Project 2 - Non-parametric Classification with Dimensionality Reduction

Adrian Cross

November 25, 2019

across19@vols.utk.edu
Machine learning
COSC 522

1 Abstract

In this project a data-set of diabetes diagnoses with varying biological factors has been investigated. The data has first been normalized and then put through various dimensional reduction techniques consisting of principle component analysis (PCA) and Fisher's linear discriminant (FLD). The results of these dimensional reduction techniques has been compared by several different factors, such as accuracy of classification. The reduced data is then put through a non-parametric (kNN) classifier which is then compared with several parametric classifiers. By performing this analysis a clear difference in both accuracy and computing performance is perceived.

2 Introduction

In this project the data inputted contains several informational features on the human body which were taken as shown in figure 1. The number of dimensions in this data is seven, which will be reduced during further analysis. It also contains whether the person has or doesn't have diabetes which defines class 0, does not have diabetes, and class 1, does have diabetes.

npreg	glu	bp	skin	bmi	ped	age	type
5	86	68	28	30.2	0.364	24	No
7	195	70	33	25.1	0.163	55	Yes
5	77	82	41	35.8	0.156	35	No
0	165	76	43	47.9	0.259	26	No
0	107	60	25	26.4	0.133	23	No
5	97	76	27	35.6	0.378	52	Yes
3	83	58	31	34.3	0.336	25	No
1	193	50	16	25.9	0.655	24	No
3	142	80	15	32.4	0.200	63	No
2	128	78	37	43.3	1.224	31	Yes
0	137	40	35	43.1	2.288	33	Yes

Figure 1: Raw input data for analysis

This data has been processed by normalization first by taking away each features mean value, and dividing by the their standard deviation, as shown in equation 1. The mean and standard deviation value has been determined from the training set and applied to both the training and testing dataset.

$$X_{normalized} = \frac{X - \mu}{\sigma} \tag{1}$$

Where X is the datapoint, μ is the mean and σ is the standard deviation. The class features are then altered from True and False to 1 and 0 for easy analysis. After normalization the normalized data has been put through several pre-processing techniques outlined below, then classifying each data point into different classes using reduced features.

The prior probabilities have been calculated from this normalized dataset. It is assumed that the number of each class in the training dataset is proportional to the prior probability. This assumption is essentially saying that the training sample is a random dataset so that the prior probabilities are determined by equation 2.

$$p_0 = \frac{n_0}{n_0 + n_1} \quad p_1 = \frac{n_1}{n_0 + n_1} \tag{2}$$

Where n_1 and n_0 are the frequency of class 0 and class 1 appearing in the training dataset.

3 Technical approach

3.1 Principal component analysis (PCA)

PCA is an unsupervised statistical method used to reduce the dimensionality of a correlated dataset. This is done by reducing a set of correlated variables into a set of linearly uncorrelated variables called principal components. Each of these principal components has the largest possible variance, therefore retaining the most information from the original dataset. The components with the smallest variance are dropped in order to reduce dimensionality. The trade off for less dimensions remaining is the loss of more information.

This is first done by calculating the covariance matrix for the training dataset and then finding its eigenvalues and eigenvectors $\lambda_1, \lambda_2 ... \lambda_n$which are associated with each of the original features $f_1, f_2 ... f_n$. These eigenvalues are then sorted from largest to smallest from which the smallest ones can be dropped, depending on how much data is required to be retained. When these eigenvalues are dropped the corresponding eigenvectors are also dropped. The matrix of eigenvectors are multiplied by the original dataset, which projects them into the new set of principle components.

3.2 Fisher's linear discriminant (FLD)

FLD is a dimensionality reducing method which uses the differences in means and covariance matrices between two classes on a training set of data to find the dimensionality reducing technique which best retains the separation between the two class distributions. This is done by finding a vector which makes the projected means as far apart as possible. This vector is also weighted by the intraclass separation, which is the separation between points inside of a class calculated from the covariance matrix, shown in equation 3

$$\overrightarrow{S_w} = (n_0 - 1)\overrightarrow{\Sigma_0} + (n_1 - 1)\overrightarrow{\Sigma_1} \tag{3}$$

Where n_0,n_1 are the number of points in each class and Σ_0,Σ_1 are the covariance matrices for each class. Using this with the mean values allows a projection of the data, as shown in equation 4.

$$\overrightarrow{w} = \overrightarrow{S_w^{-1}}(\overrightarrow{\mu_0} - \overrightarrow{\mu_1}) \tag{4}$$

Where $\overrightarrow{w}$ is the projection matrix and $\overrightarrow{\mu_0}$,$\overrightarrow{\mu_1}$ are the mean values. Multiplying the data by $\overrightarrow{w}$ projects the testing data to a reduced dimension.

3.3 Closest near neighbour approach (kNN)

kNN stands for the closest nearest neighbour, non parametric approach to classifying data into classes. This is simply done by counting what training points are closest to the point being tested. If more training points close to the testing point belong to a specific class then that testing point also belongs to that class. The k number of points which are counted are determined by the k value inputted. Therefore if $k = 1$ you count the training data point closest to the testing point and consider the testing point to also be part of the same class, as shown in figure 2.

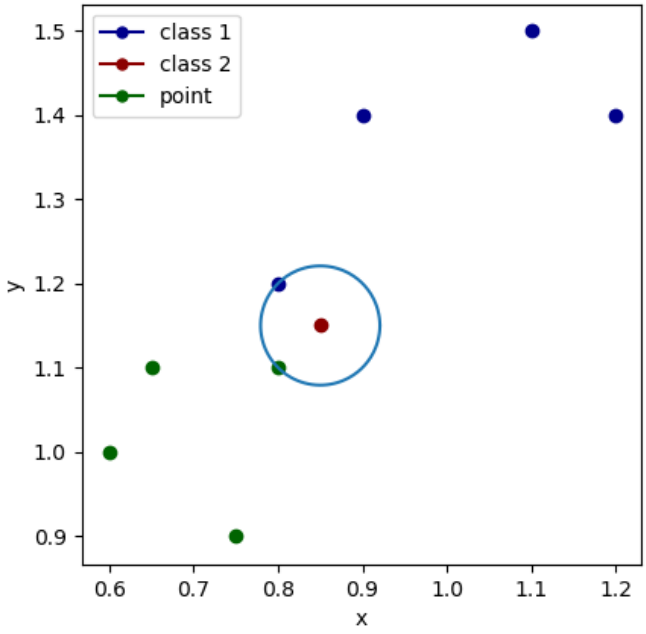

Figure 2: Graph showing a kNN approach to classifying testing points with a value of k=1

The number of data points has been normalized by the prior probability derived in equation 1.

4 Results

4.1 Principle component analysis

PCA has been performed on the raw input. As stipulated in the original problem the analysis has been performed so that 90% of the original data has been retained. figure 3 shows how the data retention rate for data changes with the number of dimensions.

