Ensemble Learning and Random Forests

- **Ensemble** is A group of predictors
- **Ensemble Learning** is aggregating the predictions of a group of predictors and select the most voted prediction
- **Ensemble method** is an Ensemble Learning algorithm

Voting Classifiers

- **hard voting classifier** aggregates the predictions of each classifier: the class that gets the most votes is the ensemble's prediction.

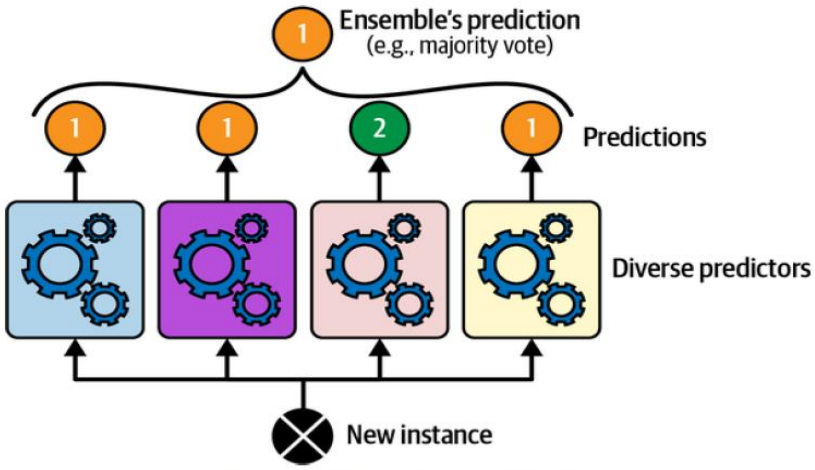


Figure 7-2. Hard voting classifier predictions

- 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 in the ensemble, and they are sufficiently diverse.
- **The law of large numbers** is a theorem that describes the result of performing the same experiment a large number of times. According to the law, the average of the results obtained from a large number of trials should be close to the expected value and tends to become closer to the expected value as more trials are performed.

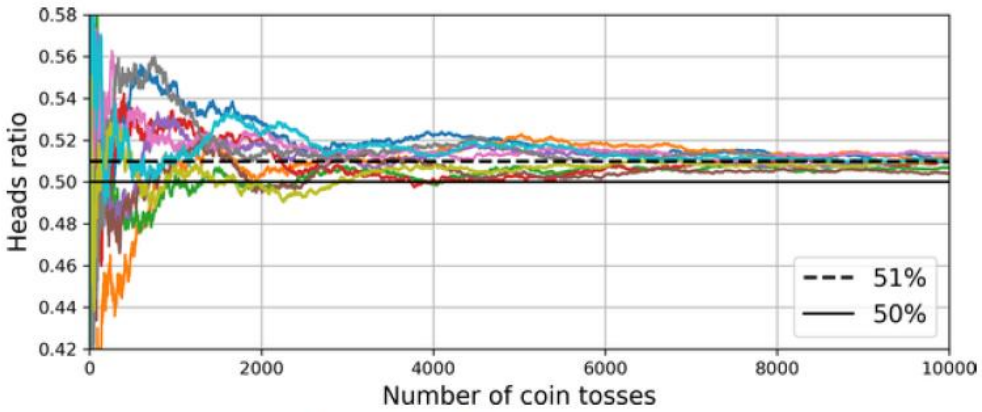


Figure 7-3. The law of large numbers

- Ensemble methods works best when the predictors are independent from one another as possible, like training them using very different algorithms
- Scikit-learn provides a **VotingClassifier** class, takes a list of name/predictor pairs, and use it like a normal classifier
- When you fit a **VotingClassifier**, it clones every estimator and fits the clones. The original estimators are available via the **estimators** attribute, while the fitted clones are available via the **estimators_** attribute. If you prefer a dict rather than a list, you can use **named_estimators** or **named_estimators_**

```python
from sklearn.datasets import make_moons
from sklearn.ensemble import RandomForestClassifier, VotingClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.svm import SVC

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)

voting_clf = VotingClassifier(
    estimators=[
        ('lr', LogisticRegression(random_state=42)),
        ('rf', RandomForestClassifier(random_state=42)),
        ('svc', SVC(random_state=42))
    ]
)
voting_clf.fit(X_train, y_train)
```

Using 3 diverse classifiers

Hard voting

- Each classifier's accuracy on the test set

```
>>> for name, clf in voting_clf.named_estimators_.items():
...     print(name, "=", clf.score(X_test, y_test))
...
lr = 0.864
rf = 0.896
svc = 0.896
```

- Performance of the voting classifier on the test set, The voting classifier outperforms all the individual classifiers

```
>>> voting_clf.score(X_test, y_test)
0.912
```

- How to apply Soft Voting?
  - Make sure all classifiers are able to estimate class probabilities(have a **predict_proba()** method), for SVC class you need to set its **probability** hyperparameter to **True**
  - Replace **voting= "hard"** to **voting= "soft"**
- How Soft Voting works?
  - Soft voting ensembles calculates the average predicted probabilities(or scores) and compares it to a threshold value.
- Do we always use mean in Soft voting?
  - it's possible to use the **median** instead of the mean, as it's less sensitive to outliers, so it will usually represent the underlying set of outputs better than the mean.
  - that doesn't imply that the median is always a better choice.

```
>>> voting_clf.voting = "soft"
>>> voting_clf.named_estimators["svc"].probability = True
>>> voting_clf.fit(X_train, y_train)
>>> voting_clf.score(X_test, y_test)
0.92
```

We reach 92% accuracy by simply using soft voting

Bagging and Pasting

- Another way to get a diverse set of classifiers is to use the same training algorithm for every predictor and train them on different random subsets of the training set.
- What is the difference between Bagging and Pasting?

- Both Bagging and pasting allow training instances to be sampled several times across multiple predictors
- Bagging(**bootstrap aggregating)**, the sampling is performed with **replacement**; allows the training samples to be sampled several times for the same predictor.
- Pasting, sampling is performed without **replacement**; the training samples can't appear more than one time in a single sample
- Bagging is preferred in case of small datasets, however, pasting is preferred in case of large datasets

