Data Mining and Machine Learning

Lecture 4: Decision Trees and Ensemble Learning

Dr Sarat Moka *UNSW, Sydney*

Key Topics

- Decision Trees
- Introduction to Ensemble Methods
- Out-of-Bag Approach
- Random Forests
- Boosting Methods

Books:

(A) Data Science and Machine Learning: Mathematical and Statistical Methods, by Kroese, Botev, Taimre, and Vaisman. Click here to download a pdf copy.

(B) Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow (2nd Ed.) by Aurélien Géron (2019).

4.1 Decision Trees for Classification and Regression

Decision trees are versatile machine learning models used for both classification and regression tasks. They work by recursively partitioning the data space and fitting a simple prediction model within each partition.

Example

The left panel of Figure 4.1 shows a training set of 15 two-dimensional points (two features) falling into two classes (red and blue). How should the new feature vector (black point) be classified?

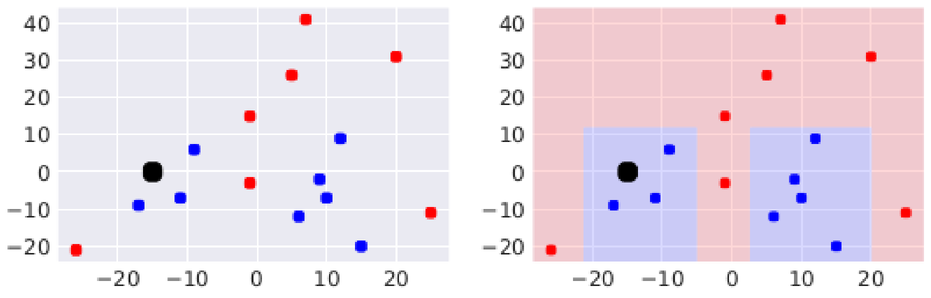

Figure 4.1: Left: training data and a new feature. Right: a partition of the feature space.

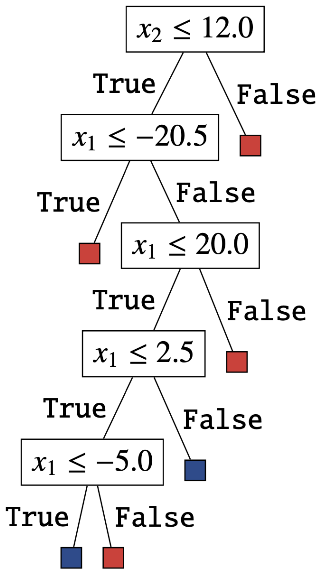

Figure 4.2: The decision- tree that corresponds to the partition in Figure 4.1.

Decision Tree Construction

1. **Splitting Criterion**: A decision tree splits the data at each node based on a criterion that maximizes the separation of the data. For classification, the common criteria are Gini impurity and information gain, while for regression, it is typically the reduction in variance.

Optimal split: Select the feature j^* and threshold t^* that yield the best split according to the chosen criterion. This involves finding the feature-threshold pair (j^*, t^*) that maximizes (or minimizes, depending on the criterion) the splitting criterion.

2. **Recursive Partitioning**: Starting with the full training dataset at the root node, the dataset at each node is recursively partitioned by selecting the feature and threshold that maximizes the chosen splitting criterion. For a feature j and a threshold t, split the dataset D into D_{left} and D_{right}:

$$D_{\text{left}} = \{(x, y) \in D \mid x_j \leq t\} \tag{4.1}$$

$$D_{\text{right}} = \{(x, y) \in D \mid x_j > t\} \tag{4.2}$$

where x_j is the value of feature j for the data point x. The process is repeated for each child node until a stopping criterion is met (e.g., maximum depth, minimum number of samples in a node).

3. **Stopping Criteria**: The recursive partitioning stops when one of the following criteria is met:

- Maximum tree depth is reached.
- Minimum number of samples in a node is less than a specified threshold.
- No further information gain or variance reduction is possible.

