Ca’ Foscari
University
Venezia

[CM0623-2] FOUNDATIONS OF ARTIFICIAL INTELLIGENCE (CM90)

Discriminative and Generative Classifiers

Student
Serena Zanon
Matricola 887050

Academic Year
2023 / 2024

Indice

Introduction

1.1 The assignment

The proposed assignment asks to train and test four different algorithms over the MNIST database.
Following, here is the assignment request:

Write a handwritten digit classifier for the MNIST database. These are composed of 70000 28x28 pixel gray-scale images of handwritten digits divided into 60000 training set and 10000 test set. In python you can automatically fetch the dataset from the net and load it using the following code:

```
from sklearn.datasets import fetch\_openml
X,y = fetch\_openml('mnist\_784', version=1, return\_X\_y=True)
y = y.astype(int)
X = X/255.
```

This will result in 784-dimensional feature vectors (28*28) of values between 0 (white) and 1 (black). Train the following classifiers on the dataset:

1. SVM using linear, polynomial of degree 2, and RBF kernels;

2. Random forest;

3. Naive Bayes classifier where each pixel is distributed according to a Beta distribution of parameters α, β:
   $d(x; a, b) = \frac{\gamma(\alpha+\beta)}{\gamma(\alpha)+\gamma(\beta)} x^{\alpha-1}(1-x)^{\beta-1}$
4. k-NN

You can use scikit-learn or any other library for SVM and random forests, but you must implement the Naive Bayes and k-NN classifiers yourself.
Use 10 way cross validation to optimize the parameters for each classifier.
Provide the code, the models on the training set, and the respective performances in testing and in 10 way cross validation.
Explain the differences between the models, both in terms of classification performance, and in terms of computational requirements (timings) in training and in prediction.
P.S. For a discussion about maximum likelihood for the parameters of a beta distribution you can look here. However, for this assignment the estimators obtained with he moments approach will be fine:
$\alpha = KE[x]$
$\beta = K(1 - E[X])$
with $K = \frac{E[X](1-E[X]}{Var(x)} - 1$
Note: $\frac{\alpha}{\alpha+\beta}$ is the mean of the beta distribution. If you compute the mean for each of the 784 models and reshape them into 28x28 images you can have a visual indication of what the model is learning.

1.2 The MNIST dataset and the project structure

The proposed dataset is composed of 70.000 instances divided into training set and test set, respectively with 60.000 instances and 10.000 instances. A single instance is a 28x28 pixel image so the number of the dataset's columns is 784.
In order to manage, in the best way possible, the dataset provided, some changes have been made: all columns characterized by only one value, meaning that its minimum value and its maximum value coincides, are discarded. The reason for making such a decision lies in the fact that, since they have only one value, they cannot help the classifier to discriminate instances from one class and another, so they are useless.
As well as the training set and the test set, another smaller training set is created: it is composed of only 15.000 instances retrieved from the original training set and it is used to choose the best value of the models' hyper-parameters. This was deemed necessary due to the huge dimension of the original dataset which could lead to a definitely too high computational time.
All of the created sets are obtained with stratified sampling technique, so the proportion of each class in the provided dataset is maintained in all of the sets; then, they are transformed into *csv* files in order to be able to import them directly without repeating the same procedure for every single model.
A single model is trained and tested in its own notebook file, in which can be found the same approach used for all of them:

- If the model is characterized by an hyper-parameter to which is important to find a good value in order to have an high accuracy, the *GridSearchCV* method is applied.

The principal characteristic of this method is that it's possible to configure some interesting parameters:

  - The *param_grid* parameter is used to specify the possible values that the model's hyper-parameter can take on;
  - The *scoring* parameter indicates the measure through which the model is evaluated (in this case *scoring* = *accuracy*);
  - The *cv* parameter allows to exploit the cross validation technique by assign an integer value (the value corresponds to the number of folds in which the dataset is divided; in this case *cv* = 10);
  - The *n_jobs* parameter specify the number of jobs to run in parallel;
  - Finally, the *verbose* parameter (if it is set to a value higher than 1) shows messages during the computation.

  This method requires a lot of time because it has to compare the model as many times as the number of the values in *param_grid* times the value of *cv*.

- The best model obtained from the previous step is fitted with the whole training set (the one with 60.000 instances);
- As last step, the *predict* method is computed over the test set and the final accuracy is retrieved.

Supervised Learning

2.1 What is supervised learning?

Supervised learning is a specific subset of machine learning and artificial intelligence. [6]
It is characterized by the usage of labeled datasets in order to train algorithms whose aim is to classify or predict correctly the given instances. [7]
Typically, some data is given to the model and, during the learning phase, its weights are modified with the intention to achieve the highest accuracy possible.
Supervised learning models are divided in two different categories:

- Classification models:
- Regression models.

Classification algorithms try to identify the correct category (or class) of the input object and these kinds of algorithms are called classifiers.
On the other hand, regression algorithms are expected to identify a numerical relationship between input data and output data.
Based on the purpose of the proposed project, in this report only the classification task is described in detail.

2.2 Discriminative and generative classifiers

The real basis of all possible existing models is mathematics, especially some of its branches like calculus, probability and statistics, and it's possible to identify two groups of algorithms: [1]

- Discriminative models;
- Generative models.

Discriminative models are also called conditional models and, during the training phase, they try to separate points in different categories and identify their boundaries using probability estimates.
They learn according to the conditional probability $P(y|x)$ which is computed directly by the algorithm and can be expressed as the probability that the label is equal to *y* knowing that the chosen sample is *x*. [8]
This means that, when the classifier sees data never seen before, it tries to find the best possible side of the learnt decision boundaries in such a way that the above probability is maximized.
Now, the problem can be summarized as

$$\hat{y} = \underset{y}{\operatorname{argmax}} P(y|x).$$

SVM and Random Forest models are discriminative classifiers.

Instead, generative models rely on the joint probability distribution $P(X, Y)$ where *Y* is the set of all possible labels and *X* is the input dataset.
From this distribution, the model computes the conditional probability $P(x|y)$, which can be expressed as the probability that the element is *x* given that the label is equal to *y*, and the probability $P(y)$, which refers to the probability of observing the label *y* in the dataset. [9]
This computation can be done with the support of the Bayes' Theorem

$$P(y|x) = \frac{P(x|y)P(y)}{P(x)}.$$

The $P(x)$, so the probability of observing the element *x* in the dataset, can be ignored since it doesn't depend on *y*.

Thanks to this observation, the problem from this form

$$\hat{y} = \underset{y}{\operatorname{argmax}} P(y|x)$$

Which is equal to

$$\hat{y} = \underset{y}{\operatorname{argmax}} \frac{P(x|y)P(y)}{P(x)}$$

Can be written as

$$\hat{y} = \underset{y}{\operatorname{argmax}} P(x|y)P(y).$$

$P(y|x)$ is the posterior probability and $P(y)$ is the prior probability since it doesn't take into consideration the variable *X*. *k*-NN and Naive Bayes models are generative classifiers.

