Ensemble Learning and Random Forests

CSCI-P 556

ZORAN TIGANJ

Today

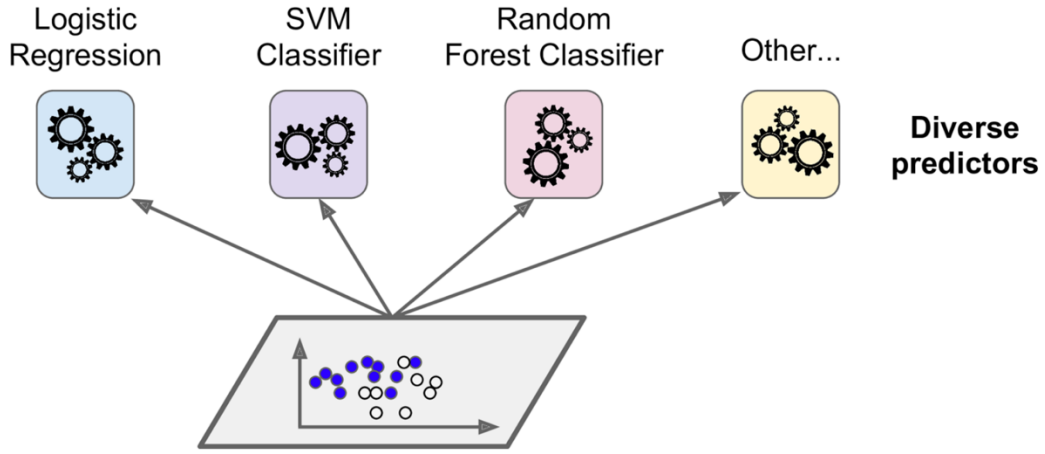
- Ensemble Learning and Random Forests
- Slides are based on Chapter 7 from the Hands on Machine Learning Textbook.

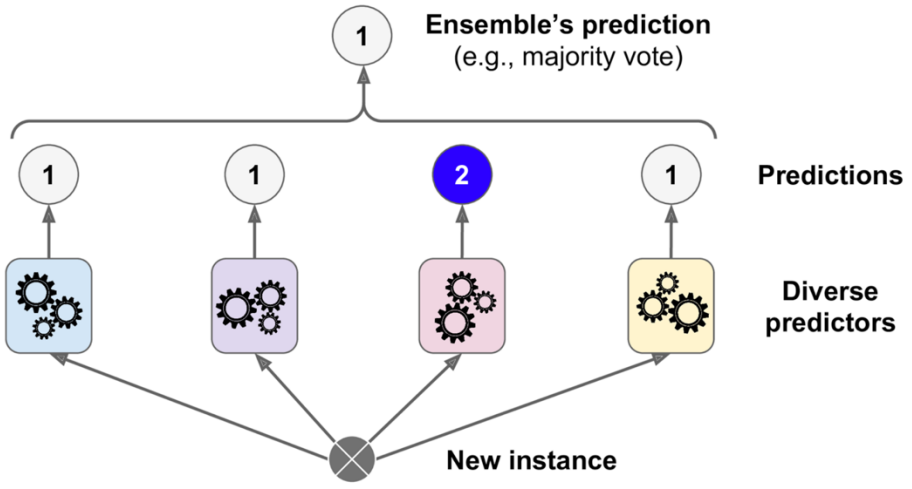
Ensemble Learning and Random Forests

- 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.**
  - For example, we can train a group of Decision Tree classifiers, each on a different random subset of the training set. To make predictions, we obtain the predictions of all the individual trees, then predict the class that gets the most votes.
  - Such an ensemble of Decision Trees is called a **Random Forest**, and despite its simplicity, this is one of the most powerful Machine Learning algorithms available today.
- People often use Ensemble methods near the end of a project, once they have already built a few good predictors, to combine them into an even better predictor.

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 Neighbor s classifier, and perhaps a few more
  - 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.**





Voting Classifiers - example

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

```
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)
```

```
>>> 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
```

Voting Classifiers – soft voting

- If all classifiers are able to 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.