Splitting using Gini Index

One commonly used criterion for classification is the Gini Index for select the optimal feature j^* and threshold t^*. The Gini Index measures the impurity of a node, and our goal is to find the feature-threshold pair (j^*, t^*) that minimizes the Gini Index after the split, leading to purer child nodes. To find the best split, we follow these steps:

1. **Compute Gini Index for the parent node**: Calculate the Gini Index (or, Gini impurity) for the parent node before the split. At any node with dataset D, the Gini impurity is defined as:

$$\text{Gini}(D) = 1 - \sum_{i=1}^{C} p_i^2 \tag{4.3}$$

where p_i is the proportion of samples belonging to class i in dataset D, and C is the number of classes.

2. **Evaluate all possible splits**: For each feature j and each possible threshold t (which can be a midpoint between any two consecutive values of the feature), split the data into two subsets:

$$D_{\text{left}} = \{(x_i, y_i) \mid x_i[j] \leq t\}$$
$$D_{\text{right}} = \{(x_i, y_i) \mid x_i[j] > t\}$$

3. **Compute Gini Index for the Child Nodes**: Calculate the Gini Index for each of the child nodes resulting from the split.

4. **Calculate Weighted Gini Index for the Split**: Compute the weighted Gini Index for the split as follows:

$$G_{\text{split}}(j,t) = \frac{N_{\text{left}}}{N} G(D_{\text{left}}) + \frac{N_{\text{right}}}{N} G(D_{\text{right}})$$

where N is the total number of instances in the parent node, N_{left} and N_{right} are the number of instances in the left and right child nodes respectively, and $G(D_{\text{left}})$ and $G(D_{\text{right}})$ are the Gini Indices for the left and right child nodes.

5. **Select the Best Split**: Find the feature-threshold pair (j^*, t^*) that minimizes the weighted Gini Index:

$$(j^*, t^*) = \arg\min_{(j,t)} G_{\text{split}}(j,t)$$

Remark

By selecting the feature j^* and threshold t^* that minimize the weighted Gini Index, we ensure that the resulting child nodes are as pure as possible, which leads to more effective splits and, ultimately, a more accurate decision tree.

Remark

For classification, instead of Gini Index, we can use *entropy impurity* defined by

$$\text{Entropy}(D) = -\sum_{i=1}^{C} p_i \log_2 p_i. \tag{4.4}$$

Example

Let's consider a simple example with a binary classification problem (two classes: 0 and 1) and one feature 1.

Compute the Gini Index at the parent node: Suppose we have 10 instances at the parent node, with 6 instances of class 0 and 4 instances of class 1. The Gini Index for the parent node is:

$$G_{\text{parent}} = 1 - \left(\left(\frac{6}{10} \right)^2 + \left(\frac{4}{10} \right)^2 \right) = 1 - (0.36 + 0.16) = 1 - 0.52 = 0.48$$

Evaluate All Possible Splits: Assume the feature has the values $\{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$ with corresponding class labels $\{0, 0, 1, 1, 0, 0, 1, 1, 0, 1\}$. We evaluate splits at midpoints between consecutive values.

Compute Gini Index for Child Nodes: For a threshold $t = 5.5$:

$$D_{\text{left}} = \{1, 2, 3, 4, 5\} \quad \text{(3 instances of class 0, 2 instances of class 1)}$$

$$D_{\text{right}} = \{6, 7, 8, 9, 10\} \quad \text{(3 instances of class 1, 2 instances of class 0)}$$

$$G(D_{\text{left}}) = 1 - \left(\left(\frac{3}{5} \right)^2 + \left(\frac{2}{5} \right)^2 \right) = 1 - (0.36 + 0.16) = 1 - 0.52 = 0.48$$

$$G(D_{\text{right}}) = 1 - \left(\left(\frac{2}{5} \right)^2 + \left(\frac{3}{5} \right)^2 \right) = 1 - (0.16 + 0.36) = 1 - 0.52 = 0.48$$

Calculate Weighted Gini Index for the Split:

$$G_{\text{split}}(1, 5.5) = \frac{5}{10} \times 0.48 + \frac{5}{10} \times 0.48 = 0.48$$

Select the Best Split: Repeat the above calculations for all possible thresholds and select the one that minimizes the weighted Gini Index.

