18 Trees, Forests, Bagging, and Boosting

18.1 Classification and regression trees (CART)

Classification and regression trees or **CART** models [BFO84], also called **decision trees** [Qui86; Qui93], are defined by recursively partitioning the input space, and defining a local model in each resulting region of input space. The overall model can be represented by a tree, with one leaf per region, as we explain below.

18.1.1 Model definition

We start by considering regression trees, where all inputs are real-valued. The tree consists of a set of nested decision rules. At each node i, the feature dimension d_i of the input vector $\boldsymbol{x}$ is compared to a threshold value t_i, and the input is then passed down to the left or right branch, depending on whether it is above or below threshold. At the leaves of the tree, the model specifies the predicted output for any input that falls into that part of the input space.

For example, consider the regression tree in Figure 18.1(a). The first node asks if x_1 is less than some threshold t_1. If yes, we then ask if x_2 is less than some other threshold t_2. If yes, we enter the bottom left leaf node. This corresponds to the region of space defined by

$$R_1 = \{\boldsymbol{x} : x_1 \leq t_1, x_2 \leq t_2\} \tag{18.1}$$

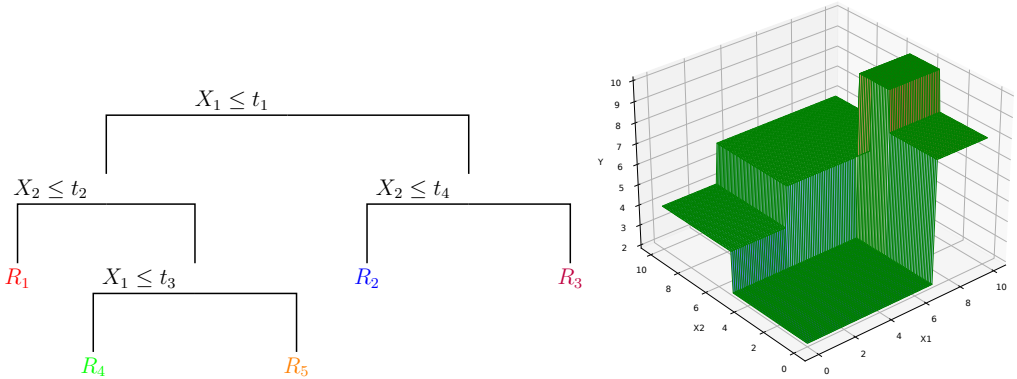
We can associate this region with the predicted output computing using other branches of the tree define different regions in terms of **axis parallel splits**. The overall result is that we partition the 2d input space into 5 regions, as shown in Figure 18.1(b).[1] We can now associate a mean response with each of these regions, resulting in the piecewise constant surface shown in Figure 18.1(b). For example, the output for region 1 can be estimated using

$$w_1 = \frac{\sum_{n=1}^{N} y_n \mathbb{I}\left(\boldsymbol{x}_n \in R_1\right)}{\sum_{n=1}^{N} \mathbb{I}\left(\boldsymbol{x}_n \in R_1\right)} \tag{18.2}$$

Formally, a regression tree can be defined by

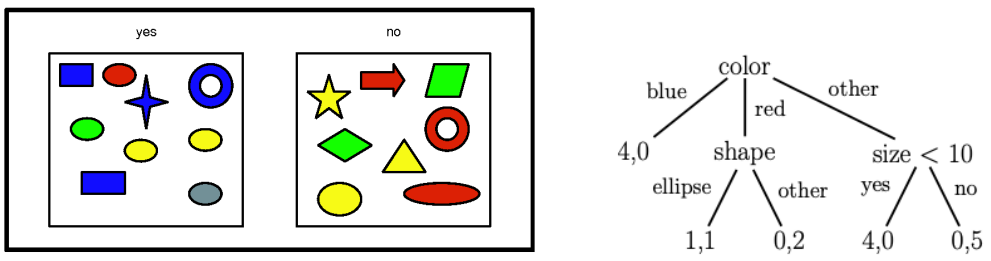
$$f(\boldsymbol{x}; \boldsymbol{\theta}) = \sum_{j=1}^{J} w_j \mathbb{I}\left(\boldsymbol{x} \in R_j\right) \tag{18.3}$$

1. By using enough splits (i.e., deep enough trees), we can make a piecewise linear approximation to decision boundaries with more complex shapes, but it may require a lot of data to fit such a model.



(a) (b)

Figure 18.1: (a) A regression tree on two inputs. (b) Corresponding piecewise constant surface. Adapted from Figure 9.2 of [HTF09]. Generated by regtreeSurfaceDemo.ipynb.



(a) (b)

Figure 18.2: (a) A set of shapes with corresponding binary labels. The features are: color (values "blue", "red", "other"), shape (values "ellipse", "other"), and size (real-valued). (b) A hypothetical classification tree fitted to this data. A leaf labeled as (n_1, n_0) means that there are n_1 positive examples that fall into this partition, and n_0 negative examples.

where R_j is the region specified by the j'th leaf node, w_j is the predicted output for that node, and $\boldsymbol{\theta} = \{(R_j, w_j) : j = 1 : J\}$, where J is the number of nodes. The regions themselves are defined by the feature dimensions that are used in each split, and the corresponding thresholds, on the path from the root to the leaf. For example, in Figure 18.1(a), we have $R_1 = [(d_1 \leq t_1), (d_2 \leq t_2)]$, $R_2 = [(d_1 \leq t_1), (d_2 > t_2), (d_3 \leq t_3)]$, etc. (For categorical inputs, we can define the splits based on comparing feature d_i to each of the possible values for that feature, rather than comparing to a numeric threshold.) We discuss how to learn these regions in Section 18.1.2.

For classification problems, the leaves contain a distribution over the class labels, rather than just the mean response. See Figure 18.2 for an example of a classification tree.

18.1.2 Model fitting

To fit the model, we need to minimize the following loss:

$$\mathcal{L}(\boldsymbol{\theta}) = \sum_{n=1}^{N} \ell(y_n, f(\boldsymbol{x}_n; \boldsymbol{\theta})) = \sum_{j=1}^{J} \sum_{\boldsymbol{x}_n \in R_j} \ell(y_n, w_j) \tag{18.4}$$

Unfortunately, this is not differentiable, because of the need to learn the discrete tree structure. Indeed, finding the optimal partitioning of the data is NP-complete [HR76]. The standard practice is to use a greedy procedure, in which we iteratively grow the tree one node at a time. This approach is used by CART [BFO84], **C4.5** [Qui93], and **ID3** [Qui86], which are three popular implementations of the method.

The idea is as follows. Suppose we are at node i; let $\mathcal{D}_i = \{(\boldsymbol{x}_n, y_n) \in N_i\}$ be the set of examples that reach this node. We will consider how to split this node into a left branch and right branch so as to minimize the error in each child subtree.

If the j'the feature is a real-valued scalar, we can partition the data at node i by comparing to a threshold t. The set of possible thresholds $\mathcal{T}_j$ for feature j can be obtained by sorting the unique values of $\{x_{nj}\}$. For example, if feature 1 has the values $\{4.5, -12, 72, -12\}$, then we set $\mathcal{T}_1 = \{-12, 4.5, 72\}$. For each possible threshold, we define the left and right splits, $\mathcal{D}_i^L(j,t) = \{(\boldsymbol{x}_n, y_n) \in N_i : x_{n,j} \leq t\}$ and $\mathcal{D}_i^R(j,t) = \{(\boldsymbol{x}_n, y_n) \in N_i : x_{n,j} > t\}$.

If the j'th feature is categorical, with K_j possible values, then we check if the feature is equal to each of those values or not. This defines a set of K_j possible binary splits: $\mathcal{D}_i^L(j,t) = \{(\boldsymbol{x}_n, y_n) \in N_i : x_{n,j} = t\}$ and $\mathcal{D}_i^R(j,t) = \{(\boldsymbol{x}_n, y_n) \in N_i : x_{n,j} \neq t\}$.) (Alternatively, we could allow for a multi-way split, as in Figure 18.2(b). However, this may cause **data fragmentation**, in which too little data might "fall" into each subtree, resulting in overfitting. Therefore it is more common to use binary splits.)

