Study of feature ranking using Bhattacharyya distance

Daniel Rugeles
drugeles@purdue.edu

May 2012

Abstract

In this document, we study how a ranking of features could help improving an optimized KNN classifier. For the the K-Neirest Neighbor classifier (KNN) we optimized the K value, the L_p norm and the weighting function for a particular dataset.

1 Introduction

An anonymous company is trying to find the best classifier for some data that they have collected. In the course ECE662 Pattern Recognition taught by Professor Mireille Boutin. Students have agreed to have a contest using the mentioned data. The purpose of this contest is to practice the theory studied in the course and at the same time help the company finding the most accurate classifier using this data. For information about the data, see section 2.

In the present work, we study how to get the most accurate classification using KNN Classifier. For this, we have decided to optimize KNN classifier using different selections of the features provided. The feature selector is presented in section 4. The considerations for optimizing KNN are presented in 5 and the experiments done are presented in 6.

Motivation for this work, comes from the idea of testing the competitiveness of KNN against other methods. For this, we optimize several parameters of the KNN Classifier and we compare the result against classifiers from other students. Results from the whole class can be found at:
https://www.projectrhea.org/rhea/index.php/Hw3_ECE662_S12.

2 Data used in this work

The data provided comes from a five-class classification problem using 13 features. The training data consists of 550 data points (i.e. 550 points in a 13 dimensional space) along with the correct label for each point. The number of labels correspond to the number of classes, in this case five classes.

3 KNN Classifier

KNN classifier is a geometric based classifier in the area of Statistical Pattern Recognition. Like other Statistical Classifiers, the goal is to find a decision hypersurface that will define the class of an unknown element. Unlike other statistical classifiers, geometric based classifiers decision boundaries are constructed directly from the data or the features without using the information of the class conditional densities. By instance we mean one point in a particular space given in the training set.

KNN classifier operate on the premise that classification of unknown instances can be done by relating the unknown instance to the known instances according to some distance/similarity function, see 5.2. Intuitively, instances in the space who are closer according to a distance metric are more likely to be in the same class. So that, when an unknown instance needs to be classified. There is only need to check for the classes of the closest point to the unknown instance. In other words, the K neirest neighbors (K $\geq$ 1).

Differently from other classifiers such Neural networks, KNN classifier does not abstract any information from the training data during the learning phase. Learning is merely a question of querying the training data at the time of classification. There exists three parameters that we can change according to our selection of the KNN classifier. The metric, the value of K, and the neighbor weighting.

It is expected that more robust models for a KNN classifier can be achieved by optimizing the number K of neighbours and letting the majority vote decide the outcome of the class labelling. A higher value of K should result in a less locally sensitive outcome. The metric should be choosen so that the way the data the way the classes are distributed in space will match the shape of the equidistance curves for our defined metric. An intuitive picture about this equidistances is shown in figure 1. The weighting function should give a measure of how important is the vote. Closest neighbors should have more importance when voting.

4 Feature Ranking and aggregation

Features can be ranked according to how much information they provide about the labeled classes. In this document, we use the Bhattacharyya distance ap-

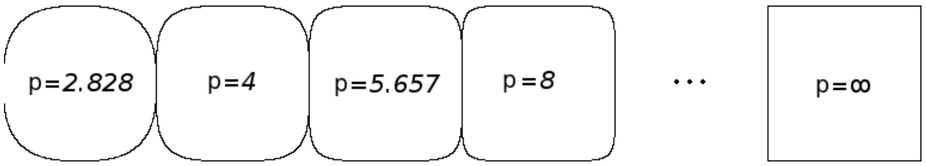


Figure 1: Unit distance for the mankowski metric

proach. The Bhattacharyya distance is a common method for measuring the separation between two multivariate gaussians. Therefore, we will have to use this method based on the assumption that data is drawn from a Gaussian distribution. Because we have five classes in our data, we first estimate a Gaussian distribution from which the data is drawn, then we calculate all the possible combinations among the classes. Finally we add all this distances to produce a ranking for each feature. The Bhattacharyya distance for two multivariate distributions P1 and P2 can be calculated as follows:

$$BhatDistance(P1,P2) = \frac{1}{8}\left(m1-m2\right)^T P^{-1}\left(m1-m2\right)+\frac{1}{2}ln\left(\frac{detP}{\sqrt{detP1detP2}}\right)$$

$$P = \frac{P1+P2}{2}$$

Once we found a way to order the importance of each feature. We proceed to do an aggregate composition of the features as follows. First, we test using only information of the highest ranked feature, then we test using the information of the highest and the second highest feature, we repeat this procedure until we use the information from all the given features. We will call this process Aggregation of features.

The ranking found for this data will be posted in the result section 7. Intuition indicates we should have an improvement as we use more features in the classifier, we will discuss about this in the conclusions section. 10

5 Optimizing the KNN Classifier

First, let's define the parameters that we can optimize in a KNN classifier.

- K number of neighbors.
- Distance Metric used.
- Neighbor Weighting.

5.1 K number of neighbors

K could be any integer value greater than 0. We have chosen to test the range from 1-19 neighbors. For each number K we fix other KNN parameters and we record the percentage of successes. More explanation about the trials made are explained in section 6.

