CS 189 Introduction to Machine Learning
Spring 2018 Note 26

1 Boosting

We have seen that in the case of random forests, combining many imperfect models can produce a single model that works very well. This is the idea of **ensemble methods**. However, random forests treat each member of the forest equally, taking a plurality vote or an average over their outputs. The idea of **boosting** is to combine the models (typically called *weak learners* in this context) in a more principled manner. The key idea is as follows: to improve our combined model, we should focus on finding learners that correctly predict the points which the overall boosted model is currently predicting inaccurately. Boosting algorithms implement this idea by associating a weight with each training point and iteratively reweighting so that mispredicted points have relatively high weights. Intuitively, some points are "harder" to predict than others, so the algorithm should focus its efforts on those.

These ideas also connect to matching pursuit. In both cases, our overall predictor is an additive combination of pieces which are selected one-by-one in a greedy fashion. The algorithm keeps track of residual prediction errors, chooses the "direction" to move based on these, and then performs a sort of line search to determine how far along that direction to move.

1.1 AdaBoost

There are many flavors of boosting. We will discuss one of the most popular versions, known as **AdaBoost** (short for *adaptive boosting*), which is a method for binary classification. Its developers won the prestgious Gödel Prize for this work.

1.2 Algorithm

We present the algorithm first, then derive it later. Assume access to a dataset $\{(\mathbf{x}_i, y_i)\}_{i=1}^n$, where $\mathbf{x}_i \in \mathbb{R}^d$ and $y_i \in \{-1, 1\}$.

1. Initialize the weights $w_i = \frac{1}{n}$ for all $i = 1, \dots, n$ training points.

2. Repeat for $m = 1, \dots, M$:

   (a) Build a classifier $G_m : \mathbb{R}^d \to \{-1, 1\}$, where in the training process the data are weighted according to w_i.

   (b) Compute the weighted error $e_m = \frac{\sum_{i \text{ misclassified}} w_i}{\sum_i w_i}$.

(c) Re-weight the training points as

$$w_i \leftarrow w_i \cdot \begin{cases} \sqrt{\frac{1-e_m}{e_m}} & \text{if misclassified by } G_m \\ \sqrt{\frac{e_m}{1-e_m}} & \text{otherwise} \end{cases}$$

(d) Optional: normalize the weights w_i to sum to 1.

We first address the issue of step (a): how do we train a classifier if we want to weight different samples differently? One common way to do this is to resample from the original training set every iteration to create a new training set that is fed to the next classifier. Specifically, we create a training set of size n by sampling n values from the original training data with replacement, according to the distribution w_i. (This is why we might renormalize the weights in step (d).) This way, data points with large values of w_i are more likely to be included in this training set, and the next classifier will place higher priority on such data points.

Suppose[1] that our weak learners always produce an error $e_m < \frac{1}{2}$. To make sense of the formulas we see in the algorithm, note that for step (c), if the i-th data point is misclassified, then the weight w_i gets increased by a factor of $\sqrt{\frac{1-e_m}{e_m}}$ (more priority placed on sample i), while if it is classified correctly, the priority gets decreased. AdaBoost does have a practical weakness in that this aggressive reweighting can cause the classifier to focus too much on certain training examples – if the data contains outliers or a lot of noise, the boosting algorithm's generalization performance may suffer as it overfits to a few challenging examples.

We have not yet discussed how to make a prediction on test points given our classifiers $G_1, \ldots, G_M$. One conceivable method is to use logistic regression with $G_m(\mathbf{x})$ as features. However, a smarter choice that is based on the AdaBoost algorithm is to set

$$\alpha_m = \frac{1}{2} \ln \left(\frac{1-e_m}{e_m} \right)$$

and classify $\mathbf{x}$ by

$$h(\mathbf{x}) = \operatorname{sgn}\left(\sum_{m=1}^{M} \alpha_m G_m(\mathbf{x}) \right)$$

Note that this choice of α_m (derived later) gives high weight to classifiers that have low error:

- As $e_m \to 0$, $\frac{1-e_m}{e_m} \to \infty$, so $\alpha_m \to \infty$.
- As $e_m \to 1$, $\frac{1-e_m}{e_m} \to 0$, so $\alpha_m \to -\infty$.

