POLITECNICO DI TORINO

Master's Degree in Computer Engineering

Master's Degree Thesis

Custom incremental learning approaches in real world scenarios

Supervisor

Prof. Paolo GARZA

Company supervisor

Dr. Edoardo CONTE

Candidate

Enrico POSTOLOV

April 2021

Summary

Nowadays, machine learning approaches have become an **integral** part in a wide variety of technologies and use-cases, ranging through a lot of diverse fields: from healthcare to automotive, or from banking to supply chain optimization, arriving also at the point of directly influencing our everyday's life (e.g. with smart-home devices, or even just with smartphones).
In each of the aforementioned, usually, a big quantity of data needs to be collected, so that the various algorithms which are applied can provide better models and thus, give results that most of the times coincide with the *"ground truth"*.
To improve these performances, what is usually done by several companies developing machine learning-based solutions, is to cyclically **train from zero** their models with bigger quantities of data, which are accumulated by their platforms over time.
As it can be imagined, the process of defining an entirely new classifier after every predefined time window, can be **expensive** in terms of **time** and, more importantly, **resources**.
In an era where companies, especially start-ups, rely on external infrastructures provided *"as a service"* to deliver their products to the customers, saving is of vital importance, and thus the training process mentioned above can't be the best way to go.
In this work, we collaborated with a small company to try and solve this issue, exploiting those approaches defined as **incremental**: a single classifier is kept over time and updates due to the presence of new data are applied directly to that instance, giving it thus the possibility to adapt to possible changes in data without the need of starting from the beginning the learning mechanism.
The classifiers taken into consideration are the following three: **Naïve-Bayes classifier**, **Random Forest classifier** and **Extra-Trees classifier**.

From our results, the performances are comparable to the ones obtained following the standard approaches, outlining so a new path that can be pursued for the implementation of this type of solutions.

Acknowledgements

Ringraziamenti

Finalmente sono giunto alla fine di questo percorso, che è stato impegnativo, molto, ma gratificante, almeno il doppio.
Tutto questo non sarebbe stato possibile senza un adeguato "mix" di persone al mio fianco, pronte a spronarmi quando la stanchezza si faceva sentire e la tentazione di lasciar stare bussava alla porta, a rallegrarmi quando tutto sembrava andare per il verso sbagliato, e soprattutto a farmi capire che per me ci saranno sempre.
Pertanto, nonostante io sia una persona di poche parole, penso sia doveroso spendere qualche parola per ringraziarle in maniera adeguata.
A Denis, mio fratello, che oltre ad essere il mio "bro" è il mio migliore amico, la mia ancora e la persona con il cuore più grande che io conosca. Potrei ringraziarlo per almeno un migliaio di ragioni, e comunque non sarebbero abbastanza. Spesso mi dice che io sono il suo esempio, ma in realtà è il contrario.
Ai miei genitori, Gorgi e Martinka (o Giorgio e Martina, se vogliamo italianizzare), che nemmeno trent'anni fa sono emigrati in cerca di una vita migliore da un Paese più piccolo del Piemonte, arrivando qui in Italia. Non mi hanno mai fatto mancare nulla, cercando sempre di accontentarmi e facendo i salti mortali, pur di vedere un sorriso sulla mia faccia. Mi hanno sempre assecondato nelle scelte prese durante la mia vita, dandomi consigli e soprattutto dandomi la possibilità di sbagliare.
Il raggiungimento di questo traguardo è un sogno che condividevamo, e sono orgoglioso di poter dire che finalmente ce l'abbiamo fatta: non sarebbe stato possibile senza tutto l'amore che mi avete dato, e che continuate a darmi ogni giorno.
Ai miei nonni, che so quanto vorrebbero celebrare questo momento insieme a me, ma che per cause di forza maggiore non possono esserci. So quanto

siete orgogliosi di me in questo momento.
A Dani, Fra, Giu, Ila, Rick e Simo, che in questi due anni sono stati la mia seconda famiglia, e che hanno condiviso con me gioie e dolori. Non potevo desiderare compagni migliori, che la vostra genuinità e magnanimità vi contraddistinguano per quello che siete: persone meravigliose. Un giorno potremo tornare ad abbracciarci, e sono sicuro che recupereremo tutto quello che questo virus ci sta togliendo.
Agli amici di Castagnole, i "Marci", con i quali sono cresciuto e ho vissuto bellissime esperienze. Le risate che ho fatto insieme a voi sono infinite.
Al Prof. Garza, per essere stato il miglior professore che io abbia mai avuto, e per avermi dato la disponibilità a farmi da relatore in questo lavoro che spero si possa considerare interessante.
A Edo, che mi ha permesso di mettermi in gioco con questa avvincente sfida.
A tutti voi, un grandissimo e sincero *GRAZIE*.
L'ultimo ringraziamento lo voglio fare a me, che ho sofferto e lottato tanto per arrivare fino a questo punto, contro lo scetticismo di molti che mi davano per spacciato.
Sono curioso di affrontare le sfide che verranno con la stessa mentalità: quella del duro lavoro.

"It is not the critic who counts; not the man who points out how the strong man stumbles, or where the doer of deeds could have done them better. The credit belongs to the man who is actually in the arena, whose face is marred by dust and sweat and blood; who strives valiantly; who errs, who comes short again and again, because there is no effort without error and shortcoming; but who does actually strive to do the deeds; who knows great enthusiasms, the great devotions; who spends himself in a worthy cause; who at the best knows in the end the triumph of high achievement, and who at the worst, if he fails, at least fails while daring greatly, so that his place shall never be with those cold and timid souls who neither know victory nor defeat."

Theodore Roosevelt

Table of Contents

List of Tables

List of Figures

Acronyms

ML
Machine Learning

IL
Incremental Learning

NB
Naïve-Bayes

MAP
Maximum A Priori

MLE
Maximum Likelihood Estimation

RF
Random Forest

ET
Extra-Trees

Chapter 1

Introduction

With **Machine Learning (ML)** we refer to the study of that set of algorithms and methodologies having the ability of improving themselves over experience, as stated by Tom Mitchell in [1].
This kind of algorithms has been increasingly adopted by the industry for solving several tasks, becoming thus a core element of a lot of platforms and technologies.
The experience mentioned above is expressed in the form of **sample data**, which is collected over time by its related system, and has the role of building the knowledge that can be used for defining a model: the latter is what will make decisions or predictions without the need of being programmed to do so [2].
The quality, which can be defined by performing a wide set of measures, about how a given model makes its predictions/decisions is influenced by the amount of data which is provided to the model. So, what is usually done by companies in the "practical" world, is to define an entirely new estimator whenever a certain amount of time elapses, or when a given amount of data is collected.
The main drawback of exploiting this type of approach is that a lot of resources are wasted, since it is needed to define again a new model, needing to go through the whole data set when one of the two conditions (or both) stated above are met.
To address the issue of reusing the entire data set several times, **incremental learning** approaches can be considered. The goal of these techniques is to update a given model *without* the need of analyzing again what can be defined as "old" knowledge, but instead using only "new" training data.

In this way, the resulting model keeps its dependency to old knowledge, but has the ability of adapting to new training data without looking again at what happened in the past.
In the following work, we collaborated with RestWorld S.r.l., a start-up based in Turin, and we implemented these techniques on three classifiers, which will be explained in a more detailed way in chapter 2: **Naïve-Bayes** classifier, **Random-Forest** classifier, and **Extra-Trees** classifier.
The implementations, instead, will be discussed in chapter 3.
The different implementations have been validated by deriving several metrics that are well known in machine learning, such as accuracy, precision, recall, f1-score, macro and weighted average. The values for the mentioned metrics have been obtained performing different tests on a local machine.
The various tests and the obtained results are going to be commented in chapter 4, in which there will also be a comparison with what is provided by the standard techniques.
Finally, in chapter 5, we are going to suggest some future work and possible enhancements that can be followed.

Chapter 2

Problem statement and state of the art techniques

In this work we are interested in adopting incremental learning approaches for solving a real-world problem, and in comparing them to the standard ones, by using the same "core" algorithms.
Here we are defining in detail the scenario in which the solution is conceived and developed, and also the goal of the project. At the end, we are considering some state of the art techniques for the given scenario.

2.1 Goal and scenario

RestWorld S.r.l. is a small start-up based in Turin which operates in the job market, more specifically in the catering world. They provide a platform which has the role of finding optimal candidates for a given restaurant: the underlying system needs candidates and restaurateurs to fill a form (the one for the first different from the one for the second), and then the "matching" is performed by analyzing the collected data.
For what concerns the candidates, the following information is collected:

- **Sector** in which the candidate would like to find a position (one possible choice among *Saloon*, *Kitchen* and *Cafe*);
- **Time availability**: it is when the candidate is available for working (chosen among *Full-time*, *Part-time*, *By Call* and *On weekends*);
- **Workshift**, which is chosen among *Breakfast*, *Lunch* and *Dinner*;

- **Premise**, that has the role of representing the kind of activity(ies) for which the applicant would like to work for (several categories can be selected from here, e.g. *Fast food*, *Risto-pub*, *Pizzeria*, *Farmhouse*, *Diner*, *Catering*, *Restaurant*, *Starred restaurant*);
- **Start Time**, indicating from when the candidate is available (a single choice can be made amid *Now*, *Next week*, *Next month*);
- **Transport type**, that states how the user can reach the work place (*Car*, *Motorbike*, *Bike*, *Public transport*, *Sharing services*, *Walk*);
- **Transfers availability**, a flag depicting the availability from the applicant to transfer for job-related causes;
- **Gender** of the candidate;
- **Address** of the candidate;
- **Name** of the candidate;
- **Surname** of the candidate;
- **Birth date** of the candidate.

Instead, regarding restaurants' data, the platform collects the subsequent information:

- **Restaurant name**;
- **Referent**, that is basically the restaurateur's name;
- **Address** where the restaurant is located;
- **Management type**, which has to be chosen among *Family run*, *Sole proprietorship*, *Franchise*;
- **Restaurant type**: several values can be chosen for this attribute (e.g. *Cafe*, *Brewery*, *Fast food*, *Starred restaurant*, *Wine shop*, *Canteen*, *Diner* and many others);
- **Cuisine type**: it represents what the restaurant can offer to its customers (the possible values are *Homemade*, *Gourmet*, *Minimal*, *Piedmontese*, *Vegetarian*, *Meat*, *Fish*);

- **Customer Type**, indicating the categories of customers frequenting mostly the activity into consideration (chosen amid *Young people*, *University students*, *Office workers*, *Families*, *Informal*, *Formal* and, finally, *Tourists*);
- **Number of employees** working in the restaurant (selected among *Less than 5*, *Between 5 and 10*, *Between 10 and 15*, *Between 15 and 20*, and *More than 20*);
- **Average served people**: it has the role of giving an idea about how many people are served by the restaurant in a day (this attribute can take a single value among *Less than 30*, *Between 30 and 60*, *Between 60 and 100*, and *More than 100*);
- **Average price** that is spent having a meal in the restaurant (chosen amid *Less than 15€*, *Between 15€ and 20€*, *Between 20€ and 30€*, *More than 30€*);
- **Opening days** of the restaurant;
- **Workshifts** adopted by the restaurant (*Breakfast*, *Lunch*, *Dinner*, and *Pub*);
- **Contact** platform on which the restaurant can be reached (multiple options, like *Mail*, *WhatsApp*, *SMS*, *Phone*, and *Facebook* are available);
- **E-mail**;
- **Telephone**.

Currently, the matching process is completed without the use of any particular ML solution, in fact the **candidate-restaurant pairing** is made by manually analyzing the gathered information, and trying to derive the best couples possible, in order to satisfy both the applicants and the restaurateurs. The crisis caused by the COVID-19 pandemic, that we are still facing and struggling against, has surely impacted this sector in a huge manner. But, it is quite clear that the approach adopted in here can become unsustainable in the future, assuming that the amount of data grows over time.
Also, it has become more and more common the adoption of those solutions provided *"as a service"* for storing data, especially in small companies like the one that we are talking about right now. Thus, it is likely that having

to deal with a big amount of information entails high costs for maintaining the needed infrastructure.
So, the goal of this work is to build a ML solution able to perform the pairing process introduced before with an **incremental** approach, in order to not keep and reanalyze old data, relying only on new knowledge: this could permit to have a faster model and lower costs for the used services.

2.2 State of the art techniques

Given the predominantly categorical data we have to deal with, and the "pairing" that we prefer to express through a label y (e.g. $y = -1$ denoting a **"no-match"**, $y = 0$ a **"possible match"**, and $y = 1$ indicating a **"match"**), the problem under study can be represented as a **supervised** task, in which the training samples are tagged and it is needed to define a model able to provide one class after receiving certain data (i.e. the information related to a candidate and a restaurant) in input.
This kind of problem, by the state of the art, can be solved with several different techniques, but in here we want to focus on the following, as they are going to be extended for obtaining the incremental models:

- **Naïve-Bayes classifier**;
- **Random Forest classifier**;
- **Extra-Trees classifier**.

Here below, we are going to analyze them one by one, denoting their working principles.

2.2.1 Naïve-Bayes classifier

The **Naïve-Bayes classifiers**, in statistics, are part of those classifiers so called "probabilistic". Probabilistic classifiers have the peculiarity of being able to predict a **probability distribution** over a set of several classes, accepting in input an observation, instead of only providing in output the most likely class the latter belongs to.
As stated in [3] by Trevor Hastie, this type of classifiers have the advantage

of providing a classification which can be advantageous in its own right, but also when we need to combine several classifiers for defining ensemble models.
Going back to the Naïve-Bayes classifier, it exploits the popular **Bayes' theorem**, assuming a strong independence among the features of the samples provided in input (from here the adjective *naïve*).
The Bayes' theorem has the role of describing the probability of a particular event, by basing on some prior knowledge of conditions which can be somehow related to the event taken into consideration [4]. Mathematically, it is described by the following equation [5]:

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)} \tag{2.1}$$

in which A and B are the different **events** of interest, with P(B) being different from 0.
The following characteristics have to be kept in mind:

- P($A|B$) is a **conditional probability** representing how likely the event A occurs given that the event B is true;
- similarly, P($B|A$) is the **conditional probability** indicating the possibility of event B to occur given that A is true;
- both P(A) and P(B) are the probabilities of observing separately A and B, and they are called **marginal probabilities**.

The probabilistic model

In the precise context of the classifier taken into consideration, the theorem stated above has the role of defining the **probabilistic model** behind the classifier. In fact, in an abstract manner, the Naïve-Bayes classifier is a **conditional probability model**. Starting from a problem instance (which can be called **sample**) to be classified, that can be represented by a vector $\mathbf{x} = (x_1, \ldots, x_n)$, in which n features are described and assumed as strongly **independent** among each other, conditional probabilities are defined for each of the possible K classes, and they are expressed as $p(C_k \mid \mathbf{x} = (x_1, \ldots, x_n))$.
Given a situation in which n is high, or one in which each feature could take a lot of different values, the model can be reformulated exploiting the Bayes

theorem, as described in [6]:

$$p(C_k \mid \mathbf{x}) = \frac{p(C_k)\ p(\mathbf{x}_k)}{p(\mathbf{x})} \tag{2.2}$$

The equation stated above can be "translated" into a more readable form as the following:

$$\text{posterior} = \frac{\text{prior} \times \text{likelihood}}{\text{evidence}} \tag{2.3}$$

Since the denominator is not depending on C, and we are given the various values $\mathbf{x_i}$ of the features, we can state that the denominator can be considered as constant.
The numerator, instead, is considered equal to the **joint probability model** $p(C_k, x_1, \ldots, x_n)$ that, using the well known **chain rule** for multiple recurrences of the conditional probability, can be rewritten in the subsequent way:

$$\begin{aligned} p(C_k, x_1, \ldots, x_n) &= p(x_1, \ldots, x_n, C_k) \\ &= p(x_1 \mid x_2, \ldots, x_n, C_k)\ p(x_2, \ldots, x_n, C_k) \\ &= p(x_1 \mid x_2, \ldots, x_n, C_k)\ p(x_2 \mid x_3, \ldots, x_n, C_k)\ p(x_3, \ldots, x_n, C_k) \end{aligned} \tag{2.4}$$

Obtaining thus:

$$p(C_k, x_1, \ldots, x_n) = p(x_1 \mid x_2, \ldots, x_n, C_k) \cdots p(x_{n-1} \mid x_n, C_k)\ p(x_n\ C_k)\ p(C_k) \tag{2.5}$$

It is at this point that the **conditional independence** assumptions made before become very important. Assuming that all the features in $\mathbf{x}$ are **mutually independent**, it can be said that:

$$p(x_i \mid x_{i+1}, \ldots, x_n, C_k) = p(x_i \mid C_k) \tag{2.6}$$

Thanks to this, the joint model expression can be rewritten as follows:

$$p(C_k \mid x_1, \ldots, x_n) \propto p(C_k) \prod_{i=1}^{n} p(x_i \mid C_k) \tag{2.7}$$

From the probabilistic model to the classifier

Up until now, only the definition of the probabilistic model lying behind the Naïve-Bayes classifier has been defined, and it can be called the **Naïve-Bayes probability model**. For obtaining the classifier from the aforementioned model, it needs to be combined with a **decision rule**.
In the field of decision theory, the decision rule is a function which is able to define a **mapping** of an observation to a correspondent appropriate action. As it can be clear, this type of rules play a fundamental role in the theory of statistics, and actually can be considered as related to the concept of "strategy" in a game theory.
The effectiveness of a decision rule is always defined through the adoption of a *loss function*, that should specify the outcome obtained from an action under several "states". In the case of Naïve-Bayes, it is the **negative joint log-likelihood** (at least for most of the practical situations).
The rule which is usually applied is to select the hypothesis having the highest probability (i.e. the most probable), exploiting the so called **MAP decision rule**, where MAP stands for "**M**aximum **A P**osteriori". This technique can be used to retrieve a **point estimate** of some quantity, currently unobserved, by basing on data that is empirical.
More in detail, it can be considered as the regularization of the **MLE** (**M**aximum **L**ikelihood **E**stimation), since it shares the same goal. but including an optimization objective that includes also a **prior distribution**. This prior distribution has the function of quantifying the additional information which is available from prior knowledge of a given event.
The prior of a given class can be calculated in two ways:

- by considering classes as *equiprobable*, with $p(C_k) = \frac{1}{K}$;
- by defining an estimate for the probability of the class from the training set.

Thus, to define a proper classifier and estimate the parameters needed for the distribution of a feature, one must consider a distribution or define models that are nonparametric, as specified by [7].

There are three main probability distributions which are often employed in the definition of a Naïve-Bayes classifier:

- **Gaussian (normal) distribution**;
- **Multinomial distribution**;
- **Bernoulli distribution**.

Each one of them aims at solving a particular and well defined problem: for example, the Gaussian distribution is used in those situations in which it is necessary to deal with continuous data, and thus it is assumed that the continuous values that are associated to each of the possible classes are distributed via a Gaussian distribution.
Instead, the Multinomial distribution is adopted whenever there is an interest in considering the **frequencies** with which certain events have been generated via a multinomial $(p_1, \ldots, p_n)$ where p_i is considered as the probability of event i to occur.
For the problem described in this work, the most suitable distribution, in our opinion, is the Bernoulli distribution. This because it permits to describe inputs considering the features as independent binary variables (booleans), keeping the problem discrete.
The Bernoulli probability density function is the following:

$$\begin{cases} q = 1 - p & \text{if } k = 0 \\ p & \text{if } k = 1 \end{cases} \tag{2.8}$$

And can also be rewritten in this way:

$$p^k (1-p)^{1-k} \tag{2.9}$$

Since we have to work with strictly categorical data, the assumptions provided by the Bernoulli distribution highly reflect our use case, and so we decide to take into consideration a Naïve-Bayes classifier using this kind of distribution. In the following chapters we will dig deeper in the estimation process of this classifier.
After all, the main advantage of Naïve-Bayes classifiers is that of being highly scalable, since they can keep their number of parameters linear with respect to the number of variables in the learning problem. Also, they have the "pro" of being able to evaluate their Maximum Likelihood with a closed form expression taking linear time, as specified in [8].

2.2.2 Random Forest classifier

The **Random Forest classifier** is a particular classifier which is based on the Random Forest. With Random Forest (or Random Decision Forest) we refer to a specific family of ML techniques, also known as **ensemble learning** techniques: in them, several different algorithms are used together, in order to achieve higher **predictive performances** with respect to the ones that could be obtained by using only one of the algorithms alone [9].
Here below, we are going to describe briefly the background necessary to understand the working principles of the Random Decision Forest and a bit of its history.

Decision Trees

To understand correctly Random Forests, it is mandatory to introduce first the concept of **Decision Trees**. With decision tree learning we refer to a widespread predictive modeling approach used in ML, which exploits the notion of decision tree for performing a classification/regression task: observations of a given item (that are represented in the branches of the tree) are exploited in order to draw conclusions about the target value of the item under study (that instead is represented in the leaves). In the case of a classification task, the decision tree is also called **Classification Tree**, and each leaf has the role of representing each of the possible labels. Instead, the branches have the task of describing the conjunctions of the features which lead to a given label.
In the field of decision analysis, decision trees are well suited for decision making tasks, since they are able to represent them both explicitly and visually, but they are very often used also in data mining [10].
The objective of a decision tree, in data mining, is to define a model with the ability of predicting the value of a target variable by considering several input variables.
Assuming that, for simplicity, the input features are discrete-valued and that there is a single feature (i.e. the **target feature**) which has to be classified, the decision tree is a tree in which the internal nodes (non-leaf) have the role of representing distinctly the features, while the arcs coming from the mentioned nodes do represent the possible values that can be taken by them. Each arc leads to an adjuvant decision node, and thus to another input feature.

