Non-parametric Classification with Dimensionality Reduction

Kevin De Angeli
kevindeangeli@utk.edu
COSC 522 - Machine Learning
University of Tennessee, Knoxville

October 10, 2019

Abstract

This paper explores the application of K-nearest neighbors (Knn) for classification tasks on a data set that describes diabetes in prime Indian heritage living near Phoenix Arizona. I used Principal Component Analysis (PCA) and Fisher's Linear Discriminant (FLD) to reduce the dimensions of the data set. I have developed a extensive analysis of the performance of Knn with the three different data sets obtained that include accuracy, confusion matrices and running time. Finally, I compared the performance of Knn with three Bayesian classifier under different assumptions.

1 Introduction

K_n Nearest Neighbors (kNN) is one of the core methods for non-parametric classification tasks [1]. This method makes no assumptions about the nature of a population, and makes decisions by merely comparing test points with the training data. This characteristic makes kNN computationally demanding, because each testing point is compared with the entire training data set. The kNN method has been successfully applied to problems in diverse domains. For example, in [2] Sadegh Bafandeh applied kNN to predict economic events, text categorization [3], [4], and visual recognition [5]. In this project, I developed a kNN algorithm from the foundations (using just numpy) and apply it to a data set that contains information about diabetes in prima indians. The algorithms and methods presented in this paper can be easily applied to other data set. Additionally, I applied previously developed Gaussian classifiers and compare the performance. The Python scripts created for this project are provided in the Appendix.

2 Methods

2.1 Data set

The Diabetes Prima Indians data set comes from [1]. It contains data from a population of women who were 21 years old and older. They were women of Prima Indian heritage living in Phoenix, Arizona. Each woman in the data set was tested for diabetes according to World Health Organization criteria. The actual data was collected by the US National Institute of Diabetes and Digestive and Kidney Diseases. The training data set contains 200 rows and the test set has 332 rows. Both have 8 columns, where the last column contains "yes" or "no" values which tells if the person has diabetes. These values were replaced by 0s and 1s at the beginning of the program. 1 shows the specific information contained in the data set.

2.2 Data Pre-processing

2.2.1 Normalization

The first step was to normalize the data so that the values in the data set are scaled similarly. I normalized the data by simply subtracting the mean and dividing by the standard deviation of the specific class:

$$x_i = \frac{x_i - \mu_i}{\sigma_i}$$

I used the normalized data as a data set on its own, and I called it nX.

Column Name	Description
npreg	number of pregnancies
glu	plasma glucose concentration in an oral glucose tolerance test
bp	diastolic blood pressure (mm Hg)
skin	triceps skin fold thickness (mm)
ins	serum insulin (micro U/ml)
bmi	body mass index (weight in $kg/(heightinm)^2$)
ped	diabetes pedigree function
age	in years
type	No / Yes

Table 1

2.2.2 Principal Component Analysis (PCA)

PCA is a unsupervised method for data representation that allows us to reduce the complexity of the data. In order to find the principal components of the data set, I first computed the $7x7$ Variance-Covariance matrix of the training data set and the corresponding eigenvalues and eigenvectors:

$$\Sigma = \begin{bmatrix} 1.005025 & 0.171382 & 0.253328 & 0.109597 & 0.058629 & -0.120073 & 0.601932 \\ 0.171382 & 1.005025 & 0.270735 & 0.21869 & 0.217879 & 0.061015 & 0.345133 \\ 0.253328 & 0.270735 & 1.005025 & 0.266295 & 0.240021 & -0.047638 & 0.393039 \\ 0.109597 & 0.21869 & 0.266295 & 1.005025 & 0.662347 & 0.095882 & 0.253192 \\ 0.058629 & 0.217879 & 0.240021 & 0.662347 & 1.005025 & 0.191508 & 0.132582 \\ -0.120073 & 0.061015 & -0.047638 & 0.095882 & 0.191508 & 1.005025 & -0.071768 \\ 0.601932 & 0.345133 & 0.393039 & 0.253192 & 0.132582 & -0.071768 & 1.005025 \end{bmatrix}$$

$$\lambda = \begin{bmatrix} 2.42136801 & 1.50396726 & 0.91644987 & 0.80399111 & 0.69360322 & 0.39177751 & 0.3040189 \end{bmatrix}$$

Figure 1 shows the information expressed by each of the eigenvalues. Running some analysis of these values leads to the conclusion that using 5 of the 7 eigenvectors leads to a error of less than 10%. The corresponding eigenvectors have not been added here, but they can be easily visualized by running the python program. The 5 eigenvectors associated with the 5 greatest eigenvalues where put into a matrix of $5x7$ dimensions and they were used to reduce the data set into 5 dimensions.

2.2.3 Fisher Linear Discriminant (FLD)

Unlike, PCA Fisher Linear Discriminant is a supervised technique of dimensional reduction. In other words, this method takes the label of the data into consideration to calculate the necessary statistics. Using FLD, I projected the data set into one dimension. In order to find the projection vector, I calculated the following statistics:

$$\begin{gathered} \mu_1, \qquad \mu_2 \\ S_1 = (n_1 - 1)\Sigma_1, \qquad S_2 = (n_2 - 1)\Sigma_2 \\ S_w = S_1 + S_2 \\ S_w^{-1} = inv(S_w) \\ V = S_w^{-1}(\mu_1 - \mu_2) \end{gathered}$$

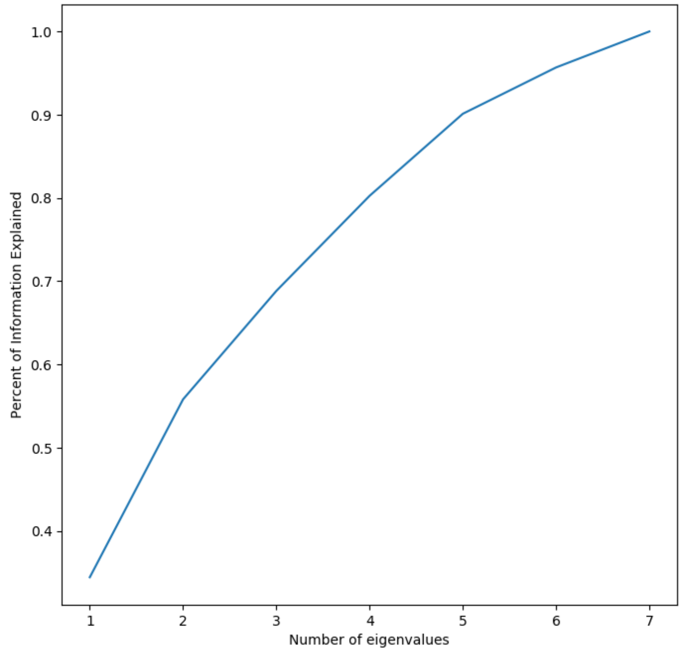


Figure 1: Cumulative information captured by number of eigenvalues. The eigenvalues were ordered from greatest to smallest and then they were added one by one. When x=7, all the eigenvalues are used and there is no dimensional reduction.

