Classification Using Non-Parametric Density Estimation and Performance Evaluation

Pattern Recognition/ECE 471

William Willie Wells
wwells4@utk.edu

Abstract

Parametric probability density functions such as the Gaussian are convenient estimations of data distribution but often do not characterize actual densities observed. High dimensional densities often have more than one local optima and are not accurately represented as a product of functions with a single dimension. Nonparametric density methods are used to classify data points without assuming the nature of the probability density function and they strive to improve classification. One method of nonparametric density estimation is to compare the k nearest neighboring points and classify the multidimensional point in question according to majority rule. This method is known as the k-nearest neighbor algorithm. The Minkowski distance, a metric on Euclidean space, was used to calculate the distance between points. The Minkowski distance was varied in degree of the norm used and the amount of dimensions of the feature space used. A weighted contribution of the neighbors based on their nearness was utilized in the multiple class case where the amount of neighbors of 2 or more classes was equivalent. With the above variances different values of k were used on an unmodified data set, the same data set dimensionality reduced using Principal Component Analysis, the same data set with its dimensions reduced using Fisher's Linear Discriminant, and a different unmodified data set. Cross validation, a process to validate the accuracy of the classifier used, was used on the last data set. The best value of k for this set, fglass, was k=3. Bayes decision rule was more accurate for the Pima data set.

Introduction:

Multidimensional data sets, data sets with many degrees of freedom, are common across the science, technology, engineering, and mathematics fields. These data sets can be inadequately represented by parametric probability density functions. Nonparametric density estimation methods estimate the probability density function without assuming a particular form for the probability density function. These estimation methods are able to achieve estimation optimality for all sets as the amount of observed data is increased, which is something parametric density estimation cannot achieve. The k-Nearest Neighbor algorithm implicitly estimates the decision boundary. A Minkowski distance with a norm of 2, Euclidean distance, is commonly used to derive the distance between points. In k-Nearest Neighbor the distance between the test point and all training points is calculated. The resulting vector is sorted in ascending order. The classes of the first k elements are totaled. The individual class sums are compared. The class with the highest sum is assigned to the classification of the test point. In the case of equal sums different class weighting techniques have been developed. (1)(3)(4)(5)

Minkowski distance is a generalization of Manhattan, Euclidean, and Cheybyshev distances. The normed root of the sum of the difference between vector components of two points raised to the norm power is the Minkowski distance. The most popular Minkowski norms (1, 2, and ∞) correspond respectively to Manhattan, Euclidean, and Cheybyshev distances. Points of equal distance form a rhombus with 45 degree angles to the axes using Manhattan distance in the two feature case. In the case of Euclidean distance a circle is formed from equal distance points using two features. The Cheybyshev distance forms a square in the two feature case with respect to equal distant points. (2)

Evaluation of an algorithms performance in classifying a test point can be measured in multiple ways, two of which are algorithm execution time and accuracy of classification.

Algorithm execution time for large data sets with $O(dn^2)$ can still take hours on a supercomputer to complete. In general algorithm execution time is represented in big O notation but actual execution time is used as well in performance evaluation. Implementing partial Minkowski distance with k-Nearest neighbor is an attempt to lower the algorithm execution time. The accuracy of a multiclass system is measured using the number of points/items classified correctly divided by the total amount of points/items classified. The sum of the error rate and the accuracy is equivalent to one. (1)

A model validation technique that measures how an algorithm will perform in a generalized independent case is cross validation. A nearly unbiased estimate of the expected true error rate is able to be obtained from cross validation. Leave-One-Out and m-fold cross validation are popular types of cross validation that use an equal number of test points as the validation set on each iteration of the validation process. The training set is divided into a validation (test) set and the remaining training set. The classifier is designed using the unique training set and used on the test set for each validation set. The overall performance evaluation is the mean of the individual performance evaluations. (6) (7)

Technical Approach:

The variables needed to implement Minkowski distance are a training point vector (x_i), a test point vector (y_i), number of elements of the vectors to be summed (d), and value of the norm (p). A generic Minkowski distance calculation is computed thus:

$$L_p(\mathbf{x},\mathbf{y}) = (\sum_{i=1}^{d} |x_i-y_i|^p)^{(1/p)}$$

The difference, x_i-y_i, is calculated first and the difference is the multiplied by itself p times. The result of that calculation is then summed. The pth root of this sum is taken using the pow(x,n) function of <cmath>.

With k-Nearest Neighbor the function described above wMinkowski(x,y_j,d,p) is used to calculate the unsorted vector of distances from the point vector (y_j) to each point in x. The unsorted vector (u) is sorted using insertsort(u,s,i) provided in the Matrix library and written by Hairong Qi, where s is the sorted vector and i is a respective vector of indices. The number of classes is determined; a count is iterated each time a different class of the training set is encountered. This is used for storage, to keep track, of the number of nearest neighbors. The number of nearest neighbors is then counted and a weight (w_c) based on the sum of nearest neighbors of that respective class (c) is used.

$$w_c=\sum_{i=0}^{k-1}[i \text{ iff } x_i \in c \mid 0 \text{ otherwise}]$$

The point is then classified according to which class has the highest count. If the counts are equal the weights are compared and the point is assigned the class with the lower weight. This is repeated for all point vectors (y_j), for all odd integers less than or equal to k, and all Minkowski distances up to p such that:

$$\forall k,\ 1 < k \le k_{Max},\ k \text{ is odd};\ \forall p,\ 1 < p \le p_{Max}$$

The performance of the classification is then evaluated.

Using <ctime> a variable of type clock_t is set to clock() before utilizing k-Nearest Neighbor. At the completion of classifying all point vectors in the test set another call to clock() is made and the difference of the two is divided by CLOCKS_PER_SEC. An output in seconds is the result. The known classification is compared to the classification from k-Nearest Neighbor and the accuracy of the classification is calculated.

For cross validation the training set is split into a validation set and a test set. Since the given fglass.grp has uneven groupings of indices, the number of items in the respective test set is found by counting all non-zero elements. The testing and training sets are separated by comparing each points index to the test set indices. Classification is then implemented using k-Nearest Neighbor. Then the mean accuracy of all runs is calculated:

$$\mu_A = \frac{1}{n}\sum_{i=1}^{n} |A_i|$$

Where n = 10, the number of uneven validation groups in fglass.grp.

Experiments and Results:

Using integers 1 through 5 for the norm (p) of the Minkowski distance metric on the fglass data set, k=3 had the highest accuracy for all but a norm value of 1, for which the best k was k=1. For the Pima and fglass sets floor(k = sqrt(n)) = 14, where n is equal to the number of training points. For analysis, k was varied up to 19. The partial dimension used for both the unmodified and PCA data sets is d=3. Execution time (t) in seconds and classification accuracy (true/total):