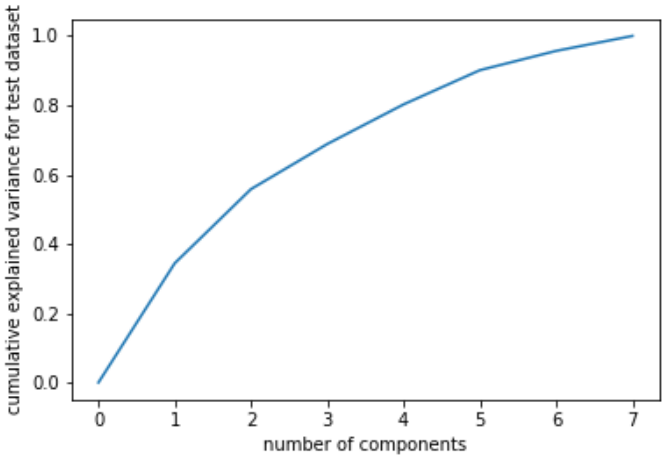

Figure 3: Graph showing how the percentage of original data retention changes depending on the number of dimensions the data has been reduced to.

From this graph it was found that to retain the 90% of information required the minimum amount of dimensions the data needs to be reduced to is five. These reduced dimensions have been analyzed using the kNN technique using a varying amount of k values, which is shown in section 4.3.

4.2 Fisher's linear discriminant

Using Fisher's linear discriminant the number of dimensions has been reduced from seven, in the original normalized dataset, down to one. Figure 4 shows how the two classes are separated in the new, single component system.

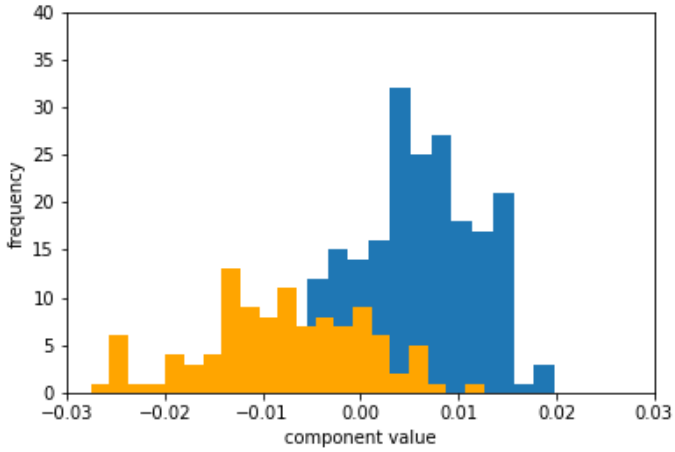

Figure 4: Graph showing how the normalized dataset has been reduced from seven dimensions down to one dimension.

4.3 kNN

Using the above method three different datasets have been produced from the original dataset. The first, Xn is simply the original data normalized via the methods outlined in 2. The second, pX, is the normalized dataset after it has been put through PCA with five features. The third and final dataset,fX, is the normalized dataset after it has been put through FLD with one feature. figure 5 shows how the k value alters the accuracy for these three different datasets.

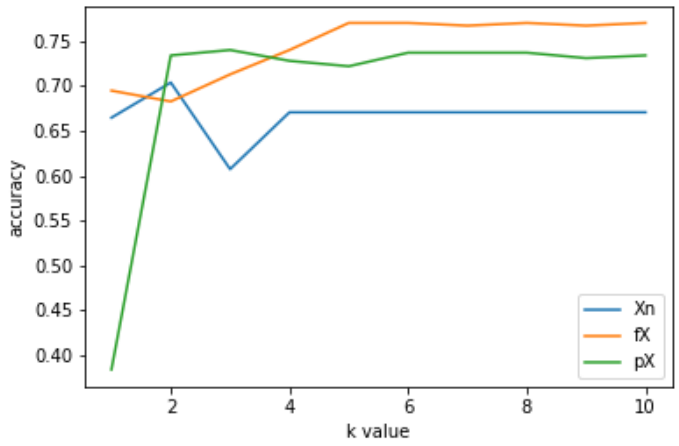

Figure 5: Graph showing how the accuracy of the classification of data changes depending the value of k in the kNN technique for various reduced datasets. Xn is the reduced dataset, pX is from PCA, fX is from FLD.

From this graph it can be seen that the optimal accuracy is reached at approximately $k = 5$. figure 6,7 and 8 show the variation with different k values for precision, recall and specificity. These values are derived from true

positive, true negative, false positive and false negative values. These value calculations are shown in equation 5.

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \quad precision = \frac{TP}{TP + FP} \quad Recall = \frac{TP}{TP + FN} \quad Specificity = \frac{FP}{TN + FP} \tag{5}$$

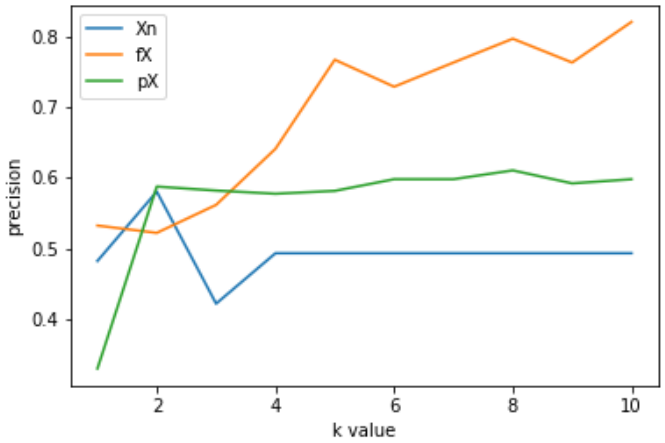

Figure 6: Graph showing how the precision of the classification of data changes depending the value of k in the kNN technique for various reduced datasets. Xn is the reduced dataset, pX is from PCA, fX is from FLD.

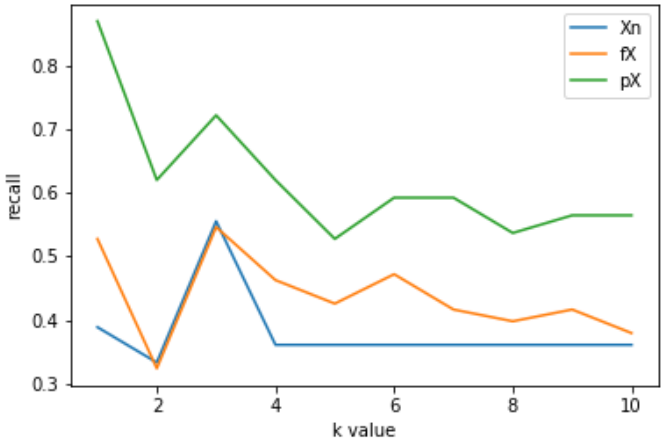

Figure 7: Graph showing how the recall of the classification of data changes depending the value of k in the kNN technique for various reduced datasets. Xn is the reduced dataset, pX is from PCA, fX is from FLD.

