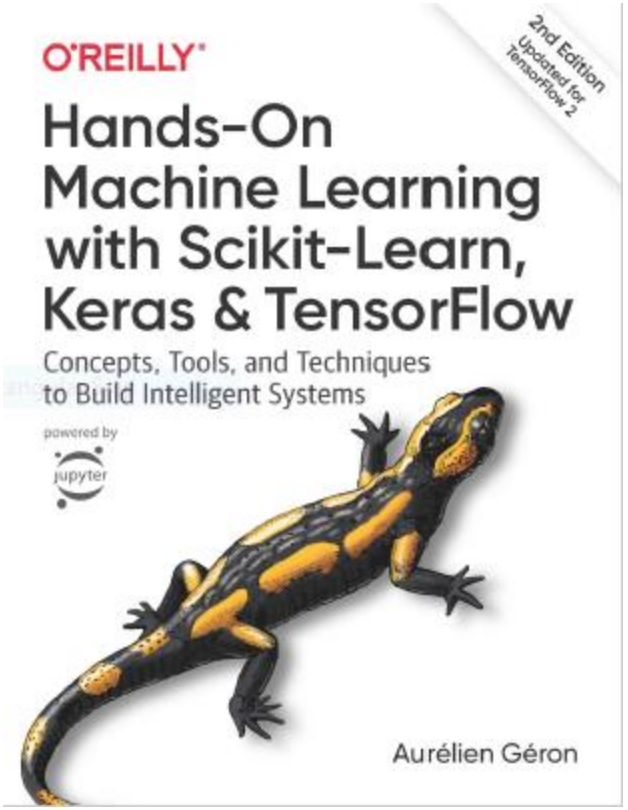
Classical Techniques

Prof. Gheith Abandah

Reference

- Chapter 5: **Support Vector Machines**
- Chapter 6: **Decision Trees**
- Chapter 7: **Ensemble Learning and Random Forests**

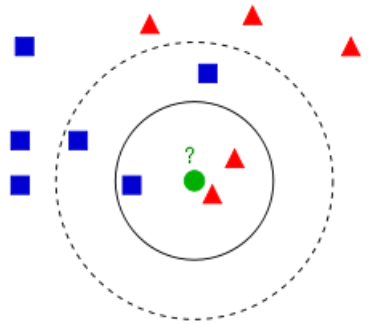


- Aurélien Géron, **Hands-On Machine Learning with Scikit-Learn, Keras and TensorFlow**, O'Reilly, 2nd Edition, 2019
  - Material: https://github.com/ageron/handson-ml2

Outline

1. k-Nearest Neighbors
2. Support Vector Machines
3. Decision Trees
4. Ensemble Learning and Random Forests
5. Exercises

k-Nearest Neighbors



- Find a predefined number of training samples (*k*) closest in distance to the new point and predict the label from them: **regression** or **classification**.
- The number of samples can be a user-defined constant (**k-nearest neighbor learning**), or vary based on the local density of points (**radius-based neighbor learning**).
- The distance can be any metric measure: standard **Euclidean distance** is the most common choice.
- Reference: https://scikit-learn.org/stable/modules/neighbors.html

Nearest Neighbors Classification

```
class sklearn.neighbors.KNeighborsClassifier(n_neighbors=5,
                                              weights='uniform', … )
```

- **weights** can be: **uniform**: All points in each neighborhood are weighted equally, and **distance**: Weight points by the inverse of their distance.
- Example:

```
from sklearn.neighbors import KNeighborsClassifier
knn_clf = KNeighborsClassifier()
knn_clf.fit(X_train, y_train)
```

Nearest Neighbors Regression

```
class sklearn.neighbors.KNeighborsRegressor(n_neighbors=5,
                                            weights='uniform', … )
```

- The label assigned to a query point is computed based on the mean of the labels of its nearest neighbors.
- Example:

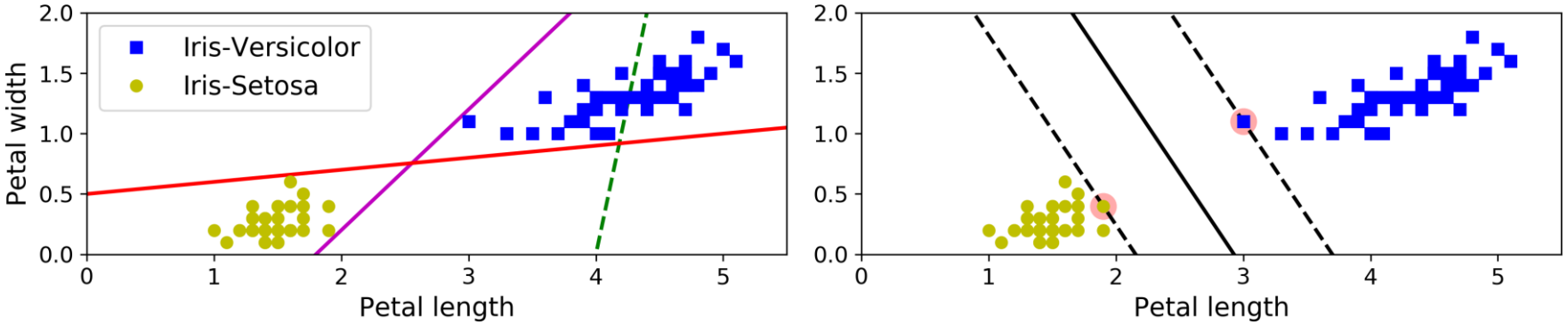
```
from sklearn.neighbors import KNeighborsRegressor
model = KNeighborsRegressor(n_neighbors=3)
model.fit(X, y)
```

Outline

1. k-Nearest Neighbors
2. Support Vector Machines
3. Decision Trees
4. Ensemble Learning and Random Forests
5. Exercises

