**Lab 02 基于Scikit-Learn的机器学习分类器实验指导**

**1 实验内容：**

学习学术界和工业界常用分类的监督学习算法，了解它们各自的优势和劣势。学会使用scikit-learn库，它为高效、有生产力地使用这些算法提供了用户友好且一致的接口。

* 分类算法，如逻辑回归、支持向量机、决策树和K-近邻算法
* 使用scikit-learn机器学习库
* 讨论线性和非线性决策边界分类器的优势和不足

选择分类算法

为具体的问题任务选择合适的分类算法需要实践和经验加持，每种算法都有自己的局限，是基于一些假设前提的。套用David H. Wolpert所提的没有免费午餐定理，没有哪个分类器对所有场景都表现最佳（《学习算法间缺少先验差别》，D.H. Wolpert, 1996: 1341-1390）。在实践中，推荐至少对比一部分线性算法的表现来选择具体问题的最佳模型，可能随特征或样本的不同、数据集中噪声的数量以及类是否线性可分割而不同。

最终分类器的表现-计算性能以及预测能力-极度依赖于底层供学习的数据。训练机器学习算法可概括为如下5个步骤：

1. 选择特征及收集打标签的训练样本
2. 选择性能指标
3. 选择学习算法及训练模型
4. 评估模型的表现
5. 修改算法的配置并对模型调优

本章集中讲解不同算法的核心概念，后续再详细讨论特征选取和预处理、性能指标及超参数调优。

**2 学习scikit-learn的第一步-训练感知机**

前面学习了Python和NumPy进行感知机学习规则和自适应线性神经网络。下面学习scikit-learn API，它将用户友好度与一致性接口和调度优化的多种分类算法实现相结合。scikit-learn库不止提供了大量的学习算法，还有很多的方便的函数进行数据预处理、调优及评估模型。

**2.1 基于scikit-learn库训练感知机模型**

在以下各小节中使用到的是已经很熟悉的鸢尾花数据集。为使用方便，scikit-learn中已经内置了鸢尾花数据集，因为它是测试和试验算法时常用的简单、知名数据集。为方便可视化我们只使用鸢尾花数据集中的两个特征。

我们将150个花样本的花萼长度和花瓣长度赋值给特征矩阵X，花品种的类标签则赋值给向量数组y：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | >>> 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] |

np.unique(y)函数返回iris.target中存储的三个唯一类标签，可以看到鸢尾花的类名Iris-setosa、Iris-versicolor和Iris-virginica，已经存储为了整数（此外为: 0, 1, 2）。虽然很多scikit-learn函数和类方法也能处理字符串格式的类标签，但更推荐使用整型标签，这样可以避免技术问题并因占用内存较小而提升计算性能，此外，将类标签编码为整型是机器学习库中常见的一种约定。

为评估训练对未知数据的表现如何，我们进一步将数据集分为训练数据集和测试数据集。使用scikit-learn中model\_selection模块的train\_test\_split函数，我们随机地将数组X和y分割成30%的测试数据（45个样本）和70%的训练数据（105个样本）：

|  |  |
| --- | --- |
| 1  2  3  4 | >>> 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  ... ) |

注意train\_test\_split函数已经**在分割前对训练数据集进行了内部打散**，否则，训练集中的所有样本都是类0和类1，而测试集会包含45个来自类2的样本。通过random\_state参数，我们为用于在分割前打散数据集的伪随机数生成器提供了**固定的随机种子**(random\_state=1) 。使用固定的random\_state可保证我们的结果可重现。

最后，我们通过stratify=y使用了对分层的内置支持。在这里，分层表示train\_test\_split方法**返回具有相同比例的类标签训练集和测试集**作为输入数据集。我们可以使用NumPy中用于统计数组中各个值出现次数的bincoun函数，进行检验：

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | >>> print('Labels counts in y:', np.bincount(y))  Labels counts in y: [50 50 50]  >>> print('Labels counts in y\_train:', np.bincount(y\_train))  Labels counts in y\_train: [35 35 35]  >>> print('Labels counts in y\_test:', np.bincount(y\_test))  Labels counts in y\_test: [15 15 15] |

很多机器学习和优化算法还要求使用特征缩放来优化性能。这里我们会使用scikit-learn中preprocessing模块的StandardScaler类来标准化特征：

|  |  |
| --- | --- |
| 1  2  3  4  5 | >>> 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) |

使用上述的代码，我们从preprocessing模块加载了StandardScaler类并初始化了一个新的StandardScaler对象，赋值给sc变量。使用fit方法，StandardScaler为训练数据中的每个特征维度评估了参数μ（样本均值）和σ（标准差）。然后通过调用transform方法，我们使用这些评估出的参数μ和σ标准化了训练数据。注意需要使用同样的缩放参数来标准化测试数据集，这样训练集和测试集中的值可进行比较。

标准化好了训练数据后，现在可以训练感知模型了。scikit-learn中的大部分算法**默认通过一对剩余(OvR)方法支持多类别分类**，这一方法允许我们同时喂入三种花的类别。代码如下：

|  |  |
| --- | --- |
| 1  2  3 | >>> from sklearn.linear\_model import Perceptron  >>> ppn = Perceptron(eta0=0.1, random\_state=1)  >>> ppn.fit(X\_train\_std, y\_train) |

scikit-learn接口在从linear\_model模块加载了Perceptron类后，初始化新的Perceptron对象并通过fit方法进行了训练。这里的模型参数eta0等价于我们自己实现感知机时使用的学习率eta。

如果学习率过大，算法会超出全局损失最小值。如果学习率过小，算法会需要更多次的迭代才能收敛，这会让学习变慢，尤其是对于大数据集。我们同样使用了random\_state参数来保证每次迭代后训练集的初始打散可重现。

在scikit-learn中训练好模型后，使用predict方法来完成预测。代码如下：

|  |  |
| --- | --- |
| 1  2  3 | >>> y\_pred = ppn.predict(X\_test\_std)  >>> print('Misclassified examples: %d' % (y\_test != y\_pred).sum())  Misclassified examples: 1 |

执行以上代码，我们会看到感知机对45个花样本的分类中有一个错误。因为，对测试集的分类误差率大约为0.022，或2.2%(1/45≈0.022)。

**2.2 分类误差与准确度**

很多机器学习从业者不报告分类误差，而是报告模型的准确度，计算方法如下：

1–*error* = 0.978, 或97.8%

使用分类误差还是准确度取决于个人偏好。

scikit-learn模块还在metrics模块中实现了大量的性能指标。例如，我们可以这样计算感知机对测试集的分类准确度：

|  |  |
| --- | --- |
| 1  2  3 | >>> from sklearn.metrics import accuracy\_score  >>> print('Accuracy: %.3f' % accuracy\_score(y\_test, y\_pred))  Accuracy: 0.978 |

这里，y\_test是真实类标签，而y\_pred是我们此前所预测的类标签。相对应的，scikit-learn中的**每个分类器都有一个score方法**，通过组合的predict调用和accuracy\_score来计算预测准确度，如下所示：

|  |  |
| --- | --- |
| 1  2 | >>> print('Accuracy: %.3f' % ppn.score(X\_test\_std, y\_test))  Accuracy: 0.978 |

**过拟合**

注意我们是根据本章中的测试数据集来计算模型的表现。后续我们会学习到一些有用的技术，包括图形分析，比如学习曲线来检测和防止过拟合。本章稍后还会提到过拟合，它的意思是模型对训练数据所提取的模式很好，但对未知数据的表现不好。

最后，使用plot\_decision\_regions函数来绘制新训练的感知机模型的决策区域，通过可视化的方式来看对花样本的分类表现如何。但我们来做一些小修改来通过小圆圈高亮显示测试集中的数据实例：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36 | 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 = ('o', 's', '^', '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))      lab = classifier.predict(np.array([xx1.ravel(), xx2.ravel()]).T)      lab = lab.reshape(xx1.shape)      plt.contourf(xx1, xx2, lab, alpha=0.3, cmap=cmap)      plt.xlim(xx1.min(), xx1.max())      plt.ylim(xx2.min(), xx2.max())      # plot class examples      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=f'Class {cl}',                      edgecolor='black')      # highlight test examples      if test\_idx:          # plot all examples          X\_test, y\_test = X[test\_idx, :], y[test\_idx]            plt.scatter(X\_test[:, 0], X\_test[:, 1],                      c='none', edgecolor='black', alpha=1.0,                      linewidth=1, marker='o',                      s=100, label='Test set') |

通过对plot\_decision\_regions函数所做的一点修改，我们指定了希望在结果图中标记的样本索引。代码如下：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | >>> 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.show() |

在结果图中可以看出，线性决策边界并不能很好分出三种花：

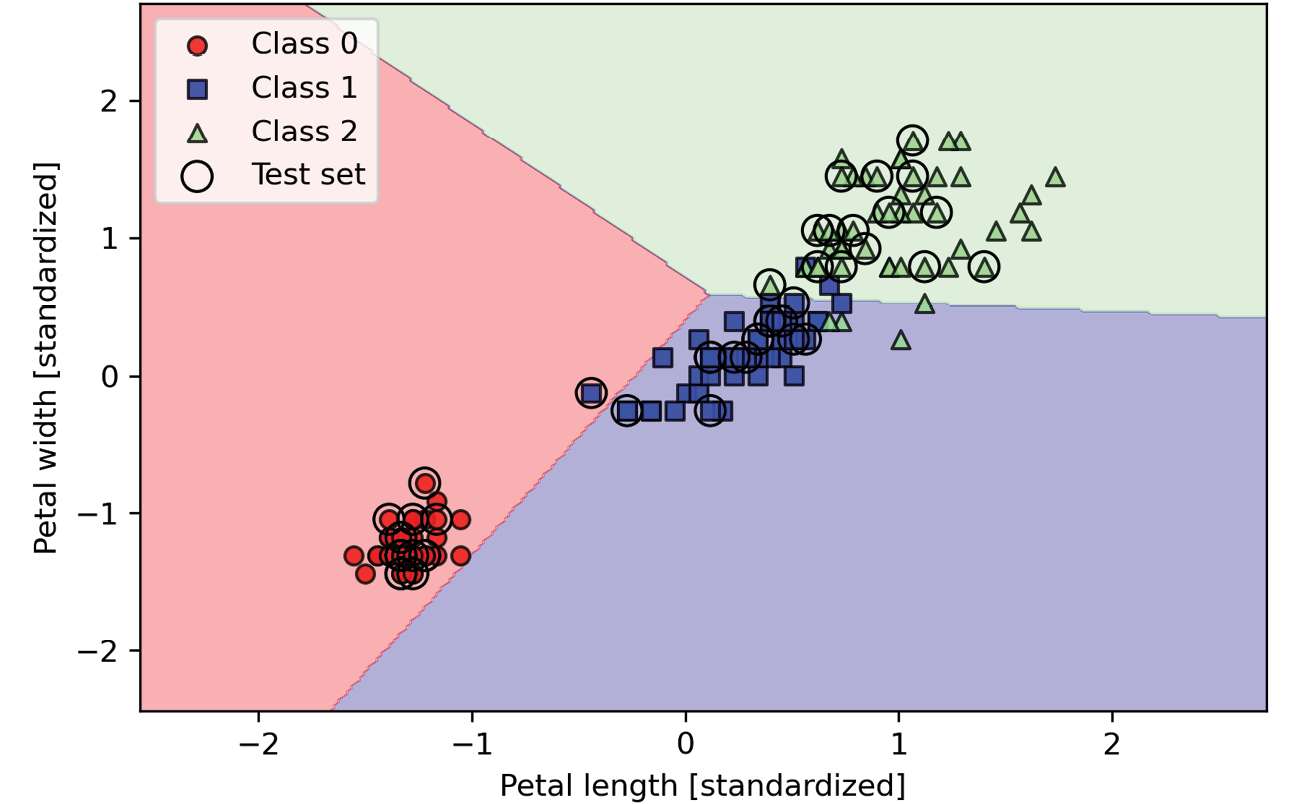


图3.1 多类别感知机拟合鸢尾花数据集后的决策边界

对无法完整线性分割的数据集感知机永远无法收敛，这也是为什么在实际应用中通常不推荐感知机算法。在下面的小节中，我们会学到更多强大的线性分类器，即使类无法完美地线性分割，也能收敛出损失最小值。

**2.3 其它感知机配置**

Perceptron，以及scikit-learn中的其它函数和类，通常一些为保持清晰而省略的参数。可以使用Python中的help函数来阅读更多这样的参数，也可以阅读scikit-learn优秀的在线文档[http://scikit-learn.org/stable/](http://scikit-learn.org/stable/" \o "" \t "_blank)。

**3 通过逻辑回归建模分类概率**

虽然感知机规则对机器学习分类算法是一个很好、很轻松的入门，但它有一个最大的缺陷是对非线性完美分割的类永远不会收敛。前一小节的分类任务就是这种场景的例子。这是由于每次迭代训练样本中都至少有一个分类错误导致权重一直在更新。当然我们可以修改学习率并增加迭代次数，但注意这一数据集的感知机永远不会收敛。

接下来学习另一个简单但更强大的解决**线性二元分类问题**的算法：逻辑回归（logistic regression）。注意虽然名字里带回归，但这是一个分类模型，而不是回归模型。

**3.1 逻辑回归和条件概率**

逻辑回归是一个易于实现且对线性可分割模型类性能很好的分类模型。它是业界最常使用的分类算法之一。类似于感知机和自适应线性神经网络，逻辑回归模型也是一个用于二分类的线性模型。

用于多类别的逻辑回归

线性逻辑回归可用于归纳多类场景，称为多类逻辑回归（multinomial logistic regression），或softmax回归。另一种将逻辑回归用于多类场景的方式是通过OvR技术。

为讲解逻辑回归作为二元分类概率模型背后的主要机制，我们先介绍几率（odds）：几率是指具体事件发生的比率。几率可写作p/(1−p)，其中*p*表示正事件（positive event）的概率。“正事件”不一定是好的，它指的是希望预测的事件，例如，有某种症状的病人患某一疾病的概念；可以把正事件想成是类标签*y* = 1，而症状是特征x。因此为保持简洁，可以将概率*p*定义为*p* := *p*(*y* = 1|x)，给定特征x时某一样本属于类1的条件概念。

然后我们可以进一步定义logit函数，它是对数发生比（*log-odds*）：



注意*log*指的是自然对数，这是计算机科学中常见的约定。*logit*函数接收0到1范围内的输入值，将它们转换为整个实数范围内的值。

在逻辑回归模型中，我们假设在加权输入和对数发生比之间存在线性关性：



虽然前面描述了对数发生比和净输入线性关联的假设，但我们实际感兴趣的是概率*p*，给定特性时样本的类成员概念。logit函数将概率与实际数字范围之间进行了映射，可以认为函数反过来就是实际数字范围对概率*p*的[0, 1]范围的映射。

这种logit的逆通常称为逻辑sigmoid函数，有时也因其典型的S形状简称为sigmoid函数：



这里的*z*是净输入，权重和输入的线性组合（也即与训练样本相关联的特征）：

*z* = wTx + *b*

下面我们来简单绘制-7到7之间一些值的sigmoid函数来看下效果：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | >>> 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)  >>> sigma\_z = sigmoid(z)  >>> plt.plot(z, sigma\_z)  >>> plt.axvline(0.0, color='k')  >>> plt.ylim(-0.1, 1.1)  >>> plt.xlabel('z')  >>> plt.ylabel('σ(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.show() |

执行以上示例代码后，我们会看到S形状的（sigmoidal）曲线：

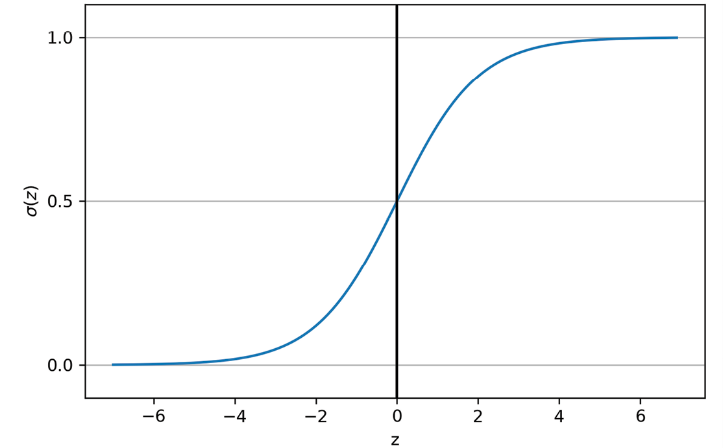


图3.2：逻辑sigmoid函数的图形

可以看到在*z趋向*无穷大(*z*→∞) 时σ(z)趋向于1，因为在*z*很大时*e*–z会变得非常小。类似地，*z*→–∞导致分母变大使用σ(z)接近于0。因此，可以总结为sigmoid函数接收实际数值作为输入，将它们转换为[0, 1]之间的值，偏置为σ(0)=0.5。

