机器学习基础

朱明超
Email: deityrayleigh@gmail.com
Github: github.com/MingchaoZhu/DeepLearning

1 学习算法

机器学习算法描述一种能够**从数据中学习的算法**。学习指对于某类任务 T，为其定义性能度量 P，一个计算机程序被认为可以从经验 E 中学习是指：**通过经验 E 改进后，它在任务 T 上的性能度量 P 有所提高**。

任务 T：机器学习**任务**定义为**机器学习系统应该如何处理样本**（Example）。例如，识别手写体数字识别的任务为：通过将输入的图片处理后，输出该图片对应的数字（分类）。**样本**是**量化的特征（Feature）的集合**，用向量 $\boldsymbol{x} \in \mathbb{R}^n$ 表示，其中向量的每个元素 x_i 是一个特征。例如一张图片的特征就是这张图片里的像素点的值。

性能度量 P：为了**评估机器学习的优劣**，需要对算法的输出结果进行**定量的衡量分析**，这就需要合适的性能度量指标。

	指标	说明
True Positive	TP	将正样本预测为正例数目
True Negative	TN	将负样本预测为负例数目
False Positive	FP	将负样本预测为正例数目
False Negative	FN	将正样本预测为负例数目

- 针对**分类**任务 (详细描述见第十一章)：
  - 准确率 (Accuracy)：$acc = \frac{\mathrm{TP+TN}}{\mathrm{TP+TN+FP+FN}}$。
  - 错误率 (Error-rate)：$err = 1 - acc$
  - 精度 (Precision)：$\mathrm{P} = \frac{\mathrm{TP}}{\mathrm{TP+FP}}$
  - 召回率 (Recall)：$\mathrm{R} = \frac{\mathrm{TP}}{\mathrm{TP+FN}}$
  - F_1 值：$\mathrm{F}_1 = \frac{2\mathrm{PR}}{\mathrm{P+R}}$
- 针对**回归**任务：距离误差

经验 E：根据经验 E 的不同，机器学习算法可以分为：无监督 (Unsupervised) 算法和监督 (Supervised) 算法。

- 监督学习算法 (Supervised Learning)：训练集的数据中**包含样本特征和标签值**，常见的分类和回归算法都是有监督的学习算法。
- 无监督学习算法 (Unsupervised Learning)：训练集的数据中**只包含样本特征**，算法需要从中学习出特征中隐藏的结构化特征，聚类、密度估计等都是无监督的学习算法。

1.1 举例：线性回归

线性回归（Linear Regression）的目标：获得一个函数 f，满足 $f(\boldsymbol{x}) = \hat{y}$，其中 $\boldsymbol{x} \in \mathbb{R}^n, \hat{y} \in \mathbb{R}$，使得 $\hat{y}$ 接近于真实的标签 y。

我们定义线性回归的输出为：

$$f(x) = \boldsymbol{w}^\top \boldsymbol{x} \tag{1}$$

其中 $\boldsymbol{w} \in \mathbb{R}^n$ 是我们需要学习的参数 (Parameter)。

在线性回归中，对任务 T 的定义：通过输出 $\hat{y} = \boldsymbol{w}^\top \boldsymbol{x}$，从 $\boldsymbol{x}$ 预测 y。

性能度量 P 的定义：假设测试集的特征和标签分别用 $\boldsymbol{X}^{(test)}$ 和 $\boldsymbol{y}^{(test)}$ 表示。可以采用的性能度量方式是均方误差（Mean Squared Error），如果 $\hat{\boldsymbol{y}}^{(test)}$ 表示模型在测试集上的预测值，那么均方误差公式为：

$$\mathrm{MSE}_{test} = \frac{1}{m} \sum_i (\hat{\boldsymbol{y}}^{(test)} - \boldsymbol{y}^{(test)})_i^2 = \frac{1}{m} ||\hat{\boldsymbol{y}}^{(test)} - \boldsymbol{y}^{(test)}||_2^2 \tag{2}$$

为了构建一个机器学习算法，需要设计一个算法，通过观察训练集 $(\boldsymbol{X}^{(train)}, \boldsymbol{y}^{(train)})$ 获得经验，改进权重 $\boldsymbol{w}$ 以减少 MSE_{test}。一种直观的方式是

最小化训练集上的均方误差，即 MSE_{train}。最小化 MSE_{train}，我们可以简单地求解其导数为 0 的情况：

$$
\begin{aligned}
\nabla_{\boldsymbol{w}} \text{MSE}_{train} &= 0 \\
\Longrightarrow \nabla_{\boldsymbol{w}} \frac{1}{m} &\| \hat{\boldsymbol{y}}^{(train)} - \boldsymbol{y}^{(train)} \|_2^2 = 0 \\
\Longrightarrow \nabla_{\boldsymbol{w}} \frac{1}{m} &\| \boldsymbol{X}^{(train)} \boldsymbol{w} - \boldsymbol{y}^{(train)} \|_2^2 = 0 \\
\Longrightarrow \boldsymbol{w} &= (\boldsymbol{X}^{(train)\top} \boldsymbol{X}^{(train)})^{-1} \boldsymbol{X}^{(train)\top} \boldsymbol{y}^{(train)}
\end{aligned}
\tag{3}
$$

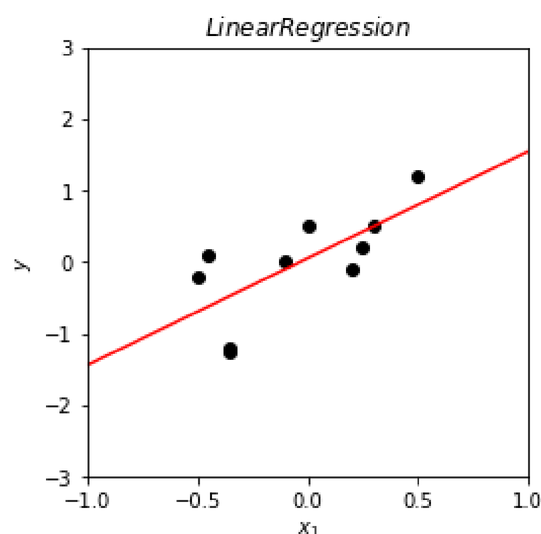
方程的解：$\boldsymbol{w} = (\boldsymbol{X}^{(train)\top} \boldsymbol{X}^{(train)})^{-1} \boldsymbol{X}^{(train)\top} \boldsymbol{y}^{(train)}$ 被称为正规方程。

函数 $f(\boldsymbol{x}) = a\boldsymbol{x} + b$ 称为仿射函数，其中，当 $b = 0$ 时，变为 $f(\boldsymbol{x}) = a\boldsymbol{x}$，称为线性函数，即线性函数是仿射函数的一个特例。

[1]:
```
import numpy as np
import math
import matplotlib.pyplot as plt
```

[2]:
```
X = np.hstack((np.array([[-0.5,-0.45,-0.35,-0.35,-0.1,0,0.2,0.25,0.3,0.5]]).reshape(-1, 1), np.ones((10,1))*1))
y = np.array([-0.2,0.1,-1.25,-1.2,0,0.5,-0.1,0.2,0.5,1.2]).reshape(-1,1)
# 用公式求权重
w = np.linalg.inv(X.T.dot(X)).dot(X.T).dot(y)
hat_y = X.dot(w)
print("Weight:{}".format{list(w)})
x = np.linspace(-1, 1, 50)
hat_y = x * w[0] + w[1]
plt.figure(figsize=(4,4))
plt.xlim(-1.0, 1.0)
plt.xticks(np.linspace(-1.0, 1.0, 5))
plt.ylim(-3, 3)
plt.plot(x, hat_y, color='red')
plt.scatter(X[:,0], y[:,0], color='black')
plt.xlabel('$x_1$')
plt.ylabel('$y$')
plt.title('$Linear Regression$')
plt.show()
```

```
Weight:[array([1.49333333]), array([0.04966667])]
```



2 容量、过拟合、欠拟合

2.1 泛化问题

机器学习的主要挑战在于**在未见过的数据输入上表现良好**，这个能力称为泛化能力 (Generalization)。我们量化一下模型在训练集和测试集上的表现，将其分别称为训练误差 (Training Error) 和测试误差 (Test Error)，后者也经常称为泛化误差 (Generalization Error)。可以说，**理想的模型就是在最小化训练误差的同时，最小化泛化误差**，具有良好泛化能力的算法才是符合需求的。

在实际的应用过程中，会采样两个数据集，减小训练误差得到参数后，再在测试集中验证。这个过程中，就会发生测试误差的期望大于训练误差的期望的情况。以下是决定机器学习算法效果是否好的因素：

- 降低训练误差。

- 缩小训练误差与测试误差之间的差距。

这俩个因素分别对应了机器学习的两个大挑战：欠拟合 (Underfitting) 和过拟合 (Overfitting)。**欠拟合**指的是**模型在训练集上的误差较大**，这通常是由于训练不充分或者模型不合适导致；**过拟合**指的是**模型在训练集和测试集上的误差差距过大**，通常由于模型过分拟合了训练集中的随机噪音，导致泛化能力较差。采用**正则化**，可以**降低泛化误差**，我们会在第七章进一步的介绍。

2.2 容量

通过调节机器学习模型的容量，可以控制模型是否偏于过拟合还是欠拟合，容量 (Capacity) 是描述了**整个模型拟合各种函数的能力**。如果容量不足，模型将不能够很好地表示数据，表现为欠拟合；如果容量太大，那么模型就很容易过分拟合数据，因为其记住了不适合于测试集的训练集特性，表现为过拟合。容量的控制可以通过多种方法控制，包括：

- 控制模型的假设空间。
- 添加正则项对模型进行偏好排除。

当机器学习算法的容量适合于所执行任务的复杂度和所提供训练数据的数量时，算法效果通常会最佳。统计学习方法理论提供了量化模型的容量的不同方法，其中最为出名的是 Vapnik-Chervonenkis 维度 (Vapnik-Chervonenkis dimension)。统计学习理论中最重要的结论阐述了**训练误差和泛化误差之间差异的上界随着模型容量增长而增长，但随着训练样本增多而下降**。

通常，当模型容量上升时，训练误差会下降，直到其渐近最小可能误差（假设误差度量有最小值），而泛化误差会是一个关于模型容量的 U 形曲线函数。

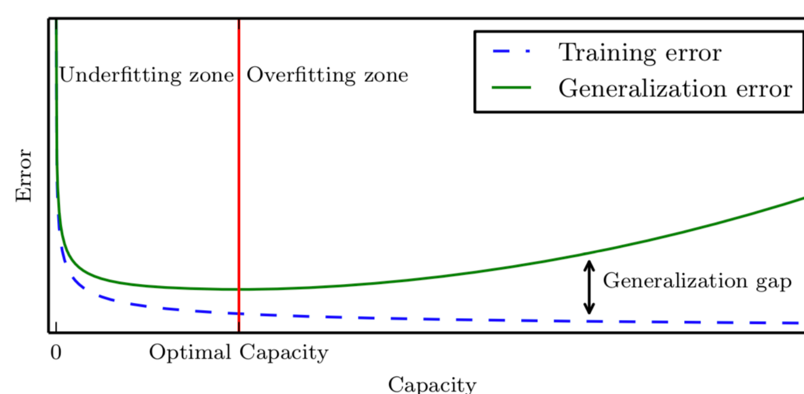


图 1. 容量和误差之间的典型关系

3 超参数与验证集

超参数：用来**控制学习算法的参数**而非学习算法本身学出来的参数。例如，进行曲线的回归拟合时，曲线的次数就是一个超参数；在构建模型对一些参数的分布假设也是超参数。

验证集 (Validation Set)：通常在需要选取超参数时，将训练集再划分为训练和验证集两部分，使用新的训练集训练模型，**验证集用来进行测试和调整超参**。通常，80% 的训练数据用于训练学习参数，20% 用于验证。

k 折交叉验证：将数据集均分为**不相交的 k 份**，每次选取其中的一份作为测试集，其他的为训练集，**训练误差为 k 次的平均误差**。

```
[3]: def KFoldCV(D, A, k):
    """
    k-fold 交叉验证

    参数说明:
    D: 给定数据集
    A: 学习函数
    k: 折数
    """
    np.random.shuffle(D)
    dataset = np.split(D, k)
    acc_rate = 0
    for i in range(k):
        train_set = dataset.copy()
        test_set = train_set.pop(i)
        train_set = np.vstack(train_set)
        A.train(train_set[:,:-1], train_set[:,-1])   # 每次的训练集
        labels = A.fit(test_set[:,:-1])  # 每次的测试集
        acc_rate += np.mean(labels==test_set[:,-1])  # 计算平均误差
```

```
    return acc_rate/k
```

4 偏差和方差

4.1 偏差

估计的偏差 (Bias) 被定义为：

$$bias(\hat{\theta}_m) = \mathbb{E}(\hat{\theta}_m) - \theta \tag{4}$$

其中期望作用在所有数据上，θ 是用于定义数据生成分布的真实值。偏差反映的是模型在样本上的输出与真实值之间的误差，即模型本身的精准度，或者说**算法本身的拟合能力**。

- 如果 $bias(\hat{\theta}_m) = 0$，那么估计量 $\hat{\theta}_m$ 被称为是无偏 (Unbiased)。
- 如果 $\lim_{m\to\infty} bias(\hat{\theta}_m) = 0$，那么估计量 $\hat{\theta}_m$ 被称为是渐进无偏 (Asymptotically Unbiased)。