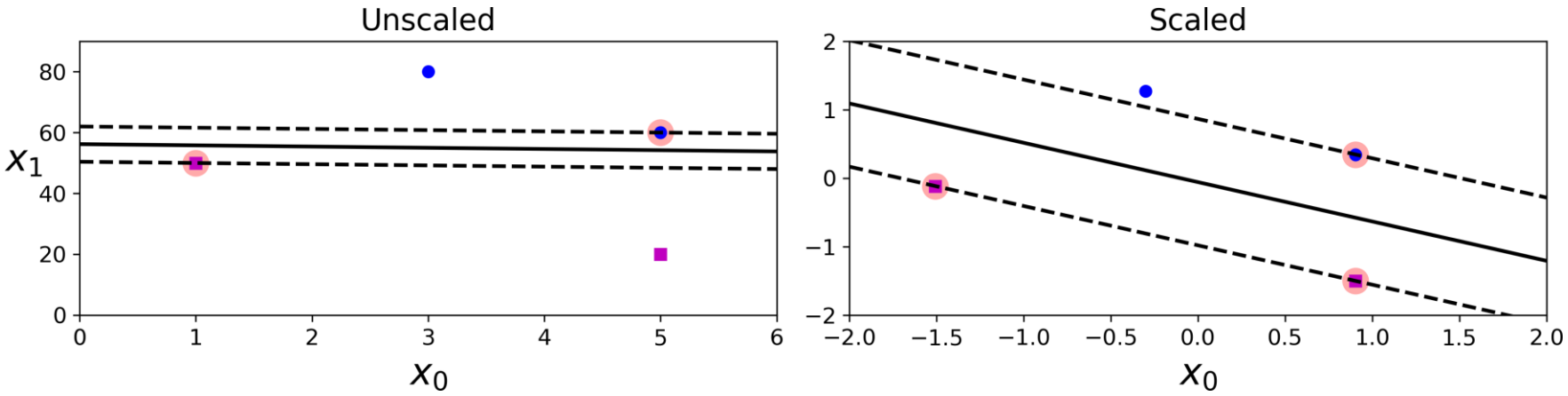
Support Vector Machine (SVM)

- Very **powerful** and **versatile** Machine Learning model, capable of performing **linear** or **nonlinear classification**, **regression**, and outlier detection.
- Well suited for classification of **complex** but **small-** or **medium-sized** datasets.
- SVM gives **large margin classification**.



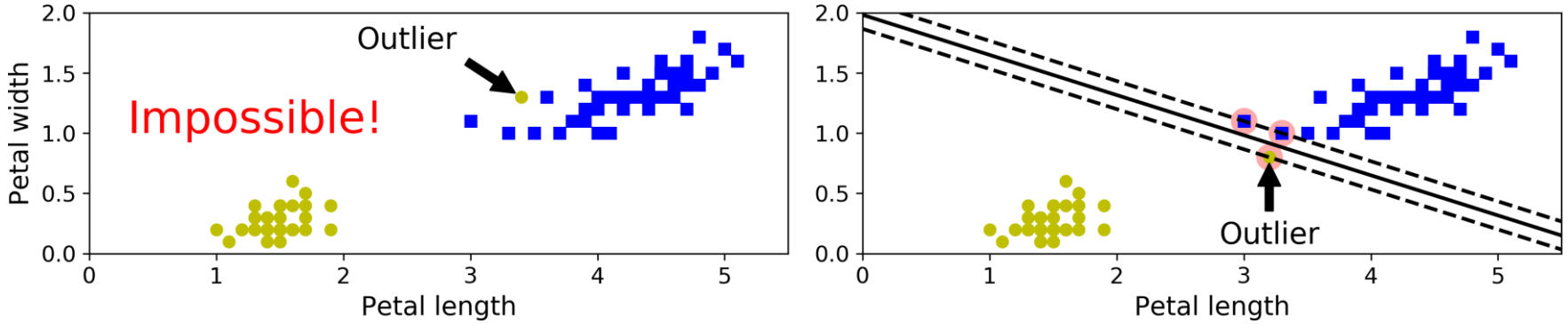
Linear SVM Classification

- The **decision boundary** is fully determined by the instances located on the edge. These instances are called the **support vectors**.
- SVMs are **sensitive** to the **feature scales**.



Soft Margin Classification

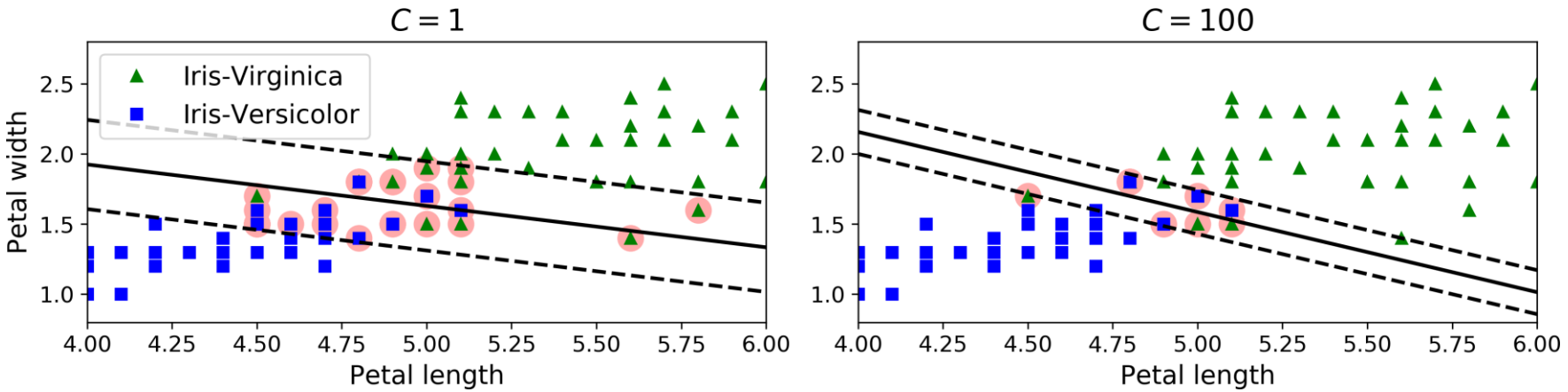
- **Hard margin classification** cannot handle linearly inseparable classes and is sensitive to outliers.



- **Soft margin classification** finds a balance between keeping the margin as large as possible and limiting the margin violations.

Soft Margin Classification

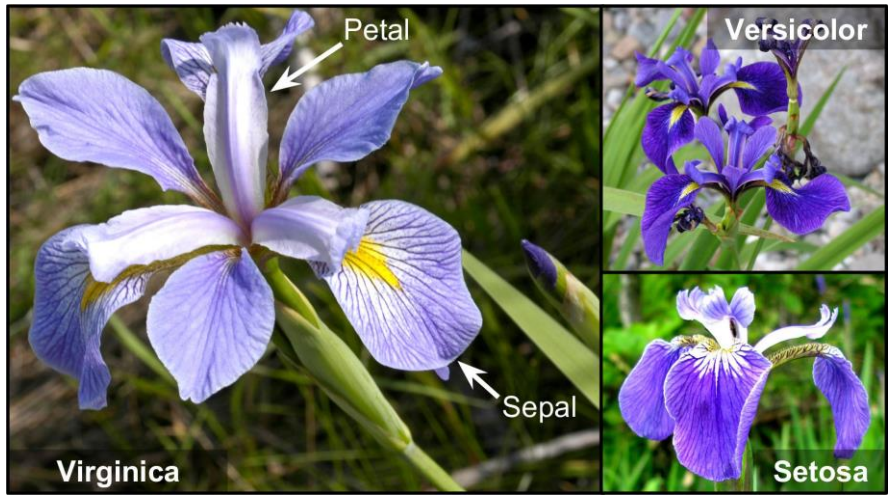
- You can control the number of violations using the **C hyperparameter**.