在Adaline中，我们使用了恒等函数σ(z)=z作为激活函数。在逻辑回归中，这一激活函数变成了我们前面所定义的sigmoid函数。

自适应感知神经元与逻辑回归的不同参见下图，其中唯一的不同之处就是激活函数：

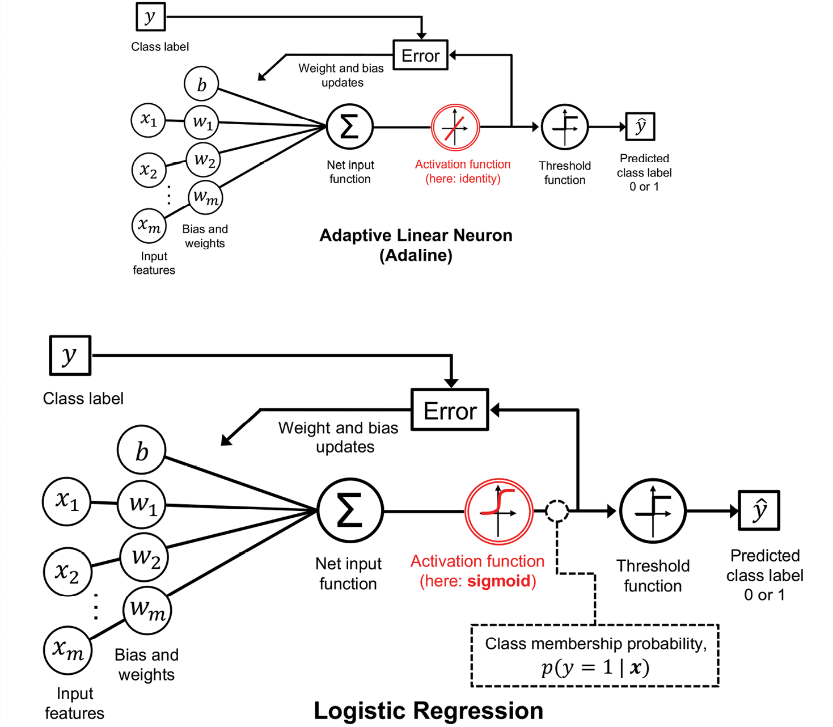
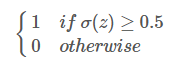


图3.3：逻辑回归与自适应感知神经元的对比

然后sigmoid函数的输出会翻译成具体样式属于类1的概率，σ(z)=p(y=1|x;w,b)，给定了特征*x*，并且通过权重*w*和偏置*b*进行参数化。例如，假如我们对具体样本花计算σ(z)=0.8，它表示样本是Iris-versicolor的概率是80%。因此，这朵花是Iris-setosa的概率可计算为p(y=0|x;w,b)=1−p(y=1|x;w,b)=0.2，或20%。

预测的概率可通过阈值函数转换为二元输出：



上图中的sigmoid函数等价于：



事实上，很多应用不仅对所预测的类标签感兴趣，而且类标签概率的评估也极为有用（应阈值函数前的sigmoid函数输出）。比如逻辑回归用于天气预报，不仅预测某一天是否下雨，还报告下雨的可能性。类似地，逻辑回归可用于预测有特定症状的病人患某种疾病的概率，这也是为什么逻辑回归在医药行业有很高的知名度。

**3.2 通过逻辑损失函数学习模型权重**

我们已经学习到可以通过逻辑回归模型预测概率和类标签，下面来简单讨论下如何拟合模型的参数，比如权重*w*和偏置单元*b*。上一章中我们定义了一个如下的方差损失函数：



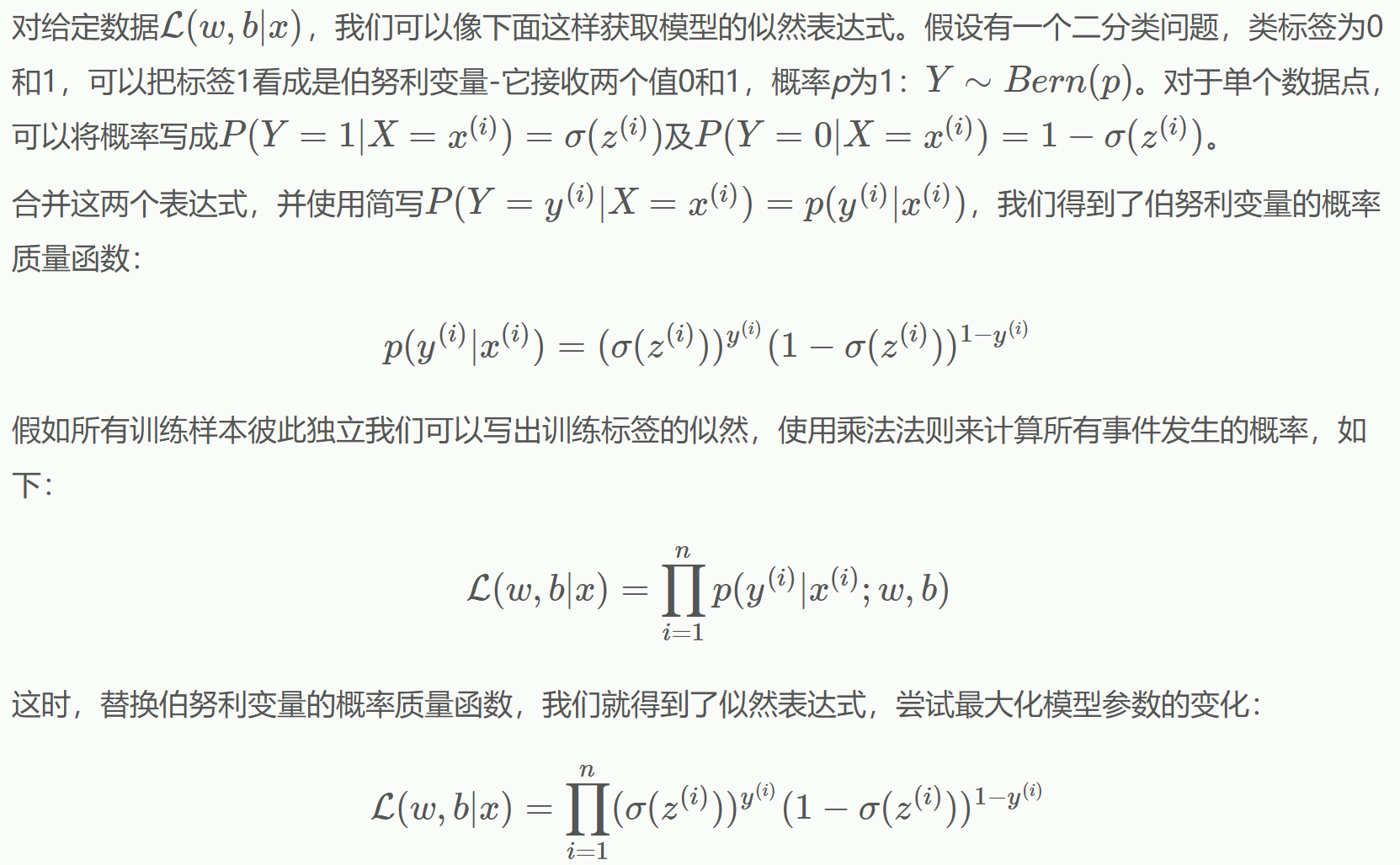
我们对函数进行了最小化以学习Adaline分类标签的参数。为讲解如何通过逻辑回归获取损失函数，我先定义似然L，假设数据集中的样本彼此并不关联，我们会希望构建一个逻辑回归模型使其最大化。公式如下：



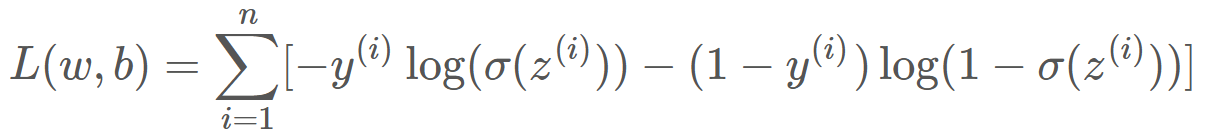
在实操中，最大化该等式的（自然）对数会更容易，称之为对数似然方程：



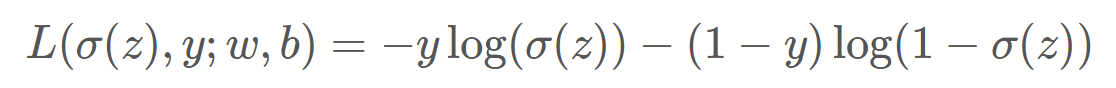
**推导似然函数**



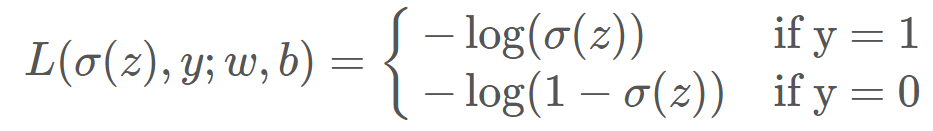
现在，我们可以使用梯度提升（gradient ascent）等优化算法来最大化该对数似然函数。（梯度提升梯度下降完全一样，只是梯度提升是将最小化替换为最大化函数。）我们这里将对数似然重写为损失函数，*L*，可使用梯度下降实现最小化：



为更好的掌握损失函数，我们来看我们为单训练样本所计算的损失：



从等式可以看出在如果*y* = 0则第一项为零，*y* = 1时第二项为零：



我们来写一个简短的代码脚本创建一张描绘分类针对σ(z)不同值的分类单训练样本损失的图：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | >>> def loss\_1(z):  ...     return - np.log(sigmoid(z))  >>> def loss\_0(z):  ...     return - np.log(1 - sigmoid(z))  >>> z = np.arange(-10, 10, 0.1)  >>> sigma\_z = sigmoid(z)  >>> c1 = [loss\_1(x) for x in z]  >>> plt.plot(sigma\_z, c1, label='L(w, b) if y=1')  >>> c0 = [loss\_0(x) for x in z]  >>> plt.plot(sigma\_z, c0, linestyle='--', label='L(w, b) if y=0')  >>> plt.ylim(0.0, 5.1)  >>> plt.xlim([0, 1])  >>> plt.xlabel('σ(z)')  >>> plt.ylabel('L(w, b)')  >>> plt.legend(loc='best')  >>> plt.tight\_layout()  >>> plt.show() |

绘出的图为范围在0到1之间*x*轴上sigmoid激活（对sigmoid函数的输入为在范围在-10到10之间的*z*值）以及*y*轴上关联的逻辑损失：

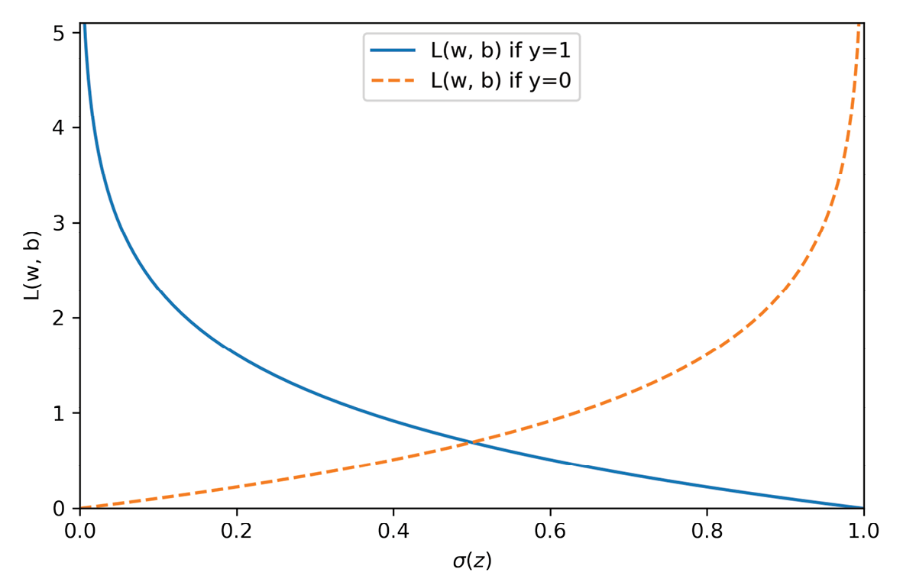
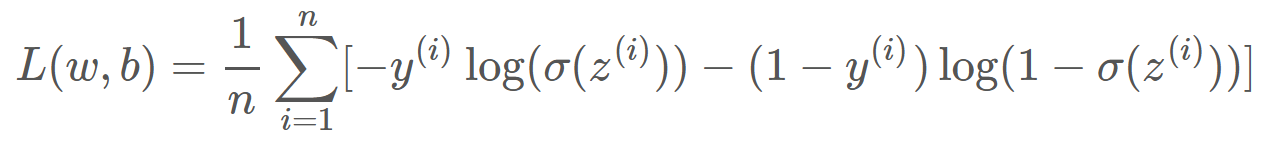


图3.4：使用逻辑回归的损失函数图

可以看到如果我们正确预测样本属于类1时损失接近0（实线）。类似地，可以看到在正确预测*y* = 0时*y*轴的损失接近于0（虚线）。但如果预测错误，损失会趋向无穷大。我们惩罚错误预测的要点是增大损失。

将Adaline 实现转化为逻辑回归算法

如果我们自己实现逻辑回归，可以将自适应线性神经网络实现的损失函数*L*替换为新的损失函数：



我们使用它计算每次迭代分类所有训练样本的损失。我们还需要将线性激活函数换成sigmoid。如对Adaline代码做出这些修改，就会得到一个可运行的逻辑回归实现。以下是对全批量梯度下降的实现（但注意也可对随机梯度下降版本做同样的修改）：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62 | class LogisticRegressionGD:      """Gradient descent-based logistic regression classifier.      Parameters      ------------      eta : float        Learning rate (between 0.0 and 1.0)      n\_iter : int        Passes over the training dataset.      random\_state : int        Random number generator seed for random weight        initialization.      Attributes      -----------      w\_ : 1d-array        Weights after training.      b\_ : Scalar        Bias unit after fitting.      losses\_ : list        Mean squared error loss function values in each epoch.      """      def \_\_init\_\_(self, eta=0.01, n\_iter=50, random\_state=1):          self.eta = eta          self.n\_iter = n\_iter          self.random\_state = random\_state      def fit(self, X, y):          """ Fit training data.          Parameters          ----------          X : {array-like}, shape = [n\_examples, n\_features]            Training vectors, where n\_examples is the            number of examples and n\_features is the            number of features.          y : array-like, shape = [n\_examples]            Target values.          Returns          -------          self : Instance of LogisticRegressionGD          """          rgen = np.random.RandomState(self.random\_state)          self.w\_ = rgen.normal(loc=0.0, scale=0.01, size=X.shape[1])          self.b\_ = np.float\_(0.)          self.losses\_ = []          for i in range(self.n\_iter):              net\_input = self.net\_input(X)              output = self.activation(net\_input)              errors = (y - output)              self.w\_ += self.eta \* 2.0 \* X.T.dot(errors) / X.shape[0]              self.b\_ += self.eta \* 2.0 \* errors.mean()              loss = (-y.dot(np.log(output))                     - ((1 - y).dot(np.log(1 - output)))                      / X.shape[0])              self.losses\_.append(loss)          return self      def net\_input(self, X):          """Calculate net input"""          return np.dot(X, self.w\_) + self.b\_      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.activation(self.net\_input(X)) >= 0.5, 1, 0) |

在拟合逻辑回归模型时，我们让铭记它只对二元分类任务有效。

因此我们只考虑山鸢尾和变色鸢尾（类0和1）并检查我产的逻辑回归实现是否有效：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | >>> X\_train\_01\_subset = X\_train\_std[(y\_train == 0) | (y\_train == 1)]  >>> y\_train\_01\_subset = y\_train[(y\_train == 0) | (y\_train == 1)]  >>> lrgd = LogisticRegressionGD(eta=0.3,  ...                             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.show() |

