4.
Boosting

Boosting

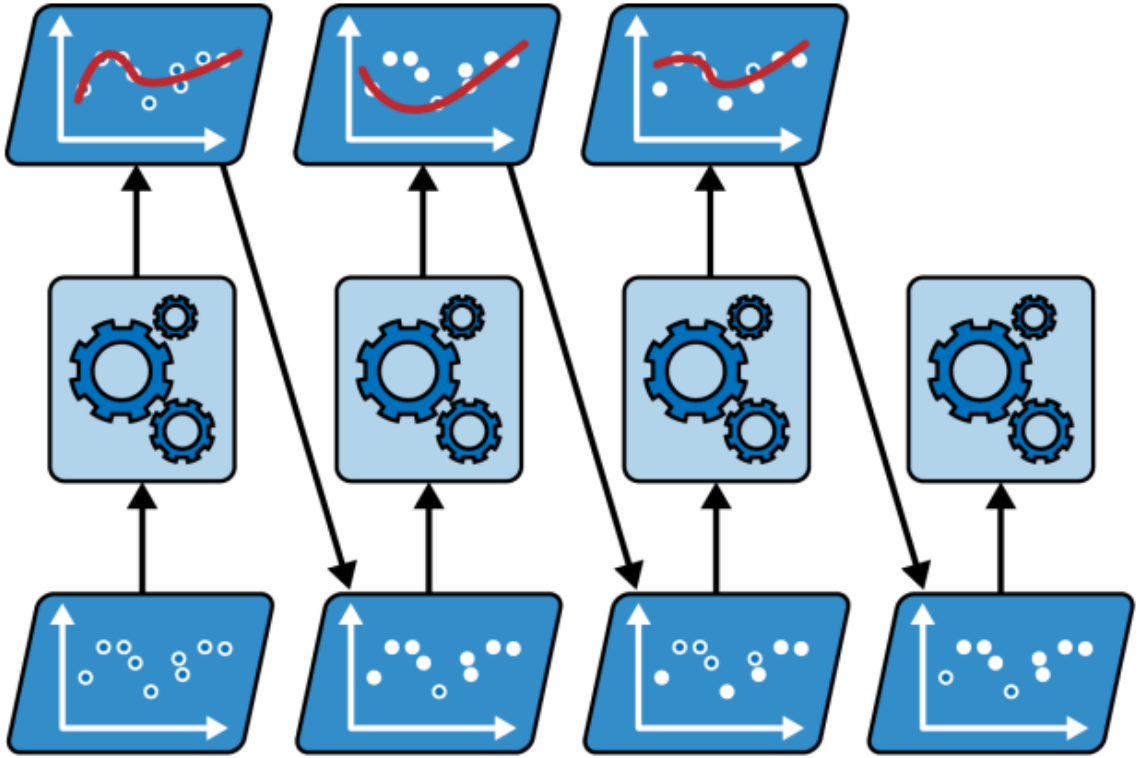
- *Boosting* refers to any ensemble method that can combine several weak learners into a strong learner.
- The general idea of most boosting methods is to train predictors sequentially, each trying to correct its predecessor.
- The most popular methods are *AdaBoost* (short for *adaptive boosting*) and *gradient boosting*.

