Analysis of tree and SVM based ensemble classification techniques

Frattin Fabio

272785

Mathematics in Machine Learning

MSc in Data Science and Engineering, Politecnico di Torino

Contents

1 Problem definition

The main goal of my work is to understand which is the impact of some **ensemble** techniques, such as *bagging* and *boosting*, and how they can enhance the performances of a default binary classifier if properly used.

First, a brief analysis of the dataset under consideration will be performed, trying to make it suitable for the task and to understand if we need some pre-processing activities such as *resampling* or *dimensionality reduction*, just to name a couple of techniques.

Then, the theoretical mathematical basis of the two ensemble techniques previously mentioned will be introduced, thus giving a solid background to the further performed analysis.

Such methods will be specifically implemented in the context of tree based classifiers, where those methods are already widely used by the community. A less intuitive but nevertheless interesting attempt will be done also on support vector based classifiers.

Eventually, the results in terms of performance measures will be presented in order to understand the goodness of the ensemble framework.

2 Exploratory data analysis and preparation

The dataset under analysis is specifically suited for the design of a *spam filter*. It contains some statistics about the content of 4601 e-mails, such as the frequencies of some words/characters and the degree of presence of capital letters, along with the label assigned to the mail: spam (1813 samples) or not (2788 samples).

2.1 Attributes

More specifically, all the attributes are reported in the table below.

Attribute No.	Attribute Name	Explanation
1 - 48	word_freq_WORD	frequency of WORD in the e-mail
49 - 54	char_freq_CHAR	frequency of CHAR in the email (specifically, the chars here analyzed are: ";([!$#")
55	capital_run_length_average	average length of uninterrupted sequences of capital letters
56	capital_run_length_longest	length of longest uninterrupted sequence of capital letters
57	capital_run_length_total	total number of capital letters in the e-mail
58	target	denotes whether the e-mail was considered spam (1) or not (0)

2.2 Data cleaning: outliers and correlation

This dataset has no missing values.

The only operation performed directly on the data was an **outlier detection** and deletion. I decided only to manually eliminate those rows containing values having little sense with respect to some attribute, such as negative or unrealistic (more than 100%) frequencies and counts. Since the all the statistics were previously performed by the authors of the dataset, the amount of data labeled as outlier was very limited (5 samples).

The **feature selection** was carried out through an analysis of the correlation among features: if an high (more than 0.9) correlation was detected, one of the two features was dropped. The following word frequencies were dropped: *857, 415*. Moreover, two other features were dropped, since they were too specific for this dataset: the frequencies of the word *george* and *650*.

2.3 Data visualization: KDE

To better visualize which features could be the most imporant in the classification phase, two of the most interesting *kernel density estimates* (KDE) plots are here reported. The KDE technique is a way to estimate the probability density function of a random variable through the empirical data distribution. For each feature, it was estimated both the distribution for spam and for not-spam, separately.

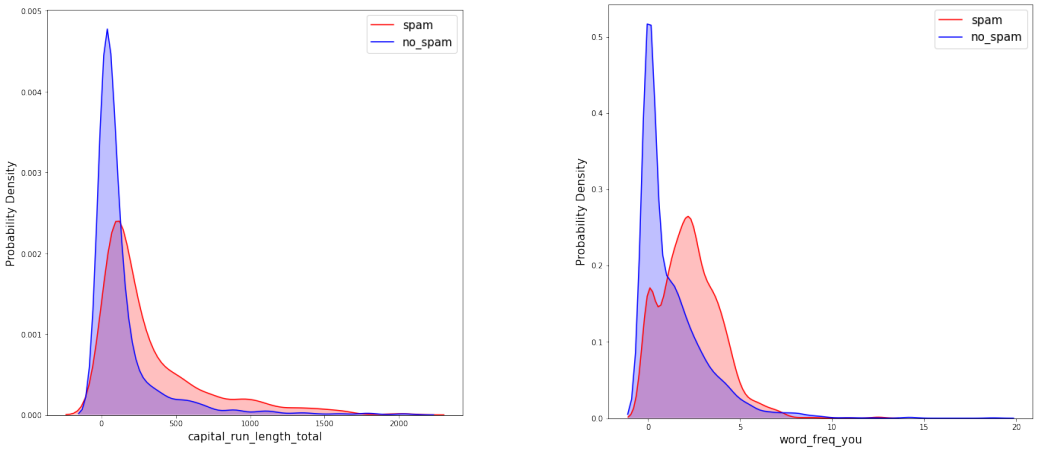

Figure 1: KDE plots for *capital_run_length_total* and *word_freq_you*. The comparison between thr estimated distributions for the two classes show that these values can be useful to predict the class label.

