Machine Learning I

Lecture 19: Boosting

Nathaniel Bade

Northeastern University Department of Mathematics

Table of contents

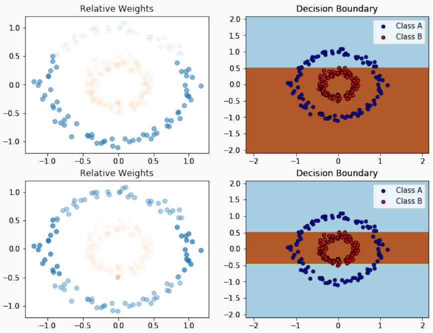
Introduction to Boosting

Introduction to Boosting

Boosting started as a theoretical question: Can a committee of fast learning, slightly-better-than-average algorithms combine to form an algorithm with strong predictive power?

In 1990, Robert Schapire answer the theoretical question in the affirmative. Five years later, he produced with Yoav Freund a practical implementation called Adaboost.

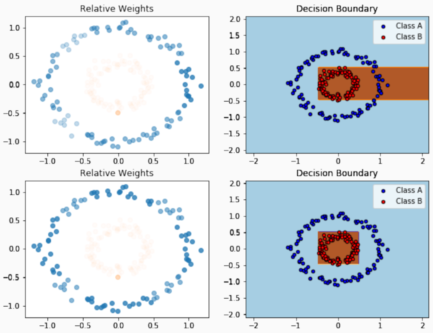
Introduction to Boosting



Consider a two class problem, $\mathcal{Y} = \{-1, 1\}$ with a vector space $\mathcal{X}$ of features. Roughly, a weak classifier G is one whose error rate is slightly better than random guessing.

The key innovation in boosting algorithms is to fit each basis element **sequentually** instead of in paralllel. By sequentially applying weak classifiers to repeated modified version of the data, we can form a weighted sum of prediction classes.

Introduction to Boosting

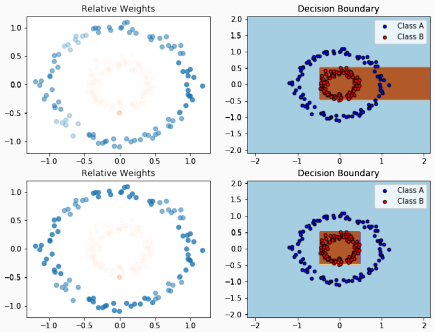


By sequentially applying weak classifiers to repeated modified version of the data, we can form a weighted sum of prediction classes,

$$G(x) = \text{sign}\left(\sum_{m=1}^{M} \alpha_m G_m(x)\right) ,$$

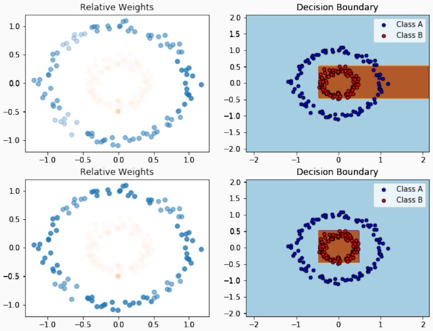
where α_i are computed by the boosting algorithm.

Introduction to Boosting



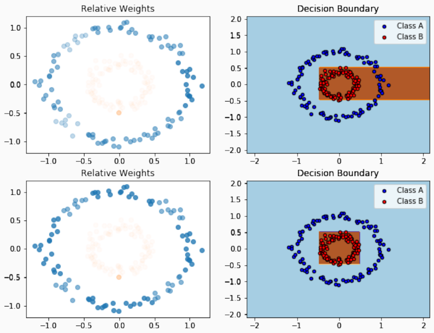
At each step of the algorithm, the data points which are misclassified in step m will be weighted more strongly in step $m+1$. For regression boosting, this means at each step we fit the weighted residuals of the previous step.

Lecture Overview



In this lecture, we will see that by combining high bias, easy to train classifiers, boosting produces strong, computationally efficient classifiers with well controlled bias/variance trade off and VC dimension. Of course the cost is going to be interpretability. Boosting produces solidly "blackbox" classifiers. That said, at the end of the lecture we will discuss some ways we can understand these results.

Lecture Overview



We will focus in this lecture on Adaboost, the first and most popular boosting algorithm. We will show both some theoretical results, as well as discussing practical fitting methodology. Finally, we follow ESLII and UML and look closely at boosted tree classifiers.

Adaboost

Adaboost Algorithm: Setup

Consider a two class problem, $\mathcal{Y} = \{-1, 1\}$ with a vector space $\mathcal{X}$ of features. Let $\mathcal{T} = \{(x_i, y_i)\}$ be training set of size N and let $G_m(x)$, $m = 1, \ldots, M$ be a sequence of weak classifiers.