The resulting vector V has dimensions 1x7 and is used to transform the data into just one dimension.
For this project, the resulting vector obtained was:

$$V = [-0.00138917, -0.00423824, -0.00021323, 0.00039388, -0.00220392, -0.00186505, -0.00192072]$$

2.3 kNN

First, I want to point out that kNN can be understood in terms of posterior probabilities:

$$p(w_i|x) = \frac{p(x|w_i)p(w_i)}{p(x)}$$

$$p(w_i|x) = \frac{\frac{k_i}{n_i V}\frac{n_i}{n}}{\frac{k}{nV}} = \frac{k_i}{k}$$

In terms of algorithms, I have written 3 functions that work together to classify data based on the k closest Neighbors. The function *knn* accepts the training and testing data set, and an integer value for k. This function loops through the testing data set and calculates the accuracy. The function *euclidiandsitance* computes the distance between the test point that it receives and every other point in the training data set. Then, it sends the results to *guessLabel*, which sorts the array of data and makes a guess based on the k closest training data points.

2.4 Performance Evaluation

Different performance matrices has been used to analyze the models. Additionally, the computational time of each model was computed.

In terms of accuracy, I computed the ratio:

$$\frac{\text{Number of labels guessed correctly}}{\text{Number of total rows}}$$

For kNN the accuracy was calculated through a range of K values from 1 to 100.

In order to further evaluate the classifier's accuracy, I have also compared class-wise accuracy. I created bar plots that compares the True Positive (TP), True Negative (TN), False Positive(FP), and False Negative(FN) values. Assuming that class 1 is "positive" and 0 is "negative", I calculated the Sensitivity or True Positive Rate (TPR):

$$\frac{TP}{TP+FN}$$

and the Specificity or True Negative Rate (TNR):

$$\frac{TN}{TN+FP}$$

Sensitivity and Specificity were plotted against different values of prior probabilities.

2.5 Prior Probabilities

In order to run a prior probability analysis with the kNN model, I introduced two constant α_1 and α_2. Note that based on the data, the prior probability for class 0 is 132/200. In order to find α_1 and α_2 I solved the following equation:

$$\alpha_1 \frac{132}{200} + \alpha_2 \frac{68}{200} = 1$$

We can first check the range of values that α_1 can take:

$$0 \leq \alpha_1 \frac{132}{200} \leq 1$$

So,

$$0 \leq \alpha_1 \leq \frac{200}{132}$$

And therefore,

$$\alpha_2 = \frac{200}{68} - \frac{132}{68}\alpha_1$$

3 Results

3.1 Performance Curves

3.1.1 Normalized Data - nX

Figure 2a shows the performance of kNN using the normalized data set. Running the algorithm with k=1 lead to an accuracy of 71.3% and it took 0.958 seconds. The maximum accuracy is obtained when k=17, and that leads to an accuracy of 80.1%.

3.1.2 Transformed Data set: PCA - pX

In the case of the data set resulting from the PCA transformation, I obtained an accuracy of 60.2% when k=1, and a total accuracy of 73.1% when k=11. The perfomance curve is displayed in Figure 2b. Running kNN with k=1 took 0.803 seconds.

3.1.3 Transformed Data set: FLD - fX

The FLD data set lead to the highest accuracy (Figure 2c). It obtained a total accuracy of 80.4% when k=21. Additionally, the FLD data set also provided the greatest accuracy with k=1, since it classified 79.2% of the entries correctly.

3.1.4 Comparison Table

Table 2 provides a direct comparison of the performance curve analysis presented in this section.

Performance Curves Comparison			
Data Set	Accuracy when K=1	K When Maximum Accuracy	Maximum Accuracy
nX	71.3	17	80.1
pX	60.2	11	73.1
fX	79.2	21	79.2

Table 2

3.2 Performance Comparison

3.2.1 Normalized Data - nX

Graph 3 shows class-wise comparison of kNN vs. each of the 3 Gaussian classifiers. Here, I chose k=1 as the kNN parameter. The total accuracy for Case 1 with the nX data set was 75.0%. Plot 3a shows that kNN produce a large amount of false negatives, but much fewer false positives than the Case I Gaussian classifier. Case II of the Gaussian classifier obtained a total accuracy of 77%, and Case III lead to a 70.5% accuracy.

3.2.2 Transformed Data set: PCA - pX

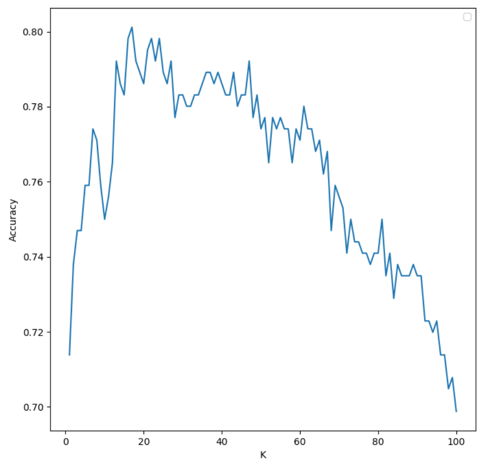
When running the pX data set through the Gaussian cases, I obtained 67.5%, 70.8%, and 60.8% for case I,II,and III respectively (Figure 4a). Here, I also run kNN with k=1. One interesting aspect of these results is how poorly case III performed. Even though, it's the graph that makes the least assumptions about the data.