2.4 Data transformation: normalization and dimensionality reduction

To bring all the features' values to the same scaling, a **normalization** process has been carried out. For each attribute, all the values have been subtracted by the mean and divided by the standard deviation in order to have a distribution with vanishing mean and unitary standard deviation.

On the other hand, since our dataset is not very big (around 4500 samples), a **dimensionality reduction** could be helpful. The reduction technique here adopted is the Principal Component Analysis (PCA), whose aim is to represent data in a lower dimensional space preserving as much as possible the proportion of variance explained by the new features obtained, each of which is a linear combination of the existing ones. Obviously, the higher the number of dimensions, the most variance will be kept, but we would like to reduce the dimensionality. Therefore, it is usual to look for a tradeoff between the proportion of variance explained and the number of dimensions: in

my case, I decided to keep the minimum number of dimensions such that the total variance explained was greater than 70% of the total, which resulted in mapping the original space into a 27-dimensional one.

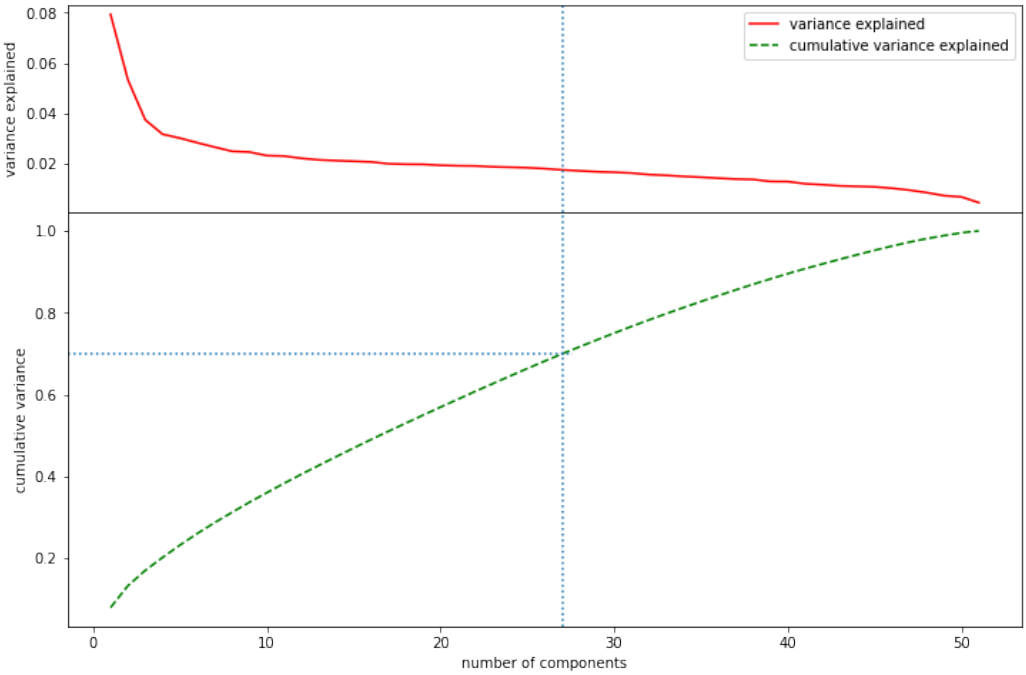

Figure 2: The values of variance explained by each of the new feaures (top), and the cumulative counterpart (below). The blue lines are in correspondence of the value 0.7 for the cumulative variance and 27 for the number of features.

3 Ensemble classification techniques

An *ensemble* technique uses multiple learning algorithms (usually homogeneous, thus belonging to the same "family") to obtain a better predictive performance than a single one.

The main aim of this kind of techniques is to try to tackle the *bias-variance-complexity tradeoff* introduced when learning through the *empirical risk minimization* restricted to a specific hypotesis class. The more expressive the hypotesis class the learner is searching over, the smaller the approximation error (due to the restriction to that specific class), but the larger the estimation one.

Depending on the degree of bias/variance introduced by the single model (called *weak learner*), different ensemble techniques can be used to improve the performances.

3.1 Weak learnability

The weak learner is the base model used as building block to design the final model by combining several of them. They are built such that on their own they either lack in complexity, thus having an high bias, or they are too much variant to be robust. Nevertheless, they are very easy to build and learn, so that it is not a big computational issue to combine many of them.

