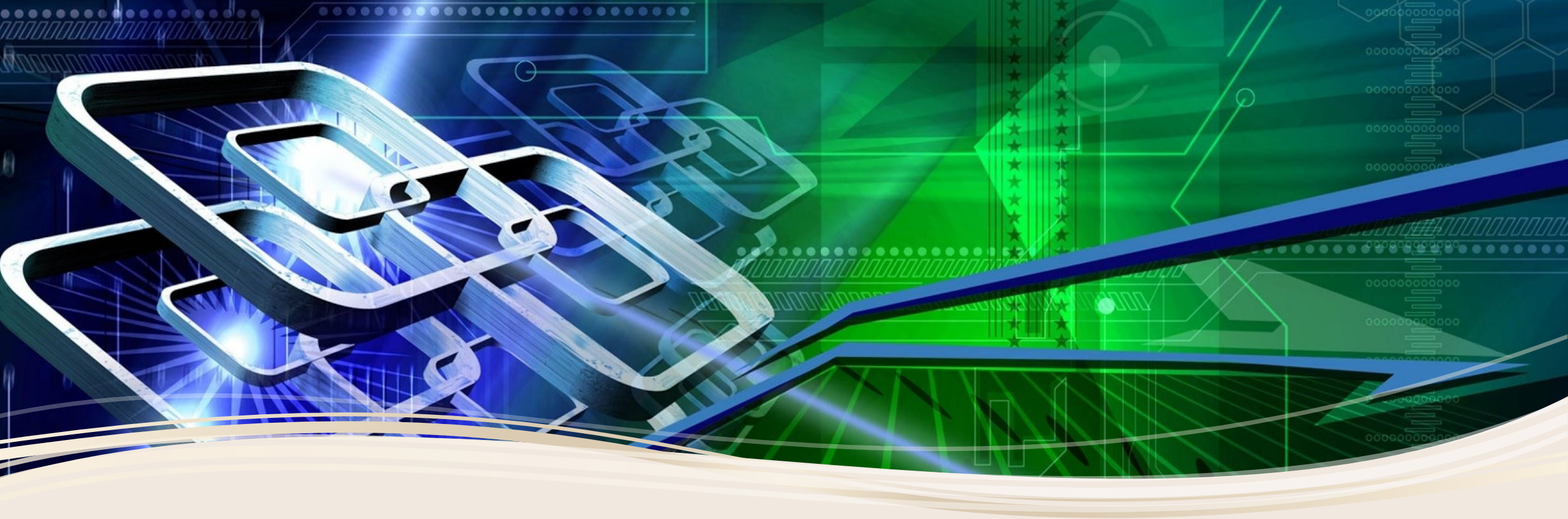


IN242 - Kecerdasan Mesin
Chapter 7

Ensemble Learning and Random Forests

Introduction

- Aggregated Answer = the wisdom of the crowd.
- A group of predictors is called an ensemble → Ensemble Learning → Ensemble Algorithm
- In this chapter, we will discuss the most popular Ensemble methods, including bagging, boosting, and stacking.

Ensemble Learning dan Random Forests, Voting Classifiers

- In Voting Classifiers, each one achieving about 80% accuracy, which has a Logistic Regression classifier, an SVM classifier, a Random Forest classifier, a K-Nearest Neighbors classifier, and other classifiers (Figure 7-1).
- Not only that, but a straightforward way to create a better classifier is to aggregate the predictions of each classifier and predict the class that gets the most votes, called a **hard voting classifier** (see Figure 7-2).

