Artificial Intelligence: Knowledge Representation and Planning

CM0472: Assignment 2 – Spam Filter

Team:
Simion Vrabii - [867882]
Ivanoiu Alexandru Paul - [870383]

Index:

Assignment:

Write a spam filter using discriminative and generative classifiers. Use the "***Spambase***" dataset which already represents spam/ham messages through a bag-of-words representations through a dictionary of 48 highly discriminative words and 6 characters.

The first 54 features correspond to word/symbols frequencies; ignore features 55-57; feature 58 is the class label (1 spam/0 ham).

1. Perform SVM classification using **linear, polynomial of degree 2, and RBF kernels** over the **TF/IDF** representation.

   Can you transform the kernels to make use of angular information only (i.e., no length)? Are they still positive definite kernels?

2. Classify the same data also through a **Naive Bayes** classifier for continuous inputs, modelling each feature with a **Gaussian distribution**, resulting in the following model:
   a. $p(y = k) = \alpha_k$
   b. $p(x|y = k) = \prod_{i=1}^{D}[(2\Pi\sigma_{ki}^2)^{-\frac{1}{2}} exp\{-\frac{1}{2\sigma_{ki}^2}(x_i - \mu_{ki})^2\}]$

   where αk is the frequency of class k, and μ_{ki}, $\sigma^2 ki$ are the means and variances of feature i given that the data is in class k.

3. Perform **KNN** classification with **k=5**

Provide the code, the models on the training set, and the respective performances in **10-way cross validation**.

Explain the differences between the three models.

P.S. you can use a library implementation for SVM, but do implement the Naive Bayes on your own. As for k-NN, you can use libraries if you want, but it might just be easier to do it on your own.

Goal of this assignment:

The goal of this assignment is use the dataset from ***"Spambase"*** and perform the SVM classification [using the: linear/ polynomial of degree 2 / RBF kernels] , Naive Bayes classification and KNN classification.

For each classifier we must perform a 10-way cross validation and once we have done all this steps, we have to compare the performance of each classifier with the others so we can understand and reflect on the results and check what are the ups and downs of each of them and what affects the performance.

What is a classification problem and what are the types of classifiers?

Informal explanation:

In machine learning one of the most if not the most common problem is the **classification problem**.

The first part of the hole process is the input data that is used by a **classification algorithm** to learn from it and then in the future based on what it learned it will be able to assign new **labels** to the new data that the algorithm never saw before.

We mentioned before the term **label**. Let us see what this label is in our case. So, in our assignment the labels are two and their representation is if a message is **SPAM** or **HAM**.

Now that we have the two notions, **label** and **classifier**, we can make an example so we can understand better how these two things work together. So the first thing, given a dataset that contains "E-mails" labeled **1 [SPAM]** and **0 [HAM],** a classification algorithm will learn a model that will assign a label to a new email never seen before based on what it learned from the input dataset.

Formal explanation:

Predictors of our problems:

$$\boldsymbol{X} = \{\boldsymbol{X_1} \ldots . \boldsymbol{X_N}\} \in \boldsymbol{\chi}$$

Target variables also known as labels in our case we have two of them: {0 and 1}:

$$\boldsymbol{Y} = \{\boldsymbol{0}, \boldsymbol{1}\} \in \boldsymbol{\Upsilon}$$

The classification algorithm contains a learning function that learns from the training set:

$$\boldsymbol{\phi} : \boldsymbol{X} \rightarrow \boldsymbol{Y}$$

Taking into consideration our problem we can say that $\boldsymbol{\Upsilon} = \{\boldsymbol{0}, \boldsymbol{1}\}$ and so we get the following:

$$\boldsymbol{\phi}(\boldsymbol{x}) = \boldsymbol{g}(\boldsymbol{f}(\boldsymbol{x}))$$

where f(x) allows us to separate the objects from class 0 from the objects of class 1 **and** g(z) where z = f(x) is the function that returns the right label to assign to the x according to the result of f(x).

After all this we can write the final formula for our problem:

$$g(z) = \begin{cases} 1 \text{ if the message is spam} \\ 0 \text{ if the message is ham} \end{cases}$$

Now that we talked a little bit about the classification problem, we can talk about how many categories there are for the learning algorithms but before that lets see what a **Generative** and **Discriminative** classifier means.

Some machine learning models belong to either the "generative" or "discriminative" model categories. But what is the difference between these two categories of models? What does it mean for a model to be **discriminative** or **generative**?

So the **generative** models are the once that include the distribution of the data set, returning a probability for a given example. **Generative** models are often used to predict what occurs next in a sequence.

Discriminative models are used for either classification or regression and they return a prediction based on conditional probability.

So during our course we saw that there are 3 types of learning algorithms: **supervised learning**, **unsupervised learning** and **reinforces learning** but we also know that there is another category which is called **semi-supervised learning**.

Supervised learning:

The idea of supervised learning follows the idea of learning from example is like having a teacher that teaches you what to do. The supervised learning takes in input a set of observation X and its label Y and it tries to learn a model good enough that after the training it will be able to label the new object in the right way.

Unsupervised learning:

All the observations in the dataset are unlabeled and the algorithms learn to inherent structure from the input data.

