Extending AdaBoost:Varying the Base Learners and Modifying the Weight Calculation

by

Erico Neves De Souza

Thesis submitted to the
Faculty of Graduate and Postdoctoral Studies
In partial fulfillment of the requirements
For the Ph.D. degree in
Computer Science

School of Information Technology and Engineering
Faculty of Engineering
University of Ottawa

Abstract

AdaBoost has been considered one of the best classifiers ever developed, but two important problems have not yet been addressed. The first is the dependency on the "weak" learner, and the second is the failure to maintain the performance of learners with small error rates (i.e. "strong" learners). To solve the first problem, this work proposes using a different learner in each iteration - known as AdaBoost Dynamic (AD) - thereby ensuring that the performance of the algorithm is almost equal to that of the best "weak" learner executed with AdaBoost.M1. The work then further modifies the procedure to vary the learner in each iteration, in order to locate the learner with the smallest error rate in its training data. This is done using the same weight calculation as in the original AdaBoost; this version is known as AdaBoost Dynamic with Exponential Loss (AB-EL). The results were poor, because AdaBoost does not perform well with strong learners, so, in this sense, the work confirmed previous works' results. To determine how to improve the performance, the weight calculation is modified to use the sigmoid function with algorithm output being the derivative of the same sigmoid function, rather than the logistic regression weight calculation originally used by AdaBoost; this version is known as AdaBoost Dynamic with Logistic Loss (AB-DL). This work presents the convergence proof that binomial weight calculation works, and that this approach improves the results for the strong learner, both theoretically and empirically. AB-DL also has some disadvantages, like the search for the "best" classifier and that this search reduces the diversity among the classifiers. In order to attack these issues, another algorithm is proposed that combines AD "weak" leaner execution policy with a small modification of AB-DL's weight calculation, called AdaBoost Dynamic with Added Cost (AD-AC). AD-AC also has a theoretical upper bound error, and the algorithm offers a small accuracy improvement when compared with AB-DL, and traditional AdaBoost approaches. Lastly, this work also adapts AD-AC's weight calculation approach to deal with data stream problem, where classifiers must deal with very large data sets (in the order of millions of instances), and limited memory availability.

Acknowledgements

No thesis can be written by the author alone. The support of various family members, professors and colleagues also play an important role in this journey.

Firstly, I would like to say my thanks to my wife, Luciane, and my son, Henrique. Both kept me in the right direction during these long years. Without doubt, you both made a real difference in my work and way of thinking.

I also would like to extend my thanks to my parents, Jorge and Maria, and, specially, to my brother Enio, who kindly took some time to review some of the theoretical proofs presented in this work. Without your help, this thesis would not be complete.

My father-in-law and mother-in-law, Joao and Elci, also offered their support from Rio Grande do Sul/Brazil.

My advisor, Prof. Matwin, who offered me an opportunity and I tried to hold on it. Your advises and directions helped to form my ideas and shape this thesis. Thanks for all chances to work with multiple projects that opened my view to multiple experiences.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Classification is a well established problem in Computer Science, and determining the important classes or concepts in the data is an open issue. Humans can detect certain types of patterns easily, as in face recognition, but this is not the case with other types of data; numerical data for example. In this case, machine support helps to detect both patterns and possible changes in the behaviours of numerical data. Various algorithms have been developed to solve these problems, and some of these solutions are based on committees of classifiers, which rely on the premise that a combination of different classifiers could reduce classification errors. One of the solutions proposed to work with decision committees was a new algorithm, known as AdaBoost. AdaBoost is a decision committee algorithm that may be able to improve the execution of a classifier's performance slightly better than 50% accuracy [16]. It also has advantages, such as reducing both the variance among the classifiers and the error rate of the final combination; however, it still relies on the performance of a classifier chosen by the user. This work offers a solution to this problem, by varying the classifiers and improving the policy of the variation to build the final classification algorithm.

Currently, there is no optimal solution to detect all the concepts found in different data sets, though various algorithms, such as Neural Networks and Bayes Networks, have been developed to achieve this objective. Neural networks were created in the 1950s, and are based on how the human brain functions, while Bayes networks are based on a probabilistic approach. They use the Bayes Theorem, which was developed in 1700s and evaluates the data distribution, to calculate the odds that a certain instance belongs to a particular class.

One of the developed strategies is the combination of solutions presented by various learning algorithms to deliver a final result. This approach is based on the common assumption that a group of experts may be more confident that a certain data instance belongs to a specific class than a single specialist would be. One of the first solutions is the Mixture of Experts model [42], which combines the output of various learners in a final hypothesis using an activation formula, such as an exponential function. Another approach is based on a voting strategy, in which various classifiers are combined by majority vote. A first attempt at this was Bagging, which iteratively executes a learner in the training data, applying resampling in the data set instances and features. In this method, the output is a simple majority vote for the classes. This approach motivated a search for solutions that combine various classifiers in which the final output is a weighted majority vote (the algorithm tries to increase the importance of classifiers that performed better than others). AdaBoost is this type of algorithm.

AdaBoost, developed by Freund and Schapire [16], combines certain learners (called weak learners by the authors because their performance is only slightly better than the random classifier) in such a way that, by using the weighted sum of the errors of each classifier in a sequence of rounds defined by the user, the algorithm returns the hypothesis for an instance to belong to a certain class. AdaBoost has been used for different types of problems, including image processing [41] and Network traffic classification [49], with significant results. However, according to its creators, AdaBoost still has two dependency problems: One is its reliance on the weak learner, and the other is the amount of data available for the training.

The dependency concerning the available data required to execute the training is a problem without a solution, because it is highly correlated to the classification problem the ML analysts intend to solve. The problem related to the weak learner used by AdaBoost is the topic this work will address to improve the algorithm, because AdaBoost only uses one classifier that is defined at the beginning of the algorithm, and applies this same learner, varying the weight given, to each training instance. The learner is the same, but the weight of each training instance varies where the algorithm increases the weight of instances that had more errors. The user must test various algorithms with AdaBoost to determine which has improved performance. As it will be presented in this work, AdaBoost also has a problem when trying to improve a classifier with an error rate close to zero, because it will likely not improve the final hypothesis. If the error rate is close to zero, it means that AdaBoost will try to classify possible noise in the data set. Indeed, it will likely decrease the performance of the learner, when the performance of AdaBoost is compared to that of the learner alone.

The main contribution of this work is the development of a new AdaBoost, termed AdaBoost Dynamic, that will improve the results of the original AdaBoost. The key objectives of the work are:

- To analyze the modification of AdaBoost, to allow the use of various learners;

  This modification will help the system to reduce the dependency on the one single learner, since the user must test which learner has some improvement with a particular data set.

  Two approaches to execute the learners are proposed: One considers that the learners are executed in the sequence provided by the user, and the second considers that the "best" learner (the one with smallest error rate for each iteration) is chosen to be executed[1].

- To examine and improve the weight calculation strategy when AdaBoost works with a strong learner.

  In this case, the work will present a comparison of various cost functions that can be used to calculate the weights, and this process will derive a new algorithm. These approaches presented will also offer a solution to the problem of AdaBoost with noisy data sets.

- To combine different weight calculation, allowing the use of strong learners, which are known to reduce the diversity among the classifiers, and assuming the execution of the learners in the sequence provided by the user. The effect will be to execute strong and weak learners without any previous information about their performances on the data set.

- The algorithm is tested in two forms. Firstly, a normal implementation is done to deal with data sets that allow each weak algorithm to access the whole data set. The second implementation considers the algorithm's use with data streams, which do not allow multiple access to the data set, because of high volume of data - on the order of millions of instances for a typical data set. As main results, it can be said that while the AdaBoost Dynamic accuracy results are comparable to traditional AdaBoost, its main advantage is related with model size and evaluation

[1]This will have the effect to reduce the error rate to zero, which will transform the algorithms into strong learners.

time reduction, which optimizes the performance of the algorithm for large amounts of data.

In addition, parts of this thesis were published congresses, which guarantees that the work was evaluated by different group of researchers. Here it is presented the list of publications:

- E. N. de Souza and S. Matwin, "Extending adaboost to iteratively vary its base classifiers", in Advances in Artificial Intelligence, ser. Lecture Notes in Computer Science, C. Butz and P. Lingras, Eds. Springer Berlin/Heidelberg, 2011, vol. 6657, pp. 384-389.
- E. N. de Souza and S. Matwin, "Improvements to adaboost dynamic", in Canadian Conference on AI, ser. Lecture Notes in Computer Science, L. Kosseim and D. Inkpen, Eds., vol. 7310. Springer Berlin/Heidelberg, 2012, pp. 293-298.
- E. N. de Souza and S. Matwin, "Improvements to boosting with data streams", in Canadian Conference on AI, ser. Lecture Notes in Computer Science, O. R. Zaïane and S. Zilles, Eds., vol. 7884. Springer, 2013, pp. 248-255.
- E. N. de Souza, S. Matwin, and S. Fernandes, "Network traffic classification using adaboost dynamic", in Communications Workshops (ICC), 2013 IEEE International Conference on, 2013, pp. 1319-1324.

This work is organized as follows. Chapter 2 presents a literature review of AdaBoost, and its variations. Chapter 3 presents the major contributions of this work, and the main differences between the proposed solutions and the previous works presented in Chapter 2. Chapter 4 presents the results achieved with AdaBoost Dynamic with various data sets. Chapter 5 shows the adaptation of AdaBoost Dynamic algorithm to deal with data stream problem. And Chapter 6 provides conclusions and directions for future work.

Chapter 2

AdaBoost

2.1 A Brief History of Ensemble Methods

The premise for using an ensemble of classifiers is the assumption that a committee of experts can make a more accurate judgment than a single classifier. For example, if a man receives a prescription for medication from a doctor, but he is not fully convinced that it is the correct medication, he has the alternative of seeking the opinion of a different doctor. This process resembles the definition of a decision committee, in that a machine learning analyst tries to combine different learning algorithms to improve the final hypothesis.

The combination process has two basic phases. The first is to choose the learners to be combined, and the second is to determine the correct process to combine them. One of the most important goals when combining classifiers is to maintain the diversity among them [32]. Diversity guarantees that the classifiers are not correlated, so that each classifier will make mistakes in different parts of the data set. The combination process applied to various classifiers is another issue that has been addressed in a number of studies. In this section, previous works that served as the basis for the development of AdaBoost will be presented.

Bootstrap

Bootstraping is a technique for resampling that is used to estimate statistics. As described by Duda *et al* [10], the idea is to resample the data set with replacement. The resampling step is repeated B times, and the accuracy is checked using variance or another statistical process. It is important to note that the algorithm does not define a specific method to combine the classifiers, and it allows the use of any statistical technique to evaluate the accuracy.

Bagging

Bagging [5], a name derived from "bootstrap aggregation" [10], was based on Bootstrap, and uses a different part of the resampled data set in each iteration as input to the classifiers. Thus, each learner is trained with a different part of the training data, and the final output is based on the vote of each classifier. According to Duda et al, Bagging is the first algorithm with multi-classifier systems (i.e. a final output classifier is based on the outputs of a different number of classifiers). Its main advantage is to reduce the variance [43], which, in the case of Decision Trees, helps reduce the potential errors committed by different trees. AdaBoost, in turn, is based on Bagging strategy, but it uses a different approach to combine the classifiers; instead of voting, boosting uses a weighted linear combination of the classifiers.

2.2 AdaBoost

The main objective of AdaBoost is to improve the accuracy of weak classifiers, through the use of a weighted combination of their outputs. The output of this combination is a strong final classifier. AdaBoost.M1 is presented in Table 2.1.

The algorithm works as follows: The input of the algorithm is a set $S = \langle (x_1, y_1), ..., (x_m, y_m) \rangle$ where x_i is an instance drawn from some space X of features, and $y_i \in Y$ is the class label of x_i. The second input is the "weak" learner that will be executed T times, where T is the number of iterations defined by the user. The algorithm starts by creating an array of weights of the same size as the instances, with each position having an equal weight as the value of $\frac{1}{m}$ (step 1). Then, inside the iteration, the algorithm executes the learner (step 3), gets back the hypothesis (step 4), and calculates the error committed (step 5). AdaBoost.M1 checks if the error is larger than $\frac{1}{2}$ and then the algorithm stops, otherwise the algorithm continues executing. In step 8, the algorithm gets the error from the "weak" learner, and uses this information to update the output array (β_t) with $\frac{\varepsilon_t}{1-\varepsilon_t}$, where ε_t where ε_t is the weak learner error in iteration t. In the last step of the iteration, the algorithm updates the weights with the β_t value if the instance class is different than the one predicted with the "weak" learner. As the final output, the algorithm returns the sign of the sum of the $\beta_t f_t$.

Input: sequence of examples m $\langle(x_1,y_1),...,(x_m,y_m)\rangle$ with labels $y_i \in Y = \{1,...,k\}$
the WeakLearner algorithm (f)
integer T specifying number of iterations
1.**Initialize** $D_i = \frac{1}{m}$ for all i
2.**Do for** t=1...T
3. Call WeakLearn, providing it with the distribution D_t
4. Get back hypothesis $f_t : X \to Y$
5. Calculate the error of $f_t : \varepsilon_t = \sum_{i:f_t(x_i)\neq y_i} D_t(i)$
6. If $\varepsilon_t > \frac{1}{2}$, then abort loop
7. Set $\beta_t = \frac{\varepsilon_t}{1-\varepsilon_t}$
8. Update distribution $D_t : D_{t+1}(i) = \frac{D_t(i)}{Z_{t+1}} \times \begin{cases} \beta_t & \text{if } f_t(x_i) \neq y_i \\ 1 & \text{otherwise} \end{cases}$,
where Z_{t+1} is a normalization constant.
end_do
Output:$H_{final} = sign \sum_{t=1}^{T}(\beta_t f_t(x_i))$

Table 2.1: AdaBoost.M1 Algorithm

This section focuses on why AdaBoost.M1 is organized as described. AdaBoost.M1 objective is to approximate the y_i by a linear combination of features. In this sense, the objective is to find a vector $\beta_t \in \Re^n$ such that $F(x_i) = \sum_t \beta_t f_t(x_i)$, where $F(x_i)$ is a "good approximation" of y_i [8], with β_t represents the weight given to a specific classifier in an iteration t, and $f_t(x_i)$ is the "weak" learner hypothesis. The measure of how good this estimator is depends on the task the user is doing [8].

For classification problems the natural approach is to match $F(x)$ to y_i, so the attempt is to minimize [8]

$$\sum_{i=1}^{m}[[y_i F(x_i) \leq 0]] \tag{2.1}$$

where $[[\pi]]$ is equal to 1 if it π is true, or 0 otherwise. Unfortunately, this minimization task is intractable, as presented in [28], and thus it motivates the minimization of a non-negative loss function. AdaBoost.M1 minimizes the exponential loss as follows:

$$L(y, F(x)) = \exp(-yF(x)) \tag{2.2}$$

where $y \in Y$, $F(x)$ is the estimated class returned by the "weak" learner, such that $F(x) \in [-1, 1]$. By expanding the $F(x)$ function

$$L(x, F(x)) = \sum_{i=1}^{m} \exp(-y_i(F_t(x_i) + w_{t+1}f_{t+1}(x_i)))$$

where f_t is the "weak" learner used by AdaBoost in iteration t. For AdaBoost.M1, we wish to minimize the w_{t+1}

$$\begin{aligned}
\arg\min_{w_{t+1},f_{t+1}} \sum_{i=1}^{m} \exp(-y_iF_t(x_i) - y_iw_{t+1}f_{t+1}(x_i)) = \\
\arg\min_{w_{t+1},f_{t+1}} \sum_{i=1}^{m} \exp(-y_iF_t(x_i)) \exp(-y_iw_{t+1}f_{t+1}(x_i)) = \\
\arg\min_{w_{t+1},f_{t+1}} \sum_{i=1}^{m} D_t^i \exp(-y_iw_{t+1}f_{t+1}(x_i))
\end{aligned} \tag{2.3}$$

where $D_t^i = \exp(-y_if_t(x_i))$. The solution to Equation 2.3 can be obtained in two steps. First, for any value of $w_{t+1} > 0$, the solution will be the minimization of

$$f_m = \arg\min_f \sum_{i=1}^{m} D_t^i I(y_i \neq f(x)) \tag{2.4}$$

where $I(y_i \neq f(x))$ is a function that returns 0 if the $y_i = f(x)$, or 1 if $y_i \neq f(x)$. According to Hastie and Friedman[25], Equation 2.4 is the classifier that minimizes the weighted error rate in predicting y. This can be seen by expanding equation presented in (2.3) as shown in Equation 2.5[25]

$$\begin{aligned}
\sum_{i=1}^{m} D_t^i e^{(-y_iw_{t+1}f_{t+1}(x_i))} = \\
\sum_{i=1}^{m} D_t^i e^{w_{t+1}} I(y_i \neq f_t(x_i)) + \sum_{i=1}^{m} D_t^i e^{-w_{t+1}} I(y_i = f_t(x_i)) \\
e^{w_{t+1}} \sum_{i=1}^{m} D_t^i I(y_i \neq f_t(x_i)) + e^{-w_{t+1}} \sum_{i=1}^{m} D_t^i I(y_i = f_t(x_i))
\end{aligned} \tag{2.5}$$

In order to minimize (2.5) the only value that makes it equal to 1 is Equation 2.6 [25]

$$\beta_{t+1} = w_{t+1} = \frac{1}{2} \ln(\frac{1 - \varepsilon_t}{\varepsilon_t}) \tag{2.6}$$

where ε_t is the error identified by the "weak" learner. Now it is possible to plug this result into $F_{t+1}(x)$, that is our accumulative function, in the following way

$$F_{t+1}(x) = F_t(x) + \beta_t f_t(x_i) \tag{2.7}$$

and this will cause the weights in the next iteration to be [25]

$$D_{t+1} = D_t \cdot e^{-\beta_t y_i f_t(x_i)} \tag{2.8}$$

and using the fact presented by [25], that $-y_i f_t(x_i) = 2 \cdot I(y_i \neq f_t(x_i)) - 1$, it can be applied to Equation 2.8

$$D_{t+1} = D_t \cdot e^{2\beta_t \cdot I(y_i \neq f_t(x_i))} \cdot e^{-\beta_t} \tag{2.9}$$

In Equation 2.9, the last term $e^{-\beta_t}$ can be ignored, as it only multiplies the entire formula by the same value. The results can be easily mapped to the algorithm presented in Table 2.1. The stop condition presented in line 6 checks if the error of the "weak" learner is above 50% and Equation 2.6 calculates the final weights presented in line 7. Finally, Equation 2.9, which updates the weights from the instances, is presented in line 8. It is important to note that the algorithm presents $\beta_t = \frac{\varepsilon_t}{1-\varepsilon_t}$, because when $e^{\ln(\frac{\varepsilon_t}{1-\varepsilon_t})} = \frac{\varepsilon_t}{1-\varepsilon_t}$, this can speed up the weight calculation of the misclassified instances.

The final output, $H_{final}(x)$, will be

$$H_{final} = sign \sum_{t=1}^{T} (\beta_t f_t(x_i)) = \arg\max_{y \in Y} \sum_{t: f_t(x) = y} \log \frac{1}{\beta_t} \tag{2.10}$$

The theoretical performance of AdaBoost.M1 is presented in [18], and it is bounded by the Theorem 2.1.

Theorem 2.1 *Suppose that the weak learning algorithm WeakLearn, when called by AdaBoost, generates hypotheses with errors* ε_1 *, ...,* ε_T*. Then the error* $\varepsilon_{final} = Pr_{i \sim D}[F(xi) \neq y_i]$ *of the final hypothesis* f_t *output by AdaBoost is bounded above by*

$$\varepsilon_{final} \leq 2^T \prod_{t=1}^{T} \sqrt{\varepsilon_t (1 - \varepsilon_t)} \tag{2.11}$$

2.2.1 Boosting and Support Vector Machines

AdaBoost has many similarities to Support Vector Machines (SVM). Both algorithms look for instances that better separate the classes. AdaBoost algorithm increases the weights of instances that are incorrectly classified. SVM maps the training data into a higher dimensional space, and then attempts to create a linear separation (hyperplane) between the instances. SVM finds the hyperplane using support vectors ("essential" training tuples) and margins (defined by the support vectors) [24]. The support vectors in SVM can be viewed as instances with higher weight in AdaBoost.

In [15], Freund and Shapire describe the main differences between SVM and Boosting as follows:

- **Computation requirements are different:** According to [15], the computation used in SVM is quadratic programming, while AdaBoost uses linear programming. Freund and Schapire show in [17] that the AdaBoost algorithm is in fact a repeated game, and the minmax strategy should be employed. The way to implement the minmax strategy is to use linear programming[17]. In [38], Schapire also mentions that

  > "AdaBoost seeks to find such a final classifier with high margin on all examples by combining many base classifiers; so in a sense, the minmax theorem tells us that AdaBoost at least has the potential for success since, given a "good" base learner, there must exist a good combination of base classifiers."

- **Methods to search in high dimensional space are different:** One reason for the effectiveness of SVM and AdaBoost is that they are able to find linear classifiers for high dimensional spaces [15]. SVM uses *kernels* to execute the mapping between the original space and the higher dimensional space. The kernels execute an inner product in the original space that is equivalent to calculations in a higher dimension. The boosting approach employs greedy search [15], and from this perspective, the weak learner finds coordinates of $f(x)$ that have a correlation with the label. As presented by Mitchell [34], when instances are ambiguous, they increase the power of the learner to discover the right target function. Ultimately, the main problem for SVM is to choose the right kernel function, and in the case of AdaBoost it is the choice of the "weak" learner.

- **Different norms can result in very different margins:** Boosting and SVM have different norms, and this generates different results for classification. With AdaBoost, there is interpretation in terms of *margins* for the training examples. The margin of an instance (x, y) is defined by Equation 2.12, and is a number defined in the interval $[-1, +1]$ that is positive if it correctly classifies the instance x [15].

$$\frac{y \sum_t (\beta_t f_t(x_i))}{\sum_t \beta_t} \tag{2.12}$$

Assuming that if, for the same instance (x, y), a sequence of weak hypothesis $f(x) = \langle f_1(x), f_2(x), \cdots, f_N(x) \rangle$ and the vector of weights defined by $\beta = \langle \beta_1, \beta_2, \cdots, \beta_N \rangle$, it can be plugged into Equation 2.12, it will find [15]

$$\max_{\beta} \min_{i} \frac{(\beta \cdot f(x_i)) y_i}{\| \beta \| \| f(x_i) \|} \tag{2.13}$$

where the norms for Boosting will be [15]

$$\| \beta \|_1 = \sum_t | \beta_t |$$
$$\| f(x_i) \|_\infty = \max_t | f_t(x_i) |$$

By comparison, SVM has the same goal as described in Equation 2.12, but the norms are Euclidean [15]

$$\| \beta \|_2 = \sqrt{\sum_t \beta_t^2}$$
$$\| f(x_i) \|_2 = \sqrt{\sum_t f_t(x_i)^2}$$

As described in [15], when considering very large dimensions, this difference in the norms creates a significant difference in the margin values.

2.3 Improvements to Boosting

Various authors have tried to improve the original AdaBoost algorithm, and this section presents some of these important works. An attempt by Freund and Shapire, known as

AdaBoost.M2 [18], is intended to improve confidence in AdaBoost results. Another is LogitBoost [19], which helps to view the AdaBoost approach as an additive logistic regression. In MarginBoost [33], the authors present a different view of AdaBoost in terms of a gradient descent search, and provide two key theorems that generalize the convergence of *any* AdaBoost-like algorithm. AdaCost [12] presents a modification that makes AdaBoost cost sensitive, using a cost matrix defined by the user. Another approach is the MultiBoosting algorithm [43], which combines AdaBoost with Bagging and Wagging. The SMOTEBoost algorithm [7] attempts to improve the accuracy of AdaBoost by executing the SMOTE algorithm in each iteration step, thereby modifying the data distribution in the iterations. There is also RareBoost [30], which uses a combination of weights to improve the accuracy for unbalanced data sets. The 2-AdaBoost approach [21] uses multiple learners with boosting, and combines two learners into AdaBoost. Finally, OzaBoost [36] is presented a strategy to combine various learners in parallel and merge their results with AdaBoost.M1 weighted sum. OzaBoost is an algorithm developed to work with data streams.

2.3.1 AdaBoost.M2

This extension was proposed by Freund and Shapire in [18]. Their objective was to improve the confidence of the final hypothesis generated by the weak learner. In the authors' words, [18] this modification

> "In particular, it sometimes enables the weak learner make useful contributions to the accuracy of the final hypothesis even when the weak hypothesis does not predict the correct label with probability greater than 1/2."

Freund and Shapire surmised that if a certain weak learner generates hypotheses that have the same value for all classes, this weak learner is *uninformative* [18]. Conversely, if the weak learner can vary its response depending on the class, it could provide improved accuracy, as it could predict if a certain label is more plausible than another [18]. In order to assess this information the authors implemented two small modifications to the original AdaBoost.M1 (Table 2.1). Initially, it was defined that $f_t(x_i, y_i)$ returns a value inside the interval $[0, 1]$, and therefore we can interpret this function as a randomized decision to detect the right label. Thus, the first modification to AdaBoost.M1 is the error calculation, as presented in Equation 2.14

$$\varepsilon_t = \frac{1}{2}\sum_{i=1}^{m} D_t(1 - f_t(x_i, y_i) + \sum_{y \neq y_i} q(i, y) f_t(x_i, y)) \tag{2.14}$$

where $q(x_i, y_i)$ is a function that gives an importance level to the possible labels under evaluation. For example, if the predicted label from $f_t(x_i, y_i)$ is different from the real value, $q(x_i, y_i)$ assigns a weight that helps the algorithm recognize the correct label. In terms of implementation, the function $q(x_i, y_i)$ is considered equal to 1, reducing the Equation 2.14 into $\varepsilon_t = \frac{1}{2}D_t(1-f_t(x_i, y_i)+f_t(x_i, y))$ in each iteration step. One important observation is that ε_t will always be smaller than $\frac{1}{2}$, which eliminates the stop condition in AdaBoost.M1.

The second modification integrates ε_t into the weight calculation D_t, and the function takes into consideration the new ε_t calculated in Equation 2.14, as presented in Equation 2.15.

$$D_t : D_{t+1}(i) = \frac{D_t(i)}{Z_t} \times \begin{cases} \beta_t^{\varepsilon_t} & \text{if } h_t(x_i) = y_i \\ 1 & \text{otherwise} \end{cases} \tag{2.15}$$

As mentioned by Freund and Schapire this pseudo-loss function (Equation 2.14)[18]:

> "... is generally more successful when the weak learners use very restricted hypotheses. However, for more powerful weak learners, such as decision-tree learning algorithms, there is little difference between using pseudo-loss and prediction error."

2.3.2 LogitBoost

In [19], Friedman *et al* propose that when AdaBoost is applied to a two-class problem, it is an approximation to additive logistic regression using Bernoulli likelihood as a criterion. The fact that boosting fits an additive model $\sum_t f_t(x)$, explains why it tends to be better than a single learner. They also argue that it is possible to attempt to fit an additive model, in order to minimize the squared error loss $(E(y - \sum_t f_t(x))^2)$. This idea would be implemented in the following way: Assuming in tth iteration one can fix all $f_1(x), \cdots, f_{t-1}(x)$, then it is possible to minimize the error to obtain $f_t(x) = E(y - \sum_{t=1}^{T-1} f_t(x)|x)$[19]. As shown by the authors, this solution does not work correctly, and boosting approximates a logistic regression model similar to the binomial log-likelihood

(If $p(x)$ are the class probabilities, then additive logistic regression model approximates $\log \frac{p(x)}{1-p(x)}$ by an additive function $\sum_t f_t(x)$ [19]).

Friedman *et al* also provide the following motivation for using $e^{-yF(x)}$ as the minimization cost function employed in boosting [19]:

- $e^{-yF(x)}$ is a differentiable function.
- The function is sensible to population minimizer, i.e. it answers the question: "What does a minimization of expected loss estimate on a population level?". This cost function predicts the minimization of the expected risk $F^*(x)$ in the form, according to Lemma 1 in [19]

$$F^*(x) = \arg\min_{F(x)} E_{yx} e^{-yF(x)} = \frac{1}{2} \log \frac{\Pr(y=1|x)}{\Pr(y=-1|x)}$$

After proving that AdaBoost actually fits the additive logistic regression, the authors presented LogitBoost, which modifies the original Boosting. They initially considered this to be a two class problem, where the probability of $y = 1$ is given by Equation 2.16. Table 2.2 presents the LogitBoost algorithm. The main difference is related to the calculation of weights that use the Bernoulli log-likelihood. LogitBoost accepts the same parameters as AdaBoost.M1, but it also requires a threshold called $Z\_Max \in [2, 4]$. Z_Max was empirically created because of numerical problems related to the Bernoulli formula. Since the weights will be very small when the probabilities $p(x)$ are very close to 0 or 1, the authors created this threshold to ensure that the probabilities are correctly calculated. The recommended interval is also given by [19].

The algorithm iterates in all instances (line 3), calculating the weights of all instances according to the specified Z_Max formula. Then a decision is made: If the label is equal to 1, the algorithm updates the z to $\frac{1}{p(x_i)}$ and the threshold is verified (lines 5 and 6). Otherwise, if the label is different from 1, the z will be equal to $\frac{-1}{1-p(x)}$ (line 8), and another check is made to verify if this value is below the threshold: if it is z will be equal $-Z\_Max$. Line 10 updates the weights for all instances, and in line 11 the function h_t is approximated by a weighted least-squares regression of z to x_i, using weights D_i. In line 12, the final output function $F(x)$ is updated, and the probabilities $p(x_i)$ are also approximated by Equation 2.16. The algorithm is not exactly equal to that proposed by [19], because the authors did not present Steps 4 to 10 in detail. In order to address these gaps, it was decided to show the steps as implemented in Weka software [23].

Input: sequence of examples m $\langle(x_1, y_1), ..., (x_m, y_m)\rangle$ with labels $y_i \in Y = \{1, ..., k\}$

the WeakLearner algorithm

integer T specifying number of iterations

integer $Z\_Max \in [2, 4]$

1.**Initialize** $D_i = \frac{1}{k}$ and the probability estimates $p(x_i) = \frac{1}{2}$ for all i and z = 0

2.**Do for** t=1...T

3. **Do for** i=1...m

4. **if** $(y_i = 1))$ then

5. $z = \frac{1}{p(x_i)}$

6. **if** $(z > Z\_Max)$ $z = Z\_Max$

7. else

8. $z = \frac{-1}{1-p(x_i)}$

9. **if** $(z < -Z\_Max)$ $z = -Z\_Max$

end_if

10. $D_i = \frac{y-p(x_i)}{z}$

end_do

11. Fit the function $h_t(x)$ by a weighted least-squares regression of z to x_i using weights D_i.

12. Update $F(x) = F(x) + \frac{1}{2}h_t(x)$ and update all probabilities with Equation 2.16

end_do

Output:$H_{final} = sign F(x)$

Table 2.2: LogitBoost Algorithm for two class problems.

$$\Pr(y = 1|x) = \frac{e^{F(x)}}{e^{F(x)} + e^{-F(x)}} \tag{2.16}$$

It is important to note that LogitBoost is a Bernoulli log-likelihood approximation because, as shown in [19], the algorithm updates $f_t(x)$ by a weighted least-square regression of z to each instance using weights D_i, in the following form

$$z_i = \frac{y_i - p(x_i)}{p(x_i)(1 - p(x_i))}$$

Another important contribution of [19] is generalizing the use of AdaBoost and LogitBoost to the multi-class problem, by allowing the algorithms to execute in the form of one class against all others. This allows the logistic assumptions for the two-class problem to remain valid to the multi-class problem.

