**EECS**

**University of Tennessee**

**Pattern Recognition – ECE 571**

### Project 2 – Classification with Dimensionality Reduction and Performance Evaluation

**Submitted by:**

Mahendra Duwal Shrestha

[mduwalsh@vols.utk.edu](mailto:mduwalsh@vols.utk.edu)

**Abstract**

In pattern recognition, generally if the number of features in samples is higher, features are reduced using transformations such a way that its effect on classification accuracy is minimum or within tolerance. This process of reducing features or dimensions of data using transformation is called dimensionality reduction. This is preprocessing of data before using it for classification purpose and is practically significant in pattern recognition since it highly reduces complexity in analysis without affecting end result by much.

The objective of this project was to implement two different methods of dimensionality reduction methods, Principal Component Analysis (PCA) and Fisher’s Linear Discriminant (FLD) methods on training and testing data sets, classify transformed testing data set and evaluate performance of each method.

Both training set and testing set were first normalized using sample mean and standard deviation and each set was transformed using PCA and FLD methods separately. The transformed sets from each method were classified using Maximum Posterior Probability classifier and performance of classification was calculated for each data set from different transformations. Classifier performance for both transformation methods was evaluated calculating TP, TN, FP and FN rates. For further illustration, ROC curve was drawn for classification after each transformation.

**Introduction**

In pattern recognition process, dimensions of data are often reduced before imposing classification on the data. Theoretically higher dimensions yields less error and hence more accurate results but in real world problems it is quite often observed that opposite happens more. It is because we assume the distribution of samples is Gaussian which is only approximation and entirely not true. With higher number of dimensions, higher number of samples will be needed as training samples to model classifier decision rule with higher accuracy that can surpass reasonable number of training samples. With limited number of training samples with higher number of features, we may be overfitting the data, which may work brilliantly on training samples but give poor predictive performance on testing samples. So in practice, before performing classification on data set, number of dimensions is reduced.

This project used two distinguished methods of reducing dimensions of samples, namely Principal Component Analysis (PCA) and Fisher’s Linear Discriminant (FLD) methods. Each method is imposed on normalized sets of data, both training and testing and performance of classification after transformation using each method was evaluated. The performance of classifier after FLD transformation was found to be better than PCA but not by much though.

**Technical Approach**

The training and testing samples with seven dimensions were provided as pima.tr and pima.te respectively. Programs needed to be developed for implementing dimension reduction methods PCA and FLD. The programs were developed using C++. Samples in both training set and testing set were first normalized using sample mean and standard deviation from training set. Sample distribution was assumed Gaussian. The normalized training set was used to estimate parameters of Gaussian using maximum likelihood learning. MPP classifier was chosen to classify test samples and equal prior probabilities were assumed. Then all three cases of MPP classifier were tested on the normalized set and the method with highest accuracy was selected for further testing. And also prior probabilities for all classes were varied in order to find the probabilities that yield highest accuracy. Dimension reduction methods PCA and FLD both were imposed on training and testing sets. Classification was done on testing set after each method was imposed on normalized sets and performance of classification was evaluated using true positive (TP), true negative (TN), false positive (FP) and false negative (FN) rates.

**Fisher’s Liner Discriminant (FLD) method**

It is one of commonly used dimension reduction methods which tries to discriminate transformed data best possible way. This method tries to project mean vector of each class as further as possible and minimize scatterings of sample data within each class as best as possible. Since this uses class information of training samples, it is supervised learning. Using this method, number of dimensions d can be reduced to c-1 where c is number of classes and is supposed less than d. If we consider two class case, then data of d dimension is projected onto a line. So we want to find projection vector ‘**w**’ such that the data can be best separated. And projected data points are given by,



And



is within class scatter matrix and given by



And **m1** and **m2** are mean of each class.

**Principal Component Analysis (PCA) method**

It is also known as K-L transform. This method tries to represent the data in best possible way. It finds a new feature space (m- dimensional) that is sufficient to describe data in original feature space where m<d.

A vector **x** described in terms of a set of basis vectors **b***i*.



