HANDWRITTEN DIGIT RECOGNITION USING SVM, RANDOM FORESTS, NATIVE BAYES CLASSIFIER and K-NN

Ca' Foscari University of Venice
Master's Degree in Computer Science and Information Technology
[LM-18]

Palmisano Tommaso, 886825

I. Abstract

Handwritten digit recognitions may appear trivial; in fact, nowadays we are used to problems that are significantly more complex. However, trying to solve it by using classical programming methods results in a quite challenging task. In this document, we will explore how it's possible to crack this problem by using Machine Learning techniques; in particular, we will analyse the differences between two different approaches: **Discriminative** and **Generative classifiers**.

II. Introduction

Discriminative and Generative classifiers are Machine Learning models used to classify data by learning from labeled examples. Discriminative Classifiers try to separate the data by inferring the boundary between different classes or by estimating the probability distribution $P(Y|X)$ of the output Y and input X. To this class belong **SVM** and **Random Forests**. On the other hand, Generative Classifiers model the joint probability distribution $P(X, Y)$ between X and Y, and computes $P(Y|X)$ by using the *Bayes Theorem*. One example is the **Naive Bayes algorithm** [1].

2.1. A bit of history

The **MNIST database** (Modified National Institute of Standards and Technology database) is a famous collection of images of handwritten digits, often used in the training of classifiers. The original database, the NIST, was composed of a training set (SD-3) build over American Census Bureau employees handwrites, while the testing set (SD-1) was taken from American high school students.

However, since the images of the training set were much cleaner, the algorithms trained on the NIST performed poorly on the test set. In order to avoid this issue, the MNIST training set is composed of 30.000 images from SD-3 and 30.000 images from SD-1, while the test set is composed of 5.000 images from SD-3 and 5.000 images from SD-1, while also making sure that the sets of writers of the training and test set are disjoint.

The result are two sets of 60.000 and 10.000 gray scare images. Each image, originally 128x128 pixel, was compressed and centred into a 28x28 image [2].

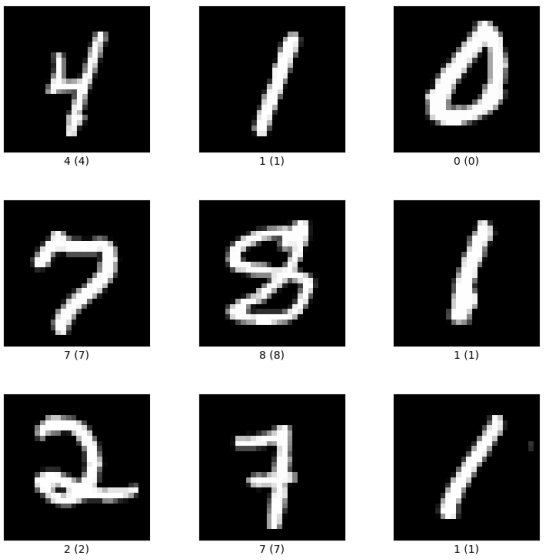


Figure 1: Sample of digits from the MNIST database.

2.2. Background

Before exploring the methodologies for solving the problem, we need to introduce some fundamental concepts in Artificial Intelligence, such as: **Bias**, **Variance** and **Linearly Separable Problems**. With *Bias* we mean the inability of a model to capture the true relationship between the data, mainly because the model is too simple. The *Variance* is the difference in fit between two training sets, often used to measure the difference in performance on the training and test sets. The error of a model is calculated using a *Loss Function*, a simple one is *Sum of Squares* (SSE).

$$SSE = \sum_{i=1}^{n} (y_i - \hat{y}_i)^2$$

where y_i is the desired output for input x_i, and $\hat{y}_i$ is the observed output.

When training a ML model, we need a way to represent the features of objects so that the model can manage them effectively. For example, if we want a model to categorize fruits, we can provide it with features like weight, color, and shape. We can then characterize a fruit using a vector, where each entry represents a feature. In our case, we are dealing with images. We can then represent an image using a vector, where the domain of each component is integers from 0 to 255, representing the intensity of the color black to white. Each entry in the vector corresponds to a

pixel in the image, resulting in a vector of dimension 784. Therefore, we can display the entire dataset in a high-dimensional vector space, where each element is a vector. For the sake of our discussion, let's now consider vectors of dimension two, so that they can be plotted on a plane:

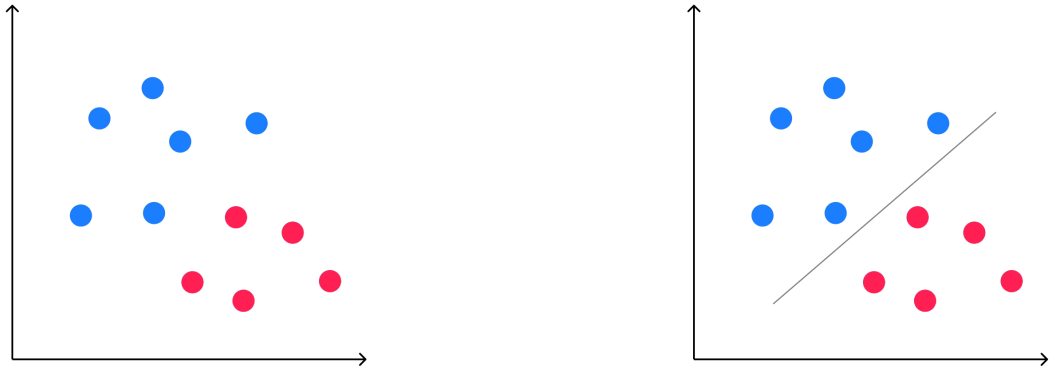


Figure 2: Example of a linearly separable problem.

Note that the points belongs to two different classes, red and blue. A **Linearly Separable** problem is one where the points can be distinct into two classes by an *hyperplane*. We recall that an hyperplane in a space of dimension n, is a flat surface of dimension $n-1$, a line in our case [3].

2.3. Cross Validation and Grid Search

When training a machine learning model, what the training algorithm does is to use the training set to fine-tune a set of parameters, often called *weights*. Most models have another set of parameters called **hyperparameters**, which are set before the training phase. The **Cross Validation** algorithm is a methodology to test a set of hyperparameters for a model, and works in the following way: the training set is split into k subsets, typically 5 or 10; one of them becomes the **validation set**, and the training of the model is performed on the remaining ones. Then the model is tested on the validation set and the process is repeated k times, each with a different validation set. By taking the mean value of the rate of errors, we can obtain an accurate estimate of the performances of the model with the given hyperparameters.

Thanks to cross validation, we can implement an algorithm called **Grid Search**, used to determine the best hyperparameters for a model. The idea is very simple: we decide a priori a set of possible values for the model's hyperparameters: let's suppose $\alpha = \{1, 2, 3\}$ and $\beta = \{10, 20, 30\}$. The algorithm will create n models, where n is the number of possible combinations of α and β, and then use *cross validation* to test their performances. The best performing model is the one that will be trained on the whole training set.

