A Tour of Machine Learning Classifier Using Scikit-learn

Hsi-Pin Ma 馬席彬

http://lms.nthu.edu.tw/course/40724
Department of Electrical Engineering
National Tsing Hua University

Outline

- First Step with Scikit-learn: Training a Perceptron
- Modeling Class Probabilities via Logistic Regression
- Maximum Margin Classification with Support Vector Machines
- Solving Nonlinear Problems Using a Kernel SVM
- Decision Tree Learning
- Random Forest
- K-nearest Neighbors - a Lazy Learning Algorithm

Choosing a Classification Algorithm

- No classifier works best across all scenarios
  - No free lunch theorem
  - Always need to consider the specifics of the problem
- The performance (***computational performance*** and ***predictive power***) of a classifier depends heavily on the underlying data that is available for learning
- Steps involved in training a ML algorithm
  - Select features and collect training samples
  - Choose performance metrics
  - Choose a classifier and optimization algorithm
  - Evaluate performance of the model
  - Tune the algorithm

First Steps with Scikit-learn: Training a Perceptron

Preprocessing (1/3)

- Load the Iris database from scikit-learn
  - Scikit-learn's **datasets** module includes utilities to load datasets
- x_2: petal length, x_3: petal width
- Label y_i: 0=Iris-Setosa, 1= Iris-Veriscolor, 2=Iris-Virginica

```
from sklearn import datasets
import numpy as np

iris = datasets.load_iris()
X = iris.data[:, [2, 3]]
y = iris.target

print('Class labels:', np.unique(y))
```

```
Class labels: [0 1 2]
```

Preprocessing (2/3)

```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.3, random_state=1, stratify=y)
```

- Split data into 70% training and 30% testing
- Shuffle the data before splitting
- Support for stratification: the training and testing subsets have the same proportions of class labels as the whole Iris dataset
- Check the stratification has been done

```
print('Labels counts in y:', np.bincount(y))
print('Labels counts in y_train:', np.bincount(y_train))
print('Labels counts in y_test:', np.bincount(y_test))
```

```
Labels counts in y: [50 50 50]
Labels counts in y_train: [35 35 35]
Labels counts in y_test: [15 15 15]
```

Preprocessing (3/3)

- Standardization
  - Standardize the testing dataset by using the same scaling parameters (μ, σ) as for the training dataset

```
from sklearn.preprocessing import StandardScaler

sc = StandardScaler()
sc.fit(X_train)
X_train_std = sc.transform(X_train)
X_test_std = sc.transform(X_test)
```

Training a Perceptron

- Import the **Perceptron** class from Scikit-learn **linear_model** module
- Instantiate the **Perceptron** class to create an object named **ppn**
- Train the **ppn** object by the standardized training data X_train_std and y_train

```
from sklearn.linear_model import Perceptron

ppn = Perceptron(n_iter=40, eta0=0.1, random_state=0)
ppn.fit(X_train_std, y_train)
```

```
Perceptron(alpha=0.0001, class_weight=None, eta0=0.1, fit_intercept=True,
      n_iter=40, n_jobs=1, penalty=None, random_state=0, shuffle=True,
      verbose=0, warm_start=False)
```

Multiclass Classification via One-versus-Rest (OvR)/One-versus-All (OvA)

- k: number of class labels
- First, learn k binary classifiers h_l, $1 \leq l \leq k$, each seeking to discriminate the class l from all the others
- Assume that for a data point $\mathbf{x}$, the binary classification $h_l(\mathbf{x}) = \text{sgn}(f_l(\mathbf{x}))$ is based on a real-valued scoring function f_l on data points $\mathbf{x}$. ($f_l(\mathbf{x}) = \mathbf{w}_l \cdot \mathbf{x}$)
- The multiclass classifier h defined by OvR is

$$h(\mathbf{x}) = \arg \max_{l \in \{1,2,\ldots,k\}} f_l(\mathbf{x}) = \arg \max_{l \in \{1,2,\ldots,k\}} \mathbf{w}_l \cdot \mathbf{x}.$$

- Most algorithms in scikit-learn support multiclass classification, including **Perceptron**

Performance Measurement (1/2)

- Make prediction to the standardized testing set
- Show performance measures such as the number of misclassification or accuracy

```
In [7]: y_pred = ppn.predict(X_test_std)
        print('Misclassified samples: %d' % (y_test != y_pred).sum())

Misclassified samples: 3


In [8]: from sklearn.metrics import accuracy_score

        print('Accuracy: %.2f' % accuracy_score(y_test, y_pred))

Accuracy: 0.93
```

Performance Measurement (2/2)

- Each classifier in scikit-learn has a **score** method, which computes a classifier's prediction accuracy by combining the **predict** method with **accuracy_score** function

```
In [9]: print('Accuracy: %.2f' % ppn.score(X_test_std, y_test))

Accuracy: 0.93
```

Re-define **plot_decision_region** (1/2)

```
from matplotlib.colors import ListedColormap
import matplotlib.pyplot as plt


def plot_decision_regions(X, y, classifier, test_idx=None, resolution=0.02):

    # setup marker generator and color map
    markers = ('s', 'x', 'o', '^', 'v')
    colors = ('red', 'blue', 'lightgreen', 'gray', 'cyan')
    cmap = ListedColormap(colors[:len(np.unique(y))])

    # plot the decision surface
    x1_min, x1_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    x2_min, x2_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    xx1, xx2 = np.meshgrid(np.arange(x1_min, x1_max, resolution),
                           np.arange(x2_min, x2_max, resolution))
    Z = classifier.predict(np.array([xx1.ravel(), xx2.ravel()]).T)
    Z = Z.reshape(xx1.shape)
    plt.contourf(xx1, xx2, Z, alpha=0.3, cmap=cmap)
    plt.xlim(xx1.min(), xx1.max())
    plt.ylim(xx2.min(), xx2.max())
```

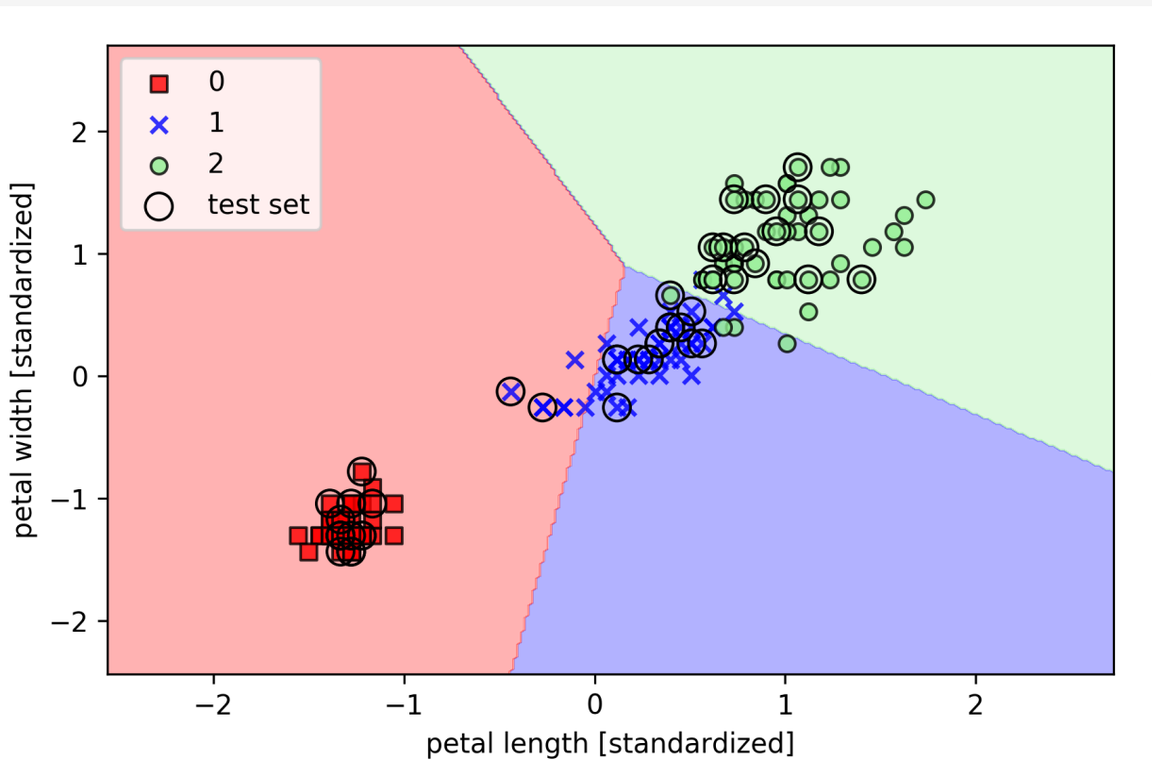
Re-define plot_decision_region (2/2)

```
for idx, cl in enumerate(np.unique(y)):
    plt.scatter(x=X[y == cl, 0],
                y=X[y == cl, 1],
                alpha=0.8,
                c=colors[idx],
                marker=markers[idx],
                label=cl,
                edgecolor='black')

# highlight test samples
if test_idx:
    # plot all samples
    X_test, y_test = X[test_idx, :], y[test_idx]

    plt.scatter(X_test[:, 0],
                X_test[:, 1],
                c='',
                edgecolor='black',
                alpha=1.0,
                linewidth=1,
                marker='o',
                s=100,
                label='test set')
```

Training a Perceptron Model Using the Standardization Training Data

- Note that the Perceptron algorithm never converges on datasets that are not linearly separable.

```
X_combined_std = np.vstack((X_train_std, X_test_std))
y_combined = np.hstack((y_train, y_test))

plot_decision_regions(X=X_combined_std, y=y_combined,
                      classifier=ppn, test_idx=range(105, 150))
plt.xlabel('petal length [standardized]')
plt.ylabel('petal width [standardized]')
plt.legend(loc='upper left')

plt.tight_layout()
#plt.savefig('images/03_01.png', dpi=300)
plt.show()
```



Modeling Class Probabilities via Logistic Regression

Modeling Class Possibilities

- If classes are not linearly separable
  - The algorithms never converge
  - Weights never stop updating as long as there is at least one misclassified sample in each epoch
- **Logistic regression** is a better option
  - despite the name, it is a classification model, not a regression model.
  - Most widely used for classification in industry
  - Can be extended to multiclass classification

Modeling *logit* Function (1/2)

- Log-odds as a linear function of the features
- Binary classification with class labels denoted as 0 and 1 (instead of -1 and 1)
- Odds ratio $\frac{p}{1-p}$
  - p stands for the probability of the positive event (the event we want to predict)
  - $p = P(y = 1|\mathbf{x})$ the conditional probability that the class label y is 1 given a data point $\mathbf{x}$
- Define ***logit*** function $\text{logit}(p) = \log \frac{p}{1-p}$

Modeling *logit* Function (2/2)

- We model ***logit*** function as a linear combination of features

$$\text{logit}(p) = \text{logit}(P(y=1|\mathbf{x})) = \mathbf{w}\cdot\mathbf{x} = w_0x_0 + w_1x_1 + \ldots + w_mx_m$$

  - Where $p = P(y=1|\mathbf{x})$ is the conditional probability that the class label y is 1 given a data point $\mathbf{x}$

- This is equivalent to express p as

$$p(y=1|\mathbf{x}) = \frac{1}{1+e^{-\mathbf{w}^T\mathbf{x}}}$$

Logistic Sigmoid Function (1/2)

- The logistic sigmoid function $\phi(z)$ is the inverse function of ***logit*** function from $\mathbb{R}$ to [0,1]

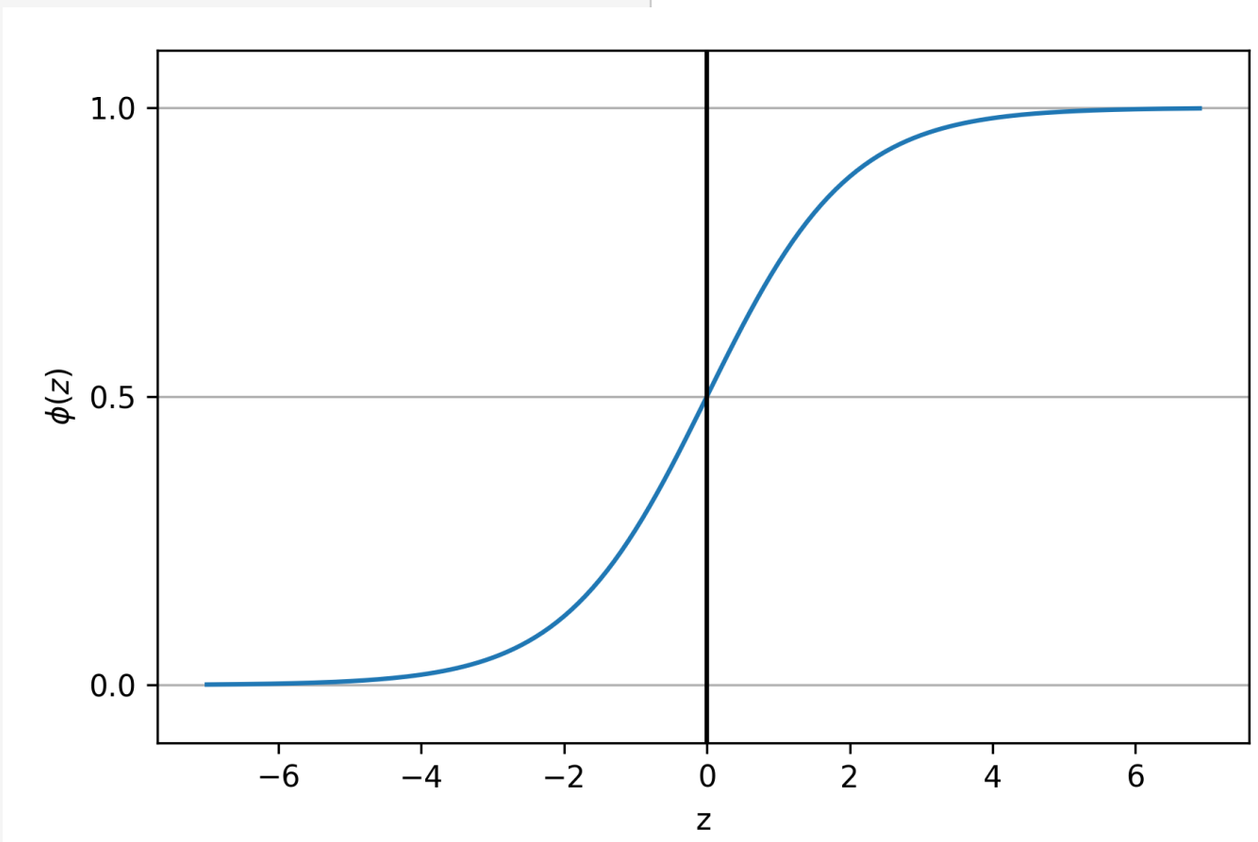
$$\phi(z) = \frac{1}{1+e^{-z}} \qquad z = \mathbf{w}^T\mathbf{x} = w_0x_0 + w_1x_1 + ... + w_mx_m$$

- Convert the log-odds to the conditional probability $p = P(y=1|\mathbf{x})$, which is usually called the likelihood of having class label y=1 given the instance feature vector **x**.

Logistic Sigmoid Function (2/2)

- S-shaped curve

```
import matplotlib.pyplot as plt
import numpy as np


def sigmoid(z):
    return 1.0 / (1.0 + np.exp(-z))

z = np.arange(-7, 7, 0.1)
phi_z = sigmoid(z)

plt.plot(z, phi_z)
plt.axvline(0.0, color='k')
plt.ylim(-0.1, 1.1)
plt.xlabel('z')
plt.ylabel('$\phi (z)$')

# y axis ticks and gridline
plt.yticks([0.0, 0.5, 1.0])
ax = plt.gca()
ax.yaxis.grid(True)

plt.tight_layout()
# plt.savefig('./figures/sigmoid.png', dpi=300)
plt.show()
```



Probability Distributions over Classes

- Output of sigmoid often interpreted as probability
  - For example, $p(y = 1|\mathbf{x}; \mathbf{w}) = 0.8$, given its features $\mathbf{x}$ parameterized by weights $\mathbf{w}$
  - Probability can be converted to a binary outcome (quantizer)

$$\hat{y} = \begin{cases} 1 & \text{if } \phi(z) \geq 0.5 \\ 0 & \text{otherwise} \end{cases}$$

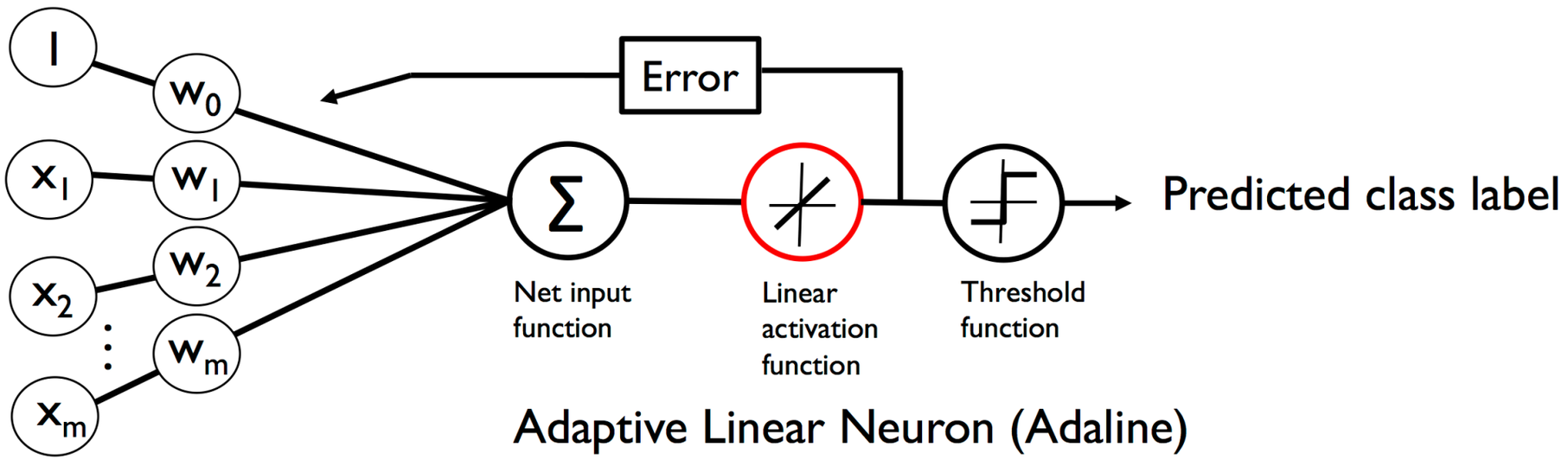
  - The same as Perceptron/Adaline, is equivalent to the following

$$\hat{y} = \begin{cases} 1 & \text{if } \phi(z) \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