- If your SVM model is **overfitting**, you can try **regularizing** it by **reducing C**.

Iris Dataset

- A famous dataset that contains the sepal and petal length and width of **150 iris flowers** of three different species: **Setosa**, **Versicolor**, and **Virginica**.




```
>>> from sklearn import datasets
>>> iris = datasets.load_iris()
>>> list(iris.keys())
['data', 'target', 'target_names', 'DESCR', 'feature_names', 'filename']
```

SVM Classification Example

```
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
svm_clf = Pipeline([
        ("scaler", StandardScaler()),
        ("linear_svc", LinearSVC(C=1, loss="hinge")), ])
svm_clf.fit(X, y)

>>> svm_clf.predict([[5.5, 1.7]])
array([1.])
```

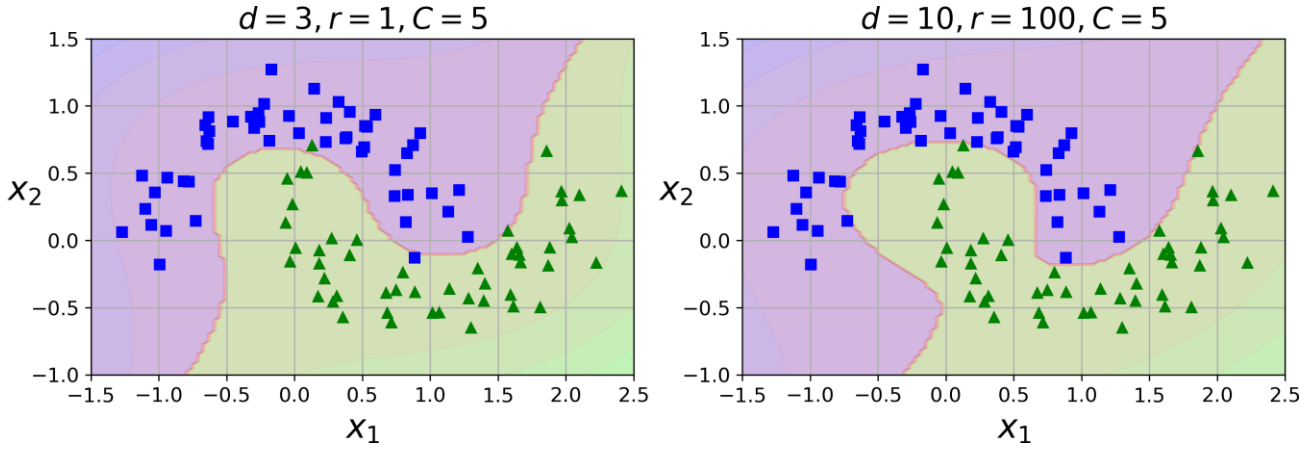
Used for maximum-margin classification.

Nonlinear SVM Classification

- The SVM class supports nonlinear classification using the **`kernel`** option.

Controls how much the model is influenced by high-degree polynomials versus low-degree

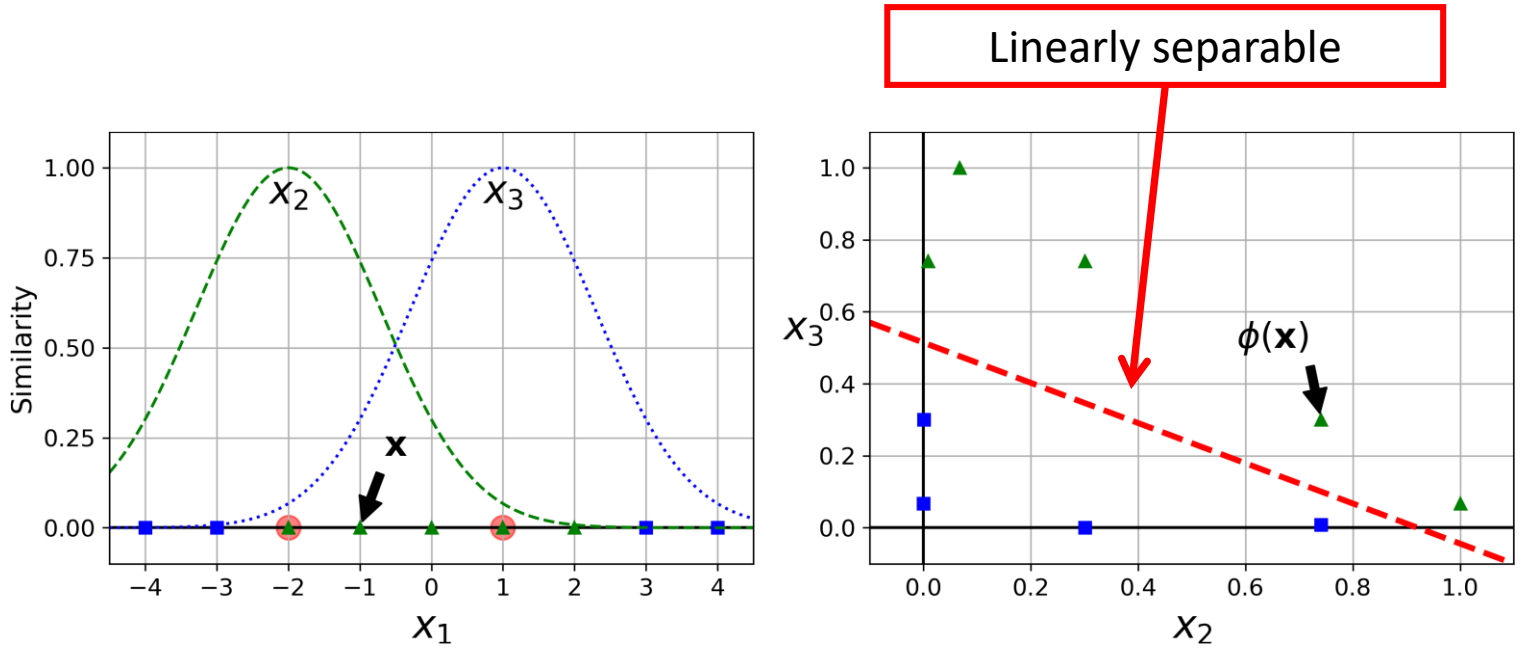
```
from sklearn.svm import SVC
poly_kernel_svm_clf = Pipeline([
        ("scaler", StandardScaler()),
        ("svm_clf", SVC(kernel="poly", degree=3, coef0=1, C=5))
    ])
poly_kernel_svm_clf.fit(X, y)
```