We now proceed to demystify the formulas in the algorithm above by presenting a matching pursuit interpretation of AdaBoost. This interpretation is also useful because it generalizes to a powerful technique called Gradient Boosting, of which AdaBoost is just one instance.

[1] This is a reasonable thing to ask. A classifier with error $e_m \geq \frac{1}{2}$ is even worse than the trivial classifier which predicts the class with the most total weight without regard for the input $\mathbf{x}_i$.

1.3 Derivation of AdaBoost

Suppose we have computed classifiers $G_1, \dots, G_{m-1}$ along with their corresponding weights α_k and we want to compute the next classifier G_m along with its weight α_m. The output of our model so far is $F_{m-1}(\mathbf{x}) := \sum_{i=1}^{m-1} \alpha_i G_i(\mathbf{x})$, and we want to minimize the risk:

$$\alpha_m, G_m = \arg\min_{\alpha, G} \sum_{i=1}^{n} L(y_i, F_{m-1}(\mathbf{x}_i) + \alpha G(\mathbf{x}_i))$$

for some suitable loss function $L(y, \hat{y})$. Loss functions we have previously used include mean squared error for linear regression, cross-entropy loss for logistic regression, and hinge loss for SVM. For AdaBoost, we use the **exponential loss**:

$$L(y, \hat{y}) = e^{-y\hat{y}}$$

This loss function is illustrated in Figure 1. Observe that if $y\hat{y} > 0$ (i.e. $\hat{y}$ has the correct sign), the loss decreases exponentially in $|\hat{y}|$, which should be interpreted as the confidence of the prediction. Conversely, if $y\hat{y} < 0$, our loss is increasing exponentially in the confidence of the prediction.

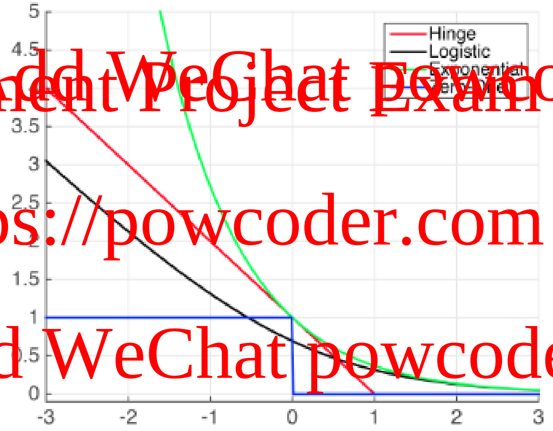

Figure 1: The exponential loss provides exponentially increasing penalty for confident incorrect predictions. This figure is from Cornell CS4780 notes.

Plugging the exponential loss into the general optimization problem above yields

$$\begin{aligned}\alpha_m, G_m &= \arg\min_{\alpha, G} \sum_{i=1}^{n} e^{-y_i(F_{m-1}(\mathbf{x}_i) + \alpha G(\mathbf{x}_i))} \\ &= \arg\min_{\alpha, G} \sum_{i=1}^{n} e^{-y_i F_{m-1}(\mathbf{x}_i)} e^{-y_i \alpha G(\mathbf{x}_i)}\end{aligned}$$

The term $w_i^{(m)} := e^{-y_i F_{m-1}(\mathbf{x}_i)}$ is a constant with respect to our optimization variables. We can split out this sum into the components with correctly classified points and incorrectly classified points:

$$\alpha_m, G_m = \arg\min_{\alpha, G} \sum_{i=1}^{n} w_i^{(m)} e^{-y_i \alpha G(\mathbf{x}_i)}$$

$$\begin{aligned}
&= \arg\min_{\alpha,G} \sum_{y_i=G(\mathbf{x}_i)} w_i^{(m)} e^{-\alpha} + \sum_{y_i\neq G(\mathbf{x}_i)} w_i^{(m)} e^{\alpha} && (*)\\
&= \arg\min_{\alpha,G} e^{-\alpha}\left(\sum_{i=1}^{n} w_i^{(m)} - \sum_{y_i\neq G(\mathbf{x}_i)} w_i^{(m)}\right) + e^{\alpha} \sum_{y_i\neq G(\mathbf{x}_i)} w_i^{(m)}\\
&= \arg\min_{\alpha,G} (e^{\alpha}-e^{-\alpha}) \sum_{y_i\neq G(\mathbf{x}_i)} w_i^{(m)} + e^{-\alpha}\sum_{i=1}^{n} w_i^{(m)}
\end{aligned}$$

