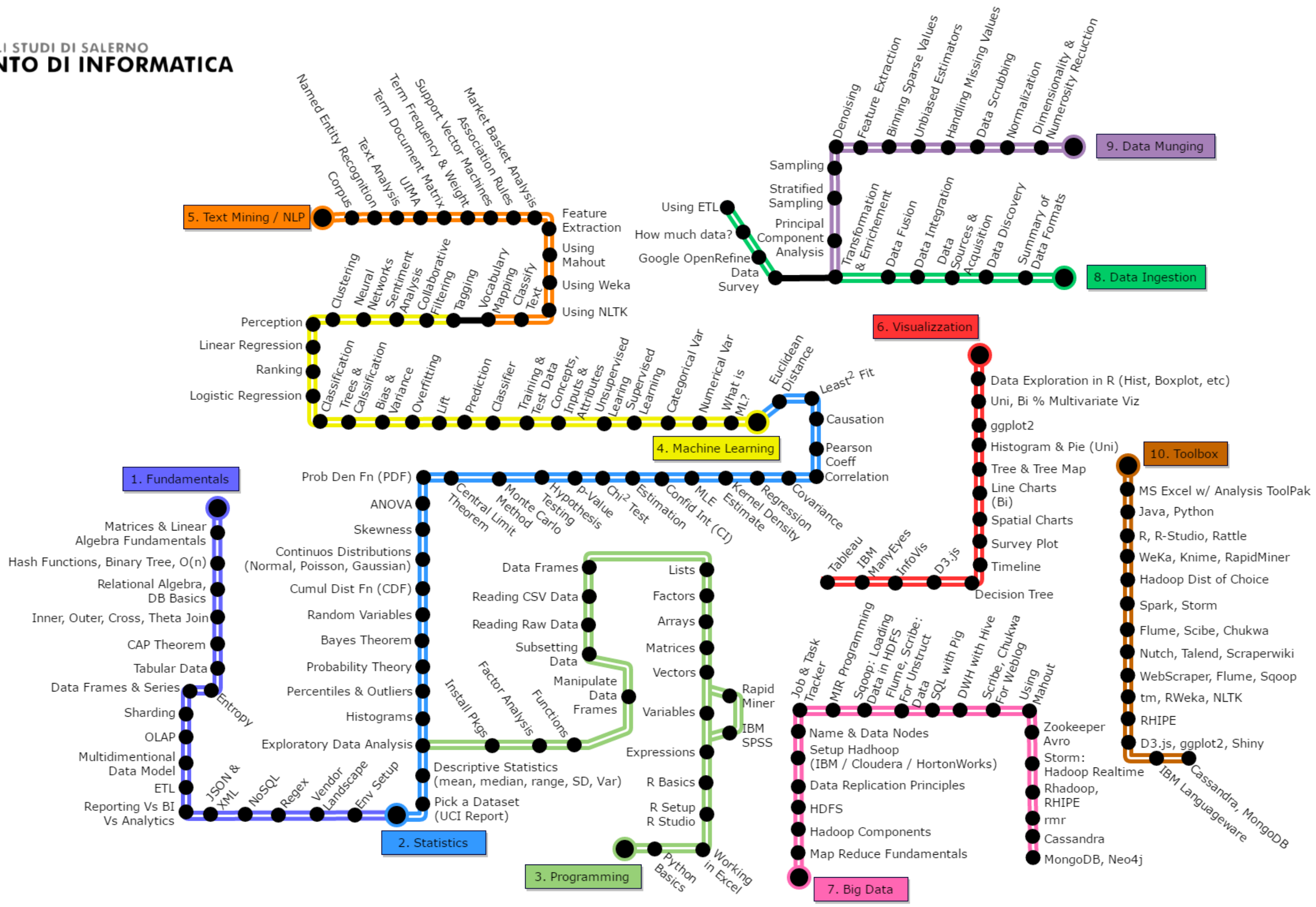
UNIVERSITÀ DEGLI STUDI DI SALERNO
DIPARTIMENTO DI INFORMATICA



Fondamenti di Data Science e Machine Learning

Random Forests (Chapter 7 Geron's Book)

Aurelien Geron: «Hands on Machine Learning with Scikit Learn and TensorFlow, O'Reilly ed.

Prof. Giuseppe Polese, aa 2024-25

Outline

- Ensemble Learning and Random Forests
  - Voting Classifiers
  - Bagging and Pasting
  - Out-of-Bag Evaluation
  - Random Patches and Random Subspaces
- Random Forests
  - Extra-Trees
  - Feature Importance
  - Boosting
  - AdaBoost
  - Gradient Boosting
  - Stacking

Ensemble Learning

- Suppose we ask a complex question to thousands of random people, then aggregate their answers
  - In many cases you will find that this aggregated answer is better than an expert’s answer
  - This is called the wisdom of the crowd
- If we aggregate the predictions of a group of predictors (such as classifiers or regressors), we often get better predictions than with the best individual predictor
  - A group of predictors is called an ensemble
  - this technique is called Ensemble Learning
  - an Ensemble Learning algorithm is called an Ensemble method

Random Forests

- For example, we can train a group of Decision Tree classifiers, each on a different random subset of the training set
  - To make predictions, you just obtain the predictions of all individual trees, then predict the class that gets the most votes

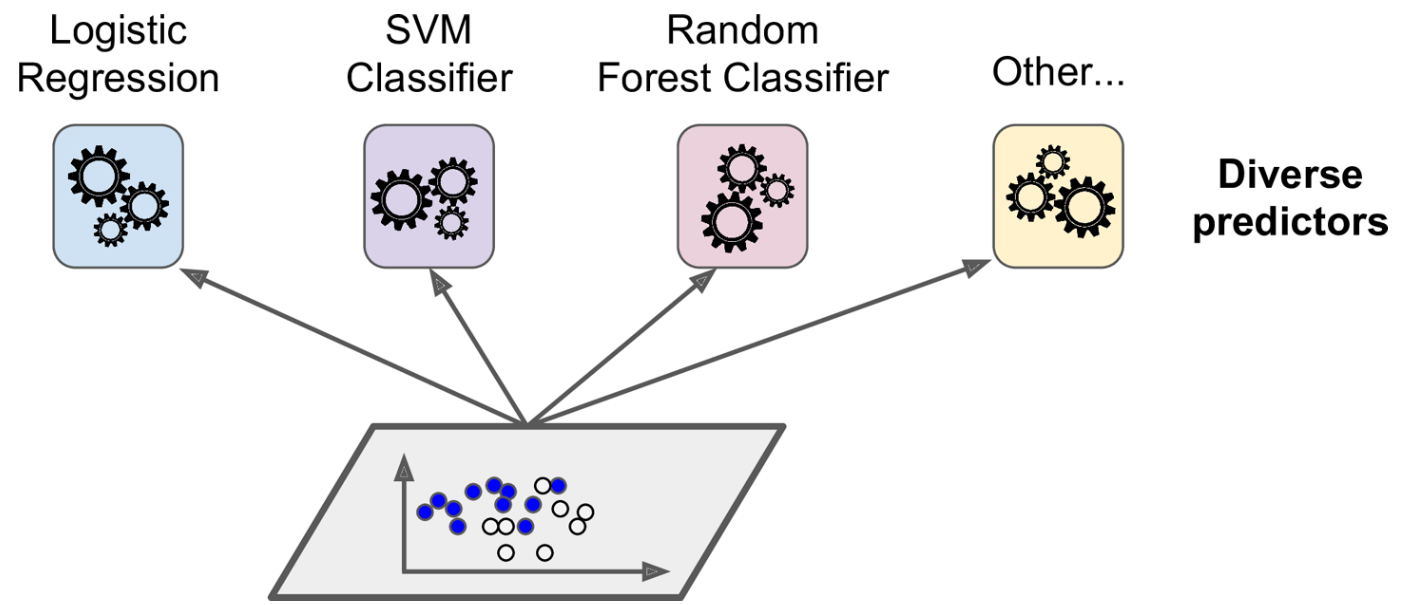
Such an ensemble of Decision Trees is called a

