**CHAPTER 7**

**Ensemble Learning and Random Forests**



With Early Release ebooks, you get books in their earliest form— the author’s raw and unedited content as he or she writes—so you can take advantage of these technologies long before the official release of these titles. The following will be Chapter 7 in the final release of the book.



Suppose you ask a complex question to thousands of random people, then aggregate their answers. In many cases you will find that this aggregated answer is better than an expert’s answer. This is called the *wisdom of the crowd*. Similarly, 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 pre‐ dictors is called an *ensemble*; thus, this technique is called *Ensemble Learning*, and an Ensemble Learning algorithm is called an *Ensemble method*.

For example, you can train a group of Decision Tree classifiers, each on a different random subset of the training set. To make predictions, you just obtain the predic‐ tions of all individual trees, then predict the class that gets the most votes (see the last exercise in Chapter 6). Such an ensemble of Decision Trees is called a *Random Forest*, and despite its simplicity, this is one of the most powerful Machine Learning algo‐ rithms available today.

Moreover, as we discussed in Chapter 2, 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 fact, the winning solutions in Machine Learn‐ ing competitions often involve several Ensemble methods (most famously in the [Net‐](http://netflixprize.com/) [flix Prize competition](http://netflixprize.com/)).

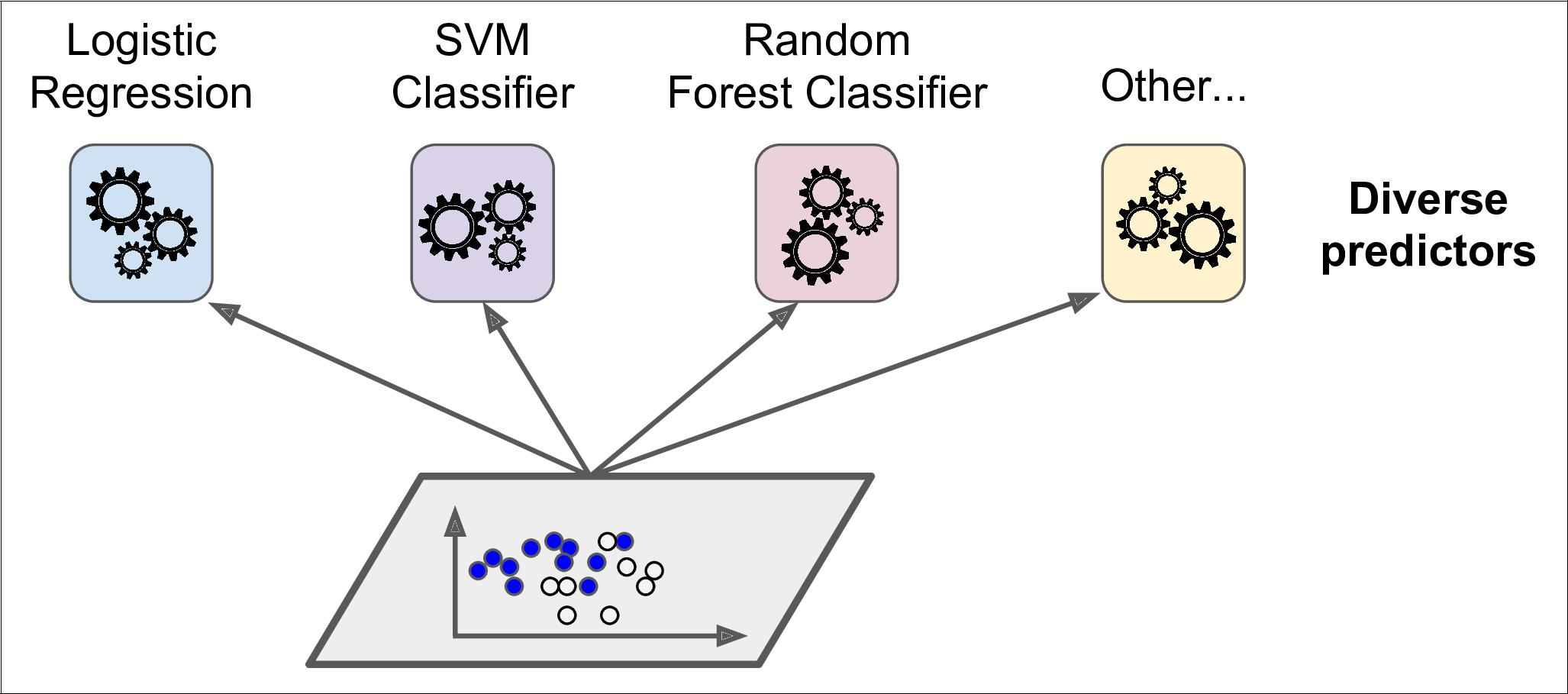
In this chapter we will discuss the most popular Ensemble methods, including *bag‐* *ging*, *boosting*, *stacking*, and a few others. We will also explore Random Forests.



**191**

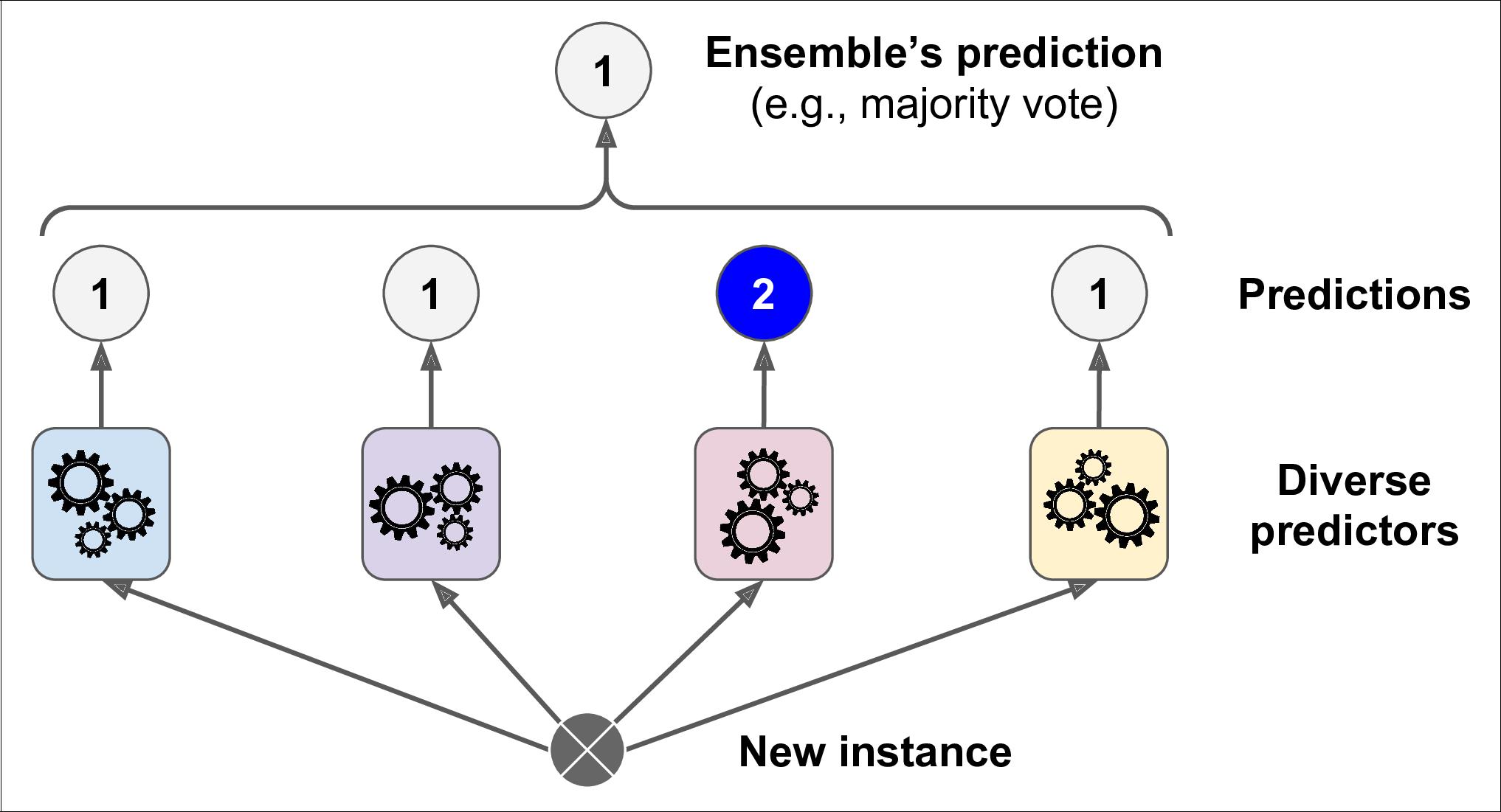
**Voting Classi€ers**

Suppose you have trained a few classifiers, each one achieving about 80% accuracy. You may have a Logistic Regression classifier, an SVM classifier, a Random Forest classifier, a K-Nearest Neighbors classifier, and perhaps a few more (see [Figure 7-1](#page2)).



*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 classi‐ fier is called a *hard voting* classifier (see [Figure 7-2](#page2)).



