Ensemble Learning and Random Forests

Ensemble Learning

Grouping multiple predictors aka models is called ensemble learning.

A group of predictors is called an **ensemble**; thus, this technique is called **Ensemble Learning**, and an **Ensemble Learning algorithm** is called an **Ensemble method.**

The winning solutions in Machine Learning competitions often involve several Ensemble methods.

What we'll learn in this session ?

- What are Ensemble Methods
  - Voting Classifier
- Bagging and Pasting
  - Bagging or Bootstrap Aggregating
  - Pasting
  - Out of Bag Evaluation
- Random Patches and Random Subspaces

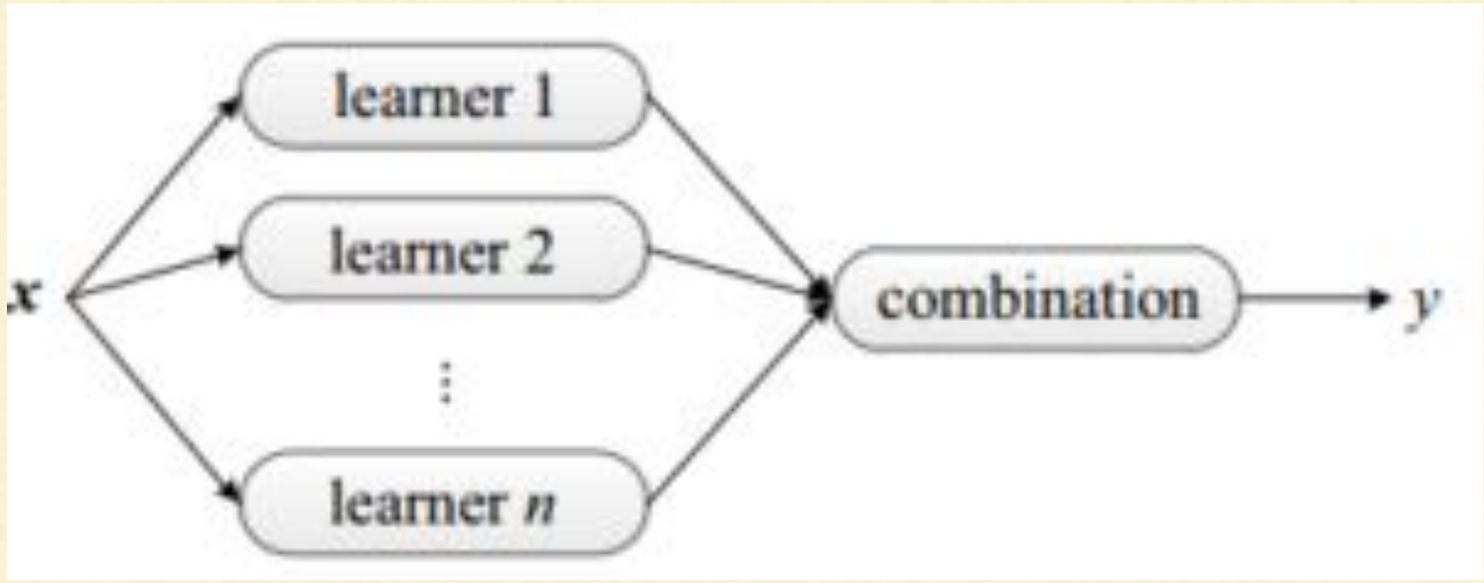
What we'll learn in this session ?

- Random Forests
  - Extra-Trees
  - Feature Importance
- Boosting
  - AdaBoost
  - Gradient Boosting
- Stacking
- XGBoost

Ensemble Learning

Suppose you 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**.

What is Ensemble Learning?



Similarly, if you aggregate the predictions of a group of predictors (such as classifiers or regressors), you will often get better predictions than with the best individual predictor.

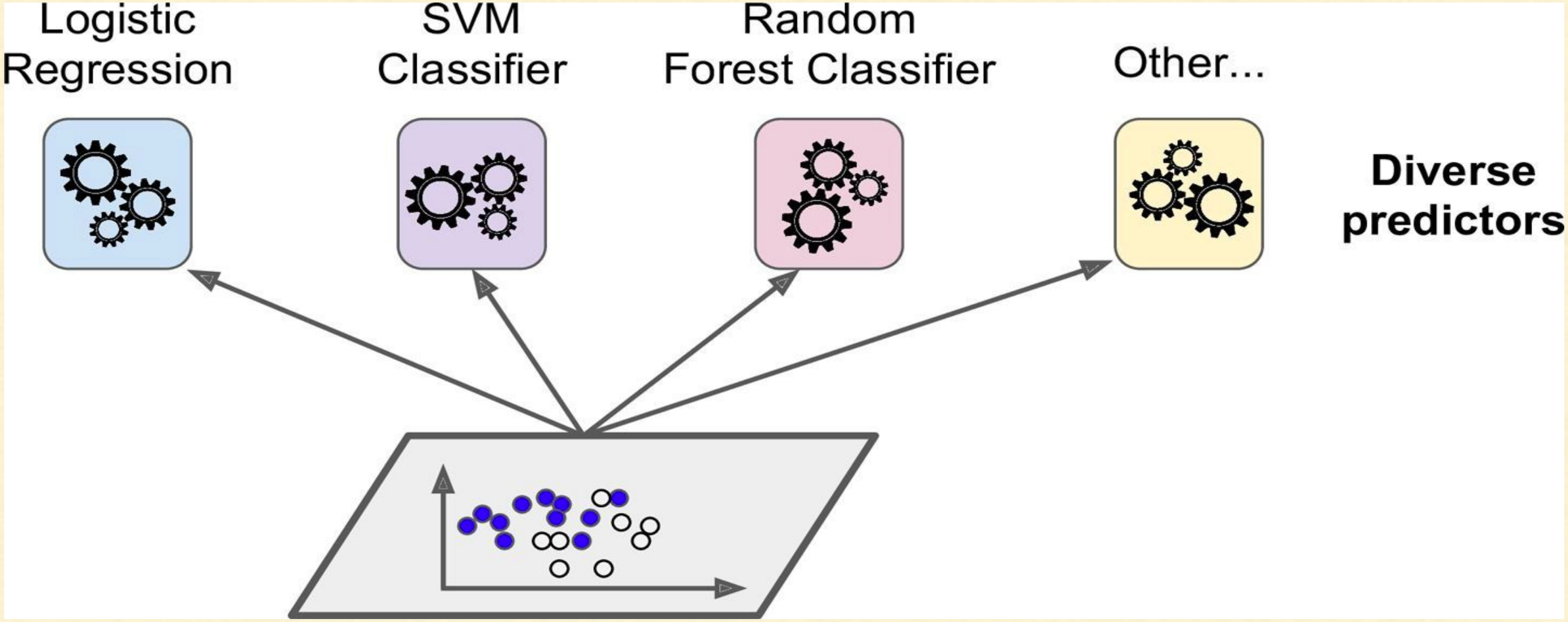
Voting Classifiers

Suppose you have trained a few classifiers, each one achieving about **80% accuracy**.

You may have a

- **Logistic Regression classifier**,
- a **SVM classifier**,
- a **Random Forest classifier**,
- a **K-Nearest Neighbors classifier**,
- and perhaps a few more.

Voting Classifiers



Suppose each of the classifier gives a accuracy of 80% on the given data

Voting Classifiers

Is there a way to achieve higher accuracy using the given models

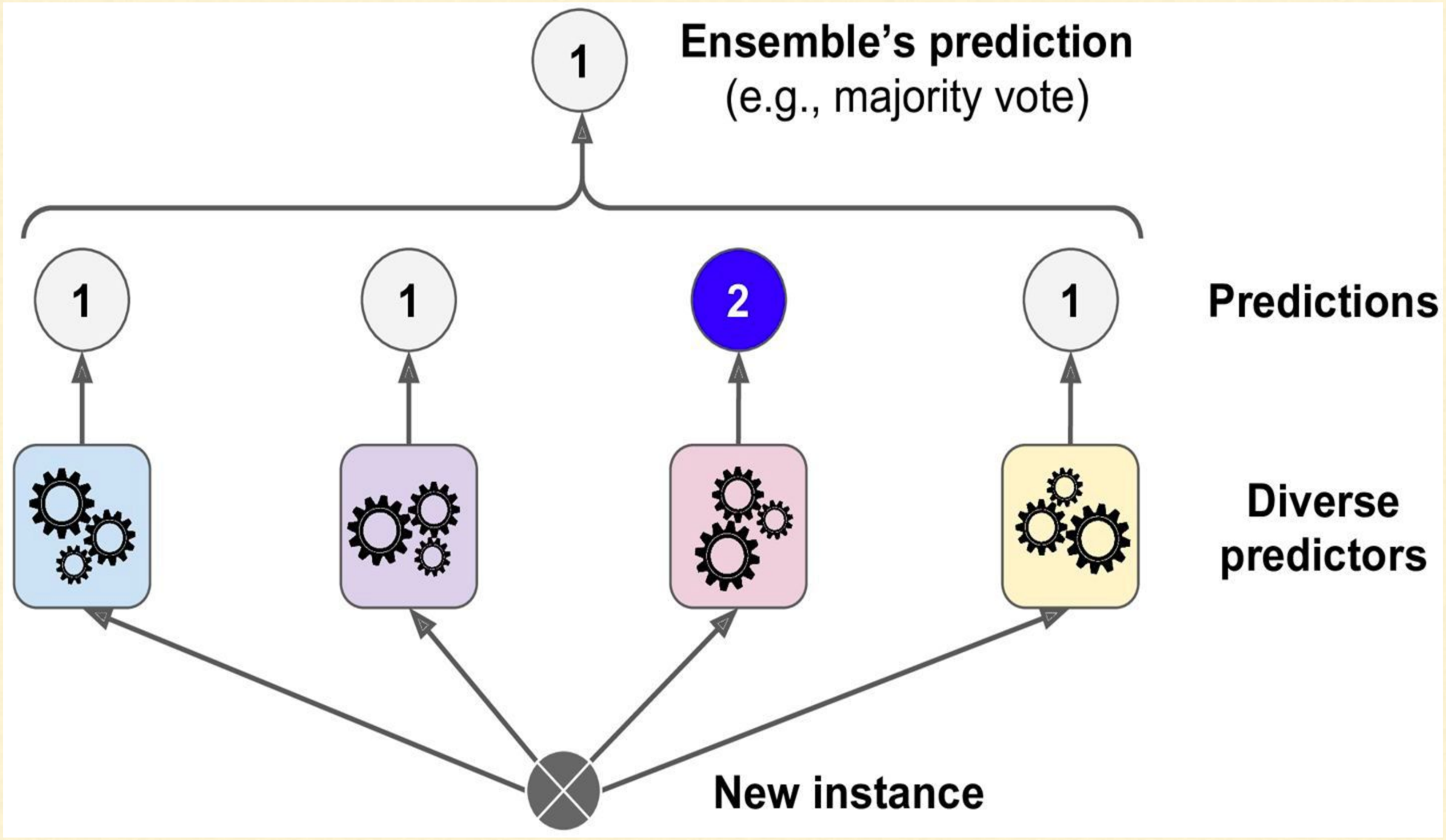
???

Voting Classifiers

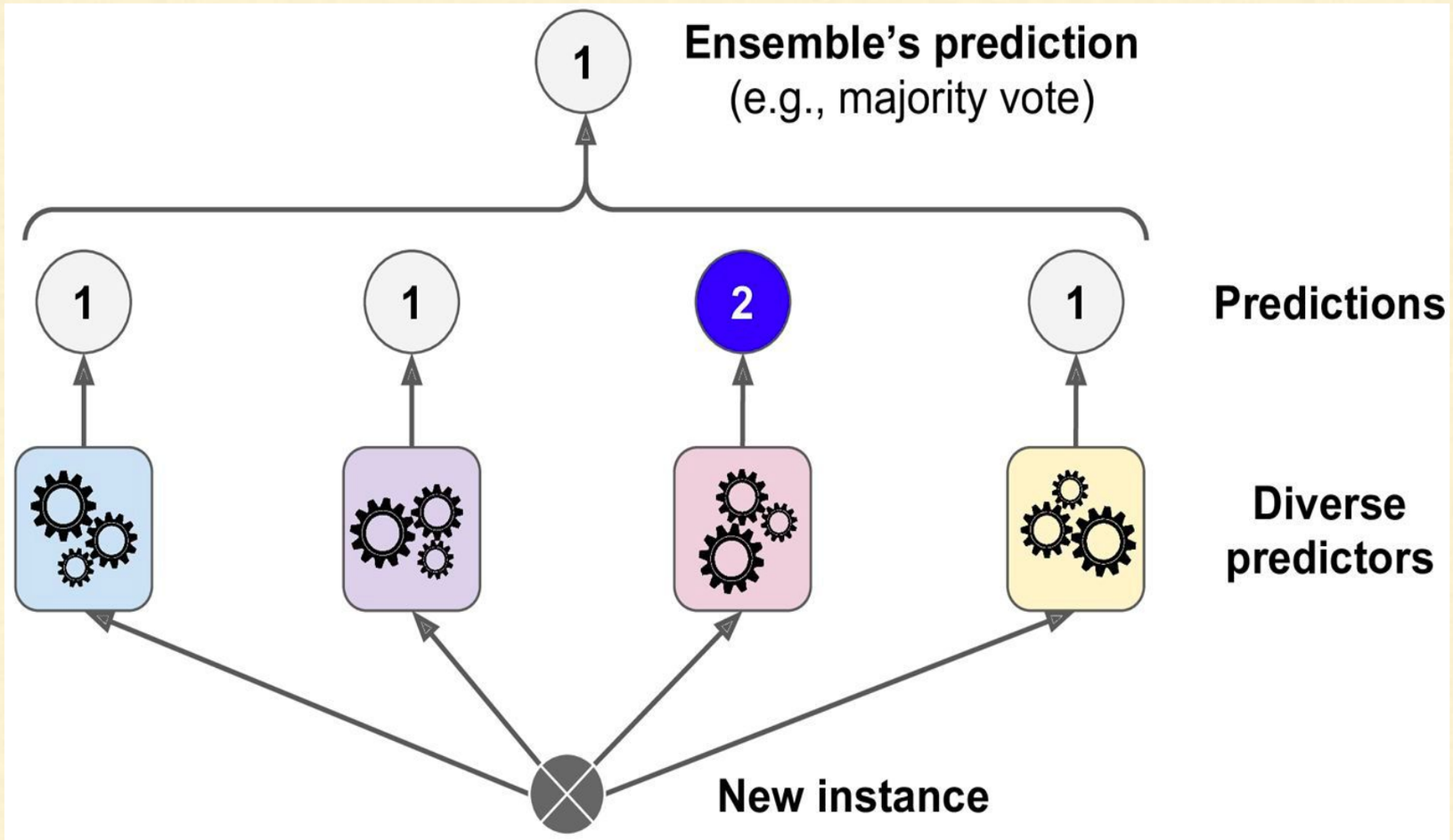
The answer is **YES !!**

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.**

Voting Classifiers



Voting Classifiers



Here the prediction of the ensemble will be decided by the votes from all the classifiers.
The class that gets the highest vote is the final output of the Ensemble.

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

Voting Classifiers

Somewhat 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** (meaning it does only slightly better than random guessing), the ensemble can still be a **strong learner** (achieving high accuracy), provided there are a sufficient number of weak learners and they are sufficiently diverse.

Voting Classifiers

How is it possible that the ensemble performs better than the individual classifiers ???

Voting Classifiers

Consider the following analogy

Suppose you have a slightly biased coin that has a **51%** chance of coming up heads, and **49%** chance of coming up tails.

Voting Classifiers

If you toss it **1,000 times**, you will generally get more or less **510 heads** and **490 tails**, and hence a majority of heads.

Voting Classifiers

Let's look into the probability distribution of this biased coin

No.of Tosses	No.of Heads	No.of tails	Probability
1	1 0	0 1	0.51 0.49
2	2 0 1	0 2 1	0.51 x 0.51 0.49 x 0.49 2 x 0.49 x 0.51

Voting Classifiers

No.of Tosses	No.of Heads	No.of tails	Probability
3	3	0	0.51 x 0.51 x 0.51
	0	3	0.49 x 0.49 x 0.49
	1	2	$3 \times 0.51 \times (0.49)^2$
	2	1	$3 \times (0.51)^2 \times 0.49$

Here permutation of coins are also considered

Voting Classifiers

From this observation we find that the probabilities of the coin tosses follows the binomial expansion pattern.

So, if we toss a coin n number of times the probabilities will be terms of the binomial expansion

$$^{n}C_{r}\, a^{n-r}\, b^{r}$$

Here **a is probability of heads** and **b is the probability of tails.**

Voting Classifiers

Now let us find the probability that after tossing a coin n times what will be the probability that heads appeared in majority ?

For head to be in majority the power of a (i.e probability of heads) should be more than power of b (i.e probability of tails)

=> n - r > r

=> n > 2r

Voting Classifiers

Coming back from our analogy to the question "**Why does ensemble method perform better than individual classifiers ?**"

Suppose you have 1000 classifiers each having an accuracy of only 51%. For an ensemble to output a particular class, that class must be the output of majority of classifiers.

Hence the accuracy of the ensemble will be decided by the probability, that the class is selected in majority by the 1000 classifiers

Voting Classifiers

The accuracy of the ensemble will be

$$^{n}C_{r}\ a^{n-r}\ b^{r}$$

For all n > 2r

Hence for an ensemble of 1000 classifiers the accuracy comes out to be

≈ 72.6 %

Hence for a combination of 1000 classifiers with only 51% accuracy, the combination has a accuracy of over 72.6% !!!

Run it on Notebook

Voting Classifiers

The law of large numbers

As you keep tossing the coin, the ratio of heads gets closer and closer to the probability of heads (51%).

This is due to the law of large numbers.

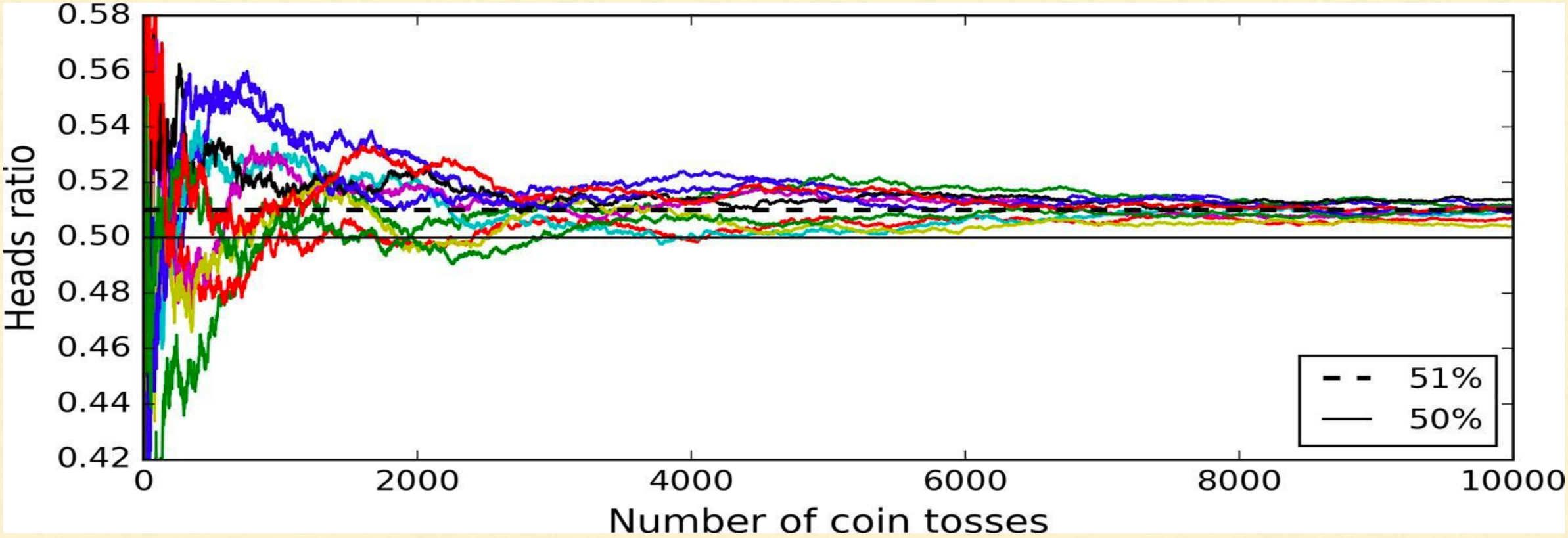
Voting Classifiers

The law of large numbers

Let's visualize, that if we perform 10 series of coin tosses for 10,000 iterations the probability of head reaches 51%.