2.3.3 MarginBoost

Mason *et al* [33] point out that various studies show that AdaBoost is successful because it produces large margin classifiers. The margin of an instance is defined as the total weight assigned to the correct label minus the largest weight assigned to the wrong label. Margin can be seen as confidence in the correct classification; that is, an example is classified correctly only if it has a positive margin [33]. Considering this, the authors propose MarginBoost, which combines classifiers to optimize the sample average of any cost function of the margin [33].

The premise of MarginBoost is to make a gradient descent search and execute a linear combination of elements, in order to minimize some cost function. It is important to note that the theoretical proposal does not link MarginBoost to a specific cost function (such as the exponential cost used by AdaBoost). This contribution creates a general framework, in which *any* type of cost function that follows a set of constraints can be used to implement a variation of AdaBoost.

Beginning with a general description of `AnyBoost` (see in Table 2.3), which is a theoretical algorithm that has the same principles as AdaBoost, the authors derive their main conclusions about theoretical basis that governs a generic Boosting algorithm. Their first assumption is to concentrate on a combination of classifiers that have the same form as Equation 2.10, and the margin of an instance (x, y) is defined as $yH_{final}(x)$[33].

Mason *et al* promote the idea of using a voted classifier that minimizes the cost function applied to the margin presented in Equation 2.17. This cost function will be $C : \mathbb{R} \mapsto \mathbb{R}$. The authors show that Boosting must use a function C that decreases most rapidly.

$$C(H_{final}(x)) = \frac{1}{m} \sum_{i=1}^{m} C(y_i H_{final}(x_i)) \tag{2.17}$$

The goal is to produce a weighted combination of classifiers that optimizes Equation 2.17. The authors introduce a more abstract treatment, which shows that Boosting methods can be viewed as a gradient descent in an *inner product* way. Some modifications are necessary to achieve this objective and process it correctly. As mentioned in [33], assuming:

- a set S of instances;
- a set of base hypothesis $f_t \in \mathbb{F}$. Since $f_t : X \mapsto [-1, +1]$, then $\mathbb{F}$ is a set of base classifiers; and,

- the combinations of f as inner products F, such that $(S, \langle,\rangle)$. In this case S is a linear set of linear space functions that contain $lin(F)$, the set of all linear combinations of functions in $\mathbb{F}$. Then the inner product is defined by

$$\langle F(x), G(x)\rangle := \frac{1}{m}\sum_{i=1}^{m} F(x_i)G(x_i)$$

for all $F(x) and G(x) \in lin(F)$ [33].

Suppose there is a function $F(x) \in lin(F)$, and it is necessary find a new $f_t \in \mathbb{F}$, such that the cost $C(F + \varepsilon f_t)$ decreases for some small value ε. In this case, the direction of f_t is searched so that $C(F + \varepsilon f_t)$ is decreased, and this approach is simply the negative derivative of $\bigtriangledown C(F)(x)$, such that :

$$\bigtriangledown C(F)(x) = \frac{\partial C(F + \alpha 1_x)}{\partial \alpha}|_{\alpha=0} \tag{2.18}$$

where 1_x is a function that returns 0 for $f_t(x) \neq y$, and 1 if $f_t(x) = y$. However, to choose a new function f_t from $\mathbb{F}$ is normally not possible [33], so to solve the problem, the search is constrained in the greatest inner product with $- \bigtriangledown C(F)$, such that it is possible to maximize $-\langle F(x), f_t\rangle$.

The above discussion requires some explanation about the algorithm presented in Table 2.3. The inputs are equal to AdaBoost (Table 2.1), the weights (D_t) have a uniform distribution, and the accumulator $(F(x))$ has all values set to zero. The algorithm calls the WeakLearner and collects the error (lines 3 to 5), then calculates the derivative of the cost function (line 6). At this point, the algorithm checks the orientation of the new f_t (the "weak" learner) with the inner product, using the accumulator. If it is higher or equal to zero the derivative is not decreasing, and the algorithm can stop. The weight update (D_{t+1}) is in line 7, with $\frac{-\langle \bigtriangledown C(F(x), f_t)\rangle}{L||f_t(x)||^2}$, where $L > 0$ such that $|| \bigtriangledown C(F(x)) - \bigtriangledown C(F'(x))|| \leq L||F(x) - F'(x)||$ for all $F, F' \in \mathbb{F}$, characterizes this function as a Lipschitz differentiable cost functional [33]. In line 8 the algorithm update the accumulator. The output is the sign of $F(x)$.

Mason *et al* [33] state that this algorithm will cause overfitting, as it can return an arbitrary linear combination of elements of the "weak" learner. This was the motivation to improve the algorithm with a second version, in which the accumulator $F(x)$ is normalized by the sum of the weights (D_t), and the stop condition is modified to

Input: sequence of examples m $\langle(x_1, y_1), ..., (x_m, y_m)\rangle$ with labels $y_i \in Y = \{1, ..., k\}$ the WeakLearner algorithm integer T specifying number of iterations
1.**Initialize** $F(x) = 0$ for all i, and $D_i = \frac{1}{m}$ for all i 2.**Do for** t=1...T 3. Call WeakLearn, providing it with the distribution D_t 4. Get back hypothesis $f_t : X \to Y$ 5. Calculate the error of $f_t : \varepsilon_t = \sum_{i:f_t(x_i)\neq y_i} D_t(i)$ 6. If $-\langle \bigtriangledown C(F(x), f_t)\rangle \leq 0$, return $F(x)$ 7. Choose a new $D_{t+1} = \frac{-\langle \bigtriangledown C(F(x), f_t)\rangle}{L\|\|f_t(x)\|\|^2}$ 8. $F(x) = F(x) + D_{t+1} f_t(x)$ **end_do** **Output:**$H_{final} = signF(x)$

Table 2.3: AnyBoost Algorithm

$-\langle \bigtriangledown C(F(x)), f_t - F(x)\rangle$ [33], which makes the final proposal from MarginBoost consistent with Table 2.4. Mason *et al* [33] calculated the following results for the stop condition and weight calculation respectively:

$$-\langle \bigtriangledown C(F(x)), f_t - F(x)\rangle = \sum_{i=1}^{m} D_t y_i f_{t+1}(x_i) \text{ and } D_{t+1} = \frac{C'(y_i F(x_i)}{\sum_{i=1}^{m} C'(y_i F(x_i)}$$

Slightly different from the original AnyBoost algorithm is the use of a new variable ε, which has the same effect as the learning rate does during the weight update in Neural Networks: that is, to moderate the degree of change of the weights. In the case of MarginBoost, this variable is defined as a fixed small number [33].

Again, it should be noted that the presented algorithms are not linked to a specific cost function. Mason *et al* [33] prove the convergence of the algorithms, and derive two main restrictions to the cost functions: i) if the weak learner can always find the best weak hypothesis in each round, and ii) if the cost function is convex, then AnyBoost is guaranteed to converge to a global minimum [33]. These results are the reason for some of the adaptations presented in Chapter 3. Also, in [33] the authors tested with the derivative of the cost *tanh*. Though the results were better than with AdaBoost, it takes over 1,000 iterations to achieve the performance described in their paper.

Input: sequence of examples m $\langle(x_1, y_1), ..., (x_m, y_m)\rangle$ with labels $y_i \in Y = \{1, ..., k\}$ the WeakLearner algorithm integer T specifying number of iterations
1.**Initialize** $F(x) = 0$ for all i, and $D_i = \frac{1}{m}$ for all i 2.**Do for** t=1...T 3. Call WeakLearn, providing it with the distribution D_t 4. Get back hypothesis $f_t : X \to Y$ 5. Calculate the error of $f_t : \varepsilon_t = \sum_{i:f_t(x_i) \neq y_i} D_t(i)$ 6. If $-\langle C(F(x), f_t - F(x))\rangle \leq 0$, return $F(x)$ 7. Choose a new weight ε 8. $D_{t+1} = \frac{C'(y_i F(x_i)}{\sum_{i=1} mC'(y_i F(x_i)}$ 9. $F(x) = F(x) + \varepsilon f_t(x)$ **end_do** **Output:**$H_{final} = signF(x)$

Table 2.4: MarginBoost Algorithm

2.3.4 AdaCost

Another approach to improve AdaBoost is presented by Fan *et al* [12], who convert AdaBoost into a misclassification cost-sensitive algorithm. The authors are aware that AdaBoost already assigns high initial weights to costly examples (i.e. instances that are misclassified). However, they believe the weight updating rule should also consider the cost of misclassification into account, and increase the weights of misclassified instances more than traditional boosting, and decrease the weights of correctly classified instances less, compared to AdaBoost. The reference algorithm in Table 2.1 explains the major changes introduced by Fan *et al.*

The first change was to add a new variable, $c_i \in \mathbb{R}^+$, to the input set. The variable contains the cost of the misclassification of a specific instance, so the new input is $S = \langle(x_1, c_1, y_1), ..., (x_m, c_m, y_m)\rangle$. The other modification was to create a cost function, $\beta(sign(y_i f_t(x_i), c_i)$, that has a cost adjustment function with two arguments: $sign(y_i f_t(x_i))$ to show if $f_t(x_i)$ is correct, and the cost factor c_i [12]. This function transforms line 8, in Table 2.1 into the following formula

$$D_t : D_{t+1}(i) = \frac{D_t(i)}{Z_t} \times \beta(i) \text{ where } \beta(i) = \text{sign}(y_i f_t(x_i), c_i)$$

The aim, as explained by Fan *et al*, is to make the function increase the weight "more" if there is a misclassification for a particular case of an instance with a high cost factor, and decrease the weight "less" if the instance is misclassified [12]. Assuming the data set is a two-class problem, this function $\beta(i)$ can be divided into two spaces, such that $\beta_t^+(sign(y_i f_t(x_i)) = 1$ and $\beta_t^-(sign(y_i f_t(x_i)) = -1$.

As suggested in [12], for some problems it may be necessary to adjust the weights, first by initially normalizing, and then by using the $\beta_t^+(sign(y_i f_t(x_i)) = -0.5c_i + 0.5$, and $\beta_t^-(sign(y_i f_t(x_i)) = 0.5c_i + 0.5$ that offer good results in most applications. These coefficients can be adjusted according to specific needs.

The main problem with this approach is the requirement to define the cost of the errors for each instance in advance. In various implementations, the user can set up a *cost matrix* that contains the cost of misclassifying the classes, but they still need an in-depth knowledge of the data set.

2.3.5 MultiBoosting

MultiBoosting [43], as proposed by Webb, combines three different approaches: AdaBoost, Bagging [5] and Wagging [1]. Bagging, mentioned in Section 2.1, is different from boosting in two ways: i) it uses a bootstrap approach to choose the training data to be used by the learner in an iteration, and ii) the final output is a voted combination of the classifiers. Bootstrap resamples 63% of the training data in each iteration [5], while AdaBoost does not do resampling directly on the training data. Instead, it changes the weight distribution and leaves the training data untouched.

Wagging is a variant of Bagging [43]. Rather than using a bootstrap technique, it generates random weights for the instances in each training iteration. Bauer and Kohavi's [1] original implementation used Gaussian noise, but they found that some instances had their weight reduced to zero, which triggered the algorithm to remove them from the training set. To address this, they applied a continuous Poisson (Equation 2.19) distribution to assign the weights [1].

$$Poisson() = -\log\left(\frac{Random(1..999)}{1000}\right) \tag{2.19}$$

MultiBoosting combines Wagging in AdaBoost. The inputs of the MultiBoosting algorithm (presented in Table 2.5) are the set of instances (S), the "weak" learner, the number of iterations, and an array of integers (I): these are used to determine when to apply Equation 2.19. The first step in the algorithm is to set all weights to a value of 1, and initialize the variable k to 1. Next, the algorithm begins executing inside the loop, and checks if the first position of the array I is equal to the specified iteration defined by the user to update all weights with the random value calculated from Equation 2.19. If it is equal, the algorithm updates the values (line 5), normalizes them (line 6) and increments the index k (line 7). Otherwise, the algorithm executes the "weak" learner (line 8), and it calculates the error (line 9). If the error is greater than 50%, the algorithm executes the set of lines from 5 to 7 and executes the "weak" learner again. If the error is equal to zero, the algorithm sets β_t equal to 10^{-10}, thereby making the weight very small, and executes lines 5 to 7 again. If the error is smaller than 50%, the algorithm makes β_t equal to the regular AdaBoost weight, and recalculates the weight for each instance by checking if the hypothesis is equal to the real label and the weight is equal to $2\varepsilon_t$. Otherwise the weight equals $2(1 - \varepsilon_t)$.

According to Webb, the main advantage is that Wagging reduces variance, and AdaBoost reduces both variance and bias [1]. In his paper [1], Webb argues that Wagging uses the continuous Poisson distribution, which is not as effective at variance reduction as Bagging. He also maintains that MultiBoost achieves almost the same bias reduction as AdaBoost, and most of the variance reduction of Wagging.

A final comment about MultiBoost: Basically, the algorithm tries to change the weight calculation by applying Equation 2.19 at every interval defined by variable k, and resamples the training data with the step used in Bagging (Wagging). In other cases, the algorithm updates β_t with the regular AdaBoost weight, and updates the weights of the instances with $2\varepsilon_t$ when the hypothesis is equal to the real label. Otherwise, it updates the value to $2(1 - \varepsilon_t)$.

2.3.6 SMOTEBoost

The SMOTEBoost [7] algorithm uses a different approach to improve AdaBoost. It applies a resampling step to change the training distribution, and adds new instances using the SMOTE algorithm. The following sections discuss how resampling is used in AdaBoost, introduces the SMOTE algorithm, and explains how SMOTEBoost integrates these approaches.

Different Approaches to Resampling and Boosting

Elkan [11] described and proved a theorem for the two-class case that shows how to change the proportion of negative examples in a training set to make better classification decisions using a classifier learned by a standard learning method. This theorem provides a motivation to use the resampling strategy to improve classifier results, because it relaxes the traditional machine learning rule that a learning algorithm must use the same population to train and test its hypothesis.

By its nature, AdaBoost uses reweighting strategy to change the distribution of weights in the training data, but it does not change the training data distribution itself. In resampling associated with AdaBoost, the weights are used to change the *entire* training data, because each iteration removes instances with very “low” weights, and replaces them with instances with “higher” weights. This is a direct application of under-sampling and over-sampling. Another strategy is the SMOTE algorithm, as explained in the next section.

Synthetic Minority Oversampling Technique

SMOTE (Synthetic Minority Oversampling Technique) [6] is used to execute over-sampling of the minority class, by creating artificial data with similar distance in the feature space. The algorithm uses the K-nearest neighbors for each example, and the distance is calculated according to the smallest distance along the n-dimensional feature space. Figure 2.1 presents only two features, and the new data is created based on the Euclidean distance between two points. To create a synthetic sample, the algorithm randomly selects one of the K-nearest neighbors, multiplies the corresponding feature vector difference by a random number between [0, 1], and adds this vector to x_i:

$$x_{new} = x_i + \delta(\hat{x_i} - x_i) \tag{2.20}$$

where x_i is a point belonging to the minority class, $\hat{x_i}$ also belongs to the minority class (and is selected using the K-nearest algorithm) and δ is a random value defined between 0 and 1. Thus, according to Equation 2.20 the algorithm will create a new instance, based on the difference from points in the feature space.

SMOTE has the over-generalization problem [27], because it generates new data without considering if a certain point in the feature space is in the middle of a set of points of the majority class, thereby likely making this point a noise in the data set, which could increase the overlapping between classes [27]. To avoid this, some algorithms (e.g. Borderline SMOTE and Adaptive Synthetic Sampling (ADASYN) [27]) were developed to try to remove data from the minority class that is overlapping with the majority class. The next section shows how SMOTEBoost uses this algorithm.

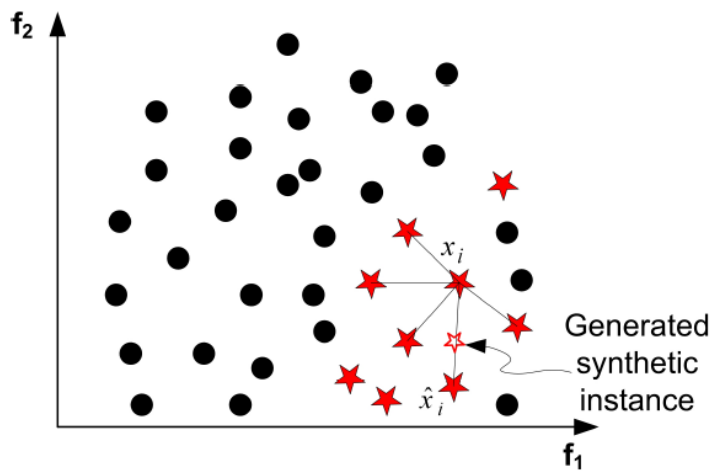


Figure 2.1: Data generation with SMOTE (Figure reproduced from [26])

SMOTEBoost Approach

SMOTEBoost was developed to work with the detection of the minority class in an unbalanced data set. For a two-class problem specifically, SMOTEBoost attempts to increase detection of the positive class by sacrificing the negative class[1].

With respect to the problem of unbalanced data sets, AdaBoost assigns a standard weight to all misclassified instances, and it has a natural tendency to become biased toward the majority class, even though it reduces the variance and bias. In this context, the main idea of SMOTEBoost as described in [7] is:

> "... to better model the minority class in the data set, by providing the learner not only with the minority class instances that were misclassified in previous boosting iterations, but also with a broader representation of those instances, and with minimal accuracy of the majority class."

[1] Assuming that the data set is highly unbalanced (e.g. 98% for the negative class, and the remaining data for the positive class)

SMOTEBoost tries to reduce the bias by changing the distribution of skewed instances, and increasing the weights for the skewed class. To achieve this, the SMOTE technique is applied in each iteration by increasing the number of skewed class instances, and then the weights (D_t) are increased in each boosting round. Chawla *et al* suggest that this approach increases the diversity among the classifiers in the ensemble [7]. SMOTEBoost is a modification of AdaBoost.M2 (Section 2.3.1).

Chawla *et al* demonstrated that SMOTEBoost provides good results, and improves the accuracy in skewed classes. However, they did not describe how to initialize the weights for the new instances created by the SMOTE process, which causes a problem with implementing the algorithm. Other authors, as in [39], built their own proposal to calculate the weights, but that is not a standard approach to solving the problem.

2.3.7 RareBoost

In [30], Joshi *et al* proposed RareBoost, another algorithm that improves the capability of AdaBoost to manage unbalanced data sets. SMOTEBoost, as previously shown, relies on the fact that new data from skewed classes is generated and aggregated in the training data at each iteration. RareBoost modifies AdaBoost.M1, by considering the relationship between the majority and minority classes. It focuses on the weight update process, and the authors created three different categories to evaluate the weak learners [30]: i) those that are able to assign true or false for all labels, ii) those that do not make any decisions on some examples, and iii) those that take classification costs into account. The focus of [30] is the first category: algorithms that assign true or false for all labels.

Joshi *et al* proposed the following modification. The first step is to compute the negative and positive predictions separately, by calculating the weights for positive instances:

$$TP_t = \sum_{i:f_t(x_i)\geq 0; y_i>0} D_t(x_i)f_t(x_i) \text{ and } FP_t = \sum_{i:f_t(x_i)\geq 0; y_i<0} D_t(x_i)f_t(x_i)$$

and for negative instances:

$$TN_t = \sum_{i:f_t(x_i)<0; y_i<0} D_t(x_i)f_t(x_i) \text{ and } FN_t = \sum_{i:f_t(x_i)<0; y_i>0} D_t(x_i)f_t(x_i)$$

These formulas must modify line 5 in AdaBoost.M1 (see Table 2.1). After these steps, it is still necessary to update β (line 7 in Table 2.1), but the main difference is that there are two betas, one for the true positive (β_t^+) and the other for false negative (β_t^-), as follows:

$$\beta_t^+ = \frac{1}{2}\log\frac{TP_t}{FP_t} \text{ and } \beta_t^- = \frac{1}{2}\log\frac{TN_t}{FN_t}$$

and as final modification the new output of this algorithm will be:

$$H_{final} = sign(\sum_{t:h_t\geq 0}(\beta_t^+ f_t(x_i)) + \sum_{t:h_t<0}(\beta_t^- f_t(x_i))) \tag{2.21}$$

2.3.8 2-AdaBoost

In [21], Golestani *et al* tried to improve AdaBoost using an unusual approach. Initially, they tested various classifiers, and calculated a diversity measure among these classifiers. Based on this information, the algorithm chooses two classifiers that have a maximum diversity, and they use these "weak" learners in *two* AdaBoost algorithms built in parallel. As output, 2-AdaBoost simply combine the results of these two AdaBoost classifiers, thus, in this approach, they tried to combine two learners, instead of one "weak" learner.

If a single classifier did not make errors there would be no need for an ensemble method, but in practice a single classifier does make errors. However, if a single classifier were combined with another that makes errors in different instances, a better final classifier could be created based on the combination [32].

In the Golestani *et al* AdaBoost experiments, the single weak learner creates coincident errors. This inspired the idea of combining different classifiers in AdaBoost, and evaluating their performance based on diversity measure.

This work produced significant improvements, but the authors did not report *how* to generate the final combiner, which makes the implementation of the algorithm very complex and open to interpretation. Another limitation is the fact that the algorithm combines only two classifiers.

Input: S, a sequence of examples m $\langle(x_1, y_1), ..., (x_m, y_m)\rangle$ with labels $y_i \in Y = \{1, ..., k\}$
the WeakLearner algorithm
integer T specifying number of iterations
vectors of integers I_i specifying the iteration at which
each subcommittee $i \geq 1$ should terminate.

1.**Initialize** $S' = S$ with instance weight equal to 1
2.**Initialize** set $k = 1$
3.**Do for** t=1...T
4. if $I_k = t$ then
5. reset S' to random weights drawn from the continuous Poisson distribution(Equation 2.19).
6. normalize S' to sum 1.
7. increment k.
8. $C_t = \text{WeakLearn}(S')$
9. Calculate the error of $C_t : \varepsilon_t = \frac{\sum_{x_j \in S' : C_t(x_j) \neq y_i} weight(x_j)}{m}$
10. If $\varepsilon_t > \frac{1}{2}$, then
11. reset S' to random weights drawn from the continuous Poisson distribution(Equation 2.19).
12. normalize S' to sum 1
13. increment k.
14. go to step 8.
15. Otherwise If $\varepsilon_t = 0$, then
16. set $\beta_t = 10^{-10}$
17. reset S' to random weights drawn from the continuous Poisson distribution(Equation 2.19).
18. normalize S' to sum 1.
19. increment k.
20. Otherwise
21. Set $\beta_t = \frac{\varepsilon_t}{1-\varepsilon_t}$
22. For each $x_j \in S'$
23. divide $weight(x_j)$ by $2\varepsilon_t$ if $C_t(x_j) \neq y_i$ and $2(1 - \varepsilon_t)$ otherwise
24. if $weight(x_j) < 10^{-8}$, set $weight(x_j) = 10^{-8}$
end_do
Output:$H_{final} = sign \sum_{t=1}^{T}(\beta_t f_t(x_i))$

Table 2.5: MultiBoosting Algorithm

Chapter 3

AdaBoost Dynamic

As presented in Chapter 2, AdaBoost has two main problems: i) dependency on the "weak" learner, and ii) dependency on the data set. Since it is very difficult to find a solution to the data dependency problem, it is crucial to find solutions that deal with dependency on the learner. Considering the algorithms used to address these problems discussed in Chapter 2, it is evident that most of them try to modify the weight calculation to achieve better accuracy. However, with the exception of 2-AdaBoost they do not address the problem of dependency on the "weak" learner, since they still allow only one classifier to be iteratively executed. In the case of 2-AdaBoost, the boosting procedure is executed for only two algorithms, and the final response is a combination of previous boosting executions, which is a way to address the problem of dependency on the learner. But there are also problems with implementation, because the 2-AdaBoost paper does not indicate how to choose the classifier in detail, and it is based on a limited combination of learners.

This work proposes using a different algorithm in each iteration, and reducing the dependency on one classifier. The first section will present a small modification to the AdaBoost.M1 algorithm, which will allow it to execute a different learner in each iteration; and this version is called AdaBoost Dynamic (AD). The weight update rule, which uses the exponential loss function, will remain unchanged, and thus the theoretical performance will be maintained. The second section will investigate the effect of reducing the error of the classifiers in each iteration, by searching for the classifier with the smallest error with the same weight calculation used in AdaBoost.M1. This modification is called AdaBoost Dynamic with Exponential Loss (AB-EL). While still considering the classifiers with low error, we also present a third algorithm, called AdaBoost Dynamic with Logistic Loss Function (AB-DL), in which the main modification of the weight calculation use the sigmoid function to update the weights. The weights will be calculated according to the derivative of the sigmoid function, rather than the logistic regression calculated with AdaBoost.M1. This modification is required because, when working with a learner that has an error rate close to zero, the logistic regression - used by AdaBoost.M1 - function does not perform well. The theoretical properties of AB-DL are also investigated.

Although AB-DL provides better results in practical evaluations, it is based on a search for the best classifier, and that can increase the training time. That is why this work proposes one further modification: combining the learner execution presented in AB-D with a small change in AB-DL's weight calculation; this is called AdaBoost Dynamic with Added Cost (AD-AC). As will be shown, AD-AC is upper bounded, and when applied to large data sets, it also helped reduce memory footprint.

3.1 AdaBoost Dynamic

AdaBoost attempts to improve the quality of a learner iteratively. Considering the weight distribution of the instances, it calculates a new weight by increasing it for instances where the learner committed an error by increasing it, while reducing the weights where the learner performed correctly. This approach has proven to be an efficient way to reduce errors in classification, but it could be improved by applying one weak learner in a specific iteration, and a different one in the next. This is because, depending on the type of data, it is possible that a certain distribution may offer better results with a different base learner.

AdaBoost Dynamic is presented in Table 3.5. It is the AdaBoost.M1 algorithm shown in Table 2.1, with an added task to consider a list of learners as input. Line 3 in the algorithm calls a different WeakLearner in each iteration. This information is represented by the input list W in the algorithm. In this case, the final output will be $H_{final} = \arg\max_{y \in Y} \sum_{t:W_j(x)=y} \log \frac{1}{\beta_t}$, which means that the new output hypothesis will be calculated considering that $W_j(x) \neq W_{j+1}(x)$. This algorithm will have the same error boundary as the original AdaBoost.M1, because the weight update has not changed.

The major difference of AdaBoost Dynamic is the input function, which now accepts a list of "weak" learners and automatically changes the executed learner in each step.

Experimental results (see Chapter 4) suggest that for a large majority of data sets, the performance of AdaBoost Dynamic is as effective as that of AdaBoost.M1, using the best

Input: sequence of examples m $\langle(x_1, y_1), ..., (x_m, y_m)\rangle$ with labels $y_i \in Y = \{1, ..., k\}$ list W of the WeakLearner algorithms integer T specifying number of iterations
1.**Initialize** $D_i = \frac{1}{m}$ for all i 2.**Do for** t=1...T 3. Call W[j], providing it with the distribution D_t 4. Get back hypothesis $W_j : X \to Y$ 5. Calculate the error of h_t : $\varepsilon_t = \sum_{i:W_j(x_i))\neq y_i} D_t(i)$ 6. If $\varepsilon_t > \frac{1}{2}$, then abort loop 7. Set $\beta_t = \frac{\varepsilon_t}{1-\varepsilon_t}$ 8. Update distribution D_t : $D_{t+1}(i) = \frac{D_t(i)}{Z_t} \times \begin{cases} \beta_t & \text{if } h_t(x_i) \neq y_i \\ 1 & \text{otherwise} \end{cases}$, where Z_t is a normalization constant. 9. If Length(W) = j then j = 1, else j = j+1 **Output:**$H_{final} = \arg\max_{y\in Y} \sum_{t:W_j(x)=y} \log \frac{1}{\beta_t}$.

Table 3.1: AdaBoost Dynamic Algorithm with the proposed modification

single “weak” learner. Therefore, AdaBoost Dynamic can be used as a default algorithm, and a benchmark for comparison with other “weak” learners. AdaBoost Dynamic will be most successful when an analyst is not sure about which base learner is the best to use with AdaBoost for a particular data set.

However, this raises a question: What if it is possible to choose the best classifier in each iteration, and execute it? The term ‘best’ means the classifier with the smallest error available in the list provided by a user. This question highlights the requirement for another modification to the algorithm, which is presented in the next section.

3.2 AdaBoost Dynamic with Search for the Best Classifier

Using the same weight calculation presented in AdaBoost.M1 (Table 2.1), the AdaBoost Dynamic with Exponential Loss (AB-EL) algorithm in Table 3.5 searches for the classifier with the smallest error in the list. One important difference between AB-EL and

Input: sequence of examples m $\langle(x_1, y_1), ..., (x_m, y_m)\rangle$ with labels $y_i \in Y = \{1, ..., k\}$
list W of the WeakLearner algorithms
integer T specifying number of iterations

1.**Initialize** $D_i = \frac{1}{m}$ for all i
2.**Initialize** $weight_i = 0$ for all i
3.**Initialize** $\epsilon_i = 0$ for all i
4.**Initialize** index to keep track of the best classifier $class_t = 0$, for all T
5.**Do for** t=1...T
6. **Do for** j=1...Size(W)
7. Call W[j], providing it with the distribution D_t
8. Get back hypothesis $W_j : X \to Y$
9. Calculate the error of $h_t : \varepsilon_j = \sum_{i:W_j(x_i) \neq y_i} D_t(i)$
10. If $\varepsilon_j < \frac{1}{2}$ and $\varepsilon_j \neq 0$
11. then if $\epsilon_t = 0$ or $\varepsilon_j \leq \epsilon_t$
12. then $\epsilon_t = \varepsilon_j$ and and $class_t$ = j
end_do
13. Set $\beta_t = \frac{\epsilon_t}{1-\epsilon_t}$
14. Update distribution $D_t : D_{t+1}(i) = \frac{D_t(i)}{Z_t} \times \begin{cases} \beta_t & \text{if } h_t(x_i) \neq y_i \\ 1 & \text{otherwise} \end{cases}$,
where Z_t is a normalization constant.
end_do Output: $H_{final} = \arg\max_{y \in Y} \sum_{t:W_{class_t}(x)=y} \log \frac{1}{\beta_t}$

Table 3.2: AdaBoost Dynamic with Exponential Loss (AB-EL) algorithm with the search for the "best" weak classifier

AdaBoost.M1, is that AB-EL considers the possibility of a learner returning zero error. In this case, the algorithm disregards this learner and continues the process with another. This decision is made in order to avoid overfitting, when a weak learner returns a zero error rate and stops execution of the algorithm. The objective is to execute the T iterations defined by the user.

The main effect of this search is that the errors of the classifiers in each iteration are close to zero, which makes them strong classifiers. AdaBoost.M1 will not improve the final hypothesis if the weak learner is actually a strong learner. This was also observed by [35]. Therefore, AB-EL will not improve the final hypothesis.

This behaviour suggests another important question: Is there a loss function that can provide better results, considering strong classifiers? To answer this, it is necessary to compare the available loss functions , and adapt them to the presented algorithm.

In order to identify other types of cost functions that could improve the accuracy of the final hypothesis, Figure 3.1 shows a performance comparison of the exponential, Binomial deviance, squared error and Truncated Quadratic (TQ) functions, and Table 3.3 shows the formulas that generated the graph. The first observation from Figure 3.1 suggests that caution must be taken when working with the TQ and squared error functions, because they are not good candidates for the loss function. The reason for this is that for margins $yf(x) > 1$, TQ and squared error increase the error on instances that are correctly classified [25]. Note that in real implementations, TQ and squared error can be truncated [25], which gives the algorithm performance close to that of Binomial deviance [25].