Random Forest

  - this is one of the most powerful Machine Learning algorithms available today

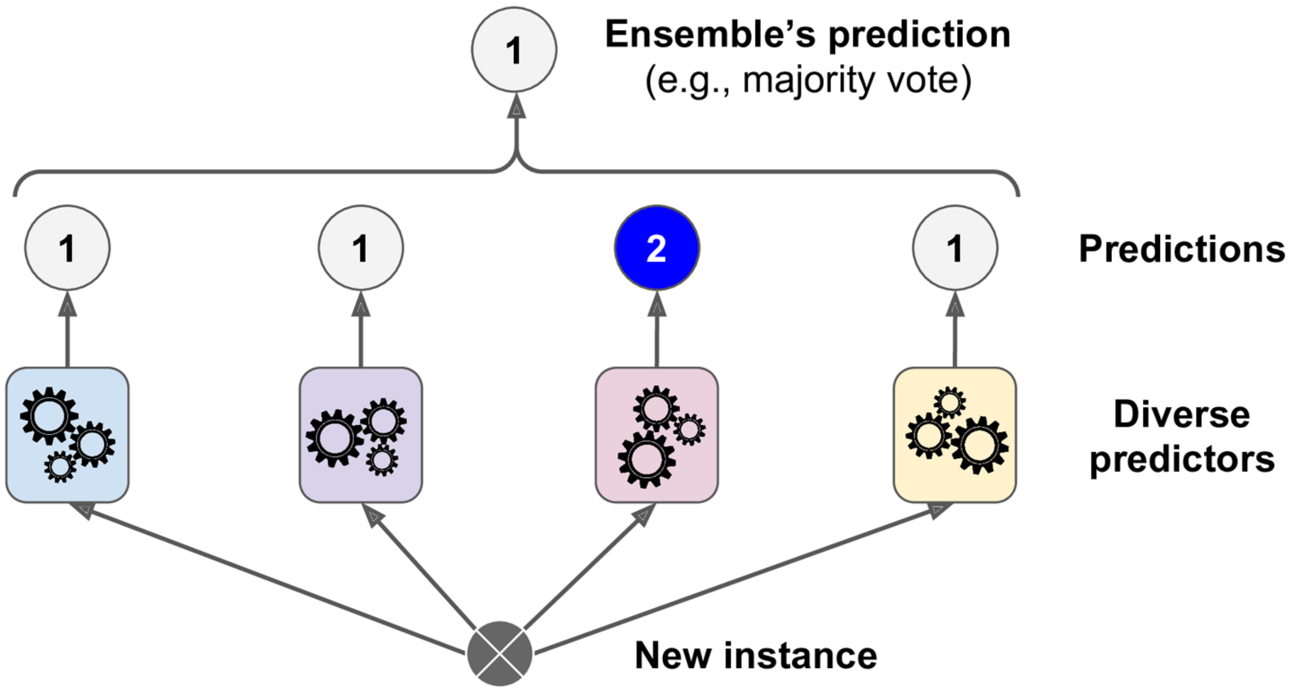
Voting Classifiers (1)

- Suppose we have trained a few classifiers, each one achieving about 80% accuracy
  - We may have a Logistic Regression classifier, an SVM classifier, a Random Forest classifier, a K-Nearest Neighbors classifier, and perhaps a few more



Voting Classifiers (2)

- 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 (3)

- Somewhat surprisingly, this voting classifier often achieves a higher accuracy than the best classifier in the ensemble
- Even if each classifier is a weak learner
  - meaning it does only slightly better than random guessing
- The ensemble can still be a strong learner
  - achieving high accuracy

Voting Classifiers: An example (1)

- Suppose we have a slightly biased coin that has a 51% chance of heads, and 49% chance of tails
  - If we toss it 1,000 times, we will generally get more or less 510 heads and 490 tails, and hence a majority of heads
- We will find that the probability of obtaining a majority of heads after 1,000 tosses is close to 75%
  - The more we toss the coin, the higher the probability
    - e.g., with 10,000 tosses, the probability climbs over 97%
- This is due to the law of large numbers
  - as we keep tossing the coin, the ratio of heads gets closer and closer to the probability of heads (51%)

Voting Classifiers: An example (2)

- Suppose you build an ensemble containing 1,000 classifiers that are individually correct only 51% of the time
  - barely better than random guessing
- If you predict the majority voted class, you can hope for up to 75% accuracy!
- This is only true if all classifiers are perfectly independent, making uncorrelated errors, which is clearly not the case since they are trained on the same data
  - They are likely to make the same types of errors
  - there will be many majority votes for the wrong class, reducing the ensemble's accuracy

Voting Classifiers in Scikit-Learn (1)

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

```python
from sklearn.model_selection import train_test_split
from sklearn.datasets import make_moons
from sklearn.ensemble import RandomForestClassifier
from sklearn.ensemble import VotingClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score
#Define make_moons dataset
X, y = make_moons(n_samples=500, noise=0.30, random_state=42)
#Splitting in training and Testing
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=42)
#Usage of LogisticRegression method
log_clf = LogisticRegression(solver="liblinear", random_state=42)
#Usage of RandomForestClassifier method
rnd_clf = RandomForestClassifier(n_estimators=10, random_state=42)
#Usage of SVC method
svm_clf = SVC(gamma="auto", random_state=42)
#Defining hard voting classifier
voting_clf = VotingClassifier(
    estimators=[('lr', log_clf), ('rf', rnd_clf), ('svc', svm_clf)],
    voting='hard')
voting_clf.fit(X_train, y_train)
#Print 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 0.872
SVC 0.888
VotingClassifier 0.896

Process finished with exit code 0
```

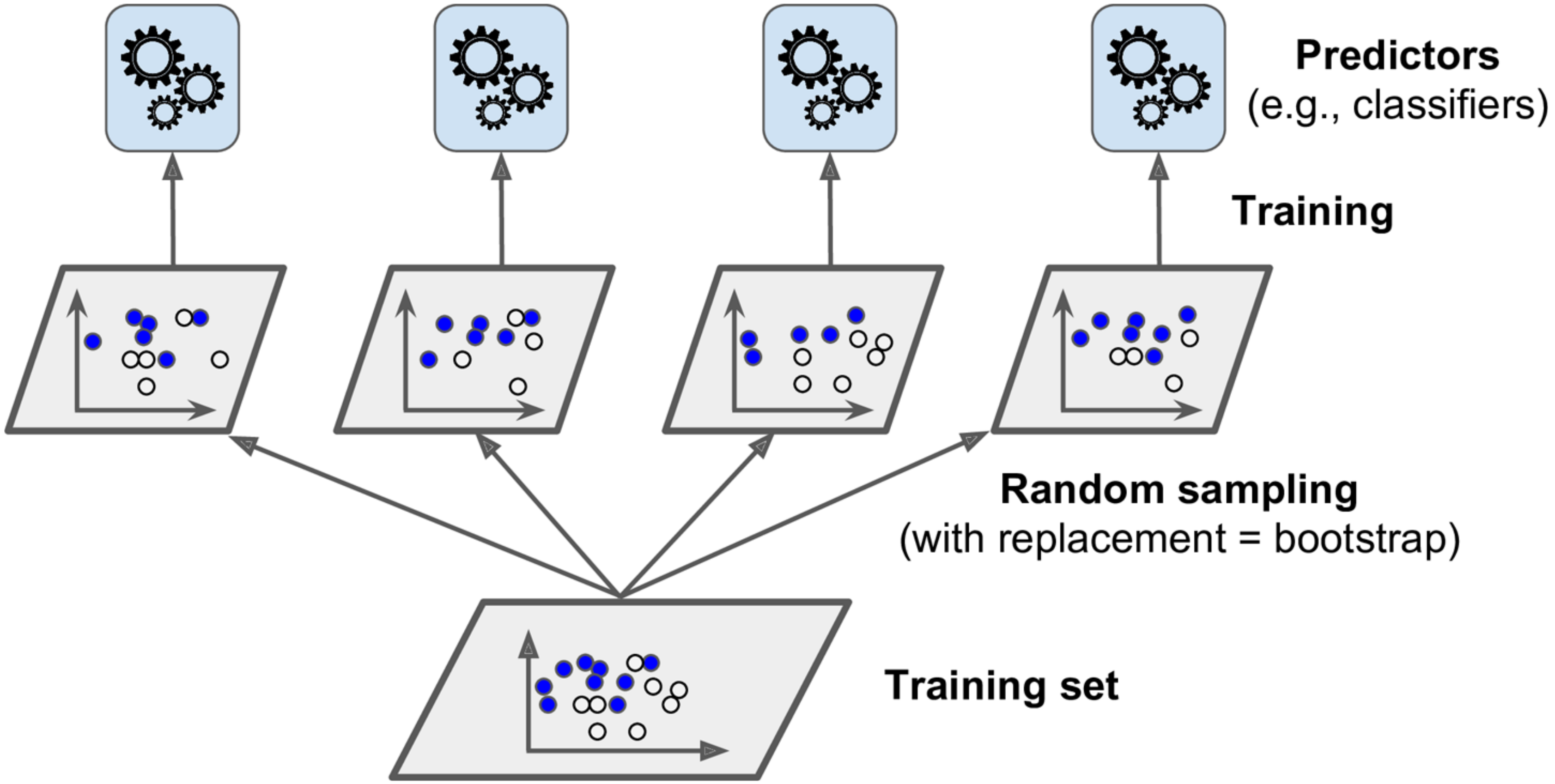
Voting Classifiers in Scikit-Learn (2)

- The voting classifier slightly outperforms all the individual classifiers
- If all classifiers are able to estimate class probabilities (i.e., they 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
  - All you need to do is replace *voting="hard"* with *voting="soft"* and ensure that all classifiers can estimate class probabilities
- This is not the case of the SVC class by default
  - you need to set its probability hyperparameter to *True*
    - this will make the SVC class use cross-validation to estimate class probabilities, slowing down training, and it will add a *predict_proba()* method

Bagging and Pasting (1)

- A different approach is to use the same training algorithm for every predictor, but to train them on different random subsets of the training set
  - When sampling is performed with replacement (allows training instances to be sampled several times for the same predictor), this method is called bagging
  - When sampling is performed without replacement for the same predictor, it is called pasting
- Thus, both bagging and pasting allow training instances to be sampled several times, but while bagging allows it also for the same predictor, pasting allows it only across multiple predictors.

