Intro — Introduction to machine learning and ensemble decision trees

Description Remarks and examples References Also see

Description

Machine learning methods are commonly used to solve various research and business problems. These methods can be used to predict the probability of a patient having a disease based on their symptoms, forecast customer churn for the coming year, determine whether a customer is likely to default on a loan based on their background characteristics, predict changes in house prices in the coming month, and identify important factors in predicting the outcome of an election. And these are just a few examples. These types of problems often require more sophisticated modeling approaches than, for instance, a linear regression or generalized linear models. Ensemble decision tree methods, which combine multiple decision trees to improve model predictive performance, have emerged as some of the more popular and more effective methods for solving such problems because they perform well in practice (Shmuel, Glickman, and Lazebnik 2024; Shwartz-Ziv and Armon 2022; and Borisov et al. 2024).

This entry provides a software-free introduction to ensemble decision tree methods. In particular, we focus on two popular methods: gradient boosting machine (GBM) and random forest. See [H2OML] **h2oml** for the Stata implementation.

Remarks and examples

Remarks are presented under the following headings:

Why machine learning?
Preliminaries
Fundamentals of machine learning
Decision trees
  Classification trees
  Regression trees
  Pros and cons of decision trees
Ensemble methods
  Bagging
  Random forest
  Boosting
  GBM
  Trees with monotonicity constraints
Model selection in machine learning
  Three-way and two-way holdout methods
  k-fold cross-validation
  Hyperparameter tuning
  Method comparison
Interpretation and explanation
  Global surrogate models

Why machine learning?

Linear and generalized linear models are among the most widely used models in various fields. However, they may not always capture more complex patterns in the data well and thus may lead to poor prediction. As an example, consider a fictional dataset used to predict employee attrition based on salary and performance. Figure 1 provides the scatterplot of the data, with blue dots representing employees who stayed with the company and red dots representing those who left.

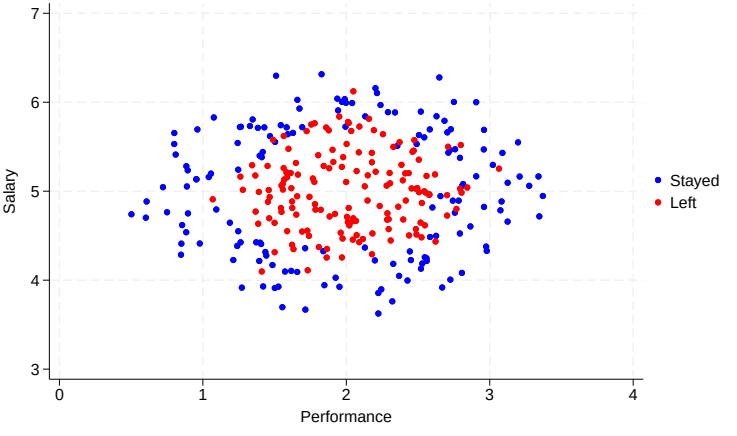

Figure 1.

The data-generating mechanism is complex, and there is no one line that can separate the blue and red dots. That is, the dataset is not linearly separable. To illustrate this point further, figure 2 shows the decision surface, the predicted attrition based on performance and salary, for the logistic regression. It predicts that an employee will leave (`attrition = 1`) for observations on the orange surface and that an employee will stay for observations on the light-blue surface. As we can see, the linear decision boundary misclassifies many blue dots as red and vice versa.

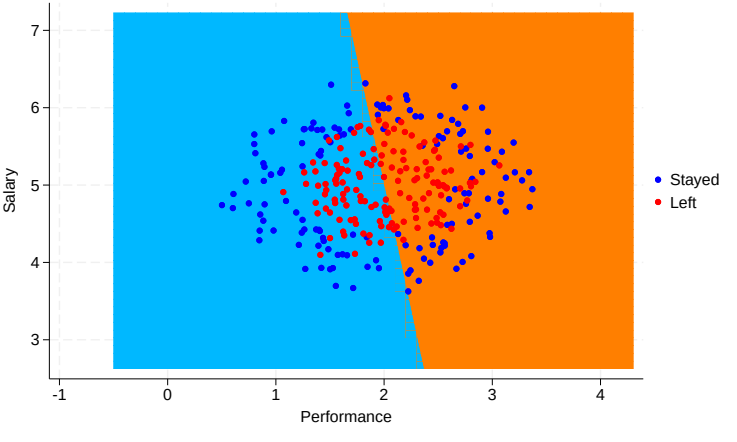

Figure 2. Logistic regression decision surface

On the other hand, machine learning methods can capture the complex structure better. Figure 3 displays the decision surface for the random forest. Here we can easily see that the random forest performs much better, with predictions more closely matching the observed attrition values.

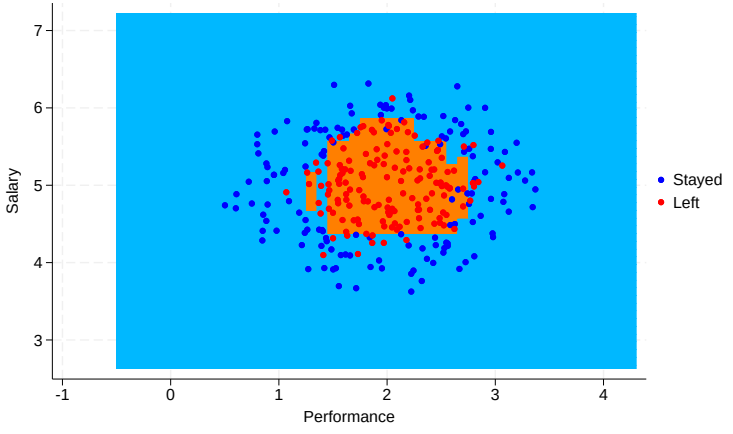

Figure 3. Random forest decision surface

Preliminaries

Before describing ensemble decision trees, we introduce the machine learning terminology that we will use throughout this manual.

Predictors. The inputs for a machine learning model. In classical statistics, these may be referred to as independent variables, covariates, x variables, or predictors. In the machine learning literature, they are also referred to as features.

Responses. The outputs for a machine learning model. In classical statistics, these may be referred to as dependent variables, y variables, or outcomes. In the machine learning literature, they are also referred to as targets.

Learning, **training**. In the machine learning context, learning refers to the process when a model uses data to adjust its parameters to increase prediction accuracy.

Learner. A model that is used for learning.

Supervised learning. A type of machine learning in which a method is trained on data where there is an associated response for each observation.

Unsupervised learning. A type of machine learning where there is no response variable.

Hyperparameter. A parameter whose value is adjusted to control and improve a training process.

Tuning. A process where the hyperparameters of a model are optimized to improve model performance.

Training data. A subset of the data that a model uses to learn.

Validation data. A subset of the data used to evaluate model performance during training as hyperparameters change.

Testing data. A subset of the data that is used to evaluate the performance of a trained model.

Performance metric. A quantitative measure or metric used to evaluate model performance.

Hyperparameter space. Possible values and ranges of the hyperparameters.

Grid search. A process of evaluating different hyperparameter configurations in the hyperparameter space to find the best configuration that improves model performance.

Generalization. A concept that a model performs well not only on the training data but also on the new (testing) data.

Generalization error, **test error**. A quantitative measure of how well a machine learning model can predict responses for new (testing) data.

Overfitting. Fitting a model too well to the training data.

Metric scoring. A process of evaluating the performance of a machine learning method by using a specified performance metric.

In a typical machine learning scenario, the goal is to predict a response based on a set of predictors. To achieve this goal, a researcher uses training data to build (or train) a prediction model. A good model, or learner, is one that accurately predicts the response for new or testing data and minimizes a generalization error or test error. A generalization error of a learning model is a quantitative measure of how well a machine learning model can predict responses for new data or, more formally, an expected error on any testing data sampled from the data-generating distribution. In other words, the focus is on predictive modeling, which is the process of "developing a mathematical tool or model that generates accurate prediction" (Kuhn and Johnson 2013). Intuitively, success in predictive modeling depends on finding a model that 1) has low generalization error, 2) is simple, and 3) can be used on a sufficiently large training dataset.

Most machine learning problems can be divided into two categories: supervised learning and unsupervised learning. In supervised learning, there is an associated response for each observation of the predictors. Most types of regression and many tree-based methods are examples of supervised learning. In contrast, in unsupervised learning, there is no response variable, and only the predictors are observed. Cluster analysis is an example of unsupervised learning.

In what follows, we provide a more technical introduction to machine learning, including decision trees and ensemble decision trees. For a brief and more gentle exposition of a machine learning workflow by using the `h2oml` command, see *h2oml in a nutshell* in [H2OML] **h2oml**.

Fundamentals of machine learning

One of the main issues in machine learning, also known as a fundamental problem of machine learning (Chollet 2021), is balancing learning and generalization. Recall that learning refers to the process of adjusting a model to achieve the best performance on the training data, whereas generalization refers to evaluating the performance of the model on the data it has never seen before such as the testing data. Unfortunately, generalization cannot be fully controlled by a researcher because we observe only the training data, and overfitting (fitting a model too well on the training data) can hurt the generalization of the model. This is why it is important to "mimic" the presence of testing data by splitting the observed training data, as we discuss in *Three-way and two-way holdout methods*.

The tradeoff between learning and generalization is related to the well-known bias–variance tradeoff, where the aim is to lower the generalization error by reducing the bias and variance of the proposed method. Suppose we have a supervised learning problem, where the relationship between predictors and the response is described by some unknown function $f(\cdot)$ plus an additive error,

$$y_i = f(\mathbf{x}_i) + \varepsilon_i \qquad i = 1, 2, \ldots, n$$

where $E(\varepsilon_i) = 0$ and $\text{Var}(\varepsilon_i) = \sigma^2$.

The goal is to estimate $f(\cdot)$ by $\hat{f}(\cdot)$ using a specific machine learning method on training data. However, if we use different training data, the learned $\hat{f}(\cdot)$ is likely to be different. The amount by which $\hat{f}(\cdot)$ changes as we use different training data is the variance. Machine learning methods, like other statistical estimation methods, often introduce bias because they typically impose simplifying assumptions during the estimation of $f(\cdot)$.