得到的决策区域图如下：

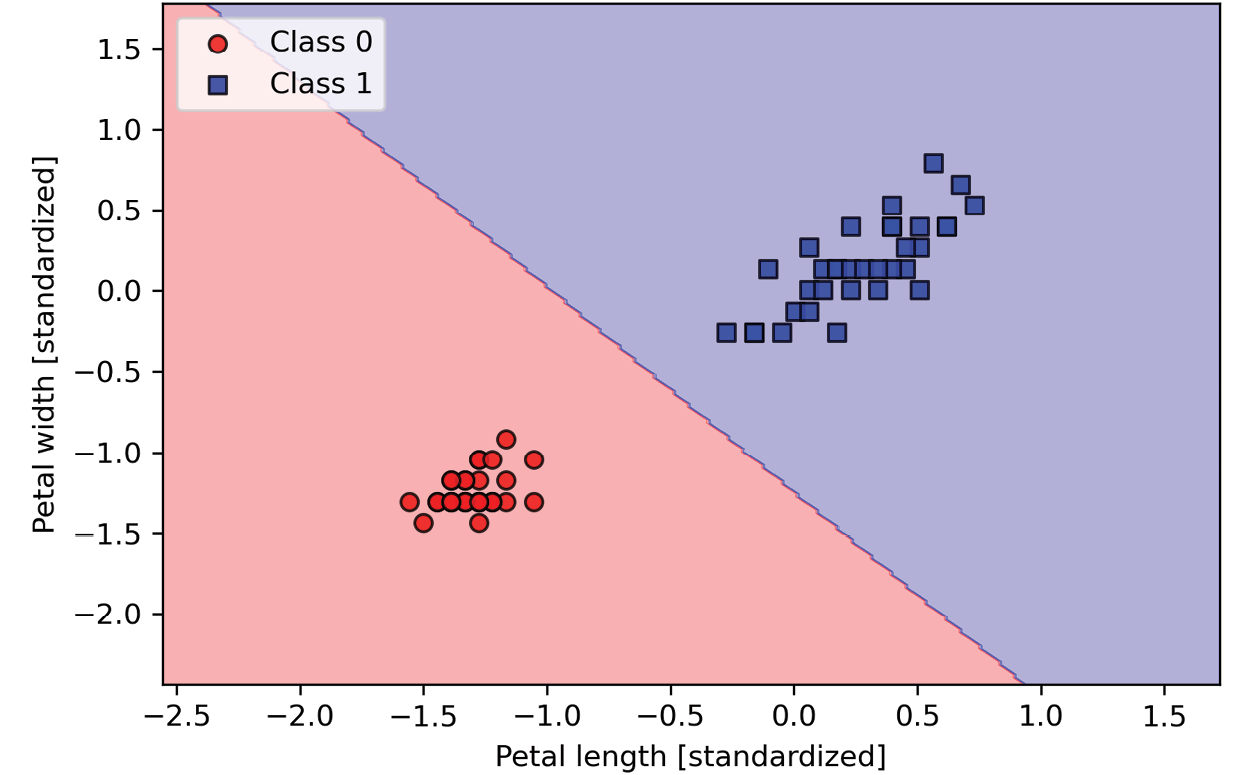


图3.5：逻辑回归模型的决策区域图

**3.3 针对逻辑回归的梯度下降学习算法**

如果对比LogisticRegressionGD与此前AdalineGD的代码，可能会注意权重和偏置规则保持不变（除了对第2个因子的缩放）。使用微积分，可以展示通过梯度下降的参数更新与逻辑回归和Adaline确实很像。但请注意下面对梯度下降学习规则的推导是为那些对逻辑回归梯度下降学习规则背后的数学概念感兴趣的读者准备的。

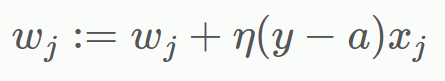
图3.6总结了如何计算对数似然函数对第*j*个参数的偏导数：



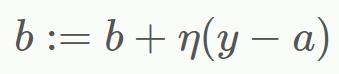
图3.6：计算对数似然函数的偏导数

这里为简洁起见省略了对训练样本求平均。

我们采取负梯度。因此调换了∂Lwj=−(y−a)xj并像下面这样更新了第 *j*个权重，包括学习率η:



虽然没有显示出损失函数对偏置的偏导数，使用链式法则的偏置推导整体概念一致，产生了如下的更新规则：



权重及偏置单元的更新与自适应线性神经网络一致。

**3.4 使用scikit-learn训练逻辑回归模型**

我们在前面的小节学习了一些有用的代码和数学练习，这有助于我们理解Adaline与逻辑回归概念上的不同。现在我们来学习使用scikit-learn对逻辑回归更优化的实现，它还内置支持多类别配置。请注意在scikit-learn的新版本中，用于多类别、多项因子或OvR的技术，是自动选择的。在下面的代码示例中，我们会使用sklearn.linear\_model.LogisticRegressio类以及熟悉的fit方法训练标准化花训练样本中所有三个类别的模型。同时，我们为方便绘图设置了multi\_class='ovr'。作为练习，可以使用multi\_class='multinomial'来比较结果。multinomial配置现在是scikit-learn的LogisticRegression类的默认选择并且在实践中也是互斥类的推荐选项，鸢尾花数据集就属于这种情况。这里的“互斥”表示每个训练样本只能属于一个分类（在多标签分类中，训练样本可以是多个分类的成员）。

下面我们来看示例代码：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | >>> from sklearn.linear\_model import LogisticRegression  >>> lr = LogisticRegression(C=100.0, solver='lbfgs',  ...                         multi\_class='ovr')  >>> 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.show() |

在对训练数据拟合模型后，我们绘制了决策区域、训练样本和测试样本，如图3.7所示：

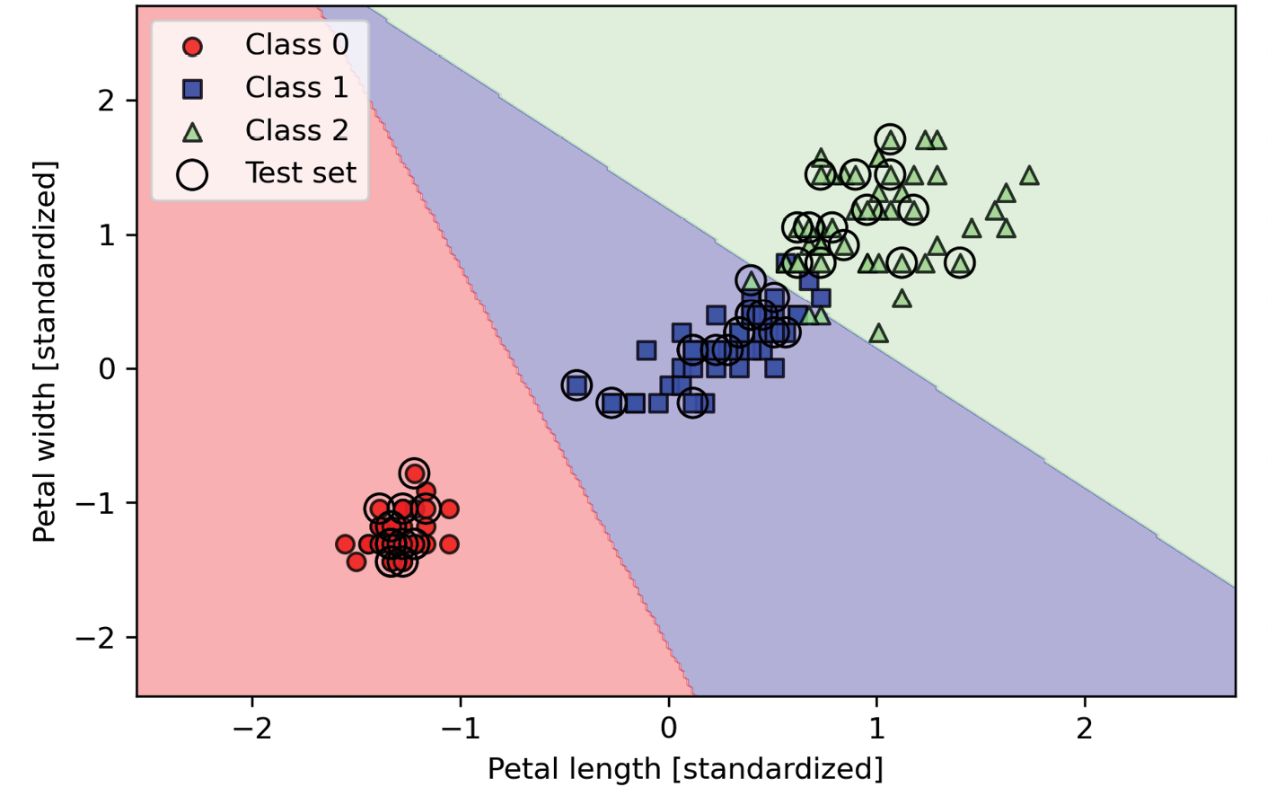


图3.7：scikit-learn多类逻辑回归模型的决策区域

凸优化算法

现存有很多算法可解决优化问题。为最小化凸损失函数，比如逻辑回归损失，推荐使用比普通的随机梯度下降 (SGD)更高级的方法。事实上，scikit-learn实现了非常多的这类算法，可通过solver参指定，其中有'newton-cg'、'lbfgs'、'liblinear'、'sag'和'saga'。

虽然逻辑回归损失是凸损失，但大部分算法应该都可以轻松对全局最小损失收敛。然后，使用其中一些算法相对另一些有某些优势。比如，在之前的版本中（如v 0.21），scikit-learn默认使用'liblinear'，它无法用于多项损失（multinomial loss），仅限于针对多类分类的OvR模式。但在scikit-learn v 0.22中将默认算法改为了'lbfgs'，表示内存限定Broyden–Fletcher–Goldfarb–Shanno (BFGS)算法，在这方面要更为灵活。

来看前面我们用于训练LogisticRegression模型代码，读者可能会想，“神秘的参数C是什么？”我们会在下一小节中讨论这一参数，其中会介绍过拟合与正则化的概念。但在进行这些讨论之前，先完成对类成员概率的讨论。

训练样本属于某一类的概率可使用predict\_proba方法计算。例如，我们可以这样预测测试集前三个样本的概率：

|  |  |
| --- | --- |
| 1 | >>> lr.predict\_proba(X\_test\_std[:3, :]) |

这段代码返回如下数组：

|  |  |
| --- | --- |
| 1  2  3 | array([[3.81527885e-09, 1.44792866e-01, 8.55207131e-01],         [8.34020679e-01, 1.65979321e-01, 3.25737138e-13],         [8.48831425e-01, 1.51168575e-01, 2.62277619e-14]]) |

第一行对应第一朵花的类成员概率，第二行对应第二朵花的类成员概率，以此类推。各行列的加和是1。（可以通过执行lr.predict\_proba(X\_test\_std[:3, :]).sum(axis=1)来进行确认。）

第一行的最高值大约是0.85，表示对第一个样本属于类3(Iris-virginica) 预测的概率是85%。读者可能已经注意到，我们可以通过找到每行中的最大列来预测类标签，比如使用NumPy的argmax函数：

|  |  |
| --- | --- |
| 1 | >>> lr.predict\_proba(X\_test\_std[:3, :]).argmax(axis=1) |

返回的类标签如下所示（分别对应Iris-virginica、Iris-setosa和Iris-setosa）：

|  |  |
| --- | --- |
| 1 | array([2, 0, 0]) |

在上面的示例代码中，我们计算了条件概率并手动使用NumPy的argmax函数将它们转化为类标签。实操中，使用scikit-learn获取类标签更便捷的方式是直接调用predict方法：

|  |  |
| --- | --- |
| 1  2 | >>> lr.predict(X\_test\_std[:3, :])  array([2, 0, 0]) |

最后，如果希望预测单个花样本的类标签需要注意：scikit-learn接收的数据输入是二维数组，因此，我们需要将单行切片先转化为这种格式。将单行输入转化为二维数组的一种方式是使用NumPy的reshape方法来添加一维，如下所示：

|  |  |
| --- | --- |
| 1  2 | >>> lr.predict(X\_test\_std[0, :].reshape(1, -1))  array([2]) |

**3.5 通过正则化处理过拟合**

过拟合是机器学习中一种常见问题，这时模型对训练数据表现很好但对于未知数据（测试数据）的归纳不好。如果模型出现了过拟合，我们还会说模型的方差很高，这可是由参数过多产生的，导致模型对于给定的底层数据过于复杂。类似地，模型也可能会出现欠拟合（高偏差），这表示模型不够复杂未能很好地提取训练数据中的模式，因此对未知数据的表现不佳。

虽然我们还只遇到了线性分类模型，但过拟合和欠拟合的问题可通过比较线性决策边界与更复杂的非线性决策边界来更好的描绘，如图3.8所示：

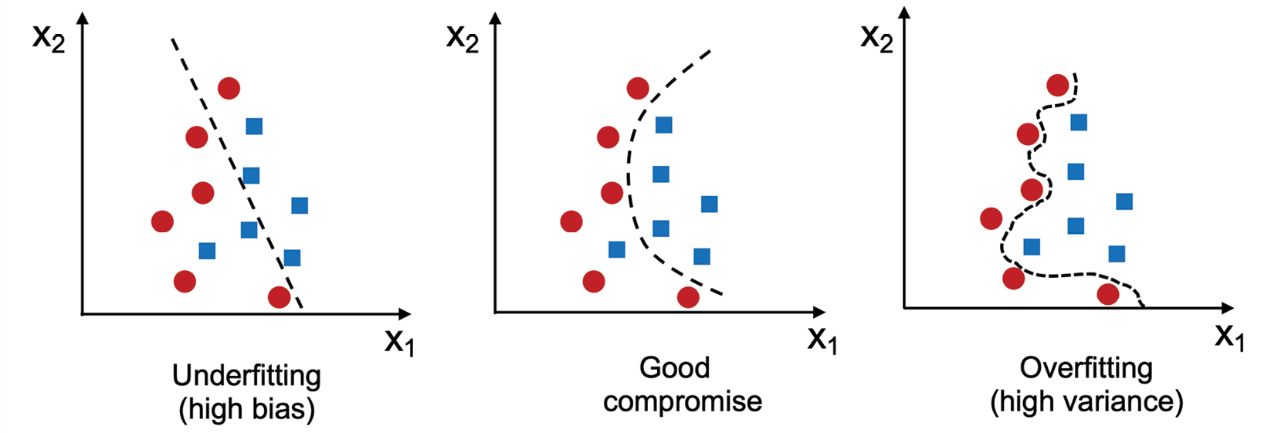


图3.8：欠拟合、良好拟合和过拟合模型示例

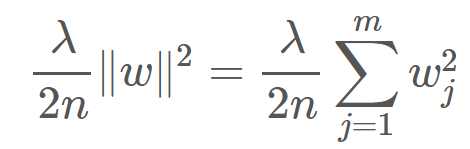
偏差-方差均衡

通常研究人员使用“偏差”和“方差”或“偏差-方差均衡”来描述模型的性能，也就是说读者在演讲、书或文章中可能会碰到说模型是“高方差”或“高偏差”的。那这是什么意思呢？一般我们会说“高方差”与过拟合成正比，而“高偏差”与欠拟合成正比。

在机器学习模型的上下文中，方差度量如果多次重新训练模型时对某一样本分类的模型预测的一致性（或可变性），比如对训练集的不同子集进行训练。我们可以说模型对训练数据中随机性是敏感的。相反，偏差度量在对不同训练数据集多次重建模型时预测距正确值总体有多远，偏差是一种与随机性无关的系统误差的度量。

一种发现好的偏差-方差均衡的方式是通过正则化调优模型的复杂度。正则对于处理共线性（特征高度关联）、过滤数据噪声及最终防止过拟合是一种非常有用的方法。

正则化背后的概念是引入了额外的信息来惩罚极端参数（权重）值。正则化最常见的形式称L2正则化（有时也称为L2收缩或权重衰减），可以写成下面这样：



这里的λ称为正则化参数。分母里的2只是一个缩放因子，这样在计算损失梯度时约去。类似损失添加了样本大小 *n*来缩放正则项。

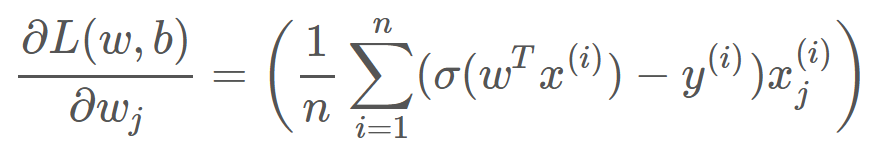
正则化和特征归一化

**正则化是标准化这样的特征缩放很重要的另一个原因**。为让正则化正常动作，我们需要保障所有的特征在可比较的量级上。

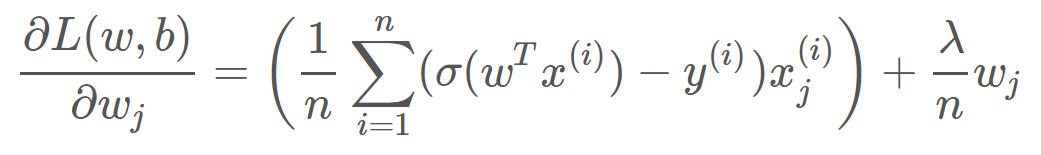
逻辑回归的损失函数可通过添加简单正则项来正则化，正则项会在模型训练时收缩权重：



未正则化的偏导数定义为：



对损失添加正则项会将这一偏导数变成如下的形式：



通过正则化参数，λ，我们可以控制拟合训练数据的紧密程度，同时也保持权重很小。通过增大λ的值，我们增加了正则化强度。请注意学习过的偏置单元，基本是截距项或负阈值，通常不做正则化。

scikit-learn中为LogisticRegression类实现的参数C，源自支持向量机的约定。C项与正则化参数λ成反比。因此，减少其值可会翻转正则化参数C，也即增加正则化的强度，可通过绘制有两个权重系数的L2正则化路径来进行可视化：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | >>> weights, params = [], []  >>> for c in np.arange(-5, 5):  ...     lr = LogisticRegression(C=10.\*\*c,  ...                             multi\_class='ovr')  ...     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.show() |

