# Project 2

**Boling, Kenneth S.**[1]

*(1)Earth and Planetary Sciences, University of Tennessee, Knoxville, TN 37996*

*kboling4@vols.utk.edu*

ECE 571

## Abstract

The curse of dimensionality is a common problem in data analysis: As the number of dimensions in a data set increases, the number of samples needed to support the results of any analysis of that dataset increases exponentially. Consequently, a reduction in dimensionality is often required to adequately analyze a high dimensionality dataset. This project implemented Principal Component Analysis (PCA) and Fisher's Linear Discriminant functions to reduce the dimensionality of a d=7 dataset of a population of women of Pima Indian heritage near Phoenix, Arizona who were tested for diabetes. Four classifiers were trained using the resulting data and the results of these classifiers were compared.

## Contents

Abstract ........ 1

Introduction ........ 2

Methods and Technical Approach ........ 2

- Preprocessing ........ 2
- Principal Component Analysis (PCA) ........ 2
- Fisher's Linear Discriminant ........ 5
- k-Nearest Neighbor (kNN) ........ 6

Experiments and Results ........ 7

References ........ 8

Appendix ........ 12

## Introduction

The objective of this project was to reduce the dimensionality of a d=7 dataset and apply four different classifiers to the resulting data. The provided data set is in the form of two tab delimited text files: *pima.tr* and *pima.te* which contain the training and testing datasets respectively. These files were obtained from: (B.D. Ripley, 1996). The *pima.tr* data set, contains n=200 samples, with d=7 dimensions and a label indicating the presence or absence of diabetes. Within the dataset 68 samples were labeled "Yes" and 132 are labeled "no".

## Methods and Technical Approach

In order to facilitate the data analysis it was necessary to develop a script in Python. This script was written in Python version 3.6.2, using the PyCharm development environment. The script developed to accomplish this task employed the use of several open source Python packages from the *SciPy* library (Jones et al., 2014). The *Pandas* (McKinney, 2010) package was used primarily for the initial data loading and conversation from the provided tab delimited files, and the *NumPy* (Oliphant, 2006) package was used as the primary tool for data management and processing.

### Preprocessing

Before reducing the dimensionality of the dataset a preprocessing step was required. This was done by using a normalization or z-score formula:

**Equation 1:**

$$z = \frac{\overleftarrow{x} - \mu_i}{\sigma_i}$$

Where $x_i$ is a sample vector, $\mu_i$is the mean of each feature, and $\sigma_i$ is the standard deviation of each feature. This was done to center the data around the origin ([0,0] in 2d space) resulting in a mean of zero for each feature.

### Principal Component Analysis (PCA)

The primary goal of PCA is to reduce the dimensionality of the data. This can have the effects of both reducing the "noise" in the data, and getting rid of redundant information.

The normalized data was then transformed using Principal Component Analysis (PCA). The first step was to calculate the covariance matrix:

**Equation 2:**

$$\Sigma = \frac{1}{\mathrm{n}-1}\sum_{i=1}^{n}[(\mathrm{x}-\mu_i)^{\mathrm{T}}(\mathrm{x}-\mu_i)]$$

Covariance matrix of the training data set *pima_tr*

**Table 1**

| | | | | | | |
|---|---|---|---|---|---|---|
| 1.005 | 0.171 | 0.253 | 0.110 | 0.059 | -0.120 | 0.602 |
| 0.171 | 1.005 | 0.271 | 0.219 | 0.218 | 0.061 | 0.345 |
| 0.253 | 0.271 | 1.005 | 0.266 | 0.240 | -0.048 | 0.393 |
| 0.110 | 0.219 | 0.266 | 1.005 | 0.662 | 0.096 | 0.253 |
| 0.059 | 0.218 | 0.240 | 0.662 | 1.005 | 0.192 | 0.133 |
| -0.120 | 0.061 | -0.048 | 0.096 | 0.192 | 1.005 | -0.072 |
| 0.602 | 0.345 | 0.393 | 0.253 | 0.133 | -0.072 | 1.005 |

The next step is to determine the eiganvalues and eiganvectors, which will be used to determine the new axes along which the linear transformation acts. The eigenvector with the largest eigenvalue is the direction along which the data set has the maximum variance and is the first principle component. These values were used to determine the amount of explained variance for each principle component.

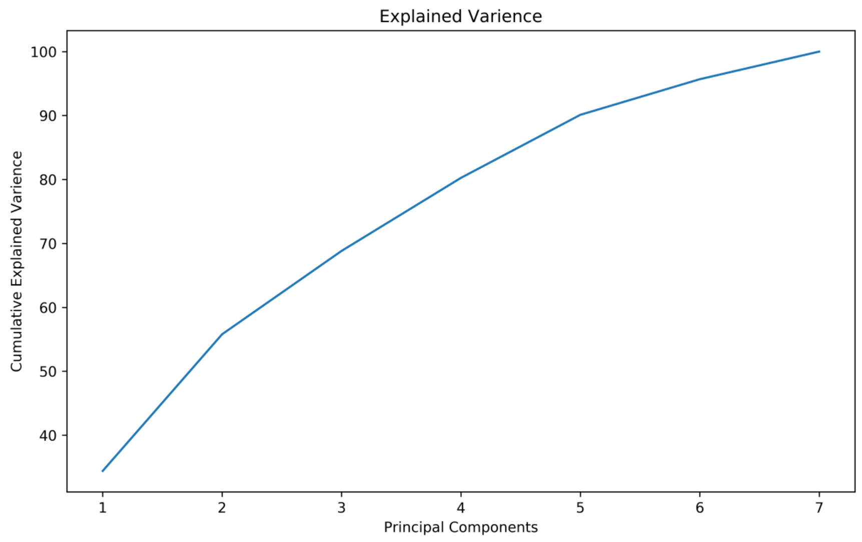


**Figure 1: Cumulative percentage of explained variance of each principal component.**

Table 2

| Component | Explained Variance Ratio | Explained Variance (Cumulative) |
|---|---|---|
| 1 | 0.344 | 0.344 |
| 2 | 0.214 | 0.558 |
| 3 | 0.130 | 0.688 |
| 4 | 0.114 | 0.803 |
| 5 | 0.099 | 0.901 |
| 6 | 0.056 | 0.957 |
| 7 | 0.043 | 1.000 |