Gaussian Radial Basis Function

$$\phi_\gamma(\mathbf{x}, \ell) = \exp\left(-\gamma \| \mathbf{x} - \ell \|^2\right)$$

- The Gaussian RBF can be used to find **similarity features** (x_2 and x_3) of the one-dimensional dataset with two **landmarks** to it at $x_1 = -2$ and $x_1 = 1$



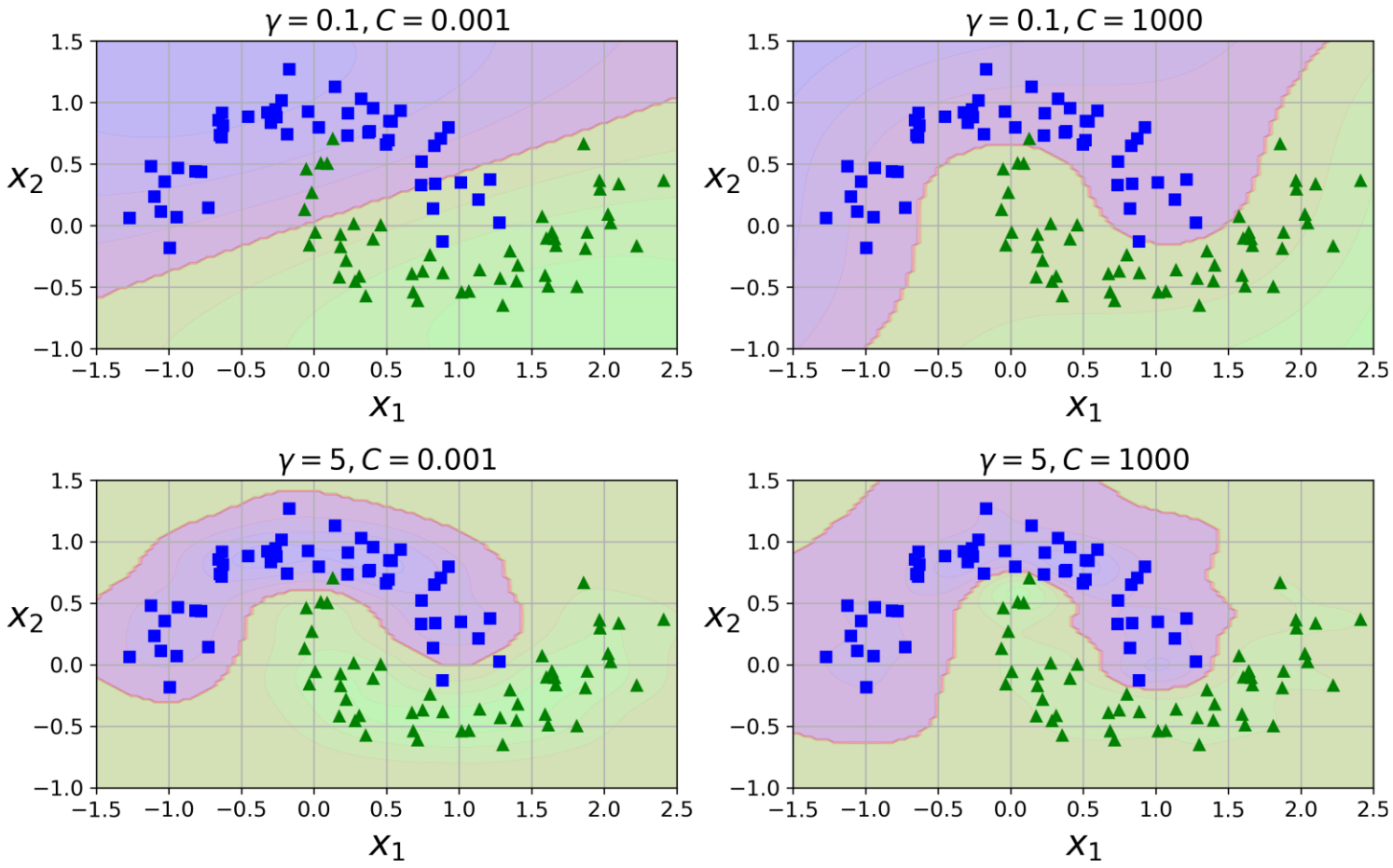
Gaussian RBF Kernel

- Is **popular** with SVM to **solve nonlinear problems**.

```
rbf_kernel_svm_clf = Pipeline([
        ("scaler", StandardScaler()),
        ("svm_clf", SVC(kernel="rbf", gamma=5, C=0.001))
    ])
rbf_kernel_svm_clf.fit(X, y)
```

- **Transforms** a training set with *m* instances and *n* features to *m* instances and *m* features.
- `gamma` and `C` are used for **regularization** with smaller values.

