Ensemble Learning Course Notes

Latreche Sara

June 23, 2025

Contents

1 Motivation: Why Ensemble Learning?

A Theorem from 1785

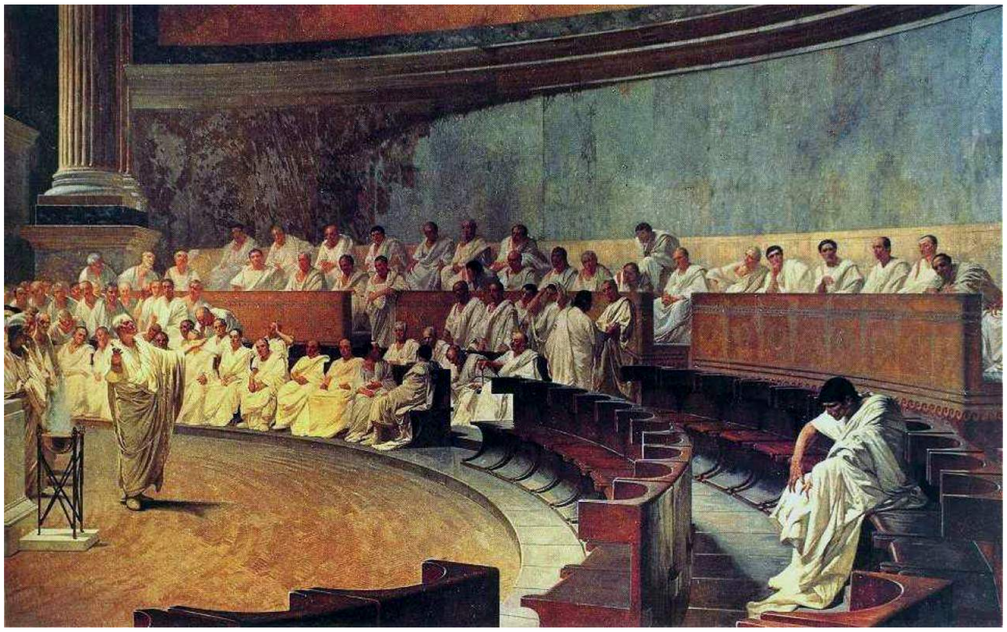

Figure 1: Jury Theorem

All the way back in 1785, Marquis de Condorcet, a French philosopher and mathematician, introduced a powerful idea now known as **Condorcet's Jury Theorem**. The theorem states:

Condorcet's Jury Theorem

If each member of a jury has a probability $p > 0.5$ of making the correct decision independently, then the probability that the majority decision is correct increases as more members are added. As the jury size grows, the probability approaches 1.

This idea surprisingly echoes in modern machine learning. If we treat each model (say, a decision tree) as a juror, and each one is slightly better than random (e.g., 60% accuracy), then a majority vote from multiple such models can result in significantly higher overall accuracy.

For instance, if three decision trees each predict with 60% accuracy, their majority vote has an accuracy of about 65%. With 11 trees, the ensemble can achieve around 75% accuracy — assuming the predictions are independent and diverse.

Wisdom of the Crowd

Suppose you pose a complex question to thousands of random people, then aggregate their answers. Surprisingly, the average response is often more accurate than that of a single expert. This phenomenon is called the **Wisdom of the Crowd**.

In machine learning, we apply a similar strategy. Instead of relying on a single model, we combine multiple models to produce a stronger, more reliable prediction. This is the core idea behind:

Ensemble Learning

Ensemble Learning is the technique of combining multiple models (classifiers or regressors) to produce a better predictive model.

A group of models is called an **ensemble**, and the algorithms that build and combine them are called **ensemble methods**.

A Powerful Example: Random Forests

Imagine training several decision trees, each on a different random subset of the training data. To predict a label, you ask each tree to vote, and select the majority class. This simple yet powerful ensemble method is called a **Random Forest**. Despite its simplicity, Random Forests are among the most effective and widely-used machine learning algorithms.

