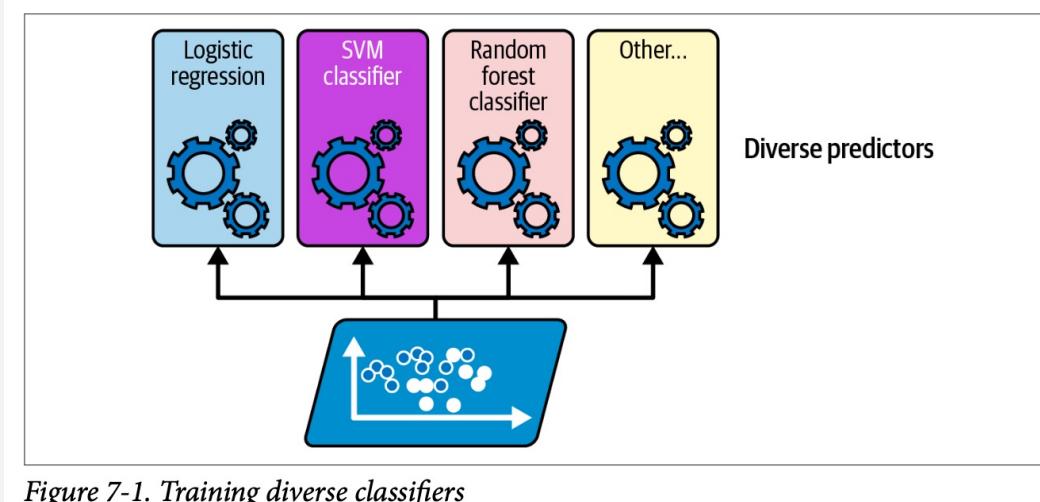


به نام خدا
خلاصه فصل ۷
کتاب

**HANDS-ON
MACHINE
LEARNING**

ENSEMBLE LEARNING AND RANDOM FORESTS

- اگه بیايم يه مطلب رو از عموم بپرسيم احتمالا نتیجه بهتری داره نسبت به اينکه نظر صرفا يه متخصص رو بپرسيم. به اين ميگن خرد جمعی. همین قضيه هم تو مدل های machine learning صدق ميکنه. اگه بیايم پيش بینی کلی مدل رو کنار هم بذاريم و بر اساس اون پيش بینی کنيم ميشه ensemble method. يه راه برای پياده سازی اين کار روی decision trees ميتوانيم يه گروهي از درخت هارو روی فيچر های رندوم train یا زيرمجموعه های متفاوت training set، train کنيم و بر اساس اينکه بيشترین پيش بینی روی تک تک درخت ها چيye پيش بینی کنيم که بهش ميگن random forest. اين مدل يکی از قوي ترین مدل های machine learning تا الان هستش.
- بیايم فرض کنيم چند تا مدل داریم که هرکدام درصد دقت 80 دارن.



CONTINUE

- یه راه ساده برای اینکه بیايم یک classifier بهتر بکنیم اینه که تمامی پیش‌بینی مدل‌های مختلف رو جمع آوری کنیم و اون کلاسی که بیشترین پیش‌بینی انجام شده داخل تک تک مدل‌ها رو به عنوان خروجی نهایی بدیم بیرون. به این میگن hard voting classifier.

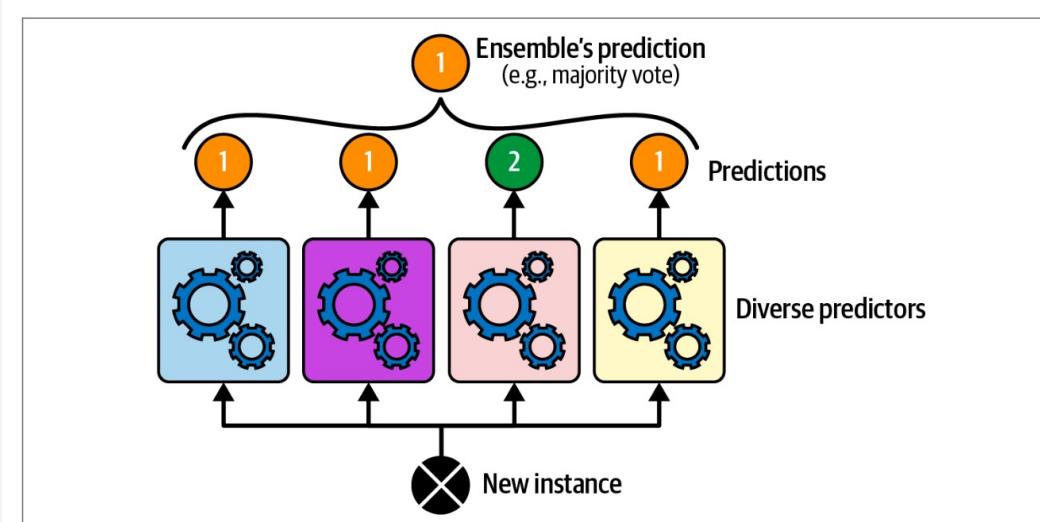


Figure 7-2. Hard voting classifier predictions

- نکته جالب اینه با این کار دقتمون به بیشتر از ۸۰ درصد میرسه.

CONTINUE

- اما چطوری همچین چیزی واقعیت داره؟ در واقع پشت این یه حقیقت آمار احتمالی خوابیده که اگه تعداد خیلی زیادی از مدل با دقت ۵۰ درصد داشته باشیم و همچون از هم مستقل باشن (که معمولاً خیلی پیش نمیاد) چون همپوشانی خوبی روی کلاس درست دارن تو اکثر موقع تا ۷۵ درصد (که معمولاً بخاطر نقض شرط بالا عدش کوچیکتر از این هست) دقت داریم. یه راه برای این که مدل هامون رو مستقل از هم تشکیل بدیم اینه که با الگوریتم های مختلف مدل هامون رو train کنیم.

```
from sklearn.datasets import make_moons
from sklearn.ensemble import RandomForestClassifier, VotingClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.svm import SVC

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)

voting_clf = VotingClassifier(
    estimators=[
        ('lr', LogisticRegression(random_state=42)),
        ('rf', RandomForestClassifier(random_state=42)),
        ('svc', SVC(random_state=42))
    ]
)
voting_clf.fit(X_train, y_train)
```

CONTINUE

- به این شکل هم میتوانیم بایام دقت تک تک مدل هارو بررسی کنیم:

```
>>> for name, clf in voting_clf.named_estimators_.items():
...     print(name, "=", clf.score(X_test, y_test))
...
lr = 0.864
rf = 0.896
svc = 0.896
```

