# Ensemble Learning and Random Forests

Bagging

## Ensembling: Overview

Ensembling is the process of combining multiple models together to form a single model that is more accurate than its individual parts.  The concept is that the mistakes of each individually imperfect (and independent) models will cancel out when the models are combined together.

**Regression** model: take the average of the predictions

**Classification**: take the voted most common prediction

## Ensembling: Classification (majority vote)

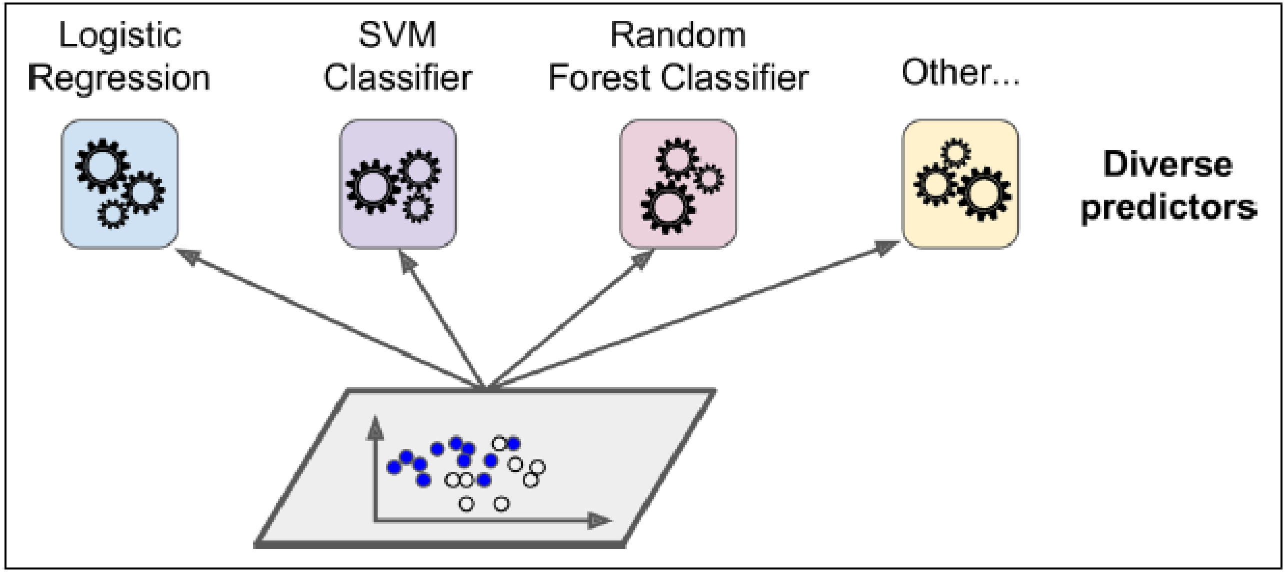


*Figure 7-1. Training diverse classifiers*

# Ensembling: Classification (majority vote)

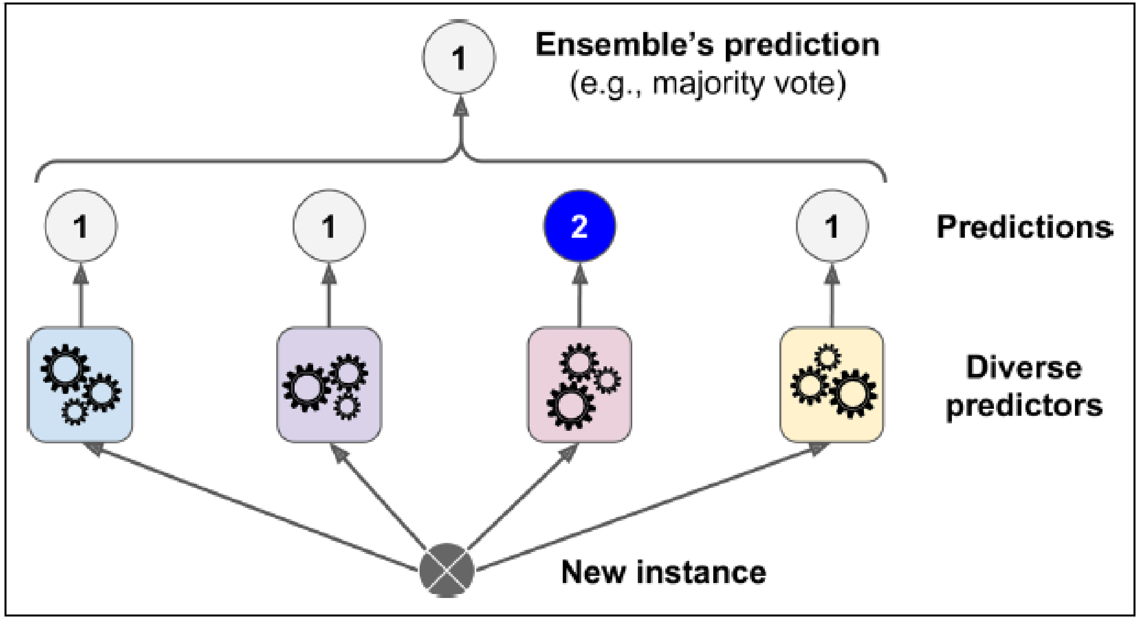
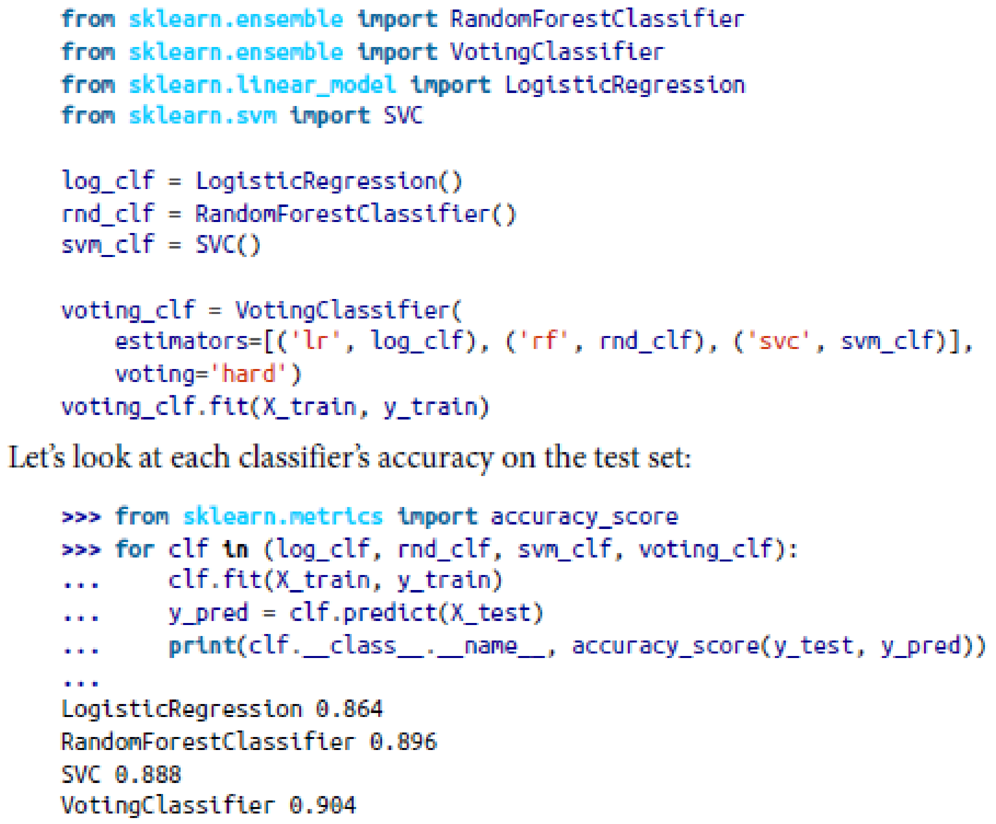