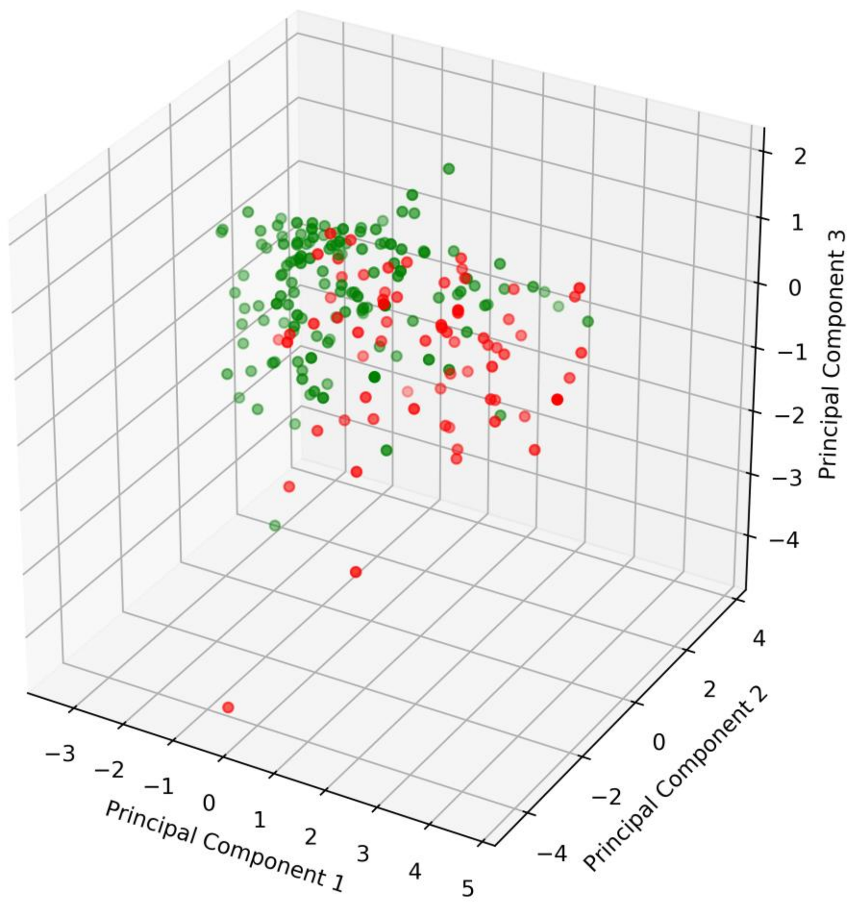


Figure 2: 3-d plot of the first three principal components for the *pima_tr* training dataset. The first three principal components have a cumulative explained variance of 68.8%, and appear to be roughly separable even using only the first three principal components.

From this analysis it was determined that the last two principle components could be excluded while still maintaining 90% of the explained variance. This lowered the number of dimensions from d=7 to d=5. This was then used to generate a projection matrix (**W**) which was used to project the original dataset into d=5 dimensions. This new testing dataset was defined as *pimate_pX.*

## Fisher's Linear Discriminant

Fisher's Linear Discriminant FLD is somewhat similar to PCA in that it can be used to reduce the dimensionality of a dataset. The difference is that the objective of PCA is to find the best basis to re-project the dataset along the axes of highest variance, while the FLD method is used to reproject the dataset along the axes that maximize the differences between classes. More simply, FLD is a form of supervised learning which maximizes the differences (separation distance) between classes, while PCA can be considered a form of unsupervised learning, since the reprojection of the data is performed without regard to the classes.

The projection matrix (W) is found by first calculating the within class scatter matrix (Sw) and between class scatter matrix (Sb).

Within class scatter matrix:

**Equation 3:**

$$S_W = \sum_{\text{classes}} \sum_{i \in c} (x_i - \mu_c)(x_i - \mu_c)^T$$

Between class scatter matrix:

**Equation 4:**

$$S_B = \sum_{\text{classes}} \mathrm{N}_c (\mu_c - \mu)(\mu_c - \mu)^T$$

The Fisher's Linear Discriminant was applied to the *pimate_nX* dataset to reduce the dimensionality of the dataset and the projected dataset was defined as *pimate_fX.*

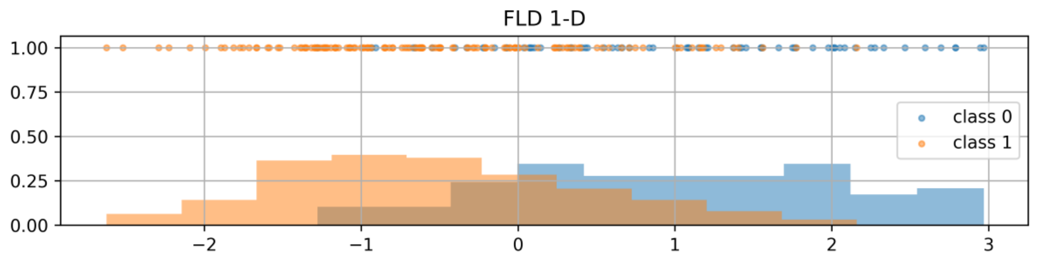


**Figure 3 Plot showing the training data points and probability density of the two classes. Class 0 = Yes, Class 1= No.**

## k-Nearest Neighbor (kNN)

One example of non-parametric learning is k-Nearest Neighbor. The biggest advantage to kNN is that it does not require making assumptions about parameters of the distribution of data. A kNN classifier was developed in Python 3 using a Pandas data frame as the data structure. There are several computational advantages to using a Pandas data frame for the implementation of a kNN classifier, most notably that basic functions are executed in C and are highly optimized for performance. Another advantage is the ability to apply a function to each row of a data frame using *DataFrame.apply* function instead of using for loops (though the functionality is very similar). The kNN classifier (defined in the code section as *Kens_kNN* ) completed the classification of the *pimate_nX* (k=1) dataset in an average of ~2.2 seconds for 10 trials. Further time savings could still be made by optimizing parts of the code that deal with appending the results into the final data frame.