2.4. Data Management and Modeling Tools

Scikit-learn, or *sklearn* from now on, is a popular Python open-source library built on top of SciPy, which provides simple and efficient tools for predictive data analysis [4]: we will use it to train SVMs and Random Forests.

Before starting the implementation of our models, we must retrieve the dataset and separate the training set from the testing set:

```
def retrive_mnist():
    X, y = fetch_openml('mnist_784', version=1, return_X_y=True)
    y = y.astype(int)
    X = X / 255.

    X_train, X_test, y_train, y_test = train_test_split(X, y,
    ↪  test_size=0.2, random_state=42)

    return X_train, X_test, y_train, y_test
```

The function fetches the dataset using $fetch\_openml$ from sklearn and saves the images in the variable X, while the corresponding labels are saved in the variable y. The type of the data is **pandas.DataFrame**, which provides several advantages including various methods for easily handling the data, seamless integration with sklearn models, and enhanced data readability thanks to the use of a tabular format. Then the data are *normalized*, ensuring each pixel value ranges from 0 to 1: this is a common practice in ML, especially when dealing with features of different magnitudes since it helps models converge more quickly during training [5]. Finally, the dataset and labels are separated into training and testing sets using $train\_test\_split$.

As mentioned earlier, we will use the models offered by sklearn for our experiments. We will begin by explaining the theory behind a model, then proceed to outline the experiment's setup and, finally, some analysis and considerations regarding the experimental results.

III. SVMs

We already introduced the concept of linear classification; the hyperplane separating two classes of objects is also called **threshold** or **separator**, and the distance between the observations and the threshold **margin**. We could try to use a threshold that gives the largest possible margin, in this case we are using a **Maximal Margin Classifier**. However, such method is sensitive to outliers in the data. One solution is to use a **Soft Margin**, i.e. to allow misclassifications, and find the right trade off between *bias* and *variance*. When we use a soft margin to determine the location of the threshold, then we are using a **Soft Margin Classifier**, also called **Support Vector Classifier**. These classifiers are based on the concept of **Support Vectors**, which are the data points that lie closest to the threshold.

Formally speaking, given the support vector x: being $x_\perp$ the projection of x on the *hyperplane*, r its perpendicular distance and w the normal vector of the *hyperplane*, the margin is given by

$$x = x_\perp + r\frac{w}{\|w\|}$$

since $x_\perp$ belongs to the hyperplane:

$$w^T x_\perp + b = 0$$

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

$$w^T x - r\frac{w^T w}{\|w\|} + b = 0$$

$$w^T x - r\|w\| + b = 0$$

$$r = \frac{w^T + b}{\|w\|}$$

We have the freedom of choosing the normalization of w and b, so we choose the canonical form for the positive support vector x_1 and the negative one for x_2

$$w^T x_1 + b = 1 \qquad w^T x_2 + b = -1$$

Hence, the total margin is given by:

$$\frac{|w^T x_1 + b|}{\|w\|} + \frac{|w^T x_2 + b|}{\|w\|} = \frac{2}{\|w\|}$$

We now want to maximize it, or equivalently solve:

$$\begin{aligned} \min_w \quad & \frac{1}{2}\|w\|^2 \\ \text{s.t.} \quad & y_i(w^T x_i + b) - 1 \geq 0 \\ & \forall i \in \{1, ..., n\} \end{aligned} \tag{1}$$

The problem is constrained, requiring every margin to be grater than or equal to 1. We turn it into an unconstrained one using *Lagrange multipliers*:

$$\min_{w} \quad \mathscr{L} = \frac{1}{2} w^T w - \sum_{i=1}^{n} \alpha_i [y_i (w^T x_i + b) - 1] \tag{2}$$

where each α_i is the Lagrange multiplier. Then, we take the partial derivatives in respect to w and b, obtaining:

$$\frac{\partial \mathscr{L}}{\partial w} = w - \sum_{i=1}^{n} \alpha_i y_i x_i = 0 \qquad \frac{\partial \mathscr{L}}{\partial w} = -\sum_{i=1}^{n} \alpha_i y_i = 0$$

and by substituting those into equation (2), we and up with:

$$w(\alpha, b) = \frac{1}{2} w^T \sum_{i=1}^{n} \alpha_i y_i x_i - w^T \sum_{i=1}^{n} \alpha_i y_i x_i - b \sum_{i=1}^{n} \alpha_i y_i + \sum_{i=1}^{n} \alpha_i$$

$$w(\alpha, b) = \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \left(\sum_{i=1}^{n} \alpha_i y_i x_i \right) \cdot \left(\sum_{j=1}^{n} \alpha_j y_j x_j \right) - b \sum_{i=1}^{n} \alpha_i y_i$$

$$w(\alpha) = \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i=1}^{n} \sum_{i=1}^{n} \alpha_i \alpha_j y_i y_j (x_i^T x_j) \tag{3}$$

And the following optimization problem is called **dual problem**.

$$\begin{aligned} \max_{\alpha} \quad & w(\alpha) = \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i=1}^{n} \sum_{i=1}^{n} \alpha_i \alpha_j y_i y_j (x_i^T x_j) \\ \text{s.t.} \quad & \alpha_i \geq 0 \\ & \forall \in \{1, ..., n\}, \quad \sum_{i=1}^{n} \alpha_i y_i = 0 \end{aligned} \tag{4}$$

We can now optimize with respect to α, note that the problem is convex. If $\alpha_i = 0$ the constraint is satisfied with no distortion, and if $\alpha_i > 0$ the constraint is satisfied with equality. In this case x_i is a support vector.

In order to introduce the *soft margin* we discussed before, we add a new term to allow for deviation and to make a few errors if necessary.

So our constraints optimization now becomes:

$$\begin{aligned} \min_{w} \quad & \frac{1}{2}\|w\|^2 + C\sum_{i=0}^{n}\zeta_i \\ \text{s.t.} \quad & y_i(w^T x_i + b) \geq 1 - \zeta_i \\ & \zeta_i \geq 0 \\ & \forall i \in \{1, ..., n\} \end{aligned} \tag{5}$$

where ζ is the distance of a misclassified point from its correct hyperplane, and C controls how soft the margin is. In particular, by choosing a large value of C we try to correctly classify as many point a possible, while a low value let's us give up on more points [6] [7].

3.1. The kernel Trick

Let's consider the following example:

Figure 3: Example of non-linearly separable data.