*Figure 7-2. Hard voting classifier predictions*

## Ensembling: Hard Voting



```
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)
```

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
```

## Ensembling: 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*.

replace voting="hard" with voting="soft"

## Ensembling: In Layman’s Terms

One way of ensembling is to have multiple models as discussed just now, an alternative would be sampling.

**Bagging/Pasting** takes a council of experts, gives them each a portion of the understanding of a problem, asks them to make a best educated guess at a solution, and tallies up the total solutions to make a final decision.

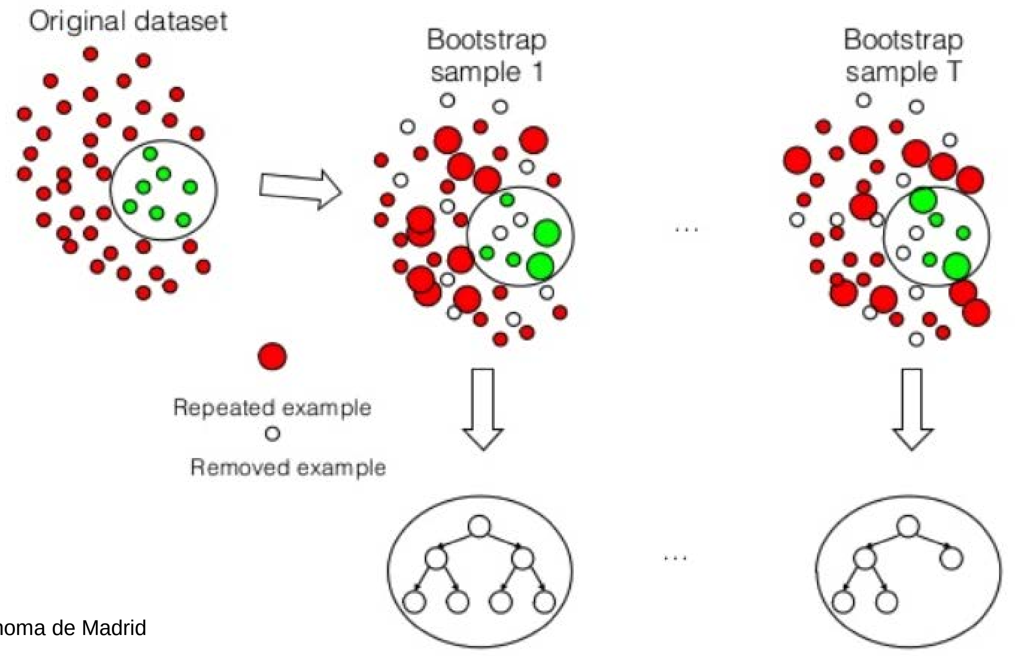
**Boosting** takes a village of idiots, asks them one by one to step up to try to solve the problem. Each idiot picks up where the previous idiot left off, and focuses on fixing the part that the previous idiot messed up on.

## Ensembling: In Technical Terms

**Bagging/Pasting (**1996) also known as the aggregation of bootstrap samples, trains multiples of the  same type of model (decision trees) using bootstrap samples drawn from the training dataset, and combines the prediction results (average for regression, vote for classification). When samples are taking out with replacement, it is called bagging. When samples are taking out without replacement, it is called pasting.

**Boosting** (1999) uses weak learners (typically decision tree classifers) to build a forward stagewise additive model that applies the weak classifiers one by one to repeatedly reweighted versions of the data, so that each new weak learner tries to correct the error of the previous learners.

## Ensemble: Bagging Illustration



# Ensembling: Bagging and Pasting

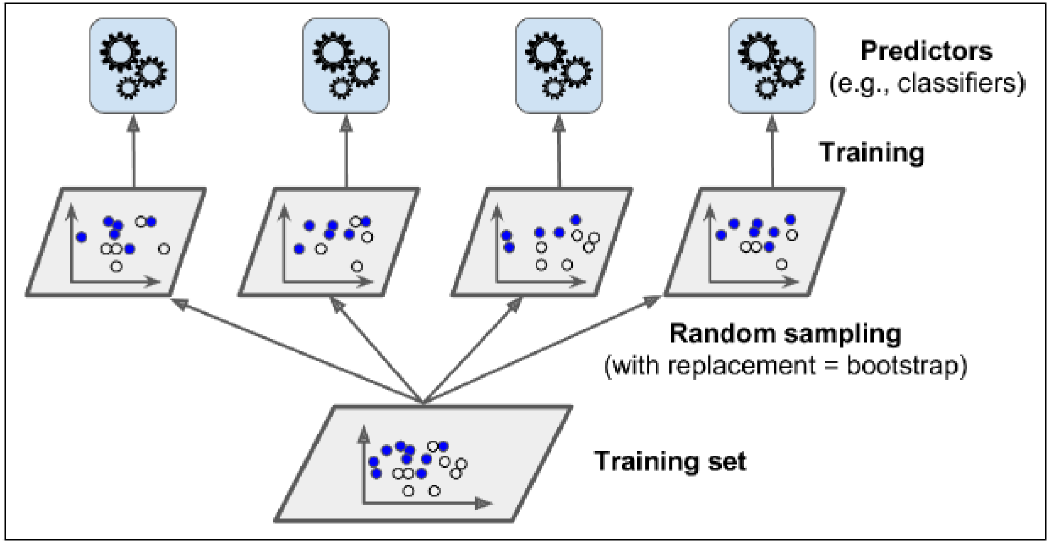


*Figure 7-4. Bagging and pasting involves training several predictors on different random samples of the training set*