Once we have computed $\mathcal{D}_i^L(j,t)$ and $\mathcal{D}_i^R(j,t)$ for each j and t at node i, we choose the best feature j_i to split on, and the best value for that feature, t_i, as follows:

$$(j_i, t_i) = \arg \min_{j \in \{1,\ldots,D\}} \min_{t \in \mathcal{T}_j} \frac{|\mathcal{D}_i^L(j,t)|}{|\mathcal{D}_i|} c(\mathcal{D}_i^L(j,t)) + \frac{|\mathcal{D}_i^R(j,t)|}{|\mathcal{D}_i|} c(\mathcal{D}_i^R(j,t)) \tag{18.5}$$

We now discuss the cost function $c(\mathcal{D}_i)$ which is used to evaluate the cost of node i. For regression, we can use the mean squared error

$$\text{cost}(\mathcal{D}_i) = \frac{1}{|\mathcal{D}|} \sum_{n \in \mathcal{D}_i} (y_n - \overline{y})^2 \tag{18.6}$$

where $\overline{y} = \frac{1}{|\mathcal{D}|} \sum_{n \in \mathcal{D}_i} y_n$ is the mean of the response variable for examples reaching node i.

For classification, we first compute the empirical distribution over class labels for this node:

$$\hat{\pi}_{ic} = \frac{1}{|\mathcal{D}_i|} \sum_{n \in \mathcal{D}_i} \mathbb{I}(y_n = c) \tag{18.7}$$

Given this, we can then compute the **Gini index**

$$G_i = \sum_{c=1}^{C} \hat{\pi}_{ic}(1 - \hat{\pi}_{ic}) = \sum_c \hat{\pi}_{ic} - \sum_c \hat{\pi}_{ic}^2 = 1 - \sum_c \hat{\pi}_{ic}^2 \tag{18.8}$$

This is the expected error rate. To see this, note that $\hat{\pi}_{ic}$ is the probability a random entry in the leaf belongs to class c, and $1 - \hat{\pi}_{ic}$ is the probability it would be misclassified.

Alternatively we can define cost as the entropy or **deviance** of the node:

$$H_i = \mathbb{H}(\hat{\boldsymbol{\pi}}_i) = -\sum_{c=1}^{C} \hat{\pi}_{ic} \log \hat{\pi}_{ic} \tag{18.9}$$

A node that is **pure** (i.e., only has examples of one class) will have 0 entropy.

Given one of the above cost functions, we can use Equation (18.5) to pick the best feature, and best threshold at each node. We then partition the data, and call the fitting algorithm recursively on each subset of the data.

18.1.3 Regularization

If we let the tree become deep enough, it can achieve 0 error on the training set (assuming no label noise), by partioning the input space into sufficiently small regions where the output is constant. However, this will typically result in overfitting. To prevent this, there are two main approaches. The first is to stop the tree growing process according to some heuristic, such as having too few examples at a node, or reaching a maximum depth. The second approach is to grow the tree to its maximum depth, where no more splits are possible, and then to **prune** it back, by merging split subtrees back into their parent (see e.g., [BA97b]). This can partially overcome the greedy nature of top-down tree growing. (For example, consider applying the top-down approach to the xor data in Figure 13.1: the algorithm would never make any splits, since each feature on its own has no predictive power.) However, forward growing and backward pruning is slower than the greedy top-down approach.

18.1.4 Handling missing input features

In general, it is hard for discriminative models, such as neural networks, to handle missing input features, as we discussed in Section 1.5.5. However, for trees, there are some simple heuristics that can work well.

The standard heuristic for handling missing inputs in decision trees is to look for a series of "backup" variables, which can induce a similar partition to the chosen variable at any given split; these can be used in case the chosen variable is unobserved at test time. These are called **surrogate splits**. This method finds highly correlated features, and can be thought of as learning a local joint model of the input. This has the advantage over a generative model of not modeling the entire joint distribution of inputs, but it has the disadvantage of being entirely ad hoc. A simpler approach, applicable to categorical variables, is to code "missing" as a new value, and then to treat the data as fully observed.

18.1.5 Pros and cons

Tree models are popular for several reasons:

- They are easy to interpret.
- They can easily handle mixed discrete and continuous inputs.

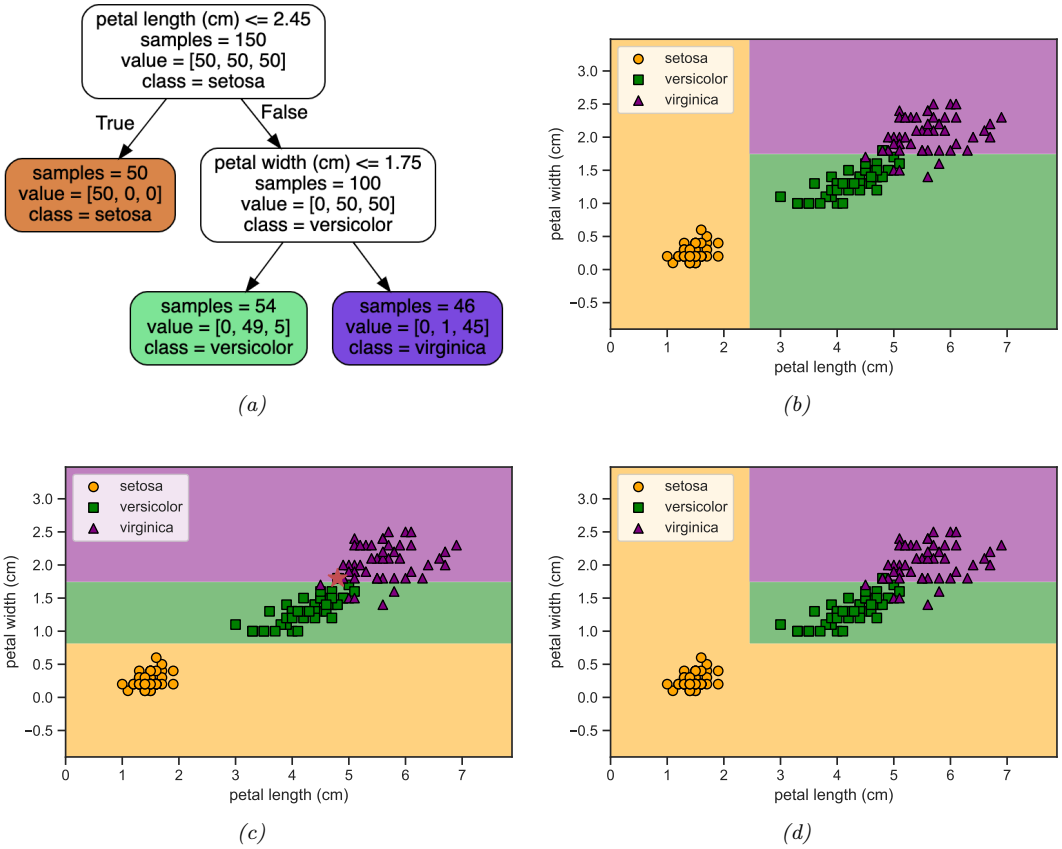


Figure 18.3: (a) A decision tree of depth 2 fit to the iris data, using just the petal length and petal width features. Leaf nodes are color coded according to the majority class. The number of training samples that pass from the root to each node is shown inside each box, as well as how many of these values fall into each class. This can be normalized to get a distribution over class labels for each node. (b) Decision surface induced by (a). (c) Fit to data where we omit a single data point (shown by red star). (d) Ensemble of the two models in (b) and (c). Generated by dtree_sensitivity.ipynb.

