Boosting (Adaboost) Derivations

Mickey Chao

February 2016

1 Intuition

For binary classification problems, we may run into trouble trying to find a single best classifier to separate data. So, one question we may ask is, "Why do we have to use a single classifier?" The general idea behind Boosting is that by combining several weaker classifiers and letting them vote on what class each training sample should take, we can build a strong classifier.

If our weak classifiers are $h_1, h_2, \ldots, h_m$ and each weak classifier outputs either $+1$ or -1, then one way we can define the majority vote would be $\text{sign}(h_1 + h_2 + \cdots + h_m)$.

The problem we now face is how to create these classifiers and how to get a diverse population of classifiers to help us account for all kinds of samples. One idea is to iteratively generate classifiers where each classifier helps correct the previous classifiers' mistakes. For example, we may train classifier h_1 on an unweighted dataset. We could then train the next classifier h_2 on a weighted dataset where the mistakes that h_1 made are emphasized. Then, we could train the third classifier h_3 on a weighted dataset where h_1 and h_2 disagree or are both incorrect. Thus, our process looks something like this:

$$\begin{array}{l}
\text{Data} \xrightarrow{\hspace{12em}} h_1 \\
\text{Data exaggerating } h_1 \text{ errors} \xrightarrow{\hspace{6em}} h_2 \\
\text{Data exaggerating } h_1 + h_2 \text{ errors} \xrightarrow{\hspace{4em}} h_3 \\
\qquad\qquad\qquad\qquad \cdots
\end{array}$$

We can exaggerate points by assigning weights, w_i to each of them. If we express the error, ϵ, associated with applying the classifier $\text{sign}(h_1 + h_2 + \cdots + h_m)$ at a given iteration m as

$$\epsilon_m = \sum_{\text{wrong}} w_i^t$$

then points with higher weights will cause more penalty if we misclassify them.

In the beginning, we can assign $w_i = \frac{1}{n}$ for all i because nothing suggests that some points should be weighted more than others. As we produce more and more classifiers, these weights will change. Overall, though, we need to enforce that $\sum w_i = 1$ so the weights remain a meaningful distribution.

Finally, we might want to consider giving the individual classifiers weights as well. We can slightly alter the classification function to be

$$H(x) = \text{sign}(\alpha_1 h_1(x) + \alpha_2 h_2(x) + \alpha_3 h_3(x) + \cdots)$$

In this way, better-generalized classifiers might get more influence. We can think of this as combining the "wisdom of a weighted crowd of experts."

2 Adaboost Algorithm

In English, the Adaboost algorithm can be described easily, so here is some pseudocode:

```
Let $W_0^{(i)} = \frac{1}{N}$ where $N$ is the number of samples.
For $m = 1, 2, 3, \ldots$
    Pick optimal $h_m(x)$ that minimizes $\epsilon_t$ for $\text{sign}(h_1 + h_2 + \cdots + h_m)$
    Update $\alpha_m$ values to minimize error
    Calculate values of $W_m$ for the next iteration
```

3 Mathematical Derivation of Update Rules

Adaboost uses exponential loss. Exponential loss is defined to be

$$\text{Loss}(y, h(x)) = e^{-yh(x)}$$

where y is the label and $h(x)$ is our predicted label. It has a shape that is very similar to that of logistic loss. Since our classifiers are binary, the only actual values of exponential loss we'll end up using are e and e^{-1}. We see that if the label and the prediction match up in sign, then our loss is e^{-1}, which is small. If the label and the prediction to not match in sign, then our loss is e, which is large.

Suppose we have $m - 1$ classifiers already and we are now searching for α_m and h_m to minimize the training error. For convenience, let us define

$$\text{pred}_m(\bar{x}) = \text{sign}\left(\sum_{i=1}^{m} h_i(\bar{x})\right)$$

which is essentially the prediction of the Adaboost algorithm at iteration m.

3.1 Update 1: Finding $h_m^*(x)$

For the first step of the iterative updates in the Adaboost algorithm, it is left to the programmer to figure out how to find an optimal h_m^* that minimizes the weighted error from applying pred_m to classify the training data. We cannot compute this because our derivations allow any general h_m^* which could be a SVM, a simple linear classifier, or anything else.

