CS5339 Sample Project – AdaBoost Algorithm

Jonathan Scarlett

Notes on this Sample Project

- This project is only meant to give an example indication of style, format, etc., rather than being a model for an "ideal" project. (Though, hypothetically, it could be considered one that would receive a fairly high grade)
- To avoid overlap with the suggested topics, this sample project is based on a topic that will be covered in the lectures. Your own project should of course avoid significant overlap with the lectures (though comparisons/connections to that material can be highlighted).
- It is not necessary to include everything here (e.g., experiments, detailed proofs, extensions, citations to several published papers) in your report. You should try to judge this based on your chosen topic and the assessment criteria.
- You may notice the use of the first-person "we" despite only having a single author. This has become conventional in academic writing, but you can use the first-person "I" or the passive voice if you prefer.
- This sample report is relatively light on the "Additional Material" part regarding experiments and published papers, though it includes a bit of both. You should ideally include slightly more in your own report.

1 Introduction

The fundamental problem of binary classification is central to both classical and modern machine learning. Many of the earliest works focused on simple classification rules, such as linear classifiers or nearest-neighbor rules, whereas more recent developments have led to more sophisticated ideas such as kernel methods, artificial neural networks, and more. Many such methods can be viewed as *combining simple classifiers* to produce a *complex classifier*.

In this report, we will overview a famous algorithm falling under this category, known as AdaBoost, which was introduced by Freund and Schapire in 1997 [5], and subsequently significantly impacted both the theory and practice of machine learning. The authors were awarded the 2003 Gödel prize, and the algorithm is often considered to be one one of the most effective "off-the-shelf" algorithms requiring minimal tuning and tweaking [3, 6].

Before proceeding with a formal description, we outline some of the main high-level concepts and ideas behind AdaBoost:

- The algorithm is given a class of *weak learners*, also known as *base learners*, containing classifiers that are not very powerful in themselves. Despite this, suitably "combining" many such classifiers can lead to a much more powerful classifier.
- The combining is done by a simple voting strategy – each base learner classifies a given point as positive or negative, and the final classification is performed according to a weighted vote of these base learners, where higher votes are given to the base learners that are believed to be more accurate. The role of AdaBoost is to *select the base learners*, as well as choosing the voting weight for each of them.
- The base learners are selected one at a time, and are chosen to minimize a *weighted training error* according to some weights that are maintained on the data points (not to be confused with the voting weights). The more a point has been classified correctly by previously-selected base learners, the lower weight it receives, whereas a point that was previously classified incorrectly many times is assigned a higher weight. This ensures that more attention is focused on the points where we need to make up for earlier mistakes.
- The weights are updated using the idea of *multiplicative weight updates*, which is a far-reaching tool that has also been explored in numerous other fields [2].

To appreciate the third of these points, it is useful to think of the following analogy: If a teacher wants to teach their students a number of concepts, then they should focus more attention on the concepts that many students scored poorly on in previous tests/exams.

2 Description of AdaBoost

In this section, we formally introduce the AdaBoost algorithm, and give some numerical examples.

2.1 Preliminaries

Following the lecture notes, we consider binary classification ($y \in \{-1, 1\}$) of d-dimensional input vectors ($\mathbf{x} \in \mathbb{R}^d$), and we seek to learn a classifier from a set of n data points, $\mathcal{D} = \{(\mathbf{x}_t, y_t)\}_{t=1}^n$. While AdaBoost can be defined with an arbitrary class of base learners, we will focus our attention on one of the simplest possible such classes, namely, *decision stumps*. A decision stump performs classification by simply thresholding one of the d input features. More formally, such a classifier can be written as

$$h(\mathbf{x}; \boldsymbol{\theta}) = \text{sign}(s(x_k - \theta_0)),$$

where there are three parameters $\boldsymbol{\theta} = \{s, k, \theta_0\}$:

(i) k is the index of the (only) feature that the decision is made based on;

