Machine Learning

Lecture 5: SVMs, Decision Trees, and Random Forests

Instructor: Dr. Farhad Pourkamali Anaraki

Support Vector Machines

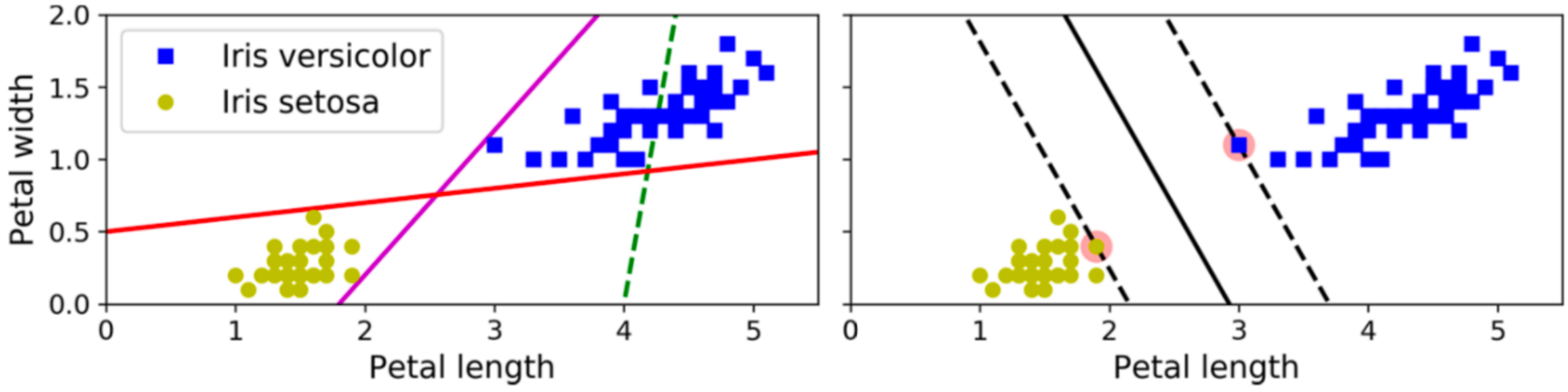
Introduction

- A Support Vector Machine (SVM) is a powerful model to perform linear or nonlinear classification and regression

- SVMs are suitable for analyzing complex small- to medium-sized datasets

- We start this section by discussing linear classification using SVMs

  - Idea: large margin classification

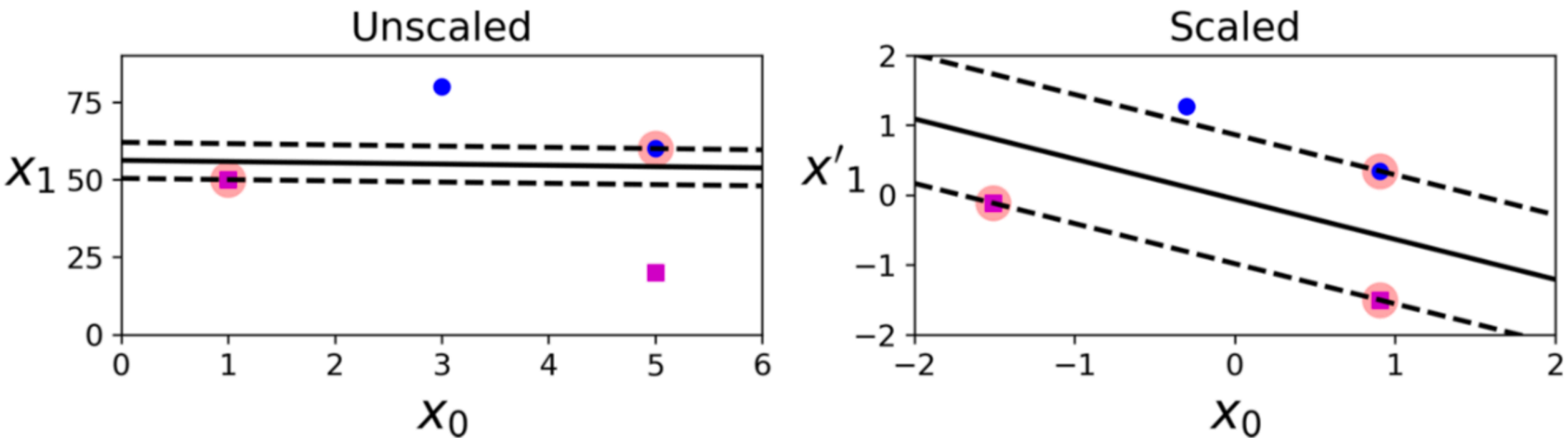


- Stay far away from the closest instances as possible

- We should find “support vectors” (circled in the right figure)

Challenges of using SVMs

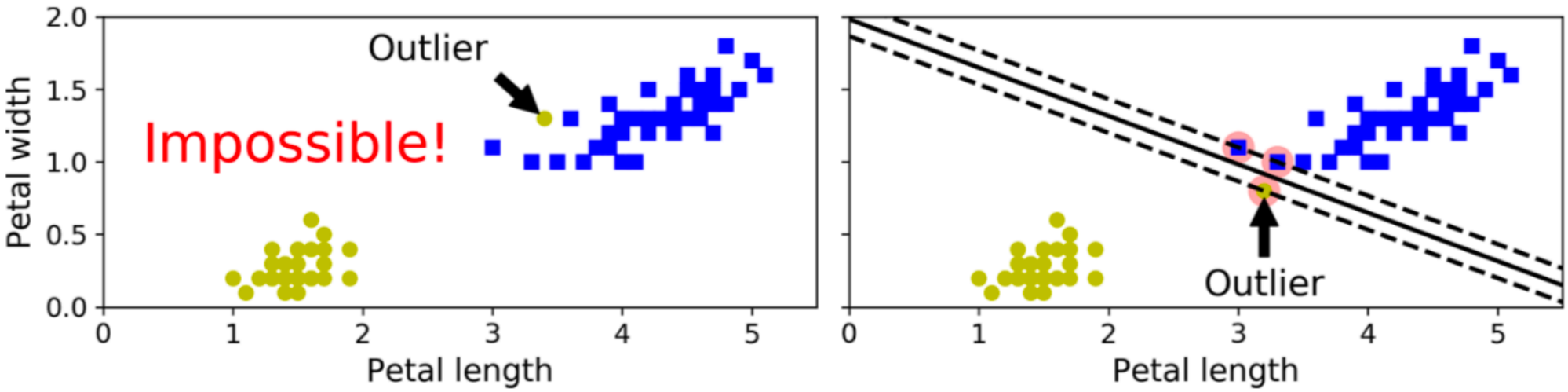
- SVMs are very sensitive to the feature scales



- A significant challenge is to find appropriate features to improve model performance
  - Usually we need to bring in domain expertise

Challenges of using SVMs

- Hard margin classification: requires that all training instances are correctly classified
  - It only works if the dataset is linearly separable
  - Sensitive to outliers