- They are insensitive to monotone transformations of the inputs (because the split points are based on ranking the data points), so there is no need to standardize the data.
- They perform automatic variable selection.
- They are relatively robust to outliers.
- They are fast to fit, and scale well to large data sets.
- They can handle missing input features.

However, tree models also have some disadvantages. The primary one is that they do not predict very accurately compared to other kinds of model. This is in part due to the greedy nature of the tree construction algorithm.

A related problem is that trees are **unstable**: small changes to the input data can have large effects on the structure of the tree, due to the hierarchical nature of the tree-growing process, causing errors at the top to affect the rest of the tree. For example, consider the tree in Figure 18.3b. Omitting even a single data point from the training set can result in a dramatically different decision surface, as shown in Figure 18.3c, due to the use of axis parallel splits. (Omitting features can also cause instability.) In Section 18.3 and Section 18.4, we will turn this instability into a virtue.

18.2 Ensemble learning

In Section 18.1, we saw that decision trees can be quite unstable, in the sense that their predictions might vary a lot if the training data is perturbed. In other words, decision trees are a high variance estimator. A simple way to reduce variance is to average multiple models. This is called **ensemble learning**. The result model has the form

$$f(y|\boldsymbol{x}) = \frac{1}{|\mathcal{M}|} \sum_{m \in \mathcal{M}} f_m(y|\boldsymbol{x}) \tag{18.10}$$

where f_m is the m'th base model. The ensemble will have similar bias to the base models, but lower variance, generally resulting in improved overall performance (see Section 4.7.6.3 for details on the bias-variance tradeoff).

Averaging is a sensible way to combine predictions from regression models. For classifiers, it can sometimes be better to take a majority vote of the outputs. (This is sometimes called a **committee method**.) To see why this can help, suppose each base model is a binary classifier with an accuracy of θ, and suppose class 1 is the correct class. Let $Y_m \in \{0, 1\}$ be the prediction for the m'th model, and let $S = \sum_{m=1}^{M} Y_m$ be the number of votes for class 1. We define the final predictor to be the majority vote, i.e., class 1 if $S > M/2$ and class 0 otherwise. The probability that the ensemble will pick class 1 is

$$p = \Pr(S > M/2) = 1 - B(M/2, M, \theta) \tag{18.11}$$

where $B(x, M, \theta)$ is the cdf of the binomial distribution with parameters M and θ evaluated at x. For $\theta = 0.51$ and $M = 1000$, we get $p = 0.73$ and with $M = 10,000$ we get $p = 0.97$.

The performance of the voting approach is dramatically improved, because we assumed each predictor made independent errors. In practice, their mistakes may be correlated, but as long as we ensemble sufficiently diverse models, we can still come out ahead.

18.2.1 Stacking

An alternative to using an unweighted average or majority vote is to learn how to combine the base models, by using

$$f(y|\boldsymbol{x}) = \sum_{m \in \mathcal{M}} w_m f_m(y|\boldsymbol{x}) \tag{18.12}$$

This is called **stacking**, which stands for "stacked generalization" [Wol92]. Note that the combination weights used by stacking need to be trained on a separate dataset, otherwise they would put all their mass on the best performing base model.

18.2.2 Ensembling is not Bayes model averaging

It is worth noting that an ensemble of models is not the same as using Bayes model averaging over models (Section 4.6), as pointed out in [Min00]. An ensemble considers a larger hypothesis class of the form

$$p(y|\boldsymbol{x}, \boldsymbol{w}, \boldsymbol{\theta}) = \sum_{m \in \mathcal{M}} w_m p(y|\boldsymbol{x}, \boldsymbol{\theta}_m) \tag{18.13}$$

whereas BMA uses

$$p(y|\boldsymbol{x}, \mathcal{D}) = \sum_{m \in \mathcal{M}} p(m|\mathcal{D}) p(y|\boldsymbol{x}, m, \mathcal{D}) \tag{18.14}$$

The key difference is that in the case of BMA, the weights $p(m|\mathcal{D})$ sum to one, and in the limit of infinite data, only a single model will be chosen (namely the MAP model). By contrast, the ensemble weights w_m are arbitrary, and don't collapse in this way to a single model.

18.3 Bagging

In this section, we discuss **bagging** [Bre96], which stands for "bootstrap aggregating". This is a simple form of ensemble learning in which we fit M different base models to different randomly sampled versions of the data; this encourages the different models to make diverse predictions. The datasets are sampled with replacement (a technique known as bootstrap sampling, Section 4.7.3), so a given example may appear multiple times, until we have a total of N examples per model (where N is the number of original data points).

The disadvantage of bootstrap is that each base model only sees, on average, 63% of the unique input examples. To see why, note that the chance that a single item will not be selected from a set of size N in any of N draws is $(1 - 1/N)^N$. In the limit of large N, this becomes $e^{-1} \approx 0.37$, which means only $1 - 0.37 = 0.63$ of the data points will be selected.

The 37% of the training instances that are not used by a given base model are called **out-of-bag instances** (oob). We can use the predicted performance of the base model on these oob instances as an estimate of test set performance. This provides a useful alternative to cross validation.

The main advantage of bootstrap is that it prevents the ensemble from relying too much on any individual training example, which enhances robustness and generalization [Gra04]. For example, comparing Figure 18.3b and Figure 18.3c, we see that omitting a single example from the training set can have a large impact on the decision tree that we learn (even though the tree growing algorithm is otherwise deterministic). By averaging the predictions from both of these models, we get the more reasonable prediction model in Figure 18.3d. This advantage generally increases with the size of the ensemble, as shown in Figure 18.4. (Of course, larger ensembles take more memory and more time.)

Bagging does not always improve performance. In particular, it relies on the base models being unstable estimators, so that omitting some of the data significantly changes the resulting model fit.

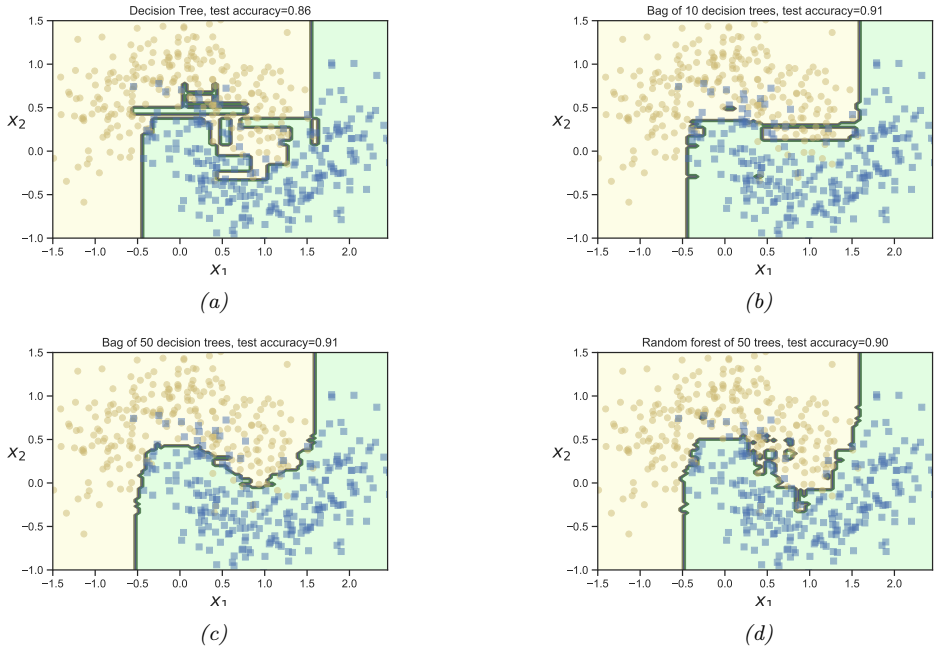


Figure 18.4: (a) A single decision tree. (b-c) Bagging ensemble of 10 and 50 trees. (d) Random forest of 50 trees. Adapted from Figure 7.5 of [Gér19]. Generated by bagging_trees.ipynb and rf_demo_2d.ipynb.