4.2 方差

估计的方差 (Variance) 被定义为：

$$\operatorname{Var}(\hat{\theta}) \tag{5}$$

方差反映的是模型每一次输出结果与模型输出期望之间的误差，即**模型的稳定性**。

标准差被记为

$$\operatorname{SE}(\hat{\mu}_m) = \sqrt{\operatorname{Var}\left[\frac{1}{m}\sum_{i=1}^{m}\boldsymbol{x}^{(i)}\right]} = \frac{\sigma}{\sqrt{m}} \tag{6}$$

其中，σ^2 是样本 $\{\boldsymbol{x}^{(i)}\}$ 的真实方差，标准差通常被标记为 σ。

4.3 误差与偏差和方差的关系

一个复杂的模型并不总是能在测试集上表现出更好的性能，那么误差源于哪？

以**回归**为例，对测试样本 $\boldsymbol{x}$，令 y_D 为 $\boldsymbol{x}$ 在数据集上的标记，y 为 $\boldsymbol{x}$ 的真实标记。由于噪声的存在，有可能 $y_D \neq y$，$f(\boldsymbol{x}; D)$ 为在训练集 D 上学得函数 f 对 $\boldsymbol{x}$ 的预测输出。因此，算法的期望预测可以表示为：

$$\bar{f}(\boldsymbol{x}) = \mathbb{E}_D[f(\boldsymbol{x}; D)] \tag{7}$$

不同训练集学得的函数 f 的预测输出的方差 (Variance) 为：

$$var(\boldsymbol{x}) = \mathbb{E}_D[(f(\boldsymbol{x}; D) - \bar{f}(\boldsymbol{x}))^2] \tag{8}$$

期望输出与真实标记之间的差距称为偏差 (Bias) 为：

$$bias^2(\boldsymbol{x}) = (\bar{f}(\boldsymbol{x}) - y)^2 \tag{9}$$

噪声 (真实标记与数据集中的实际标记间的偏差) 为：

$$\varepsilon^2 = \mathbb{E}_D[(y_D - y)^2] \tag{10}$$

假定噪声期望为零，即 $\mathbb{E}_D[y_D - y] = 0$。算法的期望泛化误差为：

$$\begin{aligned}
\mathbb{E}(f; D) &= \mathbb{E}_D[(f(\boldsymbol{x}; D) - y_D)^2] \\
&= \mathbb{E}_D[(f(\boldsymbol{x}; D) - \bar{f}(\boldsymbol{x}) + \bar{f}(\boldsymbol{x}) - y_D)^2] \\
&= \mathbb{E}_D[(f(\boldsymbol{x}; D) - \bar{f}(\boldsymbol{x}))^2] + \mathbb{E}_D[(\bar{f}(\boldsymbol{x}) - y_D)^2] + \mathbb{E}_D[2(f(\boldsymbol{x}; D) - \bar{f}(\boldsymbol{x}))(\bar{f}(\boldsymbol{x}) - y_D)] \\
&= \mathbb{E}_D[(f(\boldsymbol{x}; D) - \bar{f}(\boldsymbol{x}))^2] + \mathbb{E}_D[(\bar{f}(\boldsymbol{x}) - y_D)^2] \\
&= \mathbb{E}_D[(f(\boldsymbol{x}; D) - \bar{f}(\boldsymbol{x}))^2] + \mathbb{E}_D[(\bar{f}(\boldsymbol{x}) - y + y - y_D)^2] \\
&= \mathbb{E}_D[(f(\boldsymbol{x}; D) - \bar{f}(\boldsymbol{x}))^2] + \mathbb{E}_D[(\bar{f}(\boldsymbol{x}) - y)^2] + \mathbb{E}_D[(y - y_D)^2] + \mathbb{E}_D[2(\bar{f}(\boldsymbol{x}) - y)(y - y_D)] \\
&= \mathbb{E}_D[(f(\boldsymbol{x}; D) - \bar{f}(\boldsymbol{x}))^2] + (\bar{f}(\boldsymbol{x}) - y)^2 + \mathbb{E}_D[(y - y_D)^2]
\end{aligned} \tag{11}$$

式中，第一个加红公式等于 0，因为 $(f(\boldsymbol{x}; D) - \bar{f}(\boldsymbol{x}))$ 与 $(\bar{f}(\boldsymbol{x}) - y_D)$ 相互独立，所以 $\mathbb{E}_D[2(f(\boldsymbol{x}; D) - \bar{f}(\boldsymbol{x}))(\bar{f}(\boldsymbol{x}) - y_D)] = 2\mathbb{E}_D[(f(\boldsymbol{x}; D) - \bar{f}(\boldsymbol{x}))]\mathbb{E}_D[\bar{f}(\boldsymbol{x}) - y_D)]$。根据期望预测公式 $\bar{f}(\boldsymbol{x}) = \mathbb{E}_D[f(\boldsymbol{x}; D)]$ 有 $\mathbb{E}_D[(f(\boldsymbol{x}; D) - \bar{f}(\boldsymbol{x}))] = 0$。同理第二个加红公式等于 0，因为噪声期望为 0。于是：

$$\mathbb{E}(f; D) = bias^2(\boldsymbol{x}) + var(\boldsymbol{x}) + \varepsilon^2 \tag{12}$$

也就是说，泛化误差可分解为偏差、方差与噪声之和。噪声无法人为控制，所以通常我们认为：

$$\mathbb{E}(f; D) = bias^2(\boldsymbol{x}) + var(\boldsymbol{x}) \tag{13}$$

我们需要在模型复杂度之间权衡，使偏差和方差得以均衡 (trade-off)，这样模型的整体误差才会最小。

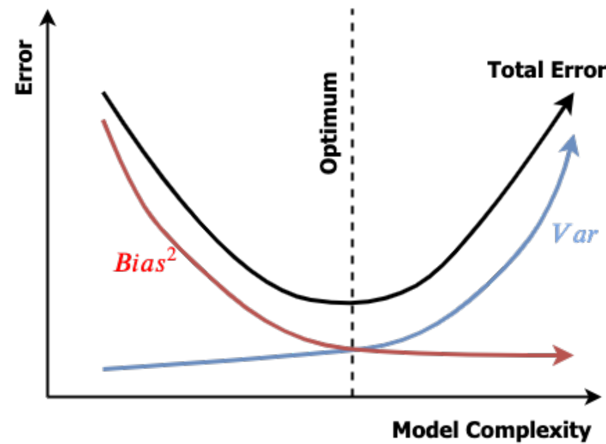


图 2. 当容量增大 (x 轴) 时，偏差 (红线) 随之减小，而方差 (蓝线) 随之增大，使得泛化误差 (黑线) 产生了另一种 U 形。

5 最大似然估计

最大似然估计 (Maximum Likelihood Estimation，MLE) 是一种最为常见的估计准则，其思想是在**已知分布产生的一些样本**而**未知分布具体参数**的情况下**根据样本值推断最有可能产生样本的参数值**。将数据的真实分布记为 $P_{data}(\boldsymbol{x})$，为了使用 MLE，需要先假设样本服从某一簇有参数确定的分布 $P_{model}(\boldsymbol{x};\theta)$，现在的目标就是使用估计的 P_{model} 来拟合真实的 P_{data} (**条件一：“模型已定，参数未知”**)。

对于一组由 m 个样本组成的数据集 $\boldsymbol{X}=\{\boldsymbol{x}^{(1)},\cdots,\boldsymbol{x}^{(m)}\}$，假设数据独立且由未知的真实数据分布 $\boldsymbol{P}_{data}(\boldsymbol{x})$ 生成 (**条件二：独立同分布采样的数据**)，可以通过最大似然估计：

$$\begin{aligned}\theta_{ML} &= \arg\max_{\theta} P_{model}(\boldsymbol{X};\theta) \\ &= \arg\max_{\theta} \prod_{i=1}^{m} P_{model}(\boldsymbol{x}^{(i)};\theta)\end{aligned} \tag{14}$$

获得真实分布的参数。

通常为了计算方便，会对 MLE 加上 log，将乘积转化为求和然后将求和变为期望：$\theta_{ML}=\arg\max_{\theta}\Sigma_{i=1}^{m}\log P_{model}(\boldsymbol{x}^{(i)};\theta)$。

使用训练数据经验分布 $\hat{P}_{data}$ 相关的期望进行计算：$\theta_{ML}=\arg\max_{\theta}\mathbb{E}_{\boldsymbol{x}\sim\hat{P}_{data}}\log P_{model}(\boldsymbol{x};\theta)$。该式是许多监督学习算法的基础假设。

最大似然估计的一种解释是使 P_{model} 与 P_{data} 之间的差异性尽可能的小，形式化的描述为最小化两者的 KL 散度。

6 贝叶斯统计

最大似然估计属于典型的频率学派统计方法，它假设数据是**由单一的最优参数值 θ 生成**，并在此基础上对参数进行估计。而另一种方法是考虑到所有的的参数值以及这些参数的先验概率分布，通过贝叶斯准则来估计参数的后验分布情况，**贝叶斯统计 (Bayesian Statistics) 认为训练数据是确定的，而参数是随机且不唯一的，每个参数都有相应的概率**。

在观察到数据前，将 θ 的已知知识称为先验概率分布 (Prior Probability Distribution) $p(\theta)$。现在有一组数据样本 $\{\boldsymbol{x}^{(1)},\cdots,\boldsymbol{x}^{(m)}\}$，将数据似然及先验代入贝叶斯规则，得到：

$$p(\theta\mid\boldsymbol{x}^{(1)},\cdots,\boldsymbol{x}^{(m)})=\frac{p(\boldsymbol{x}^{(1)},\cdots,\boldsymbol{x}^{(m)}\mid\theta)p(\theta)}{p(\boldsymbol{x}^{(1)},\cdots,\boldsymbol{x}^{(m)})} \tag{15}$$

在贝叶斯常用情景下，**先验是相对均匀的分布或高熵的高斯分布**，观测数据通常会使后验的熵下降，并集中在几个可能性很高的值。当训练数据有限时，贝叶斯方法通常泛化得更好。

贝叶斯估计**假设已知的是参数的先验分布情况和模型的类簇，之后利用数据集的样本点根据贝叶斯准则来对参数的分布情况进行修正**，它得到的结果不是一个单一的参数值，而是根据参数先验分布和真实样本得到的修正过后的参数分布，即参数的后验分布 (Posterior Distribution)。根据贝叶斯估计，在已知 m 个样本后，估计第 $m+1$ 的样本分布的公式如下：

$$p(\boldsymbol{x}^{(m+1)}\mid\boldsymbol{x}^{(1)},\boldsymbol{x}^{(2)}...,\boldsymbol{x}^{(m)})=\int p(\boldsymbol{x}^{(m+1)}\mid\theta)p(\theta|\boldsymbol{x}^{(1)}\boldsymbol{x}^{(2)}...,\boldsymbol{x}^{(m)})d\theta \tag{16}$$

可以看到，不同于频率学派单点估计的方法，贝叶斯估计在对未知数据预测时，将所有的参数分布都进行了考虑，同时按照参数的概率密度情况进行加权。(贝叶斯方法用于推断和预测可以参考第十一章)

7 最大后验估计

完整的贝叶斯估计需要使用参数的完整分布进行预测，然而对绝大多数的机器学习任务而言，这将会导致十分繁重的计算。一种合理的方式是利用最大后验估计 (Maximum A Posteriori, MAP) 来**选取一个计算可行的单点估计参数作为贝叶斯估计的近似解**，公式如下：

$$\theta_{MAP}=\arg\max_{\theta}\log p(\theta\mid\boldsymbol{x})=\arg\max_{\theta}\log p(\boldsymbol{x}\mid\theta)+\log p(\theta) \tag{17}$$

可以看到 MAP 的估计实际上就是对数似然加上参数的先验分布。实际上，在参数服从高斯分布的情况下，上式的右边就对应着 L2 正则项；在 Laplace 的情况下，对应着 L1 的正则项；在均匀分布的情况下则为 0，等价于 MLE。

7.1 举例：线性回归

线性回归 (`Linear Regression`) 的目标：获得一个函数 f，满足 $f(\boldsymbol{x})=\hat{y}$，其中 $\boldsymbol{x}\in\mathbb{R}^n,\hat{y}\in\mathbb{R}$，使得 $\hat{y}$ 接近于真实的标签 y。

我们定义线性回归的输出为：

$$f(\boldsymbol{x})=\boldsymbol{w}^\top\boldsymbol{x}$$

其中 $\boldsymbol{w}\in\mathbb{R}^n$ 是我们需要学习的参数。

在线性回归中，对任务 `T` 的定义：通过输出 $\hat{y}=\boldsymbol{w}^\top\boldsymbol{x}$，从 $\boldsymbol{x}$ 预测 y。假设一共 m 个样本。