5.2 Metrics

A metric is a function that defines a distance between elements of a set. For optimizing the metric, we have fixed the KNN parameters for each metric.

A metric on a set $\mathbb{S}$ is defined as a function

$$d : \mathbb{S} \text{ x } \mathbb{S} \rightarrow \mathbb{R}$$

Such that $\forall x, y \in \mathbb{S}$:

1. $d(x, y) \geq 0$
2. $d(x, y) = 0$ if and only if $x = y$
3. $d(x, y) = d(y, x)$
4. $d(x, z) \leq d(x, y) + d(y, z)$. Also known as the triangle inequality.

$\mathbb{R}$ is the set of real numbers.

In the literature, examples of metrics are the euclidean distance, the manhattan distance (Both are special cases of the Minkowski metric) and the Canberra distance. Implementations of this distances can be found in [6].

We will study how to optimize the family of Lp norm metrics, or Minkowski metrics. We will not prove this metrics, but we will use them to study their effect in finding our hypothesis. In specific we will study:

- $L_{\frac{1}{2}}$, $L_{\frac{1}{4}}$
- L_1 Norm, also known as manhattan distance.
- L_2 Norm, also known as euclidean distance.
- L_4 Norm, L_8 Norm.
- $\lim_{p\rightarrow\infty} L_p$ Norm, also known as maximum distance and supremum norm.

where

$$L_p = \left(\sum_{i=1}^{n} |x_i + y_i|^p \right)^p$$

To get an intuitive idea, figure 1 shows the unit distance in $\mathbb{R}$x$\mathbb{R}$ for different values of p

To design your own distance function you can use the proxy package available in R packages via CRAN see [4]. Proxy not only has a large number of pre-specified metrics, but it provides a framework for specifying your own distance function that is called from compiled code and thus it is reasonably fast.

5.3 Neighbor's Weight

Neighbor's weight represent how much importance do we give to the vote of the neighbors, when using a KNN Classifier, one approach will be using the same weight for all the neighbors, as they will just provide a vote given their labeled class. This kind of weighting is called uniform weighting.

$$w_i = 1$$

Where w_i is the weight assign to the ith neiresth neighbor and d_i is the distance from the test sample to the ith nearest neighbor using the specified metric.

Another common approach is using the inverse distance weighting. It is very intuitive as less weight is assign to instances with farther distances [3]. In other words, a farther instance implies less knowledge about the unknown instance.

$$w_i = \{ \tfrac{1}{d_i} \text{ if } d_i \neq 0$$

Another approach could be an adaptive gaussian function. It is called adaptive because its parameters will depend on the distance of the K neighbors. In this document we set the mean of the Gaussian to be the position of the point that we are querying and the variance to be the distance to the K/2 neighbor so that if the K neighbor is to far away its value will be exponentially lower. Next, we show the function used to compute the weight for the vote of the X neighbor.

$$f_{(\mathbf{X})} = \frac{1}{(2\pi)^{n/2}\det(\Sigma)^{\frac{1}{2}}} \exp - \left(\tfrac{1}{2} (\mathbf{x} - \mu)^T (\mathbf{\Sigma})^{-1} (\mathbf{x} - \mu) \right)$$

where $f_{(\mathbf{X})}$ represents the weight of $\mathbf{X}$, $\mathbf{X}$ is a vector in $\mathbb{R}^{13}$ representing the known neighbor instance μ is the coordinate of the unknown distance and Σ is the set of variances in matrix form that will define how quickly the weights will vanish.Σ depends directly on the distance of X to the point that we are classifying.

In this work we have tested the three weighting functions mentioned above.

- Uniform Weighting
- Inverse distance weighting
- Adaptive gaussian weighting

For each one of this, we have fixed the other parameters of the KNN classifier. In the next section all the experiments trials are explained.

6 Experiment

We will perform 13x19x7x3x3 trials of experiments for the KNN classifier.
The five testing variables correspond to:

- 13 different aggregation of the features.
- $0 \leq K \leq 20$ neighbors.
- 7 distance metrics. $L_{\frac{1}{2}}$, $L_{\frac{1}{4}}$, L_1,L_2, L_4 L_8, and $\lim_{p\to\infty} L_p$ Norm.
- 3 weighting functions.
- 3 different cross validation test.

The amount of trials is a large number, dynamic programming is used to reduce the computational load. The code is presented in A.

In every cross validation trial about 33% of the training data was used for testing and 66% was used for training. The results were averaged over the cross validation tests and they are presented in the next section.

