Lecture 9: Tree-Based Methods and Ensembling Methods

Readings: ESL (Ch. 9.2-10, 15), ISL (Ch. 8), Bach (Ch. 10); code

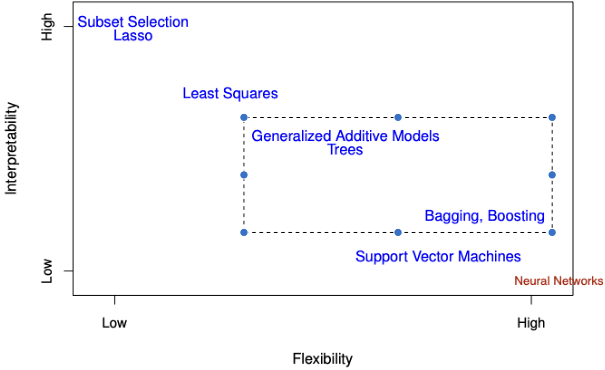
Soon Hoe Lim

September 23, 2025

Outline

Where Are We So Far

So far, we've focused on models with strong global structures, like linear and additive models. We now turn to a more flexible, non-parametric approach that works by partitioning the data into local regions.



Decision Trees for Regression and Classification

A simple class of methods are **tree-based**: the input space is partitioned into simple regions and a constant prediction is assigned to each region.

Definition 1: Tree-Based Model

Partition feature space into M disjoint regions $R_1, \ldots, R_M$:

$$f(\mathbf{x}) = \sum_{m=1}^{M} c_m \mathbb{I}(\mathbf{x} \in R_m)$$

where c_m is the prediction in region R_m.

Decision trees can be applied to both regression and classification problems.

- **Regression:** $c_m = \text{mean}(y_i : \mathbf{x}_i \in R_m)$
- **Classification:** $c_m = \text{mode}(y_i : \mathbf{x}_i \in R_m)$

Tree-based methods are simple and useful for interpretation. See blackboard for an example (also see the example based on baseball salary data in ISL).

Details of the Tree-Building Process

- We divide the predictor space — that is, the set of possible values for $X_1, X_2, \dots, X_p$ — into J distinct and non-overlapping regions, $R_1, R_2, \dots, R_J$.
- For every observation that falls into the region R_j, we make the same prediction, which is simply the mean of the response values for the training observations in R_j.
- In theory, the regions could have any shape. However, we choose to divide the predictor space into high-dimensional rectangles (boxes), for simplicity and for ease of interpretation of the resulting predictive model.
- The goal is to find boxes $R_1, \dots, R_J$ that minimize the RSS, given by

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

where $\hat{c}_j$ is the mean response (for regression) for the training observations within the jth box.

Top Down, Greedy Approach to Building Trees

- Unfortunately, it is computationally infeasible to consider every possible partition of the feature space into J boxes.
- For this reason, we take a top-down, greedy approach that is known as **recursive binary splitting**.
- The approach is top-down because it begins at the top of the tree and then successively splits the predictor space; each split is indicated via two new branches further down on the tree.
- It is greedy because at each step of the tree-building process, the best split is made at that particular step, rather than looking ahead and picking a split that will lead to a better tree in some future step.

💡 Since the set of splitting rules used to segment the predictor space can be summarized in a tree, these types of approaches are known as decision-tree methods. Trees create rectangular partitions through recursive binary splitting, making them naturally interpretable as decision rules.

Building Trees: The Recursive Binary Splitting Process

Definition 2: Binary Split

At each internal node, split data into two regions:

$$R_1(j, s) = \{X | X_j \leq s\} \quad (1)$$
$$R_2(j, s) = \{X | X_j > s\} \quad (2)$$

where j is the splitting variable and s is the split point.

Goal: Find the "best" binary split at each step: select the predictor X_j and the cutpoint s such that splitting the predictor space leads to the greatest reduction in the residual sum of squares (RSS).

Optimization Problem: Find (j^*, s^*) that minimizes:

- **Regression:** $\sum_{i: x_i \in R_1(j,s)} (y_i - \hat{c}_1)^2 + \sum_{i: x_i \in R_2(j,s)} (y_i - \hat{c}_2)^2$
- **Classification:** Weighted impurity $N_1 Q(R_1) + N_2 Q(R_2)$

where $\hat{c}_1, \hat{c}_2$ are region means/modes, N_i is number of data samples that fall into the ith region, and $Q(R)$ is impurity measure for region R (see later slides).

Partitions and CART

Regression and classification models using decision trees are called CART (classification and regression trees).