More formally, a γ-weak learner is a learning algorithm that, under the PAC learning framework in a binary classification context, is able to return an hypothesis whose true error is $\leq 1/2-\gamma$. Differently from strong learnability, where the error is required to be arbitrarily small, in the weak paradigm the hypothesis returned can be just slightly better than random guessing. The lower the γ value, the weaker the learner, the more complex the ensemble technique should be to build an efficient strong learner.

3.2 Bootstrap

The first issue when building many different learners is to understand where we can get the data for a single learner from. One way could be to split the training data into as many splits as the number of learners, but usually the amount of data available is not big enough. Here the *bootstraping* technique comes in handy. Given a standard training set D of size n, when building m sources for the weak learner (usually having the same size n), one simply sample *with replacement* from the original dataset. Each of the new generated set will have roughly 64% of the unique samples of D, the others being

duplicates (not deleted). All these samples can be considered almost independent from each other, their approximation of the true distribution being more accurate the larger n is.

3.3 Bagging

Bagging (short for **B**ootstrap **agg**regat**ing**) is an ensemble technique whose main aim is to reduce the variability of the simple model. First, each weak learner is trained on a boostrapped sample. To make a prediction, the predictions made by all the weak learners are aggregated in a way that depends on the specific task. In a classification context, usually a *majority voting* policy is followed, thus giving uniform importance to all the learners. If the weak learner is able to output predictions in form of probabilities, the predicted class will be the one with the highest average probability.

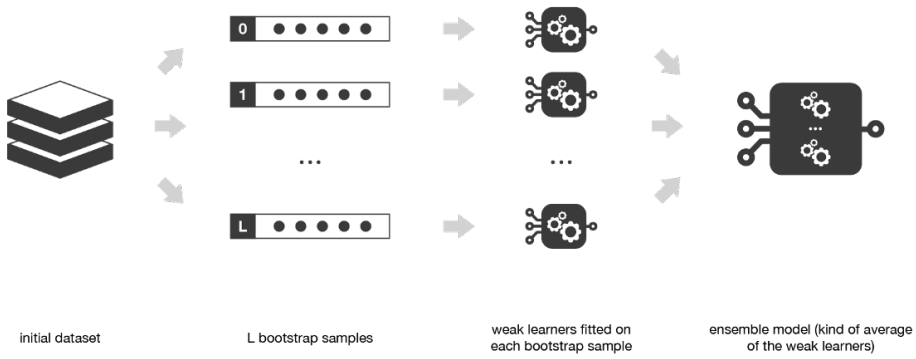

Figure 3: The bootstrap aggregating framework.

The rationale behind bagging is that, since the bootstrapped samples are almost i.i.d., averaging their predictions will not change the expected answer, but it will reduce its variance.

Then, if we have T weak learners, each one outputting a probability $p_i(x)$ for each class $i \in I$ (class set) for a query sample x (for direct predictions, we can consider $p_{trueClass} = 1$ and all the others 0), the ensemble prediction will be:

$$f_{bag}(x) = \underset{i}{\operatorname{argmax}} \left\{ \frac{1}{T} \sum_{t=1}^{T} p_i^t(x) : i \in I \right\}$$

3.4 Boosting

A different approach is followed by the *boosting* framework. Rather than working in parallel, building independent weak learners, the boosting is a **sequential** method, fitting the models iteratively giving more importance to training samples misclassified by the previously trained model. In this way, the boosting tries not only to reduce the variance, but also the bias. Nevertheless, it cannot work in parallel and it is therefore more computationally demanding.

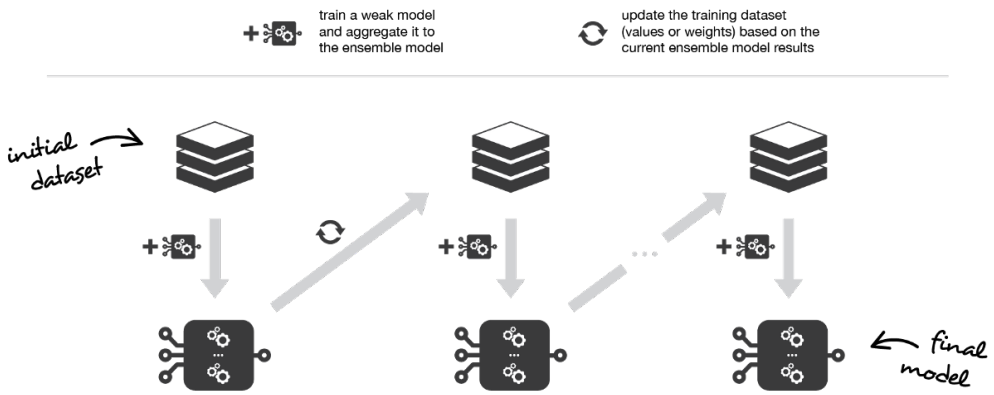

Figure 4: The boosting framework.