Splitting Criterion using Sum of Squared Errors (SSE)

For regression tasks, a commonly used criterion is the Sum of Squared Errors (SSE) to select the optimal feature j^* and threshold t^*. The SSE measures the variance within a node, and our goal is to find the feature-threshold pair (j^*, t^*) that minimizes the SSE after the split, leading to child nodes with lower variance. To find the best split, we follow these steps:

1. **Compute SSE for the parent node**: Calculate the SSE for the parent node before

the split. At any node with dataset D, the SSE is defined as:

$$\text{SSE}(D) = \sum_{i=1}^{N} (y_i - \bar{y})^2 \tag{4.5}$$

where y_i is the target value for instance i, $\bar{y}$ is the mean target value of the dataset D, and N is the number of instances in D.

2. **Evaluate all possible splits**: For each feature j and each possible threshold t (which can be a midpoint between any two consecutive values of the feature), split the data into two subsets:

$$D_{\text{left}} = \{(x_i, y_i) \mid x_i[j] \leq t\}$$

$$D_{\text{right}} = \{(x_i, y_i) \mid x_i[j] > t\}$$

3. **Compute SSE for the Child Nodes**: Calculate the SSE for each of the child nodes resulting from the split:

$$\text{SSE}(D_{\text{left}}) = \sum_{i \in D_{\text{left}}} (y_i - \bar{y}_{\text{left}})^2$$

$$\text{SSE}(D_{\text{right}}) = \sum_{i \in D_{\text{right}}} (y_i - \bar{y}_{\text{right}})^2$$

where $\bar{y}_{\text{left}}$ and $\bar{y}_{\text{right}}$ are the mean target values of the left and right child nodes, respectively.

4. **Calculate Weighted SSE for the Split**: Compute the weighted SSE for the split as follows:

$$\text{SSE}_{\text{split}}(j, t) = \frac{N_{\text{left}}}{N}\text{SSE}(D_{\text{left}}) + \frac{N_{\text{right}}}{N}\text{SSE}(D_{\text{right}})$$

where N is the total number of instances in the parent node, N_{left} and N_{right} are the number of instances in the left and right child nodes respectively, and $\text{SSE}(D_{\text{left}})$ and $\text{SSE}(D_{\text{right}})$ are the SSE for the left and right child nodes.

By evaluating all possible splits and selecting the one that minimizes the weighted SSE, the decision tree is able to create splits that reduce the overall variance in the target variable, leading to more accurate predictions.

Prediction

1. **Classification**: For a given input x, traverse the tree from the root to a leaf node. The predicted class $\hat{y}$ is the majority class in the leaf node:

$$\hat{y} = \arg\max_{c \in C} \left(\frac{1}{|D_{\text{leaf}}|} \sum_{(x_i, y_i) \in D_{\text{leaf}}} \mathbb{I}(y_i = c) \right) \tag{4.6}$$

where D_{leaf} is the dataset in the leaf node and $\mathbb{I}$ is the indicator function.

2. **Regression**: For a given input x, traverse the tree from the root to a leaf node. The predicted value $\hat{y}$ is the mean of the target values in the leaf node:

$$\hat{y} = \frac{1}{|D_{\text{leaf}}|} \sum_{(x_i,y_i)\in D_{\text{leaf}}} y_i \tag{4.7}$$

Evaluation

1. **Performance Metrics**: Evaluate the performance of the decision tree using appropriate metrics. For classification, common metrics include accuracy, precision, recall, and the F1 score. For regression, common metrics include Mean Squared Error (MSE) and Mean Absolute Error (MAE).

$$\text{Accuracy} = \frac{1}{N}\sum_{i=1}^{N} \mathbb{I}(\hat{y}_i = y_i) \tag{4.8}$$

$$\text{MSE} = \frac{1}{N}\sum_{i=1}^{N} (\hat{y}_i - y_i)^2 \tag{4.9}$$

$$\text{MAE} = \frac{1}{N}\sum_{i=1}^{N} |\hat{y}_i - y_i| \tag{4.10}$$

2. **Visualization**: Visualize the decision tree to understand the decision rules and the structure of the model. Tools such as tree diagrams can be used to illustrate the splits and the criteria used at each node.

4.2 Introduction to Ensemble Methods

Ensemble methods are a powerful technique in machine learning that combine multiple base models to produce a single, more robust predictive model. The fundamental idea is that by aggregating the predictions of several models, the ensemble can achieve better generalization performance than any individual model. This approach leverages the diversity among the base models to reduce variance, bias, or both.