Reinforced learning:

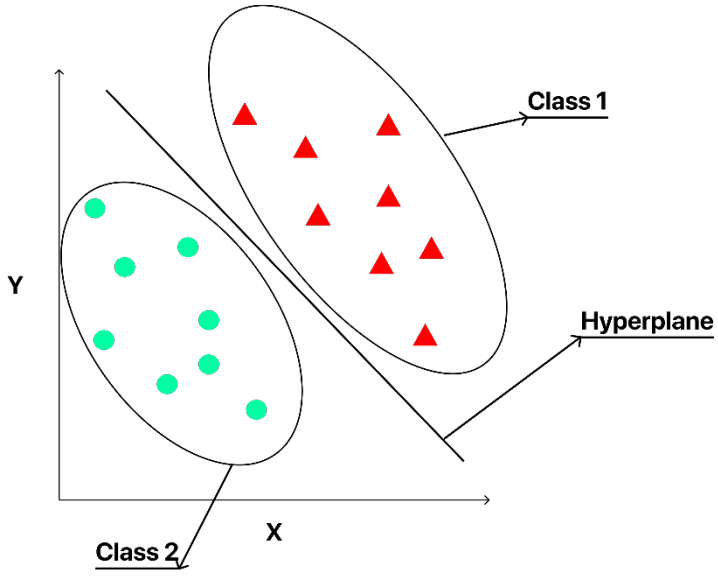
This family of models consists of algorithms that use the estimated errors as rewards or penalties. If the error is big, then the penalty is high and the reward low. If the error is small, then the penalty is low and the reward high.

Theory explanation behind SVM and behind the first point of the assignment:

Starting with the SVM:

The support vector machines are a type of ML classifier that is one of the most popular kinds of classifiers and makes part of the discriminative classifiers. We use support vector machines for the following things: numerical prediction, classification, and pattern recognition tasks.

SVM draw a decision boundary between data points of the two classes. This decision boundary is called hyperplane and is aiming to separate the data points into classes. You can have more than one decision boundary but we are aiming just for one that is as large as possible so that the distance between any point of the two classes and the boundary is maximized.



Formally we can see the SVM in the following way based on the theory seen above:

$$h_{w,b}(x) = g(w^T x + b)$$

$$g(z) = \begin{cases} 1 & if\ z > 0 \\ 0 & otherwise \end{cases}$$

Explanation of the formulas:

1. $w^T x + b$ represents a separating hyperplane for the 2 classes (check photo to understand better)
2. $g(z)$ is the decision function which returns the predicted class for the observation x

That being said our algorithm would predict **1 [SPAM]** if $h_{w,b}(x) > 0$ and **0 [HAM]** otherwise.

The larger $w^T x + b$ the higher is the probability that the label is 1.

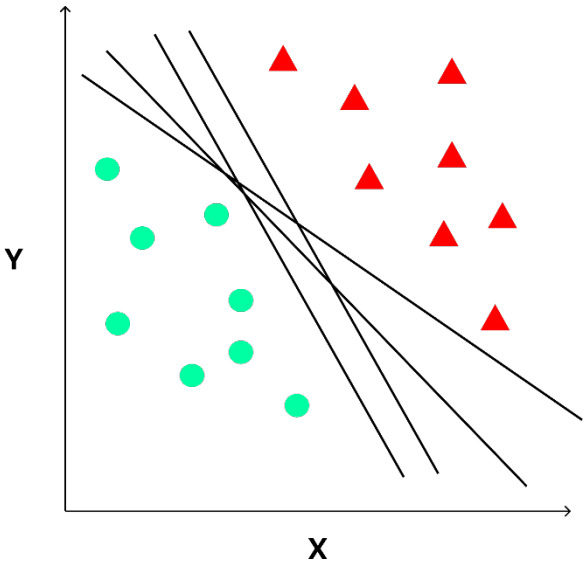


Fig. 1 Different hyperplanes

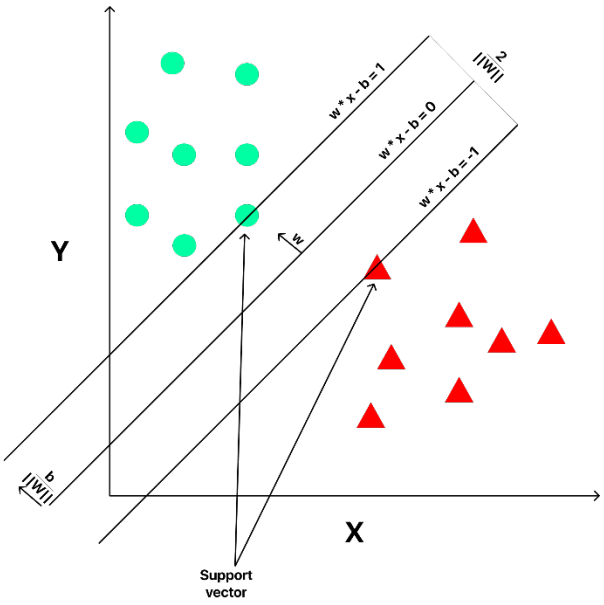


Fig. 2 SVM classifiers

So in the first figure we can see different hyperplanes that divide the classes and as we said we can have more than one hyperplane but we aim for the one with the maximum decision boundary.

In the second figure we can see an application of the SVM classifier. We can see in the figure that we have the representation of the class 1 and 2 in a 2D dimensional space using green points for one class and red triangles for the second class.

The hyperplane in the figure 2 is given as we can see by the formula $w^T x + b = 0$ and works as a decision boundary for us.