The generalization error for training data $D = \{(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \ldots, (\mathbf{x}_n, y_n)\}$ and test observation $(\mathbf{x}, y)$, sampled from the data-generating distribution, can be written as the sum of the error variance and the squared bias and the variance of the estimate:

$$E_{(\mathbf{x},y,D)}\left[\left\{\hat{f}(\mathbf{x}) - f(\mathbf{x})\right\}^2\right] = \sigma^2 + \text{Bias}^2\{\hat{f}(\mathbf{x})\} + \text{Var}\{\hat{f}(\mathbf{x})\}$$

The error variance σ^2 is inherited from the data and cannot be reduced. However, the bias, which is the average difference between $\hat{f}(\cdot)$ and $f(\cdot)$, is a result of underfitting and can be reduced. And the variance, which is inextricably linked to overfitting, where the model fits the training data too well and thus the variance of the model increases for new data, can also be reduced. Thus, an ideal machine learning method reduces the bias without increasing the variance or reduces the variance without increasing the bias. In practice, decreasing one will necessarily increase the other, so the preferred method strives to achieve the best tradeoff between the bias and the variance.

Consider a hypothetical example below that shows two methods, `Method 1` and `Method 2`. The red points correspond to the training data and blue points to the testing data. From the left graph, `Method 2` predicts the training points very well with possibly small bias and mean squared error (MSE). However, compared with `Method 1`, the prediction of `Method 2` deteriorates on the testing data because of the high variance. `Method 2` predicts the testing data poorly because it overfits the training data.

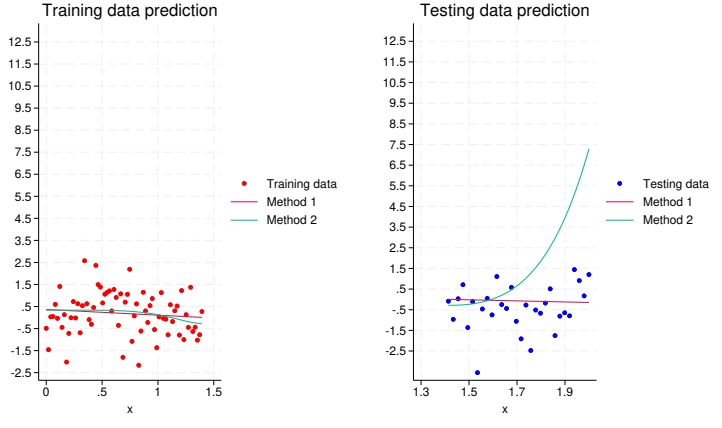

Figure 4.

The above example demonstrated the generalization of machine learning methods in just one dimension. In general, the ability of these methods, such as ensemble decision trees, to generalize well to high-dimensional data can be explained by the so-called manifold hypothesis (Chollet 2021; Wyner et al. 2017 ; and Belkin et al. 2019). According to this hypothesis, the observed high-dimensional data can be approximated by a low-dimensional manifold, or subspace. Informally, this means that a complex structure of the high-dimensional data can be represented by a simpler, lower-dimensional structure, which machine learning methods tend to capture well.

Decision trees

Decision trees are versatile and powerful supervised machine learning methods that can be used for both regression and classification. Decision trees repeatedly partition the data based on values of the predictors by asking a series of Boolean-type ("yes" or "no") questions. For each question, the data are partitioned into two branches such that the response observations in each branch are more homogeneous. Then a simple regression model is fit to each partition. Such repeated partitioning creates a treelike structure with the branches based on the values of the predictors. Some popular methods for building decision trees are CART (Breiman et al. 1984) and C4.5 (Quinlan 1993).

The hierarchical structure of a tree is inherently designed to capture and represent the interactions between predictors. Decision trees are insensitive to outliers and can easily handle missing data in predictors. In practice, decision trees are grown using greedy-type methods that make locally optimal splits at each step, instead of finding the globally optimal tree. Even though this can potentially lead to suboptimal trees, decision trees are effective in many applications. Decision trees are fast to train and can handle high-dimensional data with many predictors. They are also easy to interpret and visualize, making them a popular choice for many machine learning tasks. Decision trees have been widely used in scientific fields such as biomedicine, genetics, and marketing, among many other fields.

We first focus on introducing decision trees for classification, where the dependent variable is categorical. Then we describe decision trees for regression, where the dependent variable is continuous.

Classification trees

To motivate the concept of a decision tree, we consider a toy dataset where the goal is to predict whether a mushroom is edible or poisonous, coded as `e` and `p`, respectively, based on two predictors: cap diameter and season. The cap diameter is a continuous variable and season is categorical, where `s` and `w` denote summer and winter, respectively.

	capdiam	season	class
1.	7.3	s	e
2.	7.68	s	e
3.	8.4	s	e
4.	8.86	w	p
5.	9.03	s	e
6.	9.1	s	e
7.	9.59	w	p
8.	9.59	s	e
9.	10.42	w	e
10.	10.5	s	e
11.	12.85	s	e
12.	13.55	w	p
13.	14.07	w	p
14.	14.17	s	p
15.	14.64	s	p
16.	14.85	s	p
17.	14.86	s	p
18.	15.26	w	p
19.	15.34	s	p
20.	16.6	w	p

Based on the training data, a classification tree learns an ordered sequence of questions, where the answer to each question in the sequence affects the type of question asked in the next step. The tree diagram below shows the decision tree for our toy example. The method starts at the top of the tree, called the root node, and uses the entire training dataset. In this example, the root node splits the dataset into two parts based on the cap diameter predictor. By convention, the "yes" answer to the question at the node splits to the left, and the "no" answer splits to the right. A node is a subset of predictors. It can be classified as a terminal or nonterminal. A nonterminal node or parent node splits the data into two regions using the predictor that results in the best fit. (We will describe later how such a predictor is selected.) A terminal node or leaf node does not split the data further.

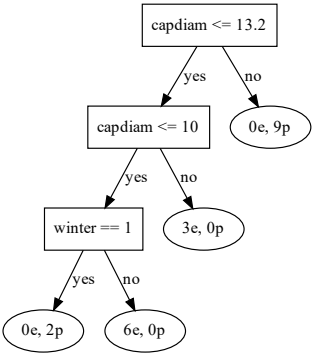

Figure 5.

For example, at the root node, the best split occurs for the predictor $x_i = \texttt{capdiam}$ at the split point $t_1 = 13.2$. This split partitions the data into the $\{\mathbf{x}|x_i \leq t_1\}$ and $\{\mathbf{x}|x_i > t_1\}$ regions. Throughout this entry, we will denote the split points by t_s, where s denotes the number of the split, counted from top to bottom and left to right on the above tree. The partition of the predictor space continues recursively until some stopping criterion is applied or there are no more splits. The set of all terminal nodes is called a partition of the data. Each observation from the training data falls into one of the terminal nodes.

Below, we show the partition of the predictor space into the regions that correspond to the above tree diagram. The red and yellow vertical lines correspond to the $\texttt{capdiam} \leq 13.2$ and $\texttt{capdiam} \leq 10$ conditions, and the horizontal line depicts the $\texttt{winter} = 1$ condition. The green and blue dots correspond to the observations with classes `p` and `e`, respectively.

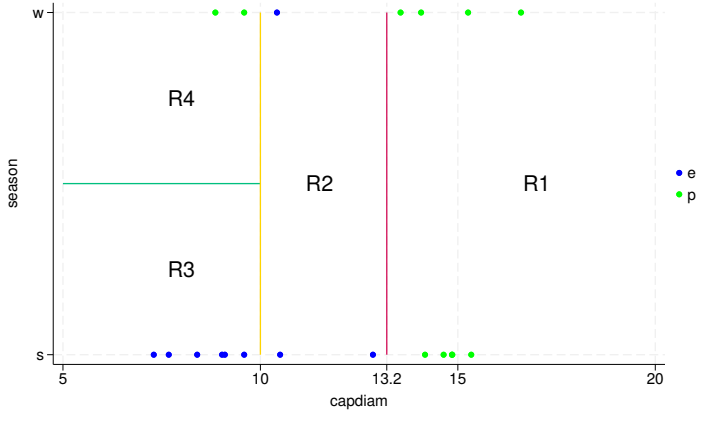

Figure 6.

We can now classify observations by first determining to which terminal node they belong based on their predictor values and then finding the most common class in that terminal node. Thus, for an observation in the terminal node j with the corresponding region R_j, an observation is predicted to be in the class with the largest proportion of observations from the training data, $\max_k p_{jk}$, where p_{jk} is the proportion of training observations in R_j belonging to class k and $k = 1, 2, \ldots, K$. Suppose we have

a new observation for which `capdiam` $= 8.32$ and `season` $=$ `winter`. If we "put" this observation in the classification tree above, it will end up in the terminal node 4 in the region R_4 with 0 edible and 2 poisonous mushrooms. Therefore, our tree will classify the new observation as a poisonous mushroom.

We now discuss how to choose which predictor to split on and how to determine the best split in each nonterminal node in a decision tree. To choose the predictor and split point, we need to introduce impurity measures that quantify the splitting criteria. One such measure is the misclassification error rate. For a terminal node j with the corresponding region R_j, the misclassification error rate is the fraction of training observations that do not belong to the most common class, that is, $1 - \max_k \widehat{p}_{jk}$, where $\widehat{p}_{jk}$ is an estimate of p_{jk}. Unfortunately, the misclassification error rate is not very sensitive to changes in the class probabilities of each node, meaning that multiple splits may correspond to the same class probabilities, making it difficult to select the best splits. Thus, the misclassification error rate is not recommended for growing a classification tree.

Instead, the following measures are used: The Gini index,

$$\sum_{k=1}^{K} \widehat{p}_{jk}(1 - \widehat{p}_{jk})$$

and cross-entropy,

$$-\sum_{k=1}^{K} \widehat{p}_{jk} \ln \widehat{p}_{jk}$$

The Gini index and cross-entropy are close to zero when all proportions $\widehat{p}_{jk}$'s are close to zero or one. This explains the name "impurity measure"—a small value indicates that the node contains many observations from the same class.

Here we focus on cross-entropy. When the number of groups $K = 2$, cross-entropy is

$$\imath_j = -\widehat{p}_{j1} \ln \widehat{p}_{j1} - (1 - \widehat{p}_{j1}) \ln(1 - \widehat{p}_{j1})$$

The goal of classification trees is to partition the predictor space into regions $R_1, R_2, \ldots, R_J$ that minimize cross-entropy. In practice, the consideration of every possible partition of the predictor space into J rectangles is computationally infeasible. A typical remedy for such problems is to use a greedy approach and successively split the predictor space into two new regions through binary splitting. The binary splitting is performed by first selecting the predictor x_i and the split point t such that it leads to the greatest possible reduction in cross-entropy. In other words, the method examines all predictors x_1 through x_p and considers all possible values of the split point t such that the selected predictor x_i and cutpoint t result in the lowest cross-entropy. Once we have determined the best split point for a given predictor, we can use this information to split the data into two sets and repeat the process for each of the two new sets, continuing until we reach a terminal node or until a stopping criterion is reached.

