Techniques of Classification with Python Programming

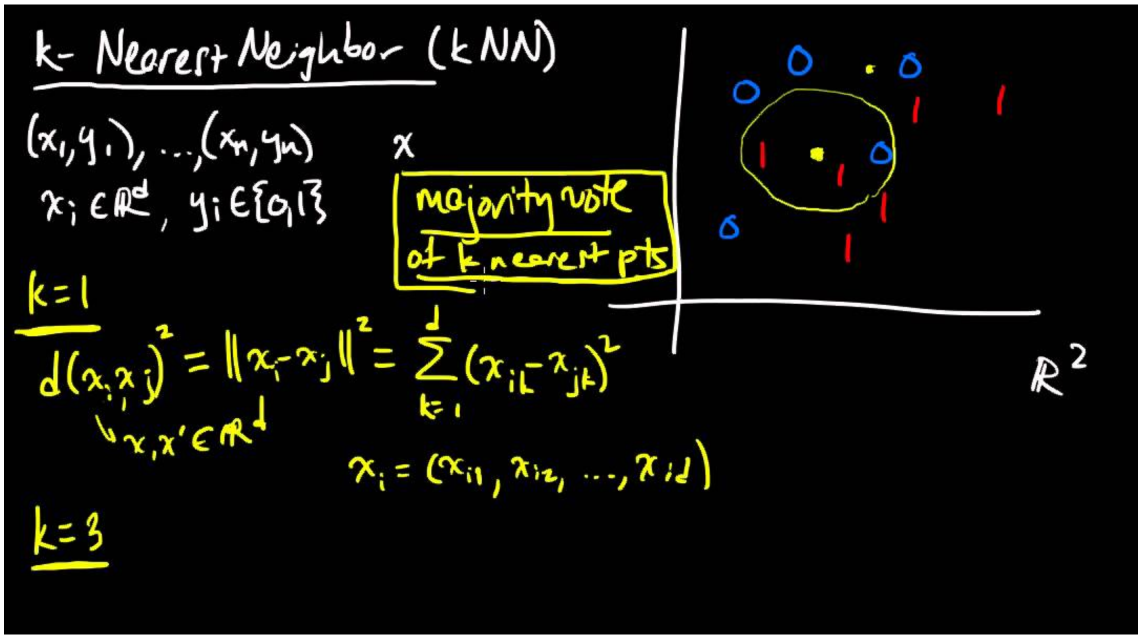
Chrysovalantou Kalaitzidou

May 2017

University of Crete Faculty of Medicine
Bioinformatics M.Sc.

Methods in Bioinformatics

Third Assignment


k- Nearest Neighbor (kNN)
$(x_1, y_1), \ldots, (x_n, y_n)$
$x_i \in \mathbb{R}^d$, $y_i \in \{0,1\}$
x
majority vote of k nearest pts
k=1
$d(x_i, x_j)^2 = \|x_i - x_j\|^2 = \sum_{k=1}^{d} (x_{ik} - x_{jk})^2$
$x, x' \in \mathbb{R}^d$
$x_i = (x_{i1}, x_{i2}, \ldots, x_{id})$
k=3
R^2

Contents

1 Introduction

The current report is part of the third assignment in class *Methods in Bioinformatics*, refering to ***Classification*** and, in particular, two of its methods ***K Nearest Neighbours*** and ***Naive Bayes***. Certain data set has been provided for analysis with each method implemented in Python programming language. In the following chapters the reader can find some theoretical notes for the methods and their algorithms for the implementation, the results derived from data set's analysis along with some discussion and, at last, the Python Code.
The data set that has been provided for analysis, is part of a project which references to *A Probablistic Classification System for Predicting the Cellular Localization Sites of Proteins* and resulted in the following:

- 1484 yeast proteins were classified into 10 classes with an accuracy of 55% for Yeast data.
- 336 E. coli proteins were classified into 8 classes with an accuracy of 81%.

with an ad hoc structured probability model. Also similar accuracy for Binary Decision Tree and Bayesian Classifier methods applied by the same authors unpublished results. For more information upon the discription of the data set, please visit `http://mlearn.ics.uci.edu/databases/yeast/yeast.names`

Our task was to perform classification in the *Yeast Data*, make predictions using a hidden set (arbitrary excluded from the data) and report the accuracy of our predictions, for each method.

Chrysovalantou Kalaitzidou

2 Theoretical Background

2.1 Supervised Machine Learning

Supervised machine learning is the search for algorithms that reason from externally supplied instances to produce general hypotheses, which then make predictions about future instances. In other words, the goal of supervised learning is to build a concise model of the distribution of class labels in terms of predictor features. The resulting classifier is then used to assign class labels to the testing instances where the values of the predictor features are known, but the value of the class label is unknown. The process of applying supervised Machine Learning to a real-world problem is described in Figure 1:

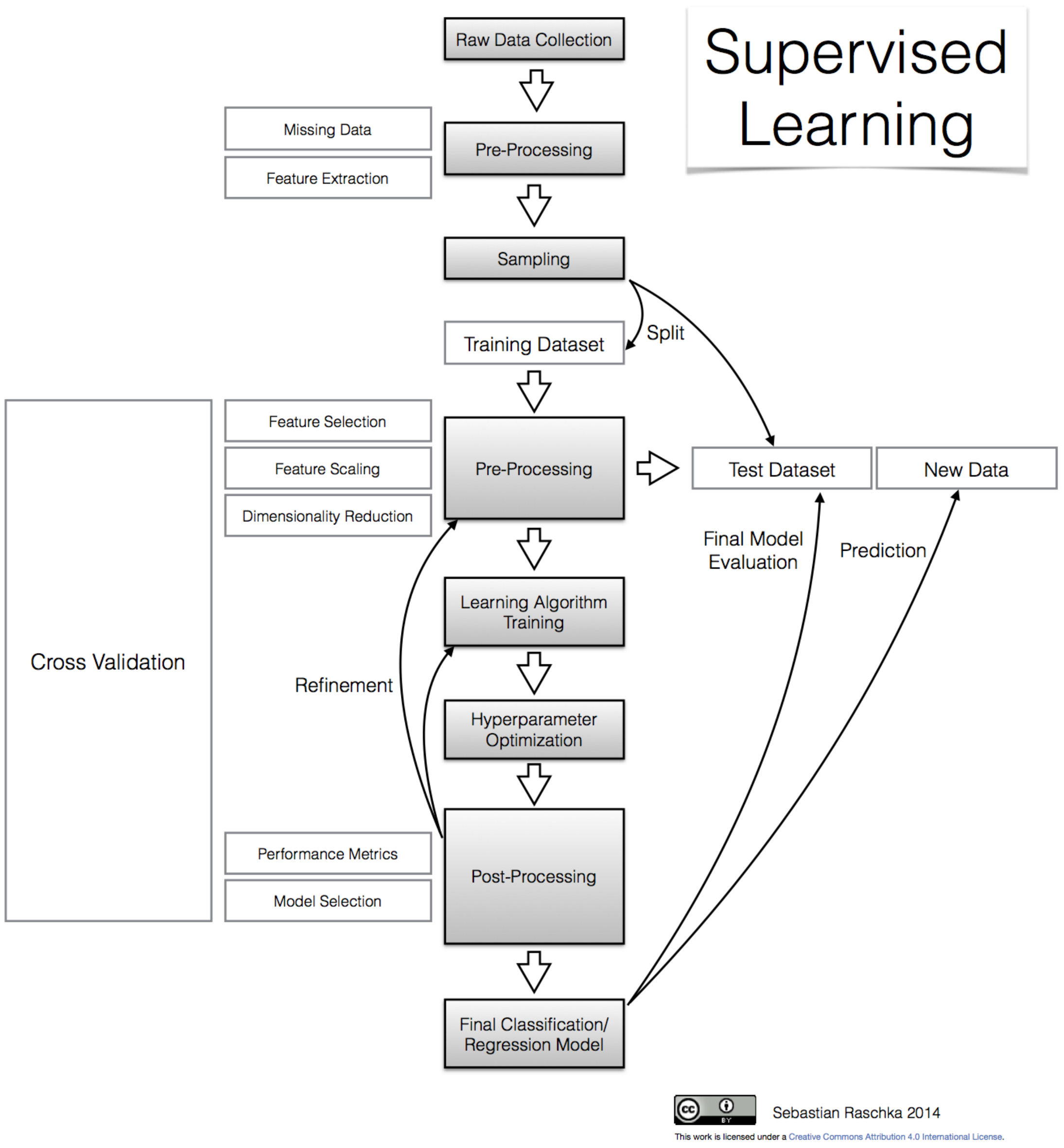


Figure 1: Machine Learning process

The most critical step in supervised machine learning is the choice of which specific learning algorithm should be used. Once preliminary testing is judged to be satisfactory, the classifier (mapping from unlabeled instances to classes) is available for routine use. The classifier's evaluation is most often based on *prediction accuracy*, the percentage of correct prediction divided by the total number of predictions. There are at least three techniques which are used to calculate a classifier's accuracy:

- One technique is to split the training set by using two-thirds for training and the other third for estimating performance.
- In another technique, known as **cross-validation**, the training set is divided into mutually exclusive and equal-sized subsets and for each subset the classifier is trained on the union of all the other subsets. The average of the error rate of each subset is therefore an estimate of the error rate of the classifier.
- **Leave-one-out validation** is a special case of cross validation. All test subsets consist of a single instance. This type of validation is, of course, more expensive computationally, but useful when the most accurate estimate of a classifier's error rate is required.

2.2 Stratifying Multi-Label Data

As we mentioned above, in supervised learning, experiments typically involve a first step of distributing the examples of a dataset into two or more disjoint subsets. When training data abound, the holdout method is used to distribute the examples into a training and a test set, and sometimes also into a validation set. When training data are limited, cross-validation is used, which starts by splitting the dataset into a number of disjoint subsets of approximately equal size. In classification tasks, the *stratified version* of these two methods is typically used, which splits a dataset so that the proportion of examples of each class in each subset is approximately equal to that in the complete dataset. **Stratification** has been found to improve upon standard cross-validation both in terms of bias and variance. In contrast, random distribution of multi-label training examples into subsets suffers from the following practical problem: it can lead to test subsets lacking even just one positive example of a rare label, which in turn causes calculation problems for a number of multi-label evaluation measures. The typical way these problems get by-passed in the literature is through complete removal of rare labels. This, however, implies that the performance of the learning systems on rare labels is unimportant, which is seldom true.
Stratified sampling is a sampling method that takes into account the existence of disjoint groups within a population and produces samples where the proportion of these groups is maintained. Achieving this kind of stratification is expected to be beneficial, in two directions:

- it is expected to improve upon random distribution in terms of estimate bias and variance.
- it will lower the chance of producing subsets with zero positive examples for one or more labels.

