```
X,y = mglearn.datasets.load_extended_boston()
X_train,X_test,y_train,y_test = train_test_split(X,y,random_state=0)
lr = LinearRegression().fit(X_train,y_train)

'''
```

比较一下训练集和测试集的分数就可以发现，我们在训练集上的预测非常准确，但测试集上的R^2要低很多：

```
print("Training set score:{:.2f}".format(lr.score(X_train,y_train)))
print("Test set score:{:.2f}".format(lr.score(X_test,y_test)))

'''
运行以上程序可以得到输出:
Training set score:0.95
Test set score:0.61
```

由结果可看出训练集和测试集之间的性能差异是过拟合的明显标志。

4.4.4 Logistic 回归

上一节讨论了如何使用线性模型进行回归学习，但若要做的是分类任务该怎么办?答案蕴涵在式(4.78)的广义线性模型中：只需找一个单调可做函数将分类任务的真实标记 y 与线性回归模型的预测值联系起来。考虑二分类任务，其输出标记 $y \in \{0,1\}$，而线性回归模型产生的预测值 $z = w^T x + b$ 是实值，于是，我们需将实值 z 转换为 0/1 值。最理想的是"单位阶跃函数" (unit-step function)。

$$y = \begin{cases} 0, & z < 0; \\ 0.5, & z = 0; \\ 1, & z > 0, \end{cases} \tag{4.79}$$

即若预测值 z 大于零就判为正例小于零则判为反例预测值为临界值零则可任意判别，如图 4.39 所示。

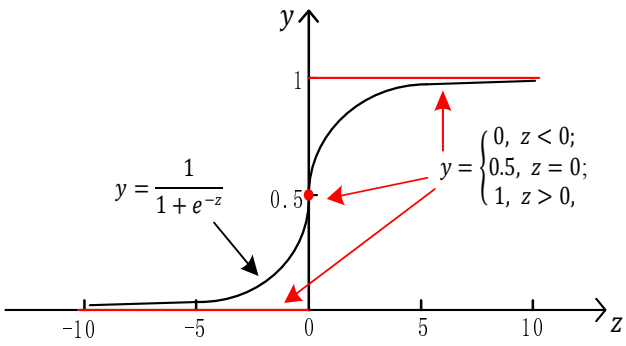


图 4.39 单位阶跃函数与对数几率函数

但从图 4.39 可看出，单位阶跃函数不连续，因此不能直接用作式(4.78)中的 g-(.)。于是我们希望找到能在一定程度上近似单位阶跃函数的"替代函数" (surrogate function)，并希望它单调可微。对数几率函数 (logistic function) 正是这样一个常用的替代函数：

$$y = \frac{1}{1+e^{-z}} \tag{4.80}$$

从图 4.39 可看出，对数几率函数是一种"Sigmoid 函数"，它将 z 值转化为一个接近 0 或 1 的 y 值并且其输出值在 $z = 0$ 附近变化很陡。将对数几率函数作为 g-(.)代入式(4.78) ，得到：

$$y = \frac{1}{1+e^{-(w^T x+b)}} \tag{4.81}$$

类似于式(4.77) ，式(4.81)可变化为：

$$\ln \frac{y}{1-y} = w^T x + b \tag{4.82}$$

若将 y 视为样本 z 作为正例的可能性，则 1-y 是其反例可能性，两者的比值：

$$\frac{y}{1-y} \tag{4.83}$$

称为"几率" (odds) ，反映了 x 作为正例的相对可能性。对几率取对数则得到"对数几率" (log odds，亦称 logit)

$$\ln \frac{y}{1-y} \tag{4.84}$$

由此可看出，式(4.81)实际上是在用线性回归模型的预测结果去逼近真实标记的对数几率，因此，其对应的模型称为"对数几率回归" (logistic regression，亦称 logit regression)。特别需注意到，虽然它的名字是"回归"，但实际却是一种分类学习方法。这种方法有很多优点，例如它是直接对分类可能性进行建模，无需事先假设数据分布，这样就避免了假设分布不准确所带来的问题；它不是仅预测出"类别"，而是可得到近似概率预测，这对许多需利用概率辅助决策的任务很有用；此外，对率函数是任意阶可导的凸函数，有很好的数学性质，现有的许多数值优化算法都可直接用于求取最优解。

线性模型也广泛应用于分类问题。不同的算法使用不同的方法来度量“对训练集拟合好坏”。由于数学上的技术原因，不可能调节 w 和 b 使得算法产生的误分类数量最少。对于我们的目的，以及对于许多应用而言，上面第一点(称为损失函数)的选择并不重要。

最常见的两种线性分类算法是 Logistic 回归(logistic regression)和线性支持向量机(linear support vector machine，线性 SVM)，前者在 linear_model.LogisticRegression 中实现，后者 svm.LinearSVC(SVC 代表支持向量分类器)中实现。虽然 LogisticRegression 的名字中含有回归(regression)，但它是一种分类算法，并不是回归算法，不应与 LinearRegression 混淆。

我们可以将 LogisticRegression 和 LinearSVC 模型应用到 forge 数据集上，并将线性模型找到的决策边界可视化(图 4.40)：

```
from sklearn.linear_model import LogisticRegression
from sklearn.svm import LinearSVC
X, y = mglearn.datasets.make_forge()
fig, axes = plt.subplots(1, 2, figsize=(10, 3))
for model, ax in zip([LinearSVC(), LogisticRegression()], axes):
    clf = model.fit(X, y)
    mglearn.plots.plot_2d_separator(clf, X, fill=False, eps=0.5,
    ax=ax, alpha=.7)
    mglearn.discrete_scatter(X[:, 0], X[:, 1], y, ax=ax)
    ax.set_title("{}".format(clf.__class__.__name__))
    ax.set_xlabel("Feature 0")
    ax.set_ylabel("Feature 1")
axes[0].legend()

'''
```

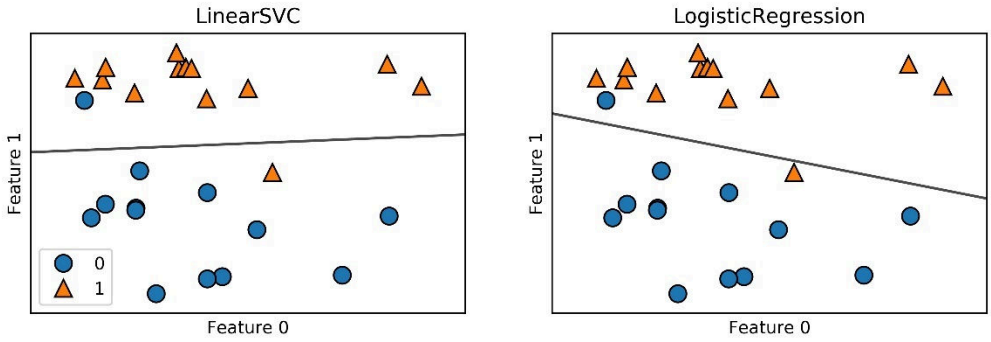


图 4.40 线性 SVM 和 Logistic 回归在 forge 数据集上的决策边界(均为默认参数)

在这张图中，forge 数据集的第一个特征位于 x 轴，第二个特征位于 y 轴，与前面相同。图中分别展示了 LinearSVC 和 LogisticRegression 得到的决策边界，都是直线，将顶部归为类别 1 的区域和底部归为类别 0 的区域分开了。换句话说，对于每个分类器而言，位于黑线上方的新数据点都会被划为类别 1，而在黑线下方的点都会被划为类别 0。两个模型得到了相似的决策边界。注意，两个模型中都有两个点的分类是错误的。

两个模型都默认使用 L2 正则化，就像 Ridge 对回归所做的那样。对于 LogisticRegression 和 LinearSVC，决定正则化强度的权衡参数叫作 C。C 值越大，对应的正则化越弱。换句话说，如果参数 C 值较大，那么 LogisticRegression 和 LinearSVC 将尽可能将训练集拟合到最好，而如果 C 值较小，那么模型更强调使系数向量(w)接近于 0。

接下来我们在乳腺癌数据集上详细分析 LogisticRegression：

```
from sklearn.datasets import load_breast_cancer
cancer = load_breast_cancer()
X_train, X_test, y_train, y_test = train_test_split(
        cancer.data, cancer.target, stratify=cancer.target, random_state=42)
logreg = LogisticRegression().fit(X_train, y_train)
print("Training set score: {:.3f}".format(logreg.score(X_train, y_train)))
print("Test set score: {:.3f}".format(logreg.score(X_test, y_test)))

'''
运行以上程序可以得到输出:
Training set score: 0.953
Test set score: 0.958
```

C=1 的默认值给出了相当好的性能，在训练集和测试集上都达到 95% 的精度。但由于训练集和测试集的性能非常接近，所以模型很可能是欠拟合的。我们尝试增大 C 来拟合一个更灵活的模型：

```
logreg100 = LogisticRegression(C=100).fit(X_train, y_train)
print("Training set score: {:.3f}".format(logreg100.score(X_train, y_train)))
print("Test set score: {:.3f}".format(logreg100.score(X_test, y_test)))

'''
运行以上程序可以得到输出:
Training set score: 0.972
Test set score: 0.965
```

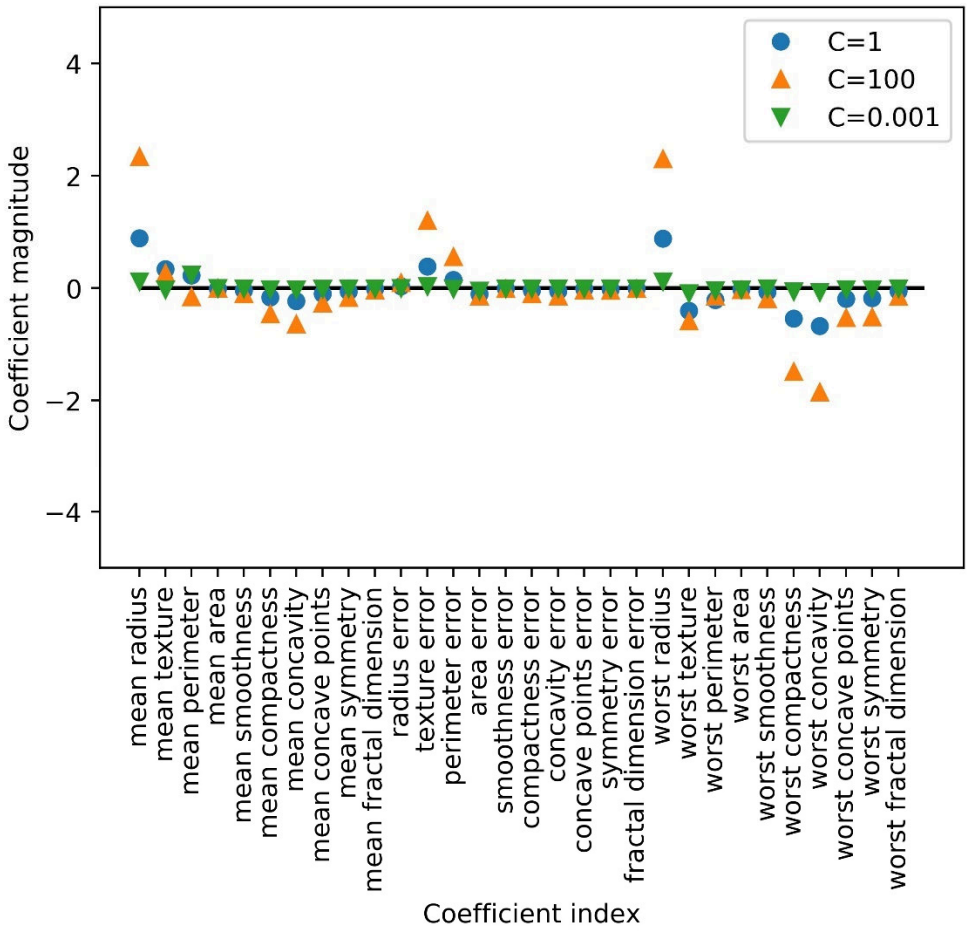
使用 C=100 可以得到更高的训练集精度，也得到了稍高的测试集精度，这也证实了我们的直觉，即更复杂的模型应该性能更好。我们还可以研究使用正则化更强的模型时会发生什么。设置 C=0.01：

```
logreg001 = LogisticRegression(C=0.01).fit(X_train, y_train)
print("Training set score: {:.3f}".format(logreg001.score(X_train, y_train)))
print("Test set sc.ore: {:.3f}".format(logreg001.score(X_test, y_test)))
'''
运行以上程序可以得到输出:
Training set score: 0.934
Test set score: 0.930
```

正如我们所料，在图 4.4 中将已经欠拟合的模型继续向左移动，训练集和测试集的精度都比采用默认参数时更小。最后，来看一下正则化参数 C 取三个不同的值时模型学到的系数(图 4.41)：

```
plt.plot(logreg.coef_.T, 'o', label="C=1")
plt.plot(logreg100.coef_.T, '^', label="C=100")
plt.plot(logreg001.coef_.T, 'v', label="C=0.001")
plt.xticks(range(cancer.data.shape[1]), cancer.feature_names, rotation=90)
plt.hlines(0, 0, cancer.data.shape[1])
plt.ylim(-5, 5)
plt.xlabel("Coefficient index")
plt.ylabel("Coefficient magnitude")
plt.legend()

,,,
```



\

图 4.41 不同 C 值的 Logistic 回归在乳腺癌数据集上学到的系数

由于 LogisticRegression 默认应用 L2 正则化，更强的正则化使得系数更趋向于 0，但系数永远不会正好等于 0。进一步观察图像，还可以在第 3 个系数那里发现有趣之处，这个系数是“平均周长”（mean perimeter）。C=100 和 C=1 时，这个系数为负，而 C=0.001 时这个系数为正，其绝对值比 C=1 时还要大。在解释这样的模型时，人们可能会认为，系数可以告诉我们某个特征与哪个类别有关。例如，人们可能会认为高“纹理错误”（texture error）特征与“恶性”样本有关。但“平均周长”系数的正负号发生变化，说明较大的“平均周长”可以被当作“良性”的指标或“恶性”的指标，具体取决于我们考虑的是哪个模型。

这也说明，对线性模型系数的解释应该始终持保留态度。

如果想要一个可解释性更强的模型，使用 L1 正则化可能更好，因为它约束模型只使用少数几个特征。下面是使用 L1 正则化的系数图像和分类精度(图 4.42)。

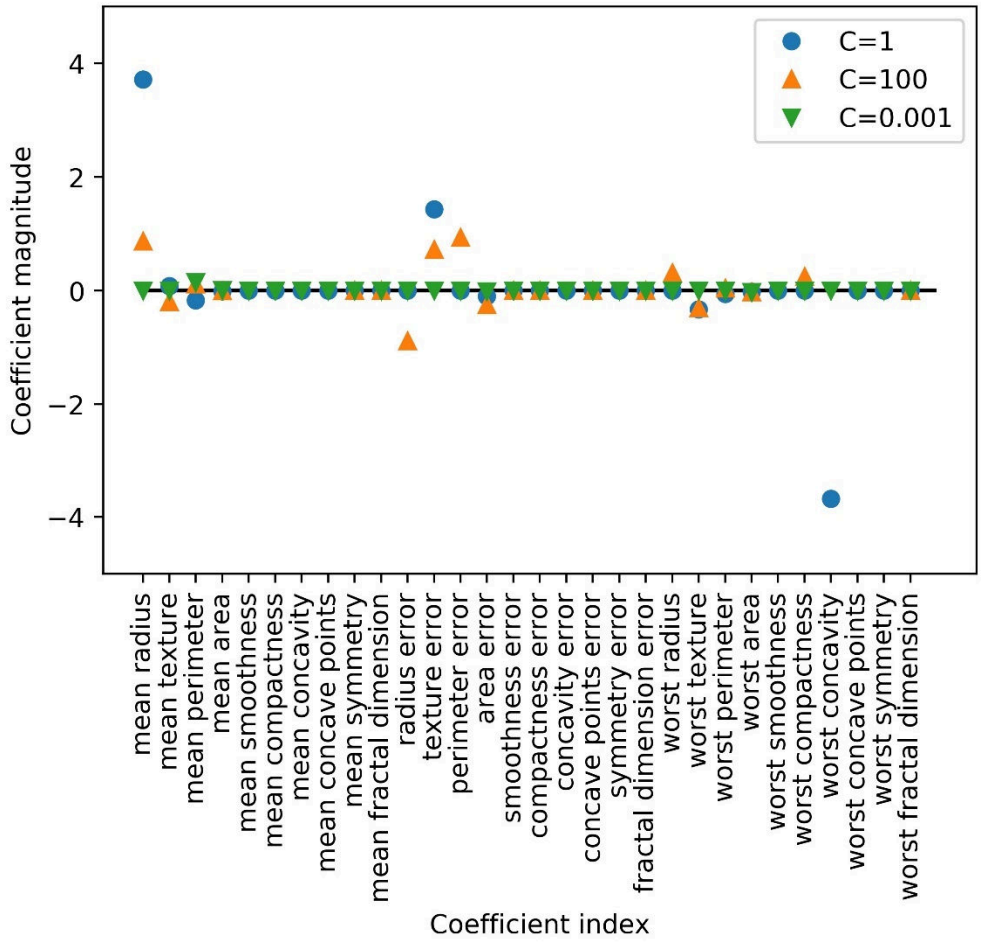


图 4.42 对于不同的 C 值，L1 惩罚的 Logistic 回归在乳腺癌数据集上学到的系数

```
for C, marker in zip([0.001, 1, 100], ['o', '^', 'v']):
lr_l1 = LogisticRegression(C=C, penalty="l1").fit(X_train, y_train)
print("Training accuracy of l1 logreg with C={:.3f}: {:.2f}".format(
C, lr_l1.score(X_train, y_train)))
print("Test accuracy of l1 logreg with C={:.3f}: {:.2f}".format(
C, lr_l1.score(X_test, y_test)))
plt.plot(lr_l1.coef_.T, marker, label="C={:.3f}".format(C))
plt.xticks(range(cancer.data.shape[1]), cancer.feature_names, rotation=90)
plt.hlines(0, 0, cancer.data.shape[1])
plt.xlabel("Coefficient index")
plt.ylabel("Coefficient magnitude")
plt.ylim(-5, 5)
plt.legend(loc=3)

'''
```

运行以上程序可以得到输出:

```
Training accuracy of l1 logreg with C=0.001: 0.91
Test accuracy of l1 logreg with C=0.001: 0.92
Training accuracy of l1 logreg with C=1.000: 0.96
Test accuracy of l1 logreg with C=1.000: 0.96
Training accuracy of l1 logreg with C=100.000: 0.99
Test accuracy of l1 logreg with C=100.000: 0.98
```

如你所见，用于二分类的线性模型与用于回归的线性模型有许多相似之处。与用于回归的线性模型一样，模型的主要差别在于 penalty 参数，这个参数会影响正则化，也会影响模型是使用所有可用特征还是只选择特征的一个子集。

4.4.5 支持向量机

1. 支持向量机间隔

给定训练样本集$D = \{(x_1, y_1), (x_2, y_2), \cdots, (x_m, y_m)\}, y_i \in \{-1, +1\}$，分类学习最基本的想法就是基于训练集 D 在样本空间中找到一个划分超平面，将不同类别的样本分开，但能将训练样本分开的划分超平面可能有很多，如图 4.43，我们应该努力去找到哪一个呢？

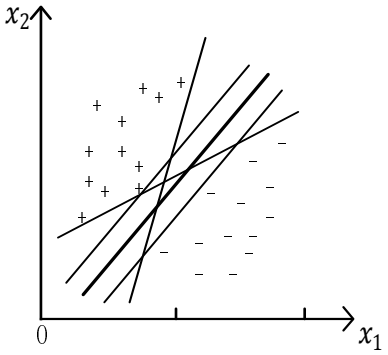


图 4.43 在多个划分超平面将两类训练样本分开

直观上看，应该去找位于两类训练样本“正中间”的划分超平面，因为该划分超平面对训练样本局部扰动的“容忍”性最好。例如，由于训练集的局限性或噪声的因素，训练集外的样本可能比图 4.43 训练样本更接近两个类的分隔界，这将使许多划分超平面出现错误，而红色的超平面受影响最小。换言之，这个划分超平面所产生的分类结果是最鲁棒的，对未见示例的泛化能力最强。

在样本空间中，划分超平面可通过如下线性方程来描述：

$$w^T x + b = 0 \tag{4.85}$$

其中$w_i = (w_1; w_2; \cdots; w_d)$为法向量，决定了超平面的方向；b 为位移项，决定了超平面与原点之间的距离。显然，划分超平面可被法向量 w 和位移 b 确定，下面我们将其记为(w，b)。样本空间中任意点 x 到超

平面(w，b)的距离可写为：

$$\mathrm{r}=\frac{|w^{T}x+b|}{||w||} \tag{4.86}$$

假设超平面(w，b)能将训练样本正确分类，即对于$(x_i,y_i)\in \mathrm{D}$，若$y_i=+1$，则有$w^{T}x_i+b>0$；若$y_i=-1$，则有$w^{T}x_i+b<0$。令：

$$\begin{cases} w^{T}x_i+b\geq +1，\ y_i=+1; \\ w^{T}x_i+b\leq -1，\ y_i=-1. \end{cases} \tag{4.87}$$

如图 4.44 所示，距离超平面最近的这几个训练样本点使式（4.45）的等号成立，它们被称为“支持向量”(support vector)，两个异类支持向量到超平面的距离之和为：

$$\gamma=\frac{2}{||w||} \tag{4.88}$$

它被称为“间隔”(Margin)。

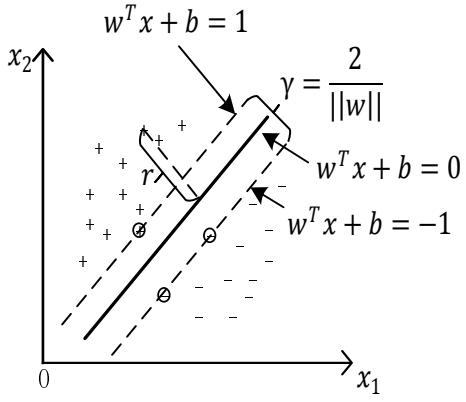


图 4.44 支持向量与间隔

欲找到具有“最大间隔”（maximum margin）的划分超平面，也就是要找到能满足式（4.87）约束的参数 w 和 b，使得 γ 最大，即：

$$\begin{aligned} &\max_{w,b}\frac{2}{||w||} \\ &s.t.\quad y_i(w^{T}x_i+b)\geq 1, i=1,2,\cdots,m. \end{aligned} \tag{4.89}$$

显然，为了最大化间隔，仅需最大化$||w||^{-1}$，这等价于最小化$||w||^{2}$。于是，式（4.89）写为：

$$\begin{aligned} &\min_{w,b}\frac{1}{2}||w||^{2} \\ &.t.\quad y_i(w^{T}x_i+b)\geq 1, i=1,2,\cdots,m. \end{aligned} \tag{4.90}$$

这就是支持向量机（Support Vector Machine，简称 SVM）的基本型。

2. 核函数

在本章前面的讨论中，我们假设训练样本是线性可分的，即存在一个划分超平面能将训练样本正确分类。然而在现实任务中，原始样本空间内也许并不存在一个能正确划分两类样本的超平面。例如图 6.18 中的“异或”问题就不是线性可分的。

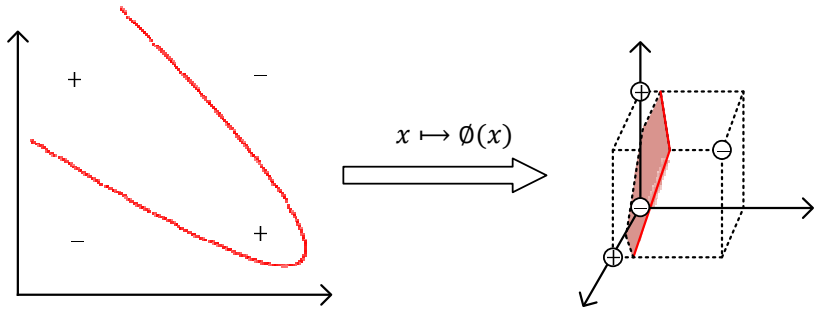


图 4.45 异或问题与非线性映射

对这样的问题，可将样本从原始空间映射到一个更高维的特征空间，使得样本在这个特征空间内线性可分。例如在图 4.45 中，若将原始的二维空间映射到一个合适的三维空间，就能找到一个合适的划分超平面。幸运的是，如果原始空间是有限维，即属性数有限，那么一定存在一个高维特征空间使样本可分。