Our goal is to produce a weighted majority vote classifier

$$G(x) = \operatorname{sign}\left(\sum_{m=1}^{M} \alpha_m G_m(x)\right),$$

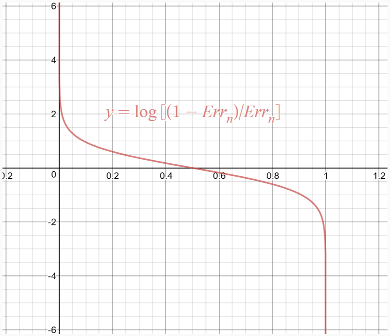
where the weights are chosen algorithmically. We will now take a moment to describe the sequential training for the Adaboost algorithm.

Adaboost Algorithm

Adaboost algorithm:

```
Initialize weights $w_i = 1/N$ .
For $m = 1$ to $M$,
  (1) Fit a classifier $G_m(x)$ to the training data weighted by $w_i$.
  (2) Compute the training error:
        $$\mathrm{Err}_m = \frac{1}{\sum_{i=1}^N w_i} \sum_{i=1}^{N} w_i \, \mathbb{1}(y_i \neq G(x_i)) \, .$$
  (3) Compute $\alpha_m = \log((1 - \mathrm{Err}_m)/\mathrm{Err}_m)$
  (4) Update each weight to $w_i = w_i \cdot \exp\left[\alpha_m \mathbb{1}(y_i \neq G(x_i))\right]$,
      $i = 1, \ldots, N$.
The output is $G(x) = \mathrm{sign}\left(\sum_{m=1}^M \alpha_m G_m(x)\right)$.
```

Adaboost Algorithm: Details

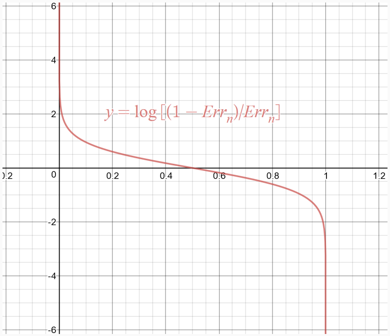


The error in step 2

$$\mathrm{Err}_m = \frac{1}{\sum_{i=1}^N w_i} \sum_{i=1}^N w_i \, \mathbb{1}(y_i \neq G(x_i))\,.$$

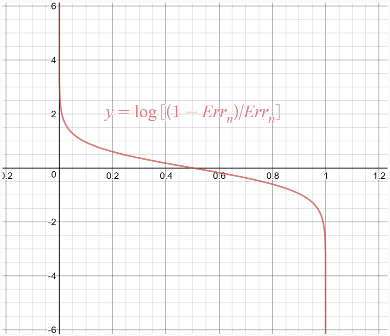
lies between 0 and 1. The weights $\alpha_m = \log((1 - \mathrm{Err}_m)/\mathrm{Err}_m$ then interpolate between ∞ for $\mathrm{Err}_m \to 0$, and $-\infty$ for $\mathrm{Err}_m \to 1$.

Adaboost Algorithm: Details



Practically, this means that $\alpha G_m(x)$ for which the weighted error is small contribute more to final boosted classifier, with classifiers being weighted negatively if their accuracy is less than .5.

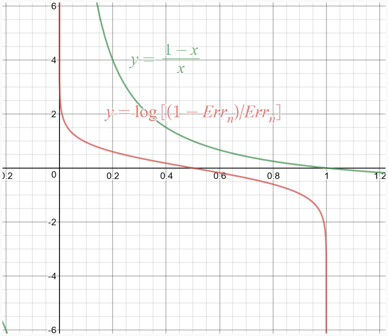
Adaboost Algorithm: Details



Note that in the weight update step, $w_i = w_i \cdot \exp\left[\alpha_m \mathbb{1}(y_i \neq G(x_i))\right]$, $i = 1, \ldots, N$, the weights are only updated if G_m misclassifies x_i, ie $y_i \neq G_m(x_i)$. If so,

$$w_i = w_i \frac{(1 - \text{Err}_m)}{\text{Err}_m} .$$

Adaboost Algorithm: Details

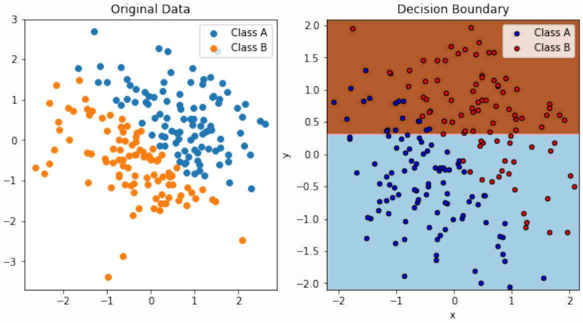


The update

$$w_i = w_i \frac{(1 - \mathrm{Err}_m)}{\mathrm{Err}_m}$$

leaves w_i untouched if the error is too high, but increases the weight as the total error of $G_m(x)$ drops towards 0. This means points not classified byma good classifier are weighted higher.

