Co-funded by the
Erasmus+ Programme
of the European Union

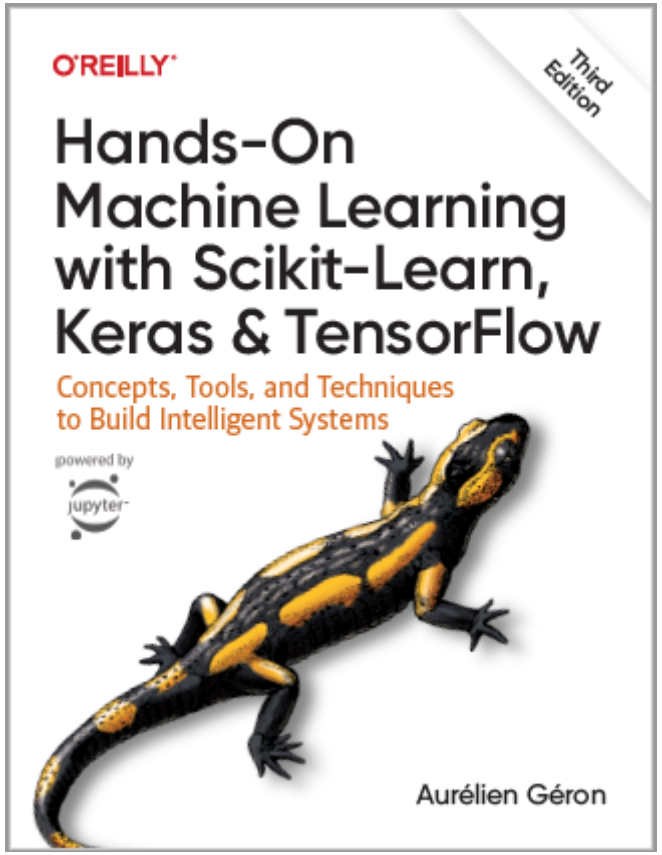
DeCAIR

Classical Techniques

Prof. Gheith Abandah

Reference

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

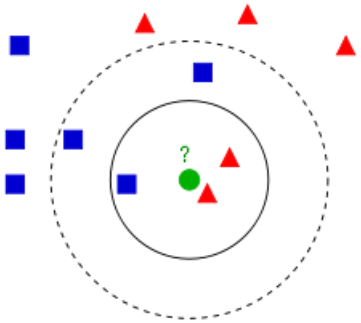


- Aurélien Géron, **Hands-On Machine Learning with Scikit-Learn, Keras and TensorFlow**, O'Reilly, 3rd Edition, 2022
  - Material: https://github.com/ageron/handson-ml3

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:

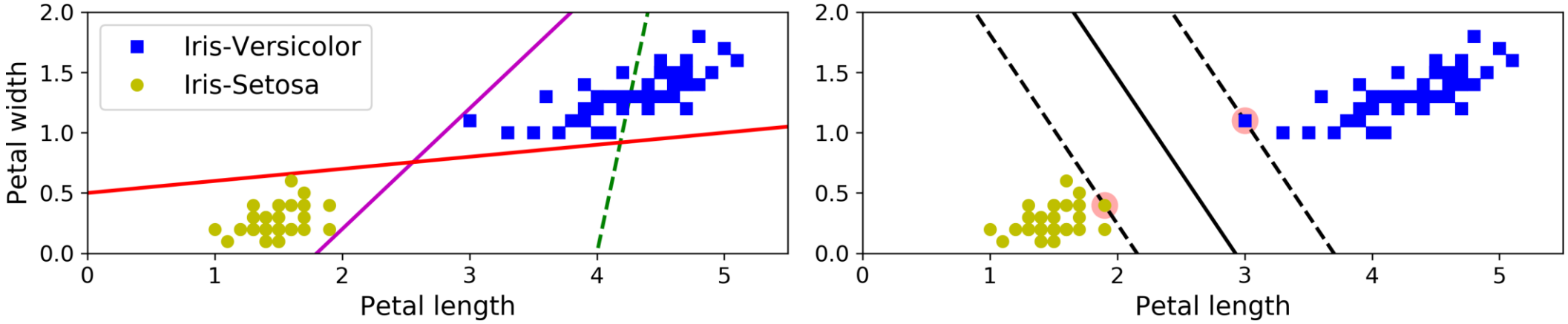
```
from sklearn.neighbors import KNeighborsRegressor
model = KNeighborsRegressor(n_neighbors=3)
model.fit(X, y)
```

Outline

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

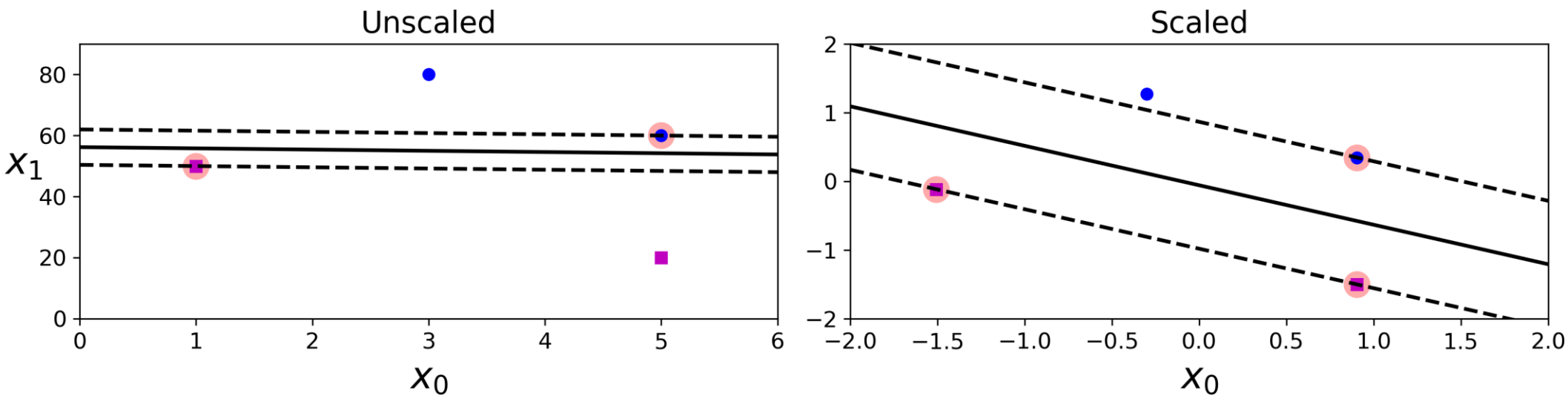
Support Vector Machine (SVM)

- 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**.



