Module 9: Boosting and additive trees

TMA4268 Statistical Learning V2025

Stefanie Muff and Sara Martino, Department of Mathematical Sciences, NTNU

March 13 and 14, 2025

Learning material for this module

- James et al. (2021): Section 8.2.3 (Boosting)
- Hastie, Tibshirani, and Friedman (2009): The Elements of Statistical Learning, Chapter 10

In addition, some material we use is from the book by Boehmke and Greenwell (2020)

- https://bradleyboehmke.github.io/HOML/gbm.html

What will you learn?

- Boosting methods – general terms
- AdaBoost
- Gradient boosting
- Stochastic gradient boosting
- XGBoost: eXtreme Gradient Boosting
- Outlook to modern methods: Light GBM, catboost, ngboost
- Boosting of linear models (experience from a master student)

Boosting methods

- "Boosting" is one of the most powerful learning ideas that is currently around.
- First ideas in the 1990s and early 2000s (*e.g.*, Freund and Schapire (1997), Ridgeway (1999), J. Friedman, Hastie, and Tibshirani (2000))
- Boosting is a general method for building an ensemble out of simpler models.
- Boosting is usually applied to models with high bias and low variance, usually in the context of boosted regression and classification trees.
- It is also possible to boost, *e.g.*, linear or penalized regression models.

Boosted trees vs deep learning (neural networks)

- Boosted trees and neural networks (deep learning) are currently the most powerful competitors in statistical learning.
- Kaggle is a website where you find open competitions – many of them are won by tree-based methods.
- Maybe you will earn a living from this in the future...? See here: https://www.kaggle.com/competitions

Examples (recent literature)

Research | Open access | Published: 08 April 2022

Machine learning models outperform deep learning models, provide interpretation and facilitate feature selection for soybean trait prediction

Mitchell Gill, Robyn Anderson, Haifei Hu, Mohammed Bennamoun, Jakob Petereit, Babu Valliyodan, Henry T. Nguyen, Jacqueline Batley, Philipp E. Bayer & David Edwards

BMC Plant Biology **22**, Article number: 180 (2022) | Cite this article

From the abstract:

accurate prediction models. For 13/14 sets of predictions, XGBoost or random forest outperformed deep learning models in prediction performance. Top ranked SNPs by F-score were identified from

Why do tree-based models still outperform deep learning on typical tabular data?

Léo Grinsztajn, Edouard Oyallon, Gaël Varoquaux

See here to access the paper:

https://hal.science/hal-03723551v2

Recap: Trees and random forests

- See Module 8: Regression and classification trees.
- Trees are simple, but not very good in predicting.
- Improvements via bagging and random forests.
- Bagging: Build trees on bagged datasets and average over predictions.
- Random forests: Same as bagging, but only use a subset of all variables for each new split. Regression: $p/3$, classification $\sqrt{p}$ of all variables.
- Loss of interpretability is compensated by variable importance measures.

Motivation for Boosting in the context of trees

- Question: Could we address the shortcomings of single decision trees models in some other way than by using random forests?
- For example, rather than performing variance reduction on complex trees, can we decrease the bias by only using simple trees?
- A solution to this problem, making a good model from simple trees, is another class of ensemble methods called *boosting*.

Boosting is the process of iteratively adding basis functions in a greedy fashion so that each additional basis function further reduces the selected loss function.

Simple example 1: Boosting for a classification problem: AdaBoost

Chapter 10.1 in Hastie, Tibshirani, and Friedman (2009), including Fig 10.1 and Algorithm 10.1

- Assume we have a binary classification problem for $Y \in \{-1, 1\}$, a vector of predictor variables X, and a classifier $G(X)$.
- The error rate for a given training sample is then

$$\text{err} = \frac{1}{N} \sum_{i=1}^{N} I(y_i \neq G(x_i)) \ .$$

- Here, $G(X)$ is assumed to be a *weak classifier*, that is, its error rate is only *slightly better than a random guess.*
- $G(X)$ can for example be a tree which is only a "stump". But it can also be any other classifier.

- How would such a weak classifier ever give good predictions?

- Idea:
  - Sequentially apply the weak classifier on *modified versions* of the data, producing a *sequence* $G_m(x)$ of weak classifiers for $m = 1, 2, \ldots, M$.
  - At the end, use a weighted sum $G(x) = \text{sign}\left(\sum_{m=1}^{M} \alpha_m G_m(x)\right)$ to make the final prediction.

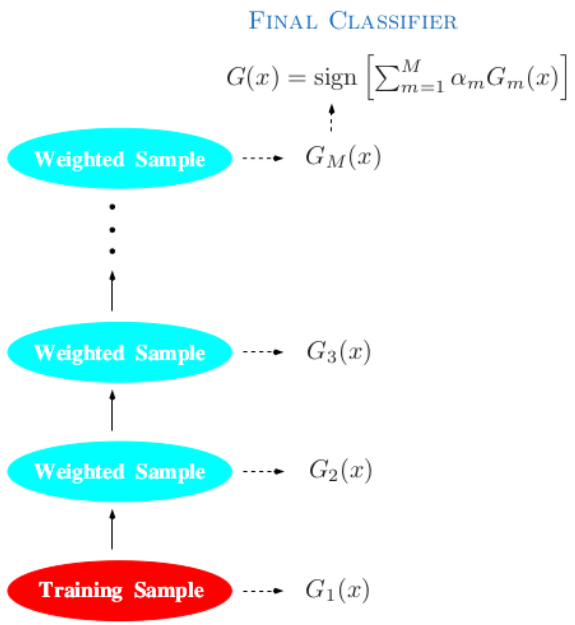


FIGURE 10.1. *Schematic of AdaBoost. Classifiers are trained on weighted versions of the dataset, and then combined to produce a final prediction.*

(Taken from Hastie, Tibshirani, and Friedman (2009))

How do we then find the sequence $G_m(x)$ and the corresponding weights α_m?