Adaboost Algorithm: Details

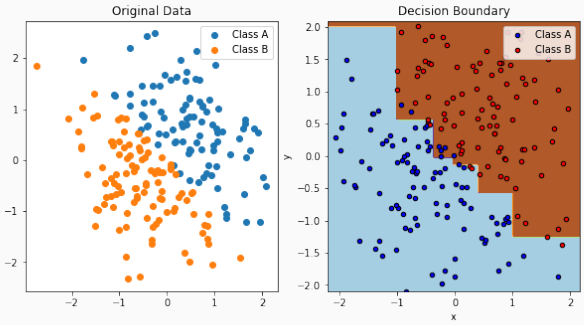


As an example of boosting, lets consider boosting the **decision stump**

$$h_{j,\theta}(X) = \text{sign}(X_j - \theta)\,, \qquad j = 1, \ldots, p\,.$$

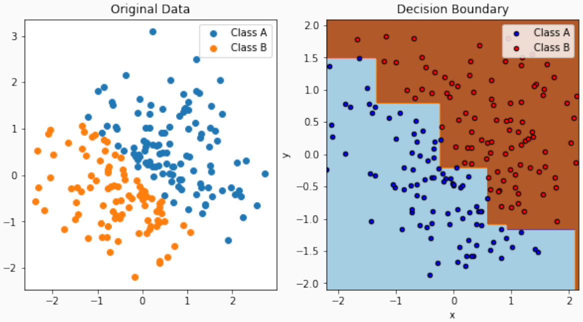
Note, that decision stumps are the node wise decision functions for a tree classifier. In general, a decision stump may do only a little better than .5.

Adaboost Algorithm: Details



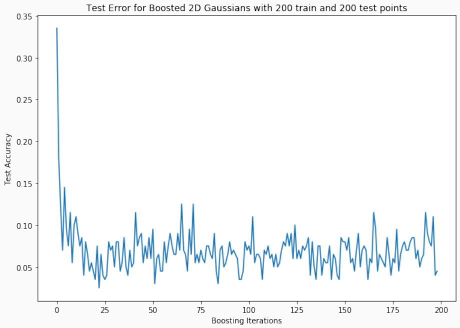
But as M becomes large, the classifier becomes better,

Adaboost Algorithm: Details



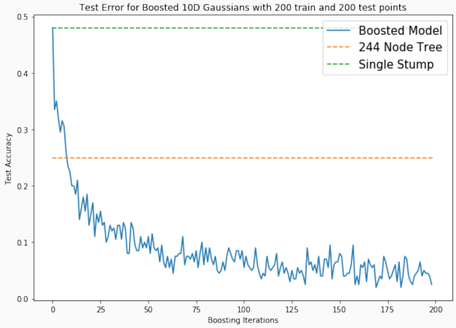
But as M becomes large, the classifier becomes better, and better.

Adaboost Algorithm: Details



But as M becomes large, the classifier becomes better, and better. Charting the test error vs the number of iterations, we see that the test error rapidly drops below 10%. In this case, the data was drown from a Gaussian distribution and labeled by $\text{sign}(x_1 + x_2)$.

Adaboost Algorithm: Details



For a higher dimensional example, we look at a Gaussian distribution in $\mathbb{R}^{10}$ labeled by

$$Y = \text{sign}(X_1 + \ldots + X_{10}) \,.$$

For $N = 2000$ training points, the classifier rapidly converges to below 10% accuracy, easily beating out a much larger tree model. In fact, boosting the simpler classifier reduces the error by almost a factor of 10.

Boosting as and Additive Models

Boosting is in effect an additive expansion of a solution in terms of simple basis functions G_m:

$$f(x) = \sum_{m=1}^{M} h_m^T(x)\alpha_m = \sum_{m=1}^{M} \alpha_m G_m(x)\,.$$

We can compare boosting to other examples of **basis expansions**, like

Smoothing methods and wavelet bases.

Single layer perceptrons, there $h(x) = \sigma(\beta_0 + X^T\beta_1)$.

Random forests.

Trivially, linear classifiers.

Adaboost in the Loss Minimization Framework

Forward Stagewise Additive Modeling

We see how Adaboost works, but what exactly is it minimizing?
Typically, we fit additive models by minimizing a total loss function

$$\textbf{Loss}(\theta) = \sum_{i=1}^{N} L\left(y_i, \sum_{m=1}^{M} h_m^T(x, \beta_m)\alpha_m\right)$$

by adjusting all parameters $\theta = \{\alpha_m, \beta_m\}$ more or less simultaneously.
Does such a loss function exist for Adaboost?

In addition, it's clear Adaboost isn't working directly by gradient decent or Newtons method, so how does it work?

Forward Stagewise Additive Modeling

Forward stagewise modeling (FSM) attempts to minimize **Loss**(θ) by sequentially adding basis functions without adjusting the parameters of the basis elements that have already been added.