Depending on the way the models are sequentially fitted and how their predictions are aggregated, it is possible to define different boosting techniques. More specifically my work focuses on one of those techniques, the *Adaptive Boosting* (AdaBoost).

3.4.1 AdaBoost

The boosting process is composed of a sequence of consecutive rounds. At each step, it is required to define a distribution over the samples S, $D^{(t)}$. The weak learner gets as input the distribution and is required to return an hypothesis h_t, whose error ϵ_t is at most $1/2 - \gamma$, as previously defined.

The *adaptive* policy introduced by AdaBoost is to assign a weight to this weak hypothesis, whose magnitude is inversely proportional to the error. Using this weight, the last step is to update the probability distribution in such a way that the training samples on which h_t makes mistakes will have a higher probability mass. Since this will be the distribution input for the next learner, it introduces an enforcement to focus on the *problematic* samples.

The following pseudocode goes into more details.

AdaBoost

input:

training set $S = (\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_m, y_m)$

weak learner WL

number of rounds T

initialize $\mathbf{D}^{(1)} = (\frac{1}{m}, \ldots, \frac{1}{m})$.

for $t = 1, \ldots, T$:

invoke weak learner $h_t = \text{WL}(\mathbf{D}^{(t)}, S)$

compute $\epsilon_t = \sum_{i=1}^{m} D_i^{(t)} \mathbb{1}_{[y_i \neq h_t(\mathbf{x}_i)]}$

let $w_t = \frac{1}{2} \log\left(\frac{1}{\epsilon_t} - 1\right)$

update $D_i^{(t+1)} = \frac{D_i^{(t)} \exp(-w_t y_i h_t(\mathbf{x}_i))}{\sum_{j=1}^{m} D_j^{(t)} \exp(-w_t y_j h_t(\mathbf{x}_j))}$ for all $i = 1, \ldots, m$

output the hypothesis $h_s(\mathbf{x}) = \text{sign}\left(\sum_{t=1}^{T} w_t h_t(\mathbf{x})\right)$.

Figure 5: The AdaBoost pseudocode functioning.

As the last row of the pseudocode states, the final strong classifier is a combination of the weak hypothesis, whose contribution depends on their degree of correctness. It is then likely that the last learners will be the most important ones.

To summarize, the main differences between *bagging* and *boosting* are:

- the first tackles the variance, the second one mainly the bias, but also the variance if properly set;
- the first can be executed in parallel, while the second one requires a sequential approach;
- the final strong learner always relies on a combination of the weak ones, but in the first case all the contributions are equal, in the second case the contributions are weighted;

4 Application to tree and SVM based algorithms

My analysis focuses on the application of the aforementioned ensemble techniques to two families of classifiers: decision trees and SVMs. I will briefly introduce their characteristics and try to exlain why they could benefit from an ensemble approach (or not).

4.1 Decision Trees

Decision trees classify query samples by proceeding in a root-to-leaf path, where at each node the child (usually between two possible children) is chosen with respect to a split of the input space based on a particular feature and a treshold T defining the nature of the split. The final leaf represents the class the sample is assigned to.

The predictor space is segmented and stratified in a number of distinct and non overlapping regions depending on the splits defined by the tree, thus a *partition* of the space is created. The splits defined by the tree derive from a learning process, where some measures, like the *gini-index*, are defined to establish the goodness of a split based on the relationship between the *entropy* of the parent and children nodes.

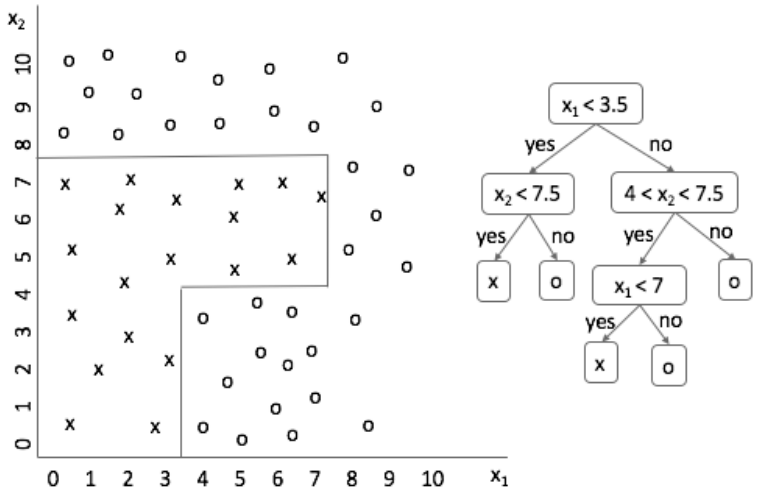

Figure 6: An example of decision tree and the resulting partitioning of the space.