2.3 Algorithms

2.3.1 k Nearest Neighbours

Suppose each sample in our data set has n attributes which we combine to form an n-dimensional vector:

$$\mathbf{x} = (x_1, ..., x_n)$$

These n attributes are considered to be the *independent* variables. Each sample also has another attribute, denoted by y (the dependent variable), whose value depends on the other n attributes $\mathbf{x}$. We assume that y is a categoric variable, and there is a scalar function, f, which assigns a class, $y = f(x)$ to every such vectors. We do not know anything about f, otherwise there is no need for data mining, except that we assume that it is smooth in some sense. We suppose that a set of N such vectors are given together with their corresponding classes:

$$x^{(i)}, y^{(i)}, i = 1, 2, ..N$$

This set is referred to as the *training* set.
The problem we want to solve is the following: Supposed we are given a new sample where $x = u$. We want to find the class that this sample belongs. If we knew the function f , we would simply compute $v = f(u)$ to know how to classify this new sample, but of course we do not know anything about f except that it is sufficiently smooth.
The idea in **k-Nearest Neighbor methods** is to identify k samples in the training set whose independent variables x are similar to u, and to use these k samples to classify this new sample into a class, v. If all we are prepared to assume is that f is a smooth function, a reasonable idea is to look for samples in our training data that are near it, in terms of the independent variables, and then to compute v from the values of y for these samples. When we talk about neighbours we are implying that there is a distance or *dissimilarity measure* that we can compute between samples based on the independent variables, such as *Euclidean Distance, Mahalanobis, Manhattan, Cosine*, etc.
The simplest case is $k = 1$ where we find the sample in the training set that is closest, the nearest neighbour, to u and set $v = y$ where y is the class of the nearest neighboring sample. It is a remarkable fact that this simple, intuitive idea of using a single nearest neighbour to classify samples can be very powerful when we have a large number of samples in our training set. It is possible to prove that if we have a large amount of data and used an arbitrarily sophisticated classification rule, we would be able to reduce the misclassification error at best to half that of the simple $1 - NN$ rule.
For $k - NN$ we extend the idea of $1 - NN$ as follows: Find the nearest k neighbors of u and then use a *majority decision rule* to classify the new sample. The advantage is that higher values of k provide smoothing that reduces the risk of over-fitting due to noise in the training data. In typical applications k is in units or tens rather than in hundreds or thousands. Notice that if $k = n$, the number of samples in the training data set, we are merely predicting the class that has the majority in the training data for all samples irrespective of u. This is clearly a case of over-smoothing unless there is no information at all in the independent variables about the dependent variable.

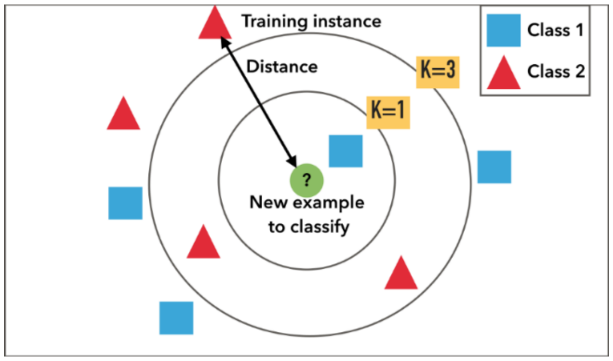


Figure 2: kNN presentation

2.3.2 Naive Bayes Classification

Naive Bayes classifiers are linear classifiers that are known for being simple yet very efficient. The probabilistic model of naive Bayes classifiers is based on Bayes' theorem, and the adjective naive comes from the assumption that the features in a dataset are mutually independent. In practice, the independence assumption is often violated, but naive Bayes classifiers still tend to perform very well under this unrealistic assumption. Especially for small sample sizes, naive Bayes classifiers can outperform the more powerful alternatives. Being relatively robust, easy to implement, fast, and accurate, naive Bayes classifiers are used in many different fields. Some examples include the diagnosis of diseases and making decisions about treatment processes, the classification of RNA sequences in taxonomic studies, and spam filtering in e-mail clients. However, strong violations of the independence assumptions and non-linear classification problems can lead to very poor performances of naive Bayes classifiers. We have to keep in mind that the type of data and the type problem to be solved dictate which classification model we want to choose. In practice, it is always recommended to compare different classification models on the particular dataset and consider the prediction performances as well as computational efficiency.
The probability model that was formulated by Thomas Bayes (1701-1761) is quite simple yet powerful; it can be writtendown in simple words as follows:

$$\text{posterior probability} = \frac{\text{conditional probability} \cdot \text{prior probability}}{\text{evidence}}$$

Bayes' theorem forms the core of the whole concept of naive Bayes classification. The posterior probability, in the context of a classification problem, can be interpreted as: *"What is the probability that a particular object belongs to class i given its observed feature values?"*
The general notation of the posterior probability can be written as

$$P(\omega_j|x_i) = \frac{P(x_i|\omega_j) \cdot P(\omega_j)}{P(x_i)}$$

Let

- x_i be the $feature$ vector of sample i, $i \in 1, 2, ., n$
- ω_j be the notation of *class* j, $j \in 1, 2, ., n$
- $P(x_i|\omega_j)$ be the probability of observing sample x_i given that it belongs to class ω_j .

The objective function in the naive Bayes probability is to maximize the posterior probability given the training data in order to formulate the *decision rule*:

$$\text{predicted class label} \leftarrow \underset{j=1,..m}{\text{argmax}} P(\omega_j|x_i)$$

Additional notes: [4]

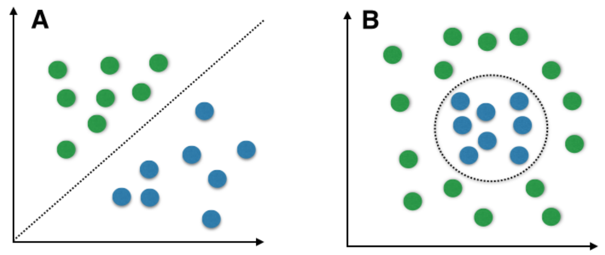


Figure 3: Linear (A) vs. non-linear problems (B). Random samples for two different classes are shown as colored spheres, and the dotted lines indicate the class boundaries that classifiers try to approximate by computing the decision boundaries. A non-linear problem (B) would be a case where linear classifiers, such as naive Bayes, would not be suitable since the classes are not linearly separable. In such a scenario, non-linear classifiers (e.g.,instance-based nearest neighbor classifiers) should be preferred.