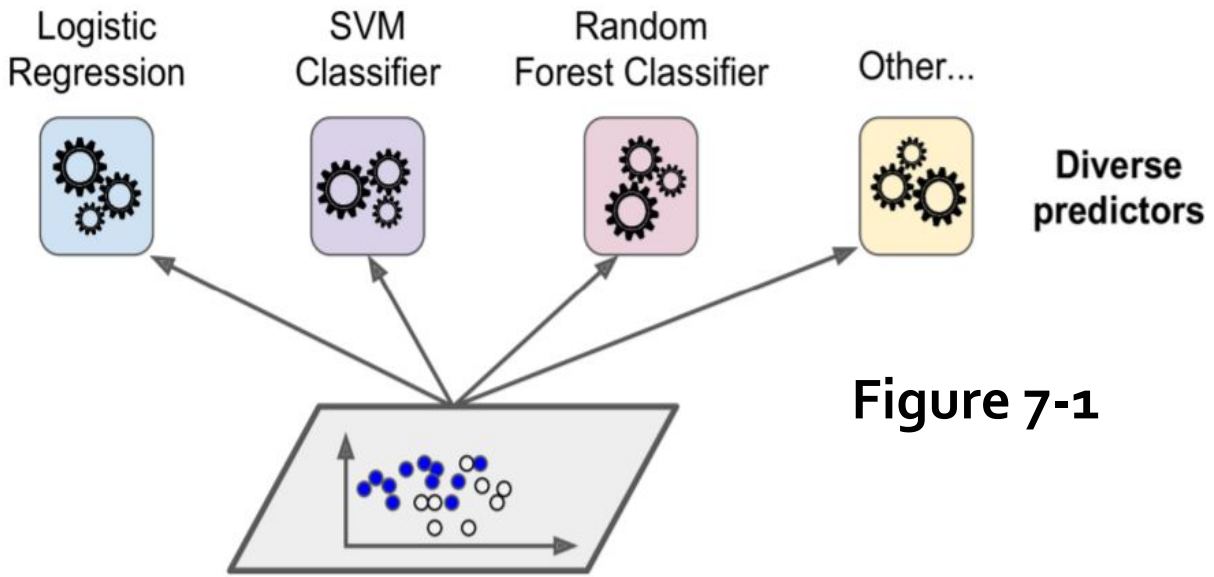


Figure 7-1. Training diverse classifiers

Figure 7-1

Figure 7-2

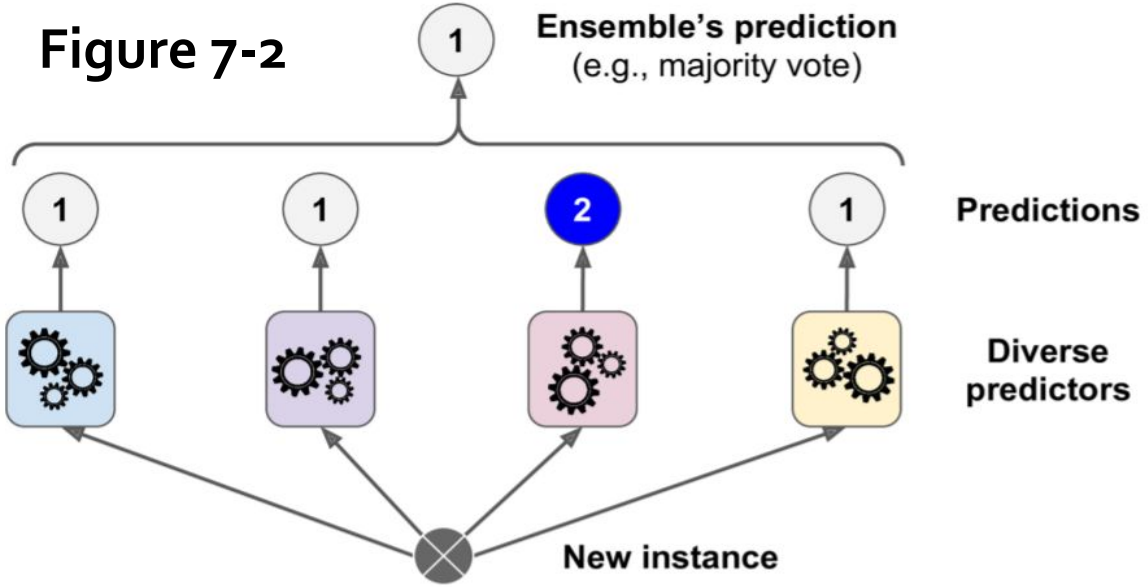


Figure 7-2. Hard voting classifier predictions

Ensemble Learning dan Random Forests, Voting Classifiers

- Surprisingly, this voting classifier often achieves a higher accuracy in Machine Learning than the best classifier in the ensemble.
- Because, even if each classifier is a weak learner (which means only slightly does better than random guessing), the ensemble can still be a strong learner (achieving high accuracy), provided that there are a sufficient number of weak learners, and they are sufficiently diverse.

Ensemble Learning dan Random Forests, Voting Classifiers

- Ensemble methods work best when the predictors 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.

Ensemble Learning dan Random Forests, Voting Classifiers

```
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)
```

Ensemble Learning dan Random Forests, Voting Classifiers

- 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
```

Ensemble Learning dan Random Forests, Voting Classifiers

- If all classifiers can estimate class probabilities (i.e., they all have a **predict_proba()** method), then you can tell Scikit-Learn to predict the class with the highest-class probability, averaged over all the individual classifiers.
- This is called **soft voting**. It often achieves higher performance than hard voting because it gives more weight to highly confident votes.
- All you need to do is replace **voting="hard"** with **voting="soft"** and *ensure that all classifiers can estimate class probabilities*.

Voting Classifiers

Soft voting:

In [8]:
```
log_clf = LogisticRegression(solver="lbfgs", random_state=42)
rnd_clf = RandomForestClassifier(n_estimators=100, random_state=42)
svm_clf = SVC(gamma="scale", 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)
```

Out[8]:
```
VotingClassifier(estimators=[('lr', LogisticRegression(random_state=42)),
                             ('rf', RandomForestClassifier(random_state=42)),
                             ('svc', SVC(probability=True, random_state=42))],
                 voting='soft')
```

In [9]:
```
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 0.896
VotingClassifier 0.92
```

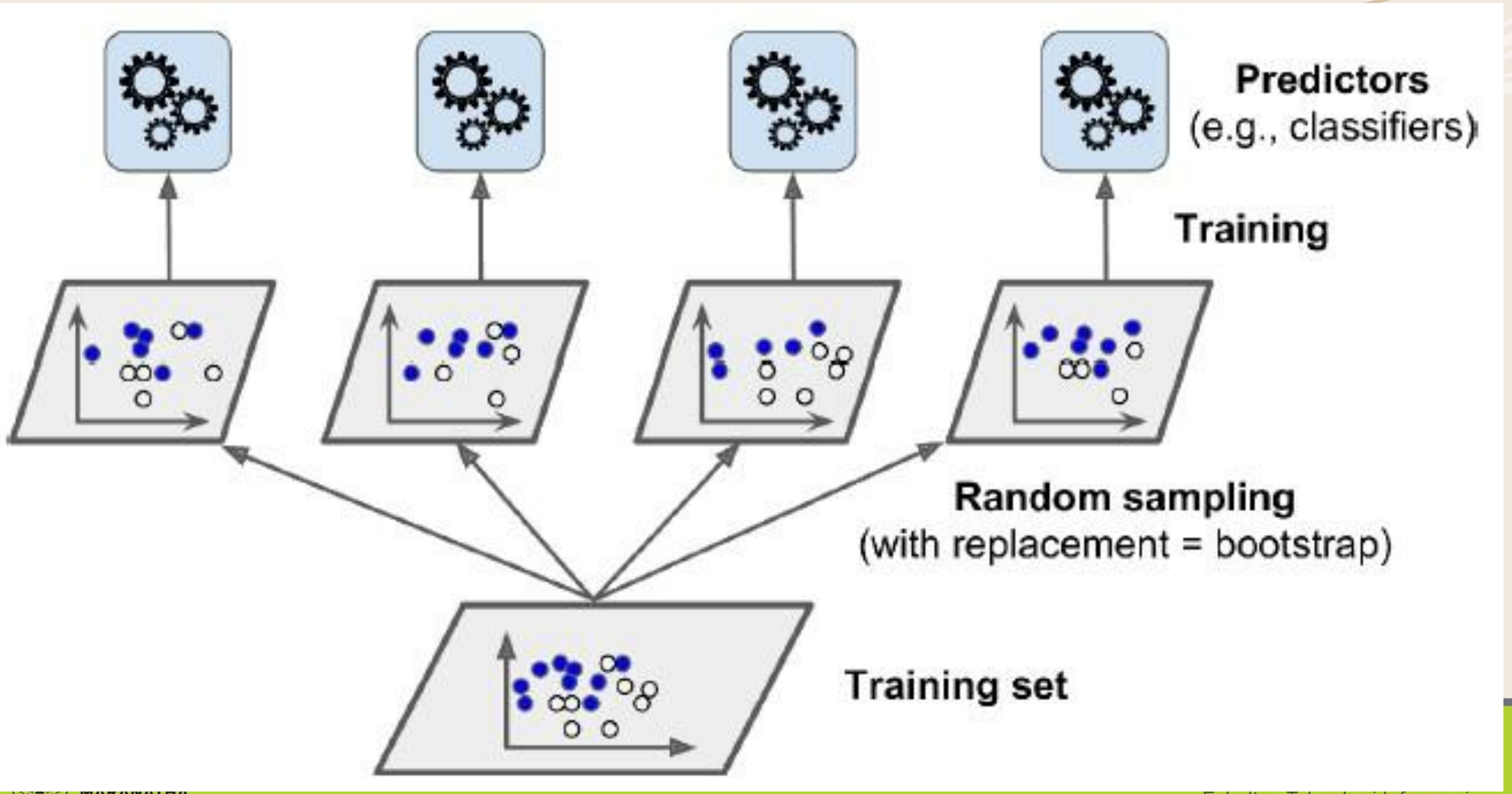
Chapter 7: Bagging and Pasting, Bagging and Pasting in Scikit-Learn, Out-of-Bag Evaluation

Bagging and Pasting

Use the same training algorithm for every predictor and train them on different random subsets of the training set

- Bagging (short for bootstrap aggregating) is when sampling performed with replacement
- Pasting is when sampling performed without replacement

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.


Predictors
(e.g., classifiers)
Training
Random sampling
(with replacement = bootstrap)
Training set

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 (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.

Bagging and Pasting in Scikit-Learn

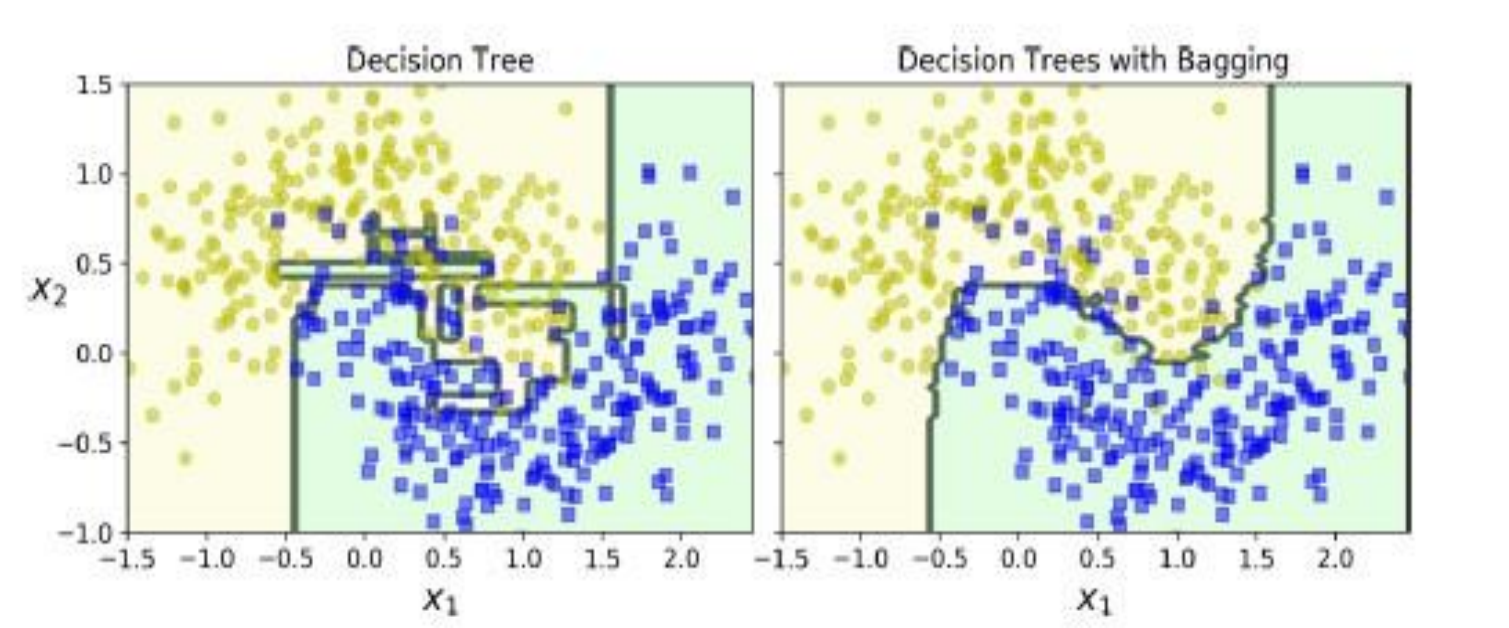
```
from sklearn.ensemble import BaggingClassifier
from sklearn.tree import DecisionTreeClassifier

