Lecture 10 - Ensemble Learning

Overview

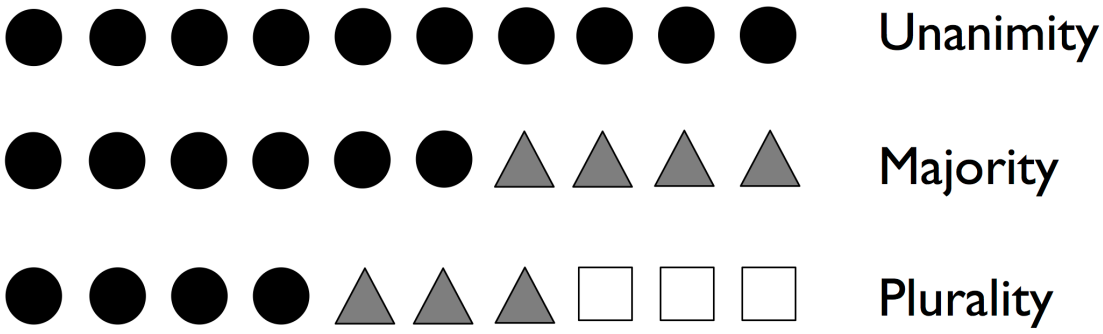
[1]:
```
from IPython.display import Image
%matplotlib inline
```

Learning with ensembles

The goal of ensemble methods is to combine different classifiers (not necessarily the same classification model) into a meta-classifier that has better generalisation performance than each individual classifier alone.

[2]:
```
Image(filename='figures/07_01.png', width=500)
```

[2]:



In **majority voting** we select the class label that has been predicted by the majority of classifiers, that is, received more than 50 percent of the votes.

Majority voting principle can be generalised to multiclass settings, which is known as **plurality voting**.

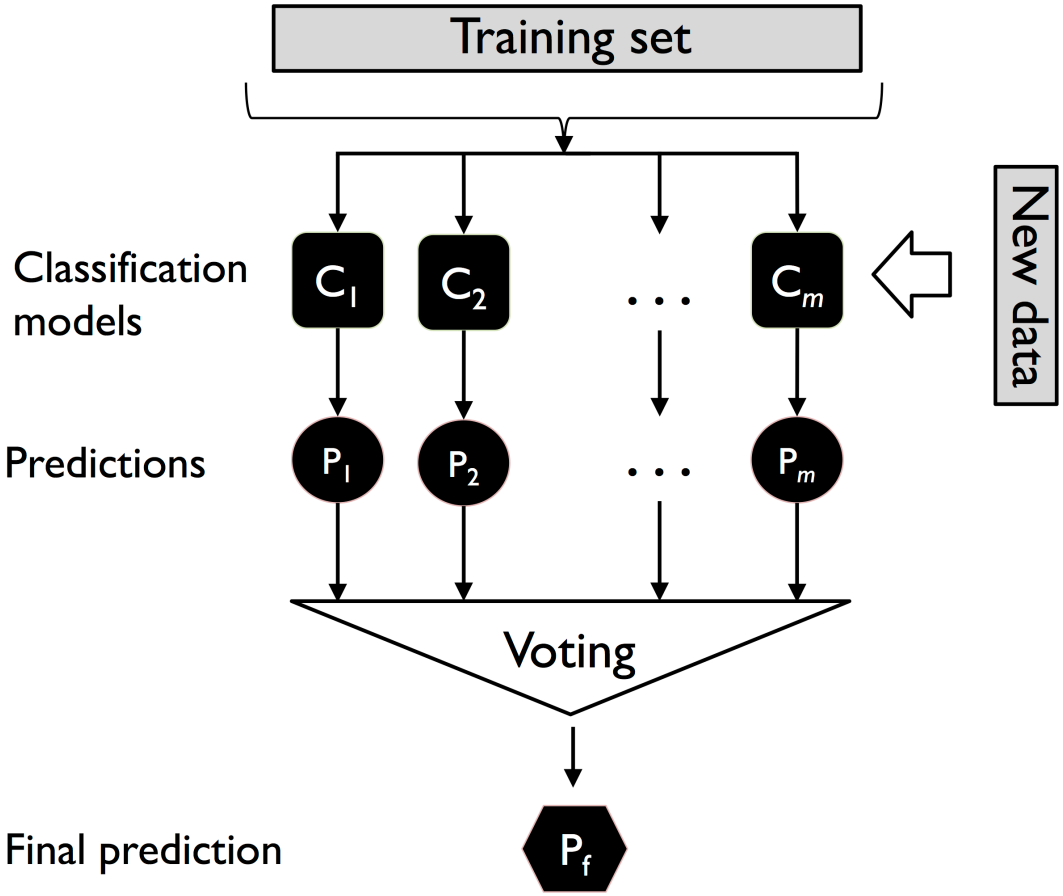
Using the training dataset, we start by training m different classifiers $(C_1, \dots, C_m)$.

Depending on the technique, the ensemble can be built from different classification algorithms, for example, decision trees, support vector machines, logistic regression classifiers, etc.

Alternatively, we can also use the same base classification algorithm, fitting different subsets of the training dataset, for example, random forest algorithm which combines different decision tree classifiers.

[3]:
```
Image(filename='figures/07_02.png', width=500)
```

[3]:



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:

$$\hat{y} = \text{mode}\{C_1(\mathbf{x}), C_2(\mathbf{x}), \dots, C_m(\mathbf{x})\}.$$

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

Why ensemble methods can work better than individual classifiers alone?

Suppose that all m base classifiers $C_1, \ldots, C_m$ for a **binary classification** task have an equal error rate of ϵ. Moreover, suppose that the classifiers are independent and the error rates are not correlated.

Then, the error probability of an ensemble of m base classifiers can be written as a probability mass function of a binomial distribution:

$$\mathbb{P}(y \geq k) = \sum_{i=k}^{m} \binom{m}{i} \epsilon^i (1-\epsilon)^{m-i} = \epsilon_{\text{ensemble}}.$$

$\binom{m}{i}$ is number of ways we can choose subsets of i unordered elements from a set of size m. It is given as

$$\binom{m}{i} = \frac{m!}{(m-i)!i!}.$$

The symbol ! stands for factorial and it is defined as $m! := m \times (m-1) \times (m-2) \times \ldots \times 1$. For example, $3! = 3 \times 2 \times 1 = 6$.

Now for a collection of 11 base classifiers ($m = 11$), where each classifier has an error rate of $\epsilon = 0.25$, the ensemble error based on majority voting (we are doing binary classification) would be given the sum of errors when at least half of the base classifiers make the same prediction. Thus, for $m = 11$, we would require a collection of at least 6 base classifiers making the same prediction for our ensemble to make the same prediction.

Thus, ensemble error is $\mathbb{P}(y \geq 6) = \sum_{i=6}^{11} \binom{11}{i} 0.25^i 0.75^{11-i} = 0.034$.

Clearly, 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.

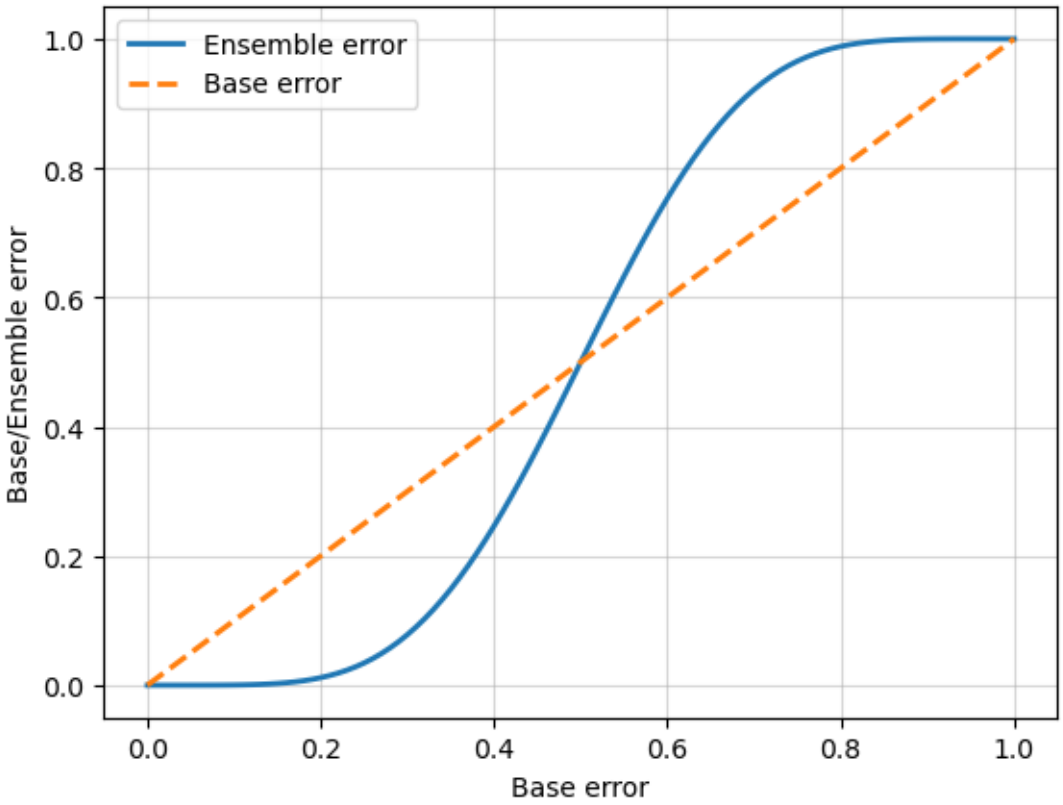
[4]:
```
from scipy.special import comb
import math


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)
```

