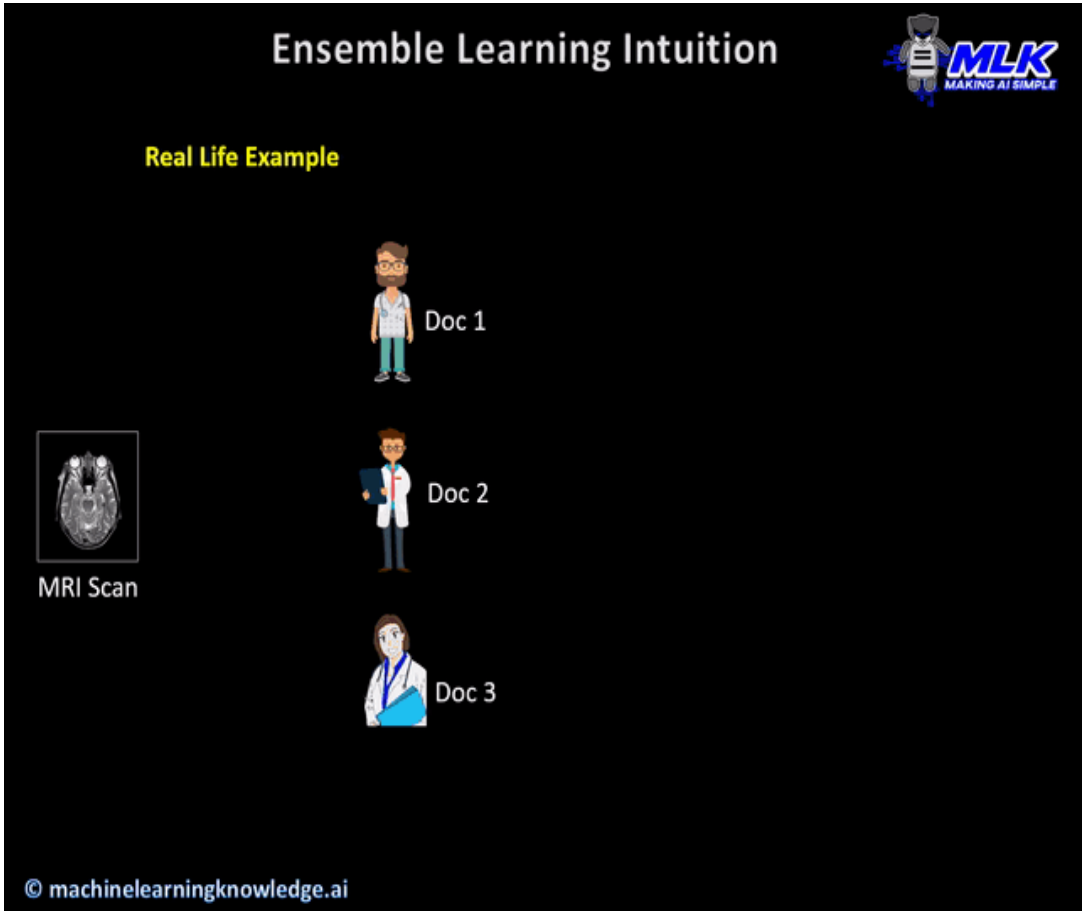
Ensemble Learning and Random Forests

Chapter 7

KALAMAZOO COLLEGE

Today

- Ensemble Learning
- Bagging and Pasting
- Random Forests

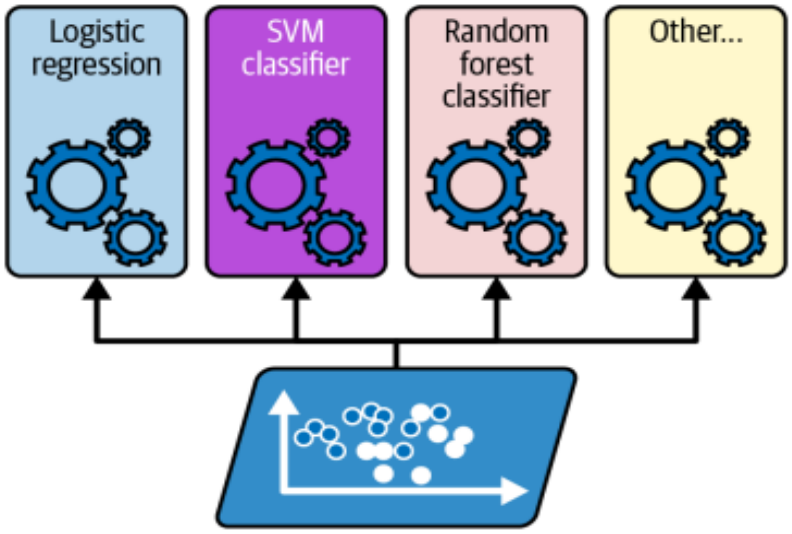


wisdom of the crowd

Ensemble Learning

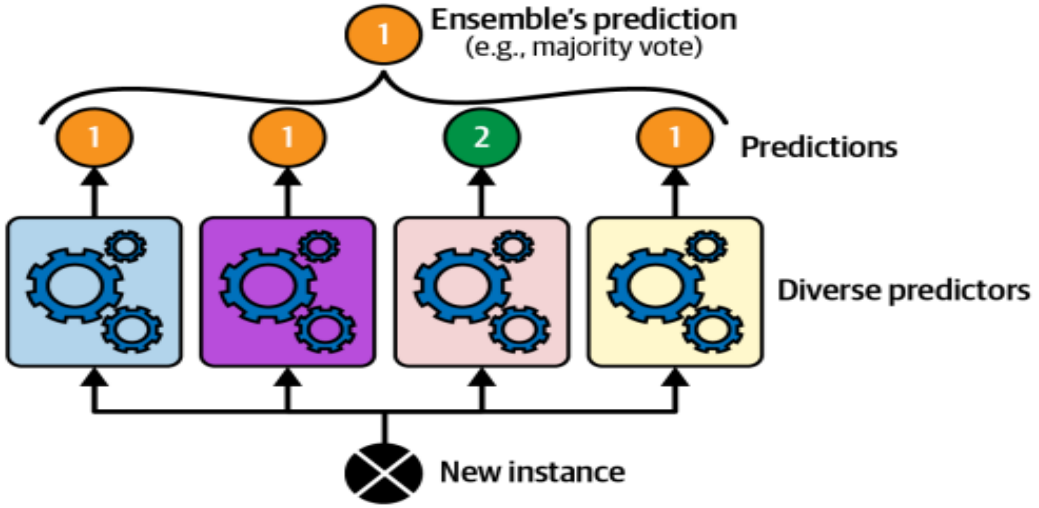
- A group of predictors(models) is called an **ensemble**.
- Ensemble learning is the process by which multiple models are strategically generated and combined to solve a particular computational intelligence problem.
- The winning solutions in machine learning competitions often involve several ensemble methods.