For example, for squared error loss, $L(y, \hat{y}) = (y - \hat{y})^2$, FSM is equivalent to sequentially fitting the residuals R_{im} of the previous fit:

$$\begin{aligned} L(y_i, f_{m-1}(x_i) + \alpha_m h_m(x_i)) &= (y_i - f_{m-1}(x_i) - \alpha_m h_m(x_i)) \\ &= (R_{im} - \alpha_m h_m(x_i))^2 \,. \end{aligned}$$

Forward Stagewise Additive Modeling

It turns out Adaboost is equivalent to forward FSM with an exponential loss function

$$L(y, \hat{y}) = \exp(-y\,\hat{y})\,.$$

Given basis functions $G_m(x)$, let f_{m-1} be the total function at the $m-1$'st step of Adaboost. At the m'th step, we need to find G_m and α_m that minimize

$$\begin{aligned}\mathbf{Loss}(\alpha_m, G_m) &= \sum_{i=1}^{N} \exp\Big[-y_i\big(f_{m-1}(x_i) + \alpha_m G_m(x_i)\big)\Big] \\ &= \sum_{i=1}^{N} w_i^{(m)} \exp(-\alpha_m y_i G_m(x_i))\,.\end{aligned}$$

The $w_i^{(m)}$ depend on neither α_m nor G_m and so can be considered weights in the fitting.

Forward Stagewise Additive Modeling

Rewriting

$$\sum_{i=1}^{N} w_i^{(m)} \exp(-\alpha_m y_i G_m(x_i)) = \sum_{y_i = G_m(x_i)} e^{-\alpha_m} w_i^{(m)} + \sum_{y_i \neq G(x_i)} e^{\alpha_m} w_i^{(m)} \,,$$

and fixing $\alpha > 0$, the loss is minimized by

$$G_m(x) = \operatorname*{argmin}_G \sum_{i=1}^{N} w_i^{(m)} \mathbb{1}(y_i \neq G(x_i)) \,.$$

Once G is found, we solve

$$\alpha_m = \operatorname*{argmin}_\beta \sum_{i=1}^{N} w_i^{(m)} \exp(-\alpha y_i G_m(x_i))$$

for

$$\alpha_m = \frac{1}{2} \log \frac{1 - \mathrm{Err}_m}{\mathrm{Err}_m} \,.$$

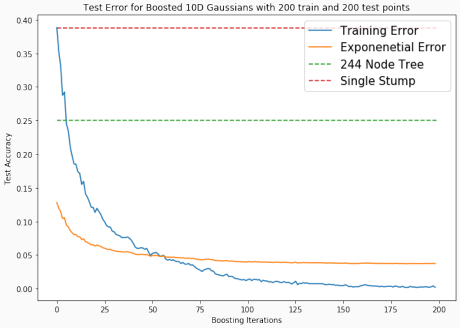
Forward Stagewise Additive Modeling

So the determination of α and G are given by FSM (up to a factor of 2). At the $m+1$'st step, the weights are updated to

$$w_i^{(m+1)} = w_i^{(m)} e^{-\alpha_m y_i G_m(x_i)} = w_i^{(m)} e^{-\alpha_m |y_i - G_m(x_i)| - \alpha_m} .$$

The overall coefficient of $E^{-\alpha_m}$ multiplies all of the weights and so has no effect. This up to an overall factor of 2 in α the weight update equation for Adaboost.

Forward Stagewise Additive Modeling



For example, we see here the training error plotted along with the average exponential error. It's clear that Adaboost is actually trying to minimize the average exponential error.

Boosting and PAC learning

Boosting and PAC Learning

Lets take a moment to discuss boosting in its place of origin: the PAC learning framework. Boosting begun in the world of theoretical machine learning, in an attempt to see whether a basis of weak learners could be ensembled together into a strong learner.

Boosting is of great theoretical interest, but it is also an example of theory guiding practice. In this section, we will take some time to carefully define weak learnability, and prove some results about the theoretical effectiveness of boosting both on empirical and true risk minimization.

PAC Learnability

Recall that a hypothesis class $\mathcal{H}$ is PAC learnable over $\mathcal{X}$ if there exists $m : (0,1)^2 \to \mathbb{N}$ and a learning algorithm A with the following properties:

For every δ, $\epsilon \in (0,1)$ and every distribution $\mathcal{D}$ over $\mathcal{X}$ and for every labeling function $f : \mathcal{X} \to \{\pm 1\}$, if realizability holds then if the number of training samples $|\mathcal{T}| > m(\delta, \epsilon)$,

$$L_{(\mathcal{D},f)}\big[A(\mathcal{T})\big] \leq \epsilon \qquad \text{with probability } 1 - \delta\,.$$

PAC learnability does not guarantee computational simplicity, even in the realizable case. However, maybe we can trade some accuracy for computational hardness.

