CHAPTER 7

Ensemble Learning and Random Forests

Ensemble Learning is built on the "wisdom of the crowd" idea: combining multiple models (predictors) often produces better generalization than relying on a single best model. The key reason is error reduction through aggregation—if individual models make different mistakes, their errors can cancel out when you combine their outputs. This chapter introduces several mainstream ensemble strategies: **voting**, **bagging/pasting**, **Random Forests and Extra-Trees**, **boosting (AdaBoost and Gradient Boosting)**, and **stacking**. In practice, ensemble methods are frequently applied toward the end of a project after you already have several reasonably strong predictors, then you combine them to push performance further.

Voting Classifiers

Voting is the simplest ensemble approach: train several diverse classifiers and let them "vote" on the final prediction. In **hard voting**, the predicted class is simply the majority class among the models' predicted labels. This can outperform the best single classifier, even when each model is only moderately accurate, because majority voting reduces the chance that a single model's mistake dominates the final output—this is analogous to the law of large numbers in repeated coin tosses. However, voting only works well when the models are reasonably independent (i.e., not making highly correlated errors). Diversity is therefore essential; mixing different algorithms (e.g., Logistic Regression, SVM, Random Forest) is a common way to achieve that. In **soft voting**, instead of voting on labels, each model contributes class probabilities; the ensemble averages these probabilities and picks the class with the highest average probability. Soft voting often performs better than hard voting because it implicitly gives more weight to models that are more confident in their predictions (assuming their probability estimates are calibrated).

