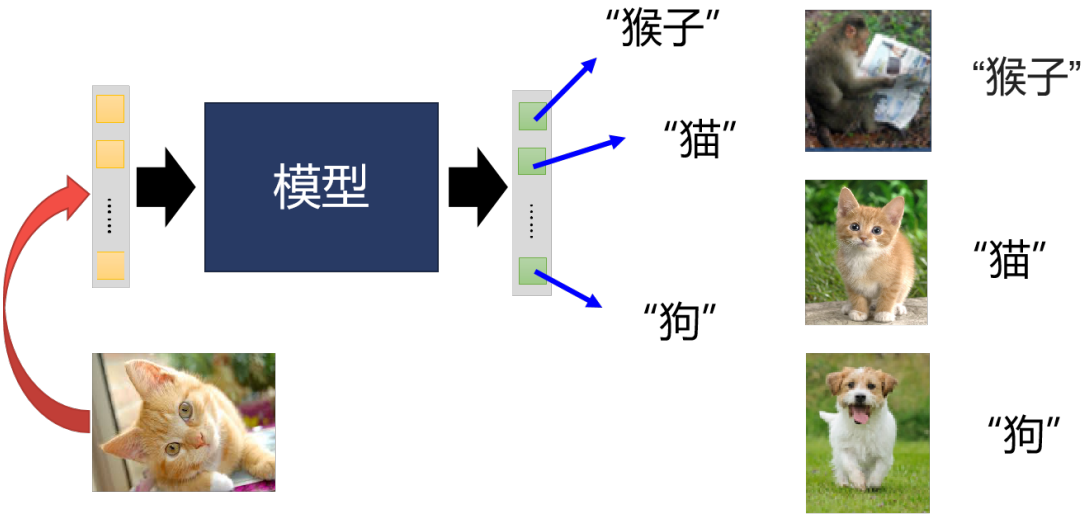
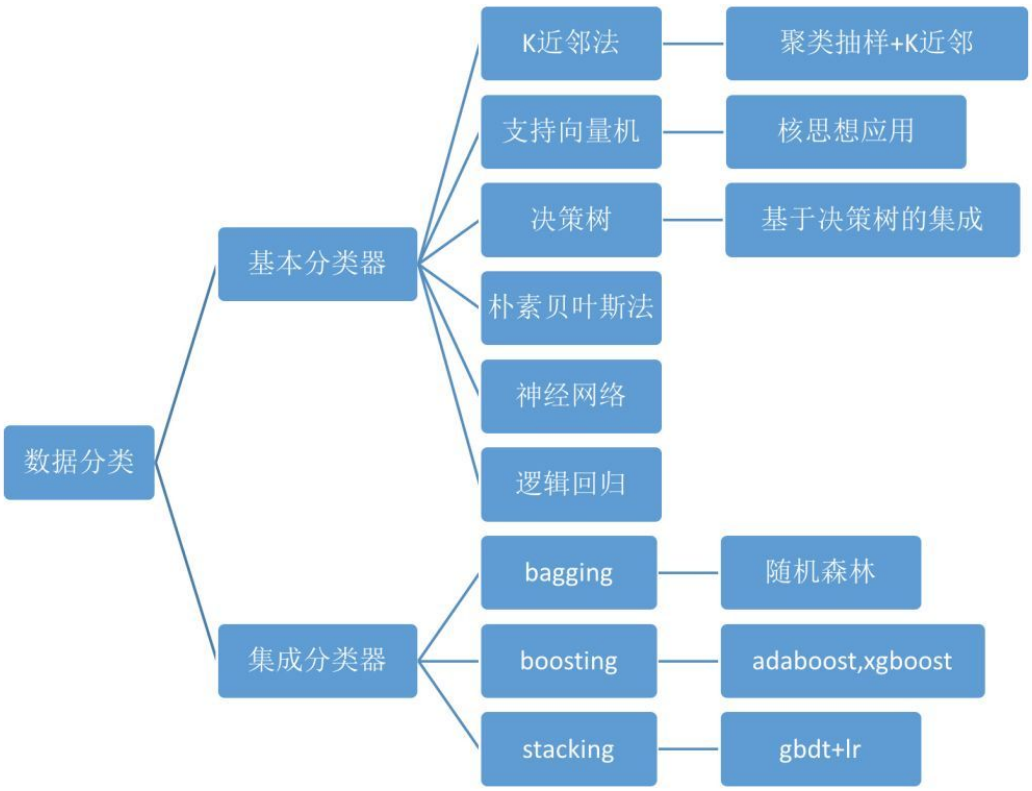
分类

2020 年 3 月 25 日

目录

1 分类算法





2 数据集的线性可分性

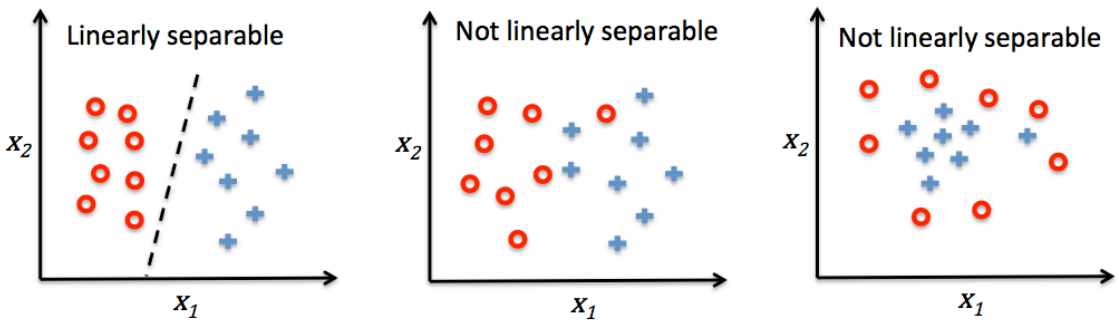
给定一个数据集 $D = \{(\mathbf{x}^{(1)}, y^{(1)}), (\mathbf{x}^{(2)}, y^{(2)}), \cdots, (\mathbf{x}^{(n)}, y^{(n)})\}$，其中 $\mathbf{x}^{(i)} = (x_1^{(i)}, x_1^{(i)}, \cdots, x_m^{(i)}), y^{(i)} \in \{-1, +1\}$，如果存在一个正的常数 γ 和权重向量 w，对所有的 i 满足

$$y^{(i)} \cdot \left(w \cdot \mathbf{x}^{(i)} + b \right) > \gamma$$

训练集 D 就被叫做**线性可分**, 也就是可以找到一个超平面 S

$$w \cdot \mathbf{x}^{(i)} + b = 0$$

能够将数据集的正例和负例完全正确的划分到超平面的两侧。否则称训练集 D **线性不可分**。



问题：线性可分问题解是否唯一？

3 感知机

感知器（Perceptron），是神经网络中的一个概念，在 1950s 由 Frank Rosenblatt 第一次引入。感知器是形式最简单的前馈神经网络，是一种二元线性分类器, 把输入 x （实数值向量）映射到输出值 $f(x)$ 上。

$$f(x) = \begin{cases} +1 & \text{if } w \cdot x + b > 0 \\ -1 & \text{else} \end{cases}$$

3.1 学习算法

我们首先定义一些变量：- x_j 表示 n 维输入向量中的第 j 项 - w_j 表示权重向量的第 j 项 - $f(x)$ 表示神经元接受输入 x 产生的输出 - 更进一步，为了简便我们可以令 w_0 等于 b，x_0 等于 1。

感知器的学习通过对**所有训练实例**进行**多次迭代更新**的方式来建模。

令 $D_m = \{(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)})\}$ 表示一个有 m 个训练实例的训练集。

我们损失函数的优化目标，就是期望使误分类的所有样本，到超平面的距离之和最小。所以损失函数定义如下：

$$L(w, b) = -\frac{1}{||w||} \sum_{x^{(i)} \in M} y^{(i)}(w \cdot x^{(i)} + b)$$

其中 M 集合是误分类点的集合。

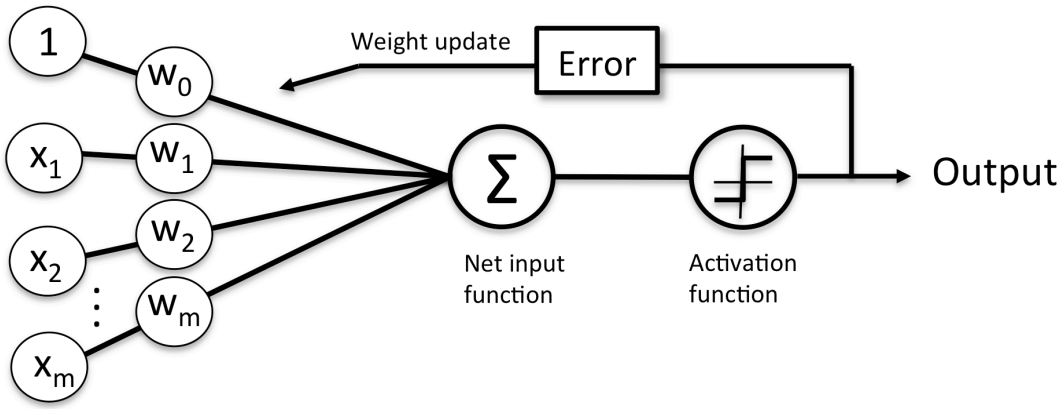
采用随机梯度下降法，每次迭代权重向量以如下方式**更新**：对于 $D_m = \{(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)})\}$ 中的每个 (x, y) ，

$$w_j := w_j + \eta(y - f(x))x_j \quad (j = 1, \ldots, n)$$

其中 η 是学习率，在 0 到 1 之间取值。

注意这意味着，仅当针对给定训练实例 (x, y) 产生的输出值 $f(x)$ 与预期的输出值 y 不同时，权重向量才会发生改变。

注意，如果训练集不是线性可分的，那么这个算法则不能保证会收敛。



3.2 利用 Python 实现感知器学习算法

[1]:
```
import numpy as np


class Perceptron(object):
    """Perceptron classifier.

    Parameters
    ------------
    eta : float
        学习率 (between 0.0 and 1.0)
    n_iter : int
        迭代次数

    属性
    -----------
    w_ : 1d-array
        拟合后的权重
    errors_ : list
        每轮的错误分类的数量

    """
    def __init__(self, eta=0.01, n_iter=10):
        self.eta = eta
```

```
        self.n_iter = n_iter  # 轮数

    def fit(self, X, y):
        """Fit training data.

        Parameters
        ----------
        X : {array-like}, shape = [n_samples, n_features]
            训练数据
        y : array-like, shape = [n_samples]
            目标向量

        Returns
        -------
        self : object

        """
        # 权重，初始值设置为 0，在后面的迭代过程中，会不断更新
        self.w_ = np.zeros(1 + X.shape[1])
        self.errors_ = []

        # 对每个 sample 循环更新
        for _ in range(self.n_iter):
            errors = 0
            for xi, target in zip(X, y):
                #(learning rate)*(error)
                update = self.eta * (target - self.predict(xi))
                self.w_[1:] += update * xi
                self.w_[0] += update
                errors += int(update != 0.0)
            self.errors_.append(errors)  # 错误的分类结果
        return self

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

```
    def predict(self, X):
        """Return class label after unit step"""
        return np.where(self.net_input(X) >= 0.0, 1, -1)
```

3.3 在 Iris 数据集上训练一个感知器

Iris 鸢尾花数据集是一个经典数据集。数据集内包含 3 类共 150 条记录，每类各 50 个数据，每条记录都有 4 项特征：花萼长度、花萼宽度、花瓣长度、花瓣宽度，可以通过这 4 个特征预测鸢尾花卉属于（iris-setosa, iris-versicolour, iris-virginica）中的哪一品种。

Sepal（花萼），花的最外一轮叶状构造称为花萼。花萼通常为绿色，可大可小，包在花蕾外面，起保护花蕾的作用。

3.4 读入数据

[2]:
```
import pandas as pd

df = pd.read_csv('data/iris.csv')
df.head()
```

[2]:

	Sepal_Length	Sepal_Width	Petal_Length	Petal_Width	Species
0	5.1	3.5	1.4	0.2	setosa
1	4.9	3.0	1.4	0.2	setosa
2	4.7	3.2	1.3	0.2	setosa
3	4.6	3.1	1.5	0.2	setosa
4	5.0	3.6	1.4	0.2	setosa

[3]:
```
df.tail()
```

[3]:

	Sepal_Length	Sepal_Width	Petal_Length	Petal_Width	Species
145	6.7	3.0	5.2	2.3	virginica
146	6.3	2.5	5.0	1.9	virginica
147	6.5	3.0	5.2	2.0	virginica
148	6.2	3.4	5.4	2.3	virginica
149	5.9	3.0	5.1	1.8	virginica

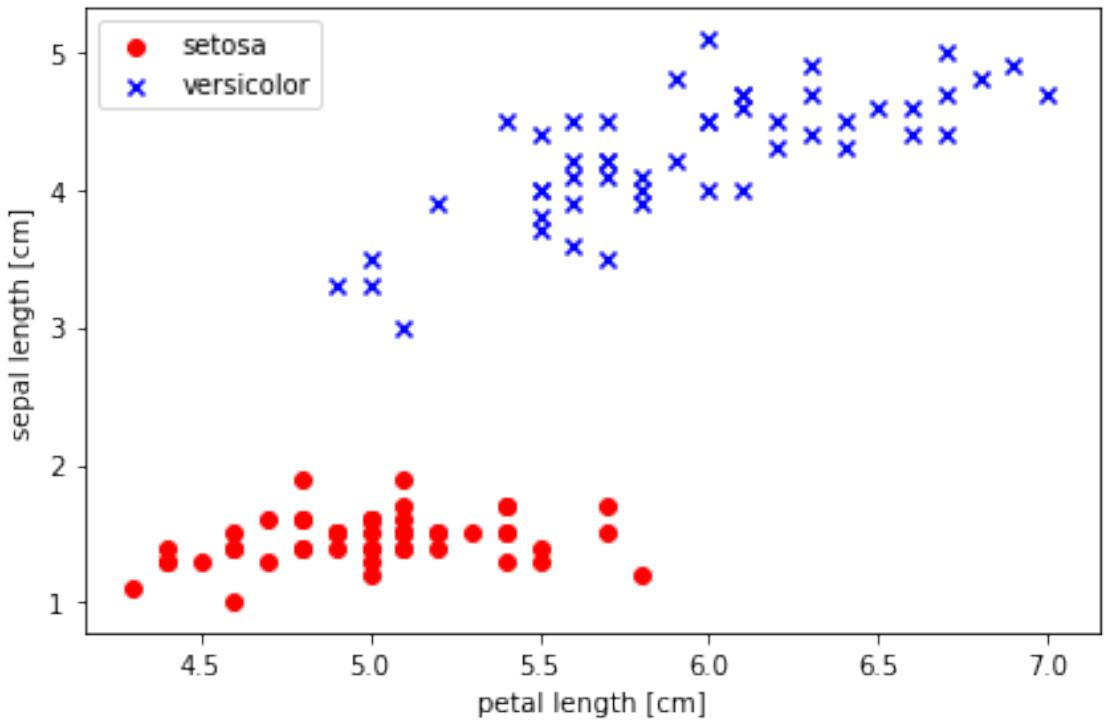
[4]:
```
df.describe()
```

[4]:

	Sepal_Length	Sepal_Width	Petal_Length	Petal_Width
count	150.000000	150.000000	150.000000	150.000000
mean	5.843333	3.054000	3.758667	1.198667
std	0.828066	0.433594	1.764420	0.763161
min	4.300000	2.000000	1.000000	0.100000
25%	5.100000	2.800000	1.600000	0.300000
50%	5.800000	3.000000	4.350000	1.300000
75%	6.400000	3.300000	5.100000	1.800000
max	7.900000	4.400000	6.900000	2.500000

3.5 数据观察

[5]:
```
# 将两个分类先可视化
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np

# select setosa and versicolor
# 两类别各 50 条数据，把类别改为 -1 和 1，方便画图
y = df.iloc[0:100, 4].values
y = np.where(y == 'setosa', -1, 1)

# 提取 sepal length 和 petal length
X = df.iloc[0:100, [0, 2]].values

# 绘制图形
plt.scatter(X[:50, 0], X[:50, 1],
            color='red', marker='o', label='setosa')
plt.scatter(X[50:100, 0], X[50:100, 1],
            color='blue', marker='x', label='versicolor')