The basis vectors (**b***i*) should be linearly independent and orthonormal, that is,



If we want to consider m (m<d) components of **y** and still represent **x** though with some error, we will calculate the first *m* elements of **y** and replace the others with constants.



Error:

Instead of plain error, mean square of error is used to quantify error and can be expressed as



Eigenvectors of covariance matrices of all classes are optimal basis vectors. Eigenvectors can be sorted corresponding to its eigenvalues in descending order and eigenvector with largest eigenvalue is principal component. Dimensions of data can be reduced to m from d by omitting eigenvectors corresponding to eigenvalues whose total sum (summing from smallest to larger eigenvalue) is not than error tolerance.

**Performance parameters**

Sensitivity

It is probability of occurrence of true positive out of actual positive and given by TP/(TP+FN).

Specificity

It is probability of occurrence of true negative out of actual negative and given by TN/(TN+FP).

Precision

It is probability of true positive out of apparent positive and given by TP(TP+FP).

Recall

It is same as sensitivity.

Accuracy

It is probability of correct decision and given by (TP+TN)/(TP+TN+FP+FN).

**Experiments and Results**

**Normalized data set**

The given training set and test set were modified by replacing ‘yes’ and ‘no’ with ‘1’ and ‘0’. The modified data sets were normalized using sample mean and standard deviation for each feature in training set. The distribution for data set was assumed Gaussian.

**Classification with MPP cases:**

The normalized test set was classified using all three cases of MPP with equal prior probabilities for both classes. Accuracy with each cases were

Case I -> 0.259036

Case II -> 0.231928

Case III -> 0.259036

Since case II classifier gave the highest accuracy the case II classifier was chosen for further testing.

**Prior probability adjustment**

Prior probabilities were varied from 0.1 to 0.9 in step of 0.1 for each set where these are complementary, i.e., Pw1=(1 - Pw0). Highest accuracy was found for Pw0=0.6 and Pw1=0.4 which is 0.192771.

So the testing of PCA and FLD were carried out using case II MPP classifier with prior probabilities 0.6 and 0.4.

**PCA**

For the maximum error rate 0.10, the dimension of data set was reduced to 5 from 7. The training and testing set both were transformed to the reduced dimension using PCA and then classified using case II MPP classifier and prior probabilities mentioned above. Accuracy of classification was found to be 0.210843.

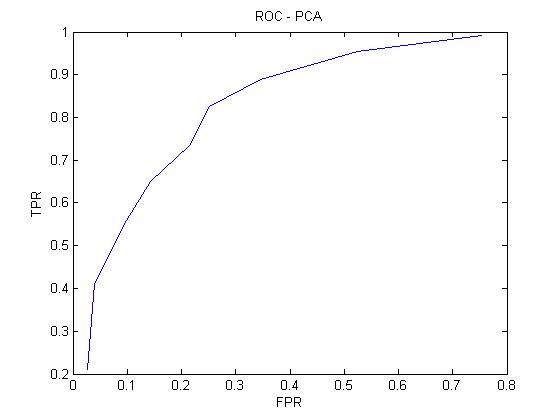
TP=71; TN=191; FP=32; FN=38

Sensitivity or Recall = 0.651376

Specificity = 0.856502

Precision = 0.68932

In order to vary TP rate and FP rate, prior probabilities were varied and ROC curve was plotted for PCA method which is shown in figure below.



**FLD**

After FLD transformation, dimension was reduced to 1 which is equal to one less than number of classes. The transformed data set was classified and accuracy was found to be 0.192771.

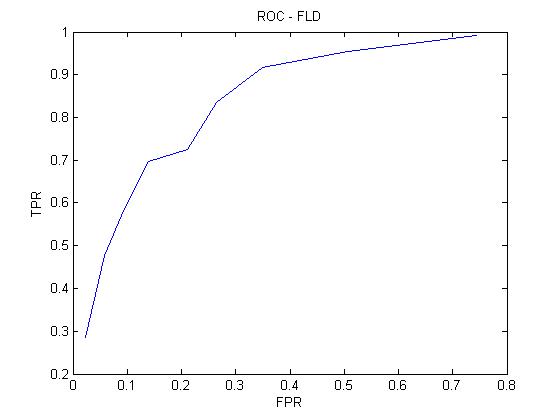
TP=76; TN=192; FP=31; FN=33

Sensitivity or Recall = 0.697248

Specificity = 0.860987

Precision = 0.71028

ROC curve was plotted using similar method as in PCA which is shown below.