- همچنین میتوانیم پیش‌بینی مدل‌های زیرمدل را ببینیم و همچنین میزان دقت مدل کلیمون:

```
>>> voting_clf.predict(X_test[:1])
array([1])
>>> [clf.predict(X_test[:1]) for clf in voting_clf.estimators_]
[array([1]), array([1]), array([0])]
```

Now let's look at the performance of the voting classifier on the test set:

```
>>> voting_clf.score(X_test, y_test)
0.912
```

SOFT VOTING

- اگه مدلمون این شکلی باشه که از يه سري زيرمدل تشکيل شده باشه که توانايي اينو داشته باشن برای هر نمونه احتمال عضويت به يه کلاس رو حساب کنن ميتوnim با ميانگين گيري از همه بيايم خروجي بدیم که به مراتب از مدل قبلی بهتره چون با درصد اطمینان بيشتری داریم رو مدلامون تصمیم گيري ميکنیم و از حالت ° و ۱ خارج شده.

fier's voting hyperparameter to "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 hyperparameter 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). Let's try that:

```
>>> voting_clf.voting = "soft"
>>> voting_clf.named_estimators["svc"].probability = True
>>> voting_clf.fit(X_train, y_train)
>>> voting_clf.score(X_test, y_test)
0.92
```

We reach 92% accuracy simply by using soft voting—not bad!

BAGGING AND PASTING

- یه راه این بود چند تا الگوریتم مختلف امتحان کنیم. اما میشه یه الگوریتم train کنیم و برای این کار میتونیم از زیر مجموعه های متفاوتی برای هر مدل استفاده کنیم. اگه نمونه هایی که برمیداریم برای یه مدل قابل تکرار در زیرمجموعه رو داشته باشه و عضو مجموعه بشه دوباره، به این روش bagging و اگه توانایی تکرار رو نداشته باشه pasting میگیم.

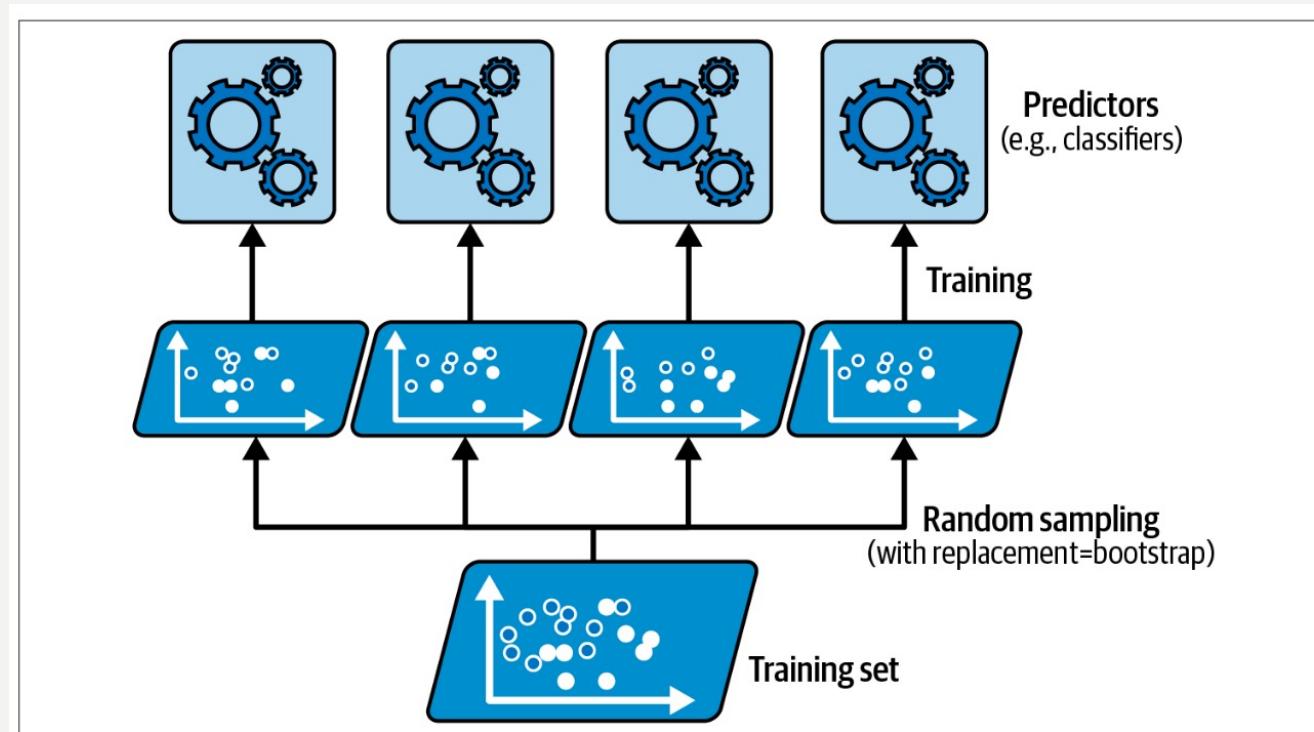


Figure 7-4. Bagging and pasting involve training several predictors on different random samples of the training set

CONTINUE

- وقتی که همه مدل هارو train کردیم. با ensemble کردن میتوانیم برای یه نمونه جدید پیش بینی داشته باشیم. به این شکل که برای classificaiton از دو حالت قبل میتوانیم کمک بگیریم و برای regression میتوانیم بیايم از میانگین اعداد خروجی استفاده کنیم. هر مدل به تنهايی که train کردیم چون یه زیرمجموعه از training set دارای bias بیشتری از حالتیه که روی کل variance میومدیم این حرکتو میزدیم. اما این حرکتی که میزنیم باعث میشه هم bias کم بشه.

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. This sampling and training process is represented in [Figure 7-4](#).