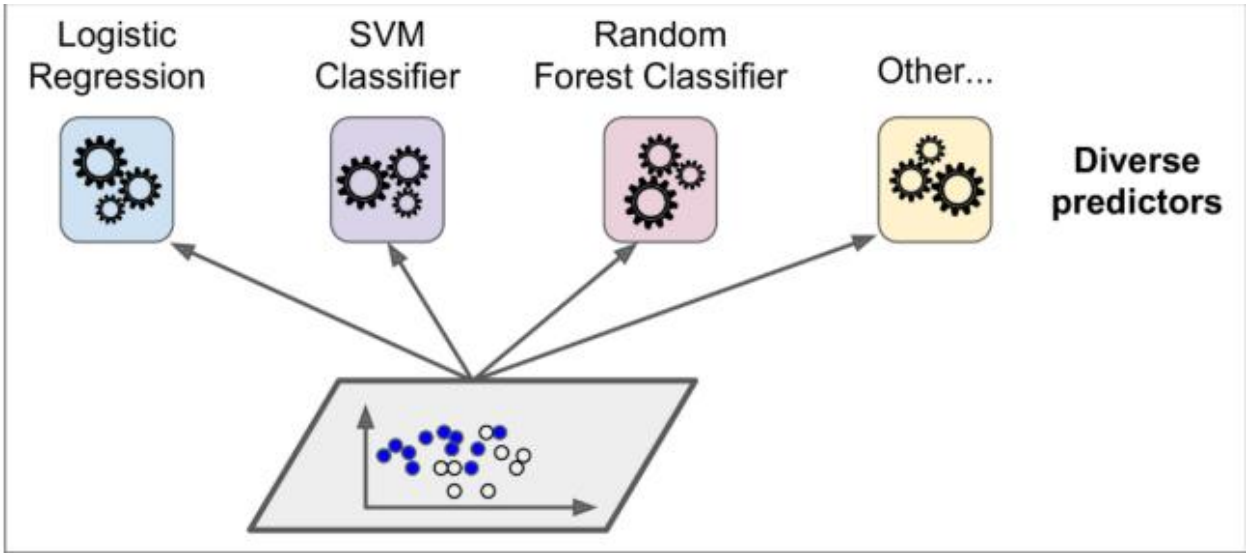

Figure 7-1. Training diverse classifiers

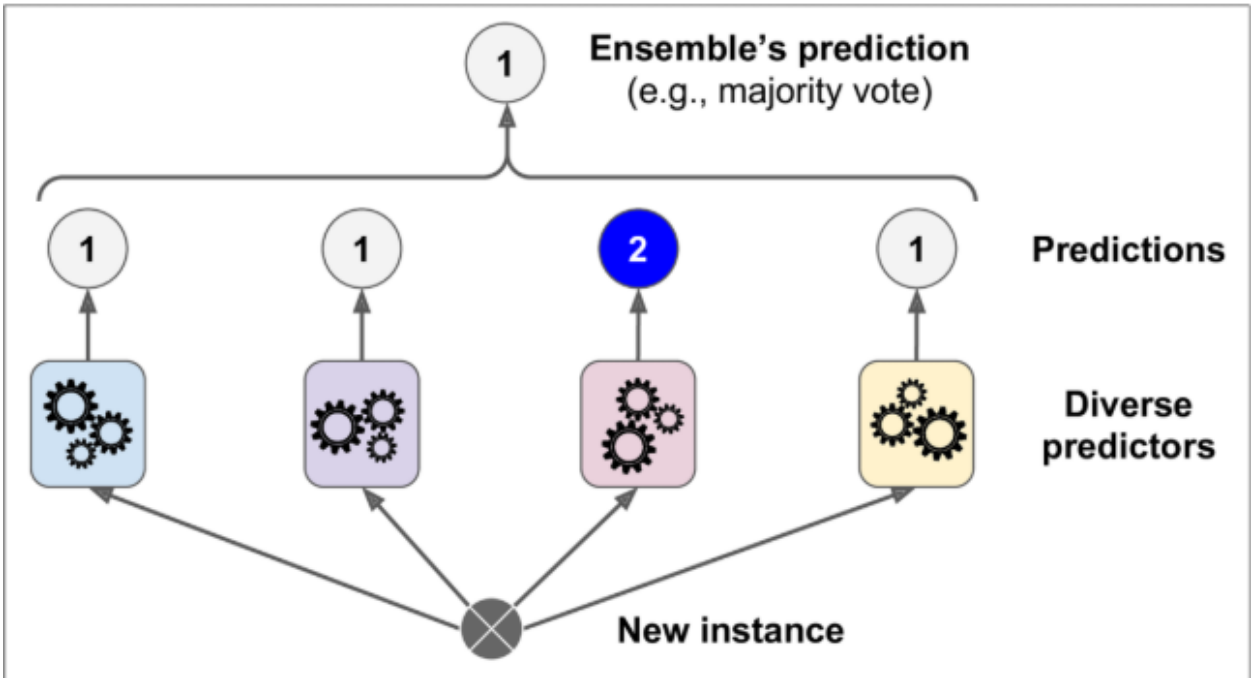

Figure 7-2. Hard voting classifier predictions

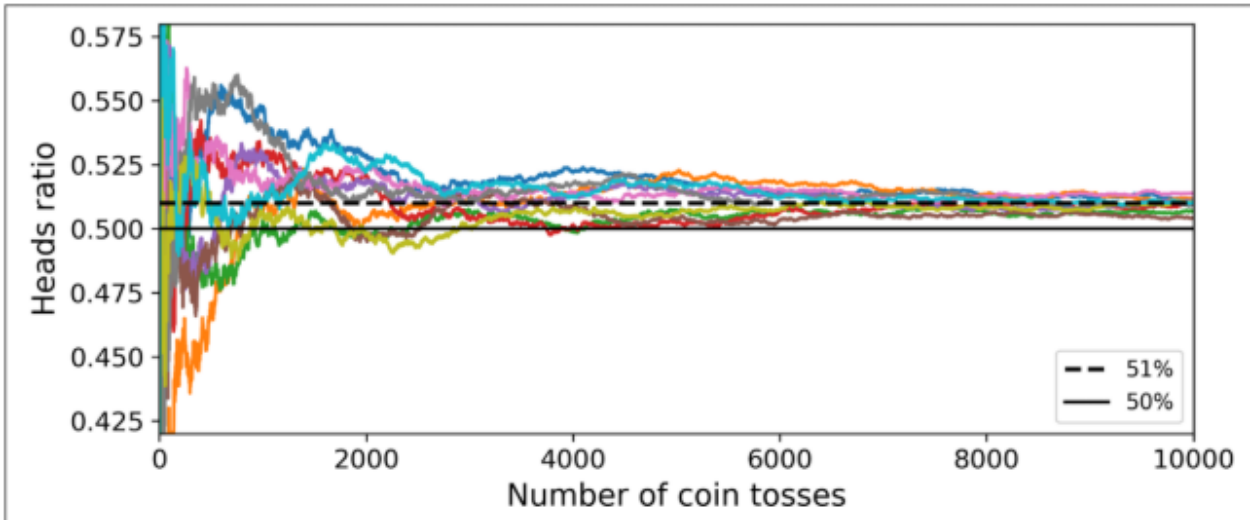

Figure 7-3. The law of large numbers

Bagging and Pasting

Bagging and pasting are ensemble techniques that create diversity by training the same algorithm multiple times on different random subsets of the training data. **Pasting** samples subsets *without replacement*, while **bagging** samples *with replacement* (bootstrapping), meaning a training instance can appear multiple times in the same subset. After training, the ensemble aggregates predictions, typically using majority vote for classification and averaging for regression. Although each individual predictor trained on a subset tends to have higher bias than one trained on the full dataset, the aggregation effect significantly reduces variance; the ensemble commonly

ends up with similar bias but noticeably lower variance than a single model, improving generalization. Bagging is often preferred because sampling with replacement increases subset diversity and reduces correlation between predictors, which tends to improve the final result despite slightly higher bias.

