

## Combining classifiers, boosting

We can combine any set of classifiers into an ensemble where each classifier votes for one label versus another. Ensembles can be useful even if generated through randomization. For example, we can generate random subsets of smaller training sets from the original one, and train a classifier, e.g., an SVM, based on each such set. The outputs of the SVM classifiers, trained with slightly different training sets, can be then combined into an ensemble classifier where each SVM is given an equal vote. This randomization, known as *bagging*, is a method of reducing *variance* of the resulting classifier. In other words, the ensemble provides more stable predictions and therefore often generalizes better. Weighting each SVM output equally in the ensemble, regardless of how well they solve their corresponding subtasks, does not help reduce *bias*. Put another way, bagging does not lead to a more complex classifier (closer to the true underlying relation), just one that is more stable.

Here we will explore methods such as *boosting* for combining simple base classifiers into a stronger (more complex) ensemble. Note that the process of combining simple “weak” classifiers into one “strong” classifier is analogous to the use of kernels to go from a simple linear classifier to a non-linear classifier. The difference is that here we are *learning* a small number of highly *non-linear features* from the inputs rather than using a fixed process of generating a large number of features from the inputs as in the polynomial kernel.

We can combine any set of classifiers into an ensemble. Here we will use the simplest possible components, the decision stumps:

$$h(\underline{x}; \theta) = \text{sign}(\theta_1^s x_k + \theta_0^s) \quad (1)$$

where  $\theta = (k, \theta_1^s, \theta_0^s)$ , i.e., the parameters include the coordinate  $k$  that the stump depends on as well as the two parameters  $\theta_1^s$  and  $\theta_0^s$  to encode the location and the direction (positive side) of the stump. Figure 1 below shows a possible decision boundary (horizontal) when the input vectors  $\underline{x}$  are two dimensional.

The boosting algorithm finds and combines stumps (as base learners) into an ensemble such that, after  $m$  rounds of boosting, the ensemble classifier has the following form

$$h_m(\underline{x}) = \sum_{j=1}^m \alpha_j h(\underline{x}; \theta_j) \quad (2)$$

where  $\alpha_j \geq 0$ . We can always normalize the “votes”  $\alpha_j$  such that  $\sum_{j=1}^m \alpha_j = 1$ . The ensemble can be viewed as a voting combination. Given  $\underline{x}$ , each stump votes for a label  $h(\underline{x}; \theta_j)$  using  $\alpha_j$  votes allocated to it. The ensemble then classifies the example according

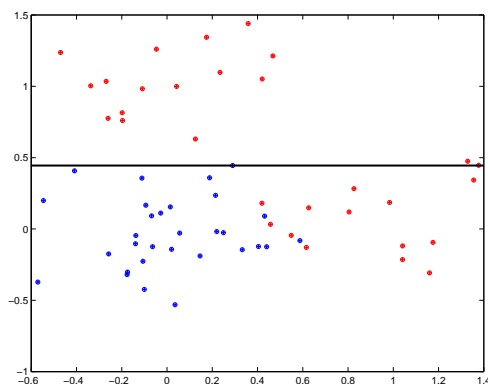


Figure 1: A possible decision boundary from a trained decision stump. The stump in the figure depends only on the vertical  $x_2$ -axis.

to which label receives the most votes. Note that  $h_m(\underline{x}) \in [-1, 1]$  whenever the votes are normalized to sum to one. So,  $h_m(x) = 1$  only if all the stumps agree that the label should be  $y = 1$ .

The ensemble can also be viewed as a linear classifier based on a feature vector

$$\phi(\underline{x}; \theta) = [h(\underline{x}; \theta_1), \dots, h(\underline{x}; \theta_m)]^T \quad (3)$$

The linear parameters (the votes) we learn are constrained to be positive  $\alpha_j \geq 0$ . As a result,  $h_m(\underline{x}) = \phi(\underline{x}; \theta) \cdot \underline{\alpha}$ , which is linear in  $\underline{\alpha}$ . However, the key difference here is that we learn both which features to include, i.e., the “coordinates”  $h(\underline{x}; \theta_j)$  in the feature vector, as well as how they are combined, the  $\alpha$ ’s. This is a difficult learning problem to solve.

