Spam Filter using Support Vector Machine, Naive Bayes and K-Nearest Neighbors

Ca' Foscari University of Venice
Department of Environmental Sciences, Informatics and Statistics

Artificial Intelligence: Knowledge Representation and Planning [CM0472-1]
Academic Year 2021 - 2022

Students
Zuliani Riccardo 875532
Fortin Luca 858986

Contents

1 Introduction **1**

2 Classification Problem **2**
2.1 Discriminative and Generative Models 3
2.1.1 Discriminative Models 3
2.1.2 Generative Models 3

3 Spambase Dataset **4**

4 Cross Validation Score **6**

5 Support Vector Machine (SVM) **8**
5.1 Theoretical Definition 8
5.2 Angular Kernel 12
5.3 Implementation 13
5.4 Results 16
5.4.1 Using Original Dataset 16
5.4.2 Using Normalized Dataset 17

6 Naive Bayes **19**
6.1 Theoretical Definition 19
6.2 Implementation 20
6.3 Results 22

7 K-Nearest Neighbors **23**
7.1 Theoretical Definition 23
7.2 Implementation 24
7.3 Results 26

8 Summary & Comparison **27**

9 Conclusion **28**

List of Figures

3.1 Correlation matrix between features 4

4.1 KFold Cross Validation 7

5.1 SVM Classifier 9
5.2 Possible hyperplanes 9
5.3 Geometric Margin Intuition 10
5.4 CM Linear Kernel - Original Dataset 16
5.5 CM Poly Kernel - Original Dataset 16
5.6 CM RBF Kernel - Original Dataset 16
5.7 CM Linear Kernel - Normalized Dataset 17
5.8 CM Poly Kernel - Normalized Dataset 17
5.9 CM RBF Kernel - Normalized Dataset 17

7.1 K-Nearest Neighbours Example 24
7.2 CM K-Nearest Neighbours 26

List of Tables

5.1 SVM - Cross Validation results using original dataset 17
5.2 SVM - Cross Validation results using normalized dataset 17

6.1 SVM - NB Cross Validation Results 22

7.1 SVM - KNN Cross Validation Results 26

8.1 Summary of Cross Validation Results 27

Chapter 1

Introduction

Write a spam filter using **discriminative** and **generative** classifiers. Use the Spambase dataset[1] 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, modeling each feature with a Gaussian distribution, resulting in the following model:

$$p(y = k) = \alpha_k$$

$$p(y = k) = \prod_{i=1}^{D} \left[(2\pi\sigma_{ki}^2)^{-1/2} exp\left\{ -\frac{1}{2\sigma_{ki}^2}(x_i - \mu_{ki})^2 \right\} \right]$$

   Where α_k is the frequency of class k, and μ_{ki}, σ_{ki}^2 are the means and variances of feature i given that the data is in class k.

3. Perform k-NN classification with k = 5

Provide the code, the models on the training set, ad respective performances in 10 way cross validation. Explain the differences between the two models.

[1] https://archive.ics.uci.edu/ml/datasets/spambase

Chapter 2

Classification Problem

In *machine learning* one of the most common and usual problem regard the classification of a specific entity in a given context. Normally all the algorithm that solve this problem start by learning on the context looking at the input data given by the user, this part is called the **learning phase** because it learn and get knowledge by using this data and its features that describe all observations. The next step is try to predict the **label** of unseen data. Normally the label in classification problem is marked as 0 or 1, so in our give problem 0 represent *ham* and 1 *spam*.

Now mathematically speaking we first rephrase the concept of learning features or more technically **predictors** since are used to predict the actual label of the observations. We define all the predictors as the set:

$$X = \{X_1 ... X_N\} \in \mathcal{X}$$

Moreover the previous cited label is more precisely called **target features** since like the others features it describe the record and indeed is the target of our discussion because we want to predict it.

The classification algorithm consists in a learning function $\phi(x) : \mathcal{X} \to \mathcal{Y}$ that is generated by the give training set. Furthermore setting our target feature by $\mathcal{Y} = \{0, 1\}$ we can say that:

$$\phi(x) = g(f(x))$$

In which:

- $f(x)$ is the function that is able to split our observation in the two classes, 0 for the ham email and 1 for the spam email
- $g(z)$ where $z = f(x)$, is the function that returns the correct *label* of a given new observation x passed to the function $f(x)$. Thus $g(x)$ can be seen like so:

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

2.1 Discriminative and Generative Models

In general there are three main type of learning algorithm:

- **Supervised Learning:** is a type of learning in which the algorithm learn to classify correctly unseen data from previous given labelled observations.
- **Unsupervised Learning:** is completely opposite to Supervised Learning, in fact the algorithm learn to predict data from a previous given unlabeled observations
- **Reinforced Learning:** it is not based on supervised learning and unsupervised learning. But there is the idea of an agent that tries to manipulate the environment, by making observations. With them it gets a reward or a punishment, thus it can learn the way to be always rewarded.

Our task since is a classification problem it belongs to the class of Supervised Learning. Moreover we can distinguish two type of supervised learning based on what they focus on to predict unseen data, the two methods are Discriminative models and Generative models.

In simple words, a **discriminative model** makes predictions on the unseen data based on *conditional probability* and can be used either for classification or regression problem statements. On the contrary, a **generative model** focuses on the *distribution of a dataset* to return a probability for a given example. [4]

2.1.1 Discriminative Models

Discriminative model are also called conditional models since they learn boundaries between classes or labels in a dataset. These models separates classes instead of modelling the conditional probability, moreover they do not make any assumption on the distribution of data points. If we have outliers discriminative models works much better rather than generative model but they suffer form missclassification problem. [4] So mathematically speaking, discriminative models directly assume functional form $P(Y|X)$ and estimate parameters of $P(Y|X)$ directly from training data. [1]

2.1.2 Generative Models

On the other hand generative models are considered as a class of statistical model that can generate new data. These models focus on the distribution of individual classes in a dataset and the learning algorithms tend to model the underlying patterns or distribution. They uses probability estimates and likelihood to model data points. [4] Thus generative models estimate the prior $P(Y)$ and likelihood $P(X|Y)$ from training data and use Bayes rule to calculate the posterior probability $P(Y|X)$. [1]

Chapter 3

Spambase Dataset

SpamBase is a classification dataset containing 4601 emails sent to HP (Hewlett-Packard) during some period of time. The SpamBase contains 57 numeric features for each email and a binary label, 1 for spam, 0 for email. Most of the attributes indicate whether a particular word or character was frequently occuring in the e-mail. The run-length attributes (55-57) measure the length of sequences of consecutive capital letters, but in our analysis we do not consider them. This is a typical binary classification problem with the added need for clever feature selection, as many of the features provided in the dataset might be useless (Hopkins et al., 1998). [7]

The collection of spam e-mails came from HP's postmaster and individuals who had filed spam. The collection of non-spam e-mails came from filed work and personal e-mails, and hence the word 'george' and the area code '650' are indicators of non-spam. These are useful when constructing a personalized spam filter. One would either have to blind such non-spam indicators or get a very wide collection of non-spam to generate a general purpose spam filter. [2]

To make an initial analysis of the feature, we try to see if there is some correlation between each couple of features by plotting the correlation matrix.