Bagging and Pasting (2)



Bagging and Pasting (3)

- Once all predictors are trained, the ensemble can make a prediction for a new instance by simply aggregating the predictions of all predictors
- The aggregation function is typically the *statistical mode* for classification, or the average for regression
  - i.e., the most frequent prediction, just like a hard voting classifier
- Each individual predictor has a higher bias than if it were trained on the original training set, but aggregation reduces both bias and variance

Bagging and Pasting (4)

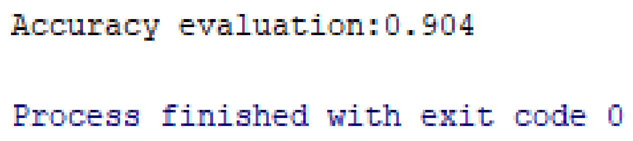
- Generally, the net result is that the ensemble has a similar bias but a lower variance than a single predictor trained on the original training set
- In general
  - predictors can all be trained in parallel, via different CPU cores or even different servers.
  - predictions can be made in parallel
- This is one of the reasons why bagging and pasting are such popular methods – they scale very well!

Bagging and Pasting in Scikit-Learn (1)

- **Scikit-Learn** offers a simple API for both bagging and pasting with the BaggingClassifier class
- The code in the next slide
  - trains an ensemble of *500* Decision Tree classifiers
  - each trained on *100* training instances randomly sampled from the training set *make_moons* with replacement
- This is an example of bagging
  - if you want to use pasting instead, just set `bootstrap=False`
- The `n_jobs` parameter tells **Scikit-Learn** the number of CPU cores to use for training and predictions
  - –1 tells **Scikit-Learn** to use all available cores

Bagging and Pasting in Scikit-Learn (2)

```
from sklearn.ensemble import BaggingClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import train_test_split
from sklearn.datasets import make_moons
from sklearn.metrics import accuracy_score
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)
#Define BaggingClassifier
bag_clf = BaggingClassifier(
    DecisionTreeClassifier(random_state=42), n_estimators=500,
    max_samples=100, bootstrap=True, n_jobs=-1, random_state=42)
bag_clf.fit(X_train, y_train)
y_pred = bag_clf.predict(X_test)
#Evaluation of accuracy
print("Accuracy evaluation:{}".format(accuracy_score(y_test,
y_pred)))
```

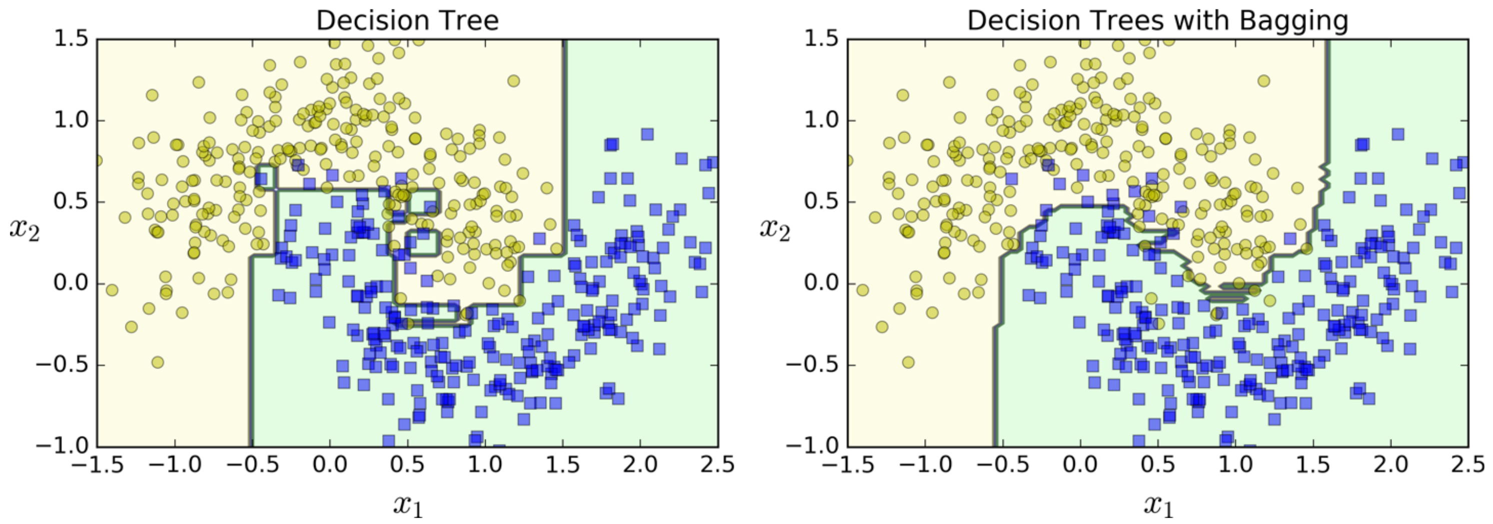


```
Accuracy evaluation:0.904

Process finished with exit code 0
```

Bagging and Pasting in Scikit-Learn (3)

- The Figure compares the decision boundary of a single Decision Tree with the decision boundary of a bagging ensemble of 500 trees



- As you can see, 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
    - It makes roughly the same number of errors on the training set, but the decision boundary is less irregular

Summarizing 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
  - This also means that predictors end up being less correlated so the ensemble’s variance is reduced
- Bagging often results in better models
  - This explains why it is generally preferred
- However, if you have spare time and CPU power you can use cross-validation to evaluate both bagging and pasting and select the one that works best

Out-of-Bag Evaluation (1)

- With bagging
  - some instances may be sampled several times for any given predictor, while
  - others may not be sampled at all
- By default a *BaggingClassifier* samples *m* training instances with replacement (`bootstrap=True`)
  - where *m* is the size of the training set
- This means that
  - only about 63% of the training instances are sampled on average for each predictor
  - the remaining 37% are called out-of-bag (oob) instances
- Note that they are not the same 37% for all predictors

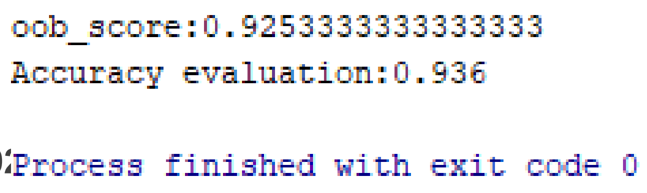
Out-of-Bag Evaluation (2)

- Since a predictor never sees the oob instances during training, it can be evaluated on these instances, without the need for a separate validation set or cross-validation
  - You can evaluate the ensemble itself by averaging out the oob evaluations of each predictor
- In **Scikit-Learn**, you can set `oob_score=True` when creating a BaggingClassifier to request an automatic oob evaluation after training

Out-of-Bag Evaluation in Scikit-Learn

- The resulting evaluation score is available through the `oob_score_` variable

```python
from sklearn.ensemble import BaggingClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import train_test_split
from sklearn.datasets import make_moons
from sklearn.metrics import accuracy_score
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)
#Define BaggingClassifier with oob_score=True
bag_clf = BaggingClassifier(
    DecisionTreeClassifier(random_state=42), n_estimators=500,
    max_samples=100, bootstrap=True, n_jobs=-1, oob_score=True)
bag_clf.fit(X_train, y_train)
y_pred = bag_clf.predict(X_test)
#Evaluation of oob_score
print("oob_score:{}".format(bag_clf.oob_score_))
#Evaluation of accuracy
print("Accuracy evaluation:{}".format(accuracy_score(y_test, y_pred)))
```



```
oob_score:0.9253333333333333
Accuracy evaluation:0.936

Process finished with exit code 0
```

Random Patches and Random Subspaces (1)

- The *BaggingClassifier* class supports sampling the features as well
  - This is controlled by two hyperparameters:
    - `max_features`
    - `bootstrap_features`
- They work the same way as `max_samples` and `bootstrap`, but for feature sampling instead of instance sampling
  - Each predictor will be trained on a random subset of the input features

Random Patches and Random Subspaces (2)

