CS 446 Machine Learning Fall 2016 Oct 18, 2016

Boosting

Professor: Dan Roth *Scribe: Ben Zhou, C. Cervantes*

1 Boosting

Boosting is a general learning paradigm for putting together a *strong learner*, given a collection (possibly infinite) of *weak learners*.

The original boosting algorithm was proposed as an answer to a theoretical question in PAC learning *The Strength of Weak Learnability; Schapire, 89*. This paper shows that boosting has interesting theoretical implications, like the relation between PAC learnability and compression.

If a concept class is efficiently PAC learnable, then it is efficiently PAC leanrable by an algorithm whose required memory is bounded by a polynomial in n, size c, and $\log(\frac{1}{\epsilon})$. Therefore, there is no concept class for which efficient PAC learnability requires that the entire sample be contained in memory at one time; there is always another algorithm that "forgets" most of the sample.

The key contribution of boosting, however, has been in providing a practical way to compose a strong learner from multiple weak learners. In fact, boosting is a member of a family of *ensemble algorithms*, but has stronger guarantees.

1.1 Example: How May I Help You

Boosting can be viewed in the context of chatbots. In the early 1990s AT&T had a chatbot system such that calling a number and answering a set of automated questions would enable the system to navigate you to the right recipient.

Goal
Automatically categorize type of call requested by phone customer.

> "I'd like to place a collect call long distance please" → (Collect)
>
> "Operator I need to make a call but I need to bill it to my office" → (ThirdNumber)
>
> "I'd like to place a call on my master card please" → (CallingCard)

Observation
It is easy to find "rules of thumb" that are often correct (eg. if the utterance contains "card", predict "CallingCard"). However, it is hard to find a single highly accurate prediction rule. Therefore, we want an algorithm that can generate a strong learner from a set of weak learners.

1.2 Procedure

1. Select a small subset of examples
2. Derive a rough rule of thumb
3. Examine second set of examples
4. Derive second rule of thumb
5. Repeat T times
6. Combine learned rules of thumb into a single hypothesis

1.3 Theoretical Motivation

We define a *Strong PAC algorithm* to be – for any distribution, for any $\epsilon, \delta > 0$, given polynomially many random examples – an algorithm that finds hypothesis with error $\leq \epsilon$ with probability $\geq (1 - \delta)$.

We can also define a *Weak PAC algorithm* as one as above, except for $\epsilon \leq \frac{1}{2} - \gamma$. In this case, ϵ is not required to be as small as we want, instead, it is only bounded by the notion that it predicts correctly more than it predicts incorrectly. This weaker requirement means that such an algorithm is still learning something, just not as well.

Kearns and Valiant '88 asked the question, Does weak learnability imply strong learnability? In other words, can we boost from weak to strong?

1.4 History

Schapire '89
First provable boosting algorithm: Call weak learner three times on three modified distributions, get slight boost in accuracy, then apply recursively.

Freund '90
"Optimal" algorithm that "boosts by majority".

Drucker, Schapire and Simard '92
First experiments using boosting, limited by practical drawbacks.

Freund and Schapire '95
Introduced AdaBoost algorithm, has strong practical advantages over previous boosting algorithms.

AdaBoost was followed by a huge number of papers and practical applications.

1.5 A Formal View of Boosting

Given training set $(x_1, y_1), ...(x_m, y_m)$
$y_i \in \{-1, +1\}$ is the correct label of instance $x_i \in X$
For $t = 1...T$
    Construct a distribution D_t on $\{1, ...m\}$
    Find weak hypothesis ("rule of thumb")
        $h_t : X \to \{-1, +1\}$
    with small error ϵ_t on D_t
        $\epsilon_t = Pr_D[h_t(x_i) \neg = y_i]$
Output: final hypothesis H_{final}

2 AdaBoost

Typically when people talk about boosting, they are referring to AdaBoost.