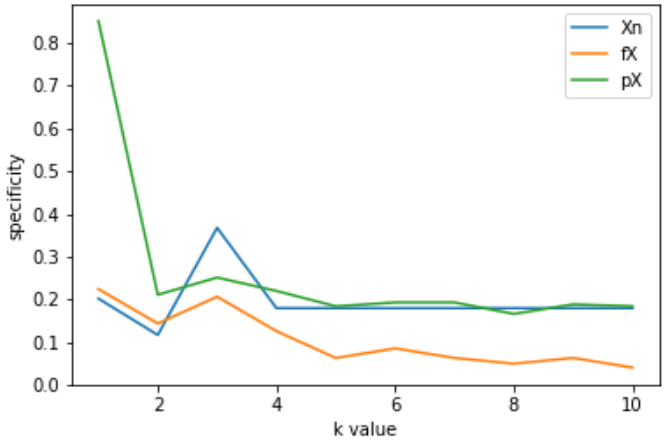

Figure 8: Graph showing how the recall of the classification of data changes depending the value of k in the kNN technique for various reduced datasets. Xn is the reduced dataset, pX is from PCA, fX is from FLD.

After varying the k values, the values for the prior were altered. This was done by multiplying the prior value for class one by a constant factor, alpha. figure 9, 10, 11, 12 show how this alpha value varies the various factors shown above.

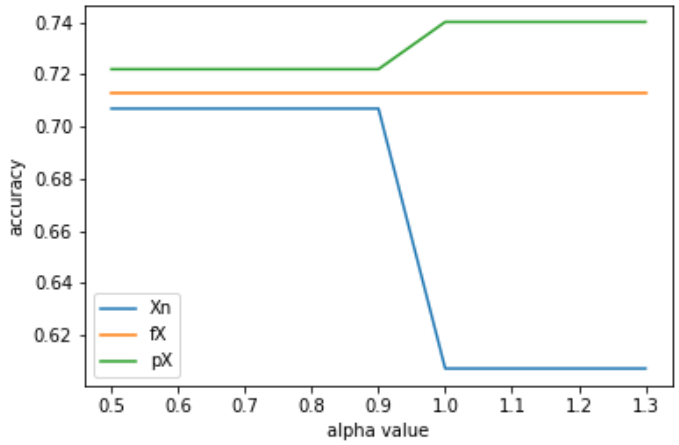

Figure 9: Graph showing how the accuracy of the classification of data changes depending the value of alpha in the kNN technique for various reduced datasets. Xn is the reduced dataset, pX is from PCA, fX is from FLD.

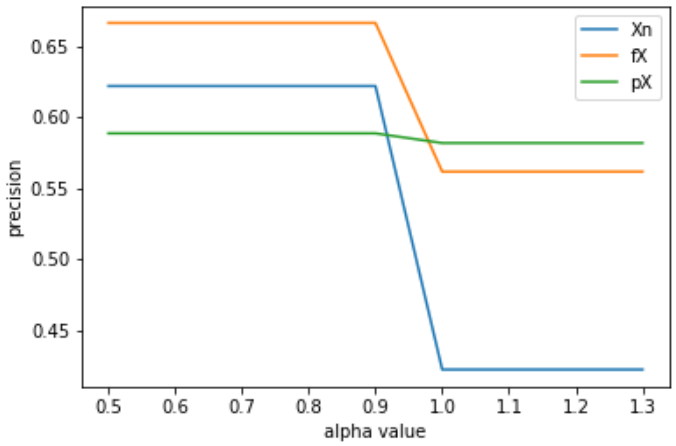

Figure 10: Graph showing how the precision of the classification of data changes depending the value of alpha in the kNN technique for various reduced datasets. Xn is the reduced dataset, pX is from PCA, fX is from FLD.

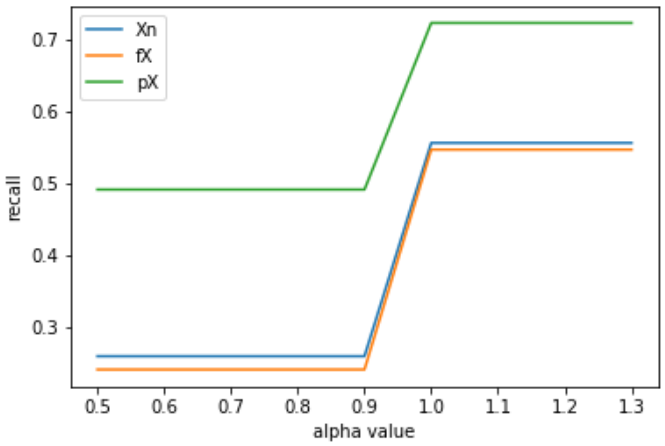

Figure 11: Graph showing how the recall of the classification of data changes depending the value of alpha in the kNN technique for various reduced datasets. Xn is the reduced dataset, pX is from PCA, fX is from FLD.

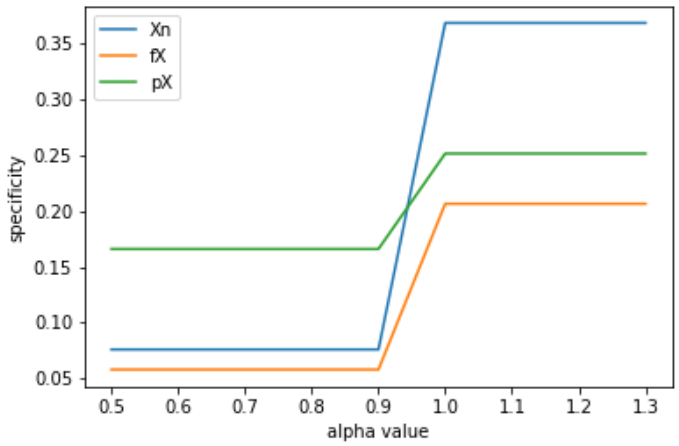

Figure 12: Graph showing how the recall of the classification of data changes depending the value of alpha in the kNN technique for various reduced datasets. Xn is the reduced dataset, pX is from PCA, fX is from FLD.

The final part that was varied was the number of dimensions for the PCA analysis. in this case the number of dimensions used was varied and the corresponding graphs for specificity, recall, precision and accuracy were outputted. From these graphs a clear increase in data accuracy can be seen with increased number of dimensions.

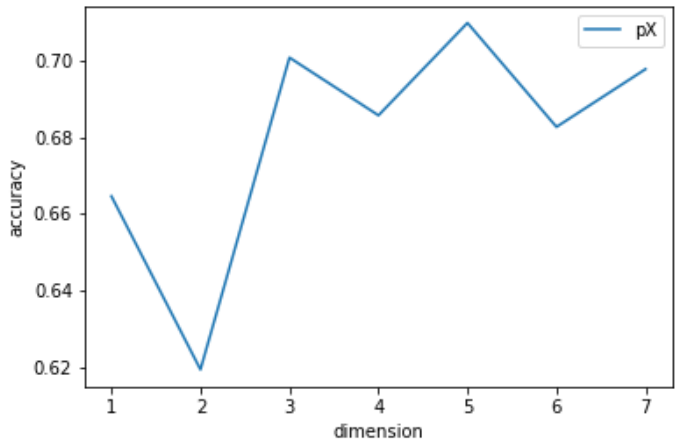

Figure 13: Graph showing how the accuracy of the classification of data changes depending the dimensions in the kNN technique for various reduced datasets. Xn is the reduced dataset, pX is from PCA, fX is from FLD.