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)
```

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

Bagging and Pasting

- This is an example of bagging, but if you want to use pasting instead, just set **bootstrap = False**
- **n_jobs** parameters tells Scikit-Learn the number of CPU cores to use for training and predictions (**-1** tells Scikit-Learn to use all available cores)

Bagging and Pasting in Scikit-Learn



- Figure 7.5 compares the decision boundary of a single Decision Tree with the decision boundary of a bagging ensemble of 500 trees (from the preceding code), both trained on the moon's dataset.
- 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)

Bagging and Pasting in Scikit-Learn

- **Bagging often results in better models, which explains why it is generally preferred**.

- 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 Bagging Classifier samples *m* training instances with replacement (**bootstrap=True**), where *m* is the size of the training set and only about 63% of the training instances are sampled on average for each predictor and 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.
- We can evaluate the ensemble itself by averaging out the oob evaluations of each predictor.

Out-of-Bag Evaluation

- In Scikit-Learn, you can set **oob_score=True** when creating a Bagging Classifier to request an automatic oob evaluation after training. The following code demonstrates this.
- The resulting evaluation score is available through the oob_score_ variable:

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

```
>>> bag_clf.oob_score_
0.90133333333333332
```

Out-of-Bag Evaluation

- According to this oob evaluation, this Bagging Classifier is likely to achieve about 90.1% accuracy on the test set. Then, verify this:

```
>>> from sklearn.metrics import accuracy_score
>>> y_pred = bag_clf.predict(X_test)
>>> accuracy_score(y_test, y_pred)
0.9120000000000003
```

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

Out-of-Bag Evaluation

- The **oob decision function** for each training instance is also available through the **oob_decision_function_** attribute.
- In this case (since the base estimator has a predict_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]])
```

Chapter 7: Random Patches and Random Subspaces, Random Forests, Extra-Trees, Feature Importance

Random Patches and Random Subspaces

- The BaggingClassifier class supports sampling the features as well.
- Sampling is controlled by two hyperparameters: **max_features** and **bootstrap_features**.
- They work the same way as max_samples and bootstrap, but for feature sampling instead of instance sampling. Thus, each predictor will be trained on a random subset of the input features.
- This technique is particularly **useful** when **you are dealing with high-dimensional inputs (such as images)**.

Random Patches and Random Subspaces Method

- Sampling both training instances and features is called the **Random Patches method**. Keeping all training instances (by setting bootstrap=False and max_samples=1.0) but sampling features (by setting bootstrap_features to True and/or max_features to a value smaller than 1.0) is called the **Random Subspaces method**.
- Sampling features result in even more predictor diversity, trading a bit more bias for a lower variance.

Random Forest

- Random Forest is an ensemble of Decision Trees, generally trained using the **bagging method** (or sometimes pasting), typically with **max_samples** set to the size of the training set.