The various leaves of the tree are instead labeled with a probability distribution over the various possible classes or with a class, denoting thus the fact that the data set provided in input is classified into a probability distribution or into a class.
In order to build a decision tree, a **splitting** of the source data has to be performed, so that the root node can be defined and, similarly, the children ones. This operation is based on a set of rules regarding the classification features, as mentioned in [11]. Then, the process is recursively repeated on the obtained subsets, with the **recursive partitioning** approach. When the splitting adds no more value to the classification, the partitioning is interrupted and considered complete.
This entire process is also known as **Top-Down Induction of Decision Trees (TDIDT)**, and is considered as a greedy algorithm [12].
Nowadays, several decision trees algorithms are present, and the most important ones are the following:

- ID3 (that is the short for Iterative Dichotomiser 3);
- C4.5 (which is the successor of ID3);
- CART (short for **C**lassification **A**nd **R**egression **T**ree) [13];
- CHAID (Chi-square automatic interaction detection), which is able to perform multi-level splits in the definition of the classification tree [14];
- MARS, that is an extension of decision tree with the ability of handling better numerical data;
- Conditional Inference Trees, which are based on a statistic approach that exploits non-parametric tests as splitting criteria.

As it has been described previously, the algorithms used for the construction of the decision trees work following a top-down approach, by selecting at each iteration a variable that splits in the best manner the set of items [15]. This concept of *best* can be represented in different ways, depending on the algorithm, by using different **metrics**.
The metrics have the role of measuring the degree of homogeneity of the target variable within the considered subset, and they are in fact computed over each candidate subset. Then, the results obtained are combined (averaged, for example) in order to give a measure of quality of the considered split.
The most used metrics are **Gini impurity** (used by CART), that shouldn't

be confused with the Gini index, and the **Information gain** (used by ID3 and C4.5), which is based on the concepts of entropy and information content.
Decision trees have the advantage of being very easy to use and to understand [16], and this, of course, is good for non-experts so that they can interpret them without struggling.
Also, they are very good in handling both **categorical** and **numerical** data [16]: this is very important, because not all ML algorithms have this peculiarity.
Other very important powers of the decision trees are that they can perform well with large data sets, and also that they don't require a high data preparation.
Instead, for what concerns the limitations, we can find that a small change in the training data could have a big impact on the tree, influencing the predictions made, and making it not very robust.
Another disadvantage is given by the fact that in order to learn completely the tree an NP-complete problem has to be solved, and this also for simple concepts [17]. Thus, in practical cases, heuristics like the greedy algorithm are used for obtaining locally optimal decisions at each of the nodes, without giving the guarantee that the globally optimal tree is returned. To solve this issue, methods such as the **Dual Information Distance (DID)** can be applied [18].
Last but not least, decision trees could have the problem of not generalizing well from the training data and becoming thus very complex, falling into an overfitting problem. Pruning or similar mechanisms could be exploited to avoid this situation.

History

The first method behind the Random Forest was introduced by Tin Kam Ho in 1995 [9]. He stated that forests of trees in which the splitting is performed by using oblique hyperplanes could permit to have a gain in accuracy as they grow without suffering from the overtraining problem, as long as the forests are randomly restricted to be sensitive to the dimensions of the selected features.
Subsequently, other works have been conducted (such as [19]), in which has been concluded that also other splitting methods tend to have the same behavior. This observation that a more complex classifier (e.g. a larger forest) can achieve to higher accuracy in an almost monotonically way is

actually in contrast to the common belief that a classifier at a certain point is hit by overfitting.
The explanation of this resistance has been stated by Kleinberg in his theory of stochastic discrimination [20].
Ho's random subspace selection idea [19] has been very influential in the design of random forests: in fact, in this method, a forest of trees is grown, and the variation among them is computed by performing a projection of the training data into a randomly chosen subspace before fitting each node or tree.
Instead, the idea of randomized node optimization, in which a random procedure is in charge of selecting the decision at each node, has been introduced by Thomas Dietterich in [21].

Ensemble learning

In the field of supervised learning (which is the one under study), the chosen algorithm, in order to perform the classification, must complete firstly the activity of searching through the entire hypothesis space in such a way to retrieve the optimal one able to carry out good predictions for a particular problem, as said in [22].
Even in situations in which the mentioned hypothesis space is made of well defined hypotheses for the problem that has to be solved, it could be tricky to find the correct one. The goal of ensembles is to combine the multiple hypotheses present in the hypothesis space, so that a *better* hypothesis can be obtained.
An ensemble itself can be considered as a supervised learning algorithm, since it could be trained and, later, used to perform classification tasks. In fact, a trained ensemble could be considered as the single hypothesis, which is not necessarily contained in the hypothesis space that we mentioned above. This leads to a high flexibility in the function which is represented: flexibility that could lead, theoretically, to an higher **over-fit** of the training data with respect to the adoption of the single model, but that from a practical standpoint of view could be limited with approaches such as **Bagging** (see later).
With ensembles, the obtained results have the tendency of being better if there is a significant assortment among the adopted models, as said in [23]. Thus, in many ensemble methods, diversity is promoted among the combined

models [24]: but of course, this doesn't the prevent the utilization of the same model more times.

Bagging

Previously, we introduced the concept of **Bagging**. It is the abbreviation for **Bootstrap aggregating**, and it refers to that meta-algorithm created with the goal of improving the accuracy and the stability of a ML algorithm. It is considered as an extension of the **Bayesian Model Averaging (BMA)** approach, which is another algorithm that permits to perform predictions by adopting an average defined over several models, for which weights are defined from the posterior probability obtained in each model given the data [25].
What is performed by the Bootstrap aggregating technique is the following: starting from a training set D, which has size n, m new training sets D_i (of size n') are generated, by performing a sampling with replacement from D in a uniform way.
Since the sampling is completed with replacement, it is clear that some observations can be repeated in the various D_i sets. According to [26], if $n' = n$ and n is a large value, what happens is that the set D_i is expected to have a number of unique examples representing $\approx$63.2% of D, with the rest of the observations being duplicates. The described type of sample can be defined as a **bootstrap sample**.
The adoption of the sampling with replacement permits to secure that each of the generated bootstrap samples is independent from its peers, because of the fact that they do not depend on the samples chosen previously while sampling.
After the definition of the m bootstrap samples, m models (that in the case of a Random Forest are **Decision trees**) are trained using the aforementioned bootstrap samples, and a combination of them is defined by performing an average of the output (in the case of a regression problem), or by performing a majority voting (in the situation of a classification task, i.e. the one of interest).

The classification algorithm can be summarized in the following way:

Algorithm 1 Bootstrap aggregating algorithm.

procedure BAGGING(D, I, m)
    ▷ D is the training set
    ▷ I is the inducer
    ▷ m is the number of bootstrap samples

    ▷ Execution
    for $i \leftarrow 1$ to m **do**
        ▷ Create new training set D_i from D with replacement
        $D_i \leftarrow$ bootstrap sample from D
        ▷ Build classifier C_i from the set D_i, using I
        $C_i \leftarrow I(D_i)$
    end for

    ▷ Classifier C^* is generated using the set of classifiers C_i on D
    $C^*(x) \leftarrow \arg\max_{y \in Y} \sum_{i:C_i(x)=y} 1$ ▷ Get the most often predicted label y
end procedure

This technique has its advantages and disadvantages, which we are going to list here below.

Advantages:

- The fact of having several "weak" learners permits to outperform a single learner over a data set, with a reduced risk of over-fitting the data;
- As proved by [27], bagging helps in removing variance in those data sets called *high variance low-bias*;
- Since each bootstrap can be manipulated on its own before the combination step, the entire algorithm can be executed in parallel [28].

For what concerns, instead, the **disadvantages**, they are the following:

- When dealing with a data set with high bias, bagging could bear this high bias into the aggregates [27];
- The resulting model has a loss of interpretability, because of the intermediary steps that are done;
- It could be very expensive from the computational standpoint of view, depending on the data set taken into consideration.

To summarize, Random Forests have the very big advantage of achieving higher accuracies with respect to single decision trees, and this is one of the main reasons why this model has been also adopted in this study. Also, the nice thing about Random Decision Forests is that they can be actually used as a **black-box** in practical cases, since almost no configuration is required. The only drawback that is introduced regards the loss of interpretability of the model, which has been also mentioned above.

2.2.3 Extra-Trees Classifier

The **Extra-Trees Classifier** is considered as a variant of the Random Forest. In fact, it shares the same structure of the Random Decision Forest, performing an additional step of randomization.
Two differences can be denoted:

- Instead of using a bootstrap sample, each tree of the forest is trained with the entire learning sample;
- The top-down splitting which is performed in the tree learner is randomized.

The randomization mentioned above implies that a *random* "cut-point" is selected, instead of finding the the locally *optimal* one for each of the features under consideration (e.g. Gini impurity or Information gain).
In order to select the value, a uniform distribution within the empirical range of the feature is selected (in the training set of the tree). Then, from all the splits that are generated, the one yielding the highest score is chosen in order to split the node.
As in an ordinary random forest, it can be specified the number of randomly

selected features which have to be examined at each node. This number, for classification tasks, is by default equal to $\sqrt{p}$, while in regression tasks it is usually set to p [29].

Now that we have described the state of the art techniques which can be adopted in order to solve the problem of interest, we are going to describe how we exploited them in such a way to design and implement an incremental solution for it.

Chapter 3

Design and implementation

In order to solve the problem stated in the previous chapter, we decide to design and implement an incremental version for each of the algorithms described.
In this way, the incremental versions can be compared with the standard ones, in such a way to find differences in their behaviors and to determine which of them performs better.
In the following, we want to describe in detail the various steps needed in order to come with a solution, starting from a high-level design of the algorithm and arriving at last at the practical implementations.

3.1 Design

In this section, we want to describe the design process that we faced in order to solve the given problem.
The goal of our design process was to come up with the definition of a high-level algorithm with the ability of being *common* to all the classifiers that we made work in an incremental way, and that we are going to describe in the next sections.
For *common*, we want to intend the capability of the algorithm's structure to adapt to the various types of classifiers that can be used with it. This implies a great **flexibility**, because of the fact that the underlying system

has to be constructed once and can be reused several times just by making little changes in the initialization phase (e.g. the choice of the classifier to use).

3.1.1 Algorithm definition

As introduced in the previous chapters, we want our classifiers to perform the retraining operations only on the subset of data D which is collected in a given amount of time t or that matches a certain amount of information (e.g. a number of collected samples equal to m).
In this way, the adopted classifier doesn't have to reconsider information which has been already processed in previous iterations, but instead analyzes only recent data, with an operation of **partial fitting** (which will be described in a more detailed way later).
From a conceptual point of view, the algorithm we propose is not very articulated or complex, in fact it could be summarized by the following key points:

- Collect an amount of data D such that a time window t is met or a number of samples m is collected;
- Let the adopted classifier C fit on D, via a partial fitting operation: it is important to denote the fact that, at this step, C is being **updated**, so a **single instance** of the classifier is present for the whole process, and, iteratively, it is "tuned" by using the new amount of data. Thus, the knowledge gained by the classifier during the previous iterations is not lost, instead, it is used as a starting point for the new portion of data;
- Evaluate C to monitor its overall performance;
- Repeat the previous steps when conditions are met again.

Please note that this algorithm doesn't act as a streaming or online learning technique: in fact, the retraining process is not performed whenever a new sample of data arrives, but instead it is completed when the given conditions are met, exploiting the well known concept of **batch**.
Of course, to perform the evaluation step, it is necessary to define a test split over which predictions are made when the retraining is performed: we decide to define it in the first iteration of the algorithm.
The described algorithm, as it can be noted, follows a very simple logic,

and this can be considered as an advantage: it aims to solve a problem which maybe is not fundamental at the initial stages of a ML algorithm in a production environment, but that, as time passes, gains more and more importance when a lot of data is collected and has to be reprocessed every time a retraining of the model has to be performed.
We can provide a more detailed description of the aforementioned algorithm by the following:

Algorithm 2 Incremental learning algorithm.

```
    procedure INCREMENTAL(I, m, t)
2:      ▷ I is the inducer used for defining the classifier
        ▷ m is the number of samples needed before retraining C
4:      ▷ t is the time window needed before retraining C

6:      ▷ Initialization
        i ← 0
8:
        ▷ Execution
10:     while there is the need of training the model do
            if i = 0 then
12:             D_0 ← data collected at the beginning
                D_train0 ← training split from D_0
14:             D_test0 ← test split from D_0
                C ← I(D_train0)              ▷ Starting instance of the classifier
16:             Acc_0 ← performance metrics of C over D_test0
            else
18:             if m samples are collected OR t time has passed then
                    D_i ← data collected in m samples or in time t
20:                 C ← I(D_i)               ▷ Update the classifier with new data
                    Acc_i ← performance metrics of the updated C over D_test0
22:             end if
            end if
24:         i ← i + 1
        end while
26: end procedure
```

As it can be observed, it is formulated easily, and it is very simple to be understood: this, for what concerns us, was one of the main objectives of

the work. In fact, we wanted to provide a solution that, at least from a high level, could be interpreted easily even by company figures who do not deal frequently with ML approaches.

In the following sections, we are going to discuss how we implemented it for the various classifiers taken into consideration, describing also the initial conditions with which we had to deal with. But, before doing that, it is necessary to explain the concept of **partial fitting** and how it relates to the various models.

3.1.2 Partial fitting

The concept of partial fitting is one that is gaining more and more importance in ML. The idea lying behind it is the following: there are situations in which estimators have to be trained on very big amounts of data and, as a consequence, it is not always possible to fit the model with the entire load of information.
Thus, a particular learning technique can be adopted: **Out-of-Core learning**. This technique is used whenever the data which the model needs to fit on can't be kept in the computer's main memory, and actually there are three main approaches that can be followed in order to deal with it [1]:

- Extract non important features from instances so that the size of data can diminish;
- Define an incremental algorithm;
- Define a way to stream the instances.

In the case under study we have decided to follow the second option, because of the fact that it is very expensive to maintain an infrastructure able to deal with the instances in a streaming fashion (e.g. as soon as a sample comes, process it). Also, we did not want to define a strong discrimination among the features before the training process.
As it is true that initially, in the given scenario, the amount of available data is very reduced, we are interested in following this approach in order to have

[1] `https://scikit-learn.org/0.15/modules/scaling_strategies.html`

a good scalability for future situations, since it is likely that the amount of data will grow exponentially once that the new platform will be published.
Given the incremental algorithm which has been described previously, we now want to focus on how the fitting operation of the model can be performed partially, and so by only considering a given amount of data.
It is necessary to say that there isn't a unique solution for the partial fitting which can be applied to all the classifiers indistinctly, but instead in each classifier it has to be performed by following approaches that are a bit different.
Here below, we are going to explain how the partial fitting is completed in the various classifiers that we decided to consider as state of the art techniques, and so with the Naïve-Bayes classifier, the Random Forest classifier and the Extra-Trees classifier.

Partial fitting in Naïve-Bayes classifier

As we have explained in chapter 2, we decide to consider in our study a Naïve-Bayes classifier.
Since the data we have to work on is available in a limited amount (see later) and that it is strictly categorical, it is correct to assume that the training data is distributed following a Bernoulli distribution.
In order to understand how the partial fitting can be performed in here, it is necessary to think about how the model performs the predictions. In the previous chapter we stated that they are made by the model exploiting the concepts of **MLE** and of **MAP**.
We want to focus mainly on the first of the two mentioned above. The Maximum Likelihood Estimation for the Bernoulli distribution can be easily computed and can lead to interesting observations, thus we are going to perform it from a theoretical point of view here below.
We recall the definition of the Bernoulli distribution:

$$\begin{cases} q = 1 - p & \text{if } k = 0 \\ p & \text{if } k = 1 \end{cases} \tag{3.1}$$

And also this more compact definition for it:

$$p^k (1-p)^{1-k} \tag{3.2}$$

Since we have to deal with multiple features, the expression becomes the following:

$$\prod_{i=1}^{n} p^{x_i}(1-p)^{(1-x_i)} \tag{3.3}$$

The equation defined above, in very simple terms, represents the **likelihood** function of p, and so we can rewrite it:

$$L(p) = \prod_{i=1}^{n} p^{x_i}(1-p)^{(1-x_i)} \tag{3.4}$$

In order to compute the MLE, it is way easier to reason in terms of **log-likelihood** instead, which is:

$$l(p) = \log p \sum_{i=1}^{n} x_i + \log(1-p) \sum_{i=1}^{n} (1-x_i) \tag{3.5}$$

The MLE, as it can be read from the name, consists of determining the maximum value for the parameter p of the log-likelihood function.
Thus, a partial derivative for p has to be computed:

$$\frac{\partial l(p)}{\partial p} = \frac{\sum_{i=1}^{n} x_i}{p} - \frac{\sum_{i=1}^{n} (1-x_i)}{(1-p)} \tag{3.6}$$

Given that the log-likelihood function is concave, it is enough to set the partial derivative equal to 0 in order to find the value of p:

$$\sum_{i=1}^{n} x_i - p \sum_{i=1}^{n} x_i = p \sum_{i=1}^{n} (1-x_i) \tag{3.7}$$

With a little simplification step, we can easily come up to this:

$$p = \frac{1}{n} \sum_{i=1}^{n} x_i \tag{3.8}$$

As it can be clear, the parameter p results to be simply the **arithmetic mean** of the features.
Thus, it is possible to say that a classifier which is based on this probability distribution, as the Naïve-Bayes classifier chosen by us, needs only the mean of the input features in order to perform its predictions.
This observation permits to state that the classifier can be incrementally trained by updating p in a very simple way: in fact, as it is represented by

the mean, it can be locally stored in pair with the total number of already considered input features, and then adjourned just by looking at the new data points and at the mentioned values.
The tools which we are going to describe and use in the following sections perform this step completely in an autonomous way and with the method outlined above, so it is not necessary to give additional details.

Partial fitting in Random Forest classifier

For what concerns the Random Forest classifier, described conceptually in chapter 2, the topic of partial fitting is quite delicate.
As far as we have been able to verify, in the literature there aren't proper research studies or methods that implement directly the partial fitting operation in a Random Decision Forest.
Instead, there are several tools - one in particular - which try to perform something similar by exploiting particular packages that are usually adopted for parallelization tasks (remember that partial fitting has been firstly defined in the field of out-of-core learning).
More in detail, the tool taken into consideration deals with the partial fitting concept in the following way:

- A Random Forest classifier C is defined, with a given amount of estimators (decision trees) N;
- On the first amount of data which has to be processed, the classifier C is trained by fitting the N estimators on the data, using rather Gini impurity or Information Gain as metric;
- For each of the following data batches, additional N estimators are added to the classifier C: these "new" estimators are the only ones which will be fitted on the data of the most recent batch.

In this way, in the classifier, there will be a number of estimators dedicated to a single batch of data, without needing to have all the estimators of the model to fit on the whole data set.
Of course, a maximum number of estimators has to be defined in order to not have a resulting model which is very slow and big.
The mentioned tool is going to be presented in the section related to the implementation, since it is related to the more "practical" part of the problem.

Partial fitting in Extra-Trees classifier

Since the Extra-Trees classifier is simply a variant of the Random Decision Forest, there are actually no differences with what has been previously stated for the random forests. In fact, in this context, the same tool can be used, and so the partial fitting mechanism adopted in this situation is the same.

Having described the general incremental algorithm and the concept of partial fitting that has to be exploited in order to make it effective, we can now discuss about how it has been implemented from a practical point of view.
In the following, we are going to explain in detail the steps followed in order to come up with a solution, starting from the data retrieval phases and arriving then at the various implementations defined for the classifiers taken into consideration.

3.2 Implementation

For implementing the algorithm introduced in the previous section, we had to perform several intermediate steps. In the following pages, we want to describe all of them exhaustively, starting from a brief description of the used tools.

3.2.1 Used tools

Nowadays, several programming languages could be used for solving ML problems: from C++ to Python, or from R to Java. More and more frameworks are being released for each of them almost regularly, making thus ML accessible to wider audiences.
In this work, we decided to use **Python** because of its simple syntax which leads to a better readability. This is a big advantage, since it permits to people who is not very confident with coding to understand what is being described by the analyzed source code.
Python is an **object-oriented** programming language, and it is available for free in several versions, where each of them has its own peculiarities. In here, we have chosen the version **3.7** because of the fact that it is quite recent (and so more likely to be supported for a longer time) and because of its good compatibility with the packages that we have favored for solving the given task.
Detailed information about it, with the related documentation, can be found at the official website [2].
Also, we have made the decision of using it because of the high offer of ML frameworks with respect to the other programming languages mentioned above: in fact, there are lots of them for almost all of the tasks that we can imagine, from simple ML tasks to more articulated computer vision problems which need to exploit deep learning techniques.
Since the problem under study is a **supervised** learning problem, which we have decided to solve by applying and comparing different algorithms, we needed a framework offering all of them. This framework is well represented by **scikit-learn**.