Since each leaf represents a class, a tree with k leaves can shatter any set of k instances. Thus, if we don't limit in some way the number of leaves,

letting the tree growing indefinitely, the **VC dimension** becomes infinite and the method would be prone to overfitting.

That's why usually a first imposed constraint is on the *maximum depth* of the tree, which turns to be also a simple way to affect the variance and the bias of a single tree. Namely, the deeper a tree, the more accurate its predictions on the training set are likely to be, but also it will have an high variance, since it has a lot of splits and a small change in a single feature can lead to a completely different path.

4.1.1 Bagging and Random Forest

To reduce the variance introduced by the complexity of a tree (e.g. its depth), a *bagging* technique seems very suitable. Although the simple bootstrap can enforce the different trees to be almost uncorrelated, a simple but effective *feature bagging* approach can be introduced to enhance this constraint.

More specifically, at **each** split in each tree, a random subset (of size s) of the d features is selected as candidates for the split, rather than selecting among all the features. In fact, it may happen that one or few predictors are so strong that they will be selected by most of the trees, without exploiting all the other predictors and making some trees more correlated than others.

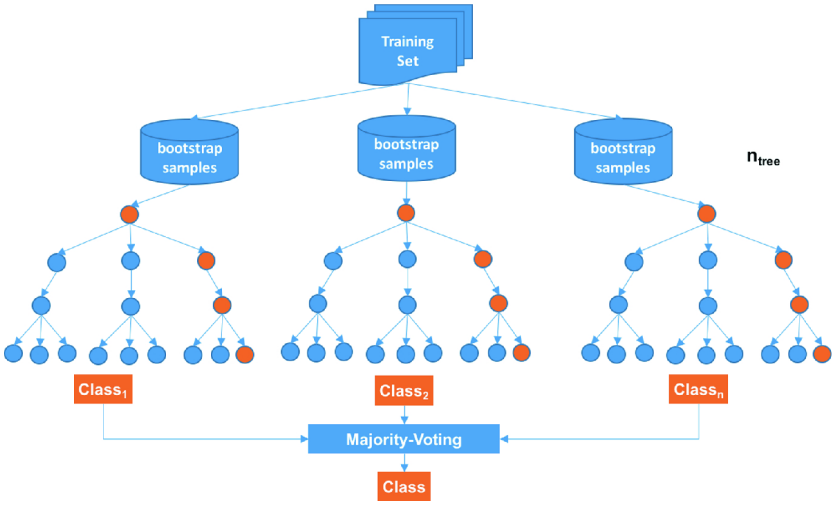

Figure 7: A random forest made up of three trees, following a majority voting policy.

This kind of double bagging technique applied to decision trees is named **Random Forest**, and usually a good number of features is $s = \sqrt{d}$.

4.1.2 Boosting Trees

As previously stated, the *boosting* approach mainly aims at reducing the bias, not the variance. Moreover, it requires simpler (and thus possibly weaker) learner since it cannot work in parallel.

For all these reasons, it is not possible to boost very deep trees. Contrariwise, what the default approach suggests is to use *stumps*, that are the simplest trees one could build, made up of one single split. Such a tree is characterized by an high variance and a low bias, thus it is perfectly suited to be boosted.