Ensemble methods are often used at the end of a project to combine multiple good models into an even better one. In fact, most top-performing solutions in machine learning competitions (such as Kaggle or the Netflix Prize) involve ensemble methods like **bagging**, **boosting**, or **stacking**.

In the remainder of this chapter, we will study these ensemble techniques in detail.

2 Voting Classifiers

Earlier, we saw that combining multiple diverse models can lead to improved performance — this is the foundation of ensemble learning. But how exactly do we combine predictions from different models?

One of the simplest and most intuitive ensemble strategies is the **Voting Classifier**. Here, several different classifiers (e.g., Logistic Regression, SVM, Random Forest) are trained independently, and their predictions are aggregated to make a final decision.

Hard Voting

In hard voting, each classifier casts a "vote" for a class label, and the majority wins. For instance, if three classifiers predict `[0, 1, 1]`, the final output is `1`.

This method works well when individual models are diverse and slightly better than random guessing — their collective decision tends to be more reliable.

Soft Voting

When classifiers can estimate class probabilities, soft voting often performs better. Instead of majority class labels, we average the predicted probabilities and select the class with the highest mean.

Soft voting is especially effective when models are well-calibrated.

Summary

- Hard voting uses majority labels.
- Soft voting uses average probabilities (preferred when possible).
- Voting classifiers often outperform single models, especially when the models are diverse and independent.

This sets the stage for more advanced ensemble methods like bagging, boosting, and stacking, which we'll explore next.

3 Bagging and Pasting

Bagging, short for Bootstrap Aggregating, is an ensemble technique where multiple models are trained independently on different random subsets of the training data. These subsets are drawn *with replacement*, meaning the same data point can appear multiple times in a single subset. By averaging or voting over these models' predictions, bagging helps reduce variance and improves the overall robustness and accuracy of the ensemble.

A closely related technique is *pasting*, which also trains multiple models on random subsets of data, but samples these subsets *without replacement.* This ensures that the subsets are distinct, which can also promote diversity among the models and reduce overfitting, though the variance reduction might be less than in bagging.

Remark: Random Forests are a well-known example of bagging ensembles, where decision trees are trained on bootstrap samples of the data, combined with random feature selection at each split to further increase diversity and improve performance.

- Sampling Without Replacement (Pasting):When you sample without replacement, each data point in the training set can be selected only once per subset. Once it's picked, it's not returned to the pool. So, every subset contains unique examples.

- Bagging (with replacement): The same example may be reused in a single model's training set. This introduces randomness and diversity across models.

- Pasting (without replacement): Ensures every example in a model's subset is unique — no repeats — though the same example can still appear in other models' subsets.