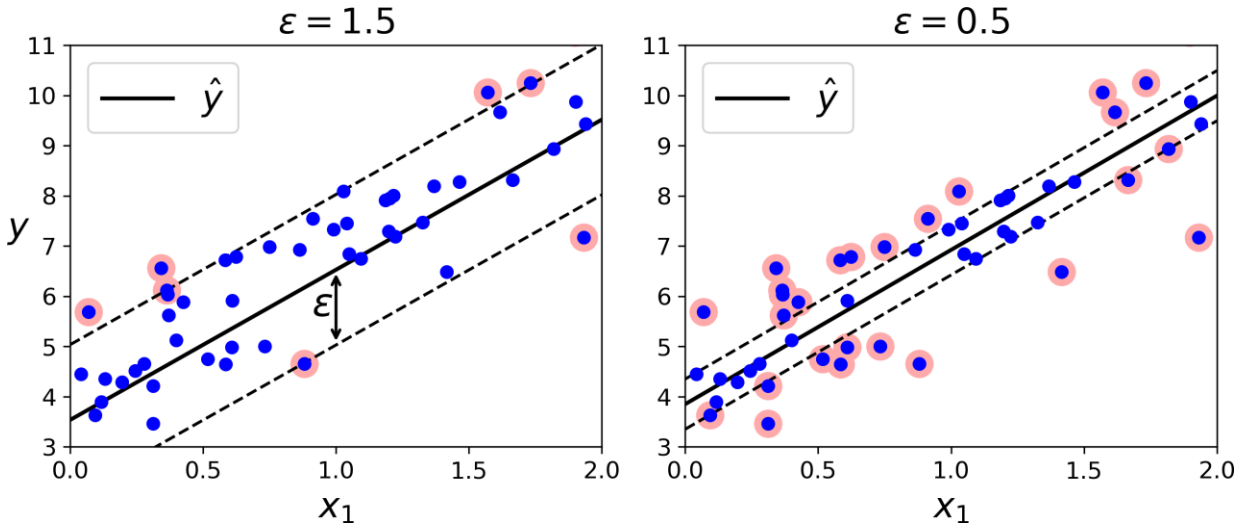
Gaussian RBF Kernel



Linear SVM Regression

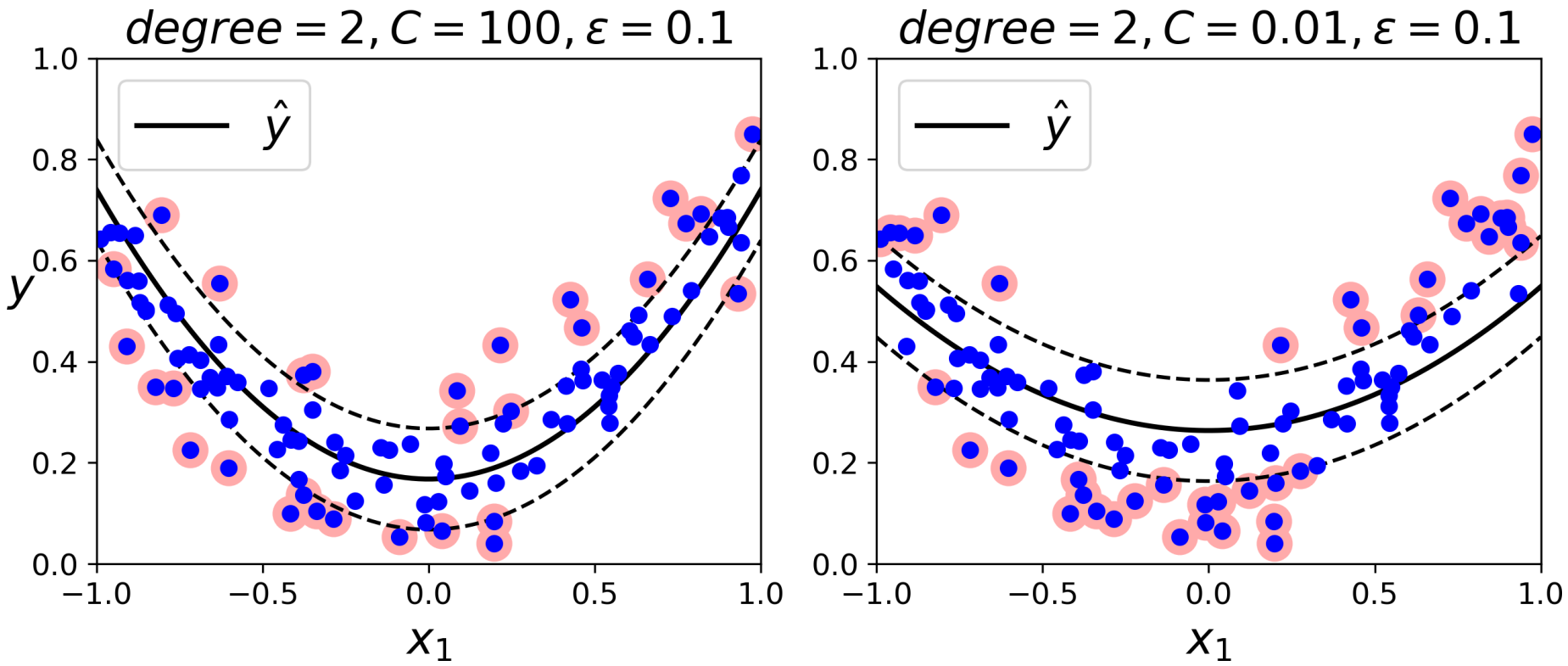
- Fits as many instances as possible on the margin while limiting margin violations. The width of the street is controlled by a hyperparameter ϵ.

```
from sklearn.svm import LinearSVR

svm_reg = LinearSVR(epsilon=1.5)
svm_reg.fit(X, y)
```



Nonlinear SVM Regression

```
from sklearn.svm import SVR

svm_poly_reg = SVR(kernel="poly", degree=2, C=100, epsilon=0.1)
svm_poly_reg.fit(X, y)
```



SVM Conclusion

- The `LinearSVC` has complexity of $O(m \times n)$.
- The **SVC** time complexity is usually between $O(m^2 \times n)$ and $O(m^3 \times n)$.
- This algorithm is perfect for complex but small or medium training sets. However, it scales well with the number of features.

Outline

1. k-Nearest Neighbors
2. Support Vector Machines
3. Decision Trees
4. Ensemble Learning and Random Forests
5. Exercises