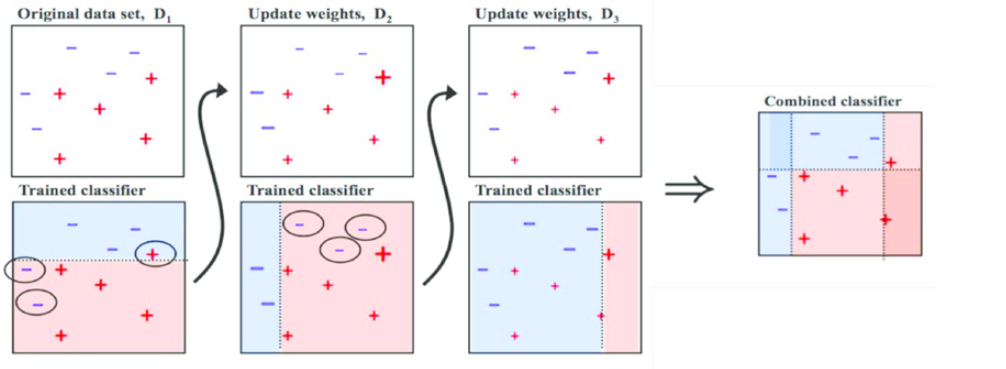

Figure 8: The AdaBoost algorithm applied to stumps. Each stump divides the feature space into two parts, defining a sort of dividing hyperplane parallel to one of the axis.

4.2 Support Vector Machines

SVMs are part of the family of the *halfspaces*, those algorithms that try to find a separating hyperplane between two (or more) classes. A typical halfspace decision function, in the context of binary classification, looks like that:

$$f(x) = \begin{cases} 1 & \text{if } \mathbf{x}^T\mathbf{w} + b > 0 \\ -1 & \text{if } \mathbf{x}^T\mathbf{w} + b < 0 \end{cases}$$

where $\mathbf{w}$ and $\mathbf{b}$ are the parameters defining the hyperplane. Unlike other halfspaces, SVM looks for the hyperplane which *best* separates the two classes, thus the one having the largest *margin*. The margin is defined as the minimum distance between the hyperplane and a training point.

Nevertheless, most of the times data is not perfectly lineraly separable, or could be more easily separable if represented in an higher dimensional space.

To overcome the first problem, the concept of *soft-margin* has been introduced. The hyperplane is allowed to make mistakes, each giving a penalty (ξ_i) to the objective function proportional (C) to its distance from the margin. The new objective function will be a combination of the maximization of the margin and the minimization of the penalties given by the mistakes:

$$\underset{w,b,\xi}{\text{minimize}} \quad \mathbf{w}^T\mathbf{w} + C\sum_{i=1}^{m}\xi_i$$

$$\text{s.t.} \quad y_i(\mathbf{x}_i^T\mathbf{w} + b) \geq 1 - \xi_i, \quad \text{and} \quad \xi_i \geq 0; \quad (i = 1, \cdots, m)$$

The second issue requires to map the training data into another feature space (through some mapping $\phi(x)$), usually higher in dimension to allow for more complexity. Even though it may work from a theoretical point of view, it is usually too much computationally demanding and also useless: as we saw before, at the end what it is really interesting for the SVM are the inner products. Thanks to the *Mercer's Theorem*, we know that there are some functions, defined in the smaller original feature space, that under certain conditions can easily give the results of inner products in higher dimensional spaces. Such functions are known as *kernel functions*.

$$k(x, x') = \phi(x)^T\phi(x')$$

4.2.1 Ensemble of SVMs: feasible?

Trying to apply an ensemble approach to SVMs is not as easy as it could be for decision trees.

The first issue is that, by nature, SVM is a *strong learner* since it requires to solve an optimization problem to be built, thus lacking one of the main features of a weak learner, being easy and fast to build. To overcome this problem, one proposed solution is to train each single SVM with a small amount of data.

On the other hand, they strongly depend on the C parameter (and also others, if the kernel used is parametric), that has to be properly tuned to make a SVM effective, since a wrong value can lead to a very poorly performing classifier. In principle, every weak learner's parameters should be tuned. There are papers of reserach entirely focused on this argument, but it honestly goes beyond the scope of my analysis.

That's why I decided to simply keep it fixed along both the ensemble processes, even though it is not the most proper way to do it.

5 Results

5.1 Implementation details

All the results here reported rely on the python source code available at `https://github.com/frattinfabio/polito-mml`.

Since I just want to compare some methods, I don't need any validaton phase: all the models will be trained on the train data and their performances will be evaluated directly on the test set.

For each of the two families of algorithms (trees and SVM), I decided to inspect the effect of one only parameter , considered to be the most influent, while keeping fixed to their default value all the others: *max_depth* for the decision trees and C, the penalty parameter, for the linear svm;

5.2 Decision Trees

By varying the *max_depth* value, I inspected the accuracy values. Each of the ensemble techniques involves 50 weak learners to make their results comparable.