Ensemble Learning



Voting Classifiers

A Voting Classifier is a machine learning model that trains on an ensemble of numerous models and predicts an output (class) based on their highest probability of chosen class as the output.

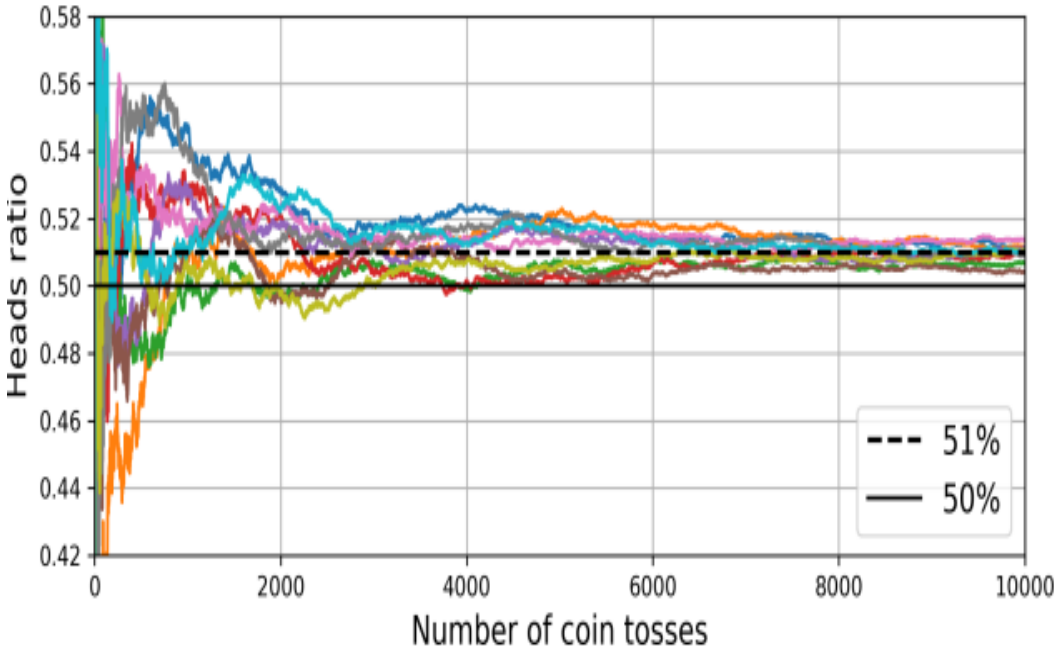


Ensemble Learning

- Voting classifier often achieves a higher accuracy than the best classifier in the ensemble. How?

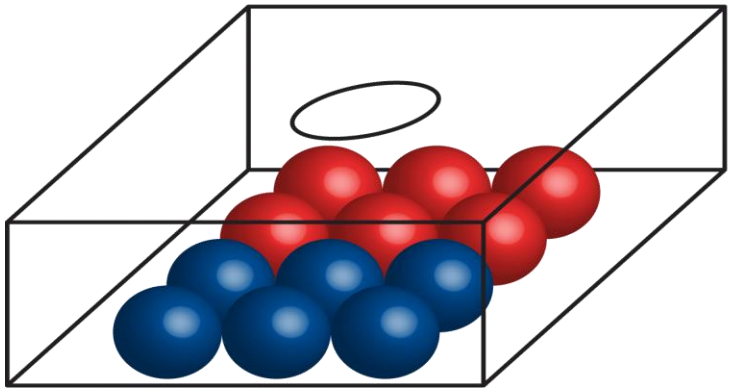
Suppose you have a slightly biased coin that has a 51% chance of coming up heads and 49% chance of coming up tails.