令$\emptyset(x)$表示将 x 映射后的特征向量，于是，在特征空间中划分超平面所对应的模型可表示为：

$$f(x) = w^T\emptyset(x) + b \tag{4.91}$$

其中 w 和 b 是模型参数。类似式（4.90），有：

$$\min_{w,b} \frac{1}{2}||w||^2 \tag{4.92}$$
$$s.t. \quad y_i(w^T\emptyset(x_i) + b) \geq 1, i = 1,2,\cdots,m.$$

其对偶问题是：

$$\max_{\alpha} \sum_{i=1}^{m} \alpha_i - \frac{1}{2}\sum_{i=1}^{m}\sum_{j=1}^{m} \alpha_i\alpha_j y_i y_j \emptyset(x_i)^T\emptyset(x_j) \tag{4.93}$$
$$\text{s.t.} \sum_{i=1}^{m} \alpha_i y_i = 0,$$
$$\alpha_i \geq 0, i = 1,2,\cdots,m.$$

求解式（4.93）涉及到计算$\emptyset(x_i)^T\emptyset(x_j)$，这是样本$x_i$与$x_j$映射到特征空间之后的内积。由于特征空间维数可能很高，甚至可能是无穷维，因此直接计算$\emptyset(x_i)^T\emptyset(x_j)$通常是困难的。为了避开这个障碍，可以设想这样一个函数：

$$\kappa(x_i, x_j) = \langle \emptyset(x_i), \emptyset(x_j) \rangle = \emptyset(x_i)^T \emptyset(x_j) \tag{4.94}$$

即x_i与x_j在特征空间的内积等于它们在原始样本空间中通过函数$\kappa(\cdot,\cdot)$计算的结果。有了这样的函数，我们就不必直接去计算高维甚至无穷维特征空间中的内积，于是式（4.93）可重写为：

$$\begin{aligned} &\max_{\alpha} \sum_{i=1}^{m} \alpha_i - \frac{1}{2} \sum_{i=1}^{m} \sum_{j=1}^{m} \alpha_i \alpha_j y_i y_j \kappa(x_i, x_j) \\ &\text{s.t.} \sum_{i=1}^{m} \alpha_i y_i = 0, \\ &\alpha_i \geq 0, i = 1,2,\cdots,m. \end{aligned} \tag{4.95}$$

求解后即可得到：

$$\begin{aligned} f(x) &= w^T \emptyset(x) + b \\ &= \sum_{i=1}^{m} \alpha_i y_i \emptyset(x_i)^T \emptyset(x) + b \\ &= \sum_{i=1}^{m} \alpha_i y_i \kappa(x, x_i) + b \end{aligned} \tag{4.96}$$

这里的函数$\kappa(\cdot,\cdot)$就是“核函数”(kernel function)。式（4.96）显示出模型最优解可通过训练样本的核函数展开，这一展式亦称“支持向量展式”(support vector expansion)。

显然，若已知合适映射$\emptyset(\cdot)$的具体形式，则可写出核函数$\kappa(\cdot,\cdot)$。但在现实任务中我们通常不知道$\emptyset(\cdot)$是什么形式，那么，合适的核函数是否一定存在呢？什么样的函数能做核函数呢？我们有下面的定理：

定理 4.1(核函数). 令χ为输入空间，$\kappa(\cdot,\cdot)$是定义在$\chi \times \chi$上的对称函数，则κ是核函数当且仅当对于任意数据$D = \{x_1, x_2, \cdots, x_m\}$，“核矩阵”(kernel matrix)K 总是半正定的：

$$X = \begin{bmatrix} \kappa(x_1, x_1) & \cdots & \kappa(x_1, x_j) & \cdots & \kappa(x_1, x_m) \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ \kappa(x_i, x_1) & \cdots & \kappa(x_i, x_j) & \cdots & \kappa(x_i, x_m) \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ \kappa(x_m, x_1) & \cdots & \kappa(x_m, x_j) & \cdots & \kappa(x_m, x_m) \end{bmatrix}$$

定理 4.1 表明，只要一个对称函数所对应的核矩阵半正定，它就能作为核函数使用。事实上，对于一个半正定核矩阵，总能找到一个与之对应的映射$\emptyset$。换言之，任何一个核函数都隐式地定义了一个称为“再生核希尔伯特空间”(Reproducing Kernel Hilbert Space，简称 RKHS)的特征空间。

通过前面的讨论可知，我们希望样本在特征空间内线性可分，因此特征空间的好坏对支持向量机的性能至关重要。需注意的是，在不知道特征映射的形式时，我们并不知道什么样的核函数是合适的，而核函数也仅是隐式地定义了这个特征空间。于是，“核函数选择”成为支持向量机的最大变数。若核函数选择不合适，则意味着将样本映射到了一个不合适的特征空间，很可能导致性能不佳。

表 4.5 列出了几种常用的核函数。

表 4.5 常用核函数

名称	表达式	参数
线性核	$\kappa(x_i, x_j) = x_i^T x_j$	
多项式核	$\kappa(x_i, x_j) = (x_i^T x_j)^d$	$d \geq 1$为多项式的次数

高斯核	$\kappa(x_i, x_j) = exp\left(-\frac{\|\|x_i - x_j\|\|^2}{2\sigma^2}\right)$	$\sigma > 0$为高斯核的带宽（width）
拉普拉斯核	$\kappa(x_i, x_j) = exp\left(-\frac{\|\|x_i - x_j\|\|}{2\sigma}\right)$	$\sigma > 0$
Sigmoid 核	$\kappa(x_i, x_j) = \tan h\left(\beta x_i{}^T x_j + \theta\right)$	tan h为双曲正切函数，$\beta > 0, \theta < 0$

此外，还可通过函数组合得到，例如：

- 若κ_1和κ_2为核函数，则对于任意正数γ_1、γ_2，其线性组合

$$\gamma_1\kappa_1 + \gamma_2\kappa_2 \tag{4.97}$$

也是核函数；

- 若κ_1和κ_2为核函数，则核函数的直积：

$$\kappa_1 \otimes \kappa_2(x, z) = \kappa_1(x, z)\kappa_2(x, z) \tag{4.98}$$

也是核函数；

- 若κ_1为核函数，则对于任意函数$g(x)$，

$$\kappa(x, z) = g(x)\kappa_1(x, z)g(z) \tag{4.99}$$

也是核函数。

3. 核方法

给定训练样本$\{(x_1, y_1), (x_2, y_2), \cdots, (x_m, y_m)\}$，若不考虑偏移项 b，则无论 SVM 还是 SVR，学得的模型总能表示成核函数$\kappa(x, x_i)$的线性组合，不仅如此，事实上我们有下面这个称为“表示定理”(representer theorem)的更一般的结论：

定理 4.2(表示定理). 令$\mathcal{H}$为核函数κ对应的再生核希尔伯特空间，$||h||_{\mathcal{H}}$表示$\mathcal{H}$空间中关于 h 的范数，对于任意单调递增函数Ω：$[0, \infty] \mapsto \mathbb{R}$和任意非负损失函数$\imath$：$\mathbb{R}^m \mapsto [0, \infty]$，优化问题：

$$\min_{h \in \mathcal{H}} F(h) = \Omega(||h||_{\mathcal{H}}) + \imath\big(h(x_1), h(x_2), \cdots, h(x_m)\big) \tag{4.100}$$

的解总可写为：

$$h^*(x) = \sum_{i=1}^{m} \alpha_i \kappa(x, x_i) \tag{4.101}$$

表示定理对损失函数没有限制，对正则化项Ω仅要求单调递增，甚至不要求Ω是凸函数，意味着对于一般的损失函数和正则化项，优化问题(4.100)的最优解$h^*(x)$都可表示为核函数$\kappa(x, x_i)$的线性组合；这显示出核函数的巨大威力。

人们发展出一系列基于核函数的学习方法，统称为“核方法”(kernel methods)。最常见的，是通过“核化”（即引入核函数）来将线性学习器拓展为非线性学习器。下面我们以线性判别分析为例来演示如何通过核化来对其进行非线性拓展，从而得到“核线性判别分析”(Kernelized Linear Discriminant Analysis，简称 KLDA)。

我们先假设可通过某种映射ϕ：χ ⟼ F将样本映射到一个特征空间 F，然后在 F 中执行线性判别分析，以求得：

$$h(x) = w^T \phi(x) \tag{4.102}$$

KLDA 的学习目标是：

$$\max_w J(w) \frac{w^T S_b^\phi w}{w^T S_w^\phi w} \tag{4.103}$$

其中S_b^ϕ和S_w^ϕ分别为训练样本在特征空间 F 中的类间散度矩阵和类内散度矩阵。令X_i表示第$i \in \{0,1\}$类样本的集合，其样本数为m_i；总样本数 m=m_0+m_1。第 i 类样本在特征空间 F 中的均值为：

$$\mu_i^\phi = \frac{1}{m_i} \sum_{x \in X_i} \phi(x) \tag{4.104}$$

两个散度矩阵分别为：

$$S_b^\phi = \left(\mu_1^\phi - \mu_0^\phi\right)\left(\mu_1^\phi - \mu_0^\phi\right)^T \tag{4.105}$$

$$S_w^\phi = \sum_{i=0}^1 \sum_{x \in X_i} \left(\phi(x) - \mu_i^\phi\right)\left(\phi(x) - \mu_i^\phi\right)^T \tag{4.106}$$

通常我们难以知道映射ϕ的具体形式，因此使用核函数$\kappa(x, x_i) = \phi(x_i)^T \phi(x)$来隐式地表达这个映射和特征空间 F。把$J(w)$作为式（4.100）中的损失函数ι，再令Ω ≡ 0，由表示定理，函数$h(x)$可写为：

$$h(x) = \sum_{i=1}^m \alpha_i \kappa(x, x_i) \tag{4.107}$$

于是由式（4.100）可得：

$$w = \sum_{i=1}^m \alpha_i \phi(x_i) \tag{4.108}$$

令$K \in \mathbb{R}^{m \times m}$为核函数κ所对应的核矩阵，$(K)_{ij} = \kappa(x_i, x_j)$。令$I_i \in \{1,0\}^{m \times 1}$为第 i 类样本的指示向量，即$I_i$的第 j 个分量为 1 当且仅当$x_j \in X_i$，否则$I_i$的第 j 个分量为 0。再令：

$$\hat{\mu}_0 = \frac{1}{m_0} K I_0 \tag{4.109}$$

$$\hat{\mu}_1 = \frac{1}{m_1} K I_1 \tag{4.110}$$

$$M = (\hat{\mu}_0 - \hat{\mu}_1)(\hat{\mu}_0 - \hat{\mu}_1)^T \tag{4.111}$$

$$N = KK^T - \sum_{i=0}^1 m_i \hat{\mu}_i \hat{\mu}_i^T \tag{4.112}$$

于是，式（4.101）等价为：

$$\max_\alpha J(\alpha) = \frac{\alpha^T M \alpha}{\alpha^T N \alpha} \tag{4.113}$$

显然，使用线性判别分析求解方法即可得到α。

4. 核支持向量机实验

1） 线性模型与非线性特征

线性模型在低维空间中可能非常受限，因为线和平面的灵活性有限。有一种方法可以让线性模型更加灵活买就是添加更多的特征——举个例子，添加输入特征的交互项或多项式。

我们来看以下模拟数据集：

```
from sklearn.datasets import make_blobs
import matplotlib.pyplot as plt
import mglearn

X,y = make_blobs(centers=4,random_state=8)
y = y%2
mglearn.discrete_scatter(X[:,0],X[:,1],y)
plt.xlabel("Feature 0")
plt.ylabel("Feature 1")

'''
```

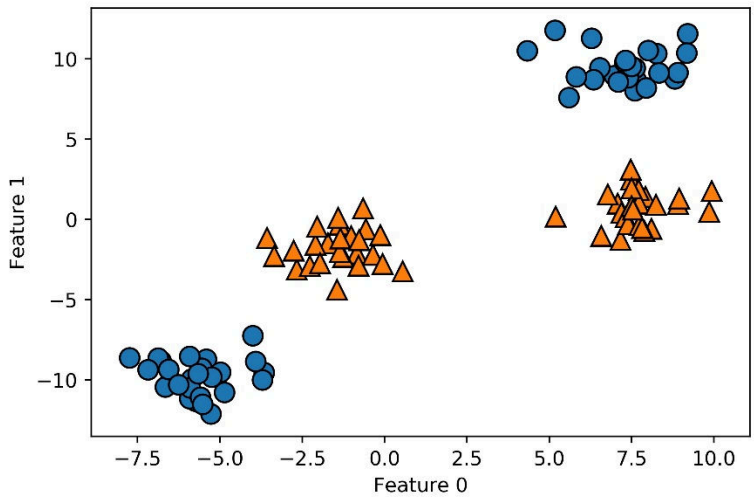


图 4.46 二分类数据集，其类别并不是线性可分的

用于分类的线性模型只能用一条直线来划分数据点，对这个数据集无法给出较好的结果（见图 4.47）：

```
import matplotlib.pyplot as plt
from sklearn.datasets import make_blobs
from sklearn.svm import LinearSVC
import mglearn

X,y = make_blobs(centers=4,random_state=8)
y = y%2
linear_svm = LinearSVC().fit(X,y)
mglearn.plots.plot_2d_separator(linear_svm,X)
mglearn.discrete_scatter(X[:,0],X[:,1],y)
plt.xlabel("Feature 0")
plt.ylabel("Feature 1")

'''
```

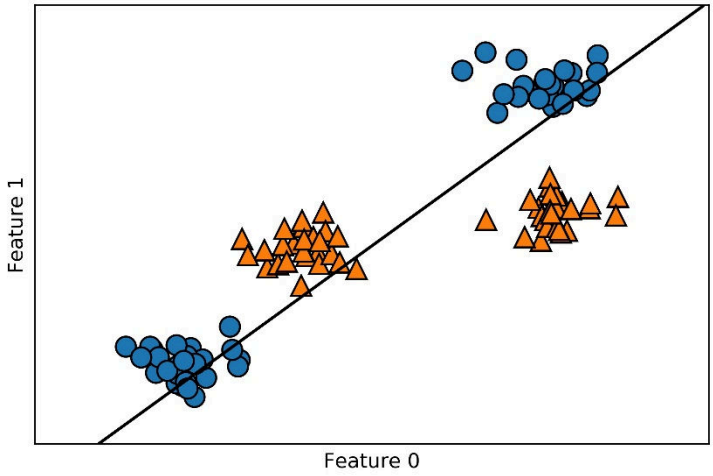


图 4.47 二线性 SVM 给出的决策边界

现在我们对输入特征进行扩展，比如说添加第二个特征的平方（feature**2）作为一个新特征。现在我们将每个数据点表示为三维点（feature0,feature1,feature1**2），而不是二维点（feature0,feature1）。这个新的表示可以画成图 4.48 中的三维散点图：

```
import matplotlib.pyplot as plt
from sklearn.datasets import make_blobs
from sklearn.svm import LinearSVC
from mpl_toolkits.mplot3d import Axes3D,axes3d
import numpy as np
import mglearn

X,y = make_blobs(centers=4,random_state=8)
y = y%2
#添加第二个特征的平方，作为一个新特征
X_new = np.hstack([X,X[:,1:]**2])
figure = plt.figure()
#将 3D 可视化
ax = Axes3D(figure,elev=-152,azim=-26)
#首先画出所有 y==0 的点，然后画出所有 y==1 的点
mask = y == 0
ax.scatter(X_new[mask,0],X_new[mask,1],X_new[mask,2],c='b',cmap=mglearn.cm2,s=60)
ax.scatter(X_new[~mask,0],X_new[~mask,1],X_new[~mask,2],c='r',marker='^',
cmap=mglearn.cm2,s=60)
ax.set_xlabel("feature0")
ax.set_ylabel("feature1")
ax.set_zlabel("feature1**2")

'''
```

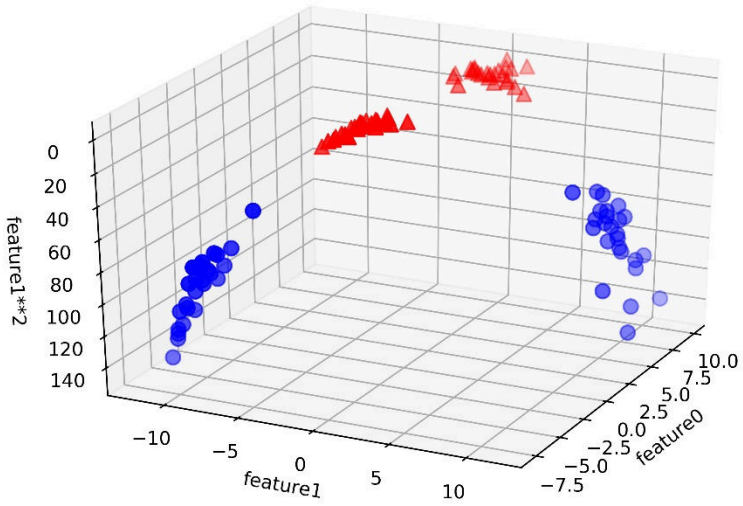


图 4.48 对图 4.47 中的数据集进行扩展，新增由 feature1 导出的第三个特征

在数据的新表示中，现在可以用线性模型（三维空间中的平面）将这两个类别分开。我们可以用线性模型拟合扩展后的数据来验证这一点（见图 4.49）：

```
    #添加第二个特征的平方，作为一个新特征
    X_new = np.hstack([X,X[:,1:]**2])
    linear_svm_3d = LinearSVC().fit(X_new,y)
    coef,intercept = linear_svm_3d.coef_.ravel(),linear_svm_3d.intercept_

    #显示线性决策边界
    figure = plt.figure()
    ax = Axes3D(figure,elev=-152,azim=-26)
    xx = np.linspace(X_new[:,0].min()-2,X_new[:,0].max()+2,50)
    yy = np.linspace(X_new[:,1].min()-2,X_new[:,1].max()+2,50)
    XX,YY = np.meshgrid(xx,yy)
    ZZ = (coef[0]*XX+coef[1]*YY+intercept/-coef[2])
    mask = y == 0
    ax.plot_surface(XX,YY,ZZ,rstride=8,cstride=8,alpha=0.3)
    ax.scatter(X_new[mask,0],X_new[mask,1],X_new[mask,2],c='b',cmap=mglearn.cm2,s=60)
    ax.scatter(X_new[~mask,0],X_new[~mask,1],X_new[~mask,2],c='r',marker='^',cmap=mglearn.c
m2,s=60)
    ax.set_xlabel("feature0")
    ax.set_ylabel("feature1")
    ax.set_zlabel("feature1**2")

    ,,,
```

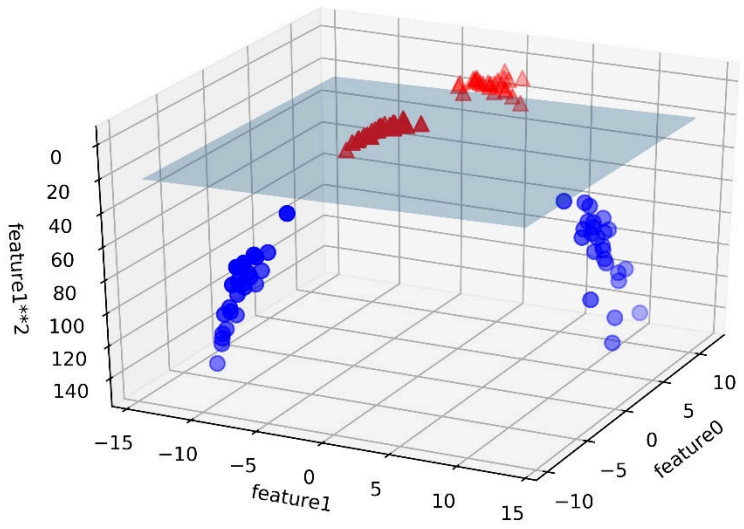


图 4.49 线性 SVM 对扩展后的三维数据集给出的决策边界

如果将线性 SVM 模型看作原始特征的函数，那么它实际上已经不是线性的了。它不是一条直线，而是一个椭圆，你可以在下图中看出（图 4.50）：

```
import matplotlib.pyplot as plt
from sklearn.datasets import make_blobs
from sklearn.svm import LinearSVC
import numpy as np
import mglearn
X,y = make_blobs(centers=4,random_state=8)
y = y%2

#添加第二个特征的平方，作为一个新特征
X_new = np.hstack([X,X[:,1:]**2])
linear_svm_3d = LinearSVC().fit(X_new,y)
coef,intercept = linear_svm_3d.coef_.ravel(),linear_svm_3d.intercept_

#显示线性决策边界
figure = plt.figure()
xx = np.linspace(X_new[:,0].min()-2,X_new[:,0].max()+2,50)
yy = np.linspace(X_new[:,1].min()-2,X_new[:,1].max()+2,50)
XX,YY = np.meshgrid(xx,yy)
ZZ = YY**2
dec = linear_svm_3d.decision_function(np.c_[XX.ravel(),YY.ravel(),ZZ.ravel()])
plt.contourf(XX,YY,dec.reshape(XX.shape),levels=[dec.min(),0,dec.max()])
mglearn.discrete_scatter(X[:,0],X[:,1],y)
plt.set_xlabel("feature0")
plt.set_ylabel("feature1")

'''
```

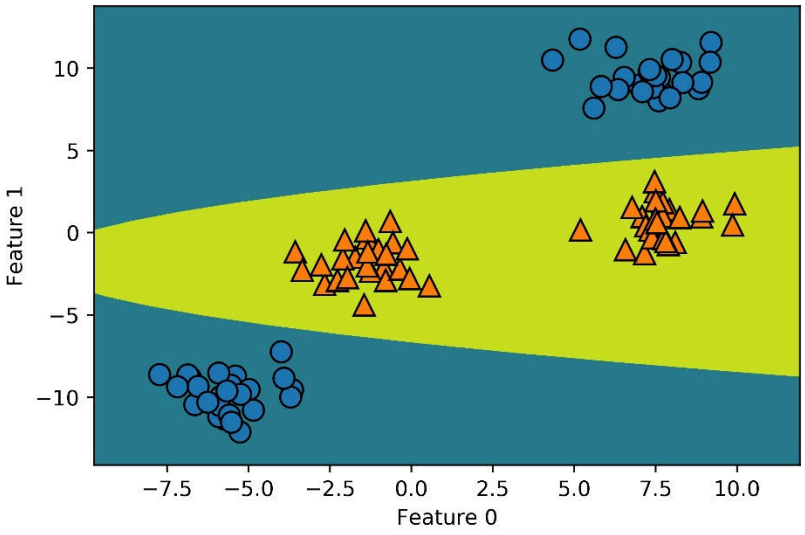


图 4.50 将图 4.49 给出的决策边界作为两个原始特征的函数

2） 核技巧

这里需要记住的是，向数据表示中添加非线性特征，可以让线性模型变得更强大。但是，通常来说我们并不知道要添加哪些特征，而且添加许多特征（比如 100 维特征空间所有可能的交互项）的计算开销可能会很大。幸运的是，有一种巧妙的数学技巧，让我们可以在更高维空间中学习分类器，而不用实际计算可能非常大的新的数据表示。这种技巧叫作核技巧(kernel trick)，它的原理是直接计算扩展特征表示中数据点之间的距离（更准确地说是内积），而不用实际对扩展进行计算。

