Classification with Dimensionality Reduction and Performance Evaluation

Pattern Recognition/ECE 471

William Willie Wells
wwells4@utk.edu

Abstract

Dimensionality Reduction techniques were created to solve issues such as overfitting, which generates poor classification performance, and computational complexity that is inherent in problems involving higher dimensionality. Two different approaches to reduce the dimension of a training set while improving the accuracy of the classification model are Principal Component Analysis and Fisher's Linear Discriminant.   Principal Component Analysis is an algorithm the best represents the projected data set. Fisher's Linear Discriminant utilizes an algorithm the best separates, discriminates between classes, the data set. An algorithm utilizing the principal of maximum a posteriori and Gaussian probability density functions was used to classify the original and projected data sets. Prior to classification of the projected Principal Component Analysis and Fisher's Linear Discriminant data set a prior probability that maximized classification accuracy on the original training set was found. The algorithm that resulted in the highest classification accuracy, precision, and specificity for the training set used was Fisher's Linear Discriminant with respective values of 80.1%, 76.5%, and 91.5%. While classification of the original data set without dimensionality reduction and equal prior probabilities resulted in the highest classification sensitivity and recall of the training set used with a value of 61.5%.

Introduction:

Excessive dimensionality of high dimensional data may result in irrelevant features being used in the classification process. This is part of the curse of dimensionality inherent in analysis of high dimension feature spaces. Irrelevant features may contribute to noise that distorts the nature of the perfect fit curve in hyper space. On the other hand the number of training samples available might not be adequate enough to represent the exponential search space defined by the number of dimensions and possible values of each dimension. Thousands of training samples are required to generate an appropriate probability density function that accurately represents the data in higher dimension data spaces. Inappropriate probability density functions may just describe the specific training set and provide poor classification accuracy for other sets of the same type of data, commonly known as overfitting. The ability to classify a data set accurately increases as the number of features increases. In higher dimensional space the number of computations required to generate a classification grows exponentially with the space dimension. (1)(2)(3)(5)

Reduction of dimensions involves feature extraction, transformation of high dimensional space to a reduced dimension projection. Many linear and non-linear techniques to reduce dimensions exist. The most common linear method of dimensionality reduction is Principal Component Analysis. Principal Component Analysis uses a linear combination of features to project onto a reduced dimension space. The eigenvectors and eigenvalues of the correlation matrix of the data are computed. The vector of eigenvectors is then sorted by corresponding decreasing eigenvalues. The largest eigenvalue is the principal component. A lower dimension space is achieved by choosing a threshold value for the sum of the eigenvalues and keeping only the corresponding subset of eigenvectors. The data set is that multiplied by the subset of eigenvectors. Another linear approach to dimensionality reduction is to find a projection that best discriminates the data set which is what Fisher's Linear Discriminant does. For a two class data set Fisher's Linear Discriminant reduces a matrix of d dimensions into one that has only a single dimension using the sum of the individual class scatter matrices and the difference of the individual class means. (1)(2)(4)(6)

Use of a feature vector to assign an object to a category is the task of a classifier. Of the two general types of classification techniques, supervised classification is the one that utilizes a training set to define the decision rules. In pattern recognition problems the multivariate normal density is the most appropriate probability density function for a majority of cases. An object is assigned to a category by determining the category that results in the maximum of the probability density function multiplied by its prior probability. (1)(7)

Evaluation of the performance of a classification in a two category environment involves analysis of the number of true positive, true negative, false positive, and false negative classifications of the decision rule used. This numbers are then used to determine the sensitivity (recall), specificity, precision, and accuracy among other evaluation values. (8)

Technical Approach:

After the mean and covariance (C) of the training set is found Principal Component Analysis may be utilized. The eigenvectors (V) and eigenvalues (D, a diagonal matrix) of the covariance Matrix have this relationship: $V^{-1}CV = D$, $CV = DV$