In AdaBoost, the basic procedure requires choosing a distribution over the data, learning a weak hypothesis based on the data as drawn from that distribution, and choosing a new distribution – such that mistakes are weighted more heavily – from which a new weak hypothesis can be learned, where this process can be repeated for any arbitrary number of weak learners.

To begin, the first distribution D_1 (where $D_1(i)$ is the weight of the i^{th} example) is uniform

$$D_1(i) = \frac{1}{m}$$

Now, for any arbitrary time t, we can compute the next distribution D_{t+1} given the current distribution D_t and the current hypothesis h_t.

$$D_{t+1}(i) = \frac{D_t(i)}{z_t} \times e^{-\alpha_t y_i h_t(x_i)}$$

In one equation, this expresses that the next distribution $D_{t+1}(i)$ is going to correspond to the normalized current distribution $\frac{D_t(i)}{z_t}$ times either $e^{-\alpha_t}$ – when h_t predicts correctly – or e^{α_t} when h_t makes a mistake.

Note that z_t is a normalization constant

$$z_t = \sum_i D_t(i) \times e^{-\alpha_t y_i h_t(x_i)}$$

and α_t is given by

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

Note that in weak learners, $\epsilon < \frac{1}{2}$ so since α_t is half of the natural log of quantity that's greater than 1, we know that α must be a positive number ($\alpha > 0$).

Therefore, examples on which h_t makes a correct prediction are demoted and examples on which h_t makes mistakes are promoted. Intuitively, this means that the next hypothesis will have to focus more on those examples that the previous hypothesis got wrong, because they will be heavier according to the distribution.

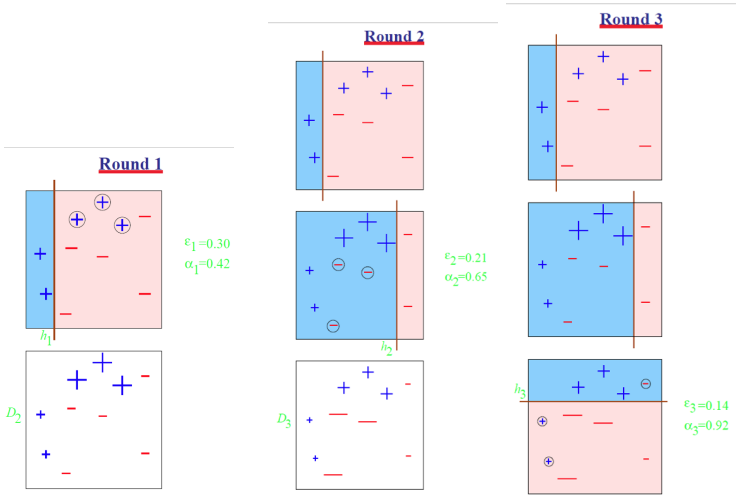
The final hypothesis, then, is a linear combination of all the hypotheses weighted by α

$$H_{final}(x) = sign(\sum_t \alpha_t h_t(x))$$

Recall that if ϵ_t is small, α_t is large, which corresponds to the notion that if h_t made few mistakes, it will contribute more to the final hypothesis.

2.1 A Toy Example

Consider the following AdaBoost example.



(a) Round 1 (b) Round 2 (c) Round 3

In the first round (top left), the vertical line indicates that positive examples are on the left, negative are on the right. The three examples on which mistakes were made are then weighed more heavily by the next distribution (bottom left). This results in the next learner choosing a vertical line that classifies the previous mistakes correctly (center middle), but it makes mistakes of its own, which are in turn re-weighted (bottom middle). In round three, these mistakes are classified correctly via the horizontal line (bottom right).

To produce the final hypothesis, we create a weighted combination of the three hypotheses, shown in Figure 2.