The second observation is that Binomial deviance has less influence on negative observations than Exponential function. According to [25], "...it is therefore far more robust in noisy settings where the Bayes error rate is not close to zero, and especially in situations where there is misspecification of the class labels in the training data". Also, Hastie [25] observed that Binomial deviance empirically showed improvements when applied to AdaBoost, because of the reasons presented.

	Formula
Exponential	$e^{-yf(x)}$
Binomial deviance	$\ln(1 + e^{-2yf(x)})$
Squared error	$(y - f(x))^2$
Truncated quadratic (TQ)	$(\max(0, 1 - yf(x)))^2$

Table 3.3: List of loss functions

3.3 AdaBoost Dynamic with Logistic Loss Function

In light of the lower performances of AB-EL, which will be presented in Chapter 4, and considering that there are different weight calculations that can be explored, this work presents a modification in the AB-EL algorithm, to improve its performance with strong learners.

Originally, Freund and Schapire proposed using the exponential loss function, but [8] also has shown that the sigmoid function can be used to estimate the probability of a

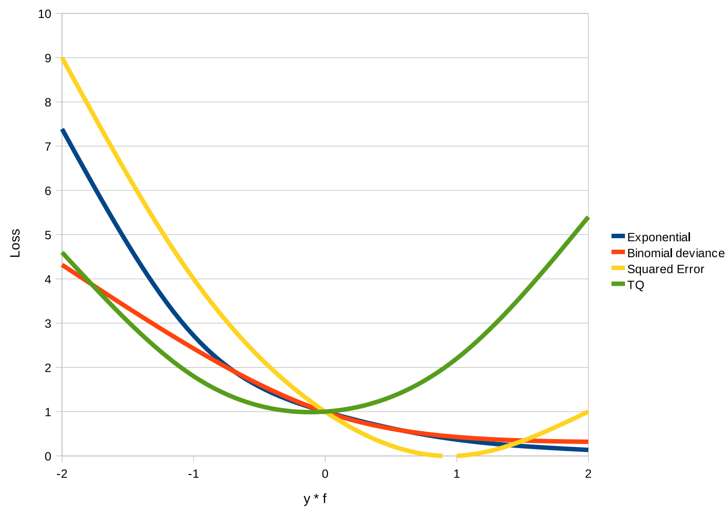


Figure 3.1: : Loss functions for two-class classification. The response is $y = \pm 1$ the prediction is f , with class prediction sign(f). Each function has been scaled so that it passes through point (0, 1). The functions are the same as those presented in Table 3.3.

certain instance belonging to class 1:

$$\widehat{\Pr}[y = 1|x] = \frac{1}{1 + e^{[-f_\beta(x_i)]}} \tag{3.1}$$

where $f_\beta(x_i) = \sum_t \beta_t h_t(x_i)$, such that $f_\beta(x_i)$ is a "good approximation" of y_i [8], then β_t represents the weight given to a specific classifier in an iteration t, and $h_t(x_i)$ is the "weak" learner hypothesis. The authors of [8] did not present a solution to predict β_t, so this work will assume the β_t is defined as follows:

$$\beta_t = \epsilon_t(1 - \epsilon_t) \tag{3.2}$$

where ϵ_t is the error rate of each classifier $h_t(\mathbf{x})$ in each iteration. The second assumption is that Equation 3.2 will be used to update the weights of the misclassified instances. In this work, it will be assumed that the weight update used by AB-DL is defined as $\frac{1}{1+e^{-\beta_t}}$.

In [37], Rudin *et al* state that the minimization of the exponential loss function does not necessarily make the algorithm achieve high margins, and since it uses these margins to calculate the class of an instance, this increases the chance of the algorithm misclassifying some instances. In addition, as presented in [33], AdaBoost is a gradient descent algorithm, which makes it practical to use the derivative of the sigmoid function to guide the search.

Since AB-EL accuracy results are poor in comparison to AdaBoot.M1 performance[1], and both use β_t equal to $\ln \frac{1-\epsilon_t}{\epsilon_t}$, this update step was modified to improve the final hypothesis when working with strong learners.

The weight update used by this proposal will be

$$D_{t+1}(i) = \frac{D_t(i)}{Z_t} \times \begin{cases} \frac{1}{1+e^{-\beta_t}} & \text{if } h_t(x_i) \neq y_i \\ 1 & \text{otherwise} \end{cases} , \tag{3.3}$$

where Z_t is a normalization constant, and $D_{t+1}(i)$ are the weights calculated for each instance. And,

$$H_{final} = \arg\max_{y \in Y} \sum_{t:h_t(x)=y} \beta_t \tag{3.4}$$

which is how to choose the final prediction of the ensemble. This output is a weighted vote scheme that is the same as the AdaBoost.M1 scheme.

The algorithm in Table 3.4 presents AdaBoost Dynamic with Logistic Loss (AB-DL), which uses the weight calculation above. The first step is to initialize all the m instances' weights to $\frac{1}{m}$. The second step initializes all the error rates (ϵ_t) and a variable to keep the classifier with smallest error, then the algorithm runs the iteration from one until T. In the first iteration, the search for the best "weak" learner begins using the classifier list provided by the user. This search executes each learner, captures the hypothesis and calculates the learner's error rate, then tests if the error rate is the smallest in the list. The next step calculates β_t, taking into consideration the smallest error in the list, and updates all instances according to Equation (3.3).

Since the main objective is to reduce the overfitting when AdaBoost deals with strong learners, AB-DL does not stop its execution when the error rate is close to zero, or when it is bigger than $\frac{1}{2}$. AB-DL will continue searching for another classifier until the number of iterations reaches the value of T.

[1]These results will appear in Chapter 4 .

Input:
sequence of examples m $\langle(x_1, y_1), ..., (x_m, y_m)\rangle$ with labels $y_i \in Y = \{1, ..., k\}$
list W of the WeakLearner algorithms
integer T specifying number of iterations
1.**Initialize** $D_i = \frac{1}{m}$ for all i
2.**Initialize** $\epsilon_t = 0$ for all T
3.**Do for** t=1...T
4. **Do for** j=1...Size(W)
5. Call W[j], providing it with the distribution D_t
6. Get back hypothesis $W_j : X \rightarrow Y$
7. Calculate the error of $W_j : \varepsilon_i = \sum_{i:W_j(x_i) \neq y_i} D_t(i)$
8. If $\varepsilon_t < \frac{1}{2}$ and $\varepsilon_t \neq 0$, then
9. If $\epsilon_t = 0$ or $\varepsilon_t \leq \epsilon_t$ then
10. $\epsilon_t = 0$ and $class = j$
end_if
end_if
end_do
10. Set $\beta_t = \epsilon_t(1 - \epsilon_t)$
11. Update distribution $D_t : D_{t+1}(i) = \frac{D_t(i)}{Z_t} \times \begin{cases} \frac{1}{1+e^{-\beta_t}} & \text{if } h_t(x_i) \neq y_i \\ 1 & \text{otherwise} \end{cases}$,
where Z_t is a normalization constant.
end_do
Output:$H_{final} = \arg\max_{y \in Y} \sum_{t:W_j^t(x))=y} \beta_t$

Table 3.4: AB-DL with the search for the "best" weak classifier, considering changes in weight calculation

Another issue is the fact that, compared to AdaBoost.M1, AB-DL weight calculation is not optimal, because its output is based on the derivative of the sigmoid function, rather than the logistic regression.

The new weight calculation implies that some theoretical guarantees must also hold, therefore we present Theorem 3.1, which shows that the value of Z_t will be bounded. This theorem is important for AB-DL, because it means that the sum of the weights of misclassified instances and correctly classified ones will be limited for all iterations.

Theorem 3.1 *Let h_t, where $t \in [1, T]$ is an arbitrary sequence of classifiers. Let $I =$*

$\{x_i, y_i\}_{i=1}^{m}$ be a sequence of training instances of m observations. Let us recursively define $D_{t+1}(i) = D_{t+1}(x_i) = D_t(x_i)\left(\frac{1}{1+e^{-\beta_t}}\right)^{|h_t(x_i)-y_i|}$ (as presented in Table 3.4), where $D_{t+1}(i)$ is the normalized weight distribution , $\beta_j = (1-\epsilon_j)\epsilon_j$, and $|h_t(x_i) - y_i| = 1$ when $h_t(x_i) \neq y_i$ and $|h_t(x_i) - y_i| = 0$ when $h_t(x_i) = y_i$. Then Z_t that is the normalization used in Table 3.4 is defined as:

$$\begin{aligned}
Z_t &= \gamma_1 + \gamma_2 = \frac{(1-\epsilon_t)\left(e^{\epsilon_t{}^2} + e^{\epsilon_t}\right) + \epsilon_t\, e^{\epsilon_t}}{e^{\epsilon_t{}^2} + e^{\epsilon_t}} \text{ , where} \\
\gamma_1 &= \sum_{i:h_t(x_i)\neq y_i} D_{t-1}(x_i)\left(\frac{1}{1+e^{-((1-\epsilon_t)\epsilon_t)}}\right)^{|h_t(x_i)-y_i|} \\
\gamma_2 &= \sum_{i:h_t(x_i)=y_i} D_{t-1}(x_i)
\end{aligned}$$

Proof Z_t is a normalization factor that must sum the weights of both misclassified and correctly classified weights $D_{t-1}(i)$ for all instances. Thus, AB-DL it will have the form

$$\begin{aligned}
Z_t &= \gamma_1 + \gamma_2 \text{ , where} \\
\gamma_1 &= \sum_{i:h_t(x_i)\neq y_i} D_{t-1}(x_i)\left(\frac{1}{1+e^{-((1-\epsilon_t)\epsilon_t)}}\right)^{|h_t(x_i)-y_i|} \\
\gamma_2 &= \sum_{i:h_t(x_i)=y_i} D_{t-1}(x_i)
\end{aligned}$$

Assuming that $|h_t(x_i)-y_i| = 1$ when $h_t(x_i) \neq y_i$ and $|h_t(x_i)-y_i| = 0$ when $h_t(x_i) = y_i$, and factoring $\frac{1}{1+e^{-(1-\epsilon_t)\epsilon_t}}$ from γ_1, we have

$$\begin{aligned}
Z_t &= \gamma_1 + \gamma_2 \text{ , where} \\
\gamma_1 &= \frac{1}{1+e^{-(1-\epsilon_t)\epsilon_t}} \sum_{i:h_t(x_i)\neq y_i} D_{t-1}(x_i) && (3.5) \\
\gamma_2 &= \sum_{i:h_t(x_i)=y_i} D_{t-1}(x_i) && (3.6)
\end{aligned}$$

Applying algebra to the factor, gives us [2]

$$\frac{1}{e^{-(1-\epsilon_t)\,\epsilon_t} + 1} = \frac{e^{\epsilon_t}}{e^{\epsilon_t^2} + e^{\epsilon_t}} \tag{3.7}$$

[2]The proof of equality of Equation 3.7 is presented in Appendix A.

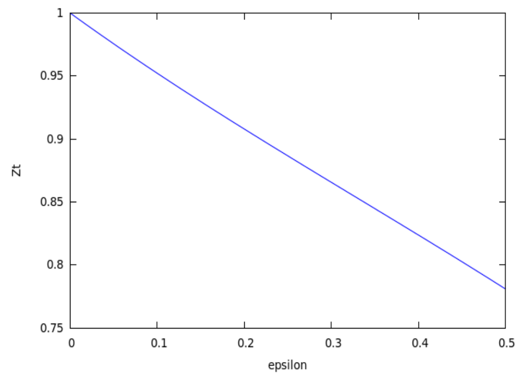


Figure 3.2: Graph shows the theoretical total sum of Z_t in relation to the error rate, *epsilon* (ϵ_t), for each iteration of AB-DL algorithm. It is important to notice that Z_t has a close to linear performance only when $\epsilon_t \in [0, \frac{1}{2}]$.

and making the substitutions in (γ_1), we find

$$Z_t \quad = \frac{e^{\epsilon_t}}{e^{\epsilon_t^2} + e^{\epsilon_t}} \sum_{i:h_t(x_i) \neq y_i} D_{t-1}(i) + \sum_{i:h_t(x_i) = y_i} D_{t-1}(i) \tag{3.8}$$

In (3.8), one last modification can be executed since, as presented [47, 25], $\sum_{i:h_t(x_i) \neq y_i} D_{t-1}(i) = \epsilon_t$ because it represents the error committed by the learner for some instances, and $\sum_{i:h_t(x_i) = y_i} D_{t-1}(i) = 1 - \epsilon_t$ represents the set of instances that do not have their weights changed in relation to the previous iteration. Plugging this in γ_1 and γ_2 results in

$$\begin{aligned} Z_t \quad &= \frac{e^{\epsilon_t}}{e^{\epsilon_t^2} + e^{\epsilon_t}} \epsilon_t + (1 - \epsilon_t) \\ Z_t \quad &= \frac{(1 - \epsilon_t) \left(e^{\epsilon_t^2} + e^{\epsilon_t} \right) + \epsilon_t \, e^{\epsilon_t}}{e^{\epsilon_t^2} + e^{\epsilon_t}} \square \end{aligned} \tag{3.9}$$

Figure 3.2 shows how Z_t behaves according to Theorem 3.1, and the performance of AB-DL is dependent on the learner's performance. As the learner error rate (ϵ_t) gets closer to zero, Z_t will get closer to one. When the error rate approximates $\frac{1}{2}$, the Z_t value is reduced. The important message property is that $Z_t \leq 1$.

As presented in Theorem 3.1, AB-DL combines weight reduction and the use of strong learners, which means that the algorithm will gradually reduce the weights of the misclassified instances, and increase the weights of correctly classified instances. This is useful, because AdaBoost.M1 with strong learners gives too much importance to misclassifications, which decreases the performance of the final model. On the other hand, AB-DL tries to assign importance to correct classifications, in order to build a better final model. This makes unsuitable for imbalanced data sets, because there would be a bias toward the majority class, and the algorithm will tend to remove instances from less representative labels.

Conjecture 3.1 hypothesizes that the provided new weight calculation has a theoretical error bound. However, it is not a general proof, because the weights depend on the number of iterations defined by user and the error rate of the classifier.

Conjecture 3.1 *Suppose the weak learning algorithm WeakLearn, when called by AB-DL, generates hypotheses with errors $\epsilon_1, \epsilon_2, ..., \epsilon_T$. Assuming that $y_i \in [0,1]$, $h_t \in [0,1]$ and $T \geq k$, where $k > 0$ is an integer that defines a minimum number of iterations for AB-DL, then the error $\varepsilon \leq \Pr_{i \sim D}[h_t(x_i) \neq y_i]$ of the final hypothesis h_t output by AB-DL is bounded by*

$$\varepsilon \leq \frac{(1-\epsilon_t)\left(e^{\epsilon_t{}^2} + e^{\epsilon_t}\right) + \epsilon_t\, e^{\epsilon_t}}{(e^{\epsilon_t{}^2} + e^{\epsilon_t}) \sum_{t=1}^{T}((1-\epsilon_t)\epsilon_t)}, \text{ where } T \geq k \tag{3.10}$$

where k represents the minimum number of iterations necessary to guarantee that $\varepsilon \in [0,1]$.

Proof The proof follows the same steps as the original AdaBoost.M1 proof (Theorem 6 proof) in [18]. As initial assumptions, the update rule in AB-DL is given by $w_i^{t+1} = \frac{w_i^t}{Z_t}(\frac{1}{1+e^{-\beta_t}})^{|h_t(x_i)-y_i|} = D_t(i)(\frac{1}{1+e^{-\beta_t}})^{|h_t(x_i)-y_i|}$, as presented in Table 3.4, where $D(i)$ is a normalized weight distribution. Since Z_t is a normalization constant, all weights will be in the [0,1] interval, and using the result from Theorem 3.1, the sum of weights (w_i^{t+1}) of all instances is defined as

$$\begin{aligned} \sum_{i=1}^{m} w_i^{t+1} &= \sum_{i=1}^{m} \frac{w_i^t}{Z_t}(\frac{1}{1+e^{-\beta_t}})^{|h_t(x_i)-y_i|} \\ &= \frac{(1-\epsilon_t)\left(e^{\epsilon_t{}^2} + e^{\epsilon_t}\right) + \epsilon_t\, e^{\epsilon_t}}{e^{\epsilon_t{}^2} + e^{\epsilon_t}} \end{aligned} \tag{3.11}$$

The final weight of any instance i can also be written as

$$w_i^{T+1} = D(i) \prod_{t=1}^{T} (\frac{1}{1+e^{-(1-\epsilon_t)\epsilon_t)}})^{|h_t(x_i)-y_i|} \tag{3.12}$$

The final hypothesis H_{final} in Table 3.4 is written as

$$H_{final} = \sum_{t=1}^{T} \beta_t = \sum_{t=1}^{T}((1-\epsilon_t)\epsilon_t) \tag{3.13}$$

Then, with the H_{final} expressed as in (3.13), it is possible to bound it as follows

$$\sum_{t=1}^{T}((1-\epsilon_t)\epsilon_t) > \prod_{t=1}^{T}(\frac{1}{1+e^{-(1-\epsilon_t)\epsilon_t}})^{|h_t(x_i)-y_i|} \tag{3.14}$$

This inequality holds[3], because every time there is a misclassification, $|h_t(x_i) - y_i| = 1$, so the right hand side will be the regular sigmoid function.

Making the same substitutions as presented in [18], it follows that

$$\sum_{i=1}^{m} w_i^{T+1} \quad \geq \sum_{i:h_t(x_i)\neq y_i} w_i^{T+1}$$

and substituting the right hand side with (3.12), we have

$$\sum_{i=1}^{m} w_i^{T+1} \quad \geq \sum_{i:h_t(x_i)\neq y_i} D(i) \prod_{t=1}^{T}(\frac{1}{1+e^{-\beta_t}})^{|h_t(x_i)-y_i|} \tag{3.15}$$

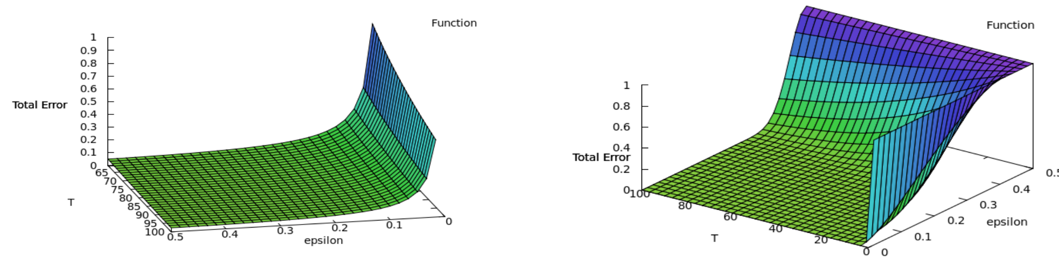
Since the right hand side of (3.15) is smaller than the inequality left hand side in (3.14), we can reach

$$\begin{aligned} \sum_{i=1}^{m} w_i^{T+1} \quad &\geq \sum_{i:h_t(x_i)\neq y_i} D(i) \sum_{t=1}^{T}((1-\epsilon_t)\epsilon_t) \\ \sum_{i=1}^{m} w_i^{T+1} \quad &\geq (\sum_{i:h_t(x_i)\neq y_i} D(i)) \sum_{t=1}^{T}((1-\epsilon_t)\epsilon_t) \end{aligned} \tag{3.16}$$

Considering that $\sum_{i:h_t(x_i)\neq y_i} D(i) = \varepsilon$ [18], because the whole sum is done only with the misclassified instances, therefore that is the total error committed by the classifier. Since the left hand side of (3.16) is $\sum_{i=1}^{m} w_i^{T+1} = \frac{(1-\epsilon_t)\left(e^{\epsilon_t{}^2}+e^{\epsilon_t}\right)+\epsilon_t\, e^{\epsilon_t}}{e^{\epsilon_t{}^2}+e^{\epsilon_t}}$, and combining with (3.11), we get

$$\begin{aligned} \varepsilon \sum_{t=1}^{T}((1-\epsilon_t)\epsilon_t) &\leq \frac{(1-\epsilon_t)\left(e^{\epsilon_t{}^2}+e^{\epsilon_t}\right)+\epsilon_t\, e^{\epsilon_t}}{e^{\epsilon_t{}^2}+e^{\epsilon_t}} \\ \varepsilon &\leq \frac{(1-\epsilon_t)\left(e^{\epsilon_t{}^2}+e^{\epsilon_t}\right)+\epsilon_t\, e^{\epsilon_t}}{(e^{\epsilon_t{}^2}+e^{\epsilon_t})\sum_{t=1}^{T}((1-\epsilon_t)\epsilon_t)} \text{ where } T \geq k\Box \end{aligned} \tag{3.17}$$

[3]The proof of inequality 3.14 can be found in Appendix A



(a) (b)

Figure 3.3: Theoretical errors for AB-DL (a) and AdaBoost.M1 (b). The Total Error is calculated considering the number of iterations T, and the error rate (epsilon). The graphs assume that a certain learner is executed T times and it has the same error rate in all iterations.

In order to compare the findings in Conjecture 3.1 in relation to the AdaBoost.M1 error, it is necessary to assume that all the learners have the same error rate in each set of iterations. Then the error bound (3.17) is transformed into

$$\varepsilon \leq \frac{(1-\epsilon_t)\left(e^{{\epsilon_t}^2}+e^{\epsilon_t}\right)+\epsilon_t\, e^{\epsilon_t}}{\left(e^{{\epsilon_t}^2}+e^{\epsilon_t}\right)T((1-\epsilon_t)\epsilon_t)} \tag{3.18}$$

because the sum will be executed over the same set of values. Similarly, it is possible to reduce the AdaBoost.M1 error bound, presented in (2.11), to the form

$$\varepsilon_{AdaBoost.M1} \leq ((1-\epsilon_t)\ \epsilon_t)^{\frac{T}{2}}\ 2^T \tag{3.19}$$

Using these modified formulas, Figure 3.3 presents the comparison between AB-DL (a) and AdaBoost.M1 (b). The first observation is that the error rate of the weight update of AB-DL is smaller than the AdaBoost.M1 error rate, particularly considering the small values of ϵ_t and high number of iterations. A second observation is related to the location of the theoretical AB-DL improvement being in the interval where AdaBoost.M1 has its higher errors: between $0.5 < \epsilon_t < 0.4$. The main AB-DL disadvantage is that the user is required to define at least k iterations in order to guarantee the theoretical convergence. To plot the graph the value k must be equal to 61; if the user defines a smaller k, under the assumptions used to build the graph, the total error will be larger than 1, and thus it cannot be assumed to be a probability distribution.

Empirical results will demonstrate that AB-DL works with a small number of iterations, and that it will tend to improve the AdaBoost.M1 results, since it combines only strong learners, and AdaBoost.M1 overfits in the same situation.

3.4 AdaBoost Dynamic with Added Cost (AD-AC)

The main disadvantage of AB-DL is related to the impossibility of improving its performance, since the error rate of its learners is already close to zero. To add more diversity among the classifiers it is necessary to change the learner execution policy, with the consequence that the user may use classifiers with poor performance with others that have high accuracy. AB-DL weight calculation is not suitable for this configuration, because it was projected to deal with classifiers with high accuracy. Another important issue is related to the minimum number of iterations k, which AB-DL requires to theoretically guarantee the error rate.

The idea is to improve the AB-D algorithm, because it offers more diversity among the classifiers used in the list. Unlike AB-DL, which executes a search for the classifier with the smallest error rate in the list, AD-AC executes the algorithms in the sequence provided by the user, which does not guarantee they will have error rate close to zero.

In order to aggregate the "weak" learner's confidence in its prediction for the training data, AdaBoost.M2 [18] was modified through the addition of a learning rate factor (α). Another modification to AB-DL is related to the final output calculation, which uses the derivative of the sigmoid function. This value was changed to $\beta_t = 1-(\varepsilon_t(1-\varepsilon_t))$, where ε_t is the error rate of the "weak" learner in each iteration t. One reason for this change was that the new β_t will give more importance to classifiers with error rates close to zero. A second reason is empirical, in that tests showed that this formula combined with the learner's confidence offered the best results, with a small improvement in relation to AB-DL.

Table 3.5 presents the AdaBoost Dynamic with Added Cost (AD-AC). The inputs are the same as those for AB-DL with exception of the α value, which is defined by the user. Like AdaBoost.M1, AD-AC also initiates the weights for all the training instances with uniform distribution, then initiates the loop to execute the learners. In each iteration the algorithm chooses a different learner from the list, executes it with the weights given to the training instances, then calculates the error and checks if the error rate is higher

Input: sequence of examples m $\langle(x_1,y_1),...,(x_m,y_m)\rangle$ with labels $y_i \in Y = \{1,...,k\}$ list W of the WeakLearner algorithms integer T specifying number of iterations double α that represents the learning rate.
1.**Initialize** $D_i = \frac{1}{m}$ for all i 2.**Do for** t=1...T 3. Call W_t, providing it with the distribution D_t 4. Get back hypothesis $W_t : X \rightarrow Y$ 5. Calculate the error of $W_t : \varepsilon_t = \sum_{i:W_t(x_i))\neq y} D_t(i)$ 6. If $\varepsilon_t > \frac{1}{2}$, then abort loop 7. Set $\beta_t = 1-(\varepsilon_t(1-\varepsilon_t))$ 8. Set $w_i = \beta_t^{\alpha((1-W_t(x_i,y_i))+W_t(x_i,y))}$ for all i, where $W_t(x_i) \neq y$ 9. Update distribution $D_t : D_{t+1}(i) = \frac{D_t(i)}{Z_t} \times \begin{cases} \frac{1}{1+e^{-w_i}} & \text{if } W_j(x_i) \neq y \\ 1 & \text{otherwise} \end{cases}$, where Z_t is a normalization constant. 10. If Length(W) = j then j = 1, else j = j+1 **Output:**$H_{final} = \arg\max_{y\in Y} \sum_{t:W_j(x)=y} \beta_t$.

Table 3.5: AdaBoost Dynamic with Added Cost (AD-AC) that aggregates the "weak" leaner's confidence in its weight calculation.

than $\frac{1}{2}$. It then calculates $\beta_t = 1-(\varepsilon_t(1-\varepsilon_t))$, and creates a variable w_i for all instances, which will calculate the weight of each misclassified instance, aggregating the response given by the "weak" classifier and the learning rate defined by the user. The learning rate must be $\alpha \in [0,1]$, as this will guarantee that $\beta_t^{\alpha((1-W_j(x_i,y_i))+W_j(x_i,y))}$ will decrease when the error rate increases. The function w_i is inspired by the same function used in AdaBoost.M2 [18]. The last two steps are the weight update, which uses the sigmoid function, and the final output that is equal to AB-DL. Theorem 3.2 shows that Z_t is also bounded, following the results of Theorem 3.1.

Theorem 3.2 *Let h_t, where $t \in [1,T]$, be an arbitrary sequence of "weak" classifiers. Let $I = \{x_i, y_i\}_{i=1}^m$ be a sequence of training instances of m observations. Let us recursively define $D_{t+1}(i) = D_{t+1}(x_i) = D_t(x_i)\left(\frac{1}{1+e^{-w_i}}\right)^{|h_t(x_i)-y_i|}$ (as presented in Table 3.5), where $D_{t+1}(i)$ is the normalized weight distribution, $w_i = \beta_t^{\alpha(1-h_t(x_i,y_i)+h_t(x_i,y))}$, $\beta_t = 1-((1-\epsilon_t)\epsilon_t)$, $\alpha \in [0,1]$, $\epsilon_t \in [0,\frac{1}{2}]$, with $|h_t(x_i)-y_i| = 1$ when $h_t(x_i) \neq y_i$ and $|h_t(x_i)-y_i| = 0$*

when $h_t(x_i) = y_i$, then

$$\begin{aligned} Z_t &= \gamma_1 + \gamma_2 = \frac{(1-\epsilon_t)\left(e^{\left(\epsilon_t^2-\epsilon_t+1\right)^\alpha}+1\right) + \epsilon_t\, e^{\left(\epsilon_t^2-\epsilon_t+1\right)^\alpha}}{e^{\left(\epsilon_t^2-\epsilon_t+1\right)^\alpha}+1}\text{ , } \textit{where} \qquad (3.20) \\ \gamma_1 &= \sum_{i:h_t(x_i)\neq y_i} D_t(x_i)\left(\frac{1}{1+e^{-(1-((1-\epsilon_t)\epsilon_t))^\alpha}}\right)^{|h_t(x_i)-y_i|} \\ \gamma_2 &= \sum_{i:h_t(x_i)=y_i} D_t(x_i) \end{aligned}$$

Proof Since $w_i = \beta_j^{(\alpha)(1-h_j(x_i,y_i)+h_j(x_i,y))} < 1$, $h_j(x_i,y_i) \geq h_j(x_i,y)$, both values are in the interval [0,1], and all values of $\alpha \in [0,1]$, we can assume that α is an upper value defined by the user. Then, in order to make the formula simpler, it is assumed that $w_i = \beta_j^{(\alpha)}$. This reduces the update rule to

$$\begin{aligned} D_{t+1}(i) &= D_{t+1}(x_i) = D_t(x_i)\left(\frac{1}{1+e^{-\beta_t^\alpha}}\right)^{|h_t(x_i)-y_i|} \\ &= D_{t+1}(x_i) = D_t(x_i)\left(\frac{e^{\beta_t^\alpha}}{1+e^{\beta_t^\alpha}}\right)^{|h_t(x_i)-y_i|} \end{aligned}$$

Again Z_t is a normalization factor, therefore we will have

$$\begin{aligned} Z_t &= \gamma_1 + \gamma_2 \text{ , where} \\ \gamma_1 &= \sum_{i:h_t(x_i)\neq y_i} D_{t-1}(x_i)\left(\frac{1}{1+e^{-(1-((1-\epsilon_t)\epsilon_t))^\alpha}}\right)^{|h_t(x_i)-y_i|} \\ \gamma_2 &= \sum_{i:h_t(x_i)=y_i} D_{t-1}(x_i) \end{aligned}$$

then, considering that $|h_t(x_i) - y_i| = 1$ when $h_t(x_i) \neq y_i$ and $|h_t(x_i) - y_i| = 0$ when $h_t(x_i) = y_i$, it is possible to factorize (γ_1) in the same form done in Theorem 1

$$\begin{aligned} Z_t &= \gamma_1 + \gamma_2 \text{ , where} \\ \gamma_1 &= \frac{1}{1+e^{-(1-((1-\epsilon_t)\epsilon_t))^\alpha}} \sum_{i:h_t(x_i)\neq y_i} D_{t-1}(x_i) \\ \gamma_2 &= \sum_{i:h_t(x_i)=y_i} D_{t-1}(x_i) \end{aligned}$$

Then, applying algebra to the factor, we can find

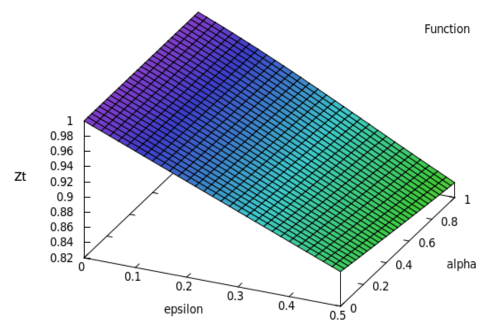


Figure 3.4: Graph shows the theoretical total sum of Z_t in relation to the error rate (ϵ) and the α for each iteration of AD-AC algorithm.

$$\frac{1}{1+e^{-(1-((1-\epsilon_t)\epsilon_t))^{\alpha}}} = \frac{e^{\left(\epsilon_t^2-\epsilon_t+1\right)^{\alpha}}}{e^{\left(\epsilon_t^2-\epsilon_t+1\right)^{\alpha}}+1} \tag{3.21}$$

and making the substitution in (γ_1), Z_t will have the form

$$Z_t = \frac{e^{\left(\epsilon_t^2-\epsilon_t+1\right)^{\alpha}}}{e^{\left(\epsilon_t^2-\epsilon_t+1\right)^{\alpha}}+1} \sum_{i:h_t(x_i)\neq y_i} D_{t-1}(x_i) + \sum_{i:h_t(x_i)=y_i} D_{t-1}(x_i) \tag{3.22}$$

Considering that, as presented in [47], $\sum_{i:h_t(x_i)\neq y_i} D_{t-1}(i) = \epsilon_t$, which is the misclassification error committed by the classifiers in each iteration, and $\sum_{i:h_t(x_i)=y_i} D_{t-1}(i) = 1-\epsilon_t$ is the probability of a correct classification (when $y_i = h_t(x_i)$), we have

$$\begin{aligned} Z_t &= \frac{e^{\left(\epsilon_t^2-\epsilon_t+1\right)^{\alpha}}}{e^{\left(\epsilon_t^2-\epsilon_t+1\right)^{\alpha}}+1}\epsilon_t + (1-\epsilon_t) \\ Z_t &= \frac{(1-\epsilon_t)\left(e^{\left(\epsilon_t^2-\epsilon_t+1\right)^{\alpha}}+1\right)+\epsilon_t\, e^{\left(\epsilon_t^2-\epsilon_t+1\right)^{\alpha}}}{e^{\left(\epsilon_t^2-\epsilon_t+1\right)^{\alpha}}+1}\square \end{aligned} \tag{3.23}$$

The relationship among Z_t, the error rate (ϵ_t) and the learning rate (α) is presented in Figure 3.4. The figure shows that α provides a small contribution to the weight calculation where the weight increases when $\alpha \to 0$. Another observation is that Z_t is also bounded, and its value is equal to 1 when the error rate gets close to zero, and it decreases when the error rate approximates to $\frac{1}{2}$; the same behaviour as AB-DL.

