**EECS**

**University of Tennessee**

**Pattern Recognition – ECE 571**

### Project 3 – Classification Using Non-Parametric Density Estimation and Performance Evaluation

**Submitted by:**

Mahendra Duwal Shrestha

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

**Abstract**

In pattern recognition, classification can be progressed assuming data distribution follow certain pattern as function of some parameters or assuming data does not follow such pattern. The latter one is called non parametric learning. kNN is a non parametric learning where density function is estimated without any assumption that pdf follows particular pattern.

The objective of this project was to implement non parametric density function or kNN and then use it to classify test data and evaluate its performance against other implementation of classifiers.

kNN classifier was implemented in C++ where user can configure value of k and use of algorithm of computation of k nearest neighbors in training set. Majority voting was used for identifying class of a sample test data. Testing an entire set of test data set, classification accuracy was calculated and it was compared with others. Partial distance algorithm was also implemented in kNN to optimize execution time performance.

**Introduction**

In pattern recognition, kNN is a non parametric method used for classification purpose. It estimates density function for data distribution without assuming the probability distribution has any particular pattern.

kNN can be taken as instance based learning where the function is approximated locally and all essential computations are done at the time of classification. It is one of the simplest learning algorithms in pattern recognition. For classification purpose, it can be useful to take contributions from all neighbors so that nearer neighbors has more contributions than farther neighbors. A common weighting scheme can be used which gives weight of 1/d to a neighbor where d is distance to the neighbor.

This project used above weighting scheme for kNN classifier and was implemented as C++ program. Different values of k were used for classification however square root of number of samples is considered best value. Different metrics of distance were used to calculate distance between sample test data and neighbors such as Euclidean and Minkowski distance. Euclidean distance is just a case of Minkowski distance with order 2. The accuracy of classification was calculated and compared with others. However, Euclidean distance is commonly used distance metric.

kNN with partial distance calculation algorithm was also used for implementing kNN. Time performance of classification with kNN using partial distance was also compared with normal kNN implementation.

**Technical Approach**

**kNN(k- Nearest Neighbors)**

To estimate *p*(*x*) from *n* samples, we can center a cell at *x* and let it grow until it contains *kn* samples, and *kn* can be some function of *n*



Normally, we let



Given *c* training sets from *c* classes, the total number of samples is

Given a point **x** at which we wish to determine the statistics, we find the hypersphere of volume ***V*** which just encloses *k* points from the combined set. If within that volume, *km* of those points belongs to class *m*, then we estimate the density for class *m* by



Using Bayesian decision rule and posterior probability,



So from the above posterior probability expression, we can derive a decision rule which tells us to look in a neighborhood of the unknown feature vector for *k* samples. If within that neighborhood, more samples lay in class *i* than any other class, we assign the unknown as belonging to class *i*. This is using majority voting principle.

For distance calculation, most commonly used is Euclidean distance metric while we can also use Minkowski distance.

In the project, distance from a sample data to all data in training set was calculated and sorted in ascending order. Then picking only k of the nearest distances, majority of class of training data in k samples was identified.

**Euclidean Distance**

Euclidean distance is the "ordinary" distance between two points that one would measure with ruler. In Cartesian coordinates, if p = (*p*1, *p*2,..., *pn*) and q = (*q*1, *q*2,..., *qn*) are two points in Euclidean *n*-space, then the distance (d) from p to q, or from q to p is given by:

**Minkowski Distance**

The Minkowski distance of order *p* between two points

*P*=(*x*1,*x*2,…,*xn*) and *Q*=(*y*1,*y*2,…,*yn*)∈R*n*

is defined as:

If p=1, then it is Manhattan distance or city block distance and if p=2, then it is Euclidean distance.