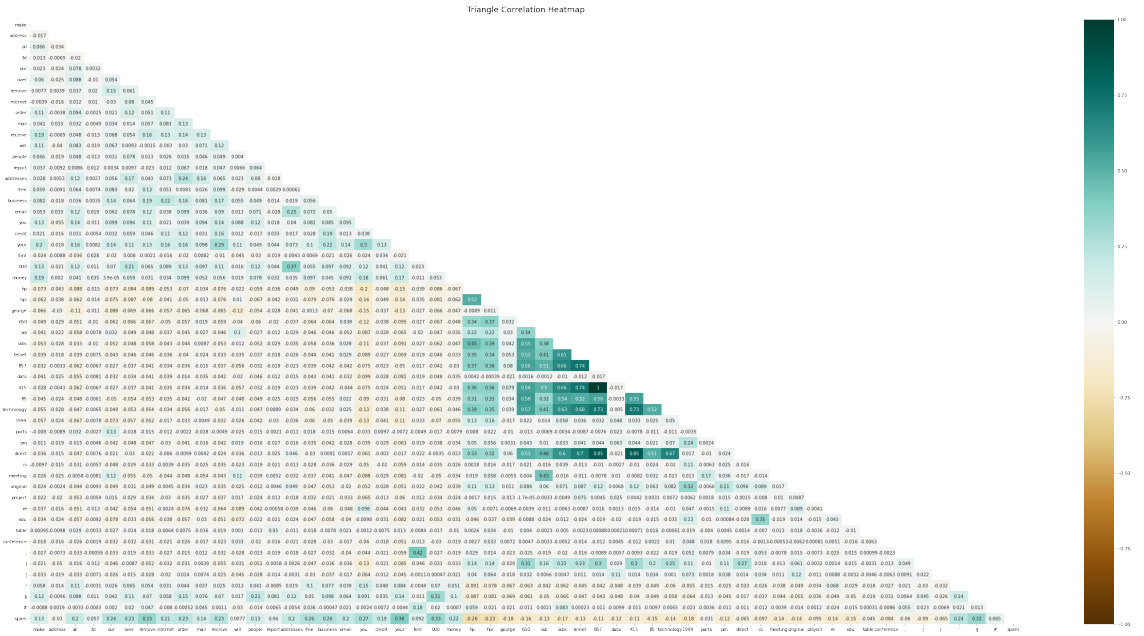

Figure 3.1: Correlation matrix between features

We can deduce that at most all the feature couple are not correlated to each other, except to some unusual once, like feature *857* and *451* that have a correlation of 1, which means that these two are very strongly correlated.

Chapter 4

Cross Validation Score

Before discussing on the three models, we analyse the behaviour of the Cross Validation Score. As cited in the introduction we compute the performance of SVM, Naive Bayes and K-Nearest Neighbor using a 10-way cross validation, but *"what exactly is this 10-way cross validation?"*

Cross Validation is an iterative technique where we evaluate the given model for a finite number of time by splitting our original dataset in a train set to train our model and a test set to evaluate it. This technique avoids the usual **overfitting**, which occurs when a statistical model or machine learning algorithm captures the noise of the data. Overfitting a model means that it gets a very good accuracy in the training set, but on the other hand it gets poore performance in the test set.

There are lots of types of Cross Validation methods provided by Sklearn. In our analysis we focus in the vanilla version, which is stated as follow:

```
1 cross_val_score(model(), X, y, cv = 10, n_jobs = -1)
```

- **model():** is the actual estimator
- **X:** is the data to fit
- **y:** the target variable to try to predict
- **n_jobs = -1:** indicate to allow the usage of all processors available in the running machine

We have take in part **cv = 10** because needs more clarification. CV determines the cross-validation splitting strategy and when the cv argument is an integer, cross_val_score uses the KFold or StratifiedKFold strategies by default. The latter being used if the estimator derives from ClassifierMixin, which is Mixin class for all classifiers in scikit-learn. [8]

Thus since both SVM, Naive Bayes and K-Nearest Neighbors implement them selves the last cited class, we have to provide an equal comparison. In order to do so in the definition of the implemented version of Naive Bayes and KNN we add to the already present *BaseEstimator* the *ClassifierMixin*. Continuing the two initialization classes are defined as follow:

```
class NaiveBayes(BaseEstimator, ClassifierMixin)

class KNearesNeighbour(BaseEstimator, ClassifierMixin)
```

Defined that we now are sure that all the made evaluation with cross_val_score adopt the StratifiedKFold strategy with k = 10. To introduce StratifiedKFold we have to discuss about **KFold** since they are very similar.

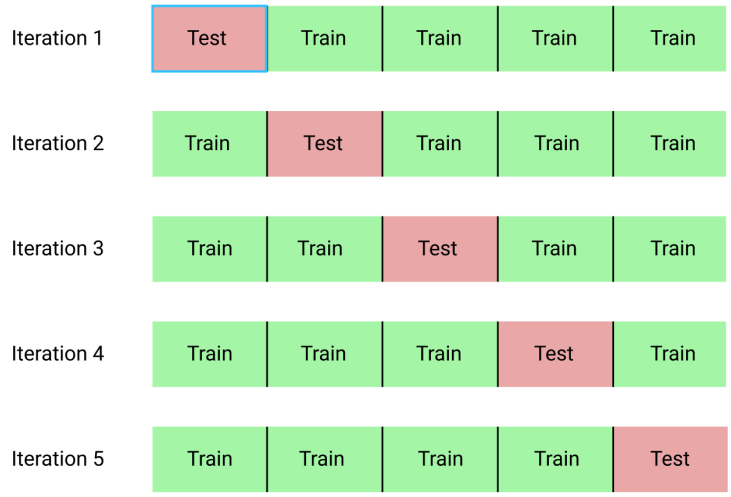

Figure 4.1: KFold Cross Validation

The parameter K refers to the number of groups that a given dataset is to be split into. In Figure 4.1 the K = 5 and the computation is:

1. Take a dataset partition as a test set
2. Take the others partitions as a single training set
3. Fit the model on the training set and fit the model on the test set
4. Retain the evaluation score and discard the model

The **StratifiedKFold** strategy differs only on the the splitting of data into folds, that may be governed by criteria such as ensuring that each fold has the same proportion of observations with a given categorical value, such as the class outcome value. [10]

Chapter 5

Support Vector Machine (SVM)

In this chapter we discuss about the Support Vector Machine one of the most popular machine learning algorithm to solve classification problem. It is part of the group of discriminative classifier since it learn how to classify correctly unseen observations by previous labeled data $y = \{0, 1\}$ which in our case we substitute with $y = \{-1, 1\}$. Here we analyse the theoretical behaviour, the code implementation and the obtained results.

