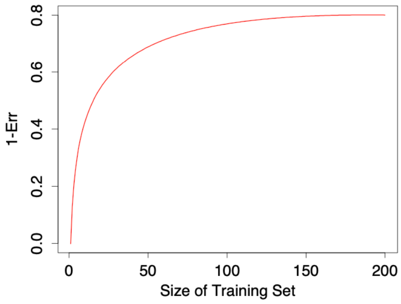

FIGURE 7.8. *Hypothetical learning curve for a classifier on a given task: a plot of* $1 - \text{Err}$ *versus the size of the training set* N*. With a dataset of* 200 *observations,* 5*-fold cross-validation would use training sets of size* 160*, which would behave much like the full set. However, with a dataset of* 50 *observations fivefold cross-validation would use training sets of size* 40*, and this would result in a considerable overestimate of prediction error.*

Figure 9: Figure 7.8 from ESL!

15 Thursday, April 10: Bagging and Random Forests

15.1 Model selection recap

On Monday, we talked about how to select a model if the main thing we care about is predictive accuracy. Roughly speaking, if your main goal is predictive accuracy, then you should be selecting the model with the lowest error on a test set.

- You could use a single train/test split. This let's you compare specific models fit to your specific training set.
- You could use something like 5-fold cross-validation. This makes more efficient use of the data, but you are no longer looking at the error of a specific model fit to the specific training set. You are looking at an average error that results from a model-fitting procedure applied to 80% of your training set.

There is a lot more that I could say about model selection or about cross-validation. But, at a basic level, you all know how to do this already. You all did it on your take-home midterm, for example. The one thing I want to emphasize that I mentioned briefly last time is the idea of the **winner's curse.**

- Suppose that, in reality, there are three models that are equally good in terms of their expected prediction error over all possible realizations of new test sets.
- On the single test set that we observed, one of these models will "win" due to random chance (i.e. it will have the lowest error on our observed test set).
- This test error is biased downwards for the expected test error of this model on a NEW test set. Selecting a winner caused us to overfit to our test set.
- For an unbiased estimate of a selected model's performance on new, unseen data, we actually need three datasets: train / validation / test. We can select the model that achieves minimum error on the validation set, and only at the end do we evaluate the accuracy on the totally unseen test set.

The idea of the winner's curse is really important! Selected models tend to let us down in the future :(. Selected policies in government, econ, etc. tend to underperform in the future: they "won" the first stage of selection partially due to random chance, and now they regress to their mean. This is important to be aware of, any time you are doing model selection!

That's all about model selection for now! If you love model selection, consider a related topic for your final project!

15.2 Intro to Ensemble Methods

Today, let's talk about a totally new and potentially crazy idea: if we only care about predictive accuracy, why do we need to select one model? Why can't we take a bunch of different models that all do pretty well, and average or combine their predictions somehow?

- If the different models tend to make mistakes on slightly different types of datapoints, or if their mistakes are uncorrelated with one another, this could really help us!
- We need to have room to improve; if we are already achieving essentially irreducible error, then this will not

help.

In other words: forget model selection. Let's use an *ensemble model*, that combines the results of several models!

You might be thinking that you are going to fit a KNN, a regression tree, and a lasso regression all to the same dataset, and then average the predictions from all of these different models. We could certainly do that! But, for now, let's not go too crazy with ensemble methods. Let's talk about two relatively simple methods for taking a single model fitting procedure and improving it by fitting it many times in a row.

- The general idea of *bagging* is to take one model that has high variance, and reduce the variance by averaging over several repeated copies of this model. We will talk about this today!
- The general idea of *boosting* is to take one model that has high bias, and reduce the bias by iteratively focusing attention on the examples that we are currently messing up on. We will talk about this next time!

15.3 Bagging (Breiman, 1996)

Suppose we have a single base learner, such as a single deep decision tree, that we could train to our entire dataset. Call this model $\hat{f}()$. There is a fair amount that we like about a deep decision tree (it can uncover interaction terms, it has low bias). However, there is a thing that we do not like about this decision tree. Due to the high variance of decision trees, we know that if we had observed a very slightly different training set, we would have a totally different tree. This high variance compromises our predictive accuracy!