(ii) s is the sign, which is included to allow both combinations of $+$ on one side and $-$ on the other;

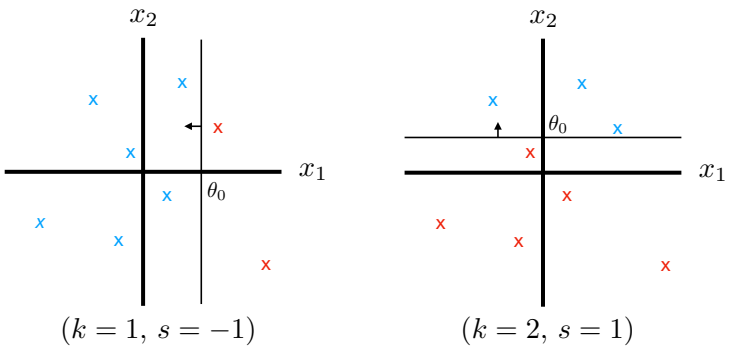

Figure 1: Two examples of decision stumps, each applied to the same unlabeled data set. The points classified as positive are colored blue, and the points classified as negative are colored red.

(iii) θ_0 is the threshold at which different decisions are made on either side.

Examples of decision stumps in the case $d = 2$ are depicted in Figure 1. As outlined above, AdaBoost will produce a classifier according to a weighted combination of M decision stumps:

$$f_M(\mathbf{x}) = \sum_{m=1}^{M} \alpha_m h(\mathbf{x}; \boldsymbol{\theta}_m), \tag{1}$$

leading to the final decision rule $\hat{y} = \text{sign}(f_M(\mathbf{x}))$.

2.2 The Algorithm

AdaBoost takes as input the data set $\mathcal{D}$ and the number of iterations M, and performs the following steps:

1. Initialize weights $w_0(t) = \frac{1}{n}$ for $t = 1, \ldots, n$.
2. For $m = 1, \ldots, M$, do the following:
   (a) Choose the next base learner $h(\cdot; \hat{\boldsymbol{\theta}}_m)$ as follows:
   $$\hat{\boldsymbol{\theta}}_m = \arg\min_{\boldsymbol{\theta}} \sum_{t\,:\,y_t \neq h(\mathbf{x}_t;\boldsymbol{\theta})} w_{m-1}(t). \tag{2}$$
   (b) Set $\hat{\alpha}_m = \frac{1}{2} \log \frac{1-\hat{\epsilon}_m}{\hat{\epsilon}_m}$, where $\hat{\epsilon}_m = \sum_{t\,:\,y_t \neq h(\mathbf{x}_t;\hat{\boldsymbol{\theta}}_m)}^{n} w_{m-1}(t)$ is the minimal value attained in (2).
   (c) Update the weights:
   $$w_m(t) = \frac{1}{Z_m} w_{m-1}(t) e^{-y_t h(\mathbf{x}_t;\hat{\boldsymbol{\theta}}_m)\hat{\alpha}_m} \tag{3}$$
   for each $t = 1, \ldots, n$, where Z_m is defined so that the weights sum to one:
   $$Z_m = \sum_{t=1}^{n} w_{m-1}(t) e^{-y_t h(\mathbf{x}_t;\hat{\boldsymbol{\theta}}_m)\hat{\alpha}_m}. \tag{4}$$
3. Output $f_M(\mathbf{x}) = \sum_{m=1}^{M} \hat{\alpha}_m h(\mathbf{x}; \hat{\boldsymbol{\theta}}_m)$, corresponding to the classifier $\hat{y} = \text{sign}(f_M(\mathbf{x}))$.