Support vector machine model

3.1 Theory behind Support Vector Machine

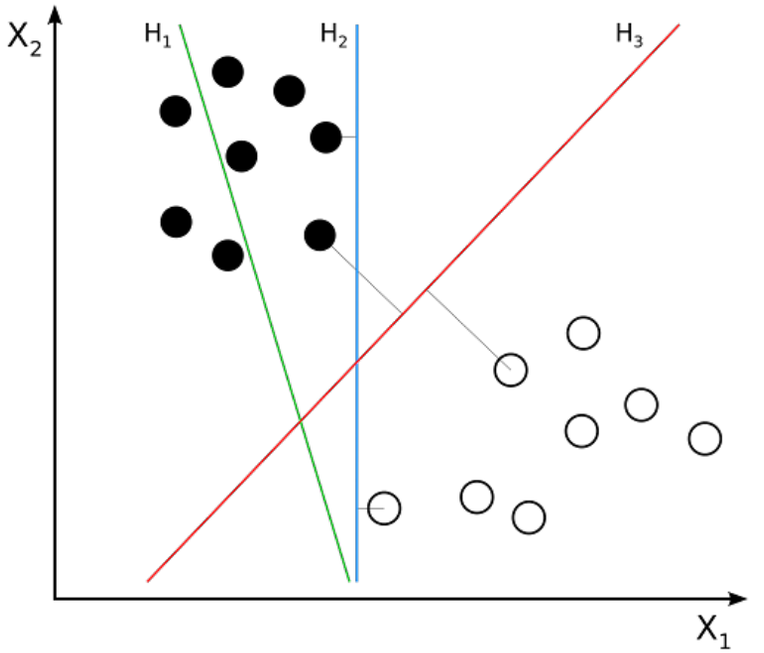
Support vector machines (SVM) are models belonging to the supervised learning field of machine learning and they are used for both classification and regression analysis.

As written before, let's focus only on the classification analysis.

In this context, SVM can be defined as a non-probabilistic binary linear classifier: given a dataset with only two possible classes (binary), the algorithm projects each point on the feature space and tries to completely separate the two categories identifying an hyperplane (linear) that divides them perfectly.

Doing so, when new data are presented to the model, each new element is portrayed on the feature space and the category to which it belongs is the one on which it falls (non-probabilistic).

The goal of the model is to find the best possible hyperplane which maximizes the gap between the two classes.[13]



In the figure above is represented a two-dimension training set and three different hyperplanes:

- H1 and H2 are not the desired output:
  - H1 doesn't separate completely the two categories;
  - H2 does separates the two categories but doesn't maximize their distance.
- H3 is the desired output since it separates perfectly the two categories and maximizes their gap.

Maximizing the gap between the two classes is the most crucial point of the algorithm: if some perturbation is applied to the data, the predictions made by the model both before and after adding the noise cannot be different and this is achieved by maximizing the distance between one class and the other.

3.1.1 Linear SVM and linearly separable problems

Given a training set of *n* points represented as $(x_1, y_1), \cdots, (x_n, y_n)$ where y_i can be either equal to 1 or -1 and each x_i is a point in p-dimension space, the goal is to find the best hyperplane that separates class 1 ($y_i = 1$)

and class -1($y_i = -1$).

The hyperplane can be defined by the equation $w^T x - b = 0$: ***w*** is the orthogonal vector to the hyperplane and the parameter $\frac{b}{||w||}$ determines the hyperplane offset from the origin.

If it is actually possible to linearly separate the points of the dataset, in other words, if there exists an hyperplane capable of classifying correctly the whole dataset, then it's possible to identify other two parallel hyperplanes that divide the two classes in a way that their distance is the maximum possible.

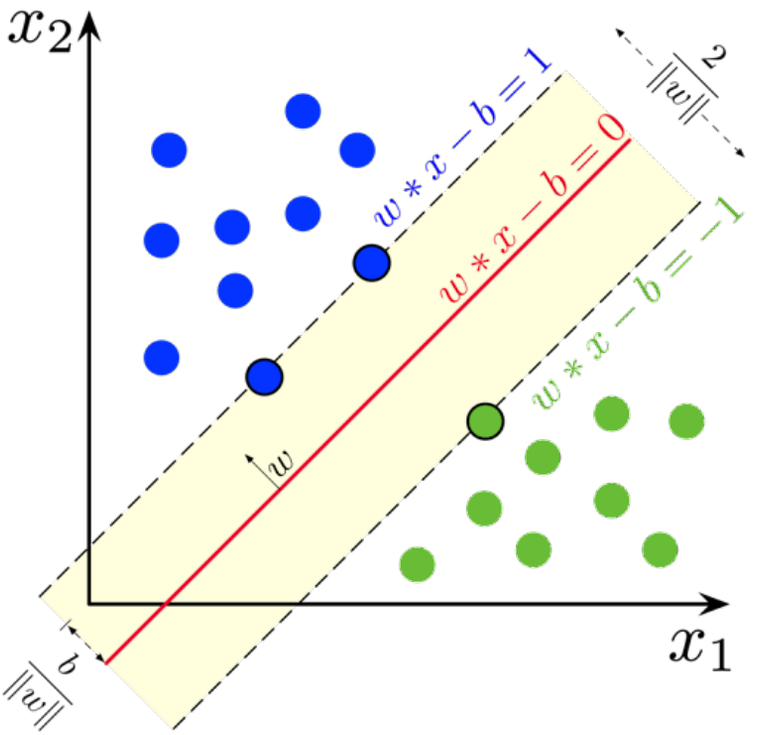
The region within these two new hyperplanes is called margin and the goal of the model is exactly to find the halfway hyperplane between them. [15]

The equations of the two new hyperplanes are:

- $w^T x - b = 1$

- $w^T x - b = -1$

And, geometrically speaking, the width of the margin, which is the distance between these new hyperplanes, is equal to $\frac{2}{||w||}$, in this way the problem can be simply reduced to minimize $||w||$.



Therefore, each point must lie on the correct side of the margin:

- $w^T x_i - b \geq 1, if y_i = 1$
- $w^T x_i - b \leq -1, if y_i = -1$

These constraints can be rewritten in a more compact form:

$y_i(w^T x_i - b) \geq 1, \forall 1 \leq i \leq n$

At this point, it is possible to formulate better the problem that the model solves:

$$\max_w \frac{2}{||w||}$$

With constraints

$$\begin{cases} w^T x_i - b \geq 1, if y_i = 1 \\ w^T x_i - b \leq -1, if y_i = -1 \end{cases} for i = 1, \cdots, n$$

This problem can be formulated also as a minimization problem:

$$\min_w \frac{1}{2}||w||^2$$

With constraints

$$y_i(w^T x_i - b) \geq 1, \forall 1 \leq i \leq n$$

As it can be noticed, this problem is constrained but it can be transformed into an unconstrained one by simply adding *n* Lagrangian multipliers:

$$\min_{w} L(w, b, \Lambda) = \frac{1}{2}||w||^2 - \sum_{i=1}^{n} \lambda_i [y_i(w^T x_i + b) - 1]$$
$$\forall i = 1, \cdots, n \text{ and } \Lambda = (\lambda_i, \cdots, \lambda_n) \geq 0$$

If the imposition of partial derivatives equal to 0 is applied, the following is obtained:

$$\frac{\partial L(w,b,\Lambda)}{\partial w} \text{ is } w = \sum_{i=1}^{n} \lambda_i y_i x_i$$
$$\frac{\partial L(w,b,\Lambda}{\partial b} \text{ is } \sum_{i=1}^{n} \lambda_i y_i = 0$$

Assuming so, the problem now becomes:

$$L(\lambda_i, \cdots, \lambda_n) = \sum_{i=1}^{n} \lambda_i - \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} \lambda_i \lambda_j y_i y_j x_i^T x_j$$

With constraints

$$\begin{cases} \sum_{i=1}^{n} \lambda_i y_i = 0 \\ \lambda_i \geq 0, \forall i = 1, \cdots, n \end{cases}$$

If $\lambda_i = 0$ then the constraint is satisfied without any distortion; on the other hand, if $\lambda_i > 0$ then the constraint is satisfied producing some distortion. The points that fall under this last case are called support vectors.
The achieved problem is convex meaning that there exists an optimal solution and given an unknown vector μ, the prediction of its class (1 or -1) can be done following the below function:

$$h(\mu) = sgn(\sum_{i=1}^{k} \lambda_i y_i x_i^T \mu)$$

The sum is over k support vectors.

