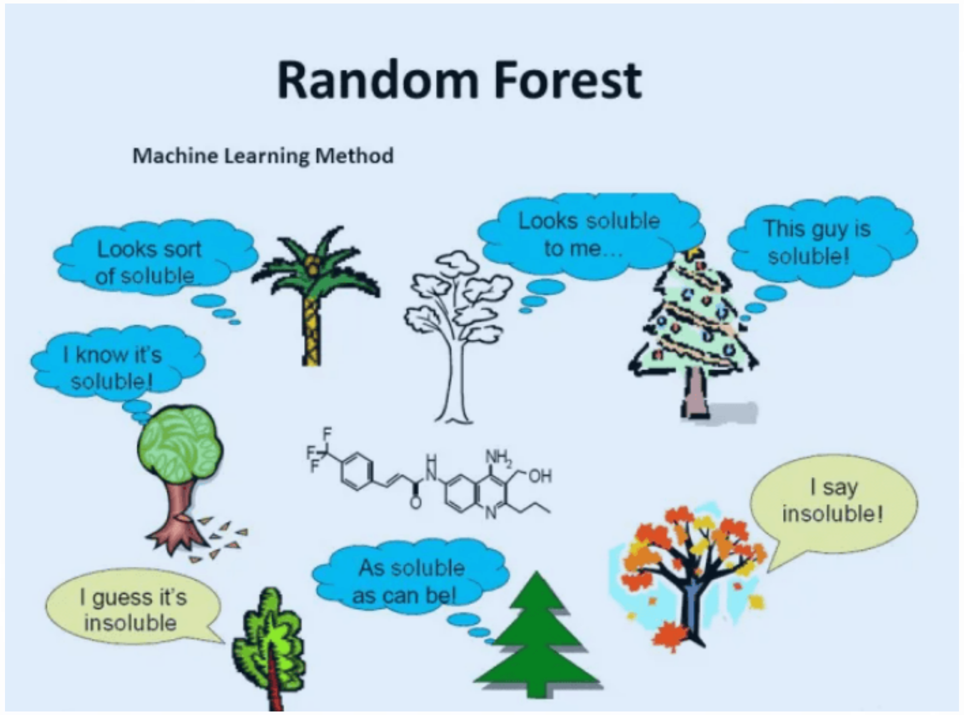
ستاد توسعه فناوری های
هوش مصنوعی و رباتیک

دانشگاه شاهد

AI in Biomedical Data

Dr. M.B. Khodabakhshi
Amir Hossein Fouladi
Alireza Javadi

github.com/mbkhodabakhshi/AI_in_BiomedicalData



یادگیری ماشین در زیست پزشکی

Chapter 7. Ensemble Learning and Random Forest

دکتر محمدباقر خدابخشی
mb.khodabakhshi@gmail.com

Preface

- **Wisdom of the crowd**: Suppose you pose a complex question to thousands of random people, then aggregate their answers.
- If you aggregate the predictions of a group of predictors (such as classifiers or regressors), you will often get better predictions than with the best individual predictor.
- A group of predictors is called an **ensemble**; thus, this technique is called Ensemble Learning, and an Ensemble Learning algorithm is called an Ensemble method.

- As an example of an Ensemble method, you can train a group of Decision Tree classifiers, each on a different random subset of the training set. To make predictions, you obtain the predictions of all the individual trees, then predict the class that gets the most votes.

- Such an ensemble of Decision Trees is called a ***Random Forest***, and despite its simplicity, this is one of the most powerful Machine Learning algorithms available today.

- you will often use Ensemble methods near the end of a project, once you have already built a few good predictors, to combine them into an even better predictor.
- In this chapter we will discuss the most popular Ensemble methods, including *bagging*, *boosting*, and *stacking*. We will also explore Random Forests.

Voting Classifiers

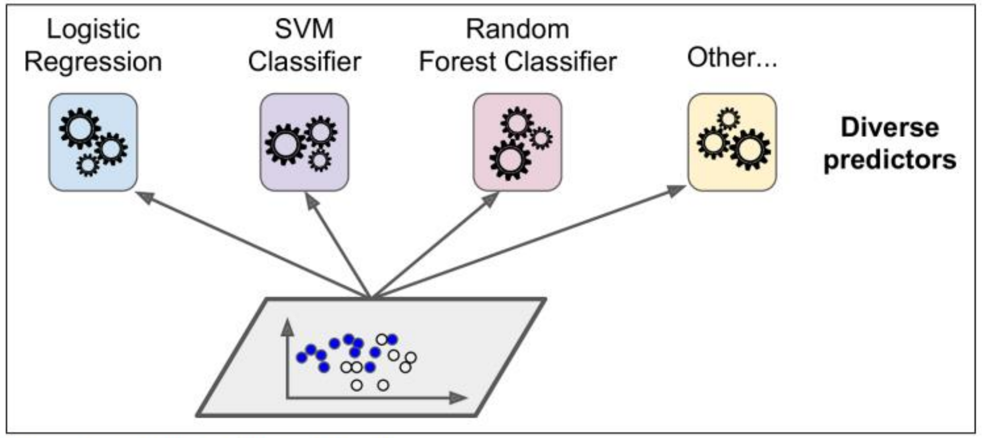


Figure 7-1. Training diverse classifiers

A very simple way to create an even better classifier is to aggregate the predictions of each classifier and predict the class that gets the most votes. This majority-vote classifier is called a *hard voting* classifier

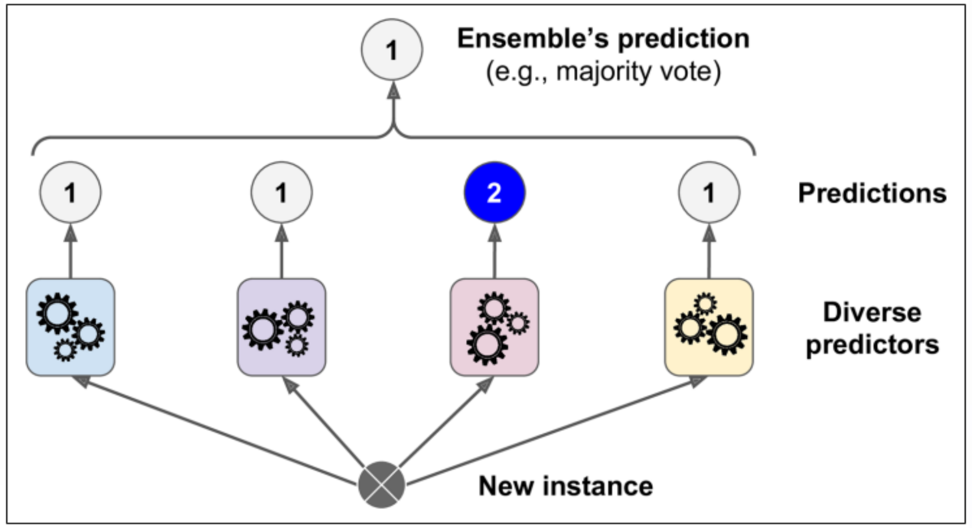


Figure 7-2. Hard voting classifier predictions

- This voting classifier often achieves a higher accuracy than the best classifier in the ensemble. In fact, even if each classifier is a *weak learner* (meaning it does only slightly better than random guessing), the ensemble can still be a *strong learner* (achieving high accuracy), provided there are a sufficient number of weak learners and they are sufficiently diverse.
- Suppose you have a slightly **biased coin** that has a 51% chance of coming up heads and 49% chance of coming up tails. If you toss it 1,000 times, you will generally get more or less 510 heads and 490 tails, and hence a majority of heads. If you do the math, you will find that the probability of obtaining a majority of heads after 1,000 tosses is close to 75%.
- The more you toss the coin, the higher the probability (e.g., with 10,000 tosses, the probability climbs over 97%). This is due to the law of large numbers: as you keep tossing the coin, the ratio of heads gets closer and closer to the probability of heads (51%).