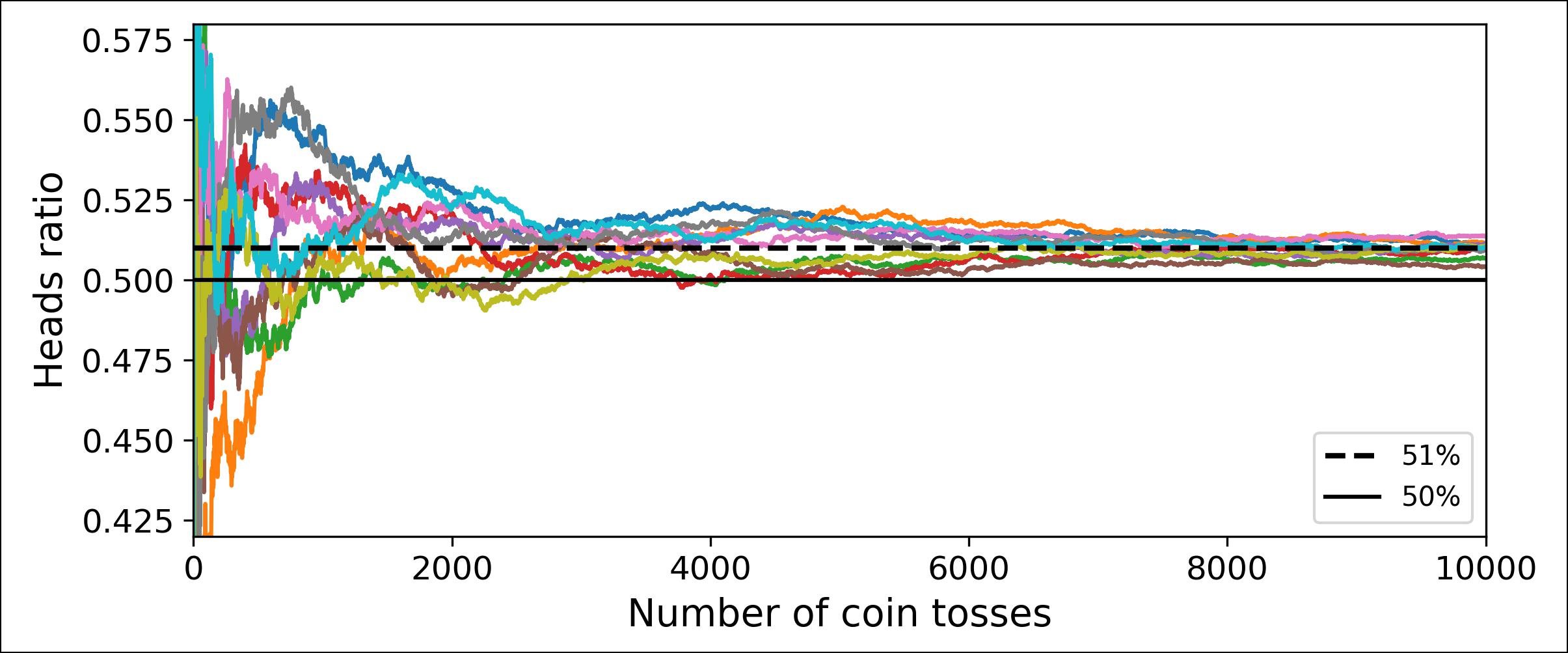
*Figure 7-2. Hard voting classifier predictions*



**192** **|** **Chapter 7: Ensemble Learning and Random Forests**

Somewhat surprisingly, 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* (mean‐ ing 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 ofweak learners and they are sufficiently diverse.

How is this possible? The following analogy can help shed some light on this mystery. 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 theprobability of heads (51%). [Figure 7-3](#page3) shows 10 series of biased coin tosses. You can see that as the number of tosses increases, the ratio of heads approaches 51%. Eventu‐ ally all 10 series end up so close to 51% that they are consistently above 50%.



*Figure 7-3. The law of large numbers*

Similarly, suppose you build an ensemble containing 1,000 classifiers that are individ‐ ually correct only 51% of the time (barely better than random guessing). If you pre‐ dict 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 since 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.



**Voting Classi€ers** **|** **193**

Ensemble methods work best when the predictors are as independ‐ ent 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.

The following code creates and trains a voting classifier in Scikit-Learn, composed of three diverse classifiers (the training set is the moons dataset, introduced in Chap‐ ter 5):

**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 classifi‐ ers.

If all classifiers are able to estimate class probabilities (i.e., they have a pre dict\_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 *so•* *voting*. It often achieves higher performance than hard voting because it gives moreweight 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 of the SVC class by default, so you need to set its probability hyperpara‐ meter to True (this will make the SVC class use cross-validation to estimate class prob‐ abilities, slowing down training, and it will add a predict\_proba() method). If you



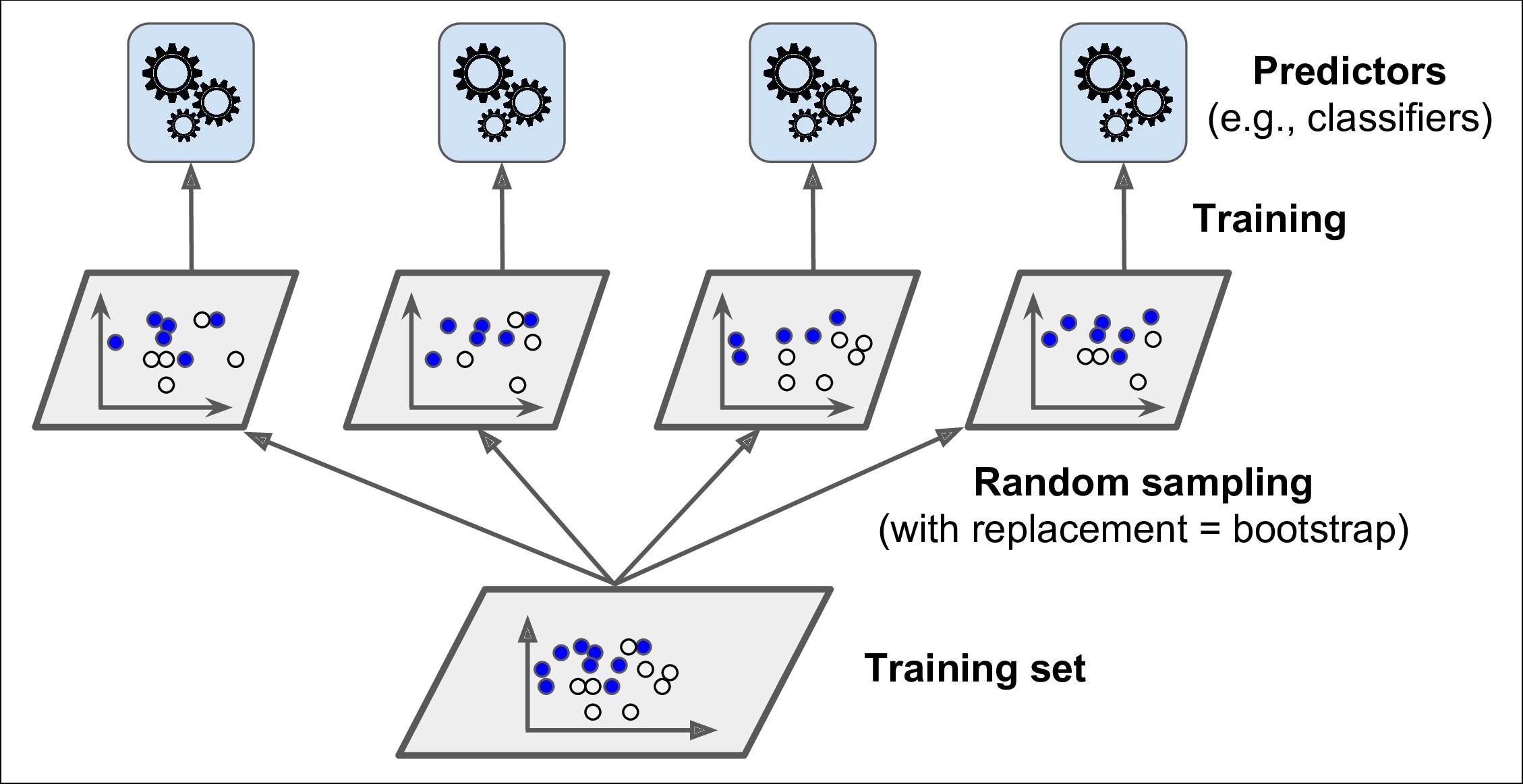
**194** **|** **Chapter 7: Ensemble Learning and Random Forests**

modify the preceding code to use soft voting, you will find that the voting classifier achieves over 91.2% accuracy!

**Bagging and Pasting**

One way to get a diverse set of classifiers is to use very different training algorithms, as just discussed. Another approach is to use the same training algorithm for every predictor, but to train them on different random subsets of the training set. When sampling is performed *with* replacement, this method is called [*bagging*](https://homl.info/20)[1](#page5) (short for *bootstrap aggregating*[2](#page5)). When sampling is performed *without* replacement, it is called[*pasting*](https://homl.info/21).[3](#page5)

In other words, both bagging and pasting allow training instances to be sampled sev‐ eral times across multiple predictors, but only bagging allows training instances to be sampled several times for the same predictor. This sampling and training process is represented in [Figure 7-4](#page5).

**

*Figure 7-4. Pasting/bagging training set sampling and training*

Once all predictors are trained, the ensemble can make a prediction for a new instance by simply aggregating the predictions of all predictors. The aggregation function is typically the *statistical mode* (i.e., the most frequent prediction, just like a hard voting classifier) for classification, or the average for regression. Each individual

**

1. “Bagging Predictors,” L. Breiman (1996).
2. In statistics, resampling with replacement is called *bootstrapping*.
3. “Pasting small votes for classification in large databases and on-line,” L. Breiman (1999).

**

**Bagging and Pasting** **|** **195**

predictor has a higher bias than if it were trained on the original training set, but aggregation reduces both bias and variance.[4](#page6) Generally, the net result is that the ensemble has a similar bias but a lower variance than a single predictor trained on the original training set.

As you can see in [Figure 7-4](#page5), predictors can all be trained in parallel, via different CPU cores or even different servers. Similarly, predictions can be made in parallel. This is one of the reasons why bagging and pasting are such popular methods: they scale very well.