Algorithm 10.1 (Hastie, Tibshirani, and Friedman (2009)): AdaBosot.M1

1. Initialize the observation weights $w_i = 1/N$, $i = 1, 2, \ldots, N$.

2. For $m = 1, 2, ..., M$:
   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 $G(x) = \text{sign}\left[\sum_{m=1}^{M} \alpha_m G_m(x)\right]$.

Explanations:

- Weights α_m: assigns larger overall weight to good classifiers[1].
- Weights w_i modify the original data. This ensures that observations i with wrong classification in $G_{m-1}(x)$ obtain larger weights, namely

$$w_i \leftarrow w_i \cdot \exp(\alpha_m) \ , \qquad \text{if } y_i \neq G(x_i) \ ,$$
$$w_i \leftarrow w_i \ , \qquad \text{if } y_i = G(x_i) \ .$$

- The sum $\sum_{m=1}^{M} \alpha_m G_m(x)$ is a continuous number, so we take its sign to get a classification into -1 or 1.

[1] Note: When $\text{err}_m < 0.5$, then $\alpha_m > 0$. Otherwise, the classifier needs to be inverted and $\alpha_m < 0$ makes sense.

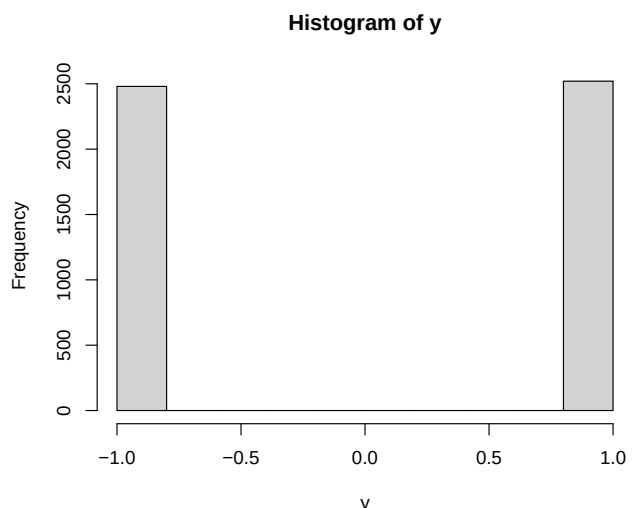
Example

Inspired by (10.2) in the Elements book:

- Generate features $X_1, \ldots, X_{10}$ multivariate Gaussian.
- Classify $y = 1$ if $\sum_{j=1}^{10} X_j^2 > 9.34$, $y = -1$ otherwise.
- 1000 training and 4000 test observations.

Yields roughly half/half of each category:

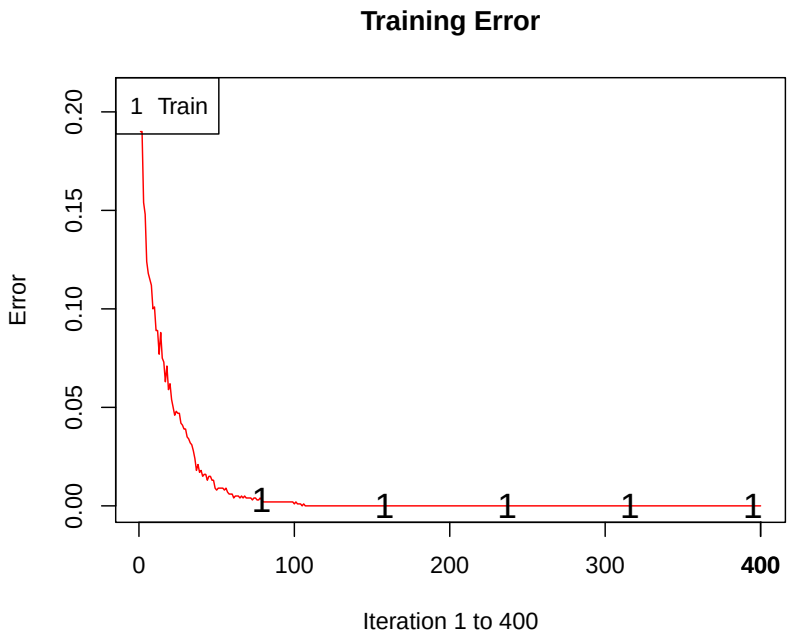
```
library(mvtnorm)
set.seed(123)
X <- rmvnorm(5000,rep(0,10))
y <- ifelse(rowSums(X^2)>9.34,1,-1)
hist(y,title="")
```



Fit AdaBoost on the training data using the function `ada()` from the `ada` R package. The function uses classification trees as weak learners. We use default choice for tree depth (tree depth is decided on a case-by-case basis):

```
library(ada)
dd <- data.frame(X,y)
dd.train <- dd[1:1000,]
dd.test <- dd[1001:5000,]

set.seed(4268)
r.ada <- ada(y~.,dd.train,iter=400,type="discrete", loss="ada", control=rpart.control())
plot(r.ada)
```



Error rates: AdaBoost vs random forest

AdaBoost test error:

```
ada.pred <- predict(r.ada,dd.test)
sum(dd.test$y!=ada.pred)/4000
```

```
## [1] 0.099
```

Random forest test error:

```
library(randomForest)
set.seed(123)
rf.boston=randomForest(as.factor(y)~.,dd.train,mtry=3,
                       ntree=1000)
rf.pred <- predict(rf.boston,newdata=dd.test)
sum(dd.test$y!=rf.pred)/4000
```

```
## [1] 0.17425
```

Simple example 2: Boosting regression trees

In tree boosting, trees are grown *sequentially* so that each tree is grown using information from the previous tree.

1. Build a decision tree with d splits (and $d + 1$ terminal notes).
2. Improve the model in areas where the model didn't perform well. This is done by fitting a decision tree to the *residuals of the model*. This procedure is called *learning slowly*.
3. The first decision tree is then updated based on the residual tree, but with a weight.

The procedure is repeated until some stopping criterion is reached. Each of the trees can be very small, with just a few terminal nodes (or just one split).