kNN		**Pima, Full**		**Pima, Partial**		**Pima, PCA, Full**		**Pima, PCA, Partial**		**Fglass, Full**	
k	**p**	**t**	**Accuracy**	**t**	**Accuracy**	**t**	**Accuracy**	**t**	**Accuracy**	**t**	**Accuracy**
1	1	0.06	0.704819	0.08	0.71988	0.06	0.731928	0.07	0.614458	0.01	**0.710589**
3	1	0.07	0.756024	0.05	0.762048	0.06	0.753012	0.06	0.644578	0.05	0.702812
5	1	0.07	0.746988	0.06	0.753012	0.07	0.756024	0.06	0.653614	0.02	0.676854
7	1	0.06	0.756024	0.06	0.774096	0.06	0.771084	0.06	0.674699	0.04	0.664127
9	1	0.07	0.76506	0.06	0.75	0.06	0.78012	0.06	0.677711	0.04	0.643575
11	1	0.07	0.759036	0.06	0.746988	0.07	0.76506	0.06	0.668675	0.02	0.657461
13	1	0.06	0.746988	0.06	0.753012	0.06	0.768072	0.06	0.701807	0.05	0.667568
15	1	0.07	0.746988	0.06	0.762048	0.06	0.76506	0.06	0.680723	0.03	0.671265
17	1	0.07	0.75	0.06	0.756024	0.07	0.75	0.06	0.665663	0.04	0.661216
19	1	0.06	0.753012	0.06	0.762048	0.06	0.771084	0.06	0.674699	0.03	0.645133
1	2	0.08	0.704819	0.07	0.728916	0.07	0.75	0.07	0.60241	0.05	0.691196
3	2	0.07	0.740964	0.06	0.746988	0.06	0.786145	0.06	0.650602	0.02	**0.69925**
5	2	0.07	0.743976	0.06	0.73494	0.08	0.756024	0.07	0.683735	0.06	0.65714
7	2	0.07	0.753012	0.07	0.75	0.06	0.762048	0.06	0.686747	0.04	0.63078
9	2	0.07	0.76506	0.06	0.746988	0.07	0.768072	0.07	0.704819	0.01	0.641944
11	2	0.08	0.76506	0.07	0.743976	0.07	0.771084	0.06	0.677711	0.06	0.630221
13	2	0.07	0.76506	0.06	0.756024	0.07	0.756024	0.07	0.674699	0.02	0.643332
15	2	0.07	0.771084	0.07	0.76506	0.07	0.756024	0.06	0.686747	0.07	0.630875
17	2	0.08	0.762048	0.06	0.759036	0.07	0.75	0.07	0.677711	0.02	0.633468
19	2	0.07	0.759036	0.06	0.777108	0.07	0.759036	0.06	0.665663	0.04	0.624067
1	3	0.07	0.710843	0.07	0.707831	0.07	0.731928	0.07	0.608434	0.04	0.688171
3	3	0.08	0.743976	0.06	0.731928	0.07	0.771084	0.06	0.650602	0.04	**0.702128**
5	3	0.07	0.73494	0.07	0.731928	0.07	0.768072	0.07	0.680723	0.06	0.66109
7	3	0.08	0.746988	0.07	0.737952	0.07	0.768072	0.06	0.692771	0.02	0.630095
9	3	0.07	0.771084	0.06	0.740964	0.07	0.76506	0.07	0.701807	0.06	0.624855
11	3	0.08	0.768072	0.07	0.743976	0.07	0.771084	0.06	0.698795	0.01	0.618816

13	3	0.07	0.756024	0.06	0.759036	0.07	0.76506	0.07	0.686747	0.07	0.618138
15	3	0.08	0.759036	0.07	0.76506	0.07	0.756024	0.07	0.677711	0.02	0.594656
17	3	0.07	0.762048	0.06	0.762048	0.07	0.75	0.06	0.668675	0.07	0.603543
19	3	0.08	0.762048	0.07	0.774096	0.07	0.743976	0.07	0.674699	0.01	0.588027
1	4	0.08	0.707831	0.07	0.710843	0.08	0.740964	0.07	0.614458	0.05	0.667341
3	4	0.07	0.73494	0.07	0.722892	0.07	0.746988	0.06	0.656627	0.03	**0.701143**
5	4	0.08	0.73494	0.07	0.731928	0.07	0.759036	0.07	0.686747	0.06	0.632843
7	4	0.08	0.746988	0.06	0.73494	0.07	0.76506	0.07	0.695783	0.04	0.628933
9	4	0.07	0.759036	0.07	0.753012	0.07	0.774096	0.06	0.704819	0.05	0.623645
11	4	0.08	0.76506	0.06	0.75	0.08	0.756024	0.07	0.707831	0.02	0.621653
13	4	0.08	0.759036	0.07	0.762048	0.07	0.762048	0.07	0.692771	0.07	0.608741
15	4	0.08	0.759036	0.07	0.76506	0.07	0.746988	0.06	0.683735	0.04	0.581098
17	4	0.08	0.759036	0.07	0.771084	0.07	0.753012	0.07	0.668675	0.04	0.58479
19	4	0.07	0.762048	0.06	0.78012	0.07	0.756024	0.07	0.683735	0.03	0.56747
1	5	0.08	0.716867	0.07	0.713855	0.08	0.737952	0.06	0.611446	0.07	0.655245
3	5	0.08	0.728916	0.07	0.716867	0.07	0.759036	0.07	0.656627	0.01	**0.689747**
5	5	0.08	0.740964	0.07	0.728916	0.08	0.768072	0.07	0.686747	0.05	0.620243
7	5	0.08	0.746988	0.07	0.725904	0.07	0.771084	0.07	0.692771	0.05	0.613833
9	5	0.08	0.756024	0.07	0.753012	0.08	0.759036	0.07	0.71988	0.05	0.620598
11	5	0.08	0.76506	0.07	0.743976	0.07	0.75	0.06	0.710843	0.05	0.612346
13	5	0.08	0.759036	0.07	0.743976	0.07	0.762048	0.07	0.698795	0.03	0.595778
15	5	0.08	0.756024	0.07	0.771084	0.08	0.759036	0.07	0.689759	0.05	0.580503
17	5	0.08	0.759036	0.07	0.768072	0.07	0.743976	0.07	0.674699	0.04	0.579145
19	5	0.08	0.753012	0.07	0.78012	0.07	0.762048	0.07	0.671687	0.07	0.575846

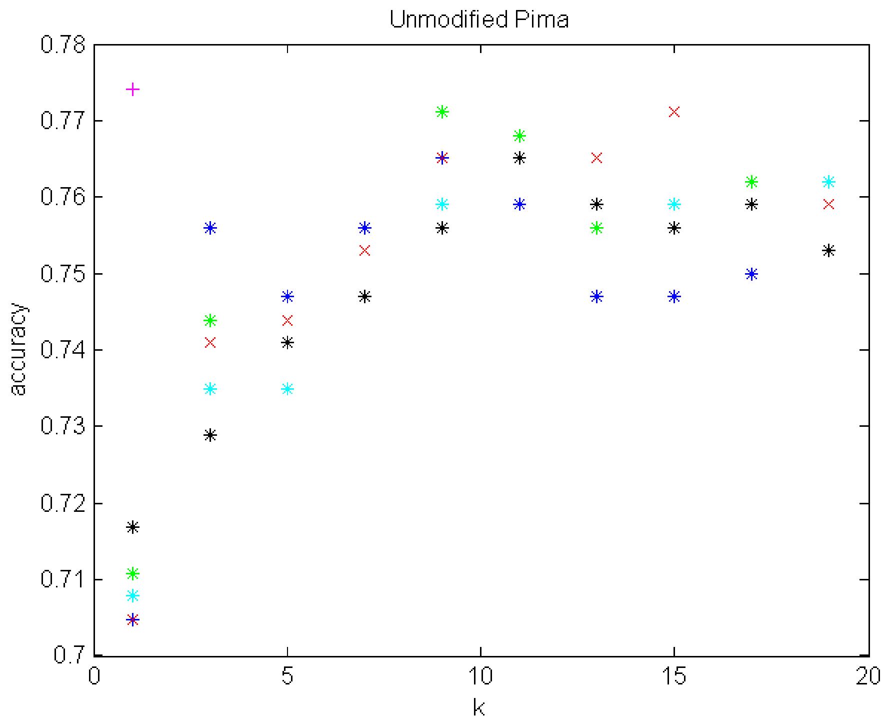
The Fisher's Linear Discriminant reduced Pima data set only has a single dimension so varying the Minkowski norm had no effect on the result, for the same reason partial distance is not possible with this set.

kNN		**Pima, FLD, Full**	
k	**p**	**t**	**Accuracy**
1	1	0.06	0.731928
3	1	0.05	0.771084
5	1	0.06	0.759036
7	1	0.06	0.756024
9	1	0.06	0.759036
11	1	0.05	0.771084
13	1	0.06	0.768072
15	1	0.06	0.771084
17	1	0.06	0.76506
19	1	0.06	0.771084