Basic Concepts

- **Base Models (Learners):** Let $h_1, h_2, \ldots, h_M$ denote M base models trained on the same dataset $\mathcal{D}$. Each model h_i makes a prediction $h_i(x)$ for a given input x.

- **Ensemble Model:** The ensemble model H combines the predictions of the M base models. The way these predictions are combined defines the type of ensemble method. Formally, the prediction of the ensemble model $H(x)$ can be represented as:

  $$H(x) = \mathcal{F}(h_1(x), h_2(x), \ldots, h_M(x)),$$

  where $\mathcal{F}$ is a function that aggregates the base model predictions.

Types of Ensemble Methods

Bagging (Bootstrap Aggregating)

Bagging aims to reduce variance by training each base model on a different bootstrap sample (a random sample with replacement) of the original dataset. The final prediction is typically an average (for regression) or majority vote (for classification) of the base model predictions.

- **Mathematical Formulation:**

  $$H(x) = \frac{1}{M} \sum_{i=1}^{M} h_i(x)$$

  for regression, or

  $$H(x) = \text{mode}\{h_1(x), h_2(x), \ldots, h_M(x)\}$$

  for classification.

Pasting

Pasting is a method similar to bagging, but it involves creating subsets of the data without replacement. This means each subset contains unique instances from the original dataset, and no instance is repeated within a single subset.

- **Mathematical Formulation:** Given a dataset $D = \{(x_i, y_i)\}_{i=1}^{N}$, pasting generates M subsets $D_1, D_2, \ldots, D_M$, each created by sampling without replacement. Each base learner h_i is trained on D_i, and the final prediction is an average (for regression) or majority vote (for classification) of the base learners:

  $$H(x) = \frac{1}{M} \sum_{i=1}^{M} h_i(x)$$

  for regression, or

  $$H(x) = \text{mode}\{h_1(x), h_2(x), \ldots, h_M(x)\}$$

  for classification.

Boosting

Boosting focuses on reducing bias by sequentially training base models. Each new model is trained to correct the errors made by the previous models. The final prediction is a weighted sum of the base model predictions.

- **Mathematical Formulation:**

  $$H(x) = \sum_{i=1}^{M} \alpha_i h_i(x),$$

  where α_i are the weights assigned to each base model, often based on their performance.

Stacking

Stacking involves training a meta-model to combine the predictions of multiple base models. The base models are trained on the original dataset, and the meta-model is trained on the predictions of the base models.

- **Mathematical Formulation:** Let $\mathbf{H}(x) = (h_1(x), h_2(x), \ldots, h_M(x))$ be the vector of base model predictions. The meta-model g takes this vector as input:

  $$H(x) = g(\mathbf{H}(x)).$$

Advantages of Ensemble Methods

- **Improved Performance:** By combining multiple models, ensemble methods often achieve higher accuracy and robustness compared to single models.
- **Reduction in Overfitting:** Techniques like bagging reduce overfitting by averaging out the predictions, which helps to smooth out the errors made by individual models.
- **Flexibility:** Ensemble methods can be applied to a wide range of base models, including decision trees, neural networks, and more.

Theoretical Insights

- **Bias-Variance Trade-off:** Ensemble methods, particularly bagging and boosting, help to manage the bias-variance trade-off. Bagging primarily reduces variance, while boosting can reduce both bias and variance by focusing on difficult-to-predict instances.

- **Error Decomposition:** The prediction error of an ensemble model can be decomposed into three components: bias, variance, and noise. The goal of ensemble methods is to reduce the bias and variance components, thus minimizing the overall prediction error.
- **Diversity:** The success of ensemble methods heavily relies on the diversity among the base models. Diverse models make different errors, which can be averaged out to improve overall performance.

4.3 Out-of-Bag (OOB) Approach in Ensemble Methods

The Out-of-Bag (OOB) approach is a technique used to evaluate the performance of ensemble methods, specifically in bagging. It leverages the bootstrap sampling process to provide an internal estimate of the model's performance without the need for a separate validation set.

Bagging and Bootstrap Sampling

Recall that in bagging, multiple base models are trained on different bootstrap samples of the original dataset. A bootstrap sample is created by randomly selecting data points from the original dataset with replacement. Consequently, some data points are included multiple times in a bootstrap sample, while others are excluded.