3.2 Update 2: Finding α_m

For the second step of the iterative updates, we wish to calculate an α_t and along the way, we will define what our weights W are. Our training loss using exponential loss is

$$J(\alpha_m, \text{pred}_m) = \sum_{i=1}^{n} e^{-y^{(i)}\text{pred}_m(\bar{x}^{(i)})} = \sum_{i=1}^{n} e^{-y^{(i)}(\text{pred}_{m-1}(\bar{x}^{(i)}) + \alpha_m h_m(\bar{x}^{(i)}))}$$

$$\sum_{i=1}^{n} e^{-y^{(i)}\text{pred}_{m-1}(\bar{x}^{(i)})} e^{-y^{(i)}\alpha_m h_m(\bar{x}^{(i)})}$$

Let us define our unnormalized weights of training samples from the previous iteration to be

$$W_{m-1}^{(i)} = e^{-y^{(j)}\text{pred}_{m-1}(\bar{x}^{(j)})}$$

We can now normalize the weights. Define the normlization constant $z_{m-1} = \sum_{i=1}^{n} W_{m-1}^{(i)}$. Then the normalized weight $\widetilde{W}_{m-1}^{(i)}$ is

$$\widetilde{W}_{m-1}^{(i)} = \frac{W_{m-1}^{(i)}}{z_{m-1}}$$

Substituting the weights into the equation for the loss function gives us

$$J(\alpha_m, \text{pred}_m) = \sum_{i=1}^{n} z_{m-1} \widetilde{W}_{m-1}^{(i)} e^{-y^{(i)} \alpha_m h_m(\bar{x}^{(i)})}$$

We can split the expression for J into halves: one half is for the samples we classify correctly and one half is for the samples we classify incorrectly. For the samples we classify correctly, there will be a $-\alpha$ term in the exponent because the sign between $y^{(i)}$ and $\text{pred}_m(\bar{x})$ will cancel. For the samples we classify incorrectly, there we be an α term in the exponent because the sign between $y^{(i)}$ and $\text{pred}_m(\bar{x})$ will not cancel. We can factor this α term out. We have

$$\begin{aligned} J(\alpha_m, h_m) &= e^{-\alpha_m} \sum_{\text{correct}} \widetilde{W}_{m-1}^{(i)} + e^{\alpha_m} \sum_{\text{wrong}} \widetilde{W}_{m-1}^{(i)} \\ &= \left(e^{\alpha_m} - e^{-\alpha_m}\right) \sum_{i=1}^{n} \widetilde{W}_{m-1}^{(i)} \left[\left[y^{(i)} \neq h_m(\bar{x}^{(i)})\right]\right] + e^{-\alpha_m} \sum_{i=1}^{n} \widetilde{W}_{m-1}^{(i)} \end{aligned}$$

Note that we first overcount by multiplying $e^{-\alpha_m}$ by the weights of correctly-classified, but we accommodate this by adding those terms back in the last term of the summation.

The second summation conveniently becomes 1, since our weights are normalized. Since the last term becomes a constant $e^{-\alpha_m}$, we can drop it when we perform minimization. We now minimize the following function:

$$J(\alpha_m, h_m) = (e^{\alpha_m} - e^{-\alpha_m}) \sum_{i=1}^{N} \widetilde{W}_{m-1}^{(i)} \left[\left[y^{(i)} \neq h_m(\bar{x}^{(i)})\right]\right]$$

Notice that the summation is essentially the error on misclassified samples, ϵ_m. If we let

$$\hat{\epsilon}_m = \sum_{i=1}^{N} \widetilde{W}_{m-1}^{(i)} \left[\left[y^{(i)} \neq h_m(\bar{x}^{(i)})\right]\right]$$

be the minimum error possible by using the optimal h_m^* classifier on this iteration, then we have

$$J(\alpha_m, h_m^*) = L(\alpha_m) = (e^{\alpha_m} - e^{-\alpha_m}) \hat{\epsilon}_m$$

We can minimize L with respect to α_m by taking the derivative and setting it to 0. We have

$$\frac{dL}{dm} = (e^{\alpha_m} + e^{-\alpha_m}) \hat{\epsilon}_m - e^{-\alpha_m} = 0$$

