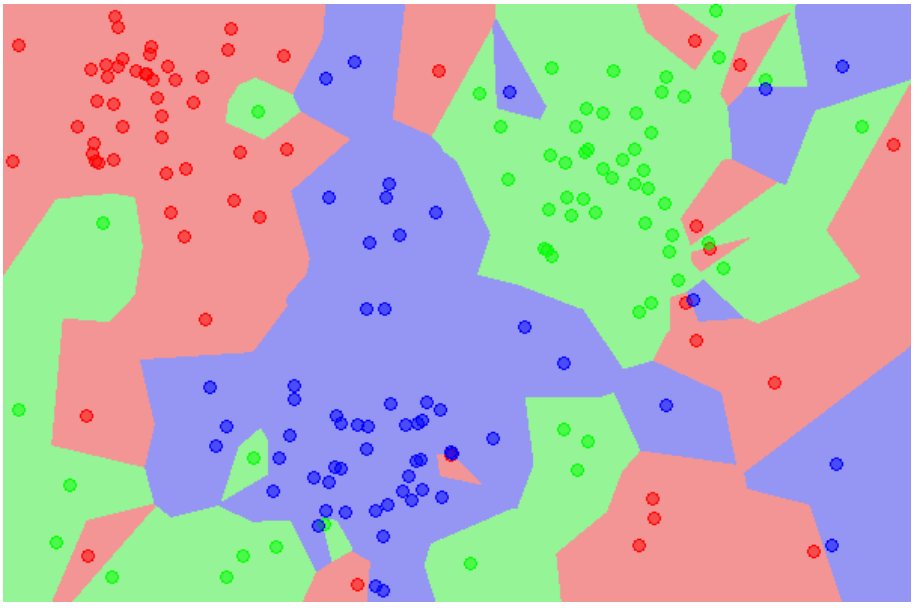
Classification Techniques

K-nn
Naive Bayesian

Methods in Bioinformatics

Kyriakis Dimitris



Contents

1 Introduction . . . . . **ii**
1.1 Stratification iii
1.2 Cross Validation iv
1.3 Limitations vii

2 K-nn . . . . . **viii**

3 Naive Bayesian . . . . . **xiii**

4 Bibliography . . . . . **xv**

1 Introduction

In machine learning, classification is the problem of identifying to which of a set of categories, a new observation belongs, on the basis of a training set of data containing observations whose category membership is known. Classification is one of the most widely used techniques in machine learning, for medical diagnosis and image classification etc. Linear classifiers are amongst the most practical classification methods. At the begging we hide a 10% of our data in order to use it as test ("unknown dataset"). Our Model was not trained for these observations. We tried not to remove any class in this procedure.

```
def Create_Test(X,labels,full_indexes):
    '''
    **Description:** Hide the 10% of the data\n
    **Input:**\n
    - X: all data\n
    - Labels: all labels (and the hidden)\n
    **Output:**\n
    - Train indexes (all indexes without test)\n
    - Test indexes\n
    '''
    n_samples = len(labels)
    labels_new=[]
    while len(set(labels)) !=  len(set(labels_new)):
        Test_indexes = rd.sample(full_indexes,round(0.1*n_samples))
        labels_new = labels[list(set(full_indexes) - set(Test_indexes))]

    Train_indexes = list(set(full_indexes) - set(Test_indexes))
    return Test_indexes,Train_indexes
```

Then, we calculated the mean and std of the features from Training data-set and Normalize it. Then we normalize the hidden data based on these means and standard deviations. That procedure is necessary because if someone give us a Test-set in order to find their labels we must first normalize them so there is no batch effect.

