CS181 Lecture 4 — Committees and Boosting

Avi Pfeffer; Revised by David Parkes

Feb 1, 2011

First we try to understand how the error of a learning algorithm can be decomposed into two components: the bias, or systematic error of the algorithm, and the variance, or error due to fluctuation in the prediction of the algorithm on different training sets. Using this approach, we look at learning committees of classifiers, and in particular consider the bagging and boosting methods. For boosting we consider the margin or "decisiveness" of the committee hypothesis to gain an explanation for its remarkable robustness against overfitting.

1 The Bias-Variance Decomposition

Consider some learning algorithm $\mathcal{L}$ that learns a hypothesis h from a training set $\mathcal{D}$.

We will assume that there is a single probability distribution P from which all training and test instances are independently drawn. This assumption enshrines the idea that the "future resembles the past," which is necessary for any kind of learning to be possible.

To be precise, $P(X, Y)$ is a joint distribution over instances $(\mathbf{x}, y) \in (X, Y)$. Equivalently, we can view P as defining a distribution $P(X)$ and a conditional distribution $P(Y|X)$, with $P(X, Y) = P(X)P(Y|X)$.

We focus here on *deterministic* domains, for which there is a true model $f : X \to Y$, so that for every $\mathbf{x}$, there is some $y \in Y$ $(y = f(\mathbf{x}))$ such that $P(y|\mathbf{x}) = 1$.

Given this, the expected error of hypothesis h is defined as:

$$error_P(h) = P_{\mathbf{x},y}(h(\mathbf{x}) \neq y), \tag{1}$$

Equivalently, considering only the marginal distribution on X, which we write $P_{\mathbf{x}}$, then we have

$$error_P(h) = P_{\mathbf{x}}(h(\mathbf{x}) \neq f(\mathbf{x})). \tag{2}$$

In particular, if the possible classes are labeled $Y = \{0, 1\}$, then

$$error_P(h) = \mathbb{E}_{\mathbf{x}}[(h(\mathbf{x}) - f(\mathbf{x}))^2], \tag{3}$$

where $\mathbb{E}_{\mathbf{x}}[\cdot]$ means expectation over all possible values of $\mathbf{x}$, taken with respect to the distribution $P_{\mathbf{x}}$, and $(h(\mathbf{x}) - f(\mathbf{x}))^2$ will be 1 precisely when $h(\mathbf{x}) \neq f(\mathbf{x})$, and 0 otherwise.

Let $h_{\mathcal{L}}(\mathbf{x}; \mathcal{D})$ denote the hypothesis generated by training learner $\mathcal{L}$ on data $\mathcal{D}$, in particular when evaluated on example $\mathbf{x}$ (so that $h_{\mathcal{L}}(\mathbf{x}; \mathcal{D}) \in Y$).

We are interested in studying the error of a learning algorithm $\mathcal{L}$ in regard to a distribution on possible training data, where there are n examples and each example is sampled according to P. That is, the expected error of a learning algorithm $\mathcal{L}$, for a given true concept f, is

$$error(\mathcal{L}) = \mathbb{E}_{\mathcal{D}}[\mathbb{E}_{\mathbf{x}}[(h_{\mathcal{L}}(\mathbf{x}; \mathcal{D}) - f(\mathbf{x}))^2]]. \tag{4}$$

Here, $\mathbb{E}_{\mathcal{D}}[\cdot]$ denotes the expectation over all possible training sets $\mathcal{D}$ of size n. In words, this is the expected (generalization) error of the algorithm when trained on data $\mathcal{D}$ sampled from the same distribution as that on which it is tested.

1.1 Bias-Variance Tradeoff

We will now analyze this expression by introducing the quantity $\mathbb{E}_{\mathcal{D}}[h_{\mathcal{L}}(\mathbf{x};\mathcal{D})]$. This is the expected prediction of the hypothesis h learned by $\mathcal{L}$ on a *particular instance* $\mathbf{x}$. Again the expectation is taken over training sets. Naturally we would like this quantity to be close to $f(x)$.