Algorithm 8.2: Boosting for regression trees

1. Set $\hat{f}(x) = 0$ and $r_i = y_i$ for all i in the training set.
2. For $b = 1, 2, ..., B$, repeat:
   a) Fit a tree $\hat{f}^b$ with d splits ($d + 1$ terminal nodes) to the training data (X, r).
   b) Update $\hat{f}$ by adding in a shrunken version of the new tree:

$$\hat{f}(x) \leftarrow \hat{f}(x) + \nu \hat{f}^b(x).$$

   c) Update the residuals,

$$r_i \leftarrow r_i - \nu \hat{f}^b(x_i).$$

3. The boosted model is $\hat{f}(x) = \sum_{b=1}^{B} \nu \hat{f}^b(x)$.

Boosting has three tuning parameters which need to be set (B, ν, d), and can be found using cross-validation.

Tuning parameters

- **Number of trees** B. Could be chosen using cross-validation. A too small value of B would imply that much information is unused (remember that boosting is a slow learner), whereas a too large value of B may lead to overfitting.
- **Shrinkage parameter** ν. Controls the rate at which boosting learns. ν scales the new information from the b-th tree, when added to the existing tree $\hat{f}$. A small value for ν ensures that the algorithm learns slowly, but will require a larger B. Typical values of ν is 0.1 or 0.01.
- **Interaction depth** d: The number of splits in each tree. This parameter controls the complexity of the boosted tree ensemble (level of interaction between variables). By choosing $d = 1$ a tree stump will be fitted at each step and this gives an additive model.

A very simple illustration of this idea can be found here:

https://arogozhnikov.github.io/2016/06/24/gradient_boosting_explained.html

Revisit the Boston data from module 8

We are now finally boosting the Boston trees! We use the `gbm()` function from the respective R package. We boost with 5000 trees and allow the interaction depth (number of splits per tree) to be of degree 4:

```
library(gbm)
set.seed(1)
boost.boston=gbm(medv~.,data=Boston[train,],
                 distribution="gaussian",
                 n.trees=5000,interaction.depth=4)
summary(boost.boston,plotit=FALSE)
```

```
##             var    rel.inf
## rm           rm 43.9919329
## lstat     lstat 33.1216941
## crim       crim  4.2604167
## dis         dis  4.0111090
## nox         nox  3.4353017
## black     black  2.8267554
## age         age  2.6113938
## ptratio ptratio  2.5403035
## tax         tax  1.4565654
## indus     indus  0.8008740
## rad         rad  0.6546400
## zn           zn  0.1446149
## chas       chas  0.1443986
```

Prediction on the test set

- Calculate the MSE on the test set, first for the model with $\nu = 0.001$ (default), then with $\nu = 0.2$.
- Ideally, we should do a cross-validation to find the best ν over a grid (comes later), but here it seems not to make a big difference.

```
yhat.boost=predict(boost.boston,newdata=Boston[-train,],n.trees=5000)
mean((yhat.boost-boston.test)^2)
```

```
## [1] 18.84709
```

```
boost.boston=gbm(medv~.,data=Boston[train,],distribution="gaussian",
                 n.trees=5000,interaction.depth=4,shrinkage=0.2,verbose=F)
yhat.boost=predict(boost.boston,newdata=Boston[-train,],n.trees=5000)
mean((yhat.boost-boston.test)^2)
```

```
## [1] 18.33455
```

Boosting more generally

- AdaBoost.M1 and the above regression tree example are both relatively simple special cases of (tree) boosting.
- We therefore step a bit back and look at boosting methods in more generality.
- In general terms, we are looking for expansions that take the form

$$f(x) = \sum_{m=1}^{M} \beta_m b(x; \gamma) \ ,$$

  where $b(x; \gamma)$ are (simple) *basis functions* in x characterized by a set of parameters γ, and β_m are the corresponding expansion coefficients.
- In the following, we focus on the case where the $b(x; \gamma)$ correspond to trees.

Boosting trees – what do we want to minimize?

Start by looking again at *a single tree*. We need to find regions R_j and values $f(x) = \gamma_j$ if $x \in R_j$, for $j = 1, 2, \ldots, J$. The tree can then be expressed as

$$T(x; \Theta) = \sum_{j=1}^{J} \gamma_j I(x \in R_j) \ ,$$

with parameters $\Theta = \{R_j, \gamma_j\}_1^J$. For a given division into regions, we want to minimize

$$\hat{\Theta} = \arg\min_{\Theta} \sum_{j=1}^{J} \sum_{x_i \in R_j} L(y_i, \gamma_i) \ , \tag{1}$$

for a set of observations (x_i, y_i) $(i = 1, \ldots, N)$ and for loss function $L()$, for example squared-error, Gini, deviance loss etc.

The optimization problem can thus be split into two parts:

1) **Find γ_j given R_j**: Given R_j, estimating γ_j is easy. Often it is $\gamma_j = \overline{y_j}$ or a majority vote for classification.

2) **Finding the regions R_j**: This is the hard part. Usually approximated via a greedy algorithm. Often, use smoother approximations of the optimization criterion

$$\tilde{\Theta} = \arg\min_{\Theta} \sum_{j=1}^{J} \sum_{x_i \in \tilde{R}_j} \tilde{L}(y_i, T(x_i, \Theta)) \ , \tag{2}$$

and then use $\hat{R}_j = \tilde{R}_j$ to find γ_j using criterion (1).

From single trees to boosting

- Previous slide summarizes how to find *one tree*.
- The result of boosting is a *sum of M trees*

$$f_M(x) = \sum_{m=1}^{M} T(x; \Theta_m) \ .$$

- Tree boosting is about *how to find each tree*.
- At each step, one must solve

$$\hat{\Theta}_m = \arg\min_{\Theta_m} \sum_{i=1}^{N} L\left(y_i, f_{m-1}(x_i) + T(x_i; \Theta_m)\right) \tag{3}$$

to find the next set Θ_m, given $f_{m-1}(x)$.