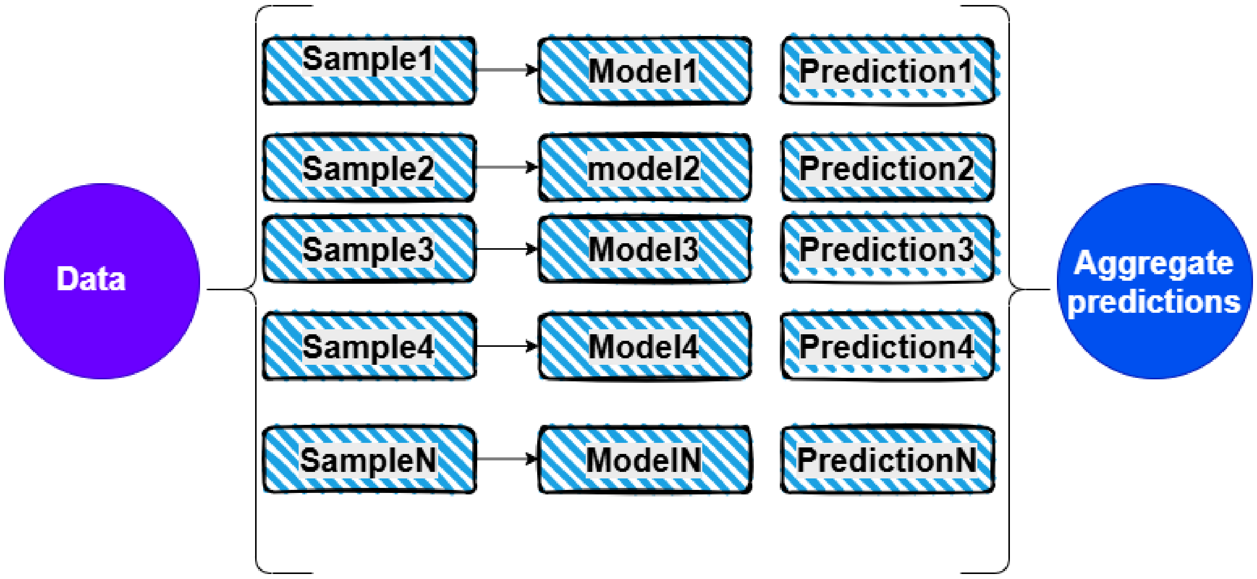

Figure 2: Illustration of Bagging (sampling with replacement)

3.1 Random Forest

Random Forest is one of the most popular and powerful ensemble methods in machine learning. It is essentially a collection of Decision Trees trained using the bagging technique — that is, each tree is trained on a bootstrap sample of the training data.

But Random Forests go further: during tree construction, each split considers only a random subset of the features rather than all features. This added randomness helps to decorrelate the trees, improving the ensemble's overall performance and reducing overfitting.

The final prediction is obtained by aggregating the predictions of all trees: algorithm algpseudocode

Example: Applying Random Forest on a Simple Dataset

Consider the following dataset:

Algorithm 1 Random Forest

Precondition: A training set $S := \{(x_1, y_1), \ldots, (x_n, y_n)\}$, features F, and number of trees B.

```
1: function RandomForest($S$, $F$)
2:     $H \leftarrow \emptyset$
3:     for $i = 1$ to $B$ do
4:         $S^{(i)} \leftarrow$ A bootstrap sample from $S$
5:         $h_i \leftarrow$ RandomizedTreeLearn($S^{(i)}$, $F$)
6:         $H \leftarrow H \cup \{h_i\}$
7:     end for
8:     return $H$
9: end function
10: function RandomizedTreeLearn($S$, $F$)
11:     At each node:
12:         $f \leftarrow$ very small subset of $F$
13:         Split on best feature in $f$
14:     return The learned tree
15: end function
```

Instance	A	B	C	Target
1	1	0	1	yes
2	0	1	0	no
3	1	1	1	yes
4	0	0	1	no
5	1	0	0	yes

Step 1: Bootstrap Sampling

We train $B = 3$ trees, each on a bootstrap sample (sampled *with replacement*) from the dataset.

- Tree 1: Instances {1, 3, 5}
- Tree 2: Instances {2, 3, 4}
- Tree 3: Instances {1, 2, 4}

Step 2: Random Feature Selection

For each tree, we randomly select a subset of features to consider at each split:

- Tree 1: Features {A, C}
- Tree 2: Features {B, C}

- Tree 3: Features {A, B}

Step 3: Tree Construction

Each tree is trained independently using its bootstrap sample and the randomly selected features. The best splits at each node are chosen only from that tree's feature subset.