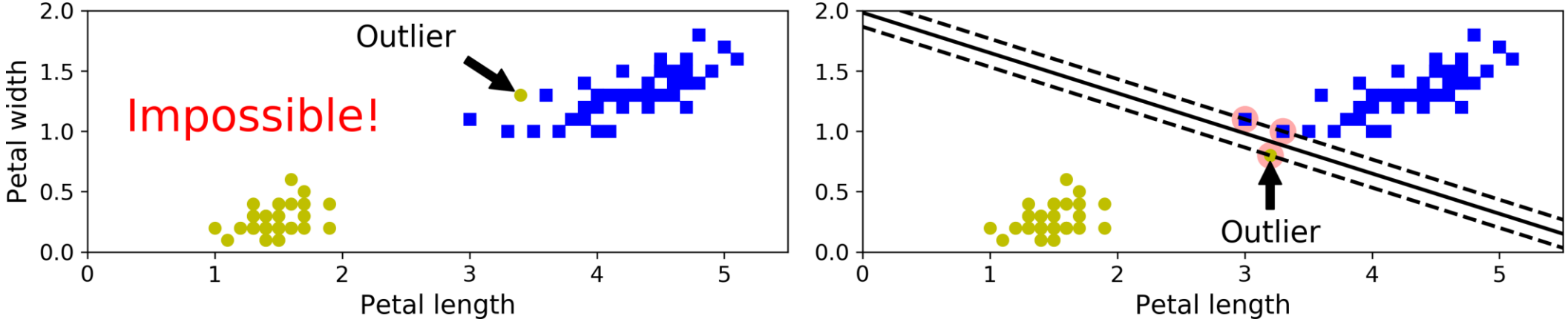
Linear SVM Classification

- 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

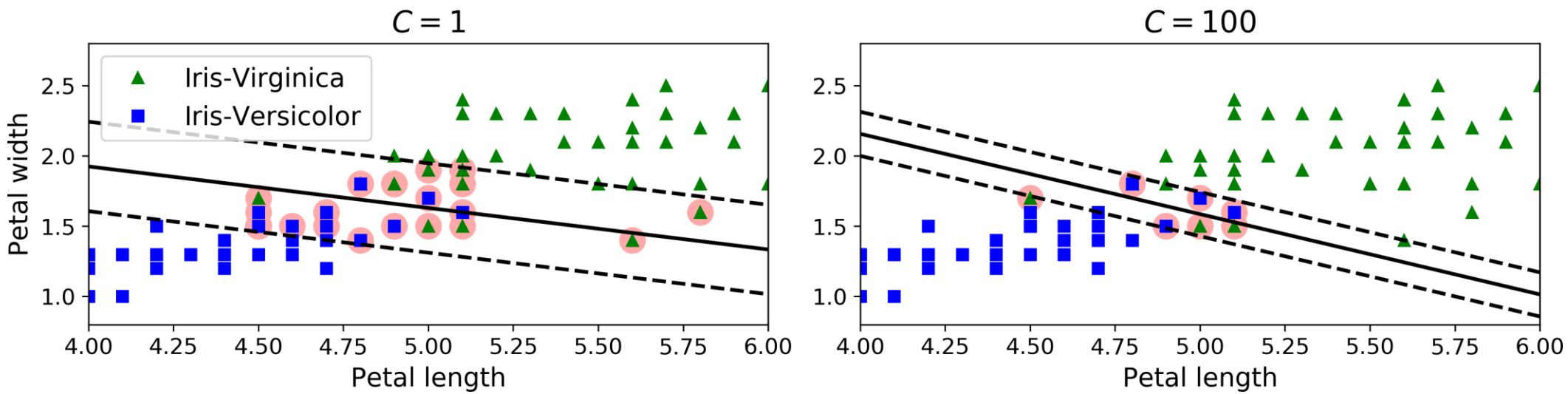
- **Hard margin classification** cannot handle linearly inseparable classes and is sensitive to outliers.



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

Soft Margin Classification

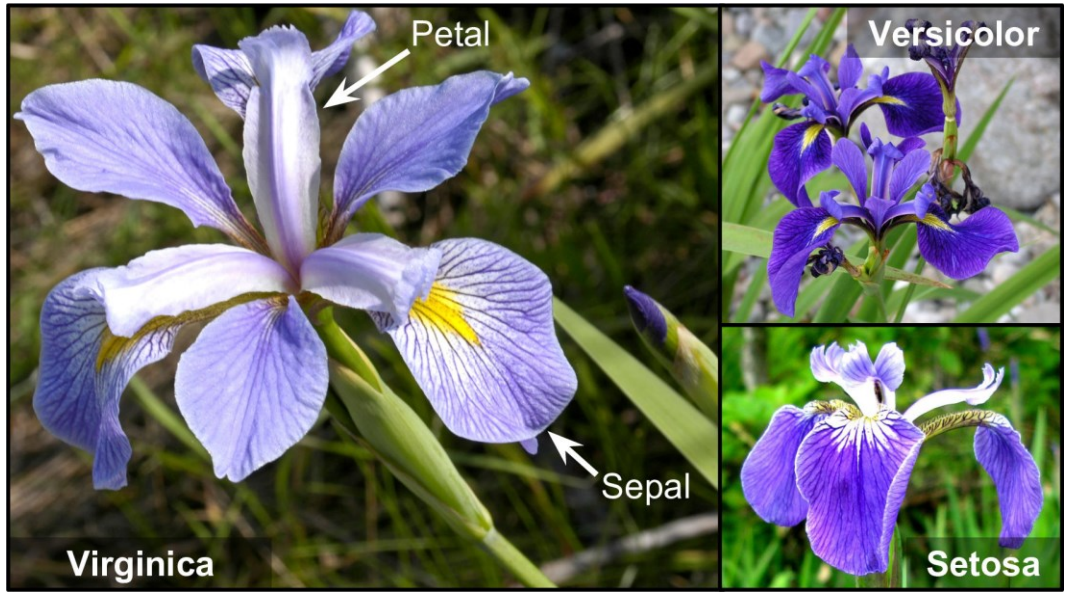
- You can control the number of violations using the **C hyperparameter**.



- 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**.



SVM Classification Example

```
from sklearn.datasets import load_iris
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.svm import LinearSVC

iris = load_iris(as_frame=True)
X = iris.data[["petal length (cm)",
               "petal width (cm)"]].values
y = (iris.target == 2) # Iris virginica

svm_clf = make_pipeline(StandardScaler(),
                        LinearSVC(C=1, random_state=42))
svm_clf.fit(X, y)
```

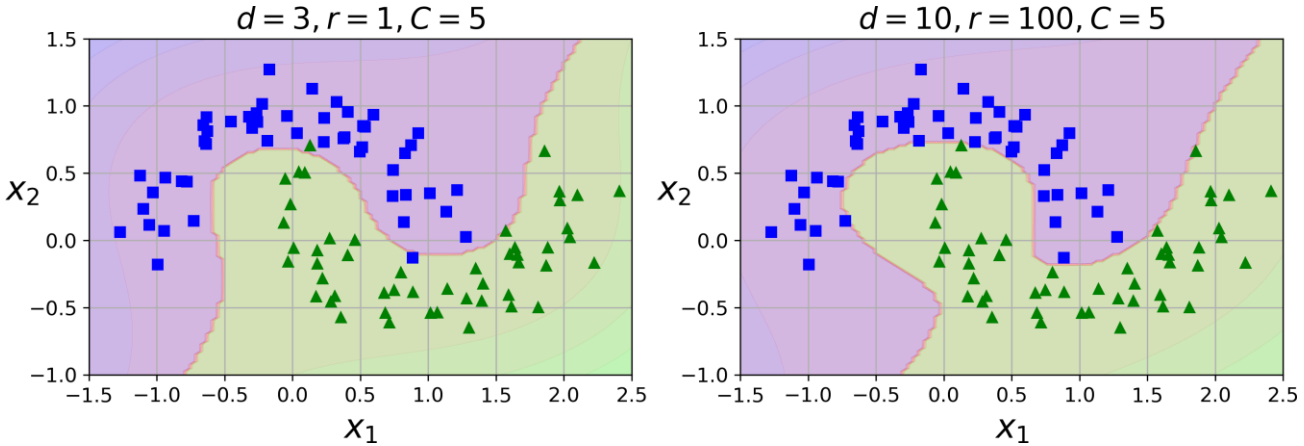
```
>>> X_new = [[5.5, 1.7], [5.0, 1.5]]
>>> svm_clf.predict(X_new)
array([ True, False])
>>> svm_clf.decision_function(X_new)
array([ 0.66163411, -0.22036063])
```