3.2.3 Transformed Data set: FLD - fX

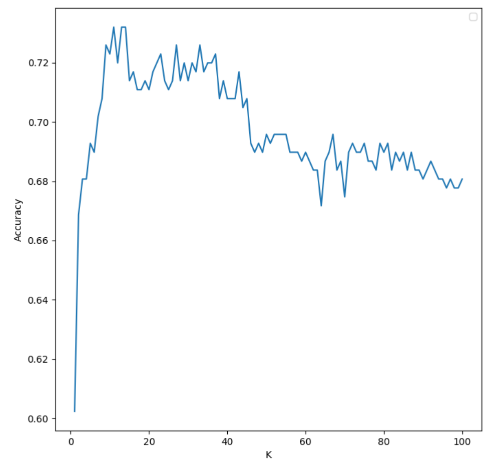
I was not able to come up with results for Case I, Case II, and Case III in the case of the fX data set. The main reason seems to be problems related to calculating statistics such as the covariance matrix, which don't work when the data is in one dimension. I attempted to just use standard deviation, but the program became convoluted enough that it was not possible to provide a solution on time.

3.3 Prior Probability Analysis

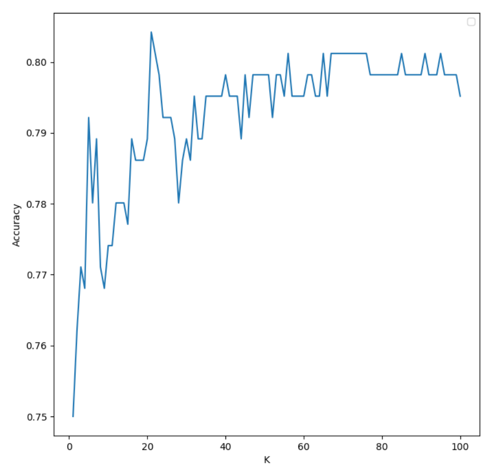
I have run the 3 Gaussian cases together and check the class-wise accuracy. Based on the model output, I created sensitivity and specificity plots.



(a) nX Data set



(b) pX Data Set



(c) fX Dta Set

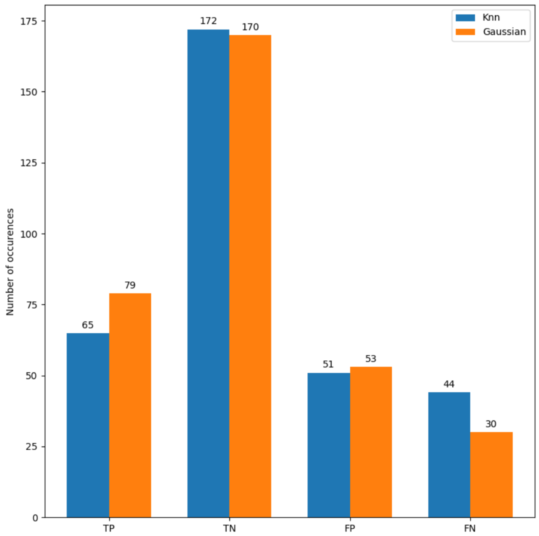
Figure 2: The plots display accuracy vs. K values for the three transformed data sets.

3.3.1 Normalized Data - nX

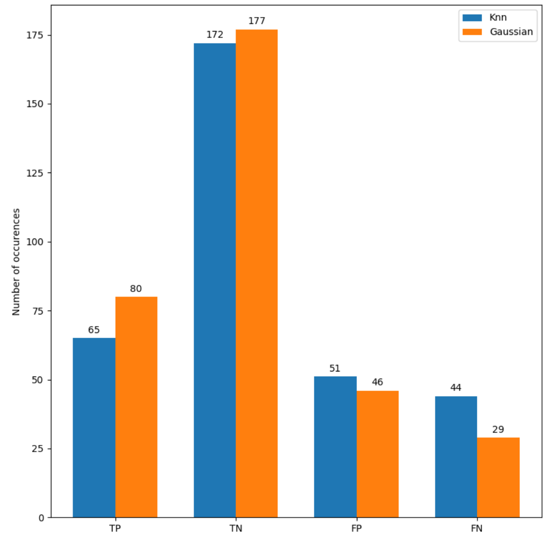
Figure 5 shows the class-wise accuracy for the three Gaussian classifiers used in this paper. All the classifier were trained using the normalized data. From Figure 5a, Case I seems to provide similar results than kNN.

3.3.2 Transformed Data set: PCA - pX

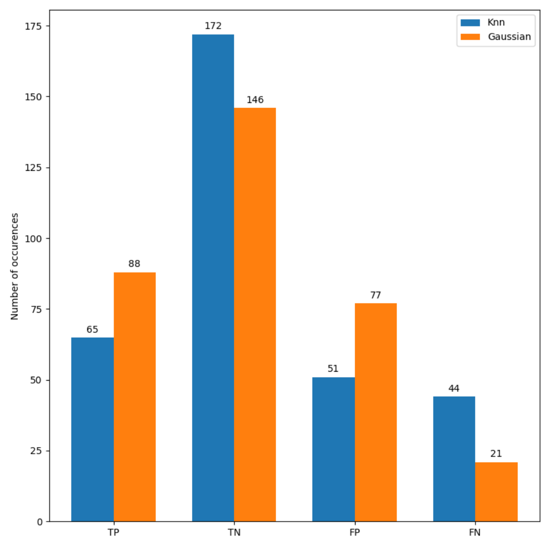
Similarly as in the previous section, I also plotted the class-wise accuracy of the three Gaussian classifiers and kNN for the pX Data set (Figure 6). Unlike the previous picture, here kNN seems to lead to a more similar result to Case II than the rest.



(a) Gaussian Case I vs. kNN



(b) Gaussian Case II vs. kNN



(c) Gaussian Case III vs. kNN

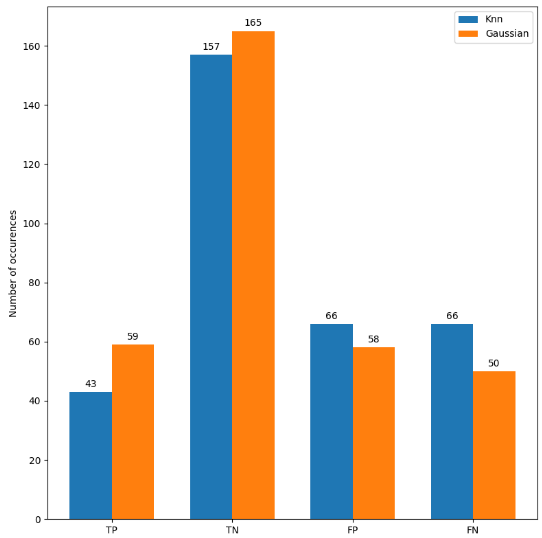
Figure 3: Class-wise accuracy comparison for the Gaussian cases vs. kNN on the nX data set