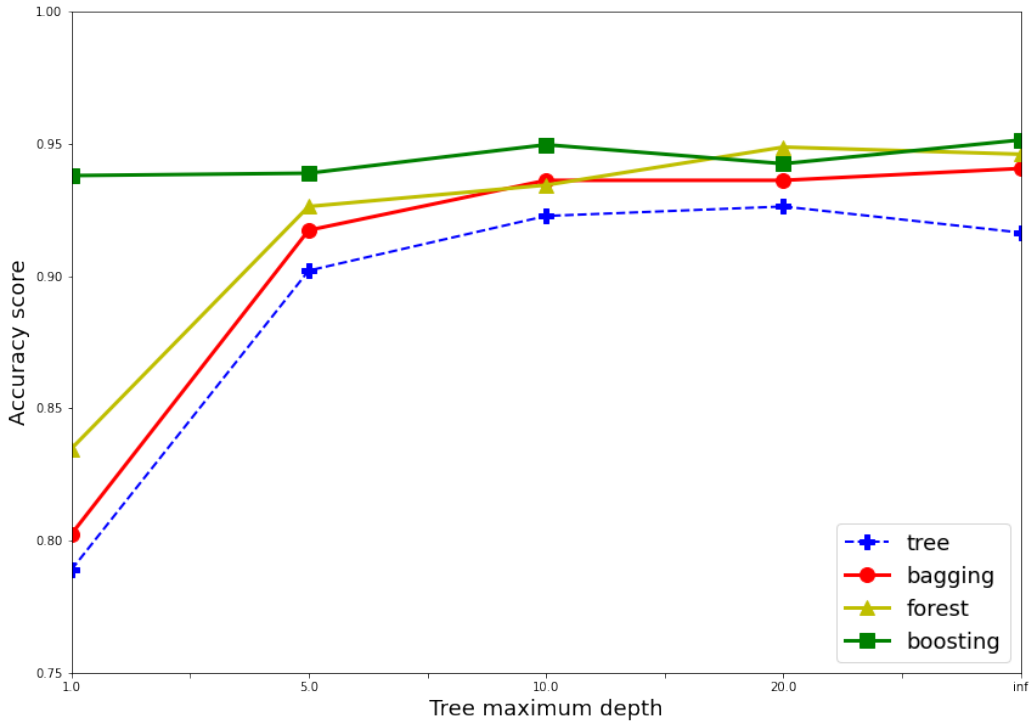

Figure 9: Accucacy results obtained by the different techniques with respect to the maximum depth of the single tree.

From the accuracy graph we can extract some insights:

- each ensemble technique is profitable, meaning it always works better than a simple decision tree;
- the only method able to well perform using only stumps (namely, one split only) is boosting, confirming what previously mentioned about its ability to reduce the bias and correct the mistakes;
- if we let the single tree grow indefinitely, some sort of overfitting occurs, and the performances get worse for a single decision tree. Nevertheless, it looks like an ensemble approach is able to fight this phenomenon;

The results in the table below also shows a good balance in terms of performances on the two different classes, even though the dataset is not perfectly balanced.

Method	**Accuracy**	**Precision**	**Recall**	**F1-score**
Tree	0.910	0.913	0.912	0.912
Bagging	0.936	0.936	0.936	0.936
Random Forest	0.934	0.934	0.934	0.934
AdaBoost	0.951	0.950	0.951	0.951

Table 1: Classification report, fixing the value of *max_depth* to 10.

5.3 SVMs

Even though, as previously mentioned, the ensemble approach with SVMs is not that straightforward, I though it was worth an attempt to see what the results would be by appying the same procedure defined for the decision trees (this time fixing the value of C to 0.5).

The only thing that one could make to try to easily obtain a weaker learner from a SVM is to reduce the number of samples the algorithm is trained with. For this reason, I decided to train each single weak learner on only the 50% of the total number of training data.

As the accuracy graph and the table reporting the classification results show, such techniques require much more attention when applied to SVMs, otherwise they could be counterproductive in terms of accuracy.

