Ensemble Learning

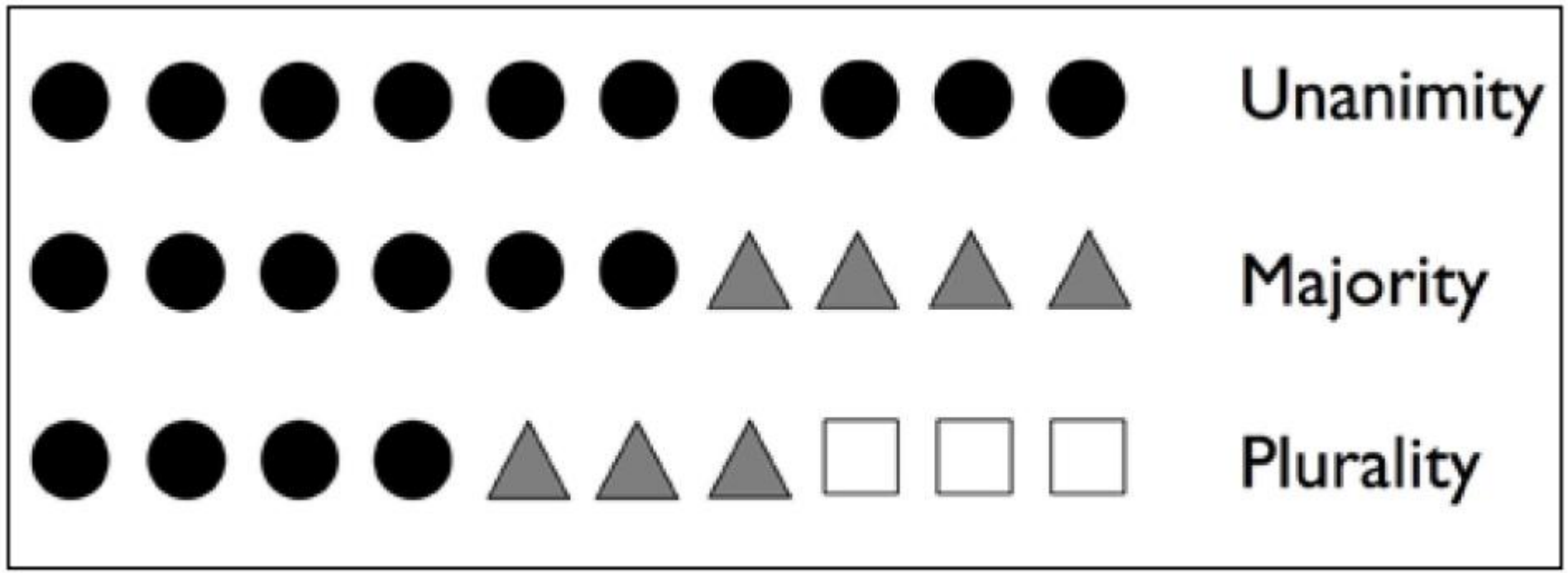
Combining Different Models for Ensemble Learning

- Make predictions based on majority voting
- Use bagging to reduce overfitting by drawing random combinations of the training set with repetition
- Apply boosting to build powerful models from *weak learners* that learn from their mistakes

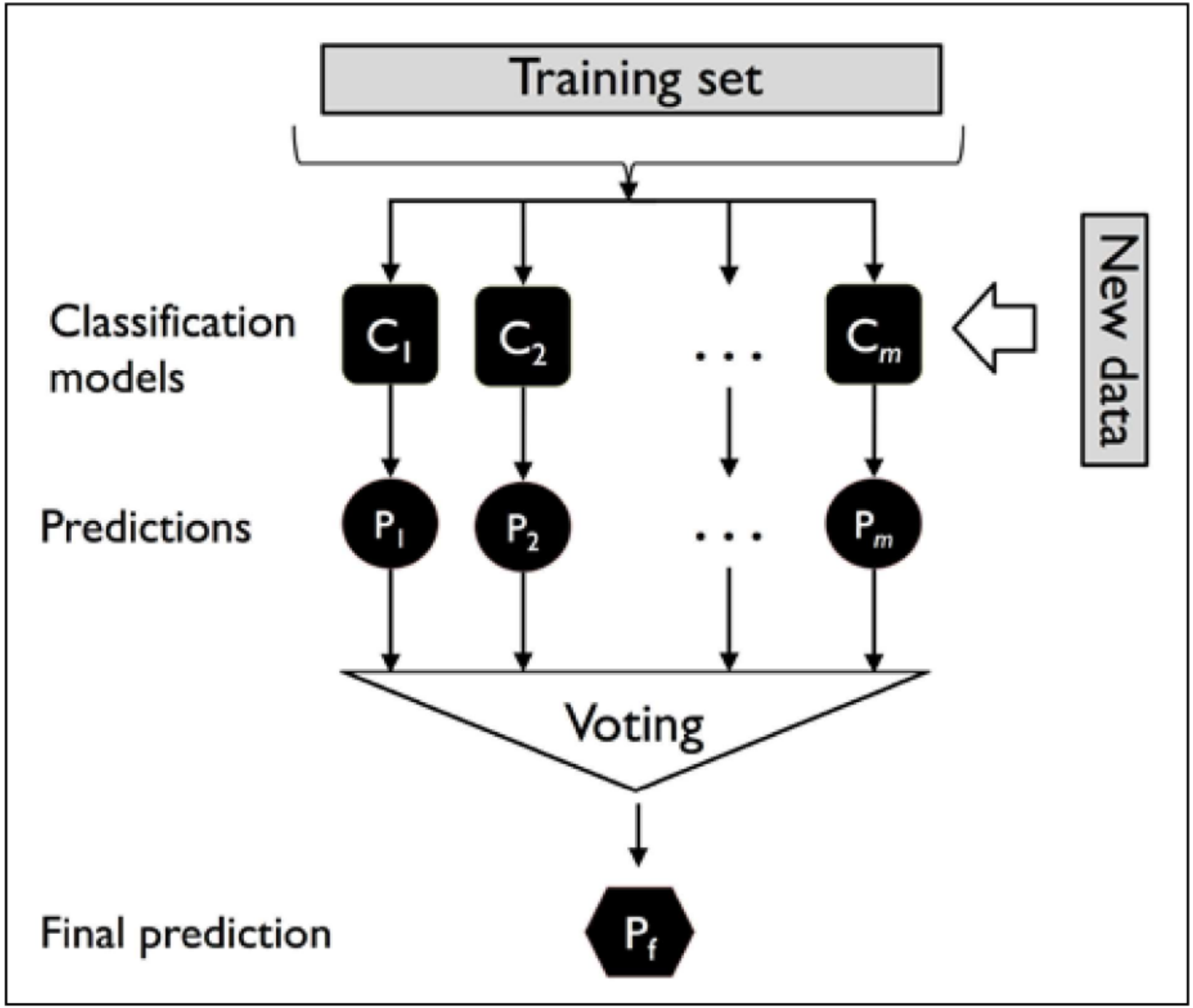
Learning with ensembles

The goal of **ensemble methods** is to combine different classifiers into a meta-classifier that has better generalization performance than each individual classifier alone.