Not Sure

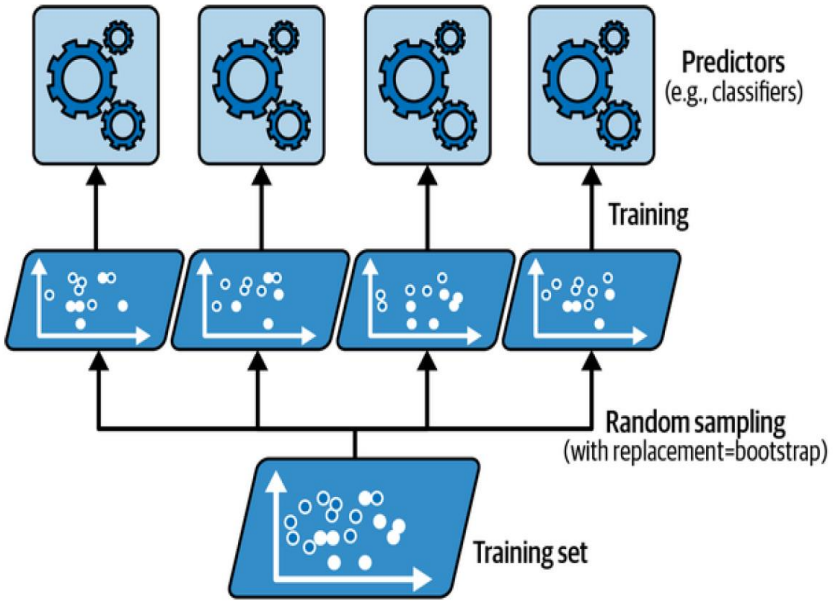


Figure 7-4. Bagging and pasting involve training several predictors on different random samples of the training set

- 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 (the most frequent prediction) or the average for regression.
- Each individual predictor has a higher **bias** than if it were trained on the original training set, but aggregation reduces both **bias** and **variance**.
- Generally, the result is that the ensemble has a similar **bias** but a lower **variance** than a single predictor trained on the original training set.
- predictors can all be trained in parallel, via different CPU cores or even different servers.
- predictions can be made in parallel too.
- This is one of the reasons **bagging** and **pasting** are such popular methods. They scale very well.

Bagging and pasting in Scikit-Learn

- You can implement both bagging and pasting with the **BaggingClassifier** class or **BaggingRegressor** for regression
- **Bootstrap=True** applies bagging, **bootstrap=False** applies pasting
- **n_jobs** parameter tells scikit-learn the no. of CPU core to use for training and predictions(-1 value use all cores)

```
from sklearn.ensemble import BaggingClassifier
from sklearn.tree import DecisionTreeClassifier

bag_clf = BaggingClassifier(DecisionTreeClassifier(), n_estimators=500,
                            max_samples=100, n_jobs=-1, random_state=42)
bag_clf.fit(X_train, y_train)
```

- A BaggingClassifier automatically performs soft voting if the base classifier can estimate class probabilities

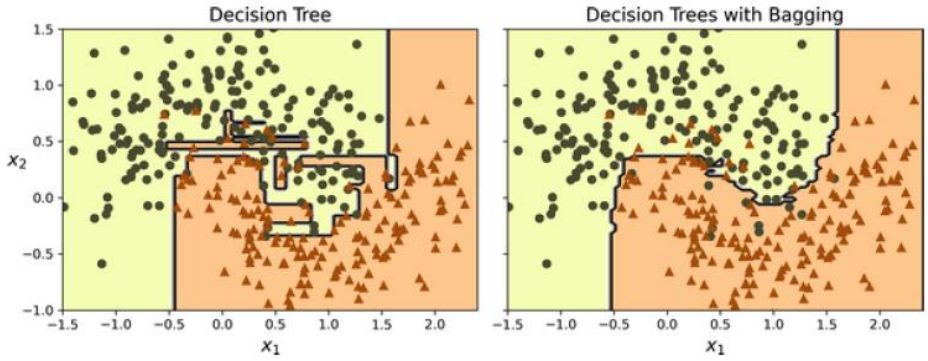


Figure 7-5. A single decision tree (left) versus a bagging ensemble of 500 trees (right)

- 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)
- Bagging introduces a bit more diversity in the subsets, so bagging ends up with a slightly higher bias than pasting; but the extra diversity also means that the predictors end up being less correlated, so the ensemble's variance is reduced.
- Overall, bagging often results in better models, which explains why it's generally preferred.

Out-of-Bag Evaluation

- With bagging, some training instances may be sampled several times for any given predictor, while others may not be sampled at all.
- In an ideal case 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,** they are not the same instances for all predictors.

If there are N rows in the training data set. Then, the probability of not picking a row in a random draw is

$$\frac{N-1}{N}$$

Using sampling-with-replacement the probability of not picking N rows in random draws is

$$\left(\frac{N-1}{N}\right)^N$$

which in the limit of large N becomes equal to

$$\lim_{N\to\infty}\left(1-\frac{1}{N}\right)^N = e^{-1} = 0.368$$

- A bagging ensemble can be evaluated using OOB instances, without the need for a separate validation set.
- you can set **oob_score=True** when creating a BaggingClassifier to request an automatic OOB evaluation after training.
- You can access the evaluation score result through **oob_score_** attribute

```
>>> bag_clf = BaggingClassifier(DecisionTreeClassifier(), n_estimators=500,
...                             oob_score=True, n_jobs=-1, random_state=42)
...
>>> bag_clf.fit(X_train, y_train)
>>> bag_clf.oob_score_
0.896
```

The OOB Score

```
>>> from sklearn.metrics import accuracy_score
>>> y_pred = bag_clf.predict(X_test)
>>> accuracy_score(y_test, y_pred)
0.92
```

The test set score

OOB evaluation was a bit too pessimistic, just over 2% too low.

- You can access the OOB decision function through the **oob_decision_function_** attribute, the decision function returns the class probabilities for each training instance(Since the base estimator has a predict_proba() method)

```
>>> bag_clf.oob_decision_function_[:3]  # probas for the first 3 instances
array([[0.32352941, 0.67647059],

       [0.3375    , 0.6625    ],
       [1.        , 0.        ]])
```

Random Patches and Random Subspaces

- The BaggingClassifier class supports sampling the features as well. Sampling Features is controlled by two hyperparameters: **max_features** and **bootstrap_features**.
- Each predictor will be trained on a random subset of the input features.
- This technique is particularly useful when you are dealing with high-dimensional inputs (such as images), as it can considerably speed up training.
- **random patches** method is sampling both training instances and features
- **random subspaces** method keeps all training instances(by setting **bootstrap=False** and **max_samples=1.0**) but sampling features (by setting **bootstrap_features** to **True** and/or **max_features** to a value smaller than **1.0**)
- Sampling features results in even more predictor diversity, trading a bit more bias for a lower variance.

Random Forests

- **Random Forests** is an ensemble of Decision Trees generally trained via the bagging method (or sometimes pasting)
- **RandomForestClassifier** class, which is more convenient and optimized for decision trees (similarly, there is a **RandomForestRegressor** class for regression tasks)
- The **BaggingClassifier** class remains useful if you want a bag of something other than decision trees.

```
from sklearn.ensemble import RandomForestClassifier

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)
```

Produce roughly equivalent models

```
bag_clf = BaggingClassifier(
    DecisionTreeClassifier(max_features="sqrt", max_leaf_nodes=16),
    n_estimators=500, n_jobs=-1, random_state=42)
```

- a **RandomForestClassifier** has all the hyperparameters of a **DecisionTreeClassifier** (to control how trees are grown),
  - except **splitter** is absent(forced to **"random"**), **presort** is absent(forced to **False**), **max_samples** is absent(forced to **1.0**), and **base_estimator** is absent(forced to **DecisionTreeClassifier** with the provided hyperparameters
- a **RandomForestClassifier** has all the hyperparameters of a **BaggingClassifier** to control the ensemble itself.
- 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. By default, it samples **√n** features (where n is the total number of features).
- The algorithm results in greater tree diversity, which (again) trades a higher bias for a lower variance, generally yielding an overall better model.

