Shanghai Advanced Institute of Finance
上海高级金融学院

量化俱乐部-机器学习-Ensemble Learning and Random Forests

2019-08-31

Menu

1. Voting Classifiers
2. Bagging and Pasting
3. Random Forests
4. Boosting
5. Stacking

自我介绍

张骁喆，2010年大连理工大学软件学院本科毕业，高金FMBA 2017PTE。9年计算机工作经验，熟悉开发，测试，运维，项目管理，产品设计各环节。
曾就职eBay，唯品会，2016加入SAP，现担任SAP Cloud部门开发团队主管。
2017开始接触量化投资和python，目前毕业论文研究方向使用机器学习在A股进行量化投资。
量化投资/机器学习咨询培训/项目管理产品管理。

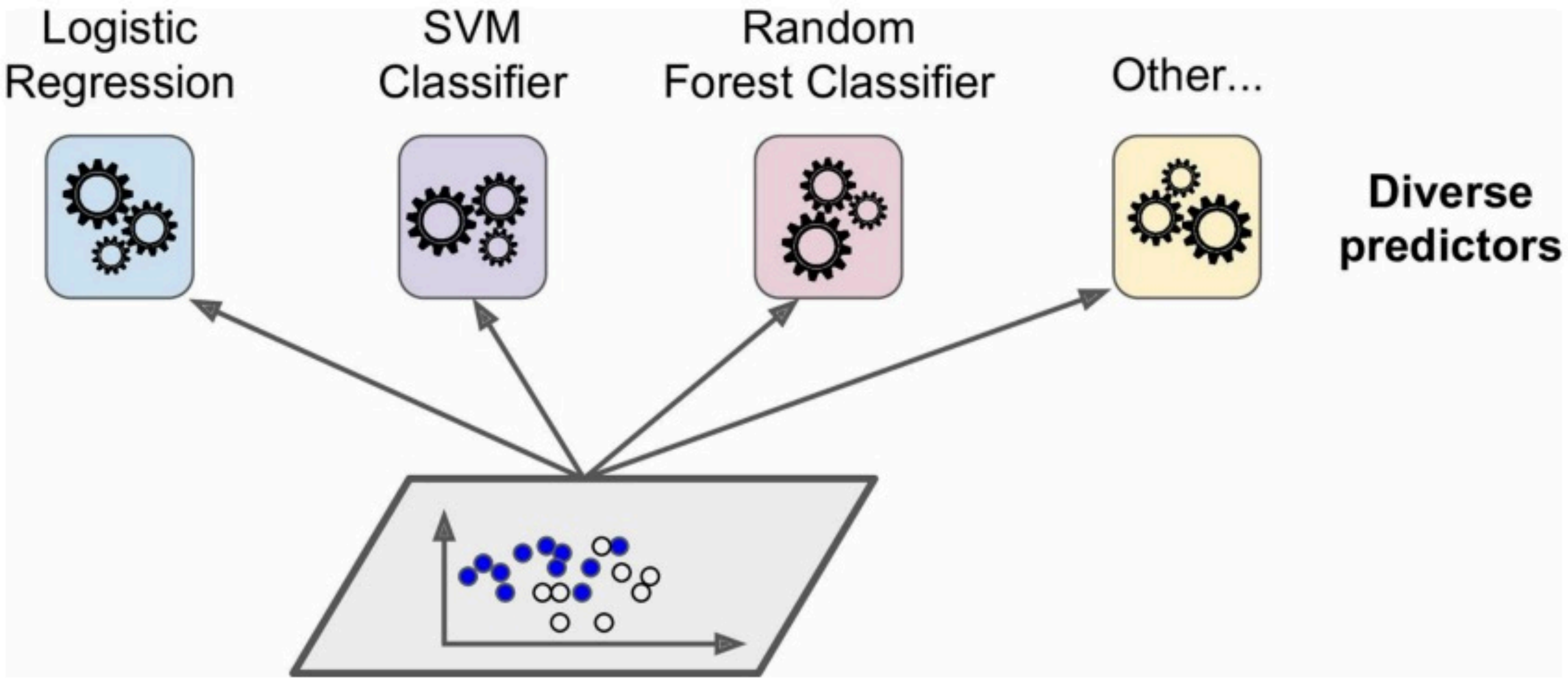
Intro

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 predictors is called an ensemble; thus, this technique is called Ensemble Learning, and an Ensemble Learning algorithm is called an Ensemble method.

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

Voting Classifiers

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.



Voting 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.

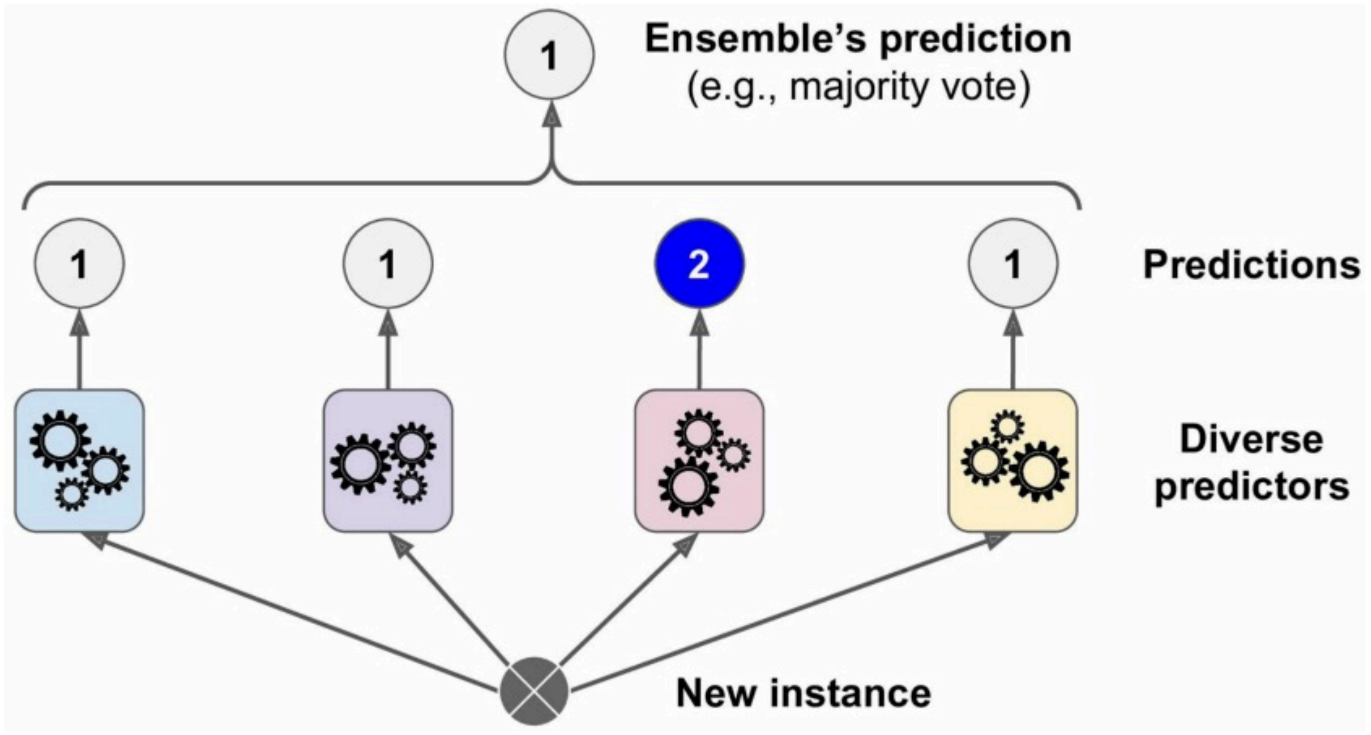


Figure 7-2. Hard voting classifier predictions

Voting Classifiers

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 (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.

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.

Voting Classifiers

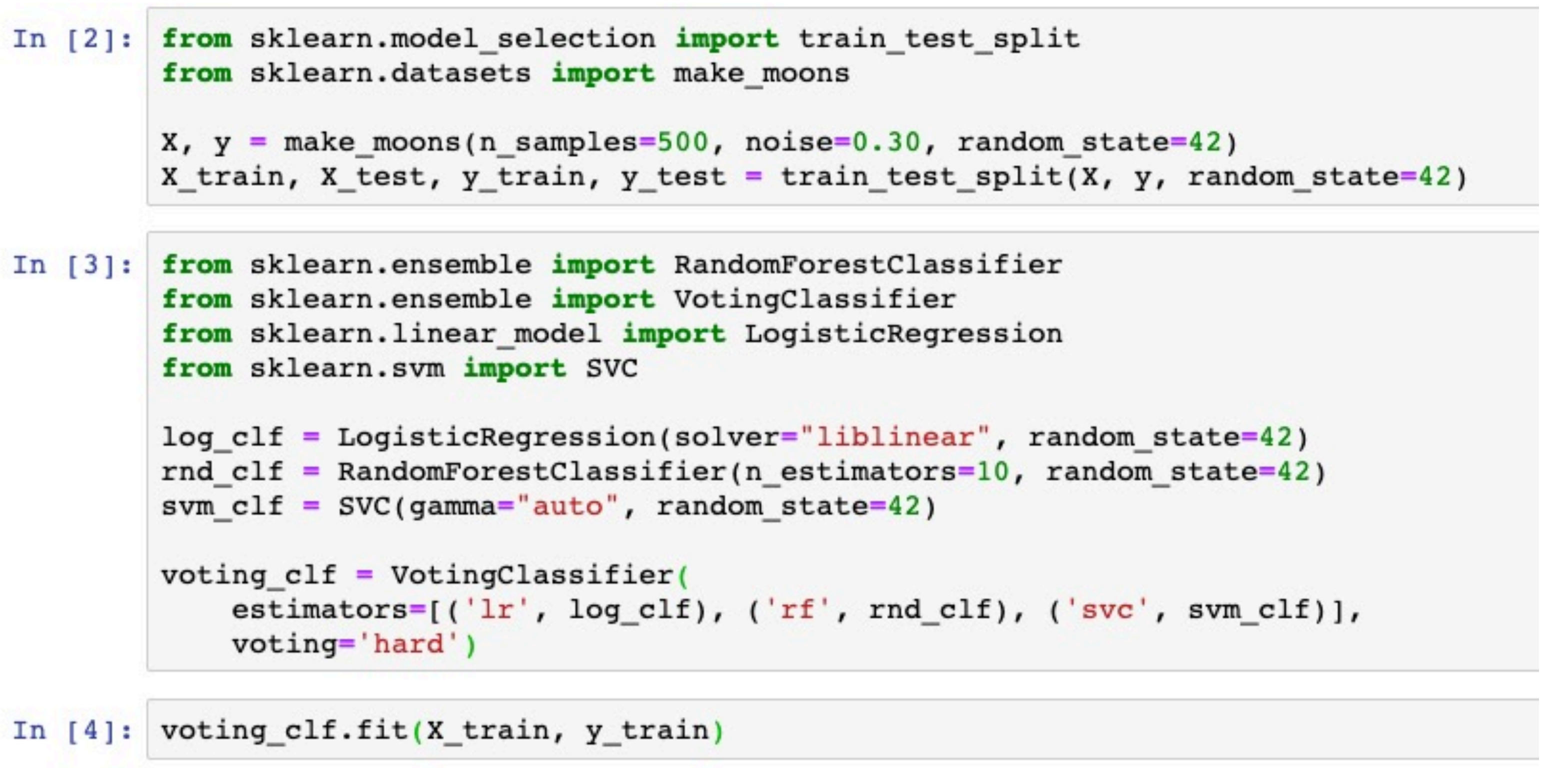
Similarly, suppose you build an ensemble containing 1,000 classifiers that are individually correct only 51% of the time (barely better than random guessing). If you predict the majority voted class, you can hope for up to 75% accuracy! However, this is only true if all classifiers are perfectly independent, making uncorrelated errors, which is clearly not the case 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.