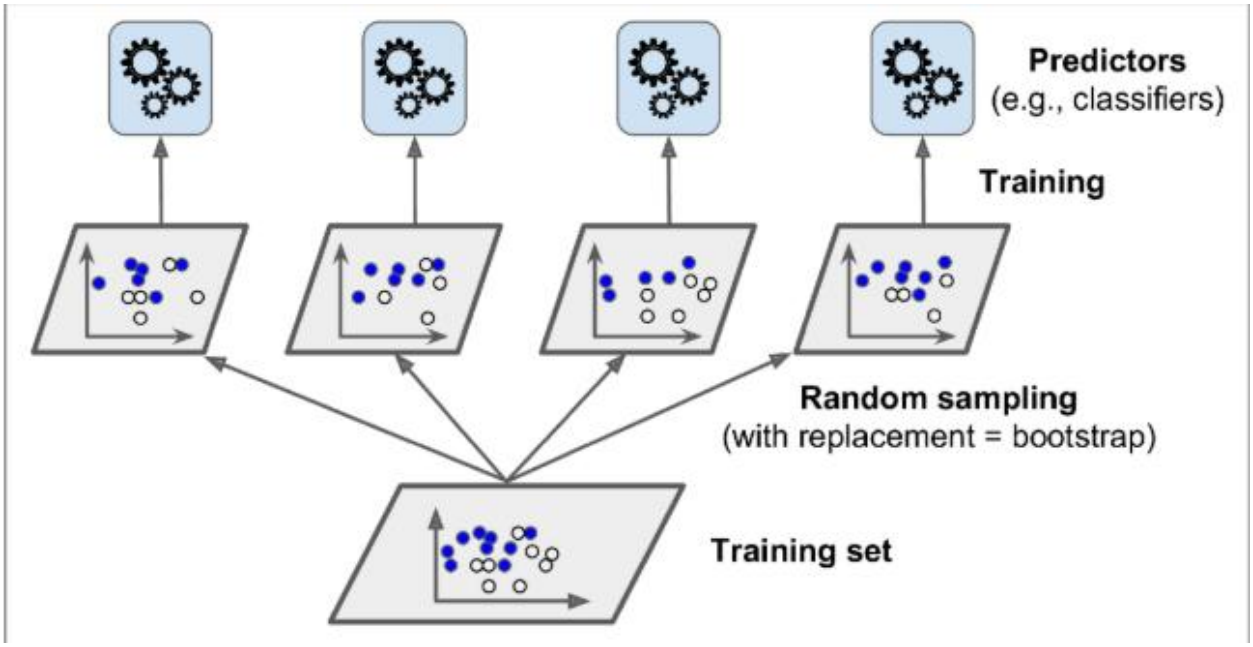

Figure 7-4. Bagging and pasting involves training several predictors on different random samples of the training set

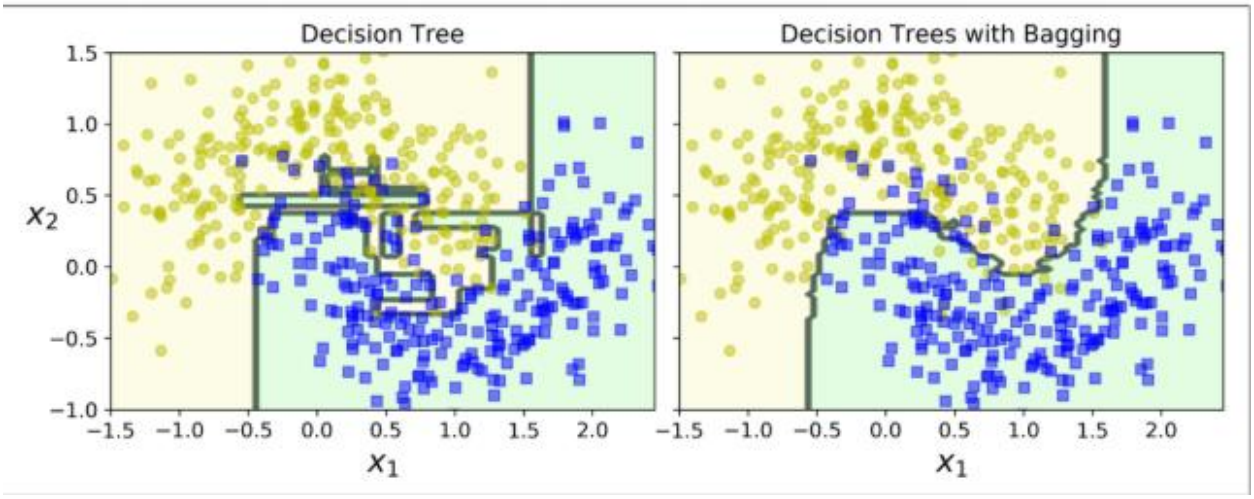

Figure 7-5. A single Decision Tree (left) versus a bagging ensemble of 500 trees (right)

Out-of-Bag Evaluation

A practical bonus of bagging is **out-of-bag (OOB) evaluation**. Because bootstrapping leaves out roughly 37% of training instances for any given predictor, those excluded instances can serve as a built-in validation set for that predictor. By aggregating the OOB performance across all predictors, you can estimate generalization performance without needing a separate validation

split. In Scikit-Learn, enabling oob_score=True provides an automatic OOB score after fitting; you can also access OOB class-probability estimates via oob_decision_function_ when the base estimator supports probability prediction.

Random Patches and Random Subspaces
Bagging can also sample **features** (not just instances). When each predictor is trained on a random subset of features, the ensemble becomes more diverse. Sampling both instances and features is called **Random Patches**, while using all instances but sampling features is called **Random Subspaces**. This is particularly useful for high-dimensional datasets (e.g., images), where feature sampling can reduce correlation between predictors and lower variance further, at the cost of some additional bias.

Random Forests
A **Random Forest** is essentially a bagging ensemble of Decision Trees with an additional layer of randomness during tree construction. Instead of searching for the best split among all features at each node, the algorithm searches among a random subset of features. This forces trees to be less similar to one another, reducing correlation among their errors and improving generalization. In practice, Random Forests are strong "default" models: they are robust, work well with nonlinear patterns, handle mixed feature importance naturally, and require relatively little preprocessing. Conceptually, you can view Random Forests as a specialized, optimized BaggingClassifier/BaggingRegressor for Decision Trees, with feature-subsampling embedded in the split selection procedure.

Extra-Trees
Extra-Trees (Extremely Randomized Trees) push randomness further: in addition to using a random subset of features at each node, they also choose split thresholds more randomly instead of searching for the optimal threshold. This makes training faster (since "best threshold search" is expensive) and often reduces variance even more, though it can increase bias. Whether Extra-Trees beats Random Forests depends on the dataset; the usual approach is to try both and compare via cross-validation.

Feature Importance
Random Forests provide an interpretable byproduct: **feature importance**. Scikit-Learn computes each feature's importance by measuring how much splits using that feature reduce impurity on average across the forest, weighted by the number of samples reaching each split node. The importances are normalized so they sum to 1. This can be used for quick feature selection or to sanity-check whether the model is focusing on meaningful signals.

