


ML2R Coding Nuggets

AdaBoost with Pre-Trained Hypotheses

Christian Bauckhage 
Machine Learning Rhine-Ruhr
Fraunhofer IAIS
St. Augustin, Germany

ABSTRACT

In preparation for things to come, we discuss the general ideas behind AdaBoost (for binary classifier training) and present efficient *NumPy* code for boosting pre-trained weak hypotheses.

1 INTRODUCTION

Boosting algorithms are **meta learning** algorithms commonly used for training regression or classification models. The basic idea is to assemble several weak models, which individually are not really good at solving the task at hand, into a strong model and the crucial characteristic of boosting algorithms is that they build such an ensemble in a theoretically well founded and principled manner.

Note that we just used the terms *weak model* and *strong model*. Synonyms commonly found in the literature include *weak learner* or *weak hypothesis* for the former and *strong learner* or *boosted model* for the latter. What all these terms actually signify will become apparent in the next section.

Over the years, many boosting algorithms have been reported. Notable examples include (in chronological order): AnyBoost [9], LogitBoost [5], InfoBoost [1], gradient boosting [6], BrownBoost [2], SemiBoost [8], functional boosting [18], and precision-based boosting [11]. But the granddaddy of them all is **AdaBoost** due to Freund and Shapire [3, 4].

It is fair to say that, when it came out, AdaBoost revolutionized the theoretical understanding and practical applicability of machine learning methods. Originally, it was developed for binary classifier training, however, extensions to multi-class classification and regression are straightforward [15]. We are interested in AdaBoost because it naturally lends itself to *informed machine learning* which integrates data- and knowledge-driven techniques [17]. We will discuss this in great detail in a series of upcoming notes; here, we merely set the stage for this discussion.

To keep things simple, we focus on AdaBoost for binary classifier training and consider the following scenario: Given annotated training data

$$\mathcal{D} = \left\{ (x_1, y_1), (x_2, y_2), \dots, (x_n, y_n) \right\}$$

where the data points $x_j \in \mathbb{R}^m$ have been sampled from two classes Ω_1 and Ω_2 and their labels $y_j \in \{-1, +1\}$ indicate class membership in that

$$y_j = \begin{cases} +1, & \text{if } x_j \in \Omega_1 \\ -1, & \text{if } x_j \in \Omega_2 \end{cases}$$

we want to train a binary classifier $H : \mathbb{R}^m \rightarrow \{-1, +1\}$ which can predict (hopefully correct) class labels for previously unobserved data points.

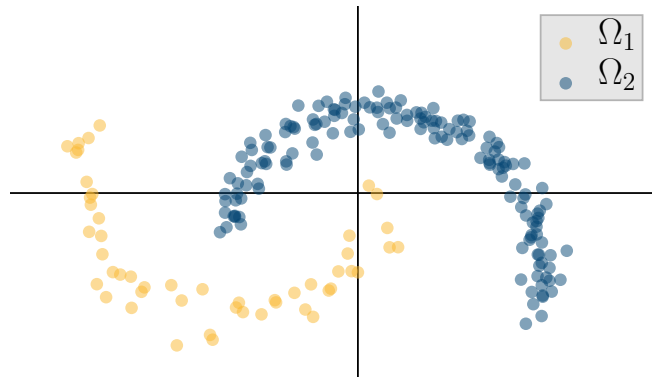


Figure 1: Didactic training data $x_j \in \mathbb{R}^2$. Note that classes Ω_1 (orange dots) and Ω_2 (blue dots) are not linearly separable.

In other words, our practical problem is to train a classifier H such that it learns to recognize if an input x belongs to class Ω_1 or Ω_2 and thus is able to predict a corresponding class label

$$y = H(x) = \begin{cases} +1, & \text{if } x \in \Omega_1 \\ -1, & \text{if } x \in \Omega_2 \end{cases}$$

Our strategy will be to run AdaBoost in order to assemble a strong model H from a set of weak models h_1, h_2, \dots

However, we will deviate from the currently predominant use of the method and consider a few twists and turns. For instance, we will not train the weak learners on the fly but assume that an expert has provided us a reasonable set of pre-trained models among which we need to select the most appropriate ones. This is not as revolutionary as it may sound but reflects how boosting was often done in the olden days. Also, we will not consider our weak models to be decision trees (or stumps) but will work with linear classifiers instead. This is surprisingly uncommon but —as we shall see later— connects boosting to neural network training.

Next, we first clarify the meaning behind the terms weak and strong model or hypothesis (section 2). We then discuss the theory behind AdaBoost and its adaptation to settings with given hypotheses (section 3). Afterwards, we put theory into practice and discuss corresponding *NumPy* code (section 4). Finally, we conclude with a summary and an outlook.

Readers who would like to experiment with our code snippets should be passingly familiar with *NumPy* and *SciPy* [12] and only need to

```
import numpy as np
```

2 SETTING THE STAGE

Before we turn to boosting, let us take a step back, reconsider what we said in the introduction, and ponder the following reasonable and truly fundamental questions:

Q1: What does it mean to train a model?

Q2: Where does the model come from?

Well, machine learning is the science of mathematical model fitting where mathematical models are understood to be mathematical functions with a certain, problem specific input/output behavior. For instance, in binary classification, the problem is to classify data, the inputs are data points, and the outputs are class labels.

The crux is that there are infinitely many mathematical functions which can map data points to labels. Since it is practically impossible to consider them all, we have to make modeling assumptions and must consider which *family of functions* might be suited for the task at hand. In other words, we human experts must choose what kind of model we want to train in order to solve our problem. In machine learning parlance we say that we have to decide for a *hypothesis class* \mathcal{H} .

Now, a hypothesis class can be as complex and powerful as the class of all deep neural networks with input/output layers whose dimensions match the dimensions of the in- and outputs we are dealing with. Or it could be as simple as the classes of shallow decision trees or linear classifiers of appropriate input/output dimensions. In either case, the hypothesis class \mathcal{H} once again contains infinitely many models or hypotheses h .

It is therefore important to require that the $h \in \mathcal{H}$ are models with adjustable *parameters*. For neural networks, the model parameters are connection weights and bias values; for decision trees; they are split dimensions and thresholds; for linear classifiers, they are again weights and bias values albeit much less than in the case of neural networks.

This is where model training comes into play. Once we have decided for a model or hypothesis class \mathcal{H} , we also need to decide for a training algorithm which automatically adjusts the model parameters such that the final hypothesis $h_* \in \mathcal{H}$ resulting from this optimization procedure solves our problem well. For neural networks, we typically use (variants of) the backpropagation algorithm, decision trees are trained using algorithms such as C4.5 or CART, and linear classifiers can be trained by a variety of methods each tailored towards a different loss function.

Speaking of loss functions, we also recall that model training is statistical optimization. The objective typically consists in minimization a loss function

$$\mathcal{L}(\mathcal{D}) = \sum_j L(h(\mathbf{x}_j), y_j)$$

which quantifies deviations between model predictions $h(\mathbf{x}_j)$ and ground truths y_j . Depending on the model complexity, training may or may not require several passes over the training data. In either case, the statistical nature of model training is due to two implicit characteristics of the whole approach.

