Advanced Machine Learning **Fall 2020**

Lecture 9: Ensemble Methods

Note: *LaTeX template courtesy of UC Berkeley EECS dept.*

Disclaimer: *These notes are adapted from ETH's Advanced Machine Learning Course, Cornell's CS4780 Course, "The Element of Statistical Learning" and "Boosting: Foundations and Algorithms" books.*

9.1 Bagging

Suppose we fit a model to our training data $\mathcal{Z} = \{(x_1, y_1), (x_2, y_2), ..., (x_n, y_n)\}$, obtaining the prediction $\hat{f}(x)$ at input x. Bootstrap aggregation or bagging averages this prediction over a collection of bootstrap samples, thereby reducing its variance. For each bootstrap sample $\boldsymbol{\mathcal{Z}}^{*b}$, $b = 1, 2, ..., B$, we fit our model, giving prediction $\hat{f}^{*b}(x)$. The bagging estimate is defined by:

$$\hat{f}_{bag}(x) = \frac{1}{B} \sum_{b=1}^{B} \hat{f}^{*b}(x)$$

Bagging can dramatically reduce the variance of unstable procedures like trees, leading to improved prediction. A simple argument shows why bagging helps under squared-error loss, in short because averaging reduces variance and leaves bias unchanged.

Theorem 9.1 *Assume our training observations $(x_i, y_i), i = 1, ..., N$ are independently drawn from a distribution $\mathcal{P}$, and consider the ideal aggregate estimator $f_{ag}(x) = \mathbb{E}_{\mathcal{P}}[\hat{f}^*(x)]$. Here x is fixed and the bootstrap dataset $\boldsymbol{\mathcal{Z}^*}$ consists of observations x_i^*, y_i^*, $i = 1, 2, ..., N$ sampled from $\mathcal{P}$. Then:*

$$\mathbb{E}_{\mathcal{P}}[Y - \hat{f}^*(x)]^2 = \mathbb{E}_{\mathcal{P}}[Y - \hat{f}_{ag}(x)]^2 + \overbrace{\mathbb{E}_{\mathcal{P}}[\hat{f}^*(x) - \hat{f}_{ag}(x)]^2}^{variance} \geq \mathbb{E}_{\mathcal{P}}[Y - \hat{f}_{ag}(x)]^2$$

Proof:

$$\begin{aligned} \mathbb{E}_{\mathcal{P}}[Y - \hat{f}^*(x)]^2 &= \mathbb{E}_{\mathcal{P}}[Y + \hat{f}_{ag}(x) - \hat{f}_{ag}(x) - \hat{f}^*(x)]^2 \\ = \mathbb{E}_{\mathcal{P}}[Y - \hat{f}_{ag}(x)]^2 + \mathbb{E}_{\mathcal{P}}[\hat{f}^*(x) - \hat{f}_{ag}]^2 &+ \overbrace{2\mathbb{E}_{\mathcal{P}}[(Y - \hat{f}_{ag}(x))(\hat{f}^*(x) - \hat{f}_{ag}(x))]}^{=0} \\ &\geq \mathbb{E}_{\mathcal{P}}[Y - \hat{f}_{ag}(x)]^2 \end{aligned}$$

■

Note that $f_{ag}(x)$ is a bagging estimate, drawing bootstrap samples from the actual population $\mathcal{P}$ rather than the data. It is not an estimate that we can use in practice, but is convenient for analysis. Furthermore, note the strong assumption of independence, which unfortunately does not hold when sampling from $\boldsymbol{\mathcal{Z}}$. The extra error on the right-hand side comes from the variance of $\hat{f}^*(x)$ around its mean $\hat{f}_{ag}(x)$. Therefore true population aggregation never increases mean squared error. This suggests that bagging—drawing samples from the training data— will often decrease mean-squared error. However, the above argument does not hold for classification under 0-1 loss, because of the nonadditivity of bias and variance. In that setting, bagging a good classifier can make it better, but bagging a bad classifier can make it worse. Note that when we bag a model, **any simple structure in the model is lost**. For interpretation of the model this is clearly a drawback.