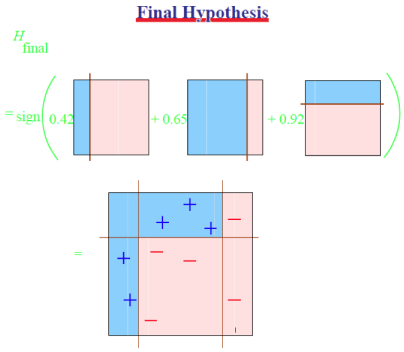


Figure 2: Final Hypothesis

Note that it is possible that the combined hypothesis makes no mistakes on the training data, but boosting can still learn by adding more weak hypotheses. Doing so does not improve performance on the training data, but can generalize well (perform better on test).

2.2 Theorem

Consider running AdaBoost, where the error the hypothesis at time t is bounded away from one half

$$\epsilon_t = \frac{1}{2} - \gamma_t$$

Claim
If we can bound ϵ_t away from a half, then the training error is bounded as below

$$\begin{aligned} training\ error(H_{final}) &\leq \prod_t [2\sqrt{\epsilon_t(1-\epsilon_t)}] \\ &\leq \prod_t \sqrt{1-4\gamma_t^2} \\ &\leq exp(-2\sum_t \gamma_t^2) \end{aligned} \tag{1}$$

Consider that these bounds are the result of the following

$$\epsilon_t(1-\epsilon_t) = (\frac{1}{2} - \gamma_t)(\frac{1}{2} + \gamma_t) = \frac{1}{4} - \gamma_t^2$$

$$1 - (2\gamma_t)^2 \leq exp(-(2\gamma_t)^2)$$

Therefore, if – for all t – $\gamma_t \geq \gamma \geq 0$, then

$$training\ error(H_{final}) \leq e^{-2\gamma^2 T}$$

This means that given a fixed γ, we can make training error as small as we want, given a large enough T.

This is stated in terms of training error, which we care less about, considering that we want to minimize error during test time. However, framing this in terms of training error is satisfying enough because – according to PAC learning – if we can bound our hypothesis space (in this context, T), we will perform well on test if we perform well on train.

Now we must prove this inequality.

Proof
Let $f(x) = \sum_t \alpha_t h_t(x) \Rightarrow H_{final}(x) = sign(f(x))$

Step 1: Unwrapping Recursion

$$\begin{aligned} D_{final} &= \frac{1}{m} \cdot \frac{exp(-y_i \sum_t \alpha_t h_t(x_i))}{\prod_t Z_t} \\ &= \frac{1}{m} \cdot \frac{e^{-y_i f(x_i)}}{\prod_t Z_t} \end{aligned} \tag{2}$$

Here $D_{final}(i)$ is the final distribution for example i.

Step 2: $training\ error(H_{final}) \leq \prod_t Z_t$

$$H_{final}(x) \neq y \Rightarrow yf(x) \leq 0 \Rightarrow e^{-yf(x)} \geq 1$$

Therefore

$$\begin{aligned} training\ error(H_{final}) &= \frac{1}{m} \sum_i (1\ if\ y_i \neq H_{final}(x_i)), (0\ else) \\ &\leq \frac{1}{m} e^{-y_i f(x_i)} \\ &= \sum_i D_{final}(i) \prod_t Z_t \\ &= \prod_t Z_t \end{aligned} \tag{3}$$

Step 3: $Z_t = 2\sqrt{\epsilon_t(1-\epsilon_t)}$

$$\begin{aligned} Z_t &= \sum_i D_t(i) exp(-\alpha_t y_t h_t(x_i)) \\ &= \sum_{i:y_i \neq h_t(x_i)} D_t(i) e^{\alpha_t} + \sum_{i:y_i = h_t(x_i)} D_t(i) e^{-\alpha_t} \\ &= \epsilon_t e^{\alpha_t} + (1-\epsilon_t) e^{-\alpha_t} \\ &= 2\sqrt{\epsilon_t(1-\epsilon_t)} \end{aligned} \tag{4}$$

Step 2 and 3 together prove the theorem, that the error of the final hypothesis can be as low as you want.