3 Analysis

In 2.2, we talked about stratifying multi-label data which splits a dataset so that the proportion of examples of each class in each subset is approximately equal to that in the complete dataset. In our approach, we choose this method of distributing the samples of the dataset into k disjoint *folds*.

First of all, we hide a portion of 10% of the data set that we keep it *hidden* from the algorithm and only use it after the training takes places to compute some metrics that can give a hint on how the algorithm behaves. For each item in the test set, its "value" is predicted and then, using the built model, we compare it against its real "value". Accordingly, the *training set* is the data for which the algorithm knows the "labels" and which we will feed it to the training process to build our model:

Test and Train subsets

```python
def Splitting(Data,Labels):

    X = Data.T ; N = X.shape[1]  ## X: (8,1484)

    real_index = list(range(N))

    label_idx = {}          ## Keys: Classes, Values: the indices of their samples

    for i in range(Labels.shape[0]):
            if Labels[i] not in label_idx.keys():
                    label_idx[Labels[i]] = [i]
            else:
                    label_idx[Labels[i]].append(i)
    #print("label_idx = {}".format(label_idx))


    print("\n")
    print("Length of each class:")
    for k in label_idx:
            print("Class {}: {}".format(k,len(label_idx[k])))

    Hidden_set = []                         ## Indices of Hidden set
    Train_set = []                          ## The rest of them for Train (and Validation) set

    for key in label_idx:
            indices_removed = []

            n_01 = math.floor(0.1*len(label_idx[key])) #round(0.1*len(label_idx[key]))
            n_02 = np.random.choice(np.array(label_idx[key]),n_01,replace = False)

            Hidden_set.extend(n_02)

            indices_removed.extend(n_02)        ## remove index of Hidden:
            label_idx[key] = list(set(label_idx[key]) - set(indices_removed))

    print("\n")
    print("Length of each class after the split:")
    for k in label_idx:
            print("Class {}: {}".format(k,len(label_idx[k])))

    for idx in range(N):
            if idx not in Hidden_set:
                    Train_set.append(idx)
    return(Train_set,Hidden_set,label_idx)
```

What we have to mention here is that, we choose to hide 10% of each class, but we adjust this proportion to its floor number, as you can see in line 28. Thus, we make sure that the smallest class of our data set, which has only 5 samples will remain to the train set.

We proceed to the *stratified cross - validation* to calculate our classifies' accuracies. Let us have a look at the stratified k folding that train set is subjected to:

Stratification

```
 1  def Stratification(Train_set,Dict_idx,folds):
 2
 3          n_03 = min(len(Dict_idx[key]) for key in Dict_idx)
 4
 5          if folds > n_03:
 6                  folds = n_03
 7
 8          Partitions = []
 9
10          for i in range(folds):
11                  Partitions.append([])
12
13          #fold_size = round(len(Train_set) / folds)
14
15          Hold_indices = Dict_idx.copy()
16
17          for k in range(folds):
18                  for key in Dict_idx:
19
20                          indices_removed = []
21                          n_04 = round((1/folds)* len(Hold_indices[key]))
22
23                          if k <folds -1:
24                                  indices = np.random.choice(np.array(Dict_idx[key])
25                                  ,n_04,
26                                  replace = False)
27                          else:
28                                  indices = np.array(Dict_idx[key])
29
30                          Partitions[k].extend(indices)
31                          indices_removed.extend(indices)
32                          Dict_idx[key] = list(set(Dict_idx[key]) - set(indices_removed))
33
34          for i in range(folds):
35                  print("Samples in each fold: {} ".format(len(Partitions[i])))
36
37          return(Partitions,Hold_indices,folds)
```

Practically speaking, we have hidden a portion of the data set for *test*, while making sure that the smallest class remains whole for training. At this point, we prefer to enforce the algorithm so that, no matter how many folds it will be asked to split the train set for validation, it will take as optimal number of folds the length of the smallest class. In this way, we make sure that each fold during training will contain at least one sample of the smallest class. (lines: 5,6)

Since the length of the smallest class is 5, one round of cross-validation involves partitioning the rest of the data in 5 subsets, performing the analysis on one subset the **training** set, and validating the analysis on the other subset, the **validation** set. To reduce variability, multiple rounds of cross-validation are performed using different partitions, and the validation results are *averaged* over the rounds.

We have implemented two algorithms, as we have already discussed, *kNN* and *Naive Bayes.*

kNN, Validation

Before using kNN, let us revisit some of its assumptions:

- kNN assumes that the data is in a $feature$ space. More exactly, the data points are in a metric space. The data can be scalars or possibly even multidimensional vectors. Since the points are in feature space, they have a notion of distance. This need not necessarily be Euclidean distance although it is the one commonly used. We made use of six metrics to compare their accuracies:
    - (i) Euclidean
    - (ii) Manhattan
    - (iii) Mahalanobis
    - (iv) Cosine
    - (v) Correlation
    - (vi) Chebyshev
- We are also given a single number "k" . This number decides how many *neighbours* influence the classification and they are defined based on one of the above distance metrics.

For the number of k, we prefered to make use of the so called **Pigeonhole principle**. In mathematics, the pigeonhole principle states that if n items are put into m containers, with $n > m > 0$, then at least one container must contain more than one item. During validation, the train set contains the 10 different classes, as we have stratified it to do so. With the Pigeonhole principle, we assume that each of these classes will have at least two samples. So, the optimal number for k to begin with is $c + 1 = 11$, where c is the number of classes. Thus, hypothetically, if each class will have 2 samples, the additional +1 to c will give us 3 neigbours belonging to the same class.

For initial $k = 11$, and then for $k = 12, 13, 14, 15, 16, 17, 21, 31$ and using all the metrics we mentioned above, we concluded in an interval of mean accuracies. Bibliography and papers suggest that metric distances taking into account the samples distributions, such as Mahalanobis or Correlation,are expected to make better predictions. However, for the data set we worked with each metric gave round about the same accuracies as the other ones.

