CITS5508 Machine Learning

Débora Corrêa (Unit Coordinator and Lecturer)

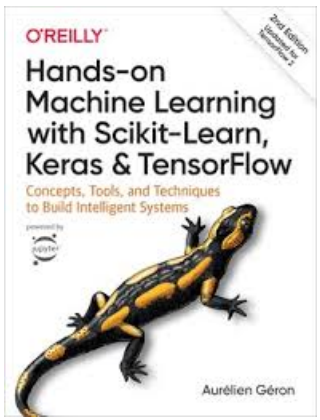
2023

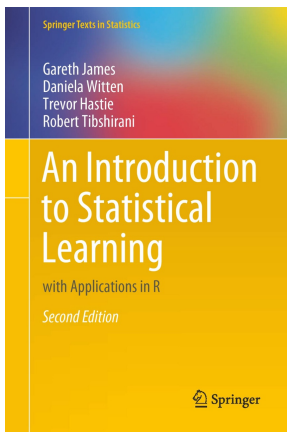
Today

Ensemble Learning and Random Forests.

Hands-on Machine Learning with Scikit-Learn & TensorFlow (chapter 7)

An Introduction to Statistical Learning (chapter 8)





Chapter Seven

We will look at how different classifiers can be combined to improve the performance of the algorithm. Here are the main topics we will cover:

- Voting Classifiers
- Bagging and Pasting
- Random Forests
- Boosting
- Stacking

Ensemble Learning

The idea of *ensemble learning* is to combine the predictions of a group of predictors (such as classifiers or regressors) to obtain a single and potentially better model than by using the best individual model.

- A group of predictors is called an *ensemble*.
- This technique is therefore called *ensemble learning*.
- An ensemble learning algorithm is called an e*nsemble method*.
- For example, a *Random Forest* is an ensemble of Decision Trees. Each tree is trained using a different random subset of the training set. Then, obtain the predictions of all the individual trees, and the class that gets the most votes is the ensemble's prediction.

Resampling Methods

Resampling methods involve repeatedly drawing samples from a training set and refitting a model of interest on each sample in order to obtain additional information about the fitted model. Two of the most commonly used resampling methods are cross-validation and the bootstrap.

Training error rate: proportion of mistakes that are made if we apply our model to the training observations. It is computed on the data that was used to train the classifier.

Test error rate: proportion of mistakes that are made if we apply our model to a new set of data known as the test data.

Resampling Methods

We want a model that produces a small test error and has lower variability.

The decision trees are said to be high variance models. This means that fitting a decision tree on different random splits of the training data will produce results that are quite different.

Low variance would imply that the results are similar.

We will look at general-purpose ensemble methods for reducing the variance, such as bootstrap aggregation (bagging), and random forests.

Voting Classifiers

Suppose that you have trained a number of classifiers, each one achieving about 80% accuracy.

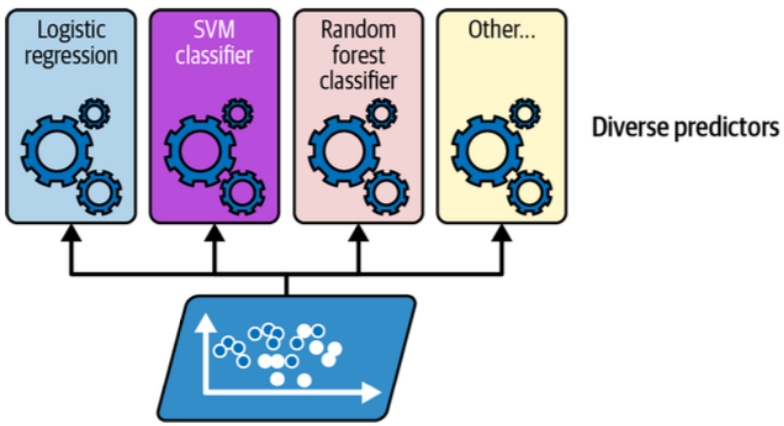


Figure 7-1. Training diverse classifiers

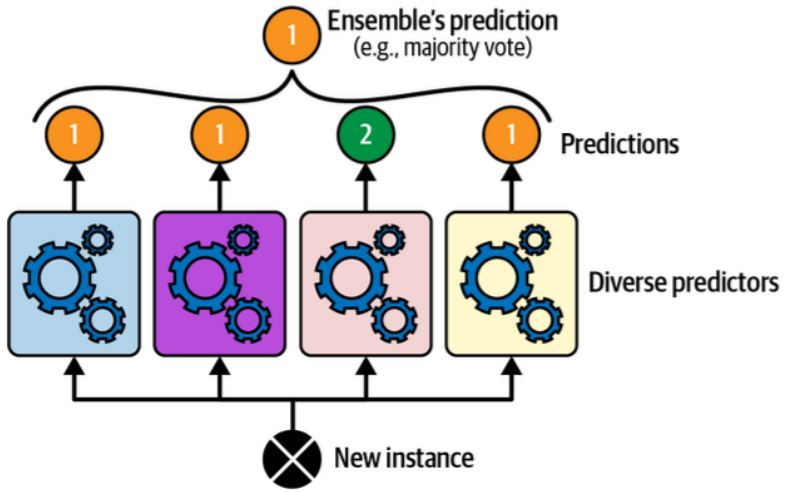


Figure 7-2. Hard voting classifier predictions

You can aggregate the predictions of the classifiers and predict the class that gets the most votes.

This *majority-vote classifier* is called a *hard voting classifier*.

Voting Classifiers

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.

That is, the ensemble can be a strong learner, provided that there are enough weak learners and they are sufficiently diverse.

Here, diversity means that the classifiers are independent and make different types of error.

One way to get diverse classifiers is to train them using very different algorithms.