In this chapter, we will focus on the most popular ensemble methods that use the **majority voting** principle. Majority voting simply means that we select the class label that has been predicted by the majority of classifiers, that is, received more than 50 percent of the votes. Strictly speaking, the term **majority vote** refers to binary class settings only. However, it is easy to generalize the majority voting principle to multi-class settings, which is called **plurality voting**.


Unanimity
Majority
Plurality

Using t
Depend
algorith
classifie
algorith
this app
tree clas
approac



.
fication
gression
tion
ple of
ision
mble

To predict a class label via simple majority or plurality voting, we combine the predicted class labels of each individual classifier, C_j, and select the class label, $\hat{y}$, that received the most votes:

$$\hat{y} = mode\left\{C_1(\boldsymbol{x}), C_2(\boldsymbol{x}), \ldots, C_m(\boldsymbol{x})\right\}$$

For example, in a binary classification task where $class1 = -1$ and $class2 = +1$, we can write the majority vote prediction as follows:

$$C(\boldsymbol{x}) = sign\left[\sum_j^m C_j(\boldsymbol{x})\right] = \begin{cases} 1 \ \ if \sum_i C_j(\boldsymbol{x}) \geq 0 \\ -1 \ otherwise \end{cases}$$

Why ensemble better than individual classifiers?

To illustrate why ensemble methods can work better than individual classifiers alone, let's apply the simple concepts of combinatorics. For the following example, we make the assumption that all n-base classifiers for a binary classification task have an equal error rate, ε. Furthermore, we assume that the classifiers are independent and the error rates are not correlated. Under those assumptions, we can simply express the error probability of an ensemble of base classifiers as a probability mass function of a binomial distribution:

$$P(y \geq k) = \sum_{k}^{n} \binom{n}{k} \varepsilon^{k} (1- \varepsilon)^{n-k} = \varepsilon_{ensemble}$$

Where

$$\binom{n}{k} = \frac{n!}{(n-k)!k!}$$

Here, $\binom{n}{k}$ is the binomial coefficient **n choose k**. In other words, we compute the probability that the prediction of the ensemble is wrong.

more concrete example of 11 base classifiers ($n = 11$), where each classifier has an error rate of 0.25 ($\varepsilon = 0.25$):

$$P(y \geq k) = \sum_{k=6}^{11} \binom{11}{k} 0.25^k (1-\varepsilon)^{11-k} = 0.034$$

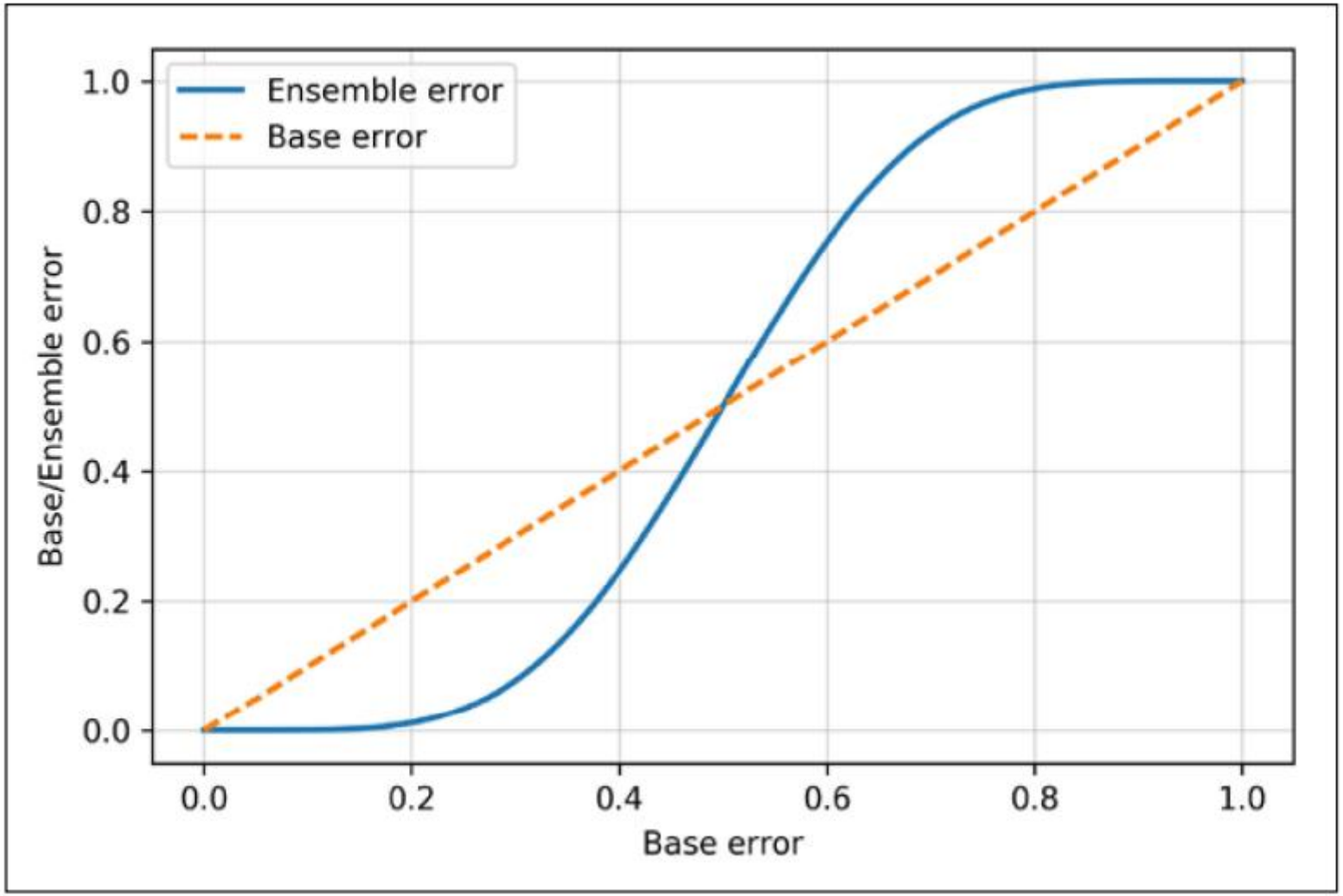
As we can see, the error rate of the ensemble (0.034) is much lower than the error rate of each individual classifier (0.25) if all the assumptions are met.