Theorem 3.3 shows the theoretical error in AD-AC. This theorem depends on the results of Theorem 3.2. The final error is important to establish the algorithm's theoretical performance.

Theorem 3.3 *Suppose the weak learning algorithm WeakLearn, when called by AD-AC, generates hypotheses with errors* $\epsilon_1, \epsilon_2, ..., \epsilon_T$ *and* $\epsilon_t \in [0, 0.5]$*, then the* $\varepsilon \geq \Pr_{i\sim D}[h_t(x_i) \neq y_i]$ *of the final hypothesis* h_t *output by AD-AC is bounded by*

$$\varepsilon \quad \leq \frac{(1-\epsilon_t)\left(e^{\left(\epsilon_t^2-\epsilon_t+1\right)^\alpha}+1\right)+\epsilon_t\, e^{\left(\epsilon_t^2-\epsilon_t+1\right)^\alpha}}{\left(e^{\left(\epsilon_t^2-\epsilon_t+1\right)^\alpha}+1\right)\left(\sum_{t=1}^{T}(1-(1-\epsilon_t)\epsilon_t)\right)} \tag{3.24}$$

Proof The proof follows the same steps as the original AdaBoost.M1 proof (Theorem 6 proof) in [18]. Considering that the weight values are normalized by a constant Z_t, then the AD-AC weight update rule is

$$w_i^{t+1} = \frac{w_i^t}{Z_t}\left(\frac{1}{1+e^{-\beta_t^{1-\alpha}}}\right)^{|h_t(x_i)-y_i|} = \sum_{i=1}^{m} D_t(i)\left(\frac{1}{1+e^{-\beta_t^{(1-\alpha)}}}\right)^{|h_t(x_i)-y_i|}$$

where $D(i)$ is a normalized weight distribution. Using the results from Theorem 3.2, the $\sum_{i=1}^{m} w_i^{t+1}$ is defined as

$$\begin{aligned}\sum_{i=1}^{m} w_i^{t+1} \quad &= \sum_{i=1}^{m} D_t(i)\left(\frac{1}{1+e^{-\beta_t^{(1-\alpha)}}}\right)^{|h_t(x_i)-y_i|} \\ &= \frac{(1-\epsilon_t)\left(e^{\left(\epsilon_t^2-\epsilon_t+1\right)^\alpha}+1\right)+\epsilon_t\, e^{\left(\epsilon_t^2-\epsilon_t+1\right)^\alpha}}{e^{\left(\epsilon_t^2-\epsilon_t+1\right)^\alpha}+1}\end{aligned} \tag{3.25}$$

The final hypothesis H_{final} in Table 3.5 can be written as

$$H_{final} = \sum_{t=1}^{T}(1-((1-\epsilon_t)\epsilon_t)) \tag{3.26}$$

then (3.26) bounds[4]

$$\sum_{t=1}^{T}(1-((1-\epsilon_t)\epsilon_t)) \quad > \prod_{t=1}^{T}\left(\frac{1}{1+e^{-((1-(1-\epsilon_t)\epsilon_t)^\alpha}}\right)^{|h_t(x_i)-y_i|} \tag{3.27}$$

[4]The proof of Inequality 3.27 can be found in Appendix A

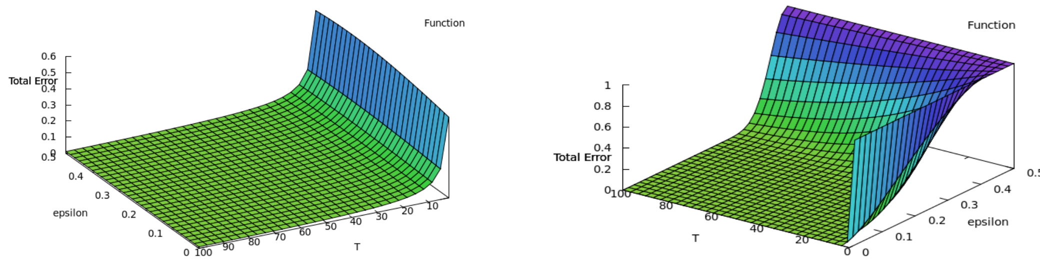
Making the same substitutions as presented in [18] and Conjecture 3.1, it follows that

$$\begin{aligned}
\sum_{i=1}^{m} w_i^{T+1} &\geq \sum_{i:h_t(x_i)\neq y_i} w_i^{T+1} \\
&\geq \sum_{i:h_t(x_i)\neq y_i} D(i) \prod_{t=1}^{T} \Big(\frac{e^{\beta_t^{\alpha}}}{1+e^{\beta_t^{\alpha}}}\Big)^{|h_t(x_i)-y_i|} \\
\sum_{i=1}^{m} w_i^{T+1} &\geq \Big(\sum_{i:h_t(x_i)\neq y_i} D(i)\Big) \sum_{t=1}^{T} (1-(1-\epsilon_t)\epsilon_t) \qquad (3.28)
\end{aligned}$$

Considering that $(\sum_{i:h_t(x_i)\neq y_i} D_t(i)) = \varepsilon$, because the whole sum is done only with the misclassified instances, therefore that is the total error committed by the classifier. Since $\sum_{i=1}^{m} w_i^{T+1} = \frac{(1-\epsilon_t)\left(e^{\left(\epsilon_t^2-\epsilon_t+1\right)^{\alpha}}+1\right)+\epsilon_t\, e^{\left(\epsilon_t^2-\epsilon_t+1\right)^{\alpha}}}{e^{\left(\epsilon_t^2-\epsilon_t+1\right)^{\alpha}}+1}$, by (3.25), and combining this with (3.28), one can find

$$\begin{aligned}
\varepsilon \sum_{t=1}^{T} (1-(1-\epsilon_t)\epsilon_t) &\leq \frac{(1-\epsilon_t)\left(e^{\left(\epsilon_t^2-\epsilon_t+1\right)^{\alpha}}+1\right)+\epsilon_t\, e^{\left(\epsilon_t^2-\epsilon_t+1\right)^{\alpha}}}{e^{\left(\epsilon_t^2-\epsilon_t+1\right)^{\alpha}}+1} \\
\varepsilon &\leq \frac{(1-\epsilon_t)\left(e^{\left(\epsilon_t^2-\epsilon_t+1\right)^{\alpha}}+1\right)+\epsilon_t\, e^{\left(\epsilon_t^2-\epsilon_t+1\right)^{\alpha}}}{\left(e^{\left(\epsilon_t^2-\epsilon_t+1\right)^{\alpha}}+1\right)\left(\sum_{t=1}^{T}(1-(1-\epsilon_t)\epsilon_t)\right)} \square \qquad (3.29)
\end{aligned}$$

The performance of the error bound found in Theorem 3.3 in relation to AdaBoost.M1 is now presented. Figure 3.5 (a) shows the error rate performance for AD-AC, and Figure 3.5 (b) shows the performance for AdaBoost.M1. The graphs were built considering that all the error rates are equal for all the iterations defined by user. This reduces the inequality (3.29) into $\varepsilon \leq \frac{(1-\epsilon_t)\left(e^{\left(\epsilon_t^2-\epsilon_t+1\right)^{\alpha}}+1\right)+\epsilon_t\, e^{\left(\epsilon_t^2-\epsilon_t+1\right)^{\alpha}}}{\left(e^{\left(\epsilon_t^2-\epsilon_t+1\right)^{\alpha}}+1\right)(T(1-(1-\epsilon_t)\epsilon_t))}$. The last parameter needed to build Figure 3.5 (a) is the α that was assumed equal to 0.35 (which is the same α used in the experiments). Figure 3.5 (b) was built in the same manner as Figure 3.3(b). The first observation is related to the error reduction of AD-AC compared to AdaBoost.M1; the maximum AD-AC error, presented in Figure Figure 3.5 (a), is 0.6, and the AdaBoost.M1 performance improved with a higher number of iterations. The second observation is that AD-AC also performs better than AdaBoost.M1 in the interval $0.5 < \epsilon_t < 0.4$ for all iterations, which is an improvement over AB-DL. The last observation is related to the fact that, in theory, AD-AC does not require a high number of iterations to reach better accuracy, as opposed to AdaBoost.M1, which does not perform well even with strong classifiers and a low number of iterations.



(a) (b)

Figure 3.5: Theoretical errors for AD-AC (a) and AdaBoost.M1 (b). The Total Error is calculated considering the number of iterations T, and the error rate (epsilon).

Chapter 5 will present the adaptation of AD-AC's weight calculation and learner execution to the problem of data streams. The theoretical results presented in this chapter will remain valid.

Chapter 4

Experimental Results

This chapter presents some results of applying the different versions of AdaBoost Dynamic discussed in Chapter 3 with UCI data sets [14]. Chapter 3 introduced four modifications to the AdaBoost.M1 algorithm. The first allows the use of a different learner in each iteration: this version is called AdaBoost Dynamic (AB-D). The second modification forces the algorithm to choose the "best" learner, while keeping the AdaBoost.M1 weight calculation; this version is known as AdaBoost Dynamic with Exponential function (AB-EL). The third modification changes the weight calculation by considering the sigmoid function, with the algorithm still searching for the "best" learner in each iteration; this last version is called AdaBoost Dynamic with Logistic function (AB-DL). The main AB-DL disadvantage is related to the reduction of diversity among classifiers, which negates its improvement. In order to solve this problem, AdaBoost Dynamic with Added Cost (AD-AC) was developed by combining the AB-D learner execution with AB-DL's weight calculation.

As an application of the AD-AC weight calculation and learner execution, this approach was adapted to work with a parallel boosting algorithm, known as OzaBoost. The new version is called OzaBoostDynamic (OzaDyn), and will be presented in Chapter 5 along with experiments. Results show that the new strategy is better than OzaBoost with respect to the model size and evaluation time, while maintaining the same accuracy as OzaBoost.

The first results presented are related to the AB-D algorithm with various AdaBoost.M1 algorithms, using different "weak" learners and with the "weak" learners in isolation. Then, AB-EL is compared with some of the AdaBoost.M1 algorithms, to show that it does overfit, which motivated its modification to AB-DL. The AB-DL and

AD-AC algorithms are compared using the same learners used as AB-D, and the rationale for using resampling as a technique to improve the final hypothesis is examined.

Name	# of Instances	# of Attributes	# of Classes	Missing Values	Balanced?
au1	1000	21	2	No	No
au1-balanced	1258	21	2	No	Yes
balance-scale	625	5	3	No	No
blood[48]	748	5	2	No	No
BreastTissue	105	10	6	No	No
car	1728	7	4	No	No
cmc	1473	10	3	No	Yes
crx	690	15	2	Yes	Yes
CTG	2128	36	10	Yes	No
glass	214	11	7	No	No
ionosphere	351	35	2	No	Yes
iris	150	4	3	No	No
labor	57	17	2	Yes	Yes
post-operative	90	9	3	Yes	No
segment	1499	19	7	No	No
soybean	682	35	19	No	No
wine	178	14	3	No	Yes
zoo	101	18	7	No	No

Table 4.1: Characteristics of the evaluated data sets from UCI.

4.1 Data Set Configuration

The AB-D, AB-EL, AB-DL and AD-AC algorithms were implemented in Weka[23]. All the AdaBoost Dynamic implementations used the following ten different “weak” learners:

- Neural Networks;
- Naive Bayes implementation;
- Decision Stumps;
- Bayes Networks;
- Random Tree;
- Random Forest;
- SVM (Support Vector Machine);
- Bagging;

- ZeroR rules; and,
- Naive Bayes Tree [31].

The AB-EL and AB-DL algorithms do not need to define the sequence of the "weak" learner execution, since they search for the learner with the smallest error rate for a particular weight or data distribution.

All the "weak" learner algorithms in the above list were used taking only their default Weka configuration into consideration, which makes the experiments repeatable with other data sets. The results were obtained with 10x10 fold cross validation, which was chosen because it ensures that the results are representative of what independent test sets would yield [44]. All runs included a pair-wise T-Test with a confidence interval of 95%.

Table 4.1 describes the 18 data sets chosen from the UCI database for this experiment. All the result tables use the following notation: if an algorithm is statistically better than the algorithm in the first column, the symbol $\circ$ is beside it, and if it is not better the symbol $\bullet$ shows. If there is no symbol, the difference in performance is not statistically significant. The main reason to use T-Test is to show results of the classifiers in relation to each data set. This allows a clear evaluation of classifier's performance. On the other hand, as it is well know that T-Test has limitations, as presented in [29], this work also presents the ANOVA test results.

4.2 AB-D Results

Table 4.2 compares the results of AB-D with some "weak" learners, using 100 iterations. The first observation is related to the average performance of AB-D compared to the other algorithms. The number of symbols indicating statistically significant better performance of AB-D show that it was clearly superior to Decision Stumps, Random Trees and Naive Bayes. Bayes Networks and Random Trees each had two cases with better performance than AB-D, but they also had five and four worse cases respectively. The only algorithm with comparable statistical performance was C4.5 with three wins and three losses.

Table 4.3 continues the comparison of AB-D with simple classifiers, without boosting algorithm to improve their results. The classifiers compared are Bagging, Neural Networks (NN), ZeroR and Naive Bayes Trees, and again the AB-D average was superior to the others. The first result shows that ZeroR had unacceptable performance compared to AB-D (15 losses and one win relative to AB-D). NBTree had two cases with statistical

Data set	AB-D	C4.5	Stumps	R. Forest	R. Tree	N. Bayes	Bayes Net.
au1	72.25± 3.79	77.77± 3.69 ○	74.10± 0.30	74.98± 3.60	68.07± 3.76 •	72.66± 2.00	74.02± 0.86
au1-balanced	76.95± 3.86	76.84± 3.03	59.30± 4.33 •	80.33± 2.95 ○	72.47± 3.90 •	69.13± 3.78 •	73.15± 4.23
balance-scale	89.71± 4.72	77.55± 3.55 •	56.72± 3.70 •	79.71± 3.40 •	77.83± 3.80 •	90.53± 1.67	71.56± 4.77 •
blood	77.93± 3.56	78.20± 3.71	76.21± 0.41	73.34± 3.94 •	71.16± 4.61 •	75.28± 3.47	75.01± 4.45
BreastTissue	64.28±11.46	65.41±12.83	40.65± 4.97 •	70.85±12.54	66.25±14.15	67.75±13.69	65.17±12.54
car	99.14± 0.92	92.30± 2.03 •	70.02± 0.16 •	93.46± 1.86 •	85.09± 3.13 •	85.46± 2.56 •	85.61± 2.56 •
cmc	54.08± 3.81	52.63± 3.99	42.70± 0.25 •	50.64± 3.62 •	47.33± 3.90 •	49.04± 3.99 •	50.07± 3.85 •
crx	81.42± 4.49	85.55± 3.94 ○	85.51± 3.96 ○	85.16± 4.18 ○	79.03± 4.99	77.87± 4.19	86.22± 3.87 ○
CTG	100.00± 0.00	100.00± 0.00	45.30± 0.27 •	99.91± 0.23	97.56± 1.59 •	96.63± 1.24 •	98.83± 0.76 •
glass	96.31± 3.95	97.33± 3.35	67.82± 2.50 •	97.43± 3.42	92.95± 6.11	83.13± 7.59 •	93.71± 4.91
iris	95.13± 4.63	94.73± 5.30	66.67± 0.00 •	94.27± 5.10	93.27± 4.97	95.53± 5.02	93.20± 5.92
labor-neg-data	90.27±11.96	78.60±16.58	78.77±14.37	86.90±13.47	83.13±16.07	93.57±10.27	90.60±12.99
post-operative	56.11±13.25	67.33± 7.87 ○	67.11± 8.19 ○	60.11±11.51	56.67±14.43	67.22± 7.64 ○	66.56± 8.43 ○
segment	96.59± 1.58	95.71± 1.85	30.39± 0.34 •	97.37± 1.28	94.57± 1.91 •	81.12± 2.26 •	90.52± 2.18 •
soybean	93.45± 2.82	91.78± 3.19	27.96± 2.13 •	91.80± 3.02	84.71± 5.14 •	92.94± 2.92	93.06± 2.98
zoo	95.66± 5.83	92.61± 7.33	60.43± 3.06 •	90.90± 7.01	71.41±18.18 •	96.95± 4.75	96.05± 5.61
ionosphere	92.39± 3.62	89.74± 4.38	82.57± 4.88 •	93.39± 4.13	88.71± 5.48	82.17± 6.14 •	89.54± 4.90
wine	98.02± 3.26	93.20± 5.90 •	57.91± 6.09 •	97.02± 3.89	93.21± 6.41 •	97.46± 3.70	98.65± 2.56
Average	84.98	83.74	60.56	84.31	79.08	81.91	82.86

○, • statistically significant improvement or degradation
in relation to AB-D (first column)

Table 4.2: Results of the T-Test comparing the implemented solution AdaBoost Dynamic (AB-D) with C4.5, Decision Stumps (Stumps), Random Forest (R. Forest), Random Tree (R. Tree), Naive Bayes(N. Bayes) and Bayes Network (Bayes Net.)

improvement compared to AB-D, but also five losses. As for Bagging and NN, the AB-D performance was very close to that of both algorithms; in the case of NN, AB-D had only one statistical improvement, and for Bagging both algorithms had four wins and four losses.

Table 4.4 presents the comparison of AB-D with AdaBoost.M1 using the same “weak” learners used in Table 4.2. Again, the number of iterations was 100 for all algorithms in the table. The reason for this number of iterations was to ensure a fair comparison between the new algorithm with AdaBoost.M1 using different classifiers, while avoiding possible advantages for any algorithm. Table 4.4 shows the results of the comparison of AB-D with AdaBoost.M1 with Random Forest, C4.5, Random Tree, Decision Stump, Naives Bayes and Bayes Network. The results indicate that Decision Stump had weak performance in ten of the simulations, statistically better performance in four simulations, and equal performance the others. AdaBoost with Random Trees had performance almost equal to AdaBoost with Decision Stumps, with one statistical victory over to AB-D. AdaBoost with C4.5 had statistically better performance four times and worse

Data set	AB-D	Bag.	NN	ZeroR	NBTree
au1	72.25± 3.79	80.28± 3.15 ○	69.73± 3.77	74.10±0.30	81.24± 3.78 ○
au1-balanced	76.95± 3.86	80.34± 2.87 ○	69.08± 3.77 ●	58.86±0.26 ●	78.92± 3.63
balance-scale	89.71± 4.72	83.27± 3.66 ●	90.98± 3.32	45.76±0.53 ●	75.90± 5.26 ●
blood	77.93± 3.56	78.17± 3.49	78.47± 2.85	76.21±0.41	75.36± 3.73
BreastTissue	64.28±11.46	70.37±12.52	64.37±11.42	19.01±1.42 ●	65.85±12.69
car	99.14± 0.92	92.08± 2.19 ●	99.40± 0.65	70.02±0.16 ●	94.44± 1.73 ●
cmc	54.08± 3.81	54.44± 4.19	54.08± 3.81	42.70±0.25 ●	51.21± 4.39 ●
crx	81.42± 4.49	85.65± 3.96 ○	83.00± 4.13	55.51±0.67 ●	84.94± 3.98 ○
CTG	100.00± 0.00	99.98± 0.10	100.00± 0.00	27.23±0.16 ●	98.06± 1.39 ●
glass	96.31± 3.95	97.52± 3.01	95.66± 4.10	35.51±2.08 ●	94.75± 5.18
iris	95.13± 4.63	94.20± 5.58	96.93± 4.07	33.33±0.00 ●	93.80± 5.30
labor-neg-data	90.27±11.96	83.63±16.98	90.27±11.96	64.67±3.07 ●	92.27±11.79
post-operative	56.11±13.25	68.78± 6.83 ○	55.44±14.08	70.00±5.12 ○	63.89± 9.91
segment	96.59± 1.58	96.23± 1.73	96.45± 1.59	15.73±0.33 ●	94.17± 1.94 ●
soybean	93.45± 2.82	87.00± 3.64 ●	93.35± 2.68	13.47±0.60 ●	92.87± 3.07
zoo	95.66± 5.83	42.59± 4.93 ●	95.66± 5.83	40.61±2.92 ●	95.52± 6.21
ionosphere	92.39± 3.62	91.17± 4.39	90.77± 4.52	64.10±1.36 ●	89.12± 4.99
wine	98.02± 3.26	94.48± 5.66	98.02± 3.26	39.91±2.01 ●	96.62± 3.92
Average	84.98	82.23	84.54	47.04	84.38

○, ● statistically significant improvement or degradation
in relation to AB-D (first column)

Table 4.3: Results of the T-Test comparing the implemented solution AdaBoost Dynamic (AB-D) with Bagging(Bag.), Neural Network (NN), ZeroR and NBTree

performance three times. Of the other algorithms, AdaBoost with Random Forest and AdaBoost with Bayes Network had better performance three times and worse performance four times, and AdaBoost with Naive Bayes had better performance twice and worse performance four times. In general, on average only AdaBoost with C4.5 was superior to AB-D.

A conclusion can be derived from the empirical results, namely that the AB-D algorithm has performance very close to the best "weak" learner used by AdaBoost.M1. This gives the ML analyst an advantage, because there is no need to define the specific learner to be boosted; the analyst can define a list of algorithms, and the final hypothesis will be a combination of different learners. As presented in Chapter 3, AB-D uses a different learner in each iteration, but the new AB-EL algorithm can be used to search for the "best" learner for a particular weight distribution. In the next section, the performance of the AB-EL is compared with that of AdaBoost.M1 and AB-D are presented in next section.

Data set	AB-D	Ada+R.Forest	Ada+C4.5	Ada+R.Tree	Ada+Stump	Ada+N.Bayes	Ada+BayesNet
au1	67.31± 4.80	73.96± 3.61 ○	75.89± 3.50 ○	68.50± 4.24	72.95± 2.46 ○	72.60± 1.99 ○	74.02± 0.86 ○
au1-balanced	74.09± 3.65	80.71± 3.19 ○	78.99± 3.74 ○	73.65± 4.35	77.90± 3.44 ○	76.16± 3.91	73.15± 4.23
balance-scale	89.47± 5.12	74.74± 4.85 •	75.98± 4.28 •	78.09± 3.88 •	71.77± 4.24 •	92.13± 2.90	74.44± 6.86 •
blood	77.38± 5.11	73.13± 4.11 •	77.43± 3.66	72.92± 4.11 •	78.84± 3.40	77.01± 3.07	75.01± 4.09
BreastTissue	64.28±11.60	71.14±11.75	71.17±13.05	66.91±12.70	40.65± 4.97 •	67.75±13.69	65.17±12.54
car	99.06± 1.03	93.87± 2.02 •	96.94± 1.47 •	84.90± 3.35 •	70.02± 0.16 •	90.25± 2.48 •	90.60± 2.34 •
cmc	54.08± 3.81	50.83± 3.69 •	49.62± 4.14 •	49.38± 4.09 •	42.70± 0.25 •	49.04± 3.98 •	50.15± 3.85 •
crx	78.00± 5.09	84.96± 3.87 ○	86.39± 3.88 ○	84.23± 5.20 ○	86.17± 4.22 ○	81.06± 4.14	86.28± 3.77 ○
CTG	100.00± 0.00	99.89± 0.25	100.00± 0.00	97.37± 1.76 •	45.30± 0.27 •	99.35± 0.53 •	99.86± 0.27
glass	96.22± 3.97	97.84± 3.22	97.33± 3.35	92.53± 7.64	67.82± 2.50 •	93.37± 5.63	97.54± 3.27
iris	95.20± 4.55	94.73± 5.04	94.53± 5.05	93.53± 5.48	94.60± 5.33	95.07± 5.73	93.73± 5.98
labor-neg-data	90.27±11.96	86.17±13.65	88.90±14.11	86.03±13.69	91.40±12.39	87.80±14.33	88.63±14.43
post-operative	57.22±14.42	58.33±10.98	56.33±15.09	59.78±12.61	67.11± 8.19 ○	66.89± 8.05 ○	66.44± 8.79 ○
segment	96.58± 1.56	97.62± 1.37	98.17± 1.10 ○	94.49± 1.95 •	30.39± 0.34 •	81.17± 2.32 •	93.71± 1.79 •
soybean	93.53± 2.77	92.16± 2.91	93.32± 2.81	89.87± 4.39 •	27.96± 2.13 •	92.05± 3.05	93.35± 2.65
zoo	95.66± 5.83	89.92± 8.49	95.48± 5.94	60.75±20.49 •	60.43± 3.06 •	96.95± 4.75	96.05± 5.61
ionosphere	91.37± 4.33	93.19± 4.27	93.93± 3.81	89.01± 5.76	92.63± 4.16	91.09± 4.77	93.25± 3.88
wine	98.02± 3.26	96.52± 4.21	96.45± 4.22	92.86± 5.87 •	91.69± 7.10 •	96.68± 3.83	97.13± 4.11
Average	84.32	83.87	84.83	79.71	67.24	83.69	83.81

○, • statistically significant improvement or degradation in relation to AB-D (first column)

Table 4.4: Results of the T-Test comparing the implemented solution (AdaBoost Dynamic (AB-D)) with AdaBoost.M1 with Random Forest(Ada+R.Forest), C4.5 (Ada+C.45), Random Tree (Ada+R.Tree), Decision Stump (Ada+Stump), Naives Bayes (Ada+N.Bayes) and Bayes Network(Ada+BayesNetwork). All classifiers were executed with 100 iterations.

4.3 AB-EL Comparison

Table 4.5 shows the comparison of AB-EL with AB-D, AdaBoost.M1 with Bagging, Neural Network, ZeroR and Naive Bayes Tree. The number of iterations for all the algorithms was 100, because they use the same AdaBoost.M1 weight calculation strategy. The only difference is that AB-EL chooses the classifier with the smallest error in the training set in each iteration. AdaBoost.M1 with ZeroR had low performance compared with AB-EL, with 13 losses and three wins. All the other algorithms had better performance than AB-EL, and the reason for this is that AdaBoost.M1 accuracy decreases when the learners have an error rate close to zero. These results motivated the weight adaptation presented in Chapter 3 that transformed AB-EL into AB-DL.

Data set	AB-EL	AB-D	Ada+Bagging	Ada+NN	Ada+ZeroR	Ada+NBTree
au1	63.84± 5.36	67.31± 4.80	73.56± 3.32 ∘	73.22± 3.84 ∘	74.10 ± 0.30 ∘	76.30± 3.92 ∘
au1-balanced	64.83± 4.21	74.09± 3.65 ∘	79.49± 2.81 ∘	74.08± 3.83 ∘	58.86 ± 0.26 •	80.19± 3.15 ∘
'balance-scale	74.83± 5.68	89.47± 5.12 ∘	77.25± 4.38	93.06± 3.13 ∘	45.76 ± 0.53 •	80.78± 4.59 ∘
blood	64.08± 5.27	77.38± 5.11 ∘	73.04± 4.16 ∘	78.47± 2.85 ∘	76.21 ± 0.41 ∘	77.61± 3.68 ∘
'BreastTissue	59.38±15.73	64.28±11.60	70.51±13.11 ∘	65.31±11.38	19.01 ± 1.42 •	68.67±11.97
car	85.67± 4.80	99.06± 1.03 ∘	97.11± 1.29 ∘	99.40± 0.65 ∘	70.02 ± 0.16 •	98.58± 0.85 ∘
cmc	43.53± 4.41	54.08± 3.81 ∘	51.21± 3.63 ∘	54.33± 3.90 ∘	42.70 ± 0.25	51.90± 4.52 ∘
crx	69.96± 6.80	78.00± 5.09 ∘	83.36± 5.01 ∘	83.14± 4.18 ∘	55.51 ± 0.67 •	86.30± 3.96 ∘
'CTG	98.18± 3.77	100.00± 0.00	99.99± 0.08	100.00± 0.00	27.23 ± 0.16 •	98.13± 1.40
glass	91.31± 9.72	96.22± 3.97	97.75± 3.23	95.89± 4.00	35.51 ± 2.08 •	95.72± 5.02
iris	89.87±10.49	95.20± 4.55	94.13± 5.21	96.20± 4.37	33.33 ± 0.00 •	94.33± 5.47
labor-neg-data	85.43±14.50	90.27±11.96	86.07±15.99	90.27±11.96	64.67 ± 3.07 •	93.67±10.44
post-operative	49.00±16.11	57.22±14.42	57.22±12.27	57.22±12.96	70.00 ± 5.12 ∘	56.56±14.05
segment	90.65± 4.35	96.58± 1.56 ∘	98.33± 0.98 ∘	96.45± 1.59 ∘	15.73 ± 0.33 •	98.19± 1.13 ∘
soybean	66.55± 8.31	93.53± 2.77 ∘	88.05± 3.18 ∘	93.35± 2.68 ∘	13.47 ± 0.60 •	94.01± 2.62 ∘
zoo	87.15±14.48	95.66± 5.83	42.59± 4.93 •	95.66± 5.83	40.61 ± 2.92 •	95.84± 5.97
ionosphere	83.82± 7.89	91.37± 4.33 ∘	93.71± 4.06 ∘	90.54± 4.59 ∘	64.10 ± 1.36 •	92.79± 4.37 ∘
wine	90.95± 8.61	98.02± 3.26 ∘	97.12± 3.87	98.02± 3.26 ∘	39.91 ± 2.01 •	96.50± 3.90
Average	75.50	84.32	81.14	85.26	47.04	85.34

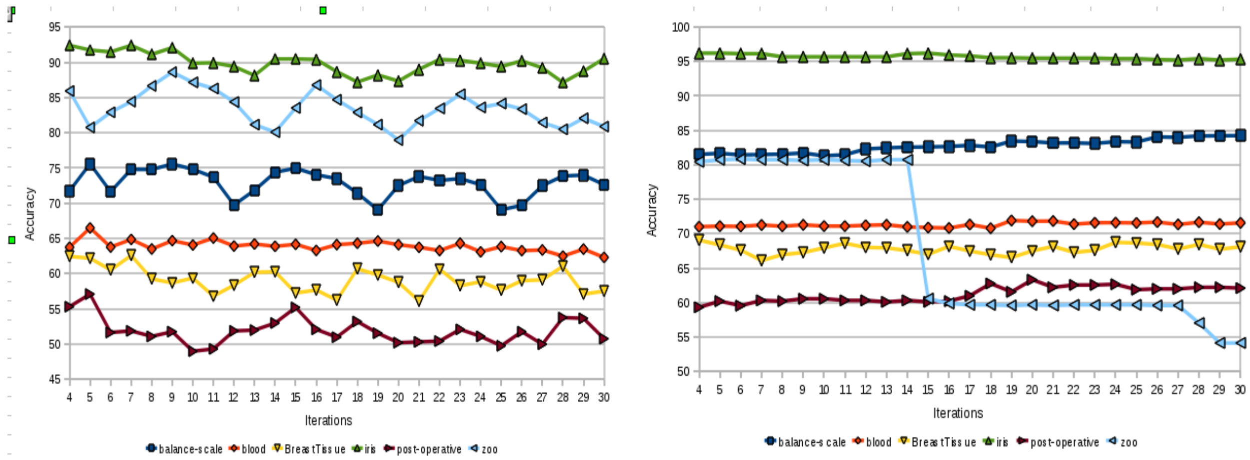
∘, • statistically significant improvement or degradation in relation to the first column (AB-EL)

Table 4.5: Implemented solution of AdaBoost Dynamic with Exponential function (AB-EL) against AB-D, AdaBoost.M1 with Bagging (Ada+Bagging), AdaBoost.M1 with Neural Network (Ada+NN), AdaBoost.M1 with ZeroR (Ada+ZeroR) and AdaBoost.M1 with NBTree (Ada+NBTree).