9.1.1 Advantages of Bagging

- Reduces variance, so has a strong beneficial effect on high variance classifiers.
- As the prediction is an average of many classifiers, you obtain a mean score and variance. The latter can be interpreted as the uncertainty of the prediction. Especially in regression tasks, such uncertainties are otherwise hard to obtain.
- Bagging provides an unbiased estimate of the test error, which we refer to as the **out-of-bag error**. The idea is to average the classifiers $\hat{f}^{*b}$ that have not seen a certain sample. Thus, we obtain a classifier that was not trained on (x_i, y_i) ever. If we compute the error of all these classifiers, we obtain an estimate of the true test error. **The beauty is that we can do this without reducing the training set**. We just run bagging as it is intended and obtain this so called out-of-bag error for free.

  More formally, for each training point $(x_i, y_i) \in \mathcal{Z}$ let $S_i = \{k|(x_i, y_i) \notin \mathcal{Z}^{*k}\}$ - in other words S_i contains the indexes of all the training sets which do not contain (x_i, y_i). Let the averaged classifier over all these data sets be:

  $$h_i(x) = \frac{1}{|S_i|} \sum_{k \in S_i} f^{*k}(x)$$

  The out-of-bag error becomes simply the average loss that all these classifiers yield:

  $$\epsilon_{OOB} = \frac{1}{n} \sum_{i=1}^{n} l(h_i(x_i), y_i)$$

  This is an estimate of the test error, because for each sample we used the subset of classifiers that never saw that sample during training. If **B** is sufficiently large, the fact that we take out some classifiers has no significant effect and the estimate is pretty reliable.

9.1.2 Random Forest

One of the most famous and useful bagged algorithms is the Random Forest. A Random Forest is essentially nothing else but bagged decision trees, with a slightly modified splitting criteria.

The algorithm works as follows:

1. Sample B datasets $\mathcal{Z}^{*1}, \ldots, \mathcal{Z}^{*B}$ from $\mathcal{Z}$ with replacement.
2. For each $\mathcal{Z}^{*b}$ train a full decision tree $\hat{f}^{*b}(x)$ with one small modification: before each split randomly subsample $k \leq d$ features (without replacement) and only consider these for your split. It can be shown that this step further increases the variance of the trees.
3. The final classifier will be $\hat{f}(x) = \frac{1}{B} \sum_{b=1}^{B} \hat{f}^{*b}(x)$

Random Forest algorithm has two main advantages:

- It only has two hyper-parameters, **B** and **k**. It is extremely insensitive to both of these. A good choice for k is $k = \sqrt{d}$ (where d denotes the number of features). You can set B as large as you can afford.
- Decision trees do not require a lot of preprocessing. For example, the features can be of different scale, magnitude, or slope. This can be highly advantageous in scenarios with heterogeneous data, for example the medical settings where features could be things like blood pressure, age, gender, ..., each of which is recorded in completely different units.

9.2 Boosting

In his Machine Learning class project in 1988 Michael Kearns famously asked the question: **Can weak learners be combined to generate a strong learner with low bias?**
The answer is yes. Create an ensemble classifier $H_T(x) = \sum_{t=1}^{T} \alpha_t h_t(x)$. This ensemble classifier is built in an iterative fashion: during iteration t we add the classifier $\alpha_t h_t(x)$ to the ensemble. At test time we evaluate all classifier and return the weighted sum.

The process of constructing such an ensemble in a stage-wise fashion is very similar to gradient descent (we can think of it as gradient descent in functional space). However, instead of updating the model parameters in each iteration, we add functions to our ensemble. Let ℓ denote a (convex and differentiable) loss function:

$$\ell(H) = \frac{1}{n} \sum_{i=1}^{n} \ell(H(x_i), y_i)$$

Assume we have already finished **t** iterations and already have an ensemble classifier $H_t(\mathbf{x})$. Now in iteration $t+1$ we want to add one more weak learner h_{t+1} to the ensemble. To this end we search for the weak learner that minimizes the loss:

$$h_{t+1} = argmin_{h \in \mathbb{H}}\ \ell(H_t + \alpha h_t)$$

Once h_{t+1} has been found, we add it to our ensemble, i.e. $H_{t+1} := H_t + \alpha h_{t+1}$.