1.1 Stratification

One common issue in data mining is the size of the data set. It is often limited. When this is the case, the test of the model is an issue. Usually, 2/3 of the data are used for training and validation and 1/3 for final testing. By chance, the training or the test set may not be representative of the overall data set. Consider for example a data set of 200 samples and 10 classes. It is likely that one of these 10 classes is not represented in the validation or test set.
To avoid this problem, you should take care of the fact that each class should be correctly represented in both the training and testing sets. This process is called stratification. One way to avoid doing stratification, regarding the training phase is to use k-fold cross-validation. Instead of having only one given validation set with a given class distribution, k different validation sets are used. However, this process does not guarantee a correct class distribution among the training and validation sets. So we can select the data in every fold based on the probability of a class.

```
def Stratification(labels):
 '''**Description:** Calculate the probability of each class, indexes, mean , std\n'''
 Percent_dic ={}
 mini = mt.inf
 for label in set(labels):
     count_label = list(labels).count(label)
     # calcilate the rare label
     mini = min(mini,count_label)
     indexies_label = list(np.where(labels == label)[0])

     # Correct label names
     label = "".join(label.split("'"))[1:]
     # list indexies per label
     Percent_dic[label] = [count_label,count_label/len(labels),indexies_label]
 return Percent_dic, mini
```

Because some classes are more rare than others, in my code there is no option for not stratified analysis. Thus, the number of folds must be equal (or +1) of the size of the most rare class. The most common class presents in 30% of the population. So we can conclude that a classifier with accuracy greater than 30% is not random. I tried to normalize the data but seems to be already normalized. Normalization seem that does not affect the result of KNN. My Naive Bayesian seems that is not working proper. I couldn't find the bug of my code and the results are around the random prediction. With normalization are much worse.

1.2 Cross Validation

Cross-validation, is a model validation technique for assessing the results of your classifier. It is mainly used ,when we want to estimate how accurately a predictive model will perform in practice. In a prediction problem, where a dataset of known data is given, we split our data in folds and every time we take the n-1 folds as training dataset and the one fold that left, as test dataset ("unknown data").

```
############### CROSS VALIDATION #############
def Cross_Validation( X, Folds, Train_indexes, All_Labels, number_of_neighbours, Method, D
accuracy = 0
for fold in Folds:
    valid_indx = fold
    valid_data = X[:,valid_indx].T
    train_indx = list(set(Train_indexes) - set(fold))
    train_data = X[:,train_indx].T
    Percent_dic,mini_fold = Stratification(X,All_Labels[train_indx])
    # ORDER LABELS AND PROBABILITIES
    Labels_list = []
    Percentage_ci_list = []
    for label in Percent_dic.keys():
        Labels_list.append(label)
        Percentage_ci_list.append( Percent_dic[label][1])


    #=====================================================#
    #################  CHOOSE METHOD ####################
    #=====================================================#
    if Method == "2":
        predict_labels = Naive(X,valid_indx,All_Labels,Labels_list,Percentage_ci_list,Perc

    elif Method == "1":
        ## DISTANCE MATRIX ##
        Dist_Matrix = np.zeros((valid_data.shape[0],train_data.shape[0]))

        for  i  in range(valid_data.shape[0]):
           for j in range(train_data.shape[0]):
               Dist_Matrix[i,j] = Dist_func(train_data[j,:],valid_data[i,:],Distance,S)
        Matrix = Dist_Matrix.copy()
        ## ACCURACY TEST ##
        predict_labels = K_nn(Matrix,All_Labels[train_indx],All_Labels[valid_indx],number_

    fold_accuracy = Accuracy(predict_labels,valid_indx,All_Labels)
    accuracy += fold_accuracy

```

```
38    return round(accuracy/len(Folds),2)
```

The goal of cross validation is to define a dataset to "test" the model in the training phase (i.e., the validation dataset), in order to limit problems like overfitting, give an insight on how the model will generalize to an independent dataset (i.e., an unknown dataset, for instance from a real problem) etc.