We now proceed as follows. The derivation uses several properties of expectation. Two important ones are *linearity of expectation* which means that you can distribute expectation through sums, and the fact that the expectation of the expectation of $\mathbf{x}$ is equal to the expectation of $\mathbf{x}$. Fixing $\mathbf{x}$, then $\mathbb{E}_{\mathcal{D}}[(h_{\mathcal{L}}(\mathbf{x};\mathcal{D}) - f(\mathbf{x}))^2] =$

$$
\begin{aligned}
&\mathbb{E}_{\mathcal{D}}[\{h_{\mathcal{L}}(\mathbf{x};\mathcal{D}) - \mathbb{E}_{\mathcal{D}}[h_{\mathcal{L}}(\mathbf{x};\mathcal{D})] + \mathbb{E}_{\mathcal{D}}[h_{\mathcal{L}}(\mathbf{x};\mathcal{D})] - f(\mathbf{x})\}^2] \\
&= \mathbb{E}_{\mathcal{D}}[(h_{\mathcal{L}}(\mathbf{x};\mathcal{D}) - \mathbb{E}_{\mathcal{D}}[h_{\mathcal{L}}(\mathbf{x};\mathcal{D})])^2 + 2(h_{\mathcal{L}}(\mathbf{x};\mathcal{D}) - \mathbb{E}_{\mathcal{D}}[h_{\mathcal{L}}(\mathbf{x};\mathcal{D})])(\mathbb{E}_{\mathcal{D}}[h_{\mathcal{L}}(\mathbf{x};\mathcal{D})] - f(\mathbf{x})) \\
&\quad + (\mathbb{E}_{\mathcal{D}}[h_{\mathcal{L}}(\mathbf{x};\mathcal{D})] - f(\mathbf{x}))^2] \\
&= \mathbb{E}_{\mathcal{D}}[(h_{\mathcal{L}}(\mathbf{x};\mathcal{D}) - \mathbb{E}_{\mathcal{D}}[h_{\mathcal{L}}(\mathbf{x};\mathcal{D})])^2] + 2\mathbb{E}_{\mathcal{D}}[h_{\mathcal{L}}(\mathbf{x};\mathcal{D})\mathbb{E}_{\mathcal{D}}[h_{\mathcal{L}}(\mathbf{x};\mathcal{D})]] \\
&\quad - 2\mathbb{E}_{\mathcal{D}}[\mathbb{E}_{\mathcal{D}}[h_{\mathcal{L}}(\mathbf{x};\mathcal{D})]\mathbb{E}_{\mathcal{D}}h_{\mathcal{L}}(\mathbf{x};\mathcal{D})] - 2\mathbb{E}_{\mathcal{D}}[h_{\mathcal{L}}(\mathbf{x};\mathcal{D})f(\mathbf{x})] + 2\mathbb{E}_{\mathcal{D}}[\mathbb{E}_{\mathcal{D}}[h_{\mathcal{L}}(\mathbf{x};\mathcal{D})]f(\mathbf{x})] + \\
&\quad \mathbb{E}_{\mathcal{D}}[(\mathbb{E}_{\mathcal{D}}[h_{\mathcal{L}}(\mathbf{x};\mathcal{D})] - f(\mathbf{x}))^2] \\
&= \mathbb{E}_{\mathcal{D}}[(h_{\mathcal{L}}(\mathbf{x};\mathcal{D}) - \mathbb{E}_{\mathcal{D}}[h_{\mathcal{L}}(\mathbf{x};\mathcal{D})])^2] + 2(\mathbb{E}_{\mathcal{D}}[h_{\mathcal{L}}(\mathbf{x};\mathcal{D})])^2 - 2(\mathbb{E}_{\mathcal{D}}[h_{\mathcal{L}}(\mathbf{x};\mathcal{D})])^2 \\
&\quad - (2\mathbb{E}_{\mathcal{D}}[h_{\mathcal{L}}(\mathbf{x};\mathcal{D})f(\mathbf{x})] - 2\mathbb{E}_{\mathcal{D}}[h_{\mathcal{L}}(\mathbf{x};\mathcal{D})f(\mathbf{x})]) + \mathbb{E}_{\mathcal{D}}[(\mathbb{E}_{\mathcal{D}}[h_{\mathcal{L}}(\mathbf{x};\mathcal{D})] - f(\mathbf{x}))^2] \\
&= \mathbb{E}_{\mathcal{D}}[(h_{\mathcal{L}}(\mathbf{x};\mathcal{D}) - \mathbb{E}_{\mathcal{D}}[h_{\mathcal{L}}(\mathbf{x};\mathcal{D})])^2] + \mathbb{E}_{\mathcal{D}}[(\mathbb{E}_{\mathcal{D}}[h_{\mathcal{L}}(\mathbf{x};\mathcal{D})] - f(\mathbf{x}))^2] \\
&= \mathbb{E}_{\mathcal{D}}[(h_{\mathcal{L}}(\mathbf{x};\mathcal{D}) - \mathbb{E}_{\mathcal{D}}[h_{\mathcal{L}}(\mathbf{x};\mathcal{D})])^2] + (\mathbb{E}_{\mathcal{D}}[h_{\mathcal{L}}(\mathbf{x};\mathcal{D})] - f(\mathbf{x}))^2
\end{aligned}
$$