9.2.1 Gradient descent in functional space

Given H, we want to find the step-size α and (weak learner) h to minimize the loss $\ell(H + \alpha h)$. For this purpose, we can use Taylor Approximation as we did for gradient descent:

$$\ell(H + \alpha h) \approx \ell(H) + \alpha \nabla \ell(H) \cdot h$$

This approximation (of ℓ as a linear function) only holds within a small region around $\ell(H)$, i.e. as long as α is small. We therefore fix it to a small constant (e.g. $\alpha \approx 0.1$). With the step-size α fixed, we can use the approximation above to find an almost optimal h:

$$argmin_{h \in \mathbb{H}}\ \ell(H + \alpha h) \approx argmin_{h \in \mathbb{H}}\ \nabla \ell(H) \cdot h = argmin_{h \in \mathbb{H}}\ \sum_{i=1}^{n} \frac{\partial \ell}{\partial H}(x_i) \cdot h(x_i)$$

Hence, we can do boosting if we have an algorithm $\mathbb{A}$ to solve:

$$h_{t+1} = argmin_{h \in \mathbb{H}} \sum_{i=1}^{n} \overbrace{\frac{\partial \ell}{\partial H}(x_i)}^{r_i} \cdot h(x_i)$$

Note that we make progress as long as $\sum_{i=1}^{n} r_i h(x_i) < 0$.

```
Input: $\ell$, $\alpha$, $\{(\mathbf{x}_i, y_i)\}$, $\mathbb{A}$
$H_0 = 0$
for $t$=0:T-1 do
    $\forall I : r_i = \frac{\partial \ell((H_t(\mathbf{x}_1), y_1), \ldots, (H_t(\mathbf{x}_n), y_n))}{\partial H(\mathbf{x}_i)}$
    $h_{t+1} = \mathbb{A}(\{(\mathbf{x}_1, r_1), \ldots, (\mathbf{x}_n, r_n)\}) = \mathrm{argmin}_{h \in \mathbb{H}} \sum_{i=1}^{n} r_i h(\mathbf{x}_i)$
    if $\sum_{i=1}^{n} r_i h_{t+1}(\mathbf{x}_i) < 0$ then
        $H_{t+1} = H_t + \alpha_{t+1} h_{t+1}$
    else
        return $(H_t)$
    end
end
return $H_T$
```

Algorithm 1: AnyBoost in Pseudo-Code

9.2.2 AdaBoost

We begin by describing the most popular boosting algorithm due to Freund and Schapire (1997) called "AdaBoost". Consider a two-class problem where:

- The output variable are coded as $y_i \in \{-1, 1\}$, $\forall i$

- Weak learners $h \in \mathbb{H}$ are binary, $h(x_i) \in \{-1, 1\}$, $\forall i$

- The loss is the exponential loss: $\ell(H) = \sum_{i=1}^{n} e^{-y_i H(x_i)}$

First we compute the gradient $\mathbf{r_i} = \dfrac{\partial \ell}{\partial H}(x_i) = -y_i e^{-y_i H(x_i)}$

For notational convenience, let us define $\mathbf{w_i} = \dfrac{e^{-y_i H(x_i)}}{\sum_{i=1}^{n} e^{-y_i H(x_i)}}$ so that $\sum_{i=1}^{n} w_i = 1$.

Each weight $\mathbf{w_i}$ therefore has a very nice interpretation: it is the relative contribution of the training point (x_i, y_i) towards the overall loss. Now, in order to find the best next weak learner, we need to solve the optimization problem posed earlier:

$$
\begin{aligned}
h(x_i) &= argmin_{h\in\mathbb{H}} \sum_{i=1}^{n} r_i h(x_i) \\
&= argmin_{h\in\mathbb{H}} -\sum_{i=1}^{n} y_i e^{-y_i H(x_i)} h(x_i) && \text{(substitute in } r_i\text{)} \\
&= argmin_{h\in\mathbb{H}} -\sum_{i=1}^{n} y_i w_i h(x_i) && \text{(we can divide by } \sum_{i=1}^{n} e^{-y_i H(x_i)} \text{ since it is a constant)} \\
&= argmin_{h\in\mathbb{H}} \sum_{i:h_(x_i)\neq y_i} w_i - \sum_{i:h_(x_i)=y_i} w_i && \text{(since } h(x_i)y_i = 1 \iff h(x_i) = y_i\text{)} \\
&= argmin_{h\in\mathbb{H}} \sum_{i:h_(x_i)\neq y_i} w_i && \text{(since } \sum_{i:h_(x_i)\neq y_i} w_i = 1 - \sum_{i:h_(x_i)=y_i} w_i\text{)}
\end{aligned}
$$

Let us denote this weighted classification error as $\epsilon = \sum_{i:h_(x_i)\neq y_i} w_i$. Then, in order for the inner-product $\sum_{i=1}^{n} r_i h(x_i)$ to be negative, it just needs less than $\epsilon < 0.5$ weighted training error.

The next step is finding the optimal stepsize α (i.e. the one that minimizes ℓ the most).