[2] `https://www.python.org`

Scikit-learn [3] is a very popular open source framework which can be used also commercially (since it is licensed under the BSD license) that offers a very large number of tools for performing predictive data analysis tasks. It is built on top of other very famous Python packages, such as **NumPy** [4] and **SciPy** [5], and the problems which it helps to solve are the following:

- Clustering;
- Classification;
- Regression;
- Model selection;
- Preprocessing;
- Dimensionality reduction.

In our case, we used it for performing the classification tasks, and for evaluating them. It offers a variety of classifiers based on as many algorithms: Naïve-Bayes classifiers, SVMs (Support Vector Machines), Gaussian Processes, Nearest Neighbors, Decision Trees, Random Forests (with their variants) and many many others.
As it can be noted, the classifiers which we wanted to try are all offered in the scikit-learn toolkit, so the choice of the mentioned framework can be considered coherent.
Last but not least important, scikit-learn, for some of its classifiers, offers a direct API for performing the partial fitting operation, which is compulsory for our incremental algorithm.
In our case, the API could be exploited only for the Naïve-Bayes classifier, since it is not available yet for the Random Forest classifier. In fact, for the latter, we had to rely on another package which we are going to describe in the following pages.

Another tool that we have decided to adopt is **pandas** [6].
Pandas is an open source tool which is built on top of Python, and that

[3] `https://scikit-learn.org/stable/`

[4] `https://numpy.org`

[5] `https://www.scipy.org`

[6] `https://pandas.pydata.org`

offers several ways to manipulate and analyze data in a very efficient way.
Among its interesting features, it can be found that it provides a particular object called **DataFrame** which permits to handle data exploiting integrated indexing, and so in an efficient and, most importantly, fastly.
It offers interesting approaches exploiting in-memory structures for reading and writing data, supporting also a wide variety of formats, such as CSV files, SQL databases or even the HDF5 format.
Among its high number of offered functionalities, it is important to denote the ability of joining and merging data sets keeping a high performance, and also the proficiency in performing operations such as slicing, subsetting or even indexing data sets with a large size. In it, code paths that are considered critical are written in a lower level language like C, which underlines the fact that it has been developed in order to maintain standards of high performance.
We employed it primarily for manipulating the different data sources representing the customers' information and the restaurateurs' information, in such a way to join them correctly and define properly the inputs for the various classifiers.

As mentioned previously, the partial fitting API offered by scikit-learn is not available yet for all the classifiers. Among the excluded classifiers, we can find the Random Forest classifier.
Because of this, we had to search for alternatives offering something similar, and we luckily found **IncrementalTrees**.
IncrementalTrees is an open source library, available on GitHub [7], which can be used for commercial purposes, as it has a MIT license.
The goal of this library is to provide the partial fitting API offered by scikit-learn for the Random Forest classifier and for its variants (such as the Extra-Trees classifier), in order to allow incremental training also in them.
As it is stressed from the authors of the library, it doesn't directly implement partial fitting for single decision trees (e.g. for Decision Tree classifiers), rather it removes the requirement that in a Random Decision Forest the individual decision trees are always trained with the same data.
This approach, generally, requires that the number of weak learners is increased (probably), but it "reduces memory burden, training time and

[7] `https://github.com/garethjns/IncrementalTrees`

variance", the authors claim.
For achieving this, the mentioned library exploits another library, which is **Dask** [8].
Dask is a library born for performing *parallel computing* tasks in Python. Among its interfaces, it provides scalable ML by working at the side of other libraries, such as scikit-learn.
In fact, it tries to provide a single unified interface around familiar APIs of scikit-learn, pandas and NumPy, in such a way that whoever is familiar with them can easily use also this.
A nice feature which is offered by the library is the ability of dealing with IL tasks. As a matter of fact, it provides a bridge between scikit-learn and Dask by wrapping the needed scikit-learn estimator in its **Incremental** meta-estimator.
A meta-estimator is an estimator which takes in input another estimator, and in this case it permits to use the concept of Dask Arrays wherever a scikit-learn estimator expects NumPy arrays [9].
Coming back at the **IncrementalTrees** library, it provides incremental versions of the following models:

- Random Forest classifier;
- Extra-Trees classifier;
- Random Forest regressor;
- Extra-Trees regressor.

In this work, we have used the first two.

The last library that we used was **matplotlib** [10].
Matplotlib is a library that permits to create static, animated, and also interactive visualizations in Python. It is an open source project, of which the first version has been implemented by John Hunter [30].
We adopted it extensively in the experimental validation phase of our work, which is described in a detailed way in the next chapter.

[8] `https://docs.dask.org/en/latest/`

[9] `https://ml.dask.org/incremental.html`

[10] `https://matplotlib.org`

After having introduced the various tools that we have used, we can now describe all the steps that we had to face in order to implement the designed IL algorithm from a practical point of view.

3.2.2 Data retrieval

The first operation that was needed to be performed was the retrieval of the data necessary for training the model.
As it has been described in chapter 2, the data related to customers and restaurants is collected via ad hoc forms made available to the platform users through Google Forms. Even if it could seem easy to retrieve the gathered data from the filled in forms and work on it, this was not possible.
This is because of the fact that, up until now, the forms adopted by the company are still referring to an old structure which it is planned to be dismissed in the future, and to be substituted with another configuration which collects all the information described in the statement of the problem.
Thus, we had to find a workaround and behave in a different way: in fact, we committed to generate "fake" data instances using mock information. This can be also justified by the fact that the pandemic which we are still facing had a huge impact on the job market in the catering world: there has been a significant decrease in the offer from restaurateurs, being the restaurants closed for very long times and constrained to follow the various preventive rules given by the Ministry of Health.
Hence, we now want to describe entirely the process with which it has been possible to generate a fictitious data set able to represent the real situation of the platform.

3.2.3 Data generation

As it has been made clear by the company with which the work has been conducted, the available amount of data was low and there weren't the conditions for which it could have become accessible in a short time.
For this reason, we agreed to proceed with the generation of a fictitious data set able to represent the typology of information which the platform would have been ready to collect in the future.
To do this, the first thing we did was to consider all the needed features by the platform (and thus by the model), by distinguishing what was demanded for the candidates, and what for the restaurateurs.
They are listed in the problem statement, but in order to be more clear, we are going to indicate them also in here.
Needed features for the candidates:

- **Sector**;
- **Time availability**;
- **Workshift**;
- **Premise**;
- **Start Time**;
- **Transport type**;
- **Transfers availability**;
- **Gender**;
- **Address**;
- **Name**;
- **Surname**;
- **Birth date**.

Instead, for the restaurateurs:

- **Restaurant name**;
- **Referent**;

- **Address**;
- **Management type**;
- **Restaurant type**;
- **Cuisine type**;
- **Customer type**;
- **Number of employees**;
- **Average served people**;
- **Average price**;
- **Opening days**;
- **Workshifts**;
- **Contact**;
- **E-mail**;
- **Telephone**.

First of all, we decided to generate the data sources separately. So, we devoted a Python script to the candidates generation, and another script to the formation of the information related to restaurateurs.

Constant values

Starting from the candidates, we defined some constant data structures in order to represent the possible values that the various features can take. For example, we designated a constant set of italian common names to be assigned to the applicants, differentiated by gender (e.g. *["Mario", "Luca", "Luigi", ...]* for males and *["Sofia", "Giulia", "Aurora", ...]* for females), and a constant set of common surnames, like *["Bianchi", "Rossi", "Verdi", ...]*. For the attributes that we consider *strong*, and so those attributes that the applicants are forced by the platform to choose from a given set of possibilities, we had to deal with all the possible values, which are the following:

- **Sector**: *["Saloon", "Kitchen", "Cafe"]*;
- **Time availability**: *["Full", "Part", "Call", "Weekends"]*;
- **Workshift**: *["Breakfast", "Lunch", "Dinner"]*;
- **Premise**: *["FastFood", "Risto-pub", "Pizzeria", "Farmhouse", "Diner", ecc.]*;
- **Start Time**: *["Now", "NextWeek", "NextMonth"]*;
- **Transport type**: *["Car", "Motorbike", "Bike", "PublicTransport", ecc.]*.

For the **Transfers availability** attribute, which is provided through a *yes/no* option, we decided to use simple a boolean flag. We adopted a similar approach for the **Gender** feature of the candidate.
For what concerns the **Address**, we determined also in this case a constant data structure listing the most important cities in Italy, like *["Torino", "Milano", "Venezia", "Roma", "Bologna", ecc.]*. Because of the fact that, in any case, the platform used in production will perform a filtering task *a priori* by considering candidates and restaurants in the same city, we made the choice of considering only the 10 most common cities in Italy. This has been done mainly for the sake of simulating in a more accurate way the real scenario.
For each city, we created a CSV configuration file listing the most known streets of the city represented by the file. This has been done in order to declare a sort of helper function able to generate addresses which are some sort of real, at least for the pair *city-street name* (see later).
Instead, for defining birth dates having sense, we chose to set thresholds for the year, represented by the inclusive range [1965, 2000]. To obtain valid birth dates, we made the decision of using only days from 1 to 28, which are present for all the months of the year.
By the way, the birth date is not one of the attributes that is used for matching a given candidate with a restaurateur, so, like others, it has been generated for being the most faithful as possible to the real situation.

For the restaurants, we followed a very similar approach, with some small changes. In fact, for the attribute **Restaurant name**, we wanted to keep the data generation process very simple, so we simply defined a constant prefix for it, such as *"Restaurant"*. In the generation phase it has been exploited and paired to an index - different for each restaurant entry - in order to have

distinct names.
The exact same thing has been done for the **Referent** feature, by using instead as prefix the constant value *"Restaurateur"*.
The **Address** attribute has been managed in the same manner described for the candidates.
Also in here, for those attributes that we consider **strong**, we decided to adopt constant data structures aiming at keeping all the possible values for each of them.
We recall the entire set of possible values:

- **Management type**: *["FamilyRun", "SoleProp", "Franchise"]*;
- **Restaurant type**: *["Cafe", "Brewery", "FastFood", "Starred", ecc.]*;
- **Cuisine type**: *["Homemade", "Gourmet", "Minimal", "Piedmontese", "Vegetarian", ecc.]*;
- **Customer type**: *["Young", "University", "Office", "Families", "Informal", ecc.]*;
- **Number of employees**: *["Less5", "5And10", "10And15", "15And20", "More20"]*;
- **Average served people**: *["Less30", "30And60", "60And100", ecc.]*;
- **Average price**: *["Less15", "15And20", "20And30", "More30"]*;
- **Opening days**: *["Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday", "Sunday"]*;
- **Workshifts**: *["Breakfast", "Lunch", "Dinner", "Pub"]*;
- **Contact**: *["Mail", "WhatsApp", "SMS", "Phone", "Facebook"]*

For the **E-mail** attribute, as we did not want to use real addresses, we simply defined a common suffix *"@mail.com"* to be preceded by the **Restaurant name** value while generating data, in order distinct values also for this feature.
We behaved similarly with the **Telephone** attribute, by designating a constant prefix like *"+393310123"* to be followed by a unique index associated to each restaurant entry.

Now that we have explained the basic configuration which we exploited for generating the fictitious data, we are going to describe how the two mock data sources have been generated.

Generation

As we have mentioned previously, we made the decision of dedicating a script for the generation of data related to the candidates and another one for the information of the candidates.
We followed a very similar approach in both of them, and now we want to describe it, starting from the generation of the candidates.
What we had in mind was to obtain a file in CSV format containing all the information related to the applicants. To do this, we simply used the **filereader** and **filewriter** APIs provided by default in Python.
Since the type of data that we had to deal with - as it can be noted - was strictly categorical, we wanted to assume a Bernoulli probability distribution for the features. In order to do this, we had to format the CSV header properly.
In fact, we desired to have a data set which could have been used directly with all the classifiers that we wanted to test. As the RF classifier and the ET classifier (in both standard and incremental versions) don't have strict requirements in how - intending the structure - data should be provided to the model, some constraints have to be respected for the input data in the case of the **BernoulliNB** Naïve-Bayes classifier offered by scikit-learn [11].
That particular classifier requires that the data provided in input is formatted only through **binary features**. So, we agreed in defining a data set in which all the attributes are expressed with binary values, and so with values equal to 0 or equal to 1.
For achieving this situation, we prepared the CSV file header by defining one column for each possible value of each categorical (or strong) feature described previously. Conceptually, something like this:

- **Sector** -> columns *Sector_Saloon, Sector_Kitchen, Sector_Cafe*;

[11] `https://scikit-learn.org/stable/modules/generated/sklearn.naive_bayes.BernoulliNB.html`

- **Time availability** -> columns *Time_Full, Time_Part, Time_Call, Time_Weekends*;
- **Workshift** -> columns *Workshift_Breakfast, Workshift_Lunch, Workshift_Dinner*;
- **Premise** -> columns *Premise_FastFood, Premise_Risto-pub, ecc.*;
- **Start Time** -> columns *Start_Now, Start_NextWeek, ecc.*;
- **Transport type** -> columns *Transport_Car, Transport_Motorbike, Transport_Bike ecc.*

For the attributes **Transfers availability** and **Gender**, as we mentioned previously, we used boolean flags, which in this context have been translated into the columns *Transfers_Yes, Transfers_No* for the first, and the columns *Gender_Male, Gender_Female* for the second.
The column for the **Address** remained *Address* without the need of being "separated", since it did not represent a categorical feature.
Having defined all the needed columns for all the categorical attributes, it has been possible to define the complete header for the desired CSV file representing the candidates data source: this has been done by exploiting the filewriter API briefly introduced in the previous page.
After the definition of the header, data had to be finally generated. In order to not dedicate a very long time for this task, and to mimic the real situation of the system for which the available amount data was limited, we decided to create a number of candidates equal to 150.
This has been done simply by keeping the CSV file open, and perform 150 iterations in the open stream. In each iteration, we set up a random value (or a set of random values) for each of the categorical features. The value - or values, in case of multiple choices available for an attribute - has been used to extract one or more options from the constant data structure representing the set of possibilities for a feature.
After the options have been extracted from the constant data structures, we had to map them properly to the various columns of the header. For doing this, we implemented a helper function for each attribute, and in it we used **dictionaries** where the keys were represented by a header (e.g. *"Premise_FastFood"*) and the values by the binary 0 or 1.
With a little portion of code, we checked the extracted value and we provided a suitable mapping for the CSV file.

A little exception was made for the **Transfers availability** and the **Gender**. For the first, we selected a random number r between 0 and 1, leading to the following consequences:

1. if $r = 0$: $Transfers\_Yes = 0$ and $Transfers\_No = 1$;
2. if $r = 1$: $Transfers\_Yes = 1$ and $Transfers\_No = 0$.

Instead, for the second, we did something different. As a matter of fact, for the sake of simplicity, we decided to perform a check on the current index i of the iteration:

1. if i was even: $Gender\_Male = 1$ and $Gender\_Female = 0$;
2. if i was odd: $Gender\_Male = 0$ and $Gender\_Female = 1$.

The operation described above permitted to have an equal number of males and females, and also made possible to generate correctly the names and surnames for the candidates: by checking the value of i, we had been able to know if we had to extract a random name from the male ones, or from the female ones.
In order to not come up with duplicate names, we designed an appropriate function checking the existence of a given name in the available collection, that when finds an occurrence it takes in charge the re-generation of the *name-surname* pair.
Previously, we mentioned that in order to generate the **Address** we used a configuration CSV file for each city available in the data structure representing them.
More precisely, we defined a file for each city, containing the most common streets of the related city and the various neighborhoods in which they are located. From a helper function, we build a dictionary containing all the associations *city-neighborhood-street*, and from another we permit to extract a valid address.
In this way, it was possible to assign an address to a candidate.
The **Birth date**, instead, was simply created by extracting three random numbers:

- one in the inclusive range - mentioned in the previous pages - describing the thresholds for the birth years;
- one in the inclusive range [1,12], for defining a month;

- one in the inclusive range [1,28], for selecting a day available in all the months and avoid invalid dates.

After the extraction of all the needed information from the various data structures, it has been possible to finally write an entry representing a candidate in the CSV file. This was the last step for the creation of the fictitious candidates, so we can now focus on the generation of the restaurateurs.

For the restaurateurs - as mentioned - we decided to follow a similar approach. In fact, the first thing that we did was to define properly the header of the CSV file, in such a way to be used directly with all the classifiers under study.
The header definition mimed the one previously described for the candidates, and it has been done again by defining a column for each possible value of each categorical feature. And so, in the following way:

- **Management type** -> columns *Mgmt_Family, Mgmt_SoleProp, ecc.*;
- **Restaurant type** -> columns *Rest_Cafe, Rest_Brewery, ecc.*;
- **Cuisine type** -> columns *Cuisine_Homemade, Cuisine_Gourmet, Cuisine_Minimal, ecc.*;
- **Customer type** -> columns *Cust_Young, Cust_University, ecc.*;
- **Number of employees** -> columns *Emp_Less5, Emp_5And10, ecc.*;
- **Average served people** -> columns *People_Less30, People_30And60, ecc.*;
- **Average price** -> columns *Price_Less15, Price_15And20, ecc.*;
- **Opening days** -> columns *Opening_Monday, Opening_Tuesday, ecc.*;
- **Workshifts** -> columns *Workshift_Breakfast, Workshift_Lunch, ecc.*;
- **Contact** -> columns *Contact_Mail, Contact_WhatsApp, ecc..*

The columns related to the non-categorical attributes like **Restaurant name**, **Referent** and **Address** did not need to be decoupled as the previous ones.
At that point, we have been able to write the correct header in the file.
Also in here, for saving time and staying loyal to the real scenario, we decided to create a number of restaurateurs equal to 150. The generation of these entries for the CSV file related to the restaurants information has been performed with the same methodology of the other data source: so, we iterated 150 times in the open file stream.
In each iteration, denoted by the index i, we again extracted a random number for each categorical feature. These numbers - as in the case of the candidates - are used to get one or more possible values from the constant data structures representing the possible choices for the attributes, simply by accessing them by index.
Again, we exploited a helper function to perform a mapping among the extracted values and the header columns, in the same way described for the other data source.
For what concerns the **Restaurant name** and the **Referent**, we directly used the iteration index i as a suffix for the various prefixes mentioned in the preceding pages, like in the following:

- $i = 0$ (first entry) -> *Restaurant1* and *Restaurateur1*;
- $i = 5$ (sixth entry) -> *Restaurant6* and *Restaurateur6*.

The **Address** for the restaurant was retrieved with the same procedure explained for the other data source, while the **Telephone** and the **E-mail** have been again defined by taking advantage of the index i and the predefined prefix and suffix:

- $i = 0$ (first entry) -> *restaurant1@mail.com* and *+393310123001*;
- $i = 99$ (entry n°100) -> *restaurant100@mail.com* and *+393310123100*.

At this point, it has been conceivable to have a data source able to represent also the restaurateurs.

Having the two needed data sources, the next step that we had to deal with was the generation of the **labels** to be used in the training and evaluation phases of the model. We are now going to describe how this has been done.

Defining the labels

In order to train - and also to test - supervised learning models, it is necessary that the used data set contains **labels**.
In the situation under study, a label should be assigned to each *candidate-restaurateur* couple. We decided to call this label **Affinity score**, and we assumed the following possible values for it:

- $y = -1$ for denoting a **"no-match"** situation: the candidate and the restaurateur are not compatible at all;
- $y = 0$ for denoting a **"possible match"**: the candidate and the restaurateur have a slight compatibility;
- $y = 1$ for denoting, instead, a **"match"**: the candidate and the restaurateur are compatible.

The situation represented by the affinity score $y = 0$ has the aim of assigning some candidates to those restaurants for which, in the real case, there won't be any matches.
Since the number of labels is higher than two, this supervised learning problem is considered to be a **multi-class** one.
The process of assigning the affinity scores to the data sources generated in the previous steps has been completed with the help of the company. They cooperated with us by analyzing a part of the couples *candidate-restaurateur*, looking at the various attributes in order to find a degree of affinity.
For saving time, it has been made the decision to define manually only a number of affinity scores equal to 25, for each candidate.
The 25 restaurateurs for which the score has been provided faithfully have been chosen iteratively, meaning that, in the case of the first candidate, the labels were provided for the first 25 restaurants, while for the second candidate the scores were given from the 26th restaurateur to the 50th, and so on.
Initially, for the 125 restaurateurs not considered with a candidate, we used as default label $y = 0$. Anyway, this proved to be inappropriate: the resulting set of scores had a vast majority of "possible matches", influencing in a negative way the models.
Thus, for each candidate, we decided to keep the scores related to the associations with 25 restaurateurs, and no more.
We made the decision of keeping these affinity scores in an appropriate CSV file, declared with the following column headers:

- *Candidate_Id*;
- *Restaurant_Id*;
- *Affinity_score*.

Once that the affinity scores have been defined, we had the need of building properly the actual data set to be used as input.
We are now going to describe what we did in order to achieve this goal.

Merging the data sources

As we have described in the previous pages, we arrived at a point in which we had three data sources:

1. A data source for the information related to **candidates**;
2. A data source for the information related to **restaurants**;
3. A data source for the **affinity scores**.

In order to have a consistent data set, the mentioned sources had to be merged properly, in such a way to have entries representing the following information:

- $Candidate_i$, $Restaurant_j$, $Score_{ij}$

For doing this, we exploited **pandas** and its functionalities.
More in detail, we started by copying the content of the various CSV files representing the sources into three separate **DataFrame** objects, one for each source. Then, we performed some additional operations on them.
Starting from the DataFrame related to the candidates, we decided to remove those attributes that we did consider as not fundamental for the matching, which are:

- **Address**, because of the a priori filtering performed by the application's back-end;
- **Name**;
- **Surname**;

- **Gender**, represented by the two columns *Gender_Male* and *Gender_Female*;
- **Birth date**.

