CITS5508 Machine Learning
Lectures for Semester 1, 2021
Lecture Weeks 5&6: Book Chapter 7

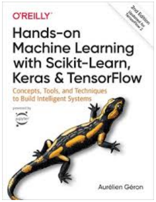
Dr Du Huynh (Unit Coordinator and Lecturer)
UWA

2021

Today

Chapter 7.

Hands-on Machine Learning with Scikit-Learn & TensorFlow


O'REILLY
2nd Edition Updated for TensorFlow 2
Hands-on Machine Learning with Scikit-Learn, Keras & TensorFlow
Concepts, Tools, and Techniques to Build Intelligent Systems
Aurélien Géron

Chapter Seven

Ensemble Learning and Random Forests

In this chapter, 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 Patches and Random Subspaces
- Random Forests
- Boosting
- Stacking

Ensemble Learning

Suppose that you are seeking the answer to a complex question. You could ask the question to thousands of random people and then aggregate their answers. In many cases you will find that this aggregated answer is better than the answer from a single expert.

This is called the *wisdom of the crowd*.

Ensemble learning adopts this similar concept.

Ensemble Learning

Given that you have implemented a group of predictors (such as classifiers or regressors), if you aggregate the predictions of all the predictors together, you will often get better predictions than with the best individual predictor.

- A group of predictors is called an *ensemble*.
- This technique is therefore called *Ensemble Learning*.
- An Ensemble Learning algorithm is called an *Ensemble method*.

  e.g., a *Random Forest* is an ensemble of Decision Trees.

Voting Classifiers

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

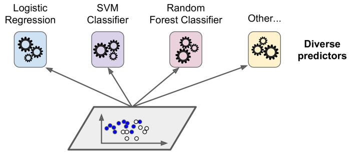


Figure 7-1. Training diverse classifiers

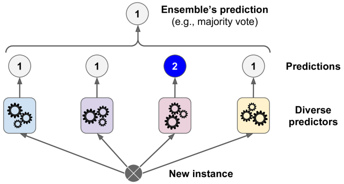


Figure 7-2. Hard voting classifier predictions

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 perfectly independent and making uncorrelated errors.

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

A Voting Classifier Example

Instead of a *hard voting classifier*, 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). We can tell Scikit-Learn to predict the class with the highest class probability, averaged over all the classifiers.

- **Training a voting classifier:**

```
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(probability=True)

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

A Voting Classifier Example (cont.)

- **Testing individual classifier's accuracy versus the voting classifier's accuracy:**

```
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))
```

- **Output:**

```
LogisticRegression 0.864
RandomForestClassifier 0.872
SVC 0.888
VotingClassifier 0.912
```

Hard Voting vs Soft Voting

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

(See lecture recording)

Bagging and Pasting

We mentioned earlier that to get a diverse set of classifiers we should use very different training algorithms. Another approach is to use the same training algorithm for every classifier but to train each classifier on different random subsets of the training set. This means that we need to draw random samples from the training set.

There are two sampling methods:

1. **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 terms "predictor" and "classifier" interchangeably)

[1] "Bagging Predictors," L. Breiman (1996)

Bagging and Pasting

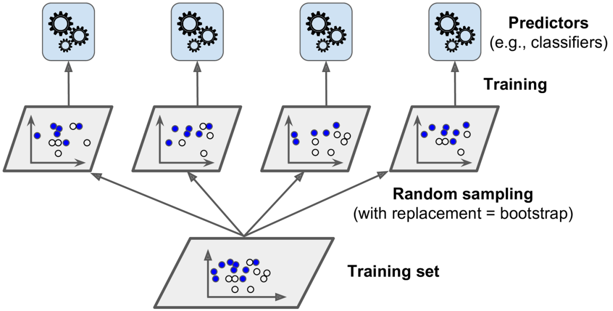


Figure 7-4. Bagging and pasting involves training several predictors on different random samples of the training set

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.

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**.

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 are popular methods because they scale very well:

- the predictors can be trained in parallel, using different CPU cores or even on different servers.
- the predictors can also be tested in parallel.

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, bootstrap=True, n_jobs=-1)
bag_clf.fit(X_train, y_train)
y_pred = bag_clf.predict(X_test)
```

