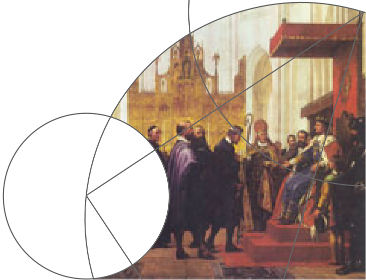
Faculty of Science

Combining Multiple Learners

Elements of Machine Learning

Jens Petersen

with content from Elements of Statistical Learning, Hastie, Tibshirani and Friedman



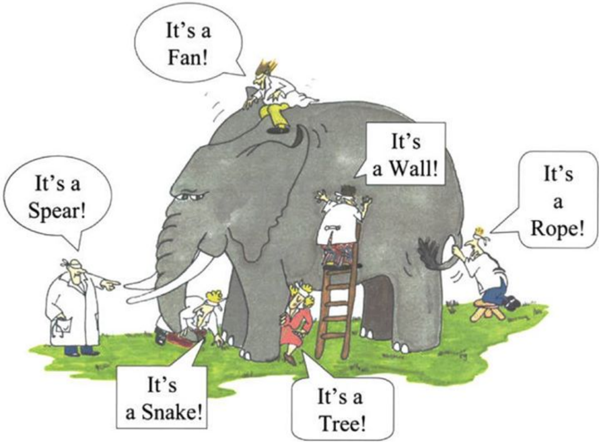
About this lecture

Combining multiple learners..

- Boostrap methods
- Bagging
- Model averaging and stacking
- Boosting

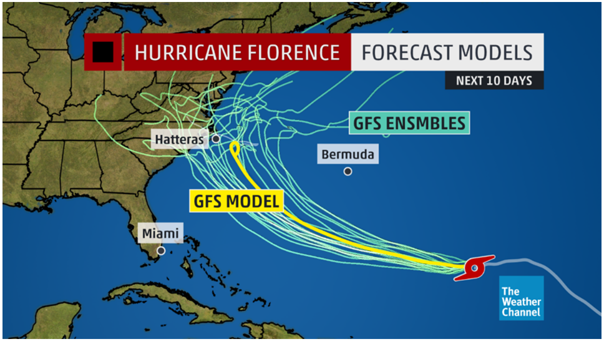
Whats the point?

- One can almost always improve results over a single ML method by combining multiple ML methods.



Whats the point?

- One can almost always improve results over a single ML method by combining multiple ML methods.



Bootstrap methods

The bootstrap method can be used estimate statistical properties of a population from small samples.

Possibly originates from the phrase ”pull oneself over a fence by one’s bootstraps”.[1]

[1] Figure from
https://za.toluna.com/opinions/4015146/What-Pull-Yourself-Up-By-The-Bootstraps-Really-Means...

Bootstrap methods

The bootstrap method can be used estimate statistical properties of a population from small samples.

Given a dataset $\mathbf{Z} = (z_1, z_2, \ldots, Z_n)$ where $z_i = (x_i, y_i)$.

Bootstrap dataset Sample a dataset $\mathbf{Z}^{*b}$ from $\mathbf{Z}$ with replacement. Commonly $\mathbf{Z}^{*b}$ and $\mathbf{Z}$ are chosen to be of the same size.

Bootstrap methods

The bootstrap method can be used estimate statistical properties of a population from small samples.

Given a dataset $\mathbf{Z} = (z_1, z_2, \ldots, Z_n)$ where $z_i = (x_i, y_i)$.

Bootstrap dataset Sample a dataset $\mathbf{Z}^{*b}$ from $\mathbf{Z}$ with replacement. Commonly $\mathbf{Z}^{*b}$ and $\mathbf{Z}$ are chosen to be of the same size.

As an example consider the case $\mathbf{Z} = (z_1, z_2, z_3, z_4)$ bootstrap sampling from this could return

$$\begin{gathered}\mathbf{Z}^{*1} = (z_1, z_2, z_2, z_4)\\ \mathbf{Z}^{*2} = (z_3, z_3, z_4, z_4)\\ \vdots\end{gathered}$$

Bootstrap methods

1. Iterate B times
   a. Sample bootstrap dataset from the training data with replacement
   b. Train a model on the bootstrap dataset
2. Examine the behaviour of the fits over the B replications

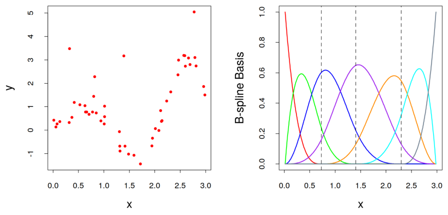
Bootstrap methods

1. Iterate B times
   a. Sample bootstrap dataset from the training data with replacement
   b. Train a model on the bootstrap dataset
2. Examine the behaviour of the fits over the B replications

We can use this to estimate

- variance $\hat{\mathrm{Var}}[S(\mathbf{Z})] = \frac{1}{B-1}\sum_{b=1}^{B}(S(\mathbf{Z}^{*b}) - \bar{S}^{*})^2$
- where the mean is estimated as $\bar{S}^{*} = \sum_b S(\mathbf{Z}^{*b})/B$

Bootstrap methods - uncertainty

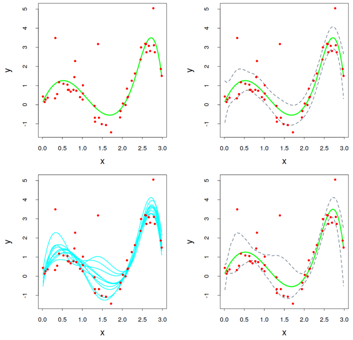


Suppose we want to fit B-spline basis functions (right) to a one-dimensional input and output problem (left).

$$\mu(x) = \sum_{j}^{7} \beta_j h_j(x)$$

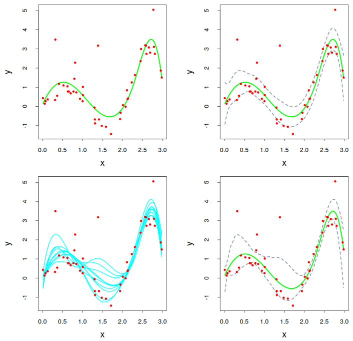
Here 7 is the number of basis functions covering the space of the x-coordinate.

