

Python 机器学习教程

1 基础知识

机器学习在最近是非常火热的词汇，机器学习是英文单词“Machine Learning”（简称 ML）的直译，从字面上便说明了这门技术是让机器进行“学习”的技术。我们知道，机器终究是死的，所谓的“学习”归根结底亦只是人类“赋予”机器的一系列运算。这个“赋予”的过程可以有很多种工具可以实现。Python 正是其中相对容易上手，同时性能又相当不错的一门语言。本节内容主要介绍机器学习相关的一些基础知识，再说明为何要使用 Python 来作为机器学习的工具。最后，我们会提供一个简短易懂的、具有实际意义的例子来给大家提供一个直观的感受。具体而言，本章主要涉及的知识点有：

- 机器学习的定义及基础概念、术语；
- 机器学习一般的处理步骤；
- Python 在机器学习领域的优势。

如前所说，由于近期的各种最新成果，使得“机器学习”成为了非常热门的词汇。机器学习在各种领域的优异表现（围棋界的 Master 是最具代表性的存在），使得各行各业的人们都或多或少地对机器学习产生了兴趣与敬畏。然而与此同时，对机器学习有所误解的群体也日益壮大；他们或将机器学习想得过于神秘，或将它想得过于万能。本节拟对机器学习进行一般性的介绍，同时会说明机器学习中一些常见的术语以方便之后章节的叙述。

1.1 机器学习定义

清晨的一句“今天天气真好”、朋友之间的寒暄“你刚刚是去吃饭了吧”、考试过后的感叹“复习了那么久终有收获”……这些日常生活中随处可见的话语，其背后却已蕴含了“学习”的思想——它们都是利用以往的经验、对未知的新情况作出的有效的决策。而把这个决策的过程交给计算机来做，可以说就是“机器学习”的一个最直观的定义。

先来看看机器学习与以往的计算机工作样式有什么不同。传统的计算机如果想要得到某个结果，需要开发（使用）人员赋予它一串实打实的指令，然后计算机就根据这串指令一步步地执行下去。这个过程因果关系非常明确，只要人的理解不出偏差，运行结果是可以准确预测的。但是在机器学习中，这一传统样式被打破了：计算机确实仍然需要外界赋予它一串指令，但这串指令往往不能直接得到结果；相反，它是一串赋予了机器“学习能力”的指令。在此基础上，计算机需要进一步地接受“数据”，并根据之前人类赋予它的“学习能力”，从中“学习”出最终的结果。这个结果往往是无法仅仅通过直接编程得出的。因此这里就导出了稍微深一点的机器学习的定义：它是一种让计算机利用数据而非指令来进行各种工作的方法。在这背后，最关键的就是“统计”的思想，它所推崇的“相关而非因果”的概念是机器学习的理论根基。在此基础上，机器学习可以说是计算机使用输入给它的数据，利用人类赋予它的算法得到某种模型的过程，其最终的目的则是使用该模型，预测未来未知数据的信息。

既然提到了统计，那么一定的数学理论就不可或缺。机器学习在统计理论下的、比较深刻的本质是：它追求的是合理的假设空间（Hypothesis Space）的选取和模型的泛化

（Generalization）能力：

- ◆ 假设空间，就是模型在数学上的“适用场合”。
- ◆ 泛化能力，就是模型在未知数据上的表现。

注意：上述本质上是计算学习的本质；在其余的理论框架下，机器学习是可以具有不同的内核的。

从上面的讨论可以看出，机器学习和人类思考的过程有或多或少的类似。而机器学习中的神经网络（Neural Network, NN）和卷积神经网络（Convolutional Neural Network, CNN）就是以相应的神经科学的理论背景为依据发展起来的。

需要注意的是，机器学习并非是一个“会学习的机器人”和“具有学习能力的人造人”之类的，这一点从上面诸多讨论也可以明晰。相反的，它是被人类利用的、用于发掘数据背后信息的工具，即初级的、具有灵活性的处理工具。

1.2 常用术语

机器学习领域有着许多非常基本的术语，这些术语在外人听来可能相当高深莫测。它们事实上也可能拥有非常复杂的数学背景，但需要知道：它们往往也拥有着相对浅显平凡的直观理解（上一小节的假设空间和泛化能力就是两个例子）。本小节会对这些常用的基本术语进行说明与解释，它们背后的数学理论会有所阐述，但不会涉及过于本质的东西。

1) 数据类定义

正如前文反复强调的，数据在机器学习中发挥着不可或缺的作用；而用于描述数据的术语有好几个，需要被牢牢记住的如下。

- ◆ “数据集”（Data Set），就是数据的集合，数据集中每一条单独的数据被称为“样本”（Sample）。若没有进行特殊说明，本书都会假设数据集中样本之间在各种意义下相互独立。实际中，除了某些特殊的模型（如隐马尔可夫模型和条件随机场），该假设在大多数场景下都是相当合理的。
- ◆ 对于每个样本，它通常具有一些“属性”（Attribute）或者说“特征”（Feature），特征所具体取的值就被称为“特征值”（Feature Value）。
- ◆ 特征和样本所张成的空间被称为“特征空间”（Feature Space）和“样本空间”（Sample Space），可以把它们简单地理解为特征和样本“可能存在的空间”或“可能存在的场景”。
- ◆ 相对应的，有“标签空间”（Label Space），它描述了模型的输出“可能存在的空间”；当模型是分类器时，我们通常会称之为“类别空间”。
- ◆ 使用（开发）过程中，数据集又可以分为以下三类：
 - 训练集（Training Set）；顾名思义，它是总的数据集中用来训练我们模型的部分。虽说将所有数据集都拿来当作训练集也无不可，不过为了提高及合理评估模型的泛化能力，我们通常只会取数据集中的一部分来当训练集。
 - 测试集（Test Set）；顾名思义，它是用来测试、评估模型泛化能力的部分。测试集不会用在模型的训练部分，换句话说，测试集相对于模型而言是“未知”的，所以拿它来评估模型的泛化能力是相当合理的。
 - 交叉验证集（Cross-Validation Set, CV Set）；这是比较特殊的一部分数据，它是用来调整模型具体参数的。

注意：获取数据集这个过程很繁琐，处理过程工作量比较大；推荐一个著名数据集的网站：<http://archive.ics.uci.edu/ml/datasets.html>，后续内容会用到其中一些合适的数据集来评估我们自己实现的模型。

通过具体实例理解上述概念。假设小明是一个在北京读了一年书的学生，某天他想通过宿舍窗外的风景（能见度、温度、湿度、路人戴口罩的情况等）来判断当天的雾霾情况并据此决定是否戴口罩。此时，他过去一年的经验就是他拥有的数据集，过去一年中每一天的情况就是一个样本。“能见度”、“温度”、“湿度”、“路人戴口罩的情况”就是四个特征，而（能见度）“低”、（温度）“低”、（湿度）“高”、（路人戴口罩的）“多”就是相对应的特征值。现在小明想了想，决定在脑中建立一个模型来帮自己做决策，该模型将利用过去一年的数据集来对如今的情况做出“是否戴口罩”的决策。此时小明可以用过去一年中 8 个月的数据量来做训练集、2 个月的量来做测试集、2 个月的量来做交叉验证集，那么小明就需要不断地思考（训练模型）下列问题：

- ◆ 用训练集训练出的模型是怎样的？
- ◆ 该模型在交叉验证集上的表现怎么样？
 - 如果足够好了，那么思考结束（得到最终模型）。
 - 如果不够好，那么根据模型在交叉验证集上的表现，重新思考（调整模型参数）。

最后，小明可能会在测试集上评估自己刚刚思考后得到的模型的性能，然后根据这个性能和模型做出的“是否戴口罩”的决策来综合考虑自己到底戴不戴口罩。

2) 功能类定义

功能类定义主要包括假设空间和泛化能力方面的一些重要概念：

- ◆ 泛化能力针对的其实是学习方法，它用于衡量该学习方法学习到的模型在整个样本空间上的表现。

这一点当然是十分重要的，因为我们拿来训练模型的数据终究只是样本空间的一个很小的采样，如果只是过分专注于它们，就会出现所谓的“过拟合”（Over Fitting）的情况。当然，如果过分罔顾训练数据，又会出现“欠拟合”（Under Fitting）。可以用一张图来直观地感受过拟合和欠拟合（如图 1 所示，左为欠拟合，右为过拟合）。

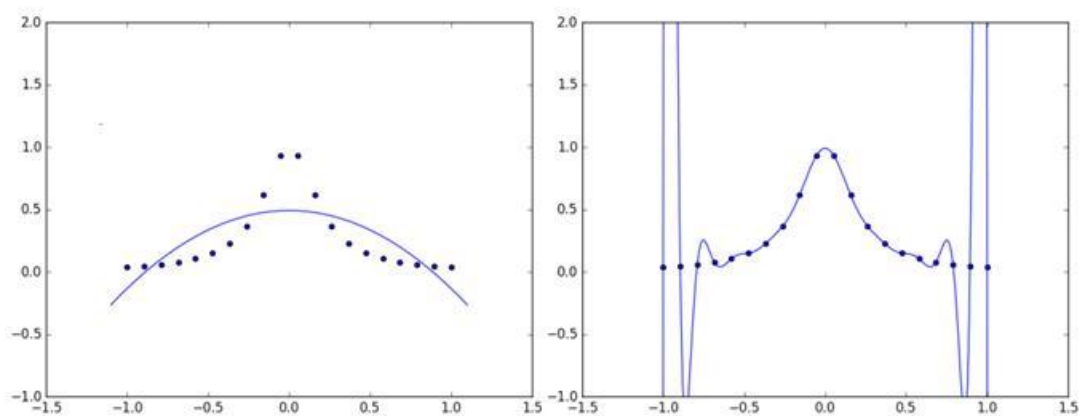


图 1 欠拟合与过拟合

所以需要“张弛有度”，找到最好的那个平衡点。统计学习中的结构风险最小化（Structural Risk Minimization, SRM）就是研究这个的，它和传统的经验风险最小化（Empirical Risk Minimization, ERM）相比，注重于对风险上界的最小化，而不是单纯地使经验风险最小化。它有一个原则：在使风险上界最小的函数子集中挑选出使经验风险最小的函数。而这个函数

子集，正是我们之前提到过的假设空间。

注意：所谓经验风险，可以理解为训练数据集上的风险。相对应的，ERM 则可以理解为只注重训练数据集的学习方法，它的理论基础是经验风险在某种足够合理的数学意义上一致收敛于期望风险，亦即所谓的“真正的”风险。

关于 SRM 和 ERM 的详细内容请参阅相关文档。目前只需建立简单直观的认识：为了使我们学习方法训练出的模型泛化能力足够好，需要对模型做出一定的“限制”，而这个“限制”就表现在假设空间的选取上。一个非常普遍的做法是对模型的复杂度做出一定的惩罚，从而使模型趋于精简。

相比起通过选取合适的假设空间来规避过拟合，使用交叉验证（Cross Validation）则可以帮助我们知道过拟合的程度，从而帮助我们选择合适的模型。有关交叉验证的详细内容请参见第二章相关章节。

1.3 机器学习的重要性

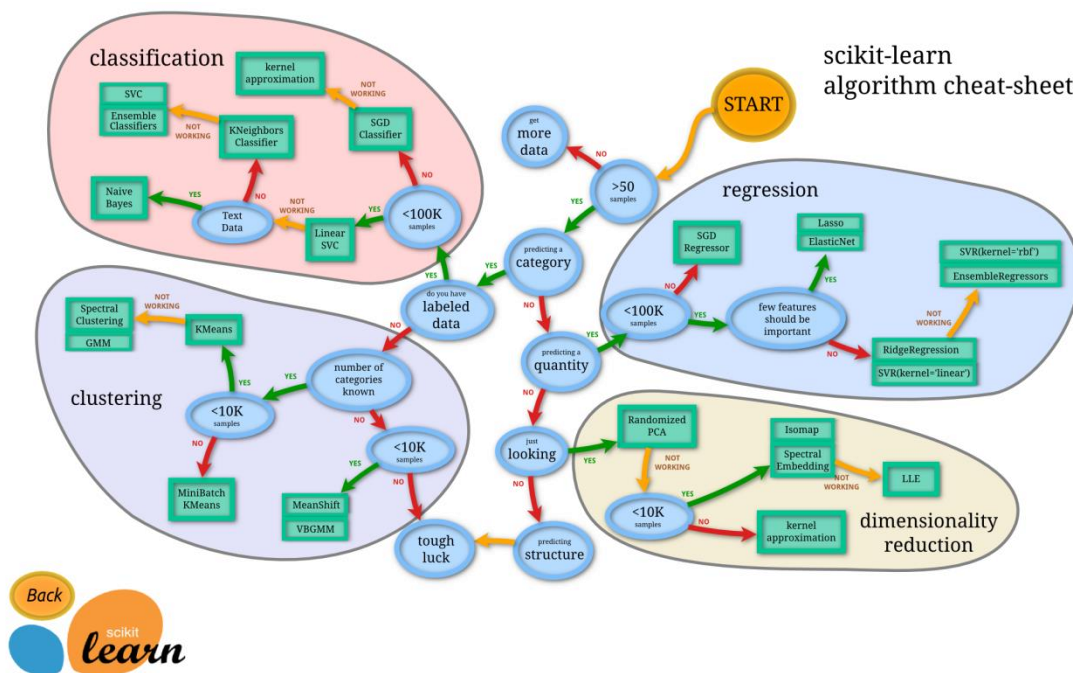
前面了解了机器学习的基本概念，但为什么要用机器学习，即机器学习的好处及重要性在哪里？这个问题可以从人类历史的发展可以找到答案：人类的发展是从简单体力劳动向复杂脑力劳动的过渡转化，所有种类的工作都是这样，编码开发、信息处理也不例外。过去的工作基本上都有着明确的定义，告诉你这一步怎么做、下一步再怎么做。而如今这一类的工作已经越来越少，取而代之的是更为宽泛模糊的、概念性的东西，比如说“将本季度的产品推向最合适的市场，在最大化期望利润的同时，尽量做到风险最小化”这种需求。想要完成好这样的任务，需要获取相应的数据；虽说网络的存在让我们能够得到数之不尽的数据，然而从这些数据中获得信息与知识却不是一项简单的工作。我们当然可以人工地、仔细地逐项甄选，但这样显然就又回到了最初的原点。机器学习这门技术，可以说正因此应运而生。