现在通过 MLE 解释为什么性能度量方式是均方误差（`Mean Squared Error`）。

我们将模型预测结果和真实标签的差值定义为残差 (`residual`)：$\epsilon=y-f(\boldsymbol{x})$。如果每一次的观测都属于独立事件，所有观测误差的期望和方差应该都一致：这符合中心极限定理，应该构成正态分布，并且误差的期望值应该是 0。所以大多数情况下，可以认为这个误差服从高斯分布，如下：

$$\epsilon\sim N(0,\sigma^2) \tag{18}$$

于是可以得到我们的观测到的标签服从如下高斯分布：$y\sim N(f(\boldsymbol{x}),\sigma^2)$。此时，我们定义了产出观测数据的模型，处于“模型已定，参数未知”的情况，找到一组参数使我们观测到一系列 y 的概率最大 (最大似然估计的思路)。观察到结果 y 的概率密度函数如下：

$$p(y\mid\boldsymbol{x},\boldsymbol{w},\sigma)=\frac{1}{\sqrt{2\pi\sigma^2}}\exp(-\frac{(y-f(\boldsymbol{x}))^2}{2\sigma^2}) \tag{19}$$

将似然函数记作：$\mathcal{L}(\boldsymbol{w},\boldsymbol{X},\sigma)=\prod_{i=1}^m p(y^{(i)}\mid\boldsymbol{x}^{(i)},\boldsymbol{w},\sigma)$

对其取对数并化简：

$$\begin{aligned}\ln\mathcal{L}(\boldsymbol{w},\boldsymbol{X},\sigma)&=\ln\prod_{i=1}^m p(y^{(i)}\mid\boldsymbol{x}^{(i)},\boldsymbol{w},\sigma)\\&=\sum_{i=1}^m\ln p(y^{(i)}\mid\boldsymbol{x}^{(i)},\boldsymbol{w},\sigma)\\&=\sum_{i=1}^m\ln\{\frac{1}{\sqrt{2\pi\sigma^2}}\exp(-\frac{(y^{(i)}-\boldsymbol{w}^\top\boldsymbol{x}^{(i)})^2}{2\sigma^2})\}\\&=m\ln\{\frac{1}{\sqrt{2\pi\sigma^2}}\}-\frac{1}{2\sigma^2}\sum_{i=1}^m(y^{(i)}-\boldsymbol{w}^\top\boldsymbol{x}^{(i)})^2\end{aligned} \tag{20}$$

σ 可以假设为任意大于 0 的常数，优化问题化为剩下部分最大化 (因为负号变成最小化)：

$$\boldsymbol{w}_{\text{MLE}}=\arg\min_{\boldsymbol{w}}\sum_{i=1}^m(y^{(i)}-\boldsymbol{w}^\top\boldsymbol{x}^{(i)})^2 \tag{21}$$

这与 MSE 一致。

接下来考虑参数具有先验知识的情况

如果对 MLE 加入**高斯先验分布**，假设我们要求解的参数 $\boldsymbol{w}$ 本身服从一个先验分布：$\boldsymbol{w}\sim N(\boldsymbol{0},\boldsymbol{\Sigma})$。这里我们就简单化 $\boldsymbol{w}\sim N(0,\gamma^2)$。此时，MAP 最大化的目标函数如下：

$$\begin{aligned}\mathcal{L}(\boldsymbol{w})&=p(y\mid\boldsymbol{X},\boldsymbol{w})p(\boldsymbol{w})\\&=\prod_{i=1}^m\{\frac{1}{\sqrt{2\pi\sigma^2}}\exp(-\frac{(y^{(i)}-\boldsymbol{w}^\top\boldsymbol{x}^{(i)})^2}{2\sigma^2})\}\prod_{j=1}^n\{\frac{1}{\sqrt{2\pi\gamma^2}}\exp(-\frac{(\boldsymbol{w}_j)^2}{2\gamma^2})\}\end{aligned} \tag{22}$$

取对数后，化简整理得如下结果：

$$\ln\mathcal{L}(\boldsymbol{w})=n\ln\frac{1}{\sqrt{2\pi\sigma^2}}+m\ln\frac{1}{\sqrt{2\pi\gamma^2}}-\frac{1}{2\sigma^2}\sum_{i=1}^m(y^{(i)}-\boldsymbol{w}^\top\boldsymbol{x}^{(i)})^2-\frac{1}{2\gamma^2}\boldsymbol{w}^\top\boldsymbol{w} \tag{23}$$

其中，σ 和 γ 均看作是常数，它们的取值会影响两个目标 (likelihood prior) 的权重，所以引入超参数 λ 来表示先验的权重，最终有：

$$\boldsymbol{w}_{\text{MAP}}=\arg\min_{\boldsymbol{w}}\sum_{i=1}^m(y^{(i)}-\boldsymbol{w}^\top\boldsymbol{x}^{(i)})^2+\lambda||\boldsymbol{w}||^2 \tag{24}$$

上式就是标准的岭回归 (`Ridge Regression`) 公式，就是在最小二乘法的基础上增加的参数本身的先验分布，并认为参数本身服从高斯分布。

如果对 MLE 加入**拉普拉斯分布**，同样的步骤，通过 MAP 可以化简得到 (LASSO)：

$$\boldsymbol{w}_{\text{MAP}}=\arg\min_{\boldsymbol{w}}\sum_{i=1}^m(y^{(i)}-\boldsymbol{w}^\top\boldsymbol{x}^{(i)})^2+\lambda||\boldsymbol{w}||^1 \tag{25}$$

如果对 MLE 加入**均匀分布**，最终 MAP 得到的和 MLE 一样。

自定义实现

[4]:
```
class NaiveBayes():

    def __init__(self):
        self.parameters = [] # 保存每个特征针对每个类的均值和方差
        self.y = None
        self.classes = None

    def fit(self, X, y):
        self.y = y
        self.classes = np.unique(y) # 类别
        # 计算每个特征针对每个类的均值和方差
        for i, c in enumerate(self.classes):
            # 选择类别为 c 的 X
            X_where_c = X[np.where(self.y == c)]
            self.parameters.append([])
            # 添加均值与方差
            for col in X_where_c.T:
                parameters = {"mean": col.mean(), "var": col.var()}
                self.parameters[i].append(parameters)

    def _calculate_prior(self, c):
        """
        先验函数。
        """
        frequency = np.mean(self.y == c)
        return frequency

    def _calculate_likelihood(self, mean, var, X):
        """
        似然函数。
        """
        # 高斯概率
        eps = 1e-4 # 防止除数为 0
        coeff = 1.0 / math.sqrt(2.0 * math.pi * var + eps)
        exponent = math.exp(-(math.pow(X - mean, 2) / (2 * var + eps)))
        return coeff * exponent

    def _calculate_probabilities(self, X):
        posteriors = []
        for i, c in enumerate(self.classes):
            posterior = self._calculate_prior(c)
            for feature_value, params in zip(X, self.parameters[i]):
                # 独立性假设
                # P(x1,x2|Y) = P(x1|Y)*P(x2|Y)
                likelihood = self._calculate_likelihood(params["mean"], params["var"], feature_value)
                posterior *= likelihood
            posteriors.append(posterior)
        # 返回具有最大后验概率的类别
        return self.classes[np.argmax(posteriors)]

    def predict(self, X):
        y_pred = [self._calculate_probabilities(sample) for sample in X]
        return y_pred

    def score(self, X, y):
        y_pred = self.predict(X)
        accuracy = np.sum(y == y_pred, axis=0) / len(y)
        return accuracy
```

用自定义贝叶斯估计，*iris* 数据集测试

[5]:
```
import pandas as pd
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
```

[6]:
```
def create_data():
    iris = load_iris()
    df = pd.DataFrame(iris.data, columns=iris.feature_names)
    df['label'] = iris.target
    df.columns = ['sepal length', 'sepal width', 'petal length', 'petal width', 'label']
    data = np.array(df.iloc[:100, :])
    return data[:,:-1], data[:,-1]

X, y = create_data()
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3)
print(X_train[0], y_train[0])
```

```
[5.8 2.6 4.  1.2] 1.0
```

[7]:
```
model = NaiveBayes()
model.fit(X_train, y_train)
print(model.score(X_test, y_test))
```

```
1.0
```

用 sklearn 实现贝叶斯估计，*iris* 数据集测试

[8]:
```
# 从 sklearn 包中调用 GaussianNB 实现贝叶斯估计
from sklearn.naive_bayes import GaussianNB
skl_model = GaussianNB()
skl_model.fit(X_train, y_train)
print(skl_model.score(X_test, y_test))
```

```
1.0
```

8 监督学习方法

8.1 概率监督学习

通过定义一族不同的概率分布，可以将线性回归扩展到分类情况中：$p(y \mid \boldsymbol{x};\theta) = N(y;\theta^\top \boldsymbol{x}, I)$。

由于二元变量上的分布中，均值必须始终在 0 和 1 之间，为解决这个问题，可以使用 logistic sigmoid 函数将线性函数的输出压缩到区间 (0, 1) 上，则概率为：

$$p(y=1 \mid \boldsymbol{x};\theta) = \sigma(\theta^\top \boldsymbol{x}) \tag{26}$$

这个方法称为**逻辑回归** (`Logistic Regression`)。其中 logistic sigmoid 函数描述为 $\sigma(x) = \frac{1}{1+e^{-x}} = \frac{e^x}{1+e^x}$。其导数 $\sigma'(x) = \sigma(x)(1-\sigma(x))$。

具体展开为：

$$\begin{aligned} p(y=1 \mid \boldsymbol{x};\theta_0,\theta_1) &= \frac{\exp(\theta_0+\theta_1\boldsymbol{x})}{1+\exp(\theta_0+\theta_1\boldsymbol{x})} = \pi(\boldsymbol{x};\theta_0,\theta_1) \\ p(y=0 \mid \boldsymbol{x};\theta_0,\theta_1) &= \frac{1}{1+\exp(\theta_0+\theta_1\boldsymbol{x})} = 1-\pi(\boldsymbol{x};\theta_0,\theta_1) \end{aligned} \tag{27}$$

对于二分类问题训练目标, 训练目标为最大似然:

$$\prod_i^m [\pi(\boldsymbol{x}^{(i)};\theta_0,\theta_1)]^{y^{(i)}} [1-\pi(\boldsymbol{x}^{(i)};\theta_0,\theta_1)]^{1-y^{(i)}} \tag{28}$$

最大化似然相当于最小化其负对数似然形式：

$$\mathcal{L}(\theta_0,\theta_1) = -\sum_{i=1}^m \left[y^{(i)} \log \pi(\boldsymbol{x}^{(i)};\theta_0,\theta_1) + (1-y^{(i)}) \log(1-\pi(\boldsymbol{x}^{(i)};\theta_0,\theta_1)) \right] \tag{29}$$

梯度下降法更新参数：

$$
\begin{aligned}
\frac{\partial \mathcal{L}(\theta)}{\partial \theta_j} &= -\sum_{i=1}^{m}(y^{(i)}\frac{1}{\pi(\boldsymbol{x}^{(i)};\theta_0,\theta_1)} - (1-y^{(i)})\frac{1}{1-\pi(\boldsymbol{x}^{(i)};\theta_0,\theta_1)})\frac{\partial \pi(\boldsymbol{x}^{(i)};\theta_0,\theta_1)}{\partial \theta_j} \\
&= -\sum_{i=1}^{m}(y^{(i)}\frac{1}{\pi(\boldsymbol{x}^{(i)};\theta_0,\theta_1)} - (1-y^{(i)})\frac{1}{1-\pi(\boldsymbol{x}^{(i)};\theta_0,\theta_1)})\pi(\boldsymbol{x}^{(i)};\theta_0,\theta_1)(1-\pi(\boldsymbol{x}^{(i)};\theta_0,\theta_1))\frac{\partial \theta^\top \boldsymbol{x}^{(i)}}{\partial \theta_j} \\
&= -\sum_{i=1}^{m}(y^{(i)}(1-\pi(\boldsymbol{x}^{(i)};\theta_0,\theta_1)) - (1-y^{(i)})\pi(\boldsymbol{x}^{(i)};\theta_0,\theta_1))x_j^{(i)} \\
&= -\sum_{i=1}^{m}(y^{(i)} - \pi(\boldsymbol{x}^{(i)};\theta_0,\theta_1))x_j^{(i)}
\end{aligned}
\tag{30}
$$

然后梯度更新 $\theta \leftarrow \theta - \epsilon\frac{\partial \mathcal{L}(\theta)}{\partial \theta}$。$\epsilon$ 为学习率，表示更新的步长。

预测过程：用最优 $\widehat{\theta}$ 预测新的输入 $\boldsymbol{x}$。

$$
\widehat{\pi}(x) = \frac{\exp\left(\widehat{\theta}_0 + \widehat{\theta}_1 \boldsymbol{x}\right)}{1+\exp\left(\widehat{\theta}_0 + \widehat{\theta}_1 \boldsymbol{x}\right)} \tag{31}
$$

逻辑回归的另一种描述：逻辑回归实质上是用线性模型拟合**对数几率** (`log odds`)：

$$
\log\left(\frac{p(y=1)}{1-p(y=1)}\right) = \theta^\top \boldsymbol{x} \tag{32}
$$