- This is particularly useful when you are dealing with high-dimensional inputs
  - such as images
- Sampling both training instances and features is called the Random Patches method
- Keeping all training instances (i.e., `bootstrap=False` and `max_samples=1.0`) but sampling features (i.e., `bootstrap_features=True` and/or `max_features` smaller than 1.0) is called the Random Subspaces method

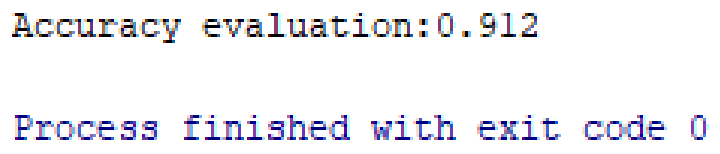
Random Forests

- A 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
- You can use the RandomForestClassifier class
  - Instead of building a BaggingClassifier and passing it a DecisionTreeClassifier
  - RandomForestClassifier is more convenient and optimized for Decision Trees

Random Forests in Scikit-Learn

- The following code trains a Random Forest classifier with 500 trees (each limited to maximum 16 nodes), using all available CPU cores on `make_moons` dataset

```
from sklearn.model_selection import train_test_split
from sklearn.datasets import make_moons
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score
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)
#Define Random Forest Classifier
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)
#Evaluation of accuracy
print("Accuracy evaluation:{}".format(accuracy_score(y_test, y_pred_rf)))
```



```
Accuracy evaluation:0.912

Process finished with exit code 0
```

Random Forest Algorithm

- The Random Forest algorithm introduces extra randomness when growing trees
  - instead of searching for the very best feature when splitting a node, it searches for the best feature among a random subset of features
- This results in a greater tree diversity, which trades a higher bias for a lower variance, generally yielding an overall better model

Extra-Trees (1)

- 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
    - like regular Decision Trees do
- A forest of such extremely random trees is simply called an Extremely Randomized Trees ensemble
  - or Extra-Trees for short

Extra-Trees (2)

- This trades more bias for a lower variance
  - It also makes Extra-Trees much faster to train than regular Random Forests
    - Finding the best possible threshold for each feature at every node is one of the most time-consuming tasks of growing a tree

- It is possible to create an Extra-Trees classifier using Scikit-Learn's `ExtraTreeClassifier` class
  - Its API is identical to the `RandomForestClassifier` class

Feature Importance

- Another great quality of Random Forests is that they make it easy to measure the relative importance of each feature

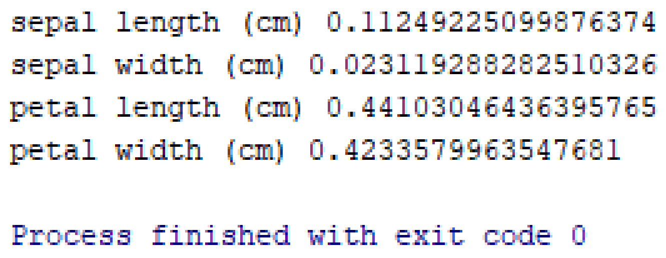
- **Scikit-Learn** measures a feature's importance by looking at how much the tree nodes that use that feature reduce impurity on average (across all trees in the forest)
  - It is a weighted average, where each node's weight is equal to the number of training samples that are associated with it

- **Scikit-Learn** computes this score automatically for each feature after training, then it scales the results so that the sum of all importances is equal to 1
  - Use the `feature_importances_` variable for accessing the result

Feature Importance in Scikit-Learn

- The following code trains a RandomForestClassifier on the iris dataset and outputs each feature’s importance
- It seems that the most important features are the petal length (44%) and width (42%), while sepal length and width are rather unimportant in comparison (11% and 2%, respectively)

```python
from sklearn.datasets import load_iris
from sklearn.ensemble import RandomForestClassifier
iris = load_iris()
#Define Random Forest Classifier
rnd_clf = RandomForestClassifier(n_estimators=500, n_jobs=-1,
random_state=42)
rnd_clf.fit(iris["data"], iris["target"])
#Feature evaluations
for name, score in zip(iris["feature_names"],
rnd_clf.feature_importances_):
    print(name, score)
```



```
sepal length (cm) 0.11249225099876374
sepal width (cm) 0.023119288282510326
petal length (cm) 0.44103046436395765
petal width (cm) 0.4233579963547681

Process finished with exit code 0
```

Boosting

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

- The general idea of most boosting methods is to train predictors sequentially
  - each trying to correct its predecessor

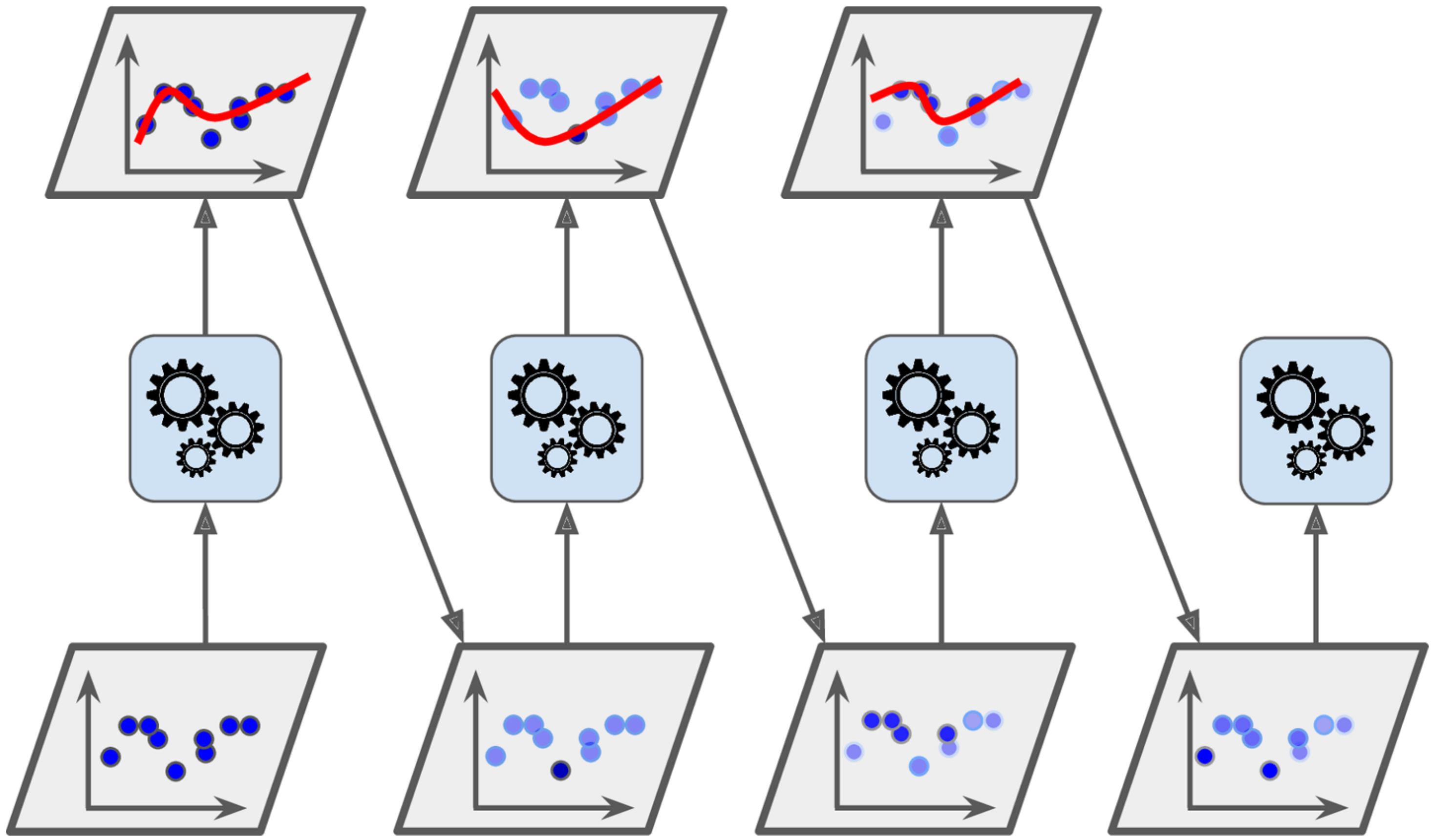
- There are many boosting methods available, but by far the most popular are
  - AdaBoost (short for Adaptive Boosting), and
  - Gradient Boosting

AdaBoost (1)