对于支持向量机，将数据映射到更高维空间中有两种常用的方法：一种是多项式核，在一定阶数内计算原始特征所有可能的多项式（比如 feature1**2 *feature2**5）；另一种是径向基函数(radial basis function，RBF)核，也叫高斯核。高斯核有点难以解释，因为它对应无限维的特征空间。一种对高斯核的解释是它考虑所有阶数的所有可能的多项式，但阶数越高，特征的重要性越小。

不过在实践中，核 SWM 背后的数学细节并不是很重要，可以简单地总结出使用 RBF 核 SVM 进行预测的方法一一我们将在下一节介绍这方面的内容。

3） 理解 SVM

在训练过程中，SVM 学习每个训练数据点对于表示两个类别之间的决策边界的重要性。通常只有一部分训练数据点对于定义决策边界来说很重要：位于类别之间边界上的那些点。这些点叫作支持向量(support vector)，支持向量机正是由此得名。

想要对新样本点进行预测，需要测量它与每个支持向量之间的距离。分类决策是基于它与支持向量之间的距离以及在训练过程中学到的支持向量重要性（保存在 SVC 的 dual_coef_属性中）来做出的。

数据点之间的距离由高斯核给出：

$$\kappa(x_i, x_j) = exp(-\gamma||x_i - x_j||^2) \qquad (4.114)$$

这里是x_i和x_j数据点，$||x_i - x_j||$表示欧氏距离，γ（gamma）是控制高斯核宽度的参数。

下列代码将在 forge 数据集上训练 SVM 并创建此图：

```
from sklearn.svm import SVC
X,y = mglearn.tools.make_handcrafted_dataset()
svm = SVC(kernel='rbf',C=10,gamma=0.1).fit(X,y)
mglearn.plots.plot_2d_separator(svm,X,eps=.5)
mglearn.discrete_scatter(X[:,0],X[:,1],y)
#画出支持向量
sv = svm.support_vectors_
#支持向量的类别标签由 dual_coef_的正负号给出
sv_labels = svm.dual_coef_.ravel()>0
mglearn.discrete_scatter(sv[:,0],sv[:,1],sv_labels,s=15,markeredgewidth=3)
plt.xlabel("feature 0")
plt.ylabel("feature 1")

,,,
```

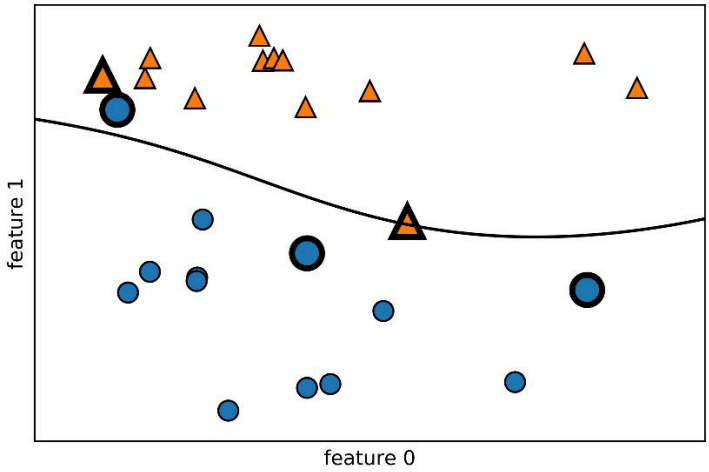


图 4.51 RBF 核 SVM 给出的决策边界和支持向量

在这个例子中，SVM 给出了非常平滑且非线性（不是直线）的边界。这里我们调节了两个参数：C 参数和 gamma 参数，下面我们将详细讨论。

4） SVM 调参

gamma 参数用于控制高斯核的宽度。它决定了点与点之间“靠近”是指多大的距离。C 参数是正则化参数，与线性模型中用到的类似。它限制每个点的重要性（或者更确切地说，每个点的 dual_coef_）。

我们来看一下，改变这些写参数时会发生什么（图 4.52）：

```
import matplotlib.pyplot as plt
from sklearn.datasets import make_blobs
from sklearn.svm import SVC
import numpy as np
import mglearn

fig,axes = plt.subplots(3,3,figsize=(15,10))
for ax,C in zip(axes,[-1,0,3]):
    for a,gamma in zip(ax,range(-1,2)):
        mglearn.plots.plot_svm(log_C=C,log_gamma=gamma,ax=a)
axes[0,0].lengend(["class 0","class 1","sv class 0","sv class 1"],ncol=4,loc=(.9,1.2))

,,,
```

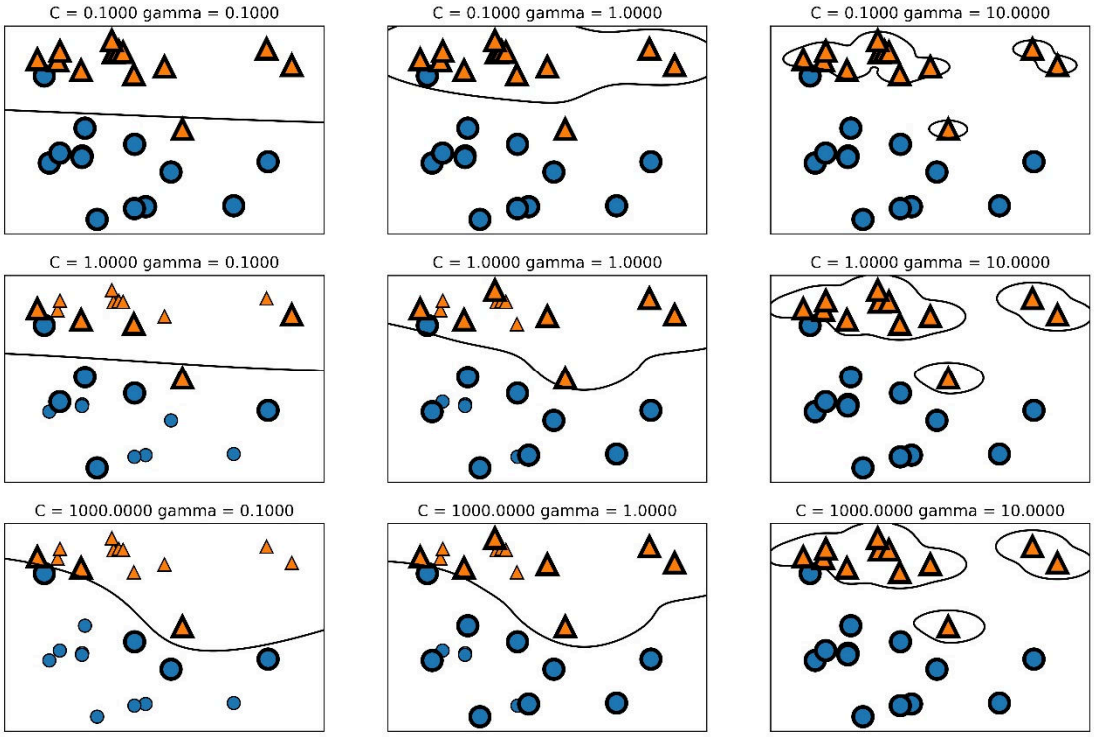


图 4.52 设置不同的 C 和 gamma 参数对应的决策边界和支持向量

从左到右，我们将参数 gamma 的值从 0.1 增加到 10。gamma 较小，说明高斯核的半径较大许多点都被看作比较靠近。这一点可以在图中看出：左侧的图决策边界非常平滑，越向右的图决策边界更关注单个点。小的 gamma 值表示决策边界变化很慢，生成的是复杂度较低的模型，而大的 gamma 值则会生成更为复杂的模型。

从上到下，我们将参数 C 的值从 0.1 增加到 1000。与线性模型相同，C 值很小，说明模型非常受限，每个数据点的影响范围都有限。你可以看到，左上角的图中，决策边界看起来几乎是线性的，误分类的点对边界几乎没有任何影响。再看左下角的图，增大 C 之后这些点对模型的影响变大，使得决策边界发生弯曲来将这些点正确分类。

我们将 RBF 核 SVM 应用到乳腺癌数据集集上。默认情况下，C=1，gamma=1/n_ features：

```
X_train,X_test,y_train,y_test = train_test_split(cancer.data,cancer.target,random_state=0)
svc = SVC()
svc.fit(X_train,y_train)
print("Accuracy on training set:{:.2f}".format(svc.score(X_train,y_train)))
print("Accuracy on test set:{:.2f}".format(svc.score(X_test,y_test)))

'''
运行以上程序可以得到输出:
Accuracy on training set:0.90
Accuracy on test set:0.94
```

这个模型在训练集上的分数十分完美，在测试集上的精度 94%。虽然 SVM 的表现通常都很好，但它对参数的设定和数据的缩放非常敏感。特别地，它要求所有特征有相似的变化范围。我们来看一下每个特征的最小值和最大值，它们绘制在对数坐标上（图 4.53）：

```
plt.plot(X_train.min(axis=0),'o',label="min")
plt.plot(X_train.max(axis=0),'^',label="max")
plt.legend(loc=4)
plt.xlabel("feature index")
plt.ylabel("feature magnitude")
plt.yscale("log")

'''
```

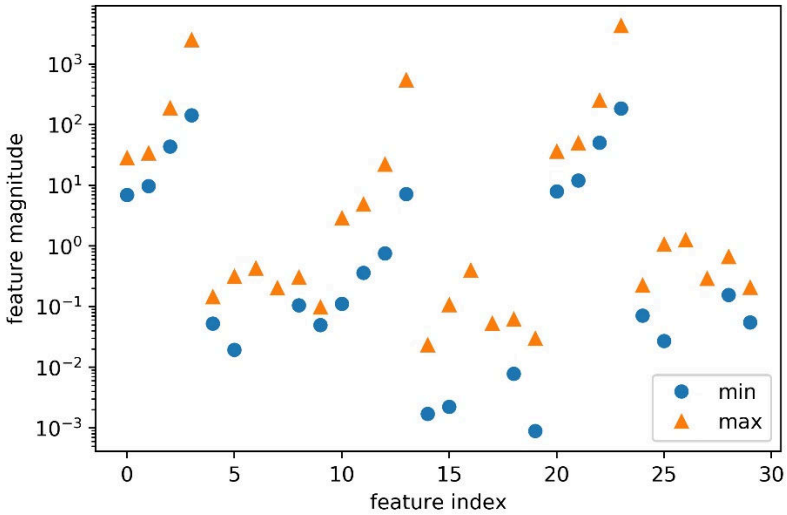


图 4.53 乳腺癌数据集的特征范围（注意 y 轴的对数坐标）

从这张图中，我们可以确定乳腺癌数据集的特征具有完全不同的数量级。这对其他模型来说（比如线性模型）可能是小问题，但对核 SVM 却有极大影响。我们来研究处理这个问题的几种方法。

5） 为 SVM 预处理数据

解决这个问题的一种方法就是对每个特征进行缩放，使其大致都位于同一范围。核 SVM 常用的缩放方法就是将所有特征缩放到 0 和 1 之间。

```
#计算训练集中每个特征的最小值
min_on_training = X_train.min(axis=0)
#计算训练集中每个特征的范围（最大值-最小值)
range_on_training = (X_train-min_on_training).max(axis=0)

#减去最小值，然后除以范围
#这样每个特征向量都是 min=0 和 max=1
X_train_scaled = (X_train-min_on_training)/range_on_training
print("Minimum for each feature\n{}".format(X_train_scaled.min(axis=0)))
print("Maximum for each feature\n{}".format(X_train_scaled.max(axis=0)))

‘’’
运行以上程序可以得到输出:
Minimum for each feature
[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.]
Maximum for each feature
[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.]
```

```
#利用训练集的最小值和范围对测试集做相同的变换
X_test_scaled = (X_test-min_on_training)/range_on_training
svc = SVC()
svc.fit(X_train_scaled,y_train)
print("Accuracy on training set:{:.3f}".format(svc.score(X_train_scaled,y_train)))
print("Accuracy on test set:{:.3f}".format(svc.score(X_test_scaled,y_test)))

‘’’
运行以上程序可以得到输出:
Accuracy on training set:0.948
Accuracy on test set:0.951
```

数据缩放的作用很大！实际上模型现在处于欠拟合的状态，因为训练集和测试集的性能非常接近，但还没有接近 100％的精度。从这里开始，我们可以尝试增大 C 或 gamma 来拟合更为复杂的模型。例如：

```
svc = SVC(C=1000)
svc.fit(X_train_scaled,y_train)
print("Accuracy on training set:{:.3f}".format(svc.score(X_train_scaled,y_train)))
print("Accuracy on test set:{:.3f}".format(svc.score(X_test_scaled,y_test)))

‘’’
运行以上程序可以得到输出:
Accuracy on training set:0.988
Accuracy on test set:0.972
```

在这个例子中，增大 C 可以显著改进模型，得到 97.2%的精度。

6） 优点、缺点和参数

核支持向量机是非常强大的模型，在各种数据集上的表现都很好。SVM 允许决策边界很复杂，即使数据只有几个特征。它在低维数据和高维数据（即很少特征和很多特征）上的表现都很好，但对样本个数的缩放表现不好。在有多达 10000 个样本的数据上运行 SVM 可能表现良好，但如果数据量达到 100000 甚至更大，在运行时间和内存使用方面可能会面临挑战。

SVM 的另一个缺点是，预处理数据和调参都需要非常小心。这也是为什么如今很多应用中用的都是基于树的模型，比如随机森林或梯度提升（需要很少的预处理，甚至不需要预处理）。此外，SVM 模型很难检查，可能很难理解为什么会这么预测，而且也难以将模型向非专家进行解释。不过 SVM 仍然是值得尝试的，特别是所有特征的测量单位相似（比如都是像素密度）而且范围也差不多时。

核 SVM 的重要参数是正则化参数 C、核的选择以及与核相关的参数。虽然我们主要讲的是 RBF 核，但 scikit-learn 中还有其他选择。RBF 核只有一个参数 gamma，它是高斯核宽度的倒数。gamma 和 C 控制的都是模型复杂度，较大的值都对应更为复杂的模型。因此，这两个参数的设定通常是强烈相关的，应该同时调节。

4.4.6 神经网络

1. 神经元模型

神经网络(neural networks)方面的研究很早就已出现，今天“神经网络”已是一个相当大的、多学科交叉的学科领域。各相关学科对神经网络的定义多种多样，本书采用目前使用得最广泛的一种，即“神经网络是由具有适应性的简单单元组成的广泛并行互连的网络，它的组织能够模拟生物神经系统对真实世界物体所作出的交互反应”[Kohonen, 1988] .。我们在机器学习中谈论神经网络时指的是“神经网络学习”，或者说，是机器学习与神经网络这两个学科领域的交叉部分。

神经网络中最基本的成分是神经元(neuron)模型，即上述定义中的“简单单元”。在生物神经网络中，每个神经元与其他神经元相连，当它“兴奋”时，就会向相连的神经元发送化学物质，从而改变这些神经元内的电位；如果某神经元的电位超过了一个“阈值”(threshold)，那么它就会被激活，即“兴奋”起来，向其他神经元发送化学物质。

1943 年，[McCulloch and Pitts, 1943]将上述情形抽象为图 4.54 所示的简单模型，这就是一直沿用至今的“M-P 神经元模型”。在这个模型中，神经元接收到来自 n 个其他神经元传递过来的输入信号，这些输入信号通过带权重的连接(connection)进行传递，神经元接收到的总输入值将与神经元的阈值进行比较，然后通过“激活函数”(activation function)处理以产生神经元的输出。

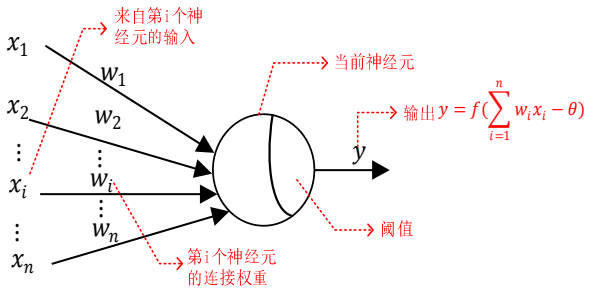
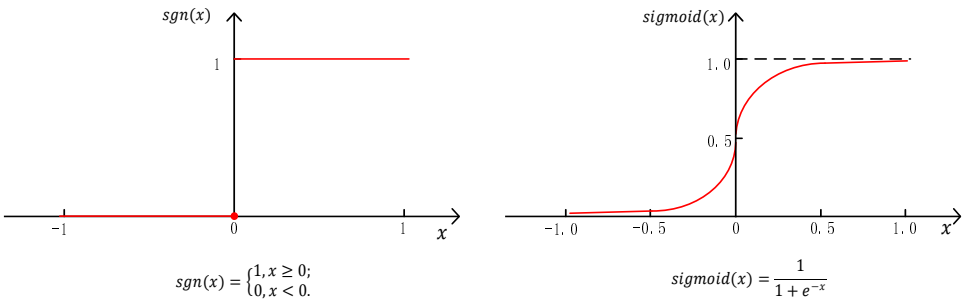


图 4.54 MLP 神经元模型

理想中的激活函数是图 4.55(a)所示的阶跃函数，它将输入值映射为输出值“0”或“1”，显然“1”对应于神经元兴奋，“0”对应于神经元抑制。然而，阶跃函数具有不连续、不光滑等不太好的性质，因此实

际常用 Sigmoid 函数作为激活函数，典型的 Sigmoid 函数如图 4.55(b)所示，它把可能在较大范围内变化的输入值挤压到(0，1)输出值范围内，因此有时也称为“挤压函数”(squashing function)。



$sgn(x) = \begin{cases} 1, x \geq 0; \\ 0, x < 0. \end{cases}$

(a) 阶跃函数

$sigmoid(x) = \frac{1}{1+e^{-x}}$

(b) Sigmoid函数

图 4.55 典型的神经元激活函数

把许多个这样的神经元按一定的层次结构连接起来，就得到了神经网络。事实上，从计算机科学的角度看，我们可以先不考虑神经网络是否真的模拟了生物神经网络，只需将一个神经网络视为包含了许多参数的数学模型，这个模型是若干个函数，例如$y_i = f\left(\sum_i w_i x_i - \theta_j\right)$相互（嵌套）代入而得。有效的神经网络学习算法大多以数学证明为支撑。

2. 感知机与多层网络

感知机(Perceptron)由两层神经元组成，如图 4.56 所示，输入层接收外界输入信号后传递给输出层，输出层是 M-P 神经元，亦称“阈值逻辑单元”(threshold logic unit)。

感知机能容易地实现逻辑与、或、非运算。注意到 y= $f(\sum_i w_i x_i - \theta)$，假定 f 是图 4.55 中的阶跃函数，有：

- “与”$(x_1 \wedge x_2)$：令$w_1 = w_2 = 1$，$\theta = 2$，则y $= f(1 \cdot x_1 + 1 \cdot x_2 - 2)$，仅在$x_1 = x_2 = 1$时，y $= 1$；
- “或”$(x_1 \vee x_2)$：令$w_1 = w_2 = 1$，$\theta = 0.5$,，则y $= f(1 \cdot x_1 + 1 \cdot x_2 - 0.5)$，当$x_1 = 1$或$x_2 = 1$时，y $= 1$；
- “非”$(\neg x_1)$：令$w_1 = -0.6$，$w_2 = 0$，$\theta = -0.5$，则y $= f(-0.6 \cdot x_1 + 0 \cdot x_2 + 0.5)$，当$x_1 = 1$时，y $= 0$；当$x_1 = 0$时，y $= 1$。

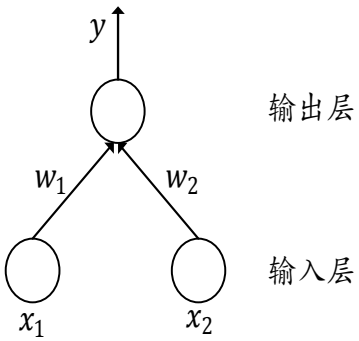


图 4.56 两个输入神经元的感知机网络结构示意图

更一般地，给定训练数据集，权重$w_i(i=1,2,\cdots,n)$以及阈值θ可通过学习得到。阈值θ可看作一个固定输入为-1.0 的“哑结点”(dummy node)所对应的连接权重w_{n+1}，这样，权重和阈值的学习就可统一为权重的学习。感知机学习规则非常简单，对训练样例(x,y)，若当前感知机的输出为$\hat{y}$，则感知机权重将这样调整：

$$w_i \longleftarrow w_i + \Delta w_i \tag{4.115}$$

$$\Delta w_i = \eta(y-\hat{y})x_i \tag{4.116}$$

其中$\eta \in (0,1)$称为学习率(learning rate)。从式（4.72）可看出，若感知机对训练样例(x,y)预测正确，即$\hat{y}=y$，则感知机不发生变化，否则将根据错误的程度进行权重调整。

需注意的是，感知机只有输出层神经元进行激活函数处理，即只拥有一层功能神经元(functional neuron)，其学习能力非常有限。事实上，上述与、或、非问题都是线性可分(linearly separable)的问题，可以证明[Minsky and Papert,1969]，若两类模式是线性可分的，即存在一个线性超平面能将它们分开，如图4.30(a)-(c)所示，则感知机的学习过程一定会收敛(converge)而求得适当的权向量$w(w_1,w_2,\cdots,w_{n+1})$；否则感知机学习过程将会发生振荡(Fluctuation)，w难以稳定下来，不能求得合适解，例如感知机甚至不能解决如图4.30(d)所示的异或这样简单的非线性可分问题。

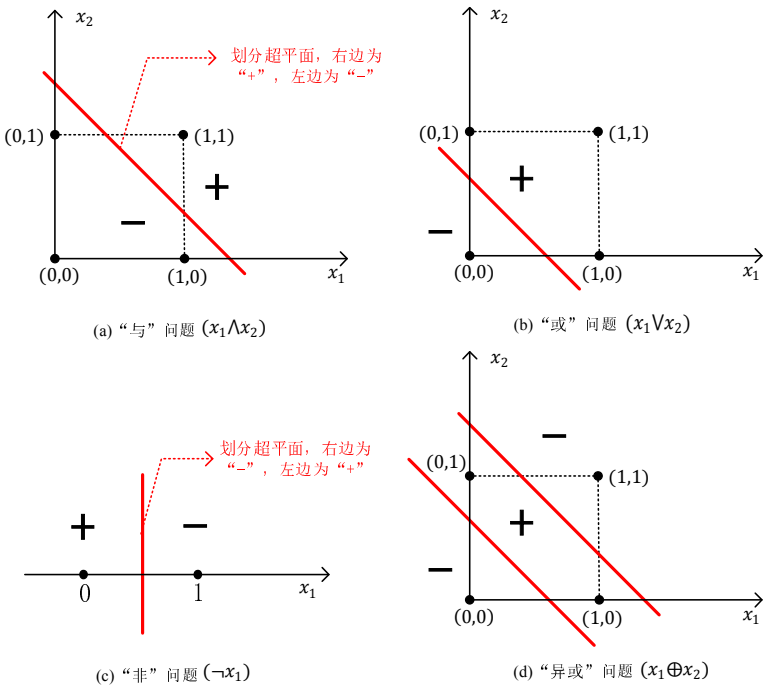
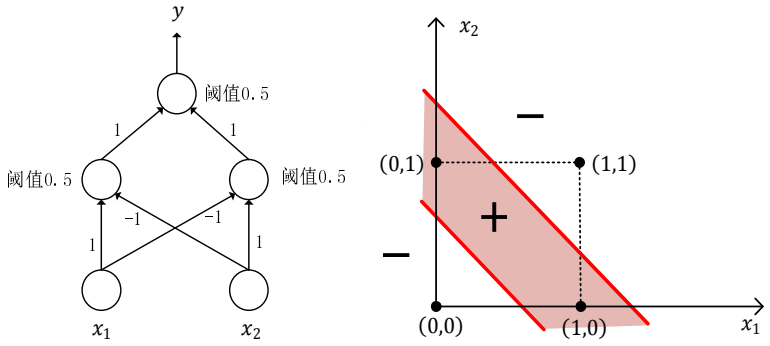