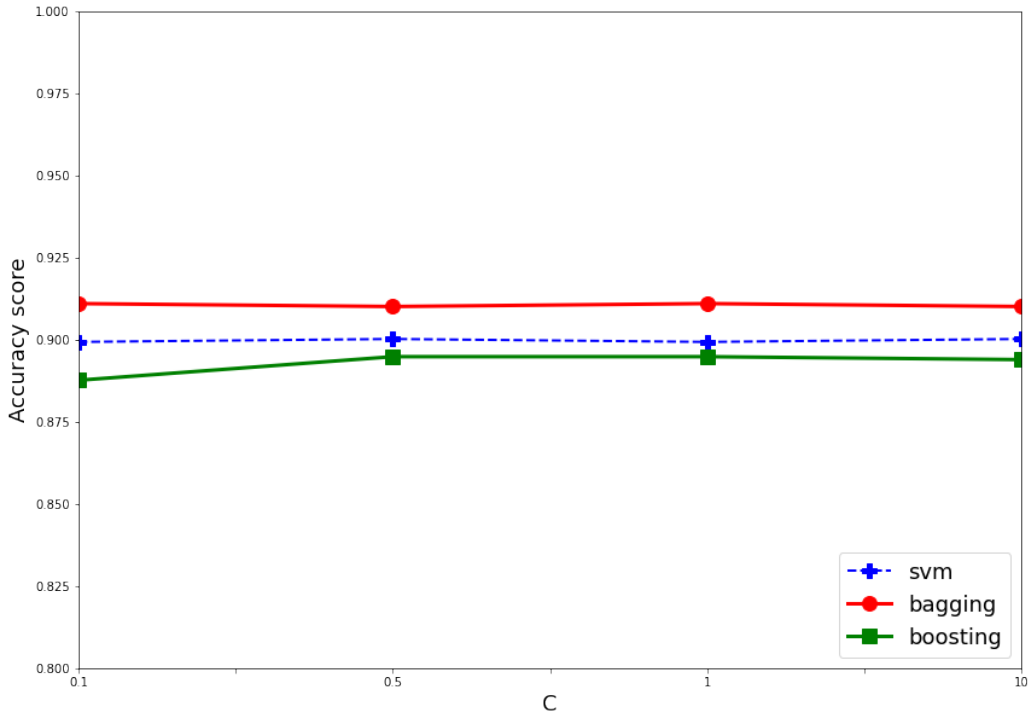

Figure 10: Accucacy results obtained by the different techniques with respect to the penalty coefficient.

Apparently, only a bagging approach can improve the results of a single-learner model, but a more suited implementation (for example, tuning the C parameter for each weak learner) would certainly lead to different results.

Method	**Accuracy**	**Precision**	**Recall**	**F1-score**
SVM	0.914	0.914	0.913	0.914
Bagging	0.916	0.916	0.917	0.916
AdaBoost	0.908	0.907	0.907	0.907

Table 2: Classification report for SVMs

5.4 Comparing weights and errors in AdaBoost

To better understand why the AdaBoost algorithm worked for decision trees and did not for SVMs, I decided to inspect its behaviour during the training phase. More specifically, the *sklearn* implementation of AdaBoost allows to easily retrieve the weight and the error of each single estimator in the ensemble.

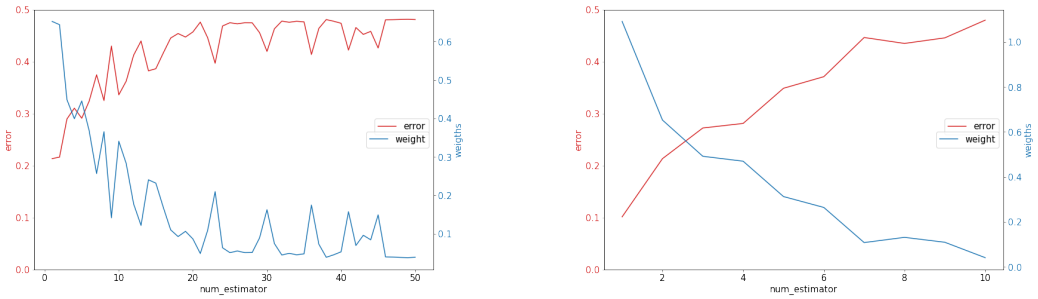

Figure 11: Weight and error for each estimator in the esemble for decision tree stumps (left) and SVM (right). The C parameter for the SVM was kept fixed at 0.5.

From the two grapgh above, we can immediately see that the boosting quickly fails for the SVM, stopping the iterations after having trained 10 estimators out of 50. This happens because the AdaBoost implementation of *sklearn* has some stopping conditions and one of them is when the error returned by the single learner goes above 1/2. This is probably due to the fact that, when changing the underlying distribution, the penalty parameter should be changed and possibly tuned as well.

We can also notice that the weight of the single estimator drops very fast, thus giving much more importance to the first estimator. This is why the performances of the boosted and simple SVM are very similar.

On the other hand, applying the boosting technique to decision stumps makes sense, since all the 50 estimators are exploited. In our case, the first learner is not that weak (achieves almost 80% of accuracy), in fact turns out to be the *heaviest* when performing a prediction.