Ensemble Learning

COMP3314
Machine Learning

Outline

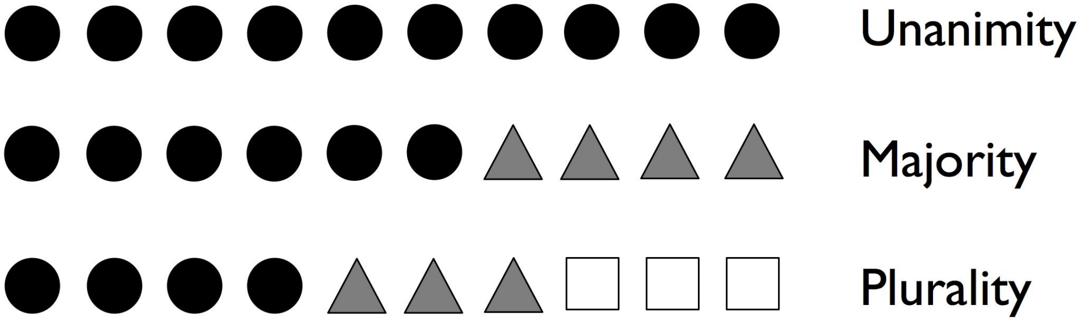
- A set of classifiers can often have a better predictive performance than any of its individual members
- We will learn how to do the following
  - Make predictions based on majority voting
  - Use bagging to reduce overfitting by drawing random combinations of the training set with repetition
  - Apply boosting to build powerful models from weak learners that learn from their mistakes