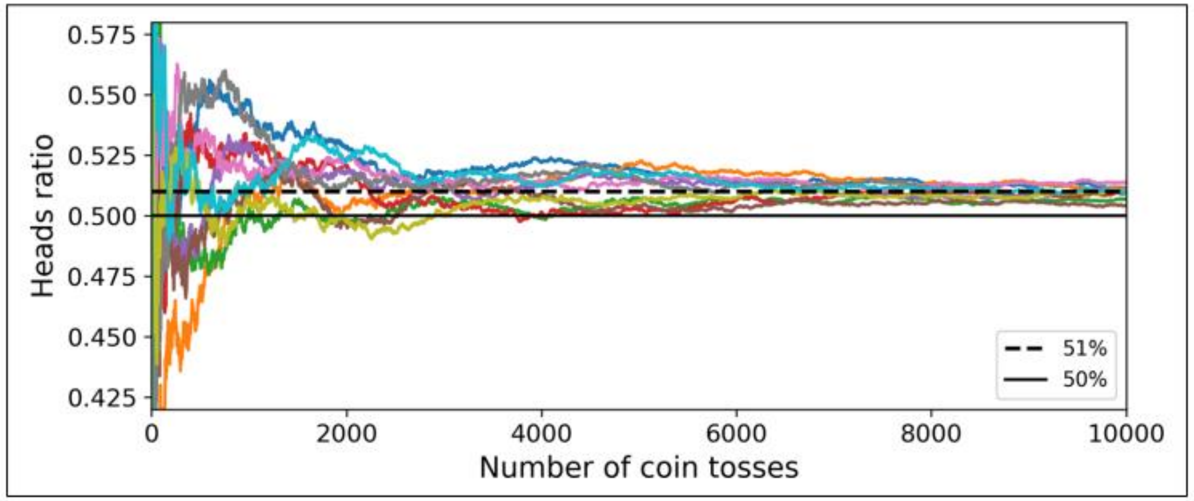


Figure 7-3. The law of large numbers

- Similarly, suppose you build an ensemble containing 1,000 classifiers that are individually correct only 51% of the time (barely better than random guessing).
- If you predict the majority voted class, you can hope for up to 75% accuracy! However, this is only true if all classifiers are perfectly independent, making uncorrelated errors, **which is clearly not the case because they are trained on the same data**.
- They are likely to make the same types of errors, so there will be many majority votes for the wrong class, reducing the ensemble's accuracy.

Ensemble methods work best when the predictors are as independent from one another as possible.
One way to get diverse classifiers is to train them using very different algorithms. This increases the chance that they will make very different types of errors, improving the ensemble's accuracy.

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.ensemble import VotingClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC

log_clf = LogisticRegression()
rnd_clf = RandomForestClassifier()
svm_clf = SVC()

voting_clf = VotingClassifier(
    estimators=[('lr', log_clf), ('rf', rnd_clf), ('svc', svm_clf)],
    voting='hard')
voting_clf.fit(X_train, y_train)
```

Let's look at each classifier's accuracy on the test set:

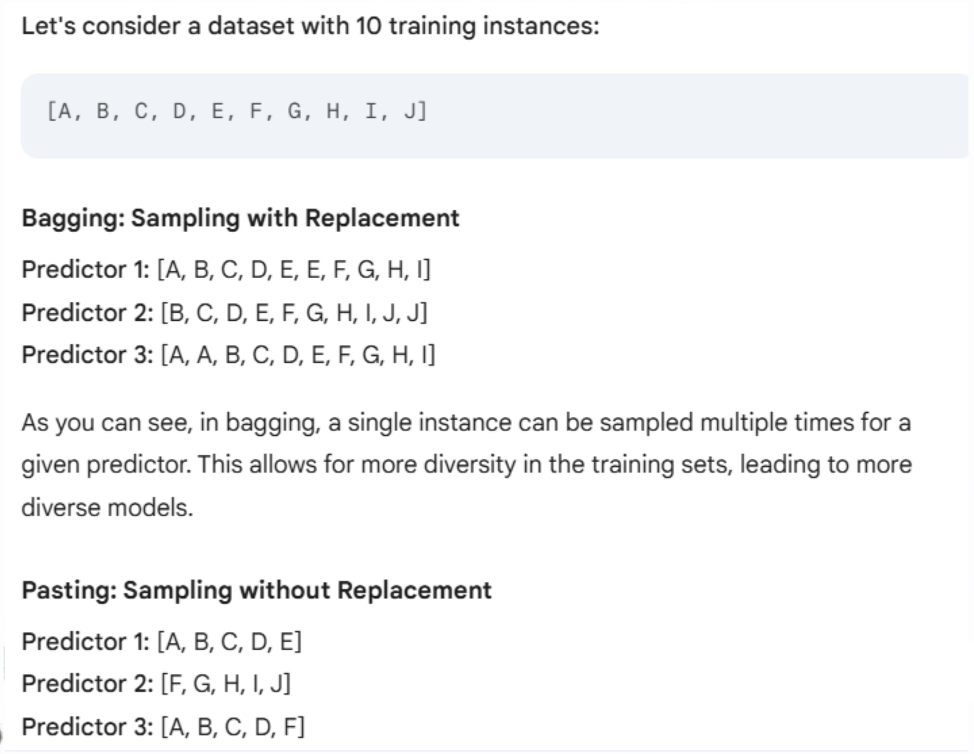
```
>>> from sklearn.metrics import accuracy_score
>>> for clf in (log_clf, rnd_clf, svm_clf, voting_clf):
...     clf.fit(X_train, y_train)
...     y_pred = clf.predict(X_test)
...     print(clf.__class__.__name__, accuracy_score(y_test, y_pred))
...
LogisticRegression 0.864
RandomForestClassifier 0.896
SVC 0.888
VotingClassifier 0.904
```

There you have it! The voting classifier slightly outperforms all the individual classifiers.

- If all classifiers are able to estimate class probabilities (i.e., they all have a `predict_proba()` method), then you can tell Scikit-Learn to predict the class with the highest-class probability, averaged over all the individual classifiers.
- This is called ***soft voting***. It often achieves higher performance than hard voting because it gives more weight to highly confident votes.
- All you need to do is replace `voting="hard"` with `voting="soft"` and ensure that all classifiers can estimate class probabilities.
- This is not the case for the `SVC` class by default, so you need to set its `probability` hyper-parameter to `True` (this will make the `SVC` class use cross-validation to estimate class probabilities, slowing down training, and it will add a `predict_proba()` method).
- If you modify the preceding code to use soft voting, you will find that the voting classi-fier achieves over 91.2% accuracy!

Bagging and Pasting

- Another approach is to use the same training algorithm for every predictor and train them on different random subsets of the training set.
- When sampling is performed *with* replacement, this method is called *bagging* (short for *bootstrap aggregating*).
- When sampling is performed *without* replacement, it is called *pasting*.
- In other words, both bagging and pasting allow training instances to be sampled several times across multiple predictors, but only bagging allows training instances to be sampled several times for the same predictor.



Let's consider a dataset with 10 training instances:

```
[A, B, C, D, E, F, G, H, I, J]
```

Bagging: Sampling with Replacement

Predictor 1: [A, B, C, D, E, E, F, G, H, I]

Predictor 2: [B, C, D, E, F, G, H, I, J, J]

Predictor 3: [A, A, B, C, D, E, F, G, H, I]

As you can see, in bagging, a single instance can be sampled multiple times for a given predictor. This allows for more diversity in the training sets, leading to more diverse models.

Pasting: Sampling without Replacement

Predictor 1: [A, B, C, D, E]

Predictor 2: [F, G, H, I, J]

Predictor 3: [A, B, C, D, F]

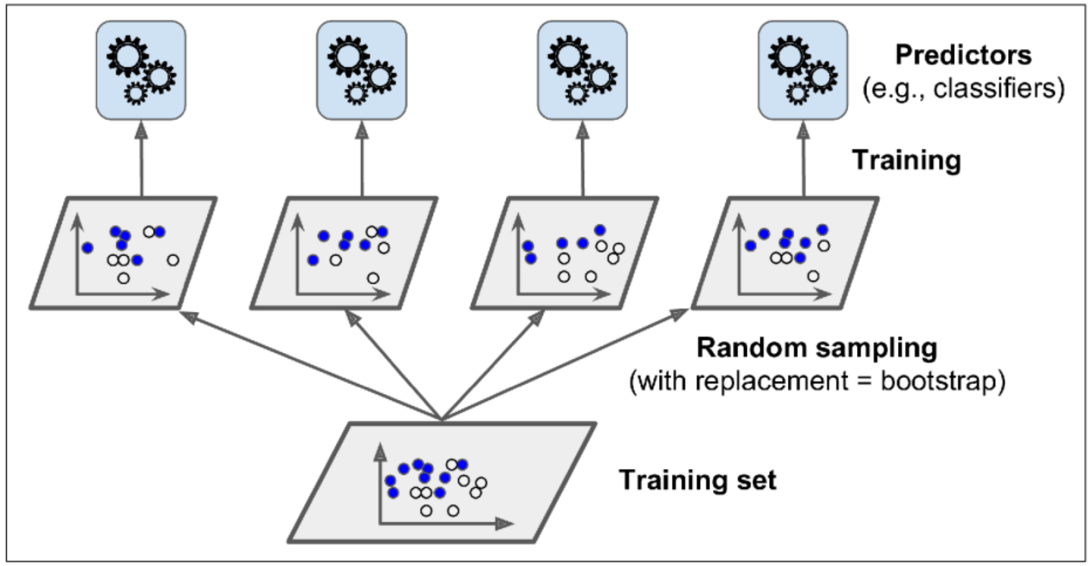


Figure 7-4. Bagging and pasting involves training several predictors on different random samples of the training set

Bagging and Pasting in Scikit-Learn

- ❖ Scikit-Learn offers a simple API for both bagging and pasting with the BaggingClassifier class (or BaggingRegressor for regression).
- ❖ The following code trains an ensemble of 500 Decision Tree classifiers: each is trained on 100 training instances randomly sampled from the training set with replacement (this is an example of bagging, but if you want to use pasting instead, just set bootstrap=False).
- ❖ The n_jobs parameter tells Scikit-Learn the number of CPU cores to use for training and predictions (-1 tells Scikit-Learn to use all available cores):