Going back to the first figure as we said we have to choose one of the hyperplanes, but the questions is which is the right one to choose?

We have to take into consideration that every single hyperplane can produce different results and the performance can be affected so we should be pretty careful on this decision.

Before talking more about the hyperplanes, we have to introduce other 2 concepts that will help us in choosing the right hyperplanes.

The two concepts are called: **geometric margin** and. **functional margin.**

Functional margin:

The functional margin allows us to test and see if a point is right classified or not and to do so you just have to calculate the margin of the hyperplane and compare the classifier's prediction $w^T x + b$ with the actual class γ_i since we know that our training dataset has a labeled class for each point in our dataset.

The formula for the functional margin is the following:

$$\gamma^i = yi(w^T x + b)$$

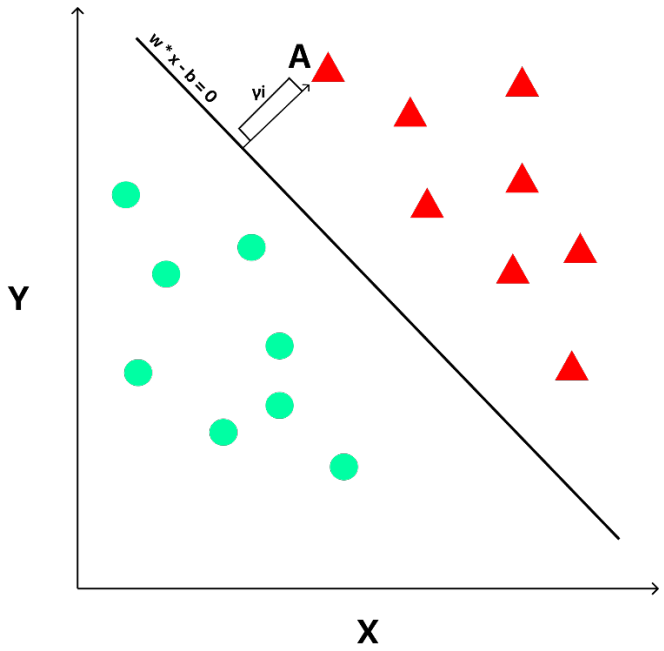
When we get a $\gamma^i = 1$ we need a large number for the $w^T x + b$ just so we can have a large function margin.

If we have $\gamma^i = -1$ than we need a negative large number for the $w^T x + b$ just so we can have a large function margin. Check the picture before (Fig. 2).

Best way to maximize the **FUNCTIONAL MARGIN** is to scale the **W** and the **B**.

Geometric margin:

The geometric margin is the distance between a data point and the decision boundary. How can we calculate it? Well using the following formula, but first let us check the image below and explain a little bit what is going on.



So, we have our point A and our decision boundary, our margin γ_i defines the length from our data point to the decision boundary while $\frac{w}{||w||}$ stands for the shortest path between the point A and the decision boundary.

So, know we have the formula:

$$A - \gamma_i * (\frac{w}{|w|})$$

Now that we defined the functional margin and the geometric margin, we can go back to the problem of choosing the hyperplane. The problem is related to the two margins previously described.

So, the idea of an SVM is to find a hyperplane which maximize the geometric margin and so as a result we have the hyperplane that is considered the best one because it separates the two classes in the best way possible.

We can say that the main goal of the Support Vector Machine is to solve the optimization problem:

$$max_{\gamma,w,b}\gamma$$

$$y^{(i)}\left(w^T x^{(i)} + b\right) \geq \gamma \qquad i = 1 \ldots \ldots m$$

$$||w|| = 1$$

We use or better we need ||w|| = 1 to be sure that the functional margin is equal to the geometric one and so the problem can be written now as follows :

$$max_{\gamma,w,b} \ \frac{\gamma\ hat}{||w||}$$

$$y^{(i)}\left(w^T x^{(i)} + b\right) \geq \gamma\ hat \qquad i = 1 \ldots \ldots m$$

If we put $\gamma\ hat$= 1 than me have a minimization problem.

The solution of all this is an optimal margin classifier. The advantage of all this is that SVM can take into consideration only the support vectors that are the points with the minimum distance between them and the decision boundary and thanks to this we consider just these points in order to determine the optimum decision boundary.

Now that we finished talking about problems linearly separable, we can take a look at the problems that are not linearly separable.

To deal with such problems we can think of the so-called **kernel trick** that allows us to learn a possible separating hyperplane in a new space.

Kernel trick is like a function that applies the mapping of a feature vector to another one.

So now our algorithm SVM instead of learning from a vector x learns from $\boldsymbol{\phi(x)}$.

The formula for the kernel function that allows us to do the inner product between the feature mappings is the following:

$$K(x,z) = \phi(x)^T \phi(z)$$

So instead of doing the inner products between the input vectors (x,z) as we have seen in the SVM we can replace this inner product with the kernel function in order to be able to learn in a different feature space.

We have to be careful though because the kernel function has to satisfy the following two rules in order the consider the kernel a valid one:

$$1.\ K(x,z) = \phi(x)^T \phi(z) \quad \forall\ x,z\ \in S$$

The second rule regards the positivity of the kernel.