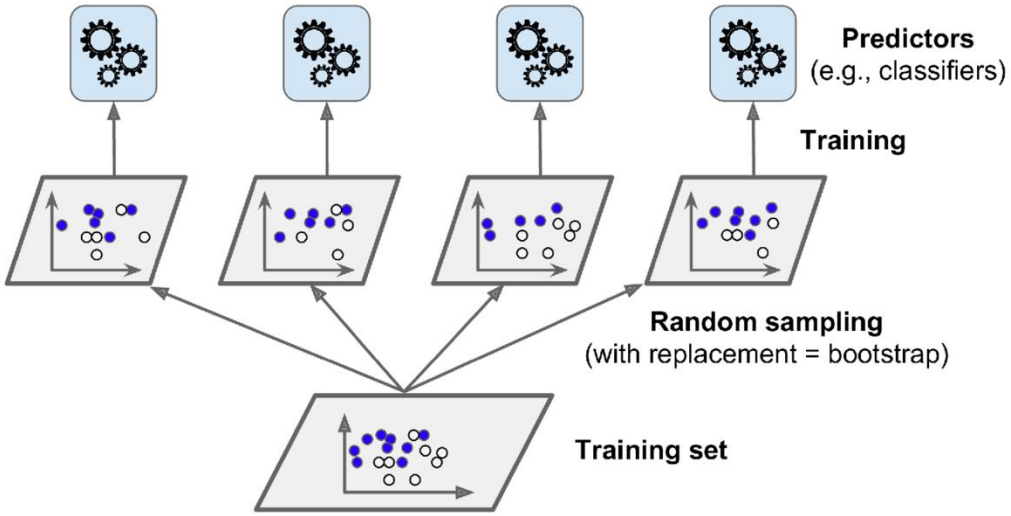
```
log_clf = LogisticRegression()
rnd_clf = RandomForestClassifier()
svm_clf = SVC()

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

soft

With hard voting we get accuracy of 0.904
With soft voting we get accuracy of 0.912

Bagging and Pasting

- To get a diverse set of classifiers we can use the same training algorithm for every predictor and 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

- 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.

THE BIAS/VARIANCE TRADE-OFF

An important theoretical result of statistics and Machine Learning is the fact that a model's generalization error can be expressed as the sum of three very different errors:

Bias

This part of the generalization error is due to wrong assumptions, such as assuming that the data is linear when it is actually quadratic. A high-bias model is most likely to underfit the training data.[8]

Variance

This part is due to the model's excessive sensitivity to small variations in the training data. A model with many degrees of freedom (such as a high-degree polynomial model) is likely to have high variance and thus overfit the training data.

Irreducible error

This part is due to the noisiness of the data itself. The only way to reduce this part of the error is to clean up the data (e.g., fix the data sources, such as broken sensors, or detect and remove outliers).

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 trade-off.

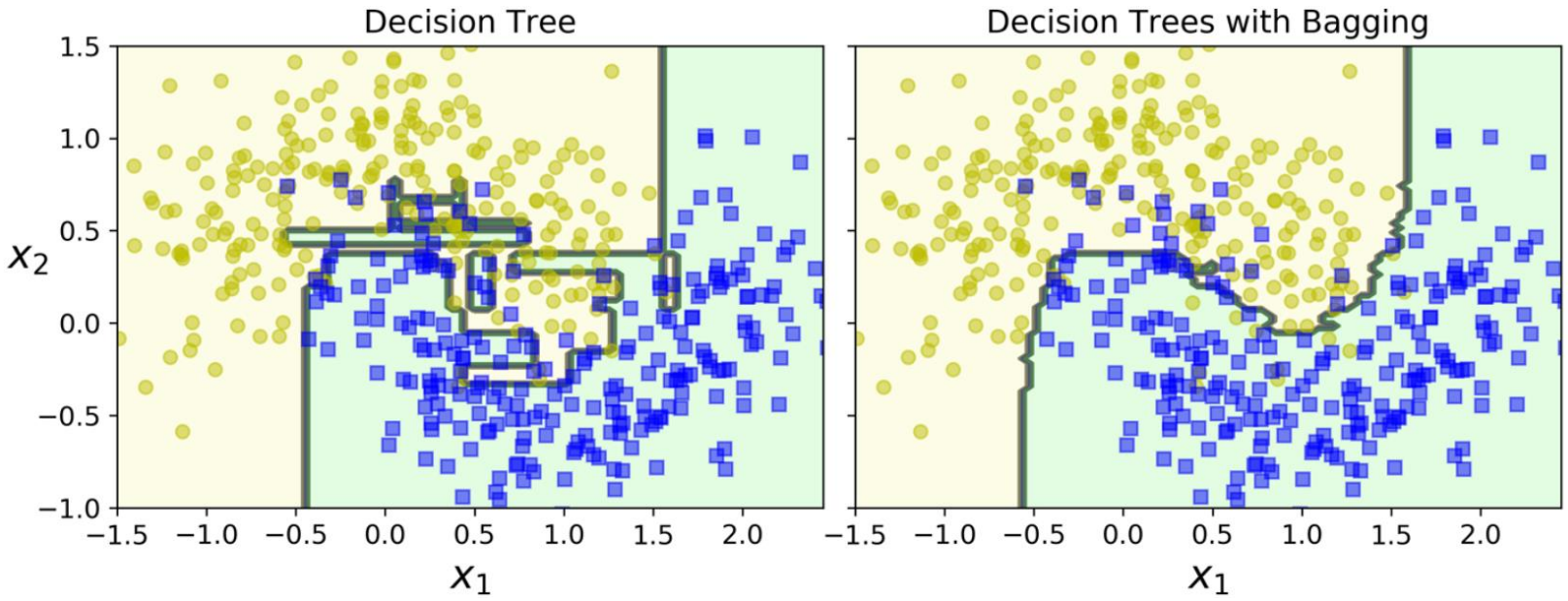
Bagging and Pasting in Scikit-Learn

- 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 (this is an example of bagging, but if you want to use pasting instead, just set bootstrap=False).