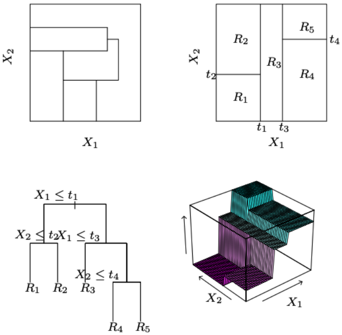


FIGURE 9.2. *Partitions and CART. Top right panel shows a partition of a two-dimensional feature space by recursive binary splitting, as used in CART, applied to some fake data. Top left panel shows a general partition that cannot be obtained from recursive binary splitting. Bottom left panel shows the tree corresponding to the partition in the top right panel, and a perspective plot of the prediction surface appears in the bottom right panel.*

Growing Regression Trees Using a Greedy Approach

CART Algorithm

```
Input: Training data {(x_i, y_i)}_{i=1}^N, stopping criteria
Initialize: Root node containing all data
for each internal node t do
    if stopping criterion not met then
        for each variable X_j and split point s do
            Define: t_L = {x ∈ t : x_j ≤ s}, t_R = {x ∈ t : x_j > s}
            Compute improvement:

                Δ(s, j) = RSS(t) − RSS(t_L) − RSS(t_R)

        end for
        Choose: (s*, j*) = arg max_{s,j} Δ(s, j)
        Split node if Δ(s*, j*) > threshold
    end if
end for
```

Here, $\text{RSS}(t) = \sum_{\mathbf{x}_i \in t} (y_i - \bar{y}_t)^2$, and the stopping criteria could be minimum observations per region, minimum improvement, maximum depth.

The Overfitting Issue

- The process described above may produce good predictions on the training set, but is likely to **overfit** the data, leading to poor test performance (why?).
- A smaller tree with fewer splits (that is, fewer regions $R_1, \ldots, R_J$) might lead to lower variance and better interpretation, at the cost of a little bias.
- One possible alternative to the process described above is to grow the tree only so long as the decrease in the RSS due to each split exceeds some (high) threshold.
- This strategy will result in smaller trees, but is too short-sighted: a seemingly worthless split early on in the tree might be followed by a very good split — i.e., a split that leads to a large reduction in RSS later on.

Pruning a Tree: Regression Case

- A better strategy is to grow a very large tree T_0, and then prune it back in order to obtain a subtree.
- **Cost complexity pruning** — also known as **weakest link pruning** — is used to do this.
- We consider a sequence of trees indexed by a nonnegative tuning parameter α. For each value of α there corresponds a subtree $T \subset T_0$ such that

$$\sum_{m=1}^{|T|} \sum_{i: x_i \in R_m} (y_i - \hat{y}_{R_m})^2 + \alpha |T|$$

  is as small as possible.
- Here $|T|$ indicates the number of terminal nodes of the tree T, R_m is the rectangle (i.e., the subset of predictor space) corresponding to the mth terminal node, and $\hat{y}_{R_m}$ is the mean of the training observations in R_m.
- The tuning parameter α controls a trade-off between the subtree's complexity and its fit to the training data.
- We select an optimal value $\hat{\alpha}$ using cross-validation, then return to the full data set and obtain the subtree corresponding to $\hat{\alpha}$.

Regression Trees: Summary of Algorithm

1. Use recursive binary splitting to grow a large tree on the training data, stopping only when each terminal node has fewer than some minimum number of observations.
2. Apply cost complexity pruning to the large tree in order to obtain a sequence of best subtrees, as a function of α.
3. Use K-fold cross-validation to choose α. For each $k = 1, \ldots, K$:
   3.1 Repeat Steps 1 and 2 on the $\frac{K-1}{K}$ fraction of the training data, excluding the kth fold.
   3.2 Evaluate the mean squared prediction error on the data in the left-out kth fold, as a function of α.

   Average the results, and pick α to minimize the average error.
4. Return the subtree from Step 2 that corresponds to the chosen value of α.

Revisit the baseball dataset example in ISL.

Classification Trees

- Very similar to a regression tree, except that it is used to predict a **qualitative response** rather than a quantitative one.
- For a classification tree, we predict that each observation belongs to the most commonly occurring class of training observations in the region to which it belongs.
- Just as in the regression setting, we use recursive binary splitting to grow a classification tree.
- In the classification setting, RSS cannot be used as a criterion for making the binary splits.
- A natural alternative to RSS is the **classification error rate**. This is simply the fraction of the training observations in that region that do not belong to the most common class:

$$E = 1 - \max_k(\hat{p}_{mk}),$$

  where $\hat{p}_{mk}$ represents the proportion of training observations in the mth region that are from the kth class.
- However, classification error is not sufficiently sensitive for tree-growing, and in practice two other measures are preferable.