4.4 AB-DL Comparison

Table 4.6 presents graphs comparing the accuracy performance of AB-EL (Figure a) and AB-DL (Figure b) using UCI data sets, in relation to the number of iterations for both algorithms. The first evident result is that compared to AB-EL, the accuracy performance of AB-DL is almost linear. This is due to the weights being calculated using the derivative of the sigmoid function, rather than exponential function. For the rest of this work, the simulations comparing AB-DL to other algorithms will consider the fact that AB-DL has only ten iterations because of its linear performance. Another point about AB-DL can be seen in the `zoo` data set, which shows a significant loss of performance after 15 iterations. This problem is related to the low diversity among the classifiers, as well as the difficulty the learner has working with instances with weights.

In order to avoid problems with classifiers that ignore weights, all experiments in this chapter were done with AB-DL using 10 iterations with resampling, since works like [39, 4] showed that boosting algorithm improves their accuracy with resampling. The low number of iterations was determined by cross-validation on the data sets that had the best performance results. In order to make experiments more complete, this work also shows on Table 4.13 results considering AB-DL with 61 and 100 iterations.



(a) (b)

Table 4.6: Performance comparison of AB-EL and AB-DL with various data sets.

Table 4.7 presents the results of comparing AB-DL with resampling to some "weak" learners. The first conclusion is that, like AB-D, AB-DL performed better on average than all evaluated classifiers. Another observation is that Decision Stumps, Naive Bayes and Random Trees had high statistical losses. The performance of AB-DL compared to C4.5 had five statistical wins and only two losses; as opposed to AB-D, which was a statistical draw with C4.5 with three wins and three losses. AB-DL compared with Random Forest had almost the same performance with one win and two losses, but on average AB-DL was almost one percent better than Random Forest.

Table 4.8 shows the comparison of AB-DL with AdaBoost.M1 with Random Forest, AdaBoost.M1 with Decision Stumps, AdaBoost.M1 with C4.5, AdaBoost.M1 with Naive Bayes and AdaBoost.M1 with Bayes Networks. The average performance of AB-DL was better than all the other algorithms. With respect to AdaBoost.M1 with Decision Stumps, AB-DL had ten statistical wins and one loss. AdaBoost.M1 with Naive Bayes and AdaBoost.M1 with Bayes Network had almost equal performance, with six and five statistical losses respectively. AdaBoost.M1 with C4.5 improved compared to the results presented in Table 4.7, making it a good choice for a "weak" learner to deal with the data sets used in this work. AdaBoost.M1 with Random Forest did not improve on the results in Table 4.7, which shows that models generated by Random Forest could result

Data set	AB-DL	C4.5	Stumps	R. Forest	R. Tree	N. Bayes	Bayes Net.
au1	77.25± 3.29	77.77± 3.69	74.10± 0.30 •	74.98± 3.60	68.07± 3.76 •	72.66± 2.00 •	74.02± 0.86 •
au1-balanced	79.58± 3.19	76.84± 3.03 •	59.30± 4.33 •	80.33± 2.95	72.47± 3.90 •	69.13± 3.78 •	73.15± 4.23 •
'balance-scale	81.76± 5.02	77.55± 3.55 •	56.72± 3.70 •	79.71± 3.40	77.83± 3.80 •	90.53± 1.67 ○	71.56± 4.77 •
blood	74.96± 3.75	78.20± 3.71 ○	76.21± 0.41	73.34± 3.94	71.16± 4.61 •	75.28± 3.47	75.01± 4.45
'BreastTissue	66.17±12.19	65.41±12.83	40.65± 4.97 •	70.85±12.54	66.25±14.15	67.75±13.69	65.17±12.54
car	97.81± 1.37	92.30± 2.03 •	70.02± 0.16 •	93.46± 1.86 •	85.09± 3.13 •	85.46± 2.56 •	85.61± 2.56 •
cmc	52.20± 3.74	52.63± 3.99	42.70± 0.25 •	50.64± 3.62	47.33± 3.90 •	49.04± 3.99	50.07± 3.85
crx	85.46± 3.74	85.55± 3.94	85.51± 3.96	85.16± 4.18	79.03± 4.99 •	77.87± 4.19 •	86.22± 3.87
'CTG'	98.84± 1.38	100.00± 0.00 ○	45.30± 0.27 •	99.91± 0.23 ○	97.56± 1.59	96.63± 1.24 •	98.83± 0.76
glass	97.15± 3.82	97.33± 3.35	67.82± 2.50 •	97.43± 3.42	92.95± 6.11	83.13± 7.59 •	93.71± 4.91
iris	95.47± 4.43	94.73± 5.30	66.67± 0.00 •	94.27± 5.10	93.27± 4.97	95.53± 5.02	93.20± 5.92
labor-neg-data	92.10±11.79	78.60±16.58 •	78.77±14.37	86.90±13.47	83.13±16.07	93.57±10.27	90.60±12.99
post-operative	63.44±10.75	67.33± 7.87	67.11± 8.19	60.11±11.51	56.67±14.43	67.22± 7.64	66.56± 8.43
soybean	92.80± 2.79	91.78± 3.19	27.96± 2.13 •	91.80± 3.02	84.71± 5.14 •	92.94± 2.92	93.06± 2.98
segment	95.92± 1.53	95.71± 1.85	30.39± 0.34 •	97.37± 1.28 ○	94.57± 1.91 •	81.12± 2.26 •	90.52± 2.18 •
zoo	94.49± 6.45	92.61± 7.33	60.43± 3.06 •	90.90± 7.01	71.41±18.18 •	96.95± 4.75	96.05± 5.61
ionosphere	91.86± 4.65	89.74± 4.38	82.57± 4.88 •	93.39± 4.13	88.71± 5.48	82.17± 6.14 •	89.54± 4.90
wine	97.63± 3.41	93.20± 5.90 •	57.91± 6.09 •	97.02± 3.89	93.21± 6.41 •	97.46± 3.70	98.65± 2.56
Average	85.27	83.74	60.56	84.31	79.08	81.91	82.86

○, • statistically significant improvement or degradation
in relation to AB-DL (first column)

Table 4.7: Results of the T-Test comparing the implemented solution AdaBoost Dynamic with Logistic function (AB-DL) with resampling with C4.5, Decision Stumps (Stumps), Random Forest (R. Forest), Random Tree (R. Tree), Naive Bayes (N. Bayes) and Bayes Network (Bayes Net.)

in overfitting. Since the objective of this work is to compare the performance of AB-DL with AdaBoost.M1, the decision was made to use the same number of iterations for all "weak" learners.

Table 4.9 shows the results of AB-DL with resampling compared to Bagging, Neural Network (NN), ZeroR and Naive Bayes Tree (NBTree). As previously, all algorithms used the same default configurations presented by Weka, and on average AB-DL was superior to the others. AB-DL had 16 statistical wins when compared with ZeroR; the others had one, two and four statistical victories. NN was better than AB-DL four times, with two losses. Bagging was a statistical draw with AB-DL, with three losses and three wins, and NBTree had only one statistical win and three losses.

Table 4.10 shows AB-DL results compared with AdaBoost.M1, with the same learners as in Table 4.9. Again, the "weak" learner algorithms used the default configurations provided by Weka. AdaBoost.M1 with NN was the only algorithm to achieve results that were as good as AB-DL, with three statistical victories and losses. The results show

Data set	AB-DL	Ada+R.Forest	Ada+Stump	Ada+C4.5	Ada+N.Bayes	Ada+Bayes Net.
au1	77.23± 2.87	73.96± 3.61 •	72.95± 2.46 •	75.89± 3.50	72.60± 1.99 •	74.02± 0.86 •
au1-balanced	79.72± 3.40	80.71± 3.19	77.90± 3.44	78.99± 3.74	76.16± 3.91 •	73.15± 4.23 •
balance-scale	82.12± 4.01	74.74± 4.85 •	71.77± 4.24 •	75.98± 4.28 •	92.13± 2.90 ◦	74.44± 6.86 •
blood	75.52± 3.96	73.13± 4.11	78.84± 3.40 ◦	77.43± 3.66	77.01± 3.07	75.01± 4.09
BreastTissue	67.34±12.68	71.14±11.75	40.65± 4.97 •	71.17±13.05	67.75±13.69	65.17±12.54
car	97.89± 1.24	93.87± 2.02 •	70.02± 0.16 •	96.94± 1.47	90.25± 2.48 •	90.60± 2.34 •
cmc	52.43± 4.03	50.83± 3.69	42.70± 0.25 •	49.62± 4.14	49.04± 3.98	50.15± 3.85
crx	85.83± 4.15	84.96± 3.87	86.17± 4.22	86.39± 3.88	81.06± 4.14 •	86.28± 3.77
'CTG	99.94± 0.16	99.89± 0.25	45.30± 0.27 •	100.00± 0.00	99.35± 0.53 •	99.86± 0.27
glass	96.58± 4.12	97.84± 3.22	67.82± 2.50 •	97.33± 3.35	93.37± 5.63	97.54± 3.27
iris	95.47± 4.91	94.73± 5.04	94.60± 5.33	94.53± 5.05	95.07± 5.73	93.73± 5.98
labor-neg-data	89.57±13.11	86.17±13.65	91.40±12.39	88.90±14.11	87.80±14.33	88.63±14.43
post-operative	64.33±11.20	58.33±10.98	67.11± 8.19	56.33±15.09	66.89± 8.05	66.44± 8.79
segment	95.61± 1.58	97.62± 1.37 ◦	30.39± 0.34 •	98.17± 1.10 ◦	81.17± 2.32 •	93.71± 1.79 •
soybean	92.80± 2.79	92.16± 2.91	27.96± 2.13 •	93.32± 2.81	92.05± 3.05	93.35± 2.65
zoo	95.15± 6.21	89.92± 8.49	60.43± 3.06 •	95.48± 5.94	96.95± 4.75	96.05± 5.61
ionosphere	92.62± 4.42	93.19± 4.27	92.63± 4.16	93.93± 3.81	91.09± 4.77	93.25± 3.88
wine	96.62± 4.78	96.52± 4.21	91.69± 7.10	96.45± 4.22	96.68± 3.83	97.13± 4.11
Average	85.38	83.87	67.24	84.83	83.69	83.81

◦, • statistically significant improvement or degradation
in relation to AB-DL (first column)

Table 4.8: T-Test comparison of AB-DL with resampling against AdaBoost.M1 with Random Forest (Ada+R.Forest), AdaBoost.M1 with Decision Stumps (Ada+Stump), AdaBoost.M1 with C4.5 (Ada+C4.5), AdaBoost.M1 with Naive Bayes (Ada+N.Bayes) and AdaBoost.M1 with Bayes Networks (Ada+Bayes Net.)

that AdaBoost with ZeroR did not improve compared to the simple ZeroR algorithm, AdaBoost with Bagging had decreased performance with four statistical losses, and AdaBoost with NBTree increased its performance with one win and one loss. All algorithms were inferior to AB-DL when comparing their respective average accuracy.

4.5 AD-AC Simulations

Table 4.11 presents the comparison of AD-AC with AdaBoost.M1 using the following learners Random Forest, C4.5, Random Tree, Decision Stump, Naive Bayes and Bayes Network. All the AdaBoost.M1 experiments had 100 iterations, and AD-AC had ten iterations. As in the experiments with AB-DL, AD-AC was also tested with resampling and considering $\alpha = 0.35$. On average, AdaBoost.M1 was inferior to AD-AC, and AdaBoost.M1 with Random Tree and Decision Stumps had the worst results compared with AD-AC. On the other hand, AdaBoost.M1 with C4.5 had two statistical losses and one

Data set	AB-DL	Bag.	NN	ZeroR	NBTree
au1	77.25± 3.29	80.28± 3.15 ○	69.73± 3.77 ●	74.10±0.30 ●	81.24± 3.78 ○
au1-balanced	79.58± 3.19	80.34± 2.87	69.08± 3.77 ●	58.86±0.26 ●	78.92± 3.63
'balance-scale	81.76± 5.02	83.27± 3.66	90.98± 3.32 ○	45.76±0.53 ●	75.90± 5.26 ●
blood	74.96± 3.75	78.17± 3.49 ○	78.47± 2.85 ○	76.21±0.41	75.36± 3.73
'BreastTissue	66.17±12.19	70.37±12.52	64.37±11.42	19.01±1.42 ●	65.85±12.69
car	97.81± 1.37	92.08± 2.19 ●	99.40± 0.65 ○	70.02±0.16 ●	94.44± 1.73 ●
cmc	52.20± 3.74	54.44± 4.19	54.08± 3.81	42.70±0.25 ●	51.21± 4.39
crx	85.46± 3.74	85.65± 3.96	83.00± 4.13	55.51±0.67 ●	84.94± 3.98
'CTG	98.84± 1.38	99.98± 0.10 ○	100.00± 0.00 ○	27.23±0.16 ●	98.06± 1.39
glass	97.15± 3.82	97.52± 3.01	95.66± 4.10	35.51±2.08 ●	94.75± 5.18
iris	95.47± 4.43	94.20± 5.58	96.93± 4.07	33.33±0.00 ●	93.80± 5.30
labor-neg-data	92.10±11.79	83.63±16.98	90.27±11.96	64.67±3.07 ●	92.27±11.79
post-operative	63.44±10.75	68.78± 6.83	55.44±14.08	70.00±5.12	63.89± 9.91
segment	95.92± 1.53	96.23± 1.73	96.45± 1.59	15.73±0.33 ●	94.17± 1.94 ●
zoo	94.49± 6.45	42.59± 4.93 ●	95.66± 5.83	40.61±2.92 ●	95.52± 6.21
ionosphere	91.86± 4.65	91.17± 4.39	90.77± 4.52	64.10±1.36 ●	89.12± 4.99
wine	97.63± 3.41	94.48± 5.66	98.02± 3.26	39.91±2.01 ●	96.62± 3.92
soybean	92.80± 2.79	87.00± 3.64 ●	93.35± 2.68	13.47±0.60 ●	92.87± 3.07
Average	85.27	82.23	84.54	47.04	84.38

○, ● statistically significant improvement or degradation
in relation to AB-DL (first column)

Table 4.9: Results of the T-Test comparing the implemented solution AdaBoost Dynamic with Logistic function (AB-DL) using resampling against Bagging(Bag.), Neural Network (NN), ZeroR and NBTree

win compared with AD-AC. AdaBoost.M1 with Naive Bayes and Bayes Network had seven and six statistical losses respectively, and no victories.

Table 4.12 presents the comparisons of AD-AC with AdaBoost.M1 with the “weak” learners: Bagging, Neural Network, ZeroR and Naive Bayes Tree. All AdaBoost.M1 experiments had 100 iterations, and AD-AC had ten iterations with resampling and considering $\alpha = 0.35$. Again, the average performance of AD-AC was superior to all the algorithms tested. In terms of statistical significance, only AdaBoost.M1 with Naive Bayes Tree had performance equal to AD-AC.

Table 4.13 shows the comparison of AD-AC with AB-D and AB-DL. AB-DL was tested with 10, 61 and 100 iterations, while AD-AC was tested with ten iterations, because it, theoretically, will tend to have better results than AdaBoost.M1 with small number of iterations; and AB-D was tested with 100 iterations. The reason AB-DL was tested with three different configurations is related to the theoretical restrictions presented in Chapter 3, which showed that AB-DL requires a minimum number of iterations to achieve a theoretical error bound. In Chapter 3, it was presented that AB-DL

Data set	AB-DL	Ada+Bag	Ada+NN	Ada+ZeroR	Ada+NBTree
au1	77.23± 2.87	73.56± 3.32 •	73.22± 3.84 •	74.10±0.30 •	76.30± 3.92
au1-balanced	79.72± 3.40	79.49± 2.81	74.08± 3.83 •	58.86±0.26 •	80.19± 3.15
'balance-scale	82.12± 4.01	77.25± 4.38 •	93.06± 3.13 ◦	45.76±0.53 •	80.78± 4.59
blood	75.52± 3.96	73.04± 4.16	78.47± 2.85 ◦	76.21±0.41	77.61± 3.68
'BreastTissue	67.34±12.68	70.51±13.11	65.31±11.38	19.01±1.42 •	68.67±11.97
car	97.89± 1.24	97.11± 1.29	99.40± 0.65 ◦	70.02±0.16 •	98.58± 0.85
cmc	52.43± 4.03	51.21± 3.63	54.33± 3.90	42.70±0.25 •	51.90± 4.52
crx	85.83± 4.15	83.36± 5.01	83.14± 4.18 •	55.51±0.67 •	86.30± 3.96
'CTG	99.94± 0.16	99.99± 0.08	100.00± 0.00	27.23±0.16 •	98.13± 1.40 •
glass	96.58± 4.12	97.75± 3.23	95.89± 4.00	35.51±2.08 •	95.72± 5.02
iris	95.47± 4.91	94.13± 5.21	96.20± 4.37	33.33±0.00 •	94.33± 5.47
labor-neg-data	89.57±13.11	86.07±15.99	90.27±11.96	64.67±3.07 •	93.67±10.44
post-operative	64.33±11.20	57.22±12.27	57.22±12.96	70.00±5.12	56.56±14.05
segment	95.61± 1.58	98.33± 0.98 ◦	96.45± 1.59	15.73±0.33 •	98.19± 1.13 ◦
soybean	92.80± 2.79	88.05± 3.18 •	93.35± 2.68	13.47±0.60 •	94.01± 2.62
zoo	95.15± 6.21	42.59± 4.93 •	95.66± 5.83	40.61±2.92 •	95.84± 5.97
ionosphere	92.62± 4.42	93.71± 4.06	90.54± 4.59	64.10±1.36 •	92.79± 4.37
wine	96.62± 4.78	97.12± 3.87	98.02± 3.26	39.91±2.01 •	96.50± 3.90
Average	85.38	81.14	85.26	47.04	85.34

◦, • statistically significant improvement or degradation
in relation to AB-DL (first column)

Table 4.10: Results of the T-Test comparing the implemented solution AdaBoost Dynamic with Logistic function (AB-DL) using resampling against AdaBoost.M1 with Bagging(Ada+Bag), AdaBoost.M1 with Neural Network (Ada+NN), AdaBoost.M1 with ZeroR (Ada+ZeroR) and AdaBoost.M1 with NBTree (Ada+NBTree).

total error would be bounded if the number of iterations was 61 assuming that all "weak" learners have the same error rate. Even though it is not possible to guarantee that classifiers will have the same error rate in practical experiments, it was decided to test AB-DL with 61 iterations, and to show that it also would not suffer from overfitting, tests were also conducted with 100 iterations. It was found that AB-DL reached constant performance with a high number of iterations, indicating that smaller number of iterations may provide better accuracy. Results indicate that AD-AC improves the results of all AdaBoost Dynamic versions, while avoiding the need to search for the learner with the smallest error rate. And, as expected, AD-AC also achieves higher accuracy with a small number of iterations.

Data set	AD-AC	Ada+RForest	Ada+C4.5	Ada+RTree	Ada+Stump	Ada+NBayes	Ada+BayesNet
au1	77.18± 3.07	73.96± 3.61 •	75.89± 3.50	68.50± 4.24 •	72.95± 2.46 •	72.60± 1.99 •	74.02± 0.86 •
au1-balanced	80.05± 3.13	80.71± 3.19	78.99± 3.74	73.65± 4.35 •	77.90± 3.44	76.16± 3.91 •	73.15± 4.23 •
'balance-scale	90.98± 3.32	74.74± 4.85 •	75.98± 4.28 •	78.09± 3.88 •	71.77± 4.24 •	92.13± 2.90	74.44± 6.86 •
blood	75.75± 3.84	73.13± 4.11 •	77.43± 3.66	72.92± 4.11 •	78.84± 3.40 ∘	77.01± 3.07	75.01± 4.09
'BreastTissue	64.37±11.42	71.14±11.75	71.17±13.05	66.91±12.70	40.65± 4.97 •	67.75±13.69	65.17±12.54
car	96.76± 1.32	93.87± 2.02 •	96.94± 1.47	84.90± 3.35 •	70.02± 0.16 •	90.25± 2.48 •	90.60± 2.34 •
crx	85.94± 3.41	84.96± 3.87	86.39± 3.88	84.23± 5.20	86.17± 4.22	81.06± 4.14 •	86.28± 3.77
'CTG	100.00± 0.00	99.89± 0.25	100.00± 0.00	97.37± 1.76 •	45.30± 0.27 •	99.35± 0.53 •	99.86± 0.27
glass	95.66± 4.10	97.84± 3.22	97.33± 3.35	92.53± 7.64	67.82± 2.50 •	93.37± 5.63	97.54± 3.27
iris	96.93± 4.07	94.73± 5.04	94.53± 5.05	93.53± 5.48	94.60± 5.33	95.07± 5.73	93.73± 5.98
labor-neg-data	92.80±10.98	86.17±13.65	88.90±14.11	86.03±13.69	91.40±12.39	87.80±14.33	88.63±14.43
post-operative	61.11±10.77	58.33±10.98	56.33±15.09	59.78±12.61	67.11± 8.19	66.89± 8.05	66.44± 8.79
segment	96.45± 1.59	97.62± 1.37 ∘	98.17± 1.10 ∘	94.49± 1.95 •	30.39± 0.34 •	81.17± 2.32 •	93.71± 1.79 •
soybean	93.35± 2.68	92.16± 2.91	93.32± 2.81	89.87± 4.39 •	27.96± 2.13 •	92.05± 3.05	93.35± 2.65
zoo	95.66± 5.83	89.92± 8.49	95.48± 5.94	60.75±20.49 •	60.43± 3.06 •	96.95± 4.75	96.05± 5.61
cmc	54.08± 3.81	50.83± 3.69 •	49.62± 4.14 •	49.38± 4.09 •	42.70± 0.25 •	49.04± 3.98 •	50.15± 3.85 •
ionosphere	93.42± 3.85	93.19± 4.27	93.93± 3.81	89.01± 5.76 •	92.63± 4.16	91.09± 4.77	93.25± 3.88
wine	98.02± 3.26	96.52± 4.21	96.45± 4.22	92.86± 5.87 •	91.69± 7.10 •	96.68± 3.83	97.13± 4.11
Average	86.03	83.87	84.83	79.71	67.24	83.69	83.81

∘, • statistically significant improvement or degradation in relation to first column (AD-AC)

Table 4.11: T-Test comparison among AD-AC and AdaBoost.M1 with Random Forest (Ada+RForest), AdaBoost.M1 with C4.5 (Ada+C4.5), AdaBoost.M1 with Random Tree (Ada+RTree), AdaBoost.M1 with Decision Stumps (Ada+Stumps), AdaBoost.M1 with Naive Bayes (Ada+NBayes), and AdaBoost.M1 with Bayes Network (Ada+BayesNet). All AdaBoost.M1 simulations considered 100 iterations and AD-AC had only ten iterations. AD-AC considered $\alpha = 0.35$.

4.6 AD-AC Evaluation with Noisy Data Set

Table 4.14 presents the T-Test comparison considering the same data sets, but with 10% additional noise. The noise is generated by the `AddNoise` class, and is available in the pre-processing stage of Weka's Explorer option. As presented in Chapter 3, AD-AC will have a tendency to remove the noise, because the algorithm reduces the weight of the misclassified instances. All AdaBoost.M1 evaluations used 100 iterations, and AD-AC used ten iterations with resampling. Table 4.14 shows the comparison of AD-AC with AdaBoost.M1 with the Random Forest, C4.5, Decision Stumps, Random Tree, Naive Bayes and Bayes Network learners. Average results indicate that AD-AC was better than all the algorithms, despite of the fact that the algorithms had reduced performances. Individual performances showed that AdaBoost.M1 with Random Forest and Naive Bayes had six statistical losses, while AdaBoost.M1 with Decision Stump and Random Tree had

Data set	AD-AC	Ada+Bagg	Ada+NN	Ada+ZeroR	Ada+NBTree
au1	77.18± 3.07	73.56± 3.32 •	73.22± 3.84 •	74.10± 0.30 •	76.30± 3.92
au1-balanced	80.05± 3.13	79.49± 2.81	74.08± 3.83 •	58.86± 0.26 •	80.19± 3.15
'balance-scale	90.98± 3.32	77.25± 4.38 •	93.06± 3.13 ◦	45.76± 0.53 •	80.78± 4.59 •
blood	75.75± 3.84	73.04± 4.16 •	78.47± 2.85	76.21± 0.41	77.61± 3.68
'BreastTissue	64.37±11.42	70.51±13.11	65.31±11.38	19.01± 1.42 •	68.67±11.97
car	96.76± 1.32	97.11± 1.29	99.40± 0.65 ◦	70.02± 0.16 •	98.58± 0.85 ◦
crx	85.94± 3.41	83.36± 5.01	83.14± 4.18 •	55.51± 0.67 •	86.30± 3.96
'CTG	100.00± 0.00	99.99± 0.08	100.00± 0.00	27.23± 0.16 •	98.13± 1.40 •
glass	95.66± 4.10	97.75± 3.23	95.89± 4.00	35.51± 2.08 •	95.72± 5.02
iris	96.93± 4.07	94.13± 5.21	96.20± 4.37	33.33± 0.00 •	94.33± 5.47
labor-neg-data	92.80±10.98	86.07±15.99	90.27±11.96	64.67± 3.07 •	93.67±10.44
post-operative	61.11±10.77	57.22±12.27	57.22±12.96	70.00± 5.12 ◦	56.56±14.05
segment	96.45± 1.59	98.33± 0.98 ◦	96.45± 1.59	15.73± 0.33 •	98.19± 1.13 ◦
soybean	93.35± 2.68	88.05± 3.18 •	93.35± 2.68	13.47± 0.60 •	94.01± 2.62
zoo	95.66± 5.83	42.59± 4.93 •	95.66± 5.83	40.61± 2.92 •	95.84± 5.97
cmc	54.08± 3.81	51.21± 3.63 •	54.33± 3.90	42.70± 0.25 •	51.90± 4.52
ionosphere	93.42± 3.85	93.71± 4.06	90.54± 4.59 •	64.10± 1.36 •	92.79± 4.37
wine	98.02± 3.26	97.12± 3.87	98.02± 3.26	39.91± 2.01 •	96.50± 3.90
Average	86.03	81.14	85.26	47.04	85.34

◦, • statistically significant improvement or degradation in relation to first column (AD-AC).

Table 4.12: T-Test comparing AD-AC with AdaBoost.M1 with Bagging (Ada+Bagg), AdaBoost.M1 with Neural Network (Ada+NN), AdaBoost.M1 with ZeroR (Ada+ZeroR) and AdaBoos.M1 with Naive Bayes Tree (Ada+NBTree). All AdaBoost.M1 experiments considered 100 iterations. AD-AC was executed with ten iterations and with resampling. Also, AD-AC was executed with $\alpha = 0.35$.

nine and twelve losses respectively, and AdaBoost.M1 with C4.5 and Bayes Network had five losses.

Table 4.15 continues the comparisons of AD-AC with AdaBoost.M1 using the following algorithms: Bagging, Neural Network and Naive Bayes Tree. Once again, AD-AC was superior to all. It had eight statistical victories compared with AdaBoost.M1 using Bagging. Compared with the results in Table 4.12, AdaBoost.M1 using Bagging had only six statistical losses, which indicates that it may not be noise resistant. AdaBoost.M1 with Neural Network had the same performance as in Table 4.12. AdaBoost.M1 with Naive Bayes Trees had only two statistical losses, while in Table 4.12 it had two losses and two wins. AD-AC had less statistical losses with data sets containing noise than in previous experiments without noise, which indicates that the ideas presented in Chapter 3 about AD-AC resistance to noise are confirmed in practise.

Data set	AD-AC	AB-D	AB-DL (10)	AB-DL (61)	AB-DL (100)
au1	77.18± 3.07	67.31± 4.80 ●	77.23± 2.87	77.25± 3.29	77.25± 3.29
au1-balanced	80.05± 3.13	74.09± 3.65 ●	79.72± 3.40	79.58± 3.19	79.58± 3.19
balance-scale	90.98± 3.32	89.47± 5.12	82.12± 4.01 ●	81.76± 5.02 ●	81.76± 5.02 ●
blood	75.75± 3.84	77.38± 5.11	75.52± 3.96	74.96± 3.75	74.96± 3.75
BreastTissue	64.37±11.42	64.28±11.60	67.34±12.68	66.17±12.19	66.17±12.19
car	96.76± 1.32	99.06± 1.03 ○	97.89± 1.24 ○	97.81± 1.37 ○	97.81± 1.37 ○
cmc	54.08± 3.81	54.08± 3.81	52.43± 4.02	52.20± 3.73	52.20± 3.73
crx	85.94± 3.41	78.00± 5.09 ●	85.83± 4.15	85.46± 3.74	85.46± 3.74
CTG	100.00± 0.00	100.00± 0.00	99.94± 0.16	98.84± 1.38 ●	98.84± 1.38 ●
glass	95.66± 4.10	96.22± 3.97	96.58± 4.12	97.15± 3.82	97.15± 3.82
iris	96.93± 4.07	95.20± 4.55	95.47± 4.91	95.47± 4.43	95.47± 4.43
labor-neg-data	92.80±10.98	90.27±11.96	89.57±13.11	92.10±11.79	92.10±11.79
post-operative	61.11±10.77	57.22±14.42	64.33±11.20	63.44±10.75	63.44±10.75
segment	96.45± 1.59	96.58± 1.56	95.61± 1.58	95.92± 1.53	95.92± 1.53
soybean	93.35± 2.68	93.53± 2.77	92.80± 2.79	92.55± 2.95	92.55± 2.95
zoo	95.66± 5.83	95.66± 5.83	95.15± 6.21	94.49± 6.45	94.49± 6.45
ionosphere	93.42± 3.85	91.37± 4.33	92.62± 4.42	91.86± 4.65	91.86± 4.65
wine	98.02± 3.26	98.02± 3.26	96.62± 4.78	97.63± 3.41	97.63± 3.41
Average	86.03	84.32	85.38	85.26	85.26

○, ● statistically significant improvement or degradation in relation to AD-AC

Table 4.13: T-Test results comparing the AD-AC with resampling and $\alpha = 0.35$, with AB-D and AB-DL (with resampling). AD-AC had ten iterations, AB-D had 100 iterations, and AB-DL was tested with different three number of iterations ten, 61 and 100.