as an error, whereas this is only true half of the time. To compare such an idealistic ensemble classifier to a base classifier over a range of different base error rates, let's implement the probability mass function in Python:

```
>>> from scipy.misc import comb
>>> import math
>>> def ensemble_error(n_classifier, error):
...     k_start = int(math.ceil(n_classifier / 2.))
...     probs = [comb(n_classifier, k) *
...              error**k *
...              (1-error)**(n_classifier - k)
...              for k in range(k_start, n_classifier + 1)]
...     return sum(probs)
>>> ensemble_error(n_classifier=11, error=0.25)
0.03432750701904297
```

After we have impl
ensemble error rate
the relationship bet

```
>>> import num
>>> import mat
>>> error_rang
>>> ens_errors
...
>>> plt.plot(e
...              l
...              l
>>> plt.plot(e
...              l
...              l
>>> plt.xlabel
>>> plt.ylabel
>>> plt.legend
>>> plt.grid(a
>>> plt.show()
```



Implementing a simple majority vote classifier

The algorithm that we are going to implement in this section will allow us to combine different classification algorithms associated with individual weights for confidence. Our goal is to build a stronger meta-classifier that balances out the individual classifiers' weaknesses on a particular dataset. In more precise mathematical terms, we can write the weighted majority vote as follows:

$$\hat{y} = \arg\max_{i} \sum_{j=1}^{m} w_j \chi_A \left(C_j \left(\boldsymbol{x} \right) = i \right)$$

Here, w_j is a weight associated with a base classifier, C_j, $\hat{y}$ is the predicted class label of the ensemble, χ_A (Greek chi) is the characteristic function $\left[C_j \left(\boldsymbol{x} \right) = i \in A \right]$, and A is the set of unique class labels. For equal weights, we can simplify this equation and write it as follows:

$$\hat{y} = mode \left\{ C_1 \left(\boldsymbol{x} \right), C_2 \left(\boldsymbol{x} \right), \ldots, C_m \left(\boldsymbol{x} \right) \right\}$$

In statistics, the *mode* is the most frequent event or result in a set. For example, *mode{1,2,1 1,2,4,5,4} = 1*.

To better understand the concept of *weighting*, we will now take a look at a more concrete example. Let us assume that we have an ensemble of three base classifiers, C_j ($j \in \{0,1\}$), and want to predict the class label of a given sample instance, **x**. Two out of three base classifiers predict the class label 0, and one, C_3, predicts that the sample belongs to class 1. If we weight the predictions of each base classifier equally, the majority vote would predict that the sample belongs to class 0:

$$C_1(\boldsymbol{x}) \rightarrow 0,\ C_2(\boldsymbol{x}) \rightarrow 0,\ C_3(\boldsymbol{x}) \rightarrow 1$$

$$\hat{y} = mode\{0,0,1\} = 0$$

Now, let us assign a weight of 0.6 to C_3 and weight C_1 and C_2 by a coefficient of 0.2:

$$\hat{y} = \arg\max_i \sum_{j=1}^{m} w_j \chi_A \left(C_j(\boldsymbol{x}) = i \right)$$

$$= \arg\max_i \left[0.2 \times i_0 + 0.2 \times i_0 + 0.6 \times i_1 \right] = 1$$

More intuitively, since 3 x 0.2 = 0.6, we can say that the prediction made by C_3 has three times more weight than the predictions by C_1 or C_2, which we can write as follows:

$$\hat{y} = mode\{0,0,1,1,1\} = 1$$

Using the majority voting principle to make predictions

Furthermore, we will only select two features, **sepal width** and **petal length**, to make the classification task more challenging for illustration purposes. Although our `MajorityVoteClassifier` generalizes to multiclass problems, we will only classify flower samples from the `Iris-versicolor` and `Iris-virginica` classes, with which we will compute the ROC AUC later. The code is as follows:

```
>>> from sklearn import datasets
>>> from sklearn.model_selection import train_test_split
>>> from sklearn.preprocessing import StandardScaler
>>> from sklearn.preprocessing import LabelEncoder
>>> iris = datasets.load_iris()
>>> X, y = iris.data[50:, [1, 2]], iris.target[50:]
>>> le = LabelEncoder()
>>> y = le.fit_transform(y)
```

Next, we split the Iris samples into 50 percent training and 50 percent test data:

```
>>> X_train, X_test, y_train, y_test =\
...         train_test_split(X, y,
...                          test_size=0.5,
...                          random_state=1,
...                          stratify=y)
```

Using the training dataset, we now will train three different classifiers:

- Logistic regression classifier
- Decision tree classifier
- k-nearest neighbors classifier

We then evaluate the model performance of each classifier via 10-fold cross-validation on the training dataset before we combine them into an ensemble classifier:

```
>>> from sklearn.model_selection import cross_val_score
>>> from sklearn.linear_model import LogisticRegression
>>> from sklearn.tree import DecisionTreeClassifier
>>> from sklearn.neighbors import KNeighborsClassifier
>>> from sklearn.pipeline import Pipeline
>>> import numpy as np
>>> clf1 = LogisticRegression(penalty='l2',
...                           C=0.001,
...                           random_state=1)
>>> clf2 = DecisionTreeClassifier(max_depth=1,
...                               criterion='entropy',
...                               random_state=0)
```

```
>>> clf3 = KNeighborsClassifier(n_neighbors=1,
...                             p=2,
...                             metric='minkowski')
>>> pipe1 = Pipeline([['sc', StandardScaler()],
...                   ['clf', clf1]])
>>> pipe3 = Pipeline([['sc', StandardScaler()],
...                   ['clf', clf3]])
>>> clf_labels = ['Logistic regression', 'Decision tree', 'KNN']
>>> print('10-fold cross validation:\n')
>>> for clf, label in zip([pipe1, clf2, pipe3], clf_labels):
...     scores = cross_val_score(estimator=clf,
...                              X=X_train,
...                              y=y_train,
...                              cv=10,
...                              scoring='roc_auc')
...     print("ROC AUC: %0.2f (+/- %0.2f) [%s]"
...           % (scores.mean(), scores.std(), label))
```

