24 Boosting

ADABOOST (Yoav Freund and Robert Schapire, 1997)

[We're done with unsupervised learning. This week I'm going back to classifiers.]

AdaBoost ("adaptive boosting") is an ensemble method for classification (or regression) that

- trains multiple learners on weighted sample points [like bagging];
- uses different weights for each learner;
- increases weights of misclassified sample points;
- gives bigger votes to more accurate learners.

Input: $n \times d$ design matrix X, vector of labels $y \in \mathbb{R}^n$ with $y_i = \pm 1$.

Ideas:

- Train T classifiers $G_1, \ldots, G_T$.
- Weight for sample point X_i in G_t grows according to how many of $G_1, \ldots, G_{t-1}$ misclassified it.
  [Moreover, if X_i is misclassified by very accurate learners, its weight grows even more.]
  [And, the weight shrinks every time X_i is correctly classified.]
- Train G_t to try harder to correctly classify sample pts with larger weights.
- Metalearner is a linear combination of learners. For test point z, $M(z) = \sum_{t=1}^{T} \beta_t G_t(z)$.
  Each G_t is ± 1, but M is continuous. Return sign of $M(z)$.

[In the previous lecture on ensemble methods, I talked briefly about how to assign different weights to sample points. It varies for different learning algorithms. For example, in regression we usually modify the risk function by multiplying each point's loss function by its weight. In a soft-margin support vector machine, we modify the objective function by multiplying each point's slack by its weight.]
[Boosting works with most learning algorithms, but it was originally developed for decision trees, and boosted decision trees are still very popular and successful. For decision trees, we use a weighted entropy where instead of computing the proportion of points in each class, we compute the proportion of weight in each class.]

In iteration T, what classifier G_T and coefficient β_T should we choose? Pick a loss fn L(prediction, label).

Find G_T & β_T that minimize

$$\text{Risk} = \frac{1}{n} \sum_{i=1}^{n} L(M(X_i), y_i), \qquad M(X_i) = \sum_{t=1}^{T} \beta_t G_t(X_i).$$

AdaBoost metalearner uses exponential loss function

$$L(\rho, \ell) = e^{-\rho \ell} = \begin{cases} e^{-\rho} & \ell = +1 \\ e^{\rho} & \ell = -1 \end{cases}$$

[This loss function is for the metalearner only. The individual learners G_t usually use other loss functions, if they use a loss function at all.]
Important: label ℓ is binary, G_t is binary, but $\rho = M(X_i)$ is continuous!

[The exponential loss function has the advantage that it pushes hard against badly misclassified points. That's one reason why it's usually better than the squared error loss function for classification in a metalearner. It's similar to why in neural networks we prefer the cross-entropy loss function to the squared error.]

$$
\begin{aligned}
n \cdot \text{Risk} &= \sum_{i=1}^{n} L(M(X_i), y_i) = \sum_{i=1}^{n} e^{-y_i M(X_i)} \\
&= \sum_{i=1}^{n} \exp\left(-y_i \sum_{t=1}^{T} \beta_t G_t(X_i)\right) = \sum_{i=1}^{n} \prod_{t=1}^{T} e^{-\beta_t y_i G_t(X_i)} \qquad \Leftarrow \begin{array}{l} y_i G_t(X_i) = \pm 1 \\ -1 \to G_t \text{ misclassifies } X_i \end{array} \\
&= \sum_{i=1}^{n} w_i^{(T)} e^{-\beta_T y_i G_T(X_i)}, \quad \text{where } w_i^{(T)} = \prod_{t=1}^{T-1} e^{-\beta_t y_i G_t(X_i)} \\
&= e^{-\beta_T} \sum_{y_i = G_T(X_i)} w_i^{(T)} + e^{\beta_T} \sum_{y_i \neq G_T(X_i)} w_i^{(T)} \qquad \text{[correctly classified and misclassified]} \\
&= e^{-\beta_T} \sum_{i=1}^{n} w_i^{(T)} + (e^{\beta_T} - e^{-\beta_T}) \sum_{y_i \neq G_T(X_i)} w_i^{(T)}.
\end{aligned}
$$

What G_T minimizes the risk? The learner that minimizes the sum of $w_i^{(T)}$ over all misclassified pts X_i!