```python
from sklearn.ensemble import BaggingClassifier
from sklearn.tree import DecisionTreeClassifier

bag_clf = BaggingClassifier(
    DecisionTreeClassifier(), n_estimators=500,
    max_samples=100, bootstrap=True, n_jobs=-1)
bag_clf.fit(X_train, y_train)
y_pred = bag_clf.predict(X_test)
```

The BaggingClassifier automatically performs soft voting instead of hard voting if the base classifier can estimate class probabilities (i.e., if it has a predict_proba() method), which is the case with Decision Tree classifiers.

- Figure 7-5 compares the decision boundary of a single Decision Tree with the decision boundary of a bagging ensemble of 500 trees (from the preceding code), both trained on the moons dataset.
- As you can see, the ensemble's predictions will likely generalize much better than the single Decision Tree's predictions: the ensemble has a comparable bias but a smaller variance (it makes roughly the same number of errors on the training set, but the decision boundary is less irregular).

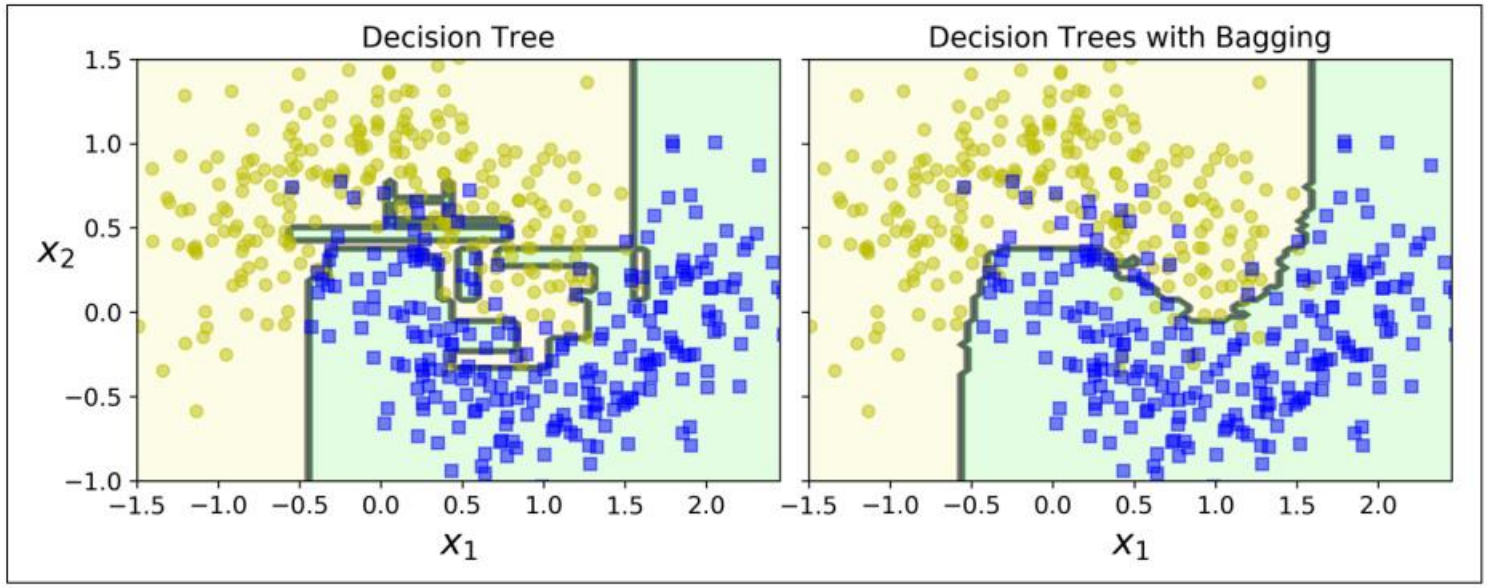


Figure 7-5. A single Decision Tree (left) versus a bagging ensemble of 500 trees (right)

Out-of-Bag Evaluation

- With bagging, some instances may be sampled several times for any given predictor, while others may not be sampled at all.
- By default a `BaggingClassifier` samples m training instances with replacement (`bootstrap=True`), where m is the size of the training set. This means that only about 63% of the training instances are sampled on average for each predictor.
- The remaining 37% of the training instances that are not sampled are called *out-of-bag* (oob) instances. Note that they are not the same 37% for all predictors.
- Since a predictor never sees the oob instances during training, it can be evaluated on these instances, without the need for a separate validation set.
- You can evaluate the ensemble itself by averaging out the oob evaluations of each predictor.
- In Scikit-Learn, you can set `oob_score=True` when creating a `BaggingClassifier` to request an automatic oob evaluation after training.
- The following code demonstrates this. The resulting evaluation score is available through the `oob_score_` variable:

```
>>> bag_clf = BaggingClassifier(
...     DecisionTreeClassifier(), n_estimators=500,
...     bootstrap=True, n_jobs=-1, oob_score=True)
...
>>> bag_clf.fit(X_train, y_train)
>>> bag_clf.oob_score_
0.90133333333333332
```

According to this oob evaluation, this `BaggingClassifier` is likely to achieve about 90.1% accuracy on the test set. Let's verify this:

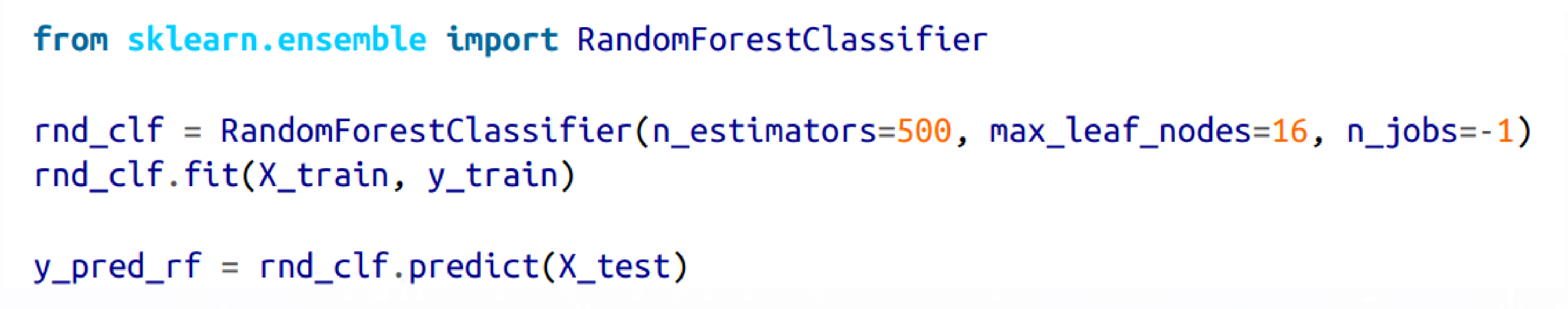
```
>>> from sklearn.metrics import accuracy_score
>>> y_pred = bag_clf.predict(X_test)
>>> accuracy_score(y_test, y_pred)
0.91200000000000003
```

Random Patches and Random Subspaces

- The `BaggingClassifier` class supports sampling the features as well. Sampling is controlled by two hyperparameters: `max_features` and `bootstrap_features`.
- They work the same way as `max_samples` and `bootstrap`, but for feature sampling instead of instance sampling. Thus, each predictor will be trained on a random subset of the input features.
- This technique is particularly useful when you are dealing with high-dimensional inputs (such as images).
- Sampling both training instances and features is called the *Random Patches* method.
- Keeping all training instances (by setting `bootstrap=False` and `max_samples=1.0`) but sampling features (by setting `bootstrap_features` to `True` and/or `max_features` to a value smaller than `1.0`) is called the *Random Subspaces* method.
- Sampling features results in even more predictor diversity, trading a bit more bias for a lower variance.

Random Forests

- As we have discussed, a Random Forest is an ensemble of Decision Trees, generally trained via the bagging method (or sometimes pasting), typically with `max_samples` set to the size of the training set.
- Instead of building a `BaggingClassifier` and passing it a `DecisionTreeClassifier`, you can instead use the `RandomForestClassifier` class, which is more convenient and optimized for Decision Trees.
- (similarly, there is a `RandomForestRegressor` class for regression tasks).