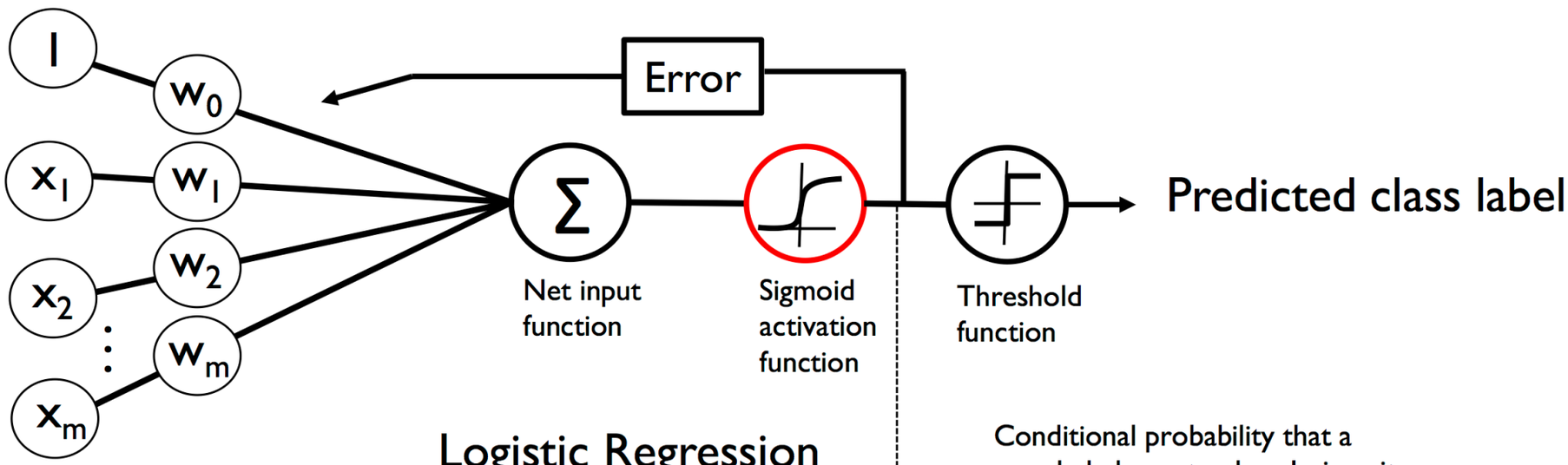
  - For many applications, we want the probability (e.g., weather, disease)

Logistic Regression vs. Adaline

- Different learning rule to update weight vector **w**



Adaptive Linear Neuron (Adaline)



Logistic Regression

Cost Function for Logistic Regression (1/3)

- To seek a weight vector **w** *maximizing* the likelihood function
  - Assume the data points are selected statistically independent

$$L(\boldsymbol{w}) = P(\boldsymbol{y} \mid \boldsymbol{x}; \boldsymbol{w}) = \prod_{i=1}^{n} P\left(y^{(i)} \mid \boldsymbol{x}^{(i)}; \boldsymbol{w}\right) = \prod_{i=1}^{n} \left(\phi\left(z^{(i)}\right)\right)^{y^{(i)}} \left(1 - \phi\left(z^{(i)}\right)\right)^{1-y^{(i)}}$$

- In practice, it is easier to maximize the log of the equation (called log-likelihood function)

$$l(\boldsymbol{w}) = \log L(\boldsymbol{w}) = \sum_{i=1}^{n} \left[y^{(i)} \log\left(\phi\left(z^{(i)}\right)\right) + \left(1 - y^{(i)}\right) \log\left(1 - \phi\left(z^{(i)}\right)\right) \right]$$

Cost Function for Logistic Regression (2/3)

- Rewrite the log-likelihood as a cost function *J*

$$J(\boldsymbol{w}) = \sum_{i=1}^{n}\left[-y^{(i)}\log\left(\phi\left(z^{(i)}\right)\right) - \left(1-y^{(i)}\right)\log\left(1-\phi\left(z^{(i)}\right)\right)\right]$$

  - J(**w**) is a differentiable convex function of **w**
  - Can be minimized by gradient descent

- For one single training instance

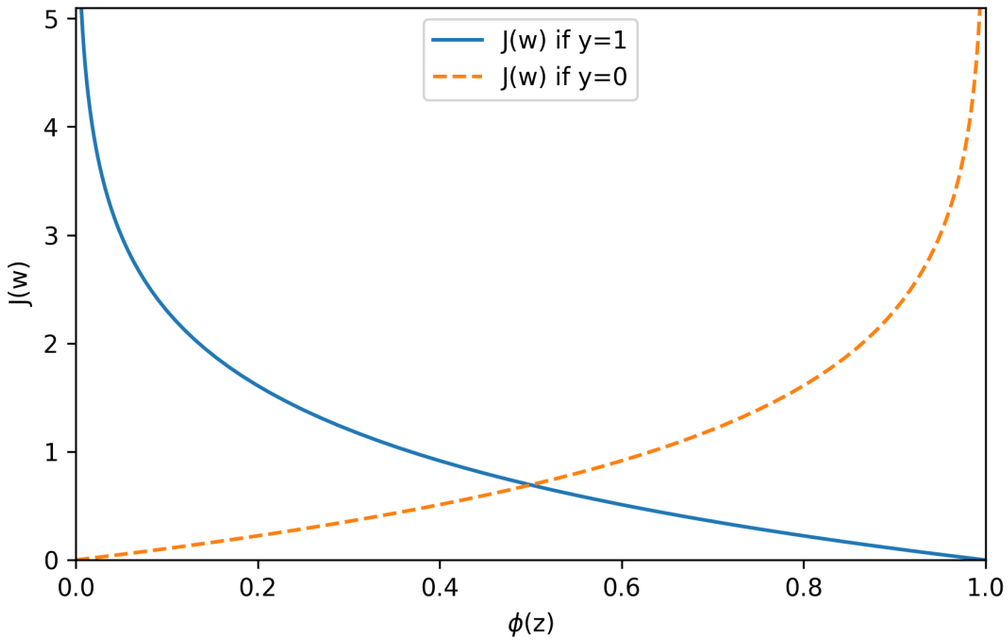
$$J\left(\phi(z), y; \boldsymbol{w}\right) = -y\log\left(\phi(z)\right) - (1-y)\log\left(1-\phi(z)\right)$$

  - We can get

$$J\left(\phi(z), y; \boldsymbol{w}\right) = \begin{cases} -\log\left(\phi(z)\right) & if\ y=1 \\ -\log\left(1-\phi(z)\right) & if\ y=0 \end{cases}$$

Cost Function for Logistic Regression (3/3)

```
def cost_1(z):
    return - np.log(sigmoid(z))


def cost_0(z):
    return - np.log(1 - sigmoid(z))

z = np.arange(-10, 10, 0.1)
phi_z = sigmoid(z)

c1 = [cost_1(x) for x in z]
plt.plot(phi_z, c1, label='J(w) if y=1')

c0 = [cost_0(x) for x in z]
plt.plot(phi_z, c0, linestyle='--', label='J(w) if y=0')

plt.ylim(0.0, 5.1)
plt.xlim([0, 1])
plt.xlabel('$\phi$(z)')
plt.ylabel('J(w)')
plt.legend(loc='best')
plt.tight_layout()
# plt.savefig('./figures/log_cost.png', dpi=300)
plt.show()
```



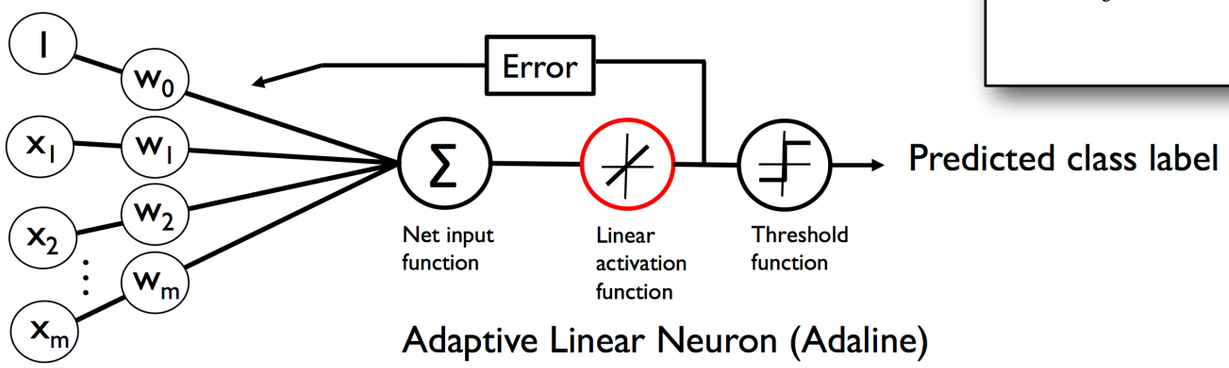
Logistic Regression Learning Algorithm
A Batch Gradient Descent Technique

- The weight update in logistic regression via gradient descent is equal to the update in Adaline
- The cost function J(w) is minimized by

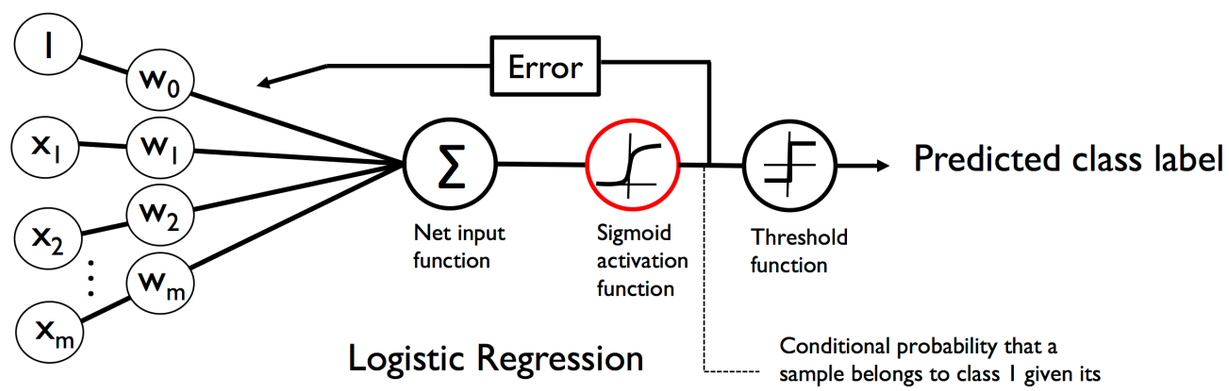
$$\nabla_{\mathbf{w}} J(\mathbf{w}) = -\sum_{i=1}^{N} (y^{(i)} - \phi(\mathbf{w} \cdot \mathbf{x}^{(i)}))\mathbf{x}^{(i)}$$

- Weight update for w_j

$$\Delta w_j = -\eta \frac{\partial J}{\partial w_j} = \eta \sum_i \left(y^{(i)} - \phi\left(z^{(i)} \right) \right) x_j^{(i)}$$



Adaptive Linear Neuron (Adaline)



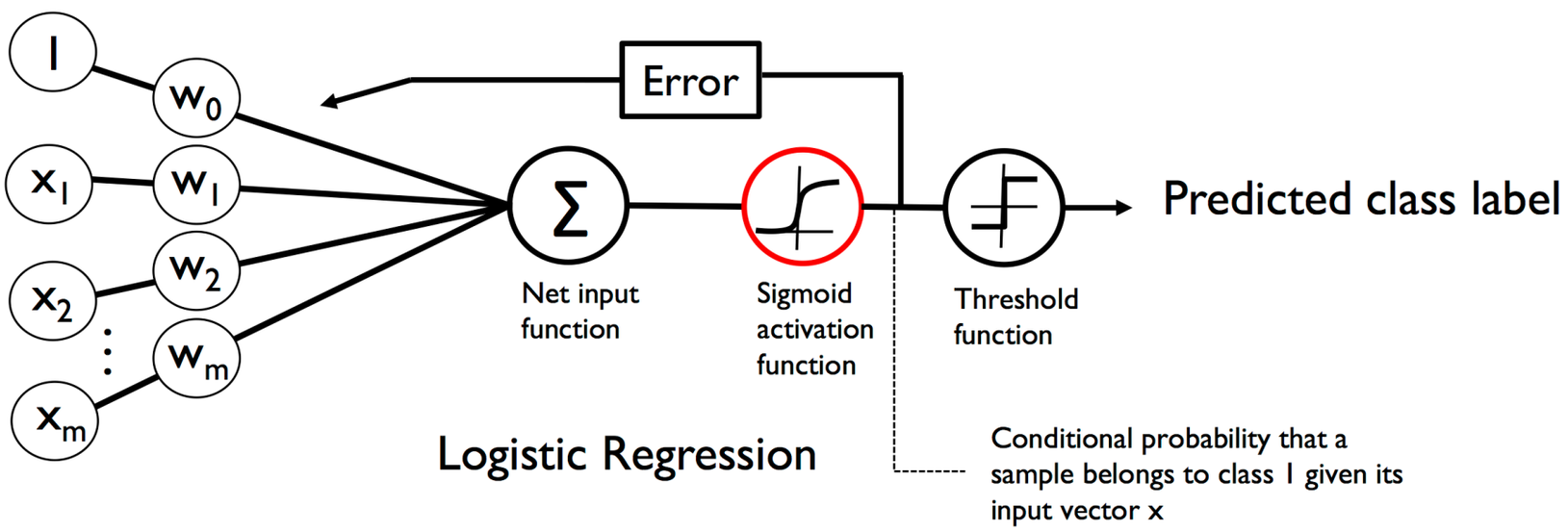
Logistic Regression

Implementation of Logistic Regression Classifier with Python (1/2)

```
class LogisticRegressionGD(object):
    def __init__(self, eta=0.05, n_iter=100, random_state=1):
        self.eta = eta
        self.n_iter = n_iter
        self.random_state = random_state

    def fit(self, X, y):
        rgen = np.random.RandomState(self.random_state)
        self.w_ = rgen.normal(loc=0.0, scale=0.01, size=1 + X.shape[1])
        self.cost_ = []

        for i in range(self.n_iter):
            net_input = self.net_input(X)
            output = self.activation(net_input)
            errors = (y - output)
            self.w_[1:] += self.eta * X.T.dot(errors)
            self.w_[0] += self.eta * errors.sum()

            # note that we compute the logistic `cost` now
            # instead of the sum of squared errors cost
            cost = -y.dot(np.log(output)) - ((1 - y).dot(np.log(1 - output)))
            self.cost_.append(cost)
        return self
```

Implementation of Logistic Regression Classifier with Python (2/2)

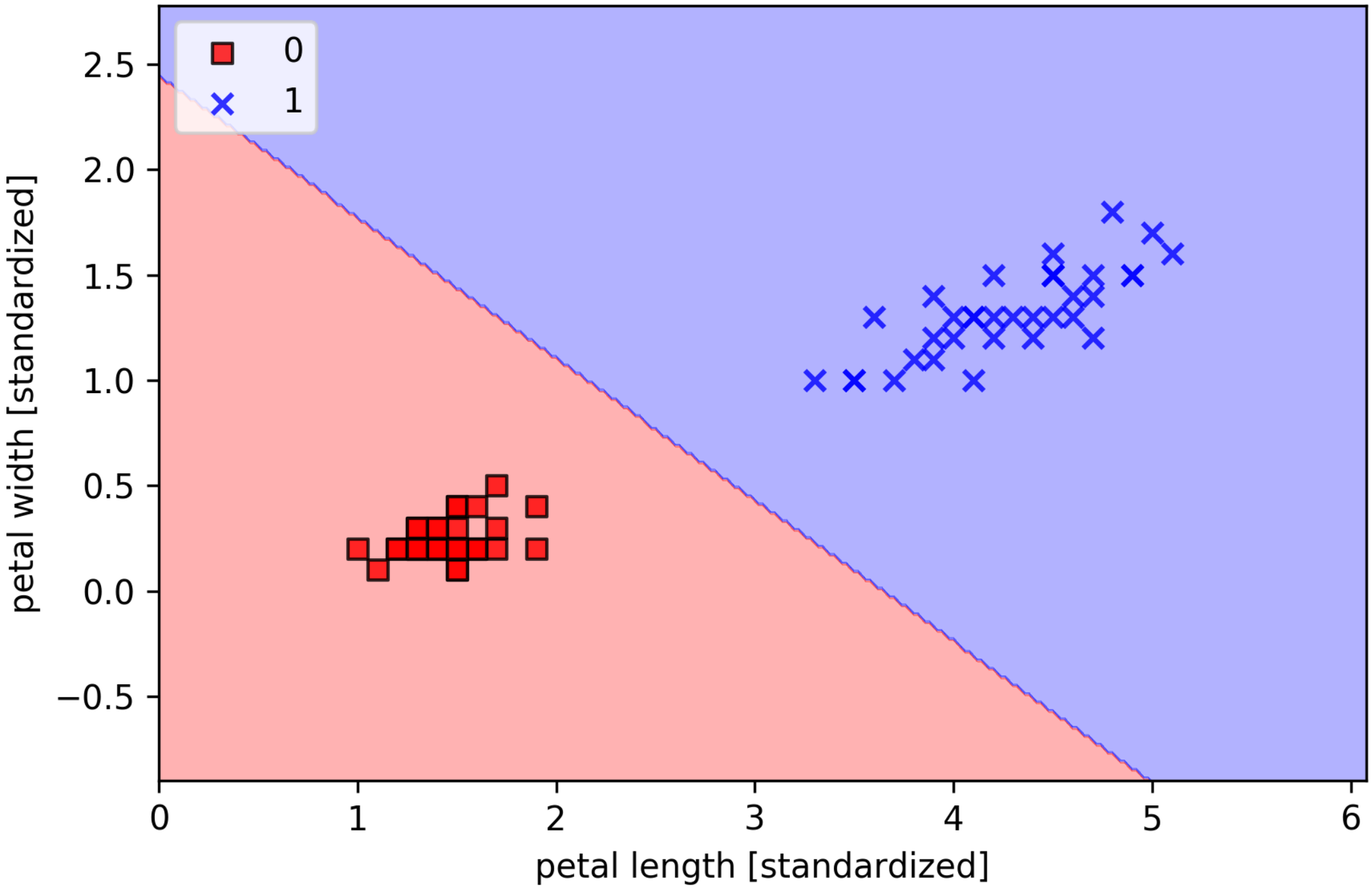
```
    def net_input(self, X):
        """Calculate net input"""
        return np.dot(X, self.w_[1:]) + self.w_[0]

    def activation(self, z):
        """Compute logistic sigmoid activation"""
        return 1. / (1. + np.exp(-np.clip(z, -250, 250)))

    def predict(self, X):
        """Return class label after unit step"""
        return np.where(self.net_input(X) >= 0.0, 1, 0)
        # equivalent to:
        # return np.where(self.activation(self.net_input(X)) >= 0.5, 1, 0)
```