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

plt.tight_layout()
# plt.savefig('./iris_1.png', dpi=300)
```

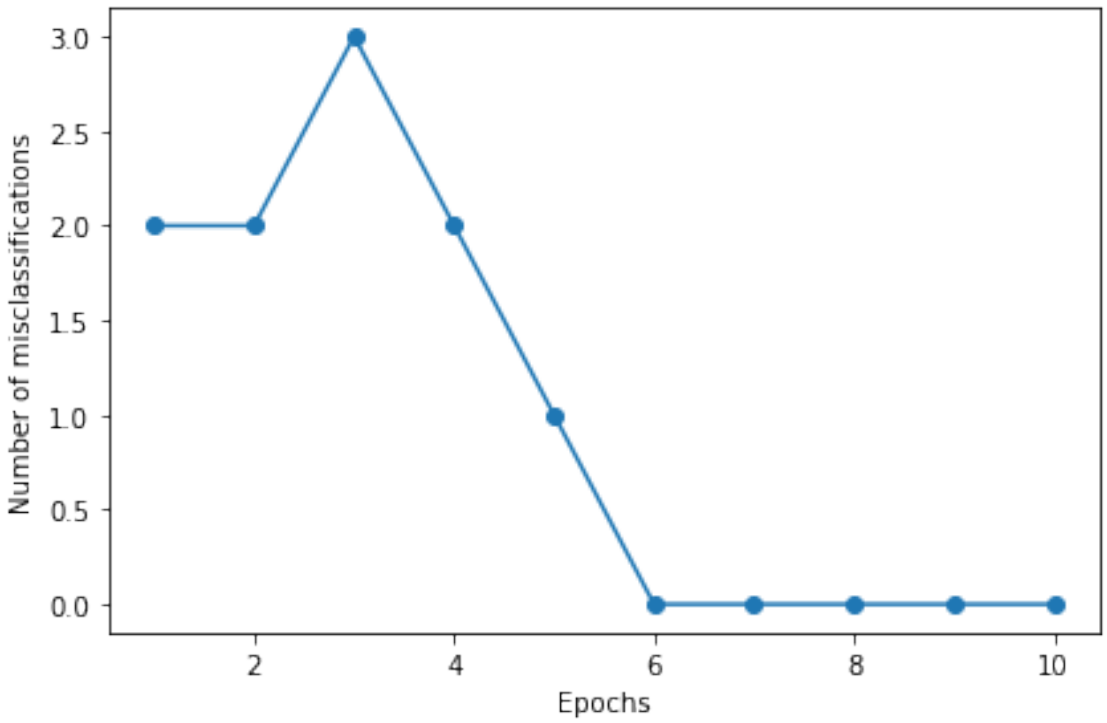


3.6 训练感知器模型

[6]:
```
ppn = Perceptron(eta=0.1, n_iter=10)
ppn.fit(X, y)
ppn.errors_
```

[6]:
```
[2, 2, 3, 2, 1, 0, 0, 0, 0, 0]
```

[7]:
```
# 绘制 error 的图形，在几轮更新之后，检查 error 是否趋近于 0
plt.plot(range(1, len(ppn.errors_) + 1), ppn.errors_, marker='o')
plt.xlabel('Epochs')
plt.ylabel('Number of misclassifications')

plt.tight_layout()
# plt.savefig('./perceptron_1.png', dpi=300)
```



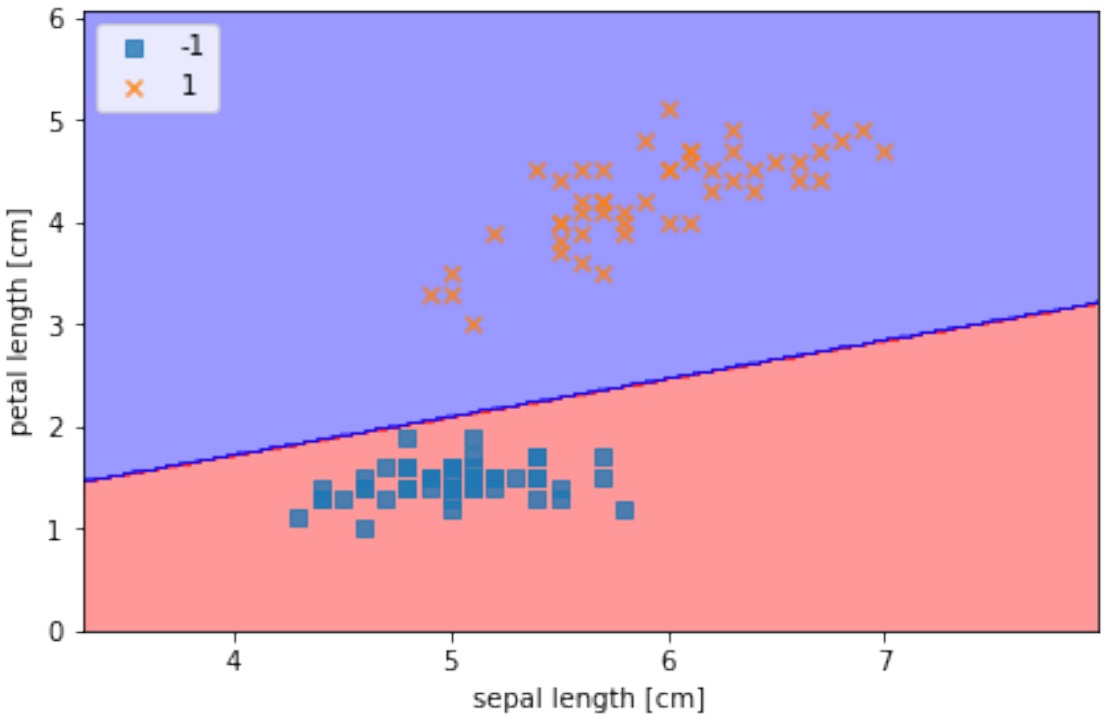
通过图形发现，结果 error 的确最后为 0, 并且收敛。

3.7 编写一个函数用于绘制决策边界

[8]:
```
from matplotlib.colors import ListedColormap
# Colormap object generated from a list of colors.

def plot_decision_regions(X, y, classifier, resolution=0.02):

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

    # 确定横纵轴边界
    x1_min, x1_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    x2_min, x2_max = X[:, 1].min() - 1, X[:, 1].max() + 1
```

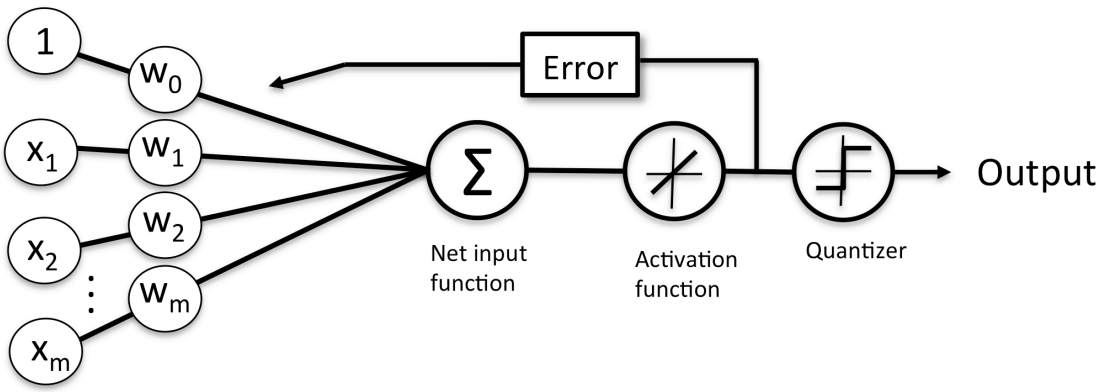
```
    # 建立一对 grid arrays
    # 铺平 grid arrays, 然后进行预测
    xx1, xx2 = np.meshgrid(np.arange(x1_min, x1_max, resolution),
                           np.arange(x2_min, x2_max, resolution))

    #ravel() Return a contiguous flattened array.
    Z = classifier.predict(np.array([xx1.ravel(), xx2.ravel()]).T)

    # 把分类结果分配到每个点上
    Z = Z.reshape(xx1.shape)

    # 将不同的决策边界对应不同的颜色
    plt.contourf(xx1, xx2, Z, alpha=0.4, cmap=cmap)
    plt.xlim(xx1.min(), xx1.max())
    plt.ylim(xx2.min(), xx2.max())

    # 绘制样本点
    for idx, cl in enumerate(np.unique(y)):
        plt.scatter(x=X[y == cl, 0], y=X[y == cl, 1],
                    #alpha=0.8, c=cmap(idx),
                    alpha=0.8,
                    marker=markers[idx], label=cl)
```

[9]:
```
plot_decision_regions(X, y, classifier=ppn)
plt.xlabel('sepal length [cm]')
plt.ylabel('petal length [cm]')
plt.legend(loc='upper left')

plt.tight_layout()
# plt.savefig('./perceptron_2.png', dpi=300)
```



虽然 Perceptron Model 在上面 Iris 例子里表现得很好，但在其他问题上却不一定表现得好。Frank Rosenblatt 从数学上证明了，在线性可分的数据里，Perceptron 的学习规则会收敛，但在线性不可分的情况下，却无法收敛。

4 自适应线性神经网络 Adaptive linear neurons(Adaline)



4.1 对感知器的改进

ADAptive LInear NEuron classifier 也是一个单层神经网络. 它的重点就是定义及最优化损失函数, 对于理解更高层次更难的机器学习分类模型是非常好的入门.

它与感知器不同的地方在于**更新 weights** 时用的是线性激活函数, 而不是阶梯函数.

Adaline 中这个线性激活函数输出等于输入, $\phi(w^T x) = w^T x$.

令

$$z =$$

定义损失函数为 SSE: Sum of Squared Errors

$$J(w) = \frac{1}{2}\sum_i (y^{(i)} - \phi(z^{(i)}))^2$$

这个损失函数是可导的, 并且是凸的, 可以使用梯度下降算法进行最优化.

梯度

$$\frac{\partial J}{\partial w_j} = -\sum_{i=1}^{n}(y^{(i)} - \phi(z^{(i)}))x_j^{(i)}$$

量化器采用符号函数对标签进行预测。

$$f(x) = \begin{cases} +1 & \text{if } w \cdot x + b \geq 0 \\ -1 & \text{otherwise} \end{cases}$$

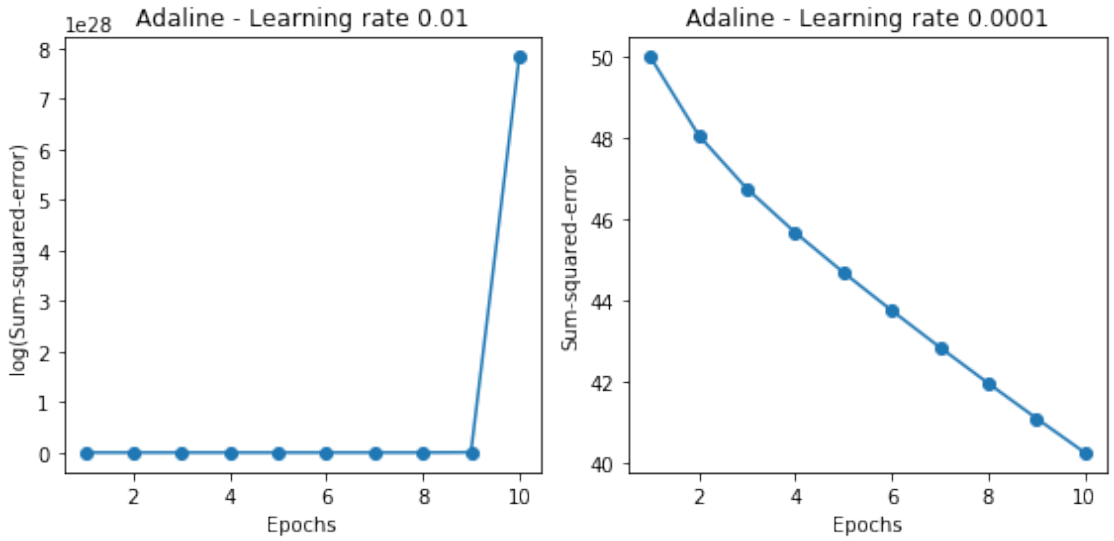
4.2 利用 Python 实现自适应线性神经网络

[10]:
```
class AdalineGD(object):
    """ADAptive LInear NEuron classifier.

    Parameters
    ------------
    eta : float
        学习率 (between 0.0 and 1.0)
    n_iter : int
        迭代次数

    Attributes
    -----------
    w_ : 1d-array
```

```
    拟合后的权重.
errors_ : list
    每轮的错误分类的数量.

"""
def __init__(self, eta=0.01, n_iter=50):
    self.eta = eta
    self.n_iter = n_iter

def fit(self, X, y):
    """ Fit training data.

    Parameters
    ----------
    X : {array-like}, shape = [n_samples, n_features]
        Training vectors, where n_samples is the number of samples and
        n_features is the number of features.
    y : array-like, shape = [n_samples]
        Target values.

    Returns
    -------
    self : object

    """
    self.w_ = np.zeros(1 + X.shape[1])
    self.cost_ = []

    # gradient descent-梯度下降
    for i in range(self.n_iter):
        output = self.net_input(X)
        errors = (y - output)
        self.w_[1:] += self.eta * X.T.dot(errors)
        self.w_[0] += self.eta * errors.sum()
        cost = (errors**2).sum() / 2.0
        self.cost_.append(cost) # cost list, 验证收敛性
```

```
        return self

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

    def activation(self, X):
        """Compute linear activation"""
        return self.net_input(X)

    def predict(self, X):
        """Return class label after unit step"""
        return np.where(self.activation(X) >= 0.0, 1, -1)
```

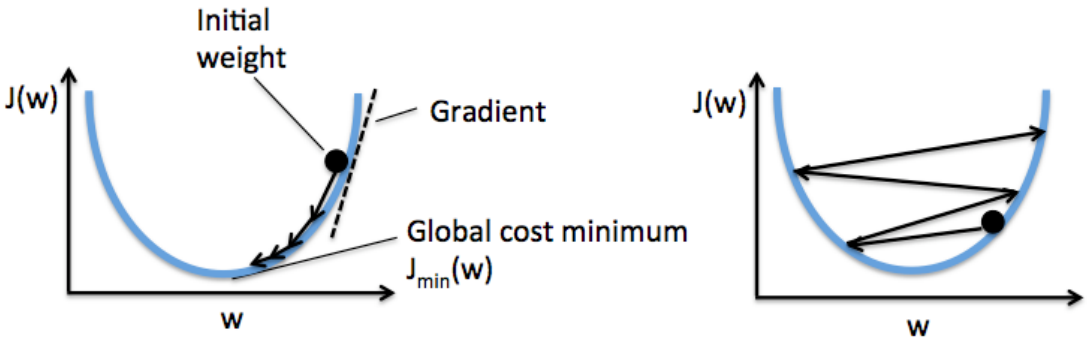
[11]:
```
# 测试两种 learning rate, 0.01 和 0.0001
fig, ax = plt.subplots(nrows=1, ncols=2, figsize=(8, 4))

ada1 = AdalineGD(n_iter=10, eta=0.01).fit(X, y)
ax[0].plot(range(1, len(ada1.cost_) + 1), ada1.cost_, marker='o')
# 误差 blow up, 因此加上 log 进行显示
#ax[0].plot(range(1, len(ada1.cost_) + 1), np.log10(ada1.cost_), marker='o')
ax[0].set_xlabel('Epochs')
ax[0].set_ylabel('log(Sum-squared-error)')
ax[0].set_title('Adaline - Learning rate 0.01')

ada2 = AdalineGD(n_iter=10, eta=0.0001).fit(X, y)
ax[1].plot(range(1, len(ada2.cost_) + 1), ada2.cost_, marker='o')
ax[1].set_xlabel('Epochs')
ax[1].set_ylabel('Sum-squared-error')
ax[1].set_title('Adaline - Learning rate 0.0001')

plt.tight_layout()
# plt.savefig('./adaline_1.png', dpi=300)
```



左图显示 learning rate 太大, error 没有变小, 反而变大了.

右图显示 learning rate 太小, error 变化速度太小

思考：原因？



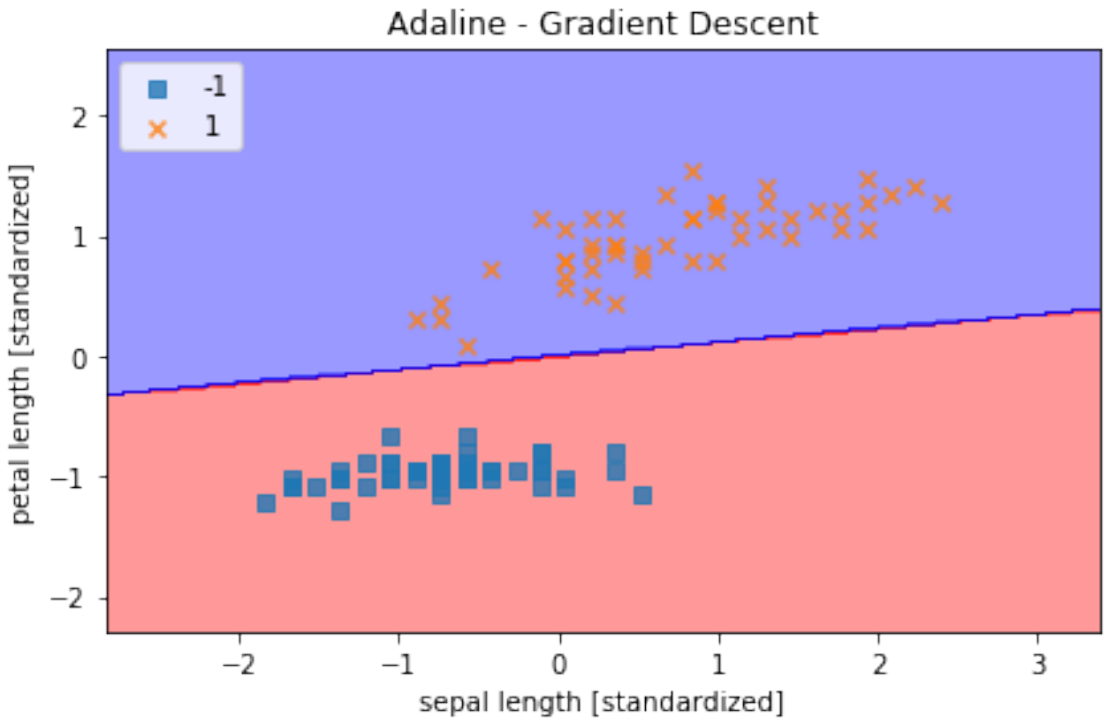
4.3 特征标准化之后再次进行训练

$$x'_j = \frac{x_j - \mu_j}{\sigma_j}$$

[12]:
```
# 特征标准化
X_std = np.copy(X)
```

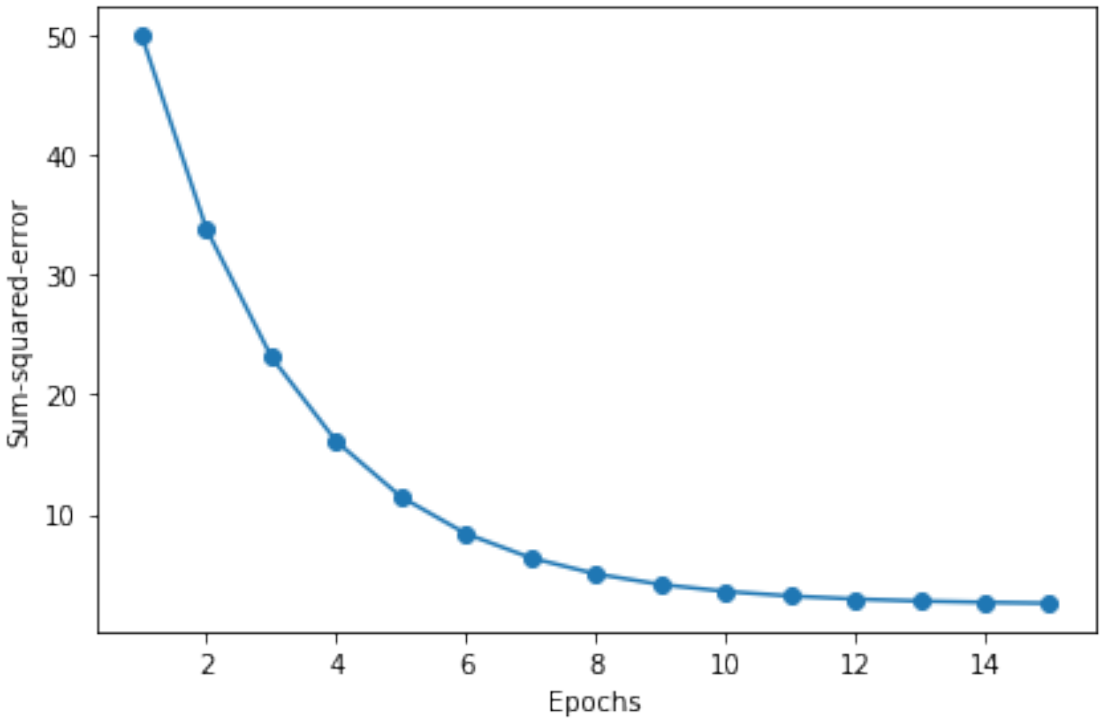
```
X_std[:,0] = (X[:,0] - X[:,0].mean()) / X[:,0].std()
X_std[:,1] = (X[:,1] - X[:,1].mean()) / X[:,1].std()
```

[13]:
```
ada = AdalineGD(n_iter=15, eta=0.01)
ada.fit(X_std, y)

plot_decision_regions(X_std, y, classifier=ada)
plt.title('Adaline - Gradient Descent')
plt.xlabel('sepal length [standardized]')
plt.ylabel('petal length [standardized]')
plt.legend(loc='upper left')
plt.tight_layout()
#plt.savefig('./adaline_2.png', dpi=300)
```



[14]:
```
plt.plot(range(1, len(ada.cost_) + 1), ada.cost_, marker='o')
plt.xlabel('Epochs')
plt.ylabel('Sum-squared-error')
```