A Voting Classifier Example

```
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)
```

A Voting Classifier Example

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

```
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
```

The voting classifier's predict() method will perform hard voting

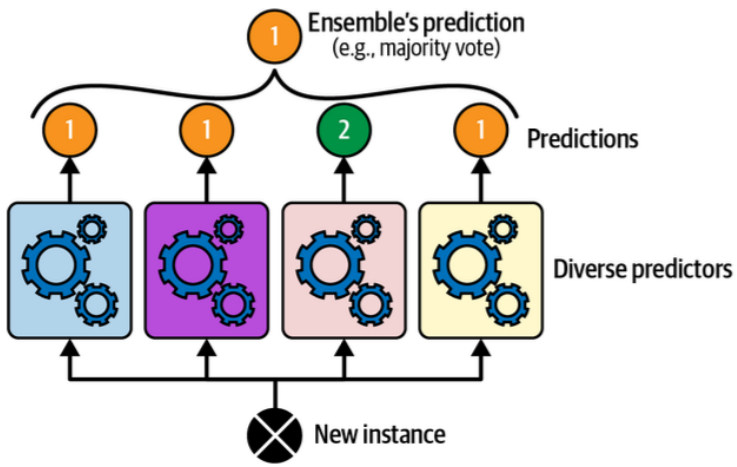
```
>>> voting_clf.predict(X_test[:1])
array([1])
>>> [clf.predict(X_test[:1]) for clf in voting_clf.estimators_]
[array([1]), array([1]), array([0])]
```

Finally, the performance of the voting classifier on the test set:

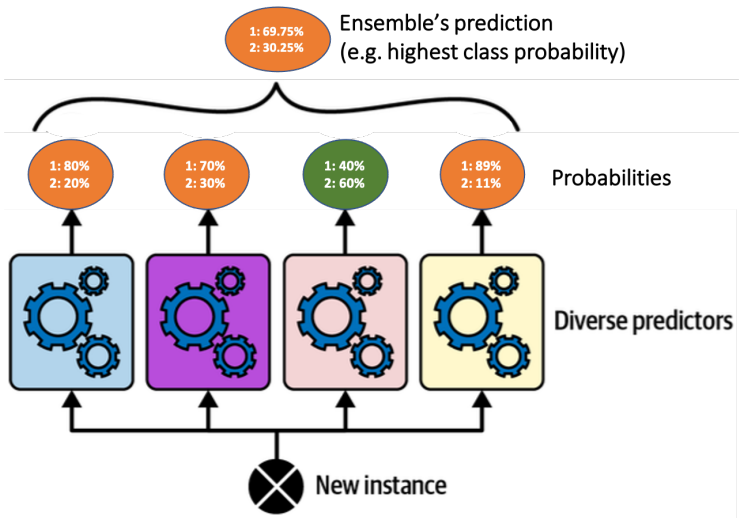
```
>>> voting_clf.score(X_test, y_test)
0.912
```

A Voting Classifier Example

We can implement a *soft voting classifier* if all the classifiers are able to estimate the class probabilities (i.e., they have a `predict_proba()` method), and then predict the class with the highest class probability, averaged over all the classifiers.



Hard voting classifier predictions



Soft voting classifier predictions

Hard voting – Ensemble's predicted class is the majority voted class. *Soft voting* – Ensemble's predicted class is the class that receives the highest average class probability.

Bagging and Pasting

The idea of this approach is to use the same training algorithm for every predictor but train them on different random subsets of the training set.
This means that we need to draw random samples from the (single) training set.

- Sampling *with* replacement – this method is called *bagging*[1] (short for bootstrap aggregating[2])
- Sampling *without* replacement – this method is called *pasting*

(Note that we use the term "predictor" interchangeably to represent "regressors" or "classifiers")

[1] "Bagging Predictors," L. Breiman (1996)
[2] In statistics, resampling with replacement is called bootstrapping.

Bootstrapping

Bootstrapping is an approach to emulate the process of obtaining new sample sets and estimate model parameters (without generating additional samples).

The procedure is:

- Select randomly from the data m observations with replacement, where m is the size of the training set.
- Use that sample to fit the model and estimate its parameters.
- Repeat this B times (e.g. B is the number of predictors).
- Aggregate the predictions of all predictors.

Bagging (general method, based on the same principles as bootstrapping) is particularly good for reducing the variance and so it is often applied to decision trees.

Bagging and Pasting

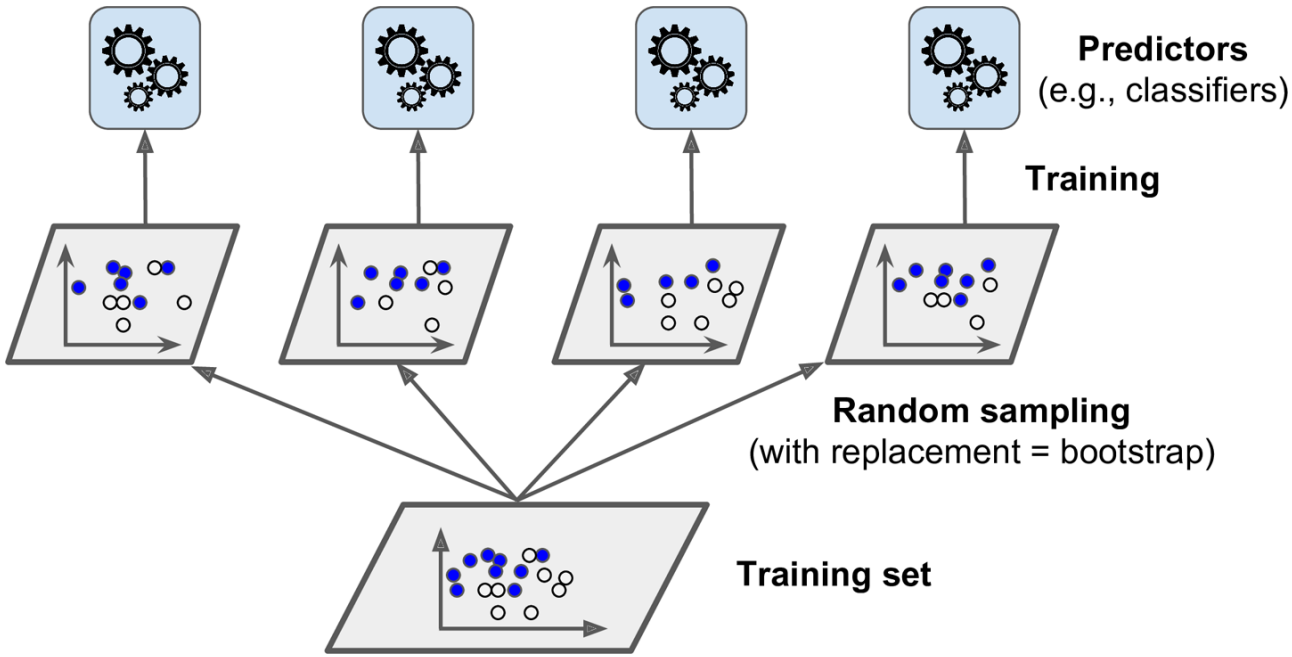


Figure 7-4. Bagging and pasting involve training several predictors on different random samples of the training set

Once all the predictors are trained, the ensemble can make a prediction for a new instance by aggregating the predictions from all the predictors. Typically, the *statistical mode* (i.e., the most frequent prediction) is used as the aggregation function for classification (or the average of the resulting predictions for regression).

