# 目錄

# SciPyCon 2018 sklearn 教程

> 原文：SciPy 2018 Scikit-learn Tutorial
>
> 译者：飞龙
>
> 自豪地采用谷歌翻译
>
> 欢迎任何人参与和完善：一个人可以走的很快，但是一群人却可以走的更远。

- ApacheCN 机器学习交流群 629470233
- ApacheCN 学习资源
- Scikit-learn 中文文档 0.19

- 在线阅读
- PDF格式
- EPUB格式
- MOBI格式
- 代码仓库

## 贡献指南

本教程中的代码缺少输出，希望大家能运行代码并补上输出。

## 赞助我


## 协议

# 一、Python 机器学习简介

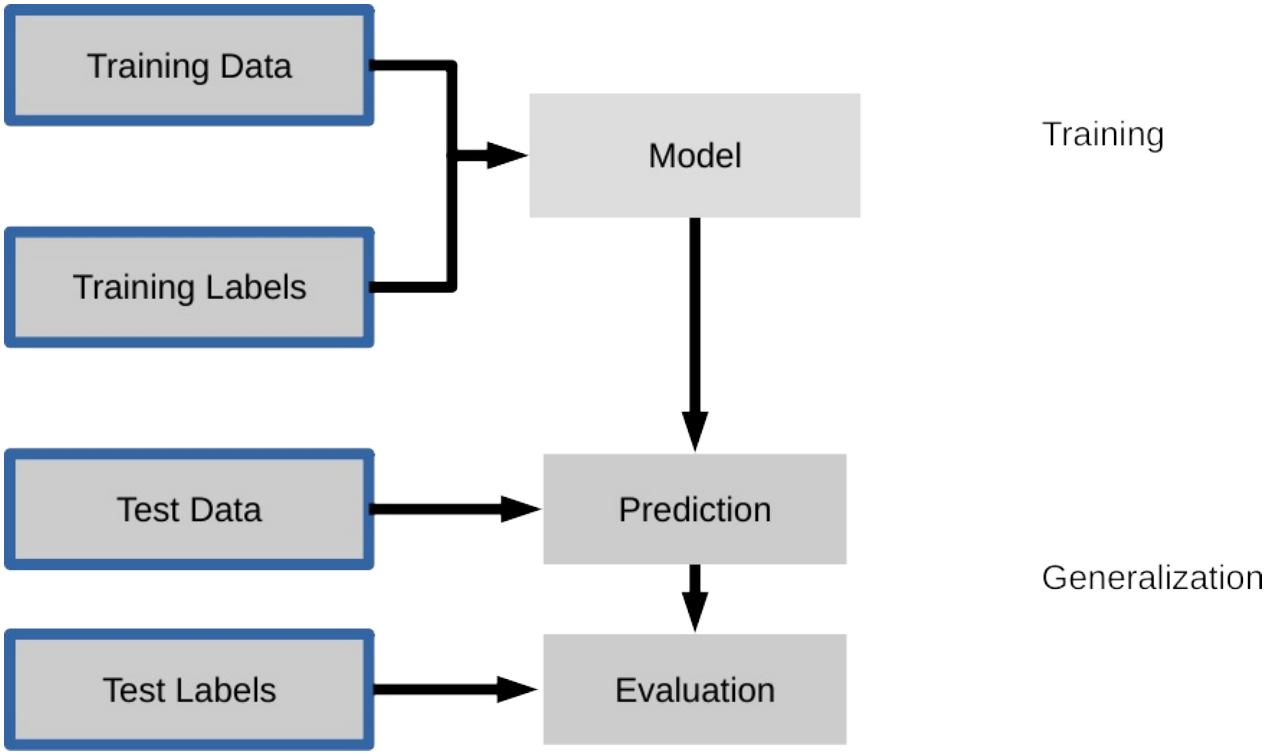
## 什么是机器学习？

机器学习是自动从数据中提取知识的过程，通常是为了预测新的，看不见的数据。一个典型的例子是垃圾邮件过滤器，用户将传入的邮件标记为垃圾邮件或非垃圾邮件。然后，机器学习算法从数据“学习”预测模型，数据区分垃圾邮件和普通电子邮件。该模型可以预测新电子邮件是否是垃圾邮件。

机器学习的核心是根据数据来自动化决策的概念，无需用户指定如何做出此决策的明确规则。

对于电子邮件，用户不提供垃圾邮件的单词或特征列表。相反，用户提供标记为垃圾邮件和非垃圾邮件的示例。

第二个核心概念是泛化。机器学习模型的目标是预测新的，以前没见过的数据。在实际应用中，将已标记的电子邮件标记为垃圾邮件，我们不感兴趣。相反，我们希望通过自动分类新的传入邮件来使用户更轻松。




数据通常作为数字的二维数组（或矩阵）展示给算法。 我们想要学习或做出决策的每个数据点（也称为样本或训练实例）表示为数字列表，即所谓的特征向量，其包含的特征表示这个点的属性。

稍后，我们将使用一个名为鸢尾花（Iris）的流行数据集 - 在许多其他数据集中。鸢尾花是机器学习领域的经典基准数据集，包含来自 3 种不同物种的 150 种鸢尾花的测量值：Iris-Setosa（山鸢尾），Iris-Versicolor（杂色鸢尾）和 Iris-Virginica（弗吉尼亚鸢尾）。

| 物种 | 图像 |
| --- | --- |
| 山鸢尾 |  |
| 杂色鸢尾 |  |
| | |

| | |
|---|---|
| 弗吉尼亚鸢尾 |  |

我们将每个花样本表示为数据阵列中的一行，列（特征）表示以厘米为单位的花测量值。 例如，我们可以用以下格式表示这个鸢尾花数据集，包括 150 个样本和 4 个特征，一个 `150×4` 的二维数组或矩阵：

$$\mathbf{X} = \begin{bmatrix} x_1^{(1)} & x_2^{(1)} & x_3^{(1)} & \dots & x_4^{(1)} \\ x_1^{(2)} & x_2^{(2)} & x_3^{(2)} & \dots & x_4^{(2)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ x_1^{(150)} & x_2^{(150)} & x_3^{(150)} & \dots & x_4^{(150)} \end{bmatrix}$$

（上标表示第 `i` 行，下标分别表示第 `j` 个特征。

我们今天将讨论两种机器学习：监督学习和无监督学习。

## 监督学习：分类和回归

在监督学习中，我们有一个数据集，由输入特征和所需输出组成的，例如垃圾邮件/非垃圾邮件示例。 任务是构建一个模型（或程序），它能够在给定特征集的情况下预测未见过的对象的所需输出。

一些更复杂的例子是：

- 通过望远镜给定物体的多色图像，确定该物体是星星，类星体还是星系。
- 给定一个人的照片，识别照片中的人物。
- 给定一个人观看的电影列表和他们对电影的个人评价，推荐他们想要的电影列表。
- 给定一个人的年龄，教育程度和职位，推断他们的薪水

这些任务的共同之处在于，存在与该对象相关联的一个或多个未知量，其需要从其他观察量确定。

监督学习进一步细分为两类，分类和回归：

在分类中，标签是离散的，例如“垃圾邮件”或“无垃圾邮件”。换句话说，它提供了类别之间的明确区分。此外，重要的是注意类标签是标称的，而不是序数变量。标称和序数变量都是类别变量的子类别。序数变量意味着顺序，例如，T恤尺寸 `XL> L> M> S` 。相反，标称变量并不意味着顺序，例如，我们（通常）不能假设“橙色>蓝色>绿色”。 在回归中，标签是连续的，即浮点输出。例如，在天文学中，确定物体是星星，星系还是类星体的任务是分类问题：标签来自三个不同的类别。另一方面，我们可能希望根据这些观察来估计物体的年龄：这将是一个回归问题，因为标签（年龄）是一个连续的数量。

在监督学习中，在提供期望结果的训练集与需要根据它推断期望结果的测试集之间，总是存在区别。模型的学习使预测模型拟合训练集，我们使用测试集来评估其泛化表现。

## 无监督学习

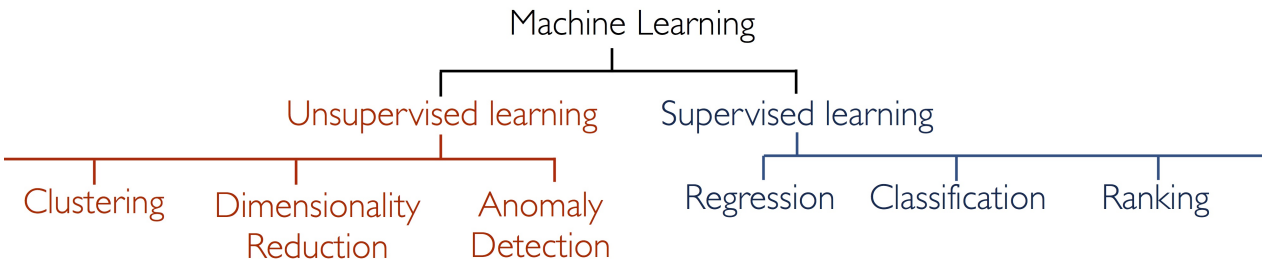
在无监督学习中，没有与数据相关的期望输出。相反，我们有兴趣从给定的数据中提取某种形式的知识或模型。从某种意义上说，你可以将无监督学习视为从数据本身发现标签的一种手段。无监督学习通常难以理解和评估。

无监督学习包括降维，聚类和密度估计之类的任务。例如，在上面讨论的鸢尾花数据中，我们可以使用无监督方法来确定显示数据结构的最佳测量值组合。我们将在下面看到，这种数据投影可用于在二维中可视化四维数据集。更多涉及无监督学习的问题是：

- 给定对遥远星系的详细观察，确定哪些特征或特征组合总结了最佳信息。
- 给定两个声源的混合（例如，一个人的谈话和一些音乐），将两者分开（这称为盲源分离问题）。
- 给定视频，隔离移动物体并相对于已看到的其他移动物体进行分类。
- 给定大量新闻文章，在这些文章中找到重复出现的主题。
- 给定一组图像，将相似的图像聚集在一起（例如，在可视化集合时对它们进行分组）

有时两者甚至可以合并：例如无监督学习可用于在异构数据中找到有用的特征，然后可以在监督框架内使用这些特征。

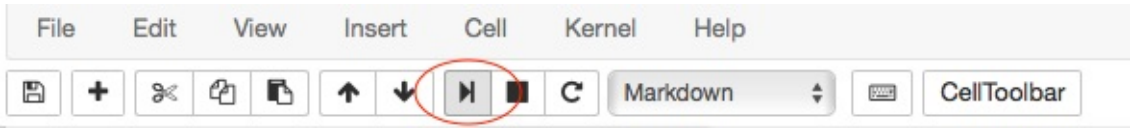
# （简化的）机器学习分类法


Machine Learning
Unsupervised learning
Supervised learning
Clustering
Dimensionality Reduction
Anomaly Detection
Regression
Classification
Ranking

# 二、Python 中的科学计算工具

## Jupyter Notebooks

你可以按 `[shift] + [Enter]` 或按菜单中的“播放”按钮来运行单元格。




在 `function(` 后面按 `[shift] + [tab]` ，可以获得函数或对象的帮助。

```
import numpy as np

np.mean(
```

```
Signature: np.mean(a, axis=None, dtype=None, out=None, keepdims=False)
Docstring:
Compute the arithmetic mean along the specified axis.
```

你还可以通过执行 `function?` 获得帮助。

```
In [3]: import numpy as np

        np.mean?
```

```
Signature: np.mean(a, axis=None, dtype=None, out=None, keepdims=False)
Docstring:
Compute the arithmetic mean along the specified axis.
```

## NumPy 数组

操作 `numpy` 数组是 Python 机器学习（或者，实际上是任何类型的科学计算）的重要部分。 对大多数人来说，这可能是一个简短的回顾。 无论如何，让我们快速浏览一些最重要的功能。

```
import numpy as np

# 设置随机种子来获得可重复性
rnd = np.random.RandomState(seed=123)

# 生成随机数组
X = rnd.uniform(low=0.0, high=1.0, size=(3, 5))  # a 3 x 5 array

print(X)
```

（请注意，NumPy 数组使用从 0 开始的索引，就像 Python 中的其他数据结构一样。）

```
# 元素访问

# 获取单个元素
# （这里是第一行第一列的元素）
print(X[0, 0])

# 获取一行
# （这里是第二行）
print(X[1])

# 获取一列
# （这里是第二列）
print(X[:, 1])

# 数组转置
print(X.T)
```

$$\begin{bmatrix} 1 & 5 \\ 2 & 6 \\ 3 & 7 \\ 4 & 8 \end{bmatrix}$$

```
# 在指定的时间间隔内创建均匀间隔的数字的行向量。
y = np.linspace(0, 12, 5)
print(y)

# 将行向量转换为列向量
print(y[:, np.newaxis])

# 获得形状或改变数组形状

# 生成随机数组
rnd = np.random.RandomState(seed=123)
X = rnd.uniform(low=0.0, high=1.0, size=(3, 5))  # a 3 x 5 array

print(X.shape)

# 将 X 大小变为 (3, 5)
X_reshaped = X.reshape(5, 3)
print(X_reshaped)

# 使用整数数组的索引（花式索引）
indices = np.array([3, 1, 0])
print(indices)
X[:, indices]
```

还有很多东西要知道，但是这些操作对于我们在本教程中将要做的事情至关重要。

## SciPy 稀疏数组

我们不会在本教程中大量使用它们，但稀疏矩阵在某些情况下非常好用。 在一些机器学习任务中，尤其是与文本分析相关的任务，数据可能大多为零。 存储所有这些零是非常低效的，并且以仅包含“非零”值的方式表示可以更有效。 我们可以创建和操作稀疏矩阵，如下所示：

```
# 创建一个包含大量零的随机数组
rnd = np.random.RandomState(seed=123)

X = rnd.uniform(low=0.0, high=1.0, size=(10, 5))
print(X)

# 将大多数元素设置为零
X[X < 0.7] = 0
print(X)
from scipy import sparse

# 将 X 转换为 CSR（压缩稀疏行）矩阵
X_csr = sparse.csr_matrix(X)
print(X_csr)

# 将稀疏矩阵转换为密集数组
print(X_csr.toarray())
```

（你可能偶然发现了一种将稀疏表示转换为密集表示的替代方法：`numpy.todense`；`toarray` 返回一个 NumPy 数组，而 `todense` 返回一个 NumPy 矩阵。在本教程中，我们将使用 NumPy 数组，而不是矩阵；scikit-learn 不支持后者。）

CSR 表示对于计算非常有效，但它不适合添加元素。为此，LIL（List-In-List）表示更好：

```
# 创建一个空的 LIL 矩阵并添加一些项目
X_lil = sparse.lil_matrix((5, 5))

for i, j in np.random.randint(0, 5, (15, 2)):
    X_lil[i, j] = i + j

print(X_lil)
print(type(X_lil))

X_dense = X_lil.toarray()
print(X_dense)
print(type(X_dense))
```

通常，一旦创建了 LIL 矩阵，将其转换为 CSR 格式很有用（许多 scikit-learn 算法需要 CSR 或 CSC 格式）

```
X_csr = X_lil.tocsr()
print(X_csr)
print(type(X_csr))
```

可用于各种问题的可用稀疏格式包括：

- `CSR`（压缩稀疏行）
- `CSC`（压缩稀疏列）
- `BSR`（块稀疏行）
- `COO`（坐标）
- `DIA`（对角线）
- `DOK`（键的字典）
- `LIL`（列表中的列表）

`scipy.sparse` 子模块还有很多稀疏矩阵的函数，包括线性代数，稀疏求解器，图算法等等。

# Matplotlib

机器学习的另一个重要部分是数据可视化。 Python 中最常用的工具是 `matplotlib` 。 这是一个非常灵活的包，我们将在这里介绍一些基础知识。

由于我们使用的是 Jupyter 笔记本，让我们使用 IPython 方便的内置“魔术函数”，即“ `matoplotlib` 内联”模式，它将直接在笔记本内部绘制图形。

```
%matplotlib inline

import matplotlib.pyplot as plt

# 绘制直线
x = np.linspace(0, 10, 100)
plt.plot(x, np.sin(x));

# 散点图
x = np.random.normal(size=500)
y = np.random.normal(size=500)
plt.scatter(x, y);

# 使用 imshow 展示绘图
# - note that origin is at the top-left by default!

x = np.linspace(1, 12, 100)
y = x[:, np.newaxis]

im = y * np.sin(x) * np.cos(y)
print(im.shape)

plt.imshow(im);

# 轮廓图
# - 请注意，此处的原点默认位于左下角！
plt.contour(im);

# 3D 绘图
from mpl_toolkits.mplot3d import Axes3D
ax = plt.axes(projection='3d')
xgrid, ygrid = np.meshgrid(x, y.ravel())
ax.plot_surface(xgrid, ygrid, im, cmap=plt.cm.viridis, cstride=2
, rstride=2, linewidth=0);
```

有许多可用的绘图类型。探索它们的一个实用方法是查看matplotlib库。

你可以在笔记本中轻松测试这些示例：只需复制每页上的源代码链接，然后使用 `%load magic` 将其放入笔记本中。例如：

```
# %load http://matplotlib.org/mpl_examples/pylab_examples/ellips
e_collection.py
```

# 三、数据表示和可视化

机器学习关于将模型拟合到数据；出于这个原因，我们首先讨论如何表示数据以便计算机理解。 除此之外，我们将基于上一节中的 `matplotlib` 示例构建，并展示如何可视化数据的一些示例。

## sklearn 中的数据

scikit-learn 中的数据（极少数例外）被假定存储为形状为 `[n_samples, n_features]` 的二维数组。许多算法也接受形状相同的 `scipy.sparse` 矩阵。

- `n_samples` ：样本数量：每个样本是要处理（例如分类）的项目。样本可以是文档，图片，声音，视频，天文对象，数据库中的行或 CSV 文件，或者你可以使用的一组固定数量的特征描述的任何内容。
- `n_features` ：特征或不同形状的数量，可用于以定量方式描述每个项目。特征通常是实值，但在某些情况下可以是布尔值或离散值。

必须事先固定特征的数量。然而，它可以是非常高的维度（例如数百万个特征），对于给定的样本，它们中的大多数是“零”。这是 `scipy.sparse` 矩阵可能有用的情况，因为它们比 NumPy 数组更具内存效率。

我们从上一节（或 Jupyter 笔记本）中回顾，我们将样本（数据点或实例）表示为数据数组中的行，并将相应的特征（“维度”）存储为列。

## 简单示例：鸢尾花数据集

作为简单数据集的一个例子，我们将看一下 scikit-learn 存储的鸢尾花数据。 数据包括三种不同鸢尾花的测量值。 在这个特定的数据集中有三种不同的鸢尾花，如下图所示：

| 物种 | 图像 |
| --- | --- |
| 山鸢尾 | |

| | |
|---|---|
| 山鸢尾 |  |
| 杂色鸢尾 |  |
| | |

| | |
|---|---|
| 弗吉尼亚鸢尾 |  |

简单问题：

让我们假设我们有兴趣对新观测值进行分类; 我们想分别预测未知的花是 Iris-Setosa，Iris-Versicolor 还是 Iris-Virginica。 根据我们在上一节中讨论的内容，我们将如何构建这样的数据集？

记住：我们需要一个大小为 `[n_samples x n_features]` 的二维数组。

- `n_samples` 指代什么？
- `n_features` 可能指代什么？

请记住，每个样本必须有固定数量的特征，并且对于每个样本，特征编号 `j` 必须是同一种数量。

## 在 **sklearn** 中加载鸢尾花数据集

对于将来使用机器学习算法实验，我们建议你收藏 UCI 机器学习仓库，该仓库托管许多常用的数据集，这些数据集对于机器学习算法的基准测试非常有用 - 这是机器学习实践者和研究人员非常流行的资源。 方便的是，其中一些数据集已经包含在 scikit-learn 中，因此我们可以跳过下载，读取，解析和清理这些文本/ CSV 文件的繁琐部分。你可以在这里找到 scikit-learn 中可用数据集的列表。

如，scikit-learn 拥有这些鸢尾花物种的非常简单的数据集。 数据包括以下内容：

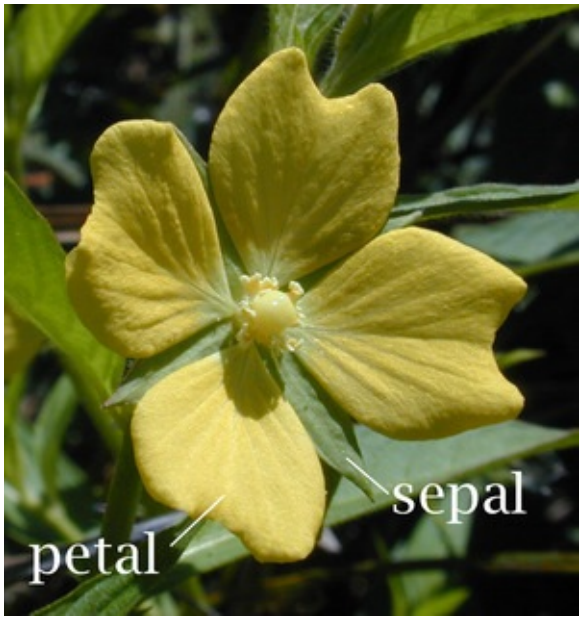
鸢尾花数据集中的特征：

- 萼片长度，厘米
- 萼片宽度，厘米
- 花瓣长度，厘米

- 花瓣宽度，厘米

要预测的目标类别：

- 山鸢尾
- 杂色鸢尾
- 弗吉尼亚鸢尾




（图片来源：“Petal-sepal”。通过 Wikimedia Commons 在 CC BY-SA 3.0 下获得许可）

scikit-learn 自带了鸢尾花 CSV 文件的副本以及辅助函数，用于将其加载到 `numpy` 数组中：

```
from sklearn.datasets import load_iris
iris = load_iris()
```

生成的数据集是一个 `Bunch` 对象：你可以使用方法 `keys()` 查看可用的内容：

```
iris.keys()
```

每个花样本的特征都存储在数据集的 `data` 属性中：

```
n_samples, n_features = iris.data.shape
print('Number of samples:', n_samples)
print('Number of features:', n_features)
# 第一个样本（第一朵花）的萼片长度，萼片宽度，花瓣长度和花瓣宽度
print(iris.data[0])
```

每个样本的类别信息存储在数据集的 `target` 属性中：

```
print(iris.data.shape)
print(iris.target.shape)

print(iris.target)

import numpy as np

np.bincount(iris.target)
```

使用 NumPy 的 `bincount` 函数（上图），我们可以看到类别在这个数据集中均匀分布 - 每个物种有 50 朵花，其中：

- 类 0：山鸢尾
- 类 1：杂色鸢尾
- 类 2：弗吉尼亚鸢尾

这些类名存储在最后一个属性中，即 `target_names` ：

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

这个数据是四维的，但我们可以使用简单的直方图或散点图一次可视化一个或两个维度。 再次，我们将从启用 `matplotlib` 内联模式开始：

```
%matplotlib inline

import matplotlib.pyplot as plt
x_index = 3

for label in range(len(iris.target_names)):
    plt.hist(iris.data[iris.target==label, x_index],
             label=iris.target_names[label],
             alpha=0.5)

plt.xlabel(iris.feature_names[x_index])
plt.legend(loc='upper right')
plt.show()

x_index = 3
y_index = 0

for label in range(len(iris.target_names)):
    plt.scatter(iris.data[iris.target==label, x_index],
                iris.data[iris.target==label, y_index],
                label=iris.target_names[label])

plt.xlabel(iris.feature_names[x_index])
plt.ylabel(iris.feature_names[y_index])
plt.legend(loc='upper left')
plt.show()
```

> 练习
>
> - 在上面的脚本中，更改 `x_index` 和 `y_index` ，找到两个参数的组合，最大限度地将这三个类分开。
> - 本练习是降维的预习，我们稍后会看到。

## 旁注：散点图矩阵

分析人员使用的常用工具称为散点图矩阵，而不是一次查看一个绘图。

散点图矩阵显示数据集中所有特征之间的散点图，以及显示每个特征分布的直方图。

```
import pandas as pd

iris_df = pd.DataFrame(iris.data, columns=iris.feature_names)
pd.plotting.scatter_matrix(iris_df, c=iris.target, figsize=(8, 8
));
```

## 其它可用的数据

Scikit-learn 提供了大量用于测试学习算法的数据集。 它们有三种形式：

- 打包数据：这些小数据集与 scikit-learn 安装打包在一起，可以使用 `sklearn.datasets.load_ *` 中的工具下载
- 可下载数据：这些较大的数据集可供下载，scikit-learn 包含简化此过程的工具。 这些工具可以在 `sklearn.datasets.fetch_ *` 中找到
- 生成的数据：有几个数据集是基于随机种子从模型生成的。 这些可以在 `sklearn.datasets.make_ *` 中找到

你可以使用 IPython 的制表符补全功能探索可用的数据集加载器，提取器和生成器。 从 `sklearn` 导入 `datasets` 子模块后，键入：

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

或者：

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

或者：

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

来查看可用函数列表。

```
from sklearn import datasets
```

请注意：许多这些数据集非常庞大，可能需要很长时间才能下载！

如果你在 IPython 笔记本中开始下载并且想要将其删除，则可以使用 ipython 的“内核中断”功能，该功能可在菜单中使用或使用快捷键 `Ctrl-m i` 。

你可以按 `Ctrl-m h` 获取所有 ipython 键盘快捷键的列表。

## 加载数字数据

现在我们来看看另一个数据集，我们必须更多考虑如何表示数据。 我们可以采用与上述类似的方式探索数据：

```
from sklearn.datasets import load_digits
digits = load_digits()

digits.keys()

n_samples, n_features = digits.data.shape
print((n_samples, n_features))

print(digits.data[0])
print(digits.target)
```