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.

Voting Classifiers

The following code creates and trains a voting classifier in Scikit-Learn, composed of three diverse classifiers.

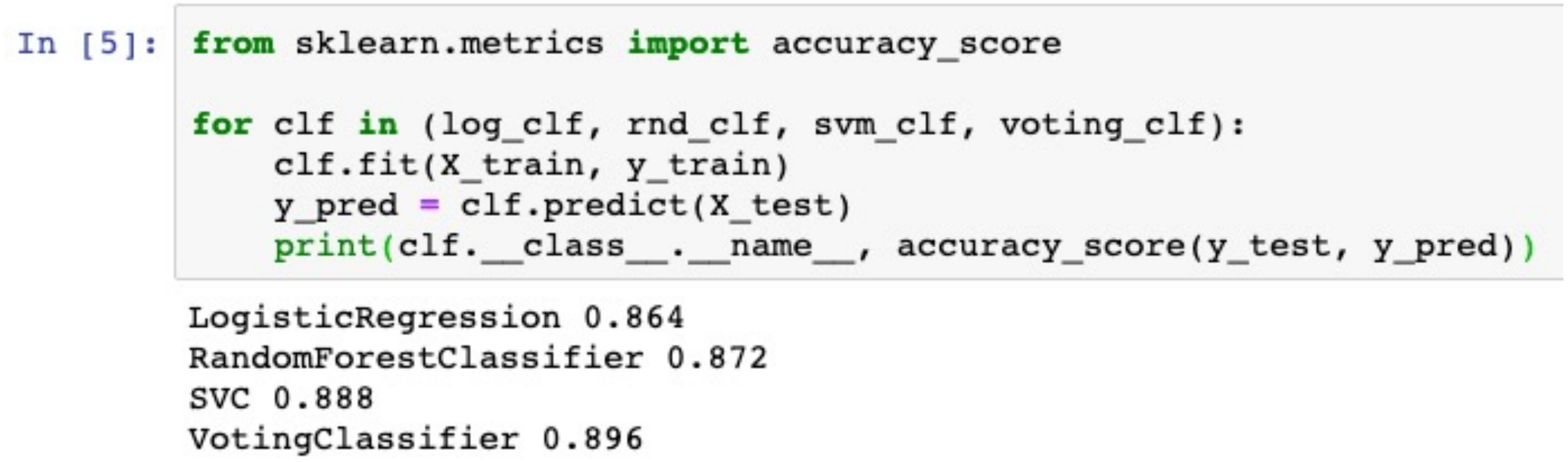
Let’ s look at each classifier’ s accuracy on the test set:



```
In [2]: from sklearn.model_selection import train_test_split
        from sklearn.datasets import make_moons

        X, y = make_moons(n_samples=500, noise=0.30, random_state=42)
        X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=42)

In [3]: from sklearn.ensemble import RandomForestClassifier
        from sklearn.ensemble import VotingClassifier
        from sklearn.linear_model import LogisticRegression
        from sklearn.svm import SVC

        log_clf = LogisticRegression(solver="liblinear", random_state=42)
        rnd_clf = RandomForestClassifier(n_estimators=10, random_state=42)
        svm_clf = SVC(gamma="auto", random_state=42)

        voting_clf = VotingClassifier(
            estimators=[('lr', log_clf), ('rf', rnd_clf), ('svc', svm_clf)],
            voting='hard')

In [4]: voting_clf.fit(X_train, y_train)
```

Voting Classifiers

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

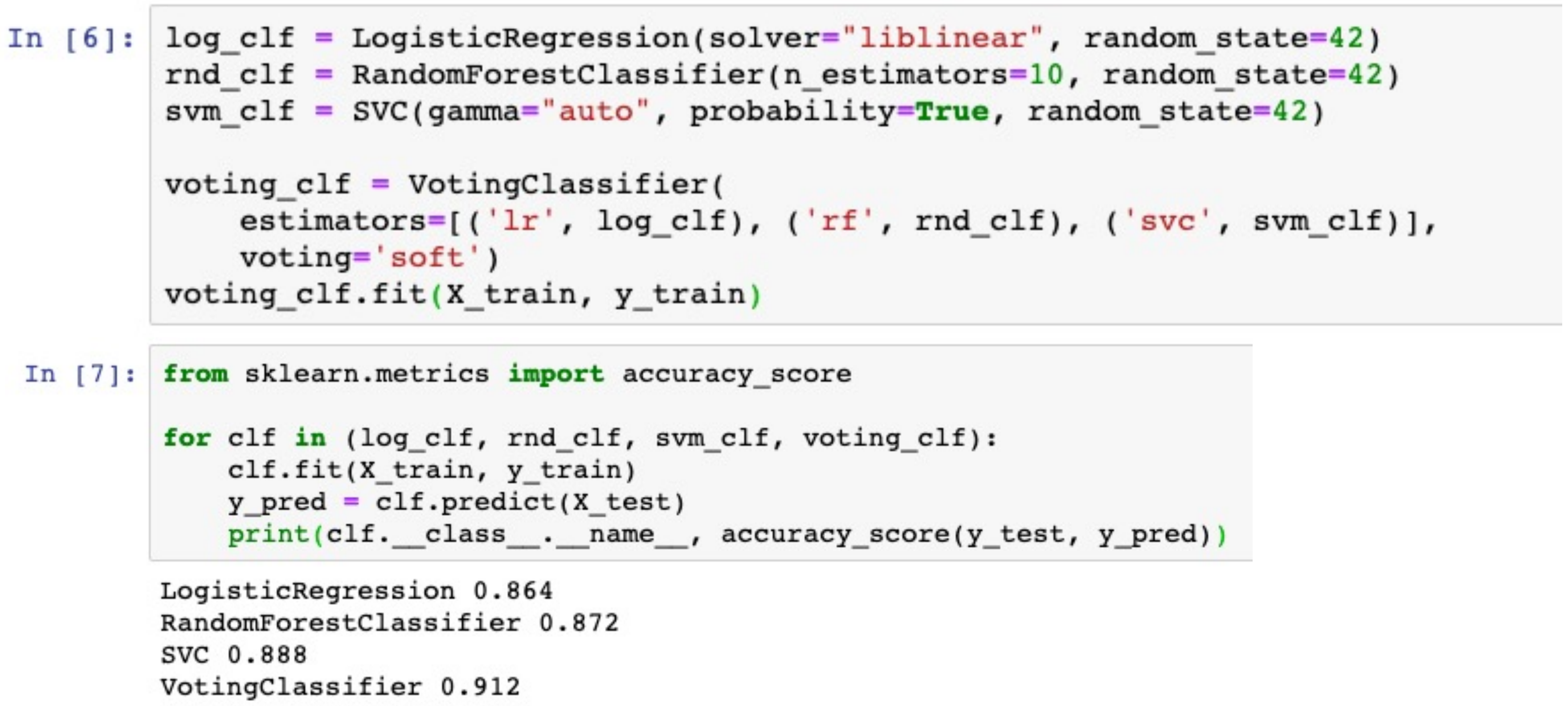


```
In [5]: from sklearn.metrics import accuracy_score

        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 0.872
SVC 0.888
VotingClassifier 0.896
```

Voting Classifiers

If all classifiers are able to estimate class probabilities (i.e., they 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.



```
In [6]: log_clf = LogisticRegression(solver="liblinear", random_state=42)
        rnd_clf = RandomForestClassifier(n_estimators=10, random_state=42)
        svm_clf = SVC(gamma="auto", probability=True, random_state=42)

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