Random Forest Algorithm

- The Random Forest algorithm introduces **extra randomness when growing trees**; instead of searching for the very best feature when splitting a node), **it searches for the best feature among a random subset of features**.
- The algorithm results in greater tree diversity, which (again) trades a higher bias for a lower variance, generally yielding an overall better model.

```
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)
```

Random Forest Algorithm

- The following BaggingClassifier is roughly equivalent to the previous RandomForestClassifier:

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

Feature Importance

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

Entropy, Gini Index, and Classification Error

$$\text{Entropy} = -\sum_{i=0}^{c-1} p_i(t) \log_2 p_i(t), \quad (3.4)$$

$$\text{Gini index} = 1 - \sum_{i=0}^{c-1} p_i(t)^2, \quad (3.5)$$

$$\text{Classification error} = 1 - \max_i [p_i(t)], \quad (3.6)$$

Entropy, Gini Index, and Classification Error

Node N_1	Count
Class=0	0
Class=1	6

$$\text{Gini} = 1 - (0/6)^2 - (6/6)^2 = 0$$
$$\text{Entropy} = -(0/6)\log_2(0/6) - (6/6)\log_2(6/6) = 0$$
$$\text{Error} = 1 - \max[0/6, 6/6] = 0$$

Node N_2	Count
Class=0	1
Class=1	5

$$\text{Gini} = 1 - (1/6)^2 - (5/6)^2 = 0.278$$
$$\text{Entropy} = -(1/6)\log_2(1/6) - (5/6)\log_2(5/6) = 0.650$$
$$\text{Error} = 1 - \max[1/6, 5/6] = 0.167$$

Node N_3	Count
Class=0	3
Class=1	3

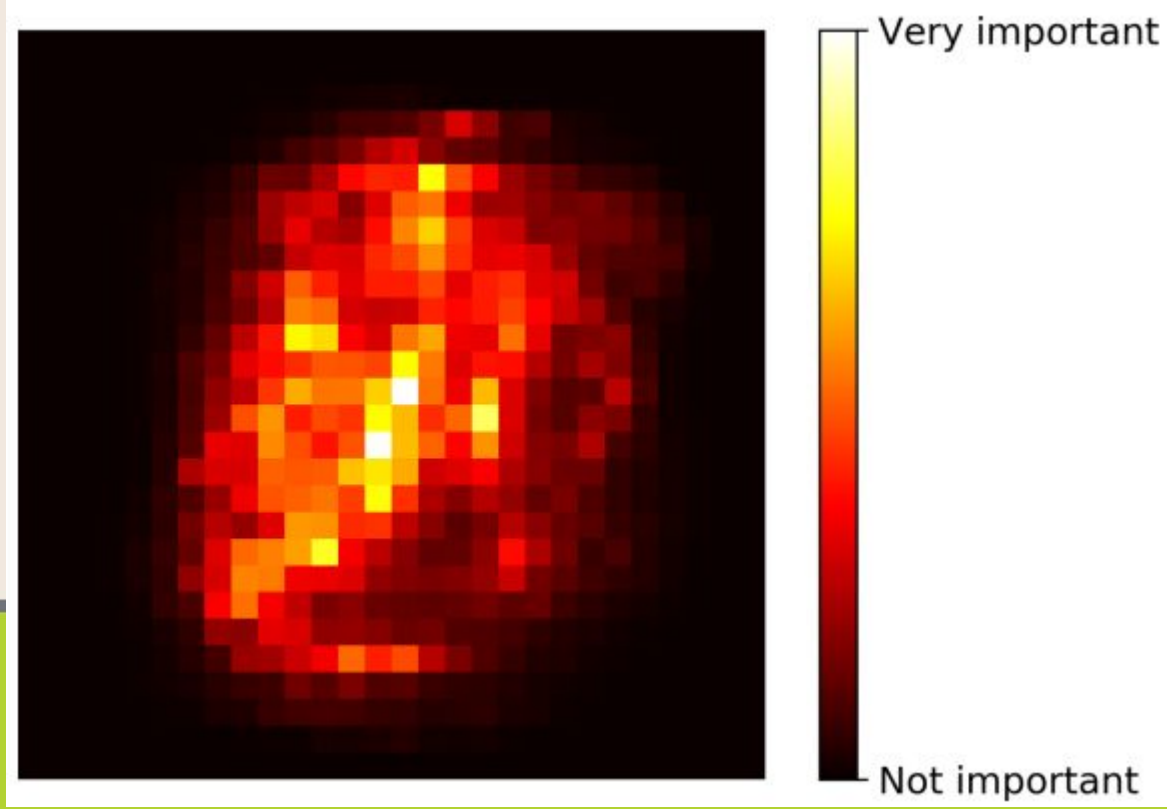
$$\text{Gini} = 1 - (3/6)^2 - (3/6)^2 = 0.5$$
$$\text{Entropy} = -(3/6)\log_2(3/6) - (3/6)\log_2(3/6) = 1$$
$$\text{Error} = 1 - \max[3/6, 3/6] = 0.5$$

Feature Importance

```
>>> 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
```

Feature Importance

- if you train a Random Forest classifier on the MNIST dataset and plot each pixel's importance, you get the image represented in this image:



Chapter 7: Boosting, Adaboost, Gradient Boosting, Stacking

Boosting, Adaboost, Gradient Boosting, Stacking

- **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** and **Gradient Boosting**.

Boosting, Adaboost, Gradient Boosting, Stacking

- One way for a new predictor to correct its predecessor is to pay a bit more attention to the training instances that the predecessor under fitted.
- This results in new predictors focusing more and more on the hard cases. This is the technique used by **AdaBoost.**