**Partial Distance and kNN**

If ‘r’ is subset of full dimension‘d’, then partial distance of r dimensions is given by



Since subspace is indicative of full space ff this distance is too large then we don’t need to include further dimension in calculation. This can significantly decrease computation time of distance in kNN classifier algorithm. Full distance of k points only are calculated and then for others partial distance. The partial distance is larger than maximum of k distances calculated, then no need to add another dimension in distance calculation and move on to next point; else another dimension is brought in for distance calculation for same point. This can help in reducing computing time most of the time with large dimensions but it is not always guaranteed. It also saves storage space as only k distances are stored. However, if another neighbor distance is smaller than largest distance in storage, it needs to be replaced and sorted.

**Experiments and Results**

**Basic kNN and partial distance implementation**

Basic kNN with calculation of all distances from sample data to all of training samples has been implemented. Also kNN using partial distance has been implemented. Both of these can be configured from command line arguments while executing program.

But implementation of partial distance was twisted a bit for further more execution time optimization. Instead of calculating actual distance between two points, it calculates squared distances in case of Euclidean distance and p power distances in Minkowski distance. This reduced execution time significantly (time < normal partial distance) giving constant accuracy. Also instead of complete sorting array of distances of length k every time element of array got replaced, it is sorted only to identify maximum distance to end of array. And this also helped reducing execution time to some extent.

Both implementation were tested against pima data set using Euclidean distance metric and k=15 to measure accuracy and run time of whole program.

For basic kNN,

Accuracy=0.228916; runtime = 0.095921 sec.

For partial distance kNN,

Accuracy=0.228916; run time = 0.022662 sec.

This showed significance execution time reduction using partial distance.

**Experimentation with pima data set nX, tX and fX**

**For nX**,:

kNN with Euclidean distance

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | project 2 (best result) | k=1 | k=5 | k=15 |
| accuracy | 0.192771 | 0.295181 | 0.256024 | 0.228916 |
| run time (milli sec) | 15.796 | 13.578 | 17.297 | 23.334 |

The best accuracy found for kNN was at k=15 with 0.228916 but time taken seemed to increase with k. Performance result from project 2 seemed better for the data set.

kNN with Minkowski distance p=3

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | project 2 (best result) | k=1 | k=5 | k=15 |
| accuracy | 0.192771 | 0.28915 | 0.26506 | 0.240964 |
| run time (milli sec) | 15.796 | 10.56 | 15.316 | 21.594 |

The best accuracy found for kNN was at k=15 with 0.240964 but time taken was 21.594 millisecond. With increasing p, time taken for execute seemed to decrease but performance accuracy also degraded.

**For tX**,:

kNN with Euclidean distance

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | project 2 (best result) | k=1 | k=5 | k=15 |
| accuracy | 0.213855 | 0.310241 | 0.231928 | 0.256024 |
| run time (milli sec) | 4.178 | 7.831 | 11.629 | 16.672 |

The best accuracy found for kNN was at k=5 with 0.231928 and time taken was11.629.

kNN with Minkowski distance p=3

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | project 2 (best result) | k=1 | k=5 | k=15 |
| accuracy | 0.213855 | 0.310241 | 0.231928 | 0.259036 |
| run time (milli sec) | 4.178 | 7.729 | 10.945 | 16.714 |

The best accuracy found for kNN was at k=5 with 0.231928 and time taken was 10.945 millisecond.

**For fX**,:

kNN with Euclidean distance

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | project 2 (best result) | k=1 | k=5 | k=15 |
| accuracy | 0.192771 | 0.268072 | 0.240964 | 0.228916 |
| run time (milli sec) | 6.221 | 5.129 | 6.387 | 15.464 |

The best accuracy found for kNN was at k=15 with 0. 228916 and time taken was 15.464.

kNN with Minkowski distance p=3

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | project 2 (best result) | k=1 | k=5 | k=15 |
| accuracy | 0.268072 | 0.310241 | 0.240964 | 0.228916 |
| run time (milli sec) | 7.785 | 7.729 | 7.955 | 15.014 |

The best accuracy found for kNN was for k=15 with 0. 228916 and time taken was 15.014 millisecond.