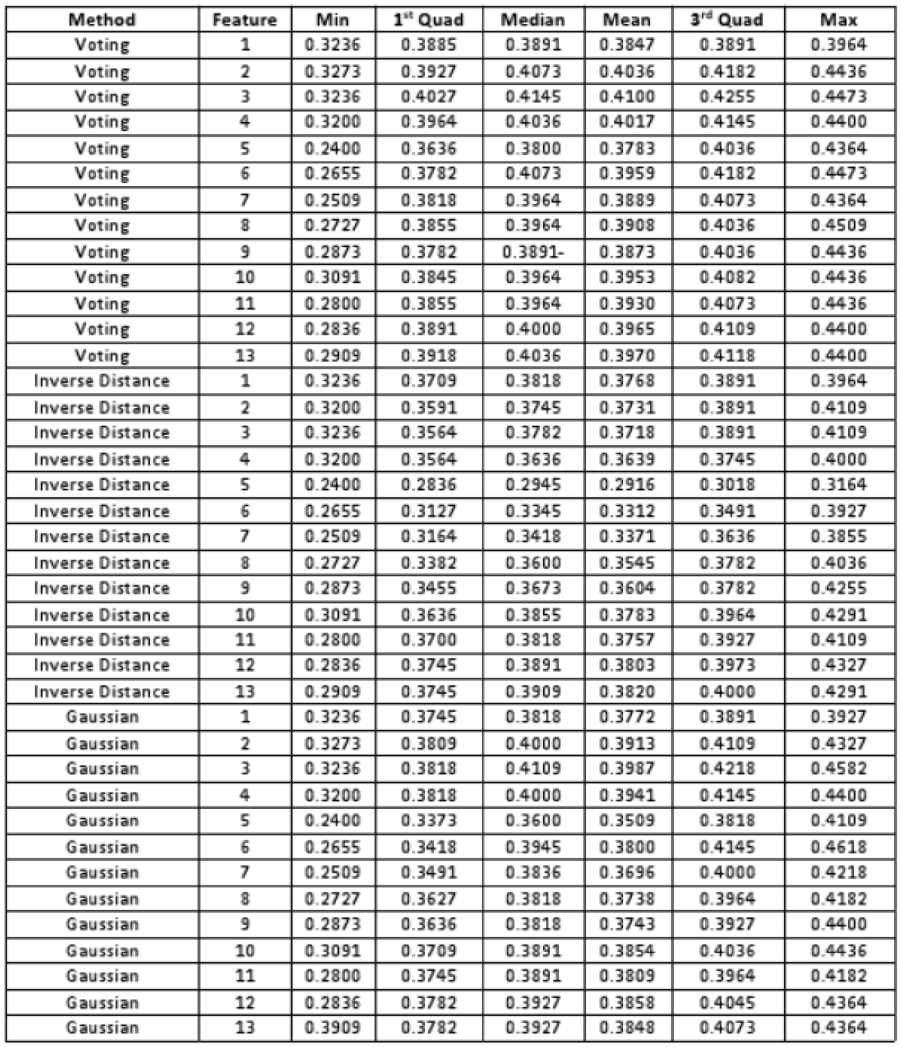
7 Training Results

First, let us show the results from ranking the features.

feat 1	feat 2	feat 3	feat 4	feat 5	feat 6	feat 7	feat 8	feat 9	feat 10	feat 11	feat 12	feat 13
0.29	0.47	0.38	0.14	0.19	0.34	0.37	0.56	0.87	0.73	0.11	0.19	0.18

Table 1: Results averaged over cross validation tests.

This results imply that feature 9, feature 8 and feature 10 have better probability of discriminating among classes.



Method	Feature	Min	1st Quad	Median	Mean	3rd Quad	Max
Voting	1	0.3236	0.3885	0.3891	0.3847	0.3891	0.3964
Voting	2	0.3273	0.3927	0.4073	0.4036	0.4182	0.4436
Voting	3	0.3236	0.4027	0.4145	0.4100	0.4255	0.4473
Voting	4	0.3200	0.3964	0.4036	0.4017	0.4145	0.4400
Voting	5	0.2400	0.3636	0.3800	0.3783	0.4036	0.4364
Voting	6	0.2655	0.3782	0.4073	0.3959	0.4182	0.4473
Voting	7	0.2509	0.3818	0.3964	0.3889	0.4073	0.4364
Voting	8	0.2727	0.3855	0.3964	0.3908	0.4036	0.4509
Voting	9	0.2873	0.3782	0.3891-	0.3873	0.4036	0.4436
Voting	10	0.3091	0.3845	0.3964	0.3953	0.4082	0.4436
Voting	11	0.2800	0.3855	0.3964	0.3930	0.4073	0.4436
Voting	12	0.2836	0.3891	0.4000	0.3965	0.4109	0.4400
Voting	13	0.2909	0.3918	0.4036	0.3970	0.4118	0.4400
Inverse Distance	1	0.3236	0.3709	0.3818	0.3768	0.3891	0.3964
Inverse Distance	2	0.3200	0.3591	0.3745	0.3731	0.3891	0.4109
Inverse Distance	3	0.3236	0.3564	0.3782	0.3718	0.3891	0.4109
Inverse Distance	4	0.3200	0.3564	0.3636	0.3639	0.3745	0.4000
Inverse Distance	5	0.2400	0.2836	0.2945	0.2916	0.3018	0.3164
Inverse Distance	6	0.2655	0.3127	0.3345	0.3312	0.3491	0.3927
Inverse Distance	7	0.2509	0.3164	0.3418	0.3371	0.3636	0.3855
Inverse Distance	8	0.2727	0.3382	0.3600	0.3545	0.3782	0.4036
Inverse Distance	9	0.2873	0.3455	0.3673	0.3604	0.3782	0.4255
Inverse Distance	10	0.3091	0.3636	0.3855	0.3783	0.3964	0.4291
Inverse Distance	11	0.2800	0.3700	0.3818	0.3757	0.3927	0.4109
Inverse Distance	12	0.2836	0.3745	0.3891	0.3803	0.3973	0.4327
Inverse Distance	13	0.2909	0.3745	0.3909	0.3820	0.4000	0.4291
Gaussian	1	0.3236	0.3745	0.3818	0.3772	0.3891	0.3927
Gaussian	2	0.3273	0.3809	0.4000	0.3913	0.4109	0.4327
Gaussian	3	0.3236	0.3818	0.4109	0.3987	0.4218	0.4582
Gaussian	4	0.3200	0.3818	0.4000	0.3941	0.4145	0.4400
Gaussian	5	0.2400	0.3373	0.3600	0.3509	0.3818	0.4109
Gaussian	6	0.2655	0.3418	0.3945	0.3800	0.4145	0.4618
Gaussian	7	0.2509	0.3491	0.3836	0.3696	0.4000	0.4218
Gaussian	8	0.2727	0.3627	0.3818	0.3738	0.3964	0.4182
Gaussian	9	0.2873	0.3636	0.3818	0.3743	0.3927	0.4400
Gaussian	10	0.3091	0.3709	0.3891	0.3854	0.4036	0.4436
Gaussian	11	0.2800	0.3745	0.3891	0.3809	0.3964	0.4182
Gaussian	12	0.2836	0.3782	0.3927	0.3858	0.4045	0.4364
Gaussian	13	0.3909	0.3782	0.3927	0.3848	0.4073	0.4364