Substituting back into (4), and remembering that we can switch the order of expectation operators, we get

$$
\begin{aligned}
error(\mathcal{L}) = &\mathbb{E}_{\mathbf{x}}[\mathbb{E}_{\mathcal{D}}[(h_{\mathcal{L}}(\mathbf{x};\mathcal{D}) - \mathbb{E}_{\mathcal{D}}[h_{\mathcal{L}}(\mathbf{x};\mathcal{D})])^2]] \\
&+ \mathbb{E}_{\mathbf{x}}[(\mathbb{E}_{\mathcal{D}}[h_{\mathcal{L}}(\mathbf{x};\mathcal{D})] - f(\mathbf{x}))^2]
\end{aligned} \tag{5}
$$

Look closely at this formula. The generalization error of a learning algorithm for the classification problem consists of two terms. This is the *bias-variance decomposition.*

The first term takes the expectation of the *variance* of h, taken over all instances $\mathbf{x}$. *How sensitive is the learner to details of the training data?* (Recall that the variance of random variable Z is defined by $\mathbb{E}[(Z - \mathbb{E}[Z])^2]$, and is a measure of how far Z tends to stray from its average value). Thus, this term measures how much the predictions on point $\mathbf{x}$ vary as a result of applying the learner to different data. How much does the prediction of a hypothesis fluctuate as the training data is varied?

The second term takes the expected squared *bias* of h, where the bias of h is defined to be $\mathbb{E}_{\mathcal{D}}[h_{\mathcal{L}}(\mathbf{x};\mathcal{D})] - f(\mathbf{x})$, i.e., the difference between the average prediction and the true prediction. *Does the learner have a systematic error even when averaged over multiple independent training data samples?* The squared bias term measures the systematic (or *persistent*) error due to the learning algorithm, that would occur even the prediction is averaged over the result of applying $\mathcal{L}$ to many different data sets.

This decomposition suggests that if we want to improve the generalization performance of a learning algorithm, we have two paths to follow: *reduce the bias, or reduce the variance.*

Note: This statistical use of the word "bias" is distinct from what what we called "inductive bias." However, the two are related. If you have strong inductive bias, you will likely also have large statistical bias. On the other hand, if you have strong inductive bias you will likely also have smaller variance, and so we can begin to see that this bias-variance decomposition coincides with our understanding of the problem of overfitting, and the task of ensuring good generalization performance.

In fact, it is typical that we see a *bias-variance tradeoff* in designing learning algorithms. As the bias improves the variance will often deteriorate (and vice versa.)

Consider for example what happens with a strong inductive bias. Suppose your hypothesis space consists only of conjunctive Boolean concepts. Then you will have a statistical bias for any true concept that is not a conjunctive concept. On the other hand, your variance will be small. Because the hypothesis space is so

impoverished, variation in the training data will have little effect on the learning algorithm. As you widen the hypothesis space and weaken the restriction bias, your statistical bias will decrease, but your algorithm will be more susceptible to noise and variation in the training data, and so the variance will increase. The trick is to find the best point in the tradeoff, where both are small.

For decision trees, it is well known that ID3 and its extensions (e.g., C4.5 that introduces post-pruning methods and other tricks) have high variance, with learned trees that are very sensitive to changing a single training instance. This can be combatted by insisting on decision stumps (depth one trees with a single branching decision), but while enjoying lower variance the hypotheses in this class have higher bias.