5.1 Theoretical Definition

The abstract idea of Support Vector Machine is to generate and draw a decision boundary between point or in other words observations of two distinct classes. In a space with 2 features this decision boundary is a line, then with the increasing of features the decision boundary becomes a plane and finally a hyperplane. We can have lots of different decision boundary but our goal is to try to find the best one that maximise the distance to the nearest point between the two classes. A nice way to rephrase the SVM problem is to plot a road with the straight line in the middle that separate in two lanes and we have to find the road with maximum width of the lanes on both sides.

Now we focus more deeply in the mathematical aspect of Support Vector Machine by first introducing the formal definition of the SVM classifier:

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

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

In which:

- $g(w^T x + b)$ represent the actual **decision boundary** or **separating hyperplane** for the two classes
- $g(z)$ represent the decision function which returns the predicted class for the new observation x

Our classifier will return the predicted label 1, so spam, in case we get $h_{w,b}(x) \geq 0$ or in other words if $g(w^T x + b) \geq 0$ and -1 so ham in case we get $g(w^T x + b) < 0$. Moreover we mention the fact that larger is the value of $g(w^T x + b)$ the higher is the

confidence that the new observation is labeled with 1 and on the other hand lower is the value of $g(w^T x + b)$ the higher is the probability that the new observed data is labeled with -1.

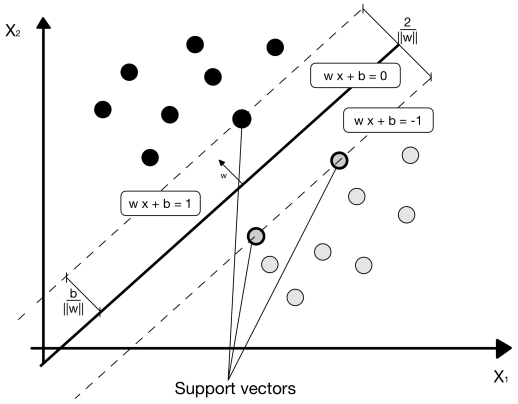

Figure 5.1: SVM Classifier

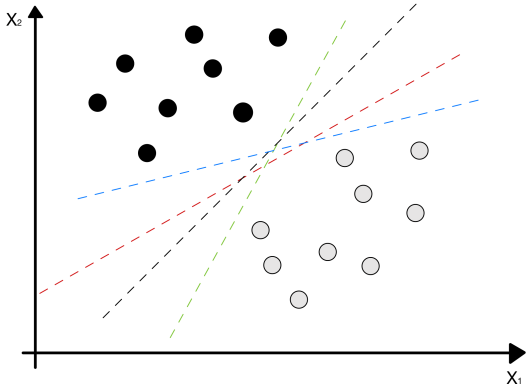

Figure 5.2: Possible hyperplanes

In Figure 5.1 we can see an application of Support Vector Machine Classifier, in a world of two classes labeled with -1 and 1 like our case. The decision boundary draw is indeed as cited previously the straight line that divide the road in two equal width lanes and is given by the equation of $w^T x + b = 0$. On the other hand in Figure 5.2 we see a bunch of possible hyperplanes that actually divide our observations in two separated classes, but we do not want the first decision boundary that separate the observations, so this aspect introduce the question of *"How we can decide which hyperplane is the best for our problem?"* Different hyperplanes can bring with them different performance and result when facing in unseen data. Moreover in the world of SVMs, this "space" between the decision boundary and the data points is called the margin. By maximizing the size of this margin, we can build an optimal classifier. To answer the previous question we have to spend a bit of paragraphs explaining two useful definition of margin, we are talking about **functional margin** and **geometric margin**.

Functional Margin

Functional Margin allow us to verify if a specific given point is classified correctly or not. We can calculate the margin of our hyperplane by comparing the classifier's prediction $w^T x + b$ to the actual class y_i, and it id defined like so:

$$\hat{\gamma}_i = y_i(w^T x_i + b)$$

When $y_i = 1$ we would need a large positive number for $g(w^T x + b)$ in order to have a large functional margin, or in others words in order to be sure on the predicted label. While if $y_i = -1$ we would have large negative number for $g(w^T x + b)$. Furthermore we can combine these two constraints into a single one which say that our classifier make correct prediction for all values where $\hat{\gamma}_i > 0$.

One way to maximize this functional margin is to arbitrarily scale w and b, since classification with $h_{w,b}$ only depends on the sign and not the magnitude of the returned value. However, this provides little value and is a weakness of functional margins, as we have no useful way to maximize this expression.

Geometric Margin

The geometric margin is the distance from a data point to the decision boundary hyperplane. But we actually calculate it? The Figure 5.3 bellow gives an intuitive answer.

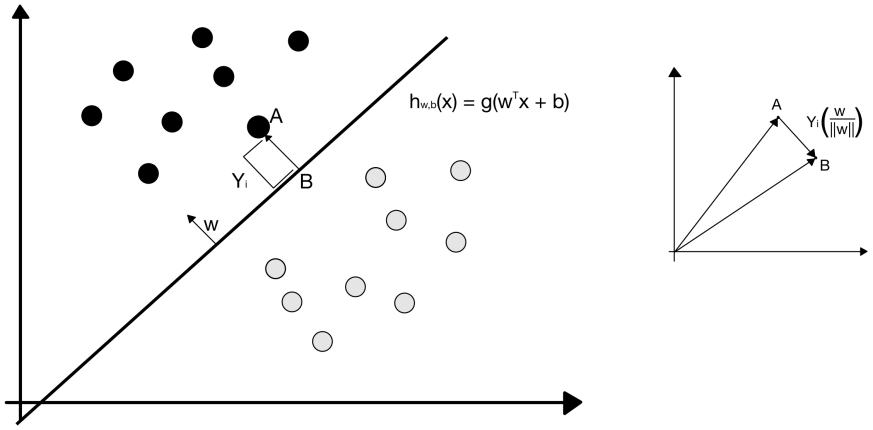

Figure 5.3: Geometric Margin Intuition

Considering the two labeled point A and B, our margin $\hat{\gamma}_i$ defines the scalar length from our given new observation x_i to the hyperplane, while $\frac{w}{||w||}$ represent the unit vector to the decision boundary. The point B can be formulated as:

$$B = A - \gamma_i \left(\frac{w}{||w||} \right)$$

Or in a more simpler way:

$$B = A - margin$$

Moreover since the B point is on the decision boundary, so it must satisfy the equation $g(w^T x_B + b) = 0$. Further we know we can calculate x_B as

$$x_B = x^i - \gamma_i \left(\frac{w}{||w||} \right)$$

And if we substitute x_B in the first equation we obtain:

$$w^T \left(x^i - \gamma_i \left(\frac{w}{||w||} \right) \right) + b = 0$$

And after some simplification we get

$$\gamma_i = \frac{w^T x^i + b}{||w||} = \left(\frac{w}{||w||} \right)^T x^i + \frac{b}{||w||}$$

Finally we have to make a little adjustment to keep the margin to be positive when dealing our classes.

$$\gamma_i = y_i \left(\left(\frac{w}{||w||} \right)^T x_i + \frac{b}{||w||} \right)$$