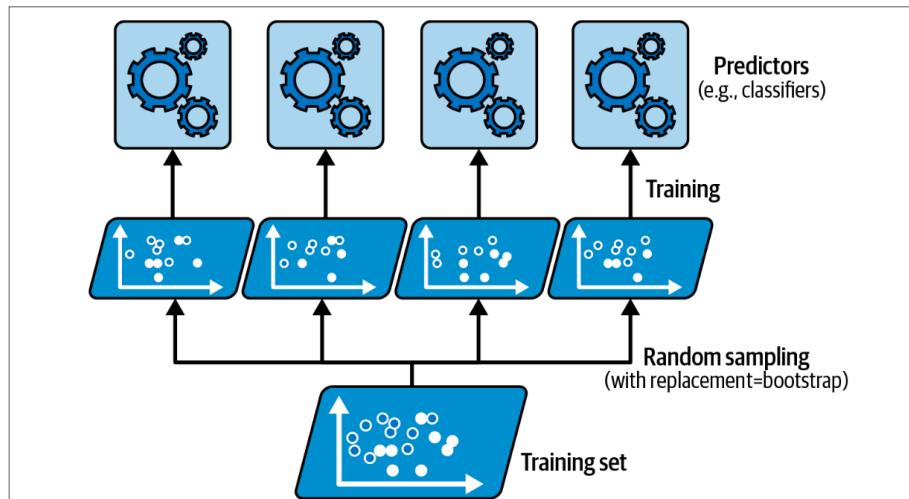


Figure 7-4. Bagging and pasting involve training several predictors on different random samples of the training set

CONTINUE

- مدل زیر میاد 500 تا train، decision tree میکنه. هر کدوم روی ۱۰۰ تا سمپل دیتابی. یه نکته دیگه ای که هست میتونیم مدل هارو به شکل اینارو parallel train کنه برای همین n_jobs که میاد تعداد ۱ بینی از همه core cpu استفاده parallel train کردن کردن برای ۵۰۰ میکنیم. ۱ یعنی از همه core ها استفاده کن.

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

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

همونطور که میدونیم این ensemble کردن باعث میشه احتمال overfit کردن رو خیلی کمتر کنیم و این یعنی generalize مدل نهاییم خیلی بیشتر از تک تک مدل ها میشه.

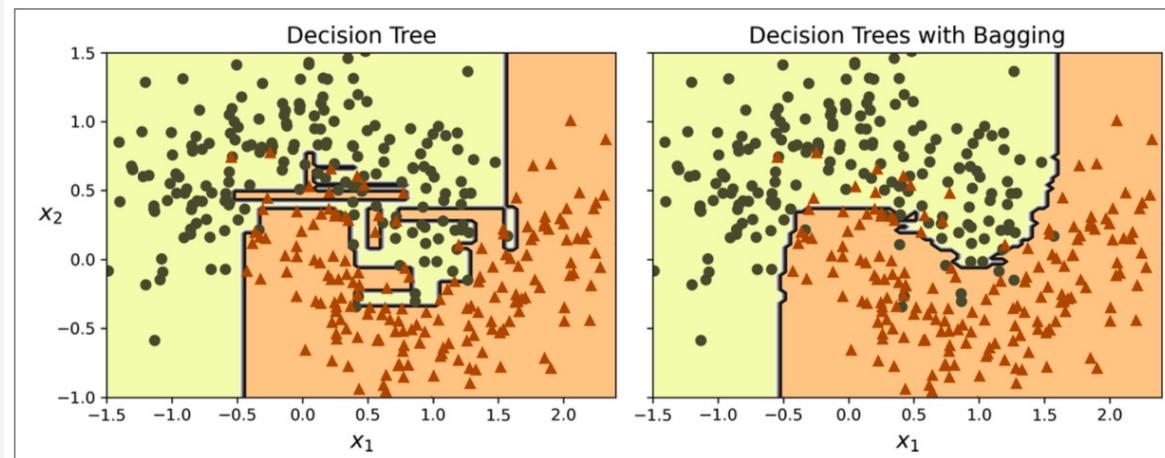


Figure 7-5. A single decision tree (left) versus a bagging ensemble of 500 trees (right)

BAGGING VS PASTING

- بین variance و bias کدامو باید انتخاب کنیم؟ در واقع یه tradeoff بین variance و bias هستش این مسئله. چون داخل bagging یه نمونه رو چندین برای train کردن یه مدل استفاده کنیم bias بیشتری داره نسبت به variance ولی pasting کمتری داره. همین استدلال رو برای pasting میاریم و میگیم variance کمتری داره و bias بیشتری داره. دیگه اینجا بستگی به این که کدام میتونه عملکرد بهتری داشته باشه (با cross-validation بررسی کنیم) مدل رو انتخاب میکنیم.
- با bagging فهمیدیم که یه نمونه میتونه چندین بار انتخاب بشه برای فرایند train کردن. اگه بخوایم m تا نمونه انتخاب کنیم که دقیقا سایز training set هست. میتونیم نشون بدیم به طور متوسط 63 درصد داده ها انتخاب میشن. به 37 درصد باقی مونده که انتخاب نمیشه میگن (oob).
در واقع هر نمونه احتمال داره oob یه سری مدل باشه. میتونیم اخر سر وقتی مدل نهایی تشکیل شد بیایم رو نمونه های oob امتحانش کنیم و ببینیم چه عملکردی داره.

```
>>> bag_clf = BaggingClassifier(DecisionTreeClassifier(), n_estimators=500,  
...                                oob_score=True, n_jobs=-1, random_state=42)  
...  
>>> bag_clf.fit(X_train, y_train)  
>>> bag_clf.oob_score_  
0.896
```

BIAS VS VARIANCE

High Variance:

- **Definition:** A model with high variance is one that is very sensitive to the specific data it is trained on. If you train this model on different subsets of the training data, its predictions can vary significantly.
- **Characteristics:**
 - **Overfitting:** High-variance models often overfit the training data. This means they perform very well on the training data but poorly on unseen test data because they have learned too much from the noise or specific details of the training data.
 - **Complex Models:** Typically, models that are very complex (e.g., deep neural networks, decision trees with many splits) have high variance. They can capture intricate patterns in the training data but may fail to generalize to new data.
 - **Example:** Imagine a decision tree model that splits the data into many small subsets. This tree might perfectly classify the training data, but when presented with new data, it may fail because it is too narrowly focused on the specific examples it has seen.

Low Variance:

- **Definition:** A model with low variance does not change its predictions much when trained on different subsets of the training data. It provides stable and consistent predictions, regardless of the specific training data.
- **Characteristics:**
 - **Underfitting:** Low-variance models might underfit the training data, meaning they oversimplify the patterns in the data and may not capture important trends or relationships. This can lead to poor performance on both the training and test data.
 - **Simple Models:** Models that are simple, such as linear regression with few features or a shallow decision tree, tend to have low variance. They generalize well but may not capture the complexity of the data.
 - **Example:** A linear regression model that fits a straight line through data points is an example of a low-variance model. It might miss more complex patterns in the data but will provide consistent results across different datasets.