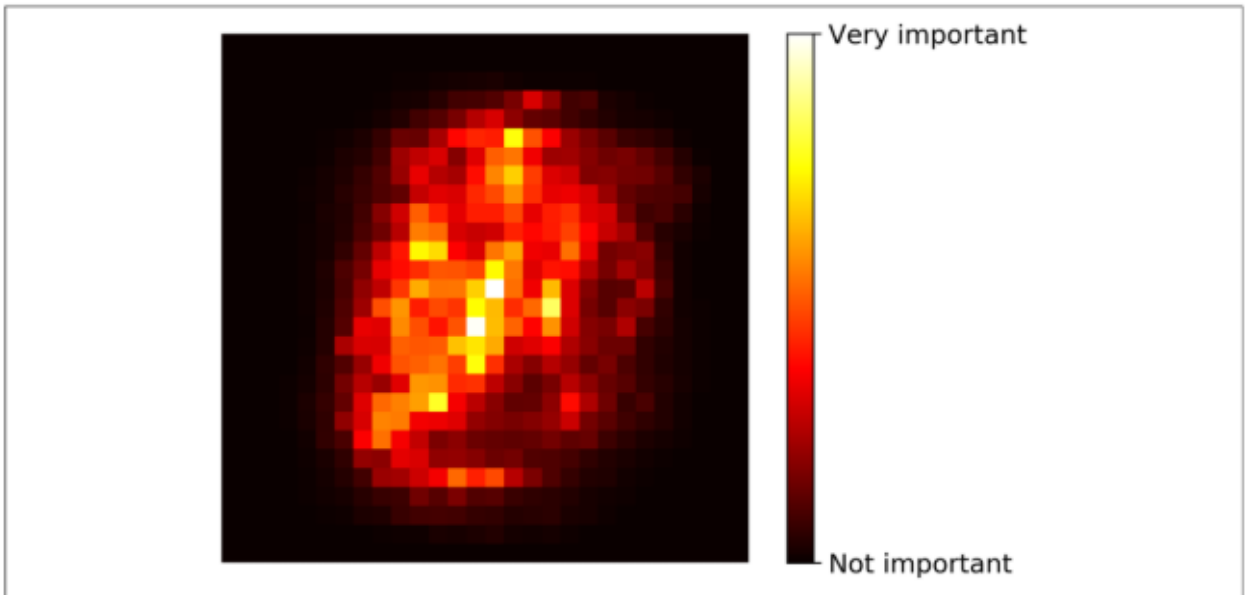

Figure 7-6. MNIST pixel importance (according to a Random Forest classifier)

Boosting

Boosting refers to ensemble methods that turn many weak learners into a strong learner by training predictors **sequentially**, where each new predictor focuses on correcting the previous predictors' mistakes. Unlike bagging (which is parallel-friendly because predictors are independent), boosting is inherently sequential and therefore harder to parallelize. This chapter emphasizes two widely used boosting families: **AdaBoost** and **Gradient Boosting**.

AdaBoost

AdaBoost works by repeatedly reweighting training instances. Initially, all instances have equal weight. After training the first model, instances that were misclassified receive higher weights, forcing the next model to focus more on the "hard" cases. Each predictor receives its own weight in the final ensemble based on its weighted error rate; better predictors receive higher weights, and predictors performing worse than random guessing can receive negative weights. Final predictions are made by a weighted vote across all predictors. In Scikit-Learn, the multiclass implementation is SAMME/SAMME.R, and the default base estimator is usually a **Decision Stump** (a depth-1 tree), which keeps each learner weak but fast, allowing the ensemble to build complexity over many stages. Overfitting can be reduced by shrinking the number of estimators, lowering the learning rate, or increasing regularization of the base estimator.