**Bagging and Pasting in Scikit-Learn**

Scikit-Learn offers a simple API for both bagging and pasting with the BaggingClas sifier class (or BaggingRegressor for regression). The following code trains an ensemble of 500 Decision Tree classifiers,[5](#page6) each trained on 100 training instances ran‐ domly 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 param‐ eter tells Scikit-Learn the number of CPU cores to use for training and predictions (–1 tells Scikit-Learn to use all available cores):

**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 proba‐ bilities (i.e., if it has a predict\_proba() method), which is the case with Decision Trees classifiers.



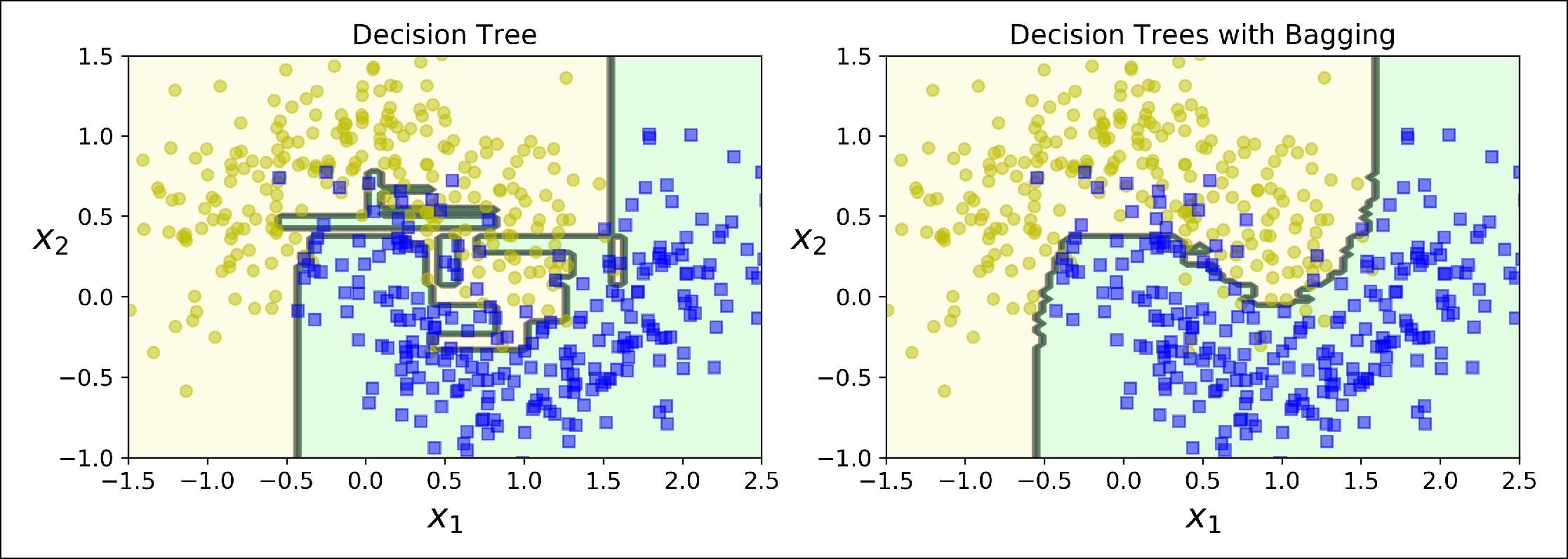
[Figure 7-5](#page7) compares the decision boundary of a single Decision Tree with the deci‐ sion boundary of a bagging ensemble of 500 trees (from the preceding code), both trained on the moons dataset. As you can see, the ensemble’s predictions will likely generalize much better than the single Decision 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).



1. Bias and variance were introduced in Chapter 4.
2. max\_samples can alternatively be set to a float between 0.0 and 1.0, in which case the max number of instances to sample is equal to the size of the training set times max\_samples.



**196** **|** **Chapter 7: Ensemble Learning and Random Forests**



*Figure 7-5. A single Decision Tree versus a bagging ensemble of 500 trees*

Bootstrapping introduces a bit more diversity in the subsets that each predictor is trained on, so bagging ends up with a slightly higher bias than pasting, but this also means that predictors end up being less correlated so the ensemble’s variance is reduced. Overall, bagging often results in better models, which explains why it is gen‐ erally preferred. However, if you have spare time and CPU power you can use cross-validation to evaluate both bagging and pasting and select the one that works best.

**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.[6](#page7) 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)



6 As *m* grows, this ratio approaches 1 – exp(–1) ≈ 63.212%.



**Bagging and Pasting** **|** **197**

**>>>** 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

We get 91.2% accuracy on the test set—close enough!

The oob decision function for each training instance is also available through the oob\_decision\_function\_ variable. In this case (since the base estimator has a pre dict\_proba() method) the decision function returns the class probabilities for each training instance. For example, the oob evaluation estimates that the first training instance has a 68.25% probability of belonging to the positive class (and 31.75% of belonging to the negative class):

**>>>** bag\_clf.oob\_decision\_function\_array([[0.31746032, 0.68253968],

[0.34117647, 0.65882353], [1. , 0. ],

...

[1. , 0. ], [0.03108808, 0.96891192], [0.57291667, 0.42708333]])

**Random Patches and Random Subspaces**

The BaggingClassifier class supports sampling the features as well. This is con‐ trolled 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 is particularly useful when you are dealing with high-dimensional inputs (such as images). Sampling both training instances and features is called the [*Random*](https://homl.info/22) [*Patches* method](https://homl.info/22).[7](#page8)Keepingall training instances (i.e.,bootstrap=Falseandmax\_samples=1.0) but sampling features (i.e., bootstrap\_features=True and/or max\_fea tures smaller than 1.0) is called the [*Random Subspaces* method](https://homl.info/23).[8](#page8)

**

1. “Ensembles on Random Patches,” G. Louppe and P. Geurts (2012).
2. “The random subspace method for constructing decision forests,” Tin Kam Ho (1998).

**

**198** **|** **Chapter 7: Ensemble Learning and Random Forests**

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](https://homl.info/24)[9](#page9) 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 pass‐ ing it a DecisionTreeClassifier, you can instead use the RandomForestClassifier class, which is more convenient and optimized for Decision Trees[10](#page9) (similarly, there is a RandomForestRegressor class for regression tasks). The following code trains a Random Forest classifier with 500 trees (each limited to maximum 16 nodes), using all available CPU cores:

**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 hyperpara‐ meters of a BaggingClassifier to control the ensemble itself.[11](#page9)

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. This results in a greater tree diversity, which (once 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)



1. “Random Decision Forests,” T. Ho (1995).
2. The BaggingClassifier class remains useful if you want a bag of something other than Decision Trees.
3. There are a few notable exceptions: splitter is absent (forced to "random"), presort is absent (forced to

False), max\_samples is absent (forced to 1.0), and base\_estimator is absent (forced to DecisionTreeClassi fier with the provided hyperparameters).



**Random Forests** **|** **199**

**Extra-Trees**

When you are growing a tree in a Random Forest, at each node only a random subset of the features is considered for splitting (as discussed earlier). It is possible to make trees even more random by also using random thresholds for each feature rather than searching for the best possible thresholds (like regular Decision Trees do).

A forest of such extremely random trees is simply called an [*Extremely Randomized*](https://homl.info/25) [*Trees*](https://homl.info/25) ensemble[12](#page10)(or *Extra-Trees* for short). Once again, this trades more bias for alower variance. It also makes Extra-Trees much faster to train than regular Random Forests since finding the best possible threshold for each feature at every node is one of the most time-consuming tasks of growing a tree.

You can create an Extra-Trees classifier using Scikit-Learn’s ExtraTreesClassifier class. Its API is identical to the RandomForestClassifier class. Similarly, the Extra TreesRegressor class has the same API as the RandomForestRegressor class.



It is hard to tell in advance whether a RandomForestClassifier will perform better or worse than an ExtraTreesClassifier. Gen‐ erally, the only way to know is to try both and compare them using cross-validation (and tuning the hyperparameters using grid search).

**Feature Importance**

Yet another great quality of Random Forests is that they make it easy to measure the relative importance of each feature. 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 (see Chapter 6).

Scikit-Learn computes this score automatically for each feature after training, then it scales the results so that the sum of all importances is equal to 1. You can access the result using the feature\_importances\_ variable. For example, the following code trains a RandomForestClassifier on the iris dataset (introduced in Chapter 4) and outputs each feature’s importance. It seems that the most important features are the petal length (44%) and width (42%), while sepal length and width are rather unim‐ portant in comparison (11% and 2%, respectively).



12 “Extremely randomized trees,” P. Geurts, D. Ernst, L. Wehenkel (2005).