Well, here is a really simple idea that works well if we care about predictions. We can take fit B different deep decision trees to B different bootstrap samples of our dataset. Remember that, to get a bootstrap sample, we draw n observations with replacement from our n original training observations. We now have B different prediction models, $\hat{f}_b()$ for $b = 1, \ldots, B$: each fit to n observations. We let our final, bagged model (bagging stands for bootstrap aggregation), be:

$$\hat{f}_{bag}(x) = \frac{1}{B} \sum_{b=1}^{B} \hat{f}_b(x).$$

And that's it! That is the entire idea of bagging!

15.4 Why does bagging reduce variance?

We know in statistics that averaging is a nice way of reducing variance. But also, bagging shouldn't be some magical way to make our model really good: taking B bootstrap samples from our data doesn't actually increase the total amount of information in our data! So, what is going on and why does bagging work?

Let $Var(\hat{f}_{bag}(x))$ be the variance of a prediction for a single fixed test-point x. The randomness is taken over the training of $\hat{f}_{bag}()$ for many different training sets. Let $Var(\hat{f}(x)) = \sigma^2$. What is $Var(\hat{f}_{bag}(x))$?

Well, based on the form of $\hat{f}_{bag}(x)$, we have that:

$$Var_{bag}\left(\hat{f}(x)\right) = Var\left(\frac{1}{B}\sum_{b=1}^{B}\hat{f}_b(x)\right) = \frac{1}{B^2}\left(\sum_{b=1}^{B} Var(\hat{f}_b(x)) + \sum_{b=1}^{B}\sum_{b'!=b} Cov(\hat{f}_b(x), \hat{f}_{b'}(x))\right).$$

Note that, $\hat{f}_b(x)$ and $\hat{f}_{b'}(x)$ are identically distributed random variables: they both result from applying the same procedure to a random subsample of the same data. Let $Var(\hat{f}_b(x)) = \sigma_b^2$ for any $b = 1, \ldots, B$. Also, note that when n is large, σ_b^2 should be the same as σ^2: fitting a model to the whole training set should be the same as fitting a model to a bootstrap sample from the whole dataset. Next, note that $\hat{f}_b(x)$ and $\hat{f}_{b'}(x)$ are not independent, because they are trained on overlapping subsamples of data. Let $Cov(\hat{f}_b(x), \hat{f}_{b'}(x)) = \rho\sigma_b^2$, where ρ is the correlation induced by the overlapping data. Note that this value is the same for all b, b' from 1 to B.

So,

$$Var\left(\hat{f}_{bag}(x)\right) = \frac{1}{B^2}\left(B\sigma_b^2 + B \times (B-1)\rho\sigma_b^2\right) = \frac{1}{B^2}\left(B\sigma_b^2(1-\rho) + B^2\rho\sigma_b^2\right) = \frac{\sigma_b^2(1-\rho)}{B} + \rho\sigma_b^2.$$

Suppose that we are working with a very stable model fitting procedure. In this case, there probably isn't much change from bootstrap sample to bootstrap sample. This means that there is a lot of correlation between $\hat{f}_b(x)$ and $\hat{f}_{b'}(x)$; $\rho \approx 1$. In this case:

$$Var\left(\hat{f}_{bag}(x)\right) \approx \frac{\sigma_b^2(1-\rho)}{B} + \rho\sigma_b^2 = \sigma_b^2 \approx \sigma^2.$$

Uh oh! This means that bagging did not help us reduce our variance! We did a lot of computational work to get the same variance that we would have had if we had fit just one model to our full dataset. The issue here is the stability– when we get basically the same model from every bootstrap sample ($\rho \approx 1$), bagging really does not help us!

On the other hand, suppose we have an unstable model, and ρ is positive but less than 1[8]. In this case, especially if B is large, the variance of our bagged model is smaller than the variance of an individual bootstrap-sample model: $\hat{f}_b(x)$, which is similar in large samples to the variance of our single model applied to the whole dataset: $\hat{f}(x)$. The smaller that ρ is, the better!