It should be evident that there's no way to set a threshold in such a way that the data are well separated. What we can do is use a function to move the points to a higher-dimensional space, and then try to find a SVC:

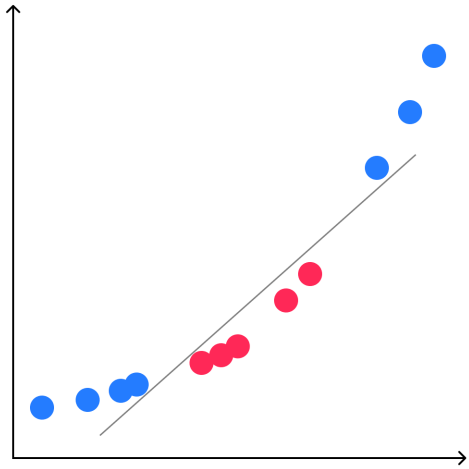


Figure 4: Application of the Kernel Trick.

The function used to calculate the higher dimension relationship between two points is called **Kernel Function**. The main advantage is that thanks to it, we don't need to actuality move the points to a higher dimension space, but we can just calculate the relationships as if they were in a higher dimension, this is called the **Kernel Trick**; we will see an example in the following section.

3.2. Polynomial Kernel

Given two points, the simplest kernel function we can imagine is the **Linear Kernel**:

$$K(x_1, x_2) = x_1 \cdot x_2$$

So, the linear kernel is used to simply calculate the relationship between two point in the the original input space by computing the dot product. It is useful when the data are linearly separable.

In order to implement the *kernel trick* we need a function to calculate the relationship as if the points where in a higher dimension space. An example of such function is the **Polynomial Kernel:**

$$K(x_1, x_2) = (x_1 \cdot x_2 + r)^d$$

where d is the degree of the polynomial, and r is a constant parameter. To understand how the points are mapped, let's consider the *Polynomial Kernel* of degree 2; we also choose $r = 1$ for simplicity:

$$\begin{aligned} K(x_1, x_2) &= (x_1 \cdot x_2 + 1)^2 \\ &= (x_1 \cdot x_2 + 1)(x_1 \cdot x_2 + 1) \\ &= x_1^2 x_2^2 + 2x_1 x_2 + 1 \\ &= (x_1^2, \sqrt{2}x_1, 1) \cdot (x_2^2, \sqrt{2}x_2, 1) \end{aligned}$$

Now it's clear why plotting the value of the kernel function is equivalent to moving the point to a higher dimension, in fact the plot is equal to the dot product of the point as if the were in 3 dimensions, although we can ignore the third term since it is constant [8].

3.3. Radial Basis Function Kernel

The last Kernel function we will introduce and use for our experiments is the **Radial Basis Function Kernel**, **RBF** from now on:

$$K(x_1, x_2) = e^{-\gamma(x_1 - x_2)^2}$$

What this function does is calculating the relationship between two points in function of their distance, so the farther apart two points are, the less influence they have on each other; the parameter γ scales the influence the distance has. The RBF function actually calculates the relationship between the two point in a infinite-dimension space, but what does it mean?

Let's consider again the polynomial kernel, this time with $r = 0$; no mater what value of d we choose, the relationship between the two points stays in the same dimension of the input space. What happens is that the coordinates of the points get stretched, but this doesn't give us any advantage in finding a better hyperplane:

$d = 1$

$$K(x_1, x_2) = x_1 x_2 \ = (x_1) \cdot (x_2)$$

$d = 2$

$$K(x_1, x_2) = x_1^2 x_2^2 \ = (x_1^2) \cdot (x_2^2)$$

and so on...

What we can do is add those polynomials together starting from grade 0 up to $d = \infty$, obtaining a polynomial of grade infinity, which would allow us to calculate the higher dimension relationship between two points as if they were in a infinite-dimension space, which would be great for our applications:

$$\begin{aligned} K(x_1, x_2) &= 1 + x_1 x_2 + x_1^2 x_2^2 + ... + x_1^\infty x_2^\infty \\ &= (x_1, x_1^2, ..., x_1^\infty) \cdot (x_2, x_2^2, ..., x_2^\infty) \end{aligned}$$

How does the RBF achieve this? Let's develop the square:

$$\begin{aligned} K(x_1, x_2) = K(x_1, x_2) &= e^{-\gamma(x_1 - x_2)^2} \\ &= e^{-\gamma(x_1^2 + x_2^2 - 2x_1 x_2)} \\ &= e^{-\gamma(x_1^2 + x_2^2)} e^{2\gamma(x_1 x_2)} \end{aligned}$$

and set $\gamma = \frac{1}{2}$ for simplicity

$$= e^{-\frac{1}{2}(x_1^2+x_2^2)} e^{(x_1 x_2)}$$

We can now expand the second term using the Taylor expansion, and by setting $\bar{x} = 0$ we get:

$$e^{(x_1 x_2)} = 1 + \frac{1}{1!}x_1 x_2 + \frac{1}{2!}(x_1 x_2)^2 + ... + \frac{1}{\infty!}(x_1 x_2)^\infty$$

$$= \left(1, \sqrt{\frac{1}{1!}}x_1, \sqrt{\frac{1}{2!}}x_1^2, ..., \sqrt{\frac{1}{\infty!}}x_1^\infty\right) \cdot \left(1, \sqrt{\frac{1}{1!}}x_2, \sqrt{\frac{1}{2!}}x_2^2, ..., \sqrt{\frac{1}{\infty!}}x_2^\infty\right)$$

and by finally multiplying by the first term, which we will write as $s = e^{\sqrt{-\frac{1}{2}(x_1^2+x_2^2)}}$, we finally obtain:

$$\left(s, s\sqrt{\frac{1}{1!}}x_1, s\sqrt{\frac{1}{2!}}x_1^2, ..., s\sqrt{\frac{1}{\infty!}}x_1^\infty\right) \cdot \left(s, s\sqrt{\frac{1}{1!}}x_2, s\sqrt{\frac{1}{2!}}x_2^2, ..., s\sqrt{\frac{1}{\infty!}}x_2^\infty\right)$$

and we see that the Radial Kernel is equal to a Dot Product that has coordinates for an infinite number of dimensions.[9]

3.4. Handwritten digit recognition using SVMs

As we already said, we will use sklearn for creating, training and evaluating our models. All experiments were conducted on an M4 Mac mini, and the parameters tuning was performed through *grid search* with 10 folds *cross validation*. The process of training a model is the following:

1. Retrieve the MNIST.
2. Create the model from sklearn's class *svm.SVC* with the desired kernel.
3. Create a dictionary containing all the parameters we want to test.
4. Create a *GridSearch* object by passing our model, the *params* dictionary and setting all the other necessary parameters, then, launch the fitting process.
5. Once we obtain the result, we can train the model on the whole training set, and evaluate its performance on the test set.
6. Finally, the model is saved to be used when needed.