The output that we receive, as shown in the following snippet, shows that the predictive performances of the individual classifiers are almost equal:

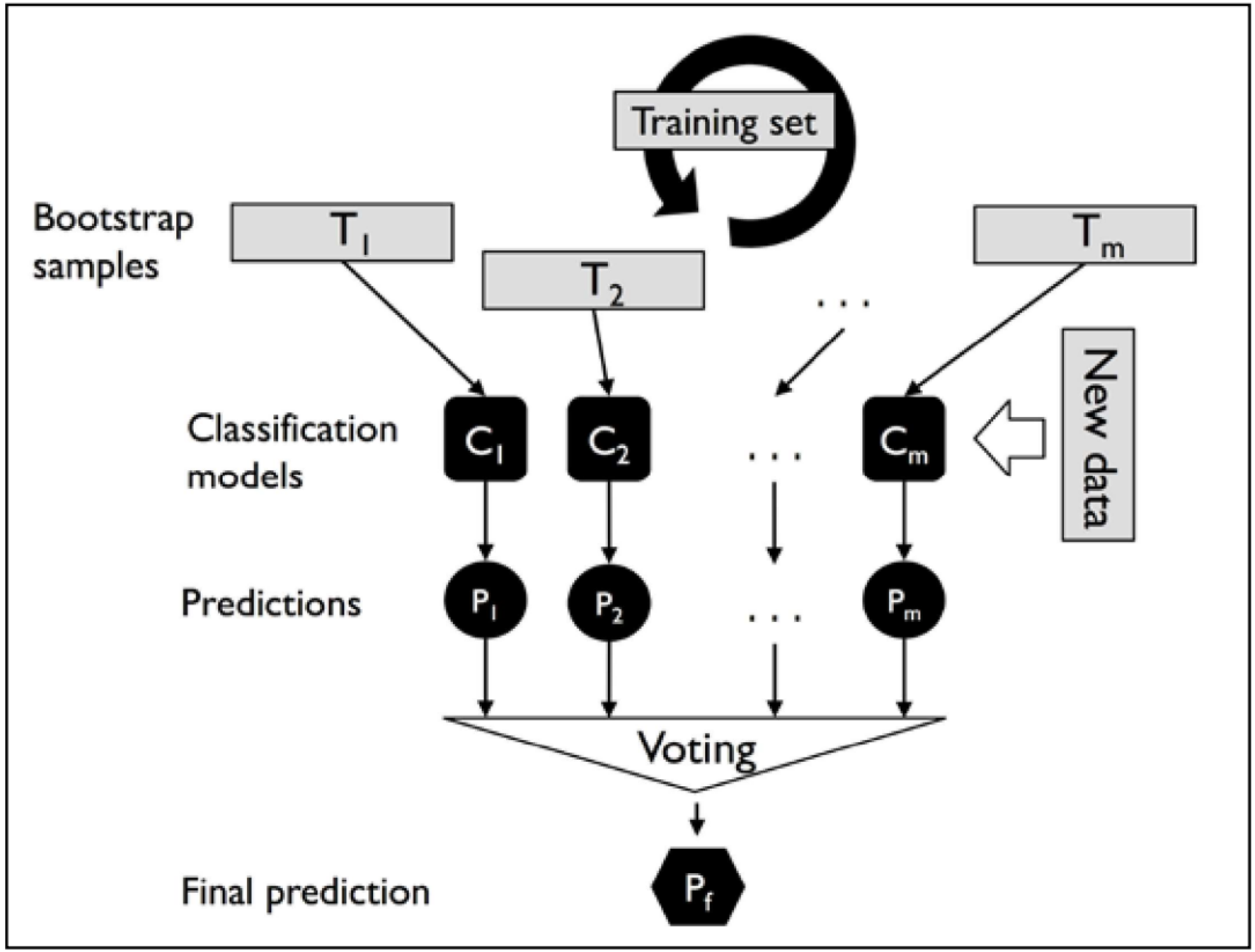
```
10-fold cross validation:

ROC AUC: 0.87 (+/- 0.17) [Logistic regression]
ROC AUC: 0.89 (+/- 0.16) [Decision tree]
ROC AUC: 0.88 (+/- 0.15) [KNN]
```