Notice that the geometric margin differ from the functional margin only for the fact that w and b are scaled by factor $\frac{1}{||w||}$ and if we set $||w|| = 1$ the two margins become

equal. Finally the margin of a hyperplane is formally defined to be the smallest one in the entire set of data points:

$$\gamma = \min_{i=1,...,m} \gamma_i$$

Now we have all what is needed to actually compute hyperplane. So the idea of SVM is to find the best separating hyperplane for the two classes maximizing the geometric margin and it it formulated like so:

$$\begin{aligned} \max_{\gamma,w,b} \quad & \gamma \\ \text{s.t.} \quad & (w^T x_i + b) \geq \gamma \qquad i = 1, ..., m \\ & ||w|| = 1 \end{aligned} \tag{5.1}$$

We added the last constraint to make equal the two margin, so we can re-write the maximization problem saying also that $\gamma = \frac{\hat{\gamma}}{||w||}$

$$\begin{aligned} \max_{\hat{\gamma},w,b} \quad & \frac{\hat{\gamma}}{||w||} \\ \text{s.t.} \quad & y_i(w^T x_i + b) \geq \hat{\gamma} \qquad i = 1, ..., m \end{aligned} \tag{5.2}$$

Now this is still a difficult problem, but provides us a shortcut. In fact recalling that $\hat{\gamma}$ can be scaled by w and b, we can introduce a scaling constraint $\hat{\gamma} = 1$ and we transform the maximization problem of $\frac{1}{||w||}$ into a minimization problem of $||w||^2$.

$$\begin{aligned} \min_{\gamma,w,b} \quad & \frac{1}{2}||w||^2 \\ \text{s.t.} \quad & y_i(w^T x_i + b) \geq 1 \qquad i = 1, ..., m \end{aligned} \tag{5.3}$$

This formulation represent the **optimal margin classifier** and is a **quadratic programming** problem which can be solved in a straight way using the Lagrangian function. But before that we specify that, we denote the fact that SVM take into account only the points with minimum margin from the decision boundary, which are called **support vectors**, in fact it use just these points to generate the optimal hyperplane to split the data. [5]

Finally using the Lagrangian function with m Lagrange multipliers we can re-write the minimization problem into a new optimization problem as follow:

$$\begin{aligned} \max \quad & L_D(\lambda_1, ..., \lambda_m) = \sum_{i=1}^{m} \lambda_i - \frac{1}{2} \sum_{i=1}^{m} \sum_{j=1}^{m} \lambda_i \lambda_j y_i y_j x_i^T x_j \\ \text{s.t.} \quad & \sum_{i=1}^{m} \lambda_i y_i = 0 \quad \lambda_i \geq 0, \forall i = 1, ..., m \end{aligned} \tag{5.4}$$

Up to now we have discuss on **linearly separable problem**, but what about **NON linearly separable problem**? SVM take into account also this fact using a strategy called **kernel trick** for learning a possible separating hyperplane in a new space. In some critical case could be useful evaluate a transformation of the original space, this is defined with a function $\phi(x)$. In others words it is a function that

applies the mapping of a feature vector to another one. This function is nothing else than an inner product between feature mapping of ϕ:

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

Moreover instead of doing the inner product between the input vectors (x,z), the SVM classifier use the kernel function in order to be able to learn in a different feature space.

But we have to bee careful because the Kernel Trick has to satisfy two rules in order to consider the **kernel a valid one:**

- $K(x,z) = \phi(x)^T \phi(z) \quad \forall x,z \in S$
- **Positivity of a kernel:** A symmetric function $K : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}$ is a **positive define kernel** if:

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

There exist lots of kernel function, but in this report we focus only in the most common three, which are:

- **Linear:** $K(x_i,x_j) = x_i^T x_j$
- **Polynomial of degree 2:** $K(x_i,x_j) = (1 + x_i^T x_j)^2$
- **Gaussian Radial Basis Function:** $K(x_i,x_j) = exp\Big(-\frac{||x_i - x_j||^2}{2\sigma^2}\Big)$

5.2 Angular Kernel

One of the topic of the report is to create a kernel which uses only the angular information. We analyze this because the three kernel defined previously are affected by the length of each vector, this means that if one of the three kernel take into account two vectors with same direction but different length it see them as not similar. Thus to consider only the direction of each vector is necessary to make a normalization of them such that at the end $||x||^2 = 1$.

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

$$\phi : \mathbb{R}^m \to \mathbb{R}^m$$

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

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

As we can see the previous kernel takes only in consideration the cosine of the angle generated by the vectors x_i and x_j. Moreover the last thing that we have to analyse is to prove that this kernel is a positive define kernel. So we have to prove that:

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

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

$$= \sum_{i=1}^{n}\sum_{j=1}^{n} c_i c_j \sum_{b=1}^{m} \frac{x_{ib}}{||x_{ib}||}\frac{x_{jb}}{||x_{jb}||}$$

$$= \sum_{i=1}^{n}\sum_{j=1}^{n}\sum_{b=1}^{m} c_i \frac{x_{ib}}{||x_{ib}||} c_j \frac{x_{jb}}{||x_{jb}||}$$

$$= \sum_{b=1}^{m}\Big(\sum_{i=1}^{n} c_i \frac{x_{ib}}{||x_{ib}||}\Big)\Big(\sum_{j=1}^{n} c_j \frac{x_{jb}}{||x_{jb}||}\Big)$$

$$= \sum_{b=1}^{m}\Big(\sum_{i=1}^{n} c_i \frac{x_{ib}}{||x_{ib}||}\Big)^2 \geq 0$$

5.3 Implementation

In this section we have a look on the software implementation of the Support Vector Machine.

So starting from the beginning after loading the dataset into Numpy Array containing each of the observations, we generate the instance of the class *Svm* passing the feature vector and the label vector, and we call the *result* function which contains all the behaviour.

```
Svm(X, y).result()
```

Before jumping into the actual detail three useful constant value are instantiated, CV which indicate the 10 way cross validation that we discuss later in this section, KERNELS which indicates the three types of kernel that we adopt and PRINT which helps the user print to be more understandable.

```
CV = 10
KERNELS = ["linear", "poly", "rbf"]
PRINT = ["Measures of kernels without angular kernel evaluation:",
        "Measures of kernels with angular kernel evaluation:"]
```

Now we look into the initialization of the SVM class.

```
def __init__ (self, X, y):
    self.X = self.tfidf(X)
    self.norm_X = self.normalization(self.X)
    self.y = y
    self.non_normalized_data = train_test_split(
        self.X, self.y, test_size = 0.3)
    self.normalized_data = train_test_split(
        self.norm_X , self.y, test_size = 0.3)
```

In the init function the first attribute that we fill is the X attribute in which we pass the result of the *tfidf* function, where as the name it say it compute the TF-IDF transformation of the original dataset. That measure the relevance of a term respect to term frequency inside the document and its presence on the other document in the collection.

```
# Calculate the term frequency of each word in each mail
def tf_idf(self, X):
    idf = np.log10(X.shape[0] / (X != 0).sum(0))
    return X / 100.0 * idf
```

Next we initialize similar attribute but this time normalized, this one is useful for the evaluation of the angular kernel that we discuss later. For the normalization we adopt the *preprocessing.normalize* offered by sklearn package and apply the normalization to only non zero vectors for avoiding warning on division. We adopt this strategy because the new builded kernel it is nothing less than the original kernel with vector's length equal to 1 so in other word we have normalize the vectors.

```
# Normalize dataset
def normalization(self,X):
    x = copy.deepcopy(X)
    norms = preprocessing.normalize(x)
    nonzero = norms > 0
    x[nonzero] /= norms[nonzero]
    return x
```

The last operations assigned to the init function are the initialization of the y attribute and the two lists of vector respectively containing the *train_test_split* of the original and normalized data. For both split we decide to assign 70% of the data to the training phase and 30% to the testing phase.