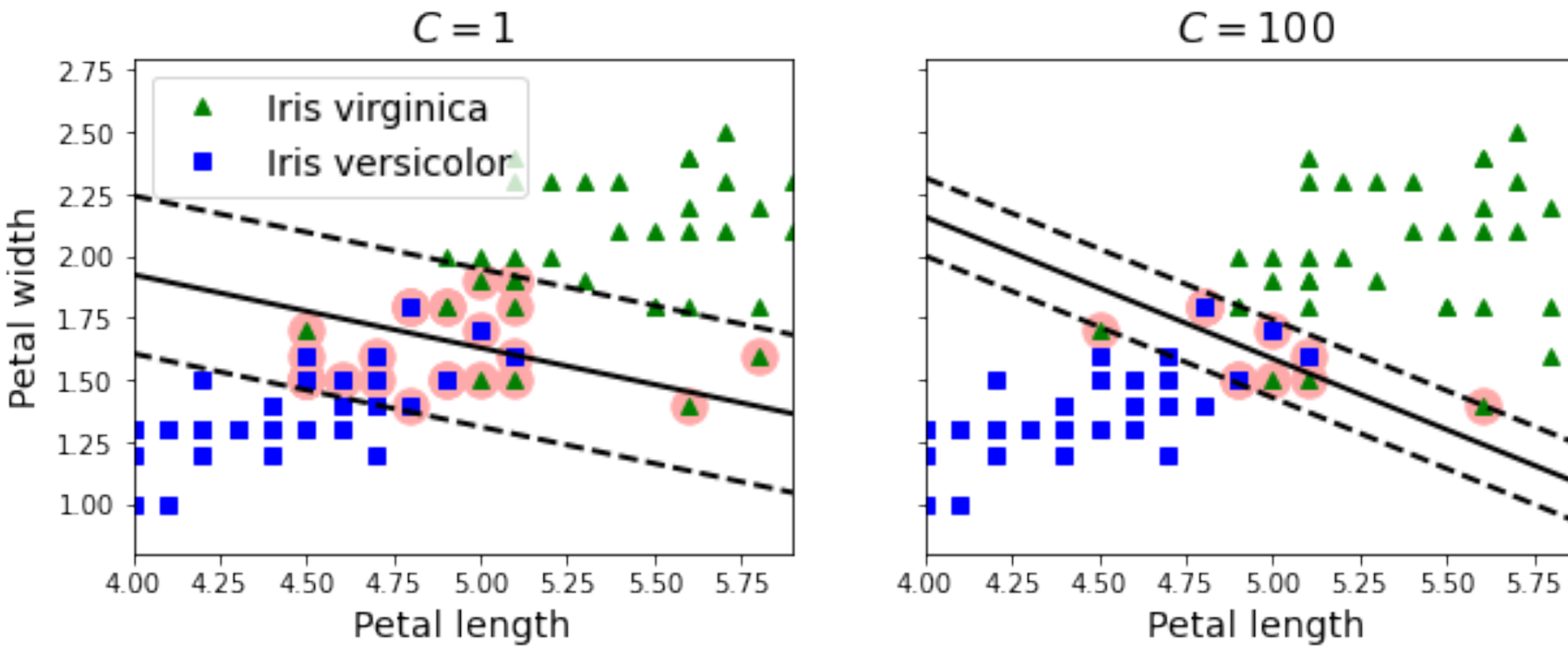


- How to solve this problem? Soft margin classification
  - We use a more flexible model to find a good balance between a large margin and limiting margin violations

SVM in Scikit-Learn

```
import numpy as np
from sklearn import datasets
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.svm import LinearSVC

iris = datasets.load_iris()
X = iris["data"][:, (2, 3)]  # petal length, petal width
y = (iris["target"] == 2).astype(np.float64)  # Iris virginica

scaler = StandardScaler()
svm_clf1 = LinearSVC(C=1, loss="hinge", random_state=42)
svm_clf2 = LinearSVC(C=100, loss="hinge", random_state=42)

scaled_svm_clf1 = Pipeline([
        ("scaler", scaler),
        ("linear_svc", svm_clf1),
    ])
scaled_svm_clf2 = Pipeline([
        ("scaler", scaler),
        ("linear_svc", svm_clf2),
    ])

scaled_svm_clf1.fit(X, y)
scaled_svm_clf2.fit(X, y)
```

SVM in Scikit-Learn



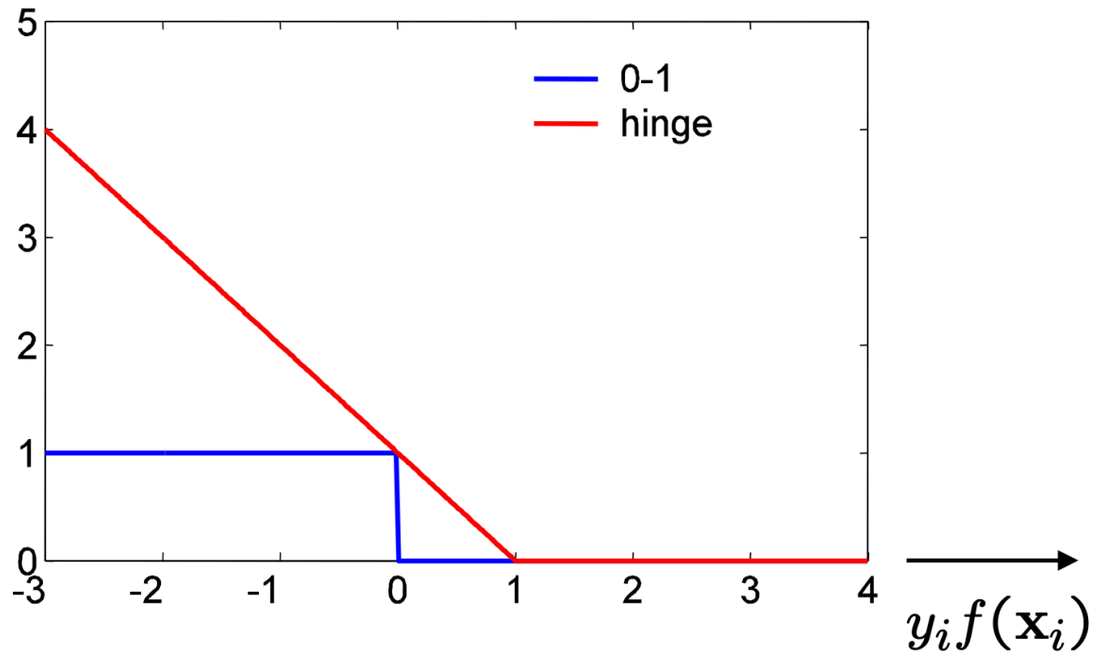
- The strength of the regularization is inversely proportional to C
- If your SVM model is overfitting, you can try regularizing it by decreasing C

Loss function

- Given training data $(\mathbf{x}^{(i)}, y^{(i)})$ with $y_i \in \{-1, +1\}$, we want to learn a classifier

$$f(\mathbf{x}^{(i)}) \begin{cases} \geq 0 & \text{if } y^{(i)} = +1 \\ < 0 & \text{if } y^{(i)} = -1 \end{cases}$$