Node Impurity Measures

Node impurity tells us how "mixed up" the node is in terms of class labels:

1. **Gini Index:** $G = \sum_{k=1}^{K} \hat{p}_{mk}(1 - \hat{p}_{mk}) = 1 - \sum_{k=1}^{K} \hat{p}_{mk}^2$
2. **Cross-Entropy (Deviance):** $CE = -\sum_{k=1}^{K} \hat{p}_{mk} \log \hat{p}_{mk}$

Properties:

- E, G, CE are zero when node is pure (all same class), and are maximum when classes are equally represented: $\hat{p}_{mk} = 1/K$.
- G is a measure of total variance across the K classes, and takes on a small value if all of the $\hat{p}_{mk}$'s are close to zero or one.
- For this reason, the Gini index is referred to as a measure of **node purity** — a small value indicates that a node contains predominantly observations from a single class.
- G and CE are more sensitive to probability changes than misclassification. They are differentiable (better for optimization) and lead to better intermediate splits during construction.
- For binary case: $G = 2p(1 - p)$, $CE = -p \log p - (1 - p) \log(1 - p)$.

Recommendation: Use Gini or entropy for growing; any measure for pruning.

Tree Pruning: Cost-Complexity Pruning

⚠ Large trees overfit; simple stopping rules are suboptimal.

💡 To overcome this, we usually devise a pruning procedure that involves minimizing a loss function plus a term that penalizes the complexity of a tree.

For subtree $T \subseteq T_0$ (full tree), minimize: $C_\alpha(T) = \sum_{m=1}^{|T|} N_m Q_m(T) + \alpha|T|$, where $|T|$ = number of terminal nodes (leaves), N_m = number of observations in node m, $Q_m(T)$ = node impurity, α = complexity parameter.

💡 For any α, there exists a unique smallest subtree T_α minimizing $C_\alpha(T)$. This means instead of searching over all subtrees, we only need to work with a sequence of nested candidates.

Minimal Cost-Complexity Pruning

1: Grow large tree T_0 using stopping rule (e.g., min 5 obs per node)
2: Compute sequence $T_0 \supset T_1 \supset \ldots \supset \{t_1\}$ by iteratively pruning weakest link
3: For each subtree, estimate generalization error via cross-validation
4: Choose subtree with minimum CV error (or within 1 SE rule)

Understanding Tree Instability

High Variance Problem: Small changes in training data can lead to very different tree structures and predictions.

Sources of instability:

- **Hierarchical Effect:** Errors in early splits propagate throughout tree
- **Greedy Selection:** Locally optimal splits may be globally suboptimal
- **Discrete Splits:** Small data changes can move split points dramatically
- **Hard Boundaries:** Abrupt transitions between regions (see ESL Ch. 9.5 for an alternative method based on probabilistic splits)

Bias-Variance Tradeoff for Trees

- **Low Bias:** Can approximate complex decision boundaries
- **High Variance:** Very sensitive to training data

💡 Poor generalization despite good training fit! If $\mathbf{x}$ is near split point s, small training changes can move $\mathbf{x}$ to different sides, causing completely different predictions. MARS is a more stable alternative (see ESL Ch. 9.4).

Overview: From Single Trees to Powerful Ensembles

Decision Trees: The Building Blocks

- Decision trees work by sequentially splitting variables to create rectangular regions in the feature space.
- Predictions are then made locally within each of these regions.
- They are intuitive, easy to implement, and can be highly interpretable.
- However, they generally have lower prediction accuracy on their own and are considered **weak learners**.

Ensembling Methods: The Solution

- This leads to the famous "Boosting Problem" posed by Kearns & Valiant: *Can a set of weak learners*[1] *be combined to create a single strong learner?*
- **Bagging**, **random forests** and **boosting** are methods that achieve this by making predictions based on an ensemble of many trees.
- Bagging and boosting are **general methodologies** and are not limited to just trees, even though that is our focus.

[1] Weak learners perform just slightly better than random chance. In practice, people still prefer to use strong base learners.

Ensembling Methods

Instead of finding one "perfect" model, we combine many "good" models to achieve better performance than any individual model. In fact, we often need both accurate and diverse learners to increase the generalization ability of the ensemble (see Exercise 9.1).