Classification of 100 Iris Flowers by Logistic Regression (1/2)

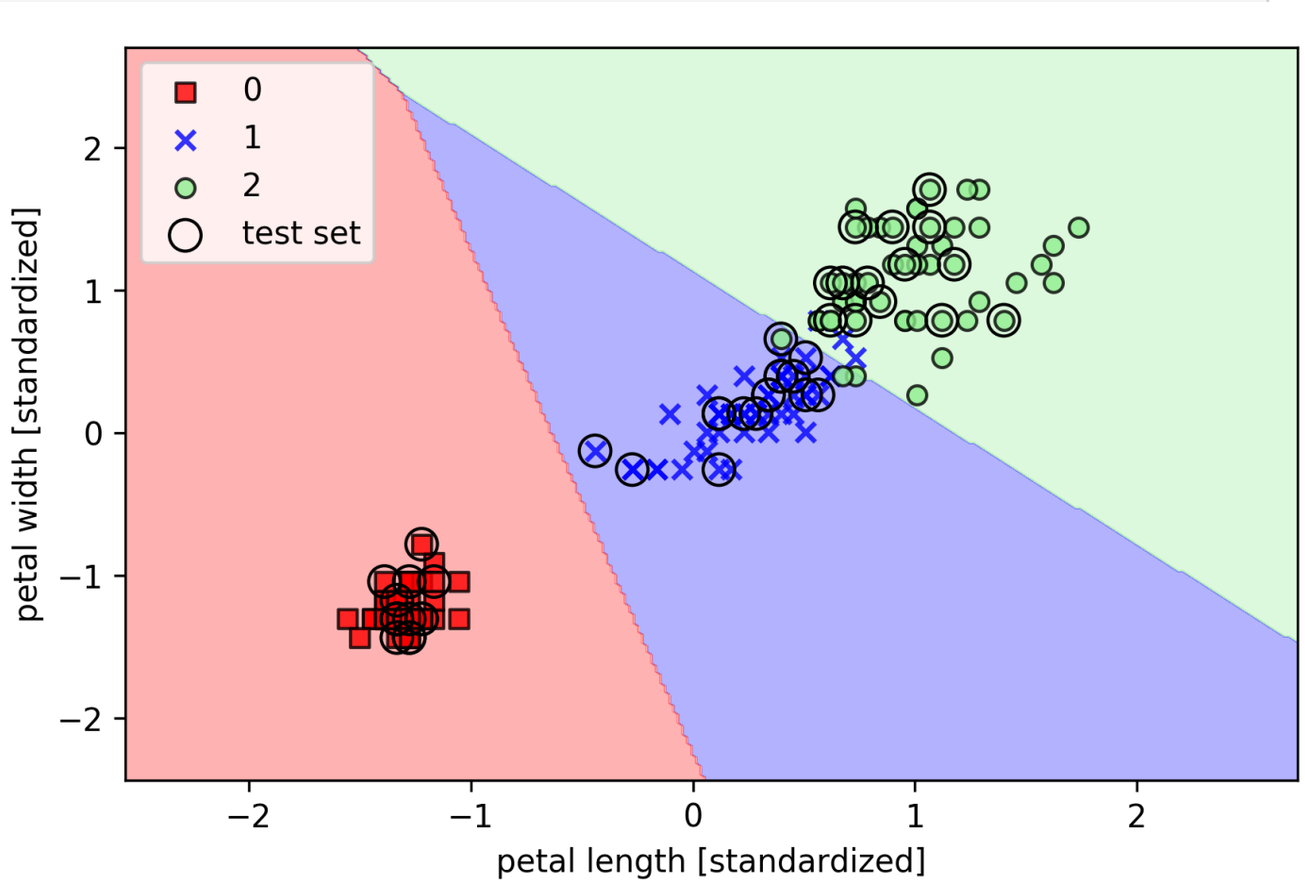
```
X_train_01_subset = X_train[(y_train == 0) | (y_train == 1)]
y_train_01_subset = y_train[(y_train == 0) | (y_train == 1)]

lrgd = LogisticRegressionGD(eta=0.05, n_iter=1000, random_state=1)
lrgd.fit(X_train_01_subset,
         y_train_01_subset)

plot_decision_regions(X=X_train_01_subset,
                      y=y_train_01_subset,
                      classifier=lrgd)

plt.xlabel('petal length [standardized]')
plt.ylabel('petal width [standardized]')
plt.legend(loc='upper left')

plt.tight_layout()
#plt.savefig('images/03_05.png', dpi=300)
plt.show()
```

Classification of 100 Iris Flowers by Logistic Regression (2/2)



Training a Logistic Regression Model with Scikit-learn for Multiclass Classification

```
from sklearn.linear_model import LogisticRegression

lr = LogisticRegression(C=100.0, random_state=1)
lr.fit(X_train_std, y_train)

plot_decision_regions(X_combined_std, y_combined,
                      classifier=lr, test_idx=range(105, 150))
plt.xlabel('petal length [standardized]')
plt.ylabel('petal width [standardized]')
plt.legend(loc='upper left')
plt.tight_layout()
#plt.savefig('images/03_06.png', dpi=300)
plt.show()
```



Training a Logistic Regression Model with Scikit-learn for Multiclass Classification

- Use **predict_proba** method to compute probability
  - Predict the probabilities of the first three samples in the test set

```
lr.predict_proba(X_test_std[:3, :])
```

```
array([[  3.20136878e-08,   1.46953648e-01,   8.53046320e-01],
       [  8.34428069e-01,   1.65571931e-01,   4.57896429e-12],
       [  8.49182775e-01,   1.50817225e-01,   4.65678779e-13]])
```

- Take the row sum in the predicted conditional probability matrix

```
lr.predict_proba(X_test_std[:3, :]).sum(axis=1)
```

```
array([ 1.,  1.,  1.])
```

- Take the **argmax** function of each row

```
lr.predict_proba(X_test_std[:3, :]).argmax(axis=1)
```

```
array([2, 0, 0])
```

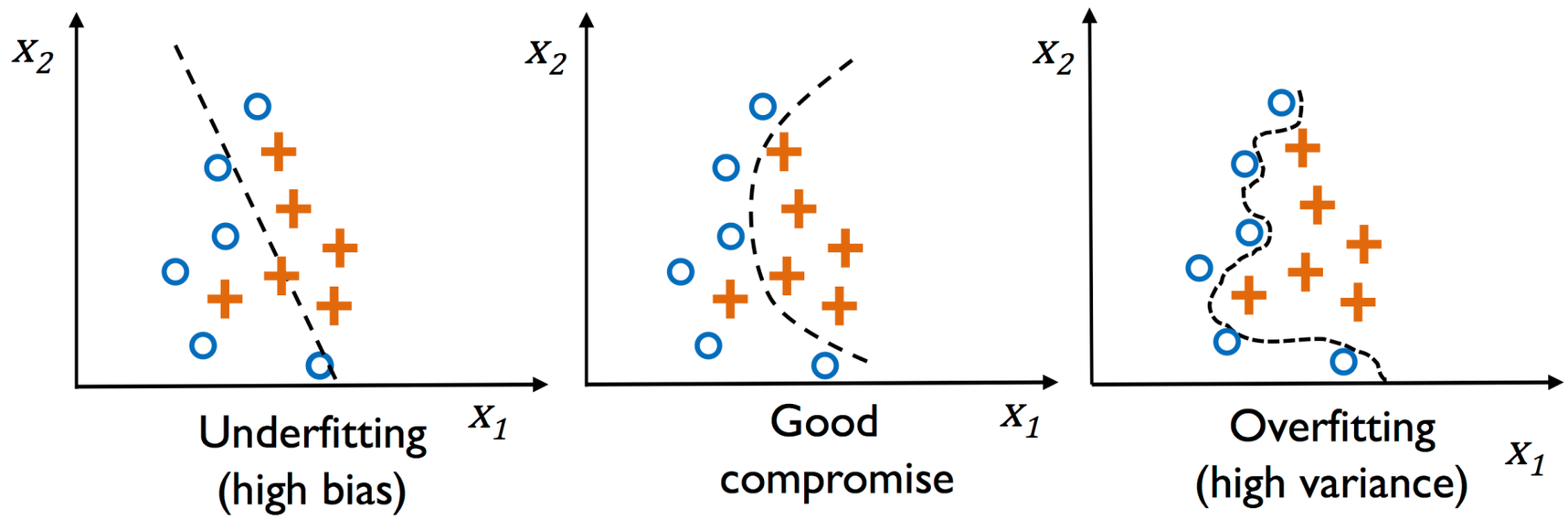
- **predict** function = **predict_proba** and **argmax** functions

```
lr.predict(X_test_std[:3, :])
```

```
array([2, 0, 0])
```

Overfitting and Underfitting

- Overfitting
  - Model performs well on training data but does not generalize well to unseen data (overfitting)
  - The model has high variance
  - Often caused by too complex model (too many parameters)
- Underfitting can also occur (high bias)
  - Caused by a model's not being complex enough
- Both suffers from low performance on unseen data



Variance and Bias

- ***Variance*** measures the consistency (or the variability) of the model prediction for a particular sample instance if we were to retrain the model multiple times
- ***Bias*** measures how far off the predictions are from the correct values in general if we rebuild the model multiple times on different training datasets

Tackling Overfitting via Regularization

- A concept to confine the complexity of a model
  - Helps to filter out noise from training data
- Introduce additional information (bias, **penalty**) to penalize extreme parameter (weight) values
  - Most common form is L2 regularization. Sometimes called L2 shrinkage or weight decay) $\frac{\lambda}{2}\|\boldsymbol{w}\|^2 = \frac{\lambda}{2}\sum_{j=1}^{m} w_j^2$
- Regularized cost function

λ: regularization parameter

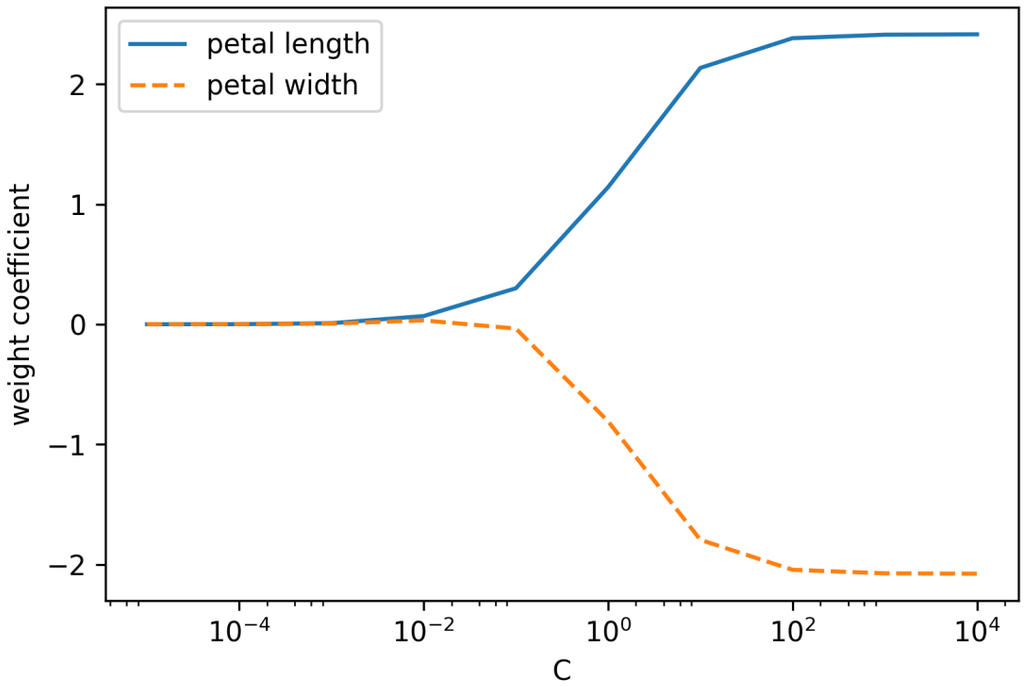
$$J(\boldsymbol{w}) = \sum_{i=1}^{n}\left[-y^{(i)}\log\left(\phi\left(z^{(i)}\right)\right) - \left(1-y^{(i)}\right)\log\left(1-\phi\left(z^{(i)}\right)\right)\right] + \frac{\lambda}{2}\|w\|^2$$

Regularization Parameter

- Control how well we fit the training data via λ
- By increasing λ , we increase the strength of regularization
- Sometimes, SVM terminology is used $C = \frac{1}{\lambda}$
  - cost function in previous page divided by λ

Regularization Illustrated (1/2)

```
weights, params = [], []
for c in np.arange(-5, 5):
    lr = LogisticRegression(C=10.**c, random_state=1)
    lr.fit(X_train_std, y_train)
    weights.append(lr.coef_[1])
    params.append(10.**c)

weights = np.array(weights)
plt.plot(params, weights[:, 0],
         label='petal length')
plt.plot(params, weights[:, 1], linestyle='--',
         label='petal width')
plt.ylabel('weight coefficient')
plt.xlabel('C')
plt.legend(loc='upper left')
plt.xscale('log')
#plt.savefig('images/03_08.png', dpi=300)
plt.show()
```



Regularization Illustrated (2/2)

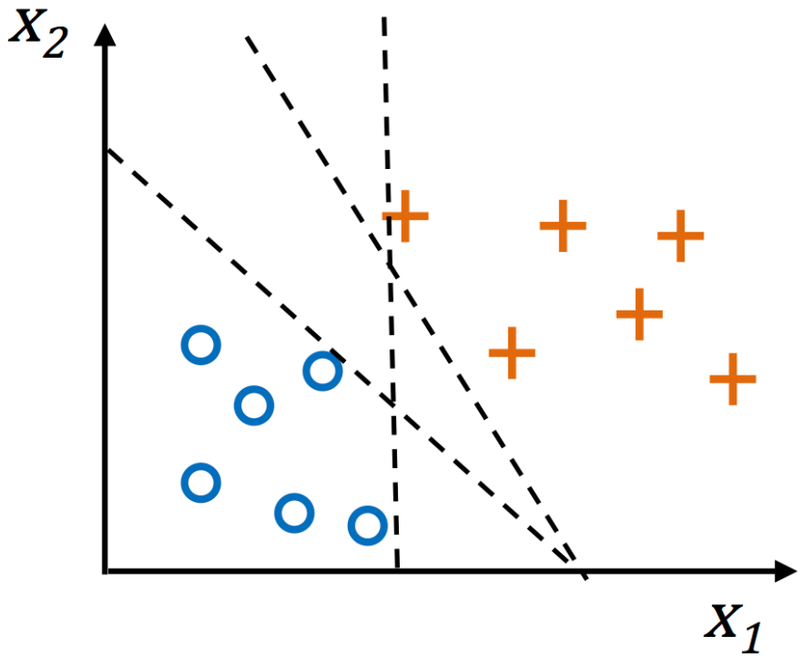
- Figure setup
  - Ten logistic regression models are fitted with different values of C
  - Only the weight coefficients in the binary logistic regression classifier for *class 1* versus all the others are collected
- Decreasing the value of C means increasing the regularization strength
- Weights shrink to zero as C decrease

Maximum Margin Classification with Support Vector Machines

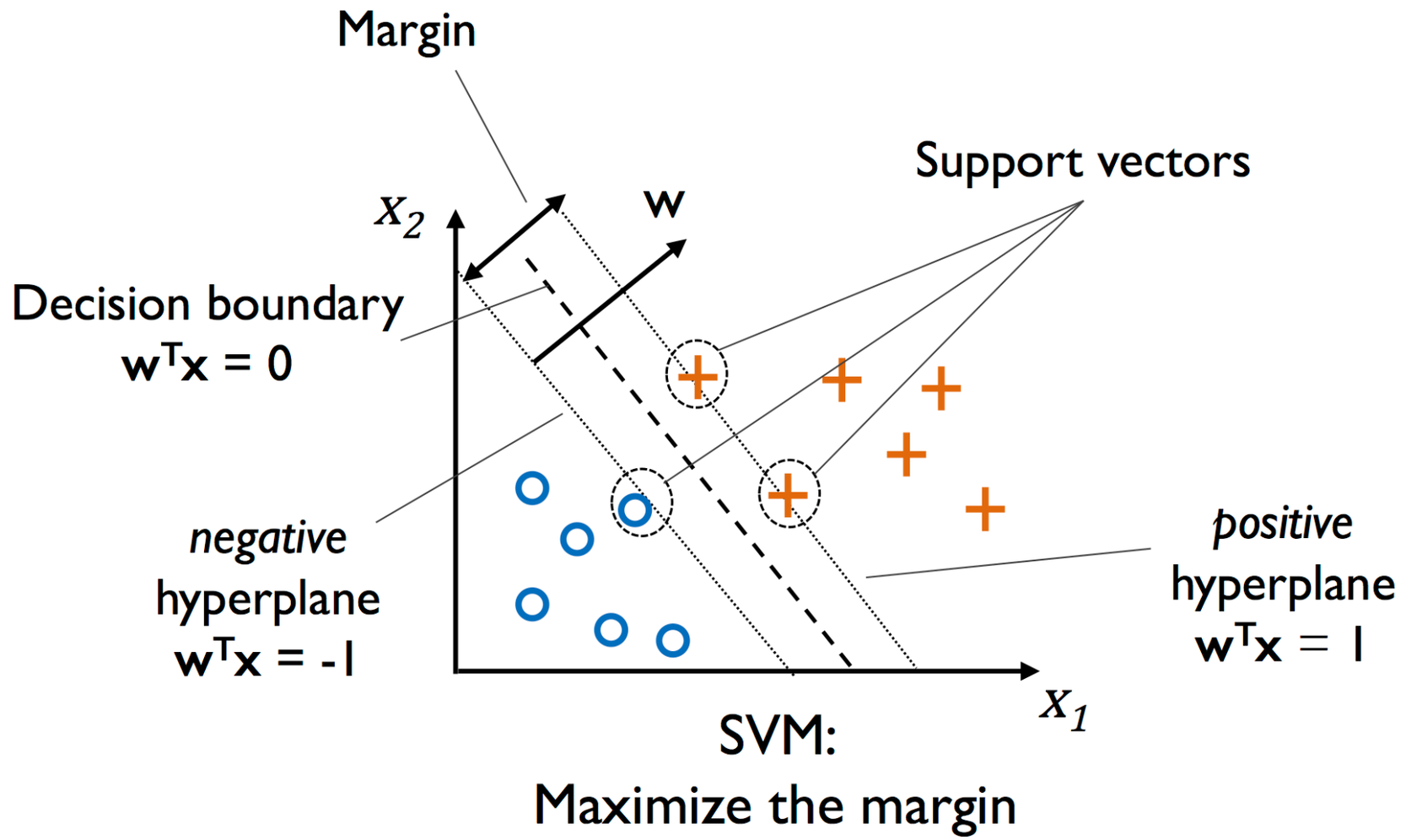
Support Vector Machines

- In SVMs, the optimization objective is to maximize the ***margin***
  - In Perceptron, we minimize the misclassification errors
  - The **margin** is defined as the distance between the separating hyperplane and the training samples that are closest to this hyperplane (***support vectors***)
- Intuitively, the larger the margin, the lower generalization error
- Models with small margin prone to overfitting

Maximum Margin of a Linear Separable Labeled Training Dataset



Which hyperplane?