More precisely, we established that the matching between a candidate and a restaurateur should be performed by taking into account only those attributes that can be considered as sort of common between the sources, avoiding thus a useless and wrong overhead.
We did a similar thing in the DataFrame related to the restaurateurs, by removing the following features:

- **Restaurant name**;
- **Referent**;
- **Address**, also here because of the a priori filtering;
- **E-mail**;
- **Telephone**.

Then, we performed a first merge between the candidates and the restaurants, by paying attention to respecting the structure outlined in the data source related to the affinity scores.
This has been done by defining a fake pivot attribute on the two DataFrame instances, and by using the **merge** API offered by pandas. Afterwards, the result of the merge has been written to a single DataFrame exploiting the **concat** functionality.
Of course, several checks have been made to guarantee consistency with the arrangement of the scores and, at merge completion, the pivot attribute has been dropped.
At this point, the labels were still missing in the DataFrame.
For that, we removed first the useless information from the DataFrame related to the affinity scores, such as the *Candidate_Id* and the *Restaurant_Id*: subsequently, we again exploited the concat functionality to have a single DataFrame representing the entire data set.
Finally, we exported the DataFrame to a CSV file.

Having the complete data set, we have then been able to implement the IL algorithm - with the various incremental models - and the standard approaches, to be used as a benchmark.
In the following pages, we want to discuss about how this has been done.

3.2.4 Incremental models

In order to implement the IL algorithm described in the previous section, the needed code was not much.
Starting from the very beginning, the first thing we did was to divide the entire data set into a **training** and a **testing** portion. We decided to use for training the 80% of the original data set, and thus the 20% for testing.
Since the platform was offline at the time of this work, we had the need of simulating an environment in which new - or unseen - data was available, such that the various models could be trained with an incremental approach.
For doing this, we chose to define a variable called **split factor**.
The **split factor** had the goal of splitting the training data set into **chunks**, to be used iteratively as new samples for our models. In this way, it was at least possible to simulate the satisfaction of the IL algorithm's condition related to the *collection of m samples*.
For example, knowing that the original data set had information for 150 candidates, and that for each candidate we had a number of restaurants associated equal to 25, it is evident that the length - in terms of number of entries - (or cardinality) of the data set was $150 * 25 = 3750$.
Keeping as training data set the 80% of it, it can be said that 3000 samples were available for performing the incremental training.
Assuming a split factor $s = 6$, for example, it was possible to define six chunks of 500 samples each.
In the chapter related to the experimental validation, we assumed different values for the split factor, in order to compare the various results and to find the most appropriate one for the needs of the company.

Coming back at the implementation aspect, once that we defined the approach for simulating the arrival of new data to the system, the IL algorithm was pretty straightforward to implement.
In that context, the initial thing to do was to instantiate the incremental model to be used.

As mentioned before, we considered three different models:

- **BernoulliNB** as the NB classifier provided by scikit-learn;
- **StreamingRFC** as the RF classifier provided by the IncrementalTrees library, which is an extension of the **RandomForestClassifier** available in scikit-learn;
- **StreamingEXTC** as the ET classifier provided by the IncrementalTrees library, which is an extension of the **ExtraTreesClassifier** available in scikit-learn.

For the BernoulliNB, we chose to not set any particular parameter in the definition of the model, keeping it thus with its default configuration.
Instead, for the other two classifiers based on the RF, we had to provide the following parameters:

- **n_estimators_per_chunk**, which defines the number of estimators that will be learned during the training over a chunk;
- **max_n_estimators**, that sets a threshold for the number of learners in the forest. It can be also set to NumPy's infinite.

Also for the **n_estimators_per_chunk** we tried out several values, which will be listed in the next chapter. Instead, for the **max_n_estimators** we assumed infinite, since we did not want to give a constraint on the forest's size, letting it instead grow on its own.
With the models defined, we have then been able to implement the IL algorithm simply by looping s times.
In each iteration i, we retrieved a different chunk of data D_i, and we used it to partially fit the models. To fit partially a model, both scikit-learn and IncrementalTrees provide a **partial_fit** API: it simply requires in input the set of features - denoted by X - and the set of labels - denoted by y - of the current chunk.
After the completion of the partial fitting in the iteration, we computed the performances of the models at the step i, by predicting the labels for the testing set previously defined. For this task several metrics have been considered, which will be explained in detail in the following chapter.
At the end of the loop, it was possible to consider the IL algorithm as implemented.

What we did later was to implement the standard models, in order to have a comparison and to draw some conclusions about the obtained results in the incremental versions.

3.2.5 Standard models

As already said, for being able to state the quality of the obtained incremental models, we decided to implement also their standard versions.
In order to do this, we simply considered the entire data set, without splitting it in chunks. But, as in the incremental implementation, we derived from it a **training** set and a **testing** one, with proportions respectively equal to the 80% and the 20% of the original set.
Then, we instantiated the following standard models:

- **BernoulliNB** as the standard NB classifier;
- **RandomForestClassifier** as the standard RF classifier;
- **ExtraTreesClassifier** as the standard ET classifier.

All the models mentioned above are provided by scikit-learn.
As in the incremental implementation, the BernoulliNB has been defined without any parameter, keeping thus its default configuration.
For what concerns instead the RandomForestClassifier and the ExtraTreesClassifier, the following parameters have been set in their definition:

- **n_estimators**, representing the total number of estimators present in the forest;
- **criterion**, representing the metric used by the forest in order to determine the best attribute for performing the splitting (e.g. *'gini'* for the Gini impurity, or *'entropy'* for the Information Gain).

In order to train them, it has been enough to use the **fit** function, which behaves in the same manner of the partial_fit used for the incremental models: in fact, it takes in input the set of features X and the set of labels y.
Then, after the completion of the training, we let the various models perform the predictions on the test data set, in order to evaluate their performances.
At that point, the entire system has been considered ready for its experimental validation, which will be described later.

Having described the entire implementation process that we had to face, from the data retrieval to the accomplishment of the incremental models, we now want to give a short review about the code has been organized.

3.2.6 Code structure

Earlier, some hints regarding the structure of our code have been given.
In here, we want to dig deeper and provide a complete overview of how the various portions of code have been organized.
We destined a Python script for each data source to be generated: thus, we had a script for the candidates and one for the restaurants.
Each of them relied on some helper functions. We decided to have all of these functions in a unique place, so we had also a script dedicate to them only.
The various functions, on their own, relied to some CSV configuration files, like in the case of the streets and neighborhoods of a city. All of these files have been kept in other folders outside the project, separated properly by city name.
Then, for merging the two data sources - three if we want to consider also the file containing the labels - we used again a separate script, which provided at output the complete data set in CSV format.
Finally, we dedicated one Python script for the IL algorithm implementation, and another one for the standard one.

Now that the design and the implementation of our solution to the problem stated in chapter 2 have been explained, we can focus on its experimental validation.
In the next chapter, we are first going to list and describe all the performance metrics that we used, and then we are going to show the obtained results.

Chapter 4

Experimental validation

After having implemented the IL algorithm, it was needed to establish its performance by comparing it to its standard counterpart. In order to do this, and to obtain the best configuration possible for the models, we measured the quality by adopting several performance metrics.
In this chapter, we want to explain in a detailed manner the chosen metrics, and then we want to show the results obtained with the various configurations.

4.1 Metrics

As introduced, we first want to list and describe the metrics that we have chosen for drawing our conclusions about the solution provided for the problem stated in chapter 2.
We decided to use the following performance metrics:

- **Confusion matrix**;
- **Accuracy**;
- **Precision**;
- **Recall**;
- **F1-score**;
- **Macro averages**;
- **Weighted averages**;

- **Execution time**.

The choice of considering a number of metrics this high was mainly due to the fact that we wanted to have a broader knowledge of what was happening under the hood in the various models: this permitted us to establish clearer if, for the problem under study, the incremental approaches were better than the standard ones.
We now want to focus on each metric, describing it in an accurate way.

4.1.1 Confusion matrix

The confusion matrix is in absolute the most important metric in the field of ML. From it, in fact, it is possible to derive almost all of the other metrics listed above.
This matrix has been also defined as **Error matrix** by Stephen Stehman in [31], and it is provided through a specific table design in order to visualize the performance of a classification algorithm.
In it, each row has the role of representing the number of instances *predicted* for a given label, while each column acts as a depiction of the number of *actual* instances of a given label.
Also, it can be seen as a special version of contingency table.
In a binary classification task, its layout would be the following:

Label y	$y = 1$	$y = 0$
$y = 1$	TP	FP
$y = 0$	FN	TN

Table 4.1: Confusion matrix layout in a binary problem

From this basic layout, four values can be differentiated:

- TP **(True Positives)**: it represents the number of actually positive samples that have been predicted as positive by the classifier, making thus a *correct* classification;

- FP **(False Positives)**: it represents the number of actually negative samples that the classifier has predicted as positive, making a *mistake* in the classification;

- FN **(False Negatives)**: it represents the number of actually positive samples that the classifier has *wrongly* predicted as negative;
- TN **(True Negatives)**: it represents the number of actually negative samples that the classifier has *correctly* predicted as negative.

In the case under study, that is a multi-class classification problem, we imagined the confusion matrix structure to be the following:

Affinity score y	$y = 1$	$y = 0$	$y = -1$
$y = 1$	T_1	F_{10}	F_{1-1}
$y = 0$	F_{01}	T_0	F_{0-1}
$y = -1$	F_{-11}	F_{-10}	T_{-1}

Table 4.2: Confusion matrix layout in the multi-class problem

From the table above, instead, we can derive these values:

- T_1: it represents the number of **"match"** situations *correctly* predicted by the classifier taken into consideration;
- F_{10}: it represents the number of **"possible match"** situations for which the classifier has *wrongly* predicted a **"match"**;
- F_{1-1}: it represents the number of **"no match"** cases for which the classifier has *wrongly* predicted a **"match"**;
- F_{01}: it represents the number of **"match"** situations for which the classifier has *wrongly* predicted a **"possible match"**;
- T_0: it represents the number of **"possible match"** situations that have been *correctly* predicted by the classifier;
- F_{0-1}: it represents the number of **"no match"** situations for which the classifier has *wrongly* predicted a **"possible match"**;
- F_{-11}: it represents the number of **"match"** situations for which the classifier has *wrongly* predicted a **"no match"**;
- F_{-10}: it represents the number of **"possible match"** situations for which the classifier has *wrongly* predicted a **"no match"**;

- T_{-1}: it represents the number of **"no match"** situations that have been *correctly* predicted by the classifier.

From this layout, almost all of the other metrics can be derived. In the following sections, we want to explain how.

4.1.2 Accuracy

The accuracy is a performance metric that is directly derived from the confusion matrix.
Probably, it is one of the simplest that can be found, and it simply represents the ratio of the number of samples *correctly predicted* by the classifier over the *total number* of samples.
Keeping in mind the values of the confusion matrix described in the previous section, it can be said that the equation for computing the accuracy of binary classifier is the following:

$$Accuracy = \frac{(TP + TN)}{(TP + FP + FN + TN)} \tag{4.1}$$

As it can be seen, the sum $(TP + TN)$ represents the total number of samples correctly classified, while $(TP + FP + FN + TN)$ is simply the totality of samples.
Instead, by looking at the confusion matrix for the multi-class problem, the accuracy equation becomes the following one:

$$Accuracy = \frac{(T_1 + T_0 + T_{\text{-}1})}{(T_1 + F_{10} + F_{\text{1-}1} + F_{01} + T_0 + F_{\text{0-}1} + F_{\text{-}11} + F_{\text{-}10} + T_{\text{-}1})} \tag{4.2}$$

This one has been used in the current study.
As comfortable and good as it can seem, the accuracy metric is not always a pure source of truth.
With highly unbalanced data sets, for example, it could easily happen that the classifier learns to predict only the most frequent label, without knowing absolutely nothing for the other/s.
In a situation like this, the accuracy would be high, but actually the classifier would perform really bad, by predicting always the same label.
That's why it is important to consider also other performance metrics, especially when the used data set is not perfectly balanced.

4.1.3 Precision

Precision is another very common metric.
It is also called **Positive Predictive Value** and, in a classification task, it has the goal of representing the ratio between the number of correctly predicted positive items - i.e. the *true positives* - and the total number of elements that the classifier predicted as positive - i.e. the sum of *true positives* and *false positives*.
In the binary case, its formula can be written in the following way:

$$Precision = \frac{TP}{(TP + FP)} \tag{4.3}$$

In this way, it is possible to tell how many samples are really positive from all the ones predicted as positives by the classifier.
In a multi-class problem, the computation of this metric should be performed for each one of the classes.
Thus, in this study, there would be the need to compute three separate precision values, which we decide to call $Precision_1$, $Precision_0$ and $Precision_{-1}$.
$Precision_1$ is the precision which is computed for the class $y = 1$, and so for the **"match"** situations. The following equation is used:

$$Precision_1 = \frac{T_1}{(T_1 + F_{10} + F_{1-1})} \tag{4.4}$$

As it can be seen, the denominator is simply the sum over the first row related to the label $y = 1$ of the confusion matrix.
$Precision_0$ is the precision computed for the affinity score $y = 0$, and so for the **"possible match"** cases.

$$Precision_0 = \frac{T_0}{(T_0 + F_{01} + F_{0-1})} \tag{4.5}$$

Finally, it is possible to write also the equation for $Precision_{-1}$, which instead is related to the **"no match"** situations.

$$Precision_{-1} = \frac{T_{-1}}{(T_{-1} + F_{-11} + F_{-10})} \tag{4.6}$$

4.1.4 Recall

The Recall, also known as **Sensitivity** or **True Positive Rate (TPR)** is a very important metric.
It has the goal of representing the ratio between the number of positive samples that are correctly classified by the classifier - i.e. the *true positives* - and the total number of actually positive samples - i.e. the sum of *true positives* and *false negatives* -.
In a binary scenario, this translates into the following formula:

$$Recall = \frac{TP}{(TP + FN)} \tag{4.7}$$

In this way, it is possible to establish how many samples are correctly classified as positives from the actual positives.
Also here, for the multi-class problem, it is necessary to compute a recall value for each one of the classes: in our case, we defined $Recall_1$, $Recall_0$, and $Recall_{-1}$.
$Recall_1$ is the recall value related to the affinity score $y = 1$, and so for the **"match"** cases. It is driven by the following equation:

$$Recall_1 = \frac{T_1}{(T_1 + F_{01} + F_{-11})} \tag{4.8}$$

As it can be seen, the denominator is represented simply by the sum of the values over the column related to the label $y = 1$ in the confusion matrix.
$Recall_0$, instead, is the recall value for the class $y = 0$, that represents the **"possible match"** situations.
It is computed with the following equation:

$$Recall_0 = \frac{T_0}{(T_0 + F_{10} + F_{-10})} \tag{4.9}$$

Ultimately, the recall value related to the affinity score $y = -1$ is $Recall_{-1}$.
Its equation is the subsequent:

$$Recall_{-1} = \frac{T_{-1}}{(T_{-1} + F_{1-1} + F_{0-1})} \tag{4.10}$$

4.1.5 F1-score

From the precision and recall metrics it is possible to retrieve the **F1-score** (F_1).
F_1 is a metric having the role of measuring the accuracy of a given test, and in a binary scenario it is computed with the **harmonic mean** of precision and recall.
Here its equation:

$$F_1 = 2 \cdot \frac{Precision \cdot Recall}{(Precision + Recall)} \tag{4.11}$$

This value can take a value in the range [0,1], and the following considerations can be made:

- if $F_1 = 1$, the classifier has perfect values for both $Precision$ and $Recall$;
- if $F_1 = 0$, the classifier has $Precision = 0$, or $Recall = 0$.

In the multi-class problem under consideration, it is necessary to derive a F_1 value for each one of the classes available. Thus, we defined F_{11}, F_{10} and F_{1-1}.
F_{11} is the F_1 value related to the class $y = 1$. So, it needs to be computed in the following way:

$$F_{11} = 2 \cdot \frac{Precision_1 \cdot Recall_1}{(Precision_1 + Recall_1)} \tag{4.12}$$

Similarly, F_{10} is the F_1 value for the class $y = 0$, and it can be found by applying the subsequent equation:

$$F_{10} = 2 \cdot \frac{Precision_0 \cdot Recall_0}{(Precision_0 + Recall_0)} \tag{4.13}$$

Finally, F_{1-1} is the F_1 value for the **"no match"** cases, and thus the affinity score $y = -1$.
The equation is this:

$$F_{1-1} = 2 \cdot \frac{Precision_{-1} \cdot Recall_{-1}}{(Precision_{-1} + Recall_{-1})} \tag{4.14}$$

4.1.6 Macro average

As explained in the previous pages, in a multi-class problem it is necessary to compute a metric value for each of the represented labels.
It is easy to understand that having several values could be not so comfortable for deriving *generic* conclusions about the classifier being used.
That's for this reason that the **Macro average** can be used.
The Macro average is simply the standard average of a given number of values over the number of classes. In our case, there would be the following three Macro averages:

- $Macro\text{-}Precision$;
- $Macro\text{-}Recall$;
- $Macro\text{-}F_1$

The $Macro\text{-}Precision$ is the average of the Precision values previously defined:

$$Macro\text{-}Precision = \frac{(Precision_1 + Precision_0 + Precision_{-1})}{3} \tag{4.15}$$

Similarly, the $Macro\text{-}Recall$ is the average of the Recalls:

$$Macro\text{-}Recall = \frac{(Recall_1 + Recall_0 + Recall_{-1})}{3} \tag{4.16}$$

And finally, the $Macro\text{-}F_1$:

$$Macro\text{-}F_1 = \frac{(F_{11} + F_{10} + F_{1-1})}{3} \tag{4.17}$$

It can be noted that this metric has a small disadvantage. More in detail, like the accuracy, it is sensitive to unbalanced data sets: in fact - as it can be seen on the denominator - it assumes that the different classes are equally distributed in the data set.
In a data set where one of the classes is a lot less present, it is highly probable that the *Precision* and *Recall* values for the class - and as a consequence the F_1 - are low, decreasing the Macro average value.
That is the reason why we decided to consider also the **Weighted average**, which we are now going to explain.

4.1.7 Weighted average

The Weighted average, similarly to the Macro average, has the goal of combining several values related to a given metric into a single one.
Instead of computing a standard average - which assumes equal weights for all the classes -, it is derived by considering the *actual* number of samples related to a class as its *weight*.
Thus, considering a metric M in a multi-class problem represented by a number of classes $y = k$, the generic equation can be expressed as:

$$Weighted\text{-}M = \frac{(w_1 \cdot M_1 + w_2 \cdot M_2 + \ldots + w_k \cdot M_k)}{(w_1 + w_2 + \ldots + w_k)} \tag{4.18}$$

The w values represent the various weights for the classes.
For the problem under study, we retrieved the following weighted averages:

- $Weighted\text{-}Precision$;
- $Weighted\text{-}Recall$;
- $Weighted\text{-}F_1$.

It is important to denote that the weights can be easily retrieved by summing up the values present on the fixed columns of the confusion matrix. So, they would be the following:

- $w_1 = (T_1 + F_{01} + F_{-11})$;
- $w_0 = (F_{10} + T_0 + F_{-10})$;
- $w_{-1} = (F_{1-1} + F_{0-1} + T_{-1})$.

Having defined the weights, the three metrics described above can be defined.
The $Weighted\text{-}Precision$ is the weighted average of the $Precision$ values.
Abbreviating $Precision_1, Precision_0$ and $Precision_{-1}$ into P_1, P_0 and P_{-1} respectively, we can write the following equation:

$$Weighted\text{-}Precision = \frac{(w_1 \cdot P_1 + w_0 \cdot P_0 + w_{-1} \cdot P_{-1})}{(w_1 + w_0 + w_{-1})} \tag{4.19}$$

Similarly, we can derive the $Weighted\text{-}Recall$.
Again, we abbreviate $Recall_1, Recall_0$ and $Recall_{-1}$ into R_1, R_0 and R_{-1} respectively, and we define the subsequent:

$$Weighted\text{-}Recall = \frac{(w_1 \cdot R_1 + w_0 \cdot R_0 + w_{-1} \cdot R_{-1})}{(w_1 + w_0 + w_{-1})} \tag{4.20}$$

Finally, the $Weighted\text{-}F_1$ can be defined:

$$Weighted\text{-}F_1 = \frac{(w_1 \cdot F_{11} + w_0 \cdot F_{10} + w_{-1} \cdot F_{-11})}{(w_1 + w_0 + w_{-1})} \tag{4.21}$$

In this way, we are able to give to each class its "importance" in the classifier, without conditioning the metrics.

4.1.8 Training time

The last metric that we chose to consider was the **training time**.
By training time t we mean the time spent by the classifier to train on a chunk of data D_i at an iteration i in the IL algorithm.
We used it mainly for having an idea about the time spent by the classifier/s when a partial fitting operation is performed: in this way, it could have been possible to have a comparison with the time spent by the respective standard versions when the training is directly executed over the entire data set.
We know that the mentioned metric highly depends on the used hardware, and so can't be considered as a pure "truth". Anyway, it helped us to draw conclusions about the overall implementation of the solution.
For testing purposes, the various training time values have been computed on a MacBook Pro (Mid 2014) with the following technical characteristics:

- **CPU**: Intel Core i5 Dual-Core 2,6 Ghz;
- **RAM**: 8 GB 1600 MHz DDR3;
- **Graphics**: Intel Iris 1536 MB.