We start by considering a possible split for the root node. Because the variable `season` is binary, we can tabulate it to determine the possible split point t.

```
. tabulate class season, column
```

Key
frequency *column percentage*

class	season s	 w	 Total
e	8 61.54	1 14.29	9 45.00
p	5 38.46	6 85.71	11 55.00
Total	13 100.00	7 100.00	20 100.00

From the above table, `season` splits the dataset into two nodes: summer, `s`, and winter, `w`. The summer node contains 8 edible and 5 poisonous mushrooms, and the winter node contains 1 edible and 6 poisonous mushrooms, respectively. The cross-entropy for the summer and winter nodes can be computed as

$$\imath(\text{summer}) = -\frac{8}{13}\ln\frac{8}{13} - \frac{5}{13}\ln\frac{5}{13} \approx 0.666$$

and

$$\imath(\text{winter}) = -\frac{1}{7}\ln\frac{1}{7} - \frac{6}{7}\ln\frac{6}{7} \approx 0.410$$

The summer and winter nodes contain different numbers of observations. Thus, to find the cross-entropy for the split, we take the weighted average of the entropies in each region:

$$\imath(\text{season}) = -\frac{13}{20}0.666 - \frac{7}{20}0.410 \approx 0.576$$

We can also find the importance or the goodness of fit of the split by measuring the improvement of the impurity measure gained from splitting the parent node into the summer and winter children nodes,

$$\imath(\text{summer}, \text{winter}) = \imath(\text{season}_b) - \imath(\text{season}) \tag{1}$$

where season_b indicates the cross-entropy before the split. Here

$$\imath(\text{season}_b) = -\frac{9}{20}\ln\frac{9}{20} - \frac{11}{20}\ln\frac{11}{20} \approx 0.688$$

Therefore, $\imath(\text{summer}, \text{winter}) = 0.112$. This value indicates the improvement attributed to this split and can be used as a measure of the predictor's importance.

Next we consider splits for the cap diameter predictor. Conventionally, to estimate the cross-entropy for a continuous variable, we first need to sort the data and consider all possible cutpoints (Breiman et al. 1984). For example, for the cap diameter, a possible cutpoint t between the respective 1st and 2nd values of 7.3 and 7.68 is selected as $t = (7.3 + 7.68)/2$, between the 2nd and 3rd values of 7.68 and 8.4, $t = (7.68 + 8.4)/2$, and so on. However, for high-dimensional data such an approach is computationally

expensive. To overcome this, some software packages, such as H2O, divide the data into discrete equal-size sections by using histogram bins and then estimate the best split among those sections (Ben-Haim and Tom-Tov 2010; Chen and Guestrin 2016; and Ke et al. 2017).

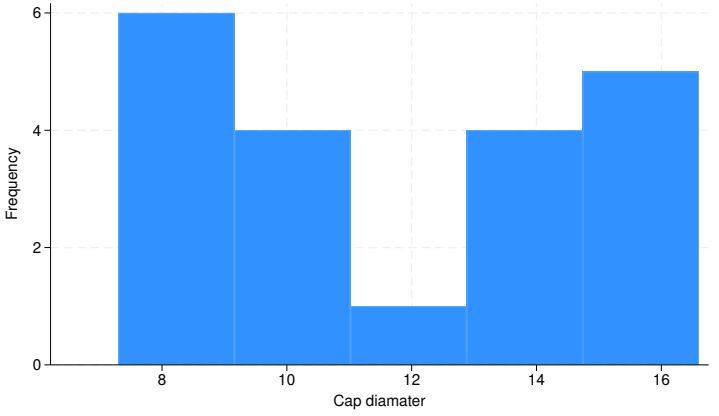

Figure 7.

For illustration purposes, we considered five bins for the histogram of `capdiam`. The number of splits to be evaluated is then determined by the number of bins in the histogram. In practice, the number of bins is a hyperparameter, that is, a parameter that we learn or tune using the training data such that the tuned parameters minimize the generalization error; see *Hyperparameter tuning*. After binning, the number of possible split points reduces to five. For example, because the 1st bin contains 6 observations, a potential split point can be computed by averaging the 6th and 7th observations for `capdiam` in the dataset: $t = (9.1 + 9.59)/2 = 9.345$. Similarly, we can compute all 5 split points, which are $\{9.345, 11.68, 13.2, 14.75, 16.6\}$.

We show the calculation of the cross-entropy only for the split point $t = 13.2$, which is the best split point. You can calculate the cross-entropy for the other split points similarly. The criterion ($\texttt{capdiam} \leq 13.2$) splits the data into two regions, where the left region contains 9 edible and 2 poisonous mushrooms and the right region contains 0 edible and 9 poisonous mushrooms. The right region, which contains observations for which ($\texttt{capdiam} > 13.2$), is called pure because it is homogeneous and is a terminal node. Analogously to the splits for the season predictor, we can compute the cross-entropy for the left and right regions as

$$\imath(\text{left}) = -\frac{9}{11}\ln\frac{9}{11} - \frac{2}{11}\ln\frac{2}{11} \approx 0.474$$

and

$$\imath(\text{right}) = 0$$

Therefore, the cross-entropy for the split is equal to

$$\imath(\texttt{capdiam} \leq 13.2) = \frac{11}{20}0.474 + \frac{9}{20}0 \approx 0.261$$

The cross-entropy before the split can be computed by using the actual class distribution of `class`:

$$\imath(\texttt{capdiam}_b) = -\frac{11}{20}\ln\frac{11}{20} - \frac{9}{20}\ln\frac{9}{20} \approx 0.688$$

From the above, the importance of the `capdiam` split is

$$\imath(\texttt{capdiam} \leq 13.2, \texttt{capdiam} > 13.2) \approx 0.688 - 0.261 = 0.427$$

Thus, in the root node we select the cap diameter with the best split $t = 13.2$, because the gain from the cap diameter split (0.427) is larger than the gain from the season split (0.112). The next best split is found following the same steps but by considering only the subset of the dataset that satisfies the criterion ($\texttt{capdiam} \leq 13.2$). The tree grows recursively until all observations are classified.

In the last recursive split ($\texttt{winter} = 1$), the left region contains only two observations. Splits with few observations may lead to overfitting. To avoid overfitting, we recommend to limit the minimum number of observations that a leaf node may have for the node to be considered for splitting. For example, if we limit the minimum number of observations in the leaf nodes to three, then the last split ($\texttt{winter} = 1$) will not occur because this criterion requires that both branches have at least three observations.

In general, each split increases the depth of the decision tree, and large trees usually overfit the data. On the other hand, small trees may not capture a complex structure hidden in the data. Thus, the tree size is treated as a hyperparameter, and its optimal value is chosen from the data.

For the multiclass classification with K classes, the preferred approach is to compare each class k with the rest (Rifkin and Klautau 2004). That is, we grow K different trees and for each k find the probability of class k, p_k. Then the final class prediction is computed as $\max_k p_k$.

Regression trees

The general idea for growing a regression tree is similar to a classification tree. The main goal is to partition the predictor space into distinct and nonoverlapping regions by using binary splits. However, because in regression trees the response is continuous, we use the residual sum of squares $\text{RSS} = \sum_{i=1}^{N}(y_i - \hat{y}_i)^2$ as an impurity measure instead of the cross-entropy to determine the best split at each node. Then, for each terminal node, the prediction is computed as the mean of the response values $\mathbf{y}$ in the region corresponding to the terminal node. For example, if the mean response of the training observations in the first region R_1 is $\hat{c}_1 = 5$, then for a given observation $\mathbf{x}_i \in R_1$, the regression tree will predict a value of $\hat{c}_1 = 5$. Thus, the regression model prediction for J distinct and nonoverlapping regions, which correspond to J terminal nodes, can be represented as

$$\hat{f}(\mathbf{x}) = \sum_{j=1}^{J} \hat{c}_j I\{\mathbf{x} \in R_j\}$$

where $\hat{c}_j = \text{Mean}(y_i | \mathbf{x}_i \in R_j)$.

In general, growing a regression tree can be summarized by the following two steps (James et al. 2021):

1. Partition the predictor space into J distinct and nonoverlapping regions $R_1, R_2, \ldots, R_J$.
2. For each observation that belongs to the region R_j, predict the response as the mean of the response values for the training observations in R_j.

Therefore, the goal of a regression tree is to partition the predictor space into rectangles $R_1, R_2, \ldots, R_J$ that minimize the RSS:

$$\sum_{j=1}^{J} \sum_{i \in R_j} (y_i - \hat{c}_j)^2$$

Similar to a classification tree, the binary splitting is performed by first selecting the predictor x_i and the cutpoint t such that it leads to the greatest possible reduction in RSS. Mathematically, in each nonterminal node, a regression tree tries to select the predictor x_i and cutpoint t such that the following expression is minimized,

$$\min_{i,t} \left\{ \sum_{\mathbf{x}_i \in R_1(i,t)} (y_i - \hat{c}_1)^2 + \sum_{\mathbf{x}_i \in R_2(i,t)} (y_i - \hat{c}_2)^2 \right\}$$

where $R_1(i,t) = \{\mathbf{x} | x_i \leq t\}$ and $R_2(i,t) = \{\mathbf{x} | x_i > t\}$. Then the above process is repeated recursively to minimize the RSS within each region. As for a classification tree, the importance of the split $\imath(\cdot)$ is defined as the difference between the RSS before and after the split.

It is recommended to apply a stopping criterion to avoid overfitting. For example, the node splitting may be terminated if the method reaches some predetermined tree depth or the terminal regions contain no more than a prespecified number of observations.

After the terminal nodes and the corresponding regions are determined, we obtain predictions for the test observations by first identifying to which terminal nodes the test observations belong. Then the predicted response is computed as the mean of the training observations in the corresponding terminal node. This is in contrast with classification trees, where the predicted response is determined by the most common class among the training observations in the terminal node.

One issue with decision trees is that the partitioning of a categorical predictor can take different but equally justifiable paths. For example, we can decompose categories into binary predictors and include them individually in the model (also known as one-hot encoding) or implement more dynamic splits, such as groups of two or more categories. The best approach depends on the specific data and model. In general, the partitioning algorithm tends to favor categorical predictors with many levels, leading to severe overfitting when the number of categories is large; see, for instance, *Effect of categorical predictors* in [H2OML] **h2oml**. Therefore, it is recommended to avoid such predictors.