```
plt.tight_layout()
```



分类效果不错, error 最终接近于 0. 值得注意的是，虽然我们的分类全部正确，但 error 也不等于 0.

[15]:
```
ada.w_ # 权重
```

[15]:
```
array([ 1.36557432e-16, -1.26256159e-01,  1.10479201e+00])
```

4.4 大数据量的机器学习与随机梯度下降（stochastic gradient descent）

Stochastic gradient descent 随机梯度下降 比一般的梯度下降更有优势, 因为每一步计算的 cost 更小, 每一步更新都是取其中一个数据点进行更新.

batch gradient descent 一次更新需要计算一遍整个数据集

$$\Delta w = \eta \sum_i (y^{(i)} - \phi(z^{(i)}))x^{(i)}$$

stochastic gradient descent 一次更新只需计算一个数据点

$$\Delta w = \eta(y^{(i)} - \phi(z^{(i)}))x^{(i)}$$

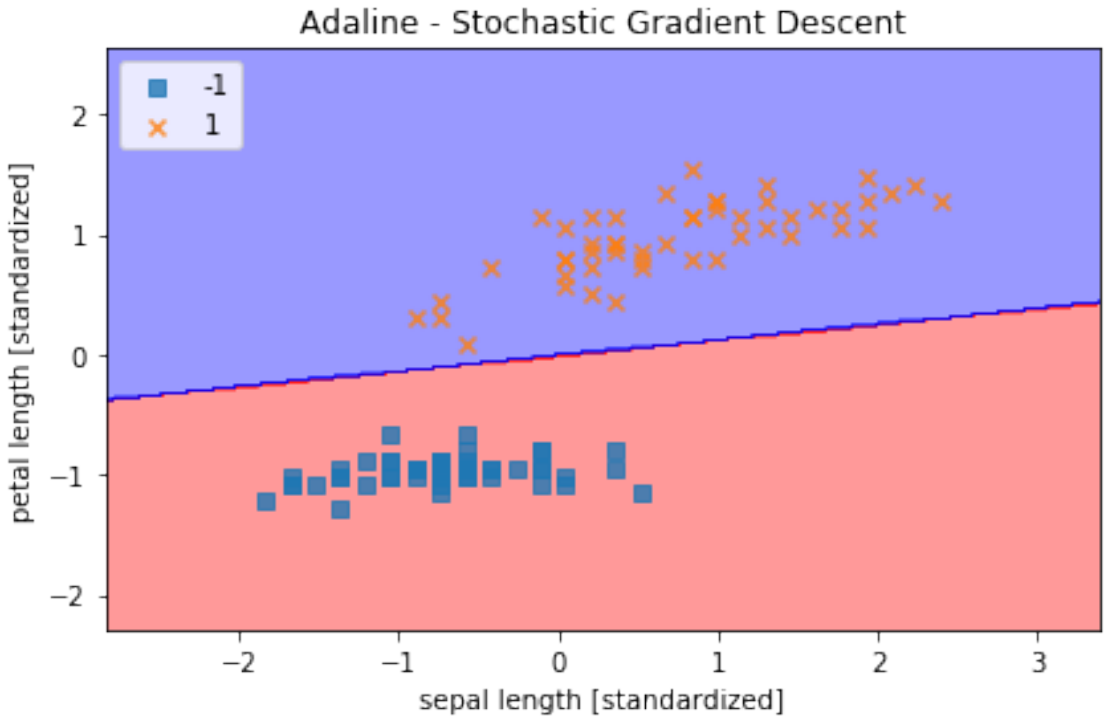
[16]:
```
class AdalineSGD(object):
    """ADAptive LInear NEuron classifier.

    Parameters
    ------------
    eta : float
        Learning rate (between 0.0 and 1.0)
    n_iter : int
        Passes over the training dataset.

    Attributes
    -----------
    w_ : 1d-array
        Weights after fitting.
    errors_ : list
        Number of misclassifications in every epoch.
    shuffle : bool (default: True)
        Shuffles training data every epoch if True to prevent cycles.
    random_state : int (default: None)
        Set random state for shuffling and initializing the weights.

    """
    def __init__(self, eta=0.01, n_iter=10, shuffle=True, random_state=None):
        self.eta = eta
        self.n_iter = n_iter
        self.w_initialized = False
        self.shuffle = shuffle
        if random_state:  # allow the specication of a random seed for␣
↪consistency
            np.random.seed(random_state)

    def fit(self, X, y):
        """ Fit training data.
```

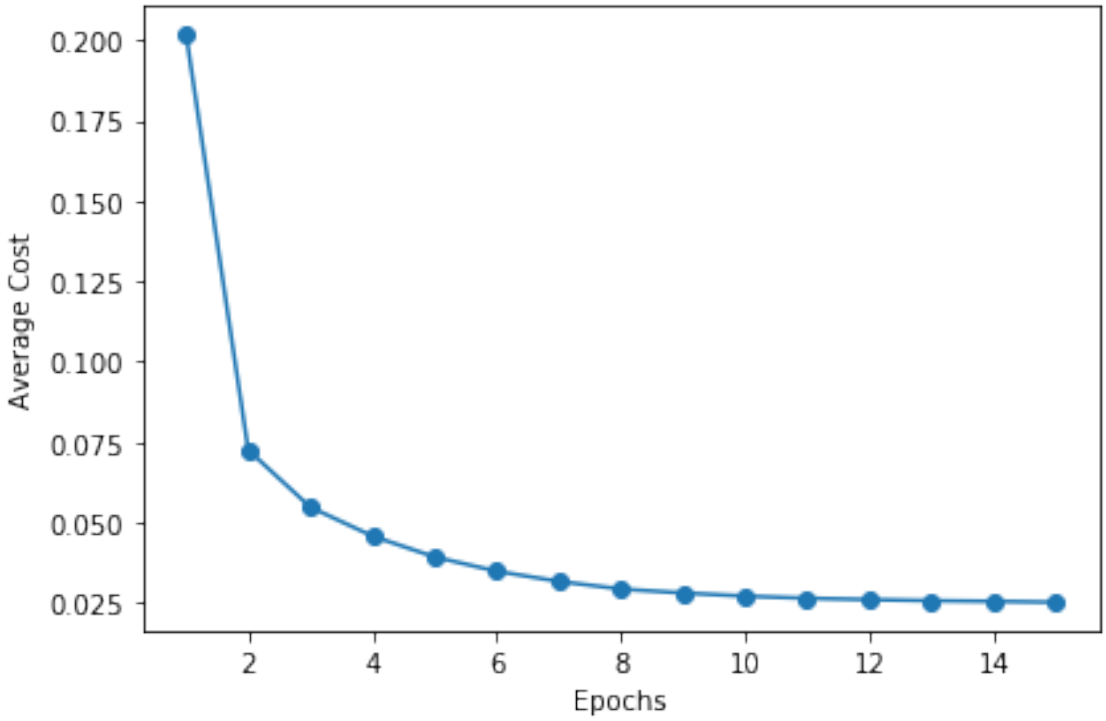
```
    Parameters
    ----------
    X : {array-like}, shape = [n_samples, n_features]
        Training vectors, where n_samples is the number of samples and
        n_features is the number of features.
    y : array-like, shape = [n_samples]
        Target values.

    Returns
    -------
    self : object

    """
    self._initialize_weights(X.shape[1])
    self.cost_ = []
    for i in range(self.n_iter):
        if self.shuffle:
            X, y = self._shuffle(X, y)
        cost = []
        for xi, target in zip(X, y): # 多了一层循环
            cost.append(self._update_weights(xi, target))
        avg_cost = sum(cost) / len(y)
        self.cost_.append(avg_cost)
    return self


# 在每个 epoch （遍历整个 training data）前是否 shuffle data
def _shuffle(self, X, y):
    """Shuffle training data"""
    r = np.random.permutation(len(y))
    return X[r], y[r]

def _initialize_weights(self, m):
    """Initialize weights to zeros"""
    self.w_ = np.zeros(1 + m)
```

```
        self.w_initialized = True

    #stochastic gradient descent
    def _update_weights(self, xi, target):
        """Apply Adaline learning rule to update the weights"""
        output = self.net_input(xi)
        error = (target - output)
        self.w_[1:] += self.eta * xi.dot(error) # 仅一个 error 相乘
        self.w_[0] += self.eta * error # 仅仅是一个 error, 而非 sum
        cost = 0.5 * error**2
        return cost

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

    def activation(self, X):
        """Compute linear activation"""
        return self.net_input(X)

    def predict(self, X):
        """Return class label after unit step"""
        return np.where(self.activation(X) >= 0.0, 1, -1)
```

[17]:
```
# plot result
ada = AdalineSGD(n_iter=15, eta=0.01, random_state=1)
ada.fit(X_std, y)

plot_decision_regions(X_std, y, classifier=ada)
plt.title('Adaline - Stochastic Gradient Descent')
plt.xlabel('sepal length [standardized]')
plt.ylabel('petal length [standardized]')
plt.legend(loc='upper left')

plt.tight_layout()
#plt.savefig('./adaline_4.png', dpi=300)
```

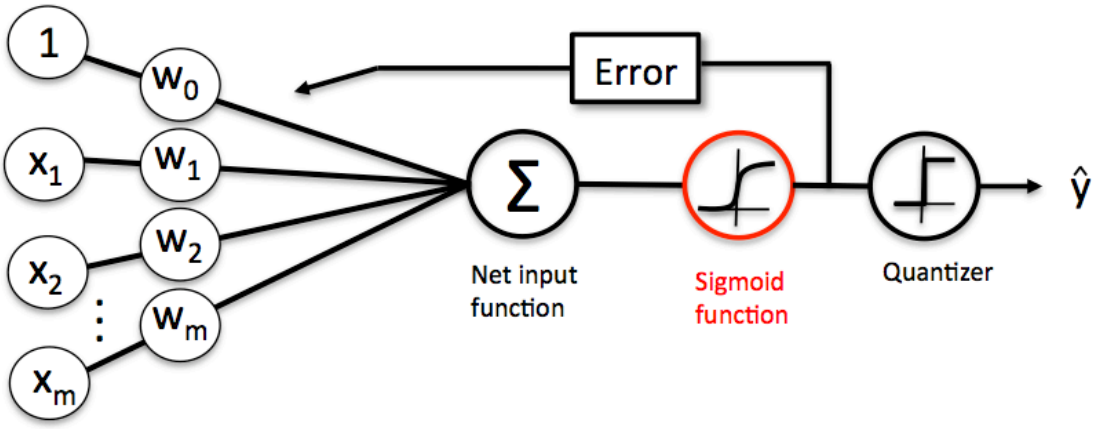
```
plt.show()

plt.plot(range(1, len(ada.cost_) + 1), ada.cost_, marker='o')
plt.xlabel('Epochs')
plt.ylabel('Average Cost')

plt.tight_layout()
# plt.savefig('./adaline_5.png', dpi=300)
```





实际中，采用折中的办法：mini-batch learning.

5 利用 Python 实现 logistic regression



logistic regression(逻辑回归)：二分类任务的有力算法。

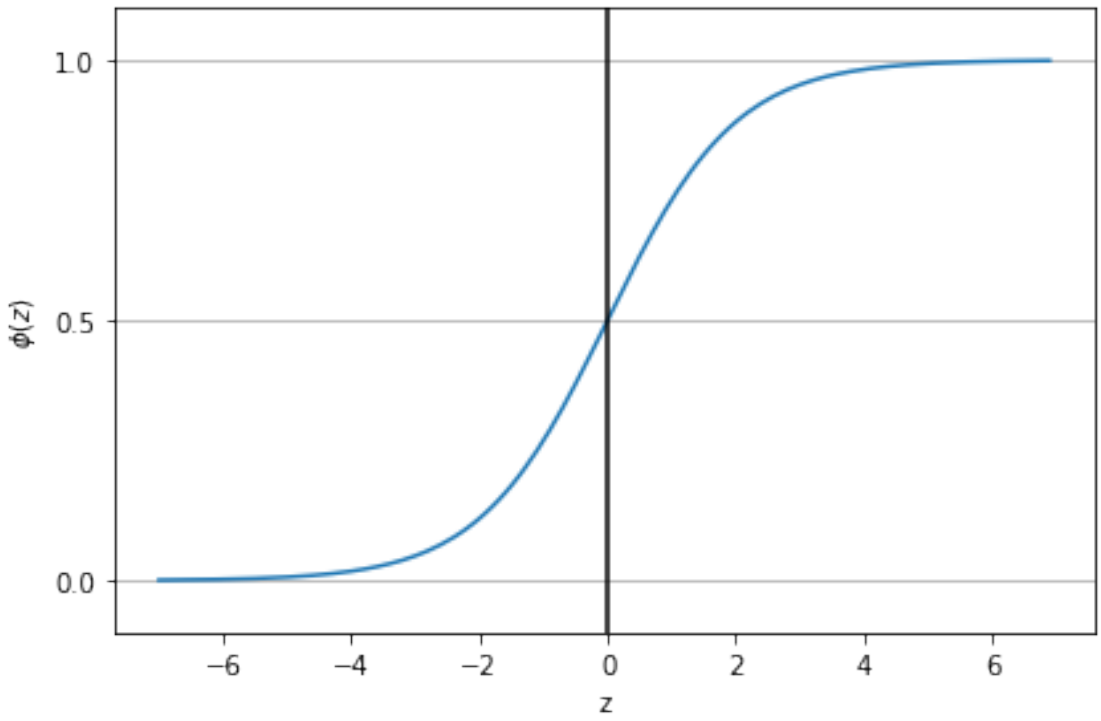
activation funciton 用 **sigmoid function**

$$h_\theta(x) = \frac{1}{1 + e^{-\theta^T x}}$$

sigmoid function 的输出是样本属于 class 1 的概率，因此样本属于 class 0 的概率为 $1 - h_\theta(x)$.

5.1 绘制 sigmoid function:

[18]:
```
%matplotlib inline
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) # 从-7 到 7 画图
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)
```



$\phi(z)$ 接近 1 如果 $z \to \infty$, 接近 0 如果 $z \to -\infty$

使用对数似然函数重新定义损失函数

若想让预测出的结果全部正确的概率最大, 根据最大似然估计, 就是所有样本预测正确的概率相乘得到的 P(总体正确) 最大,

$$L(\theta) = \Pi_{i=1}^{m} h_\theta(x)^{y^{(i)}} (1 - h_\theta(x))^{1-y^{(i)}}$$

两边同时取 log 让其变成连加

$$l(\theta) = \sum_{i=1}^{m} y^{(i)} log\Big(h_\theta(x^{(i)})\Big) + \big(1 - y^{(i)}\big) log\Big(1 - h_\theta(x^{(i)})\Big).$$

因为在函数最优化的时候习惯让一个函数越小越好, 所以我们在前边加一个负号，然后取平均得到损失函数

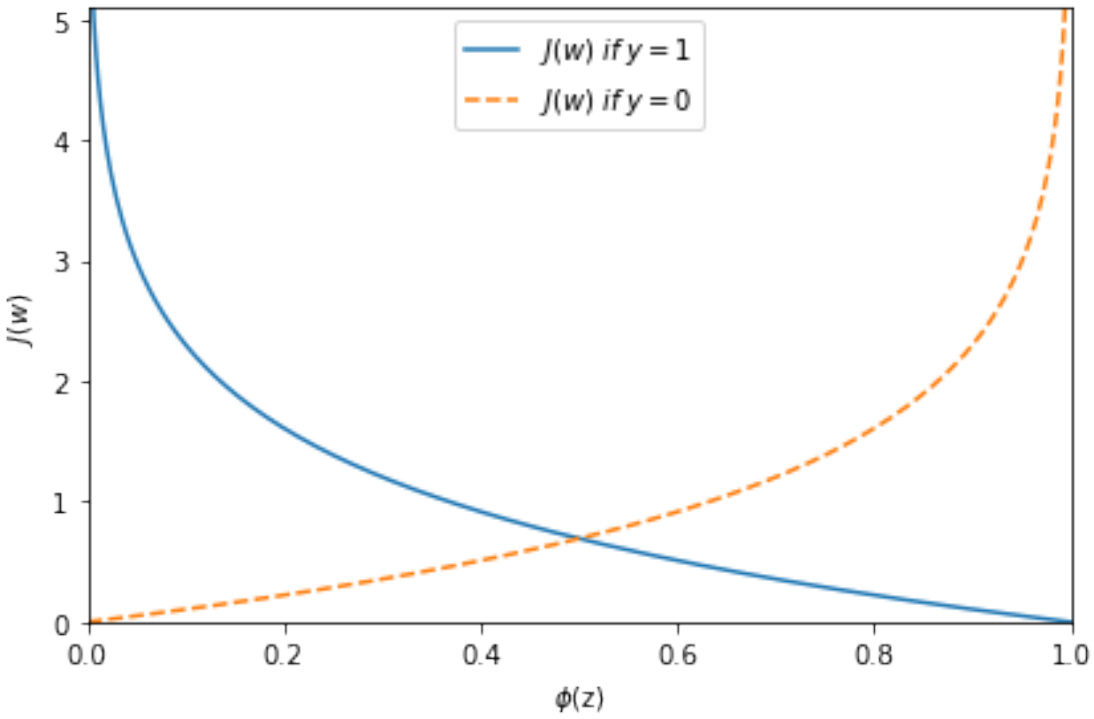
$$J(\theta) = \frac{1}{m} \sum_{i=1}^{m} -y^{(i)} log\Big(h_\theta(x^{(i)})\Big) - \big(1 - y^{(i)}\big) log\Big(1 - h_\theta(x^{(i)})\Big).$$

定义

$$Cost(h_\theta(x), y) = \begin{cases} -log(h_\theta(x)) & \text{if } y = 1 \\ -log(1 - h_\theta(x)) & \text{if } y = 0 \end{cases}$$

5.2 绘制损失函数:

[19]:
```
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)
```



上图描述了对一个样本点进行分类的损失，正确分类时损失趋于 0.

5.3 利用 Python 实现

使用梯度下降法更新参数

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}\left(y^{(i)}\log\left(h_\theta(x^{(i)})\right) + (1-y^{(i)})\log\left(1-h_\theta(x^{(i)})\right)\right)$$

$$\begin{aligned}
\frac{\partial J(\theta)}{\partial \theta_j} &= -\frac{1}{m}\sum_{i=1}^{m}\left(y^{(i)}\cdot\frac{1}{h_\theta(x^{(i)})} - (1-y^{(i)})\cdot\frac{1}{1-h_\theta(x^{(i)})}\right)\cdot\frac{\partial h_\theta(x^{(i)})}{\partial \theta_j} \\
&= -\frac{1}{m}\sum_{i=1}^{m}\left(\frac{y^{(i)}}{h_\theta(x^{(i)})} - \frac{1-y^{(i)}}{1-h_\theta(x^{(i)})}\right)\cdot h_\theta(x^{(i)})(1-h_\theta(x^{(i)}))\frac{\partial \theta^T x}{\partial \theta_j} \\
&= -\frac{1}{m}\sum_{i=1}^{m}\left(y^{(i)}\cdot\left(1-h_\theta(x^{(i)})\right) - (1-y^{(i)})\cdot h_\theta(x^{(i)})\right)\cdot x_j^{(i)} \\
&= -\frac{1}{m}\sum_{i=1}^{m}\left(y^{(i)} - y^{(i)}\cdot h_\theta(x^{(i)}) - h_\theta(x^{(i)}) + y^{(i)}\cdot h_\theta(x^{(i)})\right)\cdot x_j^{(i)} \\
&= -\frac{1}{m}\sum_{i=1}^{m}\left(y^{(i)} - h_\theta(x^{(i)})\right)\cdot x_j^{(i)} \\
&= \frac{1}{m}\sum_{i=1}^{m}\left(h_\theta(x^{(i)}) - y^{(i)}\right)\cdot x_j^{(i)}
\end{aligned}$$

故参数更新公式为

$$\theta_j = \theta_j + \frac{1}{m}\sum_{i=1}^{m}(y^{(i)} - h_\theta(x^{(i)}))\cdot x_j^{(i)}$$

[20]:
```
class LogisticRegression(object):
    """LogisticRegression classifier.