## Training ensembles, boosting

Now, let’s figure out how to train ensembles. We will need a loss function and there many possibilities, including the logistic loss. For simplicity, we will use the *exponential loss*:

$$\text{Loss}(y, h(\underline{x})) = \exp(-yh(\underline{x})) \quad (4)$$

The loss is small if the ensemble classifier agrees with the label  $y$  (the stronger the agreement, the smaller the loss is). The loss is large if the ensemble strongly disagrees with the label. This is a common loss function in boosting.

The simplest way to optimize the ensemble is to do it sequentially in stages (a.k.a. forward fitting). In other words, we will first find a single stump (an ensemble of one), then add another while keeping the first one fixed, and so on, never retraining those already added into the ensemble. To facilitate this type of estimation, we will assume that  $\alpha_j \geq 0$  but won't require that they will sum to one (we can always renormalize the votes after having trained the ensemble).

Suppose now that we have already added  $m - 1$  base learners into the ensemble and call this ensemble  $h_{m-1}(\underline{x})$ . This part will be fixed for the purpose of adding  $\alpha_m h(\underline{x}; \theta_m)$ . We can then try to minimize the training loss corresponding to the ensemble

$$h_m(\underline{x}) = \sum_{j=1}^{m-1} \hat{\alpha}_j h(\underline{x}; \hat{\theta}_j) + \alpha_m h(\underline{x}; \theta_m) \quad (5)$$

$$= h_{m-1}(\underline{x}) + \alpha_m h(\underline{x}; \theta_m) \quad (6)$$

with respect to  $\alpha_m$  and  $\theta_m$ . To this end, let's write

$$J(\alpha_m, \theta_m) = \sum_{t=1}^n \text{Loss}(y_t, h_m(\underline{x}_t)) \quad (7)$$

$$= \sum_{t=1}^n \exp\left(-y_t h_{m-1}(\underline{x}_t) - y_t \alpha_m h(\underline{x}_t; \theta_m)\right) \quad (8)$$

$$= \sum_{t=1}^n \overbrace{\exp\left(-y_t h_{m-1}(\underline{x}_t)\right)}^{W_{m-1}(t)} \exp\left(-y_t \alpha_m h(\underline{x}_t; \theta_m)\right) \quad (9)$$

$$= \sum_{t=1}^n W_{m-1}(t) \exp\left(-y_t \alpha_m h(\underline{x}_t; \theta_m)\right) \quad (10)$$

In other words, for the purpose of estimating the new base learner, all we need to know from the previous ensemble are the weights  $W_{m-1}(t)$  associated with the training examples. These weights are exactly the losses of the  $m - 1$  ensemble on each of the training example. Thus, the new base learner will be “directed” towards examples that were misclassified by the  $m - 1$  ensemble  $h_{m-1}(\underline{x})$ .

The estimation problem that couples  $\alpha_m$  and  $\theta_m$  is still a bit difficult. We will simplify this further by figuring out how to estimate  $\theta_m$  first and then decide how many votes  $\alpha_m$  we should assign to the new base learner (i.e., how much we should rely on its predictions). But what is the criterion for  $\theta_m$  independent of  $\alpha_m$ ? Consider as a thought experiment

that we calculate for all possible  $\theta_m$  the derivative

$$\frac{d}{d\alpha_m} J(\alpha_m, \theta_m) \Big|_{\alpha_m=0} = - \sum_{t=1}^m W_{m-1}(t) y_t h(\underline{x}_t; \theta_m) \quad (11)$$

This derivative tells us how much we can reduce the overall loss by increasing the vote (from zero) of the new base learner with parameters  $\theta_m$ . This derivative is expected to be negative so that the training loss is decreased by adding the new base learner. It makes sense then to find a base learner  $h(\underline{x}; \theta_m)$ , parameters  $\theta_m$ , so as to minimize this derivative (making it as negative as possible). Such a base learner would be expected to lead to a large reduction of the training loss. Once we have this  $\hat{\theta}_m$  we can subsequently optimize the training loss  $J(\alpha_m, \hat{\theta}_m)$  with respect to  $\alpha_m$  for a fixed  $\hat{\theta}_m$ .

We have now essentially all the components to define the *Adaboost algorithm*. We will make one modification which is that the weights will be normalized to sum to one. This is advantageous as they can become rather small in the course of adding the base learners. The normalization won't change the optimization of  $\theta_m$  nor  $\alpha_m$ . We denote the normalized weights with  $\tilde{W}_{m-1}(t)$ . The boosting algorithm is now defined as

(0) Set  $\tilde{W}_0(t) = 1/n$  for  $t = 1, \dots, n$ .