This leads to a question: can we do anything to retain the low variance of "decision stumps" while reducing the bias?

2 Committees (or Ensemble Methods)

One tried and true method for reducing variance without increasing bias much is to use a *committee* or *ensemble* of learners. In this method, instead of learning a single hypothesis, one learns a whole set of them. This could be done by running the same learning algorithm many times with slightly different parameters, or on different training data, or running completely different algorithms. After all the hypotheses in the committee have been learned, classification is performed by having each individual hypothesis classify the instance, and then voting on the final outcome. In the simplest version of this method, each hypothesis has one vote, and the outcome is the majority vote. In more complex versions, each learned hypothesis gets a weight, and the result of the vote is the weighted majority.

One simple way of learning a committee is to use a cross-validation method to generate multiple learners from the same data. For example, in a 10-fold method, each learner might perform validation set pruning, learning a tree from 90% of the data, and pruning with the other 10% that is held-out and used as a validation set. Each of the folds is used once as the validation set. Cross-validation is used here to produce 10 different learners from the same data, and in doing so to form a committee.

Committees tend not to increase bias, because bias is defined as the difference between the average performance of a learner and the true concept, and the average performance of a committee tends to be similar to the average performance of individual members. However, committees do tend to decrease variance, because individual variation between different training sets tends to be smoothed out in the voting process. Even if one hypothesis learned something spurious from its training set, this spurious pattern might not have been present in the other training sets and thus out-voted.

But for a committee to reduce variance relies on an important assumption, namely, that the individual hypotheses learned by the different learners in the committee be somewhat independent of each other. The reason is that when they are not independent, then when one committee member overfits to a spurious pattern in the training data, other members are also likely to do so because they see similar data, and so the variance does not get reduced.

How well does the independence assumption hold in the case of the ten decision trees constructed through cross-validation? On the one hand, any two trees share 9/10 of their training data, and so will tend to be similar. However, all is not lost, since the validation set is different for every tree, and the validation set is crucial for pruning away parts of the tree that are not relevant. In fact, a tree will only model spurious patterns if they are present in both its training set and its validation set. So the analysis is inconclusive in this case — there is reason to hope that the committee will perform well, but this needs to be confirmed empirically.

The bottom line is, that when one builds a committee of classifiers, the goal is to

(1) Choose members that individually have low bias, and
(2) Choose members that are different from each other

By achieving (1), most members will make the correct classification most of the time, and so will the committee as a whole. By achieving (2), members will be able to compensate for the occasional mistakes of other members, to achieve better results overall.

3 Bagging

Bagging (Breiman 1996) — a name derived from "bootstrap aggregation" — is another way to use a single data set to generate multiple learners, each of which will hopefully be somewhat independent and therefore reduce the variance when combined in a committee.

In bagging, multiple hypotheses $h_1, \ldots, h_r$ are learned by running a base learner $\mathcal{L}$ on $n' < n$ random samples from data set $\mathcal{D}$ (of size n), where the data is sampled with replacement. The final classification is determined using simple voting.

Bagging is effective when the learning algorithm has high variance and acts to reduce this variance. For example, decision tree algorithms typically have high variance and bagging can help.

4 Boosting: The AdaBoost Algorithm

In improving on bagging, we will now consider the method of *boosting*, which is also known as an "arcing" method, meaning a *adaptive reweighting and combining* method. The approach is similar to bagging in that it generates multiple learners. It is different from bagging in that the examples that are sampled from $\mathcal{D}$ for each learner are chosen to concentrate on the "hardest" examples to classify. The result is that boosting will typically signficantly outperform bagging in experimental tests.

Boosting is a general method for improving the performance of a learning algorithm, by turning an algorithm that learns a single hypothesis into an algorithm that learns a committee whose hypotheses are combined by weighted majority. We focus in particular on the widely used AdaBoost algorithm (Freund and Schapire 1995).

AdaBoost has a remarkable property:

> *it is empirically found not to suffer from overfitting, even when training continues past zero training error!*

We will see that the explanation for this property is a bit more subtle than suggested by bias-variance alone, and will give us a first look at the concept of *margin*, which is a measure of the decisiveness of a classifier. We return to the decision margin in a few lectures in the context of *support vector machines*.