**200** **|** **Chapter 7: Ensemble Learning and Random Forests**

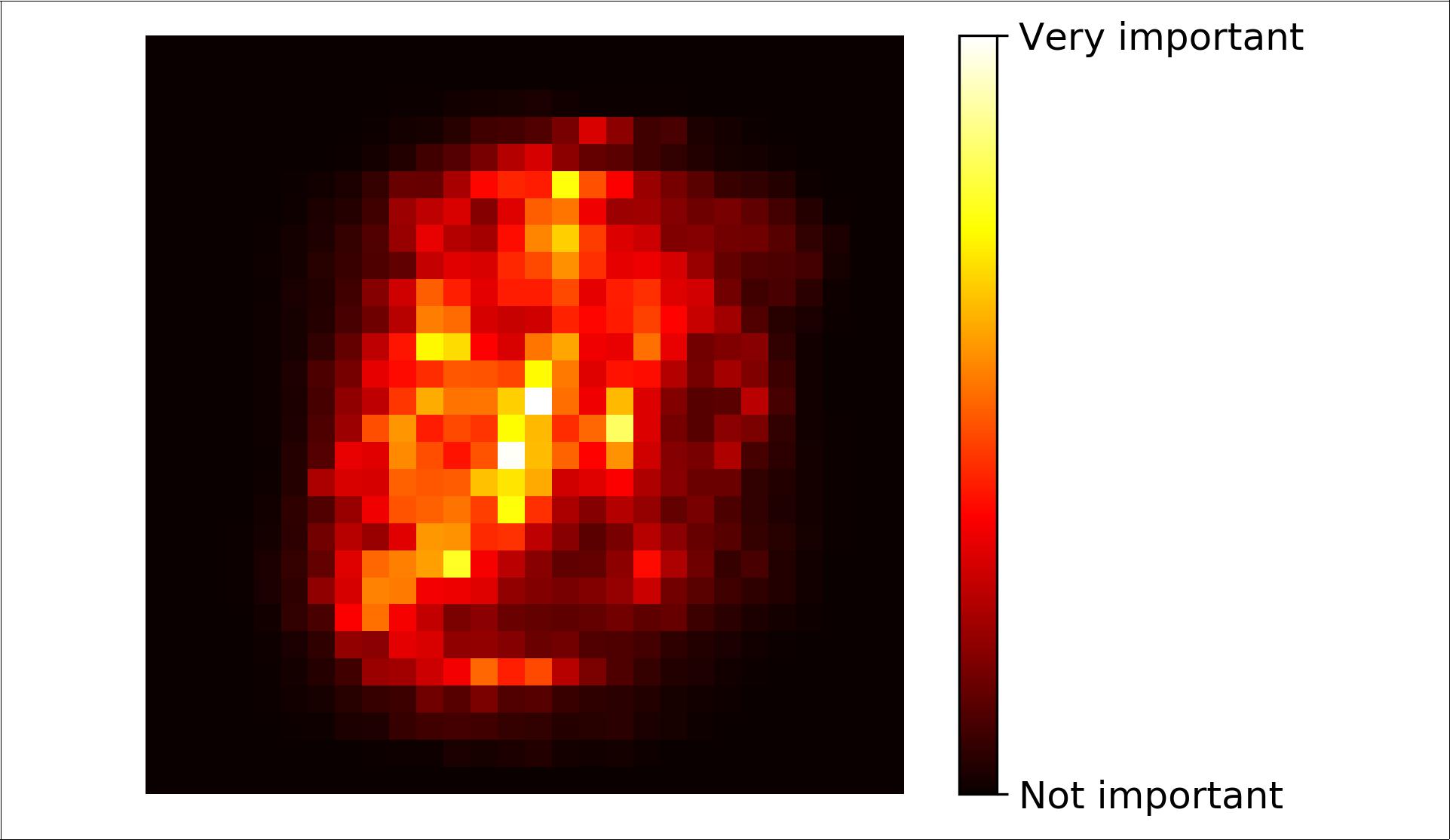
* **from sklearn.datasets import** load\_iris
* iris = load\_iris()
* rnd\_clf = RandomForestClassifier(n\_estimators=500, n\_jobs=-1)
* rnd\_clf.fit(iris["data"], iris["target"])
* **for** name,score **in** zip(iris["feature\_names"],rnd\_clf.feature\_importances\_):

**...print**(name,score)

**...**

sepal length (cm) 0.112492250999 sepal width (cm) 0.0231192882825 petal length (cm) 0.441030464364 petal width (cm) 0.423357996355

Similarly, if you train a Random Forest classifier on the MNIST dataset (introduced in Chapter 3) and plot each pixel’s importance, you get the image represented in [Figure 7-6](#page11).



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

Random Forests are very handy to get a quick understanding of what features actually matter, in particular if you need to perform feature selection.

**Boosting**

*Boosting* (originally called *hypothesis boosting*) refers to any Ensemble method thatcan 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 prede‐ cessor. There are many boosting methods available, but by far the most popular are



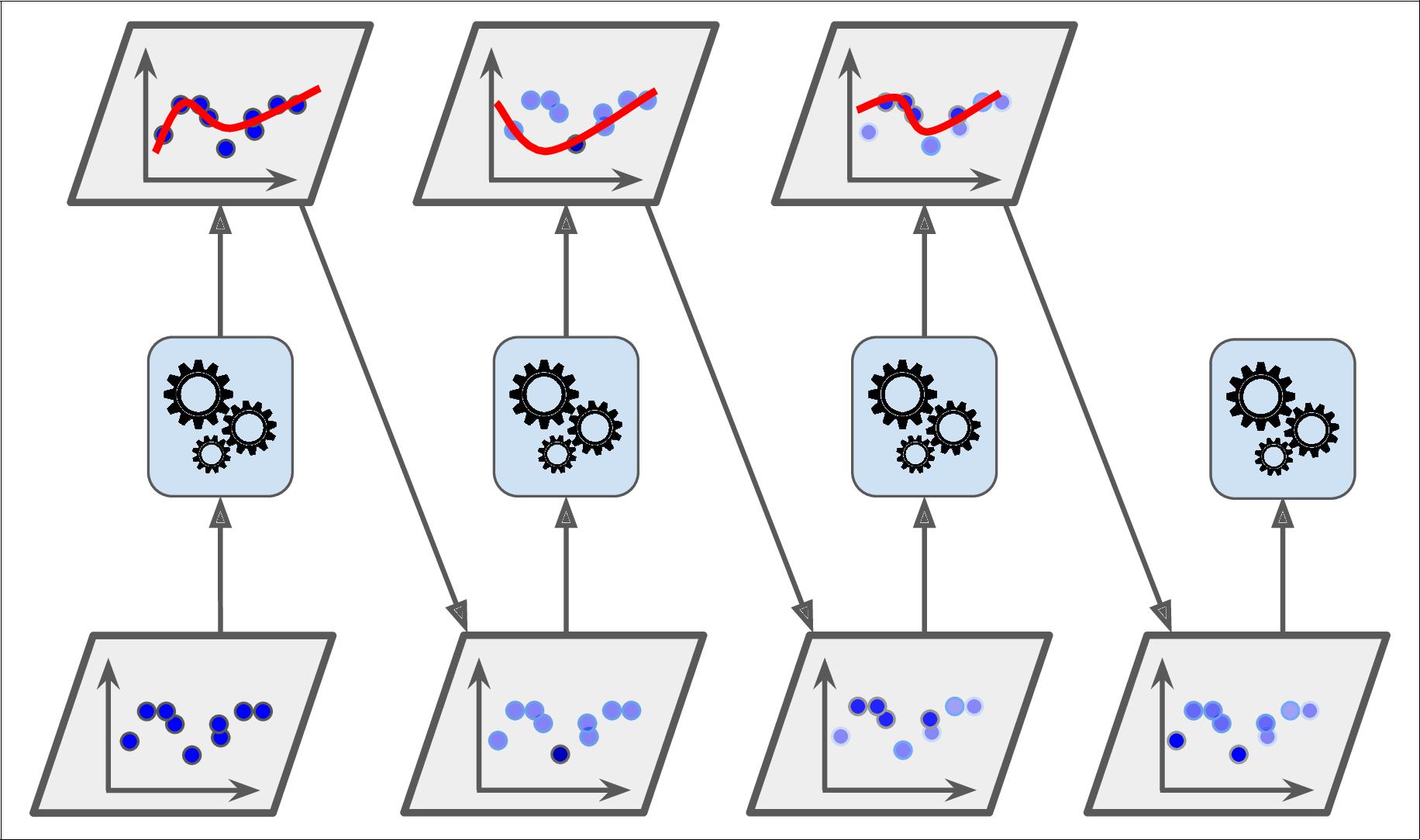
**Boosting** **|** **201**

[*AdaBoost*](https://homl.info/26)[13](#page12)(shortfor *Adaptive Boosting*) and *Gradient Boosting*. Let’s start with Ada‐Boost.

**AdaBoost**

One way for a new predictor to correct its predecessor is to pay a bit more attention to the training instances that the predecessor underfitted. This results in new predic‐ tors focusing more and more on the hard cases. This is the technique used by Ada‐ Boost.

For example, to build an AdaBoost classifier, a first base classifier (such as a Decision Tree) is trained and used to make predictions on the training set. The relative weight of misclassified training instances is then increased. A second classifier is trained using the updated weights and again it makes predictions on the training set, weights are updated, and so on (see [Figure 7-7](#page12)).



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

[Figure 7-8](#page13) shows the decision boundaries of five consecutive predictors on the moons dataset (in this example, each predictor is a highly regularized SVM classifier with an RBF kernel[14](#page12)). The first classifier gets many instances wrong, so their weights