(1) At stage  $m$ , find a base learner  $h(\underline{x}; \hat{\theta}_m)$  that approximately minimizes

$$- \sum_{t=1}^n \tilde{W}_{m-1}(t) y_t h(\underline{x}_t; \theta_m) = 2\epsilon_m - 1 \quad (12)$$

where  $\epsilon_m$  is the weighted classification error (zero-one loss) on the training examples, weighted by the normalized weights  $\tilde{W}_{m-1}(t)$ .

(2) Given  $\hat{\theta}_m$ , set

$$\hat{\alpha}_m = 0.5 \log \left( \frac{1 - \hat{\epsilon}_m}{\hat{\epsilon}_m} \right) \quad (13)$$

where  $\hat{\epsilon}_m$  is the weighted error corresponding to  $\hat{\theta}_m$  chosen in step (1). For binary  $\{-1, 1\}$  base learners,  $\hat{\alpha}_m$  exactly minimizes the weighted training loss (loss of the ensemble):

$$J(\alpha_m, \hat{\theta}_m) = \sum_{t=1}^n \tilde{W}_{m-1}(t) \exp \left( -y_t \alpha_m h(\underline{x}_t; \hat{\theta}_m) \right) \quad (14)$$

In cases where the base learners are not binary (e.g., return values in the interval  $[-1, 1]$ ), we would have to minimize Eq.(14) directly.

- (3) Update the weights on the training examples based on the new base learner:

$$\tilde{W}_m(t) = c_m \cdot \tilde{W}_{m-1}(t) \exp\left(-y_t \hat{\alpha}_m h(\underline{x}_t; \hat{\theta}_m)\right) \quad (15)$$

where  $c_m$  is the normalization constant to ensure that  $\tilde{W}_m(t)$  sum to one after the update. The new weights can be again interpreted as normalized losses for the new ensemble  $h_m(\underline{x}_t) = h_{m-1}(\underline{x}_t) + \hat{\alpha}_m h(\underline{x}_t; \hat{\theta}_m)$ .

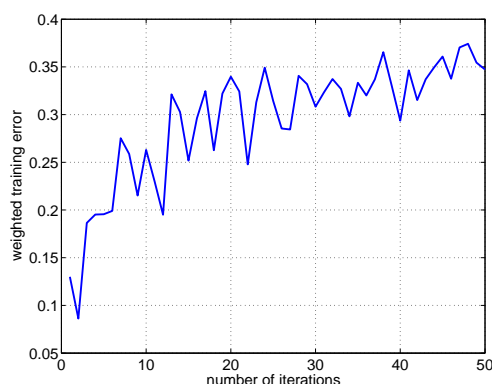
The Adaboost algorithm sequentially adds base learners to the ensemble so as to decrease the training loss.

## Understanding boosting

Let's try to understand the boosting algorithm from several different perspectives. First of all, there are several different types of errors (errors here refer to zero-one classification losses, not the surrogate exponential losses). We can talk about the weighted error of base learner  $m$ , introduced at the  $m^{\text{th}}$  boosting iteration, relative to the weights  $\tilde{W}_{m-1}(t)$  on the training examples. This is the weighted training error  $\hat{\epsilon}_m$  in the algorithm. We can also measure the weighted error of the same base classifier relative to the updated weights, i.e., relative to  $\tilde{W}_m(t)$ . In other words, we can measure how well the current base learner would do at the next iteration. Finally, in terms of the ensemble, we have the unweighted training error and the corresponding generalization (test) error, as a function of boosting iterations. We will discuss each of these in turn.

**Weighted error.** The weighted error achieved by a new base learner  $h(\underline{x}; \hat{\theta}_m)$  relative to  $\tilde{W}_{m-1}(t)$  tends to increase with  $m$ , i.e., with each boosting iteration (though not monotonically). Figure 2 below shows this weighted error  $\hat{\epsilon}_m$  as a function of boosting iterations. The reason for this is that since the weights concentrate on examples that are difficult to classify correctly, subsequent base learners face harder classification tasks.