For pima data set, MPP classifier seemed to perform better than kNN.

**Experimentation with fglass data set for cross validation (10 folds)**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | k=1 | k=5 | k=10 | k=15 |
| accuracy | 0.333333 | 0.4 | 0.4 | 0.4 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | k=1 | k=5 | k=10 | k=15 |
| accuracy | 0.333333 | 0.238095 | 0.333333 | 0.333333 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | k=1 | k=5 | k=10 | k=15 |
| accuracy | 0.285714 | 0.25 | 0.285714 | 0.285714 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | k=1 | k=5 | k=10 | k=15 |
| accuracy | 0.277778 | 0.444444 | 0.444444 | 0.5 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | k=1 | k=5 | k=10 | k=15 |
| accuracy | 0.411765 | 0.529412 | 0.529412 | 0.529412 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | k=1 | k=5 | k=10 | k=15 |
| accuracy | 0.192308 | 0.307692 | 0.384615 | 0.384615 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | k=1 | k=5 | k=10 | k=15 |
| accuracy | 0.363636 | 0.409091 | 0.454545 | 0.454545 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | k=1 | k=5 | k=10 | k=15 |
| accuracy | 0.333333 | 0.333333 | 0.375 | 0.333333 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | k=1 | k=5 | k=10 | k=15 |
| accuracy | 0.346154 | 0.269231 | 0.269231 | 0.346154 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | k=1 | k=5 | k=10 | k=15 |
| accuracy | 0.277778 | 0.333333 | 0.222222 | 0.333333 |

For these ten folds experiment with four different values of k, average is:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | k=1 | k=5 | k=10 | k=15 |
| average accuracy | 0.3155132 | 0.3514631 | 0.369852 | 0.390044 |

So, the best k for ‘fglass’ data set can be taken as 1.

**Discussion**

In this project, we’ve implemented basic implementation of kNN and partial distance method of kNN. Partial distance algorithm reduces computation time significantly in most of the cases but it does not necessarily happen so. And performance of kNN was measured and compared against MPP classifiers where in this case MPP seemed to outperform kNN algorithm.

**References**

* Lectures notes from class ECE 471/571 Pattern Recognition (Prof. Qi)
* <http://en.wikipedia.org/wiki/K-nearest_neighbors_algorithm>

**Appendix**

**/lib/distance.cpp**

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

\* distance.cpp - distance calculation

\*

\* - euc: Euclidean distance between two vectors

\* - mah: Mahalanobis distance between a sample and a cluster

\*

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

\*

\* Created: 02/17/08

\*

\* Modification:

\* - 10/25/13: add "const" to matrix arguments

\*

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

#include "Matrix.h"

#include "Pr.h"

#include <iostream>

#include <cstdlib>

#include <cmath>

using namespace std;

/\*\*

\* Calculate the Euclidean distance between two vectors.

\* @param x Vector x (row vector or col vector).

\* @param y Vector y (row vector or col vector).

\* @return Euclidean distance between x and y.

\*/

double euc(const Matrix &x, const Matrix &y)

{

int nrx, nry, ncx, ncy;

int i, j;

double sum = 0.0;

nrx = x.getRow();

nry = y.getRow();

ncx = x.getCol();

ncy = y.getCol();

if (!(nrx\*nry==1 || ncx\*ncy==1)) {

cout << "The euclidean() routine can only calculate distance of vectors\n";

exit(3);

}

if (nrx!=nry || ncx!=ncy) {

cout << "Two vectors are not of the same dimension\n";

exit(3);

}

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

for (j=0; j<ncx; j++)

sum += (x(i,j)-y(i,j)) \* (x(i,j)-y(i,j));

return sqrt(sum);

}

/\*\*

\* Calculate the Mahalanobis distance between a sample and a cluster.

\* @param x The sample (a column vector).