3.1.2 SVM and non-linearly separable problems

SVM can be also used to solve non-linearly separable problems and to do so there are two approaches:

- Soft-margin approach;

- Mercer's Kernel approach.

The Soft-margin approach relies on the hinge loss function which is defined as follows:

$$max(0, 1 - y_i(w^T x_i - b)).$$

Where y_i is the actual label of the element x_i while $(w^T x_i - b)$ is the output given by the model.
If the function returns 0, then the constraint $y_i(w^T x_i - b) \geq 1$ is satisfied, which means that the element x_i lies on the correct side of the margin.
On the other hand, if the function does not return 0, then its output is proportional to the distance between x_i and the margin itself.
Due to this, the problem is formulated in a different way than in the linear case:

$$\min_w \lambda ||w||^2 + [\tfrac{1}{n} \textstyle\sum_{i=1}^{n} max(0, 1 - y_i(w^T x_i - b))].$$

The parameter λ must be greater than 0 and manages the trade-off between increasing the width of the margin and ensuring that each x_i lies on the correct side.
Formalizing the problem, the following is obtained:

$$\min_w \tfrac{1}{2} ||w||^2 + C \textstyle\sum_{i=1}^{n} \xi_i$$

With constraints

$$\begin{cases} y_i(w^T x_i + b) \geq 1 - \xi_i \\ \xi_i \geq 0, \forall i = 1, \cdots, n \end{cases}$$

Where

- Each $\xi_i = max(0, 1 - y_i(w^T x_i - b))$ variable is called slack variable and they are as many as the number of items composing the dataset;

- C is an hyper-parameter which controls the trade-off between the accuracy of the models and the maximization of the margin; in

other words, it tells how much flexible the algorithm is to accept misclassification:

- If C is a small value, then the model is considered flexible enough to easily permit to violate the classification;
- If C is a large value, then the model is strict and doesn't easily permit misclassification;
- If C tends to infinity, the problem becomes like the original one where each slack variable is equal to 0.

The Lagrangian (dual) representation of this problem is as follows:

$$\max_{\Lambda} L(w, b, \Lambda) = \sum_{i=1}^{n} -\frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} \lambda_i \lambda_j y_i y_j x_i^T x_j$$

With constraints

$$\begin{cases} \sum_{i=1}^{n} \lambda_i y_i = 0 \\ 0 \leq \lambda_i \leq C, \forall i = 1, \cdots, n \end{cases}$$

It is very similar to the one seen before except for the adding upper bound constraint for each λ_i.

The other possible approach is using Mercer's kernel which rely on Mercer's Theorem:

Let K : X × X $\rightarrow \mathbb{R}$ be a positive-definite function, then there exist a (possibly infinite-dimensional) vector space Y and a function ϕ : X $\rightarrow$ Y such that $K(x_1, x_2) = \phi(x_i)\phi(x_2)$.

This theorem states that it's possible to substitute the dot-product with a function K, called kernel, and implicitly a non-linear mapping to a high-dimensional space is defined.
The logic under this approach is to transform the original features values in a non-linear way obtaining a linearly separable problem. The data are mapped through the $\phi(-)$ function.

Typically, the transformation is from a lower dimensional feature space to a higher one.

From what it has been seen so far, it has to be noticed that in the dual representation the input data never appear alone but only in a dot-product and this justify the declaration: $K(x_1, x_2) = \phi(x_1)\phi(x_2)$.

There are different kind of kernels and some of them are:

- Linear kernel;

  $K(x_i, x_j) = x_i * x_j$: it's the common dot-product.

- Polynomial kernel;

  $K(x_i, x_j) = (1 + x_i^T * x_j)^n$: with n ¿ 0, it represents the dot-product over polynomials of the original variables. The typical choice is n = 2 because numbers greater than that tend to lead the model to overfit data.

- Radius Basis Function.

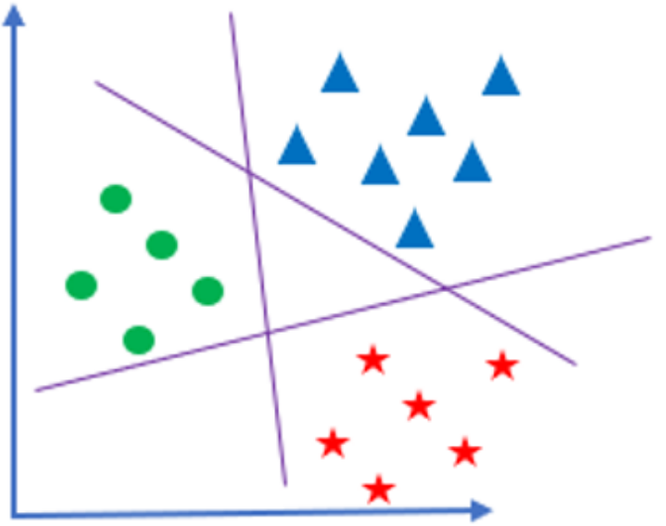
  $K(x_i, x_j) = exp(-\frac{1}{2}\frac{||x_i - x_j||^2}{\sigma^2})$: it represents the dot-product of Gaussian bumps centered on support vectors.

3.1.3 SVM and multiple classes

Obviously, not all datasets have only two categories, the majority of them have more than two classes.

In this case, the binary classification framework can help.

Indeed, it's possible to build a model for each class that discriminates between the class itself and the rest of the categories. However, there might be a zone belonging to more classes: the model simply trusts the highest score on that particular point within all the classifiers.



3.2 Implementation and results

Both training and testing phases for the SVM classifier are contained in three different Python notebooks, one for each kernel requested:

- *2a_SVC.ipynb* contains the support vector machine model with the linear kernel;
- *2b_SVC.ipynb* contains the support vector machine model with the polynomial kernel (degree = 2);
- *2c_SVC.ipynb* contains the support vector machine model with the RBF kernel.

For each of these versions, the tuning over the hyper-parameter C is computed testing these values $C = [0.01, 0.1, 1.0, 10.0, 100.0]$.
Below, there are three tables, one for each kernel, and all of them are composed by two columns:

- The first column represents the parameter C and the values it can take;
- The second column shows the accuracy on the smaller training set as the value C changes.

First version (linear kernel)

Value of C	Accuracy
0.01	0.929200
0.1	0.933600
1.0	0.919867
10.0	0.912200
100.0	0.911467

Second version (polynomial kernel, degree equal to 2)

Value of C	Accuracy
0.01	0.838800
0.1	0.932000
1.0	0.961667
10.0	0.966667
100.0	0.966000

Third version (RBF kernel)

Value of C	Accuracy
0.01	0.851933
0.1	0.936667
1.0	0.965333
10.0	0.971600
100.0	0.971600

Now, let's report only the best accuracy retrieved on the whole training set and their relative value of the C hyper-paramater.

Kernel adopted	Value of C	Accuracy
Linear kernel	0.1	0.9336
Polynomial kernel	10.0	0.9666666666666668
RBF kernel	10.0	0.9716000000000001

It can be noticed that the accuracy of the linear version of the support vector machine classifier has a lower accuracy with respect to the other two kernel adopted: this is a clear symptom that the dataset is not linearly separable, so there doesn't exist any hyperplane which can correctly classify all elements.

