机器学习基础(II)

第6讲：支持向量机、随机森林、集成学习等

主讲：严钢 魏泽勇 张潇竹

助教：张鑫洁

2022/10/24

➢ **Part 1: Fundamentals of Machine Learning** (Scikit-Learn)

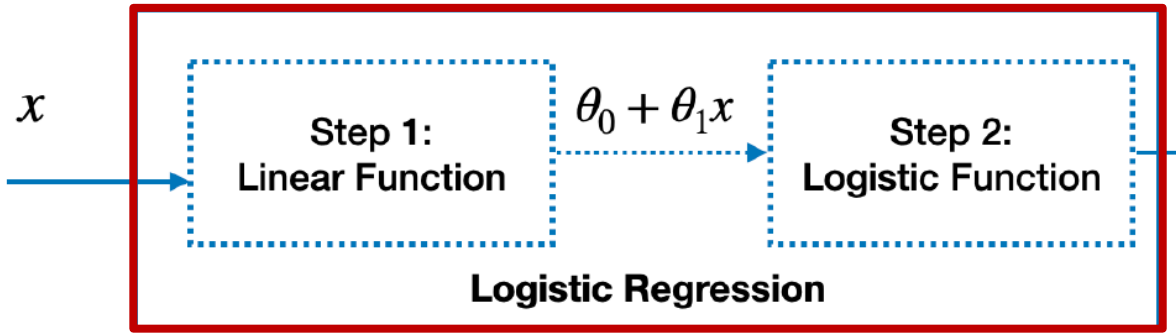
- Review of Python's main scientific libraries: NumPy, pandas, and Matplotlib
- Steps in a typical machine learning project
- Learning by fitting a model to data and optimizing a cost function
- Challenges of using machine learning systems

➢ **Part 2: Neural Networks and Deep Learning** (TensorFlow, PyTorch)

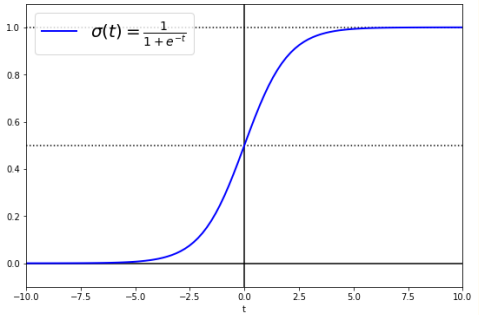
- What neural nets are and what they are good for
- Building and training feedforward neural nets (basics and implementation)
- CNN, RNN, GNN, etc (mainly fundamental concepts and cases)

➢ **Part 3: Case study at the interface between ML and Physics**

Logistic Regression and Softmax (Classifier - actually)



$$P(C_1 \mid x) = \frac{1}{1 + e^{-(\theta_0 + \theta_1 x)}}$$

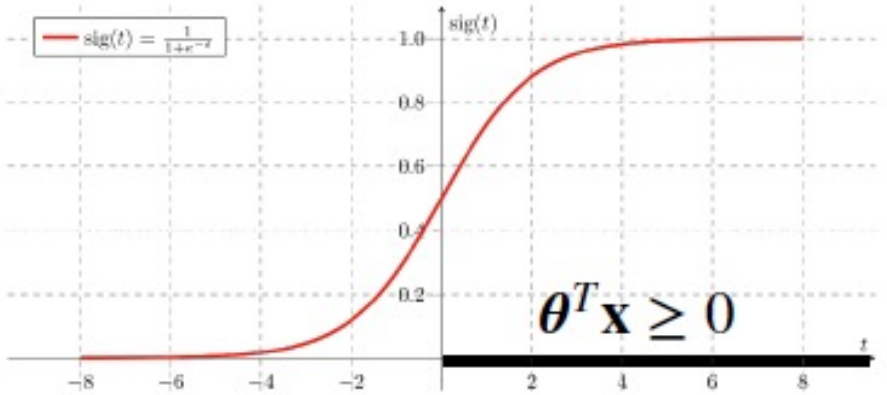


- Let us find $P(C_0 \mid x)$
  - There are two possible outcomes

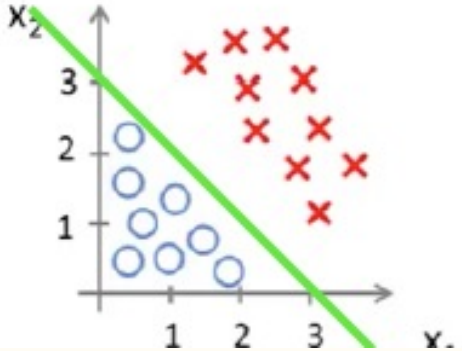
$$P(C_0 \mid x) + P(C_1 \mid x) = 1 \Rightarrow P(C_0 \mid x) = 1 - P(C_1 \mid x)$$

How to find decision boundary?

- Recall that we predict $y = 1$ if $\hat{p} = h(\mathbf{x}) = \sigma(\boldsymbol{\theta}^T\mathbf{x}) \geq 0.5$



- Example: consider $h(\mathbf{x}) = \sigma(\theta_0 + \theta_1 x_1 + \theta_2 x_2)$ with $\theta_0 = -3$, $\theta_1 = \theta_2 = 1$



Softmax regression

- The logistic regression model can be generalized to support multiple classes
- Let K be the number of classes and compute a score $s_k(\mathbf{x})$ for each $k \in \{1,\ldots,K\}$

$$s_k(\mathbf{x}) = \mathbf{x}^T \boldsymbol{\theta}^{(k)}$$

- Estimate the probability that the instance $\mathbf{x}$ belongs to class k

$$\hat{p}_k = \sigma(s(\mathbf{x}))_k = \frac{\exp(s_k(\mathbf{x}))}{\sum_{i=1}^{K} \exp(s_j(\mathbf{x}))}$$

Prediction

- Softmax regression predicts the class with the highest estimated probability

$$\hat{p} = \text{argmax}\ \sigma(s(\mathbf{x}))_k = \text{argmax}\ s_k(\mathbf{x}) = \text{argmax}\ \mathbf{x}^T\boldsymbol{\theta}^{(k)}$$

- Cross entropy cost function

$$J(\boldsymbol{\Theta}) = -\frac{1}{n}\sum_{i=1}^{n}\sum_{j=1}^{K} y_k^{(i)} \log(\hat{p}_k^{(i)})$$

  - where $y_k^{(i)} \in \{0,1\}$ is the target probability

Softmax regression in Scikit-Learn

```
from sklearn.linear_model import LogisticRegression
X = iris["data"][:, (2, 3)]  # petal length, petal width
y = iris["target"]

softmax_reg = LogisticRegression(multi_class="multinomial",solver="lbfgs"
softmax_reg.fit(X, y)
```

```
softmax_reg.predict([[5, 2]])
```

```
array([2])
```

```
softmax_reg.predict_proba([[5, 2]])
```

```
array([[6.38014896e-07, 5.74929995e-02, 9.42506362e-01]])
```

multi_class : *{'auto', 'ovr', 'multinomial'}, default='auto'*

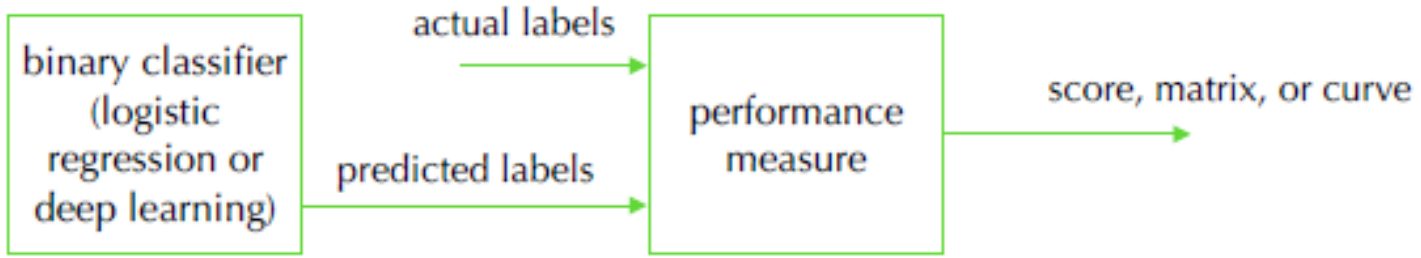
If the option chosen is 'ovr', then a binary problem is fit for each label. For 'multinomial' the loss minimised is the multinomial loss fit across the entire probability distribution, *even when the data is binary*. 'multinomial' is unavailable when solver='liblinear'. 'auto' selects 'ovr' if the data is binary, or if solver='liblinear', and otherwise selects 'multinomial'.