4.1 Weak Learners

Boosting was originally developed for converting so-called *weak learners* into a powerful learning algorithm.[1] But it is not critical that weak learners be used for boosting and in fact (as one would expect) the performance of boosting tends to improve as stronger learners are introduced.

For a binary classification problem, a weak learner $\mathcal{L}$, is one in which

$$error(\mathcal{L}) = P_{\mathbf{x}\sim\mathcal{D}}(h_{\mathcal{L}}(\mathbf{x};\mathcal{D}) \neq f(\mathbf{x})) \;\leq\; \frac{1}{2} - \gamma \tag{6}$$

for some $\gamma > 0$, for all data sets $\mathcal{D}$ and all true models f. This is the probability of a classification error in a world in which the instances $\mathbf{x}$ are sampled according to a distribution induced by data $\mathcal{D}$; i.e., with instances $\mathbf{x}$ sampled with replacement from $\mathcal{D}$. Informally, a weak learner is one that does (at least) slightly better than random guessing on its training data.

Many practical algorithms fall into this class. Loosely speaking, a weak learner is an algorithm that does better than random guessing.

An example we shall use of a weak learner is an algorithm for learning *decision stumps*. A decision stump is a decision tree of depth 1; i.e., it has only a single non-leaf node and splitting occurs on only a single

[1] Working in Valiant's PAC (probably approximately correct) learning model (1984), Kearns and Valiant (1988) were the first to pose whether a weak learning algorithm that performs slightly better than random guessing can be boosted into an arbitrarily accurate strong learning algorithm. Schapire (1989) developed the first polynomial time boosting algorithm. We will consider the PAC framework towards the end of this course.

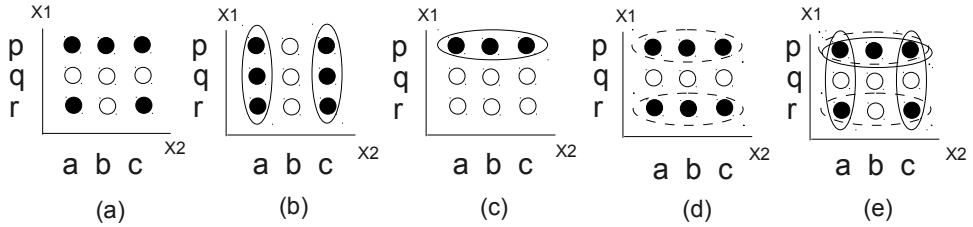

Figure 1: Decision stump hypotheses (b)-(d). Solid points predicted to be positive. Training data (a). Majority hypothesis (e).

feature. A decision stump can easily be learned from ID3, by choosing the feature with the best information gain.

Figure 1 (a) illustrates training data on features $X_1 = \{a, b, c\}$ and $X_2 = \{p, q, r\}$. This is a binary classification problem (solid points = positive target class.) The hypothesis space of decision stumps consists of subsets of these nine points that can be expressed as sets of rows or sets of columns. Three example hypotheses are shown in parts (b), (c) and (d) of the figure. They are axis-aligned because of the way the decision-tree learning algorithm works. Don't be confused; only the decision stump in (d) would be immediately learned by ID3.

4.2 Voting on Weighted Hypotheses

The goal of boosting is to convert a learner (perhaps even a weak learner) into a learner with low test set error, and ultimately low training set (or generalization) error. The approach is to learn a committee of hypotheses, where each hypothesis will be assigned a weight, in a manner to be described, and the committee will classify instances by taking the weighted majority of votes.

More precisely, after running a base learning algorithm $\mathcal{L}$ a total of r times, let the learned hypotheses be $h_1, \ldots, h_r$, with associated *hypothesis weights* $\alpha_1, \ldots, \alpha_r$. Now, given an instance $\mathbf{x}$, we classify this by taking a weighted vote over the classes assigned by each hypotehsis. That is, we compute score $\sum_{k=1}^{r} \alpha_k \cdot \mathbb{I}(h_k(\mathbf{x}) = y)$, for each $y \in Y$, and choose the class with the highest weighted votes. Here $\mathbb{I}(A)$ is an indicator function that is 1 when event A is true and 0 otherwise.