3.3.3 Transformed Data set: FLD - fX

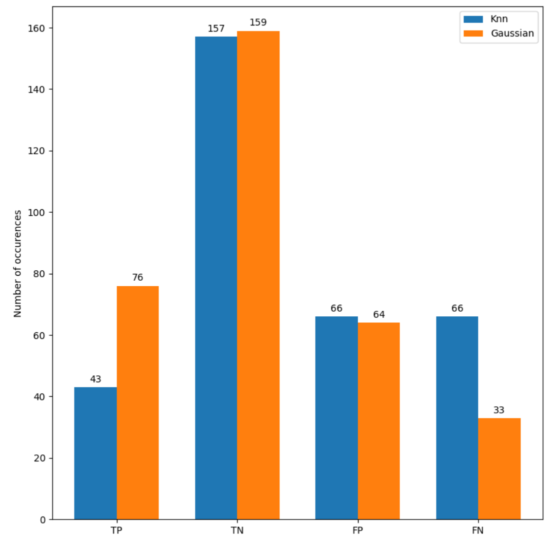
Lastly, the class-wise accuracy for the fX dataset (Figure 7) provide results in similar patterns with Case II and kNN providing almost identical results, and Case III and kNN showing the most differences.

3.4 Eigenvalues vs. Sensitivity and Specificity

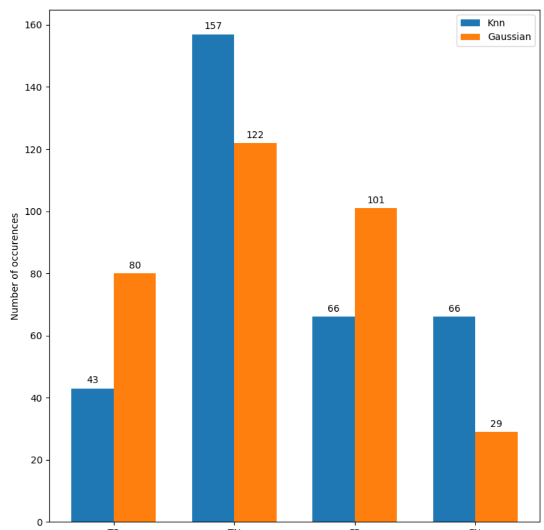
Figure 8 shows the resulting graphs after transforming the data through different numbers of eigenvectors. I first ordered the eigenvalues from greatest to smallest, and I added their eigenvectors to a transformation matrix one by one to compare the results. Each time I added one of the vectors, I transformed the datasets and check the results.



(a) Gaussian Case I vs. kNN

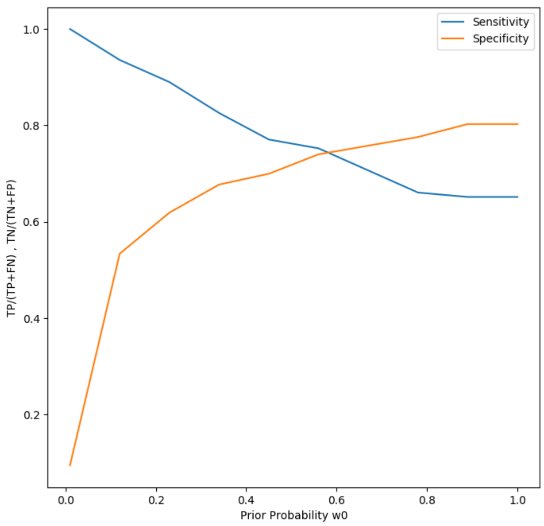


(b) Gaussian Case II vs. kNN

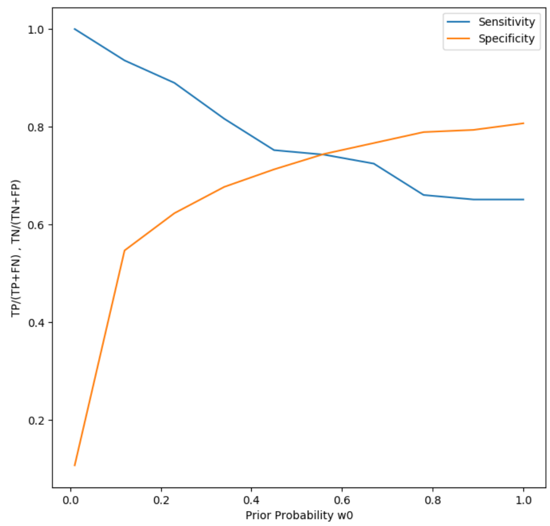


(c) Gaussian Case III vs. kNN

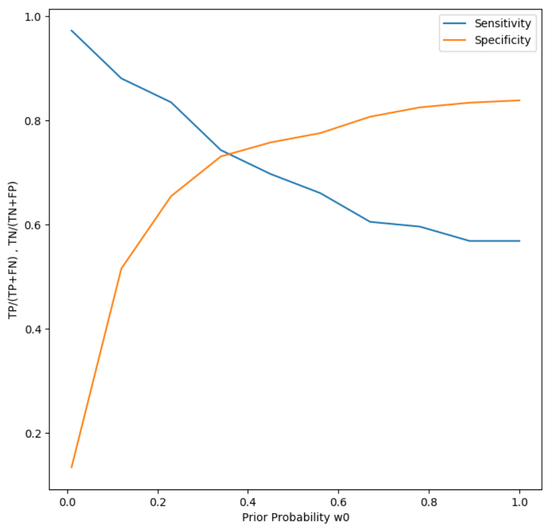
Figure 4: Class-wise accuracy comparison for the Gaussian cases vs. kNN on the pX data set