自定义实现

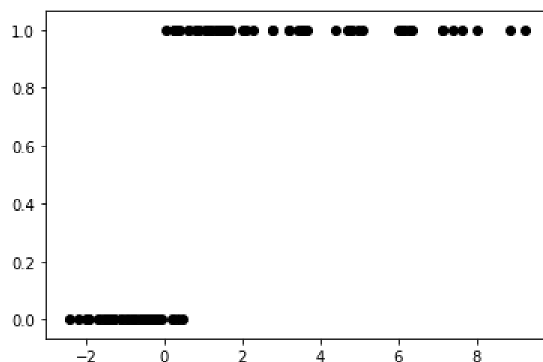
[9]:
```python
def Sigmoid(x):
    return 1/(1 + np.exp(-x))

class LogisticRegression():

    def __init__(self, learning_rate=.1):
        self.param = None
        self.learning_rate = learning_rate
        self.sigmoid = Sigmoid

    def _initialize_parameters(self, X):
        n_features = np.shape(X)[1]
        # 初始化参数 theta， [-1/sqrt(N), 1/sqrt(N)]
        limit = 1 / math.sqrt(n_features)
        self.param = np.random.uniform(-limit, limit, (n_features,))

    def fit(self, X, y, n_iterations=4000):
        self._initialize_parameters(X)
        # 参数 theta 的迭代更新
        for i in range(n_iterations):
            # 求预测
            y_pred = self.sigmoid(X.dot(self.param))
            # 最小化损失函数，参数更新公式
            self.param -= self.learning_rate * -(y - y_pred).dot(X)

    def predict(self, X):
        y_pred = self.sigmoid(X.dot(self.param))
        return y_pred

    def score(self, X, y):
        y_pred = self.predict(X)
        accuracy = np.sum(y == y_pred, axis=0) / len(y)
        return accuracy
```

用自定义逻辑回归，合成数据集测试

[10]:
```python
# 二分类问题，生成数据
n_samples = 100
np.random.seed(0)
x = np.random.normal(size=n_samples)
```

```
y = (x > 0).astype(np.float)
x[x > 0] *= 4
x += .3 * np.random.normal(size=n_samples)
x = x[:, np.newaxis]  # 输入增加一维，用于与 theta_0 结合
plt.scatter(x, y, color='black')
```

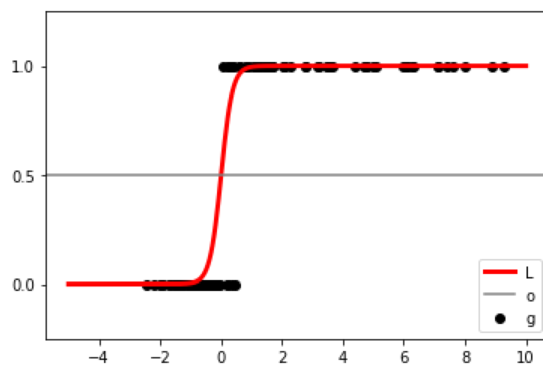
[10]: <matplotlib.collections.PathCollection at 0x113db1ac8>



[11]:
```
# 训练一个逻辑回归模型
model = LogisticRegression()
model.fit(x, y)
```

[12]:
```
# 预测
x_test = np.linspace(-5, 10, 300)
x_test = x_test[:, np.newaxis]
prob = model.predict(x_test).ravel()

plt.plot(x_test, prob, color='red', linewidth=3)
plt.scatter(x, y, color='black');
plt.axhline(0.5, color='0.5');
plt.ylim(-0.25, 1.25);
plt.yticks([0, 0.5, 1]);
plt.legend(('Logistic Regression Model'), loc='lower right')
```

[12]: <matplotlib.legend.Legend at 0x113e21198>

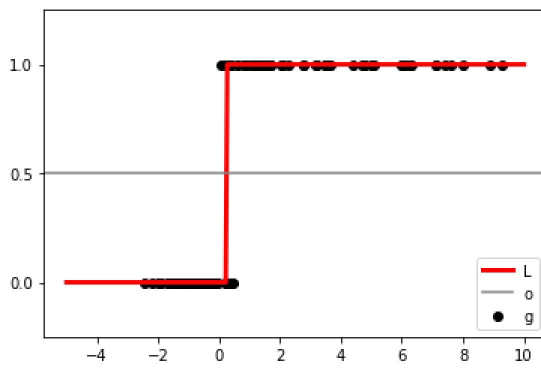


用 sklearn 实现逻辑回归，合成数据集测试

[13]:
```
from sklearn.linear_model import LogisticRegression
# C 表示正则化系数   的倒数， solver 表示优化算法选择参数
skl_model = LogisticRegression()
skl_model.fit(x, y)
# 预测
x_test = np.linspace(-5, 10, 300)
x_test = x_test[:, np.newaxis]
prob = skl_model.predict(x_test).ravel()

plt.plot(x_test, prob, color='red', linewidth=3)
plt.scatter(x, y, color='black');
plt.axhline(0.5, color='0.5');
plt.ylim(-0.25, 1.25);
plt.yticks([0, 0.5, 1]);
```

```
plt.legend(('Logistic Regression Model'), loc='lower right')
```

[13]: <matplotlib.legend.Legend at 0x1143a2f60>



用自定义逻辑回归，*iris* 数据集测试

[14]:
```
def create_data():
    iris = load_iris()
    df = pd.DataFrame(iris.data, columns=iris.feature_names)
    df['label'] = iris.target
    df.columns = ['sepal length', 'sepal width', 'petal length', 'petal width', 'label']
    data = np.array(df.iloc[:100, :])
    return data[:,:-1], data[:,-1]

X, y = create_data()
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3)
print(X_train[0], y_train[0])
```

```
[5.  3.5 1.6 0.6] 0.0
```

[15]:
```
model = LogisticRegression()
model.fit(X_train, y_train)
# 测试数据
print(model.score(X_test, y_test))
```

```
1.0
```

用 sklearn 实现逻辑回归，*iris* 数据集测试

[16]:
```
# 从 sklearn 包中调用 LogisticRegression 测试
from sklearn.linear_model import LogisticRegression
skl_model = LogisticRegression()
skl_model.fit(X_train, y_train)
# 测试数据
print(skl_model.score(X_test, y_test))
```

```
1.0
```

8.2 支持向量机

支持向量机 (Support Vector Machine，SVM) 同样基于线性函数 $\boldsymbol{w}^\top \boldsymbol{x} + b$，不同于逻辑回归，SVM 不输出概率，只输出类别。

SVM 的思想是**找到一个分割线 (或分割平面)，把两个类别的点分得越开越好**。

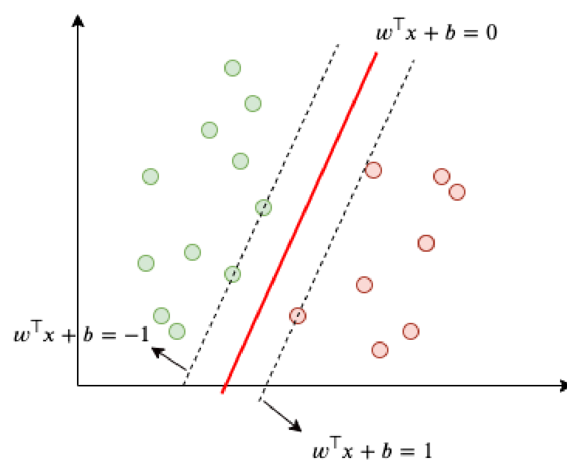


图 3. 支持向量机

在超平面 $\boldsymbol{w}^\top\boldsymbol{x}+b=0$ 确定的情况下，$|\boldsymbol{w}^\top\boldsymbol{x}+b|$ 能够表示点 $\boldsymbol{x}$ 到超平面的距离远近，而通过观察 $\boldsymbol{w}^\top\boldsymbol{x}+b$ 的符号与类型标记 y 符号是否一致，可以判断分类是否正确。于是定义函数间隔 (Functional Margin) 的概念：$\hat{\gamma}=y(\boldsymbol{w}^\top\boldsymbol{x}+b)=yf(\boldsymbol{x})$。**超平面 $(\boldsymbol{w},b)$ 关于训练数据集中所有样本点 $(\boldsymbol{x}^{(i)},y^{(i)})$ 的函数间隔最小值，便成为超平面 $(\boldsymbol{w},b)$ 关于数据集的的函数间隔**：$\hat{\gamma}=\min\hat{\gamma}_i(i=1,\dots m)$。

但成比例的改变 $\boldsymbol{w},b$ (如都变成原来的 2 倍)，则函数间隔 $f(\boldsymbol{x})$ 的值变成了原来的 2 倍，但此时超平面却没有改变。因此引入真正定义**点到超平面的距离–几何间隔** (Geometrical Margin) 的概念：$\tilde{\gamma}=\frac{\hat{\gamma}}{\|\boldsymbol{w}\|}$。

因此，目标函数可以写作：$\max\tilde{\gamma}$。

同时，限定约束条件：$s.t., y^{(i)}(\boldsymbol{w}^\top\boldsymbol{x}^{(i)}+b)=\hat{\gamma}_i\geq\hat{\gamma}, \qquad i=1,...,m$

下面为了和传统写法统一以及方便理解，在本章的剩余内容中，我会将 $(\boldsymbol{x}^{(i)},y^{(i)})$ 写作 $(\boldsymbol{x}_i,y_i)$，将样本 $\boldsymbol{x}$ 的第 j 个特征写作 x^j。如果令函数间隔 $\hat{\gamma}=1$，则有 $\tilde{\gamma}=\frac{1}{\|\boldsymbol{w}\|}$。上述目标函数便转化成：

$$\begin{cases}\max\dfrac{1}{\|\omega\|}\\ s.t., y_i(\omega^\top\boldsymbol{x}_i+b)=\hat{\gamma}_i\geq 1, i=1,...,m\end{cases}\tag{33}$$

该式等价于求解：

$$\begin{cases}\min\dfrac{1}{2}\|\boldsymbol{w}\|^2\\ s.t., y_i(\boldsymbol{w}^\top\boldsymbol{x}_i+b)=\hat{\gamma}_i\geq 1, i=1,...,m\end{cases}\tag{34}$$

由于现在的目标函数是二次的，约束条件是线性的，所以它是一个凸二次规划问题。这个问题可以用现成的 QP (Quadratic Programming) 优化包进行求解。由于这个问题的特殊结构，还可以通过拉格朗日对偶性 (Lagrange Duality) 变换到对偶变量 (dual variable) 的优化问题，即通过求解与原问题等价的对偶问题 (Dual Problem) 得到原始问题的最优解，这就是线性可分条件下支持向量机的对偶算法。其优点在于：

- 对偶问题往往更容易求解。
- 可以自然的引入核函数，进而推广到非线性分类问题。

定义拉格朗日函数如下式：

$$\mathcal{L}(\boldsymbol{w},b,\alpha)=\frac{1}{2}\|\boldsymbol{w}\|^2-\sum_{i=1}^{m}\alpha_i[y_i(\boldsymbol{w}^\top\boldsymbol{x}_i+b)-1]\tag{35}$$

求解这个对偶学习问题，分为三个步骤 (参考第四章 KKT 条件)：

- 令 $\mathcal{L}(\boldsymbol{w},b,\alpha)$ 关于 $\boldsymbol{w}$ 和 b 最小化。
- 利用 SMO 算法求解对偶问题中的拉格朗日乘子，求对 α 的极大。
- 求参数 $\boldsymbol{w}$，b 。

首先固定 α，令 $\mathcal{L}$ 关于 $\boldsymbol{w}$,b 最小化

分别对 $\boldsymbol{w}$，b 求偏导数：

$$\begin{cases}\dfrac{\partial\mathcal{L}}{\partial\boldsymbol{w}}=0\Rightarrow\boldsymbol{w}=\displaystyle\sum_{i=1}^{m}\alpha_i y_i\boldsymbol{x}_i\\ \dfrac{\partial\mathcal{L}}{\partial b}=0\Rightarrow\displaystyle\sum_{i=1}^{m}\alpha_i y_i=0\end{cases}\tag{36}$$

代入原式中得到：

$$\begin{aligned}\mathcal{L}(\boldsymbol{w},b,\alpha)&=\frac{1}{2}\boldsymbol{w}^\top\cdot\sum_{i=1}^m\alpha_i\boldsymbol{x}_iy_i-\boldsymbol{w}^\top\sum_{i=1}^m\alpha_iy_i\boldsymbol{x}_i-b\sum_{i=1}^m\alpha_iy_i+\sum_{i=1}^m\alpha_i\\&=-\frac{1}{2}\boldsymbol{w}^\top\sum_{i=1}^m\alpha_iy_i\boldsymbol{x}_i-b\sum_{i=1}^m\alpha_iy_i+\sum_{i=1}^m\alpha_i\\&=-\frac{1}{2}[\sum_{i=1}^m\alpha_iy_i\boldsymbol{x}_i]^\top\sum_{i=1}^m\alpha_iy_i\boldsymbol{x}_i-b\sum_{i=1}^m\alpha_iy_i+\sum_{i=1}^m\alpha_i\\&=-\frac{1}{2}\sum_{i=1}^m\alpha_iy_i(\boldsymbol{x}_i)^\top\cdot\sum_{i=1}^m\alpha_iy_i\boldsymbol{x}_i-b\sum_{i=1}^m\alpha_iy_i+\sum_{i=1}^m\alpha_i\\&=-\frac{1}{2}\sum_{i,j=1}^m\alpha_i\alpha_jy_iy_j\boldsymbol{x}_i^\top\boldsymbol{x}_j+\sum_{i=1}^m\alpha_i\end{aligned}$$