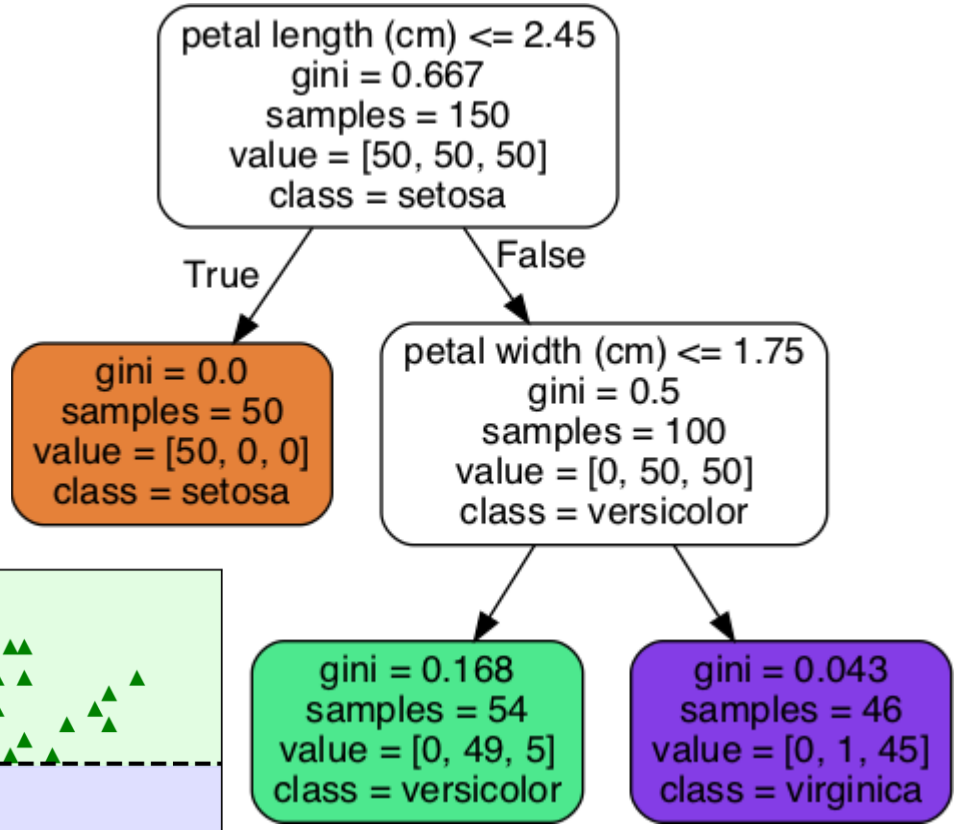
Decision Trees

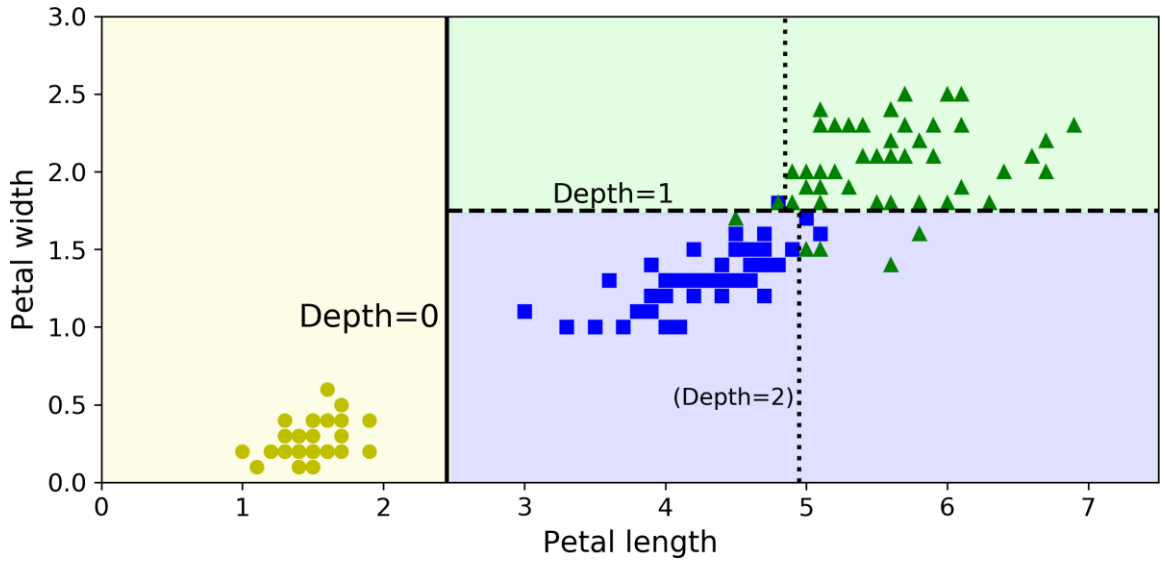
- Decision Trees are **versatile** Machine Learning algorithms that can perform both **classification** and **regression** tasks, and even multioutput tasks.
- They are very powerful algorithms, capable of fitting complex datasets.

```
from sklearn.datasets import load_iris
from sklearn.tree import DecisionTreeClassifier
iris = load_iris()
X = iris.data[:, 2:] # petal length and width
y = iris.target
tree_clf = DecisionTreeClassifier(max_depth=2)
tree_clf.fit(X, y)
```

Visualizing a Decision Tree

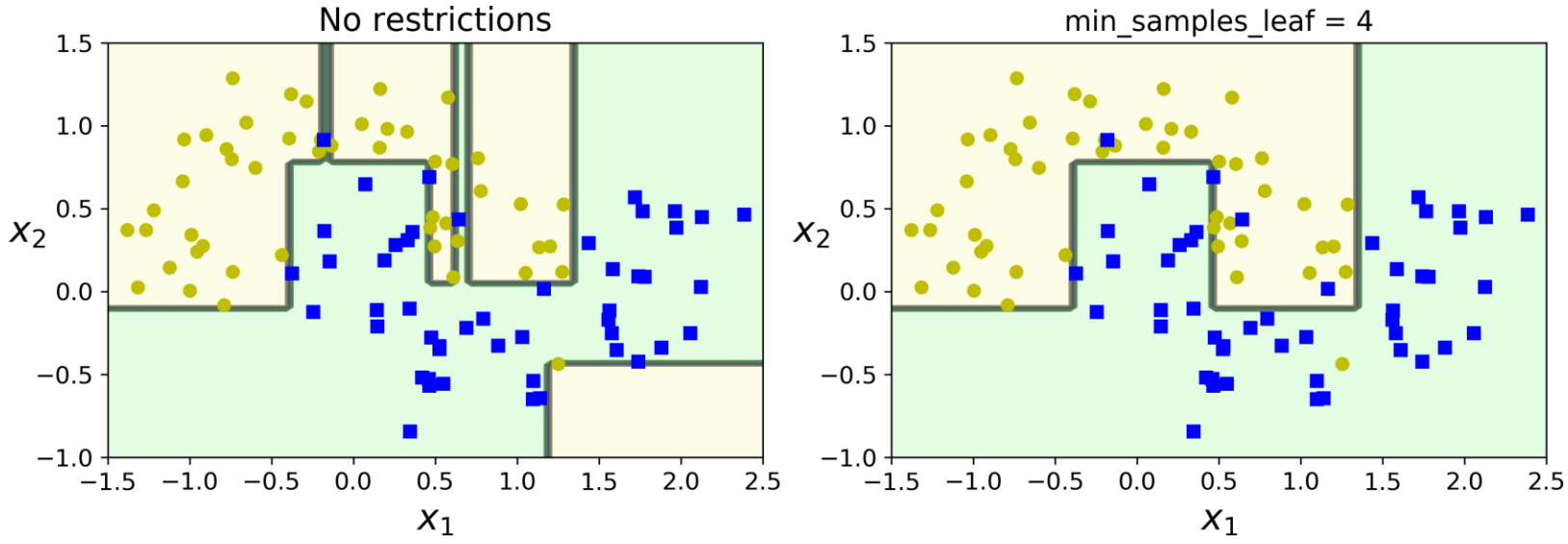
```
from sklearn.tree import export_graphviz

export_graphviz(
        tree_clf,
        out_file=image_path("iris_tree.dot"),
        feature_names=iris.feature_names[2:],
        class_names=iris.target_names,
        rounded=True,
        filled=True
    )
```