4.7 Algorithm Evaluation with ANOVA

Figure 4.1 shows the comparison results of AD-AC with other AdaBoost.M1 algorithms, using ANOVA with Bonferroni *post-hoc* test. The key finding is that the accuracy of AD-AC is not a statistically different than other algorithms, and its main advantage is related to the number of iterations to reach the same level of accuracy as the others. AD-AC required only ten iterations, while the other algorithms tested used 100 iterations. These results motivated adapting the same strategies used in AD-AC in OzaBoost.

Figure 4.2 shows the comparison results of AD-AC using the same ANOVA configuration as presented in Figure 4.1, with the main difference that the data sets had 10% noise added. The number of experiments was reduced, because the main objective was to show the AD-AC performance in relation to AdaBoost.M1 with different learners. The results show that there is no statistical difference between AD-AC and the other algorithms, with exception of AdaBoost.M1 with Decision Stumps. It is important to note that AdaBoost.M1 with Bagging (Ada+Bagg) and AdaBoost.M1 with Random

Data set	AD-AC	Ada+R.Forest	Ada+C4.5	Ada+Stump	Ada+R.Tree	Ada+NBayes	Ada+BayesNet
au1	67.20± 3.71	65.92± 3.77	65.98± 3.91	68.63± 2.69	64.74± 4.49	68.24± 2.26	69.30± 0.46
au1	70.82± 3.80	71.45± 3.52	69.32± 3.91	71.35± 3.37	65.40± 4.31 •	70.00± 3.60	66.13± 2.67 •
'balance-scale	81.39± 4.15	64.58± 4.89 •	66.26± 4.41 •	66.62± 5.22 •	67.46± 4.48 •	81.10± 2.98	68.64± 4.51 •
blood	69.37± 4.35	66.58± 4.60 •	71.28± 3.70	71.57± 3.37	66.69± 4.17	71.42± 3.38	69.69± 2.97
'BreastTissue	54.55±12.85	59.35±13.15	62.17±14.14	34.25± 8.17 •	55.46±14.66	55.28±13.13	56.32±13.66
car	83.95± 2.46	81.92± 2.30 •	81.67± 2.45 •	64.00± 0.23 •	74.32± 3.73 •	75.32± 2.54 •	75.35± 2.51 •
cmc	51.24± 4.32	47.96± 3.91 •	47.18± 3.63 •	41.28± 0.26 •	46.15± 3.96 •	47.30± 3.96 •	48.80± 3.71
crx	75.64± 5.16	72.32± 5.24 •	73.06± 4.68	78.46± 5.10	72.71± 5.64	76.43± 5.42	78.06± 5.54
'CTG	88.03± 1.96	87.23± 2.00	87.14± 2.10	41.53± 0.70 •	85.72± 2.93 •	86.08± 2.27 •	86.41± 2.13 •
glass	84.11± 7.44	86.22± 6.70	85.71± 6.60	61.15± 4.63 •	75.89±10.42 •	68.52±10.78 •	83.10± 7.75
iris	85.00± 8.86	81.27± 9.41	81.20±10.00	86.87± 9.16	77.73±11.22 •	85.40± 9.70	86.60± 8.86
labor-neg-data	84.33±15.18	79.83±17.80	80.27±16.22	77.60±16.70	77.50±18.04	70.03±18.04 •	76.40±16.71
post-operative	51.67±11.54	48.89±12.93	54.33±13.30	61.56± 7.96 ◦	47.89±12.60	56.78±11.16	55.44±10.65
segment	85.79± 2.77	85.93± 2.65	84.25± 3.06 •	28.57± 1.00 •	84.21± 2.97 •	65.77± 2.80 •	79.53± 3.04 •
soybean	80.44± 4.67	76.56± 5.62 •	79.31± 4.99	24.48± 2.21 •	72.74± 5.61 •	82.66± 4.31	82.47± 4.30
wine	83.20± 7.58	85.03± 8.03	83.15± 8.62	80.45± 8.72	74.45± 9.44 •	85.61± 7.96	87.53± 7.14 ◦
zoo	83.75± 9.88	80.58±10.36	63.11±21.80 •	55.49± 5.68 •	56.86±19.17 •	80.95±11.88	81.57±10.70
ionosphere	83.84± 4.32	82.08± 5.84	82.05± 5.32	81.53± 5.30	73.92± 7.13 •	81.39± 5.85	82.88± 5.42
Average	75.80	73.54	73.19	60.86	68.88	72.68	74.12

◦, • statistically significant improvement or degradation in relation to first column (AD-AC)

Table 4.14: T-Test comparison of AD-AC with AdaBoost.M1 using Random Forest (Ada+R.Forest), AdaBoost.M1 using C4.5 (Ada+C4.5), AdaBoost.M1 with Decision Stump (Ada+Stump), AdaBoost.M1 with Random Tree (Ada+R.Tree), AdaBoost.M1 with Naive Bayes (Ada+NBayes), and AdaBoost.M1 with Bayes Network (Ada+BayesNet). All AdaBoost.M1 experiments had 100 iterations, and AD-AC had ten iterations with resampling. AD-AC considered $\alpha = 0.35$. Data sets used had 10% noise added.

Tree (Ada+R.Tree) were close to the least accuracy of AD-AC, which suggests that these algorithms are not reliable with noisy data sets.

4.8 Conclusions About Results

Results presented in this chapter call for some conclusions. The first is the observation that AB-DL's weight calculation helps to reduce the overfitting. This can be noticed by results presented in Table 4.13, which showed the execution of the algorithms with multiple number of iterations. Results indicate that there are no numerical difference between 61 and 100 iterations.

The second conclusion is that AD-AC offers better models than previous algorithms - even AdaBoost.M1 algorithms - with much lower number of iterations. The main reason

Data set	AD-AC	Ada+Bag	Ada+NN	Ada+NBTree
au1	67.20± 3.71	65.04± 3.96	64.01± 4.86	67.56± 3.86
au1	70.82± 3.80	70.14± 3.79	66.09± 4.05 •	72.36± 3.73
'balance-scale	81.39± 4.15	66.42± 4.77 •	81.39± 4.15	70.07± 4.28 •
blood	69.37± 4.35	66.31± 4.63 •	72.76± 3.64 ○	69.59± 3.40
'BreastTissue	54.55±12.85	58.10±13.81	54.49±12.11	55.73±14.09
car	83.95± 2.46	80.61± 2.51 •	86.74± 1.85 ○	83.71± 2.37
cmc	51.24± 4.32	47.88± 3.98 •	51.46± 4.05	49.52± 4.09
crx	75.64± 5.16	69.52± 4.74 •	71.36± 5.45 •	73.17± 4.59
'CTG	88.03± 1.96	87.35± 2.04	88.03± 1.96	88.20± 2.37
glass	84.11± 7.44	86.13± 6.67	80.47± 8.40 •	86.42± 6.72
iris	85.00± 8.86	81.20± 9.73	84.87± 8.93	85.67± 9.73
labor-neg-data	84.33±15.18	78.67±17.21	77.80±16.62	80.67±15.15
post-operative	51.67±11.54	48.67±13.10	47.11±12.94	48.67±12.61
segment	85.79± 2.77	84.79± 3.13	85.79± 2.77	83.64± 3.48 •
soybean	80.44± 4.67	65.70± 4.80 •	79.55± 4.83	82.60± 4.27
wine	83.20± 7.58	82.53± 8.70	83.13± 8.24	84.32± 7.98
zoo	83.75± 9.88	38.63± 5.68 •	83.75± 9.88	83.73± 9.97
ionosphere-10noise	83.84± 4.32	80.74± 5.21 •	77.43± 6.35 •	82.88± 5.13
Average	75.80	69.91	74.23	74.92

○, • statistically significant improvement or degradation

Table 4.15: T-Test comparison of AD-AC with AdaBoost.M1 using Bagging (Ada+Bag), AdaBoost.M1 with Neural Network (Ada+NN), and AdaBoost.M1 with Naive Bayes Tree (Ada+NBTree). All AdaBoost.M1 experiments considered 100 iterations and AD-AC had 10 iterations with resampling. AD-AC considered $\alpha = 0.35$. Data sets used had 10% noise added.

for using only ten iterations is related to AD-AC's theoretical results, which indicate that it will achieve higher levels of accuracy with low number of iterations. To test this hypothesis, it was decided to execute the experiments with a low number of iterations. In general, it is also recommended to use AD-AC, instead of AB-DL or AB-D, because its classifier combination offers higher accuracy in relation to other algorithms.

4.9 Summary

Chapter 4 presents the experimental results comparing AdaBoost Dynamic accuracy performance with AdaBoost.M1 with various "weak" learners and with the "weak" learners in isolation, without boosting. The main conclusions about the results are listed:

- AdaBoost Dynamic (AB-D) algorithm has a similar accuracy performance to AdaBoost.M1 algorithm using a specific "weak" learner with higher accuracy.

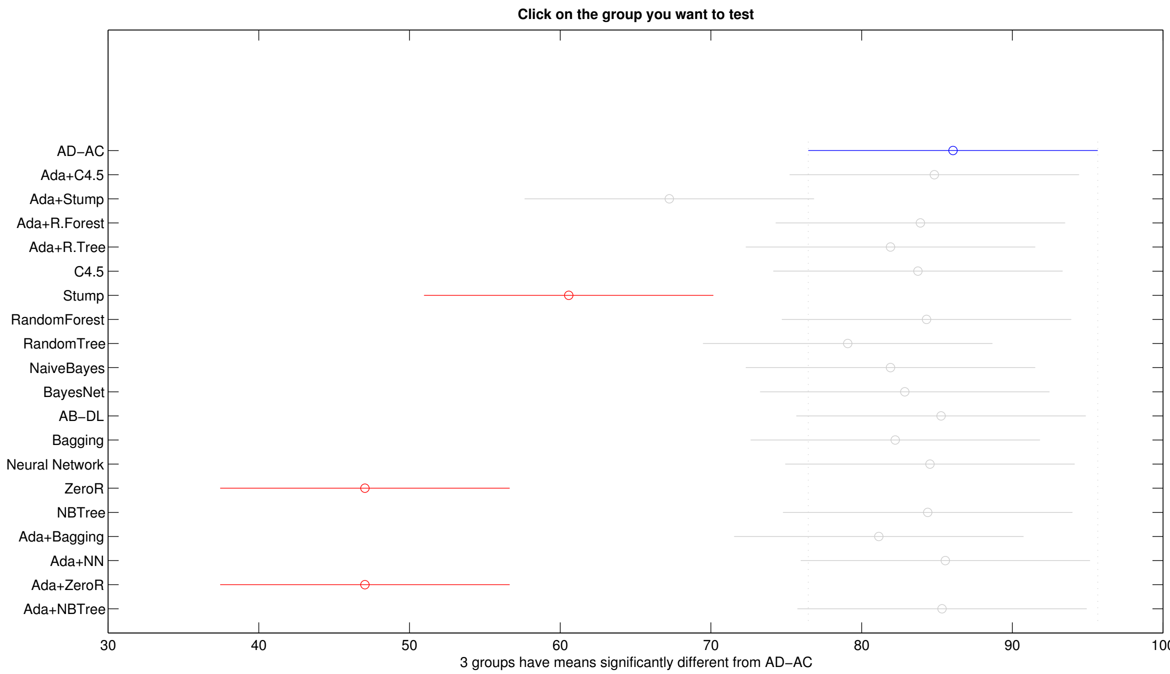


Figure 4.1: Comparison of all algorithms, considering AD-AC as base algorithm. AD-AC and AB-DL were tested considering ten iterations, and all AdaBoost.M1 algorithms were executed with 100 iterations. ANOVA was used to evaluate the algorithms and for dual comparison Bonferroni test was applied.

- In order to evaluate if AB-D would increase its accuracy, search for the algorithm with smallest error rate was implemented. This version is called AB-EL and the accuracy results were lower than the AB-D and AdaBoost.M1. Since AB-EL uses the same weight calculation used in AB-D and all learners have their respective error rates close to zero, AB-EL tends to overfit, and reduces its capacity to generalize on unseen data. This behaviour is aligned with previous works (see [35]).

- AB-EL has accuracy performance limitations, but the idea to use a classifier with smallest error in the list makes sense, and the main limitation of AB-EL is on its weight calculation. A new algorithm, called AB-DL, was implemented to allow the search for best classifier, and it uses a a different loss function based on sigmoid function. The accuracy results have improved in relation to AB-D and AdaBoost.M1, although there was no statistical significant differences among the

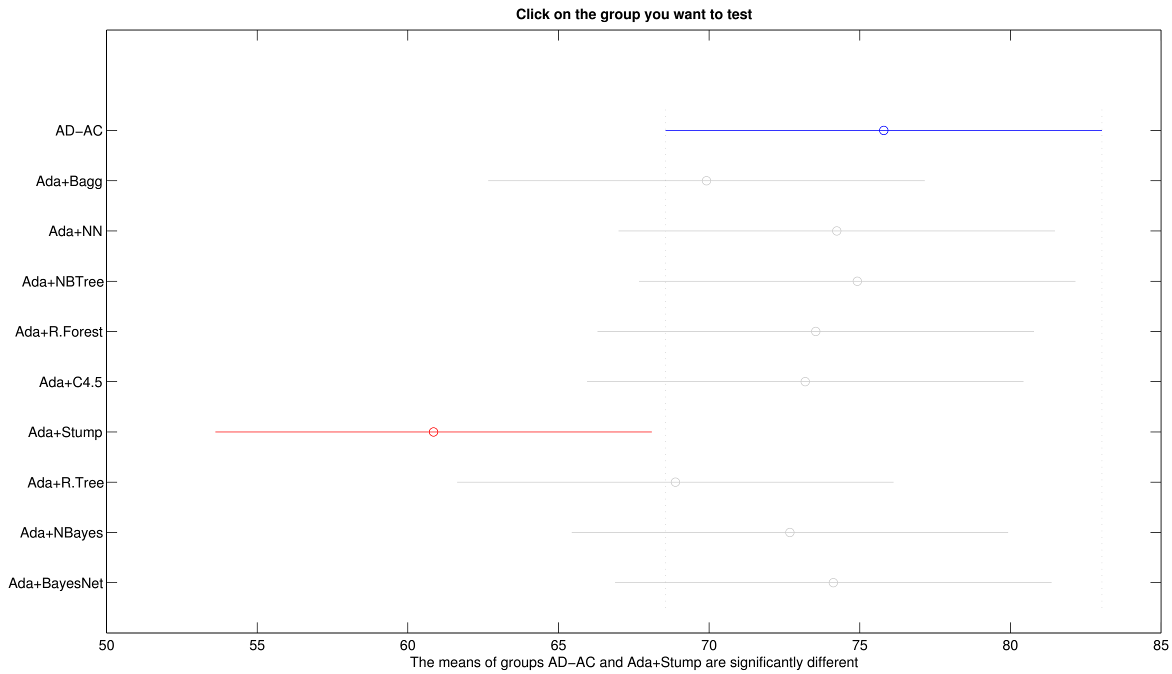


Figure 4.2: Comparison of all algorithms, considering AD-AC as base algorithm. AD-AC and AB-DL were tested considering ten iterations, and all AdaBoost.M1 algorithms were executed with 100 iterations. ANOVA was used to evaluate the algorithms and for dual comparison Bonferroni test was applied.

algorithms. The main issue with AB-DL is that its learners have accuracy performance close to zero in all iterations, which reduces the diversity among the classifiers, and limits further improvements to this algorithm.

- Considering the limitations presented by previous algorithms, it was decided to combine the "weak" learner execution used in AB-D and weight calculation used in AB-DL. This new version is called AD-AC. AD-AC also improved the accuracy results in relation to all other algorithms, but it did not have a significant statistical difference. Since AD-AC does not require a search, the algorithm allows more variation related to learner execution, which increases the diversity among classifiers.

- It was also conjectured that AB-DL and AD-AC may give better accuracy results when considering data with noise. This chapter presents the results with the same data sets added with 10% noise. All algorithms had accuracy reductions, but AD-AC had the smallest decrease in relation to others. However, using the ANOVA with Bonferroni test, it was not possible to find statistical differences among the algorithms.

- In general, it is noticeable that AD-AC provides better results than previous algorithms, for two reasons: 1) it does not require a search to build the model; and 2) the algorithm has better theoretical constraints than AB-DL. AD-AC's theoretical boundary also strongly indicates that the algorithm will achieve better models with lower number of iterations.

Chapter 5

AdaBoost Dynamic and Data Streams

This chapter shows the adaptation of AD-AC algorithm to work with data stream problem. There are two main reasons to work with this type of problem:

- There is a demand to extract patterns from massive amounts of data. Data stream mining is the process of extracting information from rapidly produced data records. The data can be from different sources, such as networks or stock markets. Since the data set is typically generated very quickly, it is difficult to store it for further evaluation. This makes it a challenge to use traditional Machine Learning (ML) algorithms in a data stream environment, because some ML algorithms demand access to training instances at various times, and they do not scale up to the data volume generated by data streams. Instead of creating an ML algorithm from scratch for data streams, the usual approach is to adapt these existing algorithms to deal with data streams.

- The other reason is related with the use of ML algorithms inside mobile devices. The increasing computational power in cell phones and tablets generated the demand to use ML algorithms to improve pattern detection in this environment. These devices still have memory constraints, but it is possible to adapt ML algorithms to work with these limitations.

OzaBoost [36], the AdaBoost adaptation to work with data streams, suffers from theoretical limitations. The limitations are based on the estimation of weights assigned to instances during training process to be assumed as a Poisson distribution. Since AD-

AC has strong theoretical background, it does make sense to use this algorithm to work with data streams. This adaptation is called OzaBoost Dynamic (OzaDyn). This chapter will present the adaptations executed in OzaBoost and the main results of experiments with synthetic and real data sets. The OzaDyn comparison uses a different strategy than other algorithms proposed in this work, because data streams are typically generated very quickly, and it is difficult for algorithms such as boosting to make multiple passes on the training data. The results presented in this chapter confirm those shown in Chapter 4, but it is also easy to observe that OzaDyn reduces memory footprint when compared with OzaBoost, because it combines multiple algorithms and a new weight calculation.

5.1 OzaBoost: AdaBoost for Data Streams

One of the first algorithms adapted for this purpose was Decision Trees (DT). Domingos and Hulten proposed Very Fast Decision Trees (VFDT) in [9], that modified the traditional splitting policy used by decision trees such as InfoGain, and applied the Hoeffding boundary as the splitting criterion.

A natural extension of this is to use ensemble methods to combine different algorithms in order to predict results from data streams. The issue with ensemble methods is that they are slower than decision trees, though they do offer better accuracy if the data stream cannot be linearly separated. The two main problems to adapt AdaBoost and Bagging for data streams are: 1) both algorithms depend on multiple passes on the data set, and 2) they require on previous knowledge of the size of the data set.

As presented in [3], there are two approaches to address the issues of transforming AdaBoost into an algorithm to process data streams: 1) Block boosting, which involves storing some sets of instances, reweighting these instances following the same process as AdaBoost, and then training a learner to produce new models to include in the ensemble; and 2) Parallel boosting that, "(...)involves feeding examples as they arrive into a base data stream algorithm that builds each ensemble member in parallel"[3]. One problem with the block approach is that it needs to prune the models that do not help the accuracy [1]. OzaBoost was introduced by Oza and Russell [36], and it is a parallel boosting strategy, which follows the same approach as AdaBoost with the exception of weight calculation,

[1] On the other hand, parallel boosting "(...)does not directly emulate the strictly sequential process of AdaBoost, as models further down the sequence will start learning from weights that depend on earlier models that are also evolving over time, but the hope is that in time, the models will converge towards an AdaBoost-like configuration"[3].

since AdaBoost depends on previous knowledge of the number of instances available for training. In order to address this issue, Oza and Russell used a Poisson distribution to calculate the weights. The disadvantages of OzaBoost are related to model size, and evaluation time of the algorithm, which can grow exponentially - making this approach not suitable for some data sets.

The OzaBoost algorithm adapts AdaBoost.M1 to process data streams, and updates the weights with a Poisson distribution, as it does not have previous information about the number of instances available for training. Another reason to use the Poisson distribution is related with the number of instances used for training. OzaBoost will update models in parallel for *one* instance per iteration, which means that the algorithm will not have the real performance of the learners during the training stage. Thus, OzaBoost will use the information if each "weak" learner correctly classified each instance or not, and for each misclassification the weight of that instance will be increased by the Poisson distribution, and decreased if the instance is correctly classified.

The algorithm presented in Table 5.1 performs as follows: The first step initializes the array h_t, indexing all the learners and the variables that will store the number of correct classifications (λ_t^{sc}) and incorrect classifications (λ_t^{sw}). Then for all training examples the algorithm updates a variable k with the Poisson value (line 5), then uses this value to increase or decrease the weight of the instance for each h_t (line 6). The algorithm then checks if the new h_t correctly classifies the instance (line 7). If it does, λ_t^{sc} is updated with the value of λ_d (line 8), and λ_d is updated so its value is equal to $\lambda_d \frac{N}{2\lambda_t^{sc}}$ (line 9). When it is detected that h_t misclassified the example, the algorithm updates λ_t^{sw} in the same way as in line 8, and λ_d will be equal to $\lambda_d \frac{N}{2\lambda_t^{sw}}$(lines 11 and 12). At any time, the algorithm can output the final hypothesis $H_{final} = \arg\max_{y \in Y} \sum_{t:h_t(x)=y} \log \frac{1}{\beta_t}$, that is equal to the original AdaBoost.M1, with $\beta_t = \frac{\epsilon_t}{1-\epsilon_t}$, and $\epsilon_t = \frac{\lambda_t^{sw}}{\lambda_t^{sw}+\lambda_t^{sc}}$.

Massive Online Analysis (MOA) [3] documentation shows that the developers tried to use the original boosting weight calculation instead of the Poisson, but: "Overall the algorithm performed better with the random Poisson element than without it, so it seems to be a worthwhile approach even if the theoretical underpinning is weak"[3].

MOA improves to this algorithm by adapting data windows into OzaBoost; in MOA this is known as `OzaBoostAdwin`. In simple terms, this version adapts the algorithm `ADWIN`[2] to OzaBoost. `ADWIN` is a statistical drift detector that keeps a window of length W using $O(logW)$ memory and $O(logW)$ processing time per item. The MOA documentation [3] explains that `ADWIN` dynamically adapts the window size for the variations of the data stream, which reduces the costs related to the growing window process, and

Initialize base models h_t for all $t \in \{1, 2, ..., T\}$, $\lambda_t^{sc} = 0, \lambda_t^{sw} = 0$

for *all training examples* **do**

  Set "weight" of example $\lambda_d = 1$

  for all t **do**

    Set $k = Poisson(\lambda_d)$

    Update h_t with the current example with the weight k

    if h_t correctly classifies the example **then**

      $\lambda_t^{sc} \leftarrow \lambda_t^{sc} + \lambda_d$

      $\lambda_d \leftarrow \lambda_d \frac{N}{2\lambda_t^{sc}}$

    else

      $\lambda_t^{sw} \leftarrow \lambda_t^{sw} + \lambda_d$

      $\lambda_d \leftarrow \lambda_d \frac{N}{2\lambda_t^{sw}}$

    end if

  end for

end for

Anytime output

Calculate the $\epsilon_t = \frac{\lambda_t^{sw}}{\lambda_t^{sw} + \lambda_t^{sc}}$, and $\beta_t = \frac{\epsilon_t}{1-\epsilon_t}$ for all m

Return:$H_{final} = \arg\max_{y \in Y} \sum_{t:h_t(x)=y} \log \frac{1}{\beta_t}$

Table 5.1: Oza and Russell's Online Boosting as implemented by MOA. N is the number of examples seen, and T is the number of models.

eliminates the need to check all the "subwindows" for possible reductions. All the experiments in this work, and the modifications on the OzaBoost algorithm, were done in `OzaBoostAdwin`, due to of these advantages for detecting variations in the data.

5.2 AD-AC and Data Streams

The last contribution of this work is the adaptation of the AD-AC weight calculation - presented in Chapter 3 - in OzaBoost [36], which is a parallel boosting algorithm developed to work with data streams. OzaBoost does not allow multiple passes on the instances during the training phase, but for each iteration the algorithm builds a different

Input: Base models h_t, where $h_t \neq h_{t+1}$;
T number of base models;
User defined $\phi \in [0, 1]$;
Sequence of examples m $\langle (x_1, y_1), ..., (x_m, y_m) \rangle$ with labels $y_i \in Y = \{1, ..., k\}$

Initialize base models h_t for all $t \in \{1, 2, ..., T\}$, $\lambda_t^{sc} = 0, \lambda_t^{sw} = 0$
for *all training examples* **do**
  Set "weight" of example $\lambda_d = 1$
  Set current error $\varepsilon = 0.99$
  for all t **do**
    Update h_t with current example with weight $\frac{1}{1+e^{-(1-(\varepsilon(1-\varepsilon)))^{(1-\phi)}}}$
    if h_t correctly classifies the example **then**
      $\lambda_t^{sc} \leftarrow \lambda_t^{sc} + \lambda_d$
      $\lambda_d \leftarrow \lambda_d \frac{N}{2\lambda_t^{sc}}$
    else
      $\lambda_t^{sw} \leftarrow \lambda_t^{sw} + \lambda_d$
      $\lambda_d \leftarrow \lambda_d \frac{N}{2\lambda_t^{sw}}$
      $\varepsilon \leftarrow \frac{\lambda_t^{sw}}{\lambda_t^{sw} + \lambda_t^{sc}}$
    end if
  end for
end for
Anytime output
Calculate the $\epsilon_t = \frac{\lambda_t^{sw}}{\lambda_t^{sw} + \lambda_t^{sc}}$, and $\beta_t = 1 - (\epsilon_t(1 - \epsilon_t))$ for all t
Return: $H_{final} = \arg\max_{y \in Y} \sum_{t:h_t(x)=y} \beta_t$

Table 5.2: OzaBoostDynamic algorithm that adapts the weight calculation used in AD-AC algorithm to OzaBoost algorithm. N is the number of examples seen.

learner that is updated with the values of the new instance. This process can be halted at any time, and the output is the same as AdaBoost.M1. The first problem with OzaBoost is related to weight calculation of misclassified instances. AdaBoost.M1 knows the data set size, but with data streams this is not possible. So, instead of normalizing the weights based on the "weak" learner error rate, OzaBoost uses the Poisson distribution to generate the weights. This has some theoretical issues [3], because the weights are based on a distribution that is not related to the data set's distribution. To provide a solution, this work proposes changing the OzaBoost weight calculation to AD-AC weight calculation. The second issue with OzaBoost is related to the number of possible learners to be used; the AD-AC learner variation policy will be adapted to deal with the data streams problem.

Table 5.2 presents the OzaBoostDynamic, a modification of the original OzaBoost. The main change is in line 6, where it does not take the Poisson distribution into consideration. Instead, the algorithm executes a simplified weight calculation as in AD-AC, because this test only considers if the classification was correct or not. The learners do not have any information about the confidence in their decision. The second change is in the base model's initialization (line 1), which can be changed to an array of different classifiers executed in each iteration. The algorithm has the same output as AD-AC, and the same theoretical properties.

The practical advantages of this approach are related to model size and improvements in evaluation time, while keeping accuracy performance close to that of the OzaBoost algorithm. The results are be presented in Section 5.4.

5.3 Datasets and Environment Configuration

All experiments involving data streams were implemented using Massive Online Analysis(MOA), because it has implemented specific algorithms for working with large volumes of data - on the order of millions of instances. Weka cannot support this type of problem, because the whole data set must be in memory for training purposes. Algorithms in MOA assume that there are memory constraints, and thus they update their models for each instance separately. This process avoids storing large volumes of data in memory.

Another MOA feature is that it has implemented various artificial data streams generators. This work uses two of these: Random Tree Generator (RTG) and the Function Generator, also known as Agrawal data stream. Agrawal stream produces a stream containing nine attributes, six numeric and three categorical, which were not explicitly described by the authors, but a sensible conclusion is that these attributes describe hypothetical loan applications[3]. MOA also allows the addition of artificial drifts to any type of data stream generator, by using the class `ConceptDriftStream`. `ConceptDriftStream` defines the probability that every new instance of the stream belongs to the new concept after the drift. The sigmoid function is used to calculate these probabilities.

Figure 5.1 shows the sigmoid function $f(t) = \frac{1}{1+e^{-s(t-t_0)}}$, where the derivative $f'(t_0) = \frac{s}{4}$ [3] and the tangent of angle α is equal to the derivative $f'(t_0)$. Notice also that

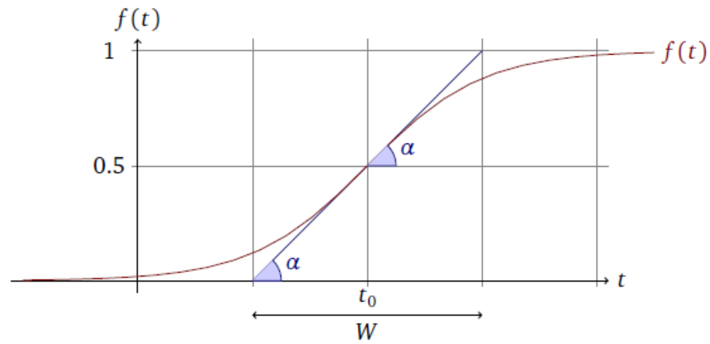


Figure 5.1: Sigmoid function $f(t) = \frac{1}{1+e^{-s(t-t_0)}}$, reproduced from [3].

$\tan\alpha = \frac{\frac{1}{2}}{\frac{W}{2}} = \frac{1}{W}$ and since $s = 4\tan\alpha$, then $s = \frac{4}{W}$. This shows that the parameter s gives the length W and the angle α. So, to generate the new values for the data streams it is necessary to know the parameter t_0, and the length of the change W. In MOA implementation, developers also created the option for users to change the angle α.

The tests were conducted as follows. First the RTG and Agrawal data stream were executed without any artificial drift, and then the Agrawal data stream was executed with the addition of concept drift. The number of instances used was 60 million for the RTG and 50 million for the Agrawal data stream. The difference is due to the fact that OzaBoost crashes once the Agrawal data set reaches 50 million instances.

All the simulations were executed a machine with 4 Core Intel(R) Xeon(R) 64 bits CPU 2.67GHz with 8GB RAM. The operating system was Linux, with Kernel 2.6.32.26-175. The Java heap size was 3GB for all algorithms, with exception of OzaBoost which required the memory to be extended to 7GB.

In terms of evaluation, cross-validation is not the best approach to deal with data streams, as it is most effective for data sets of limited size, with examples that have the assumption of independence of among the examples, and that consider the examples generated from stationary distributions [20]. With these limitations, the evaluation options for data streams are limited to [20]:

- Train-Test: this approach separates the data set into training and test data.