通过执行以上代码，我们使用用于翻转正则化参数C的不同值拟合了10个逻辑回归模型。为便于绘图，我们只收集了类1（这里是数据集中的第二个类：Iris-versicolor）对所有分类器的权重系数，还记得我们使用OvR技术来进行多类分类吧。

可以通过输出图看出，如果减小参数C也即增加正则化强度则权重系数收缩：

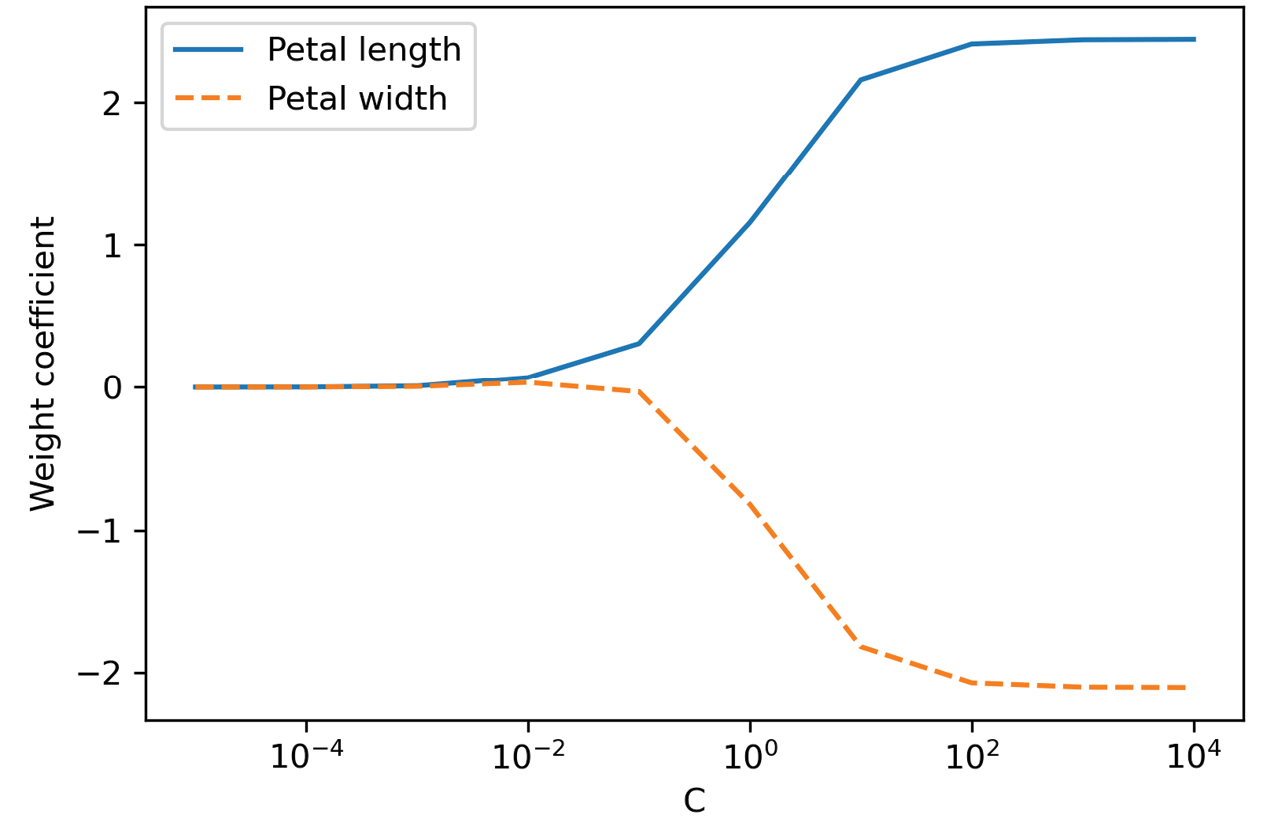


图3.9：翻转L2正则化模型结果上的正则化强度参数C的影响

增加正则化强度会减小过拟合，那么读者可能会问为什么不默认对所有模型进行强正则化呢？原因是在调整正则化强度时也格外小心。例如，如果**正则化强度过高而权重系统接近于零，模型会因欠拟合而表现很差**，参见图3.8。

**4 支持向量机实现最大间隔分类**

另一种强大又广泛使用的学习算法是支持向量机(SVM)，可看成是对感知机的扩展。使用感知机算法，我们最小化误分类错误。但在SVM中，我们的优化目标是最大化间隔（margin）。间隔定义为分隔的超平面（决策边界）之间的距离，距离超平面最近的训练样本称为支持向量。

如图3.10所示：

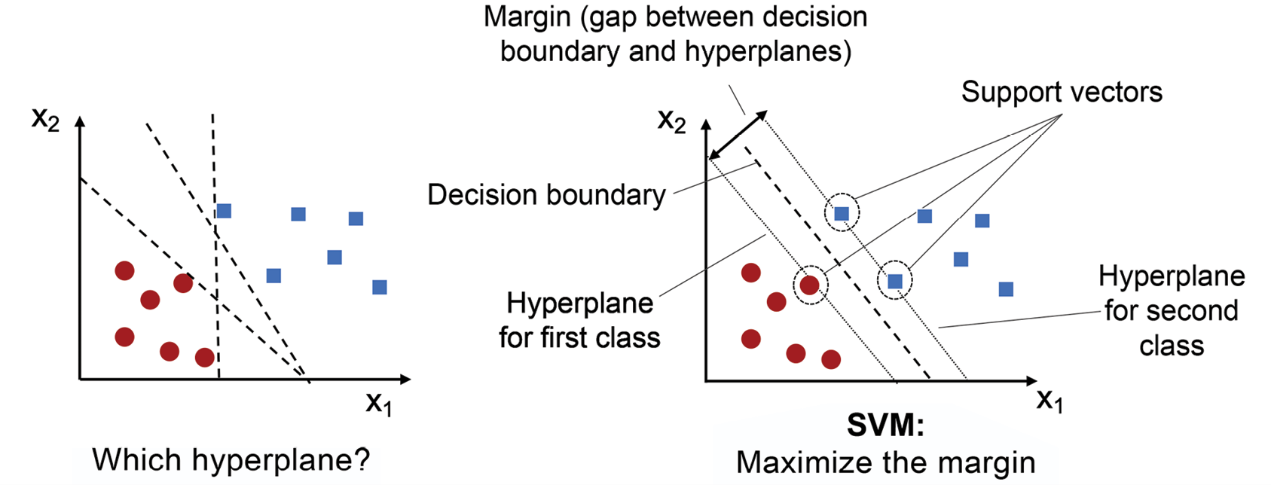


图3.10：SVM最大化决策边界与训练数据点之间的间距

**4.1 最大间隔构想**

让决策边界具有大间隔背后的根本原因是那样会趋向更低的泛化错误，而具有小间隔的模型更容易出现过拟合。虽然SVM背后的主要构想相对简单，但不幸的是背后的数据很高阶，要求对约束最优化有很好的掌握。

使用松弛变量处理线性不可分割案例

虽然我们不希望深度介入最大间隔分类背后的数据概念，但还是要简单讲解一下松弛变量（*slack variable*），它由Vladimir Vapnik于1995年引入并产生了所谓的软间隔分类（soft-margin classification）。引入松弛变量的动机是SVM优化目标中的线性约束需要松弛线性不可分割数据来在出现错误分类时使用合适的损失惩罚实现优化的收敛。

而松弛变量的使用，又引入了SVM上下文中经常被称为*C*的变量。可以把*C*看成是控制错误分类处罚的超参数。大值的*C*对应大的错误惩罚，而如果选择小值的*C*则表示对错误分类误差没那么严格。然后我们可以使用参数*C*来控制间隔宽度，因而调优偏差-方差均衡，如图3.11所示：

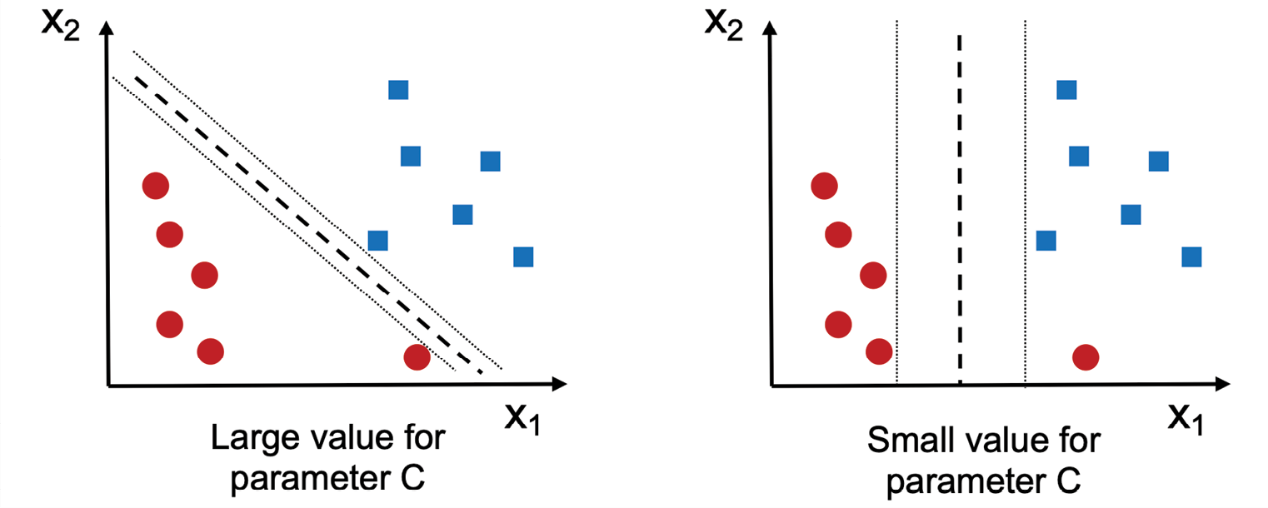


图3.11：分类上翻转正则化强度*C*大小值的影响

这一概念与正则化相关，我们在前一节的正则化回归上下文中讨论过，减小C的值会增加偏差（欠拟合）及降低模型的方差（过拟合）。

既然我们已经学习了线性支持向量机背后的基本概念，下面来训练SVM模型对鸢尾花中不同的花：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | >>> 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.show() |

SVM的三个决策区域，在执行以上示例代码对鸢尾花训练分类器可视化为图3.12：

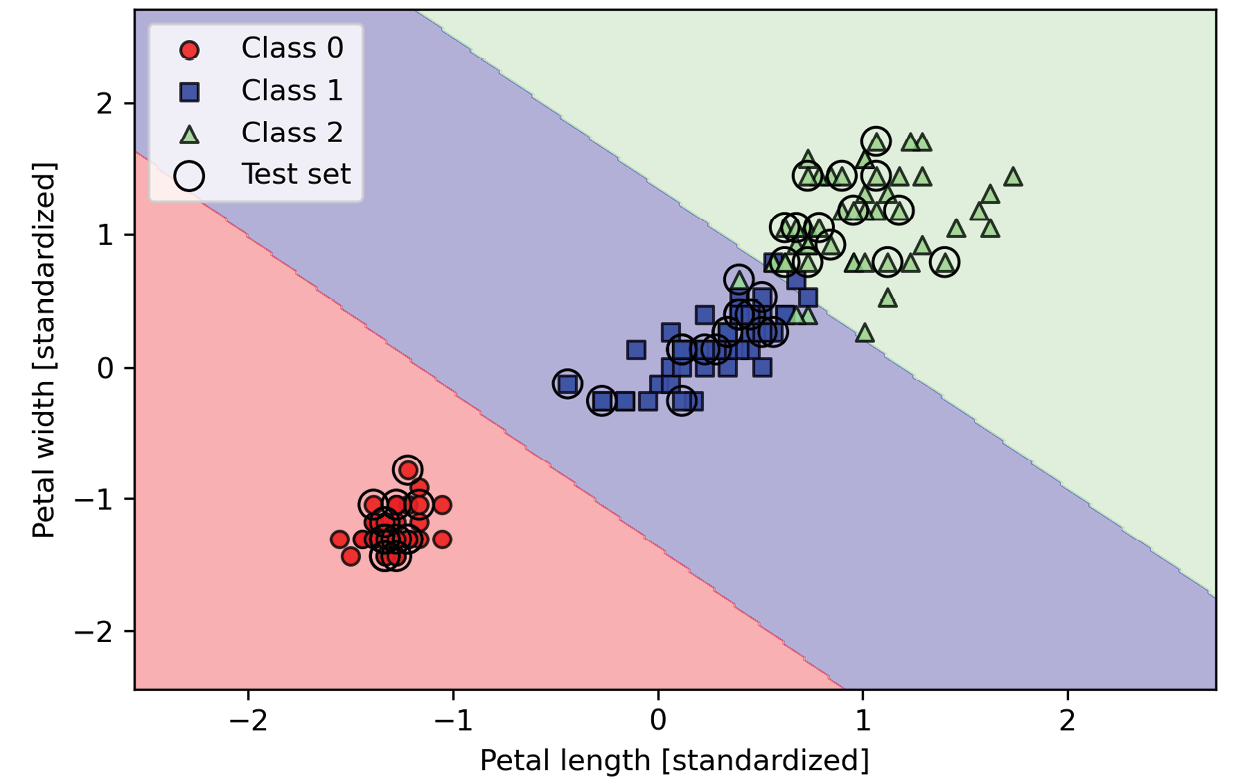


图3.12：SVM的决策区域

**逻辑回归 vs. SVM**

在实际分类任务中，线性逻辑回归与线性SVM经常产生非常相似的结果。逻辑回归尝试最大化训练数据的条件似然，这会比SVM更容易接近异常值，后者更关心的是最接近决策边界（支持向量）的那些点。另一方面，逻辑回归的优势是模型更简单、实现更容易，并且在数学上也更容易解释。此外，逻辑回归模型可以很易于更新，这对于处理流式数据很有吸引力。

scikit-learn中的替代实现

上一节中我们使用的scikit-learn库的LogisticRegression类，可通过设置solver='liblinear'来利用LIBLINEAR库。LIBLINEAR是台湾大学开发的高度优化的C/C++库(<http://www.csie.ntu.edu.tw/~cjlin/liblinear/>)。

类似地，用于训练支持向量机的SVC类利用了LIBSVM，这是一个等价的专门用于SVM的C/C++库(<http://www.csie.ntu.edu.tw/~cjlin/libsvm/>)。

使用LIBLINEAR和LIBSVM的优势，比如相比原生Python实现来说，它们使训练大量的线性分类器变得极其快速。但有时数据集过大电脑内存无法容纳。因此scikit-learn还提供了通过SGDClassifier类的一些替代实现，该类还支持通过partial\_fit方法进行的在线学习。SGDClassifier背后的概念类似于Adaline所实现的随机梯度算法。

我们可以初始化SGD版本的感知机(loss='perceptron')、逻辑回归(loss='log')及带默认参数的SVM(loss='hinge')，如下：

|  |  |
| --- | --- |
| 1  2  3  4 | >>> from sklearn.linear\_model import SGDClassifier  >>> ppn = SGDClassifier(loss='perceptron')  >>> lr = SGDClassifier(loss='log')  >>> svm = SGDClassifier(loss='hinge') |

**4.2 使用核函数SVM解决非线性问题**

SVM在机器学习从业者中知名度很高的另一个原因是，很容易将它们核化（kernelized）来解决非线性分类问题。在讨论最常见SVM的变体核SVM背后的主要概念之前，我们先创建一个人工数据集来看这种非线性分类问题是什么样的。

线性不可分割数据的核方法

使用如下代码，我们会使用NumPy中的logical\_xor函数创建异或门（XOR）形式的简单数据集，其中100个样式会被赋类标签1，100个样本会被赋类标签-1：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | >>> 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, 0)  >>> plt.scatter(X\_xor[y\_xor == 1, 0],  ...             X\_xor[y\_xor == 1, 1],  ...             c='royalblue', marker='s',  ...             label='Class 1')  >>> plt.scatter(X\_xor[y\_xor == 0, 0],  ...             X\_xor[y\_xor == 0, 1],  ...             c='tomato', marker='o',  ...             label='Class 0')  >>> plt.xlim([-3, 3])  >>> plt.ylim([-3, 3])  >>> plt.xlabel('Feature 1')  >>> plt.ylabel('Feature 2')  >>> plt.legend(loc='best')  >>> plt.tight\_layout()  >>> plt.show() |