这里的目标只是数据所代表的数字。 数据是长度为 64 的数组.....但这些数据意味着什么？

实际上有个线索，我们有两个版本的数据数组：数据和图像。 我们来看看它们：

```
print(digits.data.shape)
print(digits.images.shape)
```

通过简单的形状改变，我们可以看到它们是相关的：

```
import numpy as np
print(np.all(digits.images.reshape((1797, 64)) == digits.data))
```

让我们可视化数据。 它比我们上面使用的简单散点图更复杂，但我们可以很快地完成它。

```
# 建立图形
fig = plt.figure(figsize=(6, 6))  # figure size in inches
fig.subplots_adjust(left=0, right=1, bottom=0, top=1, hspace=0.05
, wspace=0.05)

# 绘制数字：每个图像是 8x8 像素
for i in range(64):
    ax = fig.add_subplot(8, 8, i + 1, xticks=[], yticks=[])
    ax.imshow(digits.images[i], cmap=plt.cm.binary, interpolatio
n='nearest')

    # 用目标值标记图像
    ax.text(0, 7, str(digits.target[i]))
```

我们现在看到这些特征的含义。 每个特征是实数值，表示手写数字的 8×8 图像中的像素的暗度。

即使每个样本具有固有的二维数据，数据矩阵也将该 2D 数据展平为单个向量，该向量可以包含在数据矩阵的一行中。

> 练习：处理人脸数据集
>
> 这里，我们将花点时间亲自探索数据集。稍后我们将使用 Olivetti faces 数据集。花点时间获取数据（大约 1.4MB），并可视化人脸。你可以复制用于可视化上述数字的代码，并为此数据进行修改。

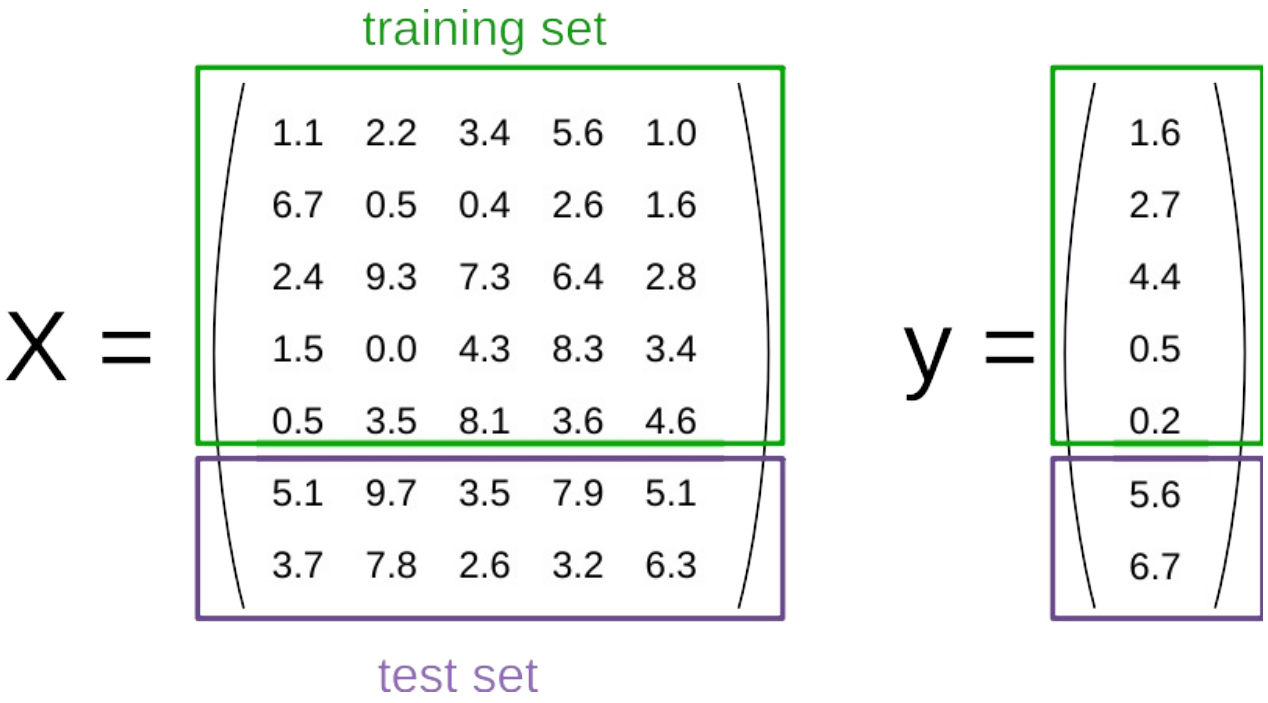
```
from sklearn.datasets import fetch_olivetti_faces
# 获取人脸数据
# 使用上面的脚本绘制人脸图像数据。
# 提示：plt.cm.bone 是用于这个数据的很好的颜色表
```

答案：

```
# %load solutions/03A_faces_plot.py
```

# 四、训练和测试数据

为了评估我们的监督模型的泛化能力，我们可以将数据分成训练和测试集：




```
from sklearn.datasets import load_iris

iris = load_iris()
X, y = iris.data, iris.target
```

考虑如何正常执行机器学习，训练/测试分割的想法是有道理的。真实世界系统根据他们拥有的数据进行训练，当其他数据进入时（来自客户，传感器或其他来源），经过训练的分类器必须预测全新的数据。我们可以在训练期间使用训练/测试分割来模拟 - 测试数据是“未来数据”的模拟，它将在生产期间进入系统。

特别是对于鸢尾花，其中的 150 个标签是有序的，这意味着如果我们使用比例分割来分割数据，这将导致类分布基本上改变。例如，如果我们执行常见的 2/3 训练数据和 1/3 测试数据的分割，我们的训练数据集将仅包含类别 0 和 1（Setosa 和 Versicolor），我们的测试集将仅包含类别标签为 2 的样本（Virginica）。

假设所有样本彼此独立（而不是时间序列数据），我们希望在分割数据集之前随机打乱数据集。

现在我们需要将数据分成训练和测试集。 幸运的是，这是机器学习中常见的模式，scikit-learn 具有预先构建的函数，可以将数据分成训练和测试集。 在这里，我们使用 50% 的数据来训练，50% 来测试。 80% 和 20% 是另一种常见的分割，但没有严格的规则。 最重要的是，要在训练期间未见过的数据上，公平地评估您的系统！

```
from sklearn.model_selection import train_test_split

train_X, test_X, train_y, test_y = train_test_split(X, y,
                                                    train_size=0
.5,
                                                    test_size=0.5
,
                                                    random_state=
123)
print("Labels for training data:")
print(train_y)

print("Labels for test data:")
print(test_y)
```

提示：分层分割

特别是对于相对较小的数据集，最好分层分割。分层意味着我们在测试和训练集中保持数据集的原始类比例。例如，在我们随机拆分前面的代码示例中所示的数据集之后，我们的类比例（百分比）如下：

```
print('All:', np.bincount(y) / float(len(y)) * 100.0)
print('Training:', np.bincount(train_y) / float(len(train_y)) *
100.0)
print('Test:', np.bincount(test_y) / float(len(test_y)) * 100.0)
```

因此，为了分层分割，我们可以将 `label` 数组作为附加选项传递给 `train_test_split` 函数：

```
train_X, test_X, train_y, test_y = train_test_split(X, y,
                                                    train_size=0
.5,
                                                    test_size=0.5
,
                                                    random_state=
123,
                                                    stratify=y)

print('All:', np.bincount(y) / float(len(y)) * 100.0)
print('Training:', np.bincount(train_y) / float(len(train_y)) *
100.0)
print('Test:', np.bincount(test_y) / float(len(test_y)) * 100.0)
```

通过在训练过程中看到的数据上评估我们的分类器性能，我们可能对模型的预测能力产生错误的信心。在最坏的情况下，它可能只是记住训练样本，但完全没有分类新的类似样本 - 我们真的不想将这样的系统投入生产！

不使用相同的数据集进行训练和测试（这称为“重取代评估”），为了估计训练模型对新数据的效果，使用训练/测试分割要好得多。

```
from sklearn.neighbors import KNeighborsClassifier

classifier = KNeighborsClassifier().fit(train_X, train_y)
pred_y = classifier.predict(test_X)

print("Fraction Correct [Accuracy]:")
print(np.sum(pred_y == test_y) / float(len(test_y)))
```

我们还可以可视化正确的预测.....

```
print('Samples correctly classified:')
correct_idx = np.where(pred_y == test_y)[0]
print(correct_idx)
```

...以及错误的预测。

```
print('Samples incorrectly classified:')
incorrect_idx = np.where(pred_y != test_y)[0]
print(incorrect_idx)

# 绘制两个维度

for n in np.unique(test_y):
    idx = np.where(test_y == n)[0]
    plt.scatter(test_X[idx, 1], test_X[idx, 2], label="Class %s"
 % str(iris.target_names[n]))

plt.scatter(test_X[incorrect_idx, 1], test_X[incorrect_idx, 2],
color="darkred")

plt.xlabel('sepal width [cm]')
plt.ylabel('petal length [cm]')
plt.legend(loc=3)
plt.title("Iris Classification results")
plt.show()
```

我们可以看到错误发生在绿色（类 1）和灰色（类 2）重叠的区域。 这使我们能够深入了解需要添加的特征 - 任何有助于分离类 1 和类 2 的特征都应该提高分类器的表现。

> 练习
>
> 打印3个错误预测的真实标签，并修改我们上面使用的散点图代码，来在 2D 散点图中用不同的标记可视化和区分这三个样本。 你能解释为什么我们的分类器做出了这些错误的预测吗？

```
# %load solutions/04_wrong-predictions.py
```

# 五、监督学习第一部分：分类

为了可视化机器学习算法的工作原理，研究二维或一维数据（即只有一个或两个特征的数据）通常很有帮助。 实际上，数据集通常具有更多特征，很难在二维屏幕上绘制高维数据。

在我们转向更多“真实世界”的数据集之前，我们将展示一些非常简单的示例。

首先，我们将从二维来看二分类问题。 我们使用 `make_blobs` 函数生成人造数据。

```
from sklearn.datasets import make_blobs

X, y = make_blobs(centers=2, random_state=0, cluster_std=1.5)

print('X ~ n_samples x n_features:', X.shape)
print('y ~ n_samples:', y.shape)

print('First 5 samples:\n', X[:5, :])

print('First 5 labels:', y[:5])
```

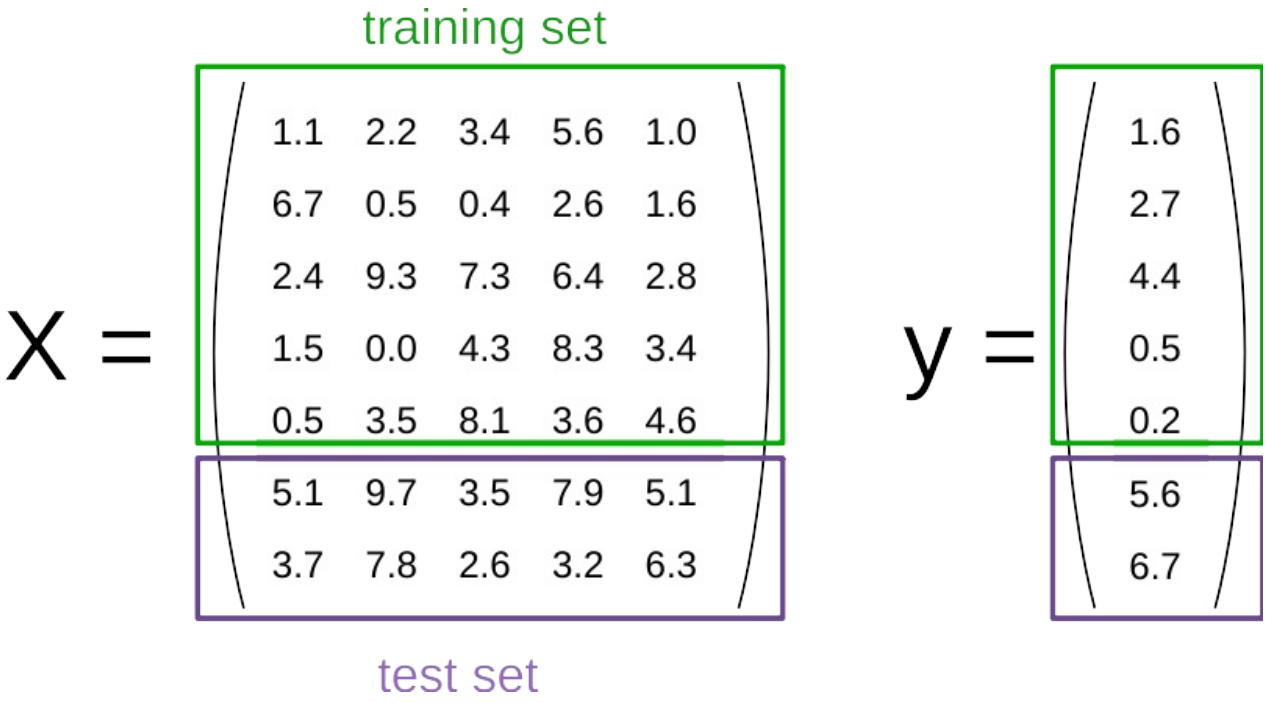
由于数据是二维的，我们可以将每个样本绘制为二维坐标系中的一个点，第一个特征是 `x` 轴，第二个特征是 `y` 轴。

```
plt.figure(figsize=(8, 8))
plt.scatter(X[y == 0, 0], X[y == 0, 1], s=40, label='0')
plt.scatter(X[y == 1, 0], X[y == 1, 1], s=40, label='1',
            marker='s')

plt.xlabel('first feature')
plt.ylabel('second feature')
plt.legend(loc='upper right');
```

分类是一项监督任务，由于我们对其在未见过的数据上的表现感兴趣，因此我们将数据分为两部分：

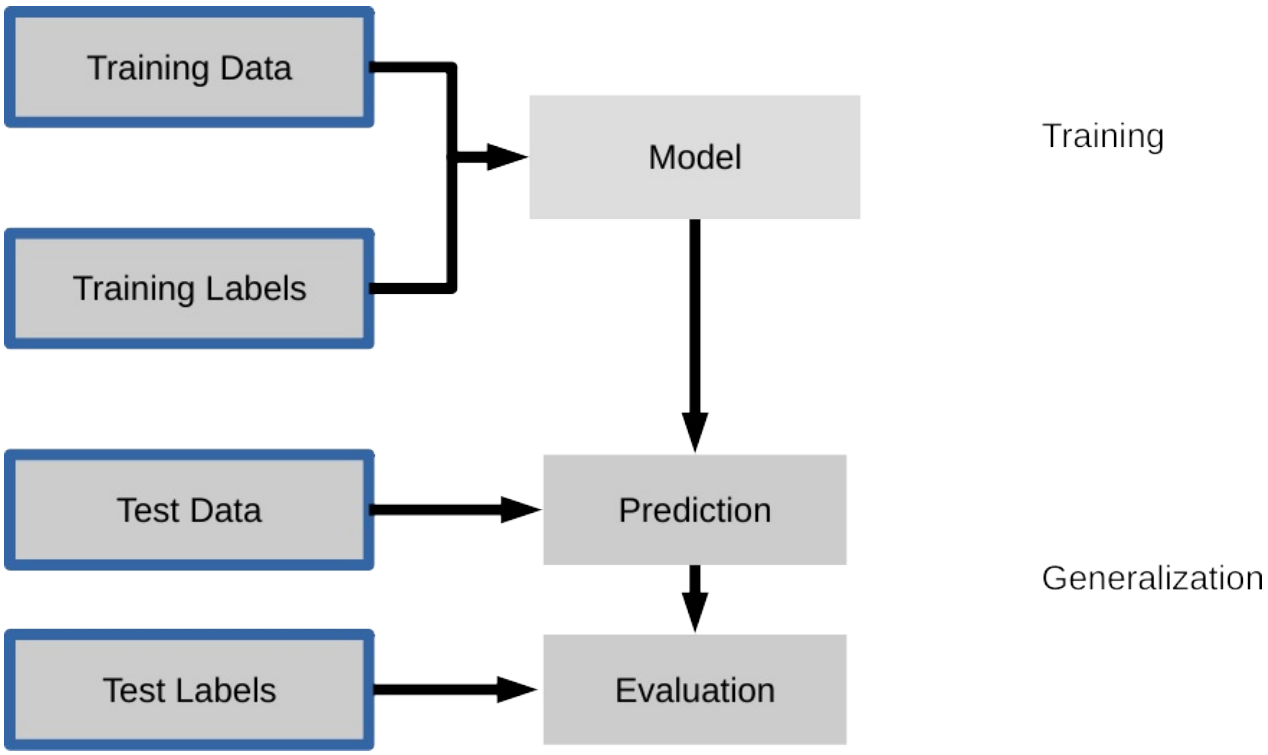
- 训练集，学习算法用它来拟合模型
- 测试集，用于评估模型的泛化性能

来自 `model_selection` 模块的 `train_test_split` 函数为我们做了这个 - 我们将使用它，将数据集拆分为 75% 的训练数据和 25% 的测试数据。




```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X, y,
                                                    test_size=0.
25,
                                                    random_state=
1234,
                                                    stratify=y)
```

## scikit-learn 估计器 API




scikit-learn 中的每个算法都通过“估计器”对象提供。（scikit-learn 中的所有模型都具有非常一致的接口）。例如，我们首先导入逻辑回归类。

```
from sklearn.linear_model import LogisticRegression
```

下面，我们实例化估计器对象。

```
classifier = LogisticRegression()
X_train.shape
y_train.shape
```

为了从我们的数据构建模型，即学习如何分类新的点，我们使用训练数据，以及相应的训练标签（训练数据点的所需输出）调用 `fit` 函数：

```
classifier.fit(X_train, y_train)
```

（默认情况下，一些估计方法如 `fit` 返回 `self` 。因此，在执行上面的代码片段之后，你将看到 `LogisticRegression` 的特定实例的默认参数。另一种获取估计器的初始化参数的方法是执行 `classifier.get_params()` ，返回参数字典。）

然后，我们可以将模型应用于未见过的数据，并使用模型使用 `predict` 方法预测估计的结果：

```
prediction = classifier.predict(X_test)
```

我们可以将它们与真实标签比较：

```
print(prediction)
print(y_test)
```

通过测量预测的正确比例，我们可以定量评估我们的分类器。这称为准确度：

```
np.mean(prediction == y_test)
```

还有一个便利函数，`score`，所有 scikit-learn 分类器必须直接从测试数据计算：

```
classifier.score(X_test, y_test)
```

将（测试集上的）泛化表现与训练集上的表现进行比较通常很有帮助：

```
classifier.score(X_train, y_train)
```

`LogisticRegression` 是一种所谓的线性模型，这意味着它将在输入空间中创建线性决策。 在 2d 中，这只是意味着它找到一条线来将蓝色与红色分开：

```
from figures import plot_2d_separator

plt.scatter(X[y == 0, 0], X[y == 0, 1], s=40, label='0')
plt.scatter(X[y == 1, 0], X[y == 1, 1], s=40, label='1', marker=
's')

plt.xlabel("first feature")
plt.ylabel("second feature")
plot_2d_separator(classifier, X)
plt.legend(loc='upper right');
```

估计参数：所有估计模型的参数都是以下划线结尾的，估计器对象的属性。 这里是直线的系数和偏移量：

```
print(classifier.coef_)
print(classifier.intercept_)
```

## 另一个分类器：K 最近邻

另一种流行且易于理解的分类器是 K 最近邻（kNN）。它有一个最简单的学习策略：给出一个新的，未知的观侧值，在你的参考数据库中查找，哪些具有最接近的特征并分配优势类别。

接口与上面的 `LogisticRegression` 完全相同。

```
from sklearn.neighbors import KNeighborsClassifier
```

这次我们设置 `KNeighborsClassifier` 的参数，告诉它我们只想查看 30 个最近的邻居：

```
knn = KNeighborsClassifier(n_neighbors=30)
```

我们使用训练数据拟合模型：

```
knn.fit(X_train, y_train)

plt.scatter(X_train[y_train == 0, 0], X_train[y_train == 0, 1],
            s=40, label='0')
plt.scatter(X_train[y_train == 1, 0], X_train[y_train == 1, 1],
            s=40, label='1', marker='s')

plt.xlabel("first feature")
plt.ylabel("second feature")
plot_2d_separator(knn, X)
plt.legend(loc='upper right');

knn.score(X_train, y_train)

plt.scatter(X_test[y_test == 0, 0], X_test[y_test == 0, 1],
            s=40, label='0')
plt.scatter(X_test[y_test == 1, 0], X_test[y_test == 1, 1],
            s=40, label='1', marker='s')

plt.xlabel("first feature")
plt.ylabel("second feature")
plot_2d_separator(knn, X)
plt.legend(loc='upper right');

knn.score(X_test, y_test)
```

> 练习
>
> 将 `KNeighborsClassifier` 应用于鸢尾花数据集。玩转 `n_neighbors` 的不同值，观察训练和测试得分的变化情况。

# 六、监督学习第二部分：回归分析

在回归中，我们试图预测连续输出变量 - 而不是我们在之前的分类示例中预测的标称变量。

让我们从一个简单的玩具示例开始，其中包含一个特征维度（解释性变量）和一个目标变量。 我们将使用一些噪声从正弦曲线创建数据集：

```
x = np.linspace(-3, 3, 100)
print(x)

rng = np.random.RandomState(42)
y = np.sin(4 * x) + x + rng.uniform(size=len(x))

plt.plot(x, y, 'o');
```

## 线性回归

我们将介绍的第一个模型是所谓的简单线性回归。 在这里，我们想要为数据拟合一条直线。

最简单的模型之一是线性模型，它只是试图预测数据位于一条线上。 找到这样一条直线的一种方法是 `LinearRegression`（也称为普通最小二乘（OLS）回归）。 `LinearRegression` 的接口与之前的分类器完全相同，只是 `y` 现在包含浮点值而不是类别。

我们记得，scikit-learn API 要求我们将目标变量（`y`）提供为一维数组; scikit-learn 的 API 期望样本（`X`）是个二维数组 - 即使它可能只包含一个特征。 因此，让我们将 1 维 NumPy 数组 `x` 转换为具有 2 个轴的数组 `X`：

```
print('Before: ', x.shape)
X = x[:, np.newaxis]
print('After: ', X.shape)
```

同样，我们首先将数据集拆分为训练（75%）和测试集（25%）：

```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X, y, test_s
ize=0.25, random_state=42)
```

接下来，我们使用 `LinearRegression` 中实现的学习算法使回归模型拟合训练数据：

```
from sklearn.linear_model import LinearRegression

regressor = LinearRegression()
regressor.fit(X_train, y_train)
```

在拟合训练数据后，我们使用以下值来参数化线性回归模型。

```
print('Weight coefficients: ', regressor.coef_)
print('y-axis intercept: ', regressor.intercept_)
```

由于我们的回归模型是线性模型，因此目标变量（ `y` ）和特征变量（ `x` ）之间的关系定义为：

```
y = weight × x + intercept
```

将最小值和最大值插入这个公式，我们可以绘制拟合我们的训练数据的回归：

```
min_pt = X.min() * regressor.coef_[0] + regressor.intercept_
max_pt = X.max() * regressor.coef_[0] + regressor.intercept_

plt.plot([X.min(), X.max()], [min_pt, max_pt])
plt.plot(X_train, y_train, 'o');
```

与之前笔记本中的分类估计器类似，我们使用 `predict` 方法来预测目标变量。我们希望这些预测值落在我们之前绘制的直线上：

```
y_pred_train = regressor.predict(X_train)

plt.plot(X_train, y_train, 'o', label="data")
plt.plot(X_train, y_pred_train, 'o', label="prediction")
plt.plot([X.min(), X.max()], [min_pt, max_pt], label='fit')
plt.legend(loc='best')
```

我们在上图中看到，直线能够捕获数据的一般斜率，但没有太多细节。

接下来，让我们试试测试集：

```
y_pred_test = regressor.predict(X_test)

plt.plot(X_test, y_test, 'o', label="data")
plt.plot(X_test, y_pred_test, 'o', label="prediction")
plt.plot([X.min(), X.max()], [min_pt, max_pt], label='fit')
plt.legend(loc='best');
```

同样，scikit-learn 提供了一种简便方法，使用 `score` 方法定量评估预测。 对于回归任务，这是 R2 得分。 另一种流行的方式是均方差（MSE）。 顾名思义，MSE 只是预测和实际目标值的均方差。

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(\text{predicted}_i - \text{true}_i)^2$$

```
regressor.score(X_test, y_test)
```

> 练习
>
> 将（非线性）特征 `sin(4x)` 添加到 `X` 并将重新拟合 `X_train` （和 `X_test` ）。 使用这个新的更丰富的还是线性的模型可视化预测。 提示：你可以使用 `np.concatenate(A, B, axis=1)` 将两个矩阵 `A` 和 `B` 水平连接（来组合列）。

```
# %load solutions/06B_lin_with_sine.py
```

## KNeighborsRegression

就像分类一样，我们也可以使用基于邻居的方法来回归。 我们可以简单地获取最近点的输出，或者我们可以平均几个最近点。 这种方法不像分类那样流行于回归，但仍然是一个很好的基线。

```
from sklearn.neighbors import KNeighborsRegressor
kneighbor_regression = KNeighborsRegressor(n_neighbors=1)
kneighbor_regression.fit(X_train, y_train)
```

再次，让我们看一下训练和测试集的行为：

```
y_pred_train = kneighbor_regression.predict(X_train)

plt.plot(X_train, y_train, 'o', label="data", markersize=10)
plt.plot(X_train, y_pred_train, 's', label="prediction", markers
ize=4)
plt.legend(loc='best');
```