Common types of cross-validation:

- **Leave-p-out cross-validation (LpO CV)**
  Leave-p-out cross-validation involves using p observations as the validation set and the remaining observations as the training set. This is repeated on all ways to cut the original sample on a validation set of p observations and a training set.

```
1  ################ CREATE FOLDS #############
2  def Create_Folds (Percent_dic,size_of_folds,mini):
3   '''
4   **Description:** Split the data in Folds
5   '''
6   def_dict_per = Percent_dic.copy()
7   folds = []
8   for i in range(mini):
9       lista =[]
10      for j in Percent_dic.keys():
11          if i < mini-1:
12              per_list = rd.sample(def_dict_per[j][2],round(def_dict_
13                      per[j][1]*size_of_folds))
14          # Take the rest in last one
15          else :
16              per_list = def_dict_per[j][2]
17          lista+=per_list
18          # Delete so we cannot select
19          def_dict_per[j][2] = list(set(def_dict_per[j][2]) - set(per_list))
20      folds.append(lista)
21  return folds
```

- **Leave-one-out cross-validation (LOOCV)**
  Leave-one-out cross-validation is a particular case of leave-p-out cross-validation with $p = 1$.

```
1   ########### LEAVE ONE OUT ##############
2   Folds = [[x] for x in range(X.shape[1])]
```

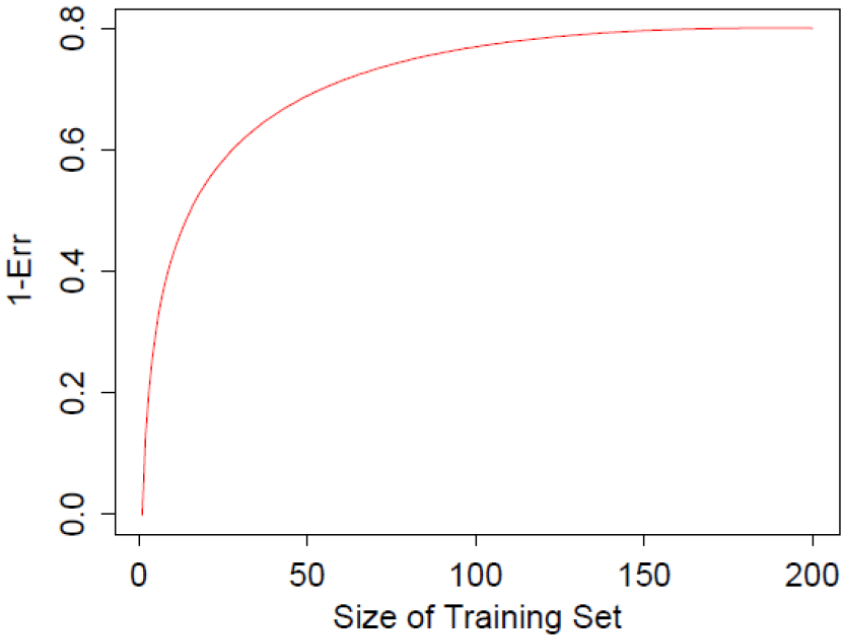


Figure 1: Hypotherical learning curve for classifier on a give task: a plot of Accuracy (1- Error) verse the size of the training set N. With a dataset of 200 observations, 5-fold cross-validation would use training sets of size 160, which would behave much like the full set. However, with a dataset of 50 observations fivefold cross-validation would use training sets of size 40, and this would result in a considerable overestimate of prediction error [**1**].

.

1.3 Limitations

Another major methodological concern is the problem of overfitting and underfitting. Overfitting is creating diagnostic models that may not generalize well to new data despite excellent performance on the training set. Since many algorithms are highly parametric and datasets consist of a relatively small number of high-dimensional samples, it is easy to overfit both the classifiers and the gene selection procedures especially when using intensive model search and powerful learners. As a result, the performance estimation is optimistic due to overfitting of the data. On the other hand, underfitting not learning characteristics that would generalize