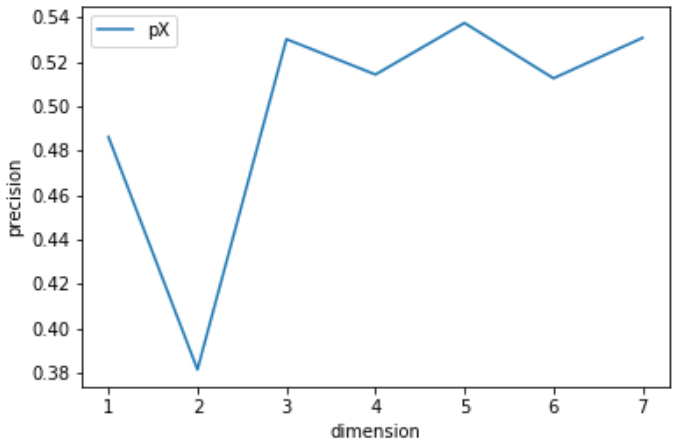

Figure 14: Graph showing how the precision of the classification of data changes depending the dimensions in the kNN technique for various reduced datasets. Xn is the reduced dataset, pX is from PCA, fX is from FLD.

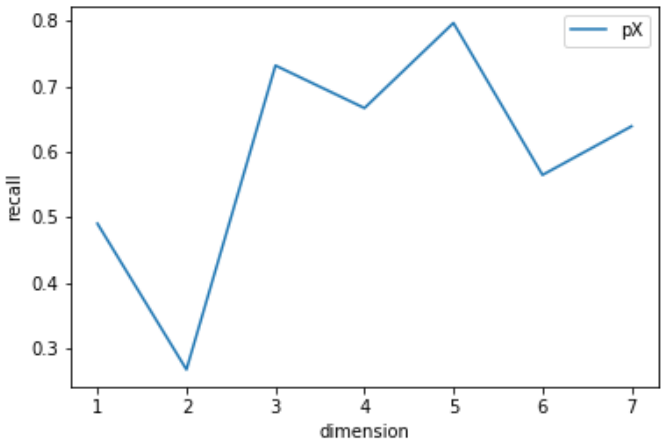

Figure 15: Graph showing how the recall of the classification of data changes depending the dimensions in the kNN technique for various reduced datasets. Xn is the reduced dataset, pX is from PCA, fX is from FLD.

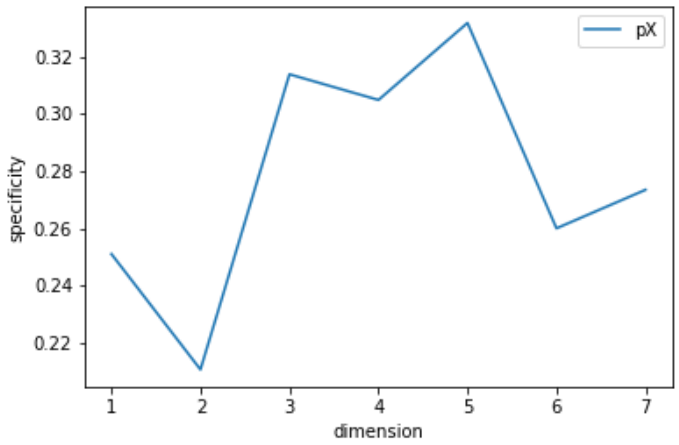

Figure 16: Graph showing how the recall of the classification of data changes depending the dimensions in the kNN technique for various reduced datasets. Xn is the reduced dataset, pX is from PCA, fX is from FLD.

This implementation also includes a measure of the time it takes for processing the data. For each k value the time taken is approximately the same, $1.5s$ for each dataset. For larger datasets the difference in dimension size would have a much larger affect on the processing time, so that reducing the dimensions would have a clearer affect however in this case the difference in processing time is small.

5 Conclusion

In this report a set of training data has been used to reduce the dimension on a set of testing data. The data has been first normalized in order to make the various features of the data comparable. In one case reducing the testing data from seven dimensions to one dimension via FLD. Another method used reduced the testing data from seven dimension down to five dimensions while retaining 90% of the information in the data set via PCA. By comparing the normalized data, the FLD data and the PCA data we can see that varying the k values, on the kNN approach to classifying data, has a different affect on the recall, precision, accuracy and specificity. These three data sets were also varied via varying the prior probabilities, using the alpha variable. From this it can again be seen that the variation has a different affect on the four factors for the three different states.
Future studies of this nature could include a more in depth analysis of this variation. By varying both the alpha value and the k value a more complete picture could be found. Other dimension reduction techniques could also be investigated, in this analysis FLD was only used to 90% accuracy however the different information retention could be investigated to see how this affects these factors.