Why ensembles work:

- **Variance Reduction:** Averaging reduces variance without increasing bias.
- **Bias Reduction:** Sequential methods can systematically reduce bias.
- **Improved Stability:** Less sensitive to outliers and data peculiarities.
- **Better Generalization:** Combine different "views" of the data.

Two main approaches:

1. **Parallel:** Create individual learners independently and parallelize the generation process, then combine (bagging, random forest).
2. **Sequential:** Create individual learners with strong correlations and generate the learners sequentially, each learning from the previous one's mistakes (boosting).

Bagging: Bootstrap Aggregating

Bagging is a general-purpose procedure for reducing the variance of a method.

Definition 3: The Bagging Algorithm

1. For $b = 1, \ldots, B$:
   1.1 Draw a bootstrap sample $\mathcal{Z}^{*b}$ of size N from the original data (i.e., sampling with replacement).
   1.2 Train a model $\hat{f}_b$ on the bootstrap sample $\mathcal{Z}^{*b}$.
2. **Final prediction:** Aggregate the results.
   - Regression: $\hat{f}_{\text{bag}}(\mathbf{x}) = \frac{1}{B}\sum_{b=1}^{B}\hat{f}_b(\mathbf{x})$
   - Classification: Take the majority vote

Key Idea: Bagging is most effective for high-variance, low-bias models like fully grown decision trees.

Out-of-Bag (OOB) Error Estimation

- There is a straightforward way to estimate the test error of a bagged model.
- Recall: the key to bagging is that trees are repeatedly fit to bootstrapped subsets of the observations. On average, each bagged tree makes use of about two-thirds of the observations.
- Consequently, each bootstrap sample omits about 36.8% of the data. These **out-of-bag (OOB)** samples can be used as a built-in validation set to estimate test error, removing the need for separate cross-validation.
- We can predict the response for the ith observation using each of the trees in which that observation was OOB. This yields around $B/3$ predictions for the ith observation, which we then average.
- This estimate is essentially the leave-one-out (LOO) cross-validation error for bagging, provided B is large.

Why Bagging Works: Variance Reduction

Assume each trained model $\hat{f}_b$ is the approximation of the true function f^*:

$$\hat{f}_b(\mathbf{x}) = f^*(\mathbf{x}) + \epsilon_b(\mathbf{x})$$

where ϵ_b is a random error function with mean 0 and variance $\sigma^2(\mathbf{x})$. The aggregated prediction is $\hat{f}(\mathbf{x}) = \frac{1}{B}\sum_{b=1}^{B} \hat{f}_b(\mathbf{x})$.

💡 Averaging reduces variance! If the errors ϵ_b were uncorrelated, the variance of the final prediction would be reduced by a factor of B.

- **Mean Squared Error of a single model $\hat{f}_b$:**

$$\mathbb{E}[(f^*(\mathbf{x}) - \hat{f}_b(\mathbf{x}))^2] = \mathbb{E}[\epsilon_b(\mathbf{x})^2] = \sigma^2(\mathbf{x}).$$

- **Mean Squared Error of the aggregated model $\hat{f}$ (uncorrelated case):**

$$\mathbb{E}[(f^*(\mathbf{x}) - \hat{f}(\mathbf{x}))^2] = \mathbb{E}\left[\left(\frac{1}{B}\sum_{b=1}^{B} \epsilon_b(\mathbf{x})\right)^2\right] = \frac{1}{B^2}\sum_{b=1}^{B} \mathbb{E}[\epsilon_b(\mathbf{x})^2] = \frac{\sigma^2(\mathbf{x})}{B}.$$

Why Bagging Isn't Enough: The Correlation Problem

⚠ In practice, bootstrap samples are correlated, so the variance reduction is not a full $1/B$. Also, bagging does not reduce a model's bias.

The Math of Averaging Correlated Variables

If we average B identically distributed variables, each with variance σ^2 and positive pairwise correlation ρ, the variance of the average is (see Exercise 9.2):

$$\text{Variance} = \rho\sigma^2 + \frac{1-\rho}{B}\sigma^2.$$

As the number of trees B grows, the second term vanishes, but the first term, $\rho\sigma^2$, remains. Since bootstrap samples are drawn from the same dataset, the resulting trees are correlated ($\rho > 0$). This correlation limits the variance reduction we can achieve with bagging.

Goal: To improve on bagging, we need to reduce the correlation ρ between the trees. This is the core idea behind **Random Forests**, which give a small tweak that decorrelates the trees to reduce the variance when we average the trees.