(a) Gaussian - Case 1

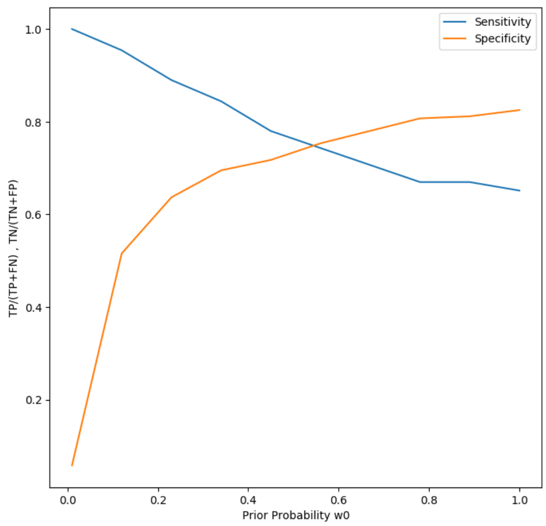


(b) Gaussian - Case 2

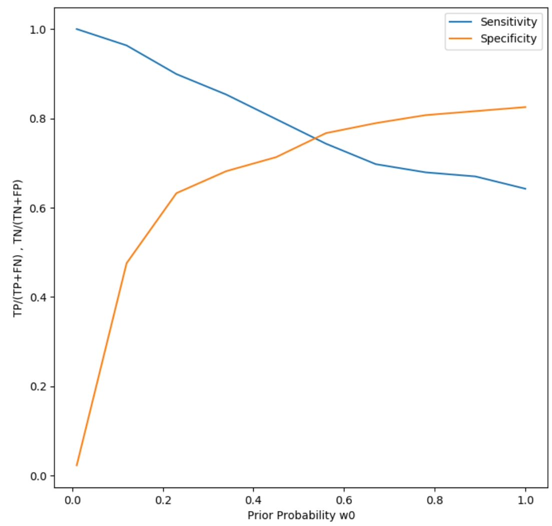


(c) Gaussian - Case 3

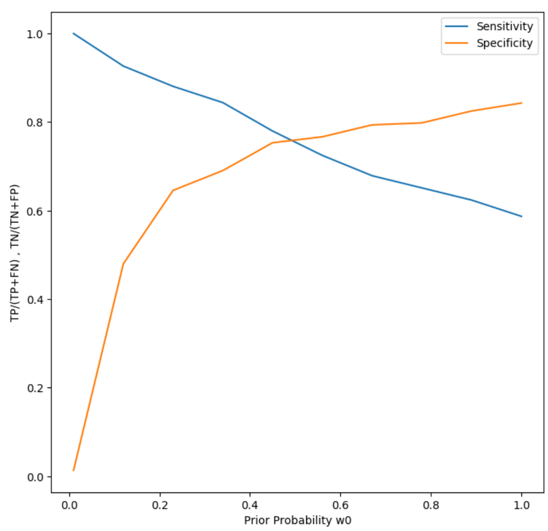
Figure 5: Sensitivity and Specificity curves for the normalized data (nX)



(a) Gaussian - Case 1

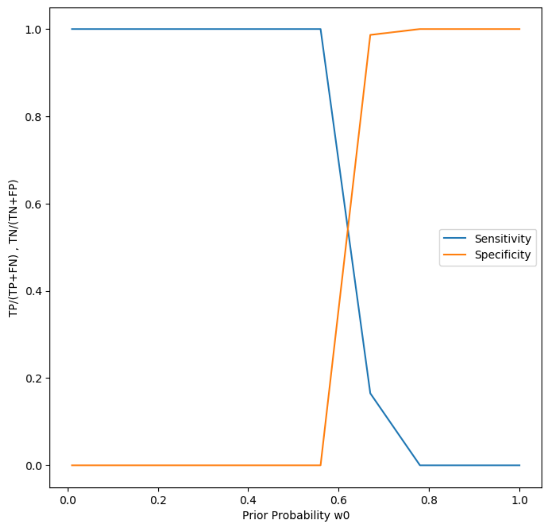


(b) Gaussian - Case 2

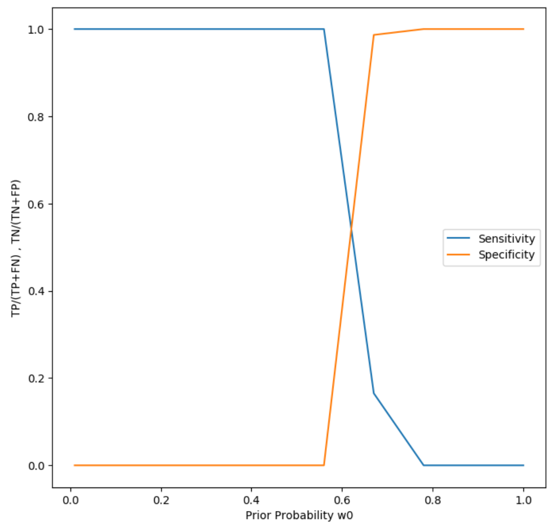


(c) Gaussian - Case 3

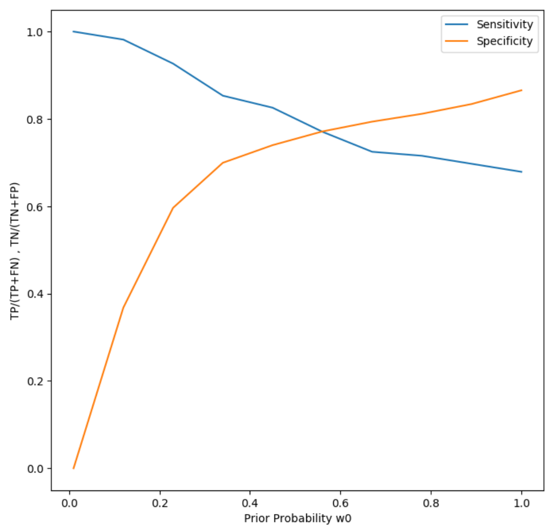
Figure 6: Sensitivity and Specificity curves for the PCA transformed data set (pX)