For binary classification problems, we can use a trick. By associating positive examples with the class $y = +1$, and negative examples with the class $y = -1$, then for a given instance $\mathbf{x}$, we can just compute $\sum_k \alpha_k h_k(\mathbf{x})$ and classify an instance as positive if this sum is positive, otherwise negative.

Putting this together, the ensemble classifier is just:

$$h(\mathbf{x}) = \text{sign}(\sum_k \alpha_k h_k(\mathbf{x})), \tag{7}$$

where sign(z) returns 1 if $z \geq 0$ and -1 otherwise.

Figure 1 (e) shows the result of voting the three hypotheses in Figure 1 (b), (c) and (d), where the hypotheses have equal weights. The points classified as positive by the majority hypothesis are filled in. As you can see, voting results in a hypothesis that cannot be represented by an individual decision stump. Introducing voting allows a richer hypothesis space to be used.

4.3 The Boosting Process

In order to create an ensemble of voting hypotheses, we have to answer two questions: How do you learn different hypotheses? How do you weight the different hypotheses? Boosting provides answers to these

questions. In particular, we will study the method of the AdaBoost algorithm.

Hypothesis Weights

Let's answer the second question first. Intuitively, it makes sense to give higher weight to hypotheses that have lower error. Since we only see the training data, we have to estimate the error of a hypothesis by its training error.

Boosting will work by training over a sequence of reweighted training data, where $w_k(i) > 0$ is the weight on training instance $\mathbf{x}_i \in \mathcal{D}$ in iteration k of boosting. Don't be confused: each learned hypothesis h_k will have a hypothesis weight α_k, and each data instance $\mathbf{x} \in \mathcal{D}$ has an *example weight*. We will keep the example weights normalized so that $\sum_{i=1}^{n} w_k(i) = 1$ in every iteration.

Define ϵ_k to be the error of hypothesis h_k on the weighted training set:

$$\epsilon_k = \sum_{i=1}^{n} w_k(i) \mathbb{I}(h_k(\mathbf{x}_i) \neq y_i) \tag{8}$$

Given ϵ_k, the training error in the kth iteration, then AdaBoost defines the weight of h_k to be,

$$\alpha_k = \frac{1}{2} \ln \frac{1 - \epsilon_k}{\epsilon_k}. \tag{9}$$

By the assumption that these are weak learners, then $\epsilon_k < 0.5$, and we see that α_k is ∞ when $\epsilon_k = 0$ and tends to zero as ϵ_k approaches 0.5. Notice that the smaller ϵ_k, the larger α_k, which is what we want. Why? This will provide more weight to a hypothesis that has small error. For $\epsilon_k = 0$ then the individual hypothesis h_k has zero training error and is given infinite weight in the committee — thus overriding all other members. Why this particular form for the adjustment makes sense is beyond the scope of this class.

Example Weights and Generating Mulitple Hypotheses

We've answered the second question, but we still haven't shown how to learn multiple hypotheses in the first place. The way to do that is to run the learning algorithm $\mathcal{L}$ using a differently weighted set of training data $\mathcal{D}$ in each iteration. For the first round we simply set example weights $w_1(i) = 1/n$ for all $i \in \{1, \ldots, n\}$. This provides an unweighted initial data set. The basic idea will be to provide additional weight to instances that are proving hard to classify.

In order to use a weighted data set we have to modify our base learning algorithm $\mathcal{L}$ to take into account the weights on each training instance. This is usually straightforward, and *can always be achieved by resampling n instances from the dataset $\mathcal{D}$ according to the current weights.* For ID3, we can do something more direct, and simply modify the definition of entropy given weights $w_k(i)$ on the data, as follows:

$$entropy(\mathcal{D}) = -\sum_{y \in Y} \frac{W_y}{W} \log_2 \frac{W_y}{W}, \tag{10}$$

where $W_y = \sum_{i=1}^{n} w_k(i) \cdot \mathbb{I}(y_i = y)$ and $W = \sum_y W_y$. This simply weights the counts of instances in each class by the weight on the instance.

A similar change is made in computing weighted entropy when determining *Remainder* and thus *Gain*. Finally, when picking the majority class to label a leaf, the weights are considered here and the *weighted majority* selected.