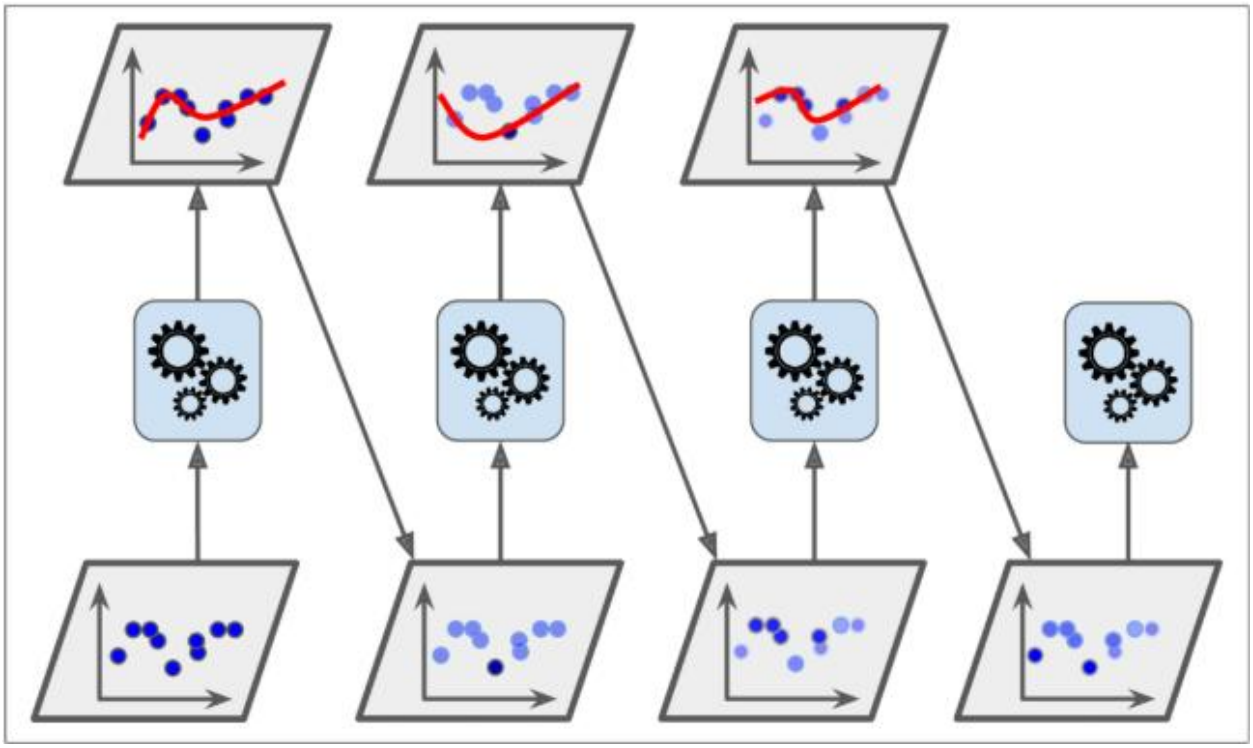

Figure 7-7. AdaBoost sequential training with instance weight updates

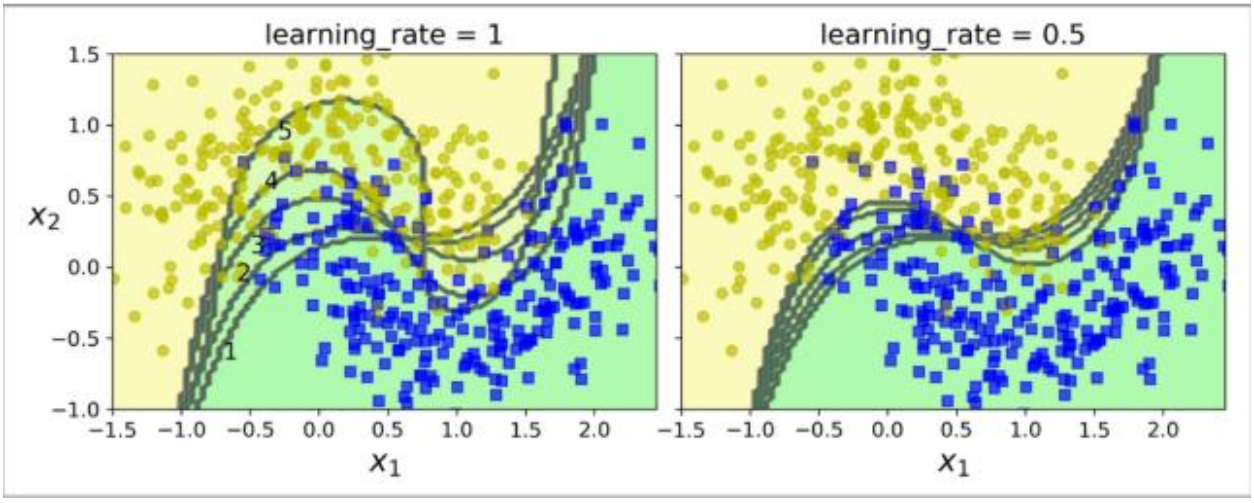

Figure 7-8. Decision boundaries of consecutive predictors

Gradient Boosting

Gradient Boosting also trains predictors sequentially, but instead of changing instance weights, each new predictor is trained to fit the **residual errors** of the ensemble so far. In regression, this is intuitive: train a tree, compute residuals, train the next tree on residuals, and repeat; the final prediction is the sum of predictions from all trees. This is often called **Gradient Boosted Regression Trees (GBRT)**. A central regularization idea here is **shrinkage**, controlled by the learning_rate: smaller learning rates typically require more trees but tend to generalize better. Another key control is the number of trees (n_estimators), which is commonly tuned using **early**

stopping by monitoring validation error as trees are added. Gradient Boosting also supports **subsampling** (training each tree on a random fraction of instances), producing **Stochastic Gradient Boosting**, which can speed up training and reduce variance at the cost of higher bias.

