberOfClasses)
        for j in range (knn):
            classNo=int(trainDataSetLabels[pointsNearest[j]])
            labelCount[classNo]=labelCount[classNo]+1  ##we cound how many times each class is a
neighbor (histogram like)

        testDataLabels[i]=np.argmax(labelCount)
    return testDataLabels
```

Δέχεται τα train data , τα labels τους καθώς και τα test data και τον αριθμό k και επιστρέφει τα labels για τα test data με βάση τον εξής αλγόριθμο

Για κάθε test data προσδιορίζω τους κ κοντινότερους γείτονες του από το train data (εδώ με χρήση ευκλείδειας απόστασης) . Στη συνέχεια ταξινομούμε το σημείο αυτό στην κλάση όπου ανήκει η πλειοψηφία των κ αυτών γειτόνων και τέλος επιστρέφω όλα τα labels αυτά.

Τώρα για τον υπολογισμό του error έκανα μια απλή συνάρτηση όπου βάζουμε τα labels των data που έχουμε πραγματικά καθώς και τα labels που κάνουμε predict με τον κάθε classifier κάθε φορά και υπολογίζει το error rate ως το άθροισμα των λάθος ταξινομήσεων προς τα συνολικά δεδομένα που έχουμε. Ο κώδικας:

```
def errorRateCalculation(classPredicted,actualClass):
    if len(classPredicted)!= len(actualClass):
        print("the vectors with the  class labels are not same size  fix!!")
        return
    else:
        numberOfData=len(actualClass) #same with class predicted!
    wrong_classifications=0
    for i in range(numberOfData):
        if classPredicted[i] != actualClass[i]:
            wrong_classifications=wrong_classifications+1

    return wrong_classifications/numberOfData
```

Με βάση αυτούς τους κώδικες τρέχουμε για τα ζητούμενα κ και έχουμε τα εξής αποτελέσματα:

```
print("====KNN CLASSIFIER====")
for k in range(1,4):
    knnLabels=ex4.k_nn_classifier(3,trainData,trainDataLabels,k,testData)
    errorRate=ex4.errorRateCalculation(knnLabels,testDataLabels)
    print("K=",k, "KNN Error Percentage: ",errorRate)

print("Trying to find the best K...")
```

```
====KNN CLASSIFIER====
K= 1 KNN Error Percentage:  0.402
K= 2 KNN Error Percentage:  0.378
K= 3 KNN Error Percentage:  0.359
```

Παρατηρώ ότι όσο αυξάνω το Κ έχω καλύτερη ακρίβεια δηλαδή μειώνεται ο αριθμός τον σφαλμάτων μου, πράγμα λογικό καθώς με μεγαλύτερο κ βλέπω για περισσότερους γείτονες τι συμβαίνει και έτσι έχω καλύτερη εικόνα. Επίσης να σημειωθεί ότι το error είχε αρκετές διακυμάνσεις με αλλαγές του seed.

Τώρα θα συγκρίνουμε με έναν bayes classifier. Επειδή έχουμε αναλύσει και έχουμε κάνει τον bayes classifier σε προηγούμενη εργασία απλά συνοπτικά θα αναφέρω ότι στον κώδικα τον υλοποιώ με τις παρακάτω συναρτήσεις που στο τέλος παίρνω έναν πίνακα με τα predicted labels:

```
def classifyFunc(d,x,m,s,pw):
    c = x - m  # not necesseary but to have cleaner code in g

    #np.det() and np.linalg.inv()  input must be at least 2D array so we must check first for
dimension of our distribution
    if d>1: #that means we have at least 2D so S is a 2D matrix
        detS=round(abs(np.linalg.det(s)),4) #roundind is needed cause linalg.det sometimes returns
floating points with 1 bit rounding error
        invS=np.linalg.inv(s)
        g = -0.5 * (c.T).dot(invS).dot(c) - (d / 2) * math.log(2 * math.pi) - 0.5 * math.log(detS) +
math.log(pw)
    else:
        detS=abs(s)
        invS=s**-1
```