Weak Learnability

A learning algorithm A is a γ-**weak learner** for a class $\mathcal{H}$ if instead of forcing the true error of $A(\mathcal{T})$ less than ϵ it can instead force it less than $1/2 - \gamma$.

Formally, A is a γ-**weak learner** if there exists $m : (0,1) \to \mathbb{N}$ such that for every $\delta \in (0,1)$ and every distribution $\mathcal{D}$ over $\mathcal{X}$ and for every labeling function $f : \mathcal{X} \to \{\pm 1\}$, if realizability holds then if the number of training samples $|\mathcal{T}| > m(\delta)$,

$$L_{(\mathcal{D},f)} A(\mathcal{T}) \leq \frac{1}{2} - \gamma \qquad \text{with probability } 1 - \delta\,.$$

A hypothesis class is γ-**weak learnable** if such an algorithm exists. We call traditional PAC learnability **strong learnability**.

Weak Learnability

Strong learnability implies that with enough data we can find arbitrary good solutions in the realizable case, while weak learnability only guarantees that we can find a learner that does better than a coin toss.

Note that the by Fundamental Theorem of Learning if $\mathcal{H}$ has VC dimension d, then

$$m(\epsilon, \delta) \geq C_1 \frac{d + \log(1/\delta)}{\epsilon} \, .$$

If $\epsilon = 1/2 - \gamma$, we see that if $d = \infty$ than $\mathcal{H}$ is not γ-weak learnable. Therefore ignoring computational considerations, both types of learnability are characterize by the VC dimension and so weak learning is just as "hard" as strong learning. However, algorithms that satisfy weak learning may be much for computationally achievable.

Adaboost Is An Efficient Learner

(UML Theorem 10.2) Let S be a training set and assume that at each iteration m of Adaboost the weak learner returns a hypothesis for which $\epsilon_m \leq 1/2 - \gamma$. Then, the training error of the output $\hat{G}(x)$ is bounded by

$$L_{\mathcal{T}}(\hat{G}) = \frac{1}{N}\sum_{i=1}^{N} \mathbb{1}(\hat{G}(x_i) \neq y_i) \leq \exp(-2\gamma^2 T)\,.$$

Proof Sketch: The proof rests on two ideas: Writing

$$Z_m = \frac{1}{N}\sum_{i=1}^{N} e^{-y_i f^{(m)}(x_i)}\,,$$

and noticing that $\mathbb{1}(\hat{G}(x_i) \neq y_i) \leq e^{-\hat{G}(x_i) y_i}$, we can bound

$$L_{\mathcal{T}}(\hat{G}) \leq Z_M = \frac{Z_M}{Z_{M-1}} \frac{Z_{M-1}}{Z_{M-2}} \cdots \frac{Z_1}{Z_0}\,.$$

The final step is to show $Z_m/Z_{m-1} \leq e^{-2\gamma^2}$, which is a straightward if tedious computation. $\square$

Decision Stumps

Each iteration of Adaboost involves $O(M)$ operations, as well as a call to the weak learner. Lets take a moment to highlight a hypothesis class that we will consider computationally efficient.

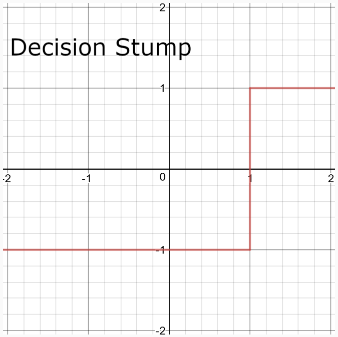
The hypothesis class of decision stumps on $\mathbb{R}^p$ is

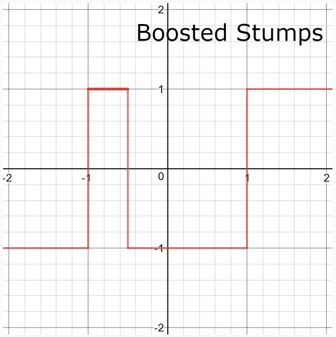
$$\mathcal{H}_{DS} = \left\{ \operatorname{sign}(x - \theta) \cdot b \,:\, \theta \in \mathbb{R},\, i \in [p],\, b \in \pm 1 \right\}.$$

It can be shown that $\mathcal{H}_{DS}$ is a $\gamma = 1/12$-weak learnable in $O(pN)$ time, where N is the number of samples in the training set. Therefore, the boosted hypothesis class can be learned in $O(pNM)$ time.

As a comparison example, the computational complexity of least squares linear regression is $O(p^2 N)$ and decision tree with depth d $O(Npd)$, or between $O(Np \log N)$ and $O(N^2 p)$ if d is not specified.

Weak Learnability





Boosting will clearly increase the VD dimension, but it does so in a controlled way.

Questions: The VC dimension of the 1d decision stump is 2. What is the VC dimension of the M boosted 1d decision stump?