The Matrix library of Hairong Qi provides the function jacobi(C,D,V) written by Xiaoling Wang based on "Numerical Recipe" to perform this operation. The resultant eigenvector (V) and eigenvalue (D) matrices are then sorted in ascending order such that:

$$D = [\lambda_1 < \lambda_2 < \cdots < \lambda_{n-1} < \lambda_n]$$

by the function eigsrt(D,V) also provided and written by those mentioned above. To find an error rate < 0.10, the λ_i are summed and the sum is multiplied by 0.9. The λ_i are summed again in descending order and once the value $0.9 * \sum \lambda_i$ is exceeded ($\sum \lambda_j$)the value of j is used to create a subset matrix of eigenvectors. This subset of eigenvectors (W) is then multiplied by the test set such that:

$$Y_j = W^T X$$

Thus, $X_{r,n}$ is projected to $Y_{r,j}$ by $W_{n,j}$.

With Fisher's Linear Discriminant the training set (X) is separated into two classes by the function splitData(X, 0 or 1). The means and scatter (S) matrices of the separated classes are then calculated:

$$\mu = (1/n) * \sum X_i, \ S = \sum (X_i - \mu) * (X_i - \mu)^T$$

The scatter matrices are summed and inverted:

$$S_w = S_1 + S_2$$

The resultant matrices S_w^{-1} is then multiplied by the difference of the class means

$$w = S_w^{-1} * (\mu_1 - \mu_2)$$

The test set is then multiplied by the vector w:

$$Y_{r,1} = X_{r,n} w_{n,1}$$

This result has a single dimension.

Four testing sets (nY: original testing set with equal prior probabilities, nY_m: original testing set with prior probabilities that maximize classification accuracy, tY: the result of performing Principal Component Analysis, and fY: the result of performing Fisher's Linear Discriminant) are then classified using Maximum a-Posteriori estimation:

$$\underset{\theta}{\operatorname{argmax}} \ P(\theta|\mathbf{x}) = \underset{\theta}{\operatorname{argmax}} \ P(\mathbf{x}|\theta) P(\theta).$$ (6)

Where the probability density function, P(x|θ), is a Gaussian, multivariate normal density.
True positive, true negative, false positive, and false negative values are added up. Sensitivity, specificity, precision, and recall respectively:
TP/(TP+FN), TN/(TN+FP), TP/(TP+FP), TP/(TP+FN) are calculated .

Experiments and Results

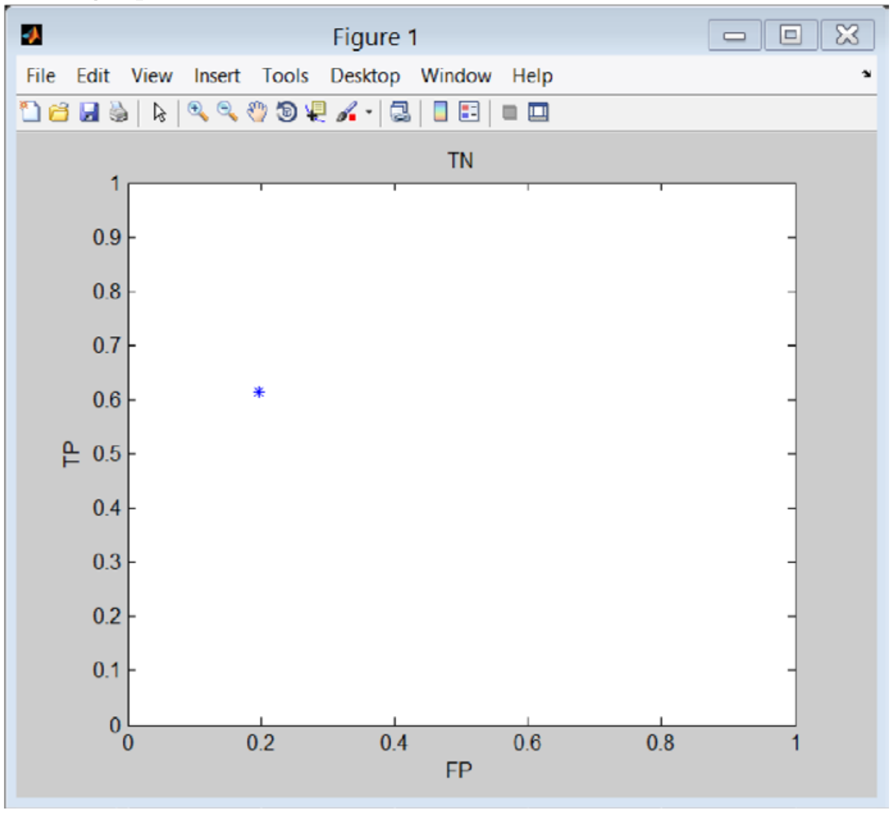
The value of the prior probabilities that maximize classification accuracy using 2 digits of precision are 73% and 27%, using 3 digits of precision are 72.7% and 27.3% negative and positive respectfully. For the Principal Component Analysis 2 features were able to be discarded. The seven dimension feature space was reduced to a five dimension feature space.
Performance evaluation of each classification method:

	TP	**TN**	**FP**	**FN**	**Sensitivity**	**Specificity**	**Precision**	**Recall**	**Accuracy**
nY	67	179	44	42	0.614679	0.802691	0.603604	0.614679	0.740964
nY*m*	62	195	28	47	0.568807	0.874439	0.688889	0.568807	0.774096
tY	64	194	29	45	0.587156	0.869955	0.688172	0.587156	0.777108
fY	62	204	19	47	0.568807	0.914798	0.765432	0.568807	0.801205

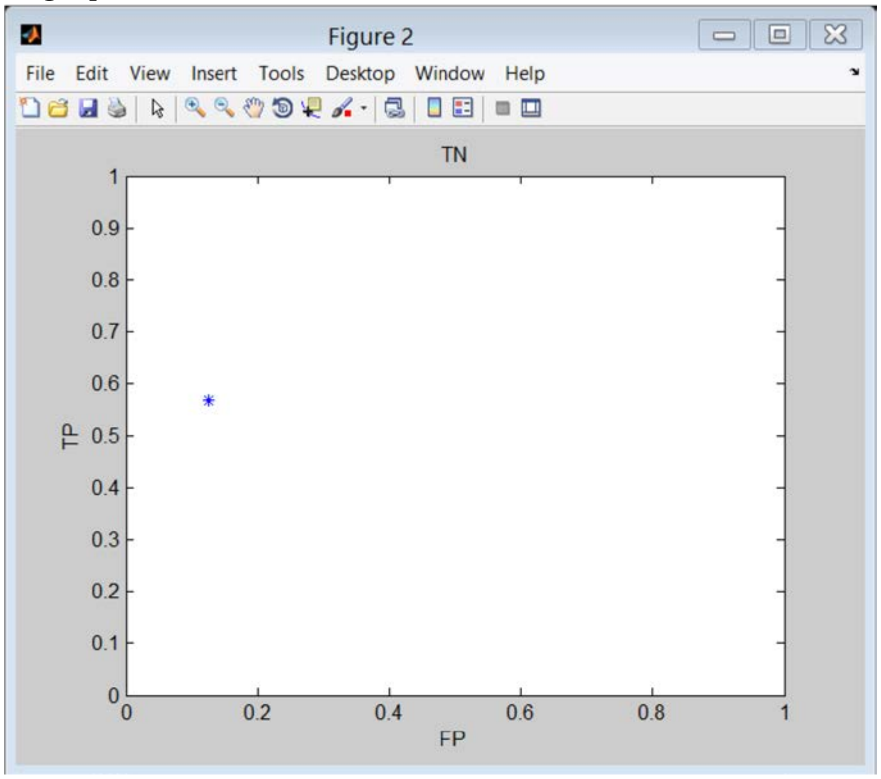
Rate values (0 < TP, TN, FP, FN < 1; TP + FN = TN + FP = 1)