Random Forest: Improving on Bagging

The Problem with Bagged Trees: If the dataset has a very strong predictor, most bagged trees will use it as the top split. This makes the trees look similar and become correlated, which limits the variance reduction from averaging.

Definition 4: The Random Forest Algorithm

The Random Forest algorithm is identical to Bagging, with one crucial modification during the training of each tree $\hat{f}_b$:

- **At each candidate split in the tree, randomly select a small subset of $m \ll p$ predictors from the full set of p predictors. Only these m predictors are considered for finding the best split.**

How it helps: This simple tweak decorrelates the trees. It prevents strong predictors from dominating every tree, forcing the ensemble to explore a more diverse set of predictive rules.

Key parameters:

- B: Number of trees (larger is usually better, until performance plateaus).
- m: Number of predictors per split (typically, $m \approx \sqrt{p}$ for classification, $m \approx p/3$ for regression).

Random Forest: Variable Importance

Two common methods for measuring a predictor's/feature's importance:

1. Mean Decrease in Impurity (Gini Importance)

- For each predictor/feature j, sum the impurity decrease over all splits in a tree where j was used.
- Average this sum across all trees in the forest.
- **Pros:** Fast to compute.
- **Cons:** Can be biased towards features with high cardinality.

2. Permutation Importance (Preferred Method)

- First, record the baseline OOB error of the trained forest.
- For each feature j: randomly permute (shuffle) the values of feature j in the OOB samples and recompute the OOB error.
- The importance of feature j is the increase in error caused by permuting it.
- **Pros:** More reliable, model-agnostic, and unbiased.

Boosting

- Bagging involves creating multiple copies of the original training data set using the bootstrap, fitting a separate decision tree to each copy, and then combining all of the trees in order to create a single predictive model.
- Each tree is built on a bootstrap data set, independent of the other trees.
- Boosting works similarly, except that the trees are grown **sequentially**: each tree is grown using information from previously grown trees.

Like bagging, boosting is a general approach that can be applied to many statistical learning methods for regression or classification.

The Boosting Strategy In General:

1. Start with equal weights on all training observations.
2. Fit a weak learner to the weighted data.
3. **Increase the weights** on observations that were misclassified.
4. Repeat: each new learner is forced to focus on the "hard" cases.
5. Combine all learners via a weighted vote, giving more accurate learners a greater say.

Boosting Algorithm for Regression Trees

We first discuss boosting for decision trees[2].

1. Set $\hat{f}(x) = 0$ and $r_i = y_i$ for all i in the training set.
2. For $b = 1, 2, \ldots, B$, repeat:
   2.1 Fit a tree $\hat{f}_b$ with d splits ($d + 1$ terminal nodes) to the training data (X, r).
   2.2 Update $\hat{f}$ by adding in a shrunken version of the new tree:
   $$\hat{f}(x) \leftarrow \hat{f}(x) + \lambda \hat{f}_b(x).$$
   2.3 Update the residuals:
   $$r_i \leftarrow r_i - \lambda \hat{f}_b(x_i).$$
3. Output the boosted model:
$$\hat{f}(x) = \sum_{b=1}^{B} \lambda \hat{f}_b(x).$$

[2] For a Bayesian approach to fitting an ensemble of trees, see Ch. 8.2.4 in ESL.

What is the Idea Behind This Procedure?

- Unlike fitting a single large decision tree to the data, which amounts to fitting the data hard and potentially overfitting, the boosting approach instead **learns slowly**.
- Given the current model, we fit a decision tree to the residuals from the model. We then add this new decision tree into the fitted function in order to update the residuals.
- Each of these trees can be rather small, with just a few terminal nodes, determined by the parameter d in the algorithm.
- By fitting small trees to the residuals, we slowly improve $\hat{f}$ in areas where it does not perform well.
- The shrinkage parameter λ slows the process down even further, allowing more and different shaped trees to attack the residuals.

Boosting: Tuning Parameters

- **Number of trees** B: Unlike bagging and random forests, boosting can overfit if B is too large, although this overfitting tends to occur slowly if at all. We use cross-validation to select B.
- **Shrinkage parameter** λ: A small positive number controlling the learning rate. Typical values are 0.01 or 0.001, and the right choice can depend on the problem. Very small λ may require using a very large B to achieve good performance.
- **Number of splits** d **in each tree**: Controls the complexity of the boosted ensemble. Often $d = 1$ works well, in which case each tree is a stump (a single split). In this case, the boosted stump ensemble fits an additive model, since each term involves only a single variable (See Exercise 9.3). More generally, d is the *interaction depth*, controlling the maximum interaction order of the boosted model (up to d variables).