This is the case for decision trees, but not for other models, such as nearest neighbor classifiers. For neural networks, the story is more mixed. They can be unstable wrt their training set. On the other hand, deep networks will underperform if they only see 63% of the data, so bagged DNNs do not usually work well [NTL20].

18.4 Random forests

Bagging relies on the assumption that re-running the same learning algorithm on different subsets of the data will result in sufficiently diverse base models. The technique known as **random forests** [Bre01] tries to decorrelate the base learners even further by learning trees based on a randomly chosen subset of input variables (at each node of the tree), as well as a randomly chosen subset of data cases. It does this by modifying Equation (18.5) so the the feature split dimension j is optimized over a random subset of the features, $S_i \subset \{1, \ldots, D\}$.

For example, consider the email spam dataset [HTF09, p301]. This dataset contains 4601 email messages, each of which is classified as spam (1) or non-spam (0). The data was open sourced by George Forman from Hewlett-Packard (HP) Labs.

There are 57 quantitative (real-valued) features, as follows:

- 48 features corresponding to the percentage of words in the email that match a given word, such as "remove" or "labs".

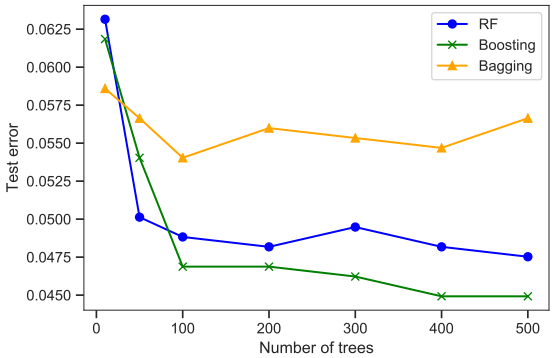


Figure 18.5: Preditive accuracy vs size of tree ensemble for bagging, random forests and gradient boosting with log loss. Adapted from Figure 15.1 of [HTF09]. Generated by spam_tree_ensemble_compare.ipynb.

- 6 features corresponding to the percentage of characters in the email that match a given character, namely `; . [ ! $ #`
- 3 features corresponding to the average length, max length, and sum of lengths of uninterrupted sequences of capital letters. (These features are called CAPAVE, CAPMAX and CAPTOT.)

Figure 18.5 shows that random forests work much better than bagged decision trees, because many input features are irrelevant. (We also see that a method called "boosting", discussed in Section 18.5, works even better; however, this requires sequentially fitting trees, whereas random forests can be fit in parallel.)

18.5 Boosting

Ensembles of trees, whether fit by bagging or the random forest algorithm, corresponding to a model of the form

$$f(\boldsymbol{x};\boldsymbol{\theta}) = \sum_{m=1}^{M} \beta_m F_m(\boldsymbol{x};\boldsymbol{\theta}_m) \tag{18.15}$$

where F_m is the m'th tree, and β_m is the corresponding weight, often set to $\beta_m = 1/M$. We can generalize this by allowing the F_m functions to be general function approximators, such as neural networks, not just trees. The result is called an **additive model** [HTF09]. We can think of this as a linear model with **adaptive basis functions**. The goal, as usual, is to minimize the empirical loss (with an optional regularizer):

$$\mathcal{L}(f) = \sum_{i=1}^{N} \ell(y_i, f(\boldsymbol{x}_i)) \tag{18.16}$$

Boosting [Sch90; FS96] is an algorithm for sequentially fitting additive models where each F_m is a binary classifier that returns $F_m \in \{-1, +1\}$. In particular, we first fit F_1 on the original data,

and then we weight the data samples by the errors made by F_1, so misclassified examples get more weight. Next we fit F_2 to this weighted data set. We keep repeating this process until we have fit the desired number M of components. (M is a hyper-parameter that controls the complexity of the overall model, and can be chosen by monitoring performance on a validation set, and using early stopping.)

It can be shown that, as long as each F_m has an accuracy that is better than chance (even on the weighted dataset), then the final ensemble of classifiers will have higher accuracy than any given component. That is, if F_m is a **weak learner** (so its accuracy is only slightly better than 50%), then we can boost its performance using the above procedure so that the final f becomes a **strong learner**. (See e.g., [SF12] for more details on the learning theory approach to boosting.)

Note that boosting reduces the bias of the strong learner, by fitting trees that depend on each other, whereas bagging and RF reduce the variance by fitting independent trees. In many cases, boosting can work better. See Figure 18.5 for an example.

The original boosting algorithm focused on binary classification with a particular loss function that we will explain in Section 18.5.3, and was derived from the PAC learning theory framework (see Section 5.4.4). In the rest of this section, we focus on a more statistical version of boosting, due to [FHT00; Fri01], which works with arbitrary loss functions, making the method suitable for regression, multi-class classification, ranking, etc. Our presentation is based on [HTF09, ch10] and [BH07], which should be consulted for further details.

18.5.1 Forward stagewise additive modeling

In this section, we discuss **forward stagewise additive modeling**, in which we sequentially optimize the objective in Equation (18.16) for general (differentiable) loss functions, where f is an additive model as in Equation 18.15. That is, at iteration m, we compute

$$(\beta_m, \boldsymbol{\theta}_m) = \underset{\beta, \boldsymbol{\theta}}{\operatorname{argmin}} \sum_{i=1}^{N} \ell(y_i, f_{m-1}(\boldsymbol{x}_i) + \beta F(\boldsymbol{x}_i; \boldsymbol{\theta})) \tag{18.17}$$

We then set

$$f_m(\boldsymbol{x}) = f_{m-1}(\boldsymbol{x}) + \beta_m F(\boldsymbol{x}; \boldsymbol{\theta}_m) = f_{m-1}(\boldsymbol{x}) + \beta_m F_m(\boldsymbol{x}) \tag{18.18}$$

(Note that we do not adjust the parameters of previously added models.) The details on how to perform this optimization step depend on the loss function that we choose, and (in some cases) on the form of the weak learner F, as we discuss below.

18.5.2 Quadratic loss and least squares boosting

Suppose we use squared error loss, $\ell(y, \hat{y}) = (y - \hat{y})^2$. In this case, the i'th term in the objective at step m becomes

$$\ell(y_i, f_{m-1}(\boldsymbol{x}_i) + \beta F(\boldsymbol{x}_i; \boldsymbol{\theta})) = (y_i - f_{m-1}(\boldsymbol{x}_i) - \beta F(\boldsymbol{x}_i; \boldsymbol{\theta}))^2 = (r_{im} - \beta F(\boldsymbol{x}_i; \boldsymbol{\theta}))^2 \tag{18.19}$$

where $r_{im} = y_i - f_{m-1}(\boldsymbol{x}_i)$ is the residual of the current model on the i'th observation. We can minimize the above objective by simply setting $\beta = 1$, and fitting F to the residual errors. This is called **least squares boosting** [BY03].