	True Positive Rate	**True Negative Rate**	**False Positive Rate**	**False Negative Rate**
nY	0.614678899	0.802690583	0.197309417	0.385321101
nY*m*	0.568807339	0.874439462	0.125560538	0.431192661
tY	0.587155963	0.869955157	0.130044843	0.412844037
fY	0.568807339	0.914798206	0.085201794	0.431192661

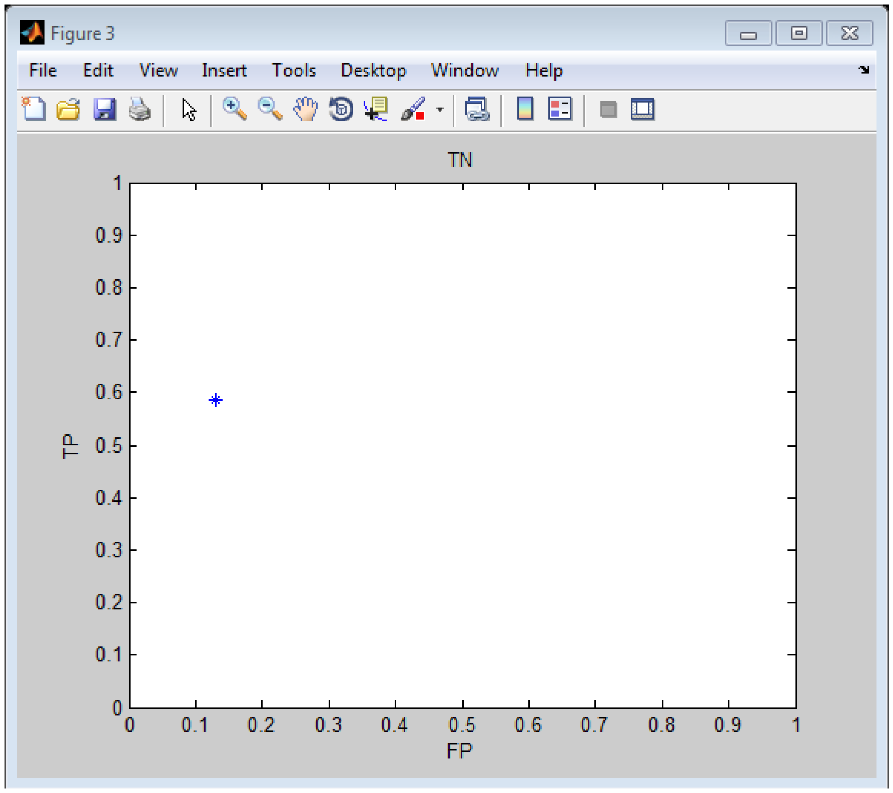
Single point ROC data for nY:



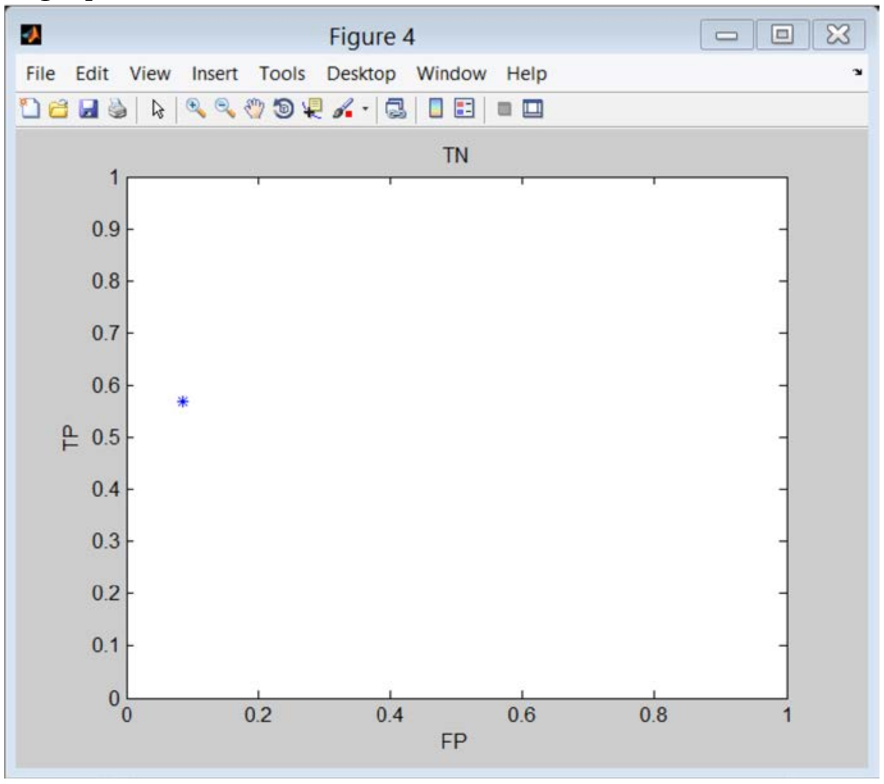
Single point ROC data for nY$_m$:



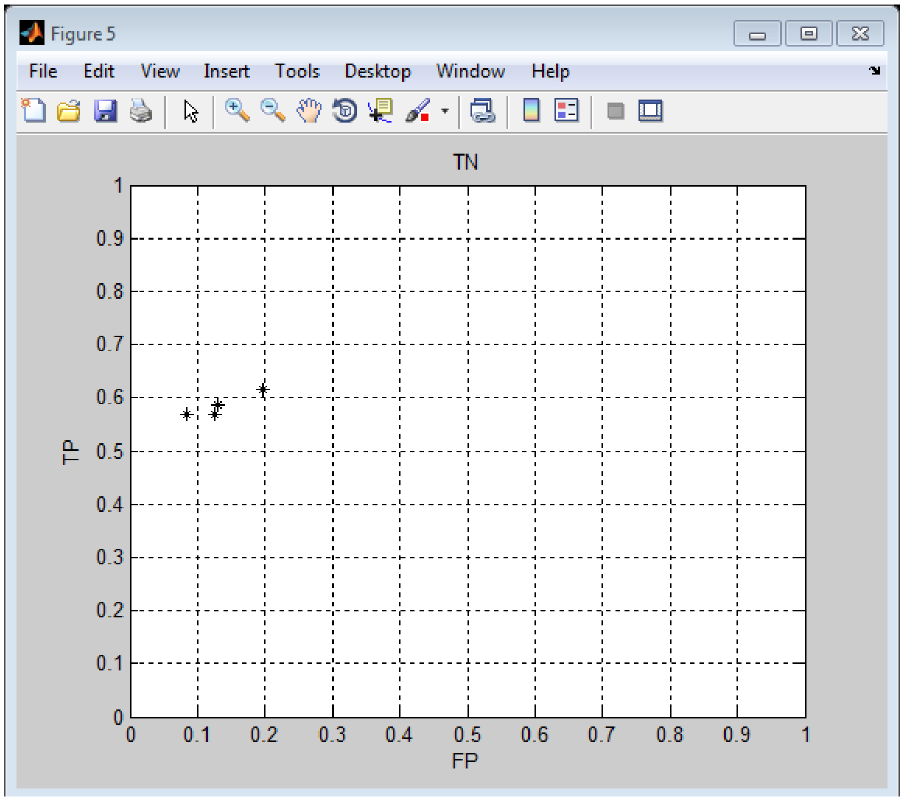
Single point ROC data for tY:



Single point ROC data for fY



Comparison of single point ROC values



Conclusion:

Fisher's Linear Discriminant has higher specificity, precision and accuracy values performing its purpose well, and discriminating data better than other comparative linear methods. Though Fisher's Linear Discriminant had a lower sensitivity (recall) than the equal prior probability original data case, it performed better that Principal Component Analysis in this category. The amount of false positives are minimized using FLD which in the case of a malignant disease could save on expenses and undue personal trauma. The improved prior probability only increased the correct number of classifications without dimensionality reduction by 11. Principal Component Analysis resulted in a slight improvement in performance from the original data set in classification of true positives and performed slightly worse in classification of true negatives (saving more lives) and thus had higher sensitivity and recall values as well as a higher overall accuracy rating.