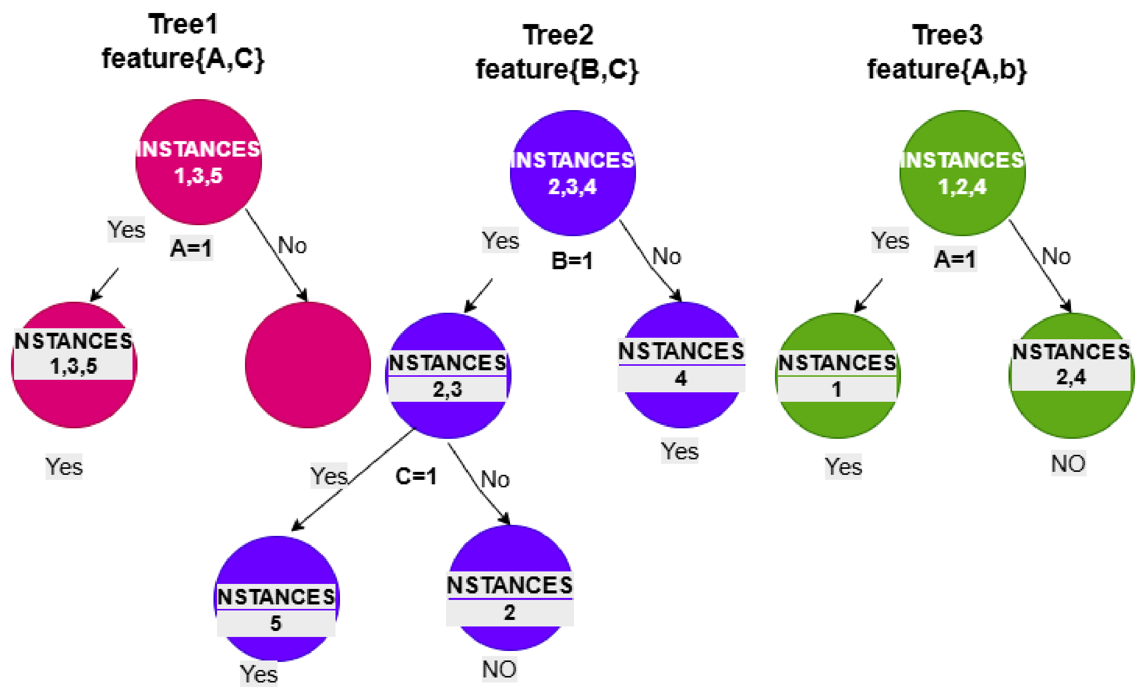

Figure 3: Decision trees trained on different bootstrap samples and random feature subsets.

Prediction: To classify a new instance let's say 6: with 1,1,0,yes we se what each tree predict tree1: says it's a yes,tree2 says its a no, tree3: says it's a no. AND the Random forest Vote for a yes.

This example demonstrates how Random Forest combines bootstrap sampling and random feature selection to build a diverse set of decision trees, improving overall model robustness.

Feature Importance. One of the key strengths of Random Forests is their ability to evaluate the relative importance of each feature in prediction. This is done by averaging how much each feature reduces impurity (e.g., Gini or entropy) across all the decision trees in the forest.each time a feature is used to split a node, the impurity decrease is multiplied by the number of samples that reach that node. The final importance is then normalized so that all importances sum to 1. This provides valuable insight into which features contribute most to the model's decisions.

4 Boosting

Boosting is an ensemble technique that aims to create a strong learner by sequentially combining multiple weak learners. Unlike bagging, where learners are trained independently, boosting trains them in sequence, with each new model attempting to correct the errors of its predecessor.