```python
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)
```

## Ensembling: Decision Tree with Bagging

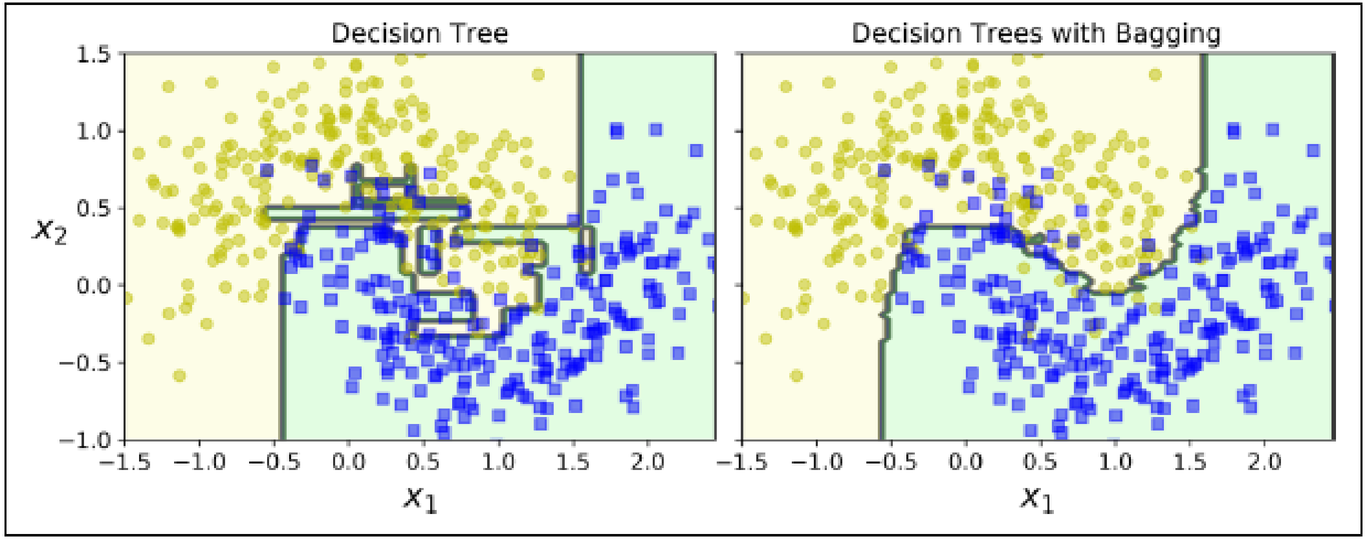
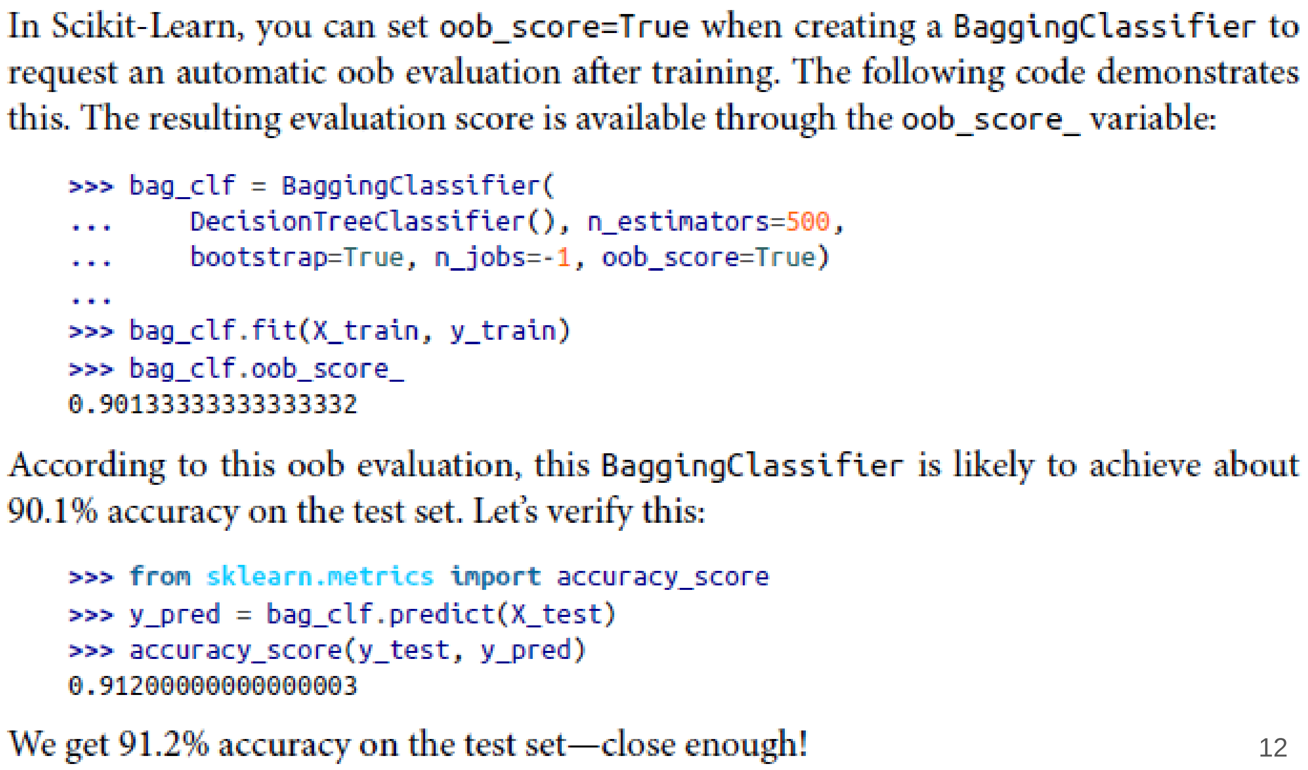


*Figure 7-5. A single Decision Tree (left) versus a bagging ensemble of 500 trees (right)*

## Ensembling: 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.



In Scikit-Learn, you can set oob_score=True when creating a BaggingClassifier 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
```

According to this oob evaluation, this BaggingClassifier is likely to achieve about 90.1% accuracy on the test set. Let's verify this:

```
>>> from sklearn.metrics import accuracy_score
>>> y_pred = bag_clf.predict(X_test)
>>> accuracy_score(y_test, y_pred)
0.91200000000000003
```

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

## Ensembling: Random Patches and Random Subspaces

- BaggingClassifier’s sampling the observations (instances) is controlled by two hyperparamters:
  - *max_sample*
  - *bootstrap*

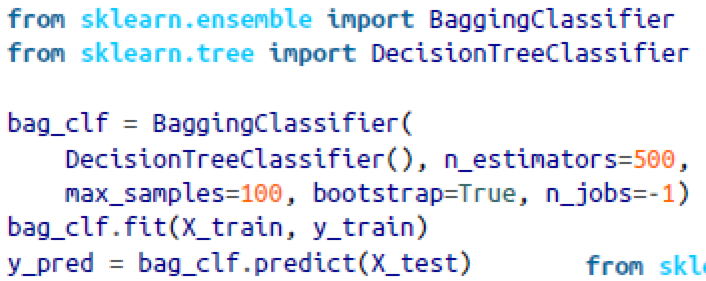
- BaggingClassifier’s sampling the features is controlled by two hyperparamters:
  - *max_features*
  - *bootstrap_features*

- Sampling both training instances and features is called the ***Random Pactches*** 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.

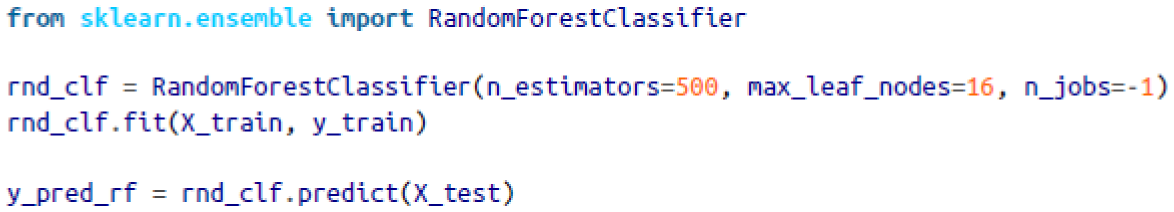
## Random Forests

- 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.



```python
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)
```



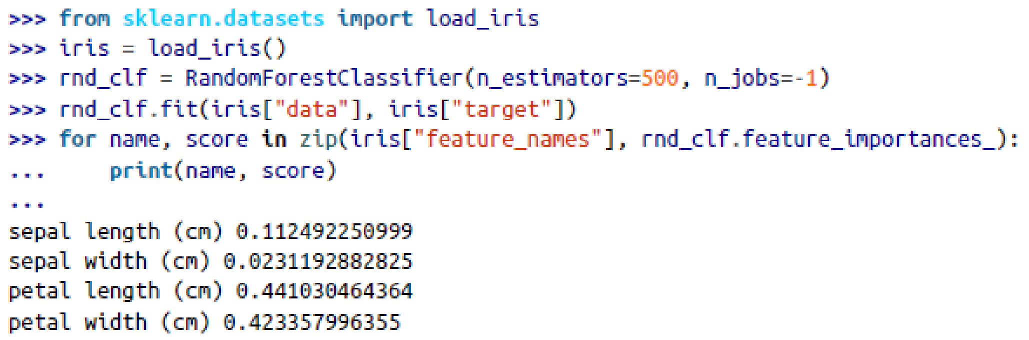
```python
from sklearn.ensemble import RandomForestClassifier

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)
```