References:

1 R. O. Duda, P. E. Hart, D. G. Stork, *Pattern Classification*, 2nd Edition, John Wiley, 2001.
2 http://web.eecs.utk.edu/~qi/ece471-571/lecture07_dimensionality_fld.pdf
3 http://en.wikipedia.org/wiki/Overfitting
4 http://en.wikipedia.org/wiki/Dimensionality_reduction
5 http://en.wikipedia.org/wiki/Curse_of_dimensionality
6 http://www.cmap.polytechnique.fr/~gaubert/PAPERS/ACC08Dist2.pdf
7 http://ceeserver.cee.cornell.edu/wdp2/cee6150/Lectures/DIP10_PatternRec_Sp11.pdf

8 http://en.wikipedia.org/wiki/Sensitivity_and_specificity

Appendix:

```
/**********************************************************
* proj2.cpp - implement dimensionality reduction
 *
* Purpose: Implement dimensionality reduction, classify,
        and perform evalutions on data sets.
*
* * Author: William Willie Wells
 *
* Created: September 2013
 *
 *
 **********************************************************/

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

using namespace std;

// Classification with Dimensionality Reduction and Performance Evaluation
int main(int argc, char **argv){
                int d;
                double mP;
                Matrix X,nX,mu,S,tX,fX,tXc,fXc;
                Matrix Y,nY,tY,fY,nYc,tYc,fYc,sE,oE;

                  /*Preprocess data*/
        // error check input format
        if(argc<3){
          cout << "file.tr # of columns in file file.te"<< endl;
                                exit(1);
        }

                // read in data
        d=atoi(argv[2]);
X = readData(argv[1],d);
   Y = readData(argv[3],d);

        // Find "No" in argv[1] and assign bit values to X(:,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);

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

        // normalize the data set so that features have the same scale
        xNorm(nX,mu,S);
xNorm(nY,mu,S);
S=cov(nX,nX.getCol()-1);

                                                /*Dimensionality Reduction*/
 // use Principal Component Analysis to derive a new set of basis, and represent the data
        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);

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

                // classify data with full dimension using MAP and find set of
        // prior probability that improves classification accuracy
        mP=maxPrior(nX,nY,1000.0);
cout << mP << endl;
sE=outEval(nX,nY,mP);
                oE=formatResult(oE,sE);

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

        // write output to file
        oE=transpose(oE);
                writeData(oE,"pima.txt");

                return 0;
}

/*******************************************************
*                               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 one dimension
 *
*                               Author: William Willie Wells
 *
*                               Created: September 2013
 *
 *
 *******************************************************/

#include <iostream>
#include <fstream>
#include <cstdlib>
#include <cmath>
#include "Matrix.h"
```

```
#include "Pr.h"
#include "mpp.h"

using namespace std;

// 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;
}

/***********************************************************************
*			mpp.cpp - functions to support data classification of d dimension,
*			c classes, and n samples
 *
*			Purpose: perform operations on training sets and test samples to   *		design classifiacation decision
			rules.
 *
*			- strToBit: find a user defined string and assigns it a binary
*			value and assign all else the opposite 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
*			- mDist: compute Mahalanobis distance
*			- MaP: calculate Gaussian Probability Density Function
*			- comparePdf: compare pdf values of 2 classes for test samples
*			- MaPClass: classify test data using Maximum a-Posteriori
*			- wAccuracy: calculate accuracy of classification
*			- maxPrior: find prior probability that maximizes classification accuracy
*			- ruleEval: calculate performance evaluation values
*			- outEval: output performance evaluation results
*			- muSigmaX: calculates Bayesian Discriminant Function
*			- sameSigma: prepares variables for case 1 discriminant
*			- sameCov: prepares variables for case 2 discriminant
*			- arbCov: prepares variables for case 3 discriminant
 *
*			Author: William Willie Wells
 *
*			Created: September 2013
 *
*			Modified: September 2013
 ***********************************************************************/

#include <iostream>
#include <fstream>
#include <cstdlib>
#include <cmath>
#include "Matrix.h"
#include "Pr.h"

using namespace std;

// 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 &x, Matrix &mu, Matrix &S){

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

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

    // find class identifier and copy row
temp.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++){
                                                       temp(count,j)=x(i,j);
                                             }
                                             count++;
                              }
                    }

                    return subMatrix(temp,0,0,count-1,temp.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 temp;
```