New in version 0.18: Stochastic Average Gradient descent solver for 'multinomial' case.

Performance measures for classifiers

Evaluating performance of classifiers

- In this discussion, we focus on general binary classification tasks



- We work with MNIST dataset consisting of 70,000 images of handwritten digits

```
from sklearn.datasets import fetch_openml
mnist = fetch_openml('mnist_784', version=1)
mnist.keys()
```

```
dict_keys(['data', 'target', 'frame', 'feature_names',
```

```
X, y = mnist["data"], mnist["target"]
print(X.shape, y.shape)
```

```
(70000, 784) (70000,)
```

Visualization

Converting to binary classification

- We simplify the problem by identifying one digit (the number 5)
  - Binary classifier with two classes: 5 or not-5

```
X_train, X_test, y_train, y_test = X[:60000], X[60000:], y[:60000], y[60000:]
y_train_5 = (y_train == 5)
y_test_5 = (y_test == 5)
y_train_5[:12]
```

```
array([ True, False, False, False, False, False, False, False, False,
       False, False,  True])
```

- Training classifier

```
from sklearn.linear_model import SGDClassifier

sgd_clf = SGDClassifier(max_iter=1000, tol=1e-3, random_state=42)
sgd_clf.fit(X_train, y_train_5)
```

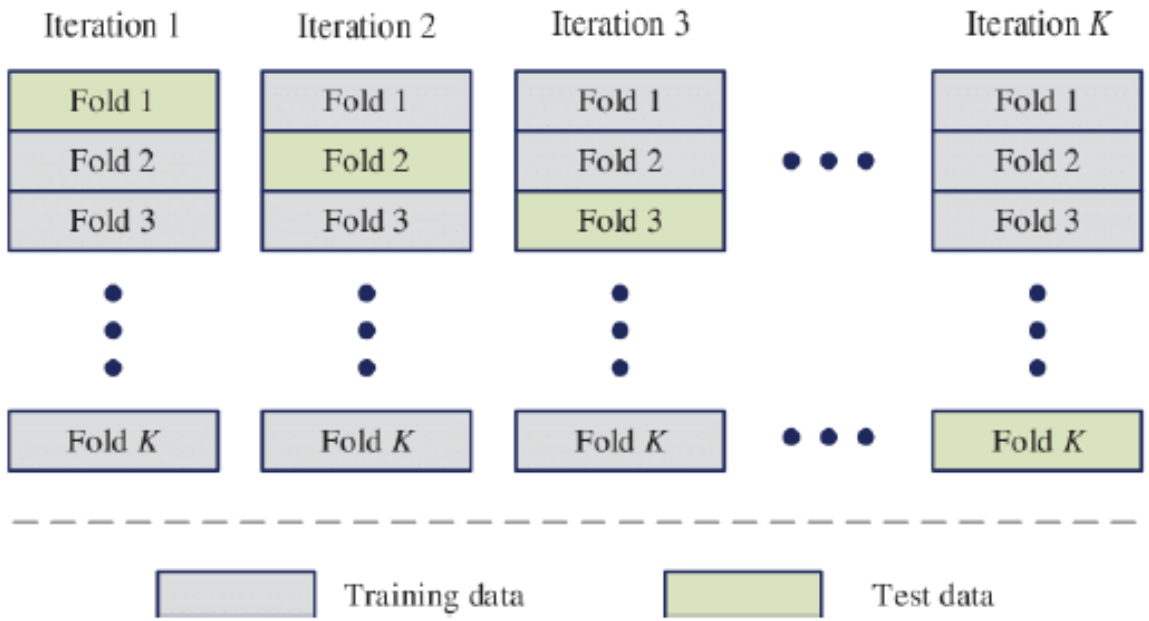
- Predicted labels

```
sgd_clf.predict([some_digit])
```

```
array([ True])
```

Implementing cross-validation

- Recall that we divide the data into K sets or "folds" and use one for testing



- To implement cross-validation, we use (1) our own code and (2) off the shelf scikit-learn function

```
from sklearn.model_selection import cross_val_score
cross_val_score(sgd_clf, X_train, y_train_5, cv=3, scoring="accuracy")
```

```
array([0.95035, 0.96035, 0.9604 ])
```

Analyzing our results

- The accuracy of our classifier is above 95%.
  - Did we really find an accurate classifier?
  - Let's do a simple experiment and define a classifier that never returns 5

```
from sklearn.base import BaseEstimator
class Never5Classifier(BaseEstimator):
    def fit(self, X, y=None):
        pass
    def predict(self, X):
        return np.zeros((len(X), 1), dtype=bool)
```

  - Perform cross-validation as before

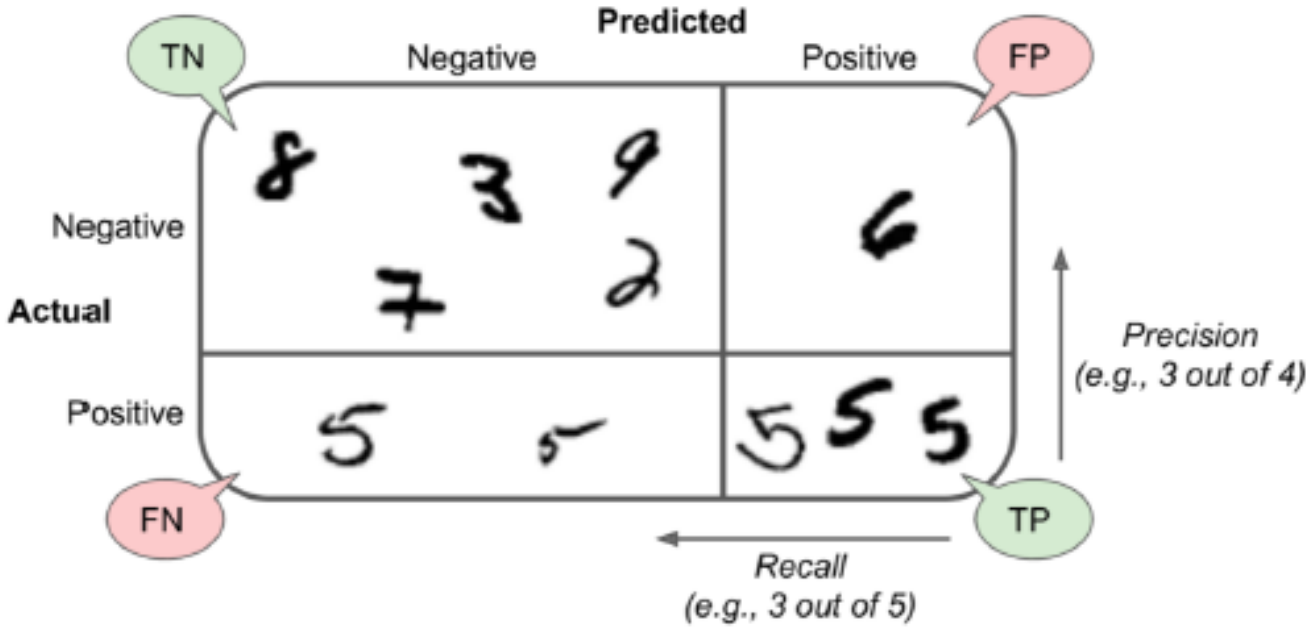
```
never_5_clf = Never5Classifier()
cross_val_score(never_5_clf, X_train, y_train_5, cv=3, scoring="accuracy")
```

```
array([0.91125, 0.90855, 0.90915])
```