A Code

A.1 Preprocessing

This code is used to perform normalization, FLD and PCA on the data and output the data.

```
#File is used to normalize, perform PCA and FLD for dimensionality reduction
import sys
import math
import timeit
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.lines as mlines
from sklearn.decomposition import PCA
pi= 3.14159265359

def input_data(filename):
    data = pd.read_csv(filename, sep='\s+', header = 0)
    return data
```

```
15
16 def normalize(data,mean,sigma): #normalizes the dataset based on information in task 1
17     data=(data-mean)/sigma
18     return data
19
20 def PCA_dimensions(data): #calculates the number of dimensions required to retain different
       variance amounts
21     cov_tr=np.cov(Xn_tr,rowvar=False)#calculates covariance matrix
22     eig_val_tr, eig_vec_tr=np.linalg.eig(cov_tr) #calculates eigenvalues and vectors of
       covariance matrix
23     eig_val_tr=np.abs(eig_val_tr)
24     print('Eigenvectors \n' ,eig_vec_tr)
25     print('\nEigenvalues \n' ,eig_val_tr)
26
27 #calculates the error after removing the smallest eigenvalue from the eigenvalues-> loops
       until no eigenvalues are left
28     eig_val_sort=sorted(eig_val_tr)
29     dim=7
30     var_ret_arr=np.zeros((8,2))
31     while len(eig_val_sort)>0:
32         var_ret=sum(eig_val_sort)/sum(eig_val_tr)#calculates the retained variance
33         var_ret_arr[dim][0]=dim
34         var_ret_arr[dim][1]=var_ret
35         print(dim, ' dimensions retains a variance of ', var_ret)
36         dim=dim-1
37         eig_val_sort.remove(min(eig_val_sort))
38     return var_ret_arr
39
40 def PCA_projection(data):#returns the projection matrix from inputted data
41     cov_tr=np.cov(Xn_tr,rowvar=False)#calculates covariance matrix
42     eig_val_tr, eig_vec_tr=np.linalg.eig(cov_tr) #calculates eigenvalues and vectors of
       covariance matrix
43     eig_pairs_tr = [((eig_val_tr[i]), eig_vec_tr[:,i]) for i in range(len(eig_val_tr))]# Make
       a list of eigenvalue, eigenvector tuples
44     eig_pairs_tr.sort(reverse=True) #sorts the typles so that the largest eigenvalue is first
45     projection_matrix=np.hstack((eig_pairs_tr[0][1].reshape(7,1),#projection matrix only
       counting the first 5 eigenvalues for 5 dimensions
46                             eig_pairs_tr[1][1].reshape(7,1),
47                             eig_pairs_tr[2][1].reshape(7,1),
48                             eig_pairs_tr[3][1].reshape(7,1),
49                             eig_pairs_tr[4][1].reshape(7,1)))
50     return projection_matrix
51
52 def FLD_intraclass(class_0, class_1):
53     S_w=(class_0.shape[0]-1)*np.cov(class_0,rowvar=False)+(class_1.shape[0]-1)*np.cov(class_1,
       rowvar=False)
54     return S_w
55
56 def FLD_projection(data,W):
57     data_t=np.transpose(data)
58     W_t=np.transpose(W)
59     projected_data=W_t@data_t
60     return projected_data
61
62 #normalization
63 X_tr=input_data('pima.tr')
64 X_te=input_data('pima.te')
65 X_tr['type']=X_tr['type'].map({'Yes':1, 'No':0}) #maps the new data type
66 X_te['type']=X_te['type'].map({'Yes':1, 'No':0})
67 X_tr.drop(X_tr.index[0],inplace=True) #drops the first row
68 X_te.drop(X_te.index[0],inplace=True)
69 X_class_tr=X_tr['type']#saves the type column
70 X_class_te=X_te['type']
71 X_tr.drop(['type'],axis=1,inplace=True) #drops the type column
72 X_te.drop(['type'],axis=1,inplace=True)
73 X_mean=X_tr.mean()
74 X_sigma=X_tr.std()
75 Xn_tr=normalize(X_tr, X_mean, X_sigma)#normalizes using the training data set and drops the
       class labels
76 Xn_te=normalize(X_te, X_mean, X_sigma)
77
78 #PCA
```

```
dim_tr=PCA_dimensions(Xn_tr) # returns an array containing the retained variance for number of
    dimensions
plt.figure()
plt.plot(dim_tr[:,1])
plt.xlabel('number of components')
plt.ylabel('cumulative explained variance for test dataset');
plt.savefig('cum_var.png')

projection_matrix=PCA_projection(Xn_tr)#projection matrix is defined by tr and then used on te
pX_tr=Xn_tr.dot(projection_matrix)
pX_te=Xn_te.dot(projection_matrix)

#pca=PCA(5)
#pX_tr=pca.fit_transform(Xn_tr)#performs PCA 7->5 dimensions
#pX_te=pca.fit_transform(Xn_te)

#pX_tr = pd.DataFrame(pX_tr)#converts from np array to pandas df
#pX_te = pd.DataFrame(pX_te)
pX_tr=pd.concat([pX_tr,X_class_tr],axis=1)#read the classes back in
pX_te=pd.concat([pX_te,X_class_te],axis=1)
pX_tr.to_csv(r'/home/across/UTK_PhD/Machine_learning_fall_2019/project_2/pX.tr',index=False)
pX_te.to_csv(r'/home/across/UTK_PhD/Machine_learning_fall_2019/project_2/pX.te',index=False)

#FLD
fX_tr=pd.concat([Xn_tr,X_class_tr],axis=1)#read the classes as we need them for FLD
fX_te=pd.concat([Xn_te,X_class_te],axis=1)
fX_tr_c0=fX_tr.loc[fX_tr.type==0]#split into different classes
fX_tr_c1=fX_tr.loc[fX_tr.type==1]
fX_tr_c0.drop(['type'],axis=1,inplace=True) #drops the type column
fX_tr_c1.drop(['type'],axis=1,inplace=True)
fX_te_c0=fX_te.loc[fX_te.type==0]#split into different classes
fX_te_c1=fX_te.loc[fX_te.type==1]
fX_te_c0.drop(['type'],axis=1,inplace=True) #drops the type column
fX_te_c1.drop(['type'],axis=1,inplace=True)

#calculates the W matrix required for projection from the training data
mean=fX_tr_c0.mean()-fX_tr_c1.mean()
W=np.linalg.inv(FLD_intraclass(fX_tr_c0,fX_tr_c1))@mean
#projects the new datasets for each class
fX_tr_c0=FLD_projection(fX_tr_c0,W)
fX_tr_c1=FLD_projection(fX_tr_c1,W)
fX_te_c0=FLD_projection(fX_te_c0,W)
fX_te_c1=FLD_projection(fX_te_c1,W)
plt.figure()
plt.hist(fX_te_c0,bins=25)
plt.xlabel('component value')
plt.ylabel('frequency');
plt.xlim([-.030,.030])
plt.ylim([0,40])
plt.hist(fX_te_c1,bins=25,color='orange')
plt.savefig('FLD_freq_combined.png')
plt.figure()
plt.hist(fX_te_c0,bins=25)
plt.xlabel('component value')
plt.ylabel('frequency');
plt.xlim([-.030,.020])
plt.ylim([0,40])
plt.savefig('FLD_freq_c0.png')
plt.figure()
plt.hist(fX_te_c1,bins=25,color='orange')
plt.xlabel('component value')
plt.ylabel('frequency');
plt.xlim([-.030,.030])
plt.ylim([0,40])
plt.savefig('FLD_freq_c1.png')

fX_tr_c0 = pd.DataFrame(fX_tr_c0)
fX_tr_c0.insert( 1,"type", 0)
fX_tr_c1 = pd.DataFrame(fX_tr_c1)
fX_tr_c1.insert( 1,"type", 1)
fX_te_c0 = pd.DataFrame(fX_te_c0)
fX_te_c0.insert( 1,"type", 0)
fX_te_c1 = pd.DataFrame(fX_te_c1)
```

```
fX_te_c1.insert( 1,"type", 1)


fX_tr=pd.concat([fX_tr_c0,fX_tr_c1],axis=0)
fX_te=pd.concat([fX_te_c0,fX_te_c1],axis=0)
fX_tr.to_csv(r'/home/across/UTK_PhD/Machine_learning_fall_2019/project_2/fX.tr',index=False)
fX_te.to_csv(r'/home/across/UTK_PhD/Machine_learning_fall_2019/project_2/fX.te',index=False)


Xn_tr=pd.concat([Xn_tr,X_class_tr],axis=1)#read the classes back in
Xn_te=pd.concat([Xn_te,X_class_te],axis=1)
Xn_tr.to_csv(r'/home/across/UTK_PhD/Machine_learning_fall_2019/project_2/Xn.tr',index=False)
Xn_te.to_csv(r'/home/across/UTK_PhD/Machine_learning_fall_2019/project_2/Xn.te',index=False)
```