执行这段代码，我们会得到一随机噪声的异或数据集，如图3.13所示：

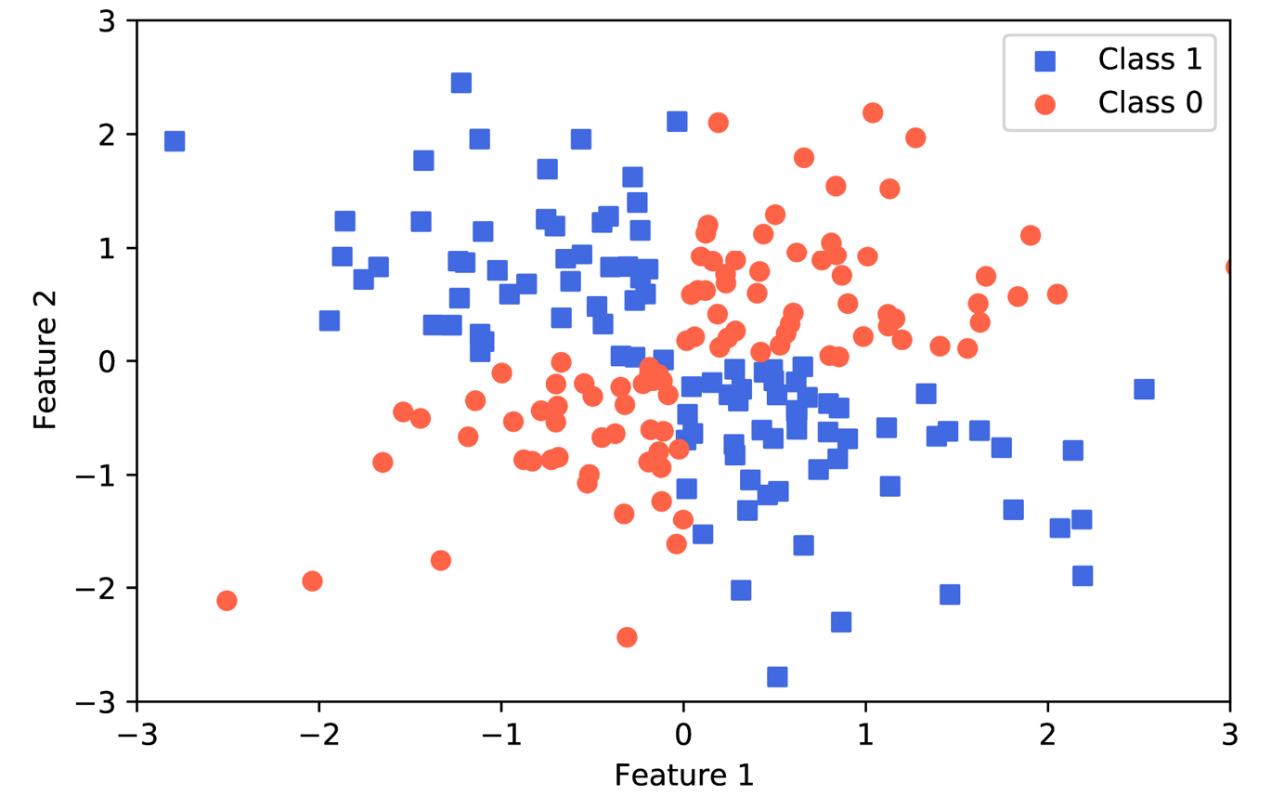
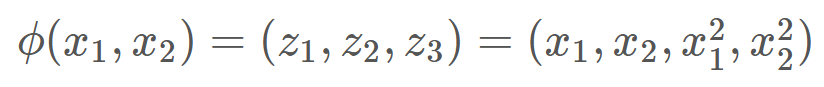


图3.13: 与或数据集绘图

很明显，我们无法通过之前小节中讨论的线性逻辑回归或线性SVM模型使用线性超平面作为决策边界去分隔正类和负类。

核方法处理这种线性不可分割数据的基本思想是创建原始特征的非线性组合来通过映射函数ϕ投射到更高维度的空间，在其中数据变得线性可分割。如图3.14所示，我们可以将二维数据集转换成一个新的三维特征空间，其中的分类经如下的投射变得可分割：



这使用得我们可以通过线性超平面分割这两类，如果将该超平面投射回原特征空间，会得到一个如下图同心圈数据集所绘制的非线性决策边界：

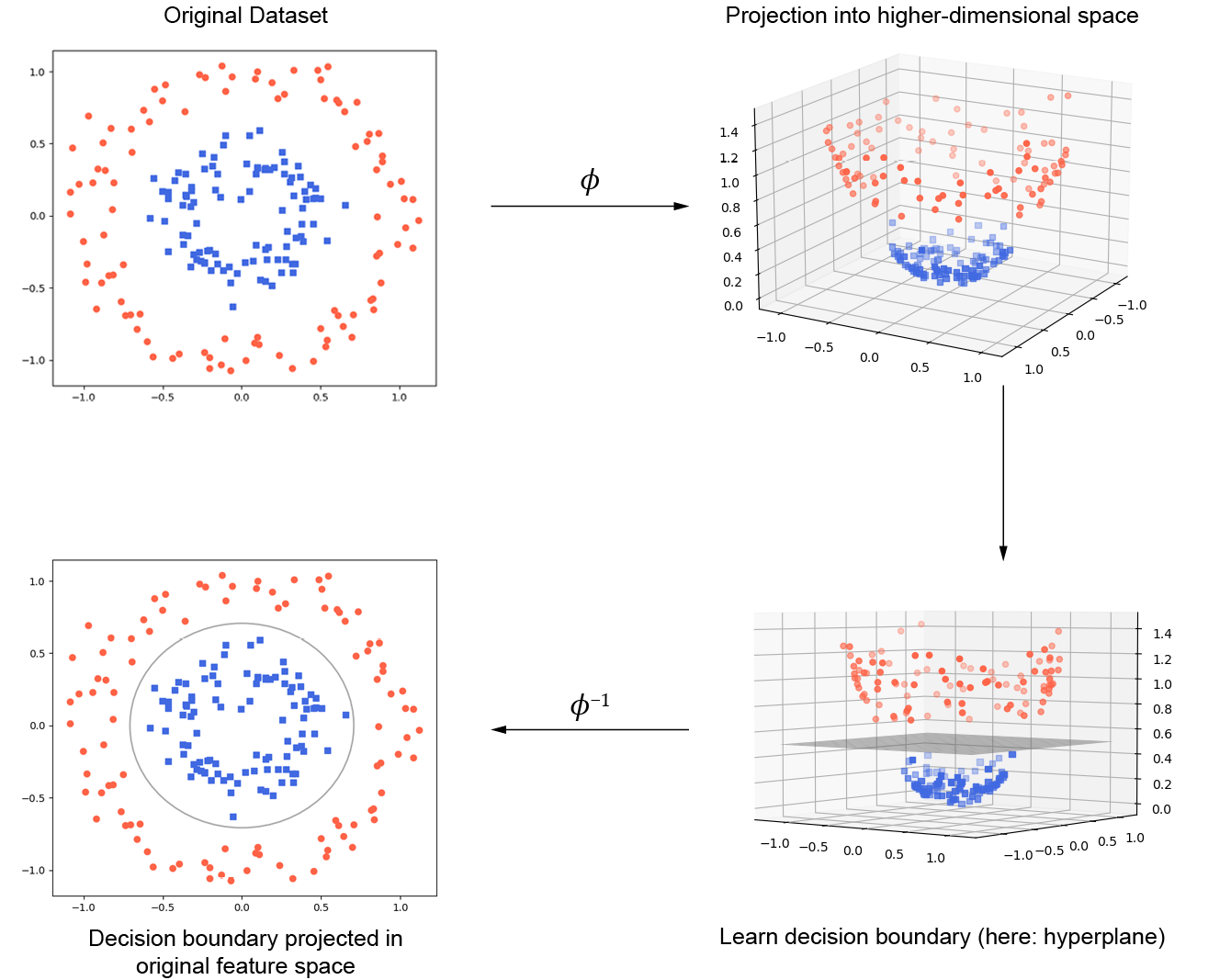


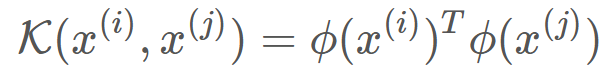
图3.4：使用核方法分类非线性数据的流程

使用核技巧在高维度平台中寻找分割超平面

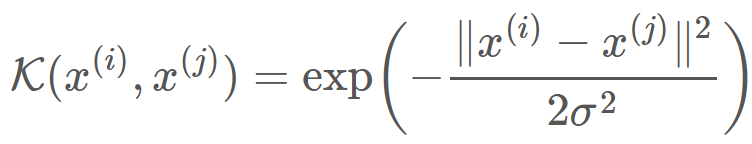
要使用SVM解决非线性问题，我们会通过映射函数ϕ将训练数据转换成更高维度的特征空间，并训练一个线性SVM模型来对新特征空间中的数据分类。然后我们可以使用同一个映射函数ϕ来转换新的未知数据，使用线性SVM模型对其分类。

但这种映射方法有一个问题，**就是构建新特征计算开销大**，处理高维度数据时尤其如此。所以就出现了所谓的核技巧（kernel trick）。

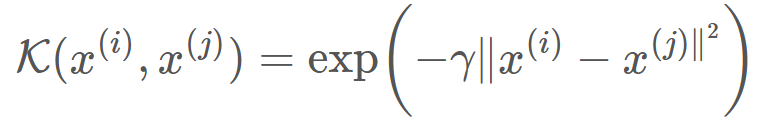
我们没有深入讲如何解二次规划（quadratic programming）任务来训练SVM，在实操时，我们只需要通过ϕ(x(i))Tϕ(x(j))替换点乘x(i)Tx(j) 。为省去显式计算两点间点乘的大开销步骤，我们定义了核函数：

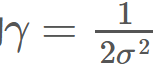


最广泛使用的核函数之一是径向基函数（radial basis function (RBF)）核，可以简单地称为高斯核（Gaussian kernel）：



常简化为：



这里的是一个可优化的超参数。

精略地说，可将“核”看成是样本对之间的相似度函数，负号将距离度量反转为相似度得分，而由于指数项的原因，得到的相似度分数会在1（完全相似的样本）和0（极其不相似的样本）之间。

我们讲解了核技巧背后的全局，下面来看是否能训练出绘制很好地分割异或数据的非线性决策边界。这里我们使用了之前导入的scikit-learn的SVC类并将参数kernel='linear替换为kernel='rbf'：

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | >>> 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.show() |

从结果图可以看出，核SVM相对较好地分割了异或数据：

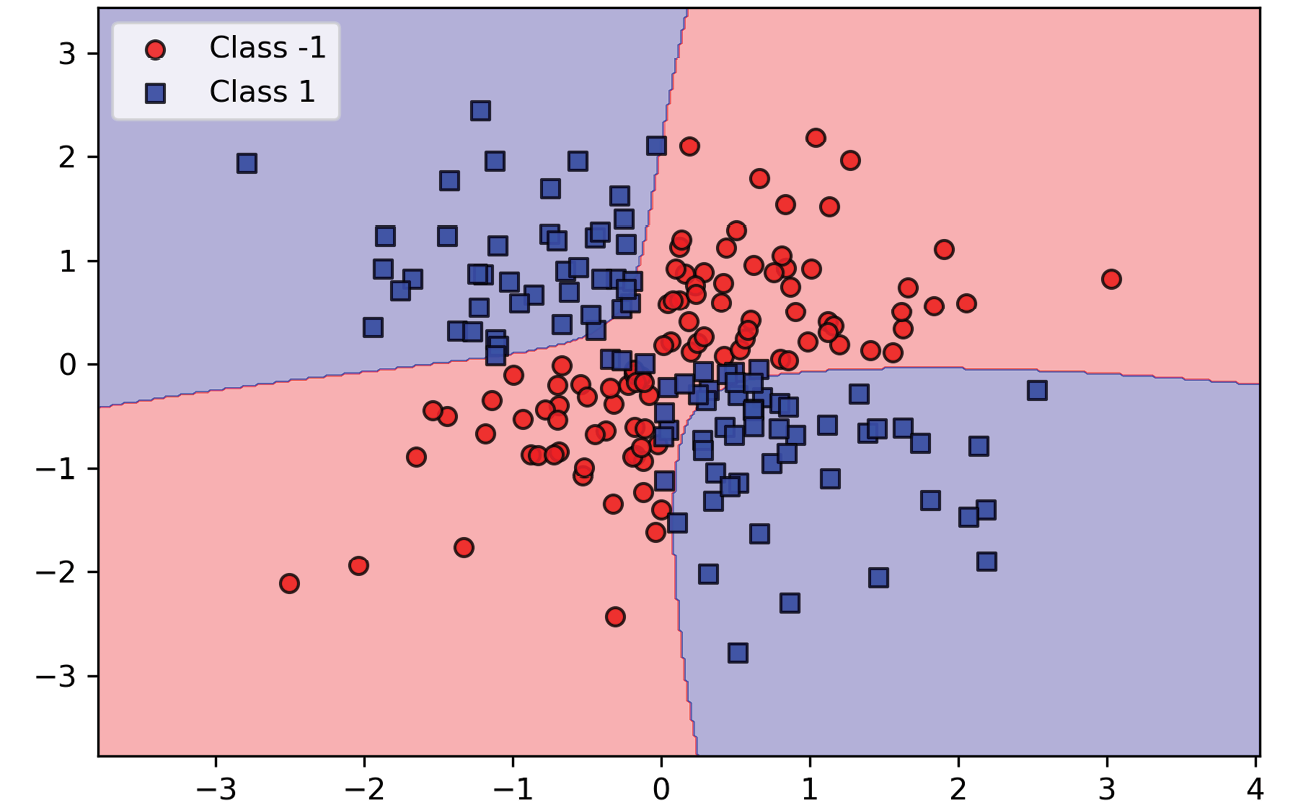


图3.15：使用核方法得到的异或数据决策边界

参数γ，我们设置为了gamma=0.1，可以理解为高斯球的临界参数。如果我们增加γ的值，就会增加训练样本的影响或范围，这会产生更紧密、更不平滑的决策边界。为更好理解γ，我们来对鸢尾花数据集应用RBF核SVM：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | >>> 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.show() |

因为我们为γ选择了一个相对较小的值，产生的径向基函数核SVM模型的决策边界相对柔和，如图3.16如示：

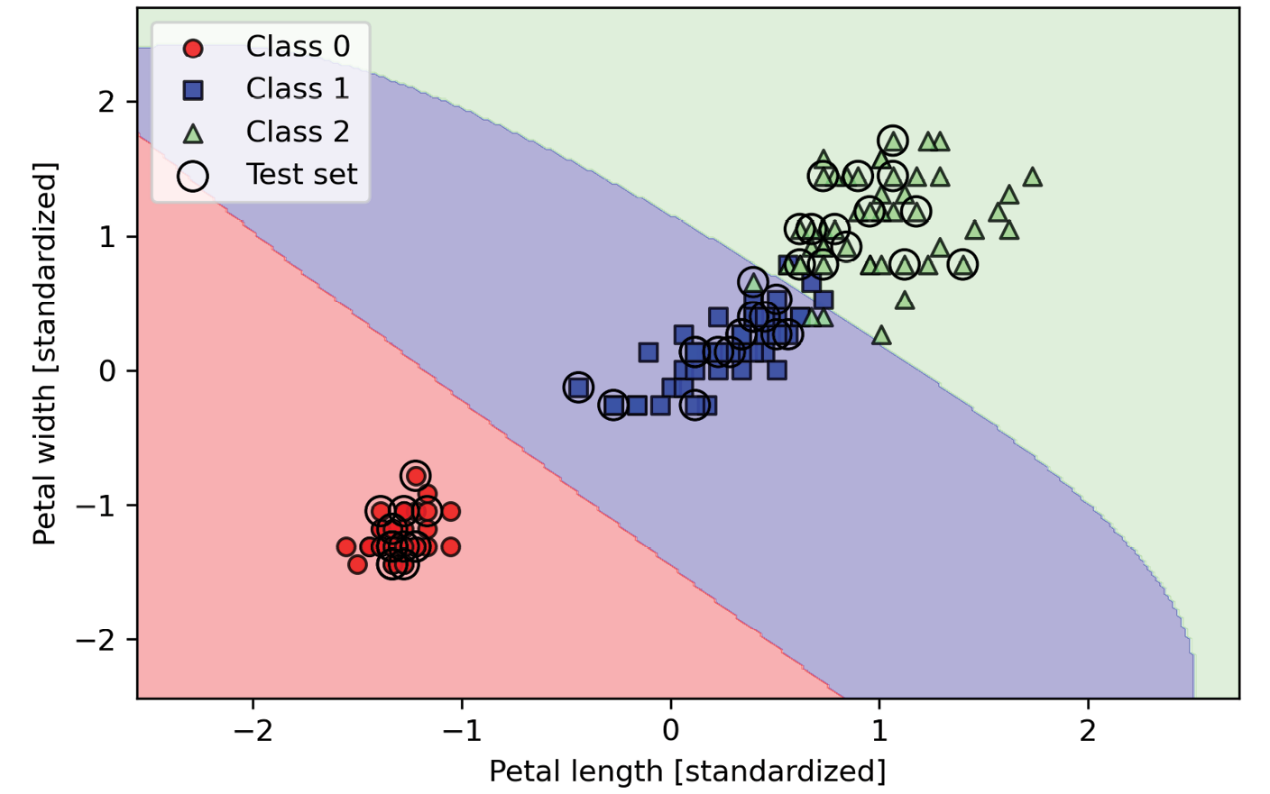


图3.16：使用小值γ的RBF核SVM模型训练鸢尾花数据集决策边界

现在我们增大γ的值并观察决策边界的效果：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | >>> 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.show() |