So we say that a symmetric function $K : R^n \times R^n \rightarrow R$ is said to be positive defined kernel if:

$$2.\ \sum_{i=1}^{n} \sum_{i=1}^{n} c_i c_j K(x_i x_j) \geq 0 \qquad c_i c_j\ \in R$$

Angular kernels:

One of the requests of this assignment is to create a new kernel that uses just the angular information. We have this request because for now our kernels: the linear, the polynomial and the Gaussian RBF were affected by the length of each vector. Now if we want to eliminate this length and care just about the direction of each vector, we have to normalize the vectors in such a way that in the end we have $||w||_2 = 1$.

So, the kernel that is not affected by the length of the vectors looks like:

$$\phi : R^m \rightarrow R^m$$

$$\phi(x) = \frac{x}{||x||}$$

$$K(x_i, x_j) = \phi(x_i)^T \phi(x_j) = \frac{x_i}{||x_i||} \frac{x_j}{||x_j||} = \cos(x_i, x_j)$$

The previous kernel only takes into consideration just the cosine of the angle created between the two vectors $x_i x_j$.

Now that we show that is possible to just consider the angular information we have to talk about the positivity of the kernel as we said earlier.

We have to prove this positivity and bellow you can find the proof.

$$\sum_{i=1}^{n}\sum_{j=1}^{n} c_i c_j \, K\left(x_i x_j\right) \geq 0 \quad c_i c_j \in R$$

$$= \sum_{i=1}^{n}\sum_{j=1}^{n} c_i c_j \, (\phi(x_i), \phi\left(x_j\right))$$

$$= \sum_{i=1}^{n}\sum_{j=1}^{n} c_i c_j \sum_{o=1}^{m} \frac{x_{io}}{||x_{io}||} \frac{x_{jo}}{||x_{jo}||} \geq 0 \quad c_i c_j \in R$$

$$= \sum_{i=1}^{n}\sum_{j=1}^{n}\sum_{o=1}^{m} c_i \frac{x_{io}}{||x_{io}||} c_j \frac{x_{jo}}{||x_{jo}||}$$

$$= \sum_{o=1}^{n} \left(\sum_{i=1}^{n} c_i \frac{x_{io}}{||x_{io}||}\right)^2 \geq 0$$

Approach that we as a team followed for the first part of the assignment + code explained:

So starting from the beginning we took in input our dataset and calculate the TF-IDF and then we split the data into X and Y which contain the features and the label columns. Once we did that we use the SVM classifier with the three kernels: Linear / Polynomial and RBF and use the 10-way cross validation and checked the accuracy, once we did that we normalized the data so we had the data affected just by the direction and not by the length as we explained before and we launched again the SVM classifier with the three kernels and as expected we had a big difference at least for the linear kernel.

Full explanation + code:

1st part reading the data :

```
data = pd.read_csv("spambase.data", "," ,index_col = False, names = cols)
data = data.drop(columns=["capital_run_length_total","capital_run_length_longest
```

2nd part calculating the TF-IDF:

```
def tfidf(data):
    ndoc = data.shape[0]
    idf = np.log10(ndoc/(data != 0).sum(0.))
    return data/100.0*idf
```

Calculating the TF-IDF we get how relevant certain words are in the given email but not only, because we get in input text, but we cannot work with it because our algorithm has to deal with numbers. So, we have to transform our text into numbers.

This process is fundamental for the machine learning and for the data analysis. TF-IDF gives us a way to associate each word in a document (in our case the e-mail) with a number that represents how relevant each word is in that document (in our case the e-mail).

3rd part dividing the data into two parts "X" and "Y":

```
Y = data["spam"]
data= data.drop(columns=["spam"])
X = tfidf(data)
```

So our Y represents the last column of our dataset and contains the SPAM or HAM label for each e-mail in the dataset.

4th part is deciding how much data goes for the training and how much for the test : 30% for the test and 70% for the train.

```
X_train, X_test, Y_train, Y_test = train_test_split(X,Y, test_size = 0.3, random_state = 1)
```

5th part contains the SVM classifiers + 10-way cross validation + confusion matrix:

5.1 Linear kernel + code:

```
clf0 = svm.SVC(kernel="linear", C = 1.0)
scores_clf0 = cross_val_score(clf0, X, Y, cv = 10)
print("Cross Validation: %0.2f accuracy with a standard deviation of %0.2f" % (s
clf0.fit(X_train,Y_train)
print("Confusion Matrix on Test Dataset")
plot_confusion_matrix(clf0, X_test, Y_test)
plt.show()
```

5.2 Polynomial kernel + code:

```
clf1 = svm.SVC(kernel="poly",degree=2, C = 1.0)
scores_clf1 = cross_val_score(clf1,X,Y,cv=10)
print("Cross Validation: %0.2f accuracy with a standard deviation of %0.2f" % (s
print("Confusion Matrix on Test Dataset")
clf1.fit(X_train,Y_train)
plot_confusion_matrix(clf1, X_test, Y_test)
plt.show()
```