图 4.57 线性可分的“与”“或”“非”问题与非线性可分的“异或”问题

要解决非线性可分问题，需考虑使用多层功能神经元。例如图 4.58 中这个简单的两层感知机就能解决异或问题。在图 4.58(a)中，输出层与输入层之间的层神经元，被称为隐层或隐含层(hidden layer)，隐含层和输出层神经元都是拥有激活函数的功能神经元。

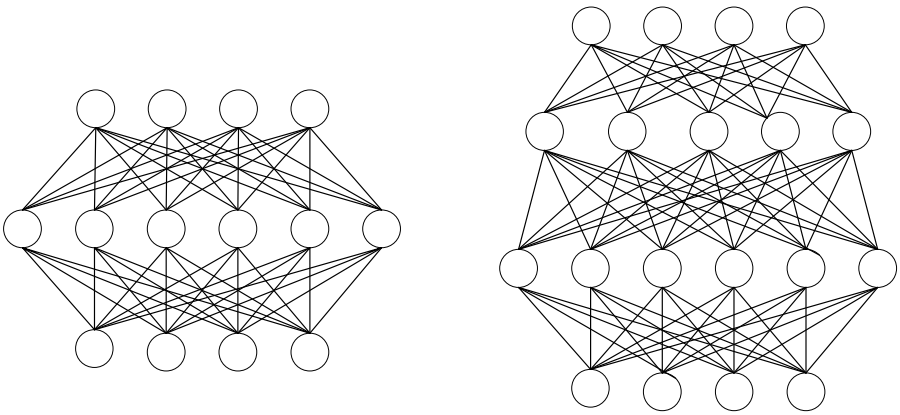
更一般的，常见的神经网络是形如图 4.59 所示的层级结构，每层神经元与下一层神经元全互连，神经元之间不存在同层连接，也不存在跨层连接。这样的神经网络结构通常称为“多层前馈神经网络”(multi-layer feedforward neural networks)，其中输入层神经元接收外界输入，隐层与输出层神经元对信号进行加工，最终结果由输出层神经元输出；换言之，输入层神经元仅是接受输入，不进行函数处理，隐层与输出层包

含功能神经元，因此，图 4.59(a)通常被称为“两层网络”。为避免歧义，本书称其为“单隐层网络”只需包含隐层，即可称为多层网络。神经网络的学习过程，就是根据训练数据来调整神经元之间的“连接权”(connection weight)以及每个功能神经元的值；换言之，神经网络“学”到的东西，蕴涵在连接权与阈值中。



(a) 网络结构　　(b) 分类区域

图 4.58　能解决异或问题的两层感知机



(a) 单隐层前馈网络　　(b) 双隐层前馈网络

图 4.59　多层前馈神经网络结构示意图

3.　误差逆传播算法

多层网络的学习能力比单层感知机强得多。欲训练多层网络，式(4.115)的简单感知机学习规则显然不够了，需要更强大的学习算法。误差逆传播(error BackPropagation，简称 BP)算法就是其中最杰出的代表，它是迄今最成功的神经网络学习算法，现实任务中使用神经网络时，大多是在使用 BP 算法进行训练。值得指出的是，BP 算法不仅可用于多层前馈神经网络，还可用于其他类型的神经网络，例如训练递归神经网络[Pineda, 1987]。但通常说“BP 网络”时，一般是指用 BP 算法训练的多层前馈神经网络。

下面我们来看看 BP 算法究竟是什么样。给定训练集集$D = \{(x_1, y_1), (x_2, y_2), \cdots, (x_m, y_m)\}$，$x_i \in \mathbb{R}^d$，$y_i \in \mathbb{R}^l$，即输入示例由 d 个属性描述，输出$l$维实值向量。为便于讨论，图 4.60 给出了一个拥有 d 个输入神经元、l个输出神经元、q 个隐层神经元的多层前馈网络结构，其中输出层第 j 个神经元的阈值用θ_j表示，隐层第 h 个神经元的阈值用$\Upsilon_{\hbar}$表示。输入层第 i 个神经元与隐层第 h 个神经元之间的连接权为v_{ih}，隐层第 h

个神经元与输出层第 j 个神经元之间的连接权为w_{hj}。记隐层第 h 个神经元接收到的输入为$\alpha_h = \sum_{i=1}^{d} v_{ih}\, x_i$，输出层第 j 个神经元接收到的输入为$\beta_h = \sum_{h=1}^{q} w_{hj}\, b_h$，其中$b_h$为隐层第 h 个神经元的输出。假设隐层和输出神经元都使用图 4.55(b)中的 Sigmoid 函数。

对训练例$(x_k,\ y_k)$，假定神经网络的输出为$\hat{y}_k = (\hat{y}_1^k, \hat{y}_2^k, \cdots, \hat{y}_l^k)$，即：

$$\hat{y}_k = f(\beta_j - \theta_j) \tag{4.117}$$

则网络在$(x_k,\ y_k)$上的均方误差为：

$$E_k = \frac{1}{2}\sum_{j=1}^{l}(\hat{y}_j^k - y_j^k)^2 \tag{4.118}$$

图 4.60 的网络中有$(d + l + l)\mathrm{q} + l$个参数需确定：输入层到隐层的 d×q 个权值、隐层到输出层的 q×l个权值、q 个隐层神经元的阈值、l个输出层神经元的阈值。BP 是一个迭代学习算法，在迭代的每一轮中采用广义的感知机学习规则对参数进行更新估计，即与式(4.115)类似，任意参数 v 的更新估计式为：

$$v \longleftarrow v + \Delta \mathrm{v} \tag{4.119}$$

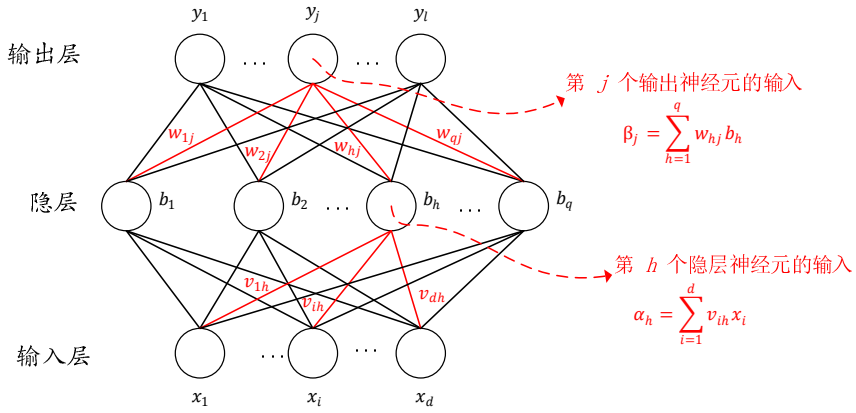


图 4.60　BP 网络及算法中的变量符号

下面我们以图 4.60 中隐层到输出层的连接权w_{hj}为例来进行推导。

BP 算法基于梯度下降(gradient descent)策略，以目标的负梯度方向对参数进行调整。对式(4.118)的误差E_k，给定学习率η，有：

$$\Delta w_{hj} = -\eta \frac{\partial E_k}{\partial w_{hj}} \tag{4.120}$$

注意到w_{hj}先影响到第 j 个输出神经元的输入值β_j，再影响到其输出值$\hat{y}_j^k$，然后影响到E_k，有：

$$\frac{\partial E_k}{\partial w_{hj}} = \frac{\partial E_k}{\partial \hat{y}_j^k} \cdot \frac{\partial \hat{y}_j^k}{\partial \beta_j} \cdot \frac{\partial \beta_j}{\partial w_{hj}} \tag{4.121}$$

根据β_j的定义，显然有：

$$\frac{\partial \beta_j}{\partial w_{hj}} = b_h \tag{4.122}$$

图 4.28 中的 Sigmoid 函数有一个很好的性质：

$$f(x) = f(x)\big(1 - f(x)\big) \tag{4.123}$$

于是根据式（4.117）和（4.118），有：

$$\begin{aligned} g_i &= -\frac{\partial E_k}{\partial \hat{y}_j^k} \cdot \frac{\partial \hat{y}_j^k}{\partial \beta_j} \\ &= -\big(\hat{y}_j^k - y_j^k\big) f'\big(\beta_j - \theta_j\big) \\ &= \hat{y}_j^k\big(1 - \hat{y}_j^k\big)\big(y_j^k - \hat{y}_j^k\big) \end{aligned} \tag{4.124}$$

将式（4.81）和（4.79）代入式（4.78），再代入式（4.77），就得到了 BP 算法中关于w_{hj}的更新公式：

$$\Delta w_{hj} = \eta g_i b_h \tag{4.125}$$

类似可得：

$$\Delta \theta_j = -\eta g_i \tag{4.126}$$

$$\Delta v_{i\hbar} = \eta e_h x_x \tag{4.127}$$

$$\Delta \Upsilon_h = -\eta e_h \tag{7.128}$$

式（7.128）和（7.127）中：

$$\begin{aligned} e_h &= -\frac{\partial E_k}{\partial b_h} \cdot \frac{\partial b_h}{\partial \alpha_h} \\ &= -\sum_{j=1}^{l} \frac{\partial E_k}{\partial \beta_j} \cdot \frac{\partial \beta_j}{\partial \alpha_h} f'(\alpha_h - \Upsilon_h) \\ &= \sum_{j=1}^{l} w_{hj} g_i f'(\alpha_h - \Upsilon_h) \\ &= b_h(1 - b_h) \sum_{j=1}^{l} w_{hj} g_i \end{aligned} \tag{7.129}$$

学习率$\eta \in (0,1)$控制着算法每一轮迭代中的更新步长，若太大则容易振荡，太小则收敛速度又会过慢，有时为了做精细调节，可令式（4.125）与（4.126）使用η_1，式（4.127）与（4.128）使用η_2，两者未必相等。

图 4.61 给出了 BP 算法的工作流程。对每个训练样例，BP 算法执行以下操作：先将输入示例提供给输入层神经元，然后逐层将信号前传，直到产生输出层的结果；然后计算输出层的误差（第 4-5 行），再将误差逆向传播至隐层神经元（第 6 行），最后根据隐层神经元的误差来对连接权和阈值进行调整（第 7 行）该迭代过程循环进行，直到达到某些停止条件为止。例如训练误差已达到一个很小的值。图 4.62 给出了在 2 个属性、5 个样本的西瓜数据上，随着训练轮数的增加，网络参数和分类边界的变化情况。

输入：训练集$D=\{(x_1,y_1),(x_2,y_2),\cdots,(x_m,y_m)\}$

属性集$A=\{a_1,a_2,\cdots,a_d\}$

过程：函数 TreeGenerate(D,A)

1：生成结点 node；

2：if D中样本全属于同一类别C then

3：　　将 node 标记为C类叶结点；return

4：end if

5：if $A=\emptyset$ OR D中样本在A上取值相同 then

6：　　将 node 标记为叶结点，其类别标记为D中样本数最多的类；return

7：end if

8：从A中选择最优划分属性a_*；

9：for a_*的每一个值a_*^v do

10：　　为 node 生成一个分支；令D_v表示D中在a_*上取值为a_*^v的样本子集；

11：　　if D_v为空 then

12：　　　　将分支结点标记为叶结点，其类别标记为D中样本最多的类；return

13：　　else

14：　　　　以 TreeGenerate$(D_v,A\{a_*\})$为分支结点

15：　　end if

16：end for

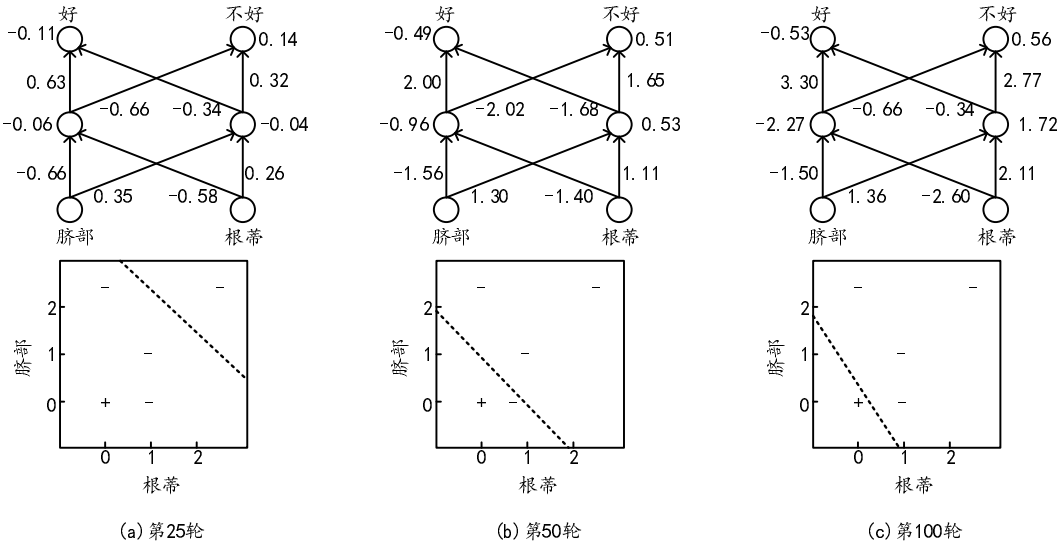
输出：以 node 为根结点的一棵决策树

图 4.61　误差逆传播算法

需注意的是，BP 算法的目标是要最小化训练集 D 上的累积误差：

$$\mathrm{E}=\frac{1}{m}\sum_{k=1}^{m}E_k \tag{4.130}$$

但我们上面介绍的“标准 BP 算法”每次仅针对一个训练样例更新连接权和阈值，也就是说，图 4.61 中算法的更新规则是基于单个的E_k推导而得。如果类似地推导出基于累积误差最小化的更新规则，就得到了累积误差逆传播(accumulated error backpropagation)算法。累积 BP 算法与标准 BP 算法都很常用。一般来说，标准 BP 算法每次更新只针对单个样例，参数更新得非常频繁，而且对不同样例进行更新的效果可能出现“抵消”现象。因此，为了达到同样的累积误差极小点，标准 BP 算法往往需进行更多次数的迭代。累积 BP 算法直接针对累积误差最小化，它在读取整个训练集 D 一遍后才对参数进行更新，其参数更新的频率低得多。但在很多任务中，累积误差下降到一定程度之后，进一步下降会非常缓慢，这时标准 BP 往往会更快获得较好的解，尤其是在训练集 D 非常大时更明显。



(a) 第25轮　　(b) 第50轮　　(c) 第100轮

图 4.62 在 2 个属性、5 个样本的西瓜数据上，BP 网络参数更新和分类边界的变化情况

[Hornik et al，1989]证明，只需一个包含足够多神经元的隐层，多层前馈网络就能以任意精度逼近任意复杂度的连续函数。然而，如何设置隐层神经元的个数仍是个未决问题，实际应用中通常靠“试错法”(trial-by-error)调整。

正是由于其强大的表示能力，BP 神经网络经常遭遇过拟合，其训练误差持续降低，但测试误差却可能上升。有两种策略常用来缓解 BP 网络的过拟合。第一种策略是“早停”(early stopping)。将数据分成训练集和验证集，训练集用来计算梯度、更新连接权和阈值，验证集用来估计误差，若训练集误差降低但验证集误差升高，则停止训练，同时返回具有最小验证集误差的连接权和阈值。第二种策略是“正则化”(regularization)，其基本思想是在误差目标函数中增加一个用于描述网络复杂度的部分，例如连接权与阈值的平方和。仍令E_k表示第 k 个训练样例上的误差，w_i表示连接权和阈值，则误差目标函数（4.130）改变为

$$\mathrm{E} = \lambda \frac{1}{m} \sum_{k=1}^{m} E_k + (1-\lambda) \sum_i w_i^2 \qquad (4.131)$$

其中$\lambda \in (0,1)$用于对经验误差与网络复杂度这两项进行折中，常通过交叉验证法来估计。

4. 代码实现

(1) 神经网络模型

MLP 可以被称为广义的线性模型，执行多层处理后得到结论。

```
display(mglearn.plots.plot_logistic_regression_graph())

'''
```

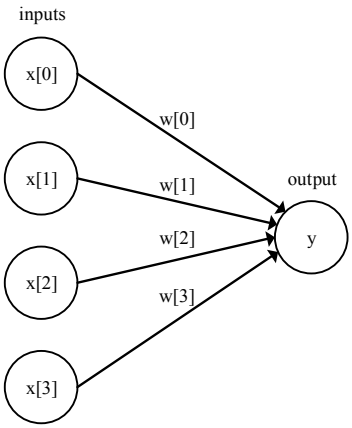


图 4.63 Logistic 回归的可视化，其中输入特征和预测结果显示为结点，系数是节点之间的连线

图中，左边的每个结点代表一个输入特征，连线代表学到的系数，右边的结点代表输出，是输入的加权求和。在 MIP 中，多次重复这个计算加权求和的过程，首先计算代表中间过程的隐单元(hidden unit)，然后再计算这些隐单元的加权求和并得到最终结果(如图 4.64 所示)：

```
display(mglearn.plots.plot_single_hidden_layer_graph())

'''
```

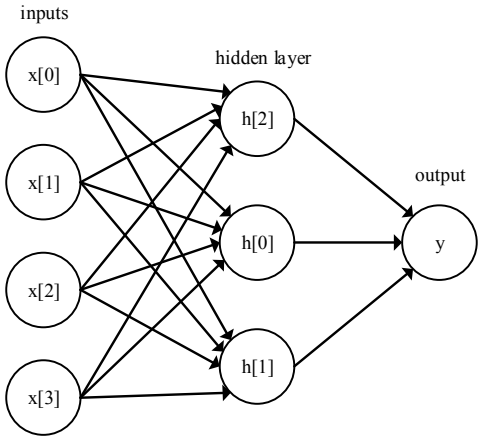


图 4.64 单隐层的多层感知机图示

这个模型需要学习更多的系数(也叫作权重)：在每个输入与每个隐单元(隐单元组成了隐层)之间有一个系数，在每个隐单元与输出之间也有一个系数。

从数学的角度看，计算一系列加权求和与只计算一个加权求和是完全相同的，因此，为了让这个模型真正比线性模型更为强大，我们还需要一个技巧。在计算完每个隐单元的加权求和之后，对结果再应用一个非线性函数——通常是校正非线性(rectifying nonlinearity，也叫校正线性单元或 relu)或正切双曲线(tangens hyperbolicus，tanh)。然后将这个函数的结果用于加权求和，计算得到输出。这两个函数的可视化效果见图 4.65。relu 截断小于 0 的值，而 tanh 在输入值较小时接近-1，在输入值较大时接近+1。有了这两种非线性函数，神经网络可以学习比线性模型复杂得多的函数。

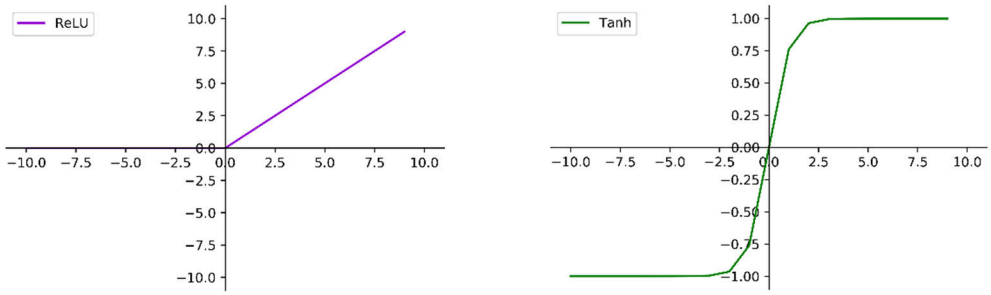


图 4.65 双曲正切激活函数与校正线性激活函数

(2) 神经网络调参

我们将 MLPClassifier 应用到 two_moons 数据集中，以此研究 MLP 的工作原理。结果如图 4.66 所示：

```
from sklearn.neural_network import MLPClassifier
from sklearn.datasets import make_moons

X,y = make_moons(n_samples=100,noise=0.25,random_state=3)
X_train,X_test,y_train,y_test = train_test_split(
            X,y,stratify=y,random_state=42)
mlp = MLPClassifier(solver='lbfgs',random_state=0).fit(X_train,y_train)
mglearn.plots.plot_2d_separator(mlp,X_train,fill=True,alpha=.3)
mglearn.discrete_scatter(X_train[:,0],X_train[:,1],y_train)
plt.xlabel("Feature 0")
plt.ylabel("Feature 1")

'''
```

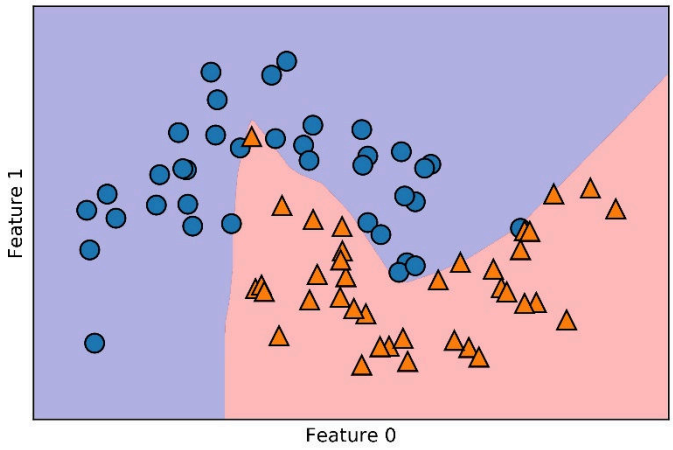


图 4.66 包含 100 个隐单元的神经网络在 two_moons 数据集上学到的决策边界

如你所见，神经网络学到的决策边界完全是非线性的，但相对平滑。我们用到了 solver='lbfs'，这一点稍后会讲到。

默认情况下，MLP 使用 100 个隐结点，这对于这个小型数据集来说已经相当多了。我们可以减少其数量(从而降低了模型复杂度)，但仍然得到很好的结果(图 4.67)：

```
mlp = MLPClassifier(solver='lbfgs',random_state=0,hidden_layer_sizes=[10])
mlp.fit(X_train,y_train)
mglearn.plots.plot_2d_separator(mlp,X_train,fill=True,alpha=.3)
mglearn.discrete_scatter(X_train[:,0],X_train[:,1],y_train)
plt.xlabel("Feature 0")
plt.ylabel("Feature 1")

,,,
```

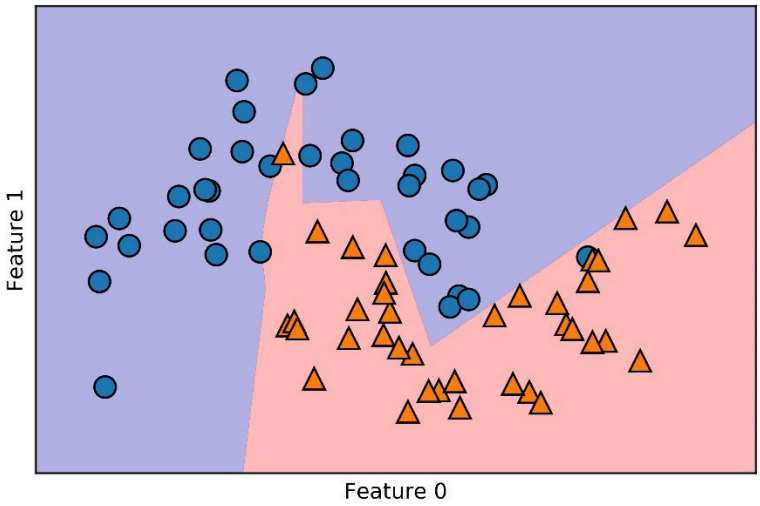


图 4.67 包含 10 个隐单元的神经网络在 two_moons 数据集上学到的决策边界

只有 10 个隐单元时，决策边界看起来更加参差不齐。默认的非线性是 relu，如图 4.65 所示。如果使用单隐层，那么决策函数将由 10 个直线段组成。如果想得到更加平滑的决策边界，可以添加更多的隐单元(见图 4.66)、添加第二个隐层(见图 4.68)或者使用 tanh 非线性(见图 4.69)。

```
#使用 2 个隐层，每个包含 10 个单元
mlp = MLPClassifier(solver='lbfgs',random_state=0,hidden_layer_sizes=[10,10])
mlp.fit(X_train,y_train)
mglearn.plots.plot_2d_separator(mlp,X_train,fill=True,alpha=.3)
mglearn.discrete_scatter(X_train[:,0],X_train[:,1],y_train)
plt.xlabel("Feature 0")
plt.ylabel("Feature 1")
#使用 2 个隐层，每个包含 10 个单元，这次使用 tanh 非线性
mlp = MLPClassifier(solver='lbfgs',activation='tanh',random_state=0,hidden_layer_sizes=[10,10])
mlp.fit(X_train,y_train)
mglearn.plots.plot_2d_separator(mlp,X_train,fill=True,alpha=.3)
mglearn.discrete_scatter(X_train[:,0],X_train[:,1],y_train)
plt.xlabel("Feature 0")
plt.ylabel("Feature 1")
```