The python code for creating the model, performing the grid search, training and saving the best model for the linear kernel is the following:

```
svcl = svm.SVC(kernel='...')

params = {
    'C': [0.001, 0.01, 0.1, 1, 10, 100, 1000]
}

gs = GridSearchCV(svcl, params, n_jobs=-1, cv=10,
↪  scoring='accuracy', verbose=5)
gs.fit(X_train, y_train)

best_C = gs.best_params_['C']

svcl = svm.SVC(kernel='linear', C=best_C)
svcl.fit(X_train, y_train)
print(f"Trained model accuracy: {svcl.score(X_test, y_test)}")

save_model(svcl, "svm", "svc_linear.pkl")
```

We will start by using a SVC with Linear Kernel: the only parameter to choose is C. The tests are sorted from the best performing model to the worst:

Table 1: SVC with Linear Kernel

	Paramater C	Mean Training Score
1	0.1	94.3267%
2	0.01	93.9517%
3	1	93.6167%
4	10	92.5317%
5	0.001	92.4500%
6	100	91.9850%
7	1000	91.7867%

The Parameter of C that gave the best result on the training set was 0.1, with an accuracy of 94.3267%, an execution time of 5.3 minutes, and an accuracy on the test set of 94.28% .

Then, we will test a SVC with a polynomial kernel of degree 2. The different parameters tested are C, Coef0, which is the term r of the polynomial, ad gamma. Note that gamma is introduced by sklearn as a parameter to control the influence the two terms have on the polynomial:

$$K(x_1, x_2) = (\gamma \cdot x_1^T x_2 + r)^d$$

The three values of gamma tested are 1, *auto*, and *scale*, where $auto = 1/n\_features$ and $scale = 1/(n\_features * X.var())$. I personally believed that including all 64 results would have been redundant since we are only interested in the best possible result. Therefore, I have included the first and last 10 outcomes.

Table 2: SVC with Polynomial Kernel

	Paramter C	Coef0	Gamma	Mean Training Score
1	10	0.01	scale	98.0733%
2	1000	0.01	auto	98.0600%
3	0.001	0.5	1	98.0450%
4	0.001	1	1	98.0417%
5	0.001	0.01	1	98.0317%
6	1	1	1	98.0200%
7	0.1	1	1	98.0200%
8	1000	1	1	98.0200%
9	10	1	1	98.0200%
10	100	1	1	98.0200%
...	...	...	...	...
54	0.1	0.5	auto	89.9517%
55	0.001	1	scale	86.1300%
56	0.001	0.5	scale	83.1833%
57	0.01	1	auto	83.0033%
58	0.1	0.01	auto	73.8783%
59	0.01	0.5	auto	71.7483%
60	0.001	0.01	scale	71.2683%
61	0.01	0.01	auto	11.2150%
62	0.001	0.01	auto	11.2083%
63	0.001	0.01	auto	11.2083%
64	0.001	0.5	auto	11.2080%

Interestingly, apart from the first two results, the parameter $\gamma = 1$ consistently gave better results, and the value 0.001 gave some of the best results, while providing even the worse. Since the first four results were very close to each other, I decided to select the first four and perform the performance test on the test set for all four of them in order to choose the model with the best trade-off between training and test set. The result was the following:

Table 3: SVC with Polynomial Kernel, Training Score vs Test Score

C	Coef0	Gamma	Mean Training Score	Test Score
10	0.01	scale	98.0733%	97.87 %
1000	0.01	auto	98.0600%	97.83 %
0.001	0.5	1	98.0450%	97.97 %
0.001	1	1	98.0417%	97.99 %

The model which gave the best score on the training set was the last one, and since the gain was one order of magnitude bigger that the loss on the test set, the model with $C = 0.001$, $Coef_0 = 1$, $\gamma = 1$, and with a training time of 4.6 minutes, was the one chosen as the best.

Finally, we will use a SVC with a RBF Kernel, we have to perform grid search only with C and γ. Initially the test was performed with scale, auto and 1 as values of gamma. However, the value $\gamma = 1$ significantly slowed down the training process, even taking 340 minutes, and consistently yielded low accuracy. So the test was interrupted an ran again excluding that values; the results were the following:

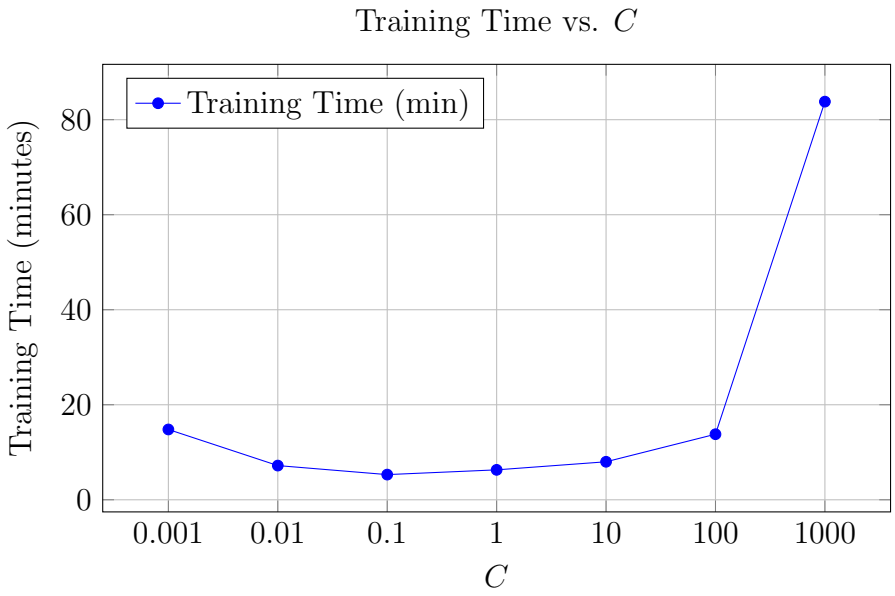
Table 4: SVC with RBF Kernel

	Paramter C	Gamma	Mean Training Score
1	10	scale	98.4083%
2	100	scale	98.3950%
3	1000	scale	98.3950%
4	1	scale	97.8650%
5	1000	auto	97.2683%
6	100	auto	97.2200%
7	10	auto	95.9617%
8	0.1	scale	95.6917%
9	1	auto	93.9083%
10	0.01	scale	91.6783%
11	0.1	auto	91.0367%
12	0.01	auto	82.2500%
13	0.001	scale	59.0833%
14	0.001	auto	11.2083%

With the best result being 98.4083%, obtained with the SVC using RBF as kernel, $C = 10$ and $\gamma = scale$. The training time took 6.1 minutes.

3.5. Execution Time

What appeared evident was that the execution time was heavily influenced by the value of C, which makes sense. In fact, with values of C very small, like 0.001, we encourage a larger margin, allowing more miss classifications. However, the optimization solver must process more slack variables to allow for these errors, making the optimization problem slower. A balanced value of C, such as 0.1 or 1, brings the focus of the optimization process on a smaller subset of important training points near the decision boundary, the support vectors, simplifying the optimization problem. Ultimately, as the values of C increase, we force the solver to fit as many points as possible, spending a lot of time on finding the best fit. The graph clearly shows this behavior, with a big jump for $C = 1000$.