在训练集上，我们做得很好：每个点都是它自己最近的邻居！

```
y_pred_test = kneighbor_regression.predict(X_test)

plt.plot(X_test, y_test, 'o', label="data", markersize=8)
plt.plot(X_test, y_pred_test, 's', label="prediction", markersiz
e=4)
plt.legend(loc='best');
```

在测试集上，我们也更好地捕捉变化，但我们的估计看起来比以前更加混乱。 我们来看看 R2 得分：

```
kneighbor_regression.score(X_test, y_test)
```

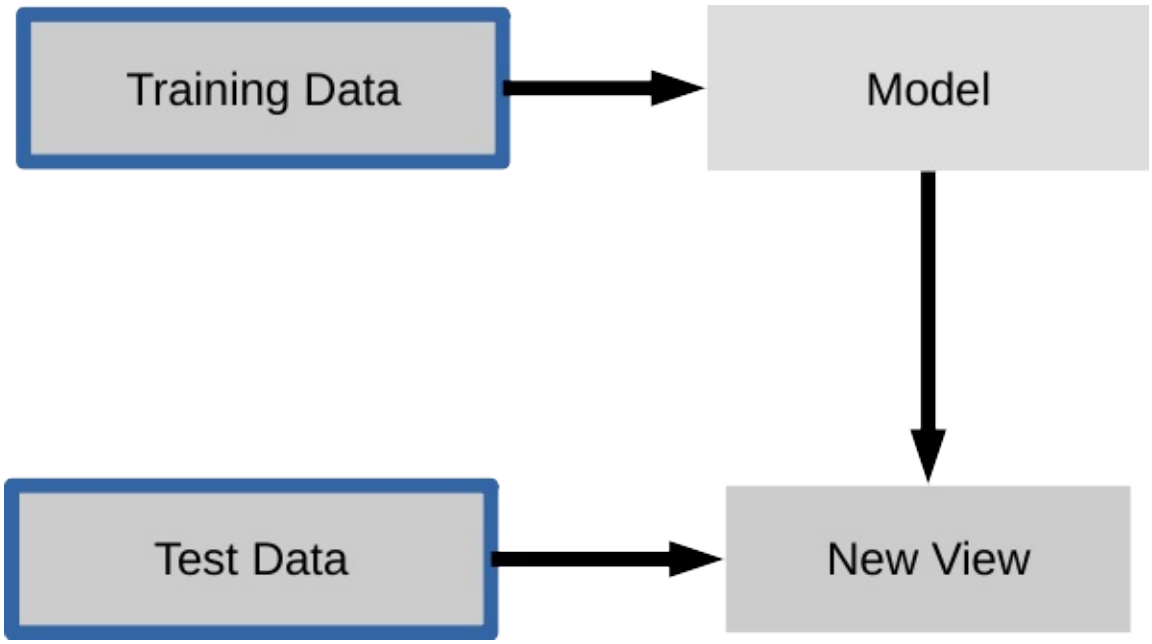
比以前好多了！ 在这里，线性模型不适合我们的问题；它缺乏复杂性，因此不适合我们的数据。

> 练习
>
> 在波士顿住房数据集中，比较 `KNeighborsRegressor` 和 `LinearRegression` 。 你可以使用 `sklearn.datasets.load_boston` 加载数据集。 你可以通过阅读 `DESCR` 属性来了解数据集。

```
# %load solutions/06A_knn_vs_linreg.py
```

# 七、无监督学习第一部分：变换

许多无监督学习的实例，例如降维，流形学习和特征提取，在没有任何额外输入的情况下找到输入数据的新表示。（与监督学习相反，如之前的分类和回归示例，无监督算法不需要或考虑目标变量）。




一个非常基本的例子是我们的数据重缩放，这是许多机器学习算法的要求，因为它们不是规模不变的 - 重缩放属于数据预处理类别，几乎不能称为学习。存在许多不同的重缩放技术，在下面的示例中，我们将看一个通常称为“标准化”的特定方法。在这里，我们将重缩放数据，使每个特征以零（均值为 0）为中心，具有单位方差（标准差为 1）。

例如，如果我们的一维数据集的值为 `[1,2,3,4,5]`，则标准化值为：

- 1 -> -1.41
- 2 -> -0.71
- 3 -> 0.0
- 4 -> 0.71
- 5 -> 1.41

通过等式 `z = (x - μ) / σ` 计算，其中 `μ` 是样本均值，`σ` 是标准差。

```
ary = np.array([1, 2, 3, 4, 5])
ary_standardized = (ary - ary.mean()) / ary.std()
ary_standardized
```

尽管标准化是最基本的预处理过程 - 正如我们在上面的代码中看到的那样 - scikit-learn 为此计算实现了 `StandardScaler` 类。在后面的部分中，我们将了解为什么以及何时 scikit-learn 接口在我们上面执行的代码片段中派上用场。

这样的预处理具有与我们迄今为止看到的监督学习算法非常相似的接口。要使用 scikit-learn 的 `Transformer` 接口做更多练习，让我们首先加载鸢尾花数据集并重缩放它：

```
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split

iris = load_iris()
X_train, X_test, y_train, y_test = train_test_split(iris.data, i
ris.target, random_state=0)
print(X_train.shape)
```

鸢尾花数据集不是“居中”的，即它具有非零均值，并且每个分量的标准差不同：

```
print("mean : %s " % X_train.mean(axis=0))
print("standard deviation : %s " % X_train.std(axis=0))
```

要使用预处理方法，我们首先导入估计器，这里是 `StandardScaler` ，并实例化它：

```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
```

与分类和回归算法一样，我们调用 `fit` 来从数据中学习模型。由于这是无监督的模型，我们只传递 `X` 而不是 `y` 。这仅仅估计平均值和标准差。

```
scaler.fit(X_train)
```

现在我们可以通过应用 `transform` （不是 `predict` ）方法来重缩放数据：

```
X_train_scaled = scaler.transform(X_train)
```

`X_train_scaled` 具有相同数量的样本和特征，但减去了平均值，并且所有特征都被缩放，来具有单位标准差：

```
print(X_train_scaled.shape)

print("mean : %s " % X_train_scaled.mean(axis=0))
print("standard deviation : %s " % X_train_scaled.std(axis=0))
```

总结一下：通过 `fit` 方法，估计器拟合我们提供的数据。 在该步骤中，估计器根据数据估计参数（这里是平均值和标准差）。 然后，如果我们转换数据，这些参数将用于转换数据集。 （请注意， `transform` 方法不会更新这些参数）。

重要的是要注意，相同的转换应用于训练和测试集。 这导致通常在缩放后测试数据的平均值不为零：

```
X_test_scaled = scaler.transform(X_test)
print("mean test data: %s" % X_test_scaled.mean(axis=0))
```

以完全相同的方式转换训练和测试数据非常重要，对于理解数据的以下处理步骤，如下图所示：

```
from figures import plot_relative_scaling
plot_relative_scaling()
```

有几种常见的方法用于缩放数据。 最常见的是我们刚刚介绍的 `StandardScaler` ，但是使用 `MinMaxScaler` 重缩放数据，来固定最小值和最大值（通常在 0 和 1 之间），或使用更鲁棒的统计量（如中位数和分位数），而不是平均值和标准差（使用 `RobustScaler` ），也很有用。

```
from figures import plot_scaling
plot_scaling()
```

## 主成分分析

主成分分析（PCA）是一种更有趣的无监督转换。这是一种技术，通过创建线性投影来降低数据维数。也就是说，我们寻找新的特征来表示数据，它是旧数据的线性组合（即我们旋转它）。因此，我们可以将 PCA 视为将数据投影到新的特征空间。

PCA 找到这些新方向的方式，是寻找最大方差的方向。通常只保留解释数据中大部分变化的少数成分。这里，前提是减少数据集的大小（维度），同时捕获其大部分信息。降维有用的原因很多：它可以在运行学习算法时降低计算成本，减少存储空间，并可能有助于所谓的“维度灾难”，我们将在后面详细讨论。

为了说明旋转的样子，我们首先在二维数据上显示它并保留两个主成分。这是一个例子：

```
from figures import plot_pca_illustration
plot_pca_illustration()
```

现在让我们更详细地介绍所有步骤：我们创建一个旋转的高斯 blob：

```
rnd = np.random.RandomState(5)
X_ = rnd.normal(size=(300, 2))
X_blob = np.dot(X_, rnd.normal(size=(2, 2))) + rnd.normal(size=2
)
y = X_[:, 0] > 0
plt.scatter(X_blob[:, 0], X_blob[:, 1], c=y, linewidths=0, s=30)
plt.xlabel("feature 1")
plt.ylabel("feature 2");
```

与往常一样，我们实例化我们的 PCA 模型。 默认情况下，保留所有方向。

```
from sklearn.decomposition import PCA
pca = PCA()
```

然后我们使用我们的数据拟合 PCA 模型。 由于 PCA 是无监督算法，因此没有输出 `y` 。

```
pca.fit(X_blob)
```

然后我们可以转换数据，投影在主成分上：

```
X_pca = pca.transform(X_blob)

plt.scatter(X_pca[:, 0], X_pca[:, 1], c=y, linewidths=0, s=30)
plt.xlabel("first principal component")
plt.ylabel("second principal component");

pca = PCA(n_components=1).fit(X_blob)

X_blob.shape

pca.transform(X_blob).shape
```

在图的左侧，你可以看到之前右上角的四个点。 PCA 发现第一个成分是沿对角线，第二个组件垂直于它。 当 PCA 发现旋转时，主成分始终彼此成直角（“正交”）。

## 将 PCA 降维用于可视化

考虑数字数据集。 它无法在单个 2D 绘图中可视化，因为它具有 64 个特征。 我们将使用 `sklearn` 示例中的示例提取 2 个维度用于可视化。

```
from figures import digits_plot

digits_plot()
```

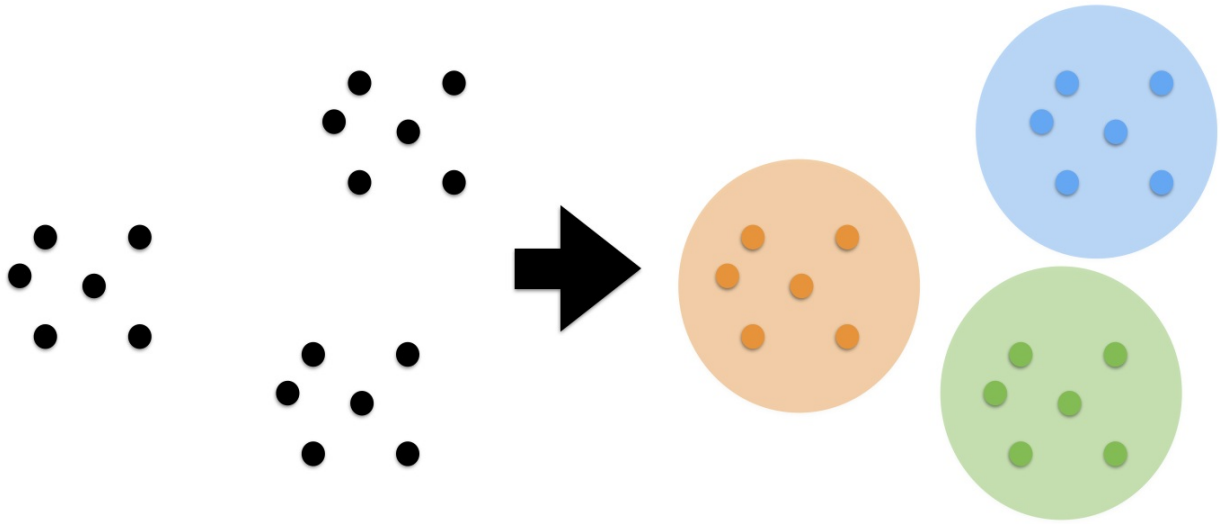
请注意，此投影是在没有任何标签的信息（由颜色表示）的情况下确定的：这是无监督学习的意义。 然而，我们看到投影让我们深入了解参数空间中不同数字的分布。

> 练习
>
> 使用前两个主成分可视化鸢尾花数据集，并将此可视化与使用两个原始特征进行比较。

```
# %load solutions/07A_iris-pca.py
```

# 八、无监督学习第二部分：聚类

聚类是根据一些预定义的相似性或距离（相异性）度量（例如欧氏距离），将样本收集到相似样本分组中的任务。



在本节中，我们将在一些人造和真实数据集上，探讨一些基本聚类任务。

以下是聚类算法的一些常见应用：

- 用于数据减少的压缩
- 将数据汇总为推荐系统的再处理步骤
- 相似性：
  - 分组相关的网络新闻（例如 Google 新闻）和网络搜索结果
  - 为投资组合管理分组相关股票报价
  - 为市场分析建立客户档案
- 为无监督特征提取构建原型样本的代码簿

让我们从创建一个简单的二维人造数据集开始：

```
from sklearn.datasets import make_blobs

X, y = make_blobs(random_state=42)
X.shape

plt.figure(figsize=(8, 8))
plt.scatter(X[:, 0], X[:, 1])
```

在上面的散点图中，我们可以看到三组不同的数据点，我们希望使用聚类来恢复它们 - 想一想“检测”类标签，我们在分类任务中认为它们是理所当然的。

即使这些分组在数据中是显而易见的，当数据存在于高维空间中时很难发现它们，我们无法在单个直方图或散点图中可视化。

现在我们将使用最简单的聚类算法之一，K-means。 这是一种迭代算法，其搜索三个簇中心，使得从每个点到其簇中心的距离最小。 K-means 的标准实现使用欧几里德距离，这就是为什么，如果我们使用真实世界的数据集，我们要确保所有变量都以相同的比例进行测量。 在之前的笔记本中，我们讨论了实现这一目标的一种技术，即标准化。

> 问题
>
> 你期望输出看起来像什么？

```
from sklearn.cluster import KMeans

kmeans = KMeans(n_clusters=3, random_state=42)
```

我们可以通过调用 `fit` 然后访问 `KMeans` 估计器的 `labels_` 属性，或者通过调用 `fit_predict` 来获取簇标签。 无论哪种方式，结果都包含分配给每个点的簇的 ID。

```
labels = kmeans.fit_predict(X)

labels

np.all(y == labels)
```

让我们可视化已发现的分配：

```
plt.figure(figsize=(8, 8))
plt.scatter(X[:, 0], X[:, 1], c=labels)
```

与真实标签相比：

```
plt.figure(figsize=(8, 8))
plt.scatter(X[:, 0], X[:, 1], c=y)
```

在这里，我们可能对聚类结果感到满意。 但总的来说，我们可能希望具有更加量化的评估。 如何将我们的簇标签与生成 blob 时得到的真实情况进行比较？

```
from sklearn.metrics import confusion_matrix, accuracy_score

print('Accuracy score:', accuracy_score(y, labels))
print(confusion_matrix(y, labels))

np.mean(y == labels)
```

> 练习 在查看“真实”标签数组 `y`，以及上面的散点图和 `labels` 之后，你能理解为什么我们的计算精度为 0.0 而不是 1.0 吗，你能解决它吗？

即使我们完全恢复了数据的簇划分，我们分配的簇 ID 也是任意的，我们不能希望恢复它们。 因此，我们必须使用不同的评分指标，例如 `adjusted_rand_score`，它对标签的排列不变：

```
from sklearn.metrics import adjusted_rand_score

adjusted_rand_score(y, labels)
```

K-means 的“缺点”之一是我们必须指定簇的数量，这是我们通常事先不知道的。 例如，让我们看一下如果我们在人造 3-blob 数据集中，将簇数设置为 2 会发生什么：

```
kmeans = KMeans(n_clusters=2, random_state=42)
labels = kmeans.fit_predict(X)
plt.figure(figsize=(8, 8))
plt.scatter(X[:, 0], X[:, 1], c=labels)

kmeans.cluster_centers_
```

## Elbow 方法

Elbow方法是一种“经验法则”，用于查找最佳簇数。 在这里，我们看一下不同 k 值的聚类散度：

```
distortions = []
for i in range(1, 11):
    km = KMeans(n_clusters=i,
                random_state=0)
    km.fit(X)
    distortions.append(km.inertia_)

plt.plot(range(1, 11), distortions, marker='o')
plt.xlabel('Number of clusters')
plt.ylabel('Distortion')
plt.show()
```

然后，我们选择类似“elbow 的凹陷”的值。 我们可以看到，在这种情况下，这是 `k = 3` ，给定我们先前对数据集的视觉预期，这是有意义的。

聚类具有以下假设：聚类算法通过假设样本应该分组到一起，来找到簇。 每种算法都会做出不同的假设，结果的质量和可解释性将取决于你的目标是否满足假设。 对于 K 均值聚类，模型是所有簇具有相等的球形方差。

通常，无法保证聚类算法找到的结构，与你感兴趣的内容有任何关系。

我们可以轻松地创建一个数据集，具有非各向同性的簇的，其中 kmeans 将失败：

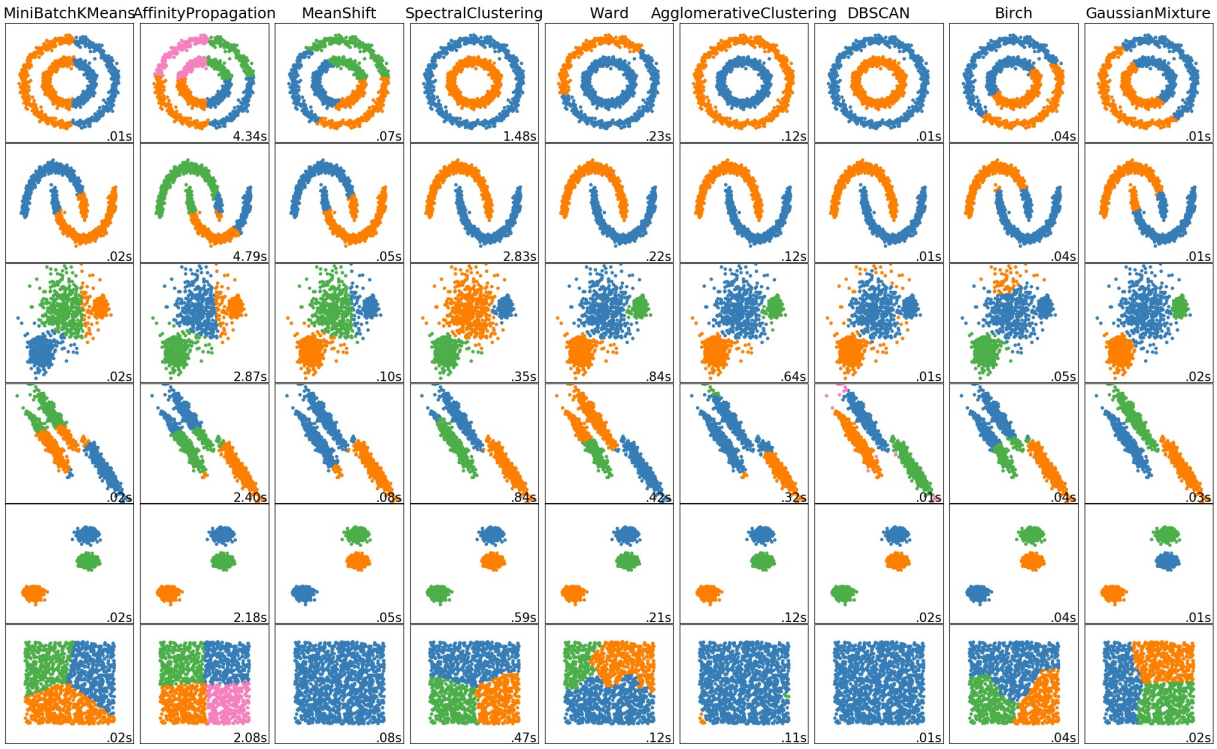
```
plt.figure(figsize=(12, 12))

n_samples = 1500
random_state = 170
X, y = make_blobs(n_samples=n_samples, random_state=random_state
)

# 簇的数量不正确
y_pred = KMeans(n_clusters=2, random_state=random_state).fit_pre
dict(X)

plt.subplot(221)
plt.scatter(X[:, 0], X[:, 1], c=y_pred)
plt.title("Incorrect Number of Blobs")

# 各向异性分布的数据
transformation = [[0.60834549, -0.63667341], [-0.40887718, 0.852
53229]]
X_aniso = np.dot(X, transformation)
y_pred = KMeans(n_clusters=3, random_state=random_state).fit_pre
dict(X_aniso)

plt.subplot(222)
plt.scatter(X_aniso[:, 0], X_aniso[:, 1], c=y_pred)
plt.title("Anisotropicly Distributed Blobs")

# 不同的方差
X_varied, y_varied = make_blobs(n_samples=n_samples,
                                cluster_std=[1.0, 2.5, 0.5],
                                random_state=random_state)
y_pred = KMeans(n_clusters=3, random_state=random_state).fit_pre
dict(X_varied)

plt.subplot(223)
plt.scatter(X_varied[:, 0], X_varied[:, 1], c=y_pred)
plt.title("Unequal Variance")

# 大小不均匀的 blobs
X_filtered = np.vstack((X[y == 0][:500], X[y == 1][:100], X[y ==
2][:10]))
y_pred = KMeans(n_clusters=3,
                random_state=random_state).fit_predict(X_filtere
d)

plt.subplot(224)
plt.scatter(X_filtered[:, 0], X_filtered[:, 1], c=y_pred)
plt.title("Unevenly Sized Blobs")
```

## 一些值得注意的聚类例程

以下是两种众所周知的聚类算法。

- `sklearn.cluster.KMeans`：最简单但有效的聚类算法。需要事先提供簇数，并假设数据为输入而标准化（但使用 PCA 模型作为预处理器）。
- `sklearn.cluster.MeanShift`：可以找到比 KMeans 更好看的簇，但不能扩展到大量样本。
- `sklearn.cluster.DBSCAN`：可以基于密度检测不规则形状的簇，即输入空间中的稀疏区域可能变成簇间边界。还可以检测异常值（不属于簇的样本）。
- `sklearn.cluster.AffinityPropagation`：基于数据点间消息传递的聚类算法。
- `sklearn.cluster.SpectralClustering`：应用于归一化图拉普拉斯算子的投影的 KMeans：如果亲和度矩阵被解释为图的邻接矩阵，则找到归一化的图切割。
- `sklearn.cluster.Ward`：`Ward` 基于 `Ward` 算法实现层次聚类，`Ward` 算法是一种方差最小化方法。在每个步骤中，它最小化所有簇内的平方差异的总和（惯性标准）。

其中，`Ward`，`SpectralClustering`，`DBSCAN` 和 `AffinityPropagation` 也可以与预先计算的相似性矩阵一起使用。




> 练习：数字聚类
>
> 对数字数据执行 K-means 聚类，搜索十个簇。将簇中心可视化为图像（即，将每个聚类中心形状变为 8x8 并使用 `plt.imshow`。）簇是否与特定数字相关？什么是 `adjusted_rand_score`？可视化上个笔记本中的投影数字，但这次使用簇标签作为颜色。你注意到了什么？

```
from sklearn.datasets import load_digits
digits = load_digits()
# ...

# %load solutions/08B_digits_clustering.py
```

# 九、sklearn 估计器接口回顾

Scikit-learn 努力在为所有方法建立统一的接口。给定名为 `model` 的 scikit-learn 估计器对象，可以使用以下方法（并非每个模型都有）：

适用于所有估计器 `model.fit()` ：拟合训练数据。对于监督学习应用，它接受两个参数：数据 `X` 和标签 `y` （例如 `model.fit(X, y)` ）。对于无监督学习应用， `fit` 仅接受单个参数，即数据 `X` （例如 `model.fit(X)` ）。 可在监督估计器中使用 `model.predict()` ：给定训练好的模型，预测一组新数据的标签。此方法接受一个参数，即新数据 `X_new` （例如 `model.predict(X_new)` ），并返回数组中每个对象的习得标签。 `model.predict_proba()` ：对于分类问题，一些估计器也提供此方法，该方法返回新观测值具有每个分类标签的概率。在这种情况下， `model.predict()` 返回概率最高的标签。
`model.decision_function()` ：对于分类问题，一些估计器提供不是概率的不确定性估计。对于二分类， `decision_function >= 0` 表示将预测为正类，而 `<0` 表示负类。 `model.score()` ：对于分类或回归问题，大多数（所有？）估计器实现了 `score` 方法。分数在 0 到 1 之间，分数越大表示拟合越好。对于分类器，分数方法计算预测的准确度。对于回归器，得分计算预测的确定系数（R^2）。 `model.transform()` ：对于特征选择算法，这会将数据集缩减为所选特征。对于某些分类和回归模型（如某些线性模型和随机森林），此方法可将数据集缩减为信息量最大的特征。因此，这些分类和回归模型也可以用作特征选择方法。 可在无监督的估算器中使用 `model.transform()` ：给定一个无监督的模型，将新数据转换为新的基。这也接受一个参数 `X_new` ，并根据无监督模型返回数据的新表示。 `model.fit_transform()` ：一些估计器实现了这个方法，它可以更有效地对相同的输入数据执行拟合和变换。 `model.predict()` ：对于聚类算法， `predict` 方法将为新数据点生成簇标签。并非所有聚类方法都具有此函数。
`model.predict_proba()` ：高斯混合模型（GMM）提供给定混合成分生成每个点的概率。 `model.score()` ：像 KDE 和 GMM这 样的密度模型，提供了数据在模型下的似然。

除了 `fit` 之外，两个最重要的函数是 `produce` ，它产生目标变量（ `y` ），以及 `transform` ，它产生数据的新表示（ `X` ）。 下表展示了哪个函数适用于哪种的模型：

| `model.predict` | `model.transform` |
| --- | --- |
| 分类 | 预处理 |
| 回归 | 降维 |
| 聚类 | 特征提取 |
| | 特征选择 |