    Parameters
    ------------
    eta : float
        Learning rate (between 0.0 and 1.0)
    n_iter : int
        Passes over the training dataset.

    Attributes
    -----------
    w_ : 1d-array
```

```
    Weights after fitting.
cost_ : list
    Cost in every epoch.

"""
def __init__(self, eta=0.01, n_iter=50):
    self.eta = eta
    self.n_iter = n_iter

def fit(self, X, y):
    """ Fit training data.

    Parameters
    ----------
    X : {array-like}, shape = [n_samples, n_features]
        Training vectors, where n_samples is the number of samples and
        n_features is the number of features.
    y : array-like, shape = [n_samples]
        Target values.

    Returns
    -------
    self : object

    """
    self.w_ = np.zeros(1 + X.shape[1])
    self.cost_ = []
    for i in range(self.n_iter):
        y_val = self.activation(X)
        errors = (y - y_val)
        neg_grad = X.T.dot(errors)
        self.w_[1:] += self.eta * neg_grad
        self.w_[0] += self.eta * errors.sum()
        self.cost_.append(self._logit_cost(y, self.activation(X)))
    return self
```

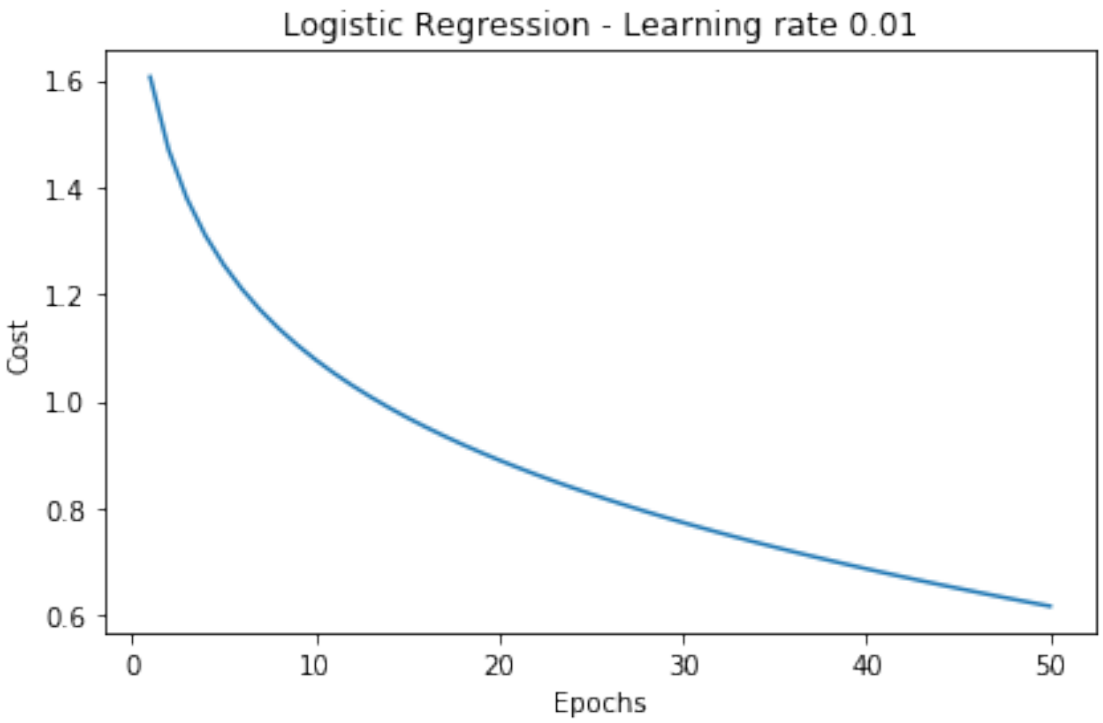
```
    def _logit_cost(self, y, y_val):
        logit = -y.dot(np.log(y_val)) - ((1 - y).dot(np.log(1 - y_val)))
        return logit

    def _sigmoid(self, z):
        return 1.0 / (1.0 + np.exp(-z))

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

    def activation(self, X):
        """ Activate the logistic neuron"""
        z = self.net_input(X)
        return self._sigmoid(z)

    def predict_proba(self, X):
        """
        Predict class probabilities for X.

        Parameters
        ----------
        X : {array-like, sparse matrix}, shape = [n_samples, n_features]
            Training vectors, where n_samples is the number of samples and
            n_features is the number of features.

        Returns
        ----------
          Class 1 probability : float

        """
        return activation(X)

    def predict(self, X):
        """
        Predict class labels for X.
```

```
        Parameters
        ----------
        X : {array-like, sparse matrix}, shape = [n_samples, n_features]
            Training vectors, where n_samples is the number of samples and
            n_features is the number of features.

        Returns
        ----------
        class : int
            Predicted class label.

        """
        # equivalent to np.where(self.activation(X) >= 0.5, 1, 0)
        return np.where(self.net_input(X) >= 0.0, 1, 0)
```

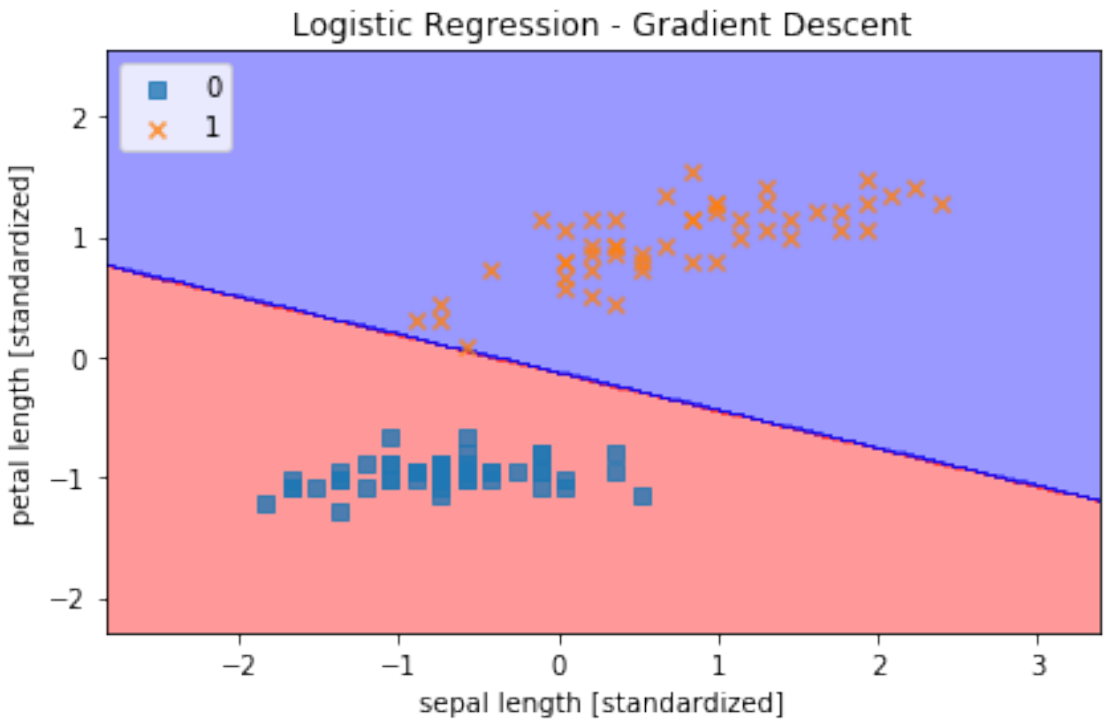
[21]:
```
y[y == -1] = 0  # 将负性标签编码为 0
y
```

[21]:
```
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1])
```

[22]:
```
lr = LogisticRegression(n_iter=50, eta=0.01).fit(X_std, y)
plt.plot(range(1, len(lr.cost_) + 1), np.log10(lr.cost_))
plt.xlabel('Epochs')
plt.ylabel('Cost')
plt.title('Logistic Regression - Learning rate 0.01')

plt.tight_layout()
```



[23]:
```
plot_decision_regions(X_std, y, classifier=lr)
plt.title('Logistic Regression - Gradient Descent')
plt.xlabel('sepal length [standardized]')
plt.ylabel('petal length [standardized]')
plt.legend(loc='upper left')
plt.tight_layout()
```



6 利用 scikit-learn 进行分类任务

6.1 数据加载和数据预处理

加载 scikit-learn 自带的数据集 Iris，这里，第三列表示 petal length, 第四列表示 petal width, 类别标签已经被转成 integer 标签：

0=Iris-Setosa, 1=Iris-Versicolor, 2=Iris-Virginica.

[24]:
```
from sklearn import datasets
import numpy as np

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

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

```
print(iris.feature_names)
print(iris.target_names)
```

```
dict_keys(['data', 'target', 'target_names', 'DESCR', 'feature_names',
'filename'])
Class labels: [0 1 2]
['sepal length (cm)', 'sepal width (cm)', 'petal length (cm)', 'petal width
(cm)']
['setosa' 'versicolor' 'virginica']
```

数据分割：

70% 训练数据，30% 测试数据

[25]:
```
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=0)
```

特征标准化

[26]:
```
from sklearn.preprocessing import StandardScaler

sc = StandardScaler()
sc.fit(X_train)
X_train_std = sc.transform(X_train) # standardize by mean & std
X_test_std = sc.transform(X_test)
```

6.2 其它数据

Scikit-learn 里有很多的现成数据可以用来测试学习算法，基本上可以分为三类：

- **Packaged Data:** 这些小数据集在安装 sklearn 的时候已经打包，可以通过 `sklearn.datasets.load_*` 的方式进行使用。
- **Downloadable Data:** 这些数据集会大一些，可以通过 `sklearn.datasets.fetch_*` 的方式下载进行使用。
- **Generated Data:** 这些数据是模式数据，可以通过 `sklearn.datasets.make_*`，进行模拟产生。比如产生一组符合线性回归的数据集。

你可以很方便的在 Notebook 里使用这些数据集，在导入 datasets 子模块之后，

输入

```
datasets.load_<TAB>
```

或者

```
datasets.fetch_<TAB>
```

或者

```
datasets.make_<TAB>
```

就可以看到一堆可用的函数。

通过 fetch_ 下载的数据会存在本地，home 目录下的子目录。你可以通过以下命令定位它：

[27]:
```
from sklearn.datasets import get_data_home
get_data_home()
```

[27]:
```
'C:\\Users\\lei\\scikit_learn_data'
```

6.3 利用 scikit-learn 训练感知器

[28]:
```
from sklearn.linear_model import Perceptron
# sklearn 中有封装好的 Perceptron 函数
ppn = Perceptron(max_iter=40, eta0=0.1, random_state=0, tol=1e-3)
ppn.fit(X_train_std, y_train)
```

[28]:
```
Perceptron(alpha=0.0001, class_weight=None, early_stopping=False, eta0=0.1,
           fit_intercept=True, max_iter=40, n_iter_no_change=5, n_jobs=None,
           penalty=None, random_state=0, shuffle=True, tol=0.001,
           validation_fraction=0.1, verbose=0, warm_start=False)
```

[29]:
```
y_test.shape
```

[29]:
```
(45,)
```

[30]:
```
y_pred = ppn.predict(X_test_std) # predict
print('Misclassified samples: %d' % (y_test != y_pred).sum()) # 错误个数
```

```
Misclassified samples: 5
```

[31]:
```
from sklearn.metrics import accuracy_score

print('Accuracy: %.2f' % accuracy_score(y_test, y_pred)) # 准确率
```

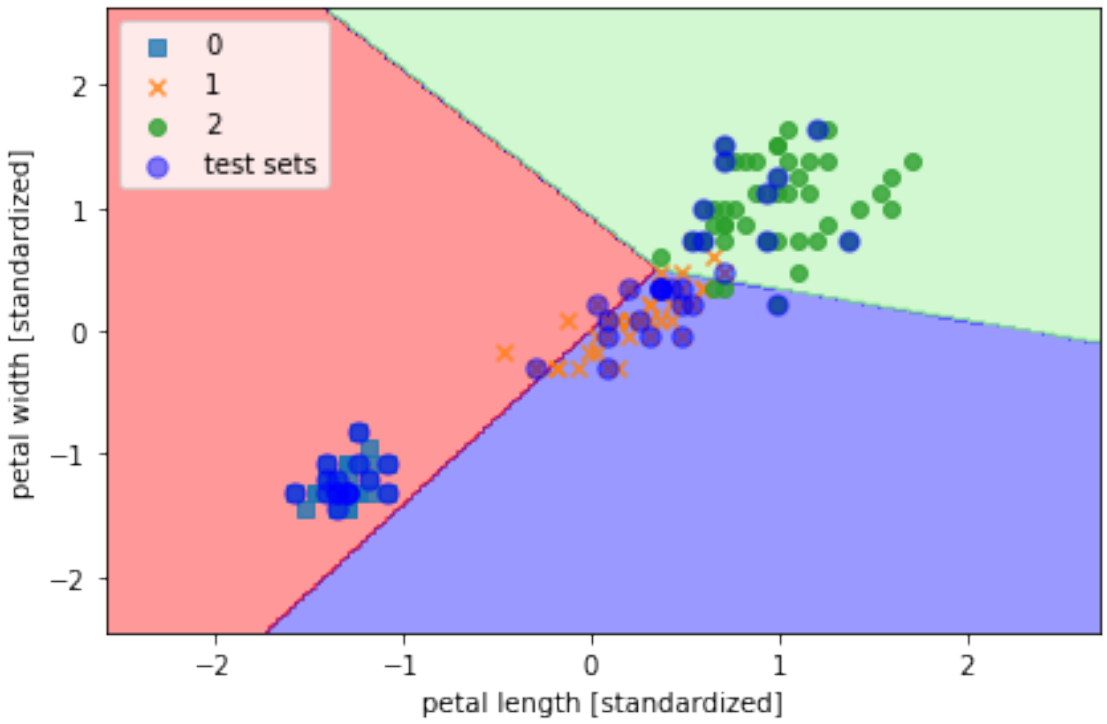
```
Accuracy: 0.89
```

[32]:
```
from matplotlib.colors import ListedColormap
import matplotlib.pyplot as plt
%matplotlib inline

# 重新定义画决策边界函数，使得能区分训练数据和测试数据
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=0.4, cmap=cmap)
    plt.xlim(xx1.min(), xx1.max())
    plt.ylim(xx2.min(), xx2.max())

    # plot all samples
    for idx, cl in enumerate(np.unique(y)):
        plt.scatter(x=X[y == cl, 0], y=X[y == cl, 1],
                    #alpha=0.8, c=cmap(idx),
                    alpha=0.8,
                    marker=markers[idx], label=cl)

    # highlight test samples
```

```
    if test_idx:
        X_test, y_test = X[test_idx, :], y[test_idx]
        plt.scatter(X_test[:, 0], X_test[:, 1], c='b',
                alpha=0.5, linewidth=1, marker='o',
                s=55, label='test sets')
```

使用标准化后的数据进行感知器训练：

[33]:
```
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('./figures/iris_perceptron_scikit.png', dpi=300)
# 这次是 3 个分类一起
```



Peceptron 模型对于并不是完全线性隔离的数据集不能收敛, 所以实际应用中用的不多.