[5]:
```
ensemble_error(n_classifier=11, error=0.25)
```

[5]:
```
0.03432750701904297
```

[6]:
```
import numpy as np


error_range = np.arange(0.0, 1.01, 0.01)
ens_errors = [ensemble_error(n_classifier=11, error=error)
              for error in error_range]
```

[7]:
```
import matplotlib.pyplot as plt


plt.plot(error_range,
         ens_errors,
         label='Ensemble error',
         linewidth=2)

plt.plot(error_range,
         error_range,
         linestyle='--',
         label='Base error',
         linewidth=2)

plt.xlabel('Base error')
plt.ylabel('Base/Ensemble error')
plt.legend(loc='upper left')
plt.grid(alpha=0.5)
plt.show()
```



The error probability of an ensemble is always better than the error of an individual base classifier, as long as the base classifiers perform better than random guessing ($\epsilon < 0.5$).

Combining classifiers via majority vote

Note that in the multiclass setting, we will use **plurality voting**, but we only use the term **"majority voting"**. This is common in the literature.

Implementing a simple majority vote classifier

We can build a strong meta-classifier which balances out the individual classifier's weaknesses on a particular datastet by putting a weight on individual performance to quantify our confidence.

More precisely, we create a **weighted majority classifier** as

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

Here,

- w_j is a weight associated with a base classifier C_j
- $\hat{y}$ is the predicted class label of the ensemble

- A is the set of unique class labels
- χ_A is the indicator function which returns 1 is the predicted class of the jth classifier matches i (C_j($\mathbf{x}$) = i)

If the weights are equal, we have

$$\hat{y} = \text{mode}\{C_1(\mathbf{x}), C_2(\mathbf{x}), \ldots, C_m(\mathbf{x})\}.$$

Let us assume that we have an ensemble of three base classifiers, $C_j(j \in \{1, 2, 3\})$, and we want to predict the class label, $C_j \in \{0, 1\}$, of a given example, $\mathbf{x}$.

Two out of three base classifiers predict the class label 0, and one, C_3, predicts that the example belongs to class 1.

If we put equal weights on each base classifier prediction, we get the majority vote as

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

If we assign a weight of 0.6 to C_3, and a weight of 0.2 to each C_1, C_2, we get

$$\hat{y} = \arg\max_i \sum_{j=1}^{3} w_j \chi_A(C_j(\mathbf{x}) = i) \tag{1}$$

$$= \arg\max_i \{0.2 \times i_0 + 0.2 \times i_0, 0.6 \times i_1\} = 1. \tag{2}$$

Alternatively, we can say that the prediction made by C_3 has three times more weight than the predictions by C_1 or C_2, then we can write

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

Weighted majority vote in NumPy

- `bincount` counts the number of occurrences of each class label
- `argmax` function returns the index position of the highest count, corresponding to the majority class label (this assumes that class labels start at 0)

[60]:
```
import numpy as np

np.argmax(np.bincount([0, 0, 1],
                      weights=[0.2, 0.2, 0.6]))
```

[60]:
```
1
```

Using the predicted class probabilities instead of the class labels for majority voting can be more useful. The modified version of the majority vote for predicting class labels from their probabilities can be written as

$$\hat{y} = \arg\max_i \sum_{j=1}^{m} w_j p_{ij},$$

where p_{ij} is the predicted probability of the jth classifier for class label i.

[8]:
```
ex = np.array([[0.9, 0.1],
               [0.8, 0.2],
               [0.4, 0.6]])

p = np.average(ex,
               axis=0,
               weights=[0.2, 0.2, 0.6])
p
```

[8]:
```
array([0.58, 0.42])
```

[12]:
```
np.argmax(p)
```

[12]:
```
0
```

[19]:
```
from sklearn.base import BaseEstimator
from sklearn.base import ClassifierMixin
from sklearn.preprocessing import LabelEncoder
from sklearn.base import clone
from sklearn.pipeline import _name_estimators
import numpy as np
import operator


class MajorityVoteClassifier(BaseEstimator,
                             ClassifierMixin):
    """ A majority vote ensemble classifier

    Parameters
    ----------
    classifiers : array-like, shape = [n_classifiers]
      Different classifiers for the ensemble

    vote : str, {'classlabel', 'probability'} (default='classlabel')
      If 'classlabel' the prediction is based on the argmax of
        class labels. Else if 'probability', the argmax of
        the sum of probabilities is used to predict the class label
        (recommended for calibrated classifiers).

    weights : array-like, shape = [n_classifiers], optional (default=None)
      If a list of `int` or `float` values are provided, the classifiers
      are weighted by importance; Uses uniform weights if `weights=None`.
```

```
    """
    def __init__(self, classifiers, vote='classlabel', weights=None):

        self.classifiers = classifiers
        self.named_classifiers = {key: value for key, value
                                  in _name_estimators(classifiers)}
        self.vote = vote
        self.weights = weights

    def fit(self, X, y):
        """ Fit classifiers.

        Parameters
        ----------
        X : {array-like, sparse matrix}, shape = [n_examples, n_features]
            Matrix of training examples.

        y : array-like, shape = [n_examples]
            Vector of target class labels.

        Returns
        -------
        self : object

        """
        if self.vote not in ('probability', 'classlabel'):
            raise ValueError(f"vote must be 'probability' or 'classlabel'"
                             f"; got (vote={self.vote})")

        if self.weights and len(self.weights) != len(self.classifiers):
            raise ValueError(f'Number of classifiers and weights must be equal'
                             f'; got {len(self.weights)} weights,'
                             f' {len(self.classifiers)} classifiers')

        # Use LabelEncoder to ensure class labels start with 0, which
        # is important for np.argmax call in self.predict
        self.lablenc_ = LabelEncoder()
        self.lablenc_.fit(y)
        self.classes_ = self.lablenc_.classes_
        self.classifiers_ = []
        for clf in self.classifiers:
            fitted_clf = clone(clf).fit(X, self.lablenc_.transform(y))
            self.classifiers_.append(fitted_clf)
        return self

    def predict(self, X):
```

```
        """ Predict class labels for X.

        Parameters
        ----------
        X : {array-like, sparse matrix}, shape = [n_examples, n_features]
            Matrix of training examples.

        Returns
        ----------
        maj_vote : array-like, shape = [n_examples]
            Predicted class labels.

        """
        if self.vote == 'probability':
            maj_vote = np.argmax(self.predict_proba(X), axis=1)
        else:  # 'classlabel' vote

            #  Collect results from clf.predict calls
            predictions = np.asarray([clf.predict(X)
                                      for clf in self.classifiers_]).T

            maj_vote = np.apply_along_axis(
                                      lambda x:
                                      np.argmax(np.bincount(x,
                                                weights=self.weights)),
                                      axis=1,
                                      arr=predictions)
        maj_vote = self.lablenc_.inverse_transform(maj_vote) #reverses the␣
↪LabelEncoder labels
        return maj_vote

    def predict_proba(self, X):
        """ Predict class probabilities for X.

        Parameters
        ----------
        X : {array-like, sparse matrix}, shape = [n_examples, n_features]
            Training vectors, where n_examples is the number of examples and
            n_features is the number of features.

        Returns
        ----------
        avg_proba : array-like, shape = [n_examples, n_classes]
            Weighted average probability for each class per example.

        """
        probas = np.asarray([clf.predict_proba(X)
```

```
                                for clf in self.classifiers_])
        avg_proba = np.average(probas, axis=0, weights=self.weights)
        return avg_proba

    def get_params(self, deep=True):
        """ Get classifier parameter names for GridSearch"""
        if not deep:
            return super().get_params(deep=False)
        else:
            out = self.named_classifiers.copy()
            for name, step in self.named_classifiers.items():
                for key, value in step.get_params(deep=True).items():
                    out[f'{name}__{key}'] = value
            return out
```

In the above, we have the function `predict_proba` which returns the averaged probabilities. These are useful when computing the **receiver operating characteristic area under the curve (ROC AUC)**.