The polynomial and RBF versions ended up returning a higher value of the accuracy meaning that they were able to capture some shape in data. The best score is achieved by the RBF kernel version with a value of C equal to 10; this value is the second highest value tested, although it can

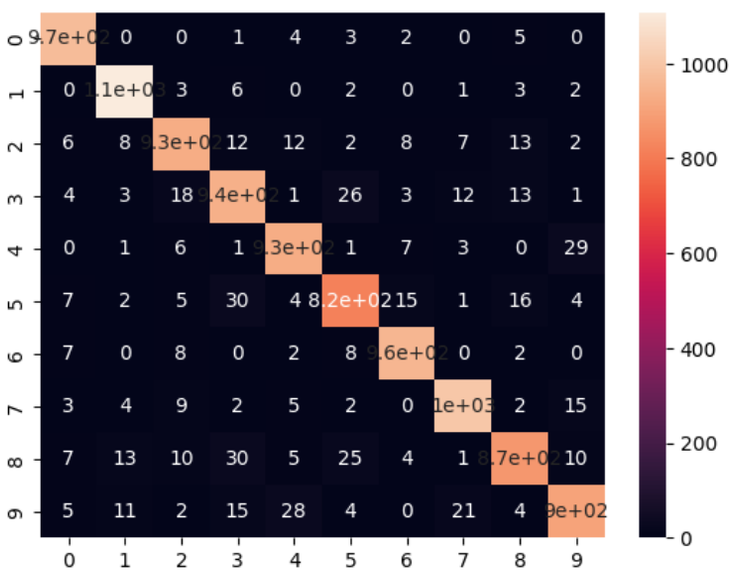
be considered “quite small”.

In the following table, the three final accuracy on the test set are shown: they are quite similar to the ones over the training set, so there is no overfitting.

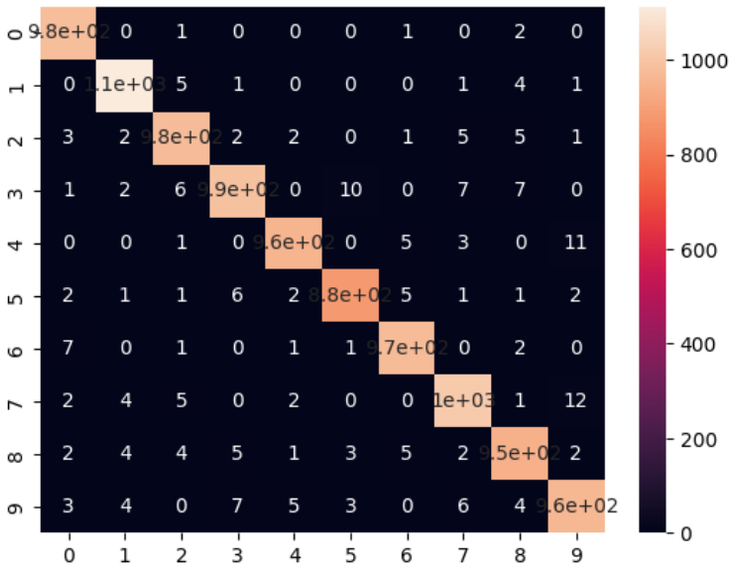
Kernel adopted	Value of C	Test accuracy
Linear kernel	0.1	0.9421
Polynomial kernel	10.0	0.9791
RBF kernel	10.0	0.9837

The best model according to the test accuracy is the RBF kernel version.

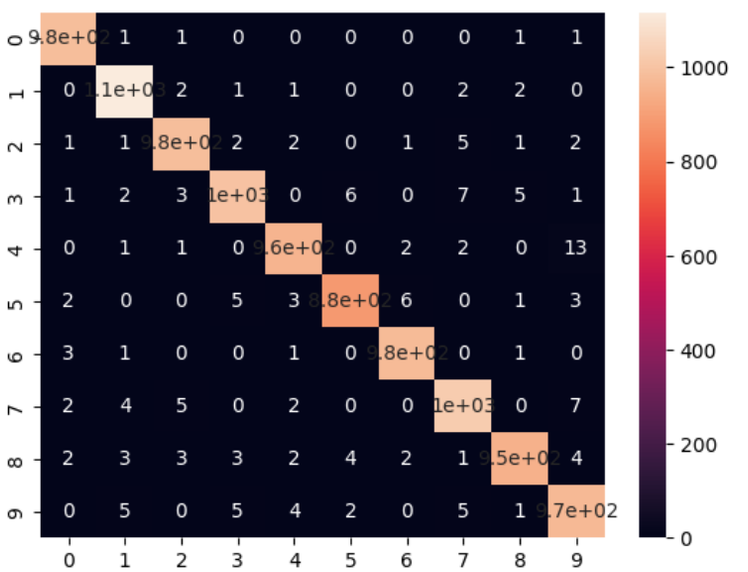
A visual representation of the scores is now presented. These have been computed thanks to the *confusion_matrix* method. First version (linear kernel)



Second version (polynomial kernel)



Third version (RBF kernel)



The optimal situation when it comes to confusion matrices is that the principal diagonal should have the highest possible values.

As it can be noticed, all three diagonals have quite high numbers due to the accuracy obtained by the three models. However, in the first confusion matrix, which refers to the linear support vector machine model, there can be found some significant values even in cells that are not part of the principal diagonal: the linear kernel version accuracy is the lowest among all the three versions.

3.2.1 Computational time

Another aspect to take into consideration while training and testing a model is the computational time taken by the algorithm to converge to a solution.

Here are presented two tables containing the computational time by the different versions of the model during the last fitting and the predicting phases.

Last fitting

Kernel adopted	Time
Linear kernel	5 mins and 43 secs
Polynomial kernel	4 mins and 26 secs
RBF kernel	2 mins and 25 secs

Predicting

Kernel adopted	Time
Linear kernel	1 min and 56 secs
Polynomial kernel	1 min
RBF kernel	57 secs

Observing both the test accuracy and the performances over the training and testing phase of all the three versions, the RBF kernel version is confirmed to be the best model.

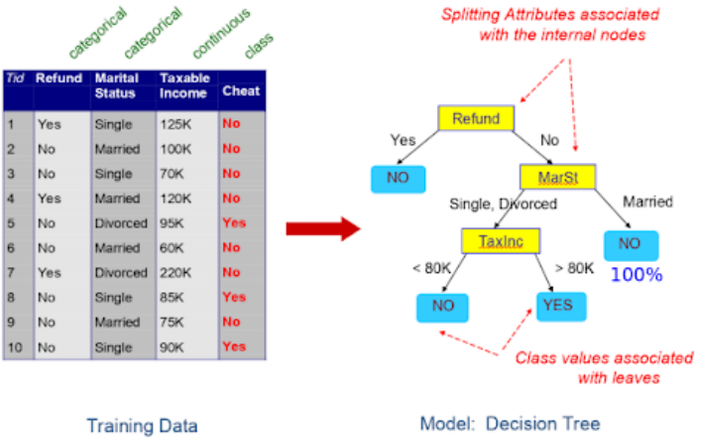
Random Forest approach

4.1 Theory behind Random Forest

Random forest or Random Decision Forest is a machine learning algorithm that combines together outputs given by multiple decision trees in order to return one single result. Since it combines together some outcomes, it is defined as an ensemble learning method.
This model can be adopted to solve both classification and regression problems. [11]