SVM:
Maximize the margin

Mathematical Intuition

- ***Positive*** and ***negative*** hyperplanes that are parallel to the decision boundary

$$w_0 + \boldsymbol{w}^T \boldsymbol{x}_{pos} = 1 \qquad w_0 + \boldsymbol{w}^T \boldsymbol{x}_{neg} = -1$$

- Subtract the two equations

$$\boldsymbol{w}^T \left(\boldsymbol{x}_{pos} - \boldsymbol{x}_{neg} \right) = 2$$

- Normalized by the length of the vector $\|\boldsymbol{w}\| = \sqrt{\sum_{j=1}^{m} w_j^2}$

$$\frac{\boldsymbol{w}^T \left(\boldsymbol{x}_{pos} - \boldsymbol{x}_{neg} \right)}{\|\boldsymbol{w}\|} = \frac{2}{\|\boldsymbol{w}\|}$$

distance between positive and negative hyperplane

margin we want to **maximize**

Constrained Optimization Problem

- Maximize the margin by minimizing $\frac{1}{2}\|\mathbf{w}\|^2$
  - Subject to the constraints that the samples are classified correctly

$$w_0 + \boldsymbol{w}^T \boldsymbol{x}^{(i)} \geq 1 \; if \; y^{(i)} = 1$$
$$w_0 + \boldsymbol{w}^T \boldsymbol{x}^{(i)} \leq -1 \; if \; y^{(i)} = -1$$

- The two equations say that all negative and positive samples should fall respectively on one side of the negative and positive hyperplanes
  - Rewritten constraint more compactly

$$y^{(i)}\left(w_0 + \boldsymbol{w}^T \boldsymbol{x}^{(i)}\right) \geq 1 \; \forall_i$$

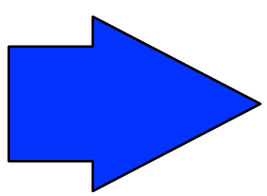
Dual Formulation (1/2)

- Primal formulation
  - minimize $\frac{1}{2}||\mathbf{w}||^2$ — objective function
  - subject to $y^{(i)}(w_0 + \mathbf{w}^T\mathbf{x}^{(i)}) - 1 \geq 0, \forall i$ — constraints
- Apply Lagrange multiplier method
  - minimize $L_P = \frac{1}{2}||\mathbf{w}||^2 - \sum_i \alpha^{(i)}(y^{(i)}(w_0 + \mathbf{w}^T\mathbf{x}^{(i)}) - 1), \forall i$

$$\alpha^{(i)} \geq 0$$

$$\frac{dL}{d\mathbf{w}} = \mathbf{w} - \sum_i \alpha^{(i)} y^{(i)} \mathbf{x}^{(i)}$$

$$\frac{dL}{dw_0} = \sum_i \alpha^{(i)} y^{(i)}$$



$$\mathbf{w} = \sum_i \alpha^{(i)} y^{(i)} \mathbf{x}^{(i)}$$

$$\sum_i \alpha^{(i)} y^{(i)} = 0$$

Dual Formulation (2/2)

- Dual formulation (replace into Lp)
  - maximize

$$L_D = \sum \alpha^{(i)} - \frac{1}{2} \sum_i \sum_j \alpha^{(i)} \alpha^{(j)} y^{(i)} y^{(j)} (\mathbf{x}^{(i)})^T \mathbf{x}^{(j)}, \forall i, j$$

  - subject to $\alpha^{(i)} \geq 0$ $\sum_i \alpha^{(i)} y^{(i)} = 0$

- Predict function (classifier)

$$w_0 + \mathbf{w}^T \mathbf{x} = w_0 + \sum_i \alpha^{(i)} y^{(i)} (\mathbf{x}^{(i)})^T \mathbf{x}, \forall i$$

new unknown sample

SVM Solution

- Classifier

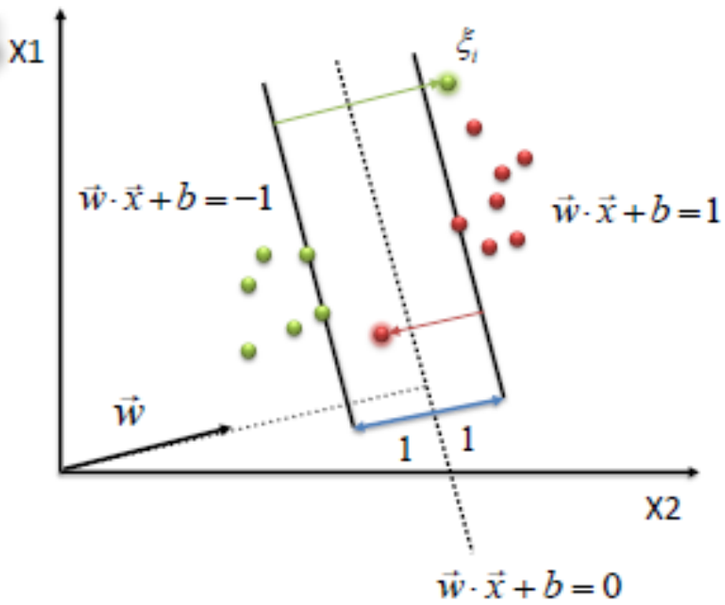
$$f(\mathbf{x}) = sgn(\mathbf{w}^T\mathbf{x} + w_0)$$

- Weights

$$\mathbf{w} = \sum_{i=1}^{N} \alpha_i y_i \mathbf{x}_i$$

Extend SVM to Nonlinear Separable Cases

- Need to ***relax*** the linear constraints for nonlinear separable data to ensure convergence in presence of misclassifications



- Introduce slack variables ξ

$$w_0 + \boldsymbol{w}^T \boldsymbol{x}^{(i)} \geq 1 - \xi^{(i)} \quad if \quad y^{(i)} = 1$$

$$w_0 + \boldsymbol{w}^T \boldsymbol{x}^{(i)} \leq -1 + \xi^{(i)} \quad if \quad y^{(i)} = -1$$

$$y^{(i)}(w_0 + \mathbf{w}^T \mathbf{x}^{(i)}) \geq 1 - \xi^{(i)}$$

- New objective to be minimized

$$\frac{1}{2}\|\boldsymbol{w}\|^2 + C\left(\sum_i \xi^{(i)}\right)$$

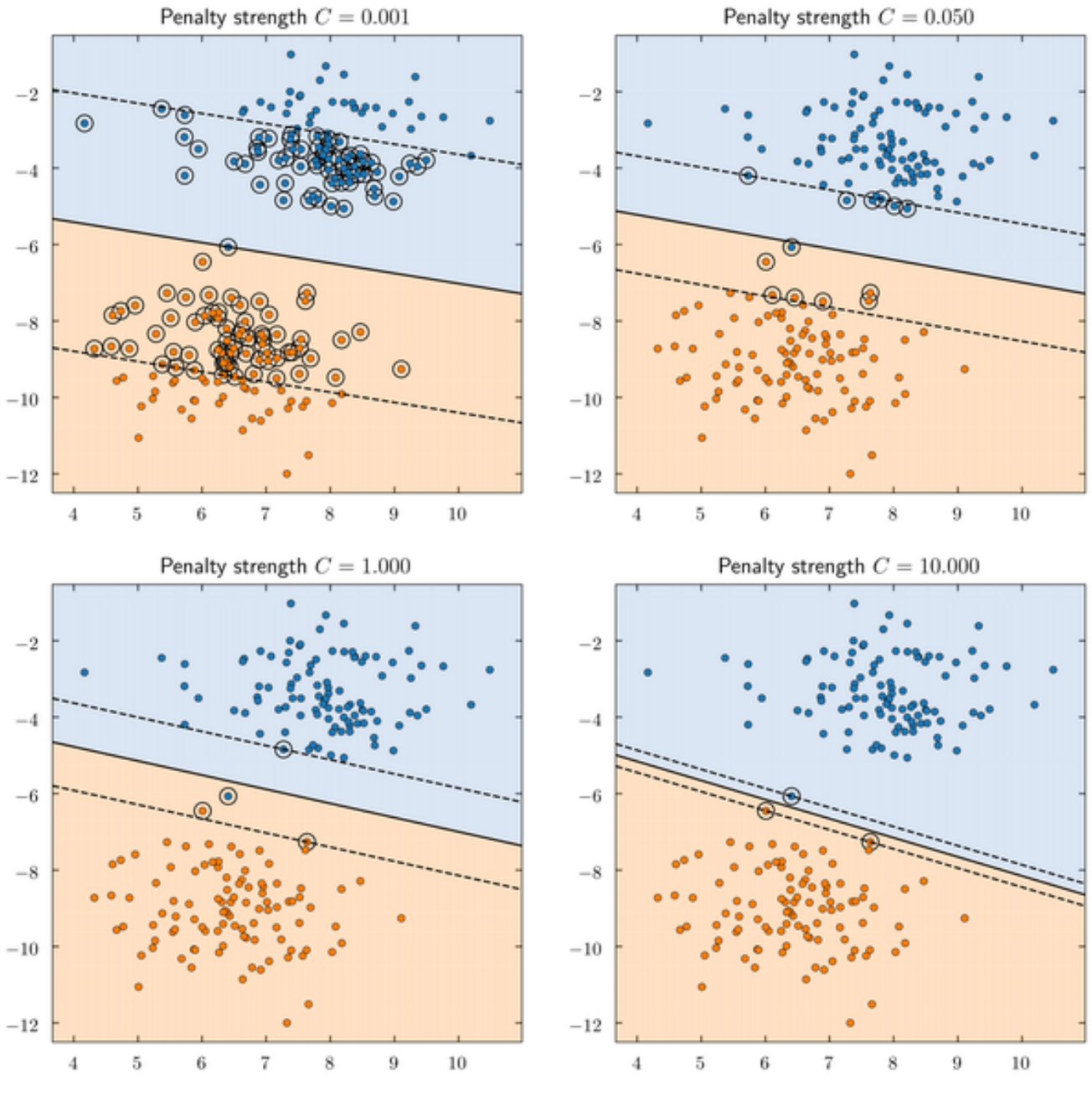
Regularization in SVMs

- Parameter C controls width of the margin
  - Large values of C - large error penalties
  - Small values of C - less strict about misclassification

$$\frac{1}{2}\|\boldsymbol{w}\|^2 + C\left(\sum_i \xi^{(i)}\right)$$

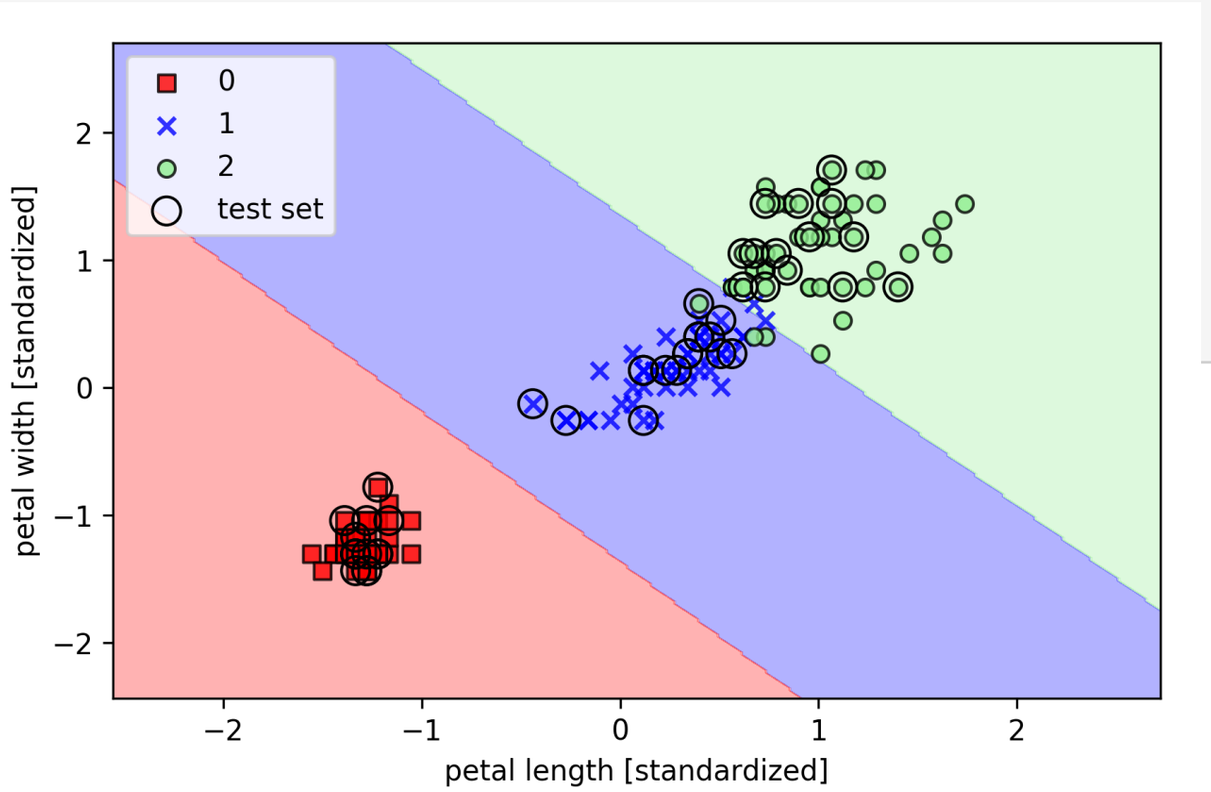
- Soft margin

$$\rho = 1/\|\mathbf{w}\|$$



Train a Scikit-learn SVM Model

```
from sklearn.svm import SVC

svm = SVC(kernel='linear', C=1.0, random_state=1)
svm.fit(X_train_std, y_train)

plot_decision_regions(X_combined_std,
                      y_combined,
                      classifier=svm,
                      test_idx=range(105, 150))
plt.xlabel('petal length [standardized]')
plt.ylabel('petal width [standardized]')
plt.legend(loc='upper left')
plt.tight_layout()
#plt.savefig('images/03_11.png', dpi=300)
plt.show()
```



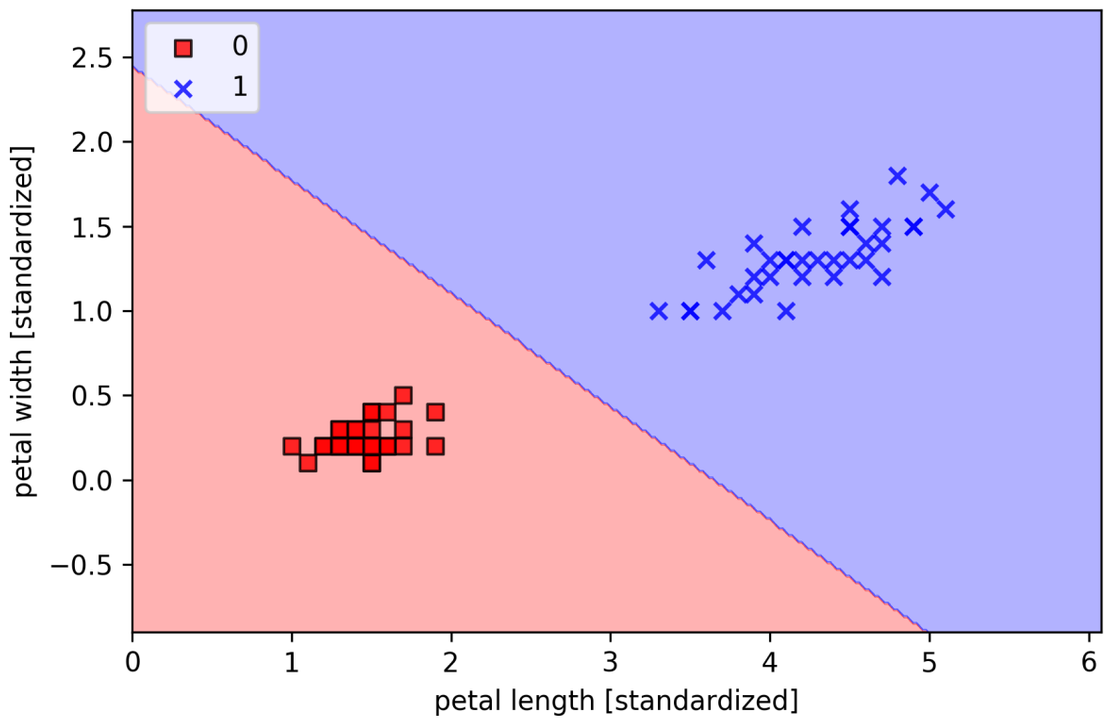
Alternative Implementation Online Learning

- Scikit-learn offers **SGDClassifier** class which supports online learning via the **partial_fit** method.
- By initialization of the stochastic gradient descent version of perceptron, logistic regression, or SVM with default parameters, we can do online learning