  - Therefore, $y^{(i)} f(\mathbf{x}^{(i)}) > 0$ for a correct prediction



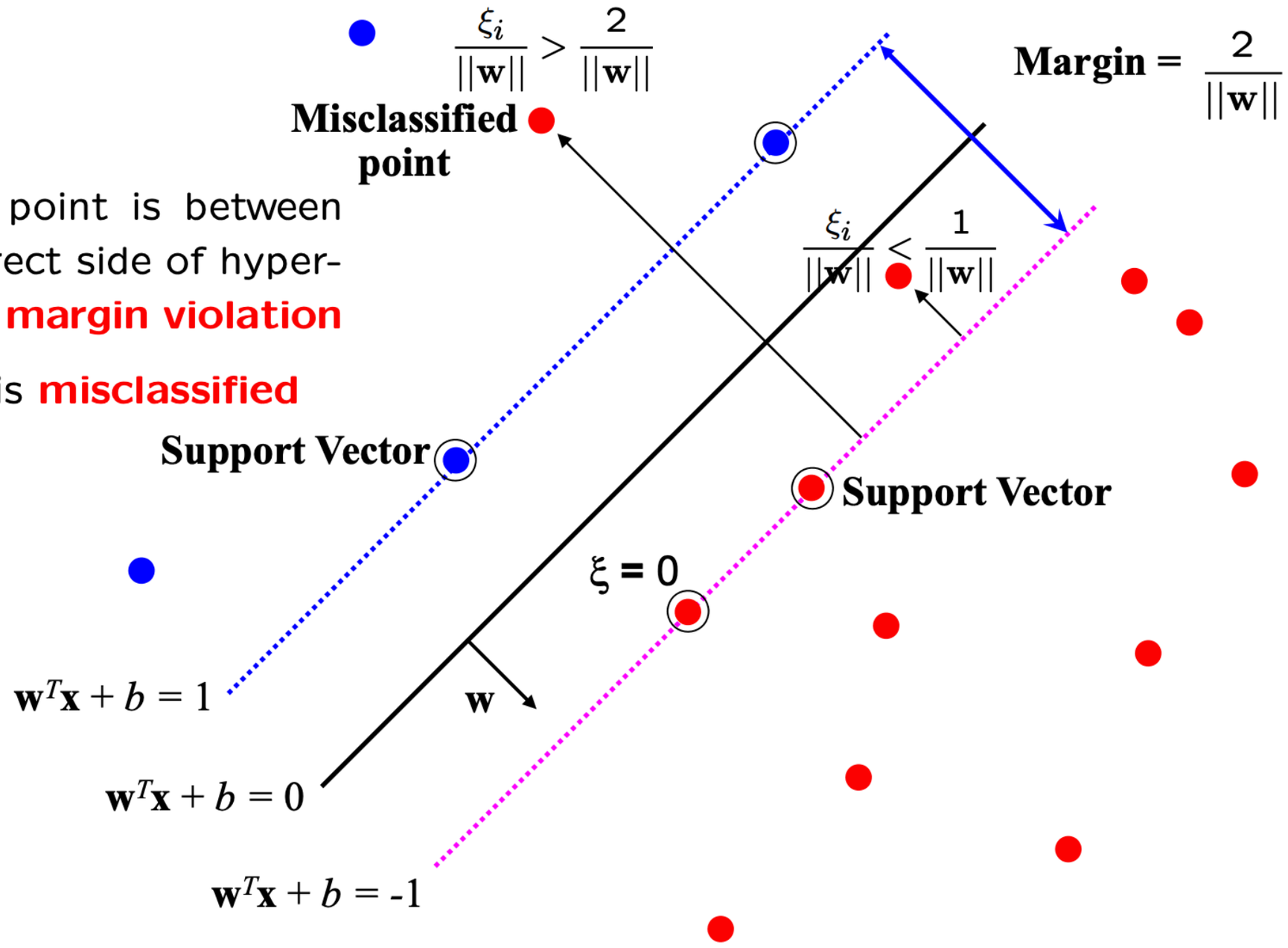
- SVM uses hinge loss

$$\max\left(0, 1 - y^{(i)} f(\mathbf{x}^{(i)})\right)$$

Optimization problem (illustration)

$$\xi_i \geq 0$$

- for $0 < \xi \leq 1$ point is between margin and correct side of hyper-plane. This is a **margin violation**
- for $\xi > 1$ point is **misclassified**



Optimization problem

- Constrained optimization problem

$$\min_{\mathbf{w}\in\mathbb{R}^d,\xi_i\in\mathbb{R}^+} \|\mathbf{w}\|^2 + C\sum_{i=1}^{n}\xi_i$$

$$y^{(i)}\left(\mathbf{w}^T\mathbf{x}^{(i)} + b\right) \geq 1 - \xi_i, \text{ for } i = 1,\ldots,n$$

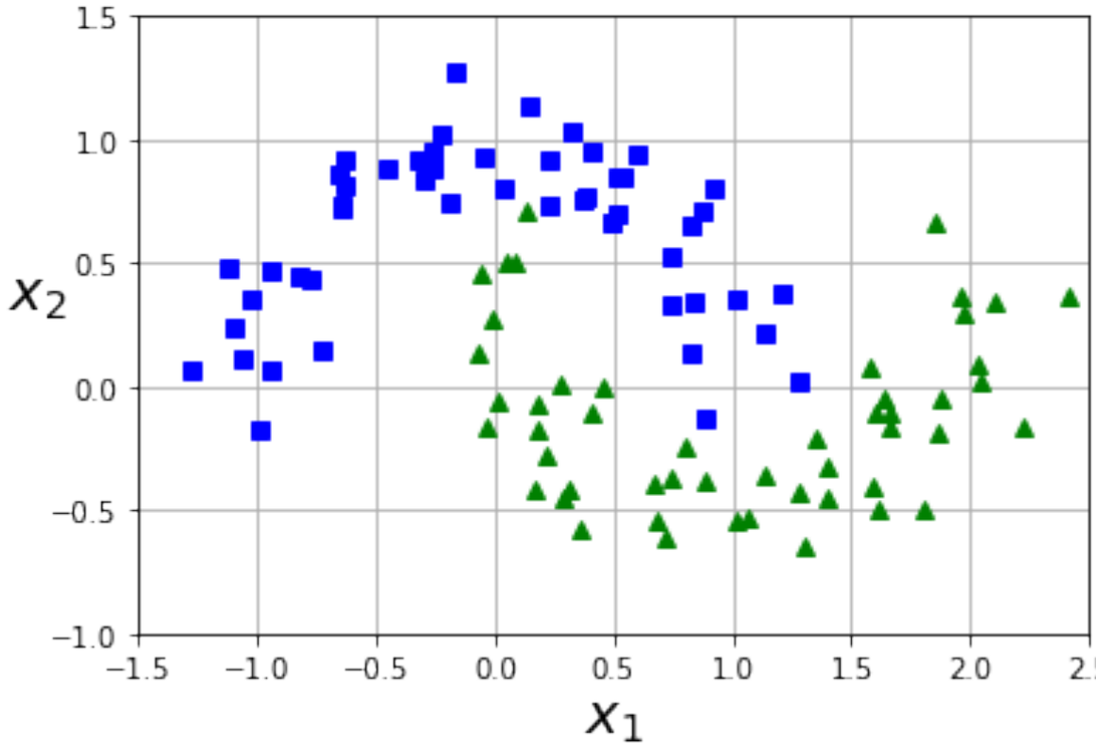
  - Small C allows constraints to be ignored, thus large margin
  - Large C makes constraints hard to ignore, narrow margin

- Unconstrained version using hinge loss

$$\min_{\mathbf{w}\in\mathbb{R}^d} \|\mathbf{w}\|^2 + C\sum_{i=1}^{n}\max\left(0,1 - y^{(i)}f(\mathbf{x}^{(i)})\right)$$