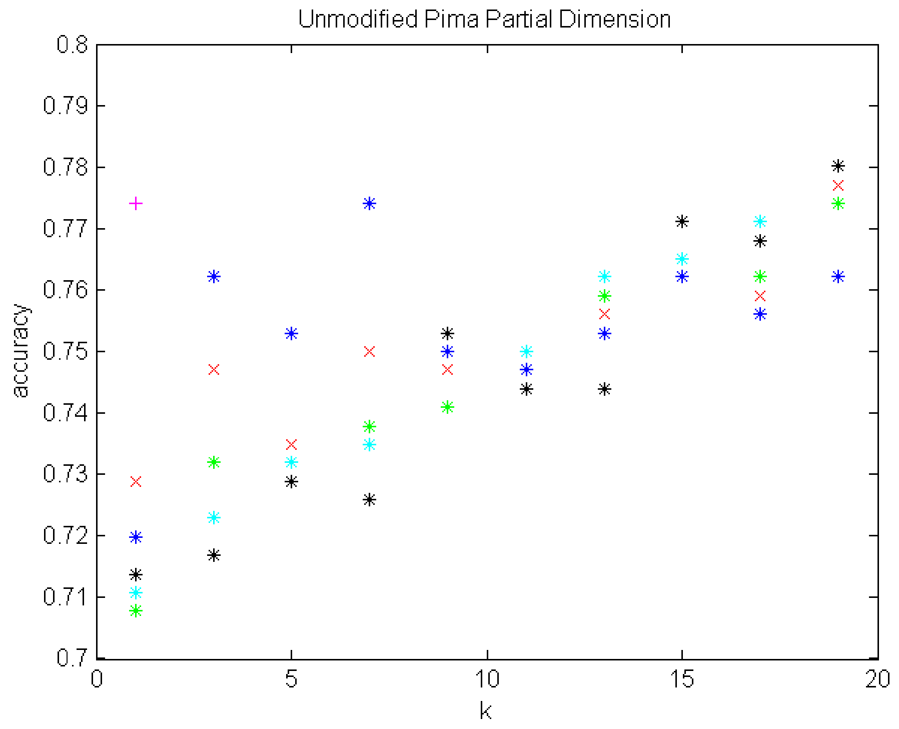
The execution time and accuracy for the MAP classification:

Pima, MAP		Pima, Improved Prior, MAP		Pima, PCA, MAP		Pima, FLD, MAP	
t	**Accuracy**	**t**	**Accuracy**	**t**	**Accuracy**	**t**	**Accuracy**
0.01	0.740964	0.01	0.774096	0.01	0.777108	0.01	0.801205

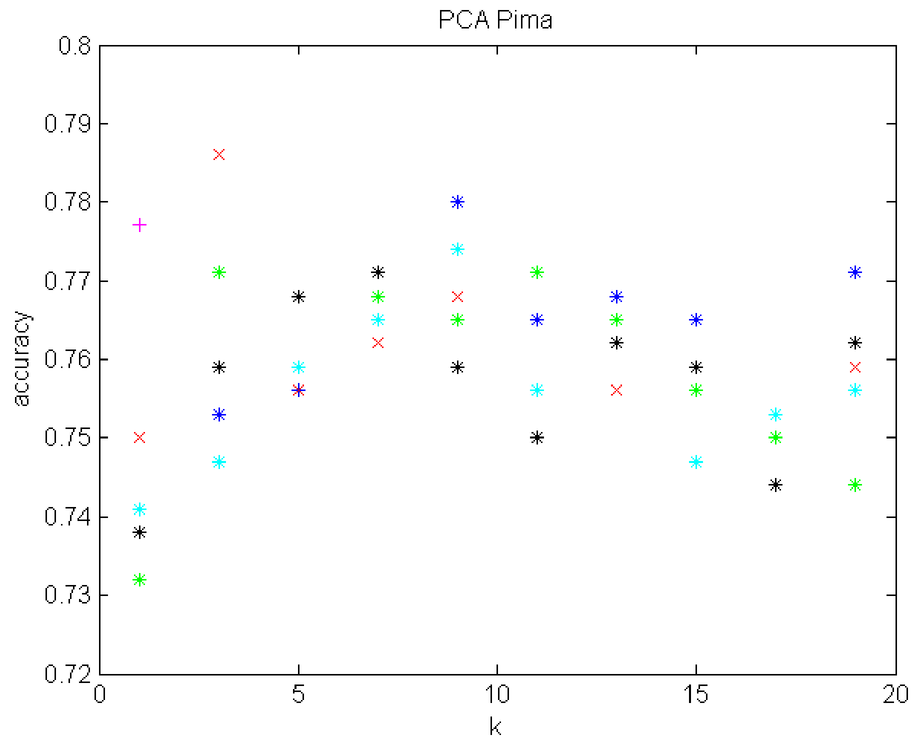
Visual comparison of different k's with the unmodified Pima data set, the + is the MAP accuracy, Euclidean distance is marked by x's:



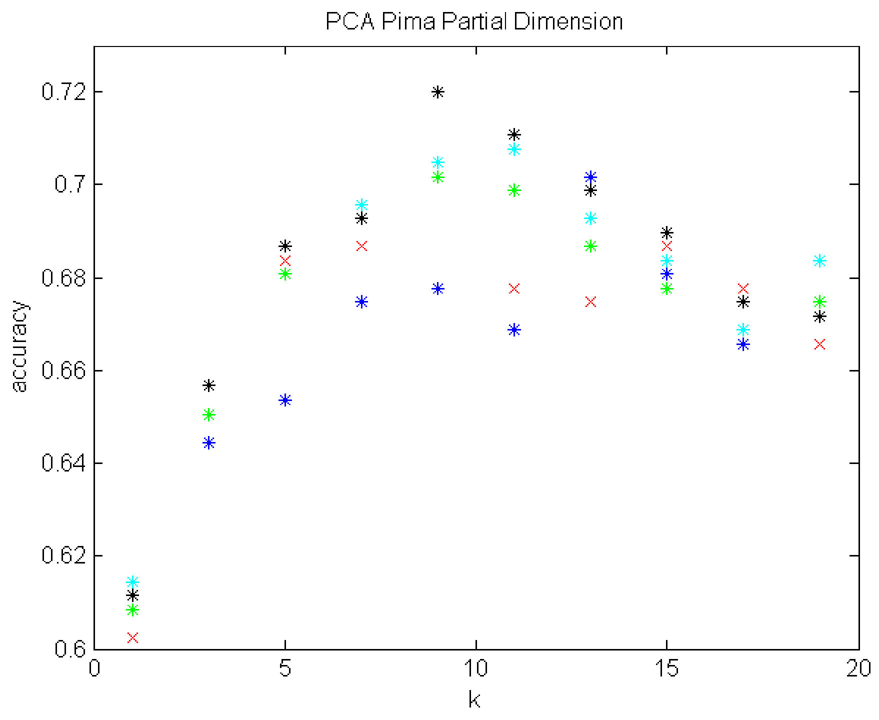
The unmodified Pima data set with partial distance:



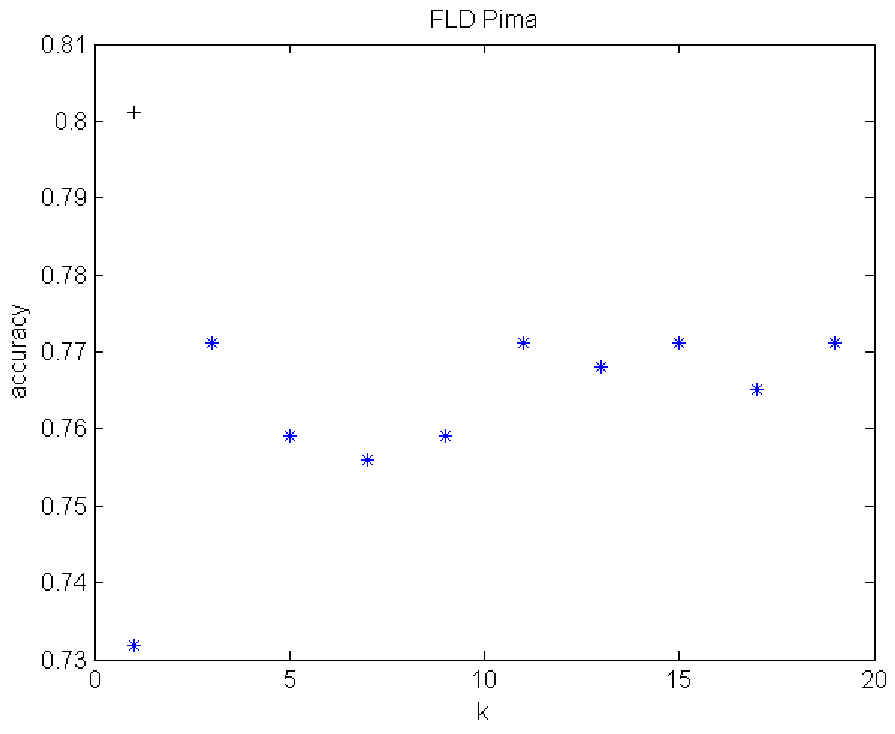
The Pima data set with Principal Component Analysis dimensionality reduction:



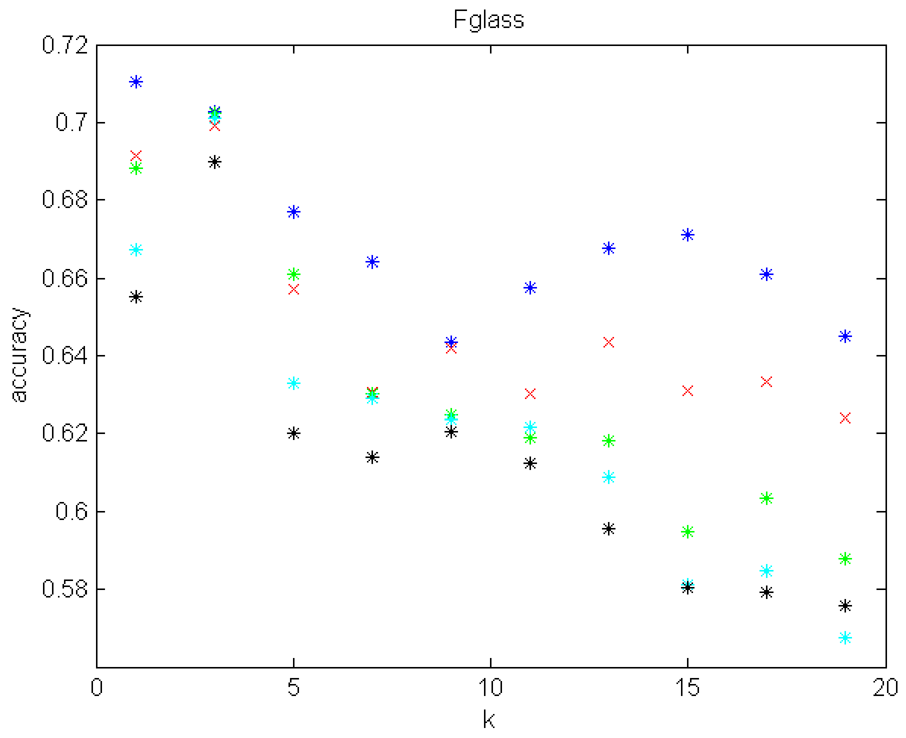
The Pima data set with PCA dimensionality reduction and partial distance:



The Pima data set with FLD dimensionality reduction:



The Fglass data set:



Conclusion:

The unmodified Pima data set partial distance k-Nearest Neighbor implementation has a higher classification accuracy than the respective full distance for k < 9 and p < 3 (Manhattan and Euclidean distance) as well as on average saving 0.01 seconds. Whereas the Pima data set with Principal Component Analysis and partial distance k-Nearest Neighbor implementation has lower classification accuracy than the respective full distance classification $\forall k, p$. Only outliers such as (k = 3, p = 2) for the full Pima PCA exceed the MAP accuracy. The FLD implementation did not approach its respective MAP classification accuracy. The MAP computational times are between 5 and 8 times faster than the respective kNN computational times. The best k for the Fglass data set is k=3, due to the low density of training points of a similar class. With more test samples the best value for k would be higher. For FLD the best k value seems to be near k = sqrt(n). The unmodified and PCA best k values seem to be around k = 9. Nonparametric density estimation is able to achieve estimation optimality but requires a large amount of training points to reach that achievement.

References:

1 R. O. Duda, P. E. Hart, D. G. Stork, *Pattern Classification*, 2nd Edition, John Wiley, 2001.
2 http://en.wikipedia.org/wiki/Distance
3 http://en.wikipedia.org/wiki/K-nearest_neighbor_algorithm
4 http://web.eecs.utk.edu/~qi/ece471-571/lecture11_nonpara_knn.pdf
5 http://www.siam.org/proceedings/datamining/2003/dm03_19GrayA.pdf
6 http://en.wikipedia.org/wiki/Cross-validation_(statistics)
7 http://web.eecs.utk.edu/~qi/ece471-571/lecture09_performance.pdf