Out-of-Bag Samples

For each bootstrap sample, the data points not selected are referred to as out-of-bag (OOB) samples. These OOB samples can be used to evaluate the performance of the model trained on the corresponding bootstrap sample.

Mathematically, for a dataset $D = \{(x_i, y_i)\}_{i=1}^{N}$:

1. Generate M bootstrap samples $D_1, D_2, \ldots, D_M$ by sampling N instances with replacement from D.
2. For each bootstrap sample D_m, identify the out-of-bag samples D_m^{OOB} which are the data points not included in D_m.
3. Train the m-th base model h_m on the bootstrap sample D_m.

OOB Prediction and Error

To estimate the performance of the ensemble using OOB samples, follow these steps:

OOB Predictions

For each data point $(x_i, y_i) \in D$, collect the predictions from all base models for which (x_i, y_i) was an OOB sample. Let $\mathcal{M}_i$ be the set of models for which (x_i, y_i) is OOB.

The OOB prediction for x_i is:

$$\hat{y}_i^{\text{OOB}} = \frac{1}{|\mathcal{M}_i|} \sum_{m \in \mathcal{M}_i} h_m(x_i)$$

where $|\mathcal{M}_i|$ is the number of models in $\mathcal{M}_i$.

OOB Error

Calculate the OOB error by comparing the OOB predictions $\hat{y}_i^{\text{OOB}}$ with the true values y_i. For regression, the OOB error can be measured using Mean Squared Error (MSE):

$$\text{MSE}_{\text{OOB}} = \frac{1}{N} \sum_{i=1}^{N} (\hat{y}_i^{\text{OOB}} - y_i)^2$$

For classification, the OOB error can be measured using the misclassification rate:

$$\text{Error}_{\text{OOB}} = \frac{1}{N} \sum_{i=1}^{N} \mathbb{I}(\hat{y}_i^{\text{OOB}} \neq y_i)$$

where $\mathbb{I}(\cdot)$ is the indicator function, which is 1 if the condition is true and 0 otherwise.

Advantages of OOB Approach

- **No Need for a Separate Validation Set:** The OOB approach provides an internal estimate of model performance, eliminating the need to set aside a separate validation set.
- **Efficient Use of Data:** Since all data points are used both for training and validation (but in different bootstrap samples), the OOB approach makes efficient use of the available data.

- **Unbiased Estimate:** The OOB error is an unbiased estimate of the true error, as each data point is evaluated on models that have not seen it during training.

The Out-of-Bag (OOB) approach is a valuable method for evaluating ensemble models, particularly bagging. By using the data points excluded from each bootstrap sample, the OOB approach provides an unbiased and efficient estimate of the model's performance. This technique leverages the inherent properties of bootstrap sampling to validate the model without requiring a separate validation set, thereby maximizing the use of available data.

4.4 Random Forests

Random Forests are an ensemble learning method primarily used for classification and regression tasks. They operate by constructing multiple decision trees during training and outputting the mode of the classes (classification) or mean prediction (regression) of the individual trees. This approach leverages the power of multiple models to improve the overall performance and robustness of the prediction.

Construction of Random Forests

The construction of a Random Forest involves the following steps:

1. **Bootstrap Sampling**: Given a training dataset D with N instances, create B bootstrap samples D_b (where $b = 1, 2, \ldots, B$). Each bootstrap sample is generated by randomly sampling N instances from D with replacement.
2. **Training Decision Trees**: For each bootstrap sample D_b, train an unpruned decision tree T_b. During the training of each tree, at each node, a random subset of features is selected from the total set of features. The best feature-threshold pair is chosen only from this subset to perform the split.
3. **Aggregating Predictions**: For a classification problem, the final prediction of the Random Forest is obtained by majority voting among the B decision trees. For a regression problem, the final prediction is obtained by averaging the predictions of the B decision trees.

Mathematical Formulation

Let $D = \{(x_i, y_i)\}_{i=1}^{N}$ be the training dataset, where x_i represents the feature vector and y_i represents the target variable. The steps can be mathematically formulated as follows:

1. **Bootstrap Sampling**:
$$D_b = \{(x_i, y_i)\}_{i=1}^N, \quad \text{where } D_b \text{ is sampled with replacement from } D$$

2. **Training Decision Trees**: For each bootstrap sample D_b, a decision tree T_b is trained. At each node, select a random subset of features $F \subseteq \{1, 2, \ldots, p\}$ (where p is the total number of features), and determine the best split:
$$(j^*, t^*) = \arg\min_{(A_j, t)} \text{Impurity}(D_{\text{left}}, D_{\text{right}})$$
where Impurity could be the Gini Index, Entropy, or SSE, depending on the problem type.