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

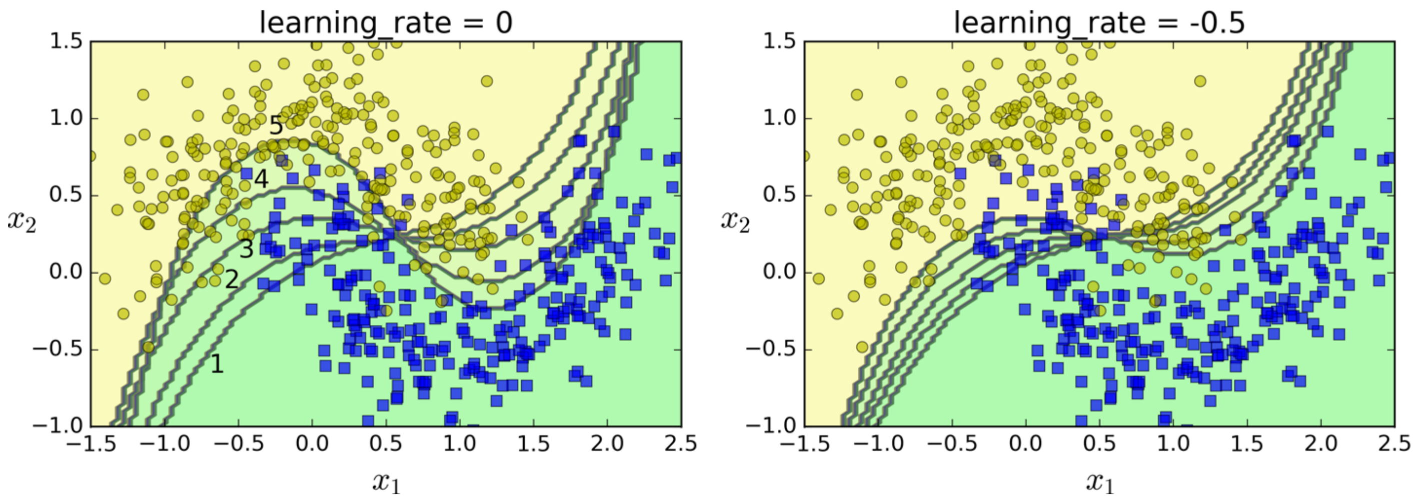
- For example, to build an AdaBoost classifier
  - A first base classifier (such as a Decision Tree) is trained and used to make predictions on the training set
    - The relative weight of misclassified training instances is then increased
  - A second classifier is trained using the updated weights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

AdaBoost (2)



AdaBoost: An example (1)

- The Figure shows the decision boundaries of five consecutive predictors on the **moons** dataset



- The first classifier gets many instances wrong, so their weights get boosted
- The second classifier therefore does a better job on these

AdaBoost: An example (2)

- The plot on the right represents the same sequence of predictors except that the learning rate is halved
  - The misclassified instance weights are boosted half as much at every iteration
- This sequential learning technique has some similarities with Gradient Descent
  - except that instead of tweaking a single predictor’s parameters to minimize a cost function, AdaBoost adds predictors to the ensemble, gradually making it better instances, and so on
- Once all predictors are trained, the ensemble makes predictions very much like bagging or pasting
  - except that predictors have different weights depending on their overall accuracy on the weighted training set

AdaBoost in Scikit-Learn (1)

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

- When there are just two classes, SAMME is equivalent to AdaBoost
- If the predictors can estimate class probabilities (i.e., they have a `predict_proba()` method), **Scikit-Learn** can use a variant called SAMME.R (the *R* stands for "Real")
  - It relies on class probabilities rather than predictions and generally performs better

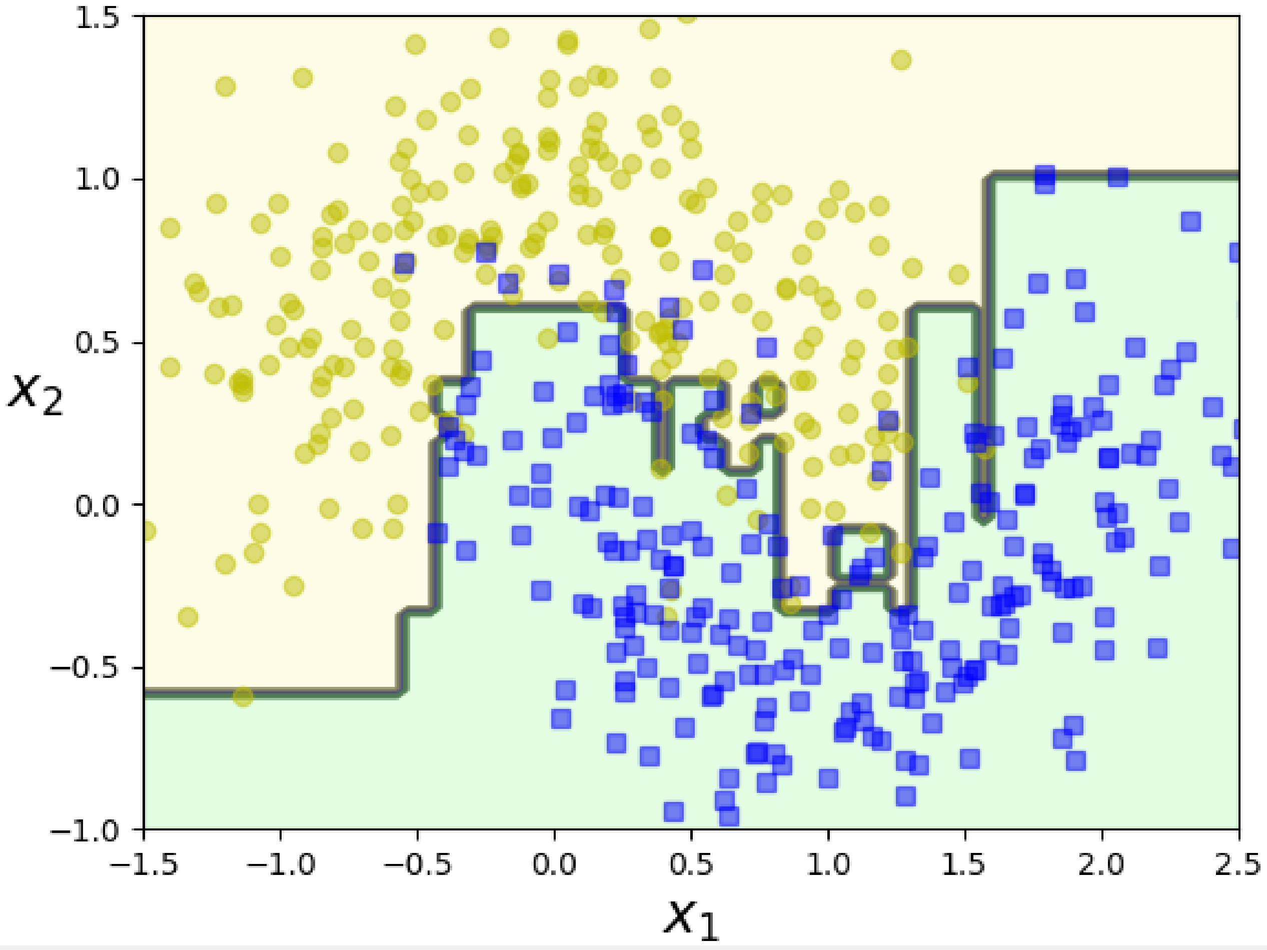
AdaBoost in Scikit-Learn (2)

- The following code trains an AdaBoost classifier based on 200 *Decision Stumps* using an AdaBoostClassifier class on **make_moons** dataset
- 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
- This is the default base estimator for the AdaBoostClassifier class

AdaBoost in Scikit-Learn (3)

```
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.datasets import make_moons
from matplotlib.colors import ListedColormap
import matplotlib.pyplot as plt
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import AdaBoostClassifier
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)
#Define AdaBoost classifier
ada_clf = AdaBoostClassifier(
DecisionTreeClassifier(max_depth=1, n_estimators=200,
algorithm="SAMME.R", learning_rate=0.5)
ada_clf.fit(X_train, y_train)
def plot_decision_boundary(clf, X, y, axes=[-1.5, 2.5, -1, 1.5], alpha=0.5, contour=True):
    x1s = np.linspace(axes[0], axes[1], 100)
    x2s = np.linspace(axes[2], axes[3], 100)
    x1, x2 = np.meshgrid(x1s, x2s)
    X_new = np.c_[x1.ravel(), x2.ravel()]
    y_pred = clf.predict(X_new).reshape(x1.shape)
    custom_cmap = ListedColormap(['#fafab0','#9898ff','#a0faa0'])
    plt.contourf(x1, x2, y_pred, alpha=0.3, cmap=custom_cmap,)
    if contour:
        custom_cmap2 = ListedColormap(['#7d7d58','#4c4c7f','#507d50'])
        plt.contour(x1, x2, y_pred, cmap=custom_cmap2, alpha=0.8)
    plt.plot(X[:, 0][y==0], X[:, 1][y==0], "yo", alpha=alpha)
    plt.plot(X[:, 0][y==1], X[:, 1][y==1], "bs", alpha=alpha)
    plt.axis(axes)
    plt.xlabel(r"$x_1$", fontsize=18)
    plt.ylabel(r"$x_2$", fontsize=18, rotation=0)
#Visualization of plotting
plot_decision_boundary(ada_clf, X, y)
plt.show()
```