CONTINUE

Bias-Variance Tradeoff:

- **Balancing Act:** The key in machine learning is to find the right balance between bias and variance:
 - **High Bias, Low Variance:** The model is too simple and doesn't capture enough of the data's complexity, leading to underfitting.
 - **Low Bias, High Variance:** The model is too complex, capturing noise in the training data and leading to overfitting.
 - **Optimal Model:** The goal is to achieve low bias and low variance, where the model is complex enough to capture the underlying patterns but not so complex that it overfits the data.

RANDOM PATCHES AND RANDOM SUBSPACES

- همونجور که میتوانیم نمونه برداری کنیم از نمونه های training set همین کار را انجام بدیم برای featurer های ورودیمون. این کار باعث میشه مدل هامون روی یه زیرمجموعه رندوم از featurer هامون بشن. این کار وقتی با یه دیتای high dimensional رو به رو هستیم، خوبه. اگه هم روی random patches features روی نمونه برداری رندوم بکنیم اسمش میشه training set .random subspaces method. اما اگه فقط روی feature ها همچین کاری کنیم میشه method.

- به جای اینکه بیایم از یه bagging classifier استفاده کنیم و به یه Random Forests random forest classifier پاس بدیم، میتوانیم مستقیم از یه decisionTreeClassifier کنیم.

```
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)
```

CONTINUE

- داخل random forest یه الگوریتمی به کار بردن اونم اینه که به جای اینکه داخل کل featurer ها بگردن برای اینکه بهترین فیچر رو مبنا تصمیم گیری اون نود داخل درخت بذارن (همون چیزی که دیدیم داخل فصل قبل) میاد این سرچ رو داخل یه سری فیچر های خاص میکنه. به طور دیفالت میاد روی رادیکال n که n تعداد فیچر ها هست سرچ میزنه. همین باعث میشه سرعت کار خیلی بیشتر بشه.

```
bag_clf = BaggingClassifier(  
    DecisionTreeClassifier(max_features="sqrt", max_leaf_nodes=16),  
    n_estimators=500, n_jobs=-1, random_state=42)
```

- داخل درخت ها دیدیم که میومدیم یه threshold پیدا کنیم برای اینکه بتونیم به درستی Extra-Trees دیتاهای را جداسازی کنیم. حالا توی random forest دیدیم صرفا روی یه سری فیچر خاص دنبال اون randomness. الان میایم یه threshold دیگه هم میذاریم رو کار و میگیم extra-trees همون یه threshold رو هم به طور رندوم برای فیچر ها انتخاب میکنن. پیاده سازی سختی هم نداره و برای اینکه بدونیم extre-trees بهتره یا random forests میایم یه cross-validation میزیم روشنون ببینیم کدام بهتر خروجی میده برای دیتاهامون.

FEATURE IMPORTANCE

- یه کار خوبی که random forests میکنن برایتون اینه که اهمیت فیچر هارو میتوونن بهمون بگن. چطوری؟ نگاه میکنه که چند تا نود داخل درخت ها به وسیله یه فیچر تونستن جداسازی خوبی انجام بدند و یه وزن هم میده به هر نود که وزنش مناسب با اینه چند تا نمونه با این فیچر تقسیم بندی شدند. در نهایت جمع اهمیت همه فیچر ها مساوی یک میشه و با شکل زیر میتوانیم بفهمیم کدام فیچر ها تاثیرگذار بودن.

```
>>> from sklearn.datasets import load_iris  
>>> iris = load_iris(as_frame=True)  
>>> rnd_clf = RandomForestClassifier(n_estimators=500, random_state=42)  
>>> rnd_clf.fit(iris.data, iris.target)  
>>> for score, name in zip(rnd_clf.feature_importances_, iris.data.columns):  
...     print(round(score, 2), name)  
...  
0.11 sepal length (cm)  
0.02 sepal width (cm)  
  
0.44 petal length (cm)  
0.42 petal width (cm)
```

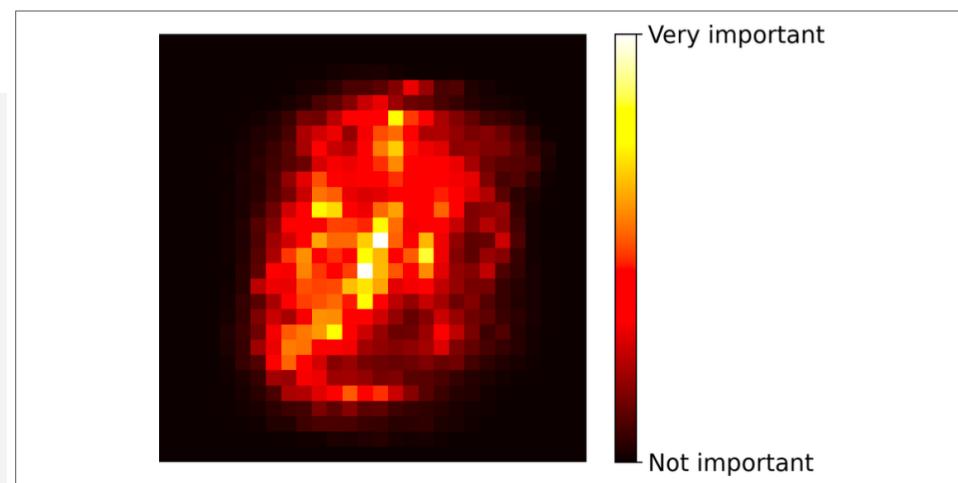


Figure 7-6. MNIST pixel importance (according to a random forest classifier)

میتوانیم تاثیر هر پیکسل رو روی دیتاست
ببینیم با این کار.

BOOSTING

- به این کانسپت که بیایم از ensemble method استفاده کنیم و یه سری مدل ضعیف تر رو ترکیب کنیم تا یه یه مدل قوی تر برسیم. ایده کلی پشت این روش اینه هر مدل رو جدا جدا train کنیم و هر مدل سعی کن اشتباهات مدل قبلی رو تکرار نکنه. یکی از متد های خیلی معروف داخل adaBoost,boosting هستش.
- AdaBoost: یه روش برای اینکه بتونیم اشتباهات مدل قبلی رو تکرار نکنیم اینه که اهمیت بیشتری به نمونه هایی که مدل قبلی اشتباه پیش بینی کرده داشته باشیم.

Figure 7-8 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 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 much less 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.