3. **Aggregating Predictions**: For a classification problem, the final prediction $\hat{y}$ for a new instance x is:
$$\hat{y} = \text{mode}\{T_b(x)\}_{b=1}^B$$
For a regression problem, the final prediction $\hat{y}$ is:
$$\hat{y} = \frac{1}{B} \sum_{b=1}^B T_b(x)$$

Out-of-Bag Error Estimation

An important property of Random Forests is the ability to estimate the generalization error using Out-of-Bag (OOB) samples. For each bootstrap sample D_b, approximately one-third of the original instances are not included (out-of-bag samples).

The OOB error estimation involves the following steps:

1. For each instance (x_i, y_i) in the original dataset, collect the predictions from all trees that did not use (x_i, y_i) in their bootstrap sample.

2. Aggregate these predictions (majority vote for classification or mean for regression) to obtain the OOB prediction $\hat{y}_{\text{OOB},i}$.

3. Compute the OOB error as follows:
$$\text{OOB Error} = \frac{1}{N} \sum_{i=1}^N \mathbb{I}(\hat{y}_{\text{OOB},i} \neq y_i) \quad \text{(for classification)}$$
$$\text{OOB Error} = \frac{1}{N} \sum_{i=1}^N (y_i - \hat{y}_{\text{OOB},i})^2 \quad \text{(for regression)}$$

By aggregating the predictions from multiple decision trees and leveraging bootstrap sampling, Random Forests achieve better generalization performance and robustness compared to individual decision trees.

4.5 Boosting Methods

- Boosting is a popular ensemble learning technique that combines multiple weak learners (typically decision trees) to create a strong learner. It works by sequentially training new models, where each model focuses on correcting the errors made by its predecessors. The final prediction is made by aggregating the predictions of all models, typically using a weighted sum.
- Boosting was initially designed for binary classification tasks, but it can be easily adapted for general classification and regression problems.
- While boosting shares similarities with bagging in that both methods utilize an ensemble of predictive models, there is a key difference between them. In bagging, predictive models are fitted to bootstrapped subsets of the data independently. In contrast, boosting involves a sequential learning process where each model is trained using information from the previous models.

Boosting Algorithm

- Similar to other ensemble methods, such as the bagging method, we have a collection of models, denoted as $h_1, \ldots, h_M$ such that the final model is a function of all the models.
- The idea is to start with a simple model (weak learner) h_1 for the given training data $D = (x_i, y_i)_{i=1}^n$, and then to improve or "boost" this learner to a learner
$$g_2 := h_1 + \gamma_2 h_2.$$
Here, the function h_2 is found by minimizing the training loss for $h_1 + h$ over all functions h in some class of functions $\mathcal{H}$. For example, $\mathcal{H}$ could be the set of prediction functions that can be obtained via a decision tree of maximal depth 3.
- The parameter γ_2 helps in reducing overfitting.
- Given a loss function Loss, the function h_2 is thus obtained as the solution to the optimization problem
$$h_2 = \arg\min_{h \in \mathcal{H}} \frac{1}{n} \sum_{i=1}^{n} \mathsf{Loss}(y_i, h_1(x_i) + h(x_i)).$$
- This process is repeated to get $g_3 = g_2 + \gamma_3 h_3 = h_1 + \gamma_2 h_2 + \gamma_3 h_3$ via
$$h_3 = \arg\min_{h \in \mathcal{H}} \frac{1}{n} \sum_{i=1}^{n} \mathsf{Loss}(y_i, g_2(x_i) + h(x_i)).$$
This is done until we get $g_M = g_{M-1} + \gamma_M h_M = \sum_{m=1}^{M} \gamma_m h_m$, with $\gamma_1 = 1$.

```
Algorithm 1: Regression Boosting with Squared-Error Loss
Input: Dataset $\{(x_1, y_1), \dots, (x_n, y_n)\}$,
       Number of boosting rounds $M$, and
       shrinkage parameters $1 = \gamma_1, \gamma_2, \dots, \gamma_M$
Output: Boosted prediction function $g_M = \sum_{m=1}^{M} \gamma_m h_m$

Set $g_1(x) = h_1(x) \leftarrow \frac{1}{n} \sum_{i=1}^{n} y_i$.
for $m = 2, \dots, M$ do
    Set $\epsilon_i^{(m)} \leftarrow y_i - g_{m-1}(x_i)$ for all $i = 1, \dots, n$
    Let $D_m \leftarrow \{(x_i, \epsilon_i^{(m)})\}_{i=1}^{n}$
    Fit a model $h_m$ using the new dataset $D_m$ and mean-squared error as loss
    Set $g_m \leftarrow g_{m-1} + \gamma_m h_m$.
end
Return $g_M$
```