Boosting for Classification: AdaBoost

Boosting for classification is similar in spirit to boosting for regression, but is a bit more complex. Consider binary classification with $Y \in \{-1, +1\}$ and weak learners $G_m(\mathbf{x}) \in \{-1, +1\}$.

AdaBoost (Freund-Schapire 1996)

1: **Initialize:** observation weights $w_i^{(1)} = 1/N$ for $i = 1, \ldots, N$.
2: **for** $m = 1$ to M **do**
3: **Fit a weak classifier** $G_m(\mathbf{x})$ to the training data using weights $w_i^{(m)}$.
4: **Compute the weighted error:** $\text{err}_m = \frac{\sum_{i=1}^N w_i^{(m)} \mathbb{I}(y_i \neq G_m(\mathbf{x}_i))}{\sum_{i=1}^N w_i^{(m)}}$.
5: **Compute the classifier's weight:** $\alpha_m = \log\left(\frac{1-\text{err}_m}{\text{err}_m}\right)$.
6: **Update observation weights:**

$$w_i^{(m+1)} = w_i^{(m)} \exp[\alpha_m \mathbb{I}(y_i \neq G_m(\mathbf{x}_i))].$$

7: **Normalize** weights $w_i^{(m+1)}$ to sum to 1.
8: **end for**
9: **Output Final Model:** $G(\mathbf{x}) = \text{sign}\left[\sum_{m=1}^M \alpha_m G_m(\mathbf{x})\right]$.

AdaBoost: A Visual Intuition

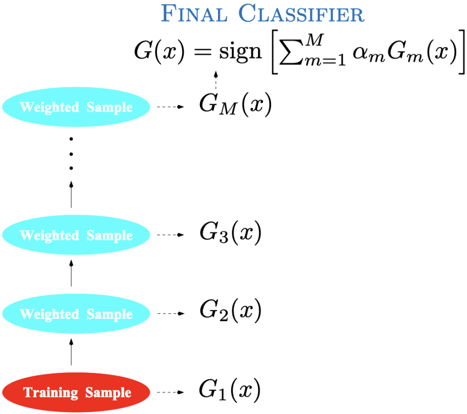


FIGURE 10.1. *Schematic of AdaBoost. Classifiers are trained on weighted versions of the dataset, and then combined to produce a final prediction.*

AdaBoost: The Statistical View

AdaBoost is not just a clever algorithm; it is a procedure for minimizing the **exponential loss function** in a stagewise fashion.

Why minimizing the exponential loss? See Exercise 9.4.

Exponential Loss

For a binary outcome $y \in \{-1, 1\}$, the exponential loss for a prediction score $f(\mathbf{x})$ is

$$L(y, f(\mathbf{x})) = \exp(-yf(\mathbf{x})).$$

The goal is to find an additive model

$$f(\mathbf{x}) = \sum_{m=1}^{M} \alpha_m G_m(\mathbf{x})$$

that minimizes

$$\sum_{i=1}^{N} \exp(-y_i f(\mathbf{x}_i)).$$

Stagewise Additive Fitting in AdaBoost

At step m, we have $f_{m-1}(\mathbf{x})$ and want to add a new term $\alpha_m G_m(\mathbf{x})$:

$$(\alpha_m, G_m) = \arg\min_{\alpha, G} \sum_{i=1}^{N} \exp\big(-y_i(f_{m-1}(\mathbf{x}_i) + \alpha G(\mathbf{x}_i))\big).$$

This can be rewritten as minimizing

$$\sum_{i=1}^{N} w_i^{(m)} \exp(-y_i \alpha G(\mathbf{x}_i)), \quad \text{where} \quad w_i^{(m)} = \exp\big(-y_i f_{m-1}(\mathbf{x}_i)\big).$$

- The AdaBoost steps for finding the best G_m (via minimizing weighted error) and α_m are exactly the solution to this minimization problem.
- **Limitation:** The exponential loss is very sensitive to outliers and mislabeled data points.

❓ We have focused on the binary case so far. But how about AdaBoost for multi-class case? See Exercise 9.5.

From AdaBoost to Gradient Boosting

💡 Boosting can be viewed as **gradient descent in function space**. We iteratively add a new function (weak learner) that points in the direction of the negative gradient of the loss function.