```
In [7]: 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 0.872
SVC 0.888
VotingClassifier 0.912
```

Voting Classifiers

- **Hard Voting**
- 模型 1：A - 99%、B - 1%，表示模型 1 认为该样本是 A 类型的概率为 99%，为 B 类型的概率为 1%；

  模型1 A-99%；B-1%

  模型2 A-49%；B-51%

  模型3 A-40%；B-60% → A-两票；B-三票

  模型4 A-90%；B-10%

  模型5 A-30%；B-70%

  最终结果为B

  Hard Voting

- **Soft Voting**
- **将所有模型预测样本为某一类别的概率的平均值作为标准；**

  模型1 A-99%；B-1%

  模型2 A-49%；B-51%

  模型3 A-40%；B-60%

  模型4 A-90%；B-10%

  模型5 A-30%；B-70%

  → A - (0.99 + 0.49 + 0.4 + 0.9 + 0.3) / 5 = 0.616

  B - (0.01 + 0.51 + 0.6 + 0.1 + 0.7) / 5 = 0.384

  最终结果为A

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***(short for bootstrap aggregating).
When sampling is performed without replacement, it is called ***pasting***.

Bagging and 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.

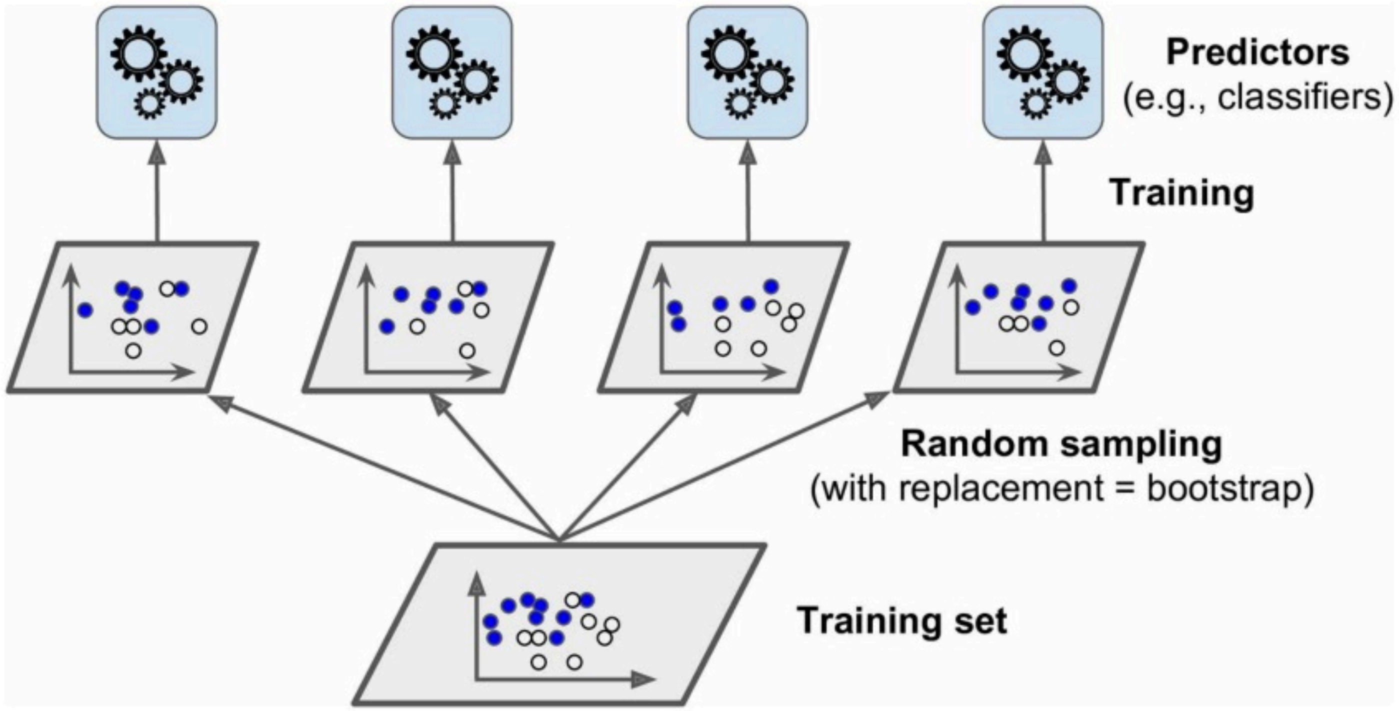


Figure 7-4. Pasting/bagging training set sampling and training

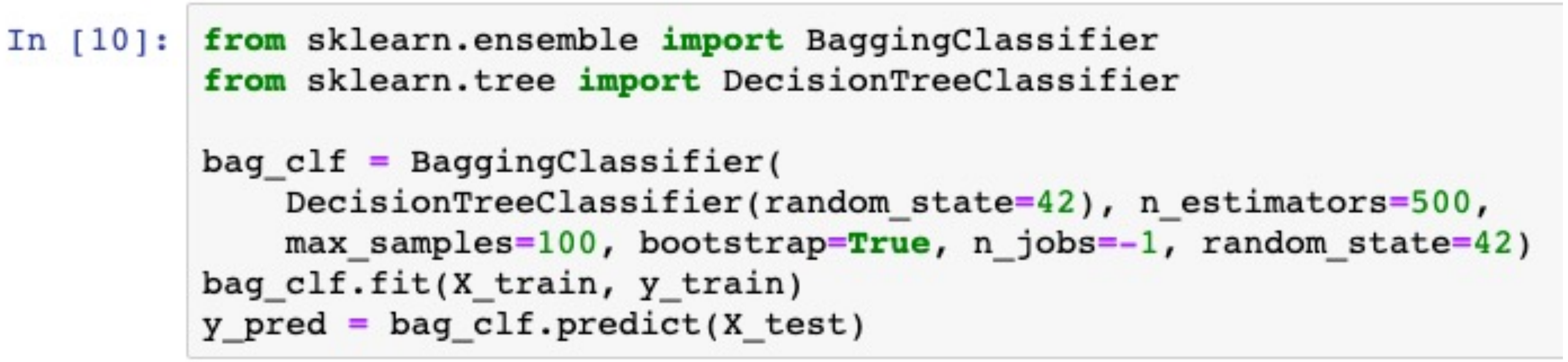
Bagging and Pasting

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 predictor has a higher bias than if it were trained on the original training set, but aggregation reduces both bias and variance. 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

Bagging and Pasting in sk-learn

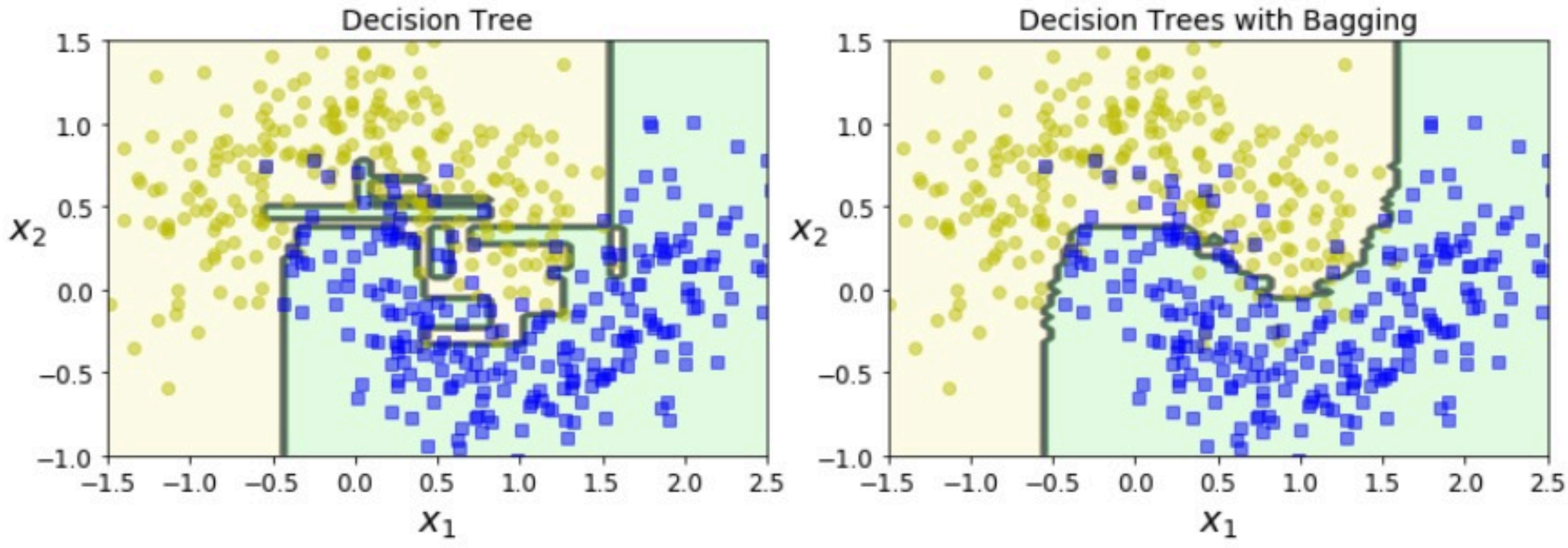
Scikit-Learn offers a simple API for both bagging and pasting with the BaggingClassifier class (or BaggingRegressor for regression). The following code trains an ensemble of 500 Decision Tree classifiers, Each trained on 100 training instances randomly sampled from the training set with replacement (this 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):



```
In [10]: from sklearn.ensemble import BaggingClassifier
         from sklearn.tree import DecisionTreeClassifier

         bag_clf = BaggingClassifier(
             DecisionTreeClassifier(random_state=42), n_estimators=500,
             max_samples=100, bootstrap=True, n_jobs=-1, random_state=42)
         bag_clf.fit(X_train, y_train)
         y_pred = bag_clf.predict(X_test)
```

Bagging and Pasting in sk-learn

Below 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)

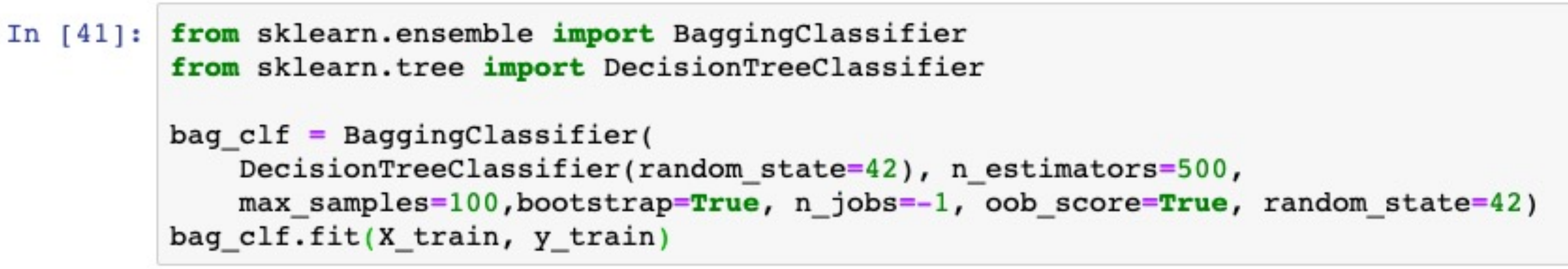


Out-of-Bag Evaluation

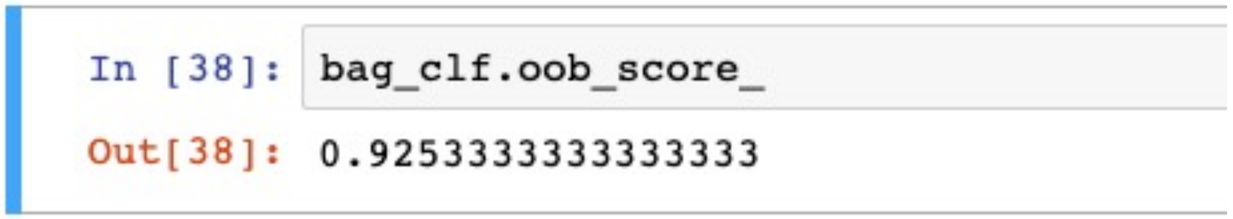
With bagging, some instances may be sampled several times for any given predictor, while others may not be sampled at all. By default a BaggingClassifier samples *m* training instances with replacement (bootstrap=True), where m is the size of the training set. This means that only about 63% of the training 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

Out-of-Bag Evaluation

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 or cross-validation. 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



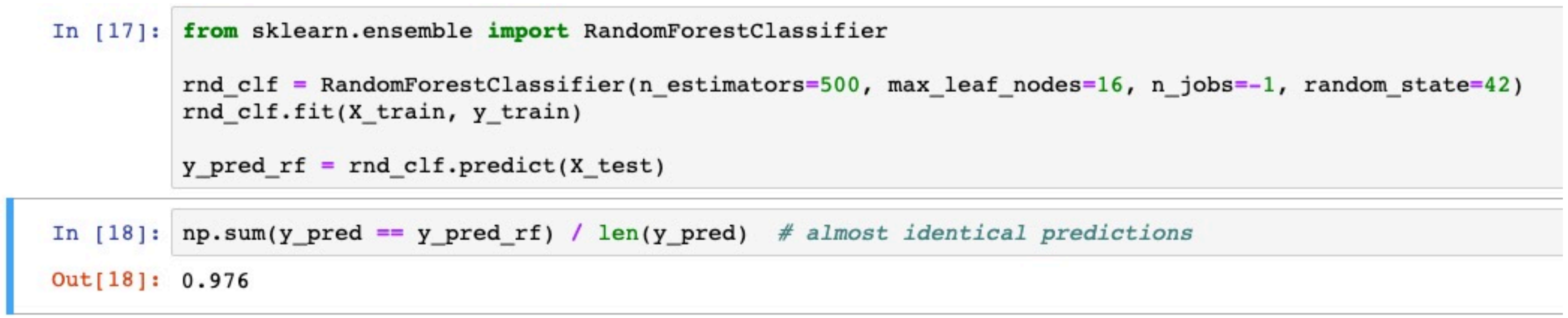
```
In [41]: from sklearn.ensemble import BaggingClassifier
         from sklearn.tree import DecisionTreeClassifier