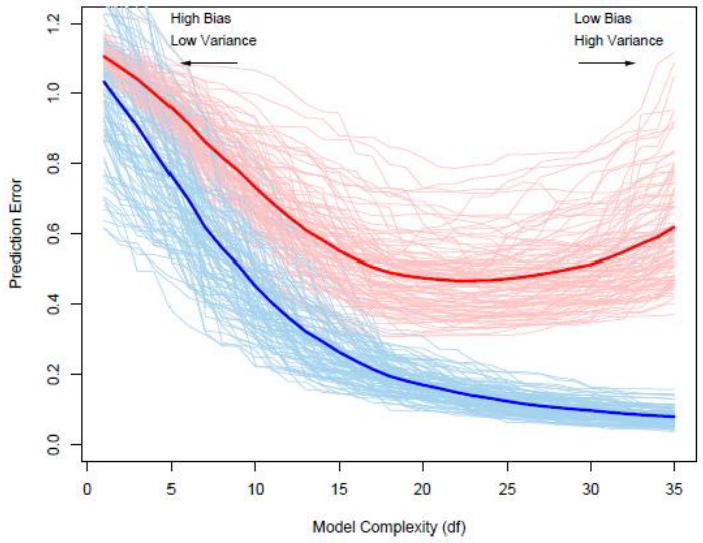


Figure 2: Behavior of test sample and training sample error as the model complexity is varied. the light blue curves show the training error, while the light red curves show the conditional test error for 100 training sets of size 50 each, as the model complexity is increased. The solid curve show the expected test error and the expected training error [**1**].

As the plot shows, the more the complexity increase the more accuracy we have until one threshold. In the plot above this threshold is around 20 number of complexity. After 20, the the CV gives us better results but our prediction in test data is falling.

2 K-nn

In KNN classification, the output is a class membership. An object is classified by a majority vote of its neighbors, with the object being assigned to the class most common among its k nearest neighbors (k is a positive integer, typically small). If k = 1, then the object is simply assigned to the class of that single nearest neighbor. Finally, kNN is powerful because it does not assume anything about the data, other than a distance measure can be calculated consistently between any two instances. The distance metric that we will in Knn depends on our data structure.
If classes are less than 10 and data are not sparse we can use :

- Euclidean
- Manhattan
- Chebychev

On the other hand if we have more than 10 classes and our data is sparse:

- Mahalanobis
- Cosine
- Correlation

```
def Dist_func(x,y,Distance,S):
D = x.shape[0]
if Distance =="1":
    Eucl_func =  lambda x,y: np.linalg.norm(x.reshape(D,1)-y.reshape(D,1))
    dist = Eucl_func(x,y)
elif Distance =="2":
    Manha_func = lambda x,y: abs(np.sum(x.reshape(D,1)-y.reshape(D,1)))
    dist = Manha_func(x,y)
elif Distance =="3":
    Mahala_func =  lambda x,y: np.sqrt((x-y).reshape(1,D).dot(np.linalg.inv(S)).dot((x.res

    dist = Mahala_func(x,y)
elif Distance =="4":
    dist = distance.cdist(x.reshape(1,D),y.reshape(1,D),"cosine")
elif Distance == "5":
    dist = distance.cdist(x.reshape(1,D),y.reshape(1,D),"correlation")
elif Distance == "6":
    dist = distance.cdist(x.reshape(1,D),y.reshape(1,D),'chebyshev')
return dist
```

I tried to plot the data after PCA but our samples does not cluster in groups, according to their labels.