Definition 5: Forward Stagewise Additive Modeling

We build models of the form $f(\mathbf{x}) = \sum_{m=1}^{M} \beta_m b(\mathbf{x}; \gamma_m)$. Starting with $f_0(\mathbf{x})$, at each step m we solve:

$$(\beta_m, \gamma_m) = \arg\min_{\beta,\gamma} \sum_{i=1}^{N} L(y_i, f_{m-1}(\mathbf{x}_i) + \beta b(\mathbf{x}_i; \gamma))$$

Gradient Boosting

The Gradient Boosting Idea (Friedman, 2001):
Instead of solving the full optimization above, we approximate the solution by fitting our new weak learner to the **negative gradient** of the loss function, evaluated at the current model f_{m-1}. These negative gradients are called **pseudo-residuals**.

$$r_{im} = -\left[\frac{\partial L(y_i, f(\mathbf{x}_i))}{\partial f(\mathbf{x}_i)}\right]_{f=f_{m-1}}.$$

💡 This generalizes boosting to **any differentiable loss function**.

Common Loss Functions and The Generic Algorithm

- **Squared Loss:** $L(y, f) = \frac{1}{2}(y - f)^2 \implies r_i = y_i - f(\mathbf{x}_i)$ (Ordinary residuals!)
- **Absolute Loss:** $L(y, f) = |y - f| \implies r_i = \text{sign}(y_i - f(\mathbf{x}_i))$
- **Huber Loss:** Robust combination of squared and absolute loss.
- **Logistic Loss:** $L(y, f) = \log(1 + \exp(-yf))$ for classification.

Generic Gradient Boosting

1: **Initialize:** $f_0(\mathbf{x}) = \arg\min_\gamma \sum_{i=1}^N L(y_i, \gamma)$
2: **for** $m = 1$ to M **do**
3: **Compute pseudo-residuals:** $r_{im} = -\left[\frac{\partial L(y_i, f(\mathbf{x}_i))}{\partial f(\mathbf{x}_i)}\right]_{f=f_{m-1}}$
4: **Fit weak learner:** Fit a base learner $h_m(\mathbf{x})$ to the pseudo-residuals.
5: **Find optimal step size:** $\rho_m = \arg\min_\rho \sum_{i=1}^N L(y_i, f_{m-1}(\mathbf{x}_i) + \rho h_m(\mathbf{x}_i))$
6: **Update model:** $f_m(\mathbf{x}) = f_{m-1}(\mathbf{x}) + \rho_m h_m(\mathbf{x})$
7: **end for**

Why Trees as Weak Learners?

Decision trees are ideal weak learners for boosting because they:

- Handle mixed data types naturally (numerical + categorical).
- Capture complex non-linear interactions automatically.
- Require no data preprocessing (e.g., feature scaling).
- Are computationally efficient to train.
- Have a tunable complexity (e.g., depth) that provides a good bias-variance trade-off when kept shallow.

Gradient Tree Boosting: The Algorithm

Gradient Tree Boosting

Initialize: $f_0(\mathbf{x}) = \arg\min_\gamma \sum_{i=1}^N L(y_i, \gamma)$
for $m = 1$ to M **do**
  Compute pseudo-residuals: $r_{im} = -\left[\frac{\partial L(y_i, f(\mathbf{x}_i))}{\partial f(\mathbf{x}_i)}\right]_{f=f_{m-1}}$
  Fit a regression tree T_m to the pseudo-residuals $\{(\mathbf{x}_i, r_{im})\}_{i=1}^N$.
  The tree produces terminal regions (leaves) $R_{jm}, j = 1, \ldots, J_m$.
  for $j = 1$ to J_m **do**
    Optimize leaf values: Find the optimal constant update γ_{jm} for each leaf by solving a simple 1D optimization problem. $\gamma_{jm} = \arg\min_\gamma \sum_{\mathbf{x}_i \in R_{jm}} L(y_i, f_{m-1}(\mathbf{x}_i) + \gamma)$
  end for
  Update model: $f_m(\mathbf{x}) = f_{m-1}(\mathbf{x}) + \nu \sum_{j=1}^{J_m} \gamma_{jm} \mathbb{I}(\mathbf{x} \in R_{jm})$
end for

The learning rate ν (shrinkage) is a crucial regularization parameter.

Tuning and Regularization

1. Key Hyperparameters

- **Number of Trees (M):** Controls model complexity. Best set via **early stopping** on a validation set.
- **Learning Rate / Shrinkage (ν):** Scales the contribution of each tree. Smaller ν (e.g., 0.01-0.1) requires larger M but often leads to better generalization.
- **Tree Depth (J):** Controls the maximum level of feature interactions. Depths of 4-8 are common.