First of all, the quality of a final hypothesis h_* crucially depends on the quality of the training data; different training samples may lead to different results. Second of all, loss functions typically reflect sample statistics such as expected values or divergences.

What does all of this have to do with classifier boosting? Well, as we said in the introduction, boosting algorithms are meta learners which implement a different philosophy of model training.

They, too, consider a hypothesis class \mathcal{H} but instead of training a final hypothesis $h_* \in \mathcal{H}$ they train a model $H \notin \mathcal{H}$. However, this model H incorporates (many) models $h \in \mathcal{H}$. For instance, in the context of binary classification, it is usually assumed to be

$$H(\mathbf{x}) = \text{sign} \left(\sum_{t=1}^T \alpha_t h_t(\mathbf{x}) \right)$$

In a sense, boosting algorithms therefore do more work: they train the individual models h_t and also determine the meta model parameters $\alpha_t \in \mathbb{R}$. However, if the chosen hypothesis class \mathcal{H} is a class of rather simple models, the effort for training the $h_t \in \mathcal{H}$ is typically low. Still, this would yield no advantage if the effort for estimating the α_t was high. The good news is that it is not.

Indeed, the breakthrough Freund and Shapire achieved with AdaBoost was to establish that estimating appropriate meta model parameters is actually easy. A considerable added benefit is that theory and practice since have shown that, even if \mathcal{H} is a hypothesis class of very weak learners h , i.e. of models h that are so simple that they will never perform really well on their own, the boosted model H is typically very strong.

In the next section, we explain all of this in greater detail and discuss simple modifications which connect AdaBoost to informed learning paradigms.

3 THEORY

For convenience, we begin our discussion of AdaBoost for binary classification by reiterating the practical problem specified in the introduction, namely: given training data

$$\mathcal{D} = \left\{ (\mathbf{x}_j, y_j) \right\}_{j=1}^n \quad (1)$$

where the data points $\mathbf{x}_j \in \mathbb{R}^m$ were sampled from two classes Ω_1 and Ω_2 and the $y_j \in \{-1, +1\}$ are corresponding class labels, we want to train a binary classifier $H : \mathbb{R}^m \rightarrow \{-1, +1\}$.

In particular, we want to run AdaBoost to train a classifier of the form

$$H(\mathbf{x}) = \text{sign} \left(\sum_{t=1}^T \alpha_t h_t(\mathbf{x}) \right) \quad (2)$$

where $\alpha_t \in \mathbb{R}$, $h_t \in \mathcal{H}$, and \mathcal{H} is an almost arbitrary hypothesis class of our liking.

We just said *almost arbitrary* because **there is one condition the hypotheses have to meet in our scenario, namely they have to be functions of the form**

$$h : \mathbb{R}^m \rightarrow \{-1, +1\} \quad (3)$$

By the same token, **we must insist on bipolar labels** $y_j \in \{-1, +1\}$ in our training data. Both these things are necessary for the math at the heart of AdaBoost to work¹.

¹Requirements like these are not uncommon. For instance, they also occur in the theory of SVM training. Of course one could argue that it would be nicer to work with any kind of label values but (products of) the numbers -1 and $+1$ have interesting properties that can be exploited to obtain compact yet expressive equations.

Algorithm 1 “classical” AdaBoost

Input: training data $\mathcal{D} = \{(\mathbf{x}_j, y_j)\}_{j=1}^n$
Input: hypothesis class \mathcal{H}
Input: parameter T

```

1: // initialize weight distribution  $\mathcal{P}$ 
2: for  $j = 1, \dots, n$ 
3:    $p_j \leftarrow \frac{1}{n}$ 
4:
5: for  $t = 1, \dots, T$ 
6:   // train weak model  $h_t \in \mathcal{H}$  on data  $\mathbf{x}_j$  weighted by  $p_j$ 
7:    $h_t \leftarrow \text{TRAINWEAKMODEL}_{\mathcal{H}}(\mathcal{D}, \mathcal{P})$ 
8:
9:   // compute classification error of model  $h_t$ 
10:   $\epsilon \leftarrow \sum_j p_j \cdot \Delta(h_t(\mathbf{x}_j), y_j)$ 
11:
12:  // compute coefficient  $\alpha_t$ 
13:   $\alpha_t \leftarrow \frac{1}{2} \cdot \ln\left(\frac{1-\epsilon}{\epsilon}\right)$ 
14:
15:  // update weight distribution  $\mathcal{P}$ 
16:  for  $j = 1, \dots, n$ 
17:     $p_j \leftarrow p_j \cdot e^{-\alpha_t \cdot h_t(\mathbf{x}_j) \cdot y_j}$ 
18:  for  $j = 1, \dots, n$ 
19:     $p_j \leftarrow \frac{1}{\sum_k p_k}$ 

```

Output: strong model $H(\mathbf{x}) = \text{sign}\left(\sum_{t=1}^T \alpha_t h_t(\mathbf{x})\right)$

There is one more thing we require from our hypothesis class \mathcal{H} : there must exist at least one training algorithm for fitting models in \mathcal{H} to labeled training data as in (1). We will generically call this algorithm $\text{TRAINWEAKMODEL}_{\mathcal{H}}$.

These very general assumption are really all we need to discuss the inner workings of AdaBoost. For now, we will therefore leave further details regarding \mathcal{H} unspecified and also do not care about how $\text{TRAINWEAKMODEL}_{\mathcal{H}}$ works.

However, we need to fix one more piece of notation before we can dive into AdaBoost: recall that the Kronecker delta of any two numbers a and b is given by

$$\delta(a, b) = \begin{cases} 1 & \text{if } a = b \\ 0 & \text{otherwise} \end{cases} \quad (4)$$

We may thus define an “anti” Kronecker delta as

$$\Delta(a, b) = 1 - \delta(a, b) = \begin{cases} 0 & \text{if } a = b \\ 1 & \text{otherwise} \end{cases} \quad (5)$$

At this point, we are good to go and can, first of all, look at the mechanics of “classical” AdaBoost and then, second of all, discuss AdaBoost with pre-trained hypotheses.

3.1 “Classical” AdaBoost

Algorithm 1 shows (rather verbose) pseudo-code for AdaBoost as introduced by Freund and Shapire [3, 4]. It does, of course, require some explanation.

During its operation, AdaBoost maintains a set of weights

$$\mathcal{P} = \{p_1, p_2, \dots, p_n\} \quad (6)$$

which are used to weight the data points

$$\mathcal{X} = \{\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_n\} \quad (7)$$

in the given training data \mathcal{D} . Note that \mathcal{P} is a **discrete probability distribution** because the $p_j \in \mathbb{R}$ contained in \mathcal{P} must obey

$$p_j \geq 0 \quad (8)$$

$$\sum_j p_j = 1 \quad (9)$$

Initially (in lines 2–3), all these weights are set to the same value $\frac{1}{n}$.