## Feature importance

- Output each feature’s 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
```

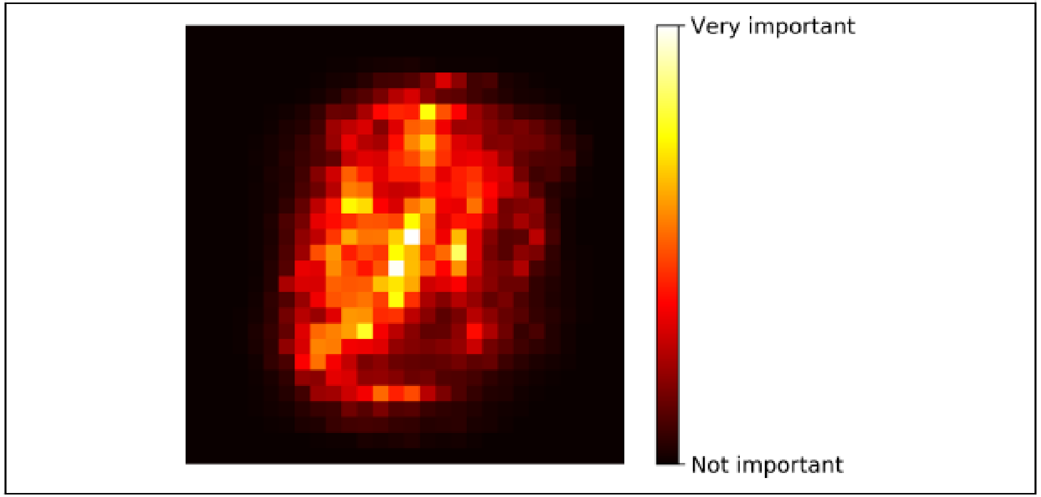


*Figure 7-6. MNIST pixel importance (according to a Random Forest classifier)*

## Ensemble Learning and Random Forests

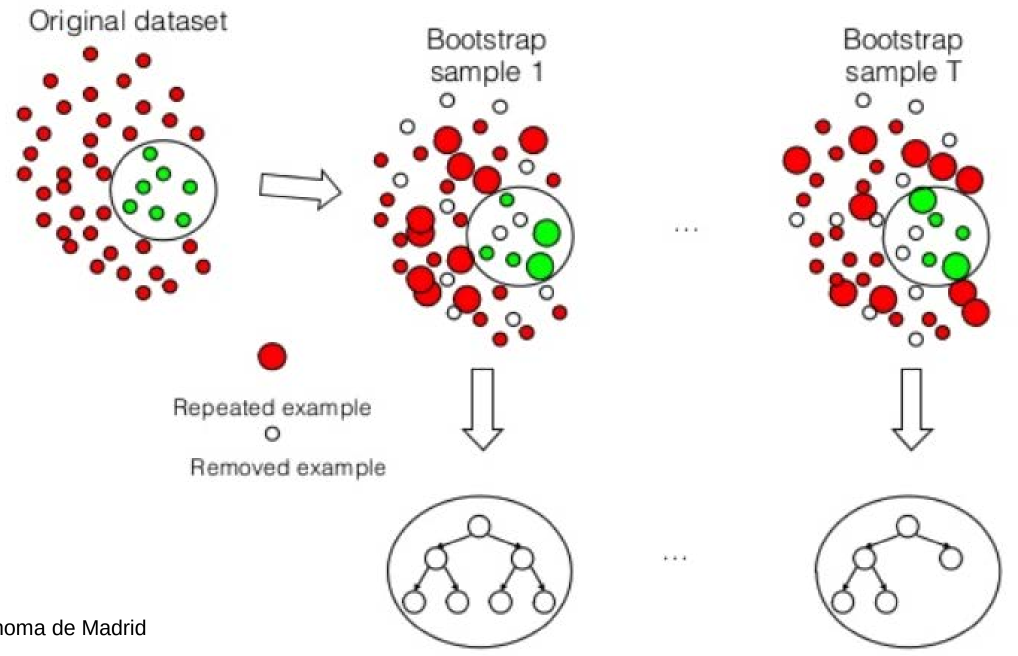
Boosting - AdaBoosting

## Ensembling: In Technical Terms

**Bagging/Pasting (**1996) also known as the aggregation of bootstrap samples, trains multiples of the  same type of model (decision trees) using bootstrap samples drawn from the training dataset, and combines the prediction results (average for regression, vote for classification). When samples are taking out with replacement, it is called bagging. When samples are taking out without replacement, it is called pasting.

**Boosting** (1999) uses weak learners (typically decision tree classifers) to build a forward stagewise additive model that applies the weak classifiers one by one to repeatedly reweighted versions of the data, so that each new weak learner tries to correct the error of the previous learners.

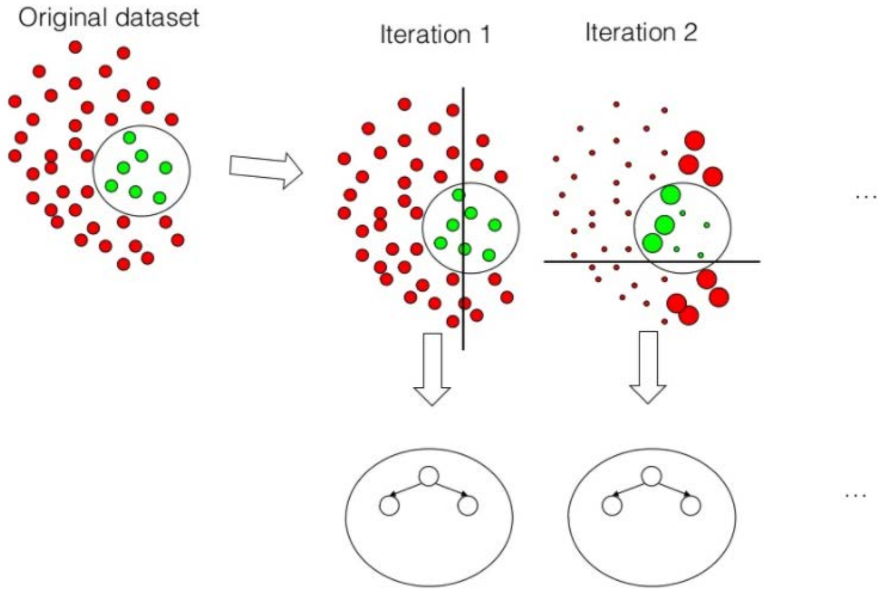
## Ensemble: Bagging Illustration



## Ensembling: Boosting

*Boosting* (originally called *hypothesis boosting*) refers to any Ensemble method that can combine several weak learners into a strong learner.

# Ensembling: Boosting Illustration



## Boosting: AdaBoost

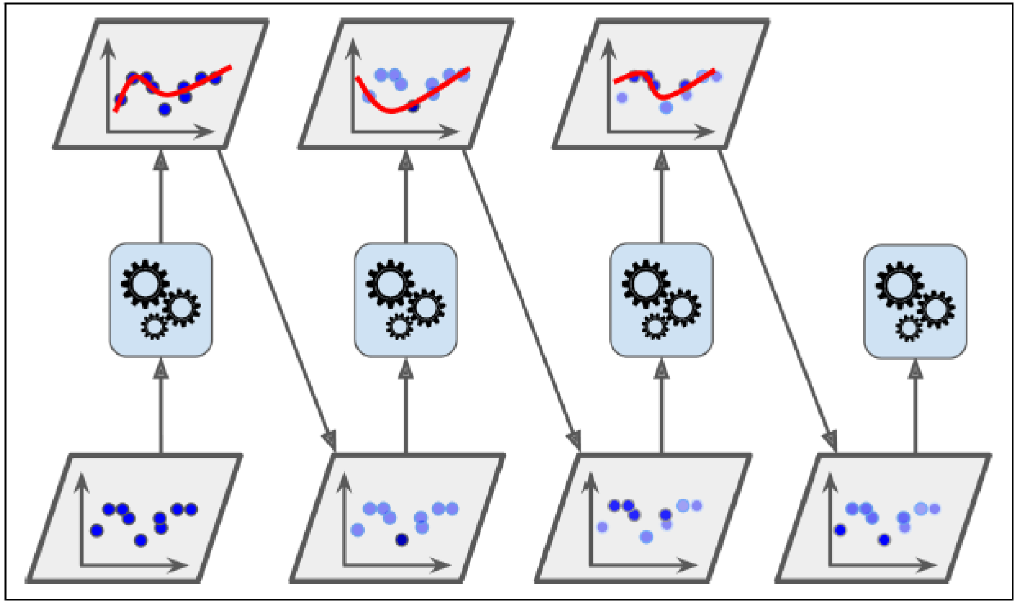


*Figure 7-7. AdaBoost sequential training with instance weight updates*

## Boosting: AdaBoost (moon data)

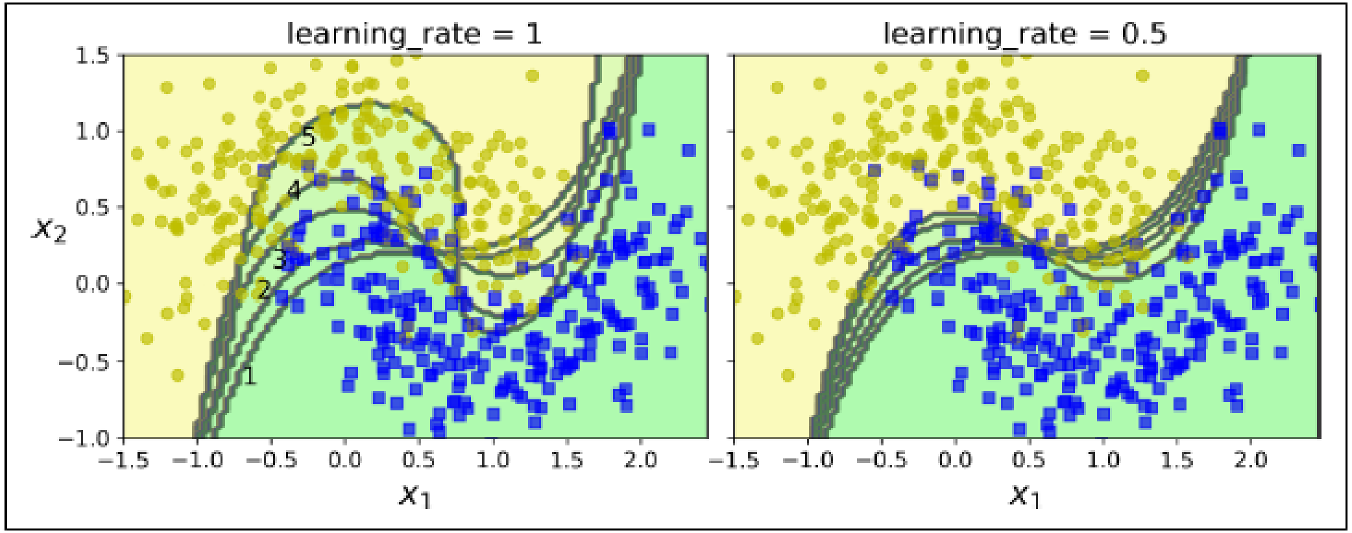


*Figure 7-8. Decision boundaries of consecutive predictors*

## AdaBoost

1. Each instance weight $w^{(i)}$ is initially set to 1/*m*.

*Equation 7-1. Weighted error rate of the jth predictor*

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

2. Error rate is used to update predictor weight. 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.

*Equation 7-2. Predictor weight*

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

## AdaBoost

3. AdaBoost update the instance weights, which boosts the weights of the misclassified instances.

*Equation 7-3. Weight update rule*

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

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

4. Then all the instance weights are normalized (i.e., divided by $\sum_{i=1}^{m} w^{(i)}$ )
5. A new predictor is trained using the updated weights, and the whole process is repeated (the new predictor's weight is computed, the instance weights are updated, then another predictor is trained, and so on).

The algorithm stops when the desired number of predictors is reached, or when a perfect predictor is found.

## AdaBoost - Prediction

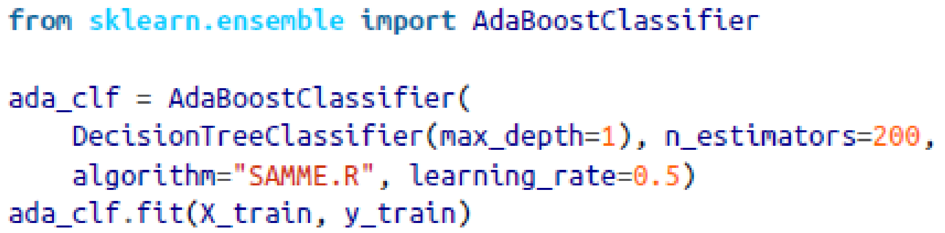
To make predictions, AdaBoost simply computes the predictions of all the predictors and weighs them using the predictor weights *αj*. The predicted class is the one that receives the majority of weighted votes

*Equation 7-4. AdaBoost predictions*

$$\hat{y}(\mathbf{x}) = \underset{k}{\operatorname{argmax}} \sum_{\substack{j=1 \\ \hat{y}_j(\mathbf{x}) = k}}^{N} \alpha_j \quad \text{where } N \text{ is the number of predictors.}$$

## AdaBoost - Implementation

Scikit-Learn uses a multiclass version of AdaBoost called *SAMME* (which stands for *Stagewise Additive Modeling using a Multiclass Exponential loss function*)