- Predictive Sequential or *Prequential*: is the approach where each instance can be used to test the model before it is used for training, and from this the accuracy can be incrementally updated in [3]. The final error (prequential-error) is calculated based on the sum of a loss function between the predicted and observed values.

This work used the prequential approach to evaluate the algorithms because it provides more information, since its output is a learning curve that can be used to measure how well the algorithms performed. In this sense, if one algorithm has a curve above another, it means it has better performance.

5.4 Prequential Results

Figure 5.2 shows the comparison of OzaBoostDynamic (OzaDyn) with OzaBoost with Very Fast Decision Trees (VFDT), VFDT, and Naive Bayes Multinomial (NBM). In MOA there are various implementations of OzaBoost, and one uses an algorithm called `ADWIN` [2], which is a data stream change detector. The class used in MOA is `OzaBoostADWIN`, and all algorithms implemented used this variation. OzaDyn was tested in two configurations. The first considered the algorithm with one classifier (VFDT), and the second considered the combination of results of two different classifiers (VFDT and NBM) by alternating their executions. OzaBoost used VFDT as its base classifier. All OzaDyn and OzaBoost algorithms considered only ten iterations, and used their default configurations.

Figure 5.2(a) presents the accuracy results of using NBM, VFDT, OzaBoost with VFDT, OzaDyn with VFDT, and OzaDyn with both VFDT and NBM, applied on the RTG data set. Figure 5.2(b) shows the accuracy results for the same classifiers with an Agrawal data set. Both data sets were without concept drift. All algorithms had accuracy close to 100%, with exception of NBM.

Figures 5.2(c) and 5.2(d) show the processing time in seconds it took to execute the model evaluation for the algorithms. Figure 5.2(c) indicates that all algorithms had linear performances relative to the number of instances. However, OzaBoost required more than four hours to evaluate the models, and OzaDyn with two classifiers had better performance than OzaBoost and OzaDyn with one classifier, since it used NBM, and took approximately two hours to finish the task. VFDT and NBM had the best performances. Figure 5.2(d) shows that OzaBoost with the Agrawal data set took almost 14 hours to evaluate when dealing with 50 million instances. The main reason for this is that OzaBoost output is a sum of various logarithms, and this operation, with the increasing number of instances, is very costly. In comparison, the performance of OzaDyn with one classifier that is based on the sums of simple multiplications is exponentially better.

Figures 5.2(e) and 5.2(f) illustrate the evolution of the model size (in bytes) in relation to the number of instances. Figure 5.2(e) shows that OzaBoost with VFDT applied to RTG maintained an almost constant size until it reached 50 million instances, then it leveled off near 350Mb, and stayed there until the end of the experiments. The explanation for this unusual size change is that the `ADWIN` algorithm automatically removes the worst classifier when a change (drift) is detected, and then a new classifier is added. OzaBoost is sensitive to this variation, because it passes the performance of the learners to the `ADWIN` algorithm iteratively, and then `ADWIN` automatically adapts a new window size, which increases the evaluation time. OzaDyn is less prone to this variation, because the weights are based on the error calculated by each model, and it automatically reduces the weights of instances that did not help improve the model. This makes the algorithm almost linear with respect to model size and evaluation time. The effect is not passed on to the `ADWIN` algorithm, which avoids learner removal and creation. Interestingly, OzaDyn combining two classifiers (VFDT and NBM) had less memory consumption than the other algorithms. This is because NBM maintains the memory below 200Mb, thereby reducing the memory footprint of the entire model. Figure 5.2(f) shows the memory consumption for the Agrawal data set, using the same set of algorithms. All the algorithms had linear performance, with only OzaDyn with two classifiers slightly worse than OzaDyn with one classifier. Both algorithms were better than regular OzaBoost.

Figures 5.4(a), 5.4(b) and 5.4(c) present the performance of the same algorithms using the Agrawal data set when artificial drift is added. Figure 5.4(a) indicates that all the algorithms, with exception of NBM, reach 100% accuracy. Figure 5.4(b) is the evaluation time in seconds for all the algorithms, and the first result shows that OzaDyn with one classifier is worse than OzaBoost, and OzaDyn with two classifiers is better than the others. This is because OzaDyn with two classifiers combines a complex classifier (VFDT) and a simple classifier (NBM). The same pattern can be seen in Figure 5.4(c), which presents the model size (in bytes) for all the algorithms. OzaDyn with one classifier and OzaBoost have exactly the same model size, while OzaDyn with two classifiers saves memory space by combining the complex and simpler models. One important observation is that when the data stream presents drifts, the models' respective sizes and evaluation times tend to increase. VFDT and NBM also increase both measures.

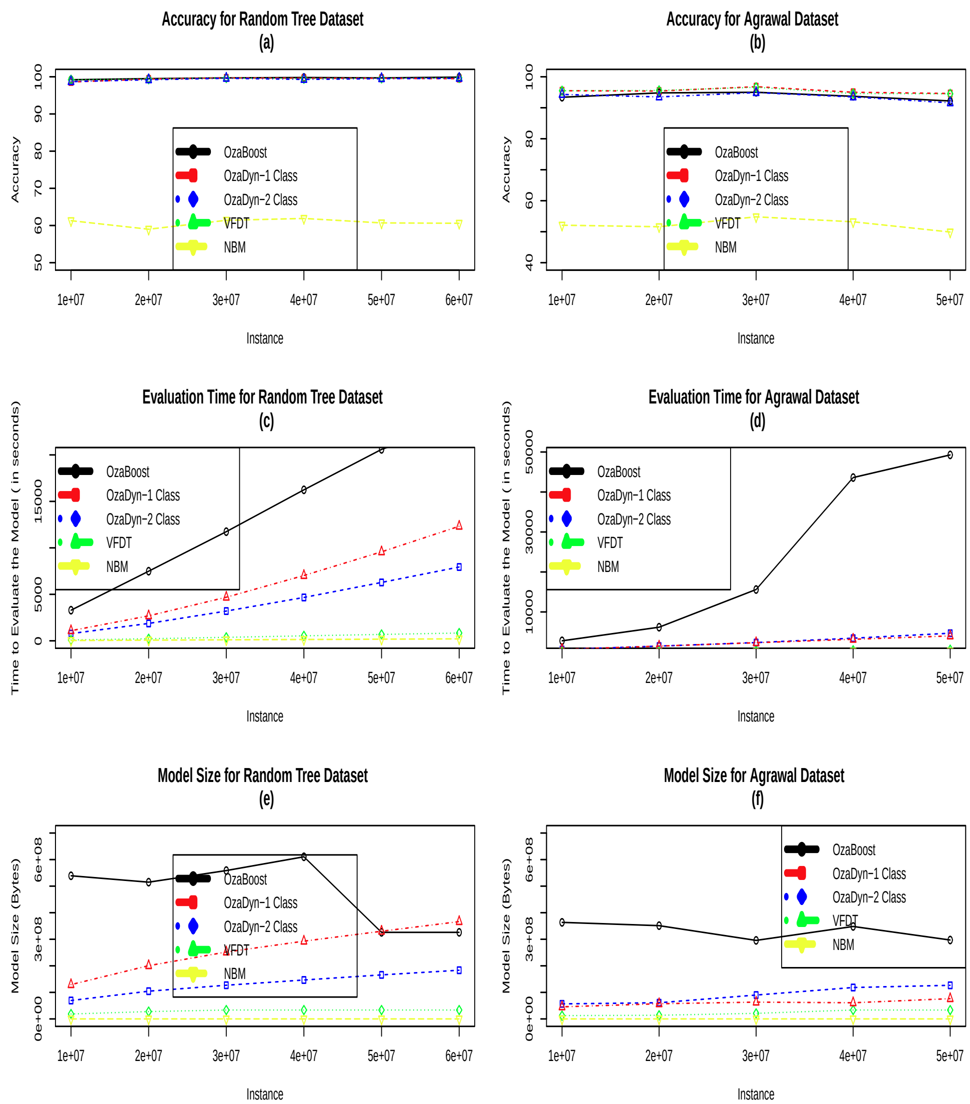


Figure 5.2: This figure presents the performance of OzaDyn with one classifier (OzaDyn - 1 Class) and with two classifiers (OzaDyn - 2Class) compared to original OzaBoost, VFDT and Naive Bayes Multinomial (NBM) for the Random Tree and Agrawal data sets. Figures (a) and (b) compare the accuracy of the algorithms, figures (c) and (d) compare the evaluation time, and (e) and (f) present the model size.

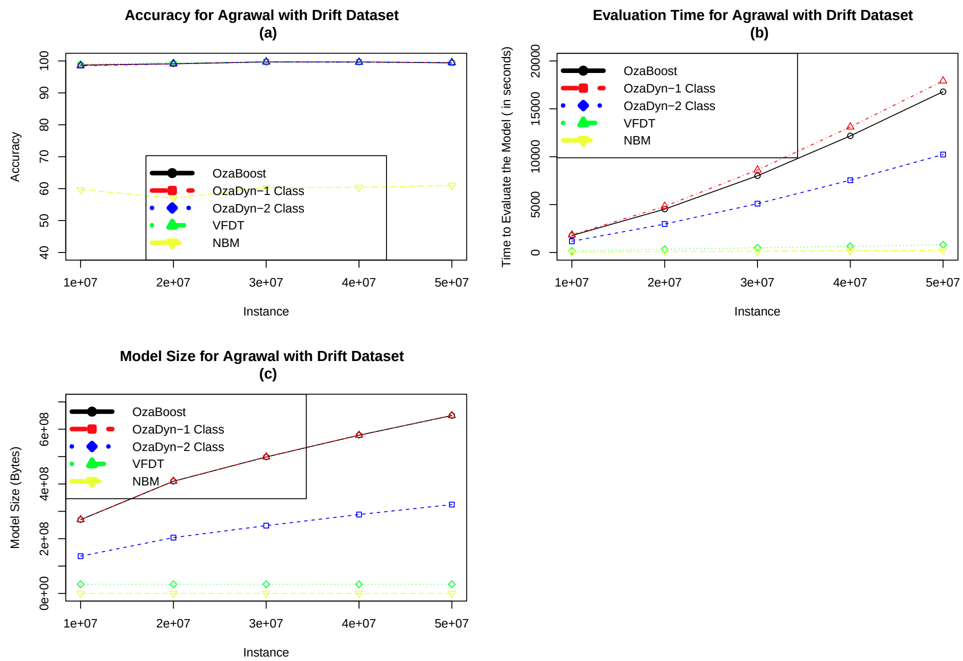


Figure 5.3: This figure presents the performance of OzaDyn compared with original OzaBoost, VFDT and Naive Bayes Multinomial (NBM) for the Agrawal data set with drifts. Figure (a) compares the accuracy of algorithms, figure (b) compares the evaluation time of the algorithms and figure (c) presents the model size.

5.5 OzaBoostDynamic Comparison with a Real Data Set

This section shows the comparison results of OzaDyn with two network data sets. The problem in this case is predicting which application is generating a data flow. This information is captured with the Ground Truth [22] tool, which performs two main tasks: 1) it executes a scan in network traffic, and 2) it associates this information with the applications executed in the end-hosts under observation. GT developers made their data publicly available in their website[2]; which can be requested to execute tests. In this work two data sets were used: the GT data set, and data collected from an university network using GT tool. The university data set is not public [13], because it has real traffic data.

The data sets show the following common features: time stamp, IP source, IP desti-

[2] `http://www.ing.unibs.it/ntw/tools/gt/`

nation, transport port source, transport port destination, Deep Packet Inspection (DPI) label(s), application name and transport protocol. The results from a signature-based analysis (known as DPI) were generated by the GT tool and the application name. The transport protocol only informs which protocol was used, Transmission Control Protocol (TCP) or User Datagram Protocol (UDP). Data from GT had both protocols, but the university data had only TCP connection. Data from GT website had 78,999 flow records and data from university had 7,669 records. Regarding to data set sizes, from the point of view of network data and the definition of training sizes used in [40], the university data set is small and GT is considered large. However, these data sets *cannot* be considered true data streams data sets, because they are very small compared to tested data sets that are synthetically generated.

The GT data set is composed of 61% Web flows, 28% Peer-to-peer (P2P), 5% Skype and 6% other flows, and has flows from TCP and UDP connections. The university data set is composed of 80% Web flows, 9% P2P, 2% Skype, and 9% other flows. These data sets also contain noise in the form of errors in the labels and the presence of unique flows, which cannot be correctly classified.

Figure 5.4 shows the performance of OzaBoostDynamic with one classifier (VFDT) and two classifiers (VFDT and NBM), with traditional OzaBoost with VFDT. The results show the same behaviour found with the synthetic data: OzaBoostDynamic with one or two classifiers had a much smaller level of memory and evaluation time footprint than OzaBoost, while maintaining close to the same accuracy performance. Figure 5.4 (a) shows the accuracy performance of all the algorithms using the GT data set, which has a high number of drifts that caused problems for all algorithms. OzaBoostDynamic with one or two classifiers had slighter lower accuracy than OzaBoost, due to the high number of drifts in the data set, because the weight calculation based on Poisson distribution seems to separate the classes better. Figure 5.4 (b) shows the accuracy for the university data set, which has fewer drifts, and OzaBoostDynamic with one or two classifiers had performance almost equal to OzaBoost.

Figures 5.4 (c) and (d) show the time it took to evaluate the models with the GT and university data sets. OzaBoostDynamic with one or two classifiers had better results than OzaBoost. In both cases, OzaBoost had a close to exponential evaluation time, while the OzaBoostDynamic time was almost linear. In the case of OzaBoostDynamic with one classifier, this is due to the fact that the weight calculation and the loss function based on the AD-AC strategy improve the result. OzaBoostDynamic with two classifiers adds the use of different algorithms, and when combined these tend to reduce evaluation time. Figures 5.4 (e) and (f) show the model size of all the algorithms for the GT and university data sets. These results are repeated for the same reasons discussed previously in Figures (c) and (d).

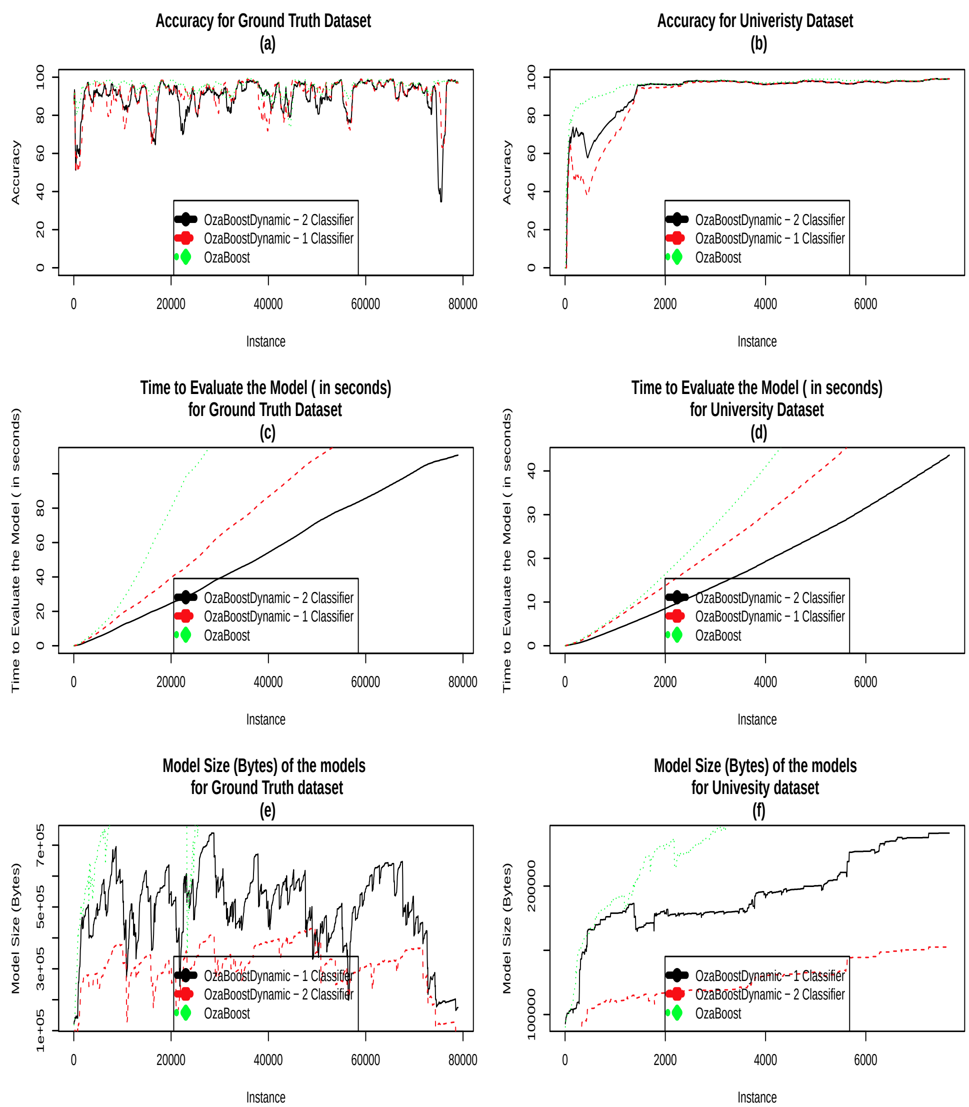


Figure 5.4: This figure presents the performance of OzaDyn with one classifier (OzaBoostDynamic - 1 Classifier) and with two classifiers (OzaBoostDynamic - 2 Classifier) compared to original OzaBoost for the network data sets. Figures (a) and (b) compare the accuracy of the algorithms, figures (c) and (d) compare the evaluation time, and (e) and (f) present the model size.

Chapter 6

Conclusions

This work introduced two key modifications to AdaBoost.M1 with four main objectives. The first objective was to reduce the AdaBoost.M1 dependency on a unique learner, which requires the user to test various classifiers to determine which offers the best result. This was achieved by allowing a list of learners be used as input, rather than a single learner. The second objective was to change the weight calculation to improve AdaBoost.M1 by guaranteeing that it will not overfit when the algorithm uses a learner with an error rate close to zero. Weight calculation is where the AdaBoost algorithm has theoretical performance guarantees, and this work also provides a mathematical proof of the proposed algorithms. The third objective was to increase the performance of the proposed algorithms with respect to noisy data sets. And the final objective was to use the modifications on algorithms for data streams, since AdaBoost is considered to be very slow for data stream learning, and its current implementations have theoretical limitations.

The first modification considered a different classifier in each iteration, independent of the performance with the training distribution, and used the same loss function currently used by AdaBoost. The algorithm, known as AdaBoost Dynamic (AD), maintained the same performance as AdaBoost using different weak learners, and the results provided as a baseline for further improvements. An unresolved question about this implementation was whether the AD algorithm could be adapted to search for classifier with smaller error rate in a given iteration.

The new algorithm, called AdaBoost Dynamic with Exponential Loss (AB-EL), has one modification: a search for the classifier that offers the smallest error rate in each iteration. Exponential loss function is still employed. As expected, the results were inferior, because AdaBoost does not perform well when boosted learners are strong learners. This raised another question: whether to consider both the search for the best algorithm and a different loss function, in order to improve the final hypothesis.

The third modification involves the use of the sigmoid function to update the weights, and is known as AdaBoost Dynamic with Logistic function (AB-DL). The decision to use the sigmoid function is based on its use to estimate the probability of an instance to belong to a specific label, as presented in [8]. The decision to use the derivative of the sigmoid function as the minimizing function, as well as resampling, resulted in the AB-DL performing better than previous versions of AdaBoost Dynamic. Theoretical investigation about this weight modification hypothesizes that AB-DL has an upper bound error defined.

This work addresses one of the main problems described by Freund: the dependency of AdaBoost on the weak learner. The ability to change learners, even when looking for the best classifier, eliminates this dependency.

The two major problems with AB-DL are that it executes a search for the "best" learner and that this reduces the diversity among the classifiers. The diversity reduction is explained by the fact that all the learners used had equal performances, with only small differences in errors. These issues do not produce more improvement in AB-DL performance, so the decision was made to apply the initial learner execution policy from AB-D, but using the weight calculation strategy employed by AB-DL. This combination created the AdaBoost Dynamic with Added Cost (AD-AC).

AD-AC also uses the lessons learned from AdaBoost.M2, in its weight calculation, by incorporating the learner confidence to calculate the misclassified instance weights. Since the weights were changed, this work also provides the theoretical convergence of AD-AC. These changes increase the average accuracy performance of AD-AC relative to all AdaBoost.M1 algorithms tested, and in relation to AB-DL when compared using the T-Test. However, tests executed with ANOVA and Bonferroni did not show any statistical differences between the algorithms, despite the fact that the AD-AC average accuracy performance was better than all others.

Tests were also executed with same data sets with the addition of 10% noise. All the algorithms had reduced performance, but AD-AC maintained the highest average accuracy.

Another problem this work dealt with was the use of the AD-AC algorithm with data stream. OzaBoost is the original boosting algorithm used in data streams, and it estimates the weight of misclassified instances based on a Poisson distribution. Since

OzaBoost learns from one instance per time, and therefore does not have the option of multiple passes on the training data, it builds various classifiers in parallel. The reason for this learning process is that data streams cannot store the training data in memory, as it is of the order of millions of instances, thus the algorithms must create accurate models with memory constraints. Poisson distribution is used to estimate the weights of misclassified instances, but this has theoretical issues, because the estimate may not be representative of the true data set distribution.

The AD-AC learner execution was adapted to allow multiple classifiers to be used in OzaBoost, and its weight calculation was also adapted to OzaBoost because the error of each parallel classifier was directly calculated inside each iteration, instead of using the Poisson distribution. The new algorithm is called OzaBoost Dynamic (OzaDyn).

In order to better evaluate the effect of each modification of OzaBoost, two versions were implemented and tested, one using a single base classifier and the other using two base classifiers. This allowed scrutiny of how the algorithm performs with the sole modification of weight calculation. Another reason was to check how using more than one classifier improves different aspects of the OzaBoost algorithm.

These modifications were tested using two very large data sets generated by MOA software. Two tests were conducted with each data set without concept drift, and one test was done with artificial drift added to the Agrawal data set. Tests on the data sets without concept drift showed that OzaBoostDynamic, with one or two classifiers, is a significant improvement over OzaBoost, with respect to the time taken to evaluate the models and the model sizes. Considering the data set with concept drift, OzaBoostDynamic with one classifier had inferior performance to OzaBoost in terms of evaluation time, but OzaBoostDynamic with two classifiers had better performance. This is because

a drift in the data set tends to increase both model size and evaluation time, and the combination of a simple model and a complex model help keep the memory consumption and processing times below those of OzaBoost.

6.1 Future Work

This work targeted an old boosting problem of dependency on the learner chosen to be boosted. Another related problem is the boosting dependency on the loss function adapted to separate the correctly classified and misclassified instances. As presented Chapter 3, there are various loss functions, and each has advantages and disadvantages. To advance the research, it will be necessary to create a policy to combine loss functions depending on the data set. This will impact boosting theory, since there will be a strong need to show that combined functions also generate models that will converge.

Similarly, it would also be possible to design an algorithm that can deal with unbalanced data sets, by changing the loss function. A traditional approach used requires the user to define a cost matrix, such as AdaCost, that will make the algorithm increase the weights of misclassifications done in the minority class so they become higher than those done in the majority class. RareBoost tries to find the minority class weights by calculating the relationship between false and true positives, but it still depends on the exponential loss. The loss function variation would allow use of a loss function that increases the weights for misclassifications with instances belonging to the minority class, and a different loss function that reduces the weights for instance misclassifications belonging to the majority class.

Another application of the weight adaptation is related to the use of different weights to improve quantify misclassifications error types. As an example, in medical problems, the cost of an erroneous diagnosis of cancer in a patient is much higher than the cost of declaring a patient with cancer as halthy. The use of multiple types of loss functions may help in this process, since, during training phase, it is possible to use a different loss function based on the type of error committed by the “weak” learner.

More experiments must also be done with real stream data sets. Results in this work show that OzaBoost Dynamic accuracy performance is equal to that of OzaBoost. However, the data sets tested were either synthetic, as generated by MOA, or they did not have enough data to be considered data streams. The use of MOA ensures that results can be reproduced, but it does not guarantee that the performance will be the same with real data.

Appendix A

Proofs and Demonstrations

A.1 Proof of Equality 3.7

Equality 3.7 is defined as

$$\frac{1}{e^{-(1-\epsilon_t)\,\epsilon_t}+1} = \frac{e^{\epsilon_t}}{e^{\epsilon_t^2}+e^{\epsilon_t}}$$

then developing the left hand side, we find

$$\begin{aligned}\frac{1}{e^{-(1-\epsilon_t)\,\epsilon_t}+1} \qquad &= \frac{1}{\frac{1}{e^{(1-\epsilon_t)\epsilon_t}}+1} = \frac{1}{\frac{1}{e^{\epsilon_t-\epsilon_t^2}}+1} = \frac{1}{\frac{e^{\epsilon_t-\epsilon_t^2}+1}{e^{\epsilon_t-\epsilon_t^2}}} = \frac{e^{\epsilon_t-\epsilon_t^2}}{e^{\epsilon_t-\epsilon_t^2}+1} = \frac{\frac{e^{\epsilon_t}}{e^{\epsilon_t^2}}}{\frac{e^{\epsilon_t}}{e^{\epsilon_t^2}}+1} \\ &= \frac{\frac{e^{\epsilon_t}}{e^{\epsilon_t^2}}}{\frac{e^{\epsilon_t}+e^{\epsilon_t^2}}{e^{\epsilon_t^2}}} = \frac{e^{\epsilon_t}}{e^{\epsilon_t}+e^{\epsilon_t^2}}, \text{ which proves the equality.}\end{aligned} \tag{A.1}$$

A.2 Conjecture of Inequality 3.14

Inequality 3.14 has the following form

$$\sum_{t=1}^{T}((1-\epsilon_t)\epsilon_t) > \prod_{t=1}^{T}(\frac{1}{1+e^{-(1-\epsilon_t)\epsilon_t}})^{|h_t(x_i)-y_i|}$$

Developing the left hand side we find

$$\sum_{t=1}^{T}\epsilon_t - \epsilon_t^2 > \prod_{t=1}^{T}(\frac{1}{1+e^{-(1-\epsilon_t)\epsilon_t}})^{|h_t(x_i)-y_i|}$$

Assuming that $|h_t(x_i) - y_i| = 1$, where this is the case when $h_t(x_i) \neq y_i$, and developing the right hand side we have

$$\begin{aligned}
\prod_{t=1}^{T}\left(\frac{1}{1+e^{-(1-\epsilon_t)\epsilon_t}}\right)^{|h_t(x_i)-y_i|} &= \prod_{t=1}^{T}\left(\frac{1}{1+e^{-(1-\epsilon_t)\epsilon_t}}\right) = \\
\prod_{t=1}^{T}\left(\frac{e^{(1-\epsilon_t)\epsilon_t}}{1+e^{(1-\epsilon_t)\epsilon_t}}\right) &= \prod_{t=1}^{T}\left(\frac{e^{\epsilon_t-\epsilon_t^2}}{1+e^{\epsilon_t-\epsilon_t}}\right) = \\
\prod_{t=1}^{T}\left(\frac{\frac{e_t^{\epsilon}}{e^{\epsilon_t^2}}}{1+\frac{e_t^{\epsilon}}{e^{\epsilon_t^2}}}\right) &= \prod_{t=1}^{T}\left(\frac{\frac{e_t^{\epsilon}}{e^{\epsilon_t^2}}}{\frac{e^{\epsilon_t^2}+e^{\epsilon_t}}{e^{\epsilon_t^2}}}\right) = \\
\prod_{t=1}^{T}\left(\frac{e^{\epsilon_t}}{e^{\epsilon_t^2}+e^{\epsilon_t}}\right) & \qquad \text{(A.2)}
\end{aligned}$$

Then inequality 3.14 becomes

$$\sum_{t=1}^{T}\epsilon_t - \epsilon_t^2 > \prod_{t=1}^{T}\left(\frac{e^{\epsilon_t}}{e^{\epsilon_t^2}+e^{\epsilon_t}}\right) \tag{A.3}$$

In order to show that the left hand side in (A.3) bounds the right hand side we assume that the ϵ_t has a fixed value for any number of iterations T, which means that the error rate will be the same for all iterations, and it is also defined as $0 < \epsilon_t \leq \frac{1}{2}$. Then, we find

$$T(\epsilon_t - \epsilon_t^2) > \left(\frac{e^{\epsilon_t}}{e^{\epsilon_t^2}+e^{\epsilon_t}}\right)^T \tag{A.4}$$

Plotting all the values for both expressions in Inequality A.4, $0 < \epsilon_t \leq \frac{1}{2}$ and $T = 100$, it is possible to see that the left hand side is bigger than the right hand side, as shown in Figure A.1. Because the values for left hand side are very big, for any ϵ_t and T assumed, and with the objective to show both graphs in the same figure, left hand side values were reduced by a factor of 10^{25}.

A.3 Proof of Inequality 3.27

Inequality 3.27 has the following form

$$\sum_{t=1}^{T}(1 - ((1-\epsilon_t)\epsilon_t)) > \prod_{t=1}^{T}\left(\frac{1}{1+e^{-((1-(1-\epsilon_t)\epsilon_t)^{\alpha}}}\right)^{|h_t(x_i)-y_i|}$$

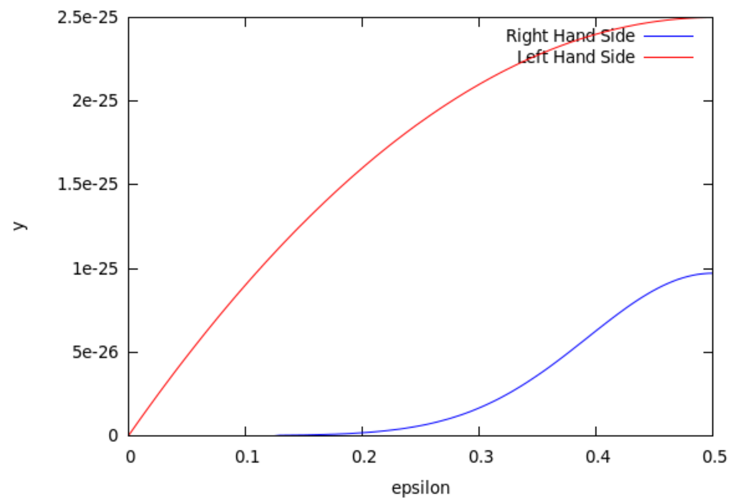


Figure A.1: Bound for AB-DL inequality.

then it is possible to rewrite the left hand side as

$$\sum_{t=1}^{T}(1-((1-\epsilon_t)\epsilon_t)) = \sum_{t=1}^{T}(1-(\epsilon_t-\epsilon_t^2)) = \sum_{t=1}^{T} 1-\epsilon_t+\epsilon_t^2 \tag{A.5}$$

Assuming that $|h_t(x_i)-y_i| = 1$, where this is the case when $h_t(x_i) \neq y_i$, and developing the right hand side we have

$$\prod_{t=1}^{T}\left(\frac{1}{1+e^{-((1-(1-\epsilon_t)\epsilon_t)^{\alpha}}}\right) = \prod_{t=1}^{T}\left(\frac{1}{1+e^{-\left(1-\epsilon_t+\epsilon_t^2\right)^{\alpha}}}\right) = \prod_{t=1}^{T}\left(\frac{e^{\left(1-\epsilon_t+\epsilon_t^2\right)^{\alpha}}}{1+e^{\left(1-\epsilon_t+\epsilon_t^2\right)^{\alpha}}}\right) \tag{A.6}$$

In order to show that the left hand side in (A.5) bounds the right hand side we assume that the ϵ_t has the same error rate for any number of iterations T, also considering that $0 \leq \epsilon_t \leq \frac{1}{2}$, and any arbitrary value for α, then, we find

$$T(1-\epsilon_t+\epsilon_t^2) > \left(\frac{e^{\left(1-\epsilon_t+\epsilon_t^2\right)^{\alpha}}}{1+e^{\left(1-\epsilon_t+\epsilon_t^2\right)^{\alpha}}}\right)^{T} \tag{A.7}$$

After the transformations, it is necessary to show that both sides are monotonic. Calculating the first derivative, assuming $\alpha = 1$ and $T = 1$ for simplicity, we have

- Left hand side:
$$\frac{\mathrm{d}}{\mathrm{d}\epsilon_t}\left(1-\epsilon_t+\epsilon_t^2\right) = 2\,\epsilon_t - 1$$

- Right hand side:

$$\frac{\mathrm{d}}{\mathrm{d}\epsilon_t}\left(\frac{e^{\left(1-\epsilon_t+\epsilon_t^2\right)}}{1+e^{\left(1-\epsilon_t+\epsilon_t^2\right)}}\right) = \frac{\left(2\,\epsilon_t - 1\right)\, e^{\epsilon_t^2-\epsilon_t+1}}{e^{\epsilon_t^2-\epsilon_t+1}+1} - \frac{\left(2\,\epsilon_t - 1\right)\, e^{2\,\epsilon_t^2-2\,\epsilon_t+2}}{\left(e^{\epsilon_t^2-\epsilon_t+1}+1\right)^2}$$

Both expressions will be negative when $\epsilon_t < \frac{1}{2}$, and equal when $\epsilon_t = \frac{1}{2}$. Figure A.2 shows the behaviour of these derivatives. Both sides are considered monotonic and increasing on the interval where $\epsilon_t \in [0, \frac{1}{2}]$.