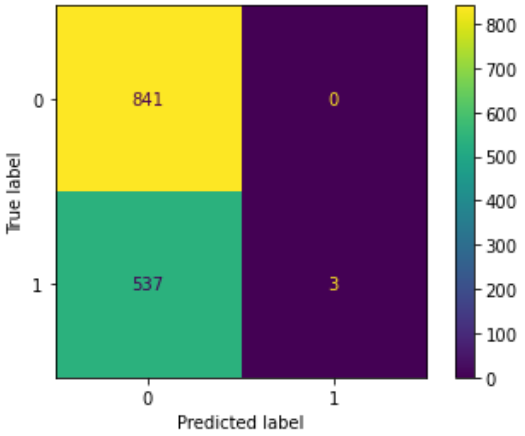
5.3 RBF kernel + code:

```
clf2 = svm.SVC(kernel="rbf", gamma='scale',C = 1)
scores_clf2 = cross_val_score(clf2,X,Y,cv=10)
print("Cross Validation: %0.2f accuracy with a standard deviation of %0.2f" % (s
print("Confusion Matrix on Test Dataset")
clf2.fit(X_train,Y_train)
plot_confusion_matrix(clf2, X_test, Y_test)
plt.show()
```

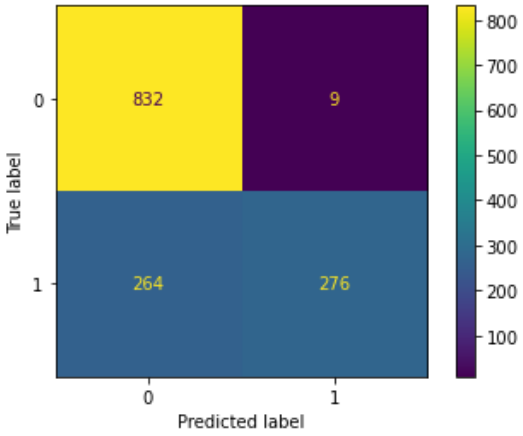
As we can see for each kernel, we launched a confusion matrix to check the differences between the kernels.

The confusion matrix allows us to see the performance of the classifiers on a set of test data for which the true values are known. As we can see from the plots below, we have the Predicted Label and True Label. The confusion matrix allows us to see the right labeled objects the wrong labeled objects, the false positive and the false negative cases.

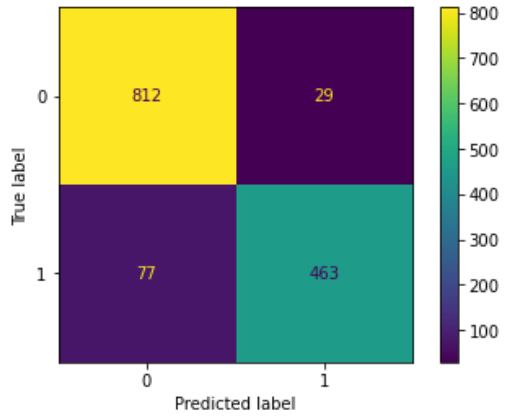
Bellow we can see the differences:



Linear Kernel



Polynomial Kernel



RBF Kernel

Once we get the accuracy for this kernels, we normalized the data so we do not have the data that is affected by length and by the direction but only by the direction.

Code:

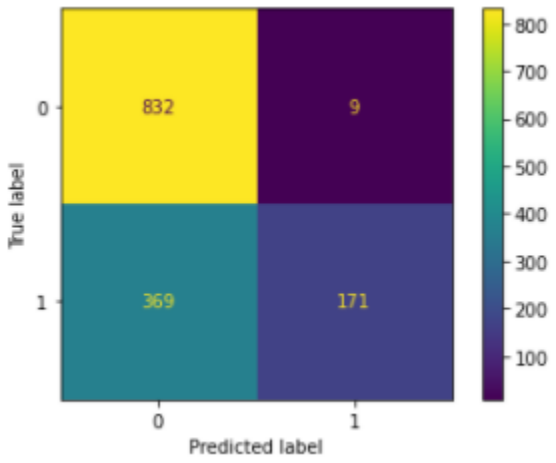
```
from sklearn import preprocessing
normalized_X = preprocessing.normalize(X)
newdata = np.where(normalized_X > 0.0, X / normalized_X, 0)
```

Once we took the new data new split it again in train and test: 30% for the test and 70% for the training.

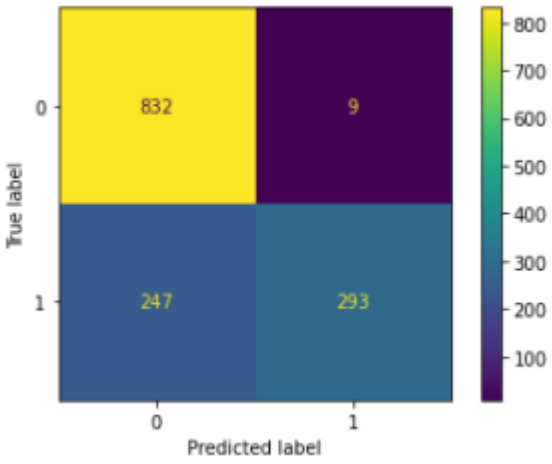
```
X2_train, X2_test, Y2_train, Y2_test = train_test_split(newdata, Y, test_size = 0.3, random_state = 1)
```

The next step was to run the SVM with the three kernels again on the new dataset, the normalized dataset with the same code and the same confusion matrices.