AdaBoost is an iterative procedure and performs T iterations where T is a parameter specified by the user.

In each iteration t , AdaBoost first calls $\text{TRAINWEAKMODEL}_{\mathcal{H}}$ to fit a model h_t to the training data in \mathcal{D} where the training data points \mathbf{x}_j are weighted by their corresponding current weights p_j . The exact nature of this weighting mechanism will depend on the nature of the hypothesis class \mathcal{H} and we do not discuss it further.

Once the trained model h_t is available, AdaBoost evaluates it on the training data and computes a weighted classification error ϵ . For correctly classified training data, we have $\Delta(h_t(\mathbf{x}_j), y_j) = 0$ so they do not contribute to ϵ . For incorrectly classified training data, we have $\Delta(h_t(\mathbf{x}_j), y_j) = 1$ so they contribute an amount p_j to ϵ .

Once the classification error ϵ of model h_t has been determined, AdaBoost uses it to compute the coefficient α_t for h_t .

Finally, once α_t has been determined, AdaBoost updates the weight distribution \mathcal{P} . This happens in lines 16–19 and we are deliberately specific about these updates. In particular, lines 18–19 re-normalize the updated weights such that they sum to one. In the literature, the weight updates are often written as

$$p_j \leftarrow \frac{1}{Z} \cdot p_j \cdot e^{-\alpha_t \cdot h_t(\mathbf{x}_j) \cdot y_j} \quad (10)$$

and accompanied by the statement that Z is a normalization constant. The reason why spelled out normalization the way we did has to do with the modern blogosphere².

Finally, once the **for** loop over t has terminated, AdaBoost has determined all the ingredients of the strong classifier in (2).

Of course there still are numerous open questions: Why and how does all of this work? Why are ϵ and α_t computed the way they are? Why do the weight updates involve exponentials?

While these are good and valid questions, answering them now would take us too far astray from the main points of this section. We therefore defer (very) detailed answers to these questions to the **appendix** and encourage interested readers to carefully go through the explanations presented there.

At this point, we simply claim that this stage-wise construction of a strong classifier $H(\mathbf{x})$ as a linear combination of several weak classifiers $h(\mathbf{x})$ is guaranteed to work. And don’t worry, we will go through a practical example below.

²There are numerous machine learning blogs on the Web, some of them of high quality, some of them of abysmal quality, many of them with a tendency to oversimplify things. With respect to posts on AdaBoost the latter two seem to abound. Indeed, googling for “AadBoost” yields many links to blogs with downright horrific explanations. For instance, we came across posts stating that normalization by Z is a cryptic or obscure or unnecessary step. But neither is the case! The $p_j \in \mathcal{P}$ must sum to one and lines 18–19 show how to ensure this after the updates in lines 16–17.

3.2 AdaBoost with Pre-Trained Hypotheses

It used to be somewhat common to run AdaBoost on a given discrete set of weak models rather than to have it train a model in each of its iterations [10, 14]. This requires slight modifications of the pseudo-code in Alg. 1, but, regarding practical implementations, can lead to rather efficient code. Here, we discuss the necessary modifications; our claims w.r.t. code will be substantiated in the next section.

Throughout, we assume to be given a set of pre-trained models

$$\mathcal{M} = \{h_1, h_2, \dots, h_k\} \subset \mathcal{H} \quad (11)$$

That is, we assume we are given models h_i which come from some hypothesis class (say, linear classifiers) but whose parameters have already been estimated and need not be trained anymore.

What difference will this make compared to “classical” AdaBoost? Well, consider this: instead of training a model at the beginning of each round of boosting, we now only need to determine which of the given h_i has the lowest weighted classification error. Hence, while “classical” AdaBoost produces a strong model as in (2) which involves a linear combination over T previously unknown hypotheses h_t , AdaBoost with pre-trained models will necessarily produce a boosted classifier

$$H(\mathbf{x}) = \text{sign} \left(\sum_{i=1}^k \lambda_i h_i(\mathbf{x}) \right) \quad (12)$$

that involves a linear combination over k given hypotheses h_i .

How do we determine the k coefficients λ_i ? Well, initially, we may set them all to zero. Then, at the beginning of each round of boosting, we need to compute the weighted classification error

$$\epsilon_i = \sum_j p_j \cdot \Delta(h_i(\mathbf{x}_j), y_j) \quad (13)$$

for each of our given models. If we denote the index of the currently best performing model by

$$l = \underset{i}{\text{argmin}} \epsilon_i \quad (14)$$

we can compute the value

$$\alpha \leftarrow \frac{1}{2} \cdot \ln \left(\frac{1 - \epsilon_l}{\epsilon_l} \right) \quad (15)$$

and then use it to update the current estimate for coefficient λ_l as

$$\lambda_l \leftarrow \lambda_l + \alpha \quad (16)$$

The final modification we must pay attention to is that the weight updates at the end of each round of boosting are now computed as

$$p_j \leftarrow \frac{1}{Z} \cdot p_j \cdot e^{-\alpha \cdot h_l(\mathbf{x}_j) \cdot y_j} \quad (17)$$

Now, let us be clever and *vectorize* all of this! Consider this: since the training data (\mathbf{x}_j, y_j) and models h_i are given in advance and fixed, the quantities $\Delta(h_i(\mathbf{x}_j), y_j)$ and $h_l(\mathbf{x}_j) \cdot y_j$ which occur in (13) and (17) can be computed before we begin boosting. That is, we can compute two matrices

$$\Delta \in \mathbb{R}^{k \times n} \quad \text{where} \quad \Delta_{ij} = \Delta(h_i(\mathbf{x}_j), y_j) \quad (18)$$

and

$$U \in \mathbb{R}^{k \times n} \quad \text{where} \quad U_{ij} = h_i(\mathbf{x}_j) \cdot y_j \quad (19)$$

and pass them to the boosting routine.