**Discussion**

In this project, we’ve implemented both dimensionality reduction methods, PCA and FLD and evaluate performance of classification after respective transformations. PCA method reduced dimensions from 7 to 5 while FLD reduced it to 1 since number of classes was 2. PCA seemed to decrease performance by some factor which is under allowed error rate while FLD didn’t decrease performance. So FLD gives good performance under classification while PCA best represents original data set. Area above ROC curve in top left corner for each method also signifies that FLD performs classification better than PCA because the area for FLD is smaller than PCA’s.

**References**

* Lectures notes from class ECE 471/571 Pattern Recognition (Prof. Qi)

**Appendix**

**/include/Pr.h**

/\*

\* pr.h - header file of the pattern recognition library

\*

\* Author: Hairong Qi, ECE, University of Tennessee

\*

\* Date: 01/25/04

\*

\* Please send all your comments to hqi@utk.edu

\*

\* Modified:

\* - 09/24/13: add "const" to the filename parameters to remove warning

\* msg in new compilers (Steven Clukey)

\* - 04/26/05: reorganized for the Spring 2005 classs

\*/

#ifndef \_PR\_H\_

#define \_PR\_H\_

#include "Matrix.h"

#include "Estimate.h"

/////////////////////////

// file I/O

Matrix readData(const char \*, // the file name

int); // the number of columns of the matrix

Matrix readData(const char \*, // the file name

int, // the number of columns

int); // the number of rows (or samples)

Matrix readData(const char \*); // read data file to a matrix with 1 row

void writeData(Matrix &, const char \*); // write data to a file

Matrix readImage(const char \*, // read the image from a file

int \*, // the number of rows (or samples)

int \*); // the number of columns

void writeImage(const char \*, // write the image to a file

Matrix &, // the matrix to write

int, // the number of rows

int); // the number of columns

////////////////////////

// distance calculation

double euc(const Matrix &, // Euclidean distance between two vectors

const Matrix &);

double mah(const Matrix &, // the Mahalanobis distance, input col vec

const Matrix &C, // the covariance matrix

const Matrix &mu); // the mean (a col vector)

// Estimates calculation

Estimate estimateCalculation(const Matrix &train, int c);

////////////////////////

// classifiers

// maximum a-posteriori probability (mpp)

int mpp(const Matrix &train, // the training set of dimension mx(n+1)

// where the last col is the class label

// that starts at 0

const Matrix &test, // one test sample (a col vec), nx1

const int, // number of different classes

const int, // caseI,II,III of the discriminant func

const Matrix &Pw); // the prior prob, a col vec

// likelyhood ratio (lr)

int lr(Matrix \*mu, Matrix \*cov, Matrix &teData, int classes, Matrix &Pw);

// preprocessing

void normalize(Matrix &tr, Matrix &te, const int nf, const int flag);

// dimension reduction

int pca(Matrix &tr, Matrix &te, const int nf, const float err, const int flag); // dimension reduction using PCA

int fld(Matrix &tr, Matrix &te, int classes, int nf, int flag); // dimension reduction by using FLD

#endif

**/lib/preprocessing.cpp**

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* preprocessing.cpp

\*

\* - normalize: normalize training and test set

\* - pca: principal component analysis

\* - fld: fisher's linear discriminant

\*

\* Author: Hairong Qi (C) hqi@utk.edu

\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#include "Matrix.h"

#include "Pr.h"

#include <iostream>

#include <cstdlib>

#include <cmath>

using namespace std;

/\*\*

\* Matrix normalization.

\* @param tr The training set.

\* @param te The test set.

\* @param nf The number of features.

