Classification: Boosting

Siva Balakrishnan
Data Mining: 36-462/36-662

February 19th, 2018

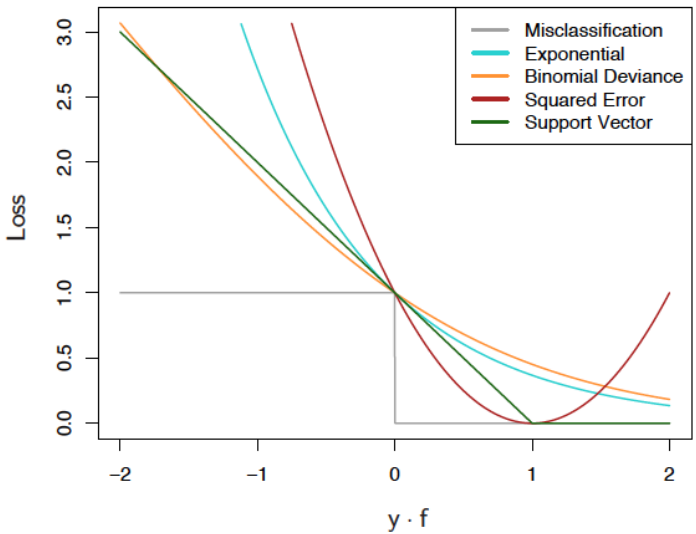
Chapter 8.3-8.4 of ISL and Chapter 10 of ESL

Announcements

- Riccardo will now have office hours on Thursday from 10am-11am in the BH 132 Lounge.

The Loss-Function View of Classification

Many classifiers we have studied so far (SVMs, logistic regression, linear regression) and ones that we will study today (Boosting) can be viewed as minimizing some loss function on the training data (usually with a regularizer).



$$\min \sum_{i=1}^{n} \max\{1 - yf, 0\}.$$

hinge loss

$$+\lambda \|\beta\|_2^2.$$

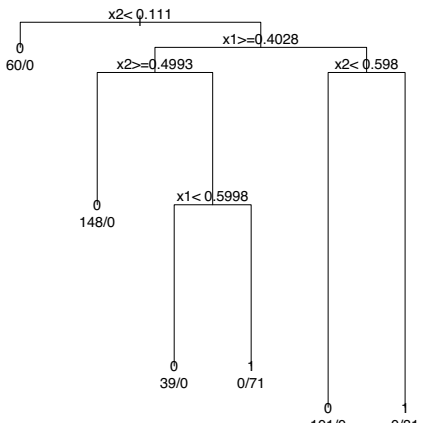
LR

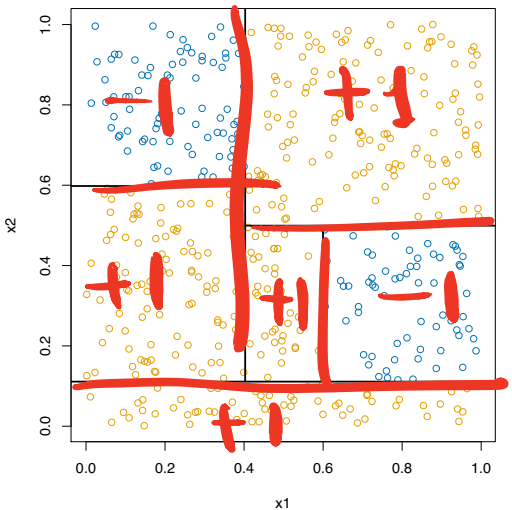
$$\min \sum_{i=1}^{n} -\log(y_i, x_i) + \lambda \|\beta\|_2^2$$

- What do you observe about all these losses?
- Why is this a nice thing?

Recap: Decision Trees Basics

- Key idea: Partition feature space into rectangles $\{R_1, \ldots, R_m\}$. Within each rectangle use a simple predictor of $y|X$.





- These are flexible, non-linear classifiers.

Recap: Simple Predictors

- **Regression:** Just use the average of points in the rectangle, i.e. for a new $x \in R_j$, we use

$$\widehat{f}(x) = \frac{1}{n_j} \sum_{i:x_i \in R_j} y_i.$$

- **Classification:** Just use the most likely class in the rectangle, i.e. for a new $x \in R_j$, we use

$$\widehat{f}(x) = \arg\max_k \widehat{p}_k(R_j),$$

where

$$\widehat{p}_k(R_j) = \frac{1}{n_j} \sum_{i:x_i \in R_j} \mathbb{I}(y_i = k).$$

fraction of pts in class k.