The fact that bagging reduces variance more for unstable models led Breiman, in his original bagging paper, to claim that bagging does not help with linear regression or ridge regression (stable) but helps with stepwise regression or trees (greedy and unstable). This is good to know!

15.5 Does bagging reduce bias?

The average of B linear models is still a linear model. Thus, bagging a bunch of linear regressions does not do anything to the expressiveness of our model, and does not impact the bias. On the other hand, consider bagging shallow decision trees (1 split). Each individual tree is so limited in its complexity; but when we combine many together we get a much more complex additive model. One one-level decision tree gets to use only one variable, whereas a bagged ensemble can take into account many variables. In general, the average of B trees cannot necessarily be written as a tree: so we have increased our model space, and therefore may have reduced bias.

15.6 Random forests (Breiman, 2001)

Before 2001, bagged regression trees were already very popular. Trees are a popular learning algorithm; people like that they can uncover interactions without needing to pre-specify them. However, trees are also notoriously unstable. This makes them a prime candidate to benefit from bagging.

However, in 2001, Breiman combined some existing ideas to make an algorithm that (often) works even better than a bagged ensemble of trees. The idea is simple: we can make the variance reduction of bagging even better by further de-correlating the trees in our ensemble of bagged trees? Or, in other words, by making the individual trees even less stable?

In random forests, we do exactly this. We add additional randomness to every individual tree in an ensemble of bagged trees.

- Recall that, to build a CART regression tree, we greedily select the variable and split point that most improves the goodness of fit of our tree right now. We repeat this process recursively; we keep making new splits by picking the best possible variable.
- For each individual CART tree in a random forest, we make a small modification. At each split, we only allow ourselves to consider a random selection of $m \leq p$ variables as the possible split variables. Thus, we may not choose the overall best split variable right now; we might need to choose something else if the "best" variable is not randomly selected.
- This makes the individual trees in the forest more different from one another (smaller ρ). While it may seem from the derivation above that this *definitely* helps reduce the variance of the bagged ensemble, we need to remember that this also makes $Var(\hat{f}_b(x)) > Var(\hat{f}(x))$: an individual tree in the forest has extra noise added compared to a single tree fit to all of the data with all of the variables. However, the gain that comes from the de-correlation tends to help.
- This can also help us with bias. Recall that, due to the greedy nature of a tree, a tree could miss out on potentially important variables or interactions if the greedy process leads it to go down a certain initial path (dominated by one important variable) while missing other paths. Randomization helps the tree explore all paths and find all possible signal or interactions. This can be really helpful- it seems to work really well on real data.
- Of course, selecting a small value of m could also be bad for bias. If there are only a few truly important predictors and we select a small value of m, for many trees in our forest these important predictors will be left out. This is too bad– it will cause us to miss out on true signal!
- In the case of many correlated predictors, a random forests helps us explore different choices of correlated predictors in different trees. This can be helpful.

[8] Negative ρ would be quite strange, given that the bootstrap samples overlap significantly. I don't think that this would happen.

15.7 More details about random forests

15.7.1 Tuning?

In a random forest, we need to choose:

- How many trees (B) should go in the forest?
- How deep should each tree in the forest be?
- How many variables m should be considered for each split?

Conventional wisdom says that it is okay to build deep trees (high variance!) inside of the forest, because we do not care about interpretability and because bagging will help us reduce the variance. Conventional wisdom also says that you might as well make B big if your computational power allows you to. Increasing B "too much" wastes computation, but cannot cause us to overfit, because it does not cause the model to become more complex, it just causes us to converge to an average. We can keep adding trees to the forest until our OOB error (see below) stops improving.

The hardest variable to tune is m. While some "rule of thumb" out there says to select $m = \sqrt{p}$, there are definitely situations out there for which $m = p$ (no extra randomization) is optimal. The RandomForest package in R has a built-in mechanism for tuning m: it seems really important to check this for your dataset.