AdaBoost in Scikit-Learn (3)



Gradient Boosting

- Another very popular Boosting algorithm is Gradient Boosting
  - It works by sequentially adding predictors to an ensemble, each one correcting its predecessor

- Instead of tweaking the instance weights at every iteration like AdaBoost does, this method tries to fit the new predictor to the residual errors made by the previous predictor

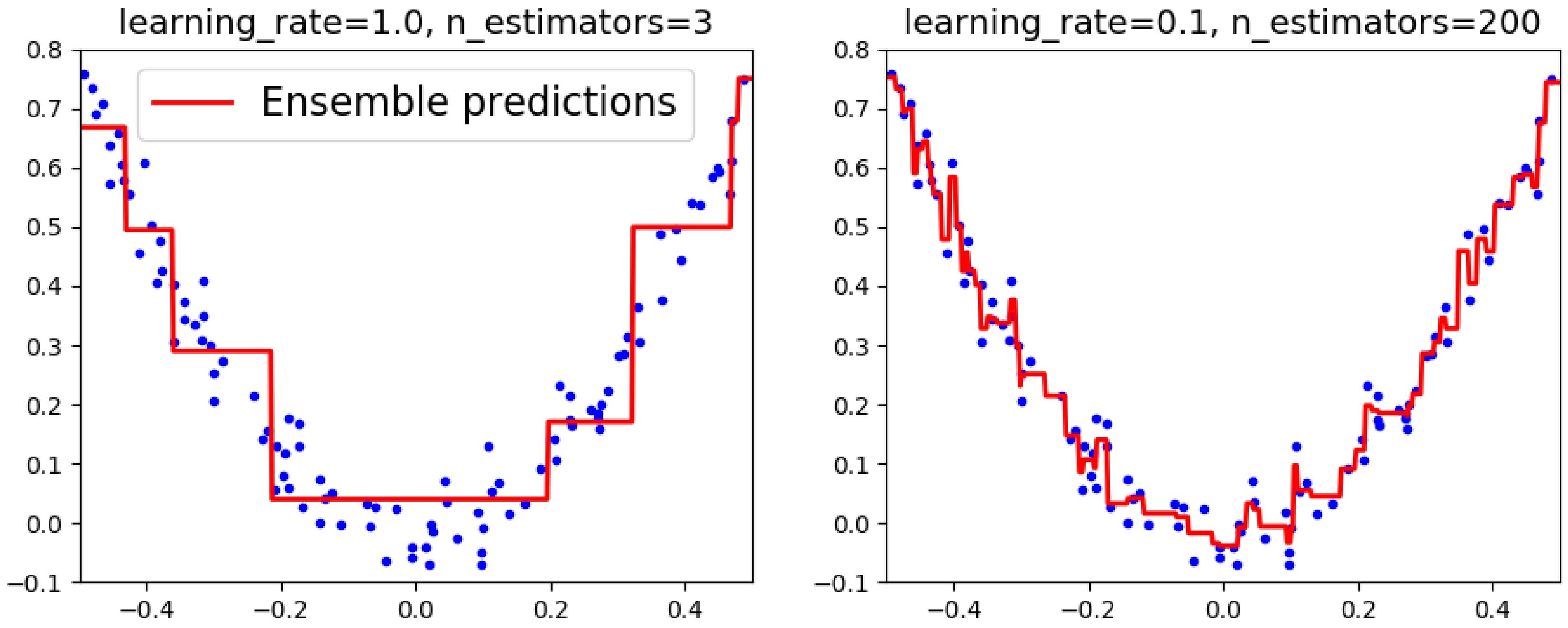
Gradient Boosting in Scikit-Learn (1)

- A simple way to train GBRT ensembles is to use Scikit-Learn's GradientBoostingRegressor class
  - Much like the RandomForestRegressor class, it has
    - hyperparameters to control the growth of Decision Trees
      - `max_depth`
      - `min_samples_leaf`,
      - and so on,
    - hyperparameters to control the ensemble training, such as the number of trees
      - `n_estimators`
- The following code shows the comparison of two GradientBoostingRegressor classifiers

Gradient Boosting in Scikit-Learn (2)

```python
import numpy as np
from sklearn.ensemble import GradientBoostingRegressor
import matplotlib.pyplot as plt
np.random.seed(42)
X = np.random.rand(100, 1) - 0.5
y = 3*X[:, 0]**2 + 0.05 * np.random.randn(100)
#Define GradientBoostingregressor classifier
gbrt = GradientBoostingRegressor(max_depth=2, n_estimators=3, learning_rate=1.0, random_state=42)
gbrt.fit(X, y)
#Define GradientBoostingregressor classifier
gbrt_slow = GradientBoostingRegressor(max_depth=2, n_estimators=200, learning_rate=0.1, random_state=42)
gbrt_slow.fit(X, y)
def plot_predictions(regressors, X, y, axes, label=None, style="r-", data_style="b.", data_label=None):
    x1 = np.linspace(axes[0], axes[1], 500)
    y_pred = sum(regressor.predict(x1.reshape(-1, 1)) for regressor in regressors)
    plt.plot(X[:, 0], y, data_style, label=data_label)
    plt.plot(x1, y_pred, style, linewidth=2, label=label)
    if label or data_label:
        plt.legend(loc="upper center", fontsize=16)
    plt.axis(axes)
plt.figure(figsize=(11,4))
#Plotting figures
plt.subplot(121)
plot_predictions([gbrt], X, y, axes=[-0.5, 0.5, -0.1, 0.8], label="Ensemble predictions")
plt.title("learning_rate={}, n_estimators={}".format(gbrt.learning_rate, gbrt.n_estimators), fontsize=14)
plt.subplot(122)
plot_predictions([gbrt_slow], X, y, axes=[-0.5, 0.5, -0.1, 0.8])
plt.title("learning_rate={}, n_estimators={}".format(gbrt_slow.learning_rate, gbrt_slow.n_estimators),
fontsize=14)
plt.show()
```

Gradient Boosting in Scikit-Learn (2)



Gradient Boosting Regularization

- The learning_rate hyperparameter scales the contribution of each tree
  - If you set it to a low value, such as 0.1, you will need more trees in the ensemble to fit the training set, but the predictions will usually generalize better
- This is a regularization technique called shrinkage
  - The output above shows two GBRT ensembles trained with a low learning rate
    - the one 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

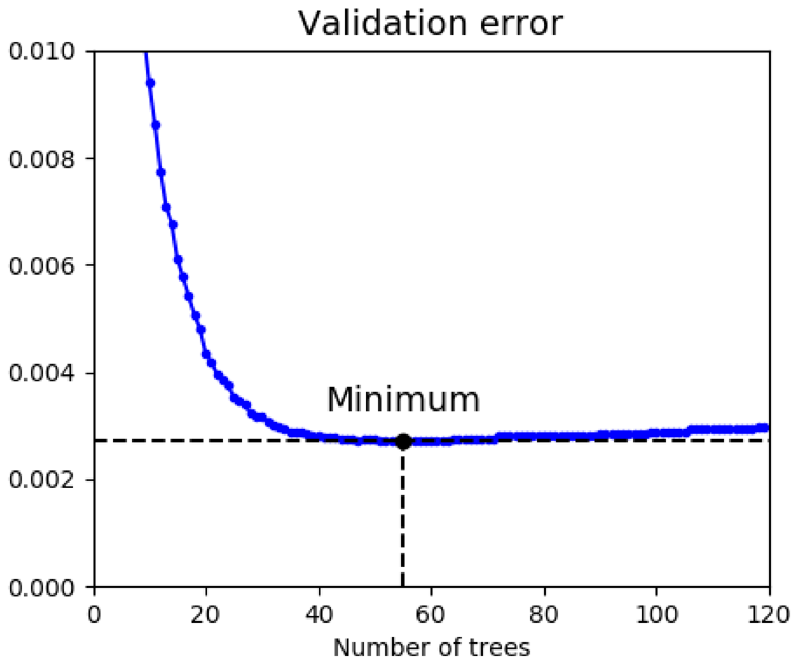
Gradient Boosting with Early Stopping

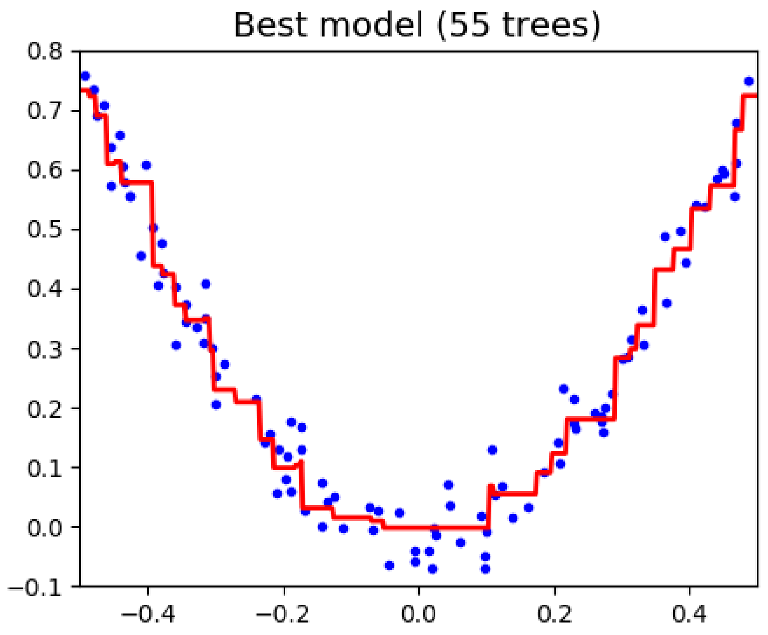
- In order to find the optimal number of trees, you can use early stopping
  - A simple way to implement this is to use the `staged_predict()` method
    - It returns an iterator over the predictions made by the ensemble at each stage of training (with one tree, with two trees, etc.)
- The following code
  - trains a GBRT ensemble with 120 trees
  - measures the validation error at each stage of training to find the optimal number of trees
  - trains another GBRT ensemble using the optimal tree number