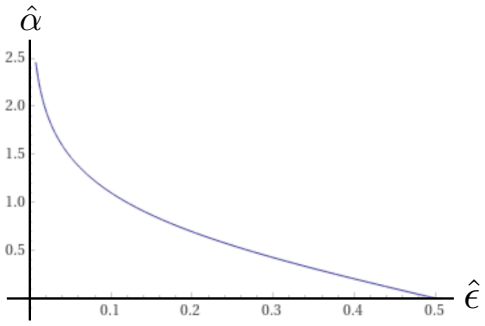

Figure 2: Plot of $\hat{\alpha}_m$ as a function of $\hat{\epsilon}_m$.

We proceed by providing some intuition behind each of the main steps. For step (a), we refer to

$$\epsilon_m = \sum_{t\,:\,y_t \neq h(\mathbf{x}_t;\boldsymbol{\theta})} w_{m-1}(t) \tag{5}$$

as *weighted training error*. The selection rule (2) is choosing a base learner that "classifies best" when certain samples are treated as more important than others, as dictated by the weights $w_{m-1}(\cdot)$.

To understand the weight update rule (3) in step (c), it is useful to note that the quantity $-y_t h(\mathbf{x}_t;\boldsymbol{\theta})$ only takes values $+1$ or -1, and as a result, (3) can be rewritten as

$$w_m(t) = \frac{1}{Z_m} w_{m-1}(t) \times \begin{cases} e^{\hat{\alpha}_m} & y_t \neq h(\mathbf{x}_t;\hat{\boldsymbol{\theta}}_m) \\ e^{-\hat{\alpha}_m} & y_t = h(\mathbf{x}_t;\hat{\boldsymbol{\theta}}_m). \end{cases} \tag{6}$$

As a result, we are *increasing the weight* if the newly-chosen base learner classifies $\mathbf{x}_t$ incorrectly, and *decreasing the weight* otherwise. This aligns with the intuition given in Section 1; future iterations should place more importance on inputs that were previously classified wrongly.

The choice of $\hat{\alpha}_m$ in step (b) arises from optimizing a certain expression in the mathematical analysis (see Section 3), but we can still give some intuition behind it here. In Figure 2, we plot this choice of $\hat{\alpha}_m$ as a function of $\hat{\epsilon}_m$, and observe that it is a decreasing function. We therefore have the intuitive property that a higher vote is given to a base learner that has a lower weighted training error. In the extreme cases, we have the following:

- If $\hat{\epsilon}_m = 0$, then we have a "perfect" base learner, so it receives an infinitely large vote. However, this case will only occur in a data set where a single decision stump can classify all points correctly, which is not to be expected.
- If $\hat{\epsilon}_m = \frac{1}{2}$, then we have a "useless" base learner, since even random guessing has a 50% success rate. Hence, in this case, the base learner receives a vote of zero.

We note that the case $\hat{\epsilon}_m > \frac{1}{2}$ will never be encountered in (2), because if we have a decision stump with weighted training error greater than $\frac{1}{2}$, then flipping its sign will make the weighted training error less than half. Hence, the latter will be preferred by the minimization performed in (2).