Voting Classifiers

The law of large numbers



As the number of tosses increases, the ratio of heads approaches 51%.

Eventually all 10 series end up so close to 51% that they are consistently above 50% !! (See code)

Voting Classifiers

The law of large numbers

```
>>> heads_proba = 0.51
>>> coin_tosses = (np.random.rand(10000, 10) <
heads_proba).astype(np.int32)
>>> cumulative_sum_of_number_of_heads =
np.cumsum(coin_tosses, axis=0)
>>> cumulative_heads_ratio =
cumulative_sum_of_number_of_heads / np.arange(1,
10001).reshape(-1, 1)
```

Run it on Notebook

Voting Classifiers

Let's make our own voting classifier using Scikit learn

We will be testing our Voting classifier on the Moons dataset.

Moons dataset is a sample dataset which could be generated using Scikit learn.

Voting Classifiers

Let's make our own voting classifier using Scikit learn

The Moons dataset

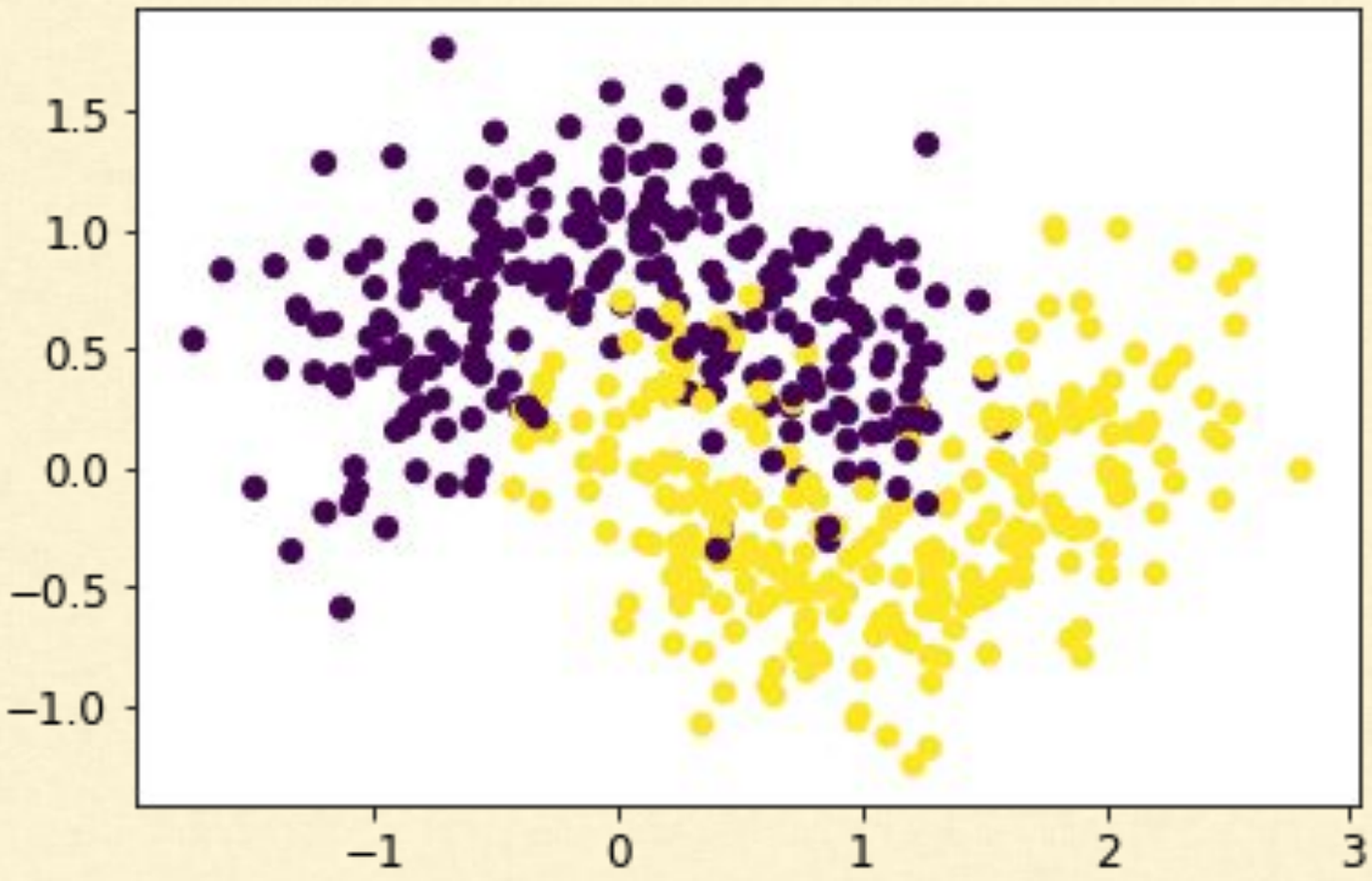
```
>>> from sklearn.datasets import make_moons
>>> X, y = make_moons(n_samples=500, noise=0.3,
random_state=42)
>>> X_train, X_test, y_train, y_test =
train_test_split(X, y, random_state=42)
plt.scatter(X[:,0], X[:, 1], c=y)
```

Run it on Notebook

Voting Classifiers

Let's make our own voting classifier using Scikit learn

The Moons dataset



Voting Classifiers

Let's make our own voting classifier using Scikit learn

```
>>> from sklearn.ensemble import RandomForestClassifier, 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')
```

Run it on Notebook

Hard and Soft Voting

Hard Voting

- When we consider only the final output from each of the classifier for our voting that it is called Hard voting.
- We have used the hard voting method by specifying **voting='hard'** when we were instantiating our VotingClassifier.

Hard and Soft Voting

Soft Voting

- 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.

Hard and Soft Voting

Soft Voting

Let's verify the fact that soft voting often achieves higher performance than hard voting.

We will find that the soft voting classifier achieves over 91% accuracy!

Run it on Notebook

Bagging and Pasting

Ensemble methods perform best when a diverse set of classifiers are used.

There are two ways in which you can achieve this objective :

- Use very **different training algorithms**.
- Or use the **same training algorithm** for every predictor, but train them on **different random subsets** of the training set.

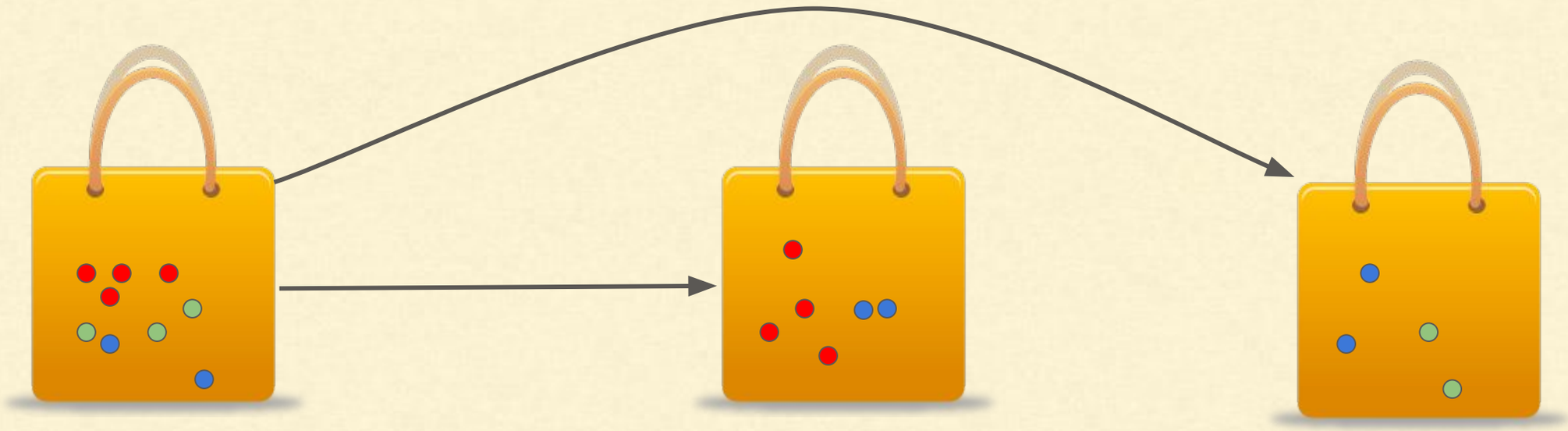
Bagging and Pasting

Bagging

When sampling is performed with replacement, this method is called **bagging** (short for bootstrap aggregating)

Bagging and Pasting

Bagging



Original Dataset Bag 1 Bag 2

Suppose the original dataset has **4 red, 3 green** and **2 blue balls**.
When we sample with replacement in Bag 1, it has **4 red** and **2 blue**
Again sampling with replacement in Bag 2, it has **2 blue** and **2 green**
Since we are sampling with replacement bag blue has 2 blue balls even though all the blue balls were in bag 1

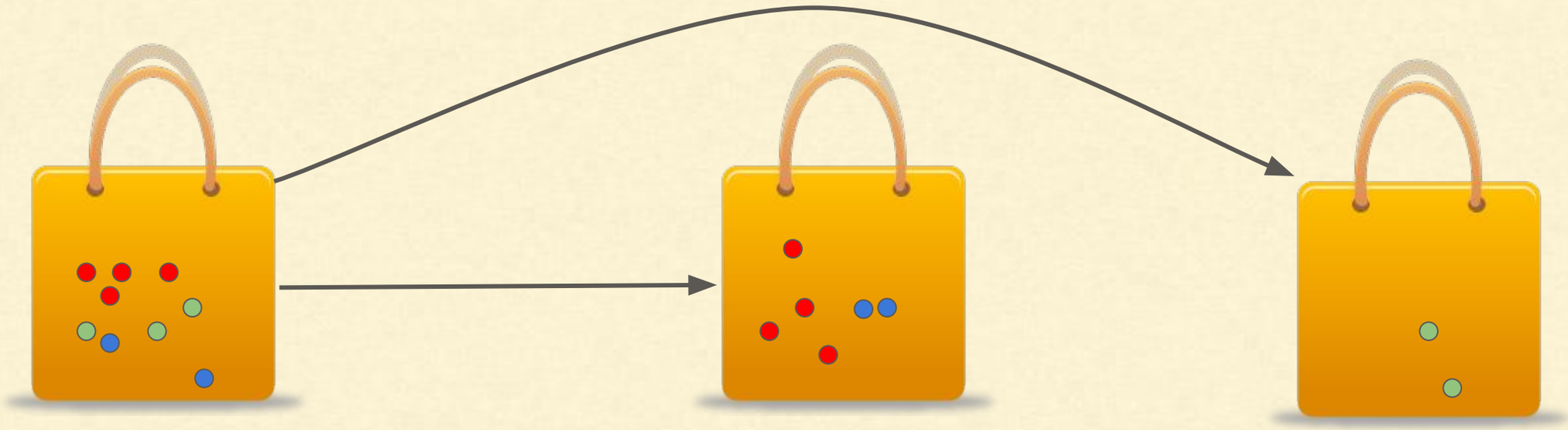
Bagging and Pasting

Pasting

When sampling is performed without replacement, it is called pasting.

Bagging and Pasting

Pasting

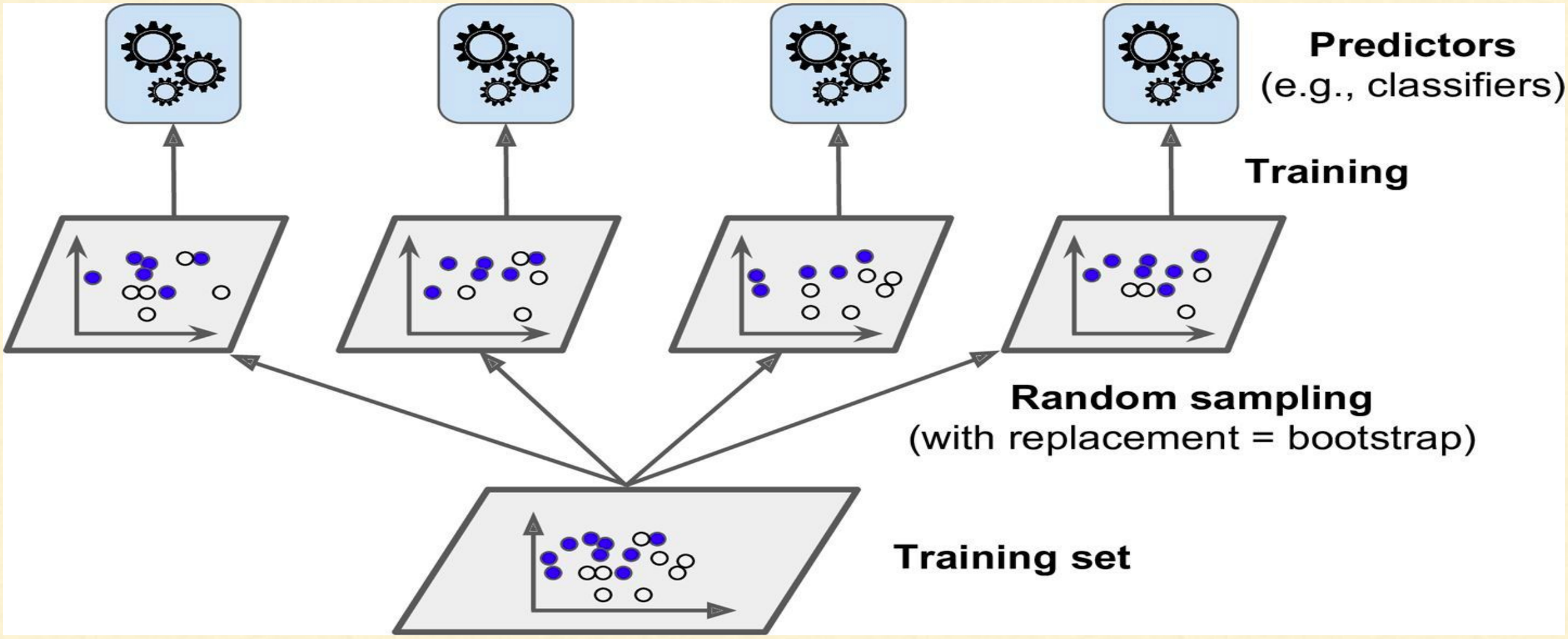


Original Dataset Bag 1 Bag 2

Suppose the original dataset has **4 red, 3 green** and **2 blue balls**.
When we sample without replacement in Bag 1, it has **4 red** and **2 blue**
Again sampling without replacement in Bag 2, it has **2 green balls**
Since we are sampling without replacement bag blue cannot have blue or red balls as all the balls are previously used

Bagging and Pasting

In other words, both bagging and pasting allow training instances to be sampled several times across multiple predictors, but only bagging allows training instances to be sampled several times for the same predictor.



Bagging and Pasting

- 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 (i.e., the most frequent prediction, just like a hard voting classifier) for classification, or the average for regression.

Bagging and Pasting

Advantage of Bagging or Pasting

- Predictors in bagging can all be trained in parallel, via different CPU cores or even different servers.
- Similarly, 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

Bagging and Pasting in Scikit Learn

- Scikit-Learn offers a simple API for both bagging and pasting with the BaggingClassifier class
- Or BaggingRegressor for regression.

Bagging and Pasting

Hands On

- 500 Decision Tree classifiers,
- Each trained on 100 training instances random with replacement
- This is **bagging**, but if you want **pasting**, set "**bootstrap=False**"

Bagging and Pasting

Bagging and Pasting in Scikit Learn

- 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

Bagging and Pasting in Scikit Learn

```
>>> 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)
```

Run it on Notebook

Bagging and Pasting

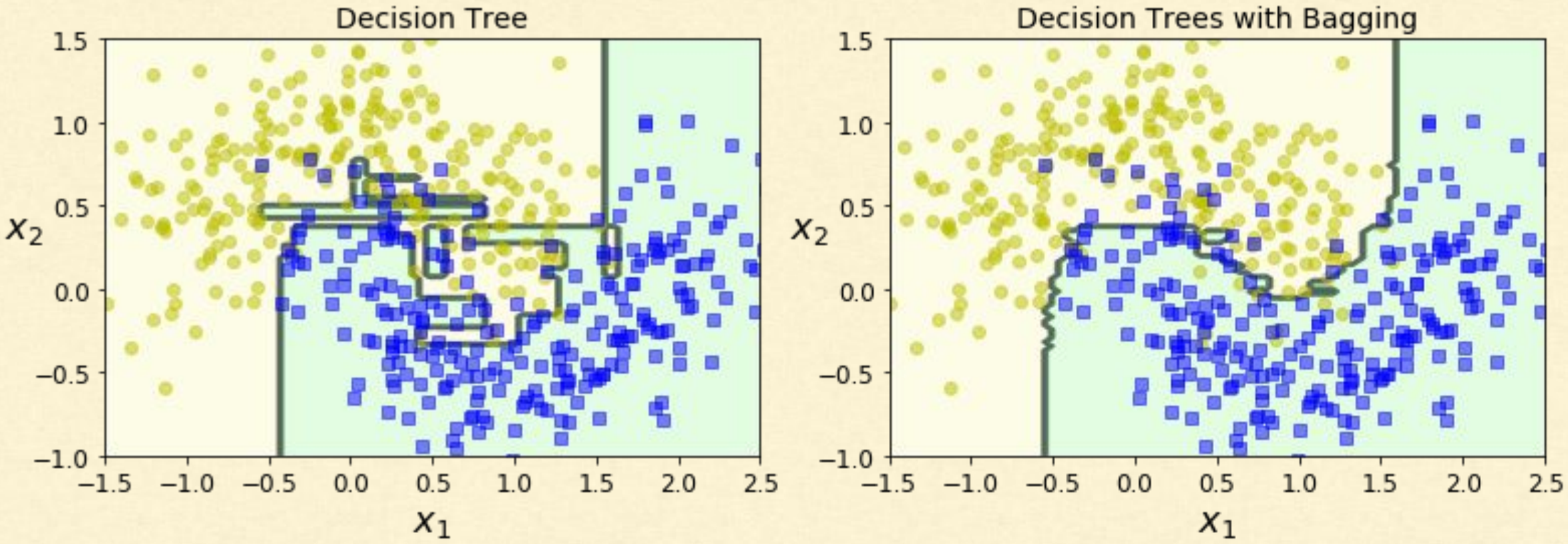
- The BaggingClassifier automatically performs **soft voting** instead of **hard voting** if the base classifier can estimate **class probabilities**
- That means, if it has a **predict_proba()** method it will automatically perform **soft voting.**
- For eg. Decision Tree classifiers, as it has a predict_proba() method

Bagging and Pasting

- Overall, **bagging** often results in **better models**,
- Which explains why it is generally preferred
- Pasting is good for large data-sets
- 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.