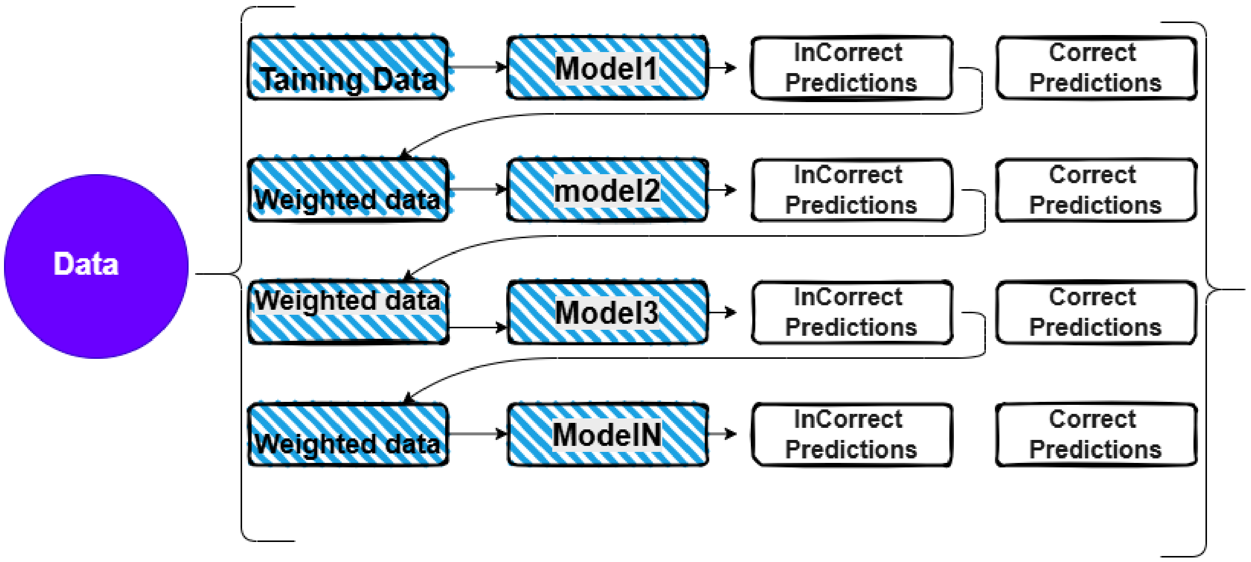

Figure 4: Boosting: sequentially combining weak learners, each focusing more on the errors of its predecessors.

4.1 AdaBoost (Adaptive Boosting)

AdaBoost is a popular boosting algorithm that adjusts the weights of training instances to focus more on hard-to-classify examples. The general idea is to sequentially apply a base learner (typically a decision stump) on weighted versions of the dataset, and then combine their outputs through a weighted majority vote.

Intuition

- All training instances start with equal weights.
- After each weak learner is trained, instance weights are updated: misclassified instances get higher weights.
- This forces the next learner to focus more on the hard examples.
- Each learner is given a weight (α) based on its accuracy.
- Final prediction is made by a weighted vote of all learners.

AdaBoost Algorithm

Algorithm 2 AdaBoost

```
Training set $S = \{(x_1, y_1), \ldots, (x_m, y_m)\}$, number of rounds $T$ Final strong classifier $H(x)$
Initialize weights $w_i = \frac{1}{m}$ for all $i = 1, \ldots, m$
for $t = 1$ $T$ do Train base classifier $h_t$ using distribution $w$   Compute weighted error:
        $$r_t = \frac{\sum_{i=1}^{m} w_i \cdot \mathbb{I}[h_t(x_i) \neq y_i]}{\sum_{i=1}^{m} w_i}$$
    Compute classifier weight:
        $$\alpha_t = \eta \cdot \log\left(\frac{1 - r_t}{r_t}\right)$$
    Update weights:
        $$w_i \leftarrow w_i \cdot \begin{cases} 1 & \text{if } h_t(x_i) = y_i \\ \exp(\alpha_t) & \text{if } h_t(x_i) \neq y_i \end{cases}$$
    Normalize weights: $\sum_{i=1}^{m} w_i = 1$   Final prediction:
        $$H(x) = \arg\max_y \sum_{t=1}^{T} \alpha_t \cdot \mathbb{I}[h_t(x) = y]$$
```

Key Properties

- Learners that perform worse than random get negative weights.
- If a perfect classifier is found (zero error), training may stop early.
- Final output is a weighted majority vote of the base classifiers.

4.2 AdaBoost Numerical Example

Consider a small binary classification dataset:

Instance	x_i	y_i
1	A	$+1$
2	B	$+1$
3	C	-1
4	D	-1

We use decision stumps as base learners and perform 3 boosting rounds.