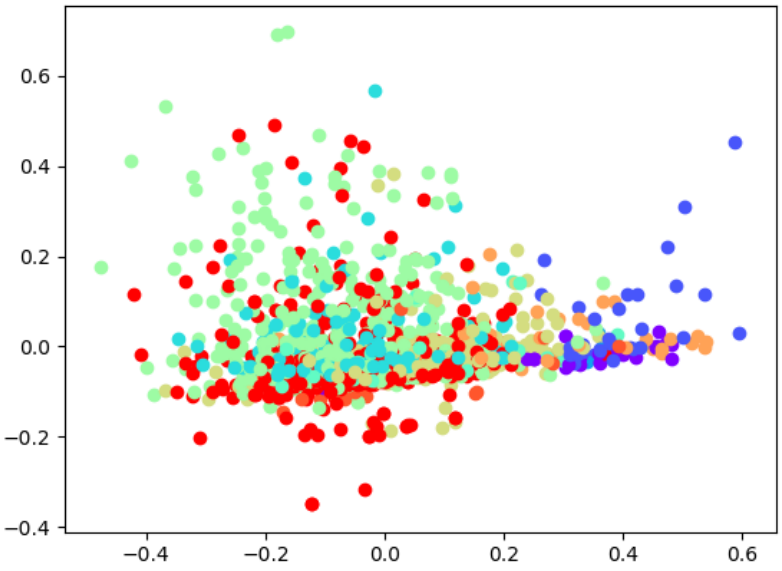


Figure 3: Principal Components Analysis. Plot the PCA1 and PCA2.

I think that the number of neighbours must be maximum 10-12, because the most rare class has only 5 samples, so in voting will be biased if we set the number o neighbours high. I run Knn for different k-neighbours and distance metrics.

```
def K_nn(Dist_Matrix,train_labels,valid_labels,num_of_neigh):
    predicted =[]
    for i in range(Dist_Matrix.shape[0]):
        k_list = []
        for k in range(num_of_neigh):
            min_dist_indx =list(Dist_Matrix[i,]).index(min(list(Dist_Matrix[i,])))
            Dist_Matrix[i,min_dist_indx] = mt.inf
            k_list.append(train_labels[min_dist_indx])
        vote_dic={}
        for k in k_list:
            if k not in vote_dic.keys():
                vote_dic[k] = 1
            else:
                vote_dic[k]+=1
```

```
            ### IF THERE IS A TIE TAKE THE CLOSEST NEIGHBOUR
            predict_label = [key for key,val in vote_dic.items() if val == max(vote_dic.values
            predicted.append(predict_label)
        return  predicted

    for number_of_neighbours in Neighbour_list:
        num_of_neig = int(number_of_neighbours)
        Test_data = X[:,Test_indexes].T
        Train_data = X[:,Train_indexes].T
        ## DISTANCE MATRIX ##
        Dist_Matrix = np.zeros((Test_data.shape[0],Train_data.shape[0]))

        for  i  in range(Test_data.shape[0]):
           for j in range(Train_data.shape[0]):
               Dist_Matrix[i,j] = Dist_func(Train_data[j,:],Test_data[i,:],Distance,S)
        Matrix = Dist_Matrix.copy()
        ## ACCURACY TEST ##
        predict_labels = K_nn(Matrix,All_Labels[Train_indexes],All_Labels[Test_indexes],num_of
```