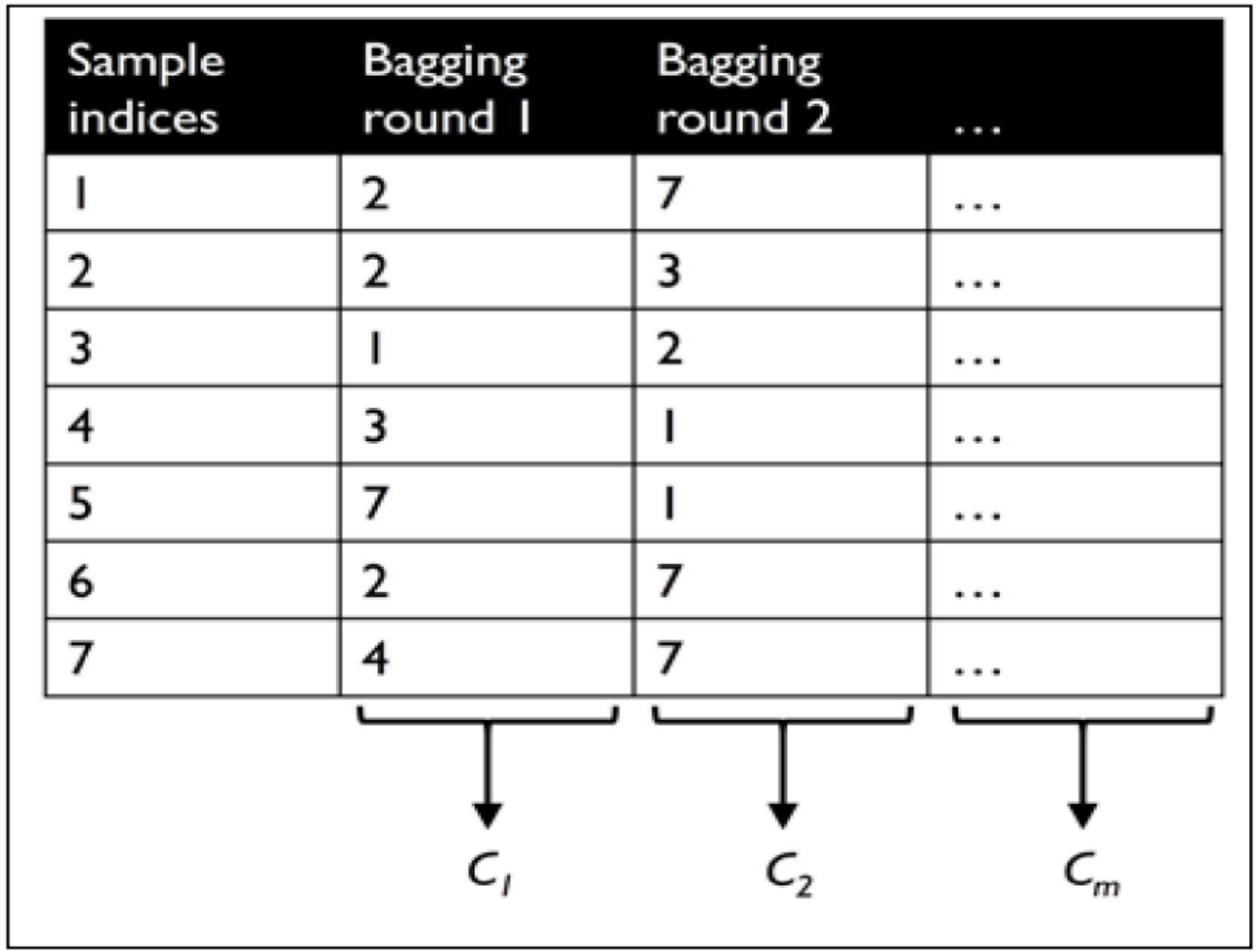
```
# Voting
from sklearn.ensemble import VotingClassifier
mv_clf=VotingClassifier(estimators=[('lr',pipe1),('dt',clf2),('knn',pipe3)],
                        voting='hard',
                        weights=[1, 1, 5])
mv_clf=mv_clf.fit(X,y)
s=mv_clf.score(X_test,y_test)
print(s)
print(mv_clf.predict(X))
```

B
cl

Baggi
Major
instea
we dr
trainir

The cc




Sample indices
Bagging round 1
Bagging round 2
...
1
2
7
...
2
2
3
...
3
1
2
...
4
3
1
...
5
7
1
...
6
2
7
...
7
4
7
...
C_1
C_2
C_m

Applying bagging to classify samples in the Wine dataset

```
>>> import pandas as pd
>>> df_wine = pd.read_csv('https://archive.ics.uci.edu/ml/'
                          'machine-learning-databases/wine/wine.data',
                          header=None)
>>> df_wine.columns = ['Class label', 'Alcohol',
...                    'Malic acid', 'Ash',
...                    'Alcalinity of ash',
...                    'Magnesium', 'Total phenols',
...                    'Flavanoids', 'Nonflavanoid phenols',
...                    'Proanthocyanins',
...                    'Color intensity', 'Hue',
...                    'OD280/OD315 of diluted wines',
...                    'Proline']
>>> # drop 1 class
>>> df_wine = df_wine[df_wine['Class label'] != 1]
>>> y = df_wine['Class label'].values
>>> X = df_wine[['Alcohol',
...              'OD280/OD315 of diluted wines']].values
```

Next, we encode the class labels into binary format and split the dataset into 80 percent training and 20 percent test sets, respectively:

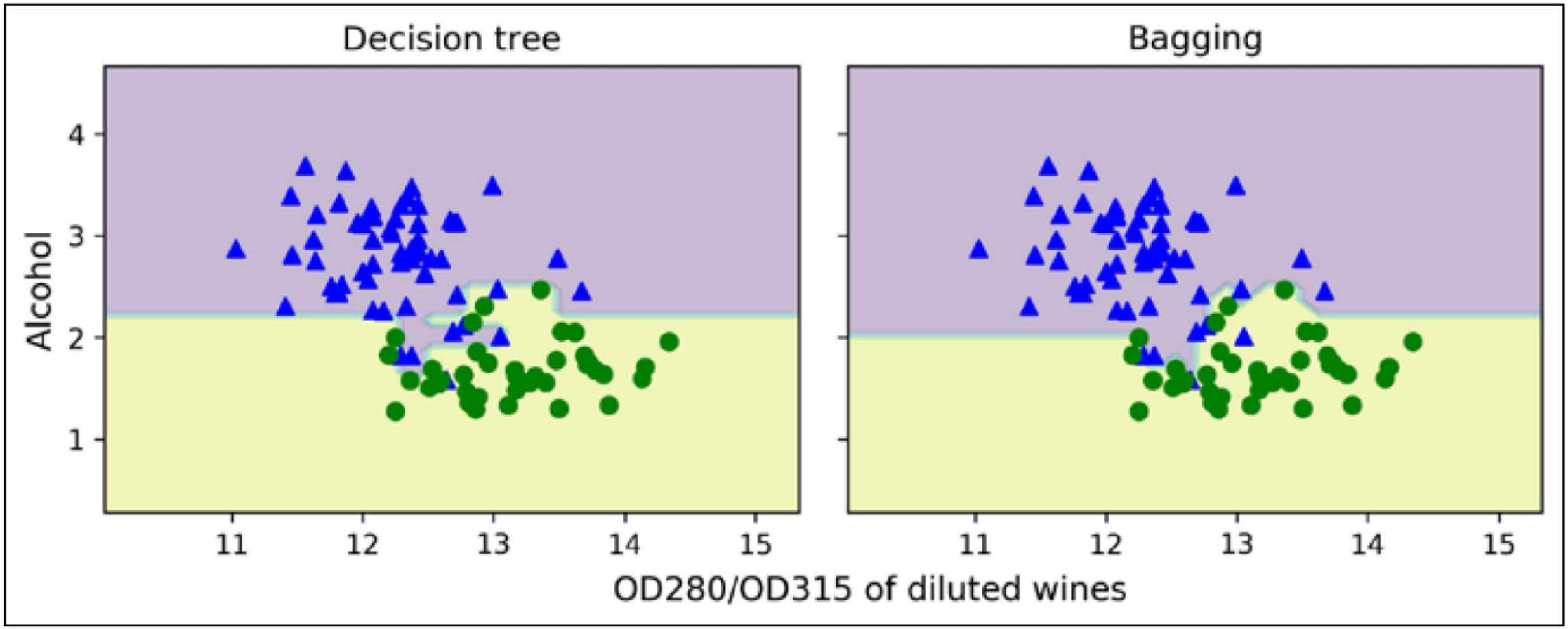
```
>>> from sklearn.preprocessing import LabelEncoder
>>> from sklearn.model_selection import train_test_split
>>> le = LabelEncoder()
>>> y = le.fit_transform(y)
>>> X_train, X_test, y_train, y_test =\
...             train_test_split(X, y,
...                              test_size=0.2,
...                              random_state=1,
...                              stratify=y)
```

```
>>> from sklearn.ensemble import BaggingClassifier
>>> tree = DecisionTreeClassifier(criterion='entropy',
...                               random_state=1,
...                               max_depth=None)
>>> bag = BaggingClassifier(base_estimator=tree,
...                         n_estimators=500,
...                         max_samples=1.0,
...                         max_features=1.0,
...                         bootstrap=True,
...                         bootstrap_features=False,
...                         n_jobs=1,
...                         random_state=1)
```

```
>>> from sklearn.metrics import accuracy_score
>>> tree = tree.fit(X_train, y_train)
>>> y_train_pred = tree.predict(X_train)
>>> y_test_pred = tree.predict(X_test)
>>> tree_train = accuracy_score(y_train, y_train_pred)
>>> tree_test = accuracy_score(y_test, y_test_pred)
>>> print('Decision tree train/test accuracies %.3f/%.3f'
...       % (tree_train, tree_test))
Decision tree train/test accuracies 1.000/0.833
```

```
>>> bag = bag.fit(X_train, y_train)
>>> y_train_pred = bag.predict(X_train)
>>> y_test_pred = bag.predict(X_test)
>>> bag_train = accuracy_score(y_train, y_train_pred)
>>> bag_test = accuracy_score(y_test, y_test_pred)
>>> print('Bagging train/test accuracies %.3f/%.3f'
...       % (bag_train, bag_test))
Bagging train/test accuracies 1.000/0.917
```

```
>>> x_min = X_train[:, 0].min() - 1
>>> x_max = X_train[:, 0].max() + 1
>>> y_min = X_train[:, 1].min() - 1
>>> y_max = X_train[:, 1].max() + 1
>>> xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.1),
...                      np.arange(y_min, y_max, 0.1))
>>> f, axarr = plt.subplots(nrows=1, ncols=2,
...                         sharex='col',
...                         sharey='row',
...                         figsize=(8, 3))
>>> for idx, clf, tt in zip([0, 1],
...                         [tree, bag],
...                         ['Decision tree', 'Bagging']):
...     clf.fit(X_train, y_train)
...
...     Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])
...     Z = Z.reshape(xx.shape)
...     axarr[idx].contourf(xx, yy, Z, alpha=0.3)
...     axarr[idx].scatter(X_train[y_train==0, 0],
...                        X_train[y_train==0, 1],
...                        c='blue', marker='^')
...     axarr[idx].scatter(X_train[y_train==1, 0],
...                        X_train[y_train==1, 1],
...                        c='green', marker='o')
...     axarr[idx].set_title(tt)
>>> axarr[0].set_ylabel('Alcohol', fontsize=12)
>>> plt.text(10.2, -1.2,
...          s='OD280/OD315 of diluted wines',
...          ha='center', va='center', fontsize=12)
>>> plt.show()
```


Decision tree
Bagging
Alcohol
4
3
2
1
11
12
13
14
15
OD280/OD315 of diluted wines

Leveraging weak learners via adaptive boosting

In this last section about ensemble methods, we will discuss **boosting** with a special focus on its most common implementation, **AdaBoost** (**Adaptive Boosting**).

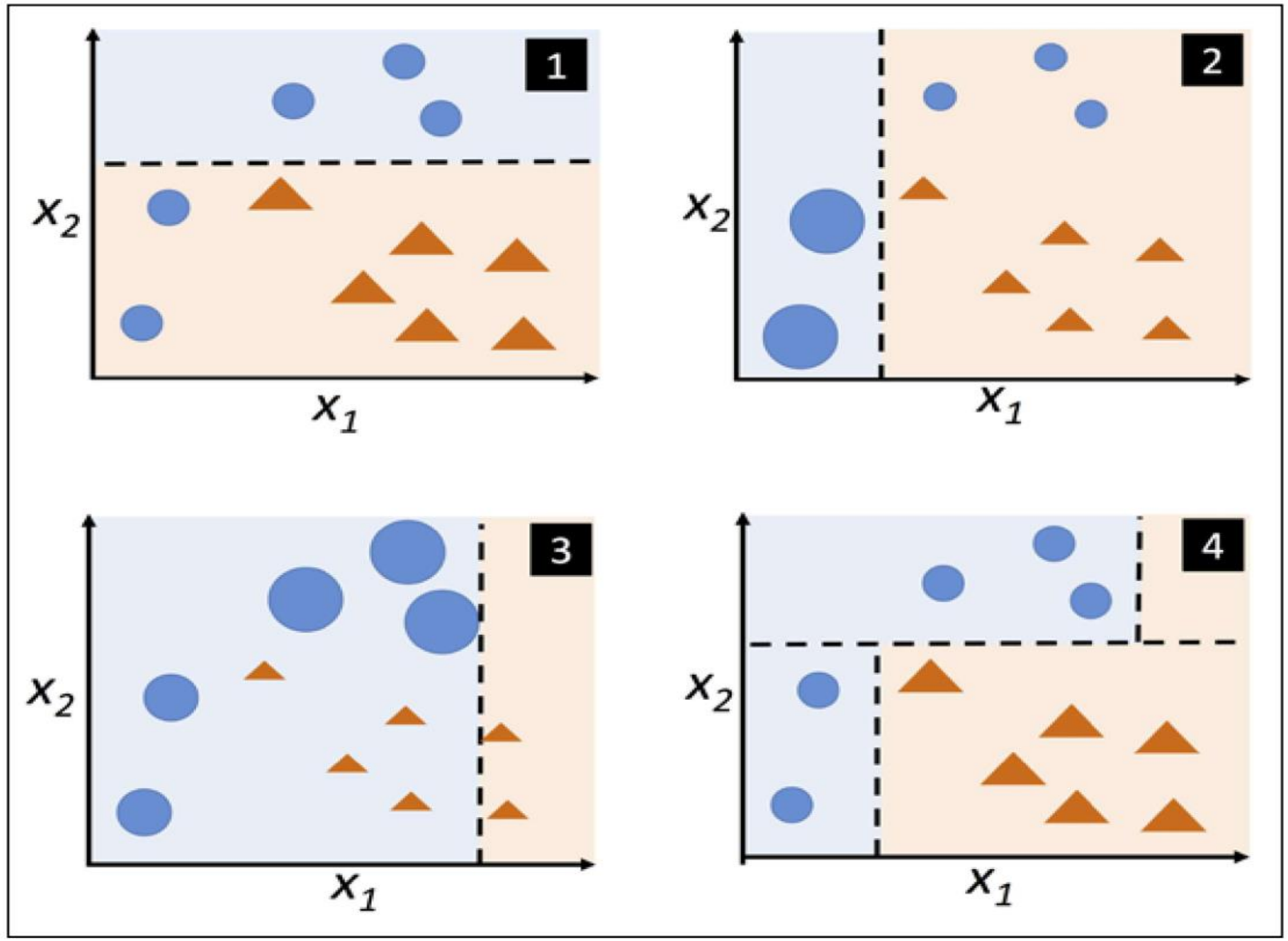
In boosting, the ensemble consists of very simple base classifiers, also often referred to as **weak learners**, which often only have a slight performance advantage over random guessing—a typical example of a weak learner is a decision tree stump. The key concept behind boosting is to focus on training samples that are hard to classify, that is, to let the weak learners subsequently learn from misclassified training samples to improve the performance of the ensemble.