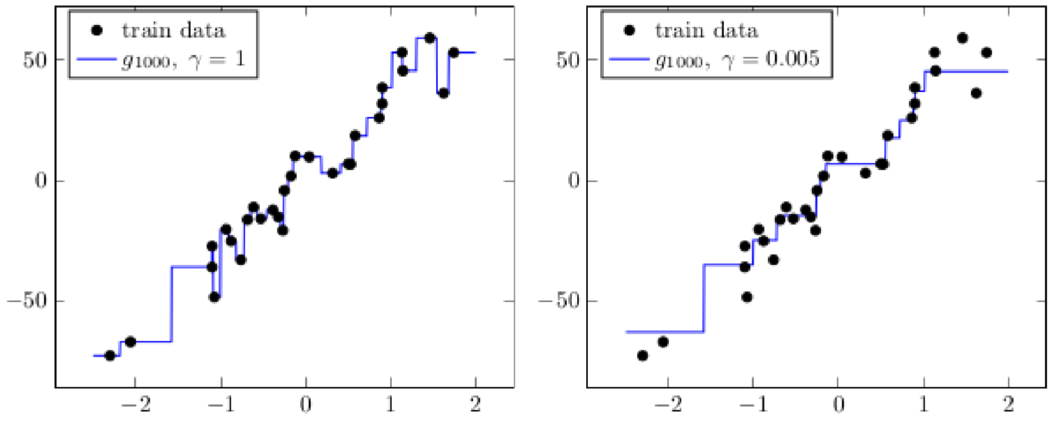

Figure 4.3: The left and the right panels show the fitted boosting regression model g_{1000} with $\gamma = 1.0$ and $\gamma = 0.005$, respectively. Note the overfitting on the left.

Remark

The γ parameters controls the speed of the fitting process. Suppose assume that $\gamma_2 = \gamma_3 = \cdots = \gamma_M = \gamma$. For small values of γ, boosting takes smaller steps towards the training loss minimization. The step-size γ is of great practical importance, since it helps the boosting algorithm to avoid overfitting. This phenomenon is demonstrated in Figure 4.3.

Gradient Boosting

- In the above algorithm, the parameter γ_m can be viewed as a step size made in the direction of the negative gradient of the squared-error training loss during the m-th update.
- To see this,

$$-\frac{\partial \mathsf{Loss}(y_i, z)}{\partial z}\Big|_{z=g_{m-1}(x_i)} = -\frac{\partial (y_i - z)^2}{\partial z}\Big|_{z=g_{m-1}(x_i)} = 2(y_i - g_{m-1}(x_i)) = 2\epsilon_i^{(m)}.$$

- Since h_m is trained to predict $\epsilon_i^{(m)}$, we have

$$g_m(x_i) = g_{m-1}(x_i) + \gamma_m h_m(x_i) \approx g_{m-1}(x_i) + \gamma_m \epsilon_i^{(m)}.$$

  Alternatively.

$$g_m(x_i) \approx g_{m-1}(x_i) - \frac{\gamma_m}{2}\frac{\partial \mathsf{Loss}(y_i, z)}{\partial z}\Big|_{z=g_{m-1}(x_i)},$$

  which is similar to update in gradient descent optimization method.