Algorithm 2 AdaBoost with pre-trained hypotheses

Input: matrix $\Delta \in \mathbb{R}^{k \times n}$

Input: matrix $U \in \mathbb{R}^{k \times n}$

Input: parameter T

```

1: // initialize coefficient vector  $\lambda$ 
2:  $\lambda \leftarrow \mathbf{0}$ 
3:
4: // initialize weight vector  $\mathbf{p}$ 
5:  $\mathbf{p} \leftarrow \frac{1}{n} \mathbf{1}$ 
6:
7: for  $t = 1, \dots, T$ 
8:   // evaluate models  $h_i \in \mathcal{M}$  on data  $\mathbf{x}_j$  weighted by  $p_j$ 
9:    $\epsilon = \Delta \mathbf{p}$ 
10:
11:   // determine best current model
12:    $l \leftarrow \underset{i}{\text{argmin}} \epsilon_i$ 
13:
14:   // compute  $\alpha$ 
15:    $\alpha \leftarrow \frac{1}{2} \cdot \ln \left( \frac{1 - \epsilon_l}{\epsilon_l} \right)$ 
16:
17:   // update coefficient  $\lambda_l$ 
18:    $\lambda_l \leftarrow \lambda_l + \alpha$ 
19:
20:   // update weight vector  $\mathbf{p}$ 
21:    $\mathbf{p} \leftarrow \frac{1}{Z} \cdot \mathbf{p} \odot \exp[-\alpha U_{l:}]$ 

```

Output: coefficients λ for model $H(\mathbf{x}) = \text{sign} \left(\sum_{i=1}^k \lambda_i h_i(\mathbf{x}) \right)$

If we further collect the weights p_j in a vector $\mathbf{p} \in \mathbb{R}^n$, we can compute all the classification errors at the beginning of each round of boosting in terms of a single matrix vector product, namely

$$\epsilon = \Delta \mathbf{p} \quad (20)$$

By the same token, the weight updates at the end of each round of boosting can then be realized as

$$\mathbf{p} \leftarrow \frac{1}{Z} \cdot \mathbf{p} \odot \exp[-\alpha U_{l:}] \quad (21)$$

where $U_{l:}$ denotes row l of matrix U , $\exp[\cdot]$ is understood to act entry-wise, and \odot is the Hadamard- or entry-wise product of two vectors.

All in all, AdaBoost with pre-trained hypotheses is therefore as simple as summarized in Alg. 2

3.3 A Didactic Example

Let us next look at a practical example for the behavior of Alg. 2. To this end, we will work with the $n = 200$ training data in Fig. 1 and boost a binary classifier composed of pre-trained hypotheses.

Note the following: while our training data are simple (the given data points \mathbf{x}_j are but elements of \mathbb{R}^2), the classification problem they pose is not. First of all, the samples drawn from classes Ω_1 and Ω_2 are obviously not linearly separable. Second of all, our data are slightly imbalanced; for class Ω_1 , there are 75 examples and, for class Ω_2 , there are 125 examples. Will boosting be able to cope with challenges like these?

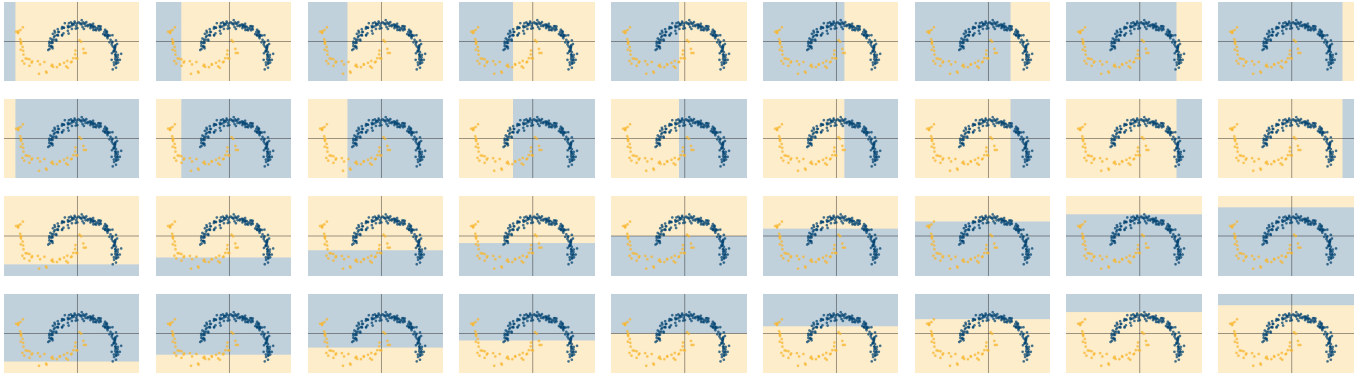


Figure 2: 36 weak models for boosting a binary classifier on the training data in Fig. 1. In each panel, the dots are the training data and the colored regions represent class regions predicted by the corresponding model. Each model is a linear classifier $h_i(\mathbf{x}) = \text{sign}(\mathbf{w}_i^\top \mathbf{x} - \theta_i)$. The \mathbf{w}_i are $\pm \mathbf{e}_{1/2}$ where $\mathbf{e}_1, \mathbf{e}_2$ are the standard basis vectors in \mathbb{R}^2 and the θ_i were drawn from an interval $[-\theta, +\theta]$. All models are truly weak learners as they classify at least a third of the training data incorrectly.

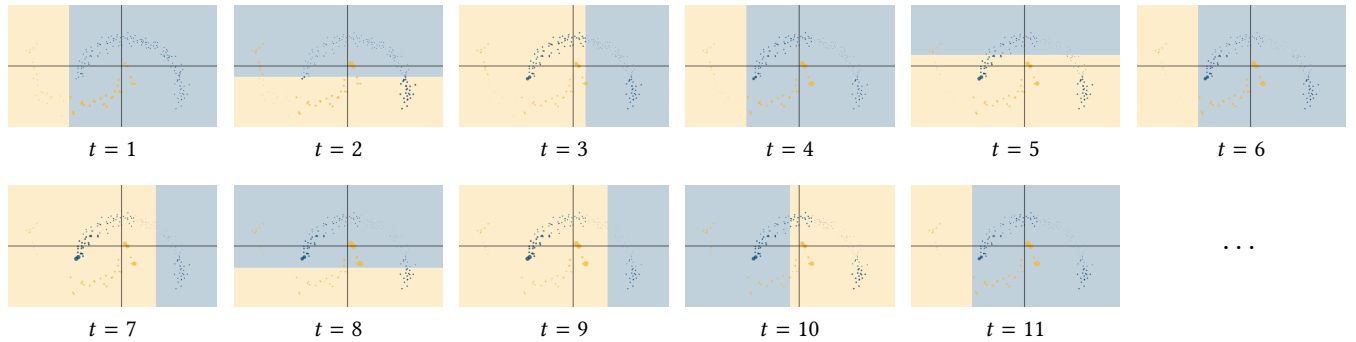


Figure 3: Visualization of the first couple of rounds of AdaBoost on the training data in Fig. 1 with the weak models in Fig. 2. Each panel visualizes the classifier chosen in round t . The dots represent the training data \mathbf{x}_j and their sizes are proportional to the (updated) weight p_j of \mathbf{x}_j at the end of round t .

An advantage of working with two-dimensional data is that we can actually visualize binary classifiers for such data. For instance, Fig. 2 depicts the set \mathcal{M} of 36 pre-trained models h_i we consider for our example.

Each of these weak models is a binary linear classifier of the form $h_i(\mathbf{x}) = \text{sign}(\mathbf{w}_i^\top \mathbf{x} - \theta_i)$ and thus splits the data space \mathbb{R}^2 into two complementary half-spaces, namely the half-space of those $\mathbf{x} \in \mathbb{R}^2$ for which $h_i(\mathbf{x}) \geq 0$ and the half-space of those $\mathbf{x} \in \mathbb{R}^2$ for which $h_i(\mathbf{x}) < 0$. Each panel in Fig. 2 visualizes these half-spaces in terms of an orange- and a blue region, respectively.