Pros and cons of decision trees

One of the key advantages of decision trees is that they represent information in an intuitive and easy-to-visualize way. In a decision tree, predictors can be of any type: numeric, binary, categorical, etc. A monotone transformation or different scales of measurements among predictors do not change the model outcome.

Another advantage of decision trees is that they can handle missing data. For instance, missing values are often treated as containing information, which does not require the common missing-at-random assumption. For categorical predictors, missing values are treated as a separate category that can split left or right; for other types of predictors, the missing values split to the left. Then, for the testing or validation data, the missing values follow the path on the tree that was determined during training. If there are no missing values in the training data, then missing values in the testing or validation data follow the path of the most training observations. Missing values in the response are also allowed, but nothing will be learned from observations containing those missing values.

Despite their advantages, decision trees are notoriously unstable and have a high variance. Even though a deep tree (with many terminal nodes) has a small bias, a small change in the data can lead to a completely different set of splits and obscure its interpretation. Moreover, decision trees have difficulties with modeling simple smooth functions; see, for instance, *Introduction* in [H2OML] ***h2oml gbm***.

One solution is to use ensemble methods, which we introduce next.

Ensemble methods

The basis for ensemble methods can be summarized as a mechanism that forms a smart committee of incompetent but carefully selected members to solve a machine learning problem. As we discussed in the previous section, despite their advantages such as efficiency and interpretability, decision trees suffer from high variance and instability. Specifically, if we slightly modify the data by splitting them or introducing nuisance predictors, the new results may differ substantially from the original results. In contrast, the low-variance methods are more robust to small changes and tend to yield similar results.

Bagging and boosting are two methods used to improve the accuracy of a machine learning method by combining unstable learners. Using unstable learners is important because they provide more variable outcomes than stable learners and thus aid in generalization. Both methods perturb the original dataset to generate an ensemble of various base learners and combine them into one method. The usefulness of ensemble methods is established for unstable base learners, but these methods may produce contradictory results for stable base learners such as a linear regression.

Both bagging and boosting methods are general-purpose procedures and are not tied to a specific learning estimation method, but in this entry, our main focus is on bagging and boosting for decision trees. The main difference between bagging and boosting is in how they perturb and generate new datasets. Bagging, which was first introduced in Breiman (1996), generates the perturbations by random and independent drawings (bootstrap samples) from the training data. In contrast, boosting, introduced by Freund and Schapire (1997) to solve classification problems, has a deterministic approach and generates perturbations by sequentially reweighting the dataset. In particular, at any step, the weights of the observations that were misclassified in the previous step increase, whereas the weights for the correctly classified observations decrease. Thus, boosting forces each successive classifier to focus on those observations that were missed by the previous ones in the sequence. By design, bagging reduces variance, whereas boosting tends to control the generalization error by reducing bias. The difference is summarized in the figure below.

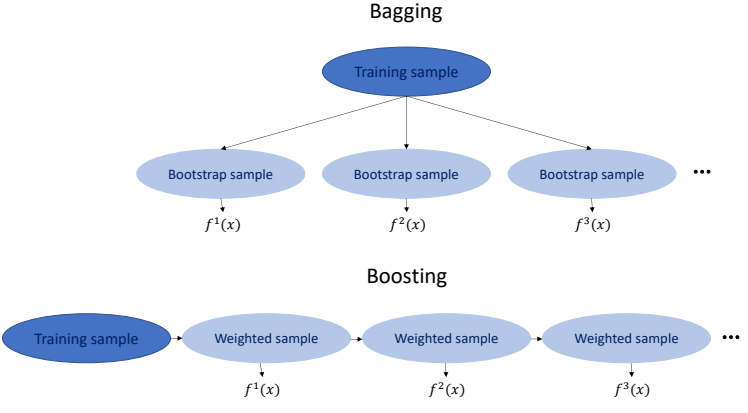

Figure 8.

Bagging

Bagging or bootstrap aggregation relies on a bootstrap procedure (Efron 1979) that combines an ensemble of learners to improve the performance of the prediction. The main idea of bagging can be motivated by the fact that the variance of the mean of n independent observations $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n$ with

variance σ^2 is σ^2/n. Consequently, averaging a set of independent observations reduces the variance. A natural extension of this idea to the machine learning is to independently sample many training datasets from the population, build a separate prediction model $\hat{f}^b(\mathbf{x})$ for each sample, and take the average. Unfortunately, this approach is not viable because, in practice, we observe only one training dataset. However, we can use bootstrap to generate samples from the training dataset. Thus, after building the $\{\hat{f}^b(\mathbf{x}), b = 1, 2, \ldots, B\}$ learners from the bootstrap samples, for the observation $\mathbf{x}$, the bagging procedure returns

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

The bias of a bagged tree is the same as that of a single tree, because each tree generated from the bootstrapped data is identically distributed and has the same expected value.

To apply bagging to regression trees, we grow B deep regression trees using B bootstrap samples and take the average of the resulting predictions. Each deep regression tree has a high variance and low bias. Therefore, averaging these B trees substantially reduces the variance and improves the prediction accuracy; see *Fundamentals of machine learning* for details about the bias–variance tradeoff.

There are several approaches for extending bagging to classification trees. The most common one is the majority-vote rule. For the ith observation of the testing data, we can record the predicted class for each of the B classification trees. The majority-vote rule returns the most frequent class among these B predictions.

A salient feature of bagging is its ability to estimate the test error of a bagged model. This feature helps avoid arduous computations and is especially useful for large datasets. Bagging repeatedly builds trees on bootstrap samples, and about 37% of the observations in the training data will not be selected for each bootstrap sample (Izenman 2008, chap. 5). Therefore, each bagged tree is grown only on the remaining two-thirds of observations. The 37% of observations that are not used to grow the tree serve as an independent testing set. Such observations are called out-of-bag observations. Now, to predict the response for the ith observation, we use each of the trees for which the ith observation was out of bag. The average (or the majority vote in the case of classification) of those predicted responses yields a single prediction for the ith observation. The estimated generalization error from the out-of-bag approach is a valid estimate of the test error and is equivalent to using an independent testing set of the same size.

Random forest

Recall that bagging averages an ensemble of unstable decision trees to reduce the variance, which leads to the improvement of the generalization error. However, this reduction may not be sufficient if the trees in the ensemble are correlated with each other. For example, if the training data have one strong and several moderately strong predictors, then in the ensemble of bagged decision trees, the majority of the trees will have this strong predictor as the top split. Therefore, most of the bagged trees will have a similar structure, resulting in predictors that are highly correlated.

Although historically a variety of tree ensembles have been referred to as a random forest (Lin and Jeon 2006), nowadays, a random forest is associated with the random forest proposed in Breiman (2001), which is a tree ensemble that uses both bagging and subsampling of predictors. It is a modification of the bagging procedure that generates an ensemble of decorrelated trees and then averages them. To overcome the shortcomings of the bagging procedure and achieve decorrelation, for each split in the tree, instead of the full set of p predictors, random forest selects a random sample of m predictors as potential split candidates. With this strategy, the strong predictors, on average, $(p - m)/p$ times are not considered

as potentially the best predictors to split on, which increases the chance that other predictors can be considered for splitting. Below, we summarize the main steps of a random forest. For $b = 1, 2, \ldots, B$, do the following:

1. Generate a bootstrap sample D^b from the training data.
2. Until the stopping criterion is reached, recursively grow a tree T_b by implementing the following steps:
   i. Randomly choose $m \leq p$ predictors.
   ii. Select the predictor with the best split point from m potential predictors.
   iii. Split the selected node.

Similar to bagging, to make a prediction for a new test point $\mathbf{x}$, random forest estimates $\hat{f}_{\text{rf}}(x) = (1/B)\sum_{b=1}^{B} \hat{f}^b(x)$ for regression, where $\hat{f}^b(\cdot)$ is a prediction model from the tree T_b, and uses the majority-vote rule for classification. In practice, it is recommended to select $m = \lfloor \sqrt{p} \rfloor$ for classification and $m = \lfloor p/3 \rfloor$ for regression, where $\lfloor \cdot \rfloor$ is a floor function. The size of the bootstrap sample D^b controls the bias–variance tradeoff of the random forest.

A smaller bootstrap sample size lowers the probability of a particular training observation to be included in the bootstrap sample, which decreases similarity among the individual trees. The latter helps reduce overfitting. Analogously, a larger bootstrap sample size increases the degree of overfitting.

The above approach describes a random forest as a complex black-box model. We find it helpful to also describe a random forest from a different perspective that connects it to the existing well-understood statistical methods. Specifically, the prediction from a random forest can be viewed as an adaptive neighborhood classification or regression procedure (Lin and Jeon 2006). Recall from decision trees that every terminal node $j = 1, 2, \ldots, J$ of a tree corresponds to a rectangular subspace R_j of a predictor space such that for every observation $\mathbf{x}_i$, there is only one terminal node j such that $\mathbf{x}_i \in R_j$. Let's focus on a prediction from a single tree T_b at a new data point $\mathbf{x}_0$. Suppose that in the tree T_b, $\mathbf{x}_0$ belongs to the terminal node j with the corresponding region $R_j(\mathbf{x}_0, b)$, where we make the dependence of the region on $\mathbf{x}_0$ and tree T_b explicit. Then the prediction is obtained by averaging the observed values y_i's in the region $R_j(\mathbf{x}_0, b)$. Let's assign the weight $w_i(\mathbf{x}_0, b)$ a positive constant if the observation $\mathbf{x}_i$ is in the region $R_j(\mathbf{x}_0, b)$ and 0 otherwise, such that

$$w_i(\mathbf{x}_0, b) = \frac{1\{\mathbf{x}_i \in R_j(\mathbf{x}_0, b)\}}{|\{k\colon \mathbf{x}_k \in R_j(\mathbf{x}_0, b)\}|}$$

where $|\cdot|$ denotes the number of observations in the region $R_j(\mathbf{x}_0, b)$ and $1(A)$ is the identity function, which is equal to 1 if the condition A holds and 0 otherwise. Note that the weights sum to one: $\sum_{i=1}^{n} w_i(\mathbf{x}_0, b) = 1$. Thus, the prediction from a single tree given a new point $\mathbf{x}_0$ is the weighted average of the original observations y_i's for $i = 1, 2, \ldots, n$:

$$\hat{f}^b(\mathbf{x}_0) = \sum_{i=1}^{n} w_i(\mathbf{x}_0, b) y_i$$

For a random forest, where B trees are ensembled, the prediction at observation $\mathbf{x}_0$ can be written as

$$\hat{f}_{\text{rf}}(\mathbf{x}_0) = \frac{1}{B}\sum_{b=1}^{B}\hat{f}^b(\mathbf{x}_0) = \frac{1}{B}\sum_{b=1}^{B}\sum_{i=1}^{n} w_i(\mathbf{x}_0, b)y_i = \sum_{i=1}^{n}\overline{W}_i(\mathbf{x}_0)y_i$$

where $\overline{W}_i(\mathbf{x}_0)$ is the average of the weights w_i's over B trees:

$$\overline{W}_i(\mathbf{x}_0) = \frac{1}{B}\sum_{b=1}^{B} w_i(\mathbf{x}_0, b)$$

Consequently, a random forest prediction can be viewed as a weighted average of the observations y_i's because $\sum_{i=1}^{n}\overline{W}_i(\mathbf{x}_0) = 1$, which makes a random forest an adaptive smoother (Curth, Jeffares, and van der Schaar 2024). For most observations, the weight $\overline{W}_i$ will be zero; see Lin and Jeon (2006), Meinshausen (2006), and Biau and Scornet (2016).