15.7.2 OOB error evaluation

To tune a random forest, we don't actually need to do cross-validation. This is because there is already some sample-splitting going on during tree-building!

Each tree in the forest is build to a bootstrap sample of the data. This bootstrap sample tends to have around 2/3 of the observations "in the bag" and around 1/3 of the observations "out of the bag". We can compute the MSE for this tree for these "out of the bag" data points! More generally, for any data-point, we can get a current "test error" by getting its prediction from the forest using only the trees for which it was out of the bag. This is nice- this MSE is not affected by overfitting.

We can keep adding trees to our forest until the OOB error stops improving. This will be when we have more trees than necessary: since the OOB MSE doesn't get to use as many trees per datapoint to get predictions as we would use for new data. But its still a useful metric that can tell us when we clearly don't need more trees.

15.7.3 Permutation importance

While random forests provide a clear way to improve on the predictive accuracy of a single tree, they also lose one of the main benefits of trees: interpretability. However, as we discussed, should we really call an unstable model (a single tree) interpretable? If you would like to use a random forest but would also like to know about which variables are important to your model, people have developed a variety of simple *explainability* techniques that try to get at this. One of them is permutation importance, which is built into the random forest R package. We will cover this next Thursday, on interpretability/explainability day!

15.8 A discussion: let's go back to no-free-lunch

Varya was working on an RDemo for you all that was supposed to show that a random forest outperformed bagging! However, with either simulated data or fairly simple real datasets, she was having trouble showing this result! Often, bagging (use all features) was still beating random forests! After looking through her code and trying to debug it (there were no bugs), I came to some conclusions.

- Just because your textbook says that random forests should outperform bagging doesn't mean they will on simple datasets: there is no free lunch! No algorithm outperforms all other algorithms on all datasets!
- For the randomization idea to be useful, there needs to be "room for improvement". For example, if your data has tricky-to-find effects with lots of important variables, then the extra randomization might really diversify the trees in your forest and really help. Real datasets tend to be super complex, which is likely why people claim random forests work so well in the real world.
- When your data is not super complex, you probably want $m = p$. Setting $m < p$ makes the bias and variance worse for every individual tree: so unless you are really gaining something from this when you average, you probably don't want to do it!
- In practice, since there is no free lunch, it is always important to tune your models! Nothing is REALLY "off the shelf".

- Don't just blindly trust what a textbook says! Everything depends on your data!

16 Monday, April 14: Boosting

Last time, we defined *ensemble methods*. We then talked more specifically about bagging, which is a general-purpose technique that is designed to take a single model with low bias but high variance (e.g. deep decision tree) and reduce the variance through averaging. Today, we will talk about boosting, which is a general-purpose technique that is designed to take a single "weak learner" with low variance but high bias and adaptively improve it until it becomes a "strong learner".

A cool thing about boosting is that there were really key innovations from both computer scientists and statisticians at around the same time, and the innovations got combined to make something really powerful! This perhaps shows that the "two cultures" discussed by Breiman aren't really at odds with one another: they compliment one another!

16.1 AdaBoost (Freund and Schapire, 1996

AdaBoost is an important early boosting algorithm! It is not covered in ISL, but is covered in depth in ESL!

16.1.1 Algorithm

The ideas for boosting were first born in theoretical computer science, and started with classification. We have n datapoints, and our responses y are either -1 or 1: these represent two classes.

The idea was to wonder: is it possible to take a weak learner or a base learner (any algorithm that performs slightly better than random guessing) and "boost" it until it becomes an arbitrarily accurate "strong" learner? It turns out that it is possible! The idea involves fitting the weak learner T times during rounds $t = 1, \dots, T$. At each round, we re-weight the data points to focus on the ones that we misclassified previously. The pseudo-code is as follows.