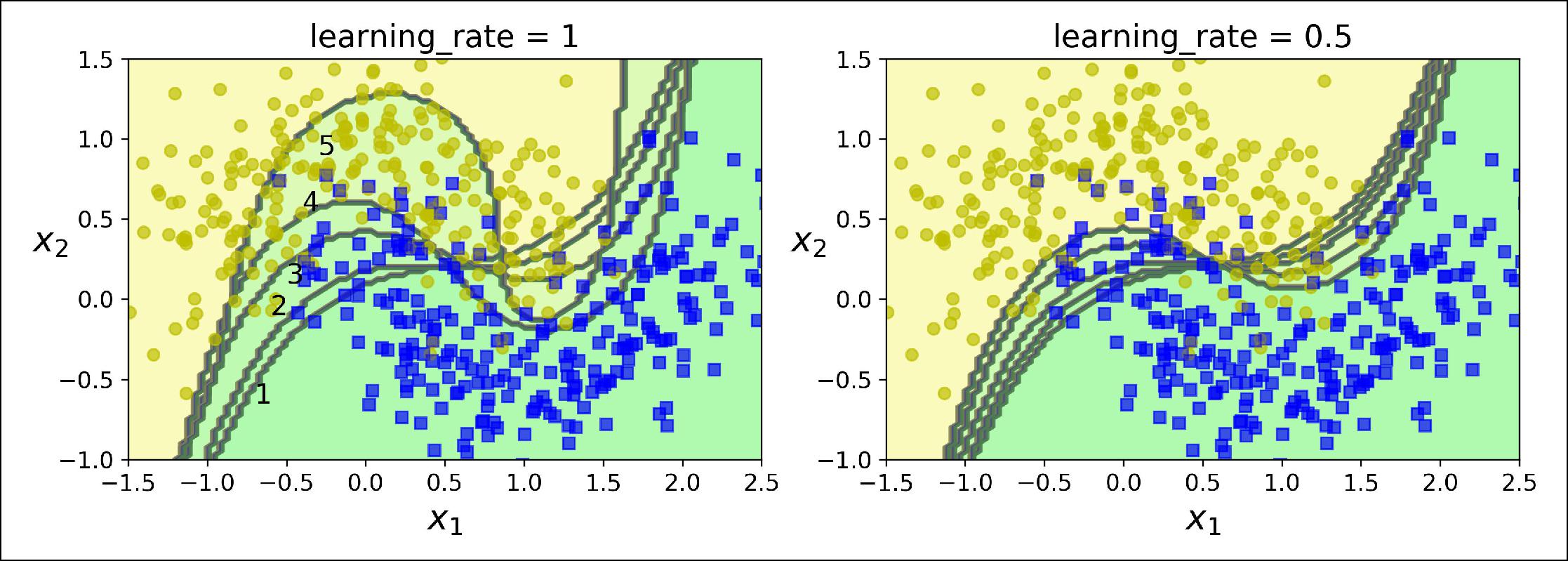


1. “A Decision-Theoretic Generalization of On-Line Learning and an Application to Boosting,” Yoav Freund, Robert E. Schapire (1997).
2. This is just for illustrative purposes. SVMs are generally not good base predictors for AdaBoost, because they are slow and tend to be unstable with AdaBoost.



**202** **|** **Chapter 7: Ensemble Learning and Random Forests**

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.



*Figure 7-8. Decision boundaries of consecutive predictors*

Once all predictors are trained, the ensemble makes predictions very much like bag‐ ging or pasting, except that predictors have different weights depending on their overall accuracy on the weighted training set.

There is one important drawback to this sequential learning techni‐ que: it cannot be parallelized (or only partially), since each predic‐ tor can only be trained after the previous predictor has been trained and evaluated. As a result, it does not scale as well as bag‐ ging or pasting.



Let’s take a closer look at the AdaBoost algorithm. Each instance weight *w(i)* is initially set to *m*1 . A first predictor is trained and its weighted error rate *r*1 is computed on the training set; see [Equation 7-1](#page13).

*Equation 7-1. Weighted error rate of the jth predictor*

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  | *m* | *w i* |  |  |  |
|  |  |  |  | ∑ |  |  |  |
|  |  |  | *i* = 1 | |  |  |  |  |
| *r* | *j* | = | *y ji* | ≠ *y* *i* |  | where *y* *i* | is the *j*th predictor’s prediction for the *i*th instance. |  |
|  | *m* |  |  |
|  |  |  |  | *j* |  |  |
|  |  |  |  | ∑ *w* *i* | |  |  |  |
|  |  |  | *i* = 1 | |  |  |  |  |



**Boosting** **|** **203**

The predictor’s weight *αj* is then computed using [Equation 7-2](#page14), where *η* is the learn‐ ing rate hyperparameter (defaults to 1).[15](#page14) The more accurate the predictor is, the higher its weight will be. If it is just guessing randomly, then its weight will be close to zero. However, if it is most often wrong (i.e., less accurate than random guessing), then its weight will be negative.

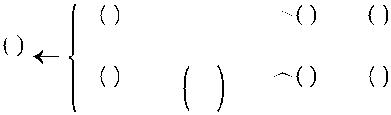
*Equation 7-2. Predictor weight*

*αj* = *η* log1 − *rj rj*

Next the instance weights are updated using [Equation 7-3](#page14): the misclassified instances are boosted.

*Equation 7-3. Weight update rule*

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| for *i* = 1, 2, ⋯, *m* | |  |  |  |  |  |  |
|  | *w i* |  | if | *y* | *i* | = *y* *i* |  |
| *w i* |  |  |  |  | *j* |  |  |
| *w i* exp *α* |  | if | *y* | *i* | ≠ *y* *i* |  |
|  | *j* |  |
|  |  |  |  | *j* |  |  |



Then all the instance weights are normalized (i.e., divided by ∑*mi*= 1 *w**i*).

Finally, a new predictor is trained using the updated weights, and the whole process is repeated (the new predictor’s weight is computed, the instance weights are updated, then another predictor is trained, and so on). 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 *αj*. The predicted class is the one that receives the majority of weighted votes (see [Equation 7-4](#page14)).

*Equation 7-4. AdaBoost predictions*

|  |  |  |  |
| --- | --- | --- | --- |
| *y* **x**= argmax | *N* | *αj* where *N* is the number of predictors. |  |
| ∑ |  |
| *k* | *j* = 1 | |  |
|  | *y j* **x** | = *k* |  |



15 The original AdaBoost algorithm does not use a learning rate hyperparameter.



**204** **|** **Chapter 7: Ensemble Learning and Random Forests**

Scikit-Learn actually uses a multiclass version of AdaBoost called [*SAMME*](https://homl.info/27)[16](#page15) (which stands for *Stagewise Additive Modeling using a Multiclass Exponential loss function*). When there are just two classes, SAMME is equivalent to AdaBoost. Moreover, 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 Ada BoostRegressor 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)



If your AdaBoost ensemble is overfitting the training set, you can try reducing the number of estimators or more strongly regulariz‐ ing the base estimator.

**Gradient Boosting**

Another very popular Boosting algorithm is [*Gradient Boosting*](https://homl.info/28).[17](#page15) Just like AdaBoost, Gradient Boosting works by sequentially adding predictors to an ensemble, each one correcting its predecessor. However, 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 predic‐ tors (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):



1. For more details, see “Multi-Class AdaBoost,” J. Zhu et al. (2006).
2. First introduced in “Arcing the Edge,” L. Breiman (1997), and further developed in the paper “Greedy Func‐ tion Approximation: A Gradient Boosting Machine,” Jerome H. Friedman (1999).



**Boosting** **|** **205**

**from sklearn.tree import** DecisionTreeRegressor

tree\_reg1 = DecisionTreeRegressor(max\_depth=2) tree\_reg1.fit(X, y)

Now 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))

[Figure 7-9](#page17) represents the predictions of these three trees in the left column, and the ensemble’s predictions in the right column. In the first row, the ensemble has just one tree, so its predictions are exactly the same as the first tree’s predictions. In the second row, a new tree is trained on the residual errors of the first tree. On the right you can see that the ensemble’s predictions are equal to the sum of the predictions of the first two trees. Similarly, in the third row another tree is trained on the residual errors of the second tree. You can see that the ensemble’s predictions gradually get better as trees are added to the ensemble.

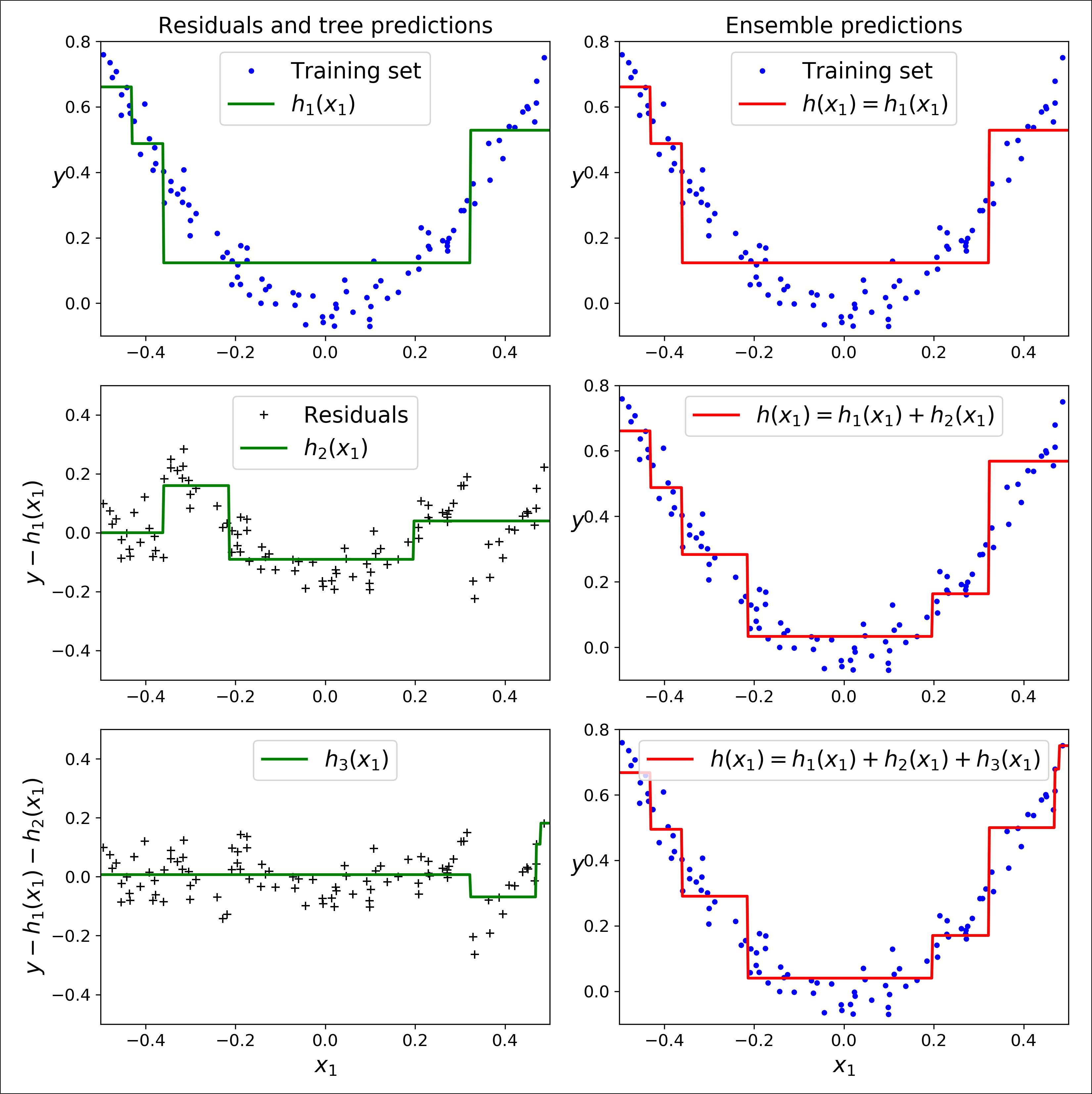
A simpler way to train GBRT ensembles is to use Scikit-Learn’s GradientBoostingRe gressor class. Much like the RandomForestRegressor class, it has hyperparameters to control the growth of Decision Trees (e.g., max\_depth, min\_samples\_leaf, and so on), as well as hyperparameters to control the ensemble training, such as the number of trees (n\_estimators). The following code creates the same ensemble as the previous one:

**from sklearn.ensemble import** GradientBoostingRegressor

gbrt = GradientBoostingRegressor(max\_depth=2, n\_estimators=3, learning\_rate=1.0) gbrt.fit(X, y)



**206** **|** **Chapter 7: Ensemble Learning and Random Forests**

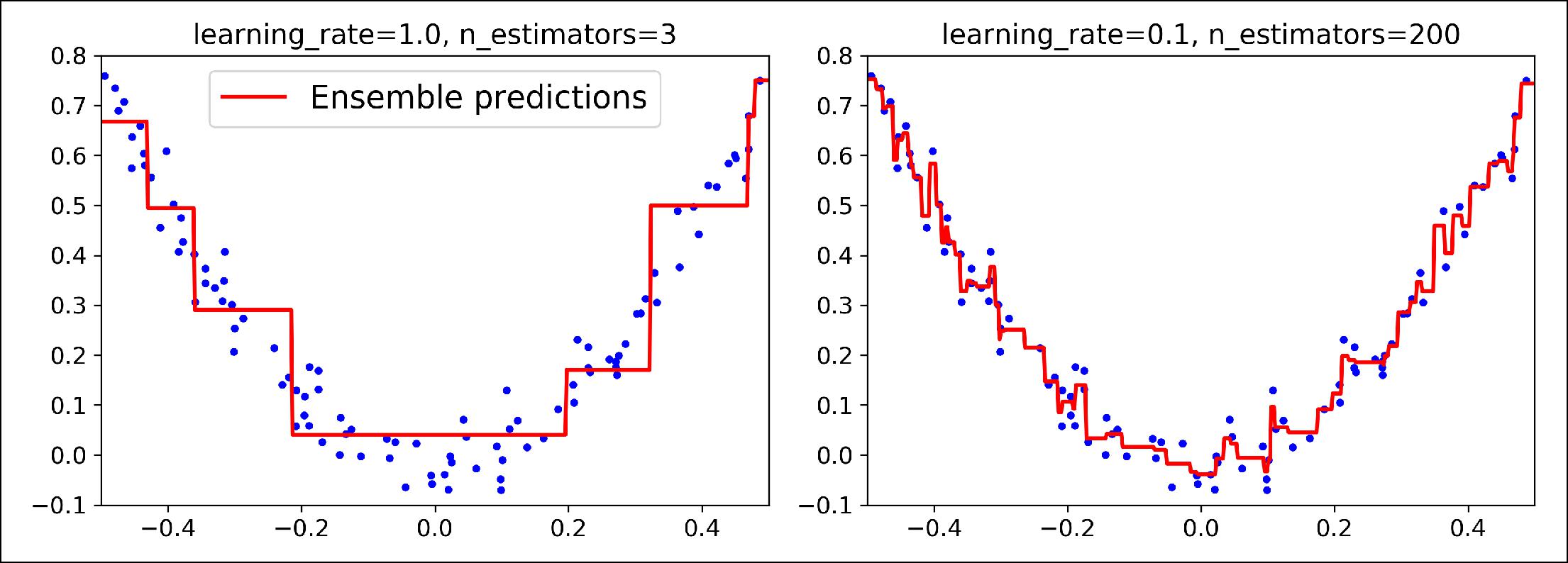


*Figure 7-9. Gradient Boosting*

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 train‐ ing set, but the predictions will usually generalize better. This is a regularization tech‐ nique called *shrinkage*. [Figure 7-10](#page18) 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.



**Boosting** **|** **207**



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

In order to find the optimal number of trees, you can use early stopping (see Chap‐ ter 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 train‐ ing (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 opti‐ mal 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)

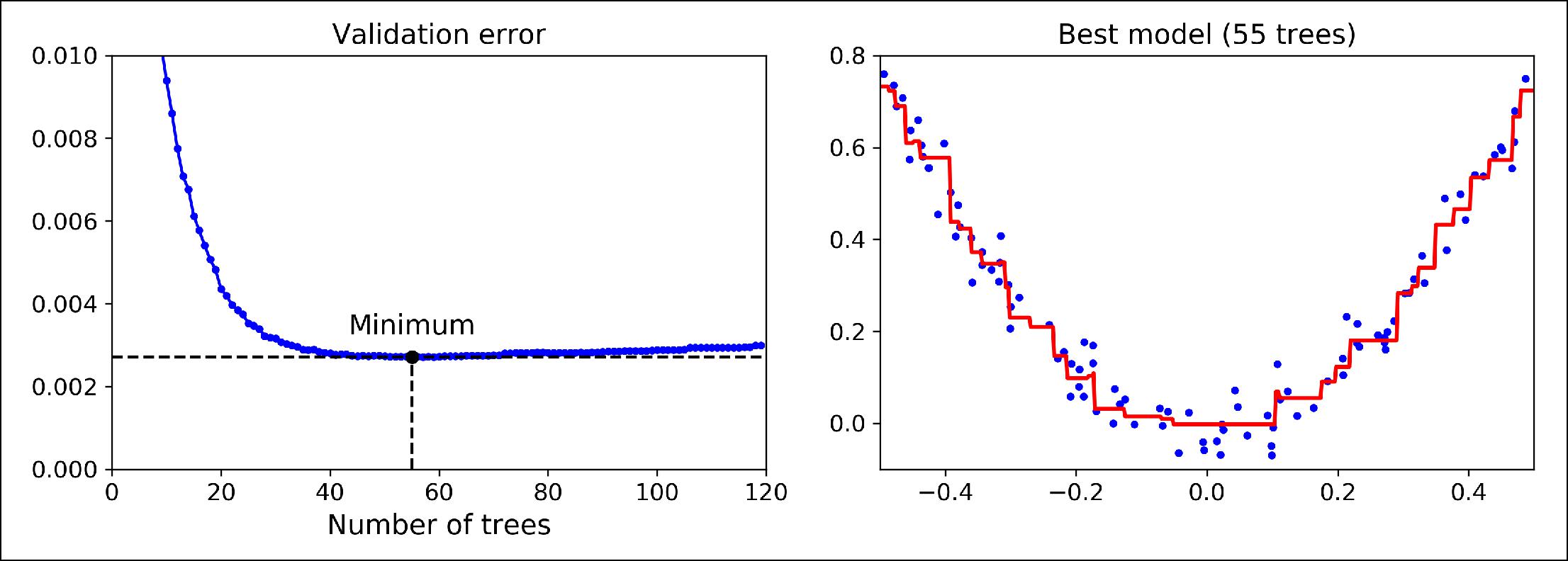
**for** y\_pred **in** gbrt.staged\_predict(X\_val)]bst\_n\_estimators = np.argmin(errors)

gbrt\_best = GradientBoostingRegressor(max\_depth=2,n\_estimators=bst\_n\_estimators) gbrt\_best.fit(X\_train, y\_train)

The validation errors are represented on the left of [Figure 7-11](#page19), and the best model’s predictions are represented on the right.



**208** **|** **Chapter 7: Ensemble Learning and Random Forests**



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

It is also possible to implement early stopping by actually stopping training early (instead of training a large number of trees first and then looking back to find the optimal number). You can do so by setting warm\_start=True, which makes Scikit-Learn keep existing trees when the fit() method is called, allowing incremental training. The following code stops training when the validation error does not improve for five iterations in a row:

gbrt = GradientBoostingRegressor(max\_depth=2, warm\_start=True)

min\_val\_error = float("inf") error\_going\_up = 0

**for** n\_estimators **in** range(1, 120): gbrt.n\_estimators = n\_estimators gbrt.fit(X\_train, y\_train) y\_pred = gbrt.predict(X\_val)

val\_error = mean\_squared\_error(y\_val, y\_pred) **if** val\_error<min\_val\_error:

min\_val\_error = val\_error error\_going\_up = 0

**else**:

error\_going\_up += 1

**if** error\_going\_up==5: **break** *# early stopping*

The GradientBoostingRegressor class also supports a subsample hyperparameter, which specifies the fraction of training instances to be used for training each tree. For example, if subsample=0.25, then each tree is trained on 25% of the training instan‐ ces, selected randomly. As you can probably guess by now, this trades a higher bias for a lower variance. It also speeds up training considerably. This technique is called

*Stochastic Gradient Boosting*.



**Boosting** **|** **209**

It is possible to use Gradient Boosting with other cost functions. This is controlled by the loss hyperparameter (see Scikit-Learn’s documentation for more details).