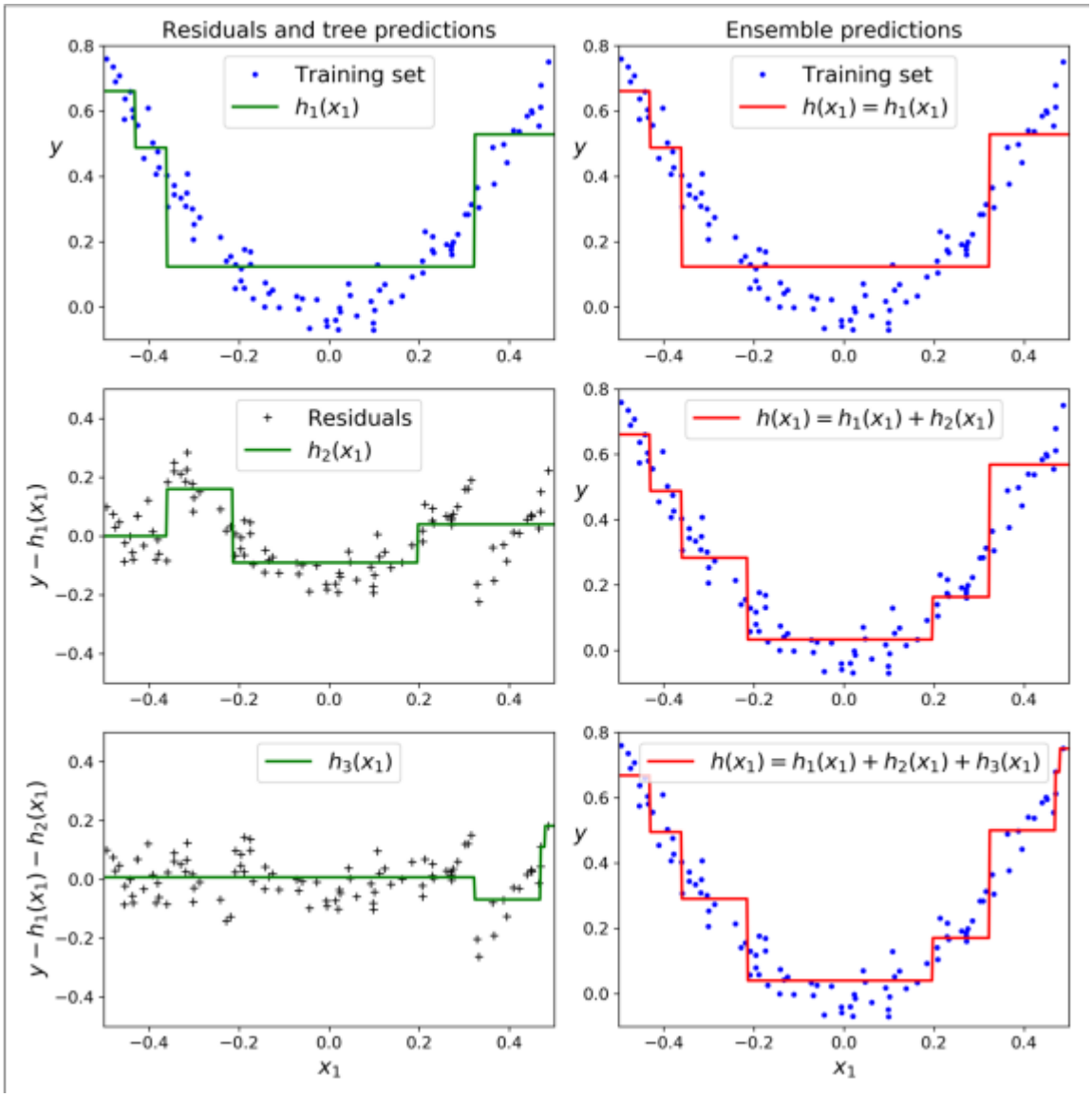

Figure 7-9. Gradient Boosting: sequential residual fitting and resulting ensemble predictions

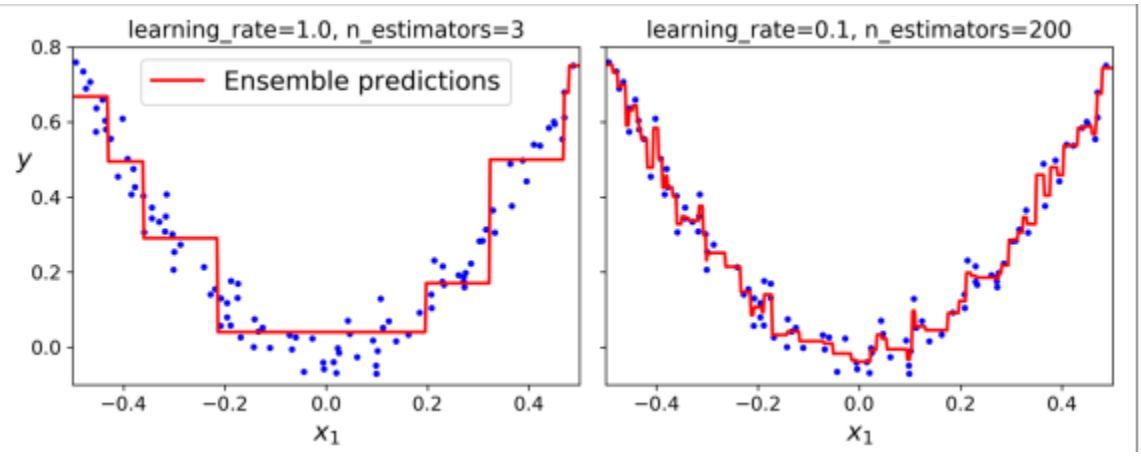

Figure 7-10. GBRT ensembles with not enough predictors (left) and too many (right)