Polynomial kernel SVM

- Adding polynomial features is easy to implement for non-linear classification



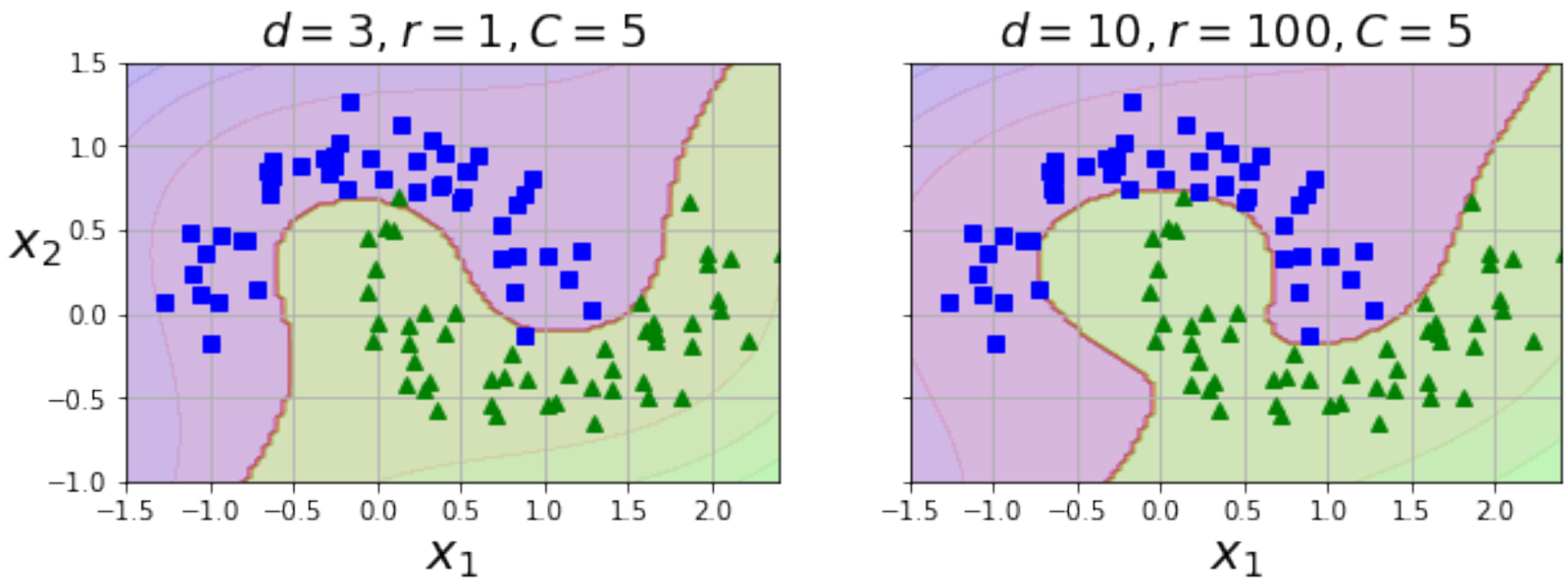
- Instead of using a linear kernel function $k(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}) = \langle \mathbf{x}^{(i)}, \mathbf{x}^{(j)} \rangle$, we use a polynomial kernel function

$$k(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}) = \left(\gamma \langle \mathbf{x}^{(i)}, \mathbf{x}^{(j)} \rangle + r \right)^d$$

  - All available kernels: https://scikit-learn.org/stable/modules/svm.html#svm-kernels

Implementation

```
from sklearn.pipeline import Pipeline
from sklearn.svm import SVC
from sklearn.preprocessing import StandardScaler

poly_kernel_svm_clf = Pipeline([
        ("scaler", StandardScaler()),
        ("svm_clf", SVC(kernel="poly", degree=3, coef0=1, C=5))
    ])
poly_kernel_svm_clf.fit(X, y)

poly100_kernel_svm_clf = Pipeline([
        ("scaler", StandardScaler()),
        ("svm_clf", SVC(kernel="poly", degree=10, coef0=100, C=5))
    ])
poly100_kernel_svm_clf.fit(X, y)
```



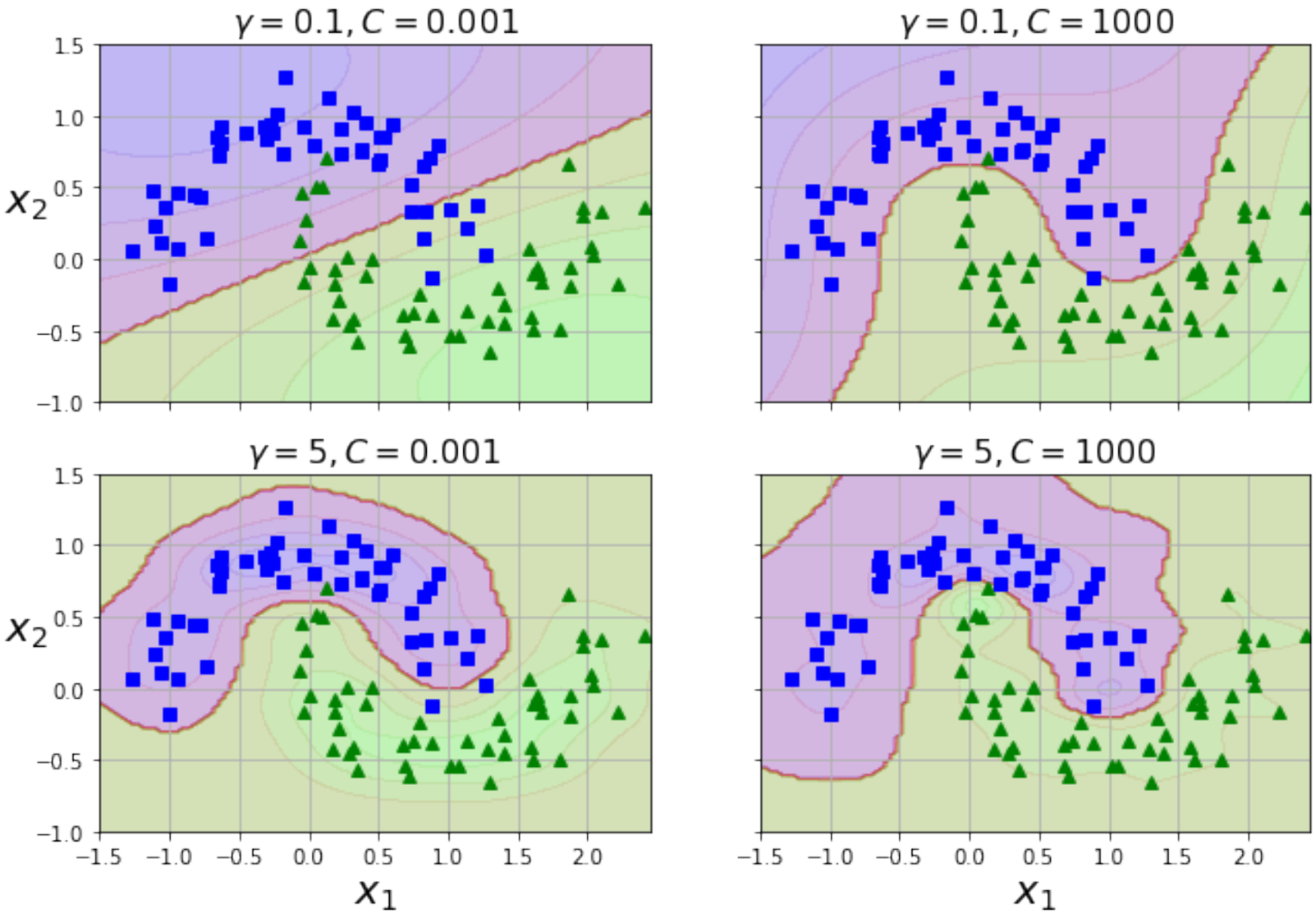
Gaussian kernel

- We can replace the polynomial kernel function with the following kernel

$$k(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}) = \exp\left(-\gamma \|\mathbf{x}^{(i)} - \mathbf{x}^{(j)}\|^2 \right)$$

```
gamma1, gamma2 = 0.1, 5
C1, C2 = 0.001, 1000
hyperparams = (gamma1, C1), (gamma1, C2), (gamma2, C1), (gamma2, C2)

svm_clfs = []
for gamma, C in hyperparams:
    rbf_kernel_svm_clf = Pipeline([
            ("scaler", StandardScaler()),
            ("svm_clf", SVC(kernel="rbf", gamma=gamma, C=C))
        ])
    rbf_kernel_svm_clf.fit(X, y)
    svm_clfs.append(rbf_kernel_svm_clf)
```