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

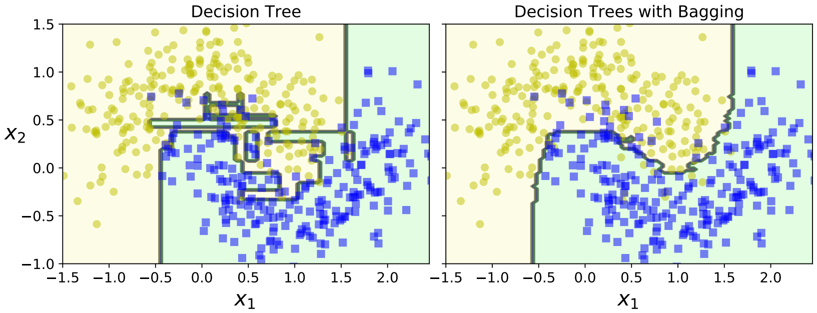


Figure 7-5. A single Decision Tree (left) versus a bagging ensemble of 500 trees (right)

Comparison of the decision boundary. Left: A single decision tree. Right: A bagging ensemble of 500 trees. Both were trained on the `moons` dataset.

Comparing Bagging and Pasting

- Bootstrapping introduces a bit more diversity in the subsets that each predictor is trained on, so *bagging* ends up with a slightly higher bias than *pasting*.
- However, the point above also means that the predictors of *bagging* are less correlated, so the ensemble has lower variance if *bagging* is used.
- Overall, *bagging* often results in better models, so it is generally more preferred than *pasting*.

One good exercise is to use cross validation to evaluate both *bagging* and *pasting* on your own problem data and select the one that works best.

Out-of-Bag Evaluation

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

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

Out-of-Bag Evaluation in Scikit-Learn

To request 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,
... bootstrap=True, n_jobs=-1, oob_score=True)
...
>>> bag_clf.fit(X_train, y_train)
```

We can inspect the oob score via

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

and the oob decision function via

```
>>> bag_clf.oob_decision_function_
array([[ 0.31746032, 0.68253968],
[ 0.34117647, 0.65882353],
...
[ 0.57291667, 0.42708333]])
```

Random Patches and Random Subspaces

The `BaggingClassifier` class has two hyperparameters:

- `max_samples`, for controlling the maximum number of samples for each classifier in the ensemble; and
- `bootstrap`, for specifying whether to use bagging or pasting.

The `BaggingClassifier` class also supports sampling the features. This is controlled by two hyperparameters `max_features` and `bootstrap_features`. This is useful when you are dealing with high-dimensional inputs (such as images).

Random Patches and Random Subspaces (cont.)

Random Patches method and *Random Subspaces* method are ways of sampling the features.

- **Sampling both training instances and features** is called the **Random Patches** method.
- **Keeping all training instances** (`bootstrap=False` and `max_samples=1.0`) but **sampling features** (bootstrap_features=True and/or `max_features < 1.0`) is called the **Random Subspaces** method.

Random Patches and Random Subspaces (cont.)

Random Patches method – each predictor samples both instances and features

feat. 1	feat. 2	$\cdots$	feat. i	$\cdots$	feat. j	$\cdots$	feat. n
xxx	xxx	xxx	xxx	xxx	xxx	xxx	xxx
xxx	**xxx**	xxx	**xxx**	xxx	**xxx**	xxx	xxx
xxx	**xxx**	xxx	**xxx**	xxx	**xxx**	xxx	xxx
...	...	...	...	...	...	...	...
xxx	xxx	xxx	xxx	xxx	xxx	xxx	xxx
...	...	...	...	...	...	...	...
xxx	**xxx**	xxx	**xxx**	xxx	**xxx**	xxx	xxx
xxx	xxx	xxx	xxx	xxx	xxx	xxx	xxx
...	...	...	...	...	...	...	...
xxx	xxx	xxx	xxx	xxx	xxx	xxx	xxx
xxx	**xxx**	xxx	**xxx**	xxx	**xxx**	xxx	xxx
xxx	**xxx**	xxx	**xxx**	xxx	**xxx**	xxx	xxx
xxx	**xxx**	xxx	**xxx**	xxx	**xxx**	xxx	xxx

Hyperparameter settings: `bootstrap = True`, `max_samples < 1.0`, `bootstrap_features = True`, and `max_features < 1.0`.

Random Patches and Random Subspaces (cont.)

Random Subspaces method – each predictor samples only features

feat. 1	feat. 2	$\cdots$	feat. i	$\cdots$	feat. j	$\cdots$	feat. n
xxx	**xxx**	xxx	**xxx**	xxx	**xxx**	xxx	xxx
xxx	**xxx**	xxx	**xxx**	xxx	**xxx**	xxx	xxx
xxx	**xxx**	xxx	**xxx**	xxx	**xxx**	xxx	xxx
...	...	...	...	...	...	...	...
xxx	**xxx**	xxx	**xxx**	xxx	**xxx**	xxx	xxx
...	...	...	...	...	...	...	...
xxx	**xxx**	xxx	**xxx**	xxx	**xxx**	xxx	xxx
xxx	**xxx**	xxx	**xxx**	xxx	**xxx**	xxx	xxx
...	...	...	...	...	...	...	...
xxx	**xxx**	xxx	**xxx**	xxx	**xxx**	xxx	xxx
xxx	**xxx**	xxx	**xxx**	xxx	**xxx**	xxx	xxx
xxx	**xxx**	xxx	**xxx**	xxx	**xxx**	xxx	xxx
xxx	**xxx**	xxx	**xxx**	xxx	**xxx**	xxx	xxx

Hyperparameter settings: `bootstrap = False`, `max_samples = 1.0`, `bootstrap_features = True`, and `max_features < 1.0`.

Random Patches and Random Subspaces (cont.)

Sampling features results in even more predictor diversity, trading a bit more bias for a lower variance.

(See lecture recording)

Which one to use? Random Patches method? or Random Subspaces method?

Rule of thumb: large dataset having many (sufficient) instances, then try Random Patches; otherwise, try Random Subspaces.

Random Forests

A *Random Forest*[2] (RF) is **an ensemble of Decision Trees**, generally trained via the bagging method (or sometimes pasting), typically with `max_samples=1.0` (i.e., the entire training set).

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.

Example:

```
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)
```

[2] "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 RF algorithm introduces extra randomness when growing trees: instead of searching for the best feature when splitting a node (see previous topic), it searches for the best feature among the random subset of features.
$\Rightarrow$ greater tree diversity (which trades a higher bias for a lower variance)
$\Rightarrow$ generally yielding an overall better model.

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)
```

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 other Decision Trees do).

A forest with such an extremely random trees is called an *Extremely Randomized Trees*[3] (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.

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

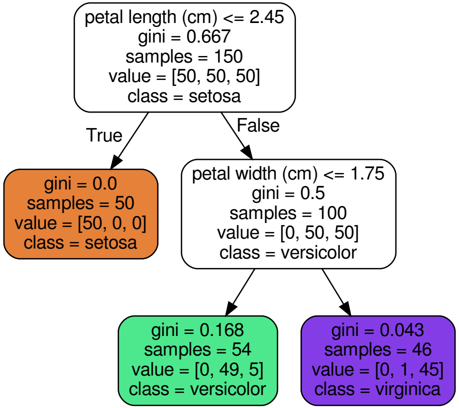
Feature Importance

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.

Feature importances:
petal length 0.562
petal width 0.438

Feature Importance (cont.)

Example:

```
>>> 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
```

The result shows that the most important features are the petal length (44%) and petal width (42%) while the sepal length and width are rather unimportant.

*** RFs are very handy to get a quick understanding of what features actually matter, particularly if you need to perform feature selection and/or feature pruning.

Feature Importance (cont.)

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

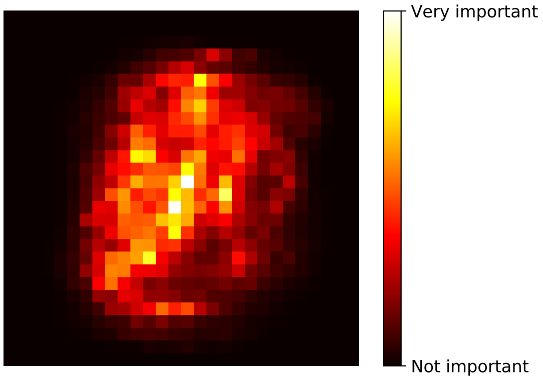


Figure 7-6. MNIST pixel importance (according to a Random Forest classifier)

Boosting

Boosting (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 the predictors **sequentially**, each trying to correct the predecessor.

Many boosting methods are available. By far the most popular are *Adaptive Boosting*[4] (*AdaBoost* for short) and *Gradient Boosting*.

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

AdaBoost

For example, 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.

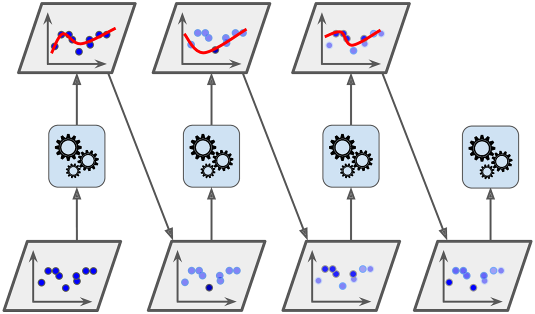


Figure 7-7. AdaBoost sequential training with instance weight updates

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.