**Table 3: Table of values of k and accuracy of classification using the pimate_nX data (please see the appendix for more detailed data).**

| k= | correct | incorrect | Accuracy |
|---|---|---|---|
| 1 | 236 | 96 | 71.08% |
| 2 | 241 | 91 | 72.59% |
| 3 | 246 | 86 | 74.10% |
| 4 | 249 | 83 | 75.00% |
| 5 | 249 | 83 | 75.00% |
| 6 | 249 | 83 | 75.00% |
| 7 | 250 | 82 | 75.30% |
| 8 | 252 | 80 | 75.90% |
| 9 | 257 | 75 | 77.41% |
| 10 | 255 | 77 | 76.81% |

## Experiments and Results

The number of True Positives (TP), True Negatives (TN), False Positives (FP), and False Negatives (FN), as well as the accuracy, True Positive Rate (TPR), and False Positive Rate (FPR) for each set of processed data were determined. Different values of 'k' were used to determine the optimal value for 'k'. For the nX data setting k= 22 yielded the highest overall accuracy and TPR (See appendix for complete data).

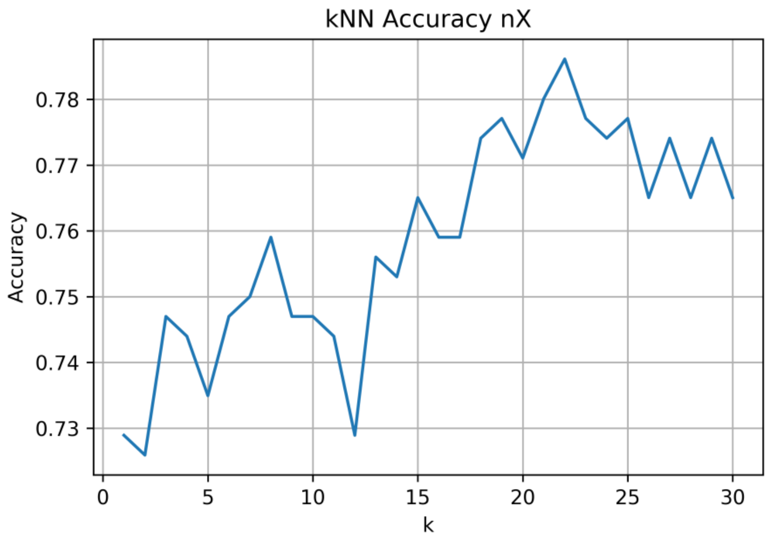


Figure 4

The PCA projected data set using the first 5 principal components had the highest accuracy at k=27, although the highest TPR was at k = 22.

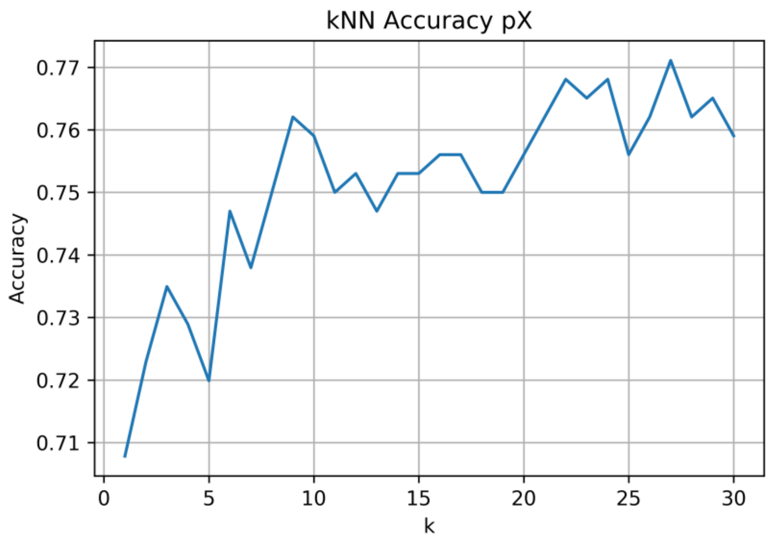


Figure 5

The FLD projected data had the highest accuracy for all the kNN classifiers at k=17,

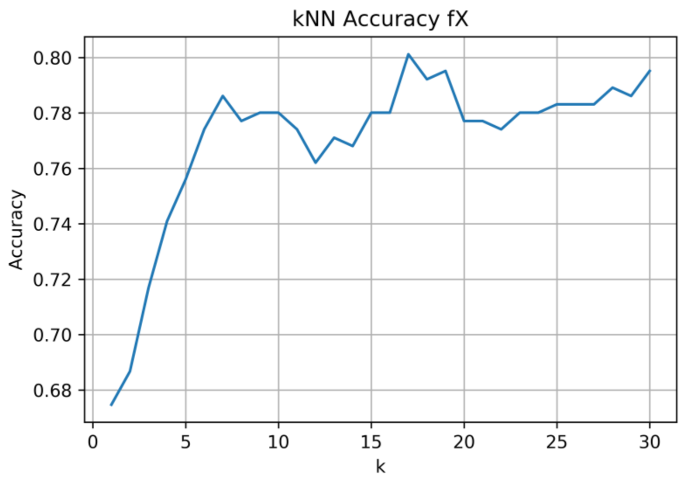


Figure 6

## Classifier Performance

### Case 1

Case 1 classifier was nothing if not the fastest, completing the classification task in 0.011s using the normalized data (nX), 0.009s for the PCA projected data (pX), and 0.006s for the FLD projected data (fX). The accuracy was surprisingly high considering that the only data used to train this classifier was the mean of each class of the training set.