Now, the only question remaining is how to reweight the data instances from iteration to iteration. The basic intuition is that we want later hypotheses to compensate for the errors of earlier hypotheses. This is achieved by providing higher weight to those instances that the earlier hypotheses tend to get wrong. AdaBoost achieves this effect using the following formula:

$$w_{k+1}(i) = \begin{cases} \frac{w_k(i) \cdot e^{\alpha_k}}{Z} & \text{if } h_k(\mathbf{x}_i) \neq y_i \\ \frac{w_k(i) \cdot e^{-\alpha_k}}{Z} & \text{if } h_k(\mathbf{x}_i) = y_i \end{cases} \tag{11}$$

where Z is a normalization factor to ensure $\sum_{i=1}^{n} w_{k+1}(i) = 1$.

So the weight of the i-th instance is increased if h_k got the instance wrong, otherwise it is decreased. Moreover, the amount of increase or decrease depends on the weight α_k of the hypothesis. Recall that accurate hypotheses have higher weight and thus have more effect on adjusting the distribution on $\mathcal{D}$ for the next learner.

Here's a summary of the AdaBoost algorithm. It takes a base learning algorithm $\mathcal{L}$ (which should be at least a weak learner), a dataset $\mathcal{D}$, a number of rounds r to run, and either returns a single hypothesis with 0 training error, or a committee of r hypotheses $\{h_1, \ldots, h_r\}$, with weights α_k.

AdaBoost($\mathcal{L}$, $\mathcal{D}$, r) =
  w_1 is $1/n$ for all instances.
  For $k = 1$ to r do
    Apply $\mathcal{L}$ to $\mathcal{D}$ with weights w_k to learn h_k.
    ϵ_k is the weighted training error of h_k.
    If $\epsilon_k = 0$
      Then return h_k
    Else
      $\alpha_k = \frac{1}{2} \ln \left(\frac{1-\epsilon_k}{\epsilon_k} \right)$.
      For each training instance $(\mathbf{x}_i, y_i) \in \mathcal{D}$
        If $h_k(\mathbf{x}_i) \neq y_i$
          $w_{k+1}(i) = (w_k(i) \cdot e^{\alpha_k})/Z$
        Else $w_{k+1}(i) = (w_k(i) e^{-\alpha_k})/Z$
        Z is the normalization to make $\sum_i w_{k+1}(i) = 1$
  End for loop
  Return $\{(h_1, \alpha_1), \ldots, (h_r, \alpha_r)\}$

Note that the error ϵ_k is of hypothesis h_k on the weighted training data and not that of the weighted ensemble hypothesis in that iteration. Furthermore, note that training continues even past the point at which the weighted ensemble has zero error on the training data. What matters is the error of the most recently learned hypothesis.

4.4 Why does Boosting work?

There is plenty of empirical evidence that boosting works very well. We present some evidence in lecture. There is also strong theory behind boosting.

What is most remarkable is that boosting does not tend to overfit data even when training continues for many rounds and generates a very complex ensemble hypothesis! This goes against all of our intuitions: boosting will even continue to improve generalization error past the point where these is *zero* training error! How can this be?

In fact, one might expect boosting to cause overfitting because boosting allows a much richer hypothesis space to be used and can tend to fit the training data exactly.

More recent theoretical results explain that the reason for the surprising empirical performance of boosting is because there is a *simple classifier with performance that is very close to the performance of the learned ensemble.*

That is, although the hypothesis represented by the committee (or *learned ensemble*) is itself complex, there is a nearby hypothesis that is simple and has just about the same training data error. Thus we see that we can in fact appeal back to the principles that come from Occam's razor. The technical details are beyond the scope of this class, but the basic argument for why there is a simple classifier with performance close to that of the learned ensemble is as follows.

A weighted classifier can be associated with a *decision boundary*, separating positive and negative points by tracing feature vectors $\mathbf{x}$ where $\sum_k \alpha_k h_k(\mathbf{x}) = 0$. The distance of the boundary from a correctly classified training example is called the *margin* on that instance; in the case when an instance is incorrectly classified

the margin is negative. Intuitively, the margin measures how well the boundary separates or explains the data.