A.2 Processing

This code is used to process the data using kNN and compare the results of the three different preprocessed data types

```

import sys
import math
import timeit
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.lines as mlines
import timeit
import argparse
import mpp
from sklearn.decomposition import PCA
pi= 3.14159265359

def input_data(filename):
    data = pd.read_csv(filename, sep=',', header = 0)
    return data

def kNN_class_sin(i,data_tr, data_te, k,prior_p_c0, prior_p_c1, alpha): #reads in training and
     testing data set, k value, prior probabilities and alpha to ouput a class for 1 dp
    eu_dist=eucl_dist(i,data_tr, data_te, k,prior_p_c0, prior_p_c1, alpha)
    data_sort=pd.concat([eu_dist,data_tr['type']],axis=1)    #assigns a class to each distance
    data_sort=data_sort.sort_values(by=[0],ascending=True) #sorts by closest data points
    k_data_points=data_sort.head(k)  #gets the first 5 closest data points
    n_0=(k_data_points.iloc[:,1]==0).sum()
    n_1=(k_data_points.iloc[:,1]==1).sum()
    M=alpha*prior_p_c1/prior_p_c0 #factor by which number of p1 points which lie in a k radius
     is multiplied by
    Mn_1=M*n_1
    if (Mn_1>n_0):
        return 1
    elif (Mn_1<n_0):
        return 0
    elif (Mn_1==n_0):
        print(n_0)
        print(n_1)
        print("error in kNN, values are equal")

def kNN_class(data_tr, data_te, k, alpha): #reads in a data frame and appends the class which
    kNN identifies it belongs to in type_kNN column
    start = timeit.default_timer()
    c0_count=(data_tr.type==0).sum()    #prior calculation
    c1_count=(data_tr.type==1).sum()
    c0_pr=c0_count/(c1_count+c0_count)
    c1_pr=c1_count/(c1_count+c0_count)
    data_updated=data_te.copy()
    data_updated['type_'+str(k)+'NN']=2
    for i in range(0,data_te.shape[0]): #loops over all points in test file
        data_updated.loc[i,'type_'+str(k)+'NN']=kNN_class_sin(i,data_tr, data_te, k,c0_pr,
    c1_pr, alpha)
    stop = timeit.default_timer()
    print('time to execute for k=',k,':', stop - start,'s')
    return data_updated
```

```
50
51 def kNN_class_alpha(data_tr, data_te, k, alpha): #reads in a data frame and appends the class
       which kNN identifies it belongs to in type_kNN column
52     start = timeit.default_timer()
53     c0_count=(data_tr.type==0).sum()    #prior calculation
54     c1_count=(data_tr.type==1).sum()
55     c0_pr=c0_count/(c1_count+c0_count)
56     c1_pr=c1_count/(c1_count+c0_count)
57     data_updated=data_te.copy()
58     data_updated['alpha='+str(alpha)]=2
59     for i in range(0,data_te.shape[0]): #loops over all points in test file
60         data_updated.loc[i,'alpha='+str(alpha)]=kNN_class_sin(i,data_tr, data_te, k,c0_pr,
       c1_pr, alpha)
61     stop = timeit.default_timer()
62     print('time to execute for k=',k,':', stop - start,'s')
63     return data_updated
64
65 def kNN_class_dim(data_tr, data_te, k, alpha,dim): #reads in a data frame and appends the
       class which kNN identifies it belongs to in type_kNN column
66     start = timeit.default_timer()
67     c0_count=(data_tr.type==0).sum()    #prior calculation
68     c1_count=(data_tr.type==1).sum()
69     c0_pr=c0_count/(c1_count+c0_count)
70     c1_pr=c1_count/(c1_count+c0_count)
71     data_updated=data_te.copy()
72     data_updated['dim='+str(dim)]=2
73     for i in range(0,data_te.shape[0]): #loops over all points in test file
74         data_updated.loc[i,'dim='+str(dim)]=kNN_class_sin(i,data_tr, data_te, k,c0_pr, c1_pr,
       alpha)
75     stop = timeit.default_timer()
76     print('time to execute for k=',k,':', stop - start,'s')
77     return data_updated
78
79
80
81 def eu_dist_mean(mean_df,df):
82     eu_dist=1
83     return eu_dist
84
85 def case_1(data_tr, data_te,alpha): #takes data and prior scaling returning what class each
       point belongs to based on case used in project 1
86     c0_count=(data_tr.type==0).sum()    #prior calculation
87     c1_count=(data_tr.type==1).sum()
88     c0_pr=c0_count/(c1_count+c0_count)
89     c1_pr=c1_count/(c1_count+c0_count)
90     return data_c1
91
92 def case_2(data_tr, data_te,alpha):
93     return data_c2
94
95 def case_3(data_tr, data_te,alpha):
96     return data_c3
97
98 #takes in data and returns and returns the TP/TN/FP/FN rate for each different column (kNN,
       case 1 etc.)
99 #Positive is having the disease, negative is not having the disease
100 def TP(data_te,groupname): #has disease and is diagnosed as such (1->1)
101     grouped_count=data_te.groupby(['type',groupname]).size()
102     TP=grouped_count.iloc[3]
103     return TP
104
105 def TN(data_te,groupname): #does not have disease and is diagnosed as such (0->0)
106     grouped_count=data_te.groupby(['type',groupname]).size()
107     TN=grouped_count.iloc[0]
108     return TN
109
110 def FP(data_te,groupname): #does not have disease but is diagnosed as having did (0->1)
111     grouped_count=data_te.groupby(['type',groupname]).size()
112     FP=grouped_count.iloc[1]
113     return FP
114
115 def FN(data_te,groupname): #does have disease but is diagnosed as not having it (1->0)
116     grouped_count=data_te.groupby(['type',groupname]).size()
```

```
    FN=grouped_count.iloc[2]
    return FN

def k_vary(data_tr,data_te,label):
    #k varying
    k_range=10 #square root of n for training data
    k_output=np.zeros((k_range,9))