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

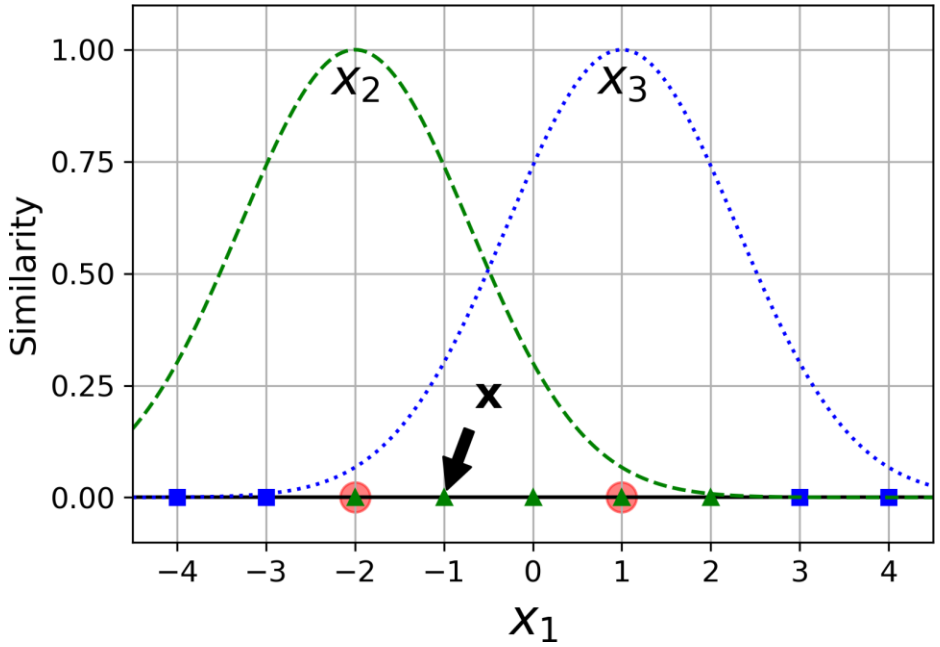
```
from sklearn.svm import SVC
poly_kernel_svm_clf = Pipeline([
        ("scaler", StandardScaler()),
        ("svm_clf", SVC(kernel="poly", degree=3, coef0=1, C=5))
    ])
poly_kernel_svm_clf.fit(X, y)
```