Gradient Boosting: A code Example

```python
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error
from sklearn.ensemble import GradientBoostingRegressor
import matplotlib.pyplot as plt
np.random.seed(42)
X = np.random.rand(100, 1) - 0.5
y = 3*X[:, 0]**2 + 0.05 * np.random.randn(100)
X_train, X_val, y_train, y_val = train_test_split(X, y, random_state=49)
#Define GradientBoostingregressor classifier
gbrt = GradientBoostingRegressor(max_depth=2, n_estimators=120, random_state=42)
gbrt.fit(X_train, y_train)
#Compute the validation error
errors = [mean_squared_error(y_val, y_pred)
          for y_pred in gbrt.staged_predict(X_val)]
bst_n_estimators = np.argmin(errors)
#Define GradientBoostingregressor classifier
gbrt_best = GradientBoostingRegressor(max_depth=2,n_estimators=bst_n_estimators, random_state=42)
gbrt_best.fit(X_train, y_train)
#Compute the min validation error
min_error = np.min(errors)
def plot_predictions(regressors, X, y, axes, label=None, style="r-", data_style="b.", data_label=None):
    x1 = np.linspace(axes[0], axes[1], 500)
    y_pred = sum(regressor.predict(x1.reshape(-1, 1)) for regressor in regressors)
    plt.plot(X[:, 0], y, data_style, label=data_label)
    plt.plot(x1, y_pred, style, linewidth=2, label=label)
    if label or data_label:
        plt.legend(loc="upper center", fontsize=16)
    plt.axis(axes)
#Plotting Figure
plt.figure(figsize=(11, 4))
plt.subplot(121)
plt.plot(errors, "b.-")
plt.plot([bst_n_estimators, bst_n_estimators], [0, min_error], "k--")
plt.plot([0, 120], [min_error, min_error], "k--")
plt.plot(bst_n_estimators, min_error, "ko")
plt.text(bst_n_estimators, min_error*1.2, "Minimum", ha="center", fontsize=14)
plt.axis([0, 120, 0, 0.01])
plt.xlabel("Number of trees")
plt.title("Validation error", fontsize=14)
plt.subplot(122)
plot_predictions([gbrt_best], X, y, axes=[-0.5, 0.5, -0.1, 0.8])
plt.title("Best model (%d trees)" % bst_n_estimators, fontsize=14)
plt.show()
```

Gradient Boosting: Example Results





- The validation errors are represented on the left
- The best model’s predictions are represented on the right

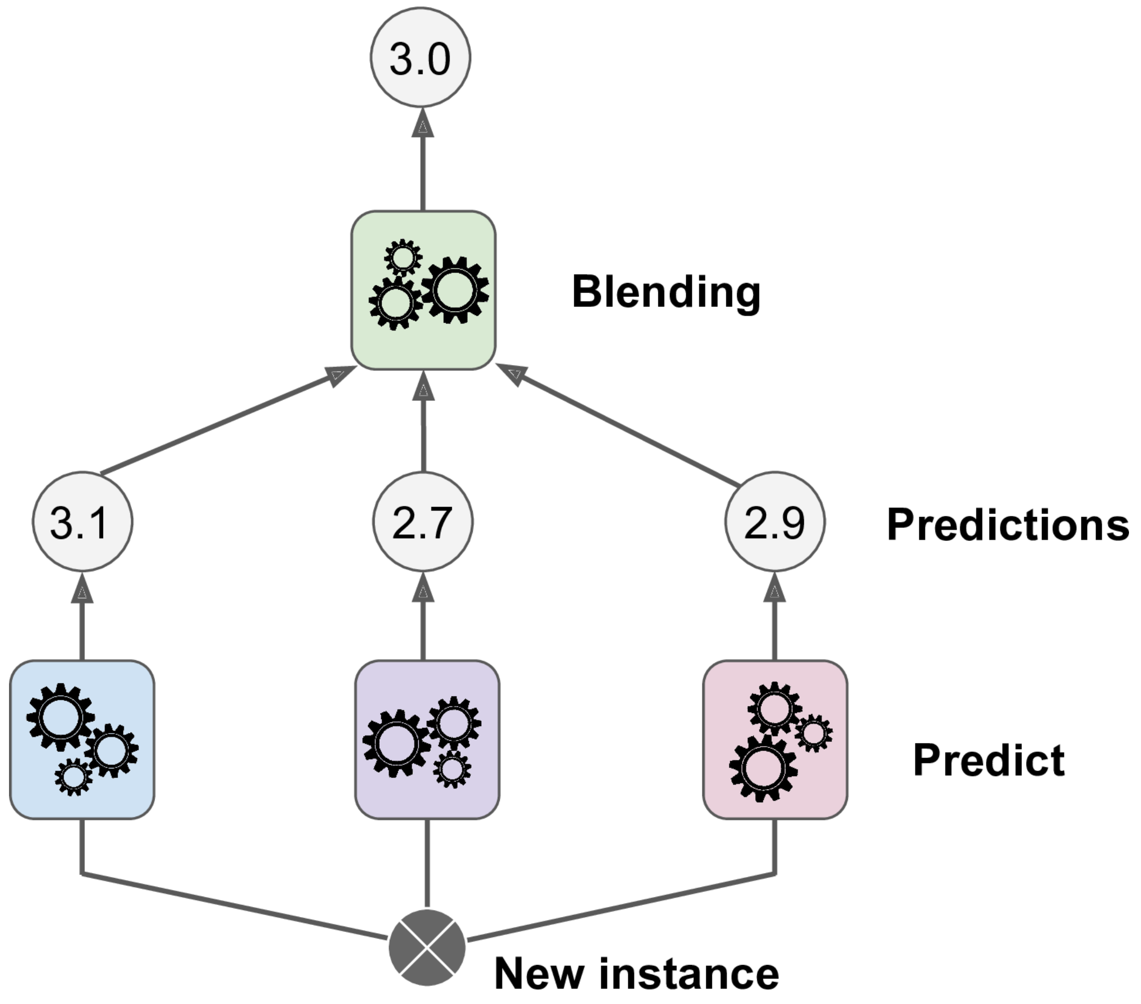
Stochastic Gradient Boosting

- The GradientBoostingRegressor class also supports a subsample hyperparameter
  - It specifies the fraction of training instances to be used for training each tree
- For example, if subsample=0.25, then each tree is trained on 25% of the training instances, selected randomly
- This trades a higher bias for a lower variance
- It also speeds up training considerably
- This technique is called Stochastic Gradient Boosting

Stacking (1)