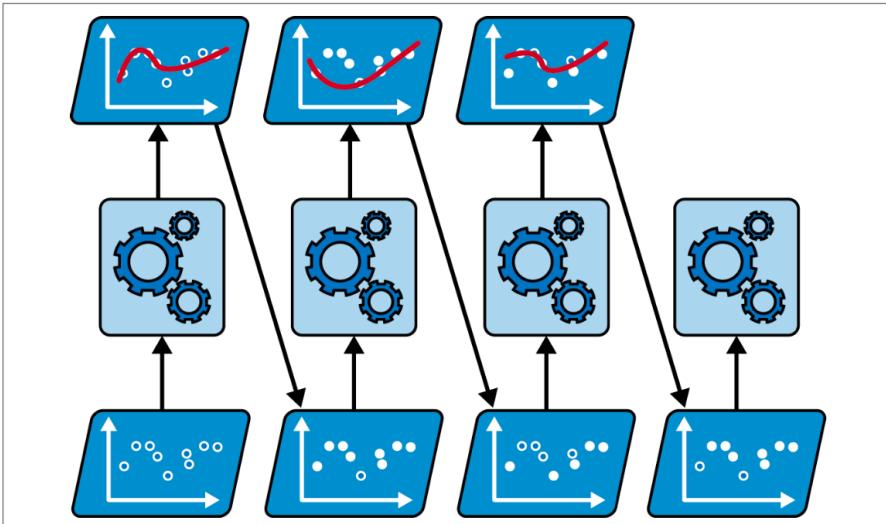


Figure 7-7. AdaBoost sequential training with instance weight updates

CONTINUE

- برحسب اینکه هر مدل از این توالی چه دققی داری بهشون یه ضریب میدیم و در نهایت برای پیش‌بینی روی یه نمونه جدید همشون پیش‌بینی انجام میدن و جمع میکنیم با این ضریب‌ها و میانگین میگیریم. تو این ماجرا تاثیر داره که چقدر نسبت به اشتباهات نمونه قبلی حساس باشیم.

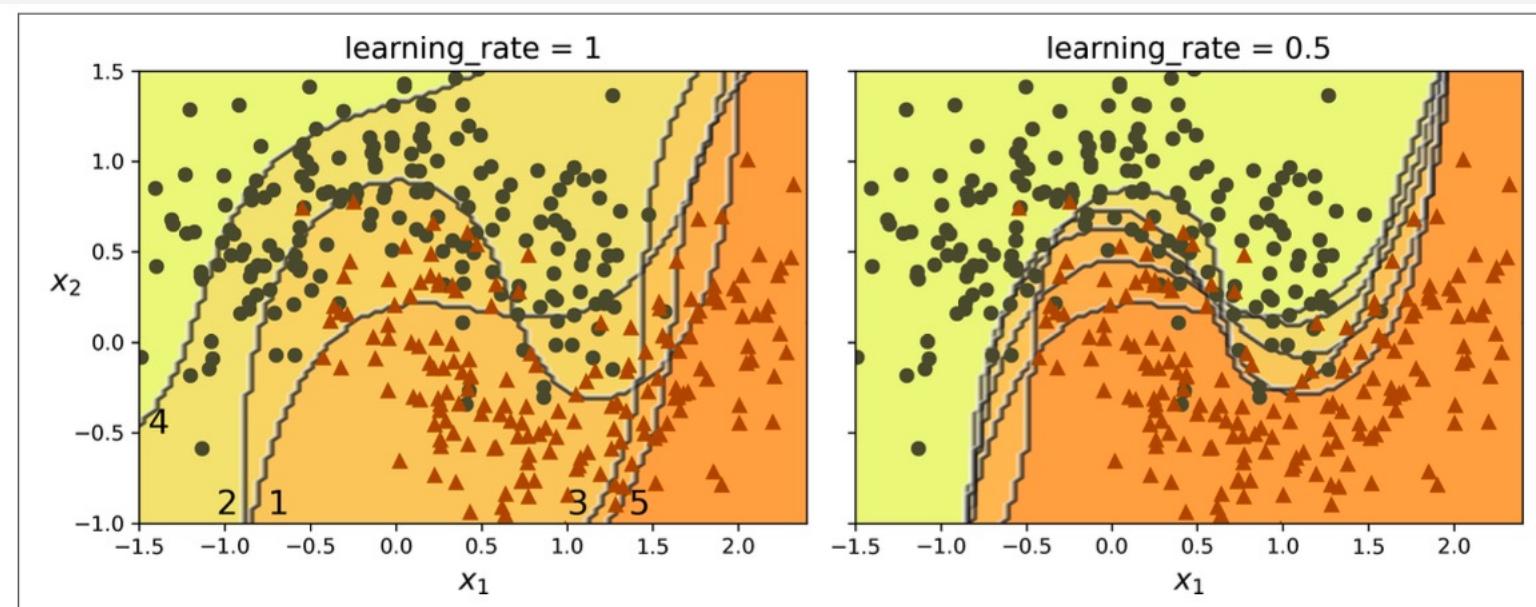


Figure 7-8. Decision boundaries of consecutive predictors

CONTINUE

Let's take a closer look at the 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

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

- به ذره به ریاضیات نگاه بندازیم:

- حالا که ضریب اشتباه هر مدل رو حساب کردیم میتونیم وزن مناسب برای خروجی رو به هر مدل بدیم.

The predictor's weight α_j is then computed using [Equation 7-2](#), where η is the learning rate hyperparameter (defaults to 1).¹⁵ 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

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

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

for $i = 1, 2, \dots, m$

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

CONTINUE

- حالا باید بیایم ضریب خطاهایی که به نمونه ها دادیم رو نرمال کنیم.

Then all the instance weights are normalized (i.e., divided by $\sum_{i=1}^m w^{(i)}$).

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.

- نحوه پیش‌بینی رو برای regression گفته‌یم، صرفا برای classification این شکلی پیش بینی می‌کنه:

Equation 7-4. AdaBoost predictions

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

IMPLEMENTATION

The following code trains an AdaBoost classifier based on 30 *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=30,
    learning_rate=0.5, random_state=42)
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 regularizing the base estimator.

GRADIENT BOOSTING

- یه راه دیگه برای gradient boosting، boosting داره، این که به جای اینکه مستقیم به خود اشتباهات بپردازه، به اختلاف ارور ها میپردازه. بیایم یه مثال ببینیم ازش.

```
import numpy as np
from sklearn.tree import DecisionTreeRegressor

np.random.seed(42)
X = np.random.rand(100, 1) - 0.5
y = 3 * X[:, 0] ** 2 + 0.05 * np.random.randn(100) # y = 3x2 + Gaussian noise

tree_reg1 = DecisionTreeRegressor(max_depth=2, random_state=42)
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, random_state=43)
tree_reg2.fit(X, y2)
```

And then we'll 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, random_state=44)
tree_reg3.fit(X, y3)
```

CONTINUE

- حالا میتوانیم با ensemble این ۳ تا مدلی که ساختیم یه نمونه جدید، مقدارش رو پیش بینی کنیم.