利用 SMO 算法求解对偶问题中的拉格朗日乘子

经过上一步后得到此时的目标函数：

$$\begin{aligned}&\max_\alpha\sum_{i=1}^m\alpha_i-\frac{1}{2}\sum_{i=1}^m\alpha_i\alpha_jy_iy_j\boldsymbol{x}_i^\top\boldsymbol{x}_j\\&s.t.,\alpha_i\geq0,i=1,...,m\\&\sum_{i=1}^m\alpha_iy_i=0\end{aligned}\tag{38}$$

通过 SMO 算法可以求解对偶问题中的拉格朗日乘子 α。

求参数 $\boldsymbol{w}$, b

上面一步求出了拉格朗日乘子 α，可以计算出：$\boldsymbol{w}^*=\sum_{i=1}^m\alpha_iy_i\boldsymbol{x}_i$。

因为支持向量处于边界点，满足：

$$\begin{cases}\max\limits_{y_i=-1}\boldsymbol{w}^\top\boldsymbol{x}_i+b=-1\\\min\limits_{y_i=1}\boldsymbol{w}^\top\boldsymbol{x}_i+b=1\end{cases}\tag{39}$$

由于对于边界上的支持向量有 $y(\boldsymbol{w}^\top\boldsymbol{x}+b)=1$，可以用支持向量求 b 值：$b^*=y_j-\sum_{i=1}^m\alpha_iy_i\langle\boldsymbol{x}_i,\boldsymbol{x}_j\rangle$。可以看出，$\boldsymbol{w}^*$ 和 b^* 只依赖于数据中 $\alpha>0$ 的样本点。

分类函数：

$$\begin{aligned}f(\boldsymbol{x})&=(\sum_{i=1}^m\alpha_iy_i\boldsymbol{x}_i)^\top\boldsymbol{x}+b\\&=\sum_{i=1}^m\alpha_iy_i\langle\boldsymbol{x}_i,\boldsymbol{x}\rangle+b\end{aligned}\tag{40}$$

8.2.1 核技巧

对于非线性的情况，SVM 的处理方法是选择一个核函数 $k(\cdot,\cdot)$，通过将数据映射到高维空间，来解决在原始空间中线性不可分的问题。其思想是**通过核函数将输入空间映射到高维特征空间，最终在高维特征空间中构造出最优分离超平面，从而把平面上本身不好分的非线性数据分开**。

计算两个向量在隐式映射过后的空间中的内积的函数叫做核函数 (`Kernel Function`)。核函数相当于将原来的分类函数 $f(\boldsymbol{x})=\sum_{i=1}^m\alpha_iy_i\langle\boldsymbol{x}_i,\boldsymbol{x}\rangle+b$ 映射成了 $f(\boldsymbol{x})=\sum_{i=1}^m\alpha_iy_i\langle\phi(\boldsymbol{x}_i),\phi(\boldsymbol{x})\rangle+b$。

其中的 α 值是可以通过求解由映射变形来的对偶问题得到：

$$\begin{aligned}&\max_\alpha\sum_{i=1}^m\alpha_i-\frac{1}{2}\sum_{i=1}^m\alpha_i\alpha_jy_iy_j\langle\phi(\boldsymbol{x}_i),\phi(\boldsymbol{x}_j)\rangle\\&s.t.,\alpha_i\geq0,i=1,...,m\\&\sum_{i=1}^m\alpha_iy_i=0\end{aligned}\tag{41}$$

这里的核函数描述为：对于所有的数据点 $\boldsymbol{x},\boldsymbol{z}$，满足 $k(\boldsymbol{x},\boldsymbol{z})=\langle\phi(\boldsymbol{x})\cdot\phi(\boldsymbol{z})\rangle$。**核函数的作用等同于用特征函数 $\phi(x)$ 预处理所有输入，然后在新的转换空间学习线性模型**。

常用核函数

线性核 (`Linear kernel`): $k(u,v)=\langle u,v\rangle$

多项式核 (`Polynomial kernel`): $k(u,v)=(1+\langle u,v\rangle)^d$

高斯核 (`RBF kernel`): $k(u,v)=N(u-v;0,\sigma^2I)=\exp\left(-\frac{||u-v||^2}{2\sigma^2}\right)$

其中 $N(x;\mu,\Sigma)$ 是标准正态密度，也被称为径向基函数。

自定义实现

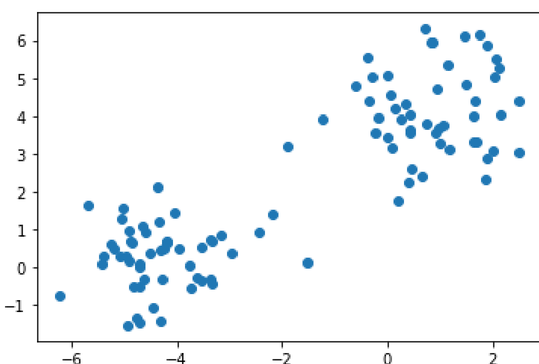
[17]:
```
import cvxopt

# 隐藏 cvxopt 输出
cvxopt.solvers.options['show_progress'] = False

def linear_kernel(**kwargs):
    """
    线性核
    """
    def f(x1, x2):
        return np.inner(x1, x2)
    return f

def polynomial_kernel(power, coef, **kwargs):
    """
    多项式核
    """
    def f(x1, x2):
        return (np.inner(x1, x2) + coef)**power
    return f

def rbf_kernel(gamma, **kwargs):
    """
    高斯核
    """
    def f(x1, x2):
        distance = np.linalg.norm(x1 - x2) ** 2
        return np.exp(-gamma * distance)
    return f

class SupportVectorMachine():

    def __init__(self, kernel=linear_kernel, power=4, gamma=None, coef=4):
        self.kernel = kernel
        self.power = power
        self.gamma = gamma
        self.coef = coef
        self.lagr_multipliers = None
        self.support_vectors = None
        self.support_vector_labels = None
        self.intercept = None

    def fit(self, X, y):
        n_samples, n_features = np.shape(X)
        # gamma 默认设置为 1 / n_features
        if not self.gamma:
            self.gamma = 1 / n_features
        # 定义核函数
        self.kernel = self.kernel(
            power=self.power,
            gamma=self.gamma,
            coef=self.coef)
        # 计算 Gram 矩阵
        kernel_matrix = np.zeros((n_samples, n_samples))
        for i in range(n_samples):
            for j in range(n_samples):
                kernel_matrix[i, j] = self.kernel(X[i], X[j])
        # 构造二次规划问题
        # 形式为 min (1/2)x.T*P*x+q.T*x, s.t. G*x<=h, A*x=b
```

```
        P = cvxopt.matrix(np.outer(y, y) * kernel_matrix, tc='d')
        q = cvxopt.matrix(np.ones(n_samples) * -1)
        A = cvxopt.matrix(y, (1, n_samples), tc='d')
        b = cvxopt.matrix(0, tc='d')
        G = cvxopt.matrix(np.identity(n_samples) * -1)
        h = cvxopt.matrix(np.zeros(n_samples))
        # 用 cvxopt 求解二次规划问题
        minimization = cvxopt.solvers.qp(P, q, G, h, A, b)
        lagr_mult = np.ravel(minimization['x'])
        # 非 0 的 alpha 值
        idx = lagr_mult > 1e-7
        # alpha 值
        self.lagr_multipliers = lagr_mult[idx]
        # 支持向量
        self.support_vectors = X[idx]
        # 支持向量的标签
        self.support_vector_labels = y[idx]
        # 通过第一个支持向量计算 b
        self.intercept = self.support_vector_labels[0]
        for i in range(len(self.lagr_multipliers)):
            self.intercept -= self.lagr_multipliers[i] * self.support_vector_labels[
                i] * self.kernel(self.support_vectors[i], self.support_vectors[0])

    def predict(self, X):
        y_pred = []
        for sample in X:
            # 对于输入的 x, 计算 f(x)
            prediction = 0
            for i in range(len(self.lagr_multipliers)):
                prediction += self.lagr_multipliers[i] * self.support_vector_labels[
                    i] * self.kernel(self.support_vectors[i], sample)
            prediction += self.intercept
            y_pred.append(np.sign(prediction))
        return np.array(y_pred)

    def score(self, X, y):
        y_pred = self.predict(X)
        accuracy = np.sum(y == y_pred, axis=0) / len(y)
        return accuracy
```

用自定义的支持向量机，合成数据集测试

[18]:
```
from sklearn import datasets
# 测试用例
X, y = datasets.make_blobs(n_samples=100, centers=2, random_state=3)
y[y == 0] = -1
y[y == 1] = 1
plt.scatter(X[:,0], X[:,1])
```

[18]: <matplotlib.collections.PathCollection at 0x114594a58>



[19]:
```
model = SupportVectorMachine()
model.fit(X, y)
print(model.predict([np.array([-0.4, -0.5])]))
print(model.predict([np.array([2.6, 5.3])]))
```

```
[1.]
[-1.]
```

用 sklearn 实现支持向量机，合成数据集测试

[20]:
```
from sklearn import svm
skl_model = svm.SVC(kernel='linear', C=1000)
skl_model.fit(X, y)
# 测试数据
print(skl_model.predict([[-0.4, -0.5]]))
print(skl_model.predict([[2.6, 5.3]]))
```

```
[1]
[-1]
```

用自定义的支持向量机，*iris* 数据集测试

[21]:
```
def create_data():
    iris = load_iris()
    df = pd.DataFrame(iris.data, columns=iris.feature_names)
    df['label'] = iris.target
    df.columns = ['sepal length', 'sepal width', 'petal length', 'petal width', 'label']
    data = np.array(df.iloc[:100, :])
    return data[:,:-1], data[:,-1]

X, y = create_data()
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3)
print(X_train[0], y_train[0])
y_train[y_train == 0] = -1  # 标签设为 1 和-1
print(X_train[0], y_train[0])
```

```
[6.1 2.9 4.7 1.4] 1.0
[6.1 2.9 4.7 1.4] 1.0
```

[22]:
```
model = SupportVectorMachine()
model.fit(X_train, y_train)
# 测试数据
print(model.score(X_test, y_test))
```

```
0.43333333333333335
```

用 sklearn 实现支持向量机，*iris* 数据集测试

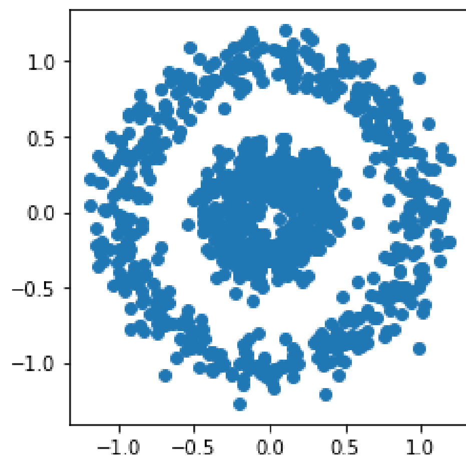
[23]:
```
from sklearn import svm
skl_model = svm.SVC(kernel='linear', C=1000)
skl_model.fit(X_train, y_train)
# 测试数据
print(skl_model.score(X_test, y_test))
```

```
0.43333333333333335
```

用 sklearn 展示核函数

[24]:
```
from sklearn import svm
from sklearn.model_selection import cross_val_score
X, y = datasets.make_circles(n_samples=1000, factor=0.3, noise=0.1, random_state=2019)
plt.subplot(111, aspect='equal')
plt.scatter(X[:,0], X[:,1])
```

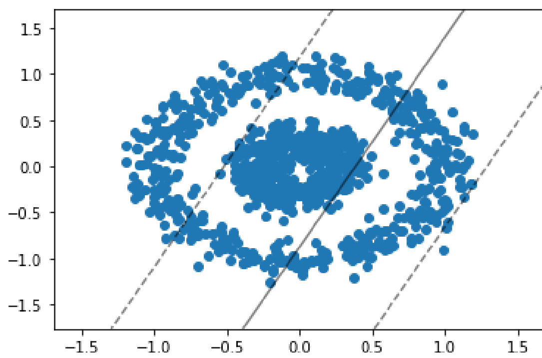
[24]:
```
<matplotlib.collections.PathCollection at 0x1146513c8>
```



[25]:
```
xx = np.linspace(X[:,0].min()-0.5, X[:,0].max()+0.5, 30)
yy = np.linspace(X[:,1].min()-0.5, X[:,1].max()+0.5, 30)
YY, XX = np.meshgrid(yy, xx)
xy = np.vstack([XX.ravel(), YY.ravel()]).T
```

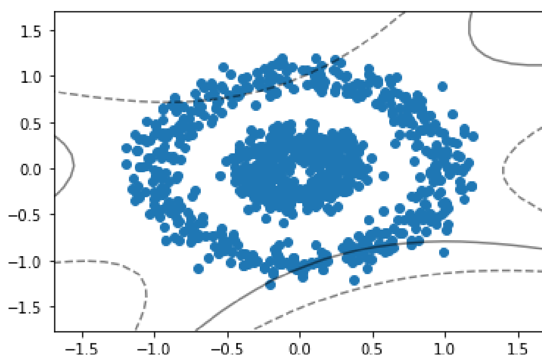
[26]:
```
# 线性核
clf = svm.SVC(kernel='linear', C=1000)
clf.fit(X,y)
Z = clf.decision_function(xy).reshape(XX.shape)
# 画决策边界和间隔
plt.contour(XX, YY, Z, colors='k', levels=[-1, 0, 1], alpha=0.5, linestyles=['--', '-', '--']);
plt.scatter(X[:,0], X[:,1]);
print('10-fold cv scores with linear kernel: ', np.mean(cross_val_score(clf, X, y, cv=10)))
```

```
10-fold cv scores with linear kernel:  0.638
```