- One of the significant breakthroughs in boosting theory was the realization that a gradient descent method could be applied to any differentiable loss function. This led to the development of the algorithm known as *gradient boosting*.
- In particular, in gradient boosting, in the m-th iteration, we compute

$$r_i^{(m)} = -\frac{\partial \mathsf{Loss}(y_i, z)}{\partial z}\Big|_{z=g_{m-1}(x_i)}, \quad i = 1, \dots, n,$$

  and obtain h_m as

$$h_m = \operatorname*{arg\,min}_{h \in \mathcal{H}} \frac{1}{n}\sum_{i=1}^{n}(r_i^{(m)} - h(x_i))^2.$$

AdaBoost (Adaptive Boosting)

- AdaBoost is another popular boosting algorithm. It assigns higher weights to the misclassified samples in each iteration, allowing subsequent weak learners to focus more on the difficult-to-classify instances.
- The idea of AdaBoost is similar to the one presented in the regression setting, that is, AdaBoost fits a sequence of prediction functions $g_1 = h_1, g_2 = h_1 + h_2, \dots$ with final prediction function

$$g_M = \sum_{m=1}^{M} h_m,$$

where each weak learner h_m is of the form

$$h_m(x) = \gamma_m c_m(x),$$

with c_m is a proper classifier from a class of weaker classifiers $\mathcal{C}$. To get this classifier, we solve

$$(\gamma_m, c_m) = \arg\min_{\gamma \geq 0, c \in \mathcal{C}} \frac{1}{n} \sum_{i=1}^{n} \mathsf{Loss}(y_i, g_{m-1}(x_i) + \gamma c(x_i)).$$

Remark

The inventors of the AdaBoost method considered a binary classification problem, where the response variable belongs to the $-1, 1$ set. In this case the loss function is defined as

$$\mathsf{Loss}(y, \hat{y}) = \exp(-y\hat{y}).$$

The AdaBoost algorithm can be summarized as follows:

1. Initialize sample weights $w_i = \frac{1}{N}$ for all training samples.
2. For $m = 1$ to M:
   (a) Train a weak learner using the weighted training data and calculate the error.
   (b) Compute the weight α_m of the weak learner based on its error.
   (c) Update the sample weights to give more weight to the misclassified samples.
3. Aggregate the predictions of all weak learners using the weighted sum.

Description of AdaBoost Steps

Step (a): Train a Weak Learner and Calculate the Error

In this step, we train a weak learner using the weighted training data and calculate the error. Let's denote the weak learner as $h_m(x)$, where m represents the m-th iteration. The error of the weak learner $h_m(x)$ on the training set can be calculated using a loss function $L(y, h_m(x))$, where y represents the true labels.

Mathematically, the error ϵ_m of the weak learner $h_m(x)$ can be expressed as:

$$\epsilon_m = \sum_{i=1}^{N} w_i^{(m)} \cdot \mathbb{I}(y_i \neq h_m(x_i))$$

where:

- N is the total number of training samples,
- $w_i^{(m)}$ is the weight of the i-th training sample at iteration m,
- $\mathbb{I}(\cdot)$ is the indicator function that returns 1 if the condition inside is true and 0 otherwise.

Step (b): Compute the Weight α_m of the Weak Learner

In this step, we compute the weight α_m of the weak learner based on its error. The weight α_m is determined by the performance of the weak learner, with better-performing weak learners being assigned higher weights.

Mathematically, the weight α_m of the weak learner $h_m(x)$ can be calculated as:

$$\alpha_m = \frac{1}{2} \ln \left(\frac{1 - \epsilon_m}{\epsilon_m} \right)$$

where ϵ_m is the error of the weak learner as calculated in Step (a).

Step (c): Update the Sample Weights

In this step, we update the sample weights to give more weight to the misclassified samples. The idea is to make the misclassified samples more influential in the subsequent iterations, so the weak learners can focus more on correcting their errors.

Mathematically, the updated weight $w_i^{(m+1)}$ of the i-th training sample at iteration $m + 1$ can be calculated as:

$$w_i^{(m+1)} = w_i^{(m)} \cdot \exp\left(-\alpha_m \cdot y_i \cdot h_m(x_i)\right)$$

where:

- α_m is the weight of the weak learner $h_m(x)$,
- y_i is the true label of the i-th training sample, taking values in $\{-1, +1\}$,
- $h_m(x_i) \in \{-1, +1\}$ is the prediction of the weak learner $h_m(x)$ for the i-th training sample.

By updating the sample weights in this manner, the misclassified samples receive higher weights, and the subsequent weak learners focus more on correcting their errors.