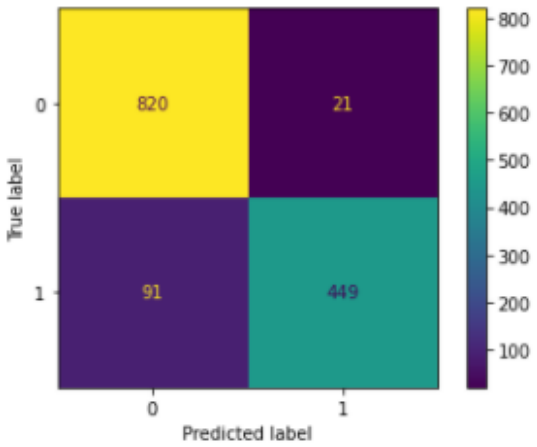
Bellow you can see the result:



Linear Kernel



Polynomial Kernel



RBF Kernel

After all these steps we can confront the data before and after the normalization.

So, before the normalization of the data we got the following results:

Kernel names	Min score	Mean score	Max score	Deviance
Linear	0.60434	0.60791	0.61739	0.00
Polynomial	0.74782	0.80787	0.85652	0.03
RBF	0.84782	0.91523	0.95869	0.03

The following data consider the angular information, so we take into consideration just the direction of the vectors not the length. In our assignment we had to build a new kernel but instead of that we decided to normalize the data in the dataset because the new kernels are nothing but the original kernels with vectors with length = 1. So we normalized the data and then we applied the SVM with the three kernels again.

So, after the normalization of the data we got the following results:

Kernel names	Min score	Mean score	Max score	Deviance
Linear	0.68695	0.74767	0.77826	0.03
Polynomial	0.75271	0.81918	0.88260	0.03
RBF	0.83478	0.90393	0.93260	0.03

Checking both tables, we can see an improvement for the linear kernel using the angular information. The values for the polynomial and RBF kernel are almost not

changed before and after the normalization so this denotes that the length does not affect them.

Theory explanation behind Naive Bayes and behind the second point of the assignment:

Naive Bayes:

The Naive Bayes classifier is a probabilistic ML model that we use for classification tasks. The classifier is based on the Bayes theorem:

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}$$

When we use this theory, we find the probability of A happening given that B has occurred, B can be called **evidence**. The main idea here is that we assume that the predictors and the features are independent.

The **Naive Bayes** classifier assigns the label that maximize the probability of P(A|B) so we have the following:

$$\mathrm{F(A)} \;=\; \arg\max P(A|B) = \frac{P(B|A)P(A)}{P(B)}$$

Naive assumption:

The naive assumption of the Bayes' theorem means independence among the features.

Independence means:

P(A, B) = P(A)*P(B)

Approach that we as a team followed for the second part of the assignment + code explained:

For the second part of the assignment, we decided to implement the algorithms ourselves and then check the results using the library.

1st step we calculated the Gaussian Distribution:

```
def __gaussianDistribution(self,x,k,mean,variance):
    temp = np.zeros(shape=(len(x)))
    for i in range(len(x)):
        temp[i] =  (1/math.sqrt(2*math.pi*variance[i])) * math.exp( (-1/(2*variance[i]))*((x[i]-mea
    return np.prod(temp)
```

In return the function gives us the product of the array temp.

2nd step we have the fit() method that take in input x and y that are actually x_train and y_train and we train the model. The fit method adjusts the weights according to data values so it can achieve better accuracy. After training, the model can be used for predictions.

```
def fit(self,x,y):
    if len(x) != len(y) :
        print("Error: X and Y have different size")
    else:

        self.__classes = np.unique(y)


        u = x.copy()
        u['class'] = np.array(y.values)


        dataset_class=[]
        prob_class = []
        for c in self.__classes:
            dataset_class.append(np.array(u[u['class']==c]))
            prob_class.append(len(dataset_class[c])/len(x))
        self.__prob = prob_class
        self.__dataset_class = dataset_class



        mean = np.zeros(shape=(2,54))
        variance = np.zeros(shape=(2,54))

        for c in self.__classes:
            for i in range(0,len(x.columns)):
                mean[c][i] = st.mean(dataset_class[c][:,i])
                variance[c][i] = (1/(len(dataset_class[c][:,i])-1))* sum((dataset_class[c][:,i]-mea
                if variance[c][i] == 0:
                    variance[c][i] = 1
        self.__mean = mean
        self.__variance = variance
        self.__trained = True
```

3rd step is the predict where we can check how the algorithm performs, by that I mean that we can see how many good predictions it did overall, more good predictions higher the accuracy.

```
    def predict(self,x):
        if self.__trained == True:
            solutions = np.zeros(shape=(len(x),len(self.__classes)))
            x = np.array(x)
            for c in self.__classes:
                 for i in range(len(x)):
                     solutions[i][c] = self.__predict_util(x[i],c)
            labels = np.zeros(shape=(len(solutions)))
            for i in range(len(solutions)):
                labels[i] = np.argmax(solutions[i])
            return labels
        else:
            print("The model is not trained\n")
            return []
```

4th step is the score where we calculate the accuracy:

```
def score(self, X,Y):
    s = self.predict(X)
    comparison = s == Y
    return (len(comparison[comparison== True])/len(comparison))
```

5th step : We do the cross validation score. **Cross validation** is a method used to measure the skill of machine learning models on unseen data.

We had to use the k-Fold Cross Validation. In this case we have just a parameter called **k** that refers to the number of groups that our data has be split into. The idea is to use a limited sample in order to estimate how the model is expected to perform in general when used to make predictions on data not used during the training of the model.