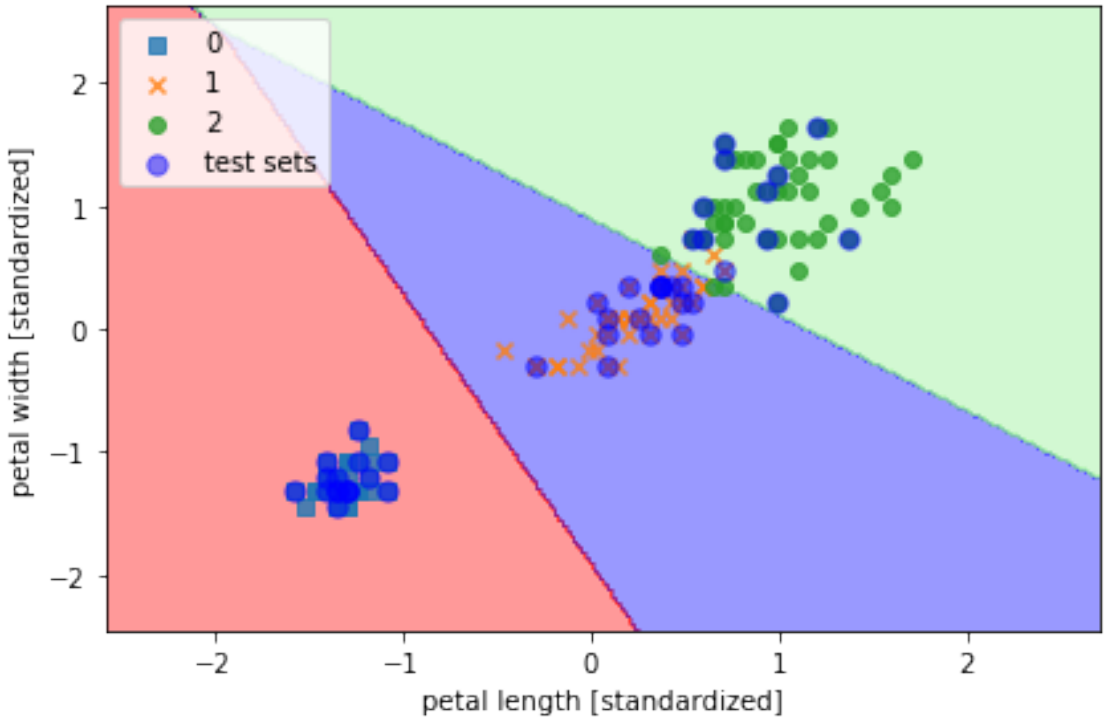
6.4 利用 logistic regression 建模类别概率

[34]:
```
# use Logistic Regression
from sklearn.linear_model import LogisticRegression
# C 正则化强度的倒数，为正的浮点数。数值越小表示正则化越强
# solver 用于优化问题的算法。
# 对于小数据集来说，“liblinear”是个不错的选择，而“sag”和'saga' 对于大型数据集会更快。
lr = LogisticRegression(C=1000.0, random_state=0,solver='liblinear')
lr.fit(X_train_std, y_train)
```

```
C:\ProgramData\Anaconda3\lib\site-packages\sklearn\linear_model\logistic.py:469:
FutureWarning: Default multi_class will be changed to 'auto' in 0.22. Specify
the multi_class option to silence this warning.
  "this warning.", FutureWarning)
```

[34]:
```
LogisticRegression(C=1000.0, class_weight=None, dual=False, fit_intercept=True,
                   intercept_scaling=1, l1_ratio=None, max_iter=100,
                   multi_class='warn', n_jobs=None, penalty='l2',
                   random_state=0, solver='liblinear', tol=0.0001, verbose=0,
                   warm_start=False)
```

[35]:
```
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('./figures/logistic_regression.png', dpi=300)
```



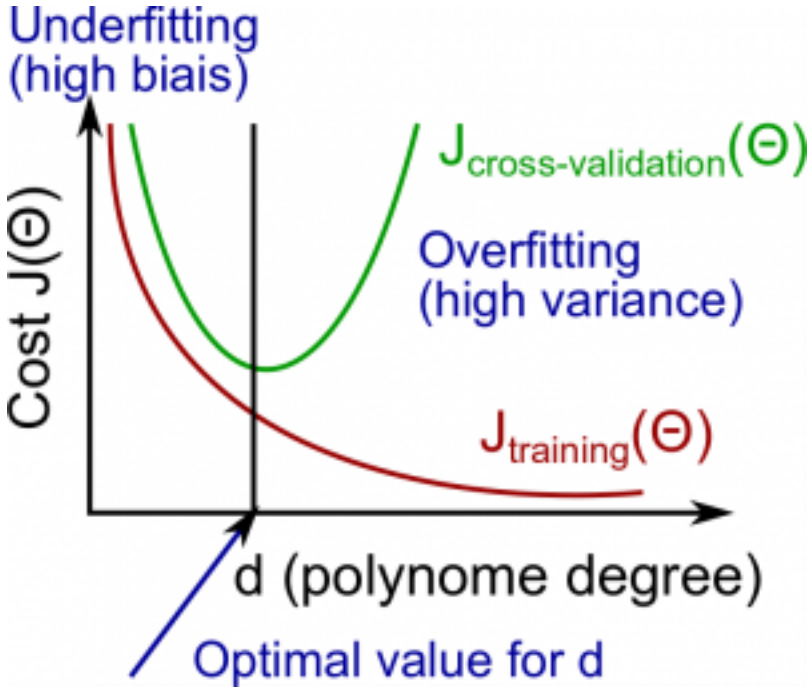
[36]:
```
lr.predict_proba(X_test_std[0,:].reshape(1,-1)) # predict probability
```

[36]:
```
array([[1.78177322e-11, 6.12453348e-02, 9.38754665e-01]])
```

7 过拟合和欠拟合

1、高偏差 (high bias) 对应着欠拟合，此时训练误差也较大，可以理解为对任何新数据（不论其是否属于训练集），都有着较大的测试误差，偏离真实预测较大。

2、高方差 (high variance) 对应着过拟合，此时训练误差很小，对于新数据来说，如果其属性与训练集类似，它的测试误差就会小些，如果属性与训练集不同，测试误差就会很大，因此有一个比较大的波动，因此说是高方差。



7.1 Regularization （正则化）

机器学习中经常可以看到损失函数后面会添加一个额外项，常用的额外项一般有两种，一般英文称作 L_1-norm 和 L_2-norm，中文称作 L_1 正则化和 L_2 正则化，或者 L_1 范数和 L_2 范数。

L_1 正则化和 L_2 正则化可以看做是损失函数的惩罚项。所谓『惩罚』是指对损失函数中的某些参数做一些限制。对于线性回归模型，使用 L_1 正则化的模型建叫做 Lasso 回归，使用 L_2 正则化的模型叫做 Ridge 回归（岭回归）。

解决 overfitting: 模型拟合的过好, 以致于没有一般性, 预测新的样本的结果就会很差。

常用的是 L_2 正则化

$$\frac{\lambda}{2}\|w\|^2 = \frac{\lambda}{2}\sum_{j=1}^{m} w_j^2$$

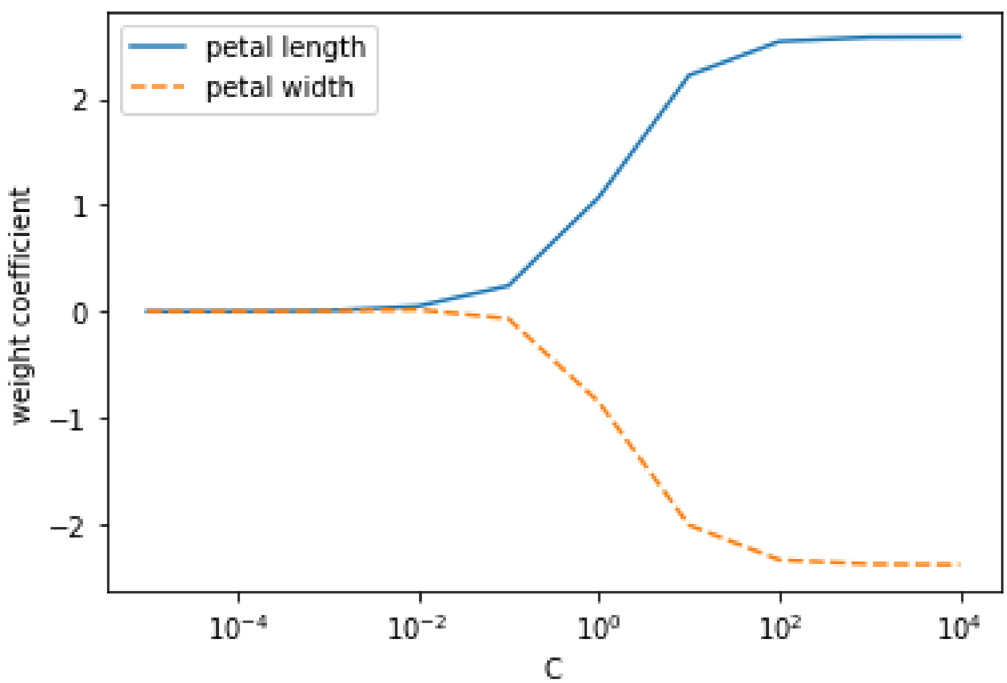
λ 就是 regularization parameter, 可以用来控制拟合训练数据的好坏, 而 $C = \frac{1}{\lambda}$ 就是前面提到过的参数。

[37]:
```
weights, params = [], []
for c in range(-5, 5):
    # 不加指定 solver 和 multi_class 会有警告
    lr = LogisticRegression(C=10**c, random_state=0, solver='liblinear')
    lr.fit(X_train_std, y_train)
    weights.append(lr.coef_[1])
```

```python
    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('./figures/regression_path.png', dpi=300)

# C 减小的话，就是增加 regularization
```

```
C:\ProgramData\Anaconda3\lib\site-packages\sklearn\linear_model\logistic.py:469:
FutureWarning: Default multi_class will be changed to 'auto' in 0.22. Specify
the multi_class option to silence this warning.
  "this warning.", FutureWarning)
C:\ProgramData\Anaconda3\lib\site-packages\sklearn\linear_model\logistic.py:469:
FutureWarning: Default multi_class will be changed to 'auto' in 0.22. Specify
the multi_class option to silence this warning.
  "this warning.", FutureWarning)
C:\ProgramData\Anaconda3\lib\site-packages\sklearn\linear_model\logistic.py:469:
FutureWarning: Default multi_class will be changed to 'auto' in 0.22. Specify
the multi_class option to silence this warning.
  "this warning.", FutureWarning)
C:\ProgramData\Anaconda3\lib\site-packages\sklearn\linear_model\logistic.py:469:
FutureWarning: Default multi_class will be changed to 'auto' in 0.22. Specify
the multi_class option to silence this warning.
  "this warning.", FutureWarning)
C:\ProgramData\Anaconda3\lib\site-packages\sklearn\linear_model\logistic.py:469:
FutureWarning: Default multi_class will be changed to 'auto' in 0.22. Specify
the multi_class option to silence this warning.
  "this warning.", FutureWarning)
C:\ProgramData\Anaconda3\lib\site-packages\sklearn\linear_model\logistic.py:469:
FutureWarning: Default multi_class will be changed to 'auto' in 0.22. Specify
the multi_class option to silence this warning.
```

```
  "this warning.", FutureWarning)
C:\ProgramData\Anaconda3\lib\site-packages\sklearn\linear_model\logistic.py:469:
FutureWarning: Default multi_class will be changed to 'auto' in 0.22. Specify
the multi_class option to silence this warning.
  "this warning.", FutureWarning)
C:\ProgramData\Anaconda3\lib\site-packages\sklearn\linear_model\logistic.py:469:
FutureWarning: Default multi_class will be changed to 'auto' in 0.22. Specify
the multi_class option to silence this warning.
  "this warning.", FutureWarning)
C:\ProgramData\Anaconda3\lib\site-packages\sklearn\linear_model\logistic.py:469:
FutureWarning: Default multi_class will be changed to 'auto' in 0.22. Specify
the multi_class option to silence this warning.
  "this warning.", FutureWarning)
C:\ProgramData\Anaconda3\lib\site-packages\sklearn\linear_model\logistic.py:469:
FutureWarning: Default multi_class will be changed to 'auto' in 0.22. Specify
the multi_class option to silence this warning.
  "this warning.", FutureWarning)
```



增大正则（惩罚），即减小 C，权重收缩（减小），模型复杂度减小。

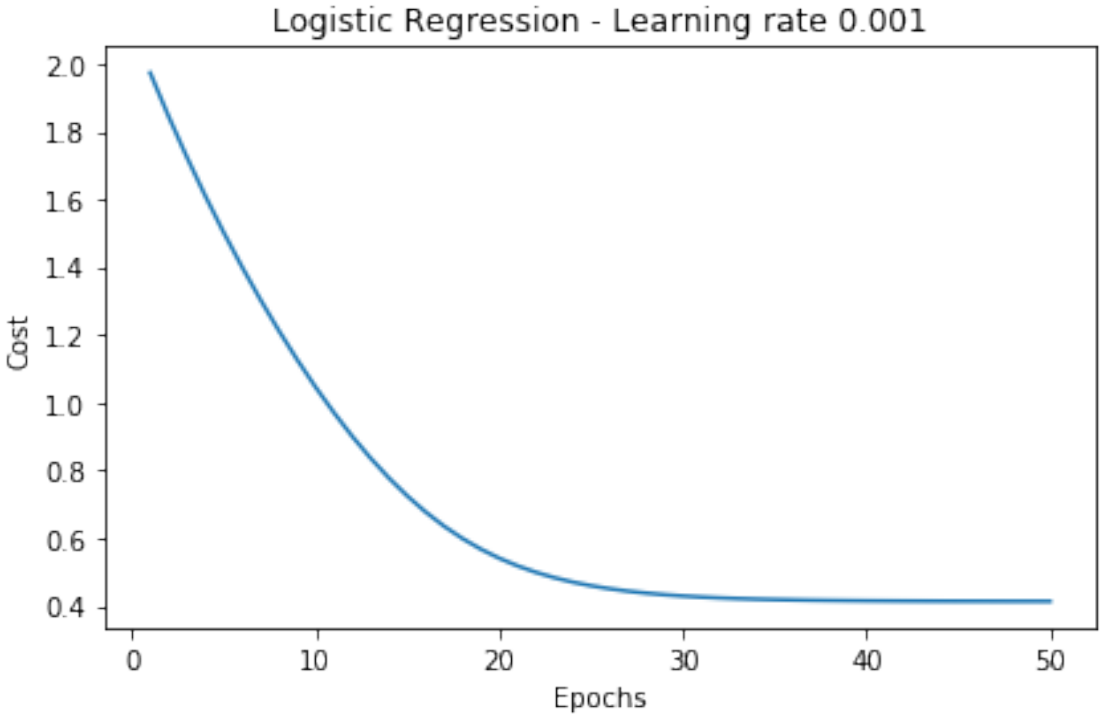
7.2 Logistic regression with regularization（手动实现）

[38]:
```
class LogitGD(object):
    """Logistic Regression classifier.

    Parameters
    ------------
    eta : float
        Learning rate (between 0.0 and 1.0)
    n_iter : int
        Passes over the training dataset.

    Attributes
    -----------
    w_ : 1d-array
        Weights after fitting.
    errors_ : list
        Number of misclassifications in every epoch.

    """
    def __init__(self, eta=0.01, lamb = 0.01, n_iter=50):
        self.eta = eta
        self.n_iter = n_iter
        self.lamb = lamb

    def fit(self, X, y):
        """ Fit training data.

        Parameters
        ----------
        X : {array-like}, shape = [n_samples, n_features]
            Training vectors, where n_samples is the number of samples and
            n_features is the number of features.
        y : array-like, shape = [n_samples]
            Target values.

        Returns
```

```
        -------
        self : object

        """
        self.w_ = np.zeros(1 + X.shape[1])
        self.cost_ = []


        for i in range(self.n_iter):
            output = self.net_input(X)
            errors = (y - output)
            self.w_[1:] += self.eta * X.T.dot(errors) - self.lamb* self.w_[1:]
            self.w_[0] += self.eta * errors.sum()
            cost = (errors**2).sum() / 2.0 + self.lamb* np.sum(self.w_[1:]**2)
            self.cost_.append(cost)
        return self

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

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

    def activation(self, X):
        """Compute linear activation"""
        return sigmoid(self.net_input(X))

    def predict(self, X):
        """Return class label after unit step"""
        return np.where(self.activation(X) >= 0.5, 1, -1)
```

[39]:
```
lr = LogitGD(n_iter=50, eta=0.001).fit(X_train_std, y_train)

plt.plot(range(1, len(lr.cost_) + 1), np.log10(lr.cost_))
plt.xlabel('Epochs')
```

```
plt.ylabel('Cost')
plt.title('Logistic Regression - Learning rate 0.001')

plt.tight_layout()
```