```
>>> X_new = np.array([[-0.4], [0.], [0.5]])
>>> sum(tree.predict(X_new) for tree in (tree_reg1, tree_reg2, tree_reg3))
array([0.49484029, 0.04021166, 0.75026781])
```

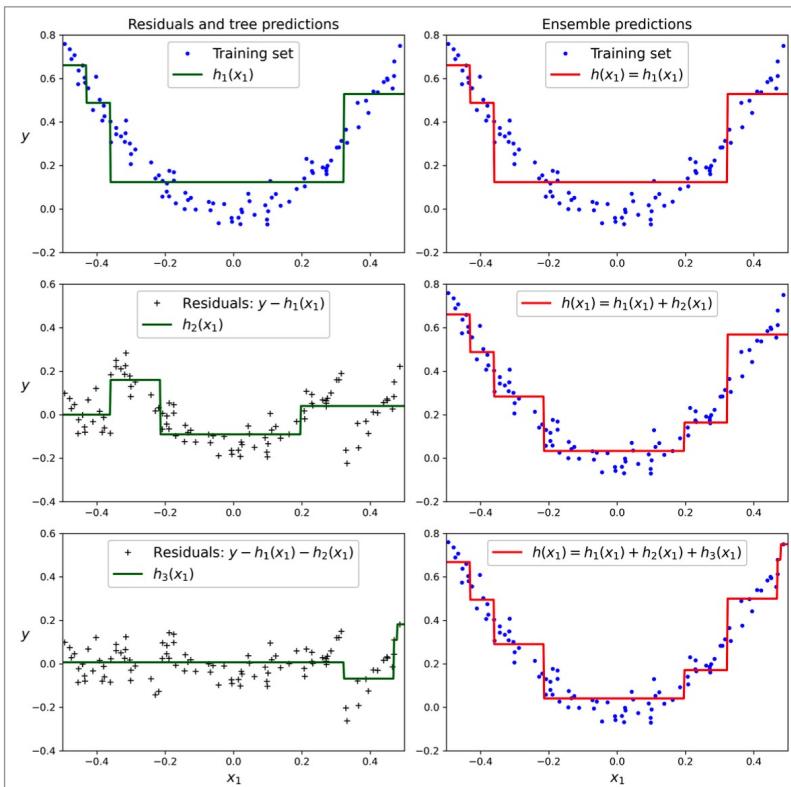


Figure 7-9. In this depiction of gradient boosting, the first predictor (top left) is trained normally, then each consecutive predictor (middle left and lower left) is trained on the previous predictor's residuals; the right column shows the resulting ensemble's predictions

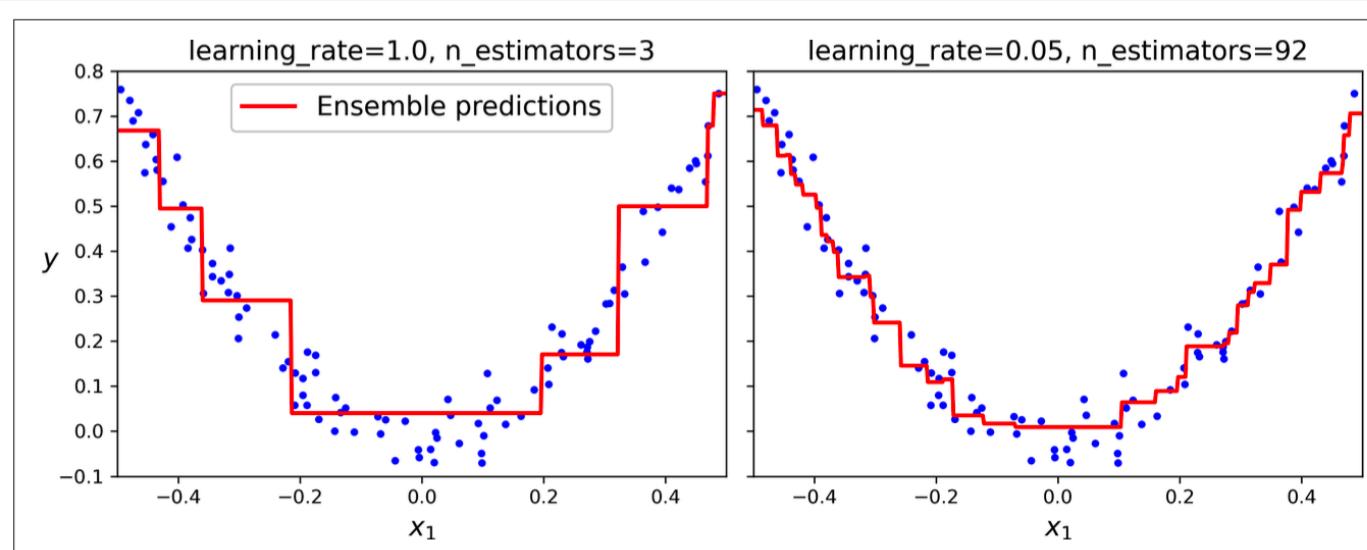
- با این شکل میتوانیم پیش بینی تک تک مدل هارو به علاوه وقتی که با هم جمعشون میکنیم بفهمیم. که در نهایت تبدیل میشه به مدل نهاییمون شکل پایین سمت راست.

CONTINUE

- به جای این کار میتوانیم خیلی راحت از خود کلاسی scikit-learn برامون درست کرده استفاده کنیم.

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

یه نکته هست صرفاً اونم learning rate که اینجا داره چی کار میکنه. در واقع کارش اینه بگه هر درخت به تنهایی چقدر تاثیر گذار باشه رو جوابمون. اگه مقدارش رو کمتر میکردیم نیاز داشتیم درخت های بیشتری رو train کنیم. البته از یه حدی بیشتر کردن درختامون میتونه منجر به overfit بشه.



CONTINUE

- برای پیدا کردن تعداد مناسب درخت میتوانیم cross-validation یا gridsearch استفاده کنیم. اما یه روش راحت وجود داره. اونم اینکه بیایم از hyperparameter randomizesearch استفاده کنیم. کارش اینه وقتی مثلا مساوی ۱۰ میذاریم، وقتی ببینه جایی بعد از اینکه ۱۰ تا درخت اضافه کرده منجر نشده به اینکه عملکرد مدل بهتر بشه. یه جورایی شبیه همون که قبلا خوندیم هستش.