\* @param flag If flag is on, it's supervised learning; otherwise, it's

\* unsupervised learning and the second argument can be empty.

\*/

void normalize(Matrix &tr, Matrix &te, const int nf, const int flag)

{

Matrix mu, Sigma, sigma;

// get the statistics from the training set

mu = mean(tr, nf);

Sigma = cov(tr, nf);

sigma.createMatrix(nf,1);

for (int j=0; j<nf; j++)

sigma(j,0) = sqrt(Sigma(j,j));

// normalize the training set

for (int i=0; i<tr.getRow(); i++) {

for (int j=0; j<nf; j++)

tr(i,j) = (tr(i,j)-mu(j,0)) / sigma(j,0);

}

// normalize the test set

if (flag) {

for (int i=0; i<te.getRow(); i++) {

for (int j=0; j<nf; j++)

te(i,j) = (te(i,j)-mu(j,0)) / sigma(j,0);

}

}

}

/\*\*

\* Principal component analysis.

\* @param tr The training set.

\* @param te The test set.

\* @param nf The number of features.

\* @param err The error rate needs to be satisfied.

\* @param flag If flag is on, it's supervised learning; otherwise, it's

\* unsupervised learning and the second argument can be empty.

\* @return The number of features after PCA based on "err"

\*/

int pca(Matrix &tr, Matrix &te, const int nf, const float err, const int flag)

{

Matrix Sigma, temp;

Matrix d(1,nf), // eigenvalue (a row vector)

V(nf,nf), // eigenvector with each col an eigenvector

pV; // eigenvectors selected based on "err"

int p, pnf;

float psum, sum;

Sigma = cov(tr, nf);

jacobi(Sigma, d, V);

eigsrt(d, V); // sort the eigenvalue in the ascending order

// determine the number of principal components to keep based on "err" given

sum = 0.0;

for (int i=0; i<nf; i++)

sum += d(0,i);

p = 0;

psum = 0.0;

while (psum/sum < err && p < nf) {

psum += d(0,p);

p++;

}

p--;

pnf = nf - (p);

pV = subMatrix(V,0,p,nf-1,nf-1);

// perform the transformation

for (int i=0; i<tr.getRow(); i++) { // for training set

temp = subMatrix(tr,i,0,i,nf-1);

temp = temp->\*pV;

for (int j=0; j<pnf; j++)

tr(i,j) = temp(0,j);

}

if (flag) {

for (int i=0; i<te.getRow(); i++) { // for test set

temp = subMatrix(te,i,0,i,nf-1);

temp = temp->\*pV;

for (int j=0; j<pnf; j++)

te(i,j) = temp(0,j);

}

}

return pnf;

}

/\*\* Fisher's linear discriminant

\* @param tr The training set.

\* @param te The test set.

\* @param classes no. of classes for classification.

\* @param nf The number of features.

\* @param flag If flag is on, FLD is applied to both training set and testing set, else only to training set

\* @return The number of features after FLD

\*/

// for now works for two classes only

int fld(Matrix &tr, Matrix &te, int classes, int nf, int flag)

{

// calculate the mean and covariance of each class

// the mean is a cxnf matrix and the cov is a c\*nf x nf matrix

Matrix \*means = new Matrix [classes];

for (int i=0; i<classes; i++)

means[i].createMatrix(nf, 1);

Matrix \*covs = new Matrix [classes];

for (int i=0; i<classes; i++)

covs[i].createMatrix(nf, nf);

Matrix tmp;

for (int i=0; i<classes; i++) {

tmp = getType(tr, i);

covs[i] = cov(tmp, nf);

means[i] = mean(tmp, nf);

}

// calculate scatter matrices

int nr = tr.getRow();

Matrix S1,S2,Sw;

S1=covs[0]\*(nr-1);

S2=covs[1]\*(nr-1);

Sw=S1+S2;

// projection vector

Matrix w;

w=inverse(Sw)->\*(means[0]-means[1]);