3 Conclusion

3.1 Boosting Confidence

Consider the distinction between confidence $(1-\delta)$ and accuracy $(1-\epsilon)$ from PAC learning. Boosting the accuracy – what we've discussed thusfar – is actually much harder than boosting the confidence.

Assume a fixed accuracy parameter to ϵ and a learning algorithm L such that for any target concept $c \in C$, for any distribution D, L outputs a hypothesis h such that $error(h) < \epsilon$ with confidence at least $\delta_0 = \frac{1}{q(n,|c|)}$ for some polynomial q.

If we are willing to tolerate a slightly higher error $\epsilon + \gamma$, then we can achieve arbitrarily high confidence $1-\delta$.

The key idea is to learn multiple times. Given the algorithm L, we construct a new algorithm L' that simulates algorithm L multiple (k) times on independent samples from the same distribution.

Assume $h_1 ... h_k$ are the hypothesis produced over k iterations. Since each is independent, the probability that all of the $h_1 ... h_k$ have $error > \epsilon$ is at most $(1-\delta_0)^k$, otherwise, at least one h_j is good.

Thus, if we want to reduce δ by 2 times, we just need to solve

$$(1-\delta_0)^k < \frac{\delta}{2} \Rightarrow k > \frac{1}{\delta_0} \ln(\frac{2}{\delta})$$

Here, L' would simply use the h_i that makes the fewest mistakes on sample S, where we would need to determine the size of S.

Boosting confidence is really the typical thing we do in computer science by running an algorithm multiple times. Boosting accuracy, however, required the sophisticated AdaBoost algorithm.

3.2 Ensemble Methods

AdaBoost is actually one of a family of algorithms called ensemble methods.

Boosting

1. Start with a collection of examples (uniform)

2. Train a model
3. Compute the error
4. Increase the weights on incorrect predictions; decrease on correct predictions
5. Predict

In boosting, the weights are an immediate function of the learned error. Note that while boosting is a very good algorithm in the general case, it is difficult to obtain probability estimates from it.

Bagging
As with boosting, bagging is a method that generates multiple predictors and combines them to make final predictions. The process of generating these hypotheses is referred to as creating *bootstrap replicates* of the learning sets

1. Generate a sample from the dataset (With repetition)
2. Learn a hypothesis on this generated sample
3. Repeat
4. Take the majority or average prediction across all learned hypotheses

Consider the case of bagged decision trees, shown in Figure 3. Here we draw

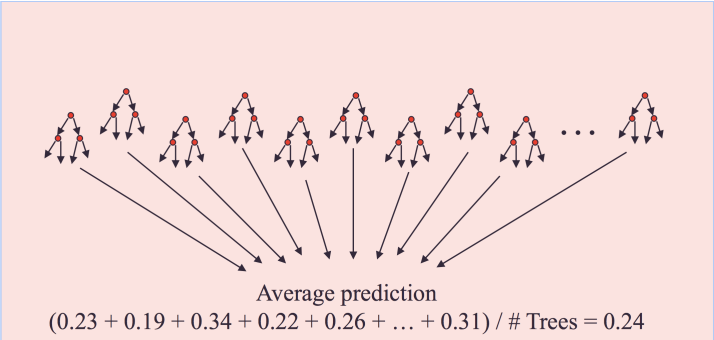


Figure 3: Bagged decision trees

100 bootstrapped samples of data, we train a tree on each sample to get 100 trees, then we take the average prediction of trees on out-of-bag samples.

Tests on real and simulated data sets using classification and regression trees and subset selection in linear regression show that bagging can give substantial gains in accuracy. The vital element is the instability of the prediction method. If perturbing the learning set can cause significant changes in the predictor constructed, then bagging can improve accuracy.

Random Forests
This is an another version of bagged decision trees where you draw 1000+ bootstrap samples of data and draw sample of available attributes at each split, train trees on each sample/attribute set and get 1000+ trees, then average prediction of trees on out-of-bag samples. In effect, we're training on subsampled examples *and* subsampled features.