CS181 Lecture 9 — Variance and Boosting

Today we continue to study the theory of learning. 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 boosting method for learning committees.

1 The Bias-Variance Decomposition

Consider some learning algorithm L that learns a hypothesis h from a training set D. Last time, we saw how to use the structure of the hypothesis space H of L to bound the probability that L will learn a bad hypothesis h. Today we analyze the error of h in a somewhat different way.

Recall that we assume that there is a single probability distribution P from which all training and test instances x are independently drawn. If the true concept is f, we defined the error of h to be P(h(x) = f(x)). If the possible classes are 0 and 1, we can write this as

$$\text{Error}(h) = E_P[(h(x) - f(x))^2]. \quad (1)$$

The notation E_P means expectation over all possible values of x, taken with respect to the distribution P. The term inside the square brackets will be 1 precisely when h(x) = f(x) and 0 otherwise. A function that has the property that it is 1 if an event holds and 0 otherwise is called an indicator of that event. The expectation of the indicator of an event is equal to the probability of the event. This error function, taking the expected square of the difference between the predicted value and the true

value, is used for many different problems in machine learning.

Now, this expression characterizes the error of a single hypothesis h. We are more interested in studying the expected error of learning algorithm L, given a certain amount of training data. That is, if we fix the true concept f, what is the expected error of the hypothesis learned by L, when the expectation is taken over the possible training data D.

To state this precisely, we need some probability distribution over training sets. An easy way to achieve this is to fix the size N of the training set, and draw each training instance from the distribution P. Once the distribution over training sets is fixed, we use the notation E_D to denote the expectation over all possible training sets

D with respect to the given distribution over training sets. We can then characterize the expected error of a learning algorithm L, for a given true concept f, by

$$\text{Error}(L) = E_D[E_P[(h(x) - f(x))^2]] \tag{2}$$

In words, this is the expectation over training sets of the error of the hypothesis learned for that training set.[1]

We will now analyze this expression by introducing the quantity $E_D[h(x)]$. This is the expected prediction of the hypothesis h learned by L from training set D on a particular instance x. Again the expectation is taken over training sets. This quantity represents what we can expect our learning algorithm to do on average, and naturally we would like it be close to f(x).

We now proceed as follows. The derivation uses several properties of expectation that you can review in a probability text. 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 x is equal to the expectation of x.

$$\begin{aligned}
E_D[(h(x) - f(x))^2] &= E_D[\{h(x) - E_D[h(x)] + E_D[h(x)] - f(x)\}^2] \\
&= E_D[(h(x) - E_D[h(x)])^2 + \\
&\quad 2(h(x) - E_D[h(x)])(E_D[h(x)] - f(x)) + \\
&\quad (E_D[h(x)] - f(x))^2] \\
&= E_D[(h(x) - E_D[h(x)])^2] + \\
&\quad 2E_D[h(x)E_D[h(x)]] - \\
&\quad 2E_D[E_D[h(x)]E_Dh(x)] - \\
&\quad 2E_D[h(x)f(x)] + \\
&\quad 2E_D[E_D[h(x)]f(x)] + \\
&\quad E_D[(E_D[h(x)] - f(x))^2]
\end{aligned}$$

$$
\begin{aligned}
&= E_D[(h(x) - E_D[h(x)])^2] + \\
&\quad 2(E_D[h(x)])^2 - 2(E_D[h(x)])^2 - \\
&\quad (2E_D[h(x)f(x)] - 2E_D[h(x)f(x)]) + \\
&\quad E_D[(E_D[h(x)] - f(x))^2] \\
&= E_D[(h(x) - E_D[h(x)])^2] + E_D[(E_D[h(x)] - f(x))^2] \\
&= E_D[(h(x) - E_D[h(x)])^2] + (E_D[h(x)] - f(x))^2
\end{aligned}
$$

Substituting back into (2), and remembering that we can switch the order of expectation operators just like integrals, we get

$$\text{Error}(L) = E_P[E_D[(h(x) - E_D[h(x)])^2]] + E_P[(E_D[h(x)] - f(x))^2] \quad (3)$$

Look closely at this formula. It consists of two terms. The first takes the expectation of the variance of h, taken over all instances x. (Recall that the variance of z is

[1]Note that the term inside the expectation depends on D and L even though they are not mentioned explicitly, because h is determined by L and D.

defined by $E[(z - E[z])]^2$, and is a measure of how far z tends to stray from its average value). Thus, this term measures how much the predictions of the learned hypothesis vary for different training sets. The second term is the expected squared bias of h, where the bias of h is defined to be $E_D(h(x)) - f(x)$, i.e., the difference between the average prediction of the learned hypotheses and the true prediction. Thus, this term is a measure of systematic error due to the learning algorithm, without regard to variance in the performance of the algorithm.[2]

This is a fundamental result, called the bias-variance decomposition. We see that the expected error of a learner is divided into two components: the squared bias of the learning algorithm, and the variance. It suggests that if we want to improve the performance of a learning algorithm, we have two paths to follow: reduce the bias, or reduce the variance.