Figure 2: Statistical significance of the results for the experiments explained in section.The percentages represent how many samples were correctly classified. Feature value represent the number of aggregated features.

In the following table, we will present the results of aggregating features. The table contains the statistical representation of the results for every experiment explained in the previous subsection. We clarify that feature ith does not mean that we classify using the information of the ith feature, it means that we used the ith ranked feature and higher ranked features as well. We also clarify that each row is drawn from 19x7x3 trials of KNN. The study of Lp norm, and K value in Optimal KNN is covered in previous studies in the course. knnoptimal

The following figures provide a visual representation of the data given in the table.

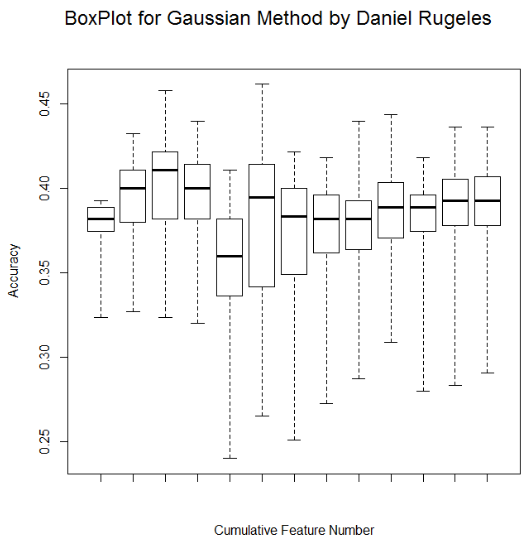


Figure 3: Results for the Gaussian weighting function

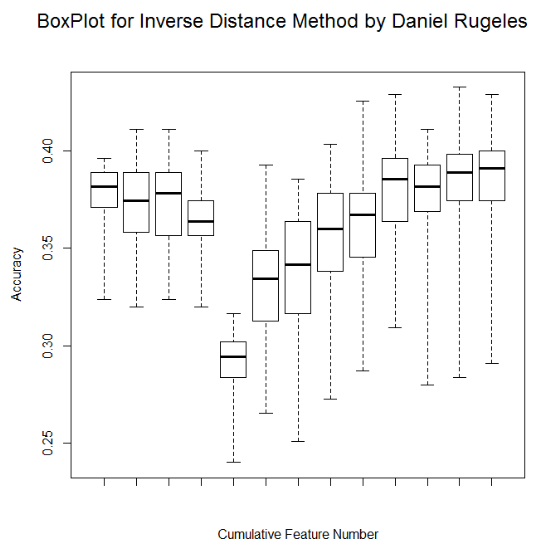


Figure 4: Results for the inverse distance

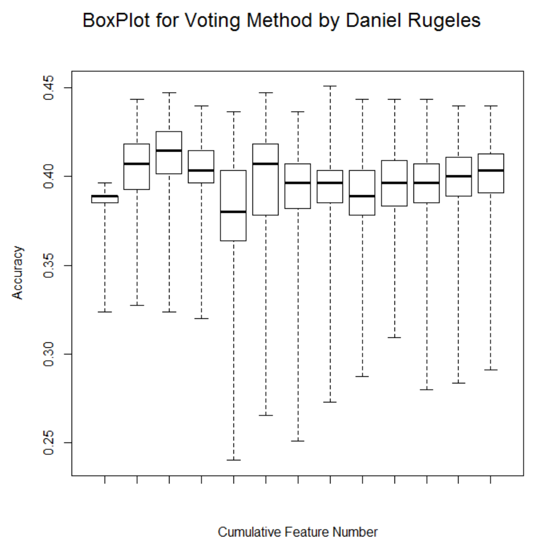


Figure 5: Results for the voting function

The following figure compares the maximum values from the previous two figures, the idea is to study the stability and the relative process of aggregating features using the three weighting features. See the next section for reviewing our conclusions.