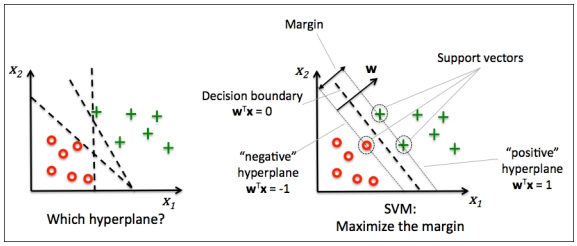
8 支持向量机（SVM）

8.1 线性可分支持向量机

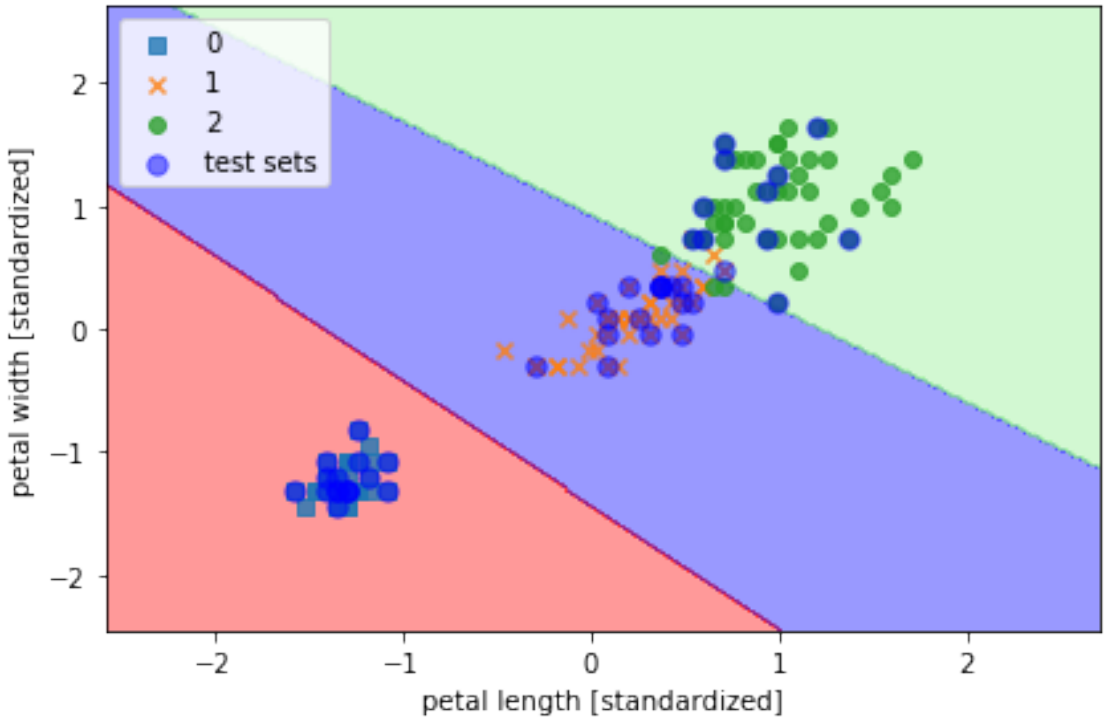
目标是找到正确划分训练数据并且间隔最大的分离超平面。

感知器：最小化错误分类数

支持向量机：最大化间隔



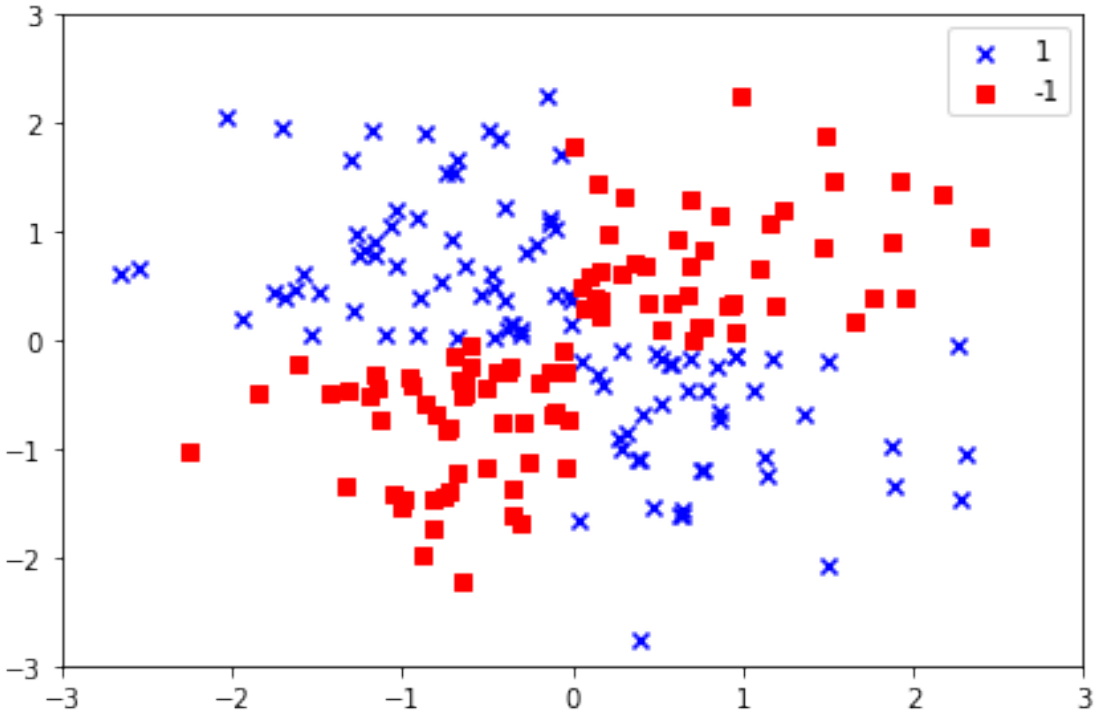
[40]:
```
# 训练 SVC, 支持向量分类器
from sklearn.svm import SVC

svm = SVC(kernel='linear', C=1.0, random_state=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('./figures/support_vector_machine_linear.png', dpi=300)
```



8.2 利用 kernel SVM 解决非线性问题

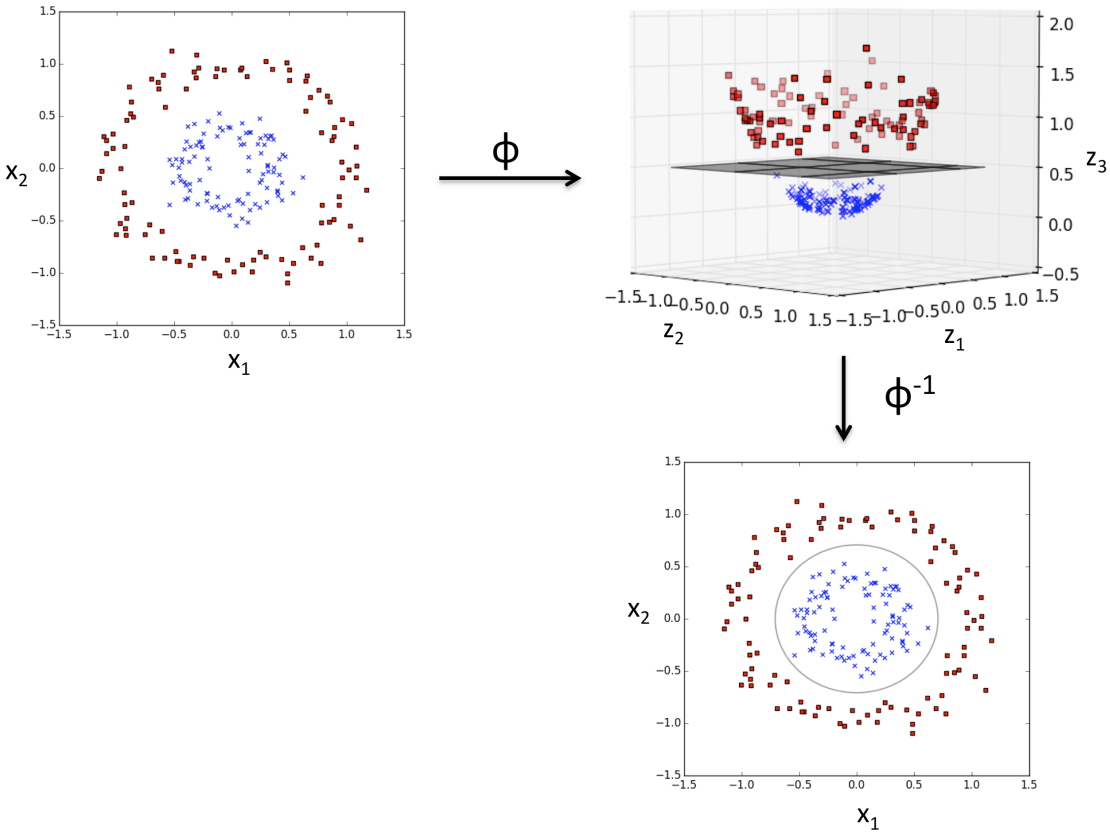
SVM 可以解决非线性问题

[41]:
```
# 创建一个简单的数据集，满足 xor 异或规则
np.random.seed(0)
X_xor = np.random.randn(200, 2)
# 100 个 标签为 1, 100 个标签为 0
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('./figures/xor.png', dpi=300)

# 由于这类数据不是线性可分的，所以
# 使用普通的  linear logistic Regression 不能很好将样本区分
```

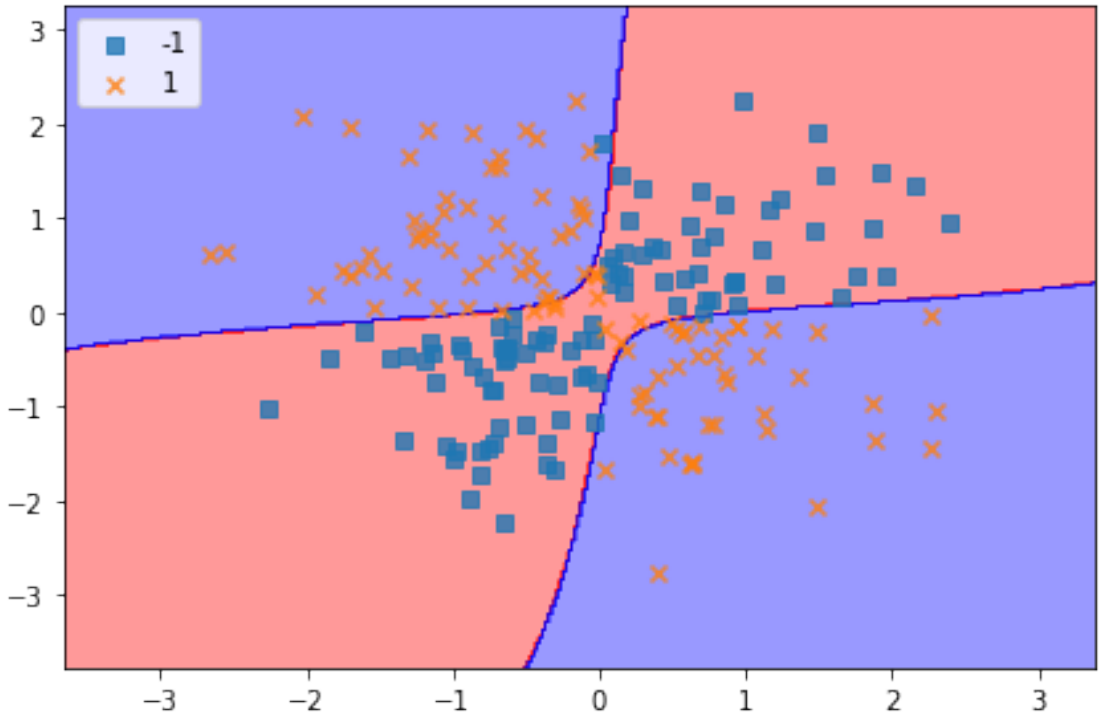


对于非线性的情况，SVM 的处理方法是选择一个核函数 κ(□,□)，通过将数据映射到高维空间，来解决在原始空间中线性不可分的问题。支持向量机首先在低维空间中完成计算，然后通过核函数将输入空间映射到高维特征空间，最终在高维特征空间中构造出最优分离超平面，从而把平面上本身不好分的非线性数据分开。

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



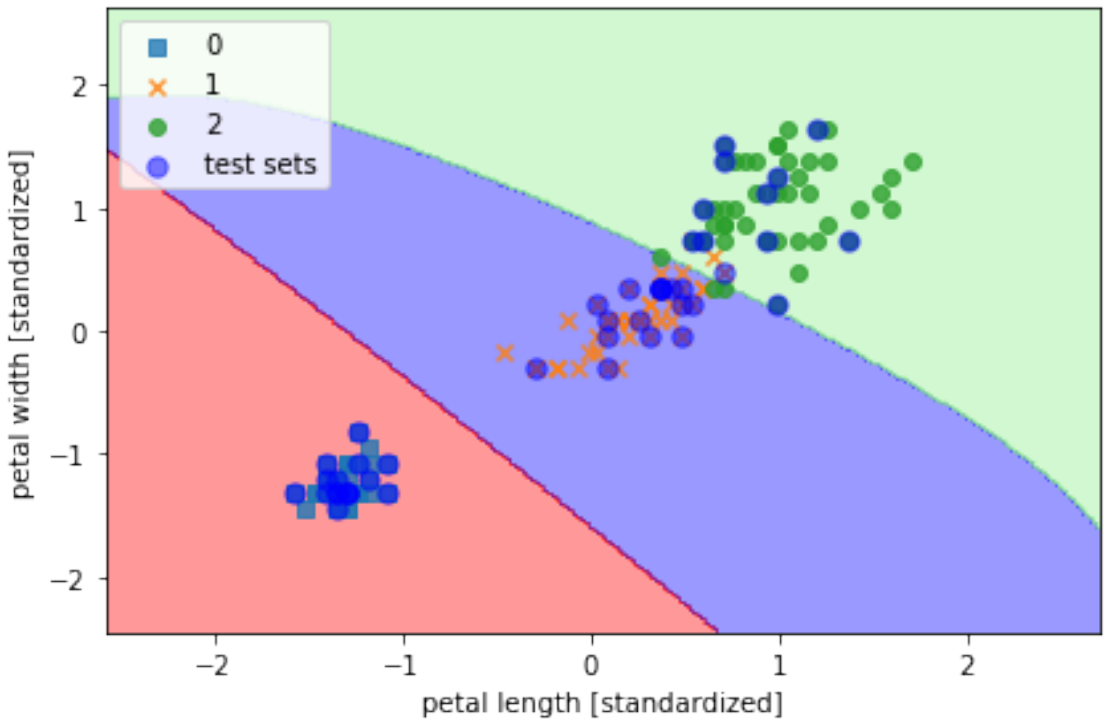
[42]:
```
# 使用 sum kernel 方法, 投射到高维度中, 使之成为线性可分的
svm = SVC(kernel='rbf', random_state=0, gamma=0.1, C=10.0)
svm.fit(X_xor, y_xor)
plot_decision_regions(X_xor, y_xor, classifier=svm)

plt.legend(loc='upper left')
plt.tight_layout()
# plt.savefig('./figures/support_vector_machine_rbf_xor.png', dpi=300)
```



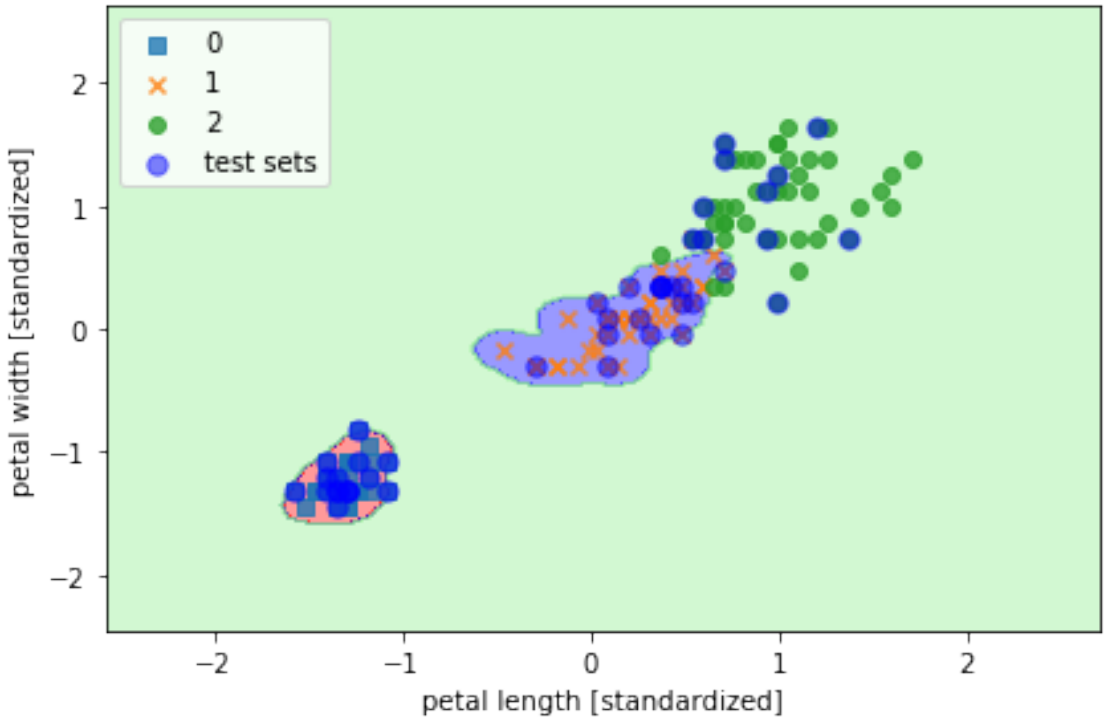
gamma: 'rbf'，'poly' 和 'sigmoid' 的核函数参数。默认是'auto'，则会选择1/n_features。

γ 增加, 也就增加了训练样本的影响, 增加在训练样本上的准确率，容易过拟合。

[43]:
```
from sklearn.svm import SVC
## gamma 较小
svm = SVC(kernel='rbf', random_state=0, 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('./figures/support_vector_machine_rbf_iris_1.png', dpi=300)
```



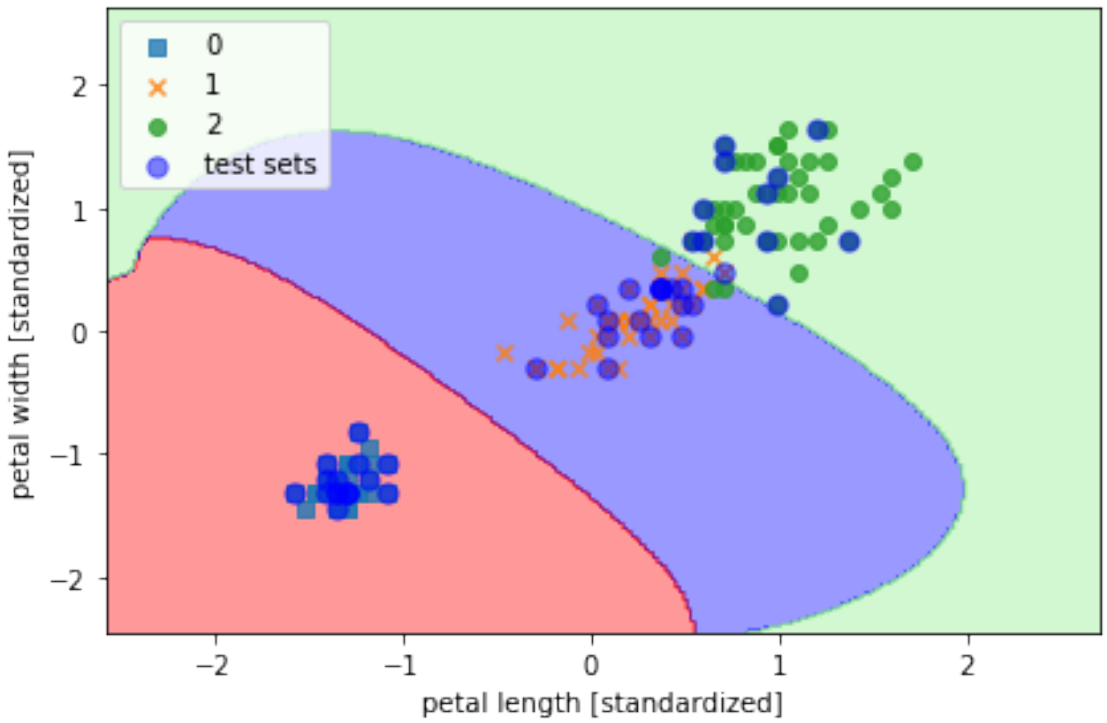
[44]:
```
# gamma 很大, 过拟合
svm = SVC(kernel='rbf', random_state=0, gamma=50, C=1.0)
svm.fit(X_train_std, y_train)

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('./figures/support_vector_machine_rbf_iris_2.png', dpi=300)
```