  - Accuracy is above 90% without learning

Confusion matrix

- A much better way to evaluate the performance of a classifier is to look at the confusion matrix



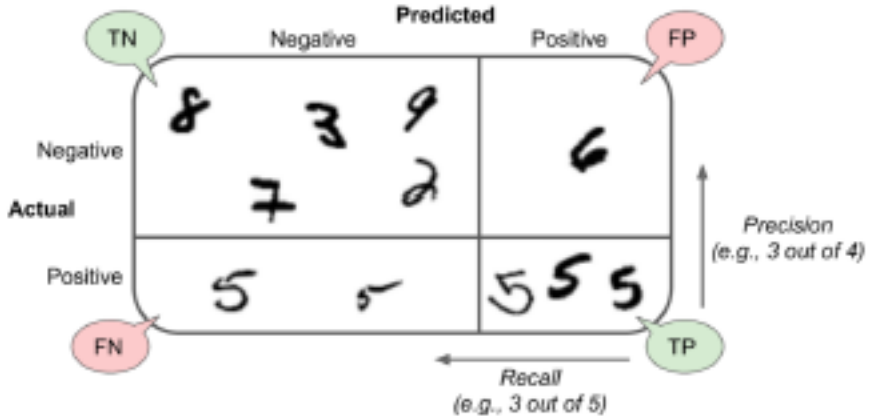
- How to create the confusion matrix?

```
from sklearn.model_selection import cross_val_predict
y_train_pred = cross_val_predict(sgd_clf, X_train, y_train_5, cv=3)
from sklearn.metrics import confusion_matrix
confusion_matrix(y_train_5, y_train_pred)
```

```
array([[53892,   687],
       [ 1891,  3530]])
```

Precision and recall

- Confusion matrix gives you a lot of information, but sometimes you prefer a more concise metric



- Precision:

$$\text{precision} = \frac{TP}{TP + FP}$$

- Recall or sensitivity:

$$\text{recall} = \frac{TP}{TP + FN}$$

Computing precision and recall

- We use off the shelf functions to compute these scores

```
from sklearn.metrics import precision_score, recall_score

precision_score(y_train_5, y_train_pred)
```

```
0.8370879772350012
```

```
recall_score(y_train_5, y_train_pred)
```

```
0.6511713705958311
```

- Why do we see a significant difference?

- We can combine these two scores into a single metric

$$F_1 = \frac{2}{\frac{1}{\text{precision}} + \frac{1}{\text{recall}}} = 2 \times \frac{\text{precision} \times \text{recall}}{\text{precision} + \text{recall}}$$

```
from sklearn.metrics import f1_score
f1_score(y_train_5, y_train_pred)
```

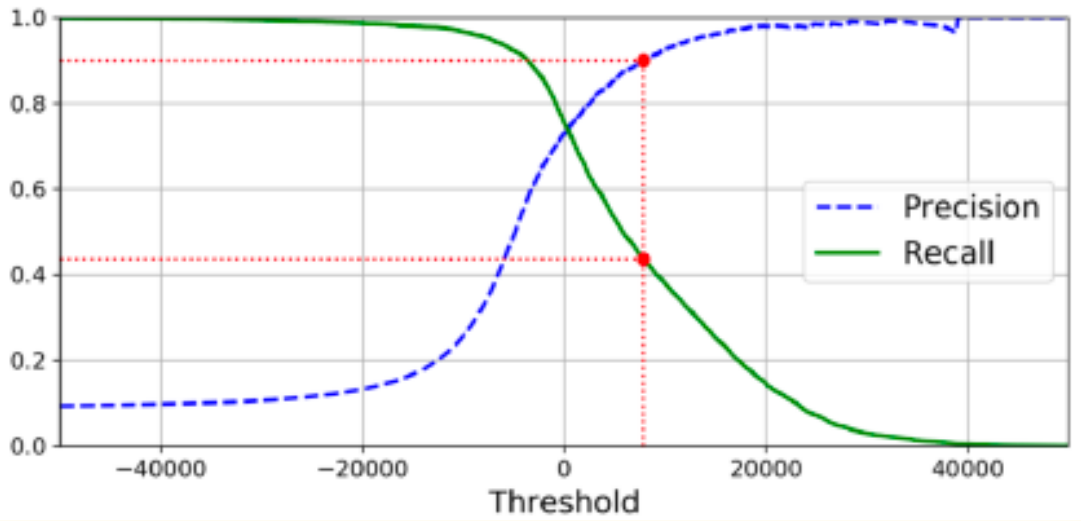
```
0.7325171197343846
```

Precision/recall trade-off

- F_1 score favors classifiers that have similar precision and recall
  - Sometimes we care about precision, and sometime recall
- We can use decision scores instead of calling the classifier's predict() method

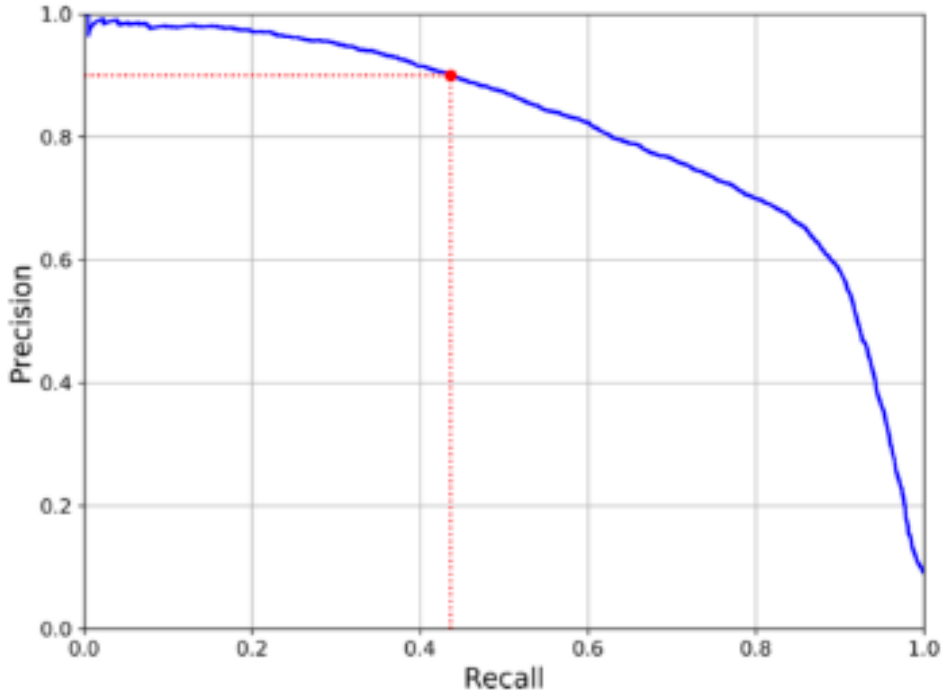
```
y_scores = cross_val_predict(sgd_clf, X_train, y_train_5, cv=3,
                             method="decision_function")
```

```
from sklearn.metrics import precision_recall_curve
precisions, recalls, thresholds = precision_recall_curve(y_train_5, y_scores)
```