Extra-Trees

- **Extremely Randomized Trees(Extra-Trees)** is a way to make trees even more random by also using random thresholds for each feature rather than searching for the best possible thresholds.
- this technique trades more bias for a lower variance.
- It also makes **extra-trees** classifiers much faster to train than regular random forests, because finding the best possible threshold for each feature at every node is one of the most **time-consuming tasks** of growing a tree
- You can create an extra-trees classifier using Scikit-Learn's **ExtraTreesClassifier** class. It is identical to the **RandomForestClassifier** class, except **bootstrap** defaults to **False**. Similarly, the **ExtraTreesRegressor** class
- You can't tell in advance whether a **RandomForestClassifier** will perform better or worse than an **ExtraTreesClassifier**

Feature Importance

- **Random Forests** can 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.**
- you can access it through **feature_importances_** attribute

```
>>> from sklearn.datasets import load_iris
>>> iris = load_iris(as_frame=True)
>>> rnd_clf = RandomForestClassifier(n_estimators=500, random_state=42)
>>> rnd_clf.fit(iris.data, iris.target)
>>> for score, name in zip(rnd_clf.feature_importances_, iris.data.columns):
...     print(round(score, 2), name)
...
0.11 sepal length (cm)
0.02 sepal width (cm)
0.44 petal length (cm)
0.42 petal width (cm)
```

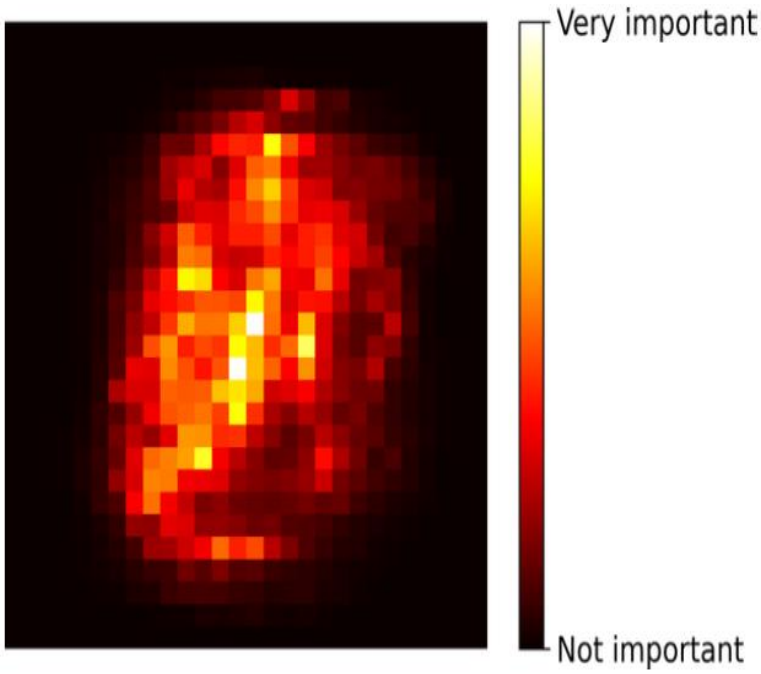


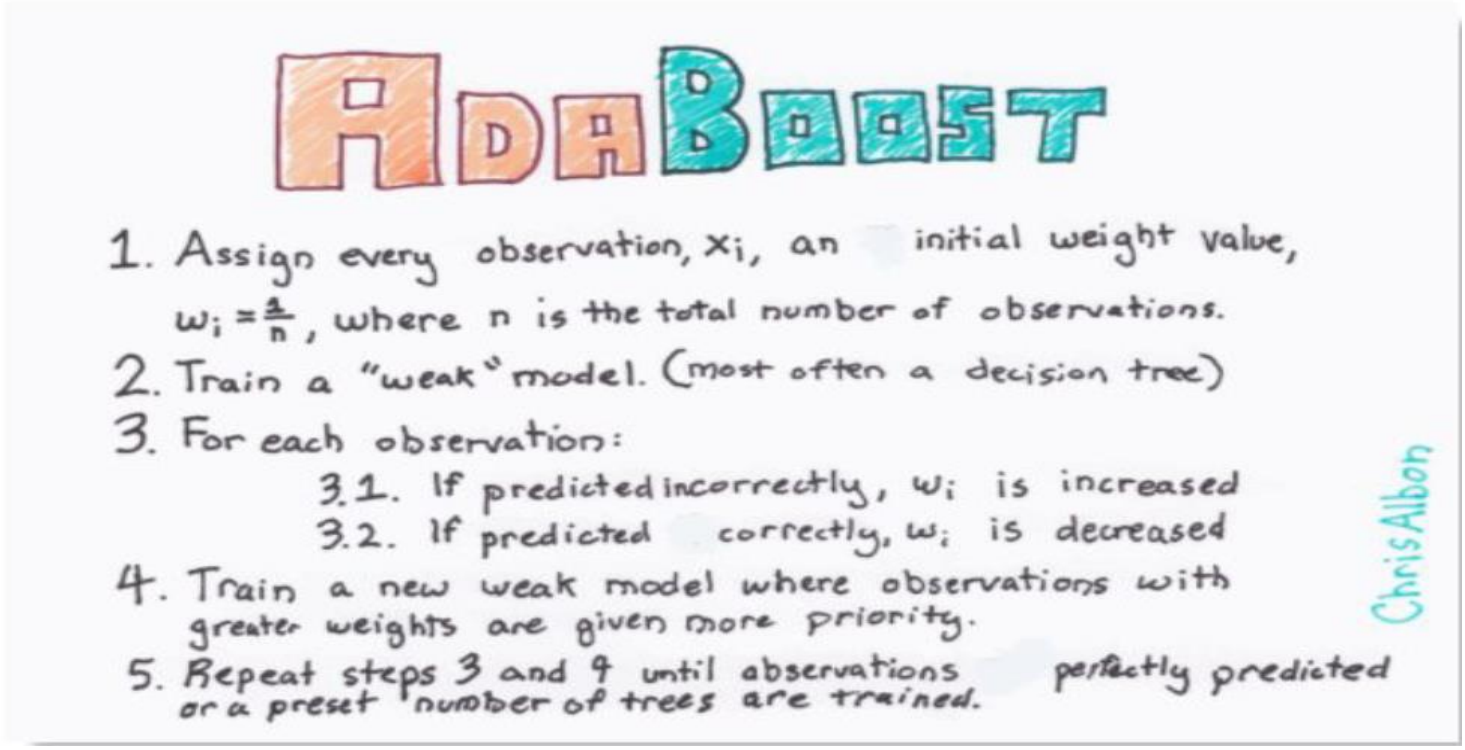
Figure 7-6. MNIST pixel importance (according to a random forest classifier)

- Random forests are very handy to get a quick understanding of what features actually matter, in particular if you need to perform **feature selection**.

