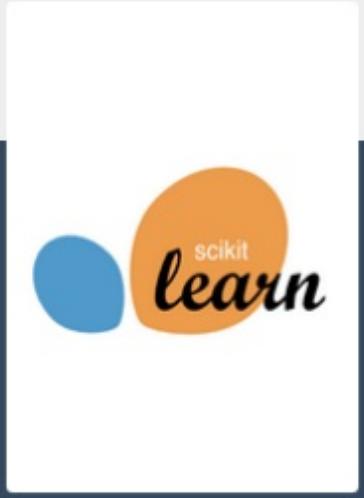
scikit-learn (sklearn) 0.21.3 官方文档中文版



scikit-learn 是基于 Python 语言的机器学习工具，简单高效的数据挖掘和数据分析工具，可供大家在各种环境中重复使用，建立在 NumPy ，SciPy 和 matplotlib 上，开源，可商业使用 - BSD许可证

下载手机APP
畅享精彩阅读

目　录

致谢

Introduction

安装 scikit-learn

1. 监督学习

    1.1. 广义线性模型

    1.2. 线性和二次判别分析

    1.3. 内核岭回归

    1.4. 支持向量机

    1.5. 随机梯度下降

    1.6. 最近邻

    1.7. 高斯过程

    1.8. 交叉分解

    1.9. 朴素贝叶斯

    1.10. 决策树

    1.11. 集成方法

    1.12. 多类和多标签算法

    1.13. 特征选择

    1.14. 半监督学习

    1.15. 等式回归

    1.16. 概率校准

    1.17. 神经网络模型（有监督）

2. 无监督学习

    2.1. 高斯混合模型

    2.2. 流形学习

    2.3. 聚类

    2.4. 双聚类

    2.5. 分解成分中的信号（矩阵分解问题）

    2.6. 协方差估计

    2.7. 新奇和异常值检测

    2.8. 密度估计

    2.9. 神经网络模型（无监督）

3. 模型选择和评估

    3.1. 交叉验证：评估估算器的表现

    3.2. 调整估计器的超参数

    3.3. 模型评估: 量化预测的质量

    3.4. 模型持久化

3.5. 验证曲线: 绘制分数以评估模型

4. 检验

4.1. 部分依赖图

5. 数据集转换

5.1. Pipeline（管道）和 FeatureUnion（特征联合）: 合并的评估器

5.2. 特征提取

5.3 预处理数据

5.4 缺失值插补

5.5. 无监督降维

5.6. 随机投影

5.7. 内核近似

5.8. 成对的矩阵, 类别和核函数

5.9. 预测目标 (y) 的转换

教程

使用 scikit-learn 介绍机器学习

关于科学数据处理的统计学习教程

机器学习: scikit-learn 中的设置以及预估对象

监督学习：从高维观察预测输出变量

模型选择：选择估计量及其参数

无监督学习: 寻求数据表示

把它们放在一起

寻求帮助

处理文本数据

选择正确的评估器(estimator.md)

外部资源，视频和谈话

常见问题

时光轴

致谢

当前文档 《scikit-learn (sklearn) 0.21.3 官方文档中文版》 由 进击的皇虫 使用 书栈网 (BookStack.CN) 进行构建，生成于 2020-03-01。

书栈网仅提供文档编写、整理、归类等功能，以及对文档内容的生成和导出工具。

文档内容由网友们编写和整理，书栈网难以确认文档内容知识点是否错漏。如果您在阅读文档获取知识的时候，发现文档内容有不恰当的地方，请向我们反馈，让我们共同携手，将知识准确、高效且有效地传递给每一个人。

同时，如果您在日常工作、生活和学习中遇到有价值有营养的知识文档，欢迎分享到书栈网，为知识的传承献上您的一份力量！

如果当前文档生成时间太久，请到书栈网获取最新的文档，以跟上知识更新换代的步伐。

内容来源：ApacheCN https://github.com/apachecn/scikit-learn-doc-zh/

文档地址：http://www.bookstack.cn/books/sklearn-0.21.3

书栈官网：https://www.bookstack.cn

书栈开源：https://github.com/TruthHun

分享，让知识传承更久远！ 感谢知识的创造者，感谢知识的分享者，也感谢每一位阅读到此处的读者，因为我们都将成为知识的传承者。

scikit-learn (sklearn) 官方文档中文版

scikit-learn 是基于 Python 语言的机器学习工具

- 简单高效的数据挖掘和数据分析工具
- 可供大家在各种环境中重复使用
- 建立在 NumPy ，SciPy 和 matplotlib 上
- 开源，可商业使用 - BSD许可证

维护地址

- Github
- 在线阅读
- EPUB 格式

目录

- 安装 scikit-learn
- 用户指南
  - 1. 监督学习
    - 1.1. 广义线性模型
    - 1.2. 线性和二次判别分析
    - 1.3. 内核岭回归
    - 1.4. 支持向量机
    - 1.5. 随机梯度下降
    - 1.6. 最近邻
    - 1.7. 高斯过程
    - 1.8. 交叉分解
    - 1.9. 朴素贝叶斯
    - 1.10. 决策树
    - 1.11. 集成方法
    - 1.12. 多类和多标签算法
    - 1.13. 特征选择
    - 1.14. 半监督学习
    - 1.15. 等式回归
    - 1.16. 概率校准
    - 1.17. 神经网络模型（有监督）
  - 2. 无监督学习
    - 2.1. 高斯混合模型
    - 2.2. 流形学习
    - 2.3. 聚类
    - 2.4. 双聚类
    - 2.5. 分解成分中的信号（矩阵分解问题）
    - 2.6. 协方差估计

        - 2.7. 新奇和异常值检测
        - 2.8. 密度估计
        - 2.9. 神经网络模型（无监督）
    - 3. 模型选择和评估
        - 3.1. 交叉验证：评估估算器的表现
        - 3.2. 调整估计器的超参数
        - 3.3. 模型评估：量化预测的质量
        - 3.4. 模型持久化
        - 3.5. 验证曲线：绘制分数以评估模型
    - 4. 检验
        - 4.1. 部分依赖图
    - 5. 数据集转换
        - 5.1. Pipeline（管道）和 FeatureUnion（特征联合）：合并的评估器
        - 5.2. 特征提取
        - 5.3 预处理数据
        - 5.4 缺失值插补
        - 5.5. 无监督降维
        - 5.6. 随机投影
        - 5.7. 内核近似
        - 5.8. 成对的矩阵，类别和核函数
        - 5.9. 预测目标（`y`）的转换
    - 6. 数据集加载工具
        - 6.1. 通用数据集 API
        - 6.2. 玩具数据集
        - 6.3 真实世界中的数据集
        - 6.4. 样本生成器
        - 6.5. 加载其他数据集
    - 7. 使用scikit-learn计算
        - 7.1. 大规模计算的策略：更大量的数据
        - 7.2. 计算性能
        - 7.3. 并行性、资源管理和配置
- 教程
    - 使用 scikit-learn 介绍机器学习
    - 关于科学数据处理的统计学习教程
        - 机器学习：scikit-learn 中的设置以及预估对象
        - 监督学习：从高维观察预测输出变量
        - 模型选择：选择估计量及其参数
        - 无监督学习：寻求数据表示
        - 把它们放在一起
        - 寻求帮助
    - 处理文本数据
    - 选择正确的评估器(estimator.md)
    - 外部资源，视频和谈话
- API 参考
- 常见问题
- 时光轴

贡献指南

项目当前处于校对阶段，请查看贡献指南，并在整体进度中领取任务。

> 请您勇敢地去翻译和改进翻译。虽然我们追求卓越，但我们并不要求您做到十全十美，因此请不要担心因为翻译上犯错——在大部分情况下，我们的服务器已经记录所有的翻译，因此您不必担心会因为您的失误遭到无法挽回的破坏。（改编自维基百科）

项目负责人

格式: GitHub + QQ

- @mahaoyang: 992635910
- @loopyme: 3322728009
- 飞龙: 562826179
- 片刻: 529815144

– 负责人要求: (欢迎一起为 `sklearn 中文版本` 做贡献)

- 热爱开源，喜欢装逼
- 长期使用 sklearn(至少0.5年) + 提交Pull Requests>=3
- 能够有时间及时优化页面 bug 和用户 issues
- 试用期: 2个月
- 欢迎联系: 片刻 529815144

项目协议

以各项目协议为准。

ApacheCN 账号下没有协议的项目，一律视为 CC BY-NC-SA 4.0。

建议反馈

- 在我们的 apachecn/pytorch-doc-zh github 上提 issue.
- 发邮件到 Email: `apachecn@163.com`.
- 在我们的 QQ群-搜索: 交流方式 中联系群主/管理员即可.

赞助我们

微信	支付宝
	

我们一直在努力

apachecn/sklearn-doc-zh



安装 scikit-learn

校验者：@小瑶@Loopy翻译者：@片刻

> Note
>
> 如果你想为这个项目做出贡献，建议你 安装最新的开发版本 .

安装最新版本

Scikit-learn 要求:

- Python (>= 3.5),
- NumPy (>= 1.11.0),
- SciPy (>= 0.17.0),
- joblib (>= 0.11).

Scikit-learn绘图功能(即，函数以“plot_”开头，需要Matplotlib(>= 1.5.1)。一些scikit-learn示例可能需要一个或多个额外依赖项:scikit-image(>= 0.12.3)、panda(>= 0.18.0)。

> 警告: Scikit-learn 0.20是支持Python 2.7和Python 3.4的最后一个版本。Scikit-learn现在需要Python 3.5或更新版本。

如果你已经有一个合适的 numpy 和 scipy版本，安装 scikit-learn 最简单的方法是使用 `pip`

```
pip install -U scikit-learn
```

或者 `conda` :

```
conda install scikit-learn
```

如果您还没有安装 NumPy 或 SciPy，还可以使用 conda 或 pip 来安装它们。 当使用 pip 时，请确保使用了 *binary wheels*，并且 NumPy 和 SciPy 不会从源重新编译，这可能在使用操作系统和硬件的特定配置（如 Raspberry Pi 上的 Linux）时发生。 从源代码构建 numpy 和 scipy 可能是复杂的（特别是在 Windows 上），并且需要仔细配置，以确保它们与线性代数程序的优化实现链接。而是使用如下所述的第三方发行版。

如果您必须安装 scikit-learn 及其与 pip 的依赖关系，则可以将其安装为 `scikit-learn[alldeps]` 。 最常见的用例是 `requirements.txt` 用作 PaaS 应用程序或 Docker 映像的自动构建过程的一部分的文件。此选项不适用于从命令行进行手动安装。

> 注意 在PyPy上安装时，需要注意PyPy3-v5.10+、Numpy 1.14.0+和scipy 1.1.0+。

有关更多发行版的安装说明，请参阅其他发行版。要从源代码编译开发版本，或者在体系结构中没有可用的发行版时构建包，请参阅高级安装说明。

第三方发行版

如果您尚未安装具有 numpy 和 scipy 的 python 安装，建议您通过软件包管理器或通过 python 软件包进行安装。 这些与 numpy, scipy, scikit-learn, matplotlib 和许多其他有用的科学和数据处理库。

可用选项有：

Canopy 和 Anaconda 适用于所有支持的平台

Canopy 和 Anaconda 都运送了最新版本的 scikit-learn，另外还有一大批适用于 Windows，Mac OSX 和 Linux 的科学 python 库。

Anaconda 提供 scikit-learn 作为其免费分发的一部分.

Warning

升级或卸载使用 Anaconda 安装的 scikit-learn，或者 `conda` 不应该使用 **pip** 命令。代替：

升级 `scikit-learn` ：

```
1. conda update scikit-learn
```

卸载 `scikit-learn` ：

```
1. conda remove scikit-learn
```

使用 `pip install -U scikit-learn` 升级 or `pip uninstall scikit-learn` 卸载 可能无法正确删除 `conda` 命令安装的文件.

pip 升级和卸载操作仅适用于通过 `pip install` 安装的软件包.

WinPython 适用于 Windows

该 WinPython 项目分布 scikit-learn 作为额外的插件。

我们一直在努力

apachecn/sklearn-doc-zh



1. 监督学习

- 1.1. 广义线性模型
  - 1.1.1. 普通最小二乘法
  - 1.1.2. 岭回归
  - 1.1.3. Lasso
  - 1.1.4. 多任务 Lasso
  - 1.1.5. 弹性网络
  - 1.1.6. 多任务弹性网络
  - 1.1.7. 最小角回归
  - 1.1.8. LARS Lasso
  - 1.1.9. 正交匹配追踪法（OMP）
  - 1.1.10. 贝叶斯回归
  - 1.1.11. logistic 回归
  - 1.1.12. 随机梯度下降， SGD
  - 1.1.13. Perceptron（感知器）
  - 1.1.14. Passive Aggressive Algorithms（被动攻击算法）
  - 1.1.15. 稳健回归（Robustness regression）： 处理离群点（outliers）和模型错误
  - 1.1.16. 多项式回归: 用基函数展开线性模型
- 1.2. 线性和二次判别分析
  - 1.2.1. 使用线性判别分析来降维
  - 1.2.2. LDA 和 QDA 分类器的数学公式
  - 1.2.3. LDA 的降维数学公式
  - 1.2.4. Shrinkage（收缩）
  - 1.2.5. 预估算法
- 1.3. 内核岭回归
- 1.4. 支持向量机
  - 1.4.1. 分类
  - 1.4.2. 回归
  - 1.4.3. 密度估计， 异常（novelty）检测
  - 1.4.4. 复杂度
  - 1.4.5. 使用诀窍
  - 1.4.6. 核函数
  - 1.4.7. 数学公式
  - 1.4.8. 实现细节
- 1.5. 随机梯度下降
  - 1.5.1. 分类
  - 1.5.2. 回归
  - 1.5.3. 稀疏数据的随机梯度下降
  - 1.5.4. 复杂度
  - 1.5.5. 停止判据
  - 1.5.6. 实用小贴士
  - 1.5.7. 数学描述
  - 1.5.8. 实现细节

- 1.6. 最近邻
  - 1.6.1. 无监督最近邻
  - 1.6.2. 最近邻分类
  - 1.6.3. 最近邻回归
  - 1.6.4. 最近邻算法
  - 1.6.5. 最近质心分类
  - 1.6.6. 邻域成分分析
- 1.7. 高斯过程
  - 1.7.1. 高斯过程回归（GPR）
  - 1.7.2. GPR 示例
  - 1.7.3. 高斯过程分类（GPC）
  - 1.7.4. GPC 示例
  - 1.7.5. 高斯过程内核
- 1.8. 交叉分解
- 1.9. 朴素贝叶斯
  - 1.9.1. 高斯朴素贝叶斯
  - 1.9.2. 多项分布朴素贝叶斯
  - 1.9.3. 补充朴素贝叶斯
  - 1.9.4. 伯努利朴素贝叶斯
  - 1.9.5. 基于外存的朴素贝叶斯模型拟合
- 1.10. 决策树
  - 1.10.1. 分类
  - 1.10.2. 回归
  - 1.10.3. 多值输出问题
  - 1.10.4. 复杂度分析
  - 1.10.5. 实际使用技巧
  - 1.10.6. 决策树算法: ID3, C4.5, C5.0 和 CART
  - 1.10.7. 数学表达
- 1.11. 集成方法
  - 1.11.1. Bagging meta-estimator（Bagging 元估计器）
  - 1.11.2. 由随机树组成的森林
  - 1.11.3. AdaBoost
  - 1.11.4. Gradient Tree Boosting（梯度树提升）
  - 1.11.5. Voting Classifier（投票分类器）
  - 1.11.6. 投票回归器(Voting Regressor)
- 1.12. 多类和多标签算法
  - 1.12.1. 多标签分类格式
  - 1.12.2. 1对其余
  - 1.12.3. 1对1
  - 1.12.4. 误差校正输出代码
  - 1.12.5. 多输出回归
  - 1.12.6. 多输出分类
  - 1.12.7. 链式分类器
- 1.13. 特征选择
  - 1.13.1. 移除低方差特征
  - 1.13.2. 单变量特征选择

        - 1.13.3. 递归式特征消除
        - 1.13.4. 使用 SelectFromModel 选取特征
        - 1.13.5. 特征选取作为 pipeline（管道）的一部分
    - 1.14. 半监督学习
        - 1.14.1. 标签传播
    - 1.15. 等式回归
    - 1.16. 概率校准
    - 1.17. 神经网络模型（有监督）
        - 1.17.1. 多层感知器
        - 1.17.2. 分类
        - 1.17.3. 回归
        - 1.17.4. 正则化
        - 1.17.5. 算法
        - 1.17.6. 复杂度
        - 1.17.7. 数学公式
        - 1.17.8. 实用技巧
        - 1.17.9. 使用 warm_start 的更多控制