```python
from sklearn.ensemble import RandomForestClassifier

rnd_clf = RandomForestClassifier(n_estimators=500, max_leaf_nodes=16, n_jobs=-1)
rnd_clf.fit(X_train, y_train)

y_pred_rf = rnd_clf.predict(X_test)
```

With a few exceptions, a `RandomForestClassifier` has all the hyperparameters of a `DecisionTreeClassifier` (to control how trees are grown), plus all the hyperparameters of a `BaggingClassifier` to control the ensemble itself.

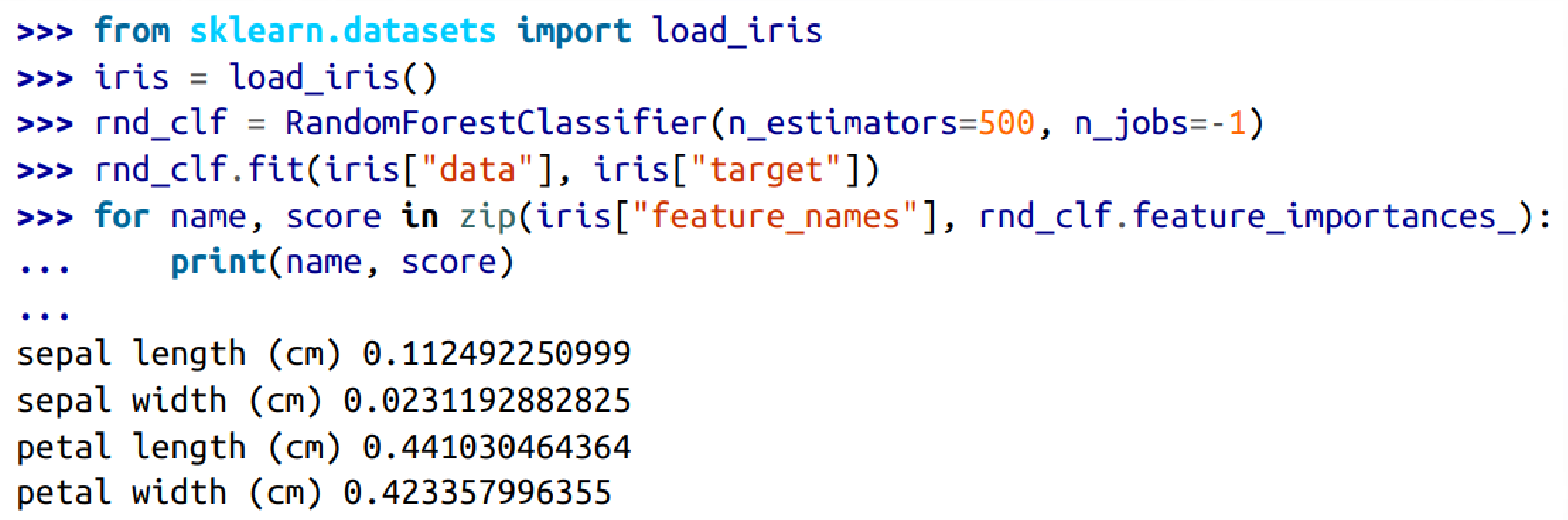
The Random Forest algorithm introduces extra randomness when growing trees; instead of searching for the very best feature when splitting a node (see Chapter 6), it searches for the best feature among a random subset of features.

The algorithm results in greater tree diversity, which (again) trades a higher bias for a lower variance, generally yielding an overall better model. The following `BaggingClassifier` is roughly equivalent to the previous `RandomForestClassifier`:

```
bag_clf = BaggingClassifier(
    DecisionTreeClassifier(splitter="random", max_leaf_nodes=16),
    n_estimators=500, max_samples=1.0, bootstrap=True, n_jobs=-1)
```

Feature Importance

Scikit-Learn measures a feature’s importance by looking at how much the tree nodes that use that feature reduce impurity on average (across all trees in the forest).
More precisely, it is a weighted average, where each node’s weight is equal to the number of training samples that are associated with it