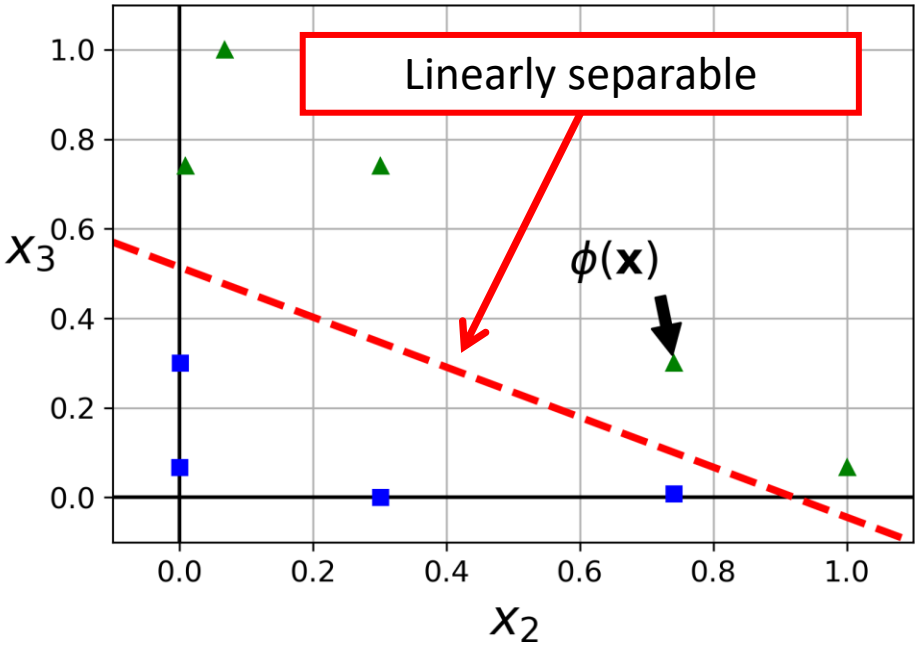


Gaussian Radial Basis Function

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

- 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 $x_1 = 1$





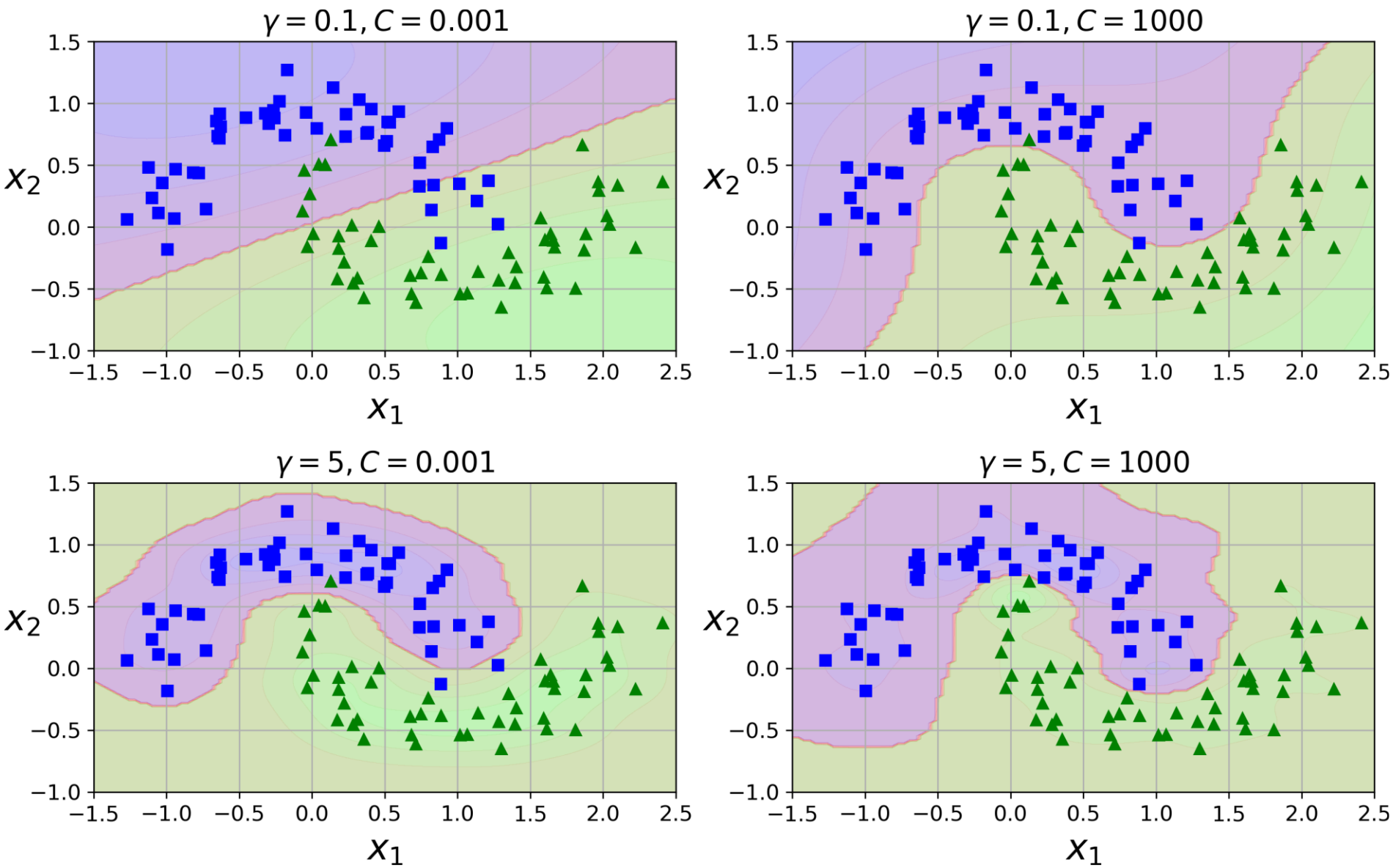
Gaussian RBF Kernel

- 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.

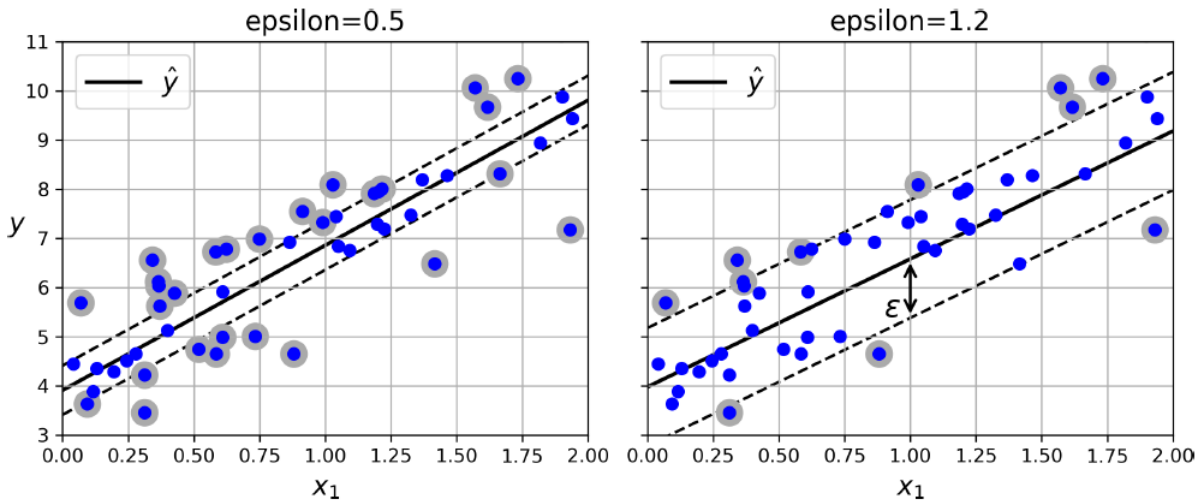
Gaussian RBF Kernel



Linear SVM Regression