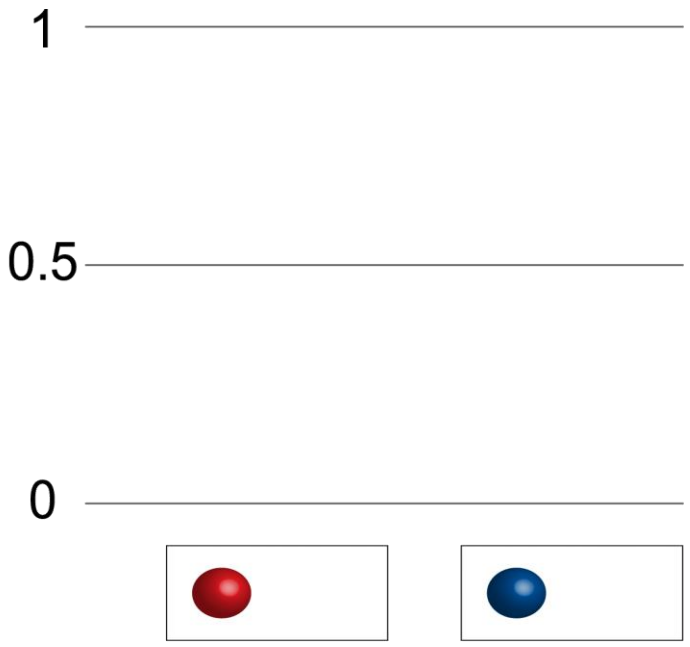
- If you toss it 1,000 times, you will get more or less 510 heads and 490 tails.
- The probability of obtaining a majority of heads after 1,000 tosses is close to 75%. The more you toss the coin the probability climbs over 97%
- This is due to the **law of large numbers**: as you keep tossing the coin, the ratio of heads gets closer and closer to the probability of heads (51%).



Ensemble Learning

- **Law of Large Numbers**: is a theorem from probability and statistics that suggests that the average result from repeating an experiment multiple times will better approximate the true or expected underlying result.





Ensemble Learning

- Voting Classifier supports two types of votings:
  1. **Hard Voting:**

     In hard voting, the predicted output class is a class with the highest majority of votes (the class which had the highest probability of being predicted by each of the classifiers).

     Example:

     Suppose three classifiers predicted the ***output class(A, A, B),*** so here the majority predicted *A* as output. Hence ***A* will be the final prediction**.