我们一直在努力

apachecn/sklearn-doc-zh



1.1. 广义线性模型

校验者：@专业吹牛逼的小明@Gladiator@Loopy@qinhanmin2014翻译者：@瓜牛@年纪大了反应慢了@Hazekiah@BWM-蜜蜂

本章主要讲述一些用于回归的方法，其中目标值 y 是输入变量 x 的线性组合。 数学概念表示为：如果 $\hat{y}$ 是预测值，那么有：

$$\hat{y}(w, x) = w_0 + w_1 x_1 + ... + w_p x_p$$

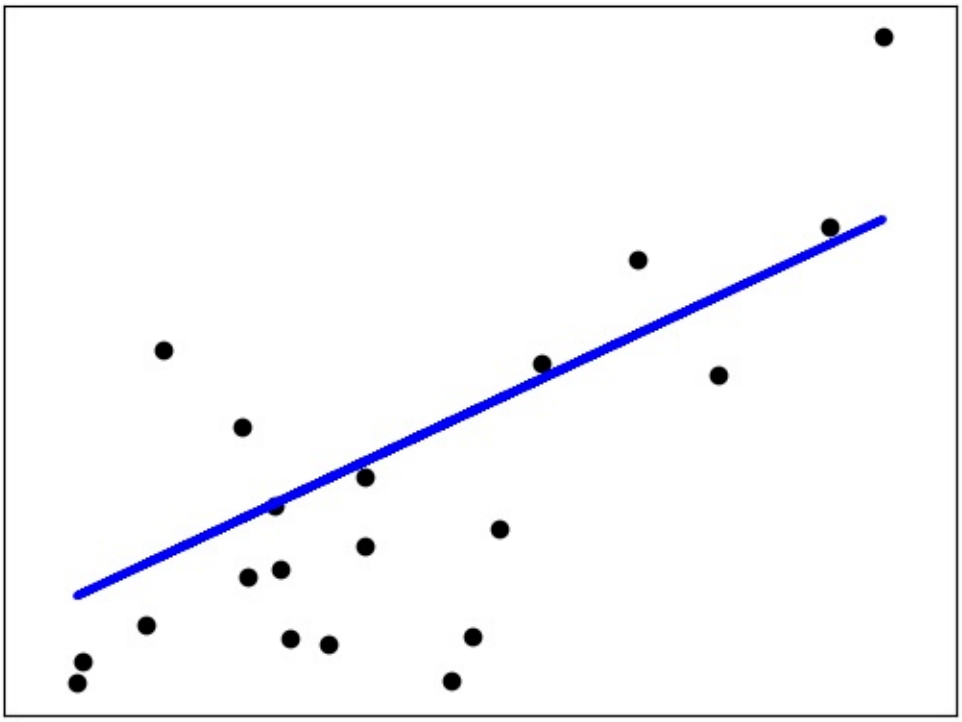
在整个模块中，我们定义向量 $w = (w_1, ..., w_p)$ 作为 `coef` ，定义 w_0 作为 `intercept` 。

如果需要使用广义线性模型进行分类，请参阅 logistic 回归 。

1.1.1. 普通最小二乘法

`LinearRegression` 拟合一个带有系数 $w = (w_1, ..., w_p)$ 的线性模型，使得数据集实际观测数据和预测数据（估计值）之间的残差平方和最小。其数学表达式为：

$$\min_w ||Xw - y||_2^2$$



`LinearRegression` 会调用 `fit` 方法来拟合数组 X， y，并且将线性模型的系数 w 存储在其成员变量 `coef_` 中：

```
>>> from sklearn import linear_model
>>> reg = linear_model.LinearRegression()
>>> reg.fit ([[0, 0], [1, 1], [2, 2]], [0, 1, 2])
LinearRegression(copy_X=True, fit_intercept=True, n_jobs=1, normalize=False)
>>> reg.coef_
array([ 0.5,  0.5])
```

然而，对于普通最小二乘的系数估计问题，其依赖于模型各项的相互独立性。当各项是相关的，且设计矩阵 X 的各列近似线性相关，那么，设计矩阵会趋向于奇异矩阵，这种特性导致最小二乘估计对于随机误差非常敏感，可能产生很大的方差。例如，在没有实验设计的情况下收集到的数据，这种多重共线性（multicollinearity）的情况可能真的会出现。

> 示例：
>
> - 线性回归示例

1.1.1.1. 普通最小二乘法的复杂度

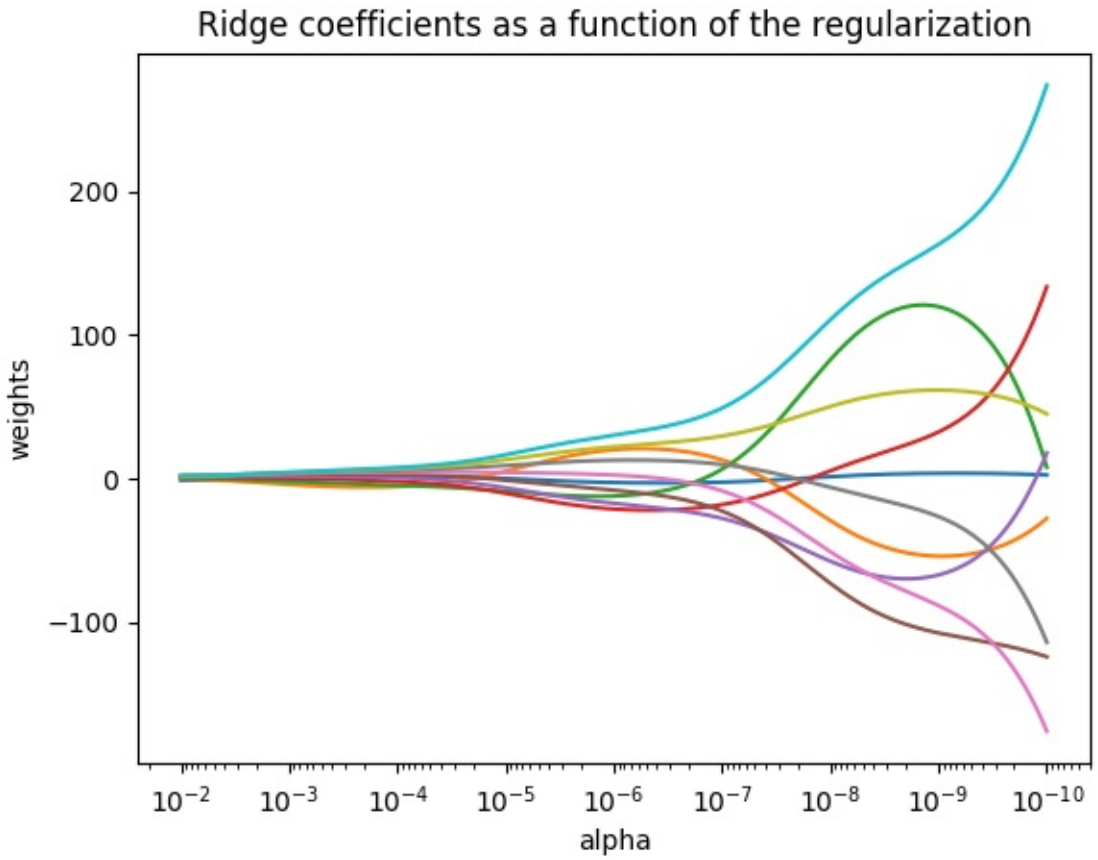
该方法使用 X 的奇异值分解来计算最小二乘解。如果 X 是一个形状为 `(nsamples, n_features)` 的矩阵，设*{% math %}n{samples} \geq n{features}{% endmath %},* 则该方法的复杂度为*{% math %}O(n{samples} n_{fearures}^2){% endmath %}*

1.1.2. 岭回归

`Ridge` 回归通过对系数的大小施加惩罚来解决 普通最小二乘法 的一些问题。 岭系数最小化的是带罚项的残差平方和，

$$\min_{w} ||Xw - y||_2^2 + \alpha ||w||_2^2$$

其中， $\alpha \geq 0$ 是控制系数收缩量的复杂性参数： α 的值越大，收缩量越大，模型对共线性的鲁棒性也更强。



与其他线性模型一样， `Ridge` 用 `fit` 方法完成拟合，并将模型系数 w 存储在其 `coef_` 成员中：

```
>>> from sklearn import linear_model
>>> reg = linear_model.Ridge (alpha = .5)
>>> reg.fit ([[0, 0], [0, 0], [1, 1]], [0, .1, 1])
Ridge(alpha=0.5, copy_X=True, fit_intercept=True, max_iter=None,
   normalize=False, random_state=None, solver='auto', tol=0.001)
>>> reg.coef_
array([ 0.34545455,  0.34545455])
>>> reg.intercept_
0.13636...
```

示例：

- 岭系数对回归系数的影响

- 分类特征稀疏的文本

1.1.2.1. 岭回归的复杂度

这种方法与 普通最小二乘法 的复杂度是相同的.

1.1.2.2. 设置正则化参数：广义交叉验证

RidgeCV 通过内置的关于的 alpha 参数的交叉验证来实现岭回归。 该对象与 GridSearchCV 的使用方法相同，只是它默认为 Generalized Cross-Validation(广义交叉验证 GCV)，这是一种有效的留一验证方法（LOO-CV）：

```
>>> from sklearn import linear_model
>>> reg = linear_model.RidgeCV(alphas=[0.1, 1.0, 10.0])
>>> reg.fit([[0, 0], [0, 0], [1, 1]], [0, .1, 1])
RidgeCV(alphas=[0.1, 1.0, 10.0], cv=None, fit_intercept=True, scoring=None,
    normalize=False)
>>> reg.alpha_
0.1
```

指定cv属性的值将触发(通过GridSearchCV的)交叉验证。例如，cv=10将触发10折的交叉验证，而不是广义交叉验证(GCV)。

参考资料

- “Notes on Regularized Least Squares”, Rifkin & Lippert (technical report, course slides).

1.1.3. Lasso

Lasso 是拟合稀疏系数的线性模型。 它在一些情况下是有用的，因为它倾向于使用具有较少参数值的情况，有效地减少给定解决方案所依赖变量的数量。 因此，Lasso 及其变体是压缩感知领域的基础。 在一定条件下，它可以恢复一组非零权重的精确集（见压缩感知_断层重建）。

在数学公式表达上，它由一个带有 ℓ_1 先验的正则项的线性模型组成。 其最小化的目标函数是：

$$\min_w \frac{1}{2n_{samples}}||Xw - y||_2^2 + \alpha||w||_1$$

lasso estimate 解决了加上罚项 $\alpha||w||_1$ 的最小二乘法的最小化，其中， α 是一个常数， $||w||_1$ 是参数向量的 ℓ_1-norm 范数。

Lasso 类的实现使用了 coordinate descent （坐标下降算法）来拟合系数。 查看 最小角回归 ，这是另一种方法：

```
>>> from sklearn import linear_model
```

```
>>> reg = linear_model.Lasso(alpha = 0.1)
>>> reg.fit([[0, 0], [1, 1]], [0, 1])
Lasso(alpha=0.1, copy_X=True, fit_intercept=True, max_iter=1000,
   normalize=False, positive=False, precompute=False, random_state=None,
   selection='cyclic', tol=0.0001, warm_start=False)
>>> reg.predict([[1, 1]])
array([ 0.8])
```

对于较简单的任务，同样有用的是函数 `lasso_path` 。它能够通过搜索所有可能的路径上的值来计算系数。

> 示例：
>
> - Lasso和Elastic Net(弹性网络)在稀疏信号上的表现
> - 压缩感知_断层重建
>
> 注意：使用 **Lasso** 进行特征选择
>
> 由于 Lasso 回归产生稀疏模型，因此可以用于执行特征选择，详见 基于 L1 的特征选取 。

下面两篇参考解释了scikit-learn坐标下降算法中使用的迭代，以及用于收敛控制的对偶间隙计算的理论基础。

> 参考资料
>
> - “Regularization Path For Generalized linear Models by Coordinate Descent”, Friedman, Hastie & Tibshirani, J Stat Softw, 2010 (Paper).
> - “An Interior-Point Method for Large-Scale L1-Regularized Least Squares,” S. J. Kim, K. Koh, M. Lustig, S. Boyd and D. Gorinevsky, in IEEE Journal of Selected Topics in Signal Processing, 2007 (Paper)

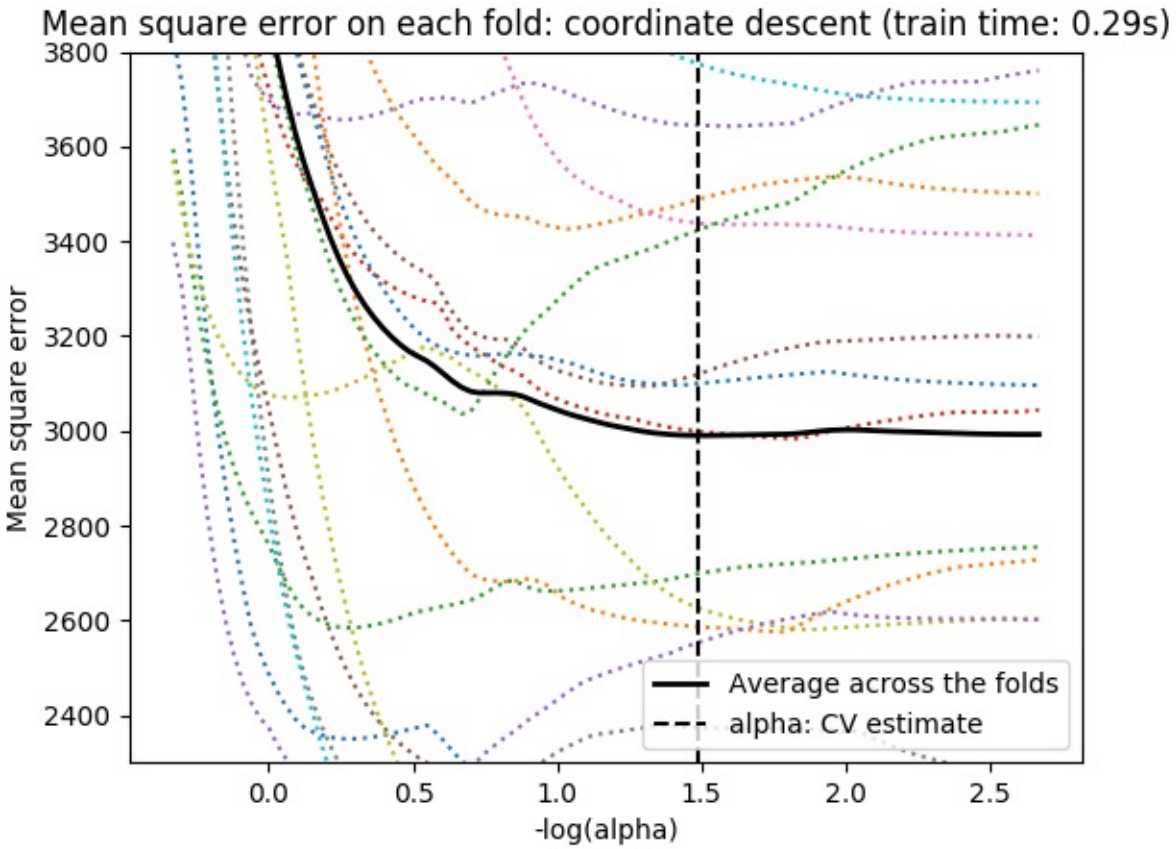
1.1.3.1. 设置正则化参数

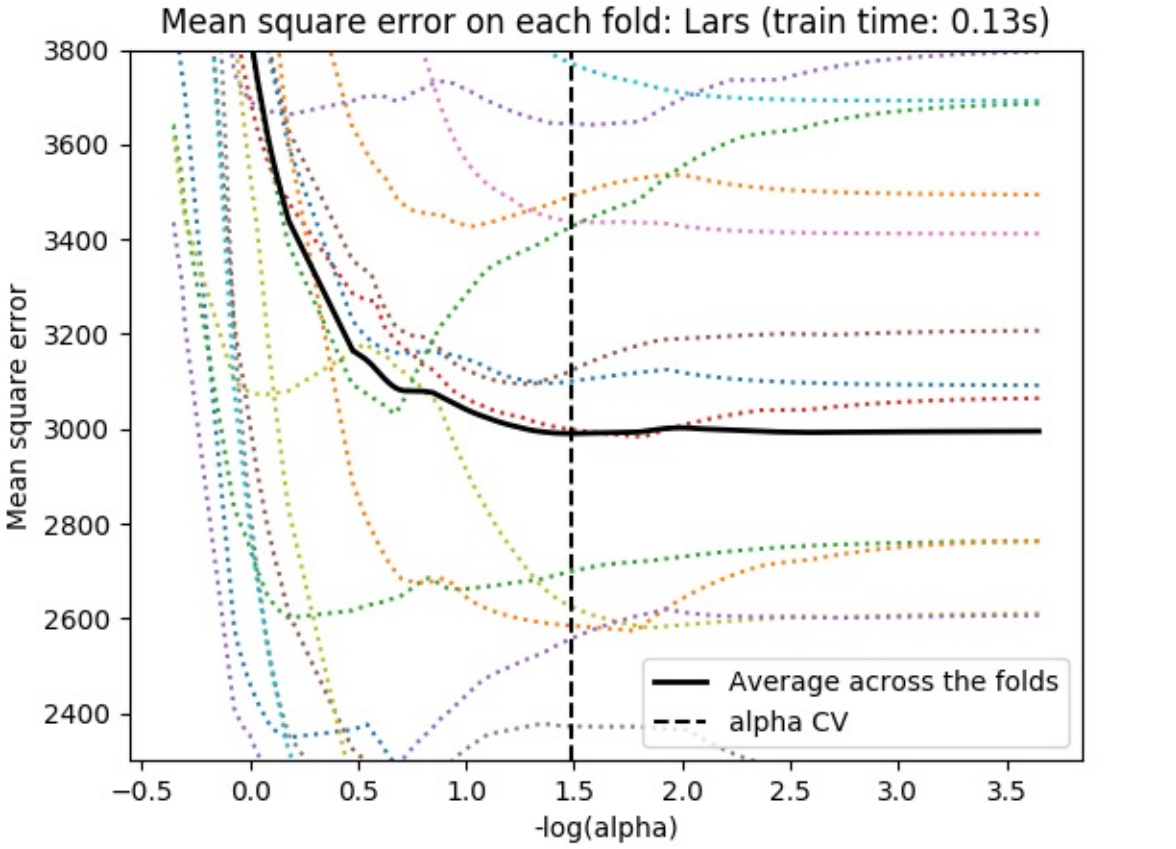
`alpha` 参数控制估计系数的稀疏度。

1.1.3.1.1. 使用交叉验证

scikit-learn 通过交叉验证来公开设置 Lasso `alpha` 参数的对象： `LassoCV` 和 `LassoLarsCV` 。 `LassoLarsCV` 是基于下面将要提到的 最小角回归 算法。