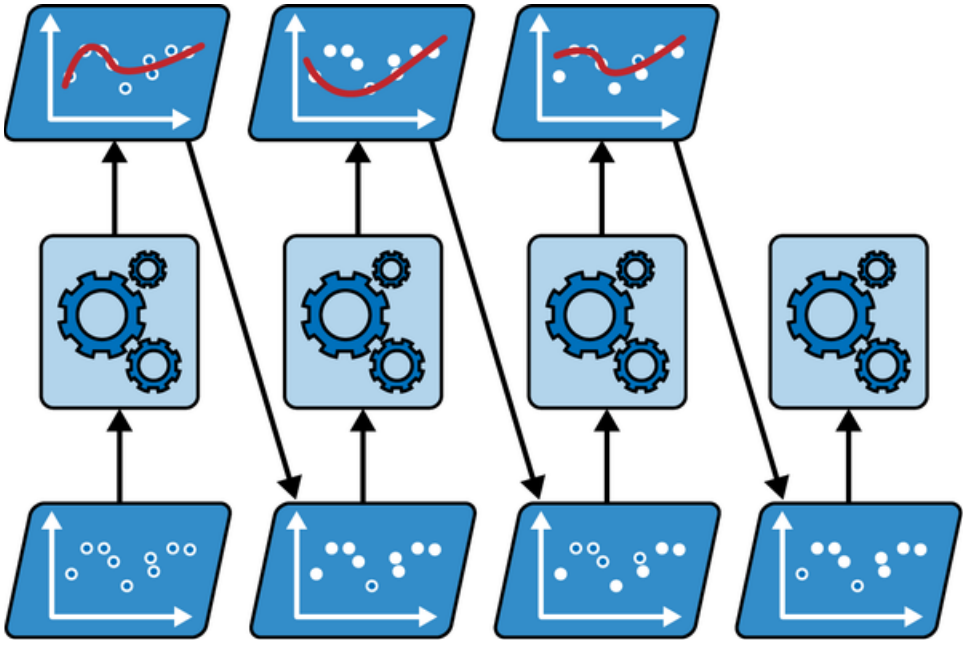
Boosting

- **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**.
- The most popular boosting methods are AdaBoost (short for adaptive boosting) and gradient boosting.

AdaBoost(Adaptive Boosting)



- **AdaBoost** is a technique that works by letting a new predictor to correct its predecessor by paying a bit more attention to the training instances that the predecessor underfitted. This results in new predictors focusing more on the hard cases.
- **The algorithm** first trains a base classifier (such as a decision tree) and uses it to make predictions on the training set. **The algorithm** then increases the relative weight of misclassified training instances. Then it trains a second classifier, using the updated weights, and again makes predictions on the training set, updates the instance weights, and so on.



- **Fig 7-8,** 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).

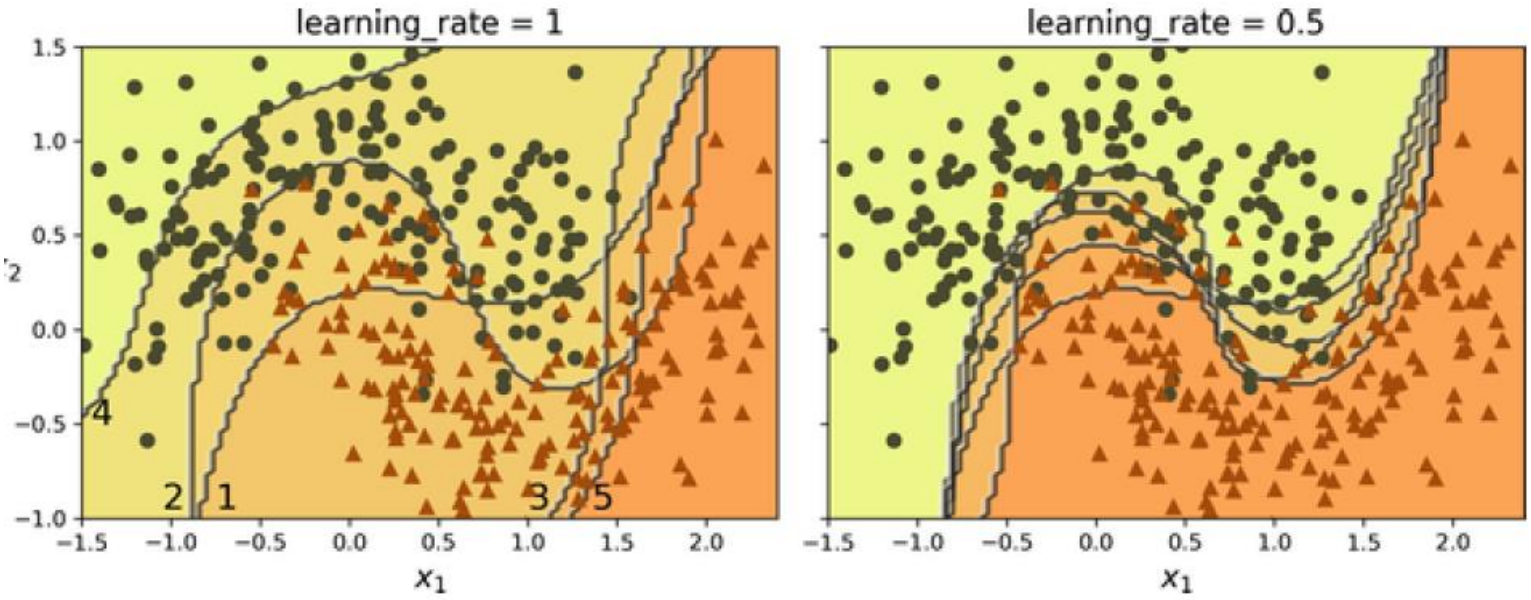
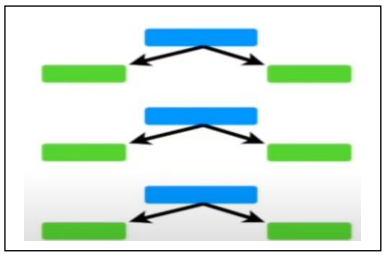


Figure 7-8. Decision boundaries of consecutive predictors

- 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.
- 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.
- In AdaBoost(sequential learning), Training cannot be parallelized 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.
- Scikit-Learn uses a multiclass version of AdaBoost called SAMME, When there are just two classes, SAMME is equivalent to AdaBoost.
- If the predictors can estimate class probabilities, Scikit-Learn can use a variant of SAMME called SAMME.R, which relies on class probabilities rather than predictions and generally performs better.