```
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)
```

Bagging and Pasting in Scikit-Learn

- 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).



Random Forests

- 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.

```
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 Forests: Extra-Trees

- When you are growing a tree in a Random Forest, at each node only a random subset of the features is considered for splitting.
- It is possible to make trees even more random by also using random thresholds for each feature rather than searching for the best possible thresholds.
- A forest of such extremely random trees is called an Extremely Randomized Trees ensemble (or Extra-Trees for short).

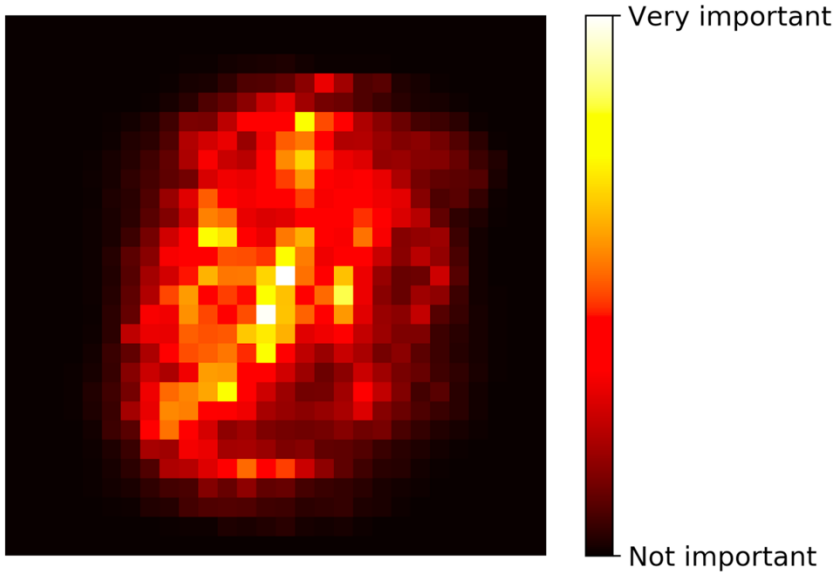
Random Forests: Feature Importance

- Random Forests can measure the relative importance of each feature.
- Scikit-Learn measures a feature's importance by looking at how much the tree nodes that use that feature reduce impurity on average (across all trees in the forest).

```
>>> 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
```

Random Forests: Feature Importance

- Importance of each pixel for Random Forest classifier trained on the MNIST dataset:



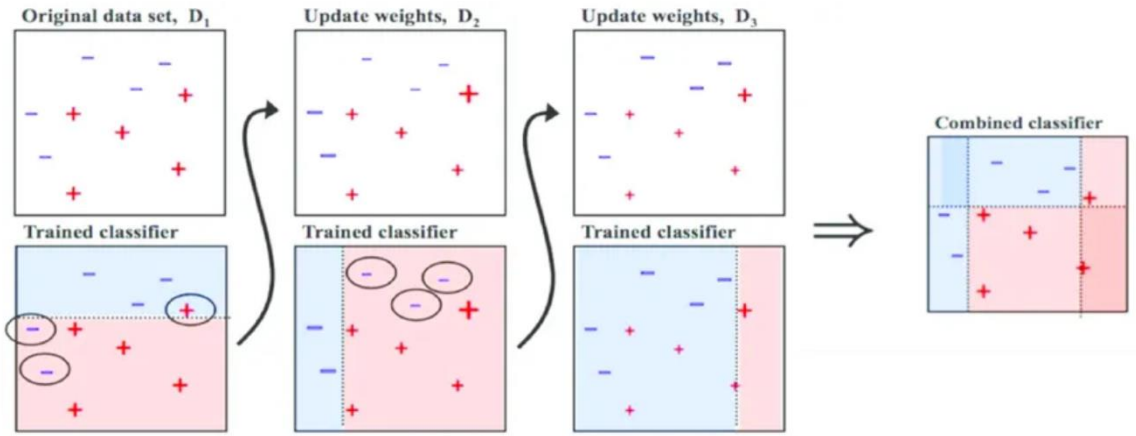
- Random Forests are very handy to get a quick understanding of what features actually matter, in particular if you need to perform feature selection.