**Weighted error relative to updated weights.** We claim that the weighted error of the base learner  $h(\underline{x}; \hat{\theta}_m)$  relative to the updated weights  $\tilde{W}_m(t)$  is exactly 0.5. This means that the base learner introduced at the  $m^{\text{th}}$  boosting iteration will be useless (at chance level) for the next boosting iteration. So the boosting algorithm would never introduce the same base learner twice in a row. The same learner might, however, reappear later on (relative

Figure 2: Weighted error  $\hat{\epsilon}_m$  as a function of  $m$ .

to a different set of weights). One reason for this is that we don't go back and update  $\hat{\alpha}_j$ 's for base learners already introduced into the ensemble. So the only way to change the previously set coefficients is to reintroduce the base learners. Let's now see that the claim is indeed true. We can equivalently show that the *weighted agreement* relative to  $\tilde{W}_m(t)$  is exactly zero:

$$\sum_{t=1}^n \tilde{W}_m(t) y_t h(\underline{x}_t; \hat{\theta}_m) = 0 \quad (16)$$

Consider the optimization problem for  $\alpha_m$  after we have already found  $\hat{\theta}_m$ :

$$J(\alpha_m, \hat{\theta}_m) = \sum_{t=1}^n \tilde{W}_{m-1}(t) \exp\left(-y_t \alpha_m h(\underline{x}_t; \hat{\theta}_m)\right) \quad (17)$$

The derivative of  $J(\alpha_m, \hat{\theta}_m)$  with respect to  $\alpha_m$  has to be zero at the optimal value  $\hat{\alpha}_m$  so that

$$\frac{d}{d\alpha_m} J(\alpha_m, \hat{\theta}_m) \Big|_{\alpha_m = \hat{\alpha}_m} = - \sum_{t=1}^n \tilde{W}_{m-1}(t) \exp\left(-y_t \hat{\alpha}_m h(\underline{x}_t; \hat{\theta}_m)\right) y_t h(\underline{x}_t; \hat{\theta}_m) \quad (18)$$

$$= -c_m \sum_{t=1}^n \tilde{W}_m(t) y_t h(\underline{x}_t; \hat{\theta}_m) = 0 \quad (19)$$

where we have used Eq.(15) to move from  $\tilde{W}_{m-1}(t)$  to  $\tilde{W}_m(t)$ . So the result is an optimality condition for  $\alpha_m$ .

**Ensemble training error.** The training error of the ensemble does not necessarily decrease monotonically with each boosting iteration. The exponential loss of the ensemble does, however, decrease monotonically. This should be evident since it is exactly the loss we are sequentially minimizing by adding the base learners. We can also quantify, based on the weighted error achieved by each base learner, how much the exponential loss decreases after each iteration. We will need this to relate the training loss to the classification error. In fact, the amount that the training loss decreases after iteration  $m$  is exactly  $c_m$ , the normalization constant for the updated weights (we have to normalize the weights precisely because the exponential loss over the training examples decreases). Note also that  $c_m$  is exactly  $J(\hat{\alpha}_m, \hat{\theta}_m)$ . Now,

$$J(\hat{\alpha}_m, \hat{\theta}_m) = \sum_{t=1}^n \tilde{W}_{m-1}(t) \exp\left(-y_t \hat{\alpha}_m h(\underline{x}_t; \hat{\theta}_m)\right) \quad (20)$$

$$= \sum_{t: y_t = h(\underline{x}_t; \hat{\theta}_m)} \tilde{W}_{m-1}(t) \exp(-\hat{\alpha}_m) + \sum_{t: y_t \neq h(\underline{x}_t; \hat{\theta}_m)} \tilde{W}_{m-1}(t) \exp(\hat{\alpha}_m) \quad (21)$$

$$= (1 - \hat{e}_m) \exp(-\hat{\alpha}_m) + \hat{e}_m \exp(\hat{\alpha}_m) \quad (22)$$

$$= (1 - \hat{e}_m) \sqrt{\frac{\hat{e}_m}{1 - \hat{e}_m}} + \hat{e}_m \sqrt{\frac{1 - \hat{e}_m}{\hat{e}_m}} \quad (23)$$

$$= 2\sqrt{\hat{e}_m(1 - \hat{e}_m)} \quad (24)$$

Note that this is always less than one for any  $\hat{e}_m < 1/2$ . The training loss of the ensemble after  $m$  boosting iterations is exactly the product of these terms (renormalizations). In other words,

$$\frac{1}{n} \sum_{t=1}^n \exp(-y_t h_m(\underline{x}_t)) = \prod_{k=1}^m 2\sqrt{\hat{e}_k(1 - \hat{e}_k)} \quad (25)$$