Bagging for Decision Trees

Bagging is particularly useful for decision trees. We simply construct B regression or classification trees using B bootstrapped training sets. These trees are grown deep, and are not pruned. Hence each individual tree has high variance and low bias.

The procedure for constructing bagged decision tress is:

- Create B bootstrapped training sets
- For each training set fit a decision tree without pruning
- For each instance in the test set calculate the prediction for each of the B decision trees
- Average the predictions or take the majority vote.

In general, the net result is: the ensemble has a similar bias but a lower variance than a single predictor trained on the original whole training set.

Bagging and Pasting in Scikit-Learn

The `BaggingClassifier` class (or `BaggingRegressor` for regression) of Scikit-Learn implements bagging and pasting.

In the example code below, we design an ensemble to have 500 predictors, each trained on 100 randomly drawn samples with replacement (i.e., bagging):

```
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)
```

To use sampling without replacement (i.e., pasting), set `bootstrap=False`.

By default, `BaggingClassifier` automatically performs soft voting if the base classifier can estimate class probabilities.

Decision Boundary

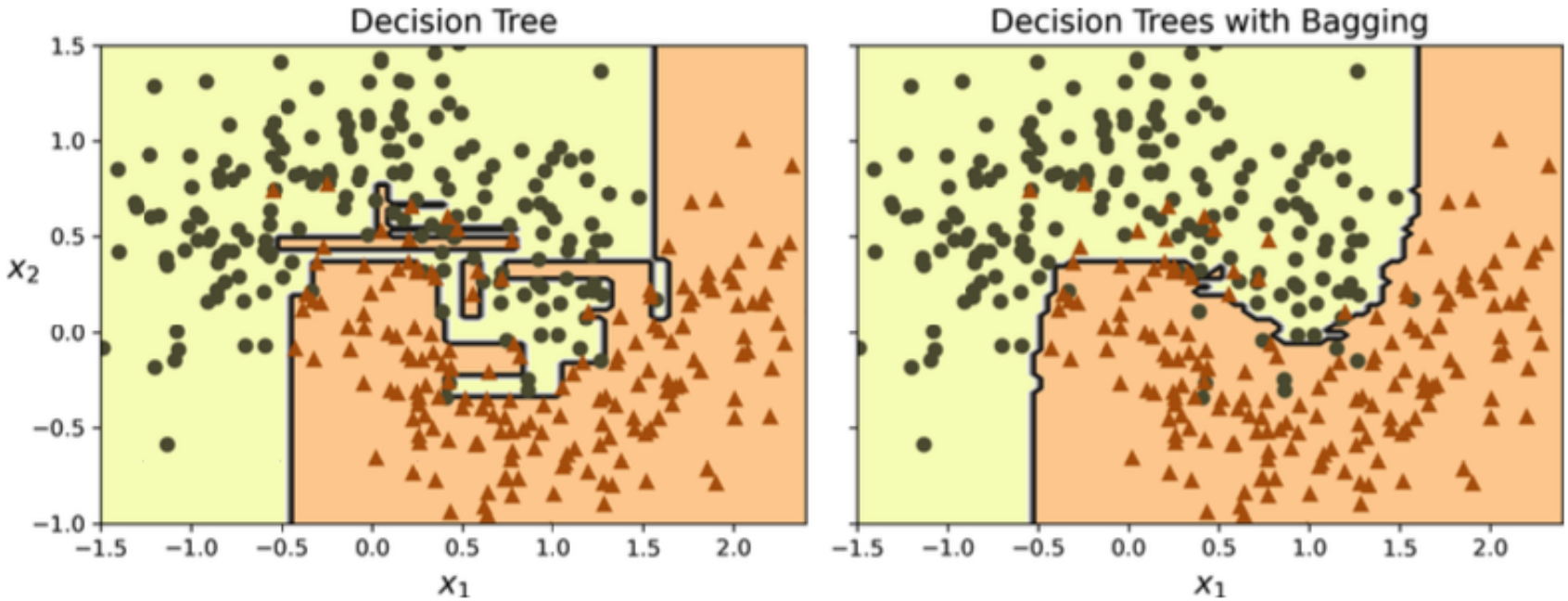


Figure 7-5. A single Decision Tree (left) versus a bagging ensemble of 500 trees (right)

Comparison of the decision boundary on the `moons` 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.

Out-of-Bag Evaluation

By default, a `BaggingClassifier` samples m training instances with replacement (i.e., option `bootstrap=True`).

Each time a bootstrap sample is selected some observations will be omitted from the training set (remember we sample with replacement). Therefore, with bagging, some instances may be sampled several times while others may not be sampled at all.

On average, only about 63% of the training instances are sampled for each predictor. The remaining 37% that are not sampled are called *out-of-bag* (*oob*) instances. Note that they are not the same 37% for all predictors.

Out-of-Bag Evaluation

Since a predictor never sees the oob instances during training, we can use these instances to form our validation set.

Therefore, we can predict a response for the ith observation using each of the trees in which that observation was out-of-bag.

Then, average the responses (regression) or take a majority vote (classification). This leads to a single oob prediction for the ith observation.

An oob prediction can be obtained in this way for each of the m observations, from which the overall oob MSE or classification error can be computed.

Out-of-Bag Evaluation in Scikit-Learn

For an automatic oob evaluation after training, we can set the option `oob_score=True` when creating a `BaggingClassifier`:

```
>>> 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
```

And the oob decision function via

```
>>> bag_clf.oob_decision_function_[:3] # probas for the first 3 instances
array([[0.32352941, 0.67647059],
       [0.3375    , 0.6625    ],
       [1.        , 0.        ]])
```

We can also verify the accuracy on the test set:

```
>>> from sklearn.metrics import accuracy_score
>>> y_pred = bag_clf.predict(X_test)
>>> accuracy_score(y_test, y_pred)
0.92
```

Random Forests

One of the problems with bagging (and bootstrapping in general) is that the trees are correlated. Random forests have a small tweak to the method to decorrelate the trees.

The tweak is to force each split to consider only a random subset of the features from the full set of features.

Typically, $\ell = \sqrt{n}$ features (where n is the total number of features) are selected. That is, the majority of the available features are not considered at each step.

The algorithm results in greater tree diversity, which (again) trades a higher bias for a lower variance, generally yielding an overall better model.

Random Forests - Rationale

Consider a data set with a very strong feature and a number of other moderately strong features. The collection of bagging trees will use the strong feature in the top split giving the set of trees some common characteristics (making the trees highly correlated).