We decided to not use as a metric the entire execution time of the IL algorithm, since in our opinion it wouldn't be totally fair: the incremental approaches - being iterative - in our environment would just introduce a lot of overhead due to the immediate repetition of the partial fit operations,

causing a higher execution time.
This wouldn't happen with the standard approaches, in which the training is performed just once.
It is important to underline the fact that in the real scenario the partial fit is executed after a certain amount of time/samples, and not as soon as the training is completed over the previous data chunk.

Having described all the metrics that we chose to adopt, we are now going to show the obtained results.

4.2 Obtained results

In this section, we want to show the obtained results.
The experiments have been carried by considering different values for the following parameters:

- Split factor s;
- Number of estimators per chunk N_{ec} (in the incremental models based on RF and ET);
- Number of estimators N_e (in standard RF and ET);
- Criterion cr (in both incremental and standard models based on RF and ET).

More in detail, the subsequent values have been used:

- s: [2, 4, 6];
- N_{ec}: [10, 50];
- N_e: [10, 50];
- cr: [*"gini"*, *"entropy"*].

We are now going to consider singularly the various incremental models, in comparison with their standard versions.

4.2.1 Naïve-Bayes classifier

For what concerns the Naïve-Bayes classifier, the tests have been conducted only considering the various values for the parameter s.
We are now going to show the most relevant information obtained in the different configurations, like the confusion matrices in some of the intermediate steps of the incremental version, and the derived metrics.

We start by considering a split factor $s = 2$. In this case, the original data set is split in two equal parts, thus the partial fitting operation is applied twice on the incremental model.
Here below, it is possible to see what has been obtained on the various iterations of the IL algorithm:

$i = 1, 2$	$Precision$	$Recall$	F_1
$y = 1$ $(i = 1)$	0.48	0.38	0.43
$y = 0$ $(i = 1)$	0.31	0.14	0.19
$y = -1$ $(i = 1)$	0.67	0.80	0.73
$y = 1$ $(i = 2)$	0.42	0.38	0.40
$y = 0$ $(i = 2)$	0.28	0.10	0.15
$y = -1$ $(i = 2)$	0.66	0.76	0.70

Table 4.3: Metrics obtained for incremental NB with $s = 2$

$i = 1, 2$	$Precision$	$Recall$	F_1
$M.avg$ $(i = 1)$	0.48	0.44	0.45
$W.avg$ $(i = 1)$	0.57	0.61	0.58
$M.avg$ $(i = 2)$	0.45	0.41	0.42
$W.avg$ $(i = 2)$	0.55	0.58	0.56

Table 4.4: Avg metrics for incremental NB with $s = 2$

It is possible to note that the classifier doesn't behave good at all, and even gets worse on the second iteration.

Anyway, this is also valid for the standard version:

Standard	*Precision*	*Recall*	F_1
$y = 1$	0.42	0.38	0.40
$y = 0$	0.28	0.10	0.15
$y = -1$	0.66	0.76	0.70

Table 4.5: Metrics obtained for the standard NB

Standard	*Precision*	*Recall*	F_1
M.avg	0.45	0.41	0.42
W.avg	0.55	0.58	0.56

Table 4.6: Avg metrics obtained for the standard NB

It is also possible to see that what is produced at $i = 2$ by the incremental classifier coincides with what is given by the standard one.
For what concerns the accuracy behavior over time, it has been observed that the one obtained for the incremental model converges to the one provided by the standard one, with a value $\approx 58\%$.

In Figure 4.1, it is possible to see that the weighted F1-score of the incremental model converges to the value obtained with the standard one.

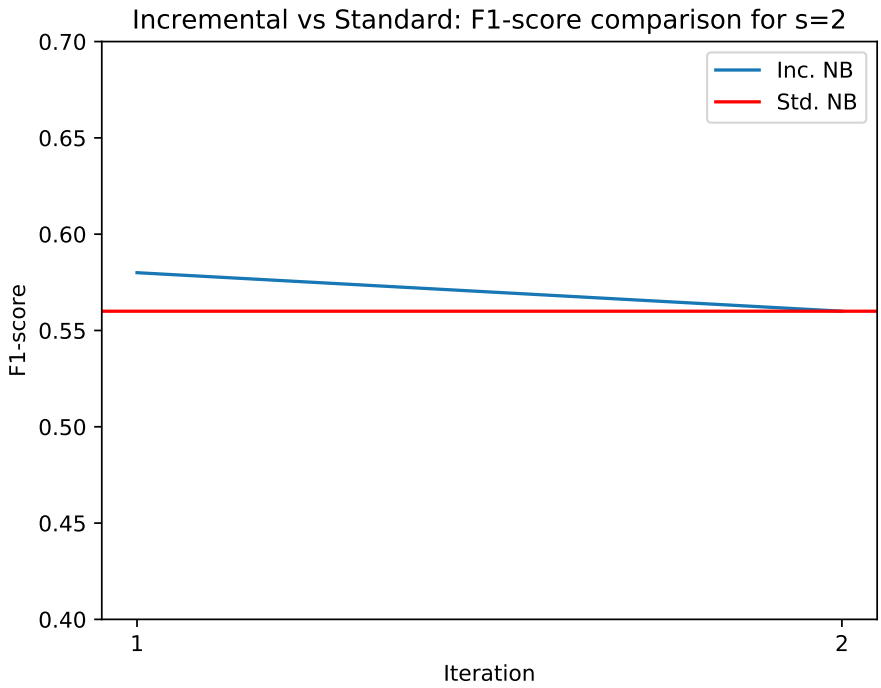

Figure 4.1: F1-score comp. for NB with $s = 2$

The next test has been executed using $s = 4$.
In this situation, we considered the metrics obtained for the iterations $i = 1, 2, 4$:

$i = 1, 2, 4$	*Precision*	*Recall*	F_1
$y = 1$ $(i = 1)$	0.44	0.37	0.40
$y = 0$ $(i = 1)$	0.24	0.12	0.16
$y = -1$ $(i = 1)$	0.66	0.77	0.71
$y = 1$ $(i = 2)$	0.48	0.38	0.43
$y = 0$ $(i = 2)$	0.31	0.14	0.19
$y = -1$ $(i = 2)$	0.67	0.80	0.73
$y = 1$ $(i = 4)$	0.42	0.38	0.40
$y = 0$ $(i = 4)$	0.28	0.10	0.15
$y = -1$ $(i = 4)$	0.66	0.76	0.70

Table 4.7: Metrics obtained for incremental NB with $s = 4$

Instead, the averaged metrics are the following:

$i = 1, 2, 4$	$Precision$	$Recall$	F_1
$M.avg\ (i = 1)$	0.45	0.42	0.42
$W.avg\ (i = 1)$	0.55	0.58	0.56
$M.avg\ (i = 2)$	0.48	0.44	0.45
$W.avg\ (i = 2)$	0.57	0.61	0.58
$M.avg\ (i = 4)$	0.45	0.41	0.42
$W.avg\ (i = 4)$	0.55	0.58	0.56

Table 4.8: Avg metrics obtained for incremental NB with $s = 4$

It is possible to note a behavior not so different from the test conducted with $s = 2$: in fact the incremental classifier, after some time, will settle and converge again to the standard one, for which the metrics are displayed in Table 4.5 and Table 4.6.
Also in here, we plotted the trend followed by the weighted F1-score, which is possible to see in Figure 4.2.

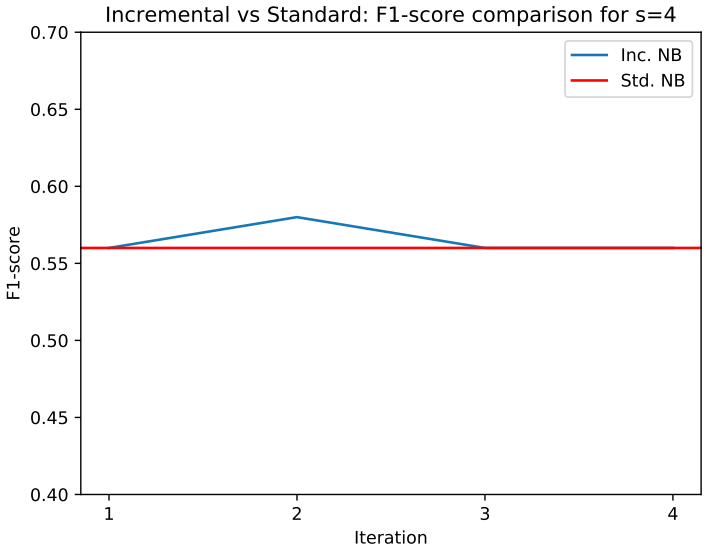

Figure 4.2: F1-score comp. for NB with $s = 4$

The last test on the Naïve-Bayes classifier has been conducted with a split factor $s = 6$.
The following metrics have been obtained:

$i = 2, 4, 6$	*Precision*	*Recall*	F_1
$y = 1$ $(i = 2)$	0.44	0.36	0.40
$y = 0$ $(i = 2)$	0.20	0.08	0.11
$y = -1$ $(i = 2)$	0.65	0.78	0.71
$y = 1$ $(i = 4)$	0.44	0.38	0.41
$y = 0$ $(i = 4)$	0.27	0.16	0.20
$y = -1$ $(i = 4)$	0.67	0.77	0.71
$y = 1$ $(i = 6)$	0.42	0.38	0.40
$y = 0$ $(i = 6)$	0.28	0.10	0.15
$y = -1$ $(i = 6)$	0.66	0.76	0.70

Table 4.9: Metrics obtained for incremental NB with $s = 6$

Regarding the averaged ones, instead:

$i = 2, 4, 6$	*Precision*	*Recall*	F_1
$M.avg$ $(i = 2)$	0.43	0.41	0.41
$W.avg$ $(i = 2)$	0.54	0.58	0.56
$M.avg$ $(i = 4)$	0.46	0.43	0.44
$W.avg$ $(i = 4)$	0.56	0.59	0.57
$M.avg$ $(i = 6)$	0.45	0.41	0.42
$W.avg$ $(i = 6)$	0.55	0.58	0.56

Table 4.10: Avg metrics obtained for incremental NB with $s = 6$

As in the previous two tests, the incremental classifier converges to the standard one, but the overall bad metrics obtained suggest to not adopt this model for this particular problem.

Also here, the weighted F1-score trend has been monitored, and can be seen in Figure 4.3.

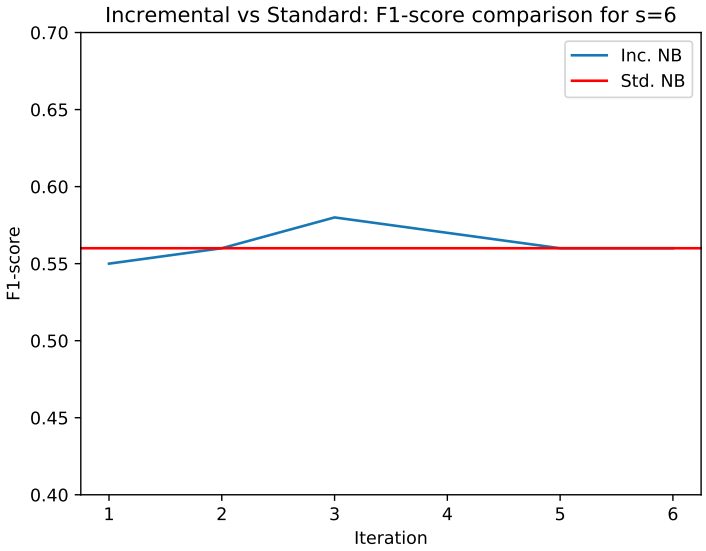

Figure 4.3: F1-score comp. for NB with $s = 6$

In the following section, we are going to discuss about the tests carried out for the incremental RF classifier.

4.2.2 Random Forest classifier

For the RF classifier, we have carried out a higher number of tests with respect to the NB classifier.
This was necessary, because of the fact that the RF - in addition to the split factor s - depends also on the number of estimators per chunk N_{ec} (in the incremental version), on the number of estimators N_e (in the standard version), and on the criterion cr used for performing the splitting.

The first test that we performed was with $s = 2$ and $cr =$ "$gini$".
In it, we considered $N = N_{ec} = N_e = 50$ and $N = N_{ec} = N_e = 100$.
The following metrics have been derived:

$i = 1, 2$	$Precision$	$Recall$	F_1
$y = 1$ $(i = 1, N = 50)$	0.86	0.73	0.79
$y = 0$ $(i = 1, N = 50)$	0.50	0.19	0.28
$y = -1$ $(i = 1, N = 50)$	0.82	0.97	0.89
$y = 1$ $(i = 2, N = 50)$	0.87	0.81	0.84
$y = 0$ $(i = 2, N = 50)$	0.70	0.09	0.16
$y = -1$ $(i = 2, N = 50)$	0.83	0.98	0.90
$y = 1$ $(i = 1, N = 100)$	0.84	0.72	0.77
$y = 0$ $(i = 1, N = 100)$	0.42	0.10	0.17
$y = -1$ $(i = 1, N = 100)$	0.81	0.97	0.88
$y = 1$ $(i = 2, N = 100)$	0.88	0.84	0.86
$y = 0$ $(i = 2, N = 100)$	0.60	0.08	0.14
$y = -1$ $(i = 2, N = 100)$	0.84	0.99	0.91

Table 4.11: Metrics for incremental RF with $s = 2$, $cr =$ "$gini$"

Instead, the averaged ones are the ones depicted in Table 4.12.

$i = 1, 2$	*Precision*	*Recall*	F_1
$M.avg$ $(i = 1, N = 50)$	0.73	0.63	0.65
$W.avg$ $(i = 1, N = 50)$	0.80	0.82	0.79
$M.avg$ $(i = 2, N = 50)$	0.80	0.63	0.63
$W.avg$ $(i = 2, N = 50)$	0.83	0.84	0.81
$M.avg$ $(i = 1, N = 100)$	0.69	0.60	0.61
$W.avg$ $(i = 1, N = 100)$	0.78	0.81	0.78
$M.avg$ $(i = 2, N = 100)$	0.77	0.63	0.64
$W.avg$ $(i = 2, N = 100)$	0.83	0.85	0.81

Table 4.12: Avg metrics for incremental RF with $s = 2$, $cr =$ "*gini*"

Of course, they should be compared with the various metrics obtained for the standard implementation, which are available in Table 4.13 and Table 4.14.
It is possible to note that with a big chunk of data, the incremental model behaves very similarly to the standard one, and this can be considered as a good result: in fact, if there is the wish of performing the partial fitting when a big amount of data is collected, this could be the way to go.

Standard	*Precision*	*Recall*	F_1
$y = 1$ $(N = 50)$	0.86	0.90	0.88
$y = 0$ $(N = 50)$	0.53	0.12	0.19
$y = -1$ $(N = 50)$	0.88	0.98	0.93
$y = 1$ $(N = 100)$	0.87	0.93	0.90
$y = 0$ $(N = 100)$	0.67	0.16	0.25
$y = -1$ $(N = 100)$	0.90	0.98	0.94

Table 4.13: Metrics obtained for the standard RF with $cr =$ "*gini*"

Standard	*Precision*	*Recall*	F_1
$M.avg\ (N = 50)$	0.76	0.67	0.67
$W.avg\ (N = 50)$	0.84	0.87	0.84
$M.avg\ (N = 100)$	0.81	0.69	0.70
$W.avg\ (N = 100)$	0.87	0.88	0.86

Table 4.14: Avg metrics obtained for the standard RF with $cr =$ "*gini*"

For what concerns the accuracy trends, they are available in Figure 4.4 and Figure 4.5.

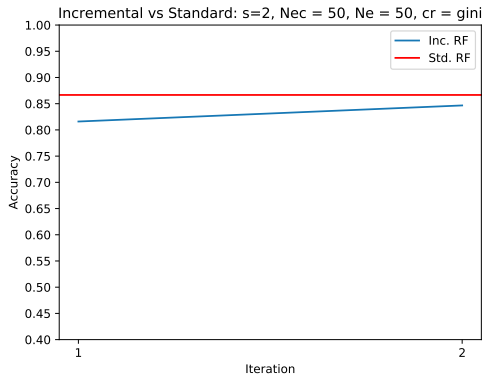

Figure 4.4: Acc. comp. for RF with $s = 2$, $N = 50$, $cr =$ "*gini*"

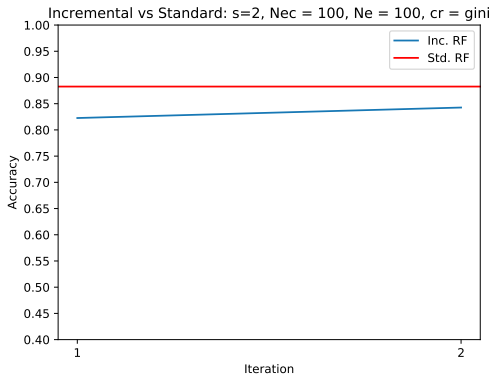

Figure 4.5: Acc. comp. for RF with $s = 2$, $N = 100$, $cr =$ "*gini*"

It can be noted that the standard RF classifier actually performs better with a higher number of estimators, while the incremental one performs basically in the same way.
In any case, they do not differ much, and this gives an advantage for the incremental version, which processed less data at a time.

The following test has been performed using $s = 4$ and keeping $cr =$ "$gini$". The metrics obtained at the various iterations can be seen in Table 4.15.

$i = 1, 2, 4$	$Precision$	$Recall$	F_1
$y = 1$ $(i = 1, N = 50)$	0.74	0.43	0.55
$y = 0$ $(i = 1, N = 50)$	0.35	0.08	0.13
$y = -1$ $(i = 1, N = 50)$	0.70	0.94	0.81
$y = 1$ $(i = 2, N = 50)$	0.81	0.47	0.60
$y = 0$ $(i = 2, N = 50)$	0.71	0.06	0.12
$y = -1$ $(i = 2, N = 50)$	0.71	0.98	0.82
$y = 1$ $(i = 4, N = 50)$	0.86	0.57	0.68
$y = 0$ $(i = 4, N = 50)$	1.00	0.01	0.03
$y = -1$ $(i = 4, N = 50)$	0.74	0.99	0.84
$y = 1$ $(i = 1, N = 100)$	0.79	0.53	0.63
$y = 0$ $(i = 1, N = 100)$	0.38	0.08	0.13
$y = -1$ $(i = 1, N = 100)$	0.73	0.96	0.83
$y = 1$ $(i = 2, N = 100)$	0.85	0.58	0.69
$y = 0$ $(i = 2, N = 100)$	0.57	0.05	0.10
$y = -1$ $(i = 2, N = 100)$	0.74	0.98	0.84
$y = 1$ $(i = 4, N = 100)$	0.89	0.62	0.73
$y = 0$ $(i = 4, N = 100)$	1.00	0.01	0.03
$y = -1$ $(i = 4, N = 100)$	0.75	0.99	0.85

Table 4.15: Metrics for incremental RF with $s = 4$, $cr =$ "$gini$"

What can be immediately seen is that by increasing the split factor the performance decreases. Also, the classifier seems to lose its ability in predicting $y = 0$, providing thus null metrics for the latter.
By the way, the averaged metrics can be considered good, at least for the labels $y = 1$ and $y = -1$.
It is possible to check them in Table 4.16.

$i = 1, 2, 4$	$Precision$	$Recall$	F_1
$M.avg\ (i = 1, N = 50)$	0.60	0.49	0.49
$W.avg\ (i = 1, N = 50)$	0.68	0.70	0.66
$M.avg\ (i = 2, N = 50)$	0.75	0.50	0.51
$W.avg\ (i = 2, N = 50)$	0.74	0.73	0.68
$M.avg\ (i = 4, N = 50)$	0.87	0.52	0.52
$W.avg\ (i = 4, N = 50)$	0.80	0.76	0.71
$M.avg\ (i = 1, N = 100)$	0.63	0.52	0.53
$W.avg\ (i = 1, N = 100)$	0.72	0.74	0.70
$M.avg\ (i = 2, N = 100)$	0.72	0.54	0.54
$W.avg\ (i = 2, N = 100)$	0.76	0.76	0.72
$M.avg\ (i = 4, N = 100)$	0.88	0.54	0.54
$W.avg\ (i = 4, N = 100)$	0.82	0.78	0.73

Table 4.16: Avg metrics for incremental RF with $s = 4$, $cr =$ "*gini*"

The accuracy trends for this test case are depicted in Figure 4.6 and 4.7.

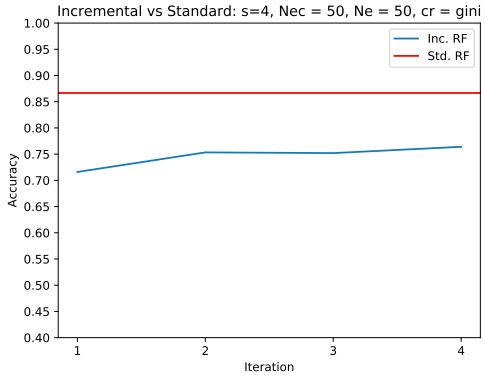

Figure 4.6: Acc. comp. for RF with $s = 4$, $N = 50$, $cr =$ "*gini*"

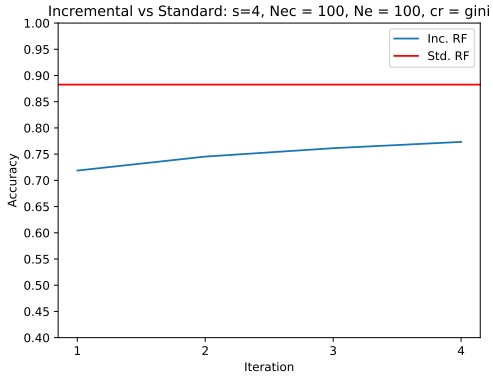

Figure 4.7: Acc. comp. for RF with $s = 4$, $N = 100$, $cr =$ "*gini*"

It is possible to see that the incremental version keeps its accuracy without considering the number of estimators: anyway, it is lower than the one obtained for $s = 2$.