Bootstrap methods - uncertainty



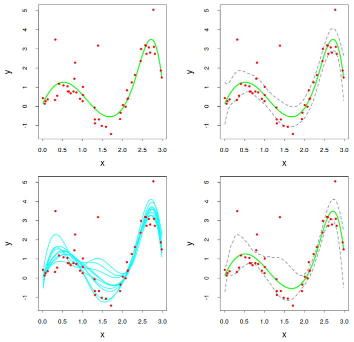
Top-left - data with one B-spline fit

Bootstrap methods - uncertainty



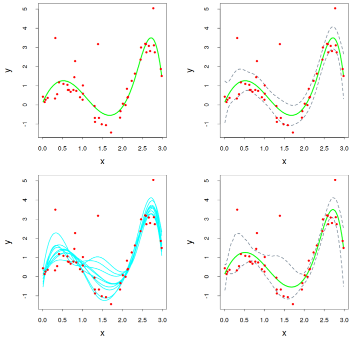
Top-right - 1.96$\times$ standard error bands based on analytical approach

Bootstrap methods - uncertainty



Bottom-left - 10 bootstrap samples

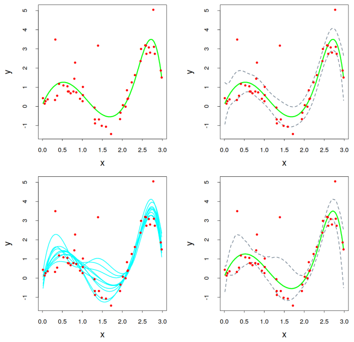
Bootstrap methods - uncertainty



Bottom-right

- Drawing B ($N = 50$) datasets with replacement and fitting the cubic spline
- Using $B = 200$ samples 95% confidence bands as the $2.5\% \times 200$ largest and smallest values at each x.

Bootstrap methods - uncertainty



Note the similarities of the bootstrap estimate and analytical approach.

Why does bootstrapping work?

Or asked in another way, why does repeatedly subsampling a sample tell us something about the distribution that the original sample was drawn from?

Why does bootstrapping work?

Or asked in another way, why does repeatedly subsampling a sample tell us something about the distribution that the original sample was drawn from?

It does not! But we are assuming it does. What we are doing instead is to replace something we do not know 'the distribution' with something we know the 'the sample'.

Why does bootstrapping work?

Or asked in another way, why does repeatedly subsampling a sample tell us something about the distribution that the original sample was drawn from?

It does not! But we are assuming it does. What we are doing instead is to replace something we do not know 'the distribution' with something we know the 'the sample'.

The bootstrap principle is that this substitution typically works, that is the sample represents the distribution well, but we need to watch out for cases where this is not so.

Parametric and non-parametric bootstrap

The described procedure is called **non-parametric bootstrap** - where we replace the distribution with an empirical distribution (sample).

We may also assume a model, e.g. Gaussian distribution, and estimate its parameters based on the sample and draw bootstrap samples from this model. This is called **parametric bootstrap**.

- One could in some cases resample both x and y given the fitted model, however, this may be too much trouble if we are only interested in the variability of the model output
- Alternatively, we may sample from the residuals...

Parametric bootstrap - example

Assume

$$Y = \mu(X) + \epsilon; \epsilon \sim N(0, \sigma^2), \tag{1}$$

$$\mu(x) = \sum_{j=1}^{7} \beta_j h_j(x) \tag{2}$$

We can then draw new bootstrap samples by adding Gaussian noise to the response

$$y_i^* = \hat{u}(x_i) + \epsilon_i^*; \epsilon_i^* \sim N(0, \hat{\sigma}^2); i = 1, 2, \ldots, N$$

Leading to new samples x_i, y_i^*.

Bootstrap versus maximum likelihood

In general bootstrap estimates agree with maximum likelihood (derivations in ESL for above example), however

- allows computation of estimates of standard errors and other quantities in settings where no formulas exist

Bagging

Bagging (also called bootstrap aggregating) averages the prediction over the bootstrap samples

$$\hat{f}_{\mathrm{bag}}(x) = \frac{1}{B} \sum_{b=1}^{B} \hat{f}^{*b}(x),$$

where $\mathbf{Z}^{*b}$, $b = 1, 2, \dots, B$ are the bootstrap samples with predictions $\hat{f}^{*b}(x)$.

Bagging

Bagging (also called bootstrap aggregating) averages the prediction over the bootstrap samples

$$\hat{f}_{\mathrm{bag}}(x) = \frac{1}{B}\sum_{b=1}^{B}\hat{f}^{*b}(x),$$

where $\mathbf{Z}^{*b}$, $b = 1, 2, \ldots, B$ are the bootstrap samples with predictions $\hat{f}^{*b}(x)$.

What about classification with K classes?

Bagging

Bagging (also called bootstrap aggregating) averages the prediction over the bootstrap samples

$$\hat{f}_{\mathrm{bag}}(x) = \frac{1}{B}\sum_{b=1}^{B} \hat{f}^{*b}(x),$$

where $\mathbf{Z}^{*b}$, $b = 1, 2, \ldots, B$ are the bootstrap samples with predictions $\hat{f}^{*b}(x)$.

What about classification with K classes?
The bagged estimate is a K-vector $[p_1(x), p_2(x), \ldots, p_K(x)]$, with $p_k(x)$ equal to the proportion of classifiers predicting k and the bagged classifier gives the class with most votes.

$$\hat{G}_{\mathrm{bag}}(x) = \arg\max_k \hat{f}_{\mathrm{bag}}(x)$$

Bagging

Bagging (also called bootstrap aggregating) averages the prediction over the bootstrap samples

$$\hat{f}_{\mathrm{bag}}(x) = \frac{1}{B}\sum_{b=1}^{B} \hat{f}^{*b}(x),$$

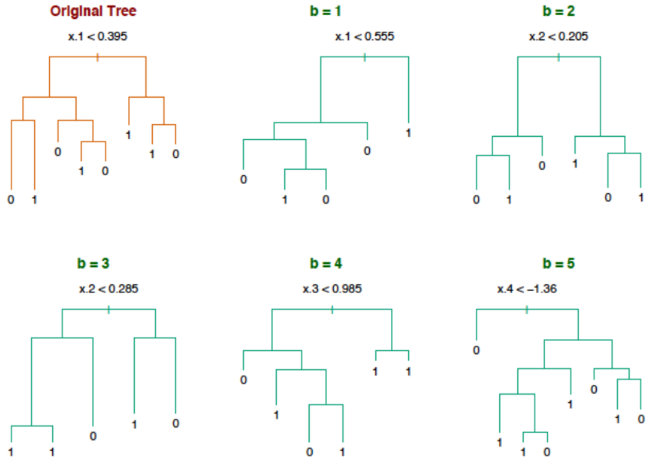
where $\mathbf{Z}^{*b}$, $b = 1, 2, \ldots, B$ are the bootstrap samples with predictions $\hat{f}^{*b}(x)$.