Initialization

All weights are equal at first:

$$w_i^{(1)} = \frac{1}{4} = 0.25$$

Round 1

Train a decision stump h_1 that predicts:

$$h_1 = [+1, -1, -1, -1]$$

Only instance 2 is misclassified:

$$r_1 = 0.25, \quad \alpha_1 = \frac{1}{2} \log \left(\frac{1 - r_1}{r_1} \right) = \frac{1}{2} \log(3) \approx 0.55$$

Update weights:

$$w_2^{(2)} = w_2^{(1)} \cdot e^{\alpha_1} \approx 0.25 \cdot e^{0.55} \approx 0.43$$

Normalize all weights:

$$Z = 3 \cdot 0.25 + 0.43 = 1.18, \quad w_2^{(2)} \approx \frac{0.43}{1.18} \approx 0.364$$

Round 2

Train a new stump h_2, suppose it misclassifies instance 3. Then:

$$r_2 = 0.25, \quad \alpha_2 \approx 0.55$$

Instance 3's weight is boosted.

Round 3

Repeat the process. The learner focuses more on instance 3.

Final Strong Classifier

The final prediction is made by a weighted vote:

$$H(x) = \text{sign}(\alpha_1 h_1(x) + \alpha_2 h_2(x) + \alpha_3 h_3(x))$$

Each weak learner contributes according to its accuracy.

4.3 Gradient Boosting Methods

Gradient boosting methods (often abbreviated as **GBMs**) share many elements with AdaBoost. In particular, they still combine several weak learners in an additive fashion and work in a stage-wise manner—i.e., at each iteration, the previously learned models are left unchanged.

However, unlike AdaBoost, gradient boosting does not reweight training examples. Instead, it adjusts the **target function itself** at each step. The central idea is to fit each new model to the **residual errors** (i.e., the difference between the true values and the current prediction), rather than the raw outputs.

Intuition. Suppose we're trying to predict a target y. We start by making a simple prediction (e.g., the average of y). Then we fit a model to the *residuals*, which represent our current mistakes. In the next round, we try to fix the mistakes of the previous model by adding a new model that predicts the residuals. This continues, each new model focusing on the remaining errors of the ensemble so far.

This intuition can be generalized mathematically. Fitting to the residuals is equivalent to moving in the direction of the **negative gradient** of a loss function. For instance, for the MSE loss function:

$$\mathcal{L} = \sum (y - \hat{y})^2,$$

the gradient with respect to $\hat{y}$ is proportional to the residuals:

$$\frac{\partial \mathcal{L}}{\partial \hat{y}} = -2(y - \hat{y}).$$

Hence, each iteration of gradient boosting can be seen as taking a step in the direction of steepest descent to reduce the error. This is why it's called **gradient** boosting.

Generalization. This gradient-based view allows gradient boosting to work with any differentiable loss function:

- MSE (Mean Squared Error) for regression tasks
- MAE (Mean Absolute Error) or Huber loss for robustness to outliers
- Log loss (Cross-Entropy) for classification tasks

At each step, the algorithm fits a base learner (often a regression tree) to the gradient of the loss, rather than to the actual target values.

Algorithm 3 Gradient Boosting

```
Training data $\{(x_i, y_i)\}_{i=1}^n$, loss function $\mathcal{L}(y, f(x))$, number of iterations $M$, learning rate $\eta$
Final predictor $f_M(x)$
Initialize $f_0(x) = \arg\min_c \sum_{i=1}^n \mathcal{L}(y_i, c)$ (e.g., mean of $y_i$)

for $m = 1$ $M$ do Compute the negative gradients (residuals) for each example:
    $$r_i^{(m)} = -\left.\frac{\partial \mathcal{L}(y_i, f(x_i))}{\partial f(x_i)}\right|_{f=f_{m-1}(x)}$$
    Fit a base learner $h_m(x)$ to the residuals $\{(x_i, r_i^{(m)})\}$
    Update the model:
    $$f_m(x) = f_{m-1}(x) + \eta \cdot h_m(x)$$
return $f_M(x)$
```

Algorithm: Gradient Boosting (Friedman, 2001)

Regularization and Overfitting. As more trees are added, the ensemble can begin to overfit the training data. To mitigate this:

- Use a small learning rate η
- Limit the depth of the regression trees
- Use subsampling or early stopping

Summary. Gradient boosting is a flexible and powerful boosting technique that applies functional gradient descent to minimize a loss function by iteratively fitting weak learners to the negative gradient of the loss. Each new model fixes the errors made so far, resulting in a strong overall predictor.