A third test has been executed by using $s = 6$, and $cr =$ "$gini$".
In this case, the results available in Table 4.17 have been achieved.

$i = 2, 4, 6$	$Precision$	$Recall$	F_1
$y = 1$ $(i = 2, N = 50)$	0.78	0.46	0.58
$y = 0$ $(i = 2, N = 50)$	0.80	0.05	0.10
$y = -1$ $(i = 2, N = 50)$	0.71	0.97	0.82
$y = 1$ $(i = 4, N = 50)$	0.86	0.44	0.58
$y = 0$ $(i = 4, N = 50)$	-	-	-
$y = -1$ $(i = 4, N = 50)$	0.70	0.99	0.82
$y = 1$ $(i = 6, N = 50)$	0.88	0.42	0.57
$y = 0$ $(i = 6, N = 50)$	1.00	0.01	0.03
$y = -1$ $(i = 6, N = 50)$	0.69	0.99	0.82
$y = 1$ $(i = 2, N = 100)$	0.79	0.35	0.49
$y = 0$ $(i = 2, N = 100)$	0.57	0.05	0.10
$y = -1$ $(i = 2, N = 100)$	0.68	0.97	0.80
$y = 1$ $(i = 4, N = 100)$	0.83	0.38	0.53
$y = 0$ $(i = 4, N = 100)$	-	-	-
$y = -1$ $(i = 4, N = 100)$	0.68	0.99	0.81
$y = 1$ $(i = 6, N = 100)$	0.84	0.38	0.53
$y = 0$ $(i = 6, N = 100)$	-	-	-
$y = -1$ $(i = 6, N = 100)$	0.68	0.99	0.81

Table 4.17: Metrics for incremental RF with $s = 6$, $cr =$ "$gini$"

As in the previous test, the incremental classifier - over time - loses its ability to predict $y = 0$, making the multi-class problem degenerate to a binary one. Also, it can be seen that the obtained metrics are worse than the ones obtained with $s = 4$, underlining the fact that the smaller the data used by the incremental algorithm is, the worse it will perform.
This is confirmed also by the averaged metrics, available in Table 4.18.

$i = 2, 4, 6$	$Precision$	$Recall$	F_1
$M.avg\ (i = 2, N = 50)$	0.76	0.49	0.50
$W.avg\ (i = 2, N = 50)$	0.74	0.72	0.67
$M.avg\ (i = 4, N = 50)$	0.52	0.48	0.47
$W.avg\ (i = 4, N = 50)$	0.68	0.72	0.67
$M.avg\ (i = 6, N = 50)$	0.86	0.48	0.47
$W.avg\ (i = 6, N = 50)$	0.78	0.72	0.66
$M.avg\ (i = 2, N = 100)$	0.68	0.46	0.46
$W.avg\ (i = 2, N = 100)$	0.70	0.69	0.63
$M.avg\ (i = 4, N = 100)$	0.50	0.46	0.44
$W.avg\ (i = 4, N = 100)$	0.66	0.70	0.64
$M.avg\ (i = 6, N = 100)$	0.51	0.46	0.45
$W.avg\ (i = 6, N = 100)$	0.66	0.71	0.64

Table 4.18: Avg metrics for incremental RF with $s = 6$, $cr =$ "*gini*"

Finally, the behavior followed by the accuracy in both cases can be found in Figure 4.8 and Figure 4.9.

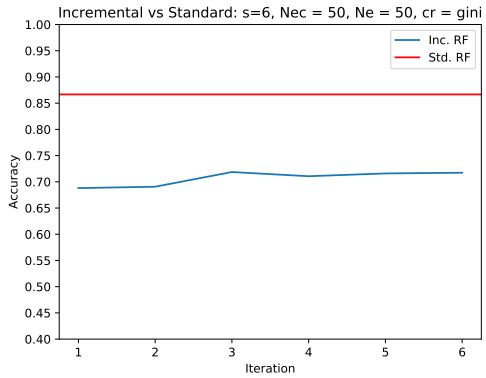

Figure 4.8: Acc. comp. for RF with $s = 6$, $N = 50$, $cr =$ "*gini*"

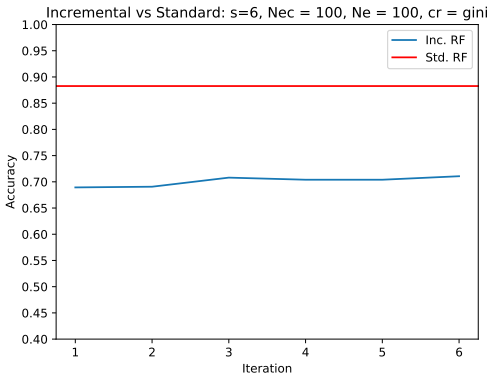

Figure 4.9: Acc. comp. for RF with $s = 6$, $N = 100$, $cr =$ "*gini*"

As introduced previously, the performance decreases as the amount of processed data decreases. Also, in this case, the difference between the accuracies starts to be notable, being higher than the 20%.

After having performed all the tests for $cr =$ "*gini*", we decided to try also $cr =$ "*entropy*".
The first test we performed in this context was with the following parameters: $s = 2$ and $cr =$ "*entropy*".
The metrics we obtained are available in Table 4.19.

$i = 1, 2$	*Precision*	*Recall*	F_1
$y = 1$ $(i = 1, N = 50)$	0.86	0.73	0.79
$y = 0$ $(i = 1, N = 50)$	0.50	0.14	0.22
$y = -1$ $(i = 1, N = 50)$	0.81	0.97	0.88
$y = 1$ $(i = 2, N = 50)$	0.88	0.81	0.84
$y = 0$ $(i = 2, N = 50)$	0.64	0.09	0.16
$y = -1$ $(i = 2, N = 50)$	0.83	0.99	0.90
$y = 1$ $(i = 1, N = 100)$	0.87	0.74	0.80
$y = 0$ $(i = 1, N = 100)$	0.57	0.16	0.24
$y = -1$ $(i = 1, N = 100)$	0.81	0.97	0.88
$y = 1$ $(i = 2, N = 100)$	0.88	0.81	0.85
$y = 0$ $(i = 2, N = 100)$	0.67	0.05	0.10
$y = -1$ $(i = 2, N = 100)$	0.82	0.99	0.90

Table 4.19: Metrics for incremental RF with $s = 2$, $cr =$ "*entropy*"

It can be immediately seen that the obtained results are very similar with the ones found using the Gini impurity as criterion, so to use one or another can be considered as a simple matter of taste.
For what concerns the averaged metrics, they can be seen in Table 4.20.

$i = 1, 2$	$Precision$	$Recall$	F_1
$M.avg$ $(i = 1, N = 50)$	0.72	0.62	0.63
$W.avg$ $(i = 1, N = 50)$	0.79	0.81	0.79
$M.avg$ $(i = 2, N = 50)$	0.78	0.63	0.64
$W.avg$ $(i = 2, N = 50)$	0.83	0.84	0.81
$M.avg$ $(i = 1, N = 100)$	0.75	0.62	0.64
$W.avg$ $(i = 1, N = 100)$	0.80	0.82	0.79
$M.avg$ $(i = 2, N = 100)$	0.79	0.62	0.61
$W.avg$ $(i = 2, N = 100)$	0.83	0.84	0.80

Table 4.20: Avg metrics for incremental RF with $s = 2$, $cr =$ "*entropy*"

As in the previous tests, we want to show also the metrics which are obtained from a standard RF using $cr =$ "*entropy*". They are available in Table 4.21.

$Standard$	$Precision$	$Recall$	F_1
$y = 1$ $(N = 50)$	0.88	0.88	0.88
$y = 0$ $(N = 50)$	0.63	0.16	0.25
$y = -1$ $(N = 50)$	0.87	0.98	0.93
$y = 1$ $(N = 100)$	0.87	0.90	0.88
$y = 0$ $(N = 100)$	0.63	0.16	0.25
$y = -1$ $(N = 100)$	0.88	0.98	0.92

Table 4.21: Metrics for the standard RF with $cr =$ "*entropy*"

The averaged ones can be seen in Table 4.22.

Standard	*Precision*	*Recall*	F_1
$M.avg\ (N = 50)$	0.80	0.68	0.69
$W.avg\ (N = 50)$	0.85	0.87	0.84
$M.avg\ (N = 100)$	0.79	0.68	0.69
$W.avg\ (N = 100)$	0.85	0.87	0.84

Table 4.22: Avg metrics for the standard RF with $cr =$ "*entropy*"

Instead, the following accuracy trends have been derived.

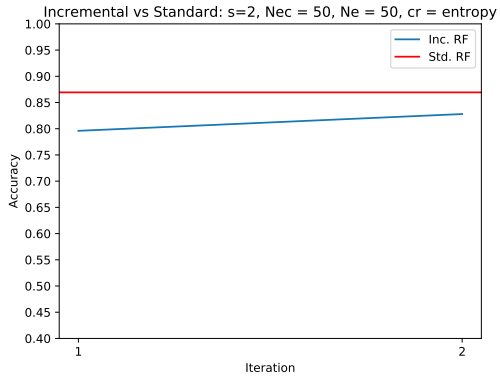

Figure 4.10: Acc. comp. for RF with $s = 2$, $N = 50$, $cr =$ "*entropy*"

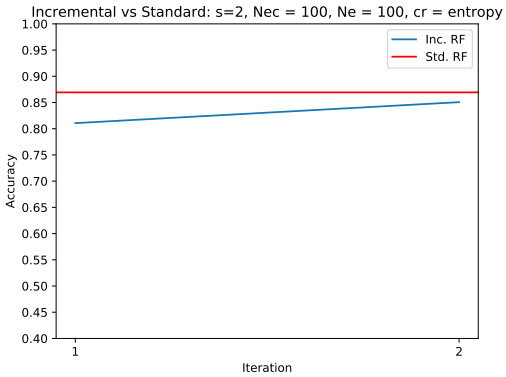

Figure 4.11: Acc. comp. for RF with $s = 2$, $N = 100$, $cr =$ "*entropy*"

It is possible to see that, by using a higher number of estimators per chunk, the incremental classifier performs really close to the standard one, having also a quite good accuracy.
This could be a good solution.

The next test has been carried out by using a split factor $s = 4$, and remaining with $cr =$ "*entropy*".
The Table 4.23 shows the metrics that have been obtained through some of the iterations.

$i = 1, 2, 4$	*Precision*	*Recall*	F_1
$y = 1$ $(i = 1, N = 50)$	0.76	0.49	0.59
$y = 0$ $(i = 1, N = 50)$	0.38	0.08	0.13
$y = -1$ $(i = 1, N = 50)$	0.72	0.94	0.81
$y = 1$ $(i = 2, N = 50)$	0.82	0.50	0.62
$y = 0$ $(i = 2, N = 50)$	0.67	0.05	0.10
$y = -1$ $(i = 2, N = 50)$	0.72	0.97	0.83
$y = 1$ $(i = 4, N = 50)$	0.86	0.56	0.68
$y = 0$ $(i = 4, N = 50)$	1.00	0.01	0.03
$y = -1$ $(i = 4, N = 50)$	0.73	0.99	0.84
$y = 1$ $(i = 1, N = 100)$	0.76	0.49	0.59
$y = 0$ $(i = 1, N = 100)$	0.43	0.08	0.13
$y = -1$ $(i = 1, N = 100)$	0.72	0.95	0.82
$y = 1$ $(i = 2, N = 100)$	0.84	0.51	0.64
$y = 0$ $(i = 2, N = 100)$	0.62	0.06	0.12
$y = -1$ $(i = 2, N = 100)$	0.73	0.98	0.83
$y = 1$ $(i = 4, N = 100)$	0.89	0.58	0.70
$y = 0$ $(i = 4, N = 100)$	0.50	0.01	0.03
$y = -1$ $(i = 4, N = 100)$	0.74	0.99	0.85

Table 4.23: Metrics for incremental RF with $s = 4$, $cr =$ "*entropy*"

Like in the previous tests, it is possible to observe that again, using less data, the results are less good.
This is confirmed by the averaged metrics, available in Table 4.24.

$i = 1, 2, 4$	$Precision$	$Recall$	F_1
$M.avg\ (i = 1, N = 50)$	0.62	0.50	0.51
$W.avg\ (i = 1, N = 50)$	0.69	0.72	0.68
$M.avg\ (i = 2, N = 50)$	0.74	0.51	0.52
$W.avg\ (i = 2, N = 50)$	0.74	0.74	0.69
$M.avg\ (i = 4, N = 50)$	0.87	0.52	0.51
$W.avg\ (i = 4, N = 50)$	0.80	0.76	0.71
$M.avg\ (i = 1, N = 100)$	0.64	0.51	0.52
$W.avg\ (i = 1, N = 100)$	0.70	0.72	0.68
$M.avg\ (i = 2, N = 100)$	0.73	0.52	0.53
$W.avg\ (i = 2, N = 100)$	0.75	0.75	0.70
$M.avg\ (i = 4, N = 100)$	0.71	0.53	0.52
$W.avg\ (i = 4, N = 100)$	0.76	0.77	0.72

Table 4.24: Avg metrics for incremental RF with $s = 4$, $cr =$ "*entropy*"

Also in this case, the behavior of the accuracies over the iterations have been plotted. They can be seen in Figure 4.12 and Figure 4.14.

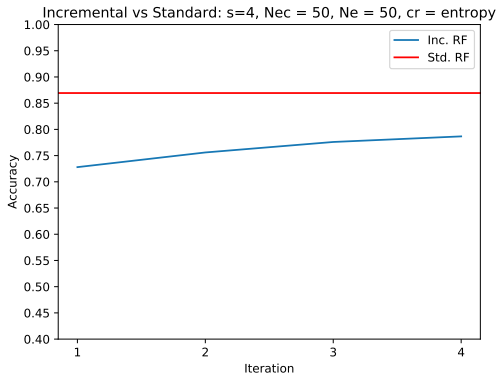

Figure 4.12: Acc. comp. for RF with $s = 4$, $N = 50$, $cr =$ "*entropy*"

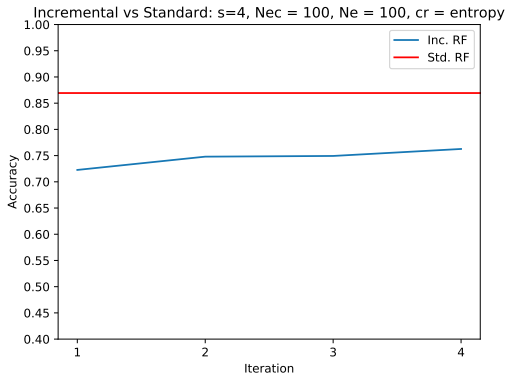

Figure 4.13: Acc. comp. for RF with $s = 4$, $N = 100$, $cr =$ "*entropy*"

The values obtained by the two incremental classifiers are very similar, with the first - having a smaller number of estimators - behaving slightly better. In any case, the difference with the standard version is $\approx 10\%$.

The last test we performed with the RF classifier was for $s = 6$. For the metrics, we obtained the values represented in Table 4.25.

$i = 2, 4, 6$	$Precision$	$Recall$	F_1
$y = 1$ $(i = 2, N = 50)$	0.80	0.33	0.47
$y = 0$ $(i = 2, N = 50)$	0.67	0.03	0.05
$y = -1$ $(i = 2, N = 50)$	0.67	0.98	0.79
$y = 1$ $(i = 4, N = 50)$	0.83	0.37	0.51
$y = 0$ $(i = 4, N = 50)$	-	-	-
$y = -1$ $(i = 4, N = 50)$	0.68	0.99	0.81
$y = 1$ $(i = 6, N = 50)$	0.84	0.35	0.49
$y = 0$ $(i = 6, N = 50)$	-	-	-
$y = -1$ $(i = 6, N = 50)$	0.67	0.99	0.80
$y = 1$ $(i = 2, N = 100)$	0.77	0.35	0.48
$y = 0$ $(i = 2, N = 100)$	0.80	0.05	0.10
$y = -1$ $(i = 2, N = 100)$	0.68	0.97	0.80
$y = 1$ $(i = 4, N = 100)$	0.85	0.39	0.53
$y = 0$ $(i = 4, N = 100)$	1.00	0.01	0.03
$y = -1$ $(i = 4, N = 100)$	0.68	0.99	0.81
$y = 1$ $(i = 6, N = 100)$	0.87	0.36	0.51
$y = 0$ $(i = 6, N = 100)$	-	-	-
$y = -1$ $(i = 6, N = 100)$	0.68	0.99	0.80

Table 4.25: Metrics for incremental RF with $s = 6$, $cr =$ "*entropy*"

As in the experiments performed with $cr =$ "*gini*", this is the worst scenario. This is also confirmed by the averaged metrics, which are depicted in Table 4.26.

$i = 2, 4, 6$	*Precision*	*Recall*	F_1
$M.avg\ (i = 2, N = 50)$	0.71	0.44	0.44
$W.avg\ (i = 2, N = 50)$	0.71	0.68	0.62
$M.avg\ (i = 4, N = 50)$	0.50	0.45	0.44
$W.avg\ (i = 4, N = 50)$	0.66	0.70	0.63
$M.avg\ (i = 6, N = 50)$	0.50	0.45	0.43
$W.avg\ (i = 6, N = 50)$	0.65	0.69	0.63
$M.avg\ (i = 2, N = 100)$	0.75	0.46	0.46
$W.avg\ (i = 2, N = 100)$	0.72	0.69	0.63
$M.avg\ (i = 4, N = 100)$	0.84	0.46	0.46
$W.avg\ (i = 4, N = 100)$	0.77	0.71	0.64
$M.avg\ (i = 6, N = 100)$	0.52	0.45	0.44
$W.avg\ (i = 6, N = 100)$	0.67	0.70	0.63

Table 4.26: Avg metrics for incremental RF with $s = 6$, $cr =$ "*entropy*"

The accuracy trends, instead, are visible in Figure 4.14 and Figure 4.15.

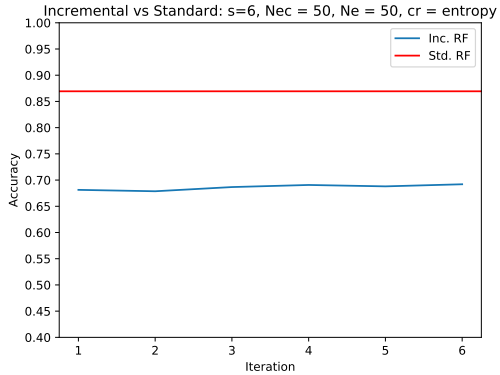

Figure 4.14: Acc. comp. for RF with $s = 6$, $N = 50$, $cr =$ "*entropy*"

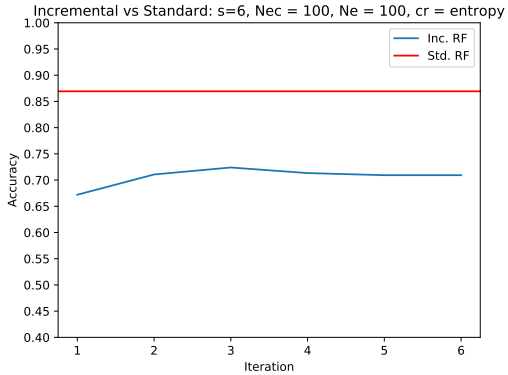

Figure 4.15: Acc. comp. for RF with $s = 6$, $N = 100$, $cr =$ "*entropy*"

It is possible to note that the accuracies decrease in both contexts, confirming what has been said in the previous scenarios.

4.2.3 Extra-Trees classifier

For what concerns the ET classifier, we performed exactly the same tests of the RF classifier.
So, again, we are first going to consider what we obtained for $cr =$ "*gini*", and then what has been retrieved by using $cr =$ "*entropy*".

The first test we carried out was with $s = 2$, and $cr =$ "*gini*".
The obtained metrics are listed in Table 4.27.

$i = 1, 2$	*Precision*	*Recall*	F_1
$y = 1$ $(i = 1, N = 50)$	0.84	0.81	0.83
$y = 0$ $(i = 1, N = 50)$	0.44	0.18	0.26
$y = -1$ $(i = 1, N = 50)$	0.86	0.96	0.90
$y = 1$ $(i = 2, N = 50)$	0.87	0.88	0.88
$y = 0$ $(i = 2, N = 50)$	0.58	0.09	0.16
$y = -1$ $(i = 2, N = 50)$	0.86	0.98	0.92
$y = 1$ $(i = 1, N = 100)$	0.86	0.80	0.83
$y = 0$ $(i = 1, N = 100)$	0.48	0.21	0.29
$y = -1$ $(i = 1, N = 100)$	0.84	0.96	0.90
$y = 1$ $(i = 2, N = 100)$	0.87	0.88	0.88
$y = 0$ $(i = 2, N = 100)$	0.57	0.10	0.18
$y = -1$ $(i = 2, N = 100)$	0.87	0.98	0.92

Table 4.27: Metrics for incremental ET with $s = 2$, $cr =$ "*gini*"

It can be immediately seen that the values obtained are higher than the ones seen on the RF with the same configuration.
This is probably due to the random effect which regulates the ET classifier's behavior.
For what concerns the averaged metrics, they are available in Table 4.28.