- Start with training set $(x_1, y_1), \dots, (x_n, y_n)$, where all weights are $w_1(i) = 1/n$ for $i = 1, \dots, n$.
- For $t = 1, \dots, T$:
    1. Take a random sample of size n, with replacement, from the training dataset, where datapoint is probability of being selected is its current weight w_i.[9] If our weak learner allows the use of "weights", we can also skip this step and go right to step 2, but run the algorithm on the weighted data.
    2. Train the weak learner on the subsampled or weighted dataset. This yields $\hat{f}_t(x)$. Compute its training error, e_t, on the weighted or subsampled dataset.
    3. Let $\alpha_t = \frac{1}{2} \log \left(\frac{1-e_t}{e_t} \right)$.
    4. Update the weights! Let:
    $$\tilde{w}_{t+1}(i) = w_t(i) \exp \left(-\alpha_t y_i \hat{f}_t(x_i) \right)$$
    and then let $w_{t+1}(1), \dots, w_{t+1}(n)$ be the $\tilde{w}_{t+1}(i)$ values but normalized to sum to 1 so that we have a probability distribution over our training examples.
- The final learner is:
$$\hat{f}_{boost}(x) = \text{sign} \left(\sum_{t=1}^{T} \alpha_t \hat{f}_t(x) \right).$$

That is it! That is AdaBoost!

16.1.2 Intuition

We need to talk about exactly what is going on in each step. Even though boosting is now much more general than AdaBoost, AdaBoost is such an important early example that I think it is worth focusing on some intuition for AdaBoost before we go further.

- First let's discuss α_t. This is the weight that each weak learner $\hat{f}_t(x)$ will end up getting towards the final prediction. This is based on the training error e_t of each weak learner. The only thing we assume about e_t is that our weak learner can do better than random guessing on our training set: thus, e_t is between 0 and 0.5 for a binary classification task. On this domain, the weight α_t is large and positive if the training error is near 0,

[9] In round 1, this is just a bootstrap sample. But this will change later!

and is near 0 if the training error is near 0.5. So ... the classifiers that do well on their own training sets get more say in the final prediction! This seems reasonable!

- Now let's talk about the weight updates w_t. Recall that we are assuming we wrote down and stored y as -1 and 1. And we are assuming that the weak learner $\hat{f}(x)$ outputs either -1 or 1 for each datapoint. So, the term $y_i \hat{f}_t(x_i)$ in the weight update term will be positive whenever we correctly classified this point, and will be negative otherwise.
  - So, correctly classified points get $\tilde{w}_{t+1} = w_t(i)\exp(-\alpha_t)$.
    - If this weak learner was doing a good job overall, then α_t is big and $\exp(-\alpha_t)$ is near 0. So, in this case, correctly classified points get small weights in the next round! We don't need to focus on them again!
    - On the other hand, if this weak learner was doing a bad job overall, then α_t is near 0 and $\tilde{w}_{t+1} \approx w_t$: we will essentially try again with current weights to see if we can improve.
  - Incorrectly classified points get re-weighted as $\tilde{w}_{t+1} = w_t(i)\exp(\alpha_t)$.
    - If this weak learner was doing a good job overall, then α_t is big and we will increase the weight of this point a LOT. The idea is like "wow, even for an overall really good learner, we missed this point. Now we need to go focus on it."
    - If this weak learner was doing a bad job overall, then α_t is near 0 and $\tilde{w}_{t+1} \approx w_t$. So we never make large updates based on a weak learner that isn't even doing well!
- At the end, each weak learner gets to contribute a $+1$ or -1 vote to the classification. Its vote matters more if it had a big α_t.