// perform the transformation

Matrix temp;

int fnf=classes-1;

for (int i=0; i<tr.getRow(); i++) { // for training set

temp = subMatrix(tr,i,0,i,nf-1);

temp = temp->\*w;

for (int j=0; j<fnf; j++)

tr(i,j) = temp(0,j);

}

if (flag) {

for (int i=0; i<te.getRow(); i++) { // for test set

temp = subMatrix(te,i,0,i,nf-1);

temp = temp->\*w;

for (int j=0; j<fnf; j++)

te(i,j) = temp(0,j);

}

}

return fnf;

}

**/lib/mpp.cpp**

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\* mpp.cpp

\*\*

\*\* Purpose: Supervised learning:

\*\* Maximum posterior probability (MPP). This algorithm assumes

\*\* Gaussian distribution and zero-one loss

\*\*

\*\* Prototype: int mpp(Matrix train, Matrix test,

\*\* int class, int case, Matrix Pw)

\*\* - train: the training set of m x (n+1) matix

\*\* where m is the nr of rows (or samples)

\*\* n is the number of features

\*\* the last column is the class label, starting at 1

\*\* - test: the testing sample to be classified, a column vector

\*\* with a dimension nx1

\*\* - class: number of different classes, assuming the class labels

\*\* starts at 0

\*\* - case: 1, 2, 3 - case I, II or III

\*\* cases==1: The features are statistically

\*\* indepdent, and have the same variance.

\*\* Can be simplified to minimum distance.

\*\* cases==2: The covariance matrices for all classes are

\*\* identical, but not a scalar of identity matrix.

\*\* We pick the average.

\*\* cases==3: The most general case. The covariance matrices

\*\* are not equal from each other.

\*\* - Pw: Prior probability. A column vector of dimension cx1

\*\*

\*\* - output: the class label of the input test sample

\*\*

\*\* Created by: Hairong Qi (hqi@utk.edu)

\*\*

\*\* Modified by

\*\* - 02/20/2008: the class label starts at 0 instead of 1

\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#include <iostream>

#include <cstdlib>

#include <cmath>

#include "Pr.h"

using namespace std;

int mpp(const Matrix &train, const Matrix &tedata, const int c, const int cases, const Matrix &Pw)

{

static int first = 1, nf;

static double varavg;

static Matrix \*means, \*covs, covavg;

int nctr, ncte, nrtr, nrte;

int i, j;

Matrix covsum, tmp;

double sum;

//////////////////////////////////////////////////////////////////////

// calculate the means and covs only when the function is called the 1st time

if (first==1) {

// get the size of input raw data

nctr = train.getCol();

ncte = tedata.getCol();

nrtr = train.getRow();

nrte = tedata.getRow();

if (nctr != (nrte+1)) {

cout << "MPP: "

<< "Training and testing set do not have same number of features\n";

exit(3);

}

nf = nctr-1;

// calculate the mean and covariance of each class

// the mean is a cxnf matrix and the cov is a c\*nf x nf matrix

means = (Matrix \*) new Matrix [c];

for (i=0; i<c; i++)

means[i].createMatrix(nf, 1);

covs = (Matrix \*) new Matrix [c];

for (i=0; i<c; i++)

covs[i].createMatrix(nf, nf);

// the following two matrices are used for case 2

// the average covariance matrix is used as the common matrix

covsum.createMatrix(nf,nf);

covavg.createMatrix(nf,nf);

for (i=0; i<c; i++) {

tmp = getType(train, i);

covs[i] = cov(tmp, nf);

means[i] = mean(tmp, nf);

covsum = covsum + covs[i];

}

// calculate the average covariance to be used by case II

covavg = covsum / (double)c;

// calculate the average variance to be used by case I

sum = 0.0;

for (i=0; i<nf; i++)

sum += covavg(i,i);

varavg = sum / (double)nf;

first++;

}

//////////////////////////////////////////////////////////

// classification

Matrix disc(1,c), sdisc(1,c), pos(1,c);

double mdist, edist;

// find the discriminant function value

switch (cases) {

case 1:

for (i=0; i<c; i++) { // for each class

edist = euc(tedata, means[i]);

disc(0,i) = -edist\*edist/(2\*varavg) + log(Pw(i,0));