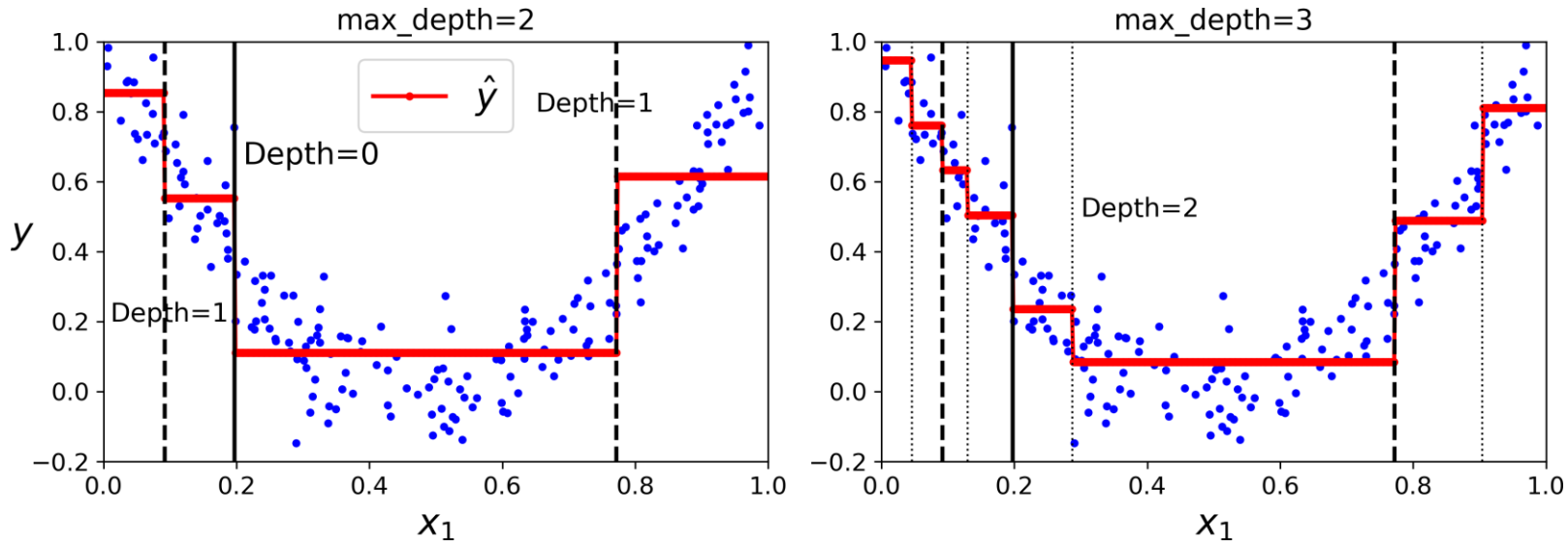
Regularization Hyperparameters

- Increase `min_*` or decrease `max_*`: **`max_depth=None, min_samples_split=2, min_samples_leaf=1, min_weight_fraction_leaf=0.0, max_features=None, max_leaf_nodes=None`**



Decision Trees Regression

```
from sklearn.tree import DecisionTreeRegressor
tree_reg = DecisionTreeRegressor(max_depth=2)
tree_reg.fit(X, y)
```



Outline

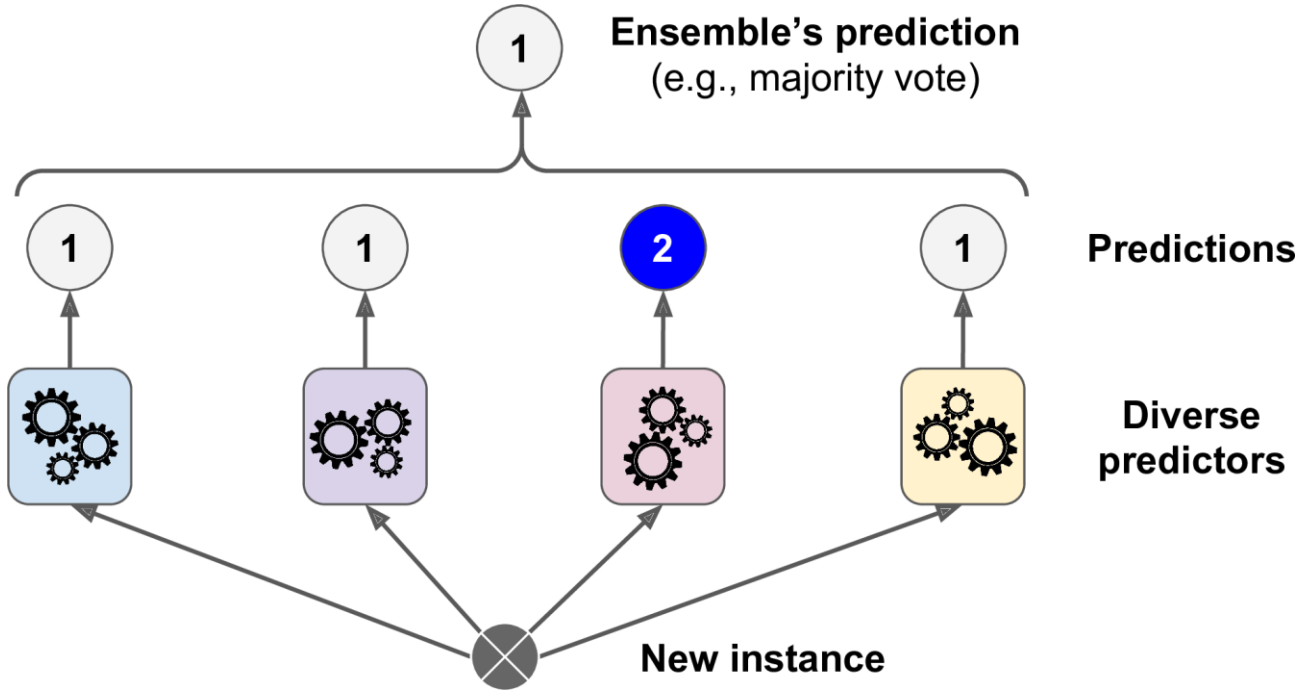
1. k-Nearest Neighbors
2. Support Vector Machines
3. Decision Trees
4. Ensemble Learning and Random Forests
5. Exercises

Ensemble Learning and Random Forests

- A group of predictors is called an **ensemble**.
- You can train a group of Decision Tree classifiers, each on a **different random subset** of the training set.
- To make predictions, obtain the predictions of all individual trees, then predict the class that gets the **most votes**.
- Such an ensemble of Decision Trees is called a **Random Forest**.