Recap: Growing Trees

- The CART algorithm grows trees greedily, i.e. at each step we find the (feature, location) to split at that "looks best" and then recurse.
- Formally, consider splitting on variable j and split point s, and define the regions

$$R_1 = \{X \in \mathbb{R}^p : X_j \leq s\}, \quad R_2 = \{X \in \mathbb{R}^p : X_j > s\}$$

We then greedily chooses j, s by minimizing the misclassification error

$$\underset{j,s}{\text{argmin}} \left(n_{R_1} \left[1 - \widehat{p}_{c_1}(R_1)\right] + n_{R_2} \left[1 - \widehat{p}_{c_2}(R_2)\right] \right)$$

Here $c_1 = \text{argmax}_{k=1,\ldots K} \ \widehat{p}_k(R_1)$ is the most common class in R_1, and $c_2 = \text{argmax}_{k=1,\ldots K} \ \widehat{p}_k(R_2)$ is the most common class in R_2.

Recap: Purity and Measuring Purity

- When we consider a split that minimizes the misclassification error we are essentially favoring splits that create "pure" rectangles (i.e. most points in each rectangle belong to a single class).
- In practice, we often use different measures of purity than the missclassification error. A common alternative is the *Gini Index*:

$$\text{Gini}(R_j) = \sum_{k=1}^{K} \widehat{p}_k(R_j)\left[1 - \widehat{p}_k(R_j)\right].$$

if pure then Gini $\simeq 0$

if impure then much larger

In this case, we would pick the split (greedily) that minimized:

$$\underset{j,s}{\text{argmin}} \left(n_{R_1} \text{Gini}(R_1) + n_{R_2} \text{Gini}(R_2) \right).$$

- There are other criteria (for instance, based on Entropy or Variance) that are also sometimes used.

Recap: Pruning

- (Fully-Grown) Decision Trees will almost always overfit. They will perfectly classify the training data but will not do so well on test data.
- There are two options:
  1. Stop early: i.e. do not grow the full tree.
  2. Grow the full tree and then prune it back to eliminate some splits.

  In practice, we always do (2).
- The precise details are somewhat complicated – but the main idea is:
  1. We grow a full tree.
  2. Create a collection of sub-trees of the full tree (i.e. trees that are obtained by pruning the original tree).
  3. Choose from the sub-trees using a validation set.

How well do trees predict?

Trees seem to have a lot of things going in the favor. So how is their predictive ability?

Unfortunately, the answer is not great.

Trees tend to suffer from high variance because they are quite unstable: a small change in the observed data can lead to a dramatically different sequence of splits, and hence a different prediction.

This instability comes from their greedy nature; once a split is made, it is permanent and can never be "unmade" further down in the tree

Can we fix trees?

- One way to fix the variance problem of trees is to grow (really) small trees. These are typically called decision stumps.
- What is the problem with just using stumps?

→ high bias.

only do a few splits.

- How can we fix this problem? → grow trees.

→ combine many stumps.

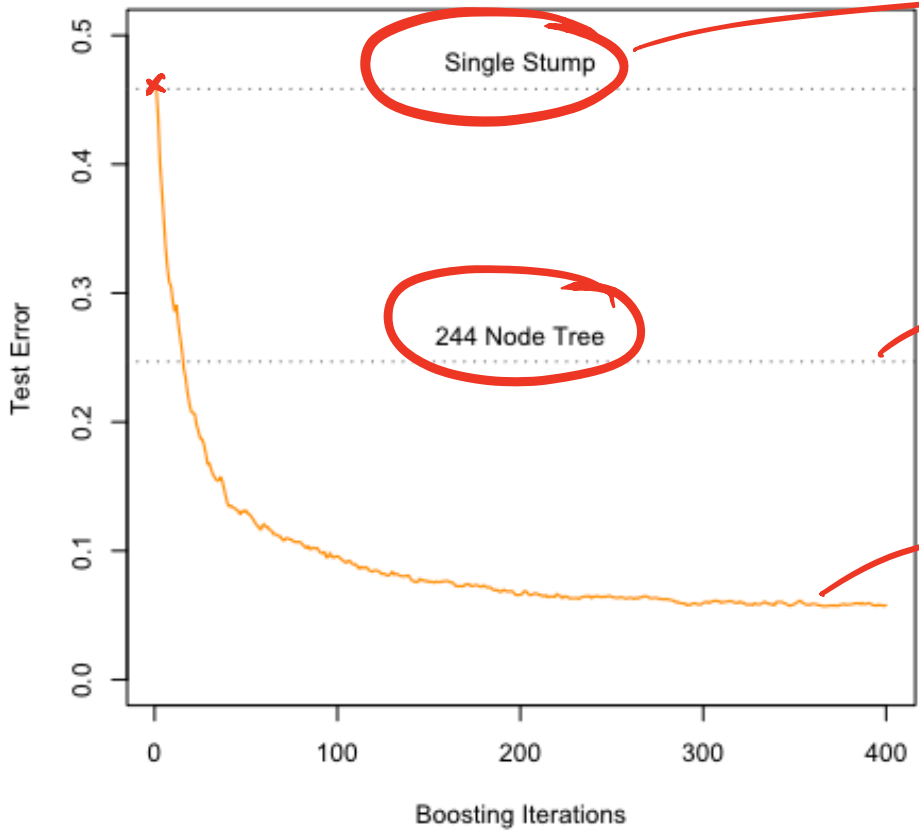
diverse.