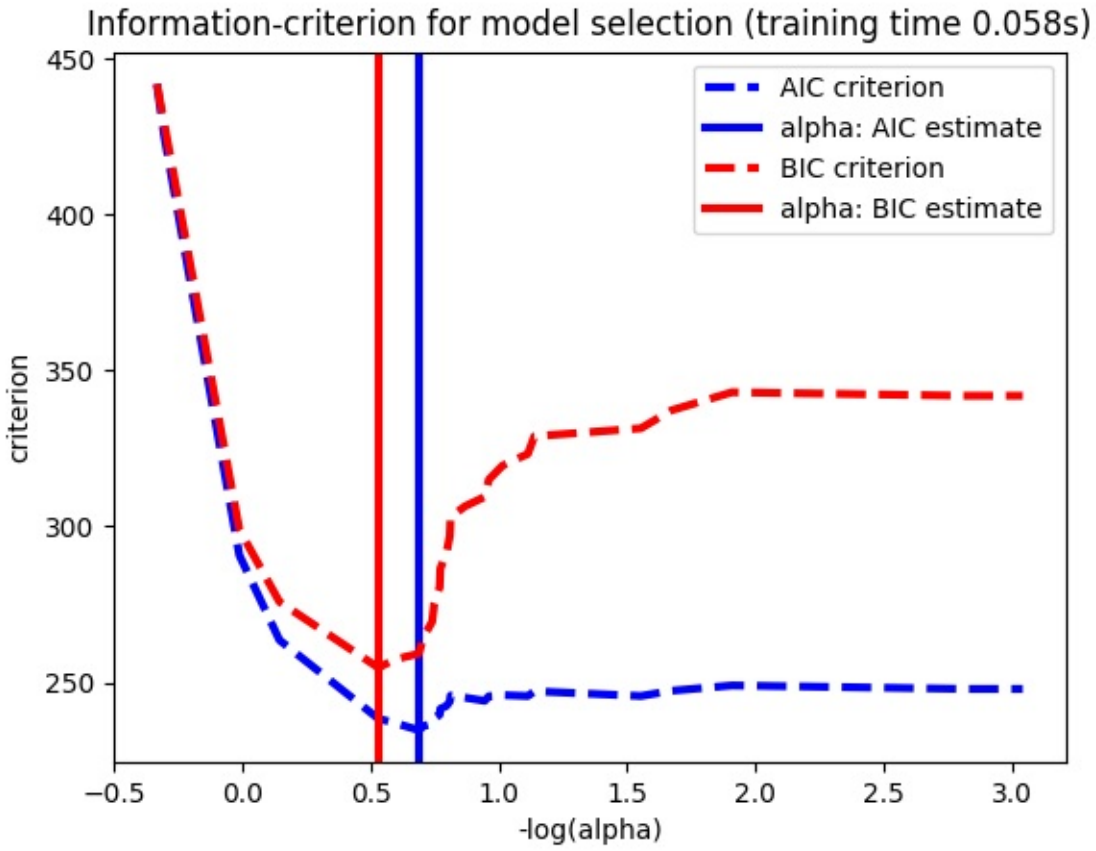
对于具有许多线性回归的高维数据集， `LassoCV` 最常见。 然而， `LassoLarsCV` 在寻找 `alpha` 参数值上更具有优势，而且如果样本数量比特征数量少得多时，通常 `LassoLarsCV` 比 `LassoCV` 要快。





1.1.3.1.2. 基于信息标准的模型选择

有多种选择时，估计器 `LassoLarsIC` 建议使用 Akaike information criterion （Akaike 信息判据）（AIC）或 Bayes Information criterion （贝叶斯信息判据）（BIC）。 当使用 k-fold 交叉验证时，正则化路径只计算一次而不是 k + 1 次，所以找到 α 的最优值是一种计算上更经济的替代方法。 然而，这样的判据需要对解决方案的自由度进行适当的估计，它会假设模型是正确的，对大样本（渐近结果）进行导出，即数据实际上是由该模型生成的。 当问题严重受限（比样本更多的特征）时，它们也容易崩溃。



示例：

- Lasso模型选择-交叉验证/AIC/BIC

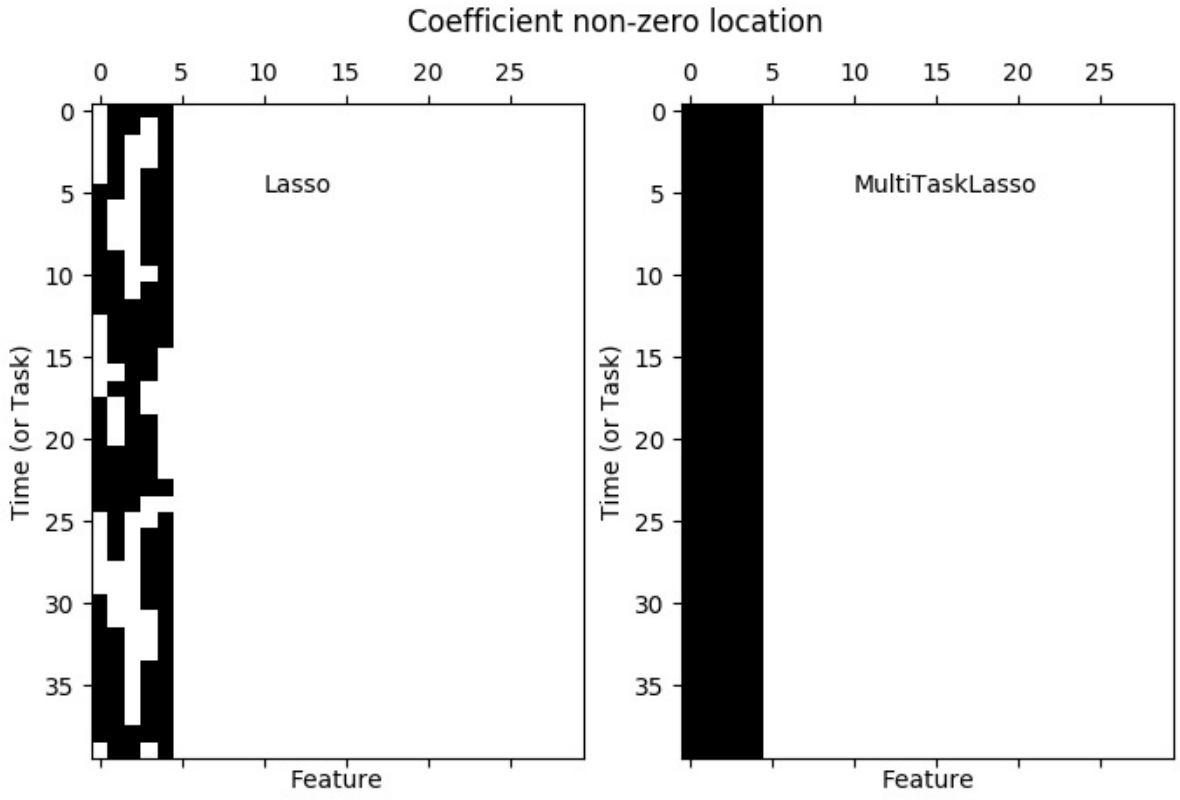
1.1.3.1.3. 与 SVM 的正则化参数的比较

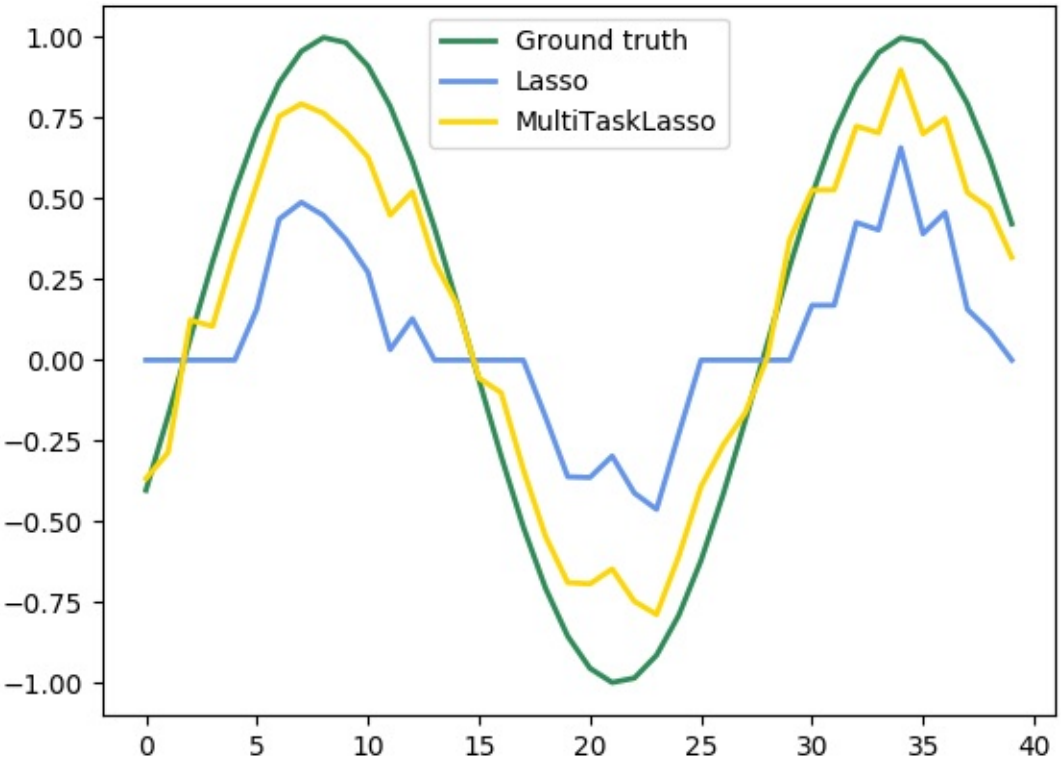
`alpha` 和 SVM 的正则化参数 `C` 之间的等式关系是 `alpha = 1 / C` 或者 `alpha = 1 / (n_samples * C)` ，并依赖于估计器和模型优化的确切的目标函数。

1.1.4. 多任务 Lasso

`MultiTaskLasso` 是一个估计多元回归稀疏系数的线性模型： `y` 是一个形状为 `(n_samples, n_tasks)` 的二维数组，其约束条件和其他回归问题（也称为任务）是一样的，都是所选的特征值。

下图比较了通过使用简单的 Lasso 或 MultiTaskLasso 得到的 W 中非零的位置。 Lasso 估计产生分散的非零值，而 MultiTaskLasso 的一整列都是非零的。





拟合 **time-series model**（时间序列模型），强制任何活动的功能始终处于活动状态。

> 示例：

- 多任务Lasso实现联合特征选择

在数学上，它由一个线性模型组成，以混合的 $\ell_1\ell_2$ 作为正则化器进行训练。目标函数最小化是：

$$\min_W \frac{1}{2n_{samples}}||XW - Y||_{Fro}^2 + \alpha||W||_{21}$$

其中 Fro 表示 Frobenius 标准：

$$||A||_{Fro} = \sqrt{\sum_{ij} a_{ij}^2}$$

并且 $\ell_1\ell_2$ 读取为：

$$||A||_{21} = \sum_i \sqrt{\sum_j a_{ij}^2}$$

`MultiTaskLasso` 类的实现使用了坐标下降作为拟合系数的算法。

1.1.5. 弹性网络

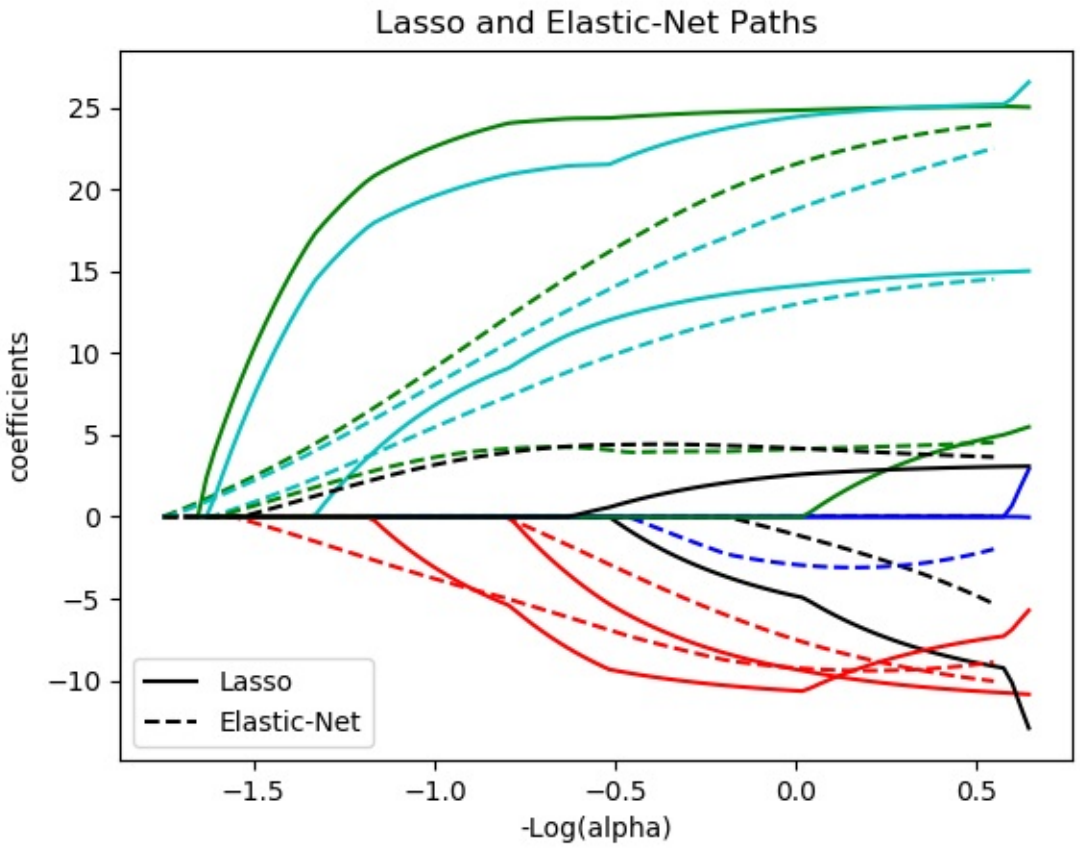
`弹性网络` 是一种使用 L1， L2 范数作为先验正则项训练的线性回归模型。 这种组合允许拟合到一个只有少量参数是非零稀疏的模型，就像 `Lasso` 一样，但是它仍然保持了一些类似于 `Ridge` 的正则性质。我们可利用 `l1_ratio` 参数控制 L1 和 L2 的凸组合。

弹性网络在很多特征互相联系的情况下是非常有用的。Lasso 很可能只随机考虑这些特征中的一个，而弹性网络更倾向于选择两个。

在实践中，Lasso 和 Ridge 之间权衡的一个优势是它允许在循环过程（Under rotate）中继承 Ridge 的稳定性。

在这里，最小化的目标函数是

$$\min_w \frac{1}{2n_{samples}}||Xw - y||_2^2 + \alpha\rho||w||_1 + \frac{\alpha(1-\rho)}{2}||w||_2^2$$



`ElasticNetCV` 类可以通过交叉验证来设置参数 `alpha` （ α ）和 `l1_ratio` （ ρ ）。

> 示例：
>
> - Lasso和Elastic Net(弹性网络)在稀疏信号上的表现
> - Lasso和Elastic Net

下面两篇参考解释了scikit-learn坐标下降算法中使用的迭代，以及用于收敛控制的对偶间隙计算的理论基础。

> 参考资料
>
> - “Regularization Path For Generalized linear Models by Coordinate Descent”, Friedman, Hastie & Tibshirani, J Stat Softw, 2010 (Paper).
> - “An Interior-Point Method for Large-Scale L1-Regularized Least Squares,” S. J. Kim, K. Koh, M. Lustig, S. Boyd and D. Gorinevsky, in IEEE Journal of Selected Topics in Signal Processing, 2007 (Paper)