IV. Random Forests

The next algorithm we will try to use to classify our handwritten digits is **Random Forests**. The key idea behind this kind of algorithms is to combine the prediction from multiple **Decision Trees** to obtain a more robust and acuate model. This approach is based on two key ideas: **Ensemble Learning**, and **Randomness** to ensure diversity among the trees.

4.1. Decision Trees

Complex decisions can often be expressed in terms of a series of questions. Let's suppose we want to decide what game to play today. We can first ask ourselves, do we want to play a FPS? If the answer is yes, we can ask again: do we want to get angry? If the answer is yes again, our choice would be VALORANT. Otherwise, we can play Battlefield. In general, a decision tree makes a statement, and takes a decision whether or not the statement is True or False. We can visualize the previous example in the following way:

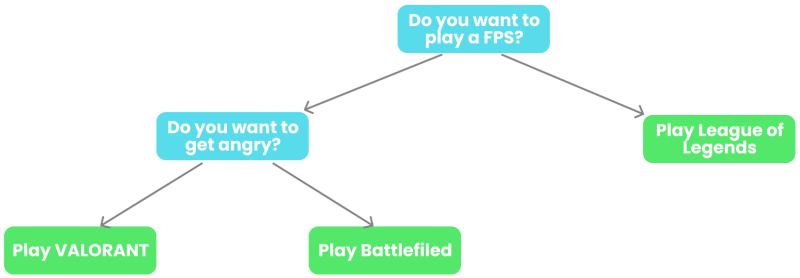


Figure 5: Example of Decison Tree

When a tree classifies objects into categories it is called a **Classification Tree**, while when it predicts numerical values it is called a **Regression Tree**. Observe that by convention the right branch of the tree corresponds to the value False, while the left brach is for True. Now, predicting the category of an object using a classification tree is quite easy: we just start from the root of the tree and work our way down until we end up on a leave. But how do we build a classification tree?

Starting from some data, the main question is which attribute to put in which position. A popular algorithm is ID3, which uses a measure called **Information Gain**, which is based on the notion of **Entropy** or **Impurity in the data**. Considering a set S of examples belonging to categories $c_1, ..., c_n$, p_i is the portion of examples belonging to c_i, then the entropy of S is:

$$Entropy(S) = -\sum_{i=1}^{n} p_i log_2(p_i)$$

Note that if p_i is near to 1, nearly all the examples are in this category, so p_i should have a low contribution to the whole entropy. This is indeed true because the closer p_i is to 1, the closer to 0 $log_2(p_i)$ gets. The inverse applies for values of p_i close to zero. In this case, very few examples are in category c_i, and similarly to the previous case, the contribution of the term to the entropy should be low. In fact, $log_2(p_i)$ becomes greater as p_i approaches 0, while never being able to dominate, so the overall term is close to 0. Let's now consider an attribute A, which can take values $\{v_1, ..., v_m\}$. Let S_v be the set of examples of S that take value v for A. The $Gain(S, A)$ is defined as:

$$Gain(S, A) = Entropy(S) - \sum_{v \in Values(A)} \frac{|S_v|}{|S|} Entropy(S_v)$$

We can now explain the ID3 algorithm in detail:

1. Firstly we calculate the gain $Gain(S, A)$ for each attribute A.
2. Then we choose as a root of the tree the attribute A with the highest Gain.
3. For each value v that A can assume, create a branch from the node and label it with the corresponding v.
4. Now, if:
    - If S_v contains only examples in category c, it means the attribute is sufficient for defining that category, so we put that category as a leaf node in the tree.
    - If S_v is empty, then find the default category (which contains the most examples from S and put it as a leaf node in the tree.
    - Otherwise we remove A from the attributes that can be chosen as a node, we replace S with S_v and after finding the attribute with the best Gain, we start again from step 2 [10].

Let's look at an example on how to build a decision tree using AC-3. Suppose we have the following data:

Table 5: Sample data of people playing and not playing VALORANT

Has a PC	Likes getting angry	Likes tactical FPS	Plays VALORANT
yes	yes	yes	yes
no	no	no	no
yes	yes	no	yes
yes	no	no	no

Firstly we need to calculate the entropy. Since $p_{yes} = \frac{1}{2}$ and $p_{no} = \frac{1}{2}$

$$E(S) = -\left(\frac{1}{2}\log_2\left(\frac{1}{2}\right) + \frac{1}{2}\log_2\left(\frac{1}{2}\right)\right) = 1$$

Now we calculate the information gain for each feature:

$$H(S_{\text{yes}}) = -\left(\frac{2}{3}\log_2\left(\frac{2}{3}\right) + \frac{1}{3}\log_2\left(\frac{1}{3}\right)\right) = 0.918$$

$$H(S_{\text{no}}) = -\left(\frac{0}{1}\log_2\left(\frac{0}{1}\right) + \frac{1}{1}\log_2\left(\frac{1}{1}\right)\right) = 0$$

$$Gain(\text{has a pc}) = 1 - \left(\frac{3}{4}\cdot 0.918 + \frac{1}{4}\cdot 0\right) = 0.311$$

And we do the same for the other attributes, obtaining $Gain$(Likes getting angry) = 1 and $Gain$(Likes tactical FPS) = 0.311. Since Likes "Likes getting angry" is the attribute with the highest "Gain" we choose it a a root for our tree, ans since both "yes" and "no" branches, contains only example in the respective categories, we set both as leaves, and we understand that the attribute "Likes getting angry" is enough for the classification the training set [11].

4.2. Advantages and disadvantages

Decision trees offers some clear advantages: small trees are easy to interpret, while scaling well to large n. They can also handle data of all types i.e. requiring little, if any, preprocessing. They can also handle missing data and they are completely nonparametric. But the biggest advantage is that they are extremely fast.

On the other hand, larger trees can be difficult to interpret and all splits depend on previous splits, i.e. they are very good at capturing feature interactions, while

performing bad when features are independent and their contribute is additive. Single trees have high variance, so they easily overfit the training data, but the most relevant disadvantage is that while decision trees can perfectly fit the training data, they are quite inaccurate when it come to new examples. To solve this issue we introduce **Random Forests**.

4.3. Random Forests

To quote from The Elements of Statistical Learning (aka The Bible of Machine Learning), "Trees have one aspect that prevents them from being the ideal tool for predictive learning, namely, inaccuracy." As we said, they perform well on the training set, while performing worse with new samples. Random Forests combine the simplicity of multiple decision trees, resulting in a vast improvement in accuracy.

Firstly, we need to create a **bootstrapped** dataset, which is a dataset constructed by randomly selecting samples from the original dataset, even allowing the same sample to be selected multiple times. This results in a dataset of the same size as the original. Next, we train a decision tree on the bootstrapped dataset, but at each step, we only consider a random subset of the attributes. We repeat this process by generating a new dataset each time until we have n trees. This process is called **Bagging**, and it ensures that we obtain a diverse range of trees, which is what provides statistical confidence to random forests and makes them more effective than single decision trees. It's worth noting that approximately $\frac{1}{3}$ of the data does not end up in the bootstrapped dataset, which is referred to as the **Out-Of-Bag Dataset**.