- **k= 11**

Metric	**Accuracy**
Euclidean	58.17
Manhattan	59.366
Mahalanobis	58.784
Cosine	59.614
Correlation	58.471
Chebyshev	58.525

- **k= 13**

Metric	**Accuracy**
Euclidean	59.351
Manhattan	59.243
Mahalanobis	59.082
Cosine	57.481
Correlation	58.554
Chebyshev	56.759

After validation process, we report the system's performance on the *hidden* test set. Indicatively, for these two numbers of neighbours we have:

- **k= 11**

Metric	**Accuracy**
Euclidean	56.552
Manhattan	55.172
Mahalanobis	62.069
Cosine	56.552
Correlation	64.828
Chebyshev	57.241

- **k= 13**

Metric	**Accuracy**
Euclidean	60.69
Manhattan	57.931
Mahalanobis	57.241
Cosine	56.552
Correlation	57.241
Chebyshev	59.31

In each run, for the same k and the same metric, accuracy is possible to change between 55% to 62%, for each k and for each metric. So, we have made many runs, kept the mean accuracies of the validation process as well as the accuracies after testing the *hidden* test , for the same number of neighbours, and we made a plot of their mean value. So after, 10 runs , for $k = 15$ neighbours and *Euclidean* distance as metric we gained the graph below:

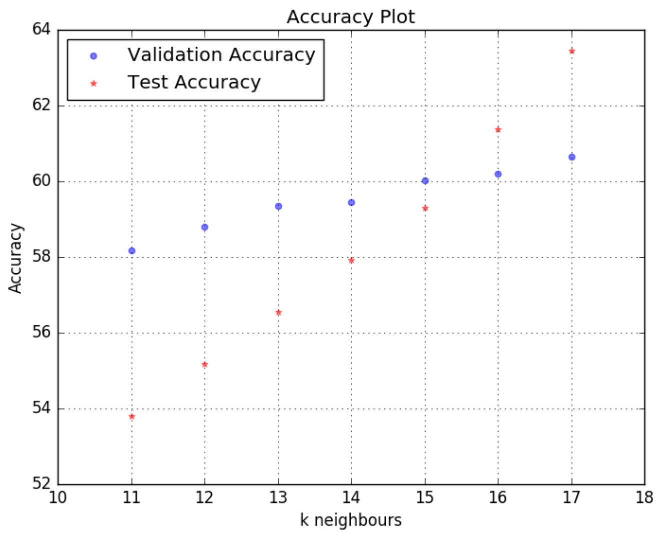


Figure 4: Mean accuracy of validation vs. mean test accuracy

And, respectively, the plot of the *mean error (1 - accuracy)* shall be:

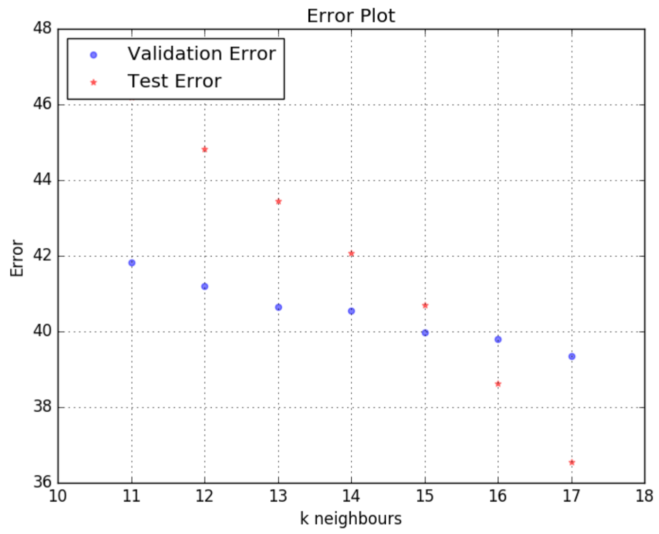


Figure 5: Mean error of validation vs. mean test error

We see, that error $\rightarrow 0$ as k increases. Therefore we would expect that the error rates decrease if k increases and from a certain $k = a$ onwards, it would be vice versa. Hence, there is a preference for k in a certain range.

In the plot below we see the mean accuracy as k increases, for the validation testing using Euclidean distance:

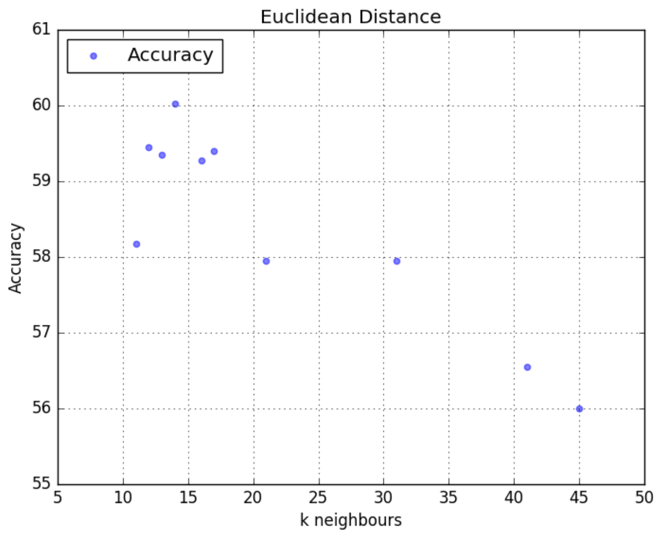


Figure 6: Accuracy rate for different k

Naive Bayes, Validation

Naive Bayes validation , after running the code 10 times, gave the mean accuracies below:

Accuracy rate	**Test accuracy**
59.264	48.966
60.133	51.724
47.863	46.207
58.636	59.31
48.074	59.31
57.358	53.103
48.171	48.966
47.577	53.793
48.339	49.655

For these two algorithms, we know a priori that:

k NN	**Naive Bayes**
Small Data	Huge Data
Slower for large data	Much faster than kNN
can't deal with noisy data	can deal with noisy data
provides high accuracy	requires a very large number of records
	for obtaining good results

A common method for comparing supervised ML algorithms is to perform *statistical* comparisons of the accuracies of trained classifiers on specific datasets. If we have sufficient supply of data, we can sample a number of training sets of size N, run the two learning algorithms on each of them, and estimate the difference in accuracy for each pair of classifiers on a large test set. The average of these differences is an estimate of the expected difference in generalization error across all possible training sets of size N, and their variance is an estimate of the variance of the classifier in the total set. Our next step is to perform **paired t-test** to check the:

- *Null Hypothesis:* the mean difference between the classifiers is zero.

