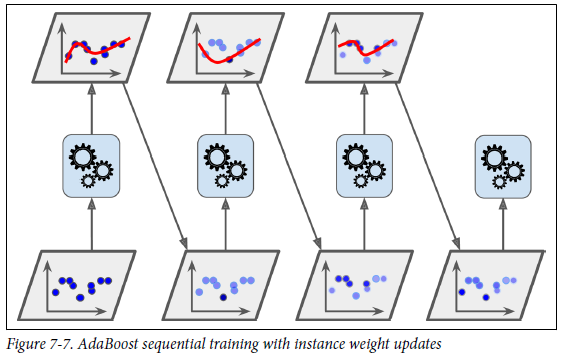
**Boosting:**

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 AdaBoost13 (short for Adaptive Boosting) and Gradient Boosting. Let’s start with Ada‐ Boost.

**AdaBoost:**

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

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



**Gradient Boosting**

Another very popular Boosting algorithm is Gradient Boosting.17 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.

**Gradient Tree Boosting, or Gradient Boosted Regression Trees (GBRT) are Decision tree based predictor in regression problem.**

from sklearn.tree import DecisionTreeRegressor

tree\_reg1 = DecisionTreeRegressor(max\_depth=2)

tree\_reg1.fit(X, y)

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

y2 = y - tree\_reg1.predict(X)

tree\_reg2 = DecisionTreeRegressor(max\_depth=2)

tree\_reg2.fit(X, y2)

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

y3 = y2 - tree\_reg2.predict(X)

tree\_reg3 = DecisionTreeRegressor(max\_depth=2)

tree\_reg3.fit(X, y3)

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

y\_pred = sum(tree.predict(X\_new) for tree in (tree\_reg1, tree\_reg2, tree\_reg3))

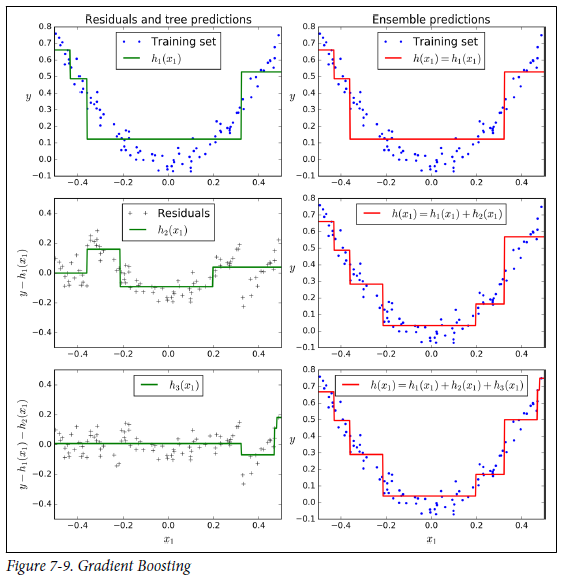


Figure 7-9 represents the predictions of these three trees in the left column, and the ensemble’s predictions in the right column. In the first row, the ensemble has just one tree, so its predictions are exactly the same as the first tree’s predictions. In the second row, a new tree is trained on the residual errors of the first tree. On the right you can

see that the ensemble’s predictions are equal to the sum of the predictions of the first two trees. Similarly, in the third row another tree is trained on the residual errors ofthe second tree. You can see that the ensemble’s predictions gradually get better as trees are added to the ensemble.

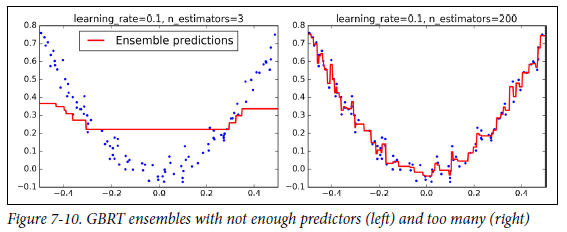
from sklearn.ensemble import GradientBoostingRegressor

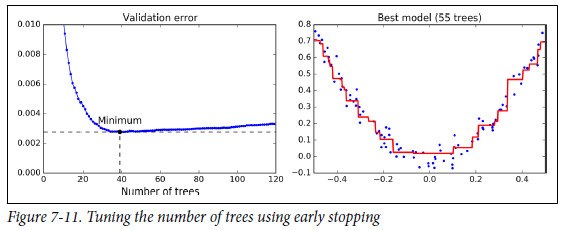
gbrt = GradientBoostingRegressor(max\_depth=2, n\_estimators=3, learning\_rate=1.0)

gbrt.fit(X, y)

**hyperparameters** to control the ensemble training, such as the **number of trees (n\_estimators), early stopping**.

The learning\_rate hyperparameter scales the contribution of each tree. If you set it to a low value, such as 0.1, you will need more trees in the ensemble to fit the training set, but the predictions will usually generalize better. This is a regularization technique called shrinkage.





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.

What are **hyperparameters**? — They are the parameters that are initialized before training a model because they cannot be learned from the algorithm. They control the behavior of the training algorithm and have a high impact on the performance of a model. The typical metaphor goes like this: hyperparameters are the knobs one turns to tweak a machine learning model. They are essential to optimization and to improve evaluation metrics.

XGBOOST from next page

Xgboost:

XGBoost is an optimized distributed gradient boosting library designed to be highly efficient, flexible and portable. It implements machine learning algorithms under the Gradient Boosting framework. **XGBoost provides a parallel tree boosting** (also known as GBDT, GBM) that solve many data science problems in a fast and accurate way. The same code runs on major distributed environment (Hadoop, SGE, MPI) and can solve problems beyond billions of examples.

**Categorical features not supported**

XGBoost does not support categorical features; if your data contains categorical features, load it as a NumPy array first and then perform [one-hot encoding](http://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.OneHotEncoder.html).

Import xgboost as xgb

The data is stored in a [DMatrix](https://xgboost.readthedocs.io/en/latest/python/python_api.html#xgboost.DMatrix) object.