| Case 1 | FN | FP | TN | TP | Acc. |
|---|---|---|---|---|---|
| nX | 54 | 32 | 169 | 77 | 0.741 |
| pX | 56 | 37 | 167 | 72 | 0.720 |
| fX | 48 | 28 | 175 | 81 | 0.771 |

(Case 1 and 2 still have to be rewritten from their original versions from project 1, since there were some poor design choices that were made at the time which made them incompatible with the workflow generated for this project.  They still do not function correctly.)

## Case 2

## Case 3

## Discussion

The best results appear to have been achieved by using a kNN classifier on the FLD projected data.  This was unexpected, since it would seem that the amount of data lost by projecting the data into a single dimension would be detrimental to the accuracy.  Case 1 classifier was also surprisingly more effective than expected particularly for the fX projected data.  Although the high accuracy for fX makes more sense when the fact that the FLD projected data is highly separated and simple to create an accurate linear decision boundary across.

## References

B.D. Ripley, 1996, Datasets used in the book Pattern Recognition and Neural Networks by B.D. Ripley (1996):, http://www.stats.ox.ac.uk/pub/PRNN/ (accessed January 2019).

Jones, E., Oliphant, T., and Peterson, P., 2014, {SciPy}: open source scientific tools for {Python}

McKinney, W., 2010, Data structures for statistical computing in python, *in* Proceedings of the 9th Python in Science Conference, Austin, TX, v. 445, p. 51–56.

Oliphant, T.E., 2006, A guide to NumPy: Trelgol Publishing USA, v. 1.

## Appendix

Please see the included Jupyter Notebook (or the .html version which should be viewable in any browser) for additional notes, compete code, and other information not included in this section.

nX kNN results:

| FN | FP | TN | TP | FPR | TPR | Acc | k |
|---|---|---|---|---|---|---|---|
| 38 | 52 | 185 | 57 | 0.219 | 0.600 | 0.729 | 1 |
| 14 | 77 | 209 | 32 | 0.269 | 0.696 | 0.726 | 2 |
| 29 | 55 | 194 | 54 | 0.221 | 0.651 | 0.747 | 3 |
| 11 | 74 | 212 | 35 | 0.259 | 0.761 | 0.744 | 4 |
| 28 | 60 | 195 | 49 | 0.235 | 0.636 | 0.735 | 5 |
| 12 | 72 | 211 | 37 | 0.254 | 0.755 | 0.747 | 6 |
| 26 | 57 | 197 | 52 | 0.224 | 0.667 | 0.750 | 7 |
| 14 | 66 | 209 | 43 | 0.240 | 0.754 | 0.759 | 8 |
| 27 | 57 | 196 | 52 | 0.225 | 0.658 | 0.747 | 9 |
| 17 | 67 | 206 | 42 | 0.245 | 0.712 | 0.747 | 10 |
| 26 | 59 | 197 | 50 | 0.230 | 0.658 | 0.744 | 11 |
| 21 | 69 | 202 | 40 | 0.255 | 0.656 | 0.729 | 12 |
| 24 | 57 | 199 | 52 | 0.223 | 0.684 | 0.756 | 13 |
| 18 | 64 | 205 | 45 | 0.238 | 0.714 | 0.753 | 14 |
| 22 | 56 | 201 | 53 | 0.218 | 0.707 | 0.765 | 15 |
| 17 | 63 | 206 | 46 | 0.234 | 0.730 | 0.759 | 16 |
| 22 | 58 | 201 | 51 | 0.224 | 0.699 | 0.759 | 17 |
| 15 | 60 | 208 | 49 | 0.224 | 0.766 | 0.774 | 18 |
| 20 | 54 | 203 | 55 | 0.210 | 0.733 | 0.777 | 19 |
| 18 | 58 | 205 | 51 | 0.221 | 0.739 | 0.771 | 20 |
| 21 | 52 | 202 | 57 | 0.205 | 0.731 | 0.780 | 21 |
| 14 | 57 | 209 | 52 | 0.214 | 0.788 | 0.786 | 22 |
| 20 | 54 | 203 | 55 | 0.210 | 0.733 | 0.777 | 23 |
| 15 | 60 | 208 | 49 | 0.224 | 0.766 | 0.774 | 24 |
| 17 | 57 | 206 | 52 | 0.217 | 0.754 | 0.777 | 25 |
| 17 | 61 | 206 | 48 | 0.228 | 0.738 | 0.765 | 26 |
| 18 | 57 | 205 | 52 | 0.218 | 0.743 | 0.774 | 27 |
| 16 | 62 | 207 | 47 | 0.230 | 0.746 | 0.765 | 28 |
| 18 | 57 | 205 | 52 | 0.218 | 0.743 | 0.774 | 29 |
| 16 | 62 | 207 | 47 | 0.230 | 0.746 | 0.765 | 30 |

pX kNN results:

| FN | FP | TN | TP | FPR | TPR | Acc | k |
|---|---|---|---|---|---|---|---|
| 52 | 45 | 171 | 64 | 0.208 | 0.552 | 0.708 | 1 |
| 25 | 67 | 198 | 42 | 0.253 | 0.627 | 0.723 | 2 |
| 40 | 48 | 183 | 61 | 0.208 | 0.604 | 0.735 | 3 |
| 28 | 62 | 195 | 47 | 0.241 | 0.627 | 0.729 | 4 |
| 42 | 51 | 181 | 58 | 0.220 | 0.580 | 0.720 | 5 |
| 26 | 58 | 197 | 51 | 0.227 | 0.662 | 0.747 | 6 |
| 33 | 54 | 190 | 55 | 0.221 | 0.625 | 0.738 | 7 |
| 25 | 58 | 198 | 51 | 0.227 | 0.671 | 0.750 | 8 |
| 29 | 50 | 194 | 59 | 0.205 | 0.670 | 0.762 | 9 |
| 23 | 57 | 200 | 52 | 0.222 | 0.693 | 0.759 | 10 |
| 31 | 52 | 192 | 57 | 0.213 | 0.648 | 0.750 | 11 |
| 24 | 58 | 199 | 51 | 0.226 | 0.680 | 0.753 | 12 |
| 31 | 53 | 192 | 56 | 0.216 | 0.644 | 0.747 | 13 |
| 24 | 58 | 199 | 51 | 0.226 | 0.680 | 0.753 | 14 |
| 28 | 54 | 195 | 55 | 0.217 | 0.663 | 0.753 | 15 |
| 23 | 58 | 200 | 51 | 0.225 | 0.689 | 0.756 | 16 |
| 26 | 55 | 197 | 54 | 0.218 | 0.675 | 0.756 | 17 |
| 24 | 59 | 199 | 50 | 0.229 | 0.676 | 0.750 | 18 |
| 27 | 56 | 196 | 53 | 0.222 | 0.663 | 0.750 | 19 |
| 23 | 58 | 200 | 51 | 0.225 | 0.689 | 0.756 | 20 |
| 27 | 52 | 196 | 57 | 0.210 | 0.679 | 0.762 | 21 |
| 21 | 56 | 202 | 53 | 0.217 | 0.716 | 0.768 | 22 |
| 26 | 52 | 197 | 57 | 0.209 | 0.687 | 0.765 | 23 |
| 20 | 57 | 203 | 52 | 0.219 | 0.722 | 0.768 | 24 |
| 27 | 54 | 196 | 55 | 0.216 | 0.671 | 0.756 | 25 |
| 20 | 59 | 203 | 50 | 0.225 | 0.714 | 0.762 | 26 |
| 22 | 54 | 201 | 55 | 0.212 | 0.714 | 0.771 | 27 |
| 21 | 58 | 202 | 51 | 0.223 | 0.708 | 0.762 | 28 |
| 23 | 55 | 200 | 54 | 0.216 | 0.701 | 0.765 | 29 |
| 19 | 61 | 204 | 48 | 0.230 | 0.716 | 0.759 | 30 |

fx kNN results

| FN | FP | TN | TP | Acc | TPR | FPR | k |
|---|---|---|---|---|---|---|---|
| 50 | 58 | 173 | 51 | 0.675 | 0.505 | 0.251 | 1 |
| 29 | 75 | 194 | 34 | 0.687 | 0.540 | 0.279 | 2 |
| 45 | 49 | 178 | 60 | 0.717 | 0.571 | 0.216 | 3 |
| 28 | 58 | 195 | 51 | 0.741 | 0.646 | 0.229 | 4 |
| 33 | 48 | 190 | 61 | 0.756 | 0.649 | 0.202 | 5 |
| 19 | 56 | 204 | 53 | 0.774 | 0.736 | 0.215 | 6 |
| 28 | 43 | 195 | 66 | 0.786 | 0.702 | 0.181 | 7 |
| 19 | 55 | 204 | 54 | 0.777 | 0.740 | 0.212 | 8 |
| 33 | 40 | 190 | 69 | 0.780 | 0.676 | 0.174 | 9 |
| 21 | 52 | 202 | 57 | 0.780 | 0.731 | 0.205 | 10 |
| 29 | 46 | 194 | 63 | 0.774 | 0.685 | 0.192 | 11 |
| 20 | 59 | 203 | 50 | 0.762 | 0.714 | 0.225 | 12 |
| 25 | 51 | 198 | 58 | 0.771 | 0.699 | 0.205 | 13 |
| 14 | 63 | 209 | 46 | 0.768 | 0.767 | 0.232 | 14 |
| 24 | 49 | 199 | 60 | 0.780 | 0.714 | 0.198 | 15 |
| 16 | 57 | 207 | 52 | 0.780 | 0.765 | 0.216 | 16 |
| 21 | 45 | 202 | 64 | 0.801 | 0.753 | 0.182 | 17 |
| 16 | 53 | 207 | 56 | 0.792 | 0.778 | 0.204 | 18 |
| 20 | 48 | 203 | 61 | 0.795 | 0.753 | 0.191 | 19 |
| 14 | 60 | 209 | 49 | 0.777 | 0.778 | 0.223 | 20 |
| 21 | 53 | 202 | 56 | 0.777 | 0.727 | 0.208 | 21 |
| 12 | 63 | 211 | 46 | 0.774 | 0.793 | 0.230 | 22 |
| 21 | 52 | 202 | 57 | 0.780 | 0.731 | 0.205 | 23 |
| 17 | 56 | 206 | 53 | 0.780 | 0.757 | 0.214 | 24 |
| 19 | 53 | 204 | 56 | 0.783 | 0.747 | 0.206 | 25 |
| 15 | 57 | 208 | 52 | 0.783 | 0.776 | 0.215 | 26 |
| 20 | 52 | 203 | 57 | 0.783 | 0.740 | 0.204 | 27 |
| 14 | 56 | 209 | 53 | 0.789 | 0.791 | 0.211 | 28 |
| 19 | 52 | 204 | 57 | 0.786 | 0.750 | 0.203 | 29 |
| 14 | 54 | 209 | 55 | 0.795 | 0.797 | 0.205 | 30 |