It is worth noting that an optimized implementation of Gradient Boosting is available in the popular python library [*XGBoost*](https://github.com/dmlc/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](http://dmlc.ml/)), and it aims at being 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:

**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:

xgb\_reg.fit(X\_train, y\_train,

eval\_set=[(X\_val, y\_val)], early\_stopping\_rounds=2) y\_pred = xgb\_reg.predict(X\_val)

You should definitely check it out!

**Stacking**

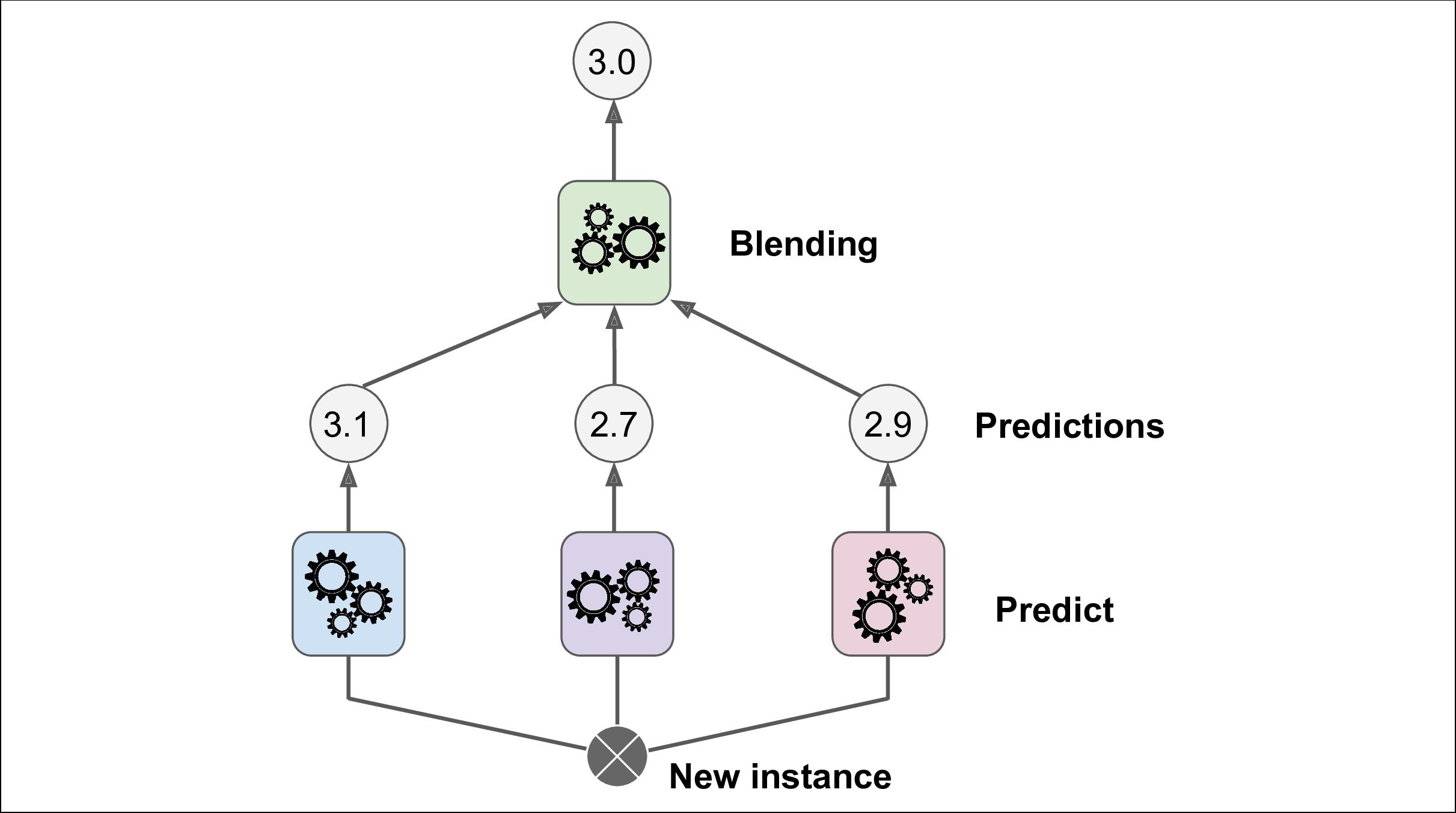
The last Ensemble method we will discuss in this chapter is called *stacking* (short for [*stacked generalization*](https://homl.info/29)).[18](#page20)Itis based on a simple idea: instead of using trivial functions(such as hard voting) to aggregate the predictions of all predictors in an ensemble, why don’t we train a model to perform this aggregation? [Figure 7-12](#page21) 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).

**

18 “Stacked Generalization,” D. Wolpert (1992).

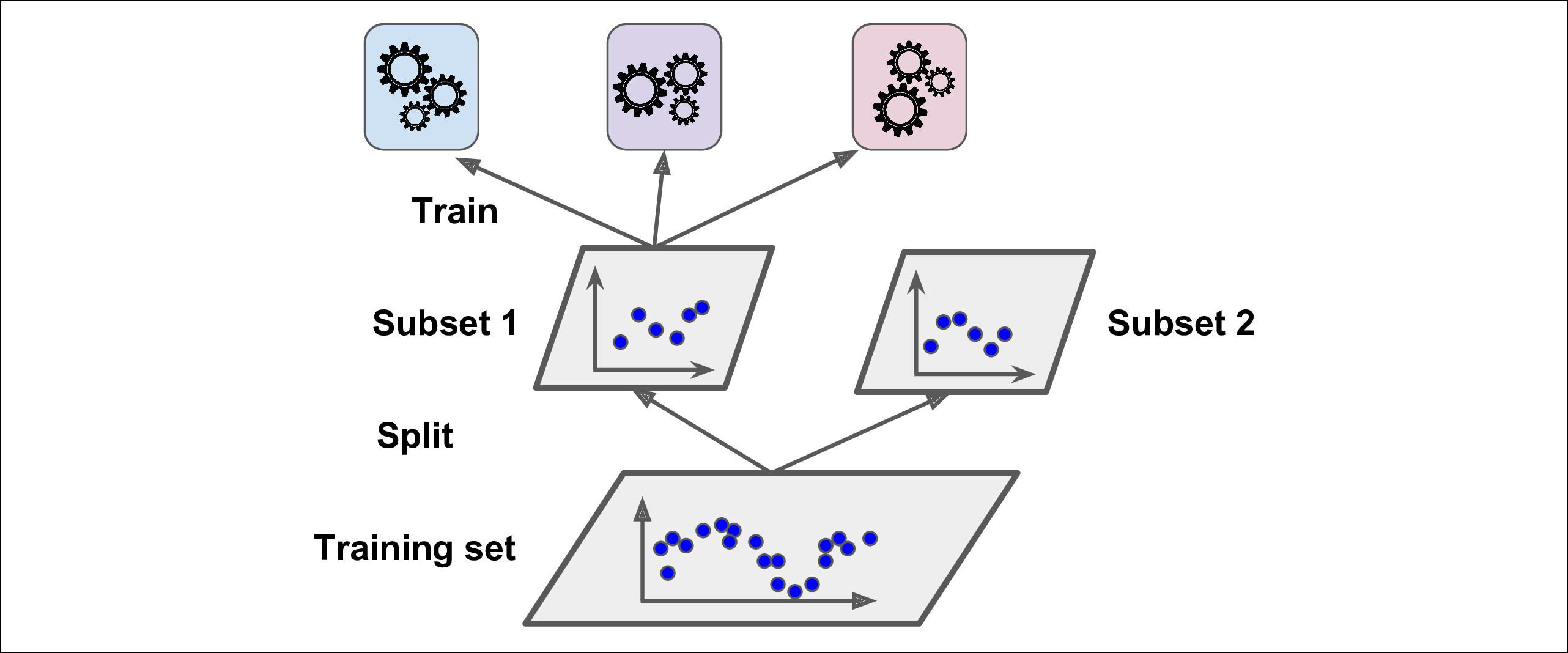
**

**210** **|** **Chapter 7: Ensemble Learning and Random Forests**



*Figure 7-12. Aggregating predictions using a blending predictor*

To train the blender, a common approach is to use a hold-out set.[19](#page21) Let’s see how it works. First, the training set is split in two subsets. The first subset is used to train the predictors in the first layer (see [Figure 7-13](#page21)).