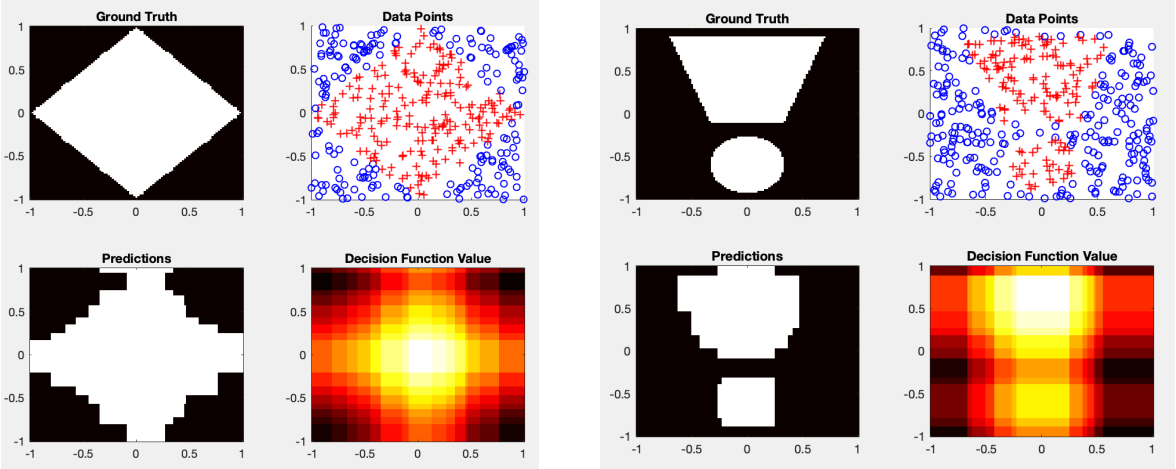

Figure 3: Experimental results for AdaBoost on two data sets with $n = 400$ data points and $M = 50$ boosting iterations. The bottom-right of each example shows a color map of the function $f_M(\mathbf{x})$, and taking its sign gives the predictions shown in the bottom-left.

2.3 Experimental Examples

In Figure 3, we show the results of running AdaBoost for $M = 50$ iterations, using two different data sets with $n = 400$ points. The $\mathbf{x}_t$ values were chosen uniformly at random from $[-1, 1]^2$, and their labels were assigned according to the "ground truth" image shown.

We see that AdaBoost is able to roughly recover the shapes of the classification functions that were used to generate the data. On the other hand, the "blocky" behavior of the final output could be considered undesirable. This behavior arises from the use of the *extremely simple* decision stump class for the base learner; see Section 4 for further discussion.

3 Mathematical Analysis

An elegant theoretical result on the AdaBoost algorithm states that as long as each base learner performs *slightly better* than random guessing (i.e., slightly better weighted training error than $\frac{1}{2}$), the proportion of misclassified points in $\mathcal{D}$ is guaranteed to decrease *exponentially fast* in the number of iterations M. This is formally stated in the following.

Theorem 1. *For any data set $\mathcal{D} = \{(\mathbf{x}_t, y_t)\}_{t=1}^n$, after M iterations, the training error of AdaBoost satisfies*

$$\frac{1}{n}\sum_{t=1}^{n} \mathbb{1}\{y_t \neq \operatorname{sign}(f_M(\mathbf{x}_t))\} \leq \exp\left(-2\sum_{m=1}^{M}\left(\frac{1}{2} - \hat{\epsilon}_m\right)^2\right). \tag{7}$$

In particular, if $\hat{\epsilon}_m \leq \frac{1}{2} - \gamma$ for all m and some $\gamma > 0$, then

$$\frac{1}{n}\sum_{t=1}^{n} \mathbb{1}\{y_t \neq \operatorname{sign}(f_M(\mathbf{x}_t))\} \leq e^{-2M\gamma^2}. \tag{8}$$

This result makes no explicit assumptions on the data set (e.g., linear separability), but rather only assumes $\hat{\epsilon}_m \leq \frac{1}{2} - \gamma$, i.e., slightly outperforming random guessing. It is also interesting to note that since the left-hand side of (8) only takes values in $\{0, \frac{1}{n}, \dots\}$, the training error is zero for $M > \frac{\log n}{2\gamma^2}$.

We only provide a brief outline of the proof of Theorem 1 here, and provide the details in the appendix:

- The first step is to upper bound the weighted training-error in terms the *exponential loss*, defined as

$$\text{Loss}_{\text{exp}}(y, f(\mathbf{x})) = \exp(-yf(\mathbf{x})). \tag{9}$$

  Also defining the usual 0-1 loss $\text{Loss}_{01}(y, f(\mathbf{x})) = \mathbb{1}\{y \neq \text{sign}(f(\mathbf{x}))\}$, a simple plot of the two losses can be used to establish that $\text{Loss}_{01}(y, f(\mathbf{x})) \leq \text{Loss}_{\text{exp}}(y, f(\mathbf{x}))$, and hence the training error satisfies

$$\frac{1}{n}\sum_{t=1}^{n} \mathbb{1}\{y_t \neq \text{sign}(f_M(\mathbf{x}_t))\} \leq \frac{1}{n}\sum_{t=1}^{n} e^{-y_t f_M(\mathbf{x}_t)}. \tag{10}$$