Precision vs recall

- We can plot precision directly against recall

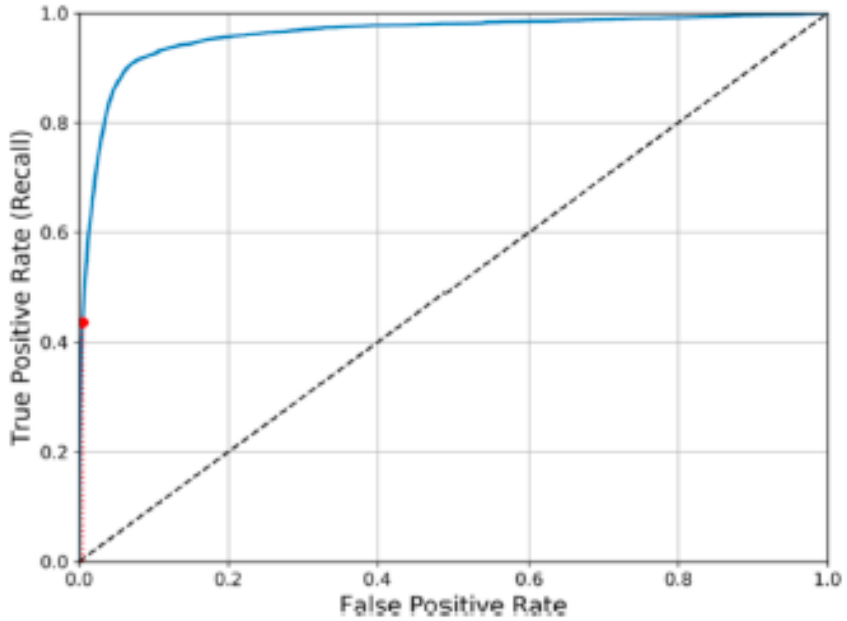


ROC curve

- Receiver Operating Characteristic (ROC) curve plots the true positive rate (TPR) as a function false positive rate (FPR)

$$TPR = \frac{TP}{TP + FN} \text{ (same as recall)}$$

$$FPR = \frac{FP}{FP + TN} = 1 - \frac{TN}{TN + FP} = 1 - TNR$$



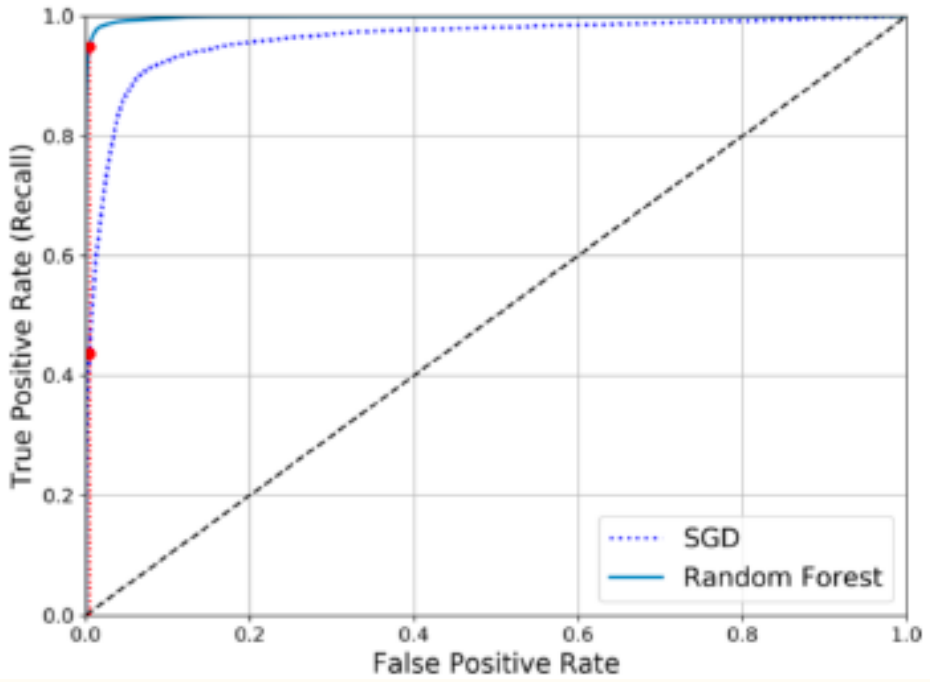
Comparing classifiers

```
from sklearn.ensemble import RandomForestClassifier
forest_clf = RandomForestClassifier(n_estimators=100, random_state=42)
y_probas_forest = cross_val_predict(forest_clf, X_train, y_train_5, cv=3,
                                    method="predict_proba")
```

```
from sklearn.metrics import roc_curve
y_scores_forest = y_probas_forest[:, 1] # score = proba of positive class
fpr_forest, tpr_forest, thresholds_forest = roc_curve(y_train_5,y_scores_forest)
```

```
print(fpr_forest.shape,tpr_forest.shape)
```

```
(101,) (101,)
```



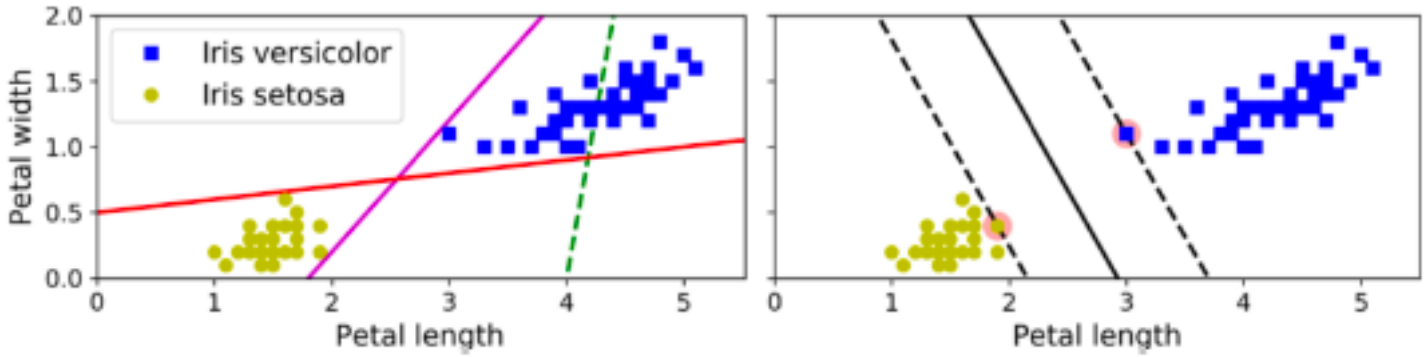
Multiclass classification

- Two main techniques:
  - One-versus-the-rest (OVR): train 10 binary classifiers, one for each digit (0-detector,…,10-detector) and select the class whose classifier outputs the highest score
  - One-versus-one (OVO): train a binary classifier for every pair of digits
    - For example, one classifier to distinguish 0s and 1s, etc.
    - We need to train $\frac{K(K-1)}{2}$ classifiers
    - However, each classifier needs to be trained on part of the training data

Support Vector Machines (SVM)

Introduction

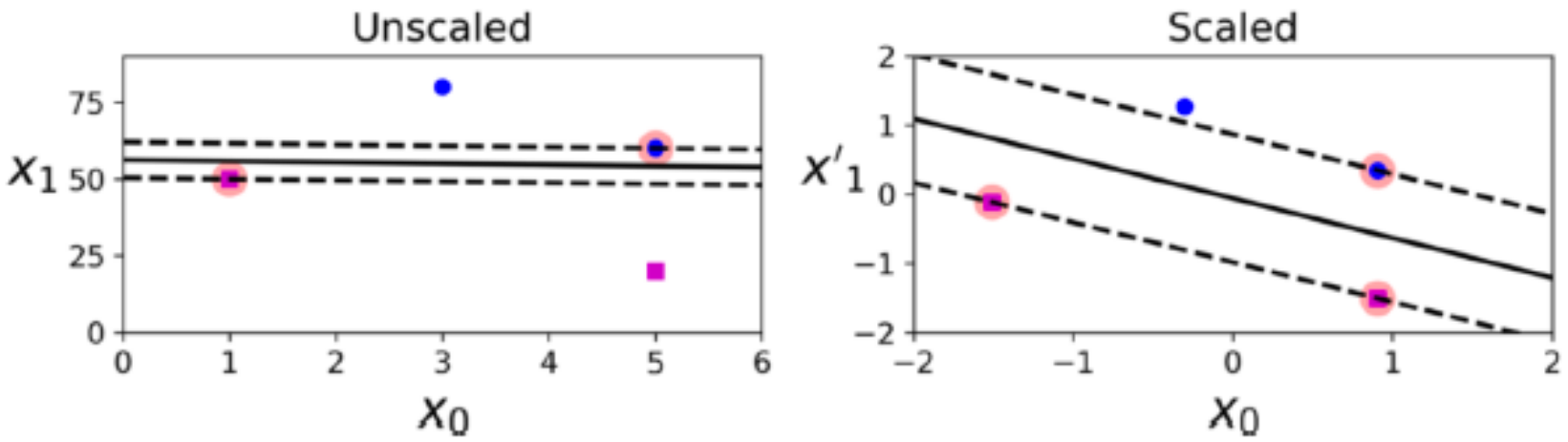
- A Support Vector Machine (SVM) is a powerful model to perform linear or nonlinear classification and regression
- SVMs are suitable for analyzing complex small- to medium-sized datasets
- We start this section by discussing linear classification using SVMs
  - Idea: large margin classification