Boosting

- Boosting is an (iterative) approach to combining many weak estimators into a powerful estimator.

- In *boosting*, we will build a sequence of simple prediction functions (usually trees). Each function will try to do well on observations that the previous functions did poorly on.

Boosting in an Example



high test error

0.25 test error

much better than a single tree.

Boosting: History

- Kearns asked an important question: can weak learners be improved?
- Suppose that you were able to classify any (reasonable) data-set just slightly better than random guessing (weak learning). Can we combine these weak learners to obtain ones that are *much* better than random?
- Schapire (and later Freund and Schapire) developed *boosting* algorithms to show that this was indeed possible.
- Their work had a tremendous amount of practical impact (for instance, won the Gödel prize) and got many different communities excited.

Boosting: High-Level

- Focus on binary classification with $y_i \in \{-1, +1\}$.
- Roughly, we are going to construct a classifier by voting or ensembling, i.e. we combine many (potentially) weak classifiers $\widehat{f}_1, \ldots, \widehat{f}_b$ to produce our classifier:

$$\widehat{f}(x) = \text{sign}\left(\sum_{t=1}^{b} \alpha_t \widehat{f}_t(x)\right),$$

  for some weights α_t.
- Two important questions:
  1. How do we obtain the classifiers $\widehat{f}_t$? We know they are "weak" (cannot be good everywhere) so we want to force them to learn about different inputs (diverse).
  2. How do we decide the weights α_t?

Original Boosting Algorithm (AdaBoost)

Initialize: Weights $w_i = 1/n$

For $b = 1, \ldots, B$**:**

1. Fit classification tree $\widehat{f}^b$ to the training data with weights $w_1, \ldots, w_n$.
2. Compute weighted misclassification error:

$$e_b = \frac{\sum_{i=1}^n w_i \mathbb{I}\{y_i \neq \widehat{f}^b(x_i)\}}{\sum_{i=1}^n w_i}$$