In the above we `get_params` method to use the `_name_estimators` function to access the parameters of individual classifiers in the ensemble. This will be useful in grid search for **hyperparameter tuning**.

A more sophisticated version of the function above is available via `sklearn.ensemble.VotingClassifier`.

Using the majority voting principle to make predictions

We use two features, *sepal width* and *petal length* in the Iris dataset.

`MajorityVoteClassifier` generalises to multiclass problems but we will only classify flower examples from the `Iris-versicolor` and `Iris-virginica` classes, with which we will compute the **ROC AUC** later.

[10]:
```
from sklearn import datasets
from sklearn.preprocessing import StandardScaler
from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import train_test_split


iris = datasets.load_iris()
X, y = iris.data[50:, [1, 2]], iris.target[50:]
le = LabelEncoder()
y = le.fit_transform(y)

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 neighbours

We will 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.

[25]:
```
import numpy as np
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.svm import SVC

from sklearn.pipeline import Pipeline
from sklearn.model_selection import cross_val_score


clf1 = LogisticRegression(penalty='l2',
                          C=0.001,
                          solver='lbfgs',
                          random_state=1)

clf2 = DecisionTreeClassifier(max_depth=1,
                              criterion='entropy',
                              random_state=0)

clf3 = KNeighborsClassifier(n_neighbors=1,
                            p=2,
                            metric='minkowski')

# clf3 = SVC(kernel='rbf', random_state=1, gamma=0.2, C=1.0)


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(f'ROC AUC: {scores.mean():.2f} '
          f'(+/- {scores.std():.2f}) [{label}]')
```

```
10-fold cross validation:

ROC AUC: 0.92 (+/- 0.15) [Logistic regression]
ROC AUC: 0.87 (+/- 0.18) [Decision tree]
ROC AUC: 0.85 (+/- 0.13) [kNN]
```

Thus, the predictive performance are almost equal. We train the logistic regression and k-nearest neighbors classifier as part of a pipeline since both of them are not scale-invariant, in contrast to decision trees.

Next, we combine the individual classifiers for majority rule voting in our `MajorityVoteClassifier`.

[26]:
```
# Majority Rule (hard) Voting

mv_clf = MajorityVoteClassifier(classifiers=[pipe1, clf2, pipe3])

clf_labels += ['Majority voting']
all_clf = [pipe1, clf2, pipe3, mv_clf]

for clf, label in zip(all_clf, clf_labels):
    scores = cross_val_score(estimator=clf,
                             X=X_train,
                             y=y_train,
                             cv=10,
                             scoring='roc_auc')
    print(f'ROC AUC: {scores.mean():.2f} '
          f'(+/- {scores.std():.2f}) [{label}]')
```

```
ROC AUC: 0.92 (+/- 0.15) [Logistic regression]
ROC AUC: 0.87 (+/- 0.18) [Decision tree]
ROC AUC: 0.85 (+/- 0.13) [kNN]
ROC AUC: 0.98 (+/- 0.05) [Majority voting]
```

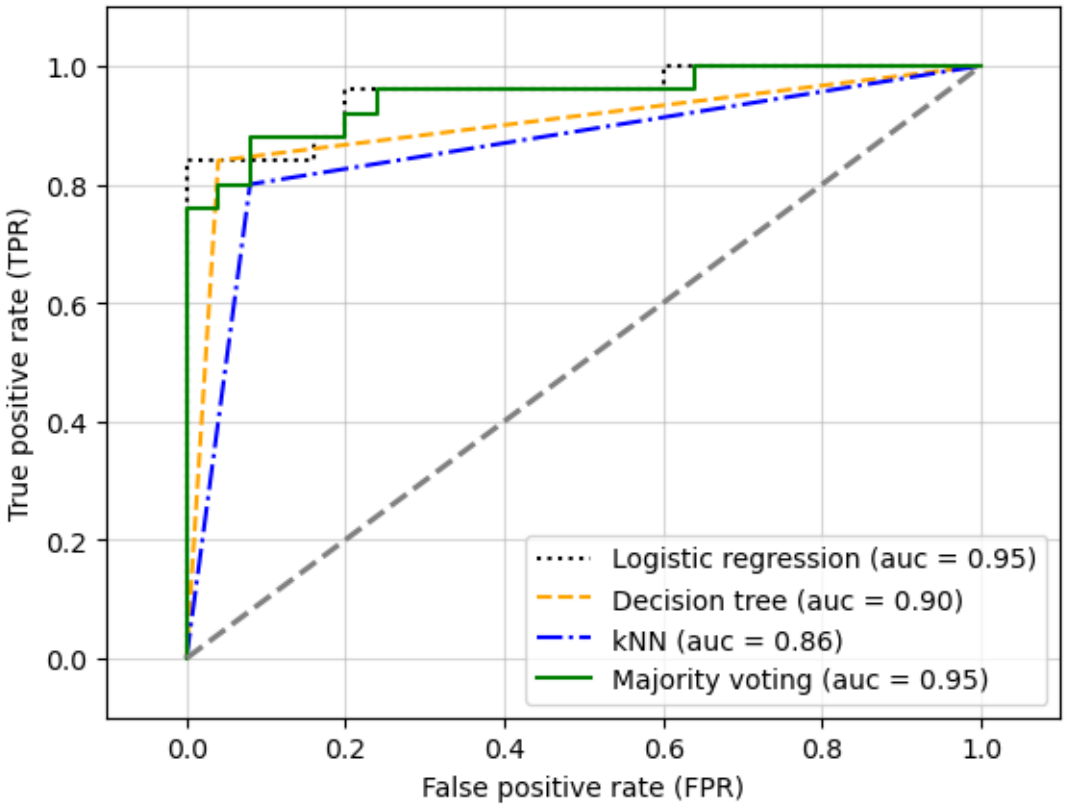
Before, we create the ensemble classifier, we perform hyperparameter tuning in the three classifiers using **10-fold cross-valiation**. We have not covered hyperparameter tuning in this course, but it is a technique which can significantly boost the performance of a machine learning classifier.

Evaluating and tuning the ensemble classifier

We compute the ROC curves from the test dataset to check that `MajorityVoteClassifier` generalises well with unseen data.

[27]:
```
from sklearn.metrics import roc_curve
from sklearn.metrics import auc
```

```
colors = ['black', 'orange', 'blue', 'green']
linestyles = [':', '--', '-.', '-']
for clf, label, clr, ls \
        in zip(all_clf,
               clf_labels, colors, linestyles):

    # assuming the label of the positive class is 1
    y_pred = clf.fit(X_train,
                     y_train).predict_proba(X_test)[:, 1]
    fpr, tpr, thresholds = roc_curve(y_true=y_test,
                                     y_score=y_pred)
    roc_auc = auc(x=fpr, y=tpr)
    plt.plot(fpr, tpr,
             color=clr,
             linestyle=ls,
             label=f'{label} (auc = {roc_auc:.2f})')

plt.legend(loc='lower right')
plt.plot([0, 1], [0, 1],
         linestyle='--',
         color='gray',
         linewidth=2)

plt.xlim([-0.1, 1.1])
plt.ylim([-0.1, 1.1])
plt.grid(alpha=0.5)
plt.xlabel('False positive rate (FPR)')
plt.ylabel('True positive rate (TPR)')