Averaging (or taking the majority vote of) many highly correlated quantities does not lead to as large of a reduction in variance as averaging many uncorrelated quantities.

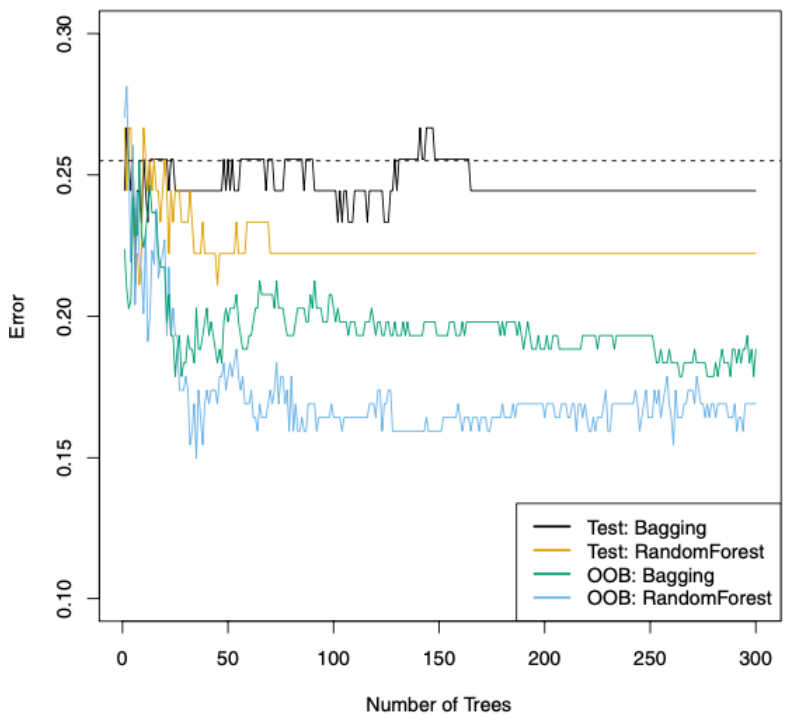
Therefore while bagging will have a reduced variance in the predictions compared with a single tree, random forest algorithm introduces extra randomness (forcing each split to consider only a subset of the features).

Bagging and Random Forests

Therefore, the main difference between bagging and random forests is the choice of feature subset size:

Bagging: $\ell = n$. That is, a special case for the random forest.

Random Forests: $\ell \approx \sqrt{n}$ (typical parametrisation).



The test and oob errors as a function of B, the number of bootstrapped training sets used.

Random Forests

A *Random Forest*[3] (RF) is **an ensemble of Decision Trees**, generally trained via the bagging method (or sometimes pasting).

We can build a *Random Forest classifier* by building a `DecisionTreeClassifier` and pass it to a `BaggingClassifier`. However, it is more convenient and optimized for Decision Trees by using the `RandomForestClassifier` class available in Scikit-Learn. Similarly, there is a `RandomForestRegressor` class for regression.

```
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)
```

[3] "Random Decision Forest," T. Ho (1995)

Random Forests (cont.)

A `RandomForestClassifier` has almost all the hyperparameters of a `DecisionTreeClassifier` plus all the hyperparameters of a `BaggingClassifier`.

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

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

Extra Trees

It is possible to make a RF tree even more random by using random thresholds for each feature when splitting the node (rather than searching for the best possible thresholds like Decision Trees do).

A forest with such an extremely random trees is called an *Extremely Randomized Trees*[4] (or *Extra-Trees* for short). Again, this trades more bias for a lower variance. It also makes Extra-Trees much faster to train than regular RFs.

Scikit-Learn has the `ExtraTreesClassifier` class to support the creation of Extra-Trees classifiers. Similarly, the `ExtraTreesRegressor` class is for regression tasks.

[4] "Extremely Randomized Trees." P. Geurts, D. Ernst, L. Wehenkel (2005)

Feature Importance

While bagging typically improves accuracy over prediction using a single tree, it hinders interpretation (one tree can be visually represented, but it is harder to represent a collection of trees).

Feature importance can be used to provide overall summary about the contribution of each feature to the trees.

For a given feature, relative feature importance is the average total decrease in MSE (regression trees) or average total decrease in the Gini index (classification trees) due to splits using that feature.

Therefore, a large value indicates an important feature.

Feature Importance

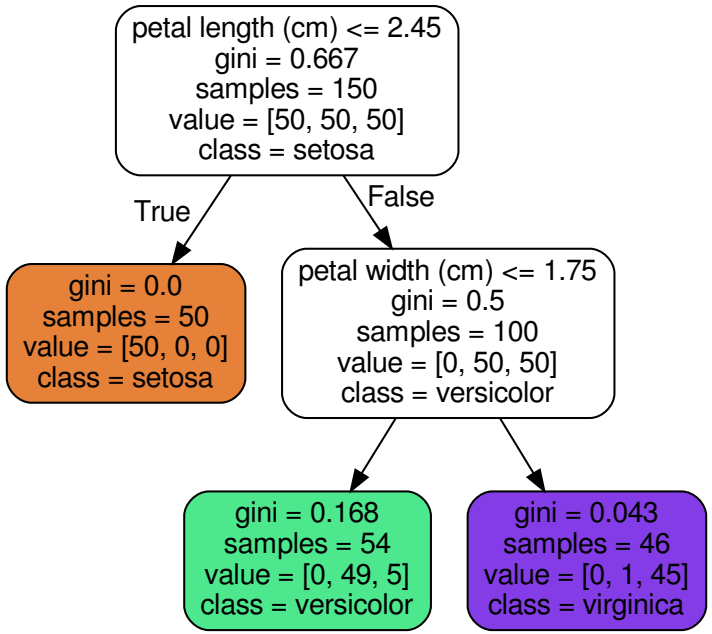
Scikit-Learn measures a feature's (say x_i) importance as follows:

- collects all the nodes (in all the trees) that use x_i for node splitting;
- computes the impurity reduction at each of these nodes, weighted by the number of training instances associated with the node;
- computes the average impurity reduction over all these nodes.

Scikit-Learn computes this score automatically for each feature x_i after training, then it scales the results so that the sum of all importances is equal to 1.

We can access this result via the `feature_importances_` variable.

Feature Importance (cont.)



A decision tree of `max_depth=2` on the Iris dataset.

```
>> 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)
```

Feature Importance (cont.)