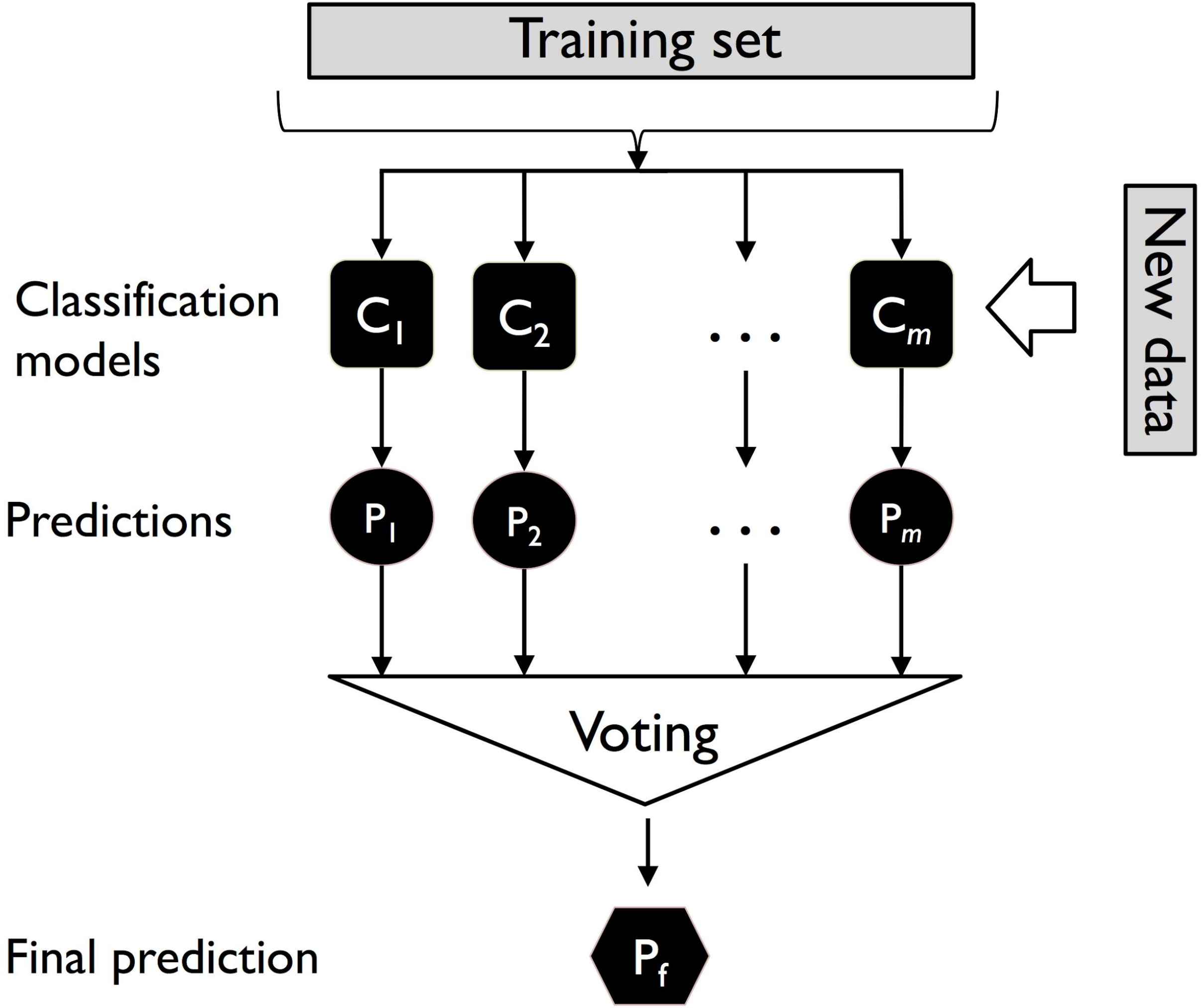
Learning with Ensembles

- Goal
  - Combine different classifiers into a meta-classifier that has better generalization performance than each individual classifier alone
- E.g., assuming that we collected predictions from 10 experts
  - Ensemble methods let us strategically combine these predictions by the 10 experts to come up with a more accurate and robust prediction

Majority/Plurality Voting

- 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
- 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
    - Select the class label that received the most votes (mode)




Training set
Classification models
C_1
C_2
C_m
New data
Predictions
P_1
P_2
P_m
Voting
Final prediction
P_f

Majority/Plurality Voting

- 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, ŷ that received the most votes

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

- E.g., in a binary classification task where class_1 = −1 and class_2 = +1, we can write the majority vote prediction as follows

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

Does an ensemble method always work better than an individual classifier?

Task

- Suppose we have 101 completely independent binary classifiers, each with an error rate of $\varepsilon = 30\%$
- Calculate the majority vote error

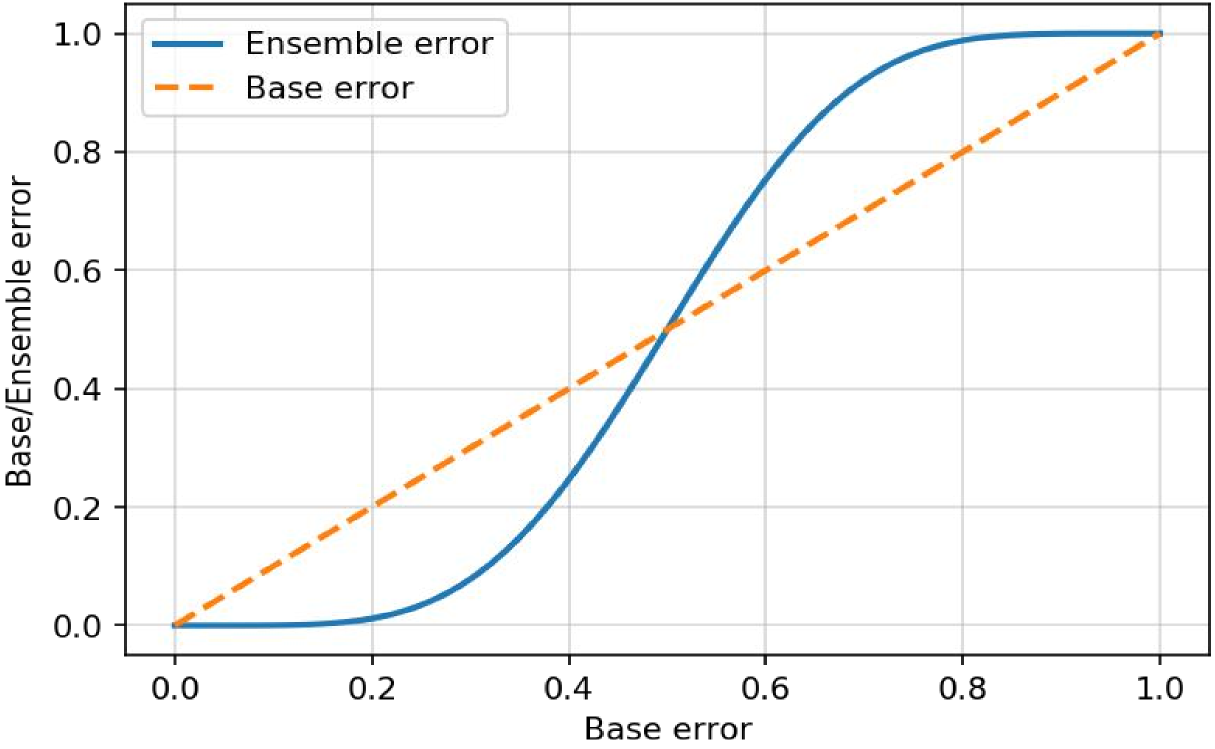
Ensemble Methods vs. Individual Classifier