1.1.6. 多任务弹性网络

`MultiTaskElasticNet` 是一个对多回归问题估算稀疏参数的弹性网络： `Y` 是一个二维数组，形状是 `(n_samples,n_tasks)` 。 其限制条件是和其他回归问题一样，是选择的特征，也称为 `tasks` 。

从数学上来说， 它包含一个混合的 $\ell_1\ell_2$ 先验和 ℓ_2 先验为正则项训练的线性模型 目标函数就是最小化：

$$\min_W \frac{1}{2n_{samples}}||XW - Y||^2_{Fro} + \alpha\rho||W||_{21} + \frac{\alpha(1-\rho)}{2}||W||^2_{Fro}$$

在 `MultiTaskElasticNet` 类中的实现采用了坐标下降法求解参数。

在 `MultiTaskElasticNetCV` 中可以通过交叉验证来设置参数 `alpha` （ α ）和 `l1_ratio` （ ρ ）。

1.1.7. 最小角回归

最小角回归 （LARS） 是对高维数据的回归算法， 由 Bradley Efron, Trevor Hastie, Iain Johnstone 和 Robert Tibshirani 开发完成。 LARS 和逐步回归很像。在每一步，它都寻找与响应最有关联的预测。当有很多预测有相同的关联时，它并不会继续利用相同的预测，而是在这些预测中找出应该等角的方向。

LARS的优点：

- 当 p >> n，该算法数值运算上非常有效。(例如当维度的数目远超点的个数)
- 它在计算上和前向选择一样快，和普通最小二乘法有相同的运算复杂度。
- 它产生了一个完整的分段线性的解决路径，在交叉验证或者其他相似的微调模型的方法上非常有用。
- 如果两个变量对响应几乎有相等的联系，则它们的系数应该有相似的增长率。因此这个算法和我们直觉 上的判断一样，而且还更加稳定。
- 它很容易修改并为其他估算器生成解，比如Lasso。

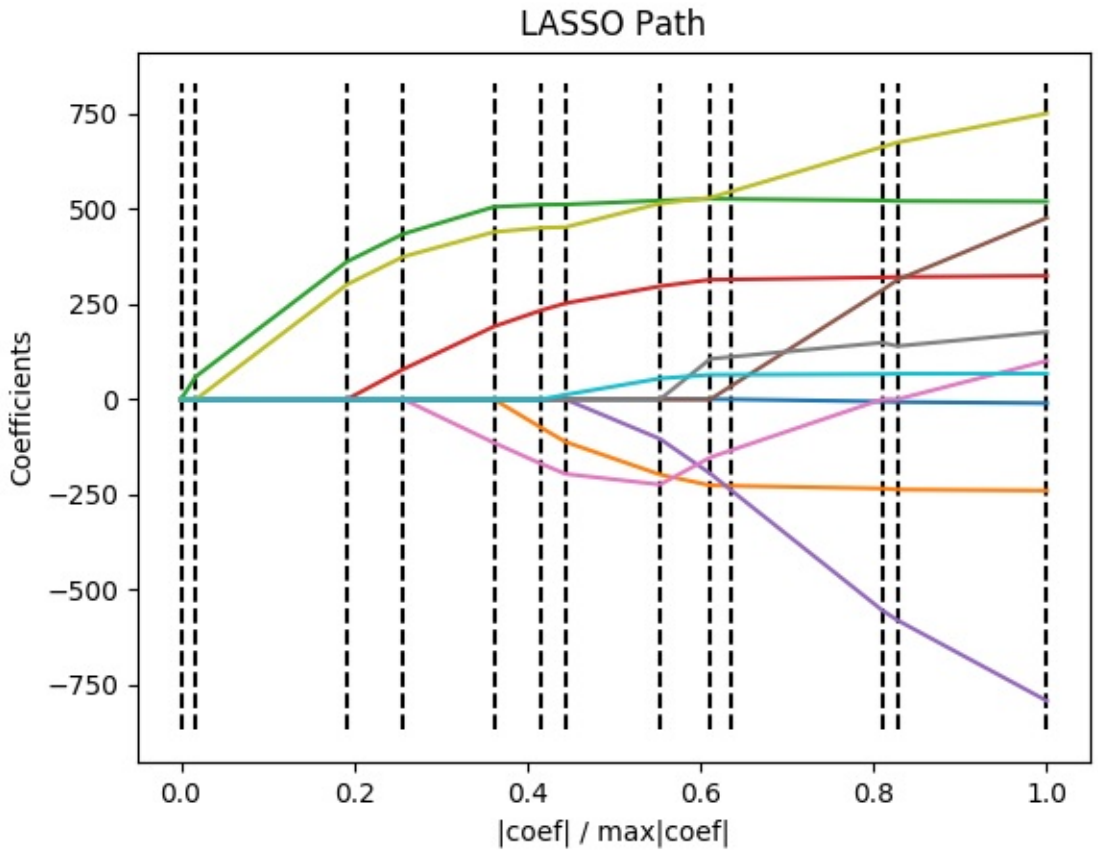
LARS 的缺点：

- 因为 LARS 是建立在循环拟合剩余变量上的，所以它对噪声非常敏感。这个问题，在 2004 年统计年鉴的文章由 Weisberg 详细讨论。

LARS 模型可以在 `Lars` ，或者它的底层实现 `lars_path` 或 `lars_path_gram` 中被使用。

1.1.8. LARS Lasso

`LassoLars` 是一个使用 LARS 算法的 lasso 模型，不同于基于坐标下降法的实现，它可以得到一个精确解，也就是一个关于自身参数标准化后的一个分段线性解。




```
>>> from sklearn import linear_model
>>> reg = linear_model.LassoLars(alpha=.1)
>>> reg.fit([[0, 0], [1, 1]], [0, 1])
LassoLars(alpha=0.1, copy_X=True, eps=..., fit_intercept=True,
     fit_path=True, max_iter=500, normalize=True, positive=False,
     precompute='auto', verbose=False)
>>> reg.coef_
array([0.717157..., 0.        ])
```

示例：

- 使用LARS计算Lasso路径

Lars 算法提供了一个几乎无代价的沿着正则化参数的系数的完整路径，因此常利用函数 `lars_path` 或 `lars_path_gram` 来取回路径。

1.1.8.1. 数学表达式

该算法和逐步回归非常相似，但是它没有在每一步包含变量，它估计的参数是根据与 其他剩余变量的联系来增加的。

在 LARS 的解中，没有给出一个向量的结果，而是给出一条曲线，显示参数向量的 L1 范式的每个值的解。 完全的参数路径存在 `coefpath` 下。它的 size 是 (n_features, max_features+1)。 其中第一列通常是全 0 列。

参考资料：

- Original Algorithm is detailed in the paper Least Angle Regression by Hastie et al.

1.1.9. 正交匹配追踪法（OMP）

`OrthogonalMatchingPursuit` （正交匹配追踪法)和 `orthogonal_mp` 使用了 OMP 算法近似拟合了一个带限制的线性模型，该限制影响于模型的非 0 系数(例：L0 范数)。

就像最小角回归一样，作为一个前向特征选择方法，正交匹配追踪法可以近似一个固定非 0 元素的最优向量解：

$$\arg\min ||y - X\gamma||_2^2 \text{ subject to } ||\gamma||_0 \leq n_{nonzero_coefs}$$

正交匹配追踪法也可以针对一个特殊的误差而不是一个特殊的非零系数的个数。可以表示为：

$$\arg\min ||\gamma||_0 \text{ subject to } ||y - X\gamma||_2^2 \leq \text{tol}$$

OMP 是基于每一步的贪心算法，其每一步元素都是与当前残差高度相关的。它跟较为简单的匹配追踪（MP）很相似，但是相比 MP 更好，在每一次迭代中，可以利用正交投影到之前选择的字典元素重新计算残差。

示例：

- 正交匹配追踪

参考资料：

- http://www.cs.technion.ac.il/~ronrubin/Publications/KSVD-OMP-v2.pdf
- Matching pursuits with time-frequency dictionaries, S. G. Mallat, Z. Zhang,

1.1.10. 贝叶斯回归

贝叶斯回归可以用于在预估阶段的参数正则化：正则化参数的选择不是通过人为的选择，而是通过手动调节数据值来实现。

上述过程可以通过引入 无信息先验 到模型中的超参数来完成。 在 岭回归 中使用的 ℓ_2 正则项相当于在 w 为高斯先验条件，且此先验的精确度为 λ^{-1} 时，求最大后验估计。在这里，我们没有手工调参数 `lambda` ，而是让他作为一个变量，通过数据中估计得到。

为了得到一个全概率模型，输出 y 也被认为是关于 Xw 的高斯分布。

$$p(y|X, w, \alpha) = \mathcal{N}(y|Xw, \alpha)$$

`Alpha` 在这里也是作为一个变量，通过数据中估计得到。

贝叶斯回归有如下几个优点：

- 它能根据已有的数据进行改变。
- 它能在估计过程中引入正则项。

贝叶斯回归有如下缺点：

- 它的推断过程是非常耗时的。

> 参考资料
>
> - 一个对于贝叶斯方法的很好的介绍 C. Bishop: Pattern Recognition and Machine learning
> - 详细介绍原创算法的一本书 Bayesian learning for neural networks by Radford M. Neal

1.1.10.1. 贝叶斯岭回归

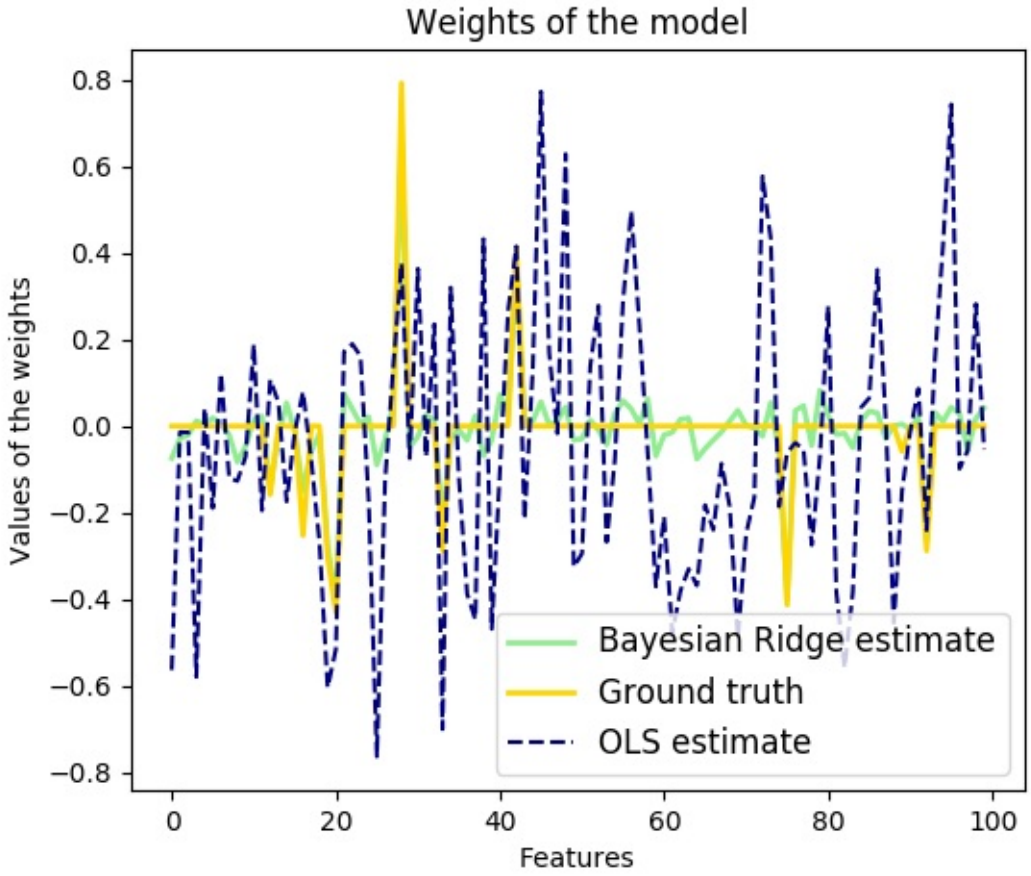
`BayesianRidge` 利用概率模型估算了上述的回归问题，其先验参数 w 是由以下球面高斯公式得出的：

$$p(w|\lambda) = \mathcal{N}(w|0, \lambda^{-1}\mathbf{I_p})$$

先验参数 α 和 λ 一般是服从 gamma 分布 ，这个分布与高斯成共轭先验关系。 得到的模型一般称为 贝叶斯岭回归，并且这个与传统的 `Ridge` 非常相似。

参数 w ， α 和 λ 是在模型拟合的时候一起被估算出来的，其中参数α和λ通过最大似然估计得到。scikit-learn的实现是基于文献（Tipping，2001）的附录A，参数α和λ的更新是基于文献（MacKay，1992）。

剩下的超参数α_1， α_2， λ_1以及λ_2是关于α和 λ 的 gamma 分布的先验。 它们通常被选择为 无信息先验 。默认 $\alpha_1 = \alpha_2 = \lambda_1 = \lambda_2 = 10^{-6}$ 。



贝叶斯岭回归用来解决回归问题：

```
>>> from sklearn import linear_model
>>> X = [[0., 0.], [1., 1.], [2., 2.], [3., 3.]]
>>> Y = [0., 1., 2., 3.]
>>> reg = linear_model.BayesianRidge()
>>> reg.fit(X, Y)
BayesianRidge(alpha_1=1e-06, alpha_2=1e-06, compute_score=False, copy_X=True,
 fit_intercept=True, lambda_1=1e-06, lambda_2=1e-06, n_iter=300,
 normalize=False, tol=0.001, verbose=False)
```

在模型训练完成后，可以用来预测新值：

```
>>> reg.predict ([[1, 0.]])
array([ 0.50000013])
```

权值 w 可以被这样访问：

```
>>> reg.coef_
array([ 0.49999993,  0.49999993])
```

由于贝叶斯框架的缘故，权值与 普通最小二乘法 产生的不太一样。 但是，贝叶斯岭回归对病态问题（ill-posed）的鲁棒性要更好。

示例：

- 贝叶斯岭回归

参考资料

- Section 3.3 in Christopher M. Bishop: Pattern Recognition and Machine Learning, 2006
- David J. C. MacKay, Bayesian Interpolation, 1992.
- Michael E. Tipping, Sparse Bayesian Learning and the Relevance Vector Machine, 2001.

1.1.10.2. 主动相关决策理论 - ARD

`ARDRegression` （主动相关决策理论）和 `Bayesian Ridge Regression` 非常相似，但是会导致一个更加稀疏的权重 `w[1][2]` 。

`ARDRegression` 提出了一个不同的 w 的先验假设。具体来说，就是弱化了高斯分布为球形的假设。

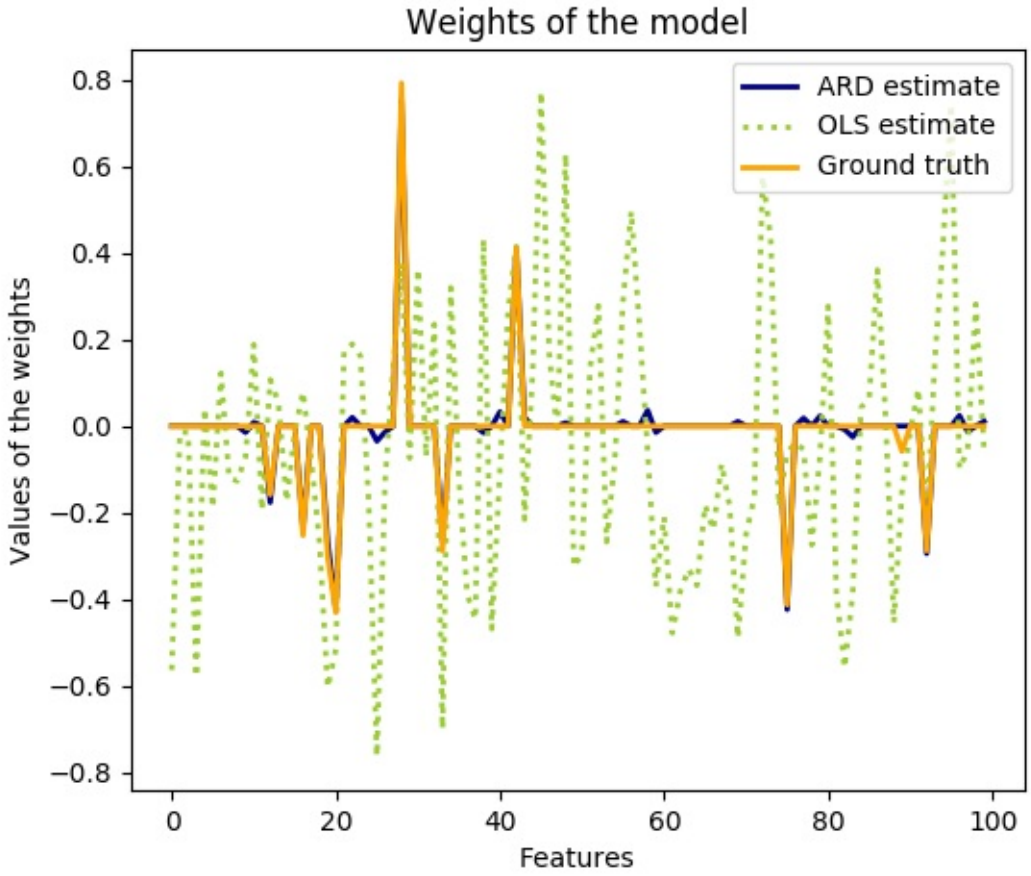
它采用 w 分布是与轴平行的椭圆高斯分布。

也就是说，每个权值 w_i 从一个中心在 `0` 点，精度为 λ_i 的高斯分布中采样得到的。

$$p(w|\lambda) = \mathcal{N}(w|0, A^{-1})$$

并且 $diag\ (A) = \lambda = \{\lambda_1, ..., \lambda_p\}$.