[This is interesting. By manipulating the formula for the risk, we've discovered what weight we should assign to each sample point. If we want to minimize the risk, we should find the classifier that minimizes the total weight of the misclassified points for this weight function $w_i^{(T)}$. It's a complicated function, but we can compute it. A useful observation is that each learner's weights are related to the previous learner's weights:]

Recursive definition of weights:

$$
w_i^{(T+1)} = w_i^{(T)} e^{-\beta_T y_i G_T(X_i)} = \begin{cases} w_i^{(T)} e^{-\beta_T} & y_i = G_T(X_i), \\ w_i^{(T)} e^{\beta_T} & y_i \neq G_T(X_i). \end{cases}
$$

[This recursive formulation is a nice benefit of the exponential loss. Notice that a weight shrinks if the point was classified correctly, and grows if the point was misclassified.]
[Now, you might wonder if we should just pick a learner that classifies all the training points correctly. But that's not always possible. If we're using a linear classifier on data that's not linearly separable, some points must be classified wrongly. Moreover, it's NP-hard to find the optimal linear classifier, so in practice G_T will be an approximate best learner, not the true minimizer of training error. But that's okay.]
[You might ask, if we use decision trees, can't we get 100% training accuracy? Usually we can. But interestingly, boosting is usually used with short, imperfect decision trees instead of tall, pure decision trees, for reasons I'll explain later.]
[Now, let's derive the optimal value of β_T.]

To choose β_T, set $\frac{\mathrm{d}}{\mathrm{d}\beta_T}\text{Risk} = 0$:

$$
\begin{aligned}
0 &= -e^{-\beta_T} \sum_{i=1}^{n} w_i^{(T)} + (e^{\beta_T} + e^{-\beta_T}) \sum_{y_i \neq G_T(X_i)} w_i^{(T)}; \qquad \text{[now divide both sides by the first term]} \\
0 &= -1 + (e^{2\beta_T} + 1)\,\text{err}_T, \quad \text{where } \text{err}_T = \frac{\sum_{y_i \neq G_T(X_i)} w_i^{(T)}}{\sum_{i=1}^{n} w_i^{(T)}}; \qquad [G_T\text{'s weighted error rate}] \\
\beta_T &= \frac{1}{2} \ln\left(\frac{1 - \text{err}_T}{\text{err}_T}\right).
\end{aligned}
$$

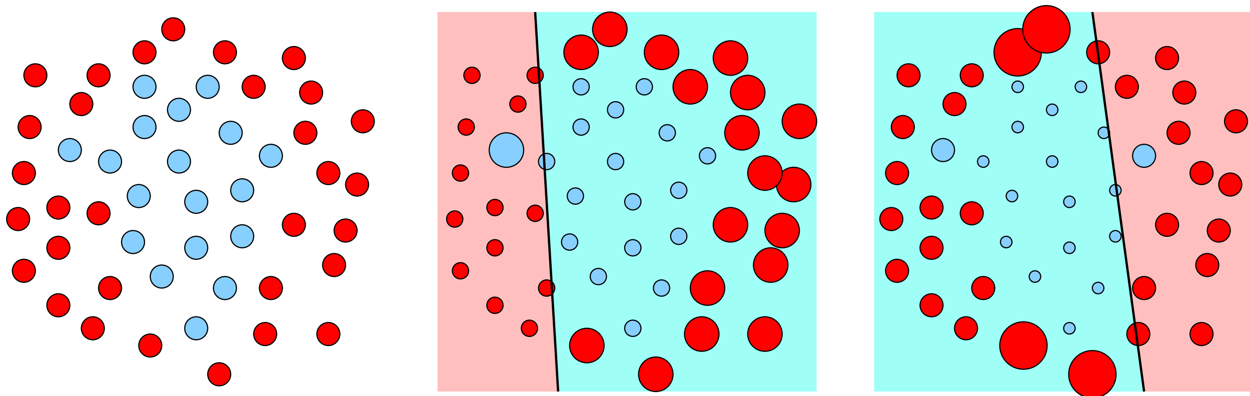
[So now we have derived the optimal metalearner!]

– If $\text{err}_T = 0, \beta_T = \infty$. [So a perfect learner gets an infinite vote.]
– If $\text{err}_T = 1/2, \beta_T = 0$. [So a learner with 50% training accuracy gets no vote at all.]

[More accurate learners get bigger votes in the metalearner. Interestingly, a learner with worse than 50% training accuracy gets a negative vote. A learner with 40% accuracy is just as useful as a learner with 60% accuracy; the metalearner just reverses the sign of its votes.]

[Now we can state the AdaBoost algorithm.]