Boosting, Adaboost, Gradient Boosting, Stacking

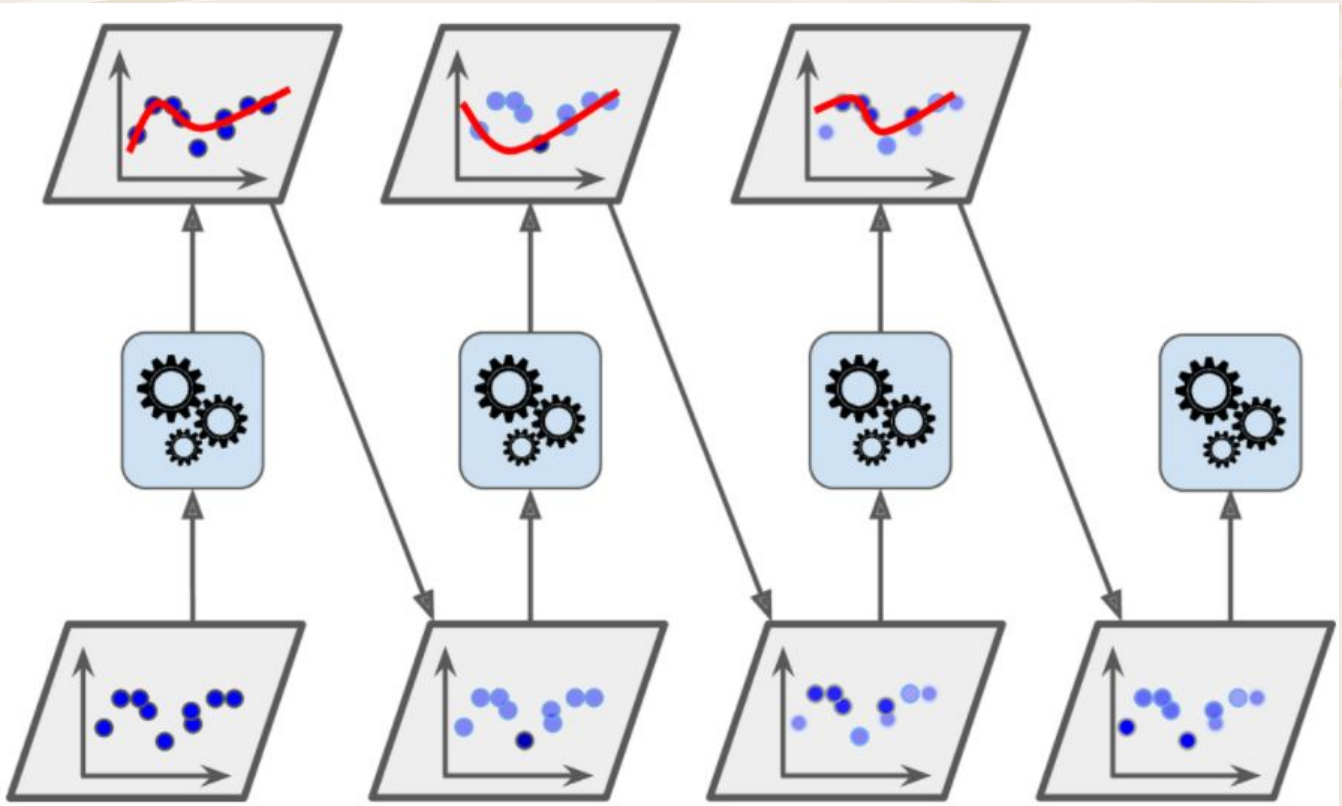


Figure 7-7. AdaBoost sequential training with instance weight updates

- The algorithm first trains a base classifier (e.g. Decision Tree) and uses it to make predictions on the training set.
- The algorithm then increases the relative weight of misclassified training instances.
- Then it trains a second classifier, using the updated weights, and again makes predictions on the training set, updates the instance weights, and so on (see Figure 7-7).

Boosting, Adaboost, Gradient Boosting, Stacking

- Figure 7-8 shows the decision boundaries of five consecutive predictors on the moon's dataset (in this example, each predictor is a highly regularized SVM classifier with an RBF kernel).
- The first classifier gets many instances wrong, so their weights get boosted. The second classifier therefore does a better job on these instances, and so on.
- The plot on the right represents the same sequence of predictors, except that the learning rate is halved (i.e., the misclassified instance weights are boosted half as much at every iteration).

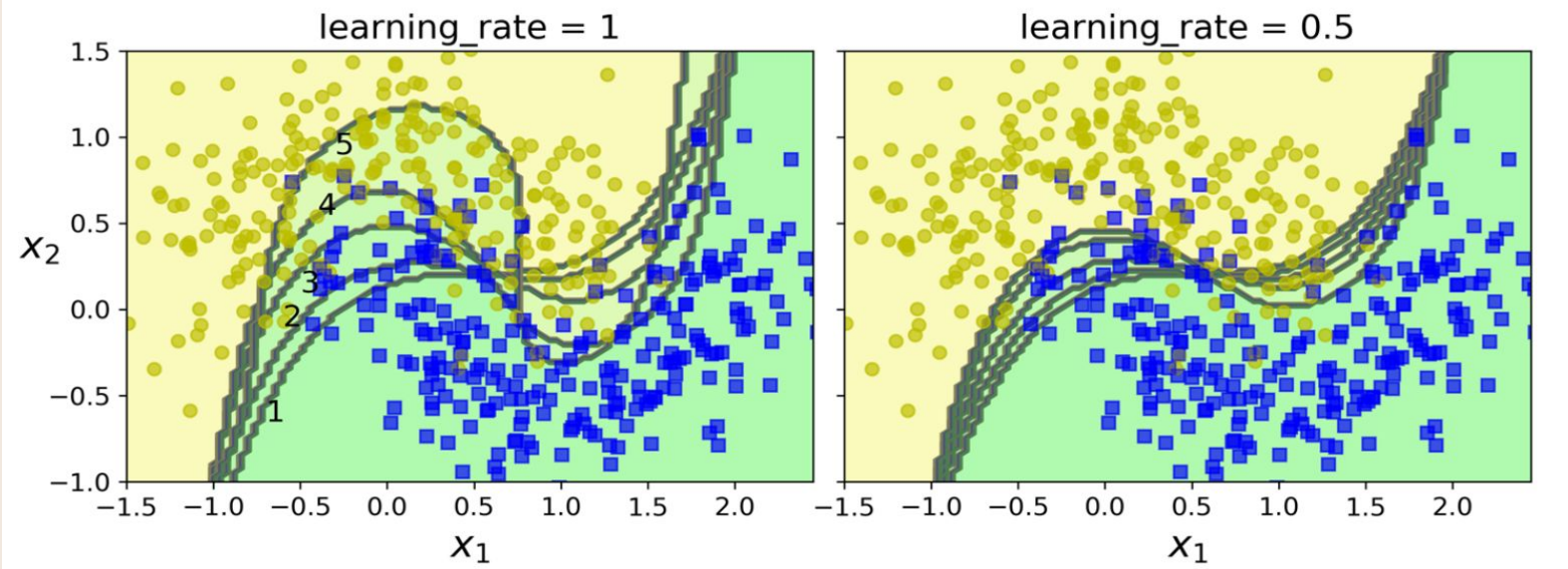


Figure 7-8. Decision boundaries of consecutive predictors

Boosting, Adaboost, Gradient Boosting, Stacking

- 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.
- 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, Gradient Boosting, Stacking

- Now look at the **AdaBoost** algorithm. Each instance weight w is initially set to 1/m. A first predictor is trained, and its weighted error rate r is computed on the training set; see Equation 7-1.

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

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

Boosting, Adaboost, Gradient Boosting, Stacking

- The predictor's weight α is then computed using Equation 7-2, where η is the learning rate hyperparameter. 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, then its weight will be negative.

Equation 7-2. Predictor weight

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

Boosting, Adaboost, Gradient Boosting, Stacking

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

Equation 7-3. Weight update rule

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

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

- Then all the instance weights are normalized

Boosting, Adaboost, Gradient Boosting, Stacking

- Finally, a new predictor is trained using the updated weights, and the whole process is repeated. The algorithm stops when the desired number of predictors is reached, or when a perfect predictor is found.
- To make predictions, **AdaBoost** simply computes the predictions of all the predictors and weighs them using the predictor weights α. The predicted class is the one that receives the majority of weighted votes (see Equation 7-4).

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, Gradient Boosting, Stacking

- Scikit-Learn uses a multiclass version of **AdaBoost** called **SAMME** (Stagewise Additive Modeling using a Multiclass Exponential loss function). When there are just two classes, **SAMME** is equivalent to **AdaBoost**.
- If the predictors can estimate class probabilities, 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, Gradient Boosting, Stacking

- 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.

Boosting, Adaboost, Gradient Boosting, Stacking

- 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 regularizing the base estimator.