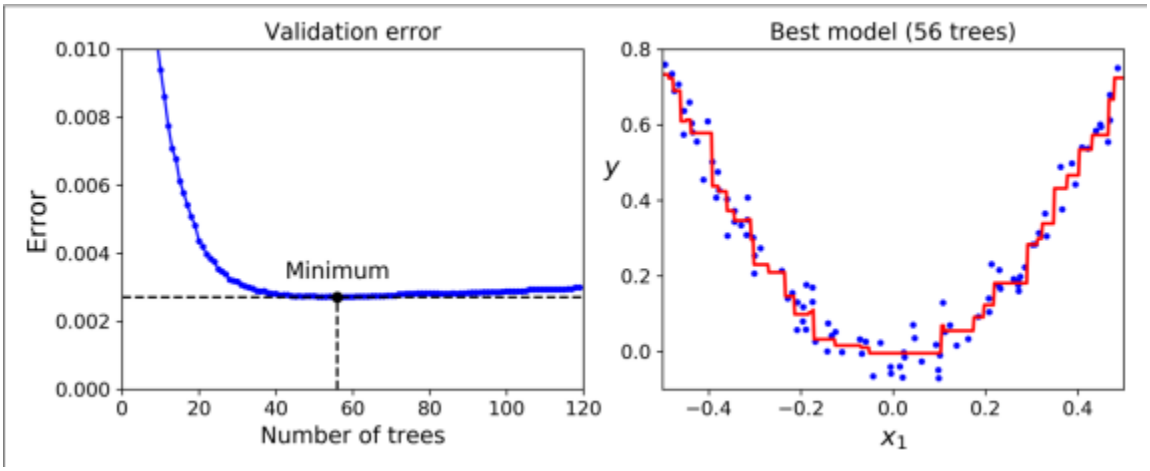

Figure 7-11. Tuning the number of trees using early stopping

Stacking

Stacking (stacked generalization) combines multiple models by training a **blender (meta-model)** that learns how to best merge the base models' predictions. The core idea is to avoid training the blender on predictions that were generated from the same data used to train the base predictors (to prevent leakage). A common approach uses a hold-out set: train base predictors on one subset, generate their predictions on a second subset, then train the blender using those predictions as input features (with the true labels as targets). This can be extended into multi-layer stacking, where each layer produces prediction features for training the next layer. Stacking can be powerful because it allows the blender to learn which base models are more reliable in which regions of the feature space, but it requires careful validation design to avoid optimistic bias.