AdaBoost alg:
1. Initialize weights $w_i \leftarrow \frac{1}{n}, \forall i \in [1, n]$.
2. for $t \leftarrow 1$ to T
   a. Train G_t with weights w_i
   b. Compute weighted error rate $\text{err} \leftarrow \frac{\sum_{\text{misclassified}} w_i}{\sum_{\text{all}} w_i}$; coefficient $\beta_t \leftarrow \frac{1}{2} \ln \left(\frac{1-\text{err}}{\text{err}} \right)$.
   c. Reweight pts: $w_i \leftarrow w_i \cdot \begin{cases} e^{\beta_t}, & G_t \text{ misclassifies } X_i \\ e^{-\beta_t}, & \text{otherwise} \end{cases} = w_i \cdot \begin{cases} \sqrt{\frac{1-\text{err}}{\text{err}}}, \\ \sqrt{\frac{\text{err}}{1-\text{err}}}. \end{cases}$
3. return metalearner $h(z) = \text{sign}\left(\sum_{t=1}^{T} \beta_t G_t(z)\right)$.

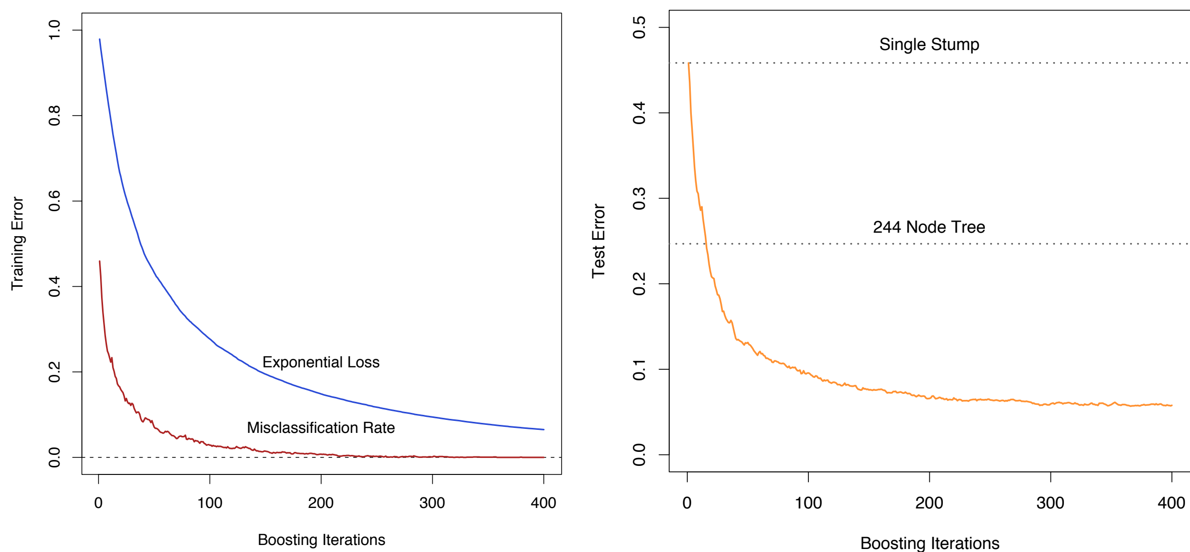


boost0.png, boost2.png, boost4.png [At left, all the training points have equal weight. After choosing a first linear classifier, we increase the weights of the misclassified points and decrease the weights of the correctly classified points (center). We train a second classifier with these weighted points, then again adjust the weights of the points according to whether they are misclassified by the second classifier.]

Why boost decision trees? [As opposed to other learning algorithms?] Why **short** trees?
– Fast. [We're training many learners, and running many learners at classification time too. Short decision trees that only look at a few features are very fast at both training and testing.]
– No hyperparameter search needed. [Unlike SVMs, neural nets, etc.] [UC Berkeley's Leo Breiman called AdaBoost with decision trees "the best off-the-shelf classifier in the world."]
– Easy to make a tree beat 55% training accuracy [or other threshold] consistently.
– Easy bias-variance control. Boosting can overfit. AdaBoost trees are usually short, to reduce overfitting!
  [As you train more learners, the bias decreases. The variance is more complicated: it often decreases at first, because successive trees focus on different features, but often it later increases. Sometimes boosting overfits after many iterations, and sometimes it doesn't; it's hard to predict when it will and when it won't.]

– AdaBoost + short trees is a form of subset selection.
  [Features that don't improve the metalearner's predictive power enough aren't used at all. This helps reduce overfitting and running time, especially if there are a lot of irrelevant features.]
– Linear decision boundaries don't boost well.
  [Boosting linear classifiers gives you an approximately linear classifier, so SVMs aren't a great choice. Methods with nonlinear decision boundaries benefit more from boosting, because they allow boosting to reduce the bias faster. Sometimes you'll see examples where people do AdaBoost with depth-one decision trees with just one decision each. But that's not ideal, because depth-one decision trees are linear. Even depth-two decision trees boost substantially better.]