Wager and Athey (2018) rely on the above approach to prove the consistency of the random forest estimator. In figure 9, we use a toy example to visualize this approach. Here, for a new data point $\mathbf{x}_0$ (denoted by $+$), each tree assigns a positive weight to the observations in the same terminal node (denoted in red) and zero weight to the rest of the observations. The random forest prediction averages the weights from the three trees and measures how frequent each observation falls into the same terminal node as $\mathbf{x}_0$.

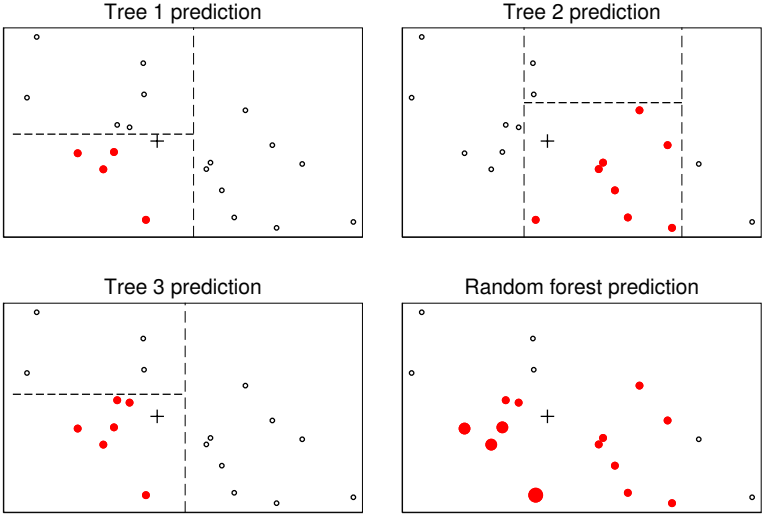

Figure 9.

Boosting

Boosting is a powerful idea that can be applied to any regression or classification problem. In contrast to bagging, where each tree in an ensemble is built on a bootstrap training dataset and independent of the other trees, boosting grows trees sequentially. One of the first boosting methods, AdaBoost (Freund and Schapire 1997), was introduced to solve classification problems. AdaBoost repeatedly applies weights to the observations to produce a sequence of classifiers. The observations that are poorly modeled get higher weights and vice versa. This way, each successive classifier is focused on those observations that received higher weights in the previous iteration. The figure below summarizes the steps of AdaBoost.

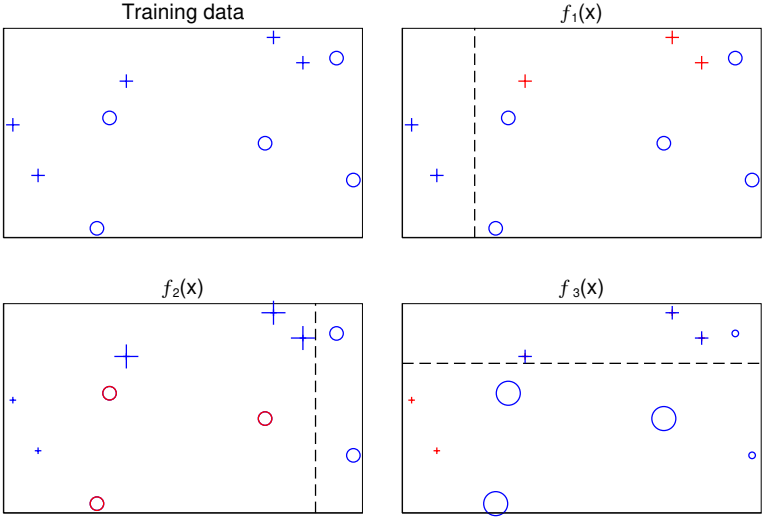

Figure 10.

Here we have three classifiers or base learners, $f_1(\mathbf{x})$, $f_2(\mathbf{x})$, and $f_3(\mathbf{x})$, which can be classification trees. The observations are classified based on $+$'s and o's. AdaBoost starts by assigning the same weight $1/n$ to all observations, where n is the number of observations. $f_1(\mathbf{x})$ incorrectly classified three $+$ observations, which are displayed in red. In the next iteration, those three observations were assigned higher weights, and $f_2(\mathbf{x})$ classified those observations correctly. Similarly, $f_3(\mathbf{x})$ assigned more weight to the three previously misclassified o observations and classified them correctly. The final ensemble or boosted classifier is obtained based on those three classifiers as $F(\mathbf{x}) = \sum_{m=1}^{M} \alpha_m f_m(\mathbf{x})$, where α_m measures the importance of the classifier $f_m(\cdot)$ and M is the number of classifiers.

This approach tends to explain boosting in terms of updating weights, which makes it difficult to evaluate its performance (Schapire 2003). To establish a connection with the statistical framework, in their seminal paper, Friedman, Hastie, and Tibshirani (2000) propose a different view of AdaBoost. In particular, the authors use a gradient-descent-based formulation to reformulate AdaBoost as an optimization problem and show that it is a greedy procedure that minimizes the exponential loss,

$$L\{y_i, F(\mathbf{x}_i)\} = \frac{1}{n} \sum_{i=1}^{n} e^{-y_i F(\mathbf{x}_i)}$$

where $F(\mathbf{x}_i) = \sum_{m=1}^{M} \alpha_m f_m(\mathbf{x}_i)$. They proposed the following coordinate descent algorithm to achieve the minimization.

1. Initialize: $F_0(\mathbf{x}) = 0$.
2. For $m = 1, 2, \ldots, M$:
   i. Choose a classifier $f_m(\cdot)$ and α_m to minimize
   $$\frac{1}{n}\sum_{i=1}^{n} \exp[-y_i\{F_{m-1}(\mathbf{x}_i) + \alpha_m f_m(\mathbf{x}_i)\}]$$
   ii. Update: $F_m(\mathbf{x}) = F_{m-1}(\mathbf{x}) + \alpha_m f_m(\mathbf{x})$.
3. Output: $F_M(\mathbf{x})$.

Thus, AdaBoost minimizes its loss function by iteratively descending toward one coordinate direction at each iteration.

The important feature of this loss-function formulation is that, instead of the exponential loss, one can use any other loss function and extend AdaBoost from solving a classification problem to solving a regression problem. For details, see Friedman, Hastie, and Tibshirani (2000), Schapire and Freund (2012), and Hastie, Tibshirani, and Friedman (2009).

GBM

The formulation discussed in the previous section and the corresponding models are called GBMs. GBM is one of the popular methods to implement boosting. Although the original method, proposed in Friedman, Hastie, and Tibshirani (2000), can work with any base learner, in practice, decision trees are some of the main choices.

In the previous section, we viewed AdaBoost as an optimization problem with some loss function $L(F)$. In *Decision trees*, we parameterized a decision tree as a model $f(\mathbf{x}) = \sum_{j=1}^{J} c_j I\{\mathbf{x} \in R_j\}$, where J is the number of terminal nodes, R_j's are nonoverlapping regions of the predictor space, and c_j is the prediction (the mean for regression and the most probable class for classification) in the terminal node j.

The main idea behind GBM is to parameterize the estimate of the ensemble function $F(\mathbf{x})$ as

$$\hat{F}(\mathbf{x}) = \sum_{i=0}^{M} \hat{F}_m(\mathbf{x})$$

where M is the number of iterations, $\hat{F}_0(\cdot)$ is an initial guess, and $\{\hat{F}_m(\cdot)\}_{m=1}^{M}$ are the function increments, also known as boosts.

Parameterizing the tree by $\Theta = \{R_j, c_j\}_{j=1}^{J}$ and following the coordinatewise approach presented in the previous section, for some loss function $L(\cdot)$, in the stage m, we can write the minimization of the tree-boosting method as

$$(\alpha_m, \Theta_m) = \text{argmin}_{\alpha,\Theta} \sum_{i=1}^{n} L\{y_i, \hat{F}_{m-1}(\mathbf{x}_i) + \alpha f(\mathbf{x}_i, \Theta)\}$$

where n is the number of observations in the training dataset, α is a learning rate, and

$$\hat{F}_m(\mathbf{x}) = \hat{F}_{m-1}(\mathbf{x}) + \alpha f(\mathbf{x}, \Theta_m)$$

Unfortunately, such minimization is practically infeasible to solve. To alleviate the issue, it was proposed, at stage m, to choose a new function $f(\mathbf{x}, \theta)$ to be the most correlated with the negative gradient

$$g_m(\mathbf{x}_i) = \Big[\frac{\partial L\{y_i, F(\mathbf{x}_i)\}}{\partial F(\mathbf{x}_i)}\Big]_{F(\mathbf{x}_i)=\widehat{F}_{m-1}(\mathbf{x}_i)}$$

by solving a classical least-squares minimization problem:

$$(\alpha_m, \Theta_m) = \text{argmin}_{\alpha,\Theta} \sum_{i=1}^{n} \{-g_m(\mathbf{x}_i) + \alpha f(\mathbf{x}_i, \Theta)\}$$

For example, if the loss function is the squared error loss $L\{y_i, F(\mathbf{x}_i)\} = (1/2)\{y_i - F(\mathbf{x}_i)\}^2$, then the gradient $g_m(\mathbf{x}_i) = -\{y_i - F(\mathbf{x}_i)\}$.

Below, we summarize the gradient-tree boosting method for the squared error loss $L(\cdot)$ and fixed learning rate α, with the number of iterations, that is, the number of trees in this context, equal to M.