4.1.1 Decision Trees

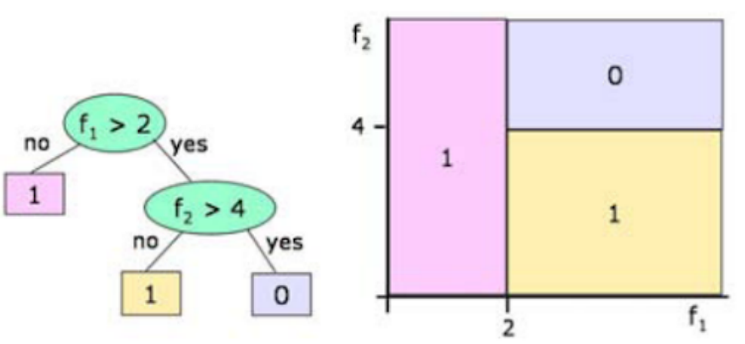
Decision Trees can be considered as the building blocks of the random forest model. They have an hierarchical and tree structure in which different elements can be identified: root node, branches, internal nodes and leaf nodes. [4]



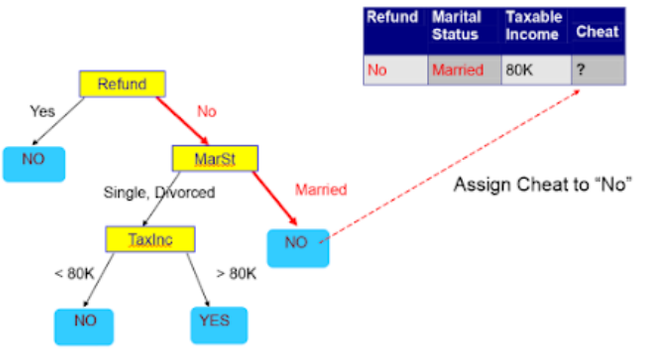
The root node is the starting point of a decision tree and it has no incoming branches, while its outgoing branches feed some internal nodes, known also as decision nodes.

This terminology is derived from the fact that each decision node can be compared to a question and each possible answer to this question is associated with another internal node or a leaf node (or terminal node), identifying decision boundaries.

In a classification context, a single terminal node contains a label which is obtained from the majority voting between all the labels of all the elements that fall into that specific leaf node.



If the decision tree is fed with a brand new element, so an element never seen before, the sample flows down in this architecture until it reaches a terminal node: it will contain the class that the model is going to associate with the input data.



In a decision tree it's important to find the best structure, so where to settle a specific node, in order to guarantee a good outcome.

To do so, some algorithms, like ID3, use a measure called Information Gain.

4.1.2 Information Gain

Information gain is one of the most common measures used to choose where a node should be put in the whole tree architecture. It is based on the concept of entropy: it is the average level of uncertainty and disorder present in the data.[12]

Given a set of examples S labeled in $C_1, \cdots, C_n$ classes, the entropy can be defined as:

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

Where p_i represents the portion of elements associated with the label C_i. If p_i is close to 1, it means that the majority of the samples belongs to the class C_i so the information that can be retrieved from these elements is low, since the label C_i is quite common in the dataset ($-log_2(p_i)$ is a small value); conversely, if p_i is close to 0, it means that only few samples belongs to that specific class so the information they give is high, since the label C_i is quite rare in the dataset ($-log_2(p_i)$ is a big value).

However, in both cases the single product returns a small value:

- With p_i close to 1, the respective $-log_2(p_i)$ is low and dominates the overall calculation;
- With p_i close to 0, the respective $-log_2(p_i)$ is high but it fails to dominate the overall calculation due to a small p_i.

Now, the actual information gain can be defined:
Given a set of examples S, an attribute $A = v_1, \cdots, v_m$ and define S_v = examples which take value v for attribute A, the information gain is derived as

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

Basically, it measures the information gain obtained knowing the value of A.

4.1.3 ID3 Algorithm

1. If S_v only contains examples in category c, then create a leaf node containing the label c;
2. If S_v is empty, then find the default category, which is the one to which most elements belong, and create a leaf node containing that label;
3. If S_v is not empty and does not contain only one category:
   (a) Calculate the information gain of all attributes;
   (b) Choose the attribute with the highest information gain as the root node;
   (c) For each value of v in A draw a branch;
   (d) For each branch:
      i. S_v = examples in S with attribute A = v;

ii. Remove A from attributes that can be chosen.

[14]

4.1.4 Some weaknesses

A non-trivial characteristic of a decision tree is its depth, even because this kind of model tends to fit perfectly the data analyzed during the learning process leading to overfitting and performing badly in the test set.
Considering the two extreme cases of this situation, it can be observed that:

- Small trees typically have low predictive accuracy (low variance but high bias);
- Deep trees more likely overfit (low bias but high variance).

Some possible solutions can be reducing either the bias (boosting) or the variance (random forest). [16]
Only the random forest is going to be explained based on the goal of the project.

4.1.5 Random Forest Classifier

As mentioned before, the random forest classifier is an ensemble method made up of several decision trees: each decision tree is fed with a randomly selected subset of input features and this process is named "feature bagging".
It is done because if some features are very strong predictors and they deeply influence the outcome, then these features will impact the results of the building trees making them, in some sense, correlated.[5]
Given a training set $X = x_1, \cdots, x_n$ with labels $Y = y_1, \cdots, y_n$, the algorithm chooses repeatedly (*B* times) a random sample with replacement of the training set and fits the trees to these samples: For $b = 1, \cdots, B$:

1. Sample, with replacement, n training examples from X, Y; call these X_b, Y_b.

2. Train a classification tree f_b on X_b, Y_b.

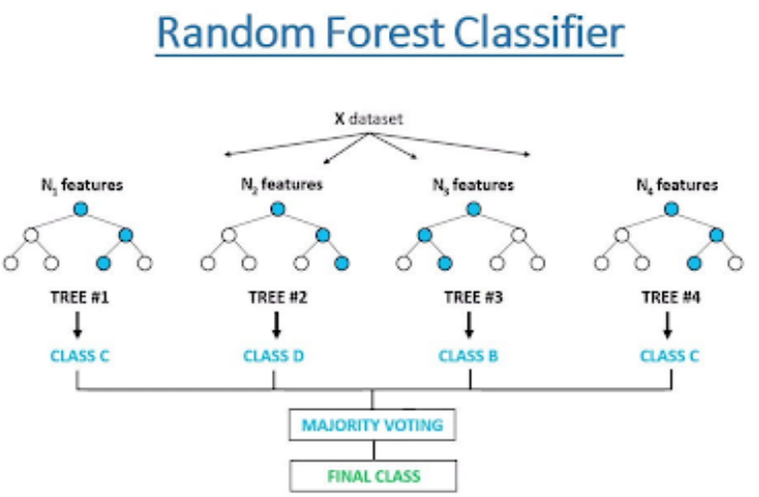
After the training phase, prediction over never seen data can be computed as the average of all the predictions made by the trees building the forest:

$$\hat{f} = \frac{1}{B} \sum_{B=1}^{B} f_b(x')$$

Or by simply taking the majority vote from all the outcome labels. In this way, the variance is contained without altering the bias.

Generally, *B* is a chosen parameter and typically this value is quite high according to the size of the training set.

For classification problems with p features, the cardinality of the subsets of random features is $\sqrt{p}$.



4.2 Implementation and results

Both training and testing phases for the Random Decision Forest model is contained in one single notebook, called *3_RF.ipynb*.

The smaller training set is used to tune the *n_estimators*, that is the number of trees composing the forest: the possible values assigned to this *n_estimators* belong to the interval [100, 1100) jumping by 100 units.

In the following table are shown the values assumed by the hyper-parameter and the respective accuracy achieved over the smaller training set.