Now we are ready to use our forest for classification: we take our new sample and we run it through each tree, obtaining different results, and after this process, we simply take a majority vote to perform the classification. The strengths of random forests are their excellent performance, as we will see later, and the very little amount of tuning required. The **Out-Of-Bag Dataset** is used as a validation set so we don't have to sacrifice data for validation. Random forests are also robust to outliers, good at handling missing data, and hardly overfit the training set. Random forests come of course with their drawbacks: although accurate, they often cannot compete with advanced boosting algorithms, may become slow on large data sets, and their behavior is hard to interpret and predict [12].

4.4. Handwritten digit recognition using Random Forests

As we did for SVMs we will again use the *sklearn* library. We can easily create and use a random forest in the following way:

```
rndfor = RandomForestClassifier(criterion='entropy', max_depth=...,
↪  n_estimators=...)
rndfor.fit(X_train, y_train)
rndfor.score(X_test, y_test)
```

In this case, the only two parameters we will tune are *max_depth* and *number_of_estimators*. The first value determines how deeply we can grow a tree: if we

reach the determined depth and we haven't found a pure set, we stop splitting the branch. The number of estimators is quite straightforwardly the number of trees in the forest. It is also worth briefly mentioning the **Gini's Impurity**, which is another popular method to determine the impurity of the tree nodes.

We perform Grid Search as follows:

```
params = {
    'n_estimators': [250],
    'max_depth': [20]
}

gs = GridSearchCV(rndfor, params, n_jobs=-1, cv=10,
↪  scoring='accuracy', verbose=5)
gs.fit(X_train, y_train)
```

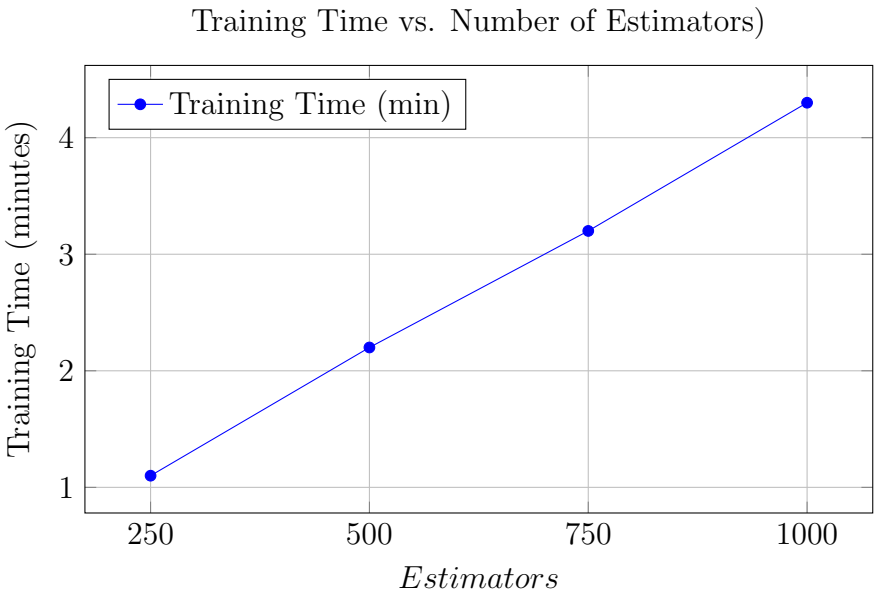
Obtaining the following result:

Table 6: Random Forests

	Max Depth	Number of Estimators	Mean Training Score
1	40	750	97.891%
2	40	250	97.781%
3	60	250	97.384%
4	40	1000	97.142%
5	60	1000	97.134 %
6	100	750	97.120%
7	100	1000	97.120%
8	80	1000	97.115%
9	60	750	97.099%
10	100	500	97.081%
11	80	250	97.071%
12	80	500	97.071%
13	80	750	97.071%
14	20	500	97.067%
15	60	500	97.060%
16	100	250	97.054%
17	40	500	97.026%
18	20	250	96.991%
19	20	750	96.991%
20	20	1000	96.988%

The results are very close to each other. The configuration which gave the best result was the one with *Max Depth = 40* and *Number of Estimators = 750*, with an accuracy on the training set of 97.891%, a training time of 3.4 minutes and a test set accuracy of 96.7%.

Finally, as we would expect, the execution time grows as the number of estimator grows, while the max depth seems to have little to no effect on the performances.



V. Naive Bayes

So far, we've been talking about learning algorithms that model the conditional distribution of output y given the input x. We will now focus on a different approach for, based on **Statistical Leaning Theory**.

5.1. Statistical Learning Theory

Supervised learning involves learning from a training set of data, where each point consists of an input and an output. The learning problem involves inferring the function that maps the input to the output, enabling the learned function to predict future outputs from new inputs. More formally, given an input space X, and an output space Y, for simplicity let $Y = \{-1, +1\}$, we want to find the relationship f between the input and output space:

$$f : X \to Y$$

and such a mapping f is called a **classifier**. In order to do this, a **classification algorithm** takes as input a training set:

$$S = \{(x_1, y_1), ..., (x_n, y_n)\} \in X \times Y$$

and outputs a classifier f. In *Statistical Learning Theory* we try to estimate the functional relationship between the input and output spaces: we make the assumption that here exists a joint probability distribution P on $X \times Y$,

$$P(x, y) = Prob\{X = x, Y = y\}$$

Note that we don't make any assumption on P, and it's distribution is unknown at the time of learning, but we assume P is fixed. We need a way to determine the effectiveness of the function f when used as *classifier*. A **Loss Function** measures the "cost" of classifying objects, the simplest *loss function* we can imagine is the **0-1** loss, defined as follows:

$$\mathscr{L}(X, Y, f(X)) = \begin{cases} 0 & if \quad f(X) \neq Y \\ 1 & otherwise \end{cases} \tag{6}$$

We also define the **Risk** of a function as the average loss over data points generated according to the underlying distribution P:

$$\begin{aligned} R(f) &:= E[\mathscr{L}(X, Y, f(X))] \\ &= \sum_{x \in X} \sum_{x \in Y} P(x, y) \mathscr{L}(X, Y, f(X)) \end{aligned}$$

The optimal classifier is the one with the lowest risk $R(f)$. However, we already said that we don't know P. Let's momentarily assume that an oracle provides us with P: in this situation the best possible classifier is the **Bayes Classifier**:

$$f_{Bayes}(x) = \begin{cases} 1 & if \quad P(Y = 1, X = x) \geq \frac{1}{2} \\ -1 & otherwise \end{cases} \tag{7}$$

In practice, it is impossible to directly compute the Bayes classifier because, as we already pointed out, P is unknown to the learner [13].