         bag_clf = BaggingClassifier(
             DecisionTreeClassifier(random_state=42), n_estimators=500,
             max_samples=100,bootstrap=True, n_jobs=-1, oob_score=True, random_state=42)
         bag_clf.fit(X_train, y_train)
```



```
In [38]: bag_clf.oob_score_

Out[38]: 0.92533333333333
```

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



```
In [17]: from sklearn.ensemble import RandomForestClassifier

         rnd_clf = RandomForestClassifier(n_estimators=500, max_leaf_nodes=16, n_jobs=-1, random_state=42)
         rnd_clf.fit(X_train, y_train)

         y_pred_rf = rnd_clf.predict(X_test)

In [18]: np.sum(y_pred == y_pred_rf) / len(y_pred)  # almost identical predictions

Out[18]: 0.976
```

Random Forests

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

Random Forests-Bias and Variance

THE BIAS/VARIANCE TRADEOFF

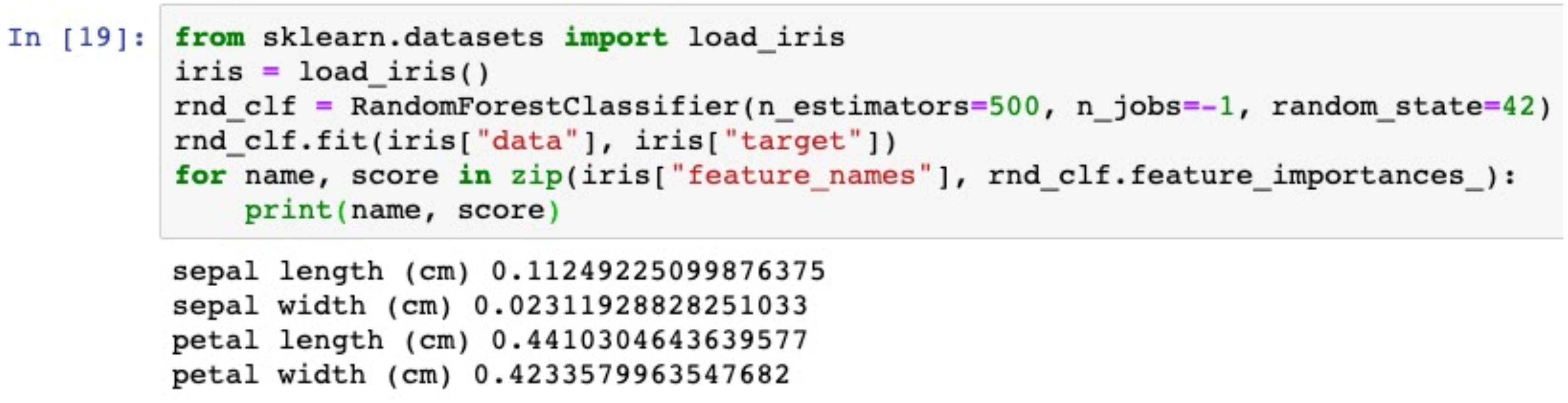
Increasing a model’ s complexity will typically increase its variance and reduce its bias. Conversely, reducing a model’ s complexity increases its bias and reduces its variance. This is why it is called a tradeoff

偏差-方差分解与bagging减少方差，boosting减少偏差
https://cloud.tencent.com/developer/article/1483259

Random Forests - Extra-Trees

Lastly, if you look at a single Decision Tree, important features are likely to appear closer to the root of the tree, while unimportant features will often appear closer to the leaves (or not at all). It is therefore possible to get an estimate of a feature’s importance by computing the average depth at which it appears across all trees in the forest.

Scikit-Learn computes this automatically for every feature after training. 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

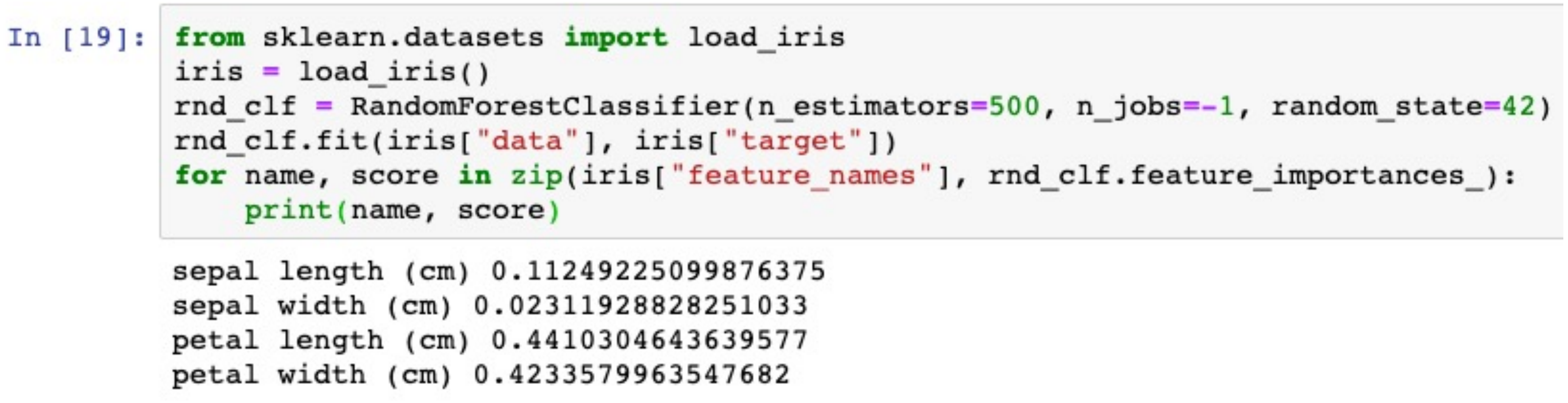


```
In [19]: from sklearn.datasets import load_iris
         iris = load_iris()
         rnd_clf = RandomForestClassifier(n_estimators=500, n_jobs=-1, random_state=42)
         rnd_clf.fit(iris["data"], iris["target"])
         for name, score in zip(iris["feature_names"], rnd_clf.feature_importances_):
             print(name, score)

         sepal length (cm) 0.11249225099876375
         sepal width (cm) 0.02311928828251033
         petal length (cm) 0.4410304643639577
         petal width (cm) 0.4233579963547682
```

Random Forests - Extra-Trees

Lastly, if you look at a single Decision Tree, important features are likely to appear closer to the root of the tree, while unimportant features will often appear closer to the leaves (or not at all). It is therefore possible to get an estimate of a feature’ s importance by computing the average depth at which it appears across all trees in the forest.

Scikit-Learn computes this automatically for every feature after training. 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