}

break;

case 2:

for (i=0; i<c; i++) {

mdist = mah(tedata, covavg, means[i]);

disc(0,i) = -0.5\*mdist\*mdist + log(Pw(i,0));

}

break;

case 3:

for (i=0; i<c; i++) {

mdist = mah(tedata, covs[i], means[i]);

disc(0,i) = -0.5\*mdist\*mdist - 0.5\*log(det(covs[i])) + log(Pw(i,0));

}

break;

}

// sort the discriminant function value in the ascending order

insertsort(disc, sdisc, pos);

// return the label of the class with the largest discriminant value

return (int)pos(0,c-1);

}

**/example/testNormalize.cpp**

#include <iostream>

#include <fstream>

#include <cmath>

#include <cstdlib>

#include "Matrix.h"

#include "Pr.h"

using namespace std;

#define Usage "Usage: ./testPCA training\_set test\_set classes features cases \n\t training\_set: the file name for training set\n\t test\_set: the file name for test set\n\t classes: number of classes \n\t features: number of features (dimension)\n\t"

int main(int argc, char \*\*argv)

{

int nrTr, nrTe, // number of rows in the training and test set

nc; // number of columns in the data set; both the

// training and test set should have the same

// column number

Matrix Tr, Te;

// check to see if the number of argument is correct

if (argc < 6) {

cout << Usage;

exit(1);

}

int classes = atoi(argv[3]); // number of classes

int nf = atoi(argv[4]); // number of features (dimension)

int cases=atoi(argv[5]); // case of MPP

// read in data from the data file

nc = nf+1; // the data dimension; plus the one label column

Tr = readData(argv[1], nc);

nrTr = Tr.getRow(); // get the number of rows

Te = readData(argv[2], nc);

nrTe = Te.getRow(); // get the number of rows

// normalization of data set

normalize(Tr, Te, nf, 1); // normalize sample data in training and test set both; flag set to 1 for testing set normalization

// assign prior probability

Matrix Pw(classes, 1);

//for (int i=0; i<classes; i++)

// Pw(i,0) = (float)1/classes; // if assuming equal prior probability

Pw(0,0)=0.6;Pw(1,0)=1-Pw(0,0); // found best for this value and case II of MPP

// prepare the labels and error count

Matrix labelMPP(nrTe,1); // a col vector to hold result for MPP

int errCountMPP = 0; // calcualte error rate for MPP

// to test for normalized data before dimension reduction method applied

// intialize TP,TN,FP,FN

int TP=0,TN=0,FP=0,FN=0;

// perform classification

for (int i=0; i<nrTe; i++) {

// classify one test sample at a time, get one sample from the test data

Matrix sample = transpose(subMatrix(Te, i, 0, i, nf-1));

// call MPP to perform classification

labelMPP(i,0) = mpp(Tr, sample, classes, cases, Pw);

// check if the classification result is correct or not

if (labelMPP(i,0) != Te(i,nf))

{

errCountMPP++;

if(Te(i,nf)==0)

{

FP++;

}

else{FN++;}

}

else

{

if(Te(i,nf)==0)

{

TN++;

}

else{TP++;}

}

}

// calculate accuracy

cout << "The error rate using MPP is = " << (float)errCountMPP/nrTe << endl;

cout<<"TP="<<TP<<endl;

cout<<"TN="<<TN<<endl;

cout<<"FN="<<FN<<endl;

cout<<"FP="<<FP<<endl;

cout<<"Sensitivity or Recall = "<<(TP/(float)(TP+FN))<<endl;

cout<<"Specificity = "<<(TN/(float)(TN+FP))<<endl;

cout<<"Precision = "<<(TP/(float)(TP+FP))<<endl;

return 0;

}

**/example/testPCA.cpp**

#include <iostream>

#include <fstream>

#include <cmath>

#include <cstdlib>

#include "Matrix.h"

#include "Pr.h"

using namespace std;