The figure also shows that the half-spaces or class regions produced by our given models are axis-aligned. Indeed, for each h_i in the first row, we have $\mathbf{w}_i = \mathbf{e}_1$ and for the h_i in the second row, we have $\mathbf{w}_i = -\mathbf{e}_1$ where $\mathbf{e}_1 = [1, 0]^\top$ denotes the first standard basis vector in \mathbb{R}^2 . For the h_i in the third and fourth row, we have $\mathbf{w}_i = \mathbf{e}_2$ and $\mathbf{w}_i = -\mathbf{e}_2$, respectively, where $\mathbf{e}_2 = [0, 1]^\top$ is the second standard basis vector in \mathbb{R}^2 . Finally, each of the threshold parameters θ_i were drawn from an interval $[-\theta, +\theta]$.

Furthermore, each panel in Fig. 2 also shows the training data we are working with. This allows us to “see” that our given models

are truly weak. For instance, the classifier in the top left panel classifies all the examples from class Ω_1 (orange dots) correctly but all the examples from class Ω_2 (blue dots) incorrectly. Its accuracy is therefore rather low, or vice versa, its classification error is rather high. Not all the models in Fig. 2 are as weak as the one in the top left panel. However, even the better models in our hypothesis set have a classification error of about 30% to 40%.

All in all, we can thus conclude that, individually, none of the given models $h_i(\mathbf{x})$ can satisfactorily solve our problem. So let us boost them into a strong model $H(\mathbf{x})$.

Figure 3 visualizes the outcomes of the first couple of rounds of AdaBoost. Its panels depict which model h_i was chosen in round t . They also show the training data \mathbf{x}_j such that the size of the respective dots is proportional to the corresponding weights p_j after their update at the end of round t .

We observe the following: if the model chosen in round t correctly classifies a point \mathbf{x}_j , its weight p_j decreases. On the other hand, if the model chosen in round t incorrectly classifies a point \mathbf{x}_j , its weight p_j increases. The model chosen in round $t + 1$ attempts to correct for mistakes made by the model chosen in round t . For

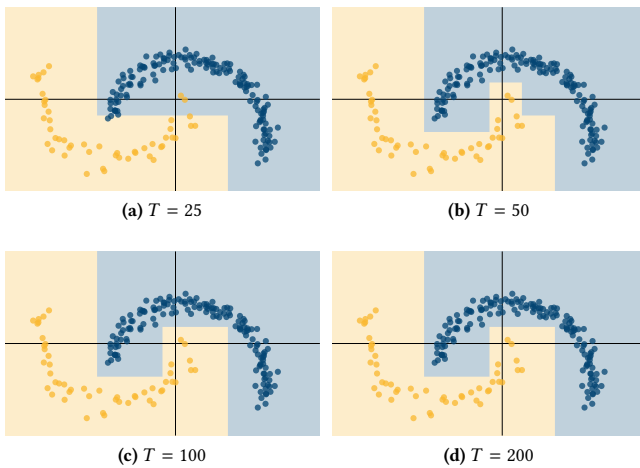


Figure 4: Boosted classifiers after T rounds of AdaBoost.

instance, the model chosen at $t = 7$ mis-classifies many blue points; the model chosen at $t = 8$ mis-classifies only a few blue points.

The are aspects of AdaBoost that are hard to visualize even when working with two-dimensional data. First of all, the whole processes of boosting with pre-trained hypotheses is incredibly fast. Second of all, however, it usually requires more than a just a few dozen rounds to produce a truly strong model $H(\mathbf{x})$. Since visualizing all these rounds would consume way too much space, Fig. 4 simply shows final models for $T \in \{25, 50, 100, 200\}$.

Looking at this figure, we observe that 25 rounds of boosting produced a classifier $H(\mathbf{x})$ that is considerably stronger than each of the weak hypotheses $h_i(\mathbf{x})$ in our model set \mathcal{M} . However, the boosted model after 50 rounds is even better. Eventually, the process leads to a model such as the one after 100 rounds which then does not change fundamentally anymore. Given that we have been working with super simple axis aligned linear classifiers as our weak models, the strong model after 100 iterations beautifully captures the gist of our training data.

4 PRACTICE

This section presents a straightforward *NumPy* implementation of the pseudo-code in Alg. 2. It is indeed so straightforward that most of our discussion will be spent on code for pre- and post-processing rather than on code for boosting itself.

First of all, we point out that **our pseudo-code for AdaBoost with pre-trained hypotheses is data- and model agnostic**. This means that it neither requires the training data nor the hypotheses set as input. Rather, it solely operates on the two matrices Δ and U which encode how the given hypotheses perform on the given training data. In order to run our algorithm, we therefore need to prepare these matrices; however, this pre-processing is completely independent of the boosting mechanism itself. In other words, our algorithm applies to whatever kind of data and weak hypotheses as long as matrices Δ and U can be set up consistently.

Sticking with the practical, binary classification example we just went through, we will therefore focus on how to set up AdaBoost

Listing 1: custom made sign function

```
def signum(x):
    return (x >= 0) * 2 - 1
```

with pre-trained hypotheses for the case where these hypotheses are linear classifiers of the form $h_i(\mathbf{x}) = \text{sign}(\mathbf{w}_i^\top \mathbf{x} - \theta_i)$.

In this setting, we can gather all our application specific ingredients, i.e. the n training data points $\mathbf{x}_j \in \mathbb{R}^m$ and labels $y_j \in \{-1, +1\}$ as well as the k weight vectors $\mathbf{w}_i \in \mathbb{R}^m$ and threshold values $\theta_i \in \mathbb{R}$, in two matrices and two vectors, namely

$$X = \begin{bmatrix} | & & | \\ \mathbf{x}_1 & \cdots & \mathbf{x}_n \\ | & & | \end{bmatrix} \quad \text{and} \quad \mathbf{y} = \begin{bmatrix} y_1 \\ \vdots \\ y_n \end{bmatrix}$$

as well as

$$W = \begin{bmatrix} -\mathbf{w}_1^\top & - \\ \vdots & \\ -\mathbf{w}_k^\top & - \end{bmatrix} \quad \text{and} \quad \boldsymbol{\theta} = \begin{bmatrix} \theta_1 \\ \vdots \\ \theta_k \end{bmatrix}$$

In *NumPy*, objects like these are naturally encoded in terms of one- and two-dimensional arrays

```
matX = ...
vecY = ...
matW = ...
vecT = ...
```

and we henceforth assume that they have been set/initialized/filled appropriately.