To arrive at $(*)$ we have used the fact that $y_i G_m(\mathbf{x}_i)$ equals 1 if the prediction is correct, and -1 otherwise. For a fixed value of α, the second term in this last expression does not depend on G. Thus we can see that the best choice of $G_m(\mathbf{x})$ is the classifier that minimizes the total weight of the misclassified points. Let

$$e_m = \frac{\sum_{y_i\neq G_m(\mathbf{x}_i)} w_i^{(m)}}{\sum_i w_i^{(m)}}$$

Once we have obtained G_m, we can solve for α_m. Dividing $(*)$ by the constant $\sum_{i=1}^{n} w_i^{(m)}$, we obtain

$$\alpha_m = \arg\min_{\alpha} (1-e_m)e^{-\alpha} + e_m e^{\alpha}$$

We can solve for the minimizer analytically using calculus. Setting the derivative of the objective function to zero gives

$$0 = -(1-e_m)e^{-\alpha} + e_m e^{\alpha} = -e^{-\alpha} + e_m(e^{-\alpha} + e^{\alpha})$$

Multiplying through by e^{α} yields

$$0 = -1 + e_m(1 + e^{2\alpha})$$

Adding one to both sides and dividing by e_m, we have

$$\frac{1}{e_m} = 1 + e^{2\alpha}$$

i.e.

$$e^{2\alpha} = \frac{1}{e_m} - 1 = \frac{1-e_m}{e_m}$$

Taking natural log on both sides and halving, we arrive at

$$\alpha_m = \frac{1}{2}\ln\left(\frac{1-e_m}{e_m}\right)$$

as claimed earlier. From the optimal α_m, we can derive the weights:

$$\begin{aligned}
w_i^{(m+1)} &= \exp\left(-y_i F_m(\mathbf{x}_i)\right)\\
&= \exp\left(-y_i[F_{m-1}(\mathbf{x}_i) + \alpha_m G_m(\mathbf{x}_i)]\right)\\
&= w_i^{(m)} \exp\left(-y_i G_m(\mathbf{x}_i)\alpha_m\right)
\end{aligned}$$

$$
\begin{aligned}
&= w_i^{(m)} \exp\left(-y_i G_m(\mathbf{x}_i)\frac{1}{2}\ln\left(\frac{1-e_m}{e_m}\right)\right) \\
&= w_i^{(m)} \exp\left(\ln\left[\left(\frac{1-e_m}{e_m}\right)^{-\frac{1}{2}y_i G_m(\mathbf{x}_i)}\right]\right) \\
&= w_i^{(m)} \left(\frac{1-e_m}{e_m}\right)^{-\frac{1}{2}y_i G_m(\mathbf{x}_i)} \\
&= w_i^{(m)} \sqrt{\frac{e_m}{1-e_m}}^{\,y_i G_m(\mathbf{x}_i)}
\end{aligned}
$$

Here we see that the multiplicative factor is $\sqrt{\frac{e_m}{1-e_m}}$ when $y_i = G_m(\mathbf{x}_i)$ and $\sqrt{\frac{1-e_m}{e_m}}$ otherwise. This completes the derivation of the algorithm.

As a final note about the intuition, we can view these α updates as pushing towards a solution in some direction until we can no longer improve our performance. More precisely, whenever we compute α_m (and thus $w^{(m+1)}$), for the incorrectly classified entries, we have

$$
\sum_{y_i \neq G_m(\mathbf{x}_i)} w_i^{(m+1)} = \sum_{y_i \neq G_m(\mathbf{x}_i)} w_i^{(m)} \sqrt{\frac{1-e_m}{e_m}}
$$

Dividing the right-hand side by $\sum_{i=1}^n w_i^{(m)}$, we obtain $e_m\sqrt{\frac{1-e_m}{e_m}} = \sqrt{e_m(1-e_m)}$. Similarly, for the correctly classified entries, we have

$$
\frac{\sum_{y_i = G_m(\mathbf{x}_i)} w_i^{(m+1)}}{\sum_{i=1}^n w_i^{(m)}} = (1-e_m)\sqrt{\frac{e_m}{1-e_m}} = \sqrt{e_m(1-e_m)}
$$

Thus these two quantities are the same once we have adjusted our α, so the misclassified and correctly classified sets both get equal total weight.