This and the observation that

$$\text{step}(z) \leq \exp(z) \quad (26)$$

for all  $z$ , where the step function  $\text{step}(z) = 1$  if  $z > 0$  and zero otherwise, suffices for our purposes. A simple upper bound on the training error of the ensemble,  $R_n(h_m)$ , follows

from

$$R_n(h_m) = \frac{1}{n} \sum_{t=1}^n \text{step}(-y_t h_m(\underline{x}_t)) \quad (27)$$

$$\leq \frac{1}{n} \sum_{t=1}^n \exp(-y_t h_m(\underline{x}_t)) \quad (28)$$

$$= \prod_{k=1}^m 2\sqrt{\hat{\epsilon}_k(1 - \hat{\epsilon}_k)} \quad (29)$$

Thus the exponential loss over the training examples is an upper bound on the training error and this upper bound goes down monotonically with  $m$  provided that the base learners are learning something at each iteration (their weighted errors less than half). Figure 3 shows the training error as well as the upper bound as a function of the boosting iterations.

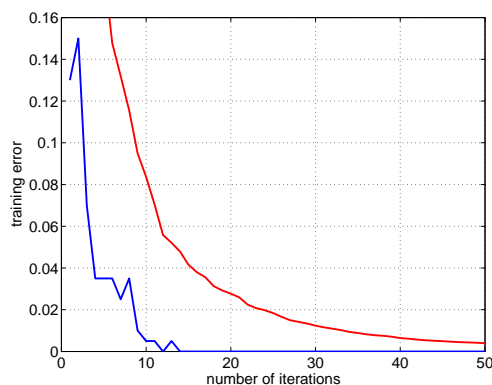


Figure 3: The training error of the ensemble as well as the corresponding exponential loss (upper bound) as a function of the boosting iterations.

**Ensemble test error.** We have so far discussed only training errors. The goal, of course, is to generalize well. What can we say about the generalization error of ensemble generated by the boosting algorithm? We have repeatedly tied the generalization error to some notion of margin. The same is true here. Consider figure 5 below. It shows a typical plot of the ensemble training error and the corresponding generalization error as a function of boosting iterations. Two things are notable in the plot. First, the generalization error seems to decrease (slightly) even after the ensemble has reached zero training error. Why should this be? The second surprising thing seems to be the fact that the generalization

error does not increase even after a large number of boosting iterations. In other words, the boosting algorithm appears to be somewhat resistant to overfitting. Let's try to explain these two (related) observations.

The votes  $\{\hat{\alpha}_j\}$  generated by the boosting algorithm won't sum to one. We will therefore renormalize ensemble

$$\tilde{h}_m(\underline{x}) = \frac{\hat{\alpha}_1 h(\underline{x}; \hat{\theta}_1) + \dots + \hat{\alpha}_m h(\underline{x}; \hat{\theta}_m)}{\hat{\alpha}_1 + \dots + \hat{\alpha}_m} \quad (30)$$

so that  $\tilde{h}_m(\underline{x}) \in [-1, 1]$ . As a result, we can define a "voting margin" for training examples as  $\text{margin}(t) = y_t \tilde{h}_m(\underline{x}_t)$ . The margin is positive if the example is classified correctly by the ensemble. It represents the degree to which the base classifiers agree with the correct classification decision (negative value indicates disagreement). Note that  $\text{margin}(t) \in [-1, 1]$ . It is a very different type of margin (voting margin) than the geometric margin we have discussed in the context linear classifiers. Now, in addition to the training error  $R_n(h_m)$  we can define a margin error  $R_n(h_m; \rho)$  that is the fraction of example margins that are at or below the threshold  $\rho$ . Clearly,  $R_n(h_m) = R_n(h_m; 0)$ . We now claim that the boosting algorithm, even after  $R_n(h_m; 0) = 0$  will decrease  $R_n(h_m; \rho)$  for larger values of  $\rho > 0$ . Figure 4a-b provide an empirical illustration that this is indeed happening. This is perhaps easy to understand as a consequence of the fact that exponential loss,  $\exp(-\text{margin}(t))$ , decreases as a function of the margin, even after the margin is positive.