However, for our following practical computations to work, we need `vecY` to be an array of shape $1 \times n$ and `vecT` to be an array of shape $k \times 1$, so we better enforce this

```
vecY = vecY.flatten()
vecT = vecT.reshape(k, 1)
```

We will also need a custom made version of the sign function, because `np.sign` implements

$$\text{sign}(x) = \begin{cases} -1 & \text{if } x < 0 \\ 0 & \text{if } x = 0 \\ +1 & \text{if } x > 0 \end{cases}$$

but what we most commonly need in the context of binary linear classification is

$$\text{sign}(x) = \begin{cases} -1 & \text{if } x < 0 \\ +1 & \text{if } x \geq 0 \end{cases}$$

Yet, we recall that we have come cross this issues several times before. Back then, we simply used the function in Listing 1 and we will do so here as well.

Next, we note the following. The label which model h_i predicts for training data point \mathbf{x}_j is given by

$$\hat{y}_{ij} = h_i(\mathbf{x}_j) = \text{sign}(\mathbf{w}_i^\top \mathbf{x}_j - \theta_i)$$

The predictions of this model for all training data points $\mathbf{x}_1, \dots, \mathbf{x}_n$ can be gathered in a vector

$$\hat{\mathbf{y}}_i = [\hat{y}_{i1} \quad \hat{y}_{i2} \quad \cdots \quad \hat{y}_{in}]^\top$$

Continuing this line of reasoning, the prediction vectors of all our models h_1, \dots, h_k can be gathered in a matrix

$$\hat{Y} = \begin{bmatrix} -\hat{y}_1^T & - \\ \vdots & \\ -\hat{y}_k^T & - \end{bmatrix}$$

and the practical computation of this matrix is as easy as

```
matY = signum(matW @ matX - vecT)
```

Next, we note that the ground truth label vector \mathbf{y} and prediction matrix \hat{Y} allow for computing matrix U defined in (19). We simply have

$$U = \mathbf{1}\mathbf{y}^T \odot \hat{Y}$$

where $\mathbf{1}$ denotes the k -dimensional vector of all ones and \odot is once again the Hadamard product. Using *NumPy*, this expression can be computed as

```
matU = vecY * matY
```

Once array `matU` is available, we can compute an array

```
matD = matU < 0
```

which implements the “anti” Kronecker delta matrix Δ in (18). This snippet exploits that $y_j \in \{-1, +1\}$ and $\hat{y}_{ij} = h_i(\mathbf{x}_j) \in \{-1, +1\}$ so that the product $y_j \cdot \hat{y}_{ij}$ equals $+1$ if y_j and \hat{y}_{ij} agree and -1 if they don’t. (Skeptical readers are encouraged to verify that array `matD` really implements matrix Δ as defined in (18).)

At this point, we have everything in place to run *AdaBoost* in order to determine a strong model $H(\mathbf{x}) = \text{sign}(\sum_i \lambda_i h_i(\mathbf{x}))$. In fact, we recall that all we need to worry about is estimating the appropriate coefficient vector $\lambda \in \mathbb{R}^k$ and so we simply call

```
vecL = AdaBoostPreTrained(matD, matU, tmax=100)
```

Our implementation of function `AdaBoostPreTrained` is shown in Listing 2 and is indeed an immediate implementation of the pseudo-code in Alg. 2. All we need to point out is that we implement the vectors λ , \mathbf{p} , and ϵ as *NumPy* arrays `vecL`, `vecP`, and `vecE`, respectively. Also, the scalars l , ϵ_l , and α in Alg. 2 become `l`, `e`, and `a` in Listing 2. That is all there is to it! Our implementation of *AdaBoost* with given hypotheses really just boils down to this tiny piece of *NumPy* code.

Once array `vecL` has been computed, we have essentially trained a strong classifier and are basically done. However, looking back at the example in the previous section, we should point out a quirk of boosting with pre-trained hypotheses. Specifically, Fig. 3 seems to suggest that it may happen that some of the given models $h_i(\mathbf{x})$ are never selected into the strong ensemble $H(\mathbf{x})$. If this happens, their corresponding coefficients λ_i will still be 0 once boosting has terminated.

To see if this is indeed the case, we may use

```
inds = np.where(vecL > 0)[0]
```

in order to determine the indices i of the coefficients of the models that have been selected into the strong classifier. Given `inds`, we may then print these indices together with the corresponding coefficients, for instance, using

```
print (*zip(inds+1, np.round(vecL[inds], 3)))
```

Listing 2: AdaBoost with pre-trained hypotheses

```
def AdaBoostPreTrained(matD, matU, tmax=100):
    k, n = matD.shape

    vecL = np.zeros(k)
    vecP = np.ones(n) / n

    for t in range(tmax):
        vecE = matD @ vecP

        l = np.argmax(vecE)
        e = vecE[l]
        a = 0.5 * np.log((1-e) / e)

        vecL[l] += a

        vecP = vecP * np.exp(-a * matU[l])
        vecP /= vecP.sum()

    return vecL
```

When we do this for the data (and weak hypotheses) we considered in our practical example, this results in

```
>>> (4, 0.376) (5, 3.905) (21, 9.439)
      (24, 0.9) (25, 8.188) (29, 0.333)
      (30, 3.765) (31, 1.218) (33, 4.009)
      (34, 0.31)
```

which tells us that indeed only 10 out of our 36 weak models have been selected into the boosted model.

Hence, in order to avoid unnecessary computations during the application phase of our strong classifier, i.e. to avoid computations of the form $0 \cdot h_i(\mathbf{x})$, we therefore reduce the weight matrix, the threshold vector, and the coefficient vector to only those components that are relevant. To this end, we use

```
matW = matW[inds]
vecT = vecT[inds]
vecL = vecL[inds]
```

Finally, to get an impression of the performance of our strong model $H(\mathbf{x})$, we may evaluate it on the training data (of course it goes without saying that it is much much better to evaluate on independent test data). Given everything we said and implemented so far, we can compute the predictions of our boosted model for all of our training data as

```
vecH = signum(vecL @ signum(matW @ matX - vecT))
```

and then compare these predictions to the ground truth labels

```
print (np.array_equal(vecY, vecH))
```

When we do this for the data (and weak hypotheses) we considered in our practical example, this results in

```
>>> True
```

which tells us that our boosted classifier has learned to classify its training data perfectly (see also once again Fig. 4(c)).

5 SUMMARY AND OUTLOOK

This has become a rather lengthy note (and there is still an appendix ahead of us ;-)). However, it was worth it. We had a first look at *AdaBoost* in general and *AdaBoost* with pre-trained hypotheses in particular. For the latter, we also presented very compact *NumPy* code.

The code we discussed in section 4 is actually the code we used to solve the binary classification problem discussed in section 3. Taking this as anecdotal evidence, we can conclude that AdaBoost is a simple yet powerful machine learning tool.

Indeed, the many favorable properties of AdaBoost (ease of implementation, speed, and reliability) have been exploited in numerous practical applications. The most famous solution based on AdaBoost is arguably the celebrated Viola-Jones object detector [16]. Other use cases include systems for human-machine interaction where the machine must be able to learn on the fly [7, 19], assisted driving [20, 21], causality detection in financial documents [13], and many, many more.