- Let's apply the simple concepts of combinatorics and assume
  - All n-base classifiers for a binary classification task have an equal error rate, ε
  - Classifiers are independent
  - Error rates are not correlated
- Probability that the prediction of the ensemble is wrong

$$P(y \geq k) = \sum_{k}^{n} \left\langle \begin{matrix} n \\ k \end{matrix} \right\rangle \varepsilon^{k} \left(1 - \varepsilon\right)^{n-k} = \varepsilon_{ensemble}$$

```python
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)
```

```python
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]
```



Weighted Majority Vote Classifier

- Consider the following simple weighted majority vote classifier

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

- 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 an indicator function
  - A is the set of unique class labels
- For equal weights, we can simplify this equation and write it as follows

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

Example

- Consider an ensemble of three base classifiers, C_j ($j \in \{1, 2, 3\}$)
  - Two 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 a weight of 0.2 to C_1 and C_2

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

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

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

```
1
```

Probabilities

- Recall that certain classifiers can return the probability of a predicted class label
  - In scikit-learn: via the predict_proba method
- Using the predicted class probabilities instead of the class labels for majority voting can be useful if the classifiers in our ensemble are well calibrated
- The modified version of the majority vote for predicting class labels from probabilities can be written as follows

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

- Here, p_{ij} is the predicted probability of the jth classifier for class label i

Example

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

- We have a binary classification problem with class labels $i \in \{0, 1\}$ and an ensemble of three classifiers C_j ($j \in \{1,2,3\}$)
- The classifiers C_j return the following class membership probabilities for a particular sample x

$$C_1(\boldsymbol{x}) \rightarrow [0.9, 0.1],\ C_2(\boldsymbol{x}) \rightarrow [0.8, 0.2],\ C_3(\boldsymbol{x}) \rightarrow [0.4, 0.6]$$

- We can then calculate the individual class probabilities as follows

$$p(i_0 \mid \boldsymbol{x}) = 0.2 \times 0.9 + 0.2 \times 0.8 + 0.6 \times 0.4 = 0.58$$

$$p(i_1 \mid \boldsymbol{x}) = 0.2 \times 0.1 + 0.2 \times 0.2 + 0.6 \times 0.6 = 0.42$$

$$\hat{y} = \arg\max_i \left[p(i_0 \mid \boldsymbol{x}), p(i_1 \mid \boldsymbol{x}) \right] = 0$$

```
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
```

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

MajorityVoteClassifier

- A majority vote classifier is available in scikit-learn as sklearn.ensemble.VotingClassifier

- Let's prepare a dataset that we can test the MajorityVoteClassifier on

```
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)
```

```python
import numpy as np
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.pipeline import Pipeline
from sklearn.model_selection import cross_val_score
clf1 = LogisticRegression(penalty='l2', solver='lbfgs', C=0.001, random_state=0)
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))
```

