1 AdaBoost

1.1 Algorithm

Recall the AdaBoost algorithm discussed last lecture:

Algorithm: AdaBoost (Adaptive Boosting)

1. Initialize all weights of the training points to $\frac{1}{n}$.

2. Repeat as $m$ goes from 1 . . . $M$:
   
   (a) Build a classifier with data weighted according to $w_i$.
   
   (b) Compute the weighted error $e_m = \frac{\sum_{i \text{ misclassified}} w_i}{\sum w_i}$ (this is computed on the original training set).
   
   (c) Re-weight the training points.

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

   (d) Normalize the weights $w_i$ to sum to 1 if necessary.

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 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 that our weak learners always produce an error $e_m < 1/2$. 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 weakness in that this aggressive reweighting can cause the classifier to focus too much on certain training examples - if the data is noisy with outliers, then this will weaken the boosting algorithm’s performance.

We have not yet discussed how to make a prediction given our classifiers (say, $G_1, . . . , G_m$). One conceivable method is to use logistic regression with $G_m(x)$ 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 $x$ as $\text{sign}(\sum_i \alpha_i G_i(x))$.

As an interesting practical note, once the training error goes to zero, the AdaBoost test error may still decrease upon performing further iterations of boosting. 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 a max-margin argument - 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.

We now proceed to demystify the formulas in the algorithm by presenting a matching pursuit interpretation of AdaBoost.

### 1.2 Derivation of AdaBoost Algorithm

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

$$\arg\min_{\alpha_m, G_m} \sum_i L(y_i, F_{m-1}(x) + \alpha_m G_m(x_i))$$

for some suitable loss function $L$. Loss functions we have previously used include mean squared error for linear regression, log-loss for logistic regression, and hinge loss for SVM. For AdaBoost, we use the exponential loss:

$$L(y, h(x)) = e^{-yh(x)}$$

This loss function is illustrated in Figure 1. We have exponential decay as we increase the input - thus if $yh(x)$ is large and positive (so $h(x)$ has the correct sign and high magnitude), our loss is decreasing exponentially. Conversely, if $yh(x)$ is a large negative value, our loss is increasing exponentially, and thus we are heavily penalized for confidently making an incorrect prediction.

![Figure 1](image.png)

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

We can write the AdaBoost optimization problem with exponential loss as follows:

$$\arg\min_{\alpha_m, G_m} \sum_i e^{-y_i(F_{m-1}(x) + \alpha_m G_m(x_i))}$$

$$= \arg\min_{\alpha_m, G_m} \sum_i e^{-y_i F_{m-1}(x)} e^{-\alpha_m G_m(x_i)}$$
The term \( w_i^{(m)} := e^{-y_i F_{m-1}(x)} \) 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:

\[
= \arg \min_{\alpha_m, G_m} \sum_{\{i : y_i = G_m(x)\}} w_i^{(x)} e^{-\alpha_m} + \sum_{\{i : y_i \neq G_m(x)\}} w_i^{(x)} e^{\alpha_m} \quad (*)
\]

\[
= \arg \min_{\alpha_m, G_m} (e^{\alpha_m} - e^{-\alpha_m}) \sum_{\{i : y_i \neq G_m(x)\}} w_i^{(m)} + e^{-\alpha_m} \sum_{i=1}^{n} w_i^{(m)}
\]

If we fix \( \alpha_m \), the second term is constant that does not depend on our optimization variables. Thus we can see that the best choice of \( G_m(x) \) is the classifier that minimizes the error given the weights \( w_i^{(m)} \). Let \( e_m \) be defined as in the AdaBoost algorithm. We can solve for \( \alpha_m \): by dividing \((*)\) by the constant \( \sum_{i=1}^{n} w_i^{(m)} \), if we fix \( G_m \) the problem becomes

\[
= \arg \min_{\alpha_m} (1 - e_m) e^{-\alpha_m} + e_m e^{\alpha_m}
\]

By differentiating with respect to \( \alpha_m \), we obtain the solution

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

as desired.

From the optimal \( \alpha_m \), we can derive the weights:

\[
w_i^{(m+1)} = e^{-y_i F_m(x_i)} = e^{-y_i[F_{m-1}(x_i) + \alpha_m G_m(x_i)]} = w_i^{(m)} e^{-y_i G_m(x_i)} \alpha_m
\]

If we substitute the value of \( \alpha_m \) we just found, we see that the multiplicative factor is \( \sqrt{\frac{e_m}{1 - e_m}} \) when \( y_i = G_m(x_i) \) 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 \( \alpha \) updates as pushing towards a solution in some direction until we can no longer improve our performance. More precisely, whenever we compute \( \alpha_m \) (and thus \( w^{(m+1)} \)), for the incorrectly classified entries, we have

\[
\sum_{y_i \neq G_m(x_i)} w_i^{(m+1)} = \sum_{y_i \neq G_m(x_i)} w_i \sqrt{\frac{1 - e_m}{e_m}}
\]

If we divide this by \( \sum_{i=1}^{n} w_i^{(m)} \), then this becomes \( e_m \sqrt{\frac{1 - e_m}{e_m}} = \sqrt{e_m (1 - e_m)} \). Similarly, for the correctly classified entries, we have

\[
\sum_{y_i = G_m(x_i)} w_i^{(m+1)} = (1 - e_m) \sqrt{\frac{e_m}{1 - e_m}} = \sqrt{e_m (1 - e_m)}
\]

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