What about classification with K classes?
The bagged estimate is a K-vector $[p_1(x), p_2(x), \ldots, p_K(x)]$, with $p_k(x)$ equal to the proportion of classifiers predicting k and the bagged classifier gives the class with most votes.

$$\hat{G}_{\mathrm{bag}}(x) = \arg\max_k \hat{f}_{\mathrm{bag}}(x)$$

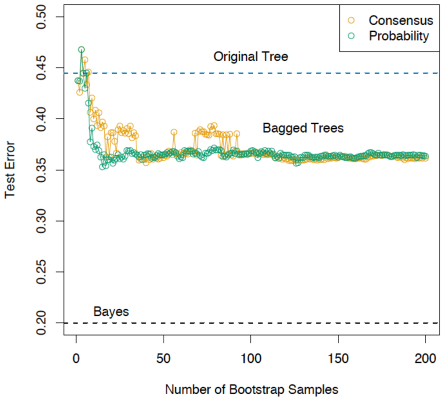
Note voting proportions are not probabilities! If the classifier can return probabilities, average these instead!

Bagging example with decision trees



Five trees grown on boostrap samples.

Bagging example with decision trees



Bagging decreases misclassification rate. Note the difference in using averages of decision tree probabilities (based on the proportion of points in each leaf) and the majority label

Why does bagging work?

- Bagging reduces variability of individual base learners
  - Working best for unstable, high variance base learners
  - Algorithms that are sensitive to small changes in training data (decision trees, KNN with small k)

Why does bagging work - wisdom of crowds

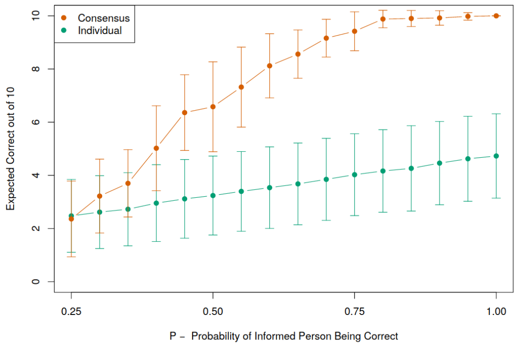


FIGURE 8.11. *Simulated academy awards voting. 50 members vote in 10 categories, each with 4 nominations. For any category, only 15 voters have some knowledge, represented by their probability of selecting the "correct" candidate in that category (so $P = 0.25$ means they have no knowledge). For each category, the 15 experts are chosen at random from the 50. Results show the expected correct (based on 50 simulations) for the consensus, as well as for the individuals. The error bars indicate one standard deviation. We see, for example, that if the 15 informed for a category have a 50% chance of selecting the correct candidate, the consensus doubles the expected performance of an individual.*

When does bagging not help?

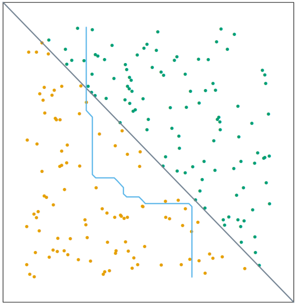
- Example: linear regression is not improved by bagging. We will simply arrive at the original sample predictions for large enough b
    - Convex problem, so we have the best possible solution given the original sample.
    - Variance is introduced through the bootstrap samples.

When does bagging not help?

- Example: linear regression is not improved by bagging. We will simply arrive at the original sample predictions for large enough b
  - Convex problem, so we have the best possible solution given the original sample.
  - Variance is introduced through the bootstrap samples.
- Bagging a bad classifier (no better than random), may lead to even worse results (example in ESL).

When does bagging work less well?

Another example, learning a diagonal decision rule with axis-aligned splits



Because each bootstrap sample will tend to give similar horizontal and vertical splits, it is difficult for bagging to learn a useful combination.

Limitations of bagging

- Loss of Interpretability: if we have an interpretable model, like a decision tree, once we combine multiple of them we lose a lot of the interpretability.
- Increased computational complexity: as opposed to training a single model we now need to train B models.

Model averaging and stacking

Instead of a simple average of models we could seek a weighted average

$$\hat{f}^{st}(x) = \sum_{m=1}^{M} w_m \hat{f}_m(x)$$

such that

$$\hat{w} = \arg\min_{w} E_{\mathcal{P}} \big[Y - \sum_{m=1}^{M} w_m \hat{f}_m(x) \big]^2,$$

That is, we are minimizing the expected $E_{\mathcal{P}}$ squared loss over the dataset distribution $\mathcal{P}$.

Model averaging and stacking

Instead of a simple average of models we could seek a weighted average

$$\hat{f}^{st}(x) = \sum_{m=1}^{M} w_m \hat{f}_m(x)$$

such that

$$\hat{w} = \arg\min_{w} E_{\mathcal{P}}\left[Y - \sum_{m=1}^{M} w_m \hat{f}_m(x)\right]^2,$$

That is, we are minimizing the expected $E_{\mathcal{P}}$ squared loss over the dataset distribution $\mathcal{P}$.

Note, we might additionally require $\sum_{m=1}^{M} w_m = 1$ and $w_m \geq 0, m = 1, 2, \ldots, M$.

Model averaging and stacking

Instead of a simple average of models we could seek a weighted average

$$\hat{f}^{st}(x) = \sum_{m=1}^{M} w_m \hat{f}_m(x)$$

such that

$$\hat{w} = \arg\min_{w} E_{\mathcal{P}} \left[Y - \sum_{m=1}^{M} w_m \hat{f}_m(x) \right]^2,$$

That is, we are minimizing the expected $E_{\mathcal{P}}$ squared loss over the dataset distribution $\mathcal{P}$.

How can we approximate the expectation and find the weights?