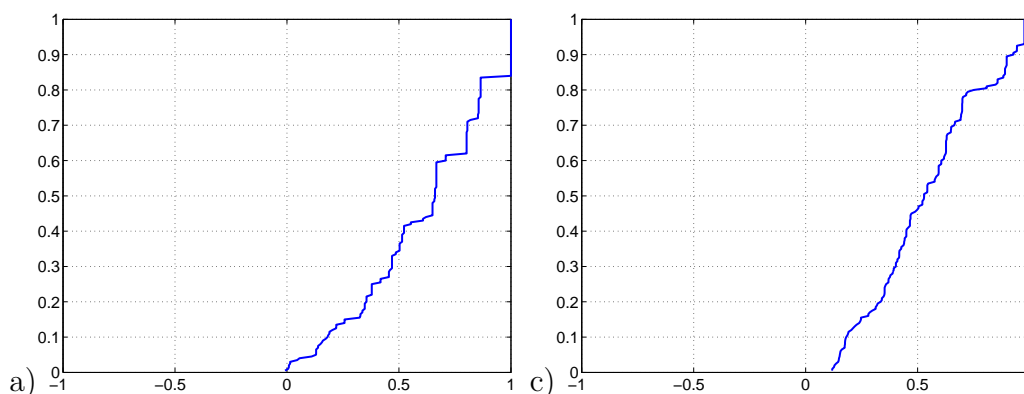


Figure 4: The margin errors  $R_n(h_m; \rho)$  as a function of  $\rho$  when a)  $m = 10$  b)  $m = 50$ .

The second issue to explain is the apparent resistance to overfitting. One reason is that the complexity of the ensemble does not increase very quickly as a function of the number of

base learners. We will make this statement more precise later on. Moreover, the boosting iterations modify the ensemble in sensible ways (increasing the margin) even after the training error is zero. We can also relate the margin, or the margin error  $R_n(h_m; \rho)$  directly to generalization error. Another reason for resistance to overfitting is that the sequential procedure for optimizing the exponential loss is not very effective. We would overfit much more quickly if we reoptimized  $\{\alpha_j\}$ 's *jointly* rather than through the sequential procedure (see the discussion of boosting as gradient descent below).

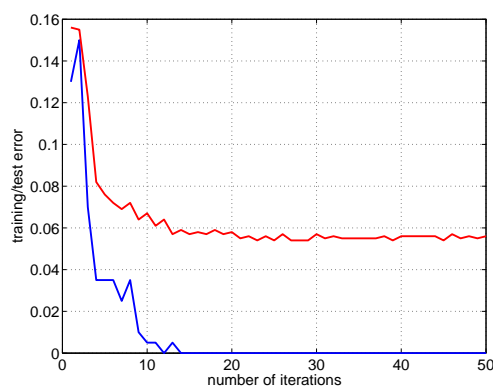


Figure 5: The training error of the ensemble as well as the corresponding generalization error as a function of boosting iterations.

**Boosting as gradient descent.** We can also view the boosting algorithm as a simple gradient descent procedure (with line search) in the space of discriminant functions. To understand this we can view each base learner  $h(\underline{x}; \theta)$  as a vector based on evaluating it on each of the training examples:

$$\vec{h}(\theta) = \begin{bmatrix} h(\underline{x}_1; \theta) \\ \vdots \\ h(\underline{x}_n; \theta) \end{bmatrix} \quad (31)$$

The ensemble vector  $\vec{h}_m$ , obtained by evaluating  $h_m(\underline{x})$  at each of the training examples, is a positive combination of the base learner vectors:

$$\vec{h}_m = \sum_{j=1}^m \hat{\alpha}_m \vec{h}(\hat{\theta}_m) \quad (32)$$

The exponential loss objective we are trying to minimize is now a function of the ensemble vector  $\vec{h}_m$  and the training labels. Suppose we have  $\vec{h}_{m-1}$ . To minimize the objective, we

can select a useful direction,  $\vec{h}(\hat{\theta}_m)$ , along which the objective seems to decrease. This is exactly how we derived the base learners. We can then find the minimum of the objective by moving in this direction, i.e., evaluating vectors of the form  $\vec{h}_{m-1} + \alpha_m \vec{h}(\hat{\theta}_m)$ . This is a line search operation. The minimum is attained at  $\hat{\alpha}_m$ , we obtain  $\vec{h}_m = \vec{h}_{m-1} + \hat{\alpha}_m \vec{h}(\hat{\theta}_m)$ , and the procedure can be repeated.

Viewing the boosting algorithm as a simple gradient descent procedure also helps us understand why it can overfit if we continue with the boosting iterations.