我们就看一下机器学习具体的应用范围，从中大概能够比较直观地看出机器学习的强大与重要。目前，机器学习已经应用到了各个角落、包括但不限于：

- 机器视觉，比如深度学习应用、自动目标识别、物体追踪等；
- 语音识别，微软 Cortana、讯飞语音输入等；
- 数据挖掘，大数据相关的领域，比如自动推荐等；

机器学习还能够进行模式识别、自然语言处理，等等，之前提到过的围棋界的 Master 和最新人工智能在德州扑克上的表现亦无不呈现着机器学习强大的潜力。简单来说，至少在目前，机器学习在计算机领域是绝对的热点，并在相当长的一段时间内保持强大的生命力。因此，值得投入精力去学习、掌握。

1.4 机器学习主要内容



按应用目的，机器学习分为四大块，分别是：**classification**（分类）、**regression**（回归）、**clustering**（聚类）、**dimensionality reduction**（降维）。

给定一个样本特征，我们希望预测其对应的属性值，如果属性是离散的，那么这就是一个分类问题，反之，如果属性是连续的实数，这就是一个回归问题。

如果给定一组样本特征，我们没有对应的属性值，而是想发掘这组样本在 维空间的分布，比如分析哪些样本靠的更近，哪些样本之间离得很远，这就是属于聚类问题。

如果我们想用维数更低的子空间来表示原来高维的特征空间，那么这就是降维问题。

无论是分类还是回归，都是想建立一个预测模型，给定一个输入（特征），得到一个输出（结果）。不同点在于，在分类问题中，（期望）输出是离散的；而在回归问题中（期望）输出是连续的。总体来说，两种问题的学习算法都很类似。所以，在上述内容图谱上，我们看到在分类问题中用到的学习算法，在回归问题中也能使用。

分类问题最常用的学习算法包括 **SVM** (支持向量机), **SGD** (随机梯度下降算法), **Bayes** (贝叶斯估计), **KNN** 等。而回归问题也能使用 **SVR**, **SGD**, **Ensemble** 等算法，以及其它线性回归算法。

聚类也是分析样本的属性，有点类似分类，不同的就是分类在预测之前是知道结果输出的范围，或者说知道到底有几个类别，而聚类是不知道属性的范围的。所以分类也常常被称为 **supervised learning**，而聚类就被称为 **unsupervised learning**。

聚类事先不知道样本的属性范围，只能凭借样本在特征空间的分布来分析样本的属性。这种问题一般更复杂。而常用的算法包括 **k-means** (K-均值), **GMM** (高斯混合模型) 等。

降维是机器学习另一个重要的领域，降维有很多重要的应用，特征的维数过高，会增加训练的负担与存储空间，降维就是希望去除特征的冗余，用更加少的维数来表示特征。降维算法最基础的就是 **PCA** 了，后面的很多算法都是以 **PCA** 为基础演化而来。

1.5 Python 的优势

之前内容叙述了 Python 的优点及基本使用方法，但在算法仿真、数据处理领域，还存在诸如 **MATLAB** 和 **Mathematica** 这样的高级程序语言，它们对机器学习的支持也不错，

MATLAB 甚至还自带许多机器学习的应用。但是，MATLAB 的正版软件需要花费数千美元，全模块费用高达 5w\$ 左右。与之相对的，Python 是开源项目，几乎所有必要的组件都是完全免费的。同时，Python 是胶水语言，因此其运行速度问题，可以通过更快更底层的语言，比如 C 和 C++ 扩展来解决。

另外，使用 Python 来学习机器学习是和“不要过早优化”这句编程界的金句有着异曲同工之妙的。Python 存在的，（几乎）唯一的速度缺陷，在初期进行快速检验算法、思想正误及开发工作时，其实基本上是不考虑的。其中的道理是显而易见的：如果解决问题的思想存在问题，那么即使拼命去提高程序的运行效率，也只能使问题越来越大而已。这种时候，先使用 Python 进行快速实现，有必要时再用底层代码重写核心代码，从各方面来说都是一个更好的选择。而 Python 语言近几年的快速发展，尤其是在学习领域的垄断性扩张也验证了上述思想的有效性。因此，本课程以 Python 作为基本开发语言。

2 快速入门

本节内容运用 Python 解决一个简单的实际问题，以便对机器学习的整个过程有一个具体直观的感受。

2.1 问题描述

该问题来自 Coursera 上的斯坦福大学机器学习课程，其叙述如下：现有 47 个房子的面积和价格，需要建立一个模型对新的房价进行预测。稍微翻译问题，可以得知：

- 输入数据为一维数据，即房子的面积。
- 输出结果也是一维数据，即房子的价格。
- 需要做的处理过程是根据已知的房子的面积和价格的关系进行机器学习。

下面我们就来一步步地进行操作。

2.2 获取与处理数据

原始数据集的前 10 个样本如表 1.1 所示，这里房子面积和房子价格的单位可以随意定夺，因为它们不会对结果造成影响。

表 1.1 房价数据集

房子面积	房子价格	房子面积	房子价格
2104	399900	1600	329900
2400	369000	1416	232000
3000	539900	1985	299900
1534	314900	1427	198999
1380	212000	1494	242500

完整的数据集可以参见 `Data/prices.txt`。虽然该数据集比较简单，但可以看到其中的数字都相当大。保留它原始形式确实有可能是有必要的，但一般而言，我们应该对它做简单的

处理以期降低问题的复杂度。在这个例子里，采取常用的将输入数据标准化的做法，其数学公式为：

$$X = \frac{X - \bar{X}}{\text{std}(X)}$$

其中， \bar{X} 表示 x （房子面积）的均值， $\text{std}(X)$ 表示 x 的标准差，代码如下：

```
#导入需要用到的库
import numpy as np
import matplotlib.pyplot as plt

# 定义存储输入数据（x）和目标数据（y）的数组
x, y = [], []
# 遍历数据集，变量 sample 对应的正是一个个样本
for sample in open("../_Data/prices.txt", "r"):
# 由于数据是用逗号隔开的，所以调用 Python 中的 split 方法并将逗号作为参数传入
    _x, _y = sample.split(",")
    # 将字符串数据转化为浮点数
    x.append(float(_x))
    y.append(float(_y))
# 读取完数据后，将它们转化为 Numpy 数组以方便进一步的处理
x, y = np.array(x), np.array(y)
# 标准化
x = (x - x.mean()) / x.std()
# 将原始数据以散点图的形式画出
plt.figure()
plt.scatter(x, y, c="g", s=6)
plt.show()
```

上面这段代码的运行结果如图 2 所示。

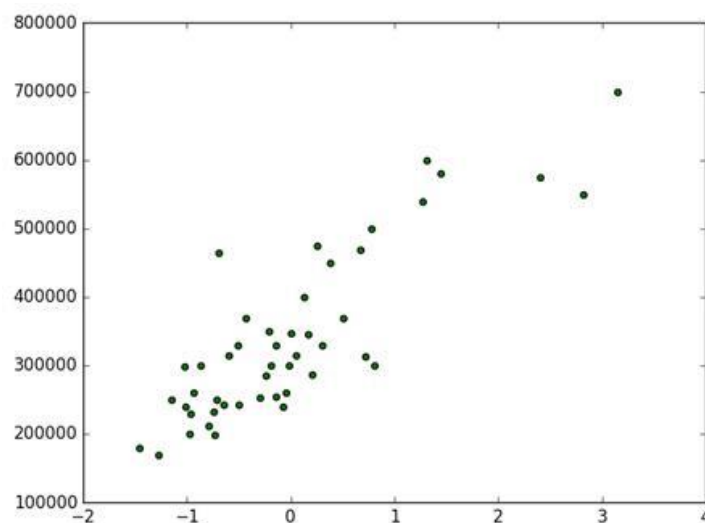


图 2 预处理后的数据散点图

上图中横轴是标准化后的房子面积，纵轴是房子价格。至此，完成了机器学习任务的第

一步：数据预处理。

2.3 选择与训练模型

准备好数据之后，下一步就要开始选择相应的学习方法和模型了。幸运的是，通过可视化原始数据，可以非常直观地感受到：很有可能通过线性回归（Linear Regression）中的多项式拟合来得到一个不错的结果。其模型的数学表达式如下。

注意：用多项式拟合散点只是线性回归的很小的一部分，但是它的直观意义比较明显。考虑到问题比较简单，我们才选用了多项式拟合。线性回归的详细内容见后面相关章节。

$$f(x|p; n) = p_0x^n + p_1x^{n-1} + \cdots + p_{n-1}x + p_n$$

$$L(p; n) = \frac{1}{2} \sum_{i=1}^m [f(x|p; n) - y]^2$$

其中 $f(x|p; n)$ 就是我们的模型， p 、 n 都是模型的参数，其中 p 是多项式 f 的各个系数， n 是多项式的次数。 $L(p; n)$ 则是模型的损失函数，这里采用了常见的平方损失函数，即欧氏距离（或说向量的二范数）。 x 、 y 则分别是输入向量和目标向量；在我们这个样例中， x 、 y 这两个向量都是 47 维的向量，分别由 47 个不同的房子面积、房子价格所构成。

在确定好模型后，就可以开始编写代码来进行训练了。在训练过程中，如果数据样本量足够，一般采用交叉验证方法来划分数据集。

对于大多数机器学习算法，训练就是最小化损失函数的过程，此处我们的目的就是让上面定义的 $L(p; n)$ 最小。在数理统计领域里有专门的理论研究这种回归问题，并对常用的正规方程更是给出了一个简单的通用解。Python 的 Numpy 库使得这个训练过程更加简单，示例如下：

```
# 在(-2,4)这个区间上取 100 个点作为画图的基础
x0 = np.linspace(-2, 4, 100)
# 利用 Numpy 的函数定义训练并返回多项式回归模型的函数
# deg 参数代表着模型参数中的 n，亦即模型中多项式的次数
# 返回的模型能够根据输入的 x（默认是 x0），返回相对应的预测的 y
def get_model(deg):
    return lambda input_x=x0: np.polyval(np.polyfit(x, y, deg), input_x)
```

这里需要解释 Numpy 里面带的两个函数：polyfit 和 polyval 的用法。

- **polyfit(x, y, deg)**：该函数会返回使得上述定义的损失函数（注：该公式中的 x 和 y 就是输入的 x 和 y ）最小的参数 p ，亦即多项式的各项系数。换句话说，该函数就是模型的训练函数。
- **polyval(p, x)**：根据多项式的各项系数 p 和多项式中 x 的值，返回多项式的值 y 。

2.4 评估与可视化结果

模型训练完成后，我们就要尝试判断各种参数下模型的优劣。为简洁起见，我们采用 $n=1, 4, 10$ 这三组参数进行评估。

1) 结果评估

由于我们训练的目的是最小化损失函数, 所以用损失函数来衡量模型的好坏似乎是一个合理的做法。代码示例如下:

```
# 根据参数 n、输入的 x、y 返回相对应的损失
def get_cost(deg, input_x, input_y):
    return 0.5 * ((get_model(deg)(input_x) - input_y) ** 2).sum()
# 定义测试参数集并根据它进行各种实验
test_set = (1, 4, 10)
for d in test_set:
    # 输出相应的损失
    print(get_cost(d, x, y))
```