Voting Classifiers

- If each classifier is a **weak learner** (meaning it does only slightly better than random guessing), the ensemble can be a **strong learner** (achieving high accuracy).



Scikit-Learn Voting Classifier 1/2

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

log_clf = LogisticRegression()
rnd_clf = RandomForestClassifier()
svm_clf = SVC()

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

voting='soft' predict the class with the highest class probability

Scikit-Learn Voting Classifier 2/2

```
>>> 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 0.888
VotingClassifier 0.904
```

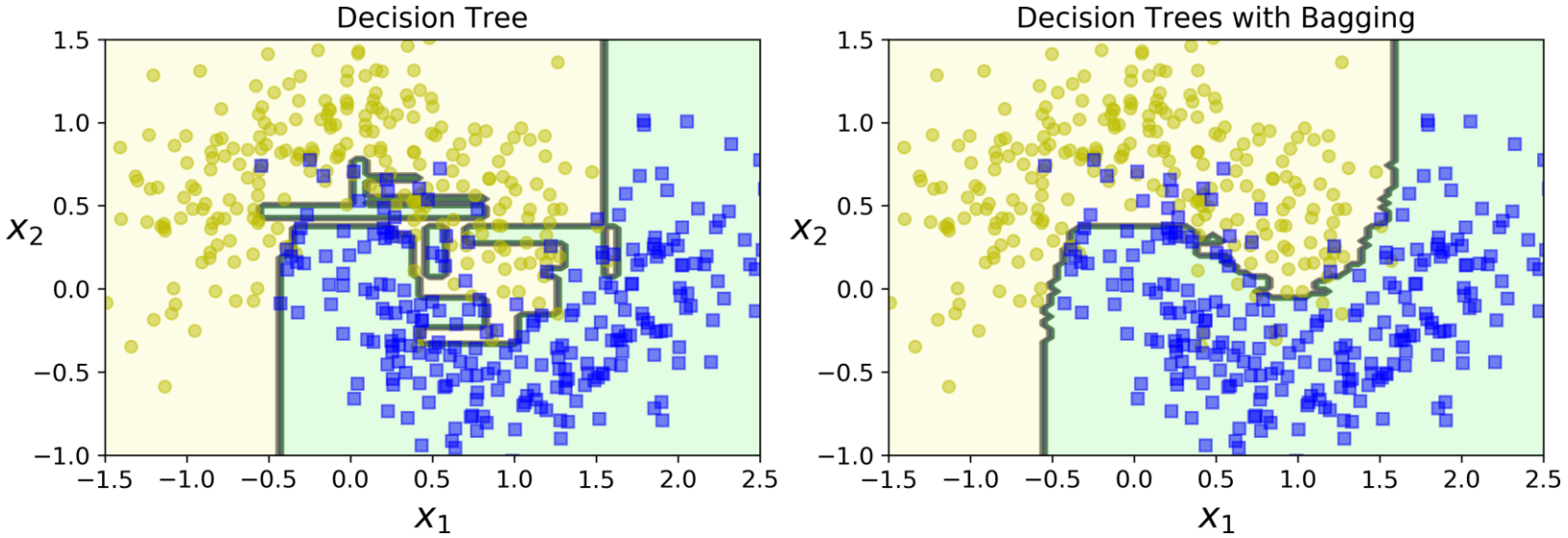
Bagging and Pasting

- Use the **same training algorithm** for every predictor, but train them on different random subsets of the training set.
- When sampling is performed **with** replacement, this method is called **bagging** (short for **bootstrap aggregating**).
- When sampling is performed **without** replacement, it is called **pasting**.
- The aggregation function is the most frequent prediction (**hard voting**) for classification, or the **average** for regression.

Bagging and Pasting

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

bag_clf = BaggingClassifier(
    DecisionTreeClassifier(), n_estimators=500,
    max_samples=100, bootstrap=True, n_jobs=-1)
bag_clf.fit(X_train, y_train)
y_pred = bag_clf.predict(X_test)
```

with replacement and use all available cores



Random Forests

- An ensemble of Decision Trees trained via the bagging with **`max_samples`** set to the size of the training set, and choosing the best random splits.

```
from sklearn.ensemble import RandomForestClassifier

rnd_clf = RandomForestClassifier(n_estimators=500, max_leaf_nodes=16, n_jobs=-1)
rnd_clf.fit(X_train, y_train)

y_pred_rf = rnd_clf.predict(X_test)
```

- Equivalent to:

```
bag_clf = BaggingClassifier(
    DecisionTreeClassifier(splitter="random", max_leaf_nodes=16),
    n_estimators=500, max_samples=1.0, bootstrap=True, n_jobs=-1)
```

Outline

1. k-Nearest Neighbors
2. Support Vector Machines
3. Decision Trees
4. Ensemble Learning and Random Forests
5. Exercises

Exercises

1. Train an **SVM classifier** on the **MNIST** dataset. Since SVM classifiers are binary classifiers, you will need to use one-versus-all to classify all 10 digits. You may want to tune the hyperparameters using small validation sets to speed up the process. What accuracy can you reach?

Exercises

2. Train and fine-tune a **Decision Tree** for the **moons dataset**.
   a) Generate a moons dataset using `make_moons(n_samples=10000, noise=0.4)`.
   b) Split it into a training set and a test set using `train_test_split()`.
   c) Use grid search with cross-validation (with the help of the `GridSearchCV` class) to find good hyperparameter values for a `DecisionTreeClassifier`. Hint: try various values for `max_leaf_nodes`.
   d) Train it on the full training set using these hyperparameters, and measure your model's performance on the test set. You should get roughly 85% to 87% accuracy.

Exercises

3. Load the **MNIST** data and split it into a training set, a validation set, and a test set (e.g., use 50,000 instances for training, 10,000 for validation, and 10,000 for testing). Then train various classifiers, such as a **Random Forest classifier**, an **Extra-Trees** classifier, and an **SVM**. Next, try to combine them into an **ensemble** that outperforms them all on the validation set, using a **soft** or **hard** voting classifier. Once you have found one, try it on the test set. How much better does it perform compared to the individual classifiers?

Summary

1. k-Nearest Neighbors
2. Support Vector Machines
3. Decision Trees
4. Ensemble Learning and Random Forests
5. Exercises