$i = 1, 2$	*Precision*	*Recall*	F_1
$M.avg$ $(i = 1, N = 50)$	0.71	0.65	0.66
$W.avg$ $(i = 1, N = 50)$	0.81	0.83	0.81
$M.avg$ $(i = 2, N = 50)$	0.77	0.65	0.65
$W.avg$ $(i = 2, N = 50)$	0.84	0.86	0.83
$M.avg$ $(i = 1, N = 100)$	0.73	0.65	0.67
$W.avg$ $(i = 1, N = 100)$	0.81	0.83	0.81
$M.avg$ $(i = 2, N = 100)$	0.77	0.66	0.66
$W.avg$ $(i = 2, N = 100)$	0.84	0.86	0.83

Table 4.28: Avg metrics for incremental ET with $s = 2$, $cr =$ "*gini*"

They confirm what has been said by looking at the single metrics.
In Table 4.29, it is possible to see the metrics obtained instead by the standard ET classifiers, with $cr =$ "*gini*".

Standard	*Precision*	*Recall*	F_1
$y = 1$ $(N = 50)$	0.87	0.94	0.90
$y = 0$ $(N = 50)$	0.62	0.21	0.31
$y = -1$ $(N = 50)$	0.91	0.97	0.94
$y = 1$ $(N = 100)$	0.87	0.93	0.90
$y = 0$ $(N = 100)$	0.59	0.21	0.31
$y = -1$ $(N = 100)$	0.90	0.97	0.94

Table 4.29: Metrics for the standard ET with $cr =$ "*gini*"

The averaged ones are instead available in Table 4.30.

Standard	*Precision*	*Recall*	F_1
$M.avg\ (N = 50)$	0.80	0.71	0.72
$W.avg\ (N = 50)$	0.87	0.88	0.86
$M.avg\ (N = 100)$	0.79	0.70	0.72
$W.avg\ (N = 100)$	0.86	0.88	0.86

Table 4.30: Avg metrics for the standard ET with $cr =$ "*gini*"

Also for the ET classifier, we wanted to plot the behavior of the accuracies in the current scenario. It can be observed in Figure 4.16 and 4.17.

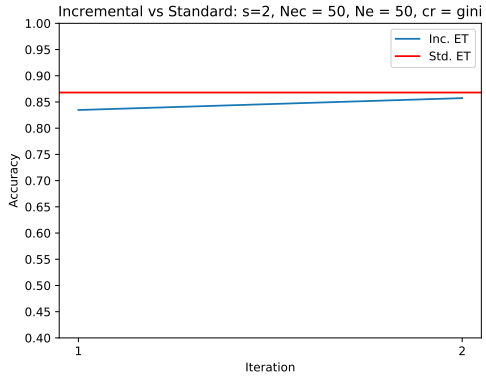

Figure 4.16: Acc. comp. for ET with $s = 2$, $N = 50$, $cr =$ "*gini*"

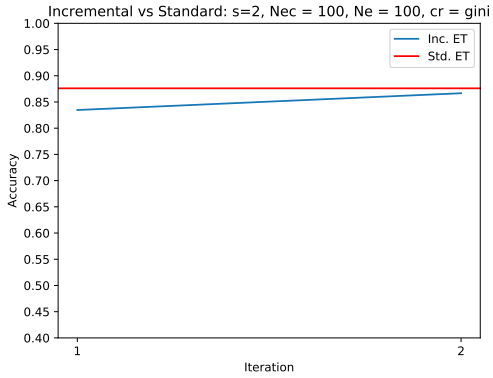

Figure 4.17: Acc. comp. for ET with $s = 2$, $N = 100$, $cr =$ "*gini*"

As it is possible to see, the accuracies obtained for the incremental ET classifiers are very close to the standard ones, especially when the number of estimators is higher. In fact, in the last case, the accuracy is higher than the 85%, while the standard one is around the 87%.
This is a very good situation, in which the incremental approach seems to be appropriate.

As for the RF, the next test has been performed by using $s = 4$, and maintaining as criterion $cr =$ "*gini*".
For this scenario, the metrics obtained can be viewed in Table 4.31.

$i = 1, 2, 4$	*Precision*	*Recall*	F_1
$y = 1$ $(i = 1, N = 50)$	0.79	0.64	0.70
$y = 0$ $(i = 1, N = 50)$	0.24	0.10	0.14
$y = -1$ $(i = 1, N = 50)$	0.78	0.93	0.85
$y = 1$ $(i = 2, N = 50)$	0.85	0.71	0.77
$y = 0$ $(i = 2, N = 50)$	0.58	0.14	0.23
$y = -1$ $(i = 2, N = 50)$	0.80	0.97	0.88
$y = 1$ $(i = 4, N = 50)$	0.90	0.81	0.85
$y = 0$ $(i = 4, N = 50)$	0.60	0.04	0.07
$y = -1$ $(i = 4, N = 50)$	0.82	0.99	0.90
$y = 1$ $(i = 1, N = 100)$	0.81	0.64	0.71
$y = 0$ $(i = 1, N = 100)$	0.29	0.10	0.15
$y = -1$ $(i = 1, N = 100)$	0.78	0.94	0.85
$y = 1$ $(i = 2, N = 100)$	0.86	0.69	0.77
$y = 0$ $(i = 2, N = 100)$	0.53	0.13	0.21
$y = -1$ $(i = 2, N = 100)$	0.79	0.97	0.87
$y = 1$ $(i = 4, N = 100)$	0.89	0.79	0.84
$y = 0$ $(i = 4, N = 100)$	0.56	0.06	0.12
$y = -1$ $(i = 4, N = 100)$	0.82	0.99	0.90

Table 4.31: Metrics for incremental ET with $s = 4$, $cr =$ "*gini*"

As it happened with the RF, when the number of chunks increases - and so less data is used - the measured values are lower.
This suggests that the incremental approach should be adopted only when the amount of data (m) is high.
The averaged metrics can be viewed in Table 4.32.

$i = 1, 2, 4$	$Precision$	$Recall$	F_1
$M.avg\ (i = 1, N = 50)$	0.60	0.56	0.57
$W.avg\ (i = 1, N = 50)$	0.73	0.76	0.73
$M.avg\ (i = 2, N = 50)$	0.74	0.61	0.63
$W.avg\ (i = 2, N = 50)$	0.79	0.81	0.78
$M.avg\ (i = 4, N = 50)$	0.77	0.61	0.61
$W.avg\ (i = 4, N = 50)$	0.82	0.84	0.80
$M.avg\ (i = 1, N = 100)$	0.62	0.56	0.57
$W.avg\ (i = 1, N = 100)$	0.74	0.77	0.74
$M.avg\ (i = 2, N = 100)$	0.73	0.60	0.62
$W.avg\ (i = 2, N = 100)$	0.78	0.80	0.77
$M.avg\ (i = 4, N = 100)$	0.75	0.62	0.62
$W.avg\ (i = 4, N = 100)$	0.81	0.84	0.80

Table 4.32: Avg metrics for incremental ET with $s = 4$, $cr =$ "*gini*"

For what concerns the accuracy trends, they are depicted in Figure 4.18 and 4.19.

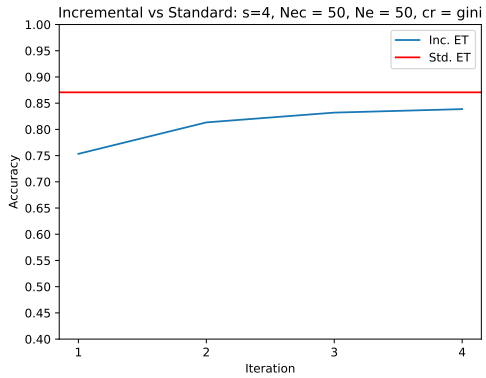

Figure 4.18: Acc. comp. for ET with $s = 4$, $N = 50$, $cr =$ "*gini*"

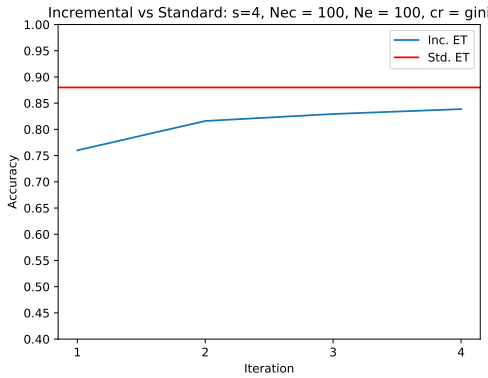

Figure 4.19: Acc. comp. for ET with $s = 4$, $N = 100$, $cr =$ "*gini*"

It is possible to observe that the performances are still very good, but a bit worse than the ones seen in the previous test.

The last test that has been carried out for $cr =$ "*gini*", was with split factor $s = 6$.
The values in Table 4.33 show the metrics that are obtained with that configuration.

$i = 2, 4, 6$	*Precision*	*Recall*	F_1
$y = 1$ $(i = 2, N = 50)$	0.81	0.55	0.65
$y = 0$ $(i = 2, N = 50)$	0.40	0.08	0.13
$y = -1$ $(i = 2, N = 50)$	0.74	0.96	0.84
$y = 1$ $(i = 4, N = 50)$	0.84	0.66	0.74
$y = 0$ $(i = 4, N = 50)$	0.50	0.03	0.05
$y = -1$ $(i = 4, N = 50)$	0.77	0.98	0.86
$y = 1$ $(i = 6, N = 50)$	0.90	0.68	0.78
$y = 0$ $(i = 6, N = 50)$	0.75	0.04	0.07
$y = -1$ $(i = 6, N = 50)$	0.77	0.99	0.87
$y = 1$ $(i = 2, N = 100)$	0.81	0.50	0.62
$y = 0$ $(i = 2, N = 100)$	0.47	0.10	0.17
$y = -1$ $(i = 2, N = 100)$	0.73	0.96	0.83
$y = 1$ $(i = 4, N = 100)$	0.85	0.65	0.74
$y = 0$ $(i = 4, N = 100)$	0.60	0.04	0.07
$y = -1$ $(i = 4, N = 100)$	0.77	0.98	0.86
$y = 1$ $(i = 6, N = 100)$	0.89	0.68	0.77
$y = 0$ $(i = 6, N = 100)$	0.75	0.04	0.07
$y = -1$ $(i = 6, N = 100)$	0.77	0.99	0.87

Table 4.33: Metrics for incremental ET with $s = 6$, $cr =$ "*gini*"

Also in here, it is possible to note that the performances degrade.
A confirmation for that is also given by the averaged metrics, available in Table 4.34.

$i = 2, 4, 6$	$Precision$	$Recall$	F_1
$M.avg\ (i = 2, N = 50)$	0.65	0.53	0.54
$W.avg\ (i = 2, N = 50)$	0.73	0.75	0.71
$M.avg\ (i = 4, N = 50)$	0.70	0.55	0.55
$W.avg\ (i = 4, N = 50)$	0.76	0.78	0.74
$M.avg\ (i = 6, N = 50)$	0.81	0.57	0.57
$W.avg\ (i = 6, N = 50)$	0.81	0.80	0.76
$M.avg\ (i = 2, N = 100)$	0.67	0.52	0.54
$W.avg\ (i = 2, N = 100)$	0.72	0.74	0.70
$M.avg\ (i = 4, N = 100)$	0.74	0.56	0.56
$W.avg\ (i = 4, N = 100)$	0.78	0.79	0.74
$M.avg\ (i = 6, N = 100)$	0.80	0.57	0.57
$W.avg\ (i = 6, N = 100)$	0.81	0.80	0.76

Table 4.34: Avg metrics for incremental ET with $s = 6$, $cr =$ "$gini$"

The accuracy trends, instead, are shown in Figure 4.20 and 4.21.

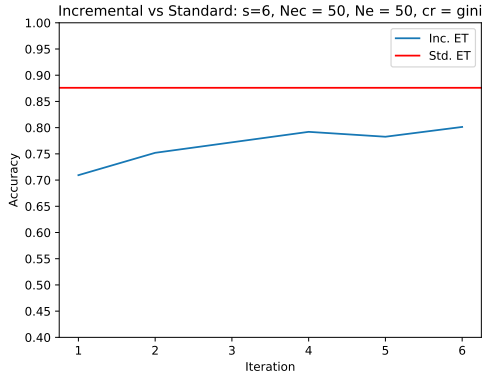

Figure 4.20: Acc. comp. for ET with $s = 6$, $N = 50$, $cr =$ "$gini$"

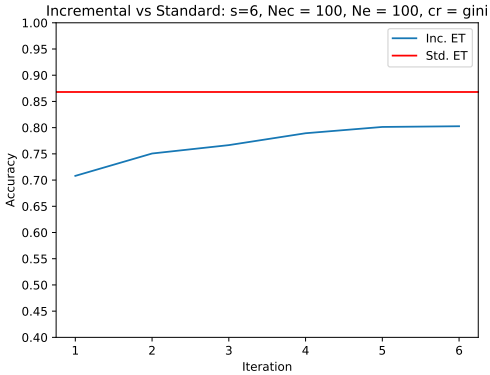

Figure 4.21: Acc. comp. for ET with $s = 6$, $N = 100$, $cr =$ "$gini$"

It can be again said that the accuracies keep degrading, but not as much as it has been observed with the RF experiments, in fact in here they are $\approx 80\%$.

After $cr =$ "*gini*", we considered $cr =$ "*entropy*".
The same tests have been performed, so we are going again to show what has been obtained for $s = 2$.
The obtained metrics are available in Table 4.35.

$i = 1, 2$	*Precision*	*Recall*	F_1
$y = 1$ $(i = 1, N = 50)$	0.85	0.81	0.83
$y = 0$ $(i = 1, N = 50)$	0.56	0.19	0.29
$y = -1$ $(i = 1, N = 50)$	0.85	0.96	0.90
$y = 1$ $(i = 2, N = 50)$	0.88	0.90	0.89
$y = 0$ $(i = 2, N = 50)$	0.62	0.13	0.22
$y = -1$ $(i = 2, N = 50)$	0.88	0.98	0.93
$y = 1$ $(i = 1, N = 100)$	0.87	0.82	0.84
$y = 0$ $(i = 1, N = 100)$	0.47	0.18	0.26
$y = -1$ $(i = 1, N = 100)$	0.85	0.97	0.91
$y = 1$ $(i = 2, N = 100)$	0.88	0.89	0.88
$y = 0$ $(i = 2, N = 100)$	0.60	0.12	0.20
$y = -1$ $(i = 2, N = 100)$	0.87	0.98	0.92

Table 4.35: Metrics for incremental ET with $s = 2$, $cr =$ "*entropy*"

It can be noted that there isn't that much difference with was obtained for the same split factor on $cr =$ "*gini*".
The averaged metrics are instead listed in Table 4.36.

$i = 1, 2$	*Precision*	*Recall*	F_1
$M.avg\ (i = 1, N = 50)$	0.75	0.66	0.67
$W.avg\ (i = 1, N = 50)$	0.82	0.84	0.82
$M.avg\ (i = 2, N = 50)$	0.79	0.67	0.68
$W.avg\ (i = 2, N = 50)$	0.85	0.87	0.84
$M.avg\ (i = 1, N = 100)$	0.73	0.66	0.67
$W.avg\ (i = 1, N = 100)$	0.82	0.84	0.82
$M.avg\ (i = 2, N = 100)$	0.78	0.66	0.67
$W.avg\ (i = 2, N = 100)$	0.84	0.87	0.84

Table 4.36: Avg metrics for incremental ET with $s = 2$, $cr =$ "*entropy*"

Also in here, we want to have a comparison with what is instead produced by the standard ET classifiers using as criterion $cr =$ "*entropy*".
The metrics they produce can be observed in Table 4.37.

Standard	*Precision*	*Recall*	F_1
$y = 1\ (N = 50)$	0.87	0.92	0.89
$y = 0\ (N = 50)$	0.50	0.17	0.25
$y = -1\ (N = 50)$	0.89	0.97	0.93
$y = 1\ (N = 100)$	0.87	0.92	0.90
$y = 0\ (N = 100)$	0.58	0.18	0.28
$y = -1\ (N = 100)$	0.90	0.97	0.93

Table 4.37: Metrics for the standard ET with $cr =$ "*entropy*"

Instead, the averaged values are presented in Table 4.38.

Standard	*Precision*	*Recall*	F_1
$M.avg\ (N = 50)$	0.75	0.69	0.69
$W.avg\ (N = 50)$	0.84	0.87	0.85
$M.avg\ (N = 100)$	0.78	0.69	0.70
$W.avg\ (N = 100)$	0.86	0.88	0.85

Table 4.38: Avg metrics for the standard ET with $cr =$ "*entropy*"

As for the other criterion, it can be said that when the split factor is low, there is almost no difference between the incremental and the standard model.
Again, the behavior of the accuracies has been plotted, and it is shown in Figure 4.22 and Figure 4.23.

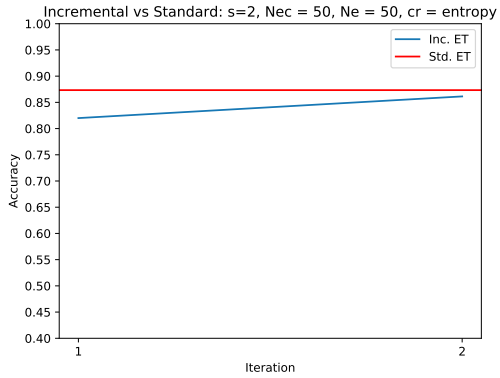

Figure 4.22: Acc. comp. for ET with $s = 2$, $N = 50$, $cr =$ "*entropy*"

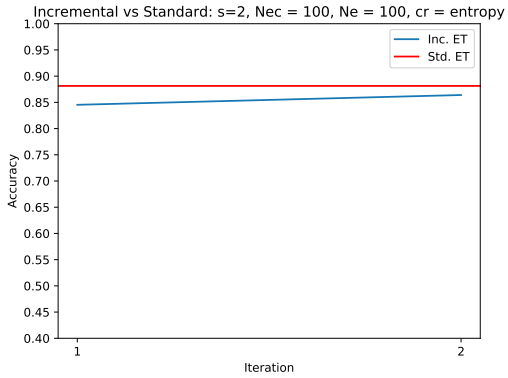

Figure 4.23: Acc. comp. for ET with $s = 2$, $N = 100$, $cr =$ "*entropy*"

The accuracies prove once again what has been said by looking at the various metrics.

The penultimate test we performed was by using as parameters $s = 4$ and $cr =$ "*entropy*".
The metrics produced by the aforementioned configuration can be observed in Table 4.39.

$i = 1, 2, 4$	*Precision*	*Recall*	F_1
$y = 1$ $(i = 1, N = 50)$	0.79	0.64	0.70
$y = 0$ $(i = 1, N = 50)$	0.28	0.12	0.17
$y = -1$ $(i = 1, N = 50)$	0.78	0.93	0.85
$y = 1$ $(i = 2, N = 50)$	0.85	0.72	0.78
$y = 0$ $(i = 2, N = 50)$	0.53	0.10	0.17
$y = -1$ $(i = 2, N = 50)$	0.79	0.97	0.87
$y = 1$ $(i = 4, N = 50)$	0.88	0.83	0.85
$y = 0$ $(i = 4, N = 50)$	0.60	0.04	0.07
$y = -1$ $(i = 4, N = 50)$	0.83	0.99	0.90
$y = 1$ $(i = 1, N = 100)$	0.80	0.62	0.70
$y = 0$ $(i = 1, N = 100)$	0.28	0.12	0.17
$y = -1$ $(i = 1, N = 100)$	0.77	0.94	0.84
$y = 1$ $(i = 2, N = 100)$	0.85	0.69	0.76
$y = 0$ $(i = 2, N = 100)$	0.61	0.14	0.23
$y = -1$ $(i = 2, N = 100)$	0.79	0.97	0.87
$y = 1$ $(i = 4, N = 100)$	0.89	0.78	0.83
$y = 0$ $(i = 4, N = 100)$	0.50	0.05	0.09
$y = -1$ $(i = 4, N = 100)$	0.81	0.99	0.89

Table 4.39: Metrics for incremental ET with $s = 4$, $cr =$ "*entropy*"

Like in the RF and in the ET with $cr =$ "*gini*", there is little decrease in the values with this split factor. Anyway, this loss is not so big, and the results have been considered acceptable.
In Table 4.40, it's possible to see the averaged metrics.