Results



Observations regarding C

- The parameter γ acts like a regularization hyperparameter
  - Increasing γ makes the decision boundary more irregular
  - When overfitting, you can reduce γ

Reading Assignment: Chapter 5 of Textbook "Support Vector Machines"

Pages 153-162

Decision Trees

Introduction

- Like SVMs, decision trees can perform both classification and regression tasks
- We start our discussion by training and making predictions with decision trees
- Let's start with the popular Iris dataset

```
from sklearn.datasets import load_iris

iris = load_iris()
X = iris.data[:, 2:] # petal length and width
y = iris.target
```

```
import numpy as np
print(X.shape, y.shape)
print(np.unique(y))
```

```
(150, 2) (150,)
[0 1 2]
```

Decision trees in Scikit-Learn

- Training

```
from sklearn.tree import DecisionTreeClassifier
tree_clf = DecisionTreeClassifier(max_depth=2, random_state=42)
tree_clf.fit(X, y)
```

```
DecisionTreeClassifier(ccp_alpha=0.0, class_weight=None, criterion='gini',
                       max_depth=2, max_features=None, max_leaf_nodes=None,
                       min_impurity_decrease=0.0, min_impurity_split=None,
                       min_samples_leaf=1, min_samples_split=2,
                       min_weight_fraction_leaf=0.0, presort='deprecated',
                       random_state=42, splitter='best')
```

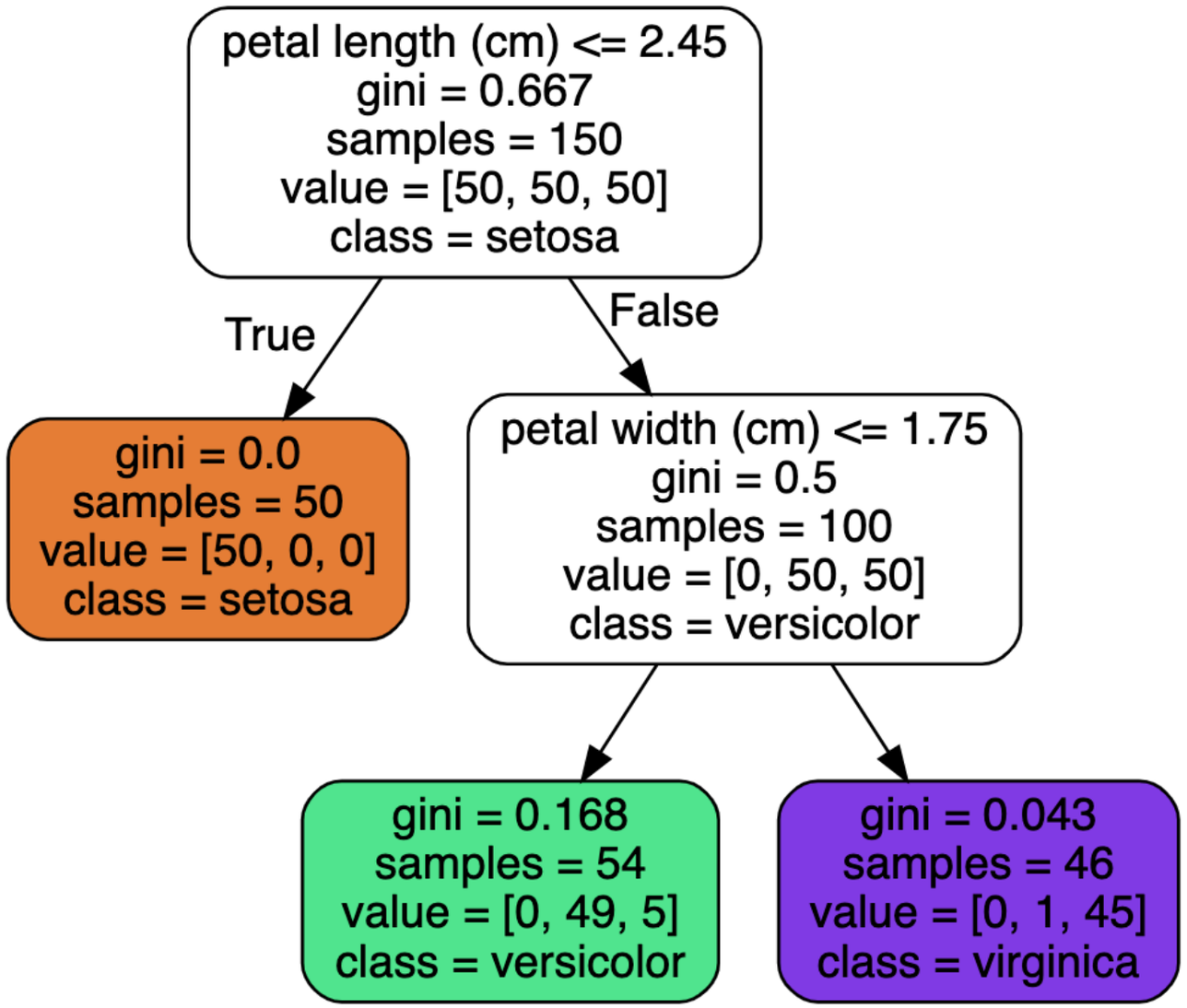
- Visualization

```
import os
# Where to save the figures
PROJECT_ROOT_DIR = "."
CHAPTER_ID = "decision_trees"
IMAGES_PATH = os.path.join(PROJECT_ROOT_DIR, "images", CHAPTER_ID)
os.makedirs(IMAGES_PATH, exist_ok=True)
from graphviz import Source
from sklearn.tree import export_graphviz
export_graphviz(
        tree_clf,
        out_file=os.path.join(IMAGES_PATH, "iris_tree.dot"),
        feature_names=iris.feature_names[2:],
        class_names=iris.target_names,
        rounded=True,
        filled=True
    )
Source.from_file(os.path.join(IMAGES_PATH, "iris_tree.dot"))
```

← Export a decision tree in DOT format.