# 十、案例学习：泰坦尼克幸存者

## 特征提取

在这里，我们将讨论一个重要的机器学习：从数据中提取定量特征。 到本节结束时，你将

- 了解如何从现实世界数据中提取特征。
- 请参阅从文本数据中提取数值特征的示例

此外，我们将介绍 scikit-learn 中的几个基本工具，可用于完成上述任务。

## 特征是什么？

## 数值特征

回想一下 scikit-learn 中的数据应该是二维数组，大小为 `n_samples×n_features` 。

以前，我们查看了鸢尾花数据集，它有 150 个样本和 4 个特征。

```
from sklearn.datasets import load_iris

iris = load_iris()
print(iris.data.shape)
```

这些特征是：

- 萼片长度，厘米
- 萼片宽度，厘米
- 花瓣长度，厘米
- 花瓣宽度，厘米

诸如此类的数值特征非常简单：每个样本都包含对应特征的浮点数列表。

## 类别特征

如果你有类别特征怎么办？ 例如，假设每个鸢尾花的颜色数据为：

```
color in [red, blue, purple]
```

> 译者注：这是个不恰当的例子，因为在计算机看来，颜色是离散的数值特征，拥有 RGB 三个分量。

你可能想为这些特征分配数字，即红色为 1，蓝色为 2，紫色为 3，但总的来说这是一个坏主意。 估计器倾向于假设，数值特征具有某些连续尺度，因此，例如，1 和 2 比 1 和 3 更相似，并且这通常不是类别特征的情况。

实际上，上面的例子是“类别”特征的子类别，即“标称”特征。 标称特征并不意味着有序，而“序数”特征是确实暗示顺序的分类特征。 序数特征的一个例子是 T 恤尺寸，例如 `XL> L> M> S` 。

将标称特征解析为防止分类算法断言顺序的格式的一种解决方法，是所谓的单热编码表示。 在这里，我们为每个类别提供自己的维度。

因此，在这种情况下，丰富的鸢尾花特征集是：

- 萼片长度，厘米
- 萼片宽度，厘米
- 花瓣长度，厘米
- 花瓣宽度，厘米
- 颜色为紫色（1.0 或 0.0）
- 颜色为红色（1.0 或 0.0）
- 颜色为蓝色（1.0 或 0.0）

请注意，使用许多这些类别特征可能会产生更好表示为稀疏矩阵的数据，我们将在下面的文本分类示例中看到。

## 使用 `DictVectorizer` 编码分类特征

当要编码的源数据有一个 `dicts` 列表，其中值是类别或数值的字符串名称时，你可以使用 `DictVectorizer` 类计算类别特征的布尔扩展，同时保持数值特征不受影响：

```
measurements = [
    {'city': 'Dubai', 'temperature': 33.},
    {'city': 'London', 'temperature': 12.},
    {'city': 'San Francisco', 'temperature': 18.},
]

from sklearn.feature_extraction import DictVectorizer

vec = DictVectorizer()
vec

vec.fit_transform(measurements).toarray()

vec.get_feature_names()
```

## 衍生特征

另一个常见的特征类型是衍生特征，其中一些预处理步骤应用于数据来生成以某种方式提供更多信息的特征。 派生特征可以基于特征提取和降维（例如 PCA 或流形学习），可以是特征的线性或非线性组合（例如在多项式回归中），或者可以是特征的一些更复杂的变换。

## 组合数值和类别特征

作为如何使用分类和数字数据的一个例子，我们将为 HMS 泰坦尼克号的乘客进行生存预测。

我们将使用泰坦尼克号（ `titanic3.xls` ）这里的版本。 我们将 `.xls` 转换为 `.csv` 以便操作，但是数据保持不变。

我们需要读取（ `titanic3.csv` ）文件中的所有行，空出第一行的键，找到我们的标签（幸存或死亡）和数据（人的属性）。 让我们看看键和一些相应的示例行。

```
import os
import pandas as pd

titanic = pd.read_csv(os.path.join('datasets', 'titanic3.csv'))
print(titanic.columns)
```

以下是键及其含义的广泛描述：

```
pclass          乘客等级
                (1 = 1st; 2 = 2nd; 3 = 3rd)
survival        是否幸存
                (0 = No; 1 = Yes)
name            名称
sex             性别
age             年龄
sibsp           船上的兄弟姐妹/配偶的数量
parch           父母/孩子数量
ticket          票号
fare            乘客票价
cabin           舱位
embarked        登船港口
                (C = Cherbourg; Q = Queenstown; S = Southampton)
boat            救生艇
body            身份证号
home.dest       家/目的地
```

一般来说，`name`，`sex`，`cabin`，`embarked`，`boat`，`body`，`homedest`可能是类别特征的候选，而其余似乎是数组特征。我们还可以查看数据集中的前几行以便更好地理解：

```
titanic.head()
```

我们显然希望丢弃`boat`和`body`列，以便将任何人分类为幸存者和非幸存者，因为它们已经包含此信息。`name`对每个人（可能）是唯一的，也是非信息性的。首次尝试中，我们将使用`pclass`，`sibsp`，`parch`，`fare`和`embarked`作为我们的特征：

```
labels = titanic.survived.values
features = titanic[['pclass', 'sex', 'age', 'sibsp', 'parch', 'f
are', 'embarked']]

features.head()
```

数据现在仅包含有用的特征，但它们不是机器学习算法可以理解的格式。我们需要将字符串`male`和`female`转换为表示性别的二元变量，类似于`embarked`。我们可以使用`pandas get_dummies`函数来实现：

```
pd.get_dummies(features).head()
```

这个转换成功编码了字符串列。但是，有人可能会认为`pclass`也是一个类别变量。我们可以使用`columns`参数显式列出要编码的列，并包含`pclass`：

```
features_dummies = pd.get_dummies(features, columns=['pclass', '
sex', 'embarked'])
features_dummies.head(n=16)

data = features_dummies.values

import numpy as np
np.isnan(data).any()
```

完成了所有困难的数据加载工作，对这些数据应用分类器变得简单明了。建立最简单的模型，我们希望使用`DummyClassifier`看到最简单的得分。

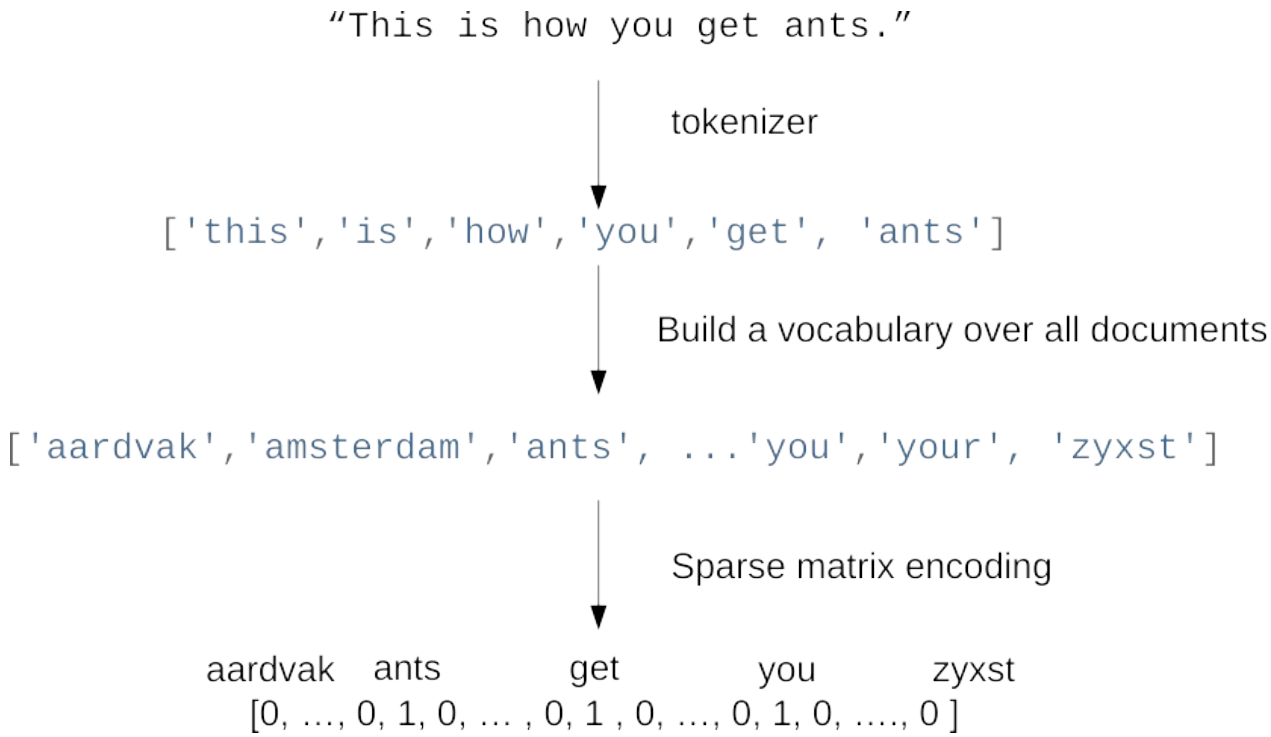
```
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import Imputer


train_data, test_data, train_labels, test_labels = train_test_sp
lit(
    data, labels, random_state=0)

imp = Imputer()
imp.fit(train_data)
train_data_finite = imp.transform(train_data)
test_data_finite = imp.transform(test_data)

np.isnan(train_data_finite).any()

from sklearn.dummy import DummyClassifier

clf = DummyClassifier('most_frequent')
clf.fit(train_data_finite, train_labels)
print("Prediction accuracy: %f"
      % clf.score(test_data_finite, test_labels))
```

> 练习
>
> 尝试使用 `LogisticRegression` 和 `RandomForestClassifier` 而不是 `DummyClassifier` 执行上述分类选择不同的特征子集会有帮助吗？

```
# %load solutions/10_titanic.py
```

# 十一、文本特征提取

在许多任务中，例如在经典的垃圾邮件检测中，你的输入数据是文本。 长度变化的自由文本与我们需要使用 scikit-learn 来做机器学习所需的，长度固定的数值表示相差甚远。 但是，有一种简单有效的方法，使用所谓的词袋模型将文本数据转换为数字表示，该模型提供了与 scikit-learn 中的机器学习算法兼容的数据结构。




假设数据集中的每个样本都表示为一个字符串，可以只是句子，电子邮件或整篇新闻文章或书籍。 为了表示样本，我们首先将字符串拆分为一个标记列表，这些标记对应于（有些标准化的）单词。 一种简单的方法，只需按空白字符分割，然后将单词变为小写。

然后，我们构建了一个所有标记（小写单词）的词汇表，标记出现在我们整个数据集中。 这通常是一个非常大的词汇表。 最后，看一下我们的单个样本，我们可以展示词汇表中每个单词出现的频率。 我们用向量表示我们的字符串，其中每个条目是词汇表中给定单词出现在字符串中的频率。

由于每个样本仅包含非常少的单词，因此大多数条目将为零，从而产生非常高维但稀疏的表示。

该方法被称为“词袋”，因为单词的顺序完全丢失。

```
X = ["Some say the world will end in fire,",
     "Some say in ice."]

len(X)

from sklearn.feature_extraction.text import CountVectorizer

vectorizer = CountVectorizer()
vectorizer.fit(X)

vectorizer.vocabulary_

X_bag_of_words = vectorizer.transform(X)

X_bag_of_words.shape

X_bag_of_words

X_bag_of_words.toarray()

vectorizer.get_feature_names()

vectorizer.inverse_transform(X_bag_of_words)
```

## TF-IDF 编码

通常应用于词袋编码的有用变换，是所谓的“词频-逆文档频率”（tf-idf）缩放，其是单词计数的非线性变换。

tf-idf 编码重缩放通常具有较少权重的单词：

```
from sklearn.feature_extraction.text import TfidfVectorizer

tfidf_vectorizer = TfidfVectorizer()
tfidf_vectorizer.fit(X)

import numpy as np
np.set_printoptions(precision=2)

print(tfidf_vectorizer.transform(X).toarray())
```

tf-idf 是一种将文档表示为特征向量的方法。 tf-idf 可以理解为原始词频（tf）的修改；tf 是给定文档中特定单词的出现频率。 tf-idf 背后的概念是。按照它们出现的文档数量，成比例减少词频的权重。 这里的想法是，对于文档分类等自然语言处理任务，许多不同文档中出现的单词可能不重要，或不包含任何有用的信息。 如果你对数学细节和方程感兴趣，请参阅此外部 IPython Notebook，它将引导你完成计算。

# Bigram 和 N-Gram

在本笔记本开头的图中所示的示例中，我们使用了所谓的 1-gram（unigram）分词：每个标记表示关于分割标准的单个元素。

完全抛弃单词顺序并不总是一个好主意，因为复合短语通常具有特定含义，而像“not”这样的修饰语可以颠倒单词的含义。

包含一些单词顺序的简单方法是 n-gram，它不仅查看单个标记，而且查看所有相邻标记对。例如，在 2-gram（bigram）分词中，我们使用一个单词的重叠将单词组合在一起；在 3-gram（trigram）分割中，我们将创建两个单词的重叠，依此类推：

- 原文："this is how you get ants"
- 1-gram："this", "is", "how", "you", "get", "ants"
- 1-gram："this", "is", "how", "you", "get", "ants"
- 2-gram："this is", "is how", "how you", "you get", "get ants"
- 3-gram："this is how", "is how you", "how you get", "you get ants"

为了在我们的预测模型中获得最佳效果，我们选择哪个“n”用于“n-gram”分词取决于学习算法，数据集和任务。或者换句话说，我们将“n-gram”中的“n”视为需要调整的参数，在后面的笔记本中，我们将看到我们如何处理它们。

现在，让我们使用 scikit-learn 的 `CountVectorizer` 创建一个 bigram 的词袋模型：

```
# look at sequences of tokens of minimum length 2 and maximum le
ngth 2
bigram_vectorizer = CountVectorizer(ngram_range=(2, 2))
bigram_vectorizer.fit(X)

bigram_vectorizer.get_feature_names()

bigram_vectorizer.transform(X).toarray()
```

通常我们想要包括 unigram（单个标记）和 bigram，我们可以将以下元组作为参数传递给 `CountVectorizer` 函数的 `ngram_range` 参数：

```
gram_vectorizer = CountVectorizer(ngram_range=(1, 2))
gram_vectorizer.fit(X)

gram_vectorizer.get_feature_names()

gram_vectorizer.transform(X).toarray()
```

# 字符 n-gram

有时不仅是查看单词，考虑单个字符也有帮助。

如果我们有非常嘈杂的数据并想要识别语言，或者我们想要预测一个单词的某些内容，那么这一点尤其有用。 我们可以通过设置 `analyzer ="char"` 来简单地查看字符而不是单词。 查看单个字符通常不是很有用，但是查看更长的 `n` 个字符可能是：

```
X

char_vectorizer = CountVectorizer(ngram_range=(2, 2), analyzer="
char")
char_vectorizer.fit(X)

print(char_vectorizer.get_feature_names())
```

> 练习
>
> 从下面给出（或者通过 `import this` ）的“zen of python”中计算 bigrams，并找到最常见的 trigram。 我们希望将每一行视为单独的文档。 你可以通过按照换行符（ `\ n` ）分割字符串来实现。 计算数据的 Tf-idf 编码。 哪个词的 tf-idf 得分最高？ 为什么？ 如果使用 `TfidfVectorizer(norm="none")` 会有什么变化？

```
zen = """Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to d
o it.
Although that way may not be obvious at first unless you're Dutc
h.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
"""

# %load solutions/11_ngrams.py
```

# 十二、案例学习：用于 SMS 垃圾检测的文本分类

我们首先从 `dataset` 目录中加载文本数据，该目录应该位于 `notebooks` 目录中，是我们通过从GitHub存储库的顶层运行 `fetch_data.py` 脚本创建的。

此外，我们执行一些简单的预处理并将数据数组拆分为两部分：

`text` ：列表的列表，其中每个子列表包含电子邮件的内容 `y` ：我们的 SPAM 与 HAM 标签，以二元形式存储；1 代表垃圾邮件，0 代表非垃圾邮件消息。

```
import os

with open(os.path.join("datasets", "smsspam", "SMSSpamCollection"
)) as f:
    lines = [line.strip().split("\t") for line in f.readlines()]

text = [x[1] for x in lines]
y = [int(x[0] == "spam") for x in lines]

text[:10]

y[:10]

print('Number of ham and spam messages:', np.bincount(y))

type(text)

type(y)
```

接下来，我们将数据集分为两部分，即测试和训练数据集：

```
from sklearn.model_selection import train_test_split

text_train, text_test, y_train, y_test = train_test_split(text,
y,
                                                          random
_state=42,
                                                          test_s
ize=0.25,
                                                          strati
fy=y)
```

现在，我们使用 `CountVectorizer` 将文本数据解析为词袋模型。

```
from sklearn.feature_extraction.text import CountVectorizer

print('CountVectorizer defaults')
CountVectorizer()

vectorizer = CountVectorizer()
vectorizer.fit(text_train)

X_train = vectorizer.transform(text_train)
X_test = vectorizer.transform(text_test)

print(len(vectorizer.vocabulary_))

X_train.shape

print(vectorizer.get_feature_names()[:20])

print(vectorizer.get_feature_names()[2000:2020])

print(X_train.shape)
print(X_test.shape)
```

## 为文本特征训练分类器

我们现在可以训练分类器，例如逻辑回归分类器，它是文本分类任务的快速基线：

```
from sklearn.linear_model import LogisticRegression

clf = LogisticRegression()
clf

clf.fit(X_train, y_train)
```

我们现在可以在测试集上评估分类器。 让我们首先使用内置得分函数，这是测试集中正确分类的比例：

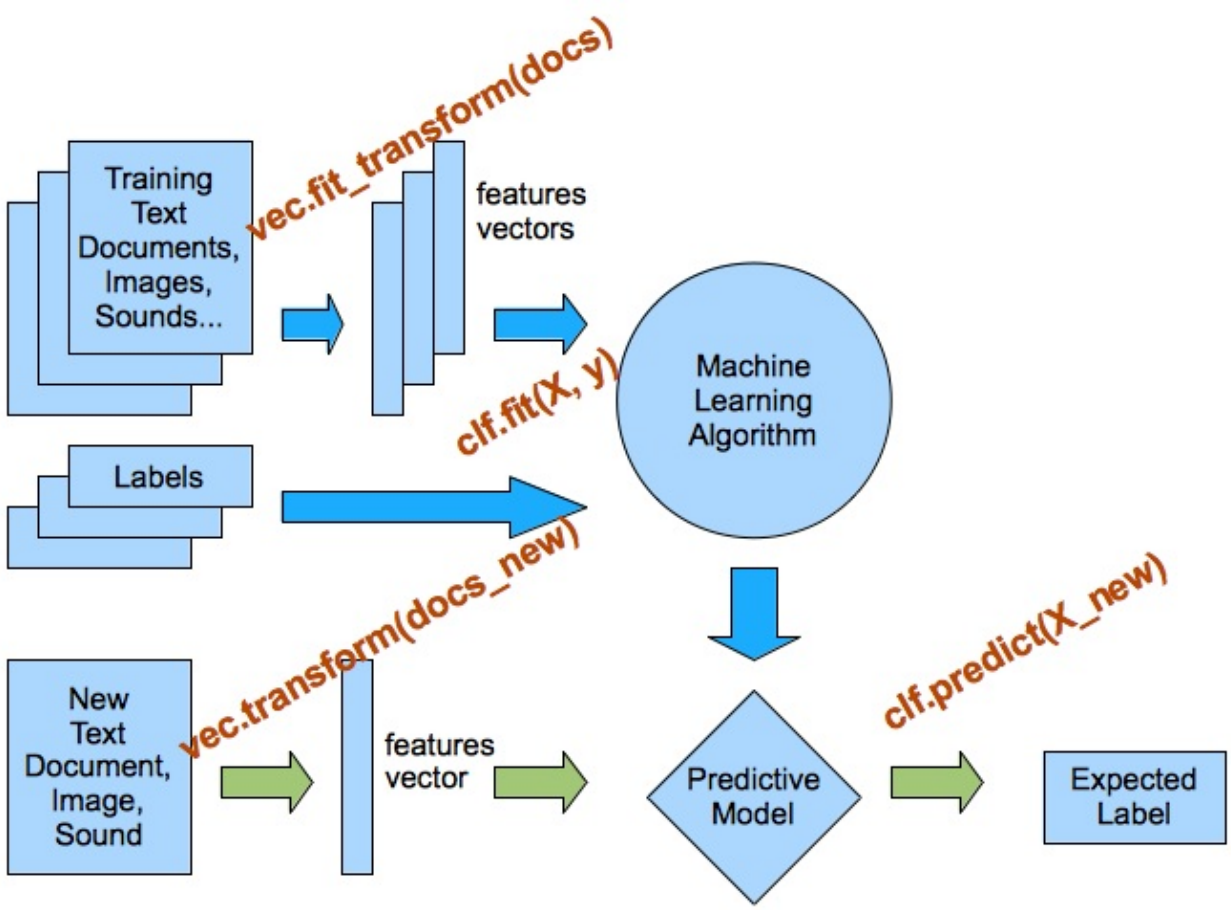
```
clf.score(X_test, y_test)
```

我们还可以计算训练集上的托分，看看我们做得如何：

```
clf.score(X_train, y_train)
```

## 可视化重要特征

```
def visualize_coefficients(classifier, feature_names, n_top_feat
ures=25):
    # get coefficients with large absolute values
    coef = classifier.coef_.ravel()
    positive_coefficients = np.argsort(coef)[-n_top_features:]
    negative_coefficients = np.argsort(coef)[:n_top_features]
    interesting_coefficients = np.hstack([negative_coefficients,
 positive_coefficients])
    # plot them
    plt.figure(figsize=(15, 5))
    colors = ["tab:orange" if c < 0 else "tab:blue" for c in coe
f[interesting_coefficients]]
    plt.bar(np.arange(2 * n_top_features), coef[interesting_coef
ficients], color=colors)
    feature_names = np.array(feature_names)
    plt.xticks(np.arange(1, 2 * n_top_features + 1), feature_nam
es[interesting_coefficients], rotation=60, ha="right");

visualize_coefficients(clf, vectorizer.get_feature_names())

vectorizer = CountVectorizer(min_df=2)
vectorizer.fit(text_train)

X_train = vectorizer.transform(text_train)
X_test = vectorizer.transform(text_test)

clf = LogisticRegression()
clf.fit(X_train, y_train)

print(clf.score(X_train, y_train))
print(clf.score(X_test, y_test))

len(vectorizer.get_feature_names())

print(vectorizer.get_feature_names()[:20])

visualize_coefficients(clf, vectorizer.get_feature_names())
```




> 练习
>
> 使用 `TfidfVectorizer` 而不是 `CountVectorizer` 。结果更好吗？系数如何不同？更改 `TfidfVectorizer` 和 `CountVectorizer` 的参数 `min_df` 和 `ngram_range` 。这如何改变重要特征？

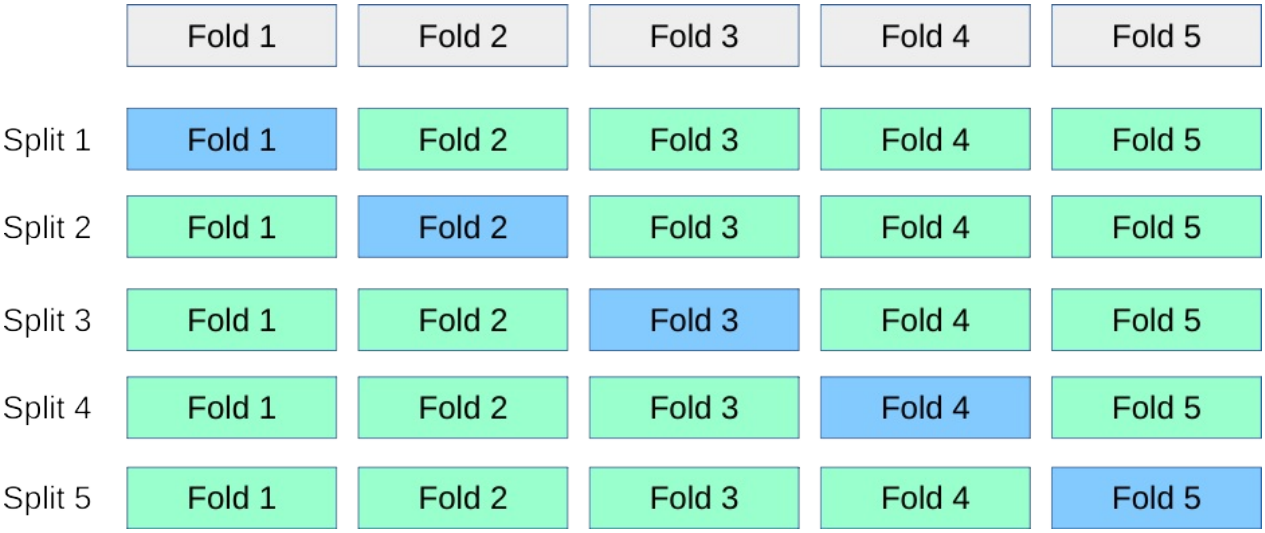
```
# %load solutions/12A_tfidf.py

# %load solutions/12B_vectorizer_params.py
```

# 十三、交叉验证和得分方法

在前面的章节和笔记本中，我们将数据集分为两部分：训练集和测试集。 我们使用训练集来拟合我们的模型，并且我们使用测试集来评估其泛化能力 - 它对新的，没见过的数据的表现情况。


然而，（标记的）数据通常是宝贵的，这种方法让我们只将约 3/4 的数据用于行训练。 另一方面，我们只会尝试将我们的 1/4 数据应用于测试。 使用更多数据来构建模型，并且获得更加鲁棒的泛化能力估计，常用方法是交叉验证。 在交叉验证中，数据被重复拆分为非重叠的训练和测试集，并为每对建立单独的模型。 然后聚合测试集的得分来获得更鲁棒的估计。