Rearranging, we see that the minimum error bound for J is achieved by setting

$$\alpha_m^* = \frac{1}{2} \ln \left(\frac{1 - \epsilon_m}{\epsilon^m} \right)$$

3.3 Updating Weights W_m

Finally, we must update the weights for each of the training samples. Recall that our definition of the weight W_m was given by

$$W_m^{(i)} = e^{-y^{(i)} \text{pred}_m(\bar{x}^{(i)})}$$

We can rewrite the pred_m term as

$$\text{pred}_m(\bar{x}^{(i)}) = \text{pred}_{m-1}(\bar{x}^{(i)}) + \alpha_m^* h_m^*(\bar{x}^{(i)})$$

This gives us

$$W_m^{(i)} = e^{-y^{(i)} \text{pred}_{m-1}(\bar{x}^{(i)})} e^{-y^{(i)} \alpha_m^* h_m^*(\bar{x}^{(i)})} = W_{m-1}^{(i)} e^{-y^{(i)} \alpha_m^* h_m^*(\bar{x}^{(i)})}$$

Then our normalized weight will be

$$\widetilde{W}_m^{(i)} = \frac{W_{m-1}^{(i)}}{z_{m-1}} e^{-y^{(i)} \alpha_m^* h_m^*(\bar{x}^{(i)})}$$

and this is the weight that will be used for the training point on iteration $m + 1$.

3.4 Additional Observations About Training Weight

There are some useful mathematical manipulations that will might make it easier for us to work with the numbers in the Adaboost updates.

As we showed previously, the update for the weights of training samples is given by

$$\widetilde{W}_m^{(i)} = \frac{\widetilde{W}_{m-1}^{(i)}}{z_{m-1}} e^{-y^{(i)}\alpha_m^* h_m^*(\bar{x}^{(i)})}$$

We can substitute in our value of α_m^* and get

$$\widetilde{W}_m^{(i)} = \frac{\widetilde{W}_{m-1}^{(i)}}{z_{m-1}} e^{-y^{(i)} h_m^*(\bar{x}^{(i)}) \frac{1}{2}\ln\left(\frac{1-\epsilon_m}{\epsilon_m}\right)}$$

If we split the expression into the cases where the classifier h_m^* predicts the correct class versus when the classifier predicts the incorrect class, we rewrite the expression

$$\widetilde{W}_m^{(i)} = \frac{\widetilde{W}_{m-1}^{(i)}}{z_{m-1}} \times \begin{cases} e^{-\frac{1}{2}\ln\left(\frac{1-\epsilon_m}{\epsilon_m}\right)} & \text{if correct} \\ e^{\frac{1}{2}\ln\left(\frac{1-\epsilon_m}{\epsilon_m}\right)} & \text{if wrong} \end{cases} = \frac{\widetilde{W}_{m-1}^{(i)}}{z_{m-1}} \times \begin{cases} \sqrt{\frac{\epsilon_m}{1-\epsilon_m}} & \text{if correct} \\ \sqrt{\frac{1-\epsilon_m}{\epsilon_m}} & \text{if wrong} \end{cases}$$

4 Properties of Adaboost

4.1 Adding Complexity

If used correctly, Adaboost adds complexity on every iteration. We can think of the weak classifiers $h_1, h_2, \ldots h_m$ as different features of the data. On every iteration, a new weak classifier is introduced. However, the rate at which complexity grows is much slower than that of other classifiers such as an SVM using the RBF kernel.

4.2 Decreasing Error Bound

Since we are minimizing the loss function J which uses exponential loss, our cost should be decreasing every iteration and if our training dataset is reasonable, the cost should approach 0 over many iterations as the classifier adds more and more complexity.

4.3 Guarding against Overfitting

Empirically, Adaboost performs well against overfitting. After running Adaboost for many iterations and plotting the test error versus the training error, we typically see that both errors are generally decreasing. There is no rigorous explanation yet as to why this happens.

One view is that Adaboost can be seen as a form of L_1 regularization which eliminates irrelevant features. Sometimes, Adaboost introduces features that happen to overfit to noise, but eventually, the coefficients of those features are zeroed out.

Another view is that Adaboost can be proven to be maximizing a voting margin. As with SVMs, by maximizing a margin, our classifier generalizes better and performs well on the test set.

4.4 Why Adaboost is Powerful

One of the reasons that Adaboost is powerful is that it does not overfit even as the classifier becomes more and more complex. Other classifiers such as SVMs or Random Forests can seriously overfit the data when too many features are introduced. Overall, if we're using Adaboost, we are in a way adding an unlimited number of features without having to worry about overfitting. This is a very useful property.