Trained decision tree

- This is the image we get by running the previous script



depth 0

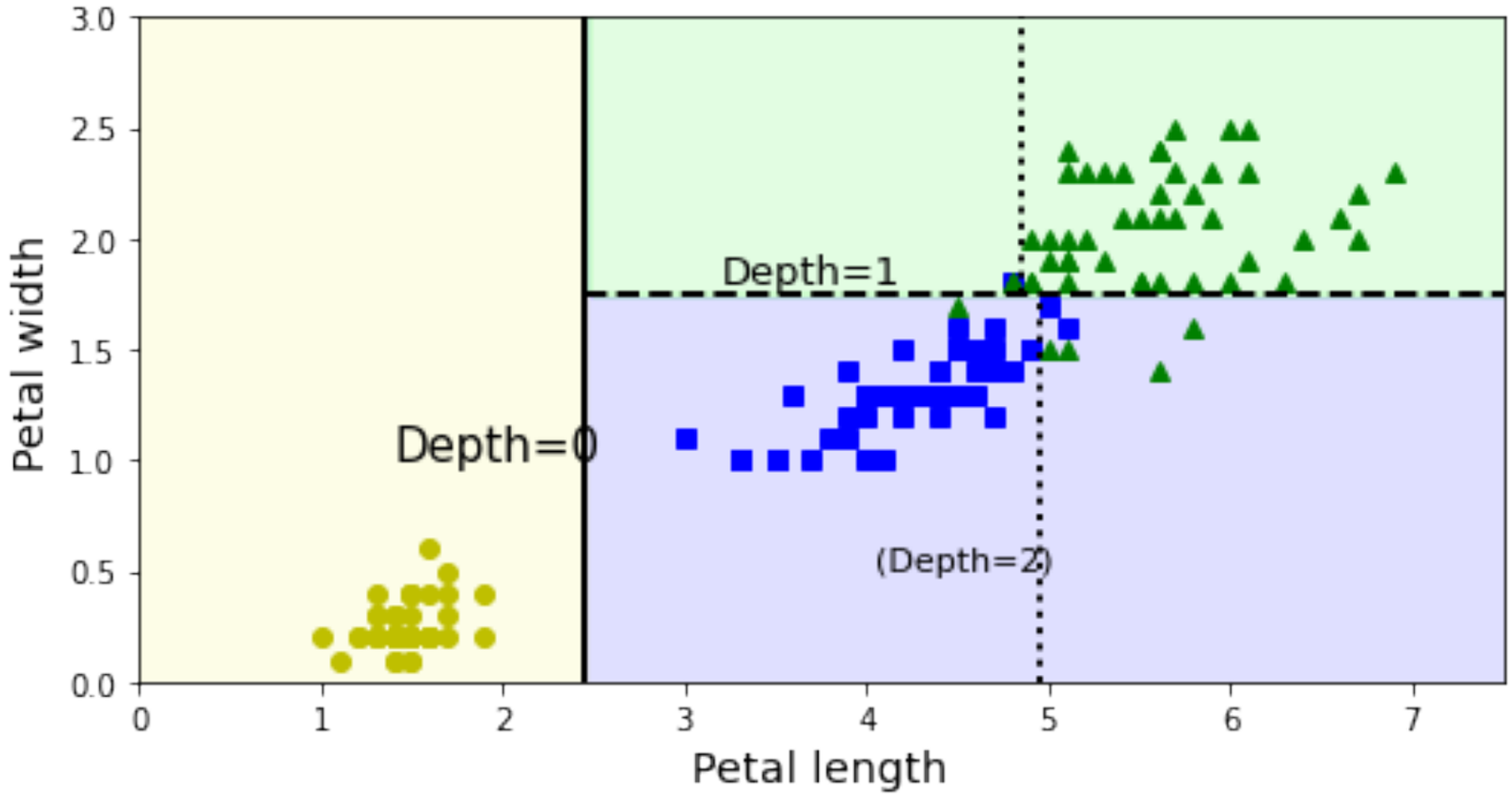
depth 1

depth 2

$$\text{gini} = 1 - (0/54)^2 - (49/54)^2 - (5/54)^2 \approx 0.168$$

- Therefore, decision trees don't require feature scaling or centering

Plotting the decision boundary



- Predictions made by Decision Trees are easy to interpret because they provide simple classification rules

- We can also estimate class probabilities

```
tree_clf.predict_proba([[5, 1.5]])
```

```
array([[0.        , 0.90740741, 0.09259259]])
```

```
tree_clf.predict([[5, 1.5]])
```

```
array([1])
```

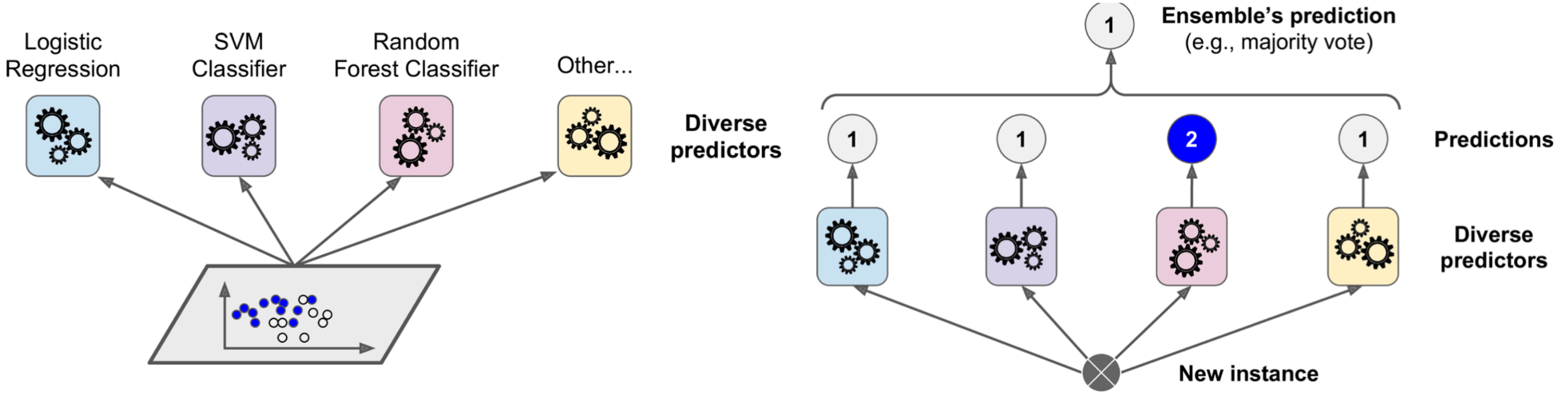
Reading Assignment: Chapter 6 of Textbook "Decision Trees"

Pages 175-180

Ensemble Learning and Random Forests

Introduction

- Ensemble learning
  - Simple idea: if you aggregate the predictions of a group of predictors (i.e., classifiers or regressors), you will often get better predictions than with the best individual predictor

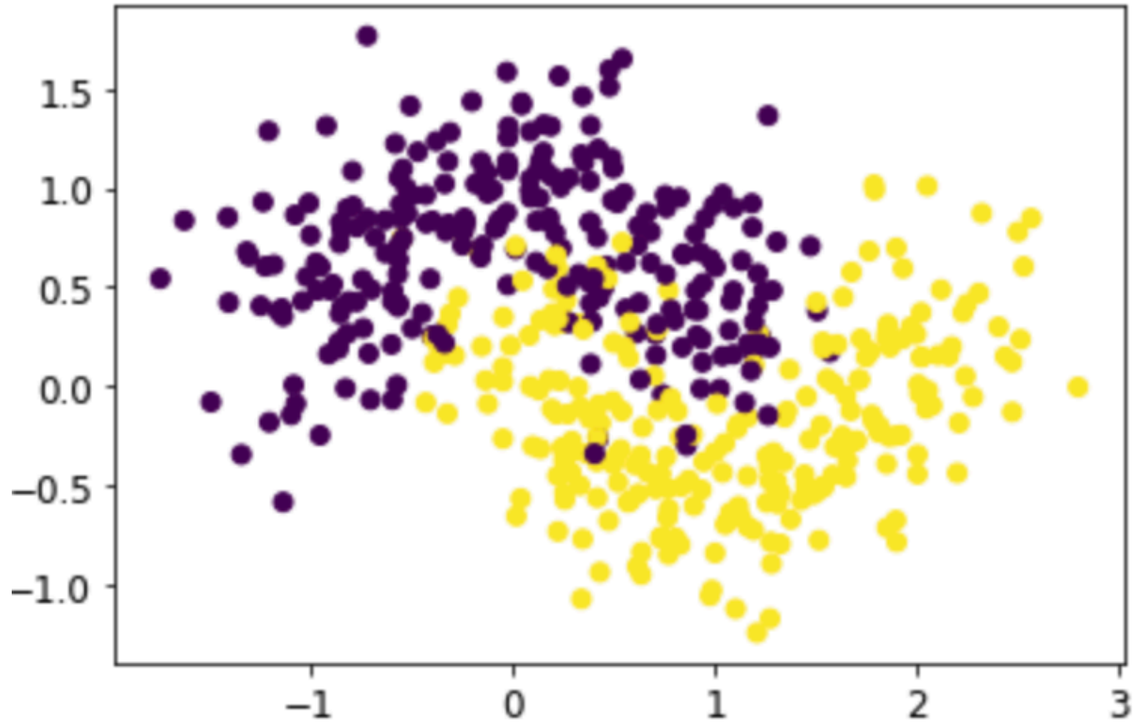


- Train a group of Decision Tree classifiers, each on a different random subset of training data (ensemble of Decision Trees is called Random Forest)