Although we can now better understand the role of AdaBoost in applications like these, there is still much left to say and study. First of all, we can think of even more efficient implementations. These could involve mechanisms for early stopping and more efficient weight normalization. Second of all, we may even think of “weightless” boosting. Especially for the case of boosting with pre-trained hypotheses, there are arguably simpler techniques we may use to determine the sought after coefficients of the given weak learners. Third of all, in the introduction of this note, we teased a connection between AdaBoost with linear classifiers and neural network training. We definitely need to further elaborate on this.

But, most importantly, we also need to further elaborate on AdaBoost and informed learning. That is, we need to answer the question of where do pre-trained hypotheses come from? Indeed, the pre-trained hypothesis we considered in our examples in this note were not at all informed. Even though we worked with linear classifiers, our models were so simple that we could have just as well been using decision stumps. We can do better than this! The fundamental questions are: How? Are there mechanisms for an informed, i.e. problem specific, choice of pre-trained hypotheses?

All these open points and question will be addressed in further notes. For now, we end with even more theory ...

APPENDIX

This appendix answers several crucial questions we had to leave open in our theoretical discussion in section 3, namely: Why and how exactly does AdaBoost in Alg. 1 work? Why are ϵ and the α_t and p_j computed the way they are? What is the rationale behind all of this?

To begin with, we let $H_{t-1}(\mathbf{x})$ be the linear combination of weak models after $t - 1$ iterations of AdaBoost. That is, we let

$$H_{t-1}(\mathbf{x}) = \alpha_1 h_1(\mathbf{x}) + \alpha_2 h_2(\mathbf{x}) + \dots + \alpha_{t-1} h_{t-1}(\mathbf{x}) \quad (22)$$

With this definition, the linear combination of weak models which is produced in iteration t can then be written as

$$H_t(\mathbf{x}) = H_{t-1}(\mathbf{x}) + \alpha_t h_t(\mathbf{x}) \quad (23)$$

Note: even for $t = T$, $H_T(\mathbf{x})$ must not be confused with the strong model $H(\mathbf{x})$ which results from running AdaBoost. This would be $H(\mathbf{x}) = \text{sign}(H_T(\mathbf{x}))$

Now, the fundamental, latent idea behind AdaBoost is that it implicitly minimizes the following **exponential loss**

$$\mathcal{L}(\mathcal{D}, t) = \sum_{j=1}^n e^{-y_j \cdot H_t(\mathbf{x}_j)} \quad (24)$$

Note: we just used the terms *latent* and *implicitly* because, looking at the pseudo-code in Alg. 1, it is not at all apparent that AdaBoost minimizes the quantity in (24).

Yet another important, latent idea concerns the weights $p_j \in \mathcal{P}$ which feature prominently in AdaBoost. Let us assume that, **at the beginning of the t -th iteration**, we have

$$p_j = \frac{1}{Z} \cdot e^{-y_j \cdot H_{t-1}(\mathbf{x}_j)} \quad (25)$$

Note: in the main text, we emphasized the importance of the normalizing factor Z . However, in each round of AdaBoost, it remains constant until it gets updated at the very end of said round. As a constant multiplicative factor it will not impact any of the arguments that follow. To improve readability we will therefore henceforth drop Z from (most of) our equations.

Given everything we said so far, we can rewrite the loss which is to be minimize in iteration t of AdaBoost as follows

$$\mathcal{L} = \sum_{j=1}^n e^{-y_j \cdot H_t(\mathbf{x}_j)} \quad (26)$$

$$= \sum_{j=1}^n e^{-y_j \cdot (H_{t-1}(\mathbf{x}_j) + \alpha_t h_t(\mathbf{x}_j))} \quad (27)$$

$$= \sum_{j=1}^n e^{-y_j \cdot H_{t-1}(\mathbf{x}_j)} \cdot e^{-\alpha_t \cdot y_j \cdot h_t(\mathbf{x}_j)} \quad (28)$$

$$= \sum_{j=1}^n p_j \cdot e^{-\alpha_t \cdot y_j \cdot h_t(\mathbf{x}_j)} \quad (29)$$

Now it will pay off that we required $y_j \in \{-1, +1\}$ as well as $h_t(\mathbf{x}_j) \in \{-1, +1\}$, because:

If the predicted label $h_t(\mathbf{x}_j)$ for \mathbf{x}_j equals the corresponding ground truth label y_j , we have $y_j \cdot h_t(\mathbf{x}_j) = +1$ and thus

$$e^{-\alpha_t \cdot y_j \cdot h_t(\mathbf{x}_j)} = e^{-\alpha_t} \quad (30)$$

If the predicted label $h_t(\mathbf{x}_j)$ for \mathbf{x}_j differs from the corresponding ground truth label y_j , we have $y_j \cdot h_t(\mathbf{x}_j) = -1$ and thus

$$e^{-\alpha_t \cdot y_j \cdot h_t(\mathbf{x}_j)} = e^{\alpha_t} \quad (31)$$

These two observations therefore allow us to rewrite the loss as

$$\mathcal{L} = \sum_{y_j = h_t(\mathbf{x}_j)} p_j \cdot e^{-\alpha_t} + \sum_{y_j \neq h_t(\mathbf{x}_j)} p_j \cdot e^{\alpha_t} \quad (32)$$

What follows next, is yet another clever trick: note that we may further write

$$\mathcal{L} = \sum_{y_j = h_t(\mathbf{x}_j)} p_j \cdot e^{-\alpha_t} + \sum_{y_j \neq h_t(\mathbf{x}_j)} p_j \cdot e^{-\alpha_t} \quad (33)$$

$$= + \sum_{y_j \neq h_t(\mathbf{x}_j)} p_j \cdot e^{\alpha_t} - \sum_{y_j \neq h_t(\mathbf{x}_j)} p_j \cdot e^{-\alpha_t} \quad (34)$$

from which we obtain

$$\mathcal{L} = \sum_{j=1}^n p_j \cdot e^{-\alpha_t} + \sum_{y_j \neq h_t(\mathbf{x}_j)} p_j \cdot (e^{\alpha_t} - e^{-\alpha_t}) \quad (35)$$

$$= e^{-\alpha_t} \sum_{j=1}^n p_j + (e^{\alpha_t} - e^{-\alpha_t}) \sum_{y_j \neq h_t(\mathbf{x}_j)} p_j \quad (36)$$

Next, we recall that the p_j are discrete probabilities which obey

$$\sum_{j=1}^n p_j = 1 \quad (37)$$

so that

$$\mathcal{L} = e^{-\alpha_t} + (e^{\alpha_t} - e^{-\alpha_t}) \sum_{y_j \neq h_t(\mathbf{x}_j)} p_j \quad (38)$$

To even further simplify this expression, we also recall our definition of the ‘‘anti’’ Kronecker delta and observe that it allows us to write

$$\sum_{y_j \neq h_t(\mathbf{x}_j)} p_j = \sum_{j=1}^n p_j \cdot \Delta(h_t(\mathbf{x}_j), y_j) \quad (39)$$

But now we recognize the right hand side of (39) as the classification error ϵ computed in line 10 of Alg. 1. All in all this is to say that

$$\mathcal{L} = e^{-\alpha_t} + (e^{\alpha_t} - e^{-\alpha_t}) \cdot \epsilon \quad (40)$$

We therefore have the important intermediate result that the exponential loss which ought to be minimized in iteration t of AdaBoost depends of the classification error ϵ of the weak classifier h_t trained or selected in that iteration and on the coefficient α_t that needs to be computed in that iteration.