#define Usage "Usage: ./testPCA training\_set test\_set classes features cases \n\t training\_set: the file name for training set\n\t test\_set: the file name for test set\n\t classes: number of classes \n\t features: number of features (dimension)\n\t"

int main(int argc, char \*\*argv)

{

int nrTr, nrTe, // number of rows in the training and test set

nc; // number of columns in the data set; both the

// training and test set should have the same

// column number

Matrix Tr, Te;

// check to see if the number of argument is correct

if (argc < 6) {

cout << Usage;

exit(1);

}

int classes = atoi(argv[3]); // number of classes

int nf = atoi(argv[4]); // number of features (dimension)

int cases=atoi(argv[5]); // case of MPP

// read in data from the data file

nc = nf+1; // the data dimension; plus the one label column

Tr = readData(argv[1], nc);

nrTr = Tr.getRow(); // get the number of rows

Te = readData(argv[2], nc);

nrTe = Te.getRow(); // get the number of rows

// normalization of data set

normalize(Tr, Te, nf, 1); // normalize sample data in training and test set both; flag set to 1 for testing set normalization

Matrix sample0 = transpose(subMatrix(Te, 0, 0, 0, nf));

cout<<"1st row before pca = "<<sample0;

// PCA method

float err=0.1; // maximum error allowed

int flag=1; // to include Testing sample or not in normalization

int pnf=pca(Tr, Te, nf, err, flag); // apply pca dimension reduction

// now crop training and test set to reduced dimensions but keep label column

// first assign value of label column to column after pnf

for(int i=0;i<nrTr;i++)

{

Tr(i,pnf)=Tr(i,nf);

}

for(int i=0;i<nrTe;i++)

{

Te(i,pnf)=Te(i,nf);

}

Matrix pTr=subMatrix(Tr,0,0,nrTr-1,pnf);

Matrix pTe=subMatrix(Te,0,0,nrTe-1,pnf);

cout<<endl<<"The reduced dimension from PCA = "<<pnf<<endl;

Matrix sample1 = transpose(subMatrix(pTe, 0, 0, 0, pnf));

cout<<"1st row after pca = "<<sample1;

// assign prior probability

Matrix Pw(classes, 1);

//for (int i=0; i<classes; i++)

// Pw(i,0) = (float)1/classes; // if assuming equal prior probability

Pw(0,0)=0.6;Pw(1,0)=1-Pw(0,0); // found best for this value and case II of MPP

// prepare the labels and error count

Matrix labelMPP(nrTe,1); // a col vector to hold result for MPP

int errCountMPP = 0; // calcualte error rate for MPP

// intialize TP,TN,FP,FN

int TP=0,TN=0,FP=0,FN=0;

// perform classification

for (int i=0; i<nrTe; i++) {

// classify one test sample at a time, get one sample from the test data

Matrix sample = transpose(subMatrix(pTe, i, 0, i, pnf-1));

// call MPP to perform classification

labelMPP(i,0) = mpp(pTr, sample, classes, cases, Pw);

// check if the classification result is correct or not

if (labelMPP(i,0) != pTe(i,pnf))

{

errCountMPP++;

if(pTe(i,pnf)==0)

{

FP++;

}

else{FN++;}

}

else

{

if(pTe(i,pnf)==0)

{

TN++;

}

else{TP++;}

}

}

// calculate accuracy

cout << "The error rate using MPP is = " << (float)errCountMPP/nrTe << endl;

cout<<"TP="<<TP<<endl;

cout<<"TN="<<TN<<endl;

cout<<"FN="<<FN<<endl;

cout<<"FP="<<FP<<endl;

cout<<"Sensitivity or Recall or TPR = "<<(TP/(float)(TP+FN))<<endl;

cout<<"FPR = "<<(FP/(float)(TN+FP))<<endl;

cout<<"Specificity = "<<(TN/(float)(TN+FP))<<endl;

cout<<"Precision = "<<(TP/(float)(TP+FP))<<endl;

return 0;

}

**/include/testFLD.cpp**

// currently only works for 2 classes

#include <iostream>

#include <fstream>

#include <cmath>

#include <cstdlib>

#include "Matrix.h"