(a) Gaussian - Case 1

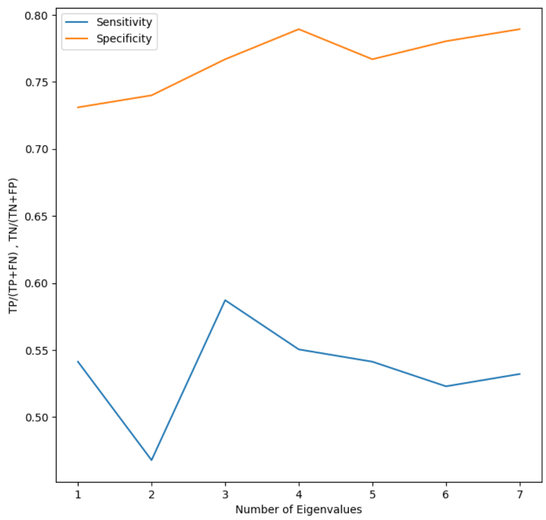


(b) Gaussian - Case 2

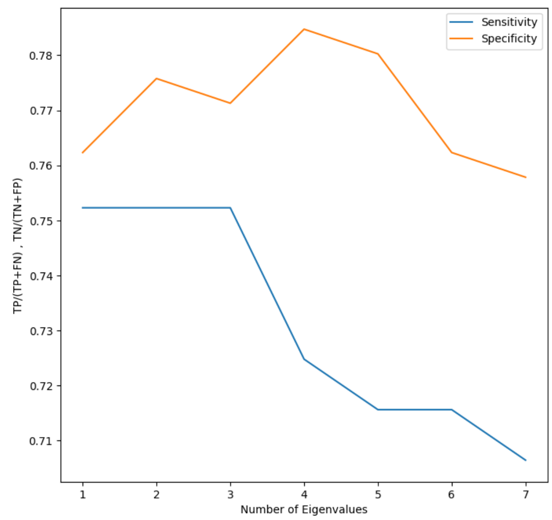


(c) Gaussian - Case 3

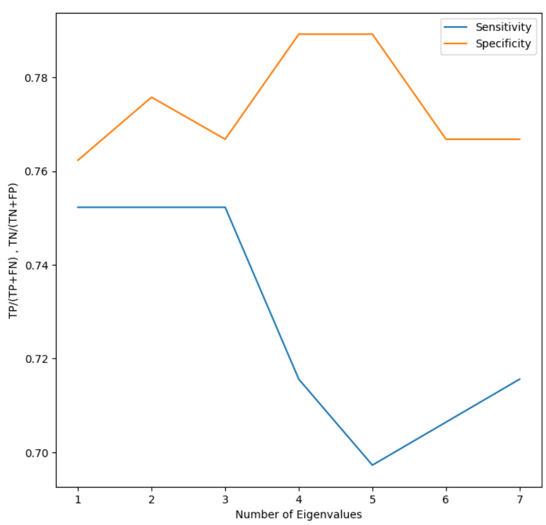
Figure 7: Sensitivity and Specificity curves for the FLD transformed data set (fX)



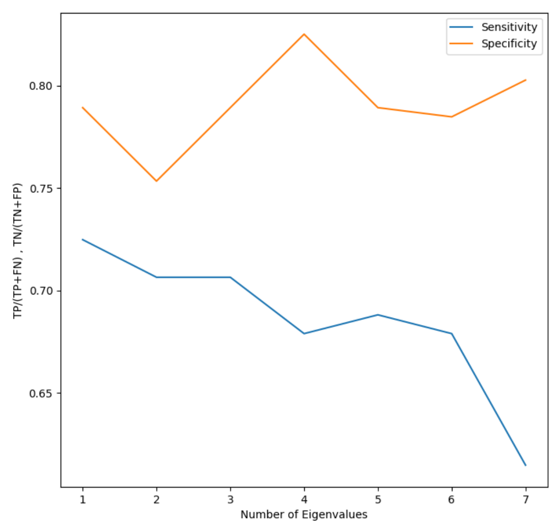
(a)



(b)



(c)



(d)

Figure 8: Cumulative eigenvectors plotted against Sensitivity and Specificity.

4 Discussion

This paper provides an example application of the k-nearest neighbors algorithm that used real world data. In addition to the algorithm itself, I presented the theoretical background behind why kNN actually provides the maximum posterior probability. This project also developed the ideas behind pre-procesing data such as normalization and dimension reduction. As expected of a supervised method, Fisher Linear Discriminant method lead to better results that Principal Component Analysis transformation in general. Overall, kNN performed above 70% with all three data sets. That's a great lower bound considering that it has also reached a 80% accuracy when k=17 in the nX data set. Additionally, I have shown that the Gaussian classifiers have succeed in terms of accuracy. One of the non-intuitive aspects that I found out is that Case II often outperformed Case I and III. I used to think that making fewer assumptions about the data is in general better, but this project proved me somehow wrong. The python program avaialble in the appendix can be easily applied to other data set and used as tools to not just classify data, but also to compare Gaussian classifiers with kNN and see contrast plots.

In a personal note, working on Project 1 and 2 have serves as great experience in becoming a more efficient and professional programmer while merging statistical theory with applications. I've enjoyed the programming aspect extensively, and I tried to make my code as efficient and generalizable as possible.

References

[1] Richard O. Duda, Peter E. Hart, David G. Stork. *Pattern Classification*. Second Edition. pdf.

[2] Sadegh B. Imandoust And Mohammad Bolandraftar *Application of K-Nearest Neighbor (KNN) Approach for Predicting Economic Events: Theoretical Background*. pdf.

[3] Gongde GuoHui WangDavid BellYaxin BiKieran Greer *An kNN Model-Based Approach and Its Application in Text Categorization*. pdf.

[4] Bijalwan V., Kumar V., Kumari P., Pascual P. *KNN based Machine Learning Approach for Text and Document Mining*. pdf.

[5] Liu Q. , Liu C *A Novel Locally Linear KNN Method With Applications to Visual Recognition*. pdf.