AdaBoost

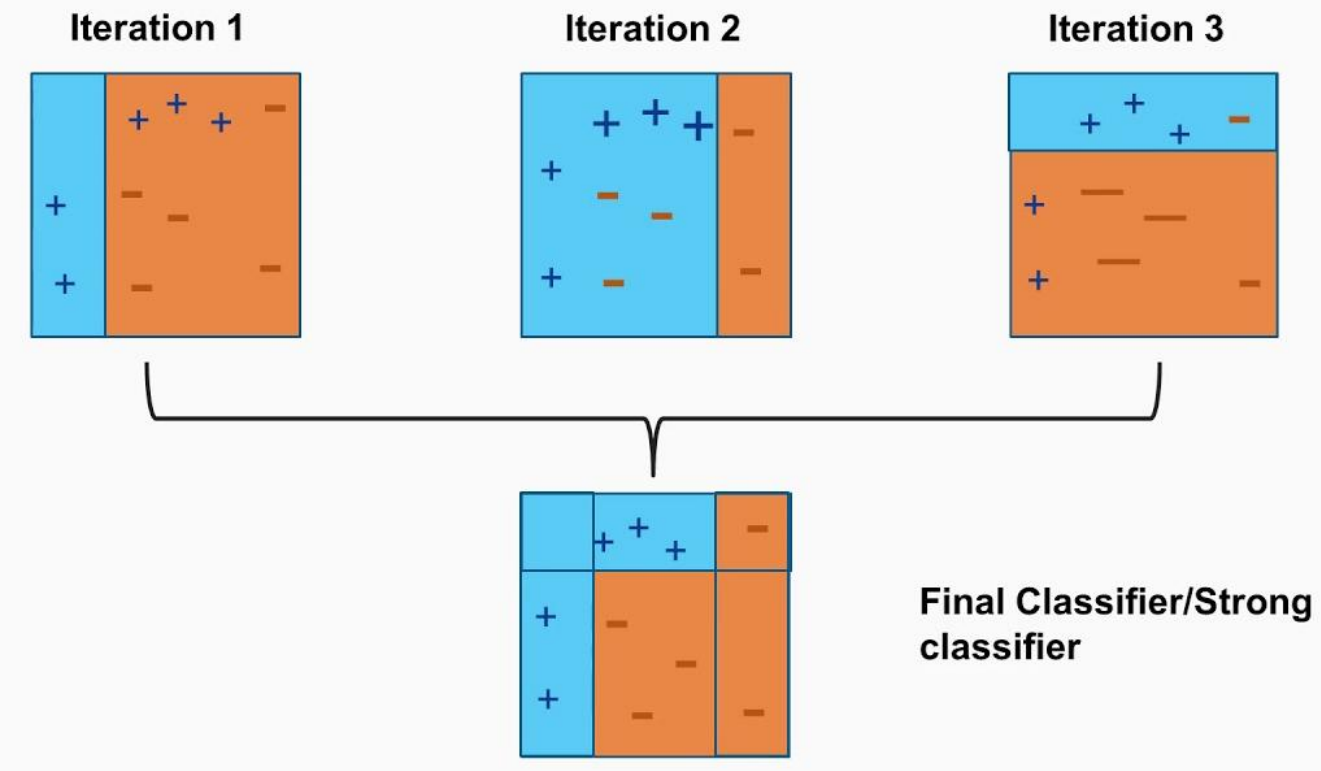
- *Adaboost*: A new predictor corrects its predecessor by paying attention to the training instances that the predecessor underfit.
  - New predictors focus more and more on the hard cases

- AdaBoost classifier:
  1. Trains a base classifier (such as a decision tree) and uses it to make predictions on the training set.
  2. Increases the relative weight of misclassified training instances.
  3. Trains a second classifier, using the updated weights, and again makes predictions on the training set, updates the instance weights, and so on.