Number of trees	Accuracy
100	0.953467
200	0.954467
300	0.954533
400	0.954867
500	0.955333
600	0.955200
700	0.955200
800	0.956200
900	0.955533
1000	0.955600

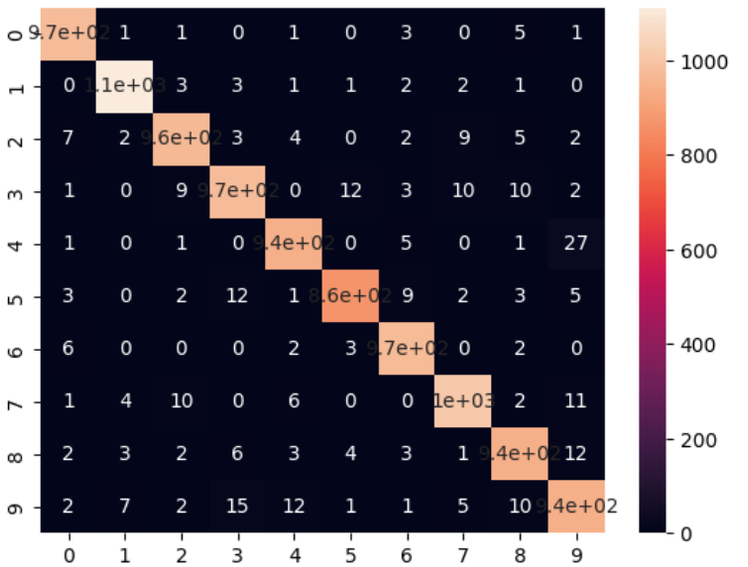
A relevant aspect about the tuning process over the *n_estimators* parameter is that there isn't a linear trend between the number of trees and the accuracy produced: indeed, it turns out that accuracy increases, decreases and then rises and falls again.

This could be possibly explained by the fact that not all trees specialize on features that are relevant, discriminant for the classification task.

The best accuracy is reached with 800 trees and the final scores over the whole training set and the test set are shown in the following table

Number of trees	Train Accuracy	Test Accuracy
800	0.9561999999999999	0.9684

The obtained confusion matrix has a good shape: the principal diagonal has the highest values, although some significant numbers characterized the other cells. This may have happened because some written digits have a quite similar shape and written in a quite similar way so the model tended to misclassify them.



4.2.1 Computational time

The computational time taken by the Random Decision Forest model is high in the fit phase. This is explained by the fact that for each single tree that composes the forest a sample of the dataset has to be chosen and this is an expensive operation.

Whereas, the test phase is really fast since the model simply needs to retrieve the most common class label by executing each tree on the same test set.

Phase	Time
Fit phase	7 mins and 6 secs
Test phase	2 secs

Naive Bayes approach

5.1 Theory behind Naive Bayes

The Naive Bayes classifier is part of the family of generative models and, due to this, it tries to find a distribution that explains the input data with intention to exploit it to make good predictions; so, this kind of model does not learn which features are determinant for the final outcome.

This classifier is a probabilistic classifier, since it relies on the Bayes' Theorem, and works under some particular assumptions from which derived the title "naive".

Firstly, it assumes that the predictors are conditionally independent (unrelated) to any of the features that characterized the input data and, secondly, it assumes that all these features contribute equally to the final result. [3]

However, in real world problems, typically, these assumptions are not always verified, so this model tries to simplify things by computing only one probability for each variable, so for each dataset element.

Given a training set whose elements are characterized by n features, $x = (x_1, \cdots, x_n)$, and given the set of classes $C = C_1, \cdots, C_k$, it's possible to assign all the k probabilities $p(C_k|x_1, \cdots, x_n)$.

Although, if the number of features is too big, these calculations can be infeasible to compute, therefore it's necessary to reformulate them in order to make them more tractable.

With the support of the Bayes' Theorem, the conditional probability can be written as

$$P(C_k|X) = \frac{P(C_k)P(X|C_k)}{P(X)}.$$

Actually, the focus is only on the numerator: the denominator does not depend on C_k and all features' values are given so it is a constant.
Due to this, the numerator is equivalent to the joint probability model

$$P(C_k, x_1, \cdots, x_n)$$

And using it, the joint probability can be rewritten, with the support of the chain rule for repeated applications of the definition of conditional probability, as: [10]

$$\begin{aligned}
P(C_k, x_1, \cdots, x_n) &= P(x_1, \cdots, x_n, C_k) \\
&= P(x_1|x_2, \cdots, x_n, C_k)P(x_2, \cdots, x_nC_k) \\
&= P(x_1|x_2, \cdots, x_n, C_k)P(x_2|x_3, \cdots, x_n, C_k)P(x_3, \cdots, x_nC_k) \\
&= \cdots \\
&= P(x_1|x_2, \cdots, x_n, C_k)P(x_2|x_3, \cdots, x_n, C_k) \cdots P(x_{n-1}|x_n, C_k)P(x_n|C_k)P(C_k)
\end{aligned}$$

With the "naive" assumptions explained above:

$$P(x_i|x_{i+1}, \cdots, x_n, C_k) = P(x_i|C_k)$$

Finally, the joint probability model can be expressed as

$$\begin{aligned}
P(C_k|x_1, \cdots, c_n) &\propto P(C_k, x_1, \cdots, x_n) \\
&\propto P(C_k)P(x_1|C_k)P(x_2|C_k)P(x_3|C_k) \cdots \\
&\propto P(C_k) \prod_{i=1}^{n} P(x_i|C_k)
\end{aligned}$$

The symbol $\propto$ is introduced since the denominator is ignored in these calculations.

Now, the problem is returning the label which posterior probability is maximum

$$\hat{y} = \underset{k \in 1, \cdots, K}{\operatorname{argmax}} P(C_k) \prod_{i=1}^{n} P(x_i | C_k)$$

An important assumption for the Naive Bayes model is that the distribution of each feature is known. Here some most common choices:

- Gaussian distribution: features' values are distributed according to a normal distribution;
- Multinomial distribution: this type of distribution is mostly used in document classification. The feature vectors represent the frequencies with which words appear in the document;
- Bernoulli distribution: features are independent binary variables and even this approach is popular in document classification.

5.2 Implementation and results

The implementation of the Naive Bayes classifier is contained in a Python file called *NAIVE.py*, while the training and the testing phases are computed in the notebook file *4_NAIVE.ipynb*.
An assumption about the probability distribution to use was given in the description of the current project: each pixel is distributed according to Beta distribution of parameters α and β:

$$d(x, \alpha, \beta) = \frac{\Gamma(\alpha+\beta)}{\Gamma(\alpha)+\Gamma(\beta)} x^{\alpha-1}(1-x)^{\beta-1}$$

The two parameters can be derived as follows:

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

It can be noticed that there could be an issue on estimating these parameters with these formulas:

- The value of K might be negative or zero if the following case happens:

  $$\frac{E[X](1-E[X])}{Var(X)} - 1 \leq 0$$

  Consequently, this would make both α and β negative or equal to 0. In order to solve this problem, the negative and zero values of α and β are set to the minimum positive value of α and β respectively, based on the pixel distribution for a specific class.

In *NAIVE.py* there can be found the class *NaiveBayesClassifier*, through which the entire classifier's implementation is given. This class is declared with the support of the *BaseEstimator* class and the three distinctive methods are:

```
def __init__(self) -> None:
```

```
def fit(self, X_train:pd.DataFrame, y_train: np.ndarray) -> None:

def predict(self, X_test: pd.DataFrame) -> pd.Series:
```

In the first method there is only one statement which is the call to the *__init()__* method of *BaseEstimator*.
The *fit(...)* method returns nothing and has as input parameters two variables:

- *X_train*: This contains all the features' values of the training set samples;
- *y_train*: It is an array in which there are the labels associated with each sample of the training set.