5 Appendix

5.1 Python Script

```
import numpy as np
import pandas as pd
import math #Used for Pi and log()
import sympy as sym
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from numpy import ones,vstack
import time


def euc2(x, y):
    # calculate squared Euclidean distance

    # check dimension
    assert x.shape == y.shape

    diff = x - y

    return np.dot(diff, diff)


def mah2(x, y, Sigma):
    # calculate squared Mahalanobis distance

    # check dimension
    assert x.shape == y.shape and max(x.shape) == max(Sigma.shape)

    diff = x - y

    return np.dot(np.dot(diff, np.linalg.inv(Sigma)), diff)



def load_data(f):
    #f is the route of the data file.
    x = pd.read_csv(f, delim_whitespace=1, header=None)
    x[x.shape[1] - 1] = x[x.shape[1] - 1].map({'Yes': 1, 'No': 0})
    #Could return x if I wanted to keep it as the first project.
    X = x.loc[:,0:x.shape[1]-2]
    Y = x.loc[:,x.shape[1]-1]
    Y=Y.to_numpy()
    X=X.to_numpy()
    X=normalization(X)
```

```
    return X, Y

def normalization(X):
    meanArr = np.mean(X, axis=0)
    varArr = np.std(X, axis=0)
    nX = X[:]
    for i in range(X.shape[1]):
        nX[:, i] = (X[:, i] - meanArr[i]) / varArr[i]
    return nX

def pca(nX,percentEerror=.9,showGraph=False):

    nX_Cov = np.cov(nX.T) #Note that covariannce in pd is calculated differently
    nX_eig, nX_eigV = np.linalg.eig(nX_Cov)
    ordered_eigs = -np.sort(-nX_eig)
    totalSum = np.sum(ordered_eigs)
    # Store the indexes of the ordered eigenvalues
    # So you can create a matrix of eigenvectors
    # it the correct order:
    order_index_eigs = []
    for i in range(ordered_eigs.shape[0]):
        order_index_eigs.append(np.where(nX_eig == ordered_eigs[i])[0].item())


    # Store the indexes of the ordered eigenvalues
    # So you can create a matrix of eigenvectors
    # it the correct order:
    order_index_ordered_eigs = []
    for i in range(ordered_eigs.shape[0]):
        order_index_ordered_eigs.append(np.where(nX_eig ==
         ↪  ordered_eigs[i])[0].item())


    def eigenValErrorAnalysis(ordered_eigs):
        plt.figure(num=None, figsize=(8, 8), dpi=100, facecolor='w', edgecolor='k')
        x = np.linspace(1, ordered_eigs.shape[0], ordered_eigs.shape[0])
        # Cumulative:
        eg = []
        totalEig = 0
        for i in range(ordered_eigs.shape[0]):
            totalEig += ordered_eigs[i]
            eg.append(totalEig / totalSum)
        plt.plot(x, eg)
        plt.xlabel(xlabel='Number of eigenvalues')
        plt.ylabel(ylabel='Percent of Information Explained')
        plt.show()
```

```python

    if showGraph == True:
        eigenValErrorAnalysis(ordered_eigs)

    def numberofEigValsToUse(eigs,percentEerror):
        tmp = 0
        ix = -1
        P = []
        for i in range(eigs.shape[0]):
            tmp += eigs[i] / totalSum
            if tmp > percentEerror:
                ix = i + 1  # Number of eigenvectors to use is 5
                for k in range(ix):
                    a = nX_eigV[order_index_eigs[k]]
                    P.append(a)
                P = np.array(P)
                return P

    P = numberofEigValsToUse(ordered_eigs,percentEerror)
    ans=np.dot(nX, np.transpose(P))
    return ans

def fld(nX,y=0, training= True):
    if training==True:
        #split the data int two classes:
        #key = y[:, 0] == 0
        key0 = y==0
        y0Values = nX[key0]
        key1 = y == 1
        y1Values= nX[key1]

        y0ValuesMean = np.mean(y0Values,axis=0)
        y1ValuesMean = np.mean(y1Values,axis=0)


        y0Cov = np.cov(y0Values.T)
        y1Cov = np.cov(y1Values.T)

        S_0 = (y0Values.shape[0] - 1) * y0Cov
        S_1 = (y0Values.shape[0] - 1) * y1Cov

        S_w = S_0 + S_1
        S_w_inv = np.linalg.inv(np.array(S_w))

        fld.v = np.dot(S_w_inv, (np.transpose(y0ValuesMean) -
         ↪  np.transpose(y1ValuesMean)))
```

```python


        y0Values=np.dot(y0Values, fld.v)
        y1Values=np.dot(y1Values, fld.v)


        nX[:, 0][key0] = y0Values
        nX[:, 0][key1] = y1Values
        nX=np.delete(nX, np.s_[1:nX.shape[1]], axis=1)
    else:
        nX = np.dot(nX, fld.v)
    return nX




class Knn:
    def __init__(self):
        self.nX = []
        self.pX = []
        self.fX = []
        self.predictionArr =[]
        self.totalTime= -1

    def showTime(self):
        print("Time in seconds: ", self.totalTime)

    def fit(self, X, y):
        #self.nX = normalization(X)
        self.nX = X
        self.pX = pca(self.nX)
        self.fX = fld(self.nX,y)
        self.y = y

    #x here is just a point
    def euclidian_dsitanceList(self, x, X):
        # x is the test point, X is the dataset
        # Using Euclidian Distance
        distancesArr = []
        # For each row in the data set:
        dist = -1
        index=0 # used to associate a distance with a y.
        for row in X:
            testPoint = row[0:X.shape[1]]
            dist = np.linalg.norm(testPoint - x)
            labelIndex = self.y[index]
```

```
            distanceAndLabel = (dist, labelIndex)
            distancesArr.append(distanceAndLabel)
            index +=1
        return distancesArr


    #x here is just a point
    def guessLabel(self,x, X, k):
        label0 = 0
        guessClass = -1
        label1 = 0
        ks = []
        distancesArr = self.euclidian_dsitanceList(x, X)
        distancesArr.sort()
        for p in range(int(k)):
            minimum = min(distancesArr)
            ks.append((minimum[0], minimum[1]))
            distancesArr.remove(minimum)
        for q in range(len(ks)):
            if ks[q][1] == 0:
                label0 += 1
            else:
                label1 += 1
        if label0 >= label1:
            guessClass = 0
        else:
            guessClass = 1
        return guessClass


    def knn(self,XTest, X, k):
        #X is the training data
        guessList =[]
        guessedLabel = -1
        #print(XTest)
        for row in XTest:
            guessedLabel = self.guessLabel(row, X, k)
            guessList.append(guessedLabel)
        return guessList



    def predict(self, XTest, k=1, data="nX"):
        start_time = time.time()
        predictionArr =[]
        #print(XTest)
```

```
        if data == "nX":
            self.predictionArr = self.knn(XTest, self.nX, k)
        elif data == "pX":
            XTest = pca(XTest)