AdaBoost Sequential Training

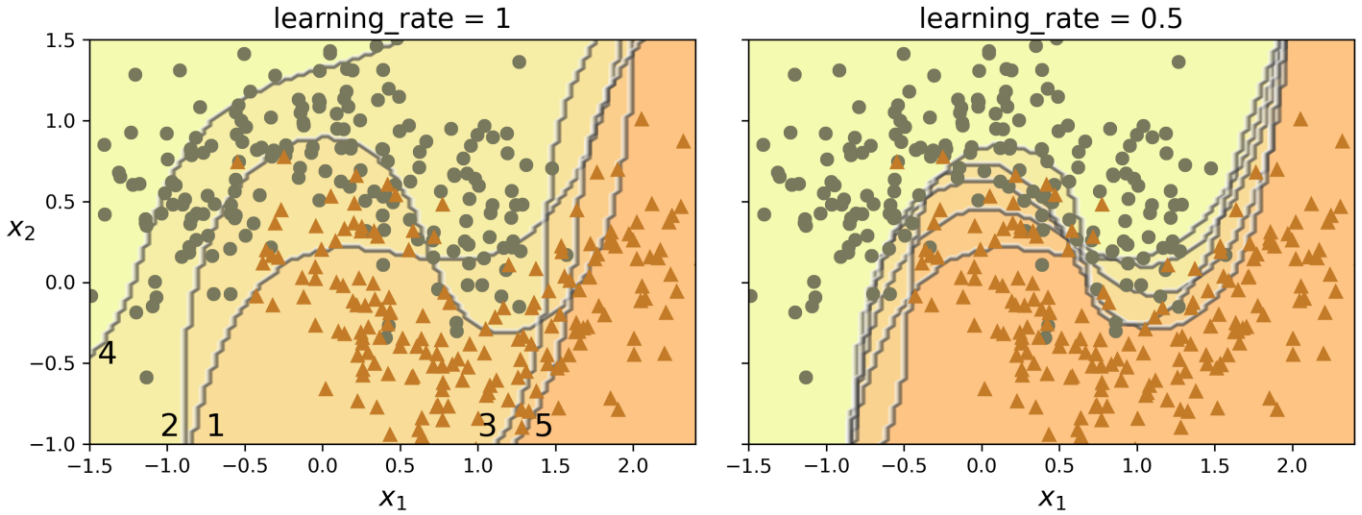


AdaBoost Classifier



Decision Boundaries of Consecutive Predictors

- The plot on the right represents the same sequence of predictors, except that the learning rate is halved (i.e., the misclassified instance weights are boosted much less at every iteration).



AdaBoost in Scikit-Learn

- Scikit-Learn uses a multiclass version of AdaBoost called *SAMME* (*Stagewise Additive Modeling using a Multiclass Exponential loss function*).
- Training an AdaBoost classifier based on 30 *decision stumps*:

```
from sklearn.ensemble import AdaBoostClassifier

ada_clf = AdaBoostClassifier(
    DecisionTreeClassifier(max_depth=1), n_estimators=30,
    learning_rate=0.5, random_state=42)
ada_clf.fit(X_train, y_train)
```

- A decision stump is a decision tree composed of a single decision node plus two leaf nodes.
  - If your AdaBoost ensemble is overfitting the training set, reduce the number of estimators or more strongly regularize the base estimator.

Gradient Boosting

- Similar to AdaBoost, but instead of tweaking the instance weights at every iteration, this method tries to fit the new predictor to the *residual errors* made by the previous predictor.
- Example of *gradient tree boosting*, or *gradient boosted regression trees* (GBRT):

1) generate a noisy quadratic dataset and fit a `DecisionTreeRegressor`:

```
import numpy as np
from sklearn.tree import DecisionTreeRegressor

np.random.seed(42)
X = np.random.rand(100, 1) - 0.5
y = 3 * X[:, 0] ** 2 + 0.05 * np.random.randn(100)  # y = 3x² + Gaussian noise

tree_reg1 = DecisionTreeRegressor(max_depth=2, random_state=42)
tree_reg1.fit(X, y)
```