  - Stay far away from the closest instances as possible
  - We should find "support vectors" (circled in the right figure)

Challenges of using SVMs

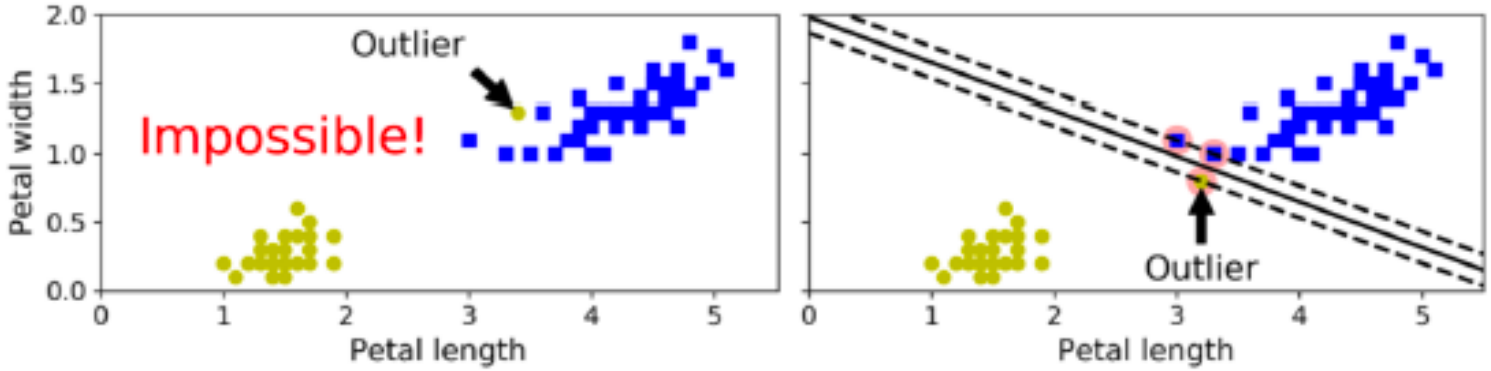
- SVMs are very sensitive to the feature scales



- 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

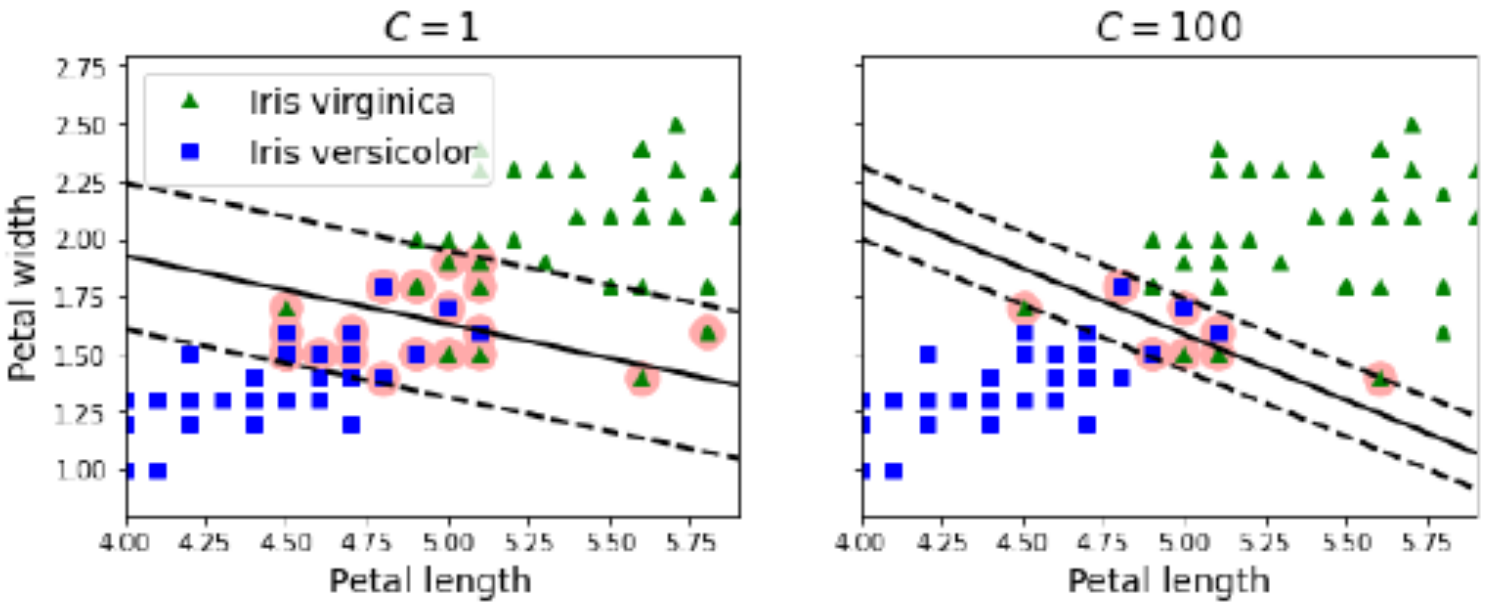


- How to solve this problem? Soft margin classification
  - We use a more flexible model to find a good balance between a large margin and limiting margin violations

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



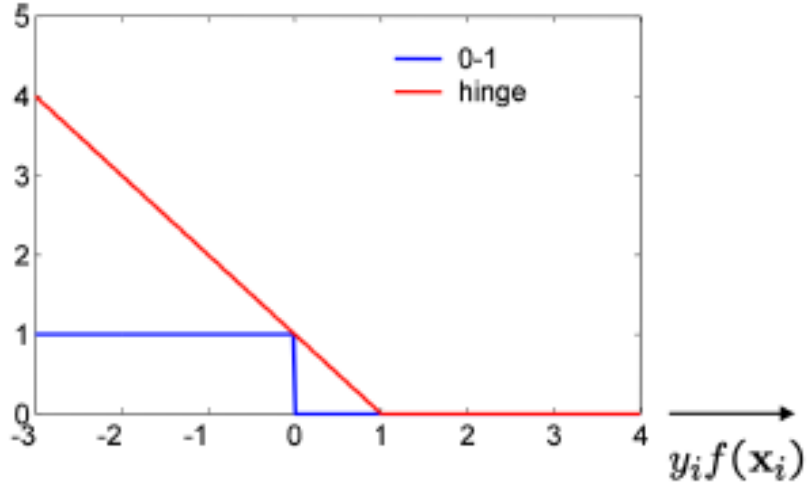
- The strength of the regularization is inversely proportional to C
- If your SVM model is overfitting, you can try regularizing it by decreasing C