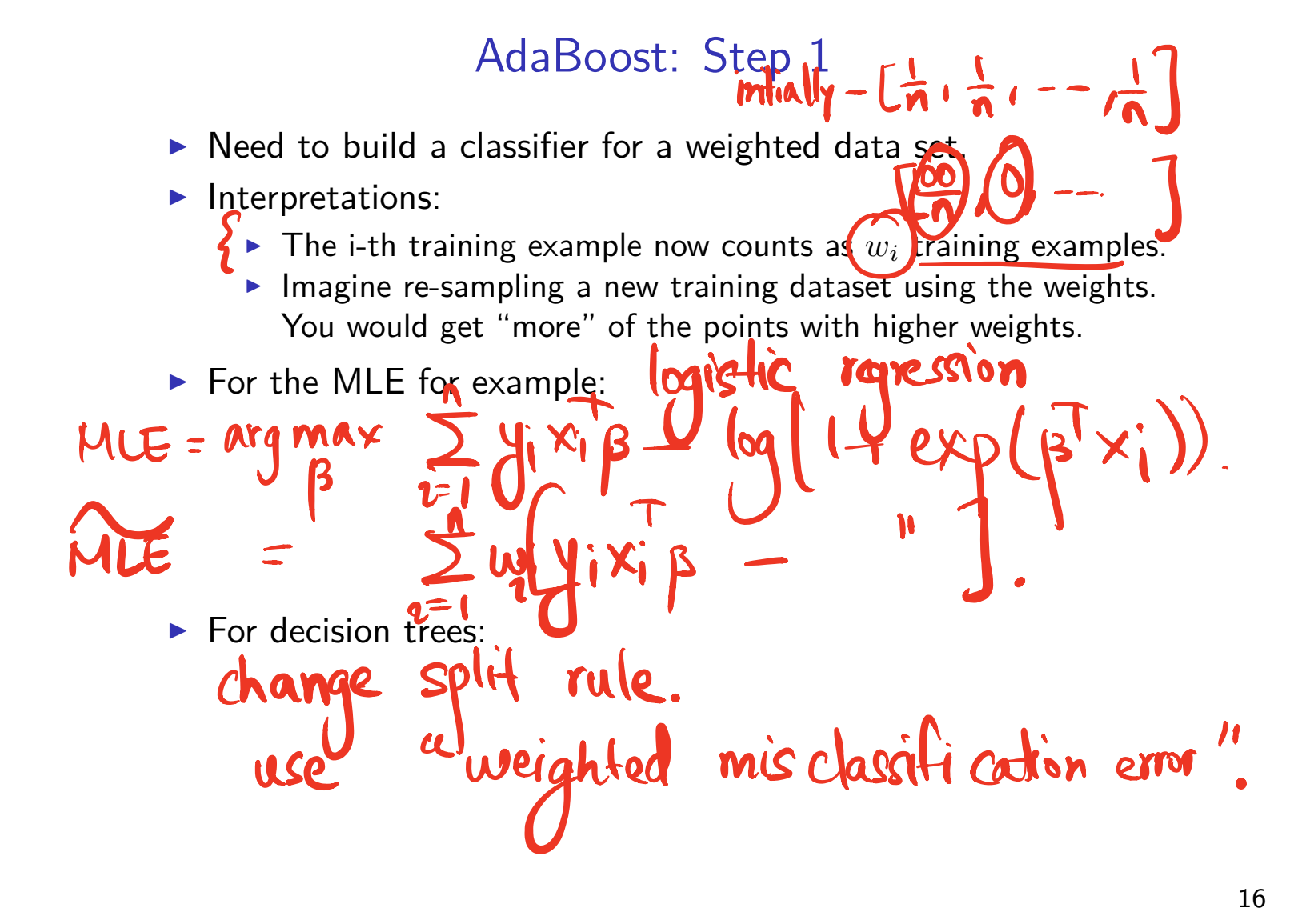
3. Define $\alpha_b = \log \frac{1-e_b}{e_b}$
4. Update the observation weights:

$$w_i \leftarrow w_i \cdot \exp\left(\alpha_b \mathbb{I}\{y_i \neq \widehat{f}^b(x_i)\}\right)$$

Result: $\widehat{f}(x) = \text{sign}\left(\sum_{b=1}^B \alpha_b \widehat{f}^b(x)\right)$

AdaBoost: Step 1

- Need to build a classifier for a weighted data set.
- Interpretations:
  - The i-th training example now counts as w_i training examples.
  - Imagine re-sampling a new training dataset using the weights. You would get "more" of the points with higher weights.
- For the MLE for example:
- For decision trees:

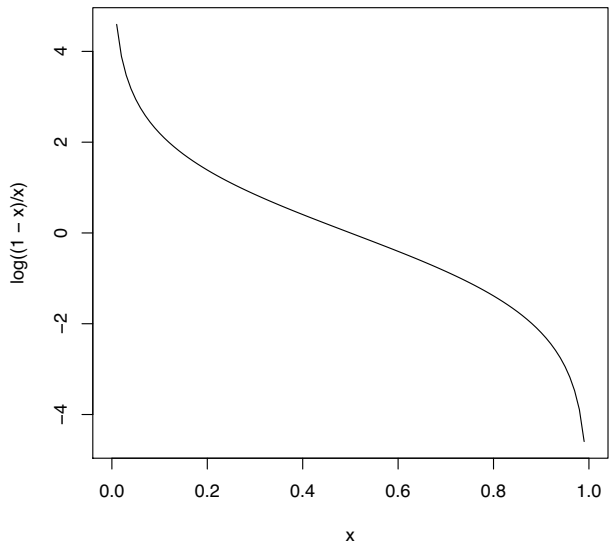


AdaBoost: Step 3

$$\alpha_b = \log \frac{1-e_b}{e_b}.$$

→ randomly guesses y_i.