Similarly, we can train a RF classifier on the MNIST dataset and visualize the importance of each pixel:

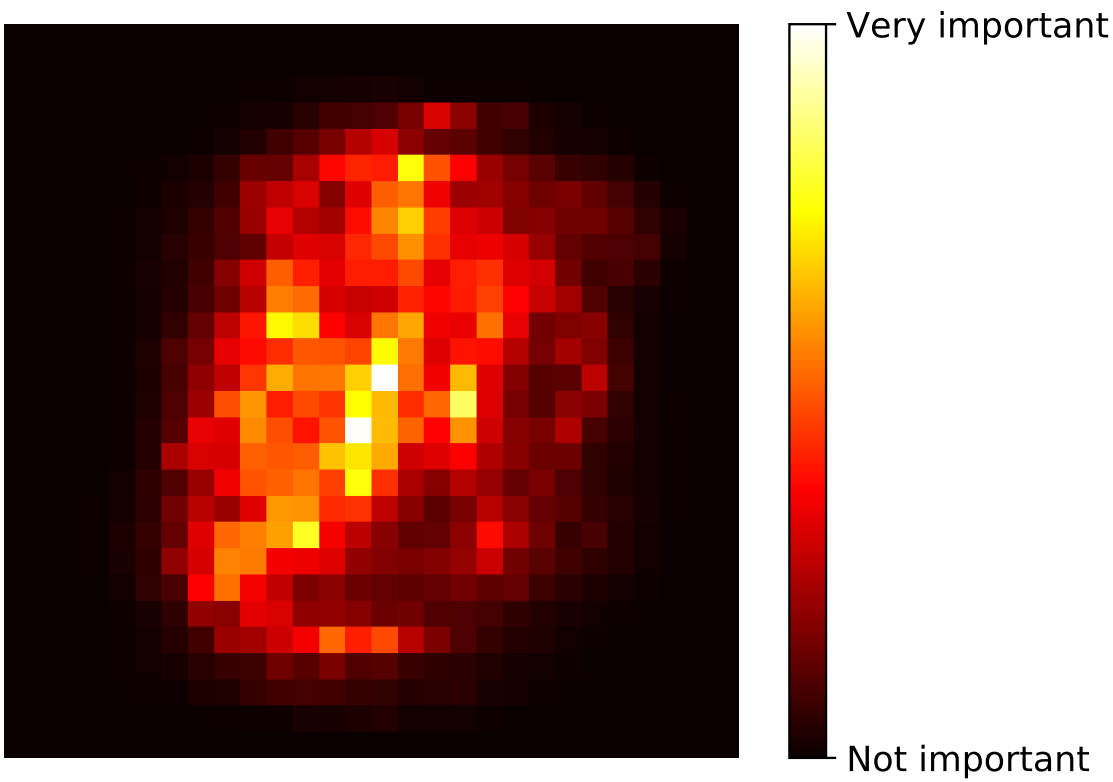


Figure 7-6. MNIST pixel importance (according to a Random Forest classifier)

Boosting

Boosting is also a general approach referring to any ensemble method that can combine several weak learners into a strong learner.

The general idea of most boosting methods is to train the predictors **sequentially**, each trying to correct the predecessor. The aim is to slowly improve the model is areas where it does not fit well.

Many boosting methods are available. By far the most popular are *Adaptive Boosting*[5] (*AdaBoost* for short) and *Gradient Boosting*.

[5] "A Decision-Theoretic Generalization of On-Line Learning and an Application in Boosting," Yoav Freund, Robert E. Schapire (1997)

AdaBoost

A first base classifier is trained and used to make predictions on the training set. The relative weights of the misclassified training instances are then increased. A second classifier is trained using the updated weights and then makes predictions and so on.

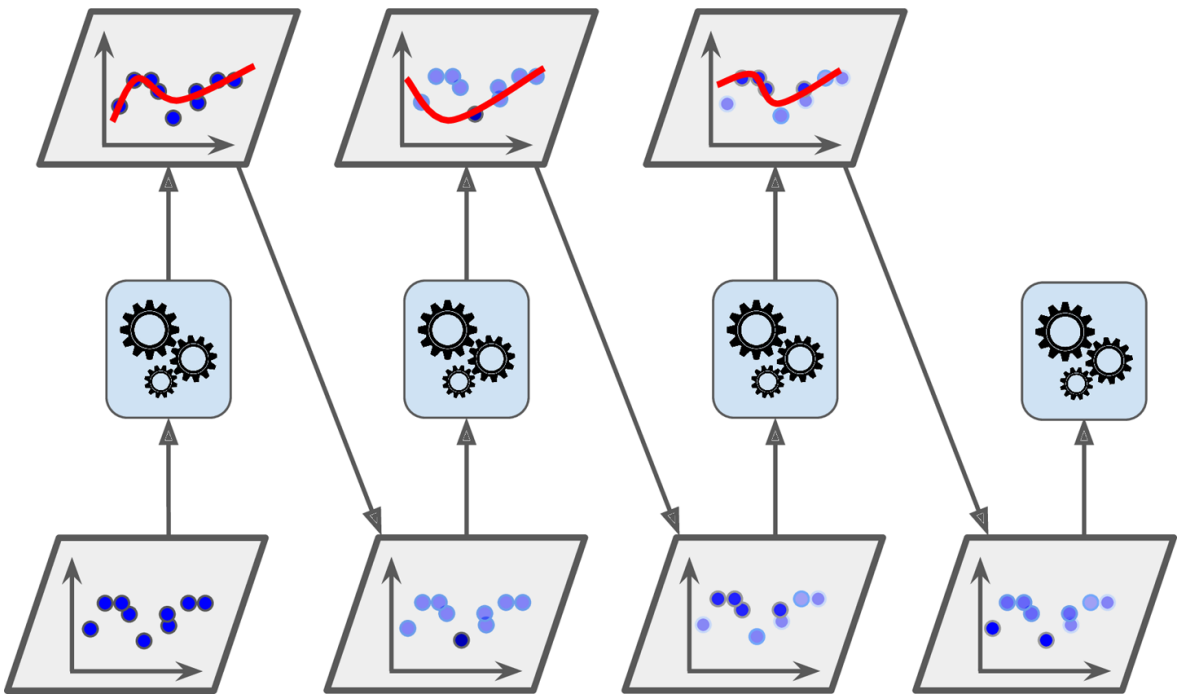


Figure 7-7. AdaBoost sequential training with instance weight updates

AdaBoost (cont.)

The example below shows the decision boundary of consecutive predictors on the *moons* dataset. Here, each predictor is a highly regularized SVM classifier with an RBF (stands for *radial basis function*) kernel.