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

```
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)
```



AdaBoost Explained by StatQuest

- The common way to use AdaBoost is with Decision trees
- A Forest of Trees made with AdaBoost are usually just a node and Two leaves
- A tree with just one node and two leaves is called a **stump, AdaBoost** combines a lot stumps to make classifications
- Stumps are not great at making accurate classifications, They can only use one Variable to make a decision, They are technically "**Weak Learners**"
- In a **Forest of Stumps,** made with **AdaBoost,** some stumps get more say in the final classification than others.
- In a **Forest of Stumps,** made with **AdaBoost,** order is important; the errors the first stump makes, influences how the second stump is made, etc.

Building a Stump with Gini index

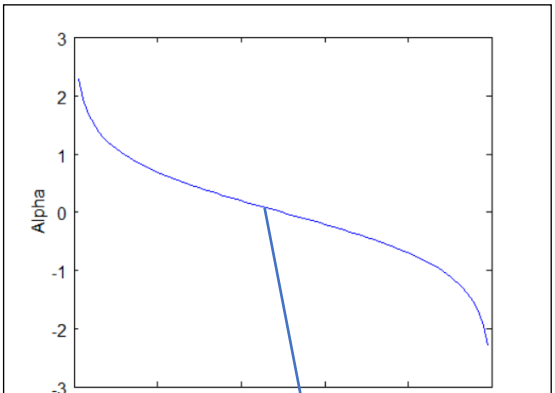
- First, We give each sample a weight that indicates how important it is to be correctly classified
- At the start, all the samples get the same weight $\mathbf{W^{(i)} = 1/m}$ **(m is total no. of samples)**
- After the First stump is made, These **weights** will change in order to guide how the **next stump** is created
- To make the first stump we need to find the **feature** and the **threshold** that does the best job classifying the samples
- We calculate the **Gini index** for all stumps, and take the **stump** with the lowest **Gini index (First stump)**

Determining the amount of Say for A Stump

- We determine how much say a stump has in the final classification based on how well it classified the samples
- The **Total Error** for a stump is the sum of the weights associated with the **incorrectly** classified samples
- **Total Error** will always be between **0, For a perfect stump,** and **1, for a horrible stump**
- We use the **Total Error** to determine **The amount of Say** this stump has in the final classification

$$\text{Amount of Say} = \frac{1}{2}\log\left(\frac{1 - \text{Total Error}}{\text{Total Error}}\right)$$

- When **Total Error** decreases, **The amount of Say** increases and vice versa
- Now we know how the **sample weights** for the incorrectly classified samples are used to determine **the amount of say** each stump gets.



Updating the Sample Weights

- For **incorrectly classified samples,** we will emphasize the need for the next stump to correctly classify it by **increasing its sample weight.**
- For **correctly classified samples,** we will **decrease its sample weight**

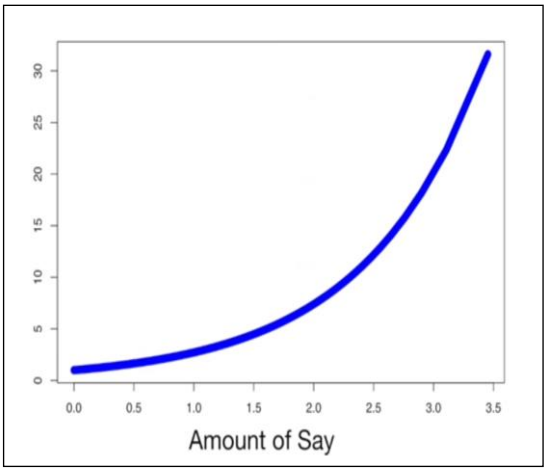
The Blue line tells us the amount of say for total error values between 0 and 1

$$New\ sample\ weight\ =\ old\ weight\ *\ e^{\pm Amount\ of\ say\ (\alpha)}$$

The amount of, say (alpha) will be ***negative*** when the sample is **correctly classified.**

The amount of, say (alpha) will be ***positive*** when the sample is **miss-classified.**

- If the **amount of say** is high, then **the new sample weight** of the incorrectly classified samples will be much larger than before and vice versa.



Normalizing the sample weights

- We need to **Normalize** the **new sample weights**, so that they will add up to **1**.
- To get the **normalized** values, we divide **each new sample weight** by the summation of all **new sample weights**.

Normalize weights (Sum of sample weights = 1)

$$w_i \leftarrow \frac{w_i}{\sum_{i=1}^{N} w_i}$$

The new samples weights add up to 1 (plus or minus a little rounding error).

- Now we can use the **modified sample weights** to make the **second stump** in the forest.

Using the Normalized Weights to make the Second Stump

- Instead of using The **Weighted Gini index,** that would put more emphasis on the **misclassified** sample of the last stump, since this sample has the **largest weight**.
- Alternatively, we can make a **new collection** of samples that contains **duplicate** copies of the samples with the **largest Sample Weights**.
- what the **algorithm** does is select random numbers from **0-1**. Since incorrectly classified records have higher sample weights, the probability of selecting those records is very high.
- With the **new collection of samples**, we give all the samples equal **sample weights** like before.
- Since the **incorrectly classified samples** is present a lot in **the new sample**, the **new stump** will emphasize the need to truly classify these samples(**creating a large penalty for being misclassified**).
- Find the **stump** that does the best job classifying the new collection of samples by finding their **Gini Index** and selecting the one with the **lowest Gini index**.
- Calculate the **Amount of Say** and **Total error** to update the previous sample weights.
- **Normalize** the new sample weights.
- Iterate through these steps until the max no. of stumps is reached or a perfect predictor is found.

Using AdaBoost to make Classifications

- To make predictions, AdaBoost simply computes the predictions of all the predictors and weighs them using the predictor weights **α(amount of say)** . The predicted class(yes/no) is the one that receives the majority of weighted votes.