进行交叉验证的最常用方法是 `k` 折交叉验证，其中数据首先被分成 `k` （通常是 5 或 10）个相等大小的折叠，然后对于每次迭代，使用 `k` 折中的一个作为测试数据，其余作为训练数据：




这样，每个数据点只在测试集中一次，我们可以使用第 `k` 个数据之外的所有数据进行训练。 让我们应用这种技术，在鸢尾花数据集上评估 `KNeighborsClassifier` 算法：

```
from sklearn.datasets import load_iris
from sklearn.neighbors import KNeighborsClassifier

iris = load_iris()
X, y = iris.data, iris.target

classifier = KNeighborsClassifier()
```

鸢尾花中的标签是有序的，这意味着如果我们像上面那样拆分数据，第一个折叠只有标签 0，而最后一个只有标签 2：

```
y
```

为了在评估中避免这个问题，我们首先将我们的数据打乱：

```
import numpy as np
rng = np.random.RandomState(0)

permutation = rng.permutation(len(X))
X, y = X[permutation], y[permutation]
print(y)
```

现在实现交叉验证很简单：

```
k = 5
n_samples = len(X)
fold_size = n_samples // k
scores = []
masks = []
for fold in range(k):
    # 为此折叠中的测试集生成一个布尔掩码
    test_mask = np.zeros(n_samples, dtype=bool)
    test_mask[fold * fold_size : (fold + 1) * fold_size] = True
    # 为可视化存储掩码
    masks.append(test_mask)
    # 使用此掩码创建训练和测试集
    X_test, y_test = X[test_mask], y[test_mask]
    X_train, y_train = X[~test_mask], y[~test_mask]
    # 拟合分类器
    classifier.fit(X_train, y_train)
    # 计算得分并记录
    scores.append(classifier.score(X_test, y_test))
```

让我们检查一下我们的测试掩码是否正确：

```
import matplotlib.pyplot as plt
%matplotlib inline
plt.matshow(masks, cmap='gray_r')
```

现在让我们看一下我们计算出的得分：

```
print(scores)
print(np.mean(scores))
```

正如你所看到的，得分广泛分布于 90% 正确到 100% 正确。 如果我们只进行一次分割，我们可能会得到任何答案。

由于交叉验证是机器学习中常见的模式，有个函数执行上面的操作，带有更多灵活性和更少代码。 `sklearn.model_selection` 模块具有交叉验证相关的所有函数。 最简单的函数是 `cross_val_score` ，它接受估计器和数据集，并将为你完成所有拆分：

```
from sklearn.model_selection import cross_val_score
scores = cross_val_score(classifier, X, y)
print('Scores on each CV fold: %s' % scores)
print('Mean score: %0.3f' % np.mean(scores))
```

如你所见，该函数默认使用三个折叠。 你可以使用 `cv` 参数更改折叠数：

```
cross_val_score(classifier, X, y, cv=5)
```

交叉验证模块中还有辅助对象，它们将为你生成各种不同交叉验证方法的索引，包括 k-fold：

```
from sklearn.model_selection import KFold, StratifiedKFold, Shuf
fleSplit
```

默认情况下， `cross_val_score` 将 `StratifiedKFold` 用于分类，这可确保数据集中的类比例反映在每个折叠中。 如果你有一个二分类数据集，其中 90% 的数据点属于类 0，那么这意味着在每个折叠中，90% 的数据点将属于类 0。如果你只是使用 `KFold` 交叉验证，你可能会生成一个只包含类 0 的分割。每当你进行分类时，通常最好使用 `StratifiedKFold` 。

`StratifiedKFold` 也消除了我们打乱鸢尾花的需要。 让我们看看在未打乱的鸢尾花数据集上，它生成什么类型的折叠。 每个交叉验证类都是训练和测试索引的集合的生成器：

```
cv = StratifiedKFold(n_splits=5)
for train, test in cv.split(iris.data, iris.target):
    print(test)
```

正如你所看到的，在每个折叠中，在开始，中间，和结束位置，都有一些样本。 这样，保留了类别比例。 让我们观察一下 `split` ：

```
def plot_cv(cv, features, labels):
    masks = []
    for train, test in cv.split(features, labels):
        mask = np.zeros(len(labels), dtype=bool)
        mask[test] = 1
        masks.append(mask)

    plt.matshow(masks, cmap='gray_r')

plot_cv(StratifiedKFold(n_splits=5), iris.data, iris.target)
```

为了比较，仍旧是标准 `KFold` ，忽略标签：

```
plot_cv(KFold(n_splits=5), iris.data, iris.target)
```

请记住，增加折叠数量会为你提供更大的训练数据集，但会导致更多重复，因此评估速度会变慢：

```
plot_cv(KFold(n_splits=10), iris.data, iris.target)
```

另一个有用的交叉验证生成器是 `ShuffleSplit` 。该生成器简单地重复分割数据的随机部分。这允许用户独立指定重复次数和训练集大小：

```
plot_cv(ShuffleSplit(n_splits=5, test_size=.2), iris.data, iris.
target)
```

如果你想要更鲁棒的估计，你可以增加分割数量：

```
plot_cv(ShuffleSplit(n_splits=20, test_size=.2), iris.data, iris
.target)
```

你可以使用 `cross_val_score` 方法来使用所有这些交叉验证生成器：

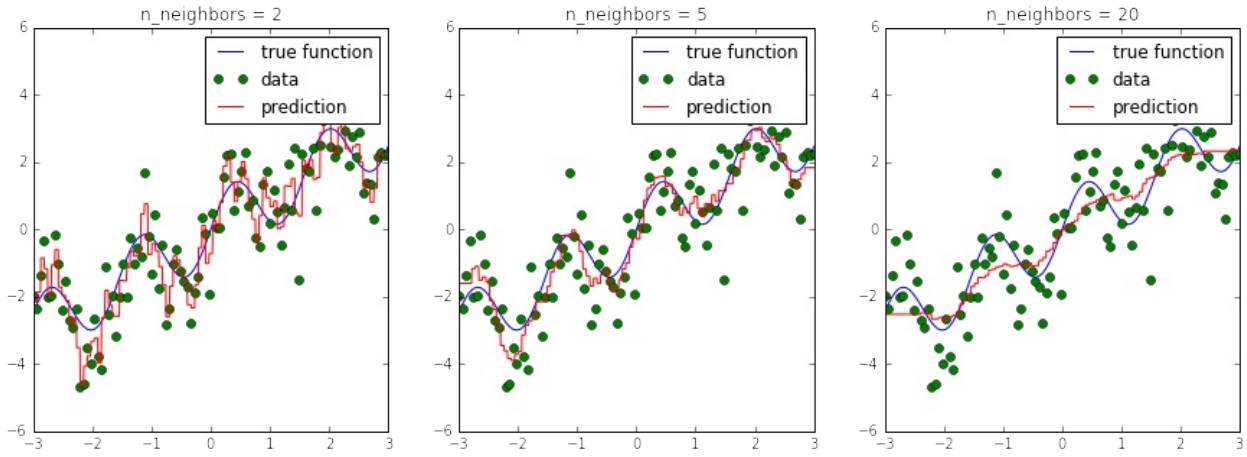
```
cv = ShuffleSplit(n_splits=5, test_size=.2)
cross_val_score(classifier, X, y, cv=cv)
```

> 练习
>
> 在鸢尾花数据集上，使用 `KFold` 类进行三折交叉验证，而不打乱数据。你能解释一下结果吗？

```
# %load solutions/13_cross_validation.py
```

# 十四、参数选择、验证和测试

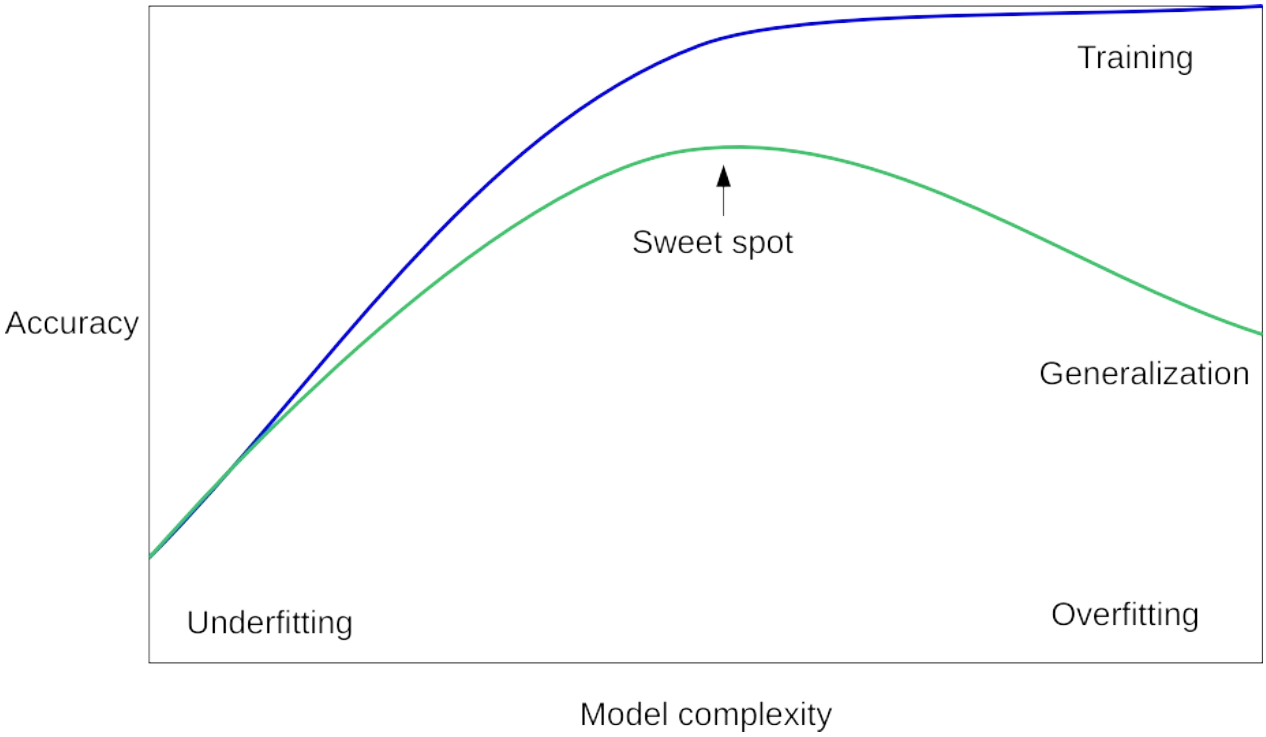
大多数模型的参数会影响他们可以学习的模型的复杂程度。回忆使用 `KNeighborsRegressor` 的时候。 如果我们改变我们考虑的邻居数量，我们会得到更平滑的预测：




在上图中，我们看到 `n_neighbors` 的三个不同值。对于 `n_neighbors = 2` ，数据过拟合，模型过于灵活，可以适应训练数据中的噪声 。对于 `n_neighbors = 20` ，模型不够灵活，无法合理建模数据中的变化。

在中间，对于 `n_neighbors = 5` ，我们找到了一个很好的中点。它非常适合数据，并且不会受到任何一个图中所见的，过拟合或欠拟合问题的影响。我们想要的是一种定量识别过拟合和欠拟合的方法，并优化超参数（这种情况是多项式次数 `d` ）来确定最佳算法。

我们要权衡过多记录训练数据的特殊性和噪声，或者没有建模足够的可变性。这是一个需要在基本上每个机器学习应用中做出的权衡，并且是一个核心概念，称为偏差 - 方差 - 权衡或“过拟合与欠拟合”。




## 超参数、过拟合和欠拟合

遗憾的是，没有找到最佳位置的一般规则，因此机器学习实践者必须通过尝试几个超参数设置，来找到模型复杂性和泛化的最佳权衡。超参数是机器学习算法的内部旋钮或可调参数（与算法从训练数据中学习的模型参数相反 - 例如，线性回归模型的权重系数）；K 近邻中的 `k` 的数量是这样的超参数。

最常见的是，这种“超参数调整”是使用暴力搜索完成的，例如在多个 `n_neighbors` 值上：

```
from sklearn.model_selection import cross_val_score, KFold
from sklearn.neighbors import KNeighborsRegressor
# 生成玩具数据集
x = np.linspace(-3, 3, 100)
rng = np.random.RandomState(42)
y = np.sin(4 * x) + x + rng.normal(size=len(x))
X = x[:, np.newaxis]

cv = KFold(shuffle=True)

# 对每个参数设置执行交叉验证
for n_neighbors in [1, 3, 5, 10, 20]:
    scores = cross_val_score(KNeighborsRegressor(n_neighbors=n_n
eighbors), X, y, cv=cv)
    print("n_neighbors: %d, average score: %f" % (n_neighbors, n
p.mean(scores)))
```

scikit-learn 中有一个函数，称为 `validation_plot` ，用于重现上面的卡通图。 它根据训练和验证误差（使用交叉验证）绘制一个参数，例如邻居的数量：

```
from sklearn.model_selection import validation_curve
n_neighbors = [1, 3, 5, 10, 20, 50]
train_scores, test_scores = validation_curve(KNeighborsRegressor
(), X, y, param_name="n_neighbors",
                                             param_range=n_neigh
bors, cv=cv)
plt.plot(n_neighbors, train_scores.mean(axis=1), label="train ac
curacy")
plt.plot(n_neighbors, test_scores.mean(axis=1), label="test accu
racy")

plt.ylabel('Accuracy')
plt.xlabel('Number of neighbors')
plt.xlim([50, 0])
plt.legend(loc="best");
```

请注意，许多邻居意味着“平滑”或“简单”的模型，因此绘图使用还原的 `x` 轴。

如果多个参数很重要，例如 SVM 中的参数 `C` 和 `gamma` （稍后会详细介绍），则尝试所有可能的组合：

```
from sklearn.model_selection import cross_val_score, KFold
from sklearn.svm import SVR

# 对每个参数设置执行交叉验证
for C in [0.001, 0.01, 0.1, 1, 10]:
    for gamma in [0.001, 0.01, 0.1, 1]:
        scores = cross_val_score(SVR(C=C, gamma=gamma), X, y, cv
=cv)
        print("C: %f, gamma: %f, average score: %f" % (C, gamma,
 np.mean(scores)))
```

由于这是一种非常常见的模式，因此在 scikit-learn 中有一个内置类 `GridSearchCV` 。 `GridSearchCV` 接受描述应该尝试的参数的字典，和一个要训练的模型。

参数网格被定义为字典，其中键是参数，值是要测试的设置。

要检查不同折叠的训练得分，请将参数 `return_train_score` 设置为 `True` 。

```
from sklearn.model_selection import GridSearchCV
param_grid = {'C': [0.001, 0.01, 0.1, 1, 10], 'gamma': [0.001, 0
.01, 0.1, 1]}

grid = GridSearchCV(SVR(), param_grid=param_grid, cv=cv, verbose=
3, return_train_score=True)
```

`GridSearchCV` 的一大优点是它是一个元估计器。它需要像上面的 SVR 这样的估计器，并创建一个新的估计器，其行为完全相同 - 在这种情况下，就像一个回归器。所以我们可以调用它的 `fit` 来训练：

```
grid.fit(X, y)
```

`fit` 所做的比我们上面做的复杂得多。首先，它使用交叉验证运行相同的循环，来找到最佳参数组合。一旦它具有最佳组合，它在所有传给 `fit` 的数据上再次执行 `fit`（无交叉验证），来使用最佳参数设置构建单个新模型。

然后，与所有模型一样，我们可以使用 `predict` 或 `score`：

```
grid.predict(X)
```

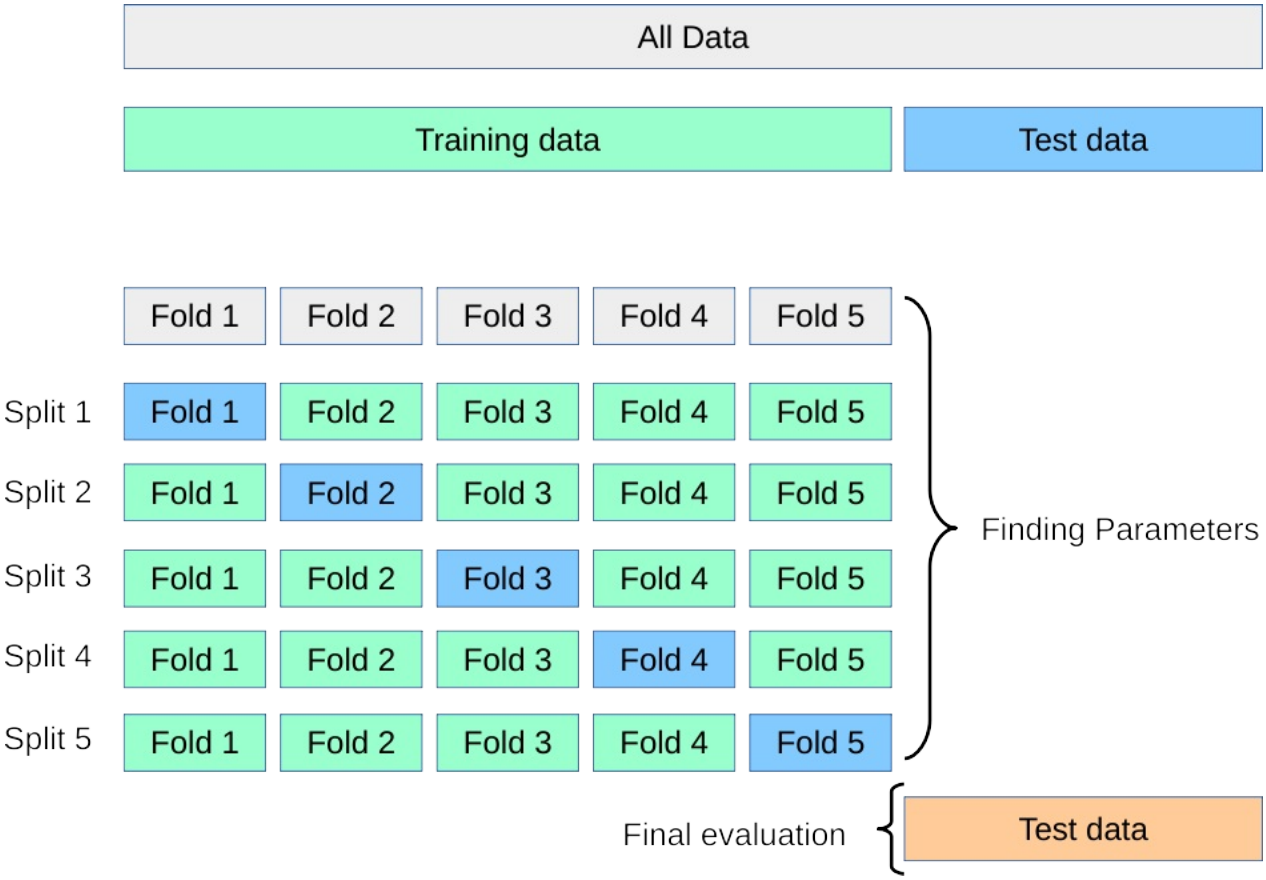
你可以在 `best_params_` 属性中检查 `GridSearchCV` 找到的最佳参数，以及 `best_score_` 属性中的最佳得分：

```
print(grid.best_score_)

print(grid.best_params_)
```

但是，你可以通过访问 `cv_results_` 属性来调查每组参数值的表现和更多信息。`cv_results_` 属性是一个字典，其中每个键都是字符串，每个值都是数组。因此，它可以用于制作 `pandas DataFrame`。

```
type(grid.cv_results_)

print(grid.cv_results_.keys())

import pandas as pd

cv_results = pd.DataFrame(grid.cv_results_)
cv_results.head()

cv_results_tiny = cv_results[['param_C', 'param_gamma', 'mean_te
st_score']]
cv_results_tiny.sort_values(by='mean_test_score', ascending=False
).head()
```

但是，将这个得分用于评估存在问题。 你可能会犯所谓的多假设检验错误。 如果你尝试了很多参数设置，其中一些参数设置只是偶然表现很好，而你获得的得分可能无法反映你的模型对新的没见过的数据的表现。 因此，在执行网格搜索之前拆分单独的测试集是很好的。 这种模式可以看作是训练-验证-测试分割，在机器学习中很常见：




我们可以非常容易地实现，通过使用 `train_test_split` 分割一些测试数据，在训练集上训练 `GridSearchCV` ，并将 `score` 方法应用于测试集：

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, random
_state=1)

param_grid = {'C': [0.001, 0.01, 0.1, 1, 10], 'gamma': [0.001, 0
.01, 0.1, 1]}
cv = KFold(n_splits=10, shuffle=True)

grid = GridSearchCV(SVR(), param_grid=param_grid, cv=cv)

grid.fit(X_train, y_train)
grid.score(X_test, y_test)
```

我们还可以查看所选的参数：

```
grid.best_params_
```

一些实践者采用更简单的方案，将数据简单地分为三个部分，即训练，验证和测试。如果你的训练集非常大，或者使用交叉验证训练许多模型是不可行的，因为训练模型需要很长时间，这是一种可能的替代方案。你可以使用 scikit-learn 执行此操作，例如通过拆分测试集，然后将 `GridSearchCV` 与 `ShuffleSplit` 交叉验证应用于单次迭代：


```
from sklearn.model_selection import train_test_split, ShuffleSpl
it

X_train, X_test, y_train, y_test = train_test_split(X, y, random
_state=1)

param_grid = {'C': [0.001, 0.01, 0.1, 1, 10], 'gamma': [0.001, 0
.01, 0.1, 1]}
single_split_cv = ShuffleSplit(n_splits=1)

grid = GridSearchCV(SVR(), param_grid=param_grid, cv=single_spli
t_cv, verbose=3)

grid.fit(X_train, y_train)
grid.score(X_test, y_test)
```

这要快得多，但可能会产生更糟糕的超参数，从而产生更糟糕的结果。

```
clf = GridSearchCV(SVR(), param_grid=param_grid)
clf.fit(X_train, y_train)
clf.score(X_test, y_test)
```

> 练习
>
> 应用网格搜索来查找`KNeighborsClassifier`中邻居数量的最佳设置，并将其应用于数字数据集。

# 十五、估计器流水线

在本节中，我们将研究如何链接不同的估计器。

## 简单示例：估计器之前的特征提取和选择

## 特征提取：向量化器

对于某些类型的数据，例如文本数据，必须应用特征提取步骤将其转换为数值特征。 为了说明，我们加载我们之前使用的 SMS 垃圾邮件数据集。

```
import os

with open(os.path.join("datasets", "smsspam", "SMSSpamCollection"
)) as f:
    lines = [line.strip().split("\t") for line in f.readlines()]
text = [x[1] for x in lines]
y = [x[0] == "ham" for x in lines]

from sklearn.model_selection import train_test_split

text_train, text_test, y_train, y_test = train_test_split(text,
y)
```

以前，我们手动应用了特征提取，如下所示：

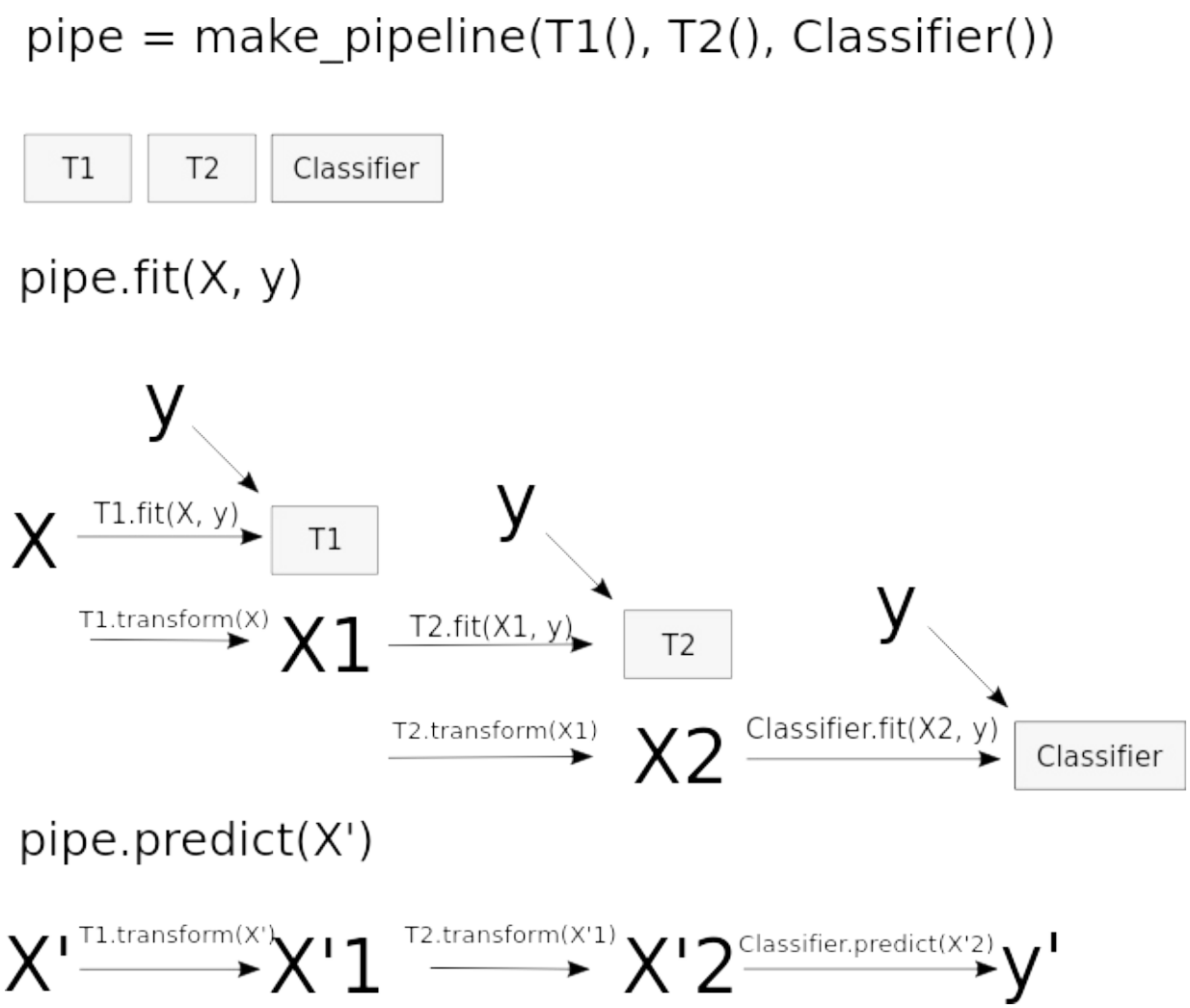
```
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.linear_model import LogisticRegression

vectorizer = TfidfVectorizer()
vectorizer.fit(text_train)