5.2. Naive Bayes Classifier

The **Discriminative Approach** used in *SVMs* and *Random Forests*, tries to directly model the probability $P(y|x)$. The so called **Generative Approach**, models $P(Y|X)$ using the **Bayes Theorem**:

$$P(Y|X) = \frac{P(X|Y)P(Y)}{P(X)} = \frac{P(X,Y)}{P(X)}$$

The main idea behind **Naive Bayes Classifiers** is the assumption that the value of a particular feature is independent from the value of any other feature. For example, let's suppose once again we want to classify fruits: a fruit may be considered to be an orange if it is orange, round, and about 10 cm in diameter. A naive Bayes classifier considers each of these features to contribute independently to the probability that this fruit is an orange, regardless of any possible correlations between the color, roundness, and diameter features.
Under this assumptions we have [1]:

$$\begin{aligned} P(x_1, ..., x_n|y) &= P(x_1|y)P(x_2|y)...p(x_n|y) \\ &= \prod_{i=1}^{n} P(x_n|y) \end{aligned}$$

Now, we will first explain the general idea behind the Naive Bayes algorithm using a toy example, and then we will explore the actual implementation for our use case. Imagine we want to predict if a person likes the Harry Potter Saga or not. Firstly we collect data from people who loves and does not loves Harry potter, lets say we are interested in 3 metrics: their age, the number of books read in a year and the fantasy movie watched in their entire lifetime. Each of this metric will have a different probability distribution for the two classes of peoples. We now want to classify a person never seen before, suppose a 23 years old, which read 6 books a year and have watched a total of 17 fantasy films. We firstly make an initial guess on the probability that he likes Harry Potter, and we call it $P(Like)$. Such guess can be based on the training data; then we do the same thing for the category "Doesn't likes Harry Potter" ($P(\neg Like)$).Those initial guess are called **Prior Probabilities**. Now, the probability that the new person loves Harry potter is:

$$\begin{aligned} P(LikeHP) = P(Like) &\cdot P(Age = 23|LikeHP) \\ &\cdot P(Books = 6|LikeHP) \\ &\cdot P(Movies = 17|LikeHP) \end{aligned}$$

Since some probabilities might be very small, we take the natural logarithm to avoir *underflow*, which result in:

$$\begin{aligned} P(LikeHP) = P(Like) &+ log[P(Age = 23|LikeHP)] \\ &+ log[P(Books = 6|LikeHP)] \\ &+ log[P(Movies = 17|LikeHP)] \end{aligned}$$

Similarly the probability that the person does not likes harry Potter is:

$$\begin{aligned} P(\neg LikeHP) = P(\neg Like) &+ log[P(Age = 23|\neg LikeHP)] \\ &+ log[P(Books = 6|\neg LikeHP)] \\ &+ log[P(Movies = 17|\neg LikeHP)] \end{aligned}$$

And we are done, we simply predict the class which have the highest probability.

5.3. Handwritten digit recognition with Naive Bayes

Let's now implement the intuition we just saw for a practical use. Once again our problem is handwritten digit recognition, but this time we will abandon sklearn, and implement Naive Bayes from scratch. We will make use of some useful libraries to simplify our work and greatly increase the performances, in particular **scipy**, **numpy** and **math**.

We start by creating our class, so a "NaiveBayes" object can be instantiated. When an object is created, we initialize a property called $alpha\_beta\_matrix = []$ which will hold the probability distributions for each pixel and for each digit, resulting in a 10x748 matrix. Next we need the **fit** method, which takes as input the training set with the corresponding labels, and trains the model. We recall that we are dealing with pandas *DataFrame*. Although very useful, the performances when operating with a large amount of data are not so great, so we convert them to numpy series. We start by iterating the whole dataset, and initializing each cell of the $alpha\_beta\_matrix$ with all the values the pixel $p_j, j = 0, ..., 783$ assumes when the image it belongs is the digit $d_i, i = 0, ..., 9$.

```
for label in self.unique_labels:
    label_mask = y_array == label
    filtered_data = X_array[label_mask]
    alpha_beta_matrix_aux = [filtered_data[:, col].tolist() for col in
    ↪  range(X_array.shape[1])]
    self.alpha_beta_matrix.append(alpha_beta_matrix_aux)
```

Then, assuming the pixels follows a **Beta distribution**, we calculate the distribution of such values. The beta distribution is defined as follows:

$$f(x;\alpha,\beta) = \frac{x^{\alpha-1}(1-x)^{\beta-1}}{\mathrm{B}(\alpha,\beta)}, \quad 0 \leq x \leq 1,\ \alpha > 0,\ \beta > 0$$

with $\mathrm{B}(\alpha,\beta)$ being:

$$\mathrm{B}(\alpha,\beta) = \int_0^1 t^{\alpha-1}(1-t)^{\beta-1}\,dt$$

It can be used to model a random phenomenon, which can assume the values [0,1]. if $\alpha = \beta$, the density of the distribution is symmetric around $\frac{1}{2}$, with the regions around $\frac{1}{2}$ having more weight as α become bigger. if $\beta > \alpha$, the density is crushed to the left, meaning the smaller values are more likely; otherwise, the distribution is moved to the right if $\alpha > \beta$ [14].

Now, for each cell of our *alpha_beta_matrix* we save a dictionary containing the values of α and β:

```
for i, row in enumerate(self.alpha_beta_matrix):
    for j, col in enumerate(row):
        mean = np.mean(col)
        var = np.var(col) + sys.float_info.min
        k = (mean * (1 - mean)) / var

        a = k * mean
        b = k * (1 - mean)

        self.alpha_beta_matrix[i][j] = {"alpha": a, "beta": b}
```

For the sake of simplicity, we will use as values of α and β:

$$\alpha = KE[X] \qquad \beta = K(1-E[X])$$

with

$$K = \frac{E[X](1-E[X])}{Var(X)}$$

Interestingly, we can plot the values of α into 28x28 images to visualize what the algorithm is "learning":

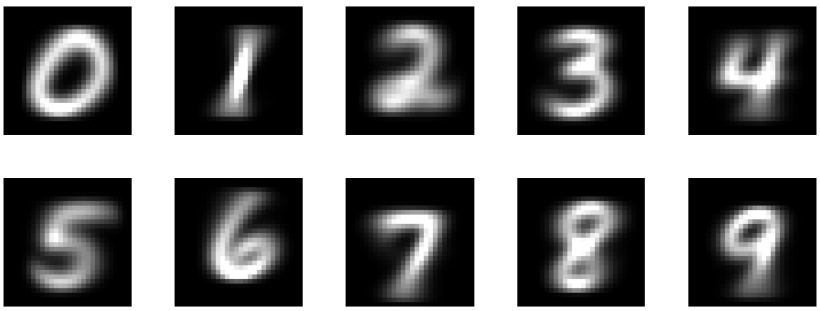


Figure 6: Plot of the values of α

Ultimately, we implement the **predict** method, which does exactly what we described in the theory section:

```
def predict(self, X):
    X_array = X.to_numpy() if hasattr(X, 'to_numpy') else np.array(X)
    predictions = []

    for i, row in enumerate(X_array):
        predictions.append(self.__score_row(row))

    return np.array(predictions)
```

We score each row of the test set by iterating through each possible digit and by using the corresponding values of α and β previously learned to calculate the likelihood of each pixel. Then the digit corresponding to the highest value is returned. The training time of the model took just 2.66 second, which is blazing fast! The evaluation on the test set took 26.62 seconds, however, the accuracy was a disappointing 72.35%, probably mainly due to the approximations made on the α and β parameters.