```
gbrt_best = GradientBoostingRegressor(  
    max_depth=2, learning_rate=0.05, n_estimators=500,  
    n_iter_no_change=10, random_state=42)  
gbrt_best.fit(X, y)
```

- اگه خیلی کم بذاریم ممکنه مدلمون underfit رخ بده روش. اگه خیلی بزرگ بذاریم ممکنه overfit کنه. ببینیم چند تا درخت استفاده کرده.

```
>>> gbrt_best.n_estimators_
```

CONTINUE

- شاید بپرسیم از کجا میفهمه عملکردش بهتر میشه یا نه که بخواه دیگه درختی اضافه نکنه. در واقع وقتی از این روش استفاده میکنیم میاد training set رو به یه validation set کوچیکتر و یه تبدیل میکنه تا بتونه عملکردش رو بسنجه.
- کلی hyperparameter هست که میتونیم فرایند یادگیری رو کنترل کنیم. هم سایز validation رو هم اینکه دقیقا چند درصد معیار ما باشه تا بگیم مدل پیشرفت نداره که طور پیشفرض 0.0001 هستش.
- همچنین میتونیم از subsample هم اینجا استفاده کنیم تا به طور رندوم یه سری نمونه رو برداریم برای train کردن. که این کار باعث میشه bias بیشتری داشته باشیم و variance کمتر. به این کار stochastic gradient boosting میگن.

HISTOGRAM-BASED GRADIENT BOOSTING

- یه مدلی که ارائه شده histogram-based gradient boosting هستش. کاری که میکنه میاد فیچر ها رو bin کنه و به جاشون عدد بذاره. از ۲۵۵ تا bin بیشتر هم نمیشه استفاده کرد. کاری که میکنه اینه احتمال threshold های ممکن برای فیچر هارو خیلی کمتر میکنه، همچنین تعداد کل threshold ها با خاطر اینکه فیچر ها به n تا bin مپ شده میشه n تا. همچنین کاری اصولا برای دیتاست های خیلی بزرگ خوبه.

As a result, this implementation has a computational complexity of $O(b \times m)$ instead of $O(n \times m \times \log(m))$, where b is the number of bins, m is the number of training instances, and n is the number of features. In practice, this means that HGB can train hundreds of times faster than regular GBRT on large datasets. However, binning causes a precision loss, which acts as a regularizer: depending on the dataset, this may help reduce overfitting, or it may cause underfitting.

CONTINUE

• همچنان که در اینجا دو تا فیچر خوب داره که باعث میشه بتوانیم از فیچرهای دسته بندی شده (categorical) و هچنین از missing values ها داخل نمونه ها پشتیبانی کنه. صرفا باید حواسمن باشه categorical features باشد به یه عدد بین ۰ تا ۱- max_bins مپ بشن. که برای این کار میتوانیم از یه ordinal encoder استفاده کنیم.

```
from sklearn.pipeline import make_pipeline
from sklearn.compose import make_column_transformer
from sklearn.ensemble import HistGradientBoostingRegressor
from sklearn.preprocessing import OrdinalEncoder