    k_output=pd.DataFrame(k_output,columns=['k','TP','TN','FP','FN','accuracy','precision','recall','specificity'])
    print("calculating kNN for ",label)
    for k in range(1,k_range+1): #performs kNN for k=1 up to k=50 including prior probability calculated from training data (not varied using alpha) and outputs accuracy
        k_string='type_'+str(k)+'NN'
        data_tmp=kNN_class(data_tr, data_te, k, 5) #returns the class each points belong to based on k input (priors are calculated in this function)
        k_output.loc[k-1][0]=k
        k_output.loc[k-1][1]=TP(data_tmp,k_string)
        k_output.loc[k-1][2]=TN(data_tmp,k_string)
        k_output.loc[k-1][3]=FP(data_tmp,k_string)
        k_output.loc[k-1][4]=FN(data_tmp,k_string)
    k_output.loc[:,'accuracy']=(k_output.loc[:,'TP']+k_output.loc[:,'TN'])/(k_output.loc[:,'FP']+k_output.loc[:,'FN']+k_output.loc[:,'TP']+k_output.loc[:,'TN']) #accuracy calculation
    k_output.loc[:,'precision']=(k_output.loc[:,'TP'])/(k_output.loc[:,'FP']+k_output.loc[:,'TP'])
    k_output.loc[:,'recall']=(k_output.loc[:,'TP'])/(k_output.loc[:,'FN']+k_output.loc[:,'TP'])
    k_output.loc[:,'specificity']=(k_output.loc[:,'FP'])/(k_output.loc[:,'FP']+k_output.loc[:,'TN'])


    del(data_tmp)
    return k_output

def classifier_comp(data_tr,data_te,k):
    k_string='type_'+str(k)+'NN'
    C_comp=np.zeros((4,4))
    C_comp=pd.DataFrame(C_comp,columns=['TP','TN','FP','FN'])
    k_string='type_'+str(k)+'NN'
    data_tmp=kNN_class(data_tr, data_te, k, 1) #returns the class each points belong to based on k input (priors are calculated in this function)
    C_comp.loc[0,'TP']=TP(data_tmp.copy(),k_string)
    C_comp.loc[0,'TN']=TN(data_tmp.copy(),k_string)
    C_comp.loc[0,'FP']=FP(data_tmp.copy(),k_string)
    C_comp.loc[0,'FN']=FN(data_tmp.copy(),k_string)
    del(data_tmp)
    C_comp=np.divide(C_comp,data_te.shape[0]) #normalizes TP,TN,FP,FN to total number of data points
    classifier=['kNN','C1','C2','C3']
    C_comp.insert(0,'classifier',label,True)
    return C_comp

def eucl_dist(i,data_tr, data_te, k,prior_p_c0, prior_p_c1, alpha):
    data_te_point=np.zeros((data_tr.shape[0],data_tr.shape[1]-1))+data_te.loc[i][0]
    data_te_point=pd.DataFrame(data_te_point)
    data_tr_temp=pd.DataFrame(data_tr.iloc[:,0:data_tr.shape[1]-1]).copy()
    eu_dist=pd.DataFrame(data_tr_temp.to_numpy()-data_te_point.to_numpy())**2  #calculate euclidena distance between points
    eu_dist=eu_dist.sum(axis=1)
    eu_dist=np.sqrt(eu_dist)
    return eu_dist

def alpha_vary(data_tr,data_te,label):
    alpha_range=10
    alpha_output=np.zeros((alpha_range,9))
    alpha_output=pd.DataFrame(alpha_output,columns=['alpha','TP','TN','FP','FN','accuracy','precision','recall','specificity'])
    print("calculating kNN for ",label)
    for i in range(1,alpha_range):
        alpha=0.4+i*0.1
        alpha_string='alpha='+str(alpha)
        data_tmp=kNN_class_alpha(data_tr, data_te, 3, alpha)
        alpha_output.loc[i-1][0]=alpha
        alpha_output.loc[i-1][1]=TP(data_tmp,alpha_string)
```

```
        alpha_output.loc[i-1][2]=TN(data_tmp,alpha_string)
        alpha_output.loc[i-1][3]=FP(data_tmp,alpha_string)
        alpha_output.loc[i-1][4]=FN(data_tmp,alpha_string)
    alpha_output.loc[:,'accuracy']=(alpha_output.loc[:,'TP']+alpha_output.loc[:,'TN'])/(alpha_output.loc[:,'FP']+alpha_output.loc[:,'FN']+alpha_output.loc[:,'TP']+alpha_output.loc[:,'TN']) #accuracy calculation
    alpha_output.loc[:,'precision']=(alpha_output.loc[:,'TP'])/(alpha_output.loc[:,'FP']+alpha_output.loc[:,'TP'])
    alpha_output.loc[:,'recall']=(alpha_output.loc[:,'TP'])/(alpha_output.loc[:,'FN']+alpha_output.loc[:,'TP'])
    alpha_output.loc[:,'specificity']=(alpha_output.loc[:,'FP'])/(alpha_output.loc[:,'FP']+alpha_output.loc[:,'TN'])
    return alpha_output

def dimension_vary(data_tr,data_te,ndim):
    pca=PCA(ndim)
    data_tr_tmp=pca.fit_transform(data_tr)#performs PCA 7->5 dimensions
    data_te_tmp=pca.fit_transform(data_te)
    data_tr_tmp = pd.DataFrame(data_tr_tmp)#converts from np array to pandas df
    data_te_tmp = pd.DataFrame(data_te_tmp)
    return data_te_tmp

def dimension_vary_overall(data_tr,data_te):
    ndim_range=8
    class_tr=data_tr['type']#saves the type column
    class_te=data_te['type']
    data_tr.drop(['type'],axis=1,inplace=True) #drops the type column
    data_te.drop(['type'],axis=1,inplace=True)
    dim_output=np.zeros((ndim_range-1,9))
    dim_output=pd.DataFrame(dim_output,columns=['dim','TP','TN','FP','FN','accuracy','precision','recall','specificity'])
    for i in range(1,ndim_range):
        dim_string='dim='+str(i)
        data_tr_tmp=dimension_vary(data_tr.copy(),data_tr.copy(),i)
        data_te_tmp=dimension_vary(data_tr.copy(),data_te.copy(),i)
        data_tr_tmp=pd.concat([data_tr_tmp,class_tr.copy()],axis=1)#read the classes back in
        data_te_tmp=pd.concat([data_te_tmp,class_te.copy()],axis=1)
        data_tmp=kNN_class_dim(data_tr_tmp,data_te_tmp,3,1,i)
        dim_output.loc[i-1][0]=i
        dim_output.loc[i-1][1]=TP(data_tmp,dim_string)
        dim_output.loc[i-1][2]=TN(data_tmp,dim_string)
        dim_output.loc[i-1][3]=FP(data_tmp,dim_string)
        dim_output.loc[i-1][4]=FN(data_tmp,dim_string)
    dim_output.loc[:,'accuracy']=(dim_output.loc[:,'TP']+dim_output.loc[:,'TN'])/(dim_output.loc[:,'FP']+dim_output.loc[:,'FN']+dim_output.loc[:,'TP']+dim_output.loc[:,'TN']) #accuracy calculation