plt.show()
```

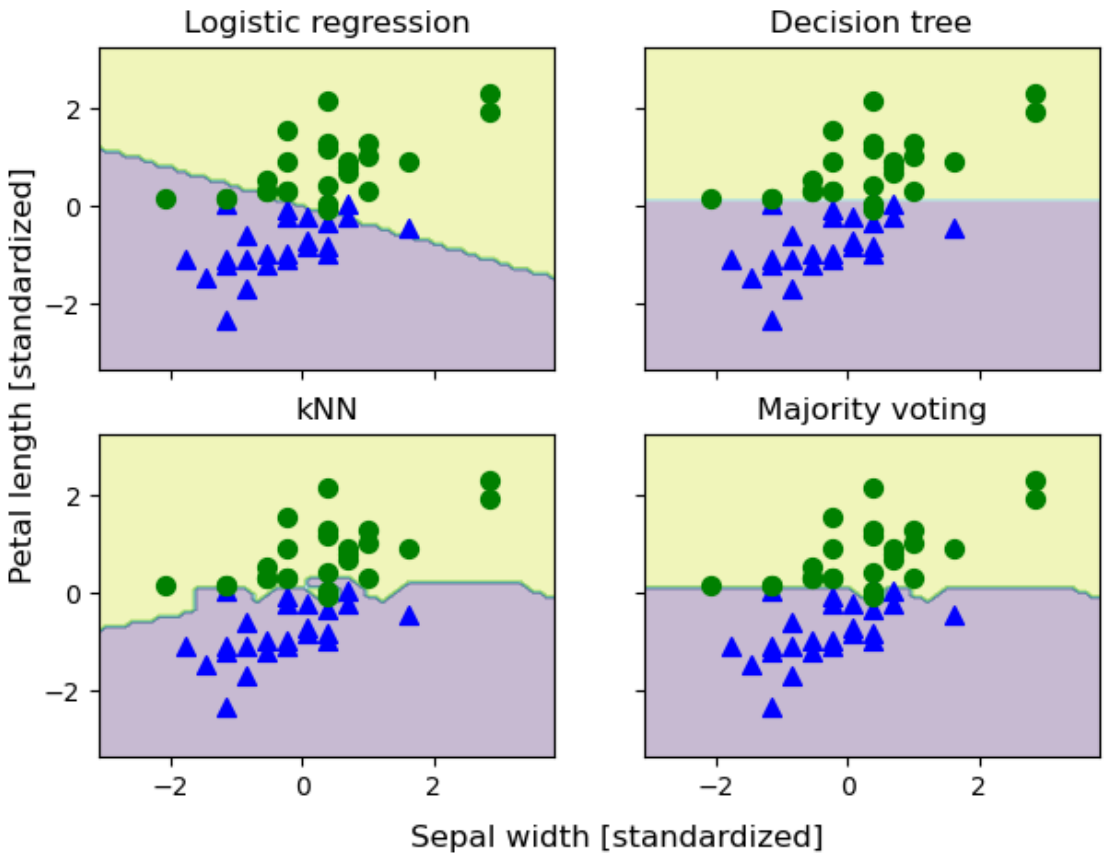


As you can see in the resulting ROC, the ensemble classifier also performs well on the test dataset (ROC AUC = 0.95). However, you can see that the logistic regression classifier performs similarly well on the same dataset (dataset is quite small).

[29]:
```
sc = StandardScaler()
X_train_std = sc.fit_transform(X_train)
```

[30]:
```
from itertools import product


all_clf = [pipe1, clf2, pipe3, mv_clf]

x_min = X_train_std[:, 0].min() - 1
x_max = X_train_std[:, 0].max() + 1
y_min = X_train_std[:, 1].min() - 1
y_max = X_train_std[:, 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=2, ncols=2,
```

```
                        sharex='col',
                        sharey='row',
                        figsize=(7, 5))

for idx, clf, tt in zip(product([0, 1], [0, 1]),
                        all_clf, clf_labels):
    clf.fit(X_train_std, y_train)

    Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])
    Z = Z.reshape(xx.shape)

    axarr[idx[0], idx[1]].contourf(xx, yy, Z, alpha=0.3)

    axarr[idx[0], idx[1]].scatter(X_train_std[y_train==0, 0],
                                  X_train_std[y_train==0, 1],
                                  c='blue',
                                  marker='^',
                                  s=50)

    axarr[idx[0], idx[1]].scatter(X_train_std[y_train==1, 0],
                                  X_train_std[y_train==1, 1],
                                  c='green',
                                  marker='o',
                                  s=50)

    axarr[idx[0], idx[1]].set_title(tt)

plt.text(-3.5, -5.,
         s='Sepal width [standardized]',
         ha='center', va='center', fontsize=12)
plt.text(-12.5, 4.5,
         s='Petal length [standardized]',
         ha='center', va='center',
         fontsize=12, rotation=90)

#plt.savefig('figures/07_05', dpi=300)
plt.show()
```



Before we tune the individual classifier's parameters for ensemble classification, let us call the `get_params` method to get a basic idea of how we can access the individual parameters inside a `GridSearchCV` object.

[31]:
```
mv_clf.get_params()
```

[31]:
```
{'pipeline-1': Pipeline(steps=[('sc', StandardScaler()),
                               ['clf', LogisticRegression(C=0.001, random_state=1)]]),
 'decisiontreeclassifier': DecisionTreeClassifier(criterion='entropy',
max_depth=1, random_state=0),
 'pipeline-2': Pipeline(steps=[('sc', StandardScaler()),
                               ['clf', KNeighborsClassifier(n_neighbors=1)]]),
 'pipeline-1__memory': None,
 'pipeline-1__steps': [('sc', StandardScaler()),
  ['clf', LogisticRegression(C=0.001, random_state=1)]],
 'pipeline-1__verbose': False,
 'pipeline-1__sc': StandardScaler(),
 'pipeline-1__clf': LogisticRegression(C=0.001, random_state=1),
 'pipeline-1__sc__copy': True,
 'pipeline-1__sc__with_mean': True,
```

```
 'pipeline-1__sc__with_std': True,
 'pipeline-1__clf__C': 0.001,
 'pipeline-1__clf__class_weight': None,
 'pipeline-1__clf__dual': False,
 'pipeline-1__clf__fit_intercept': True,
 'pipeline-1__clf__intercept_scaling': 1,
 'pipeline-1__clf__l1_ratio': None,
 'pipeline-1__clf__max_iter': 100,
 'pipeline-1__clf__multi_class': 'auto',
 'pipeline-1__clf__n_jobs': None,
 'pipeline-1__clf__penalty': 'l2',
 'pipeline-1__clf__random_state': 1,
 'pipeline-1__clf__solver': 'lbfgs',
 'pipeline-1__clf__tol': 0.0001,
 'pipeline-1__clf__verbose': 0,
 'pipeline-1__clf__warm_start': False,
 'decisiontreeclassifier__ccp_alpha': 0.0,
 'decisiontreeclassifier__class_weight': None,
 'decisiontreeclassifier__criterion': 'entropy',
 'decisiontreeclassifier__max_depth': 1,
 'decisiontreeclassifier__max_features': None,
 'decisiontreeclassifier__max_leaf_nodes': None,
 'decisiontreeclassifier__min_impurity_decrease': 0.0,
 'decisiontreeclassifier__min_samples_leaf': 1,
 'decisiontreeclassifier__min_samples_split': 2,
 'decisiontreeclassifier__min_weight_fraction_leaf': 0.0,
 'decisiontreeclassifier__random_state': 0,
 'decisiontreeclassifier__splitter': 'best',
 'pipeline-2__memory': None,
 'pipeline-2__steps': [('sc', StandardScaler()),
  ['clf', KNeighborsClassifier(n_neighbors=1)]],
 'pipeline-2__verbose': False,
 'pipeline-2__sc': StandardScaler(),
 'pipeline-2__clf': KNeighborsClassifier(n_neighbors=1),
 'pipeline-2__sc__copy': True,
 'pipeline-2__sc__with_mean': True,
 'pipeline-2__sc__with_std': True,
 'pipeline-2__clf__algorithm': 'auto',
 'pipeline-2__clf__leaf_size': 30,
 'pipeline-2__clf__metric': 'minkowski',
 'pipeline-2__clf__metric_params': None,
 'pipeline-2__clf__n_jobs': None,
 'pipeline-2__clf__n_neighbors': 1,
 'pipeline-2__clf__p': 2,
 'pipeline-2__clf__weights': 'uniform'}
```

Based on the values returned by the `get_params` method, we now know how to access the individual classifier's attributes. Recall that `pipeline-1__clf__C` is the inverse of regularisation parameter

in logistic regression.

[32]:
```
from sklearn.model_selection import GridSearchCV


params = {'decisiontreeclassifier__max_depth': [1, 2],
          'pipeline-1__clf__C': [0.001, 0.1, 100.0]}

grid = GridSearchCV(estimator=mv_clf,
                    param_grid=params,
                    cv=10,
                    scoring='roc_auc')
grid.fit(X_train, y_train)

for r, _ in enumerate(grid.cv_results_['mean_test_score']):
    mean_score = grid.cv_results_['mean_test_score'][r]
    std_dev = grid.cv_results_['std_test_score'][r]
    params = grid.cv_results_['params'][r]
    print(f'{mean_score:.3f} +/- {std_dev:.2f} {params}')
```

```
0.983 +/- 0.05 {'decisiontreeclassifier__max_depth': 1, 'pipeline-1__clf__C':
0.001}
0.983 +/- 0.05 {'decisiontreeclassifier__max_depth': 1, 'pipeline-1__clf__C':
0.1}
0.967 +/- 0.10 {'decisiontreeclassifier__max_depth': 1, 'pipeline-1__clf__C':
100.0}
0.983 +/- 0.05 {'decisiontreeclassifier__max_depth': 2, 'pipeline-1__clf__C':
0.001}
0.983 +/- 0.05 {'decisiontreeclassifier__max_depth': 2, 'pipeline-1__clf__C':
0.1}
0.967 +/- 0.10 {'decisiontreeclassifier__max_depth': 2, 'pipeline-1__clf__C':
100.0}
```

[33]:
```
print(f'Best parameters: {grid.best_params_}')
print(f'ROC AUC: {grid.best_score_:.2f}')
```