- The second step is to rewrite the right-hand side of (10) as follows:

$$\frac{1}{n}\sum_{t=1}^{n} e^{-y_t f_M(\mathbf{x}_t)} = \prod_{m=1}^{M} Z_m. \tag{11}$$

  While this is an unusual identity, it follows by a fairly simple recursion argument in which we write $w_0(t) = \frac{1}{n}$, then use the weight update expression (3) to write $w_1(t)$ in terms of Z_1, $w_2(t)$ in terms of Z_1 and Z_2, and so on, up to $w_M(t)$ in terms of $Z_1, \dots, Z_M$. The fact that $\sum_{t=1}^{n} w_M(t) = 1$ can then be used to establish (11).

- The remaining steps use some algebraic manipulations to characterize each Z_m and upper bound it in terms of simpler quantities. First, the definition of Z_m is simplified to $e^{\hat{\alpha}_m}\hat{\epsilon}_m + e^{-\hat{\alpha}_m}(1 - \hat{\epsilon}_m)$, and then solving $\frac{\partial Z_m}{\partial \hat{\alpha}_m} = 0$ shows that the choice $\hat{\alpha}_m = \frac{1}{2}\log\frac{1-\hat{\epsilon}_m}{\hat{\epsilon}_m}$ gives the smallest value of Z_m, and that this value simplifies to

$$Z_m = \sqrt{1 - (1 - 2\hat{\epsilon}_m)^2}. \tag{12}$$

  With some additional tedious but basic algebraic manipulations, (12) can then be further upper bounded as follows:

$$Z_m \leq \exp\Big(-\frac{1}{2}(1 - 2\hat{\epsilon}_m)^2\Big). \tag{13}$$

  Combining (13) with (10)–(11), we obtain the desired result in (7).

4 Extensions and Further Results

In this section, we briefly outline some more advanced algorithms and theory building on the previous sections. Due to space limitations, we only provide a short discussion on each of these.

4.1 Base Learners Beyond Decision Stumps

In principle, any class of base learners could be used, instead of only decision stumps. The choice of base learner class requires a suitable balance between various factors – having a more complex base learner may lead to more powerful classifiers and fewer boosting iterations being required (i.e., smaller M), but may also make the crucial selection step in (2) computationally challenging. In addition, overly complicated base learners may lead to problems with overfitting. In practice, a generalization of decision stumps known as *decision trees* often provides a good balance of these competing factors [6].

4.2 Multi-Class Boosting

While classification problems are commonly assumed to be binary, *multi-class* problems are also of considerable practical interest (e.g., number recognition would have 10 classes, and digit recognition would have at least 36). Techniques for converting binary classifiers to multi-class classifiers (e.g., one vs. rest and one vs. one) often pose significant practical limitations, e.g., see [4, Section 4.1.2]. Fortunately, AdaBoost permits a natural direct extension to the multi-class setting, both in terms of the algorithm and its underlying theory [7]. In addition, although it is usually described in the context of classification, it can even be applied to regression problems [1], in which the label y is continuous rather than finite-valued.

4.3 Characterization of the Test Error

In the discussion after Theorem 1, we concluded that the training error of AdaBoost decreases to zero when M is large enough. In general, zero training error in machine learning is *not* considered a good thing, as it suggests that the classifier may be overly complex and subject to overfitting. Surprisingly, however, AdaBoost has been shown to exhibit quite the opposite behavior in practice: The test error (i.e., the performance on data that was not used in training) remains small even when the training error is zero, and can even *continue to decrease* when AdaBoost is run for additional iterations after reaching zero training error.