[27]:
```
# 多项式核
clf = svm.SVC(kernel='poly', gamma='auto')
clf.fit(X,y)
Z = clf.decision_function(xy).reshape(XX.shape)
# 画决策边界和间隔
plt.contour(XX, YY, Z, colors='k', levels=[-1, 0, 1], alpha=0.5, linestyles=['--', '-', '--'])
plt.scatter(X[:,0], X[:,1]);
print('10-fold cv scores with Polynomial kernel: ', np.mean(cross_val_score(clf, X, y, cv=10)))
```

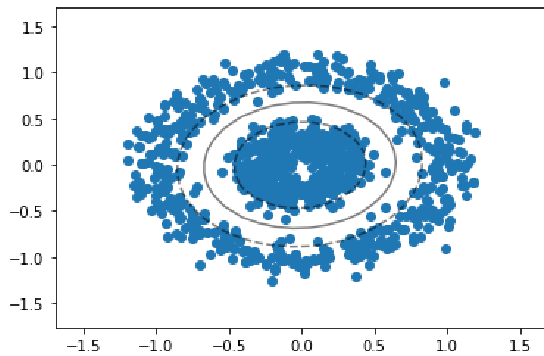
```
10-fold cv scores with Polynomial kernel:  0.5290000000000001
```



[28]:
```
# RBF 高斯核
clf = svm.SVC(kernel='rbf', gamma='auto')
clf.fit(X,y)
Z = clf.decision_function(xy).reshape(XX.shape)
```

```
# 画决策边界和间隔
plt.contour(XX, YY, Z, colors='k', levels=[-1, 0, 1], alpha=0.5, linestyles=['--', '-', '--'])
plt.scatter(X[:,0], X[:,1]);
print('10-fold cv scores with RBF kernel: ', np.mean(cross_val_score(clf, X, y, cv=10)))
```

```
10-fold cv scores with RBF kernel:  1.0
```



8.3 k-近邻

k-近邻 (k-Nearest Neighbors) 的思想是给定测试样本，基于某种距离度量 (一般使用欧几里德距离) 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 (**“物以类聚”**)。

算法步骤：

- 根据给定的距离度量，在训练集中找出与 $\boldsymbol{x}$ 最近邻的 k 个点，涵盖这 k 个点的 $\boldsymbol{x}$ 的邻域记作 $\mathcal{N}_k(\boldsymbol{x})$。
- 在 $\mathcal{N}_k(\boldsymbol{x})$ 中根据分类决策规则，如多数表决决定 $\boldsymbol{x}$ 的类别 y。

可见，决定了 k 近邻模型的三个基本要素——距离度量、k 值的选择、分类决策规则。

距离度量

- 一般使用欧几里德距离
- 相关度 (如皮尔逊相关系数)
- 曼哈顿距离 (Manhattan Distance)

k 值的选择

- 当选择比较小的 k 值的时候，表示使用较小领域中的样本进行预测，训练误差会减小，但是会导致模型变得复杂，容易导致过拟合。
- 当选择较大的 k 值的时候，表示使用较大领域中的样本进行预测，训练误差会增大，同时会使模型变得简单，容易导致欠拟合。

分类决策规则

- **多数表决法**：每个邻近样本的权重是一样的，最终预测的结果为出现类别最多的那个类。
- **加权多数表决法**：每个邻近样本的权重是不一样的，一般情况下采用权重和距离成反比的方式来计算，最终预测结果是出现权重最大的那个类别。

自定义实现

[29]:
```
class KNN():

    def __init__(self, k=10):
        self._k = k

    def fit(self, X, y):
        self._unique_labels = np.unique(y)
        self._class_num = len(self._unique_labels)
        self._datas = X
        self._labels = y.astype(np.int32)

    def predict(self, X):
        # 欧式距离计算
        dist = np.sum(np.square(X), axis=1, keepdims=True) - 2 * np.dot(X, self._datas.T)
        dist = dist + np.sum(np.square(self._datas), axis=1, keepdims=True).T
        dist = np.argsort(dist)[:,:self._k]
        return np.array([np.argmax(np.bincount(self._labels[dist][i])) for i in range(len(X))])
```

```
    def score(self, X, y):
        y_pred = self.predict(X)
        accuracy = np.sum(y == y_pred, axis=0) / len(y)
        return accuracy
```

用自定义的 k-近邻，$iris$ 数据集测试

[30]:
```
def create_data():
    iris = load_iris()
    df = pd.DataFrame(iris.data, columns=iris.feature_names)
    df['label'] = iris.target
    df.columns = ['sepal length', 'sepal width', 'petal length', 'petal width', 'label']
    data = np.array(df.iloc[:100, :])
    return data[:,:-1], data[:,-1]

X, y = create_data()
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3)
print(X_train[0], y_train[0])
```

```
[7.  3.2 4.7 1.4] 1.0
```

[31]:
```
model = KNN()
model.fit(X_train, y_train)
# 测试数据
print(model.score(X_test, y_test))
```

```
1.0
```

用 sklearn 实现 k-近邻，$iris$ 数据集测试

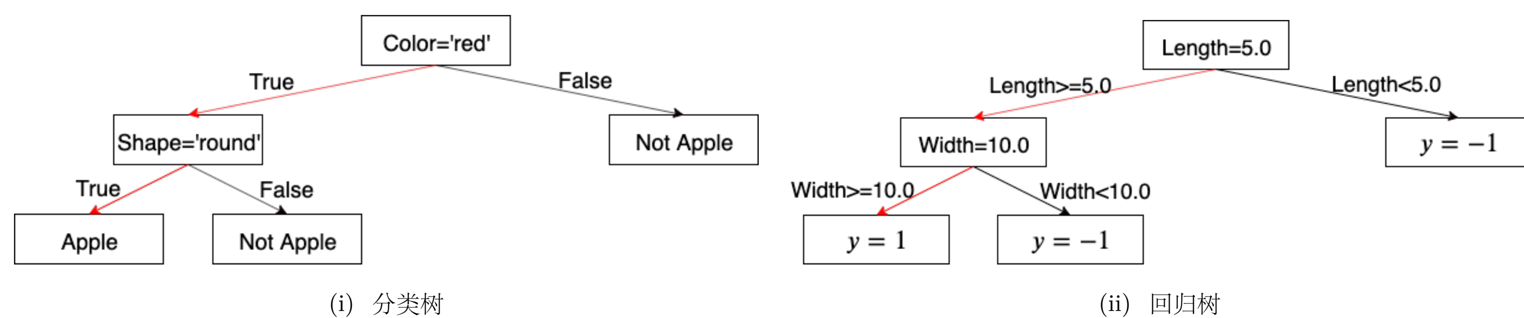
[32]:
```
# 从 sklearn 包中调用 neighbors 测试
from sklearn import neighbors
skl_model = neighbors.KNeighborsClassifier()
# 训练数据集
skl_model.fit(X_train, y_train)
# 测试数据
print(skl_model.score(X_test, y_test))
```

```
1.0
```

8.4 决策树

决策树 (Decision Tree) 由节点和有向边组成，一般一棵决策树包含一个根节点、若干内部节点和若干叶节点。**决策树的决策过程需要从决策树的根节点开始，待测数据与决策树中的特征节点进行比较，并按照比较结果选择选择下一比较分支，直到叶子节点作为最终的决策结果。**

目标变量采用一组离散值的决策树称为**分类树** (Classification Tree)(如下图左，常用的分类树算法有 ID3、C4.5、CART)，而目标变量采用连续值 (通常是实数) 的决策树被称为**回归树** (Regression Tree)(如下图右，常用的回归树算法有 CART)。



(i) 分类树 (ii) 回归树

图 4. 决策树

假设先不考虑决策树剪枝，决策树算法的核心问题在于特征选择和决策树生成。

8.4.1 特征选择

特征选择即**选择最优划分属性**，从当前数据的特征中选择一个特征作为当前节点的划分标准。我们希望在不断划分的过程中，决策树的分支节点所包含的样本尽可能属于同一类，即节点的“纯度”(Impurity) 越来越高。而选择最优划分特征的标准不同，也导致了决策树算法的不同。常见的方

法基于以下几种：

- 信息增益 (Information Gain)
- 信息增益率 (Information Gain Ratio)
- 基尼指数 (Gini Index)

信息增益:

划分标准主要基于信息论。在一个有 K 个类别的训练数据 D 中，假设输出类别的概率分布为：

$$P(y=k)=p_k \tag{42}$$

那么具有 K 个类别的样本的信息熵为：

$$H(D)=-\sum_{k=1}^{K}p_k\log p_k \tag{43}$$

可以看出，当整个训练数据集只有一个类别时，熵最低，为 $H_{min}(D)=-1\cdot log1=0$；当训练数据集各类别为均匀分布时，熵最大，为 $H_{max}(D)=-K\cdot\frac{1}{K}\cdot\log\frac{1}{K}=-\log K$。现在假设某一离散特征 $\boldsymbol{X}_j$ (表示取所有样本 $\boldsymbol{X}$ 的第 j 个特征) 有 V 个不同的的取值，那么当以特征 x_j 来划分时可以将数据集 D 分为 V 个子集：

$$D=\sum_{v=1}^{V}D_v \tag{44}$$

那么划分之后的 V 个子数据集的加权熵为：

$$H(D\mid\boldsymbol{X}_j)=\sum_{v=1}^{V}\frac{|D_v|}{|D|}H(D_v) \tag{45}$$

当按特征 $\boldsymbol{X}_j$ 来划分数据集时的信息增益 (Information Gain) 定义为：

$$G(D,\boldsymbol{X}_j)=H(D)-H(D\mid\boldsymbol{X}_j) \tag{46}$$

信息增益表示得知特征 $\boldsymbol{X}_j$ 的信息情况下使得预测类别的信息的不确定性减少程度。

信息增益有个缺点，就是会倾向于选择类别数多的特征来做划分。假设有一列特征 (例如样本 ID) 类别数与样本数相等，如果以该特征来进行划分数据集，则数据集被划分成了单样本节点，每个节点的熵均为 0，总熵也为 0。如此一来，就得到了最大的信息增益，但是这种划分显然是不合理的。

信息增益率:

为解决信息增益的缺点，而是使用信息增益率 (Information Gain Ratio) 来决定使用哪个特征来划分数据集。其定义为：

$$GR(D,\boldsymbol{X}_j)=\frac{G(D,\boldsymbol{X}_j)}{IV(\boldsymbol{X}_j)} \tag{47}$$

其中 $IV(\boldsymbol{X}_j)$ 被称为固有值 (Intrinsic Value)，它等价于这个特征在数据集中的熵。

具体来说，假设考虑数据集 D 中的特征 $\boldsymbol{X}_j$，它有 V 个不同的取值 $s_1,\cdots,s_V$，那么特征 $\boldsymbol{X}_j$ 在数据集中的概率分布为：

$$P(\boldsymbol{X}_j=s_v)=\frac{|D_v|}{|D|}=p_v \tag{48}$$

那么数据集中该特征的固有值 (熵) 为：

$$IV(\boldsymbol{X}_j)=-\sum_{v=1}^{V}p_v\log p_v \tag{49}$$

基尼指数:

对于有 K 个类别的数据集 D，某一样本属于类别 k 的概率等于该类别的分布概率：

$$P(y=k)=p_k \tag{50}$$

那么数据集 D 的基尼指数定义如下：

$$Gini(D)=1-\sum_{k=1}^{K}p_k^2 \tag{51}$$

由公式可以看出，基尼指数表示的是从数据集中随机取两个样本类别不同的概率，其值越小则数据集纯度越高。

如对一个数据集 D 以特征 $\boldsymbol{X}_j$ 的一个取值 s_v 来划分，那么数据集会被划分成 $D_{\boldsymbol{X}_j=s_v}$ 和 $D_{\boldsymbol{X}_j\neq s_v}$，那么数据集 D 依照特征 $\boldsymbol{X}_j=s_v$ 划分之后的加权基尼指数为：

$$Gini(D\mid\boldsymbol{X}_j=s_v)=\frac{|D_{\boldsymbol{X}_j=s_v}|}{|D|}Gini(D_{\boldsymbol{X}_j=s_v})+\frac{|D_{\boldsymbol{X}_j\neq s_v}|}{|D|}Gini(D_{\boldsymbol{X}_j\neq s_v}) \tag{52}$$

8.4.2 决策树生成

生成算法	划分标准
ID3	信息增益
C4.5	信息增益率
CART	基尼指数

ID3 算法：

ID3 算法的核心是**在决策树各个节点上根据信息增益来选择进行划分的特征，然后递归地构建决策树**。

算法思路：首先，树的根节点中包含整个数据集，ID3 算法会遍历所有特征分别来计算划分后的信息增益，选择信息增益最大的特征；然后由该特征的不同取值建立子节点，将数据集划分到新一层的各个子节点中；对各个子节点中的数据递归进行这个过程，直到信息增益足够小或者无特征可用，最终得到决策树。