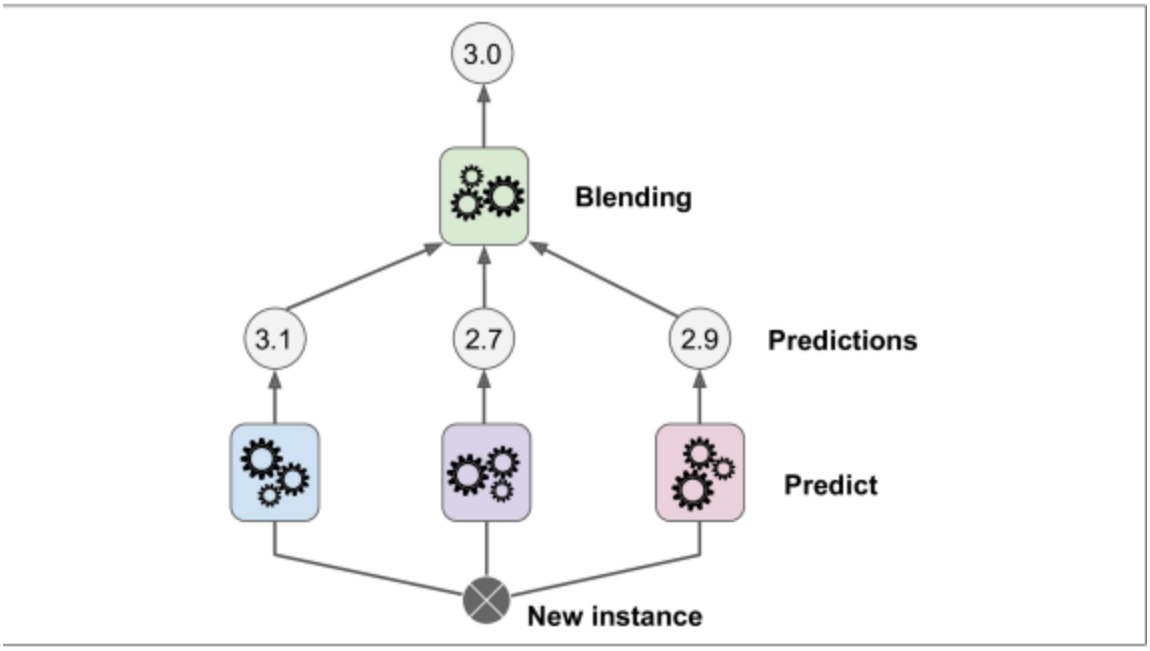

Figure 7-12. Aggregating predictions using a blending predictor

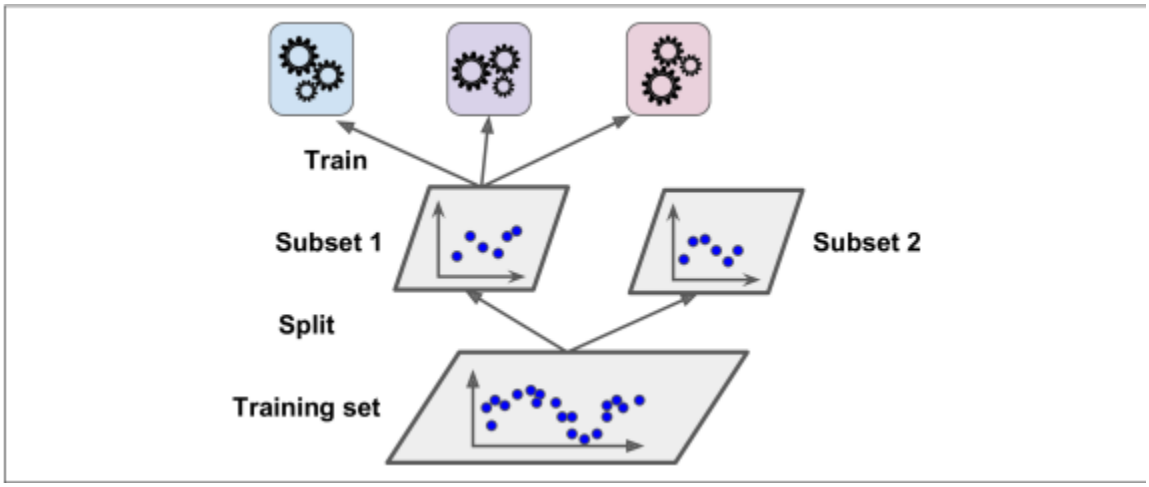

Figure 7-13. Training the first layer

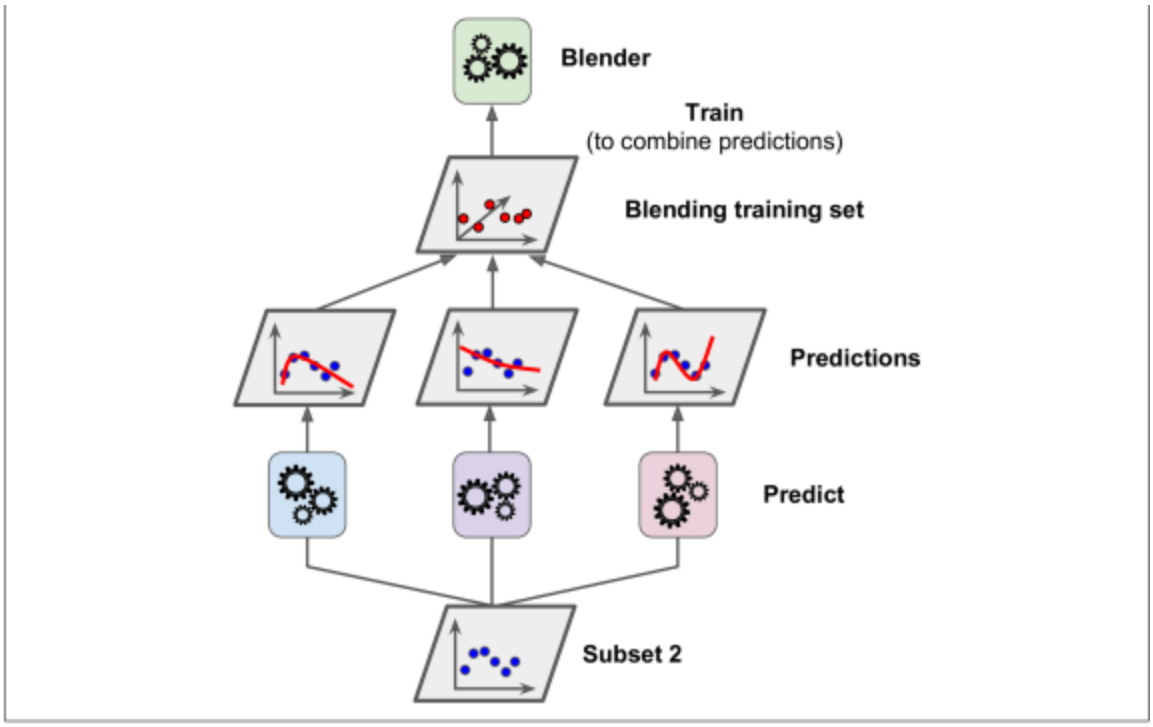

Figure 7-14. Training the blender

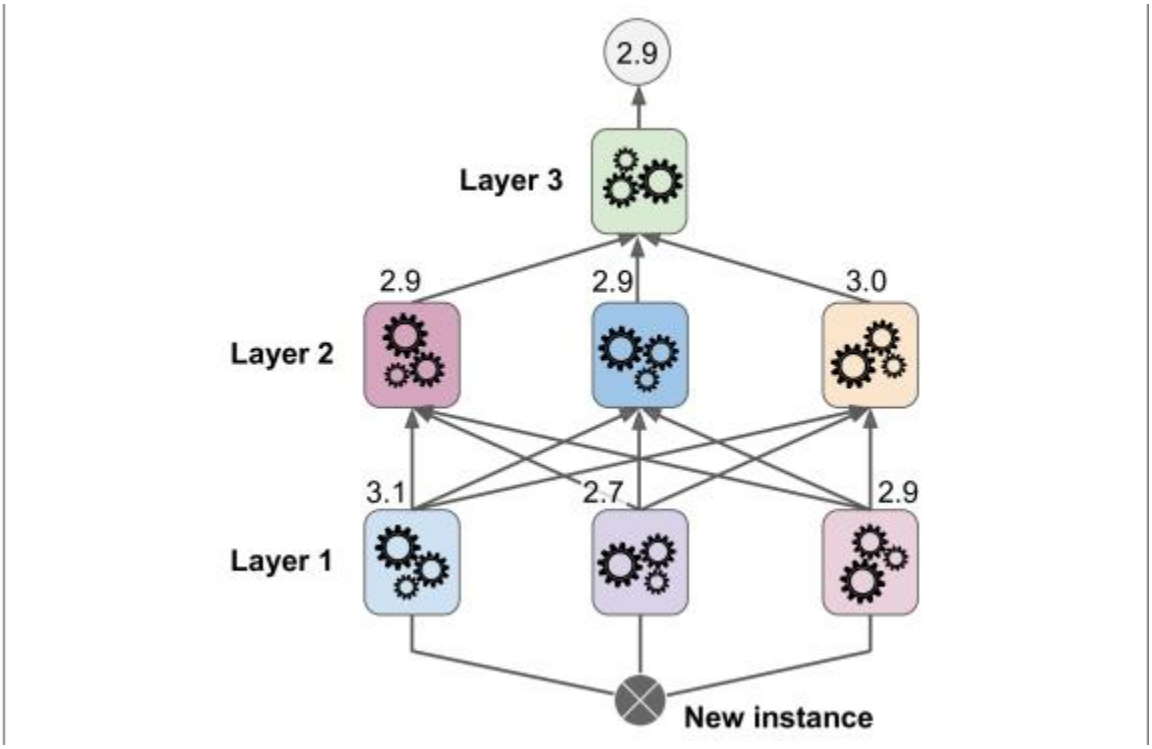

Figure 7-15. Predictions in a multilayer stacking ensemble