```
In [19]: from sklearn.datasets import load_iris
         iris = load_iris()
         rnd_clf = RandomForestClassifier(n_estimators=500, n_jobs=-1, random_state=42)
         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) 0.11249225099876375
         sepal width (cm) 0.02311928828251033
         petal length (cm) 0.4410304643639577
         petal width (cm) 0.4233579963547682
```

Boosting

Boosting (originally called hypothesis 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) and Gradient Boosting. Let’ s start with AdaBoost.

Boosting - 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 predictors focusing more and more on the hard cases. This is the technique used by AdaBoost

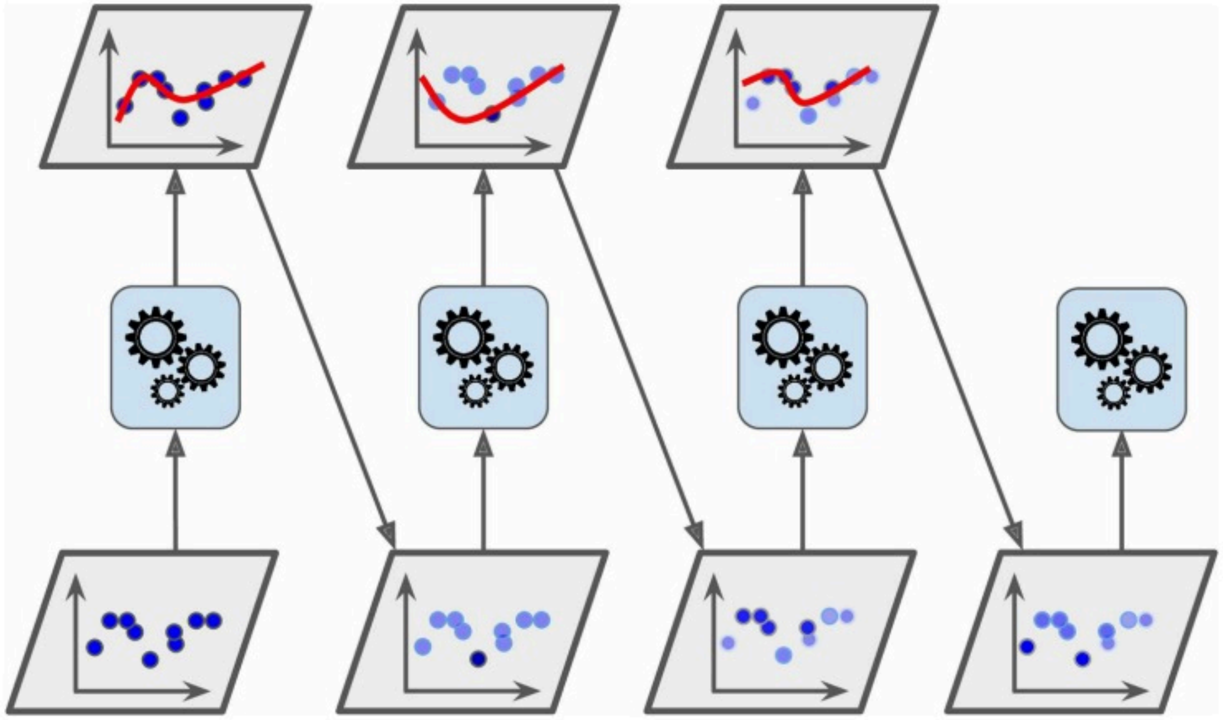
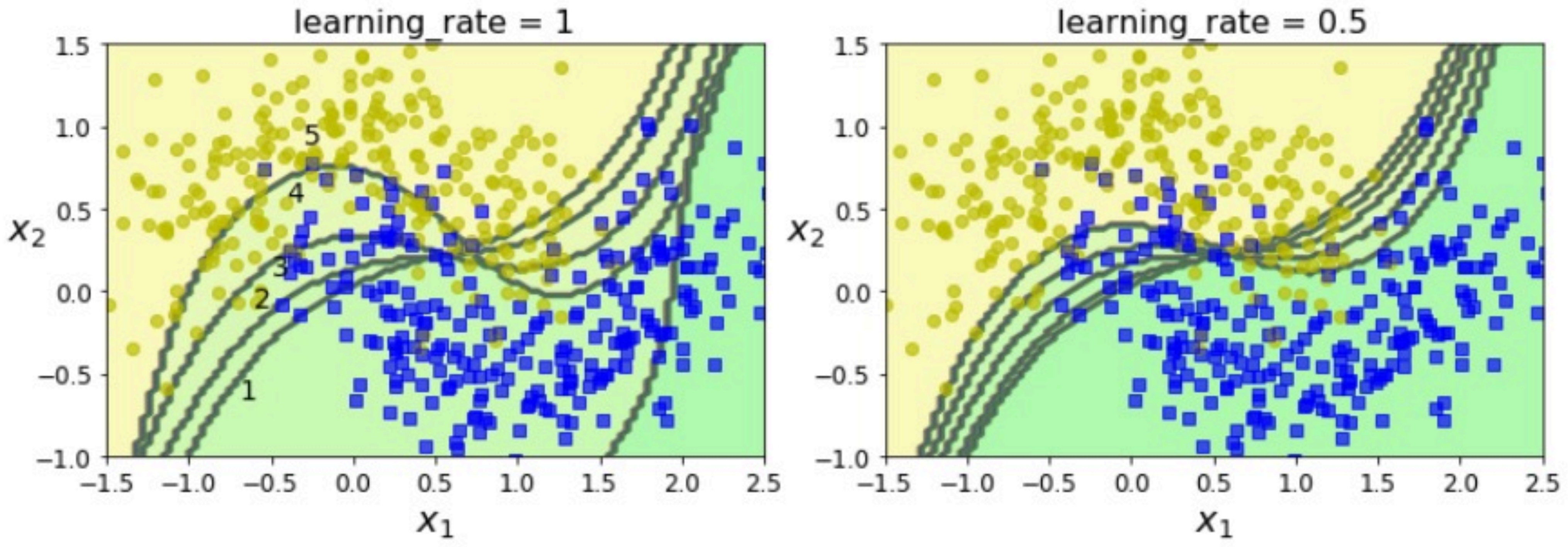


Figure 7-7. AdaBoost sequential training with instance weight updates

Boosting - AdaBoost

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).



Boosting - AdaBoost

Once all predictors are trained, the ensemble makes predictions very much like bagging 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 technique: it cannot be parallelized (or only partially), since each predictor can only be trained after the previous predictor has been trained and evaluated. As a result, it does not scale as well as bagging or pasting.

Boosting - AdaBoost

Let’ s take a closer look at the AdaBoost algorithm

1. Each instance weight w(i)is initially set to 1/m.
2. A first predictor is trained and its weighted error rate r is computed on the training set

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

$$r_j = \frac{\displaystyle\sum_{\substack{i=1 \\ \hat{y}_j^{(i)} \neq y^{(i)}}}^{m} w^{(i)}}{\displaystyle\sum_{i=1}^{m} w^{(i)}} \quad \text{where } \hat{y}_j^{(i)} \text{ is the } j^{\text{th}} \text{ predictorâ}\square\square\text{s prediction for the } i^{\text{th}} \text{ instance.}$$

3. The predictor’ s weight αj is then computed below , where η is the learning rate hyperparameter (defaults to 1)

Equation 7-2. Predictor weight

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

Boosting - AdaBoost

Let’ s take a closer look at the AdaBoost algorithm

4. Next the instance weights are updated using Equation below: the misclassified instances are boosted. Then all the instance weights are normalized.

Equation 7-3. Weight update rule

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

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

5. 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)
6. The algorithm stops when the desired number of predictors is reached, or when a perfect predictor is found.

Boosting - AdaBoost

Let’ s take a closer look at the AdaBoost algorithm:

7. 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.

Equation 7-4. AdaBoost predictions

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

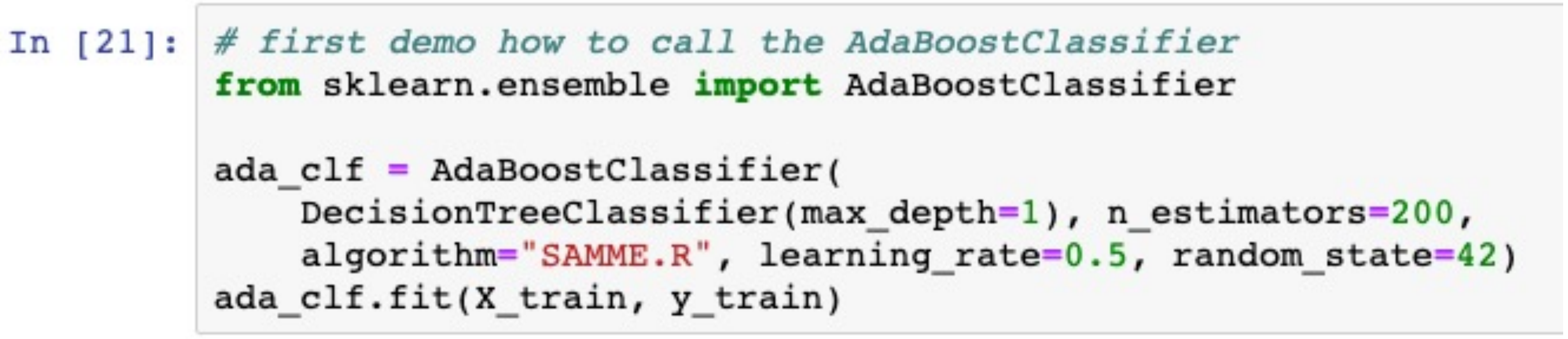
Boosting - AdaBoost

Scikit-Learn actually uses a multiclass version of AdaBoost called SAMME (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.

Boosting - AdaBoost

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:



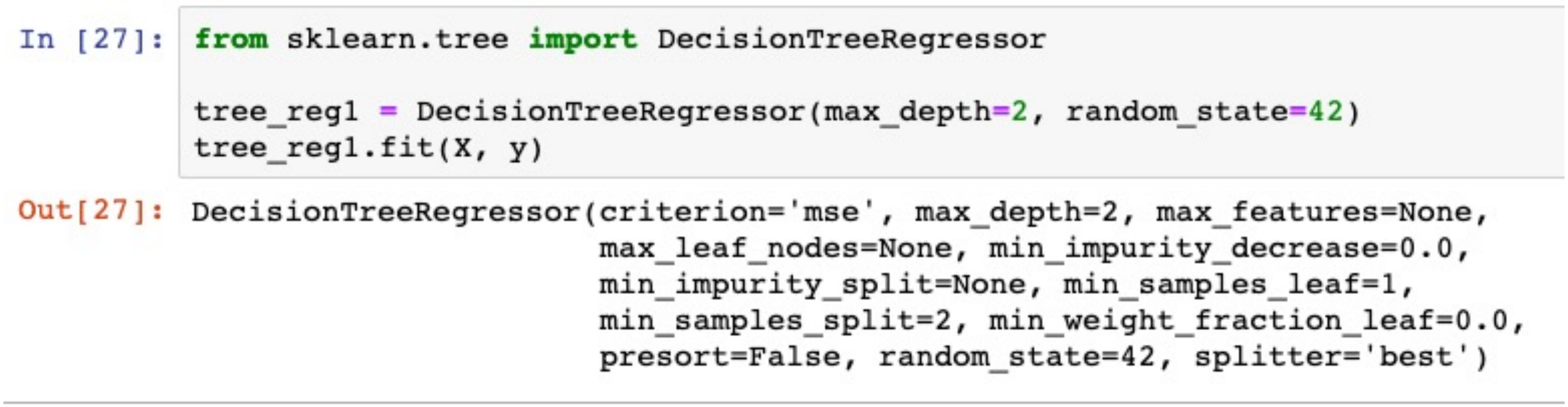
```
In [21]: # first demo how to call the AdaBoostClassifier
         from sklearn.ensemble import AdaBoostClassifier