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



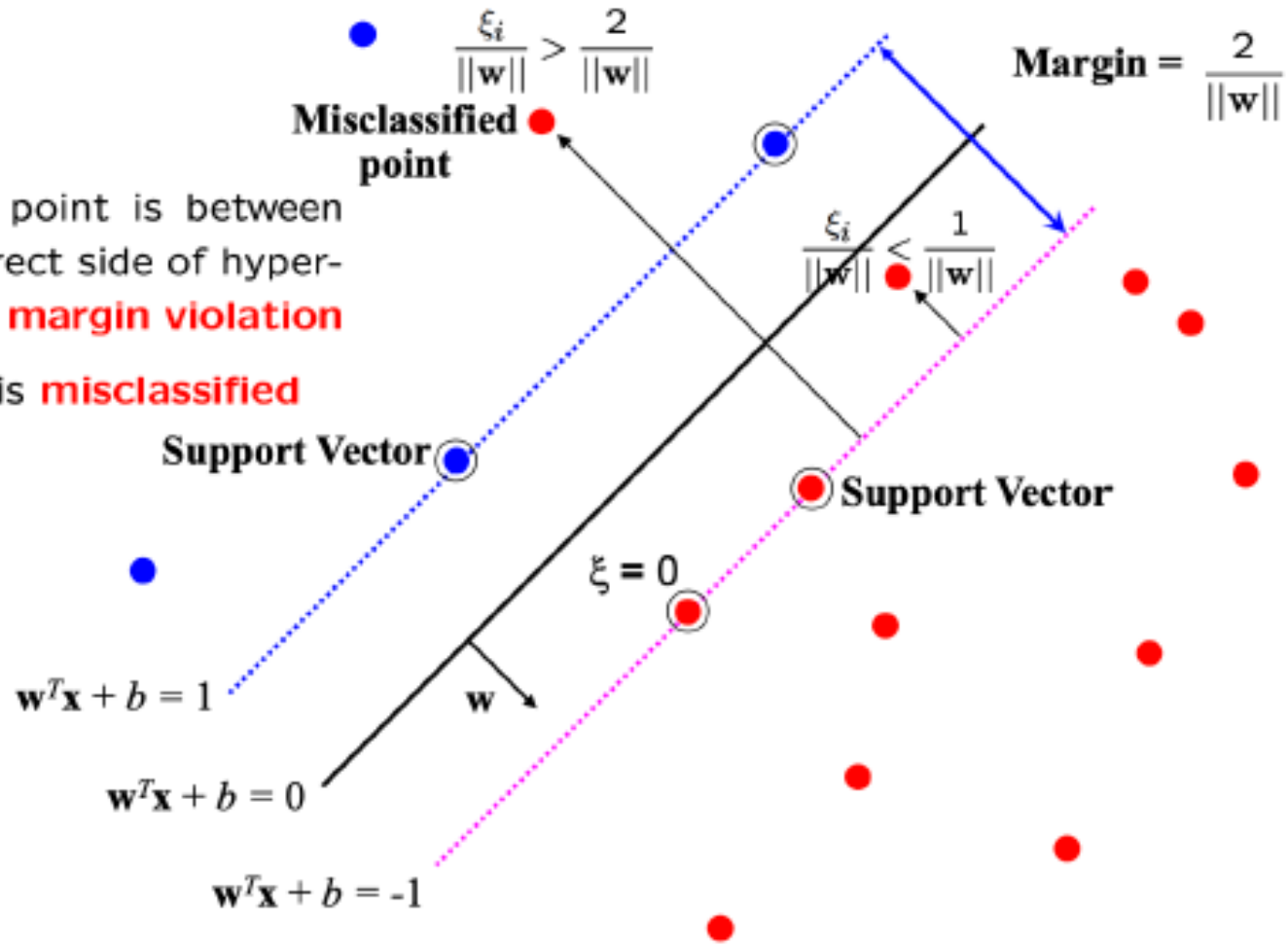
- SVM uses hinge loss

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

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

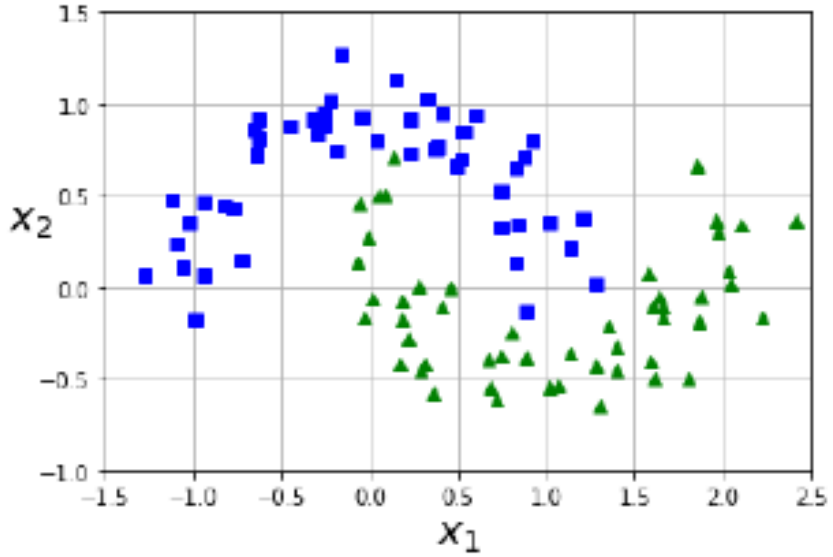
Equivalent!!!

- 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



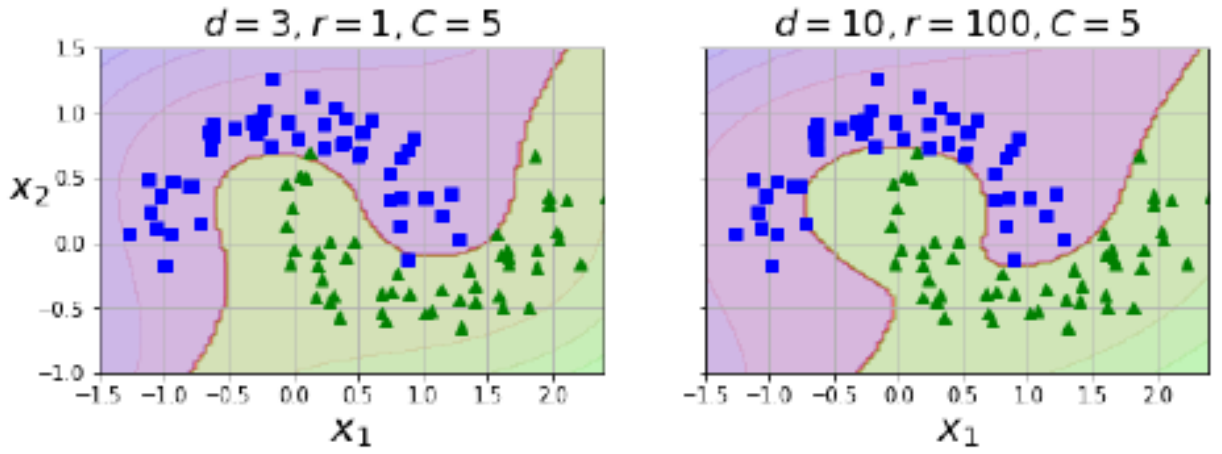
- Instead of using a linear kernel function $k(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}) = \langle \mathbf{x}^{(i)}, \mathbf{x}^{(j)} \rangle$, we use a polynomial kernel function