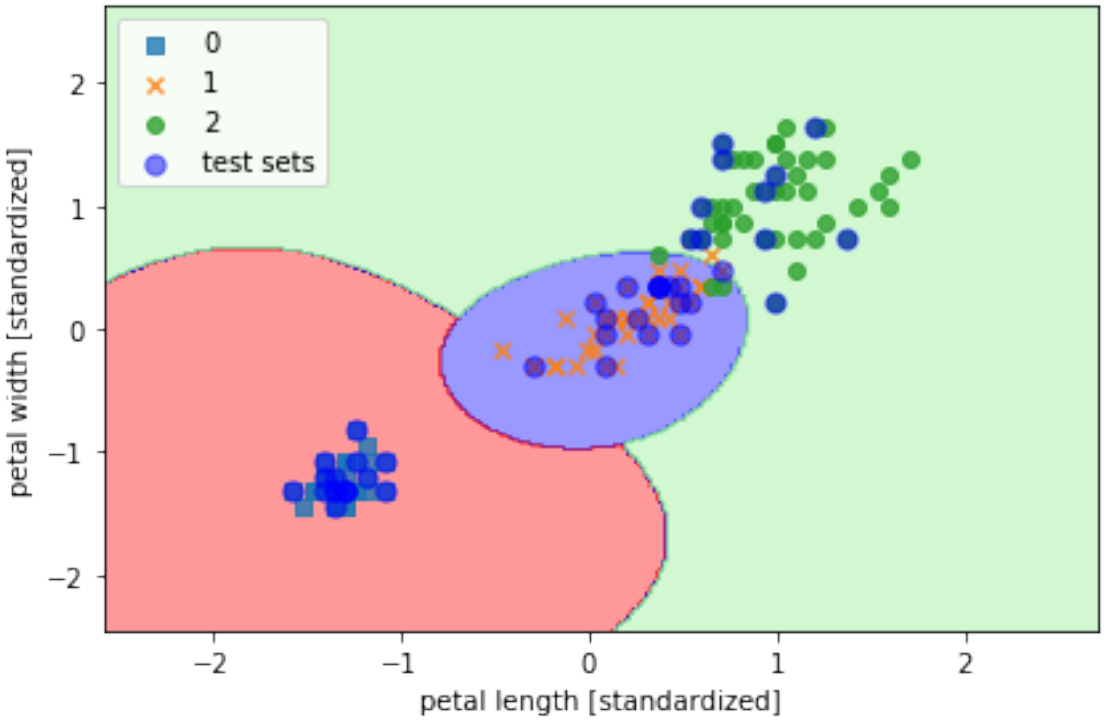
C：惩罚参数，默认值是 1.0。

C 越大，相当于惩罚松弛变量，希望松弛变量接近 0，即对误分类的惩罚增大，趋向于对训练集全分对的情况，这样对训练集测试时准确率很高，但泛化能力弱。C 值小，对误分类的惩罚减小，允许容错，将他们当成噪声点，泛化能力较强。对于训练样本带有噪声的情况，一般采用后者，把训练样本集中错误分类的样本作为噪声。

[45]:
```
# 较小的 C
svm = SVC(kernel='rbf', random_state=0, C=0.1)
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('./figures/support_vector_machine_rbf_iris_2.png', dpi=300)
```



[46]:
```
# 较大的 C
svm = SVC(kernel='rbf', random_state=0, C=100.0)
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('./figures/support_vector_machine_rbf_iris_2.png', dpi=300)
```



kernel: 核函数，默认是 rbf, 最常用的核函数是 linear 核与 rbf 核。

- linear：主要用于线性可分的情形，参数少，速度快，对于一般数据，分类效果已经很理想。
- rbf：主要用于线性不可分的情形，参数多，分类结果非常依赖于参数，通常通过交叉验证来寻找合适的参数，比较耗时间。RBF 核是应用最广的核函数，无论是小样本还是大样本，高维还是低维，RBF 核函数均适用。

9 K 近邻算法

9.1 原理

一种基于记忆的方法，不是直接学到一个模型，而是通过记住训练数据，对新的数据给出预测。

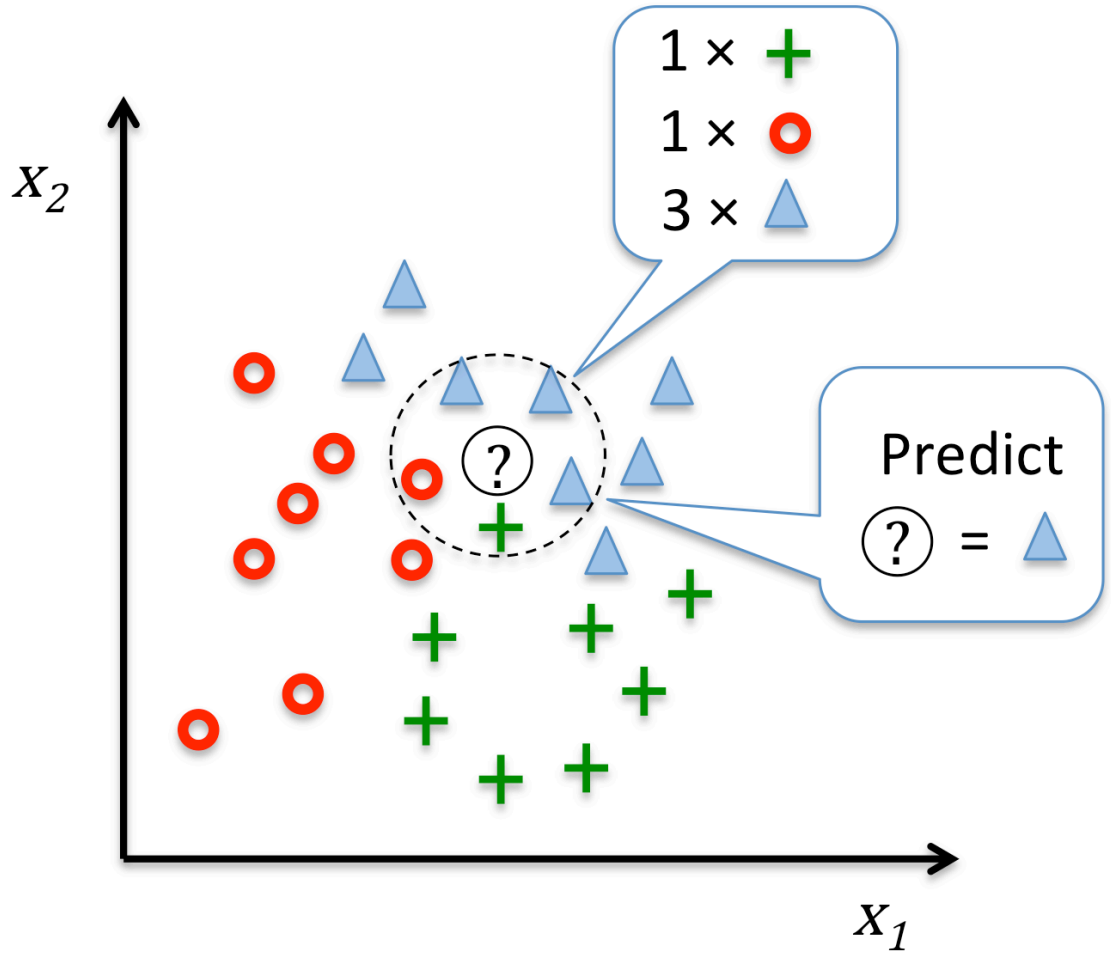
1. 选定 k 和一个距离度量
2. 找到 k 个离待分类样本最近的邻居
3. 通过投票，给出该样本的标签预测

这种方法好处在于新数据进来, 分类器可以马上学习并适应, 但是计算成本也是线性增长, 存储也是问题.

闵可夫斯基距离

$$d(x^{(i)}, x^{(j)}) = \left(\sum_k |x_k^{(i)} - x_k^{(j)}|^p\right)^{\frac{1}{p}}$$

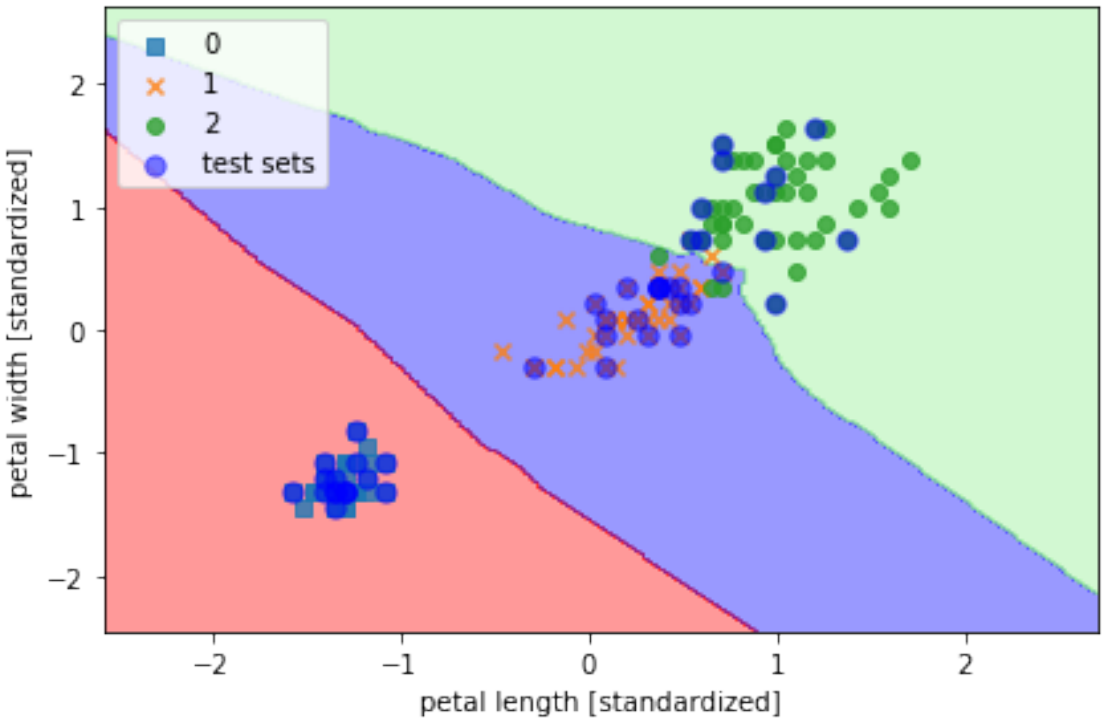
其中 $p = 2$ 是欧式距离,$p = 1$ 是曼哈顿距离.



[47]:
```
from sklearn.neighbors import KNeighborsClassifier
# 寻找 5 个邻居
knn = KNeighborsClassifier(n_neighbors=5, p=2, metric='minkowski')
knn.fit(X_train_std, y_train)
```

[47]:
```
KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='minkowski',
                     metric_params=None, n_jobs=None, n_neighbors=5, p=2,
                     weights='uniform')
```

[48]:
```
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('./figures/k_nearest_neighbors.png', dpi=300)
```



如何选择 k 是一个重点, 在应用中，我们一般取一个较小的 k 值，通常采用交叉验证法来选取最优的 k 值。

9.2 K 近邻算法的实现

实现 k 近邻算法时，主要考虑的问题是如何对训练数据进行快速 k 近邻搜索。最简单的方法是线性扫描，这时需要计算输入样本与每个训练样本的距离，非常耗时。为了提高搜索效率，可以考虑用特殊结构存储训练数据，一个常用的形结构是 kd 树。sklearn 中可以通过参数 algorithm 进行控制。

algorithm{ ‘auto’ , ‘ball_tree’ , ‘kd_tree’ , ‘brute’ }, optional Algorithm used to compute the nearest neighbors:

‘ball_tree’ will use BallTree

‘kd_tree’ will use KDTree

‘brute’ will use a brute-force search.

‘auto’ will attempt to decide the most appropriate algorithm based on the values passed to fit method.

10 分类的评价指标（Scoring metrics）

10.1 Scikit-learn 中的评价指标

[49]:
```
# 构建数据
from sklearn import datasets
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.svm import SVC

X, y = datasets.make_classification(n_samples=500, n_classes=2, random_state=0)
print(X.shape)
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.3, random_state=0)

sc = StandardScaler()
sc.fit(X_train)
X_train_std = sc.transform(X_train) # standardize by mean & std
X_test_std = sc.transform(X_test)

model = SVC(probability=True, random_state=0)
model.fit(X_train_std, y_train)
```

```
(500, 20)
```

[49]:
```
SVC(C=1.0, cache_size=200, class_weight=None, coef0=0.0,
    decision_function_shape='ovr', degree=3, gamma='auto_deprecated',
```

```
kernel='rbf', max_iter=-1, probability=True, random_state=0, shrinking=True,
tol=0.001, verbose=False)
```

sklearn 中分类任务的默认评分是 accuracy (标签预测正确的比例)

$$accuracy(y, \hat{y}) = \frac{1}{n}\sum_{i=0}^{n-1} 1(\hat{y}_i = y_i)$$

其中 $1(x)$ 为指示函数。

[50]:
```
# 测试准确率
model.score(X_test_std, y_test)
```

[50]: 0.82

[51]:
```
from sklearn.metrics import accuracy_score
y_pred = model.predict(X_test_std)
accuracy_score(y_test, y_pred)
```

[51]: 0.82

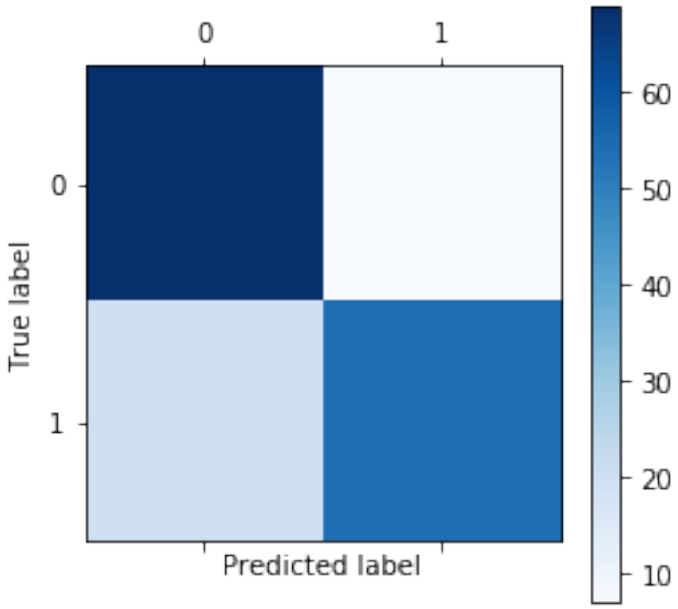
10.2 confusion matrix（混淆矩阵）

对于一个多分类的任务，我们通常知道哪些分类容易预测，哪些困难。而且我们对于不同的预测的代价也是不同。所以我们提出一种方法来补充 accuracy，称为混淆矩阵。

[52]:
```
from sklearn.metrics import confusion_matrix

y_test_pred = model.predict(X_test_std)
confmat = confusion_matrix(y_test, y_test_pred)
print(confmat)
```

```
[[69  7]
 [20 54]]
```

[53]:
```
plt.matshow(confusion_matrix(y_test, y_test_pred), cmap=plt.cm.Blues)
plt.colorbar()
plt.xlabel("Predicted label")
plt.ylabel("True label");
```



10.3 Precision（准确率）, recall（召回率）and F-measures

- **Precision** 预测为 A 类的有多少确实是 A 类
- **Recall** 有多少真实的正例被预测到
- **f1-score** 综合 P 和 R，两者的调和平均

我们用 TP, FP, TN, FN, FPR, TPR 表示 “true positive”，“false positive”，“true negative”，“false negative”，“false positive rate”，“true positive rate”：

$$PREcision = \frac{TP}{TP + FP}$$

$$RECall = TPR = \frac{TP}{FN + TP}$$

$$F1 = 2\frac{PRE \times REC}{PRE + REC}$$

$$F_\beta = (1 + \beta^2)\frac{PRE \times REC}{\beta^2 PRE + REC}$$

$$FPR = \frac{FP}{FP + TN}$$

$$TPR = \frac{TP}{FN + TP}$$

F_β 中，当 β 大于 1，更多关注 recall；当 β 小于 1，更多关注 precision。

例如：80 正常 +20 违约

预测：50 人违约（包含 20 人违约）

问：精确度，准确率，召回率

违约：正例

TP = 20，FP = 30，FN = 0，TN = 50

- acc：精确度：20+50/100
- precision：20/50 不高
- recall: 20/20
- 全预测正常呢？

[54]:
```
from sklearn.metrics import precision_score, recall_score, f1_score, fbeta_score

print('Precision: %.3f' % precision_score(y_true=y_test, y_pred=y_test_pred))
print('Recall: %.3f' % recall_score(y_true=y_test, y_pred=y_test_pred))
print('F1: %.3f' % f1_score(y_true=y_test, y_pred=y_test_pred))
print('F_beta2: %.3f' % fbeta_score(y_true=y_test, y_pred=y_test_pred, beta=2))
```

```
Precision: 0.885
Recall: 0.730
F1: 0.800
F_beta2: 0.756
```