所得的结果是: 当 $n=1, 4, 10$ 时, 损失的头两位数字分别为 96、94 和 75。这么看来似乎是 $n=10$ 优于 $n=4$, 而 $n=1$ 最差, 但从图 3 可以看出, 似乎直接选择 $n=1$ 作为模型的参数才是最好的选择。这里矛盾的来源正是前文所提到过的过拟合情况。

2) 结果可视化

可以通过图形的形式直观地了解是否出现过拟合, 示例如下:

```
# 画出相应的图像
plt.scatter(x, y, c="g", s=20)
for d in test_set:
    plt.plot(x0, get_model(d)(), label="degree = {}".format(d))
# 将横轴、纵轴的范围分别限制在(-2,4)、( $10^5, 8 \times 10^5$ )
plt.xlim(-2, 4)
plt.ylim(1e5, 8e5)
# 调用 legend 方法使曲线对应的 label 正确显示
plt.legend()
plt.show()
```

运行结果如图 3 所示。

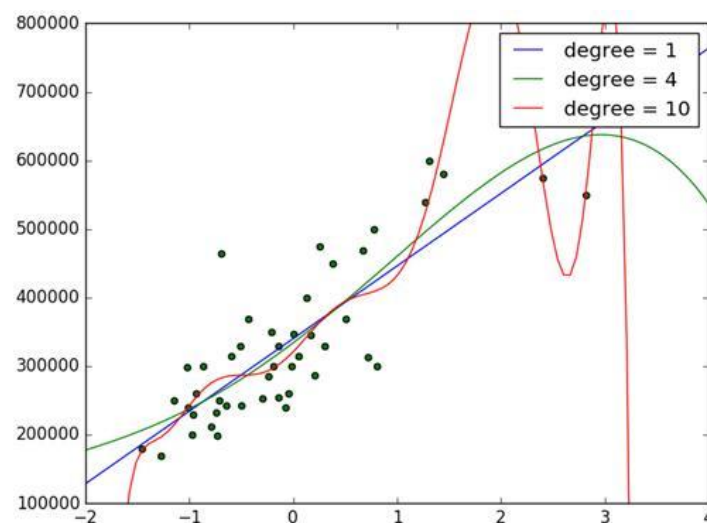


图 3 线性回归的可视化

其中，三条线分别代表 $n=1$ 、 $n=4$ 、 $n=10$ 的情况（图 1.10 的右上角亦有说明）。可以看出，从 $n=4$ 开始模型就已经开始出现过拟合现象了，到 $n=10$ 时模型已经变得非常不合理。

至此，可以说这个问题就已经基本上解决了。在这个样例里面，除了交叉验证，我们涵盖了机器学习中的大部分主要步骤，此处没有使用交叉验证的原因是数据太少。整个代码部分加起来总共 40~50 行，是一个比较合适的代码量，符合快速开发，快速验证的思想。

2.5 交叉验证

交叉验证（Cross-Validation）是按某种规则将原始数据(dataset)进行分组,一部分做为训练集(train set),剩余部分做为验证集(validation set or test set)。训练时，首先用训练集对分类器进行训练，训练完成后再利用验证集来测试训练得到的模型(model),以此来做为评价分类器的性能指标。

交叉验证对于人工智能，机器学习，模式识别，分类器等研究都具有很强的指导与验证意义。基本方法包括：简单交叉验证、k 折交叉验证、留一法。

1) 简单交叉验证

步骤如下：

- 1 从全部的训练数据 S 中随机选择指定数量（比例）的样例作为训练集 $train$ ，剩余的作为测试集 $test$ 。

- 2 通过对测试集训练，得到假设函数或者模型。

- 3 在测试集对每一个样本根据假设函数或者模型，得到训练集的分类，求出分类正确率。

- 4 选择具有最大分类率的模型或者假设。

这种方法称为 **hold-out cross validation** 或者称为简单交叉验证。由于测试集和训练集是分开的，就避免了过拟合的现象

2) k 折交叉验证

k 折交叉验证（k-fold cross validation）步骤如下：

- 1 将全部训练集 S 分成 k 个不相交的子集，假设 S 中的训练样例个数为 m ，那么每一个子集有 m/k 个训练样例，相应的子集称作 $\{s_1, s_2, \dots, s_k\}$ ；

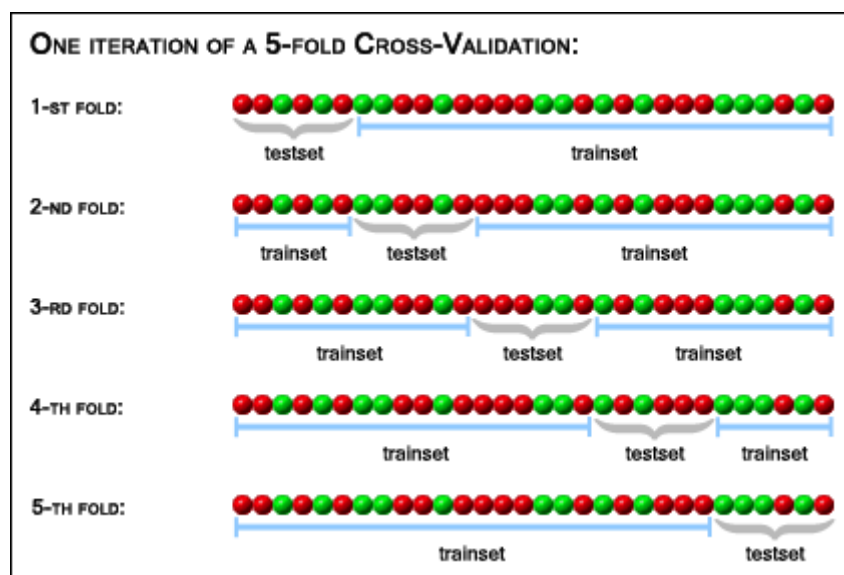
- 2 每次从分好的子集中里面，拿出一个作为测试集，其它 $k-1$ 个子集作为训练集；

- 3 根据训练集训练出模型或者假设函数；

- 4 把这个模型放到测试集上，得到分类率；

- 5 计算 k 次求得的分类率的平均值，作为该模型或者假设函数的真实分类率；

这个方法充分利用了所有样本。但计算比较繁琐，需要训练 k 次，测试 k 次。下图是 $K=5$ 时的验证过程示意图：



当 $k=10$ 时，就是常用的 10 折交叉验证方法：将数据集分成十分，轮流将其中 9 份做训练 1 份做测试，10 次的结果的均值作为对算法精度的估计，一般还需要进行多次 10 倍交叉验证 求均值，例如 10 次 10 倍交叉验证，更精确一点。

- 优点：K-CV 可以有效的避免过学习以及欠学习状态的发生,最后得到的结果也比较具有说服力。
- 缺点：K 值选取需要根据经验来定。

3) 留一法

留一法（LOO-CV, leave-one-out cross validation）就是每次只留下一个样本做测试集，其它样本做训练集，如果有 k 个样本，则需要训练 k 次，测试 k 次。

留一法计算最繁琐，但样本利用率最高。适合于小样本的情况。

- 优点：相比于前面的 K-CV 有两个明显的优点：1) 每一回合中几乎所有的样本皆用于训练模型,因此最接近原始样本的分布,这样评估所得的结果比较可。2) 实验过程中没有随机因素会影响实验数据,确保实验过程是可以被复制的。
- 缺点：计算成本高,因为需要建立的模型数量与原始数据样本数量相同,当原始数据样本数量相当多时,LOO-CV 在实作上便有困难几乎就是不显示,除非每次训练分类器得到模型的速度很快,或是可以用并行化计算减少计算所需的时间。

2.6 基于 SKLearn 库的处理过程

传统的机器学习任务从开始到建模的一般流程是：获取数据 -> 数据预处理 -> 训练建模 -> 模型评估 -> 预测、分类。本节我们将依据传统机器学习的处理流程，利用 SKLearn 机器学习库进行对应的处理，了解 SKLearn 库的使用，快速开始后续的机器学习内容。

1) 获取数据

A) 导入 sklearn 数据集

sklearn 中包含了大量的优质的数据集，在学习机器学习的过程中，可以通过使用这些

数据集实现出不同的模型，从而提高动手实践能力，同时这个过程也可以加深对理论知识的理解和把握。

要使用 sklearn 中的数据集，需要导入 datasets 模块：

```
from sklearn import datasets
```

下图中包含了大部分 sklearn 中数据集，调用方式也在图中给出：

	数据集名称	调用方式	适用算法	数据规模
小数据集	波士顿房价数据集	load_boston()	回归	506*13
	鸢尾花数据集	load_iris()	分类	150*4
	糖尿病数据集	load_diabetes()	回归	442*10
	手写数字数据集	load_digits()	分类	5620*64
大数据集	Olivetti脸部图像数据集	fetch_olivetti_faces()	降维	400*64*64
	新闻分类数据集	fetch_20newsgroups()	分类	-
	带标签的人脸数据集	fetch_lfw_people()	分类；降维	-
	路透社新闻语料数据集	fetch_rcv1()	分类	804414*4723 6
注：小数据集可以直接使用，大数据集在第一次使用的时候会下载				

用 iris 的数据来举例：

```
iris = datasets.load_iris() # 导入数据集
X = iris.data # 获得其特征向量
y = iris.target # 获得样本 label
```

B) 创建数据集

除了可以使用 sklearn 自带的数据集，还可以自己去创建训练样本，具体用法参见《Dataset loading utilities》，在 sklearn 中的 samples generator 包含的大量创建样本数据的方法：

- make_blobs: 多类单标签数据集，为每个类分配一个或多个正太分布的点集；
 - make_classification: 多类单标签数据集，为每个类分配一个或多个正太分布的点集，提供了为数据添加噪声的方式，包括维度相关性，无效特征以及冗余特征等；
 - make_gaussian-quantiles: 将一个单高斯分布的点集划分为两个数量均等的点集，作为两类；
 - make_hastie-10-2: 产生一个相似的二元分类数据集，有 10 个维度；
 - make_circle 和 make_moon 产生二维二元分类数据集来测试某些算法的性能，可以为数据集添加噪声，可以为二元分类器产生一些球形判决界面的数据。
- 更多生成数据的方法，请查阅文档。

下面用分类问题的样本生成器举例子：

```
from sklearn.datasets.samples_generator import make_classification

X, y = make_classification(n_samples=6, n_features=5,
                           n_informative=2, n_redundant=2, n_classes=2, n_clusters_per_class=2,
                           scale=1.0, random_state=20)

# n_samples: 指定样本数
# n_features: 指定特征数
# n_classes: 指定几分类
# random_state: 随机种子，使得随机状可重
>>> for x_, y_ in zip(X, y):
        print(y_, end=': ')
        print(x_)

0: [-0.6600737 -0.0558978  0.82286793  1.1003977 -0.93493796]
1: [ 0.4113583  0.06249216 -0.90760075 -1.41296696  2.059838 ]
1: [ 1.52452016 -0.01867812  0.20900899  1.34422289 -1.61299022]
0: [-1.25725859  0.02347952 -0.28764782 -1.32091378 -0.88549315]
0: [-3.28323172  0.03899168 -0.43251277 -2.86249859 -1.10457948]
1: [ 1.68841011  0.06754955 -1.02805579 -0.83132182  0.93286635]
```

C)动态获取

以下是从 UCI 机器学习数据仓库中下载的数据。样例：

```
import numpy as np
import urllib
# url with dataset
url = "http://archive.ics.uci.edu/ml/machine-learning-databases/pima-indians-diabetes/pima-indians-diabetes.data"
# download the file
raw_data = urllib.urlopen(url)
# load the CSV file as a numpy matrix
dataset = np.loadtxt(raw_data, delimiter=",")
# separate the data from the target attributes
X = dataset[:,0:7]
y = dataset[:,8]
```

我们要使用该数据集作为例子，将特征矩阵作为 X，目标变量作为 y。

注意事项：

- (1) 可以用浏览器打开那个 url，把数据文件保存在本地，然后直接用 `np.loadtxt('data.txt', delimiter=",")` 就可以加载数据了；
- (2) `X = dataset[:, 0:7]` 的意思是：把 dataset 中的所有行，所有 0-7 列的数据都保存在 X 中；

2) 数据预处理

数据预处理阶段是机器学习中不可缺少的一环，它会使得数据更加有效的被模型或者评估器识别。同前，使用之前需要先导入相关模块：

```
from sklearn import preprocessing
```

A) 数据归一化

大多数机器学习算法中的梯度方法对于数据的缩放和尺度都是很敏感的，在开始跑算法之前，我们应该进行归一化或者标准化的过程，这使得特征数据缩放到 0-1 范围中

为了使得训练数据的标准化规则与测试数据的标准化规则同步，`preprocessing` 中提供了很多 `Scaler`：

```
data = [[0, 0], [0, 0], [1, 1], [1, 1]]
# 1. 基于 mean 和 std 的标准化
scaler = preprocessing.StandardScaler().fit(train_data)
scaler.transform(train_data)
scaler.transform(test_data)

# 2. 将每个特征值归一化到一个固定范围
scaler = preprocessing.MinMaxScaler(feature_range=(0,
1)).fit(train_data)
scaler.transform(train_data)
scaler.transform(test_data)
#feature_range: 定义归一化范围，注用（）括起来
```