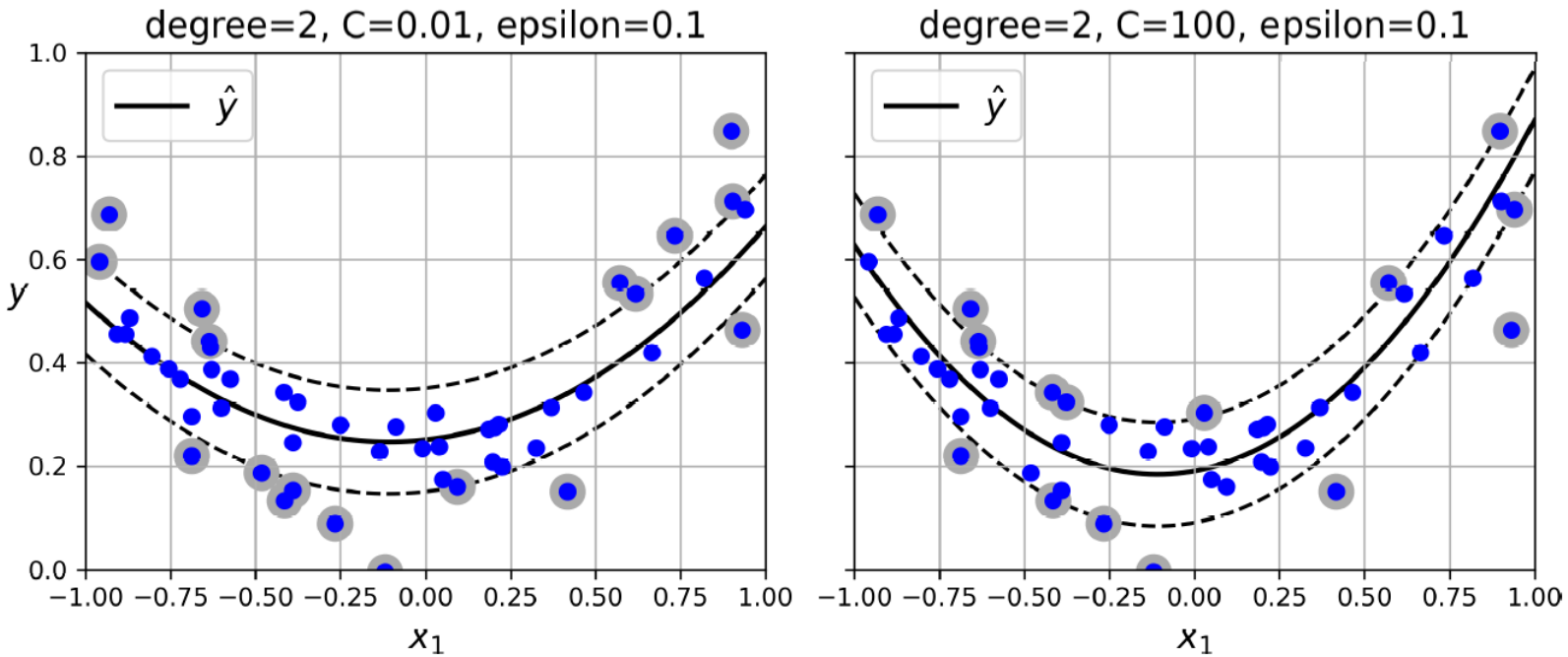
- Fits as many instances as possible on the margin while limiting margin violations. The width of the street is controlled by a hyperparameter ϵ.

```
from sklearn.svm import LinearSVR

X, y = [...]  # a linear dataset
svm_reg = make_pipeline(StandardScaler(),
                        LinearSVR(epsilon=0.5, random_state=42))
svm_reg.fit(X, y)
```