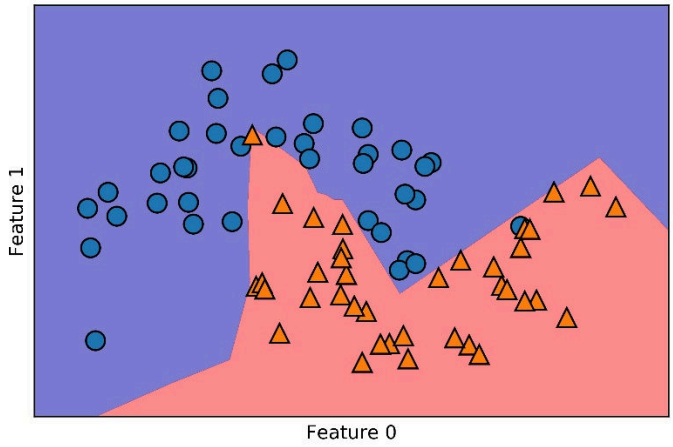


图 4.68 包含 2 个隐层、每个隐层包含 10 个隐单元的神经网络学到的决策边界（激活函数为 relu）

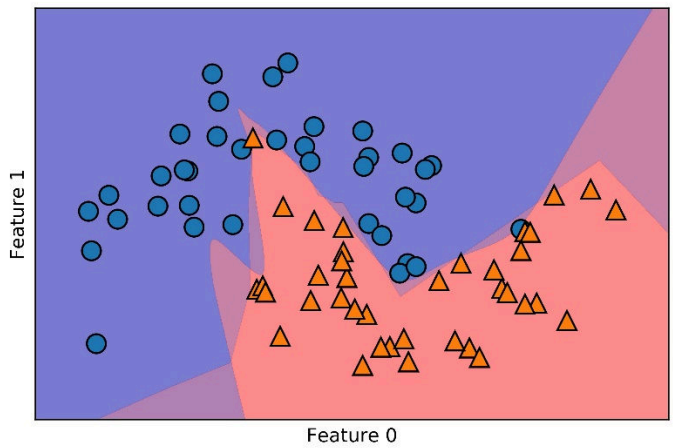


图 4.69 包含 2 个隐层、每个隐层包含 10 个隐单元的神经网络学到的决策边界（激活函数为 tanh）

最后，我们还可以利用 L2 惩罚使权重趋向于 0，从而控制神经网络的复杂度，正如我们在岭回归和线性分类器中所做的那样。MLPClassifier 中调节 L2 惩罚的参数是 alpha(与线性回归模型中的相同)，它的默认值很小(弱正则化)。图 4.70 显示了不同 alpha 值对 two_moons 数据集的影响，用的是 2 个隐层的神经网络，每层包含 10 个或 100 个单元：

```
fig,axes = plt.subplots(2,4,figsize=(20,8))
for axx,n_hidden_nodes in zip(axes,[10,100]):
    for ax,alpha in zip(axx,[0.0001,0.01,0.1,1]):
        mlp = MLPClassifier(solver='lbfgs',random_state=0,hidden_layer_sizes=
                [n_hidden_nodes,n_hidden_nodes],alpha=alpha)
        mlp.fit(X_train,y_train)
        mglearn.plots.plot_2d_separator(mlp,X_train,fill=True,alpha=.3,ax=ax)
        mglearn.discrete_scatter(X_train[:,0],X_train[:,1],y_train,ax=ax)
        ax.set_title("n_hidden=[{},{}]\nalpha={:.4f}".format(n_hidden_nodes,
                                                    n_hidden_nodes,alpha))

'''
```

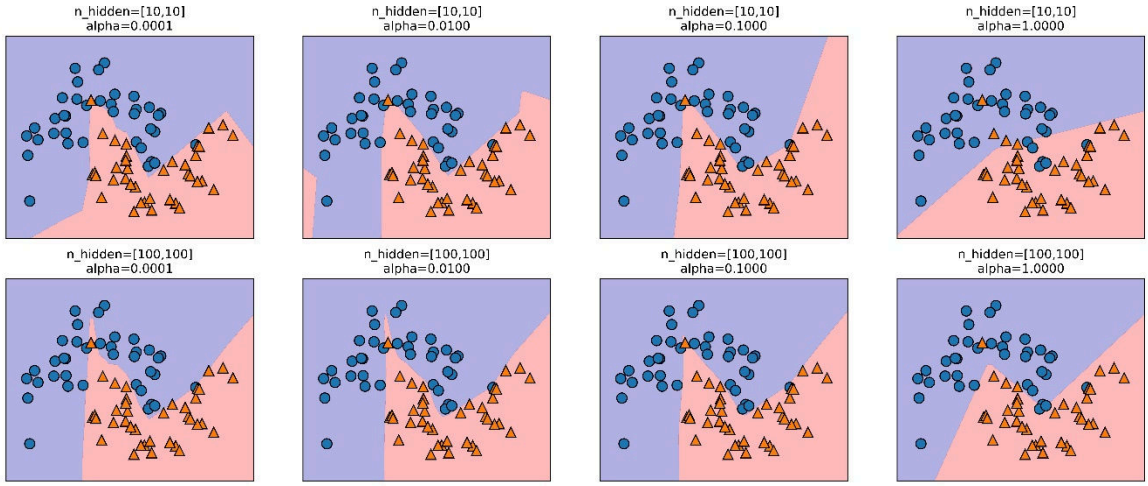


图 4.70 不同隐单元个数与 alpha 参数的不同设定下的决策函数

现在你可能已经认识到了，控制神经网络复杂度的方法有很多种：隐层的个数、每个隐层中的单元个数与正则化(alpha)。实际上还有更多，但这里不再过多介绍。

神经网络的一个重要性质是，在开始学习之前其权重是随机设置的，这种随机初始化会影响学到的模型。也就是说，即使使用完全相同的参数，如果随机种子不同的话，我们也可能得到非常不一样的模型。如果网络很大，并且复杂度选择合理的话，那么这应该不会对精度有太大影响，但应该记住这一点(特别是对于较小的网络)。图 4.71 显示了几个模型的图像，所有模型都使用相同的参数设置进行学习：

```
fig,axes = plt.subplots(2,4,figsize=(20,8))
for i,ax in enumerate(axes.ravel()):
    mlp = MLPClassifier(solver='lbfgs',random_state=i,
                        hidden_layer_sizes=[100,100])
    mlp.fit(X_train,y_train)
    mglearn.plots.plot_2d_separator(mlp,X_train,fill=True,alpha=.3,ax=ax)
    mglearn.discrete_scatter(X_train[:,0],X_train[:,1],y_train,ax=ax)

'''
```

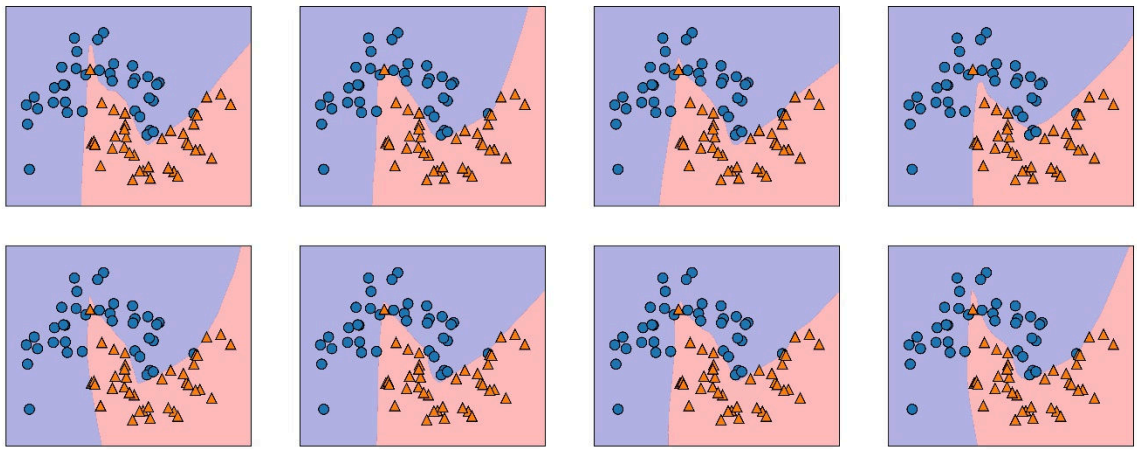


图 4.71 相同参数但不同随机初始化的情况下学到的决策函数

为了在现实世界的数据上进一步理解神经网络，我们将 MLPClassifler 应用在乳腺癌数据集上。首先使用默认参数：

```
print("Cancer data per-feature maxima:\n{}".format(cancer.data.max(axis=0)))

'''
运行以上程序可以得到输出:
Cancer data per-feature maxima:
[2.811e+01 3.928e+01 1.885e+02 2.501e+03 1.634e-01 3.454e-01 4.268e-01
 2.012e-01 3.040e-01 9.744e-02 2.873e+00 4.885e+00 2.198e+01 5.422e+02
 3.113e-02 1.354e-01 3.960e-01 5.279e-02 7.895e-02 2.984e-02 3.604e+01
```

```
X_train,X_test,y_train,y_test = train_test_split(
        cancer.data,cancer.target,random_state=0)
mlp = MLPClassifier(random_state=42)
mlp.fit(X_train,y_train)
print("Accuracy on training set:{:.2f}".format(mlp.score(X_train,y_train)))
print("Accuracy on test set:{:.2f}".format(mlp.score(X_test,y_test)))

'''
运行以上程序可以得到输出:
Accuracy on training set:0.94
Accuracy on test set:0.92
```

MLP 的精度相当好，但没有其他模型好。与较早的 SVC 例子相同，原因可能在于数据的缩放。神经网络也要求所有输入特征的变化范围相似，最理想的情况是均值为 0、方差为 1。我们必须对数据进行缩放以满足这些要求。同样，我们这里将人工完成。

```
#计算训练集中每个特征的平均值
mean_on_train = X_train.mean(axis=0)
#计算训练集中每个特征的标准差
std_on_train = X_train.std(axis=0)
#减去平均值，然后乘以标准差的倒数
#如此运算之后，mean=0,std=1
X_train_scaled = (X_train-mean_on_train)/std_on_train
#对测试集做相同的变换（使用训练集的平均差和标准差）
X_test_scaled = (X_test-mean_on_train)/std_on_train

mlp = MLPClassifier(random_state=0)
mlp.fit(X_train_scaled,y_train)
print("Accuracy on training set:{:.3f}".format(mlp.score(X_train_scaled,y_train)))
print("Accuracy on test set:{:.3f}".format(mlp.score(X_test_scaled,y_test)))

'''
运行以上程序可以得到输出:
Accuracy on training set:0.991
Accuracy on test set:0.965
```

缩放之后的结果要好得多，而且也相当有竞争力。不过模型给出了一个警告，告诉我们已经达到最大迭代次数。这是用于学习模型的 adam 算法的一部分，告诉我们应该增加迭代次数：

```
mlp = MLPClassifier(max_iter=1000,random_state=0)
mlp.fit(X_train_scaled,y_train)
print("Accuracy on training set:{:.3f}".format(mlp.score(X_train_scaled,y_train)))
print("Accuracy on test set:{:.3f}".format(mlp.score(X_test_scaled,y_test)))

'''
运行以上程序可以得到输出:
Accuracy on training set:1.000
Accuracy on test set:0.972
```

增加迭代次数仅提高了训练集性能，但没有提高泛化性能。不过模型的表现相当不错。由于训练性能和测试性能之间仍有一些差距，所以我们可以尝试降低模型复杂度来得到更好的泛化性能。这里我们选择增大 alpha 参数(变化范围相当大，从 0.0001 到 1)，以此向权重添加更强的正则化：

```
mlp = MLPClassifier(max_iter=1000,alpha=1,random_state=0)
mlp.fit(X_train_scaled,y_train)
print("Accuracy on training set:{:.3f}".format(mlp.score(X_train_scaled,y_train)))
print("Accuracy on test set:{:.3f}".format(mlp.score(X_test_scaled,y_test)))

'''
运行以上程序可以得到输出:
Accuracy on training set:0.988
Accuracy on test set:0.972
```

这得到了与我们目前最好的模型相同的性能。

虽然可以分析神经网络学到了什么，但这通常比分析线性模型或基于树的模型更为复杂。要想观察模型学到了什么，一种方法是查看模型的权重。你可以在 scikit-learn 示例库中查看这样的一个示例(http:/ scikit-learn.org/stable/auto_examples/neural_networks/plot_mnist_filters.html)。对于乳腺癌数据集，这可能有点难以理解。下面这张图(图 4.72)显示了连接输入和第一个隐层之间的权重。图中的行对应 30 个输入特征，列对应 100 个隐单元。浅色代表较大的正值，而深色代表负值。

```
plt.figure(figsize=(20,5))
plt.imshow(mlp.coefs_[0],interpolation='none',cmap='viridis')
plt.yticks(range(30),cancer.feature_names)
plt.xlabel("Columns in weight matrix")
plt.ylabel("Input feature")
plt.colorbar()

'''
```

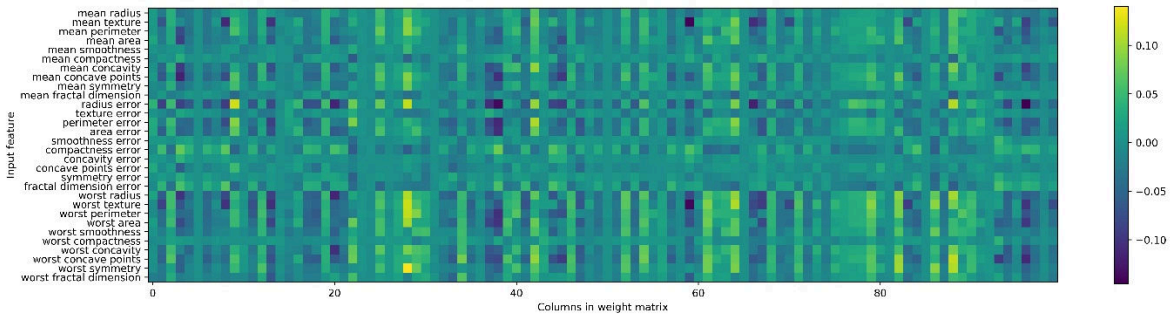


图 4.72 神经网络在乳腺癌数据集上学到的第一个隐层权重的热图

我们可以推断，如果某个特征对所有隐单元的权重都很小，那么这个特征对模型来说就“不太重要”。可以看到，与其他特征相比，“mean smoothness”“mean compactness”以及“smoothness error”和“ fractal dimension error”之间的特征的权重都相对较小。这可能说明这些特征不太重要，也可能是我们没有用神经网络可以使用的方式来表示这些特征。我们还可以将连接隐层和输出层的权重可视化，但它们更加难以解释。

虽然 MLPClassifier 和 MLPRegressor 为最常见的神经网络架构提供了易于使用的接口，但它们只包含神经网络潜在应用的一部分。如果你有兴趣使用更灵活或更大的模型，我们建议你看一下除了 scikit-learn 之外的很棒的深度学习库。对于 Python 用户来说，最为完善的是 keras、1asagna 和 tensor-flow。lasagna 是基于 theano 库构建的，而 keras 既可以用 tensor-flow 也可以用 theano。这些库提供了更为灵活的接口，可以用来构建神经网络并跟踪深度学习研究的快速发展。所有流行的深度学习库也都允许使用高性能的图形处理单元(GPU)，而 scikit-learn 不支持 GPU。使用 GPU 可以将计算速度加快 10 到 100 倍，GPU 对于将深度学习方法应用到大型数据集上至关重要。

(3) 优点、缺点和参数

在机器学习的许多应用中，神经网络再次成为最先进的模型。它的主要优点之一是能够获取大量数据中包含的信息。并构建无比复杂的模型。给定足够的计算时间和数据，并且仔细调节参数，神经网络通常可以打败其他机器学习算法(无论是分类任务还是回归任务)。

这就引出了下面要说的缺点。神经网络——特别是功能强大的大型神经网络一一通常需要很长的训练时间。它还需要仔细地预处理数据，正如我们这里所看到的。与 SVM 类似，神经网络在“均匀”数据上的性能最好，其中“均匀”是指所有特征都具有相似的含义。如果数据包含不同种类的特征，那么基于树的模型可能表现得更好。神经网络调参本身也是一门艺术。调节神经网络模型和训练模型的方法有很多种，我们只是蜻蜓点水地尝试了几种而已。

估计神经网络的复杂度。最重要的参数是层数和每层的隐单元个数。你应该首先设置 1 个或 2 个隐层，然后可以逐步增加。每个隐层的结点个数通常与输入特征个数接近，但在几千个结点时很少会多于特征个数。

在考虑神经网络的模型复杂度时，一个有用的度量是学到的权重(或系数)的个数。如果你有一个包含 100 个特征的二分类数据集，模型有 100 个隐单元，那么输入层和第一个隐层之间就有 100*100=10000 个权重。在隐层和输出层之间还有 100*1=100 个权重，总共约 10100 个权重。如果添加含有 100 个隐单元的第二个隐层，那么在第一个隐层和第二个隐层之间又有 100*100=10000 个权重，总数变为约 20100 个权重。如果你使用包含 1000 个隐单元的单隐层，那么在输入层和隐层之间需要学习 100*1000=100000 个权重隐层到输出层之间需要学习 1000*1=1000 个权重，总共 101000 个权重。如果再添加第二个隐层，就会增加 1000*1000=1000000 个权重，总数变为巨大的 1101000 个权重，这比含有 2 个隐层、每层 100 个单元的模型要大 50 倍。

神经网络调参的常用方法是，首先创建一个大到足以过拟合的网络，确保这个网络可以对任务进行学

习。知道训练数据可以被学习之后，要么缩小网络，要么增大 alpha 来增强正则化，这可以提高泛化性能。

在我们的实验中，主要关注模型的定义：层数、每层的结点个数、正则化和非线性。这些内容定义了我们想要学习的模型。还有一个问题是，如何学习模型或用来学习参数的算法，这一点由 solver 参数设定。solver 有两个好用的选项。默认选项是“adam”，在大多数情况下效果都很好，但对数据的缩放相当敏感(因此，始终将数据缩放为均值为 0、方差为 1 是很重要的)。另一个选项是“lbfgs”，其鲁棒性相当好，但在大型模型或大型数据集上的时间会比较长。还有更高级的“sgd”选项，许多深度学习研究人员都会用到。“sgd”选项还有许多其他参数需要调节，以便获得最佳结果。你可以在用户指南中找到所有这些参数及其定义。当你开始使用 MLP 时，我们建议使用“adam”和“lbfgs”。

4.4.7 随机森林

集成(ensemble)是合并多个机器学习模型来构建更强大模型的方法。在机器学习文献中有许多模型都属于这一类，但已证明有两种集成模型对大量分类和回归的数据集都是有效的，二者都以决策树为基础，分别是随机森林(random forest)和梯度提升决策树(gradient boosted decision tree)。

1. 随机森林

我们刚刚说过，决策树的一个主要缺点在于经常对训练数据过拟合。随机森林是解决这个问题的一种方法。随机森林本质上是许多决策树的集合，其中每棵树都和其他树略有不同。随机森林背后的思想是，每棵树的预测可能都相对较好，但可能对部分数据过拟合。如果构造很多树，并且每棵树的预测都很好，但都以不同的方式过拟合，那么我们可以对这些树的结果取平均值来降低过拟合。既能减少过拟合又能保持树的预测能力，这可以在数学上严格证明。

为了实现这一策略，我们需要构造许多决策树。每棵树都应该对目标值做出可以接受的预测，还应该与其他树不同。随机森林的名字来自于将随机性添加到树的构造过程中，以确保每棵树都各不相同。随机森林中树的随机化方法有两种：一种是通过选择用于构造树的数据点，另一种是通过选择每次划分测试的特征。我们来更深入地研究这一过程。想要构造一个随机森林模型， 你需要确定用于构造的树的个数(RandomForestRegressor 或 RandomForestClassifier 的 n_estimators 参数)。比如我们想要构造 10 棵树。这些树在构造时彼此完全独立，算法对每棵树进行不同的随机选择，以确保树和树之间是有区别的。想要构造一棵树，首先要对数据进行自助采样(bootstrap sample)。也就是说，从 n_samples 个数据点中有放回地(即同一样本可以被多次抽取)重复随机抽取一个样本，共抽取 n_samples 次。这样会创建一个与原数据集大小相同的数据集，但有些数据点会缺失（大约三分之一），有些会重复。

举例说明，比如我们想要创建列表['a', 'b', 'c', 'd'] 的自助采样。一种可能的自主采样是['b', 'd', 'd', 'c']，另一种可能的采样为['d', 'a', 'd', 'a']。接下来，基于这个新创建的数据集来构造决策树。但是，要对我们在介绍决策树时描述的算法稍作修改。在每个结点处，算法随机选择特征的一个子集，并对其中一个特征寻找最佳测试，而不是对每个结点都寻找最佳测试。选择的特征个数由max_features 参数来控制。每个结点中特征子集的选择是相互独立的，这样树的每个结点可以使用特征的不同子集来做出决策。由于使用了自助采样，随机森林中构造每棵决策树的数据集都是略有不同的。由于每个结点的特征选择，每棵树中的每次划分都是基于特征的不同子集。这两种方法共同保证随机森林中所有树都不相同。

在这个过程中的一个关键参数是 max_features。如果我们设置 max_features 等于 n_features，那么每次划分都要考虑数据集的所有特征，在特征选择的过程中没有添加随机性（不过自助采样依然存在随机性）。如果设置 max_features 等于 1，那么在划分时将无法选择对哪个特征进行测试，只能对随机选择的某个特征搜索不同的阈值。因此，如果 max_features 较大，那么随机森林中的树将会十分相似，利用最独特的特征可以轻松拟合数据。如果 max_features 较小，那么随机森林中的树将会差异很大，为了很好地拟合数据，每棵树的深度都要很大。

想要利用随机森林进行预测，算法首先对森林中的每棵树进行预测。对于回归问题，我们可以对这些结果取平均值作为最终预测。对于分类问题，则用到了“软投票”（soft voting）策略。也就是说，每个算法做出“软”预测，给出每个可能的输出标签的概率。对所有树的预测概率取平均值，然后将概率最大的类别作为预测结果。下面将由 5 棵树组成的随机森林应用到前面研究过的 two_moons 数据集上：

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.datasets import make_moons
X, y = make_moons(n_samples=100, noise=0.25, random_state=3)
X_train, X_test, y_train, y_test = train_test_split(X, y, stratify=y,
random_state=42)
forest = RandomForestClassifier(n_estimators=5, random_state=2)
forest.fit(X_train, y_train)

‘’’
```

作为随机森林的一部分，树被保存在 estimator_属性中。我们将每棵树学到的决策边界可视化，也将它们的总预测(即整个森林做出的预测)可视化(图 4.73)：

```
fig, axes = plt.subplots(2, 3, figsize=(20, 10))
for i, (ax, tree) in enumerate(zip(axes.ravel(), forest.estimators_)):
    ax.set_title("Tree {}".format(i))
    mglearn.plots.plot_tree_partition(X_train, y_train, tree, ax=ax)
    mglearn.plots.plot_2d_separator(forest, X_train, fill=True, ax=axes[-1, -1],
    alpha=.4)
    axes[-1, -1].set_title("Random Forest")
    mglearn.discrete_scatter(X_train[:, 0], X_train[:, 1], y_train)