```
Best parameters: {'decisiontreeclassifier__max_depth': 1, 'pipeline-1__clf__C':
0.001}
ROC AUC: 0.98
```

Note

By default, the default setting for `refit` in `GridSearchCV` is `True` (i.e., `GridSeachCV(..., refit=True)`), which means that we can use the fitted `GridSearchCV` estimator to make predictions via the `predict` method, for example:

```
grid = GridSearchCV(estimator=mv_clf,
                    param_grid=params,
                    cv=10,
                    scoring='roc_auc')
grid.fit(X_train, y_train)
```

```
y_pred = grid.predict(X_test)
```

In addition, the "best" estimator can directly be accessed via the `best_estimator_` attribute.

[34]:
```
grid.best_estimator_.classifiers
```

[34]:
```
[Pipeline(steps=[('sc', StandardScaler()),
                 ['clf', LogisticRegression(C=0.001, random_state=1)]]),
 DecisionTreeClassifier(criterion='entropy', max_depth=1, random_state=0),
 Pipeline(steps=[('sc', StandardScaler()),
                 ['clf', KNeighborsClassifier(n_neighbors=1)]])]
```

[35]:
```
mv_clf = grid.best_estimator_
```

[36]:
```
mv_clf.set_params(**grid.best_estimator_.get_params())
```

[36]:
```
MajorityVoteClassifier(classifiers=[Pipeline(steps=[('sc', StandardScaler()),
                                                    ('clf',
                                                     LogisticRegression(C=0.001,
random_state=1))]),
                                    DecisionTreeClassifier(criterion='entropy',
                                                           max_depth=1,
                                                           random_state=0),
                                    Pipeline(steps=[('sc', StandardScaler()),
                                                    ('clf',
KNeighborsClassifier(n_neighbors=1))])])
```

[37]:
```
mv_clf
```

[37]:
```
MajorityVoteClassifier(classifiers=[Pipeline(steps=[('sc', StandardScaler()),
                                                    ('clf',
                                                     LogisticRegression(C=0.001,
random_state=1))]),
                                    DecisionTreeClassifier(criterion='entropy',
                                                           max_depth=1,
                                                           random_state=0),
                                    Pipeline(steps=[('sc', StandardScaler()),
                                                    ('clf',
KNeighborsClassifier(n_neighbors=1))])])
```

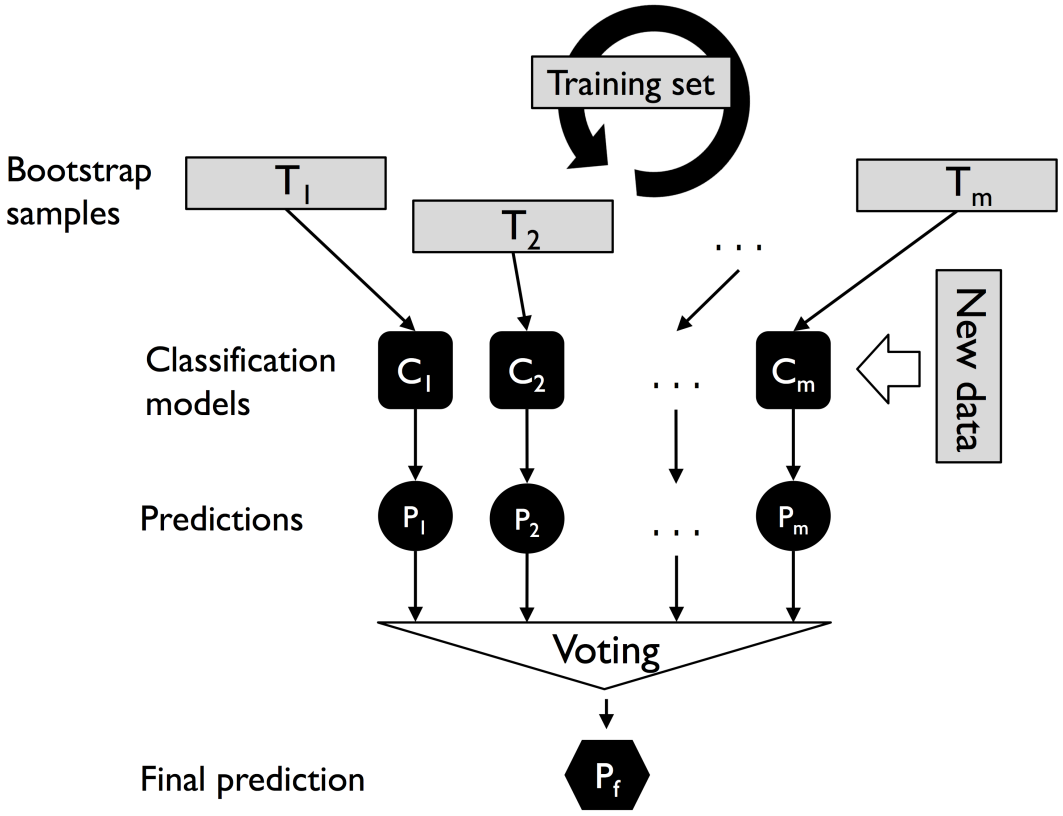
Bagging – Building an ensemble of classifiers from bootstrap samples

Bagging is an ensemble learning technique that is closely related to the `MajorityVoteClassifier`.

However, instead of using the same training dataset to fit the individual classifiers in the ensemble, we draw bootstrap samples (random samples with replacement) from the initial training dataset.

Thus, **bagging** is also known as **bootstrap aggregating**.

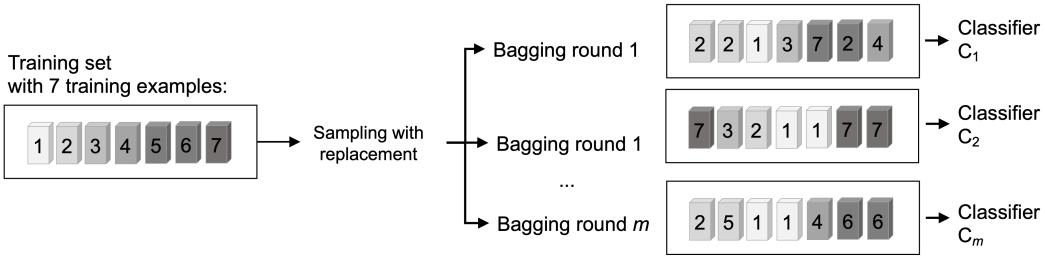
[38]:
```
Image(filename='./figures/07_06.png', width=500)
```
[38]:



Bagging in a nutshell

Typical classifier used in bagging is a decision tree.

[28]:
```
Image(filename='./figures/07_07.png', width=800)
```
[28]:



Once the individual classifiers are fitted to the bootstrap samples, the predictions are combined

using majority voting. Random forests are a special case of bagging where we also use random feature subsets when fitting the individual decision trees.

Applying bagging to classify examples in the Wine dataset

Here, we use the Wine dataset and only consider the Wine classes 2 and 3, and select two features – `Alcohol` and `OD280/OD315 of diluted wines`.

[46]:
```
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']

# if the Wine dataset is temporarily unavailable from the
# UCI machine learning repository, un-comment the following line
# of code to load the dataset from a local path:

# df_wine = pd.read_csv('wine.data', header=None)

# 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
```

[47]:
```
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)
```

A `BaggingClassifier` algorithm is already implemented in `scikit-learn`.

Here, we use an unpruned decision tree as the base classifier and create an ensemble of 500 decision

trees fit on different bootstrap samples of the training dataset.

[48]:
```
from sklearn.ensemble import BaggingClassifier
from sklearn.tree import DecisionTreeClassifier

tree = DecisionTreeClassifier(criterion='entropy',
                              max_depth=None,
                              random_state=1)

bag = BaggingClassifier(estimator=tree,
                        n_estimators=500,
                        max_samples=1.0,
                        max_features=1.0,
                        bootstrap=True,
                        bootstrap_features=False,
                        n_jobs=1,
                        random_state=1)
```

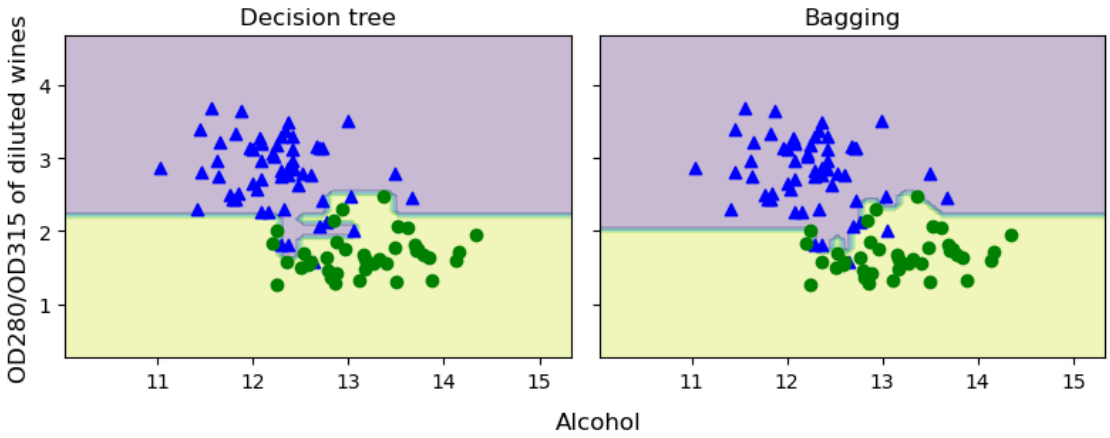
We compare the performance of the bagging classifier to the performance of a single unpruned decision tree.

[49]:
```
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(f'Decision tree train/test accuracies '
      f'{tree_train:.3f}/{tree_test:.3f}')

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(f'Bagging train/test accuracies '
      f'{bag_train:.3f}/{bag_test:.3f}')
```

```
Decision tree train/test accuracies 1.000/0.833
Bagging train/test accuracies 1.000/0.917
```

[50]:
```
import numpy as np
import matplotlib.pyplot as plt
```

```
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('OD280/OD315 of diluted wines', fontsize=12)

plt.tight_layout()
plt.text(0, -0.2,
         s='Alcohol',
         ha='center',
         va='center',
         fontsize=12,
         transform=axarr[1].transAxes)

plt.show()
```



In practice, more complex classification tasks and a dataset's high dimensionality can easily lead to overfitting in single decision trees, and this is where the bagging algorithm can really play to its strengths.

Bagging algorithm can be an effective approach to **reducing the variance of a model** (**reducing overfitting**).

However, bagging is ineffective in reducing model bias, that is, models that are too simple to capture the trends in the data well. This is why we want to perform bagging on an ensemble of classifiers with low bias, for example, unpruned decision trees.

Leveraging weak learners via adaptive boosting

The most common implementation of **boosting** approach is **Adaptive Boosting (AdaBoost)**.

How boosting works

In boosting, the ensemble consists of very simple base classifiers, called **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** (tree of depth 1: one parent, two child nodes).

The **key concept** behind boosting is to focus on training examples that are hard to classify, that is, to let the weak learners subsequently learn from misclassified training examples to improve the performance of the ensemble.

In contrast to bagging, the initial formulation of the boosting algorithm uses random subsets of training examples drawn from the training dataset **without replacement**. Boosting procedure is summarised as here:

1. Draw a random subset (sample) of training examples, d_1, without replacement from the training dataset, D, to train a weak learner, C_1.
2. Draw a second random training subset, d_2, without replacement from the training dataset and add 50 percent of the examples that were previously misclassified to train a weak learner, C_2.

3. Find the training examples, d_3, in the training dataset, 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.

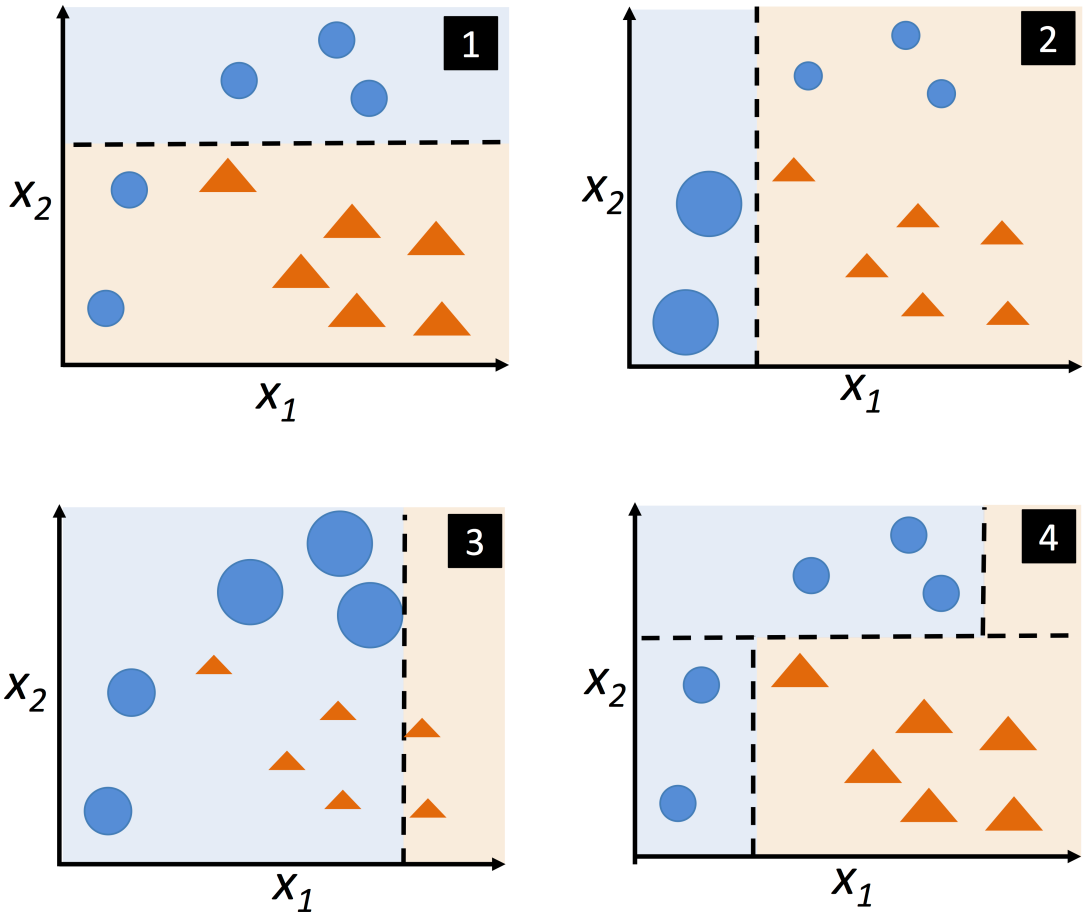
However, boosting algorithms such as **AdaBoost** are also known for their high variance, that is, the tendency to overfit the training data.

Adaboost

AdaBoost uses the complete training dataset to train the weak learners, where the training examples are reweighted in each iteration to build a strong classifier that learns from the mistakes of the previous weak learners in the ensemble.

[35]:
```
Image(filename='figures/07_09.png', width=400)
```

[35]:



In the example above, in Panel 1, all the examples are equally weighted and we train a decistion tree stump (shown as a dashed line), that tries to classify the examples of the two classes (triangles and circles), as well as possibly minimising the loss function (or the impurity score).

In Panel 2, we assign a larger weight to the two previously misclassified examples (circles).

Furthermore, we lower the weight of the correctly classified examples.

The next decision stump will now be more focused on the training examples that have the largest weights — the training examples that are supposedly hard to classify.

The weak learner in Panel 2 misclassifies three different examples from the circle class, which are then assigned a larger weight, as shown in Panel 3.

Assuming that our AdaBoost ensemble only consists of three rounds of boosting, we then combine the three weak learners trained on different reweighted training subsets by a weighted majority vote to get an ensemble classifier output as shown in Panel 4.

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

Let us consider an example of a training dataset consisting of 10 training examples.

[36]:
```
Image(filename='figures/07_10.png', width=500)
```

[36]:

Index	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

The first column of the table depicts the indices of training examples 1 to 10.

In the second column, we can see the feature values of the individual samples, assuming this is a one-dimensional dataset.#

The third column shows the true class label, y_i, for each training sample, x_i, where $y_i \in \{-1, 1\}$.

The initial weights are shown in the fourth column.

We initialise the weights uniformly (assigning the same constant value) and normalize them to sum to 1.

In the case of the 10-sample training dataset, we therefore assign 0.1 to each weight in the weight vector, w.

The predicted class labels, $\hat{y}$ are shown in the fifth column, assuming that our splitting criterion is $x \leq 3.0$.

The last column of the table then shows the updated weights based on the update rules that we defined in the pseudo code.

Compute the error rate

[37]:
```
y = np.array([1, 1, 1, -1, -1, -1,  1,  1,  1, -1])
yhat = np.array([1, 1, 1, -1, -1, -1, -1, -1, -1, -1])
correct = (y == yhat)
weights = np.full(10, 0.1)
print(weights)

epsilon = np.mean(~correct)
print(epsilon)
```

```
[0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1]
0.3
```

Compute the coefficient α_j

[38]:
```
alpha_j = 0.5 * np.log((1-epsilon) / epsilon)
print(alpha_j)
```