Voting classifier in Scikit-Learn

- Create a synthetic dataset

```
from sklearn.model_selection import train_test_split
from sklearn.datasets import make_moons

X, y = make_moons(n_samples=500, noise=0.30, random_state=42)
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=42)
```



Voting classifier in Scikit-Learn

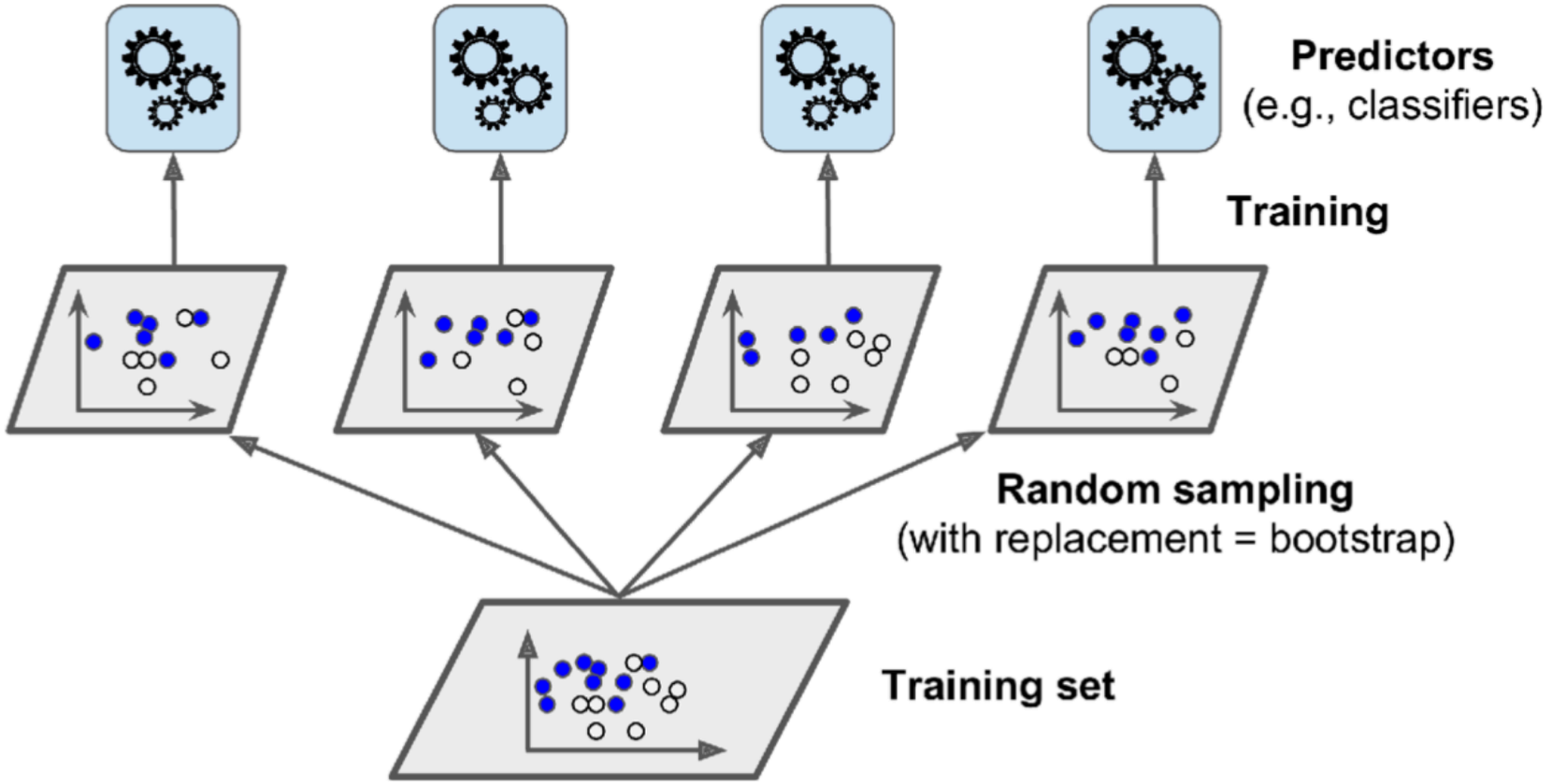
```
from sklearn.ensemble import RandomForestClassifier
from sklearn.ensemble import VotingClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC

log_clf = LogisticRegression(solver="lbfgs", random_state=42)
rnd_clf = RandomForestClassifier(n_estimators=100, random_state=42)
svm_clf = SVC(gamma="scale", random_state=42)

voting_clf = VotingClassifier(
    estimators=[('lr', log_clf), ('rf', rnd_clf), ('svc', svm_clf)],
    voting='hard')
```

```
from sklearn.metrics import accuracy_score

for clf in (log_clf, rnd_clf, svm_clf, voting_clf):
    clf.fit(X_train, y_train)
    y_pred = clf.predict(X_test)
    print(clf.__class__.__name__, accuracy_score(y_test, y_pred))
```

```
LogisticRegression 0.864
RandomForestClassifier 0.896
SVC 0.896
VotingClassifier 0.912
```

Bagging and pasting

- In the previous example, we used a diverse set of classifiers
- Another approach is to use the same learning algorithm and train them on different random subsets of the training set
  - Sampling with replacement: bagging
  - Sampling without replacement: pasting



Implementation in Scikit-Learn

- Ensemble of 500 Decision Tree classifiers, each trained on 100 training instances

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

bag_clf = BaggingClassifier(
    DecisionTreeClassifier(random_state=42), n_estimators=500,
    max_samples=100, bootstrap=True, random_state=42)
bag_clf.fit(X_train, y_train)
y_pred = bag_clf.predict(X_test)