```python
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. In other words, a tree composed of a single decision node plus two leaf nodes.

# Ensemble Learning and Random Forests

Gradient  Boosting

## Ensembling: In Technical Terms

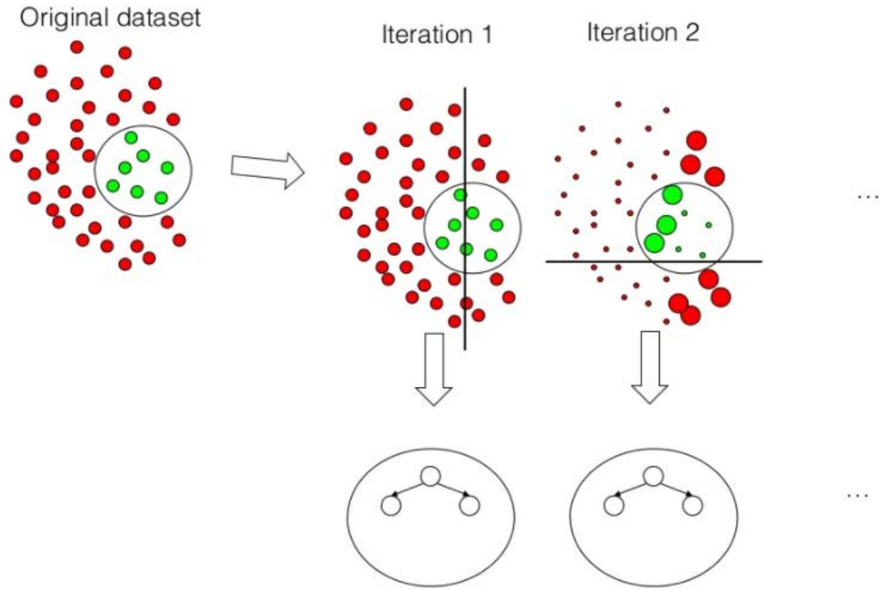
**Bagging/Pasting (**1996) also known as the aggregation of bootstrap samples, trains multiples of the  same type of model (decision trees) using bootstrap samples drawn from the training dataset, and combines the prediction results (average for regression, vote for classification). When samples are taking out with replacement, it is called bagging. When samples are taking out without replacement, it is called pasting.

**Boosting** (1999) uses weak learners (typically decision tree classifers) to build a forward stagewise additive model that applies the weak classifiers one by one to repeatedly reweighted versions of the data, so that each new weak learner tries to correct the error of the previous learners.

## Ensembling: Boosting

*Boosting* (originally called *hypothesis boosting*) refers to any Ensemble method that can combine several weak learners into a strong learner.