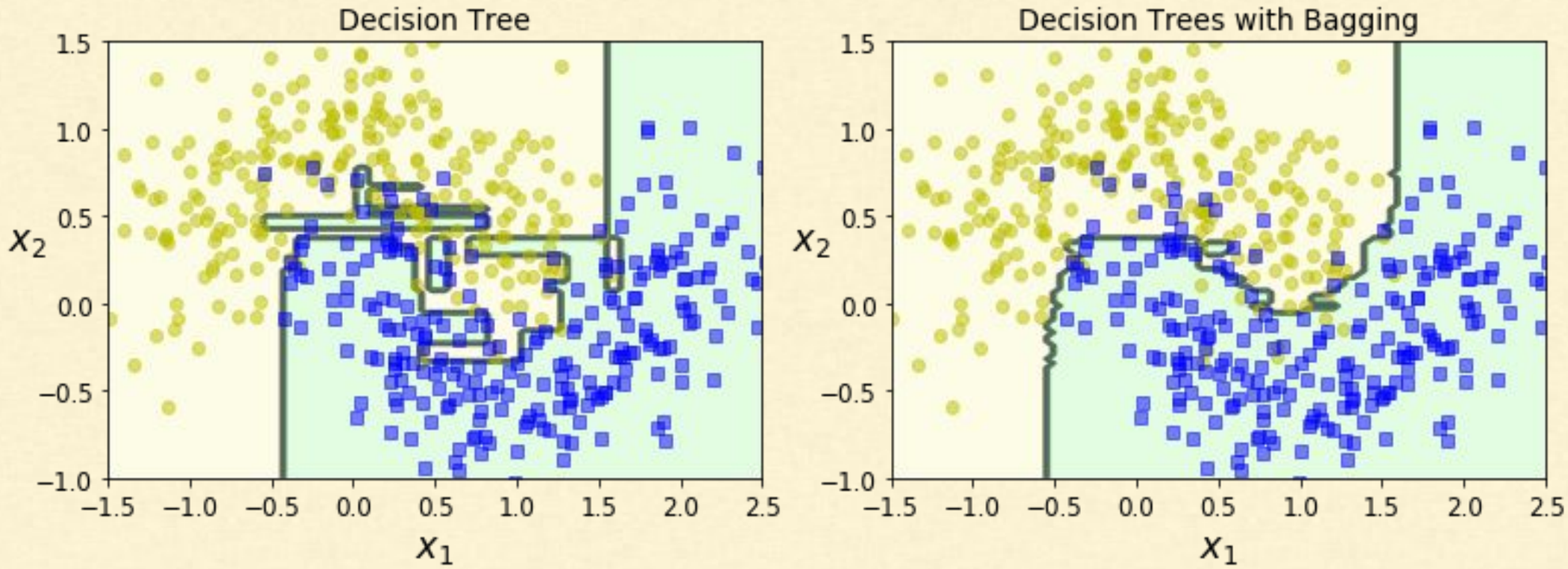
Bagging and Pasting

Decision Boundary of an Ensemble vs Decision Boundary of a single classifier



Bagging and Pasting

Decision Boundary of an Ensemble vs Decision Boundary of a single classifier



Compares the decision boundary of a **single Decision Tree** with the decision boundary of a **bagging ensemble of 500 trees,** both trained on the moons dataset.

Bagging and Pasting

Decision Boundary of an Ensemble vs Decision Boundary of a single classifier

- 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.

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.

- By default a BaggingClassifier samples **m** training instances with replacement (**bootstrap=True**), where **m** is the size of the training set.

Out-of-Bag Evaluation

- As m grows, the ratio of instances which are sampled to the instances that are not samples approaches **1 – exp(–1) ≈ 63.212%**.
- This means that only about **63%** of the training instances are sampled on average for each predictor.
- The remaining **37%** of the training instances that are not sampled are called **out-of-bag (oob)** instances.

Out-of-Bag Evaluation

- 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 on oob instances
- Each predictor is used to predict on the instances it has not seen
  - Thus we have oob error equal to number of predictors
  - The OOB MSE is computed using this OOB Errors

Out-of-Bag Evaluation

Out-of-Bag Evaluation using Scikit Learn

- You can set **oob_score=True** when creating a **BaggingClassifier** to request an automatic oob evaluation after training.
- The resulting evaluation score is available through the **oob_score_** variable.

Out-of-Bag Evaluation

Out-of-Bag Evaluation using Scikit Learn

```
>>> 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_
```

Run it on Notebook

Out-of-Bag Evaluation

Out-of-Bag Evaluation using Scikit Learn

According to this oob evaluation, this BaggingClassifier is likely to achieve about 93.1% 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)
```

Run it on Notebook

Out-of-Bag Evaluation

Out-of-Bag Evaluation using Scikit Learn

The oob decision function for each training instance is also available through the oob_decision_function_ variable

```
>>> bag_clf.oob_decision_function_
array([[ 0. , 1. ],
       [ 0.60588235, 0.39411765],
       [1. , 0. ],
         ...
       [ 0.48958333, 0.51041667]])
```

Out-of-Bag Evaluation

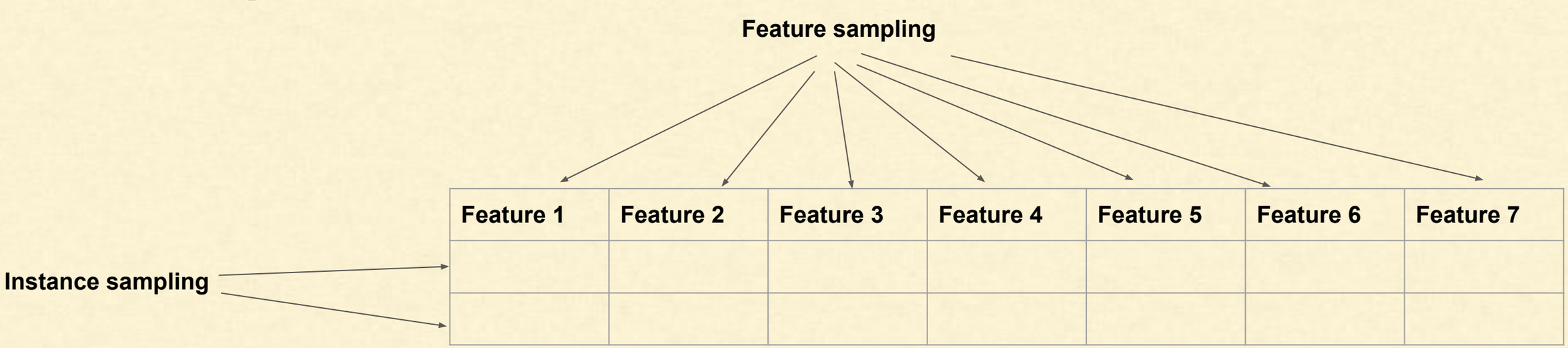
Out-of-Bag Evaluation using Scikit Learn

```
array([[ 0. , 1. ],
       [ 0.60588235, 0.39411765],
       [1. , 0. ],
       ...
       [0. , 1. ],
       [ 0.48958333, 0.51041667]])
```

Here, the oob evaluation estimates that the second training instance has a **60.6%** probability of belonging to the positive class and **39.4%** of belonging to the positive class.

Random Patches and Random Subspaces

- The **BaggingClassifier** class supports sampling the features as well.
- This is controlled by two hyperparameters: **max_features** and **bootstrap_features**.



Random Patches and Random Subspaces

- Works the same way as **max_samples** and **bootstrap**, but for **feature sampling** instead of **instance sampling**.
- Each predictor is trained on a subset of features.
- Particularly useful with high-dimensional inputs (such as images).

Ensemble Learning

Random Patches and Random Subspaces

Sampling both training instances and features is called the **Random Patches method.**

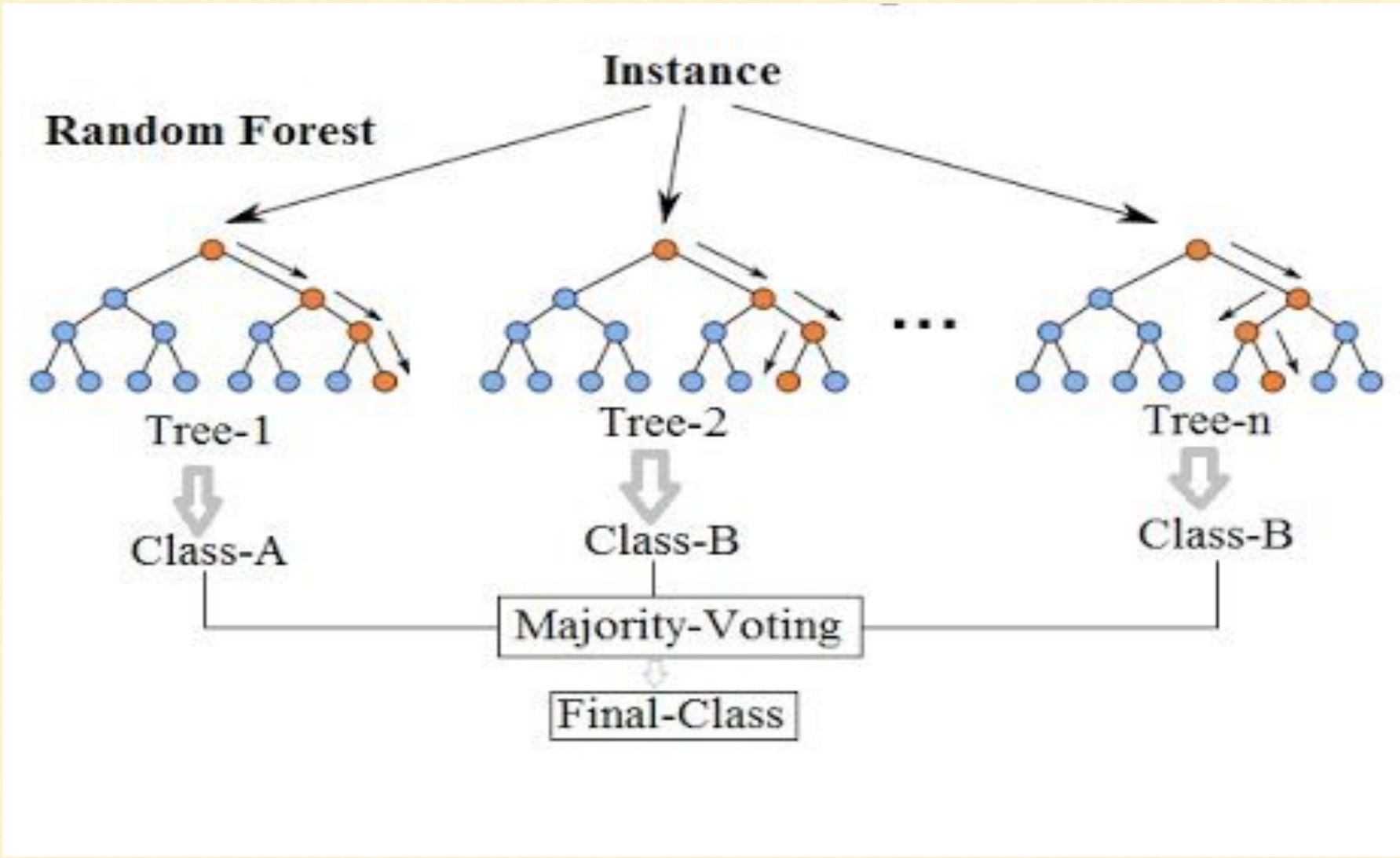
Keeping all training instances (i.e., **bootstrap=False and max_samples=1.0**) but sampling features (i.e., **bootstrap_features=True** and/or **max_features** smaller than **1.0**) is called the **Random Subspaces method.**

Ensemble Learning

max_features	bootstrap_features	bootstrap	max_samples	
< 1	True	NA	1	**Subspaces**
< 1	True	NA	< 1	**Patches**
1	NA	True	< 1	**Bagging**
1	NA	False	< 1	**Pasting**

Random Forests

A **Random Forest** is an ensemble of **Decision Trees**, generally trained via the **bagging method**, typically with **max_samples** set to the size of the training set.



Random Forests

- Instead of building a **BaggingClassifier** and passing it a **DecisionTreeClassifier**, you can instead use the **RandomForestClassifier** class, which is more convenient and optimized for Decision Trees

- Similarly, there is a **RandomForestRegressor** class for regression tasks.

Random Forests

Let us train a **Random Forest** classifier with **500** trees, each limited to maximum 16 nodes, using all available CPU cores:

```
>>> 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)
```

Run it on Notebook

Random Forests

- With a few exceptions, a **RandomForestClassifier** has all the hyperparameters of a **DecisionTreeClassifier**
- Plus all the hyperparameters of a **BaggingClassifier** to control the ensemble itself.

Random Forests

About the 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 (once again) trades a higher bias for a lower variance, generally yielding an overall better model

Random Forests - Extra Trees

- 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.

Random Forests - Extra Trees

- Once again, this trades more bias for a lower variance.
- It also makes **Extra-Trees much faster to train than regular Random Forests** since finding the best possible threshold for each feature at every node is one of the most time-consuming tasks of growing a tree.

Random Forests - Extra Trees

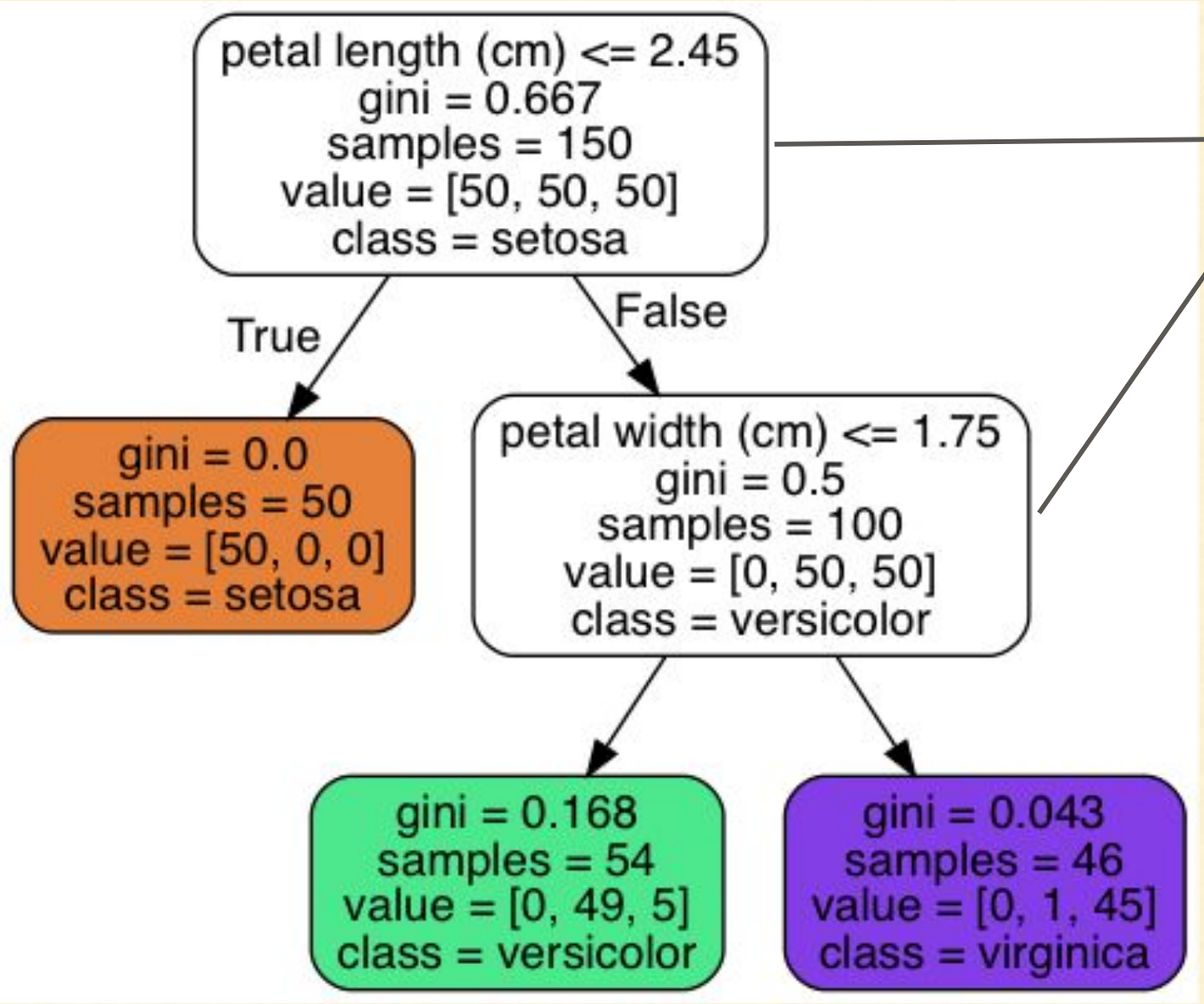
Let's train a Extra Tree using Scikit learn

```
>>> from sklearn.ensemble import ExtraTreesClassifier
>>> extra_tree_clf = ExtraTreesClassifier(n_estimators=500,
max_leaf_nodes=16, n_jobs=-1, random_state=42)
>>> extra_tree_clf.fit(X_train, y_train)
>>> y_pred_extra_trees = extra_tree_clf.predict(X_test)
```

Run it on Notebook

Random Forests - Feature Importance

- Important features are likely to appear **closer to the root** of the tree
- While unimportant features will often appear closer to the leaves or not at all



Random Forests - Feature Importance

- It is possible to get an estimate of a feature's importance by computing the **average depth** at which it appears across all trees in the forest.

- Scikit-Learn computes this automatically for every feature after training.

- You can access the result using the **feature_importances_** variable.

Random Forests - Feature 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)
```

Run it on Notebook

Random Forests - Feature Importance

It seems that the most important features are

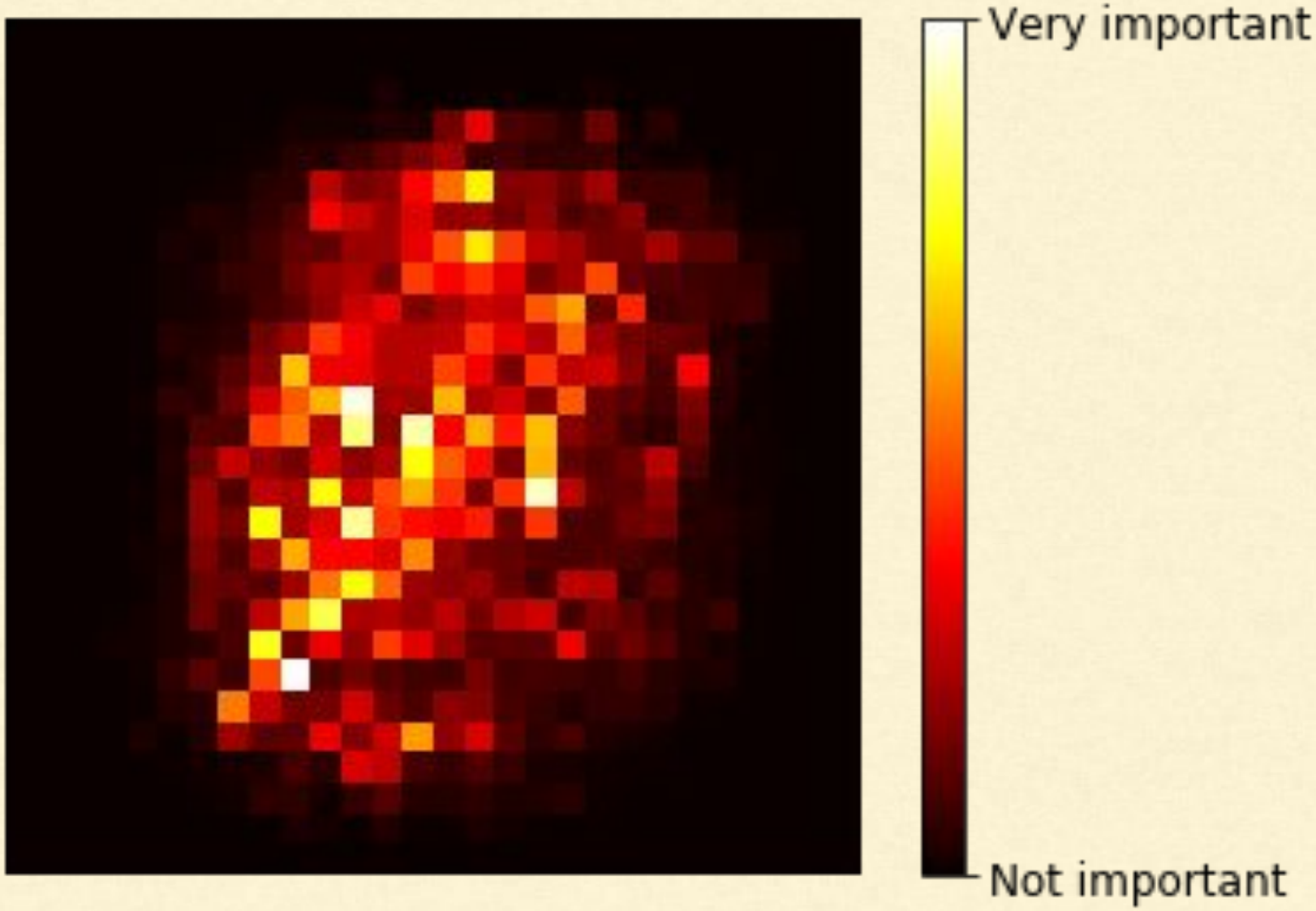
- Petal length (44%)
- Petal width (42%)

Rather unimportant in comparison to petal length and width are

- Sepal length (11%)
- Sepal width (2%)

Random Forests - Feature Importance

If we train a Random Forest classifier on the **MNIST dataset** and plot each pixel's importance, we get the image



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.