```
0.42364893019360184
```

Update the weight vector

[39]:
```
update_if_correct = 0.1 * np.exp(-alpha_j * 1 * 1)
print(update_if_correct)
```

```
0.06546536707079771
```

Increase the ith weight if $\hat{y}_i$ predicted the label incorrectly

[40]:
```
update_if_wrong_1 = 0.1 * np.exp(-alpha_j * 1 * -1)
print(update_if_wrong_1)
```

```
0.1527525231651947
```

[41]:
```
update_if_wrong_2 = 0.1 * np.exp(-alpha_j * -1 * 1)
print(update_if_wrong_2)
```

```
0.1527525231651947
```

Update the weights

[42]:
```
weights = np.where(correct == 1, update_if_correct, update_if_wrong_1)
print(weights)
```

```
[0.06546537 0.06546537 0.06546537 0.06546537 0.06546537 0.06546537
 0.15275252 0.15275252 0.15275252 0.06546537]
```

Normalise

[43]:
```
normalized_weights = weights / np.sum(weights)
print(normalized_weights)
```

```
[0.07142857 0.07142857 0.07142857 0.07142857 0.07142857 0.07142857
 0.16666667 0.16666667 0.16666667 0.07142857]
```

Applying AdaBoost using scikit-learn

We use the same Wine subset that we used to train the bagging meta-classifier.

Via the `base_estimator` attribute, we will train the `AdaBoostClassifier` on 500 decision tree stumps.

[53]:
```
from sklearn.ensemble import AdaBoostClassifier


tree = DecisionTreeClassifier(criterion='entropy',
                              max_depth=1,
                              random_state=1)

ada = AdaBoostClassifier(estimator=tree,
                         n_estimators=500,
                         learning_rate=0.1,
                         random_state=1)
```

[54]:
```
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(f'Decision tree train/test accuracies '
      f'{tree_train:.3f}/{tree_test:.3f}')

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(f'AdaBoost train/test accuracies '
      f'{ada_train:.3f}/{ada_test:.3f}')
```

```
Decision tree train/test accuracies 0.916/0.875
AdaBoost train/test accuracies 1.000/0.917
```

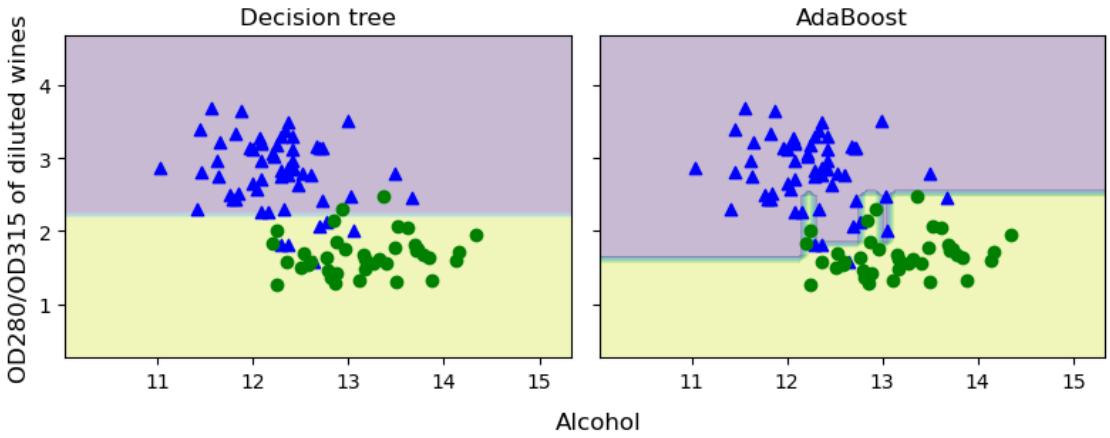
The decision tree stump seems to underfit the training data in contrast to the unpruned decision tree that we saw earlier.

The AdaBoost model predicts all class labels of the training dataset correctly and also shows a slightly improved test dataset performance compared to the decision tree stump.

However, since we introduced additional variance to reduce the model bias, there is a greater gap between training and test performance.

[55]:
```
x_min, x_max = X_train[:, 0].min() - 1, X_train[:, 0].max() + 1
y_min, y_max = X_train[:, 1].min() - 1, 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='green', marker='o')
    axarr[idx].set_title(tt)

axarr[0].set_ylabel('OD280/OD315 of diluted wines', fontsize=12)

plt.tight_layout()
plt.text(0, -0.2,
         s='Alcohol',
```

```
         ha='center',
         va='center',
         fontsize=12,
         transform=axarr[1].transAxes)