## Ensembling: Boosting Illustration

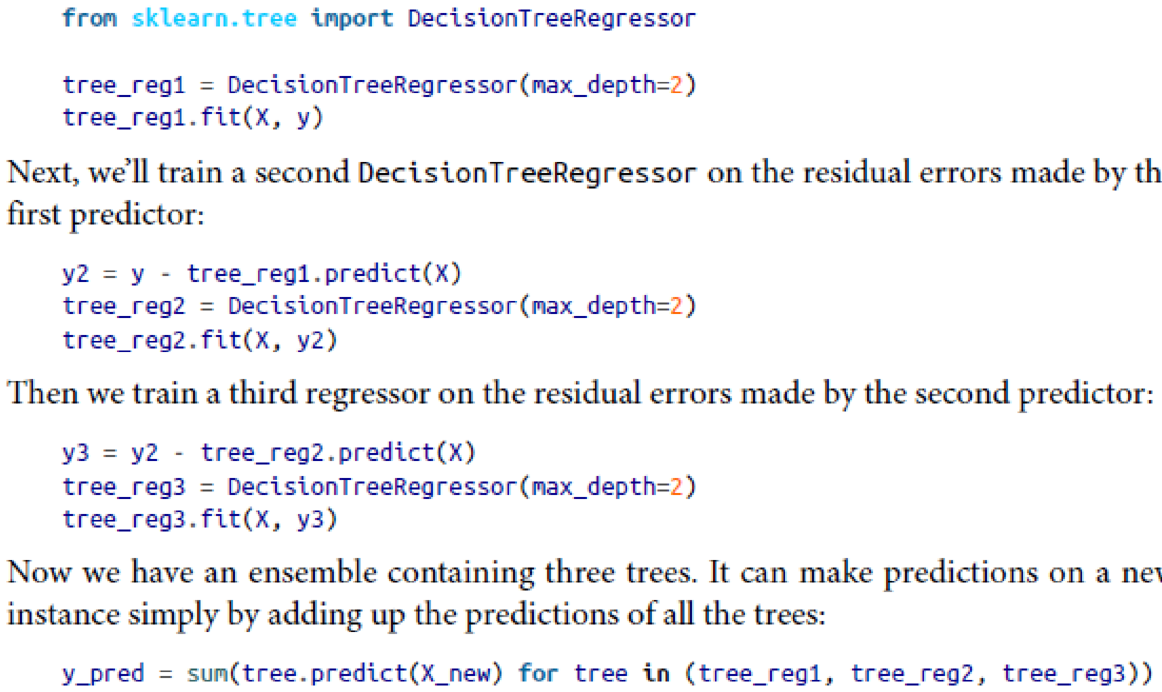


## Boosting: Gradient Boosting

Another very popular boosting algorithm is *Gradient Boosting*. 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.

## Boosting: Gradient Boosting Example



```
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)
```

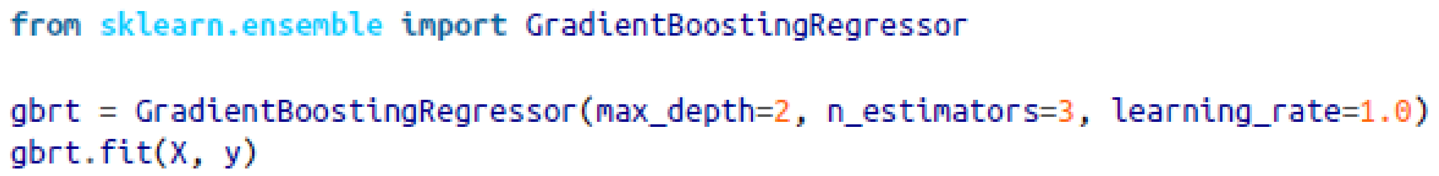
Then we train a third regressor on the residual errors made by the second predictor:

```
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))
```

## Boosting: Gradient Boosting Example



```
from sklearn.ensemble import GradientBoostingRegressor

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

# Boosting: Gradient Boosting Example

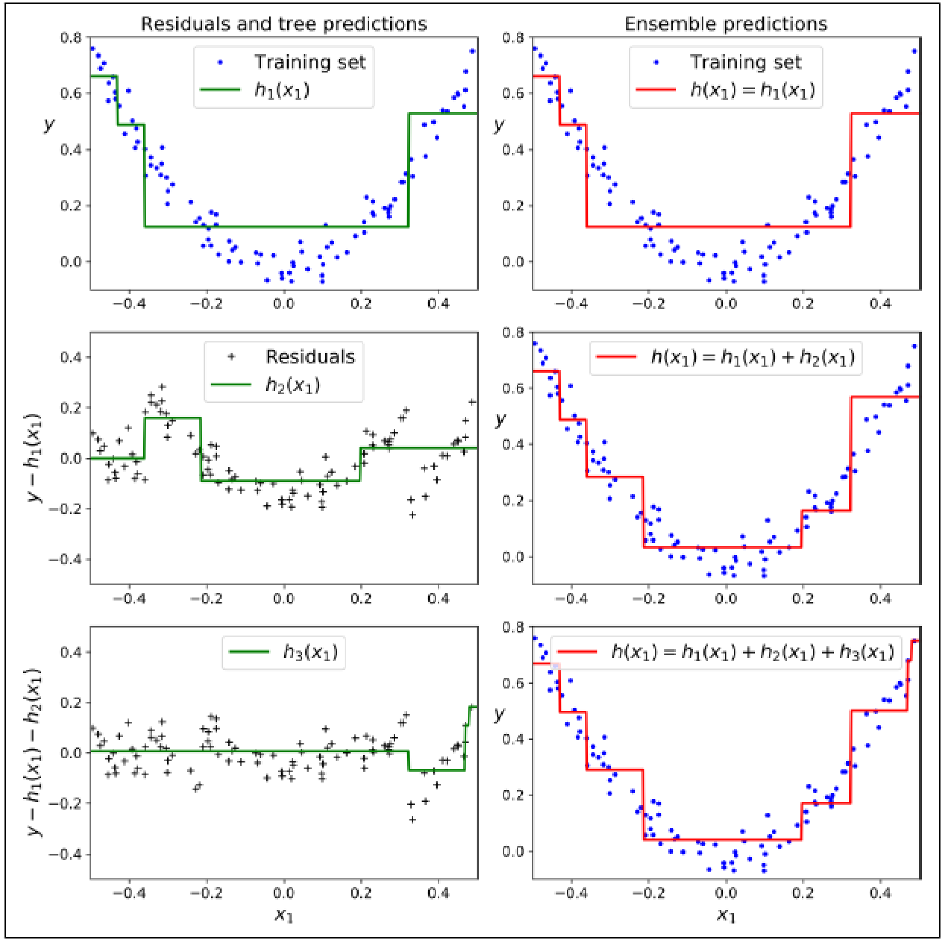


*Figure 7-9. In this depiction of Gradient Boosting, the first predictor (top left) is trained normally, then each consecutive predictor (middle left and lower left) is trained on the previous predictor's residuals; the right column shows the resulting ensemble's predictions*

## Gradient Boosting: Learning Rate

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.***

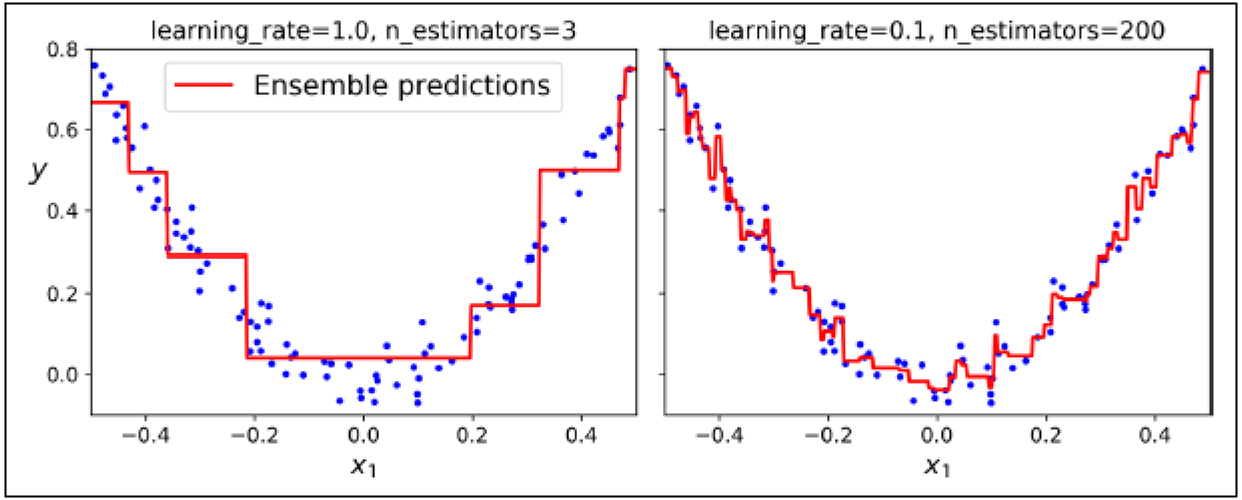


*Figure 7-10. GBRT ensembles with not enough predictors (left) and too many (right)*

## Gradient Boosting: Optimal Number of Tress

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 (with one tree, two trees, etc.).