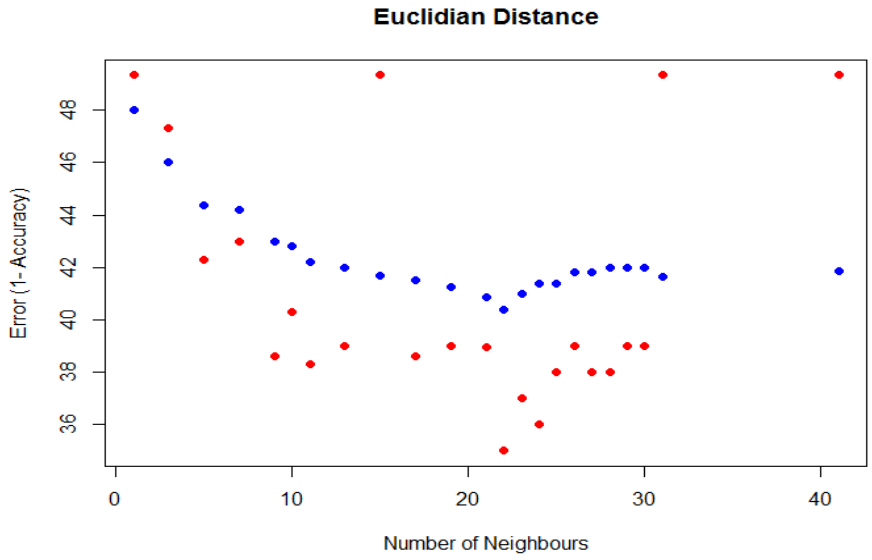


Figure 4: K-nn using euclidean distance as metric. Blue dots Cross validation accuracy, Red dots accuracy on test set.

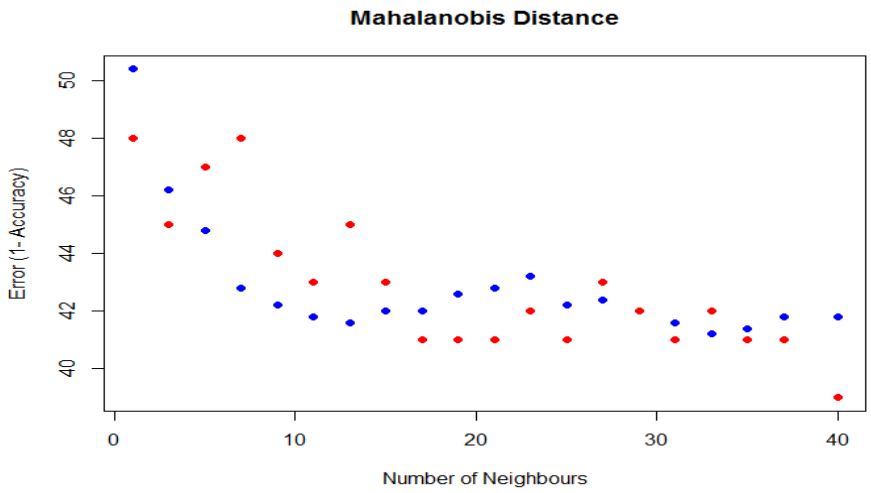


Figure 5: K-nn using Mahalanobis distance as metric. Blue dots Cross validation accuracy, Red dots accuracy on test set

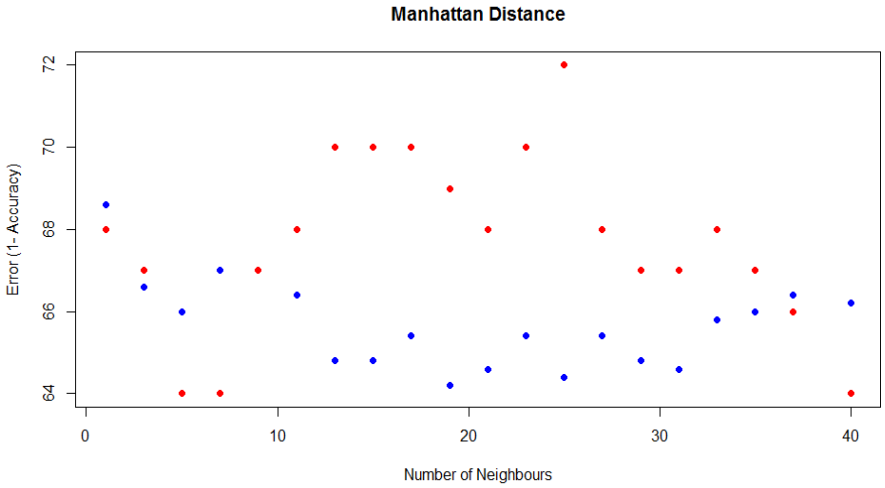


Figure 6: K-nn using Manhattan distance as metric. Blue dots Cross validation accuracy, Red dots accuracy on test set