The analysis we did here is different from that of PAC learning theory in a fundamental way. In PAC theory, the true concept f was allowed to vary, though we usually restricted it to being in some particular hypothesis space H. Here, the analysis is done relative to one particular true concept f, but it could be any concept, and in general is not necessarily in the hypothesis space of the learning algorithm.

2 Committees

The result we just saw is also often called the bias-variance tradeoff . The reason is that when one improves, the other often deteriorates. 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, noise and variation in the training data will have little effect on the learning algorithm. As you widen the hypothesis space, your bias (both inductive and statistical) 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.

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.

[2]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.

One simple way of learning a committee is to use a technique like cross-validation. 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%. Each of the folds is used once as the validation set. It is important to stress that unlike the experimental measurement technique of cross-validation, the reason to use cross-validation here is not to perform 10 separate experiments with the same data, but to produce 10 different learners from the same data to form a committee. If you wanted to test the performance of this method, you would need a completely separate test set.

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 voter learned some noise in its training set, that noise might not have been present in the other voters' training sets.

This last sentence relies on an important assumption, namely, that the individual hypotheses learned by the different learners in the committee be somewhat indepen-

dent of each other. The reason is that when they are not independent, then when one committee member overfits to some noise in the data, other members are also likely to do so, so the variance does not get reduced.

How well does the independence assumption hold in the case of the ten decision trees learned using the 10-fold method, as described above? 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 noise if that noise is 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 that 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 Boosting

We now turn to a very useful practical technique that came out of the study of computational learning theory. Boosting is a general method for improving the performance of many different learning algorithms, by turning an algorithm that learns a single hypothesis into an algorithm that learns a committee.

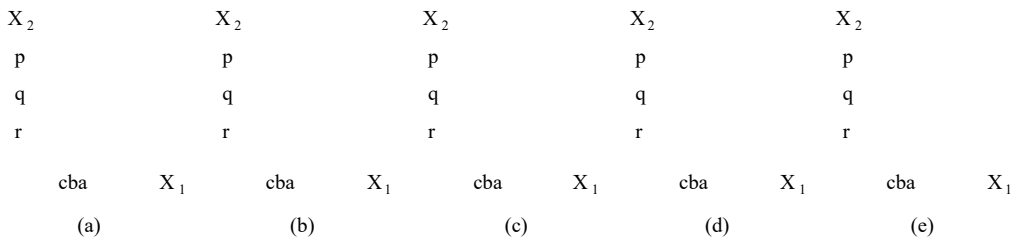

Figure 1: Decision stump hypotheses.

3.1 Weak Learners

The study of boosting begins with the observation that PAC-learnability is a very strong property. It requires that for every positive ϵ and δ, an algorithm be capable of learning with probability at least $1 - \delta$ a hypothesis with error at most ϵ. A learning algorithm that can do this is called a strong learner. Strong learners are difficult to design, and many practical machine learning algorithms are not strong learners.

In fact, there is a class of learning algorithms called weak learners, and many practical algorithms fall into this class. Loosely speaking, a weak learner is an algorithm that does better than random guessing. In other words, it is sufficient that the probability the weak learner will produce a hypothesis that correctly classifies a test example be slightly greater than 1/2.

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. A decision stump can easily be learned by choosing the attribute with the highest information gain for the non-leaf node.

Figure 1 (a) illustrates the set of points described by attributes X_1 with values a, b and c, and X_2 with values p, q and r. 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.

3.2 Voting Hypotheses

The goal of boosting is to take a weak learner such as decision stumps and convert it into a strong learner. The approach is to learn a committee of voting hypotheses, where each committee member is a weak learner. Each committee member 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 the original weak learning algorithm L a total of T times, let the learned hypotheses be $h_1,...,h_T$, with weights $w_1,...,w_T$. Now, given an instance x, we can give each possible class c a score equal to $\sum_{t:h_t(x)=c} w_t$, and choose the class with the highest score. For binary classification problems, we can use a trick. We can identify positive examples with the class c = +1, and negative examples with the class c = −1. We can then compute $\sum_t w_t c_t$, and classify an

instance as positive if this sum is positive, otherwise negative.

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 voting 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.

3.3 Weighting the hypotheses

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, in particular the AdaBoost algorithm, provides answers to these questions.

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 set, we have to estimate the error of a hypothesis by its training error.

Define ϵ_t to be the error of hypothesis h_t on the training set — ϵ_t is the probability of misclassification of h_t on the examples in the training set. This probability is defined according to some distribution P_t over the examples in the training set, which we shall discuss later.

Given ϵ_t, AdaBoost defines the weight of h_t to be

$$\alpha_t = \frac{1}{2} \ln \frac{1 - \epsilon_t}{\epsilon_t}.$$

The main motivation for this formula is that it makes the math work out. However, notice that the smaller ϵ_t, the larger α_t, which is what we want.