$e_b = 1/2$, $\alpha_b = 0$.



→ $e_b \to 0$.

$\alpha_b \to \infty$.

This step stretches out probabilities near 0 and 1. It should remind you of logistic regression.

1. If a classifier is perfect what is its weight?
2. If it is just randomly guessing?

AdaBoost: Step 4

$$w_{i+1} \leftarrow w_i \cdot \exp\left(\alpha_b \mathbb{I}\{y_i \neq \widehat{f}^b(x_i)\}\right)$$

If y_i was guessed wrong, multiply weight by $\exp(\alpha_b)$.

If y_i was guessed right, multiply weight by $\exp(0) = 1$.

If α_b is big, the classifier did its job well.

If α_b is big, the weights increase more dramatically.

Adaboost intuition

We have enough for an intuition. Adaboost builds a sequence of trees, each of which tries to classify well on points that were missed by the earlier trees.

The observation weights, w_i, focus later classifiers on difficult points.

The classifier weights α_b capture how well each component classifier does its weighted task, and is used for both

- Adjusting the observation weights
- Weighting the components in the final classifier

But why exactly these weights and functions??

Why this form for AdaBoost?

Suppose that we care about misclassification error, or 0-1 loss. When $Y \in \{-1, 1\}$, we can rewrite this.

$$\widehat{f}(x) = \underset{f}{\operatorname{argmin}}\, \mathbb{E}\mathbb{I}\{Y \neq f(X)\} = \underset{f}{\operatorname{arg\,min}}\, \mathbb{E}[\mathbb{1}\{Yf(x) < 0\}].$$

0/1 loss

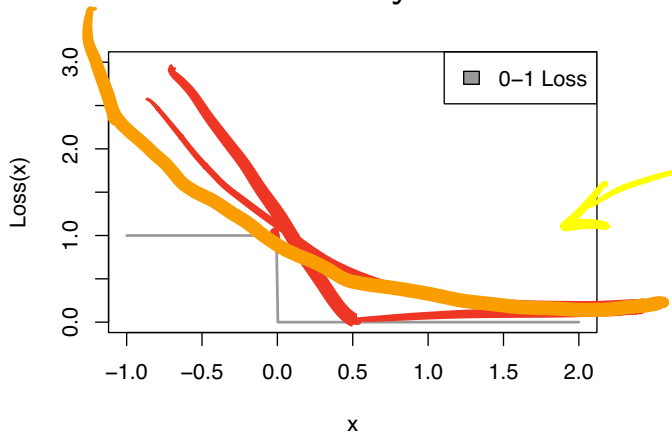
We've constrained ourselves to make $f(X)$ be an additive model built out of simple trees, so the sample version of this problem becomes

$$\widehat{f}(x) = \underset{\alpha_b, f^{(b)}}{\operatorname{argmin}} \frac{1}{n} \sum_{i=1}^{n} \mathbb{I}\left\{ y_i \left(\sum_{b=1}^{B} \alpha_b f^{(b)}(x_i) \right) < 0 \right\}$$

Now it just looks like an optimization problem! But how do we optimize it?

Approximating 0-1 loss

It turns out that 0-1 loss is not a very nice function to optimize



exponential loss.
$\exp\{-yf(x)\}$.

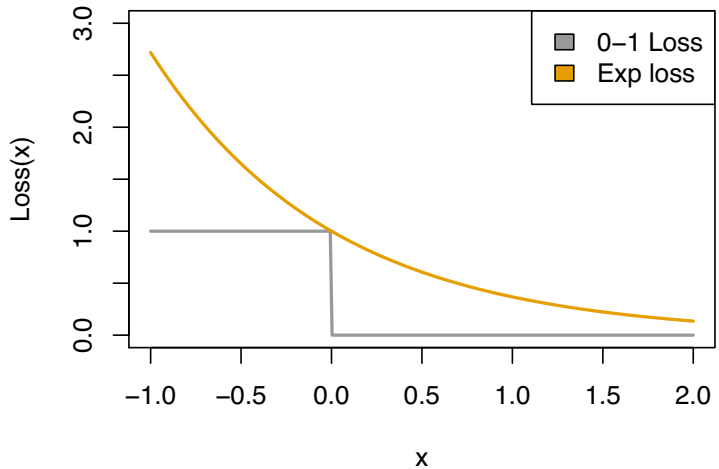
It's not differentiable or continuous. It's also constant except at 0!