How boosting works

In contrast to bagging, the initial formulation of boosting, the algorithm uses random subsets of training samples drawn from the training dataset without replacement; the original boosting procedure is summarized in the following four key steps:

1. Draw a random subset of training samples d_1 without replacement from training set D to train a weak learner C_1.
2. Draw a second random training subset d_2 without replacement from the training set and add 50 percent of the samples that were previously misclassified to train a weak learner C_2.
3. Find the training samples d_3 in training set D, which C_1 and C_2 disagree upon, to train a third weak learner C_3.
4. Combine the weak learners C_1, C_2, and C_3 via majority voting.

boosting can lead to a decrease in bias as well as variance compared to bagging models. In practice, however, boosting algorithms such as AdaBoost are also known for their high variance, that is, the tendency to overfit the training data


1
x_2
x_1
2
x_2
x_1
3
x_2
x_1
4
x_2
x_1

To walk through the AdaBoost illustration step by step, we start with subfigure **1**, which represents a training set for binary classification where all training samples are assigned equal weights. Based on this training set, we train a decision stump (shown as a dashed line) that tries to classify the samples of the two classes (triangles and circles), as well as possibly by minimizing the cost function (or the impurity score in the special case of decision tree ensembles).

For the next round (subfigure **2**), we assign a larger weight to the two previously misclassified samples (circles). Furthermore, we lower the weight of the correctly classified samples. The next decision stump will now be more focused on the training samples that have the largest weights—the training samples that are supposedly hard to classify. The weak learner shown in subfigure **2** misclassifies three different samples from the circle class, which are then assigned a larger weight, as shown in subfigure **3**.

Assuming that our AdaBoost ensemble only consists of three rounds of boosting, we would then combine the three weak learners trained on different reweighted training subsets by a weighted majority vote, as shown in subfigure **4**.

Now that have a better understanding behind the basic concept of AdaBoost, let's take a more detailed look at the algorithm using pseudo code. For clarity, we will denote element-wise multiplication by the cross symbol $(\times)$ and the dot-product between two vectors by a dot symbol $(\cdot)$:

1. Set the weight vector $\mathbf{w}$ to uniform weights, where $\sum_i w_i = 1$.
2. For j in m boosting rounds, do the following:
    a. Train a weighted weak learner: $C_j = \text{train}(\boldsymbol{X}, \boldsymbol{y}, \boldsymbol{w})$.
    b. Predict class labels: $\hat{y} = \text{predict}(C_j, \boldsymbol{X})$.
    c. Compute weighted error rate: $\varepsilon = \boldsymbol{w} \cdot (\hat{\boldsymbol{y}} \neq \boldsymbol{y})$.
    d. Compute coefficient: $\alpha_j = 0.5 \log \frac{1-\varepsilon}{\varepsilon}$.
    e. Update weights: $\boldsymbol{w} := \boldsymbol{w} \times \exp(-\alpha_j \times \hat{\boldsymbol{y}} \times \boldsymbol{y})$.
    f. Normalize weights to sum to 1: $\boldsymbol{w} := \boldsymbol{w} / \sum_i w_i$.
3. Compute the final prediction: $\hat{\boldsymbol{y}} = \left(\sum_{j=1}^{m} (\boldsymbol{\alpha}_j \times \text{predict}(C_j, \boldsymbol{X})) > 0\right)$.

Example

Sample indices	x	y	Weights	$\hat{y}$(x <= 3.0)?	Correct?	Updated weights
1	1.0	1	0.1	1	Yes	0.072
2	2.0	1	0.1	1	Yes	0.072
3	3.0	1	0.1	1	Yes	0.072
4	4.0	-1	0.1	-1	Yes	0.072
5	5.0	-1	0.1	-1	Yes	0.072
6	6.0	-1	0.1	-1	Yes	0.072
7	7.0	1	0.1	-1	No	0.167
8	8.0	1	0.1	-1	No	0.167
9	9.0	1	0.1	-1	No	0.167
10	10.0	-1	0.1	-1	Yes	0.072

$$\varepsilon = 0.1\times 0 + 0.1\times 0 + 0.1\times 0 + 0.1\times 0 + 0.1\times 0 + 0.1\times 0 + 0.1\times 1 + 0.1\times 1$$
$$+ 0.1\times 1 + 0.1\times 0 = \frac{3}{10} = 0.3$$

$$\alpha_j = 0.5 \log\left(\frac{1-\varepsilon}{\varepsilon}\right) \approx 0.424$$

After we have computed the coefficient α_j, we can now update the weight vector using the following equation:

$$0.1\times \exp(-0.424\times 1\times 1) \approx 0.065$$

Similarly, we will increase the ith weight if $\hat{y}_i$ predicted the label incorrectly, like this:

$$0.1\times \exp(-0.424\times 1\times (-1)) \approx 0.153$$

Alternatively, it's like this:

$$0.1\times \exp(-0.424\times (-1)\times (1)) \approx 0.153$$

Here, $\sum_i w_i = 7\times 0.065 + 3\times 0.153 = 0.914$.

$$\boldsymbol{w} := \frac{\boldsymbol{w}}{\sum_i w_i}$$

Thus, each weight that corresponds to a correctly classified sample will be reduced from the initial value of 0.1 to $0.065 / 0.914 \approx 0.071$ for the next round of boosting. Similarly, the weights of the incorrectly classified samples will increase from 0.1 to $0.153 / 0.914 \approx 0.167$.