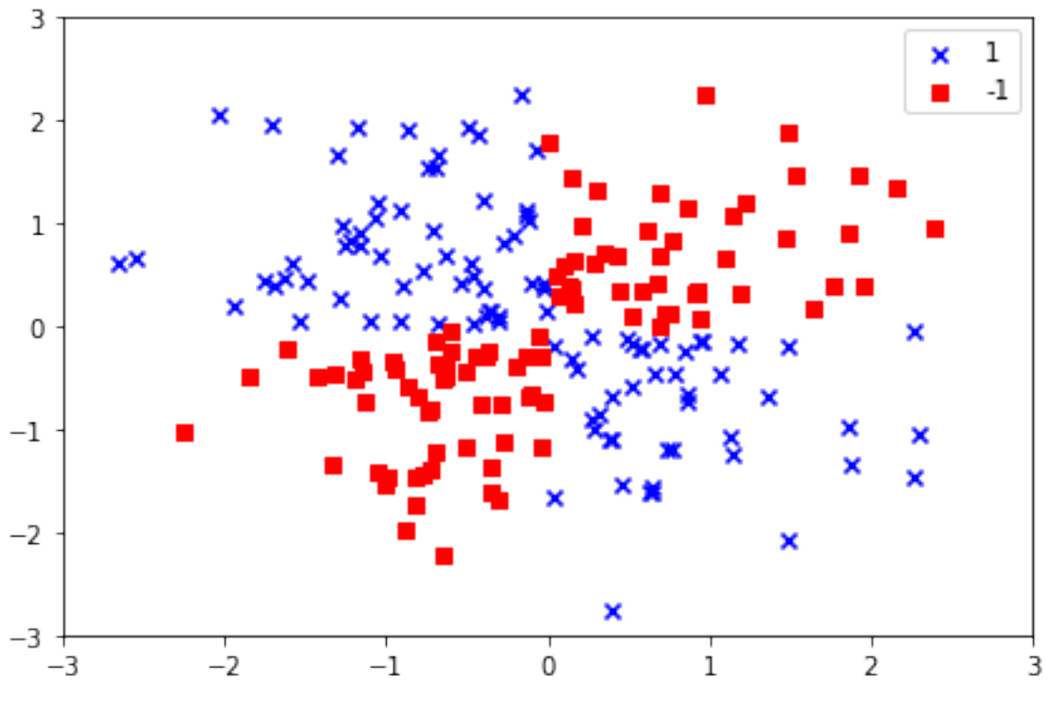
```
from sklearn.linear_model import SGDClassifier

ppn = SGDClassifier(loss='perceptron', n_iter=1000)
lr = SGDClassifier(loss='log', n_iter=1000)
svm = SGDClassifier(loss='hinge', n_iter=1000)
```

Solving Linear Inseparability Problems Using a Kernel SVM

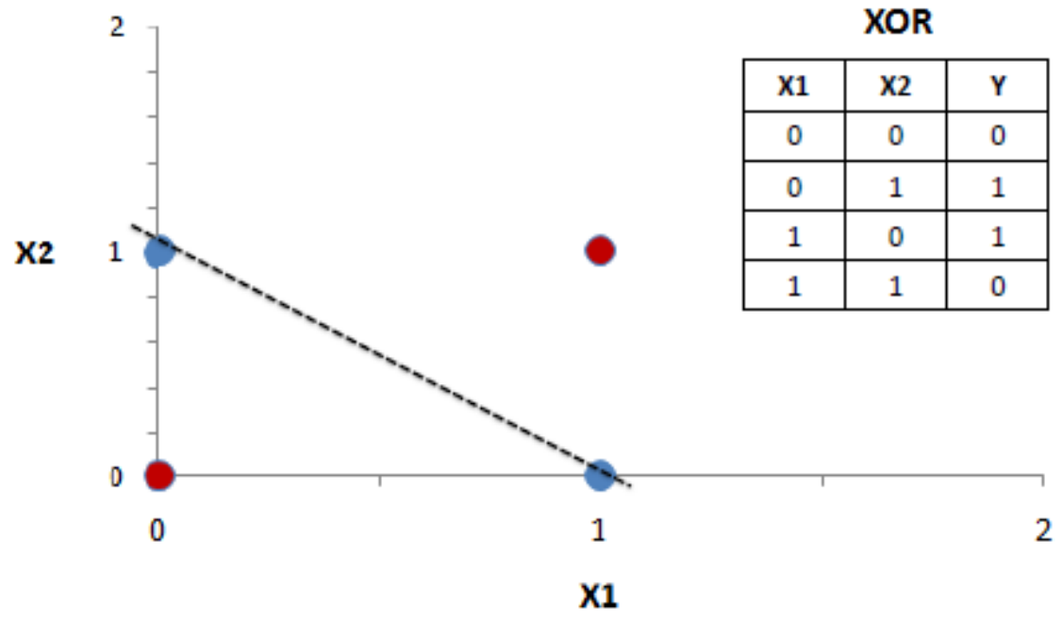
Linear Separability



linear separable

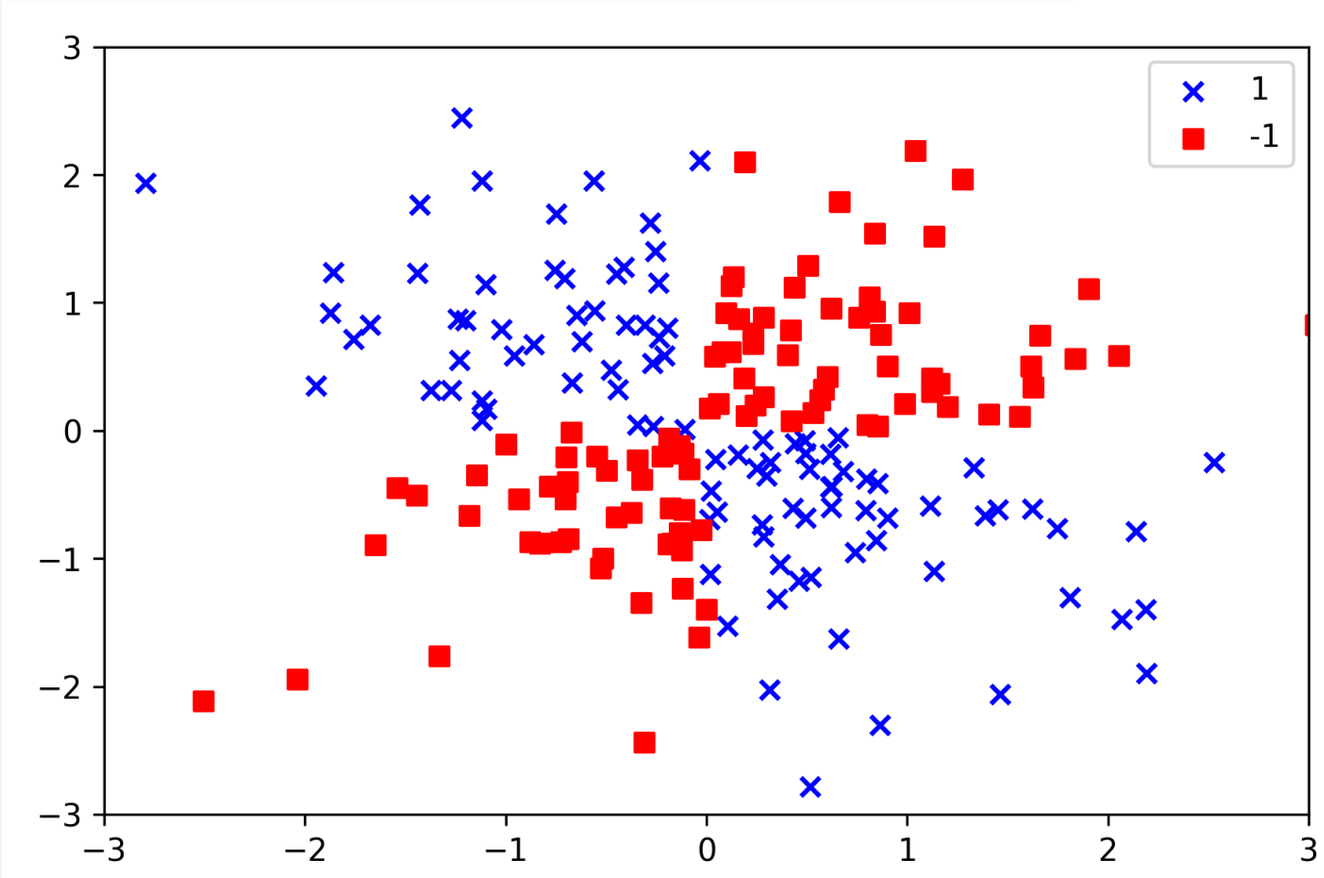


linear inseparable



Generated XOR Data

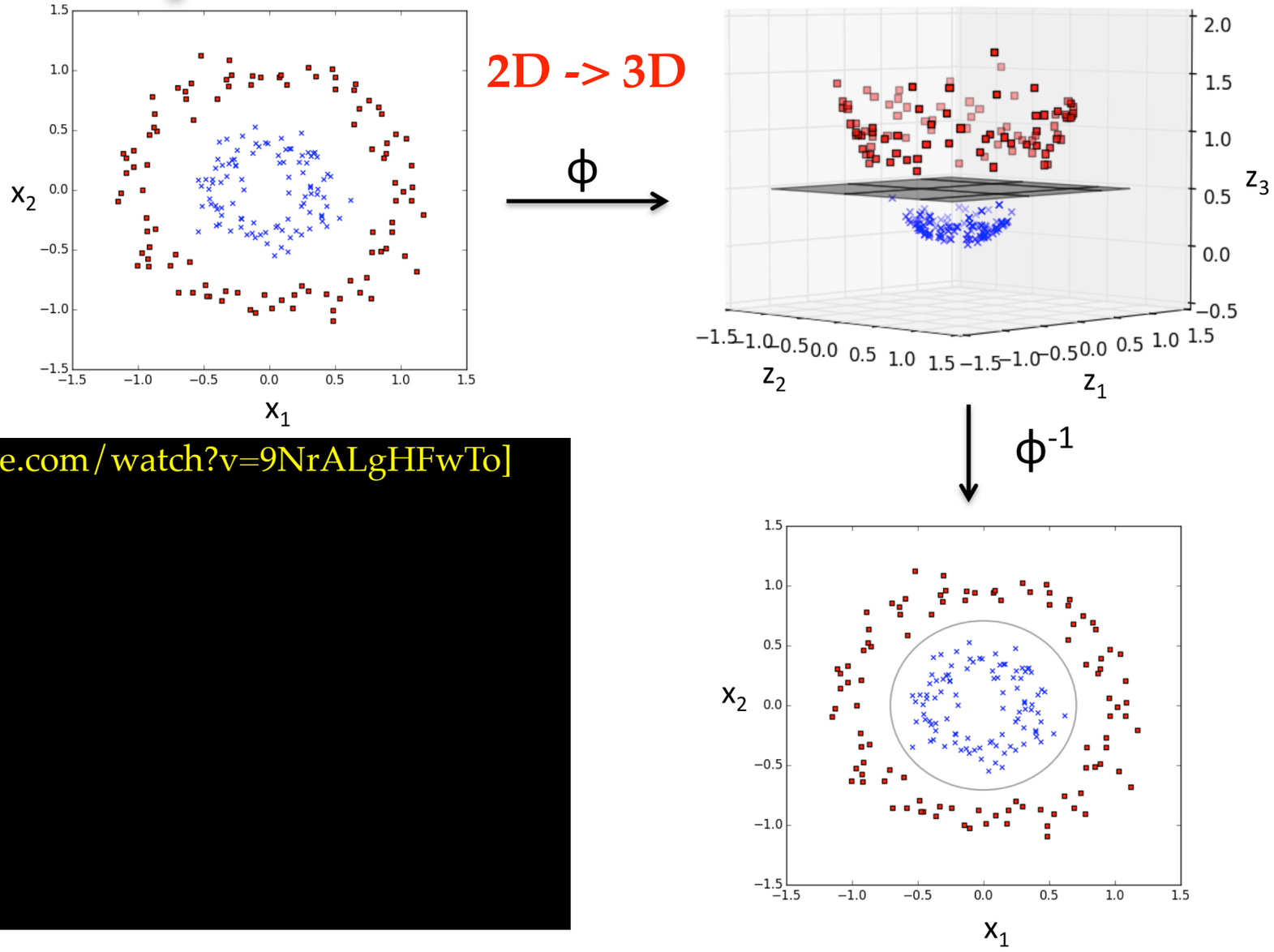
```
import matplotlib.pyplot as plt
import numpy as np

np.random.seed(1)
X_xor = np.random.randn(200, 2)
y_xor = np.logical_xor(X_xor[:, 0] > 0,
                       X_xor[:, 1] > 0)
y_xor = np.where(y_xor, 1, -1)

plt.scatter(X_xor[y_xor == 1, 0],
            X_xor[y_xor == 1, 1],
            c='b', marker='x',
            label='1')
plt.scatter(X_xor[y_xor == -1, 0],
            X_xor[y_xor == -1, 1],
            c='r',
            marker='s',
            label='-1')

plt.xlim([-3, 3])
plt.ylim([-3, 3])
plt.legend(loc='best')
plt.tight_layout()
#plt.savefig('images/03_12.png', dpi=300)
plt.show()
```



Nonlinear Separation

- Kernel methods create ***nonlinear combinations of the original features*** to ***project*** them ***onto a high dimensional space*** via a mapping function ϕ where is becomes linear ***separable***
- Example

$$\phi(x_1, x_2) = (z_1, z_2, z_3) = (x_1, x_2, x_1^2 + x_2^2)$$



[https://www.youtube.com/watch?v=9NrALgHFwTo]

Kernel Trick: Explicit Mapping for Linear Inseparable Problems

- Explicit mapping
  - Transform training data into a higher dimensional space via a mapping function ϕ
  - Train a linear SVM to classify the data in the new feature space
  - Use the same mapping function ϕ to transform new (unseen) data
  - Classify unseen data using the linear SVM model
- Computationally expensive

Kernel Functions

- Computationally expensive issue
  - During training, decision boundary only rely on ***pair-wise*** dot products $\boldsymbol{x}^{(i)T}\boldsymbol{x}^{(j)}$
  - In the transformation, we need $\phi\left(\boldsymbol{x}^{(i)}\right)^T \phi\left(\boldsymbol{x}^{(j)}\right)$
- A kernel K is a function $K : \mathbb{R}^N \times \mathbb{R}^N \to \mathbb{R}$, effectively computes dot products in a higher-dimensional space $\mathbb{R}^M$ while remaining in $\mathbb{R}^N$

$$\mathcal{K}\left(\boldsymbol{x}^{(i)}, \boldsymbol{x}^{(j)}\right) = \phi\left(\boldsymbol{x}^{(i)}\right)^T \phi\left(\boldsymbol{x}^{(j)}\right)$$

$$x^{(i)} \in \mathbb{R}^N, \phi(x^{(i)}) \in \mathbb{R}^M, M > N$$

Kernel Trick

- Dual formulation
  - maximize

$$L_D = \sum \alpha^{(i)} - \frac{1}{2} \sum_i \sum_j \alpha^{(i)} \alpha^{(j)} y^{(i)} y^{(j)} \boxed{(\mathbf{x}^{(i)})^T \mathbf{x}^{(j)}}, \forall i, j$$

  - subject to $\alpha^{(i)} \geq 0$ $\sum_i \alpha^{(i)} y^{(i)} = 0$

- After mapping

$$L_D' = \sum \alpha^{(i)} - \frac{1}{2} \sum_i \sum_j \alpha^{(i)} \alpha^{(j)} y^{(i)} y^{(j)} \boxed{(\phi(\mathbf{x}^{(i)}))^T \phi(\mathbf{x}^{(j)})}, \forall i, j$$

$$\mathcal{K}\left(\boldsymbol{x}^{(i)}, \boldsymbol{x}^{(j)}\right)$$

  - subject to $\alpha^{(i)} \geq 0$ $\sum_i \alpha^{(i)} y^{(i)} = 0$

Kernel Functions

- Kernel can be interpreted as a similarity function between a pair of samples
  - Similar data points tend to get together to form decision boundary
- Many types kernel functions
  - linear, nonlinear, polynomial, sigmoid, …
- Most used: **r**adial **b**asis **f**unction (RBF) kernel or Gaussian kernel

$$\mathcal{K}\left(\boldsymbol{x}^{(i)},\boldsymbol{x}^{(j)}\right)=\exp\left(-\frac{\left\|\boldsymbol{x}^{(i)}-\boldsymbol{x}^{(j)}\right\|^2}{2\sigma^2}\right)$$

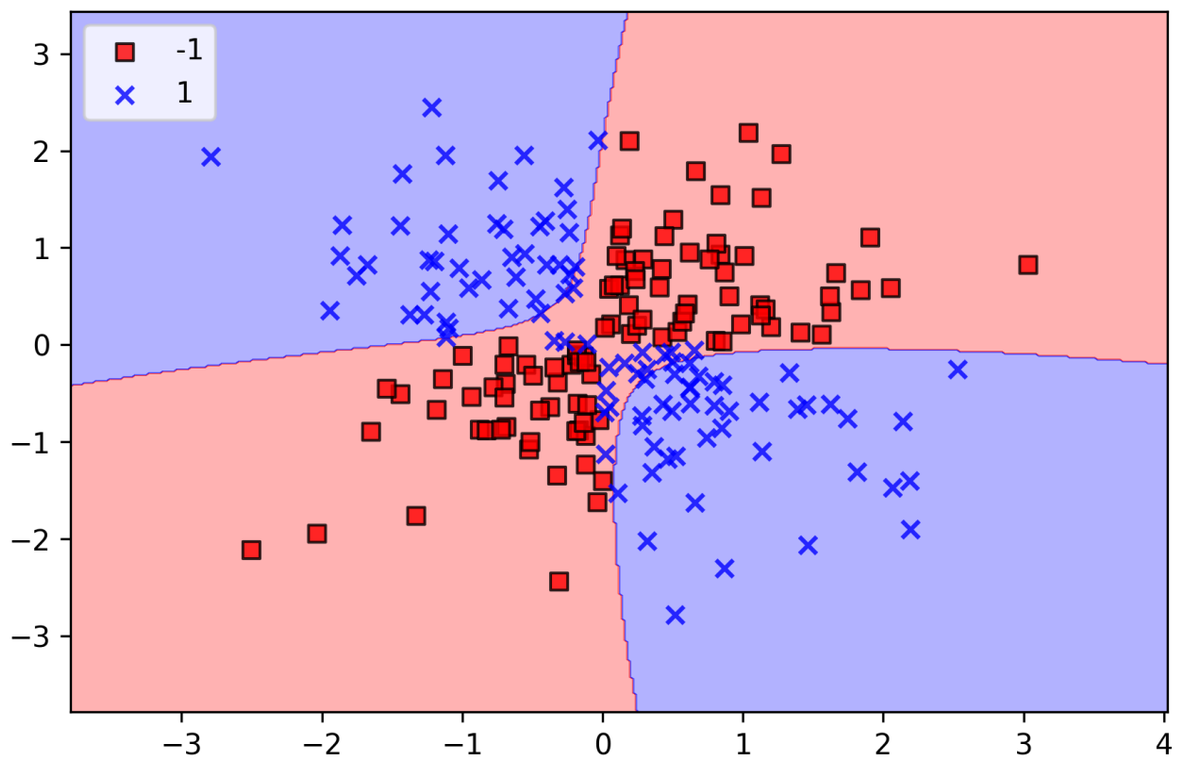
$$\mathcal{K}\left(\boldsymbol{x}^{(i)},\boldsymbol{x}^{(j)}\right)=\exp\left(-\gamma\left\|\boldsymbol{x}^{(i)}-\boldsymbol{x}^{(j)}\right\|^2\right)$$

$$\gamma=\frac{1}{2\sigma^2}$$ **decay rate, to be optimized**

  - The similarity measure by RBF kernel will decrease exponentially as the Euclidean distance between these two points increases