A classifier might get all the training data exactly right but leave some points $\mathbf{x}$ close to the margin, meaning that the classifier barely gets them right, while if they are far from the margin there is room for error. Increasing the margin makes the decision more definite. It turns out that increasing the margin leads to good generalization and reduces overfitting, and that boosting works to maximize the margin (Schapire et al. 1997).[2]

Returning to the "simplicity" argument, it is reasonable to expect that when there is a large margin that a simpler classifier can be constructed that is close to the ensemble hypothesis while having the same training set error. Imagine "dropping" a simple, thin cookie cutter shape, into the wide gaps made by a more intricate, but thick cookie cutter shape.

The process of boosting focuses on instances whose margin is small or negative. Recall that the classifier for a weighted hypothesis method takes the weighted some of the votes on +1/-1 and if the result is ≥ 0 returns 1 as the label, and otherwise 0.

Based on this, the margin of the ensemble hypothesis trained up to round $k-1$, on an instance $(\mathbf{x}_i, y_i)$, is exactly

$$\mathit{margin}(\mathbf{x}_i) = \frac{y_i \cdot \sum_{k'=1}^{k-1} \alpha_{k'} h_{k'}(\mathbf{x}_i)}{\sum_{k'=1}^{k-1} \alpha_{k'}} \tag{12}$$

This is normalized, so that the margin $\mathit{margin}(\mathbf{x}_i) \in [-1, +1]$. For example, if $y_i = +1$ but $\sum_{k'=1}^{k-1} \alpha_{k'} h_{k'}(\mathbf{x}_i) < 0$ then this point is misclassified and the product is negative. If $y_i = +1$ and $\sum_{k'=1}^{k-1} \alpha_{k'} h_{k'}(\mathbf{x}_i) > 0$ but small, then this point is correctly classified but by a small margin.

It can be shown that in round k, AdaBoost *places the most weight on examples* $(\mathbf{x}_i, y_i)$ *for which* $y_i \cdot \sum_{k'=1}^{k-1} \alpha_{k'} h_{k'}(\mathbf{x}_i)$*, and therefore the margin, is the smallest.* Moreover, if the base learner is at least a weak learner, then the training error of the combined hypothesis decreases exponentially quickly with the number of iterations. Also, it can be shown that the number of instances with a small (or negative) margin decreases exponentially quickly with the number of iterations. This provides the theory to support AdaBoost.

It is interesting, then, that the theory that underpins boosting is based not on the variance-bias decomposition but rather a more subtle argument involving margin, or the degree of decisiveness of a learned hypothesis. In fact, boosting was designed to work on weak learners which tend to have high bias but low variance, and its success cannot be explained purely through acting to minimize variance. Boosting and bagging differ in this sense. Boosting tends to be more successful in reducing bias than bagging, while bagging relies more on reducing variance.

Bagging also tends to increase the margins associated with the training data as training continues, but less aggressively than boosting. Boosting will seek to decrease the number of examples with a small margin.

In a few lectures we will turn to *support vector machines* (SVMs), which formulate an optimization problem with which to explicitly maximize the margin of a classifer on training data.

4.5 OK, when does Boosting not work?

However, boosting is not a cure-all. There are several reasons why boosting might not work:

Not enough data Boosting can't produce something out of nothing, and if there's simply not enough data to learn effectively, boosting will not help.

Base learner too weak If the base learning algorithm is too weak, boosting will not be able to help.

Base learner too strong More surprisingly, boosting can also have trouble when the base learning algorithm is too strong and overfits the data on its own.

[2] Boosting the margin: A new explanation for the effectiveness of voting methods. Schapire, Freund, Bartlett and Lee, ICML 1997.

Noisy data The most serious practical limitation of boosting is its vulnerability to "noise" in the data. By noise, we mean here a data instance that is incorrectly labeled, or associated with a very unusual class for the particular feature vector. The problem is that boosting increases the weight of instances that are difficult to explain and boosting will try very hard to produce a hypothesis that correctly classifies the instance.

Even though boosting is susceptible to noise, it also presents a way of identifying noisy instances! After running a number of rounds of boosting, points with very large weights are quite likely to be noise. Such points are identified as being vastly inconsistent with the rest of the data. So boosting can also be used to discover outliers, which can be thrown out of the training set.

In summary, boosting is a fast and simple method that can improve the performance of many learning algorithms. It is fast to program and requires no tuning. One drawback is that the hypothesis that is learned is a complex, ensemble hypothesis. This can make the results hard for people to understand. A growing number of empirical studies have established the efficacy of boosting.