CART 算法：

分类树

CART 分类树选取特征的依据是基尼指数，在划分时会遍历所有的特征与其所有可能的取值，再全局考量选取一个最佳特征与最佳划分点。若数据集有 n 个特征，每个特征 $\boldsymbol{X}_j$ 有 V_j 种不同取值，CART 分类树可以用下式来表述：

$$(\boldsymbol{X}_*, s_*) = \arg\min\left(G(D \mid \boldsymbol{X}_j = s_v)\right), \qquad j\, from\, 1 \to n, v\, from\, 1 \to V_j \tag{53}$$

回归树

首先需要说明的是前面分类树的输出是叶子节点所有样本目标值的多数表决，回归树的输出是叶子节点中所有样本目标值的均值。那么对于回归任务，如何生成树？可以采用一种直观的方式来对数据集进行划分，假设某一时刻数据集 (数据子集) D 被决策树以特征 $\boldsymbol{X}_j$ 按取值 s 划分成了两部分 (或两个叶子节点)：

$$\begin{aligned} \mathrm{R}_1(\boldsymbol{X}_j, s) &= \{\boldsymbol{X} \mid \boldsymbol{X}_j < s\} \\ \mathrm{R}_2(\boldsymbol{X}_j, s) &= \{\boldsymbol{X} \mid \boldsymbol{X}_j \geq s\} \end{aligned} \tag{54}$$

则此次划分的优劣可以用 MSE 判断，用真实值和划分区域的预测值的最小二乘来衡量：

$$\mathrm{MSE}(\boldsymbol{X}_j, s) = \sum_{\mathrm{R}_1}(y_i - \bar{y}_{\mathrm{R}_1})^2 + \sum_{\mathrm{R}_2}(y_i - \bar{y}_{\mathrm{R}_2})^2 \tag{55}$$

其中，$\hat{y}_{\mathrm{R}_1}$ 为 R_1 区域所有样本目标值的均值，$\hat{y}_{\mathrm{R}_2}$ 为 R_2 区域所有样本目标值的均值。

在生成回归树时，使用贪心策略，遍历所有特征下所有可能的取值，找到一个最优分割点，然后以此类推。决策树在做预测时，首先将测试样本输入决策树进行判定，判定该测试样本属于哪一个叶子节点，然后把该叶子结点内所有训练样本的目标均值作为测试样本的预测值。

除了使用 MSE 作为分裂依据之外，还有一个指标可以用作回归树的分裂：方差 (`Variance`)。当使用方差作为分裂依据时，生成树的目的是希望子节点内的值越稳定越好。

回归同样是考虑分割前后使得指标变化最大的特征 $\boldsymbol{X}_j$ 及特征取值 s。

8.4.3 决策树正则化

决策树算法的缺点在于极易过拟合，所以控制决策树的模型复杂度以防止过拟合是很有必要的。

首先可以设定几个参数抑制树的生长：最大的树深、分割的最少样本数、分割的最小纯度。除此之外，也可以对树的生长不做限制，然后再对树进行剪枝。CART 便使用 Cost-Complexity 剪枝方法。

我们用 T 表示一棵树，$\mathcal{L}(T)$ 表示树 T 的分类 (回归) 误差。设 α 为正则化系数，$\Omega(T)$ 是能表示树结构复杂度的函数。于是 Cost-Complexity 函数：

$$J_\alpha(T) = \mathcal{L}(T) + \alpha\Omega(T) \tag{56}$$

令树 T 的结构复杂度函数等于树的叶节点数：

$$\Omega(T) = |T| \tag{57}$$

以及树 T 的误差函数为：

$$\mathcal{L}(T) = \sum_{i=1}^{|T|} err(t_i)p(t_i) \tag{58}$$

其中 t_i 为树 T 中的第 i 个叶节点，$err(t_i)$ 为该叶节点的分类误差率，$p(t_i)$ 为该叶节点的样本比例。所以这实质上是一个加权误差率。对一个确定的 α 值，一定会有一颗最小化 J_α 的树 T_α。为了找到这颗最优剪枝树 T_α，使用最弱连接剪枝 (`Weakest Link Pruning`) 策略，自底向上地对非叶节点进行剪枝并查看效果，然后选取一个表现最好的 T_α。

假设某一时刻以节点 t 进行剪枝，那么剪枝后与剪枝前的 Cost-Complexity 函数差为：

$$\begin{aligned}\Delta J_\alpha(t) &= J_\alpha(T-T_t) - J_\alpha(T) \\ &= \mathcal{L}(T-T_t) - \mathcal{L}(T) + \alpha(|T-T_t| - |T|) \\ &= (-\mathcal{L}(T_t) + err(t)) + \alpha(-|T_t| + 1) \\ &= err(t) - \mathcal{L}(T_t) + \alpha(1 - |T_t|)\end{aligned} \tag{59}$$

其中，T_t 为树 T 中以节点 t 为根节点的子树。令 $\Delta J_\alpha(t) = 0$，得到 $g(t) = \alpha' = \frac{err(t)-\mathcal{L}(T_t)}{|T_t - 1|}$。

于是算法流程如下：

1. 生成一颗完整树 T^0，对所有的非叶节点都进行剪枝尝试，找到一个最小化 $g(t_1)$ 的剪枝节点 t_1，令 $\alpha^1 = g(t_1)$，$T^1 = T^0 - T_{t_1}$。
2. 对 T^1 所有的非叶节点都进行剪枝尝试，找到一个最小化 $g(t_2)$ 的剪枝节点 t_2，令 $\alpha^2 = g(t_2)$，$T^2 = T^1 - T_{t_2}$。
3. 依次进行下去，直到只剩下一个根节点为止，那么可以得到一个子树序列 $[T^0, T^1, ..., root]$ 和一系列参数 $[\alpha^1, \alpha^2, ...]$，然后在所有子树上使用交叉验证来选取一个最佳参数 $\hat{\alpha}$ 与最佳剪枝树 $T_{\hat{\alpha}}$。

自定义实现（主要基于 CART）

[33]:
```
class DecisionNode():

    def __init__(self, feature_i=None, threshold=None,
                 value=None, true_branch=None, false_branch=None):
        self.feature_i = feature_i          # 当前结点测试的特征的索引
        self.threshold = threshold          # 当前结点测试的特征的阈值
        self.value = value                  # 结点值（如果结点为叶子结点）
        self.true_branch = true_branch      # 左子树（满足阈值， 将特征值大于等于切分点值的数据划分为左子树）
        self.false_branch = false_branch    # 右子树（未满足阈值， 将特征值小于切分点值的数据划分为右子树）


def divide_on_feature(X, feature_i, threshold):
    """
    依据切分变量和切分点，将数据集分为两个子区域
    """
    split_func = None
    if isinstance(threshold, int) or isinstance(threshold, float):
        split_func = lambda sample: sample[feature_i] >= threshold
    else:
        split_func = lambda sample: sample[feature_i] == threshold
    X_1 = np.array([sample for sample in X if split_func(sample)])
    X_2 = np.array([sample for sample in X if not split_func(sample)])
    return np.array([X_1, X_2])


class DecisionTree(object):

    def __init__(self, min_samples_split=2, min_impurity=1e-7,
                 max_depth=float("inf"), loss=None):
        self.root = None  # 根结点
        self.min_samples_split = min_samples_split  # 满足切分的最少样本数
        self.min_impurity = min_impurity  # 满足切分的最小纯度
        self.max_depth = max_depth  # 树的最大深度
        self._impurity_calculation = None  # 计算纯度的函数，如对于分类树采用信息增益
        self._leaf_value_calculation = None  # 计算 y 在叶子结点值的函数
        self.one_dim = None  # y 是否为 one-hot 编码

    def fit(self, X, y):
        self.one_dim = len(np.shape(y)) == 1
        self.root = self._build_tree(X, y)

    def _build_tree(self, X, y, current_depth=0):
        """
        递归方法建立决策树
```

```
        """
        largest_impurity = 0
        best_criteria = None    # 当前最优分类的特征索引和阈值
        best_sets = None        # 数据子集
        if len(np.shape(y)) == 1:
            y = np.expand_dims(y, axis=1)
        Xy = np.concatenate((X, y), axis=1)
        n_samples, n_features = np.shape(X)
        if n_samples >= self.min_samples_split and current_depth <= self.max_depth:
            # 对每个特征计算纯度
            for feature_i in range(n_features):
                feature_values = np.expand_dims(X[:, feature_i], axis=1)
                unique_values = np.unique(feature_values)
                # 遍历特征 i 所有的可能值找到最优纯度
                for threshold in unique_values:
                    # 基于 X 在特征 i 处是否满足阈值来划分 X 和 y, Xy1 为满足阈值的子集
                    Xy1, Xy2 = divide_on_feature(Xy, feature_i, threshold)
                    if len(Xy1) > 0 and len(Xy2) > 0:
                        # 取出 Xy 中 y 的集合
                        y1 = Xy1[:, n_features:]
                        y2 = Xy2[:, n_features:]
                        # 计算纯度
                        impurity = self._impurity_calculation(y, y1, y2)
                        # 如果纯度更高，则更新
                        if impurity > largest_impurity:
                            largest_impurity = impurity
                            best_criteria = {"feature_i": feature_i, "threshold": threshold}
                            best_sets = {
                                "leftX": Xy1[:, :n_features],   # X 的左子树
                                "lefty": Xy1[:, n_features:],   # y 的左子树
                                "rightX": Xy2[:, :n_features],  # X 的右子树
                                "righty": Xy2[:, n_features:]   # y 的右子树
                                }

        if largest_impurity > self.min_impurity:
            # 建立左子树和右子树
            true_branch = self._build_tree(best_sets["leftX"], best_sets["lefty"], current_depth + 1)
            false_branch = self._build_tree(best_sets["rightX"], best_sets["righty"], current_depth + 1)
            return DecisionNode(feature_i=best_criteria["feature_i"], threshold=best_criteria[
                                "threshold"], true_branch=true_branch, false_branch=false_branch)
        # 如果是叶结点则计算值
        leaf_value = self._leaf_value_calculation(y)
        return DecisionNode(value=leaf_value)

    def predict_value(self, x, tree=None):
        """
        预测样本，沿着树递归搜索
        """
        # 根结点
        if tree is None:
            tree = self.root
        # 递归出口
        if tree.value is not None:
            return tree.value
        # 选择当前结点的特征
        feature_value = x[tree.feature_i]
        branch = tree.false_branch
        if isinstance(feature_value, int) or isinstance(feature_value, float):
            if feature_value >= tree.threshold:
```

```
                branch = tree.true_branch
        elif feature_value == tree.threshold:
            branch = tree.true_branch
        return self.predict_value(x, branch)

    def predict(self, X):
        y_pred = [self.predict_value(sample) for sample in X]
        return y_pred

    def score(self, X, y):
        y_pred = self.predict(X)
        accuracy = np.sum(y == y_pred, axis=0) / len(y)
        return accuracy

    def print_tree(self, tree=None, indent=" "):
        """
        输出树
        """
        if not tree:
            tree = self.root
        if tree.value is not None:
            print(tree.value)
        else:
            print("feature|threshold -> %s | %s" % (tree.feature_i, tree.threshold))
            print("%sT->" % (indent), end="")
            self.print_tree(tree.true_branch, indent + indent)
            print("%sF->" % (indent), end="")
            self.print_tree(tree.false_branch, indent + indent)
```

[34]:
```
def calculate_entropy(y):
    log2 = lambda x: math.log(x) / math.log(2)
    unique_labels = np.unique(y)
    entropy = 0
    for label in unique_labels:
        count = len(y[y == label])
        p = count / len(y)
        entropy += -p * log2(p)
    return entropy

def calculate_gini(y):
    unique_labels = np.unique(y)
    var = 0
    for label in unique_labels:
        count = len(y[y == label])
        p = count / len(y)
        var += p ** 2
    return 1 - var

class ClassificationTree(DecisionTree):
    """
    分类树，在决策树节点选择计算信息增益/基尼指数，在叶子节点选择多数表决。
    """
    def _calculate_gini_index(self, y, y1, y2):
        """
        计算基尼指数
        """
        p = len(y1) / len(y)
        gini = calculate_gini(y)
        gini_index = gini - p * \
```

```
            calculate_gini(y1) - (1 - p) * \
            calculate_gini(y2)
        return gini_index

    def _calculate_information_gain(self, y, y1, y2):
        """
        计算信息增益
        """
        p = len(y1) / len(y)
        entropy = calculate_entropy(y)
        info_gain = entropy - p * \
            calculate_entropy(y1) - (1 - p) * \
            calculate_entropy(y2)
        return info_gain

    def _majority_vote(self, y):
        """
        多数表决
        """
        most_common = None
        max_count = 0
        for label in np.unique(y):
            count = len(y[y == label])
            if count > max_count:
                most_common = label
                max_count = count
        return most_common

    def fit(self, X, y):
        self._impurity_calculation = self._calculate_gini_index
        self._leaf_value_calculation = self._majority_vote
        super(ClassificationTree, self).fit(X, y)

def calculate_mse(y):
    return np.mean((y - np.mean(y)) ** 2)

def calculate_variance(y):
    n_samples = np.shape(y)[0]
    variance = (1 / n_samples) * np.diag((y - np.mean(y)).T.dot(y - np.mean(y)))
    return variance


class RegressionTree(DecisionTree):
    """
    回归树，在决策树节点选择计算 MSE/方差降低，在叶子节点选择均值。
    """
    def _calculate_mse(self, y, y1, y2):
        """
        计算 MSE 降低
        """
        mse_tot = calculate_mse(y)
        mse_1 = calculate_mse(y1)
        mse_2 = calculate_mse(y2)
        frac_1 = len(y1) / len(y)
        frac_2 = len(y2) / len(y)
        mse_reduction = mse_tot - (frac_1 * mse_1 + frac_2 * mse_2)
        return mse_reduction

    def _calculate_variance_reduction(self, y, y1, y2):
```

```
        """
        计算方差降低
        """
        var_tot = calculate_variance(y)
        var_1 = calculate_variance(y1)
        var_2 = calculate_variance(y2)
        frac_1 = len(y1) / len(y)
        frac_2 = len(y2) / len(y)
        variance_reduction = var_tot - (frac_1 * var_1 + frac_2 * var_2)
        return sum(variance_reduction)

    def _mean_of_y(self, y):
        """
        计算均值
        """
        value = np.mean(y, axis=0)
        return value if len(value) > 1 else value[0]

    def fit(self, X, y):
        self._impurity_calculation = self._calculate_mse
        self._leaf_value_calculation = self._mean_of_y
        super(RegressionTree, self).fit(X, y)
```