Boosting Methods

- Many boosting methods are available
- Most popular are
  - **AdaBoost** (short for Adaptive Boosting)
  - **Gradient Boosting**

AdaBoost

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**.

AdaBoost

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**.

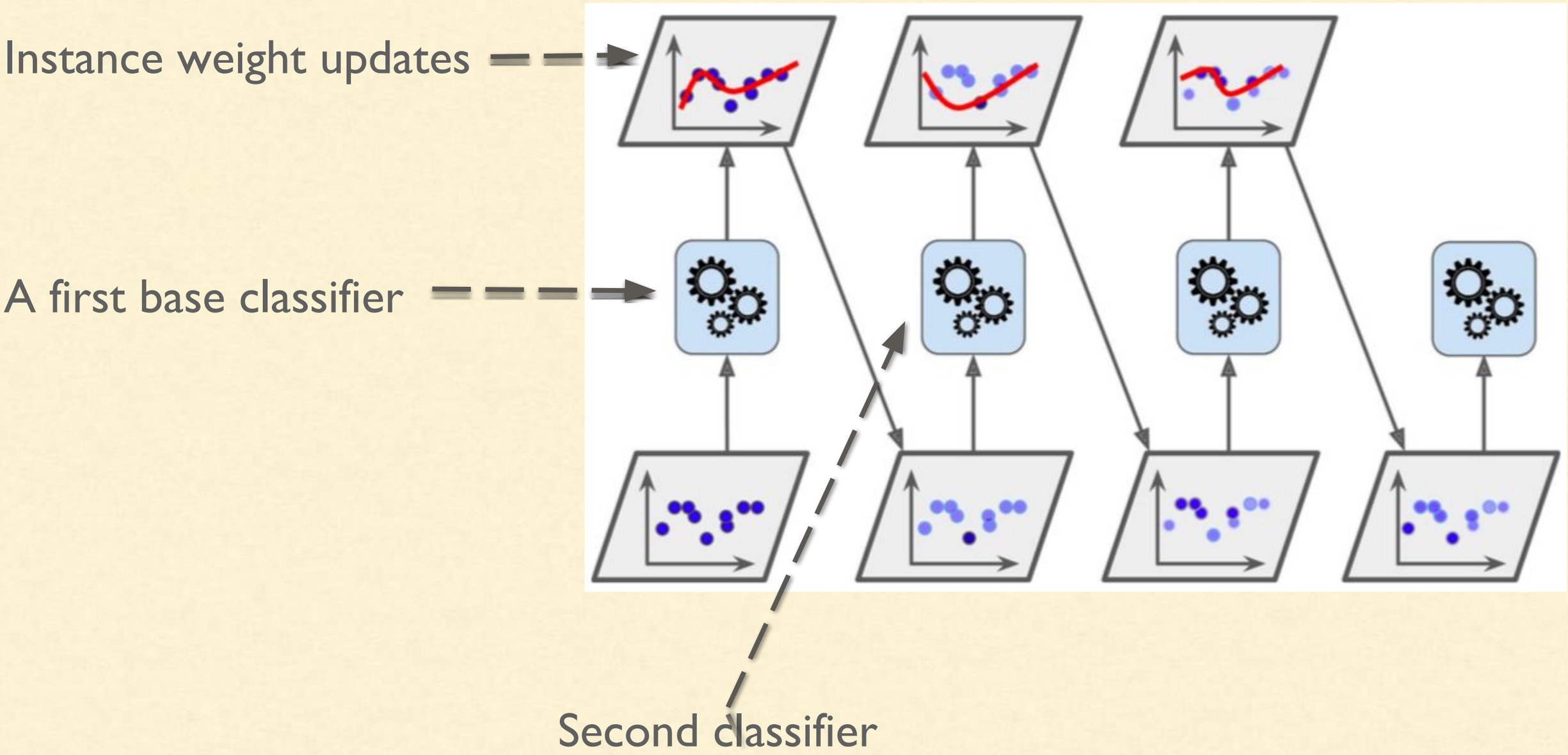
This results in new predictors focusing more and more on the **hard cases**.

This is the technique used by **AdaBoost**.

AdaBoost classifier - Example

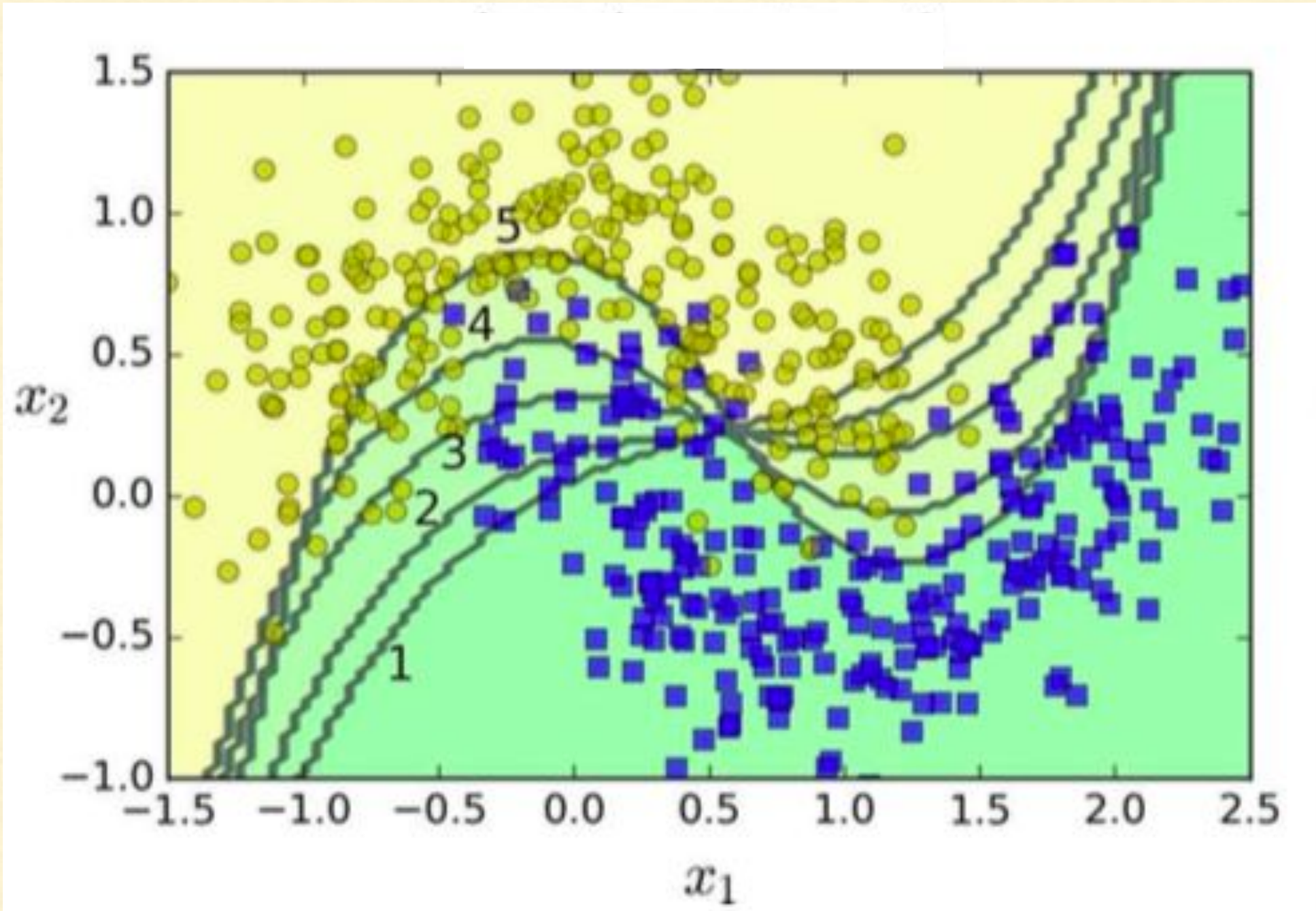
1. A first base classifier (such as a Decision Tree)
    a. Is trained and
    b. Used to make predictions on the training set
    c. The relative weight of **misclassified training** instances is then increased.
2. A second classifier
    a. is trained using the updated weights and again
    b. It makes predictions on the training set,
    c. weights are updated, and so on

AdaBoost classifier - Example



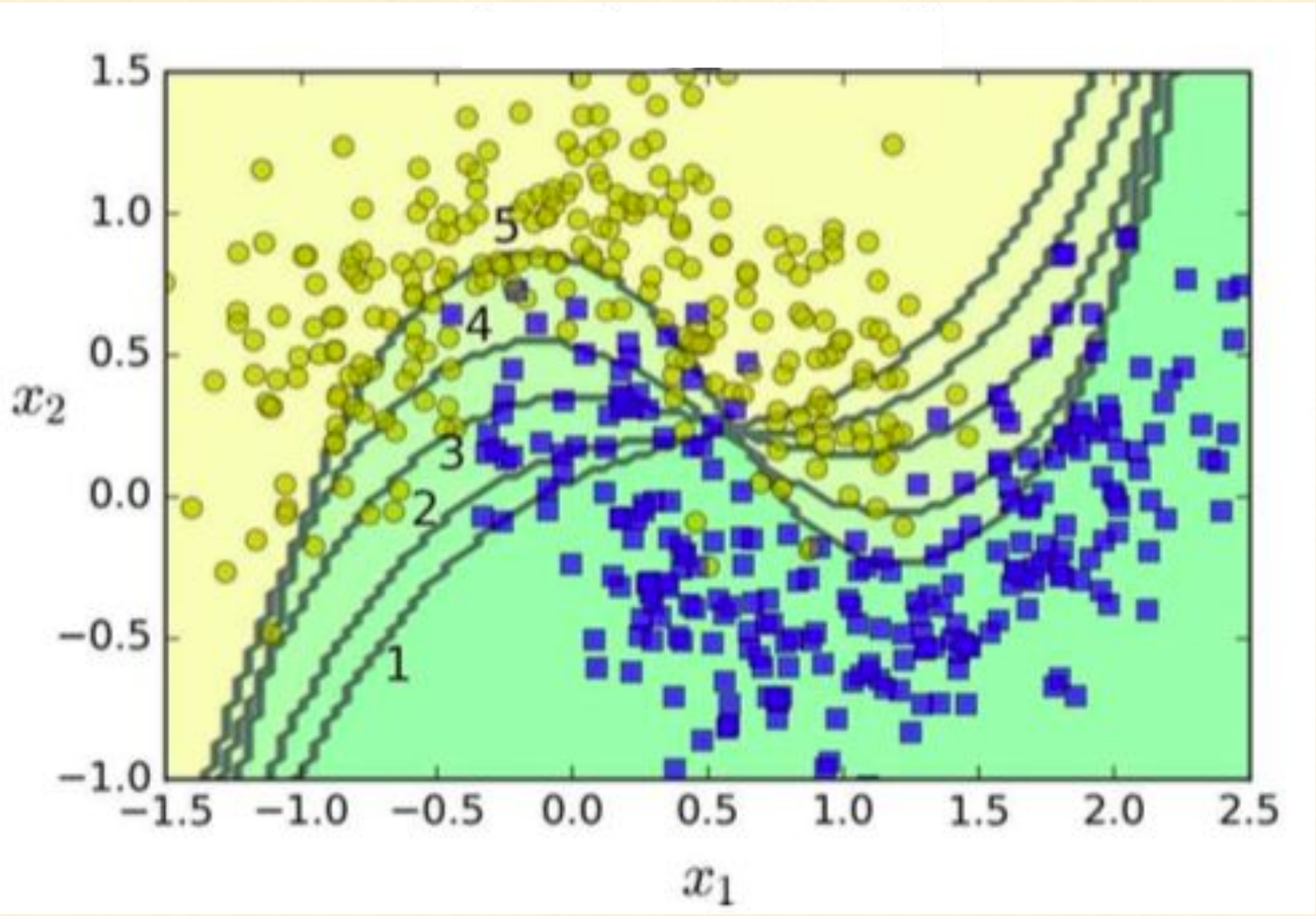
AdaBoost classifier - Example

The decision boundaries of five consecutive predictors on the moons dataset
In this example, each predictor is a highly regularized SVM classifier with an RBF kernel



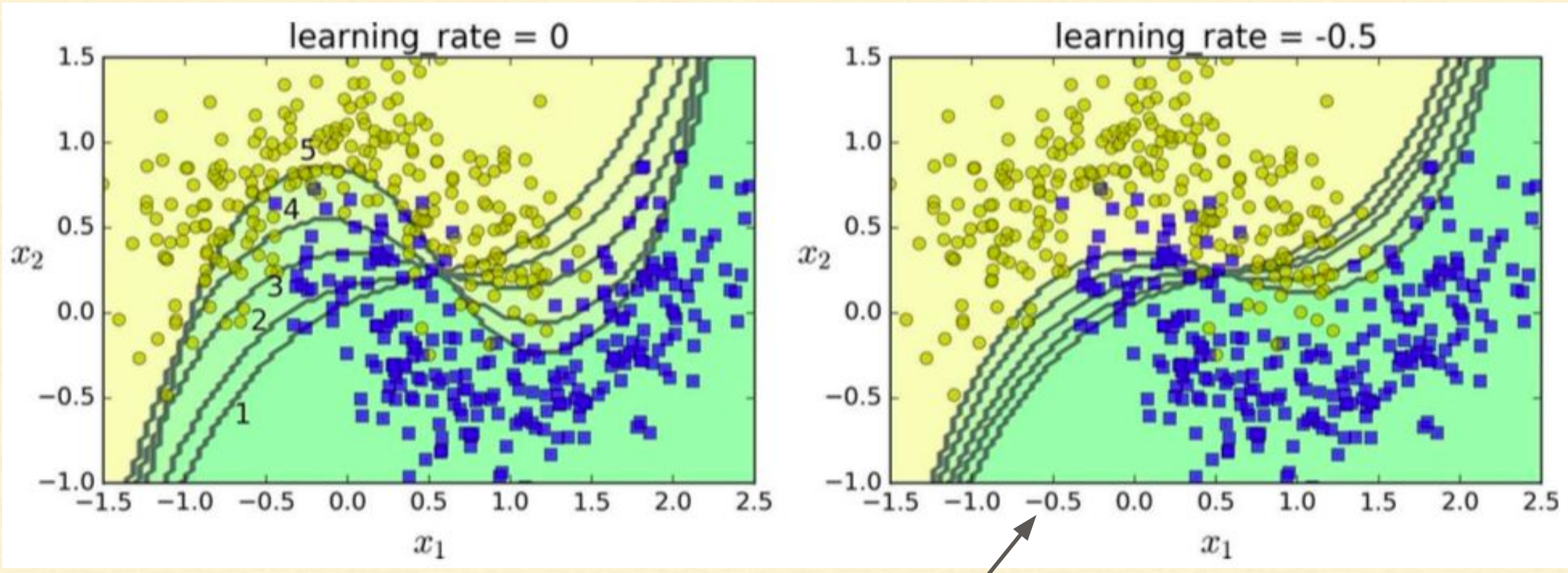
AdaBoost classifier - Example

The decision boundaries of five consecutive predictors on the moons dataset (in this example, each predictor is a highly regularized SVM classifier with an RBF kernel



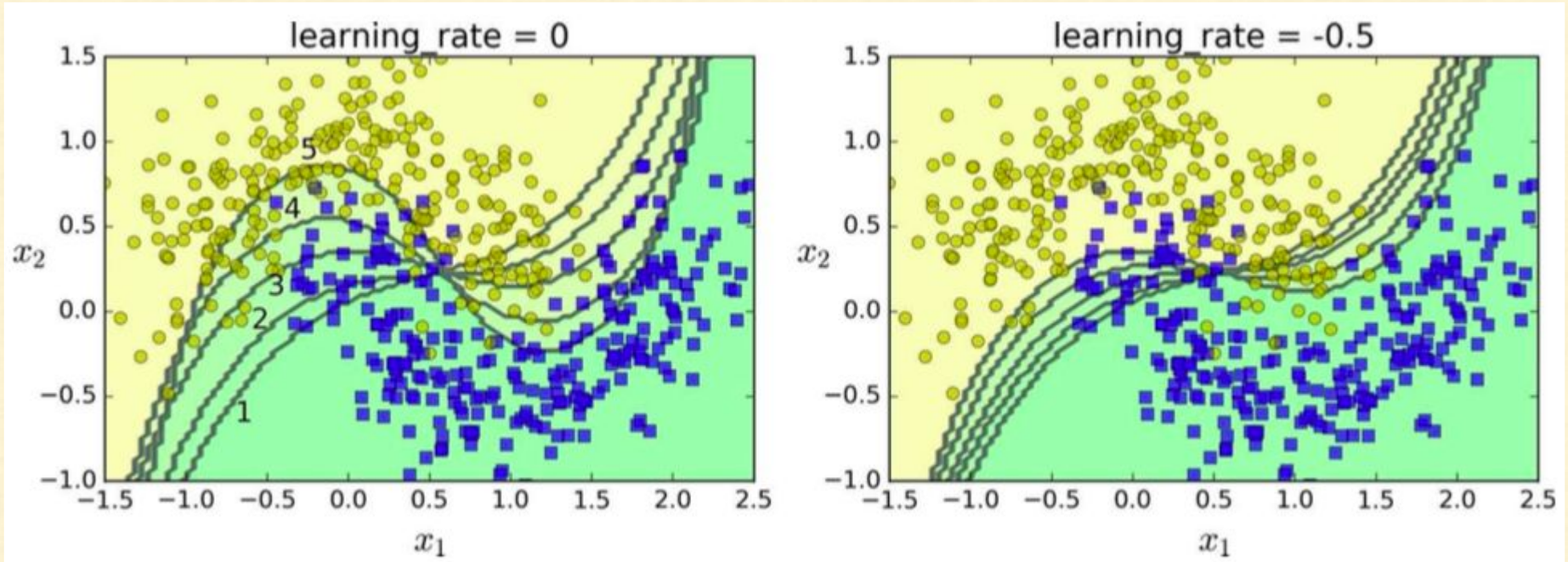
- The first classifier gets many instances wrong,
- So their weights get boosted.
- The second classifier therefore does a better job on these instances, and so on.

AdaBoost classifier - Example



Same sequence of predictors except that the learning rate is halved (i.e., the misclassified instance weights are boosted half as much at every iteration).

AdaBoost classifier - Example



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.

AdaBoost

- 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 - Drawback

- It is a sequential learning technique.
- It cannot be parallelized (or only partially),
- Since each predictor can only be trained after the previous predictor has been trained and evaluated.
- As a result, it does not scale as well as bagging or pasting.

AdaBoost Algorithm - Weighted error rate

1. Each instance weight $w^{(i)}$ is initially set to 1/m.
2. A first predictor is trained and its weighted error rate r_1 is computed on the training set
3. The weights define the probability of an instance selection.

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

Weighted error rate of the j^{th} predictor

Where $\hat{y}_j^{(i)}$ is the prediction of j^{th} predictor for i^{th} instance

AdaBoost Algorithm - Predictor weight

3. The predictor's weight α_j is then computed

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

η is the learning rate hyperparameter (defaults to 1)

- 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**.
- However, if it is most **often wrong** (i.e., less accurate than random guessing), then its weight will be **negative**.