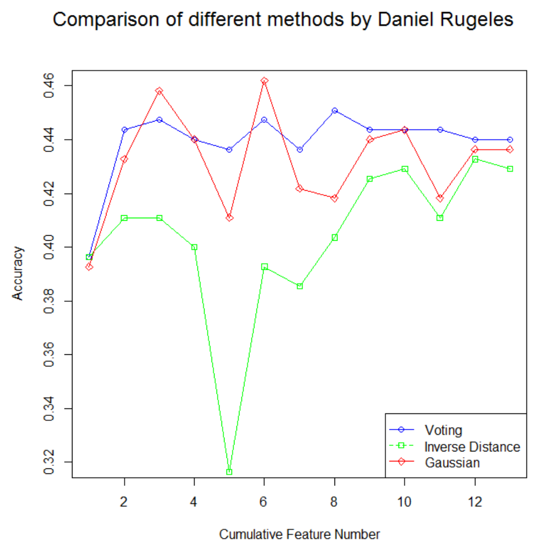


Figure 6: Comparisons of the results for the three weighting functions. This image also shows an overall behavior of feature aggregation

8 Final Decision for testing

For the given data we choose K=14. From results not presented in this work. It is not hard to see that $K \leq 7$ produce the lowest accuracies, in average the best values are around K=14 and they decrease unevenly after K=15.

For the given data we choose the Lp norm as a distance metric with p being equal to 1/2. It can be seen that, in general, when p=1/2 or close values there is always above 75th percentile ranked accuracy independently of the weighting function and the amount of features that are being used.

All the other decisions follow from the results obtained in the previous section.

- K = 14
- Weighting function = Adaptive Gaussian
- Feature Aggregation = First three ranked features
- Distance Metric = Lp norm with p=1/2

9 Testing results

Estimated accuracy: 43.54%
Test set accuracy: 41.58%

CONF	class 0	class 1	class 2	class 3	class 4
class 0	35	12	14	5	0
class 1	6	2	7	2	0
class 2	7	5	5	1	0
class 3	0	0	0	0	0
class 4	0	0	0	0	0

Table 2: Confusion Matrix

10 Conclusions

Most of the results from this work are biased towards experimenting with weighting functions and Feature Ranking and aggregation. In Homework 2, Optimizing KNN. There is a detailed exploration on optimizing the K value and the distance metric.

- KNN is definitively not the best way to classify this data. When compared against other methods the performance is 6% less than just selecting always the most likely prior. When compared against SVM, there is a 5% negative difference. This method is below the winner of the competition by 9%. That method used the less correlated two features. Interestingly those are features 4 and 9 which are almost the highest ranked and the lowest ranked features using Bhattacharyya distance. When comparing our KNN against other KNN classifier, we find a very challenging situation against the winner of this competition. That person tried KNN with LDA and K=25, our result shows higher predicted accuracy than our optimized KNN and practically the same test accuracy. While we obtained 41.58%, LDA+KNN obtained 41.81%. When comparing our KNN classifier with other KNN classifiers, we obtain a much better answer. Other KNN methods who participated in the contest obtained accuracies of 26.73% and 36.63%. Wrapping up, feature selection as well as LDA seem to be very good alternatives when classifying very correlated data with very unbalance classes.

- The performance of the inverse distance weighting function is lower than expected. In fact it is always lower than the other two weighting functions. Voting function dominates overall, but for greedy analysts, Gaussian weighting reaches by best results.

- Fifth order statistic seems to be ranked incorrectly, every weighting function shows some sign of decay in this feature. Our explanation is that fifth feature is not likely being drawn from a Gaussian distribution. We strongly recommend eliminating this feature and keep aggregating features results might improve for the first following aggregated features.

- The stability of the weighting functions can also be appreciated from Figure 6. Gaussian weighting functions will be good for those who like to take risks, while voting weighting functions will be a good option for those who rather have a. No matter which function is chosen, inverse distance definitively does not perform well on this data.

- Aggregate composition of features is an important value to optimize. When all the information is used, we experience the curse of dimensionality. Similarly when we use few information the accuracy is not so good. This can be appreciated in Figures 3,4, and 5 where results are not so good when aggregating feature number five.

11 Future work

Future work include studying how to optimize parameters for the Adaptive gaussian weighting function as it has shown to have better results than other weighting functions. It is also recommended to keep experimenting other ways of aggregating the features. For example a greedy algorithm might provide interesting results. We can choose the feature that has a better accuracy on the training data, and then adding to the analysis all of the other features one by one to see which one provides better results. This procedure should be followed until finding a decrease in the accuracy. This algorithm is very costly but it might be interesting to compare against fisher's linear discriminant.

References

[1] Daniel Adler and Duncan Murdoch. *rgl: 3D visualization device system (OpenGL)*, 2012. R package version 0.92.861.

[2] Kaggle Inc. Data mining competitions, 2010.

[3] Ron Kohavi, Pat Langley, and Yeogirl Yun. The utility of feature weighting in nearest-neighbor algorithms. In *Proceedings of the Ninth European Conference on Machine Learning*, pages 85–92. Springer-Verlag, 1997.

[4] David Meyer and Christian Buchta. *proxy: Distance and similarity measures*, 2012. R package version 0.4-7.

[5] Daniel Rugeles. Ece 662 pattern recognition. Notes taken in class, 2012.