Use RBF Kernel to Separate XOR Data (1/3)

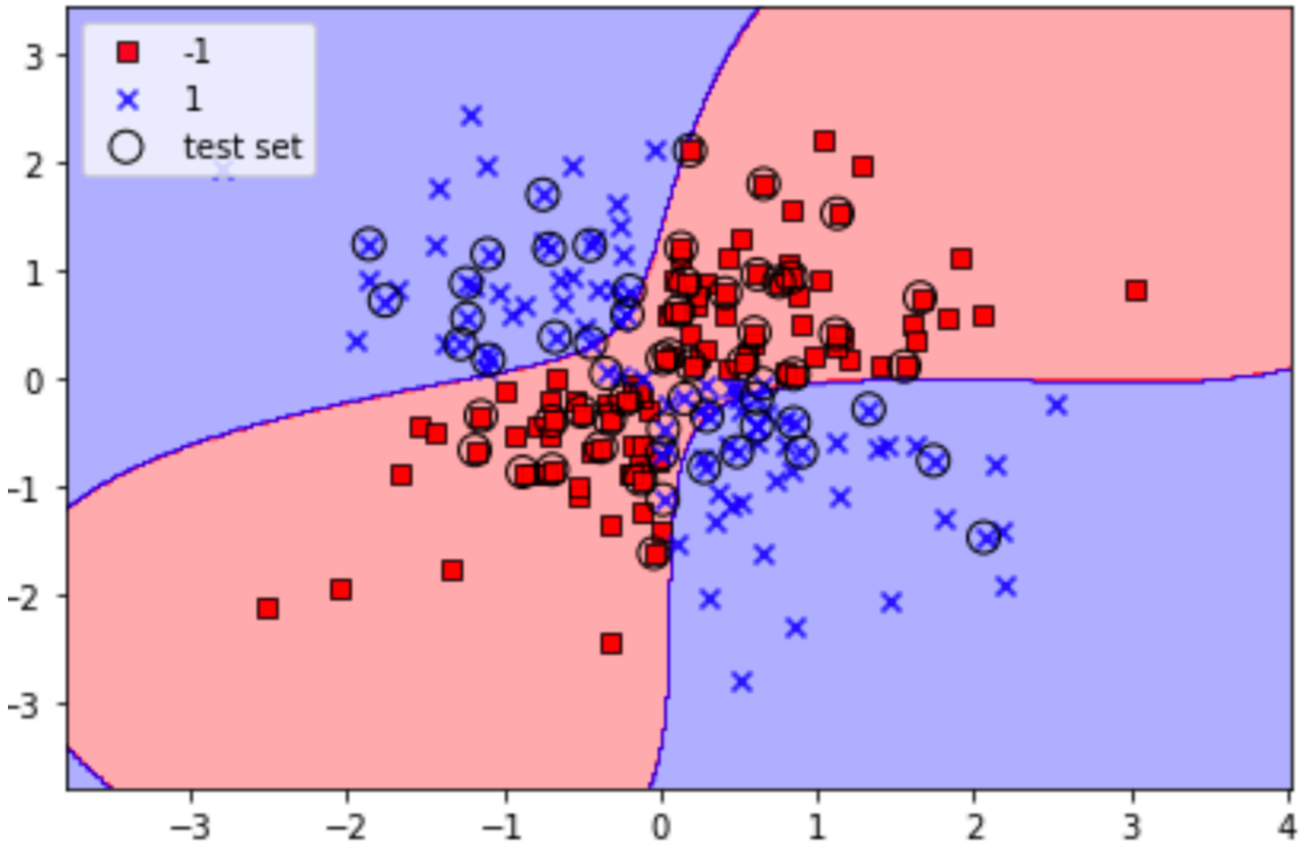
```
svm = SVC(kernel='rbf', random_state=1, gamma=0.10, C=10.0)
svm.fit(X_xor, y_xor)
plot_decision_regions(X_xor, y_xor,
                      classifier=svm)

plt.legend(loc='upper left')
plt.tight_layout()
#plt.savefig('images/03_14.png', dpi=300)
plt.show()
```



Use RBF Kernel to Separate XOR Data (2/3)

```
X_xor_train, X_xor_test, y_xor_train, y_xor_test = train_test_split(
    X_xor, y_xor, test_size=0.3, random_state=1, stratify=y_xor)

svm = SVC(kernel='rbf', random_state=1, gamma=0.17, C=10.0)
svm.fit(X_xor_train, y_xor_train)

X_xor_combined = np.vstack((X_xor_train, X_xor_test))
y_xor_combined = np.hstack((y_xor_train, y_xor_test))

plot_decision_regions(X=X_xor_combined, y=y_xor_combined,
                      classifier=svm, test_idx=range(140, 200))
```

```
plt.legend(loc='upper left')
plt.tight_layout()
#plt.savefig('images/03_14.png', dpi=300)
plt.show()
```



Use RBF Kernel to Separate XOR Data (3/3)

```
In [42]: y_xor_pred = svm.predict(X_xor_test)
         print('Misclassified instances: %d' % (y_xor_test != y_xor_pred).sum())

Misclassified instances: 7
```

```
In [43]: print('Accuracy: %.2f' % accuracy_score(y_xor_test, y_xor_pred))

Accuracy: 0.88
```

```
In [44]: print('Accuracy: %.2f' % svm.score(X_xor_test, y_xor_test))

Accuracy: 0.88
```

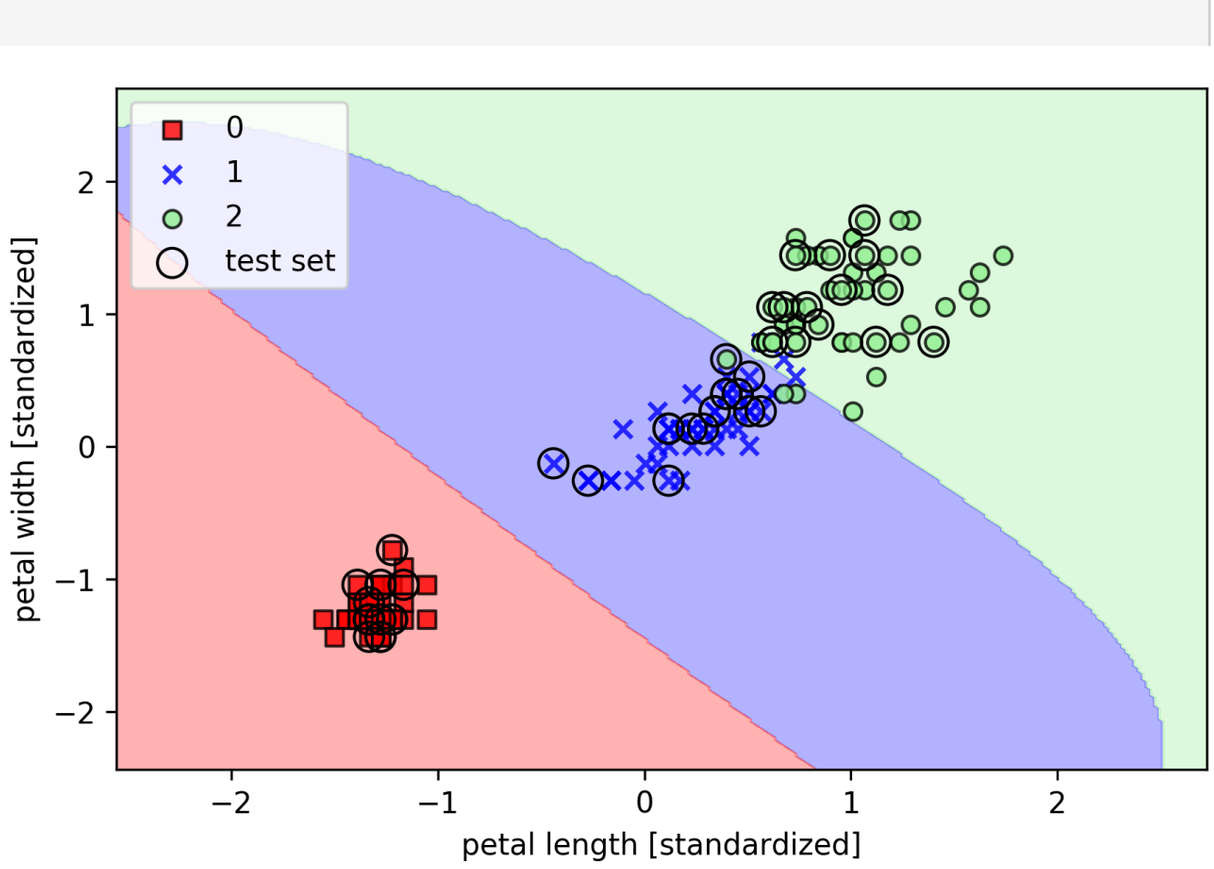
The parameter γ

- Cut-off parameter for the Gaussian sphere
- How far the influence of a single training sample reaches
- Exponential decaying rate of similarity measure
- Increase the value will ask data points to be more closer to each other to keep the same level of similarity and then leads to a tighter and bumpier decision boundary

Small Value of γ

```
from sklearn.svm import SVC

svm = SVC(kernel='rbf', random_state=1, gamma=0.2, C=1.0)
svm.fit(X_train_std, y_train)

plot_decision_regions(X_combined_std, y_combined,
                      classifier=svm, test_idx=range(105, 150))
plt.xlabel('petal length [standardized]')
plt.ylabel('petal width [standardized]')
plt.legend(loc='upper left')
plt.tight_layout()
#plt.savefig('images/03_15.png', dpi=300)
plt.show()
```

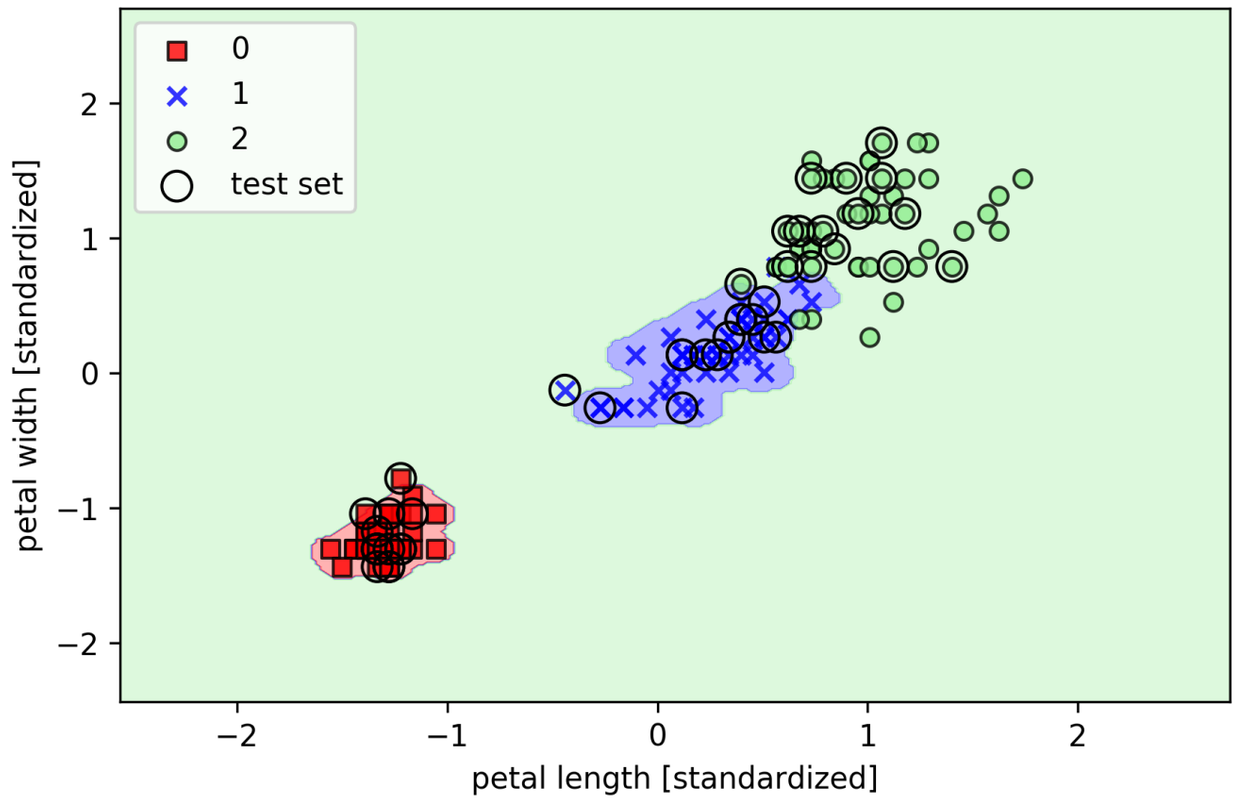
```
In [46]: y_pred = svm.predict(X_test_std)
         print('Misclassified instances: %d' % (y_test != y_pred).sum())

Misclassified instances: 1


In [47]: print('Accuracy: %.2f' % accuracy_score(y_test, y_pred))

Accuracy: 0.98


In [48]: print('Accuracy: %.2f' % svm.score(X_test_std, y_test))

Accuracy: 0.98
```



Large Value of γ

```
svm = SVC(kernel='rbf', random_state=1, gamma=100.0, C=1.0)
svm.fit(X_train_std, y_train)

plot_decision_regions(X_combined_std, y_combined,
                      classifier=svm, test_idx=range(105, 150))
plt.xlabel('petal length [standardized]')
plt.ylabel('petal width [standardized]')
plt.legend(loc='upper left')
plt.tight_layout()
#plt.savefig('images/03_16.png', dpi=300)
plt.show()
```

```
In [50]: y_pred = svm.predict(X_test_std)
         print('Misclassified instances: %d' % (y_test != y_pred).sum())

Misclassified instances: 3

In [51]: print('Accuracy: %.2f' % accuracy_score(y_test, y_pred))

Accuracy: 0.93

In [52]: print('Accuracy: %.2f' % svm.score(X_test_std, y_test))

Accuracy: 0.93
```

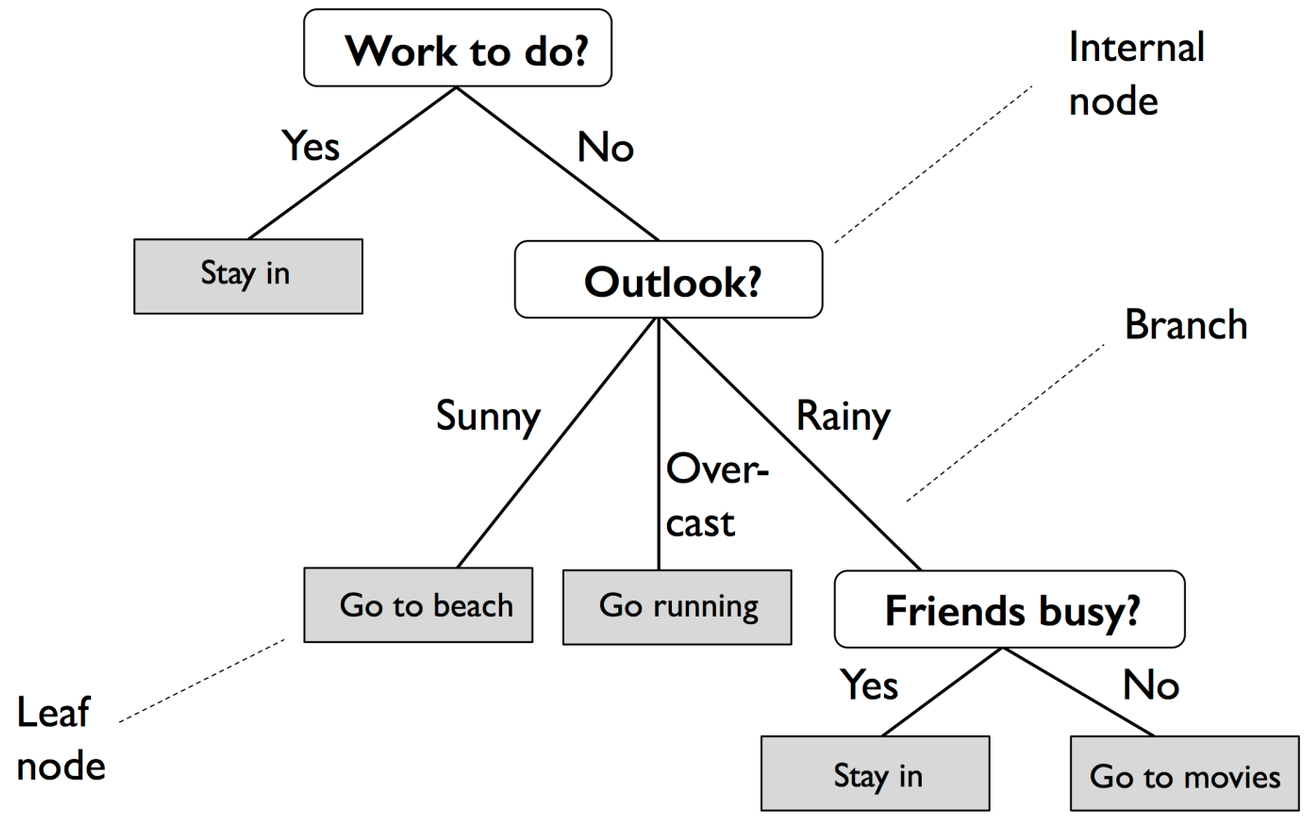


Decision Tree Learning

Decision Tree

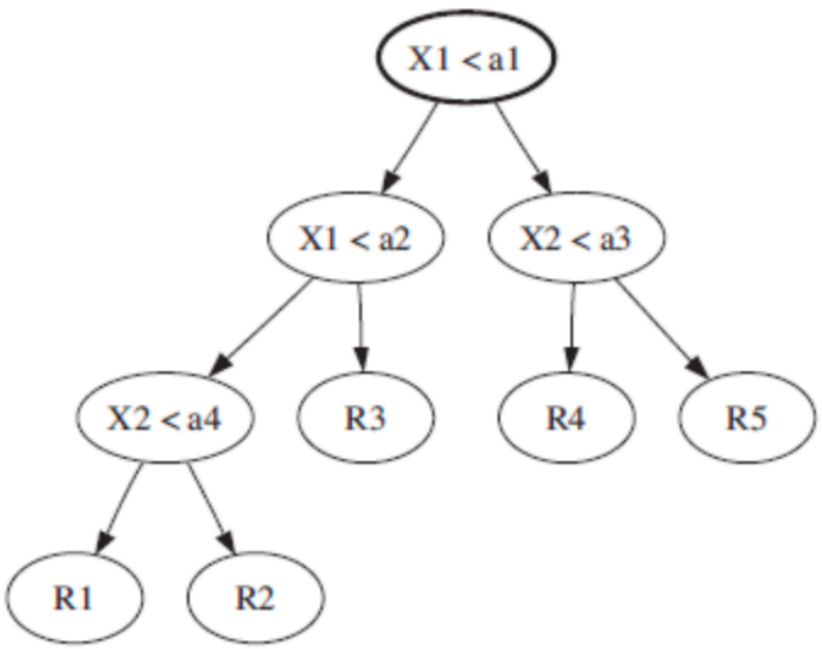
- Interpretability
- Tree representation of a partition of feature space X
  - Each **internal node** tests a feature x_j
  - **Branches**/**edges** are possible values/attributes of the feature
  - Each **leaf** (end of the branch) assigns a class label
- Classification and Regression Tree (CART)