More about AdaBoost:
– Posterior prob. can be approximated: $P(Y = 1|x) \approx 1/(1 + e^{-2M(x)})$.
– Exponential loss is vulnerable to outliers; for corrupted data, use other loss.
  [Better loss functions have been derived for dealing with outliers. Unfortunately, they have more complicated weight computations.]
– If every learner beats training accuracy μ for $\mu > 50\%$, metalearner accuracy will eventually be 100%.
– [The AdaBoost paper and its authors, Freund and Schapire, won the 2003 Gödel Prize, a prize for outstanding papers in theoretical computer science.]



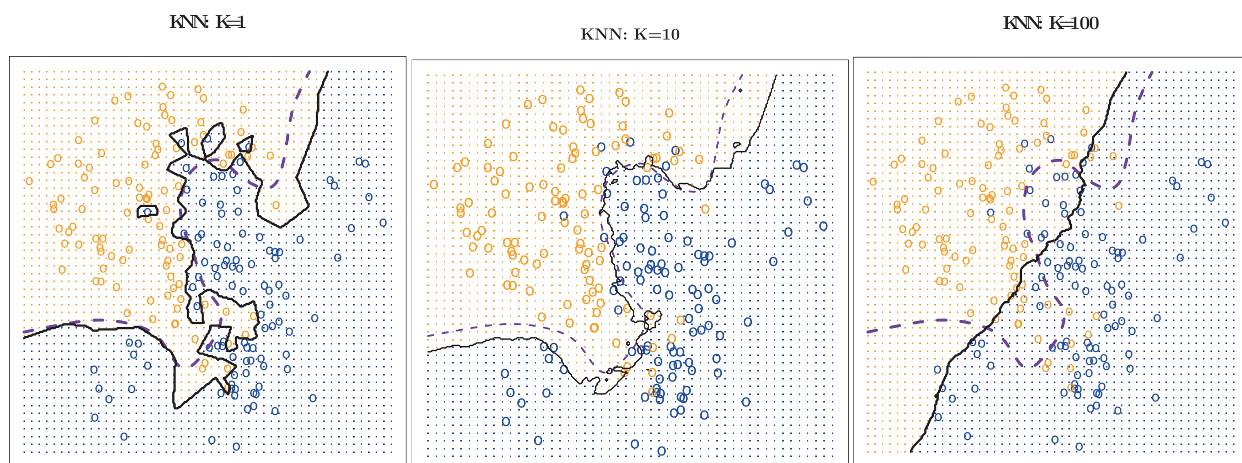
trainboost.pdf, testboost.pdf [Training and testing errors for AdaBoost with stumps, depth-one decision trees that make only one decision each. At left, observe that the training error eventually drops to zero, and even after that the average loss (which is continuous, not binary) continues to decay exponentially. At right, the test error drops to 5.8% after 400 iterations, even though each learner has an error rate of about 46%. AdaBoost with more than 25 stumps outperforms a single 244-node decision tree. In this example no overfitting is observed, but there are other datasets for which overfitting is a problem.]

NEAREST NEIGHBOR CLASSIFICATION

[I saved the simplest classifier for the end of the semester.]

Idea: Given query point q, find the k sample pts nearest q.
Distance metric of your choice.
Regression: Return average label of the k pts.
Classification: Return class with the most votes from the k pts OR
return histogram of class probabilities.

[The histogram of class probabilities tries to estimate the posterior probabilities of the classes. Obviously, the histogram has limited precision. If $k = 3$, then the only probabilities you'll ever return are 0, 1/3, 2/3, or 1. You can improve the precision by making k larger, but you might underfit. The histogram works best when you have a huge amount of data.]



allnn.pdf (ISL, Figures 2.15, 2.16) [Examples of 1-NN, 10-NN, and 100-NN. A larger k smooths out the boundary. In this example, the 1-NN classifier is badly overfitting the data, and the 100-NN classifier is badly underfitting. The 10-NN classifier does well: it's reasonably close to the Bayes decision boundary (purple). Generally, the ideal k depends on how dense your data is. As your data gets denser, the best k increases.]

[There are theorems showing that if you have a lot of data, nearest neighbors can work quite well.]

Theorem (Cover & Hart, 1967):
As $n \to \infty$, the 1-NN error rate is $< B(2 - B)$ where B = Bayes risk.
if only 2 classes, $\leq 2B(1 - B)$
[There are a few technical requirements of this theorem. The most important is that the training points and the test points all have to be drawn independently from the same probability distribution. The theorem applies to any separable metric space, so it's not just for the Euclidean metric.]

Theorem (Fix & Hodges, 1951):
As $n \to \infty$, $k \to \infty$, $k/n \to 0$, k-NN error rate converges to B. [Which means Bayes optimal.]