Appendix:

```
/*****************************************************************
 * nonParaDE.cpp - Non-Parametric Density Estimation
 *
 * Purpose: implement k-Nearest Neighbor and non-uniform size
 *                  validation set Cross Validation.
 *
 *          - wMinkowski: calculate Minkowski distances of xi-p
 *          - wKNN: classify a point (p) using k-nearest neighbor
 *          - mkNN: classify a vector of points, evaluate accuracy
 *                       and timing using kNN
 *          - divideSet: separate a data set into training and
 *                             validation sets
 *          - crossValidate: use cross validation to verify accuracy
 *
 * Author: William Willie Wells
 *
 * Created: October 2013
 *
 *
 *****************************************************************/

#include <cstdlib>
#include <cmath>
#include <ctime>
#include "PnE.h"

// Calculate Minkowski distances from each point
Matrix wMinkowski(Matrix &x, Matrix &pt, int d,int p){
	Matrix tmp,res;
	double t1;

	tmp.createMatrix(x.getRow(),1);
	res.createMatrix(x.getRow(),1);
	// (sum j=[1,p]: [abs(x(i,j)-pt(0,j))]^d)^1/d
	for(int i=0;i<x.getRow();i++){
		for(int j=0;j<p;j++){//partial feature
			t1=x(i,j)-pt(0,j);
			tmp(i,0)=1.0;
			for(int k=0;k<d;k++){//k distance metric
				tmp(i,0)*=abs(t1);
			}
			res(i,0)+=tmp(i,0);
		}
		// compute dth root
		res(i,0)=pow(res(i,0),(1/((double)d)));
	}

	return res;
}

// Use k-Nearest Neighbor (kNN) to clasify a point
double wKNN(Matrix &x,Matrix &pt,int k, int d, int p){
	Matrix uDist,sDist,iDist,ctc;
	int cnt=0;
	int cnt2=-1;
	int cnt3=-1;
	int cMax=-1;

	// calculate distances from each point
	uDist=wMinkowski(x,pt,d,p);
	uDist=transpose(uDist);

	// sort distances lowest-->highest
	sDist.createMatrix(1,uDist.getCol());
	iDist.createMatrix(1,uDist.getCol());
	insertsort(uDist,sDist,iDist);

		/*classification*/
	// determine number of classes
	ctc.createMatrix(x.getRow(),3);
	for(int i=0;i<x.getRow();i++){
		if(i==0){
			ctc(cnt,0)=x(i,x.getCol()-1);
			cnt++;
		}else{
			for(int j=0;j<cnt;j++){
				if(x(i,x.getCol()-1)==ctc(j,0)){
					break;
				}else if(j==cnt-1){
					ctc(cnt,0)=x(i,x.getCol()-1);
					cnt++;
				}
			}
		}
	}
	ctc=subMatrix(ctc,0,0,cnt-1,2);

	// count respective classes of first k nearest neighbors
	if(k>x.getRow()){
		k=x.getRow();
	}
	for(int w=0;w<k;w++){
		for(int v=0;v<ctc.getRow();v++){
			if(x((int)iDist(0,w),x.getCol()-1)==ctc(v,0)){
				ctc(v,1)+=1.0;
				ctc(v,2)+=w;
			}
		}
	}

	// classify point
	for(int c=0;c<cnt;c++){
		if(ctc(c,1)>cnt2 | (ctc(c,1)==cnt2 & ctc(c,2)<cnt3)){
			cMax=ctc(c,0);
			cnt2=ctc(c,1);
			cnt3=ctc(c,2);
		}
	}

	return cMax;
}

// Use k-Nearest Neighbor method to classify a vector of points
Matrix mkNN(Matrix &x, Matrix &y, int k, int d,int p){
	Matrix tyr,ty,oE;
	int n,o=0;
	clock_t t0;

	// initiate matrices
	tyr.createMatrix(1,y.getCol()-1);
	ty=formatResult(y,y);
	oE.createMatrix(((k+1)/2)*d,4);

	// run kNN for all points, all distances [0, d], and all odd values [1,k)
	for(int m=1;m<d+1;m++){
		n=1;
		while(n<k+1){
			t0=clock();
			for(int i=0;i<y.getRow();i++){
				for(int j=0;j<y.getCol()-1;j++){
					tyr(0,j)=y(i,j);
				}
				ty(i,y.getCol())=wKNN(x,tyr,n,m,p);
			}
			// measure run time and accuracy for each (k,d)
			oE(o,0)=n;
			oE(o,1)=m;
			oE(o,2)=execTime(t0);
			oE(o,oE.getCol()-1)=wAccuracy(ty,ty.getCol()-1);
			n+=2;
			o++;
		}
	}

	return oE;
}

// Divide data set into a training and a validation(test) set
void divideSet(Matrix &x,Matrix &M,Matrix &tr,Matrix &te,int n){
	Matrix tTr,tTe;
	int c=0;
	int rCt=0;
	int eCt=0;
	int flag=0;

	// find number of items in test set
	for(int j=0;j<M.getRow();j++){
		if(M(j,n)>0){
			c++;
		}
	}
	tTe.createMatrix(c,x.getCol());
	tTr.createMatrix(x.getRow()-c,x.getCol());

	// separate training and testing sets
	for(int i=0;i<x.getRow();i++){
		for(int k=0;k<c;k++){
			if(i+1==M(k,n)){
				for(int p=0;p<x.getCol();p++){
					tTe(eCt,p)=x(i,p);
				}
				eCt++;
				flag=1;
			}
		}
		if(flag==0){
			for(int r=0;r<x.getCol();r++){
				tTr(rCt,r)=x(i,r);
			}
			rCt++;
		}else{ flag=0;}
	}

	// return training and test sets
	tr=tTr;
	te=tTe;
}

// Run cross Validation n times with n semi-random validation sets
Matrix crossValidate(Matrix &x, Matrix &M, int kMax, int dMax, int n){
	Matrix rpe,tr,te,cva;

	// implement cross validation n times
	for(int ni=0;ni<n;ni++){
		// separate training and test data sets
		divideSet(x,M,tr,te,ni);

		// use kNN to classify data set
		rpe=mkNN(tr,te,kMax,dMax,tr.getCol()-1);

		// return mean accuracy (1-MAE not MSE)
		if(ni==0){
			cva=rpe;
		}else{
			for(int i=0;i<rpe.getRow();i++){
```

```
                                cva(i,3)+=rpe(i,3);
                                cva(i,2)+=rpe(i,2);
                                if(ni==n-1){
                                        cva(i,3)/=n;
                                }
                        }
                }
        }

        return cva;
}

/*****************************************************************
 * proj3.cpp - implement Non-Parametric Density Estimation      *
 *                                                               *
 * Purpose: Implement k-nearest neighbor classification, use    *
 *              cross validation, perform evalutions on data sets. *
 *                                                               *
 * Author: William Willie Wells                                  *
 *                                                               *
 * Created: October 2013                                         *
 *                                                               *
 *                                                               *
 *****************************************************************/

#include <fstream>
#include <cstdlib>
#include <cmath>
#include "Pr.h"
#include "mpp.h"
#include "dimRedux.h"
#include "nPDE.h"

using namespace std;

// MAP and KNN Classification with PCA and FLD Dimensionality Reduction,
// Performance Evaluation using run time, accuracy, and cross validation
int main(int argc, char **argv){
        int p,fd,fg,kMax,dMax;
        double mP;
        Matrix X,nX,tX,fX,tXc,fXc,S;
        Matrix Y,nY,tY,fY,nYc,tYc,fYc,nyk,tyk,fyk,sE,oE;
        Matrix Z,zk,M;

                        /*Preprocess data*/
        // error check input format
        if(argc<7){
        cout << "file.tr # of columns in file file.te";
                cout << " file.dat # of columns in file file.grp";
                cout << " # of columns of largest row"<< endl;
                exit(1);
    }

        // read in data
    p=atoi(argv[2]);
        fd=atoi(argv[5]);
        fg=atoi(argv[7]);
    X=readData(argv[1],p);
    Y=readData(argv[3],p);
        Z=readData(argv[4],fd);
        M=readData(argv[6],fg);

        // find "No" in argv[1], argv[3] and assign bit values to (X,Y)(:,n)
        strToBit(X,X.getCol()-1,"No",argv[1]);
        strToBit(Y,Y.getCol()-1,"No",argv[3]);

        // eleminate the first row
        nX=subMatrix(X,1,0,X.getRow()-1,X.getCol()-1);
        nY=subMatrix(Y,1,0,Y.getRow()-1,Y.getCol()-1);
        Z=subMatrix(Z,1,0,Z.getRow()-1,Z.getCol()-1);
```

```
// normalize the data sets so that features have the same scale
// using the mean and covariance
xNorm(nY,nX);
xNorm(nX,nX);
xNorm(Z,Z);

            /*Dimensionality Reduction*/
// use Principal Component Analysis to derive a new set of basis,
// and represent the data
S=cov(nX,nX.getCol()-1);
tX=wPCA(nX,S);
tY=wPCA(nY,S);

// use Fisher's Linear Discriminant method to derive
// the projection direction that best separates the data
fX=wFLD(nX,nX);
fY=wFLD(nY,nX);

            /*MAP Classification with run time and accuracy Evaluation*/