[6] R Development Core Team. *R: A Language and Environment for Statistical Computing.* R Foundation for Statistical Computing, Vienna, Austria, 2011. ISBN 3-900051-07-0.

A Code

A.1 Feature Ranking

```
###############################################################


############### BHATTACHARYYA DISTANCE ################
# Finds the Bhattacharyya distance between data drawn #
# from Normal Distributions                           #
#                                                     #
# Warning. Distributions with Variance=0              #
# will be returned as 0 distance                      #
#                                                     #
# Equivalent to                                       #
# bhattacharyya.dist(c(1,1),c(2,5),diag(2),diag(2))   #
#                                                     #
# X1: First univariate gaussian data                  #
# X2: Second univariate gaussian data                 #
#######################################################

#install.packages("fpc")
library(fpc)

bhatDistance <- function(X1,X2){

    # define means
    mX1 <- mean(X1)
    mX2 <- mean(X2)

# define difference of means
    mDiff <- mX1 - mX2

# define covariance matrix per each class
    cvX1 <- cov(X1,X1)
    cvX2 <- cov(X2,X2)

# define halfsum of cv's
    p <- (cvX1+cvX2)/2

# Bhattacharyya's equation
    a <- 0.125 * t(mDiff) * p^(-1) * mDiff + 0.5 * log( abs(p) /
                    sqrt( abs(cvX1) * abs(cvX2) ))
    return (a)
}


#Example:
#X1<-c(1,4)
#X2<-c(2,5)
#bhatDistance(X1,X2)


###############################################################


############ FEAT RANKING #############
```

```
# Finds a vector with the percentage  #
# of success for following parameters #
#                                      #
# dist: Distance among all the points #
# train: training data                 #
# K: maximum number of neighbors       #
########################################


featRanking <- function(train){

    #print(train)

    #Hold temporary the data for one class in one feature
    classtemp<-c()

    #Hold the separation for each class per feature
    classesseparation<-c()


    classfeat<-c()

    ranking<-c()

    #for each feature
    for(i in 2:length(train)){

        #list of list with data from each class
        class<-c()

        #Allocate memory. 6 is the number of classes + 1
        class[[6]][[2]] < -0

        #assign each class to a vector
        for(j in 1:length(train[[1]])){

            #use class as key
            cla<-train[[1]][[j]]
            #data to be inserted
            dat<-train[[i]][[j]]

            class<-appendToHash(class,cla,dat)

        }

        #print(class)

        ranking[i-1]<-0

        for( a in 1:(length(class)-2)){
            for( b in (a+1):length(class)-1){
                #print("distance between")
                #print(a)
                #print(b)
                #print(class[[a]])
                #print(class[[b]])
                dis<-bhatDistance(class[[a]],class[[b]])

                if(is.na(dis)){dis<-0}

                ranking[i-1]<-ranking[i-1]+dis

                #print("distance")
                #print(dis)

            }
        }
    }

    return(ranking)

}

###########################################################


#append value into "hash" table using key

appendToHash<-function(hash,key,value){


    len<-length(hash[[key+1]])

    #print("insert in vector:")
    #print(index)
    #print(value)

    hash[[key+1]][[len+1]]<-value #class could be zero

    return(hash)
}

###########################################################


#Find the ranking
train=read.csv("traintestknn.csv")
result<-featRanking(train)
```