Type I error is the probability that the test rejects the null hypothesis incorrectly (i.e. it finds a "significant" difference although there is none). Type II error is the probability that the null hypothesis is not rejected, when there actually is a difference. The test's Type I error will be close to the chosen significance level. In practice, however, we often have only one dataset of size N and all estimates must be obtained from this sole dataset. Different training sets are obtained by subsampling, and the instances not sampled for training are used for testing. Unfortunately this violates the independence assumption necessary for proper significance testing. The consequence of this is that Type I errors exceed the significance level. This is problematic because it is important for the researcher to be able to control Type I errors and know the probability of incorrectly rejecting the null hypothesis. [1]

Depending on these, we perform a *t-test* between 10 mean accuracy rates of kNN validation and 10 mean accuracy rates of *Naive bayes*. With null hypothesis above. The result amazingly agrees with these assumptions since we reject null Hypothesis, for the *p-value* is smaller than the significance level (0.05).

t test for the mean of accuracy rates

```
Welch Two Sample t-test

data:  x and y
t = -3.6248, df = 9.6798, p-value = 0.004906
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 -10.435568  -2.468032
sample estimates:
mean of x mean of y
  52.5070   58.9588
```

- x : the 10 mean accuracy rates of Naive Bayes validation and
- y: the 10 mean accuracy rates of k NN validation

In conclusion, there is no right algorithm. It always depends on the data set provided for analysis whether to choose a specific classification algorithm or not. In levels that one algorithm fails, some othermay be stronger and vice versa.

4 Suggested Bibliography

References

[1] S. B. Kotsiantis *Supervised Machine Learning: A Review of Classification Techniques* http://www.informatica.si/index.php/informatica/article/viewFile/148/140

[2] Konstantinos Sechidis, Grigorios Tsoumakas, and Ioannis Vlahavas *On the Stratification of Multi-Label Data* http://lpis.csd.auth.gr/publications/sechidis-ecmlpkdd-2011.pdf

[3] Sebastian Raschka *Python Machine Learning*

[4] Sebastian Raschka *Naive Bayes and Text Classification I* https://arxiv.org/pdf/1410.5329.pdf

[5] Li-Yu Hu, Min-Wei Huang,Shih-Wen Ke and Chih-Fong Tsai *The distance function effect on k-nearest neighbor classification for medical datasets* https://www.ncbi.nlm.nih.gov/pmc/articles/PMC4978658/

Appendices

A Python Codes

File: main

```python
import time
import os
from Functions import*
from collections import Counter

#from Knn import*


print("##.........Classification...........##")

##.............. Loading Dataset.................##

Data,Labels = Data_Loading("yeast.data.txt")

##...... Hide samples of the main data set.........##

Train_set,Hidden_set, Dict_idx = Splitting(Data,Labels)

##...... Splitting the Train Set...................##

folds = int(input("Give number of folds: "))

Partitions,Hold_indices,Folds = Stratification(Train_set,Dict_idx,folds)

##.......Main methods and crossvalidation.........##

Method = input("Choose Method.\nA: K Nearest Neighbours\nB: Naive Bayes: ")

##.......Mean Accuracy, with k folding training....##
Mean_Accuracy = Validation(Partitions,Data,Labels,Hold_indices,Train_set,Folds,Method)

##.......Main Classification: Choose Either method to predict class for hidden set....##

print("\n####.......... Predicted Classification for new data set..........####")

Hidden_Matrix, Main_Matrix, Main_Labels,Hidden_Labels = Hidden_Values(Data, Train_set, Hidden_set,
    Labels)

if Method == 'A':
  print("\n####........Classifier chosen : K nearest neighbours, KNN........####")

  k = int(input("Give number of Neighbours: "))
  Metric = (input("Give Metric Distance:\na. Euclidean\nb. Manhattan\nc. Mahalanobis\nd. Cosine\ne.
      Correlation\nf. Chebyshev : "))

  Predicted_Classes = Knn(Data,Main_Matrix,Hidden_Matrix,Main_Matrix,Main_Labels,k ,Metric)

else:
  print("\n####..........Classifier chosen : Naive Bayes....................####")

  Predicted_Classes = Naive_Bayes(Data,Labels,Hold_indices,Train_set,Hidden_set)


print("\nWith mean accuracy {} the predicted class for its sample of the new data set shall be :\n
    {}".format(Mean_Accuracy,Predicted_Classes))

Proportion = Accuracy(Hidden_Labels,Predicted_Classes)

print("\nThe hidden data set has been correctly classified, with {} ".format(round(Proportion*100,3)))
print("Which means that error, subsequently, is : {}".format(100-round(Proportion*100,3)))

c = Counter(Predicted_Classes)

#print("\nMost common class for the samples: {}".format(max(Predicted_Classes,key=
    Predicted_Classes.count)))
dif_classes = len(set(Predicted_Classes))
#print(dif_classes)
print("\nMost common class for the samples: {}".format(c.most_common(dif_classes)))
```