            self.predictionArr = self.knn(XTest, self.pX, k)
        else:
            XTest=fld(XTest, training=False)
            self.predictionArr = self.knn(XTest, self.fX, k)
        #print(self.predictionArr)
        self.totalTime= time.time() - start_time
        return  self.predictionArr

    def performanceCurve(self, X_test,ytest, K, data="fX"):
        k = np.arange(1, K+1, 1).tolist()
        accuracy_list = []
        for i in k:
            array_prediction=self.predict(X_test, i, data)

            ↪  accuracy_list.append(accuracy_score(ytest,array_prediction,showResults=False
        #print(accuracy_list)
        print("Maximum Accuracy is obtained when K=", k[np.argmax(accuracy_list)])
        print("The accuracy associated with that K = ", np.amax(accuracy_list))
        plt.figure(num=None, figsize=(8, 8), dpi=100, facecolor='w', edgecolor='k')
        plt.plot(k, accuracy_list)
        plt.legend(loc='best')
        plt.xlabel('K')
        plt.ylabel('Accuracy')
        plt.show()






def accuracy_score(yTest, y_model, showResults=True):
    TP, TN, FP, FN = 0, 0, 0, 0
    index=0
    for rightLabel in yTest:
        guessLabel=y_model[index]
        index+=1
        if guessLabel == rightLabel:
            if rightLabel == 1:
                TP += 1
            else:
                TN += 1
        else:
```

```python
            if rightLabel == 1:
                FN += 1
            else:
                FP += 1
    # print(TP)
    totalRowsInData = yTest.shape[0]
    confusion_matrix = [['TP', TP], ["TN", TN], ['FP', FP], ['FN', FN],
    ↪  ['Accuracy', (TN + TP) / totalRowsInData]]
    confusionarr = [TP, TN, FP, FN]
    accuracy = confusion_matrix[4][1]
    if showResults:
        print(confusion_matrix)
    return accuracy, confusionarr



class mpp:
    def __init__(self, case=1):
        # init prior probability, equal distribution
        # self.classn = len(self.classes)
        # self.pw = np.full(self.classn, 1/self.classn)

        # self.covs, self.means, self.covavg, self.varavg = \
        #     self.train(self.train_data, self.classes)
        self.case_ = case
        self.pw_ = None

    def fit(self, Tr, y, fX=False):
        # derive the model
        self.covs_, self.means_ = {}, {}
        self.covsum_ = None

        self.classes_ = np.unique(y)  # get unique labels as dictionary items
        self.classn_ = len(self.classes_)

        for c in self.classes_:
            arr = Tr[y == c]
            self.covs_[c] = np.cov(np.transpose(arr))
            self.means_[c] = np.mean(arr, axis=0)  # mean along rows
            if self.covsum_ is None:
                self.covsum_ = self.covs_[c]
            else:
                self.covsum_ += self.covs_[c]

        if fX==False:
            # used by case II
```

```
            self.covavg_ = self.covsum_ / self.classn_

            # used by case I
            self.varavg_ = np.sum(np.diagonal(self.covavg_)) / len(self.classes_)
        else:
            self.covavg_ = np.std(Tr)
            self.varavg  = np.var(Tr)

    def predict(self, T):
        # eval all data
        y = []
        disc = np.zeros(self.classn_)
        nr, _ = T.shape

        if self.pw_ is None:
            self.pw_ = np.full(self.classn_, 1 / self.classn_)

        for i in range(nr):
            for c in self.classes_:
                if self.case_ == 1:
                    edist2 = euc2(self.means_[c], T[i])
                    disc[c] = -edist2 / (2 * self.varavg_) + np.log(self.pw_[c])
                elif self.case_ == 2:
                    mdist2 = mah2(self.means_[c], T[i], self.covavg_)
                    disc[c] = -mdist2 / 2 + np.log(self.pw_[c])
                elif self.case_ == 3:
                    mdist2 = mah2(self.means_[c], T[i], self.covs_[c])
                    disc[c] = -mdist2 / 2 - np.log(np.linalg.det(self.covs_[c])) /
                    ↪  2 \
                              + np.log(self.pw_[c])
                else:
                    print("Can only handle case numbers 1, 2, 3.")
                    sys.exit(1)
            y.append(disc.argmax())

        return y


def plotBarsAccuracy(confusionM1,confusionM2):
    plt.figure(num=None, figsize=(8, 8), dpi=100, facecolor='w', edgecolor='k')
    labels = ['TP', 'TN', 'FP', 'FN']
    x = np.arange(len(labels))  # the label locations
    width = 0.35  # the width of the bars

    fig, ax = plt.subplots(num=None, figsize=(8, 8), dpi=100, facecolor='w',
    ↪  edgecolor='k')
```

```
    rects1 = ax.bar(x - width/2, confusionM1, width, label='Knn')
    rects2 = ax.bar(x + width/2, confusionM2, width, label='Gaussian')

    # Add some text for labels, title and custom x-axis tick labels, etc.
    ax.set_ylabel('Number of occurences')
    #ax.set_title('Scores by group and gender')
    ax.set_xticks(x)
    ax.set_xticklabels(labels)
    ax.legend()


    def autolabel(rects):
        """Attach a text label above each bar in *rects*, displaying its height."""
        for rect in rects:
            height = rect.get_height()
            ax.annotate('{}'.format(height),
                        xy=(rect.get_x() + rect.get_width() / 2, height),
                        xytext=(0, 3),  # 3 points vertical offset
                        textcoords="offset points",
                        ha='center', va='bottom')


    autolabel(rects1)
    autolabel(rects2)

    fig.tight_layout()
    plt.show()




def main():
    trainingData='/Users/kevindeangeli/Desktop/Fall2019/COSC522/Project2/Dataset_PimaTr.txt'
    testingData='/Users/kevindeangeli/Desktop/Fall2019/COSC522/Project2/Dataset_PimaTest.txt'
    Xtrain,ytrain = load_data(trainingData)
    Xtest,ytest = load_data(testingData)
    model = Knn()
    model.fit(Xtrain, ytrain)

#Predicts accepts values "nX", "pX", "fX". "nX" is by default.
    y_model = model.predict(Xtest,k=1, data="pX")
    accuracy1, confusion_matrix1 = accuracy_score(ytest, y_model)
    #model.showTime()
```

```python
    #model.performanceCurve(Xtest,ytest, K=100, data="fX")




    Xtrain2, ytrain2 = load_data(trainingData)
    Xtrain2=pca(Xtrain2)
    Xtest2, ytest2 = load_data(testingData)
    Xtest2=pca(Xtest2)
    model = mpp(3)
    model.fit(Xtrain2, ytrain2,fX=False)
    y_model2 = model.predict(Xtest2)
    accuracy2, confusion_matrix2 = accuracy_score(ytest2, y_model2)




    plotBarsAccuracy(confusion_matrix1,confusion_matrix2)



if __name__ == "__main__":
    main()
```