We would like to solve the following optimization problem:

$$\alpha = argmin_{\alpha}\ \ell(H + \alpha h) = argmin_{\alpha} \sum_{i=1}^{n} e^{-y_i[H(x_i)+\alpha h(x_i)]}$$

We differentiate w.r.t. α and equate with zero:

$$\frac{\partial \ell(H+\alpha h)}{\partial \alpha} = 0 \implies \sum_{i=1}^{n} y_i h(x_i) e^{-(y_i H(x_i) + \alpha y_i h(x_i))} = 0$$

$$-\sum_{i:h(x_i)y_i=1} e^{-(y_i H(x_i)+\alpha \overbrace{y_i h(x_i)}^{1})} + \sum_{i:h(x_i)y_i=-1} e^{-(y_i H(x_i)+\alpha \overbrace{y_i h(x_i)}^{-1})} = 0 \qquad (\ y_i h(x_i) \in \{1, -1\})$$

$$-\sum_{i:h(x_i)y_i=1} w_i e^{-\alpha} + \sum_{i:h(x_i)y_i=-1} w_i e^{\alpha} = 0 \qquad \text{(divide everything by } \sum_{i=1}^{n} e^{-y_i H(x_i)}\text{)}$$

$$-(1-\epsilon)e^{-\alpha} + \epsilon e^{\alpha} = 0 \qquad (\epsilon = \sum_{i:h_(x_i)y_i=-1} w_i)$$

$$\alpha = \frac{1}{2}\ln\frac{1-\epsilon}{\epsilon}$$

It is unusual that we can find the optimal step-size in such a simple closed form. One consequence is that AdaBoost converges extremely fast.

Finally, after you take a step, i.e. $H_{t+1} = H_t + \alpha h$, you need to re-compute all the weights and then re-normalize. It is however straight-forward to show that the unnormalized weight $\hat{w}_i$ is updated as:

$$\hat{w}_i \leftarrow \hat{w}_i e^{-\alpha h(x_i) y_i}$$

and that the normalizer $Z = \sum_{i=1}^{n} e^{-y_i H(x_i)}$ becomes:

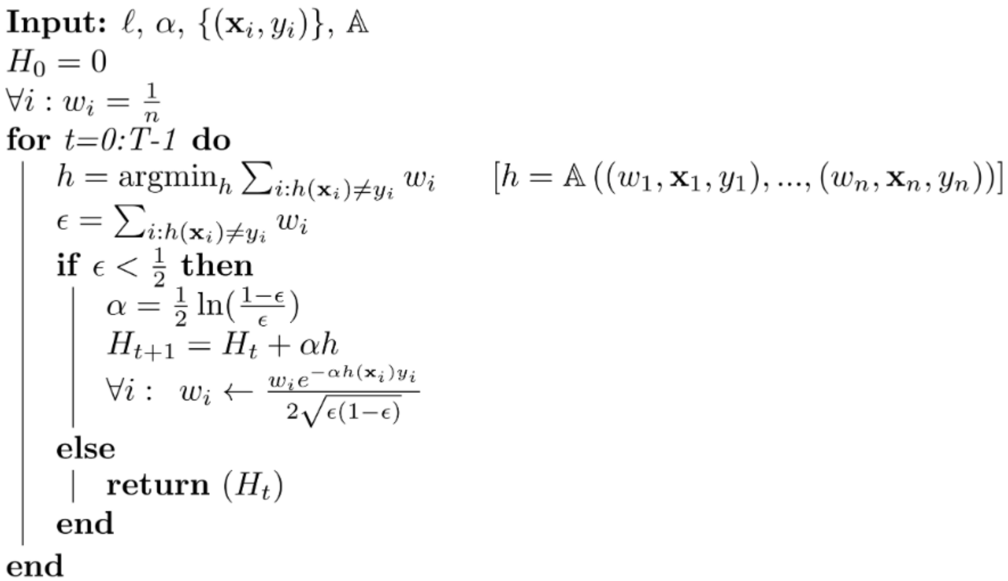
$$Z^{(t+1)} = \sum_{i=1}^{n} e^{-y_i (H(x_i)+\alpha h(x_i))} = Z^{(t)} \cdot \sum_{i=1}^{n} \frac{1}{Z^{(t)}} e^{-y_i (H(x_i)+\alpha h(x_i))} = Z^{(t)} \cdot \sum_{i=1}^{n} w_i^{(t)} e^{-y_i \alpha h(x_i)}$$
$$= Z \cdot (\sum_{i:h(x_i)y_i=1} w_i^{(t)} e^{-\alpha} + \sum_{i:h(x_i)y_i=-1} w_i^{(t)} e^{\alpha}) = Z^{(t)} \cdot [(1-\epsilon)e^{-\alpha} + \epsilon e^{\alpha}]$$
$$= Z^{(t)} \cdot [(1-\epsilon)\frac{\sqrt{\epsilon}}{\sqrt{1-\epsilon}} + \epsilon \frac{\sqrt{1-\epsilon}}{\sqrt{\epsilon}}] = Z^{(t)} \cdot 2\sqrt{\epsilon(1-\epsilon)}$$