- For a *regression tree* with **squared-error loss**, solving (3) is the same as for a single tree. Just sequentially fit regression trees to the residuals $r_i = y_i - f(x_i)$ from the previous tree.

- For *binary classification* with $Y \in \{-1, 1\}$ and **exponential loss**

$$L(y, f(x)) = \exp(-yf(x)) \ ,$$

we get AdaBoost (Algorithm 10.1)[2].

[2]The equivalence of AdaBoost to forward stagewise additive modeling using exponential loss was only discovered five years after its invention!

- Both strategies on the previous slide are straightforward, but either *not very robust and general*.
- We therefore look for general algorithms that can use any loss function $L(f)$ when f is **contrained to be a sum of trees**.
- It is then sometimes better to approximate the loss function as $\tilde{L}(f)$ in the tree-building step (step (2) on slide 26) and use $L(f)$ only to determine the γ_{jm} values in each tree m.

Main idea: Iteratively build trees for the *gradient* of the previous tree. Motivated by steepest descent.

Steepest descent – a numerical optimization technique

The following considerations are not restricted to trees!

- *Steepest descent* is a *general* numerical optimization technique.
- When trying to find the best function $f(x)$ that minimizes a loss function

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

the direction of *steepest ascent* is given by an N-dimensional *gradient vector* $\mathbf{g}^\top = (g_1, g_2, \ldots, g_N)$ with entries

$$g_i = \left[\frac{\partial L(y_i, f(x_i))}{\partial f(x_i)} \right] \ .$$

- The direction of *steepest descent* is thus given by $-\mathbf{g}$.

- If we repeat this iteratively, we can denote the gradient at step m as $\mathbf{g}_m^\top = (g_{1m}, g_{2m}, \dots, g_{Nm})$.
- To update from $\mathbf{f}_{m-1}$ to $\mathbf{f}_m$[3], we search for the largest decrease in the loss function as

$$\rho_m = \arg\min_\rho L(\mathbf{f}_{m-1} - \rho \mathbf{g}_m) \ ,$$

- Then update $\mathbf{f}_m = \mathbf{f}_{m-1} - \rho_m \mathbf{g}_m$.

[3] $\mathbf{f}_m = (f_m(x_1), f_m(x_2), \dots, f_m(x_N))^\top$

Gradient Boosting: Steepest descent for trees

Recall that we want to fit trees

$$f_m(x) = T(x; \Theta_m) = \sum_{j=1}^{J} \gamma_j I(x \in R_j) \ ,$$

not arbitrary predictor functions $\mathbf{f}_m$.

The central idea: find a tree $T(x; \Theta_m)$ that is as close as possible to the negative gradient

$$\tilde{\Theta}_m = \arg\min_{\Theta} \sum_{i=1}^{N} (-g_{im} - T(x_i; \Theta))^2 \ . \tag{4}$$

That is: Fit a tree T to the negative gradient using least squares!

- This leads to regions $\tilde{R}_{jm}$ that are close (enough) to the optimal regions R_{jm} from equation (3) on slide (27).

  $\rightarrow$ This corresponds to the step to find the regions $\hat{R}_{jm} = \tilde{R}_{jm}$ (step 2 on slide 26).

- Finally, find γ_{jm} given the regions $\hat{R}_{jm}$.

  $\rightarrow$ This is the "easy" step 1 on slide 26 and is done by

$$\hat{\gamma}_{jm} = \arg\min_{\gamma} \sum_{x_i \in \hat{R}_{jm}} L(y_i, f_{m-1}(x_i) + \gamma) \ .$$

Gradient tree boosting algorithm

Algorithm 10.3 in Hastie, Tibshirani, and Friedman (2009):

1. Initialize $f_0(x) = \arg\min_\gamma \sum_{i=1}^{N} L(y_i, \gamma)$.
2. For $m = 1$ to M:
   (a) For $i = 1, 2, \ldots, N$ compute

$$r_{im} = -\left[\frac{\partial L(y_i, f(x_i))}{\partial f(x_i)}\right] .$$

   (b) Fit a regression tree to the targets r_{im}, giving terminal regions $R_{jm}, j = 1, 2, \ldots, J_m$.
   (c) For $j = 1, 2, \ldots, J_m$ compute

$$\gamma_{jm} = \arg\min_\gamma \sum_{x_i \in R_{jm}} L(y_i, f_{m-1}(x_i) + \gamma) .$$

   (d) Update $f_m(x) = f_{m-1}(x) + \sum_{j=1}^{J_m} \gamma_{jm} I(x \in R_{jm})$.
3. Output $\hat{f}(x) = f_M(x)$.

Gradient tree boosting for regression

$r_{im} = -\left[\frac{\partial L(y_i, f(x_i))}{\partial f(x_i)}\right]$ are the components of the negative gradient.
We call them *generalized* or *pseudo-residuals*. Why?

- Look at the (scaled) **quadratic loss function**

$$L(y_i, f(x_i)) = \frac{1}{2}(y_i - f(x_i))^2.$$

- Then

$$-\partial L(y_i, f(x_i))/\partial f(x_i) = y_i - f(x_i) \ ,$$

  which is the residual.
- Gradient boosting is thus equivalent to reducing the quadratic loss function as fast as possible ("steepest descent").
- However: Gradient boosting works for *any loss function*.

Gradient tree boosting for regression – loss functions

- **Quadratic loss**: Previous slide. Not very robust – a lot of weight on extreme observations.