Model averaging and stacking

Instead of a simple average of models we could seek a weighted average

$$\hat{f}^{st}(x) = \sum_{m=1}^{M} w_m \hat{f}_m(x)$$

such that

$$\hat{w} = \arg\min_{w} E_{\mathcal{P}} \left[Y - \sum_{m=1}^{M} w_m \hat{f}_m(x) \right]^2,$$

That is, we are minimizing the expected $E_{\mathcal{P}}$ squared loss over the dataset distribution $\mathcal{P}$.

How can we approximate the expectation and find the weights?

Using the training set would not work well, as we would not consider model complexity and thus how much each model overfits

Model averaging and stacking

We could use cross-validation or leave one out and predict on the held out data set

$$\hat{w}^{st} = \arg\min_{w} \sum_{i=1}^{N} \left[y_i - \sum_{m=1}^{M} w_m \hat{f}_m^{-i}(x_i)\right]^2,$$

where $\hat{f}_m^{-i}(x_i)$, with $m = 1, 2, \ldots, M$ is the prediction at x, using model m, applied to the dataset with the ith training observation removed.

Boosting

A weak model/classifier - a model/classifier that is only slightly better than random guessing

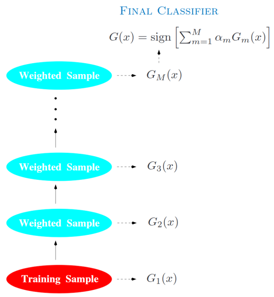
Boosting - Sequentially training weak classifiers and adding them to an overall strong classifier, reweighting data such that future weak learners focus on the currently wrongly classified data.

Boosting - AdaBoost.M1

Consider a two-class problem, with the output variable coded as $Y \in \{-1, 1\}$. We apply weak classifiers in sequence $G_m(x), m = 1, 2, \ldots M$ and combine them through a weighted majority vote

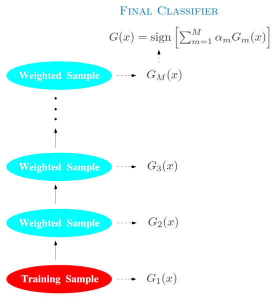
$$G(x) = \mathrm{sign}\Big(\sum_{m=1}^{M} \alpha_m G_m(x)\Big)$$

Boosting - AdaBoost.M1



Adaboost sequentially applies classifiers to the problem, each time reweighting the training observations.

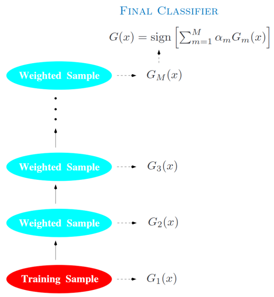
Boosting - AdaBoost.M1



Adaboost sequentially applies classifiers to the problem, each time reweighting the training observations.

- increasing the weight of observations which were misclassified by the previous classifier, and

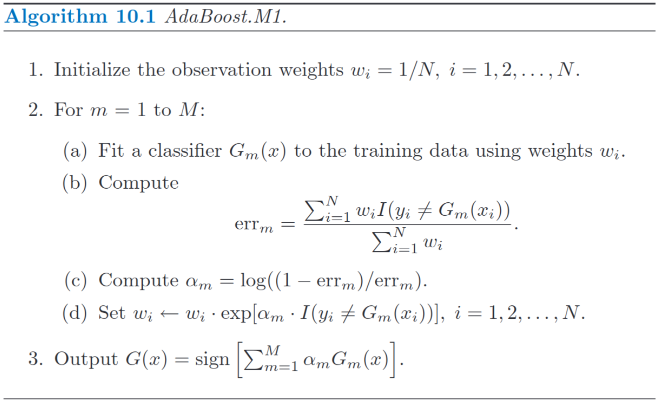
Boosting - AdaBoost.M1



Adaboost sequentially applies classifiers to the problem, each time reweighting the training observations.

- as iterations proceed, observations that are difficult to classify receive ever-increasing influence

Boosting - AdaBoost.M1

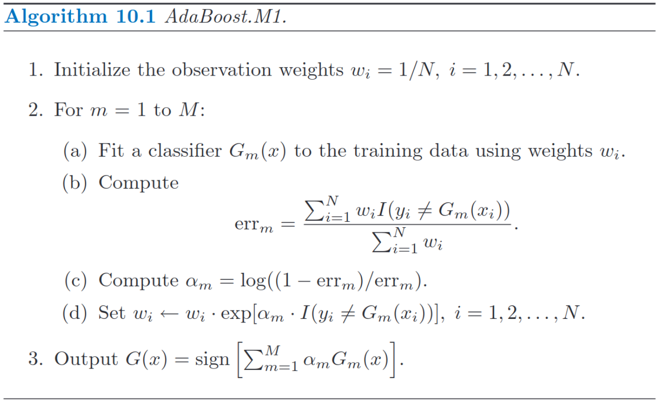


Algorithm 10.1 *AdaBoost.M1.*

```
1. Initialize the observation weights $w_i = 1/N,\ i = 1, 2, \ldots, N$.
2. For $m = 1$ to $M$:
    (a) Fit a classifier $G_m(x)$ to the training data using weights $w_i$.
    (b) Compute
        $$\text{err}_m = \frac{\sum_{i=1}^{N} w_i I(y_i \neq G_m(x_i))}{\sum_{i=1}^{N} w_i}.$$
    (c) Compute $\alpha_m = \log((1 - \text{err}_m)/\text{err}_m)$.
    (d) Set $w_i \leftarrow w_i \cdot \exp[\alpha_m \cdot I(y_i \neq G_m(x_i))],\ i = 1, 2, \ldots, N$.
3. Output $G(x) = \text{sign}\left[\sum_{m=1}^{M} \alpha_m G_m(x)\right]$.
```

The weak classifier is fitted to the weighted data points from the previous iteration in (a)