```
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)
```

## Tuning the number of tress using early stopping

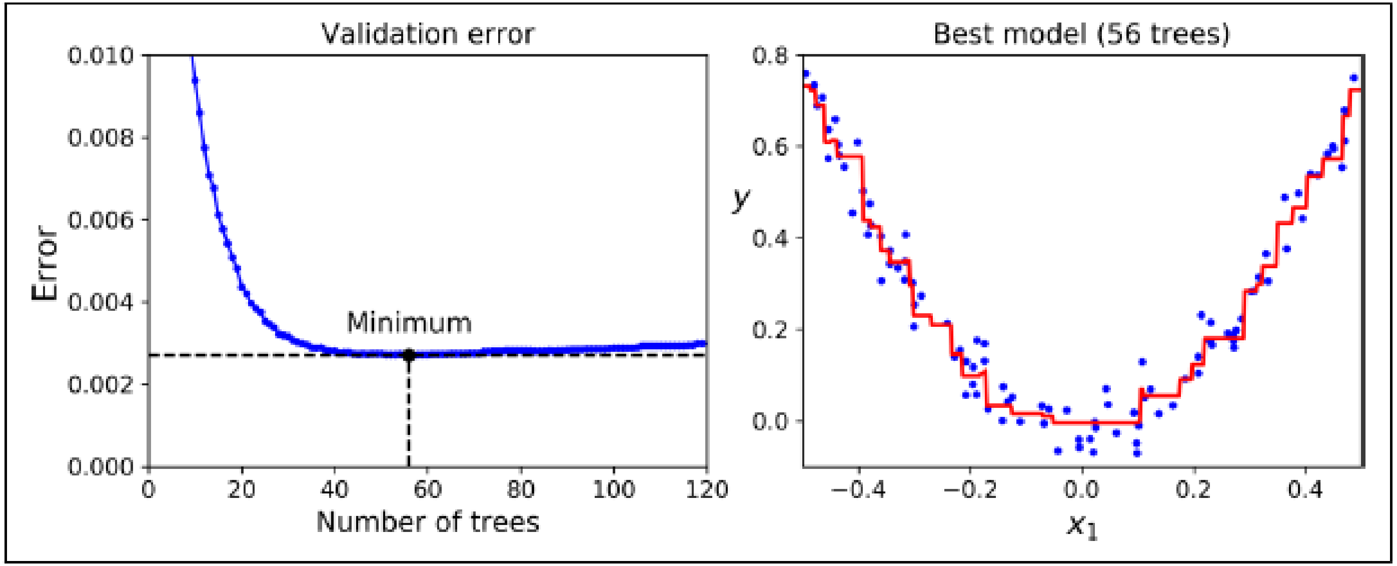
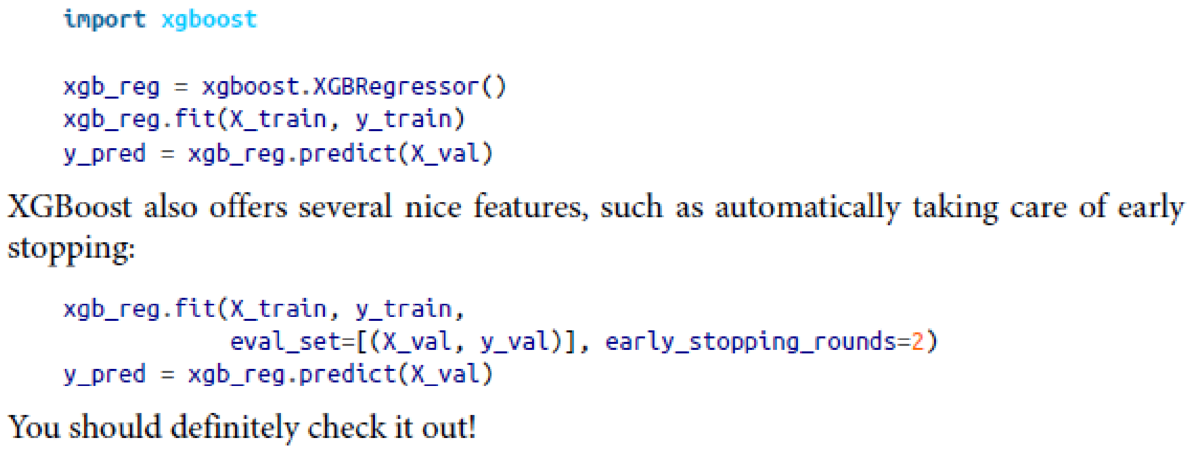


*Figure 7-11. Tuning the number of trees using early stopping*

## XGBoost

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.

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)
```