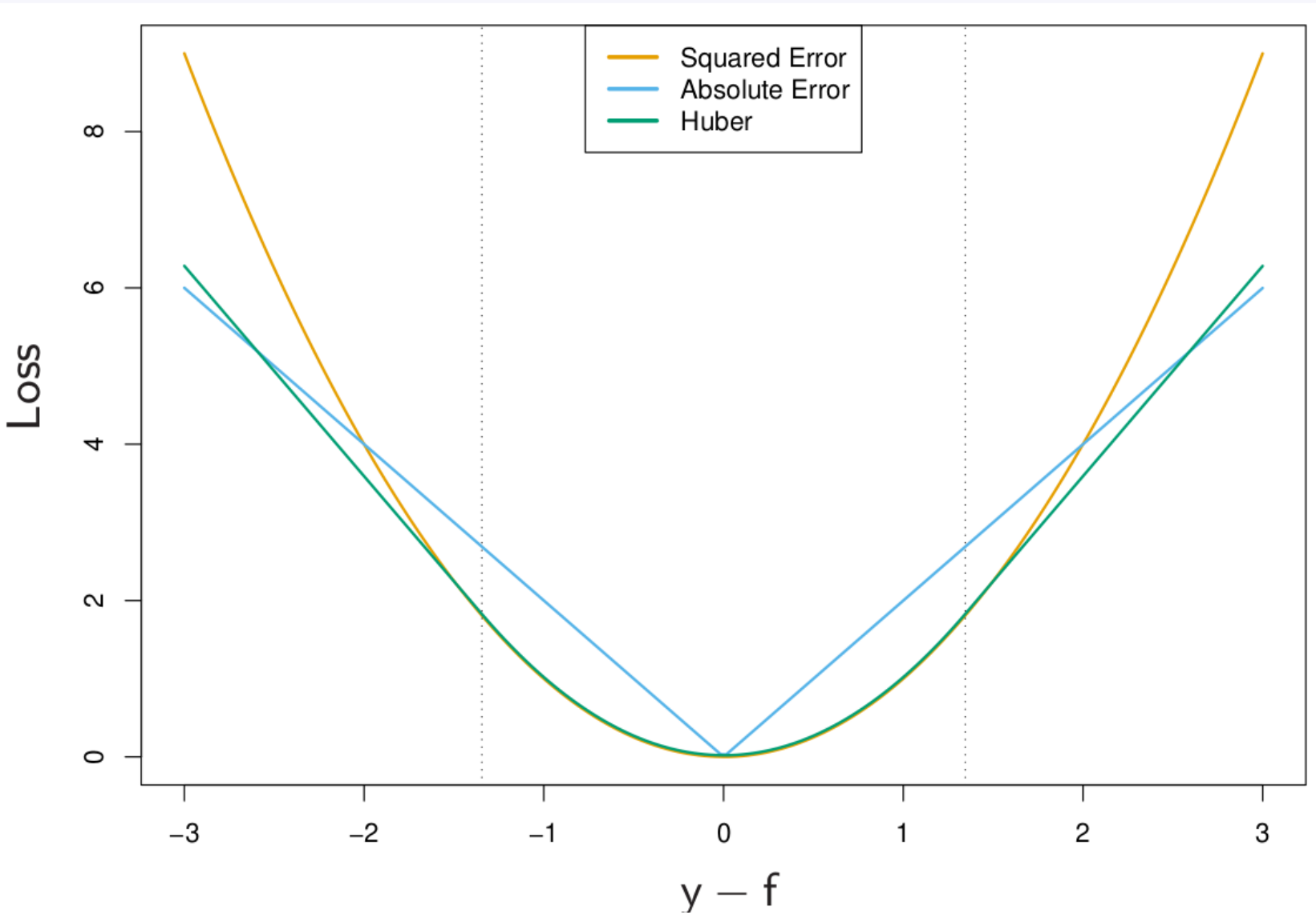
Popular alternatives:

- **Absolute loss**:

$$L(y, f(x)) = |y - f(x)|$$

- **Huber loss**:

$$L(y, f(x)) = \begin{cases} (y - f(x))^2 & \text{for } |y - f(x)| \leq \delta \ , \\ 2\delta|y - f(x)| - \delta^2 & \text{otherwise.} \end{cases}$$


Squared Error
Absolute Error
Huber
Loss
0
2
4
6
8
−3
−2
−1
0
1
2
3
y − f

Gradient tree boosting for binary classification

Replace the squared error loss by the *binomial deviance* (now with $y \in \{0, 1\}$)[4]:

$$L(y, f(x)) = -I(y = 1) \log(p(x)) - I(y = 0) \log(1 - p(x)) \ ,$$

with

$$p(x) = \frac{e^{f(x)}}{1 + e^{f(x)}} \ .$$

Here, $f(x)$ is the (linear) predictor function encoding the tree ensemble, similar to logistic regression.

[4]Note that the deviance is $-2 \log(\mathcal{L}(y; x, \Theta))$, but we ignore the factor 2.

Gradient tree boosting for classification into K classes

For $K > 2$ classes, use as loss function the *K-class multinomial deviance*

$$L(y, f(x)) = -\sum_{k=1}^{K} I(y = k) \log(p_k(x)) \tag{5}$$

$$= -\sum_{k=1}^{K} I(y = k) f_k(x) + \log(\sum_{l=1}^{K} e^{f_l(x)}) \ , \tag{6}$$

with class probabilities

$$p_k(x) = \frac{e^{f_k(x)}}{\sum_{l=1}^{K} e^{f_l(x)}} \ ,$$

and corresponding functions $f(x)$.

- Plugging $p_k(x)$ into the multinomial deviance, we see that

$$-g_{ikm} = \left[\frac{\partial L(y_i, f_1(x_i), \ldots, f_K(x_i))}{\partial f_k(x_i)}\right]_{f_{m-1}(x_i)} = I(y_i = k) - p_k(x_i) \ .$$

- For K classes, we must build K trees in each iteration of step 2 in the boosting procedure!
- Each tree T_{km} is fit to the gradient $\mathbf{g}_{km}$.
- In step 3, we aggregate the class probabilities for each class k.