These two variables are saved in *self.X_trainset* and *y_trainset* respectively, becoming proper parameters of the class.
After these two assignments, for each possible label that a sample can assume some other variables are computed; in this way it's possible to exploit them during the predicting phase.
In particular:

- All the images belonging to the same class are rescued and their mean and variance are calculated through the library methods *mean(...)* and *var(...)*;
- The α, β and K parameters are computed according to the formulas above and the obtained α and β are immediately modified in order to make them completely positive;
- Last but not least, the frequency of the class is obtained by the ratio between the number of the elements belonging to the class and the cardinality of the entire training set.

All of these parameters are saved in a dictionary in which the key is the digit on which the variables were computed and the respective value is

the collection of the above parameters.

The *predict(...)* method returns a *pandas series*, which represents the predicted labels for the given test set which is the only input parameter. The core of this method is given by two nested for loops:

- The outer one iterates through the test set samples;
- The inner one iterates on all the possible label values, so from 0 to 9.

Therefore, for each element of the test set the beta probability for each label is computed according to the related parameters and the outcome label is equal to the one whose probability is the highest possible of all. Given that the Beta distribution is a continuous distribution, to calculate the Beta probability, which is described as

$$\prod_{i=1}^{n} P(x_i|C_k),$$

the Beta cumulative density function (*cdf*) is applied. The calculation made for each point *r* is given by the following integral:

$$\int_{r-\epsilon}^{r+\epsilon} \frac{\Gamma(\alpha+\beta)}{\Gamma(\alpha)\Gamma(\beta)} x^{\alpha-1}(1-x)^{\beta-1}\,dx$$

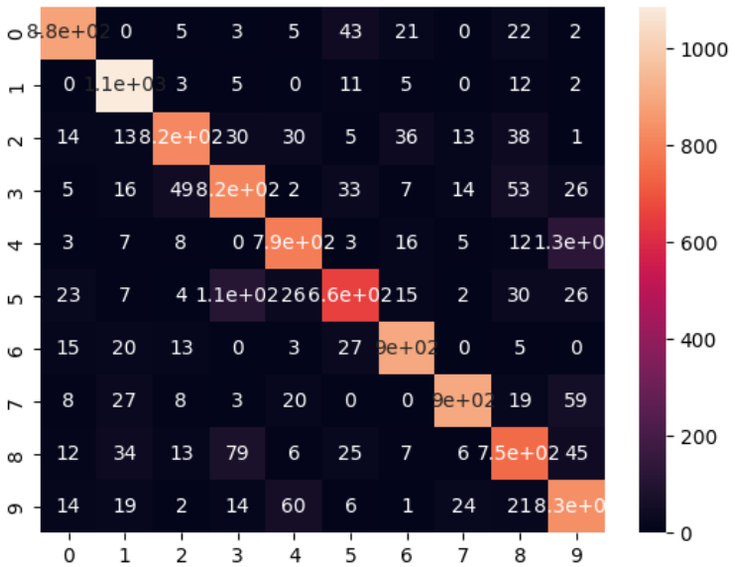
Where ϵ was chosen equal to 0.1.

And, finally, the final probability is computed as:

$$P(C_k)\prod_{i=1}^{n} P(x_i|C_k)$$

An the maximum between all of the ten possible values (one for each possible label) is saved and appended to the returned array.

In this model there were no hyper-parameter to tune so the main function of the notebook file simply trains the algorithm over the whole training set and then tests it to retrieved the final accuracy, which is equal to 0.8433. Due to this relatively low accuracy, the associated confusion matrix is characterized by high values even in cells that are not composing the principal diagonal, as it can be seen in the image below.



5.2.1 Computational time

Since the computation is mostly made up of constant calculations, the time taken by the model is really low compared to the other models.

In particular:

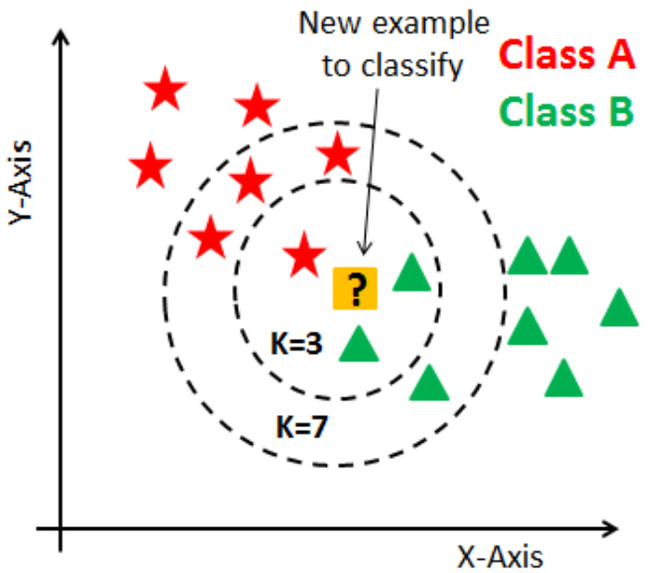
Phase	Time
Fit phase	almost 1 secs
Test phase	1 min and 6 secs

Actually, the Naive Bayes has the best performance regarding the computational time taken.

k-NN approach

6.1 Theory behind k-Nearest Neighborhood

k-NN algorithm is the acronym of k-Nearest Neighbors algorithm.
It is a non-parametric algorithm and it exploits the distance between the elements of the dataset in order to return the desired label.
In practice, if a new data is given to the algorithm, it finds the k neighbors of this point, based on some criteria, and the outcome label is obtained by the majority voting between its neighbors. [2]
Summarizing, the training phase consists of storing the feature vectors and the labels, instead the classification phase assigns to an element a label according to its neighborhood.
The number of neighbors to take into consideration (k) is a hyper-parameter: greater is its value and lower is the probability of misclassification but this leads to weak class boundaries.
A special case is when $k = 1$ because, in this situation, it is considered only the closest point to making predictions.
An example of how the hyper-parameter k influences this model is given in the following figure.



Given a new example (the yellow square) based of the number of neighbors the prediction can change:

- If the model considers three neighbors, their majority voting returns Class B (green);
- If the model considers seven neighbors, their majority voting returns Class A (red).

Anyway, as the cardinality of the dataset and the number of considered neighbors increase, (the latter more slowly than the first one), k-NN converges to the optimal classifier.

However, *k* should be chosen also considering how the distance, or similarly the proximity, between data points is computed. [17]

Typically, if the algorithm has to deal with continuous variables, a common distance measure is the Euclidean distance:

$$d(p,q) = \sqrt{(p_1 - q_1)^2 + \cdots + (p_n - q_n)^2} = \sqrt{\sum_{i=1}^{n}(p_i - q_i)^2}$$

6.2 Implementation and results

The files related to the k-Nearest Neighbors model are *KNN.py* and *5_KNN.ipynb*.
In *KNN.py* file the *KNNClassifier* class is declared with the support of the *BaseEstimator* and the three main methods necessary to implement from scratch this classifier are:

```
def __init__(self) -> None:

def fit(self, X_train:pd.DataFrame, y_train: pd.DataFrame | np.ndarray)
                                                        -> object:


def predict(self, X_test: pd.DataFrame) -> pd.Series:
```

Since the k-NN classifier has an hyper-parameter which is the number of neighbors to take into consideration (*k*), the initialization method *__init()__* accepts in input one parameter that is exactly *k* assigning it to the homonymous parameter of the class; by default, its value is equal to 3, meaning that the predicted label for each element is produced by considering its three nearest samples.
In the *fit(...)* method two input parameters are requested:

- *X_train*: This contains all the features' values of the training set samples;
- *y_train*: It is composed by the labels associated with each sample of the training set.

These two variables are saved in *self.X_trainset* and *y_trainset* respectively, becoming proper parameters of the class.
In order to manage correctly the data a check over the *y_train* parameter was deemed necessary, in fact if its type is a *pandas DataFrame* then it is converted to a *numpy array*.
At the end of the method, the classifier itself is returned.