Without being able to take a derivative, it's hard to tell what local changes to your function will improve the fit.

With regions of constant loss, there's not a sense of "close to correct," so it's hard to know which points are close to being correctly classified.

Approximating 0-1 loss

We switch 0-1 loss for another function that's easier to optimize and has similar behavior in important ways



This new function is continuous and differentiable and convex, and it is still monotonically decreasing.

If we can make the exponential function small, our 0-1 loss will also be good.

With the new loss

Making that switch, instead of solving

$$\underset{\{\alpha_b, f^{(b)}\}}{\operatorname{argmin}} \sum_{i=1}^{n} \mathbb{I}\left\{ y_i \left(\sum_{b=1}^{B} \alpha_b f^{(b)}(x_i) \right) < 0 \right\}$$

we just have to solve

$$\underset{\{\alpha_b, f^{(b)}\}}{\operatorname{argmin}} \sum_{i=1}^{n} \exp\left(-y_i \left(\sum_{b=1}^{B} \alpha_b f^{(b)}(x_i) \right) \right)$$

With the new loss

We just have to solve

$$\underset{\{\alpha_b, f^{(b)}\}}{\operatorname{argmin}} \sum_{i=1}^{n} \exp\left(-y_i \left(\sum_{b=1}^{B} \alpha_b f^{(b)}(x_i)\right)\right)$$

This is still tricky, so we cheat just a little more. We optimize one $\widehat{f}^{(b)}$ at a time while holding the previous functions fixed, and never come back.

one new tree.

$$\underset{\alpha_b, \widehat{f^b}}{\operatorname{argmin}} \sum_{i=1}^{n} \exp\left(-y_i \cdot \left(\sum_{k=1}^{b-1} \alpha_k \widehat{f}^k(x) + \alpha_b \widehat{f}^b(x)\right)\right)$$

This is a *greedy* algorithm. It seeks immediate rewards, rather than optimizing the overall objective.

- Oddly enough, optimizing this objective leads to the entire AdaBoost algorithm with all the weights and transformations!
- This is a bit difficult to see (but see Page 344 of ESL).
- The main takeaway though is important: boosting is an algorithm to minimize the *exponential loss* using an additive model that it fit incrementally.
- So important that I will repeat it again...

Adaboost summary

Adaboost is just

1. Empirical risk minimization with an exponential loss

$$\frac{1}{n}\sum_{i=1}^{n} e^{-y_i f(x_i)}$$

2. Assuming an additive model of trees:

$$\widehat{f}(x) = \sum_{b=1}^{B} \alpha_b \widehat{f}^{(b)}(x)$$

   with $\widehat{f}^{(b)}(x)$ constrained to be a tree.

3. And greedy, stepwise optimization

$$\underset{\alpha_b, \widehat{f}^b}{\operatorname{argmin}} \sum_{i=1}^{n} \operatorname{Exp}\left(-y_i \cdot \left(\sum_{k=1}^{b-1} \alpha_k \widehat{f}^k(x) + \alpha_b \widehat{f}^b(x)\right)\right)$$

Adaboost give rise to weights of

Empirical risk minimization

This general pattern:

1. We want to minimize:

$$\mathbb{E}\mathbb{I}\{Y \neq f(X)\}$$

2. And so we actually try to minimize its empirical version:

$$\frac{1}{n}\sum_{i=1}^{n}\mathbb{I}\{y_i \neq f(x_i)\}$$

3. But we can't even do that for classification. So we introduce a nicer loss L and minimize

$$\frac{1}{n}\sum_{i=1}^{n}L(y_i, f(x_i))$$

Empirical risk minimization

Different choices of L lead to different methods and different trade-offs.