Practical considerations

The three main ingredients of gradient boosting, which we need to determine in practice:

- *Weak Learner*: A weak learner is one that classifies our data but does poorly, perhaps no better than random guessing. In other words, it has a high error rate. These are typically decision trees (also called decision stumps, because they are less complicated than typical decision trees).
- *Loss Function*: The function we want to minimize. Its role is to estimate how good the model is at making predictions with the given data.
- *Additive Model*: This is the iterative and sequential approach of adding the trees (weak learners) one step at a time. After each iteration, we need to be closer to our final model, since each iteration should reduce the value of our loss function.

Parameter tuning

Given a loss function, we can tune the following parameters:

Tree hyperparameters (weak learner):

a) Tree size (depth) in each iteration

b) Minimum observations in terminal nodes

Boosting hyperparameters (additive model):

c) Number of trees M, that is, the number of boosting iterations

d) The learning rate ν

a) Finding the right tree depth in boosting

(10.11 in the Elements book)

- Historically, large trees were grown, followed by pruning, in each iteration of the boosting process.
- However, this assumes that each tree is the last one in the algorithm (why?).
- Better strategy: All trees are the same size.

a) Finding the right tree depth in boosting

(10.11 in the Elements book)

- Historically, large trees were grown, followed by pruning, in each iteration of the boosting process.
- However, this assumes that each tree is the last one in the algorithm (why?).
- Better strategy: All trees are the same size.

Given that J is the number of leaves ($J-$terminal node tree):

- **which size should we then choose?**
- **Why?**

Discuss!

Some considerations / drawings:

In the end, J is a tuning parameters, but does usually not give much better results than when using $J \approx 4$ to 6.

b) Minimum number of observations in terminal nodes

- This hyper-parameter is less relevant for large data sets.
- Moreover, by fixing tree depth and tuning the number of trees, we can hope to get this parameter "for free".
- A reason to care anyway is that larger number of observations per leaf are needed to get a good mean estimates.

c) The number of trees (regularization strategy 1)

- When the number of trees M is growing, the training error is reduced.
- For too large M, we will overfit the training data.
- Therefore, need to monitor prediction error on a validation sample and do "early stopping".
- Alternative: Monitor prediction error with cross-validation

$\rightarrow$ M is a tuning parameter and there is an optimal M.

d) The learning rate (regularization strategy 2)

- The contribution of each tree is scaled with a learning rate ν:

$$f_m(x) = f_{m-1}(x) + \nu \sum_{j=1}^{J} \gamma_{jm} I(x \in R_{jm}) \ .$$

- ν and M are connected: Smaller ν requires larger M, and vice versa.
- Empirically, it was found that small ν (*i.e.*, $\nu < 0.1$) is more robust.
- Possible strategy: Fix ν and then choose M by early stopping.

Application: Ames housing data

Inspired by Boehmke and Greenwell (2020)

- The Ames housing data is a data set available from the `AmesHousing` R package[5].
- Response: House prices, with 80 predictor variables (age, style, condition, area,..).
- 2930 observations
- The dataset is one for a continuously running competition on Kaggle (for newcomers): https://www.kaggle.com/c/house-prices-advanced-regression-techniques/data

[5] https://cran.r-project.org/web/packages/AmesHousing/index.html

```r
library(AmesHousing) # contains the data
library(gbm) # for original implementation of regular and stochastic GBMs
library(rsample) # to subsample training and test sets

# Data preparation
ames <- AmesHousing::make_ames()

# Stratified sampling with the rsample package
set.seed(123)
split <- initial_split(ames, prop = 0.7,
                       strata = "Sale_Price")
# Training and testing data:
ames_train  <- training(split)
ames_test   <- testing(split)

response <- "Sale_Price"
predictors <- setdiff(colnames(ames_train), response)
```

```r
# run a basic GBM model
set.seed(123)  # for reproducibility
ames_gbm1 <- gbm(
  formula = Sale_Price ~ .,
  data = ames_train,
  distribution = "gaussian",  # SSE loss function
  n.trees = 3000,
  shrinkage = 0.1, # learning rate
  interaction.depth = 3,
  n.minobsinnode = 10, # minimual number of observations in terminal nodes
  cv.folds = 10,
  bag.fraction =1 # The default is 0.5, but here we want to have all data in each iteration
)
```

Find index for number trees with minimum CV error and then calculate RMSE for it:

```r
(best <- which.min(ames_gbm1$cv.error))
```

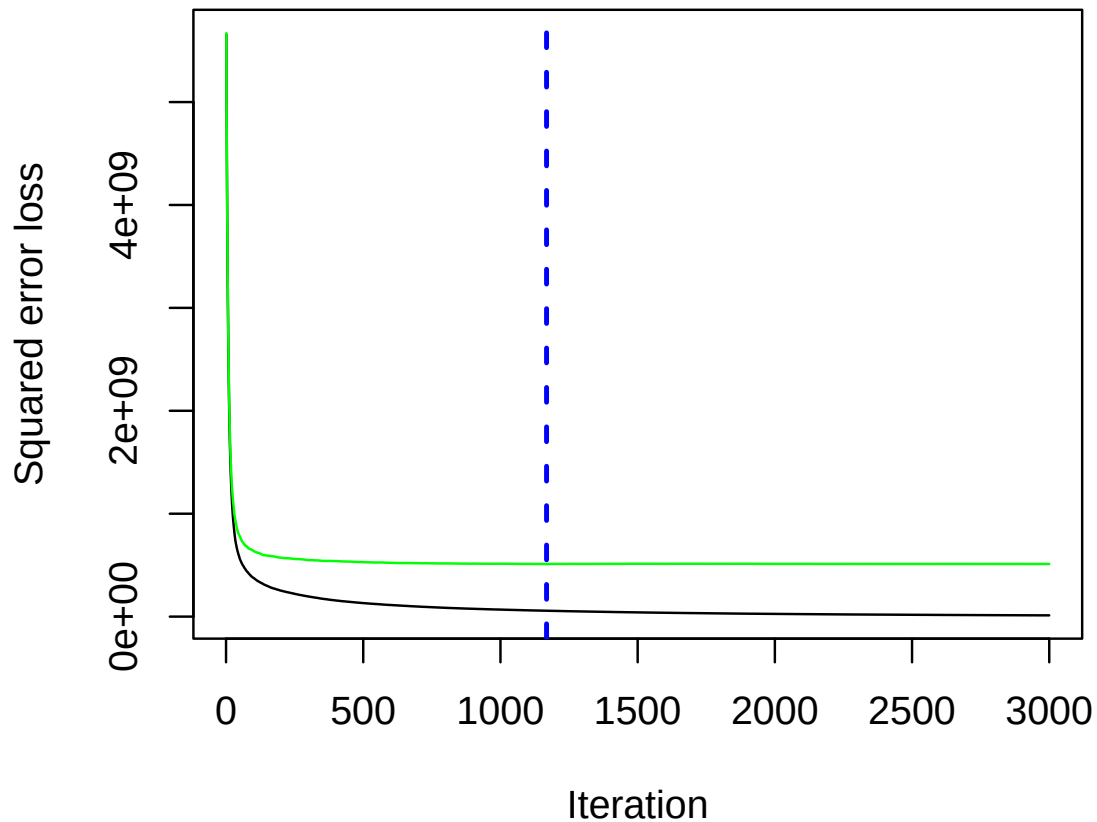
```
## [1] 1168
```

```r
sqrt(ames_gbm1$cv.error[best])
```

```
## [1] 22600.12
```