\* @param C The covariance matrix.

\* @param mu The mean (a column vector).

\* @return Mahalanobis distance between the sample and the cluster

\* characterized the mean and covariance.

\*/

double mah(const Matrix &x, const Matrix &C, const Matrix &mu)

{

int nrx, nrmu, nrC, ncx, ncmu, ncC;

// Matrix invC, diff, diffT, tmp, mdist;

nrx = x.getRow();

nrmu = mu.getRow();

nrC = C.getRow();

ncx = x.getCol();

ncmu = mu.getCol();

ncC = C.getCol();

if (ncx!=1 || ncmu!=1) {

cout << "Mahalanobis: "

<< "the input sample and mean need to be column vectors\n";

exit(3);

}

if (nrC!=nrmu || nrC!=nrx) {

cout << "Mahalanobis: "

<< "the dimension of the input parameters are not consistent\n";

exit(3);

}

/\*

invC = inverse(C);

diff = x - mu;

diffT = transpose(diff);

tmp = diffT->\*invC;

mdist = tmp->\*diff;

\*/

return sqrt(mtod(transpose(x-mu)->\*inverse(C)->\*(x-mu)));

}

// minkowski distance

/\*\*

\* Calculate the Minkowski distance between two vectors.

\* @param x Vector x (row vector or col vector).

\* @param y Vector y (row vector or col vector).

\* @param p Order of Minkowski distance

\* @return Minkowski distance between x and y of order p.

\*/

double minkowski(const Matrix &x, const Matrix &y, int p)

{

int nrx, nry, ncx, ncy;

int i, j;

double sum = 0.0;

nrx = x.getRow();

nry = y.getRow();

ncx = x.getCol();

ncy = y.getCol();

if (!(nrx\*nry==1 || ncx\*ncy==1)) {

cout << "The minkowski() routine can only calculate distance of vectors\n";

exit(3);

}

if (nrx!=nry || ncx!=ncy) {

cout << "Two vectors are not of the same dimension\n";

exit(3);

}

for (i=0; i<nrx; i++) // two loops is to make indifference to both row and column vectors

for (j=0; j<ncx; j++)

sum += pow(fabs(x(i,j)-y(i,j)),p);

return pow(sum,(1/(double)p));

}

**/lib/knn.cpp**

#include <iostream>

#include <cstdlib>

#include <cmath>

#include "Pr.h"

using namespace std;

/\*

tr: training set data

te: testing sample data

k: size of neighbours to be considered

nf: no. of features

c: no. of classes

p: minkowski distance order ; 2 equivalents to euclidean distance

\*/

int knn(Matrix &tr, Matrix &te, int k, int nf, int c, int p)

{

// the input sample needs to be row vectors

int nr= tr.getRow();

Matrix dist(1,nr);

Matrix sdist(1,nr);

Matrix pos(1,nr);

Matrix voting(1,c);

// calculate distance between sample data and each trainging sample

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

{

dist(0,i)=minkowski(subMatrix(tr,i,0,i,nf-1), te, p);

}

// sort the dist in ascending order

insertsort(dist,sdist,pos);

//initialize voting

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

{

voting(0,i)=0;

}

// do voting

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

{

voting(0,tr(pos(0,i),nf))++;

}

Matrix sVoting(1,c), vPos(1,c);

// sort the voting in ascending order

insertsort(voting,sVoting,vPos);

// return the label of largest voting

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

}

// kNN using partial distance

int knnPartDist(Matrix &tr, Matrix &te, int k, int nf, int c, int p)

{

// the input sample needs to be row vectors

int nr= tr.getRow();

double pDist;

Matrix dist(1,k);

Matrix sdist(1,k);

Matrix pos(1,k);

Matrix voting(1,c);

// calculate distance^p between sample data and 1st k training samples

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

{

dist(0,i)=pow(minkowski(subMatrix(tr,i,0,i,nf-1), te, p),p);

pos(0,i)=i;