Furthermore we can start to discuss the *result* function.

```
def result(self):
    print("SUPPORT VECTOR MACHINE")

    for norm_flag, p in enumerate(PRINT):
        print(p)
        for k in KERNELS:
            classifier = self.create_classifier(k)
            results = self.evaluation_kernel(classifier, norm_flag)
            X_test, y_test, score = self.score(classifier, norm_flag)
            print_scores(results, k, score)
            print_confusion_matrix(classifier, X_test, y_test, p, k)
```

We run each kernel two times, the first one on the original dataset and the second on the normalized dataset. Firstly we get the actual classifier from the SVC sklearn

package, noting that in both cases we set the parameter $C = 1$. The C parameter tells the SVM optimization how much you want to avoid misclassifying each training example. For large values of C, the optimization will choose a smaller-margin hyperplane if that hyperplane does a better job of getting all the training points classified correctly. Conversely, a very small value of C will cause the optimizer to look for a larger-margin separating hyperplane, even if that hyperplane misclassifies more points. For very tiny values of C, you should get misclassified examples, often even if your training data is linearly separable. [11]

```
# Create the specified classifier
def create_classifier(self, kernel):
    if kernel == "poly":
        return SVC(kernel = kernel, degree = 2, C = 1)
    else:
        return SVC(kernel = kernel, gamma = 'scale', C = 1)
```

Next we make the evaluation of the kernel using the *cross_val_score* function, specifying the classifier, the X and y datset, cross-validation splitting strategy and the number of jobs to run in parallel.

```
# Kernel Evaluation
def evaluation_kernel(self, classifier, normalization_flag):
    if normalization_flag:
        return cross_val_score(
            classifier, self.norm_X, self.y, cv = CV, n_jobs = -1)
    else:
        return cross_val_score(
            classifier, self.X, self.y, cv = CV, n_jobs = -1)
```

And the last function is *score* used to obtain the accuracy score on the original and normalized test set.

```
# Score of train test split
def score(self, classifier, normalization_flag):
    if normalization_flag:
        X_train, X_test, y_train, y_test = self.normalized_data
    else:
        X_train, X_test, y_train, y_test = self.non_normalized_data
    clf = classifier.fit(X_train, y_train)
    return X_test, y_test, clf.score(X_test, y_test)
```

5.4 Results

In this section we have a look to the result obtained by the run of the three kernel using the original data and the normalized one. In each analysis we discuss the respective confusion matrix from the test set and some statistics about the cross_val_score. The confusion matrix allow us to see explicitly the performance of the classifiers on a test dataset. In summary in every matrix there are the predicted label and the true label with the number of true positive, true negative, false positive and false negative predicted observations.

5.4.1 Using Original Dataset

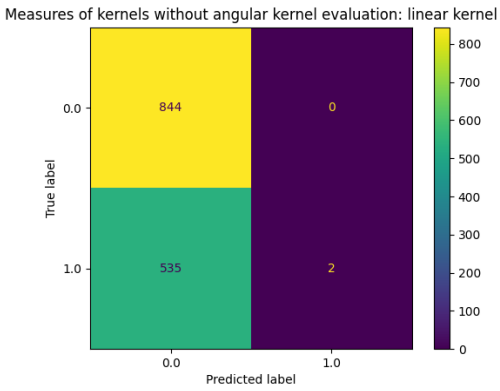

Figure 5.4: CM Linear Kernel - Original Dataset

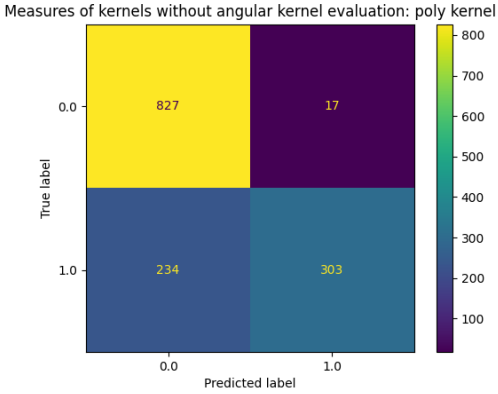

Figure 5.5: CM Poly Kernel - Original Dataset

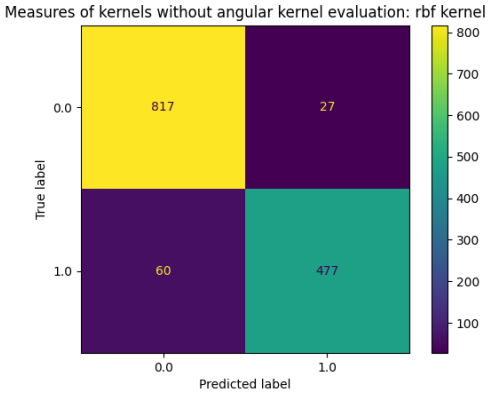

Figure 5.6: CM RBF Kernel - Original Dataset

As we can see from the Table 5.1 the linear kernel works pretty bad, where on the other hand polynomial and RBF kernel perform well.

Kernel Type	Min Accuracy	Mean Accuracy	Max Accuracy	Variance	Standard Deviation
Linear	0.60652	0.60835	0.61087	0.0	0.00155
Polynomial	0.76739	0.81047	0.8413	0.00035	0.01868
RBF	0.9087	0.92262	0.93913	8e-05	0.00909

Table 5.1: SVM - Cross Validation results using original dataset

5.4.2 Using Normalized Dataset

Now we focus on the normalized dataset, in which it consider only the angular information forgetting about vectors length. So after normalization we applied the same three kernel as before and obtain the following results.