AdaBoost Algorithm - Weight update rule

4. Next the instance weights are updated using:

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

$$w^{(i)} \leftarrow \begin{cases} w^{(i)} & \text{if } \hat{y}_j^{(i)} = y^{(i)} \\ w^{(i)} \exp(\alpha_j) & \text{if } \hat{y}_j^{(i)} \neq y^{(i)} \end{cases}$$

The misclassified instances are boosted.

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

AdaBoost Algorithm - Weight update rule

6. Finally, a new predictor is trained using the updated weights,
7. And the whole process is repeated:
    1. The new predictor's weight is computed,
    2. The instance weights are updated,
    3. 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 Algorithm - Predictions

To make predictions, AdaBoost simply

- Computes the predictions of all the predictors and
- Weighs them using the predictor weights α_j.
- The predicted class is the one that receives the majority of weighted votes (soft)

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

AdaBoost Algorithm - sklearn

Scikit-Learn actually uses a

- Multiclass version of AdaBoost called SAMME16
    - Stagewise Additive Modeling using a Multiclass Exponential loss function
- When there are just two classes, SAMME is equivalent to AdaBoost.

Moreover, if the predictors can estimate class probabilities

- (i.e., if they have a predict_proba() method),
- Scikit-Learn can use a variant of SAMME called SAMME.R
- (the R stands for “Real”),
- which relies on class probabilities rather than predictions and generally performs better.

AdaBoost Algorithm - sklearn

The following code

- Trains an AdaBoost classifier
- based on 200 Decision Stumps
- using Scikit-Learn's AdaBoostClassifier class
- (as you might expect, there is also an AdaBoostRegressor 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 — 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 Algorithm - Regularization

```
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)
```

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

Gradient Boosting

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

- But, 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

Let us try to understand how Gradient boosting works by using Decision Trees as the base predictors.

Here are the steps we will do -

- First, we'll fit a DecisionTreeRegressor to the training set

- Next we'll train a second DecisionTreeRegressor on the residual errors made by the first predictor

- Then again we'll train a third regressor on the residual errors made by the second predictor

- Then we'll have an ensemble containing three trees. It can make predictions on a new instance simply by adding up the predictions of all the trees

Gradient Boosting

We will generate a noisy quadratic training set

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

- First, we'll fit a DecisionTreeRegressor to the training set

```
>>> from sklearn.tree import DecisionTreeRegressor
>>> tree_reg1 = DecisionTreeRegressor(max_depth=2)
>>> tree_reg1.fit(X, y)
```

Gradient Boosting

- 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)
```

Gradient Boosting

- Then again we'll 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)
```

Gradient Boosting

- Then we'll have an ensemble containing three trees. It can make predictions on a new instance simply by adding up the predictions of all the trees

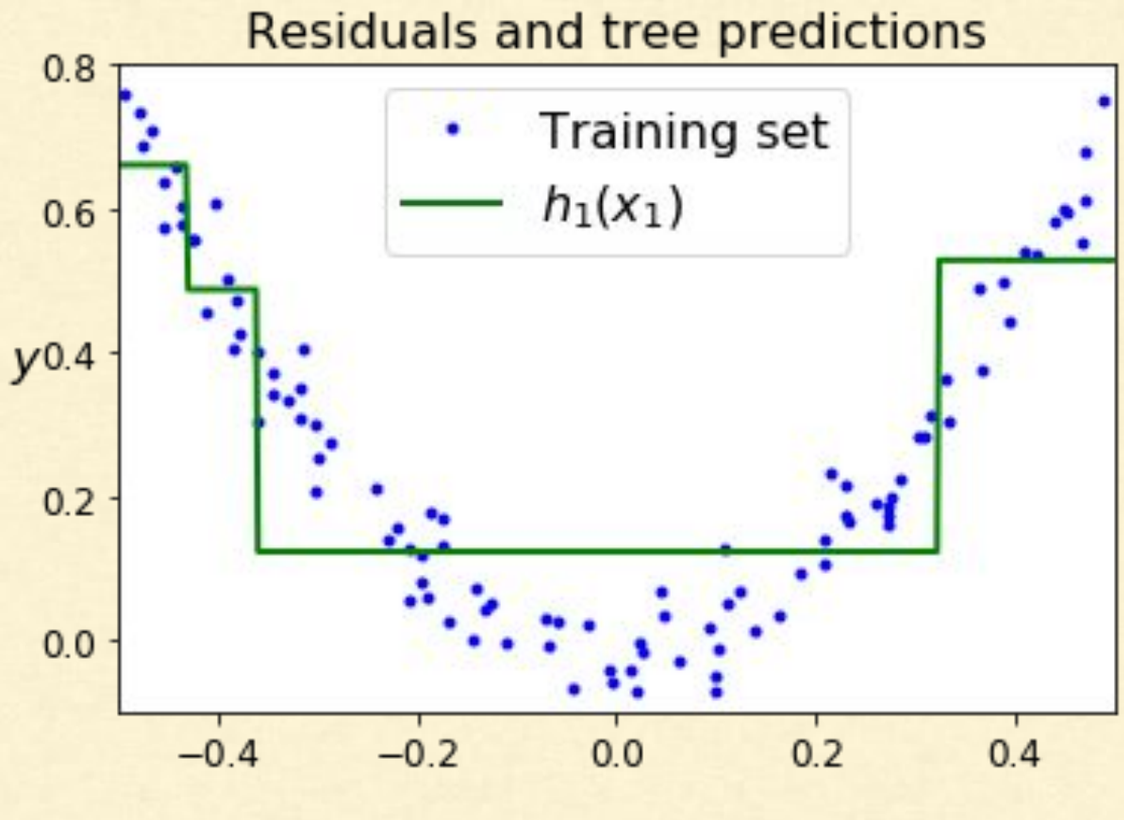
```
>>> y_pred = sum(tree.predict(X_new) for tree in
(tree_reg1, tree_reg2, tree_reg3))
```

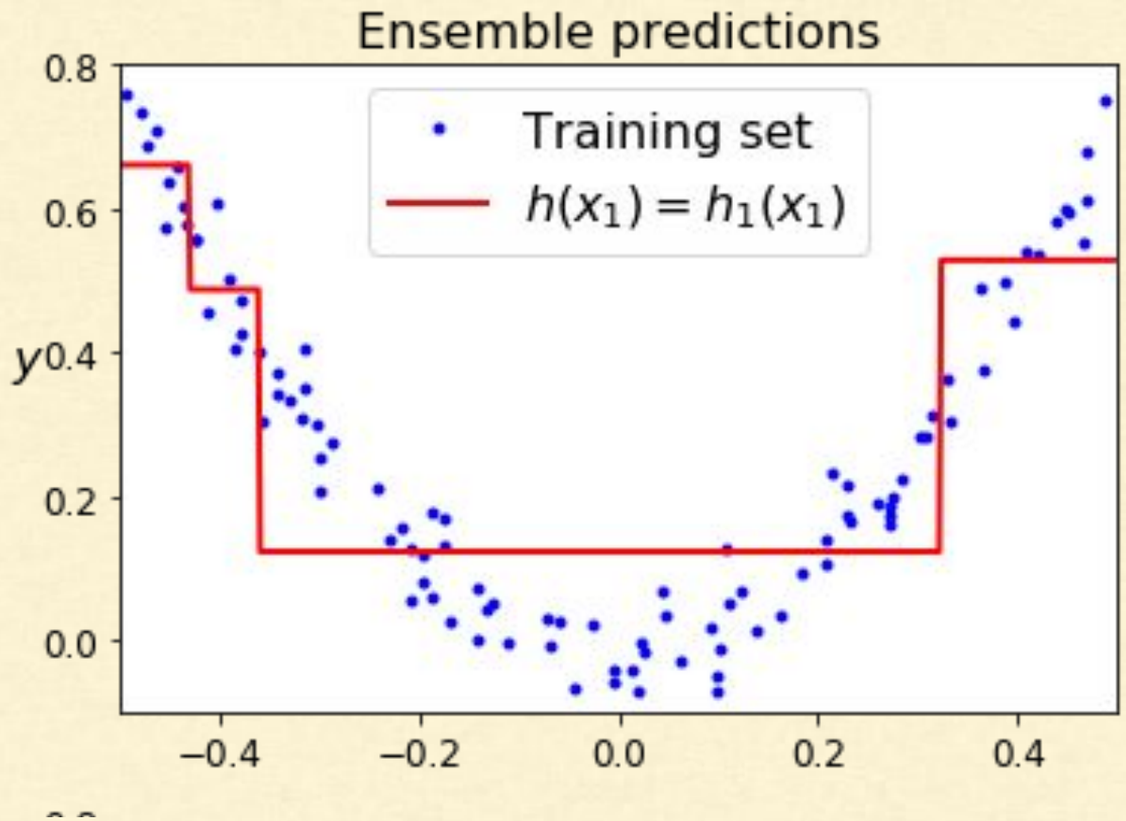
Run it on Notebook

Gradient Boosting

After the first step

The ensemble has just one tree, so its predictions are exactly the same as the first tree's predictions.

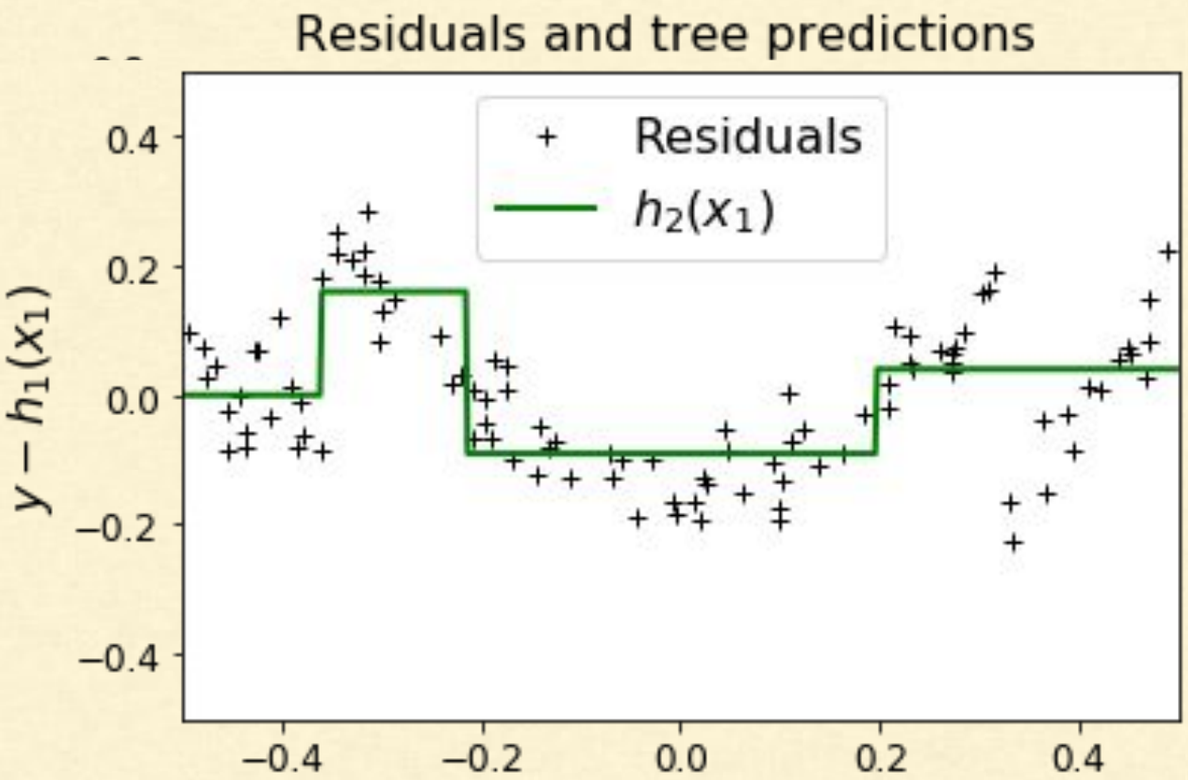


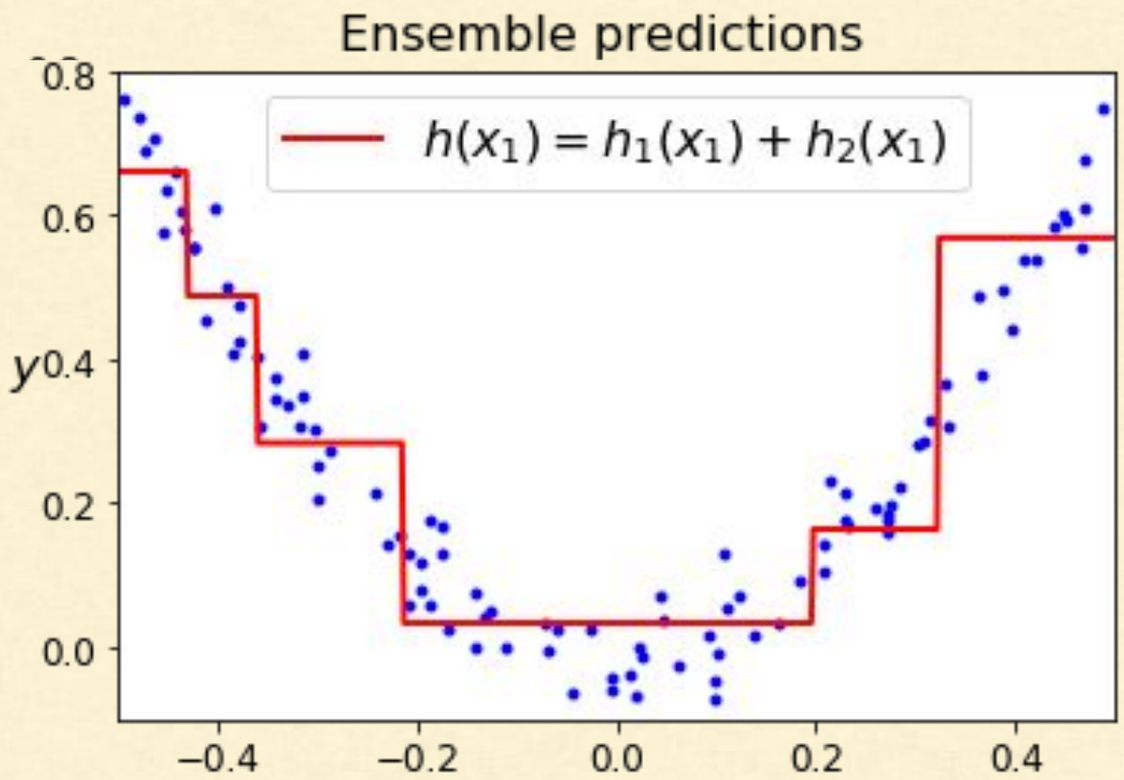


Gradient Boosting

After the second step

A new tree is trained on the residual errors of the first tree, on the left.
On the right you can see that the ensemble's predictions are equal to the sum of the predictions of the first two trees.



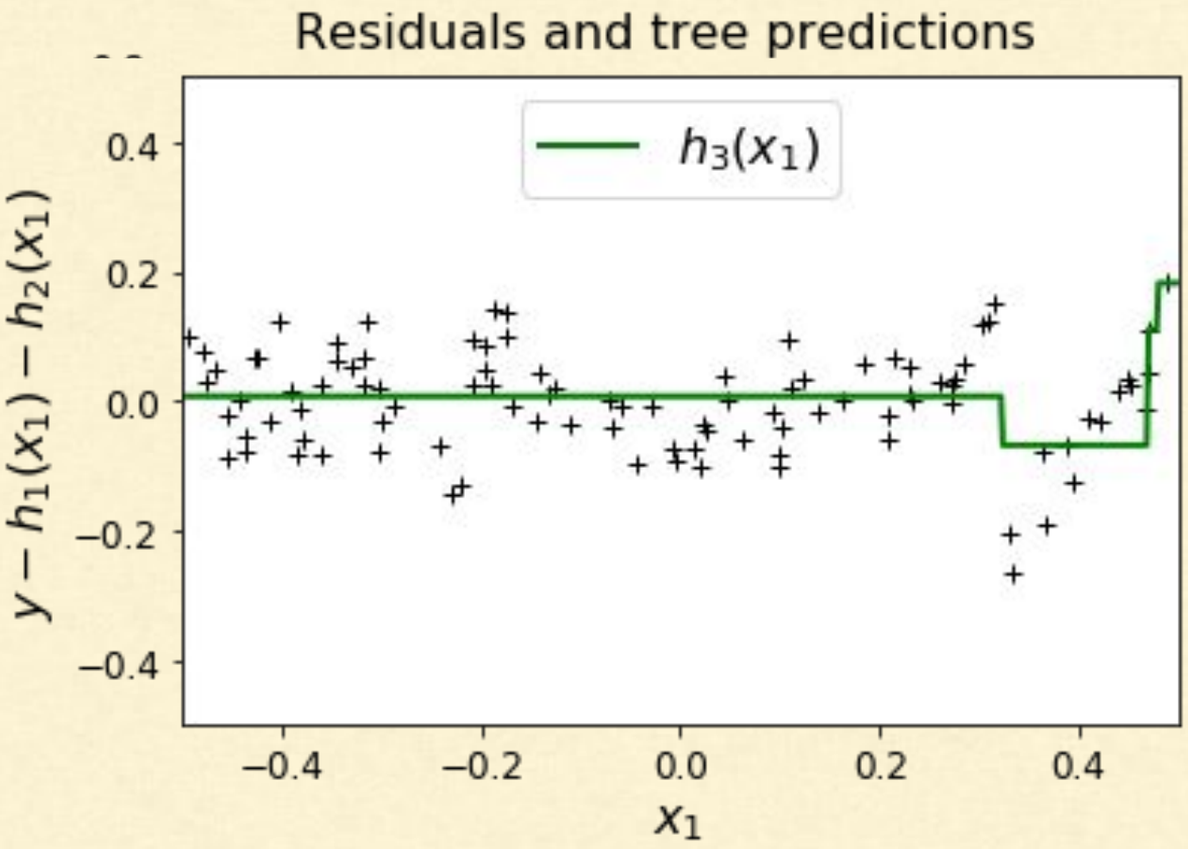


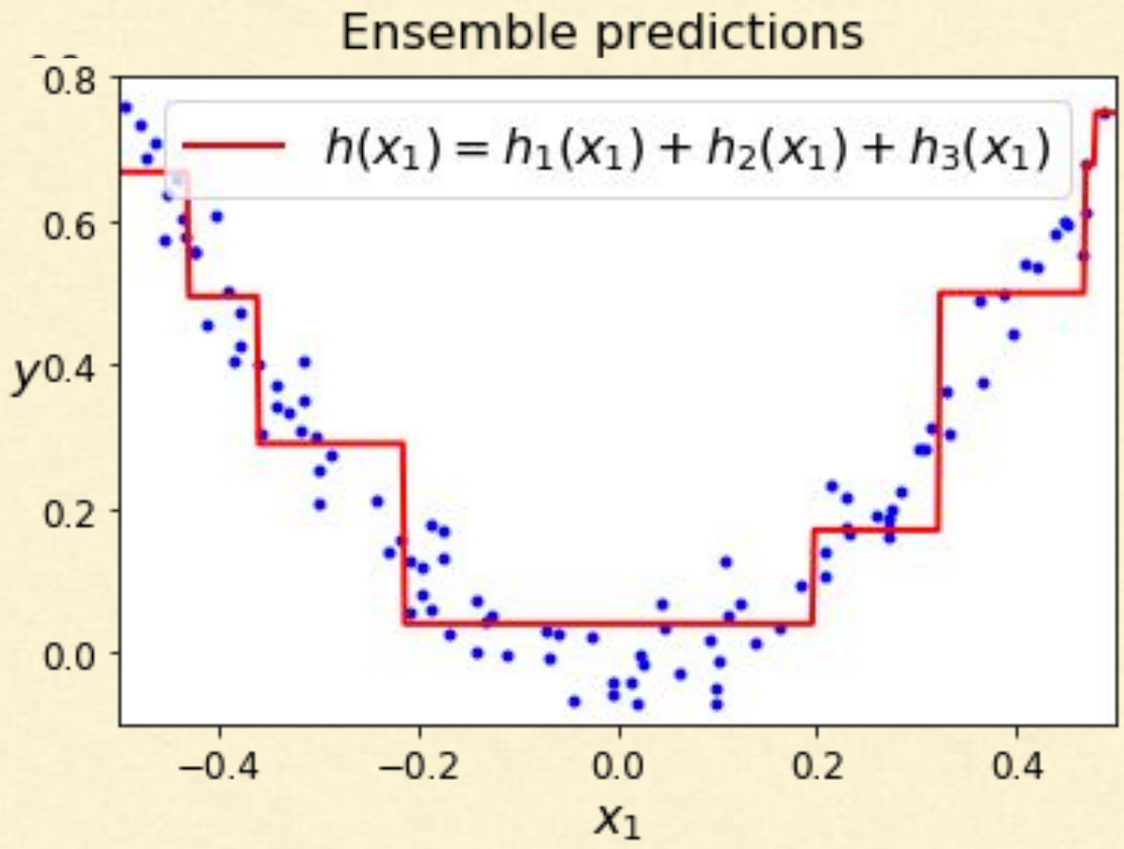
Gradient Boosting

After the third step

Another tree is trained on the residual errors of the second tree.