Nonlinear SVM Regression

```
from sklearn.svm import SVR

X, y = [...]  # a quadratic dataset
svm_poly_reg = make_pipeline(StandardScaler(),
                             SVR(kernel="poly", degree=2, C=0.01, 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

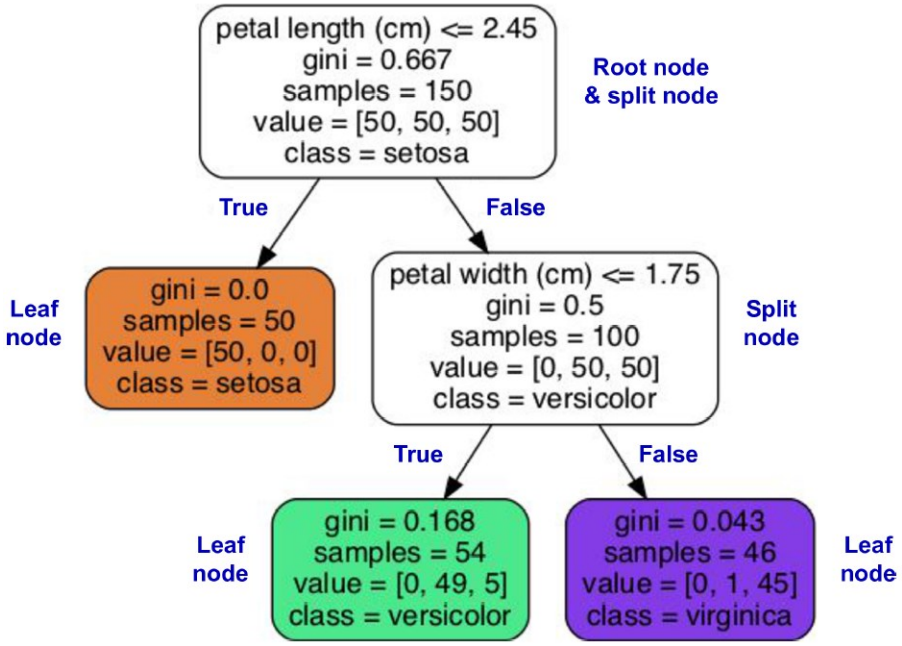
Decision Trees

- 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.

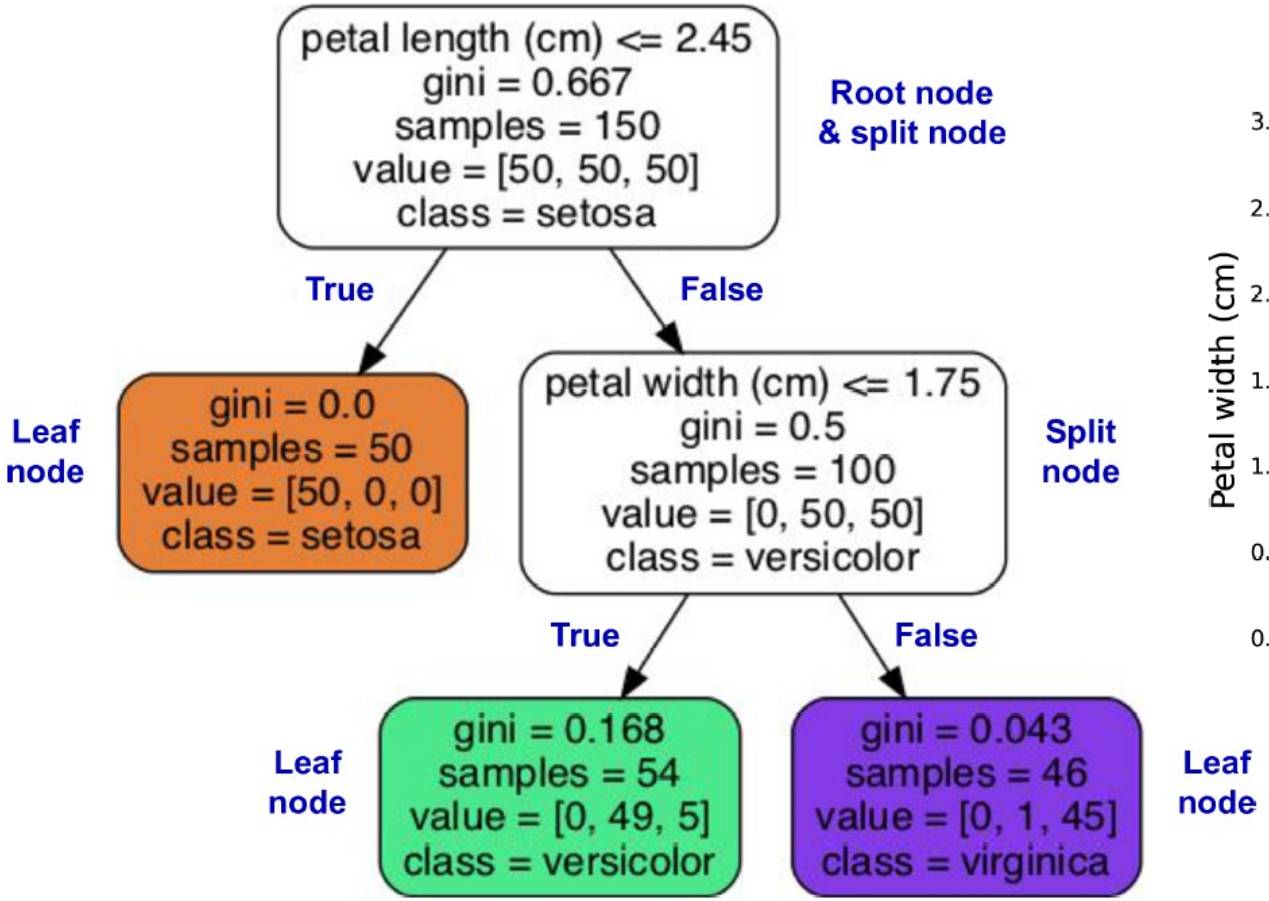
```
from sklearn.datasets import load_iris
from sklearn.tree import DecisionTreeClassifier
iris = load_iris(as_frame=True)
X_iris = iris.data[["petal length (cm)", "petal width (cm)"]].values
y_iris = iris.target
tree_clf = DecisionTreeClassifier(max_depth=2, random_state=42)
tree_clf.fit(X_iris, y_iris)
```

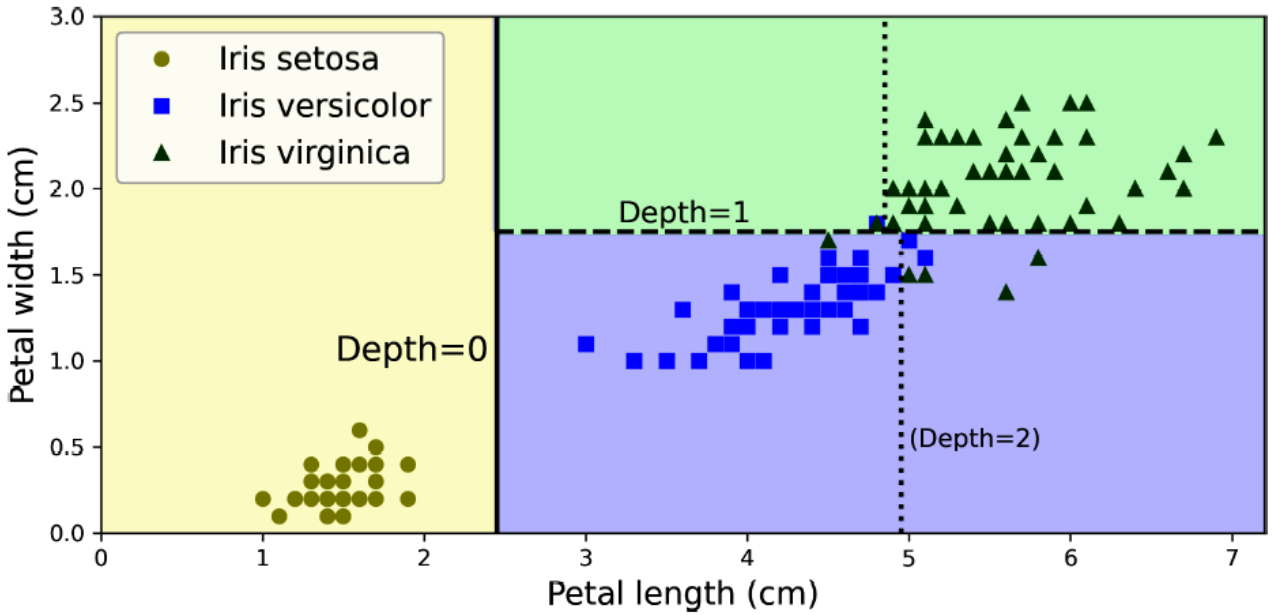
Visualizing a Decision Tree