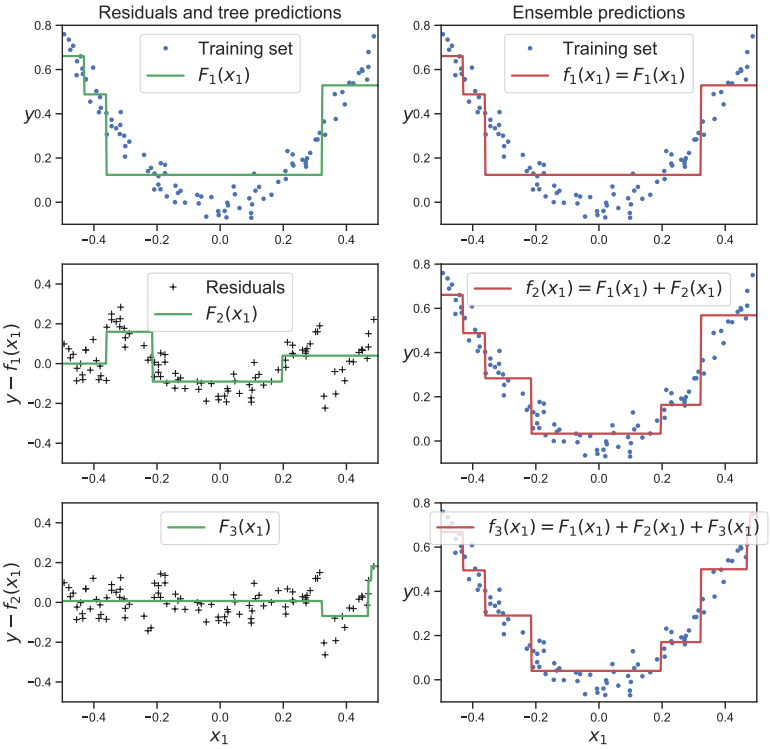


Figure 18.6: Illustration of boosting using a regression tree of depth 2 applied to a 1d dataset. Adapted from Figure 7.9 of [Gér19]. Generated by boosted_regr_trees.ipynb.

We give an example of this process in Figure 18.6, where we use a regression tree of depth 2 as the weak learner. On the left, we show the result of fitting the weak learner to the residuals, and on the right, we show the current strong learner. We see how each new weak learner that is added to the ensemble corrects the errors made by earlier versions of the model.

18.5.3 Exponential loss and AdaBoost

Suppose we are interested in binary classification, i.e., predicting $\tilde{y}_i \in \{-1, +1\}$. Let us assume the weak learner computes

$$p(y = 1|\boldsymbol{x}) = \frac{e^{F(\boldsymbol{x})}}{e^{-F(\boldsymbol{x})} + e^{F(\boldsymbol{x})}} = \frac{1}{1 + e^{-2F(\boldsymbol{x})}} \tag{18.20}$$

so $F(\boldsymbol{x})$ returns half the log odds. We know from Equation (10.13) that the negative log likelihood is given by

$$\ell(\tilde{y}, F(\boldsymbol{x})) = \log(1 + e^{-2\tilde{y}F(\boldsymbol{x})}) \tag{18.21}$$

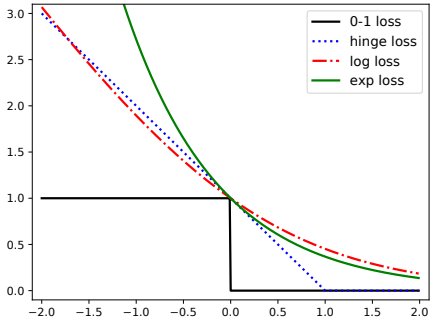


Figure 18.7: Illustration of various loss functions for binary classification. The horizontal axis is the margin $m(\boldsymbol{x}) = \tilde{y}F(\boldsymbol{x})$, the vertical axis is the loss. The log loss uses log base 2. Generated by hinge_loss_plot.ipynb.

We can minimize this by ensuring that the **margin** $m(\boldsymbol{x}) = \tilde{y}F(\boldsymbol{x})$ is as large as possible. We see from Figure 18.7 that the log loss is a smooth upper bound on the 0-1 loss. We also see that it penalizes negative margins more heavily than positive ones, as desired (since positive margins are already correctly classified).

However, we can also use other loss functions. In this section, we consider the **exponential loss**

$$\ell(\tilde{y}, F(\boldsymbol{x})) = \exp(-\tilde{y}F(\boldsymbol{x})) \tag{18.22}$$

We see from Figure 18.7 that this is also a smooth upper bound on the 0-1 loss. In the population setting (with infinite sample size), the optimal solution to the exponential loss is the same as for log loss. To see this, we can just set the derivative of the expected loss (for each $\boldsymbol{x}$) to zero:

$$\frac{\partial}{\partial F(\boldsymbol{x})}\mathbb{E}\left[e^{-\tilde{y}f(\boldsymbol{x})}|\boldsymbol{x}\right] = \frac{\partial}{\partial F(\boldsymbol{x})}[p(\tilde{y}=1|\boldsymbol{x})e^{-F(\boldsymbol{x})} + p(\tilde{y}=-1|\boldsymbol{x})e^{F(\boldsymbol{x})}] \tag{18.23}$$

$$= -p(\tilde{y}=1|\boldsymbol{x})e^{-F(\boldsymbol{x})} + p(\tilde{y}=-1|\boldsymbol{x})e^{F(\boldsymbol{x})} \tag{18.24}$$

$$= 0 \Rightarrow \frac{p(\tilde{y}=1|\boldsymbol{x})}{p(\tilde{y}=-1|\boldsymbol{x})} = e^{2F(\boldsymbol{x})} \tag{18.25}$$

However, it turns out that the exponential loss is easier to optimize in the boosting setting, as we show below. (We consider the log loss case in Section 18.5.4.)

We now discuss how to solve for the m'th weak learner, F_m, when we use exponential loss. We will assume that the base classifier F_m returns a binary class label; the resulting algorithm is called **discrete AdaBoost** [FHT00]. If F_m returns a probability instead, a modified algorithm, known as **real AdaBoost**, can be used [FHT00].

At step m we have to minimize

$$L_m(F) = \sum_{i=1}^{N} \exp[-\tilde{y}_i(f_{m-1}(\boldsymbol{x}_i) + \beta F(\boldsymbol{x}_i))] = \sum_{i=1}^{N} \omega_{i,m} \exp(-\beta \tilde{y}_i F(\boldsymbol{x}_i)) \tag{18.26}$$

where $\omega_{i,m} \triangleq \exp(-\tilde{y}_i f_{m-1}(\boldsymbol{x}_i))$ is a weight applied to datacase i, and $\tilde{y}_i \in \{-1, +1\}$. We can rewrite this objective as follows:

$$L_m = e^{-\beta} \sum_{\tilde{y}_i = F(\boldsymbol{x}_i)} \omega_{i,m} + e^{\beta} \sum_{\tilde{y}_i \neq F(\boldsymbol{x}_i)} \omega_{i,m} \tag{18.27}$$

$$= (e^{\beta} - e^{-\beta}) \sum_{i=1}^{N} \omega_{i,m} \mathbb{I}(\tilde{y}_i \neq F(\boldsymbol{x}_i)) + e^{-\beta} \sum_{i=1}^{N} \omega_{i,m} \tag{18.28}$$

Consequently the optimal function to add is

$$F_m = \operatorname*{argmin}_F \sum_{i=1}^{N} \omega_{i,m} \mathbb{I}(\tilde{y}_i \neq F(\boldsymbol{x}_i)) \tag{18.29}$$

This can be found by applying the weak learner to a weighted version of the dataset, with weights $\omega_{i,m}$.