// classify data with full dimension using equal prior probability
sE=outEval(nX,nY,0.5);
oE.createMatrix(sE.getRow(),sE.getCol());
copyMat(sE,oE);

// classify data with full dimension using a prior probability
// that improves classification accuracy
mP=maxPrior(nX,nY,1000.0);
sE=outEval(nX,nY,mP);
oE=formatResult(oE,sE);

// classify data with PCA dimensionality reduction and improved priors
tXc=formatResult(tX,nX);
tYc=formatResult(tY,nY);
sE=outEval(tXc,tYc,mP);
oE=formatResult(oE,sE);

// classify data with FLD dimensionality reduction and improved priors
fXc=formatResult(fX,nX);
fYc=formatResult(fY,nY);
sE=outEval(fXc,fYc,mP);
oE=formatResult(oE,sE);

            /*KNN Classification with run time and accuracy Evaluation*/
// vary k, 1<k<sqrt(n), k is always odd; vary distance d=[1,dMax]
kMax=(int)sqrt(X.getRow())+5;
dMax=5;

// classify Pima data full dimension and write to output file
cout<<"unreduced: max prior, kNN"<<endl;
nyk=mkNN(nX,nY,kMax,dMax,nX.getCol()-1);
writeData(nyk,"pima_kf.dat");
cout<<nyk<<endl;

// classify Pima data full dimension and partial distance
nyk=mkNN(nX,nY,kMax,dMax,nX.getCol()-4);
writeData(nyk,"pima_kp.dat");
cout<<"    partial distance:"<<endl<<nyk<<endl;

// classify Pima data PCA and write to output file
cout<<"PCA: kNN"<<endl;
tyk=mkNN(tXc,tYc,kMax,dMax,tXc.getCol()-1);
writeData(tyk,"pima_pkf.dat");
cout<<tyk<<endl;

// classify Pima data PCA and partial distance
tyk=mkNN(tXc,tYc,kMax,dMax,tXc.getCol()-3);
writeData(tyk,"pima_pkp.dat");
cout<<"    partial distance:"<<endl<<tyk<<endl;
```

```
	// classify Pima data FLD (only one dimension/distance possible)
	cout<<"FLD: kNN"<<endl;
	fyk=mkNN(fXc,fYc,kMax,1,fXc.getCol()-1);
	writeData(fyk,"pima_fk.dat");
	cout<<fyk<<endl;

	// classify fglass data using kNN with cross validation
	M=removeSpace(M,M.getRow());
	kMax=(int)sqrt(Z.getRow())+5;
	zk=crossValidate(Z,M,kMax,dMax,M.getCol());
	writeData(zk,"fglass_kc.dat");
	cout<<zk<<endl;

			/*Combined MAP Performance Evaluation*/
	// output to command line MAP run time and accuracy
	writeData(oE,"pima_m.dat");
	oE=transpose(oE);
	oE=subMatrix(oE,0,oE.getCol()-2,oE.getRow()-1,oE.getCol()-1);
	cout<<oE<<endl;

	return 0;
}

/*****************************************************************
 * prepEval.cpp - preprocessing and performance evaluation
 *
 * Purpose: preprocess data to be used with the Matrix library
 *			and evaluate classification performance.
 *
 *		- execTime: measure execution time in seconds
 *		- strToBit: find a user defined string and assign it a
 *				binary value and assign all else the other
 *				binary value
 *		- xNorm: normalize a generic data set using mean and
 *			    standard deviation
 *		- splitData: split all members of a class predefined in
 *				a training set, user defined input
 *		- copyColumn: copy a specified column in one Matrix to a
 *				specified column in another Matrix
 *		- copyMat: copy a Matrix into another Matrix
 *		- formatResult: add original classification column to a
 *				    reduced dimension Matrix
 *		- removeSpace: remove rows of all zeros and transpose
 *		- wAccuracy: calculate accuracy of classification
 *		- ruleEval: calculate performance evaluation values
 *
 * Author: William Willie Wells
 *
 * Created: September 2013
 *
 * Modified: October 2013
 *****************************************************************/

#include <fstream>
#include <cstdlib>
#include <cmath>
#include <ctime>
#include "Matrix.h"

// Measure execution time
float execTime(clock_t t0){
	return ((float)(clock()-t0))/CLOCKS_PER_SEC;
}

// Find indicated string in a file and assign bit values to x(:,n)
void strToBit(Matrix &x, int n, string xStr,char *xFile){
	string findS;

	// sift through xFile until the nth column, compare value to xStr,
	// and assign x(:,n) a binary value.
```

```
        ifstream fS(xFile,ios::in);
        for(int i=0;i<x.getRow();i++){
                for(int j=0;j<x.getCol();j++){
                        fS >> findS;
                        if(j==n){
                                if(findS==xStr){
                                        x(i,n)=0;
                                }else{
                                        x(i,n)=1;
                                }
                        }
                }
        }
}

// Normalize the data set so that features have the same scale
void xNorm(Matrix &y,Matrix &x){
        Matrix mu,S;

        // calculate the mean and covariance
        mu=mean(x,x.getCol()-1);
        S=cov(x,x.getCol()-1);

        // normalize the data set
        for(int i=0; i<y.getRow();i++){
                for(int j=0; j<y.getCol()-1;j++){
                        y(i,j)=(y(i,j)-mu(j,0))/(sqrt(S(j,j)));
                }
        }
}

// Find class identifier and copy row into Matrix
Matrix splitData(Matrix &x, int c){
        Matrix tmp;
        int count=0;

    // find class identifier and copy row
    tmp.createMatrix(x.getRow(),x.getCol());
    for (int i=0; i<x.getRow(); i++){
        if(x(i,x.getCol()-1) == c){
                        for(int j=0;j<x.getCol();j++){
                                tmp(count,j)=x(i,j);
                        }
                        count++;
                }
        }

        return subMatrix(tmp,0,0,count-1,tmp.getCol()-2);
}

// Copy a specific column in Matrix 1 to a specific column in Matrix 2
void copyColumn(Matrix &data, Matrix &rDat, int dc,int rc){

        for(int i=0;i<data.getRow();i++){
                rDat(i,rc)=data(i,dc);
        }
}

// Copy values of one Matrix into another without modifying size of target
void copyMat(Matrix &data, Matrix &rDat){

        for(int j=0;j<data.getCol();j++){
                copyColumn(data,rDat,j,j);
        }
}

// After dimensionality reduction reformat matrix for classification
Matrix formatResult(Matrix &xCopy1, Matrix &xCopy2){
        Matrix tmp;
```

```
	// append last column of Matrix 2 to Matrix 1
	tmp.createMatrix(xCopy1.getRow(),xCopy1.getCol()+1);
	copyMat(xCopy1,tmp);
	copyColumn(xCopy2,tmp,xCopy2.getCol()-1,tmp.getCol()-1);

	return tmp;
}

// Remove extra space (all zero rows) and return transpose
Matrix removeSpace(Matrix &x,int n){
	Matrix tmp;
	int j=0;

	// transpose matrix and return non-zero data
	x=transpose(x);
	tmp.createMatrix(x.getRow(),(int)((n+1)/2));
	for(int i=0;i<n;i+=2){
		copyColumn(x,tmp,i,j);
		j++;
	}

	return tmp;
}

// Calculate accuracy of classification
double wAccuracy(Matrix &rX, int nc){
	double tru=0.0;
	double tot=(double)rX.getRow();

	// sum True Positive and True Negative values
	for(int i=0;i<rX.getRow();i++){
		if(rX(i,nc-1)==rX(i,nc)){
			tru=tru+1.0;
		}
	}

	return tru/tot;// divide by total
}

// Calculate True Positive, False Negative, True Negative, False Positive,
// Sensitivity, Specificity, precision, and recall
Matrix ruleEval(Matrix &xC){
	Matrix tmp;

	// calculate total number of TP, FN, TN, and FP results
	tmp.createMatrix(1,8);
	for(int i=0;i<xC.getRow();i++){
		if(xC(i,xC.getCol()-1)==1){
			if(xC(i,xC.getCol()-2)==1){
				tmp(0,0)+=1.0;// (1,1) TP
			}else{
				tmp(0,2)+=1.0;// (0,1) FP
			}
		}else{
			if(xC(i,xC.getCol()-2)==1){
				tmp(0,3)+=1.0;// (1,0) FN
			}else{
				tmp(0,1)+=1.0;// (0,0) TN
			}
		}
	}

	// determine sensitivity--recall, specificity, and precision
	tmp(0,4)=tmp(0,0)/(tmp(0,0)+tmp(0,3));// sensitivity--recall
	tmp(0,5)=tmp(0,1)/(tmp(0,1)+tmp(0,2));// specifity
	if(tmp(0,0) >0 & tmp(0,2)>0){
		tmp(0,6)=tmp(0,0)/(tmp(0,0)+tmp(0,2));// precision
	}
	tmp(0,7)=tmp(0,4);// recall
```

```
        return tmp;
}

/*****************************************************************
 * mpp.cpp - data classification using MAP on n-by-d data sets
 *
 * Purpose: perform operations on training sets and test samples