The ensemble's predictions gradually get better as trees are added to the ensemble.





Gradient Boosting

- A simpler way to train GBRT ensembles is to use Scikit-Learn's **GradientBoostingRegressor** class.

- Just like the RandomForestRegressor class, it has hyperparameters to control the growth of Decision Trees (e.g., **max_depth**, **min_samples_leaf**), as well as hyperparameters to control the ensemble training, such as the number of trees (**n_estimators**).

Gradient Boosting

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

Run it on Notebook

Gradient Boosting - Regularization

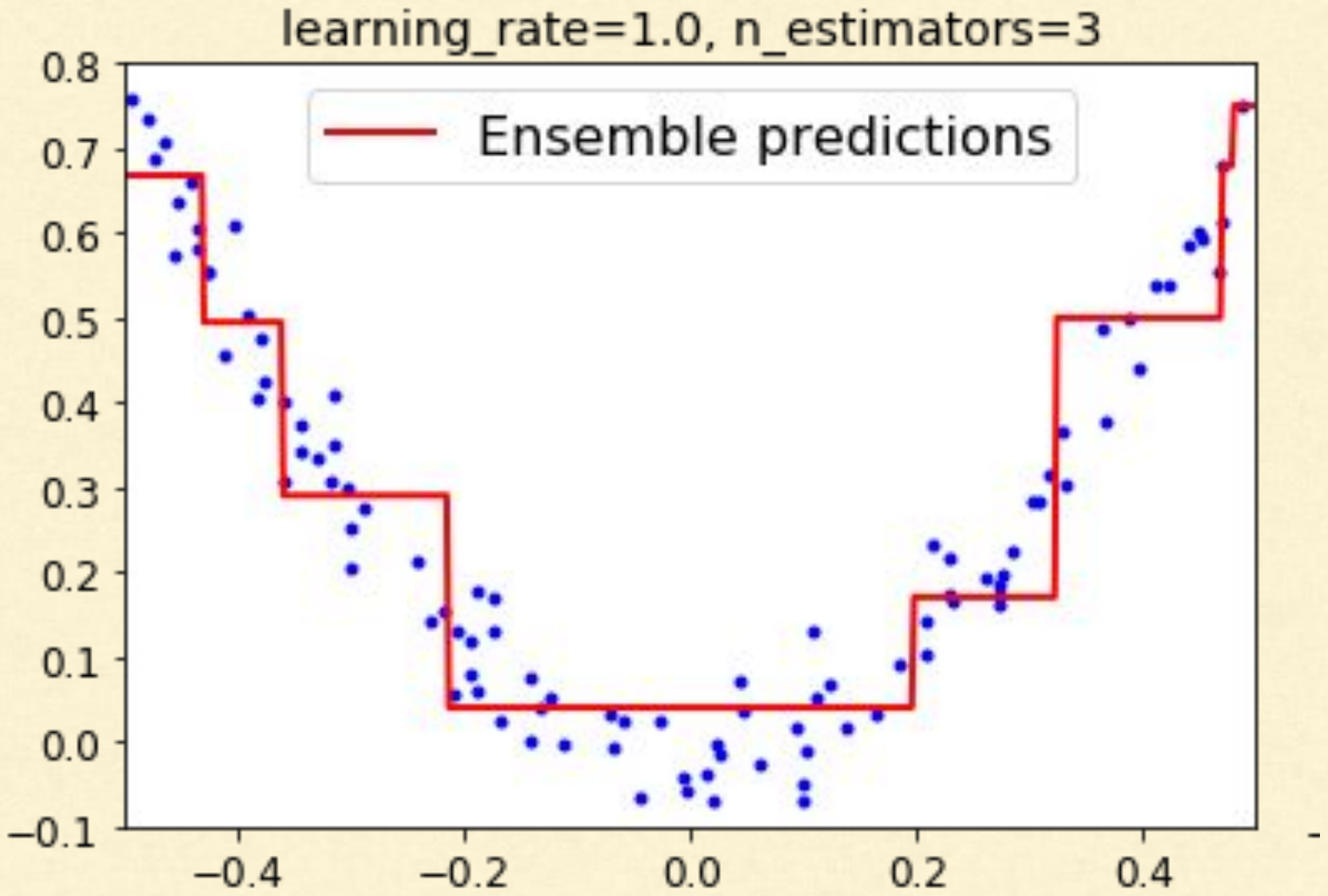
- The **learning_rate** hyperparameter scales the contribution of each tree.

- 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**.

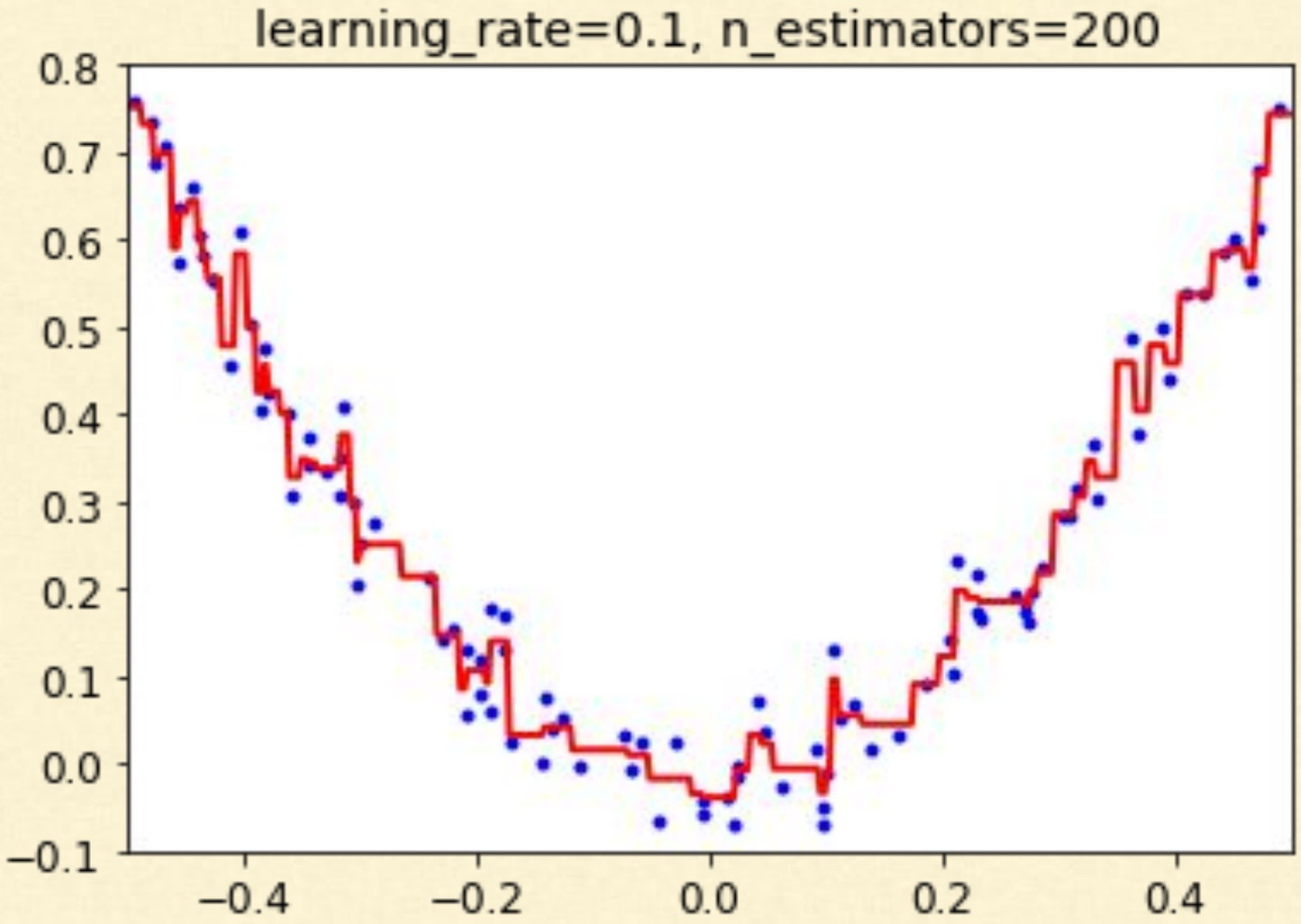
Gradient Boosting - Regularization

The below GBRT ensembles are trained with a low learning rate hence they do not have enough trees to fit the training set



Gradient Boosting - Regularization

Whereas the below GBRT ensembles are trained with a learning rate of 1, hence they have enough trees to fit the training set



Gradient Boosting

How to find the optimal number of trees ???

Gradient Boosting - 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 ,first with one tree, then two trees, and so on.

Let’s see it in action

Gradient Boosting - Early stopping

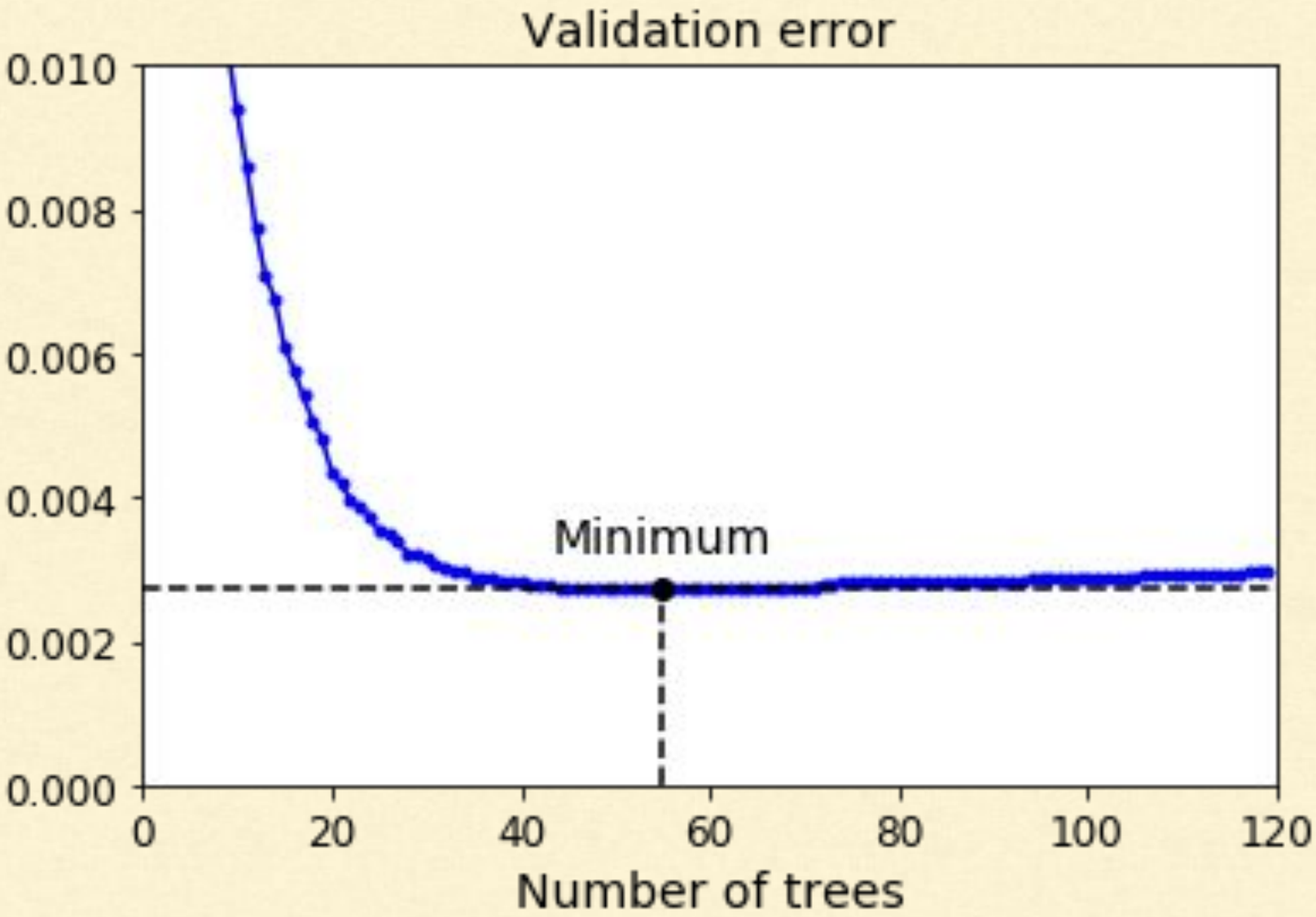
The following code trains a **GBRT ensemble** with **120** trees, then measures the validation error at each stage of training to find the optimal number of trees, and finally trains another **GBRT ensemble** using the optimal number of trees.

```
>>> 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)
>>> gbrt_best = GradientBoostingRegressor(max_depth=2, n_estimators =
bst_n_estimators)
>>> gbrt_best.fit(X_train, y_train)
```

Run it on Notebook

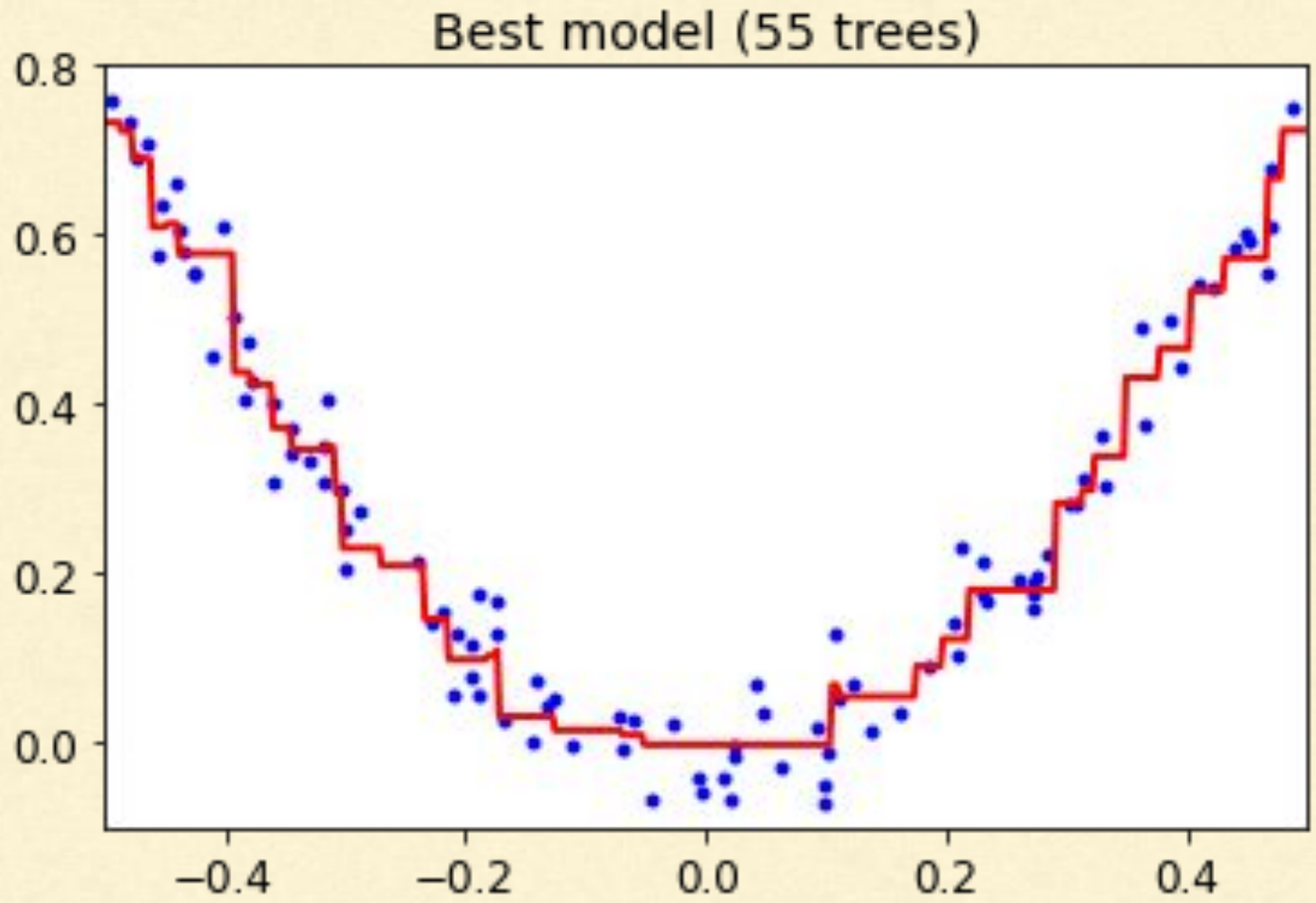
Gradient Boosting - Early stopping

The validation error varies as shown below, as we can see the best value for **`n_estimators`** `is near to 55`



Gradient Boosting - Early stopping

The best model's prediction is shown below. It is constructed with `n_estimators = 55`.



Gradient Boosting - Early stopping

- It is also 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.

- You can do so by setting **warm_start=True**, which makes Scikit-Learn keep existing trees when the fit() method is called, allowing incremental training.