Numerical Example of Gradient Boosting

To illustrate how gradient boosting works in practice, consider a very simple regression problem with three data points:

x_i	y_i
1	2
2	4
3	6

We will perform 1–2 iterations of gradient boosting using the Mean Squared Error (MSE) loss function, a learning rate $\eta = 1.0$, and simple constant learners (stumps that predict a single value).

Step 0: Initialization. We start by initializing the prediction with a constant value equal to the mean of the target values:

$$f_0(x) = \frac{2+4+6}{3} = 4$$

This gives initial predictions:

$$\hat{y}_i^{(0)} = 4 \quad \text{for all } i$$

Step 1: Compute residuals. We compute the residuals, which correspond to the negative gradients of the MSE loss:

$$r_i^{(1)} = y_i - \hat{y}_i^{(0)} = \begin{cases} 2-4=-2 \\ 4-4=0 \\ 6-4=+2 \end{cases}$$

Step 2: Fit weak learner to residuals. We now fit a weak learner $h_1(x)$ that tries to approximate these residuals. For simplicity, let:

$$h_1(x) = \begin{cases} -2 & \text{if } x=1 \\ 0 & \text{if } x=2 \\ +2 & \text{if } x=3 \end{cases}$$

Step 3: Update model. We add this learner to the previous model:

$$f_1(x) = f_0(x) + \eta \cdot h_1(x)$$

$$f_1(x) = \begin{cases} 4+(-2)=2 & \text{if } x=1 \\ 4+0=4 & \text{if } x=2 \\ 4+2=6 & \text{if } x=3 \end{cases}$$

Result: After just one iteration, the model perfectly fits the data. This example shows how gradient boosting corrects errors by iteratively fitting residuals. In real-world scenarios:

- Learners are more complex (e.g., trees).
- Learning rate η is smaller (e.g., 0.1).
- Many iterations are used to improve generalization gradually.

XGBoost (Extreme Gradient Boosting)

4.4 XGBoost (Extreme Gradient Boosting)

XGBoost (short for Extreme Gradient Boosting) is a powerful and scalable implementation of gradient boosting developed by Chen and Guestrin (2016). It builds on the core ideas of gradient boosting but introduces several important improvements that make it highly efficient and effective in practice.

Key enhancements of XGBoost:

- **Second-order optimization:** Unlike classical gradient boosting which uses only the first-order gradient (residuals), XGBoost also uses second-order derivatives (Hessians) of the loss function to better guide learning.
- **Regularization:** It includes L1 and L2 regularization terms in its objective function to penalize complex models and reduce overfitting.
- **Shrinkage (learning rate):** Like traditional boosting, XGBoost uses a shrinkage parameter η, but this is carefully combined with regularization for better control.
- **Column (feature) subsampling:** It allows random sampling of features at each boosting step, similar to random forests, improving model generalization and reducing computation.
- **Handling missing values:** XGBoost natively handles missing data by learning the best direction to take when encountering a missing value during training.

Objective Function: The learning objective at boosting iteration t combines a differentiable convex loss $\mathcal{L}$ (e.g., MSE or logistic loss) with a regularization term Ω:

$$\mathcal{L}^{(t)} = \sum_{i=1}^{n} \ell\left(y_i, \hat{y}_i^{(t-1)} + f_t(x_i)\right) + \Omega(f_t)$$

$$\text{where } \Omega(f) = \gamma T + \frac{1}{2}\lambda \sum_{j=1}^{T} w_j^2$$

Here, T is the number of leaves in the tree, and w_j are the leaf weights.

Taylor Expansion: To efficiently optimize this objective, XGBoost applies a second-order Taylor expansion of the loss around the current prediction:

$$\mathcal{L}^{(t)} \approx \sum_{i=1}^{n} \left[g_i f_t(x_i) + \frac{1}{2}h_i f_t(x_i)^2\right] + \Omega(f_t)$$

where:

$$g_i = \frac{\partial \ell(y_i, \hat{y}_i^{(t-1)})}{\partial \hat{y}_i^{(t-1)}}, \quad h_i = \frac{\partial^2 \ell(y_i, \hat{y}_i^{(t-1)})}{\partial (\hat{y}_i^{(t-1)})^2}$$

Pseudocode.