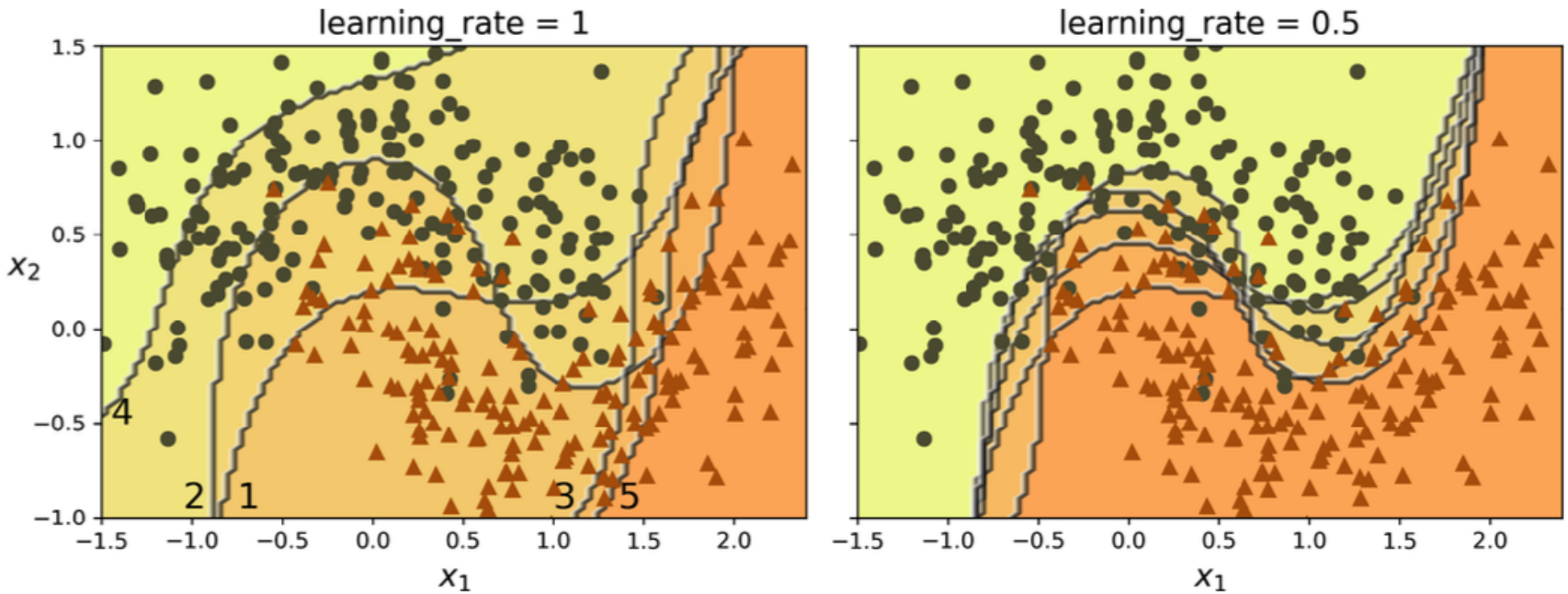


Figure 7-8. Decision boundaries of consecutive predictors

AdaBoost Algorithm

When a first predictor is trained, its weighted error rate r_1 is computed on the training set:

$$r_j = \sum_{\substack{i=1 \\ \hat{y}_j^{(i)} \neq y^{(i)}}}^{m} w^{(i)}$$

where $\hat{y}_j^{(i)}$ is the jth predictor's prediction for the ith instance. Each instance weight $w^{(i)}$ is initially set to $1/m$. Then, the predictor's weight α_j is computed:

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

where η is the learning rate hyperparameter.

The more accurate the predictor is, the height its weight will be. AdaBoost algorithm updates the instance weights (boosting the weights of the misclassified instances):

for i = 1, 2, ..., m $\qquad$ $w^{(i)} = \begin{cases} w^{(i)} & \text{if } \hat{y}_j^{(i)} = y^{(i)} \\ w^{(i)} \exp(\alpha_j) & \text{if } \hat{y}_j^{(i)} \neq y^{(i)} \end{cases}$

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

AdaBoost Algorithm (Cont.)

AdaBoost computes the predictions of all the predictors and weights them using the predictor weights α_j.

The predicted class is the one that receives the majority of weighted votes.

$$\hat{y}(\mathbf{x}) = \arg\max_{k} \sum_{\substack{i=1 \\ \hat{y}_j(\mathbf{x})=k}}^{N} \alpha_j$$

where N is the number of predictors.

Gradient Boosting

Just like AdaBoost, *Gradient Boosting*[6] works by **sequentially** adding predictors to an ensemble, each one correcting its predecessor.

Instead of tweaking the instance weights at every iteration, it tries to fit the new predictor to the residual errors made by the previous predictor.

Regression example: using Decision Trees as the base predictors with gradient boosting, we get what is called *Gradient Boosted Regression Trees* (*GBRT*)).

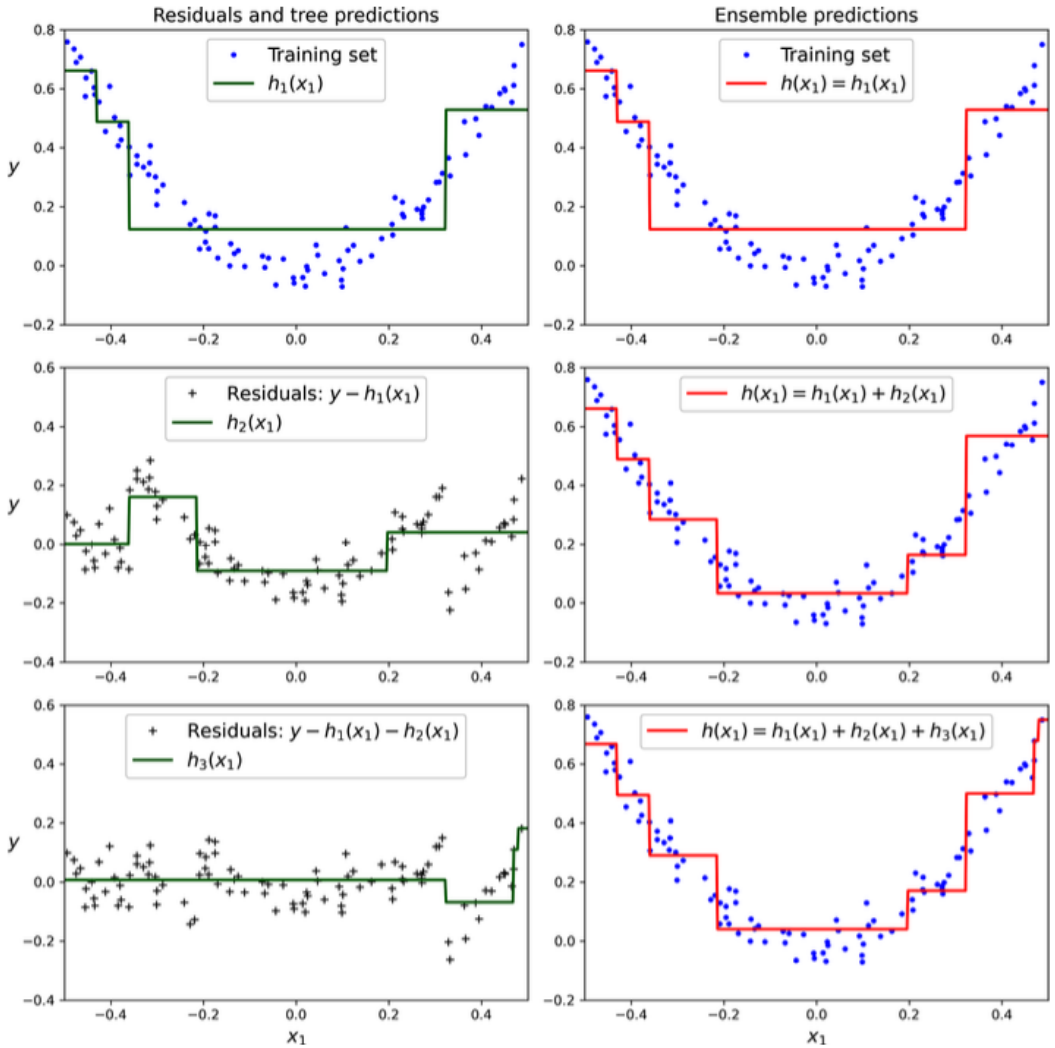
[6] First introduced in "Arcing the Edge," L. Breiman (1997).