File: Functions

```python
#####.........................File with general functions......................#####

import numpy as np
import math
import random

from Knn import*
from Bayes import*

#####.........................Data Set loading................................#####

def Data_Loading(filename):

  Data = np.loadtxt(filename,usecols =tuple(np.arange(1,9)))
  Labels = np.loadtxt(filename,usecols=[9] , dtype=str)

  return(Data,Labels)


#####............................Splitting tha Data Set..................................................#####
### Take 10% out of the original data set, for Hidden data. The selection shall be random...................###
##...however, I choose to take 10% of each class. I tried different tricks so as to make sure that hidden.....##
##..will contain or not samples of the smallest class. Sometimes it remains, practically, untouchable.........##
#........thus, I make sure that during Training, each fold will contain samples from each class................#
#.......while some others,the hidden set contains more samples, including samples of the smallest class........#

def Splitting(Data,Labels):

  X = Data.T ; N = X.shape[1]  ## X: (8,1484)

  real_index = list(range(N))

  label_idx = {}         ## Keys: Classes, Values: the indices of their samples

  for i in range(Labels.shape[0]):
    if Labels[i] not in label_idx.keys():
      label_idx[Labels[i]] = [i]
    else:
      label_idx[Labels[i]].append(i)
  #print("label_idx = {}".format(label_idx))


  print("\n")
  print("Length of each class:")
  for k in label_idx:
    print("Class {}: {}".format(k,len(label_idx[k])))

  Hidden_set = []           ## Indices of Hidden set
  Train_set = []            ## The rest of them for Train (and Validation) set

  for key in label_idx:
    indices_removed = []

    n_01 = math.floor(0.1*len(label_idx[key])) #round(0.1*len(label_idx[key])) #math.floor(0.1*len(label_idx[key]))
    n_02 = np.random.choice(np.array(label_idx[key]),n_01,replace = False)

    Hidden_set.extend(n_02)

    indices_removed.extend(n_02)
    label_idx[key] = list(set(label_idx[key]) - set(indices_removed))  ## remove index of Hidden

  print("\n")
  print("Length of each class after the split:")
  for k in label_idx:
    print("Class {}: {}".format(k,len(label_idx[k])))

  for idx in range(N):
    if idx not in Hidden_set:
      Train_set.append(idx)

  print("Lenght of Train set: {} \nLength of Hidden set: {}".format(len(Train_set),len(Hidden_set)))

  return(Train_set,Hidden_set,label_idx)


#####.......................Split the Train Set in specific number of folds....................#####
### If given number of folds is greater than the minimum class'es length, I take the length .....###
##..of this class as optimal number of folds. Thus, I ensure that its fold will contain not only .##
#......samples of each class but (at least) one sample of the smallest class for certain..........#
```

```python
def Stratification(Train_set,Dict_idx,folds):

  n_03 = min(len(Dict_idx[key]) for key in Dict_idx)

  if folds > n_03:
    folds = n_03


  ## I want practically 5 folds with equal number of indices (samples) ##
  ##........taken from the Train_set list without replacement..........##

  Partitions = []

  for i in range(folds):
    Partitions.append([])

  #fold_size = round(len(Train_set) / folds)

  Hold_indices = Dict_idx.copy()

  for k in range(folds):
    for key in Dict_idx:

        indices_removed = []
        n_04 = round((1/folds)* len(Hold_indices[key]))

        if k <folds -1:
          indices = np.random.choice(np.array(Dict_idx[key]),n_04,replace = False)
        else:
          indices = np.array(Dict_idx[key])

        Partitions[k].extend(indices)
        indices_removed.extend(indices)
        Dict_idx[key] = list(set(Dict_idx[key]) - set(indices_removed))

  for i in range(folds):
    print("Samples in each fold: {} ".format(len(Partitions[i])))

  return(Partitions,Hold_indices,folds)


#####.................Cross Validation with stratified number of folders: 5..................#####

def Validation(Partitions,Data,Labels,Hold_indices,Train_set,folds,Method):

  print("###........Cross validation with {} folds:........###".format(folds))


  if Method == 'A':
    print("\n###.........Classifier Chosen : K nearest neighbours, KNN.........###")
    k = int(input("Give number of Neighbours: "))
    Metric = (input("Give Metric Distance:\na. Euclidean\nb. Manhattan\nc. Mahalanobis\nd. Cosine\ne.
         Correlation\nf. Chebyshev : "))
  else:
    print("\n###.........Classifier Chosen : Naive Bayes.........###")

  X = Data.T ; N = X.shape[1]
  Accuracies = []

  for i in range(folds):

    Test_idx = Partitions[i]
    Train_idx = list(set(Train_set) - set(Partitions[i]))

    Train_labels = Labels[Train_idx]          ## keep current  idx for this folding
    Test_labels  = list(Labels[Test_idx])

    Test_Matrix = np.array(X[:,Test_idx])
    Train_Matrix = np.array(X[:, Train_idx])
    Stack_Matrix = np.array(X[:, Train_set])
    #print(Test_Matrix.shape)
    #print(Train_Matrix.shape)

    if Method == 'A':
      # k = int(input("Give number of Neighbours: "))
      # Metric = (input("Give Metric Distance:\nE: Euclidean\nM: Manhattan\nH: Mahalanobis: "))
      Predictions = Knn(Data,Train_Matrix,Test_Matrix,Stack_Matrix,Train_labels,k,Metric)
    else:
      Predictions = Naive_Bayes(Data,Labels,Hold_indices,Train_idx,Test_idx)

    proportion = Accuracy(Test_labels,Predictions)
    Accuracies.append(proportion)

    print("Validation: Fold {}, accuracy: {} ".format(i, round(Accuracies[i]*100,3)))

  Mean_Accuracy = round((sum(Accuracies) / float(len(Accuracies)))*100,3)
  Error = 100 - Mean_Accuracy
  print("\nMean Accuracy: {}".format(Mean_Accuracy))
  print("Mean Error: {}".format(Error))

  return(Mean_Accuracy)

def Accuracy(Test_labels,Predictions):

  #Accuracies = []
  Common_indices = []
  for label_01, label_02 in zip(Test_labels, Predictions):
    if label_01 == label_02:
      Common_indices.append(label_01)
  #print(len(Common_indices))
  proportion = len(Common_indices)/len(Test_labels)
  #Accuracies.append(proportion)

  return(proportion)



def Hidden_Values(Data, Train_set, Hidden_set, Labels):

  X = Data.T ; N = X.shape[1]

  Hidden_Matrix = np.array(X[:,Hidden_set])
  Main_Matrix   = np.array(X[:,Train_set])

  Hidden_Labels = Labels[Hidden_set]
  Main_Labels   = Labels[Train_set]

  return(Hidden_Matrix,Main_Matrix,Main_Labels,Hidden_Labels)
```