函数 classification_report：

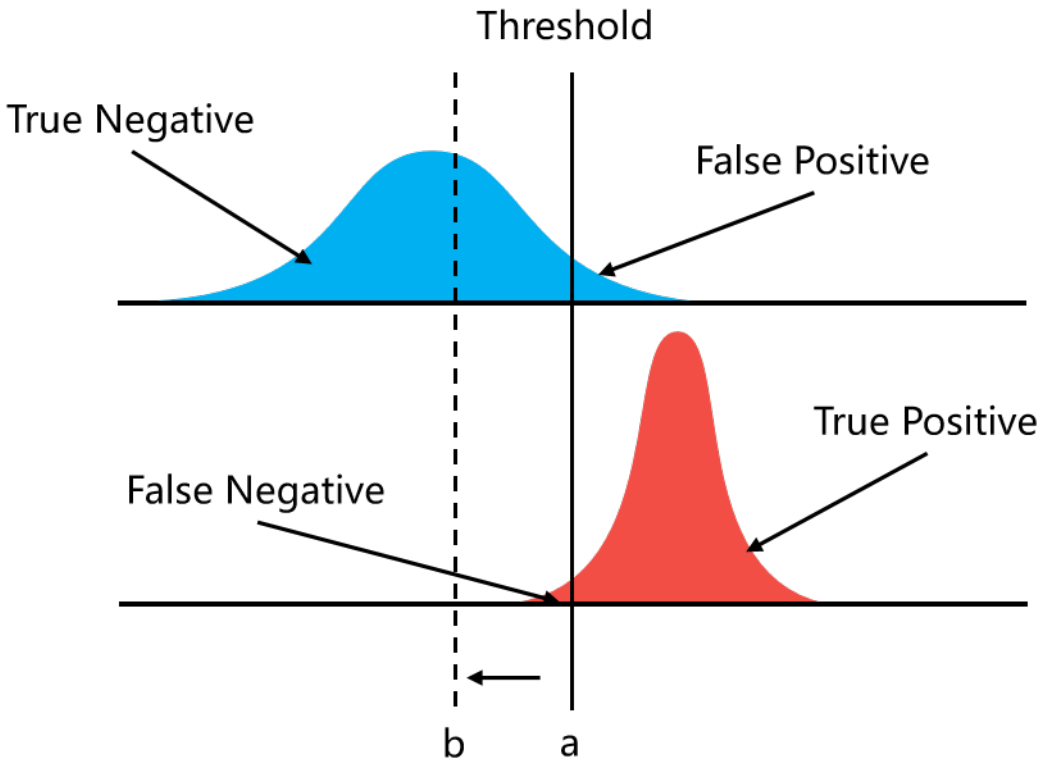
[55]:
```
from sklearn.metrics import classification_report
print(classification_report(y_test, y_test_pred))
```

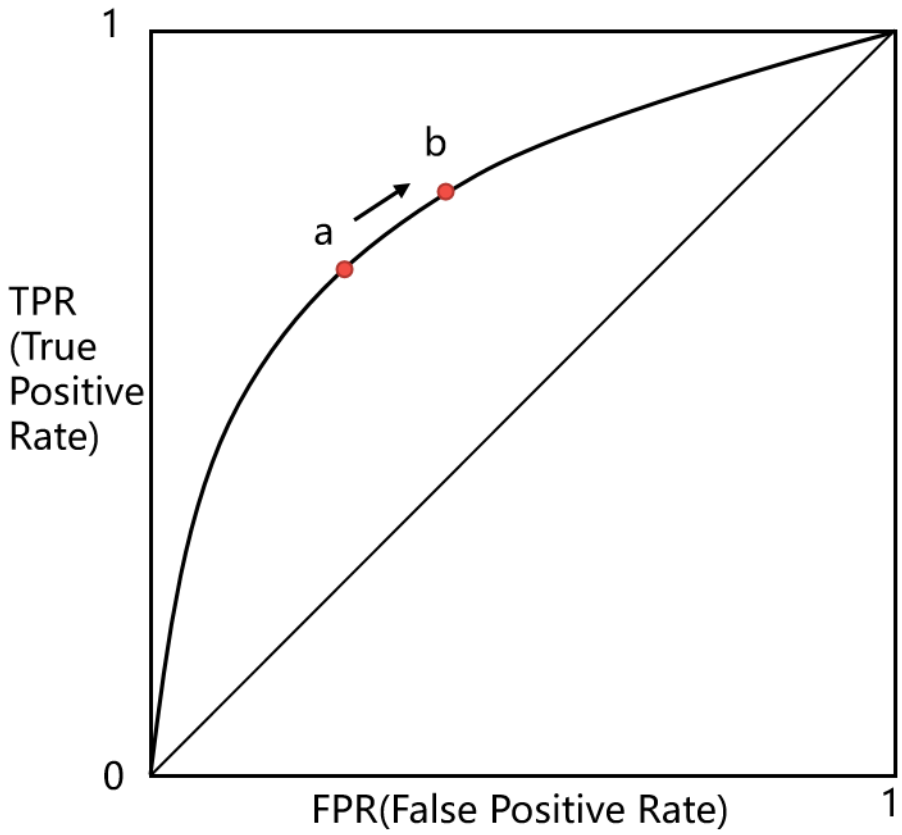
```
              precision    recall  f1-score   support
```

```
              0       0.78      0.91      0.84        76
              1       0.89      0.73      0.80        74

    accuracy                            0.82       150
   macro avg        0.83      0.82      0.82       150
weighted avg        0.83      0.82      0.82       150
```

10.4 ROC and AUC

ROC（Receiver Operating Characteristic）曲线和 AUC 常被用来评价一个二分类器（binary classifier）的优劣。





$$FPR = \frac{FP}{FP + TN}$$
$$TPR = \frac{TP}{FN + TP}$$

考虑 ROC 曲线图中的四个点和一条线:

- 第一个点，(0,1)，即 FPR=0, TPR=1，这意味着 FN（false negative）=0，并且 FP（false positive）=0。这是一个完美的分类器，它将所有的样本都正确分类。
- 第二个点，(1,0)，即 FPR=1，TPR=0，类似地分析可以发现这是一个最糟糕的分类器，因为它成功避开了所有的正确答案。
- 第三个点，(0,0)，即 FPR=TPR=0，即 FP（false positive）=TP（true positive）=0，可以发现该分类器预测所有的样本都为负样本（negative）。
- 第四个点（1,1），分类器实际上预测所有的样本都为正样本。
- 考虑 ROC 曲线图中的虚线 y=x 上的点。这条对角线上的点表示的是一个采用随机猜测策略的分类器的结果，例如 (0.5,0.5)，表示该分类器随机对于一半的样本猜测其为正样本，另外

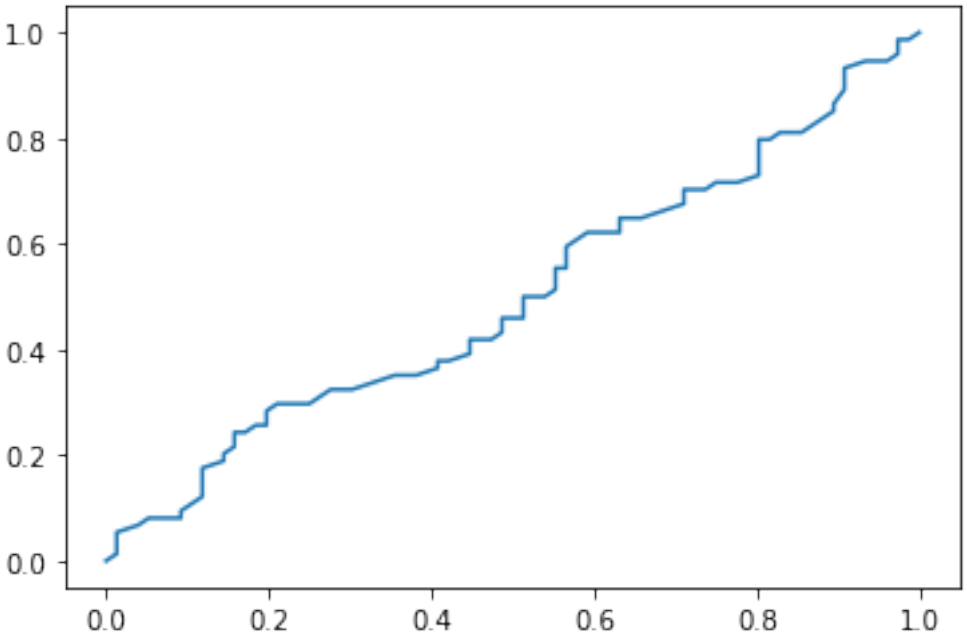
一半的样本为负样本。

因此：如果分类器效果很好, 那么图应该会在左上角. ROC 曲线越接近左上角，该分类器的性能越好。

在 ROC curve 的基础上, 可以计算 AUC – area under the curve.

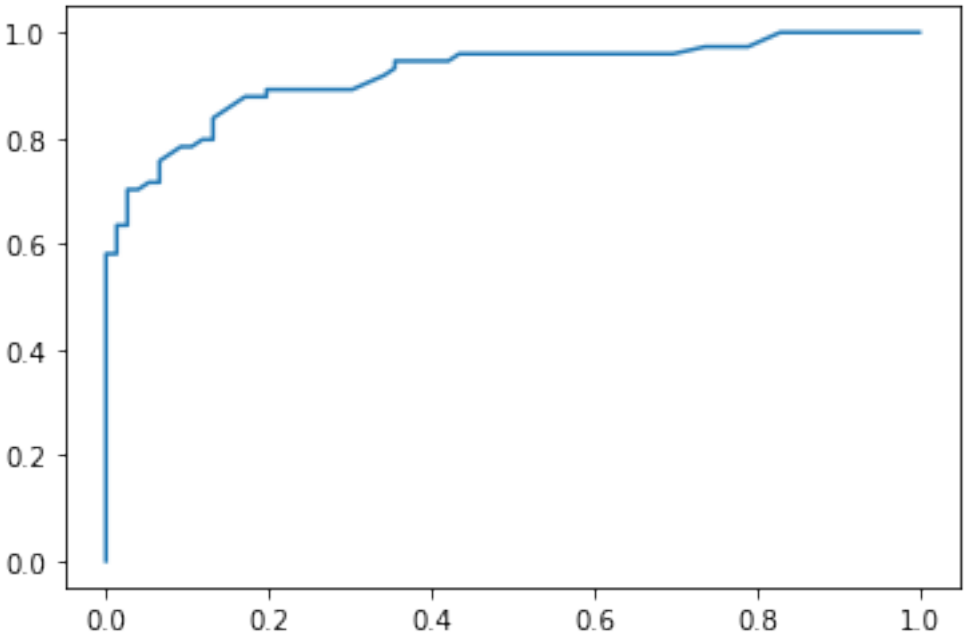
[56]:
```
def roc_curve(true_labels, predicted_probs, n_points=100, pos_class=1):
    thr = np.linspace(0,1,n_points)
    tpr = np.zeros(n_points)
    fpr = np.zeros(n_points)

    pos = true_labels == pos_class
    neg = np.logical_not(pos)
    n_pos = np.count_nonzero(pos)
    n_neg = np.count_nonzero(neg)

    for i,t in enumerate(thr):
        tpr[i] = np.count_nonzero(np.logical_and(predicted_probs >= t, pos))/␣
↪float(n_pos)
        fpr[i] = np.count_nonzero(np.logical_and(predicted_probs >= t, neg))/␣
↪float(n_neg)

    return fpr, tpr, thr
```

[57]:
```
# 随机猜测的预测值
preds = np.random.rand(len(y_test))
fpr, tpr, thr = roc_curve(y_test, preds)
plt.plot(fpr, tpr);
```



[58]:
```
preds = model.predict_proba(X_test)[:,1]
fpr, tpr, thr = roc_curve(y_test, preds)
plt.plot(fpr, tpr)
```

[58]: [<matplotlib.lines.Line2D at 0x183e5a28348>]



10.5 AUC(Area Under Curve)

AUC 是二分类问题的一个通用的平均指标，表示 ROC 曲线下面积。通常是 0.5-1 之间（乱猜就是 0.5）.

AUC 有个很好的特点：当测试集中的正负样本的分布变化的时候，ROC 曲线能够保持不变。因此很适合作为一些数据的评价指标（比如非平衡数据）.

[59]:
```
from sklearn.metrics import roc_auc_score
roc_auc_score(y_test, preds)
```

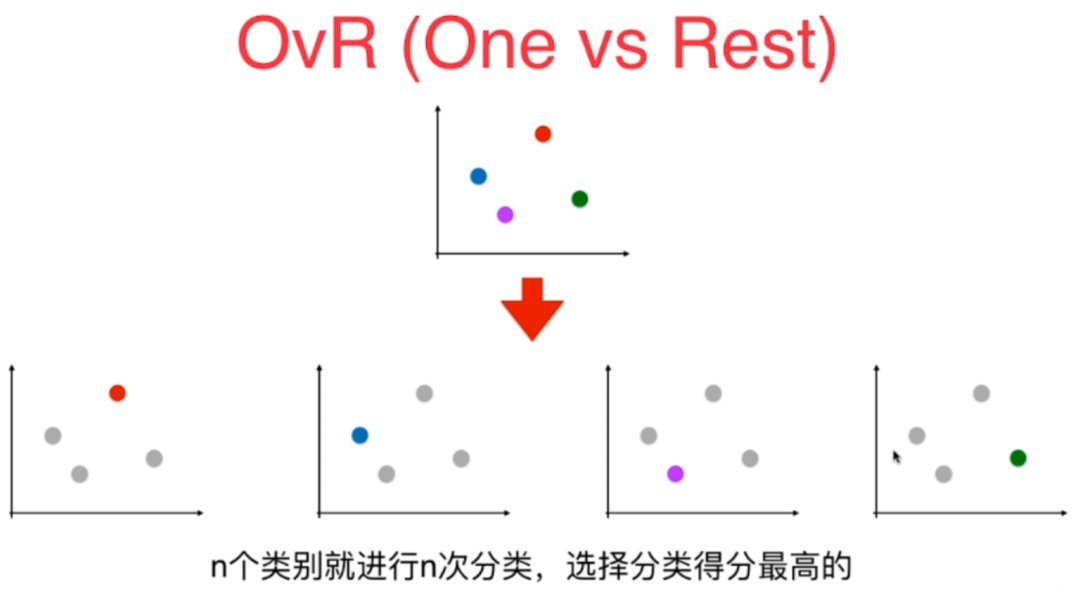
[59]: 0.9218527738264579

11 将二类分类算法改造成多类分类算法

对于类似逻辑回归的算法，只能解决二分类的问题，不过经过一定的改造便可以进行多分类问题，主要的改造方式有两大类：- OVR(One VS Rest) - OVO(One VS One)

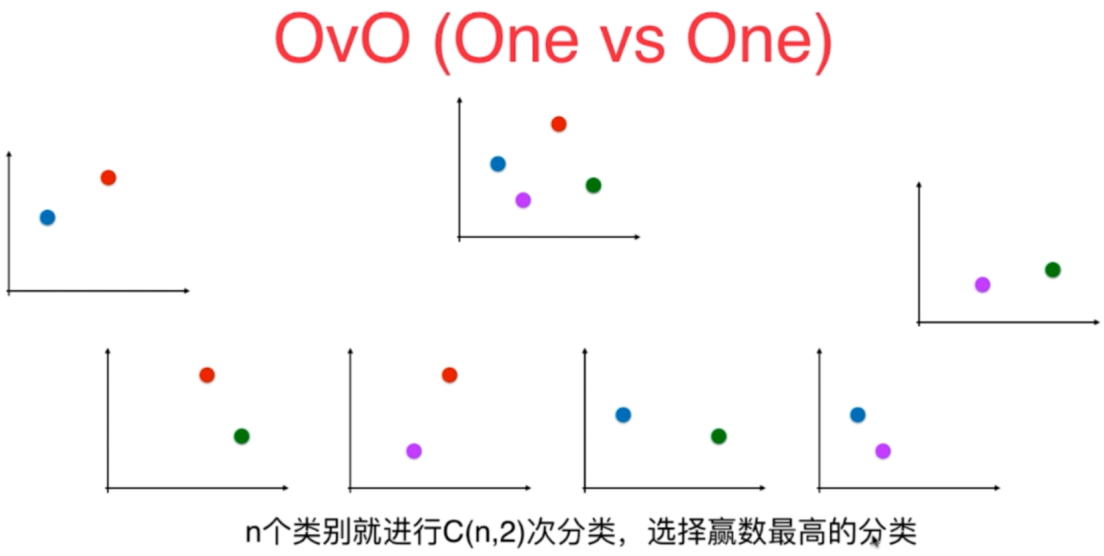
11.1 OVR

对于 OVR 的改造方式，主要是指将多个分类结果 (假设为 n) 分成是其中一种分类和其它分类的和，这样便可以有 n 种分类的模型进行训练，最终选择得分最高的类别（概率值最大）作为分类结果即可.



11.2 OVO

对于 OVO 的方式，主要是将 n 个数据分类结果任意两个进行组合，然后对其单独进行训练和预测，最终在所有的预测种类中赢数最高的即为分类结果，这样的分类方式最终将训练分为 $n(n-1)/2$ 个模型，计算时间相对较长，不过这样的方式每次训练各个种类之间不混淆也不影响，因此比较准确。



11.3 比较两种方式

[60]:
```
X = iris.data
y = iris.target
X_train,X_test,y_train,y_test = train_test_split(X,y,random_state=0)
```

[61]:
```
sc = StandardScaler()
sc.fit(X_train)
X_train_std = sc.transform(X_train) # standardize by mean & std
X_test_std = sc.transform(X_test)
```

对于 sklearn 中的 OvO 方式要注意的是，参数并不是 OvO，而是要指定 multi_class 为 multinomial，而且 solver 参数也必须指定为 newton-cg。

[62]:
```
lr = LogisticRegression(multi_class='multinomial',solver='newton-cg')
lr.fit(X_train_std,y_train)
lr.score(X_test_std,y_test)
```

[62]:
```
0.9736842105263158
```

[63]:
```
lr = LogisticRegression(multi_class='ovr',solver='newton-cg')
lr.fit(X_train_std,y_train)
lr.score(X_test_std,y_test)
```

[63]:
```
0.8947368421052632
```

可以发现 OVO 的准确率高于 OVR.