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

Gradient Boosting

- **gradient Boosting** works by sequentially adding predictors to an ensemble, each one correcting its predecessor. This method tries to fit the new predictor to the residual errors made by the previous predictor.
- **Gradient Boosting** works great with both classification(**gradient boosted classification trees (GBCT)**) and regression tasks(**gradient boosted regression trees (GBRT)**).
- you can train **GBRT** ensembles by using scikit-learn's **GradientBoostingRegressor class**
- Much like the **RandomForestRegressor** class, it has hyperparameters to control the growth of decision trees (e.g., **max_depth**, **min_samples_leaf**)
- Hyperparameters to control the ensemble training, such as the number of trees (**n_estimators**).
- The **learning_rate** hyperparameter scales the contribution of each tree. If you set it to a low value, such as 0.05, 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 following code creates the same ensemble as the one on the right:

```
from sklearn.ensemble import GradientBoostingRegressor

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

First, let's generate a noisy quadratic dataset and fit a DecisionTreeRegressor to it:

```
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 noi

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

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

```
y2 = y - tree_reg1.predict(X)
tree_reg2 = DecisionTreeRegressor(max_depth=2, random_state=43)
tree_reg2.fit(X, y2)
```

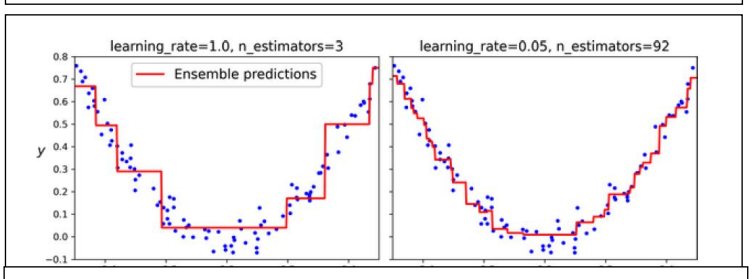
And then we'll train a third regressor on the residual errors made by the second predictor:

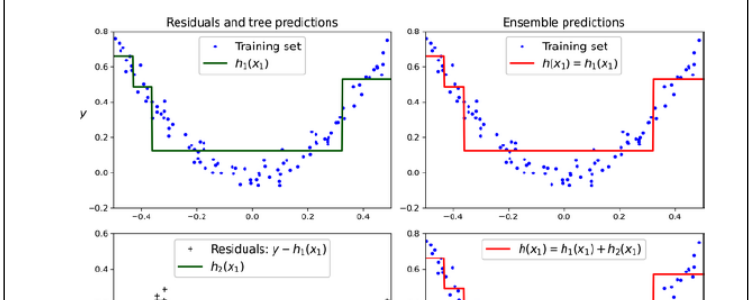
```
y3 = y2 - tree_reg2.predict(X)
tree_reg3 = DecisionTreeRegressor(max_depth=2, random_state=44)
tree_reg3.fit(X, y3)
```

Now we have an ensemble containing three trees. It can make predictions on a new instance simply by adding up the predictions of all the trees:

```
>>> 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])
```

1. Fig 7-10, Shows two GBRT ensembles trained with different hyperparameters: the one on the left does not have enough trees to fit the training set, while the one on the right has about the right amount. If we added more trees, the GBRT would start to overfit the training set.
2. Figure 7-9 represents the predictions of these three trees in the left column, and the ensemble's predictions in the right column. In the first row, the ensemble has just one tree, so its predictions are exactly the same as the first tree's predictions. In the second row, a new tree is trained on the



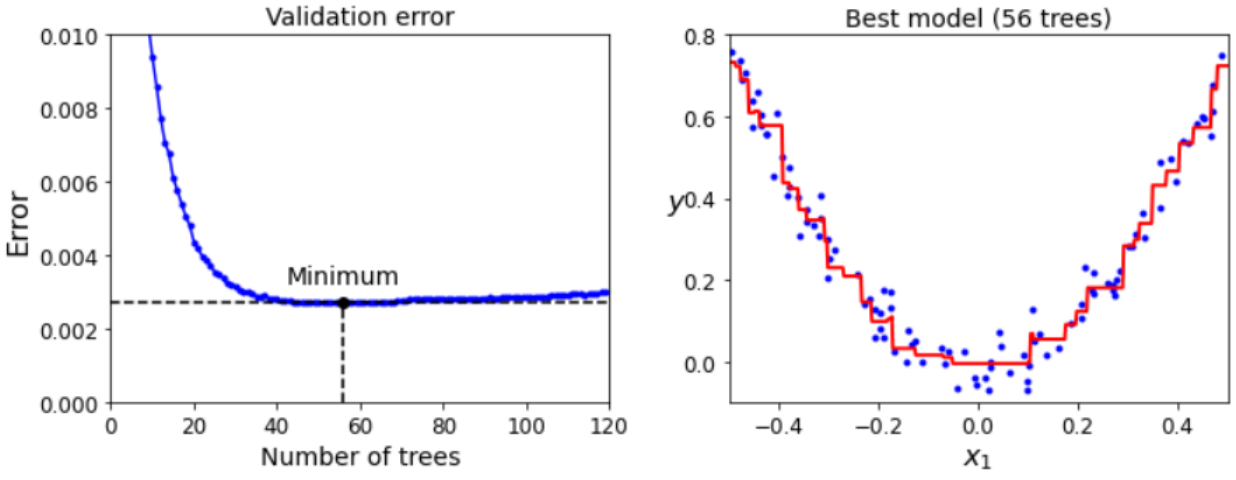


residual errors of the first tree. On the right you can see that the ensemble's predictions are equal to the sum of the predictions of the first two trees. Similarly, in the third row another tree is trained on the residual errors of the second tree. You can see that the ensemble's predictions gradually get better as trees are added to the ensemble.

1. In order to find the optimal no. of trees, you could perform cross-validation using **GridSearchCV** or **RandomizedSearchCV**,
2. You could implement Early Stopping using **staged-predict() method**(it returns an iterator over the predictions made by the ensemble at each stage of training(with one tree, two trees, etc.))

```
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error

X_train, X_val, y_train, y_val = train_test_split(X, y, random_state=49)

gbrt = GradientBoostingRegressor(max_depth=2, n_estimators=120, random_state=42)
gbrt.fit(X_train, y_train)

errors = [mean_squared_error(y_val, y_pred)
          for y_pred in gbrt.staged_predict(X_val)]
bst_n_estimators = np.argmin(errors) + 1

gbrt_best = GradientBoostingRegressor(max_depth=2, n_estimators=bst_n_estimators, random_state=42)
gbrt_best.fit(X_train, y_train)
```

```
GradientBoostingRegressor(max_depth=2, n_estimators=56, random_state=42)
```



3. You could implement **Early Stopping** by actually stopping training early(instead of training a large no. of trees and then find the optimal no.).
    a. by setting **warm_start=True,** which makes scikit-learn keep existing trees when the fit() method is called, allowing incremental learning.
    b. The code stops when the validation error does not improve for no. of iterations in a row.

```
gbrt = GradientBoostingRegressor(max_depth=2, warm_start=True, random_state=42)

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
```

```
print(gbrt.n_estimators)
```

```
61
```