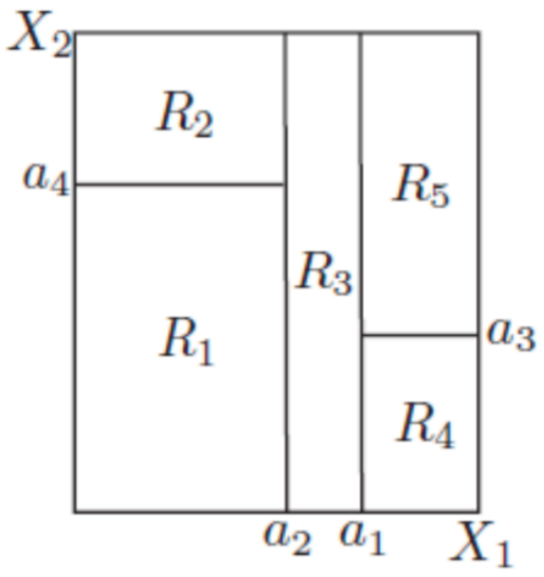
$$\mathcal{X} = \{ \textbf{Busy, Free} \} \times \{\textbf{Sunny, Overcast, Rainy} \} \times \{ \textbf{Friends\_busy, Friends\_free}\}$$



Binary Decision Tree

- For a binary decision tree a question can be
  - a numerical question
    - $x_j \leq a$ for a continuous feature variable x_j and some threshold $a \in \mathbb{R}$
  - a categorical question
    - $x_j \in$ {blue, white, red}





Considerations for Growing a Tree

- Which features to choose
- What conditions to use for splitting
- When to stop growing
  - As a tree grows arbitrarily, need to trim the tree down for it to look beautiful
  - The general problem of determining partition with minimum empirical error is NP-hard

Greedy Algorithm

- Algorithm
  - Start from empty decision tree
  - Split on next best feature $\arg\max_j IG(x_j)$
  - Recurse
- How to determine best feature
  - Maximize Information Gain (IG) $IG(D_p, f) = I(D_p) - \sum_{j=1}^{m} \frac{N_j}{N_p} I(D_j)$
    - *f*: feature to perform the split
    - D*p*, D*j*: dataset of the parent and *j*th child node
    - I: the impurity measure
    - N*p*, N*j*: number of samples at parent node and *j*th child node
  - For binary decision tree $IG(D_p, f) = I(D_p) - \frac{N_{left}}{N_p} I(D_{left}) - \frac{N_{right}}{N_p} I(D_{right})$

How to Stop Tree Growing

- Once all leaves have reached a sufficient level of impurity, when the number of items per leaf has become to small for further splitting
- Set maximum depth of the model

Three Most Commonly Used Measures of Node Impurity (1/4)

- Classification error $I_E = 1 - \max\{p(i|t)\}$
- Entropy $I_H(t) = -\sum_{i=1}^{c} p(i|t)\log_2 p(i|t)$
- Gini impurity $I_G(t) = \sum_{i=1}^{c} p(i|t)\left(1 - p(i|t)\right) = 1 - \sum_{i=1}^{c} p(i|t)^2$
  - $p(i|t)$ is the proportion of the samples that belong to attribute ***c*** for a particular node ***t***

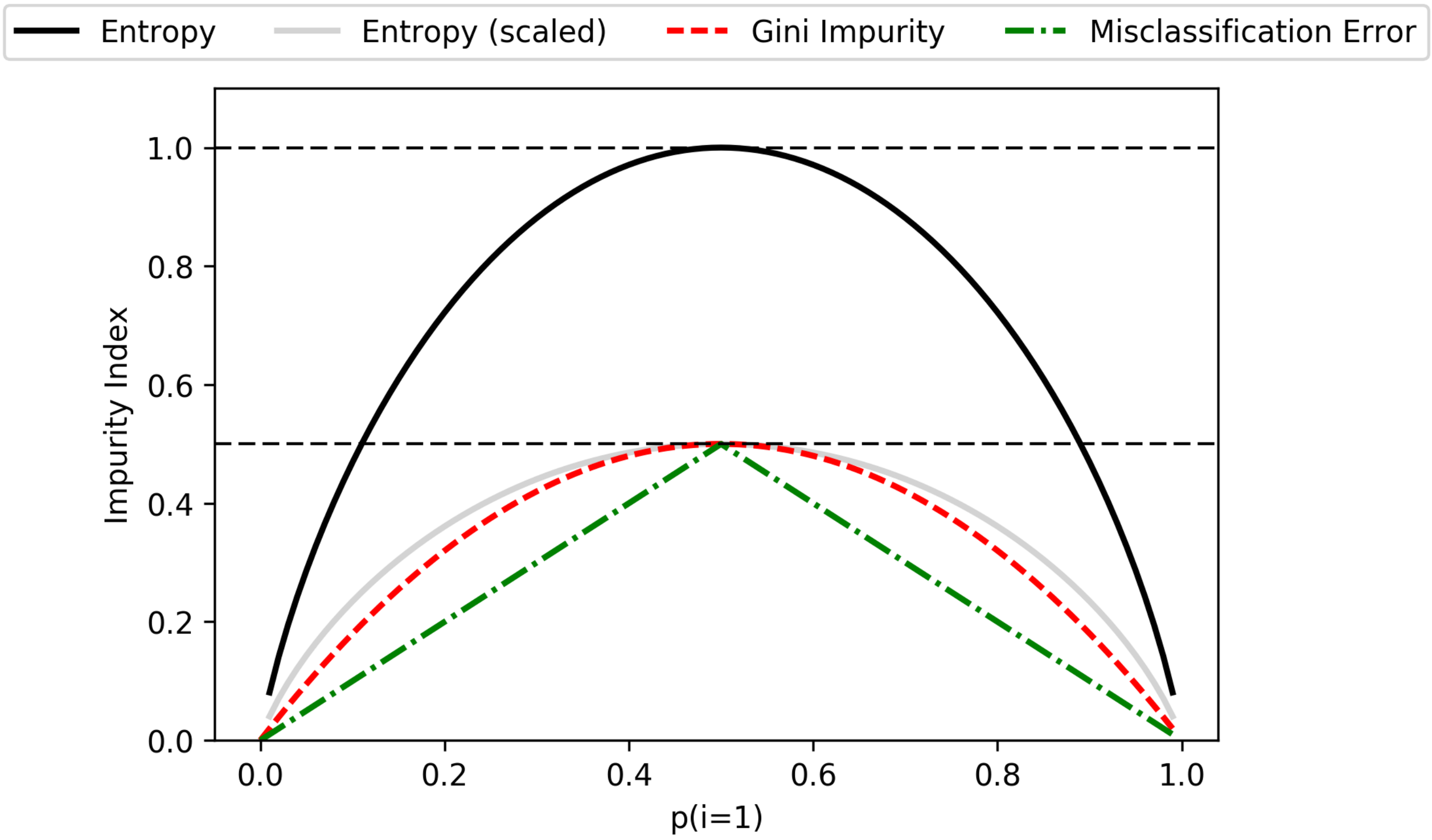
Three Most Commonly Used Measures of Node Impurity (2/4)

```
import matplotlib.pyplot as plt
import numpy as np


def gini(p):
    return p * (1 - p) + (1 - p) * (1 - (1 - p))


def entropy(p):
    return - p * np.log2(p) - (1 - p) * np.log2((1 - p))


def error(p):
    return 1 - np.max([p, 1 - p])

x = np.arange(0.0, 1.0, 0.01)

ent = [entropy(p) if p != 0 else None for p in x]
sc_ent = [e * 0.5 if e else None for e in ent]
err = [error(i) for i in x]
```

Three Most Commonly Used Measures of Node Impurity (3/4)

```
fig = plt.figure()
ax = plt.subplot(111)
for i, lab, ls, c, in zip([ent, sc_ent, gini(x), err],
                          ['Entropy', 'Entropy (scaled)',
                           'Gini Impurity', 'Misclassification Error'],
                          ['-', '-', '--', '-.'],
                          ['black', 'lightgray', 'red', 'green', 'cyan']):
    line = ax.plot(x, i, label=lab, linestyle=ls, lw=2, color=c)

ax.legend(loc='upper center', bbox_to_anchor=(0.5, 1.15),
          ncol=5, fancybox=True, shadow=False)

ax.axhline(y=0.5, linewidth=1, color='k', linestyle='--')
ax.axhline(y=1.0, linewidth=1, color='k', linestyle='--')
plt.ylim([0, 1.1])
plt.xlabel('p(i=1)')
plt.ylabel('Impurity Index')
#plt.savefig('images/03_19.png', dpi=300, bbox_inches='tight')
plt.show()
```

Three Most Commonly Used Measures of Node Impurity (4/4)



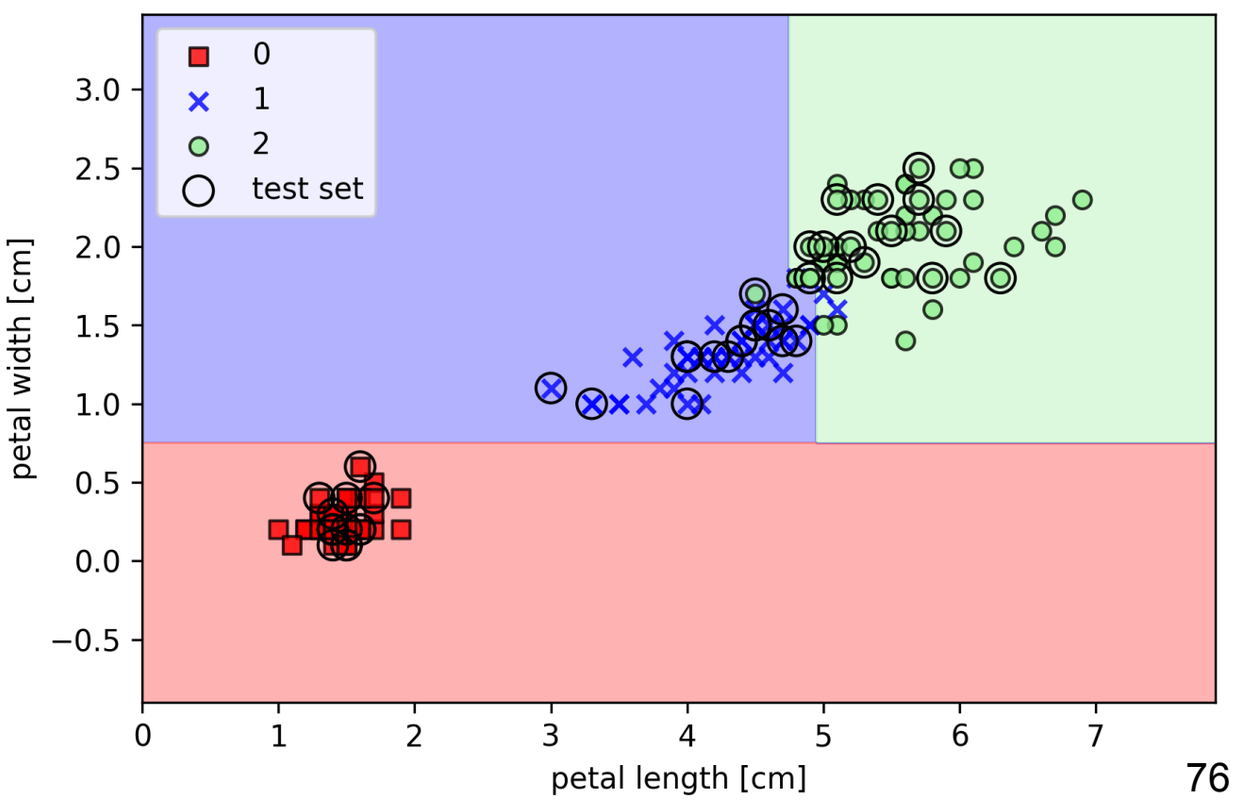
Issues of the Greedy Algorithm

- The greedy nature of the algorithm
  - A seemingly bad split may dominate subsequent useful splits, which could lead to trees with less impurity overall
- Using look-ahead of some depth *d* to determine the splitting decisions can overcome the issue
  - However, computationally cost
- To achieve some desired level of impurity, trees of relatively large sizes may be needed. But large trees may overfit.
  - Thus, typically prune the tree by setting a limit for the maximal depth of tree

Building a Decision Tree

```
from sklearn.tree import DecisionTreeClassifier

tree = DecisionTreeClassifier(criterion='gini',
                              max_depth=4,
                              random_state=1)
tree.fit(X_train, y_train)

X_combined = np.vstack((X_train, X_test))
y_combined = np.hstack((y_train, y_test))
plot_decision_regions(X_combined, y_combined,
                      classifier=tree, test_idx=range(105, 150))

plt.xlabel('petal length [cm]')
plt.ylabel('petal width [cm]')
plt.legend(loc='upper left')
plt.tight_layout()
#plt.savefig('images/03_20.png', dpi=300)
plt.show()
```

```
In [58]: y_pred = tree.predict(X_test)
         print('Misclassified instances: %d' % (y_test != y_pred).sum())

Misclassified instances: 1


In [59]: print('Accuracy: %.2f' % accuracy_score(y_test, y_pred))

Accuracy: 0.98


In [60]: print('Accuracy: %.2f' % tree.score(X_test, y_test))

Accuracy: 0.98
```



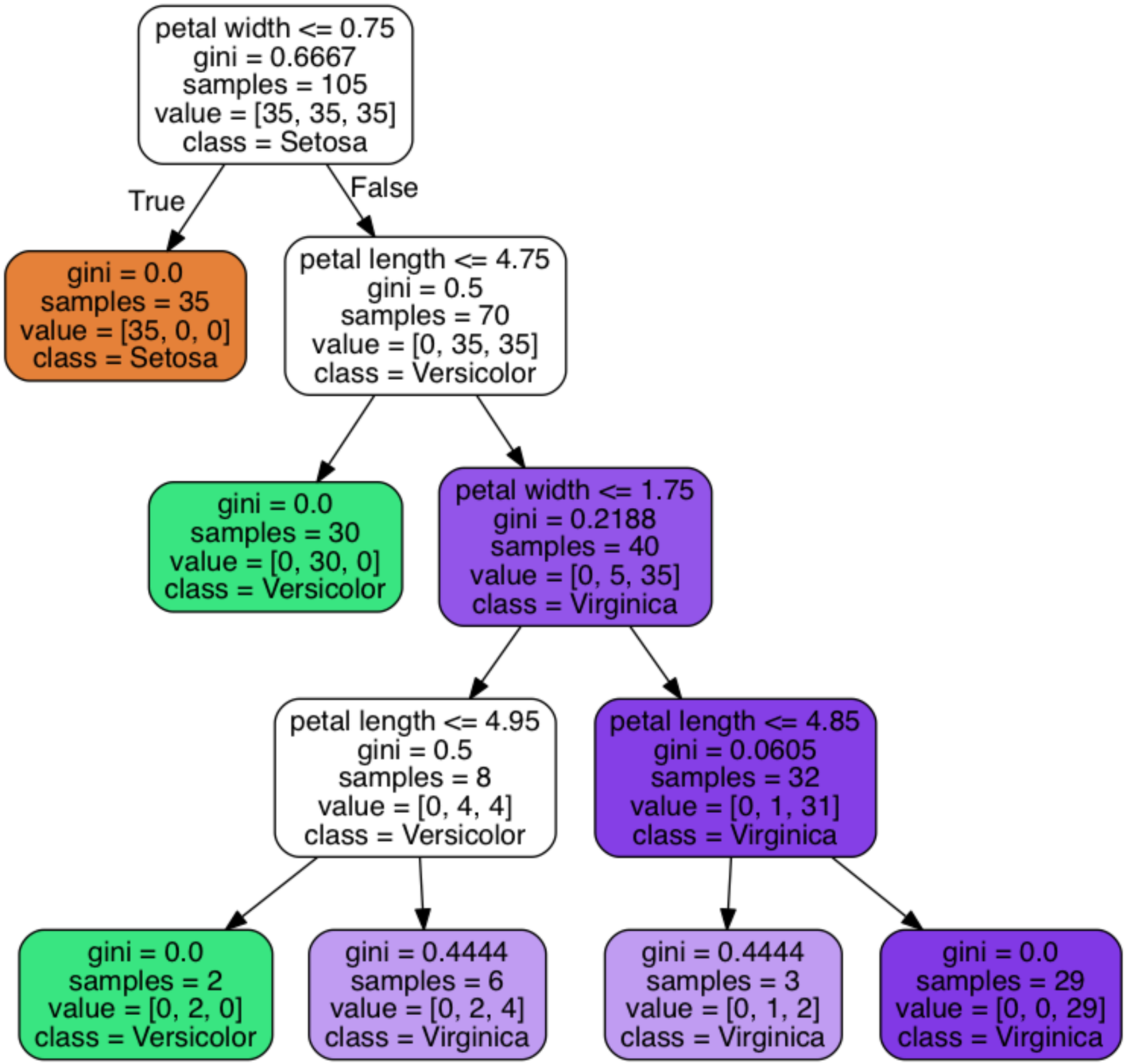
Note: the feature scaling is not required for decision tree algorithms