$i = 1, 2, 4$	$Precision$	$Recall$	F_1
$M.avg\ (i = 1, N = 50)$	0.62	0.56	0.57
$W.avg\ (i = 1, N = 50)$	0.73	0.76	0.74
$M.avg\ (i = 2, N = 50)$	0.73	0.60	0.61
$W.avg\ (i = 2, N = 50)$	0.78	0.80	0.77
$M.avg\ (i = 4, N = 50)$	0.77	0.62	0.61
$W.avg\ (i = 4, N = 50)$	0.82	0.84	0.80
$M.avg\ (i = 1, N = 100)$	0.62	0.56	0.57
$W.avg\ (i = 1, N = 100)$	0.73	0.76	0.73
$M.avg\ (i = 2, N = 100)$	0.75	0.60	0.62
$W.avg\ (i = 2, N = 100)$	0.79	0.80	0.77
$M.avg\ (i = 4, N = 100)$	0.73	0.61	0.61
$W.avg\ (i = 4, N = 100)$	0.80	0.83	0.79

Table 4.40: Avg metrics for incremental ET with $s = 4$, $cr =$ "*entropy*"

Also here the accuracies can be checked. For this purpose, they have been plotted and can be observed in Figure 4.24 and Figure 4.25.

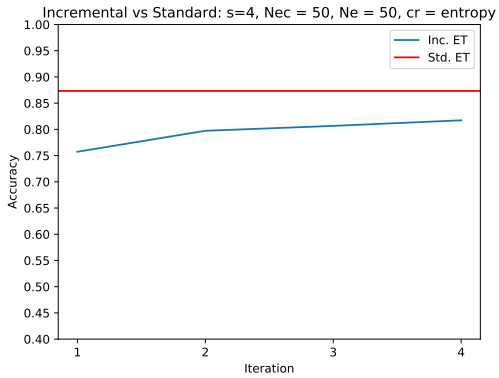

Figure 4.24: Acc. comp. for ET with $s = 4$, $N = 50$, $cr =$ "*entropy*"

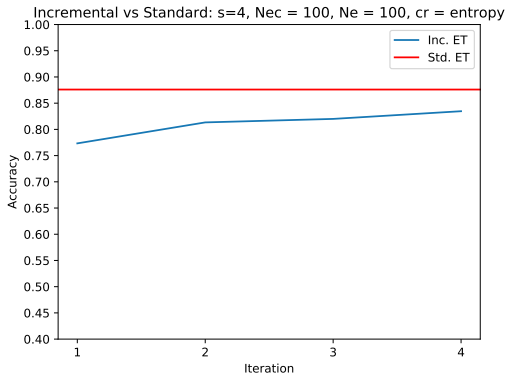

Figure 4.25: Acc. comp. for ET with $s = 4$, $N = 100$, $cr =$ "*entropy*"

The obtained behaviors are not so much different from the ones seen with $cr =$ "*gini*", and also in here a good compromise is given in terms of number of samples used (with $s = 4$, we have that $m = 750$) and accuracy.

The last test was performed with $s = 6$.
In this case, the obtained results are the ones available in Table 4.41.

$i = 2, 4, 6$	$Precision$	$Recall$	F_1
$y = 1$ $(i = 2, N = 50)$	0.83	0.58	0.68
$y = 0$ $(i = 2, N = 50)$	0.40	0.08	0.13
$y = -1$ $(i = 2, N = 50)$	0.74	0.96	0.84
$y = 1$ $(i = 4, N = 50)$	0.86	0.63	0.73
$y = 0$ $(i = 4, N = 50)$	0.33	0.01	0.03
$y = -1$ $(i = 4, N = 50)$	0.76	0.98	0.86
$y = 1$ $(i = 6, N = 50)$	0.88	0.62	0.72
$y = 0$ $(i = 6, N = 50)$	0.75	0.04	0.07
$y = -1$ $(i = 6, N = 50)$	0.76	0.99	0.86
$y = 1$ $(i = 2, N = 100)$	0.84	0.59	0.69
$y = 0$ $(i = 2, N = 100)$	0.38	0.08	0.13
$y = -1$ $(i = 2, N = 100)$	0.75	0.97	0.85
$y = 1$ $(i = 4, N = 100)$	0.87	0.65	0.74
$y = 0$ $(i = 4, N = 100)$	0.67	0.05	0.10
$y = -1$ $(i = 4, N = 100)$	0.76	0.98	0.86
$y = 1$ $(i = 6, N = 100)$	0.90	0.65	0.75
$y = 0$ $(i = 6, N = 100)$	0.67	0.03	0.05
$y = -1$ $(i = 6, N = 100)$	0.76	0.99	0.86

Table 4.41: Metrics for incremental ET with $s = 6$, $cr =$ "*entropy*"

Again, the metrics degrade because of the low amount of data used in the configuration.
This is summarized by the averaged metrics, which are instead available in Table 4.42.

$i = 2, 4, 6$	*Precision*	*Recall*	F_1
$M.avg\ (i = 2, N = 50)$	0.66	0.54	0.55
$W.avg\ (i = 2, N = 50)$	0.74	0.76	0.72
$M.avg\ (i = 4, N = 50)$	0.65	0.54	0.54
$W.avg\ (i = 4, N = 50)$	0.74	0.78	0.73
$M.avg\ (i = 6, N = 50)$	0.79	0.55	0.55
$W.avg\ (i = 6, N = 50)$	0.79	0.78	0.74
$M.avg\ (i = 2, N = 100)$	0.66	0.55	0.56
$W.avg\ (i = 2, N = 100)$	0.74	0.76	0.73
$M.avg\ (i = 4, N = 100)$	0.77	0.56	0.57
$W.avg\ (i = 4, N = 100)$	0.79	0.79	0.75
$M.avg\ (i = 6, N = 100)$	0.78	0.56	0.55
$W.avg\ (i = 6, N = 100)$	0.79	0.79	0.74

Table 4.42: Avg metrics for incremental ET with $s = 6$, $cr =$ "*entropy*"

Finally, the accuracy trends are visible in Figure 4.26 and Figure 4.27.

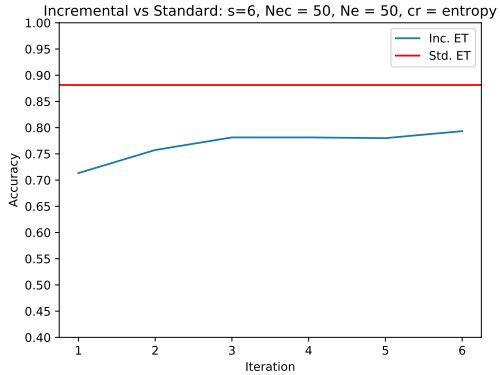

Figure 4.26: Acc. comp. for ET with $s = 6$, $N = 50$, $cr =$ "*entropy*"

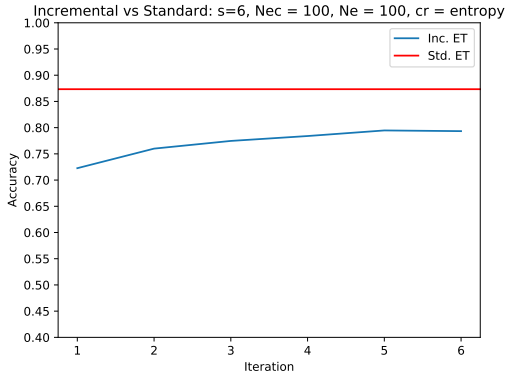

Figure 4.27: Acc. comp. for ET with $s = 6$, $N = 100$, $cr =$ "*entropy*"

These plots lead to the same considerations made for the other criterion. From this split factor on, there is no advantage in going further with the usage of smaller amounts of data: the model would lose a lot in accuracy, because of the computational overhead introduced by the incremental library.

4.2.4 Training times

For what concerns the training times, we decided to populate two tables: one related to the incremental versions, and the other for the standard versions. In the first, we listed the average training time t obtained for the various split factors s and the different values for N_{ec} in an iteration.
Instead, in the second, we just listed the training times of the various standard classifiers over the entire data set (basically $s = 1$).
For the RF and ET classifiers, since the training times with $cr =$ "*gini*" and $cr =$ "*entropy*" are very similar, we decided to list only information related to the first criterion.
The values can be seen in Table 4.42 and Table 4.43.

Incremental	$s = 2$	$s = 4$	$s = 6$
NB	0.018 *sec*	0.010 *sec*	0.011 *sec*
RF $(N = 50)$	0.457 *sec*	0.338 *sec*	0.307 *sec*
ET $(N = 50)$	0.400 *sec*	0.299 *sec*	0.251 *sec*
RF $(N = 100)$	0.772 *sec*	0.597 *sec*	0.510 *sec*
ET $(N = 100)$	0.797 *sec*	0.544 *sec*	0.446 *sec*

Table 4.43: Avg training times for the incremental classifiers

Standard	$s = 1$
NB	0.029 *sec*
RF $(N = 50)$	0.471 *sec*
ET $(N = 50)$	0.495 *sec*
RF $(N = 100)$	0.939 *sec*
ET $(N = 100)$	0.996 *sec*

Table 4.44: Avg training times for the standard classifiers

From the results obtained in the different tests, it is possible to derive a summary table indicating the best configuration that can be adopted for each of the classifiers, and describing whether it is better to follow the incremental approach or the standard one.

The aforementioned table would be Table 4.45.

Classifier	*s*	*N*	*cr*	*Incremental vs Standard*
NB	6	-	-	*Incremental*
RF	4	50	*gini*	*Incremental*
ET	4	50	*gini*	*Incremental*

Table 4.45: Best classifier configurations

It can be noted that, in average, the incremental models - considering a single chunk of data - fit faster than the standard counterparts: this seems to be obvious, since the latter have to fit on the entire data set, while the first ones only on a subset of it.
Anyway, it can be seen that the split factor influences a lot the training times too. The less data is used in the chunk, the faster is the partial fit operation.
In any case, it is important to underline the fact that the incremental models introduce a performance overhead (especially in the Random Forest and in the Extra-Trees classifiers), as stated also in the official documentation of the used tools.
Thus, incremental models should be used carefully and possibly on amounts of data that are not very small: for this reason we have concluded that, in the RF and in the ET classifiers, it is better to use $s = 4$.
In this way, the amount of considered information is not low, and an acceptable performance is provided by the model (considering also the needed training time).

Chapter 5

Conclusions and future work

In this work, we tried to implement a custom incremental learning algorithm for a real world scenario.
This has been mainly done for understanding if standard techniques - e.g. those ones training directly over an entire data set - could be replaced with others like ours, in order to perform retraining operations by only considering *new* training information instead of re-taking into account the whole data set.
Ideally, this would be really advantageous in a context in which infrastructures - e.g. for storing data, for training ML models, ecc. - are paid as a service: in fact, by using this approach, resources can be saved, as well as money.

Here, we implemented three versions of the same algorithm, which differ only in the used classifier: in fact, we adopted a Naïve-Bayes classifier based on a Bernoulli probability distribution, a Random Forest classifier, and an Extra-Trees classifier.
By performing several tests, it has been possible to observe that the Naïve-Bayes classifier, in this context, is not suitable at all: both incremental and standard versions have a really poor performance, which could be partially justified by the high dimensionality of the problem.
The Random Forest classifier, instead, showed a very good performance in a standard configuration - i.e. non incremental -, at least for the labels $y = -1$ and $y = 1$: in fact, there is an overall difficulty in predicting $y = 0$, degenerating almost in a binary classification task. This is not considered

to be a huge problem, since the latter is the less important label, while it is fundamental to behave good with the other two.
This trend has been mimicked by our incremental solution, especially in those situations where the considered amount of data was quite high (e.g. with $s = 2$).
But, it has been also possible to note that whenever the split factor s increases - and so with $s = 4$, but mainly with $s = 6$ - there is a slight worsening of the various metrics: this is due to the fact that the incremental versions are originally built to work with a quite high number of records, and so they suffer whenever the available information is low.
The number of estimators used - in both standard and incremental solutions - is not very significant.
With the Extra-Trees classifier, we arrived at the same conclusions of the Random Forest, since it derives from the latter. The only difference is that it seems to perform a bit better even with a higher value of split factor, becoming thus the preferable choice.

Considering the overall results, we can conclude that for the problem taken into consideration it is better to use a Random Forest classifier or an Extra-Trees one, but paying attention to the used split factor (thus to the amount of samples m to consider in each iteration): in this case it really depends on what we are willing to "sacrifice" a bit, meaning that if a higher accuracy is desired and the time spent for training in a single iteration is not important, then the split factor could be lower and the model would perform better.
Instead, if it is more important to save time in an iteration and there aren't high constraints on the performance, the split factor can be increased and a lower amount of data can be adopted for the partial fit.
Of course, it is also important to say that not all problems are suitable for approaches like this: the available amount of training data is fundamental, and it is preferred to have it high.

With these considerations being made, we want to propose some future work. After having implemented this solution just for testing, it would be interesting and nice to deploy it into a production environment.

For this purpose, tools like Kubeflow [1] could be adopted for defining a pipeline containing the ML model able to perform the incremental training, and for permitting a constant interaction among the model itself and the other components of the platform - e.g. the back-end -.
An additional study that can be performed on this scenario is the adoption of the Hoeffding Trees, which is an incremental decision tree learner well suited for large data streams in where the data distribution doesn't change over time [32].

[1] `https://www.kubeflow.org`

Bibliography

[1] Tom Mitchell. *Machine Learning*. New York, United States of America: McGraw Hill, 1997 (cit. on p. 1).

[2] John R. Koza, Forrest H. Bennett, David Andre, and Martin A. Keane. «Automated Design of Both the Topology and Sizing of Analog Electrical Circuits Using Genetic Programming». In: *Artificial Intelligence in Design '96*. Ed. by John S. Gero and Fay Sudweeks. Dordrecht: Springer Netherlands, 1996, pp. 151–170. ISBN: 978-94-009-0279-4. DOI: 10.1007/978-94-009-0279-4_9. URL: https://doi.org/10.1007/978-94-009-0279-4_9 (cit. on p. 1).

[3] Trevor Hastie. *The elements of statistical learning : data mining, inference, and prediction : with 200 full-color illustrations*. Ed. by Robert Tibshirani; Jerome H. Friedman. New York, United States of America: Springer, 2001 (cit. on p. 6).

[4] James Joyce. «Bayes' Theorem». In: *The Stanford Encyclopedia of Philosophy*. Ed. by Edward N. Zalta. Spring 2019. Metaphysics Research Lab, Stanford University, 2019 (cit. on p. 7).

[5] A. Stuart; K. Ord. *Kendall's Advanced Theory of Statistics: Volume I—Distribution Theory*. London: Edward Arnold, 1994 (cit. on p. 7).

[6] M. Murty Narasimha; V. Devi Susheela. *Pattern Recognition: An Algorithmic Approach*. 2011. ISBN: 978-0857294944 (cit. on p. 8).

[7] George H. John and Pat Langley. «Estimating Continuous Distributions in Bayesian Classifiers». In: *Proceedings of the Eleventh Conference on Uncertainty in Artificial Intelligence*. UAI'95. Montréal, Qué, Canada: Morgan Kaufmann Publishers Inc., 1995, pp. 338–345. ISBN: 1558603859 (cit. on p. 9).

[8] Stuart Russell; Peter Norvig. *Artificial Intelligence: A Modern Approach*. Second. Prentice Hall, 2003. ISBN: 978-0137903955 (cit. on p. 10).

[9] Tin Kam Ho. «Random Decision Forests». In: *Proceedings of the 3rd International Conference on Document Analysis and Recognition*. Montréal, QC, Canada, 1995, pp. 278–282 (cit. on pp. 11, 13).

[10] Lior Rokach; O. Maimon. *Data mining with decision trees: theory and applications*. World Scientific Pub Co Inc., 2008. ISBN: 978-9812771711 (cit. on p. 11).

[11] Shai Shalev-Shwartz; Shai Ben-David. *Understanding Machine Learning*. Cambridge University Press., 2014 (cit. on p. 12).

[12] J.R. Quinlan. «Induction of decision trees». In: *Machine Learning* (1986), pp. 81–106. DOI: 10.1007/BF00116251 (cit. on p. 12).

[13] Leo Breiman; J.H. Friedman; R.A. Olshen; C.J. Stone. *Classification and regression trees*. Monterey, CA: Wadsworth Brooks/Cole Advanced Books Software, 1984. ISBN: 978-0-412-04841-8 (cit. on p. 12).

[14] G. V. Kass. «An Exploratory Technique for Investigating Large Quantities of Categorical Data». In: *Journal of the Royal Statistical Society. Series C (Applied Statistics)* 29.2 (1980), pp. 119–127. ISSN: 00359254, 14679876. URL: http://www.jstor.org/stable/2986296 (cit. on p. 12).

[15] L. Rokach and O. Maimon. «Top-down induction of decision trees classifiers - a survey». In: *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications an Reviews)* 35.4 (2005), pp. 476–487. DOI: 10.1109/TSMCC.2004.843247 (cit. on p. 12).

[16] James Gareth; Daniela Witten; Trevor Hastie; Robert Tibshirani. *An Introduction to Statistical Learning*. New York: Springer, 2015. ISBN: 978-1-4614-7137-0 (cit. on p. 13).

[17] Laurent Hyafil and Ronald L. Rivest. «Constructing optimal binary decision trees is NP-complete». In: *Information Processing Letters* 5.1 (1976), pp. 15–17. ISSN: 0020-0190. DOI: https://doi.org/10.1016/0020-0190(76)90095-8. URL: https://www.sciencedirect.com/science/article/pii/0020019076900958 (cit. on p. 13).

[18] Irad Ben-Gal, Alexandra Dana, Niv Shkolnik, and Gonen Singer. «Efficient Construction of Decision Trees by the Dual Information Distance Method». In: *Quality Technology & Quantitative Management* 11.1 (2014), pp. 133–147. DOI: 10.1080/16843703.2014.11673330. eprint: https://doi.org/10.1080/16843703.2014.11673330. URL: https://doi.org/10.1080/16843703.2014.11673330 (cit. on p. 13).

[19] Tin Kam Ho. «The random subspace method for constructing decision forests». In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 20.8 (1998), pp. 832–844. DOI: 10.1109/34.709601 (cit. on pp. 13, 14).

[20] E.M. Kleinberg. «The random subspace method for constructing decision forests». In: *Annals of Mathematics and Artificial Intelligence* (1990), pp. 207–329. DOI: 10.1007/BF01531079 (cit. on p. 14).

[21] Thomas G. Dietterich. «An Experimental Comparison of Three Methods for Constructing Ensembles of Decision Trees: Bagging, Boosting, and Randomization». In: *Machine Learning* 40.2 (Aug. 2000), pp. 139–157. ISSN: 1573-0565. DOI: 10.1023/A:1007607513941. URL: https://doi.org/10.1023/A:1007607513941 (cit. on p. 14).

[22] Hendrik Blockeel. «Hypothesis Space». In: *Encyclopedia of Machine Learning*. Ed. by Claude Sammut and Geoffrey I. Webb. Boston, MA: Springer US, 2010, pp. 511–513. ISBN: 978-0-387-30164-8. DOI: 10.1007/978-0-387-30164-8_373. URL: https://doi.org/10.1007/978-0-387-30164-8_373 (cit. on p. 14).

[23] L. Kuncheva and C. Whitaker. «Measures of diversity in classifier ensembles». In: *Machine Learning* 51 (2003), pp. 181–207 (cit. on p. 14).

[24] J.J. García Adeva; Ulises Cerviño; R. Calvo. «Accuracy and Diversity in Ensembles of Text Categorisers». In: *CLEI Journal* 8 (2005), pp. 1–12. DOI: 10.19153/cleiej.8.2.1 (cit. on p. 15).

[25] Jennifer A. Hoeting; David Madigan; Adrian E. Raftery; Chris T. Volinsky. «Bayesian model averaging: a tutorial (with comments by M. Clyde, David Draper and E. I. George, and a rejoinder by the authors». In: *Statist. Sci.* 14.4 (Nov. 1999), pp. 382–417. DOI: 10.1214/ss/1009212519. URL: https://doi.org/10.1214/ss/1009212519 (cit. on p. 15).

[26] Javed A. Aslam; Raluca A. Popa; Ronald L. Rivest. «On Estimating the Size and Confidence of a Statistical Audit». In: *Proceedings of the Electronic Voting Technology Workshop (EVT '07)*. Boston, MA, USA, 2007 (cit. on p. 15).

[27] *What is Bagging (Bootstrap Aggregation)*. https://corporatefinanceinstitute.com/resources/knowledge/other/bagging-bootstrap-aggregation/ (cit. on pp. 16, 17).

[28] Raouf Zoghni. *Bagging (Bootstrap Aggregating), Overview.* https://medium.com/swlh/bagging-bootstrap-aggregating-overview-b73ca019e0e9 (cit. on p. 16).

[29] Pierre Geurts, Damien Ernst, and Louis Wehenkel. «Extremely randomized trees». In: *Machine Learning* 63.1 (Apr. 2006), pp. 3–42. ISSN: 1573-0565. DOI: 10.1007/s10994-006-6226-1. URL: https://doi.org/10.1007/s10994-006-6226-1 (cit. on p. 18).

[30] J. D. Hunter. «Matplotlib: A 2D graphics environment». In: *Computing in Science & Engineering* 9.3 (2007), pp. 90–95. DOI: 10.1109/MCSE.2007.55 (cit. on p. 31).

[31] Stephen V. Stehman. «Selecting and interpreting measures of thematic classification accuracy». In: *Remote Sensing of Environment* 62.1 (1997), pp. 77–89. ISSN: 0034-4257. DOI: https://doi.org/10.1016/S0034-4257(97)00083-7. URL: https://www.sciencedirect.com/science/article/pii/S0034425797000837 (cit. on p. 50).

[32] P. Domingos and G. Hulten. «Mining High-Speed Data Streams». In: *KDD* (2000). Ed. by ACM Press., pp. 71–80 (cit. on p. 98).