```
from sklearn.tree import export_graphviz

export_graphviz(
        tree_clf,
        out_file="iris_tree.dot",
        feature_names=["petal length (cm)", "petal width (cm)"],
        class_names=iris.target_names,
        rounded=True,
        filled=True
    )
```



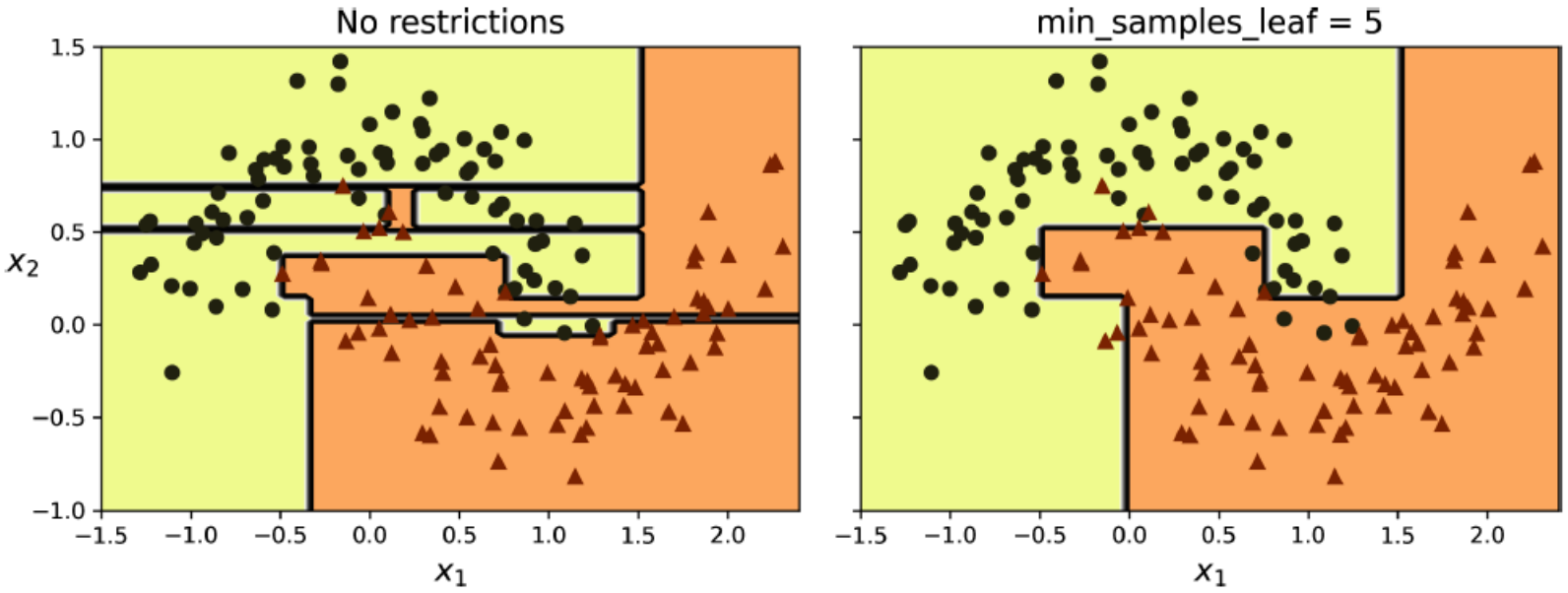
Visualizing a Decision Tree





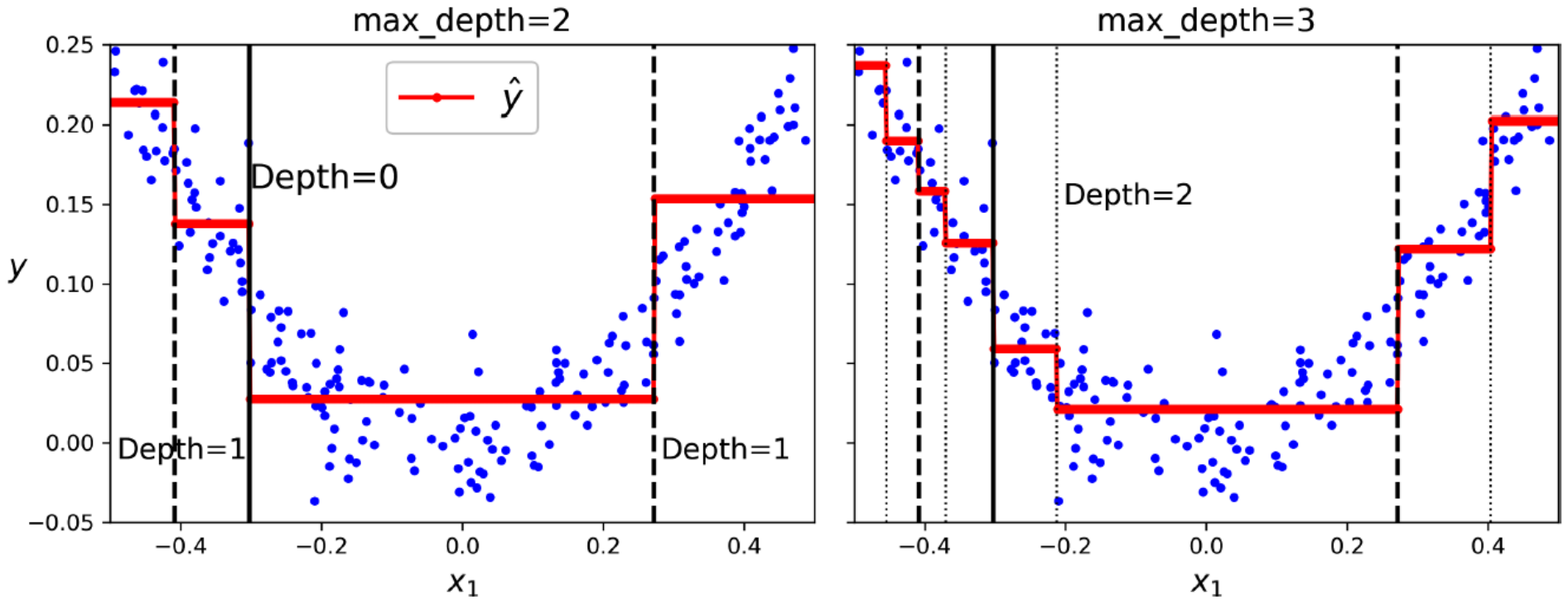
Regularization Hyperparameters

- 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

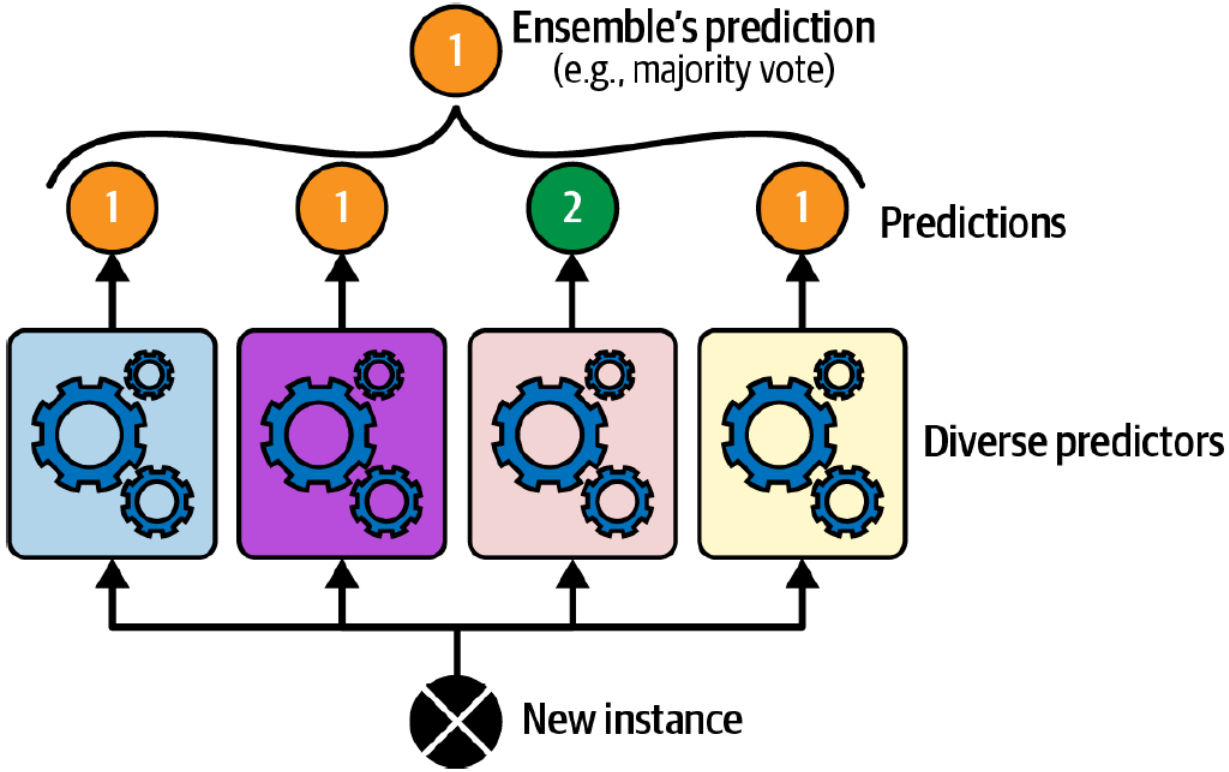
1. k-Nearest Neighbors
2. Support Vector Machines
3. Decision Trees
4. Ensemble Learning and Random Forests
5. Exercises

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 (**hard voting**),
- or predict the class with the highest-class probability (**soft voting**).
- 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.datasets import make_moons
from sklearn.ensemble import RandomForestClassifier, VotingClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.svm import SVC

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_clf = VotingClassifier(
    estimators=[
        ('lr', LogisticRegression(random_state=42)),
        ('rf', RandomForestClassifier(random_state=42)),
        ('svc', SVC(random_state=42))
    ]
)
voting_clf.fit(X_train, y_train)
```