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

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)
```



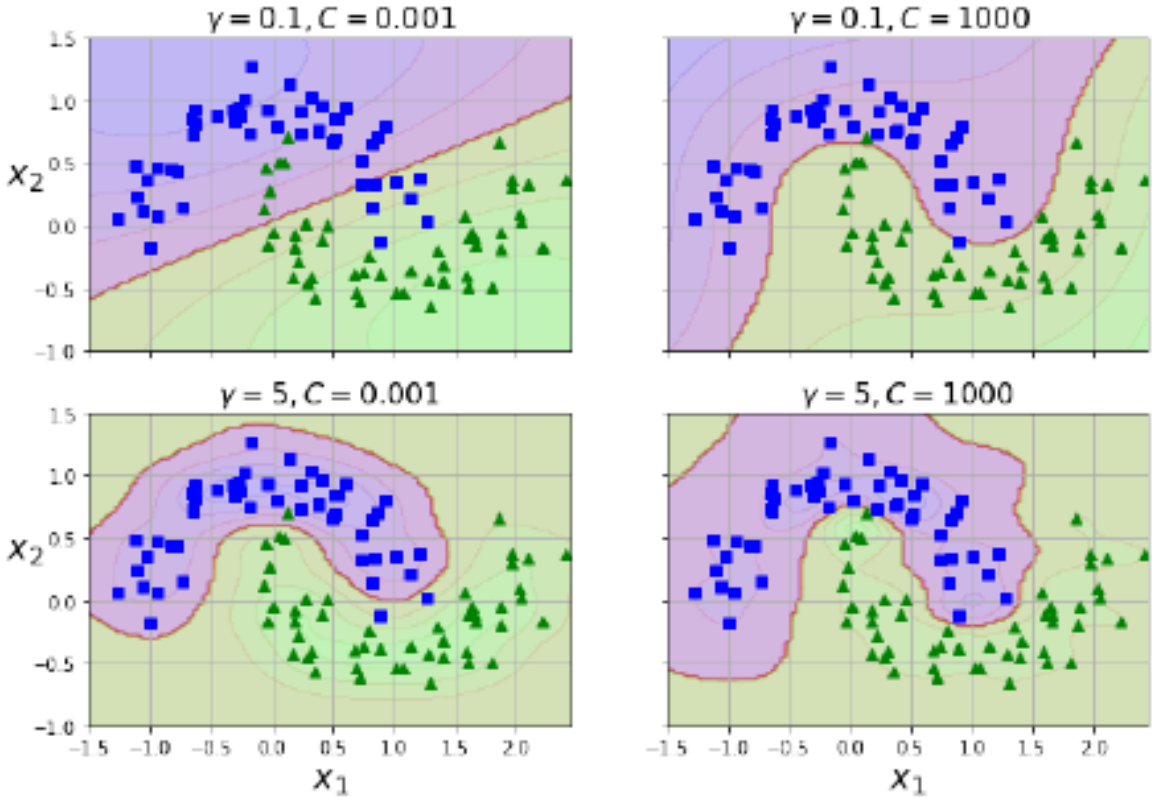
Gaussian kernel

- 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



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

Decision Trees, Random Forests & Ensemble Learning

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))
```

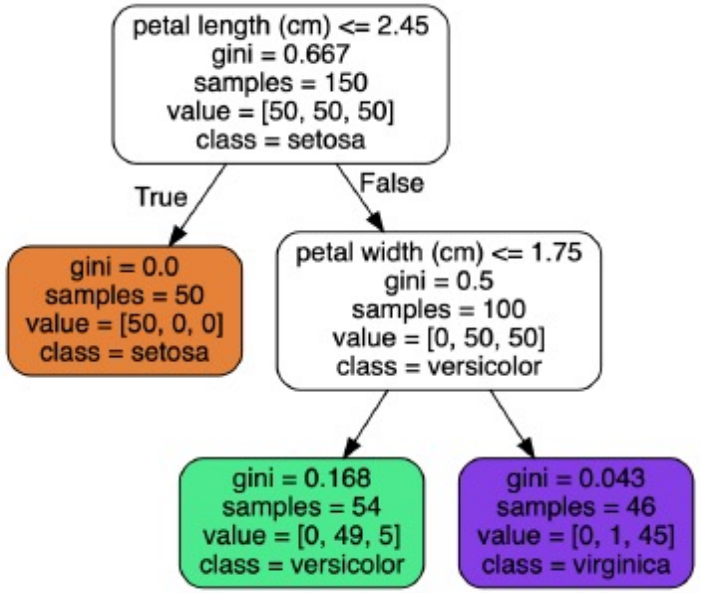
```
(150, 2) (150,)
[0 1 2]
```

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)
```

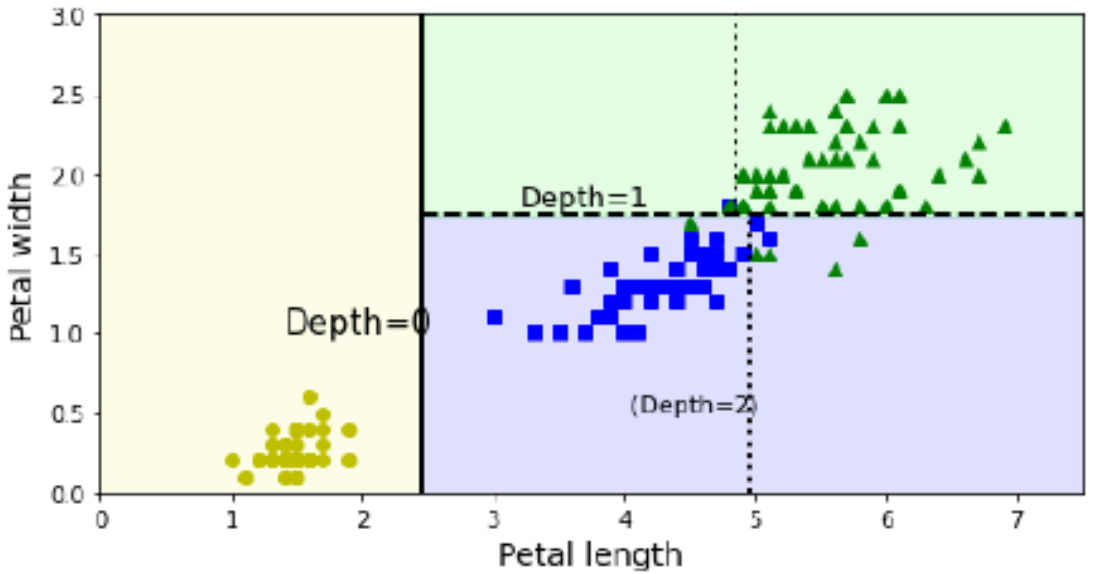
```
DecisionTreeClassifier(ccp_alpha=0.0, class_weight=None, criterion='gini'
                       max_depth=2, max_features=None, max_leaf_nodes=Non
                       min_impurity_decrease=0.0, min_impurity_split=None
                       min_samples_leaf=1, min_samples_split=2,
          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