A.2 KNN Classifier

```
################################ PLOTING

############# PLOT 3D ###############
# Plots Data in 3D                          #
#                                           #
# z: Grid (Matrix) with values              #
#                                           #
#####################################

#install.packages("rgl")
library(rgl)

plot3d <-function (z){
    z <- 10*z #Data to plot
    x <- 10*(1:nrow(z)) # 10 meter spacing (S to N)
    y <- 10*(1:ncol(z)) # 10 meter spacing (E to W)
    zlim <- range(y)
    zlen <- zlim[2] - zlim[1] + 1
    colorlut <- heat.colors(zlen,alpha=0) # height color lookup table
    col <- colorlut[ z-zlim[1]+1 ] # assign colors to heights for each↩
         point
    open3d()
    rgl.bbox(alpha=0.8, front="lines",back="lines",color="black",lit=↩
        FALSE)
    #Information Surface
    rgl.surface(x, y, z, color=col, alpha=1,lit=FALSE)
    #Information lines
    rgl.surface(x, y, z, color="black", front="lines",alpha=1,lit=↩
        FALSE)
    #Contour Map
    colorlut <- terrain.colors(zlen,alpha=1)
    col <- colorlut[ z-zlim[1]+1 ]
    rgl.surface(x, y, matrix(1, nrow(z), ncol(z)), color="black",front↩
        ="lines",back="lines",col.axis="black",lit=FALSE)
}


################################### KNN

######### INSERT K NEIGHBORS ##########
# Inserts and keeps in order the      #
# closest  K neighbors                #
#                                     #
# val: point to be inserted           #
# neigh: structure holding KNN        #
# pos: index of val                   #
#                                     #
#######################################

insertKneighbor<-function(val,neigh,pos){
    len<-length(neigh)/2
    #position holds the position of neigh
    for(i in 1:len){
        if(val<neigh[i]){
            if((len-1)>=i){
                for(j in (len-1):i){
                    neigh[(j+1)]<-neigh[j]
                    neigh[(j+1+len)]<-neigh[(j+len)]
                }
            }
            neigh[i]<-val
            neigh[(i+len)]<-pos
            return (neigh)
        }
    }
    return (neigh)
}

cummulative<-function(num){
    result<-0
    if(1<=num){
        for(i in 1:num){
            result<-result+i
        }
    }
    return(result)
}
####################################################################

########## FIND K NEIGHBORS ###########
# Finds the closest neighbors and     #
# their distances with the following  #
# parameters.                         #
#                                     #
# K: number of neighbors              #
# num: index to point of interest     #
# length: length of training data     #
# dist: Distance among all the points #
#                                     #
#######################################

findKneighbors<-function(K,num,length,dist,init,end){

    #There is K-1 neighbors maximum
    if(K>=length){
        print("K is too big")
        return()
    }

    neigh<-c()
    for (i in 1:K){
        neigh[i]= Inf
    }
    neigh<-cbind(neigh,neigh)

    #a<-c()
    if(init<=num){
        index<-(num-1)+(init-1)*(length-1)-cummulative((init-1))
    }
    else{
        index<-((num-1)+(num-1)*(length-1))-cummulative((num-1))+init-↩
            num+1
    }

    if(end<num){til<-end}
    else{til<-num-1}


    if(init<=til && init<num){

        for(i in init:til){
            index<-(num-1)+(i-1)*(length-1)-cummulative((i-1))
            #a[(i-init+1)]<-dist[index]
            neigh<-insertKneighbor(dist[index],neigh,i)
        }
    }
    index<-(num-1)+(til)*(length-1)-cummulative(til)

    #Uncomment if you want to add the same point to be his own ↩
        neighbor.
    #if(end>=num && init<=num){
    #   neigh<-insertKneighbor(0,neigh,(num))
    #   #a[til+1-init+1]<-0
    #}

    if(init>num){
        since<-init-1
        index<-index+init-num-1
    }
    else{
        since<-til+1
    }

    til<-end-1

    if(since<=til){

        for(j in (since:til)){
            index<-index+1
            #a[(j-init+2)]<-dist[index]
            neigh<-insertKneighbor(dist[index],neigh,(j+1))

        }
    }
    #print(neigh)
    #print(a)
    return(neigh)
}


####################################################################


############ MULTIPLE KNN #############
```

```
# Finds a vector with the percentage  #
# of success for following parameters #
#                                     #
# dist: Distance among all the points #
# train: training data                #
# K: maximum number of neighbors      #
#######################################


MultipleKNN<-function(K,train,dist,joinfun){

    len<-length(train[[1]])

    #Initialize result
    results<-c()
    result<-c()
    for(i in 1:K){
        result[i]<-0
    }

    #for each point in the data set
    for(point in 1:(as.integer(len/2))){#(len)<->(len/2)

        # Second half of n contains the index of the neighbor
        # The first half contains the distance to such index
        n<-findKneighbors(K,point,len,dist,as.integer(len/2)+1,len)#1 ↩
            <-> as.integer(len/2)+1

        # Classify the first K neighbors (n) of (point)
        if(joinfun==1){
            classification<-ClassifyVoting(n,train)
        }
        if(joinfun==2){
            classification<-ClassifyInvDist(n,train)
        }
        if(joinfun==3){
            classification<-ClassifyGaussian(n,train)
        }

        if(DEBUG){print("classification")
            print(classification)}

        #Find the answer which needs to be added one, because classes ↩
            start with 1
        answer<-train[point,1]
        answer<-answer+1

        if(DEBUG){
            print("answer")
            print(answer)}

        #Evaluate the classifier
        for(i in 1:length(classification)){
            if(answer==classification[i]){
                result[i]<-1
            }else{
                result[i]<-0
            }
        }
        if(DEBUG){
            print("result")
            print(result)}

        #Matrix with results: rows=test trial  columns=Col-NN
        results<-rbind(results,result)
    }

    if(DEBUG){
        print("results")
        print(results)}

    total<-apply(results, 2, sum)
    total<-total/length(results[,1])

    return (total)
}

##############################################################