具体解释参考 <http://scikit-learn.org/stable/modules/preprocessing.html>：

B) 正则化 (normalize)

当你想要计算两个样本的相似度时必不可少的一个操作，就是正则化。其思想是：首先求出样本的 p-范数，然后该样本的所有元素都要除以该范数，这样最终使得每个样本的范数都为 1。

```
>>> X = [[ 1., -1.,  2.],
...      [ 2.,  0.,  0.],
...      [ 0.,  1., -1.]]
>>> X_normalized = preprocessing.normalize(X)

>>> X_normalized
array([[ 0.40..., -0.40...,  0.81...],
       [ 1.    ...,  0.    ...,  0.    ...],
       [ 0.    ...,  0.70..., -0.70...]])
```

C) one-hot 编码

one-hot 编码是一种对离散特征值的编码方式，在 LR 模型中常用到，用于给线性模型增加非线性能力。

```
data = [[0, 0, 3], [1, 1, 0], [0, 2, 1], [1, 0, 2]]
```



```
encoder = preprocessing.OneHotEncoder().fit(data)
enc.transform(data).toarray()
```

3) 数据集拆分

在得到训练数据集时，通常我们经常会把训练数据集进一步拆分成训练集和验证集，这样有助于我们模型参数的选取。

```
# 作用：将数据集划分为 训练集和测试集
# 格式：train_test_split(*arrays, **options)
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.3, random_state=42)
"""
参数
---
arrays: 样本数组，包含特征向量和标签
test_size:
    float-获得多大比重的测试样本 （默认：0.25）
    int - 获得多少个测试样本
train_size: 同 test_size
random_state:
    int - 随机种子（种子固定，实验可复现）
shuffle - 是否在分割之前对数据进行洗牌（默认 True）

返回
---
分割后的列表，长度=2*len(arrays),
    (train-test split)
"""
```

4) 定义模型

在这一步我们首先要分析自己数据的类型，搞清出你要用什么模型来做，然后我们就可以在 `sklearn` 中定义模型了。`sklearn` 为所有模型提供了非常相似的接口，这样使得我们可以更加快速的熟悉所有模型的使用法。在这之前我们先来看看模型的常用属性和功能：

```
# 拟合模型
model.fit(X_train, y_train)
# 模型预测
model.predict(X_test)

# 获得这个模型的参数
model.get_params()
```

为模型进行打分

```
model.score(data_X, data_y) # 线性回归: R square; 分类问题: acc
```

A) 线性回归

```
from sklearn.linear_model import LinearRegression
```

定义线性回归模型

```
model = LinearRegression(fit_intercept=True, normalize=False,  
                        copy_X=True, n_jobs=1)
```

"""

参数

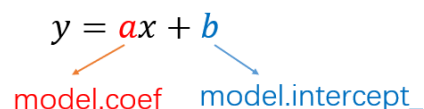
fit_intercept: 是否计算截距。False-模型没有截距

normalize: 当 fit_intercept 设置为 False 时, 该参数将被忽略。 如果为真, 则回归前的回归系数 X 将通过减去平均值并除以 12-范数而归一化。

n_jobs: 指定线程数

"""

$$y = ax + b$$


model.coef_ model.intercept_

B) 逻辑回归 LR

定义及说明: 大多数问题都可以归结为二元分类问题。这个算法的优点是可以给出数据所在类别的概率。

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

定义逻辑回归模型

```
model = LogisticRegression(penalty='l2', dual=False, tol=0.0001,  
                          C=1.0,
```

```
                          fit_intercept=True, intercept_scaling=1, class_weight=None,
```

```
                          random_state=None, solver='liblinear', max_iter=100,
```

```
multi_class='ovr',
```

```
verbose=0, warm_start=False, n_jobs=1)
```

"""参数

penalty: 使用指定正则化项 (默认: l2)

dual: n_samples > n_features 取 False (默认)

C: 正则化强度的反, 值越小正则化强度越大

n_jobs: 指定线程数

random_state: 随机数生成器

fit_intercept: 是否需要常量

"""

样例：

```
from sklearn import metrics
from sklearn.linear_model import LogisticRegression
model = LogisticRegression()
model.fit(X, y)
print('MODEL')
print(model)
# make predictions
expected = y
predicted = model.predict(X)
# summarize the fit of the model
print('RESULT')
print(metrics.classification_report(expected, predicted))
print('CONFUSION MATRIX')
print(metrics.confusion_matrix(expected, predicted))
```

结果：

```
1 MODEL
2 LogisticRegression(C=1.0, class_weight=None, dual=False,
fit_intercept=True,
3         intercept_scaling=1, max_iter=100, multi_class='ovr',
4         penalty='l2', random_state=None, solver='liblinear',
tol=0.0001,
5         verbose=0)
6 RESULT
7         precision    recall  f1-score   support
8
9         0.0         1.00      1.00      4
10        1.0         1.00      1.00      6
11
12 avg / total         1.00      1.00      1.00     10
13
14 CONFUSION MATRIX
15 [[4 0]
16  [0 6]]
```

输出结果中的各个参数信息，可以参考官方文档。

备注：逻辑回归（[官方文档](#)）

C) 朴素贝叶斯算法 NB

定义及说明：这也是著名的机器学习算法，该方法的任务是还原训练样本数据的分布密度，其在多类别分类中有很好的效果。

```

from sklearn import naive_bayes
model = naive_bayes.GaussianNB() # 高斯贝叶斯
model = naive_bayes.MultinomialNB(alpha=1.0, fit_prior=True,
class_prior=None)
model = naive_bayes.BernoulliNB(alpha=1.0, binarize=0.0,
fit_prior=True, class_prior=None)
"""
文本分类问题常用 MultinomialNB
参数
---
alpha: 平滑参数
fit_prior: 是否要学习类的先验概率; false-使用统一的先验概率
class_prior: 是否指定类的先验概率; 若指定则不能根据参数调整
binarize: 二值化的阈值, 若为 None, 则假设输入由二进制向量组成
"""

```

样例:

```

1 from sklearn import metrics
2 from sklearn.naive_bayes import GaussianNB
3 model = GaussianNB()
4 model.fit(X, y)
5 print('MODEL')
6 print(model)
7 # make predictions
8 expected = y
9 predicted = model.predict(X)
10 # summarize the fit of the model
11 print('RESULT')
12 print(metrics.classification_report(expected, predicted))
13 print('CONFUSION MATRIX')
14 print(metrics.confusion_matrix(expected, predicted))

```

结果:

MODEL

GaussianNB()

RESULT

	precision	recall	f1-score	support
0.0	0.80	1.00	0.89	4
1.0	1.00	0.83	0.91	6
avg / total	0.92	0.90	0.90	10

CONFUSION MATRIX

```
[[4 0]
```

```
[1 5]]
```

朴素贝叶斯 ([官方文档](#))

D) 决策树 DT

定义及说明：分类与回归树(Classification and Regression Trees ,CART)算法常用于特征含有类别信息的分类或者回归问题，这种方法非常适用于多分类情况。

```
from sklearn import tree
model = tree.DecisionTreeClassifier(criterion=' gini' ,
max_depth=None,
    min_samples_split=2, min_samples_leaf=1,
min_weight_fraction_leaf=0.0,
    max_features=None, random_state=None, max_leaf_nodes=None,
    min_impurity_decrease=0.0, min_impurity_split=None,
    class_weight=None, presort=False)
"""参数
---
criterion : 特征选择准则 gini/entropy
max_depth: 树的最大深度, None-尽量下分
min_samples_split: 分裂内部节点, 所需要的最小样本树
min_samples_leaf: 叶子节点所需要的最小样本数
max_features: 寻找最优分割点时的最大特征数
max_leaf_nodes: 优先增长到最大叶子节点数
min_impurity_decrease: 如果这种分离导致杂质的减少大于或等于这个
值, 则节点将被拆分。
"""
```

样例:

```
from sklearn import metrics
from sklearn.tree import DecisionTreeClassifier
# fit a CART model to the data
model = DecisionTreeClassifier()
model.fit(X, y)
print(model)
# make predictions
expected = y
predicted = model.predict(X)
# summarize the fit of the model
print(metrics.classification_report(expected, predicted))
print(metrics.confusion_matrix(expected, predicted))
```

样例:

```
DecisionTreeClassifier(class_weight=None, criterion='gini',
max_depth=None,
                        max_features=None, max_leaf_nodes=None,
min_samples_leaf=1,
                        min_samples_split=2, min_weight_fraction_leaf=0.0,
                        random_state=None, splitter='best')
```

	precision	recall	f1-score	support
0.0	1.00	1.00	1.00	4
1.0	1.00	1.00	1.00	6
avg / total	1.00	1.00	1.00	10

```
[[4 0]
 [0 6]]
```

决策树 ([官方文档](#))

E) 支持向量机 SVM

定义及说明: SVM 是非常流行的机器学习算法, 主要用于分类问题, 如同逻辑回归问题, 它可以使用一对多的方法进行多类别的分类。

```
from sklearn.svm import SVC
model = SVC(C=1.0, kernel='rbf', gamma='auto')
"""参数
---
C: 误差项的惩罚参数 C
gamma: 核相关系数。浮点数, If gamma is 'auto' then 1/n_features
will be used instead.
"""
```

样例:

```
1 from sklearn import metrics
2 from sklearn.svm import SVC
3 # fit a SVM model to the data
4 model = SVC()
5 model.fit(X, y)
6 print(model)
7 # make predictions
8 expected = y
9 predicted = model.predict(X)
10 # summarize the fit of the model
```



```
11 print(metrics.classification_report(expected, predicted))
12 print(metrics.confusion_matrix(expected, predicted))
```

结果

```
SVC(C=1.0, cache_size=200, class_weight=None, coef0=0.0, degree=3,
    gamma=0.0,
    kernel='rbf', max_iter=-1, probability=False, random_state=None,
    shrinking=True, tol=0.001, verbose=False)
```

	precision	recall	f1-score	support
0.0	1.00	1.00	1.00	4
1.0	1.00	1.00	1.00	6
avg / total	1.00	1.00	1.00	10

```
[[4 0]
 [0 6]]
```

支持向量机 ([官方文档](#))

F) k 近邻算法 KNN

定义及说明：k 近邻算法常常被用作是分类算法一部分，比如可以用它来评估特征，在特征选择上我们可以用到它。

```
from sklearn import neighbors
#定义 kNN 分类模型
model = neighbors.KNeighborsClassifier(n_neighbors=5, n_jobs=1) # 分类
model = neighbors.KNeighborsRegressor(n_neighbors=5, n_jobs=1) # 回归
"""参数
----
    n_neighbors: 使用邻居的数目
    n_jobs: 并行任务数
"""
```

样例：

```
1 from sklearn import metrics
2 from sklearn.neighbors import KNeighborsClassifier
3 # fit a k-nearest neighbor model to the data
4 model = KNeighborsClassifier()
5 model.fit(X, y)
```

```

6 print(model)
7 # make predictions
8 expected = y
9 predicted = model.predict(X)
10 # summarize the fit of the model
11 print(metrics.classification_report(expected, predicted))
12 print(metrics.confusion_matrix(expected, predicted))

```

结果:

```

KNeighborsClassifier(algorithm='auto', leaf_size=30,
metric='minkowski',
                    metric_params=None, n_neighbors=5, p=2, weights='uniform')

```

	precision	recall	f1-score	support
0.0	0.75	0.75	0.75	4
1.0	0.83	0.83	0.83	6
avg / total	0.80	0.80	0.80	10

```

[[3 1]
 [1 5]]

```

k 近邻 ([官方文档](#))

G)K 均值聚类

来看看主函数 KMeans:

```

sklearn.cluster.KMeans(n_clusters=8,
                        init='k-means++',
                        n_init=10,
                        max_iter=300,
                        tol=0.0001,
                        precompute_distances='auto',
                        verbose=0,
                        random_state=None,
                        copy_x=True,
                        n_jobs=1,
                        algorithm='auto'
                        )

```

参数的意义:

- `n_clusters`: 簇的个数, 即你想聚成几类
- `init`: 初始簇中心的获取方法
- `n_init`: 获取初始簇中心的更迭次数, 为了弥补初始质心的影响, 算法默认会初始 10 次质心, 实现算法, 然后返回最好的结果。