All that remains is to solve for the size of the update, β. Subsituting F_m into L_m and solving for β we find

$$\beta_m = \frac{1}{2} \log \frac{1 - \mathrm{err}_m}{\mathrm{err}_m} \tag{18.30}$$

where

$$\mathrm{err}_m = \frac{\sum_{i=1}^{N} \omega_{i,m} \mathbb{I}(\tilde{y}_i \neq F_m(\boldsymbol{x}_i))}{\sum_{i=1}^{N} \omega_{i,m}} \tag{18.31}$$

Therefore overall update becomes

$$f_m(\boldsymbol{x}) = f_{m-1}(\boldsymbol{x}) + \beta_m F_m(\boldsymbol{x}) \tag{18.32}$$

After updating the strong learner, we need to recompute the weights for the next iteration, as follows:

$$\omega_{i,m+1} = e^{-\tilde{y}_i f_m(\boldsymbol{x}_i)} = e^{-\tilde{y}_i f_{m-1}(\boldsymbol{x}_i) - \tilde{y}_i \beta_m F_m(\boldsymbol{x}_i)} = \omega_{i,m} e^{-\tilde{y}_i \beta_m F_m(\boldsymbol{x}_i)} \tag{18.33}$$

If $\tilde{y}_i = F_m(\boldsymbol{x}_i)$, then $\tilde{y}_i F_m(\boldsymbol{x}_i) = 1$, and if $\tilde{y}_i \neq F_m(\boldsymbol{x}_i)$, then $\tilde{y}_i F_m(\boldsymbol{x}_i) = -1$. Hence $-\tilde{y}_i F_m(\boldsymbol{x}_i) = 2\mathbb{I}(\tilde{y}_i \neq F_m(\boldsymbol{x}_o)) - 1$, so the update becomes

$$\omega_{i,m+1} = \omega_{i,m} e^{\beta_m (2\mathbb{I}(\tilde{y}_i \neq F_m(\boldsymbol{x}_i)) - 1)} = \omega_{i,m} e^{2\beta_m \mathbb{I}(\tilde{y}_i \neq F_m(\boldsymbol{x}_i))} e^{-\beta_m} \tag{18.34}$$

Since the $e^{-\beta_m}$ is constant across all examples, it can be dropped. If we then define $\alpha_m = 2\beta_m$, the update becomes

$$\omega_{i,m+1} = \begin{cases} \omega_{i,m} e^{\alpha_m} & \text{if } \tilde{y}_i \neq F_m(\boldsymbol{x}_i) \\ \omega_{i,m} & \text{otherwise} \end{cases} \tag{18.35}$$

Thus we see that we exponentially increase weights of misclassified examples. The resulting algorithm shown in Algorithm 8, and is known as **Adaboost.M1** [FS96].

A multiclass generalization of exponential loss, and an adaboost-like algorithm to minimize it, known as **SAMME** (stagewise additive modeling using a multiclass exponential loss function), is described in [Has+09]. This is implemented in scikit learn (the AdaBoostClassifier class).

Algorithm 8: Adaboost.M1, for binary classification with exponential loss

$\omega_i = 1/N$
for $m = 1 : M$ **do**
  Fit a classifier $F_m(\boldsymbol{x})$ to the training set using weights $\boldsymbol{w}$
  Compute $\text{err}_m = \frac{\sum_{i=1}^N \omega_{i,m} \mathbb{I}(\tilde{y}_i \neq F_m(\boldsymbol{x}_i))}{\sum_{i=1}^N \omega_{i,m}}$
  Compute $\alpha_m = \log[(1 - \text{err}_m)/\text{err}_m]$
  Set $\omega_i \leftarrow \omega_i \exp[\alpha_m \mathbb{I}(\tilde{y}_i \neq F_m(\boldsymbol{x}_i))]$
Return $f(\boldsymbol{x}) = \text{sgn}\left[\sum_{m=1}^M \alpha_m F_m(\boldsymbol{x})\right]$

18.5.4 LogitBoost

The trouble with exponential loss is that it puts a lot of weight on misclassified examples, as is apparent from the exponential blowup on the left hand side of Figure 18.7. This makes the method very sensitive to outliers (mislabeled examples). In addition, $e^{-\tilde{y}f}$ is not the logarithm of any pmf for binary variables $\tilde{y} \in \{-1, +1\}$; consequently we cannot recover probability estimates from $f(\boldsymbol{x})$.

A natural alternative is to use log loss, as we discussed in Section 18.5.3. This only punishes mistakes linearly, as is clear from Figure 18.7. Furthermore, it means that we will be able to extract probabilities from the final learned function, using

$$p(y = 1|\boldsymbol{x}) = \frac{e^{f(\boldsymbol{x})}}{e^{-f(\boldsymbol{x})} + e^{f(\boldsymbol{x})}} = \frac{1}{1 + e^{-2f(\boldsymbol{x})}} \tag{18.36}$$

The goal is to minimze the expected log-loss, given by

$$L_m(F) = \sum_{i=1}^N \log\left[1 + \exp\left(-2\tilde{y}_i(f_{m-1}(\boldsymbol{x}) + F(\boldsymbol{x}_i))\right)\right] \tag{18.37}$$

By performing a Newton update on this objective (similar to IRLS), one can derive the algorithm shown in Algorithm 9. This is known as **logitBoost** [FHT00]. The key subroutine is the ability of the weak learner F to solve a weighted least squares problem. This method can be generalized to the multi-class setting, as explained in [FHT00].

18.5.5 Gradient boosting

Rather than deriving new versions of boosting for every different loss function, it is possible to derive a generic version, known as **gradient boosting** [Fri01; Mas+00]. To explain this, imagine solving $\hat{\boldsymbol{f}} = \text{argmin}_{\boldsymbol{f}} \mathcal{L}(\boldsymbol{f})$ by performing gradient descent in the space of functions. Since functions are infinite dimensional objects, we will represent them by their values on the training set, $\boldsymbol{f} = (f(\boldsymbol{x}_1), \ldots, f(\boldsymbol{x}_N))$. At step m, let $\boldsymbol{g}_m$ be the gradient of $\mathcal{L}(\boldsymbol{f})$ evaluated at $\boldsymbol{f} = \boldsymbol{f}_{m-1}$:

$$g_{im} = \left[\frac{\partial \ell(y_i, f(\boldsymbol{x}_i))}{\partial f(\boldsymbol{x}_i)}\right]_{f=f_{m-1}} \tag{18.38}$$

Algorithm 9: LogitBoost, for binary classification with log-loss

$\omega_i = 1/N$, $\pi_i = 1/2$
for $m = 1 : M$ **do**
  Compute the working response $z_i = \frac{y_i^* - \pi_i}{\pi_i(1-\pi_i)}$
  Compute the weights $\omega_i = \pi_i(1 - \pi_i)$
  $F_m = \text{argmin}_F \sum_{i=1}^N \omega_i (z_i - F(\boldsymbol{x}_i))^2$
  Update $f(\boldsymbol{x}) \leftarrow f(\boldsymbol{x}) + \frac{1}{2} F_m(\boldsymbol{x})$
  Compute $\pi_i = 1/(1 + \exp(-2f(\boldsymbol{x}_i)))$;
Return $f(\boldsymbol{x}) = \text{sgn}\left[\sum_{m=1}^M F_m(\boldsymbol{x})\right]$

Name	Loss	$-\partial \ell(y_i, f(\boldsymbol{x}_i))/\partial f(\boldsymbol{x}_i)$
Squared error	$\frac{1}{2}(y_i - f(\boldsymbol{x}_i))^2$	$y_i - f(\boldsymbol{x}_i)$
Absolute error	$\lvert y_i - f(\boldsymbol{x}_i)\rvert$	$\text{sgn}(y_i - f(\boldsymbol{x}_i))$
Exponential loss	$\exp(-\tilde{y}_i f(\boldsymbol{x}_i))$	$-\tilde{y}_i \exp(-\tilde{y}_i f(\boldsymbol{x}_i))$
Binary Logloss	$\log(1 + e^{-\tilde{y}_i f_i})$	$y_i - \pi_i$
Multiclass logloss	$-\sum_c y_{ic} \log \pi_{ic}$	$y_{ic} - \pi_{ic}$

Table 18.1: Some commonly used loss functions, their gradients, and their population minimizers F^*. For binary classification problems, we assume $\tilde{y}_i \in \{-1, +1\}$, and $\pi_i = \sigma(2f(\boldsymbol{x}_i))$. For regression problems, we assume $y_i \in \mathbb{R}$. Adapted from [HTF09, p360] and [BH07, p483].