         ada_clf = AdaBoostClassifier(
             DecisionTreeClassifier(max_depth=1), n_estimators=200,
             algorithm="SAMME.R", learning_rate=0.5, random_state=42)
         ada_clf.fit(X_train, y_train)
```

Boosting - Gradient Boosting

Another very popular Boosting algorithm is Gradient Boosting.
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.

Boosting - Gradient Boosting

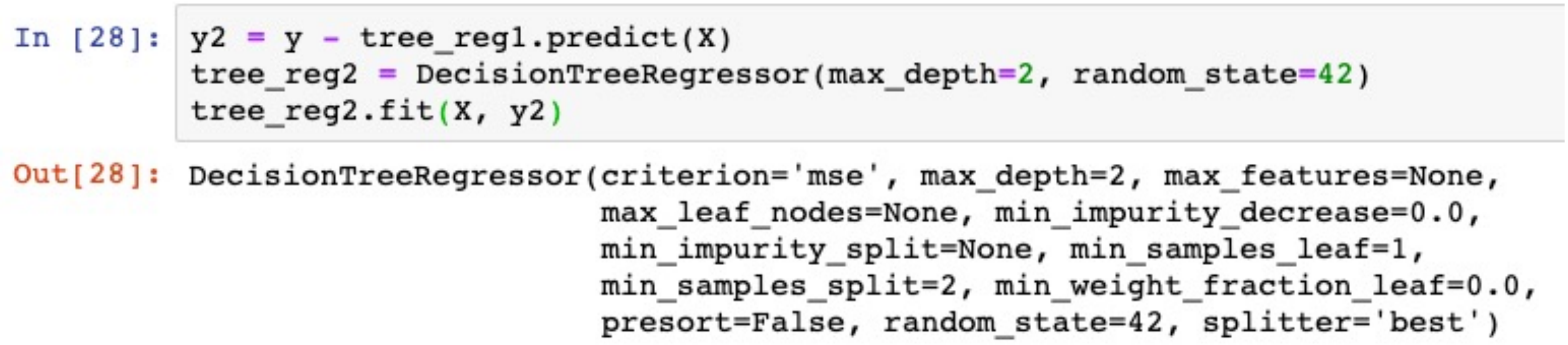
Let’ s go through a simple regression example using Decision Trees as the base predictors (of course Gradient Boosting also works great with regression tasks). **This is called Gradient Tree Boosting, or Gradient Boosted Regression Trees (GBRT).** First, let’ s fit a DecisionTreeRegressor to the training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



```
In [27]: from sklearn.tree import DecisionTreeRegressor

         tree_reg1 = DecisionTreeRegressor(max_depth=2, random_state=42)
         tree_reg1.fit(X, y)

Out[27]: DecisionTreeRegressor(criterion='mse', max_depth=2, max_features=None,
                    max_leaf_nodes=None, min_impurity_decrease=0.0,
                    min_impurity_split=None, min_samples_leaf=1,
                    min_samples_split=2, min_weight_fraction_leaf=0.0,
                    presort=False, random_state=42, splitter='best')
```

Boosting - Gradient Boosting

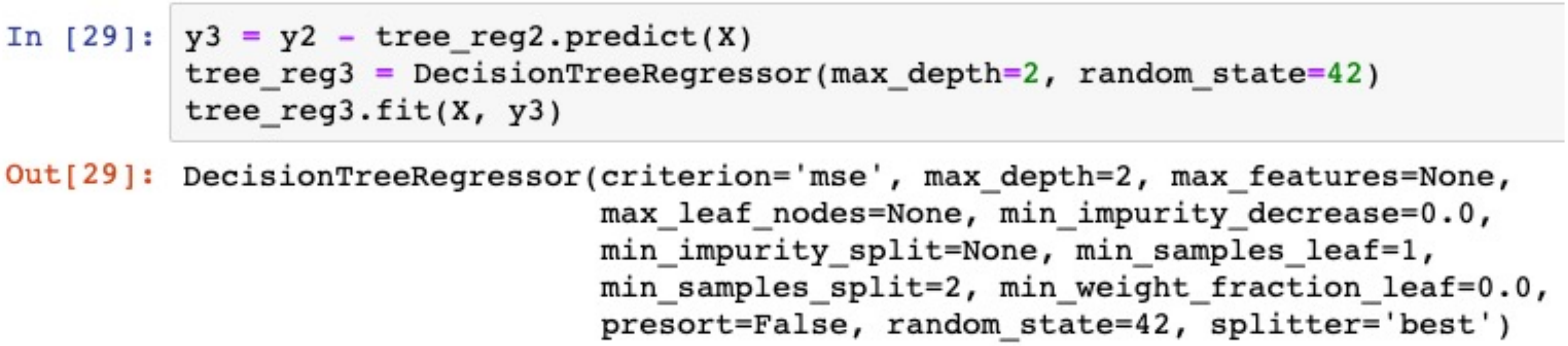
Now train a second DecisionTreeRegressor on the residual errors made by the first predictor:



```
In [28]: y2 = y - tree_reg1.predict(X)
         tree_reg2 = DecisionTreeRegressor(max_depth=2, random_state=42)
         tree_reg2.fit(X, y2)

Out[28]: DecisionTreeRegressor(criterion='mse', max_depth=2, max_features=None,
                               max_leaf_nodes=None, min_impurity_decrease=0.0,
                               min_impurity_split=None, min_samples_leaf=1,
                               min_samples_split=2, min_weight_fraction_leaf=0.0,
                               presort=False, random_state=42, splitter='best')
```

Boosting - Gradient Boosting

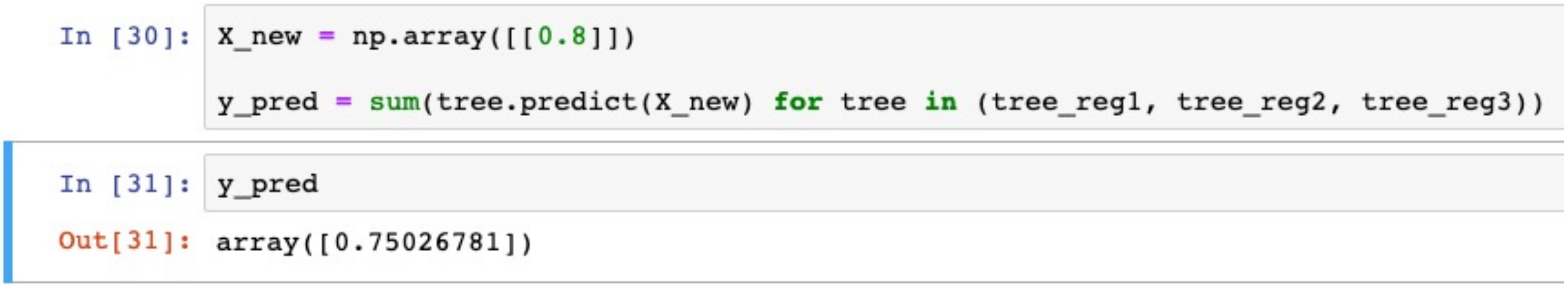
Then we train a third regressor on the residual errors made by the second Predictor :



```
In [29]: y3 = y2 - tree_reg2.predict(X)
         tree_reg3 = DecisionTreeRegressor(max_depth=2, random_state=42)
         tree_reg3.fit(X, y3)

Out[29]: DecisionTreeRegressor(criterion='mse', max_depth=2, max_features=None,
                    max_leaf_nodes=None, min_impurity_decrease=0.0,
                    min_impurity_split=None, min_samples_leaf=1,
                    min_samples_split=2, min_weight_fraction_leaf=0.0,
                    presort=False, random_state=42, splitter='best')
```

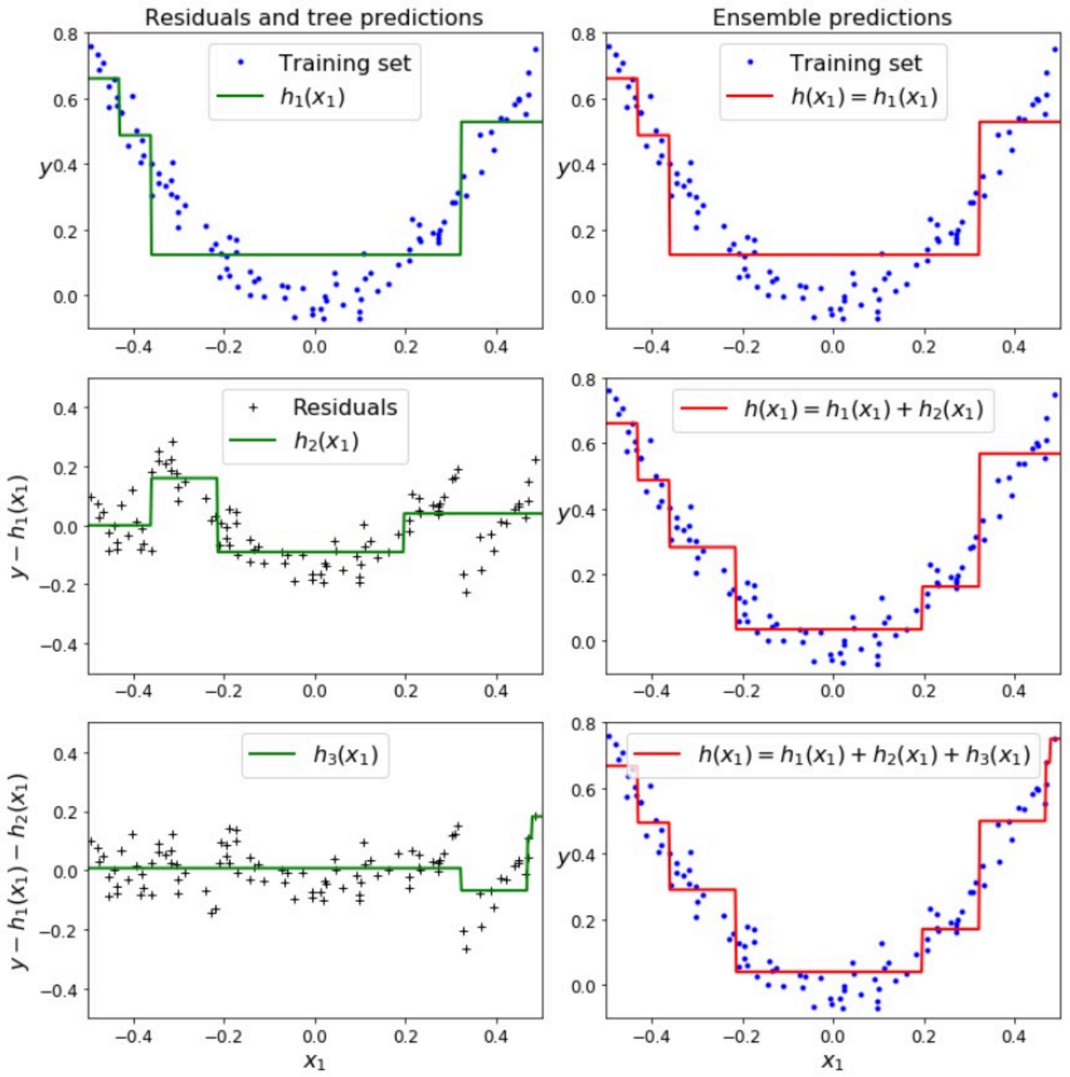
Boosting - Gradient Boosting

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 :



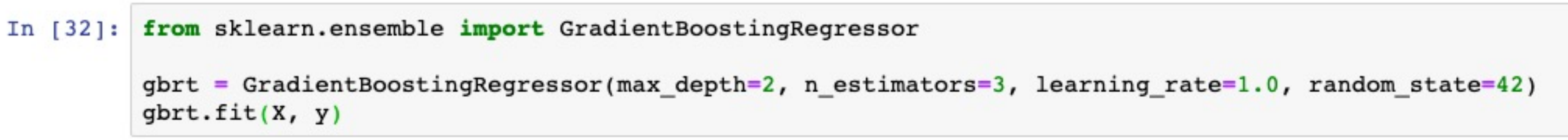
```
In [30]: X_new = np.array([[0.8]])