1. Initialize: $F_0(\mathbf{x})$ and $g_i = y_i$ for all $1 \le i \le n$.
2. For $m = 1, 2, \ldots, M$:
   i. Compute $g_m(\mathbf{x}_i) = -\{y_i - F_{m-1}(\mathbf{x}_i)\}$ for all $1 \le i \le n$.
   ii. Fit a tree $\hat{f}_m(\cdot)$ with J splits to the training data $\{\mathbf{x}_i, -g_m(\mathbf{x}_i)\}$ for $i = 1, 2, \ldots, n$.
   iii. Update $\hat{F}$: $\hat{F}_m(\mathbf{x}) = \hat{F}_{t-1}(\mathbf{x}) + \alpha \hat{f}_m(\mathbf{x})$.
3. Output: $\hat{F}(\mathbf{x}) = \sum_{m=1}^{M} \hat{F}_m(\mathbf{x}) = \sum_{m=1}^{M} \alpha f_m(\mathbf{x})$.

The learning rate α reduces the contribution of each tree as it is added to the model, which prevents overfitting. The simulation studies indicate that a smaller α reduces overfitting and provides a lower generalization error. The relationship between the learning rate and the number of trees M is reciprocal. That is, decreasing the learning rate increases the required number of trees.

Historically, researchers suggested using a stump (decision tree with depth equal to one) as a base learner in each iteration. However, current research on ensemble methods suggests that if the noise in the data is small, it is preferable to use deeper trees as base learners to improve generalization (Wyner et al. 2017). This is related to the idea that the ensemble methods are local interpolators. The depth of a tree affects the selection of the optimal number of trees. For a given learning rate, fitting more complex (deeper) trees results in a fewer number of trees being selected. Typically, the learning rate and tree complexity are inversely related: doubling the tree depth should be matched with halving the learning rate to provide roughly the same number of trees (Elith, Leathwick, and Hastie 2008).

Trees with monotonicity constraints

In some applications, it is reasonable to assume that the response is a monotone function of the predictors. For example, in economic theory the price elasticity of the normal good is assumed to be positive, or in hedonic price analysis, in which price is a function of the characteristics of the product, it is expected that some of the characteristics will always have a positive or negative effect on the price. The original decision trees and ensemble decision tree methods, described above, do not support such a constraint and may violate the monotonicity assumption. However, there are modifications to the above methods that incorporate the monotonicity constraints (Potharst and Feelders 2002).

Model selection in machine learning

Most machine learning models are defined by a set of model parameters and hyperparameters. A model parameter is initialized and computed during the learning process. A hyperparameter cannot be directly estimated from the learning process and must be prespecified before training a machine learning model (Kuhn and Johnson 2013). For example, in decision trees, the parameters correspond to the split decisions and regions, and the hyperparameters include the tree depth, impurity measures, the minimum number of observations in each terminal node, and more. The goal of machine learning models is to make accurate predictions on future data. To build an optimal model, we need to explore a wide range of values for hyperparameters and select the ones that improve the model performance the most. This process is also known as model selection. So we are interested in selecting the best-performing model from the set of potential models. That is, we want to evaluate the performances of the models and compare them with each other. The process of designing an effective machine learning model with an optimal hyperparameter configuration is called hyperparameter tuning. The material in this section closely follows Raschka (2020) and Yang and Shami (2020).

The steps for selecting the best-performing model are summarized in table 1 below.

Table 1. Steps for selecting the best-performing model

To minimize the generalization error, which measures the predictive model performance on new data, do the following:

1. Split the data for training and evaluating a model; see *Three-way and two-way holdout methods*.
2. Optimize hyperparameters to select the best-performing model; see *Hyperparameter tuning*.
3. Compare different machine learning methods and select the one that performs the best; see *Method comparison*.

In the rest of this section, we will discuss different approaches to accomplish the above steps.

Three-way and two-way holdout methods

The simplest approach to evaluating a model is the two-way holdout method, in which we take the observed data and split them into two parts: training data and testing data. A model is fit to the training data, and the prediction is obtained on the testing data. It is important to perform the training and evaluation steps using different data. Otherwise, if a sufficiently complex model fits the training data too well, it will be difficult to distinguish whether the model is memorizing the training data or generalizing well to the "new" data. Thus, the model performance will suffer from the optimism bias. Even after we randomly sample and split the data, it is essential to prevent the leakage of information from the testing data into the training process (Raschka 2020 and Lones 2021). Common, seemingly innocuous mistakes include using the information about the means and ranges of the predictors from the entire dataset to scale the predictors or performing predictor selection before partitioning the data and using the same data as testing data to evaluate the generality of multiple models. The best practical way to prevent information leakage is to partition the data at the beginning of the analysis and use the testing data only once to measure the generality of a final model at the end of the analysis (Cawley and Talbot 2010).

The two-way holdout method addresses only the first generalization step from table 1 and cannot be used to sequentially train multiple models for hyperparameter optimization, which we discuss later. In contrast, the three-way holdout method partitions the dataset into training, validation, and testing data. Model selection and hyperparameter tuning are performed on training and validation data and model evaluation on testing data. This procedure avoids repeated use of the testing data and prevents information leakage. Another advantage of including validation data is that we can impose early stopping rules, in which the model performance is measured against validation data at each iteration, and stop training when the performance score starts deteriorating or does not change over a sequence of iterations. In general, to obtain a generalization error, which is independent from how we split the data into training, validation, and testing, we recommend to repeat the holdout method multiple times with different random-number seeds and report the average performance over these repetitions. Alternatively, one can use the leave-one-out bootstrap technique and evaluate the generalization error by using the out-of-bag samples instead of the training data (Efron and Tibshirani 1993).

The steps for selecting the best-performing model with the three-way holdout method are summarized in table 2.

Table 2. Steps for selecting the best-performing model with the three-way holdout method

1. Randomly partition the data into three parts: training for model fitting, validation for model selection, and testing for the final evaluation of the selected model.
2. Hyperparameter tuning: define a grid of various hyperparameter configurations to fit models to the training data; see *Hyperparameter tuning*.
3. Model selection: evaluate and compare the estimated performance metrics on the validation data, and choose hyperparameter values that provide the best-performing metrics.
4. Use independent testing data to estimate the generalization error by comparing various metrics of the best-performing model.

In step 2, tuning can be performed by using either a Cartesian grid search (as described in table 4) or a random grid search. We treat the splitting of a dataset into training, validation, and testing data as random subsampling and assume that each observation has been drawn from the same probability distribution. However, when the dataset is imbalanced, random subsampling is not recommended. A better approach is to divide the dataset in a way that preserves the original class proportions in the resulting subsets (training, validation, and testing). This approach is called stratification.

k-fold cross-validation

For small datasets, the three-way holdout method of splitting the data is not recommended because the validation and testing data may not be representative. In such cases, k-fold cross-validation is the most common model evaluation and selection technique. It starts by splitting the data into training and testing data. For the training data, k-fold cross-validation splits them into k parts or folds. In each kth iteration, it uses one part for validation and the remaining $k - 1$ parts as a training subset for model fitting. The figure below illustrates 3-fold cross-validation for a toy example. The dataset is randomly split into three folds, and red, blue, and green observations correspond to observations in folds 1, 2, and 3, respectively. In the first cross-validation iteration, the method uses observations in folds 2 and 3 as a training set and

observations in fold 1 as a validation set. The next two iterations follow a similar procedure but use observations from folds 2 and 3, respectively, as validation sets. For example, for $k = 3$, four models are fit. The first three cross-validation models are fit using 2/3 of the training data, as described above, and a different 1/3 of the training data is held out for validation for each of the three models. Then the main fourth model is fit using the entire training data, and the cross-validation metrics are reported. Also see [H2OML] **h2omlestat cvsummary**.

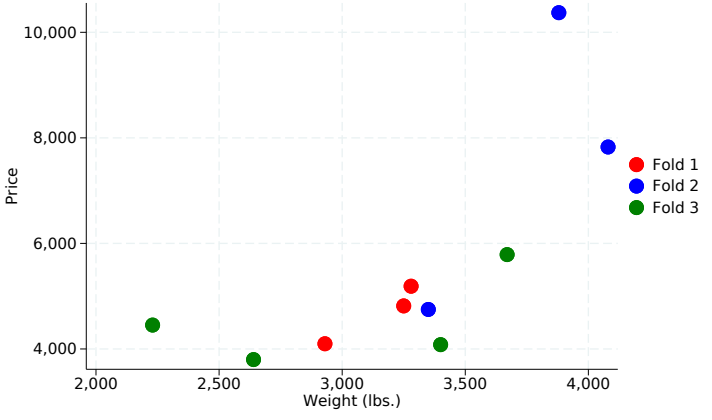

Figure 11.

Hyperparameter tuning

A typical process to build an effective machine learning model is complicated and time consuming. It involves choosing an appropriate method and selecting a model by tuning hyperparameters (see step 2 in table 1). The choice of optimal hyperparameters directly affects the model performance on the testing data. The hyperparameter tuning depends on a machine learning method and the type of hyperparameter, such as continuous, discrete, or categorical. Setting and testing hyperparameters manually is time consuming and inefficient. Therefore, there exist automatic optimization techniques for hyperparameter tuning.

The main goal of hyperparameter optimization is to achieve optimal model performance within a given budget, where budget refers to computational resources or the time allocated to tuning. We summarize the hyperparameter optimization process following Yang and Shami (2020) in table 3.

Table 3. Steps for hyperparameter optimization

1. Select the machine learning method and the performance metrics.
2. Select the hyperparameters that require tuning.
3. Determine the baseline or reference model by training the machine learning method using the default hyperparameter configuration.
4. Start with a large search space such as the hyperparameter feasible domain.
5. Refine the search space using well-performing hyperparameter values, or explore new areas if needed.
6. Select the best-performing hyperparameter configuration as the final result.

Some researchers often neglect the baseline determination step 3 and spend most of their time developing complex models, which may not outperform the simplest model. For example, if the task is binary classification or regression, then the baseline method can be the simplest known method such as logistic or linear regression. Or if our data are highly imbalanced with one of the classes containing 95% of observations, then this 95% can serve as our baseline, because the method that always predicts this class already has 95% accuracy and the preferred machine learning model should outperform this baseline.

The simplest hyperparameter tuning method is a so-called babysitting or trial and error approach, where a researcher manually experiments with various hyperparameter values using experience, intuition, or prior knowledge (Abreu 2019 and Elsken, Metzen, and Hutter 2019). Manual tuning is infeasible for most machine learning methods because they are complex and require many hyperparameters. The methods we describe next are more suitable for complex machine learning methods.