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

- 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]])
```

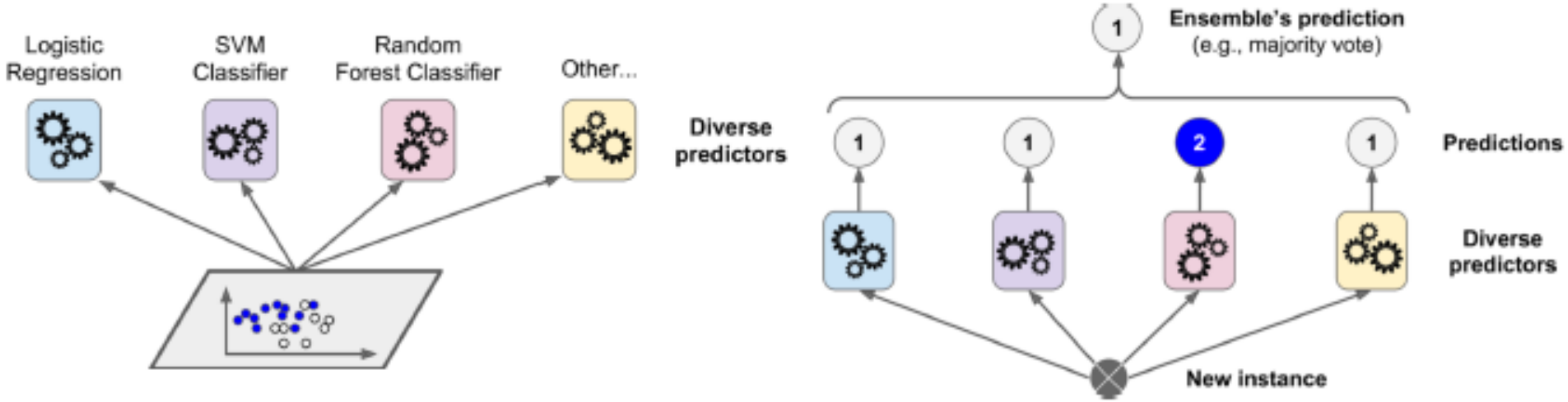
```
array([[0.        , 0.90740741, 0.09259259]])
```

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

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

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

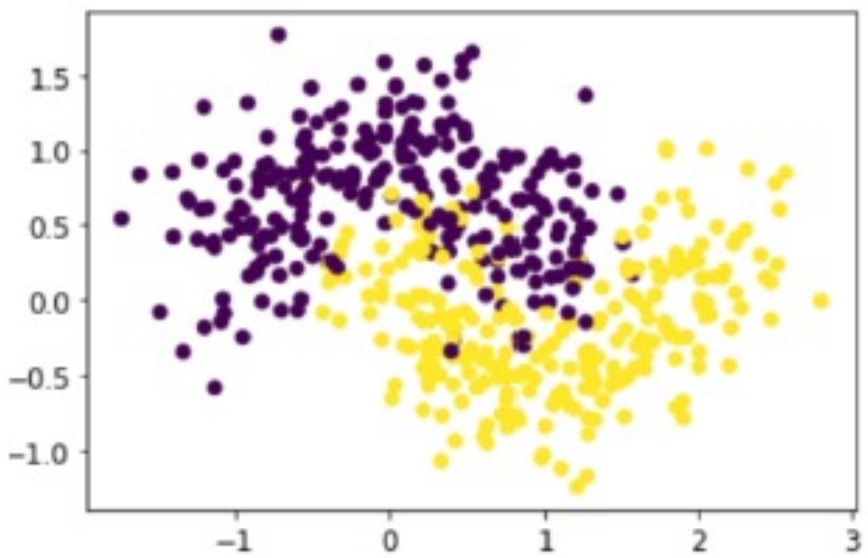


  - Train a group of Decision Tree classifiers, each on a different random subset of training data (ensemble of Decision Trees is called Random Forest)

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))
```

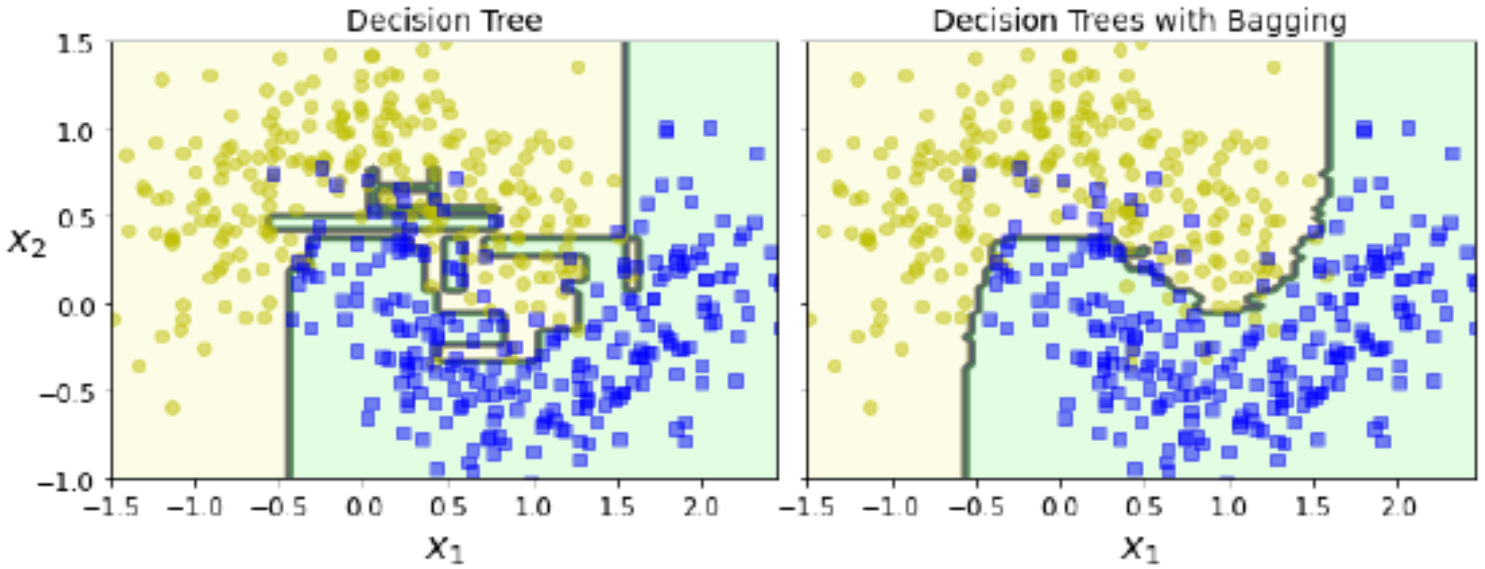
```
0.904
```

- 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"*

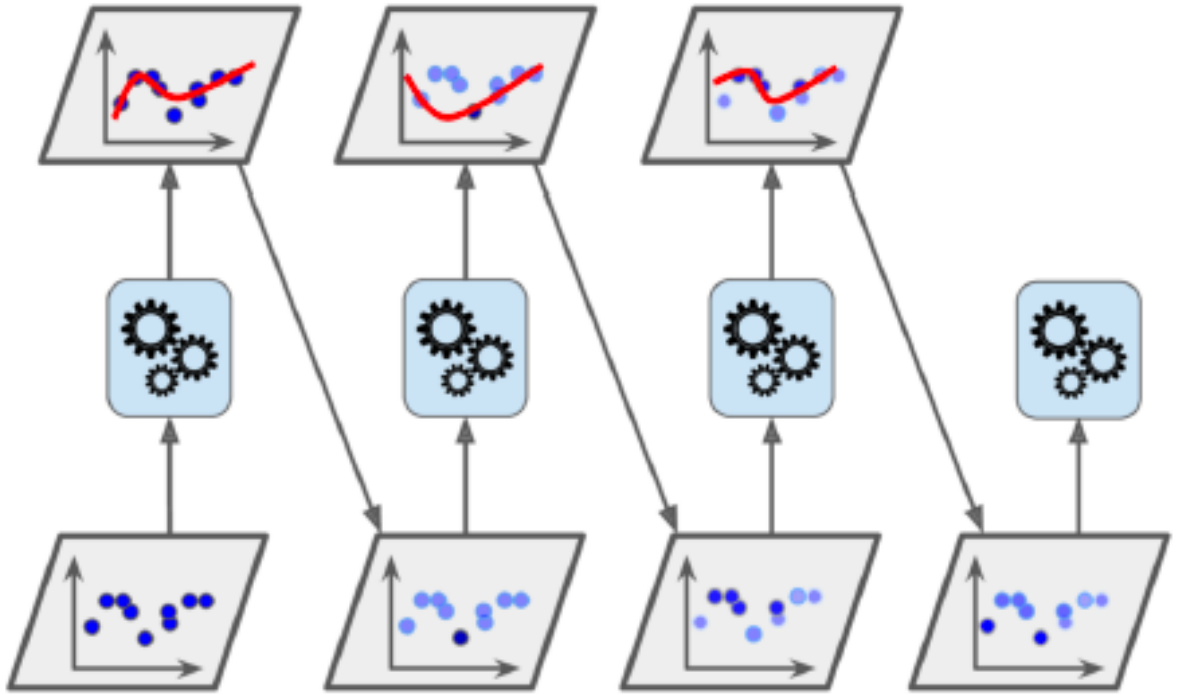
The function to measure the quality of a split. Supported criteria are "gini" for the Gini impurity and "entropy" for the information gain. Note: this parameter is tree-specific.

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



Exercises：

- **Scikit-learn**: a library for machine learning
- **TensorFlow**, **Keras**, **PyTorch**

Homework Guidelines：

- **One PDF or Word file** describing your results
- **Code**
- If you want to compress your documents, please **use .zip**, NOT .rar

Homework - 3rd：

Will be announced in the class QQ group