ClassifyVoting<-function(n,train){
    value<-0
    result<-c()

    numclasses<-max(train[1])

    #Initialize the array that holds the likelyhood per class of each ↩
        point
    a<-c()
    for(i in 1:(numclasses+1)){
        a[i]<-0
    }

    #Initialize the array that holds the classification per each K
    classification<-c()
    for(k in 1:as.integer(length(n)/2)){
        classification[k]<-0
    }

    #n/2 is the last neighbor
    for (neigh in 1:as.integer(length(n)/2)){

        if (n[neigh,1]>=Inf){
            result[neigh]<-0
        }
        else{
            neighidx<-n[neigh,2]
            if(!is.na(train[neighidx,1])){
                neighborclass<-train[neighidx,1]
                #print("neighborclass")
                #print(neighborclass)
                a[neighborclass+1]<-a[neighborclass+1]+1
                #print("a")
                #print(a)

                #Find the most likely class per neighbor
                #print("maxindex(a)")
                #print(maxindex(a))
                classification[neigh]<-MaxIndex(a)
                #print("class")
                #print(classification)

            }
        }

    }

        if(DEBUG){
        print("a")
        print(a)}

    return(classification)
}

##############################################################

ClassifyInvDist<-function(n,train){
    value<-0
    result<-c()

    numclasses<-max(train[1])

    #Initialize the array that holds the likelyhood per class of each ↩
        point
    a<-c()
    for(i in 1:(numclasses+1)){
        a[i]<-0
    }

    #Initialize the array that holds the classification per each K
    classification<-c()
    for(k in 1:as.integer(length(n)/2)){
        classification[k]<-0
    }


    #n/2 is the last neighbor
    for (neigh in 1:as.integer(length(n)/2)){

        if (n[neigh,1]>=Inf){
            result[neigh]<-0
        }
        else{
            neighidx<-n[neigh,2]
            if(!is.na(train[neighidx,1])){
                neighborclass<-train[neighidx,1]
                #print("neighborclass")
                #print(neighborclass)

                a[neighborclass+1]<-a[neighborclass+1]+1/n[neigh,1]


                #Find the most likely class per neighbor
                #print("maxindex(a)")
                #print(maxindex(a))
                classification[neigh]<-MaxIndex(a)
                #print("class")
                #print(classification)

            }
        }
    }

       if(DEBUG){
        print("a")
        print(a)}

    return(classification)
}

################################################################

ClassifyGaussian<-function(n,train){
    value<-0
    result<-c()

    numclasses<-max(train[1])

    #Initialize the array that holds the likelyhood per class of each ↩
        point
    a<-c()
    for(i in 1:(numclasses+1)){
        a[i]<-0
    }
    #Initialize the array that holds the classification per each K
    classification<-c()
    for(k in 1:as.integer(length(n)/2)){
        classification[k]<-0
    }

    #n/2 is the last neighbor
    for (neigh in 1:as.integer(length(n)/2)){

        medianneigh<-n[length(n)*1/4+1]
        if (n[neigh,1]>=Inf){
            result[neigh]<-0
        }
        else{
            neighidx<-n[neigh,2]
            if(!is.na(train[neighidx,1])){
                neighborclass<-train[neighidx,1]
                #print("neighborclass")
                #print(neighborclass)

                a[neighborclass+1]<-a[neighborclass+1]+dnorm(n[neigh↩
                    ,1],mean=0,sd=medianneigh)

                #Find the most likely class per neighbor
                #print("maxindex(a)")
                #print(maxindex(a))
                classification[neigh]<-MaxIndex(a)
                #print("class")
                #print(classification)

            }
        }

    }

    if(DEBUG){
        print("a")
        print(a)}

    return(classification)
}

##############################################################

#Find the index of the maximum value in the array
MaxIndex<-function(a){
    maximum<-1
    for(i in 1:length(a)){
        if(a[i]>a[maximum]){
            maximum<-i
        }
    }
    return(maximum)
}


##############################################################

OptimizeKNN<- function(K,train){

    for (join in 1:3){
        dist<-scan("mindist1_4.dat")
        totmin1_4<-MultipleKNN(K,train,dist,join)

        #dist<-scan("mindistr1_8.dat")
        #totminr1_8<-MultipleKNN(K,train,dist,join)

        dist<-scan("mindist1_2.dat")
        totmin1_2<-MultipleKNN(K,train,dist,join)

        dist<-scan("mindistr1_2.dat")
        totminr1_2<-MultipleKNN(K,train,dist,join)

        dist<-scan("mindist1.dat")
        totmin1<-MultipleKNN(K,train,dist,join)

        #dist<-scan("mindistr2.dat")
        #totminr2<-MultipleKNN(K,train,dist,join)

        dist<-scan("mindist2.dat")
        totmin2<-MultipleKNN(K,train,dist,join)

        #dist<-scan("mindistr8.dat")
        #totminr8<-MultipleKNN(20,train,dist)

        ##dist<-scan("mindist4.dat")
        ##totmin4<-MultipleKNN(K,train,dist,join)

        ##dist<-scan("mindist8.dat")
        ##totmin8<-MultipleKNN(K,train,dist,join)

        #dist<-scan("candist.dat")
        #totcan<-MultipleKNN(K,train,dist,join)

        #dist<-scan("mindist.dat")
        #totmin<-MultipleKNN(K,train,dist,join)

        dist<-scan("maxdist.dat")
        totmax<-MultipleKNN(K,train,dist,join)

        total<-c()
        #total<-rbind(total,totman)
        #total<-rbind(total,toteuc)
        #total<-rbind(total,totcan)
        total<-rbind(total,totmin1_4)
        #total<-rbind(total,totminr1_8)
        total<-rbind(total,totmin1_2)
        total<-rbind(total,totminr1_2)
        total<-rbind(total,totmin1)
        #total<-rbind(total,totminr2)
        total<-rbind(total,totmin2)
        #total<-rbind(total,totminr8)
        ##total<-rbind(total,totmin4)
        ##total<-rbind(total,totmin8)
        total<-rbind(total,totmax)


        print(paste("Result Join Method #",join,sep=""))

        print(total)

        resultfile<-paste("resulttest",join,sep="")
        resultfile<-paste(resultfile,".dat",sep="")

        write(total,resultfile)

        #print("after")
        #temp<-scan(resultfile)
        #print(temp)

        #plot3d(total)
    }
}
```