Decision-theoretical methods are one of the common techniques for hyperparameter optimization. The most popular ones are a Cartesian grid search (Bergstra et al. 2011) and a random grid search (Bergstra and Bengio 2012). A Cartesian grid search performs an exhaustive grid search of hyperparameter configurations and evaluates the Cartesian product of possible hyperparameter combinations. Its search is limited to the grid specified by the user and cannot explore other regions. To achieve good results, Yang and Shami (2020) suggest the steps that we summarize in table 4.

Table 4. Steps for Cartesian grid search

1. Choose a broad search space and a large step size.
2. Based on the results from step 1, refine the search space and step size using well-performing hyperparameter configurations.
3. Repeat step 2 until there is no substantive improvement in the performance metric.

A Cartesian grid search is exhaustive, which makes it infeasible for a high-dimensional hyperparameter configuration space. A random grid search overcomes this drawback by randomly choosing a set number of samples within the upper and lower bounds as candidate hyperparameter values. Those values are used to evaluate the model. The rest of the steps are the same as in table 4. Moreover, if the configuration space is large enough, then the global optimum of the tuning metric can be achieved. On a limited budget, a random grid search explores a larger search space than a Cartesian grid search. However, both Cartesian and random grid search methods share the same drawback: each hyperparameter evaluation is independent of the others, leading to wasted computational time and resources on poorly performing areas of the search space. For a review of hyperparameter optimization techniques, see Yang and Shami (2020).

Method comparison

Comparing evaluation results for different machine learning methods is fundamental to model selection (step 3 in table 1). This process typically includes a comparison of different performance metrics, visualization, and statistical analysis. The performance metrics of various machine learning methods are compared using testing data, and the best method is chosen based on the results. Visualization, such as receiver operating characteristics curves and precision–recall curves, are commonly used for comparison during binary classification. For details, see [H2OML] **h2omlgraph roc** and [H2OML] **h2omlgraph prcurve** and, more generally, [H2OML] **h2oml postestimation**. Depending on the research question, in addition to performance metrics, it may be important to also explore the explainability of the method. See the next section for details.

Interpretation and explanation

Machine learning models are ubiquitous in many fields. Despite their widespread use, they are often treated as black boxes that do not explain their predictions in a way that practitioners can understand. The misuse of black-box predictive models can lead to serious consequences, for instance, incorrectly denying parole, releasing dangerous criminals because of inadequate bail decisions, mispredicting air pollution level, and more (Rudin 2019). One of the concerns with deploying machine learning methods is whether their models and predictions can be trusted. And it is difficult to trust something that cannot be interpreted or explained. Traditionally, machine learning models are evaluated by comparing performance metrics using validation data. This may be unreliable because validation data may not always be fully representative of real-world data.

The use of interpretable models and explainable methods sheds light on model performance and encourages a transparent usage of black-box models. In machine learning, an interpretable model has the ability to explain its results in an understandable and transparent way without the need for additional methods (Doshi-Velez and Kim 2017). Commonly used interpretable models are linear and logistic regressions, decision trees, decision-set and rule-based methods and their extensions (Friedman and Popescu 2008; Letham et al. 2015 ; Lakkaraju, Bach, and Leskovec 2016; Rudin and Ustun 2018; and Chen et al. 2018). An interpretable model is judged based on several criteria, including interpretability and accuracy (Guidotti et al. 2018).

In contrast with interpretable models, explainable methods rely on external models and methods to make their predictions presentable and understandable to a human. In general, they do not create models that are inherently interpretable, but provide post hoc models that explain the prediction of the original black-box models (Goldstein et al. 2015 ; Ribeiro, Singh, and Guestrin 2016; Bastani, Kim, and Bastani 2017; and Lundberg and Lee 2017). It is not recommended to heavily rely on explainable models for high-stake decisions, such as in medicine, criminal justice, social bias, and other fields (Rudin 2019

and Ghassemi, Oakden-Rayner, and Beam 2021), but to use those techniques as a tool for analysis and algorithmic audit (Raji et al. 2020). For more information, see Slack et al. (2020), Lakkaraju and Bastani (2020), and Krishna et al. (2022).

In machine learning literature, explainable methods are divided into model specific and model agnostic. A model-specific explainable method is inherently connected to the used machine learning model such as a random forest or a deep neural network and cannot be used for other models. With a model-agnostic explainable method, a user is free to use any black-box model for data analysis, and the explainable method can be applied to that model. There are two types of model-agnostic methods: local and global. Local methods explain individual predictions and approximate a black-box model in the vicinity of an individual observation. The popular methods include local surrogate models (Ribeiro, Singh, and Guestrin 2016), individual conditional expectation curves (Goldstein et al. 2015), and Shapley values (Lundberg and Lee 2017). A global method describes the average behavior of a black-box model. Partial dependence plots (Friedman 2001), variable importance plots (Breiman 2001; Fisher, Rudin, and Dominici 2019), and global surrogate models (Bastani, Kim, and Bastani 2017) are some of the popular choices.

See [H2OML] **h2omlgraph ice**, [H2OML] **h2omlgraph shapvalues**, and [H2OML] **h2omlgraph shapsummary** for a few local model-agnostic methods and [H2OML] **h2omlgraph pdp** and [H2OML] **h2omlgraph varimp** for global model-agnostic methods. We also describe the global surrogate models in the next section.

Global surrogate models

Global surrogate models (Bastani, Kim, and Bastani 2017 and Craven and Shavlik 1995) are explainable models that approximate the predictions of a black-box model. In other words, a surrogate model uses an interpretable model to explain a black-box model. The steps for obtaining a global surrogate model are straightforward:

1. Obtain predictions from a well-tuned black-box model fit to the testing data.
2. Select and train an interpretable model (for example, a decision tree) for predictions on the testing data.
3. Measure the goodness of fit of the surrogate model for the predictions, and interpret the model.

One way to measure the goodness of fit of a surrogate model for predictions is by using the R^2 for regression and accuracy or log loss for classification,

$$R^2 = 1 - \frac{\sum_{i=1}^{n}\{\hat{g}(\mathbf{x}_i) - \hat{f}(\mathbf{x}_i)\}^2}{\sum_{i=1}^{n}\{\hat{f}(\mathbf{x}_i) - \overline{f}\}^2}$$

where $\hat{g}(\cdot)$ and $\hat{f}(\cdot)$ are the respective predictions from the surrogate and black-box models and $\overline{f}$ is the mean of the black-box predictions. The larger the R^2, the better the surrogate model replicates the black-box model.

For example, suppose we used a GBM to obtain predictions of housing prices. We could then apply the above method to explain its predictions by using a decision tree as a surrogate model. We show one such tree below. We can easily see how the predictors explain the predicted log sales prices. The terminal nodes of the tree show the predicted logarithm of the sales prices. For example, the houses with overall quality (`overallqual`) greater than 7.5 and with the lot area (`lotarea`) greater than 12,332.5 square feet have the highest predicted price of 12.74.

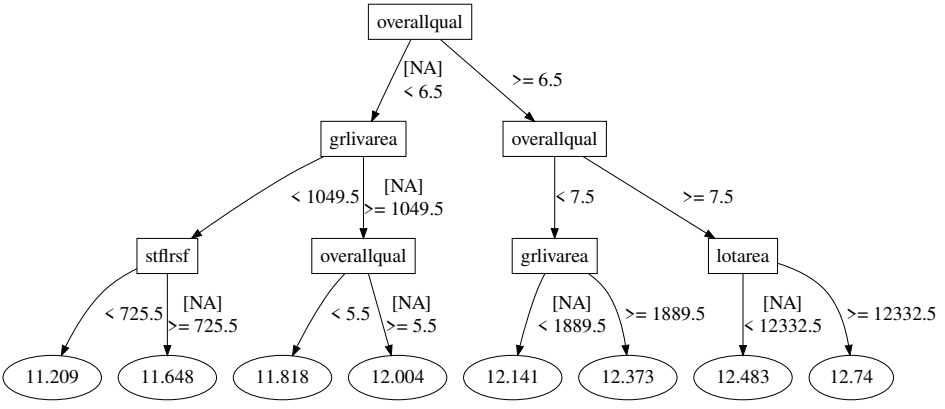

Figure 12.

References

Abreu, S. 2019. Automated architecture design for deep neural networks. arXiv:1908.10714 [cs.LG], https://doi.org/10.48550/arXiv.1908.10714.

Bastani, O., C. Kim, and H. Bastani. 2017. Interpreting blackbox models via model extraction. arXiv:1705.08504 [cs.LG], https://doi.org/10.48550/arXiv.1705.08504.

Belkin, M., D. Hsu, S. Ma, and S. Mandal. 2019. Reconciling modern machine-learning practice and the classical bias–variance trade-off. *Proceedings of the National Academy of Sciences* 116: 15849–15854. https://doi.org/10.1073/pnas.1903070116.

Ben-Haim, Y., and E. Tom-Tov. 2010. A streaming parallel decision tree algorithm. *Journal of Machine Learning Research* 11: 849–872.

Bergstra, J., R. Bardenet, Y. Bengio, and B. Kégl. 2011. "Algorithms for hyper-parameter optimization". In *Proceedings of the 25th International Conference on Neural Information Processing Systems*, edited by J. Shawe-Taylor, R. Zemel, P. Bartlett, F. Pereira, and K. Weinberger, vol. 24: 2546–2554. Red Hook, NY: Curran Associates.

Bergstra, J., and Y. Bengio. 2012. Random search for hyper-parameter optimization. *Journal of Machine Learning Research* 13: 281–305.

Biau, G., and E. Scornet. 2016. A random forest guided tour. *TEST* 25: 197–227. https://doi.org/10.1007/s11749-016-0481-7.

Borisov, V., T. Leemann, K. Seßler, J. Haug, M. Pawelczyk, and G. Kasneci. 2024. Deep neural networks and tabular data: A survey. *IEEE Transactions on Neural Networks and Learning Systems* 35: 7499–7519. https://doi.org/10.1109/tnnls.2022.3229161.

Breiman, L. 1996. Bagging predictors. *Machine Learning* 24: 123–140. https://doi.org/10.1007/BF00058655.

———. 2001. Random forests. *Machine Learning* 45: 5–32. https://doi.org/10.1023/A:1010933404324.

Breiman, L., J. H. Friedman, R. A. Olshen, and C. J. Stone. 1984. *Classification and Regression Trees*. Boca Raton, FL: Chapman and Hall/CRC.

Cawley, G. C., and N. L. C. Talbot. 2010. On over-fitting in model selection and subsequent selection bias in performance evaluation. *Journal of Machine Learning Research* 11: 2079–2107.