The **k** parameter has to be chosen carefully because if we do not choose it wisely than we can have a miss-representative idea of the model and by that, I mean that we might have a score with a high variance or high bias.

For our specific problem we chose k=10 and we got the following accuracy:

```
nbc = GaussianNaiveBayes()
nbc.fit(X_train,Y_train)
labels = nbc.predict(X_test)
comparison = labels == Y_test
scores = cross_val_score(nbc,X,Y,cv=10)
print("Accurency Test=",(len(comparison[comparison== True])/len(comparison)))
print("Accurency Cross Validation=",np.mean(scores))
```

```
Accurency Test= 0.8110065170166546
Accurency Cross Validation= 0.8108752239932094
```

After we implemented the steps manually, we decided to check the results by using the library skilearn using the following code.

```
#Test with scikit-learn
from sklearn.naive_bayes import GaussianNB
gnb = GaussianNB()
y_pred = gnb.fit(X_train,Y_train).predict(X_test)
comparison = y_pred == Y_test
scores = cross_val_score(gnb,X,Y,cv=10)
print("Accurency Test=",(len(comparison[comparison== True])/len(comparison)))
print("Accurency Cross Validation=",np.mean(scores))
```

```
Accurency Test= 0.8066618392469225
Accurency Cross Validation= 0.8150348957842122
```

As we can see from the two pictures, we managed to have the same accuracy.

Bellow we can see the full results from the Naive Bayes in compare with SVM.

Kernel names	Min score	Mean score	Max score	Deviance
Linear	0.68695	0.74767	0.77826	0.03
Polynomial	0.75271	0.81918	0.88260	0.03
RBF	0.83478	0.90393	0.93260	0.03
Naive Bayes	0.46956	0.81087	0.97396	0.03

As we can see from table above the Naive Bayes does not perform better than the SVM in most of the cases.

Theory explanation behind KNN and behind the third point of the assignment:

KNN classifier:

KNN is one of the simplest techniques used in ML and is preferred by many people because is it easy to use and it takes a short time to calculate.

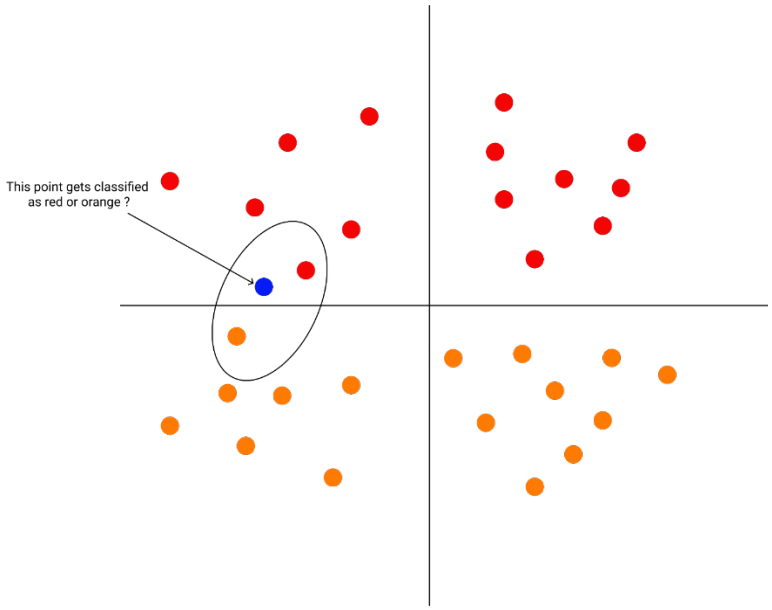
The main idea of this technique is that it classifies the data points based on the points that are most similar to it.

So based on the points that are already classified it takes another point and checks the similarities with the other once and the class with the most similar features, and the output class is assigned to the new point.

Let us see the pros and cons about this algorithm:

So as we said before the pros with this algorithm are that is easy to use, quick to calculate and does not make assumptions about the data but on the other hand the cons are that the accuracy depends on the quality of the data, you must find an optimal k value and it does not do well when you have a data point in a boundary where the point could be mapped in 2 different classes.

Let us see an example bellow:

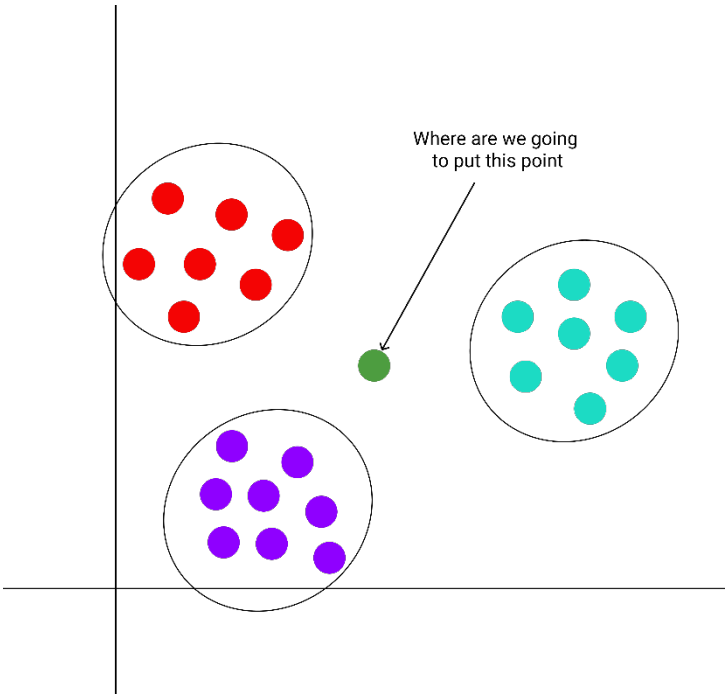


As we know and as we studied in class and from the internet the KNN algorithm is **non-parametric** and is also **a lazy algorithm**.