与 `Bayesian Ridge Regression` _ 不同， 每个 w_i 都有一个标准差 λ_i 。所有 λ_i 的先验分布 由超参数 λ_1 、 λ_2 确定的相同的 `gamma` 分布确定。



ARD 也被称为 稀疏贝叶斯学习 或 相关向量机 [3][4]。

> 示例：
>
> - Automatic Relevance Determination Regression (ARD)
>
> 参考资料：
>
> - [1] Christopher M. Bishop: Pattern Recognition and Machine Learning, Chapter 7.2.1
> - [2] David Wipf and Srikantan Nagarajan: A new view of automatic relevance determination
> - [3] Michael E. Tipping: Sparse Bayesian Learning and the Relevance Vector Machine
> - [4] Tristan Fletcher: Relevance Vector Machines explained

1.1.11. logistic 回归

logistic 回归，虽然名字里有 “回归” 二字，但实际上是解决分类问题的一类线性模型。在某些文献中，logistic 回归又被称作 logit 回归，maximum-entropy classification（MaxEnt，最大熵分类），或 log-linear classifier（对数线性分类器）。该模型利用函数 logistic function 将单次试验（single trial）的可能结果输出为概率。

scikit-learn 中 logistic 回归在 `LogisticRegression` 类中实现了二分类（binary）、一对多分类（one-vs-rest）及多项式 logistic 回归，并带有可选的 L1 和 L2 正则化。

> 注意，scikit-learn的逻辑回归在默认情况下使用L2正则化，这样的方式在机器学习领域是常见的，在统计分析领域是不常见的。正则化的另一优势是提升数值稳定性。scikit-learn通过将C设置为很大的值实现无正则化。

作为优化问题，带 L2罚项的二分类 logistic 回归要最小化以下代价函数（cost function）：

$$\min_{w,c} \frac{1}{2} w^T w + C \sum_{i=1}^{n} \log(\exp(-y_i(X_i^T w + c)) + 1).$$

类似地，带 L1 正则的 logistic 回归解决的是如下优化问题：

$$\min_{w,c} \|w\|_1 + C \sum_{i=1}^{n} \log(\exp(-y_i(X_i^T w + c)) + 1).$$

Elastic-Net正则化是L1 和 L2的组合，来使如下代价函数最小：

$$\min_{w,c} \frac{1-\rho}{2} w^T w + \rho \|w\|_1 + C \sum_{i=1}^{n} \log(\exp(-y_i(X_i^T w + c)) + 1)$$

其中ρ控制正则化L1与正则化L2的强度(对应于 `l1_ratio` 参数)。

注意，在这个表示法中，假定目标y_i在测试时应属于集合[-1,1]。我们可以发现Elastic-Net在ρ=1时与L1罚项等价,在ρ=0时与L2罚项等价

在 `LogisticRegression` 类中实现了这些优化算法： `liblinear` ， `newton-cg` ， `lbfgs` ， `sag` 和 `saga` 。

`liblinear` 应用了坐标下降算法（Coordinate Descent, CD），并基于 scikit-learn 内附的高性能 C++ 库 LIBLINEAR library 实现。不过 CD 算法训练的模型不是真正意义上的多分类模型，而是基于 “one-vs-rest” 思想分解了这个优化问题，为每个类别都训练了一个二元分类器。因为实现在底层使用该求解器的 `LogisticRegression` 实例对象表面上看是一个多元分类器。 `sklearn.svm.l1_min_c` 可以计算使用 L1时 C 的下界，以避免模型为空（即全部特征分量的权重为零）。

`lbfgs` ， `sag` 和 `newton-cg` 求解器只支持 L2罚项以及无罚项，对某些高维数据收敛更快。这些求解器的参数 `multi_class` 设为 `multinomial` 即可训练一个真正的多项式 logistic 回归 [5] ，其预测的概率比默认的 “ `one-vs-rest` ” 设定更为准确。

`sag` 求解器基于平均随机梯度下降算法（Stochastic Average Gradient descent） [6]。在大数据集上的表现更快，大数据集指样本量大且特征数多。

`saga` 求解器 [7] 是 `sag` 的一类变体，它支持非平滑（non-smooth）的 L1 正则选项 `penalty="l1"` 。因此对于稀疏多项式 logistic 回归 ，往往选用该求解器。 `saga` 求解器是唯一支持弹性网络正则选项的求解器。

`lbfgs` 是一种近似于Broyden–Fletcher–Goldfarb–Shanno算法[8]的优化算法，属于准牛顿法。 `lbfgs` 求解器推荐用于较小的数据集，对于较大的数据集，它的性能会受到影响。[9]

总的来说，各求解器特点如下：

罚项	`liblinear`	`lbfgs`	`newton-cg`	`sag`	`saga`
多项式损失+L2罚项	×	√	√	√	√

一对剩余（One vs Rest） + L2罚项	√	√	√	√	√
多项式损失 + L1罚项	×	×	×	×	√
一对剩余（One vs Rest） + L1罚项	√	×	×	×	√
弹性网络	×	×	×	×	√
无罚项	×	√	√	√	√
表现					
惩罚偏置值(差)	√	×	×	×	×
大数据集上速度快	×	×	×	√	√
未缩放数据集上鲁棒	√	√	√	×	×

默认情况下， `lbfgs` 求解器鲁棒性占优。对于大型数据集， `saga` 求解器通常更快。对于大数据集，还可以用 `SGDClassifier` ，并使用对数损失（ `log` loss）这可能更快，但需要更多的调优。

> 示例：
>
> - Logistic回归中的L1罚项和稀疏系数
> - L1罚项-logistic回归的路径
> - 多项式和OVR的Logistic回归
> - newgroups20上的多类稀疏Logistic回归
> - 使用多项式Logistic回归和L1进行MNIST数据集的分类
>
> 与 **`liblinear`** 的区别：
>
> 当 `fitintercept=False` 拟合得到的 `coef` 或者待预测的数据为零时，用 `solver=liblinear` 的 `LogisticRegression` 或 `LinearSVC` 与直接使用外部 liblinear 库预测得分会有差异。这是因为， 对于 `decision_function` 为零的样本， `LogisticRegression` 和 `LinearSVC` 将预测为负类，而 liblinear 预测为正类。 注意，设定了 `fit_intercept=False` ，又有很多样本使得 `decision_function` 为零的模型，很可能会欠拟合，其表现往往比较差。建议您设置 `fit_intercept=True` 并增大 `intercept_scaling` 。
>
> 注意:利用稀疏 **`logistic`** 回归进行特征选择
>
> 带 L1罚项的 logistic 回归 将得到稀疏模型（sparse model），相当于进行了特征选择（feature selection），详情参见 基于 L1 的特征选取。

`LogisticRegressionCV` 对 logistic 回归 的实现内置了交叉验证（cross-validation），可以找出最优的 `C` 和 `l1_ratio` 参数 。 `newton-cg` ， `sag` ， `saga` 和 `lbfgs` 在高维数据上更快，这是因为采用了热启动（warm-starting）。

> 参考资料：
>
> - [5] Christopher M. Bishop: Pattern Recognition and Machine Learning, Chapter 4.3.4
> - [6] Mark Schmidt, Nicolas Le Roux, and Francis Bach: Minimizing Finite Sums with the Stochastic Average Gradient.
> - [7] Aaron Defazio, Francis Bach, Simon Lacoste-Julien: SAGA: A Fast Incremental Gradient Method With Support for Non-Strongly Convex Composite Objectives.
> - [8] https://en.wikipedia.org/wiki/Broyden%E2%80%93Fletcher%E2%80%93Goldfarb%E2%80%93Shanno_algorithm
> - [9] “Performance Evaluation of Lbfgs vs other solvers”

1.1.12. 随机梯度下降， SGD

随机梯度下降是拟合线性模型的一个简单而高效的方法。在样本量（和特征数）很大时尤为有用。 方法

`partial_fit` 可用于 online learning （在线学习）或基于 out-of-core learning （外存的学习）

`SGDClassifier` 和 `SGDRegressor` 分别用于拟合分类问题和回归问题的线性模型，可使用不同的（凸）损失函数，支持不同的罚项。 例如，设定 `loss="log"` ，则 `SGDClassifier` 拟合一个逻辑斯蒂回归模型，而 `loss="hinge"` 拟合线性支持向量机（SVM）。

> 参考资料
>
> - 随机梯度下降

1.1.13. Perceptron（感知器）

`Perceptron` 是适用于大规模学习的一种简单算法。默认情况下：

- 不需要设置学习率（learning rate）。
- 不需要正则化处理。
- 仅使用错误样本更新模型。

最后一点表明使用合页损失（hinge loss）的感知机比 SGD 略快，所得模型更稀疏。

1.1.14. Passive Aggressive Algorithms（被动攻击算法）

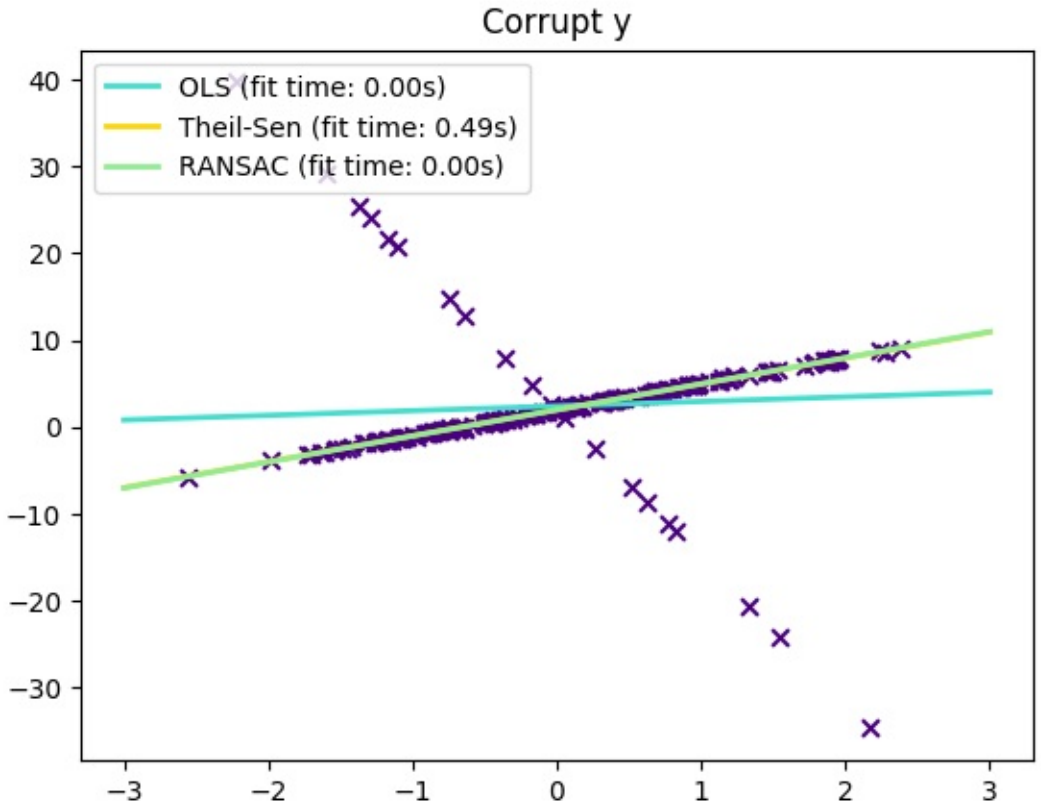
被动攻击算法是大规模学习的一类算法。和感知机类似，它也不需要设置学习率，不过比感知机多出一个正则化参数 `C` 。

对于分类问题， `PassiveAggressiveClassifier` 可设定 `loss='hinge'` （PA-I）或 `loss='squared_hinge'` （PA-II）。对于回归问题， `PassiveAggressiveRegressor` 可设置 `loss='epsilon_insensitive'` （PA-I）或 `loss='squared_epsilon_insensitive'` （PA-II）。

> 参考资料:
>
> - “Online Passive-Aggressive Algorithms” K. Crammer, O. Dekel, J. Keshat, S. Shalev-Shwartz, Y. Singer - JMLR 7 (2006)

1.1.15. 稳健回归（Robustness regression）：处理离群点（outliers）和模型错误

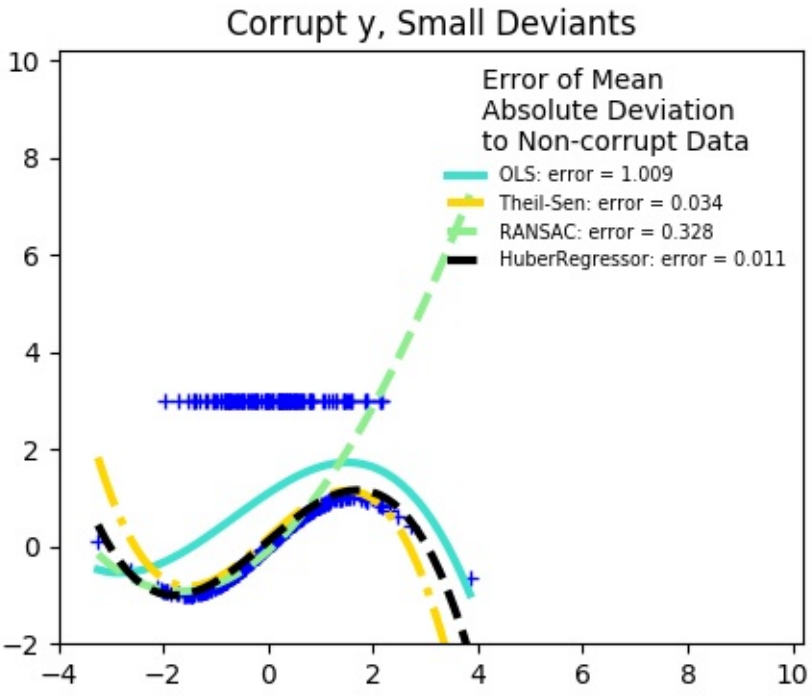
稳健回归（robust regression）特别适用于回归模型包含损坏数据（corrupt data）的情况，如离群点或模型中的错误。