#include "Pr.h"

using namespace std;

#define Usage "Usage: ./testFLD training\_set test\_set classes features cases \n\t training\_set: the file name for training set\n\t test\_set: the file name for test set\n\t classes: number of classes \n\t features: number of features (dimension)\n\t"

int main(int argc, char \*\*argv)

{

int nrTr, nrTe, // number of rows in the training and test set

nc; // number of columns in the data set; both the

// training and test set should have the same

// column number

Matrix Tr, Te;

// check to see if the number of argument is correct

if (argc < 6) {

cout << Usage;

exit(1);

}

int classes = atoi(argv[3]); // number of classes

int nf = atoi(argv[4]); // number of features (dimension)

int cases=atoi(argv[5]); // case of MPP

// read in data from the data file

nc = nf+1; // the data dimension; plus the one label column

Tr = readData(argv[1], nc);

nrTr = Tr.getRow(); // get the number of rows

Te = readData(argv[2], nc);

nrTe = Te.getRow(); // get the number of rows

// normalization of data set

normalize(Tr, Te, nf, 1); // normalize sample data in training and test set both; flag set to 1 for testing set normalization

Matrix sample0 = transpose(subMatrix(Te, 0, 0, 0, nf));

cout<<"1st row before FLD = "<<sample0;

// FLD method

int flag=1; // to include testing sample or not in normalization, '1' means include yes

int fnf=fld(Tr, Te, classes, nf, flag); // apply pca dimension reduction

// now crop training and test set to reduced dimensions but keep label column

// first assign value of label column to column after fnf

for(int i=0;i<nrTr;i++)

{

Tr(i,fnf)=Tr(i,nf);

}

for(int i=0;i<nrTe;i++)

{

Te(i,fnf)=Te(i,nf);

}

Matrix fTr=subMatrix(Tr,0,0,nrTr-1,fnf);

Matrix fTe=subMatrix(Te,0,0,nrTe-1,fnf);

cout<<endl<<"The reduced dimension from FLD = "<<fnf<<endl;

Matrix sample1 = transpose(subMatrix(fTe, 0, 0, 0, fnf));

cout<<"1st row after FLD = "<<sample1;

// assign prior probability

Matrix Pw(classes, 1);

//for (int i=0; i<classes; i++)

//Pw(i,0) = (float)1/classes; // if assuming equal prior probability

Pw(0,0)=0.6;Pw(1,0)=1-Pw(0,0); // found best for this value and case II of MPP

// prepare the labels and error count

Matrix labelMPP(nrTe,1); // a col vector to hold result for MPP

int errCountMPP = 0; // calcualte error rate for MPP

// intialize TP,TN,FP,FN

int TP=0,TN=0,FP=0,FN=0;

// perform classification

for (int i=0; i<nrTe; i++) {

// classify one test sample at a time, get one sample from the test data

Matrix sample = transpose(subMatrix(fTe, i, 0, i, fnf-1));

// call MPP to perform classification

labelMPP(i,0) = mpp(fTr, sample, classes, cases, Pw);

// check if the classification result is correct or not

if (labelMPP(i,0) != fTe(i,fnf))

{

errCountMPP++;

if(fTe(i,fnf)==0)

{

FP++;

}

else{FN++;}

}

else

{

if(fTe(i,fnf)==0)

{

TN++;

}

else{TP++;}

}

}

// calculate accuracy

cout << "The error rate using MPP is = " << (float)errCountMPP/nrTe << endl;

cout<<"TP="<<TP<<endl;

cout<<"TN="<<TN<<endl;

cout<<"FN="<<FN<<endl;

cout<<"FP="<<FP<<endl;

cout<<"Sensitivity or Recall or TPR = "<<(TP/(float)(TP+FN))<<endl;

cout<<"FPR = "<<(FP/(float)(TN+FP))<<endl;

cout<<"Specificity = "<<(TN/(float)(TN+FP))<<endl;

cout<<"Precision = "<<(TP/(float)(TP+FP))<<endl;

return 0;

}