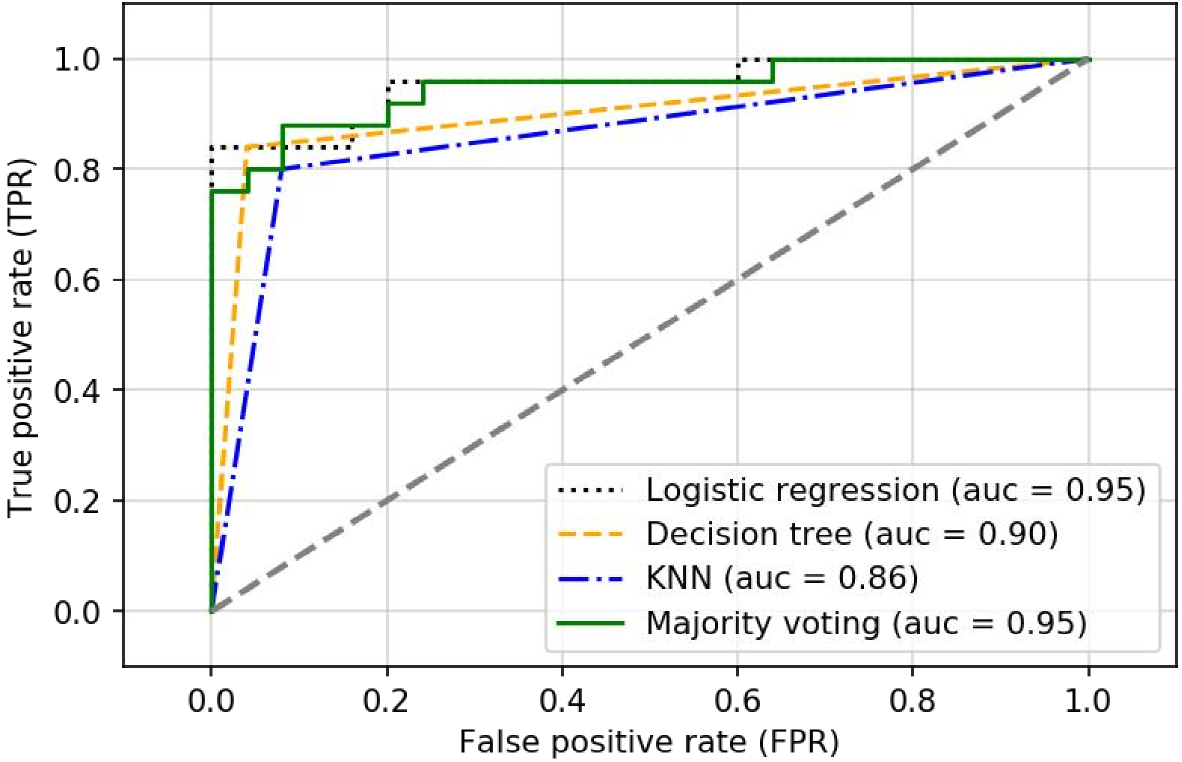
```
10-fold cross validation:

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

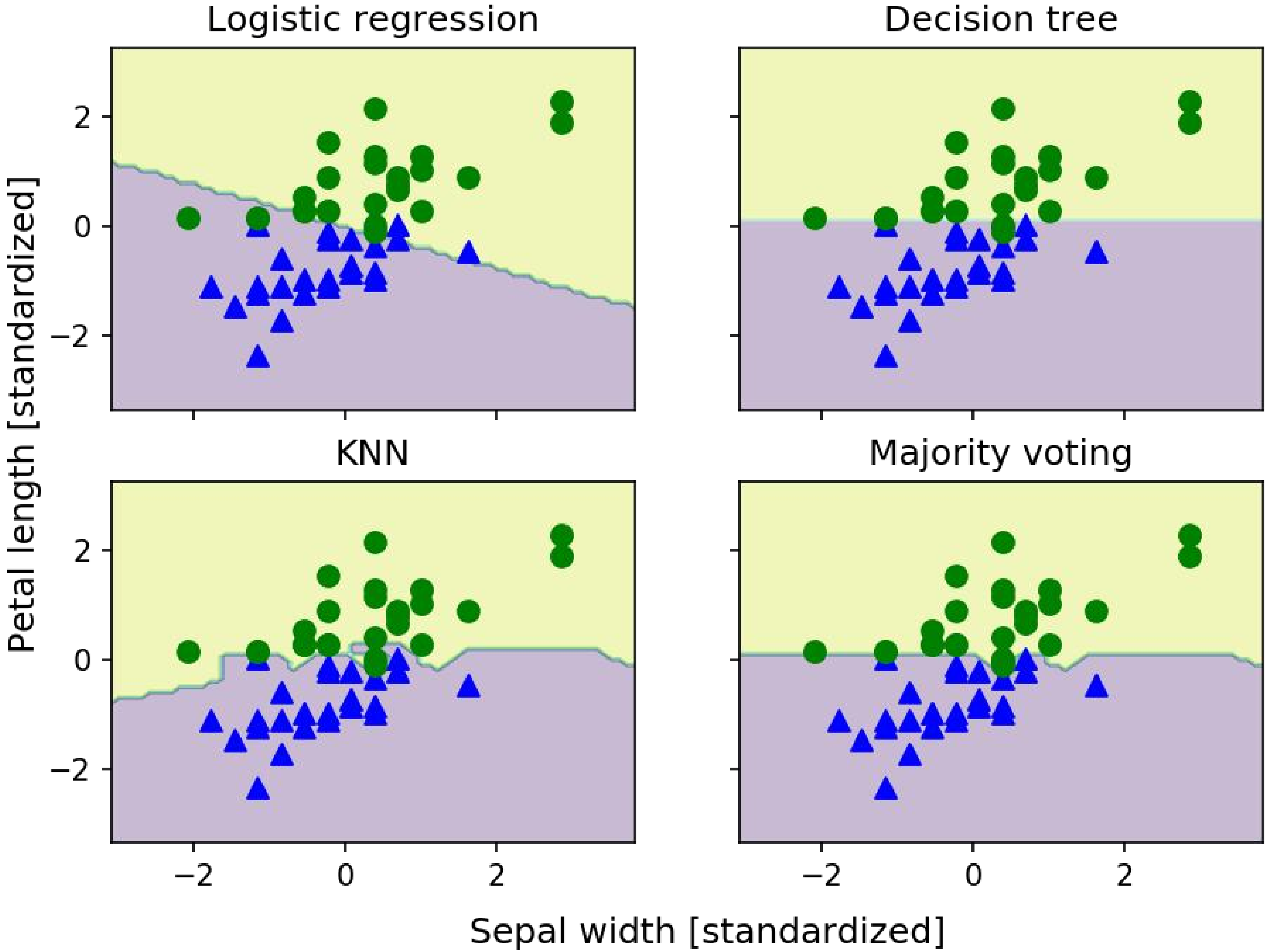
```
mv_clf = MajorityVoteClassifier(classifiers=[pipe1, clf2, pipe3])
clf_labels += ['Majority voting']
all_clf = [pipe1, clf2, pipe3, mv_clf]
for clf, label in zip(all_clf, clf_labels):
    scores = cross_val_score(estimator=clf, X=X_train, y=y_train, cv=10, scoring='roc_auc')
    print("ROC AUC: %0.2f (+/- %0.2f) [%s]" % (scores.mean(), scores.std(), 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]
```

Plotting ROC Curves

- Let’s compute the ROC curves from the test set

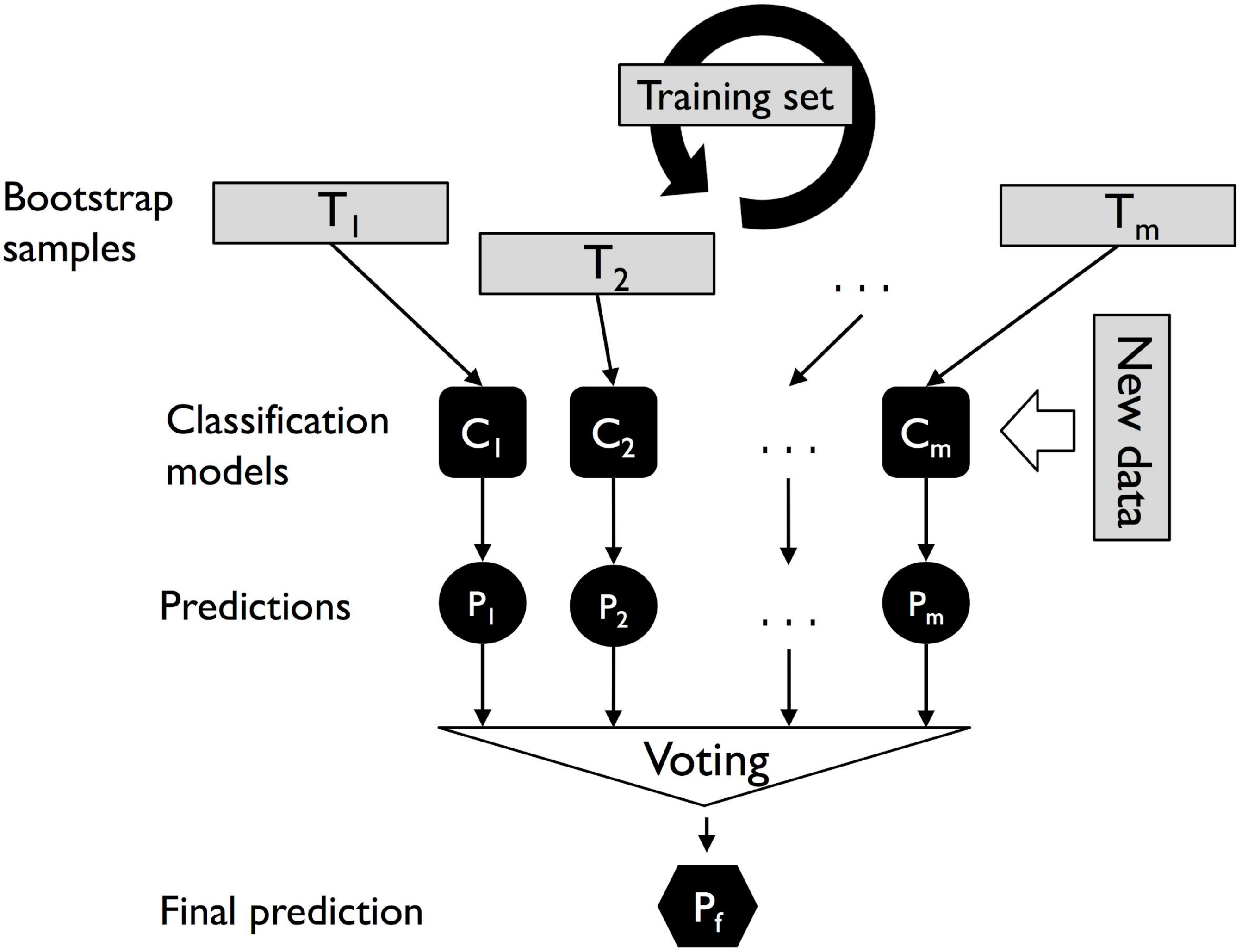




Logistic regression
Decision tree
KNN
Majority voting
Petal length [standardized]
Sepal width [standardized]
2
0
−2
−2
0
2

Outline

- In this chapter, we will construct a set of classifiers that can often have a better predictive performance than any of its individual members
- We will learn how to do the following
  - 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


Training set
Bootstrap samples
T1
T2
. . .
Tm
New data
Classification models
C1
C2
. . .
Cm
Predictions
P1
P2
. . .
Pm
Voting
Final prediction
Pf

Bagging

- Bagging is an ensemble learning technique that is closely related to the MajorityVoteClassifier implemented previously
- Instead of using the same training set to fit the individual classifiers in the ensemble, we draw bootstrap samples (random samples with replacement) from the initial training set
- Bagging is also known as bootstrap aggregation
- Bagging was first proposed by Leo Breiman in 1994
  - He showed that bagging can improve the accuracy of unstable models and decrease the degree of overfitting
  - Recommend reading:
    Bagging predictors, L. Breiman, Technical Report
    - > 20 000 citations

Bagging in a Nutshell

- Seven different training instances
  - 1 to 7
- Sampled randomly with replacement in each round of bagging
- Each bootstrap sample is then used to fit a classifier C_j
- Note that each subset contains a certain portion of duplicates and some of the original samples don't appear in a resampled dataset at all
- Once the individual classifiers are fit to the bootstrap samples, the predictions are combined using majority voting

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

Wine Bagging

- Wine Dataset
  - Let's only consider the wine classes 2 and 3 and only two features