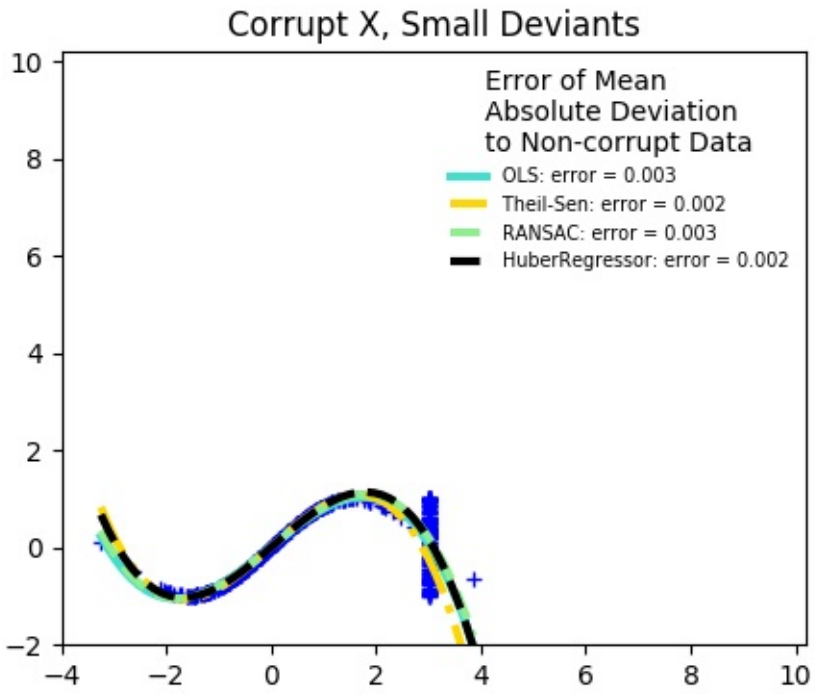
1.1.15.1. 各种使用场景与相关概念

处理包含离群点的数据时牢记以下几点：

- 离群值在 X 上还是在 y 方向上？



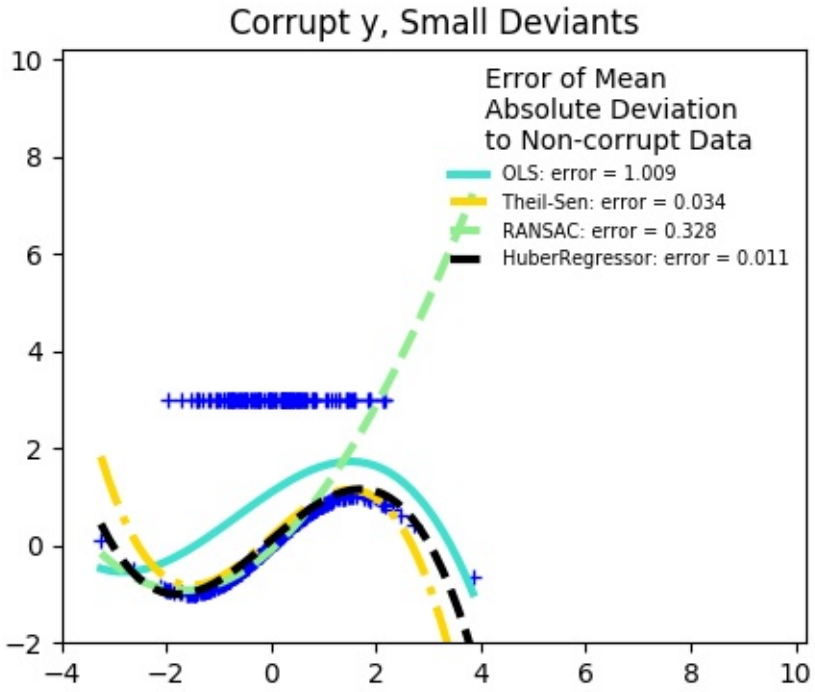
  - 离群值在 y 方向上



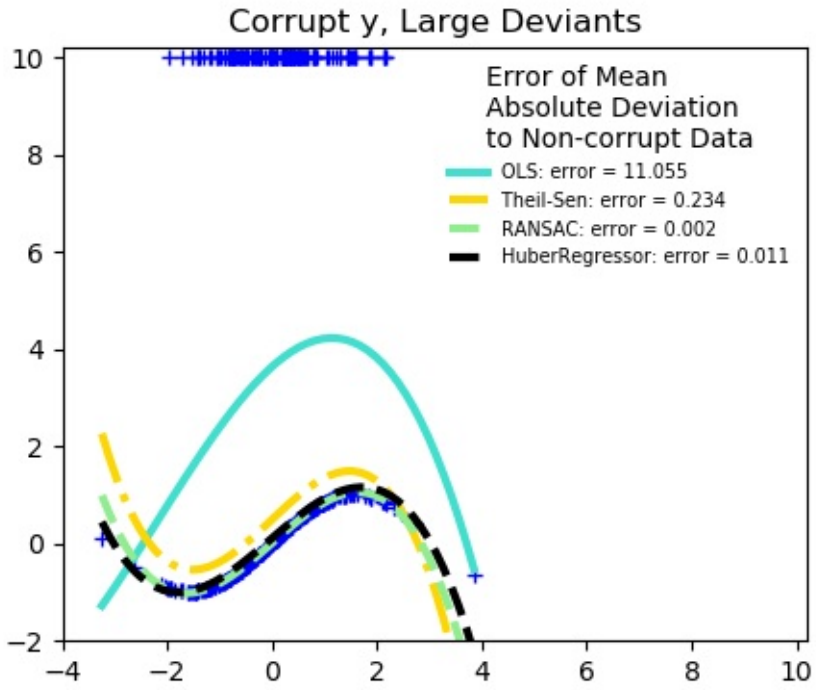
    - 离群值在 X 方向上

- 离群点的比例 **vs.** 错误的量级（**amplitude**）

离群点的数量很重要，离群程度也同样重要。



- 低离群点的数量



- 高离群点的数量

稳健拟合（robust fitting）的一个重要概念是崩溃点（breakdown point），即拟合模型（仍准确预测）所能承受的离群值最大比例。

注意，在高维数据条件下（ `n_features` 大），一般而言很难完成稳健拟合，很可能完全不起作用。

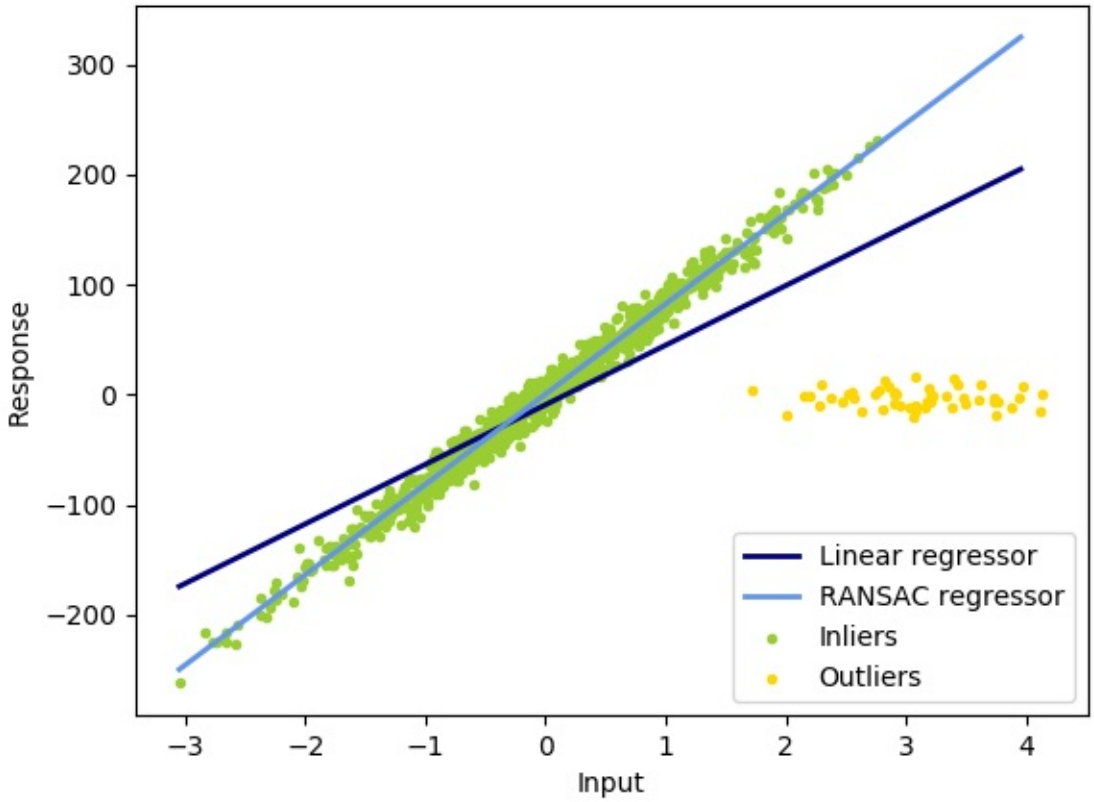
> 寻找平衡： 预测器的选择
>
> - Scikit-learn提供了三种稳健回归的预测器（estimator）： RANSAC ， Theil Sen 和 HuberRegressor
> - HuberRegressor 一般快于 RANSAC 和 Theil Sen ，除非样本数很大，即 `n_samples` >> `n_features` 。 这是因为 RANSAC 和 Theil Sen 都是基于数据的较小子集进行拟合。但使用默认参数时， Theil Sen 和 RANSAC 可能不如 HuberRegressor 鲁棒。
> - RANSAC 比 Theil Sen 更快，在样本数量上的伸缩性（适应性）更好。
> - RANSAC 能更好地处理y方向的大值离群点（通常情况下）。
> - Theil Sen 能更好地处理x方向中等大小的离群点，但在高维情况下无法保证这一特点。实在决定不了的话，请使用 RANSAC

1.1.15.2. RANSAC： 随机抽样一致性算法（RANdom SAmple Consensus）

随机抽样一致性算法（RANdom SAmple Consensus， RANSAC）利用全体数据中局内点（inliers）的一个随机子集拟合模型。

RANSAC 是一种非确定性算法，以一定概率输出一个可能的合理结果，依赖于迭代次数（参数 `max_trials` ）。这种算法主要解决线性或非线性回归问题，在计算机视觉摄影测绘领域尤为流行。

算法从全体样本输入中分出一个局内点集合，全体样本可能由于测量错误或对数据的假设错误而含有噪点、离群点。最终的模型仅从这个局内点集合中得出。



1.1.15.2.1. 算法细节

每轮迭代执行以下步骤：

- 从原始数据中抽样 `min_samples` 数量的随机样本，检查数据是否合法（见 `is_data_valid` ）。
- 用一个随机子集拟合模型（ `base_estimator.fit` ）。检查模型是否合法（见 `is_model_valid` ）。
- 计算预测模型的残差（`residual`），将全体数据分成局内点和离群点（ `base_estimator.predict(X) - y` ）。绝对残差小于 `residual_threshold` 的全体数据认为是局内点。
- 若局内点样本数最大，保存当前模型为最佳模型。以免当前模型离群点数量恰好相等（而出现未定义情况），规定仅当数值大于当前最值时认为是最佳模型。上述步骤或者迭代到最大次数（ `max_trials` ），或者某些终止条件满足时停下（见 `stop_n_inliers` 和 `stop_score` ）。最终模型由之前确定的最佳模型的局内点样本（一致性集合，`consensus set`）预测。

函数 `is_data_valid` 和 `is_model_valid` 可以识别出随机样本子集中的退化组合（`degenerate combinations`）并予以丢弃（`reject`）。即便不需要考虑退化情况，也会使用 `is_data_valid` ，因为在拟合模型之前调用它能得到更高的计算性能。

> 示例：
>
> - 基于RANSAC的稳健线性模型估计
> - 稳健线性估计拟合
>
> 参考资料：
>
> - https://en.wikipedia.org/wiki/RANSAC
> - “Random Sample Consensus: A Paradigm for Model Fitting with Applications to Image Analysis and Automated Cartography” Martin A. Fischler and Robert C. Bolles - SRI International (1981)

- “Performance Evaluation of RANSAC Family” Sunglok Choi, Taemin Kim and Wonpil Yu - BMVC (2009)

1.1.15.3. Theil-Sen 预估器：广义中值估计器（generalized-median-based estimator）

`TheilSenRegressor` 估计器：使用中位数在多个维度泛化，对多元异常值更具有鲁棒性，但问题是，随着维数的增加，估计器的准确性在迅速下降。准确性的丢失，导致在高维上的估计值比不上普通的最小二乘法。

示例：

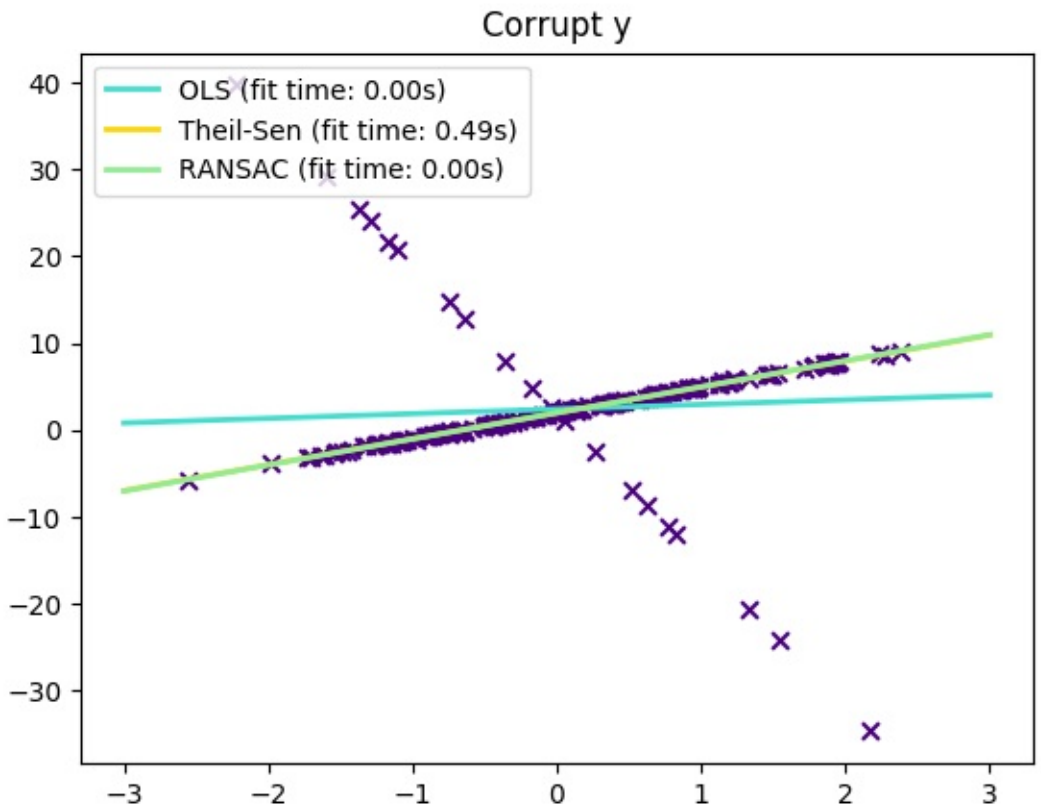
- 广义中值估计器回归
- 稳健线性估计拟合

参考资料：

- https://en.wikipedia.org/wiki/Theil%E2%80%93Sen_estimator

1.1.15.3.1. 算法理论细节

`TheilSenRegressor` 在渐近效率和无偏估计方面足以媲美 `Ordinary Least Squares (OLS)` （普通最小二乘法（OLS））。与 OLS 不同的是， Theil-Sen 是一种非参数方法，这意味着它没有对底层数据的分布假设。由于 Theil-Sen 是基于中值的估计，它更适合于损坏的数据即离群值。 在单变量的设置中，Theil-Sen 在简单的线性回归的情况下，其崩溃点大约 29.3% ，这意味着它可以容忍任意损坏的数据高达 29.3% 。



scikit-learn 中实现的 `TheilSenRegressor` 是多元线性回归模型的推广 [8] ，利用了空间中值方法，它是多维中值的推广 [9] 。

关于时间复杂度和空间复杂度，Theil-Sen 的尺度根据

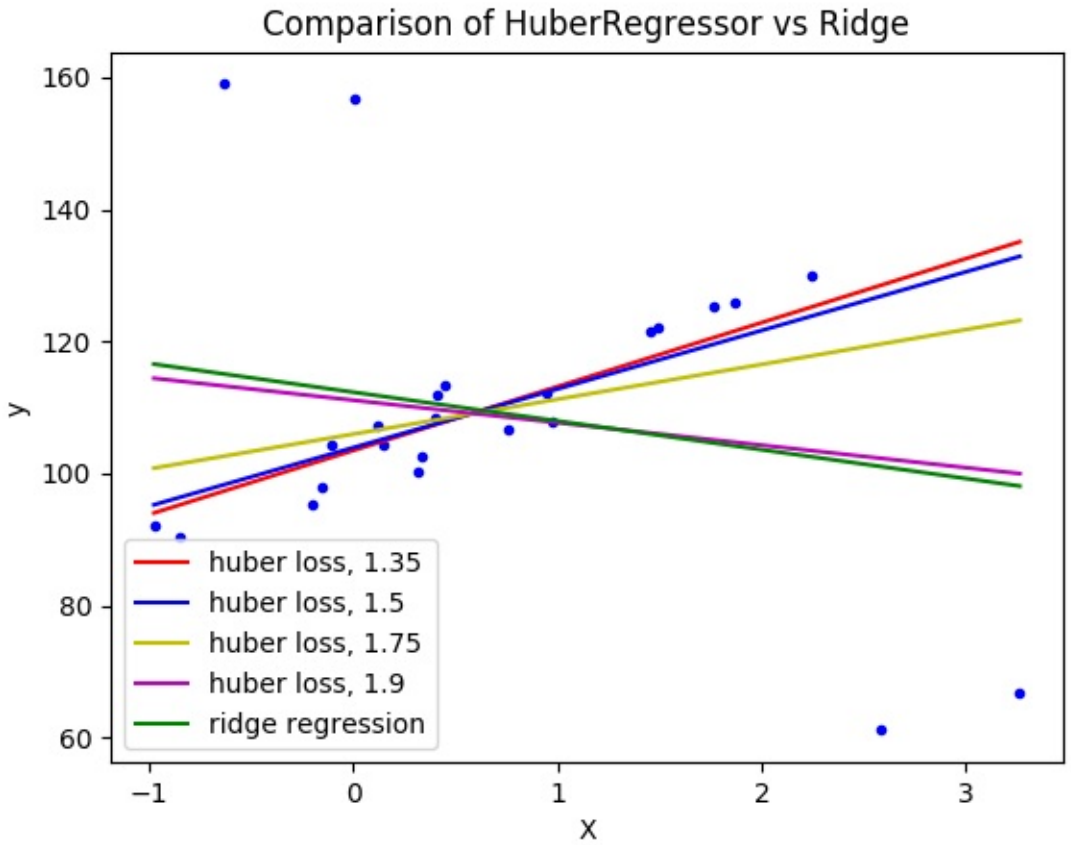
$$\binom{n_{samples}}{n_{subsamples}}$$

这使得它不适用于大量样本和特征的问题。因此，可以选择一个亚群的大小来限制时间和空间复杂度，只考虑所有可能组合的随机子集。

> 示例：
>
> - 广义中值估计器回归
>
> 参考资料：
>
> - [10] Xin Dang, Hanxiang Peng, Xueqin Wang and Heping Zhang: Theil-Sen Estimators in a Multiple Linear Regression Model. |
> - [11] Kärkkäinen and S. Äyrämö: On Computation of Spatial Median for Robust Data Mining.