PCA Eiganvalues and Eiganvectors

| Eiganvalues | Eiganvectors |
|---|---|
| 2.421 | [0.36549332 0.36600186 0.41257275 0.43153727 0.39121306 0.02913468 0.47129619] |
| 1.504 | [ 0.46252565 0.01274298 0.1081158 -0.414656 -0.51287765 -0.44983379 0.36997986] |
| 0.304 | [ 0.30622668 0.1071672 0.08473788 0.5842483 -0.57511643 0.07926388 -0.45735343] |
| 0.392 | [ 0.53225391 0.13776248 0.10826822 -0.38145872 0.41385285 -0.04162482 -0.60620764] |
| 0.694 | [ 0.13226904 0.4448877 -0.82394333 0.15432682 0.10388828 -0.25938476 0.06170272] |
| 0.804 | [ 0.50059302 -0.68010455 -0.33498173 0.13305369 0.11380797 0.32442578 0.19683508] |
| 0.916 | [-0.08559894 -0.41826987 0.08998228 0.33683691 0.24110317 -0.78501771 -0.14759294] |

PCA projection matrix:

| W |
|---|
| -0.410 |
| -1.165 |
| 0.032 |
| 0.015 |
| -0.464 |
| -0.589 |
| -0.528 |

## Covariance matrix

```
# function to calculate the covariance matrix from a data set
def covmat(matrix):

    d = matrix.ndim # gets the number of dimensions in the matrix
    Ei = np.zeros([d , d]) # Creates an array of 0s that is d by d
    n = np.shape(matrix)[0] #numberr of rows in the matrix

    summ = (np.sum(matrix, axis = 0)) #sum of each column of the matrix
    muv = np.array([[(1/n)* summ[0]],[(1/n)* summ[1]]]) #calculates Mu vector

    for irow in matrix: # for loops iterate through the rows of an numpy array by 
default
        xrow = irow.reshape(-1,1) #converts the row vector to a column vector
        isum = np.outer([(xrow - muv)] , [(np.transpose(xrow) - np.transpose(muv))]) # 
calculate (x - mu)(x - mu)T for that row
        Ei = Ei + isum # adds the matrices to the total

    covmatrix = ((1 / (n-1)) * Ei )

    return covmatrix

def covnum(df):
        #num_cols = df.columns[df.dtypes.apply(lambda c: np.issubdtype(c, np.number))]
        #dfcol_select = df[num_cols]
        dfnum = df.select_dtypes(include=np.number)
        #cov = np.cov(dfnum,rowvar=False)
        cov = np.cov(dfnum,rowvar=False)

        return pd.DataFrame(cov)
```

## PCA

```
def transformpca(df,matrix_w):
    dfnum = df.select_dtypes(include=np.number)
    dfobj = df.select_dtypes(include=object)
    dft= dfnum.dot(matrix_w)
    dfcon = pd.concat([dft,dfobj],axis=1)
    dfout = dfcon.rename(columns={0:'Principal Component 1',1:'Principal Component 
2',2:'Principal Component 3',3:'Principal Component 4',
                                  4:'Principal Component 5'})

    return dfout
```

**kNN**

```
def Kens_kNNnew(df_training, class_column, df_testing, k ):
    start = time.time()
    df_training_num = pd.DataFrame(df_training.select_dtypes(include=np.number))
    df_testing_num = pd.DataFrame(df_testing.select_dtypes(include=np.number))
    kNNlist = []

    def rowply(row):

        Xdist = pd.DataFrame(np.sqrt(np.sum(np.power(np.subtract(df_training_num,
row.T), 2), axis=1)))

        Xdist.rename(columns={0: 'dist'}, inplace=True)
        trdataX = pd.concat([Xdist, df_training[class_column]], axis=1)
        test1 = trdataX.nsmallest(n=k, columns='dist', keep='first')

        test2 = test1.groupby(class_column).size().reset_index()

        test2.rename(columns={0: 'count'}, inplace=True)

        test3 = test2.nlargest(n=1, columns='count', keep='first')

        kNNlist.append(test3[class_column].values)

        return

    df_testing_num.apply(rowply, axis=1)

    kNNclasslistapp = pd.DataFrame(np.concatenate(kNNlist),index=df_testing.index)
    kNNclasslistapp.rename(columns={0: 'kNN_class'}, inplace=True)
    Xout = pd.concat([df_testing, kNNclasslistapp], axis=1)
    end = time.time()
    print('kNN time to complete:', end - start, 'seconds')
    return Xout
```

**FLD**

```
def sk_LDA(train_df, test_df):
    num_cols = train_df.columns[train_df.dtypes.apply(lambda c: np.issubdtype(c, 
np.number))]
    num_dict = {'Yes': 0,'No':1}
    X = train_df[num_cols]
    X_test  = test_df[num_cols]
    X_arr = X.to_numpy(dtype='float64')
    X_test_arr = X_test.to_numpy(dtype='float64')

    print(X_arr.shape)
    df_y = train_df['diabetes']
    df_y_te = test_df['diabetes']

    y =  np.asarray(df_y.replace(num_dict))
    print(y.shape)

    LDA = LinearDiscriminantAnalysis(store_covariance=True,solver='eigen')
    output = LDA.fit(X=X_arr,y=y).transform(X_arr)
    df_training_transformed = pd.DataFrame(output,columns=['FLD'])

    df_training_output = pd.concat([df_training_transformed,df_y],axis=1)

    testing_set_transformed = LDA.transform(X_test_arr)
    df_testing_transformed = pd.DataFrame(testing_set_transformed,columns=['FLD'])

    df_testing_output = pd.concat([df_testing_transformed,df_y_te],axis=1)


    print(LDA.intercept_)

    W = LDA.coef_ #Weight vectors
    print(LDA.coef_ )
    return df_training_output, df_testing_output, output, y, W,
```