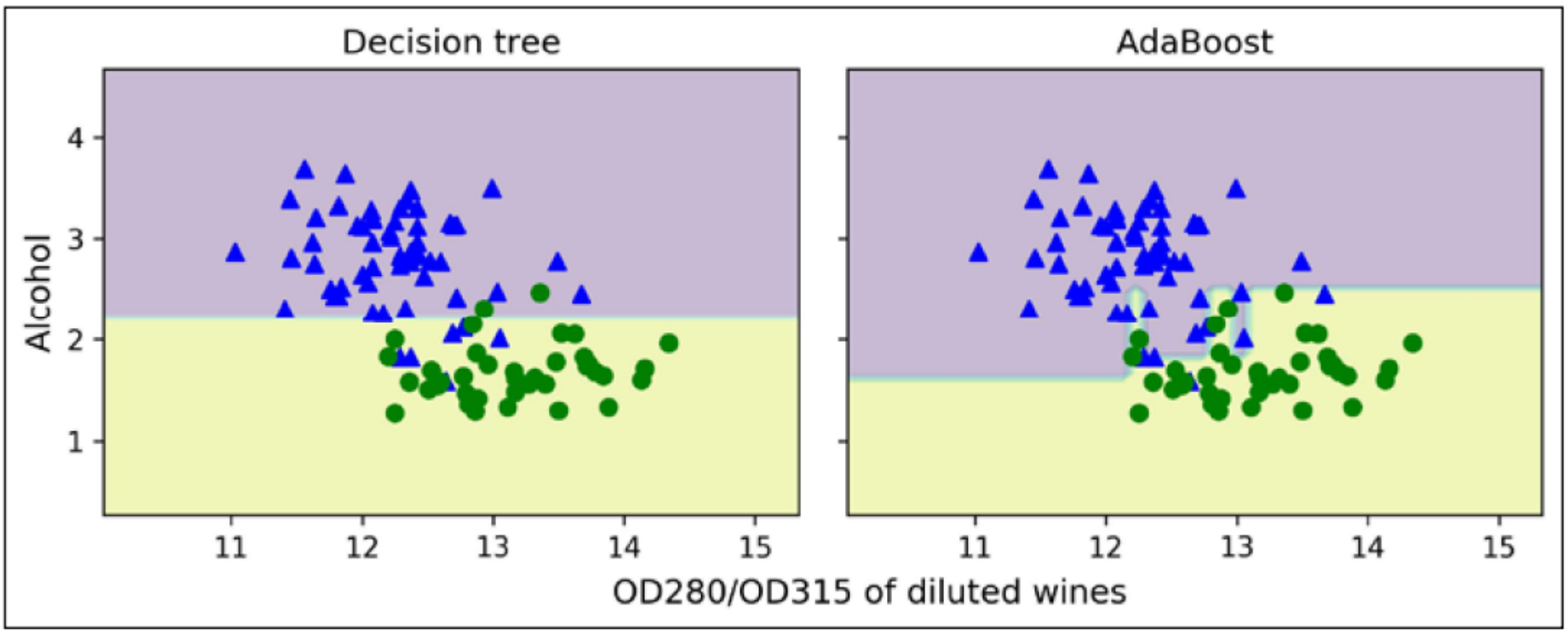
Applying AdaBoost using scikit-learn

```
>>> from sklearn.ensemble import AdaBoostClassifier
>>> tree = DecisionTreeClassifier(criterion='entropy',
                                  random_state=1,
...                               max_depth=1)
>>> ada = AdaBoostClassifier(base_estimator=tree,
...                          n_estimators=500,
...                          learning_rate=0.1,
...                          random_state=1)
>>> tree = tree.fit(X_train, y_train)
>>> y_train_pred = tree.predict(X_train)
>>> y_test_pred = tree.predict(X_test)
>>> tree_train = accuracy_score(y_train, y_train_pred)
>>> tree_test = accuracy_score(y_test, y_test_pred)
>>> print('Decision tree train/test accuracies %.3f/%.3f'
...       % (tree_train, tree_test))
Decision tree train/test accuracies 0.916/0.875
```

```
>>> ada = ada.fit(X_train, y_train)
>>> y_train_pred = ada.predict(X_train)
>>> y_test_pred = ada.predict(X_test)
>>> ada_train = accuracy_score(y_train, y_train_pred)
>>> ada_test = accuracy_score(y_test, y_test_pred)
>>> print('AdaBoost train/test accuracies %.3f/%.3f'
...       % (ada_train, ada_test))
AdaBoost train/test accuracies 1.000/0.917
```

```
>>> x_min = X_train[:, 0].min() - 1
>>> x_max = X_train[:, 0].max() + 1
>>> y_min = X_train[:, 1].min() - 1
>>> y_max = X_train[:, 1].max() + 1
>>> xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.1),
...                      np.arange(y_min, y_max, 0.1))
>>> f, axarr = plt.subplots(1, 2,
...                         sharex='col',
...                         sharey='row',
...                         figsize=(8, 3))
>>> for idx, clf, tt in zip([0, 1],
...                         [tree, ada],
...                         ['Decision Tree', 'AdaBoost']):
...     clf.fit(X_train, y_train)
...     Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])
...     Z = Z.reshape(xx.shape)
...     axarr[idx].contourf(xx, yy, Z, alpha=0.3)
...     axarr[idx].scatter(X_train[y_train==0, 0],
...                        X_train[y_train==0, 1],
...                        c='blue',
...                        marker='^')
...     axarr[idx].scatter(X_train[y_train==1, 0],
...                        X_train[y_train==1, 1],
...                        c='red',
...                        marker='o')
...     axarr[idx].set_title(tt)
```

```
...       axarr[0].set_ylabel('Alcohol', fontsize=12)
>>> plt.text(10.2, -0.5,
...              s='OD280/OD315 of diluted wines',
...              ha='center',
...              va='center',
...              fontsize=12)
>>> plt.show()
```



Thank You !!!