1.1.15.4. Huber 回归

`HuberRegressor` 与 `Ridge` 不同，因为它对于被分为异常值的样本应用了一个线性损失。如果这个样品的绝对误差小于某一阈值，样品就被分为内围值。 它不同于 `TheilSenRegressor` 和 `RANSACRegressor` ，因为它没有忽略异常值的影响，并分配给它们较小的权重。



这个 `HuberRegressor` 最小化的损失函数是：

$$\min_{w,\sigma} \sum_{i=1}^{n} \left(\sigma + H_m \left(\frac{X_i w - y_i}{\sigma} \right) \sigma \right) + \alpha ||w||_2^2$$

其中

$$H_m(z) = \begin{cases} z^2, & \text{if } |z| < \epsilon, \\ 2\epsilon|z| - \epsilon^2, & \text{otherwise} \end{cases}$$

建议设置参数 `epsilon` 为 1.35 以实现 95% 统计效率。

1.1.15.5. 注意

`HuberRegressor` 与将损失设置为 `huber` 的 `SGDRegressor` 并不相同，体现在以下方面的使用方式上。

- `HuberRegressor` 是标度不变性的. 一旦设置了 `epsilon` ， 通过不同的值向上或向下缩放 `X` 和 `y` ，就会跟以前一样对异常值产生同样的鲁棒性。相比 `SGDRegressor` 其中 `epsilon` 在 `X` 和 `y` 被缩放的时候必须再次设置。
- `HuberRegressor` 应该更有效地使用在小样本数据，同时 `SGDRegressor` 需要一些训练数据的 passes 来产生一致的鲁棒性。

> 示例：
>
> - 强异常数据集上的huberregression与 Ridge
>
> 参考资料：
>
> - Peter J. Huber, Elvezio M. Ronchetti: Robust Statistics, Concomitant scale estimates, pg 172

另外，这个估计是不同于 R 实现的 Robust Regression (http://www.ats.ucla.edu/stat/r/dae/rreg.htm) ，因为 R 实现加权最小二乘，权重考虑到每个样本并基于残差大于某一阈值的量。

1.1.16. 多项式回归：用基函数展开线性模型

机器学习中一种常见的模式，是使用线性模型训练数据的非线性函数。这种方法保持了一般快速的线性方法的性能，同时允许它们适应更广泛的数据范围。

例如，可以通过构造系数的 **polynomial features** 来扩展一个简单的线性回归。在标准线性回归的情况下，你可能有一个类似于二维数据的模型：

$$\hat{y}(w, x) = w_0 + w_1 x_1 + w_2 x_2$$

如果我们想把抛物面拟合成数据而不是平面，我们可以结合二阶多项式的特征，使模型看起来像这样：

$$\hat{y}(w, x) = w_0 + w_1 x_1 + w_2 x_2 + w_3 x_1 x_2 + w_4 x_1^2 + w_5 x_2^2$$

观察到这 还是一个线性模型 （这有时候是令人惊讶的）： 看到这个，想象创造一个新的变量

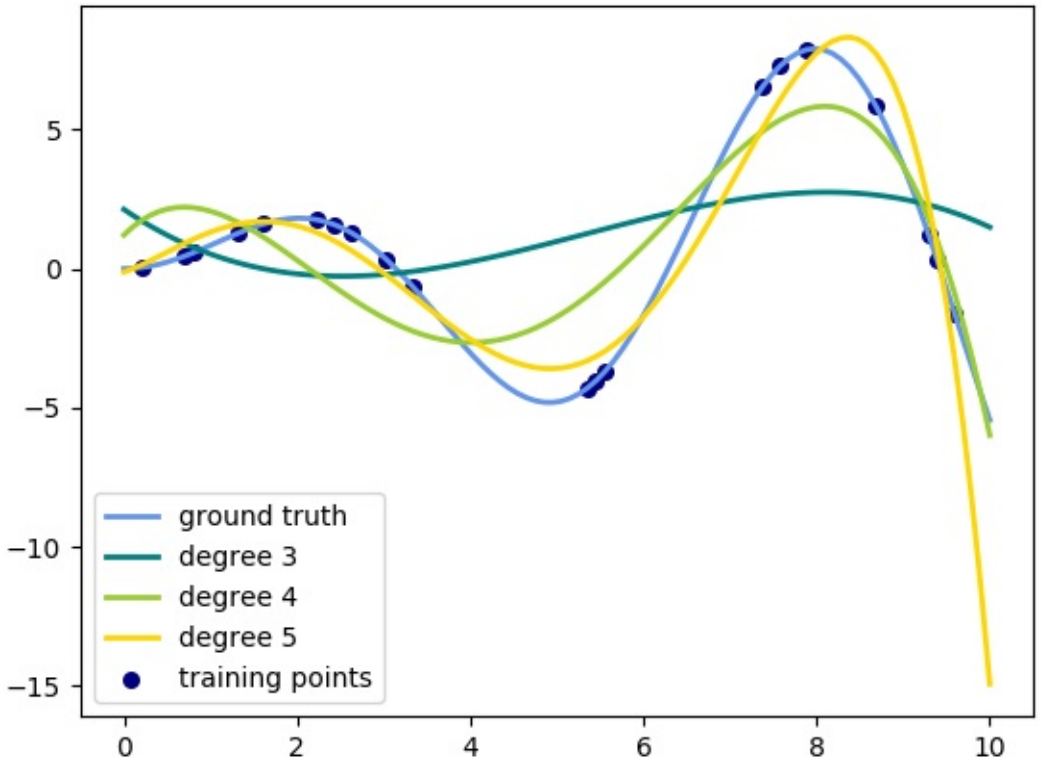
$$z = [x_1, x_2, x_1 x_2, x_1^2, x_2^2]$$

有了这些重新标记的数据，我们可以将问题写成

$$\hat{y}(w, x) = w_0 + w_1 z_1 + w_2 z_2 + w_3 z_3 + w_4 z_4 + w_5 z_5$$

我们看到，所得的 *polynomial regression* 与我们上文所述线性模型是同一类（即关于 w 是线性的），因此可以用同样的方法解决。通过用这些基函数建立的高维空间中的线性拟合，该模型具有灵活性，可以适应更广泛的数据范围。

这里是一个例子，使用不同程度的多项式特征将这个想法应用于一维数据：



这个图是使用 `PolynomialFeatures` 预创建。该预处理器将输入数据矩阵转换为给定度的新数据矩阵。使用方法如下：

```
>>> from sklearn.preprocessing import PolynomialFeatures
>>> import numpy as np
>>> X = np.arange(6).reshape(3, 2)
>>> X
array([[0, 1],
 [2, 3],
 [4, 5]])
>>> poly = PolynomialFeatures(degree=2)
>>> poly.fit_transform(X)
array([[  1.,   0.,   1.,   0.,   0.,   1.],
```

```
11.  [  1.,   2.,   3.,   4.,   6.,   9.],
12.  [  1.,   4.,   5.,  16.,  20.,  25.]])
```

`X` 的特征已经从 $[x_1, x_2]$ 转换到 $[1, x_1, x_2, x_1^2, x_1x_2, x_2^2]$，并且现在可以用在任何线性模型。

这种预处理可以通过 `Pipeline` 工具进行简化。可以创建一个表示简单多项式回归的单个对象，使用方法如下所示：

```
1.  >>> from sklearn.preprocessing import PolynomialFeatures
2.  >>> from sklearn.linear_model import LinearRegression
3.  >>> from sklearn.pipeline import Pipeline
4.  >>> import numpy as np
5.  >>> model = Pipeline([('poly', PolynomialFeatures(degree=3)),
6.  ...                   ('linear', LinearRegression(fit_intercept=False))])
7.  >>> # fit to an order-3 polynomial data
8.  >>> x = np.arange(5)
9.  >>> y = 3 - 2 * x + x ** 2 - x ** 3
10. >>> model = model.fit(x[:, np.newaxis], y)
11. >>> model.named_steps['linear'].coef_
12. array([ 3., -2.,  1., -1.])
```

利用多项式特征训练的线性模型能够准确地恢复输入多项式系数。

在某些情况下，没有必要包含任何单个特征的更高的幂，只需要相乘最多 d 个不同的特征即可，所谓 *interaction features*（交互特征）。这些可通过设定 `PolynomialFeatures` 的 `interaction_only=True` 得到。

例如，当处理布尔属性，对于所有 $n^{x_i^n} = x_i$，因此是无用的；但 $x_i x_j$ 代表两布尔结合。这样我们就可以用线性分类器解决异或问题：

```
1.  >>> from sklearn.linear_model import Perceptron
2.  >>> from sklearn.preprocessing import PolynomialFeatures
3.  >>> import numpy as np
4.  >>> X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
5.  >>> y = X[:, 0] ^ X[:, 1]
6.  >>> y
7.  array([0, 1, 1, 0])
8.  >>> X = PolynomialFeatures(interaction_only=True).fit_transform(X).astype(int)
9.  >>> X
10. array([[1, 0, 0, 0],
11.  [1, 0, 1, 0],
12.  [1, 1, 0, 0],
13.  [1, 1, 1, 1]])
14. >>> clf = Perceptron(fit_intercept=False, max_iter=10, tol=None,
15. ...                  shuffle=False).fit(X, y)
```

分类器的 “predictions” 是完美的：

```
1.  >>> clf.predict(X)
```

```
    array([0, 1, 1, 0])
    >>> clf.score(X, y)
    1.0
```

我们一直在努力

apachecn/sklearn-doc-zh

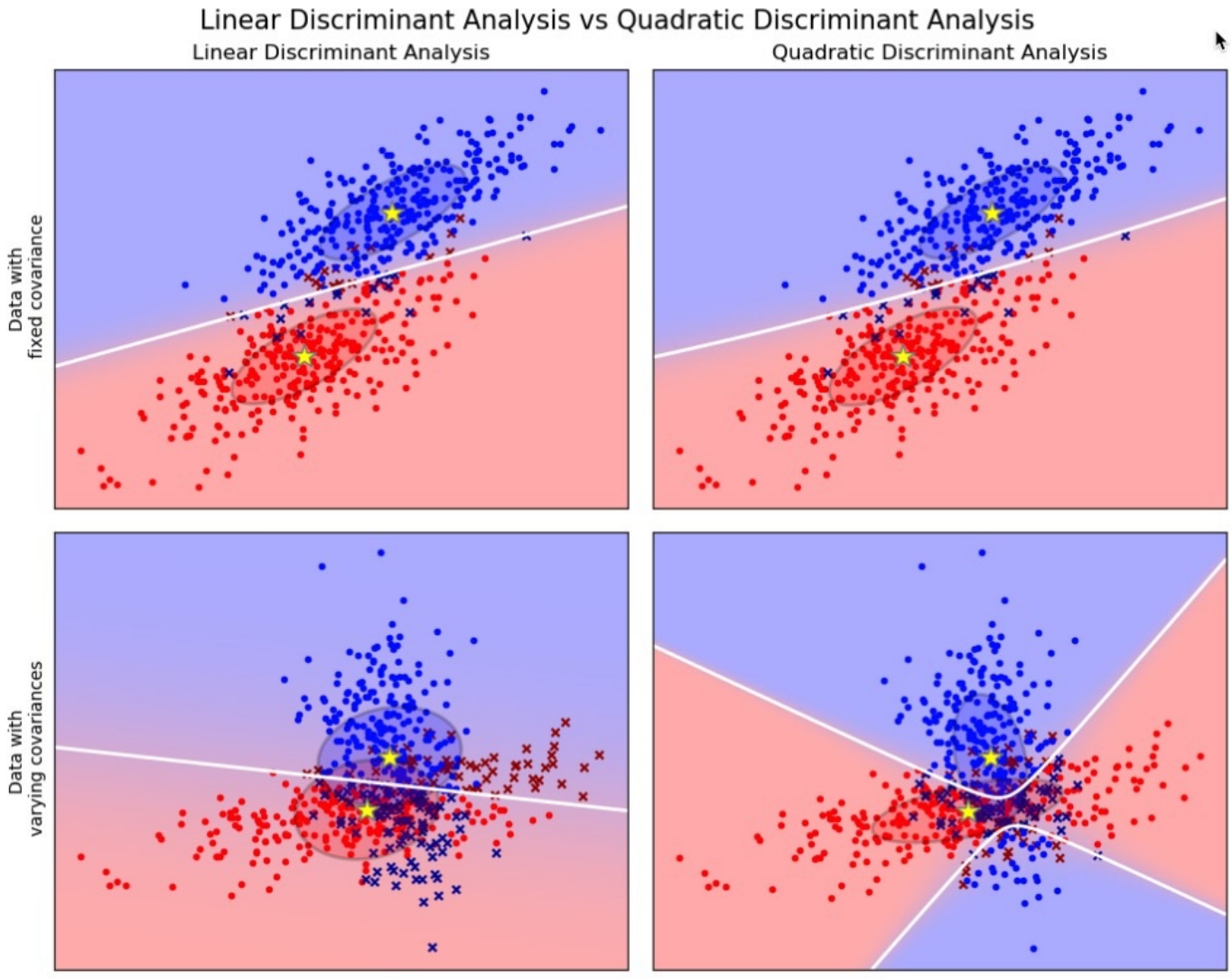


1.2. 线性和二次判别分析

校验者：@AnybodyHome@numpy@Loopy翻译者：@FAME

Linear Discriminant Analysis（线性判别分析）（`discriminant_analysis.LinearDiscriminantAnalysis`）和 Quadratic Discriminant Analysis （二次判别分析）（`discriminant_analysis.QuadraticDiscriminantAnalysis`）是两个经典的分类器。 正如他们名字所描述的那样，他们分别代表了线性决策平面和二次决策平面。

这些分类器十分具有吸引力，因为他们可以很容易计算得到闭式解(即解析解)，其天生具有多分类的特性，在实践中已经被证明很有效，并且无需调参。



以上这些图像展示了 Linear Discriminant Analysis （线性判别分析）以及 Quadratic Discriminant Analysis （二次判别分析）的决策边界。其中，最后一行表明了线性判别分析只能学习线性边界， 而二次判别分析则可以学习二次边界，因此它相对而言更加灵活。

> 示例：
>
> - Linear and Quadratic Discriminant Analysis with covariance ellipsoid: LDA和QDA在特定数据上的对比

1.2.1. 使用线性判别分析来降维

`discriminant_analysis.LinearDiscriminantAnalysis` 通过把输入的数据投影到由最大化类之间分离的方向所组成的线性子空间，可以执行有监督降维（详细的内容见下面的数学推导）。输出的维度必然会比原来的类别数量更少的。因此它总体而言是十分强大的降维方式，同样也仅仅在多分类环境下才能感觉到。

实现方式在 `discriminant_analysis.LinearDiscriminantAnalysis.transform` 中。关于维度的数量可以通过 `n_components` 参数来调节。 值得注意的是，这个参数不会对 `discriminant_analysis.LinearDiscriminantAnalysis.fit` 或者 `discriminant_analysis.LinearDiscriminantAnalysis.predict` 产生影响。

> 示例：
>
> Comparison of LDA and PCA 2D projection of Iris dataset: 在 Iris 数据集对比 LDA 和 PCA 之间的降维差异

1.2.2. LDA 和 QDA 分类器的数学公式

LDA 和 QDA 都是源于简单的概率模型，这些模型对于每一个类别 k 的相关分布 $P(X|y=k)$ 都可以通过贝叶斯定理所获得。

$$P(y=k|X) = \frac{P(X|y=k)P(y=k)}{P(X)} = \frac{P(X|y=k)P(y=k)}{\sum_l P(X|y=l)\cdot P(y=l)}$$

我们选择最大化条件概率的类别 k.