**Case 1**

```
# Case 1 (Euclidean Distance)
# Define the Case 1 function

def euclidist(X, muv1, muv2, class_column):

    classes = pd.DataFrame(data=X[class_column])

    start = time.time() # timer
    df_testing_num = pd.DataFrame(X.select_dtypes(include=np.number))



    #xy = X[X.columns[:2]]

    # creates two new columns with the distance of each point from the mean of each
class
    euclid_dist_class_0 = np.sqrt(np.sum(np.power(np.subtract(df_testing_num,
muv1.T),2),axis=1))
    euclid_dist_class_1 = np.sqrt(np.sum(np.power(np.subtract(df_testing_num,
muv2.T),2),axis=1))

    # selects the class based on the smaller of the two
    conditions = (euclid_dist_class_0 < euclid_dist_class_1,
                  euclid_dist_class_0 > euclid_dist_class_1 )
    choices = ('Yes' , 'No')
    results = np.select(conditions, choices)

    output = pd.DataFrame(results,columns=['case_1_class'],index=X.index)
    #classes['case_1_class']
    # I don't remember why I put this here, but I probably should not touch it... too
late...
    Xout = pd.DataFrame(data=pd.concat([output,classes],axis=1))

    # prints
    end = time.time()
    print ('time to complete:', end - start, 'seconds')

    return Xout
# Case 2 (Mahalanobis Distance)

# Define the Case 2 function:
```

**Case 2 (still not working)**

```
def mahalanobisdist2(X, E1, E2, muv1, muv2):
    start = time.time() # timer started
    df_testing_num = pd.DataFrame(X.select_dtypes(include=np.number))

    #xy = np.array(X.iloc[:,:2]) # select all rows in the first 2 columns
    case_2_class = [] #emtpy data object to add reslts to
    for irow in df_testing_num:
        xrow = irow.reshape(-1,1) #converts the row vector to a column vector

        # Calculates the mahalanobis distance between each data point
        # and the mean determined for each class

        md1 = np.dot(np.dot((xrow - muv1).T , np.linalg.inv(E1)),(xrow - muv1))

        md2 = np.dot(np.dot((xrow - muv2).T , np.linalg.inv(E2)) ,(xrow - muv2))

        g1 = -md1/2
        g2 = -md2/2

        conditions = (g1 > g2, g1 < g2)
        choices = ('Yes' , 'No')
        select = np.select(conditions, choices)
        case_2_class.append(select)



    discrim = pd.DataFrame(np.concatenate(case_2_class))
    discrim.rename(columns={0: 'case_2_class'}, inplace=True)

    Xout = pd.concat([X, discrim], axis=1)

    # prints
    end = time.time()
    print ('time to complete:', end - start, 'seconds')
    return Xout
```

**Case 3 (still not working)**

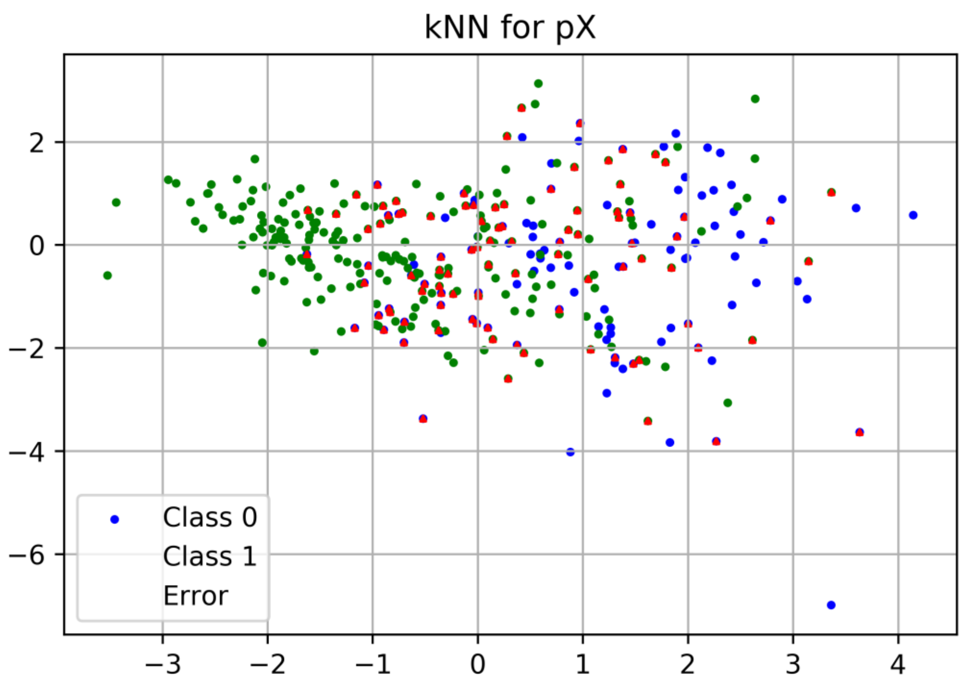
```
# Case 3 (Maximum A-Priori Probability, Mahalanobis Distance)

# Define the Case 3 function:

def case_3_discrim(X, E1, E2, muv1, muv2):
    start = time.time() # timer started
    xy = np.array(X.iloc[:,:2]) # select all rows in the first 2 columns
    case_2_class = [] #emtpy data object to add reslts to
    for irow in xy:
        xrow = irow.reshape(-1,1) #converts the row vector to a column vector
        test1 = (xrow - muv1)
        #print(test1)

        # Calculates the mahalanobis distance between each
        #  data point and the mean determined for each class

        md1 = np.dot(np.dot((xrow - muv1).T , np.linalg.inv(E1)),(xrow - muv1))

        md2 = np.dot(np.dot((xrow - muv2).T , np.linalg.inv(E2)) ,(xrow - muv2))

        g1 = -md1/2 - ((np.log(np.linalg.det(E1)))/2)
        g2 = -md2/2 - ((np.log(np.linalg.det(E2)))/2)

        #md1 = np.sqrt(
                   # np.dot(
        #                np.dot((xrow - muv1).T , np.linalg.inv(E1))
         #                   ,(xrow - muv1)))
       # md2 = np.sqrt(
        #             np.dot(
        #                np.dot((xrow - muv2).T , np.linalg.inv(E2))
        #                   ,(xrow - muv2)))
        conditions = (g1 > g2, g1 < g2)
        choices = (0, 1)
        select = np.select(conditions, choices)
        case_2_class.append(select)



    discrim = pd.DataFrame(np.concatenate(case_2_class))
    discrim.rename(columns={0: 'case_2_class'}, inplace=True)

    Xout = pd.concat([X, discrim], axis=1)

    # prints
    end = time.time()
    print ('time to complete:', end - start, 'seconds')
    return Xout
```