Decision Tree Built by Scikit-learn

```
from pydotplus import graph_from_dot_data
from sklearn.tree import export_graphviz

dot_data = export_graphviz(tree,
                           filled=True,
                           rounded=True,
                           class_names=['Setosa',
                                        'Versicolor',
                                        'Virginica'],
                           feature_names=['petal length',
                                          'petal width'],
                           out_file=None)
graph = graph_from_dot_data(dot_data)
graph.write_png('tree.png')
```

install **GraphViz** program and **pydotplus** first

Decision Tree Built by Scikit-learn

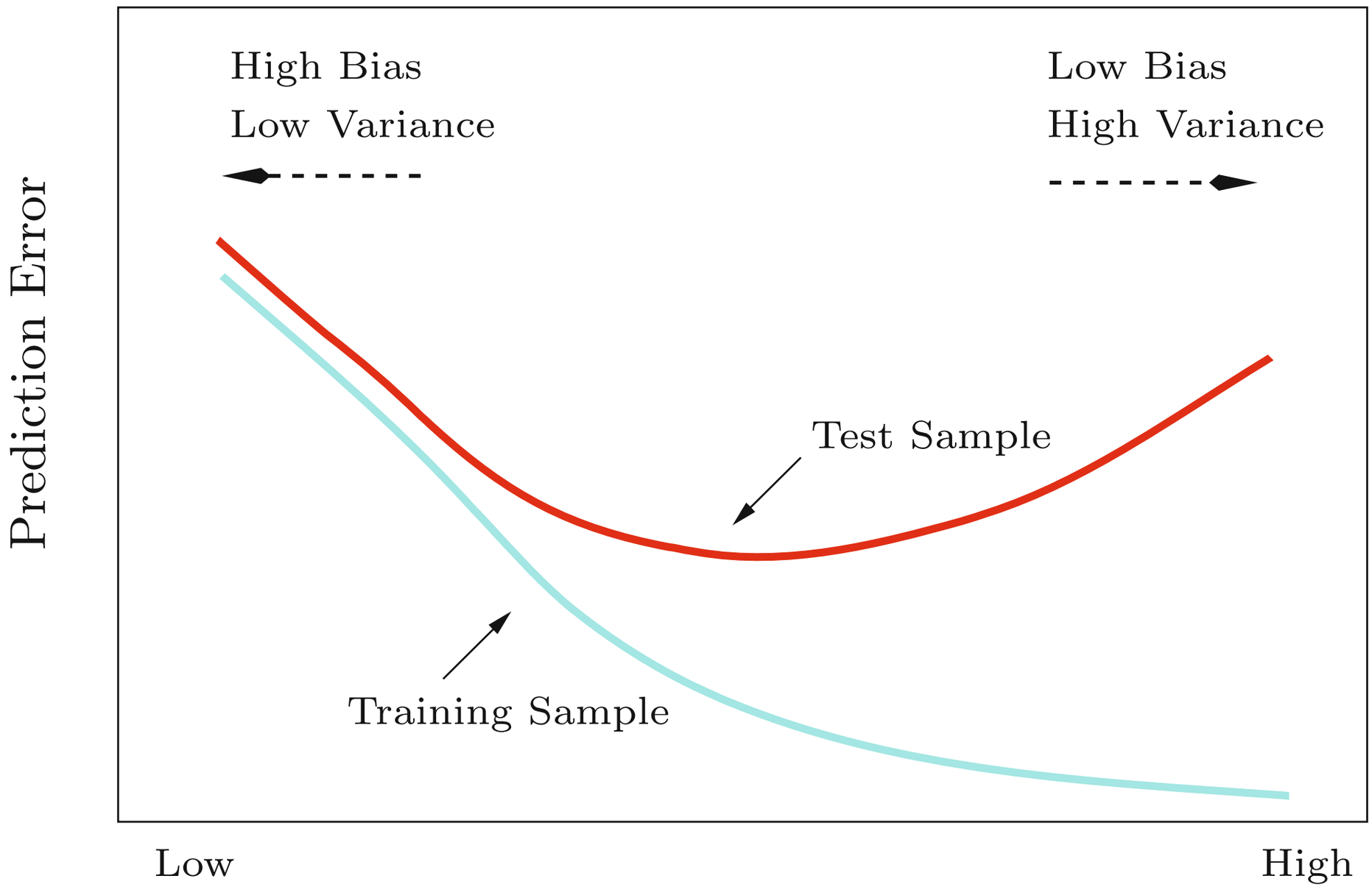


Random Forest

Random Forest

- A type of supervised learning based on ***ensemble learning***
  - Ensemble learning joins different types of algorithms or same algorithm multiple times to form a more powerful prediction model
- Random forest combines multiple decision trees, resulting in a forest of trees, hence the name.

Test and Training Error as a Function of Model Complexity



[Hastie, Tibshirani, Friedman "Elements of Statistical Learning" 2017]

Reduce Variance without Increasing Bias

- Averaging reduces variance

$$\mathrm{Var}(\bar{X}) = \frac{\mathrm{Var}(X)}{N}$$

- We can average models to reduce model variance
- Since we only have one training set, how to have multiple models?
- Ensemble

Bagging: Bootstrap Aggression

- Leo Breiman (1996)
- Take repeated bootstrap samples from training set D
- Bootstrap sampling: Given set D containing N training samples, create D′ by drawing N samples at random ***with replacement*** from D
- Bagging
  - Create k bootstrap samples D_1, D_2, …, D_k
  - Train distinct classifier on each D_i
  - Classify new instance by majority vote/average

Random Forest Algorithm

- Draw a random bootstrap sample of size ***n***
- Grow a decision tree from the bootstrap sample. At each node:
  - Randomly select *d* features *without replacement*
  - Split the node using the features that provide the best split according to the objective function, e.g., maximizing the IG
- Repeat the above 2 steps ***k*** times
- Aggregate the prediction by each tree to assign the class label by ***majority vote***.

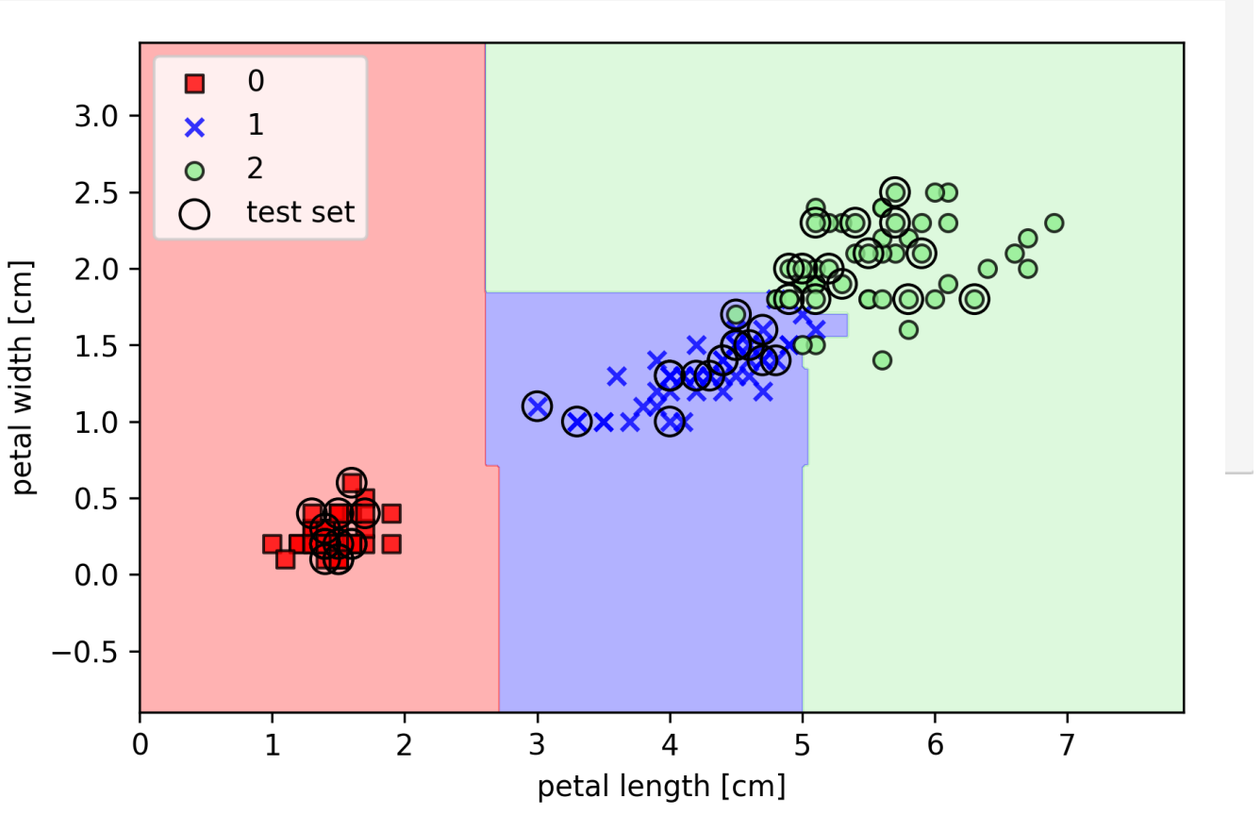
The *k*, *n*, *d* Parameters

- The larger ***k***, the better performance
  - Only the cost and time limit the number *k*
- In most implementation of random forests, including the **RandomForestClassifier** implementation of scikit-learn, ***n***=N, which usually provides a good boas-variance tradeoff
- A reasonable default of ***d*** is $\sqrt{m}$, where ***m*** is the number of features in the training dataset.
- The ***n*** and ***d*** can be optimized by techniques in Chapter 5.

Random Forest Classifier

```
from sklearn.ensemble import RandomForestClassifier

forest = RandomForestClassifier(criterion='gini',
                                n_estimators=25,
                                random_state=1,
                                n_jobs=2)
forest.fit(X_train, y_train)

plot_decision_regions(X_combined, y_combined,
                      classifier=forest, test_idx=range(105, 150))

plt.xlabel('petal length [cm]')
plt.ylabel('petal width [cm]')
plt.legend(loc='upper left')
plt.tight_layout()
#plt.savefig('images/03_22.png', dpi=300)
plt.show()
```

```
In [64]: y_pred = forest.predict(X_test)
         print('Misclassified instances: %d' % (y_test != y_pred).sum())

Misclassified instances: 1


In [65]: print('Accuracy: %.2f' % accuracy_score(y_test, y_pred))

Accuracy: 0.98


In [66]: print('Accuracy: %.2f' % forest.score(X_test, y_test))

Accuracy: 0.98
```



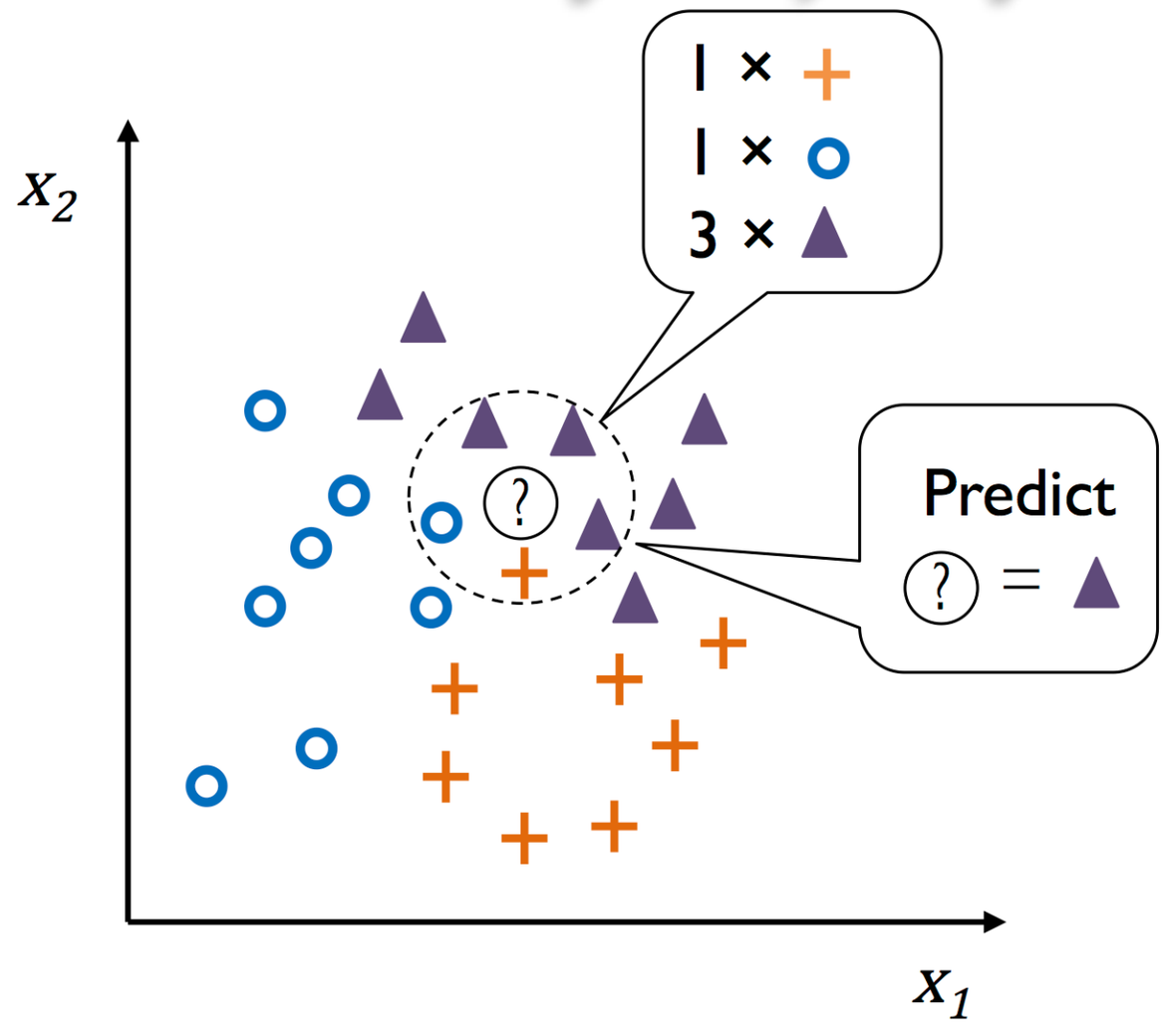
k-Nearest Neighbors - A Lazy Learning Algorithm

k-Nearest Neighbor Algorithms

- KNN ***does not learn*** any discriminative function from the training dataset, but ***memorizes*** the dataset instead
  - KNN is a non-parametric model
  - Parametric models learn parameters from training dataset
  - KNN is an instance-based learner

Basic KNN Algorithm

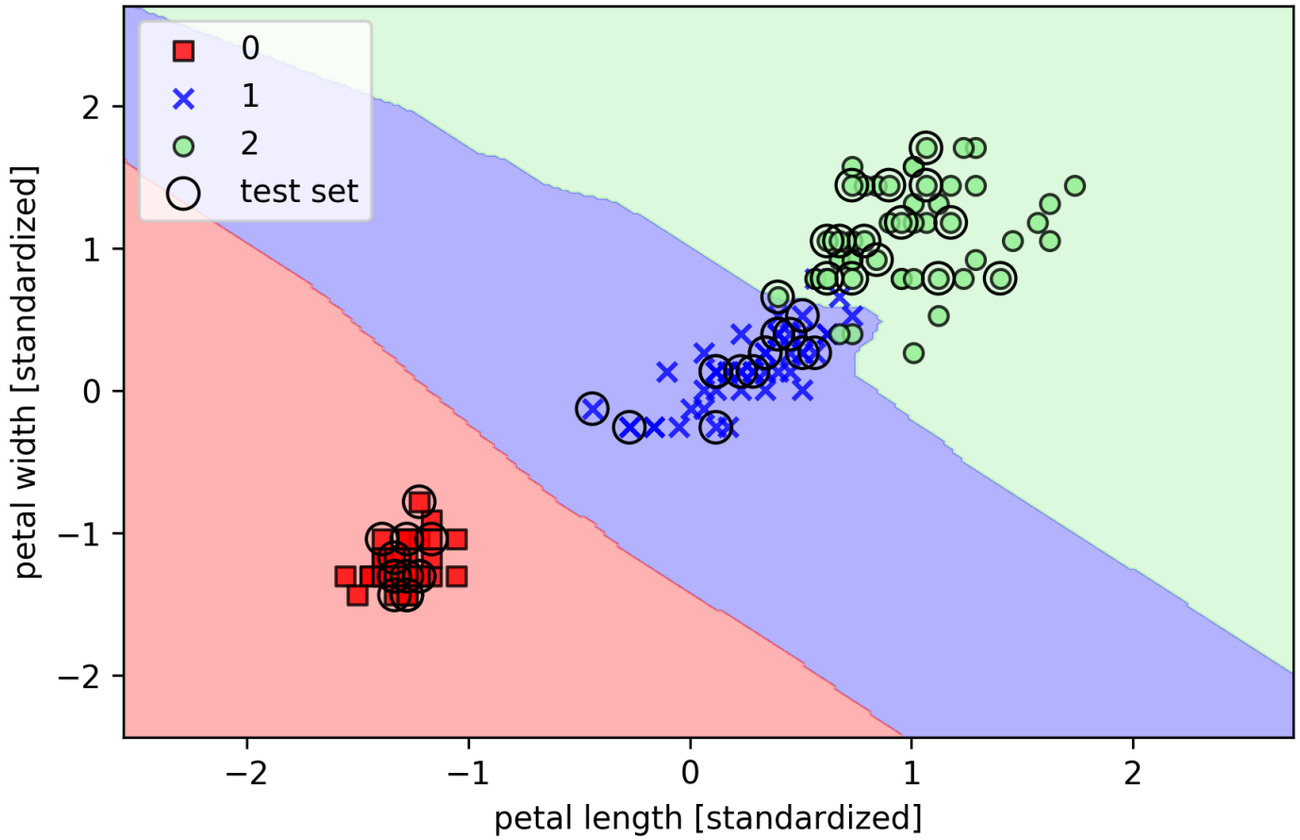
- Choose k and a distance metric
- Find the k-nearest neighbors of the sample that we want to classify
- Assign the class label by majority vote



KNN Classifier

```
from sklearn.neighbors import KNeighborsClassifier

knn = KNeighborsClassifier(n_neighbors=5,
                           p=2,
                           metric='minkowski')
knn.fit(X_train_std, y_train)

plot_decision_regions(X_combined_std, y_combined,
                      classifier=knn, test_idx=range(105, 150))

plt.xlabel('petal length [standardized]')
plt.ylabel('petal width [standardized]')
plt.legend(loc='upper left')
plt.tight_layout()
#plt.savefig('images/03_24.png', dpi=300)
plt.show()
```

```
In [69]: y_pred = knn.predict(X_test_std)
         print('Misclassified instances: %d' % (y_test != y_pred).sum())

Misclassified instances: 0


In [70]: print('Accuracy: %.2f' % accuracy_score(y_test, y_pred))

Accuracy: 1.00


In [71]: print('Accuracy: %.2f' % knn.score(X_test_std, y_test))

Accuracy: 1.00
```



KNN Classifier

- Features of KNN
  - Classifier immediately adapts as we receive new training data
  - Computational complexity grows linearly with the number of samples
  - Need efficient data structures such as KD-trees
- Distance metrics
  - p=2: Euclidean distance
  - p=1: Manhattan distance

$$d\left(\boldsymbol{x}^{(i)}, \boldsymbol{x}^{(j)}\right) = \sqrt[p]{\sum_k \left| x_k^{(i)} - x_k^{(j)} \right|^p}$$