‘’’
```

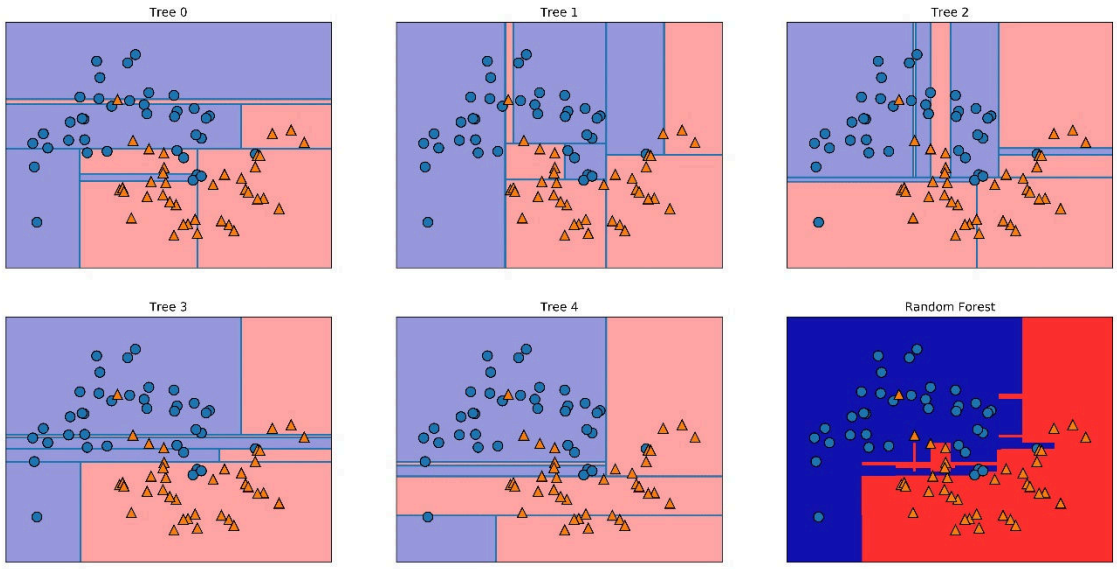


图4.73　5棵随机化的决策树找到的决策边界，以及将它们的预测概率取平均后得到的决策边界

你可以清楚地看到，这 5 棵树学到的决策边界大不相同。每棵树都犯了一些错误，因为这里画出的一些训练点实际上并没有包含在这些树的训练集中，原因在于自助采样。随机森林比单独每一棵树的过拟合都要小，给出的决策边界也更符合直觉。在任何实际应用中，我们会用到更多棵树（通常是几百或上千），从而得到更平滑的边界。再举一个例子，我们将包含 100 棵树的随机森林应用在乳腺癌数据集上：

```
X_train, X_test, y_train, y_test = train_test_split(
cancer.data, cancer.target, random_state=0)
forest = RandomForestClassifier(n_estimators=100, random_state=0)
forest.fit(X_train, y_train)
print("Accuracy on training set: {:.3f}".format(forest.score(X_train, y_train)))
print("Accuracy on test set: {:.3f}".format(forest.score(X_test, y_test)))

'''
运行以上程序可以得到输出:
Accuracy on training set: 1.000
Accuracy on test set: 0.972
```

在没有调节任何参数的情况下，随机森林的精度为 97%，比线性模型或单棵决策树都要好。我们可以调节 max_features 参数，或者像单棵决策树那样进行预剪枝。但是，随机森林的默认参数通常就已经可以给出很好的结果。与决策树类似，随机森林也可以给出特征重要性，计算方法是将森林中所有树的特征重要性求和并取平均。一般来说，随机森林给出的特征重要性要比单棵树给出的更为可靠。参见图 4.74。

```
plot_feature_importances_cancer(forest)
```

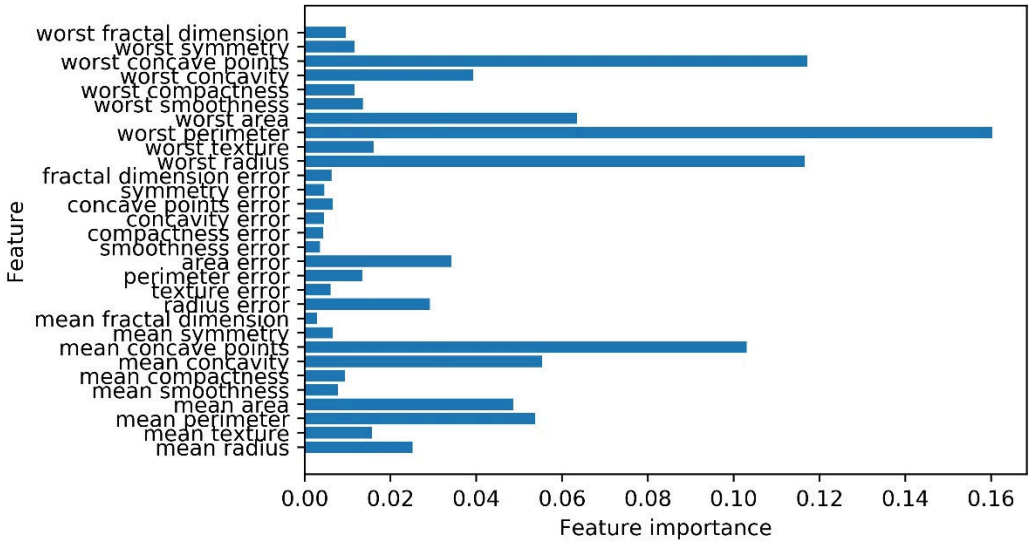


图4.74 拟合乳腺癌数据集得到的随机森林的特征重要性

如图所见，与单棵树相比，随机森林中有更多特征的重要性不为零。与单棵决策树类似，随机森林也给了“worst radius”（最大半径）特征很大的重要性，但从总体来看，它实际上却选择“worst perimeter”（最大周长）作为信息量最大的特征。由于构造随机森林过程中的随机性，算法需要考虑多种可能的解释，结果就是随机森林比单棵树更能从总体把握数据的特征。

用于回归和分类的随机森林是目前应用最广泛的机器学习方法之一。这种方法非常强大，通常不需要反复调节参数就可以给出很好的结果，也不需要对数据进行缩放。从本质上看，随机森林拥有决策树的所有优点，同时弥补了决策树的一些缺陷。仍然使用决策树的一个原因是需要决策过程的紧凑表示。基本上不可能对几十棵甚至上百棵树做出详细解释，随机森林中树的深度往往比决策树还要大（因为用到了特征子集）。因此，如果你需要以可视化的方式向非专家总结预测过程，那么选择单棵决策树可能更好。

虽然在大型数据集上构建随机森林可能比较费时间，但在一台计算机的多个 CPU 内核上并行计算也很容易。如果你用的是多核处理器（几乎所有的现代化计算机都是），你可以用 n_jobs 参数来调节使用的内核个数。使用更多的 CPU 内核，可以让速度线性增加（使用 2 个内核，随机森林的训练速度会加倍），但设置 n_jobs 大于内核个数是没有用的。你可以设置 n_jobs=-1 来使用计算机的所有内核。你应该记住，随机森林本质上是随机的，设置不同的随机状态（或者不设置 random_state 参数）可以彻底改变构建的模型。森林中的树越多，它对随机状态选择的鲁棒性就越好。

如果你希望结果可以重现，固定 random_state 是很重要的。对于维度非常高的稀疏数据（比如文本数据），随机森林的表现往往不是很好。对于这种数据，使用线性模型可能更合适。即使是非常大的数据集，随机森林的表现通常也很好，训练过程很容易并行在功能强大的计算机的多个 CPU 内核上。不过，随机森林需要更大的内存，训练和预测的速度也比线性模型要慢。对一个应用来说，如果时间和内存很重要的话，那么换用线性模型可能更为明智。需要调节的重要参数有 n_estimators 和 max_features，可能还包括预剪枝选项（如 max_depth）。n_estimators 总是越大越好。对更多的树取平均可以降低过拟合，从而得到鲁棒性更好的集成。不过收益是递减的，而且树越多需要的内存也越多，训练时间也越长。常用的经验法则就是“在你的时间/ 内存允许的情况下尽量多”。

前面说过，max_features 决定每棵树的随机性大小，较小的 max_features 可以降低过拟合。一般来说，好的经验就是使用默认值：对于分类，默认值是 max_features=sqrt(n_features)； 对于回归， 默认值是 max_features=n_features。增大 max_features 或 max_leaf_nodes 有时也可以提高性能。它还可以大大降低用于训练和预测的时间和空间要求。

2. 梯度提升树

梯度提升回归树是另一种集成方法，通过合并多个决策树来构建一个更为强大的模型。虽然名字中含有“回归”，但这个模型既可以用于回归也可以用于分类。与随机森林方法不同，梯度提升采用连续的方式构造树，每棵树都试图纠正前一棵树的错误。默认情况下，梯度提升回归树中没有随机化，而是用到了强预剪枝。梯度提升树通常使用深度很小(1 到 5 之间)的树，这样模型占用的内存更少，预测速度也更快。梯度提升背后的主要思想是合并许多简单的模型（在这个语境中叫作弱学习器），比如深度较小的树。每棵树只能对部分数据做出好的预测，因此，添加的树越来越多，可以不断迭代提高性能。

梯度提升树经常是机器学习竞赛的优胜者，并且广泛应用于业界。与随机森林相比，它通常对参数设置更为敏感，但如果参数设置正确的话，模型精度更高。除了预剪枝与集成中树的数量之外，梯度提升的另一个重要参数是 learning_rate（学习率），用于控制每棵树纠正前一棵树的错误的强度。较高的学习率意味着每棵树都可以做出较强的修正，这样模型更为复杂。通过增大 n_estimators 来向集成中添加更多树，也可以增加模型复杂度，因为模型有更多机会纠正训练集上的错误。

下面是在乳腺癌数据集上应用 GradientBoostingClassifier 的示例。默认使用 100 棵树，最大深度是 3，学习率为 0.1：

```
from sklearn.ensemble import GradientBoostingClassifier
X_train, X_test, y_train, y_test = train_test_split(
cancer.data, cancer.target, random_state=0)
gbrt = GradientBoostingClassifier(random_state=0)
gbrt.fit(X_train, y_train)
print("Accuracy on training set: {:.3f}".format(gbrt.score(X_train, y_train)))
print("Accuracy on test set: {:.3f}".format(gbrt.score(X_test, y_test)))
'''
运行以上程序可以得到输出:
Accuracy on training set: 1.000
Accuracy on test set: 0.958
```

由于训练集精度达到 100%，所以很可能存在过拟合。为了降低过拟合，我们可以限制最大深度来加强预剪枝，也可以降低学习率：

```
gbrt = GradientBoostingClassifier(random_state=0, max_depth=1)
gbrt.fit(X_train, y_train)
print("Accuracy on training set: {:.3f}".format(gbrt.score(X_train, y_train)))
print("Accuracy on test set: {:.3f}".format(gbrt.score(X_test, y_test)))

'''
运行以上程序可以得到输出:
Accuracy on training set: 0.991
Accuracy on test set: 0.972
```

```
gbrt = GradientBoostingClassifier(random_state=0, learning_rate=0.01)
gbrt.fit(X_train, y_train)
print("Accuracy on training set: {:.3f}".format(gbrt.score(X_train, y_train)))
print("Accuracy on test set: {:.3f}".format(gbrt.score(X_test, y_test)))

'''
运行以上程序可以得到输出:
Accuracy on training set: 0.988
Accuracy on test set: 0.965
```

降低模型复杂度的两种方法都降低了训练集精度，这和预期相同。在这个例子中，减小树的最大深度显著提升了模型性能，而降低学习率仅稍稍提高了泛化性能。对于其他基于决策树的模型，我们也可以将特征重要性可视化，以便更好地理解模型(图 4.75)。由于我们用到了 100 棵树，所以即使所有树的深度都是 1，查看所有树也是不现实的：

```
gbrt = GradientBoostingClassifier(random_state=0, max_depth=1)
gbrt.fit(X_train, y_train)
plot_feature_importances_cancer(gbrt)
```

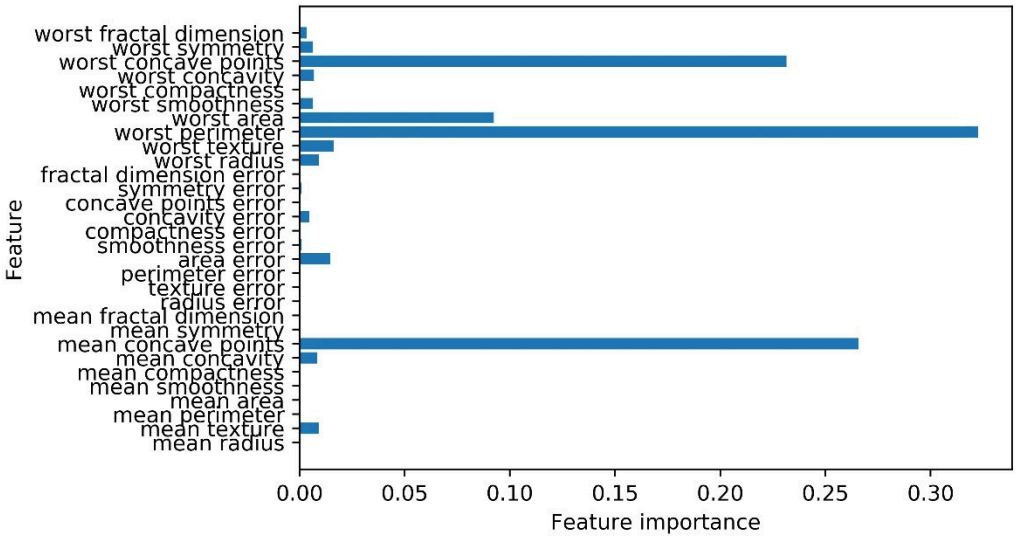


图4.75 用于拟合乳腺癌数据集的梯度提升分类器给出的特征重要性

可以看到，梯度提升树的特征重要性与随机森林的特征重要性有些类似，不过梯度提升完全忽略了某些特征。由于梯度提升和随机森林两种方法在类似的数据上表现得都很好，因此一种常用的方法就是先尝试随机森林，它的鲁棒性很好。如果随机森林效果很好，但预测时间太长，或者机器学习模型精度小数点后第二位的提高也很重要，那么切换成梯度提升通常会有用。

如果你想要将梯度提升应用在大规模问题上，可以研究一下 xgboost 包及其 Python 接口，在写作本书时，这个库在许多数据集上的速度都比 scikit-learn 对梯度提升的实现要快（有时调参也更简单）。

梯度提升决策树是监督学习中最强大也最常用的模型之一。其主要缺点是需要仔细调参，而且训练时间可能会比较长。与其他基于树的模型类似，这一算法不需要对数据进行缩放就可以表现得很好，而且也适用于二元特征与连续特征同时存在的数据集。与其他基于树的模型相同，它也通常不适用于高维稀疏数据。

梯度提升树模型的主要参数包括树的数量 n_estimators 和学习率 learning_rate，后者用于控制每棵树对前一棵树的错误的纠正强度。这两个参数高度相关，因为 learning_rate 越低，就需要更多的树来构建具有相似复杂度的模型。随机森林的 n_estimators 值总是越大越好，但梯度提升不同，增大 n_estimators 会导致模型更加复杂，进而可能导致过拟合。通常的做法是根据时间和内存的预算选择合适的 n_estimators，然后对不同的 learning_rate 进行遍历。另一个重要参数是 max_depth（或 max_leaf_nodes），用于降低每棵树的复杂度。梯度提升模型的 max_depth 通常都设置得很小，一般不超过 5。

4.4.8 K-Means 聚类、层次聚类

层次聚类（凝聚聚类）指的是许多基于相同原则构建的聚类算法，这一原则是：算法首先声明每个点是自己的簇，然后合并两个最相似的簇，直到满足某种停止准则为止。scikit-learn 中实现的停止准则是簇的个数，因此相似的簇被合并，直到仅剩下指定个数的簇。还有一些链接(linkage)准则，规定如何度量“最相似的簇”。这种度量总是定义在两个现有的簇之间。

scikit-learn 中实现了以下三种选项。

ward：默认选项。ward 挑选两个簇来合并，使得所有簇中的方差增加最小。这通常会得到大小差不多相等的簇。

average：average 链接将簇中所有点之间平均距离最小的两个簇合并。

complete：complete 链接(也称为最大链接)将簇中点之间最大距离最小的两个簇合并。

ward 适用于大多数数据集，在我们的例子中将使用它。如果簇中的成员个数非常不同（比如其中一个比其他所有都大得多），那么 average 或 complete 可能效果更好图 4.76 给出了在一个二维数据集上的层次聚类过程，要寻找三个簇。

```
import mglearn
mglearn.plots.plot_agglomerative_algorithm()

‘’’
```

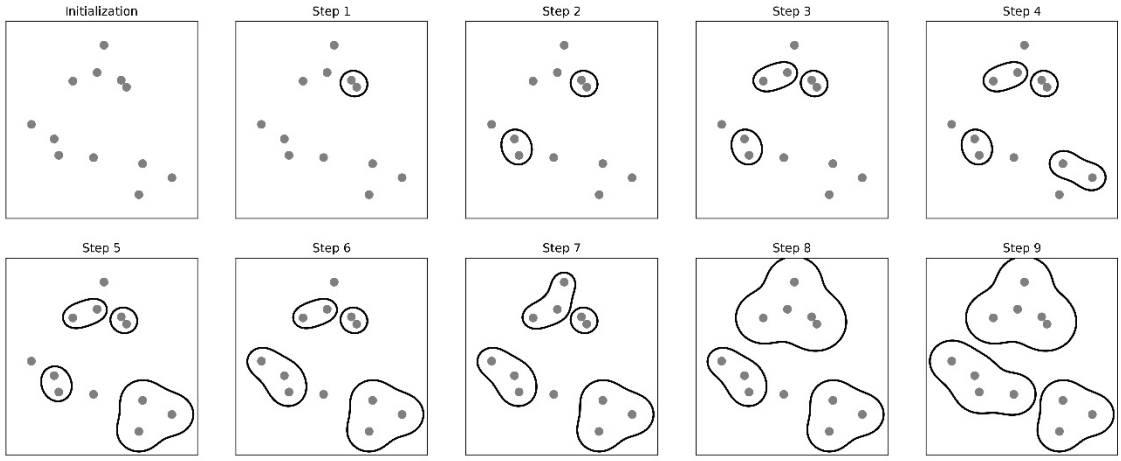


图 4.76 层次聚类用迭代的方式合并两个最近的簇

最开始，每个点自成一簇。然后在每一个步骤中，相距最近的两个簇被合并。在前四个步骤中，选出两个单点簇并将其合并成两点簇。在步骤 5(Step5)中，其中一个两点簇被扩展到三个点，以此类推。在步骤 9(Step9)中，只剩下 3 个。由于我们指定寻找 3 个簇，因此算法结束。

我们来看一下层次聚类对我们这里使用的简单三簇数据的效果如何。由于算法的工作原理，凝聚算法不能对新数据点做出预测。因此 AgglomerativeClustering 没有 predict 方法。为了构造模型并得到训练集上簇的成员关系，可以改用 fit_predict 方法。结果如图 4.77 所示。

```
from sklearn.cluster import AgglomerativeClustering

X,y = make_blobs(random_state=1)
agg = AgglomerativeClustering(n_clusters=3)
assignment = agg.fit_predict(X)
mglearn.discrete_scatter(X[:,0],X[:,1],assignment)
plt.xlabel("feature 0")
plt.ylabel("feature 1")

,,,
```

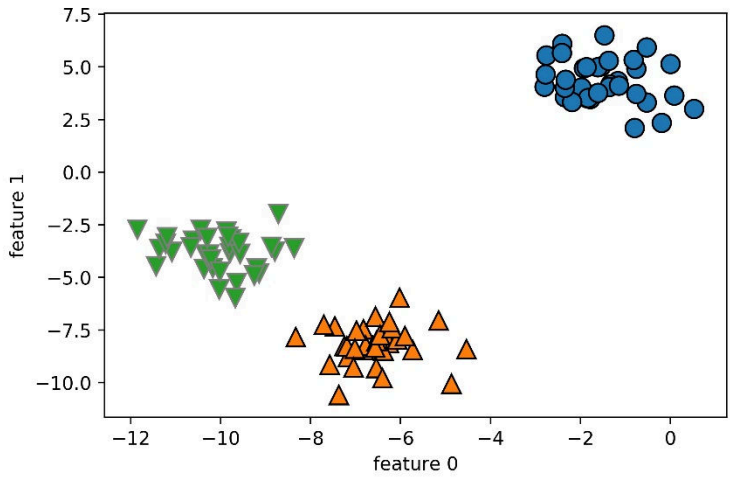


图 4.77 使用 3 个簇的层次聚类的簇分配

正如所料，算法完美地完成了聚类。虽然凝聚聚类的 scikit-learn 实现需要你指定希望算法找的的簇的个数，但凝聚聚类方法为选择正确的个数提供而一些帮助，我们将在下面讨论。

聚类过程迭代进行，每个点都从一个单点簇变为属于最终的某个簇。每个中间步骤都提供了数据的一种聚类（簇的个数也不相同）。有时候，同时查看所有可能的聚类是有帮助的。下一个例子（图 4.3）叠加显示了图 4.76 中所有可能的聚类，有助于深入了解每个簇如何分解为较小的簇：

```
mglearn.plots.plot_agglomerative()
,,,
```

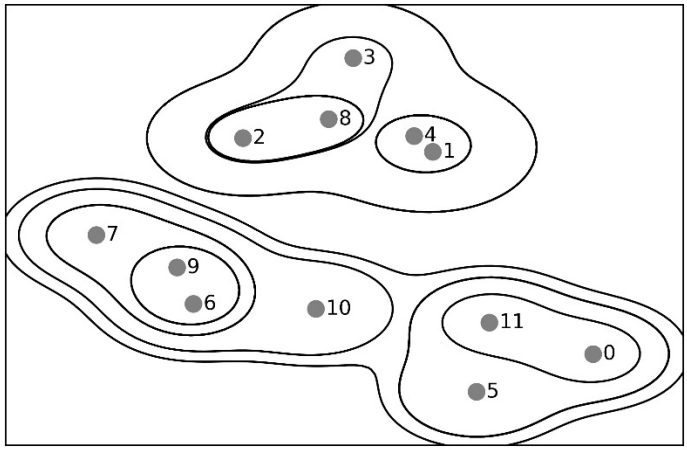


图 4.78 凝聚聚类生成的层次化的簇分配（用线表示）以及带有编号的数据点（参见图 4.79）

虽然这种可视化为层次聚类提供了非常详细的视图，但它依赖于数据的二维性质，因此不能用于具有两个以上特征的数据集。但还有另一个将层次聚类可视化的工具，叫作树状图(dendrogram)，它可以处理多维数据集。

不幸的是，目前 scikit-learn 没有绘制树状图的功能。但你可以利用 SciPy 轻松生成树状图。SciPy 的聚类算法接口与 scikit-learn 的聚类算法稍有不同。SciPy 提供了一个函数，接受数据数组 X 并计算出一个链接数组(linkage array)，它对层次聚类的相似度进行编码。然后我们可以将这个链接数组提供给 scipy 的 dendrogram 函数来绘制树状图(图 4.79)：