A partial explanation for this phenomenon was given by Schapire *et al.* [8] using the notion of *margin*. Recalling that the proof of Theorem 1 is based on upper bounding the 0-1 loss by the exponential loss, we find that a very small exponential loss not only guarantees a low 0-1 loss, but also guarantees a *large margin* (i.e., $f_M(\mathbf{x})$ is not only positive, but is far from zero). By exploiting this property and some advanced connections between margin and generalization, theoretical guarantees can be provided not only for the training error, but also the test error [8].

References

[1] Adaboost – Wikipedia. https://en.wikipedia.org/wiki/AdaBoost.

[2] The reasonable effectiveness of the multiplicative weights update algorithm. https://jeremykun.com/2017/02/27/the-reasonable-effectiveness-of-the-multiplicative-weights-update-algorithm/.

[3] sklearn.ensemble.AdaBoostRegressor. https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.AdaBoostRegressor.html.

[4] Christopher M Bishop. *Pattern recognition and machine learning*. springer, 2006.

[5] 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.

[6] Sachin Joglekar. A (small) introduction to boosting. https://codesachin.wordpress.com/tag/adaboost/.

[7] Mohammad J Saberian and Nuno Vasconcelos. Multiclass boosting: Theory and algorithms. In *Advances in Neural Information Processing Systems*, pages 2124–2132, 2011.

[8] Robert E Schapire, Yoav Freund, Peter Bartlett, Wee Sun Lee, et al. Boosting the margin: A new explanation for the effectiveness of voting methods. *The Annals of Statistics*, 26(5):1651–1686, 1998.

Appendix

A Proof of Theorem 1 (AdaBoost Training Error Guarantee)

The proof proceeds in several steps.

Step 1 (Convert to exponential loss): As stated following (9), the exponential loss upper bounds the 0-1 loss, so

$$\frac{1}{n}\sum_{t=1}^{n}\mathbb{1}\{y_t \neq \text{sign}(f_M(\mathbf{x}_t))\} \leq \frac{1}{n}\sum_{t=1}^{n} e^{-y_t f_M(\mathbf{x}_t)}. \tag{14}$$

Step 2 (An unusual equality): In this step, we show that

$$\frac{1}{n}\sum_{t=1}^{n} e^{-y_t f_M(\mathbf{x}_t)} = \prod_{m=1}^{M} Z_m. \tag{15}$$

To see this, we write out the weight updates recursively according to the initialization $w_(t) = \frac{1}{n}$ and the update rule (3):

$$\begin{aligned}
w_0(t) &= \frac{1}{n} \\
w_1(t) &= \frac{1}{n}\frac{\exp(-\hat{\alpha}_1 y_t h(\mathbf{x}_t;\boldsymbol{\theta}_1))}{Z_1} \\
w_2(t) &= \frac{1}{n}\frac{\exp(-\hat{\alpha}_1 y_t h(\mathbf{x}_t;\boldsymbol{\theta}_1))}{Z_1}\frac{\exp(-\hat{\alpha}_2 y_t h(\mathbf{x}_t;\boldsymbol{\theta}_2))}{Z_2} \\
&\vdots \\
w_M(t) &= \frac{1}{n}\frac{\exp\big(-\sum_{m=1}^{M}\hat{\alpha}_m y_t h(\mathbf{x}_t;\boldsymbol{\theta}_m)\big)}{\prod_{m=1}^{M} Z_m} = \frac{1}{n}\cdot\frac{\exp\big(-y_t f_M(\mathbf{x}_t)\big)}{\prod_{m=1}^{M} Z_m},
\end{aligned}$$

where in the last step we substituted (1). The desired claim follows since $\sum_{t=1}^{M} w_M(t) = 1$ by construction (i.e., by the definition of the weights in the algorithm).