    dim_output.loc[:,'precision']=(dim_output.loc[:,'TP'])/(dim_output.loc[:,'FP']+dim_output.loc[:,'TP'])
    dim_output.loc[:,'recall']=(dim_output.loc[:,'TP'])/(dim_output.loc[:,'FN']+dim_output.loc[:,'TP'])
    dim_output.loc[:,'specificity']=(dim_output.loc[:,'FP'])/(dim_output.loc[:,'FP']+dim_output.loc[:,'TN'])
    print(dim_output)
    return(dim_output)




#Input data
Xn_tr=input_data('Xn.tr')#data read in
Xn_te=input_data('Xn.te')
pX_tr=input_data('pX.tr')
pX_te=input_data('pX.te')
fX_tr=input_data('fX.tr')
fX_te=input_data('fX.te')

#varying dimension
pX_dim_output=dimension_vary_overall(Xn_tr.copy(),Xn_te.copy())


#variation of prior
Xn_alpha_output=alpha_vary(Xn_tr,Xn_te,'Xn')
fX_alpha_output=alpha_vary(fX_tr,fX_te,'fX')
```

```
pX_alpha_output=alpha_vary(pX_tr,pX_te,'pX')

#Classifier_comparison (using k=3 as this has best performance)
Xn_class_comp=classifier_comp(Xn_tr,Xn_te,3)
Xn_class_comp.to_csv(r'/home/across/UTK_PhD/Machine_learning_fall_2019/project_2/Xn_class_comp
    .csv',index=False)
pX_class_comp=classifier_comp(pX_tr,pX_te,3)
pX_class_comp.to_csv(r'/home/across/UTK_PhD/Machine_learning_fall_2019/project_2/pX_class_comp
    .csv',index=False)
fX_class_comp=classifier_comp(fX_tr,fX_te,3)
fX_class_comp.to_csv(r'/home/across/UTK_PhD/Machine_learning_fall_2019/project_2/fX_class_comp
    .csv',index=False)

#k varying
Xn_k_output=k_vary(Xn_tr,Xn_te,'Xn')
fX_k_output=k_vary(fX_tr,fX_te,'fX')
pX_k_output=k_vary(pX_tr,pX_te,'pX')

plt.figure() #plots accuracy vs k value
plt.plot(Xn_k_output['k'],Xn_k_output['accuracy'],label='Xn')
plt.plot(fX_k_output['k'],fX_k_output['accuracy'],label='fX')
plt.plot(pX_k_output['k'],pX_k_output['accuracy'],label='pX')
plt.xlabel('k value')
plt.ylabel('accuracy')
plt.legend()
plt.savefig('k_var_acc.png')

plt.figure()
plt.plot(Xn_k_output['k'],Xn_k_output['precision'],label='Xn')
plt.plot(fX_k_output['k'],fX_k_output['precision'],label='fX')
plt.plot(pX_k_output['k'],pX_k_output['precision'],label='pX')
plt.xlabel('k value')
plt.ylabel('precision')
plt.legend()
plt.savefig('k_var_prec.png')

plt.figure()
plt.plot(Xn_k_output['k'],Xn_k_output['recall'],label='Xn')
plt.plot(fX_k_output['k'],fX_k_output['recall'],label='fX')
plt.plot(pX_k_output['k'],pX_k_output['recall'],label='pX')
plt.xlabel('k value')
plt.ylabel('recall')
plt.legend()
plt.savefig('k_var_rec.png')

plt.figure()
plt.plot(Xn_k_output['k'],Xn_k_output['specificity'],label='Xn')
plt.plot(fX_k_output['k'],fX_k_output['specificity'],label='fX')
plt.plot(pX_k_output['k'],pX_k_output['specificity'],label='pX')
plt.xlabel('k value')
plt.ylabel('specificity')
plt.legend()
plt.savefig('k_var_spec.png')

plt.figure() #plots accuracy vs k value
plt.plot(Xn_alpha_output['alpha'],Xn_alpha_output['accuracy'],label='Xn')
plt.plot(fX_alpha_output['alpha'],fX_alpha_output['accuracy'],label='fX')
plt.plot(pX_alpha_output['alpha'],pX_alpha_output['accuracy'],label='pX')
plt.xlabel('alpha value')
plt.ylabel('accuracy')
plt.legend()
plt.savefig('alpha_var_acc.png')

plt.figure()
plt.plot(Xn_alpha_output['alpha'],Xn_alpha_output['precision'],label='Xn')
plt.plot(fX_alpha_output['alpha'],fX_alpha_output['precision'],label='fX')
plt.plot(pX_alpha_output['alpha'],pX_alpha_output['precision'],label='pX')
plt.xlabel('alpha value')
plt.ylabel('precision')
plt.legend()
plt.savefig('alpha_var_prec.png')

plt.figure()
```

```
plt.plot(Xn_alpha_output['alpha'],Xn_alpha_output['recall'],label='Xn')
plt.plot(fX_alpha_output['alpha'],fX_alpha_output['recall'],label='fX')
plt.plot(pX_alpha_output['alpha'],pX_alpha_output['recall'],label='pX')
plt.xlabel('alpha value')
plt.ylabel('recall')
plt.legend()
plt.savefig('alpha_var_rec.png')

plt.figure()
plt.plot(Xn_alpha_output['alpha'],Xn_alpha_output['specificity'],label='Xn')
plt.plot(fX_alpha_output['alpha'],fX_alpha_output['specificity'],label='fX')
plt.plot(pX_alpha_output['alpha'],pX_alpha_output['specificity'],label='pX')
plt.xlabel('alpha value')
plt.ylabel('specificity')
plt.legend()
plt.savefig('alpha_var_spec.png')

plt.figure() #plots accuracy vs k value
#plt.plot(Xn_dim_output['dim'],Xn_dim_output['accuracy'],label='Xn')
#plt.plot(fX_dim_output['dim'],fX_dim_output['accuracy'],label='fX')
plt.plot(pX_dim_output['dim'],pX_dim_output['accuracy'],label='pX')
plt.xlabel('dimension')
plt.ylabel('accuracy')
plt.legend()
plt.savefig('dim_var_acc.png')

plt.figure()
#plt.plot(Xn_dim_output['dim'],Xn_dim_output['precision'],label='Xn')
#plt.plot(fX_dim_output['dim'],fX_dim_output['precision'],label='fX')
plt.plot(pX_dim_output['dim'],pX_dim_output['precision'],label='pX')
plt.xlabel('dimension')
plt.ylabel('precision')
plt.legend()
plt.savefig('dim_var_prec.png')

plt.figure()
#plt.plot(Xn_dim_output['dim'],Xn_dim_output['recall'],label='Xn')
#plt.plot(fX_dim_output['dim'],fX_dim_output['recall'],label='fX')
plt.plot(pX_dim_output['dim'],pX_dim_output['recall'],label='pX')
plt.xlabel('dimension')
plt.ylabel('recall')
plt.legend()
plt.savefig('dim_var_rec.png')

plt.figure()
#plt.plot(Xn_dim_output['dim'],Xn_dim_output['specificity'],label='Xn')
#plt.plot(fX_dim_output['dim'],fX_dim_output['specificity'],label='fX')
plt.plot(pX_dim_output['dim'],pX_dim_output['specificity'],label='pX')
plt.xlabel('dimension')
plt.ylabel('specificity')
plt.legend()
plt.savefig('dim_var_spec.png')
```