```
#从 SciPy 中导入 dendrogram 函数和 ward 函数
from scipy.cluster.hierarchy import dendrogram,ward
X,y = make_blobs(random_state=0,n_samples=12)
#将 ward 聚类应用于数据数组 X
linkage_array = ward(X)
#现在为包含簇之间距离的 linkage_array 绘制树状图
dendrogram(linkage_array)
#在树中标记划分成两个簇或三个簇的位置
ax = plt.gca()
bounds = ax.get_xbound()
ax.plot(bounds,[7.25,7.25],'--',c='k')
ax.plot(bounds,[4,4],'--',c='k')
ax.text(bounds[1],7.25,' two clusters',va='center',fontdict={'size':15})
ax.text(bounds[1],4,' three clusters',va='center',fontdict={'size':15})
plt.xlabel("Sample index")
plt.ylabel("Cluster distance")
'''
```

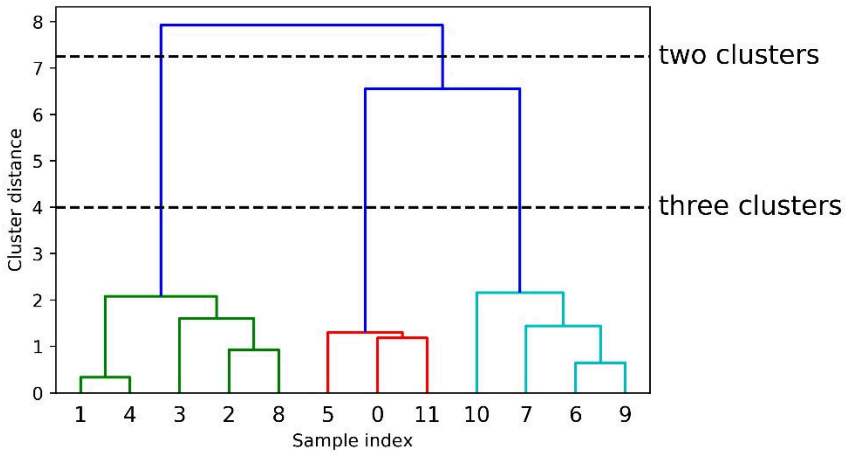


图 4.79 图 4.78 中聚类的树状图（用线表示划分成两个簇和三个簇）

树状图在底部显示数据点(编号从 0 到 11)。然后以这些点(表示单点簇)作为叶节点绘制一棵树，每合并两个簇就添加一个新的父节点。

从下往上看，数据点 1 和 4 首先被合并(正如你在图 4.1 中所见)。接下来，点 6 和 9 被合并为一个簇，以此类推。在顶层有两个分支，一个由点 11、0、5、10、7、6 和 9 组成，另一个由点 1、4,、3、2 和 8 组成。这对应于图中左侧两个最大的族。

树状图的 y 轴不仅说明凝聚算法中两个簇何时合并，每个分支的长度还表示被合并的簇之间的距离。在这张树状图中，最长的分支是用标记为“three clusters”(三个簇)的虚线表示的三条线。它们是最长的分支，这表示从三个簇到两个簇的过程中合并了一些距离非常远的点。我们在图像上方再次看到这一点。将剩下的两个簇合并为一个簇也需要跨越相对较大的距离。

4.4.9 自编码器

自编码器(autoencoder)是神经网络的一种，经过训练后能尝试将输入复制到输出。自编码器(autoencoder)内部有一个隐藏层 h，可以产生编码(code)表示输入。该网络可以看作由两部分组成：一个由函数$h = f(x)$表示的编码器和一个生成重构的解码器$r = g(h)$。图 4.80 展示了这种架构。如果一个自编码器只是简单地学会将处处设置为$g(f(x)) = x$，那么这个自编码器就没什么特别的用处。相反，我们不应该将自编码器设计成输入到输出完全相等。这通常需要向自编码器强加一些约束，使它只能近似地复制，并只能复制与训练数据相似的输入。这些约束强制模型考虑输入数据的哪些部分需要被优先复制，因此它往往能学习到数据的有用特性。

现代自编码器将编码器和解码器的概念推而广之，将其中的确定函数推广为随机映射 $P_{encoder}(h|x)$和$P_{decoder}(x|h)$。数十年间，自编码群的想法一直是神经网络历史景象的一部分(LeCun,1987; Bourlard and Kamp,1988; Hinton and Zemel,1994)。传统自编码器被用于降维或特征学习。近年来，自编码器与潜变量模型理论的联系将自编码器带到了生成式建模的前沿。自编码器可以被看作是前馈网络的一个特例，并且以使用完全相同的技术进行训练，通常使用小批量梯度下降法(其中梯度基于反向传播计算)。

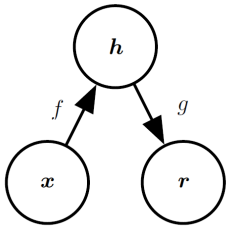


图 4.80 自编码器的一般结构，具有两个组件：编码器 f (将 x 映射到 h)和解码器 g (将 h 映射到 x)

1. 欠完备自编码器

将输入复制到输出听起来没什么用，但我们通常不关心解码器的输出。相反，我们希望通过训练自编码器对输入进行复制而使 h 获得存的特性。从自编码器获得有用特征的一种方法是限制 h 的维度比 x 小，这种编码维度小于输入维度的自编码器称为欠完备(underconplete)自编码器。学习欠完备的表示将强制自编码器捕捉训练数据中最显著的特征。

学习过程可以简单地描述为最小化一个损失函数：

$$L(x, g(f(x))), \tag{4.132}$$

其中 L 是一个损失函数惩罚 $g(f(x))$ 与 x 的差异，如均方误差。

当解码器是线性的且是均方误差，欠完备的自编码器会学习出与 PCA 相同的生成子空间。这种情况下，自编码器在训练来执行复制任务的同时学到了训练数据的主元子空间。

拥有非线性编码器函数 f 和非线性解码器函数 g 的自编码器能够学习出更强大的 PCA 非线性推广。不幸的是，如果编码器和解码器被赋予过大的容量，自编器会执行复制任务而捕捉不到任何有关数据分布的有用信息。从理论上说，我们可以设想这样一个自编码器，它只有一维编码，但它具有一个非常强大的非线性编码器，能够将每个训练数据 $x^{(i)}$ 表示为编码 i。而解码器可以学习将这些整数索引映射回特定训练样本的值。这种特定情形不会在实际情况中发生，但它清楚地说明，如果自编码器的容量太大，那训练来执行复制任务的自编码器可能无法学习到数据集的任何有用信息。

2. 正则编码器

编码维数小于输入维数的欠完备自编码器可以学习数据分布最显著的特征。我们已经知道，如果赋予这类自编码器过大的容量，它就不能学到任何有用的信息。如果隐藏编码的维数允许与输入相等，或隐藏编码维数大于输入的过完备(overcomplete)情况下，会发生类似的问题。在这些情况下，即使是线性编码器和线性解码器也可以学会将输入复制到输出，而学不到任何有关数据分布的有信息。

理想情况下，根据要建模的数据分布的复杂性，选择合适的编码维数和编码器、解码器容量，就可以成功训练任意架构的自编码器。正则自编码器提供这样的能力。正则自编码器使用的损失函数可以鼓励模型学习其他特性(除 将输入复制到输出)，而不必限制使用浅层的编码器和解码器以及小的编码维数来限制模型的容量。这些特性包括稀疏表示、表示的小导数、以及对噪声或输 缺失的鲁棒性。即使模型容量大到足以学习一个无意义的恒等函数，非线全过完备的正则自编码器仍然能够从数据中学到一些关于数据分布的有用信息。

(1) 稀疏自编码器

稀疏自编码器简单地在训练时结合编码层的稀疏惩罚 $\Omega(h)$ 和重构误差：

$$L\left(\mathrm{x},\mathrm{g}\left(f(x)\right)\right)+\Omega(h) \tag{4.133}$$

其中$g(h)$是解码器的输出，通常 h 是编码器的输出，即 $h=f(x)$。

稀疏自编码器一般用来学习特征，以便于像分类这样的任务。稀疏正则化的自编码器必须反映训练数据集的独特统计特征，而不是简单地充当恒等函数。以这种方式训练，执行附带稀疏惩罚的复制任务能学习有用特征的模型。

我们可以简单地将惩罚项 $\Omega(h)$ 视为加到前馈网络的正则项，这个前馈网络的主要任务是将输入复制到输出(无监督学习的目标)，并尽可能地根据这些稀疏特征执行一些监督学习任务(根据监督学习的目标)。不像其它正则项如权重衰减——没有直观的贝叶斯解释。权重衰减和其他正则惩罚可以被解释为一个 MAP 近似贝叶斯推断，正则化的惩罚对应于模型参数的先验概率分布。这种观点认为，正则化的最大似然对应最大化 $p(\theta|x)$，相当于最大化 $\log\ p(x|\theta)+\log\ p(\theta)$。$\log\ p(x|\theta)$即通常的数据似然项，参数的对数先验项$\log p(\theta)$则包含了对$\theta$特定值的偏好。正则自编码器不适用这样的解释是由于正则项取决于数据，因此根据定义上来说，它不是一个先验。虽然如此，我们仍可以认为这些正则项隐式地表达了对函数的偏好。我们可以认为整个稀疏自编码器框架是对带有潜变量的生成模型的近似最大似然训练，而不将稀疏惩罚视为复制任务的正则化。假如我们有一个带有可见变量 x 和潜变量 h 的模型，且具有明确的联合分布 $P_{model}(x,h)=P_{model}(h)P_{model}(x|h)$。我们将$P_{model}(h)$视为模型关于潜变量的先验分布，表示模型看到 x 的信念先验。这与我们之前使用“先验”的方式不同，之前指分布 $p(\theta)$ 在我们看到数据前就对模型参数的先验进行编码。对数似然函数可分解为：

$$logP_{model}(x)=log\sum_h P_{model}(h,x) \tag{4.134}$$

我们可以认为自编码器使用一个高似然值 h 的点估计近似这个总和。这类似于稀疏编码生成模型，但 h 是参数编码器的输出，而不是从优化结果推断出的最可能的 h。从这个角度看，我们根据这个选择的 h，最大化如下：

$$logP_{model}(h,x)=logP_{model}(h)+logP_{model}(x|h) \tag{4.135}$$

$logP_{model}(h)$项能被稀疏诱导。如 Laplace 先验

$$P_{model}(h_i)=\frac{\lambda}{2}e^{-\lambda|h_i|} \tag{4.136}$$

对应于绝对值稀疏惩罚。将对数先验表示为绝对值惩罚，我们得到：

$$\Omega(h)=\lambda\sum_i|h_i| \tag{4.137}$$

$$-\log P_{model}(h)=\sum_i(\lambda|h_i|-log\frac{\lambda}{2})=\Omega(h)+const \tag{4.138}$$

这里的常数项只跟 λ 有关。通常我们将 λ 视为超参数，因此可以丢弃不影响参数学习的常数项。其他如 Student-t 先验也能诱导稀疏性。从稀疏性导致 $P_{model}(h)$ 学习成近似最大似然的结果看，稀疏惩罚完

全不是一个正则项。这仅仅影响模型关于潜变量的分布。这个观点提供了训练自编码器的另一个动机：这是近似训练生成模型的一种途径。这也给出了为什么自编码器学到的特征是有用的另一个解释：它们描述的潜变量可以解释输入。

稀疏自编码器的早期工作(Ranzato et al,2007a，2008)探讨了各种形式的稀疏性，并提出了稀疏惩罚和 log Z 项之间的联系。这个想法是最小化 log Z 防止概率模型处处具有高概率，同理强制稀疏可以防止自编码器处处具有低的重构误差。这种情况下这种联系是对通用机制的直观理解而不是数学上的对应。在数学上更容易解释稀疏惩罚对应于有向模型 $P_{model}(h)P_{model}(x|h)$ 中的 log$P_{model}(h)$。

Glorot et al.(2011b)提出了一种在稀疏(和去噪)自编码器的 h 中实现真正为零的方式。该想法是使用整流线性单元产生编码层。基于将表示真正推向零(如绝对值惩罚)的先验，可以间接控制表示中零的平均数量。

(2) 去噪自编码器

除了向代价函数增加一个惩罚项，我们也可以通过改变重构误差项来获得一个能学到有用信息的自编码。传统的自编码器最小化以下目标：

$$L(x, g(f(x))) \tag{4.139}$$

其中是 L 是一个损失函数，惩罚$g(f(x))$与 x 的差异，如它们彼此差异的 L^2 范数。如果模型被赋予过大的容量，L 仅仅使得 $g \circ f$ 学成一个恒等函数。

相反，去噪自编码器（denoising antoencoder, DAE）最小化：

$$L(x, g(f(\tilde{x}))) \tag{4.140}$$

其中 $\tilde{x}$ 是被某种噪声损坏的 x 的副本。因此去噪自编码器必须撤消这些损坏，而不是简单地复制输人。

Alnin and Bengio (2013)和 Bengio et al.(2013c)指出去噪训练过程强制 f 和 g 隐式地学习 $p_{data}(x)$ 的结构。因此去噪自编码器也是一个通过最小化重构误差获取有用特性的例子。这也是取有用特性的例子。这也是将过完备、高容量的模型用作自编码器的一个例子——只要小心防止这些模型仅仅学习一个恒等函数。

(3) 惩罚导数作为正则

另一正则化自编码器的策略是使用一个类似稀疏自编码器中的惩罚项 Ω

$$L\left(x, g\left(f(x)\right)\right) + \Omega(h, x) \tag{4.141}$$

但 Q 的形式不同： m

$$\Omega(h, x) = \lambda \sum_i \|\nabla_x h_i\|^2 \tag{4.142}$$

这迫使模型学习一个在 x 变化小时目标也没有太大变化的函数。因为这个惩罚只对训练数据适用，它迫使自编码器学习可以反映训练数据分布信息的特征。这样正则化的自编码器被称为收缩自编码器（conttactive mutoencuder, CAE)。这种方法与去噪自编码器、流形学习和概率模型存在一定理论联系。

3. 随机编码器和解码器

自编码器本质上是一个前馈网络，可以使用与传统前馈网络相同的损失函数和输出单元。设计前馈网络的输出单元和损失函数普遍策略是定义一个输出分布 p(y|z)并最小化负对数似然-logp(y|z)。在这种情况下，v 是关于目标的向量（如类标）。

在自编码器中，x 既是输入也是目标。然而，我们仍然可以使用与之前相同的架构。给定一个隐藏编码 h，我们可以认为解码器提供了一个条件分布 $P_{model}(x|h)$ 。接着我们根据最小化$-\log P_{decoder}(x|h)$来训练自编码器。损失函数的具体形式视 $P_{decoder}$ 的形式而定。就传统的前馈网络来说，如果 x 是实值的，那么我们通常使用线性输出单元参数化高斯分布的均值。在这种情况下，负对数似然对应均方误差准则。类似地，二值 x 对应于一个 Bernoulli 分布，其参数由 sigmoid 输出单元确定的。而离散的 x 对应 softmax 分布，以此类推。在给定 h 的情况下，为了便于计算概率分布，输出变量通常被视为是条件独立的，但一些技术(如混合密度输出)可以解决输出相关的建模。

为了更彻底地与我们之前了解到的前馈网络相区别，我们也可以将编码函数(eneoding function) $f(x)$ 的概念推广为编码分布(encoding distribution) $P_{encoder}(h|x)$，如图 4.81 中所示。

任何潜变量模型$P_{model}(h|x)$定义一个随机编码器

$$P_{encoder}(h|x) = P_{model}(h|x) \tag{4.143}$$

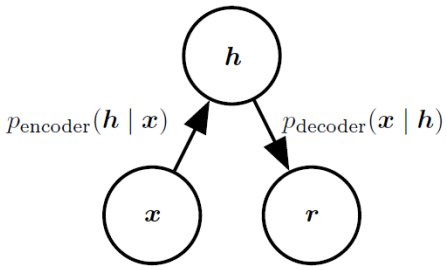


图 4.81 随机自编码器的一般结构，其中编码器和解码器包括一些噪声注入

以及一个随机解码器：

$$P_{decoder}(x|h) = P_{model}(x|h) \tag{4.144}$$

通常情况下，编码器和解码器的分布没有必要是与唯一一个联合分布 $P_{model}(x,h)$ 相容的条件分布。Alain *et.al.* (2015)指出，在保证足够的容量和样本的情况下，将编码器和解码器作为去噪自编码器的训练，能使它们渐进地相容。

4. 自编码器的实现

自编码器是一种非常通用的神经网络工具。主要思想是通过一个编码器，将原始信息编码为一组向量，然后通过一个解码器，将向量解码为原始数据。通过衡量输入与输出的差别，来对网络参数进行训练。主要可以用来进行信息压缩、降噪、添加噪声等工作。

在条件 GAN 中，生成器类似于自编码器中的解码器。都是通过给定一组输入，来得到相应的图片。大家可能好奇自编码器产生的编码信息，在手动修改编码信息后，解码出来的图像会是什么样子。我们可以

利用卷积神经网络构建一个简单的自编码器。

```
import tensorflow as tf
import keras
import numpy as np
from keras.datasets import mnist
from keras.preprocessing import sequence
from keras.models import Sequential, Model
from keras.layers import Dense, Embedding, Reshape
from keras.layers import GRU, Input, Lambda
from keras.layers import Dropout, Flatten
from keras.layers import Conv2D, MaxPooling2D, UpSampling2D
from keras.callbacks import TensorBoard, CSVLogger, EarlyStopping
from keras import backend as K
from PIL import Image
```

```
batch_size = 128
num_classes = 10
epochs = 12
img_rows, img_cols = 28, 28
(x_train, y_train), (x_test, y_test) = mnist.load_data()
if K.image_data_format() == 'channels_first':
        x_train = x_train.reshape(x_train.shape[0], 1, img_rows, img_cols)
        x_test = x_test.reshape(x_test.shape[0], 1, img_rows, img_cols)
        input_shape = (1, img_rows, img_cols)
else:
        x_train = x_train.reshape(x_train.shape[0], img_rows, img_cols, 1)
        x_test = x_test.reshape(x_test.shape[0], img_rows, img_cols, 1)
        input_shape = (img_rows, img_cols, 1)
x_train = x_train.astype('float32')
x_test = x_test.astype('float32')
x_train /= 255
x_test /= 255
print('x_train shape:', x_train.shape)
print(x_train.shape[0], 'train samples')
print(x_test.shape[0], 'test samples')
y_train = keras.utils.to_categorical(y_train, num_classes)
y_test = keras.utils.to_categorical(y_test, num_classes)
input_img = Input(shape=(28, 28, 1))
x = Conv2D(16, (3, 3), activation='relu', padding='same')(input_img)
x = MaxPooling2D((2, 2), padding='same')(x)
x = Conv2D(8, (3, 3), activation='relu', padding='same')(x)
x = MaxPooling2D((2, 2), padding='same')(x)
x = Conv2D(8, (3, 3), activation='relu', padding='same')(x)
encoded = MaxPooling2D((2, 2), padding='same')(x)
x = Conv2D(8, (3, 3), activation='relu', padding='same')(encoded)
x = UpSampling2D((2, 2))(x)
x = Conv2D(8, (3, 3), activation='relu', padding='same')(x)
x = UpSampling2D((2, 2))(x)
x = Conv2D(16, (3, 3), activation='relu')(x)
x = UpSampling2D((2, 2))(x)
decoded = Conv2D(1, (3, 3), activation='sigmoid', padding='same')(x)
autoencoder = Model(inputs=input_img, outputs=decoded)
autoencoder.compile(optimizer='adagrad', loss='binary_crossentropy')

autoencoder.fit(x_train, x_train, epochs=50, batch_size=256,
                        shuffle=True, validation_data=(x_test, x_test),
                        callbacks=[TensorBoard(log_dir='autoencoder')])
decoded_imgs = autoencoder.predict(x_test)
```

在 50 个 epoch 后，网络的 loss 在 0.11 左右，已经能很好的编码 mnist 图像了。

```
Epoch 50/50
60000/60000 [==============================] - 5s 76us/step - loss: 0.1123 - val_loss:
```

可以看出原始图像和编码-解码后的图像差别不大，只是编码-解码的图像会有点模糊：

```
plotimg(decoded_imgs[i])
```

```
plotimg(x_test[i])
```

5. 使用 tensorflow 实现对手写字(MNIST)的 AutoEncoder

```
import tensorflow as tf
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
# Import MNIST data
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("/tmp/data/", one_hot=False)
# Visualize decoder setting
# Parameters
learning_rate = 0.01
batch_size = 256
display_step = 1
examples_to_show = 10
# Network Parameters
n_input = 784    # 28x28 pix，即 784 Features
# tf Graph input (only pictures)
X = tf.placeholder("float", [None, n_input])
n_hidden_1 = 256 #  经过第一个隐藏层压缩至256个
n_hidden_2 = 128 #  经过第二个压缩至128个
weights = {
    'encoder_h1': tf.Variable(tf.random_normal([n_input, n_hidden_1])),
        'encoder_h2': tf.Variable(tf.random_normal([n_hidden_1, n_hidden_2])),
        'decoder_h1': tf.Variable(tf.random_normal([n_hidden_2, n_hidden_1])),
        'decoder_h2': tf.Variable(tf.random_normal([n_hidden_1, n_input])),
  }
```

```
biases = {
    'encoder_b1': tf.Variable(tf.random_normal([n_hidden_1])),
    'encoder_b2': tf.Variable(tf.random_normal([n_hidden_2])),
    'decoder_b1': tf.Variable(tf.random_normal([n_hidden_1])),
    'decoder_b2': tf.Variable(tf.random_normal([n_input])),
}
# Building the encoder
def encoder(x):
    # Encoder Hidden layer 使用的 Activation function 是 sigmoid #1
    layer_1 = tf.nn.sigmoid(tf.add(tf.matmul(x, weights['encoder_h1']),
                                   biases['encoder_b1']))
    # Decoder Hidden layer with sigmoid activation #2
    layer_2 = tf.nn.sigmoid(tf.add(tf.matmul(layer_1, weights['encoder_h2']),
                                   biases['encoder_b2']))
    return layer_2
# Building the decoder
def decoder(x):
    # Encoder Hidden layer with sigmoid activation #1
    layer_1 = tf.nn.sigmoid(tf.add(tf.matmul(x, weights['decoder_h1']),
                                   biases['decoder_b1']))
    # Decoder Hidden layer with sigmoid activation #2
    layer_2 = tf.nn.sigmoid(tf.add(tf.matmul(layer_1, weights['decoder_h2']),
                                   biases['decoder_b2']))
    return layer_2
# Construct model
encoder_op = encoder(X)
decoder_op = decoder(encoder_op)

# Prediction
y_pred = decoder_op
# Targets (Labels) are the input data.
y_true = X
# Define loss and optimizer, minimize the squared error
#比较原始数据与还原后的拥有 784 Features 的数据进行 cost 的对比，
#根据 cost 来提升我的 Autoencoder 的准确率
loss = tf.reduce_mean(tf.pow(y_true - y_pred, 2))#进行最小二乘法的计算(y_true - y_pred)^2
#loss = tf.reduce_mean(tf.square(y_true - y_pred))
optimizer = tf.train.AdamOptimizer(learning_rate).minimize(loss)
# Launch the graph
with tf.Session() as sess:
    init = tf.global_variables_initializer()
    sess.run(init)
    total_batch = int(mnist.train.num_examples/batch_size)
    training_epochs = 20
```

```
    # Training cycle
    for epoch in range(training_epochs):#到好的的效果，我们应进行 10～20 个 Epoch 的训练
        # Loop over all batches
        for i in range(total_batch):
            batch_xs, batch_ys = mnist.train.next_batch(batch_size)  # max(x) = 1, min(x) = 0
            # Run optimization op (backprop) and cost op (to get loss value)
            _, c = sess.run([optimizer, loss], feed_dict={X: batch_xs})
        # Display logs per epoch step
        if epoch % display_step == 0:
            print("Epoch:", '%04d' % (epoch+1),
                  "cost=", "{:.9f}".format(c))
    print("Optimization Finished!")
    # Applying encode and decode over test set
    encode_decode = sess.run(
        y_pred, feed_dict={X: mnist.test.images[:examples_to_show]})
    # Compare original images with their reconstructions
    f, a = plt.subplots(2, 10, figsize=(10, 2))
    for i in range(examples_to_show):
        a[0][i].imshow(np.reshape(mnist.test.images[i], (28, 28)))
        a[1][i].imshow(np.reshape(encode_decode[i], (28, 28)))
    plt.show()
    # encoder_result = sess.run(encoder_op, feed_dict={X: mnist.test.images})
    # sc = plt.scatter(encoder_result[:, 0], encoder_result[:, 1], c=mnist.test.labels) #散点图
    # plt.colorbar(sc) #scatter 设置颜色渐变条 colorbar
    # plt.show()