 *                   to design classification decision rules.
 *
 *             - mDist: compute Mahalanobis distance
 *             - MaP: calculate Gaussian Probability Density Function
 *             - comparePdf: compare pdf values of 2 classes for a set
 *             - MaPClass: classify test data using Maximum a-Posteriori
 *             - maxPrior: find prior probability that maximizes
 *                              classification accuracy
 *             - outEval: output performance evaluation results
 *             - muSigmaX: calculate Bayesian Discriminant Function
 *             - sameSigma: prepare variables for case 1 discriminant
 *             - sameCov: prepare variables for case 2 discriminant
 *             - arbCov: prepare variables for case 3 discriminant
 *
 * Author: William Willie Wells
 *
 * Created: September 2013
 *
 * Modified: October 2013
 *****************************************************************/

#include <fstream>
#include <cstdlib>
#include <cmath>
#include <ctime>
#include "PnE.h"

// Calculate Mahalanobis distance and return a 1-by-1 matrix
Matrix mDist(Matrix &mux, Matrix &sigma){
        Matrix temp;
        Matrix tmd,tS,tXk;

        // initiate matrices
    temp.createMatrix(mux.getRow(),1);
    tmd.createMatrix(1,1);
    tXk.createMatrix(sigma.getCol(),1);
    tS=inverse(sigma);

    // compute Mahalanobis distance squared / -2
    for(int i=0;i<mux.getRow();i++){
                for(int j=0;j<mux.getCol();j++){
                        tXk(j,0)=mux(i,j);
                }
                tmd=transpose(tXk)->*tS->*tXk;
        temp(i,0)=(-0.5)*tmd(0,0);
    }

    return temp;
}

// Calculate multivariate probability density function, return pdf
Matrix MaP(Matrix &xT, Matrix &x){
        Matrix temp, temp1, mu, sigma;
    /*const*/ double pi=3.14159265359793;
    double c_g;

        // calculate mean and covariance of training set
        mu=mean(xT,xT.getCol());
        sigma=cov(xT,xT.getCol());

    // calculate Gausian constant in multivariate
    c_g=1/sqrt(2*pi);
    for(int d=0;d<sigma.getCol()-2;d++){
        c_g*=c_g;
    }
    c_g/=sqrt(det(sigma));

    // calculate x-mu
    temp1.createMatrix(x.getRow(),x.getCol());
    for(int i=0;i<x.getRow();i++){
        for(int j=0;j<x.getCol();j++){
                        temp1(i,j)=x(i,j)-mu(j,0);
        }
    }

    // calculate pdf (probability density function)
    temp=mDist(temp1,sigma);
    for(int k=0; k<temp.getRow(); k++){
         for(int y=0; y<temp.getCol(); y++){
                         temp(k,y)=exp(temp(k,y))*c_g;
                 }
    }

    return temp;
}

// Compare test samples against 2 classes pdf's and  classify test samples
void comparePdf(Matrix &p1,Matrix &p2,Matrix &xTest,double Px1){

    // compare test sample against decision boundary of p1-p2=0
        for(int i=0;i<xTest.getRow();i++){
                if(p1(i,0)*Px1>=p2(i,0)*(1-Px1)){
                        xTest(i,xTest.getCol()-1)=0;
                }else{
                        xTest(i,xTest.getCol()-1)=1;
                }
        }

}

// Classify test data using Maximum a-Posteriori based on training set data
Matrix MaPClass(Matrix &xTrain, Matrix &xTest, double Px1){
        Matrix temp,xTe,xTr1,xTr2,px1,px2;

        // split training set data into two classes
        xTr1=splitData(xTrain,0);
        xTr2=splitData(xTrain,1);

        // create pdf's using training set mu and sigma, and test set data
        xTe=subMatrix(xTest,0,0,xTest.getRow()-1,xTest.getCol()-2);
        px1=MaP(xTr1,xTe);
        px2=MaP(xTr2,xTe);

        // classify test data using MaP comparison
        temp.createMatrix(xTest.getRow(),xTest.getCol()+1);
        copyMat(xTest,temp);
        comparePdf(px1,px2,temp,Px1);

        return temp;
}

// Find prior probability that maximizes classification accuracy
double maxPrior(Matrix &xTrain, Matrix &xTest,double n){
        Matrix temp,mc;
        double maxP=0.0;
        double mP=0.0;

        // classify test data using MAP, find individual accuracies,
        // and find maximum
        temp.createMatrix(n-1,1);
        for(int i=0;i<n-1;i++){
                mc=MaPClass(xTrain,xTest,(i+1)/n);
                temp(i,0)=wAccuracy(mc,xTrain.getCol());
```

```
            if(temp(i,0)>maxP){
                maxP=temp(i,0);
                mP=(i+1)/n;
            }
        }

        return mP;
}

// Classify a test set using MAP, find and print accuracy
Matrix outEval(Matrix &xTrain, Matrix &xTest, double n){
        clock_t t0=clock();
        Matrix rE,mA,mT,xRes;

        // classify test set
        xRes=MaPClass(xTrain,xTest,n);

        // find accuracy
        mA.createMatrix(1,1);
        mA(0,0)=wAccuracy(xRes,xRes.getCol()-1);

        // find run time
        mT.createMatrix(1,1);
        mT(0,0)=execTime(t0);

        // evaluate a two class decision rule
        rE=ruleEval(xRes);

        // output results
        rE=formatResult(rE,mA);
        rE=formatResult(rE,mT);

        return transpose(rE);
}

// Compute transpose(mu)*transpose(inverse(covariance))*x-vector
// and add x-vector Mahalanobis distance if applicable
Matrix muSigmaX(Matrix &muSigma, Matrix &xTest, Matrix &tc, Matrix &tXmd){
        Matrix temp;
    Matrix tRes,tResX,tXk;

    // initiate matrices
    temp.createMatrix(xTest.getRow(),xTest.getCol()+1);
        copyMat(xTest,temp);
    tRes.createMatrix(xTest.getRow(),muSigma.getRow());
    tXk.createMatrix(muSigma.getCol(),muSigma.getRow());

    // compute mu*sigma*x or mu*x
    for(int i=0;i<xTest.getRow();i++){
            for(int j=0;j<xTest.getCol();j++){
                tXk(j,0)=xTest(i,j);
        }
        tResX=muSigma->*tXk+tc;
        tRes(i,0)=tResX(0,0)+tXmd(i,0);
        if(tRes(i,0)<0){
                temp(i,xTest.getCol())=1;
            }
        }

    return temp;
}

// Calculate for case 1 multiple of identity matrix covariance g_1(x)-g_2(x)
Matrix sameSigma(Matrix &xTrain, Matrix &xTest){
    Matrix tC,tMix,tEmpty,xTr1,xTr2,xTe,mu1,mu2;

        // split training set data into two classes
        xTr1=splitData(xTrain,0);
        xTr2=splitData(xTrain,1);
        // calculate the mean of each class
        xTe=subMatrix(xTest,0,0,xTest.getRow()-1,xTest.getCol()-2);
        mu1=mean(xTr1,xTr1.getCol());
        mu2=mean(xTr2,xTr2.getCol());

    // calculate constants
    tC=transpose(mu1)->*mu1-transpose(mu2)->*mu2;
    tC(0,0)*=-0.5;

    // transpose of the difference between means
    tMix=(transpose(mu1)-transpose(mu2));

        // initiate empty (zero) x-vector Mahalanobis distance
        // and calculate g_1(x)-g_2(x)
    tEmpty.createMatrix(xTe.getRow(),1);
    return muSigmaX(tMix,xTe,tC,tEmpty);
}

// Calculate for case 2 same covariances g_1(x)-g_2(x)
Matrix sameCov(Matrix &xTrain, Matrix &xTest){
    Matrix tMix,tS,tC,tMu1,tMu2,tEmpty,xTr1,xTr2,xTe,mu1,mu2,sigma;

        // split training set data into two classes
        xTr1=splitData(xTrain,0);
        xTr2=splitData(xTrain,1);

        // calculate the mean of each class and the covariance of the entire set
        xTe=subMatrix(xTest,0,0,xTest.getRow()-1,xTest.getCol()-2);
        mu1=mean(xTr1,xTr1.getCol());
        mu2=mean(xTr2,xTr2.getCol());
        sigma=cov(xTrain,xTrain.getCol()-1);

        // Mahalanobis distance for each mean
    tMu1=transpose(mu1);
    tMu2=transpose(mu2);
    tC=mDist(tMu1,sigma)-mDist(tMu2,sigma);

    // differnce of means*single inverse covariance
    tS=inverse(sigma);
    tMix=(transpose(mu1)-transpose(mu2))->*transpose(tS);

        // initiate empty (zero) x-vector Mahalanobis distance
        // and calculate g_1(x)-g_2(x)
    tEmpty.createMatrix(xTe.getRow(),1);
    return muSigmaX(tMix,xTe,tC,tEmpty);