- `max_iter`: 最大迭代次数（因为 `kmeans` 算法的实现需要迭代）
- `tol`: 容忍度，即 `kmeans` 运行准则收敛的条件
- `precompute_distances`: 是否需要提前计算距离，这个参数会在空间和时间之间做权衡，如果是 `True` 会把整个距离矩阵都放到内存中，`auto` 会默认在数据样本大于 `features*samples` 的数量大于 `12e6` 的时候 `False`, `False` 时核心实现的方法是利用 `Cpython` 来实现的
- `verbose`: 冗长模式（不太懂是啥意思，反正一般不去改默认值）
- `random_state`: 随机生成簇中心的状态条件。
- `copy_x`: 对是否修改数据的一个标记，如果 `True`，即复制了就不会修改数据。`bool` 在 `scikit-learn` 很多接口中都会有这个参数的，就是是否对输入数据继续 `copy` 操作，以便不修改用户的输入数据。这个要理解 `Python` 的内存机制才会比较清楚。
- `n_jobs`: 并行设置
- `algorithm`: `kmeans` 的实现算法，有：'`auto`'，'`full`'，'`elkan`'，其中 '`full`' 表示用 EM 方式实现

虽然有很多参数，但是都已经给出了默认值。所以我们一般不需要去传入这些参数,参数的。可以根据实际需要来调用。

样例：

参考博客：[python 之 sklearn 学习笔记](#)

本案例说明了，`KMeans` 分析的一些类如何调取与什么意义。

```
import numpy as np
from sklearn.cluster import KMeans
data = np.random.rand(100, 3) #生成一个随机数据，样本大小为 100，特征数为 3
#假如我要构造一个聚类数为 3 的聚类器
estimator = KMeans(n_clusters=3)#构造聚类器
estimator.fit(data)#聚类
label_pred = estimator.labels_ #获取聚类标签
centroids = estimator.cluster_centers_ #获取聚类中心
inertia = estimator.inertia_ # 获取聚类准则的总和
estimator 初始化 Kmeans 聚类；estimator.fit 聚类内容拟合；
estimator.label_ 聚类标签，这是一种方式，还有一种是 predict；
estimator.cluster_centers_ 聚类中心均值向量矩阵
estimator.inertia_ 代表聚类中心均值向量的总和
```

H)多层感知机（神经网络）

定义及说明：

```
from sklearn.neural_network import MLPClassifier
# 定义多层感知机分类算法
model = MLPClassifier(activation='relu', solver='adam', alpha=0.0001)
"""参数
---
hidden_layer_sizes: 元祖
activation: 激活函数
solver : 优化算法{ 'lbfgs' , 'sgd' , 'adam' }
```

```
alpha: L2 惩罚(正则化项) 参数。
"""
```

5) 模型评估与选择篇

A) 交叉验证

```
from sklearn.model_selection import cross_val_score
cross_val_score(model, X, y=None, scoring=None, cv=None, n_jobs=1)
""" 参数
---
    model: 拟合数据的模型
    cv : k-fold
    scoring: 打分参数- 'accuracy'、'f1'、'precision'、
    'recall'、'roc_auc'、'neg_log_loss' 等等
"""
```

B) 检验曲线

使用检验曲线，我们可以更加方便的改变模型参数，获取模型表现。

```
from sklearn.model_selection import validation_curve
train_score, test_score = validation_curve(model, X, y, param_name,
param_range, cv=None, scoring=None, n_jobs=1)
""" 参数
---
    model: 用于 fit 和 predict 的对象
    X, y: 训练集的特征和标签
    param_name: 将被改变的参数的名字
    param_range: 参数的改变范围
    cv: k-fold

返回值
---
    train_score: 训练集得分 (array)
    test_score: 验证集得分 (array)
"""
```

C) 如何优化算法参数

一项更加困难的任务是构建一个有效的方法用于选择正确的参数，我们需要用搜索的方法来确定参数。`scikit-learn` 提供了实现这一目标的函数。

下面的例子是一个进行正则参数选择的程序：

[GridSearchCV 官方文档 1](#)（模块使用） [官方文档 2](#)（原理详解）

样例：

```
1 import numpy as np
2 from sklearn.linear_model import Ridge
3 from sklearn.grid_search import GridSearchCV
4 # prepare a range of alpha values to test
5 alphas = np.array([1, 0.1, 0.01, 0.001, 0.0001, 0])
6 # create and fit a ridge regression model, testing each alpha
7 model = Ridge()
8 grid = GridSearchCV(estimator=model,
param_grid=dict(alpha=alphas))
9 grid.fit(X, y)
10 print(grid)
11 # summarize the results of the grid search
12 print(grid.best_score_)
13 print(grid.best_estimator_.alpha)
```

结果：

```
GridSearchCV(cv=None, error_score='raise',
             estimator=Ridge(alpha=1.0, copy_X=True, fit_intercept=True,
max_iter=None,
             normalize=False, solver='auto', tol=0.001),
             fit_params={}, iid=True, loss_func=None, n_jobs=1,
             param_grid={'alpha': array([ 1.000000e+00,  1.000000e-01,
1.000000e-02,  1.000000e-03,
             1.000000e-04,  0.000000e+00])},
             pre_dispatch='2*n_jobs', refit=True, score_func=None,
             scoring=None,
             verbose=0)
-5.59572064238
0.0
```

有时随机从给定区间中选择参数是很有效的方法，然后根据这些参数来评估算法的效果进而选择最佳的那个。

[RandomizedSearchCV 官方文档](#)（模块使用）[官方文档 2](#)（原理详解）

样例：

```
1 import numpy as np
2 from scipy.stats import uniform as sp_rand
3 from sklearn.linear_model import Ridge
4 from sklearn.grid_search import RandomizedSearchCV
5 # prepare a uniform distribution to sample for the alpha parameter
6 param_grid = {'alpha': sp_rand() }
```

```

7 # create and fit a ridge regression model, testing random alpha
values
8 model = Ridge()
9 rsearch = RandomizedSearchCV(estimator=model,
param_distributions=param_grid, n_iter=100)
10 rsearch.fit(X, y)
11 print(rsearch)
12 # summarize the results of the random parameter search
13 print(rsearch.best_score_)
14 print(rsearch.best_estimator_.alpha)

```

6. 保存模型

最后，将训练好的 **model** 保存到本地，或者放到线上供用户使用。主要有两种方式保存训练好的 **model**：

A) 保存为 pickle 文件

```

from sklearn import svm
from sklearn import datasets
clf=svm.SVC()
iris=datasets.load_iris()
X,y=iris.data,iris.target
clf.fit(X,y)

#保存模型
import pickle
s=pickle.dumps(clf)
f=open('svm.model','w')
f.write(s)
f.close()

#使用模型预测
f2=open('svm.model','r')
s2=f2.read()
clf2=pickle.loads(s2)
clf2.predict(X,y)

```

B) sklearn 自带方法 joblib

```

from sklearn.externals import joblib
from sklearn import svm

#训练模型
clf = svm.SVC(kernel='linear')
rf=clf.fit(array(trainMat), array(listClasses))

```



```
# 保存模型
joblib.dump(rf, 'rf.model')

# 加载模型
RF=joblib.load('rf.model')

# 应用模型进行预测
result=RF.predict(thsDoc)# 保存模型
```

除了分类和回归算法外，**scikit-learn** 提供了更加复杂的算法，比如聚类算法，还实现了算法组合的技术，如 **Bagging** 和 **Boosting** 算法。

2.7 本章小结

与传统的计算机程序不同，机器学习是面向数据的算法，能够从数据中获得信息。它符合新时代脑力劳动代替体力劳动的趋势，是富有生命力的领域。

Python 是一门优异的语言，代码清晰可读、功能广泛强大。其最大弱点—速度问题也可以通过很多不太困难的方法弥补。

Anaconda 是 **Python** 的一个很好的集成环境，它能让我们免于人工地安装大量科学计算所需要的第三方库。

虽说机器学习算法很多，但通常而言，进行机器学习的过程会包含以下三步：

- 获取与处理数据；
- 选择与训练模型；
- 评估与可视化结果。

3 基本算法--分类

分类是数据挖掘、智能处理中一项非常重要的任务，利用分类技术可以从数据集中提取描述数据类的一个函数或模型（也常称为分类器），并把数据集中的每个对象归结到某个已知的对象类中。

从机器学习的观点，分类技术是一种有指导的学习，即每个训练样本的数据对象已经有类标识，通过学习可以形成表达数据对象与类标识间对应的知识。

分类技术预测的数据对象是离散值。例如，电子邮件是否为垃圾邮件，肿瘤是癌性还是良性等等。分类模型将输入数据分类。典型应用包括医学成像，信用评分等。

3.1 KNN 分类算法

1) 算法原理

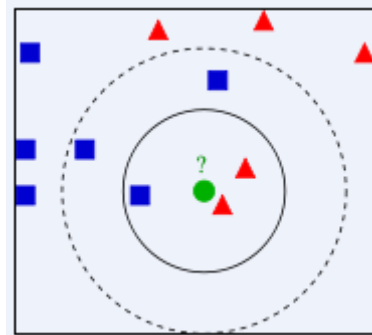
KNN 分类算法（**K-Nearest-Neighbors Classification**），又叫 **K** 近邻算法。其核心思想是如

如果一个样本在特征空间中的 k 个最相邻的样本中的大多数属于某一个类别，则该样本也属于这个类别，并具有这个类别上样本的特性。（给定一个待分类数据，通过距离计算得到离它最近的 k 个样本，由 k 个样本投票决定待分类数据归为哪一类——“少数服从多数”）。简单的说就是让最相似的 k 个样本来投票决定。

算法的三个决定因素是：（1）距离度量（2） k 值（3）分类决策规则。

这里所说的距离，一般最常用的就是多维空间的欧式距离。这里的维度指特征维度，即样本有几个特征就属于几维。KNN 示意图如下所示。

图中要确定测试样本绿色属于蓝色还是红色。显然，当 $K=3$ 时，将以 1: 2 的投票结果分类于红色；而 $K=5$ 时，将以 3: 2 的投票结果分类于蓝色。



处理过程：

1. 计算样本数据和待分类数据的距离。
2. 选择 k 个与待分类数据距离最近的样本。
3. 统计出 k 个样本中大多数样本所属的分类，将该类记作 **category**。
4. 待分类数据属于 **category**。

算法原理简单易实现，适合对稀有事件进行分类，对多分类问题（multi-modal，对象有多个类别标签）来说， k -nn 比 svm 更合适。

但当样本不平衡时，如一个类的样本容量很大，而其他类样本容量很小时，有可能导致当输入一个新样本时，该样本的 K 个邻居中大容量类的样本占多数。该算法只计算“最近的”邻居样本，某一类的样本数量很大，那么或者这类样本并不接近目标样本，或者这类样本很靠近目标样本。无论怎样，数量并不能影响运行结果；同时计算量较大，因为对每一个待分类的文本都要计算它到全体已知样本的距离，才能求得它的 K 个最近邻点。

2) 示例代码 1

```
#鸢尾花数据集为例讲述 k-nn 算法
from sklearn import datasets, neighbors

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

kn_clf = neighbors.KNeighborsClassifier()
kn_clf.fit(X, y)
kn_y = kn_clf.predict(X)
print(kn_y)
print(y)
```

```
accuracy_knn = (kn_y == y).astype(int).mean()
print(accuracy_knn)
```