GBRT

2) 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, random_state=43)
tree_reg2.fit(X, y2)
```

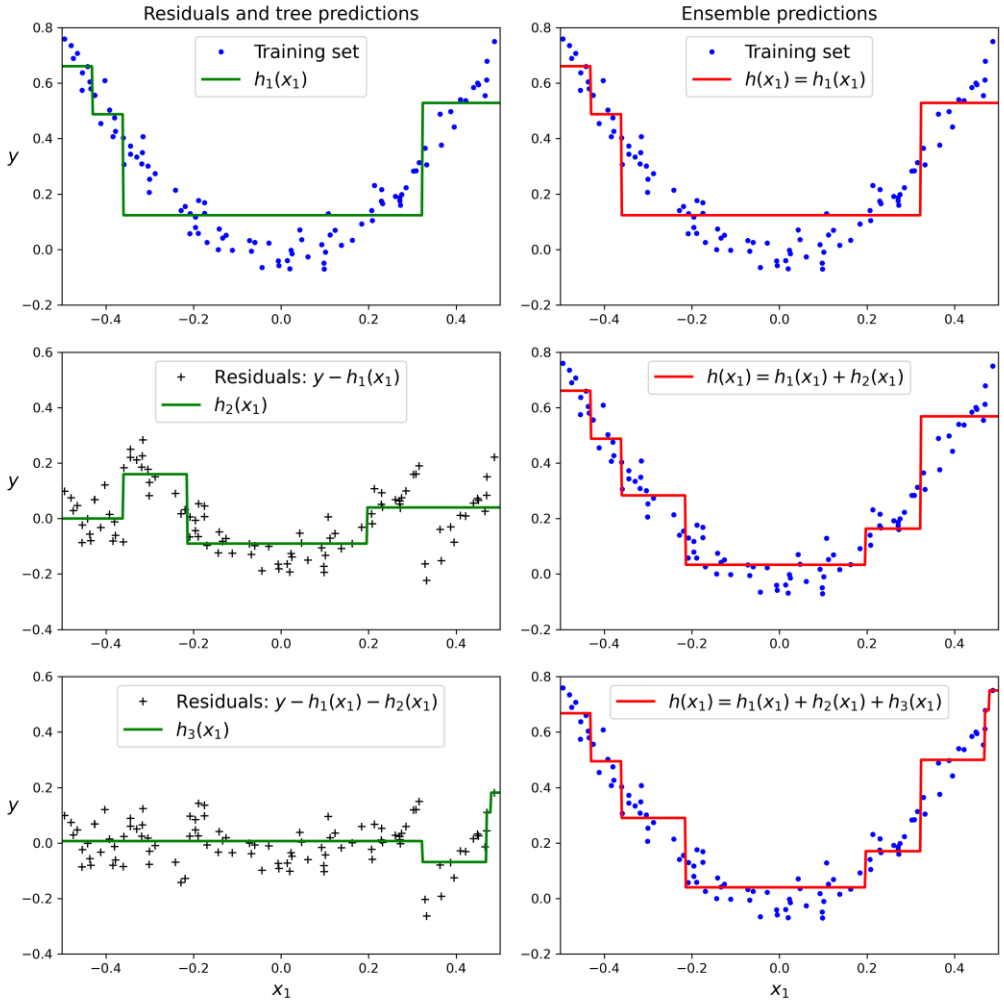
3) Train a third regressor on the residual errors made by the second predictor:

```
y3 = y2 - tree_reg2.predict(X)
tree_reg3 = DecisionTreeRegressor(max_depth=2, random_state=44)
tree_reg3.fit(X, y3)
```

4) Now we have an ensemble containing three trees and make predictions:

```
X_new = np.array([[-0.4], [0.], [0.5]])
sum(tree.predict(X_new) for tree in (tree_reg1, tree_reg2, tree_reg3))
```

```
array([0.49484029, 0.04021166, 0.75026781])
```


Residuals and tree predictions
Ensemble predictions
Training set
$h_1(x_1)$
$h(x_1) = h_1(x_1)$
Residuals: $y - h_1(x_1)$
$h_2(x_1)$
$h(x_1) = h_1(x_1) + h_2(x_1)$
Residuals: $y - h_1(x_1) - h_2(x_1)$
$h_3(x_1)$
$h(x_1) = h_1(x_1) + h_2(x_1) + h_3(x_1)$
y
x_1

Gradient Boosting in Scikit-Learn

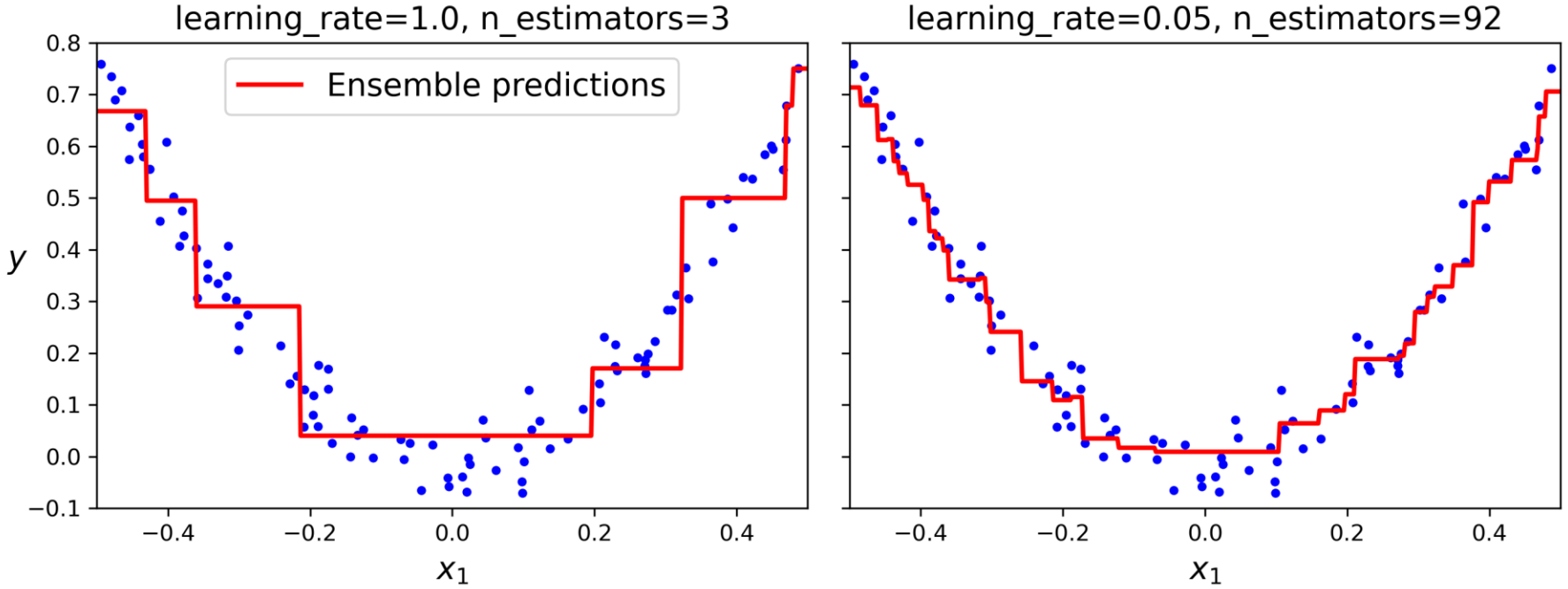
- Use Scikit-Learn's `GradientBoostingRegressor` class to train GBRT ensemble:

```
from sklearn.ensemble import GradientBoostingRegressor

gbrt = GradientBoostingRegressor(max_depth=2, n_estimators=3,
                                 learning_rate=1.0, random_state=42)
gbrt.fit(X, y)
```

- The `learning_rate` hyperparameter scales the contribution of each tree.
  - If you set it to a low value, you will need more trees in the ensemble to fit the training set, but the predictions will generalize better.
  - This is a regularization technique called *shrinkage*.

Gradient Boosting in Scikit-Learn



GBRT with Early Stopping

- To find the optimal number of trees, you could perform cross-validation using `GridSearchCV` or `RandomizedSearchCV`.
- If you set the `n_iter_no_change` hyperparameter to an integer N, then the `GradientBoostingRegressor` will automatically stop adding more trees during training if it sees that the last N trees didn't help.
  - This is early stopping.

```
gbrt_best = GradientBoostingRegressor(
    max_depth=2, learning_rate=0.05, n_estimators=500,
    n_iter_no_change=10, random_state=42)
gbrt_best.fit(X, y)
```

```
gbrt_best.n_estimators_
```

```
92
```

GBRT with Early Stopping

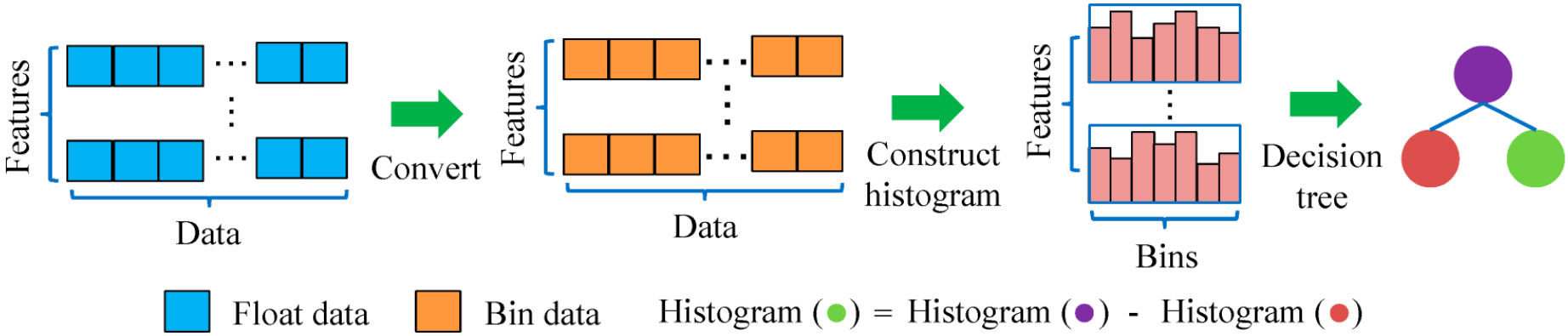
- When `n_iter_no_change` is set, the `fit`() method automatically splits the training set into a smaller training set and a validation set.
  - Allows it to evaluate the model's performance each time it adds a new tree.
  - Size of the validation set is controlled by the `validation_fraction` hyperparameter which is 10% by default.
  - The `tol` hyperparameter (defaults to 0.0001) determines the maximum performance improvement that still counts as negligible.
- The `GradientBoostingRegressor` class also supports a `subsample` hyperparameter, which specifies the fraction of training instances to be used for training each tree.