Gradients of some common loss functions are given in Table 18.1. We then make the update

$$\boldsymbol{f}_m = \boldsymbol{f}_{m-1} - \beta_m \boldsymbol{g}_m \tag{18.39}$$

where β_m is the step length, chosen by

$$\beta_m = \underset{\beta}{\text{argmin}}\, \mathcal{L}(\boldsymbol{f}_{m-1} - \beta \boldsymbol{g}_m) \tag{18.40}$$

In its current form, this is not much use, since it only optimizes f at a fixed set of N points, so we do not learn a function that can generalize. However, we can modify the algorithm by fitting a weak learner to approximate the negative gradient signal. That is, we use this update

$$F_m = \underset{F}{\text{argmin}} \sum_{i=1}^N (-g_{im} - F(\boldsymbol{x}_i))^2 \tag{18.41}$$

The overall algorithm is summarized in Algorithm 10. We have omitted the line search step for β_m, which is not strictly necessary, as argued in [BH07]. However, we have introduced a learning rate or **shrinkage factor** $0 < \nu \leq 1$, to control the size of the updates, for regularization purposes.

If we apply this algorithm using squared loss, we recover L2Boosting, since $-g_{im} = y_i - f_{m-1}(\boldsymbol{x}_i)$ is just the residual error. We can also apply this algorithm to other loss functions, such as absolute loss or Huber loss (Section 5.1.5.3), which is useful for robust regression problems.

Algorithm 10: Gradient boosting

Initialize $f_0(\boldsymbol{x}) = \text{argmin}_F \sum_{i=1}^N L(y_i, F(\boldsymbol{x}_i))$
for $m = 1 : M$ **do**
  Compute the gradient residual using $r_{im} = -\left[\frac{\partial L(y_i, f(\boldsymbol{x}_i))}{\partial f(\boldsymbol{x}_i)}\right]_{f(\boldsymbol{x}_i)=f_{m-1}(\boldsymbol{x}_i)}$
  Use the weak learner to compute $F_m = \text{argmin}_F \sum_{i=1}^N (r_{im} - F(\boldsymbol{x}_i))^2$
  Update $f_m(\boldsymbol{x}) = f_{m-1}(\boldsymbol{x}) + \nu F_m(\boldsymbol{x})$
Return $f(\boldsymbol{x}) = f_M(\boldsymbol{x})$

For classification, we can use log-loss. In this case, we get an algorithm known as **BinomialBoost** [BH07]. The advantage of this over LogitBoost is that it does not need to be able to do weighted fitting: it just applies any black-box regression model to the gradient vector. To apply this to multi-class classification, we can fit C separate regression trees, using the pseudo residual of the form

$$-g_{icm} = \frac{\partial \ell(y_i, f_{1m}(\boldsymbol{x}_i), \ldots, f_{Cm}(\boldsymbol{x}_i))}{\partial f_{cm}(\boldsymbol{x}_i)} = \mathbb{I}(y_i = c) - \pi_{ic} \tag{18.42}$$

Although the trees are fit separately, their predictions are combined via a softmax transform

$$p(y = c|\boldsymbol{x}) = \frac{e^{f_c(\boldsymbol{x})}}{\sum_{c'=1}^C e^{f_{c'}(\boldsymbol{x})}} \tag{18.43}$$

When we have large datasets, we can use a stochastic variant in which we subsample (without replacement) a random fraction of the data to pass to the regression tree at each iteration. This is called **stochastic gradient boosting** [Fri99]. Not only is it faster, but it can also generalize better, because subsampling the data is a form of regularization.

18.5.5.1 Gradient tree boosting

In practice, gradient boosting nearly always assumes that the weak learner is a regression tree, which is a model of the form

$$F_m(\boldsymbol{x}) = \sum_{j=1}^{J_m} w_{jm} \mathbb{I}(\boldsymbol{x} \in R_{jm}) \tag{18.44}$$

where w_{jm} is the predicted output for region R_{jm}. (In general, w_{jm} could be a vector.) This combination is called **gradient boosted regression trees**, or **gradient tree boosting**. (A related version is known as **MART**, which stands for "multivariate additive regression trees" [FM03].)

To use this in gradient boosting, we first find good regions R_{jm} for tree m using standard regression tree learning (see Section 18.1) on the residuals; we then (re)solve for the weights of each leaf by solving

$$\hat{w}_{jm} = \underset{w}{\text{argmin}} \sum_{\boldsymbol{x}_i \in R_{jm}} \ell(y_i, f_{m-1}(\boldsymbol{x}_i) + w) \tag{18.45}$$

For squared error (as used by gradient boosting), the optimal weight $\hat{w}_{jm}$ is the just the mean of the residuals in that leaf.

18.5.5.2 XGBoost

XGBoost (https://github.com/dmlc/xgboost), which stands for "extreme gradient boosting", is a very efficient and widely used implementation of gradient boosted trees, that adds a few more improvements beyond the description in Section 18.5.5.1. The details can be found in [CG16], but in brief, the extensions are as follows: it adds a regularizer on the tree complexity, it uses a second order approximation of the loss (from [FHT00]) instead of just a linear approximation, it samples features at internal nodes (as in random forests), and it uses various computer science methods (such as handling out-of-core computation for large datasets) to ensure scalability.[2]

In more detail, XGBoost optimizes the following regularized objective

$$\mathcal{L}(f) = \sum_{i=1}^{N} \ell(y_i, f(\boldsymbol{x}_i)) + \Omega(f) \tag{18.46}$$

where

$$\Omega(f) = \gamma J + \frac{1}{2}\lambda \sum_{j=1}^{J} w_j^2 \tag{18.47}$$

is the regularizer, where J is the number of leaves, and $\gamma \geq 0$ and $\lambda \geq 0$ are regularization coefficients. At the m'th step, the loss is given by

$$\mathcal{L}_m(F_m) = \sum_{i=1}^{N} \ell(y_i, f_{m-1}(\boldsymbol{x}_i) + F_m(\boldsymbol{x}_i)) + \Omega(F_m) + \text{const} \tag{18.48}$$

We can compute a second order Taylor expansion of this as follows:

$$\mathcal{L}_m(F_m) \approx \sum_{i=1}^{N} \left[\ell(y_i, f_{m-1}(\boldsymbol{x}_i)) + g_{im} F_m(\boldsymbol{x}_i) + \frac{1}{2} h_{im} F_m^2(\boldsymbol{x}_i) \right] + \Omega(F_m) + \text{const} \tag{18.49}$$

where h_{im} is the Hessian

$$h_{im} = \left[\frac{\partial^2 \ell(y_i, f(\boldsymbol{x}_i))}{\partial f(\boldsymbol{x}_i)^2} \right]_{f=f_{m-1}} \tag{18.50}$$

In the case of regression trees, we have $F(\boldsymbol{x}) = w_{q(\boldsymbol{x})}$, where $q : \mathbb{R}^D \to \{1, \ldots, J\}$ specifies which leaf node $\boldsymbol{x}$ belongs to, and $\boldsymbol{w} \in \mathbb{R}^J$ are the leaf weights. Hence we can rewrite Equation (18.49) as

2. Some other popular gradient boosted trees packages are **CatBoost** (https://catboost.ai/) and **LightGBM** (https://github.com/Microsoft/LightGBM).

follows, dropping terms that are independent of F_m:

$$\mathcal{L}_m(q, \boldsymbol{w}) \approx \sum_{i=1}^{N} \left[g_{im} F_m(\boldsymbol{x}_i) + \frac{1}{2} h_{im} F_m^2(\boldsymbol{x}_i) \right] + \gamma J + \frac{1}{2} \lambda \sum_{j=1}^{J} w_j^2 \tag{18.51}$$

$$= \sum_{j=1}^{J} \left[(\sum_{i \in I_j} g_{im}) w_j + \frac{1}{2} (\sum_{i \in I_j} h_i + \lambda) w_j^2 \right] + \gamma J \tag{18.52}$$

where $I_j = \{i : q(\boldsymbol{x}_i) = j\}$ is the set of indices of data points assigned to the j'th leaf.