Scikit-Learn Voting Classifier 2/2

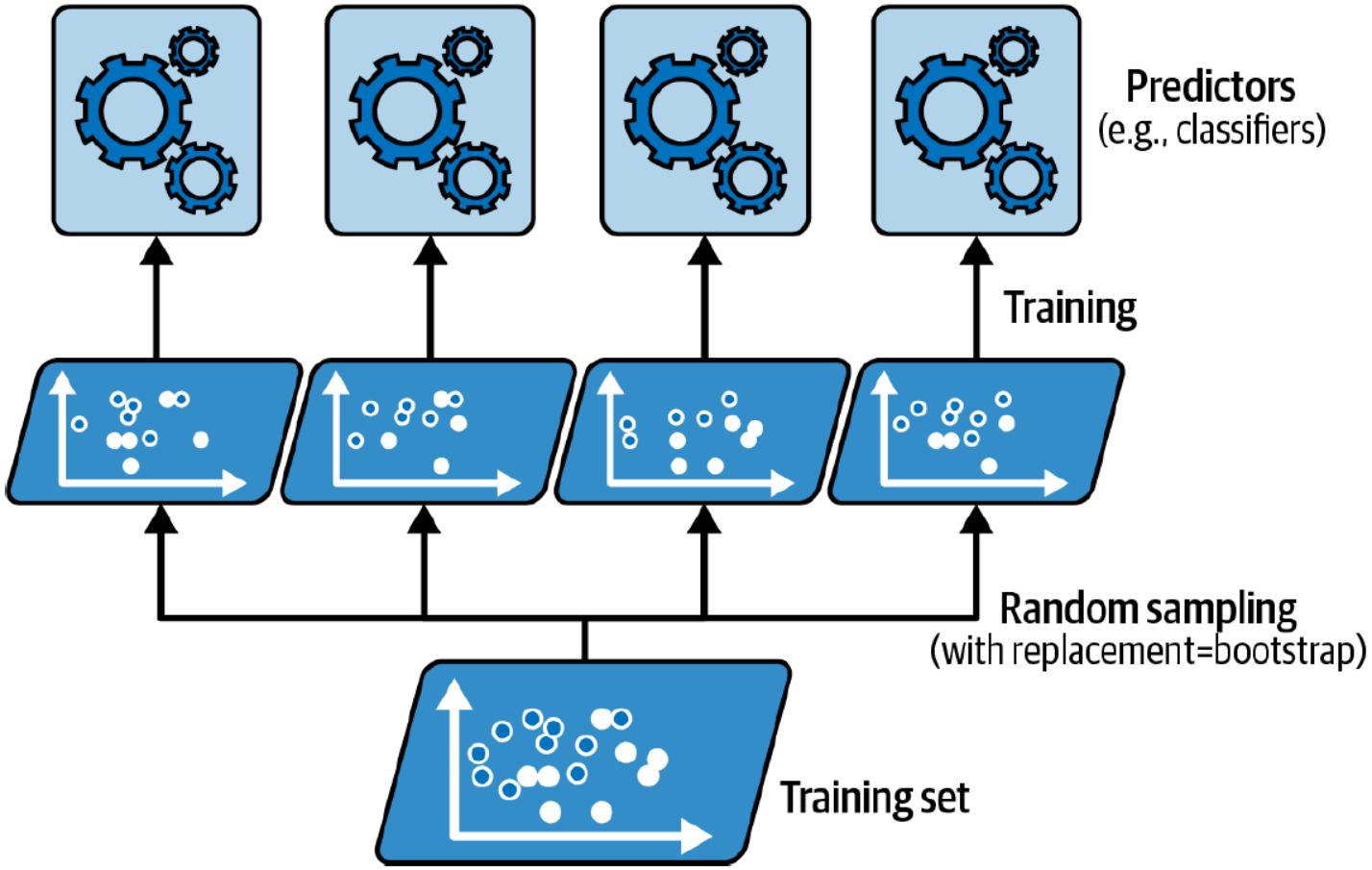
```
>>> for name, clf in voting_clf.named_estimators_.items():
...     print(name, "=", clf.score(X_test, y_test))
...
lr = 0.864
rf = 0.896
svc = 0.896
```

```
>>> voting_clf.score(X_test, y_test)
0.912
```

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, highest probability (**soft voting**), or the **average** for regression.

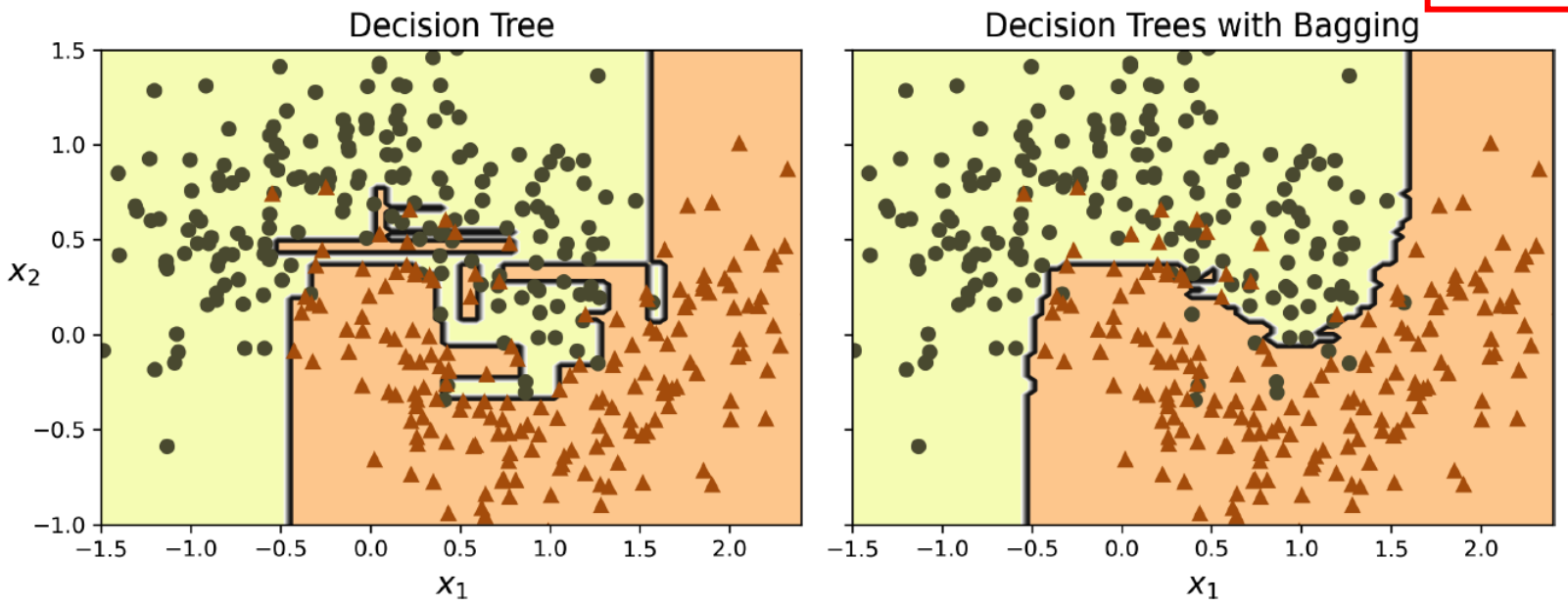
Bagging Demonstration



Bagging and Pasting

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

bag_clf = BaggingClassifier(DecisionTreeClassifier(), n_estimators=500,
                            max_samples=100, n_jobs=-1, random_state=42)
bag_clf.fit(X_train, y_train)
```

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:

It samples $\sqrt{n}$ features.

```
bag_clf = BaggingClassifier(
    DecisionTreeClassifier(max_features="sqrt", max_leaf_nodes=16),
    n_estimators=500, n_jobs=-1, random_state=42)
```

Outline

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

Exercises

1. Train and fine-tune an **SVM regressor** on the California housing dataset. You can use the original dataset rather than the tweaked version we used in Chapter 2, which you can load using `sklearn.datasets.fetch_california_housing()`. The targets represent hundreds of thousands of dollars. Since there are over 20,000 instances, SVMs can be slow, so for hyperparameter tuning you should use far fewer instances (*e.g.*, 2,000) to test many more hyperparameter combinations. What is your best model's RMSE?

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** dataset and split it into a training set and a test set (take the first 60,000 instances for training, and the remaining 10,000 for testing). Train a **random forest classifier** on the dataset and time how long it takes, then evaluate the resulting model on the test set.

Summary

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