from sklearn.metrics import accuracy_score
print(accuracy_score(y_test, y_pred))
```

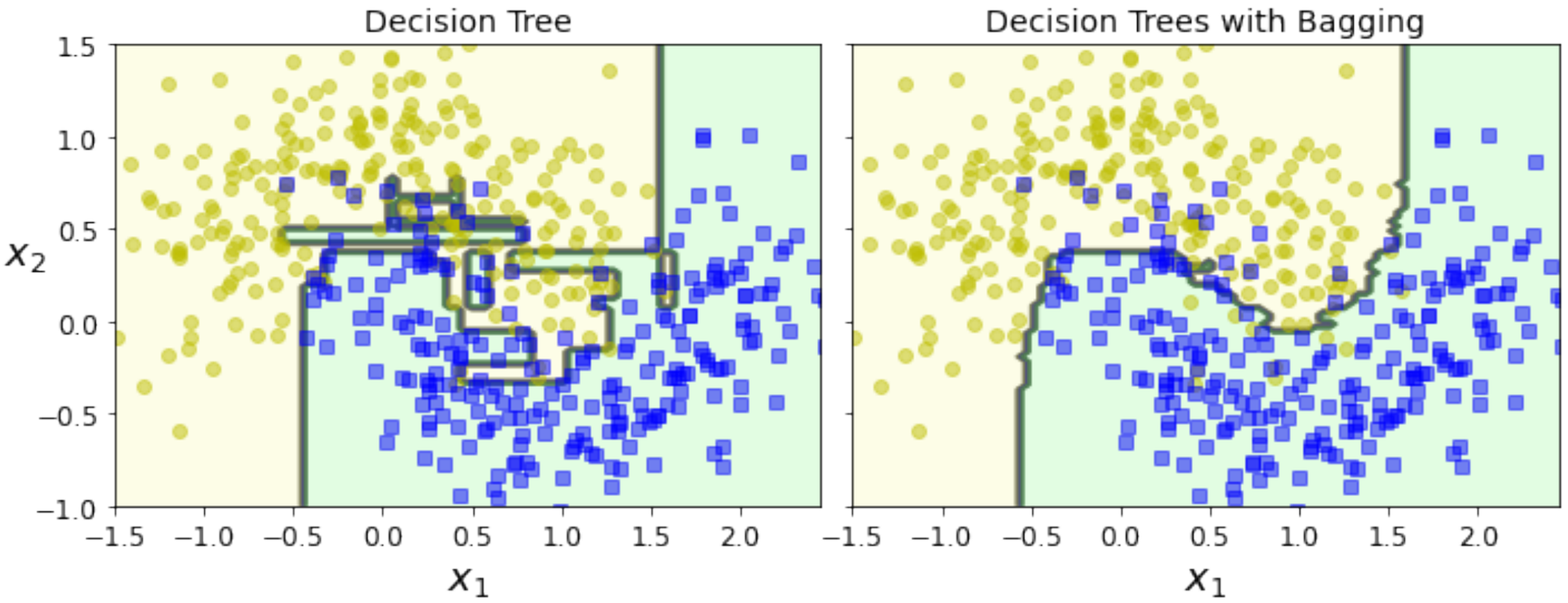
```
0.904
```

- Compare with a single Decision Tree classifier

```
tree_clf = DecisionTreeClassifier(random_state=42)
tree_clf.fit(X_train, y_train)
y_pred_tree = tree_clf.predict(X_test)
print(accuracy_score(y_test, y_pred_tree))
```

```
0.856
```

Decision boundaries



Random Forests

- We can use the following built-in function:

```
from sklearn.ensemble import RandomForestClassifier

rnd_clf = RandomForestClassifier(n_estimators=500, max_leaf_nodes=16,
                                 random_state=42)
rnd_clf.fit(X_train, y_train)

y_pred_rf = rnd_clf.predict(X_test)
```

```
np.sum(y_pred == y_pred_rf) / len(y_pred)  # almost identical predictions
```

```
0.976
```

Parameters:

n_estimators : *int, default=100*

The number of trees in the forest.

Changed in version 0.22: The default value of `n_estimators` changed from 10 to 100 in 0.22.

criterion : *{"gini", "entropy"}, default="gini"*

The function to measure the quality of a split. Supported criteria are "gini" for the Gini impurity and "entropy" for the information gain. Note: this parameter is tree-specific.

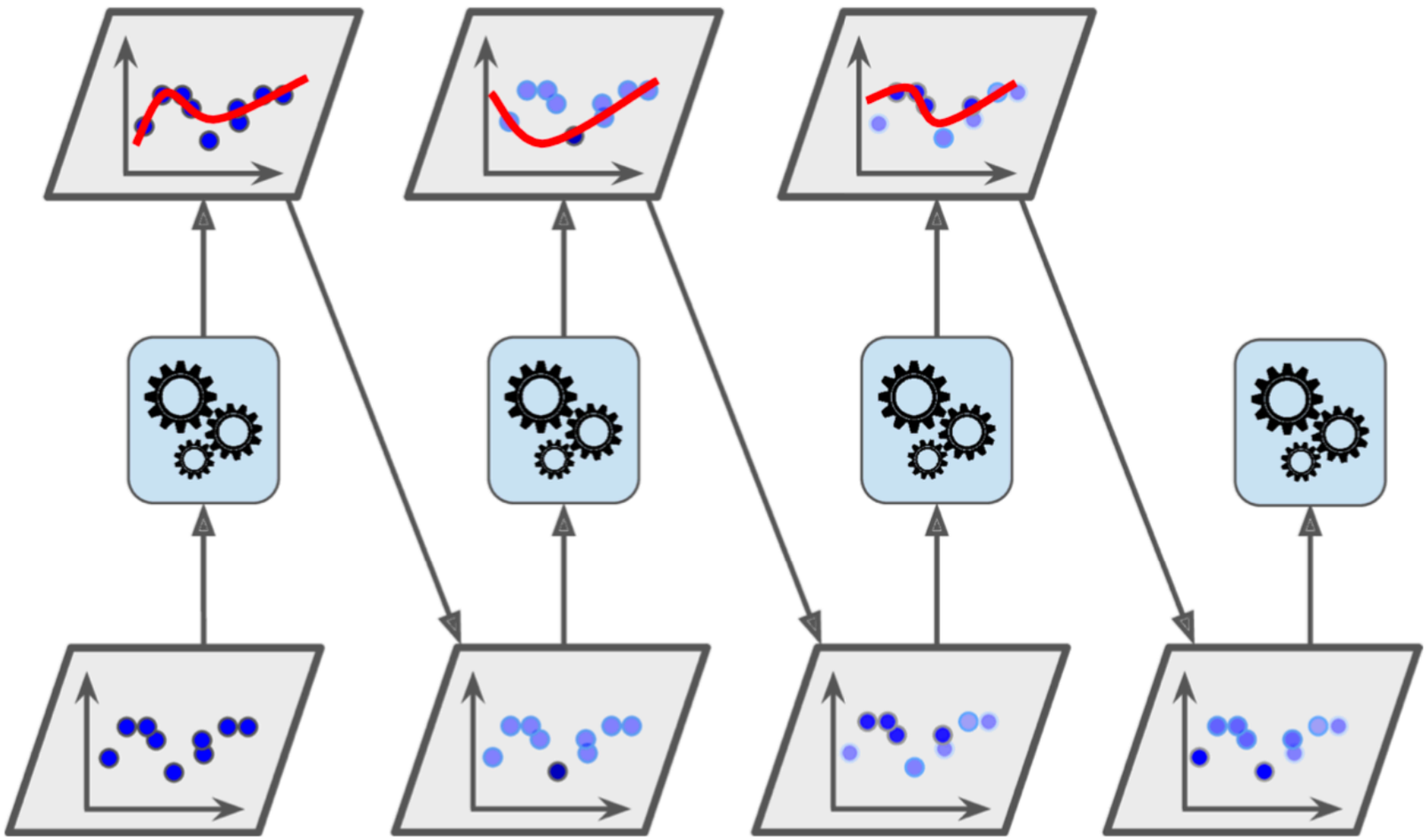
max_depth : *int, default=None*

The maximum depth of the tree. If None, then nodes are expanded until all leaves are pure or until all leaves contain less than min_samples_split samples.

Boosting

- Another ensemble method that trains predictors sequentially, each tying to correct its predecessor

  - AdaBoost: train a base classifier and increase the relative weight of misclassified training instances



Reading Assignment: Chapter 7 of Textbook "Ensemble Learning and Random Forests"

Pages 189-200