```
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']
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
```

  - Encode class labels into binary format and split the dataset into 80:20; training:testing

```
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)
```

BaggingClassifier

- A bagging classifier algorithm is already implemented in scikit-learn
  - Imported from ensemble submodule
- We will 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

```
from sklearn.ensemble import BaggingClassifier
from sklearn.tree import DecisionTreeClassifier
tree = DecisionTreeClassifier(criterion='entropy', max_depth=None, random_state=1)
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)
```

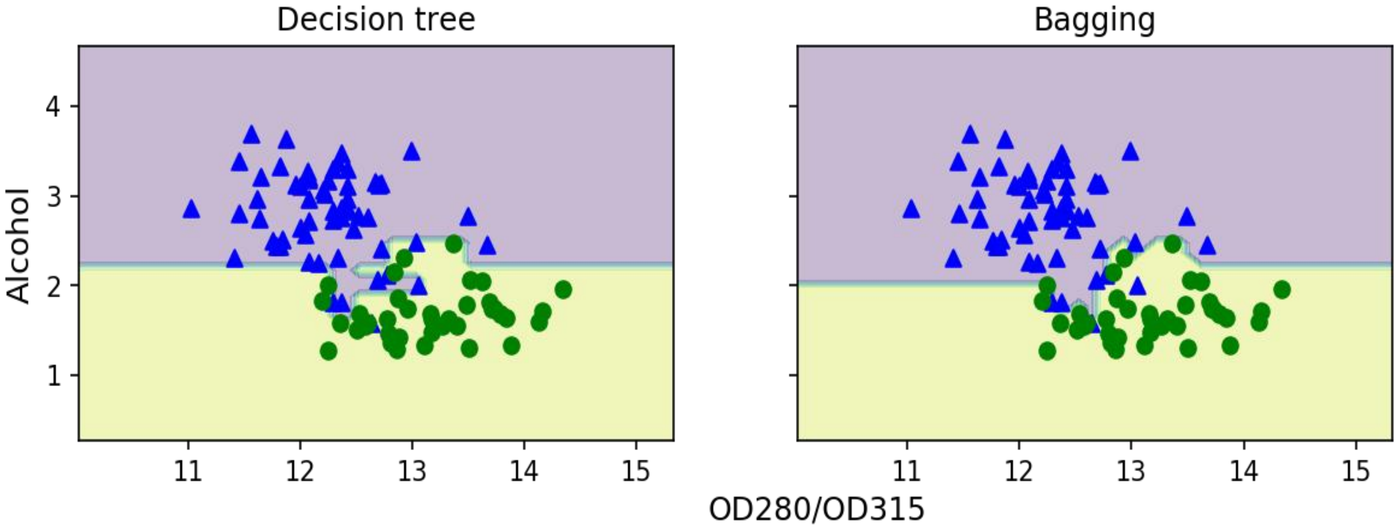
Bagging in Action

- Calculate the accuracy score of the prediction on the training and test dataset to compare the performance of the bagging classifier to the performance of a single unpruned decision tree