The Boosted VC dimension

The VC dimension of a linear combination of M classifiers is upper bounded in terms of the M and the VC dimension of the base class:

(UML Lemma 10.3) Let B be a basis of functions with fixed VC dimension d_B and let $L(B, M)$ be the sign of a weighted sum of elements of B. Assume that M and d_B are larger than 3. Then

$$\mathrm{VDdim}(L(B, T)) \leq M(d_B + 1)(3\log(M(d_B + 1)) + 2)\,.$$

This tells us more or less exactly how the VC dimension changes as we boost a weak learner. Ignoring the logarithmic terms, the VC dimension of the boosted learner grows like $M(d_B + 1)$. The parameter M gives us granular control over the bias variance tradeoff at both a theoretical and computational complexity level.

Boosting Trees

Boosting Trees

The logical extension of boosting stump functions is booting decision trees. Recall that a tree $T(x; R_j)$ can be though of as a division of $\mathcal{X}$ into distinct axis aligned rectangular regions R_j on which y_i is uniformly estimated, usually by $\bar{y}_j$.

The boosted tree model is a sum

$$f_M(x) = \sum_{m=1}^{M} T(x; R_{mj}) \,,$$

fitted in a forward stagewise manner. At each step, one must solve

$$\hat{R}_{mj} = \underset{R_{mj}}{\operatorname{argmin}} \sum_{i=1}^{N} L(y_i, f_{m-1}(x_i) + T(x_i; R_j)) \,.$$

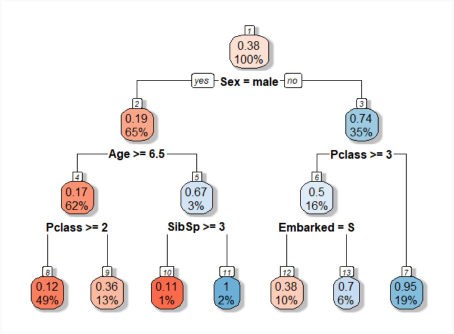
Boosting Trees: Hyperparameters

There are two hyperparameters we need to set, for boosting trees, the tree depth J and the number M of boosted models.

Traditionally, the stepwise tree building algorithms follow those discussed previously, growing and trimming as appropriate. This approach assumes that each tree is the last one in the chain, which unless it happens to be often results in trees that are much too large.

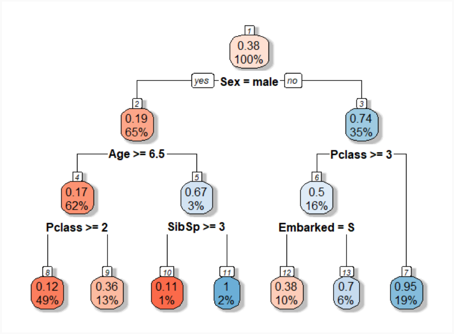
One way to avoid this problem is to restrict all trees to be of the same size J. Thus, J becomes a hyperparameter, and needs to be fit or chosen.

Tree Size



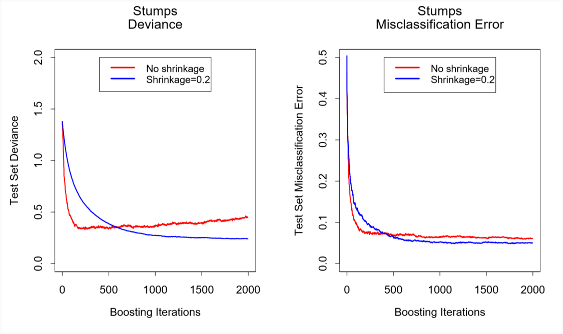
We should understand J as roughly giving the degree of interaction between the variables. That is, for a single split (decision stump) $J = 2$, we expect interaction between the variables. For $J = 3$, we would expect quadratic terms like $X^i X^j$ to be model. In general we expect interaction terms of order $J - 1$ to be model-able.

Tree Size



As a rule, J can be fit for low degrees, and usually provides reasonable results for $4 \leq J \leq 8$ (ESLII).

The Boosted VC dimension

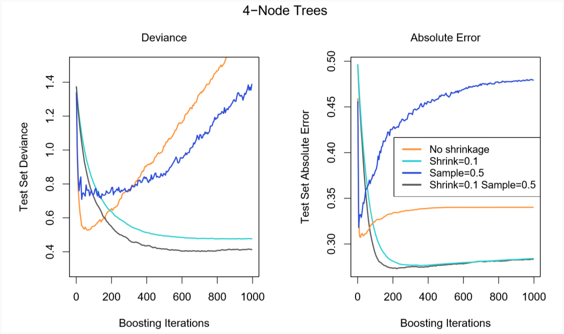


The number of trees M must be determine as well. Each iteration lowers the training risk so we must specify stopping criteria. One method is to shrink each new tree by a factor $\nu \in (0, 1)$ after fitting it:

$$f_m(x) = f_{m-1}(x) + \nu T(x; R_{jm})$$

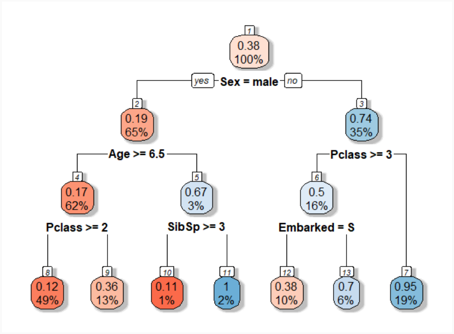
A lower shrinkage factor ν will take long to fit, and so drive M up, but compare the test results above.

The Boosted VC dimension



SGD can also improve computational speed while sometime even improving accuracy. In **stochastic gradient boosting**, at each iteration we remove a sub-sample $\mathcal{S} \subset \mathcal{T}$ from the training set and train the next classifier only on $\mathcal{S}$. Typically, $1/2$ of the data is sampled at each step.

Tree Size



We should understand J as roughly giving the degree of interaction between the variables. That is, for a single split (decision stump) $J = 2$, we expect interaction between the variables. For $J = 3$, we would expect quadratic terms like $X^i X^j$ to be model. In general we expect interaction terms of order $J - 1$ to be model-able.

Interpreting Boosted Trees

Relative Feature Importance

In boosting trees, we've gained a computationally efficient algorithm where the bias-variance tradeoff is well understood but we've lose interpretive power. One thing we can recover is a notion of relative importance between features.

Define the **relative importance of variables** $I_\ell^2(T)$ of a variable X_ℓ in a tree T to be

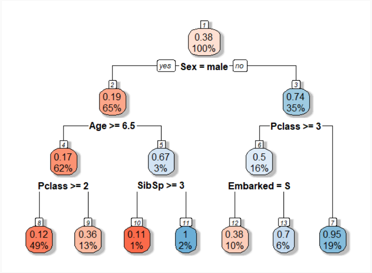
$$I_\ell^2(T) = \sum_{t=1}^{J-1} \iota_i^2 \mathbb{1}(v(t) = \ell)\,,$$

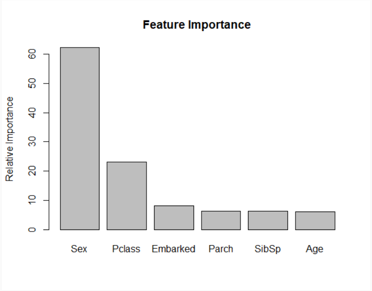
where $v(t)$ is the split variable at the t'th step and ι^2 is the difference in squared error between not splitting over R_j and splitting over R_j. That is

$$\iota^2 = \sum_{i:x_i \in R_j} (y_i - \bar{y}_{R_j})^2 - \sum_{i:x_i \in R_j} (y_i - h_{\theta_j}(x_i))^2\,,$$

where h_θ is the decision stump at R_j of the tree T.

Relative Feature Importance





For example, we see the feature importance for our tree from the titanic dataset. **Sex** and **Pclass** account for the vast majority of the difference in error along the tree. **Age** it turns out is a good predictor, but on a small set of samples and so is weighted less than **Pclass**, which is a moderate predictor for a large amount of data.

Relative Feature Importance

The relative importance of variables isn't always stable for trees, but it is much more stable for ensembles of trees. For an ensemble of tree, define the relative importance of the feature ℓ to be

$$I_\ell^2 = \frac{1}{M} \sum_{m=1}^{M} I_\ell^2(T_m)\,.$$

If the feature is categorical, we instead use

$$I_\ell^2 = \frac{1}{K} \sum_{k=1}^{K} I_{\ell k}^2\,, \quad \text{where} \quad I_{\ell k}^2 = \frac{1}{M} \sum_{m=1}^{M} I_\ell^2(T_{mk})\,.$$

Partial Dependence

After identifying the most important features X_S, $S \subset \{1, \ldots, p\}$, we can try to quantify the dependence of the less important variables $X_{\bar{S}}$ on X_S. For any blackbox method, we can define the **partial dependence** of $X_{\bar{S}}$ on X_S (or **marginal average**) by