  - This is called *stochastic gradient boosting*.
  - It speeds up training considerably and lowers the variance.

Histogram-Based Gradient Boosting

- Scikit-Learn provides another GBRT implementation, optimized for large datasets: *histogram-based gradient boosting* (HGB).
- It works by binning the values of input features.
  - The number of bins is controlled by the `max_bins`, which defaults to 255 and cannot be set any higher than this.
  - The construction of decision trees can be sped up significantly by reducing the number of values for continuous input features.
- Working with the ordinal bucket (an integer) makes it possible to use faster and more memory-efficient data structures.

Histogram-Based Gradient Boosting

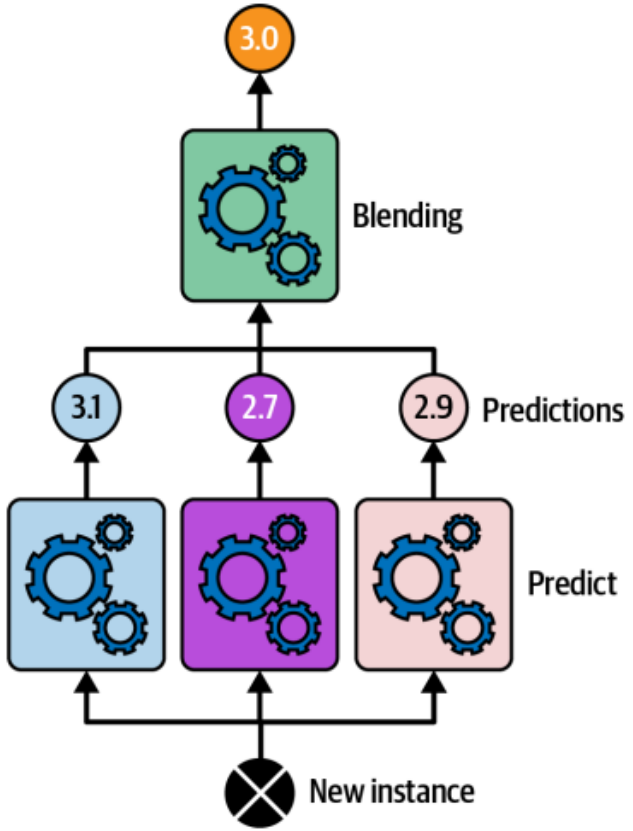
- HGB can train hundreds of times faster than regular GBRT on large datasets.
- But binning causes a precision loss, which acts as a regularizer.



5.
Stacking

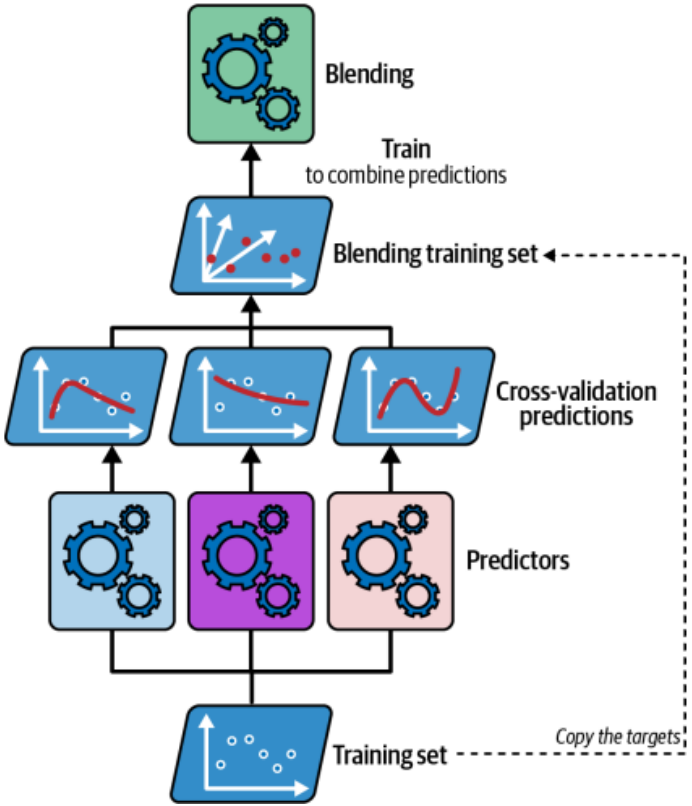
Stacking

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



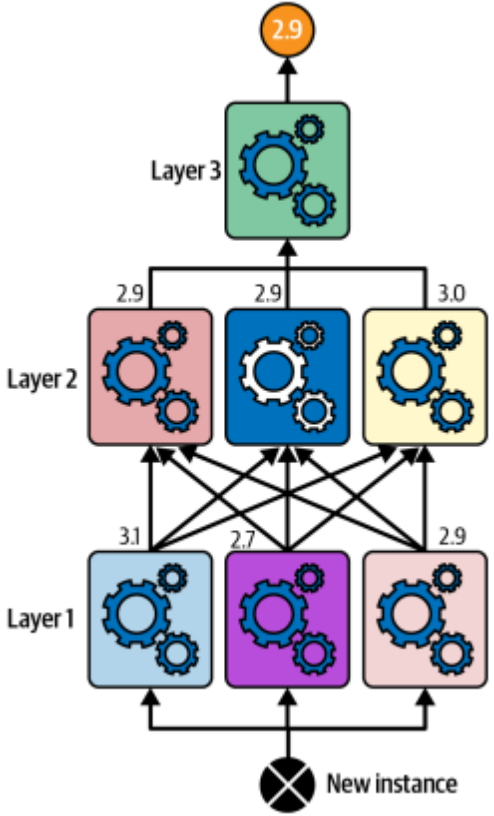
Training the Blender

- Build the blending training set: use cross validation on predictors in the ensemble to get out-of-sample predictions for each instance.
- Use these predictions as the input features to train the blender.
- The targets can simply be copied from the original training set.
- Once the blender is trained, the base predictors are retrained one last time on the full original training set.



Multilayer Stacking Ensemble

- It is possible to train several different blenders this way to get a whole layer of blenders, and then add another blender on top of that to produce the final prediction.



Stacking in Scikit-Learn

- Scikit-Learn provides two classes for stacking ensembles: `StackingClassifier` and `StackingRegressor`.