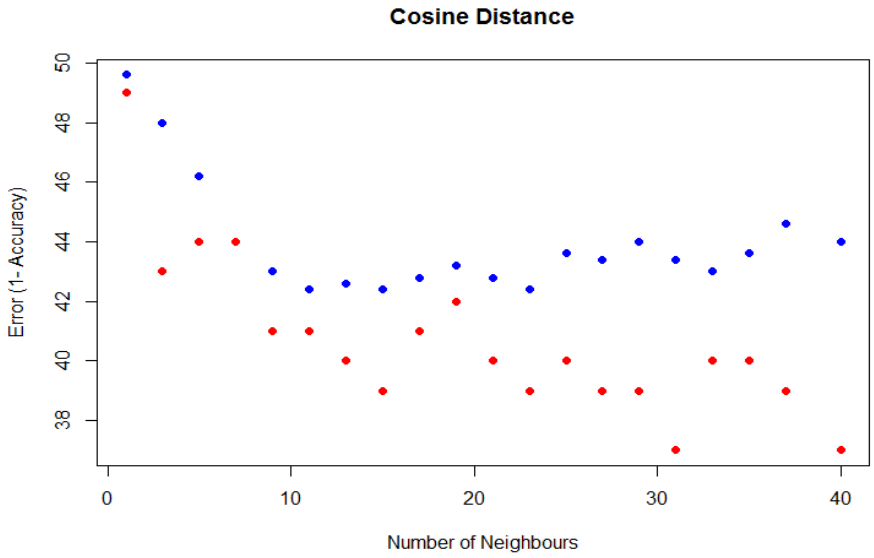


Figure 7: K-nn using Cosine distance as metric. Blue dots Cross validation accuracy, Red dots accuracy on test set

As we can see, Manhattan distance is not the appropriate distance for our data. From both Fig4,Fig5,Fig6 we can see that after 11 neighbours the accuracy has stabilized in the training data. Maybe the best choice is Euclidean distance for the combination of speed and accuracy.

Also, we expected that, when we hide bigger size of the training our accuracy must be lower. This assumption is not supported from Fig:8.

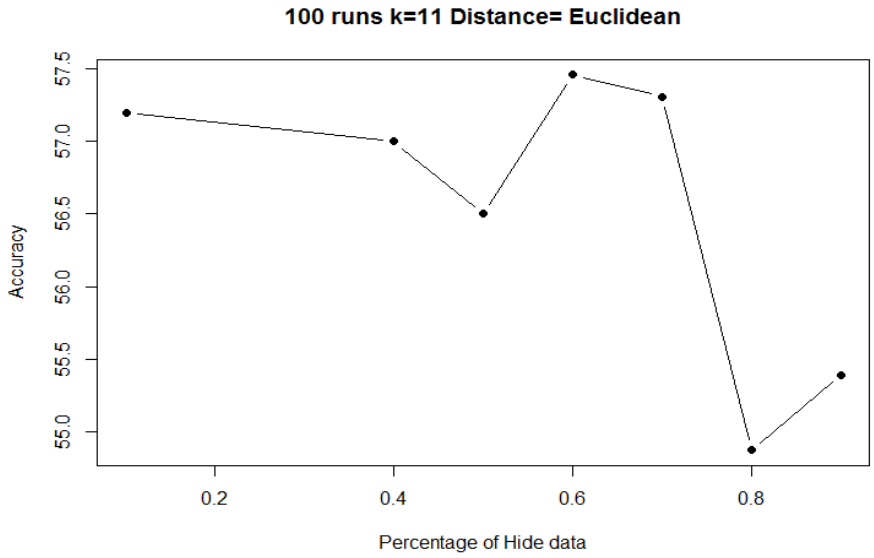


Figure 8: K-nn using Euclidean distance as metric. Run 100 times per case and calculate the mean accuracy.

3 Naive Bayesian

The Naive Bayes algorithm is an intuitive method that uses the probabilities of each attribute belonging to each class to make a prediction. It is the supervised learning approach you would come up with if you wanted to model a predictive modeling problem probabilistically.