X_train = vectorizer.transform(text_train)
X_test = vectorizer.transform(text_test)

clf = LogisticRegression()
clf.fit(X_train, y_train)

clf.score(X_test, y_test)
```

我们学习转换然后将其应用于测试数据的情况，在机器学习中非常常见。 因此 scikit-learn 有一个快捷方式，称为流水线：

```
from sklearn.pipeline import make_pipeline

pipeline = make_pipeline(TfidfVectorizer(), LogisticRegression()
)
pipeline.fit(text_train, y_train)
pipeline.score(text_test, y_test)
```

如你所见，这使代码更短，更容易处理。 在背后，与上面完全相同。 当在水流上调用 `fit` 时，它将依次调用每个步骤的 `fit` 。

在第一步的 `fit` 之后，它将使用第一步的 `transform` 方法来创建新的表示。 然后将其用于下一步的 `fit` ，依此类推。 最后，在最后一步，只调用 `fit` 。




如果我们调用 `score` ，那么每一步都只会调用 `transform` - 毕竟这可能是测试集！ 然后，在最后一步，使用新的表示调用 `score` 。 `predict` 也是如此。

流水线的构建不仅简化了代码，而且对于模型选择也很重要。 假设我们想要网格搜索 `C` 来调整上面的 Logistic 回归。

让我们假设我们这样做：

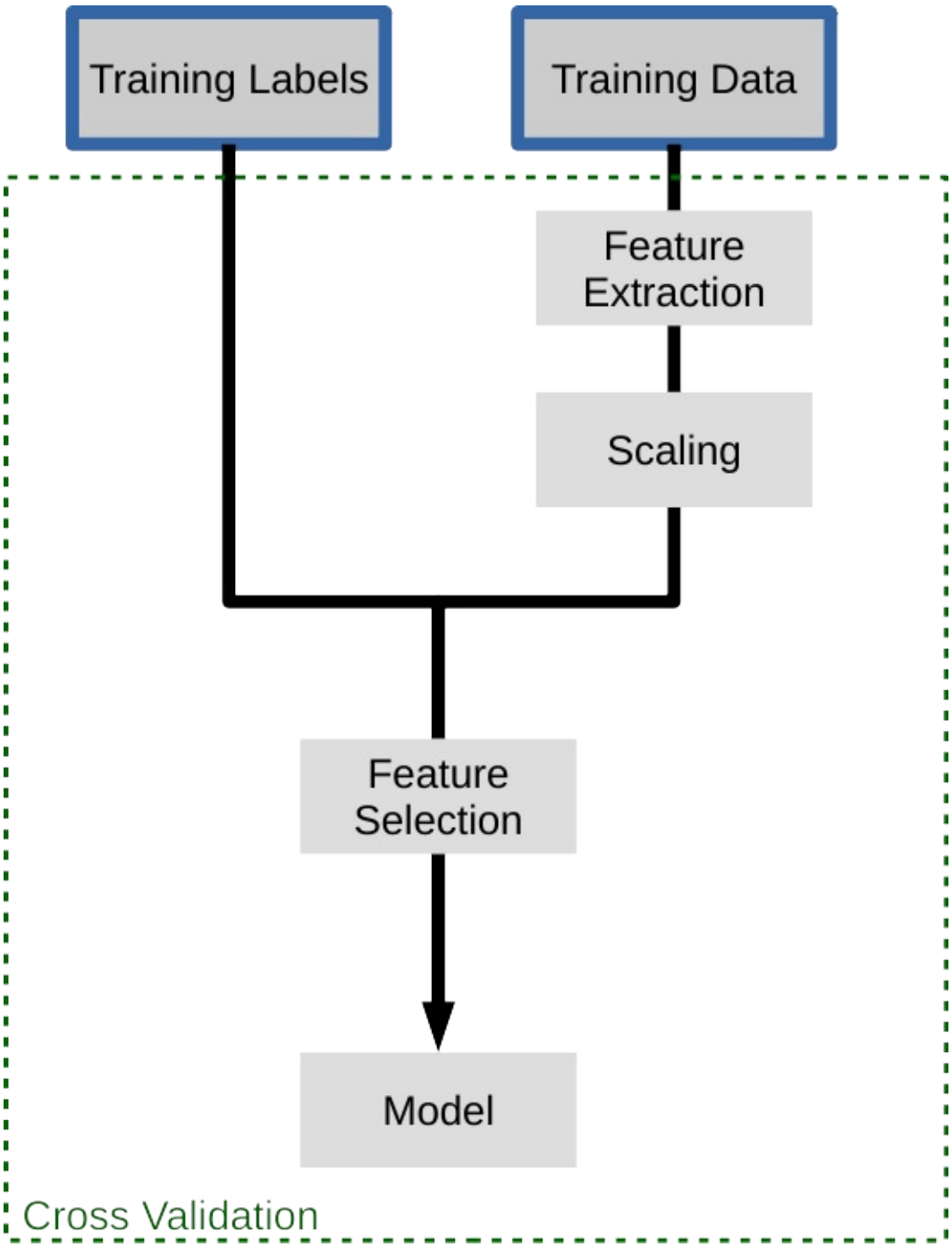
```
# This illustrates a common mistake. Don't use this code!
from sklearn.model_selection import GridSearchCV

vectorizer = TfidfVectorizer()
vectorizer.fit(text_train)

X_train = vectorizer.transform(text_train)
X_test = vectorizer.transform(text_test)

clf = LogisticRegression()
grid = GridSearchCV(clf, param_grid={'C': [.1, 1, 10, 100]}, cv=5
)
grid.fit(X_train, y_train)
```

## 我们哪里做错了？

在这里，我们使用 `X_train` 上的交叉验证进行了网格搜索。 然而，当应用 `TfidfVectorizer` 时，它看到了所有的 `X_train` ，而不仅仅是训练折叠！ 因此，它可以使用测试折叠中单词频率的知识。 这被称为测试集的“污染”，并且使泛化性能或错误选择的参数的估计过于乐观。 我们可以通过流水线解决这个问题：

```
from sklearn.model_selection import GridSearchCV

pipeline = make_pipeline(TfidfVectorizer(),
                         LogisticRegression())

grid = GridSearchCV(pipeline,
                    param_grid={'logisticregression__C': [.1, 1,
10, 100]}, cv=5)

grid.fit(text_train, y_train)
grid.score(text_test, y_test)
```

请注意，我们需要告诉流水线我们要在哪一步设置参数 `C` 。我们可以使用特殊的 `__` 语法来完成此操作。 `__` 之前的名称只是类的名称， `__` 之后的部分是我们想要使用网格搜索设置的参数。




使用流水线的另一个好处是，我们现在还可以使用 `GridSearchCV` 搜索特征提取的参数：

```
from sklearn.model_selection import GridSearchCV

pipeline = make_pipeline(TfidfVectorizer(), LogisticRegression()
)

params = {'logisticregression__C': [.1, 1, 10, 100],
          "tfidfvectorizer__ngram_range": [(1, 1), (1, 2), (2, 2
)]}
grid = GridSearchCV(pipeline, param_grid=params, cv=5)
grid.fit(text_train, y_train)
print(grid.best_params_)
grid.score(text_test, y_test)
```

> 练习
>
> 使用 `StandardScaler` 和 `RidgeRegression` 创建流水线，并将其应用于波士顿住房数据集（使用 `sklearn.datasets.load_boston` 加载）。尝试添加 `sklearn.preprocessing.PolynomialFeatures` 变换器作为第二个预处理步骤，并网格搜索多项式的次数（尝试 1,2 和 3）。

```
# %load solutions/15A_ridge_grid.py
```

# 十六、模型评估、得分指标和处理不平衡类别

在之前的笔记本中，我们已经详细介绍了如何评估模型，以及如何选择最佳模型。到目前为止，我们假设我们得到了表现的度量，它度量模型的质量。但是，应该使用什么度量标准并不总是显而易见的。scikit-learn 中的默认分数，对于分类是准确率，即正确分类的样本的比例，对于回归是 r2 得分，是确定系数。

在许多情况下，这些是合理的默认选择；但是，根据我们的任务，这些并不总是最终或推荐的选择。

让我们更详细地看一下分类，回到手写数字分类的应用。那么，如何训练分类器并使用不同的方式进行评估呢？Scikit-learn 在 sklearn.metrics 模块中有许多有用的方法，可以帮助我们完成这项任务：

```
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np
np.set_printoptions(precision=2)

from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split
from sklearn.svm import LinearSVC

digits = load_digits()
X, y = digits.data, digits.target
X_train, X_test, y_train, y_test = train_test_split(X, y,
                                                    random_state=
1,
                                                    stratify=y,
                                                    test_size=0.
25)

classifier = LinearSVC(random_state=1).fit(X_train, y_train)
y_test_pred = classifier.predict(X_test)

print("Accuracy: {}".format(classifier.score(X_test, y_test)))
```

在这里，我们正确预测了 95.3% 的样本。对于多类问题，通常很有趣的是，知道哪些类很难预测，哪些类很容易，或哪些类混淆了。获取错误分类的更多信息的一种方法，是 `confusion_matrix`，它为每个真正的类显示给定预测结果的频率。

```
from sklearn.metrics import confusion_matrix
confusion_matrix(y_test, y_test_pred)
```

绘图有时更可读：

```
plt.matshow(confusion_matrix(y_test, y_test_pred), cmap="Blues")
plt.colorbar(shrink=0.8)
plt.xticks(range(10))
plt.yticks(range(10))
plt.xlabel("Predicted label")
plt.ylabel("True label");
```

我们可以看到大多数条目都在对角线上，这意味着我们正确地预测了几乎所有样本。 非对角线的条目向我们显示许多 8 被归类为 1，并且 9 很可能与许多其他类混淆。

另一个有用的函数是 `classification_report` ，它为所有类提供精确率，召回率，f 得分和支持度。 精确率是一个类有多少预测实际上是那个类。 TP，FP，TN，FN 分别代表“真正例”，“假正例”，“真负例”和“假负例”：

```
Precision = TP / (TP + FP)
```

召回率是有多少真正例被复原：

```
Recall = TP / (TP + FN)
```

F1 得分是二者的调和均值：

```
F1 = 2 x (precision x recall) / (precision + recall)
```

上述所有这些值的值都在闭区间 `[0,1]` 中，其中 1 表示完美得分。

```
from sklearn.metrics import classification_report
print(classification_report(y_test, y_test_pred))
```

这些指标有助于实践中经常出现的两种特殊情况：

- 不平衡类别，即一个类可能比另一个类更频繁。
- 非对称成本，即一种错误比另一种更“昂贵”。

首先我们来看看第一个。 假设我们有 1:9 的不平衡类别，这是相当温和的（想想广告点击预测，只有 0.001% 的广告可能会被点击）：

```
np.bincount(y) / y.shape[0]
```

作为一个玩具示例，假设我们想要划分数字三和所有其他数字：

```
X, y = digits.data, digits.target == 3
```

现在我们在分类器上运行交叉验证，看看它有多好：

我们的分类器准确率为 90%。这样好吗？还是不好？请记住，90% 的数据“不是三”。因此，让我们看看虚拟分类器的表现如何，它始终预测最频繁的类：

```
from sklearn.dummy import DummyClassifier
cross_val_score(DummyClassifier("most_frequent"), X, y)
```

也是 90%（正如预期的那样）！所以有一种可能，我们的分类器不是很好，它并不比一个甚至不看数据的简单策略更好。不过，这个判断太快了。准确性根本不是评估不平衡数据集的分类器的好方法！

```
np.bincount(y) / y.shape[0]
```

## ROC 曲线

更好的衡量标准是使用所谓的 ROC（受试者工作特性）曲线。ROC 曲线处理分类器的不确定性输出，比如我们上面训练的 SVC 的“决策函数”。它不是在 0 处截断并查看分类结果，而是查看每个可能的截断值并记录有多少真正例预测，以及有多少假正例预测。

下图比较了在“三和其它”任务上，我们的分类器的三个参数设置的 roc 曲线。

```
from sklearn.metrics import roc_curve, roc_auc_score

X_train, X_test, y_train, y_test = train_test_split(X, y, random
_state=42)

for gamma in [.05, 0.1, 0.5]:
    plt.xlabel("False Positive Rate")
    plt.ylabel("True Positive Rate (recall)")
    svm = SVC(gamma=gamma).fit(X_train, y_train)
    decision_function = svm.decision_function(X_test)
    fpr, tpr, _ = roc_curve(y_test, decision_function)
    acc = svm.score(X_test, y_test)
    auc = roc_auc_score(y_test, svm.decision_function(X_test))
    label = "gamma: %0.3f, acc:%.2f auc:%.2f" % (gamma, acc, auc
)
    plt.plot(fpr, tpr, label=label, linewidth=3)
plt.legend(loc="best");
```

由于决策阈值非常小，假正例很低，但假负例也很少，但是阈值非常高的话，真正例率和假负例率都很高。所以一般来说，曲线将从左下角到右上角。对角线反映了机会表现，而目标是尽可能在左上角。这意味着与任何负样本相比，为所有正样本提供更高的 `decision_function` 值。

在这个意义上，该曲线仅考虑正样本和负样本的排名，而不是实际值。从图例中的曲线和准确率值可以看出，即使所有分类器具有相同的准确率，89%，甚至低于虚拟分类器，其中一个具有完美的 roc 曲线，而其中一个表现出机会水平。

对于网格搜索和交叉验证，我们通常希望将模型评估压缩为单个数字。使用 roc 曲线的一个好方法是使用曲线下面积（AUC）。我们可以通过指定 `scoring ="roc_auc"` 在 `cross_val_score` 中使用它：

```
from sklearn.model_selection import cross_val_score
cross_val_score(SVC(gamma='auto'), X, y, scoring="roc_auc", cv=5
)
```

## 内建和自定义的得分函数

还有更多可用的评分方法，可用于不同类型的任务。 你可以在 `SCORERS` 字典中找到它们。 唯一的文档解释了所有这些。

```
from sklearn.metrics.scorer import SCORERS
print(SCORERS.keys())
```

你也可以定义自己的得分指标。 你可以提供一个可调用对象作为 `scoring` 参数，而不是字符串，即具有 `__call__` 方法对象或函数。 它需要接受模型，测试集特征 `X_test` 和测试集标签 `y_test` ，并返回一个浮点数。 更高的浮点意味着更好的模型。

让我们重新实现标准准确率得分：

```
def my_accuracy_scoring(est, X, y):
    return np.mean(est.predict(X) == y)

cross_val_score(SVC(), X, y, scoring=my_accuracy_scoring)
```

练习

在前面的章节中，我们通常使用准确率度量来评估分类器的表现。 我们还没有谈到的相关措施是平均每类准确率（APCA）。 我们记得，准确性定义为：

```
ACC = (TP + TN) / n
```

其中 `n` 是样本总数。 这可以推广为：

```
ACC = T / N
```

其中 `T` 是多类设置中所有正确预测的数量。

给定以下“真实”类标签和预测类标签数组，你是否可以实现一个函数，使用准确率度量来计算平均每类准确率，如下所示？

let's assume our model made the following predictions:

| | C0 | C1 | C2 |
|---|---|---|---|
| C0 | 3 | 0 | 0 |
| C1 | 7 | 50 | 12 |
| C2 | 0 | 0 | 18 |

We compute the accuracy as:

$$ACC = \frac{3 + 50 + 18}{90} \approx 0.79$$

Now, in order to compute the ***average per-class accuracy***, we compute the binary accuracy for each class label separately; i.e., if class 1 is the positive class, class 0 and 2 are both considered the negative class.

$$APC\ ACC = \frac{83/90 + 71/90 + 78/90}{3} \approx 0.86$$

```
y_true = np.array([0, 0, 0, 1, 1, 1, 1, 1, 2, 2])
y_pred = np.array([0, 1, 1, 0, 1, 1, 2, 2, 2, 2])

confusion_matrix(y_true, y_pred)

# %load solutions/16A_avg_per_class_acc.py
```

# 十七、深入：线性模型

线性模型在可用的数据很少时非常有用，或者对于文本分类中的非常大的特征空间很有用。 此外，它们是正则化的良好研究案例。

## 用于回归的线性模型

用于回归的所有线性模型学习系数参数 `coef_` 和偏移 `intercept_` ，来使用线性特征组合做出预测：

```
y_pred = x_test[0] * coef_[0] + ... + x_test[n_features-1] * coe
f_[n_features-1] + intercept_
```

回归的线性模型之间的差异在于，除了很好地拟合训练数据之外，对系数施加什么样的限制或惩罚，作为正则化。 最标准的线性模型是“普通最小二乘回归”，通常简称为“线性回归”。 它没有对 `coef_` 施加任何额外限制，因此当特征数量很大时，它会变得行为异常，并且模型会过拟合。

让我们生成一个简单的模拟，以查看这些模型的行为。

```
from sklearn.datasets import make_regression
from sklearn.model_selection import train_test_split

X, y, true_coefficient = make_regression(n_samples=200, n_featur
es=30, n_informative=10, noise=100, coef=True, random_state=5)
X_train, X_test, y_train, y_test = train_test_split(X, y, random
_state=5, train_size=60, test_size=140)
print(X_train.shape)
print(y_train.shape)
```

## 线性回归

$$\min_{w,b} \sum_i ||w^\mathsf{T} x_i + b - y_i||^2$$

```
from sklearn.linear_model import LinearRegression
linear_regression = LinearRegression().fit(X_train, y_train)
print("R^2 on training set: %f" % linear_regression.score(X_trai
n, y_train))
print("R^2 on test set: %f" % linear_regression.score(X_test, y_
test))

from sklearn.metrics import r2_score
print(r2_score(np.dot(X, true_coefficient), y))

plt.figure(figsize=(10, 5))
coefficient_sorting = np.argsort(true_coefficient)[::-1]
plt.plot(true_coefficient[coefficient_sorting], "o", label="true"
)
plt.plot(linear_regression.coef_[coefficient_sorting], "o", labe
l="linear regression")

plt.legend()

from sklearn.model_selection import learning_curve

def plot_learning_curve(est, X, y):
    training_set_size, train_scores, test_scores = learning_curv
e(est, X, y, train_sizes=np.linspace(.1, 1, 20))
    estimator_name = est.__class__.__name__
    line = plt.plot(training_set_size, train_scores.mean(axis=1)
, '--', label="training scores " + estimator_name)
    plt.plot(training_set_size, test_scores.mean(axis=1), '-', l
abel="test scores " + estimator_name, c=line[0].get_color())
    plt.xlabel('Training set size')
    plt.legend(loc='best')
    plt.ylim(-0.1, 1.1)

plt.figure()
plot_learning_curve(LinearRegression(), X, y)
```

## 岭回归（L2 惩罚）

岭估计器是普通 `LinearRegression` 的简单正则化（称为 l2 惩罚）。特别是，它具有的优点是，在计算上不比普通的最小二乘估计更昂贵。

$$\min_{w,b} \sum_i ||w^T x_i + b - y_i||^2 + \alpha ||w||_2^2$$

正则化的总数通过 `Ridge` 的 `alpha` 参数设置。

```
from sklearn.linear_model import Ridge
ridge_models = {}
training_scores = []
test_scores = []

for alpha in [100, 10, 1, .01]:
    ridge = Ridge(alpha=alpha).fit(X_train, y_train)
    training_scores.append(ridge.score(X_train, y_train))
    test_scores.append(ridge.score(X_test, y_test))
    ridge_models[alpha] = ridge

plt.figure()
plt.plot(training_scores, label="training scores")
plt.plot(test_scores, label="test scores")
plt.xticks(range(4), [100, 10, 1, .01])
plt.xlabel('alpha')
plt.legend(loc="best")

plt.figure(figsize=(10, 5))
plt.plot(true_coefficient[coefficient_sorting], "o", label="true"
, c='b')

for i, alpha in enumerate([100, 10, 1, .01]):
    plt.plot(ridge_models[alpha].coef_[coefficient_sorting], "o"
, label="alpha = %.2f" % alpha, c=plt.cm.viridis(i / 3.))

plt.legend(loc="best")
```

调整 `alpha` 对表现至关重要。

```
plt.figure()
plot_learning_curve(LinearRegression(), X, y)
plot_learning_curve(Ridge(alpha=10), X, y)
```

## Lasso（L1 惩罚）

`Lasso` 估计器可用于对系数施加稀疏性。 换句话说，如果我们认为许多特征不相关，那么我们会更喜欢它。 这是通过所谓的 l1 惩罚来完成的。

$$\min_{w,b} \sum_i \frac{1}{2}||w^\top x_i + b - y_i||^2 + \alpha||w||_1$$

```
from sklearn.linear_model import Lasso

lasso_models = {}
training_scores = []
test_scores = []

for alpha in [30, 10, 1, .01]:
    lasso = Lasso(alpha=alpha).fit(X_train, y_train)
    training_scores.append(lasso.score(X_train, y_train))
    test_scores.append(lasso.score(X_test, y_test))
    lasso_models[alpha] = lasso
plt.figure()
plt.plot(training_scores, label="training scores")
plt.plot(test_scores, label="test scores")
plt.xticks(range(4), [30, 10, 1, .01])
plt.legend(loc="best")

plt.figure(figsize=(10, 5))
plt.plot(true_coefficient[coefficient_sorting], "o", label="true"
, c='b')

for i, alpha in enumerate([30, 10, 1, .01]):
    plt.plot(lasso_models[alpha].coef_[coefficient_sorting], "o"
, label="alpha = %.2f" % alpha, c=plt.cm.viridis(i / 3.))

plt.legend(loc="best")

plt.figure(figsize=(10, 5))
plot_learning_curve(LinearRegression(), X, y)
plot_learning_curve(Ridge(alpha=10), X, y)
plot_learning_curve(Lasso(alpha=10), X, y)
```

你也可以使用 `ElasticNet` ，而不是选择 `Ridge` 或 `Lasso` ，它使用两种形式的正则化，并提供一个参数来指定它们之间的权重。 `ElasticNet` 通常在这些模型中表现最佳。

## 用于分类的线性模型

用于分类的所有线性模型学习系数参数 `coef_` 和偏移 `intercept_` ，来使用线性特征组合做出预测：

```
y_pred = x_test[0] * coef_[0] + ... + x_test[n_features-1] * coe
f_[n_features-1] + intercept_ > 0
```

如你所见，这与回归非常相似，只是应用了零处的阈值。

同样，用于分类的线性模型之间的区别是，对 `coef_` 和 `intercept_` 施加什么类型的正则化，但是在如何测量训练集的拟合（所谓的损失函数）方面也存在微小差异。

线性分类的两种最常见的模型是 `LinearSVC` 实现的线性 SVM，和 `LogisticRegression` 。

线性分类器的正则化的良好直觉是，使用高正则化，如果大多数点被正确分类就足够了。但使用较少的正则化，每个数据点的重要性也越来越高。这里使用具有不同 `C` 值的线性 SVM 来说明。

## `LinearSVC` 中 `C` 的影响

在 `LinearSVC` 中， `C` 参数控制模型中的正则化。

较低的 `C` 产生更多的正则化和更简单的模型，而较高的 `C` 产生较少的正则化和来自各个数据点的更多影响。

```
from figures import plot_linear_svc_regularization
plot_linear_svc_regularization()
```

与 `Ridge` / `Lasso` 划分类似，你可以将 `penalty` 参数设置为 `'l1'` 来强制系数的稀疏性（类似于 `Lasso` ）或 `'l2'` 来鼓励更小的系数（类似于 `Ridge` ）。

## 多类线性分类

```
from sklearn.datasets import make_blobs
plt.figure()
X, y = make_blobs(random_state=42)
plt.figure(figsize=(8, 8))
plt.scatter(X[:, 0], X[:, 1], c=plt.cm.tab10(y))

from sklearn.svm import LinearSVC
linear_svm = LinearSVC().fit(X, y)
print(linear_svm.coef_.shape)
print(linear_svm.intercept_.shape)

plt.figure(figsize=(8, 8))
plt.scatter(X[:, 0], X[:, 1], c=plt.cm.tab10(y))
line = np.linspace(-15, 15)
for coef, intercept in zip(linear_svm.coef_, linear_svm.intercep
t_):
    plt.plot(line, -(line * coef[0] + intercept) / coef[1])
plt.ylim(-10, 15)
plt.xlim(-10, 8);
```

点以一对多（OVR）的方式分类（又名 OVA），我们将测试点分配给模型对测试点具有最高置信度的类（在 SVM 情况下，与分隔超平面的距离最大）。

> 练习
>
> 使用 `LogisticRegression` 来分类数字数据集，并网格搜索 `C` 参数。当你增加或减少 `alpha` 时，你认为上面的学习曲线如何变化？尝试更改岭和 `Lasso` 中的 `alpha` 参数，看看你的直觉是否正确。