```
    g = -0.5 * c**2*invS- (d / 2) * math.log(2 * math.pi) - 0.5 * math.log(detS) + math.log(pw)
  return g


def bayesianClassifier(trainData,trainDataLabels,testData,apriori):
  numberOfClasses=len(apriori)
  listOfSplitData=[]
  #Calculate M and S for every class
  startingIndex=0
  listofmeans=[]
  listofS=[]
  for i in range (numberOfClasses):
    numberOfElementsInClass=np.count_nonzero(trainDataLabels==i)

listofmeans.append(np.mean(trainData[startingIndex:startingIndex+numberOfElementsInClass]
))
    listofS.append(np.cov(trainData[startingIndex:startingIndex+numberOfElementsInClass]))
    startingIndex=startingIndex+numberOfElementsInClass

  testDataLabels=np.zeros(len(testData))
  for j in range(len(testData)):
    biggest_g=-100000
    classOfDataPoint=-1
    for classNo in range(numberOfClasses):
      g=classifyFunc(1,testData[j],listofmeans[classNo],listofS[classNo],apriori[classNo])
      if g>biggest_g:
        biggest_g=g
        classOfDataPoint=classNo
    testDataLabels[j]=classOfDataPoint
  return testDataLabels
```

Οπότε καλούμε τη συνάρτηση αυτή και βλέπουμε το error:

```
print("====BAYESIAN CLASSIFIER====")

bayesianPredicted=ex4.bayesianClassifier(trainData,trainDataLabels,testData,p)
bayesianError=ex4.errorRateCalculation(bayesianPredicted,testDataLabels)
print("Bayesian Classifier Error Rate  is: ",bayesianError)
```

```
====BAYESIAN CLASSIFIER====
Bayesian Classifier Error Rate  is:  0.277
```

Εδώ παρατηρούμε ότι έχουμε λίγο καλύτερα αποτελέσματα από τον knn και μάλιστα δοκιμάζοντας τον για διαφορετικά seeds το αποτέλεσμα φάνηκε αρκετά σταθερό και κοντά σε αυτό το νούμερο

Προεραιτικό ερώτημα: Αναφορικά με την εύρεση του βέλτιστου κ δεν υπάρχει το ιδανικό κ που προκύπτει για όλες τις περιπτώσεις. Ένας καλός κανόνας είναι να διαλέγουμε ως κ την ρίζα του μεγέθους του train data, στην δικιά μας περίπτωση k=10. Το αποτέλεσμα είναι το εξής:

```
print("Trying a rule of thumb of k=sqrt(numberofTrainData)=10")
knnLabels = ex4.k_nn_classifier(3, trainData, trainDataLabels, 10, testData)
errorRate = ex4.errorRateCalculation(knnLabels, testDataLabels)
print("Error Rate for K=10 is: ",errorRate)
```

```
Trying a rule of thumb of k=sqrt(numberofTrainData)=10
Error Rate for K=10 is:  0.3
```

Στην συνέχεια αποφάσισα να προσπαθήσω να το βρω πιο δοκιμαστικά με μια πιο “brute force” μέθοδο το βέλτιστο k για αυτά τα δεδομένα κάτι που είναι αρκετά εφικτό στην περίπτωση μας που έχουμε μόνο 1 χαρακτηριστικό .

Οπότε έγραψα μια συνάρτηση που δίνοντας ότι χρειάζεται ο knn classifier καθώς και το ελάχιστο k να δοκιμάσουμε, το μέγιστο k που μπορεί να θέλουμε να δοκιμάσουμε καθώς και τον μέγιστο αριθμό δοκιμών που επιτρέπουμε να γίνουν χωρίς να έχουμε βελτίωση στο error rate μας (ένα είδος tolerance). Η συνάρτηση αυτή δοκιμάζει για όλα τα k που παραμετροποιήσαμε παραπάνω να βρεί το ελάχιστο test error rate και να το επιστρέψει καθώς και το κ για το οποίο έχουμε αυτό. Ο κώδικας:

```
def
findBestKforTest(trainData,trainDataLabels,testData,testDataLabels,lowest_k,highest_k,maxTolerence):

    counter=0
    bestK=-1
    bestErrorRate=100
    for k in range(lowest_k,highest_k+1):
        knnLabels = k_nn_classifier(3, trainData, trainDataLabels, k, testData)
        errorRate = errorRateCalculation(knnLabels, testDataLabels)
        if errorRate < bestErrorRate:
            bestErrorRate=errorRate
            bestK=k
            counter=0
        elif counter>maxTolerence:
            return bestK,bestErrorRate
        else:
            counter=counter+1
    return bestK,bestErrorRate
```

Τρέχοντας τον παρακάτω αλγόριθμο με τον παρακάτω τρόπο:

```
print("Using my 'brute force' method...")
k_best,error_best=ex4.findBestKforTest(trainData,trainDataLabels,testData,testDataLabels,1,100,20)

print("Best K is: ",k_best," With Error Percentage: ",error_best)
```

έχουμε τα εξής αποτελέσματα:

```
Using my 'brute force' method...
Best K is:  13  With Error Percentage:  0.27
```

Γενικότερα βέβαια εδώ να βρούμε και το k για το οποίο το test data μας θα έχει το ελάχιστο error δεν σημαίνει ότι έχουμε γενικά την καλύτερη λύση, καθώς αν είχαμε λίγο διαφορετικό test data μπορεί να μην βρήκαμε το κ που κάνει το καλύτερο generalization. Οπότε ίσως καλύτερη πρακτική θα ήταν να σπάσουμε το dataset που κάνουμε και σε ένα validation και να κάνουμε cross validation για να βρούμε ένα καλύτερο k.

Γ. Για την ταξινόμηση μέσω Parzen Windows αυτό που πρέπει να κάνουμε είναι για κάθε σημείο test να εκτιμήσουμε με τη χρήση Parzen windows την πιθανοφάνεια του σε κάθε κλάση δηλαδή τα:

$$P(x|\omega_1), P(x|\omega_2), P(x|\omega_3)$$

Και στη συνέχεια να υπολογίσουμε την posterior πιθανότητα για κάθε κλάση πολλαπλασιάζοντας το παραπάνω με την a priori για κάθε κλάση έτσι θα βάλουμε ένα δείγμα σε κάποια κλάση με βάση τον κανόνα απόφασης bayes

Δηλαδή πχ στην κλάση i αν :

$$P(\omega_i|x) > P(\omega_j|x), \quad \forall j \neq i$$

Δηλαδή: $p(x|\omega_i)P(\omega_i) > p(x|\omega_j)P(\omega_j), \quad \forall j \neq i$

Αυτή η διαδικασία γίνεται στην παρακάτω συνάρτηση χρησιμοποιώντας και την συνάρτηση για τον υπολογισμό του parzen window που γράψαμε παραπάνω:

```
def parzenWindowClassifier(trainData,trainDataLabels,testData,h,apriori):

    numberOfClasses = len(apriori) # see how many classes we hve
    #split data of each train class
    startingIndex=0
    listofClassData=[]
    for i in range(numberOfClasses):
        numberOfElementsInClass = np.count_nonzero(trainDataLabels == i)
        dataofClass=trainData[startingIndex:startingIndex+numberOfElementsInClass]
        listofClassData.append(dataofClass)

        startingIndex=startingIndex+numberOfElementsInClass

    #now for each testData point we need to calculate likelihood P(wi/x) using Parzen Window
    #then we classify it using baysian rule P(wi)*P(wi/x) > P(wj)*P(wj/x)

    predicted_class=np.zeros(len(testData))
    for j in range(len(testData)):
        #we use our previous functtion to estimate this likelihood with a little trick to do it only at
our test point
        classOfPoint =-1
        bestPosterior = -10000
        for classNo in range(numberOfClasses):
            likelihood_ofDataPoint =
```

```
parzen_gauss_kernel(listofClassData[classNo],h,testData[j],testData[j]+(h/2)) #IMPORTANT!
with low our data point and high our datapoint+h/2 it will only estimate it for our datapoint
            posterior=likelihood_ofDataPoint * apriori[classNo]
            if posterior>bestPosterior:
                classOfPoint=classNo
                bestPosterior=posterior
        predicted_class[j]=classOfPoint
    return predicted_class
```

και την καλούμε για 4 τιμές του h όπως ζητήθηκε στην εκφώνηση :

```
print("====PARZEN WINDOW CLASSIFIER====")
for h in [0.1,0.3,0.5,0.7]:
    parzenPredictions=ex4.parzenWindowClassifier(trainData,trainDataLabels,testData,h,p)
    parzenError=ex4.errorRateCalculation(parzenPredictions,testDataLabels)
    print("h=",h, "Parzen Error Percentage: ",parzenError)
```

και έχουμε τα παρακάτω αποτελέσματα :

```
====PARZEN WINDOW CLASSIFIER====
h= 0.1 Parzen Error Percentage:  0.281
h= 0.3 Parzen Error Percentage:  0.275
h= 0.5 Parzen Error Percentage:  0.272
h= 0.7 Parzen Error Percentage:  0.268
```

Εδώ παρατηρούμε ότι τουλάχιστον για τις συγκεκριμένες τιμές μεγαλώνοντας το h έχω καλύτερο αποτέλεσμα αν και είναι αρκετά κοντά . Βέβαια και σε αυτή την μέθοδο η αλλαγή του seed των δεδομένων επηρέαζε το error rate. Φαίνεται να εμφανίζει λίγο καλύτερα αποτελέσματα από τον knn.

<u>Προεραιτικό ερώτημα:</u>

Στην ίδια λογική με την εύρεση του βέλτιστου k παραπάνω έκανα μια συνάρτηση που δοκιμάζει για ένα μεγάλος εύρος h να υπολογίσει το error κρατάει το καλύτερο και σταματάει αν δεν δει βελτίωση για συγκεκριμένο βαθμό δοκιμών. Ο κώδικας που το υλοποιεί:

```
def
findBesthForParzen(trainData,trainDataLabels,testData,testDataLabels,p,lowest_h,highest_h,step
,maxTolerence):
    counter=0
    best_h=-1
    bestErrorRate=100
    for h in np.arange(lowest_h,highest_h+step,step):
        parzenPredictions = parzenWindowClassifier(trainData, trainDataLabels, testData, h, p)
        parzenError = errorRateCalculation(parzenPredictions, testDataLabels)
        if parzenError < bestErrorRate:
            bestErrorRate=parzenError
            best_h=h
            counter=0
        elif counter>maxTolerence:
```

```
            return best_h,bestErrorRate
        else:
            counter=counter+1
    return best_h,bestErrorRate
```

τρέχοντας τον κώδικα με τις παρακάτω παραμέτρους:

```
print("Using my 'brute force' method...")
h_best,error_best=ex4.findBesthForParzen(trainData,trainDataLabels,testData,testDataLabels,p,0
.1,7,0.1,10)
print("Best h is: ",h_best," With Error Percentage: ",error_best)
```

έχουμε το εξής αποτέλεσμα:

```
Using my 'brute force' method...
Best h is:  0.6  With Error Percentage:  0.263
```

Δ. Για την υλοποίηση των PNN στη python κατά αντιστοιχία έγινε η χρήση της βιβλιοθήκης NeuralPy η οποία είναι μια βιβλιοθήκη βασισμένη στην TensorFlow και επιτρέπει την εύκολη υλοποίηση δικτύων PNN με αρκετά παρόμοιο τρόπο με το Toolbox του MATLAB. Οπότε με τον παρακάτω κώδικα κάνουμε το classification μέσω του PNN:

```
from neupy import algorithms as algo

print("====PNN CLASSIFIER====")

for hspread in [0.1,0.3,0.5,0.7]:
    pnn = algo.PNN(std=hspread, verbose=False)
    pnn.train(trainData,trainDataLabels)
    pnnPredicted=pnn.predict(testData)
    pnnError = ex4.errorRateCalculation(pnnPredicted, testDataLabels)
    print("hspread=",hspread, "PNN Error Percentage: ",pnnError)
```

κκαι έχουμε τα εξής αποτελέσματα:

```
hspread= 0.1 PNN Error Percentage:  0.31
hspread= 0.3 PNN Error Percentage:  0.305
hspread= 0.5 PNN Error Percentage:  0.309
hspread= 0.7 PNN Error Percentage:  0.311
```

Εδώ το καλύτερο h φαίνεται να είναι κοντά στο 0.3-0.5. Βέβαια υπήρχαν αρκετές διακυμάνσεις και εδώ ανάλογα το seed.

Κλείνοντας ας δούμε συγκεντρωτικά τα αποτελέσματα:

```
====KNN CLASSIFIER====
K= 1 KNN Error Percentage:  0.402
K= 2 KNN Error Percentage:  0.378
K= 3 KNN Error Percentage:  0.359
Trying to find the best K...
Trying a rule of thumb of k=sqrt(numberofTrainData)=10
Error Rate for K=10 is:  0.3
Using my 'brute force' method...
Best K is:  13  With Error Percentage:  0.27
====BAYESIAN CLASSIFIER====
Bayesian Classifier Error Rate  is:  0.277
====PARZEN WINDOW CLASSIFIER====
h= 0.1 Parzen Error Percentage:  0.281
h= 0.3 Parzen Error Percentage:  0.275
h= 0.5 Parzen Error Percentage:  0.272
h= 0.7 Parzen Error Percentage:  0.268
Using my 'brute force' method...
Best h is:  0.6  With Error Percentage:  0.263
====PNN CLASSIFIER====
hspread= 0.1 PNN Error Percentage:  0.31
hspread= 0.3 PNN Error Percentage:  0.305
hspread= 0.5 PNN Error Percentage:  0.309
hspread= 0.7 PNN Error Percentage:  0.311
```

Αναφορικά με τα αποτελέσματα που βλέπουμε, αρχικά όλα τα αποτελέσματα ήταν αρκετά κοντά και ποια χειροτέρευαν ή βελτιωνόντουσαν εξαρτιόταν αρκετά από το seed των δεδομένων . Ο μόνος classifier που φάνηκε αρκετά robust σε αυτές τις αλλαγές και ήταν πάντα κοντά στο 28% error rate ήταν ο Bayes Classifier.