Note that this α_t is ∞ when ϵ_t is 0. When this happens, the individual hypothesis h_t has zero training error and is given infinite weight in the committee — thus overriding all other members. We will see in the definition of the algorithm that it cannot happen that two hypotheses get infinite weight.

3.4 Learning multiple 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 weak learner L using a different distribution over the training instances in each round. The distribution P_t over the instances in the t-th round is the same distribution that is used to measure the training error of the hypothesis h_t. We can express this in the formula

$$\epsilon_t = P_t(h_t(x_i) = c_i) = \sum_{i:h_t(x_i)=c_i} P_t(i).$$

In order to use a distribution over training instances, we have to modify our weak learner to take into account the weights of the training instances. For decision stumps, we can modify the definition of entropy as follows:

$$\text{Entropy}(D) = f\left(\frac{p}{p+n}\right) \text{ where } f(x) = -(x \log x + (1 - x) \log(1 - x))$$

as before, but now p is the sum of the weights of positive examples, while n is the sum of the weights of negative examples.

Now, the only question remaining is how to choose the distribution over the training instances. The basic intuition is that we want later hypotheses to compensate for the errors of earlier hypotheses. The way to achieve this is to give higher weight to those instances that the earlier hypotheses tend to get wrong. AdaBoost achieves this effect using the following formula:

$$P_{t+1}(i) = \begin{cases} P_t(i)e^{\alpha_t} & \text{if } h_t(x_i) = c_i \\ P_t(i)e^{-\alpha_t} & \text{if } h_t(x_i) = c_i \end{cases}$$

So the weight of the i-th instance is increased if h_t got the instance wrong, otherwise it is decreased. For the first round, a uniform distribution is used. It is necessary to normalize P_{t+1} so that it is a probability distribution, i.e. $\sum_i P_{t+1}(i) = 1$.

Here's a summary of the AdaBoost algorithm. It takes a weak learner L, a dataset D, and a number of rounds m to run, and either returns a single hypothesis that has 0 training error, or a committee of m hypotheses h_t, with weights α_t.

AdaBoost(L, D, m) =
  P_1 is the uniform distribution.
  For t = 1 to m do
    Apply L to D with distribution P_t to learn h_t.
    $\epsilon_t = P_t(h_t(x_i) = c_i)$.
    If $\epsilon_t = 0$
      Then return h_t
    Else
      $\alpha_t = \frac{1}{2} \ln \frac{1-\epsilon_t}{\epsilon_t}$.
      For each training instance $< x_i, c_i > \in D$
        If $h_t(x_i) = c_i$
          Then $\hat{P}_{t+1}(i) = P_t(i)e^{\alpha_t}$
          Else $\hat{P}_{t+1}(i) = P_t(i)e^{-\alpha_t}$
      /* P is normalized $\hat{P}$*/
      $Z = \sum_i \hat{P}_{t+1}(i)$
      For each i, $P_{t+1}(i) = \frac{1}{Z} \hat{P}_{t+1}(i)$
    End for loop
  Return $< h, \alpha >$

3.5 Discussion

There's plenty of empirical evidence that boosting works very well. There's also strong theory behind boosting. 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.

Weak learner too weak If the weak learner is too weak (for example, decision stumps where only X_1 is allowed as the test attribute), boosting will not be able to help.

Weak learner too strong More surprisingly, boosting has trouble when the weak learner is too strong. The reason is that strong learners have a tendency to overfit the data.

In fact, one might expect boosting to cause overfitting even with a weak learner, because boosting allows a much richer hypothesis space to be used, which may allow the training data to be fit exactly. Empirically, boosting has not been found in general to cause overfitting. This fact, that boosting can greatly enhance the capability of a weak learning algorithm without causing overfitting, is one of its most attractive features.

More recent theoretical results try to explain the fact that boosting does not cause overfitting. The basic argument is as follows. Any classifier can be associated with a decision boundary, separating positive and negative points. The distance of the boundary from the nearest correctly classified training examples is called the margin. Intuitively, the margin measures how well the boundary separates the data. If there are training points close to the margin, it means the classifier barely gets them right, while if they are far from the margin there is room for error. It turns out that increasing the margin leads to good generalization and reduces overfitting, and that boosting tries to maximize the margin.

The approach of maximizing the margin is the key idea behind support vector machines (SVMs). SVMs are rapidly becoming one of the most popular machine learning approaches, and they have attained superb performance in a number of applications.

Noisy data The most serious practical limitation of boosting is its vulnerability to noise in the data. The problem is that boosting increases the weight of points that are difficult to explain. This means that the weights of noisy points will tend to become very large, so the learning algorithm will try very hard to produce a hypothesis that explains the noise. Thus noisy data tends to have a large effect on the hypothesis eventually produced by the boosting algorithm.

Even though boosting is susceptible to noise, it also presents a way of dealing with noise. 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 — they are called outliers. So boosting can be used to discover outliers, which can be thrown out of the training set.