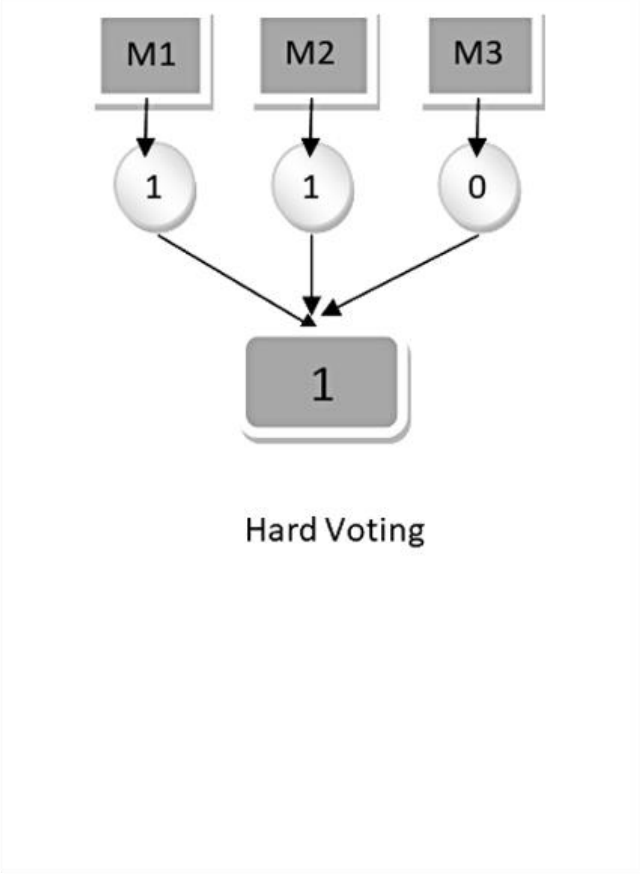
  2. **Soft Voting:**

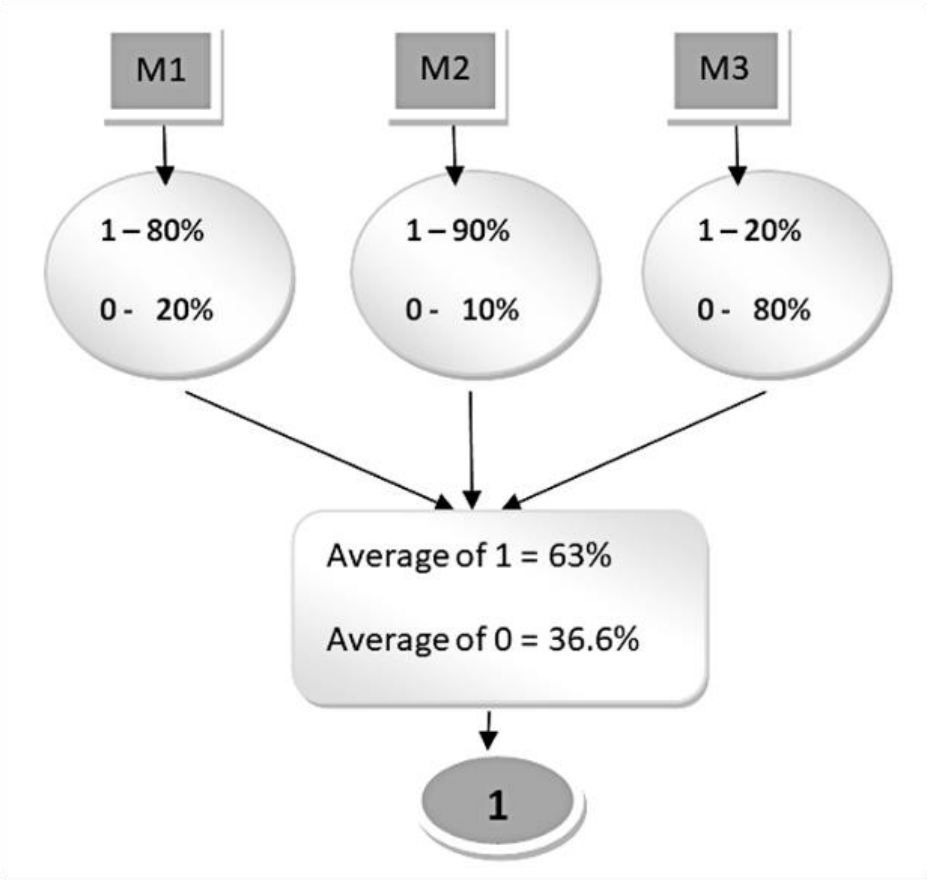
     In soft voting, the output class is the prediction based on the average of probability given to that class.

     Example:

     Suppose given some input to three models, the prediction probability for class ***A = (0.30, 0.47, 0.53)*** and ***B = (0.20, 0.32, 0.40)***. So the average for class ***A is 0.4333*** and ***B is 0.3067***, the winner is clearly **class *A*** because it had the highest probability averaged by each classifier.

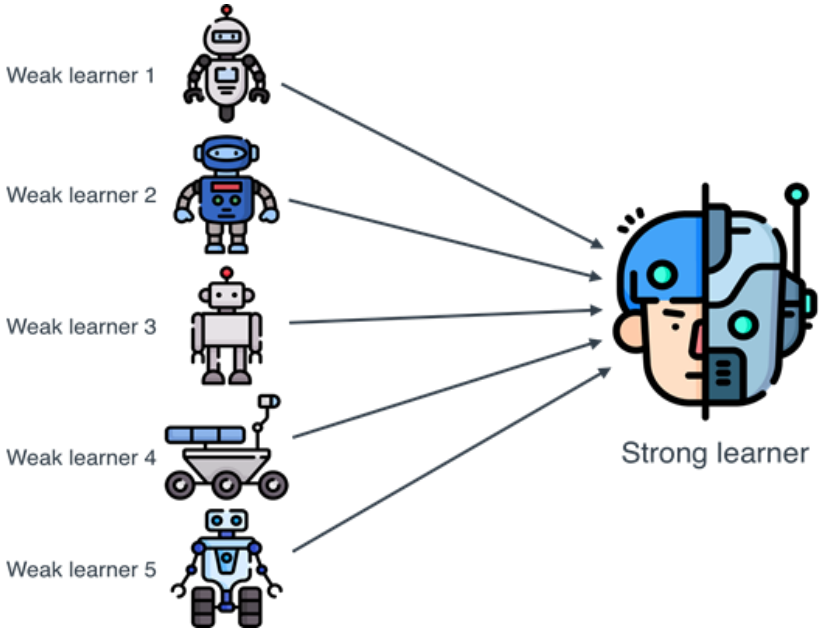
Ensemble Learning





Ensemble Learning

- Ensemble methods **work best** when the predictors are as independent from one another as possible.
- One way to get diverse classifiers is to train them using very **different algorithms**.
- This increases the chance that they will make very **different types of errors**, improving the ensemble's accuracy



Ensemble Learning

- To get a diverse set of classifiers
  - Use very different training algorithms
  - Use the same training algorithm for every predictor but train them on different random subsets of the training set.

Ensemble Learning

```
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)
```

Hard or Soft?

```
>>> 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
```

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

```
>>> voting_clf.predict(X_test[:1])
array([1])
>>> [clf.predict(X_test[:1]) for clf in voting_clf.estimators_]
[array([1]), array([1]), array([0])]
```