# plt.savefig('figures/07_11.png', dpi=300, bbox_inches='tight')
plt.show()
```



It is worth noting that ensemble learning increases the computational complexity compared to individual classifiers. In practice, we need to think carefully about whether we want to pay the price of increased computational costs for an often relatively modest improvement in predictive performance.

Gradient boosting – training an ensemble based on loss gradients

Gradient boosting is an extremely important topic because it forms the basis of popular machine learning algorithms such as **XGBoost**.

Comparing AdaBoost with gradient boosting

Gradient boosting is very similar to AdaBoost. AdaBoost trains decision tree stumps based on errors of the previous decision tree stump.

In particular, the errors are used to compute sample weights in each round **as well as for computing a classifier weight for each decision tree stump** when combining the individual stumps into an ensemble.

Gradient boosting fits decision trees in an iterative fashion using prediction errors. However, gradient boosting trees are usually deeper than decision tree stumps and have typically a maximum depth of 3 to 6 (or a maximum number of 8 to 64 leaf nodes).

Also, in contrast to AdaBoost, gradient boosting does not use the prediction errors for assigning sample weights; they are used directly to form the target variable for fitting the next tree.

Instead of having an individual weighting term for each tree, like in AdaBoost, gradient boosting uses a global learning rate that is the same for each tree.

Outlining the general gradient boosting algorithm

Let us will look at gradient boosting for classification. For simplicity, we will look at a binary classification example.

Gradient boosting builds a series of trees, where each tree is fit on the error—the difference between the label and the predicted value—of the previous tree.

In each round, the tree ensemble improves as we are nudging each tree more in the right direction via small updates. These updates are based on a loss gradient, thus the name gradient boosting.

1. Initialise a model to return a constant prediction value. For this, we use a decision tree root node; that is, a decision tree with a single leaf node. We denote the value returned by the tree as $\hat{y}$, and we find this value by minimizing a differentiable loss function L that we will define later

$$F_0(x) = \arg\min_{\hat{y}} \sum_{i=1}^{n} L(y_i, \hat{y})$$

n is the number of training examples in the dataset.

2. For each tree m = 1, …, M, where M is a user-specified total number of trees, we carry out the following computations outlined in steps 2a to 2d

a. Compute the difference between a predicted value $F(x_i) = \hat{y}_i$ and the class label y_i. This value is sometimes called the pseudo-response or pseudo-residual. we can write this pseudo-residual as the negative gradient of the loss function with respect to the predicted values:

$$r_{im} = -\left[\frac{\partial L(y_i, F(x_i))}{\partial F(x_i)}\right]_{F(x)=F_{m-1}(x)}, i = 1, \ldots, n$$

b. Fit a tree to the pseudo-residuals r_{im}. We use the notation R_{jm} to denote the $j = 1 \ldots J_m$ leaf nodes of the resulting tree in iteration m.

c. For each leaf node Rjm, we compute the following output value:

$$\gamma_{jm} = \arg\min_{\gamma} \sum_{x_i \in R_{jm}} L(y_i, F_{m-1}(x_i) + \gamma)$$

Leaf nodes R_{jm} may contain more than one training example.

d. Update the model by adding the output values γ_m to the previous tree:

$$F_m(x) = F_{m-1}(x) + \eta\gamma_m.$$

Explaining the gradient boosting algorithm for classification

We use the logistic loss function and perform binary classification. For a single training example, we can specify the logistic loss as follows:

$$L_i = -y_i \log p_i + (1 - y_i) \log(1 - p_i)$$

We use log(odds) $\hat{y} = \log(p/(1-p))$ to write $L_i = \log(1 + e^{\hat{y}_i}) - y_i \hat{y}_i$.

The derivative of this loss function with respect to the log(odds) is

$$\frac{\partial L_i}{\partial \hat{y}_i} = \frac{e^{\hat{y}_i}}{1 + e^{\hat{y}_i}} - y_i = p_i - y_i.$$

Thus in step 2c, we get $\gamma_{jm} = \log(1 + e^{\hat{y}_i + \gamma}) - y_i(\hat{y}_i + \gamma) = \frac{\sum_i y_i - p_i}{\sum_i p_i(1-p_i)}$.

Illustrating gradient boosting for classification

Consider the following toy example.

[46]:
```
Image(filename='./figures/07_12.png', width=300)
```

[46]:

	Feature x_1	Feature x_2	Class label y
1	1.12	1.4	1
2	2.45	2.1	0
3	3.54	1.2	1

Let us start with step 1, constructing the root node and computing the log(odds), and step 2a, converting the log(odds) into class-membership probabilities and computing the pseudo-residuals.

The odds can be computed as the number of successes divided by the number of failures. Here, we regard label 1 as success and label 0 as failure, so the odds are computed as: odds = 2/1

[47]:
```
Image(filename='./figures/07_13.png', width=750)
```

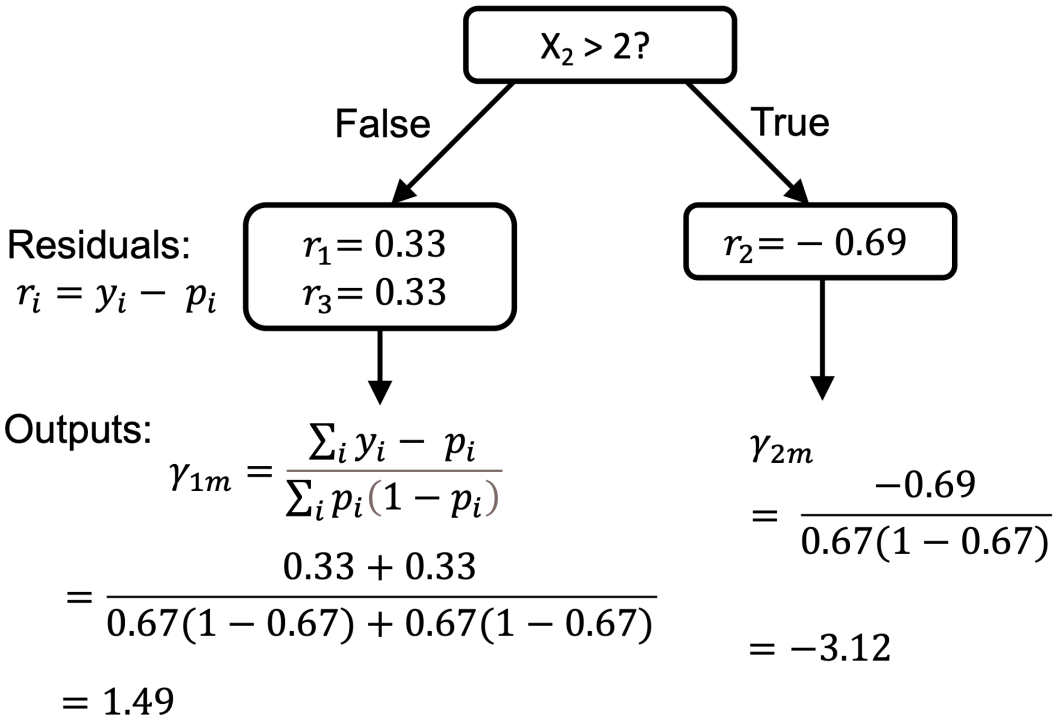
[47]:

	Feature x_1	Feature x_2	Class label y	Step 1: $\hat{y} = \log(\text{odds})$	Step 2A: $p = \frac{1}{1+e^{-\hat{y}}}$	Step 2A: $r = y - p$
1	1.12	1.4	1	0.69	0.67	0.33
2	2.45	2.1	0	0.69	0.67	-0.67
3	3.54	1.2	1	0.69	0.67	0.33

Next, in step 2b, we fit a new tree on the pseudo-residuals r. Then, in step 2c, we compute the output values, γ , for this tree.

[48]:
```
Image(filename='./figures/07_14.png', width=500)
```

[48]:



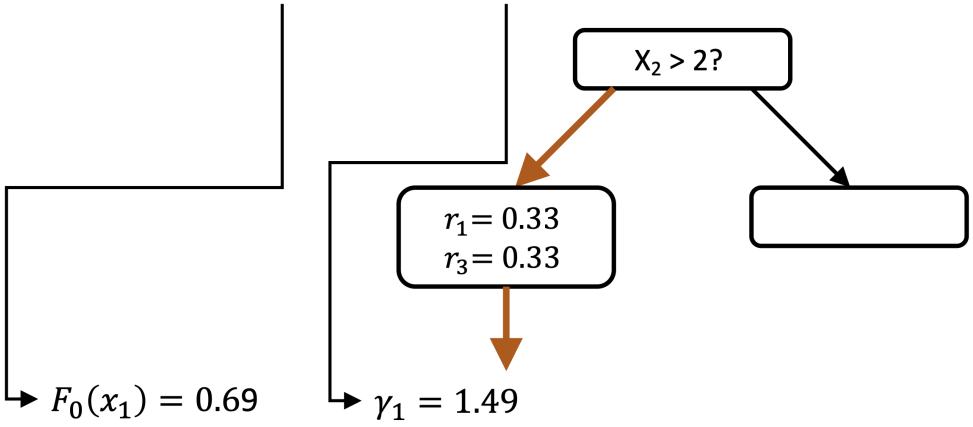
Then, in the final step 2d, we update the previous model and the current model. Assuming a learning rate of $\eta = 0.1$, the resulting prediction for the first training example is shown below.

[49]:
```
Image(filename='./figures/07_15.png', width=500)
```

[49]:

	Feature x_1	Feature x_2	Class label y
1	1.12	1.4	1
2	2.45	2.1	0
3	3.54	1.2	1

Step 2D: $F_m(x) = F_{m-1}(x) + \eta\,\gamma_m$



$$F_1(x_1) = 0.69 + 0.1 \times 1.49 = 0.839$$

Now that we have completed steps 2a to 2d of the first round, m = 1, we can proceed to execute steps 2a to 2d for the second round, m = 2. In the second round, we use the log(odds) returned by the updated model, for example, $F_1(x_1) = 0.839$.

[50]:
```
Image(filename='./figures/07_16.png', width=800)
```

[50]:

	x_1	x_2	y	Step 1: $F_0(x) = \hat{y}$ = log(odds)	Step 2A: $p = \frac{1}{1+e^{-\hat{y}}}$	Step 2A: $r = y - p$	New log(odds) $\hat{y} = F_1(x)$	Step 2A: p	Step 2A: r
1	1.12	1.4	1	0.69	0.67	0.33	0.839	0.698	0.302
2	2.45	2.1	0	0.69	0.67	-0.67	0.378	0.593	-0.593
3	3.54	1.2	1	0.69	0.67	0.33	0.839	0.698	0.302

Round $m = 1$ | Round $m = 2$

We can already see that the predicted probabilities are higher for the positive class and lower for the negative class. Consequently, the residuals are getting smaller, too.

Note that the process of steps 2a to 2d is repeated until we have fit M trees or the residuals are smaller than a user-specified threshold value.

Then, once the gradient boosting algorithm has completed, we can use it to predict the class labels by thresholding the probability values of the final model, $F_M(x)$ at 0.5 (like logistic regression).

In contrast to logistic regression, gradient boosting consists of multiple trees and produces nonlinear decision boundaries.

Using XGBoost

In scikit-learn, gradient boosting is implemented as `sklearn.ensemble.GradientBoostingClassifier`.

[61]:
```
import xgboost as xgb
```

[57]:
```
xgb.__version__
```

[57]:
```
'2.0.3'
```

[58]:
```
model = xgb.XGBClassifier(n_estimators=1000, learning_rate=0.01, max_depth=4,␣
 ↪random_state=1, use_label_encoder=False)


gbm = model.fit(X_train, y_train)

y_train_pred = gbm.predict(X_train)
y_test_pred = gbm.predict(X_test)

gbm_train = accuracy_score(y_train, y_train_pred)
gbm_test = accuracy_score(y_test, y_test_pred)
print(f'XGboost train/test accuracies '
      f'{gbm_train:.3f}/{gbm_test:.3f}')
```

```
XGboost train/test accuracies 0.968/0.917
```

Here, we fit the gradient boosting classifier with 1,000 trees (rounds) and a learning rate of 0.01.

Typically, a learning rate between 0.01 and 0.1 is recommended.

However, remember that the learning rate is used for scaling the predictions from the individual rounds. So, intuitively, the lower the learning rate, the more estimators are required to achieve accurate predictions.