VI. K-NN

"The intuition underlying Nearest Neighbor Classification is quite straightfor- ward, examples are classified based on the class of their nearest neighbors. It is often useful to take more than one neighbour into account so the technique is more commonly referred to as k-Nearest Neighbor (k-NN) Classification where k nearest neighbors are used in determining the class."

6.1. Nearest Neighbor

The **Nearest Neighbor** is perhaps the simplest and straightforward algorithm we can think about, so simple that it is even debatable if considering it an actual "learning" algorithm. As we saw, an object can be represented as a vector and each vector has a position in a vector space. We memorize the whole training set, and we calculate the distance between the vector to be classified and each vector in the training set: the lable of the nearest vector would be assigned as a label of the vector to classify. The k-NN version simply looks at the k nearest vectors, and takes a majority vote.

This the algorithm works because of an interesting result, lets resume for a moment our discussion regarding statistical *learning theory*: we stated that the *Bayes Classifier* is the best we can do, therefore R_{Bayes} is our lower bound. Being R_∞ the Risk of k-NN when the number of features tends to infinity, it can be proven that:

$$R_{Bayes} \leq R_\infty \leq 2R_{Bayes}$$

which is a great result [13].

6.2. Handwritten digit recognition with K-NN

As we did for Naive Bayes, we will also implement k-NN by ourself, let's take a look at the code. Firstly we create our class, the constructor creates two variables to hold the training data with the corresponding labels, and a variable to hold the value of k, setted to 1 by default. Then we implement the **fit** method, which simply saves the training data into the corresponding variables. The **predict** method is implemented as follows:

```
def predict(self, X):
    X_test = X.to_numpy() if hasattr(X, 'to_numpy') else np.array(X)
    distances = cdist(X_test, self.X_train)

    k_indices = distances.argsort(axis=1)[:, :self.k]

    predictions = []
    for indices in k_indices:
        k_nearest_labels = self.y_train[indices]

        counter = Counter(k_nearest_labels)
        predictions.append(counter.most_common(1)[0][0])

    return np.array(predictions)
```

as we did before, the DataFrames are converted to numpy series, for better performances. Then we calculate the distances between the training set and the image

to classify. *distances* is a 2D NumPy array where each element represents the pairwise distance between a sample in X_test and a sample in X_train. By default the function uses the **Euclidean distance**, defined as follows

$$d(\mathbf{p}, \mathbf{q}) = \|p - q\|_2$$

and where p and q are two points in vector space of dimension n and $\| \bullet \|$ is the **Euclidean Norm**. Then we use *numpy.argsort* to sort the indices of the distances in ascending order for each row, and then we select the indices of the k-nearest neighbors. Finally, we retrieve the k-nearest neighbors labels and then we take the majority vote.

The only parameter to be tuned is k: an odd value is preferable since it reduced the probability of a tie. In this case an eventual tie is broken by the order of the elements, but it would be preferable to implement a way to solve such eventuality. The test using cross validation gave the flowing result:

Table 7: K vs Training Score

	Value of K	Mean Training Score
1	3	97.300%
2	1	97.220%
3	5	97.190%
4	7	97.000%
5	9	96.830%

The best value of K was 3, with an accuracy of 97.300% and an execution time of 271 seconds. Obviously the training phase is almost instant, since we simply store the training set, the obvious drawback being the high amount of memory we need to use.

VII. Conclusions

SVMs provided an effective method for solving our problem, while also providing solid theoretical basis. The drawback represented was the performance: finding a linear bound implies analyzing the relationship between every pair of points, which scales badly with large datasets.

Random Forests provided also an effective method for solving handwritten digits recognition, while also providing good performances. On the other hand, the theory behind them is not as solid and intuitive as some other models, such as Support Vector Machines.

Naive Bayes mainly relies on statistical learning theory: since we can't exactly know the distribution of the data, reaching satisfying results requires a lot of analysis. On the bright side, the performances, compared to the previous two methods, are excellent.

K-NN is the simplest algorithm we can implement, and seems to be the perfect compromise between the previous classifiers, with incredible performances, equally good accuracy, and a great theoretical basis. The drawback is the large amount of memory needed to memorize the whole dataset, which becomes impractical for today's applications.

In conclusion, each of the models comes with its advantages and disadvantages, and the choice between one or another must be well thought depending on the use case.

References

[1] Generative Models. Local search. University Lecture, 2024.

[2] Christopher J.C. Burges Yann LeCun, Corinna Cortes. The mnist database of handwritten digits. https://yann.lecun.com/exdb/mnist/.

[3] Joshua Starmer. Support vector machines part 1 (of 3): Main ideas!!! https://www.youtube.com/watch?v=efR1C6CvhmE.

[4] Yao Xiao Arturo Amor, Lucy Liu. scikit-learn. https://scikit-learn.org/1.5/index.html.

[5] Google. Numerical data: Normalization. https://developers.google.com/machine-learning/crash-course/numerical-data/normalization.

[6] Sathvik Chiramana. Svm dual formulation. https://medium.com/@sathvikchiramana/svm-dual-formulation-7535caa84f17.

[7] Andrea Torsello. Linear classiers. University Lecture, 2024.

[8] Joshua Starmer. Support vector machines part 2: The polynomial kernel (part 2 of 3). https://www.youtube.com/watch?v=Toet3EiSFcM&t=211s.

[9] Joshua Starmer. Support vector machines part 3: The radial (rbf) kernel (part 3 of 3). https://www.youtube.com/watch?v=Qc5IyLW_hns.

[10] Andrea Torsello. Decision trees. University Lecture, 2024.

[11] Joshua Starmer. Decision and classification trees, clearly explained!!! https://www.youtube.com/watch?v=_L39rN6gz7Y.

[12] Joshua Starmer. Statquest: Random forests part 1 - building, using and evaluating. https://www.youtube.com/watch?v=J4Wdy0Wc_xQ.

[13] Marcello Pelillo. Statistical learning theory. University Lecture, 2024.

[14] S.M. Ross. *Calcolo delle probabilità*. Idee & strumenti. Apogeo, 2004.