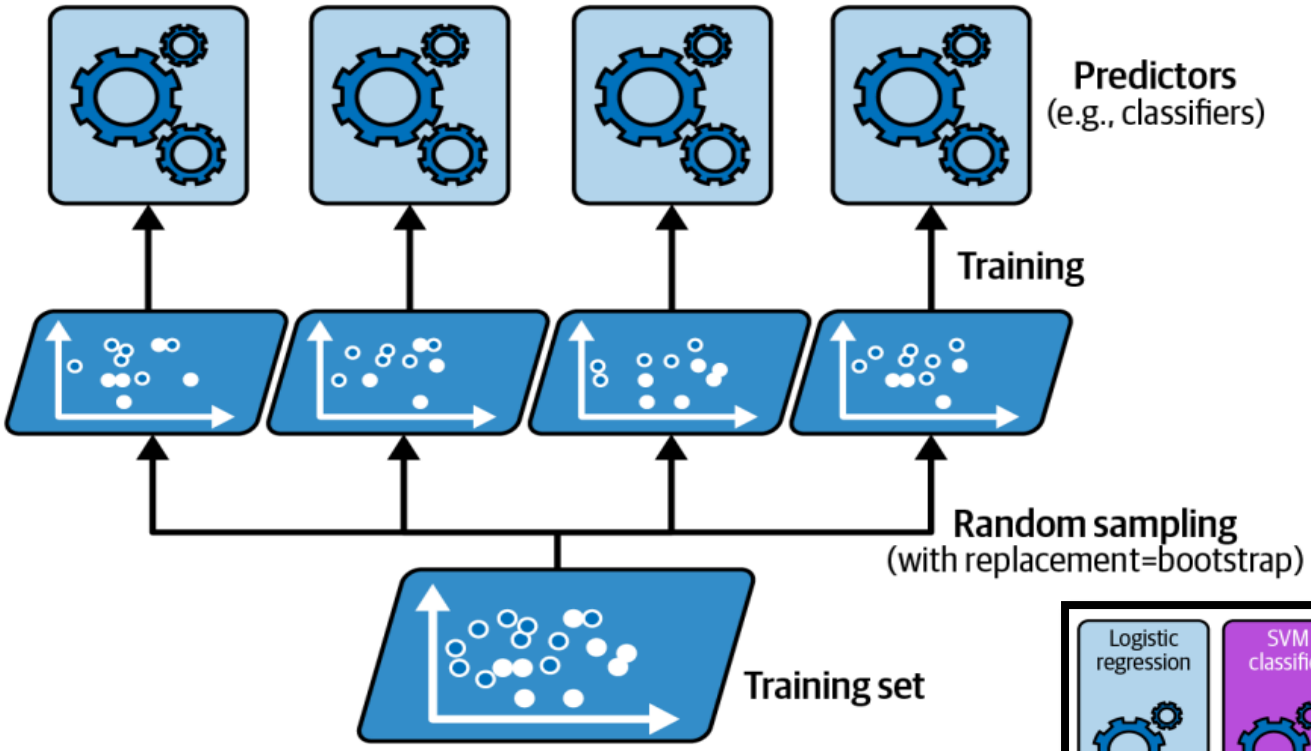
Ensemble Learning

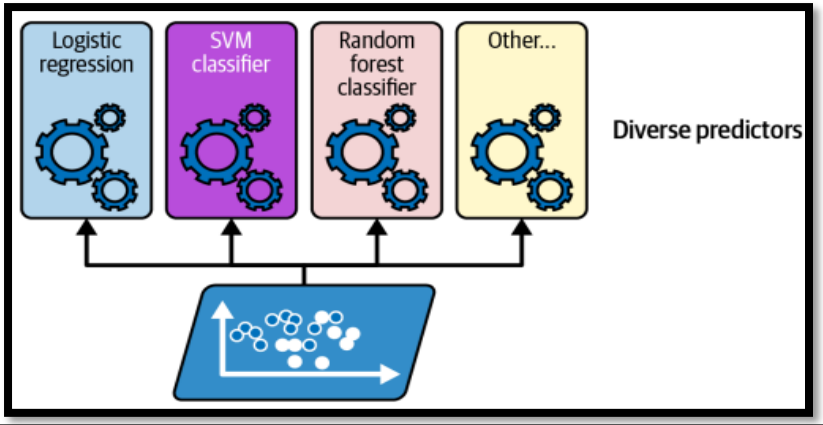
```
>>> voting_clf.voting = "soft"
>>> voting_clf.named_estimators["svc"].probability = True
>>> voting_clf.fit(X_train, y_train)
>>> voting_clf.score(X_test, y_test)
0.92
```

Bagging and Pasting

- **Bagging and Pasting** is to use the same training algorithm for every predictor but train them on different random subsets of the training set
  - When sampling is performed with replacement this method is called **bagging** (short for bootstrap aggregating).
  - When sampling is performed without replacement, it is called **pasting**.