Chen, C., K. Lin, C. Rudin, Y. Shaposhnik, S. Wang, and T. Wang. 2018. An interpretable model with globally consistent explanations for credit risk. arXiv:1811.12615 [cs.LG], https://doi.org/10.48550/arXiv.1811.12615.

Chen, T., and C. Guestrin. 2016. "XGBoost: A scalable tree boosting system". In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 785–794. New York: Association for Computing Machinery. https://doi.org/10.1145/2939672.2939785.

Chollet, F. 2021. *Deep Learning with Python*. 2nd ed. Shelter Island, NY: Manning Publications.

Craven, M. W., and J. W. Shavlik. 1995. "Extracting tree-structured representations of trained networks". In *Proceedings of the 9th International Conference on Neural Information Processing Systems*, edited by D. Touretzky, M. C. Mozer, and M. Hasselmo, 24–30. Cambridge, MA: MIT Press.

Curth, A., A. Jeffares, and M. van der Schaar. 2024. Why do random forests work? Understanding tree ensembles as self-regularizing adaptive smoothers. arXiv:2402.01502 [stat.ML], https://doi.org/10.48550/arXiv.2402.01502.

Doshi-Velez, F., and B. Kim. 2017. Towards a rigorous science of interpretable machine learning. arXiv:1702.08608 [stat.ML], https://doi.org/10.48550/arXiv.1702.08608.

Efron, B. 1979. Bootstrap methods: Another look at the jackknife. *Annals of Statistics* 7: 1–26. https://doi.org/10.1214/aos/1176344552.

Efron, B., and R. J. Tibshirani. 1993. *An Introduction to the Bootstrap*. New York: Chapman and Hall/CRC. https://doi.org/10.1201/9780429246593.

Elith, J., J. R. Leathwick, and T. J. Hastie. 2008. A working guide to boosted regression trees. *Journal of Animal Ecology* 77: 802–813. https://doi.org/10.1111/j.1365-2656.2008.01390.x.

Elsken, T., J. H. Metzen, and F. Hutter. 2019. Neural architecture search: A survey. *Journal of Machine Learning Research* 20: 1–21.

Fisher, A., C. Rudin, and F. Dominici. 2019. All models are wrong, but many are useful: Learning a variable's importance by studying an entire class of prediction models simultaneously. *Journal of Machine Learning Research* 20: 1–81.

Freund, Y., and R. E. Schapire. 1997. A decision-theoretic generalization of on-line learning and an application to boosting. *Journal of Computer and System Sciences* 55: 119–139. https://doi.org/10.1006/jcss.1997.1504.

Friedman, J. H. 2001. Greedy function approximation: A gradient boosting machine. *Annals of Statistics* 29: 1189–1232. https://doi.org/10.1214/aos/1013203451.

Friedman, J. H., T. J. Hastie, and R. J. Tibshirani. 2000. Additive logistic regression: A statistical view of boosting. *Annals of Statistics* 28: 337–407. https://doi.org/10.1214/aos/1016218223.

Friedman, J. H., and B. E. Popescu. 2008. Predictive learning via rule ensembles. *Annals of Applied Statistics* 2: 916–954. https://doi.org/10.1214/07-AOAS148.

Ghassemi, M., L. Oakden-Rayner, and A. L. Beam. 2021. The false hope of current approaches to explainable artificial intelligence in health care. *Lancet* 3: E745–E750.

Goldstein, A., A. Kapelner, J. Bleich, and E. Pitkin. 2015. Peeking inside the black box: Visualizing statistical learning with plots of individual conditional expectation. *Journal of Computational and Graphical Statistics* 24: 44–65. https://doi.org/10.1080/10618600.2014.907095.

Guidotti, R., A. Monreale, S. Ruggieri, F. Turini, D. Pedreschi, and F. Giannotti. 2018. A survey of methods for explaining black box models. arXiv:1802.01933 [cs.CY], https://doi.org/10.48550/arXiv.1802.01933.

Hastie, T. J., R. J. Tibshirani, and J. H. Friedman. 2009. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. 2nd ed. New York: Springer. https://doi.org/10.1007/978-0-387-84858-7.

Izenman, A. J. 2008. *Modern Multivariate Statistical Techniques: Regression, Classification, and Manifold Learning*. New York: Springer. https://doi.org/10.1007/978-0-387-78189-1.

James, G., D. Witten, T. J. Hastie, and R. J. Tibshirani. 2021. *An Introduction to Statistical Learning: With Applications in R*. 2nd ed. New York: Springer. https://doi.org/10.1007/978-1-0716-1418-1.

Ke, G., Q. Meng, T. Finley, T. Wang, W. Chen, W. Ma, Q. Ye, and T.-Y. Liu. 2017. "LightGBM: A Highly Efficient Gradient Boosting Decision Tree". In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, vol. 30: 3149–3157. Red Hook, NY: Curran Associates.

Krishna, S., T. Han, A. Gu, S. Wu, S. Jabbari, and H. Lakkaraju. 2022. The disagreement problem in explainable machine learning: A practitioner's perspective. arXiv:2202.01602 [cs.LG], https://doi.org/10.48550/arXiv.2202.01602.

Kuhn, M., and K. Johnson. 2013. *Applied Predictive Modeling*. New York: Springer. https://doi.org/10.1007/978-1-4614-6849-3.

Lakkaraju, H., S. H. Bach, and J. Leskovec. 2016. "Interpretable decision sets: A joint framework for description and prediction". In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 1675–1684. New York: Association for Computing Machinery. https://doi.org/10.1145/2939672.2939874.

Lakkaraju, H., and O. Bastani. 2020. ""How do I fool you?": Manipulating user trust via misleading black box explanations". In *Proceedings of the AAAI/ACM Conference on AI, Ethics, and Society*, 79–85. New York: Association for Computing Machinery. https://doi.org/10.1145/3375627.3375833.

Letham, B., C. Rudin, T. H. McCormick, and D. Madigan. 2015. Interpretable classifiers using rules and Bayesian analysis: Building a better stroke prediction model. *Annals of Applied Statistics* 9: 1350–1371. http://doi.org/10.1214/15-AOAS848.

Lin, Y., and Y. Jeon. 2006. Random forests and adaptive nearest neighbors. *Journal of the American Statistical Association* 101: 578–590. https://doi.org/10.1198/016214505000001230.

Lones, M. A. 2021. How to avoid machine learning pitfalls: A guide for academic researchers. arXiv:2108.02497 [cs.LG], https://doi.org/10.48550/arXiv.2108.02497.

Lundberg, S. M., and S. Lee. 2017. "A unified approach to interpreting model predictions". In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, vol. 30: 4768–4777. Red Hook, NY: Curran Associates.

Meinshausen, N. 2006. Quantile regression forests. *Journal of Machine Learning Research* 7: 983–999.

Potharst, R., and A. J. Feelders. 2002. Classification trees for problems with monotonicity constraints. *ACM SIGKDD Explorations Newsletter* 4: 1–10. https://doi.org/10.1145/568574.568577.

Quinlan, J. R. 1993. *C4.5: Programs for Machine Learning.* San Mateo: Morgan Kaufmann.

Raji, I. D., A. Smart, R. N. White, M. Mitchell, T. Gebru, B. Hutchinson, J. Smith-Loud, D. Theron, and P. Barnes. 2020. "Closing the AI accountability gap: Defining an end-to-end framework for internal algorithmic auditing". In *Proceedings of the 2020 Conference on Fairness, Accountability, and Transparency*, 33–44. New York: Association for Computing Machinery. https://doi.org/10.1145/3351095.3372873.

Raschka, S. 2020. Model evaluation, model selection, and algorithm selection in machine learning. arXiv:1811.12808 [cs.LG], https://doi.org/10.48550/arXiv.1811.12808.

Ribeiro, M. T., S. Singh, and C. Guestrin. 2016. ""Why should I trust you?": Explaining the predictions of any classifier". In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 1135–1144. New York: Association for Computing Machinery. https://doi.org/10.1145/2939672.2939778.

Rifkin, R., and A. Klautau. 2004. In defense of one-vs-all classification. *Journal of Machine Learning Research* 5: 101–141.

Rudin, C. 2019. Stop explaining black box machine learning models for high stakes decisions and use interpretable models instead. *Nature Machine Intelligence* 1: 206–215. https://doi.org/10.1038/s42256-019-0048-x.

Rudin, C., and B. Ustun. 2018. Optimized scoring systems: Toward trust in machine learning for healthcare and criminal justice. *INFORMS Journal on Applied Analytics* 48: 399–486. https://doi.org/10.1287/inte.2018.0957.

Schapire, R. E. 2003. "The boosting approach to machine learning: An overview". In *Nonlinear Estimation and Classification.* Lecture Notes in Statistics, edited by D. D. Denison, M. H. Hansen, C. C. Holmes, B. Mallick, and B. Yu, vol. 171: 149–171. New York: Springer. https://doi.org/10.1007/978-0-387-21579-2_9.

Schapire, R. E., and Y. Freund. 2012. *Boosting: Foundations and Algorithms.* Cambridge, MA: MIT Press.

Shmuel, A., O. Glickman, and T. Lazebnik. 2024. A comprehensive benchmark of machine and deep learning across diverse tabular datasets. arXiv:2408.14817 [cs.LG], https://doi.org/10.48550/arXiv.2408.14817.

Shwartz-Ziv, R., and A. Armon. 2022. Tabular data: Deep learning is not all you need. *Information Fusion* 81: 84–90. https://doi.org/10.1016/j.inffus.2021.11.011.

Slack, D., S. Hilgard, E. Jia, S. Singh, and H. Lakkaraju. 2020. "Fooling LIME and SHAP: Adversarial attacks on post hoc explanation methods". In *Proceedings of the AAAI/ACM Conference on AI, Ethics, and Society*, 180–186. New York: Association for Computing Machinery. https://doi.org/10.1145/3375627.3375830.

Wager, S., and S. Athey. 2018. Estimation and inference of heterogeneous treatment effects using random forests. *Journal of the American Statistical Association* 113: 1228–1242. https://doi.org/10.1080/01621459.2017.1319839.

Wyner, A. J., M. Olson, J. Bleich, and D. Mease. 2017. Explaining the success of AdaBoost and random forests as interpolating classifiers. *Journal of Machine Learning Research* 18: 1–33.

Yang, L., and A. Shami. 2020. On hyperparameter optimization of machine learning algorithms: Theory and practice. *Neurocomputing* 415: 295–316. https://doi.org/10.1016/j.neucom.2020.07.061.

Also see

[H2OML] **h2oml** — Introduction to commands for Stata integration with H2O machine learning

[H2OML] **Glossary**