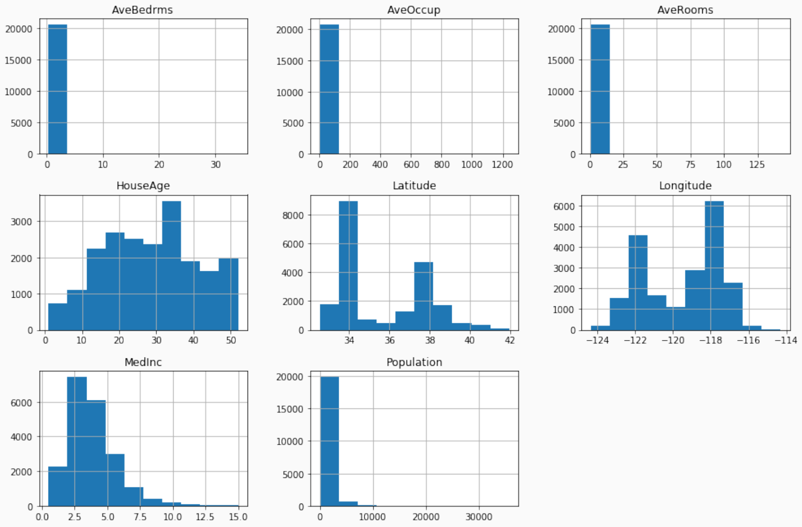
$$f_S(X_S) = E_{X_{\bar{S}}} f(X_S, \bar{S})\,.$$

We can estimate the marginal average by

$$\hat{f}_S(X_S) = \frac{1}{N} \sum_{i=1}^{N} f(X_S, x_{\bar{S}i})\,.$$

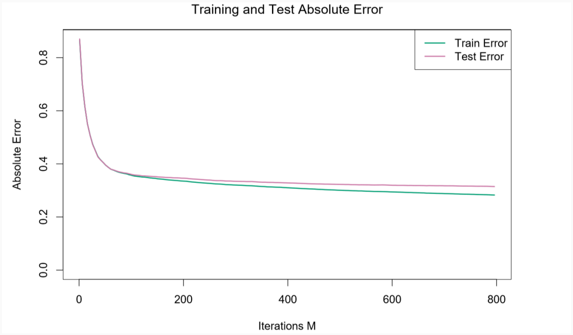
For a general hypothesis class this can be computationally intensive, but for learning trees we can simply define $\hat{f}_S(X_S)$ by removing all stumps in variables $X_{\bar{S}}$. **(Exercise)**

Example: California Housing



Consider the California Housing Dataset, where price is predicted based on features including location.

Example: California Housing

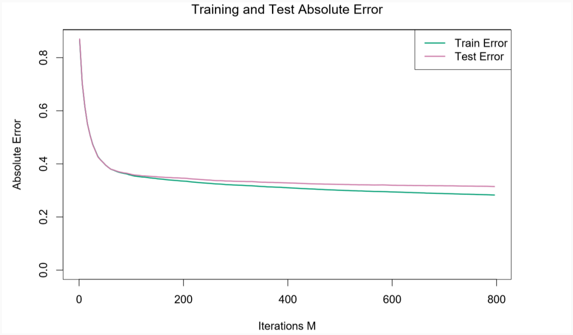


Consider the California Housing Dataset, where price is predicted based on features including location. In ESLII the data are fit with boosted tree model with $J = 6$ and learning rate $\nu = .1$. The absolute error for various M

$$\text{AAE} = E|y - \hat{f}_M(x)|$$

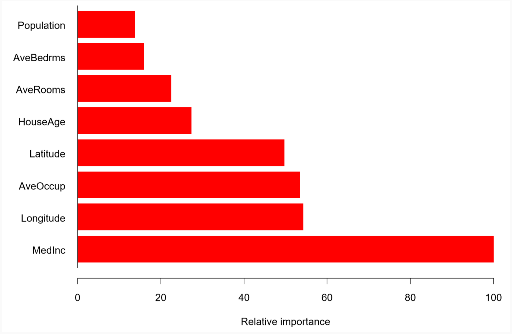
is charted above.

Example: California Housing



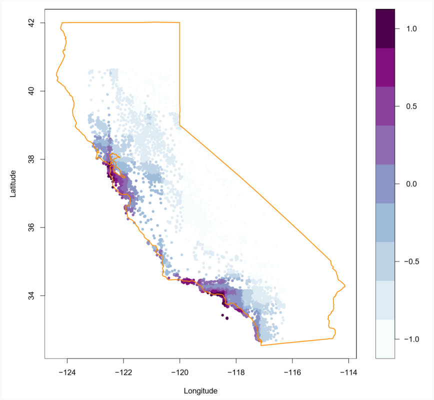
After 800 iterations, the squared multiple correlation coefficient is computed to be $R^2 = 0.84$, and that raises to $R^2 = 0.86$ when $\log Y$ is used as a response. This is compared to neighborhood based models of median housing prices with achieve $R^2 = .85$.

Example: California Housing



Charting the relative importance of the features, we find that the median income, spacial local and average occupancy are the most important features.

Example: California Housing



Finally, we compute the partial dependence on the latitude and longitude. We see that as we would expect housing is more expensive in the Bay Area and LA corridor, while the eastern desert regions tend to be cheaper. Values are in \$100,000 away from \$180,000 in 1990's dollars.

Reference

The main references are Chapter ESLII Chapter 10 and UML Chapter 10.