Algorithm 4 XGBoost (Simplified Pseudocode)

```
1: Initialize predictions $\hat{y}_i^{(0)}$ = constant
2: for $t = 1$ to $M$ (number of boosting rounds) do
3:     Compute gradients $g_i$ and Hessians $h_i$ for all training points
4:     Build a regression tree $f_t(x)$ that minimizes:
           $$\sum_{i \in \text{leaf}} \left[ g_i f_t(x_i) + \frac{1}{2} h_i f_t(x_i)^2 \right] + \Omega(f_t)$$
5:     Update the model:
           $$\hat{y}_i^{(t)} = \hat{y}_i^{(t-1)} + \eta f_t(x_i)$$
6: end for
7: return Final model $\hat{y}^{(M)}$ =0
```

Numerical example: Let us illustrate XGBoost with a small regression example. Suppose we have the following training data:

x_i	y_i
1	2.1
2	2.9
3	3.7
4	4.5

Step 1: Initial prediction.
We start with a constant prediction. For squared loss, a good choice is the mean of y_i:

$$\hat{y}^{(0)} = \frac{2.1 + 2.9 + 3.7 + 4.5}{4} = 3.3$$

Step 2: Compute gradients and Hessians.
For MSE loss, we have:

$$g_i = \hat{y}^{(0)} - y_i, \quad h_i = 1 \text{ (constant second derivative)}$$

x_i	g_i	h_i
1	$3.3 - 2.1 = 1.2$	1
2	$3.3 - 2.9 = 0.4$	1
3	$3.3 - 3.7 = -0.4$	1
4	$3.3 - 4.5 = -1.2$	1

Step 3: Fit a simple decision stump (tree with 1 split).
Suppose we try a split at $x = 2.5$:

$$\text{Left node } (x \leq 2.5)\text{: points 1 and 2,} \quad \text{Right node } (x > 2.5)\text{: points 3 and 4}$$

Step 4: Compute optimal leaf values.
XGBoost computes the score for a leaf as:

$$w^* = -\frac{\sum g_i}{\sum h_i + \lambda}$$

Assume regularization parameter $\lambda = 1$:

- Left node:

$$\sum g_i = 1.2 + 0.4 = 1.6, \quad \sum h_i = 2, \quad w_L = -\frac{1.6}{2+1} = -0.533$$

- Right node:

$$\sum g_i = -0.4 - 1.2 = -1.6, \quad \sum h_i = 2, \quad w_R = -\frac{-1.6}{2+1} = +0.533$$

Step 5: Update predictions.
Assume learning rate $\eta = 0.1$:

$$\hat{y}_i^{(1)} = \hat{y}^{(0)} + \eta \cdot w_{\text{leaf}_i}$$

x_i	$\hat{y}^{(0)}$	Leaf update	$\hat{y}^{(1)}$
1	3.3	-0.533	$3.3 - 0.053 = 3.247$
2	3.3	-0.533	3.247
3	3.3	$+0.533$	$3.3 + 0.053 = 3.353$
4	3.3	$+0.533$	3.353

Thus, the model has been slightly adjusted toward better predictions based on the first boosting round.

Step 6: Repeat.
New gradients would be computed at the updated predictions, and another tree would be fit to these new gradients. This continues iteratively, with each new tree making smaller corrective updates.

5 Conclusion

Ensemble learning is a powerful paradigm that can boost model performance, especially in complex tasks. It is widely used in practice and in competitions like Kaggle.

6 References

References

[1] T. Hastie, R. Tibshirani, J. Friedman. (2009). *The Elements of Statistical Learning*, Springer.

[2] Scikit-learn documentation. https://scikit-learn.org

[3] Zhi-Hua Zhou. (2012). *Ensemble Methods: Foundations and Algorithms*, CRC Press.