Boosting - AdaBoost.M1

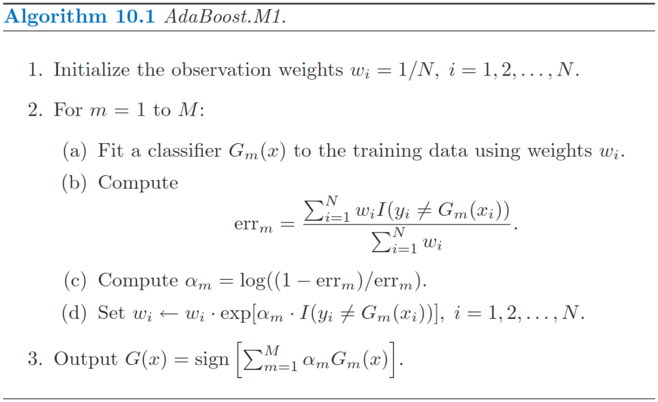


Algorithm 10.1 *AdaBoost.M1.*

```
1. Initialize the observation weights $w_i = 1/N,\ i = 1, 2, \dots, N$.
2. For $m = 1$ to $M$:
    (a) Fit a classifier $G_m(x)$ to the training data using weights $w_i$.
    (b) Compute
        $$\mathrm{err}_m = \frac{\sum_{i=1}^{N} w_i I(y_i \neq G_m(x_i))}{\sum_{i=1}^{N} w_i}.$$
    (c) Compute $\alpha_m = \log((1 - \mathrm{err}_m)/\mathrm{err}_m)$.
    (d) Set $w_i \leftarrow w_i \cdot \exp[\alpha_m \cdot I(y_i \neq G_m(x_i))],\ i = 1, 2, \dots, N$.
3. Output $G(x) = \mathrm{sign}\left[\sum_{m=1}^{M} \alpha_m G_m(x)\right]$.
```

The error rate of the weak classifier on the weighted training set is computed in (b)

Boosting - AdaBoost.M1

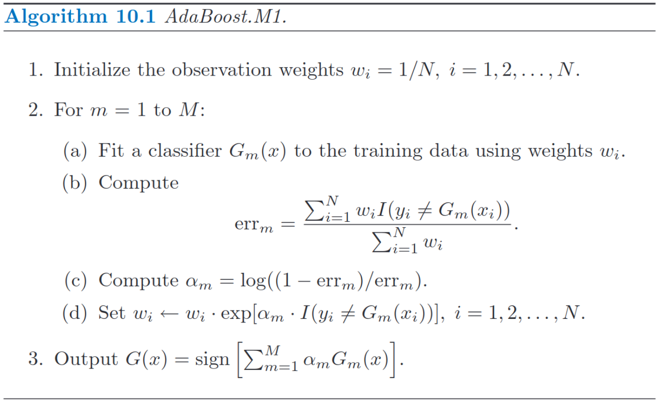


Algorithm 10.1 *AdaBoost.M1.*

```
1. Initialize the observation weights $w_i = 1/N,\ i = 1, 2, \ldots, N$.
2. For $m = 1$ to $M$:
    (a) Fit a classifier $G_m(x)$ to the training data using weights $w_i$.
    (b) Compute
        $$\text{err}_m = \frac{\sum_{i=1}^{N} w_i I(y_i \neq G_m(x_i))}{\sum_{i=1}^{N} w_i}.$$
    (c) Compute $\alpha_m = \log((1 - \text{err}_m)/\text{err}_m)$.
    (d) Set $w_i \leftarrow w_i \cdot \exp[\alpha_m \cdot I(y_i \neq G_m(x_i))],\ i = 1, 2, \ldots, N$.
3. Output $G(x) = \text{sign}\left[\sum_{m=1}^{M} \alpha_m G_m(x)\right]$.
```

The weight given to the $G_m(x)$ classifer is computed from its error rate in (c)

Boosting - AdaBoost.M1



Algorithm 10.1 *AdaBoost.M1.*

```
1. Initialize the observation weights $w_i = 1/N,\ i = 1, 2, \ldots, N$.
2. For $m = 1$ to $M$:
    (a) Fit a classifier $G_m(x)$ to the training data using weights $w_i$.
    (b) Compute
        $$\mathrm{err}_m = \frac{\sum_{i=1}^{N} w_i I(y_i \neq G_m(x_i))}{\sum_{i=1}^{N} w_i}.$$
    (c) Compute $\alpha_m = \log((1 - \mathrm{err}_m)/\mathrm{err}_m)$.
    (d) Set $w_i \leftarrow w_i \cdot \exp[\alpha_m \cdot I(y_i \neq G_m(x_i))],\ i = 1, 2, \ldots, N$.
3. Output $G(x) = \mathrm{sign}\left[\sum_{m=1}^{M} \alpha_m G_m(x)\right]$.
```

Lets look at this a little closer, $\mathrm{err}_m = 0.5 \Rightarrow \alpha_m = 0$, $\mathrm{err}_m < 0.5 \Rightarrow \alpha_m > 0$, and $\mathrm{err}_m > 0.5 \Rightarrow \alpha_m < 0$.

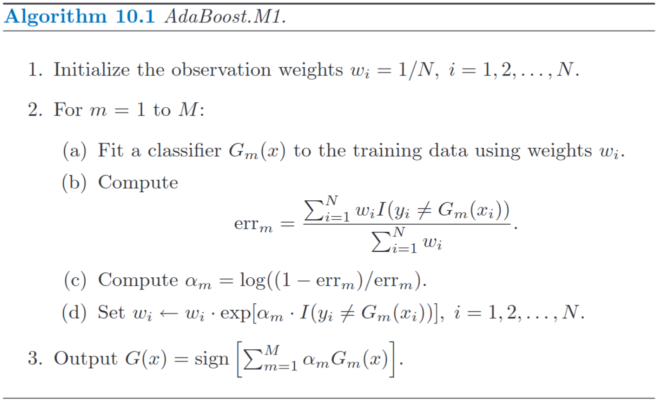
Boosting - AdaBoost.M1

Algorithm 10.1 *AdaBoost.M1.*

```
1. Initialize the observation weights $w_i = 1/N,\ i = 1, 2, \ldots, N$.
2. For $m = 1$ to $M$:
    (a) Fit a classifier $G_m(x)$ to the training data using weights $w_i$.
    (b) Compute
            $\mathrm{err}_m = \frac{\sum_{i=1}^{N} w_i I(y_i \neq G_m(x_i))}{\sum_{i=1}^{N} w_i}$.
    (c) Compute $\alpha_m = \log((1 - \mathrm{err}_m)/\mathrm{err}_m)$.
    (d) Set $w_i \leftarrow w_i \cdot \exp[\alpha_m \cdot I(y_i \neq G_m(x_i))],\ i = 1, 2, \ldots, N$.
3. Output $G(x) = \mathrm{sign}\left[\sum_{m=1}^{M} \alpha_m G_m(x)\right]$.
```

So classifiers that perform randomly do not contribute, classifiers that perform better than random, contribute possitively, and classifiers that perform worse than random contribute with negative sign.