```
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))
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))
```

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

Decision Regions



Bagging - Conclusion

- In practice, more complex classification tasks and a dataset's high dimensionality can often lead to overfitting in single decision tree
- This is where the bagging algorithm can really play to its strengths
    - It can be an effective approach to reduce the variance of a model
- However, bagging is ineffective in reducing model bias, that is, models that are too simple to capture the trend in the data well
- This is why we want to perform bagging on an ensemble of classifiers with low bias
    - E.g., unpruned decision trees

Outline

- In this chapter, we will construct a set of classifiers that can often have a better predictive performance than any of its individual members
- We will learn how to do the following
  - 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

AdaBoost (Adaptive Boosting)

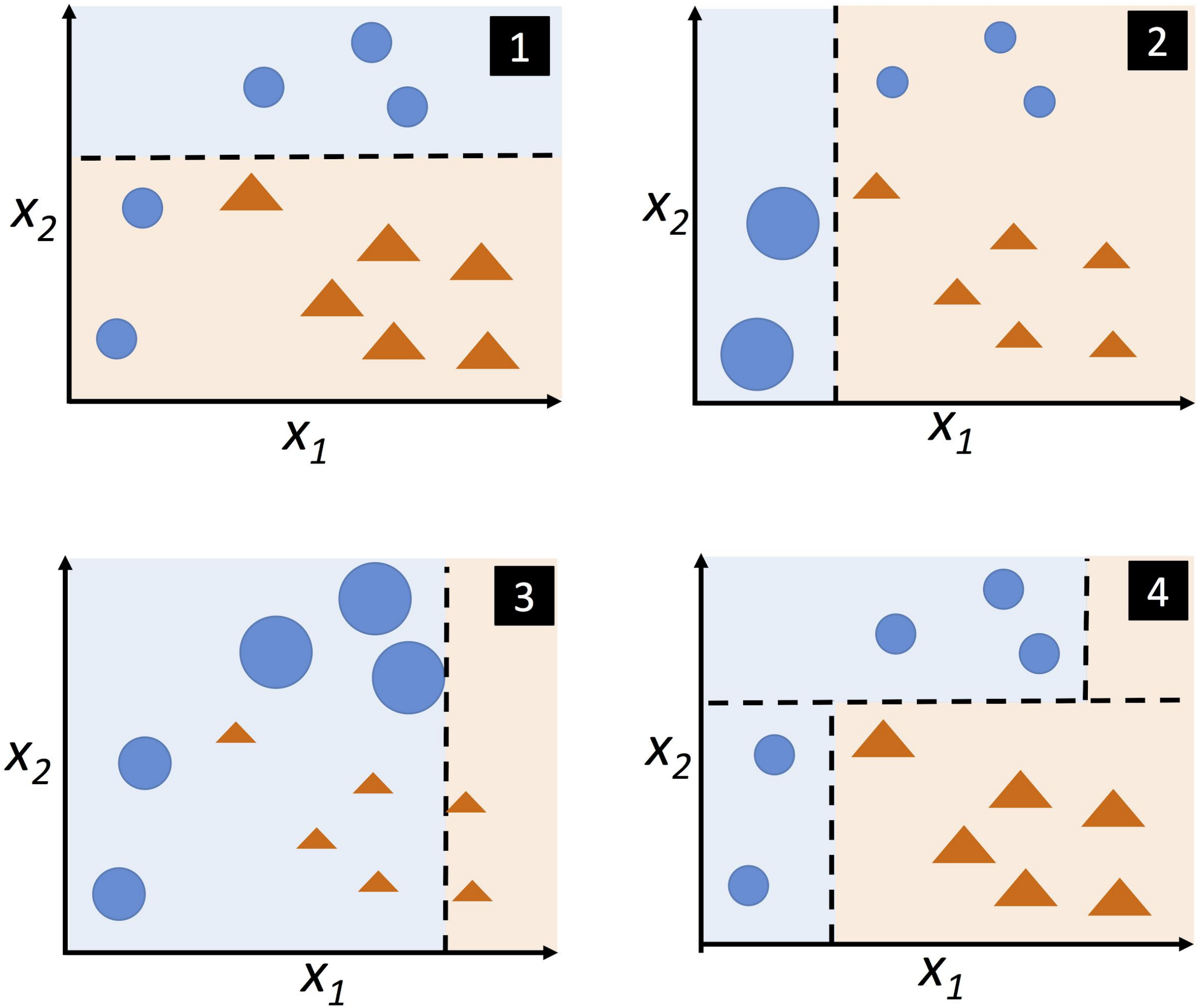
- The original idea behind AdaBoost was formulated by Robert E. Schapire in 1990
    - The Strength of Weak Learnability, R. E. Schapire, Machine Learning, 5(2): 197-227, 1990
    - Experiments with a New Boosting Algorithm, Yoav Freund and Robert E. Schapire
- In 2003, Freund and Schapire received the Goedel Prize for their groundbreaking work
    - A prestigious prize for outstanding publications in the field of computer science

Idea

- In boosting, the ensemble consists of simple base classifiers
  - Often referred to as weak learners
- The weak learners may only have a slight performance advantage over random guessing
- The key concept behind boosting is to focus on training samples that are hard to classify
- The weak learners subsequently learn from misclassified training samples to improve the performance of the ensemble

Original Boosting Procedure

- Draw a random subset of training samples d_1 without replacement from training set D to train a weak learner C_1
- 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
- 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
- Combine the weak learners C_1, C_2, and C_3 via majority voting