在图3.17中，可以看到使用相对较大的对类0和1的γ值决策边界会变得更紧密：

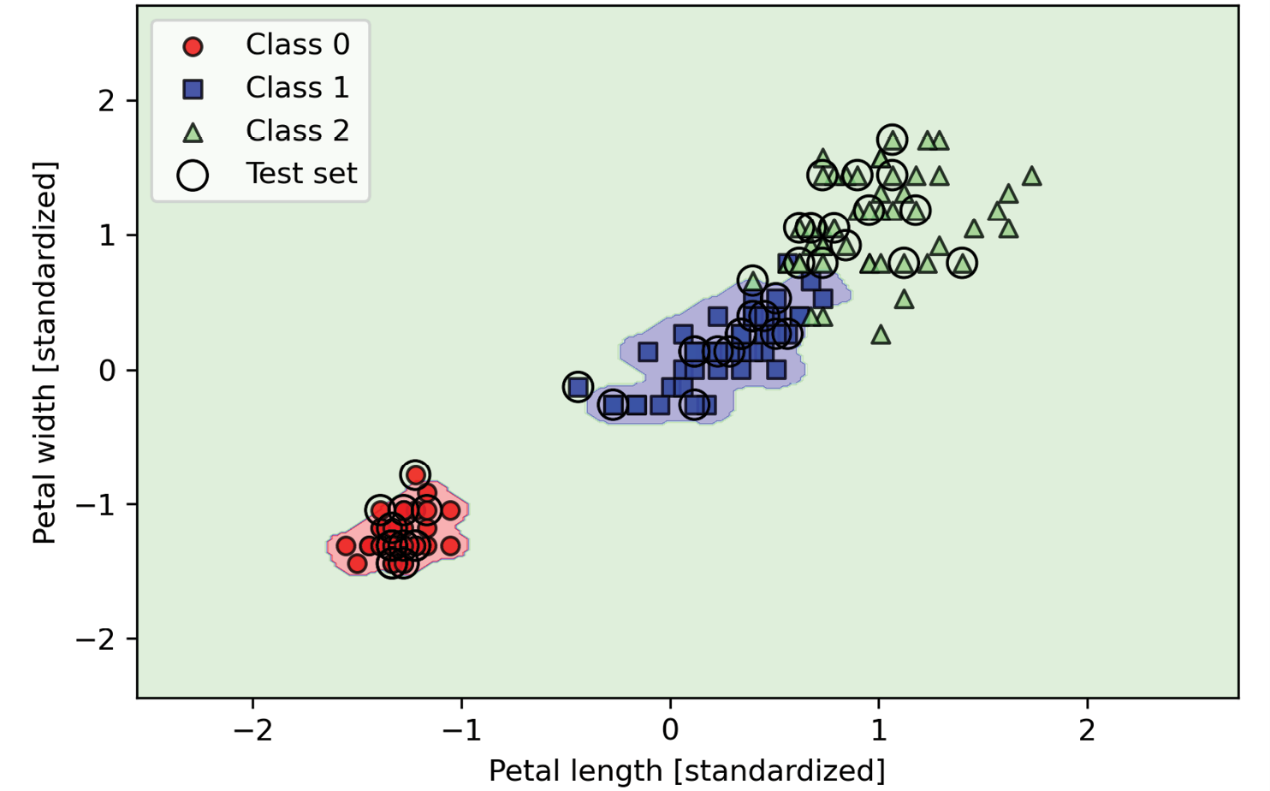


图3.17：使用大值γ的RBF核SVM模型训练鸢尾花数据集决策边界

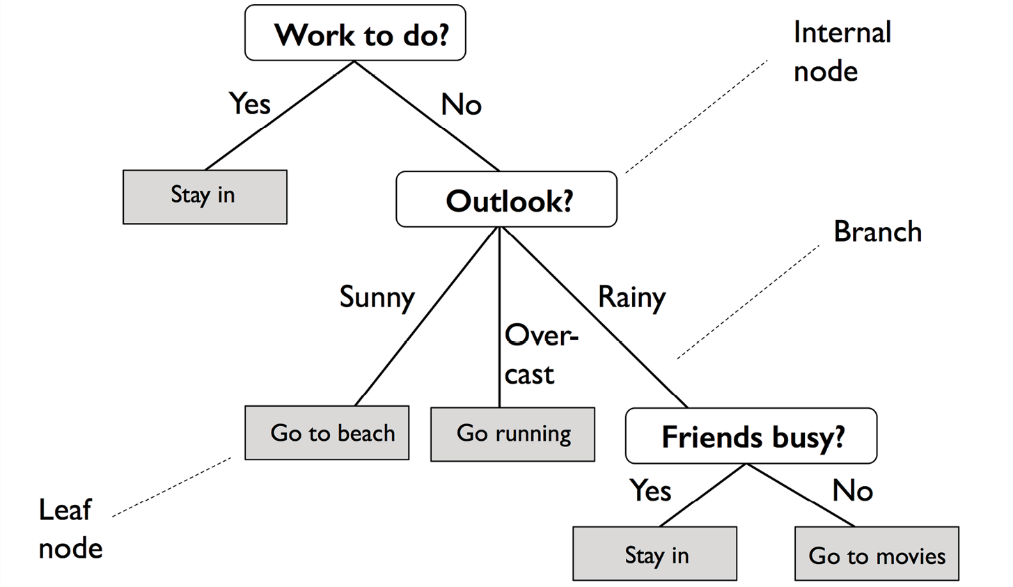
虽然模型对训练数据集的拟合非常好，这一分类很有可能会对未知数据产生很高的泛化错误。这说明参数γ对于控制算法对训练集波动过于敏感时的过拟合或是方差扮演着重要的角色。

**5 决策树学习**

如果关注**解释性**，决策树（decision tree）分类器就是非常有吸引力的模型。 通过名称“决策树”，我们就能想到这个模型是分解数据并根据询问一系列问题做出决策。

**5.1 决策树示例**

我们通过下面使用决策树来决定某一天活动的示例来进行思考：



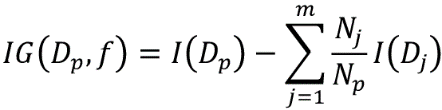
例3.18：决策树示例

根据训练数据集中的特征，决策树模型学习了一系列问题来推导样本的类标签。虽然图3.18基于分类变量描绘决策树的概念，同样的概念也适用于实数，比如鸢尾花数据集。例如，我们可以延花萼宽度特征轴定义一个临界值，并询问一个是非问题：“花萼宽度≥ 2.8 cm吗？”

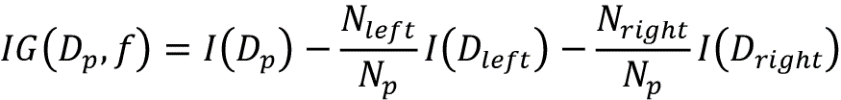
使用这一决策算法，我们从树根节点开始，并按特征的最大信息增益（information gain (IG)）分割数据，我们会在下一节中详细讲解信息增益。在迭代处理中，我们可以对每个子节点重复这一分割步骤直接到达纯叶子节点。这就意味着每个节点上的训练样本都属于同一类。在实践中，会产生有很多节点很深的树，这很容易产生过拟合。因此，我们通常需要通常会通过设置树的最大深度来修剪树。

**5.2 最大信息增益-物尽其用**

要按最多信息特征分割节点，我们需要定义一个目标函数来通过树学习算法进行优化。这里我们的目标函数是在每个分割点最大化信息增益，定义如下：



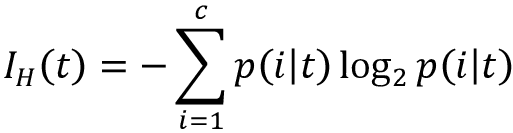
这里的*f* 是执行分割的特征，*D*p和*D*j是父级和第*j*个子点的数据集，*I*是杂度（impurity），*N*p是父节点的训练样本总数，*N*j是第*j*个子节点的样本数。可以看出，信息增益是父节点的杂度和子节点杂度总和之间的差值：子节点杂度越低，信息增益越大。但为减化及降低组合搜索空间，大部分库（包括scikit-learn）实现的是二元决策树。这表示每个父节点分割成两个子节点，*D*left和*D*right：



二元决策树中常用的三种杂度度量或分割标准是

* 基尼系数（Gini impurity (*I*G)）
* 熵（entropy (*I*H)）
* 分类误差（classification error (*I*E)）

我们先从非空类(p(i|t)≠0)熵的定义开始：



这里的*p*(*i*|*t*)是具体节点*t*属于类*i*的样本比率。如果节点上所有样本都属于同一类则熵为0，而在均匀类分布时熵为最大值。例如，在二元类场景中，如果*p*(*i*=1|*t*) = 1或*p*(*i*=0|*t*) = 0则熵为0。如果类均匀分布，即*p*(*i*=1|*t*) = 0.5且*p*(*i*=0|*t*) = 0.5，则熵为1。因此，如可以说熵准则（entropy criterion）试图最大化树中的互信息（mutual information）。

为在视觉上体现，我们通过如下代码可视化不同类分布的熵值：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | >>> def entropy(p):  ...     return - p \* np.log2(p) - (1 - p) \* np.log2((1 - p))  >>> x = np.arange(0.0, 1.0, 0.01)  >>> ent = [entropy(p) if p != 0 else None for p in x]  >>> plt.ylabel('Entropy')  >>> plt.xlabel('Class-membership probability p(i=1)')  >>> plt.plot(x, ent)  >>> plt.show() |

下面的图3.19展示了以上代码所生成的图：

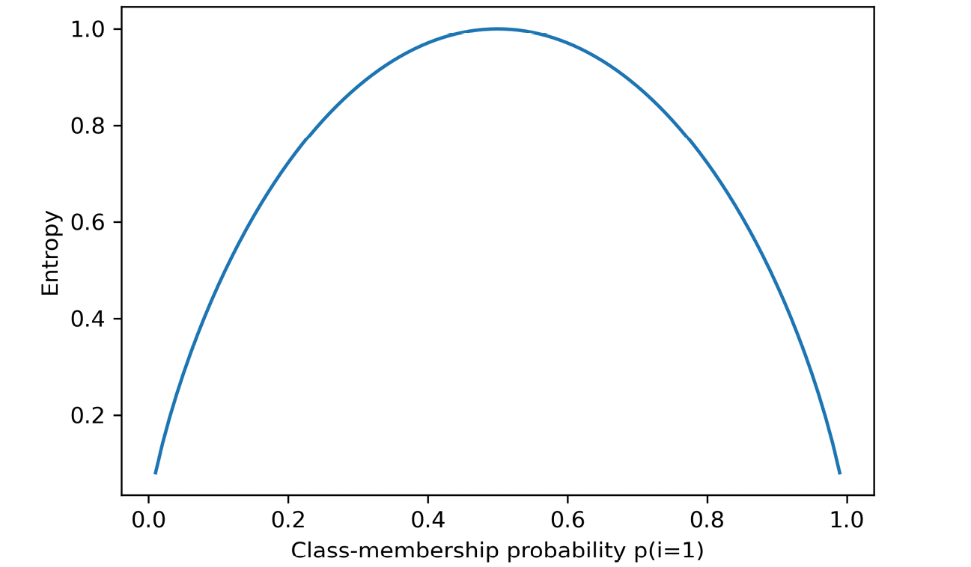
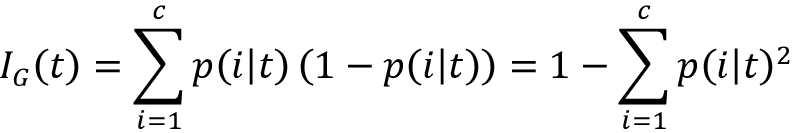


图3.19：不同类成员概率的熵值

**基尼系数**可以理解为**最小化误分类概率**的标准：



类似于熵，如果类完美混合的话基尼系数最大，例如在二元类场景(*c* = 2)中：



但是，在实践中，基尼系数和熵通常会产生相似的结果，**一般不值得花大量时间使用不同的杂度标准**，不如试验不同修剪边界来评估树。事实上，我们稍后会在图3.21中会看到，基尼系数和熵形状相似。

另一种杂度的度量是分类误差：

第3章 使用Scikit-Learn的机器学习分类器之旅

这是一种有用的修剪标准，但不推荐用于增长决策树，因为它对节点类概率中的变化敏感度更低。我们可以通过图3.20中的两种分割场景来进行描绘：

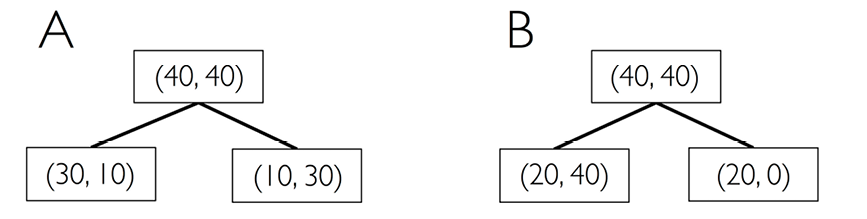
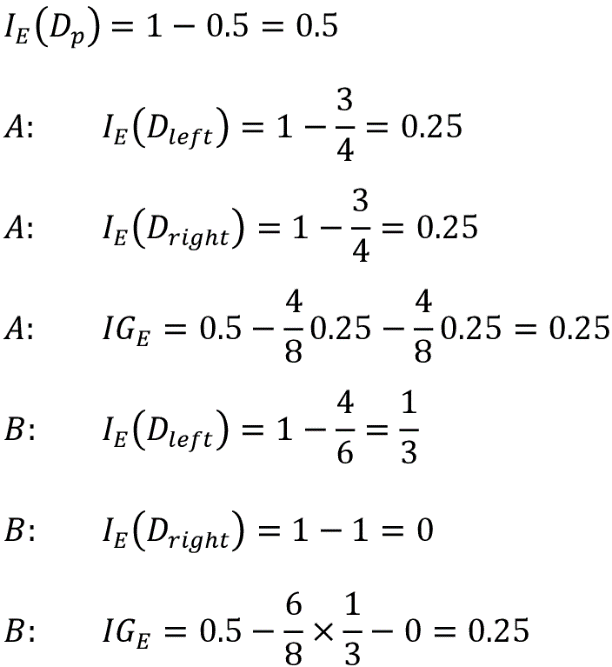
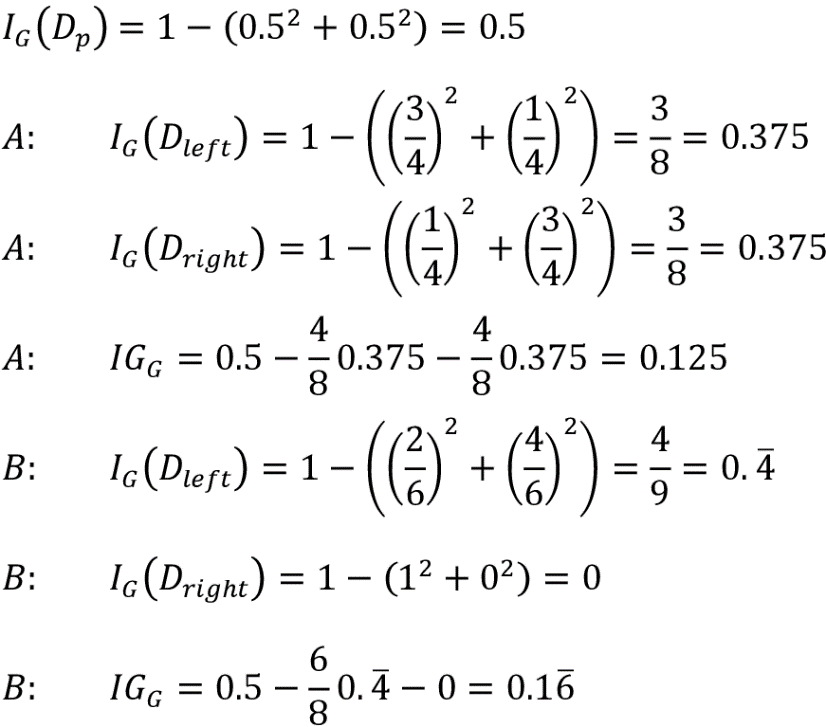