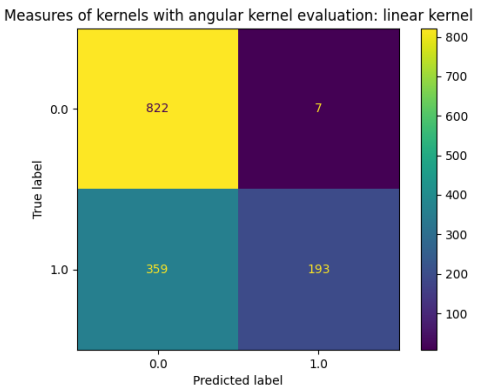

Figure 5.7: CM Linear Kernel - Normalized Dataset

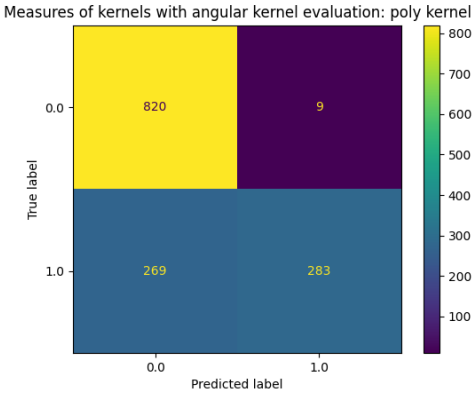

Figure 5.8: CM Poly Kernel - Normalized Dataset

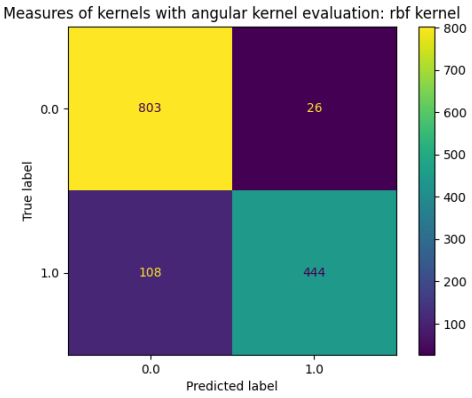

Figure 5.9: CM RBF Kernel - Normalized Dataset

Kernel Type	Min Accuracy	Mean Accuracy	Max Accuracy	Variance	Standard Deviation
Linear	0.71304	0.74722	0.77657	0.00025	0.0158
Polynomial	0.79783	0.82112	0.86304	0.00032	0.01789
RBF	0.89565	0.91719	0.93696	0.00013	0.0112

Table 5.2: SVM - Cross Validation results using normalized dataset

Checking both Table 5.1 and Table 5.2 we see a pretty big improvement for the linear kernel using only angular information since it pass from a mean accuracy of 0.60835 to 0.74767, indeed it is more on less 22.9% more accurate. On the other hand the remaining two kernels are almost the same in the two runs, which means that the vectors length does not affect them.

Chapter 6

Naive Bayes

A Naive Bayes classifier is a probabilistic machine learning model that is used for classification task. In this chapter we have a look on the theoretical definition, software implementation and comparison results.

6.1 Theoretical Definition

The crux of the classifier is based on the Bayes theorem.

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

This is the probability that a particular event A occur knowing that the event B has occurred. B is called **evidence** and A **hypothesis**. The only assumption that the theorem made is that the predictors or in other word the features are independent.

So remembering that we have to predict the label (ham or spam) of a specific new observed email given all its features, we can rewrite the Bayes formula like so:

$$P(y|X) = \frac{P(X|y)P(y)}{P(X)}$$

Where $y = \{0, 1\}$ indicate the target label and $X = \{x_1, x_2, x_3, ..., x_n\}$ is the feature vectors. Substituting X and expanding the previous equation using the chain rule we get:

$$P(y|x_1, ..., x_n) = \frac{P(x_1|y)P(x_2|y)...P(x_n|y)P(y)}{P(x_1)P(x_2)...P(x_n)}$$

For all entries in the dataset, the denominator does not change, it remain static. Therefore, we can remove it from the calculation and a proportionality can be introduced.

$$P(y|x_1, ..., x_n) \propto P(y) \prod_{i=1}^{n} P(x_i|y)$$

In our case as cited before our target variable y has only two options. So we nee to find the best class y that maximise the probability and in mathematical language this is formulated like so: [3]

$$y = \arg\max_{y} P(y) \prod_{i=1}^{n} P(x_i|y)$$

Then since all the values of the previous formula are probability between 0 and 1, it is possible that multiplying all these values we get a very small number that could led us into overflow problems. So in order to prevent this we apply the *log* function and the respective propriety to change the multiplications into sums.

$$y = \arg\max_{y}(\log P(y)) \sum_{i=1}^{n}(\log P(x_i|y))$$

In this formula we have to calculate the **prior probability** $P(y)$ which is the frequency of that class and the **class conditional probability** $P(x_i|y)$ is calculated by the following Gaussian distribution:

$$P(x_i|y) = (2\pi\sigma_y^2)^{-1/2} exp\left\{ -\frac{1}{2\sigma_y^2}(x_i - \mu_y)^2 \right\}$$

6.2 Implementation

In this section we discuss the software implementation of the Naive Bayes.

After loading the dataset we use the *train_test_split* for splitting the data into test set and train set respectively formed by the 30% and 70% of the original data. We call the function *callNb* which takes as input the original data and the splitted once.

In this function we test our implemented Naive Bayes and the original version imported from the Sklearn package using the 10 way cross validations score and the test set.

```
def callNb(X, y, X_train, X_test, y_train, y_test):
    print("IMPLEMENTED NAIVE BAYES")
    scores = cross_val_score(NaiveBayes(), X, y, cv = 10, n_jobs = -1)
    gnb_1 = NaiveBayes()
    gnb_1.fit(X_train, y_train)
    y_pred = gnb_1.predic(X_test)
    print_scores(scores, None, accuracy_score(y_test, y_pred))

    print("SKLEARN GAUSSIAN NB")
    scores = cross_val_score(GaussianNB(), X, y, cv = 10, n_jobs = -1)
    gnb_2 = GaussianNB()
    gnb_2.fit(X_train, y_train)
    y_pred = gnb_2.predict(X_test)
    print_scores(scores, None, accuracy_score(y_test, y_pred))
```

Now we have a look in the detail of the implemented version. First of all we have to discuss about the *fit* function.

```
def fit(self, X, y):
    n_samples, n_features = X.shape # Get the dataset shape
    self.__classes = np.unique(y) # Get the unique classes
```

```
    n_classes = len(self.__classes)

    # Initialize useful np.array
    self.__mean = np.zeros((n_classes, n_features), dtype=np.float64)
    self.__var = np.zeros((n_classes, n_features), dtype=np.float64)
    self.__priors = np.zeros(n_classes, dtype=np.float64)

    # Fill the np.array
    for c in self.__classes:
        X_c = X[y == c] # Get all email with labeled with c
        # Fill the row int(c) with the feature mean
        self.__mean[int(c), :] = X_c.mean(axis = 0)
        # Fill the row int(c) with the feature variance
        self.__var[int(c), :] = X_c.var(axis = 0) + 1e-128
        # Fill the row int(c) vector with its prior probability
        self.__priors[int(c)] = float(X_c.shape[0] / n_samples)
```

In this function we get the dataset shape, the unique classes and the length. Moreover the initialize the vector of mean, variance and prior probability. So on we do nothing else that populate these vectors with respect to the ham email and spam email.