hgb_reg = make_pipeline(
    make_column_transformer((OrdinalEncoder(), ["ocean_proximity"]),
                           remainder="passthrough"),
    HistGradientBoostingRegressor(categorical_features=[0], random_state=42)
)
hgb_reg.fit(housing, housing_labels)
```

The whole pipeline is just as short as the imports! No need for an imputer, scaler, or a one-hot encoder, so it's really convenient. Note that `categorical_features` must be set to the categorical column indices (or a Boolean array). Without any hyperparameter tuning, this model yields an RMSE of about 47,600, which is not too bad.

STACKING

- اخرين روشی که برای ensemble کردن ياد ميگيريم stacking هستش. ايده پشتش اينه: چرا به جاي اينکه از روش هايی مثل hard voting اين کار استفاده نکنیم؟

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).

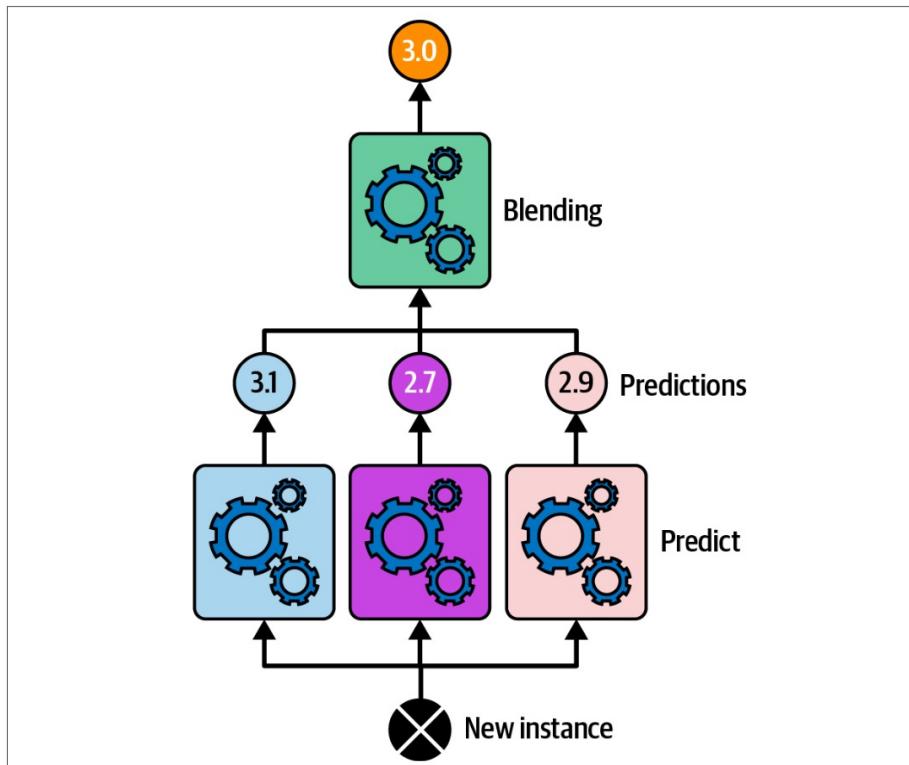
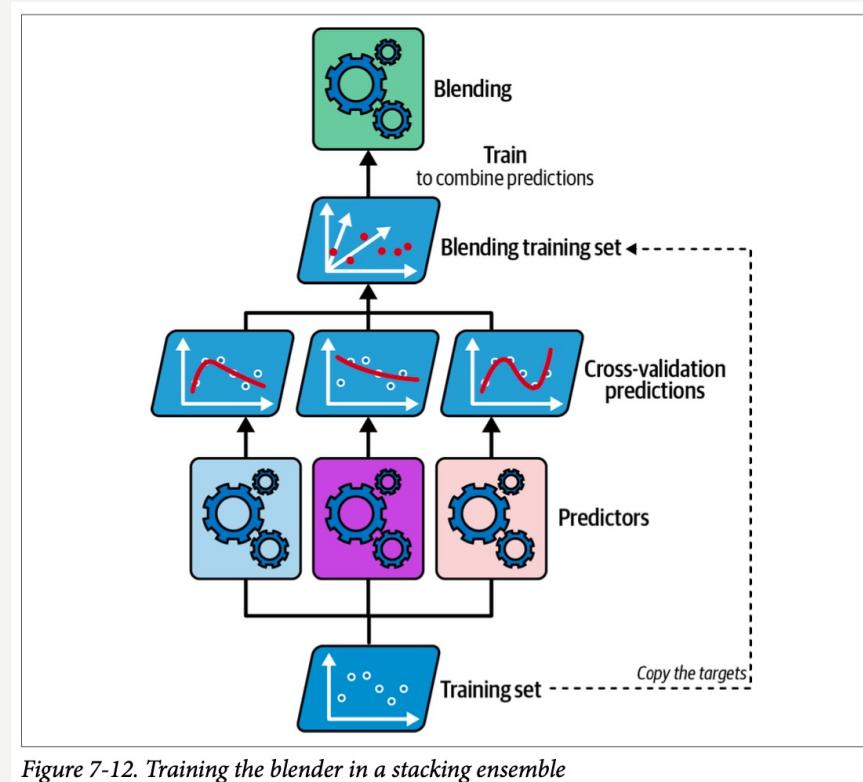


Figure 7-11. Aggregating predictions using a blending predictor

CONTINUE

- دیدیم که به مدل بالای میگن blender را train کنیم، از استفاده کنیم تا خروجی همه مدل هارو روی training set بگیریم و به عنوان ورودی به blender بدیم برای train کردن. دقت کنیم که هرچقدر هم نمونه های training set فیچر داشته باشند، فیچر ورودی ما برای این مدل صرفا تعداد مدل های پایینیه.



CONTINUE

- میتوانیم همین مفهوم blender را دوباره تکرار کنیم. یعنی یه لایه اضافه کنیم که شامل یه سری مدل برای باشه و دوباره روی اون یه predict بذاریم.

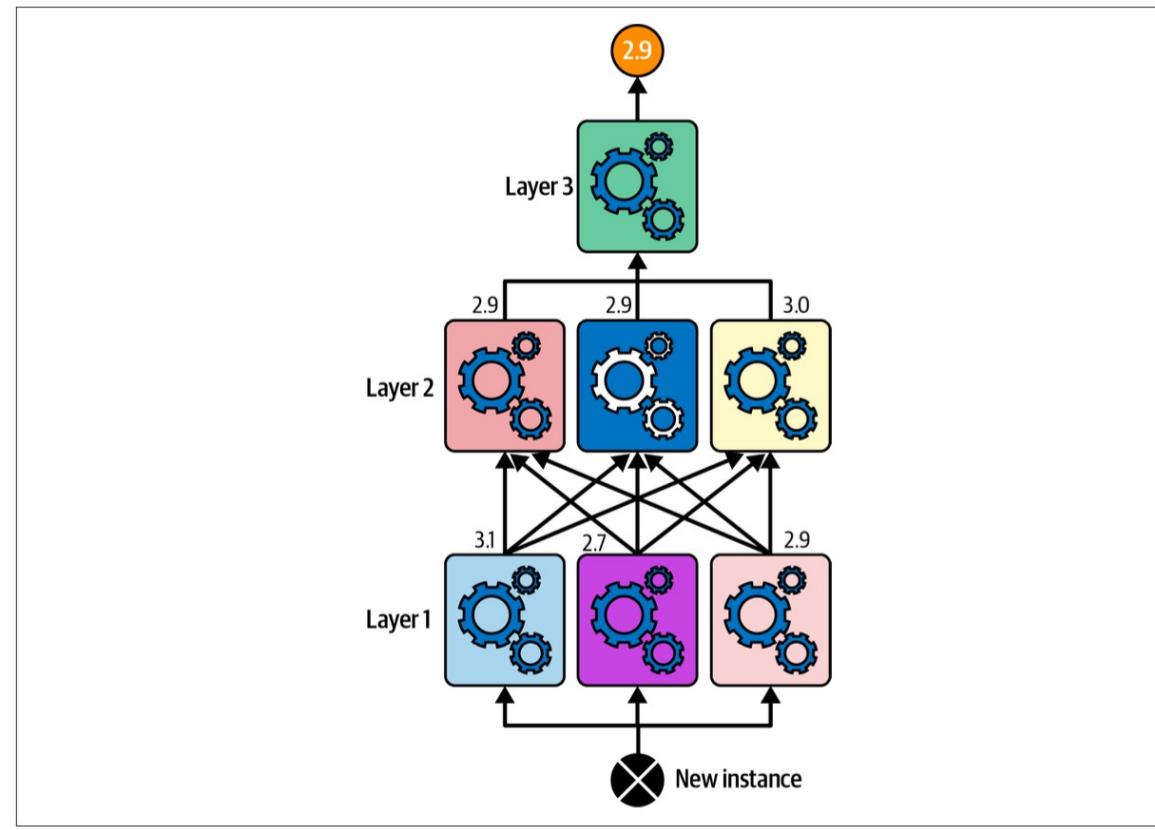


Figure 7-13. Predictions in a multilayer stacking ensemble

IMPLEMENTATION AND FINISH

Scikit-Learn provides two classes for stacking ensembles: `StackingClassifier` and `StackingRegressor`. For example, we can replace the `VotingClassifier` we used at the beginning of this chapter on the moons dataset with a `StackingClassifier`:

```
from sklearn.ensemble import StackingClassifier

stacking_clf = StackingClassifier(
    estimators=[
        ('lr', LogisticRegression(random_state=42)),
        ('rf', RandomForestClassifier(random_state=42)),
        ('svc', SVC(probability=True, random_state=42))
    ],
    final_estimator=RandomForestClassifier(random_state=43),
    cv=5 # number of cross-validation folds
)
stacking_clf.fit(X_train, y_train)
```

For each predictor, the stacking classifier will call `predict_proba()` if available; if not it will fall back to `decision_function()` or, as a last resort, call `predict()`. If you don't provide a final estimator, `StackingClassifier` will use `LogisticRegression` and `StackingRegressor` will use `RidgeCV`.