3) 示例代码 2

```
# -*- coding: utf-8 -*-
import numpy as np
from sklearn import neighbors
from sklearn.metrics import precision_recall_curve
from sklearn.metrics import classification_report
from sklearn.cross_validation import train_test_split
import matplotlib.pyplot as plt
#体形判断
""" 数据读入 """
data      = []
labels    = []
with open("person.txt") as ifile:
    for line in ifile:
        tokens = line.strip().split(' ')
        data.append([float(tk) for tk in tokens[:-1]])
        labels.append(tokens[-1])

x = np.array(data)
labels = np.array(labels)
y = np.zeros(labels.shape)

""" 标签转换为 0/1 """
y[labels=='fat']=1

""" 拆分训练数据与测试数据 """
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size = 0.2)

""" 创建网格以方便绘制 """ h = .01
x_min, x_max = x[:, 0].min() - 0.1, x[:, 0].max() + 0.1
y_min, y_max = x[:, 1].min() - 1, x[:, 1].max() + 1
xx, yy = np.meshgrid(np.arange(x_min, x_max, h),
                     np.arange(y_min, y_max, h))

""" 训练 KNN 分类器 """
clf = neighbors.KNeighborsClassifier(algorithm='kd_tree')
clf.fit(x_train, y_train)

"""测试结果的打印"""
answer = clf.predict(x)
print(x)
```

```

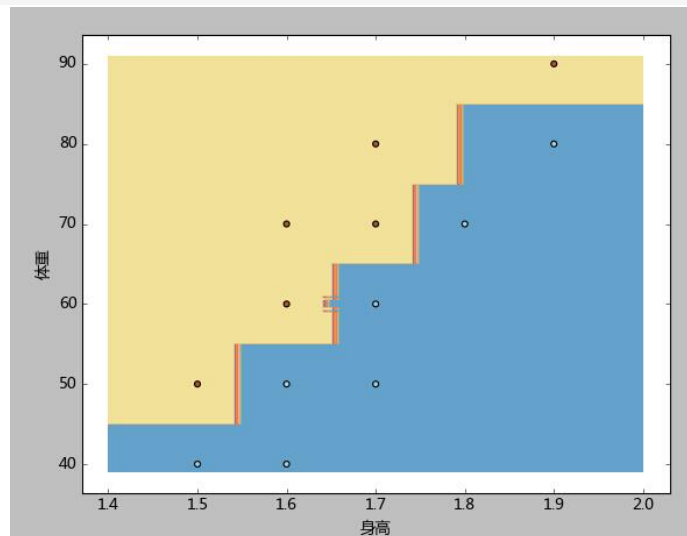
print(answer)
print(y)
print(np.mean( answer == y))

"""准确率与召回率"""
precision, recall, thresholds = precision_recall_curve(y_train, clf.predict(x_train))
answer = clf.predict_proba(x)[:1]
print(classification_report(y, answer, target_names = ['thin', 'fat']))

""" 将整个测试空间的分类结果用不同颜色区分开"""
answer = clf.predict_proba(np.c_[xx.ravel(), yy.ravel()])[:1]
z = answer.reshape(xx.shape)
plt.contourf(xx, yy, z, cmap=plt.cm.Paired, alpha=0.8)

""" 绘制训练样本 """
plt.scatter(x_train[:, 0], x_train[:, 1], c=y_train, cmap=plt.cm.Paired)
plt.xlabel(u'身高')
plt.ylabel(u'体重')
plt.show()

```



3.2 决策树

决策树算法有很多种，有 ID3、C4.5 和 CART 等算法。

1) ID3 算法原理

输入：训练数据集 D ，特征集 A ，阈值；

输出：决策树 T 。

算法流程：

1 若 D 中所有实例属于同一类 C ，则 T 为单节点树，并将 C 作为该结点的类标记，返回 T ；

2 若 $A=\emptyset$ ，则 T 为单节点树，并将 D 中实例数最多的类 C 作为该结点的类标记，返回 T ；

3 否则，计算 A 中各特征对 D 的信息增益，选择信息增益最大的特征 A_1 ；

4 对 A_1 的每一个取值，将 D 相应地分割成若干非空子集

5 以每一个子集重新作为根节点，重复上述过程，并依次递归进行

这样最终将会形成一个决策树。

C4.5 算法与 ID3 算法过程一样，不同在于 ID3 算法是用信息增益来选择特征，而 C4.5 算法使用信息增益率来选择特征。在使用信息增益作为训练数据集特征时会偏向于取值较多的特征，而用信息增益率则避免了这一问题。

信息熵定义： $Ent(D) = -\sum_{k=1}^N p_k \log_2 p_k$

信息熵表示了样本纯度，信息熵越小，样本纯度越高。其中， N 代表了样本类的个数。

基于信息熵，我们可以对某个属性 a 定义“信息增益”：

$$Gain(D, a) = Ent(D) - \sum_{v=1}^V \frac{|D^v|}{|D|} Ent(D^v)$$

其中， a 属性有 V 个可能取值，而 D 中在属性 a 上取值 v 的样本记为 D^v 。

基于信息增益定义信息增益率：

$$Gain_ratio(D, a) = \frac{Gain(D, a)}{IV(a)}$$

$$IV(a) = -\sum_{v=1}^V \frac{|D^v|}{|D|} \log_2 \frac{|D^v|}{|D|}$$

2) 示例程序 1

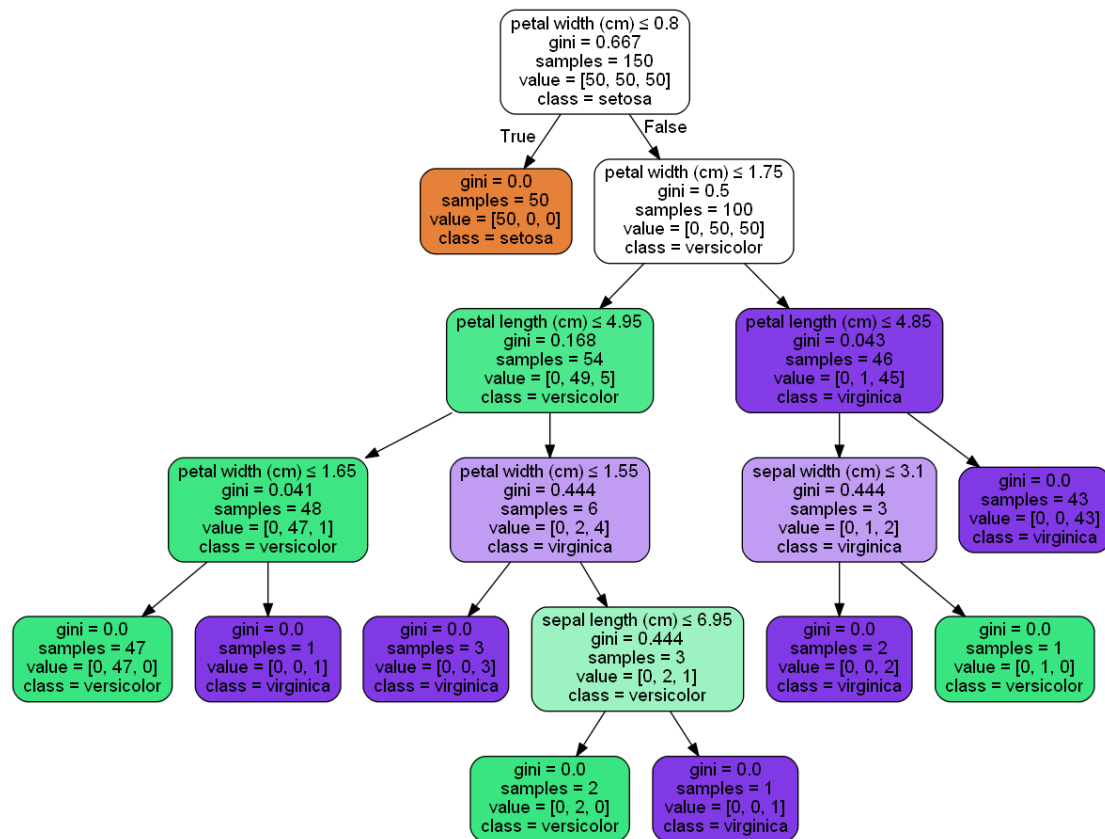
```
from sklearn.datasets import load_iris
from sklearn import tree
from sklearn.externals.six import StringIO
#import pydot
import pydotplus
from PIL import Image

iris = load_iris()
clf = tree.DecisionTreeClassifier()
clf = clf.fit(iris.data, iris.target)

with open("iris.dot", 'w') as f:
    f = tree.export_graphviz(clf, out_file=f)

dot_data = StringIO()
tree.export_graphviz(clf, out_file=dot_data,
                     feature_names=iris.feature_names,
                     class_names=iris.target_names,
                     filled=True, rounded=True,
                     special_characters=True)
```

```
graph = pydotplus.graph_from_dot_data(dot_data.getvalue())
graph.write_pdf("iris.pdf")
graph.write_png("iris.png")
img = Image.open('iris.png')
img.show()
print("处理完成")
决策树图片：
```



3.3 贝叶斯分类

贝叶斯分类算法有很多种，常用的有：朴素贝叶斯算法说明、高斯朴素贝叶斯（Gaussian Naive Bayes）、多项式朴素贝叶斯（Multinomial Naive Bayes）、伯努利朴素贝叶斯（Bernoulli Naive Bayes）等。

1) 朴素贝叶斯算法说明

朴素贝叶斯算法是建立在每一个特征值之间时独立的基础上的监督学习分类算法，而这也是称他为“朴素”贝叶斯的缘由，在现实环境中，很难达到两个特征值之间绝对的相互独立。在给定一个类变量 Y 和依赖的特征向量 x_1 通过 x_n ，贝叶斯定理的状态下面的关系：

$$P(y | x_1, \dots, x_n) = \frac{P(y)P(x_1, \dots, x_n | y)}{P(x_1, \dots, x_n)}$$

假设两个特征值之间时相互独立的，即：

$$P(x_i|y, x_1, \dots, x_{i-1}, x_{i+1}, \dots, x_n) = P(x_i|y),$$

对于所有的 i 值，这种关系被简化为：

$$P(y | x_1, \dots, x_n) = \frac{P(y) \prod_{i=1}^n P(x_i | y)}{P(x_1, \dots, x_n)}$$

由于 $P(x_1, \dots, x_n)$ 是恒定的给定的输入，我们可以使用以下的分类规则(前者正比于后者)：

$$P(y | x_1, \dots, x_n) \propto P(y) \prod_{i=1}^n P(x_i | y)$$

\Downarrow

$$\hat{y} = \arg \max_y P(y) \prod_{i=1}^n P(x_i | y),$$

我们可以使用最大后验概率（MAP）来估计 $p(y)$ 和 $p(x_i|y)$ ， $p(y)$ 是在训练集中 y 发生的概率不同朴素贝叶斯分类算法是因为他们对 $P(X_i|y)$ 做出了不同的假设尽管朴素贝叶斯的假设过于简单，但在已有的应用中，如文档分类和垃圾邮件分类，他都表现出了相当好的效果（至于理论上的原因，为什么朴素贝叶斯的效果很好，并且他适合处理的数据类型，请继续往下阅读）和其他更先进的方法相比，朴素贝叶斯算法学习和分类的过程效率更高，每个类条件特征的独立分布意味着每个类分布可以独立的估计为一维分布，这反过来有助于缓解数据降维所带来的麻烦另一方面，尽管朴素贝叶斯分类被归纳为一个高效的分类器，但是他有一个坏的估计，因为对于他的输出预测并不能被认真对待上边提到了不同的贝叶斯分类算法是因为他们对 $P(X_i|y)$ 做出了不同的假设，下面我们来看集中常见的 $P(X_i|y)$ 的假设以及 scikit-learn 中的实现方法

示例：

```
import numpy as np
from sklearn.datasets import load_iris
from sklearn import naive_bayes

iris = load_iris()
x_train = iris.data
y_train = iris.target
clf = naive_bayes.GaussianNB()
clf.fit(x_train, y_train)
print(clf.score(x_train, y_train))

clf1 = naive_bayes.MultinomialNB()
clf1.fit(x_train, y_train)
print(clf1.score(x_train, y_train))

print(x_train)
```

```
print(y_train)
```

2) 高斯朴素贝叶斯

GaussianNB 继承高斯朴素贝叶斯，特征可能性被假设为高斯：

$$P(x_i | y) = \frac{1}{\sqrt{2\pi\sigma_y^2}} \exp\left(-\frac{(x_i - \mu_y)^2}{2\sigma_y^2}\right)$$

代码示例如下：

```
[python] view plain copy
1. #高斯朴素贝叶斯
2. import numpy as np
3. X = np.array([[ -1, -1], [ -2, -1], [ -3, -2], [ 1, 1], [ 2, 1], [ 3, 2]])
4. Y = np.array([1, 1, 1, 2, 2, 2])
5. from sklearn.naive_bayes import GaussianNB
6. clf = GaussianNB().fit(X, Y)
7. print clf.predict([[ -0.8, -1]])
8.
9. ....
10. partial_fit 说明：增量的训练一批样本
11. 这种方法被称为连续几次在不同的数据集，从而实现核心和在线学习，这是特别有用的，当数据集很大的时候，不适合在内存中运算
12. 该方法具有一定的性能和数值稳定性的开销，因此最好是作用在尽可能大的数据块（只要符合内存的预算开销）
13. ...
14. clf_pf = GaussianNB().partial_fit(X, Y, np.unique(Y))
15. print clf_pf.predict([[ -0.8, -1]])
```

输出结果为：

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

更多关于高斯贝叶斯分布的请参考：[点击阅读](#)

3) 多项式分布

MultinomialNB 实现 multinomially 分布数据的贝叶斯算法，是一个经典的朴素贝叶斯文本分类中使用的变种（其中的数据是通常表示为词向量的数量，虽然 TF-IDF 向量在实际项目中表现得很好），对于每一个 y 来说，分布通过向量 $\theta_y = (\theta_{y1}, \dots, \theta_{yn})$ 参数化， n 是类别的数目（在文本分类中，表示词汇量的长度） θ_{yi} 表示标签 i 出现的样本属于类别 y 的概