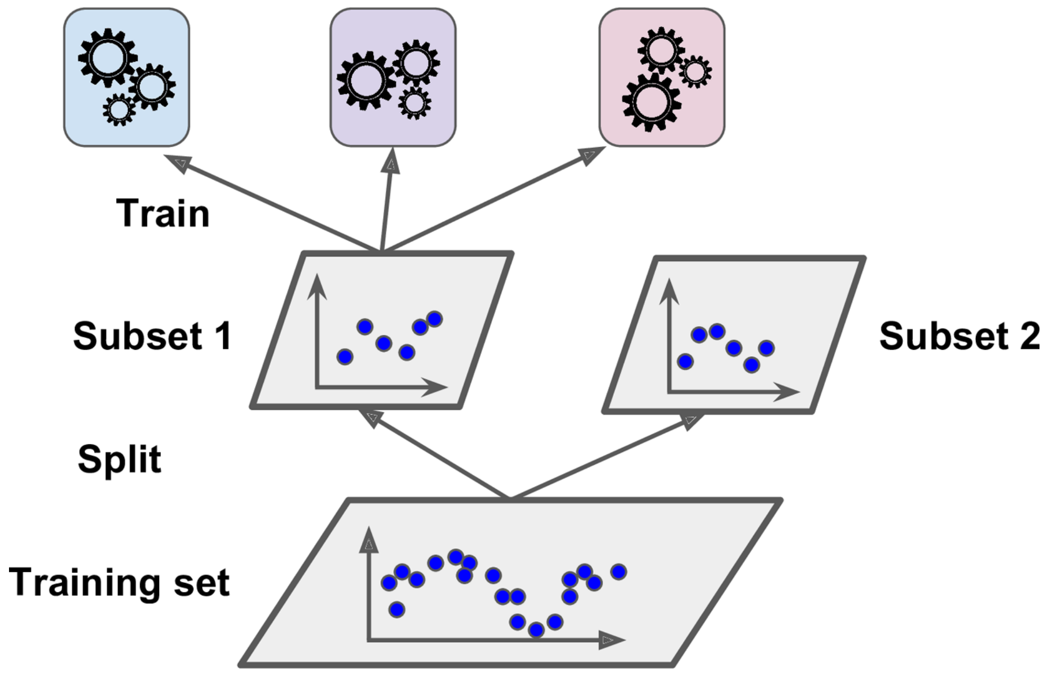
- Another Ensemble method is called stacking (short for *stacked generalization*)
- It is based on a simple idea
  - Instead of using trivial functions (such as hard voting) to aggregate the predictions of all predictors in an ensemble, we train a model to perform this aggregation?
- The Figure in the next slide shows such an ensemble performing a regression task on a new instance
  - Each of the bottom three predictors predicts a different value (3.1, 2.7, and 2.9), and
  - the final predictor (called a blender, or a meta learner) takes these predictions as inputs and makes the final prediction (3.0)

Stacking (2)



Stacking (3)

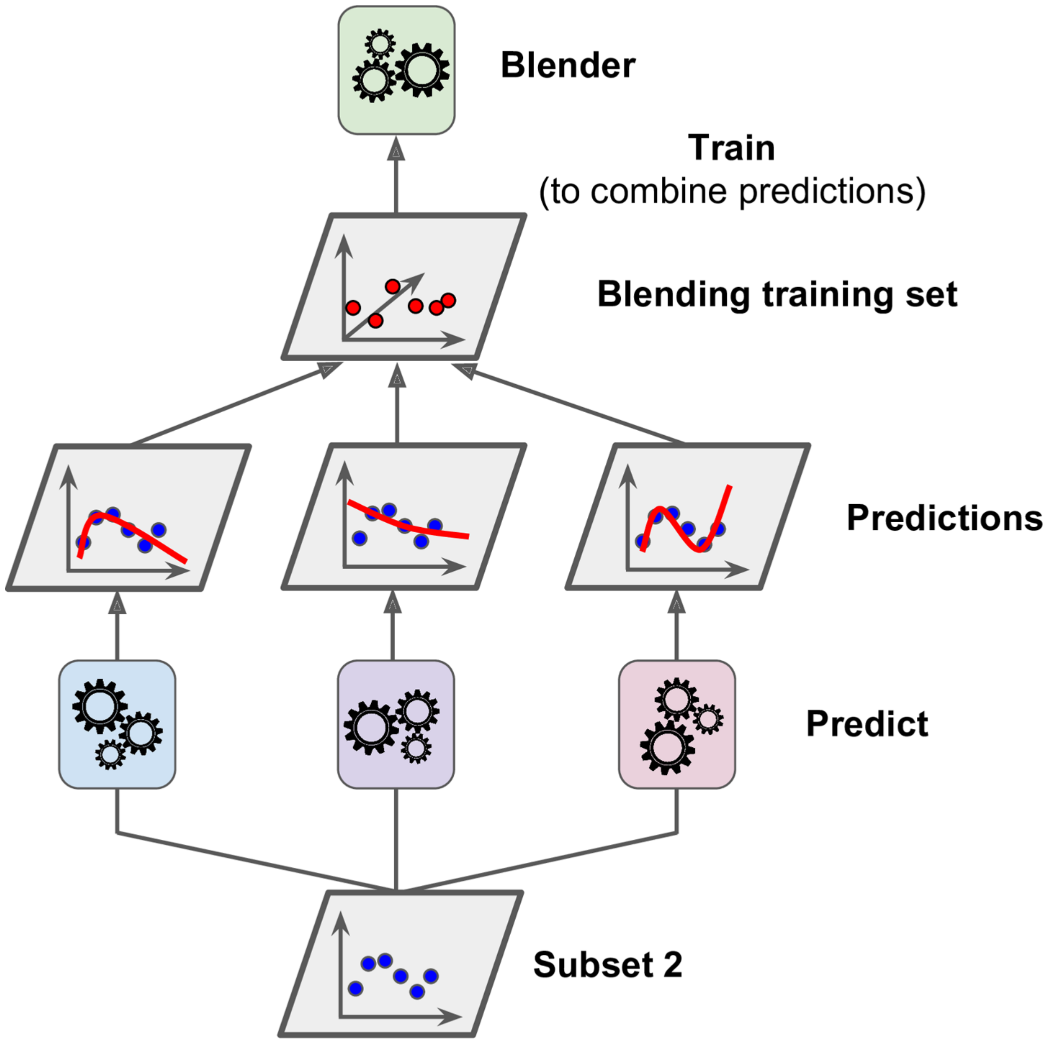
- To train the blender, a common approach is to use a hold-out set
  - First, the training set is split in two subsets
  - The first subset is used to train the predictors in the first layer



Stacking (4)

- Next, the first layer predictors are used to make predictions on the second (hold-out) set (see next slide)
  - This ensures that the predictions are “clean,” since the predictors never saw these instances during training
  - Now for each instance in the hold-out set 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 three-dimensional), and keeping the target values
- The blender is trained on this new set, so it learns to predict the target value given the first layer’s predictions

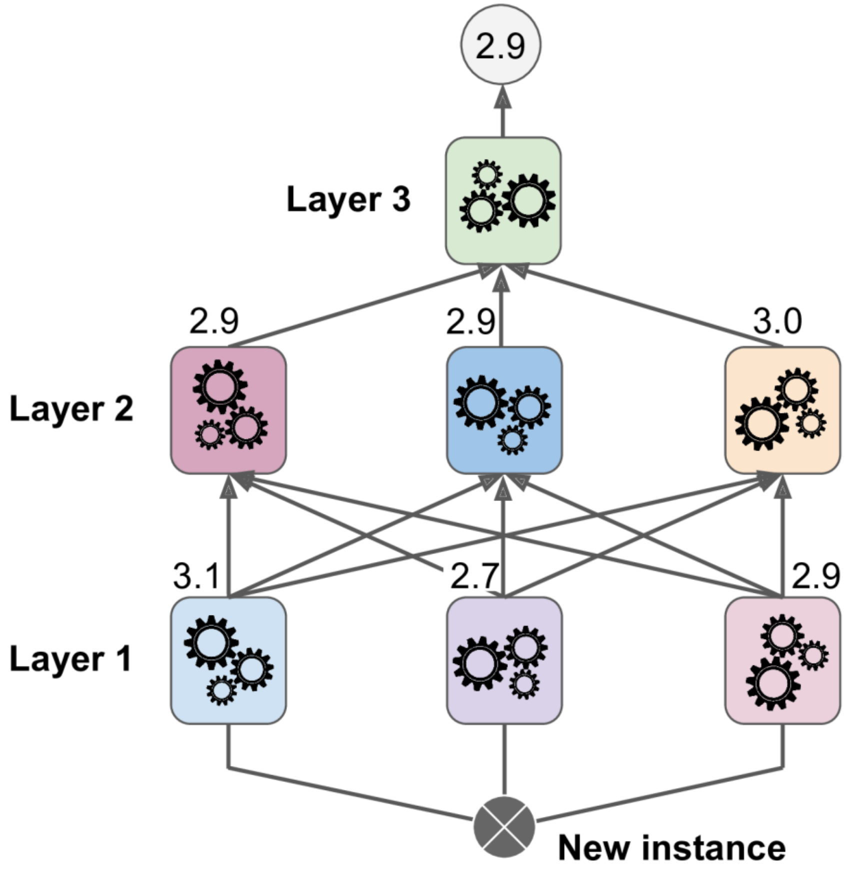
Stacking (5)


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

Stacking (6)

- It is possible to train several different blenders this way (e.g., one with Linear Regression, another with Random Forest Regression, and so on): we get a whole layer of blenders
- The trick is to split the training set into 3 subsets: the first one is used to train the first layer, the second one to create the training set used to train the second layer (using predictions made by the predictors of the first layer), and the third one to create the training set to train the third layer (using predictions made by the predictors of the second layer)
- Successively, we can make a prediction for a new instance by going through each layer sequentially, as shown in next figure

Stacking



Predictions in a multilayer stacking ensemble