Gradient Boosting, Stacking

- Another very popular boosting algorithm is **Gradient Boosting**. **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, Adaboost, Gradient Boosting, Stacking

- Now, we talk about **Gradient Boosted Regression Trees** (**GBRT**).
- First,let's fit a DecisionTreeRegressor to the training set (for example, a noisy quadratic training set):

```
from sklearn.tree import DecisionTreeRegressor

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

Boosting, Adaboost, Gradient Boosting, Stacking

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

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

Boosting, Adaboost, Gradient Boosting, Stacking

- 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)
```

Boosting, Adaboost, Gradient Boosting, Stacking

- 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))
```

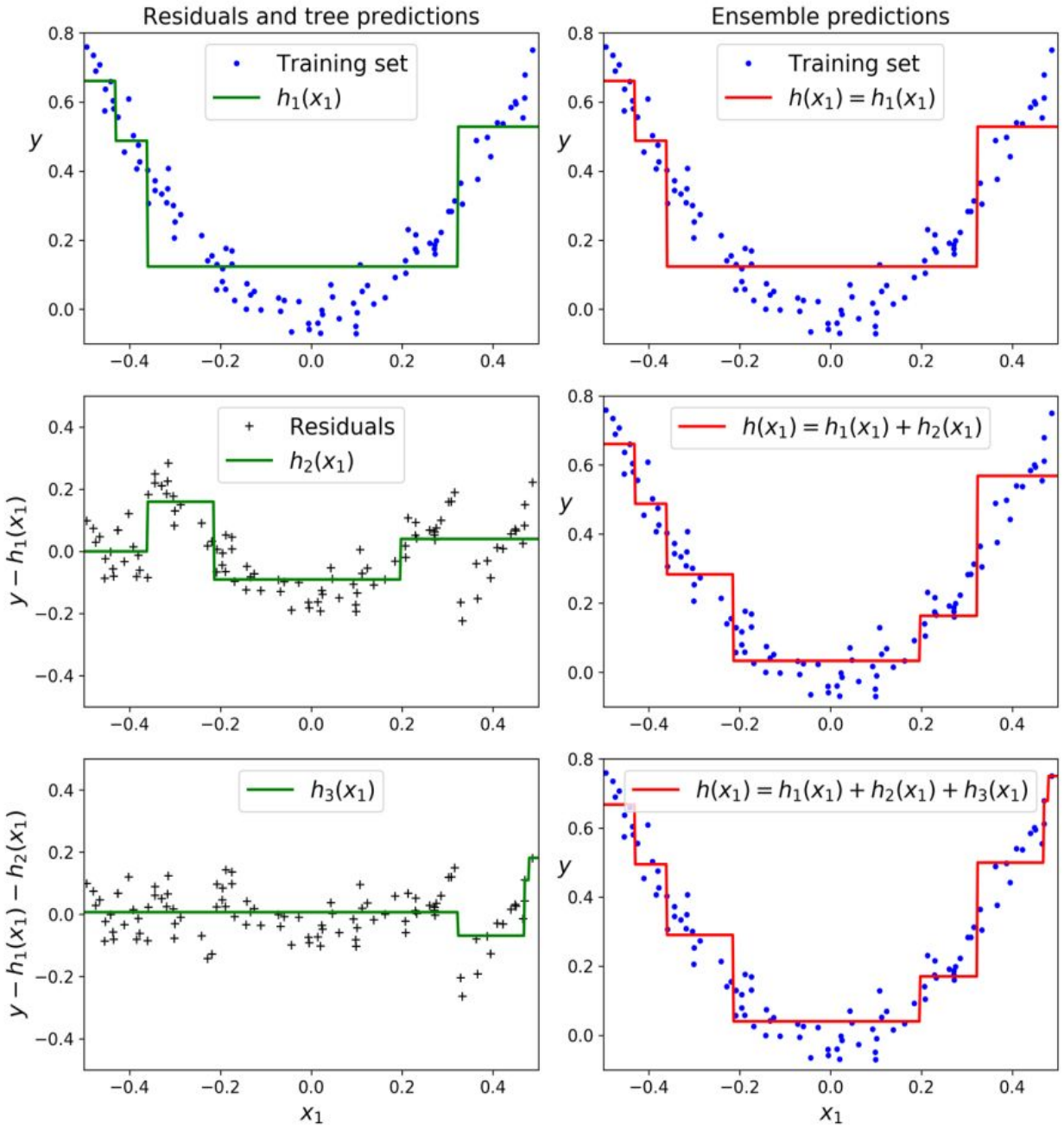


Figure 7-9. In this depiction of Gradient Boosting, the first predictor (top left) is trained normally, then each consecutive predictor (middle left and lower left) is trained on the previous predictor's residuals; the right column shows the resulting ensemble's predictions

Boosting, Adaboost, Gradient Boosting, Stacking

- Figure 7-9 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 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.

Boosting, Adaboost, Gradient Boosting, Stacking

- A simpler way to train **GBRT** ensembles is to use Scikit-Learn's GradientBoostingRegressor class. It has hyperparameters to control the growth of Decision Trees, as well as hyperparameters to control the ensemble training, such as the number of trees . 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)
```

Boosting, Adaboost, Gradient Boosting, Stacking

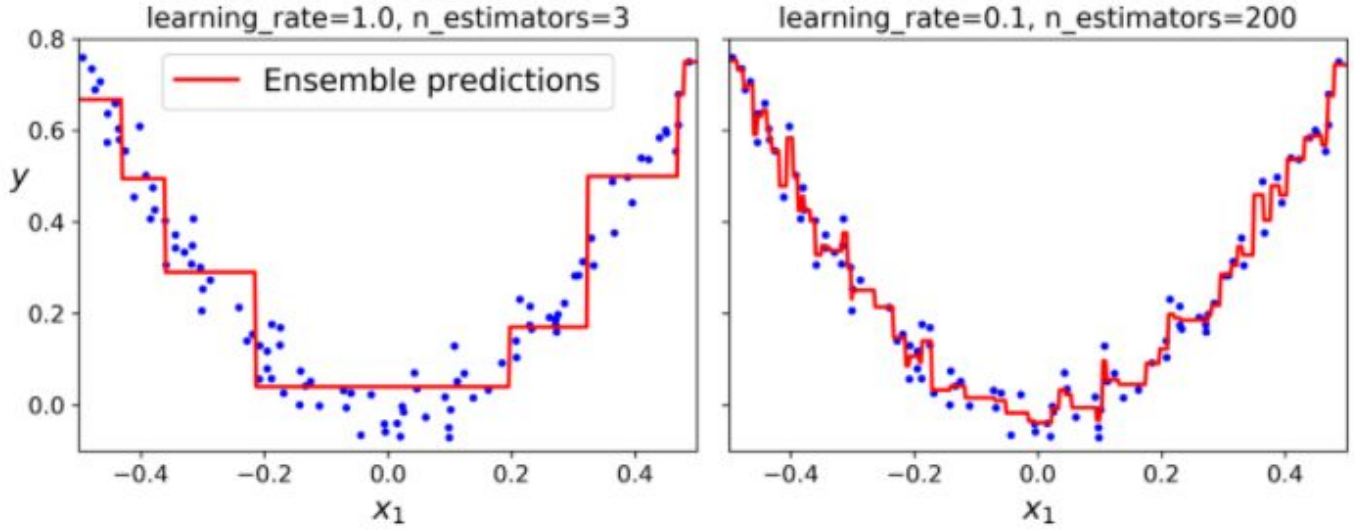


Figure 7-10. GBRT ensembles with not enough predictors (left) and too many (right)

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

Boosting, Adaboost, Gradient Boosting, Stacking

- 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.
- 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.