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

                return temp;
}

// 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;
}

// 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
}

// 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;
}

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

        // calculate total number of TP, FN, TN, and FP results
    temp.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){
    temp(0,0)+=1.0;// (1,1) TP
                        }else{
                                temp(0,2)+=1.0;// (0,1) FP
                    }
    }else{
                    if(xC(i,xC.getCol()-2)==1){
            temp(0,3)+=1.0;// (1,0) FN
                        }else{
                                temp(0,1)+=1.0;// (0,0) TN
                        }
                }
        }

    // calculate probability of TP, FN, TN, and FP
    temp(0,7)=temp(0,0)+temp(0,3);// TP + FN, all actual positives
    temp(0,0)/=temp(0,7);// 0 < TP < 1              temp(0,3)/=temp(0,7);//
0 < FN < 1      temp(0,7)=temp(0,1)+temp(0,2);// FP + TN, all actual
negatives       temp(0,1)/=temp(0,7);// 0 < TN < 1
        temp(0,2)/=temp(0,7);// 0 < FP < 1

    // determine sensitivity--recall, specificity, and precision
    temp(0,4)=temp(0,0);// sensitivity--recall      temp(0,5)=temp(0,1);//
specifity       temp(0,6)=temp(0,0)/(temp(0,0)+temp(0,2));// precision
    temp(0,7)=temp(0,0);// recall

        return temp;
}

// Classify a test set using MAP, find and print accuracy Matrix
outEval(Matrix &xTrain, Matrix &xTest, double n){
        double eR;
        Matrix rE,wE,eRm,xRes;

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

        // find and print accuracy
        eR=wAccuracy(xRes,xRes.getCol()-1);

        // evaluate decision rule
        rE=ruleEval(xRes);
        // output results
        eRm.createMatrix(1,1);
        eRm(0,0)=eR;
        wE=formatResult(rE,eRm);

        return transpose(wE);
}

// Computes transpose(mu)*transpose(inverse(covariance))*x-vector
// and adds 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);
}

% ROC.m
x=[0.614679 0.568807 0.587156 0.568807];
y=[0.197309 0.125561 0.130045 0.0852018];

figure(1)
plot(y(1,1),x(1,1),'b*');
title('TN'); xlabel('FP');
ylabel('TP');
axis([0 1 0 1]);

figure(2)
plot(y(1,2),x(1,2),'b*');
title('TN');
xlabel('FP');
ylabel('TP'); axis([0 1
0 1])

figure(3)
plot(y(1,3),x(1,3),'b*');
title('TN');
xlabel('FP');
ylabel('TP'); axis([0 1
0 1]);
```

```
figure(4)
plot(y(1,4),x(1,4),'b*');
title('TN');
xlabel('FP');
ylabel('TP'); axis([0 1
0 1])

figure(5)
plot(y,x,'k*');
axis([0 1 0 1]);
title('TN');
xlabel('FP');
ylabel('TP'); grid
on
```