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

In [31]: y_pred

Out[31]: array([0.75026781])
```

Boosting - Gradient Boosting



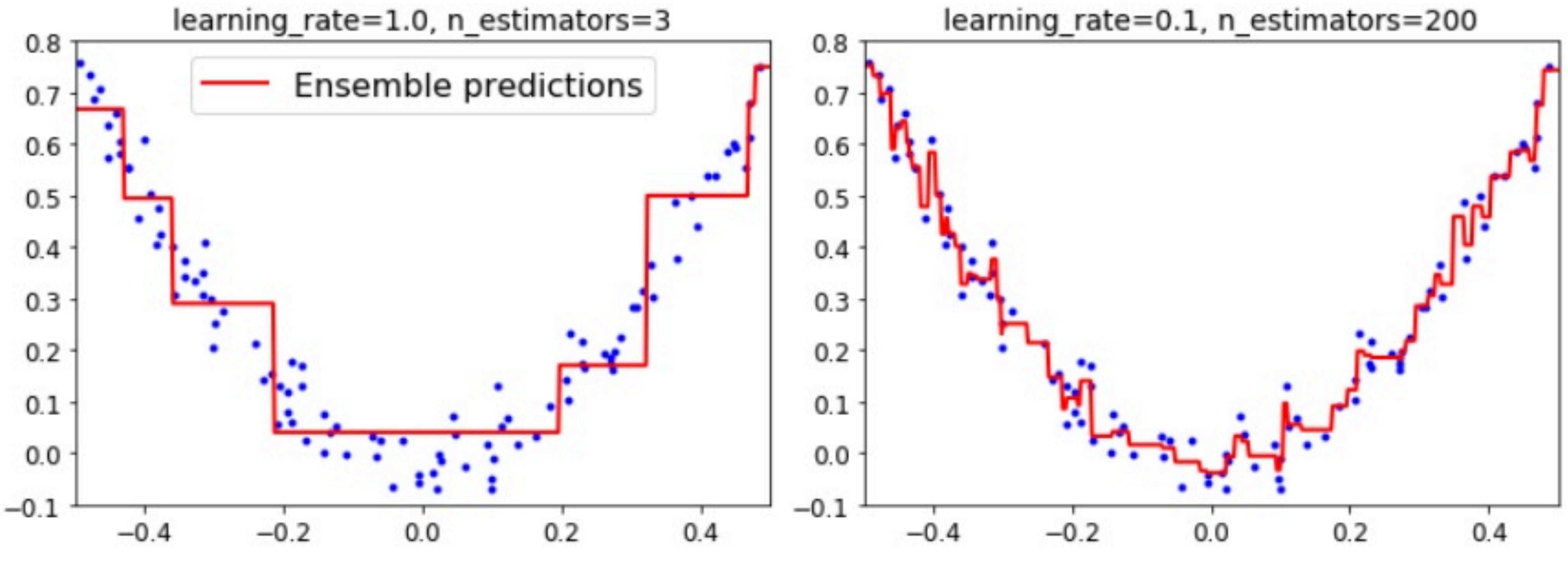
Boosting - Gradient Boosting

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, 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


```
In [32]: from sklearn.ensemble import GradientBoostingRegressor

         gbrt = GradientBoostingRegressor(max_depth=2, n_estimators=3, learning_rate=1.0, random_state=42)
         gbrt.fit(X, y)
```

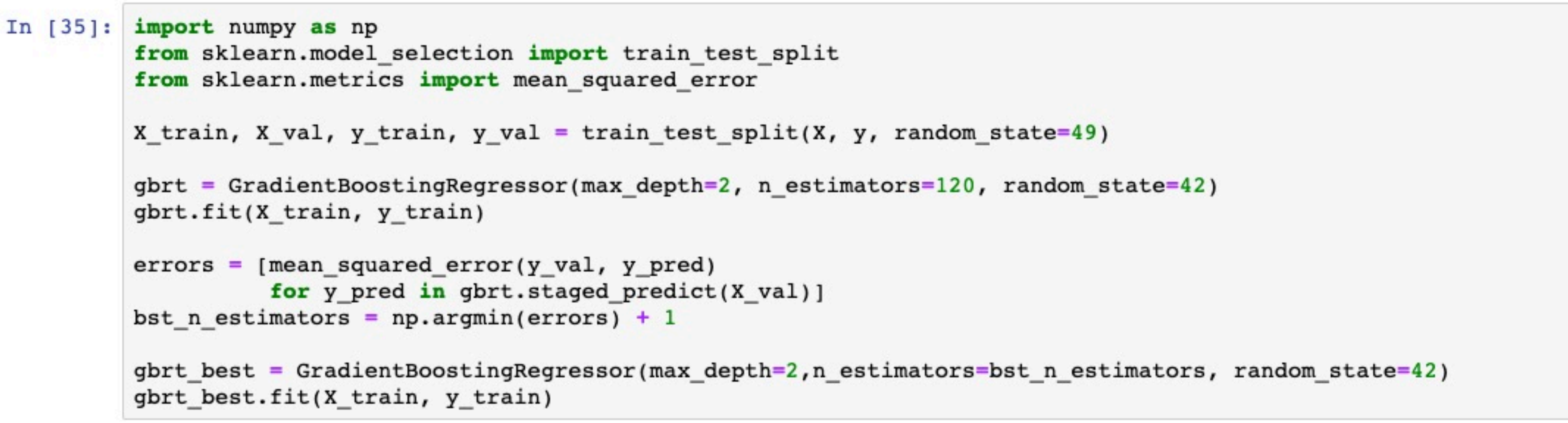
Boosting - Gradient Boosting

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 training set, but the predictions will usually generalize better.



Boosting - Gradient Boosting

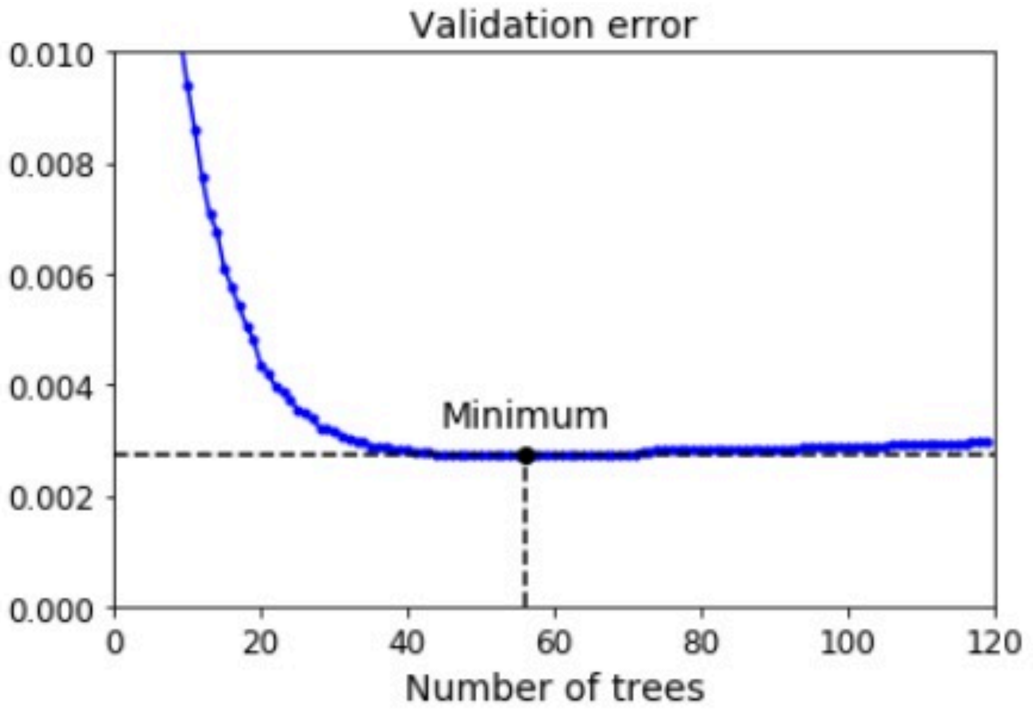
In order to find the optimal number of trees, you can use early stopping. A simple way to implement this is to use the staged_predict() method: it returns an iterator over the predictions made by the ensemble at each stage of training (with one tree, two trees, etc.). The following code trains a GBRT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

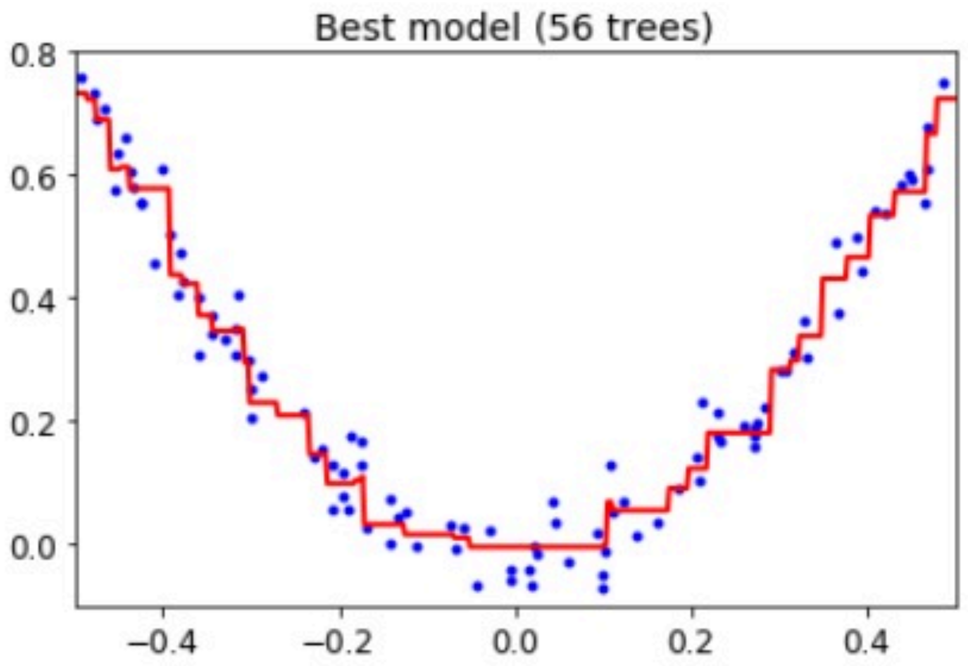