*Figure 7-13. Training the first layer*

Next, the first layer predictors are used to make predictions on the second (held-out) set (see [Figure 7-14](#page22)). This ensures that the predictions are “clean,” since the predictors never saw these instances during training. Now for each instance in the hold-out set

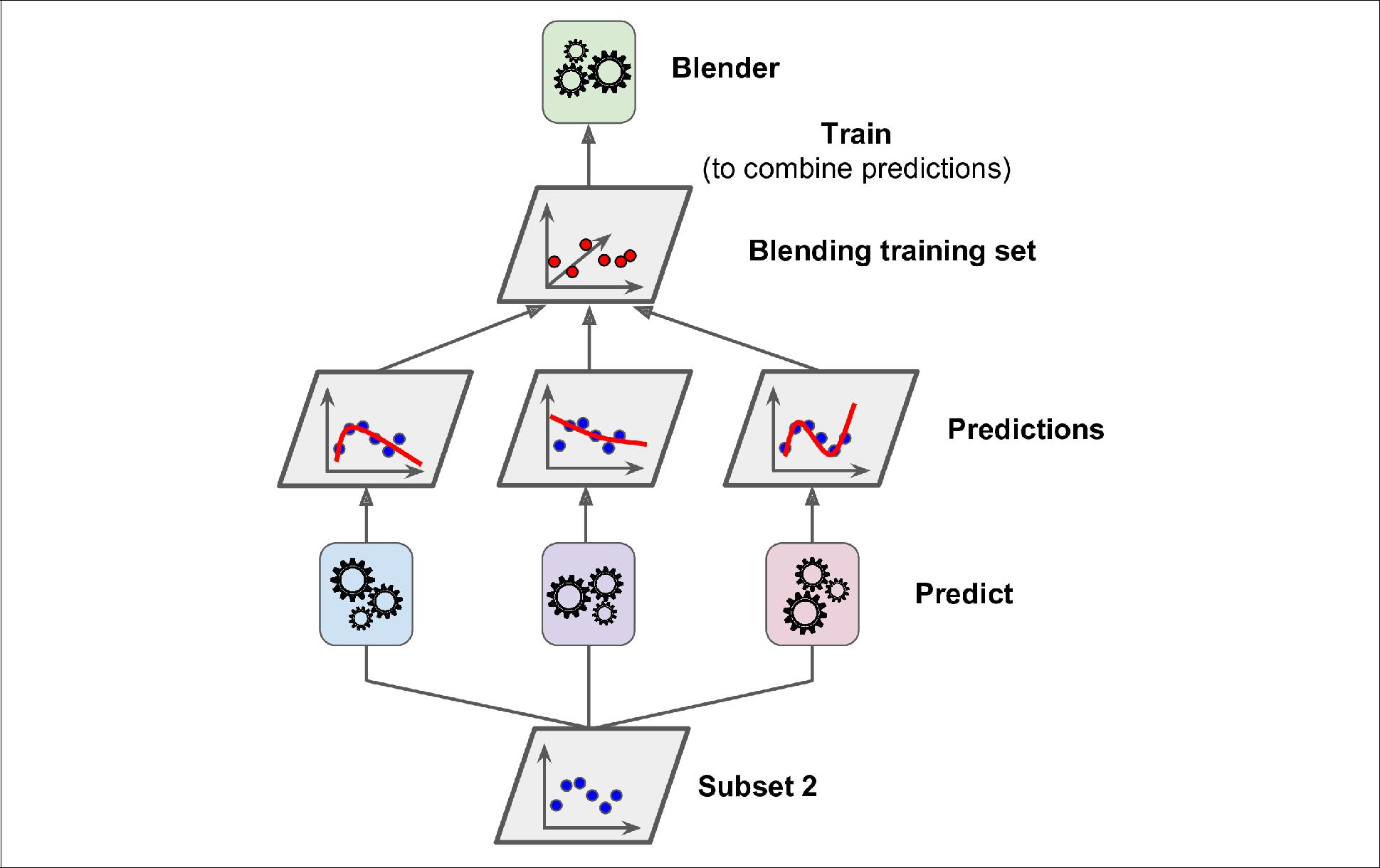


1. Alternatively, it is possible to use out-of-fold predictions. In some contexts this is called *stacking*, while using a hold-out set is called *blending*. However, for many people these terms are synonymous.



**Stacking** **|** **211**

there are three predicted values. We can create a new training set using these predic‐ ted values as input features (which makes this new training set three-dimensional), and keeping the target values. The blender is trained on this new training set, so it learns to predict the target value given the first layer’s predictions.

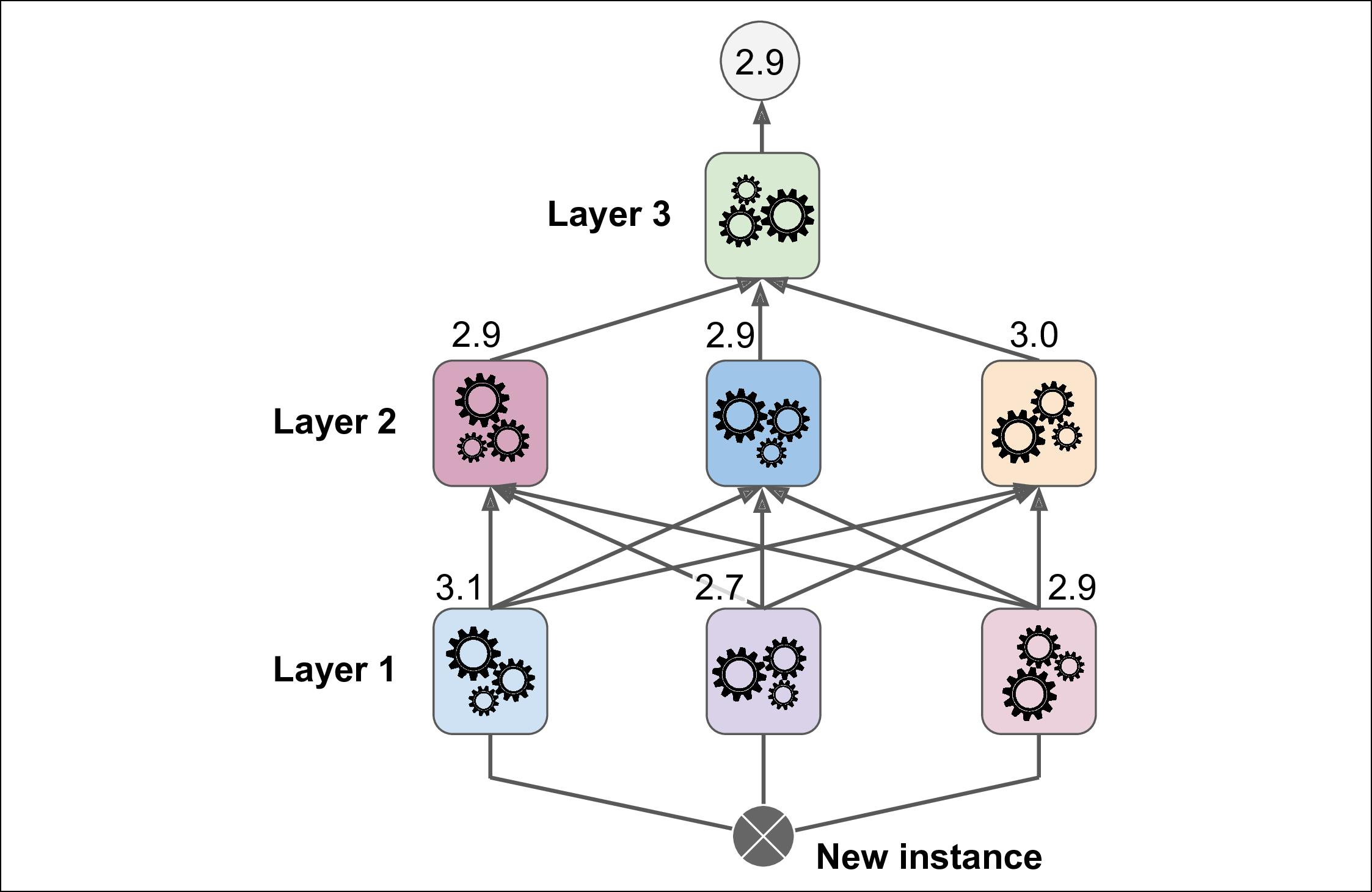


*Figure 7-14. Training the blender*

It is actually possible to train several different blenders this way (e.g., one using Lin‐ ear Regression, another using Random Forest Regression, and so on): we get a whole layer of blenders. The trick is to split the training set into three subsets: the first one is used to train the first layer, the second one is used to create the training set used to train the second layer (using predictions made by the predictors of the first layer), and the third one is used to create the training set to train the third layer (using pre‐ dictions made by the predictors of the second layer). Once this is done, we can make a prediction for a new instance by going through each layer sequentially, as shown in [Figure 7-15](#page23).



**212** **|** **Chapter 7: Ensemble Learning and Random Forests**



*Figure 7-15. Predictions in a multilayer stacking ensemble*

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 brew (available at [*https://github.com/*](https://github.com/viisar/brew) [*viisar/brew*](https://github.com/viisar/brew)).

**Exercises**

1. If you have trained five different models on the exact same training data, and they all achieve 95% precision, is there any chance that you can combine these models to get better results? If so, how? If not, why?
2. What is the difference between hard and soft voting classifiers?
3. Is it possible to speed up training of a bagging ensemble by distributing it across multiple servers? What about pasting ensembles, boosting ensembles, random forests, or stacking ensembles?
4. What is the benefit of out-of-bag evaluation?
5. What makes Extra-Trees more random than regular Random Forests? How can this extra randomness help? Are Extra-Trees slower or faster than regular Ran‐ dom Forests?
6. If your AdaBoost ensemble underfits the training data, what hyperparameters should you tweak and how?



**Exercises** **|** **213**

1. If your Gradient Boosting ensemble overfits the training set, should you increase or decrease the learning rate?
2. Load the MNIST data (introduced in Chapter 3), and split it into a training set, a validation set, and a test set (e.g., use 50,000 instances for training, 10,000 for val‐ idation, and 10,000 for testing). Then train various classifiers, such as a Random Forest classifier, an Extra-Trees classifier, and an SVM. Next, try to combine them into an ensemble that outperforms them all on the validation set, using a soft or hard voting classifier. Once you have found one, try it on the test set. How much better does it perform compared to the individual classifiers?
3. Run the individual classifiers from the previous exercise to make predictions on the validation set, and create a new training set with the resulting predictions: each training instance is a vector containing the set of predictions from all your classifiers for an image, and the target is the image’s class. Train a classifier on this new training set. Congratulations, you have just trained a blender, and together with the classifiers they form a stacking ensemble! Now let’s evaluate the ensemble on the test set. For each image in the test set, make predictions with all your classifiers, then feed the predictions to the blender to get the ensemble’s pre‐ dictions. How does it compare to the voting classifier you trained earlier?

Solutions to these exercises are available in ???.



**214** **|** **Chapter 7: Ensemble Learning and Random Forests**