The final predictions are retrieved with the *predict(...)* method which returns a *pandas series* containing the resulting labels for the input parameter, or the test set.
As in the Naive Bayes *predict(...)* method, even in this case, the core of the implementation is a for loop on the samples of the test set: for each element present in the input set:

- The euclidean distance with all the points in the training set is computed;
- The labels of the *k* nearest neighbors are retrieved from *self.y_trainset*;
- The mode of all the labels is calculated thanks to the *statistics* library through the *mode* method;
- The resulting label is appended to the *final_prediction* variable, which is the returned outcome.

As mentioned before, the hyper-parameter in this model is given by *k*, so the cardinality of the neighborhood for each element; the tested values over this parameter are $k = [2, 3, 4, 5, 6]$
In the following table are presented the accuracy achieved by the model trained on the smaller training set according to the different values that the *k* parameter can assume.

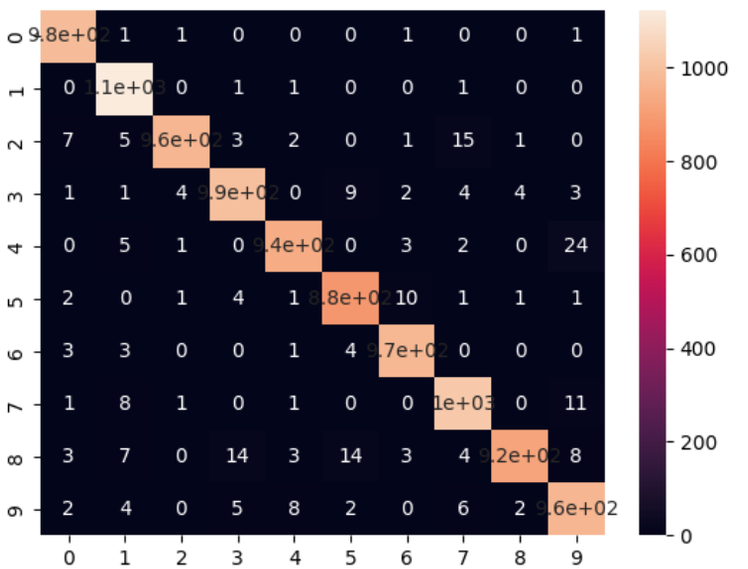
Value of k	Accuracy
2	0.955667
3	0.957533
4	0.959067
5	0.956067
6	0.957267

The value of the accuracy does not change significantly while *k* increases.

However, the best number of neighbor to take into consideration for the prediction turns out to be 4 and the accuracy on the whole training set and test set are shown below.

Number of neighbors	Train Accuracy	Test Accuracy
4	0.9590666666666667	0.9757

Seen the high and discrete accuracy of the test set, the related confusion matrix has a good shape: the principal diagonal has high values, although some significant instances were misclassified.



6.2.1 Computational time

The computational time taken by this model is deeply influenced by the calculation for each test sample of the distances with of the elements in the training set.

As it can be noticed, the fit phase is the fastest possible while the predict phase required several minutes.

Phase	Time
Fit phase	0 secs
Test phase	42 min and 2 secs

Conclusions

Let's report for the last time the accuracy and the computational time taken by all the model, in order to have a more clear visualization over the data.

Classifier	Value hyper-parameter	Test Accuracy
Linear SVM	C = 0.1	0.9421
Polynomial SVM	C = 10.0	0.9791
RBF SVM	C = 10.0	0.9837
Random Forest	trees = 800	0.9684
Naive Bayes	/	0.8433
k-NN	k = 4	0.9757

The best model regarding the accuracy achieved is 'RBF SVM' with 0.9837 and the worst model is 'Naive Bayes' with 0.8433.

Classifier	Fit Computational Time	Predict Computational Time
Linear kernel	5 mins and 43 secs	1 min and 56 secs
Polynomial kernel	4 mins and 26 secs	1 min
RBF kernel	2 mins and 25 secs	57 secs
Random Forest	7 mins and 6 secs	2 secs
Naive Bayes	almost 1 secs	1 min and 6 secs
k-NN	0 secs	42 min and 2 secs

Classifier	Total Computational Time
Linear kernel	7 mins and 4 secs
Polynomial kernel	5 mins and 26 secs
RBF kernel	3 mins and 22 secs
Random Forest	7 mins and 8 secs
Naive Bayes	1 min and almost 7 secs
k-NN	42 mins and 2 secs

Concerning the fit computational time, the best model is *k-NN* with 0 seconds, while the worst is *Random Forest* with 7 minutes and 6 seconds. Based on the predict computational time, the best model is *Random Forest* with 2 seconds and the worst is *k-NN* with 42 minutes and 2 seconds. For the total computational time, the best model is *Naive Bayes* with 1 minute and almost 7 seconds but the worst is *k-NN* with 42 minutes and 2 seconds.

Summarizing, the *k-NN* model has not good performances: the time taken is too large and its accuracy is surpassed by other 2 models whose computational time is significantly lower.
The fastest model considering the overall computational time is the *Naive Bayes* but its accuracy isn't satisfactory at all.
The *Linear SVM*, the *Random Forest* and the *Polynomial SVM* are discrete models with a good accuracy obtained in a reasonable computational time.
However, the best model is *RBF SVM*: the accuracy produced is the highest in the less computational time.

Bibliografia

[1] Generative models vs Discriminative models for Deep Learning. URL https://www.turing.com/kb/generative-models-vs-discriminative-models-for-deep-learning.

[2] What is the k-nearest neighbors algorithm? | IBM. URL https://www.ibm.com/topics/knn.

[3] What are Naive Bayes classifiers? | IBM. URL https://www.ibm.com/topics/naive-bayes.

[4] What is a Decision Tree | IBM. URL https://www.ibm.com/topics/decision-trees.

[5] What is Random Forest? | IBM. URL https://www.ibm.com/topics/random-forest.

[6] What is supervised learning? | Definition from TechTarget, . URL https://www.techtarget.com/searchenterpriseai/definition/supervised-learning.

[7] What is Supervised Learning? | IBM, . URL https://www.ibm.com/topics/supervised-learning.

[8] Discriminative model, March 2023. URL https://en.wikipedia.org/w/index.php?title=Discriminative_model&oldid=1144073383. Page Version ID: 1144073383.

[9] Generative model, December 2023. URL https://en.wikipedia.org/w/index.php?title=Generative_model&oldid=1188040323. Page Version ID: 1188040323.

[10] Naive Bayes classifier, November 2023. URL https://en.wikipedia.org/w/index.php?title=Naive_Bayes_classifier&oldid=1186105862. Page Version ID: 1186105862.

[11] Random forest, November 2023. URL https://en.wikipedia.org/w/index.php?title=Random_forest&oldid=1186786280. Page Version ID: 1186786280.

[12] Entropy (information theory), December 2023. URL https://en.wikipedia.org/w/index.php?title=Entropy_(information_theory)&oldid=1188990275. Page Version ID: 1188990275.

[13] Support vector machine, November 2023. URL https://en.wikipedia.org/w/index.php?title=Support_vector_machine&oldid=1183475870. Page Version ID: 1183475870.

[14] AshirbadPradhan. Decision Tree ID3 Algorithm |Machine Learning, June 2023. URL https://medium.com/@ashirbadpradhan8115/decision-tree-id3-algorithm-machine-learning-4120d8ba013b.

[15] Andrea Torsello. *Linear Classifiers*. .

[16] Andrea Torsello. *Decision Trees*. .

[17] Andrea Torsello. *Generative Models*. .