Boosting, Adaboost, Gradient Boosting, Stacking

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

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

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

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

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

The validation errors are represented on the left of Figure 7-11, and the best model's predictions are represented on the right.

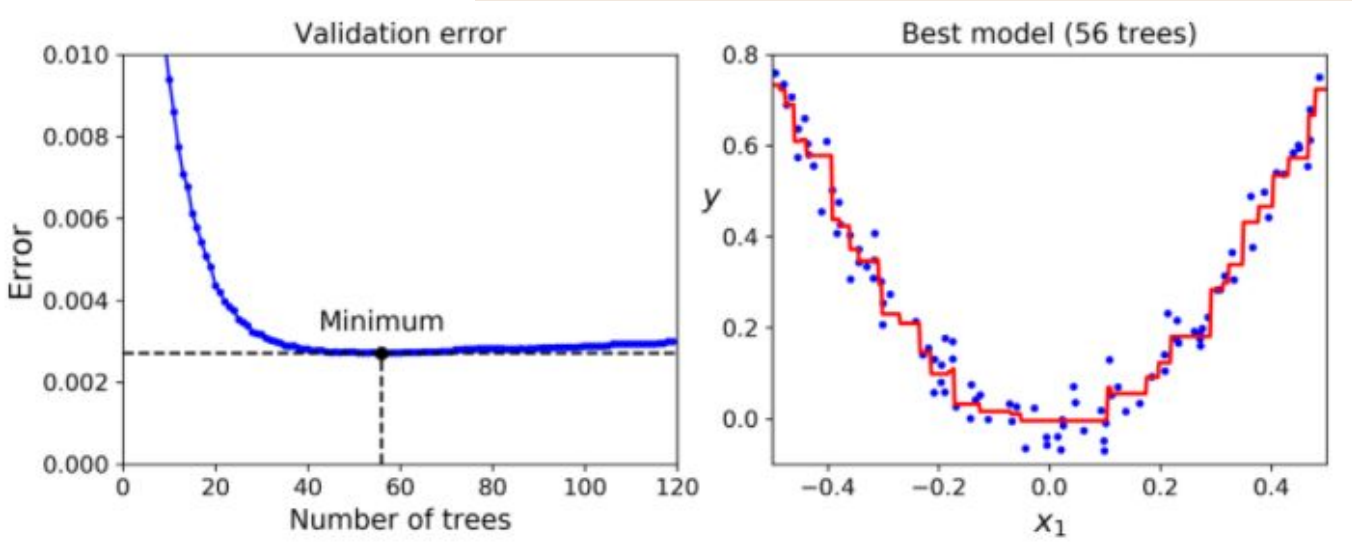


Figure 7-11. Tuning the number of trees using early stopping

Boosting, Adaboost, Gradient Boosting, Stacking

- 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.
- This technique *trades a higher bias for a lower variance*. It also speeds up training considerably.
  This is called **Stochastic Gradient Boosting**.

Boosting, Adaboost, Gradient Boosting, Stacking

- It is worth noting that an optimized implementation of **Gradient Boosting** is available in the popular Python library **XGBoost**, which stands for **Extreme Gradient Boosting**. This package aims to be extremely fast, scalable, and portable.
- In fact, **XGBoost** is often an important component of the winning entries in ML competitions. **XGBoost's API** is quite similar to Scikit-Learn's

```
import xgboost

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

Boosting, Adaboost, Gradient Boosting, Stacking

- **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)
```

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

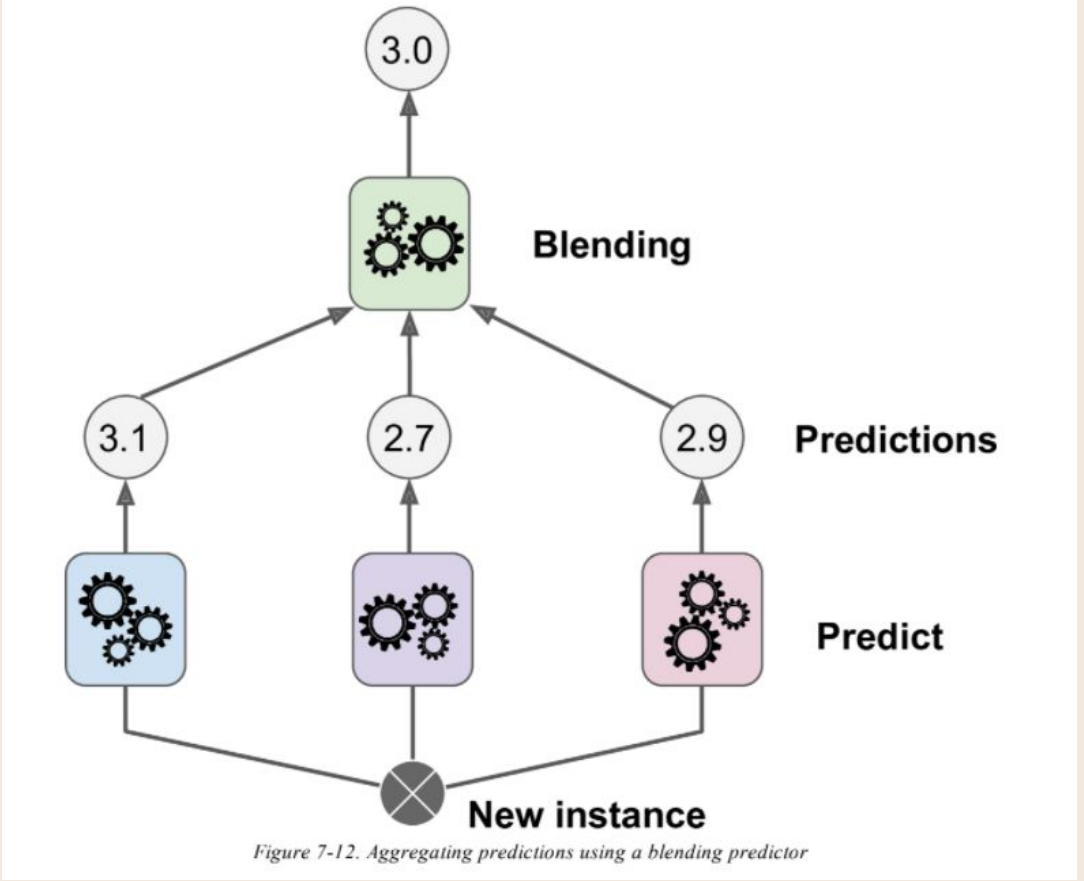


Figure 7-12. Aggregating predictions using a blending predictor

Boosting, Adaboost, Gradient Boosting, Stacking

- To train the blender, a common approach is to use a hold-out set.
- First, the training set is split into two subsets. The first subset is used to train the predictors in the first layer (see Figure 7-13).
- Next, the first layer's predictors are used to make predictions on the second set (see Figure 7-14). This ensures that the predictions are "clean," since the predictors never saw these instances during training.
- For each instance in the hold-out set, there are three predicted values. The blender is trained on this new training set, so it learns to predict the target value, given the first layer's predictions