Bagging and Pasting





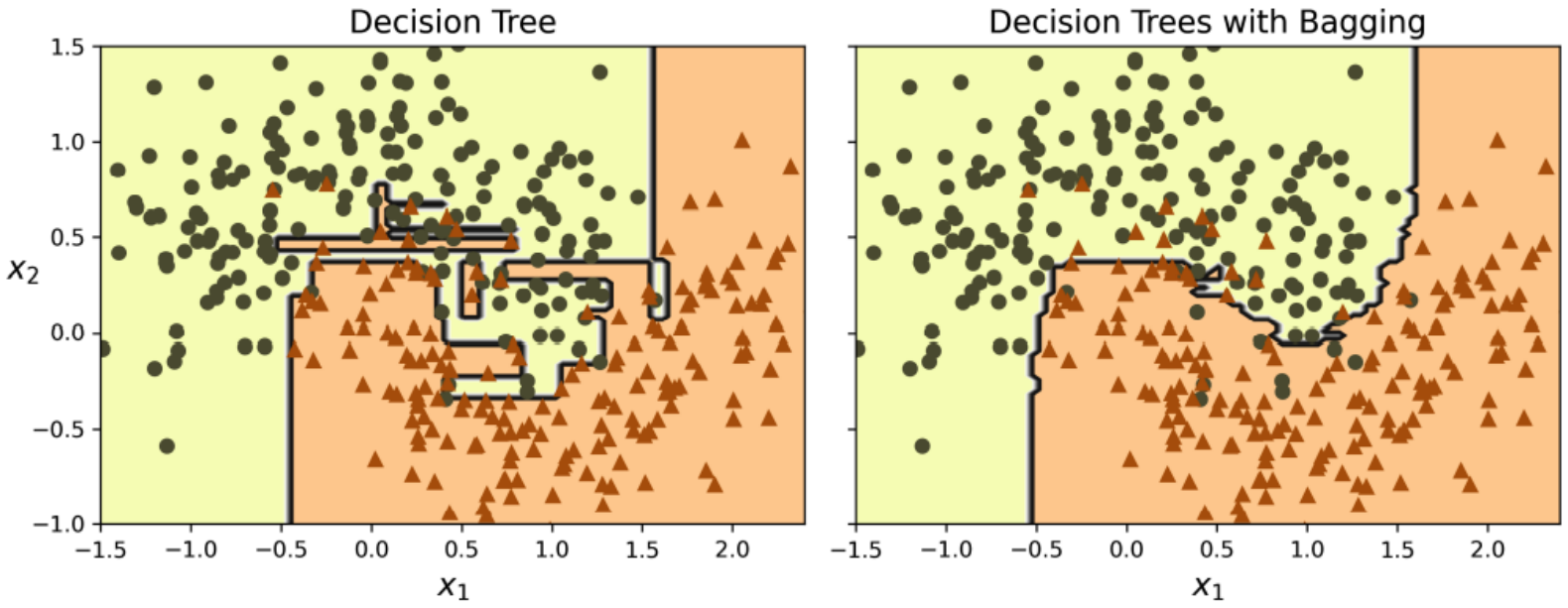
Bagging and Pasting

- Once all predictors are trained, the ensemble can make a prediction for a new instance by:
  - Aggregating the predictions of all predictors (i.e., the most frequent prediction, just like with a hard voting classifier),
  - Or the average for regression

```
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)
```

bagging
soft voting

Bagging and Pasting



Bagging and Pasting- Out-of-Bag Evaluation

- In Bagging, the remaining (about 37%) of the training instances that are not sampled are called **out-of-bag (OOB) instances.**
- A bagging ensemble can be evaluated using OOB instances, without the need for a separate validation set

```
>>> 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
```

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

Bagging and Pasting - Random Patches and Random Subspaces

- The BaggingClassifier class supports sampling the **features** as well.
- Sampling is controlled by two hyperparameters: max_features and bootstrap_features
- This technique is particularly useful when you are dealing with high-dimensional inputs (such as images), as it can considerably speed up training.
- Sampling both training instances and features is called **the random patches method**.
- Keeping all training instances (by setting bootstrap=False and max_samples=1.0) but sampling features (by setting bootstrap_features to True and/or max_features to a value smaller than 1.0) is called the **random subspaces method**.

Random Forests

- A random forest is an ensemble of decision trees, generally trained via the bagging method (or sometimes pasting), typically with max_samples set to the size of the training set.

```
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)
```

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

Random Forests

- Random forests make it easy to measure the relative importance of each feature

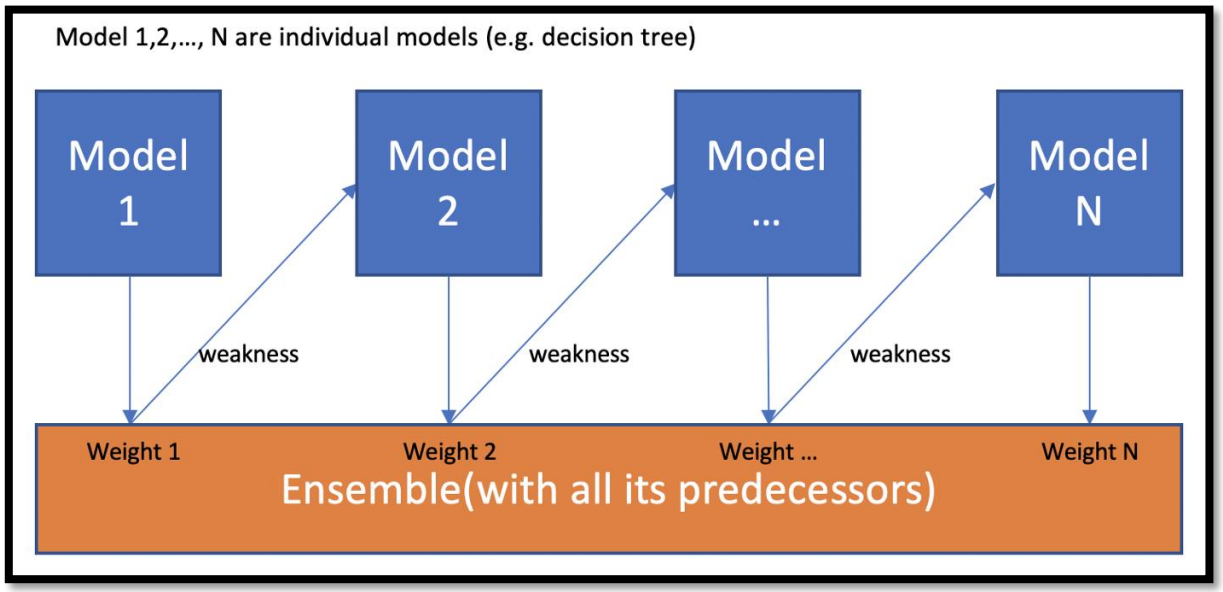
```
>>> from sklearn.datasets import load_iris
>>> iris = load_iris(as_frame=True)
>>> rnd_clf = RandomForestClassifier(n_estimators=500, random_state=42)
>>> rnd_clf.fit(iris.data, iris.target)
>>> for score, name in zip(rnd_clf.feature_importances_, iris.data.columns):
...     print(round(score, 2),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

0.44 petal length (cm)
0.42 petal width (cm)
```