Plot error curve with training black and cross-validated MSE (green):

```
gbm.perf(ames_gbm1, method = "cv")
```




```
## [1] 1168
```

Stochastic GBMs (regularization strategy 3)

See Boehmke and Greenwell (2020)

- Stochastic gradient boosting: choose a random subsample of the training data for building each tree (J. H. Friedman 2002).
- Typical value for the proportion η is $N/2$, but can be much smaller when N is large.
- Again, the idea is to *reduce variance*.
- In addition: Reduction of computation time!

$\rightarrow$ The proportion η is *another hyper-parameter* that can be tuned...

Variants of stochastic GBMs

There are a few variants of stochastic gradient boosting that can be used, all of which have additional hyperparameters:

- Subsample rows before creating each tree
- Subsample columns before creating each tree
- Subsample columns before considering each split in each tree

Here we only look at the case where we subsample rows (*i.e.*, entire data samples), which is available in the `gbm()` package.

We use `bag.fraction=0.5` to subsample 50% of the rows each time we build a tree:

```
set.seed(123)  # for reproducibility
ames_gbm2 <- gbm(
  formula = Sale_Price ~ .,
  data = ames_train,
  distribution = "gaussian",  # SSE loss function
  n.trees = 3000,
  shrinkage = 0.1, # learning rate
  interaction.depth = 3,
  n.minobsinnode = 10, # minimual number of observations in terminal nodes
  cv.folds = 10,
  bag.fraction=0.5
)
```

Same as before:

```
(best2 <- which.min(ames_gbm2$cv.error))
```

```
## [1] 1119
```

```
sqrt(ames_gbm2$cv.error[best2])
```

```
## [1] 22402.07
```

XGBoost: Extreme Gradient Boosting

Suggested by Chen and Guestrin (2016) and implemented in the R package `xgboost`.

- XGBoost is the state-of-the-art for tree boosting and the basis for many competition-wining approaches, see for example Kaggle competitions (`https://www.kaggle.com/`).
- It combines many tricks into a very efficient toolbox: Parallelization, smart approximations, shrinkage, sub-sampling, etc.

Core ideas of XGBoost:

- *Second-order gradients*: Replace pure first order gradient boosting by a second-order (Taylor series) expansion.
- *Parallelization*: The model is implemented to train with multiple CPU cores. Finding new branches is the part that is parallelized. Different cores search in different areas of the data.
- *Approximation* of the greedy algorithm for each tree by not checking each possible split.
- *Regularization*: XGBoost includes different regularization penalties to avoid overfitting. Each new tree is regularized with two terms: The number of leaves (pruning) and the weights on the leaves. The total penalizing term is typically given by[6]

$$\Omega(f) = \gamma|T| + \frac{1}{2}\lambda||w||^2 \text{ (L2 regularization)}$$

or

$$\Omega(f) = \gamma|T| + \alpha||w|| \text{ (L1 regularization) .}$$

[6]The weights w correspond to the predicted values or probabilities for each observation.

Some words on regularization in XGBoost

Previous slide:

- Pruning parameter γ: the new tree is grown to the specified depth, and then pruned to find and remove splits according to the contraint imposed by γ.
- λ and α: Constrain model complexity in the same way as ridge or Lasso (L2 and L1).

Additional regularization:

- The learning rate ν.
- *Dropout*: Randomly drop trees to avoid that the first trees dominate the result.

All these lead to additional hyperparameters.

Ames house pricing example continued

We apply XGBoost via the **xgboost** library on the same dataset as before. Start by some data preparation:

```
library(xgboost)
library(recipes)

# Training data and response
xgb_prep <- recipe(Sale_Price ~ ., data = ames_train) %>%
  step_integer(all_nominal()) %>%
  prep(training = ames_train, retain = TRUE) %>%
  juice()

X <- as.matrix(xgb_prep[setdiff(names(xgb_prep), "Sale_Price")])
Y <- xgb_prep$Sale_Price

# Test data and test response
xgb_prep_test <- recipe(Sale_Price ~ ., data = ames_test) %>%
  step_integer(all_nominal()) %>%
  prep(training = ames_test, retain = TRUE) %>%
  juice()
X_test <- as.matrix(xgb_prep_test[setdiff(names(xgb_prep_test), "Sale_Price")])
Y_test <- xgb_prep_test$Sale_Price
```

Run on training data with a set of hyperparameters that were pre-optimized[7]

```
set.seed(123)
ames_xgb <- xgboost(
  data = X,
  label = Y,
  nrounds = 6000,
  objective = "reg:squarederror",
  early_stopping_rounds = 50,

  params = list(
    eta = 0.1, # learning rate (we call it $\nu$)
    lambda=0.01, # L2 regularization term
    max_depth = 3,
    min_child_weight = 3,
    subsample = 0.8, # Proportion of datapoints used for each tree
    colsample_bytree = 0.5, # subsample ratio of columns when constructing one tree
    nthread=12),
  verbose = 0
)
```

```
X.pred.test <- predict(ames_xgb,newdata = X_test)
sqrt(sum((X.pred.test - Y_test)^2)/nrow(X_test))
```

```
## [1] 19935.63
```

[7] Adapted from https://bradleyboehmke.github.io/HOML/gbm.html

Hyperparameter tuning

- Hyperparameters in many dimensions.
- Tuning the models becomes very slow! We refer to Boehmke and Greenwell (2020) for parameter tuning runs: https://bradleyboehmke.github.io/HOML/gbm.html
- See exercise session.