```
In [35]: import numpy as np
         from sklearn.model_selection import train_test_split
         from sklearn.metrics import mean_squared_error

         X_train, X_val, y_train, y_val = train_test_split(X, y, random_state=49)

         gbrt = GradientBoostingRegressor(max_depth=2, n_estimators=120, random_state=42)
         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, random_state=42)
         gbrt_best.fit(X_train, y_train)
```

Boosting - Gradient Boosting

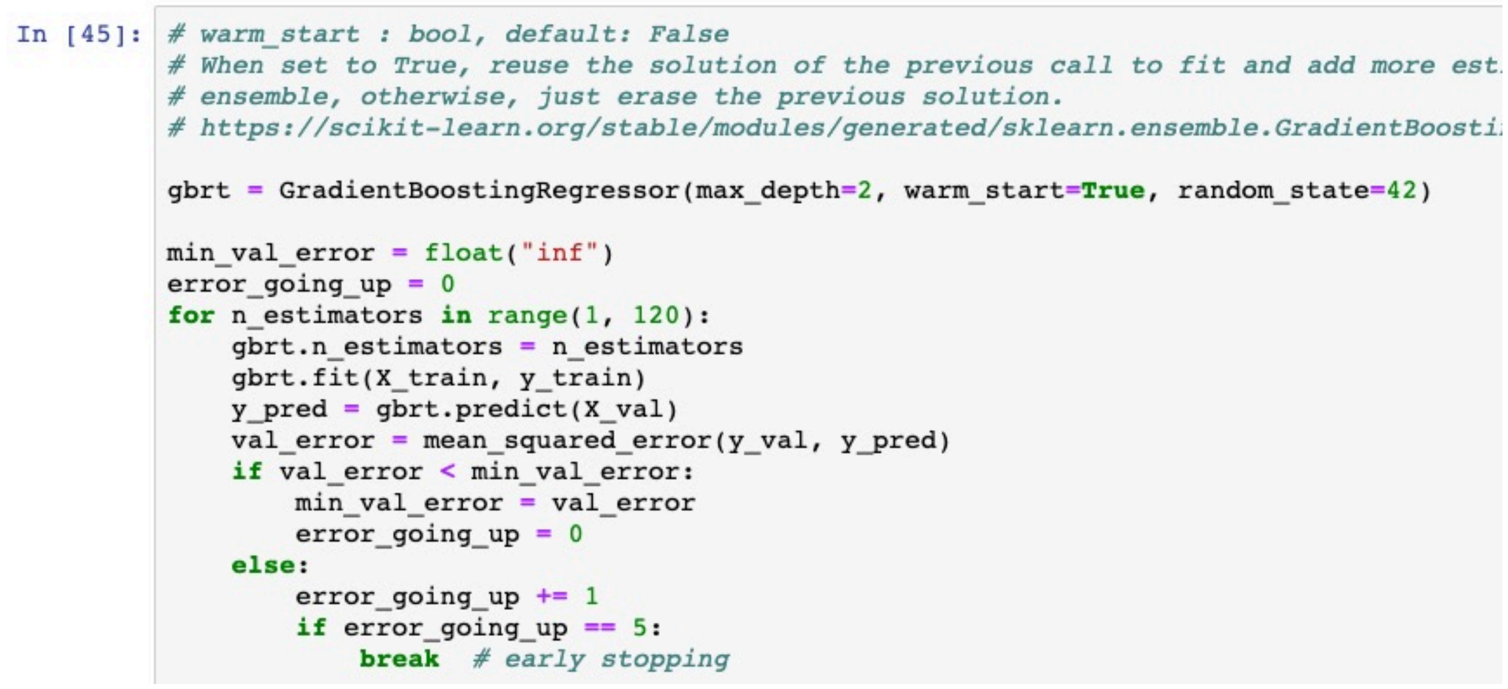
The validation errors are represented on the left, and the best model’ s predictions are represented on the right





Boosting - Gradient Boosting

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:



```
In [45]: # warm_start : bool, default: False
         # When set to True, reuse the solution of the previous call to fit and add more est
         # ensemble, otherwise, just erase the previous solution.
         # https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.GradientBoosti

         gbrt = GradientBoostingRegressor(max_depth=2, warm_start=True, random_state=42)

         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
```

Boosting - Gradient Boosting

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 instances, 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.

Stacking

The last Ensemble method we will discuss in this chapter is called stacking (short for stacked generalization). It is 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?

Stacking

Below 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)

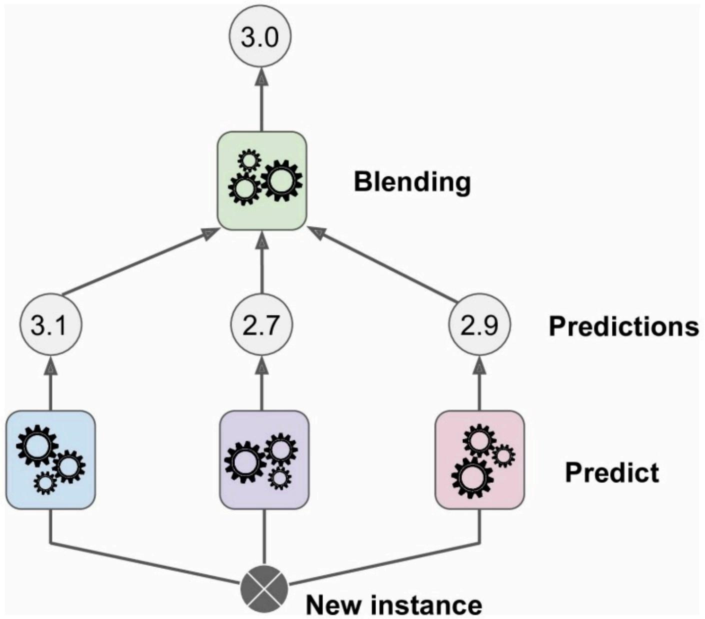


Figure 7-12. Aggregating predictions using a blending predictor

Stacking

To train the blender, a common approach is to use a hold-out set

First, the training set is split in two subsets. The first subset is used to train the predictors in the first layer
Next, the first layer predictors are used to make predictions on the second (held-out) set. Now for each instance in the hold-out set there are three predicted values. We can create a new training set using these predicted values as input features (which makes this new training set three-dimensional), and keeping the target values.
Then The blender is trained on this new training set, so it learns to predict the target value given the first layer’ s predictions.

Stacking

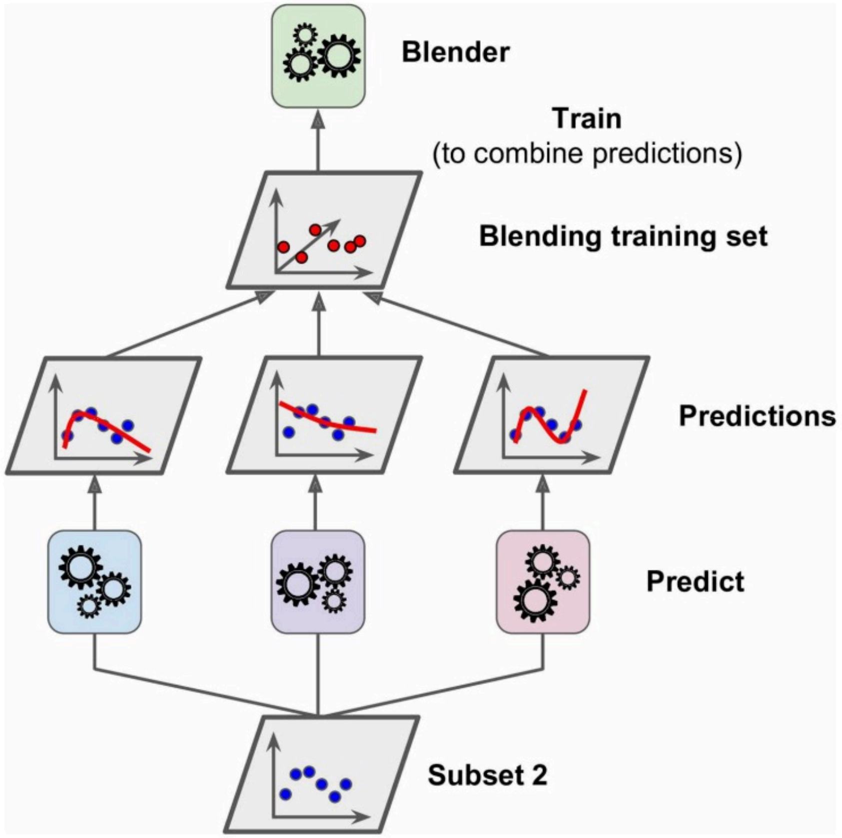


Figure 7-14. Training the blender

Stacking

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/viisar/brew)

SAIF

Shanghai Advanced

Institute of Finance

上海高级金融学院