}

// Calculate for case 3 differing, arbitrary, covariances g_1(x)-g_2(x)
Matrix arbCov(Matrix &xTrain, Matrix &xTest){
    Matrix tMix,tC,tX;
    Matrix tS,tS1,tS2,tMu,tMu1,tMu2,xTr1,xTr2,xTe,mu1,mu2,s1,s2;

        // split training set data into two classes
        xTr1=splitData(xTrain,0);
        xTr2=splitData(xTrain,1);

        // calculate the mean and covariance of each class
        xTe=subMatrix(xTest,0,0,xTest.getRow()-1,xTest.getCol()-2);
        mu1=mean(xTr1,xTr1.getCol());
        mu2=mean(xTr2,xTr2.getCol());
        s1=cov(xTr1,xTr1.getCol());
        s2=cov(xTr2,xTr2.getCol());

    // Mahalanobis distance for each covariance (x, mean)
    tX=mDist(xTe,s1)-mDist(xTe,s2);
    tMu1=transpose(mu1);
    tMu2=transpose(mu2);
    tMu=mDist(tMu1,s1)-mDist(tMu2,s2);

    // calculate constants
```

```
   tS.createMatrix(1,1);
   tS(0,0)=(det(s1))/(det(s2));
   tC.createMatrix(1,1);
   tC(0,0)=(log(tS(0,0)))*(-0.5)+tMu(0,0);

   // differnce of individual means*inverse covariance
   tS1=inverse(s1);
   tS2=inverse(s2);
   tMix=(transpose(mu1))->*transpose(tS1)-(transpose(mu2))->*transpose(tS2);

        // calculate g_1(x)-g_2(x)
   return muSigmaX(tMix,xTe,tC,tX);
}

/*******************************************************************
 * dimRedux.cpp - dimensionality reduction
 *
 * Purpose: implement Principal Component Analysis and
 *                Fisher's Linear Discriminant.
 *
 *          - wPCA: reduce dimensions of a feature space
 *                      using eigenvalues
 *          - scatter: convert a covariance Matrix into a scatter Matrix
 *          - wFLD: reduce dimensions of a feature space to 1 dimension
 *
 * Author: William Willie Wells
 *
 * Created: September 2013
 *
 *
 *******************************************************************/

#include <fstream>
#include <cstdlib>
#include <cmath>
#include "PnE.h"

// Use PCA to derive a new set of basis, and represent the data
Matrix wPCA(Matrix &x,Matrix &S){
        Matrix temp,V,D;
        double sumS=0.0;
        double sumD=0.0;
        int n;

        // compute eigenvectors and eigenvalues:
                // inv(V)*S*V=D, V=eigenvectors, D=eigenvalues
        D.createMatrix(1,S.getCol());
        V.createMatrix(S.getRow(),S.getCol());
        jacobi(S,D,V);

        // rearrange eigenvectors and eigenvalues
        eigsrt(D,V);

        // sum(eigenvalues), .9*sum(eigenvalues) threshhold
        for(int i=0;i<S.getRow();i++){
                sumS+=D(0,i);
        }
        sumS*=.9;
        for(int j=V.getCol()-1;j>-1;j--){
                sumD+=D(0,j);
                if(sumD>sumS){
                        n=j;
                        break;
                }
        }

        // create sub matrix of principal components, W=subM(V)
        V=subMatrix(V,0,n,V.getRow()-1,V.getCol()-1);

        // x*W = new data set, x = test set covariance of x!=S
        temp=subMatrix(x,0,0,x.getRow()-1,x.getCol()-2);

        return temp->*V;
}

// Convert covariance Matrix into a scatter Matrix
void scatter(Matrix &S){

        // multiply covariance matric by n-1
        for(int i=0;i<S.getRow();i++){
                for(int j=0;j<S.getCol();j++){
                        S(i,j)*=(double)(S.getRow()-1);
                }
        }
}

// Use FLD method to derive the projection direction
// that best separates the data
Matrix wFLD(Matrix &x,Matrix &ruleX){
        Matrix t1,tx,x1,x2,mu1,mu2,S1,S2,Sw;

        // call a function to separate data into a class, twice (number of classes)
        x1=splitData(ruleX,0);
        x2=splitData(ruleX,1);

        // find individual class means
        mu1=mean(x1,x1.getCol());
        mu2=mean(x2,x2.getCol());

        // find scatter matrices using covariance and # of samples
        S1=cov(x1,x1.getCol());
        scatter(S1);
        S2=cov(x2,x2.getCol());
        scatter(S2);

        // find inverse of sum of scatter matrices
        Sw=S1+S2;
        Sw=inverse(Sw);

        // w=inv(Scat)*sum((mu1-mu2)
        t1=Sw->*(mu1-mu2);

        // *generate projected data, for test(x) only step x used, else ruleX used
        tx=subMatrix(x,0,0,x.getRow()-1,x.getCol()-2);

        return tx->*t1;
}
```

```
x=[0.740964];
y=[0.774096];%unmod
z=[0.777108];%pca
w=[0.801205];%fld
k=[1 3 5 7 9 11 13 15 17 19];
pf1=[0.704819 0.756024 0.746988 0.756024 0.76506 0.759036 0.746988 0.746988 0.75 0.753012];
pp1=[0.71988 0.762048 0.753012 0.774096 0.75 0.746988 0.753012 0.762048 0.756024 0.762048];
pf2=[0.704819 0.740964 0.743976 0.753012 0.76506 0.76506 0.76506 0.771084 0.762048 0.759036];
pf3=[0.710843 0.743976 0.73494 0.746988 0.771084 0.768072 0.756024 0.759036 0.762048 0.762048];
pf4=[0.707831 0.73494 0.73494 0.746988 0.759036 0.76506 0.759036 0.759036 0.759036 0.762048];
pf5=[0.716867 0.728916 0.740964 0.746988 0.756024 0.76506 0.759036 0.756024 0.759036 0.753012];
pp2=[0.728916 0.746988 0.73494 0.75 0.746988 0.743976 0.756024 0.76506 0.759036 0.777108];
pp3=[0.707831 0.731928 0.731928 0.737952 0.740964 0.743976 0.759036 0.76506 0.762048 0.774096];
pp4=[0.710843 0.722892 0.731928 0.73494 0.753012 0.75 0.762048 0.76506 0.771084 0.78012];
pp5=[0.713855 0.7168670.728916 0.725904 0.753012 0.743976 0.743976 0.771084 0.768072 0.78012];
ppf1=[0.731928 0.753012 0.756024 0.771084 0.78012 0.76506 0.768072 0.76506 0.75 0.771084];
ppf2=[0.75 0.786145 0.756024 0.762048 0.768072 0.771084 0.756024 0.756024 0.75 0.759036];
ppf3=[0.731928 0.771084 0.768072 0.768072 0.76506 0.771084 0.76506 0.756024 0.75 0.743976];
ppf4=[0.740964 0.746988 0.759036 0.76506 0.774096 0.756024 0.762048 0.746988 0.753012 0.756024];
ppf5=[0.737952 0.759036 0.768072 0.771084 0.759036 0.75 0.762048 0.759036 0.743976 0.762048];
ppp1=[0.614458 0.644578 0.653614 0.674699 0.677711 0.668675 0.701807 0.680723 0.665663 0.674699];
ppp2=[0.60241 0.650602 0.683735 0.686747 0.704819 0.677711 0.674699 0.686747 0.677711 0.665663];
```

```
ppp3=[0.608434 0.650602 0.680723 0.692771 0.701807 0.698795 0.686747 0.677711 0.668675 0.674699];
ppp4=[0.614458 0.656627 0.686747 0.695783 0.704819 0.707831 0.692771 0.683735 0.668675 0.683735];
ppp5=[0.611446 0.656627 0.686747 0.692771 0.71988 0.710843 0.698795 0.689759 0.674699 0.671687];
ff1=[0.710589 0.702812 0.676854 0.664127 0.643575 0.657461 0.667568 0.671265 0.661216 0.645133];
ff2=[0.691196 0.69925 0.65714 0.63078 0.641944 0.630221 0.643332 0.630875 0.633468 0.624067];
ff3=[0.688171 0.702128 0.66109 0.630095 0.624855 0.618816 0.618138 0.594656 0.603543 0.588027];
ff4=[0.667341 0.701143 0.632843 0.628933 0.623645 0.621653 0.608741 0.581098 0.58479 0.56747];
ff5=[0.655245 0.689747 0.620243 0.613833 0.620598 0.612346 0.595778 0.580503 0.579145 0.575846];
pff1=[0.731928 0.771084 0.759036 0.756024 0.759036 0.771084 0.768072 0.771084 0.76506 0.771084];

figure(1)
plot(k,pf1,'b*',k,pf2,'rx',k,pf3,'g*',k,pf4,'c*',k,pf5,'k*',1,y,'m+');
title('Unmodified Pima');
xlabel('k');
ylabel('accuracy');
axis([0 20 0.7 0.78]);

figure(2)
plot(k,pp1,'b*',k,pp2,'rx',k,pp3,'g*',k,pp4,'c*',k,pp5,'k*',1,y,'m+');
title('Unmodified Pima Partial Dimension');
xlabel('k');
ylabel('accuracy');
axis([0 20 0.7 0.8]);

figure(3)
plot(k,ppf1,'b*',k,ppf2,'rx',k,ppf3,'g*',k,ppf4,'c*',k,ppf5,'k*',1,z,'m+');
title('PCA Pima');
xlabel('k');
ylabel('accuracy');
axis([0 20 0.72 0.8]);

figure(4)
plot(k,ppp1,'b*',k,ppp2,'rx',k,ppp3,'g*',k,ppp4,'c*',k,ppp5,'k*');
title('PCA Pima Partial Dimension');
xlabel('k');
ylabel('accuracy');
axis([0 20 0.6 0.73]);

figure(5)
plot(k,ff1,'b*',k,ff2,'rx',k,ff3,'g*',k,ff4,'c*',k,ff5,'k*');
title('Fglass');
xlabel('k');
ylabel('accuracy');
axis([0 20 0.56 0.72]);

figure(6)
plot(k,pff1,'b*',1,w,'k+')
title('FLD Pima');
xlabel('k');
ylabel('accuracy');
axis([0 20 0.73 0.81]);
```