Boosting

- Boosting 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**.

AdaBoost

- AdaBoost: new predictor corrects its predecessor by focusing on training instances that the predecessor underfitted.
- For example, when training an AdaBoost classifier, the algorithm first trains a base classifier (such as a 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.



https://towardsdatascience.com/understanding-adaboost-for-decision-tree-ff8f07d2851

Gradient Boosting

- Gradient Boosting tries to fit the new predictor to the residual errors made by the previous predictor.
- Example: 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)
```

- 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)
```

Gradient Boosting

- 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))
```

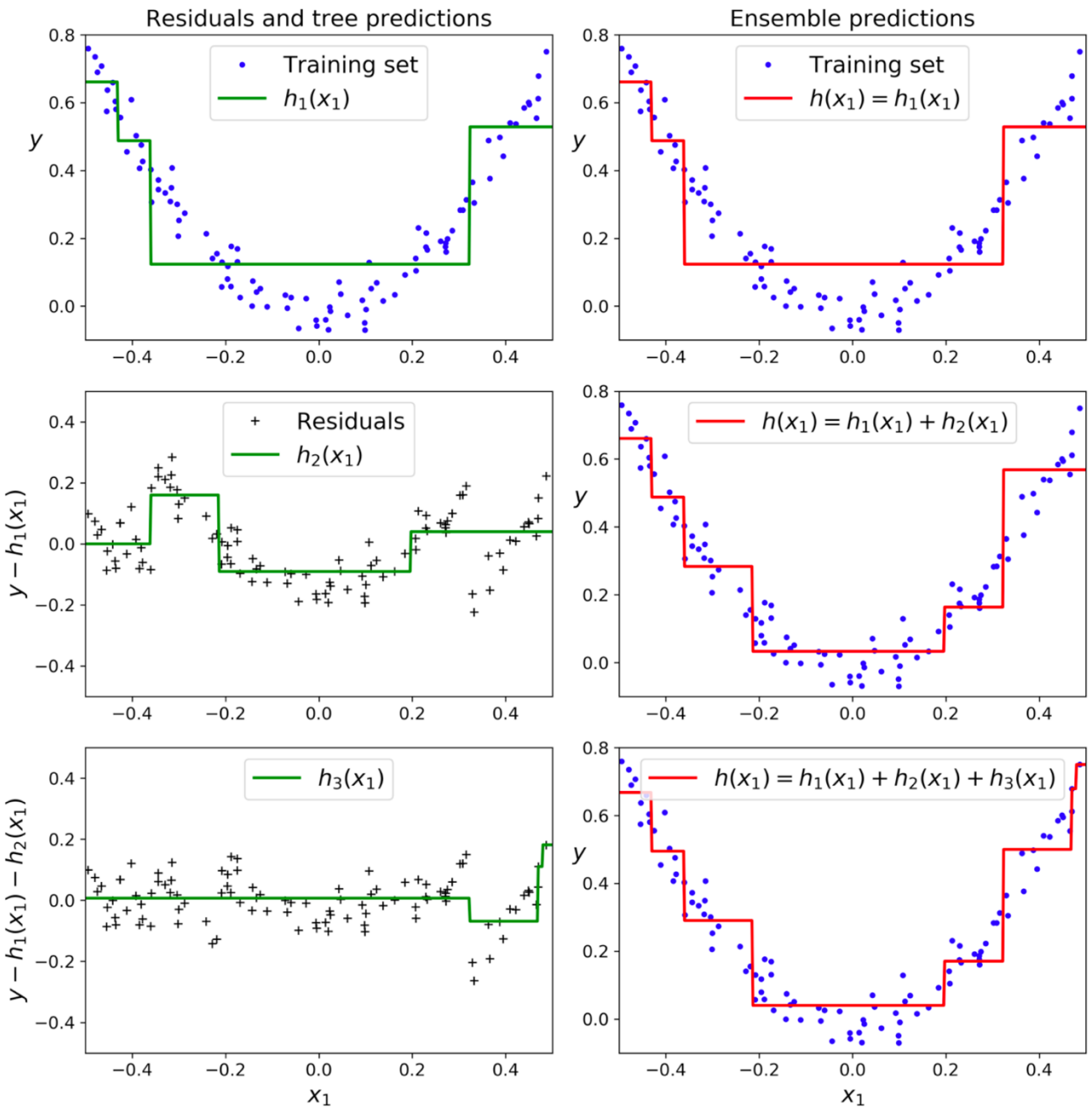


Figure 7-9. In this depiction of Gradient Boosting, the first predictor (top left) is trained normally, then each consecutive predictor (middle left and lower left) is trained on