$$Z \leftarrow Z \cdot 2\sqrt{\epsilon(1-\epsilon)}$$

Putting these two together we obtain the following multiplicative update rule:

$$w_i \leftarrow w_i \cdot \frac{e^{-\alpha h(x_i) y_i}}{2\sqrt{\epsilon(1-\epsilon)}}$$

The pseudo-code for AdaBoost will then be the following:



```
Input: $\ell$, $\alpha$, $\{(\mathbf{x}_i, y_i)\}$, $\mathbb{A}$
$H_0 = 0$
$\forall i : w_i = \frac{1}{n}$
for $t$=0:T-1 do
    $h = \text{argmin}_h \sum_{i:h(\mathbf{x}_i)\neq y_i} w_i$        $[h = \mathbb{A}((w_1, \mathbf{x}_1, y_1), ..., (w_n, \mathbf{x}_n, y_n))]$
    $\epsilon = \sum_{i:h(\mathbf{x}_i)\neq y_i} w_i$
    if $\epsilon < \frac{1}{2}$ then
        $\alpha = \frac{1}{2}\ln(\frac{1-\epsilon}{\epsilon})$
        $H_{t+1} = H_t + \alpha h$
        $\forall i : \; w_i \leftarrow \frac{w_i e^{-\alpha h(\mathbf{x}_i) y_i}}{2\sqrt{\epsilon(1-\epsilon)}}$
    else
        return $(H_t)$
    end
end
return $(H_T)$
```

Algorithm 1: AdaBoost in Pseudo-Code

Furthermore, we can use the normalizer Z to bound the loss function after $\mathbf{T}$ iterations:

$$\ell(H) = Z = Z_0 \prod_{t=1}^{T} 2\sqrt{\epsilon_t(1-\epsilon_t)} = n \prod_{t=1}^{T} 2\sqrt{\epsilon_t(1-\epsilon_t)} \quad (Z_0 = n \text{ when all weights are } \frac{1}{n})$$

$$\leq n \cdot (4c(1-c))^{\frac{T}{2}} \qquad (\text{we define } c = max_t\ \epsilon_t)$$

$$\leq n \cdot (4(\frac{1}{4} - \gamma^2))^{\frac{T}{2}} \leq n \cdot (1 - 4\gamma^2)^{\frac{T}{2}} \qquad (max_c\ c(1-c) = \frac{1}{4} \wedge c < \frac{1}{2} \implies \text{this bound: } c(1-c) = \frac{1}{4} - \gamma^2)$$

In other words, $\ell(H) \leq n \cdot (1 - 4\gamma^2)^{\frac{T}{2}}$ tells us that the training loss is **decreasing exponentially!** In fact, we can go even further and compute after how many iterations we must have zero training error (note that the training loss is an upper bound on the training error).
We can compute the number of steps required until the loss is less than 1, which would imply that not a single training sample is misclassified:

$$n \cdot (1 - 4\gamma^2)^{\frac{T}{2}} < 1 \implies T > \frac{2\log(n)}{\log(\frac{1}{1-4\gamma^2})}$$

This is an amazing result! It shows that after $\mathbf{O}(\log \mathbf{n})$ iterations your training error must be zero.

9.2.3 The Margins Explanation for Boosting's Effectiveness

We can visualize the effect AdaBoost has on the margins of the training examples by plotting their distribution. In particular, we can create a plot showing, for each $\theta \in [-1, +1]$, the fraction of training examples with margin at most θ.