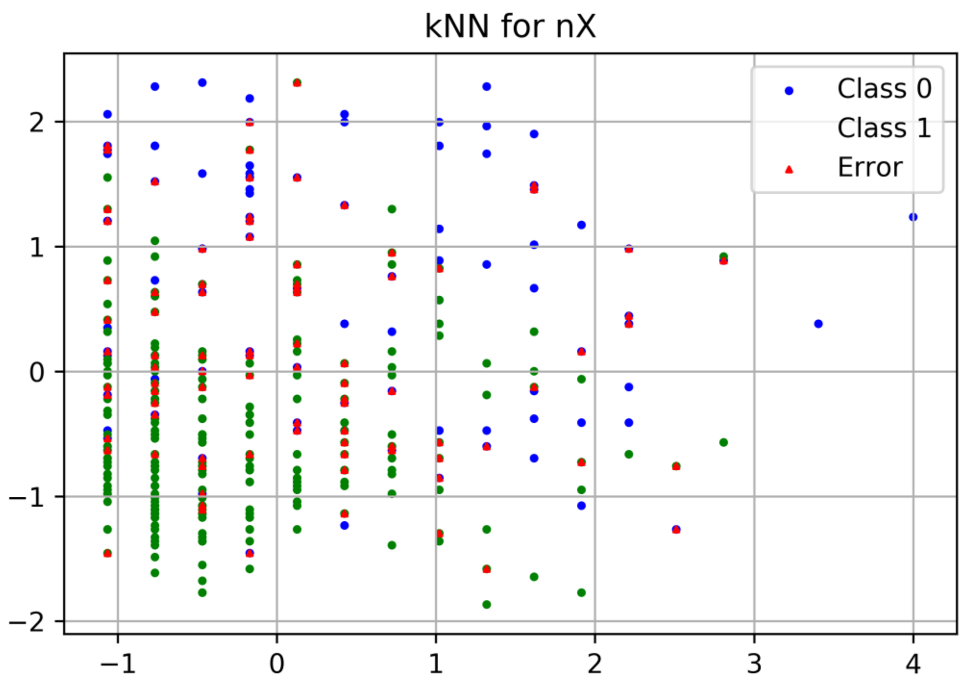
**Error Rates (TP,FP,FN,TN)**

```
def error_rates(truecol, testcol):
    dict = {'Yes': 0,'No':1}
    test = np.array(testcol.replace(dict))
    np.reshape(test,(-1,1))

    truth = np.array(truecol.replace(dict))
    truth.reshape(-1,1)

    #table = (truth == 1) & (test == 0)
   # print(table)


    #discriminator
    conditions = [(truth == 0) & (test == 0),
                  (truth == 0) & (test == 1),
                  (truth == 1) & (test == 1),
                  (truth == 1) & (test == 0)]
    choices = ['TP', 'FP', 'TN','FN']
    error_def = np.select(conditions,choices)

    #'True Positive Rate (TPR) or Sensitivity = TP / (TP + FN)'
    #'True Negative Rate (TNR) or Specificity = TN / (FP + TN)'
    #rate = error_def.groupby('error2', as_index=False).size()

    #the pandas groupby function was used to get the total counts of the errors for
each class
    labels,count = np.unique(error_def, return_counts=True)
    df_rates = pd.DataFrame(count,index=labels)
    df_rates_t = df_rates.T

    print(df_rates_t)
    df_rates_t['TPR'] = (df_rates_t['TP'])/(df_rates_t['TP'] + df_rates_t['FN'])
    df_rates_t['FPR']  = (df_rates_t['FP'])/(df_rates_t['FP'] + df_rates_t['TN'])
    df_rates_t['Acc']  =  (df_rates_t['TN'] +
df_rates_t['TP'])/(df_rates_t['TN']+df_rates_t['TP']+df_rates_t['FN']+
df_rates_t['FP'])

    #print(df_rates_t)

    return df_rates_t
```

## Iterative processing

```
knn_roc_data_nX = pd.DataFrame(columns = ['k','TP', 'FP', 'TN','FN',
'TPR','FPR','Acc'])

for i in np.arange(1,10):
    nX_data_kNN = Kens_kNNnew(df_training=pimatr_nX,class_column='diabetes',
df_testing=pimate_nX, k=i)
    df_errors_types = error_rates(truecol=nX_data_kNN['diabetes'],
testcol=nX_data_kNN['kNN_class'])
    knn_roc_data_nX = knn_roc_data_nX.append(df_errors_types,
                        ignore_index=True, # ignores index labels
                        sort=True)
    knn_roc_data_nX['k'].iloc[i-1] = i
print(knn_roc_data_nX.sort_values(by='TPR'))
#def ROC_curve(df_train, df_test):
knn_roc_data_nX.to_csv('knn_roc_data_nX.csv')
```


kNN for pX
Class 0
Class 1
Error
2
0
−2
−4
−6
−3
−2
−1
0
1
2
3
4


kNN for nX
Class 0
Class 1
Error
2
1
0
−1
−2
−1
0
1
2
3
4