16.1.3 Practical considerations, or cool properties?

- By design, AdaBoost definitely continually reduces training error. If each $\hat{f}_t(x)$ is a bit better than random guessing on the training set, the more of these that we combine the better that we do. Our training error should actually go to 0 if T is big enough. This property holds for binary classification where we start with anything better than random guessing: but might not be practically super interesting because this says nothing about test error.
- People were at first worried that AdaBoost would certainly overfit if T is too big. Remember that on midterm 3 you derived a formula that said that the difference between the training error and the expected test error depends on the degrees of freedom of an algorithm. Working with a similar concept, people showed that an upper bound on the difference between the training error and the expected test error for AdaBoost grows with T. However, in practice, it seems like we don't hit this upper bound. In practice, AdaBoost can work pretty well, even when T is so big that the training error is 0.
- What is going on?!?!?! When we have a training error of 0, aren't we overfitting? Why would we keep increasing T past this point?
- Even after AdaBoost has reached a point of 0 training error, we can keep iterating and keep re-fitting with new weights. The consequence of this tends to be that we increase our margin of confidence for our classification: much like an SVM! We know from SVMs that increasing the margin can be good for generalization error; we get more confident in our predictions. So ... that's cool! Whether or not extra iterations will increase the margin or overfit to noise depends on the signal to noise ratio in your data.
- Freund and Schapire certainly note that, while AdaBoost can work really well off-the-shelf with some nice properties, it is clearly dependent on the data and the weak learner. It can fail, and is particularly susceptible to noise.

A note on statistics vs. computer science culture. As you know from Breiman, statistics literature often starts by assuming a data-generating mechanism for the data. On the other hand, a lot of computer science machine learning literature starts from a "no-noise" setting, where we are really doing like pattern-recognition, not statistics. Boosting (like SVMs) was first studied by people in the no-noise setting! In a no-noise setting, over-fitting is less of an issue. This difference colored some early comparisons of bagging vs. boosting. In the Dietterich paper that I posted on GLOW, you can see that he finds that bagged trees outperform boosted trees in high-noise settings.

16.1.4 Comparison to Bagging

Now that we understand a bit more, we can compare AdaBoost to bagging!

- The idea of boosting is to start with a weak learner and improve it sequentially. The focus is on reducing bias by focusing on hard examples.

- The idea of bagging is to start with a learner that has high variance and reduce the variance through averaging.
- Both lose interpretability compared to a single model, but can have much higher accuracy.
- Both are supposedly general purpose and off the shelf, but you might still need to do some tweaking or tuning!
- Bagging can be trained in parallel; boosting cannot be because it is adaptive. For huge data applications, we might care about this!
- Bagging might work better in high-noise situations. And in statistics, we always think there is noise!
- Both have some beautiful theory attached that is beyond the scope of this class!

16.2 Boosting is an additive model? (Friedman, Hastie, and Tibshirani, 2000)

Boosting is far more general than just AdaBoost! It can be extended to multi-class classification and regression. In the process of extending it, people realized it has beautiful connections to other concepts in statistics!

We don't have time to do justice to all of these connections! We will really briefly talk about the ideas from one paper: Friedman, Hastie, and Tibshirani (2000). They connect boosting to additive logistic regression. These insights helped with computational efficiency of boosting, let us study its theoretical properties with a wider array of tools, and also led to further extensions! This paper is summarized really nicely in ESL Chapter 10! The notes here are not intended to provide a lot of detail: just to provide connections to things you have seen already in class!

The key insight of this paper is that AdaBoost can be seen as an iterative forward stage-wise algorithm for fitting an additive logistic regression model, that optimizes a particular loss function. This insights helps them see that AdaBoost can be both simplified but also greatly generalized!

16.2.1 Remember GAMs?

Recall from many weeks ago that a GAM (generalized additive model) has the form:

$$\hat{f}(x) = \sum_{j=1}^{p} \hat{f}_j(x_j).$$

This could be a linear regression model where we are directly modeling y, or could be a logistic regression model where $\hat{f}(x)$ is modeling the log odds of belonging to a certain class. This model does not allow predictors x_j and x_k to interact, unless we pre-specify the interaction as its own feature, which was a limitation.

Recall that we had an iterative procedure (backfitting) for fitting a GAM that allowed each individual $\hat{f}_j(x_j)$ to be something super complicated like a smoothing spline. The backfitting algorithm starts with a guess for each $\hat{f}_j()$. Then, for one j at a time, it fits a model to predict the current residuals $y - \sum_{k \neq j} \hat{f}_k(x_j)$ using x_j.[10]. It uses this model as its new, updated guess for $\hat{f}_j()$. We do this iteratively until the guesses for the $\hat{f}()$ stop changing.