Most recent advances: LightBGM, CatBoost

From https://bradleyboehmke.github.io/HOML/gbm.html:

- LightGBM (Ke et al. 2017) is a gradient boosting framework that focuses on leaf-wise tree growth and smart ways to subsample the data (keep only instances with large gradients). Speeds up conventional gradient boosting by up to a factor of 20.
- CatBoost (Dorogush, Ershov, and Gulin 2018) is another gradient boosting framework that focuses on using efficient methods for encoding categorical features during the gradient boosting process.

Both frameworks are available in R (packages `lightgbm` and `catboost`).

Interpretation of tree ensembles

- Single trees are easy to interpret.
- Tree ensembles or linear combinations of trees (like in boosting) sacrifice interpretability.
- In Module 8 we have heard about *relative importance of predictor variables*.
- Another way to gain interpretability back is via *partial dependence plots*. They show the effect of individual predictors, where the effect of the other predictor variables is integrated out (see Hastie, Tibshirani, and Friedman (2009), Section 10.13.2).

Partial dependence plots

(Section 10.13.2 in the Elements book Hastie, Tibshirani, and Friedman (2009))

- Motivation: We would like to have a *qualitative description* of how a single variable affects the response.
- Partial dependence plots provide such a description. They can be used to interpret results from any "black box" method.
- For input variable X_j, the *partial dependence* of $f(X)$ on X_j is given as

$$\mathrm{E}_{X_j} f(X_j, X_{-j}) \ ,$$

  where X_{-j} denotes the set of all variables *except* X_j.

- For boosted trees, we estimate the partial dependence by taking the average over all observations

$$\overline{f}(X_j) = \frac{1}{N} \sum_{i=1}^{N} f(X_j, (X_{-j})_i)$$

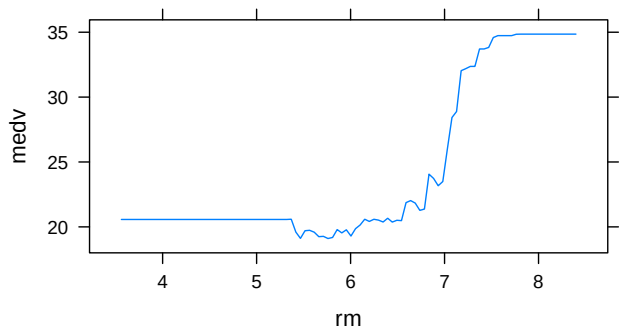
  for a set of values for X_j, where we evaluate the estimated tree value $f(X_j, (X_{-j})_i)$ for each data point i.

- Note: The partial dependence function represents the effects of X_j on $f(X)$ *after* accounting for the effects of the other variables. They are not the effects of X_j on $f(X)$ independent of the other variables!

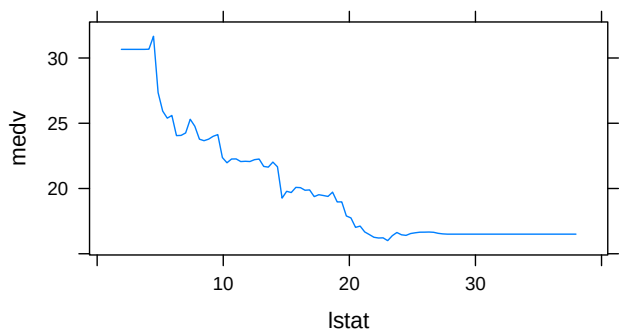
Partial dependence plots for the Boston example

`rm` (number of rooms) and `lstat` (% of lower status population) are the most important predictors.

```
plot(boost.boston,i="rm",ylab="medv")
```




```
plot(boost.boston,i="lstat",ylab="medv")
```



References

Boehmke, B., and B. Greenwell. 2020. *Hands-on Machine Learning with R.* https://bradleyboehmke.github.io/HOML/: CRC press.

Chen, Tianqi, and Carlos Guestrin. 2016. "Xgboost: A Scalable Tree Boosting System." In *Proceedings of the 22nd Acm Sigkdd International Conference on Knowledge Discovery and Data Mining*, 785–94.

Freund, Yoav, and Robert E Schapire. 1997. "A Decision-Theoretic Generalization of on-Line Learning and an Application to Boosting." *Journal of Computer and System Sciences* 55 (1): 119–39.

Friedman, Jerome H. 2002. "Stochastic Gradient Boosting." *Computational Statistics & Data Analysis* 38: 367–78.

Friedman, Jerome, Trevor Hastie, and Robert Tibshirani. 2000. "Additive Logistic Regression: A Statistical View of Boosting (with Discussion and a Rejoinder by the Authors)." *The Annals of Statistics* 28 (2): 337–407.

Hastie, Trevor, Robert Tibshirani, and Jerome Friedman. 2009. *The Elements of Statistical Learning.* Vol. 2. Springer series in statistics New York.

James, Gareth, Daniela Witten, Trevor Hastie, and Robert Tibshirani. 2021. *An Introduction to Statistical Learning.* 2nd ed. Vol. 112. Springer.

Ridgeway, Greg. 1999. "The State of Boosting." *Computing Science and Statistics*, 172–81.