the previous predictor's residuals; the right column shows the resulting ensemble's predictions

Gradient Boosting

- How many trees to use?

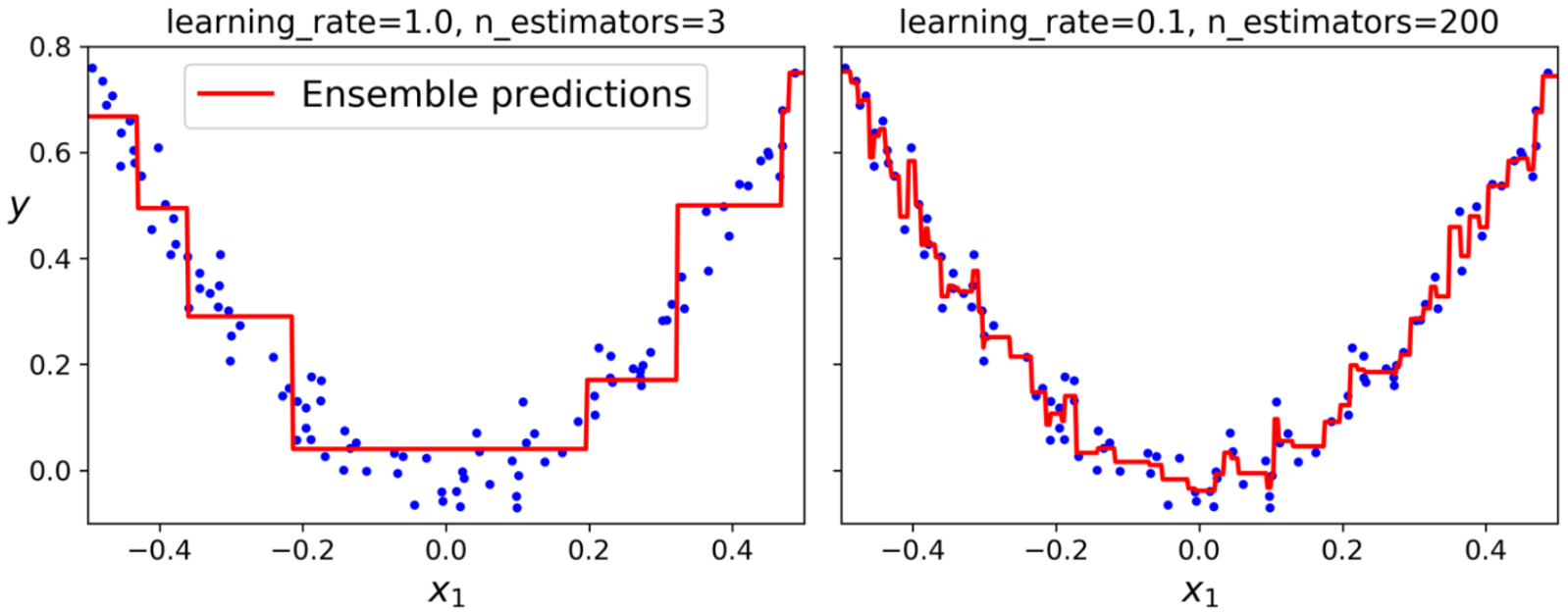
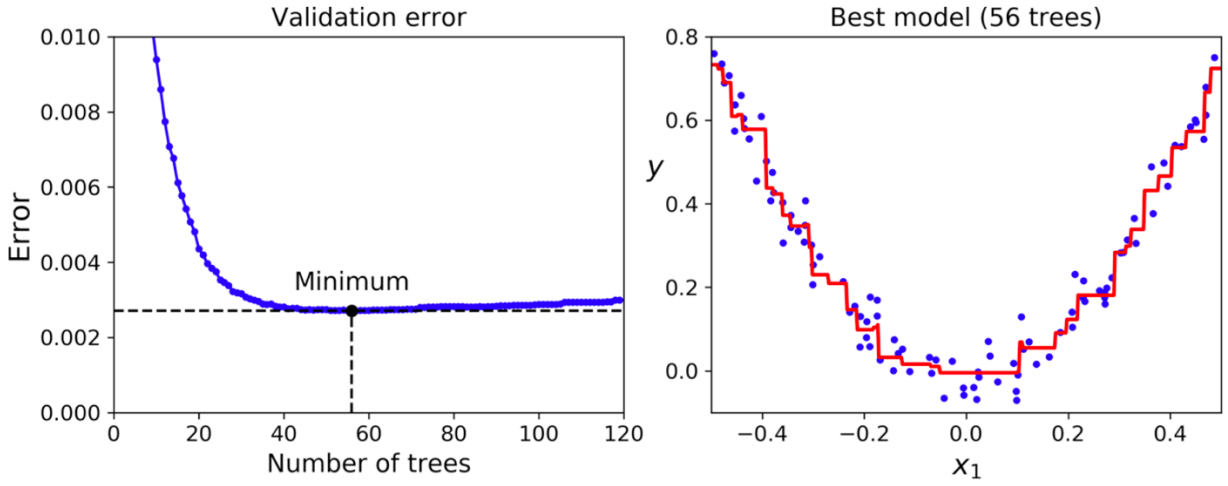


Figure on the left does not have enough trees to fit the training set, while the one on the right has too many trees and overfits the training set.

Gradient Boosting

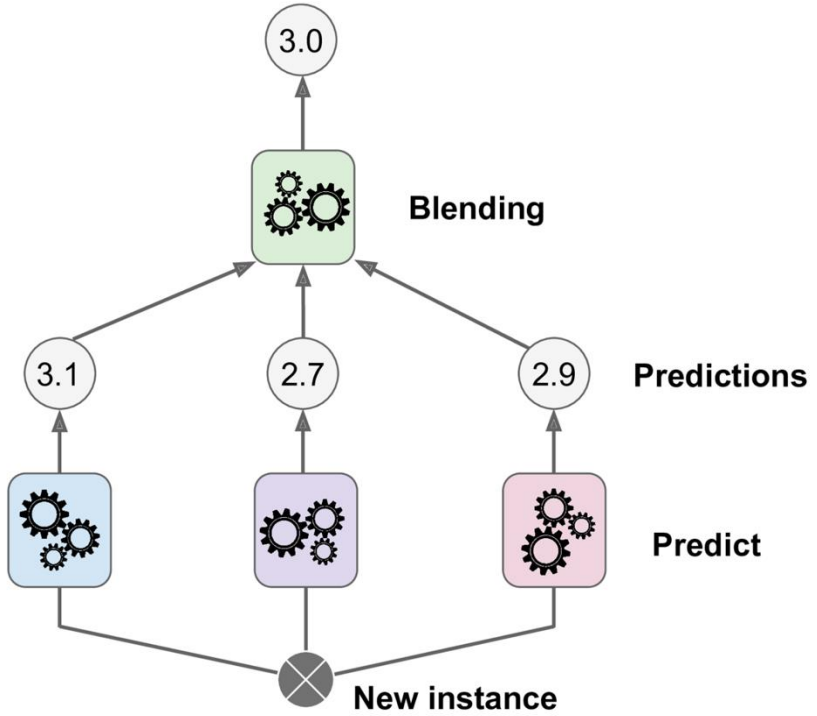
- In order to find the optimal number of trees, you can use early stopping