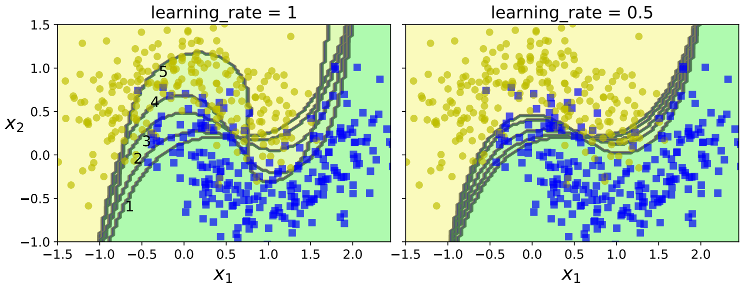


Figure 7-8. Decision boundaries of consecutive predictors

Exercise: Look at the sample code of the author and replace it by using the `sklearn.ensemble.AdaBoostClassifier` class.

AdaBoost (cont.)

- Scikit-Learn uses a multiclass version of AdaBoost called *SAMME*[5] (stands for *Stagewise Additive Modeling using a Multiclass Exponential loss function*).
- When there are just two classes, SAMME is equivalent to AdaBoost.
- A variant of SAMME is *SAMME.R* (the *R* stands for "Real"), which relies on class probabilities rather than predictions and generally performs better.

[5] "Multi-Class AdaBoost," J. Zhu et al (2006).

AdaBoost (cont.)

Example: Train an AdaBoost classifier based on 200 *Decision Stumps* using Scikit-Learn's `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)
```

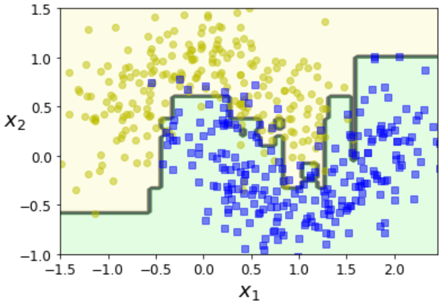
- A *Decision Stump* is a Decision Tree with `max_depth=1` – a tree composed of a single decision node plus two leaf nodes. This is the default base estimator for the AdaBoostClassifier class.
- There is also an `AdaBoostRegressor` class for regression problems.

Tip: If your AdaBoost ensemble is overfitting the training set, you can try: (i) reducing the number of estimators or (ii) more strongly regularizing the base estimator.

AdaBoost (cont.)

```
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)

plot_decision_boundary(ada_clf, X, y)  # see the author’s code
```



Gradient Boosting

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

Unlike the AdaBoost classifiers, which can take any base estimator, e.g., SVC, SGDClassifier, etc, Gradient Boosting classifiers can only have `DecisionTreeClassifier` as the base estimator. (similarly for Gradient Boosting regressors)

Using Decision Trees as the base predictors with gradient boosting, we get what is called *Gradient Tree Boosting* (or *Gradient Boosted Regression Trees* (*GBRT*) for regression).

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

Gradient Boosting (cont.)

Example:

```
from sklearn.tree import DecisionTreeRegressor

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

# define and 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)

# testing for a prediction
y_pred = sum(tree.predict(X_new) for tree in (tree_reg1,tree_reg2,tree_reg3))
```

Gradient Boosting (cont.)

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

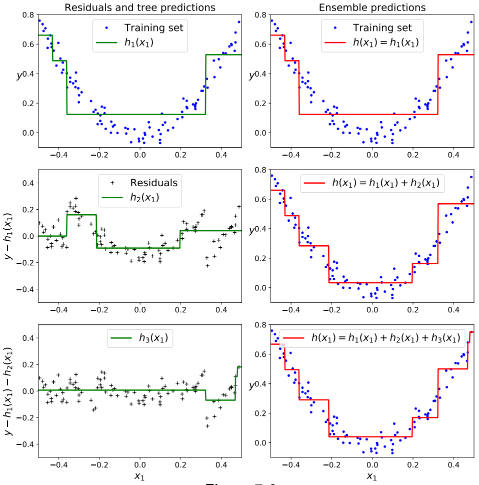


Figure 7-9

Gradient Boosting (cont.)

- Rather than building the base regressor one by one, a simpler way to train GBRT ensembles is to use Scikit-Learn's `GradientBoostingRegressor` class, 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.)

- There are other hyperparameters as well. For example, the hyperparameter learning rate can be used to scale the contribution of each tree. For a low learning rate (e.g., 0.1), 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: overfitting due to too many trees.

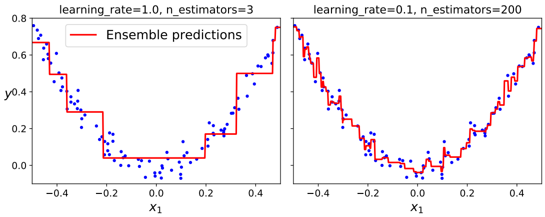


Figure 7-10.

Gradient Boosting (cont.)

- It seems that it is important to find the optimal number of trees to avoid underfitting and overfitting. To do so, we can use early stopping by calling the method `staged_predict()` in the `GradientBoostingRegressor` class. Example:

```
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)
# train a GBRT ensemble using 120 trees
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)
```

Gradient Boosting (cont.)

The evaluation errors and the best model's prediction for the code on the previous slide:

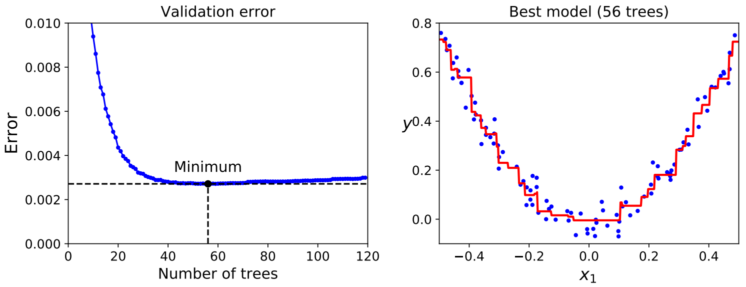


Figure 7-11. Tuning the number of trees using early stopping

Gradient Boosting (cont.)

- It is possible to implement early stopping by actually stopping training early (instead of training a large number of trees first and then looking back to find the optimal number). We can do so by setting `warm_start=True`, which makes Scikit-Learn keep existing trees when the `fit()` method is called, allowing incremental training.
- We can also specify the fraction of training instances for training each tree by setting `subsamples` appropriately, e.g., `subsamples=0.25` means that each tree is trained on 25% of the training instances randomly selected. Again, this trades a higher bias for a lower variance. It also speeds up the training considerably. This technique is called *Stochastic Gradient Boosting*.

Gradient Boosting (cont.)

Example: stop training when the validation error does not improve for 5 iterations in a row:

```
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

This is the last Ensemble method that we will cover in this chapter. *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, why don't we train a model to perform this aggregation?

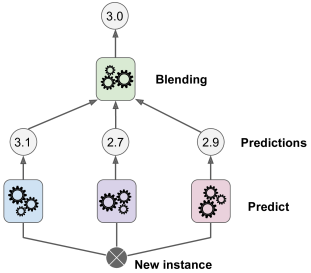


Figure 7-12. Aggregating predictions using a blending predictor

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

Stacking (cont.)

To train the blender, a common approach is to use a **hold-out set**. The main steps for implementing *Stacking* are:

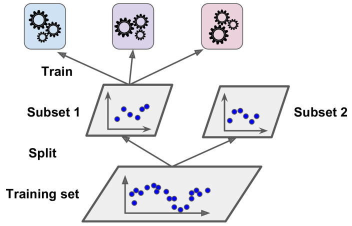


Figure 7-13. Training the first layer

1. Split the training data into 2 subsets: subset 1 and subset 2.
2. Use subset 1 to train the base predictors.

Stacking (cont.)

The main steps for implementing *Stacking* are:

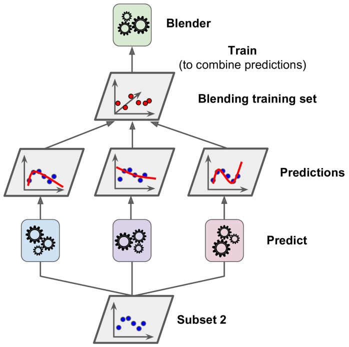


Figure 7-14. Training the blender

3. Use subset 2 to test the predictions produced by the predictors.
4. 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 (see next slide).
5. Train the *blender* on this new training set to learn to predict the target values.

Stacking (cont.)

For the example shown in Figure 7-14, the blending training set created from subset 2 looks like:

predictor 1's output	predictor 2's output	predictor 3's output	ground truth
xxx	xxx	xxx	yyy
xxx	xxx	xxx	yyy
...	...	...	...
xxx	xxx	xxx	yyy

This blending training set is used for training the blender.

The first three columns can be considered as the feature columns (3D features in this case); the last column is the class label (or ground truth value).

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.
- `stack_method` – method to called for each base estimator for the stacking.

Similarly for the `StackingRegressor` class, except for the parameter `stack_method` which is not used.

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

Read up to Chapter 8 on Dimensionality Reduction.

And that's all for the lecture.

Have a good week.