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)
```

You should definitely check it out!

# Ensemble Learning and Random Forests

Stacking

# Ensembling: Pros vs Cons

Ensembling in general reduces variance and increases accuracy, especially when using decision trees as base learners.  Ensemble models generally win online Kaggle competitions, even to this day.

Bagging, in particular, is **robust** against outliers.  Boosting is **flexible** and can be used with any loss function.   However, without careful pruning of the trees, ensemble methods will take longer to run than a single classifer and will be harder to interpret than a generalized linear model.

Finally, a third type of ensembling has been gaining favor, called **stacking.**  This ensembles a diverse group of strong learners and trains a second-level “metalearner” to learn the optimal combination of the base learners.  This is frequently used in conjunction with  neural networks.

## Ensemble: Stacking

- The last Ensemble method we will discuss is called *stacking* (short for *stacked generalization*)

- 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?

## Ensemble: Stacking

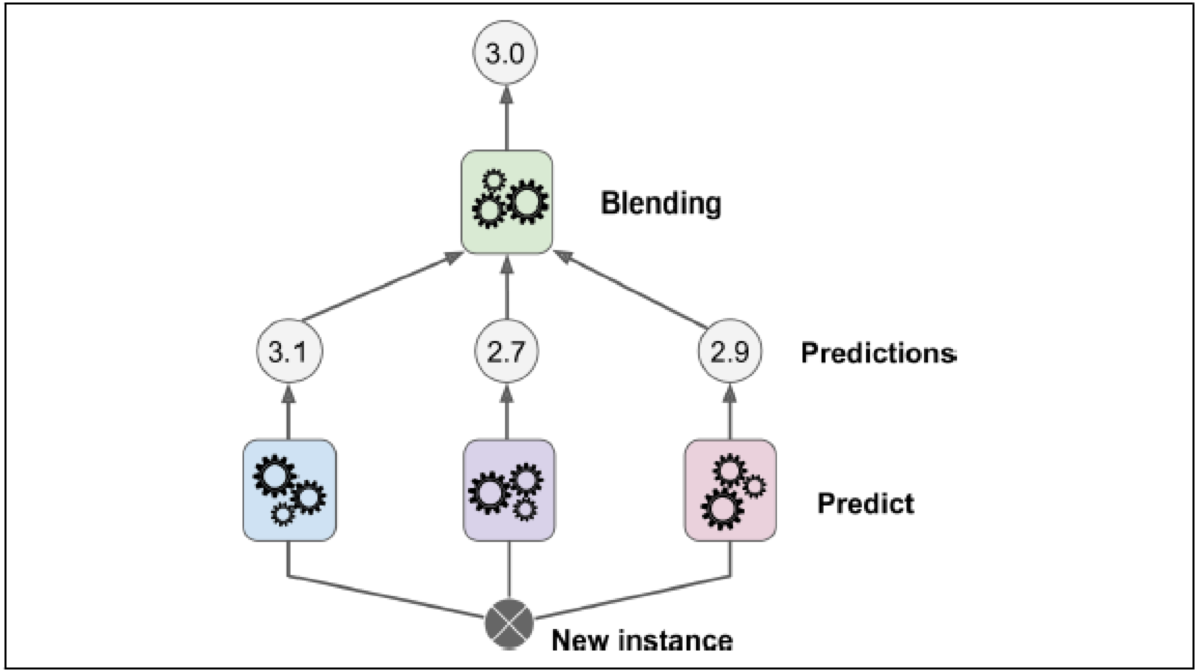


*Figure 7-12. Aggregating predictions using a blending predictor*

## Ensemble: Stacking

Let’s see how it  works. First, the training set is split into two subsets. The first subset is used to train the predictors in the first layer

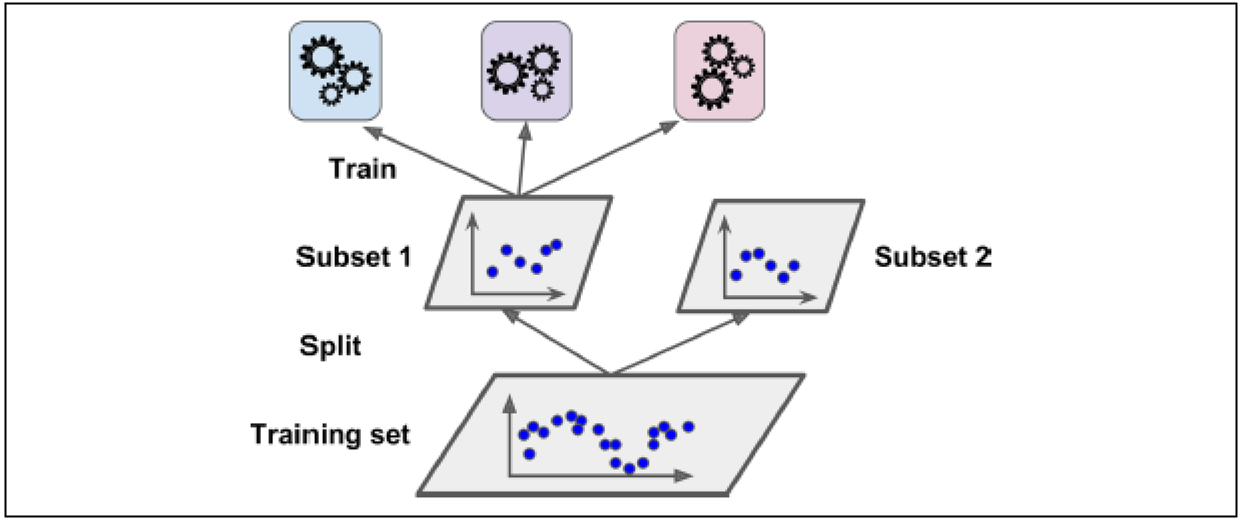


*Figure 7-13. Training the first layer*

## Ensemble: Stacking

- For each instance in the hold-out set (Subset2), there are three predicted values.
- We can create a new training set using these predicted values as input features (which makes this new training set 3D), and keeping the target values.
- The blender is trained on this new training set, so it learns to predict the target value, given the first layer's predictions.

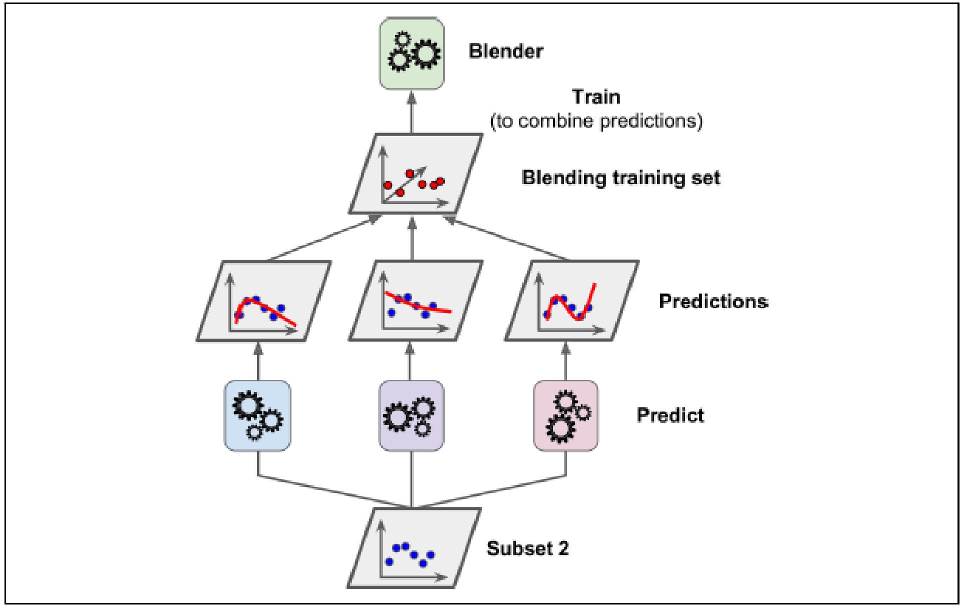


*Figure 7-14. Training the blender*

# Ensemble: Multi Layer Stacking

Unfortunately, Scikit-Learn does not support stacking directly, but it is not too hard to roll out your own implementation (see the following exercises). Alternatively, you can use an open source implementation such as DESlib

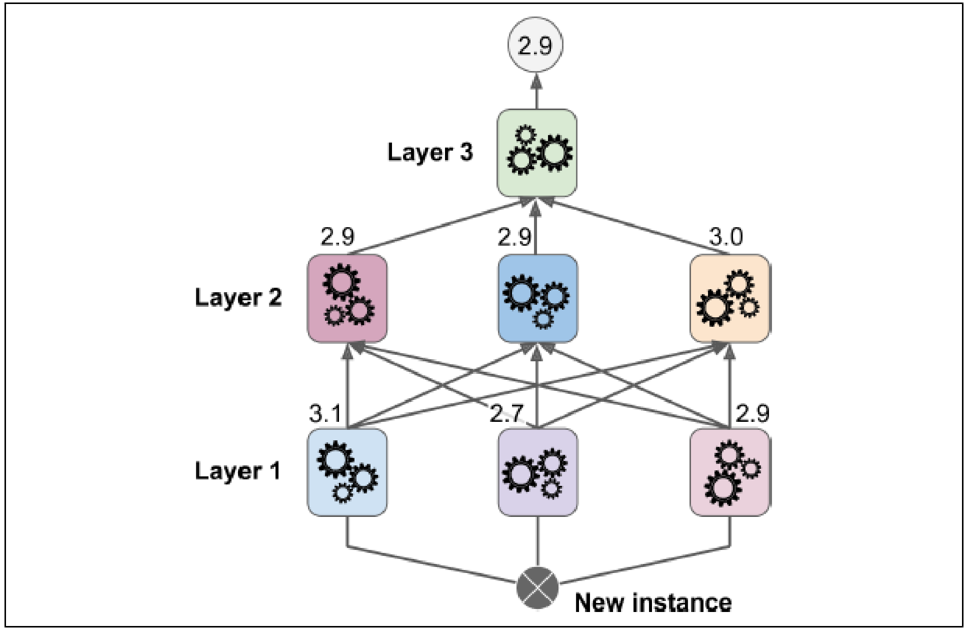


*Figure 7-15. Predictions in a multilayer stacking ensemble*

## Ensembling: Key Takeaways

- Ensembling: Majority Votes (Hard votes vs. Soft votes)
- Ensembling: Bagging/Pasting
- Ensembling: Boosting (AdaBoost and Gradient Boosting)
- Ensembling: Stacking

Note: bagging/boosting.  Still using the same TYPE Of model.

Stacking = uses DIFFERENT TYPES OF MODESL (SVM, GBM, ETC)

- **Random Forest** is an ensemble of Decision Trees, generally trained via the bagging method