Boosting - AdaBoost.M1

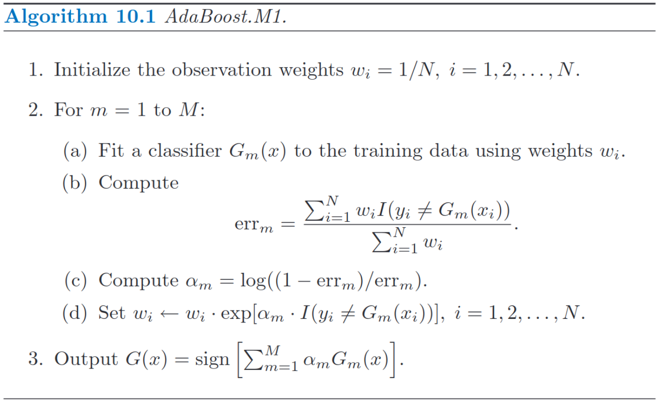


Algorithm 10.1 *AdaBoost.M1.*

```
1. Initialize the observation weights $w_i = 1/N$, $i = 1, 2, \ldots, N$.
2. For $m = 1$ to $M$:
    (a) Fit a classifier $G_m(x)$ to the training data using weights $w_i$.
    (b) Compute
            $$\text{err}_m = \frac{\sum_{i=1}^{N} w_i I(y_i \neq G_m(x_i))}{\sum_{i=1}^{N} w_i}.$$
    (c) Compute $\alpha_m = \log((1 - \text{err}_m)/\text{err}_m)$.
    (d) Set $w_i \leftarrow w_i \cdot \exp[\alpha_m \cdot I(y_i \neq G_m(x_i))]$, $i = 1, 2, \ldots, N$.
3. Output $G(x) = \text{sign}\left[\sum_{m=1}^{M} \alpha_m G_m(x)\right]$.
```

The weights for misclassified points are updated, by scaling them by $\exp(\alpha_m)$ in (d). Note that this leads to weights less than 1 for $\text{err}_m > 0.5$ and weights larger than 1 for $\text{err}_m < 0.5$.

Boosting - AdaBoost.M1

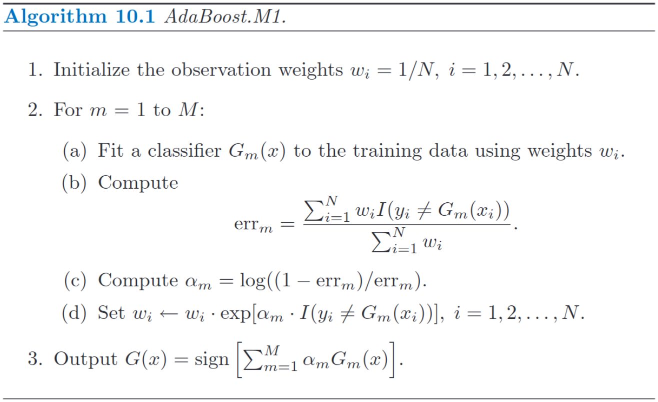


Algorithm 10.1 *AdaBoost.M1.*

```
1. Initialize the observation weights $w_i = 1/N,\ i = 1, 2, \ldots, N$.
2. For $m = 1$ to $M$:
    (a) Fit a classifier $G_m(x)$ to the training data using weights $w_i$.
    (b) Compute
        $$\text{err}_m = \frac{\sum_{i=1}^{N} w_i I(y_i \neq G_m(x_i))}{\sum_{i=1}^{N} w_i}.$$
    (c) Compute $\alpha_m = \log((1 - \text{err}_m)/\text{err}_m)$.
    (d) Set $w_i \leftarrow w_i \cdot \exp[\alpha_m \cdot I(y_i \neq G_m(x_i))],\ i = 1, 2, \ldots, N$.
3. Output $G(x) = \text{sign}\left[\sum_{m=1}^{M} \alpha_m G_m(x)\right]$.
```

The process is repeated for the next classifier $G_{m+1}(x)$ or the process terminates if M is reached

Boosting - AdaBoost.M1



Algorithm 10.1 *AdaBoost.M1.*

```
1. Initialize the observation weights $w_i = 1/N,\ i = 1, 2, \ldots, N$.
2. For $m = 1$ to $M$:
    (a) Fit a classifier $G_m(x)$ to the training data using weights $w_i$.
    (b) Compute
        $$\text{err}_m = \frac{\sum_{i=1}^N w_i I(y_i \neq G_m(x_i))}{\sum_{i=1}^N w_i}.$$
    (c) Compute $\alpha_m = \log((1 - \text{err}_m)/\text{err}_m)$.
    (d) Set $w_i \leftarrow w_i \cdot \exp[\alpha_m \cdot I(y_i \neq G_m(x_i))],\ i = 1, 2, \ldots, N$.
3. Output $G(x) = \text{sign}\left[\sum_{m=1}^M \alpha_m G_m(x)\right]$.
```

What happens for a misclassified point given a worse than random classifier G_m and why?

Gradient boosting

We can write boosting as an additive model

$$fx = \sum_{m=1}^{M} f_m(x)$$

That is the model is a sum of the results of M weak learners.

Gradient boosting

We can write boosting as an additive model

$$fx = \sum_{m=1}^{M} f_m(x)$$

That is the model is a sum of the results of M weak learners.

We typically learn these models in a greedy sequential fashion where the models up to f_{m-1} exist and we want to learn f_m.

- We have seen that AdaBoost reweighted training data prior to learning G_m to emphasize difficult data points.

Gradient boosting

We can write boosting as an additive model

$$fx = \sum_{m=1}^{M} f_m(x)$$

That is the model is a sum of the results of M weak learners.

We typically learn these models in a greedy sequential fashion where the models up to f_{m-1} exist and we want to learn f_m.

- We have seen that AdaBoost reweighted training data prior to learning G_m to emphasize difficult data points.
- Gradient boosting instead takes a gradient step (in function space, we will get back to this) when learning f_m and therefore can be thought of as doing gradient descent on the loss function (in function space).