}

// put max distance to the end of array

largesttoback(dist, pos);

// now for all other samples from k+1th to end of training samples, calculate partial distance

// and check with max of above k distances and if greater no need to progress calculation for that point and move to another

bool invalid=0;

for(int i=k;i<nr;i++)

{

pDist=0;invalid=0;

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

{

pDist+=pow(fabs(tr(i,j)-te(0,j)), p);

if(pDist>dist(0,k-1))

{

invalid=1;

break;

}

}

if(invalid==0) // if pDist as total less than max distance in dist array, then replace it

{

dist(0,k-1)=pDist;

pos(0,k-1)=i;

largesttoback(dist, pos); // again calculate max and put it to the end

}

}

//initialize voting

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

{

voting(0,i)=0;

}

// do voting

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

{

voting(0,tr(pos(0,i),nf))++;

}

Matrix sVoting(1,c), vPos(1,c);

// sort the voting in ascending order

insertsort(voting,sVoting,vPos);

// return the label of largest voting

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

}

// brings largest value to the back of the array and also does same to the corresponding original position

void largesttoback(Matrix &inmtx, Matrix &pos)

{

int nr, nc,temp2;

double temp1;

nr = inmtx.getRow();

nc = inmtx.getCol();

if (nr>1) {

cout << "INSERTSORT: The matrix to be sorted needs to be a row vector\n";

exit(3);

}

for(int i=0;i<nc-1;i++)

{

if(inmtx(0,i)>inmtx(0,i+1))

{

temp1=inmtx(0,i+1);

inmtx(0,i+1)=inmtx(0,i);

inmtx(0,i)=temp1;

temp2=pos(0,i+1);

pos(0,i+1)=pos(0,i);

pos(0,i)=temp2;

}

}

}

**/example/testkNN.cpp**

#include <iostream>

#include <fstream>

#include <cmath>

#include <cstdlib>

#include <ctime>

#include "Matrix.h"

#include "Pr.h"

using namespace std;

#define Usage "Usage: ./testkNN training\_set test\_set classes features use\_partial\_dist minkowski\_order dimension\_reduction\_method k \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 k: no. of neighbours to be considered \n\t use\_partial\_dist: to use partial distance '1' else '0' \n\t minkowski\_order: order for minkowski distance calculation \n\t \n\t dimension\_reduction\_method: 0,1,2 for do not use, fld, pca \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(0,0), Te(0,0);

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

if (argc ^ 9) {

cout << Usage;

exit(1);

}

// start time

clock\_t sTime=clock();

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

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

int use\_part\_dist=atoi(argv[5]); // use partial distance or not

int p= atoi(argv[6]); // order of minkowski distance where 2 equivalents to euclidean distance

int dr=atoi(argv[7]); // dimension reduction method to be used

int k=atoi(argv[8]); // k of nearest neighbours

// 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

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

// FLD method

if(dr==1)

{

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

Tr=fTr;Te=fTe;nf=fnf;

}

else if(dr==2)

{

float err=0.1; // maximum error allowed

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

int pnf=pca(Tr, Te, classes, 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);

Tr=pTr;Te=pTe;nf=pnf;

}

// prepare the labels and error count

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

int errCountkNN = 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 = subMatrix(Te, i, 0, i, nf-1);

// call kNN to perform classification

if(use\_part\_dist==1)

{

labelkNN(i,0) = knnPartDist(Tr, sample, k, nf, classes, p);

}

else

{

labelkNN(i,0) = knn(Tr, sample, k, nf, classes, p);

}

// check if the classification result is correct or not

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

{

errCountkNN++;

}

}

// end time

clock\_t eTime=clock();

// calculate accuracy

cout << "The error rate using kNN =" << (float)errCountkNN/nrTe << endl;

cout<<"The execution time = "<<(eTime-sTime)/(double)CLOCKS\_PER\_SEC<<endl;

return 0;

}