Naive Bayes simplifies the calculation of probabilities by assuming that the probability of each attribute belonging to a given class value is independent of all other attributes. This is a strong assumption but results in a fast and effective method.

The probability of a class value given a value of an attribute is called the conditional probability. By multiplying the conditional probabilities together for each attribute for a given class value, we have a probability of a data instance belonging to that class.

To make a prediction we can calculate probabilities of the instance belonging to each class and select the class value with the highest probability. Naive bases is often described using categorical data because it is easy to describe and calculate using ratios. A more useful version of the algorithm for our purposes supports numeric attributes and assumes the values of each numerical attribute are normally distributed (fall somewhere on a bell curve). Again, this is a strong assumption, but still gives robust results.

```
#-------------------------------------------------------------#
###================== POSTERIOR ============================###
#-------------------------------------------------------------#

def Posterior_Prob (X,Vector,Labels_list, All_Labels,Percent_dic, Percentage_ci_list):
    '''
    Description: Find the Posterior Probability\n
    - Percent_dicnt_dic: Dictionary with keys the calsses and values the # counts, percent
    - Vector: A sample with features
    '''
    Prob_Matrix = np.zeros((len(Vector),len(Percent_dic.keys())))

    for clash in range(len(Labels_list)):
        for i in range(len(Vector)):
            mean = Percent_dic[Labels_list[clash]][3][i]
            sdv = Percent_dic[Labels_list[clash]][4][i]

            if sdv == 0:
                if Vector[i] == mean:
                    cond_prob = 1
                else:
                    cond_prob = 0
            else:
                cond_prob = stats.norm.pdf(Vector[i],loc=mean,scale=sdv)

            ####### ZERO conditional probability ###
            if  mt.isnan(cond_prob) or cond_prob == 0 :   # cond_prob < 1e-20 or
                # list of train for a feature
                xj_ci = list(np.around(X[i,Percent_dic[Labels_list[clash]][2]],decimals =
#                xj_ci = list(X[i,Percent_dic[Labels_list[clash]][2]])
                nc = xj_ci.count(Vector[i])
                n = Percent_dic[Labels_list[clash]][0]
                m =1
                p = 1/len(set(xj_ci))
                cond_prob = (nc + (m*p)) / (n+ m)

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

            Prob_Matrix[i,clash] = cond_prob

    product_array = np.prod(Prob_Matrix, axis = 0)

    max_posterior_indx = np.argmax(product_array *Percentage_ci_list)
    predicted_label = Labels_list[max_posterior_indx]
    return predicted_label
```

Bayesian classification procedures provide a natural way of taking into account any available information about the relative sizes of the sub-populations associated with the different groups within the overall population. Bayesian procedures tend to be computationally expensive and, in the days before Markov chain Monte Carlo computations were developed, approximations for Bayesian clustering rules were devised. Some Bayesian procedures involve the calculation of group membership probabilities: these can be viewed as providing a more informative outcome of a data analysis than a simple attribution of a single group-label to each new observation.

```
def Naive(X, Test_indexes, All_Labels, Labels_list, Percentage_ci_list,Percent_dic):
    predicted_test =[]
    Test_data = X[:,Test_indexes]
    for num_sample in range(len(Test_indexes)):
        Sample_Vector = Test_data[:,num_sample]
        predict = Posterior_Prob(X, Sample_Vector, Labels_list, All_Labels, Percent_dic, P
        predicted_test.append(predict)
    return predicted_test
```

I could not find the bug in my implementation so I have no results to present. Naive without normalization gave a accuracy like we choose random the label of each sample around 30%. Then I tried to normalize the data before but the accuracy was too low.

4 Bibliography

References

[1] Friedman, Tibshirani, Hastie *Elements of Statistical Learning*

[2] Bishop *Pattern Recognition And Machine Learning*