1
2
3
4
x_1
x_2

AdaBoost Pseudocode

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

Note: We denote element-wise multiplication by the cross symbol (×) and the dot-product between two vectors by a dot symbol (·)

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

Example: Weight Update

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

- We start by computing the weighted error rate

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

- Next, we compute the coefficient α_j

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

- After we have computed the coefficient α_j, we can now update the weight vector using the equation $\mathbf{w} = \mathbf{w} \times \exp(-\boldsymbol{\alpha}_j \times \hat{\mathbf{y}} \times \mathbf{y})$
- Here $\hat{\mathbf{y}} \times \mathbf{y}$ is an element-wise multiplication between the vectors of the predicted and true class labels, respectively
- Thus, if a prediction $\hat{y}_i$ is correct, $\hat{y}_i \times y_i$ will have a positive sign so that we decrease the ith weight, since α_j is a positive number as well

Example: Weight Update

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

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

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:

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

After we have updated each weight in the weight vector, we normalize the weights so that they sum up to one (step 2f):

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

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

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 $0.153 / 0.914 \approx 0.167$.

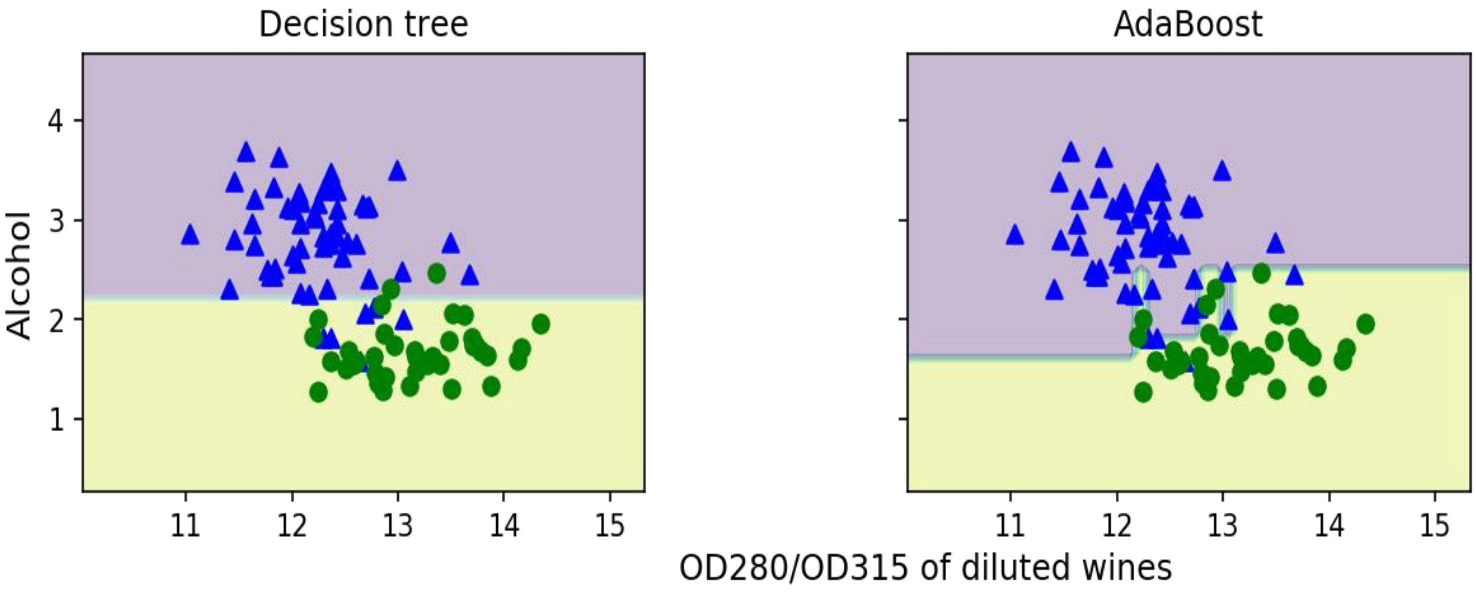
AdaBoost in scikit-learn

```
from sklearn.ensemble import AdaBoostClassifier
tree = DecisionTreeClassifier(criterion='entropy', max_depth=1, random_state=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))
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))
```

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

Decision Region Plotting



AdaBoost: Conclusion

- 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

References

- Most materials in this chapter are based on
  - Book
  - Code