Boosting

- Boosting is 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**.

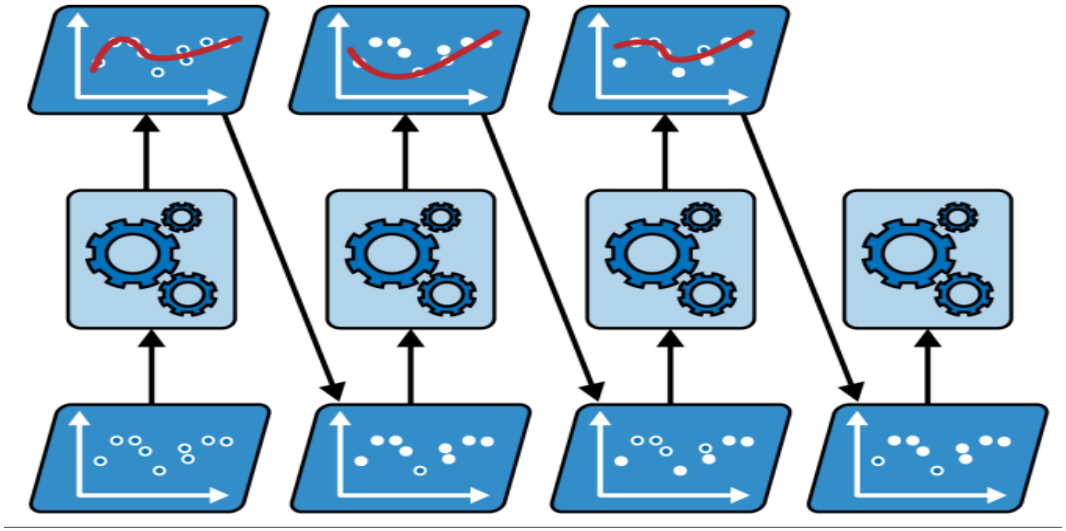
- boosting methods:
  - AdaBoost
  - gradient boosting