Gradient Boosted Regression Trees - Example

```
from sklearn.tree import DecisionTreeRegressor

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

# define and train the first regressor
tree_reg1 = DecisionTreeRegressor(max_depth=2)
tree_reg1.fit(X, y)

# train the second regressor 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)

# repeat this for the third regressor
y3 = y2 - tree_reg2.predict(X)
tree_reg3 = DecisionTreeRegressor(max_depth=2)
tree_reg3.fit(X, y3)

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

Gradient Boosting (cont.)

Left column: Predictions of the three trees from the code on the previous slide
Right column: the ensemble's predictions.



Gradient Boosting (cont.)

Scikit-Learn's `GradientBoostingRegressor` class provides a way to to train GBRT ensembles, which has hyperparameters to control the growth the Decision Trees (e.g., `max_depth`, `min_samples_leaf`, and so on), and hyperparameters to control the ensemble training (e.g., `n_estimators`).

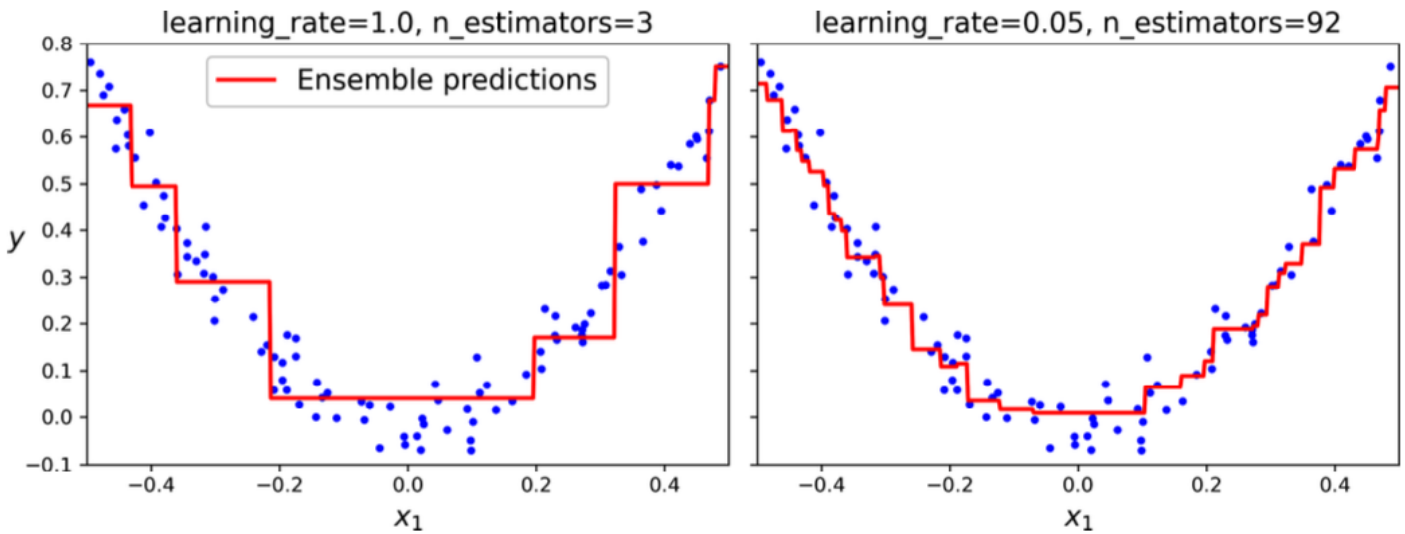
Example:
Train a GBRT on the training data `X` and training values `y`:

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

Gradient Boosting (cont.)

The hyperparameter learning rate can be used to scale the contribution of each tree. For a low learning rate (e.g., 0.05), more trees in the ensemble would be needed to fit the training set but the predictions will usually generalize better. This is a regularization technique known as *shrinkage*.

Example: Two GBRT ensembles trained with a low learning rate. Left column: underfitting due to insufficient #trees; Right column: better generalisation.



Gradient Boosting (cont.)

To find the optimal number of trees to avoid underfitting and overfitting, we can use early stopping by setting the `n_iter_no_change` hyperparameter to an integer value in the `GradientBoostingRegressor` class.

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

>>> gbrt_best.n_estimators_
92
```

`n_iter_no_change` too low: training may stop too early (prone to underfit).
`n_iter_no_change` too high: prone to overfit.

Typically we use a small learning rate and a high number of estimators (but with actual number of estimators in the trained ensemble being much slower).

Stacking

Stacking is short for *stacked generalization*[7]. 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, we train a model to perform this aggregation.