```
>>> from sklearn.datasets import load_iris
>>> iris = load_iris()
>>> rnd_clf = RandomForestClassifier(n_estimators=500, n_jobs=-1)
>>> rnd_clf.fit(iris["data"], iris["target"])
>>> for name, score in zip(iris["feature_names"], rnd_clf.feature_importances_):
...     print(name, score)
...
sepal length (cm) 0.112492250999
sepal width (cm) 0.0231192882825
petal length (cm) 0.441030464364
petal width (cm) 0.423357996355
```

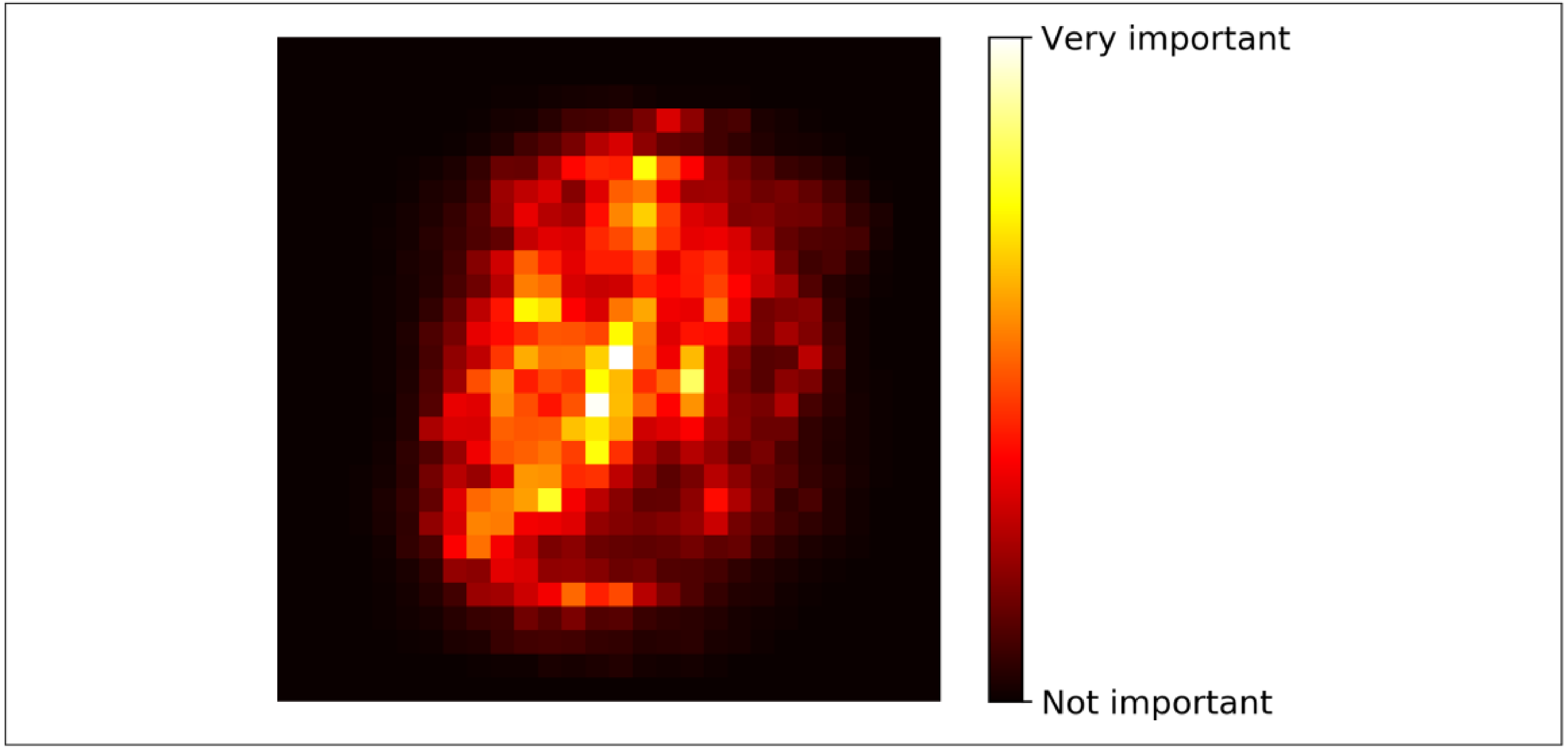


Figure 7-6. MNIST pixel importance (according to a Random Forest classifier)

Boosting

Boosting refers to any Ensemble method that can combine several weak learners into a strong learner.

The general idea of most boosting methods is to train predictors sequentially, each trying to correct its predecessor. There are many boosting methods available, but by far the most popular are:

- *AdaBoost* (short for *Adaptive Boosting*)
- *Gradient Boosting*

AdaBoost

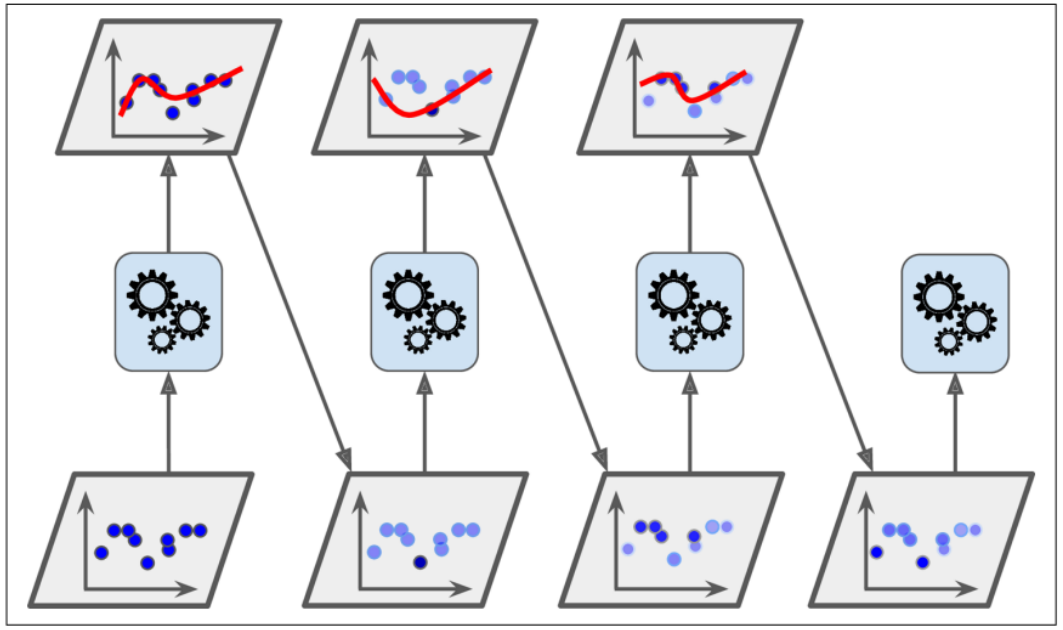
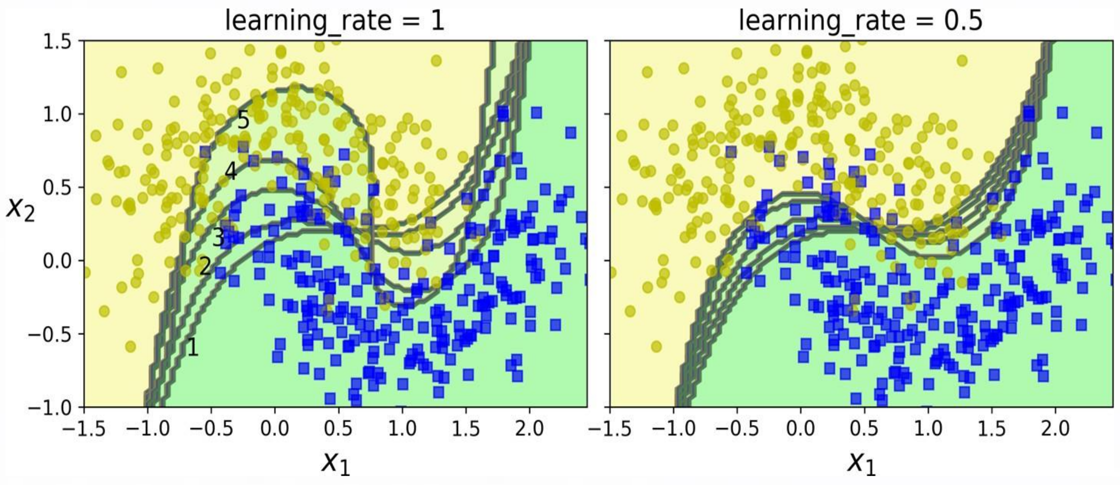


Figure 7-7. AdaBoost sequential training with instance weight updates

AdaBoost

- Figure shows the decision boundaries of five consecutive predictors on the moons dataset. The first classifier gets many instances wrong, so their weights get boosted.
- The second classifier therefore does a better job on these instances, and so on.
- The plot on the right represents the same sequence of predictors, except that the learning rate is halved (i.e., the misclassified instance weights are boosted half as much at every iteration).
- As you can see, this sequential learning technique has some similarities with Gradient Descent, except that instead of tweaking a single predictor's parameters to minimize a cost function, AdaBoost adds predictors to the ensemble, gradually making it better.



AdaBoost algorithm

Each instance weight $w^{(i)}$ is initially set to $1/m$. A first predictor is trained, and its weighted error rate r_1 is computed on the training set; see Equation 7-1

Equation 7-1. Weighted error rate of the j^th predictor

$$r_j = \frac{\sum\limits_{\substack{i=1 \\ \hat{y}_j^{(i)} \neq y^{(i)}}}^{m} w^{(i)}}{\sum\limits_{i=1}^{m} w^{(i)}} \quad \text{where } \hat{y}_j^{(i)} \text{ is the } j^{\text{th}} \text{ predictor's prediction for the } i^{\text{th}} \text{ instance.}$$

The predictor's weight α_j is then computed using Equation 7-2, where η is the learning rate hyperparameter (defaults to 1). The more accurate the predictor is, the higher its weight will be.

Equation 7-2. Predictor weight

$$\alpha_j = \eta \log \frac{1 - r_j}{r_j}$$

Next, the AdaBoost algorithm updates the instance weights, using Equation 7-3, which boosts the weights of the misclassified instances.

Equation 7-3. Weight update rule

$$\text{for } i = 1, 2, \cdots, m$$

$$w^{(i)} \leftarrow \begin{cases} w^{(i)} & \text{if } \hat{y}_j^{(i)} = y^{(i)} \\ w^{(i)} \exp\left(\alpha_j\right) & \text{if } \hat{y}_j^{(i)} \neq y^{(i)} \end{cases}$$

Then all the instance weights are normalized (i.e., divided by $\sum^m w^{\,i}$).

- Finally, a new predictor is trained using the updated weights, and the whole process is repeated. The algorithm stops when the desired number of predictors is reached, or when a perfect predictor is found.
- To make predictions, AdaBoost simply computes the predictions of all the predictors and weighs them using the predictor weights α_j. The predicted class is the one that receives the majority of weighted votes.

$$\hat{y}(\mathbf{x}) = \underset{k}{\operatorname{argmax}} \sum_{\substack{j=1 \\ \hat{y}_j(\mathbf{x}) = k}}^{N} \alpha_j \quad \text{where } N \text{ is the number of predictors.}$$

Scikit-Learn uses a multiclass version of AdaBoost called *SAMME* (which stands for *Stagewise Additive Modeling using a Multiclass Exponential loss function*).
If the predictors can estimate class probabilities (i.e., if they have a `predict_proba()` method), Scikit-Learn can use a variant of SAMME called *SAMME.R* (the *R* stands for “Real”), which relies on class probabilities rather than predictions and generally performs better.

The following code trains an AdaBoost classifier based on 200 *Decision Stumps* using Scikit-Learn’s `AdaBoostClassifier` class (as you might expect, there is also an `AdaBoostRegressor` class). A Decision Stump is a Decision Tree with `max_depth=1`—in other words, a tree composed of a single decision node plus two leaf nodes. This is the default base estimator for the `AdaBoostClassifier` class:

```
from sklearn.ensemble import AdaBoostClassifier

ada_clf = AdaBoostClassifier(
    DecisionTreeClassifier(max_depth=1), n_estimators=200,
    algorithm="SAMME.R", learning_rate=0.5)
ada_clf.fit(X_train, y_train)
```

Gradient Boosting

Instead of tweaking the instance weights at every iteration like AdaBoost does, this method tries to fit the new predictor to the *residual errors* made by the previous predictor.

Let's go through a simple regression example, using Decision Trees as the base predictors (of course, Gradient Boosting also works great with regression tasks). This is called *Gradient Tree Boosting*, or *Gradient Boosted Regression Trees* (GBRT). First, let's fit a `DecisionTreeRegressor` to the training set (for example, a noisy quadratic train- ing set):

```
from sklearn.tree import DecisionTreeRegressor

tree_reg1 = DecisionTreeRegressor(max_depth=2)
tree_reg1.fit(X, y)
```

Next, we'll train a second `DecisionTreeRegressor` on the residual errors made by the first predictor:

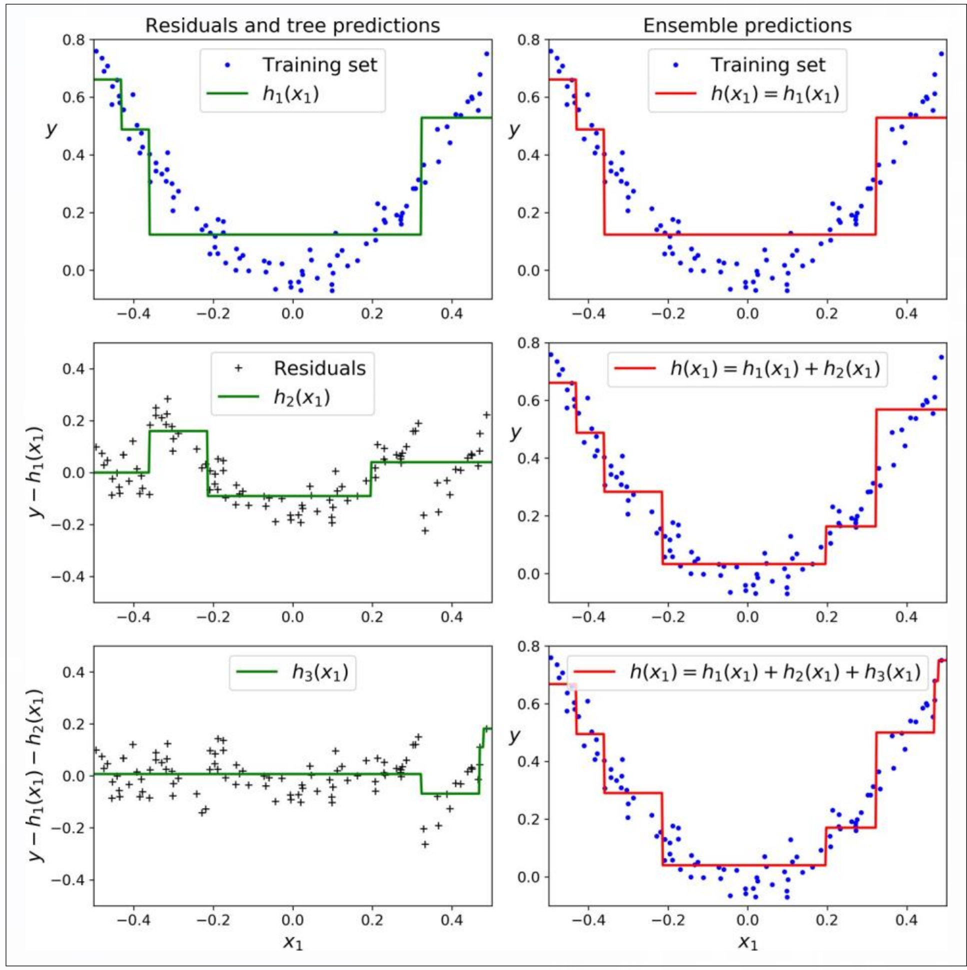
```
y2 = y - tree_reg1.predict(X)
tree_reg2 = DecisionTreeRegressor(max_depth=2)
tree_reg2.fit(X, y2)
```

- Then we train a third regressor on the residual errors made by the second predictor:

```
y3 = y2 - tree_reg2.predict(X)
tree_reg3 = DecisionTreeRegressor(max_depth=2)
tree_reg3.fit(X, y3)
```

- Now we have an ensemble containing three trees.
- It can make predictions on a new instance simply by adding up the predictions of all the trees:

```
y_pred = sum(tree.predict(X_new) for tree in (tree_reg1, tree_reg2, tree_reg3))
```



A simpler way to train GBRT ensembles is to use Scikit-Learn's `GradientBoostingRegressor` class.

Much like the `RandomForestRegressor` class, it has hyperparameters to control the growth of Decision Trees (e.g., `max_depth`, `min_samples_leaf`), as well as hyperparameters to control the ensemble training, such as the number of trees (`n_estimators`). The following code creates the same ensemble as the previous one:

```python
from sklearn.ensemble import GradientBoostingRegressor

gbrt = GradientBoostingRegressor(max_depth=2, n_estimators=3, learning_rate=1.0)
gbrt.fit(X, y)
```

- The `learning_rate` hyperparameter scales the contribution of each tree. If you set it to a low value, such as 0.1, you will need more trees in the ensemble to fit the training set, but the predictions will usually generalize better. This is a regularization technique called *shrinkage*.
- Figure 7-10 shows two GBRT ensembles trained with a low learning rate: the one on the left does not have enough trees to fit the training set, while the one on the right has too many trees and overfits the training set.

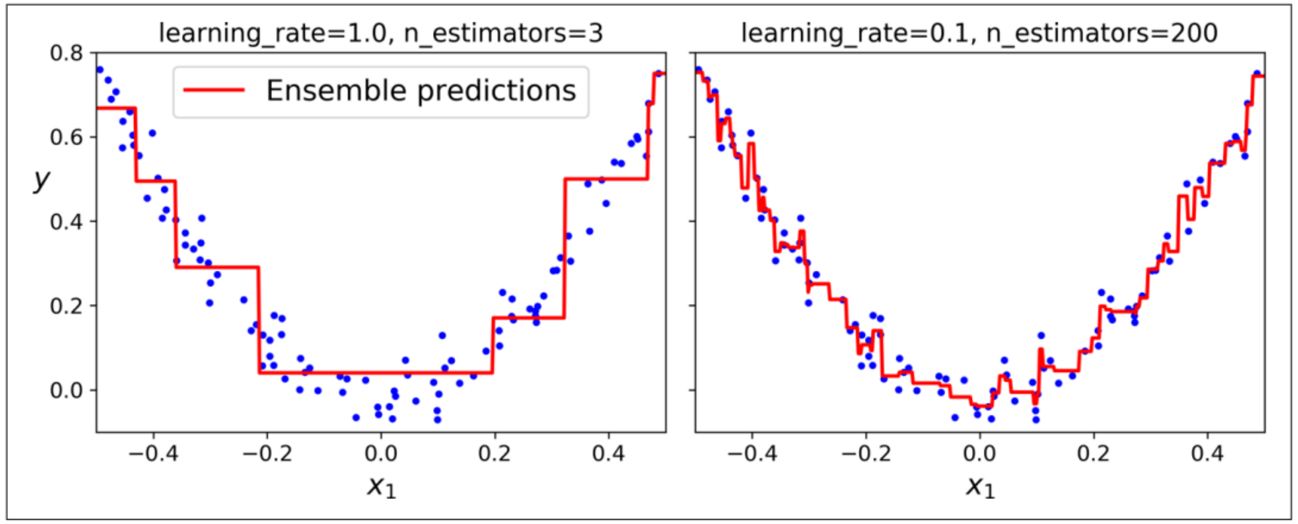
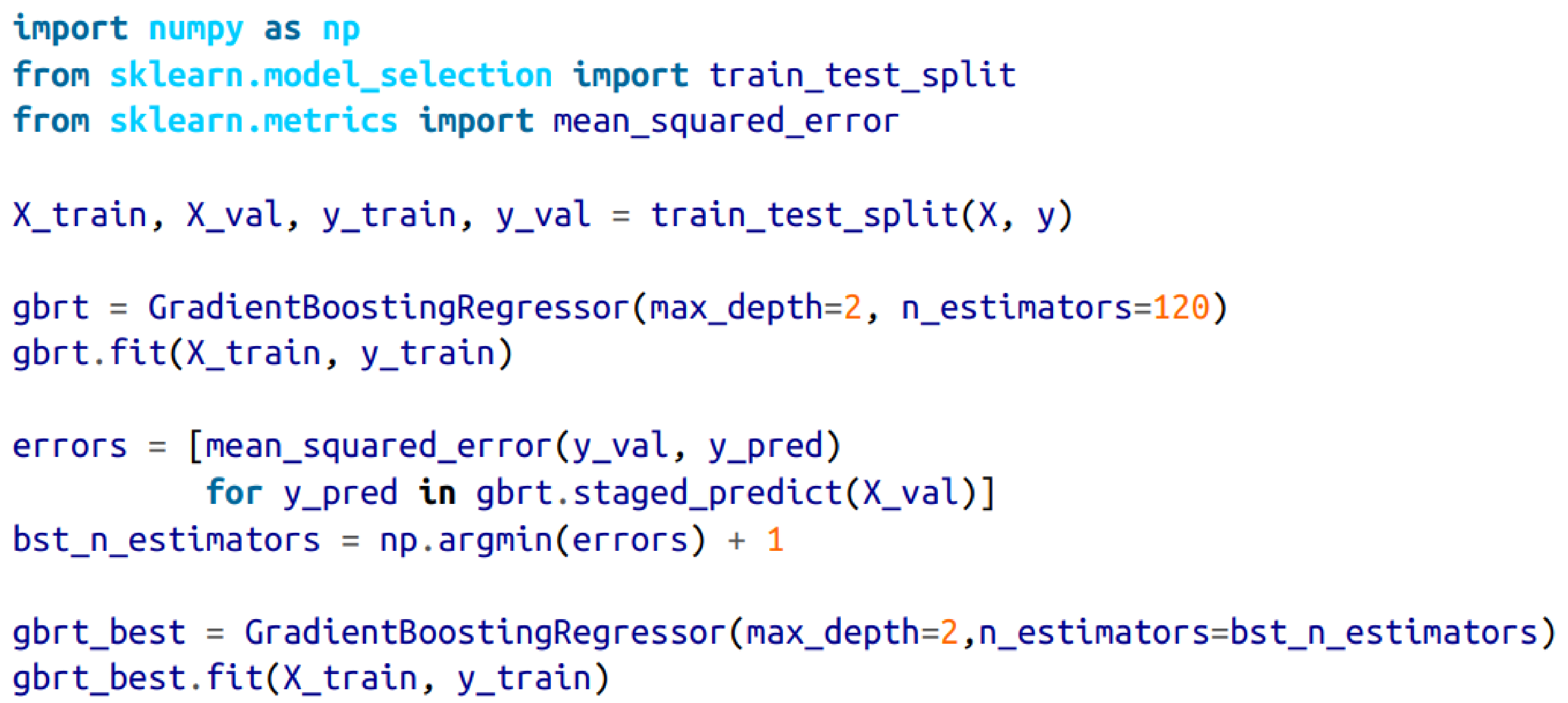


Figure 7-10. GBRT ensembles with not enough predictors (left) and too many (right)

In order to find the optimal number of trees, you can use early stopping (see Chapter 4). A simple way to implement this is to use the staged_predict() method: it returns an iterator over the predictions made by the ensemble at each stage of training (with one tree, two trees, etc.). The following code trains a GBRT ensemble with 120 trees, then measures the validation error at each stage of training to find the optimal number of trees, and finally trains another GBRT ensemble using the optimal number of trees:



```
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error

X_train, X_val, y_train, y_val = train_test_split(X, y)

gbrt = GradientBoostingRegressor(max_depth=2, n_estimators=120)
gbrt.fit(X_train, y_train)

errors = [mean_squared_error(y_val, y_pred)
          for y_pred in gbrt.staged_predict(X_val)]
bst_n_estimators = np.argmin(errors) + 1

gbrt_best = GradientBoostingRegressor(max_depth=2,n_estimators=bst_n_estimators)
gbrt_best.fit(X_train, y_train)
```

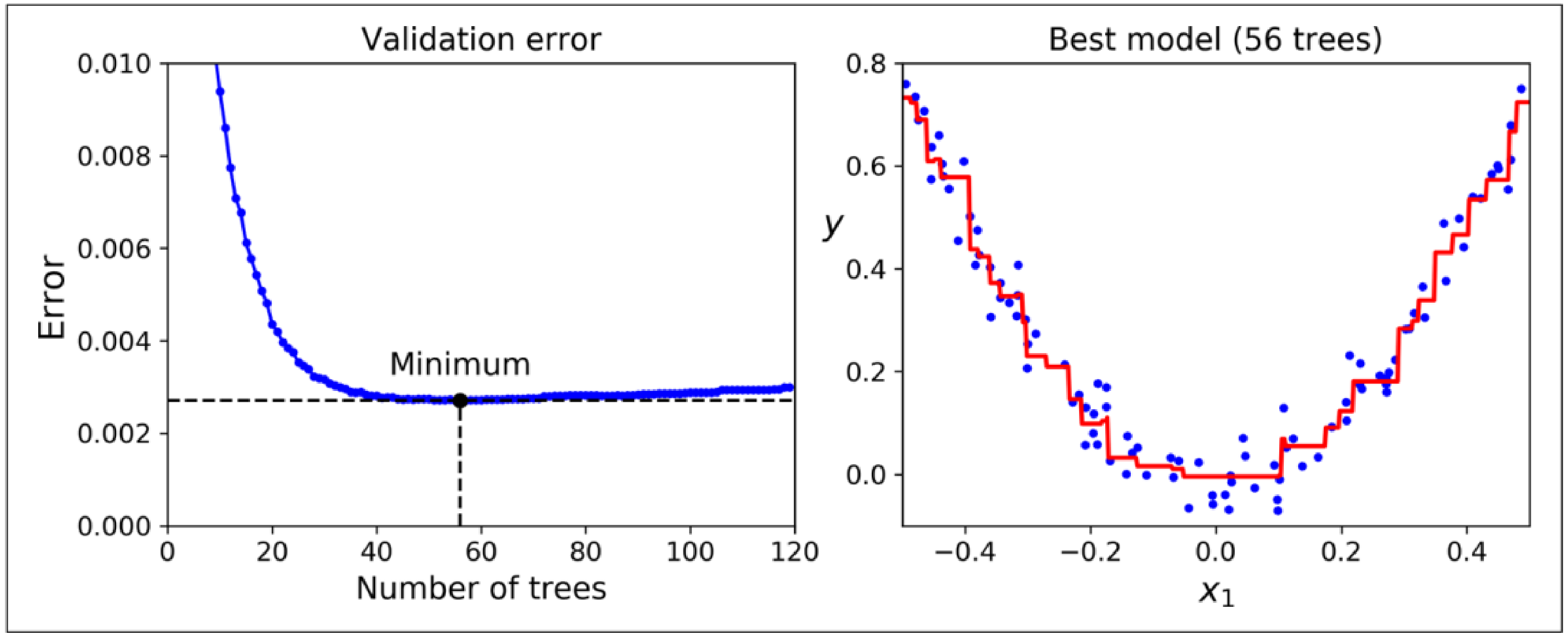


Figure 7-11. Tuning the number of trees using early stopping

ستاد توسعه فناوری های
هوش مصنوعی و رباتیک

XGBoost

It is worth noting that an optimized implementation of Gradient Boosting is available in the popular Python library XGBoost, which stands for Extreme Gradient Boosting.
This package was initially developed by Tianqi Chen as part of the Distributed (Deep) Machine Learning Community (DMLC), and it aims to be extremely fast, scalable, and portable.
In fact, XGBoost is often an important component of the winning entries in ML competitions. XGBoost's API is quite similar to Scikit-Learn's:

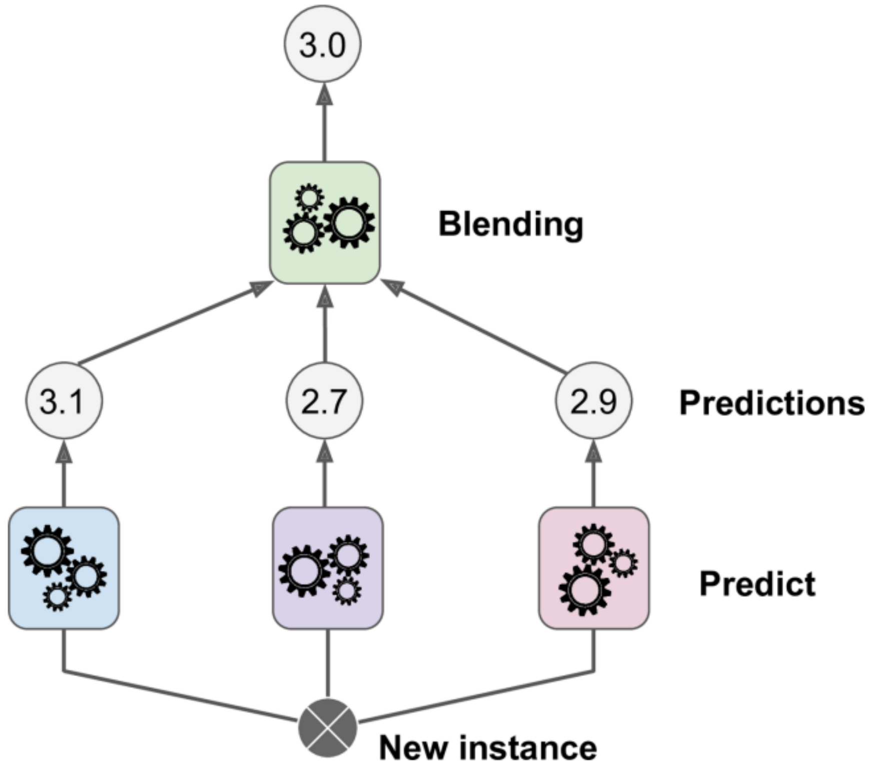
```python
import xgboost