```
from sklearn.datasets import load_digits
from sklearn.linear_model import LogisticRegression

digits = load_digits()
X_digits, y_digits = digits.data, digits.target

# split the dataset, apply grid-search

# %load solutions/17A_logreg_grid.py

# %load solutions/17B_learning_curve_alpha.py
```

# 十八、深入：决策树与森林

在这里，我们将探索一类基于决策树的算法。 最基本决策树非常直观。 它们编码一系列 `if` 和 `else` 选项，类似于一个人如何做出决定。 但是，从数据中完全可以了解要问的问题以及如何处理每个答案。

例如，如果你想创建一个识别自然界中发现的动物的指南，你可能会问以下一系列问题：

- 动物是大于还是小于一米？
  - 较大：动物有角吗？
    - 是的：角长是否超过十厘米？
    - 不是：动物有项圈吗？
- 较小：动物有两条腿还是四条腿？
  - 二：动物有翅膀吗？
  - 四：动物有浓密的尾巴吗？

等等。 这种问题的二元分裂是决策树的本质。

基于树的模型的主要好处之一是它们几乎不需要数据预处理。 它们可以处理不同类型的变量（连续和离散），并且对特征的缩放不变。

另一个好处是基于树的模型被称为“非参数”，这意味着他们没有一套固定的参数需要学习。 相反，如果给出更多数据，树模型可以变得越来越灵活。 换句话说，自由参数的数量随着样本量而增长并且不是固定的，例如在线性模型中。

## 决策树回归

决策树是一种简单的二元分类树，类似于最近邻分类。 它可以这样使用：

```
from figures import make_dataset
x, y = make_dataset()
X = x.reshape(-1, 1)

plt.figure()
plt.xlabel('Feature X')
plt.ylabel('Target y')
plt.scatter(X, y);

from sklearn.tree import DecisionTreeRegressor

reg = DecisionTreeRegressor(max_depth=5)
reg.fit(X, y)

X_fit = np.linspace(-3, 3, 1000).reshape((-1, 1))
y_fit_1 = reg.predict(X_fit)

plt.figure()
plt.plot(X_fit.ravel(), y_fit_1, color='tab:blue', label="predic
tion")
plt.plot(X.ravel(), y, 'C7.', label="training data")
plt.legend(loc="best");
```

单个决策树允许我们以非参数方式估计标签，但显然存在一些问题。在某些地区，该模型表现出高偏差并且对数据欠拟合。（请见不遵循数据轮廓的长扁形线条），而在其他区域，模型表现高方差并且过拟合数据（反映为受单点噪声影响的窄峰形）。

## 决策树分类

决策树分类原理非常相似，通过将叶子中的多数类分配给叶子中的所有点：

```
from sklearn.datasets import make_blobs
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier
from figures import plot_2d_separator
from figures import cm2


X, y = make_blobs(centers=[[0, 0], [1, 1]], random_state=61526,
n_samples=100)
X_train, X_test, y_train, y_test = train_test_split(X, y, random
_state=42)

clf = DecisionTreeClassifier(max_depth=5)
clf.fit(X_train, y_train)

plt.figure()
plot_2d_separator(clf, X, fill=True)
plt.scatter(X_train[:, 0], X_train[:, 1], c=y_train, cmap=cm2, s=
60, alpha=.7, edgecolor='k')
plt.scatter(X_test[:, 0], X_test[:, 1], c=y_test, cmap=cm2, s=60
, edgecolor='k');
```

有许多参数可以控制树的复杂性，但最容易理解的是最大深度。这限制了树可以对输入空间进行划分的精确度，或者在决定样本所在的类之前,可以询问多少 `if-else` 问题。

此参数对于调整树和基于树的模型非常重要。下面的交互式图表显示了该模型的欠拟合和过拟合。 `max_depth` 为 1 显然是一个欠拟合的模型，而 7 或 8 的深度明显过拟合。对于该数据集，树可以生长的最大深度是 8，此时每个叶仅包含来自单个类的样本。这被称为所有叶子都是“纯的”。

在下面的交互式图中，区域被指定为蓝色和红色，来表明该区域的预测类。颜色的阴影表示该类的预测概率（较暗为较高概率），而黄色区域表示任一类的预测概率相等。

```
from figures import plot_tree
max_depth = 3
plot_tree(max_depth=max_depth)
```

决策树训练快，易于理解，并且经常产生可解释的模型。 但是，单个树通常倾向于过拟合训练数据。 使用上面的滑块，你可能会注意到，即使在类之间有良好的分隔之前，模型也会开始过拟合。

因此，在实践中，更常见的是组合多个树来产生更好泛化的模型。 组合树的最常用方法是随机森林和梯度提升树。

## 随机森林

随机森林只是许多树，建立在数据的不同随机子集（带放回抽样）上，并对于每个分裂，使用特征的不同随机子集（无放回抽样）。 这使得树彼此不同，并使它们过拟合不同的方面。 然后，他们的预测被平均，产生更平稳的估计，更少过拟合。

```
from figures import plot_forest
max_depth = 3
plot_forest(max_depth=max_depth)
```

## 通过交叉验证选择最优估计

```
from sklearn.model_selection import GridSearchCV
from sklearn.datasets import load_digits
from sklearn.ensemble import RandomForestClassifier

digits = load_digits()
X, y = digits.data, digits.target

X_train, X_test, y_train, y_test = train_test_split(X, y, random
_state=42)

rf = RandomForestClassifier(n_estimators=200)
parameters = {'max_features':['sqrt', 'log2', 10],
              'max_depth':[5, 7, 9]}

clf_grid = GridSearchCV(rf, parameters, n_jobs=-1)
clf_grid.fit(X_train, y_train)

clf_grid.score(X_train, y_train)

clf_grid.score(X_test, y_test)
```

## 另一个选项：梯度提升

可能有用的另一种集合方法是提升：在这里，我们构建了一个由 200 个估计器组成的链，它迭代地改进了先前估计器的结果，而不是查看（比方说）200 个并行估计器。 我们的想法是，通过顺序应用非常快速，简单的模型，我们可以获得比任何单个部分更好的总模型误差。

```
from sklearn.ensemble import GradientBoostingRegressor
clf = GradientBoostingRegressor(n_estimators=100, max_depth=5, l
earning_rate=.2)
clf.fit(X_train, y_train)

print(clf.score(X_train, y_train))
print(clf.score(X_test, y_test))
```

> 练习：梯度提升的交叉验证
>
> 使用网格搜索在数字数据集上优化梯度提升树 `learning_rate` 和 `max_depth` 。

```
from sklearn.datasets import load_digits
from sklearn.ensemble import GradientBoostingClassifier

digits = load_digits()
X_digits, y_digits = digits.data, digits.target

# split the dataset, apply grid-search

# %load solutions/18_gbc_grid.py
```

## 特征的重要性

`RandomForest` 和 `GradientBoosting` 对象在拟合之后都会提供 `feature_importances_` 属性。此属性是这些模型最强大的功能之一。它们基本上量化了在不同树的节点中，每个特征对表现的贡献程度。

```
X, y = X_digits[y_digits < 2], y_digits[y_digits < 2]

rf = RandomForestClassifier(n_estimators=300, n_jobs=1)
rf.fit(X, y)
print(rf.feature_importances_)  # one value per feature

plt.figure()
plt.imshow(rf.feature_importances_.reshape(8, 8), cmap=plt.cm.vi
ridis, interpolation='nearest')
```

# 十九、自动特征选择

我们经常收集许多可能与监督预测任务相关的特征，但我们不知道它们中的哪一个实际上是预测性的。 为了提高可解释性，有时还提高泛化表现，我们可以使用自动特征选择来选择原始特征的子集。 有几种可用的特征选择方法，我们将按照复杂性的升序来解释。

对于给定的监督模型，最佳特征选择策略是尝试每个可能的特征子集，并使用该子集评估泛化表现。 但是，特征子集是指数级，因此这种详尽的搜索通常是不可行的。 下面讨论的策略可以被认为是这种不可行计算的替代。

## 单变量统计

选择要素的最简单方法是使用单变量统计，即通过单独查看每个特征并运行统计检验，来查看它是否与目标相关。 这种检验也称为方差分析（ANOVA）。

我们创建了一个人造数据集，其中包含乳腺癌数据和另外 50 个完全随机的特征。

```
from sklearn.datasets import load_breast_cancer, load_digits
from sklearn.model_selection import train_test_split

cancer = load_breast_cancer()

# get deterministic random numbers
rng = np.random.RandomState(42)
noise = rng.normal(size=(len(cancer.data), 50))
# add noise features to the data
# the first 30 features are from the dataset, the next 50 are no
ise
X_w_noise = np.hstack([cancer.data, noise])

X_train, X_test, y_train, y_test = train_test_split(X_w_noise, c
ancer.target,
                                                    random_state=
0, test_size=.5)
```

我们必须在统计检验的 p 值上定义一个阈值，来决定要保留多少特征。 在 scikit-learn 中实现了几种策略，一种直接的策略是 `SelectPercentile` ，它选择原始特征的百分位数（下面我们选择 50%）：

```
from sklearn.feature_selection import SelectPercentile

# use f_classif (the default) and SelectPercentile to select 50%
 of features:
select = SelectPercentile(percentile=50)
select.fit(X_train, y_train)
# transform training set:
X_train_selected = select.transform(X_train)

print(X_train.shape)
print(X_train_selected.shape)
```

我们还可以直接使用检验统计量，来查看每个特征的相关性。 由于乳腺癌数据集是一项分类任务，我们使用 `f_classif` ，F 检验用于分类。 下面我们绘制 p 值，与 80 个特征中的每一个相关（30 个原始特征和 50 个噪声特征）。 低 p 值表示信息性特征。

```
from sklearn.feature_selection import f_classif, f_regression, c
hi2

F, p = f_classif(X_train, y_train)

plt.figure()
plt.plot(p, 'o')
```

显然，前 30 个特征中的大多数具有非常小的 p 值。

回到 `SelectPercentile` 转换器，我们可以使用 `get_support` 方法获得所选特征：

```
mask = select.get_support()
print(mask)
# 展示掩码。黑色是真，白色是假
plt.matshow(mask.reshape(1, -1), cmap='gray_r')
```

几乎所有最初的 30 个特征都被还原了。 我们还可以通过在数据上训练监督模型，来分析特征选择的效果。 仅在训练集上学习特征选择非常重要！

```
from sklearn.linear_model import LogisticRegression

# 转换测试数据
X_test_selected = select.transform(X_test)

lr = LogisticRegression()
lr.fit(X_train, y_train)
print("Score with all features: %f" % lr.score(X_test, y_test))
lr.fit(X_train_selected, y_train)
print("Score with only selected features: %f" % lr.score(X_test_
selected, y_test))
```

## 基于模型的特征选择

用于特征选择的稍微复杂的方法，是使用监督机器学习模型，并基于模型认为它们的重要性来选择特征。 这要求模型提供某种方法，按重要性对特征进行排名。 这适用于所有基于树的模型（实现 `get_feature_importances` ）和所有线性模型，系数可用于确定特征对结果的影响程度。

任何这些模型都可以制作成变换器，通过使用 `SelectFromModel` 类包装它，用于特征选择：

```
from sklearn.feature_selection import SelectFromModel
from sklearn.ensemble import RandomForestClassifier
select = SelectFromModel(RandomForestClassifier(n_estimators=100
, random_state=42), threshold="median")

select.fit(X_train, y_train)
X_train_rf = select.transform(X_train)
print(X_train.shape)
print(X_train_rf.shape)

mask = select.get_support()
# 展示掩码。黑色是真，白色是假
plt.matshow(mask.reshape(1, -1), cmap='gray_r')

X_test_rf = select.transform(X_test)
LogisticRegression().fit(X_train_rf, y_train).score(X_test_rf, y
_test)
```

此方法构建单个模型（在本例中为随机森林）并使用此模型中的特征重要性。 我们可以通过在数据子集上训练多个模型，来进行更精细的搜索。 一种特殊的策略是递归特征消除：

## 递归特征消除

递归特征消除在整个特征集上构建模型，类似于上述方法，选择模型认为最重要的特征子集。但是，通常只会从数据集中删除单个要素，并使用其余要素构建新模型。重复删除特征和模型构建的过程，直到只剩下预定数量的特征：

```
from sklearn.feature_selection import RFE
select = RFE(RandomForestClassifier(n_estimators=100, random_sta
te=42), n_features_to_select=40)

select.fit(X_train, y_train)
# 可视化所选特征
mask = select.get_support()
plt.matshow(mask.reshape(1, -1), cmap='gray_r')

X_train_rfe = select.transform(X_train)
X_test_rfe = select.transform(X_test)

LogisticRegression().fit(X_train_rfe, y_train).score(X_test_rfe,
 y_test)

select.score(X_test, y_test)
```

> 练习
>
> 创建“XOR”数据集，如下面的第一个单元格：添加随机特征，并使用随机森林，在还原原始特征时，比较单变量选择与基于模型的选择。

```
import numpy as np

rng = np.random.RandomState(1)

# 在 [0,1] 范围内生成 400 个随机整数
X = rng.randint(0, 2, (200, 2))
y = np.logical_xor(X[:, 0] > 0, X[:, 1] > 0)  # XOR creation

plt.scatter(X[:, 0], X[:, 1], c=plt.cm.tab10(y))

# %load solutions/19_univariate_vs_mb_selection.py
```

# 二十、无监督学习：层次和基于密度的聚类算法

在第八章中，我们介绍了一种必不可少且广泛使用的聚类算法 K-means。 K-means 的一个优点是它非常容易实现，并且与其他聚类算法相比，它在计算上也非常有效。 然而，我们已经看到 K-Means 的一个缺点是它只有在数据可以分组为球形时才能正常工作。 此外，我们必须事先指定簇的数量 `k` - 如果我们没有我们期望找到多少个簇的先验知识，这可能是一个问题。

在本笔记本中，我们将介绍两种可选的聚类方法，层次聚类和基于密度的聚类。

## 层次聚类

层次聚类的一个很好的特性是，我们可以将结果可视化为树状图，即层次树。 使用可视化，我们可以通过设置“深度”阈值来决定我们希望数据集的簇有多“深”。 或者换句话说，我们不需要预先决定簇的数量。

## 聚合和分裂的层次聚类

此外，我们可以区分两种主要的层次聚类方法：分裂聚类和聚合聚类。 在聚合聚类中，我们从数据集中的单个样本开始，并迭代地将其与其他样本合并以形成簇 - 我们可以将其视为构建簇的树状图的自底向上的方法。 然而，在分裂聚类中，我们从作为一个簇的整个数据集开始，并且我们迭代地将其拆分成更小的子簇 - 自顶向下的方法。

在这个笔记本中，我们将使用聚合聚类。

## 单个和完整链接

现在，下一个问题是我们如何测量样本之间的相似性。 一种方法是我们已经在 K-Means 算法中使用的，熟悉的欧几里德距离度量。 作为回顾，两个 `m` 维向量 `p` 和 `q` 之间的距离可以计算为：

$$\mathrm{d}(\mathbf{q}, \mathbf{p}) = \sqrt{(q_1 - p_1)^2 + (q_2 - p_2)^2 + \cdots + (q_m - p_m)^2}$$

(1)

$$= \sqrt{\sum_{j=1}^{m} (q_j - p_j)^2}.$$

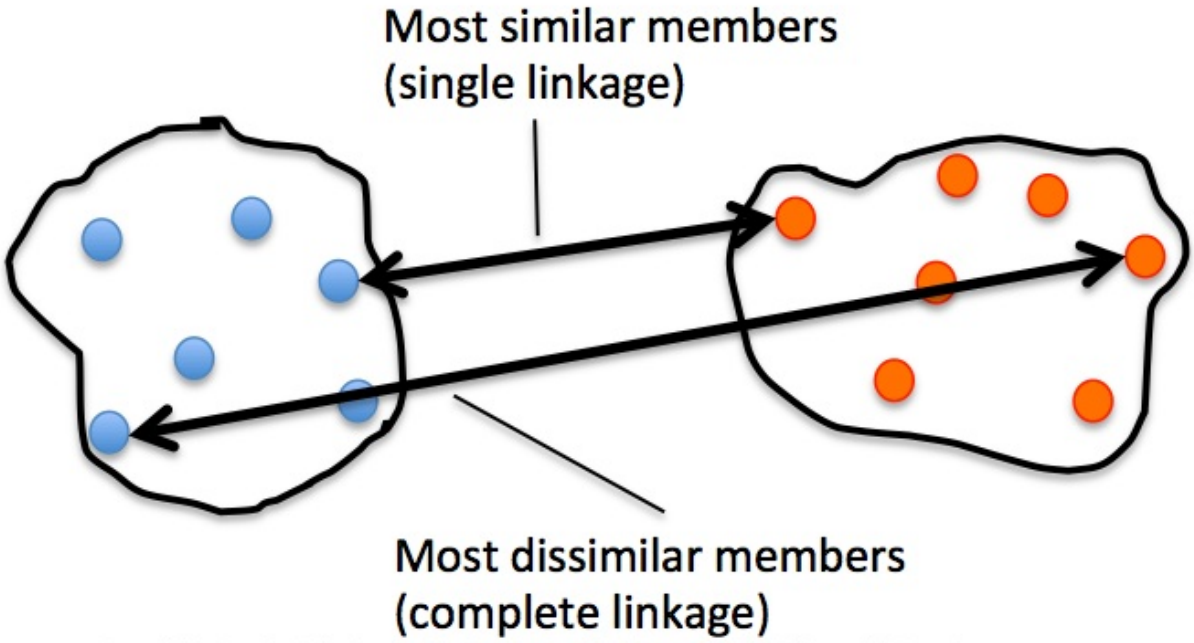
(2)

然而，这是两个个样本之间的距离。 现在，我们如何计算样本子集之间的相似性，以便在构建树状图时决定合并哪些簇？ 即，我们的目标是迭代地合并最相似的一对簇，直到只剩下一个大簇。 有许多不同的方法，例如单个和完整链接。

在单个链接中，我们在每两个簇中选取一对最相似的样本（例如，基于欧几里德距离），并将具有最相似的两个成员的两个簇合并为一个新的更大的簇。

在完整链接中，我们比较每两个簇的两个最不相似的成员，并且我们合并两个簇，其中两个最不相似的成员之间的距离最小。

> 译者注：还有比较两个簇形心的方法，算是一种折中。




[Image source: https://github.com/rasbt/python-machine-learning-book/blob/master/code/ch11/images/11_05.png]

为了看到实际的聚合层次聚类方法，让我们加载熟悉的鸢尾花数据集 - 我们假装不知道真正的类标签，并想要找出它包含多少不同的物种：

```
from sklearn.datasets import load_iris
from figures import cm3

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

plt.scatter(X[:, 0], X[:, 1], c=y, cmap=cm3)
```

首先，我们从一些探索性聚类开始，使用 SciPy 的 `linkage` 和 `dendrogram` 函数来可视化簇的树状图：

```
from scipy.cluster.hierarchy import linkage
from scipy.cluster.hierarchy import dendrogram

clusters = linkage(X,
                   metric='euclidean',
                   method='complete')

dendr = dendrogram(clusters)

plt.ylabel('Euclidean Distance')
```

接下来，让我们使用来自 scikit-learn 的 `AgglomerativeClustering` 估计器，并将数据集划分为 3 个簇。你能猜出它会重现的树状图中有哪 3 个簇吗？

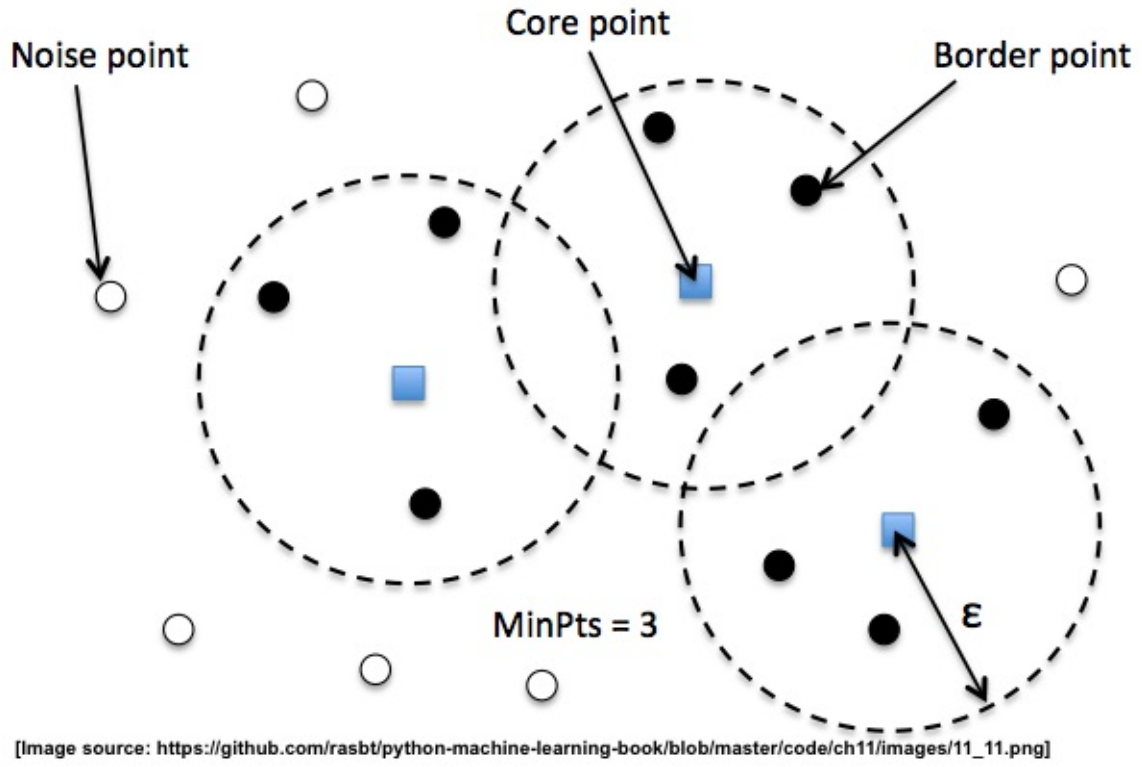
```
from sklearn.cluster import AgglomerativeClustering

ac = AgglomerativeClustering(n_clusters=3,
                             affinity='euclidean',
                             linkage='complete')

prediction = ac.fit_predict(X)
print('Cluster labels: %s\n' % prediction)

plt.scatter(X[:, 0], X[:, 1], c=prediction, cmap=cm3)
```

## 基于密度的聚类 - DBSCAN

另一种有用的聚类方法是“具有噪声的基于密度的聚类方法”（DBSCAN）。本质上，我们可以将 DBSCAN 视为一种算法，该算法根据密集的点区域将数据集划分为子分组。

在 DBSCAN 中，我们区分了 3 种不同的“点”：

- 核心点：核心点是一个点，在其半径 `epsilon` 内，至少具有最小数量（`MinPts`）的其他点。
- 边界点：边界点是一个点，它不是核心点，因为它的邻域中没有足够的 `MinPts`，但位于核心点的半径 `epsilon` 内。
- 噪点：所有其他的点，既不是核心点也不是边界点。




[Image source: https://github.com/rasbt/python-machine-learning-book/blob/master/code/ch11/images/11_11.png]

DBSCAN 的一个很好的特性是我们不必预先指定多少个簇。但是，它需要设置其他超参数，例如 `MinPts` 的值和半径 `epsilon` 。

```
from sklearn.datasets import make_moons
X, y = make_moons(n_samples=400,
                  noise=0.1,
                  random_state=1)
plt.scatter(X[:,0], X[:,1])
plt.show()
from sklearn.cluster import DBSCAN

db = DBSCAN(eps=0.2,
            min_samples=10,
            metric='euclidean')
prediction = db.fit_predict(X)

print("Predicted labels:\n", prediction)

plt.scatter(X[:, 0], X[:, 1], c=prediction, cmap=cm3)
```

> 练习
>
> 使用以下玩具数据集，两个同心圆，尝试我们到目前为止使用的三种不同的聚类算法： `KMeans` ， `AgglomerativeClustering` 和 `DBSCAN` 。哪种聚类算法能够最好地再现或发现隐藏的结构（假装我们不知道 `y` ）？你能解释为什么这个特殊的算法是一个不错的选择，而另外两个“失败”了？

```
from sklearn.datasets import make_circles

X, y = make_circles(n_samples=1500,
                    factor=.4,
                    noise=.05)

plt.scatter(X[:, 0], X[:, 1], c=y);

# %load solutions/20_clustering_comparison.py
```

# 二十一、无监督学习：非线性降维

## 流形学习

PCA 的一个弱点是它无法检测到非线性特征。 已经开发了一组称为流形学习的算法，来解决这个缺陷。流形学习中使用的规范数据集是 S 曲线：

```
from sklearn.datasets import make_s_curve
X, y = make_s_curve(n_samples=1000)

from mpl_toolkits.mplot3d import Axes3D
ax = plt.axes(projection='3d')

ax.scatter3D(X[:, 0], X[:, 1], X[:, 2], c=y)
ax.view_init(10, -60);
```

这是一个嵌入三维的二维数据集，但它以某种方式嵌入，PCA 无法发现底层数据方向：

```
from sklearn.decomposition import PCA
X_pca = PCA(n_components=2).fit_transform(X)
plt.scatter(X_pca[:, 0], X_pca[:, 1], c=y);
```

然而，`sklearn.manifold`子模块中可用的流形学习算法能够还原底层的二维流形：

```
from sklearn.manifold import Isomap

iso = Isomap(n_neighbors=15, n_components=2)
X_iso = iso.fit_transform(X)
plt.scatter(X_iso[:, 0], X_iso[:, 1], c=y);
```

## 数字数据上的流形学习

我们可以将流形学习技术应用于更高维度的数据集，例如我们之前看到的数字数据：

```
from sklearn.datasets import load_digits
digits = load_digits()

fig, axes = plt.subplots(2, 5, figsize=(10, 5),
                         subplot_kw={'xticks':(), 'yticks': ()})
for ax, img in zip(axes.ravel(), digits.images):
    ax.imshow(img, interpolation="none", cmap="gray")
```

我们可以使用线性技术（例如 PCA）可视化数据集。 我们看到这已经提供了一些数据的直觉：