'''
```

运行结果：上面一行图片为测试图片，第二行为经过自编码器后输出。

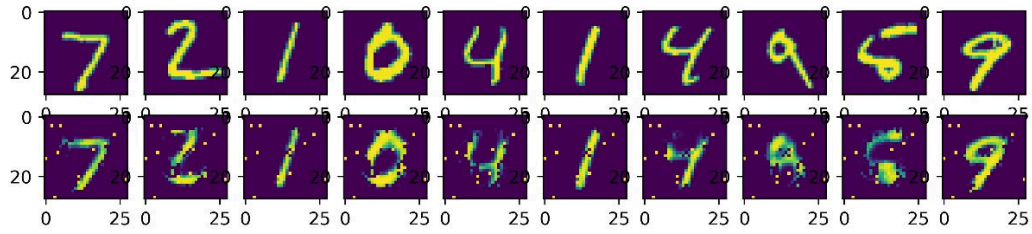


图 4.82 手写数字照片经自编码器输出与原照片对比

4.4.10 朴素贝叶斯

朴素贝叶斯(Naive Bayes)法是基于贝叶斯定理特征条件独立假设的分类方法对于给定的训练数据集,首先基于特征条件独立假设学习输入输出的联合概率分布；然后基于此模型,对给定的输入,利用贝叶斯定理求出后验概率最大的输出 y，朴素贝叶斯法实现简单,学习与预测的效率都很高，是一种常用的方法本章叙述朴素贝叶斯法,包括朴素贝叶斯法的学习与分类、朴素贝叶斯法的参数估计算法。

1． 朴素贝叶斯法的学习与分类

(1) 基本方法

设输入空间$x \subseteq R^n$为 n 维向量的集合,输出空间为类标记集合$y = \{c_1, c_2, \ldots, c_K\}$，输入为特征向量 x∈X,输出为类标记(class label) y∈Y，X 是定义在输入空间 X 上的随机向量，Y 是定义在输出空间 Y 上的随机变量.$P(X,Y)$是 X 和 Y 的联合概率分布，训练数据集由$P(X,Y)$独立同分布产生，

$$T = \{(x_1, y_1), (x_2, y_2), \cdots, (x_N, y_N)\} \tag{4.145}$$

朴素贝叶斯法通过训练数据集学习联合概率分布 P(X,n)具体地,学习以下先验概率分布及条件概率分布，先验概率分布

$$P(Y = c_k), k = 1,2, \cdots, K \tag{4.146}$$

条件概率分布

$$(X = x \mid Y = c_k) = P\left(X^{(1)} = x^{(1)}, \cdots, X^{(n)} = x^{(n)} \mid Y = c_k\right), k = 1,2, \cdots, K \tag{4.147}$$

于是学习到联合概率分布$P(X,Y)$。

条件概率分布 $P(X = x \mid Y = c_k)$有指数级数量的参数,其估计实际是不可行的。事实上，假设$x^{(j)}$可取值有S_j个，j=1，2，…，n，Y 可取值有 K 个,那么参数个数为$K\sum_{j=1}^{n} S_j$。

朴素贝叶斯法对条件概率分布作了条件独立性的假设.由于这是一个较强的假设,朴素贝叶斯法也由此得名.具体地,条件独立性假设是

$$\begin{aligned} P(X = x \mid Y = c_k) &= P\left(X^{(1)} = x^{(1)}, \cdots, X^{(n)} = x^{(n)} \mid Y = c_k\right) \\ &= \prod_{j=1}^{n} P\left(X^{(j)} = x^{(j)} \mid Y = c_k\right) \end{aligned} \tag{4.148}$$

朴素贝叶斯法实际上学习到生成数据的机制,所以属于生成模型。条件独立假设等于是说用于分类的特征在类确定的条件下都是条件独立的。这一假设使朴素贝叶斯法变得简单，但有时会牺牲一定的分类准确率。

朴素贝叶斯法分类时,对给定的输入 x,通过学习到的模型计算后验概率分布$P(Y = c_k|X = x)$，将后验概率最大的类作为 x 的类输出.后验概率计算根据贝叶斯定理进行:

$$P(Y = c_k \mid X = x) = \frac{P(X=x|Y=c_k)P(Y=c_k)}{\sum_k P(X=x|Y=c_k)P(Y=c_k)} \tag{4.149}$$

将式(4.148)代入式(4.149)有

$$P(Y = c_k \mid X = x) = \frac{P(Y=c_k)\prod_j P\left(X^{(j)}=x^{(j)}|Y=c_k\right)}{\sum_k P(Y=c_k)\prod_j P\left(X^{(j)}=x^{(j)}|Y=c_k\right)}, \quad k = 1,2, \cdots, K \tag{4.150}$$

这是朴素贝叶斯法分类的基本公式.于是,朴素贝叶斯分类器可表示为

$$y = f(x) = \arg\max_{a_k} \frac{P(Y=c_k)\prod_j P\left(X^{(j)}=x^{(j)}|Y=c_k\right)}{\sum_k P(Y=c_k)\prod_j P\left(X^{(j)}=x^{(j)}|Y=c_k\right)} \tag{4.151}$$

注意到,在式(4.151)中分母对所有c_k都是相同的，所以，

$$y = \arg\max_{c_k} P(Y=c_k)\prod_j P\left(X^{(J)}=x^{(j)} \mid Y=c_k\right) \tag{4.152}$$

(2) 后验概率最大化的含义

朴素贝叶斯法将实例分到后验概率最大的类中.这等价于期望风险最小化.假设选择 0-1 损失函数：

$$L\left(Y,f(X)\right)=\begin{cases}1, & Y\neq f(X)\\ 0, & Y=f(X)\end{cases} \tag{4.153}$$

式中 f(x)是分类决策函数。这时，期望风险函数为

$$R_{\text{cop}}(f)=E[L(Y,f(X))] \tag{4.154}$$

期望是对联合分布 P(X,Y)取的。由此取条件期望

$$R_{\text{exp}}(f)=E_X\sum_{k=1}^{K}\left[L\left(c_k,f(X)\right)\right]P(c_k \mid X) \tag{4.155}$$

为了使期望风险最小化，只需对$X=x$逐个极小化,由此得到

$$\begin{aligned} f(x) &= \arg\min_{y\in\mathcal{Y}}\sum_{k=1}^{k} L(c_k,y)P(c_k \mid X=x)\\ &= \arg\min_{y\in\mathcal{Y}}\sum_{k=1}^{K} P(y\neq c_k \mid X=x)\\ &= \arg\min_{y\in\mathcal{Y}}\left(1-P(y=c_k \mid X=x)\right)\\ &= \arg\max_{y\in\mathcal{Y}} P(y=c_k \mid X=x) \end{aligned} \tag{4.156}$$

这样一来,根据期望风险最小化准则就得到了后验概率最大化准则:

$$f(x)=\arg\max_{c_k} P(c_k \mid X=x) \tag{4.157}$$

即朴素贝叶斯法所采用的原理。

2. 朴素贝叶斯法的参数估计

(1) 极大似然估计

在朴素贝叶斯法中,学习意味着估计$P(Y=c_k)$和$P(X^{(j)}=x^{(j)}|Y=c_k)$。可以应用极大似然估计法估计相应的概率。先验概率$P(Y=c_k)$的极大似然估计是

$$P(Y=c_k)=\frac{\sum_{i=1}^{N} I(y_i=c_k)}{N}, k=1,2,\cdots,K \tag{4.158}$$

设第j个特征$x^{(j)}$可能取值的集合为$\{a_{j1},a_{j2},\ldots,a_{jS}\}$,条件概率$P(X^{(j)}=a_{ji}|Y=c_k)$的极大似然估计是

$$\begin{gathered}P\left(X^{(j)}=a_{jl}\mid Y=c_k\right)=\frac{\sum_{i=1}^{N} I\left(x_i^{(j)}=a_{jl},y_i=c_k\right)}{\sum_{i=1}^{N} I(y_i=c_k)}\\ j=1,2,\cdots,n;l=1,2,\cdots,S_j:k=1,2,\cdots,K\end{gathered} \tag{4.159}$$

式中，$x_i^{(j)}$是第i个样本的第j个特征；a_{jl}是第j个特征可能取的第 i 个值：I 为指示函数。

(2) 学习与分类算法

下面给出朴素贝叶斯法的学习与分类算法

算法 4.1(朴素贝叶斯算法((naive Bayes algorithm))

输入:训练数据$T=\{(x_1,y_1),(x_2,y_2),\cdots,(x_N,y_N)\}$，其中$x_i=\left(x_i^{(1)},x_i^{(2)},\cdots,x_i^{(n)}\right)^{\mathrm{T}}$，$x_i^{(j)}$是第 i 个样本的第 j 个特征，$x_i^{(j)}\in\left\{a_{j1},a_{j2},\cdots,a_{jS_j}\right\}$,$a_{jl}$是第 j 个特征可能的第 l 个值，$j=1,2,\cdots,n,l=1,2,\cdots,S_j,y_i\in\{c_1,c_2,\cdots,c_K\}$，实例 x；

输出：实例 x 的分类。

(1)计算先验概率及条件概率

$$\begin{gathered}P(Y=c_k)=\frac{\sum_{i=1}^{N} I(y_i=c_k)}{N}, k=1,2,\cdots,K\\ P\left(X^{(j)}=a_{jl}\mid Y=c_k\right)=\frac{\sum_{i=1}^{N} I\left(x_i^{(j)}=a_{jl}y_i=c_k\right)}{\sum_{i=1}^{N} I(y_i=c_k)}\\ j=1,2,\cdots,n;l=1,2,\cdots,S_j;k=1,2,\cdots,K\end{gathered} \tag{4.160}$$

(2)对于给定的实例$(x^{(1)},x^{(2)},\ldots,x^{(n)})^T$，计算

$$P(Y=c_k)\prod_{j=1}^{n} P\left(X^{(j)}=x^{(j)}\mid Y=c_k\right),k=1,2,\cdots,K \tag{4.161}$$

(3) 确定实例 x 的类

$$y=\arg\max_{q} P(Y=c_k)\prod_{j=1}^{n} P\left(X^{(j)}=x^{(S)}\mid Y=c_k\right) \tag{4.162}$$

例 4.1 试由表 4.6 的训练数据学习一个朴素贝叶斯分类器并确定$x=(2,S)^T$的类标记 y，表中$X^{(1)}$，$X^{(2)}$为特征，取值的集合分别为A_1={1,2,3},A_2={S,M,L}，Y 为类标记,Y∈C={1-1}。

表 4. 6 训练数据

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$X^{(1)}$	1	1	1	1	1	2	2	2	2	2	3	3	3	3	3
$X^{(2)}$	S	M	M	S	S	S	M	M	L	L	L	M	M	L	L
Y	-1	-1	1	1	-1	-1	-1	1	1	1	1	1	1	1	-1

解 根据算法 4.1,由表 4.1,容易计算下列概率：

$$P(Y=1)=\frac{9}{15}, P(Y=-1)=\frac{6}{15}$$

$$P\left(X^{(1)}=1 \mid Y=1\right)=\frac{2}{9}, P\left(X^{(1)}=2 \mid Y=1\right)=\frac{3}{9}, P\left(X^{(1)}=3 \mid Y=1\right)=\frac{4}{9}$$

$$P\left(X^{(2)}=S \mid Y=1\right)=\frac{1}{9}, P\left(X^{(2)}=M \mid Y=1\right)=\frac{4}{9}, P\left(X^{(2)}=L \mid Y=1\right)=\frac{4}{9}$$

$$P\left(X^{(1)}=1 \mid Y=-1\right)=\frac{3}{6}, P\left(X^{(1)}=2 \mid Y=-1\right)=\frac{2}{6}, P\left(X^{(1)}=3 \mid Y=-1\right)=\frac{1}{6}$$

$$P\left(X^{(2)}=S \mid Y=-1\right)=\frac{3}{6}, P\left(X^{(2)}=M \mid Y=-1\right)=\frac{2}{6}, P\left(X^{(2)}=L \mid Y=-1\right)=\frac{1}{6}$$

对于给定的$x=（2，S）^T$计算：

$$P(Y=1)P\left(X^{(1)}=2 \mid Y=1\right)P\left(X^{(2)}=S \mid Y=1\right)=\frac{9}{15}\cdot\frac{3}{9}\cdot\frac{1}{9}=\frac{1}{45}$$

$$P(Y=-1)P\left(X^{(1)}=2 \mid Y=-1\right)P\left(X^{(2)}=S \mid Y=-1\right)=\frac{6}{15}\cdot\frac{2}{6}\cdot\frac{3}{6}=\frac{1}{15}$$

因为$P(Y=-1)P\left(X^{(1)}=2 \mid Y=-1\right)P\left(X^{(2)}=S \mid Y=-1\right)$最大，所以 y=-1。

6. 贝叶斯估计

用极大似然估计可能会出现所要估计的概率值为 0 的情况.这时会影响到后验概率的计算结果,使分类产生偏差解决这一问题的方法是采用贝叶斯估计.具体地,条件概率的贝叶斯估计是

$$P_\lambda\left(X^{(j)}=a_{jl} \mid Y=c_k\right)=\frac{\sum_{i=1}^{N} I\left(x_i^{(j)}=a_{jl}, y_i=c_k\right)+\lambda}{\sum_{i=1}^{N} I\left(y_i=c_k\right)+S_j\lambda} \quad (4.x)$$

式中$\lambda \geqslant 0$.等价于在随机变量各个取值的频数上赋予一个正数 $\lambda>0$ 。当 $\lambda=0$ 时就是极大似然估计。常取$\lambda=1$,这时称为拉普拉斯平滑(Laplace smoothing)。显然，对任何$l=1,2,\ldots,S_j$，$k=1,2,\ldots,K$，有

$$\begin{aligned} &P_\lambda\left(X^{(j)}=a_{jl} \mid Y=c_k\right)>0 \\ &\sum_{l=1}^{S_f} P\left(X^{(j)}=a_{jl} \mid Y=c_k\right)=1 \end{aligned} \quad (4.x)$$

表明式(4.10)确为一种概率分布,同样,先验概率的贝叶斯估计是

$$P_\lambda(Y=c_k)=\frac{\sum_{i=1} I\left(y_i=c_k\right)+\lambda}{N+K\lambda} \quad (4.x)$$

例 4.2 问题同例 4.1,按照拉普拉斯平滑估计概率,即取$\lambda=1$

解 $A_1,=\{1,2,3\}, A_2=\{S,M,L\}, C=\{1,-1\}$，按照式(4.x)和式(4.x)计算下列概率：

$$P(Y=1)=\frac{10}{17},P(Y=-1)=\frac{7}{17}$$
$$P\left(X^{(1)}=1\mid Y=1\right)=\frac{3}{12},P\left(X^{(1)}=2\mid Y=1\right)=\frac{4}{12},P\left(X^{(1)}=3\mid Y=1\right)=\frac{5}{12}$$
$$P\left(X^{(2)}=S\mid Y=1\right)=\frac{2}{12},P\left(X^{(2)}=M\mid Y=1\right)=\frac{5}{12},P\left(X^{(2)}=L\mid Y=1\right)=\frac{5}{12}$$
$$P\left(X^{(1)}=1\mid Y=-1\right)=\frac{4}{9},P\left(X^{(1)}=2\mid Y=-1\right)=\frac{3}{9},P\left(X^{(1)}=3\mid Y=-1\right)=\frac{2}{9}$$
$$P\left(X^{(2)}=S\mid Y=-1\right)=\frac{4}{9},P\left(X^{(2)}=M\mid Y=-1\right)=\frac{3}{9},P\left(X^{(2)}=L\mid Y=-1\right)=\frac{2}{9}$$

对于给定的$x=（2，S）^T$计算：

$$P(Y=1)P\left(X^{(1)}=2\mid Y=1\right)P\left(X^{(2)}=S\mid Y=1\right)=\frac{10}{17}\cdot\frac{4}{12}\cdot\frac{2}{12}=\frac{5}{153}=0.0327$$
$$P(Y=-1)P\left(X^{(1)}=2\mid Y=-1\right)P\left(X^{(2)}=S\mid Y=-1\right)=\frac{7}{17}\cdot\frac{3}{9}\cdot\frac{4}{9}=\frac{28}{459}=0.0610$$

因为$P(Y=-1)P\left(X^{(1)}=2\mid Y=-1\right)P\left(X^{(2)}=S\mid Y=-1\right)$最大，所以 y=-1。

习题

1. 数据集包括 1000 个样本，其中 500 个正例、500 个反例，将其划分为包含 70%样本的训练集和 30%样本的测试集用于留出法评估，试估算共有多少种划分方式？

2. 试述错误率与 ROC 曲线的联系。

3. 线性判别分析仅在线性可分数据上能获得理想结果，试设计一个改进方法，使其能较好地用于非线性可分数据。

4. 试基于朴素贝叶斯模型推导出生成式半监督学习算法。

5. 对未标记样本进行标记指派与调整的过程中有可能出现类别不平衡问题，试给出考虑该问题后的改进 TSVM 算法。

6. 自训练（self-training）是一种比较原始的半监督学习方法：它先在有标记样本上学习，然后用学得分类器对未标记样本进行判别以获得其伪标记，再在有标记与伪标记样本的合集上重新训练，如此反复。试析该方法有何缺陷。

7. 试编程实现 k 均值算法，设置三组不同的 k 值、三组不同初始中心点，在西瓜数据集 4.0 上进行实验比较，并讨论什么样的初始中心有利于取得好结果。

8. 聚类结果中若每个簇都有一个凸包（包含簇样本的凸多面体），且这些凸包不相交，则称为凸聚类。试析本书中介绍的哪些聚类算法只能产生凸聚类，哪些能产生非凸聚类。

9. 降维中涉及的投影矩阵通常要求是正交的。试述正交、非正交投影矩阵用于降维的优缺点。

10. 试述 SVM 对噪声敏感的原因。

11. 试证明：二分类任务中两类数据集满足高斯分布且方差相同时，线性判别分析产生贝叶斯最优分类器。

12. 试析使用“最小训练误差”作为决策树划分选择准则的缺陷。

13. 试选择 4 个 UCI 数据集，对基于信息熵、基尼指数、对率回归进行划分选择的三种决策树算法所产生的未剪枝、预剪枝、后剪枝决策树进行实验比较，并进行适当的统计显著性检验。。

14. 试编程实现标准 BP 算法和累积 BP 算法，在西瓜数据集 3.0 上分别用这两个算法训练一个单隐层网络，并进行比较。

15. 尝试使用 scikit-learn 工具打印出 iris，cancer，make_moons 等数据集的数据结构进行观察比较。

16. 尝试使用 mglearn，matplotlib 等工具绘制 forge，wave 等数据集，了解常用数据集的坐标和布局。

17. 假设数据为多个数据源的混合观察结果，这些数据源之间在统计上是相互独立的，而在 PCA 中只假设数据是不相关的。试用 NumPy 科学工具实现 PCA 过程，思考在 3 种降维技术中，PCA 的应用为何最

为广泛。

18. 给定测试样本，基于某种距离度量找出训练集中与其最靠近的 k 个训练样本，然后基于这 k 个“邻居”的信息来进行预测。尝试使用 scikit-learn，mglearn，matplotlib 等工具实现其过程并进行必要的可视化。

19. 决策树是广泛用于分类和回归任务的模型，而随机森林是决策树的集成过程。尝试使用相应工具在 two_moons 数据集上构造和实现决策树，用 graphviz 可视化树，并用随机森林，梯度提升树实现它。比较以上方法的结果。

20. 常用的回归模型有线性回归和 logistics 回归，尝试使用 scikit-learn，mglearn 等工具实现它们，使用 matplotlib 进行可视化后比较回归结果。

21. 在现实任务中，原始样本空间内也许并不存在一个能正确划分两类样本的超平面，于是可以将样本从原始空间映射到一个更高维的特征空间，使得样本在这个特征空间内线性可分。尝试使用 SVM 的方法模拟高维数据的分类，注意使用核技巧以及 SVM 的调参。

22. 神经网络中的感知机与多层网络利用误差逆传播算法展现了比普通机器学习算法更加强大的表示能力，尝试使用 MLPClassifier 和 MLPRegressor 等工具实现一些简单分类，并比较不同激活函数(tanh,relu)的效果差异。

23. 尝试使用 scikit-learn 等工具实现层次聚类的过程，并使用可视化工具实现聚类的树状图，进行观察。

24. 自编码器是神经网络的一种，经过训练后能尝试将输入复制到输出。试用 tensorflow 深度学习框架在卷积神经网络构建一个简单的自编码器，并观察编码器模型生成的图片与原图的差异。

参考文献

[1] 蔡自兴，徐光祐. (2010).人工智能及其应用. 清华大学出版社,北京.

[2] Peter Harrington, 李锐,李鹏曲,亚东,王斌. (2013). 机器学习实战. 人民邮电出版社,北京.

[3] Boston, Farnham, Sebastopol, Tokyo O’ Reilly Media, 张亮. (2016). Python 机器学习基础教程.人民邮电出版社,北京.

[4] 周志华. (2016).机器学习. 清华大学出版社,北京.

[5] 李航. (2012). 统计学习方法. 清华大学出版社,北京.

[6] Alpaydin, E. (2004). Introduction to Machine Learning.

[7] Aloise, D., A. Deshpa P. Hansen, and P. Popat. (2009). "NP-hardness of Euclidean sum-of-squares clustering." Machine Learning, 75(2):245-248.

[8] Kohonen, T. (1988). "An introduction to neural computing." *Neural Networks,*1(1):3-16.

[9] McCulloch, W. S. and W. Pitts. (1943). "A logical calculus of the ideas immanent in nervous activity." *Bulletin of Mathematical Biophysics*, 5(4):115-133.

[10] Minsky, M. and S. Papert. (1969). Perceptrons. MIT Press, Cambridge, MA. Orr, G. B. and K.-R. Müller, eds. (1998). *Neuml Networks: Tricks of the Trade.Springer*, London, UK.

[11] Pineda, F. J. (1987). "Generalization of Back-Propagation to recurrent neural networks." *Physical Review Letters*, 59(19):2229-2232.

[12] LeCun, Y. (1987). *Modeles connexionistes de l'apprentissage*. Ph.D. thesis, Univ3333ersite de Paris VI. 16,429, 440.

[13] Quinlan, J.R. (1986). *Induction of decision trees*. Machine Learning, 1(1):81-106.

[14] Quinlan, J.R. (1993). *C4.5: Programs for Machine Learning*. Morgan Kaufmann, San Mateo, CA.

[15] Bourlard, H. and Kamp, Y. (1988). Auto-association by multilayer perceptrons and singular value decomposition. *Biological Cybernetics*, 59,291-294. 429.

[16] Hinton, G. E. and Zemel, R. S. (1994). Autoencoders, minimum description length, and Helmholtz free energy. In *NIPS'1993*.429.

[17] Ranzato, M., Poultney, C., Chopra, S., and LeCun, Y. (2007a). Efficient learning of spars representations with an energy-based model. In NIPS'2006. 13,433,451,452.

[18] Ranzato, M., Boureau, Y., and LeCun, Y. (2008). Sparse feature Puhing for deep belief networks. In *NIPS'2007*.433.

[19] Glorot Bordes, A., and Bengio, Y. (2011b). Domain adaptation for large-scale sentiment classification: A deep learning approach. In *ICML'2011*. 433.

[20] Alain, G. and Bengio, Y. (2013). What regularized auto-encoders learn from the data generating distribution. In *ICLR'2013, arXiv:1211.4246*.433,439,445.

[21] Bengio, Y., Yao, L., Alain, G., and Vincent, P.(2013c). Generalized denoising auto-encoders as generative models. In *NIPS'2013*, 433, 607,608.

[22] Alain, G. Bengio, Y., Yao, L., Eric Thibodeau-Laufer, Yosinski, J., and Vincent, P. (2015). GSNs: Generative stochastic networks. *arXiv:1503.05571*.436.607.