```
from sklearn.ensemble import StackingClassifier

stacking_clf = StackingClassifier(
    estimators=[
        ('lr', LogisticRegression(random_state=42)),
        ('rf', RandomForestClassifier(random_state=42)),
        ('svc', SVC(probability=True, random_state=42))
    ],
    final_estimator=RandomForestClassifier(random_state=43),
    cv=5  # number of cross-validation folds
)
stacking_clf.fit(X_train, y_train)
```

- If you don't provide a final estimator, `StackingClassifier` will use `LogisticRegression` and `StackingRegressor` will use `RidgeCV`.

6.
Final Thoughts

Final Thoughts

- 20 out of 22 winning solutions on Kaggle competitions in 2021, as of August 2021, use ensembles.
- As of September 2023, top 10 solutions on SQuAD 2.0, the Stanford Question Answering Dataset, are all ensembles.
- While ensembles can give your ML system a small performance improvement, ensembling tends to make a system too complex to be useful in production, e.g., slower to make predictions or harder to interpret the results.

Final Thoughts

- Ensembling methods are less favored in production because ensembles are more complex to deploy and harder to maintain.
- They are still common for tasks *where a small performance boost can lead to a huge financial gain*, such as predicting click-through rate for ads.
- Ensemble methods such as boosting and bagging, together with resampling, have shown to help with *imbalanced datasets*.