4. You could implement **Early Stopping** by setting **n_iter_no_change** hyperparameter to an integer value, say 10, this will stop the **GBRT** from adding more trees during training if it sees that the last **10** trees didn't help.
    a. If you set **n_iter_no_change** too low, training may stop too early, and the model will underfit. But if you set it too high, it will overfit instead.
    b. When **n_iter_no_change** is set, the **fit()** method automatically splits the training set into a smaller training set and a validation set: this allows it to evaluate the model's performance each time it adds a new tree.
    c. The size of the validation set is controlled by the **validation_fraction** hyperparameter, which is **10%** by default.
    d. The **tol** hyperparameter determines the maximum performance improvement that still counts as negligible. It defaults to **0.0001.**

```
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)
```

- The **GradientBoostingRegressor** class also supports a **subsample** hyperparameter, which specifies the fraction of training instances to be used for training each tree, selected randomly.
- This technique trades a higher bias for a lower variance. It also speeds up training considerably. This is called **stochastic gradient boosting**.
- It is possible to use Gradient Boosting with other cost functions, This is controlled by the **loss** hyperparameter
- Python library **XGBoost,** stands for **Extreme Gradient Boosting,** is an optimized implementation of Gradient Boosting, it aims to be extremely fast, scalable, and portable.

```
xgb_reg = xgboost.XGBRegressor(random_state=42)
xgb_reg.fit(X_train, y_train)
y_pred = xgb_reg.predict(X_val)
val_error = mean_squared_error(y_val, y_pred) # Not shown
print("Validation MSE:", val_error)           # Not shown
```

```
Validation MSE: 0.00400040950714611
```

```python
if xgboost is not None:  # not shown in the book
    xgb_reg.fit(X_train, y_train,
                eval_set=[(X_val, y_val)], early_stopping_rounds=2)
    y_pred = xgb_reg.predict(X_val)
    val_error = mean_squared_error(y_val, y_pred)  # Not shown
    print("Validation MSE:", val_error)            # Not shown
```

```
[0]     validation_0-rmse:0.22834
Will train until validation_0-rmse hasn't improved in 2 rounds.
[1]     validation_0-rmse:0.16224
[2]     validation_0-rmse:0.11843
[3]     validation_0-rmse:0.08760
[4]     validation_0-rmse:0.06848
[5]     validation_0-rmse:0.05709
[6]     validation_0-rmse:0.05297
[7]     validation_0-rmse:0.05129
[8]     validation_0-rmse:0.05155
[9]     validation_0-rmse:0.05211
Stopping. Best iteration:
[7]     validation_0-rmse:0.05129

Validation MSE: 0.0026308690413069744
```

Histogram-Based Gradient Boosting

- **histogram-based gradient boosting (HGB),** another GBRT implementation, optimized for large datasets.
- It works by binning the input features, replacing them with integers. The number of bins is controlled by the **max_bins** hyperparameter, which defaults to **255** and cannot be set any higher than this.
- Binning can greatly reduce the number of possible thresholds that the training algorithm needs to evaluate.
- working with integers makes it possible to use faster and more memory-efficient data structures.
- The way the bins are built removes the need for sorting the features when training each tree.
- This implementation has a computational complexity of **O(b×m)** instead of **O(n×m×log(m)),** where **b** is the number of **bins**, **m** is the **number of training instances**, and **n** is the **number of features**.
- In practice, this means that HGB can train hundreds of times faster than regular GBRT on large datasets.
- However, **binning** causes a **precision loss**, which acts as a **regularizer**: depending on the dataset, this may help reduce overfitting, or it may cause underfitting.
- Scikit-Learn provides two classes for HGB: **HistGradientBoostingRegressor** and **HistGradientBoostingClassifier**.
    - **Early stopping** is automatically activated if the number of instances is greater than **10,000**. You can turn early stopping always on or always off by setting the **early_stopping** hyperparameter to **True** or **False**.
    - **Subsampling** is not supported.
    - **n_estimators** is renamed to **max_iter**.
    - The only decision tree hyperparameters that can be tweaked are **max_leaf_nodes**, **min_samples_leaf**, and **max_depth**.
    - The HGB classes also have two nice features: they support both **categorical features** and **missing values**. This simplifies preprocessing quite a bit.
    - However, the **categorical features** must be represented as integers ranging from **0** to a number lower than **max_bins**. You can use an **OrdinalEncoder** for this.
    - Note that **categorical_features** must be set to the categorical column indices (or a Boolean array).
- **XGBoost**, **CatBoost**, **LightGBM**, and **TensorFlow Random Forests** library are optimized implementations of gradient boosting are available in the Python ML ecosystem,

```
from sklearn.pipeline import make_pipeline
from sklearn.compose import make_column_transformer
from sklearn.ensemble import HistGradientBoostingRegressor
from sklearn.preprocessing import OrdinalEncoder

hgb_reg = make_pipeline(
    make_column_transformer((OrdinalEncoder(), ["ocean_proximity"]),
                            remainder="passthrough"),
    HistGradientBoostingRegressor(categorical_features=[0], random_state=42)
)
hgb_reg.fit(housing, housing_labels)
```

Stacking

- **Stacking** short for **(stacked generalization)** ensemble method.
- instead of using trivial functions (such as hard voting) to aggregate the predictions of all predictors in an ensemble.
- A **blender** or **Meta Learner** (Final predictor) takes the predictions of all predictors as inputs and makes the final predictor.
- To train a **Blender,** a common approach is to use a **Hold-out set,** the training set is split into two subsets.
- The first subset is used to train the predictors in the first layer.
- The first layer's predictors are used to make predictions on the second(**Hold-out set),** ensure the predictions are clean.
- For each instance in the **Hold-out set,** there are three predicted values.
- We create a new training set using these predicted values as input features and keeping the target values of the **hold-out set.**