图3.20：决策树数据分割

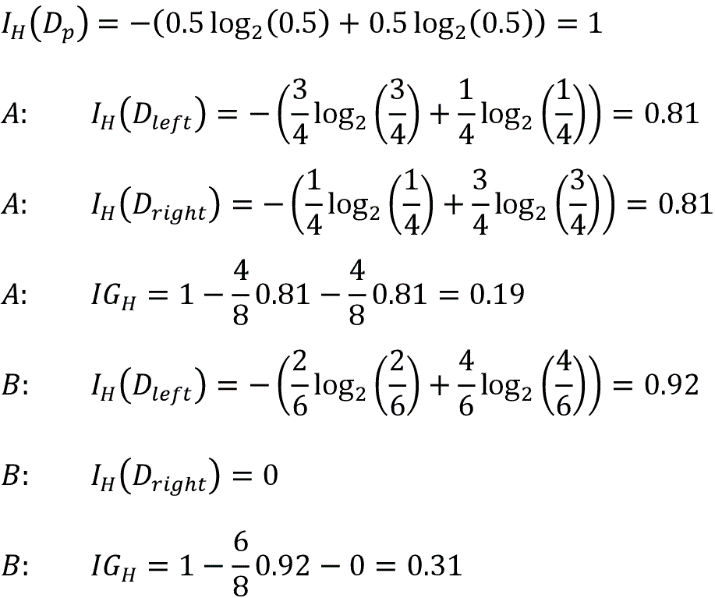
我们从你节点中的数据集*D*p开始，包含类1的40个样本和类2的40个样本，我们分割成两个数据集， *D*left和*D*righ。两种场景*A*和*B*中使用分类误差作为分割标准的信息增益相同（*IG*E = 0.25）：



但是，基尼系数对场景*B(IGG* =0.16*)*中的分类比场景*A(**IGG = 0.125)*要更优，纯度更高：



类似地，熵标准对场景B (*IG*H = 0.31)要优于场景(*IG*H = 0.19)：



对此前讨论的三种不同的杂度标准进行可视化对比，我们为类1绘制[0, 1]概率范围内的杂度指数。我们还会添加一个缩放版本的熵(熵 / 2)来观察基尼系数是熵和分类误差间的中间度量。代码如下：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31 | >>> 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]  >>> 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.show() |

上面代码生成的图如下：

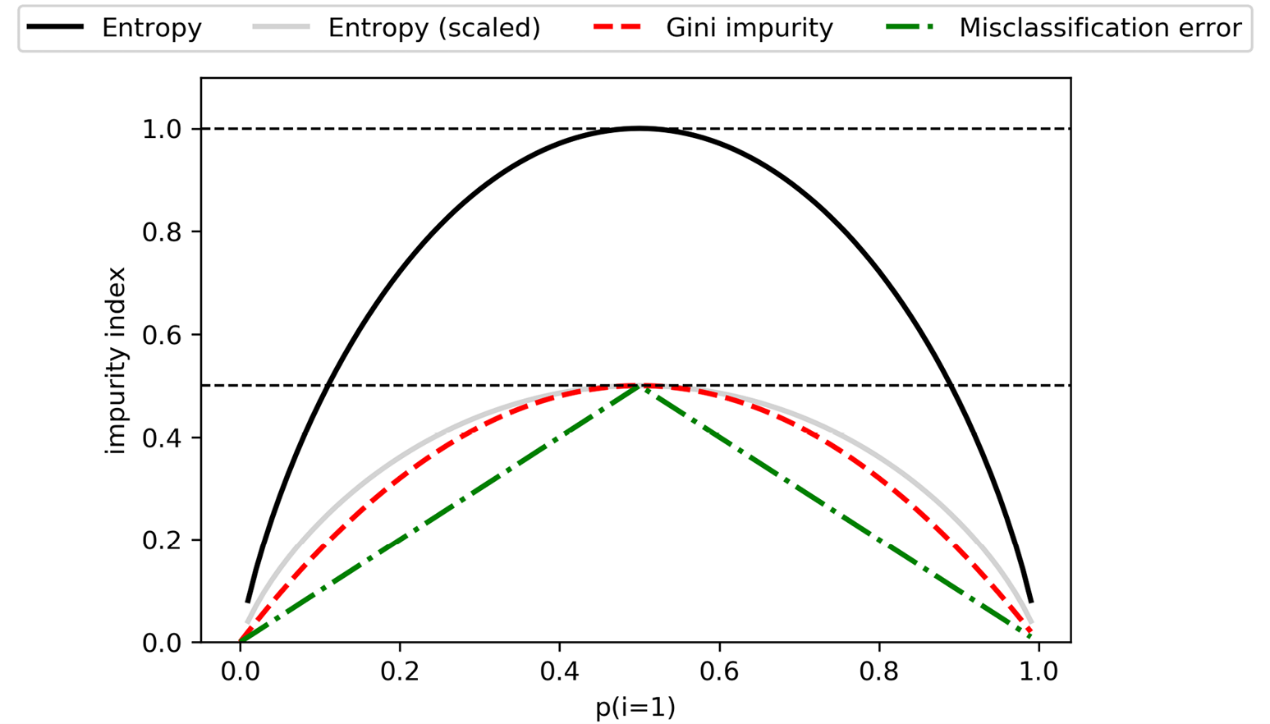


图3.21：0和1之间各类员概率的不同杂度索引

**5.3 构造决策树**

决策树可通过将特征空间分成矩形框构造复杂的决策边界。但我们需要小心，因为决策树越深，决策边界就越复杂，会很容易导致过拟合。现在我们使用scikit-learn训练一个最大尝试为4的决策树，以基尼系数作为杂度的标准。

虽然出于视图化考量会希望进行特征缩放，但注意特征缩放不是决策树算法的要求。代码如下：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | >>> from sklearn.tree import DecisionTreeClassifier  >>> tree\_model = DecisionTreeClassifier(criterion='gini',  ...                                     max\_depth=4,  ...                                     random\_state=1)  >>> tree\_model.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\_model,  ...                       test\_idx=range(105, 150))  >>> plt.xlabel('Petal length [cm]')  >>> plt.ylabel('Petal width [cm]')  >>> plt.legend(loc='upper left')  >>> plt.tight\_layout()  >>> plt.show() |

在执行示例代码后，会得到决策树的平行轴决策边界：

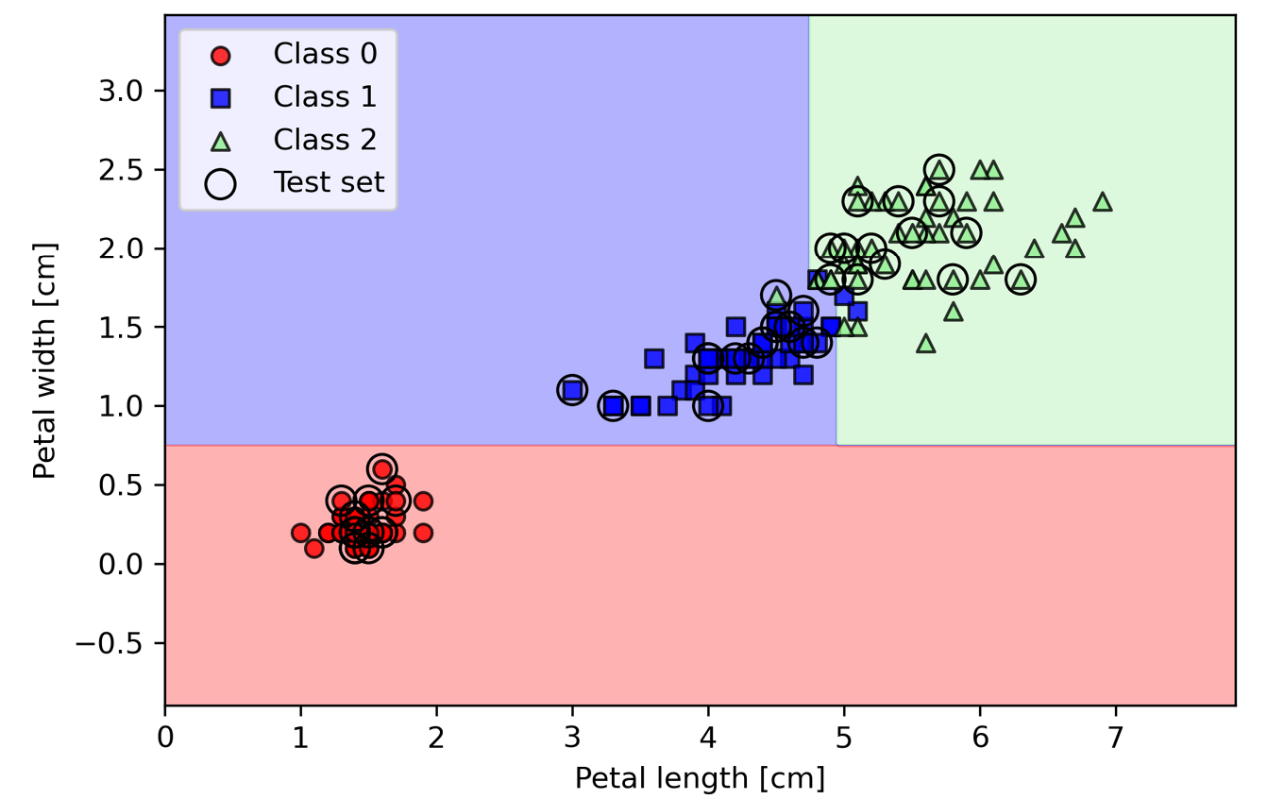


图3.22：使用决策树的鸢尾花数据决策边界

scikit-learn有一个很好的特性，那就是在通过如下代码训练后可以**可视化决策树模型**就了：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | >>> from sklearn import tree  >>> feature\_names = ['Sepal length', 'Sepal width',  ...                  'Petal length', 'Petal width']  >>> tree.plot\_tree(tree\_model,  ...                feature\_names=feature\_names,  ...                filled=True)  >>> plt.show() |

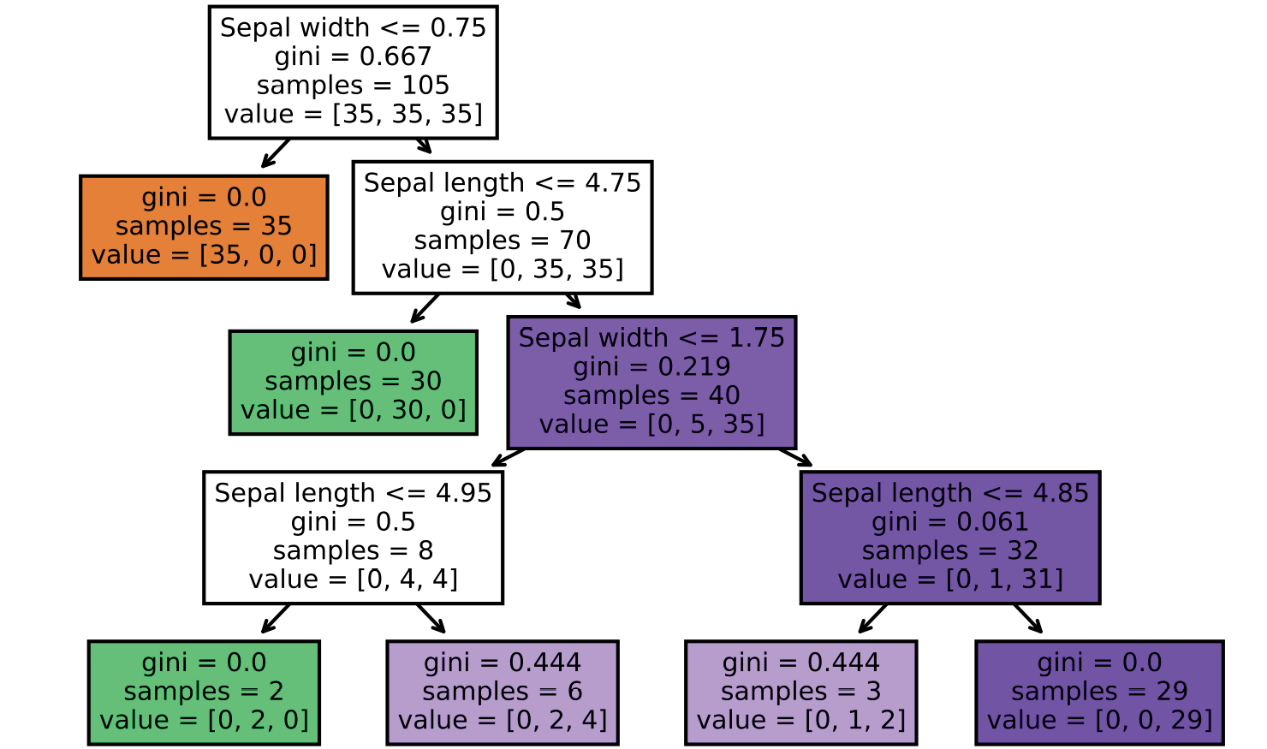


图3.23：拟合鸢尾花数据集的决策树模型

在所调用的函数plot\_tree中设置了filled=True来对节点按多数类标签进行着色。还有很多其它选项，可参见官方文档：[https://scikit-learn.org/stable/modules/generated/sklearn.tree.plot\_tree.html](https://scikit-learn.org/stable/modules/generated/sklearn.tree.plot_tree.html" \o "" \t "_blank)。

看决策树图，我们可以很好的回溯通过训练数据集决定的决策树分割方式。对于每个节点的特征分割标准，左分支对应的是True而右分支对应的是False。

顶部的根节点开始时是105个样本。第一次分割使用花萼宽度边界≤ 0.75 cm来将根分成两个子节点，分别是35个样本（左子节点）和70个样本（右子节点）。在第一次分割后，可以看到左子节点已经很纯（基尼系数= 0），仅包含Iris-setosa类的样本。然后右子节点更进一步的分割用于将样本分成Iris-versicolor和Iris-virginica类。

查看这树及其决策区域图，可以看到决策树对花进行了很好的分类。可惜**scikit-learn当前未实现手动对决策树后剪枝的功能**。但我们可以回到前面的代码示例，将决策树的max\_depth修改为3，与当前模型进行比较，这一练习留给感兴趣的读者。

此外，scikit-learn提供了一种自动化损失复杂度后剪枝程序。感兴趣的读者可以在下面的教程中找到更高阶的讲解：[https://scikit-learn.org/stable/auto\_examples/tree/plot\_cost\_complexity\_pruning.html](https://scikit-learn.org/stable/auto_examples/tree/plot_cost_complexity_pruning.html" \o "" \t "_blank)。

**5.4 通过随机森林组合多个决策树**

集成方法在过去的十年因良好的分类表现及对过拟合的健壮性在机器学习应用中获得了巨大的知名度。包括装袋法（bagging）和提升方法（boosting），但这里我们讨论基于随机森林算法的决策树，众所周知它具有良好的扩展性及易用性。随机树可以看成是决策树的组合。随机森林背后的思想是平均多个（深）决策树会在构建具有更好泛化性能及更不易过拟合的更健壮模型遭受高方差的问题。随机森林可总结为4个简单步骤：

1. 画一个尺寸为*n*的随机引导样本（随机从训练集中选取*n*个有放回的样本）。
2. 通过引导样本建立决策树。在每个节点：
   1. 随机选择无放回特征*d*。
   2. 使用按目标函数最佳分割的特征分割节点，例如，最大化信息增益。
3. 重复步骤*1*–*2* *k*次
4. 逐树累加预测按多数票分配类标签。

应当注意在训练单个决策树时对步骤2要做一个微调：将评估所有特征来决定最佳分割换成仅考虑其中的随机子集。

有放回和无放回抽样

以妨读者不熟悉采样的“有”和“无”放回，我们来做一个简单的实验。假设我们玩一个彩票游戏，随机抽取数字。一开始抽奖盒中有5个数字，0, 1, 2, 3和4，每次抽取一个数字。第一轮抽取到某个数字的机率是1/5。在无放回抽样中，每一轮抽完后不会将抽取的数字再放回抽奖盒。因此，下一轮抽取剩余的某个数字的概率取决于些前的轮次。例如，如果剩下的数字是0, 1, 2和4，下一轮抽到数字9的机率就是1/4。

但在有放回的随机采样中，我们总是会将抽取到的数字放回抽奖盒，因此抽取到某个数字的概率保持不变，同一个数字可以抽到多次。换句话说，在有放回抽样中，样本（数字）是独立的，协方差为0。例如，随机抽取5轮数字后的结果会是这样：

* 无放回随机采样：2, 1, 3, 4, 0
* 有放回随机采样：1, 3, 3, 4, 1

虽然随机森林没有提供和决策树相同级别的可解释性，随机森林的一个巨大优势是不太需要担心超参数值的选择。通常不需要对随机森林进行修剪，因为集成模型对于各决策树中预测平均值的噪声非常健壮。在实操时我们需要注意的唯一参数是从随机森林中选择的树的数量 *k*（第3步）。通常树的数量越大，随机森林分类器在计算开销增加的情况下表现也越好。

虽然在实操中很少见，随机森林分类器的其它可分别优化的超参数有：**引导样本**（第1步）的大小*n*，**每次分割随机选择的特征数*d***（第*2a*步）。通过引导样本的样本大小*n*，我们可以控制随机森林的偏差-方差均衡。