A non-parametric algorithm means as we said earlier that it makes no assumptions, so the model is made up entirely from the given data.

Lazy algorithm means that the algorithm makes no generalizations. So that means that we have a little training involved when using the method.

Let us have another example:



This is another example where we have three classes, and we have a point that needs to be classified. At this point this point that needs to be classified is compared to its nearest points and classified based on which points are the closest and most similar to.

Ok know that we get the idea of how it works we should say how to calculate this similarity, so we know how to choose the class for the new points.

The first step is to transform data points into feature vectors. Once we did that the algorithm finds the distances between the points. It finds the distance by calculating the Euclidean distance using the following formula:

$$d(p,q) = \sqrt{\sum_{i=1}^{n}(q_i - p_i)^2}$$

Where:

P and Q are two points in the Euclidean n-space

q_i and p_i are Euclidean vectors starting from the origin of the space

n is the n-space

KNN uses this formula to calculate the distance between each data point and the test data, then the algorithm finds the probability of these points being similar to the test data and classifies them based on which points share the highest probabilities.

Now that we got the idea of what KNN is and how it works we can explain how we implemented the code.

1st step : We calculated the Euclidean Distance:

```
def __euclideanDistance(self,v1,v2):
    #distanza euclidea tra due istanze cioè tra due vettori
    return np.sum(np.sqrt((v1-v2)**2))
```

2nd step contains the fit() method that allows us to "train" the algorithm.

```
def fit(self,X,Y):
    self.__X_train = X
    self.__Y_train = Y
```

3rd step contains the predict () method that allows us to predict the class for the points.

```
def __predictUtil(self,v):
    k = 5
    #trainTemp copia del dataset
    trainTemp = self.__X_train.copy()
    distances= np.empty(shape=(len(trainTemp)))
    #è un array che contiene le distanze
    trainTemp = np.array(trainTemp)
    #calcolo le distanza tra il vettore v in input e tutte le istanze del dataset
    for i in range(len(trainTemp)):
        d = self.__euclideanDistance(v,trainTemp[i])
        distances[i]= d

    trainTemp = pd.DataFrame(trainTemp)
    trainTemp['dist'] = distances
    trainTemp['class'] = self.__Y_train.values
    trainTemp=trainTemp.sort_values(by=['dist'])
    x = trainTemp[1:k+1]['class'].values

    #se nei vicini la classe dei vicini controllo se sono spam o no
    if len(x[x==0]) >= len(x[x==1]):
        return 0
    else:
        return 1
```

```
def predict(self,x):
    x = np.array(x)
    solution = np.empty(shape=(len(x)))
    for i in range(len(x)):
        solution[i] = self.__predictUtil(x[i])
    return solution
```

4th step was to calculate the score which represents the accuracy of the algorithm.

```
def score(self, X,Y):
    s = self.predict(X)
    comparison = s == Y
    return (len(comparison[comparison== True])/len(comparison))
```

5th step is to calculate the cross validation which allows us to see the performance of the algorithms. We did it by writing the following code:

```
scores = cross_val_score(knn,X,Y,cv=10)
print(np.mean(scores))
```

```
0.8770088654154484
```

6th step: For this step we used the sklilern package and used the KNN algorithm so we could see the actual result and to compare it with our result:

```
from sklearn import neighbors
kNN = neighbors.KNeighborsClassifier(n_neighbors=5)
kNN.fit(X_train,Y_train)
sol = kNN.predict(X_test)
comparison = sol == Y_test.values
print("Accurency=",(len(comparison[comparison== True])/len(comparison))*100,"%")
scores = cross_val_score(knn,X,Y,cv=10)
print(np.mean(scores))
```

```
Accurency= 92.39681390296887 %
0.8770088654154484
```

As we can see above the result from our implementation and from the sklearn package KNN classifier are identical.

Comparison:

The CONS for the SVM is the time that need for the learning phase. SVM takes more time in the learning phase because it considers only support vectors and not all the data points and so it needs more iterations to be able to find the right parameters in order to find the best hyperplane.

We cannot say the same thing about the Naive Bayes because the classifier does not require so much data and is very simple since it is done with a single iteration.

A PRO for the SVM is that it is capable of dealing with the **non-separable** data using the **kernel trick** instead the Naive Bayes is not able to do it.

If we take into consideration the KNN and the Naive Bayes we can say that the Naive Bayes is much faster than KNN because of the KNN real time execution and we also can say again that KNN is a non-parametric algorithm, but the Naive Bayes is parametric.

Another comparison can be done between the KNN and SVM and we can say that if we have not too much data to train the algorithm than at this point SVM will perform better than the KNN but if we have enough data so that the number of data is bigger than the number of features than we can say that KNN will perform better than SVM.

We can also say that the KNN can adapt more closely to nonlinear boundaries as the it has more data to work with. It has a higher variance in compare with the SVM but it is able to produce classification fits that adapt to most of the boundaries.