Gradient Boosting - Early stopping

The following code stops training when the validation error does not improve for five 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
```

Run it on Notebook

Gradient Boosting - Stochastic Gradient Boosting

- The GradientBoostingRegressor class also supports a **subsample hyperparameter**, which 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**.

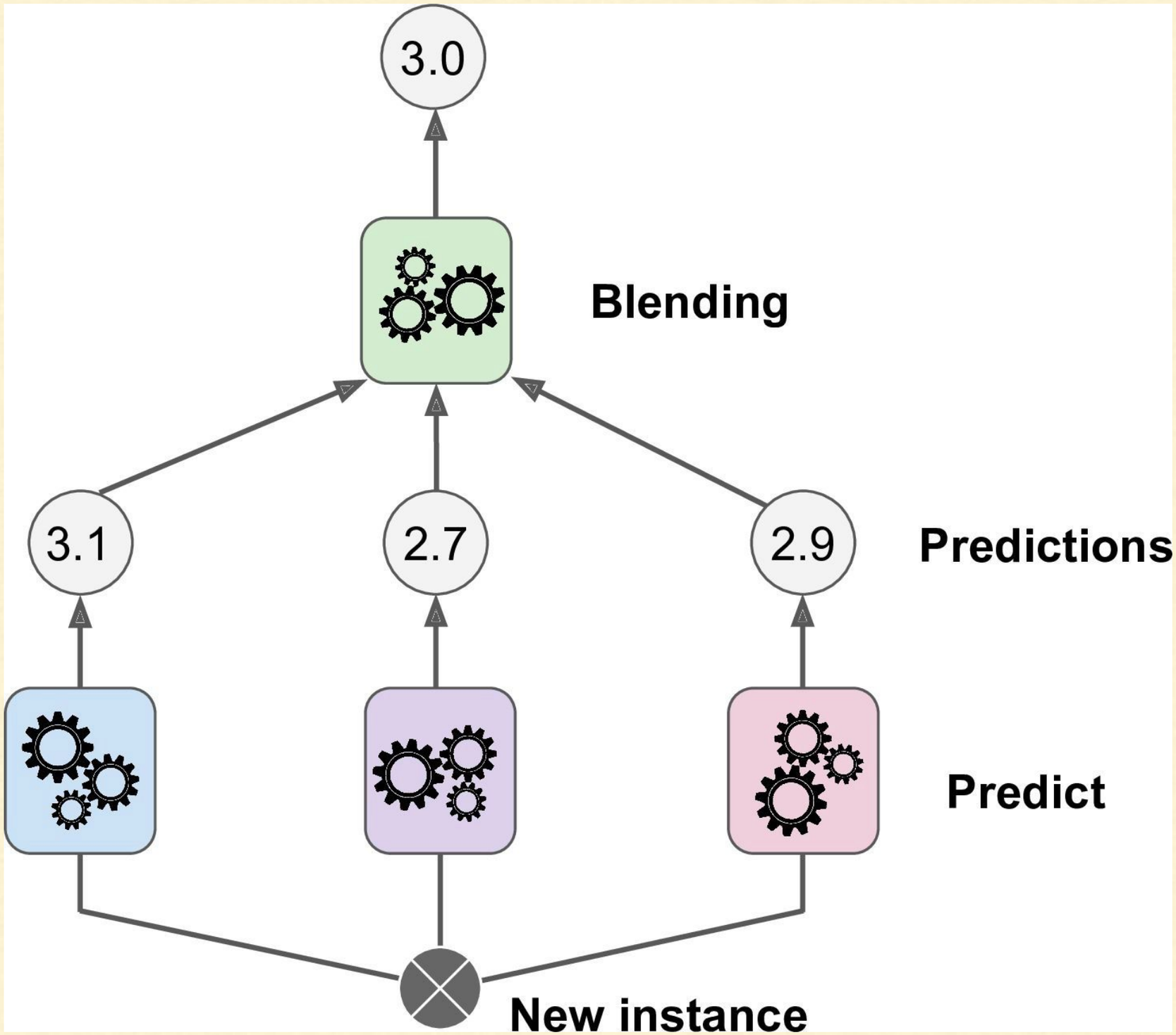
- It is possible to use Gradient Boosting with other cost functions. This is controlled by the **loss** hyperparameter

Gradient Boosting - Stacking

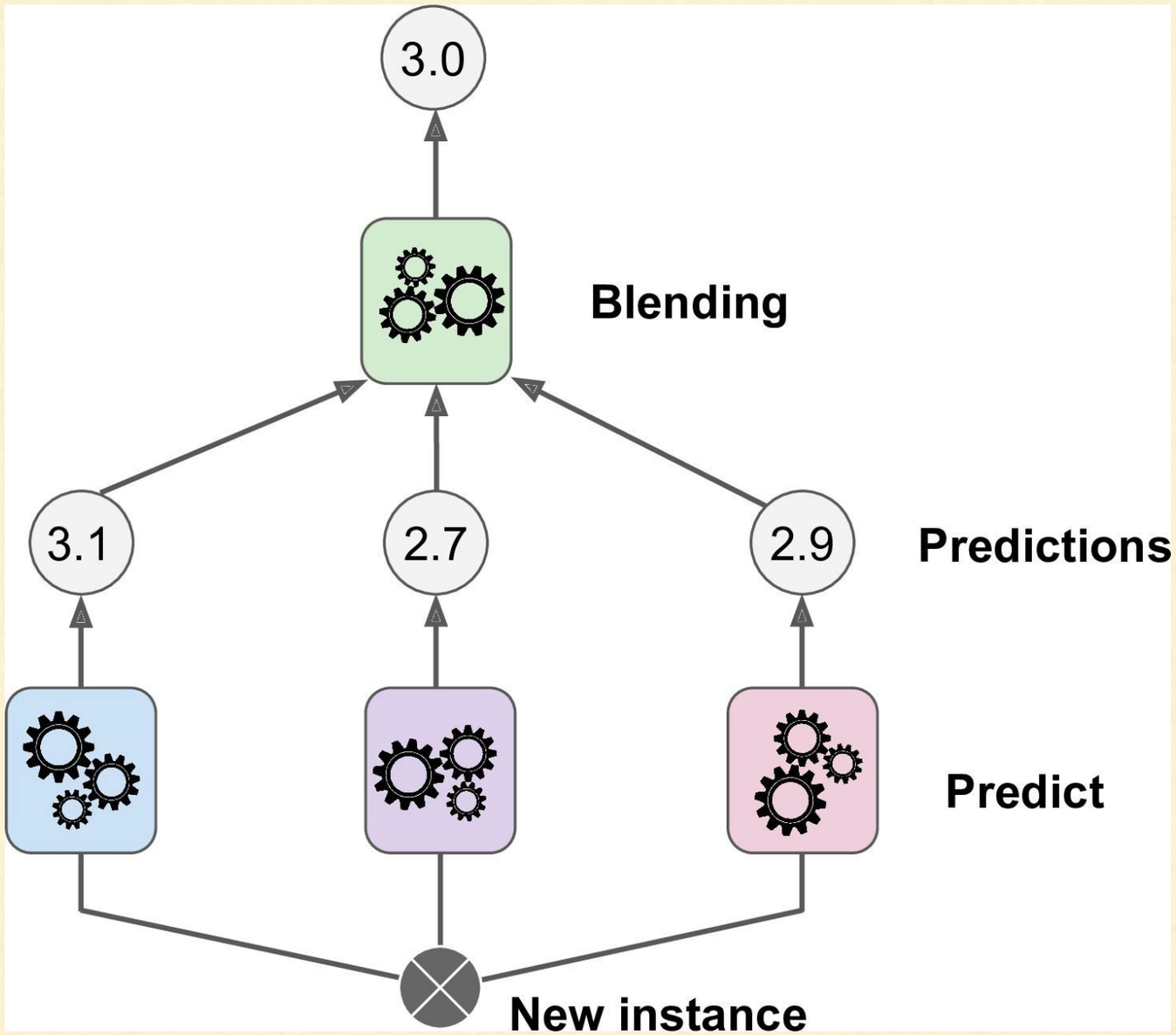
Stacking is a ensemble method that is based on a simple idea -

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

Gradient Boosting - Stacking



Gradient Boosting - Stacking

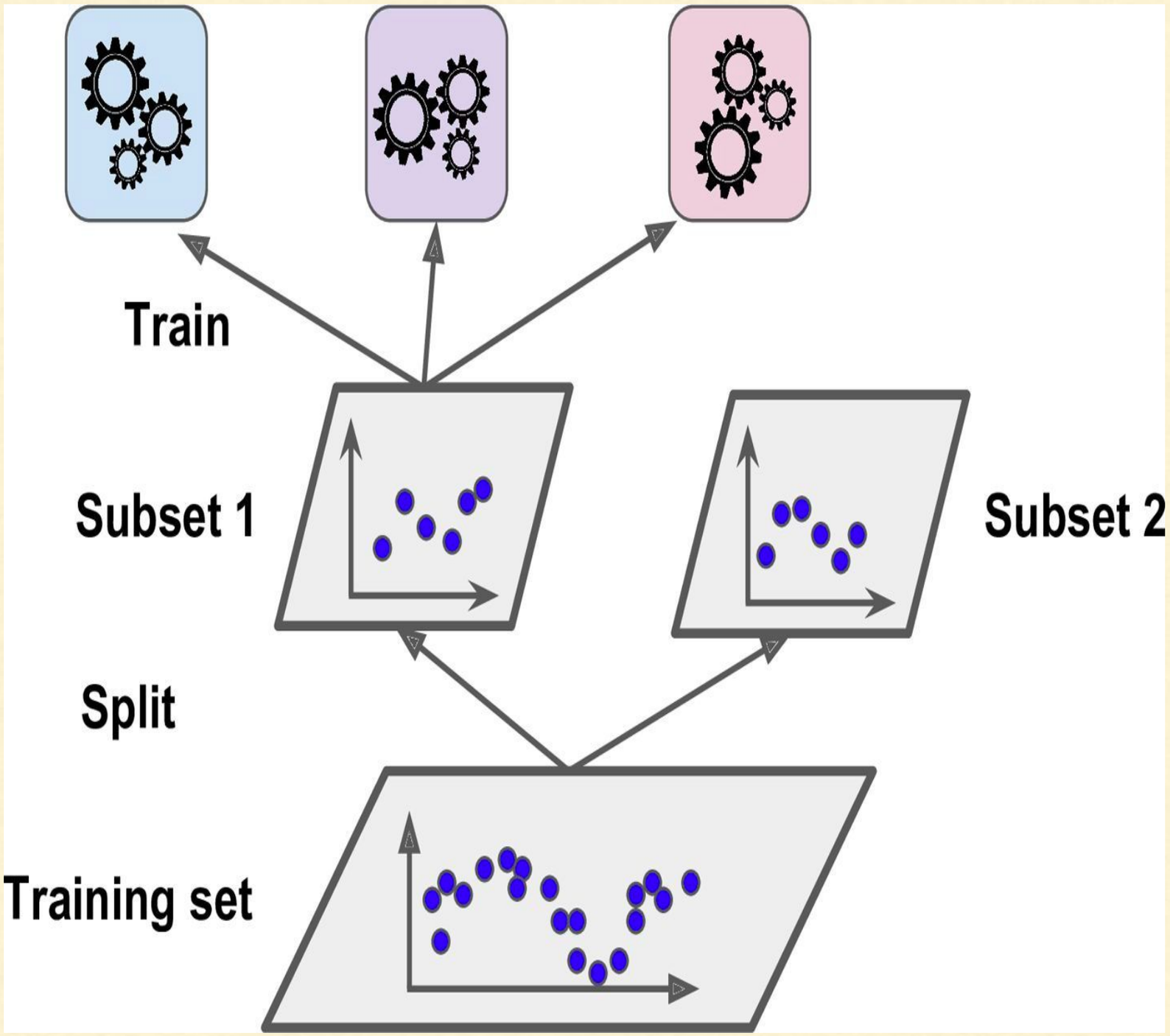


- Each of the bottom three predictors predicts a different value (**3.1**, **2.7**, and **2.9**)
- Then the final predictor, called a **blender**, or a **meta learner**) takes these predictions as inputs and makes the final prediction (**3.0**).

Gradient Boosting - Stacking

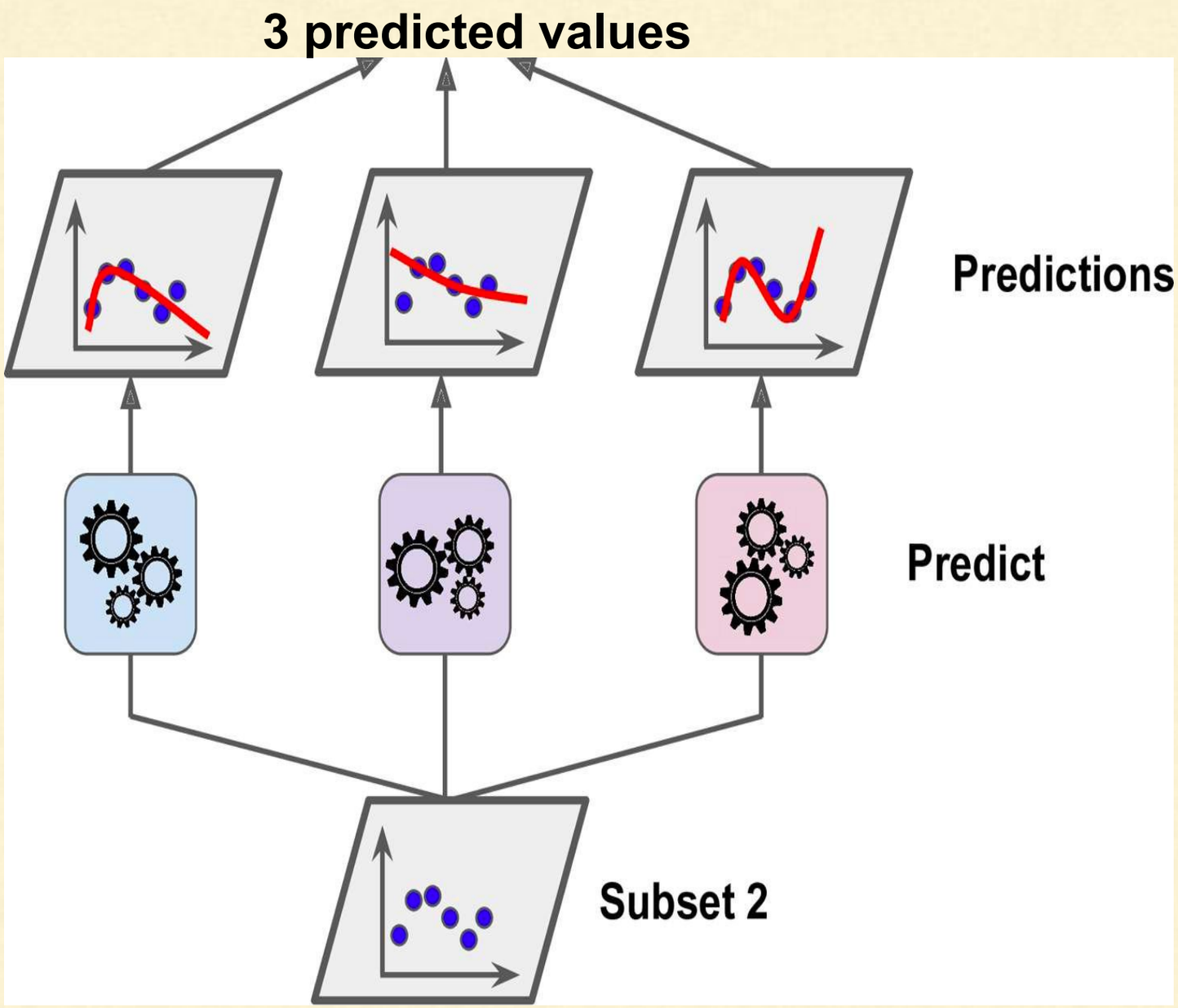
How do we actually train the blender ??

Gradient Boosting - Stacking



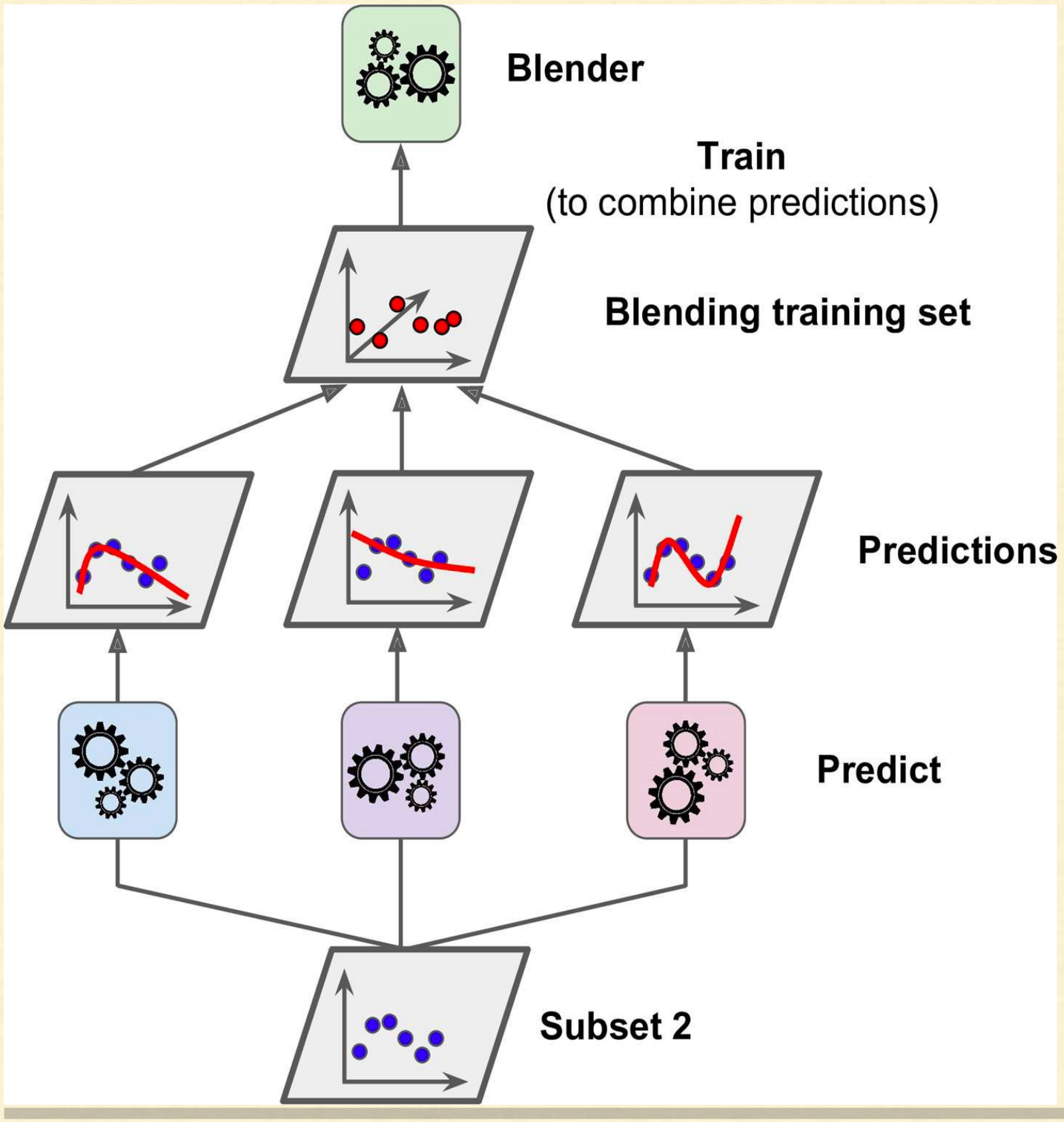
- First, the training set is split in two subsets.
- The first subset is used to train the predictors in the first layer

Gradient Boosting - Stacking



- Next, the first layer predictors are used to make predictions on the second (held-out) set.
- 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.

Gradient Boosting - Stacking



- 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 training set, so it learns to predict the target value given the first layer's predictions.

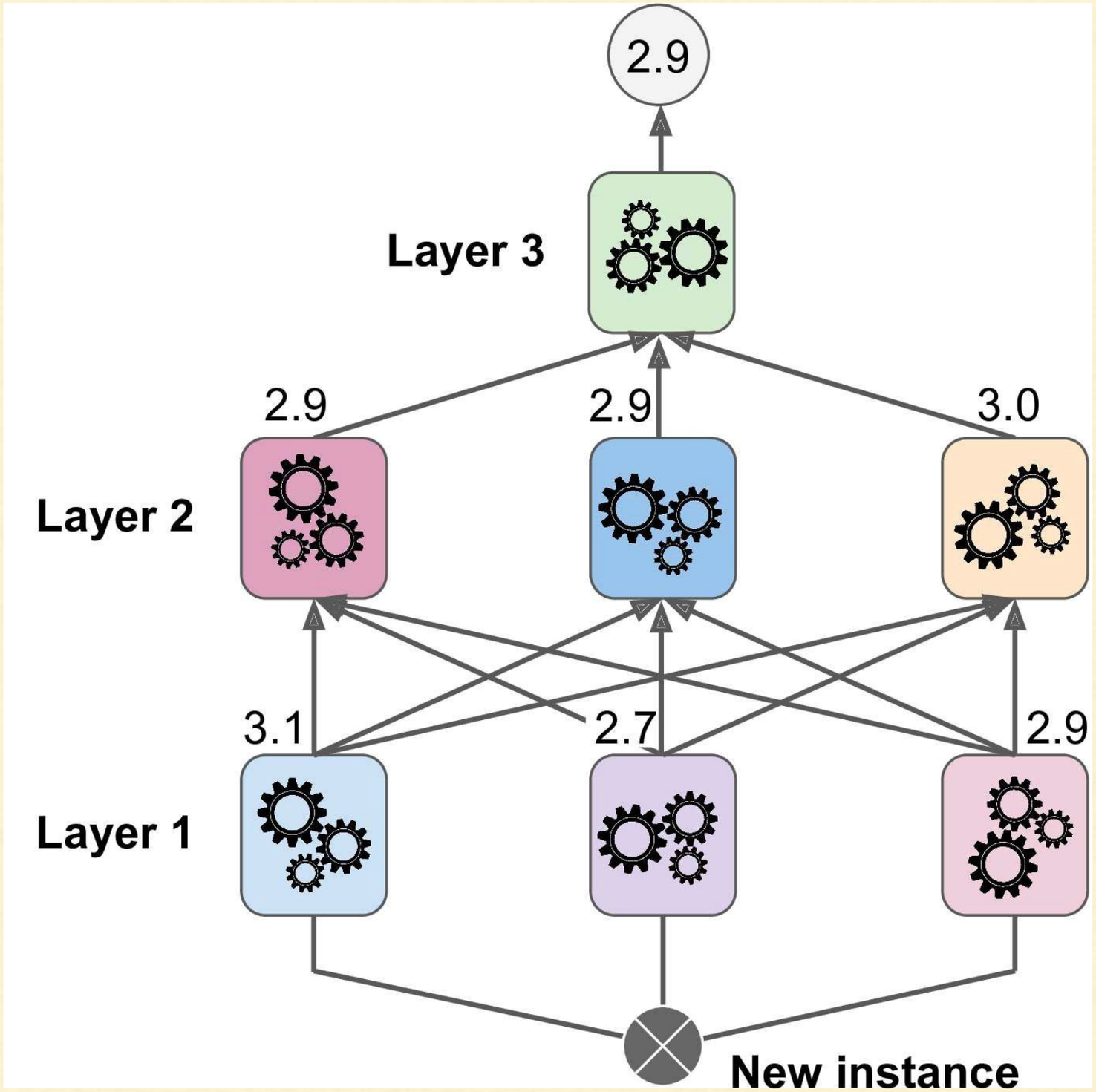
Gradient Boosting - Stacking

It is actually possible to train several different blenders this way
(e.g., one using Linear Regression, another using Random Forest Regression, and so on): we get a whole layer of blenders.

The trick is to split the training set into three subsets:

- The first one is used to train the first layer,
- The second one is used to create the training set used to train the second layer (using predictions made by the predictors of the first layer on the second data)
- and the third one is used to create the training set to train the third layer (using predictions made by the predictors of the second layer).
- Once this is done, we can make a prediction for a new instance by
- going through each layer sequentially,

Gradient Boosting - Stacking



XGBoost - Introduction

- Short for E**x**treme **G**radient **B**oosting
- Belongs to a family of **boosting** algorithms
  - that convert weak learners into strong learners.
- Optimized distributed gradient boosting library
- Used for supervised learning problems
- Uses gradient boosting (GBM) framework at core
- Inception (early 2014),
- True Love of kaggle users
- Created by Tianqi Chen, PhD Student, Univ of Washington.

XGBoost - Features

- **Enabled Parallel Computing (OpenMP):**
  - By default, uses all the cores of your laptop/machine
- **Has Regularization:**
  - Biggest advantage of xgboost.
  - GBM has no provision for regularization.
- **Enabled Cross Validation:**
  - Enabled with internal CV function
- **Missing Values:**
  - XGBoost is designed to handle missing values internally. The missing values are treated in such a manner that if there exists any trend in missing values, it is captured by the model.

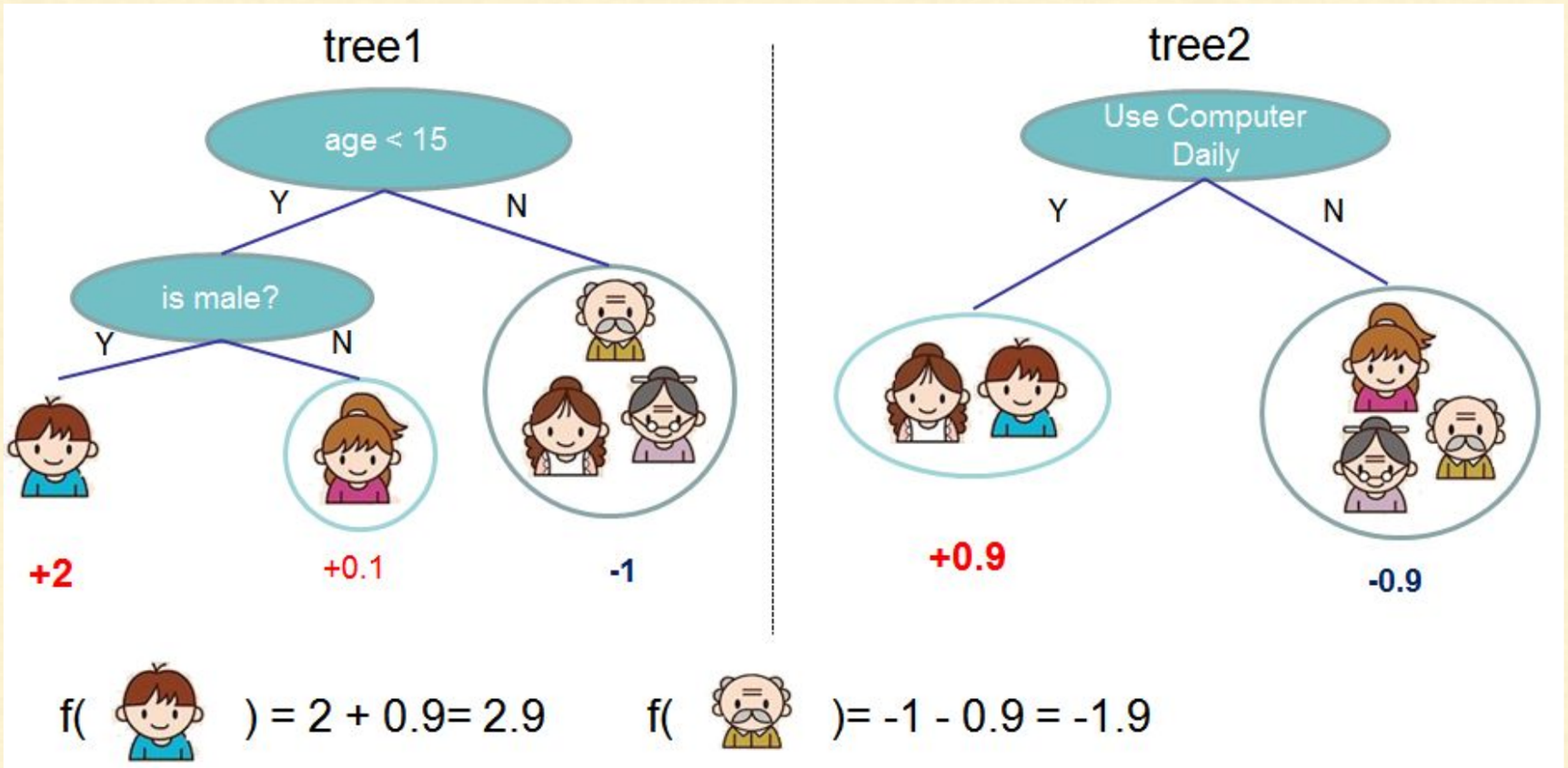
XGBoost - Features

- **Flexibility:**
  - Regression, classification, and ranking problems,
  - Supports user-defined objective functions
  - Supports user defined evaluation metrics
- **Availability:**
  - Available in R, Python, Java, Julia, and Scala.
- **Save and Reload:**
  - Feature to save its data matrix and model and reload it later

XGBoost - Getting Started

- **Lets take look in jupyter notebook**

XGBoost - Theory



The objective function optimizes trees the way we optimize weights usually

$$\hat{y}_i = \sum_{k=1}^{K} f_k(x_i), f_k \in \mathcal{F}$$

XGBoost - Theory

XGBoost adds a heavy normalization term

$$\text{obj}(\theta) = \sum_{i}^{n} l(y_i, \hat{y}_i) + \sum_{k=1}^{K} \Omega(f_k)$$

The loss function l could be anything.
Where predicted y is a function of all trees fk.

$$\hat{y}_i = \sum_{k=1}^{K} f_k(x_i), f_k \in \mathcal{F}$$

$$\Omega(f) = \gamma T + \frac{1}{2}\lambda \sum_{j=1}^{T} w_j^2$$

K - total number of trees
T - total number of leafs
Wj is the weight of each leaf

XGBoost - Theory

XGBoost adds a heavy normalization term

$$\text{obj}(\theta) = \sum_{i}^{n} l(y_i, \hat{y}_i) + \sum_{k=1}^{K} \Omega(f_k)$$

The loss function l could be anything.
Where predicted y is a function of all trees fk.

$$\hat{y}_i = \sum_{k=1}^{K} f_k(x_i), f_k \in \mathcal{F}$$

$$\Omega(f) = \gamma T + \frac{1}{2}\lambda \sum_{j=1}^{T} w_j^2$$

K - total number of trees
T - total number of leafs
Wj is the weight of each leaf

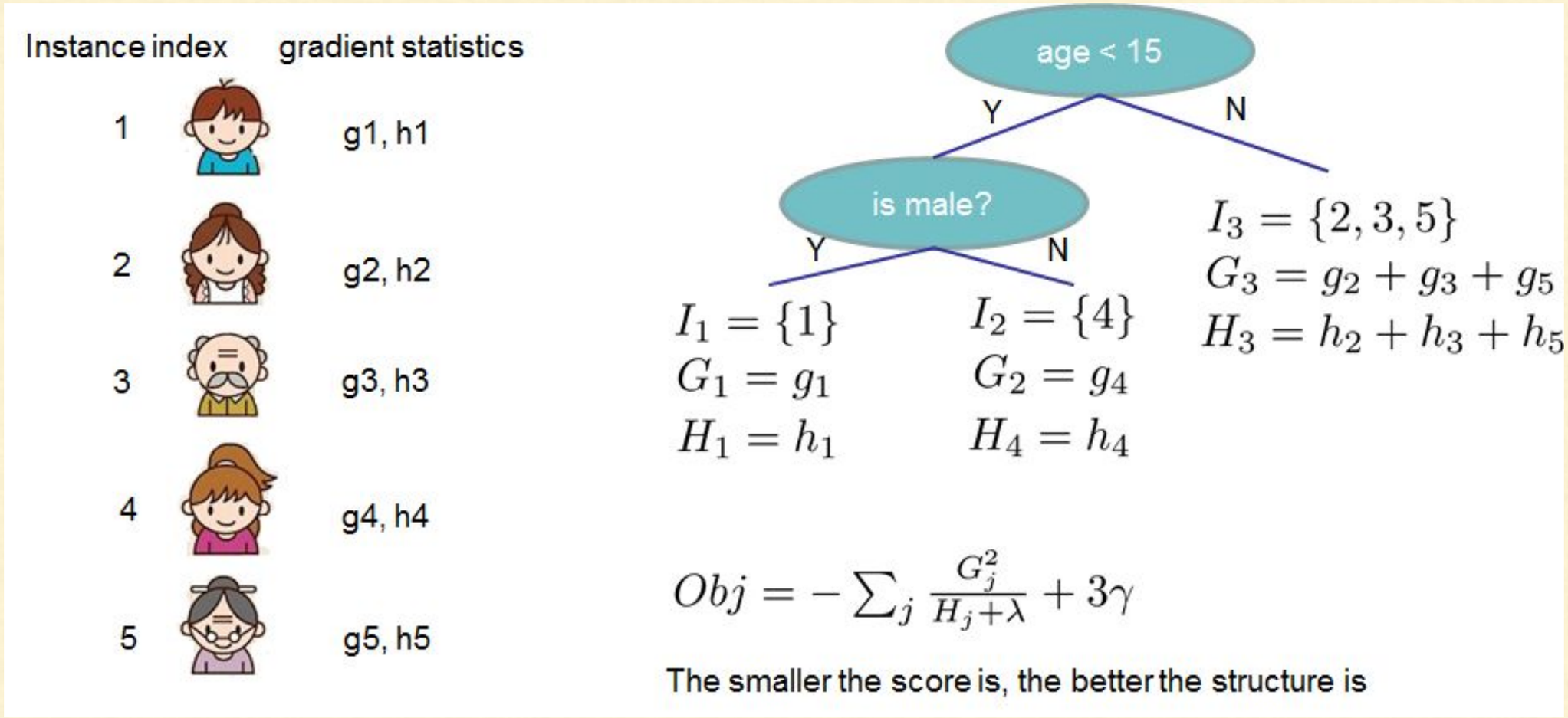
$$\text{obj}^* = -\frac{1}{2} \sum_{j=1}^{T} \frac{G_j^2}{H_j + \lambda} + \gamma T$$

Measures how good a tree structure

- Gi is sum of gi which is per leaf
  - First derivative of loss function
- Hi is sum of hi which is per leaf
  - Second derivative of loss function

See Derivation

XGBoost - Theory



XGBoost - Notes on Parameter Tuning

- Parameter tuning is a dark art in machine learning.
- The optimal parameters of a model can depend on many scenarios.
- So it is impossible to create a comprehensive guide for doing so.

XGBoost - Notes on Parameter Tuning

Understanding Bias-Variance Tradeoff

Most parameters in xgboost are about bias variance tradeoff

	Bias	*Variance*	
Simple Model			**Complicated Model (e.g. More Depth)**
Less ability to fit data - Underfit			**Better ability to fit data - Overfit**
Performs bad on training & testing dataset			**Performs training not testing dataset**

When we allow the model to get **more complicated** (e.g. more depth), the model has **better ability to fit** the training data, resulting in a **less biased** model. However, such complicated model requires more data to fit.

XGBoost - Notes on Parameter Tuning

Control Overfitting

Two ways that you can control overfitting in xgboost:

1. Directly control model complexity
    - Using **max_depth**, **min_child_weight** and **gamma**
2. Add randomness to make training robust to noise
    - This include **subsample**, **colsample_bytree**
    - You can also reduce stepsize **eta**, but needs to remember to increase **num_round** when you do so.

XGBoost - Parameter Tuning

Parameters have been divided into 3 categories:

- **General Parameters:**
  - Guide the overall functioning
- **Booster Parameters:**
  - Guide the individual booster (tree/regression) at each step
  - We are going to use only tree type of booster. Linear is hardly used
- **Learning Task Parameters:**
  - Guide the optimization performed

XGBoost - **General Parameters**

These define the overall functionality of XGBoost.

booster [default=gbtree]
Type of model to run at each iteration. 2 options:

i. **gbtree**: tree-based models, (Going to focus on this)
ii. gblinear: linear models

silent [default=0]:
Generally good to keep it 0 as the messages might help in understanding the model.

nthread

- Default to maximum number of threads available if not set]
- This is used for parallel processing and number of cores in the system should be entered
- If you wish to run on all cores, value should not be entered and algorithm will detect automatically

XGBoost - Tree Booster Parameters

1. **learning_rate [default=0.3]**
   - Makes the model more robust by shrinking the weights on each step
   - Typical final values to be used: 0.01-0.3
2. **min_child_weight [default=1]**
   - Defines the minimum sum of weights of all observations required in a child.
   - This is similar to **min_child_leaf** in GBM but not exactly. This refers to min "sum of weights" of observations while GBM has min "number of observations".
   - Used to control over-fitting. Higher values prevent a model from learning relations which might be highly specific to the particular sample selected for a tree.
   - Too high values can lead to under-fitting hence, it should be tuned using CV.

XGBoost - Tree Booster Parameters

3. gamma [default=0]

- A node is split only when the resulting split gives a positive reduction in the loss function. Gamma specifies the minimum loss reduction required to make a split.
- Makes the algorithm conservative. The values can vary depending on the loss function and should be tuned.

4. max_delta_step [default=0]

- In maximum delta step we allow each tree's weight estimation to be. If the value is set to 0, it means there is no constraint. If it is set to a positive value, it can help making the update step more conservative.
- Usually this parameter is not needed, but it might help in logistic regression when class is extremely imbalanced.
- This is generally not used but you can explore further if you wish.

XGBoost - Tree Booster Parameters

5. subsample [default=1]

- Same as the subsample of GBM. Denotes the fraction of observations to be randomly samples for each tree.
- Lower values make the algorithm more conservative and prevents overfitting but too small values might lead to under-fitting.
- Typical values: 0.5-1

6. colsample_bytree [default=1]

- Similar to max_features in GBM. Denotes the fraction of columns to be randomly samples for each tree.
- Typical values: 0.5-1

7. colsample_bylevel [default=1]

- Denotes the subsample ratio of columns for each split, in each level.
- I don't use this often because subsample and colsample_bytree will do the job for you. but you can explore further if you feel so.

XGBoost - Tree Booster Parameters

8. reg_lambda [default=1]

- L2 regularization term on weights (analogous to Ridge regression)
- This used to handle the regularization part of XGBoost. Though many data scientists don't use it often, it should be explored to reduce overfitting.

9. reg_alpha [default=0]

- L1 regularization term on weight (analogous to Lasso regression)
- Can be used in case of very high dimensionality so that the algorithm runs faster when implemented

10. scale_pos_weight [default=1]

- A value greater than 0 should be used in case of high class imbalance as it helps in faster convergence.

XGBoost - Learning Task Parameters

Define the optimization objective the metric to be calculated at each step.

1. **objective [default=reg:linear]**
    - This defines the loss function to be minimized. Mostly used values are:
        - **binary:logistic** –logistic regression for binary classification, returns predicted probability (not class)
        - **multi:softmax** –multiclass classification using the softmax objective, returns predicted class (not probabilities)
            - you also need to set an additional num_class (number of classes) parameter defining the number of unique classes
        - **multi:softprob** –same as softmax, but returns predicted probability of each data point belonging to each class.

XGBoost - Learning Task Parameters

Define the optimization objective the metric to be calculated at each step.

2. eval_metric [default according to objective]

- The metric to be used for validation data.
- The default values are rmse for regression and error for classification.
- Typical values are:
  - rmse – root mean square error
  - mae – mean absolute error
  - logloss – negative log-likelihood
  - error – Binary classification error rate (0.5 threshold)
  - merror – Multiclass classification error rate
  - mlogloss – Multiclass logloss
  - auc: Area under the curve

- **seed [default=0]**
  - The random number seed.
  - Can be used for generating reproducible results and also for parameter tuning.

XGBoost - Tree Booster Parameters