File: k NN

```python

from Functions import*
from scipy.spatial import distance
#import numpy as np

def Knn(Data,Train_Matrix,Test_Matrix,Stack_Matrix,Train_labels,k ,Metric):



  X = Data.T; N= X.shape[1]

  n = Test_Matrix.shape[1]
  m = Train_Matrix.shape[1]

  Predictions = []

  for i in range(n):
    vec_01 = Test_Matrix[:,i]
    Distances = np.zeros(m)

    for j in range(m):

      vec_02 = Train_Matrix [:,j]

      Distances[j] = dist_function(Stack_Matrix,Metric,vec_01,vec_02)

    Sorted_idx = np.argsort(Distances)
    k_indices  = Sorted_idx[0:k]

    k_labels  = list(Train_labels[k_indices])

    predict_class = max(k_labels,key=k_labels.count)

    Predictions.append(predict_class)

  return(Predictions)


####..................Basic Metric Distances Functions.................#####

def euclidean_distance(x, y):
  return np.sqrt(np.sum((x-y)**2))

def Manhattan_Distance(x,y):
  return np.sum(np.abs(x-y))

def Mahalanobis_Distance(Arr,x,y):
  D = Arr.shape[0]
  S = np.cov(Arr)
  return np.sqrt((x-y).reshape(1,D).dot(np.linalg.inv(S)).dot((x-y).reshape(D,1)))




## Change distance function here according to needs

def dist_function(Arr,Distance,x, y):
  if Distance == 'a':
    return euclidean_distance(x, y)
  elif Distance == 'b':
    return Manhattan_Distance(x,y)
  elif Distance == 'c':
    return Mahalanobis_Distance(Arr,x,y)
  elif Distance == 'd':
    D = x.shape[0]
    return distance.cdist(x.reshape(1,D),y.reshape(1,D),"cosine")
  elif Distance == 'e':
    D = x.shape[0]
    return distance.cdist(x.reshape(1,D),y.reshape(1,D),"correlation")
  else:
    D = x.shape[0]
    return distance.cdist(x.reshape(1,D),y.reshape(1,D), 'chebyshev')
```

File: Naive Bayes

```python
from Functions import*
import scipy.stats



def Naive_Bayes(Data,Labels,Hold_indices,Train_set,Test_set):

  #Train_set,Test_set,Data_dict = Splitting(Data,Labels)

  X = Data.T ; D= X.shape[0]; N = X.shape[1]

  real_index = list(range(N))
  k = len(Hold_indices) #; print(k)

  Order_labels = []     ## Ordering Labels

  for i in Hold_indices.keys():
    Order_labels.append(i)

  Class_Priors = np.zeros(k)

  for c in range(len(Order_labels)):
    key = Order_labels[c]
    Class_Priors[c] = len(Hold_indices[key]) / len(Train_set)

  print("The Prior Probabilty of its Class is: {}".format(Class_Priors))

  ### Classes: 10 objects, each one an array of the samples belonging to this particular class.
  ##  Means:   10 objects, each one an array of the means of each variable, in this particular class.
  # Stdv:  10 objects, each one an array of the std of each variable, in this particular class.

  Classes = np.zeros(k,dtype = object)
  Means = np.zeros(k,dtype = object)
  Stdv  = np.zeros(k,dtype = object)

  for c in range(len(Order_labels)):
    key = Order_labels[c]

    Classes[c] = np.array(X[:,Hold_indices[key]])
    Means[c]  = np.mean(Classes[c], axis =1)
    Stdv[c]   = np.std(Classes[c], axis=1)

  Train_Array = np.array(X[:, Train_set]); print("Shape of Train Matrix:
      {}".format(Train_Array.shape))
  Test_Array   = np.array(X[:, Test_set]);  print("Shape of Test Matrix: {}".format(Test_Array.shape))

  m = Test_Array.shape[1]   #  Samples
  n = Test_Array.shape[0]   #  Variables

  Predicted_Labels = []

  for i in range(m):
    Probability = np.zeros(k, dtype=object)
    for c in range(len(Order_labels)):
      key = Order_labels[c]
      Predictions = np.zeros(n)
      for j in range(n):
        Conditional = scipy.stats.norm.pdf(Test_Array[j,i], loc = Means[c][j], scale = Stdv[c][j])
        ## zero conditional probability
        if Conditional == 0 or np.isnan(Conditional):
          #print(Test_Array[j,i])
          vec = np.array(X[j, Hold_indices[key]])
          n_c = list(vec).count(Test_Array[j,i]) #; print("nc:{}".format(n_c))
          n_l = len(Hold_indices[key])
          t   = len(set(Hold_indices[key])); p = 1/t
          m   = 1

          Conditional = (n_c + m*p)/(n_l + m)

        Predictions[j] = Conditional
      Product = np.prod(Predictions)
      Probability[c] = np.dot(Product,Class_Priors[c])

    Max_Prob = np.argmax(Probability)
    Predicted_Labels.append(Order_labels[Max_Prob])

  return(Predicted_Labels)
```