Why are we mentioning GAMs? The idea of iteratively fitting relatively simple models to current residuals to improve the model sounds like boosting!!! There are just two differences.

- This model goes predictor-by-predictor; boosting did not have this limitation.
- This model goes back and updates the guess for $\hat{f}_k(x_j)$ multiple times; boosting does not change a simple model once it fits it once; it just adds more simple models.

16.2.2 GAMs with new basis functions

What happens if we make our additive model a little bit les additive?

Let $b(x, \gamma)$ be a basis function, which depends on parameters γ, that depends on all of the predictors x. We could let:

$$\hat{f}(x) = \sum_{m=1}^{M} \beta_m b(x, \gamma_m). \tag{18}$$

This model is additive in the basis functions, but in terms of the individual features this is a linear combination of non-linear functions of the individual features.

We have seen a model just like this before! A neural network with one hidden layer can be written in this way![11] The γ_m are the weights in the first layer that connect each x to hidden node m. Then, b is the activation function, and β_m

[10] If we are doing logistic regression, we don't quite use residuals. We use working residuals, which are more complicated

[11] We also talked about projection pursuit regression: an extension of GAMs to more complicated models, that also looks just like this!

is the weight in the second later that connects hidden node m to the output. So, if b is a non-linear transformation of a linear combination of the xs, then this is a neural network, and we could fit it with backpropagation.

But what if b is a simple regression tree, and γ_m encodes the split variables and split points of this tree? All of the sudden, (1) I am not sure how to fit it with backpropagation, but (2) this looks quite a bit like a model we could fit with AdaBoost!

16.2.3 Adaboost as an algorithm for fitting (18)

Suppose that we fit (18) in a greedy forward stagewise manner. The idea of a forward stagewise procedure is that we first figure out β_1 and γ_1, then we figure out β_2 and γ_2, etc. Unlike with backfitting, we do not let ourselves go backwards and update β_1 again later in the process. By "fit" the model, I mean pick the values for β_m and γ_m that most improve a certain loss function between $\hat{f}(x)$ and y right now.

It turns out that the weights and update steps for Adaboost are exactly equivalent to what we would get if we fit the model (18) using a greedy forward stagewise procedure with a particular exponential loss function. The reason that this perspective is so cool is that it immediately tells us that we could come up with different boosting procedures by replacing the exponential loss function with another loss function!

For classification, we could consider a loss function that is more robust to outliers. For regression, we can use something like squared error loss, which makes boosting *very* simple (see the algorithm in ISL where they simply fit regression trees to residuals!).

16.2.4 Adaboost as gradient descent?

Forward stage-wise boosting is a very greedy strategy. We want to take the step that most reduces the loss function right now. Recall that gradient descent updates initial guesses for parameters by moving them in the direction that will most reduce the lost function right now. These seem really similar!

In fact, in ESL Section 10.10.2, the authors discuss the fact that, when we add a new tree to a boosting model, we really want to add a tree that moves in the direction of the current gradient of the loss function. Thus, we should fit our new tree to the current gradient of our loss function: this way, we are moving in a direction that is close to the direction of the negative gradient, but we are moving in a way that is allowed in our model class (we are just adding a tree).

For squared error loss, this just means fitting a regression tree to the current residuals! For other loss functions, we similarly fit a tree to a form of current working residuals. Thus, we can really see boosting as fitting (18) by always taking steps in our model space (i.e. steps that add a new tree) in the direction of the gradient of the loss function.

There have been some other innovations that have come from viewing boosting as gradient descent or an additive model. For example, we know that in gradient descent we should take SMALL steps in the direction of the gradient. We shouldn't be too greedy right now, lest we overshoot our optimum or get stuck in a local optimum. This leads to the idea of a shrinkage parameter for boosting; maybe we shouldn't let ourselves take steps that are too big! So maybe we should only take a SMALL step in the direction of each new tree. We might need more trees in the model, but this seems to really help prevent overfitting.

This was a whirlwind: but now you know that gradient boosting exists!