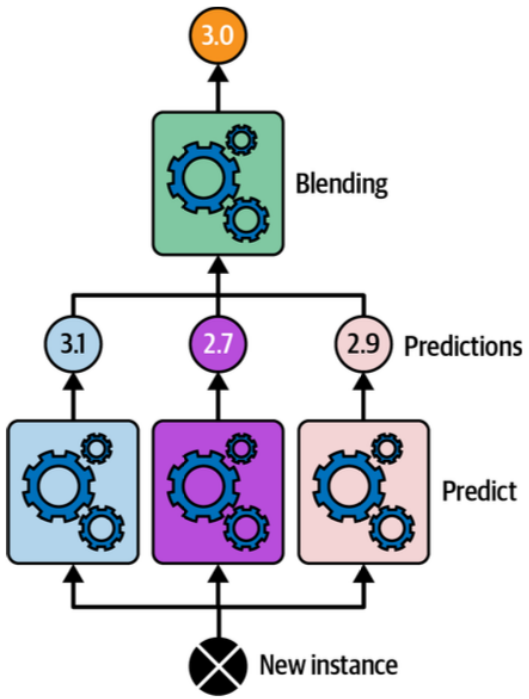
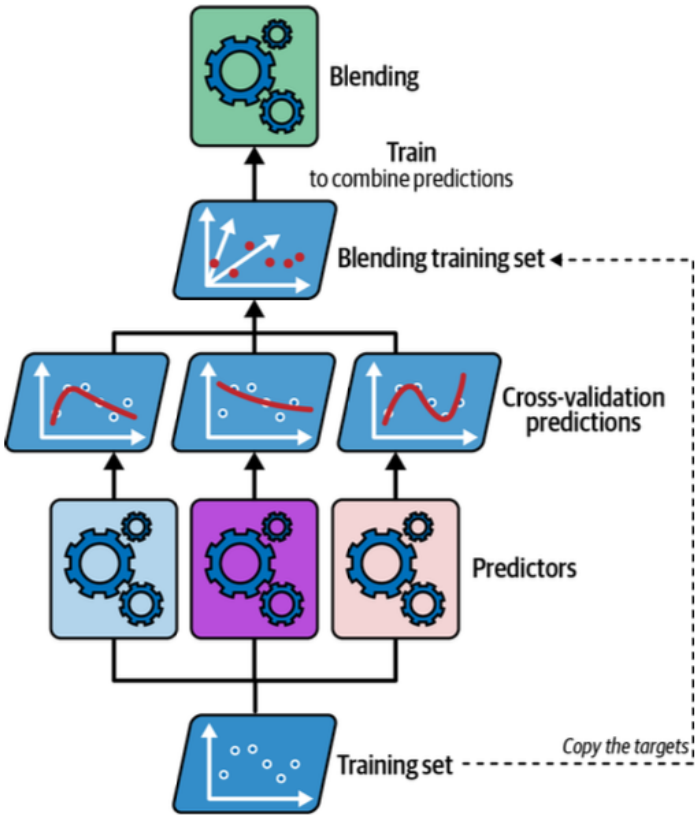


Figure 7-12. Aggregating predictions using a blending predictor

[7] "Stacked Generalization," D. Wolpert (1992).

Stacking (cont.)

To train the blender, a common approach is to build a training set from out-of-sample predictions on every predictor for each instance in the original training set (using cross-validation).



The main steps for implementing *Stacking* are:

1. Train the base predictors using cross-validation.
2. Use each validation set to test the predictions produced by the predictors (that is, generate cross-validated estimates for each input data point).

Stacking (cont.)

The main steps for implementing *Stacking* are:

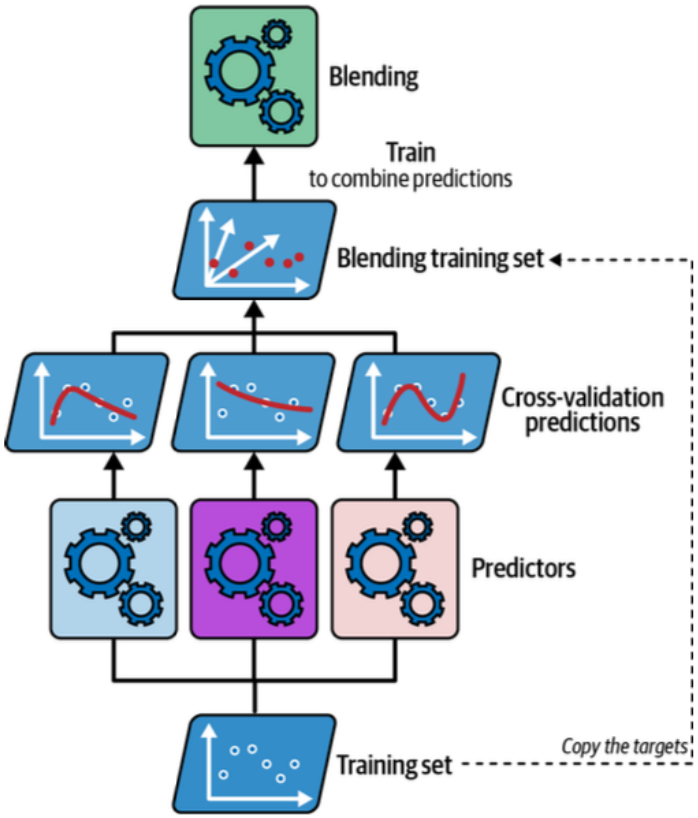


Figure 7-14. Training the blender

3. Create a new 3-dimensional (since we have 3 predictors in this example) training set using these predicted values as input features and keep the target values.
4. Train the *blender* on this new training set to learn to predict the target values.

Once the blender is trained, the base predictors are retrained one last time on the full original training set.

Stacking (cont.)

See `sklearn.ensemble.StackingClassifier` and `sklearn.ensemble.StackingRegressor`

Some parameters from the `StackingClassifier` class:

- `estimators` – list of base estimators (strings or objects)
- `final_estimator` that will be used to combine the base estimators.

```
from sklearn.ensemble import StackingClassifier

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

Summary

Bagging: The trees are grown independently on random samples of the training set and tend to be quite similar to each other.

Random Forests: The trees are grown independently on random samples of the training set. However, each split on each tree is performed using a random subset of the features, thereby decorrelating the trees.

Boosting: Only use the original data, and do not draw any random samples. The trees are grown successively: each new tree is fit to the signal that is left over from the earlier trees, and shrunken down before it is used.

For the next lecture

Work through the Assignment and attend the supervised lab. The Unit Coordinator or a casual Teaching Assistant will be there to help.

Next Tuesday is a holiday - no consultation time and lecture, but lab sessions will run normally on Thursday.

Read up to Chapter 8 on Dimensionality Reduction.

And that's all for the lecture.

Have a good week.