```
gbrt = GradientBoostingRegressor(max_depth=2, warm_start=True)

min_val_error = float("inf")
error_going_up = 0
for n_estimators in range(1, 120):
    gbrt.n_estimators = n_estimators
    gbrt.fit(X_train, y_train)
    y_pred = gbrt.predict(X_val)
    val_error = mean_squared_error(y_val, y_pred)
    if val_error < min_val_error:
        min_val_error = val_error
        error_going_up = 0
    else:
        error_going_up += 1
        if error_going_up == 5:
            break  # early stopping
```



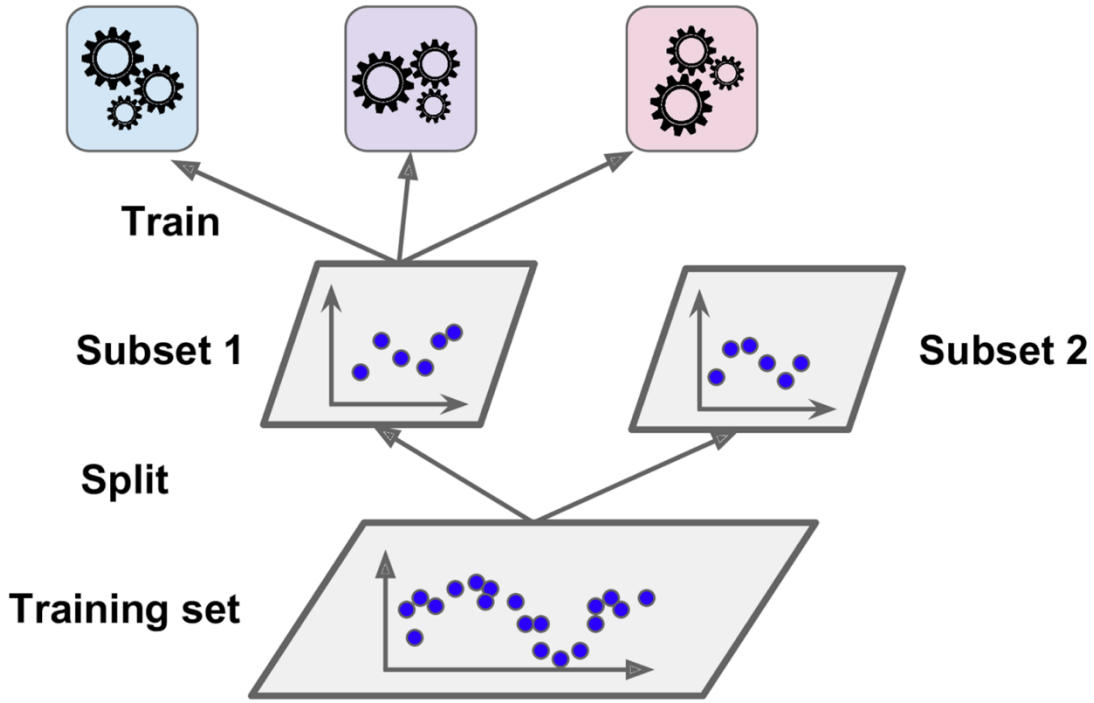
Stacking

- Instead of using trivial functions (such as hard voting) to aggregate the predictions of all predictors in an ensemble, with **stacking** we train a model to perform this aggregation.