Let us define $G_{jm} = \sum_{i \in I_j} g_{im}$ and $H_{jm} = \sum_{i \in I_j} h_{im}$. Then the above simplifies to

$$\mathcal{L}_m(q, \boldsymbol{w}) = \sum_{j=1}^{J} \left[G_{jm} w_j + \frac{1}{2} (H_{jm} + \lambda) w_j^2 \right] + \gamma J \tag{18.53}$$

This is a quadratic in each w_jm so the optimal weights are given by

$$w_j^* = -\frac{G_{jm}}{H_{jm} + \lambda} \tag{18.54}$$

The loss for evaluating different tree structures q then becomes

$$\mathcal{L}_m(q, \boldsymbol{w}^*) = -\frac{1}{2} \sum_{j=1}^{J} \frac{G_{jm}^2}{H_{jm} + \lambda} + \gamma J \tag{18.55}$$

We can greedily optimize this using a recursive node splitting procedure, as in Section 18.1. Specifically, for a given leaf j, we consider splitting it into a left and right half, $I = I_L \cup I_R$. We can compute the gain (reduction in loss) of such a split as follows:

$$\text{gain} = \frac{1}{2} \left[\frac{G_L^2}{H_L + \lambda} + \frac{G_R^2}{H_R + \lambda} - \frac{(G_L + G_R)^2}{(H_L + H_R) + \lambda} \right] - \gamma \tag{18.56}$$

where $G_L = \sum_{i \in I_L} g_{im}$, $G_R = \sum_{i \in I_R} g_{im}$, $H_L = \sum_{i \in I_L} h_{im}$, and $H_R = \sum_{i \in I_R} h_{im}$. Thus we see that it is not worth splitting a node if the gain is negative (i.e., the first term is less than γ).

A fast approximation for evaluating this objective, that does not require sorting the features (for choosing the optimal threshold to split on), is described in [CG16].

18.6 Interpreting tree ensembles

Trees are popular because they are interpretable. Unfortunately, ensembles of trees (whether in the form of bagging, random forests, or boosting) lose that property. Fortunately, there are some simple methods we can use to interpret what function has been learned.

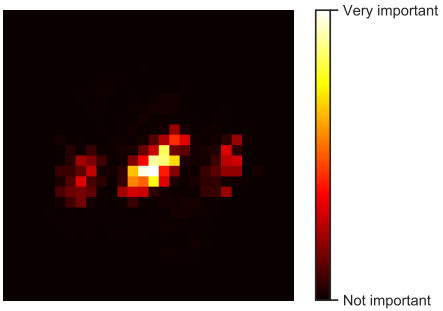


Figure 18.8: Feature importance of a random forest classifier trained to distinguish MNIST digits from classes 0 and 8. Adapted from Figure 7.6 of [Gér19]. Generated by rf_feature_importance_mnist.ipynb.

18.6.1 Feature importance

For a single decision tree T , [BFO84] proposed the following measure for **feature importance** of feature k:

$$R_k(T) = \sum_{j=1}^{J-1} G_j \mathbb{I}\left(v_j = k\right) \tag{18.57}$$

where the sum is over all non-leaf (internal) nodes, G_j is the gain in accuracy (reduction in cost) at node j, and $v_j = k$ if node j uses feature k. We can get a more reliable estimate by averaging over all trees in the ensemble:

$$R_k = \frac{1}{M} \sum_{m=1}^{M} R_k(T_m) \tag{18.58}$$

After computing these scores, we can normalize them so the largest value is 100%. We give some examples below.

Figure 18.8 gives an example of estimating feature importance for a classifier trained to distinguish MNIST digits from classes 0 and 8. We see that it focuses on the parts of the image that differ between these classes.

In Figure 18.9, we plot the relative importance of each of the features for the spam dataset (Section 18.4). Not surprisingly, we find that the most important features are the words "george" (the name of the recipient) and "hp" (the company he worked for), as well as the characters ! and $. (Note it can be the presence or absence of these features that is informative.)

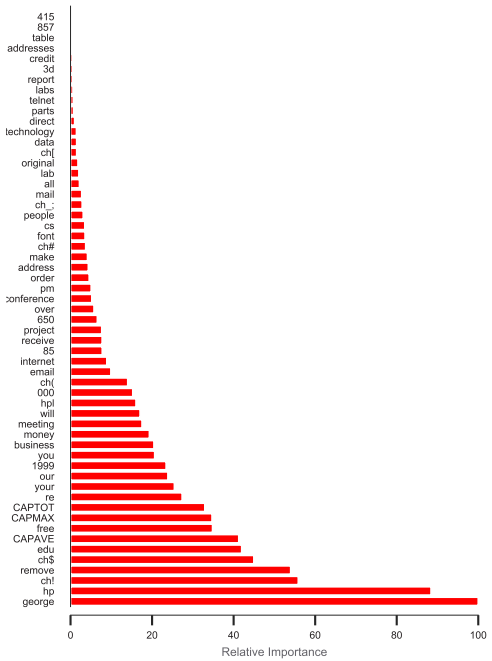
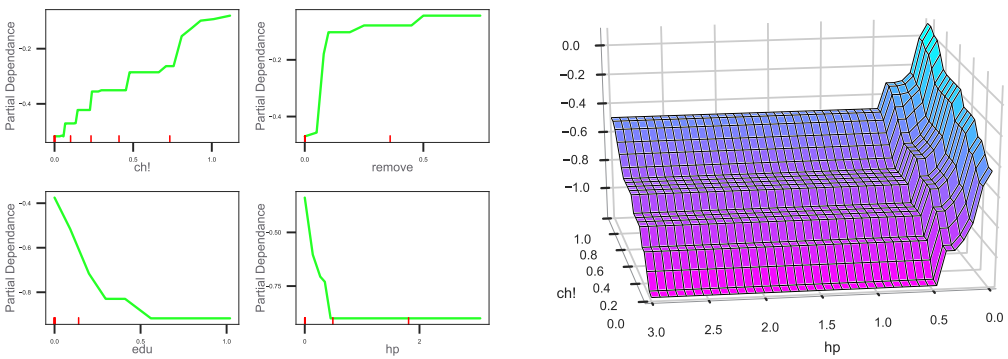


Figure 18.9: Feature importance of a gradient boosted classifier trained to distinguish spam from non-spam email. The dataset has X training examples with Y features, corresponding to token frequency. Adapted from Figure 10.6 of [HTF09]. Generated by spam_tree_ensemble_interpret.ipynb.



(a) *(b)*

Figure 18.10: (a) Partial dependence of log-odds of the spam class for 4 important predictors. The red ticks at the base of the plot are deciles of the empirical distribution for this feature. (b) Joint partial dependence of log-odds on the features hp and !. Adapted from Figure 10.6–10.8 of [HTF09]. Generated by spam_tree_ensemble_interpret.ipynb.

18.6.2 Partial dependency plots

After we have identified the most relevant input features, we can try to assess the impact they have on the output. A **partial dependency plot** for feature k is a plot of

$$\overline{f}_k(x_k) = \frac{1}{N}\sum_{n=1}^{N} f(\boldsymbol{x}_{n,-k}, x_k) \tag{18.59}$$

vs x_k. Thus we marginalize out all features except k. In the case of a binary classifier, we can convert this to log odds, $\log p(y=1|x_k)/p(y=0|x_k)$, before plotting. We illustrate this for our spam example in Figure 18.10a for 4 different features. We see that as the frequency of ! and "remove" increases, so does the probability of spam. Conversely, as the frequency of "edu" or "hp" increases, the probability of spam decreases.

We can also try to capture interaction effects between features j and k by computing

$$\overline{f}_{jk}(x_j, x_k) = \frac{1}{N}\sum_{n=1}^{N} f(\boldsymbol{x}_{n,-jk}, x_j, x_k) \tag{18.60}$$

We illustrate this for our spam example in Figure 18.10b for hp and !. We see that higher frequency of ! makes it more likely to be spam, but much more so if the word "hp" is missing.