用自定义的分类树，$iris$ 数据集测试

[36]:
```
def create_data():
    iris = load_iris()
    df = pd.DataFrame(iris.data, columns=iris.feature_names)
    df['label'] = iris.target
    df.columns = ['sepal length', 'sepal width', 'petal length', 'petal width', 'label']
    data = np.array(df.iloc[:100, :])
    return data[:,:-1], data[:,-1]

X, y = create_data()
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3)
print(X_train[0], y_train[0])
```

```
[5.  3.6 1.4 0.2] 0.0
```

[37]:
```
# 分类树
model = ClassificationTree()
model.fit(X_train, y_train)
# 测试数据
print(model.score(X_test, y_test))
```

```
1.0
```

[38]:
```
model.print_tree()
```

```
feature|threshold -> 2 | 3.0
 T->1.0
 F->0.0
```

用自定义的回归树，$iris$ 数据集测试

[39]:
```
# 回归树
model = RegressionTree()
model.fit(X_train, y_train)
# 测试数据
print(model.score(X_test, y_test))
```

```
1.0
```

[40]:
```
model.print_tree()
```

```
feature|threshold -> 2 | 3.0
 T->1.0
 F->0.0
```

用 sklearn 实现分类树，*iris* 数据集测试

[41]:
```
from sklearn import tree

skl_model = tree.DecisionTreeClassifier()
skl_model.fit(X_train, y_train)
# 测试数据
print(skl_model.score(X_test, y_test))
```

```
1.0
```

用 sklearn 实现回归树，*iris* 数据集测试

[42]:
```
from sklearn import tree

skl_model = tree.DecisionTreeRegressor()
skl_model.fit(X_train, y_train)
# 测试数据
print(skl_model.score(X_test, y_test))
```

```
1.0
```

9 无监督学习方法

无监督学习的任务是找到数据更简单的表示：低维表示、稀疏表示和独立表示。低维表示将数据压缩到维度更少的空间中；稀疏表示将数据扩展到更多维度，但数据倾向于表示在坐标轴上；独立表示将数据拆分到能独立统计的维度上。

9.1 主成分分析法

主成分分析法，亦名 PCA。在第二章我们已经介绍过，为方便阅读，这里重述一下。PCA 将输入 $\boldsymbol{x}$ 投影表示成 $\boldsymbol{c}$。$\boldsymbol{c}$ 是比原始输入维数更低的表示，同时使得元素之间线性无关。假设有一个 $m \times n$ 的矩阵 $\boldsymbol{X}$，数据的均值为零，即 $\mathbb{E}[\boldsymbol{x}]=0$，$\boldsymbol{X}$ 对应的无偏样本协方差矩阵：$\mathrm{Var}[\boldsymbol{x}]=\frac{1}{m-1}\boldsymbol{X}^\top\boldsymbol{X}$。

PCA 是**通过线性变换找到一个 $\mathrm{Var}[\boldsymbol{c}]$ 是对角矩阵**的表示 $\boldsymbol{c}=\boldsymbol{V}^\top\boldsymbol{x}$，**矩阵 $\boldsymbol{X}$ 的主成分可以通过奇异值分解 (SVD) 得到**，也就是说主成分是 $\boldsymbol{X}$ 的右奇异向量。假设 $\boldsymbol{V}$ 是 $\boldsymbol{X}=\boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^\top$ 奇异值分解的右奇异向量，我们得到原来的特征向量方程：

$$\boldsymbol{X}^\top\boldsymbol{X}=(\boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^\top)^\top\boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^\top=\boldsymbol{V}\boldsymbol{\Sigma}^\top\boldsymbol{U}^\top\boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^\top=\boldsymbol{V}\boldsymbol{\Sigma}^2\boldsymbol{V}^\top \tag{60}$$

因为根据奇异值的定义 $\boldsymbol{U}^\top\boldsymbol{U}=\boldsymbol{I}$ 。因此 $\boldsymbol{X}$ 的方差可以表示为：$\mathrm{Var}[\boldsymbol{x}]=\frac{1}{m-1}\boldsymbol{X}^\top\boldsymbol{X}=\frac{1}{m-1}\boldsymbol{V}\boldsymbol{\Sigma}^2\boldsymbol{V}^\top$。

所以 $\boldsymbol{c}$ 的协方差满足：$\mathrm{Var}[\boldsymbol{c}]=\frac{1}{m-1}\boldsymbol{C}^\top\boldsymbol{C}=\frac{1}{m-1}\boldsymbol{V}^\top\boldsymbol{X}^\top\boldsymbol{X}\boldsymbol{V}=\frac{1}{m-1}\boldsymbol{V}^\top\boldsymbol{V}\boldsymbol{\Sigma}^2\boldsymbol{V}^\top\boldsymbol{V}=\frac{1}{m-1}\boldsymbol{\Sigma}^2$，因为根据奇异值定义 $\boldsymbol{V}^\top\boldsymbol{V}=\boldsymbol{I}$ 。$\boldsymbol{c}$ 的协方差是对角的，$\boldsymbol{c}$ 中的元素是彼此无关的。

自定义实现

[44]:
```
class PCA():

    def __init__(self):
        pass

    def fit(self, X, n_components):
        n_samples = np.shape(X)[0]
        covariance_matrix = (1 / (n_samples-1)) * (X - X.mean(axis=0)).T.dot(X - X.mean(axis=0))
        # 对协方差矩阵进行特征值分解
        eigenvalues, eigenvectors = np.linalg.eig(covariance_matrix)
        # 对特征值（特征向量）从大到小排序
        idx = eigenvalues.argsort()[::-1]
        eigenvalues = eigenvalues[idx][:n_components]
        eigenvectors = np.atleast_1d(eigenvectors[:, idx])[:, :n_components]
        # 得到低维表示
```

```
        X_transformed = X.dot(eigenvectors)
        return X_transformed
```

用自定义的 PCA，*iris* 数据集测试

[45]:
```
def create_data():
    iris = load_iris()
    df = pd.DataFrame(iris.data, columns=iris.feature_names)
    df['label'] = iris.target
    df.columns = ['sepal length', 'sepal width', 'petal length', 'petal width', 'label']
    data = np.array(df.iloc[:100, :])
    return data[:,:-1], data[:,-1]

X, y = create_data()
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3)
print(X_train[0], y_train[0])
```

```
[6.4 3.2 4.5 1.5] 1.0
```

[46]:
```
model = PCA()
model.fit(X_train[:20], 2)  # 取前 20 个数据测试
```

[46]:
```
array([[-5.75535057,  6.35428159],
       [-2.20479768,  5.95454742],
       [-2.26520769,  5.87699147],
       [-5.75594076,  6.00577621],
       [-2.30122891,  6.298297  ],
       [-2.5234036 ,  6.07896967],
       [-6.28590718,  5.67677524],
       [-4.04494949,  5.07447986],
       [-5.72955335,  6.51557733],
       [-5.98904726,  5.94901496],
       [-2.0822997 ,  5.87694555],
       [-1.95898348,  5.11327258],
       [-5.71754628,  6.37514216],
       [-5.10531342,  5.27893081],
       [-5.27416173,  5.68555962],
       [-2.18424456,  5.46422474],
       [-2.18386992,  6.24395656],
       [-2.51767442,  6.40421362],
       [-5.27762276,  5.47610728],
       [-2.14860047,  6.44145051]])
```

9.2 k-均值聚类

k-均值聚类 (k-means) 算法的基本思想是**初始随机给定 k 个簇中心，按照最邻近原则把待分类样本点分到各个簇。然后按平均法重新计算各个簇的质心，从而确定新的簇心。一直迭代，直到簇心的移动距离小于某个给定的值。**k 是我们事先需要给定的聚类数目。

算法步骤：

- 初始化 k 个不同的中心点 $\{\mu_1, \cdots, \mu_k\}$, 然后迭代交换以下两个步骤直至收敛。
- 第一步：每个训练样本分配到最近的中心点 μ_i 所代表的聚类 i。
- 第二步：每一个中心点 μ_i 更新为聚类 i 中所有训练样本 $\boldsymbol{x}_j$ 的均值。

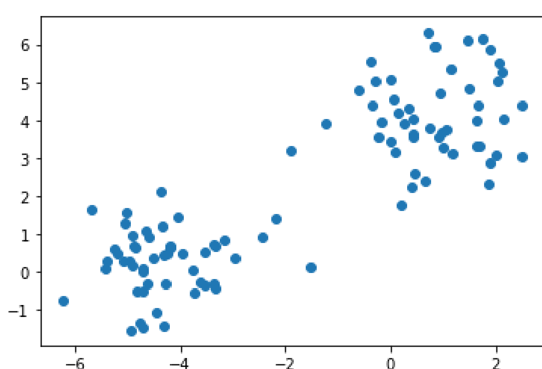
自定义实现

[47]:
```
def distEclud(x,y):
    """
    计算欧氏距离
    """
    return np.sqrt(np.sum((x-y)**2))

def randomCent(dataSet,k):
```

```python
    """
    为数据集构建一个包含 K 个随机质心的集合
    """
    m,n = dataSet.shape
    centroids = np.zeros((k,n))
    for i in range(k):
        index = int(np.random.uniform(0,m))
        centroids[i,:] = dataSet[index,:]
    return centroids

class KMeans():

    def __init__(self):
        self.dataSet = None
        self.k = None

    def fit(self, dataSet, k):
        self.dataSet = dataSet
        self.k = k
        m = np.shape(dataSet)[0]
        # 第一列存样本属于哪一簇
        # 第二列存样本的到簇的中心点的误差
        clusterAssment = np.mat(np.zeros((m,2)))
        clusterChange = True
        centroids = randomCent(self.dataSet,k)
        while clusterChange:
            clusterChange = False
            for i in range(m):
                minDist = 1e6
                minIndex = -1
                # 遍历所有的质心，找出最近的质心
                for j in range(k):
                    distance = distEclud(centroids[j,:], self.dataSet[i,:])
                    if distance < minDist:
                        minDist = distance
                        minIndex = j
                # 更新每一行样本所属的簇
                if clusterAssment[i,0] != minIndex:
                    clusterChange = True
                    clusterAssment[i,:] = minIndex, minDist**2
            # 更新质心
            for j in range(k):
                pointsInCluster = dataSet[np.nonzero(clusterAssment[:,0].A == j)[0]]  # 获取簇类所有的点
                centroids[j,:] = np.mean(pointsInCluster,axis=0)   # 对矩阵的行求均值

        return centroids,clusterAssment
```

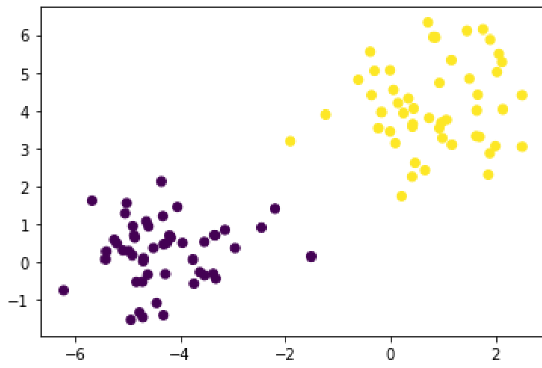
用自定义的 k-均值聚类，合成数据集测试

[48]:
```python
X, y = datasets.make_blobs(n_samples=100, centers=2, random_state=3)
plt.scatter(X[:,0], X[:,1]);
```



[49]:
```
model = KMeans()
center, clusterAssment = model.fit(X, k=2)
y_pred = np.squeeze(np.array(clusterAssment[:,0]))
```

[50]:
```
plt.scatter(X[:,0], X[:,1], c=y_pred);
```



[51]:
```
import matplotlib
import numpy
import sklearn
import pandas
import cvxopt

print("numpy:", numpy.__version__)
print("matplotlib:", matplotlib.__version__)
print("sklearn:", sklearn.__version__)
print("pandas:", pandas.__version__)
print("cvxopt:", cvxopt.__version__)
```

```
numpy: 1.14.5
matplotlib: 3.1.1
sklearn: 0.21.3
pandas: 0.25.1
cvxopt: 1.2.4
```