Furthermore we can actually *predict* unseen observations.

```
def predic(self, X_test):
    return np.array([self.__predict(test_email) for test_email in X_test])

def __predict(self, x):
    post_probs = []

    # Calculate posterior probability for each class
    for c in self.__classes:
        prior_prob = np.log(self.__priors[int(c)])
        post_prob = np.sum(np.log(
            self.__gaussianDistribution(int(c), x) + 1e-128))
        post_probs.append((prior_prob + post_prob))

    # Return class with highest posterior probability
    return self.__classes[np.argmax(post_probs)]

# Gaussian Distribution Function
def __gaussianDistribution(self, class_idx, x):
    num = np.exp(-(np.power((x - self.__mean[class_idx]), 2)) / (
        2 * self.__var[class_idx]))
    den = np.sqrt(2 * np.pi * self.__var[class_idx])
    return num / den
```

In the *__predict* function we apply the argmax function explained before returning the maximum probability between the two classes.

```
def accuracy_score(self, y_pred, y_test):
    return np.sum(y_test == y_pred) / len(y_test)

def score(self, x_test, y_test):
    y_pred = self.predic(x_test)
    return self.accuracy_score(y_test, y_pred)
```

And finally the *score* function that calculate the array of predicted labels and return the total accuracy of the prediction.

6.3 Results

In this section we compare the cross validation results of the implemented and Sklearn version of Naive Bayes and the previous SVM kernel using the normalized dataset.

Classifier	Min Accuracy	Mean Accuracy	Max Accuracy	Variance	Standard Deviation
SVM - Linear	0.71304	0.74722	0.77657	0.00025	0.0158
SVM - Polynomial	0.79783	0.82112	0.86304	0.00032	0.01789
SVM - RBF	0.89565	0.91719	0.93696	0.00013	0.0112
Implemented NB	0.75217	0.802	0.83913	0.00048	0.02188
Sklearn NB	0.78696	0.81243	0.84348	0.00025	0.01569

Table 6.1: SVM - NB Cross Validation Results

As we can see from Table 6.1 the implemented version has more on less the same accuracy as the Sklearn version, and both perform just a little less respect to the SVM classifier with polynomial kernel of degree 2.

Chapter 7

K-Nearest Neighbors

K-Nearest Neighbors is one of the simplest and intuitive techniques adopted in ML problem. In this chapter we have a look on the theoretical definition, software implementation and comparison results.

7.1 Theoretical Definition

The key concept of K-Nearest Neighbors is to find a predefinite number of training samples (K) closest to a new given point and predict the label based on most common label of the nearest points.

K-NN is also known as *non-parametric* ML algorithm since it makes no assumption on the data so the prediction procedure is made only from the given data. Despite its simplicity, nearest neighbors has been successful in a large number of classification and regression problems, including handwritten digits and satellite image scenes. Being a non-parametric method, it is often successful in classification situations where the decision boundary is very irregular. [6] Moreover K-NN is a *lazy learner* because it does not learn a discriminative function from the training data but "memorizes" the training dataset instead. [9]

Like we said before K-Nearest Neighbors is an easy to use algorithm and it makes no assumption, but it is for this reason that we have to choose carefully the number of neighbors K. So if the training data is accurate we can take into account this technique otherwise we have to discard it.

Moreover regarding the choice of K, usually if we are in a classification problem the K is selected to be odd number or we could impose k as the squared root of the number of data points in training data.

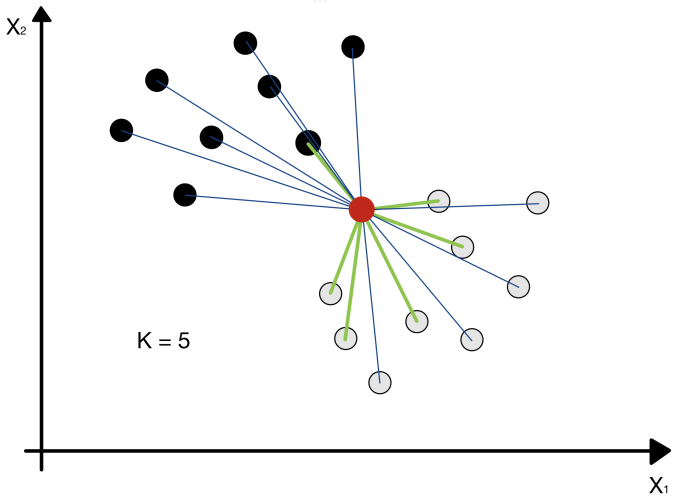

Figure 7.1: K-Nearest Neighbours Example

In Figure 7.1 we can see an application of K-NN for a new observation (red point). The first task to do is to obtain all the "distance" from the red point to all the other train point, done that we take into consideration only the first K nearest point and we label this new observation with the most popular class of these set, which in this case is "grey".

In the previous paragraph we have cite the distance, in fact K-Nearest Neighbors use the *Euclidean Distance* to calculate how much far is the new point from all the other train points. This measure is formulated like so:

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

Where:

- p and q are two points in the n-dimension Euclidean space
- q_i and p_i are Euclidean vector from the original space
- n is the dimension of the feature vector

7.2 Implementation

In this section we discuss the software implementation of the Naive Bayes.

So after loading the dataset we use the *train_test_split* for splitting the data into test set and train set respectively formed by the 30% and 70% of the original data and we call the function *callKnn* which takes as input the original data and the splitted once.

In this function we test our implemented K-Nearest Neighbors and the original version imported from the Sklearn package using the 10 way cross validations score and the test set.

```
1 def callKnn(X, y, X_train, X_test, y_train, y_test):
2     print("IMPLEMENTED K-NEAREST NEIGHBORS")
```

```
    scores = cross_val_score(KNearesNeighbour(), X, y, cv = 10,
        n_jobs = -1)
    kNN_1 = KNearesNeighbour()
    kNN_1.fit(X_train, y_train)
    y_pred = kNN_1.predict(X_test)
    print_scores(scores, None, accuracy_score(y_test, y_pred))

    print("SKLEARN K-NEAREST NEIGHBORS")
    scores = cross_val_score(KNeighborsClassifier(n_neighbors = 5),
        X, y, cv = 10, n_jobs = -1)
    kNN_2 = KNeighborsClassifier(n_neighbors = 5)
    kNN_2.fit(X_train, y_train)
    y_pred = kNN_2.predict(X_test)
    print_scores(scores, None, accuracy_score(y_test, y_pred))
    print_confusion_matrix(kNN_2, X_test, y_test)
```

In the initialization of the object with class *KNearesNeighbour* we just set the specific K fixed is known and in the *fit* function we set the train feature vector ad the train target vector.

```
def __init__(self):
    self.K = 5

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

Moreover going deeply in the computation we call the function *predict* which predict all the observations of the test feature vector.

```
def predict(self, X_test):
    # For each test point i predict the label
    return np.array([self.__predict(test_email)
        for test_email in X_test])

def __predict(self, test_email):
    # Calcualte the distance from the particular
        # point and all the trein points
    dist = [self.__euclideanDistance(test_email, train_email)
        for train_email in self.__X_train]
    # Get the first k indices of the sorted vector dist
    k_idx = np.argsort(dist)[: self.K]
    # Return the most popular label
    return Counter(self.__Y_train[k_idx]).most_common(1)[0][0]

def __euclideanDistance(self, vec1, vec2):
    return np.sum(np.sqrt(np.power((vec1 - vec2), 2)))
```

__precict__ is the auxiliary function in which we calculate the euclidean distance from a specific point and all train point. We get the first K indices from the sorted vector of distances and return the most common label.