Exponential loss:

$$L(y_i, f(x_i)) = e^{-y_i f(x_i)}$$

Logistic loss:

$$L(y_i, f(x_i)) = \log\left(1 + e^{-y_i f(x_i)}\right)$$

Hinge loss (SVM):

$$L(y_i, f(x_i)) = \max\{0, 1 - y_i f(x_i)\}$$

Improvements

(we skipped this)

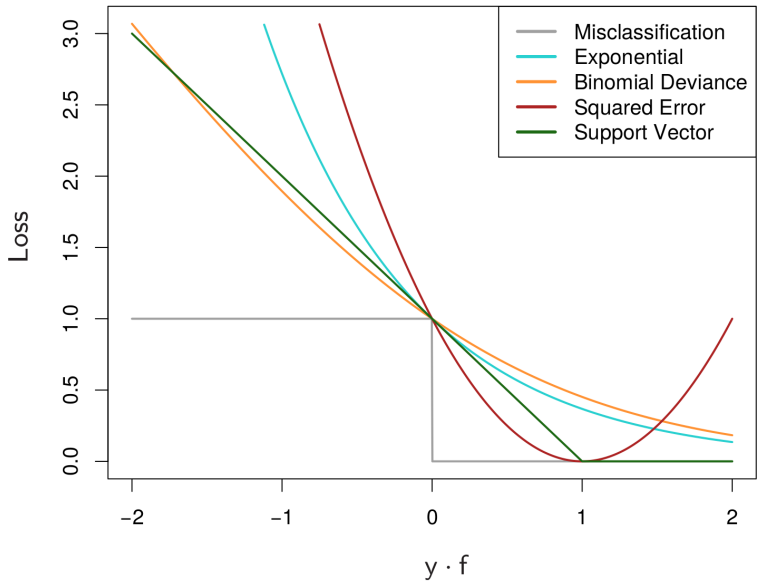
This intuition automatically leads to many improvements

- *Shrinkage:* Only add a small amount of each tree at a time. Like taking smaller steps. Provides regularization. Use $\nu\alpha_b\widehat{f}^b$ with $\nu \in [0, 1]$.

- *Subsampling:* Each step only sees a fraction, η of the data. Gives improvements to variance (and thus performance), as well as speed!

- *Other losses:* Why exponential loss?

Improvements: Why exponential loss?



(ESL Figure 10.4)

There are many losses that approximate 0-1 loss. Some can give dramatically better performance in certain settings.

Improvements: Why exponential loss?

It turns out that the exponential loss leads to a particularly simple algorithm, while the others do not. The optimization becomes harder, and the updates lose their beautiful form.

Instead, at iteration b, the gradient $g_b \in \mathbb{R}^n$ of the loss function is found. The new classifier $\widehat{f}^{(b)}$ is then fit to approximate this gradient vector well. Hence the name: *Gradient Boosting*.

We won't go into this too much but it's worth knowing the term gradient boosting. It is a very popular algorithm, at least in part because it has a great package implementing it (`xgboost`).

Gradient Boosting for Squared Loss

- Compute the gradient of the loss with respect to the function (doing this properly is a bit involved):

$$\nabla (y - f(x))^2 = -(y - f(x))$$

negative residual.

- In the boosting case:
  - compute negative residual $-(y - f(x))$.
  - fit a tree to residual.

- So gradient boosting is just incrementally fitting the residual vector. This algorithm is sometimes called *forward stagewise regression*.

Boosting discussion

In boosting, we tend to use very simple estimators, like very shallow trees. These have low variance, but are high bias. Because the sequence of trees can adjust for previous errors, they can fix the bias!

The shallower trees can also lead to computational speedups in evaluation, since you don't have to evaluate 500 deep trees.

Think of the depth of your trees as allowing interactions. Two splits let you interact two variables. You'll often find that you want somewhere between 2 and 10 splits in your trees.

Boosting discussion

Boosting is one of the most powerful classical methods. When well-tuned, it can beat most other classifiers.

However, it requires more tuning than random forests.

Tuning parameters:

- Number of trees, B
- Amount of subsampling, η
- Amount of shrinkage, ν
- Number of splits in each tree

You will find that random forest are difficult to badly overfit. In boosting, choosing B too large can in some cases cause overfitting (though miraculously not always). Choosing it too small can give a bad classifier.

B is the main tuning parameter. The others can be tuned more roughly, since it doesn't matter quite as much.

Ensemble Learning

One last comment on combining classifiers.

In boosting, we saw an approache to combining many weaker predictors into a much stronger predictor

This combination is called an *ensemble*. There are many general approaches to building ensembles (most of them were inspired by boosting).

We will discuss random forests, bagging and possibly stacking next class.

Ensembles can combine methods of different types with different strengths, so that each can perform well in cases where it is strong.

Many contests are won in this way. In practice, there are sometimes practical concerns about complexity and speed.