This observation has an interesting practical implication. Even after the training error goes to zero, the test error may continue to decrease. This may be counter-intuitive, as one would expect the classifier to be overfitting to the training data at this point. One interpretation for this phenomenon is that even though the boosted classifier has achieved perfect training error, it is still refining its fit in a max-margin fashion, which increases its generalization capabilities.

1.4 Gradient Boosting

AdaBoost assumes a particular loss function, the exponential loss function. **Gradient boosting** is a more general technique that allows an arbitrary differentiable loss function $L(y, \hat{y})$. Recall the general optimization problem we must solve when choosing the next model:

$$
\min_{\alpha, G} \sum_{i=1}^{n} L(y_i, F_{m-1}(\mathbf{x}_i) + \alpha G(\mathbf{x}_i))
$$

Here G should no longer be assumed to be a classifier; it may be real-valued if we are solving a regression problem. By a Taylor expansion in the second argument,

$$L(y_i, F_{m-1}(\mathbf{x}_i) + \alpha G(\mathbf{x}_i)) \approx L(y_i, F_{m-1}(\mathbf{x}_i)) + \frac{\partial L}{\partial \hat{y}}(y_i, F_{m-1}(\mathbf{x}_i)) \cdot \alpha G(\mathbf{x}_i)$$

We can view the collection of predictions that a model G produces for the training set as a single vector $\mathbf{g} \in \mathbb{R}^n$ with components $g_i = G(\mathbf{x}_i)$. Then the overall cost function is approximated to first order by

$$\sum_{i=1}^{n} L(y_i, F_{m-1}(\mathbf{x}_i)) + \alpha \underbrace{\sum_{i=1}^{n} \frac{\partial L}{\partial \hat{y}}(y_i, F_{m-1}(\mathbf{x}_i)) \cdot g_i}_{\langle \nabla_{\hat{y}} L(\mathbf{y}, F_{m-1}(\mathbf{X})), \mathbf{g} \rangle}$$

where, in abuse of notation, $F_{m-1}(\mathbf{X})$ is a vector with $F_{m-1}(\mathbf{x}_i)$ as its ith element, and $\nabla_{\hat{y}} L(\mathbf{y}, \hat{\mathbf{y}})$ is a vector with $\frac{\partial L}{\partial \hat{y}}(y_i, \hat{y}_i)$ as its ith element. To decrease the cost in a steepest descent fashion, we seek the direction $\mathbf{g}$ which maximizes

$$|\langle -\nabla_{\hat{y}} L(\mathbf{y}, F_{m-1}(\mathbf{X})), \mathbf{g} \rangle|$$

subject to $\mathbf{g}$ being the output of some model G in the model class we are considering.[2]

Some comments are in order. First, observe that the loss need only be differentiable with respect to its inputs, not necessarily with respect to model parameters, so we can use non-differentiable models such as decision trees. Additionally, in the case of squared loss $L(y, \hat{y}) = \frac{1}{2}(y - \hat{y})^2$ we have

$$\frac{\partial L}{\partial \hat{y}}(y_i, F_{m-1}(\mathbf{x}_i)) = -(y_i - F_{m-1}(\mathbf{x}_i))$$

so

$$-\nabla_{\hat{y}} L(\mathbf{y}, F_{m-1}(\mathbf{X})) = \mathbf{y} - F_{m-1}(\mathbf{X})$$

This means the algorithm will follow the residual, as in matching pursuit.

[2] The absolute value may seem odd, but consider that after choosing the direction $\mathbf{g}_m$, we perform a line search to select α_m. This search may choose $\alpha_m < 0$, effectively flipping the direction. The key is to maximize the magnitude of the inner product.