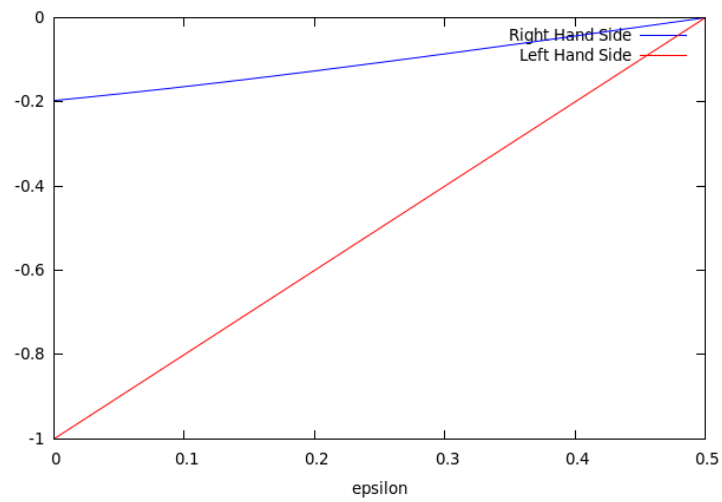


Figure A.2: Plot of the derivatives of inequality (A.7).

Using the fact that both expressions in (A.7) are monotonic, we continue the proof by showing that for any big value of T, the inequality (A.7) remains true. Assuming $T \to \infty$, and testing this when $\epsilon_t = 0$ and $\epsilon_t = 0.5$, we have

- $\epsilon_t = 0 : T > \left(\frac{e}{1+e}\right)^T$, then using the limit of $T \to \infty$ in both sides of (A.7) $\infty > 0$ [1] (Valid)

- $\epsilon_t = 0.5 : T(1 - 0.5 + 0.5^2) > \left(\frac{e^{(1-0.5+0.5^2)}}{1+e^{(1-0.5+0.5^2)}}\right)^T$, then using the limit of $T \to \infty$, in both sides, we will have in (A.7) $\infty > 0$ [2](Valid)

The fact that the expressions on both sides of (A.7) are monotonic, and that values at the end of the interval satisfy the inequality (A.7) implies that the inequality holds for all values in the interval. □

[1]The $\lim_{T\to 0}\left(\frac{e}{1+e}\right)^T = 0$ was found in [45]

[2]The $\lim_{T\to 0}\left(\frac{e^{(1-0.5+0.5^2)}}{1+e^{(1-0.5+0.5^2)}}\right)^T = 0$ was found in [46]

Plotting all the values for both expressions in Inequality A.7, $0 \leq \epsilon_t \leq \frac{1}{2}$, $T = 100$ and $\alpha = 0.35$[3], it is possible to see that the left hand side is bigger than the right hand side, as shown in Figure A.1. Because the values for left hand side are very big, for any ϵ_t and T assumed, and with the objective to show both graphs in the same figure, left hand side values were reduced by a factor of 10^{15}.

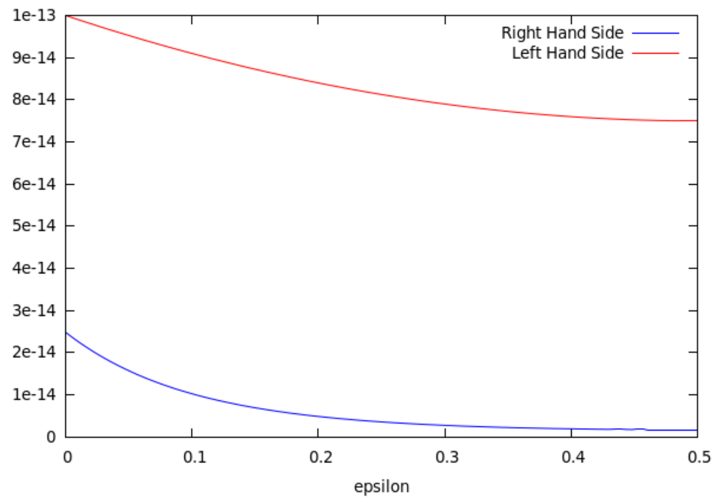


Figure A.3: Bound for AD-AC inequality.

[3] This value of α was used, because it was also used in experiments of Chapter 4.

Appendix B

Source Code of the Algorithms

B.1 AB-DL

Table B.1 shows the declaration of the AB-DL's `buildClassifierWithWeights`. In Weka, this method is responsible for training the classifier. The most important variables are `m_Betas` that store the values of the β_t, `m_NumIterationsPerformed` that controls the iterations T, the array of weights (`double [] weight`) for the decision about what classifier has the best performance, and the array of epsilons (`double [] epsilons`), which store the error rate of all classifiers during the decision. Next, in Table B.2 main algorithm is presented, where all algorithms are tested and build for all iterations. All performances are stored in previous two arrays and the search for the best learner is done with support of them. Line 13 in Table B.2 shows how the weight is calculated, by increasing the weight for a classifier with low error rate. Associated with this information, the algorithm also stores the error rate (line 15) for all classfiers executed. These steps are associated with lines 8 until 10 in the algorithm presented in Table 3.4. The last part of the code is presented in Table B.3, which shows the search for the algorithm with smallest error rate (from lines 1 to 8), and the calculation of the β_t according to the weight Equation (3.2) (line 10). The next part only calculates the weights for the misclassified instances when the method `setWeights_Mod` is called. Table B.4 presents `setWeights_Mod` that updates the weights according to Equation (3.3), and accepts as inputs the training data, the weight calculation (β_t) and the index of the best classifier.

```
protected void buildClassifierWithWeights(Instances data)
  throws Exception {
  Instances trainData, training;
  double epsilon, reweight, reweight_curr_log;
  boolean errorzero = false;
  Evaluation evaluation;
  int numInstances = data.numInstances();
  int count = 0;
  int current_classifier = 0;
  epsilon = 0;
  reweight = 0;
  double [] weight = new double[mclass.length];
  double [] epsilons = new double[mclass.length];
  // Initialize data
  m_Betas = new double [m_NumIterations];
  m_NumIterationsPerformed = 0;
  choosen_classifier = new int[m_NumIterations];
  // Create a copy of the data so that when the weights are diddled
  // with it doesn't mess up the weights for anyone else
  training = new Instances(data, 0, numInstances);
  // Do boostrap iterations
```

Table B.1: Header of the method AB-DL's `buildClassifierWithWeights`.

B.2 AD-AC

Tables B.5 and B.6 show the training implementation of AD-AC. The implementation follows the same steps presented in Table 3.5, then in last step it calls the method `setWeights_Mod` to update the weights of all instances. The algorithm executes a different classifier, which is stored in the `mclass` array (as presented in Table B.6), then evaluates each one to calculate their respective error rate. The program also stores in the array `list_class` which iteration executed a certain learner. This last information is important to calculate the output of the algorithm.

Table B.7 shows the reweight code for AD-AC algorithm. The source code is equal to the algorithm presented in Table 3.5. In order to improve the algorithm speed, line 6 shows that the algorithm tests if the classifer was able to correctly classify the instance, and stores the probability associated with each of the classes of the data set. The rest of the algorithm does what was already described in Table 3.5.

B.3 Common AdaBoost Dynamic Output

Table B.8 shows how the output of all AdaBoost Dynamic implementations was done. Since all of the sums are previously calculated and stored in array `mclass` during training, Table B.8 only calculates the final sum (lines 11 until 13), and executes a conversion from a sum to an array of probabilities where each component is associated with a class present in the data set (line 14), which uses a method implemented in other Weka class. This last step is not mandatory, and the developer may output the array `sums` directly.

B.4 OzaBoostDynamic

OzaBoostDynamic was implemented in MOA package, that is derived from Weka. The code is presented in Java, and the only difference between OzaBoostDynamic implementation and traditional OzaBoost in MOA is that the first has the following declaration `protected Classifier baseLearner1, baseLearner2`. This will store the configuration of the two *different* classifiers chosen by the user.

Table B.9 presents the source code implemented for OzaDyn with two classifiers. The algorithm initiates the ensemble with each of the chosen classifiers by the user. In this case, Hoeffding tree and Naive Bayes Multinomial. The size of the ensemble is the same as the number of iterations defined by the user. The loop, between lines 7 and 10 will create a copy of each learner inside the array of ensemble. Lines 12 until 24 present the configuration used by `ADWIN` algorithm, which is present at the MOA package.

Table B.10 shows the piece of code responsible for training OzaBoostDynamic. The missing code parts are related with the `ADWIN` algorithm, and the storage of the statistics about the algorithm performance. The code presented does exactly what AD-AC algorithm does, by executing a different learner for each iteration, and calculates the weights with the modified sigmoid function presented in Table 3.5. The last part of the algorithm does what was described in Table 5.2. This implementation has a small change in relation to the algorithms in Tables 3.5 and 5.2, by the initialization of the variable `em=0.99`, because Java language demands that the variable must be initalized before its use.

Table B.11 shows the method that can be called at any moment and present the output of the learning step. As in AD-AC, the algorithm executes a test to verify if the error is equal to zero or bigger than $\frac{1}{2}$. If it is, the algorithm only returns the value of the accumulation of the previous iterations (the `logKm1` variable). If it is not, the algorithm adds the latest weight calculation based on the AD-AC output plus the values accumulated in previous iterations.

The prediction method is not presented in this section because it did not suffer any modifications.

```
    for (m_NumIterationsPerformed = 0;
         m_NumIterationsPerformed < m_NumIterations; m_NumIterationsPerformed++) {
    // Select instances to train the classifier on
      if (m_WeightThreshold < 100) {
trainData = selectWeightQuantile(training,
 (double)m_WeightThreshold / 100);
      } else {
trainData = new Instances(training, 0, numInstances);
      }
    // Build the classifier all classifiers, and store their
    // performances in an array
     for ( count = 0; count<mclass.length; count++){
        mclass[count].buildClassifier(trainData);
      // Evaluate the classifier
        evaluation = new Evaluation(data);
        evaluation.evaluateModel(mclass[count], training);
        epsilon = evaluation.errorRate();
      // Checks to if the classifer has epsilon 0<epsilon<0.5
      // if it is, then
        if (Utils.sm(epsilon, 0.5) && !(Utils.eq(epsilon, 0))){
           reweight_curr_log = (1 - epsilon) / epsilon ;
           weight[count] = reweight_curr_log;
           epsilons[count] = epsilon;
        } else weight[count] = 0;
        epsilon = 0;
        reweight_curr_log =0;
      }//end for
```

Table B.2: Continuation of the method AB-DL's `buildClassifierWithWeights`.

```
      // Stop if error too small or error too big and ignore this model
      if (errorzero!=true){
        current_classifier = getMaxWeight(weight);
        epsilon = epsilons[current_classifier];
      }else{
        System.out.println("Error = "+epsilon+" Classifier="+current_classifier);
        epsilon = 0;
        errorzero=false;
      }
      choosen_classifier[m_NumIterationsPerformed] = current_classifier;
       m_Betas[m_NumIterationsPerformed] = (1 - epsilon) * epsilon;
       reweight =m_Betas[m_NumIterationsPerformed];

    // Determine the weight to assign to this model
       if (m_Debug) {
          System.err.println("\terror rate = " + epsilon
              +"  beta = " + m_Betas[m_NumIterationsPerformed]);
       }
     // Update instance weights
        setWeights_Mod(training, reweight, current_classifier);
        epsilon=0;
      }//end for
    }//end of the method
```

Table B.3: Final of the method AB-DL's `buildClassifierWithWeights`.

```
//Parameter count indicates which classifer had the best performance.
private void setWeights_Mod(Instances training, double reweight, int count)
  throws Exception {
  double oldSumOfWeights, newSumOfWeights;
  oldSumOfWeights = training.sumOfWeights();
  Enumeration enu = training.enumerateInstances();
  while (enu.hasMoreElements()) {
    Instance instance = (Instance) enu.nextElement();
    double result = mclass[count].classifyInstance(instance);
    double[] dist = mclass[count].distributionForInstance(instance);
    if (!Utils.eq(result,
  instance.classValue()))
      instance.setWeight(instance.weight()*(1/(1+Math.exp(-reweight))));
  }

  // Renormalize weights
  newSumOfWeights = training.sumOfWeights();
  enu = training.enumerateInstances();
  while (enu.hasMoreElements()) {
    Instance instance = (Instance) enu.nextElement();
    instance.setWeight(instance.weight() * oldSumOfWeights
                         / newSumOfWeights);
  }
}
```

Table B.4: AB-DL weight calculation (method `setWeights_Mod`).

```
protected void buildClassifierWithWeights(Instances data)
    throws Exception {

    Instances trainData, training;
    double epsilon=0.6, reweight;
    Evaluation evaluation;
    int numInstances = data.numInstances();
    Random randomInstance = new Random(m_Seed);
    int log = 0;
    int size = mclass.length;
    int count = 0;
    list_class =  new int[m_NumIterations];
    Random generator = new Random();
    // Initialize data
    m_Betas = new double [m_NumIterations];
    m_NumIterationsPerformed = 0;

    // Create a copy of the data so that when the weights are diddled
    // with it doesn't mess up the weights for anyone else
    training = new Instances(data, 0, numInstances);
    count  = getAlgorithm(training, generator, numInstances);
    // Do boostrap iterations
    for (m_NumIterationsPerformed = 0; m_NumIterationsPerformed < m_NumIterations;
 m_NumIterationsPerformed++) {

      if (m_Debug) {
System.err.println("Training classifier " + (m_NumIterationsPerformed + 1));
      }
      // Select instances to train the classifier on
      if (m_WeightThreshold < 100) {
trainData = selectWeightQuantile(training,
 (double)m_WeightThreshold / 100);
      } else {
trainData = new Instances(training, 0, numInstances);
      }
```

Table B.5: AD-AC training program (of method `buildClassifierWithWeights`).

```
      // Build the classifier
      if ( count >= size)  count = 0;
       if (mclass[count] instanceof Randomizable)
 ((Randomizable) mclass[count]).setSeed(randomInstance.nextInt());
         mclass[count].buildClassifier(trainData);
       // Evaluate the classifier
       evaluation = new Evaluation(data);
       evaluation.evaluateModel(mclass[count], training);
       epsilon = evaluation.errorRate();
       if (m_Debug) {
 System.err.println("\terror rate = " + epsilon
    +"  beta = " + m_Betas[m_NumIterationsPerformed] +
             " count = "+ count);
       }
      list_class[m_NumIterationsPerformed] = count;
       // Stop if error too small or error too big and ignore this model
       if (Utils.grOrEq(epsilon, 0.5) || Utils.eq(epsilon, 0)) {
 if (m_NumIterationsPerformed == 0) {
   m_NumIterationsPerformed = 1; // If we're the first we have to to use it
 }
 break;
       }
       // Determine the weight to assign to this model
       m_Betas[m_NumIterationsPerformed] =  1-((1 - epsilon) * epsilon);
       reweight =  1-((1 - epsilon) * epsilon);
       log = 1;
        if (m_Debug) {
 System.err.println("\terror rate = " + epsilon
    +"  beta = " + m_Betas[m_NumIterationsPerformed]);
       }
       // Update instance weights
       setWeights_Mod(training, reweight, count);

       count++;
     }
   }
```

Table B.6: AD-AC training program (continuation of method `buildClassifierWithWeights`).

```
  protected void setWeights_Mod(Instances training,
                                double reweight,
                                int classfier)
    throws Exception {
     double oldSumOfWeights, newSumOfWeights;
     oldSumOfWeights = training.sumOfWeights();
     Enumeration enu = training.enumerateInstances();
     while (enu.hasMoreElements()) {
       Instance instance = (Instance) enu.nextElement();
     double[] temp = mclass[classfier].distributionForInstance(instance);
       if (!Utils.eq(mclass[classfier].classifyInstance(instance),
       instance.classValue())){
           instance.setWeight(instance.weight() *
                            ((1/1+Math.exp(-Math.pow(
                                reweight,
                                (((1 -
                  temp[(int)mclass[classfier].classifyInstance(instance)])
                                 + temp[(int)instance.classValue()] ))*
                                 (1-learn_rate)) ))));
        }
      }
    // Renormalize weights
      newSumOfWeights = training.sumOfWeights();
      enu = training.enumerateInstances();
      while (enu.hasMoreElements()) {
        Instance instance = (Instance) enu.nextElement();
        instance.setWeight(instance.weight() * oldSumOfWeights
     / newSumOfWeights);
      }
    }
```

Table B.7: AD-AC's weight calculation (Method setWeights_Mod).

```
public double [] distributionForInstance(Instance instance)
  throws Exception {
  // default model?
    if (m_ZeroR != null) {
      return m_ZeroR.distributionForInstance(instance);
    }

    if (m_NumIterationsPerformed == 0) {
      throw new Exception("No model built");
    }
    double [] sums = new double [instance.numClasses()];

    if (m_NumIterationsPerformed == 1) {
      return mclass[0].distributionForInstance(instance);
    } else {
      for (int i = 0; i < m_NumIterationsPerformed; i++) {
        sums[(int)mclass[list_class[i]].classifyInstance(instance)] +=
            m_Betas[i];
      }
      return Utils.logs2probs(sums);
    }
  }
```

Table B.8: Implementation of the final output for all AdaBoost Dynamic versions (Method `distributionForInstance`).

```
public void resetLearningImpl() {
    this.ensemble = new Classifier[this.ensembleSizeOption.getValue()];
    baseLearner1 = (Classifier)
                     getPreparedClassOption(this.baseLearnerOption);
    baseLearner2 = new moa.classifiers.bayes.NaiveBayesMultinomial();
    baseLearner1.resetLearning();
    baseLearner2.resetLearning();
    for (int i = 0; i < this.ensemble.length; i++) {
        if (i%2==0)
             this.ensemble[i] = baseLearner1.copy();
        else this.ensemble[i] = baseLearner2.copy();
    }
    this.scms = new double[this.ensemble.length];
    this.swms = new double[this.ensemble.length];
    this.ADError = new ADWIN[this.ensemble.length];
    for (int i = 0; i < this.ensemble.length; i++) {
        this.ADError[i] = new ADWIN((double)
                            this.deltaAdwinOption.getValue());
    }
    this.numberOfChangesDetected = 0;
    if (this.outputCodesOption.isSet()) {
        this.initMatrixCodes = true;
    }
    if (this.sammeOption.isSet()) {
        this.initKm1 = true;
    }
}
```

Table B.9: OzaBoostDynamic's `resetLearning`, as implemented in MOA package.

```
public void trainOnInstanceImpl(Instance inst) {
. . .
  boolean Change = false;
  double lambda_d = 1.0;
  Instance weightedInst = (Instance) inst.copy();
  double em = 0.99;
  for (int i = 0; i < this.ensemble.length; i++) {
       if (this.outputCodesOption.isSet()) {
             weightedInst.setClassValue((double)
                 this.matrixCodes[i][(int) inst.classValue()]);
       }
       weightedInst.setWeight(
            inst.weight() * (1/(1+Math.exp(-Math.pow((1-(1-em)*em),0.9)))));
       this.ensemble[i].trainOnInstance(weightedInst);
       boolean correctlyClassifies =
             this.ensemble[i].correctlyClassifies(weightedInst);
       if (correctlyClassifies) {
           this.scms[i] += lambda_d;
           lambda_d *= this.trainingWeightSeenByModel / (2 * this.scms[i]);
       } else {
           this.swms[i] += lambda_d;
           lambda_d *= this.trainingWeightSeenByModel / (2 * this.swms[i]);
           em = this.swms[i] / (this.scms[i] + this.swms[i]);
       }
       . . .
     }
. . .
```

Table B.10: OzaBoostDynamic's `trainOnInstanceImpl`, as implemented in MOA package. This example assumes $\phi = 0.9$.

```
protected double getEnsembleMemberWeight(int i) {
    double em = this.swms[i] / (this.scms[i] + this.swms[i]);
    if ((em == 0.0) || (em > 0.5)) {
        return this.logKm1;
    }
    return (1-((1.0 - em) * em)) + this.logKm1;
}
```

Table B.11: OzaBoostDynamic's `getEnsembleMemberWeight`, as implemented in MOA package.

Bibliography

[1] Eric Bauer and Ron Kohavi. An empirical comparison of voting classification algorithms: Bagging, boosting, and variants. *Mach. Learn.*, 36:105–139, July 1999.

[2] Albert Bifet and Ricard Gavaldà. Learning from time-changing data with adaptive windowing. In *SDM*. SIAM, 2007.

[3] Albert Bifet, Geoff Holmes, Richard Kirkby, and Bernhard Pfahringer. Data stream mining: a practical approach. Technical report, University of Waikato, May 2011.

[4] Marco Botta. Resampling vs reweighting in boosting a relational weak learner. In Floriana Esposito, editor, *AI*IA 2001: Advances in Artificial Intelligence*, volume 2175 of *Lecture Notes in Computer Science*, pages 70–80. Springer Berlin / Heidelberg, 2001.

[5] Leo Breiman. Bagging predictors. *Mach. Learn.*, 24:123–140, August 1996.

[6] Nitesh V. Chawla, Kevin W. Bowyer, Lawrence O. Hall, and W. Philip Kegelmeyer. Smote: synthetic minority over-sampling technique. *J. Artif. Int. Res.*, 16:321–357, June 2002.

[7] Nitesh V. Chawla, Aleksandar Lazarevic, Lawrence O. Hall, and Kevin W. Bowyer. Smoteboost: Improving prediction of the minority class in boosting. In Nada Lavrac, Dragan Gamberger, Hendrik Blockeel, and Ljupco Todorovski, editors, *PKDD*, volume 2838 of *Lecture Notes in Computer Science*, pages 107–119. Springer, 2003.

[8] Michael Collins, Robert E. Schapire, and Yoram Singer. Logistic regression, adaboost and bregman distances. *Machine Learning*, 48:253–285, 2002.

[9] Pedro Domingos and Geoff Hulten. Mining high-speed data streams. In *KDD*, pages 71–80, 2000.

[10] Richard O. Duda, Peter E. Hart, and David G. Stork. *Pattern Classification (2nd Edition)*. Wiley-Interscience, 2000.

[11] Charles Elkan. The foundations of cost-sensitive learning. In *Proceedings of the 17th international joint conference on Artificial intelligence - Volume 2*, pages 973–978, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.

[12] Wei Fan, Salvatore J. Stolfo, Junxin Zhang, and Philip K. Chan. Adacost: Misclassification cost-sensitive boosting. In *Proceedings of the Sixteenth International Conference on Machine Learning*, ICML '99, pages 97–105, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

[13] Stenio Fernandes. Private Communication.

[14] A. Frank and A. Asuncion. UCI machine learning repository, 2010.

[15] Y. Freund and R. Schapire. A short introduction to boosting. *Japonese Society for Artificial Intelligence*, 14(5):771–780, 1999.

[16] Y. Freund and R. E. Schapire. Experiments with a new boosting algorithm. *L. Saitta, Ed., Proceedings of the 13th International Conference on Machine Learning*, pages 148–156, 1996.

[17] Yoav Freund and Robert E. Schapire. Game theory, on-line prediction and boosting. In Avrim Blum and Michael Kearns, editors, *COLT*, pages 325–332. ACM, 1996.

[18] Yoav Freund and Robert E. Schapire. A decision-theoretic generalization of on-line learning and an application to boosting. *Journal of Computer and System Sciences*, 55(1):119–139, 1997.

[19] J. Friedman, T. Hastie, and R. Tibshirani. Additive logistic regression: a statistical view of boosting. *Annals of Statistics*, 28(2):337–407, 2000a.

[20] Joao Gama and Rasmus Ulslev Pedersen. *Learning from Data Streams: Processing Techniques in Sensor Networks. Chapter 10: Predictive Learning in Sensor Networks*. Springer, November 2007.

[21] Abbas Golestani, Kushan Ahmadian, Ali Amiri, and MohammadReza JahedMotlagh. A novel adaptive-boost-based strategy for combining classifiers using diversity concept. *Computer and Information Science, ACIS International Conference on*, 0:128–134, 2007.

[22] F. Gringoli, Luca Salgarelli, M. Dusi, N. Cascarano, F. Risso, and k. c. claffy. Gt: picking up the truth from the ground for internet traffic. *SIGCOMM Comput. Commun. Rev.*, 39:12–18, October 2009.

[23] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H. Witten. The weka data mining software: An update. *SIGKDD Explorations*, 11(1), 2009.

[24] Jiawei Han. *Data Mining: Concepts and Techniques*. Morgan Kaufmann Publishers, 2006.

[25] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer Science, 2009.

[26] Haibo He, Sheng Chen, Hong Man, Sachi Desai, and Shafik Quoraishee. Imbalanced learning for pattern recognition: an empirical study. volume 7833, page 78330T. SPIE, 2010.

[27] Haibo He and Edwardo A. Garcia. Learning from imbalanced data. *IEEE Transactions on Knowledge and Data Engineering*, 21:1263–1284, 2009.

[28] Klaus-Uwe Höffgen and Hans Ulrich Simon. Robust trainability of single neurons. In *Proceedings of the fifth annual workshop on Computational learning theory*, COLT '92, pages 428–439, New York, NY, USA, 1992. ACM.

[29] Nathalie Japkowicz and Mohak Shah. *Evaluating Learning Algorithms: A Classification Perspective*. Cambridge University Press, New York, NY, USA, 2011.

[30] Mhesh V. Joshi, Vipin Kumar, and Ramesh Agarwal. Evaluating boosting algorithms to classify rare classes: Comparison and improvements. *Data Mining, IEEE International Conference on*, 1:257–265, 2001.

[31] Ron Kohavi. Scaling up the accuracy of naive-bayes classifiers: A decision-tree hybrid. *Second International Conference on Knoledge Discovery and Data Mining*, pages 202–207, 1996.

[32] Ludmila I. Kuncheva. *Combining Pattern Classifiers: Methods and Algorithms*. Wiley-Interscience, 2004.

[33] Llew Mason, Jonathan Baxter, Peter Bartlett, and Marcus Frean. Boosting algorithms as gradient descent. *In Advances in Neural Information Processing Systems 12*, pages 512–518, 2000.

[34] Thomas M. Mitchell. *Machine Learning*. McGraw-Hill, Inc., New York, NY, USA, 1 edition, 1997.

[35] Masayuki Nakamura, Hiroki Nomiya, and Kuniaki Uehara. Improvement of boosting algorithm by modifying the weighting rule. *Annals of Mathematics and Artificial Intelligence*, 41:95–109, May 2004.

[36] Nikunj C. Oza and Stuart Russell. Online bagging and boosting. In Tommi Jaakkola and Thomas Richardson, editors, *Eighth International Workshop on Artificial Intelligence and Statistics*, pages 105–112, Key West, Florida. USA, 2001. Morgan Kaufmann.

[37] Cynthia Rudin, Ingrid Daubechies, and Robert E. Schapire. On the dynamics of boosting. In Sebastian Thrun, Lawrence K. Saul, and Bernhard Schölkopf, editors, *NIPS*. MIT Press, 2003.

[38] Robert E. Schapire. The boosting approach to machine learning - an overview. *Proc. MSRI Workshop Nonlinear Estimation and Classification*, pages 149–172, march 2002.

[39] Chris Seiffert, Taghi M. Khoshgoftaar, Jason Van Hulse, and Amri Napolitano. Resampling or reweighting: A comparison of boosting implementations. In *Proceedings of the 2008 20th IEEE International Conference on Tools with Artificial Intelligence - Volume 01*, pages 445–451, Washington, DC, USA, 2008. IEEE Computer Society.

[40] Murat Soysal and Ece Guran Schmidt. Machine learning algorithms for accurate flow-based network traffic classification: Evaluation and comparison. *Performance Evaluation*, 67(6):451 – 467, 2010.

[41] Paul Viola and Michael Jones. Rapid object detection using a boosted cascade of simple features. *Computer Vision and Pattern Recognition, IEEE Computer Society Conference on*, 1:511, 2001.

[42] Andrew R. Webb. *Statistical Pattern Recognition, 2nd Edition*. John Wiley & Sons, October 2002.

[43] Geoffrey I. Webb. Multiboosting: A technique for combining boosting and wagging. *Machine Learning*, Vol.40(No.2), 2000.

[44] Ian H. Witten and Eibe Frank. *Data Mining: Practical Machine Learning Tools and Techniques*. The Morgan Kaufmann Series in Data Management Systems. Morgan Kaufmann Publishers, San Francisco, CA, 2nd edition, 2005.

[45] Wolfram—Alpha. Wolfram Alpha LLC. `http://www2.wolframalpha.com/input/?i=%28%28e%29%2F%281%2Be%29%29^K%2C+where+K%3Dinfinite`, 2014. [Online; accessed 10-January-2014].

[46] Wolfram—Alpha. Wolfram Alpha LLC. `http://www3.wolframalpha.com/input/?i=(((e^(1-0.5%2B(0.5)^2))%2F(1%2Be^(1-0.5%2B(0.5)^2))))^K%2C+where+K%3Dinfinite`, 2014. [Online; accessed 10-January-2014].

[47] A.J. Wyner. Boosting and the exponential loss. In C. M. Bishop and B. J. Frey, editors, *Proceedings of the Ninth International Workshop on Artificial Intelligence and Statistic*. C. M. Bishop and B. J. Frey, 2003.

[48] I-Cheng Yeh, King-Jang Yang, and Tao-Ming Ting. Knowledge discovery on rfm model using bernoulli sequence. *Expert Systems with Applications*, 36(3, Part 2):5866 – 5871, 2009.

[49] Guixi Zou, George Kesidis, and David J. Miller. A flow classifier with tamper-resistant features and an evaluation of its portability to new domains. *Selected Areas in Communications, IEEE Journal on*, 29(7):1449 –1460, august 2011.