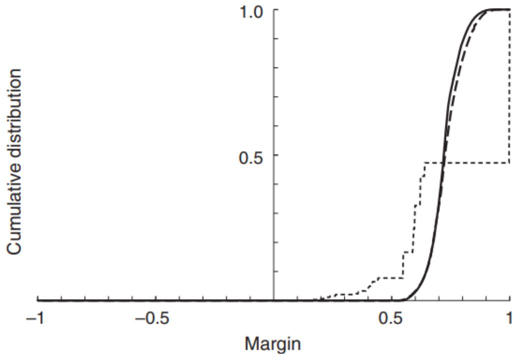


Figure 9.1: The margin distribution graph for boosting showing the cumulative distribution of margins of the training instances after 5, 100, and 1000 iterations, indicated by short-dashed, long-dashed (mostly hidden), and solid curves, respectively.

Whereas nothing at all is happening to the training error, these curves expose dramatic changes happening on the margin distribution. For instance, after five rounds, although the training error is zero (so that no examples have negative margin), a rather substantial fraction of the training examples (7.7%) have margin below 0.5. By round 100, all of these examples have been swept to the right so that not a single example has margin below 0.5, and nearly all have margin above 0.6. (On the other hand, many with margin 1.0 have slipped back to the 0.6–0.8 range.) In line with this trend, the minimum margin of any training example has increased from 0.14 at round 5 to 0.52 at round 100, and 0.55 at round 1000. **Thus, this example is indicative of the powerful effect AdaBoost has on the margins, aggressively pushing up those examples with small or negative margin**.

Indeed, as will be seen, AdaBoost can be analyzed theoretically along exactly these lines. We will first prove a bound on the generalization error of AdaBoost that depends only on the margins of the training examples, and not on the number of rounds of boosting. Thus, this bound predicts that AdaBoost will not overfit regardless of how long it is run, provided that large margins can be achieved (and provided, of course, that the base classifiers are not too complex relative to the size of the training set):

Theorem 9.2 *Let $\mathcal{D}$ be a distribution over $\mathcal{X} \times \{-1, +1\}$, and let $\mathcal{S}$ be a sample of m examples chosen independently at random according to $\mathcal{D}$. Assume that the base classifier space $\mathbb{H}$ is finite, and let $\delta > 0$. Then with probability at least $1 - \delta$ over the random choice of the training set $\mathcal{S}$, every weighted average function f satisfies the following bound:*

$$\mathbb{P}_{\mathcal{D}}[\overbrace{yf(x)}^{margin} \leq 0] \leq \mathbb{P}_{\mathcal{S}}[yf(x) \leq \theta] + O(\sqrt{\frac{\log|\mathbb{H}|}{m\theta^2} \cdot log(\frac{m\theta^2}{\log|\mathbb{H}|}) + \log\frac{1/\delta}{m}})$$

$$\textit{for all } \theta > \sqrt{\frac{log|\mathbb{H}|}{4m}}$$

The term on the left is the generalization error. The first term on the right is the fraction of training examples with margin below some threshold θ. This term will be small if most training examples have large margin (i.e., larger than θ). The second term on the right is an additional term that becomes small as the size of the training set m gets larger, provided the complexity of the base classifiers is controlled for θ bounded away from zero.

The second part of the analysis is to prove that, as observed empirically in Figure 9.1, AdaBoost generally tends to increase the margins of all training examples.

Theorem 9.3 *Given the notation of the previous section, let $\gamma_t = \frac{1}{2} - \epsilon_t$. Then the fraction of training examples with margin at most θ is at most:*

$$\prod_{t=1}^{T} \sqrt{(1 + 2\gamma_t)^{1+\theta}(1 - 2\gamma_t)^{1-\theta}}$$

To get a feeling for this bound, consider what happens when, for all t, $\epsilon_t \leq \frac{1}{2} - \gamma$ for some $\gamma > 0$. Given this assumption, we can simplify the upper bound in theorem 9.3 to:

$$\left(\sqrt{(1 + 2\gamma)^{1+\theta}(1 - 2\gamma)^{1-\theta}}\right)^T$$

When the expression inside the parentheses is strictly smaller than 1, that is, when:

$$\sqrt{(1 + 2\gamma)^{1+\theta}(1 - 2\gamma)^{1-\theta}} < 1 \tag{9.1}$$

this bound implies that the fraction of training examples with margin $\leq \theta$ decreases to zero exponentially fast with T. Moreover, by solving for θ, we see that equation (9.1) holds if and only if:

$$\theta < -\frac{\log{(1-4\gamma^2)}}{\log{(\dfrac{1+2\gamma}{1-2\gamma})}}$$

Thus, the margins of the training examples are guaranteed to be large after a sufficient number of boosting iterations.