```
from sklearn.model_selection import GridSearchCV

param_grid = [
    {'learning_rate': [0.1, 0.3],
     'min_child_weight': [0.5, 2], 'gamma': [0, 0.2],
     'max_delta_step': [0], 'subsample': [1],
     'colsample_bytree': [1], 'colsample_bylevel':[1],
     'scale_pos_weight': [1]},]

xgbc_grid = XGBClassifier()
grid_search = GridSearchCV(xgbc_grid, param_grid, cv=5,
                           scoring='neg_mean_squared_error')
grid_search.fit(X_train, y_train)
```

Please See Notebook

XGBoost - Notes from hear-and-there

```
#brute force scan for all parameters, here are the tricks
#usually max_depth is 6,7,8
#learning rate is around 0.05, but small changes may make big diff
#tuning min_child_weight subsample colsample_bytree can have
#much fun of fighting against overfit
#n_estimators is how many round of boosting
#finally, ensemble xgboost with multiple seeds may reduce variance
parameters = {'nthread':[4], #when use hyperthread, xgboost may become slower
              'objective':['binary:logistic'],
              'learning_rate': [0.05], #so called `eta` value
              'max_depth': [6],
              'min_child_weight': [11],
              'silent': [1],
              'subsample': [0.8],
              'colsample_bytree': [0.7],
              'n_estimators': [5], #number of trees, change it to 1000 for better results
              'missing':[-999],
              'seed': [1337]}

clf = GridSearchCV(xgb_model, parameters, n_jobs=5,
                   cv=StratifiedKFold(train['QuoteConversion_Flag'], n_folds=5, shuffle=True),
                   scoring='roc_auc',
                   verbose=2, refit=True)
```

https://www.kaggle.com/phunter/xgboost-with-gridsearchcv

XGBoost

To Learn more:

- ***Main Website***
  - ***http://xgboost.readthedocs.io/en/latest/***
- Introduction to Boosted Trees
  - ***http://xgboost.readthedocs.io/en/latest/model.html***
- Distributed XGBoost YARN on AWS
  - ***http://xgboost.readthedocs.io/en/latest/tutorials/aws_yarn.html***

Questions?

https://discuss.cloudxlab.com

reachus@cloudxlab.com

Thank You

reachus@cloudxlab.com