Gradient boosting

Suppose we are optimizing squared-error loss

$$L(y, f(x)) = \frac{1}{2}(y - f(x))^2$$

Gradient boosting

Suppose we are optimizing squared-error loss

$$L(y, f(x)) = \frac{1}{2}(y - f(x))^2$$

The gradient with respect to $f(x)$ can then be computed straightforwardly as

$$\frac{\partial L(y, f(x))}{\partial f(x)} = f(x) - y$$

Gradient boosting

Suppose we are optimizing squared-error loss

$$L(y, f(x)) = \frac{1}{2}(y - f(x))^2$$

The gradient with respect to $f(x)$ can then be computed straightforwardly as

$$\frac{\partial L(y, f(x))}{\partial f(x)} = f(x) - y$$

In other words with a squared-error loss, if we have fit f_{m-1} and want optimize by taking a step in the negative gradient direction, then we need to fit $f_m(x)$ to $y - f_{m-1}(x)$.

Gradient boosting

Suppose we are optimizing squared-error loss

$$L(y, f(x)) = \frac{1}{2}(y - f(x))^2$$

The gradient with respect to $f(x)$ can then be computed straightforwardly as

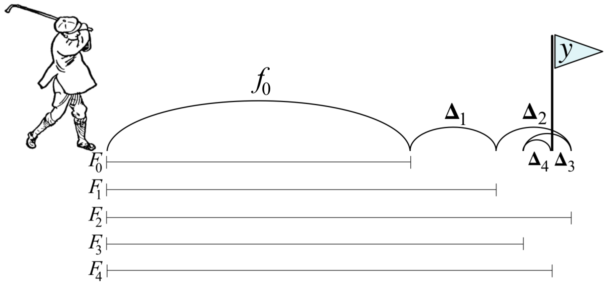
$$\frac{\partial L(y, f(x))}{\partial f(x)} = f(x) - y$$

In other words with a squared-error loss, if we have fit f_{m-1} and want optimize by taking a step in the negative gradient direction, then we need to fit $f_m(x)$ to $y - f_{m-1}(x)$.

- So we are fitting to the residuals of the previously fitted weak learner!
- In practice we scale the residuals with a learning rate (ν in ESL) to avoid overshooting

Gradient boosting

I found this drawing of the concept which I think illustrates it nicely[1]



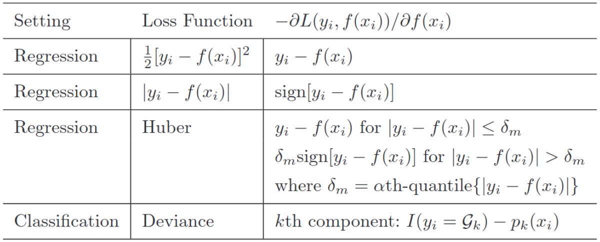
Golfer doing gradient boosting by continuously reassessing the distance to the hole

[1] from https://explained.ai/gradient-boosting/index.html

Gradient boosting

Fitting to the squared error can tend to make the boosting procedure chase outliers, so one may want to use another loss function.

TABLE 10.2. *Gradients for commonly used loss functions.*



Setting	Loss Function	$-\partial L(y_i, f(x_i))/\partial f(x_i)$
Regression	$\frac{1}{2}[y_i - f(x_i)]^2$	$y_i - f(x_i)$
Regression	$\|y_i - f(x_i)\|$	$\text{sign}[y_i - f(x_i)]$
Regression	Huber	$y_i - f(x_i)$ for $\|y_i - f(x_i)\| \leq \delta_m$ $\delta_m \text{sign}[y_i - f(x_i)]$ for $\|y_i - f(x_i)\| > \delta_m$ where $\delta_m = \alpha$th-quantile$\{\|y_i - f(x_i)\|\}$
Classification	Deviance	kth component: $I(y_i = \mathcal{G}_k) - p_k(x_i)$

Gradient boosting vs gradient descent in input space

Normal model optimization with gradient descent

- we compute the gradient of the loss with respect to the model parameters
- this way a gradient descent step directly changes model parameters in the direction of maximum decrease in the loss

Gradient boosting vs gradient descent in input space

Normal model optimization with gradient descent

- we compute the gradient of the loss with respect to the model parameters
- this way a gradient descent step directly changes model parameters in the direction of maximum decrease in the loss

With gradient boosting on the other hand

- we compute the gradient of the loss with respect to the weak models themselves and not with respect to their parameters
- the gradient step then becomes a two step process in which the weak model is fitted to the negative gradient and the overall model is updated by adding the new weak learner

Gradient boosting vs gradient descent in input space

Normal model optimization with gradient descent

- we compute the gradient of the loss with respect to the model parameters
- this way a gradient descent step directly changes model parameters in the direction of maximum decrease in the loss

With gradient boosting on the other hand

- we compute the gradient of the loss with respect to the weak models themselves and not with respect to their parameters
- the gradient step then becomes a two step process in which the weak model is fitted to the negative gradient and the overall model is updated by adding the new weak learner

One advantage of this is that we do not need to compute gradients in cases where it can be hard if not impossible (such as with decision trees)

Gradient boosting - hyper parameters

- Learning rate ν
- Number of iterations M

Gradient boosting - hyper parameters

- Learning rate ν
  - A large learning rate, sometimes called shrinkage, will increase the influence of each new weak learner and therefore increase the risk of overfitting (increase future risk)
  - A small learning rate will increase the time to predict with the overall model and time to converge on the training set.
- Number of iterations M

Gradient boosting - hyper parameters

- Learning rate ν
  - A large learning rate, sometimes called shrinkage, will increase the influence of each new weak learner and therefore increase the risk of overfitting (increase future risk)
  - A small learning rate will increase the time to predict with the overall model and time to converge on the training set.
- Number of iterations M
  - A large number of iterations will increase the accuracy and the risk of overfitting

Gradient boosting - hyper parameters

- Learning rate ν
  - A large learning rate, sometimes called shrinkage, will increase the influence of each new weak learner and therefore increase the risk of overfitting (increase future risk)
  - A small learning rate will increase the time to predict with the overall model and time to converge on the training set.
- Number of iterations M
  - A large number of iterations will increase the accuracy and the risk of overfitting

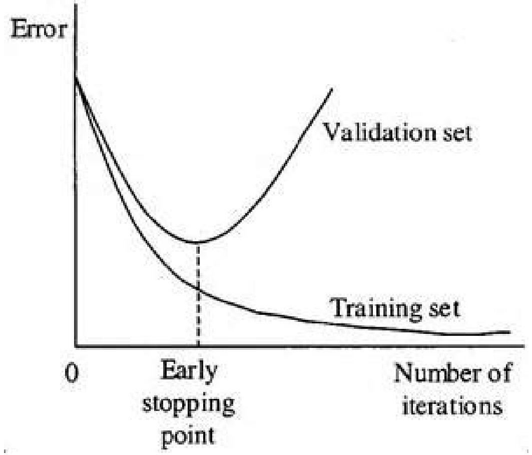
How should we choose ν and M?