```
# 构建 PCA 模型
pca = PCA(n_components=2)
pca.fit(digits.data)
# 将数字数据转换为前两个主成分
digits_pca = pca.transform(digits.data)
colors = ["#476A2A", "#7851B8", "#BD3430", "#4A2D4E", "#875525",
          "#A83683", "#4E655E", "#853541", "#3A3120","#535D8E"]
plt.figure(figsize=(10, 10))
plt.xlim(digits_pca[:, 0].min(), digits_pca[:, 0].max() + 1)
plt.ylim(digits_pca[:, 1].min(), digits_pca[:, 1].max() + 1)
for i in range(len(digits.data)):
    # 实际上将数字绘制为文本而不是使用散点图
    plt.text(digits_pca[i, 0], digits_pca[i, 1], str(digits.targ
et[i]),
             color = colors[digits.target[i]],
             fontdict={'weight': 'bold', 'size': 9})
plt.xlabel("first principal component")
plt.ylabel("second principal component");
```

但是，使用更强大的非线性技术可以提供更好的可视化效果。 在这里，我们使用 t-SNE 流形学习方法：

```
from sklearn.manifold import TSNE
tsne = TSNE(random_state=42)
# 使用 fit_transform 而不是 fit，因为 TSNE 没有 fit 方法
digits_tsne = tsne.fit_transform(digits.data)

plt.figure(figsize=(10, 10))
plt.xlim(digits_tsne[:, 0].min(), digits_tsne[:, 0].max() + 1)
plt.ylim(digits_tsne[:, 1].min(), digits_tsne[:, 1].max() + 1)
for i in range(len(digits.data)):
    # 实际上将数字绘制为文本而不是使用散点图
    plt.text(digits_tsne[i, 0], digits_tsne[i, 1], str(digits.ta
rget[i]),
             color = colors[digits.target[i]],
             fontdict={'weight': 'bold', 'size': 9})
```

t-SNE 比其他流形学习算法运行时间更长，但结果非常惊人。 请记住，此算法纯粹是无监督的，并且不知道类标签。 它仍然能够很好地分离类别（尽管类 4 和 类 9 已被分成多个分组）。

> 练习
>
> 将 isomap 应用于数字数据集的结果与 PCA 和 t-SNE 的结果进行比较。 你认为哪个结果看起来最好？ 鉴于 t-SNE 很好地将类别分开，人们可能会试图将这个处理过程用于分类。 尝试在使用 t-SNE 转换的数字数据上，训练 K 最近邻分类器，并与没有任何转换的数据集上的准确性比较。

```
# %load solutions/21A_isomap_digits.py

# %load solutions/21B_tsne_classification.py
```

# 二十二、无监督学习：异常检测

常检测是一种机器学习任务，包括发现所谓的异常值。

“异常值是一种数据集中的观测值，似乎与该组数据的其余部分不一致。”-- Johnson 1992

“异常值是一种观测值，与其他观测值有很大差异，引起人们怀疑它是由不同的机制产生的。”-- Outlier/Anomaly Hawkins 1980

## 异常检测设定的类型

- 监督 AD
  - 标签可用于正常和异常数据
  - 类似于稀有类挖掘/不平衡分类
- 半监督 AD（新奇检测）
  - 只有正常的数据可供训练
  - 该算法仅学习正常数据
- 无监督 AD（异常值检测）
  - 没有标签，训练集 = 正常 + 异常数据
  - 假设：异常非常罕见

```
%matplotlib inline

import warnings
warnings.filterwarnings("ignore")

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

让我们首先熟悉不同的无监督异常检测方法和算法。 为了可视化不同算法的输出，我们考虑包含二维高斯混合的玩具数据集。

## 生成数据集

```
from sklearn.datasets import make_blobs

X, y = make_blobs(n_features=2, centers=3, n_samples=500,
                  random_state=42)

X.shape

plt.figure()
plt.scatter(X[:, 0], X[:, 1])
plt.show()
```

## 使用密度估计的异常检测

```
from sklearn.neighbors.kde import KernelDensity

# 用高斯核密度估计器估算密度
kde = KernelDensity(kernel='gaussian')
kde = kde.fit(X)
kde

kde_X = kde.score_samples(X)
print(kde_X.shape)  # 包含数据的对数似然。 越小样本越罕见

from scipy.stats.mstats import mquantiles
alpha_set = 0.95
tau_kde = mquantiles(kde_X, 1. - alpha_set)

n_samples, n_features = X.shape
X_range = np.zeros((n_features, 2))
X_range[:, 0] = np.min(X, axis=0) - 1.
X_range[:, 1] = np.max(X, axis=0) + 1.

h = 0.1  # step size of the mesh
x_min, x_max = X_range[0]
y_min, y_max = X_range[1]
xx, yy = np.meshgrid(np.arange(x_min, x_max, h),
                     np.arange(y_min, y_max, h))

grid = np.c_[xx.ravel(), yy.ravel()]

Z_kde = kde.score_samples(grid)
Z_kde = Z_kde.reshape(xx.shape)

plt.figure()
c_0 = plt.contour(xx, yy, Z_kde, levels=tau_kde, colors='red', l
inewidths=3)
plt.clabel(c_0, inline=1, fontsize=15, fmt={tau_kde[0]: str(alph
a_set)})
plt.scatter(X[:, 0], X[:, 1])
plt.show()
```

## 单类 SVM

基于密度的估计的问题在于，当数据的维数增加时，它们往往变得低效。 这就是所谓的维度灾难，尤其会影响密度估算算法。 在这种情况下可以使用单类 SVM 算法。

```
from sklearn.svm import OneClassSVM

nu = 0.05  # theory says it should be an upper bound of the frac
tion of outliers
ocsvm = OneClassSVM(kernel='rbf', gamma=0.05, nu=nu)
ocsvm.fit(X)

X_outliers = X[ocsvm.predict(X) == -1]

Z_ocsvm = ocsvm.decision_function(grid)
Z_ocsvm = Z_ocsvm.reshape(xx.shape)

plt.figure()
c_0 = plt.contour(xx, yy, Z_ocsvm, levels=[0], colors='red', lin
ewidths=3)
plt.clabel(c_0, inline=1, fontsize=15, fmt={0: str(alpha_set)})
plt.scatter(X[:, 0], X[:, 1])
plt.scatter(X_outliers[:, 0], X_outliers[:, 1], color='red')
plt.show()
```

## 支持向量 - 离群点

所谓的单类 SVM 的支持向量形成离群点。

```
X_SV = X[ocsvm.support_]
n_SV = len(X_SV)
n_outliers = len(X_outliers)

print('{0:.2f} <= {1:.2f} <= {2:.2f}?'.format(1./n_samples*n_out
liers, nu, 1./n_samples*n_SV))
```

只有支持向量涉及单类 SVM 的决策函数。

- 绘制单类 SVM 决策函数的级别集，就像我们对真实密度所做的那样。
- 突出支持向量。

```
plt.figure()
plt.contourf(xx, yy, Z_ocsvm, 10, cmap=plt.cm.Blues_r)
plt.scatter(X[:, 0], X[:, 1], s=1.)
plt.scatter(X_SV[:, 0], X_SV[:, 1], color='orange')
plt.show()
```

> 练习
>
> 更改`gamma``参数并查看它对决策函数平滑度的影响。

```
# %load solutions/22_A-anomaly_ocsvm_gamma.py
```

## 隔离森林

隔离森林是一种基于树的异常检测算法。该算法构建了许多随机树，其基本原理是，如果样本被隔离，在非常少量的随机分割之后，它应该单独存在于叶子中。隔离森林根据样本最终所在的树的深度建立异常得分。

```
from sklearn.ensemble import IsolationForest

iforest = IsolationForest(n_estimators=300, contamination=0.10)
iforest = iforest.fit(X)

Z_iforest = iforest.decision_function(grid)
Z_iforest = Z_iforest.reshape(xx.shape)

plt.figure()
c_0 = plt.contour(xx, yy, Z_iforest,
                  levels=[iforest.threshold_],
                  colors='red', linewidths=3)
plt.clabel(c_0, inline=1, fontsize=15,
           fmt={iforest.threshold_: str(alpha_set)})
plt.scatter(X[:, 0], X[:, 1], s=1.)
plt.show()
```

> 练习
>
> 以图形方式说明树的数量对决策函数平滑度的影响。

```
# %load solutions/22_B-anomaly_iforest_n_trees.py
```

## 数字数据集上的图解

我们现在将应用 `IsolationForest` 算法来查找以非常规方式编写的数字。

```
from sklearn.datasets import load_digits
digits = load_digits()
```

数字数据集包括 `8×8` 的数字图像。

```
images = digits.images
labels = digits.target
images.shape

i = 102

plt.figure(figsize=(2, 2))
plt.title('{0}'.format(labels[i]))
plt.axis('off')
plt.imshow(images[i], cmap=plt.cm.gray_r, interpolation='nearest'
)
plt.show()
```

要将图像用作训练集，我们需要将图像展开。

```
n_samples = len(digits.images)
data = digits.images.reshape((n_samples, -1))

data.shape

X = data
y = digits.target

X.shape
```

让我们关注数字 5。

```
X_5 = X[y == 5]

X_5.shape

fig, axes = plt.subplots(1, 5, figsize=(10, 4))
for ax, x in zip(axes, X_5[:5]):
    img = x.reshape(8, 8)
    ax.imshow(img, cmap=plt.cm.gray_r, interpolation='nearest')
    ax.axis('off')
```

- 让我们使用 `IsolationForest` 来查找前 5% 最异常的图像。
- 让我们绘制他们吧！

```
from sklearn.ensemble import IsolationForest
iforest = IsolationForest(contamination=0.05)
iforest = iforest.fit(X_5)
```

使用 `iforest.decision_function` 计算“异常”的级别。越低就越异常。

```
iforest_X = iforest.decision_function(X_5)
plt.hist(iforest_X);
```

让我们绘制最强的正常值。

```
X_strong_inliers = X_5[np.argsort(iforest_X)[-10:]]

fig, axes = plt.subplots(2, 5, figsize=(10, 5))

for i, ax in zip(range(len(X_strong_inliers)), axes.ravel()):
    ax.imshow(X_strong_inliers[i].reshape((8, 8)),
              cmap=plt.cm.gray_r, interpolation='nearest')
    ax.axis('off')
```

让我们绘制最强的异常值。

```
fig, axes = plt.subplots(2, 5, figsize=(10, 5))

X_outliers = X_5[iforest.predict(X_5) == -1]

for i, ax in zip(range(len(X_outliers)), axes.ravel()):
    ax.imshow(X_outliers[i].reshape((8, 8)),
              cmap=plt.cm.gray_r, interpolation='nearest')
    ax.axis('off')
```

> 练习
>
> 用所有其他数字重新运行相同的分析。

```
# %load solutions/22_C-anomaly_digits.py
```

# 二十三、核外学习 - 用于语义分析的大规模文本分类

## 可扩展性问题

`sklearn.feature_extraction.text.CountVectorizer` 和 `sklearn.feature_e` 类受到许多可伸缩性问题的困扰，这些问题都源于 `vocabulary_` 属性（Python 字典）的内部使用，它用于将 unicode 字符串特征名称映射为整数特征索引。

主要的可扩展性问题是：

- 文本向量化程序的内存使用情况：所有特征的字符串表示形式都加载到内存中
- 文本特征提取的并行化问题：`vocabulary_` 是一个共享状态：复杂的同步和开销
- 不可能进行在线或核外/流式学习：`vocabulary_` 需要从数据中学习：在遍历一次整个数据集之前无法知道其大小

为了更好地理解这个问题，让我们看一下 `vocabulary_` 属性的工作原理。
在 `fit` 的时候，语料库的标记由整数索引唯一标识，并且该映射存储在词汇表中：

```
from sklearn.feature_extraction.text import CountVectorizer

vectorizer = CountVectorizer(min_df=1)

vectorizer.fit([
    "The cat sat on the mat.",
])
vectorizer.vocabulary_
```

在 `transform` 的时候，使用词汇表来构建出现矩阵：

```
X = vectorizer.transform([
    "The cat sat on the mat.",
    "This cat is a nice cat.",
]).toarray()

print(len(vectorizer.vocabulary_))
print(vectorizer.get_feature_names())
print(X)
```

让我们用稍大的语料库重新拟合：

```
vectorizer = CountVectorizer(min_df=1)

vectorizer.fit([
    "The cat sat on the mat.",
    "The quick brown fox jumps over the lazy dog.",
])
vectorizer.vocabulary_
```

`vocabulary_` 随着训练语料库的大小而（以对数方式）增长。请注意，我们无法在 2 个文本文档上并行构建词汇表，因为它们共享一些单词，因此需要某种共享数据结构或同步障碍，这对于设定来说很复杂，特别是如果我们想要将处理过程分发给集群的时候。

有了这个新的词汇表，输出空间的维度现在变大了：

```
X = vectorizer.transform([
    "The cat sat on the mat.",
    "This cat is a nice cat.",
]).toarray()

print(len(vectorizer.vocabulary_))
print(vectorizer.get_feature_names())
print(X)
```

## IMDB 电影数据集

为了说明基于词汇的向量化器的可扩展性问题，让我们为经典文本分类任务加载更真实的数据集：文本文档的情感分析。目标是从互联网电影数据库（IMDb）中区分出积极的电影评论。

在接下来的章节中，使用了 Maas 等人收集的来自 IMDb 的电影评论的大型子集。

> A. L. Maas, R. E. Daly, P. T. Pham, D. Huang, A. Y. Ng, and C. Potts. Learning Word Vectors for Sentiment Analysis. In the proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies, pages 142–150, Portland, Oregon, USA, June 2011. Association for Computational Linguistics.

该数据集包含 50,000 个电影评论，分为 25,000 个培训样本和 25,000 个测试样本。评论标记为负面（neg）或正面（pos）。此外，正面意味着电影在 IMDb 上收到 `> 6` 星；负面意味着电影收到 `<5` 星。

假设 `../fetch_data.py` 脚本成功运行，以下文件应该可用：

```
import os

train_path = os.path.join('datasets', 'IMDb', 'aclImdb', 'train'
)
test_path = os.path.join('datasets', 'IMDb', 'aclImdb', 'test')
```

现在，让我们通过 scikit-learn 的 `load_files` 函数，将它们加载到我们的活动会话中：

```
from sklearn.datasets import load_files

train = load_files(container_path=(train_path),
                   categories=['pos', 'neg'])

test = load_files(container_path=(test_path),
                  categories=['pos', 'neg'])
```

> 注
>
> 由于电影数据集由 50,000 个单独的文本文件组成，因此执行上面的代码片段可能需要约 20 秒或更长时间。

`load_files` 函数将数据集加载到 `sklearn.datasets.base.Bunch` 对象中，这些对象是 Python 字典：

```
train.keys()
```

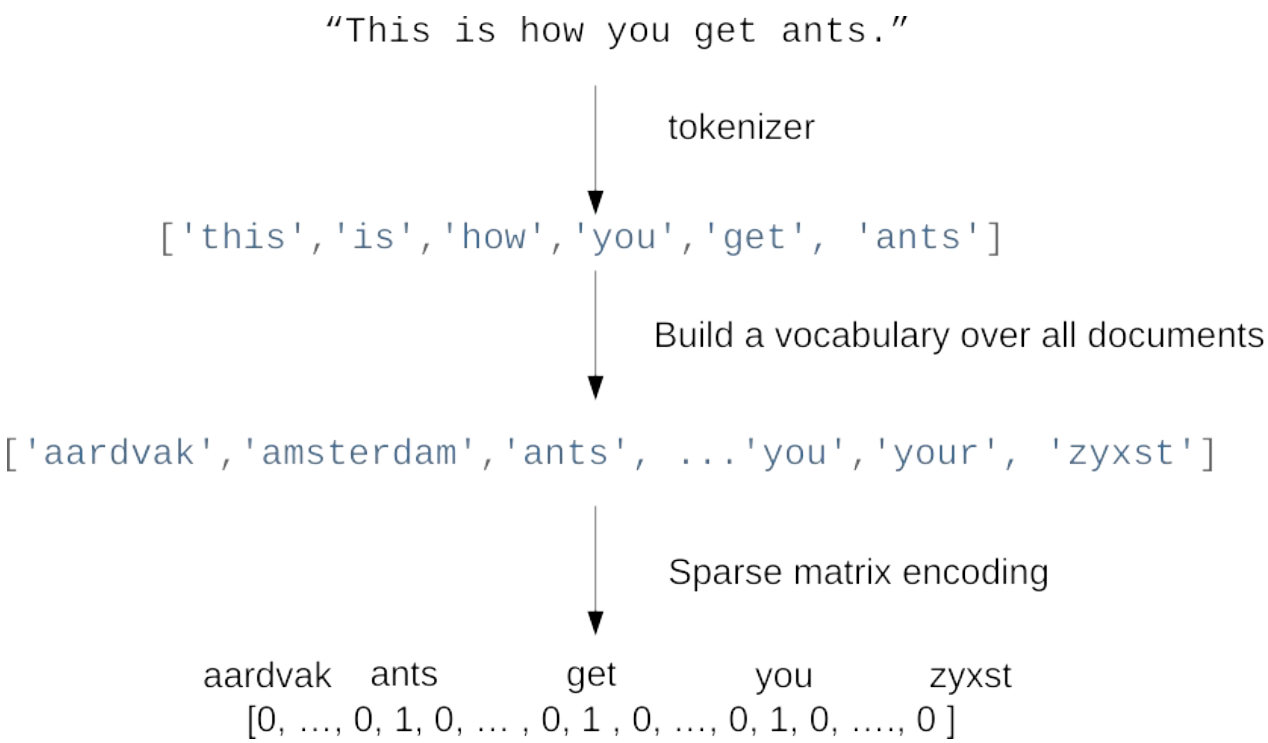
特别是，我们只对 `data` 和 `target` 数组感兴趣。

```
import numpy as np

for label, data in zip(('TRAINING', 'TEST'), (train, test)):
    print('\n\n%s' % label)
    print('Number of documents:', len(data['data']))
    print('\n1st document:\n', data['data'][0])
    print('\n1st label:', data['target'][0])
    print('\nClass names:', data['target_names'])
    print('Class count:',
          np.unique(data['target']), ' -> ',
          np.bincount(data['target']))
```

正如我们在上面所看到的，`target` 数组由整数 0 和 1 组成，其中 0 代表负面，1 代表正面。

## 哈希技巧

回忆一下，使用基于词汇表的向量化器的词袋表示：




要解决基于词汇表的向量化器的局限性，可以使用散列技巧。我们可以使用散列函数和模运算，而不是在 Python 字典中构建和存储特征名称到特征索引的显式映射：

对于哈希技巧的原始论文的更多信息和参考，请见以下网站，以及特定于语言的描述请见这里。

```
from sklearn.utils.murmurhash import murmurhash3_bytes_u32

# encode for python 3 compatibility
for word in "the cat sat on the mat".encode("utf-8").split():
    print("{0} => {1}".format(
        word, murmurhash3_bytes_u32(word, 0) % 2 ** 20))
```

这种映射完全是无状态的，并且输出空间的维度预先明确固定（这里我们使用 `2 ** 20` 的模，这意味着大约 1M 的维度）。这使得有可能解决基于词汇表的向量化器的局限性，既可用于并行化，也可用于在线/核外学习。

`HashingVectorizer` 类是 `CountVectorizer`（或 `use_idf=False` 的 `TfidfVectorizer` 类）的替代品，它在内部使用 murmurhash 哈希函数：

```
from sklearn.feature_extraction.text import HashingVectorizer

h_vectorizer = HashingVectorizer(encoding='latin-1')
h_vectorizer
```

它共享相同的“预处理器”，“分词器”和“分析器”基础结构：

```
analyzer = h_vectorizer.build_analyzer()
analyzer('This is a test sentence.')
```

我们可以将数据集向量化为 `scipy` 稀疏矩阵，就像我们使用 `CountVectorizer` 或 `TfidfVectorizer` 一样，除了我们可以直接调用 `transform` 方法：没有必要拟合，因为 `HashingVectorizer` 是无状态变换器：

```
docs_train, y_train = train['data'], train['target']
docs_valid, y_valid = test['data'][:12500], test['target'][:12500
]
docs_test, y_test = test['data'][12500:], test['target'][12500:]
```

默认情况下，输出的维度事先固定为 `n_features = 2 ** 20`（接近 1M 个特征），来最大限度地减少大多数分类问题的碰撞率，同时具有合理大小的线性模型（`coef_` 属性中的 1M 权重）：

```
h_vectorizer.transform(docs_train)
```

现在，让我们将 `HashingVectorizer` 的计算效率与 `CountVectorizer` 进行比较：

```
h_vec = HashingVectorizer(encoding='latin-1')
%timeit -n 1 -r 3 h_vec.fit(docs_train, y_train)

count_vec =  CountVectorizer(encoding='latin-1')
%timeit -n 1 -r 3 count_vec.fit(docs_train, y_train)
```

我们可以看到，在这种情况下，`HashingVectorizer` 比 `Countvectorizer` 快得多。

最后，让我们在 IMDb 训练子集上训练一个 `LogisticRegression` 分类器：

```
from sklearn.linear_model import LogisticRegression
from sklearn.pipeline import Pipeline

h_pipeline = Pipeline([
    ('vec', HashingVectorizer(encoding='latin-1')),
    ('clf', LogisticRegression(random_state=1)),
])

h_pipeline.fit(docs_train, y_train)

print('Train accuracy', h_pipeline.score(docs_train, y_train))
print('Validation accuracy', h_pipeline.score(docs_valid, y_vali
d))

import gc

del count_vec
del h_pipeline

gc.collect()
```

## 核外学习

核外学习是在不放不进内存或 RAM 的数据集上训练机器学习模型的任务。 这需要以下条件：

具有固定输出维度的特征提取层 提前知道所有类别的列表（在这种情况下，我们只有正面和负面的评论） 支持增量学习的机器学习算法（scikit-learn 中的 `partial_fit` 方法）。

在以下部分中，我们将建立一个简单的批量训练函数来迭代地训练 `SGDClassifier` 。

但首先，让我们将文件名加载到 Python 列表中：

```
train_path = os.path.join('datasets', 'IMDb', 'aclImdb', 'train'
)
train_pos = os.path.join(train_path, 'pos')
train_neg = os.path.join(train_path, 'neg')

fnames = [os.path.join(train_pos, f) for f in os.listdir(train_p
os)] +\
         [os.path.join(train_neg, f) for f in os.listdir(train_n
eg)]

fnames[:3]
```

接下来，让我们创建目标标签数组：

```
y_train = np.zeros((len(fnames), ), dtype=int)
y_train[:12500] = 1
np.bincount(y_train)
```

现在，我们实现 `batch_train` 函数，如下所示：

```
from sklearn.base import clone

def batch_train(clf, fnames, labels, iterations=25, batchsize=10
00, random_seed=1):
    vec = HashingVectorizer(encoding='latin-1')
    idx = np.arange(labels.shape[0])
    c_clf = clone(clf)
    rng = np.random.RandomState(seed=random_seed)

    for i in range(iterations):
        rnd_idx = rng.choice(idx, size=batchsize)
        documents = []
        for i in rnd_idx:
            with open(fnames[i], 'r', encoding='latin-1') as f:
                documents.append(f.read())
        X_batch = vec.transform(documents)
        batch_labels = labels[rnd_idx]
        c_clf.partial_fit(X=X_batch,
                          y=batch_labels,
                          classes=[0, 1])

    return c_clf
```

请注意，我们没有像上一节中那样使用 `LogisticRegression` ，但我们将使用具有 logistic 成本函数的 `SGDClassifier` 。 `SGD` 代表随机梯度下降，这是一种优化算法，它逐样本迭代地优化权重系数，这允许我们一块一块地将数据馈送给分类器。

我们训练 `SGDClassifier` ；使用 `batch_train` 函数的默认设置，它将在 `25 * 1000 = 25000` 个文档上训练分类器。（根据你的机器，这可能需要 `>2` 分钟）

```
from sklearn.linear_model import SGDClassifier

sgd = SGDClassifier(loss='log', random_state=1, max_iter=1000)

sgd = batch_train(clf=sgd,
                  fnames=fnames,
                  labels=y_train)
```

最后，让我们评估一下它的表现：

```
vec = HashingVectorizer(encoding='latin-1')
sgd.score(vec.transform(docs_test), y_test)
```

## 哈希向量化器的限制

使用Hashing Vectorizer可以实现流式和并行文本分类，但也可能会引入一些问题：

- 碰撞会在数据中引入太多噪声并降低预测质量，
- `HashingVectorizer` 不提供“反向文档频率”重新加权（缺少 `use_idf=True` 选项）。
- 没有反转映射，和从特征索引中查找特征名称的简单方法。
- 可以通过增加 `n_features` 参数来控制冲突问题。

可以通过在向量化器的输出上附加 `TfidfTransformer` 实例来重新引入 IDF 加权。然而，用于特征重新加权的 `idf_` 统计量的计算，需要在能够开始训练分类器之前，额外遍历训练集至少一次：这打破了在线学习方案。

缺少逆映射（ `TfidfVectorizer` 的 `get_feature_names()` 方法）更难以解决。这将需要扩展 `HashingVectorizer` 类来添加“跟踪”模式，来记录最重要特征的映射，来提供统计调试信息。

在调试特征提取问题的同时，建议在数据集的小型子集上使用 `TfidfVectorizer(use_idf=False)` ，来模拟具有 `get_feature_names()` 方法且没有冲突问题的 `HashingVectorizer()` 实例。

> 练习
>
> 在我们上面的 `batch_train` 函数的实现中，我们在每次迭代中随机抽取 `k` 个训练样本作为批量，这可以被视为带放回的随机子采样。 你可以修改 `batch_train` 函数，使它无放回地迭代文档，即它在每次迭代中使用每个文档一次。

```
# %load solutions/23_batchtrain.py
```