Which choice of α_t would minimize the loss? Well, in order to determine the optimal α_t , we can compute the derivative

$$\frac{d\mathcal{L}}{d\alpha_t} = -e^{-\alpha_t} + (e^{\alpha_t} + e^{-\alpha_t}) \cdot \epsilon = e^{\alpha_t} \epsilon + e^{-\alpha_t} (\epsilon - 1) \quad (41)$$

and equate it to 0. This provide us with

$$\frac{e^{\alpha_t}}{e^{-\alpha_t}} = \frac{1 - \epsilon}{\epsilon} \quad (42)$$

$$\Leftrightarrow e^{2\alpha_t} = \frac{1 - \epsilon}{\epsilon} \quad (43)$$

$$\Rightarrow \alpha_t = \frac{1}{2} \ln \left(\frac{1 - \epsilon}{\epsilon} \right) \quad (44)$$

which is exactly what is computed in line 13 of Alg. 1.

Finally, at this stage in round t , we can extend the linear combination $H_{t-1}(\mathbf{x})$ of models selected in previous rounds by the presently selected model to obtain $H_t(\mathbf{x}) = H_{t-1}(\mathbf{x}) + \alpha_t h_t(\mathbf{x})$. Since we assumed that the weights p_j at the beginning of a round are proportional to $H_{t-1}(\mathbf{x}_j)$, i.e.

$$p_j \propto e^{-y_j \cdot H_{t-1}(\mathbf{x}_j)} \quad (45)$$

we therefore need to update them to the form required for the next round, i.e.

$$p_j \propto e^{-y_j \cdot H_t(\mathbf{x}_j)} = p_j \cdot e^{-\alpha_t \cdot y_j \cdot h_t(\mathbf{x}_j)} \quad (46)$$

This is what happens in lines 16–17 of Alg. 1. To make sure that the updated set of weights meets the condition $\sum_j p_j = 1$, they have to be re-normalized as in lines 18–19 of Alg. 1.

ACKNOWLEDGMENTS

This material was produced within the Competence Center for Machine Learning Rhine-Ruhr (ML2R) which is funded by the Federal Ministry of Education and Research of Germany (grant no. 01IS18038C). The authors gratefully acknowledge this support.

REFERENCES

- [1] J.A. Aslam. 2000. Improving Algorithms for Boosting. In *Proc. COLT*.
- [2] Y. Freund. 2001. An Adaptive Version of the Boost by Majority Algorithm. *Machine Learning* 43, 3 (2001).
- [3] Y. Freund and R.E. Shapire. 1995. A Decision-theoretic Generalization of On-line Learning and an Application to Boosting. In *Computational Learning Theory*, P. Vitanyi (Ed.), LNCS, Vol. 904. Springer.
- [4] Y. Freund and R.E. Shapire. 1997. A Decision-Theoretic Generalization of On-Line Learning and an Application to Boosting. *J. of Computer and System Sciences* 55, 1 (1997).
- [5] J. Friedman, T. Hastie, and R. Tibshirani. 2000. Additive Logistic Regression: A Statistical View of Boosting. *Annals of Statistics* 28, 2 (2000).
- [6] J. Friedman, T. Hastie, and R. Tibshirani. 2001. Greedy Function Approximation: A Gradient Boosting Machine. *Annals of Statistics* 29, 5 (2001).
- [7] M. Hanheide, C. Bauckhage, and G. Sagerer. 2005. Combining Environmental Cues & Head Gestures to Interact with Wearable Devices. In *Proc. Int. Conf. on Multimodal Interfaces*.
- [8] K. Hatano and M.K. Warmuth. 2003. Boosting versus Covering. In *Proc. NIPS*.
- [9] L. Mason, J. Baxter, P. Bartlett, and M. Frean. 1999. Boosting Algorithms as Gradient Descent. In *Proc. NIPS*.
- [10] I. Mukherjee, C. Rudin, and R.E. Schapire. 2013. The Rate of Convergence of AdaBoost. *J. of Machine Learning Research* 14, 34 (2013).
- [11] M.H. Nikravan, M. Movahedan, and S. Zilles. 2021. Precision-based Boosting. In *Proc. AAAI*.
- [12] T.E. Oliphant. 2007. Python for Scientific Computing. *Computing in Science & Engineering* 9, 3 (2007).
- [13] M. Pielka, A. Ladi, C. Chapman, E. Brito, R. Ramamurthy, P. Mayer, A. Wahab, R. Sifa, and C. Bauckhage. 2020. Using Ensemble Methods and Sequence Tagging to Detect Causality in Financial Documents. In *Proc. FinCausal*.
- [14] C. Rudin, I. Daubechies, and R.E. Schapire. 2003. On the Dynamics of Boosting. In *Proc. NIPS*.
- [15] R.E. Shapire and Y. Singer. 2001. Improved Boosting Algorithms Using Confidence-rated Predictions. *Machine Learning* 37, 3 (2001).
- [16] P. Viola and M. Jones. 2001. Rapid Object Detection Using a Boosted Cascade of Simple Features. In *Proc. CVPR*.
- [17] L. von Rueden, S. Mayer, K. Beckh, B. Georgiev, S. Giesselbach, R. Heese, B. Kirsch, J. Pfommer, A. Pick, R. Ramamurthy, M. Walczak, J. Garcke, C. Bauckhage, and J. Schuecker. 2019. Informed Machine Learning – A Taxonomy and Survey of Integrating Knowledge into Learning Systems. *arXiv:1903.12394 [stat.ML]* (2019).
- [18] C. Wang, Y. Wang, W. E, and R.E. Schapire. 2015. Functional Frank-Wolfe Boosting for General Loss Functions. *arXiv:1510.02558 [stat.ML]* (2015).
- [19] S. Wrede, M. Hanheide, C. Bauckhage, and G. Sagerer. 2004. An Active Memory as a Model for Information Fusion. In *Proc. Int. Conf. on Information Fusion*.
- [20] S. Zhang, C. Bauckhage, and A.B. Cremers. 2014. Informed Haar-like Features Improve Pedestrian Detection. In *Proc. CVPR*.
- [21] S. Zhang, C. Bauckhage, and A.B. Cremers. 2015. Efficient Pedestrian Detection via Rectangular Features Based on a Statistical Shape Model. *IEEE Trans. on Intelligent Transportation Systems* 16, 2 (2015).