率 $P(x_i | y)$ 该参数 θ_{yi} 是一个平滑的最大似然估计，即相对频率计数：

$$\hat{\theta}_{yi} = \frac{N_{yi} + \alpha}{N_y + \alpha n}$$

$N_{yi} = \sum_{x \in T} x_i$ 表示标签 i 在样本集 T 中属于类别 y 的数目

$N_y = \sum_{i=1}^{|T|} N_{yi}$ 表示在所有标签中类别 y 出现的数目

平滑先验 $\alpha \geq 0$ 表示学习样本中不存在的特征并防止在计算中概率为 0，设置 $\alpha = 1$ 被称为拉普拉斯平滑，当 $\alpha < 1$ 称为 Lidstone 平滑

代码示例如下：

[python] [view plain copy](#)

```
1. #多项式分布
2. import numpy as np
3. X = np.random.randint(5, size=(6, 100))
4. y = np.array([1, 2, 3, 4, 5, 6])
5. from sklearn.naive_bayes import MultinomialNB
6. clf = MultinomialNB().fit(X, y)
7. print clf.predict(X[2:3])
```

输出为 [3]

更多关于多项式分布请参考：[点击阅读](#)

4) 伯努利朴素贝叶斯

BernoulliNB 实现了朴素贝叶斯训练和分类算法是根据多元伯努利分布的分布数据；例如，可能会有多个特征，但每一个被假定为一个二进制值（伯努利、布尔）变量。因此，这类要求的样品被表示为二进制值的特征向量；如果交给其他任何类型的数据，一个 **bernoullinb** 实例可以进行输入（取决于二值化参数）伯努利朴素贝叶斯决策规则的基础上

$P(x_i | y) = P(i | y)x_i + (1 - P(i | y))(1 - x_i)$ 在文本分类的情况下，词的出现

向量（而不是字计数向量）可以用来训练和使用该分类。**bernoullinb** 可能会执行一些数据集上的更好，尤其是那些短的文件。如果时间允许的话，建议对两种模型进行评估。

示例代码如下：

[python] [view plain copy](#)

```
1. #伯努利分布
2. import numpy as np
```

```

3. X = np.random.randint(2, size=(6, 100))
4. Y = np.array([1, 2, 3, 4, 4, 5])
5. from sklearn.naive_bayes import BernoulliNB
6. clf = BernoulliNB()
7. clf.fit(X, Y)
8. BernoulliNB(alpha=1.0, binarize=0.0, class_prior=None, fit_prior=True)
9. print(clf.predict(X[2:3]))

```

输出结果为 [3

5) 综合示例

```

import numpy as np
from sklearn import cross_validation
from sklearn import datasets
from sklearn import naive_bayes
from sklearn.model_selection import KFold

iris = datasets.load_iris()
data = iris.data
target = iris.target
dataDimenz = data.shape
targetDimens = target.shape
print(dataDimenz,targetDimens)

kf = KFold(n_splits=5)
for train_idx,test_idx in kf.split(data,target):
    print("Train Data:",train_idx)
    print("Train Target",test_idx)
    x_train,x_test = data[train_idx],data[test_idx]
    y_train,y_test = target[train_idx],target[test_idx]

x_train,x_test,y_train,y_test = cross_validation.train_test_split(data,target,test_size=0.3)

xtDimens = x_train.shape
ytDimens = y_train.shape

clfBayes = naive_bayes.GaussianNB().fit(x_train,y_train)
clsRes = clfBayes.predict(x_test)
print(clsRes)

```

6) 思考

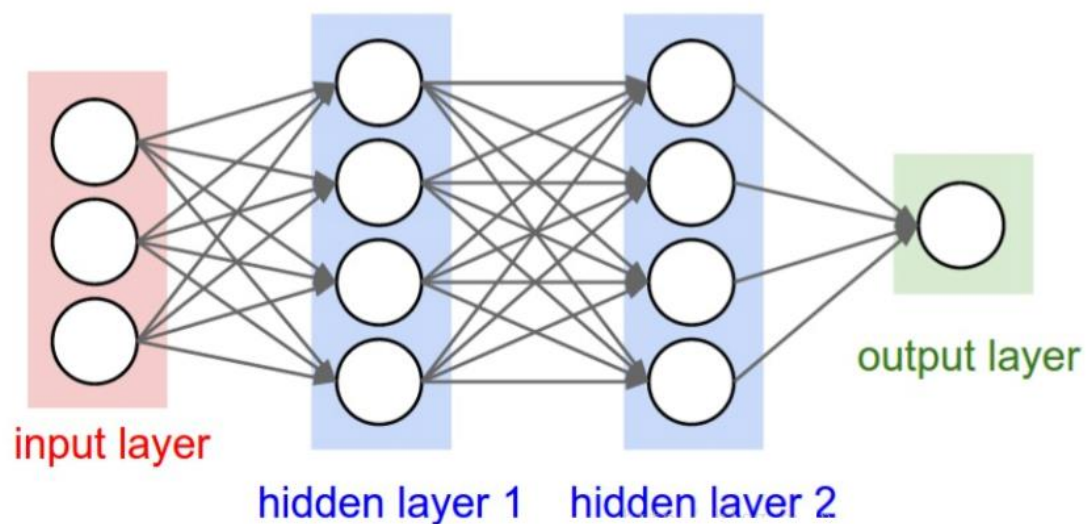
贝叶斯分类器在文本分类、垃圾邮件过滤中如何应用？

3.4 神经网络分类

1) 算法原理

人工神经网络（Artificial Neural Network，ANN），也称神经网络或类神经网络，是一种通过模仿人脑神经元做决策的一种算法。它从信息处理角度对人脑神经网络进行抽象，建立某种简单模型，按不同的连接方式组成不同的网络。

神经网络是一种运算模型，由大量的节点（或称神经元）之间相互联接构成。每个节点代表一种特定的输出函数，称为激励函数（activation function）。每两个节点间的连接都代表一个对于通过该连接信号的加权值，称之为权重，这相当于人工神经网络的记忆。网络的输出则依网络的连接方式，权重值和激励函数的不同而不同。而网络自身通常都是对自然界某种算法或者函数的逼近，也可能是对一种逻辑策略的表达。



说明：

通常一个神经网络由一个 input layer，多个 hidden layer 和一个 output layer 构成。

图中圆圈可以视为一个神经元（又可以称为感知器）

设计神经网络的重要工作是设计 hidden layer，及神经元之间的权重

添加少量隐层获得浅层神经网络 SNN；隐层很多时就是深层神经网络 DNN

2) 示例代码 1

#鸢尾花数据集为例讲述 k-nn 算法

3.5 支持向量机

1) 算法原理

支持向量机（Support Vector Machine, SVM）的基本模型是在特征空间上找到最佳的分

离超平面使得训练集上正负样本间隔最大。**SVM** 是用来解决二分类问题的有监督学习算法，在引入了核方法之后 **SVM** 也可以用来解决非线性问题。

一般 **SVM** 有下面三种：

硬间隔支持向量机（线性可分支持向量机）：当训练数据线性可分时，可通过硬间隔最大化学得一个线性可分支持向量机。

软间隔支持向量机：当训练数据近似线性可分时，可通过软间隔最大化学得一个线性支持向量机。

非线性支持向量机：当训练数据线性不可分时，可通过核方法以及软间隔最大化学得一个非线性支持向量机。

2) 算法示例 1

#鸢尾花数据集示例 svm 算法

#导入 svm 和数据集

```
from sklearn import svm, datasets
```

#调用 SVC()

```
clf = svm.SVC()
```

#载入鸢尾花数据集

```
iris = datasets.load_iris()
```

```
X = iris.data
```

```
y = iris.target
```

#fit()训练

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

#predict()预测

```
pre_y = clf.predict(X[5:10])
```

```
print(pre_y)
```

```
print(y[5:10])
```

#导入 numpy

```
import numpy as np
```

```
test = np.array([[5.1,2.9,1.8,3.6]])
```

#对 test 进行预测

```
test_y = clf.predict(test)
```

```
print(test_y)
```

3) 算法示例 2

```
from sklearn.datasets.samples_generator import make_circles
```

```
X, y = make_circles(100, factor=.1, noise=.1)
```

```
clf = SVC(kernel='linear').fit(X, y)
```

```
plt.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='spring')
```

```
plot_svc_decision_function(clf);
```

```
r = np.exp(-(X[:, 0] ** 2 + X[:, 1] ** 2))
```

```

from mpl_toolkits import mplot3d

def plot_3D(elev=30, azim=30):
    ax = plt.subplot(projection='3d')
    ax.scatter3D(X[:, 0], X[:, 1], r, c=y, s=50, cmap='spring')
    ax.view_init(elev=elev, azim=azim)
    ax.set_xlabel('x')
    ax.set_ylabel('y')
    ax.set_zlabel('r')

interact(plot_3D, elev=[-90, 90], azim=(-180, 180));
clf = SVC(kernel='rbf')
clf.fit(X, y)

plt.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='spring')
plot_svc_decision_function(clf)
plt.scatter(clf.support_vectors_[0], clf.support_vectors_[1],
            s=200, facecolors='none');

```

4 基本算法--聚类

4.1 K 均值聚类

1) 算法原理

K-means 算法是集简单和经典于一身的基于距离的聚类算法

采用距离作为相似性的评价指标，即认为两个对象的距离越近，其相似度就越大。

该算法认为类簇是由距离靠近的对象组成的，因此把得到紧凑且独立的簇作为最终目标。

其伪代码如下：

创建 k 个点作为初始的质心点（随机选择）

当任意一个点的簇分配结果发生改变时

 对数据集中的每一个数据点

 对每一个质心

 计算质心与数据点的距离

 将数据点分配到距离最近的簇

 对每一个簇，计算簇中所有点的均值，并将均值作为质心

人以类聚，物以群分，k-means 聚类算法就是体现。数学公式不要，直接用白话描述的步骤就是：

- 1.随机选取 k 个质心（k 值取决于你想聚成几类）
- 2.计算样本到质心的距离，距离质心距离近的归为一类，分为 k 类
- 3.求出分类后的每类的新质心
- 4.判断新旧质心是否相同，如果相同就代表已经聚类成功，如果没有就循环 2-3 直到相同

用程序的语言描述就是：

- 1.输入样本
- 2.随机去 k 个质心
- 3.重复下面过程直到算法收敛：
 - 计算样本到质心距离（欧几里得距离）
 - 样本距离哪个质心近，就记为那一类
 - 计算每个类别的新质心（平均值）

2) 示例程序 1

#对数据文件进行分析，将其分为 4 类

```
from sklearn.cluster import KMeans
import matplotlib.pyplot as plt

#加载数据
print("开始加载数据...")
dataSet = []
fileIn = open("data-1.txt")
for line in fileIn.readlines():
    lineArr = line.strip().split(' ')
    dataSet.append([float(lineArr[0]),float(lineArr[1])])
clf = KMeans(n_clusters=4).fit(dataSet)
numSamples = len(dataSet)
centroids = clf.labels_
print(centroids,type(centroids))
print(clf.inertia_)
mark = ['or', 'ob', 'og', 'ok', '^r', '+r', 'sr', 'dr', '<r', 'pr']
for i in range(numSamples):
    plt.plot(dataSet[i][0],dataSet[i][1],mark[clf.labels_[i]])
mark = ['Dr', 'Db', 'Dg', 'Dk', '^b', '+b', 'sb', 'db', '<b', 'pb']
centroids = clf.cluster_centers_
for i in range(k):
    plt.plot(centroids[i][0],centroids[i][1],mark[i],markersize = 12 )
plt.show()
```

3) 示例程序 2

```
#鸢尾花数据集示例 k-means 算法
from sklearn import cluster
```



```

km = cluster.KMeans(n_clusters = 3)
km.fit(X)
km_y = km.predict(X)
print(km_y)
print(y)

accuracy_km = (km_y == y).astype(int).mean()
print(accuracy_km)

```

4) 示例程序 3

```

from sklearn.cluster import KMeans
import matplotlib.pyplot as plt

#加载数据
print("开始加载数据...")
dataSet = []
fileIn = open("data-1.txt")
for line in fileIn.readlines():
    lineArr = line.strip().split(' ')
    dataSet.append([float(lineArr[0]),float(lineArr[1])])
#动态设定要分类的数量
for k in range(2,10):
    clf = KMeans(n_clusters=k).fit(dataSet)
    numSamples = len(dataSet)
    centroids = clf.labels_
    print(centroids,type(centroids))
    print(clf.inertia_)
    mark = ['or', 'ob', 'og', 'ok', '^r', '+r', 'sr', 'dr', '<r', 'pr']
    for i in range(numSamples):
        plt.plot(dataSet[i][0],dataSet[i][1],mark[clf.labels_[i]])
    mark = ['Dr', 'Db', 'Dg', 'Dk', '^b', '+b', 'sb', 'db', '<b', 'pb']
    centroids = clf.cluster_centers_
    for i in range(k):
        plt.plot(centroids[i][0],centroids[i][1],mark[i],markersize = 12 )
    plt.show()