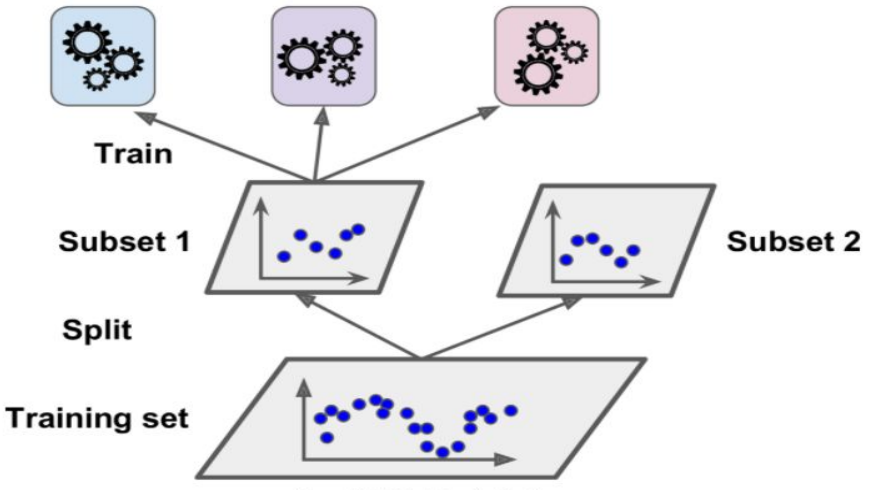


Figure 7-13. Training the first layer

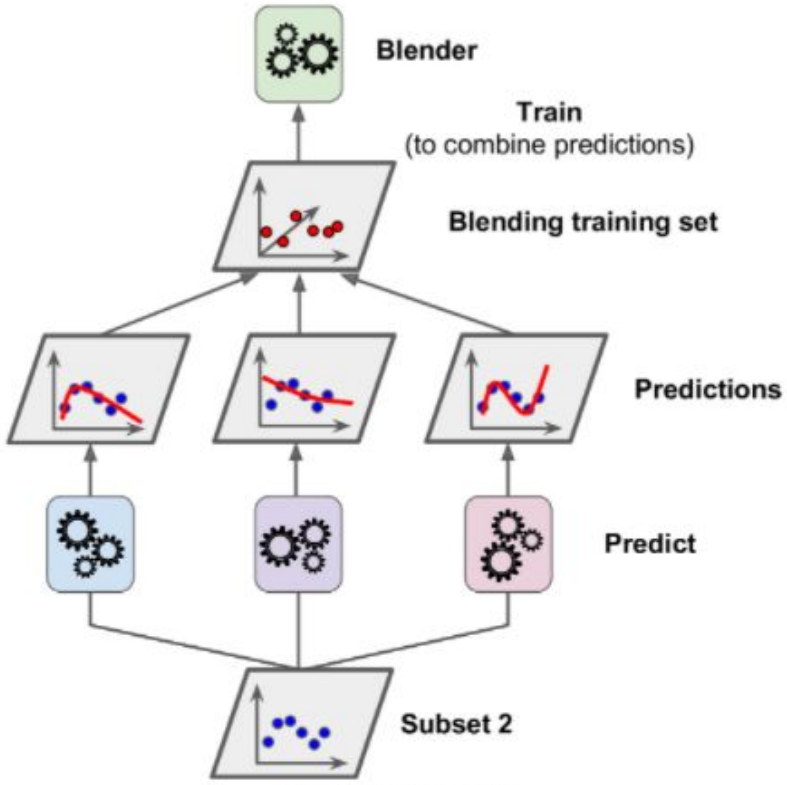


Figure 7-14. Training the blender

Boosting, Adaboost, Gradient Boosting, Stacking

- 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, and the third one is used to create the training set to train the third layer.
- Once this is done, we can predict a new instance by going through each layer sequentially, as shown in Figure 7-15.

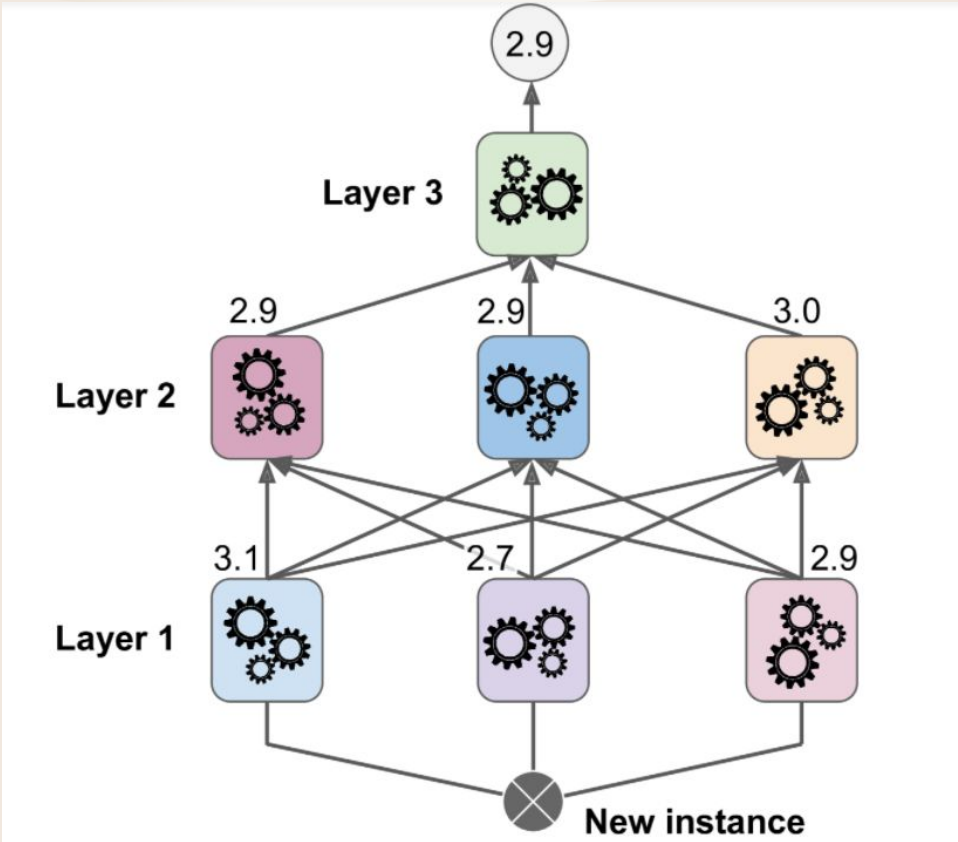


Figure 7-15. Predictions in a multilayer stacking ensemble

UNIVERSITAS
KRISTEN
MARANATHA