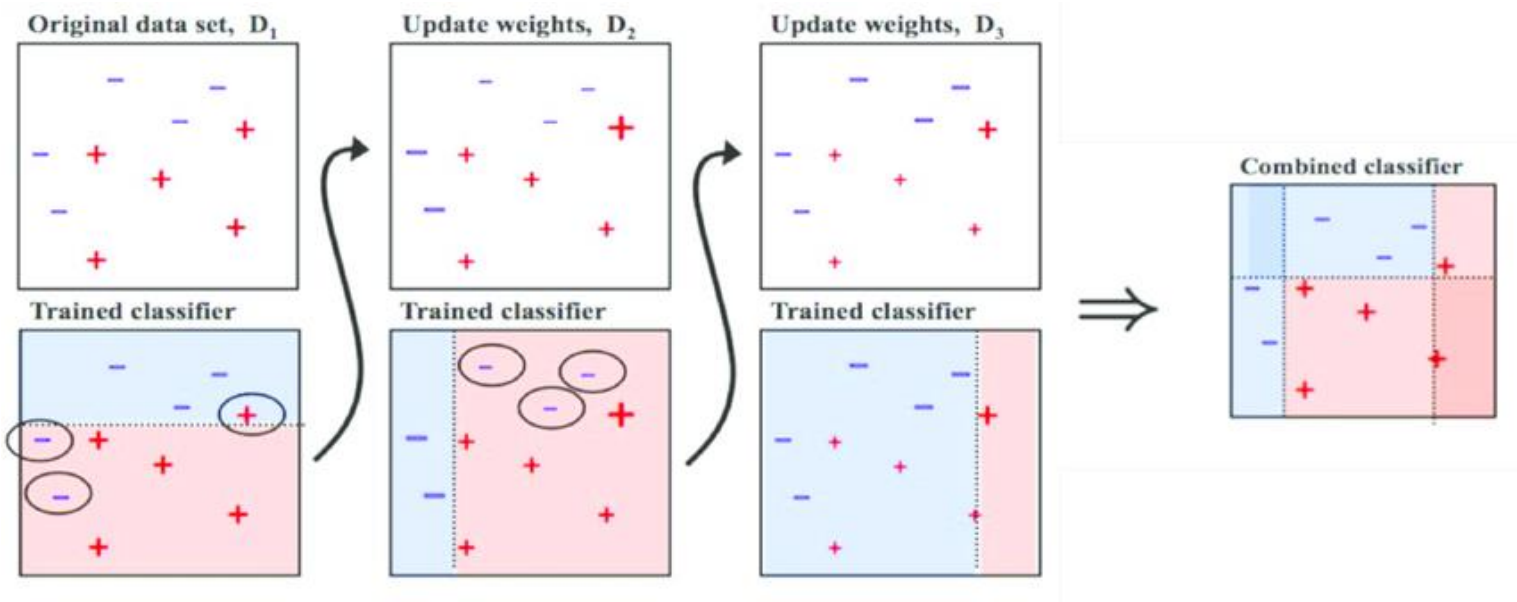
Boosting - AdaBoost

- AdaBoost: One way for a new predictor (model) to correct its predecessor is to pay a bit more attention to the training instances that the predecessor underfit, which results in **new predictors** focusing more and more on the hard cases.

AdaBoost adds predictors to the ensemble, gradually making it better.



Boosting - AdaBoost



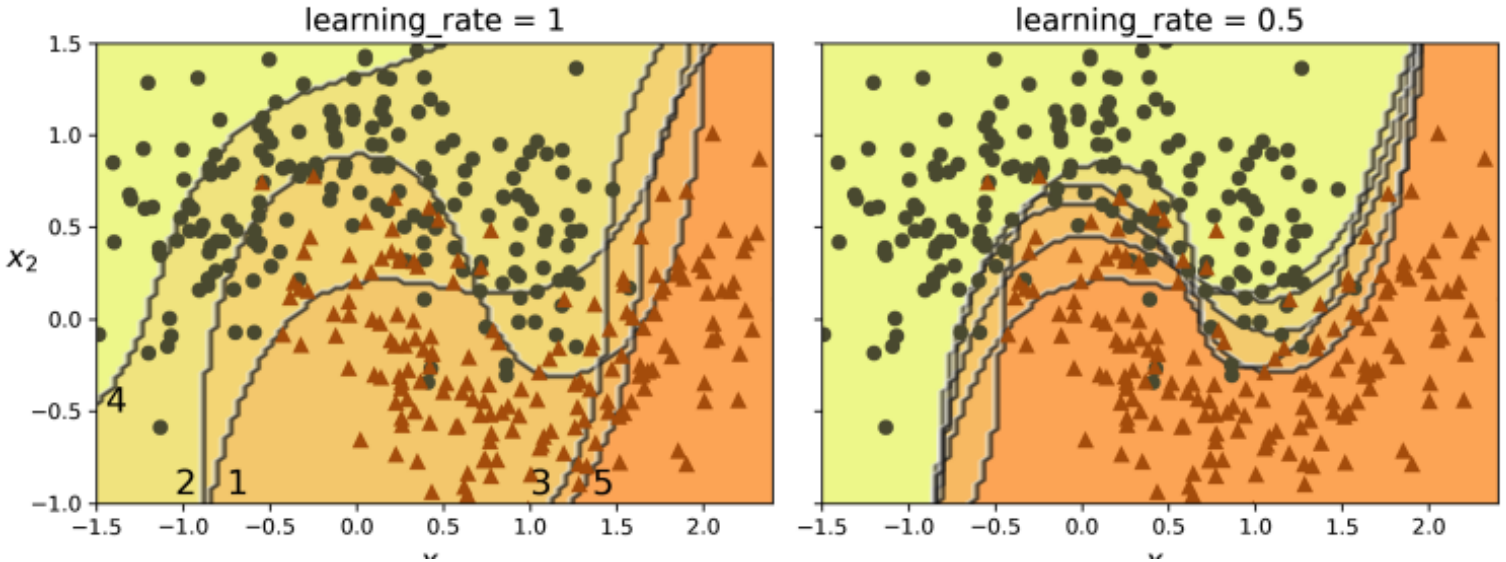
Algorithm:

1.Initialise the dataset and assign equal weight to each of the data point.
2.Provide this as input to the model and identify the wrongly classified data points.
3.Increase the weight of the wrongly classified data points.
4.if (got required results)
Goto step 5
else
Goto step 2
End

Boosting – AdaBoost (Prediction)

- For example:
  - 5 weak classifiers may predict the values **1.0, 1.0, -1.0, 1.0, -1.0.**
  - From a majority vote, it looks like the model will predict a value of 1.0 or the first class.
  - These same 5 weak classifiers may have the weights **0.2, 0.5, 0.8, 0.2 and 0.9** respectively.
  - Calculating the weighted sum of these predictions results in an output of -0.8. And which would be an ensemble prediction of -1.0 or the second class.

Boosting - AdaBoost



Each predictor is a highly regularized SVM classifier

Boosting - Gradient Boosting

- Gradient boosting works by sequentially adding predictors to an ensemble, each one correcting its predecessor.
- Instead of tweaking the instance weights at every iteration like AdaBoost does, this method tries to fit the new predictor to the **residual errors** made by the previous predictor

Boosting - Gradient Boosting

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

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

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

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

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

Ensemble containing three trees and 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])
```

Boosting - Gradient Boosting