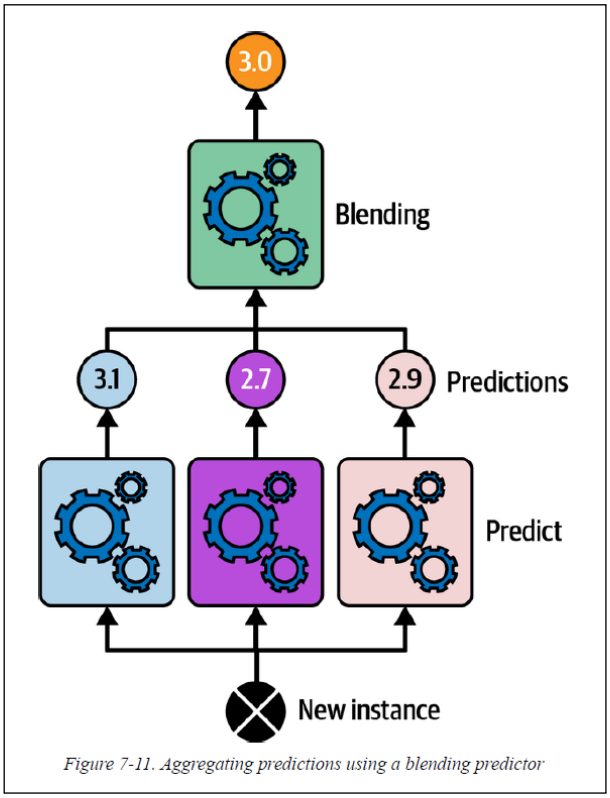
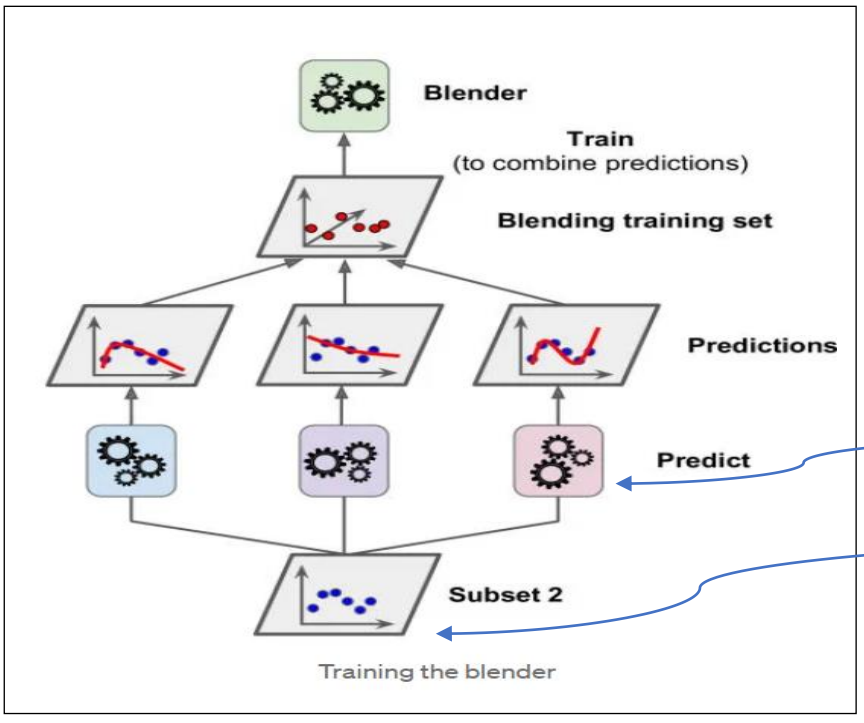
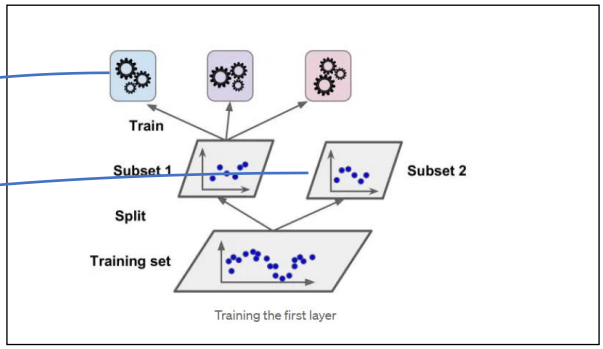


Figure 7-11. Aggregating predictions using a blending predictor





- It is actually possible to train **several different blenders** this way (e.g., one **using linear regression**, another using **random forest regression**) to get a whole layer of blenders, and then add another blender on top of that to produce the final prediction.
- The trick is to split the training set int o 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 the predictions made by the predictors of the first layer).
- The third one is used to create the training set used to train the third layer(using the predictions made by the predictors of the second layer).
- Then we can make a prediction for anew instance by going through each layer sequentially.

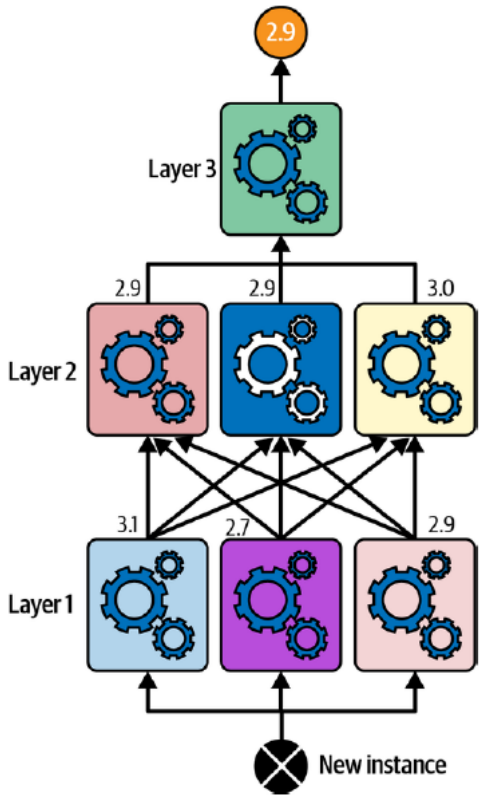


Figure 7-13. Predictions in a multilayer stacking ensemble

- 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)
```

- For each predictor, the stacking classifier will call **predict_proba()** if available; if not it will fall back to **decision_function()** or, as a last resort, call **predict()**.
- If you don’t provide a final estimator, **StackingClassifier** will use **LogisticRegression** and **StackingRegressor** will use **RidgeCV**.

- If you evaluate this stacking model on the test set, you will find 92.8% accuracy, which is a bit better than the voting classifier using soft voting, which got 92%.
- In conclusion, ensemble methods are versatile, powerful, and fairly simple to use.
- Random forests, AdaBoost, and GBRT are among the first models you should test for most machine learning tasks, and they particularly shine with heterogeneous tabular data.
- Moreover, as they require very little preprocessing, they're great for getting a prototype up and running quickly.
- Lastly, ensemble methods like voting classifiers and stacking classifiers can help push your system's performance to its limits.

Questions

1. **If you have trained five different models on the exact same training data, and they all achieve 95% precision, is there any chance that you can combine these models to get better results? If so, how? If not, why?**

   it will often give better results, it works better if the models are very different or it is trained on different training instances, but if not this will still be effective as long as the models are very different

2. **Is it possible to speed up training of a bagging ensemble by distributing it across multiple servers? What** about pasting ensembles, boosting ensembles, random forests, or stacking ensembles?
   It is possible to speed up training of a bagging , Pasting and random forest by distributing it across multiple servers.
   You can't do that with boosting ensembles, the training is sequential.
   For stacking ensembles, the predictors in each layer can be trained in parallel on multiple servers, however predictors in one layer can only be trained after the predictors in the previous layer have been trained.

3. **If your AdaBoost ensemble underfits the training data, which hyperparameters should you tweak, and how?**

   You can try increasing the no. of estimators or reducing the regularization hyperparameters of the base estimator, you may also try slightly increase the learning rate.

4. **If your gradient boosting ensemble overfits the training set, should you increase or decrease the learning rate?**
   You should try decreasing the learning rate, you could also use early stopping to find the right no. of predictors(you probably have too many)