**减小引导样本的大小会增加各树间的多样性**，因为引导样本中包含具体训练样本的概率变小了。因此，收缩引导样本的大小通常会增加随机森林的随机性，有助于减小过拟合的效果。但更小的引导样本通常会导致更低的随机森林全局表现以及训练和测试表现的小间隙，但整体测试表现更低。相反，增加引导样本的大小会增加过拟合的程度。因为引导样本，进而各决策树，彼此会变得更相近，它们会学习更紧密地拟合原训练数据集。

在大部分实现中，包含scikit-learn中的RandomForestClassifier实现，选择的引导样本的大小与原训练数据集中的训练样本数相同，通常可达到良好的偏差-方差均衡。对于每次分割的特征数*d*，我们希望选择一个小于训练集中总特征数的值。scikit-learn中使用了合理的默认值，其它实现则为，其中的*m*为训练数据集中的特征数。

我们不用自己通过各决策树构建随机森林分类器，在scikit-learn已经为了实现了可供使用的实现：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | >>> from sklearn.ensemble import RandomForestClassifier  >>> forest = RandomForestClassifier(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.show() |

执行以上代码，应该会看到由随机森林中树的组合所形成的决策区域，如图3.24所示：

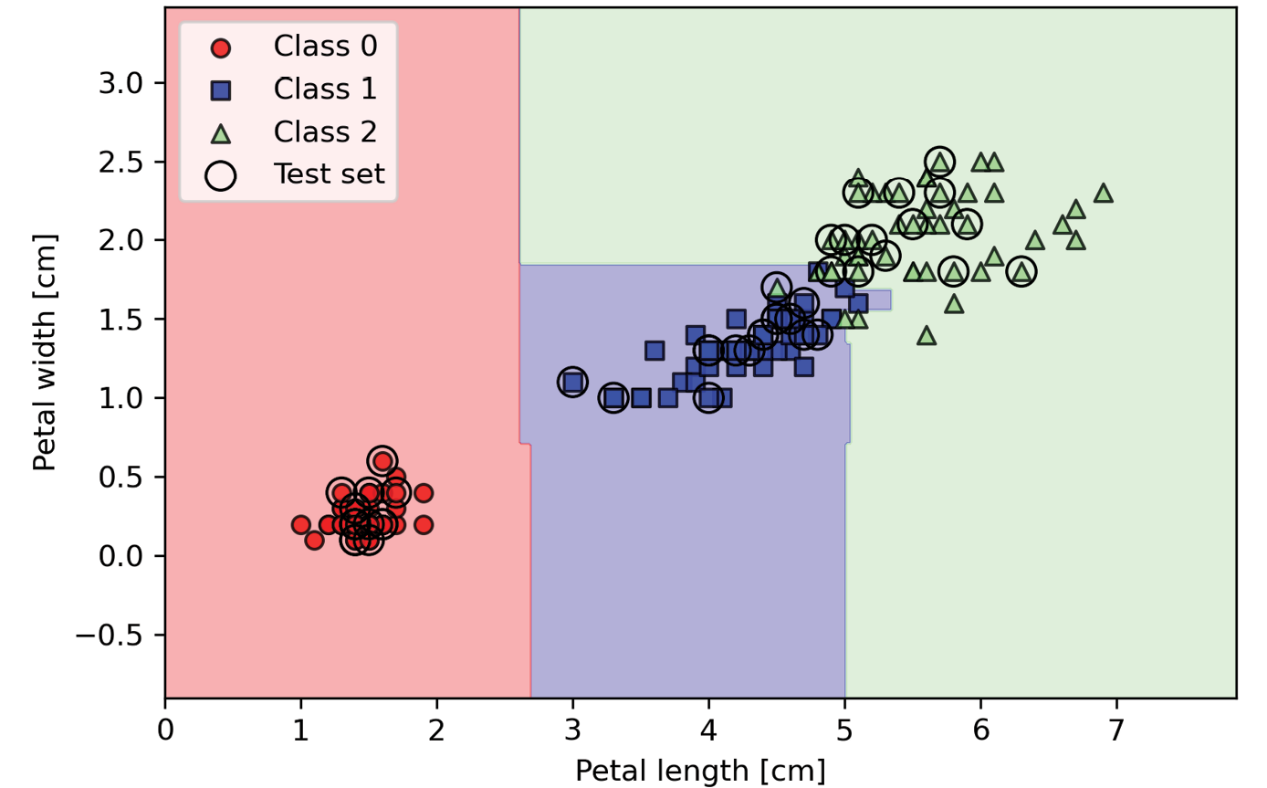


图3.24：使用随机森林训练的鸢尾花数据集的决策边界

使用上面的代码，我们通过n\_estimators参数设置的25个决策树的训练了一个随机森林。默认，它使用基尼系数度量作为分割节点的标准。我们通过很小的训练数据集生成一个很小的随机森林，出于演示目的使用了n\_jobs参数，这让我们可以使用计算机的多核（这里为双核）来并行训练模型。如果这段代码出现错误，说明你的电脑可能不支持多进程。那么可以省去n\_jobs参数或设置为n\_jobs=None。

**6 K最近邻--一种惰性学习算法**

本章我们最后要讨论的监督学习算法是K最近邻(KNN)分类器，它非常有意思，因为它与我们至此所讨论其它学习算法都有根本性的不同。

KNN是惰性学习的典型。称之为“惰性”不是因其明显的简单性，而是因为它不是从训练集学习一个判别函数，却去记住训练数据集。

**6.1 参数化和非参数化模型**

机器学习算法可分成参数化和非参数化模型。使用参数化模型，我们从训练数据集评估参数学习一个可分类新数据点的函数，无需再使用原数据集。典型的参数化模型是感知机、逻辑回归和线性SVM。相反， 非参数化模型无法通过固定的参数集特征化，并且随训练数据量的变化参数数量也发生改变。到此的两种非参数化模型的的例子是决策树分类器/随机森林和核（但非线性）SVM。

KNN属于非参数化模型的子类，称为基于实例的学习。基于实例学习的模型通过记忆训练数据集来特征化，惰性学习是基于实例的学习的到处跑特殊情况，在学习过程中无（零）损失。

**6.2 KNN算法步骤**

KNN算法本身相当直接，可总结为如下步骤：

1. 选择*k* 的数量以及距离指标
2. 查找希望分类的数据记录的*k*最近邻
3. 通过多数票分配类标签

图3.25描绘了一个新数据点(?) 如何根据5个最近邻中的多数票来分配三角类标签：

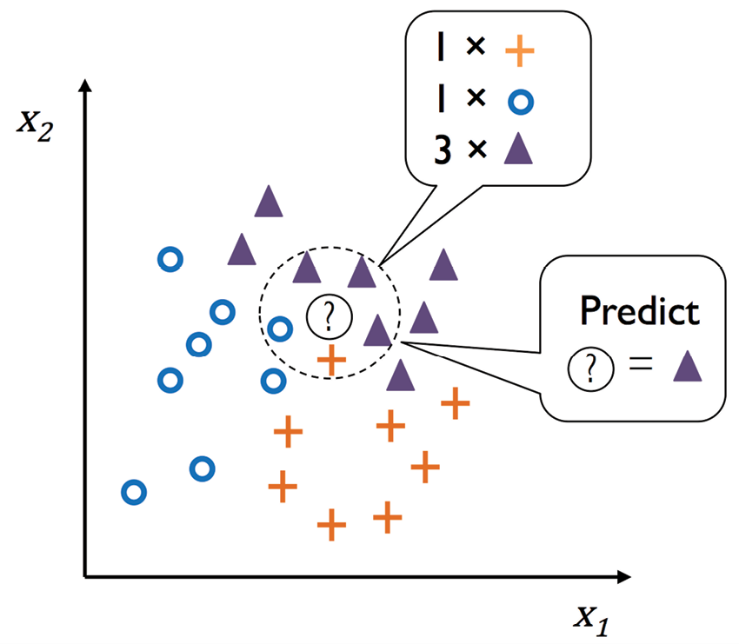


图3.25：k最近邻的实现原理

根据所选择的距离指标，KNN查找训练集中与希望分类点最近（最相似）的*k*个样本。然后通过*k*个最近邻居的多数票决定数据点的类标签

基于内存方法的优势和劣势

这种基于类型方法的主要优势是分类器可以在收集新训练数据时实时调整。但缺点是分类新样本的计算复杂度在最差场景中随训练数据集样本数的增加线性上升，除非数据维度（特征）很少并且算法使用高效数据结构实现，可高效查询训练数据。这类数据结构有k-d树(<https://en.wikipedia.org/wiki/K-d_tree>)和球树(<https://en.wikipedia.org/wiki/Ball_tree>)，在scikit-learn中对两者都提供了支持。除查询数据的计算开销外，大数据集对于有限的存储空间来说也是个问题。

但是，对于很多处理相对小到中型数据集的案例，基于内存的方法提供很好的预测和计算性能，因此对很多真实世界的问题是个好选择。使用最近邻方法近期的案例有预测制药目村的属性(*Machine Learning to Identify Flexibility Signatures of Class A GPCR Inhibition*, Biomolecules, 2020, Joe Bemister-Buffington, Alex J. Wolf, Sebastian Raschka和Leslie A. Kuhn, <https://www.mdpi.com/2218-273X/10/3/454>) 和最新的语言模型(高效最近邻语言模型 2021, Junxian He, Graham Neubig和Taylor Berg-Kirkpatrick, <https://arxiv.org/abs/2109.04212>)。

通过执行如下代码，我们使用欧式距离（Euclidean distance）指标实现scikit-learn中的KNN模型：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | >>> 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.show() |

通过对这一数据集指定KNN模型的五个邻居，我们获取了一个相对平滑的决策边界，如图3.26所示：

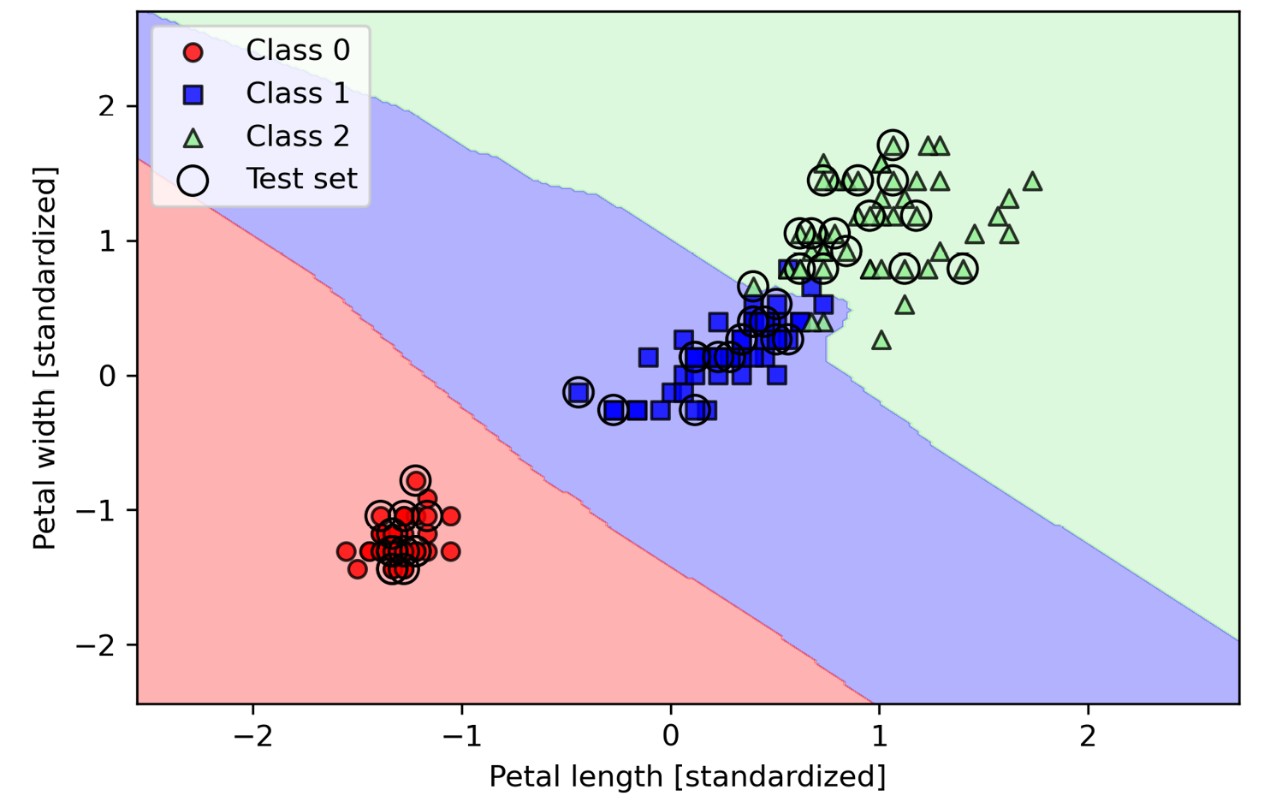
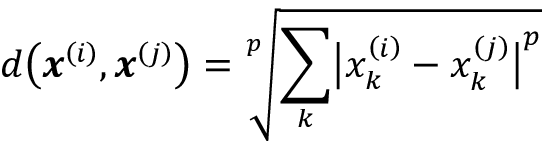


图3.26：鸢尾花数据集的k最近邻决策边界

**6.3 解决平局问题**

在出现平局时，scikit-learn中KNN算法的实现会选择距离待分类数据记录距离更近的邻居。如果这些邻居距离相似，算法会选择训练集中先出现的类标签。

对*k*的正确选择对在过拟合和欠拟合间找到平衡至关重要。我们还要确保选择适合数据集中特征的距离指标。通常，欧式距离度量用于实数值案例，比如我们鸢尾花数据集中的花，特征通过厘米度量。但如果使用欧式距离，一定要标准化数据这样每个特征对距离的贡献相同。我们在前面的代码中使用的minkowski(明可夫斯基)距离是对欧式距离和曼哈顿（Manhattan）距离的泛化，写成这样：



如果设置参数p=2就变成欧式距离，设置p=1就变成曼哈顿距离。scikit-learn中有很多其它距离度量，可通过metric参数指定。可在[https://scikit-learn.org/stable/modules/generated/sklearn.metrics.DistanceMetric.html](https://scikit-learn.org/stable/modules/generated/sklearn.metrics.DistanceMetric.html" \o "" \t "_blank)中查看。

最后，有必要提一下KNN很容易因维数灾难（curse of dimensionality）出现过拟合。维数灾难是一种特征空间随固定大小训练数据集维数增加而越来越稀疏的现象。我们可以把最近邻居看成高维空间更远距离来进行很好的评估。

我们在逻辑回归相关节中讨论到正则化的概念是避免过拟合的一种方式。但对于无法正则化的模型，比如决策树和KNN，我们可以使用特征选择和数据降维技术来帮助我们避免维数灾难。这会在接下来的两章中详细讨论。

使用的GPU其它机器学习实现

在处理大数据集时，运行k最近邻或通过多预估器拟合随机森林要求大量的计算资源和处理时间。如果电脑内置了兼容近期版本NVIDIA的CUDA库的NVIDIA GPU，推荐考虑RAPIDS生态(<https://docs.rapids.ai/api>)。比如RAPIDS的cuML(<https://docs.rapids.ai/api/cuml/stable/>)库实现了很多支持GPU加速的scikit-learn机器学习算法。可[https://docs.rapids.ai/api/cuml/stable/estimator\_intro.html](https://docs.rapids.ai/api/cuml/stable/estimator_intro.html" \o "" \t "_blank)上找到对cuML的简介。如果有兴趣学习RAPIDS生态，也可免费访问地本书作者与RAPIDS团队合作的期刊文章：*Python机器学习: 数据科学、机器学习和人工智能的主要开发和技术趋势*(<https://www.mdpi.com/2078-2489/11/4/193>)。

**小结**

本章中我们学习了很用于处理线性和非线性问题的机器学习算法。我学习了决策树，对于关心可解释性时尤具吸引力。逻辑回归不仅是基于SGD有用的在线学习模型，还可以用于预测具体事件的概率。

虽然SVM是可通过核技巧扩展至非线性问题强大的线性模型，但有很多参数需要调优才能实现良好的预测。相反， 集成方法，比如随机森林，无需很多的参数调优，不像决策树那样容易过拟合，这让其成为很多实践问题领域有吸引力的模型。KNN分类器通过惰性学习为分类提供了另一种方法，无需任何模型训练即可实现预测，但预测步骤的计算开销更大。

但比选择合适学习算法更重要的是训练集中的数据。没有有用信息和可判别特征任何算法都无法做出很好的预测。

下一章中，我们会讨论数据预处理、特征选择和数据降维相关的重要内容，这表示我们需要构建更强大的机器学习模型。稍后的[第6章 学习模型评估和超参数调优的最佳实践](https://alanhou.org/learning-best-practices-for-model-evaluation-and-hyperparameter-tuning)中，我们还会学习如何评估和比较模型的性能以及调优不同算法的有用技巧。