```

5) 思考

- 1 如何确定分类数目？
- 2 如何判断分类所属样本的有效性？

4.2 模糊 C 均值聚类

1) 算法原理

模糊 C 均值聚类 (Fuzzy-C-Means) 是从硬聚类算法 (K-Means) 推广而来。FCM 算法是基于对目标函数的优化基础上的一种数据聚类方法。聚类结果是每一个数据点对聚类中心的隶属程度，该隶属程度用一个数值来表示。

硬划分 FCM 算法的目标函数以聚类中心到样本点之间的失真度 (一般是距离) 表示。而软划分 FCM 的目标函数用每个样本点到中心的隶属度来表示。这个隶属度是一个 0~100% 的数值，而硬聚类则只有 0% 和 100%，FCM 通过这个隶属度可以使我们更加直观的了解一个数据点到中心的可信度。

$$J(U, C) = \sum_{j=1}^n \sum_{i=1}^k u_{ij}^m d_{ij}^2(x_j, c_i), \text{ 其中 } d_{ik}^2 = \|x_k - p_i\|_A^2 = (x_k - p_i)^T A (x_k - p_i)$$

求解过程示意图如下：



具体确定聚类中心 c_i 和隶属矩阵 U 过程如下：

步骤 1：用值在 0, 1 间的随机数初始化隶属矩阵 U ，使 $\sum_{i=1}^c u_{ij} = 1, \forall j = 1, \dots, n$ ；

步骤 2：用式 $c_i = \frac{\sum_{j=1}^n u_{ij}^m x_j}{\sum_{j=1}^n u_{ij}^m}$ 计算 c 个聚类中心 c_i ， $i=1, \dots, c$ 。

步骤 3：根据式 $J(U, c_1, \dots, c_c) = \sum_{i=1}^c J_i = \sum_{i=1}^c \sum_{j=1}^n u_{ij}^m d_{ij}^2$ 计算价值函数。如果它小于某个

确定的阈值，或它相对上次价值函数值的改变量小于某个阈值，则算法停止。

步骤 4: 用 $u_{ij} = \frac{1}{\sum_{k=1}^c \left(\frac{d_{ij}}{d_{kj}} \right)^{2/(m-1)}}$ 计算新的 U 矩阵。返回步骤 2。

FCM 算法是一种无监督的模糊聚类方法，在算法实现过程中不需要人为的干预。这种算法的不足之处:首先，算法中需要设定初始参数，若参数的初始化选取的不合适，可能影响聚类结果的正确性;其次，当数据样本集合较大并且特征数目较多时，算法的实时性不太好。

2) 示例程序

```
import numpy as np
import matplotlib.pyplot as plt
from skfuzzy.cluster import cmeans

train = train.T
center, u, u0, d, jm, p, fpc = cmeans(train, m=2, c=3, error=0.005, maxiter=1000)

for i in u:
    label = np.argmax(u, axis=0)
    print(label)

for i in range(50):
    if label[i] == 0:
        plt.scatter(train[0][i], train[1][i], c='r')
    elif label[i] == 1:
        plt.scatter(train[0][i], train[1][i], c='g')
    elif label[i] == 2:
        plt.scatter(train[0][i], train[1][i], c='b')
    elif label[i] == 3:
        plt.scatter(train[0][i], train[1][i], c='y')
    elif label[i] == 4:
        plt.scatter(train[0][i], train[1][i], c='k')

plt.show()
```

5 基本算法--回归降维

5.1 回归算法

1) 算法原理

线性回归是利用数理统计中回归分析，来确定两种或两种以上变量间相互依赖的定量关系的一种统计分析方法，运用十分广泛。其表达形式为 $y = w'x + e$ ， e 为误差服从均值为 0 的正态分布。回归分析中，只包括一个自变量和一个因变量，且二者的关系可用一条直线近似表示，这种回归分析称为一元线性回归分析。一般来说，线性回归都可以通过[最小二乘法](#)求出其方程，可以计算出对于 $y = bx + a$ 的直线。

如果回归分析中包括两个或两个以上的自变量，且因变量和自变量之间是线性关系，则称为多元线性回归分析。在单因子（连续变量）试验中，当回归函数不能用直线来描述时，要考虑用非线性回归函数。

多项式回归属于非线性回归的一种。这里指单因子多项式回归，即一元多项式回归。

一般非线性回归函数是未知的，或即使已知也未必可以用一个简单的函数变换转化为线性模型。这时，常用的做法是用因子的多项式。如果从散点图观察到回归函数有一个“弯”，则可考虑用二次多项式；有两个弯则考虑用三次多项式；有三个弯则考虑用四次多项式，等等。

真实的回归函数未必就是某个次数的多项式，但只要拟合得好，用适当的多项式来近似真实的回归函数是可行的。

逻辑回归逻辑回归就是这样的过程：面对一个回归或者分类问题，建立代价函数，然后通过优化方法迭代求解出最优的模型参数，然后测试验证我们这个求解的模型的好坏。

Logistic 回归与多重线性回归实际上有很多相同之处，最大的区别就在于它们的因变量不同，其他的基本都差不多。正是因为如此，这两种回归可以归于同一个家族，即广义线性模型（generalized linear model）。

这一家族中的模型形式基本上都差不多，不同的就是因变量不同。这一家族中的模型形式基本上都差不多，不同的就是因变量不同。

如果是连续的，就是多重线性回归

如果是二项分布，就是 Logistic 回归

如果是 Poisson 分布，就是 Poisson 回归

如果是负二项分布，就是负二项回归

2) 示例程序 1

```
from sklearn import linear_model
import numpy as np
import matplotlib.pyplot as plt

##样本数据(Xi,Yi)，需要转换成数组(列表)形式
Xi=np.array([6.19,2.51,7.29,7.01,5.7,2.66,3.98,2.5,9.1,4.2]).reshape(-1,1)
Yi=np.array([5.25,2.83,6.41,6.71,5.1,4.23,5.05,1.98,10.5,6.3]).reshape(-1,1)

##设置模型
model = linear_model.LinearRegression()

##训练数据
model.fit(Xi, Yi)

##用训练得出的模型预测数据
```

```

y_plot = model.predict(Xi)
##打印线性方程的权重
print(model.coef_) ### 0.90045842
##绘图
plt.scatter(Xi, Yi, color='red',label="样本数据",linewidth=2)
plt.plot(Xi, y_plot, color='green',label="拟合直线",linewidth=2)
plt.legend(loc='lower right')
plt.show()

```

2) 示例程序 2

```

from sklearn.preprocessing import PolynomialFeatures
from sklearn.pipeline import make_pipeline
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import Ridge

##样本数据(Xi,Yi)，需要转换成数组(列表)形式
Xi=np.array([1,2,3,4,5,6]).reshape(-1,1)
#Yi=np.array([9,18,31,48,69,94])
Yi=np.array([9.1,18.3,32,47,69.5,94.8]).reshape(-1,1)
##这里指定使用岭回归作为基函数
model = make_pipeline(PolynomialFeatures(2), Ridge())
model.fit(Xi, Yi)
##根据模型预测结果
y_plot = model.predict(Xi)

##绘图
plt.scatter(Xi, Yi, color='red',label="样本数据",linewidth=2)
plt.plot(Xi, y_plot, color='green',label="拟合直线",linewidth=2)
plt.legend(loc='lower right')
plt.show()

```

4) 示例程序 3

```

import numpy as np
import matplotlib.pyplot as plt
# 使用交叉验证的方法，把数据集分为训练集合测试集
from sklearn.model_selection import train_test_split
from sklearn import datasets
from sklearn.linear_model import LogisticRegression
# 加载 iris 数据集
diabetes = datasets.load_iris()
# 将数据集拆分为训练集和测试集
X_train, X_test, y_train, y_test = train_test_split(

```

```

diabetes.data, diabetes.target, test_size=0.30, random_state=0)
# 使用 LogisticRegression 考察线性回归的预测能力
# 选择模型
cls = LogisticRegression()
# 把数据交给模型训练
cls.fit(X_train, y_train)
print("Coefficients:%s, intercept %s"%(cls.coef_,cls.intercept_))
print("Residual sum of squares: %.2f"% np.mean((cls.predict(X_test) -
y_test) ** 2))
print('Score: %.2f' % cls.score(X_test, y_test))

```

5.2 降维算法（PCA）

1) 算法原理

主成分分析（Principal components analysis，以下简称 PCA）是最重要的降维方法之一。在数据压缩消除冗余和数据噪音消除等领域都有广泛的应用。一般我们提到降维最容易想到的算法就是 PCA，下面我们就对 PCA 的原理做一个总结。

PCA 顾名思义，就是找出数据里最主要的方面，用数据里最主要的方面来代替原始数据。具体的，假如我们的数据集是 n 维的，共有 m 个数据 $(x(1), x(2), \dots, x(m))$ 。我们希望将这 m 个数据的维度从 n 维降到 n' 维，希望这 m 个 n' 维的数据集尽可能的代表原始数据集。

从上面两节我们可以看出，求样本 $x(i)$ 的 n' 维的主成分其实就是求样本集的协方差矩阵 XX^T 的前 n' 个特征值对应特征向量矩阵 W ，然后对于每个样本 $x(i)$ ，做如下变换 $z(i)=W^T x(i)$ ，即达到降维的 PCA 目的。

下面我们看看具体的算法流程。

输入： n 维样本集 $D=(x(1), x(2), \dots, x(m))$ ，要降维到的维数 n' 。

输出：降维后的样本集 D'

1) 对所有的样本进行中心化： $x(i) = x(i) - 1/m \sum_{j=1}^m x(j)$

2) 计算样本的协方差矩阵 XX^T

3) 对矩阵 XX^T 进行特征值分解

4) 取出最大的 n' 个特征值对应的特征向量 $(w_1, w_2, \dots, w_{n'})$ ，将所有的特征向量标准化后，组成特征向量矩阵 W 。

5) 对样本集中的每一个样本 $x(i)$ ，转化为新的样本 $z(i)=W^T x(i)$

6) 得到输出样本集 $D'=(z(1), z(2), \dots, z(m))$

有时候，我们不指定降维后的 n' 的值，而是换种方式，指定一个降维到的主成分比重阈值 t 。这个阈值 t 在 $(0, 1]$ 之间。假如我们的 n 个特征值为 $\lambda_1 \geq \lambda_2 \geq \dots \geq \lambda_n$ 则 n' 可以通过下式得到： $\sum_{i=1}^{n'} \lambda_i / \sum_{i=1}^n \lambda_i \geq t$

PCA 算法的主要优点有：

- 1) 仅仅需要以方差衡量信息量，不受数据集以外的因素影响。
- 2) 各主成分之间正交，可消除原始数据成分间的相互影响的因素。
- 3) 计算方法简单，主要运算是特征值分解，易于实现。

PCA 算法的主要缺点有：

- 1) 主成分各个特征维度的含义具有一定的模糊性，不如原始样本特征的解释性强。
- 2) 方差小的非主成分也可能含有对样本差异的重要信息，因降维丢弃可能对后续数据处理有影响。

2) 示例程序 1

```
import numpy as np
from sklearn.decomposition import PCA
x = np.array([[ -1, -1], [-2, -1], [-3, -2], [1, 1], [2, 1], [3, 2]])
pca = PCA(n_components=2)
pca.fit(x)
print(pca.explained_variance_ratio_)
```

3) 示例程序 2

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from sklearn.datasets.samples_generator import make_blobs
# X 为样本特征，Y 为样本簇类别， 共 1000 个样本，每个样本 3 个特征，共 4 个簇
X, y = make_blobs(n_samples=1000, n_features=3, centers=[[3,3, 3], [0,0,0], [1,1,1], [2,2,2]],
                  cluster_std=[0.2, 0.1, 0.2, 0.2],
                  random_state=9)

fig = plt.figure()
ax = Axes3D(fig, rect=[0, 0, 1, 1], elev=30, azimuth=20)
plt.scatter(X[:, 0], X[:, 1], X[:, 2], marker='o')

from sklearn.decomposition import PCA
pca = PCA(n_components=3)
pca.fit(X)
print(pca.explained_variance_ratio_)
print(pca.explained_variance_)

pca = PCA(n_components=2)
pca.fit(X)
print(pca.explained_variance_ratio_)
print(pca.explained_variance_)

X_new = pca.transform(X)
plt.figure("3D1")
ax.scatter(X_new[:,0], X_new[:,1], marker='o')
plt.show()

pca = PCA(n_components=0.95)
pca.fit(X)
```

```
print(pca.explained_variance_ratio_)  
print(pca.explained_variance_)  
print(pca.n_components_)
```

```
pca = PCA(n_components=0.99)  
pca.fit(X)  
print(pca.explained_variance_ratio_)  
print(pca.explained_variance_)  
print(pca.n_components_)
```

6 综合练习

应用上述算法思想对股票、房价（新房、二手房）进行预测，或者根据一些个人信息对要购买的物品进行推荐。