2. Advanced Regularization Techniques

- **Stochastic Gradient Boosting (Subsampling):** At each iteration, train the tree on a random subsample (e.g., 50-80%) of the data. This reduces overfitting and speeds up computation.
- **Feature Subsampling:** At each tree or each split, consider only a random subset of features. This further reduces variance and is a key feature in libraries like XGBoost.

Ensemble Method Comparison

Aspect	**Boosting**	**Bagging**	**Random Forest**
Construction	Sequential	Parallel	Parallel
Primary Goal	Bias reduction	Variance reduction	Variance reduction & decorrelation
Base Learners	Weak (simple)	Strong (complex)	Strong (complex)
Overfitting Risk	Moderate to High	Low	Very low
Noise Sensitivity	High	Low	Low
Parallelization	Limited	Full	Full
Parameter Tuning	More Complex	Simple	Simple
Performance	Excellent (tabular data)	Good	Very good

In practice, random forest is a robust and easy-to-use baseline. For maximum performance on structured/tabular data, a well-tuned gradient boosting model often shines.

Key Takeaways

Combining many weak learners can be more powerful than creating a single, highly complex strong learner.

- **Bagging/Random Forest:** A parallel combination of models that reduces variance by averaging.
- **Boosting:** A sequential combination of models that reduces bias by focusing on errors.

Practical Guidance:

- Use **Random Forest** for robustness, ease of use, and a strong baseline.
- Use **Gradient Boosting** for maximum predictive accuracy, especially on tabular data, but be prepared to tune hyperparameters carefully.
- Always use a validation set and early stopping when training boosting models.
- **Gradient Boosted Trees are among the strongest supervised learners.** Modern implementations like XGBoost and LightGBM make them fast, regularized, and scalable.

Exercises

Exercise 9

1. **The Accuracy-Diversity Tradeoff in Ensemble Learning.**

   Consider an ensemble of n binary classifiers for a classification problem. Let $E_i \in \{0, 1\}$ be the error indicator for the ith classifier, where $E_i = 1$ if classifier i makes an error ($E_i = 0$ otherwise). Each classifier has individual error rate $\varepsilon := P(E_i = 1) < 1/2$. The ensemble uses majority voting and predict the class chosen by more than half the classifiers. Denote $S_n = \sum_{i=1}^n E_i$ and $\bar{E}_n = S_n/n$.

   (a) Independent Classifiers: Assume E_i independent, so $S_n \sim \text{Binomial}(n, \varepsilon)$.

   (i) Show $\mathbb{P}(\bar{E}_n \geq 1/2) = \sum_{k=\lceil n/2 \rceil}^n \binom{n}{k} \varepsilon^k (1-\varepsilon)^{n-k}$.

   (ii) Use Hoeffding's inequality to show: $\mathbb{P}(\bar{E}_n \geq 1/2) \leq \exp(-2n(1/2 - \varepsilon)^2)$.

   (b) Correlated Classifiers: Assume $\text{Corr}(E_i, E_j) = \rho$ for $i \neq j$.

   (i) Derive: $\text{Var}(\bar{E}_n) = \frac{\varepsilon(1-\varepsilon)}{n}[1 + (n-1)\rho]$.

   (ii) Use Chebyshev's inequality to bound the majority vote error probability. Analyze the limiting behavior as $n \to \infty$ when $\rho > 0$.

   (c) Use the above results to explain why effective ensembles require both *individual accuracy* ($\varepsilon < 1/2$) and *diversity* (low ρ).

Exercises

Exercise 9

2. **Understanding Bagging: Error Estimation and Estimator Correlation.**
   (a) Solve Exercise 15.1 in ESL.
   (b) Solve Exercise 15.4 in ESL.
3. **Boosting Using Depth-One Trees.** Solve Exercise 8.4.2 in ISL.
4. **Population Minimizer for the Exponential Loss.** For binary classification with $y \in \{-1, 1\}$, show that the population minimizer for the exponential loss $L(y, f) = \exp(-yf)$ is $f(x) = \frac{1}{2} \log \frac{P(Y=1|x)}{P(Y=-1|x)}$ and deduce the Bayes classifier.
5. **Multiclass Exponential Loss.** Solve Exercise 10.5 in ESL.
6. **[Experimental]**
   (a) **Random Forest Classifiers.** Solve Exercise 15.6 in ESL.
   (b) **AdaBoost with Trees.** Solve Exercise 10.4 in ESL.