Gradient boosting - hyper parameters

- Learning rate ν
    - A large learning rate, sometimes called shrinkage, will increase the influence of each new weak learner and therefore increase the risk of overfitting (increase future risk)
    - A small learning rate will increase the time to predict with the overall model and time to converge on the training set.
- Number of iterations M
    - A large number of iterations will increase the accuracy and the risk of overfitting

ν and M are obviously not independent. ESL advocates setting ν relatively small at around < 0.1 and then choose M by early stopping on an validation set (what is early stopping?)

Early stopping

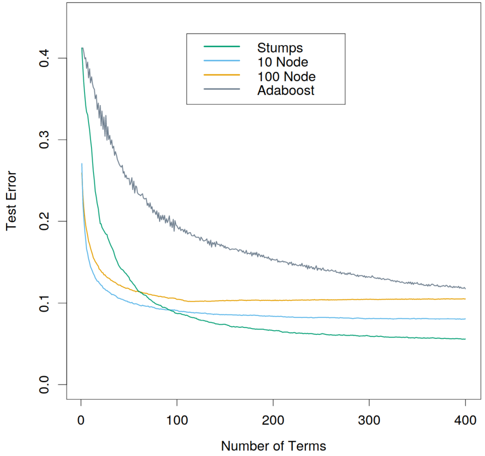


Gradient boosting with trees

How do we determine a good tree size?

- If we fit the optimal size (using some standard decision tree criteria) at each step, the first trees will be too large
- Simple strategy - use one fixed sized for trees at all iterations.
  - One extra hyper parameter - tree size
  - Optimal tree size is a function of the degree to which variables interact
  - Typically we do not need large trees. ESL claims trees with 4 to 8 terminal leaves often work the best.

Gradient boosting with trees



The generative model (ESL 10.2) is additive so stumps work best. Loss is binomial deviance.

Gradient boosting - subsampling

Suppose we have a large data and training each weak learner therefore becomes time-consuming.

- We can train each of them on a their own randomly sampled subset of the training set (without replacement).
- This is called **stochastic gradient boosting**
  - Decreases training time
  - May actually increase accuracy

Gradient boosting - subsampling

Suppose we have a large data and training each weak learner therefore becomes time-consuming.

- We can train each of them on a their own randomly sampled subset of the training set (without replacement).
- This is called **stochastic gradient boosting**
  - Decreases training time
  - May actually increase accuracy
  - Those of you familiar with stochastic gradient descent will recognize this technique.

Gradient boosting - subsampling

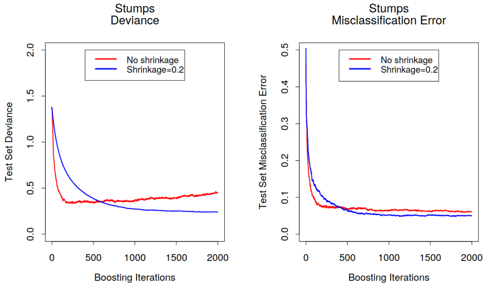
Suppose we have a large data and training each weak learner therefore becomes time-consuming.

- We can train each of them on a their own randomly sampled subset of the training set (without replacement).
- This is called **stochastic gradient boosting**
  - Decreases training time
  - May actually increase accuracy
  - Those of you familiar with stochastic gradient descent will recognize this technique.

Why do you think it may increase accuracy?

Gradient boosting - some examples

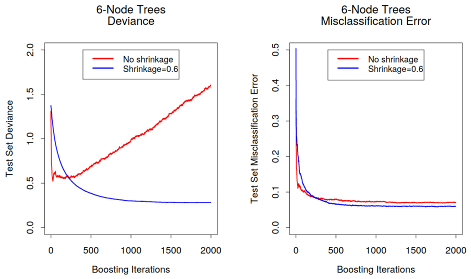
Test deviance and error curves for example in ESL 10.2.



Shrinkage (learning rates < 1) lowers overfitting, test error and in particular deviance, but needs more iterations to converge.

Gradient boosting - some examples

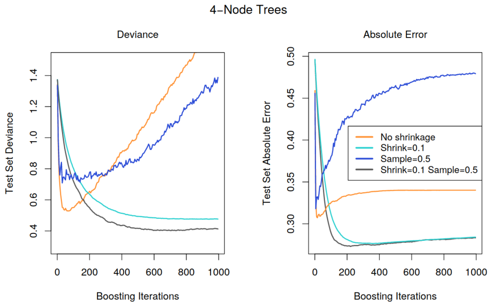
Test deviance and error curves for example in ESL 10.2.



Using 6-node trees shows increased overfitting, but shrinkage makes up for it.

Gradient boosting - some examples

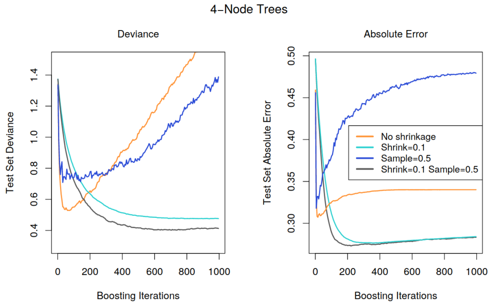
Test deviance and error curves for example in ESL 10.2.



Using subsampling with 4-node trees. Could you speculate as to why sampling works best with shrinkage?

Gradient boosting - some examples

Test deviance and error curves for example in ESL 10.2.



Using subsampling with 4-node trees. Could you speculate as to why sampling works best with shrinkage? Possible answer: maybe the subsampled data set is too small to establish a good gradient, or related, weak learners are difficult to fit on smaller data sets. However, if we combine subsampling with shrinkage (lower learning rate), then the consequence of bad gradients are smaller.