xgb_reg = xgboost.XGBRegressor()
xgb_reg.fit(X_train, y_train)
y_pred = xgb_reg.predict(X_val)
```

XGBoost also offers several nice features, such as automatically taking care of early stopping:

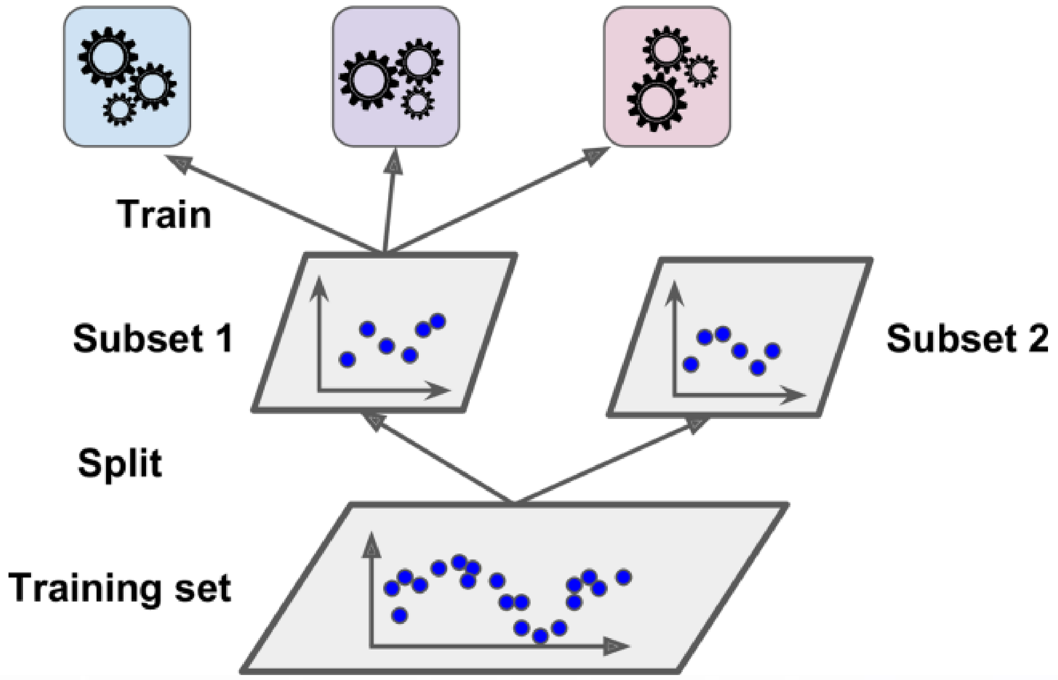
```python
xgb_reg.fit(X_train, y_train,
            eval_set=[(X_val, y_val)], early_stopping_rounds=2)
y_pred = xgb_reg.predict(X_val)
```

Stacking

- ❖ Instead of using trivial functions (such as hard voting) to aggregate the predictions of all predictors in an ensemble, why don't we train a model to perform this aggregation?
- ❖ Figure 7-12 shows such an ensemble performing a regression task on a new instance.
- ❖ Each of the bottom three predictors predicts a different value (3.1, 2.7, and 2.9), and then the final predictor (called a *blender*, or a *meta learner*) takes these predictions as inputs and makes the final prediction (3.0).



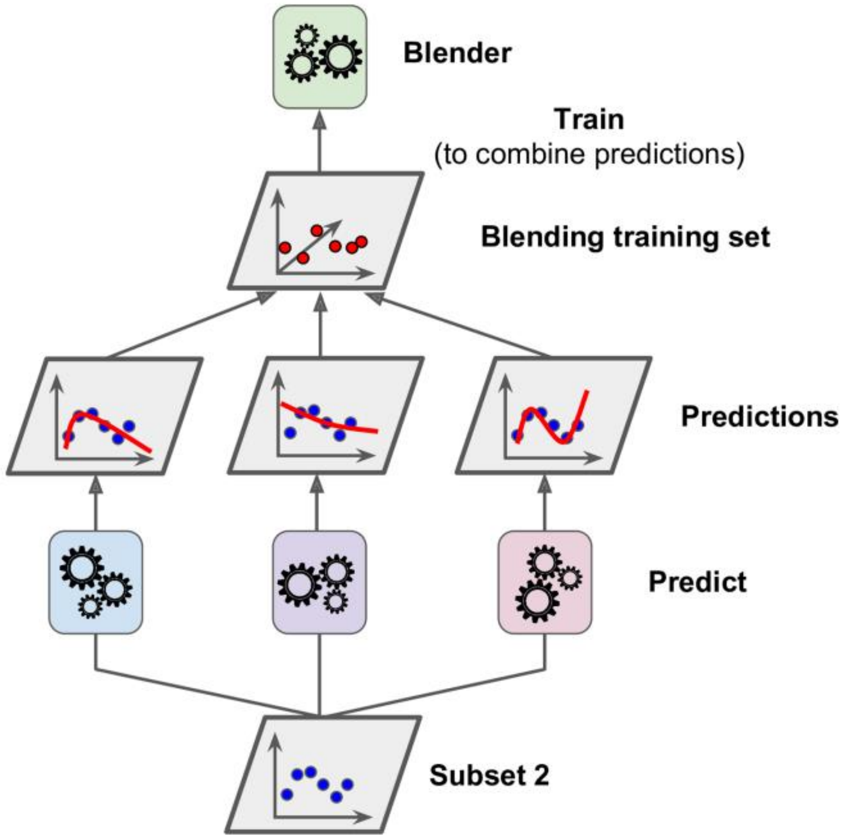
To train the blender, a common approach is to use a hold-out set. Let’s see how it works. First, the training set is split into two subsets. The first subset is used to train the predictors in the first layer (see Figure 7-13).



Next, the first layer's predictors are used to make predictions on the second (held- out) set (see Figure 7-14).

This ensures that the predictions are "clean," since the predictors never saw these instances during training. The blender is trained on this new training set, so it learns to predict the target value, given the first layer's predictions.

Unfortunately, Scikit-Learn does not support stacking directly, but it is not too hard to roll out your own implementation (see the following exercises). Alternatively, you can use an open source implementation such as `DESlib`.


Blender
Train
(to combine predictions)
Blending training set
Predictions
Predict
Subset 2