Step 3 (Rewriting Z_m): Recall the definition of Z_m in (4). We can split the sum over t into two cases: If $y_t = h(\mathbf{x}_t;\hat{\boldsymbol{\theta}}_m)$ then $y_t h(\mathbf{x}_t;\hat{\boldsymbol{\theta}}_m) = 1$, whereas if $y_t \neq h(\mathbf{x}_t;\hat{\boldsymbol{\theta}}_m)$ then $y_t h(\mathbf{x}_t;\hat{\boldsymbol{\theta}}_m) = -1$. Therefore, (4) simplifies to

$$\begin{aligned}
Z_m &= \sum_{t\,:\,y_t \neq h(\mathbf{x}_t;\hat{\boldsymbol{\theta}}_m)} e^{\hat{\alpha}_m} w_{m-1}(t) + \sum_{t\,:\,y_t = h(\mathbf{x}_t;\hat{\boldsymbol{\theta}}_m)} e^{-\hat{\alpha}_m} w_{m-1}(t) \\
&= e^{\hat{\alpha}_m}\hat{\epsilon}_m + e^{-\hat{\alpha}_m}(1-\hat{\epsilon}_m),
\end{aligned} \tag{16}$$

where we have used the fact that $\sum_{t\,:\,y_t \neq h(\mathbf{x}_t;\hat{\boldsymbol{\theta}}_m)} w_{m-1}(t) = \hat{\epsilon}_m$ by definition, and we similarly have $\sum_{t\,:\,y_t = h(\mathbf{x}_t;\hat{\boldsymbol{\theta}}_m)} w_{m-1}(t) = 1 - \hat{\epsilon}_m$ since the weights sum to one by definition.

Note that $\frac{\partial Z_m}{\partial \hat{\alpha}_m} = \hat{\epsilon}_m e^{\hat{\alpha}_m} - e^{-\hat{\alpha}_m}(1-\hat{\epsilon}_m)$; setting this to zero and solving gives $\hat{\alpha}_m = \frac{1}{2}\log\frac{1-\hat{\epsilon}_m}{\hat{\epsilon}_m}$. It is easy to check that it is a minimum, and not a maximum.

Step 4 (Substitute $\hat{\alpha}_m$ and simplify): Substituting the choice of $\hat{\alpha}_m$ into (16) gives

$$Z_m = \sqrt{\frac{1-\hat{\epsilon}_m}{\hat{\epsilon}_m}}\hat{\epsilon}_m + \sqrt{\frac{\hat{\epsilon}_m}{1-\hat{\epsilon}_m}}(1-\hat{\epsilon}_m) \tag{17}$$

$$= 2\sqrt{\hat{\epsilon}_m(1-\hat{\epsilon}_m)} \tag{18}$$

$$= \sqrt{1-(1-2\hat{\epsilon}_m)^2}, \tag{19}$$

where the last step can be verified by expanding the square. The last line allows us to use the convenient inequality $\sqrt{1-c^2} = \exp\big(\frac{1}{2}\log(1-c^2)\big) \leq \exp\big(-\frac{1}{2}c^2\big)$m where the last step uses $\log(1+a) \leq a$. The motivation behind doing this is that products of exponentials are convenient, because they simplify to $\exp\big(\sum \dots\big)$.

Applying this upper bound in (19) gives $Z_m \leq \exp\big(-\frac{1}{2}(1-2\hat{\epsilon}_m)^2\big)$, and combining with (14)–(15) gives

$$\begin{aligned}\frac{1}{n}\sum_{t=1}^{n}\mathbb{1}\{y_t \neq \operatorname{sign}(f_M(\mathbf{x}_t))\} &\leq \prod_{m=1}^{M} Z_m \\ &\leq \prod_{m=1}^{M}\exp\bigg(-\frac{1}{2}(1-2\hat{\epsilon}_m)^2\bigg) \\ &= \exp\bigg(-2\sum_{m=1}^{M}\Big(\frac{1}{2}-\hat{\epsilon}_m\Big)^2\bigg),\end{aligned}$$

which proves the desired result in (7).