```
def accuracy_score(self, y_pred, y_test):
    return np.sum(y_test == y_pred) / len(y_test)

def score(self, x_test, y_test):
    y_pred = self.predict(x_test)
    return self.accuracy_score(y_pred, y_test)
```

And finally the *score* function that calculate the array of predicted labels and return the total accuracy of the prediction.

7.3 Results

In this section we compare the cross validation results of the implemented and Sklearn version of K-Nearest neighbors and the previous SVM kernel using the normalized dataset. Moreover we have a look on the confusion matrix using the original version of K-NN in the test dataset, Figure 7.2.

Classifier	Min Accuracy	Mean Accuracy	Max Accuracy	Variance	Standard Deviation
SVM - Linear	0.71304	0.74722	0.77657	0.00025	0.0158
SVM - Polynomial	0.79783	0.82112	0.86304	0.00032	0.01789
SVM - RBF	0.89565	0.91719	0.93696	0.00013	0.0112
Implemented KNN	0.89783	0.91653	0.94143	0.00022	0.01499
Sklearn KNN	0.8913	0.90958	0.93059	0.00015	0.01237

Table 7.1: SVM - KNN Cross Validation Results

As we can see from Table 7.1 the implemented version has more on less the same accuracy as the Sklearn version, and both perform just a little less respect to the SVM classifier with RBF kernel.

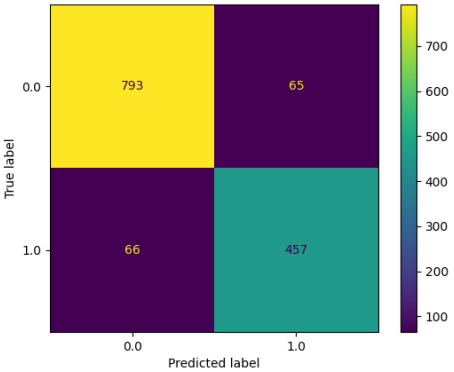

Figure 7.2: CM K-Nearest Neighbours

Chapter 8

Summary & Comparison

Classifier	Min Accuracy	Mean Accuracy	Max Accuracy	Variance	Standard Deviation
SVM - Linear	0.71304	0.74722	0.77657	0.00025	0.0158
SVM - Polynomial	0.79783	0.82112	0.86304	0.00032	0.01789
SVM - RBF	0.89565	0.91719	0.93696	0.00013	0.0112
Implemented NB	0.75217	0.802	0.83913	0.00048	0.02188
Sklearn NB	0.78696	0.81243	0.84348	0.00025	0.01569
Implemented KNN	0.89783	0.91653	0.94143	0.00022	0.01499
Sklearn KNN	0.8913	0.90958	0.93059	0.00015	0.01237

Table 8.1: Summary of Cross Validation Results

Finally in Table 8.1 we can see a summary of all the algorithm taken into account in this report, understanding that the the K-Nearest Neighbors and SVM with RBF kernel perform better than the others. Note that the SVM statistics are regarding the normalized dataset.

Moreover we can make a comparison between these three algorithms.

- **SVM Vs NB:** Support Vector Machine spend lots of time during the learning phase since it only uses the support vectors to generate the best hyperplane but in order to to this it has to discover them. The Learning phase for Naive Bayes is more on less absent since it uses statistics from all the training data, which is much faster. On the other hand SVM supports non-separable problem by the use of the kernel trick, where Naive Bayes do not support this kind of problem.

- **KNN Vs NB:** K-Nearest Neighbors is much slower than Naive Bayes since its real time executions and KNN is non-parametric since it does not make any assumption on the dataset, whereas Naive Bayes is parametric because it makes the assumptions that all the observations are independent.

- **SVM Vs KNN:** this comparison is much closer since both algorithms have to tune an hyperparameter, K for KNN and C for SVM. If the number of observations in the training data highly overcomes the number of features KNN performs better than SVM, on the other hand if we have a small training set in which each observations have lots of features SVM outperforms KNN. Moreover SVM take care of outliers in a better way than KNN, but the last one can deal more closely with non linear boundaries since it has more data to work with.

Chapter 9

Conclusion

During this report we discuss about the application of three ML classification algorithm: Support Vector Machine, Naive Bayes and K-Nearest Neighbors. We have analysed the theoretical aspect, the adopted implementation, the result and made a comparison between them. So we have all the key to make a conclusion.

In conclusion starting from SVM we can mention its powerful to split non-separable data by the already mentioned kernel trick, moreover the normalization of the dataset brings with it an increasing in prediction accuracy two out of three kernel. With RBF kernel overcoming the other two in both original and normalized data. Continuing we study the Naive Bayes algorithm, a quickly strategy that perform well in the case its assumption is satisfied, in our case there is the possibility that it is no so satisfied since it performs well but not as the level of SVM - RBF kernel. And finally K-Nearest Neighbors, one of the most intuitive ML algorithm ever made performs pretty well, in fact it is about on the same prediction accuracy of SVM - RBF, meaning that its key points, non-parametric and lazy algorithm made a huge impact in our dataset.

Bibliography

[1] Siwei Causevic. *Generative vs. Discriminative Probabilistic Graphical Models*. URL: https://towardsdatascience.com/generative-vs-2528de43a836.

[2] Dheeru Dua and Casey Graff. *UCI Machine Learning Repository*. 2017. URL: http://archive.ics.uci.edu/ml.

[3] Rohith Gandhi. *Naive Bayes Classifier*. URL: https://towardsdatascience.com/naive-bayes-classifier-81d512f50a7c.

[4] Chirag Goyal. *Deep Understanding of Discriminative and Generative Models in Machine Learning*. URL: https://www.analyticsvidhya.com/blog/2021/07/deep-understanding-of-discriminative-and-generative-models-in-machine-learning/.

[5] Jeremy Jordan. *Support vector machines*. URL: https://www.jeremyjordan.me/support-vector-machines/#:~:text=Summary%3A%20The%20functional%20margin%20represents,xi%20to%20the%20hyperplane.&text=Note%3A%20If%20%7Cw%7C%3D,%2C%20%CE%B3%2C%20are%20equivalent%20expressions.

[6] scikit learn. *Nearest Neighbors*. URL: https://scikit-learn.org/stable/modules/neighbors.html#nearest-neighbors-classification.

[7] Tornike Onoprishvili. *SpamBase — Data Exploration & Analysis*. URL: https://medium.com/@tonop15/spambase-data-exploration-analysis-9a3d6d83ee78.

[8] F. Pedregosa et al. "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[9] Sebastian Raschka. *Why is Nearest Neighbor a Lazy Algorithm?* URL: https://sebastianraschka.com/faq/docs/lazy-knn.html.

[10] Rahil Shaikh. *Cross Validation Explained: Evaluating estimator performance.* URL: https://towardsdatascience.com/cross-validation-explained-evaluating-estimator-performance-e51e5430ff85.

[11] Marc Shivers. *What is the influence of C in SVMs with linear kernel?* URL: https://stats.stackexchange.com/questions/31066/what-is-the-influence-of-c-in-svms-with-linear-kernel.