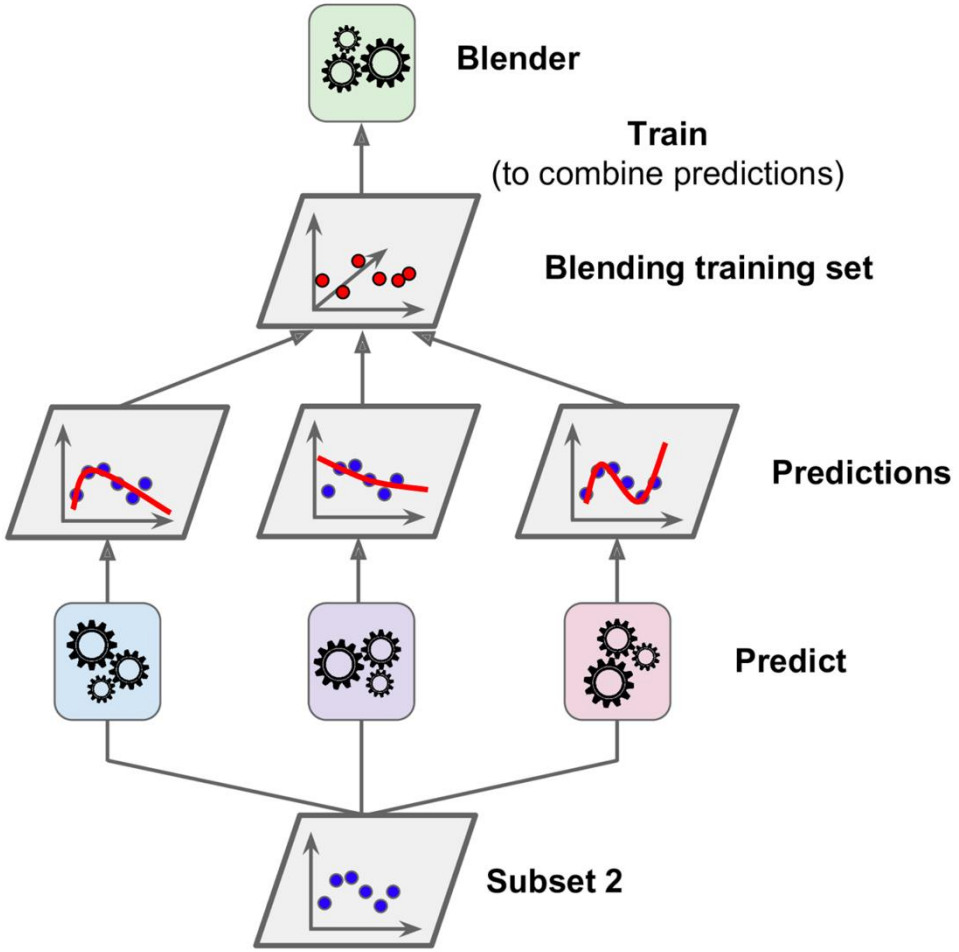


Stacking

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

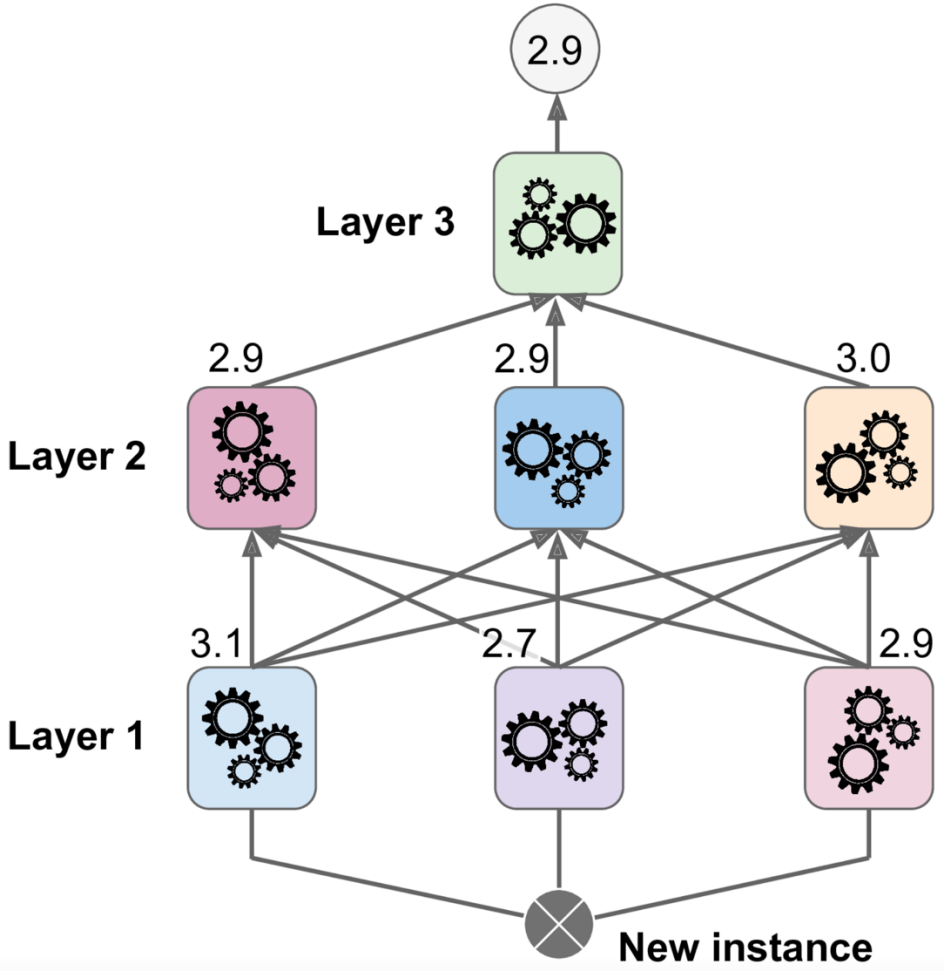


Stacking


Blender
Train
(to combine predictions)
Blending training set
Predictions
Predict
Subset 2

Stacking

- It is actually possible to train several different blenders this way (e.g., one using Linear Regression, another using Random Forest Regression), to get a whole layer of blenders.
- The trick is to split the training set into three subsets



Next time

- Dimensionality reduction, from Chapter 8, *Hands on machine learning textbook*