更具体地说，对于线性以及二次判别分析， $P(X|y)$ 被建模成密度多变量高斯分布：

$$p(X|y=k) = \frac{1}{(2\pi)^{d/2}|\Sigma_k|^{1/2}} \exp\left(-\frac{1}{2}(X-\mu_k)^t\Sigma_k^{-1}(X-\mu_k)\right)$$

其中的d是特征数量

为了把该模型作为分类器使用，我们只需要从训练数据中估计出类的先验概率 $P(y=k)$ （通过每个类 k 的实例的比例得到） 类别均值 μ_k （通过经验样本的类别均值得到）以及协方差矩阵（通过经验样本的类别协方差或者正则化的估计器 `estimator` 得到：见下面的 `shrinkage` 章节）。

在 LDA 中，每个类别k的高斯分布共享相同的协方差矩阵： $\Sigma_k = \Sigma$ 。这导致了两者之间的线性决策面，这可以通过比较对数概率比看出来

$$\log[P(y=k|X)/P(y=l|X)]:$$

$$\log\left(\frac{P(y=k|X)}{P(y=l|X)}\right) = log\left(\frac{P(X|y=k)P(y=k)}{P(X|y=l)P(y=l)}\right) = 0 \Leftrightarrow$$
$$(\mu_k-\mu_l)^t\Sigma^{-1}X = \frac{1}{2}(\mu_k^t\Sigma^{-1}\mu_k - \mu_l^t\Sigma^{-1}\mu_l) - log\left(\frac{P(y=k)}{P(y=l)}\right)$$

在 QDA 中，没有关于高斯协方差矩阵 Σ_k 的假设，因此有了二次决策平面. 更多细节见 参考文献[3].

> 注意:与高斯朴素贝叶斯的关系
>
> 如果在QDA模型中假设协方差矩阵是对角的，则输入被假设为在每个类中是条件独立的，所得的分类器等同于高斯朴素贝叶斯分类器 `naive_bayes.GaussianNB` 相同。

1.2.3. LDA 的降维数学公式

为了理解 LDA 在降维上的应用，从上面解释的 LDA 分类规则的几何重构开始是十分有用的。我们用 K 表示目标类别的总数。 由于在 LDA 中我们假设所有类别都有相同估计的协方差 Σ ,所以我们可重新调节数据从而让协方差相同。

$$X^* = D^{-1/2}U^tX \text{ with } \Sigma = UDU^t$$

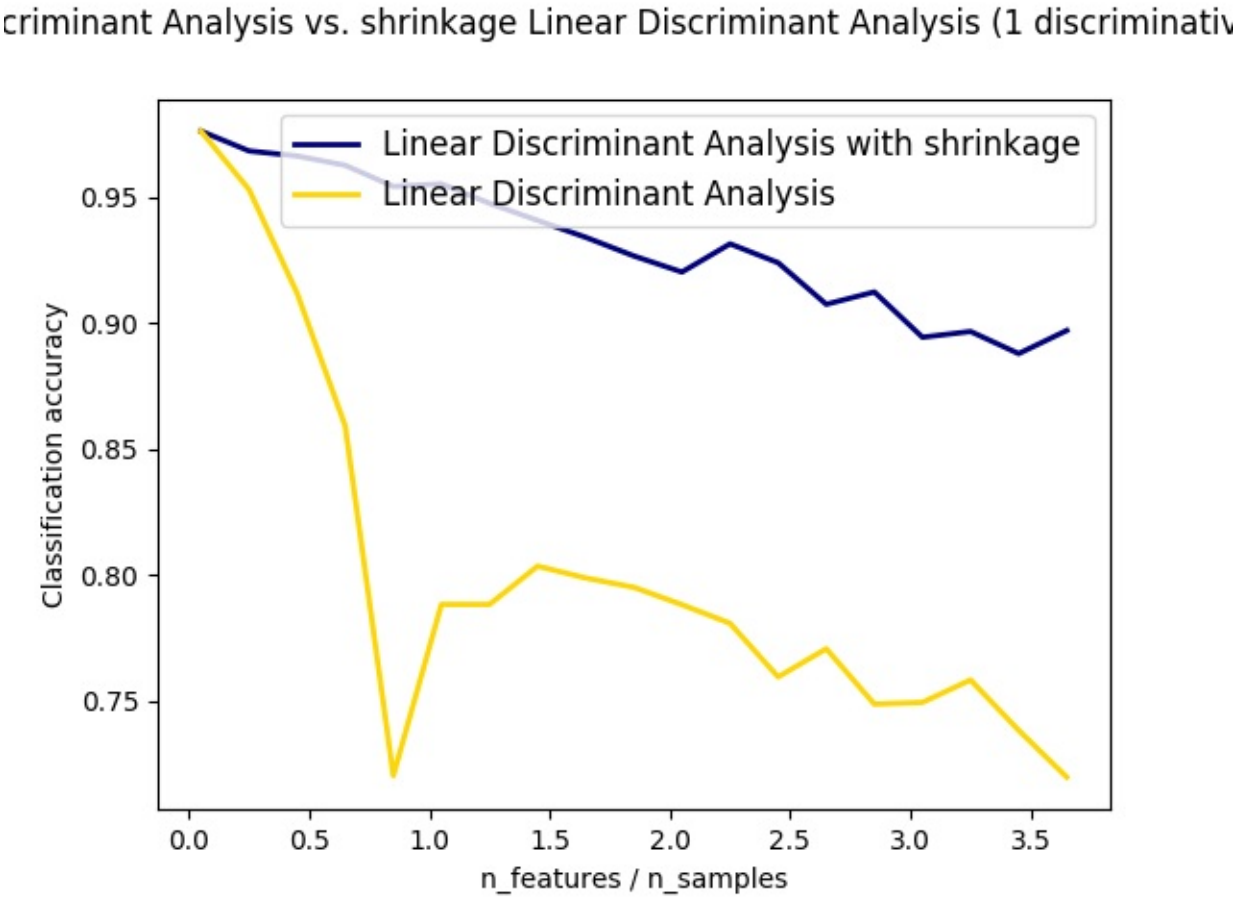
在缩放之后对数据点进行分类相当于找到与欧几里得距离中的数据点最接近的估计类别均值。但是它也可以在投影到K-1个由所有类中的所有μ_k^* 生成的仿射子空间H_K 之后完成。这也表明，LDA 分类器中存在一个利用线性投影到 $K-1$ 个维度空间的降维工具。

通过投影到线性子空间 H_L 上，我们可以进一步将维数减少到一个选定的 L ，从而使投影后的 μ_k^* 的方差最大化（实际上，为了实现转换类均值 μ_k^*，我们正在做一种形式的 PCA）。 这里的 L 对应于 `discriminant_analysis.LinearDiscriminantAnalysis.transform` 方法中使用的 `n_components` 参数。 详情参考 参考文献[3] 。

1.2.4. Shrinkage（收缩）

收缩是一种在训练样本数量相比特征而言很小的情况下可以提升的协方差矩阵预测（准确性）的工具。 在这个情况下，经验样本协方差是一个很差的预测器。收缩 LDA 可以通过设置 `discriminant_analysis.LinearDiscriminantAnalysis` 类的 `shrinkage` 参数为 ‘auto’ 来实现。

`shrinkage` parameter （收缩参数）的值同样也可以手动被设置为 0-1 之间。特别地，0 值对应着没有收缩（这意味着经验协方差矩阵将会被使用），而 1 值则对应着完全使用收缩（意味着方差的对角矩阵将被当作协方差矩阵的估计）。设置该参数在两个极端值之间会估计一个（特定的）协方差矩阵的收缩形式



1.2.5. 预估算法

默认的 solver 是 ‘svd’。它可以进行classification (分类) 以及 transform (转换),而且它不会依赖于协方差矩阵的计算（结果）。这在特征数量特别大的时候十分具有优势。然而，’svd’ solver 无法与 shrinkage （收缩）同时使用。

`lsqr` solver 则是一个高效的算法，它仅用于分类使用。它支持 shrinkage （收缩）。

`eigen` （特征） solver 是基于 class scatter （类散度）与 class scatter ratio （类内离散率）之间的优化。 它可以被用于 classification （分类）以及 transform （转换），此外它还同时支持收缩。然而，该解决方案需要计算协方差矩阵，因此它可能不适用于具有大量特征的情况。

> 示例：
>
> - Normal and Shrinkage Linear Discriminant Analysis for classification: Comparison of LDA classifiers with and without shrinkage.
>
> 参考资料：
>
> - [3] “The Elements of Statistical Learning”, Hastie T., Tibshirani R., Friedman J., Section 4.3, p.106-119, 2008.
> - [4] Ledoit O, Wolf M. Honey, I Shrunk the Sample Covariance Matrix. The Journal of Portfolio Management 30(4), 110-119, 2004.

我们一直在努力

apachecn/sklearn-doc-zh



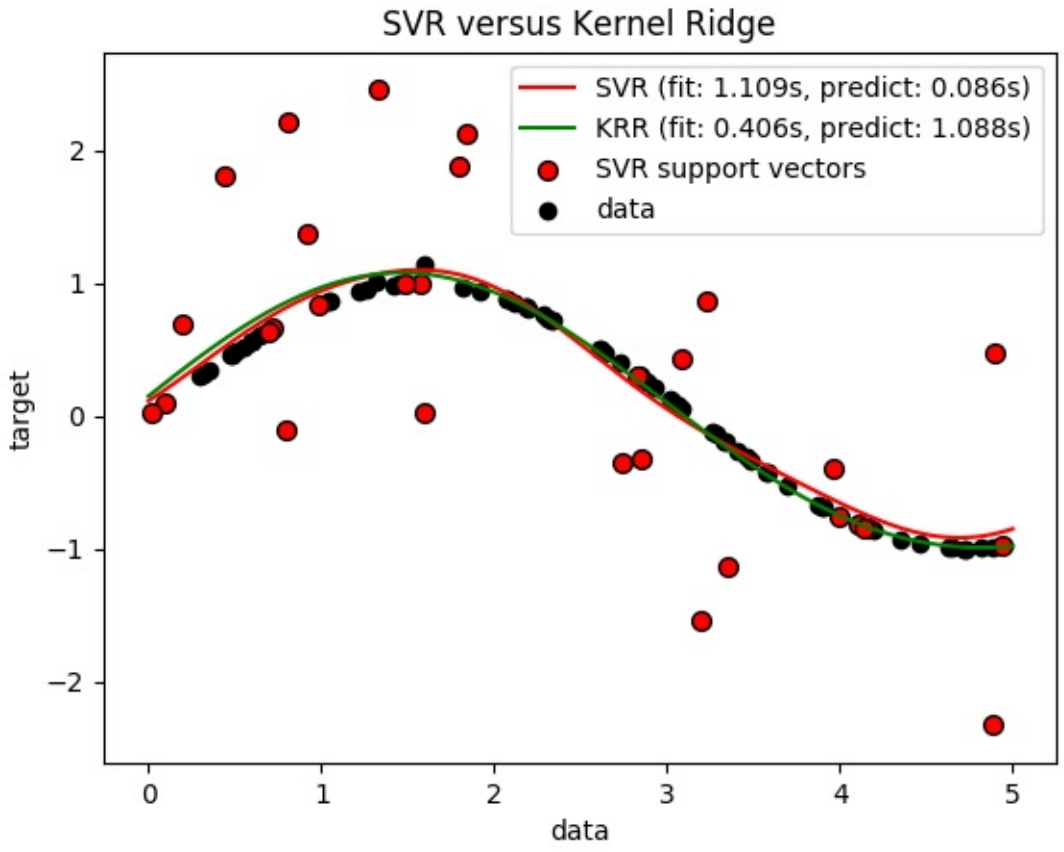
1.3. 内核岭回归

校验者：@不吃曲奇的趣多多@Loopy@qinhanmin2014翻译者：@Counting stars

内核岭回归(Kernel ridge regression-KRR)[1] 由使用内核方法的岭回归（使用 l2 正则化的最小二乘法）所组成。因此，它所拟合到的在空间中不同的线性函数是由不同的内核和数据所导致的。对于非线性的内核，它与原始空间中的非线性函数相对应。

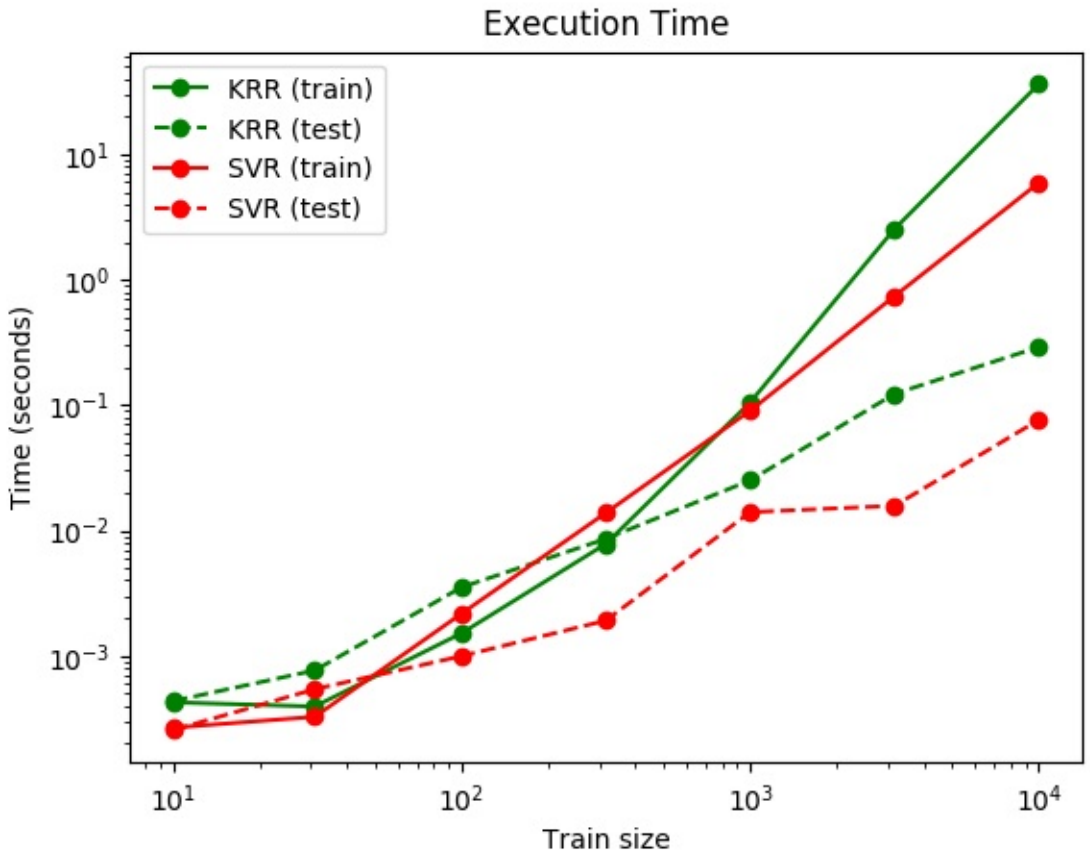
由 `KernelRidge` 学习的模型的形式与支持向量回归(`SVR` 是一样的。但是他们使用不同的损失函数：内核岭回归（KRR）使用 squared error loss （平方误差损失函数）而 support vector regression （支持向量回归）（SVR）使用 ϵ-insensitive loss （ ε-不敏感损失 ），两者都使用 l2 regularization （l2 正则化）。与 `SVR` 相反，拟合 `KernelRidge` 可以以 closed-form （封闭形式）完成，对于中型数据集通常更快。另一方面，学习的模型是非稀疏的，因此比 SVR 慢。在预测时间内，SVR 拟合的是ε>0的稀疏模型。

下图比较了人造数据集上的 `KernelRidge` 和 `SVR` 的区别，它由一个正弦目标函数和每五个数据点产生一个强噪声组成。图中分别绘制了由 `KernelRidge` 和 `SVR` 学习到的回归曲线。两者都使用网格搜索优化了 RBF 内核的 complexity/regularization （复杂性/正则化）和 bandwidth （带宽）。它们的 learned functions （学习函数）非常相似;但是，拟合 `KernelRidge` 大约比拟合 `SVR` 快七倍（都使用 grid-search （ 网格搜索 ））。然而，由于 SVR 只学习了一个稀疏模型，所以 SVR 预测 10 万个目标值比使用 KernelRidge 快三倍以上。SVR 只使用了大约1/3的数据点做为支撑向量。



下图显示不同大小训练集的 `KernelRidge` 和 `SVR` 的 fitting （拟合）和 prediction （预测）时间。 对于中型训练集（小于 1000 个样本），拟合 `KernelRidge` 比 `SVR` 快; 然而，对于更大的训练集 `SVR` 通常更

好。 关于预测时间，由于学习的稀疏解， `SVR` 对于所有不同大小的训练集都比 `KernelRidge` 快。 注意，稀疏度和预测时间取决于 `SVR` 的参数 ϵ 和 C ； $\epsilon = 0$ 将对应于密集模型。



示例：

- 内核岭回归与SVR的对比

参考：

- [1] “Machine Learning: A Probabilistic Perspective” Murphy, K. P. - chapter 14.4.3, pp. 492-493, The MIT Press, 2012

我们一直在努力

apachecn/sklearn-doc-zh



1.4. 支持向量机

校验者：@尔了个达@维@子浪@小瑶@Loopy@qinhanmin2014翻译者：@Damon@Leon晋

支持向量机 **(SVMs)** 可用于以下监督学习算法：分类，回归 和 异常检测.

支持向量机的优势在于：

- 在高维空间中非常高效.
- 即使在数据维度比样本数量大的情况下仍然有效.
- 在决策函数（称为支持向量）中使用训练集的子集，因此它也是高效利用内存的.
- 通用性：不同的核函数 核函数 与特定的决策函数一一对应.常见的 kernel 已经提供，也可以指定定制的内核.

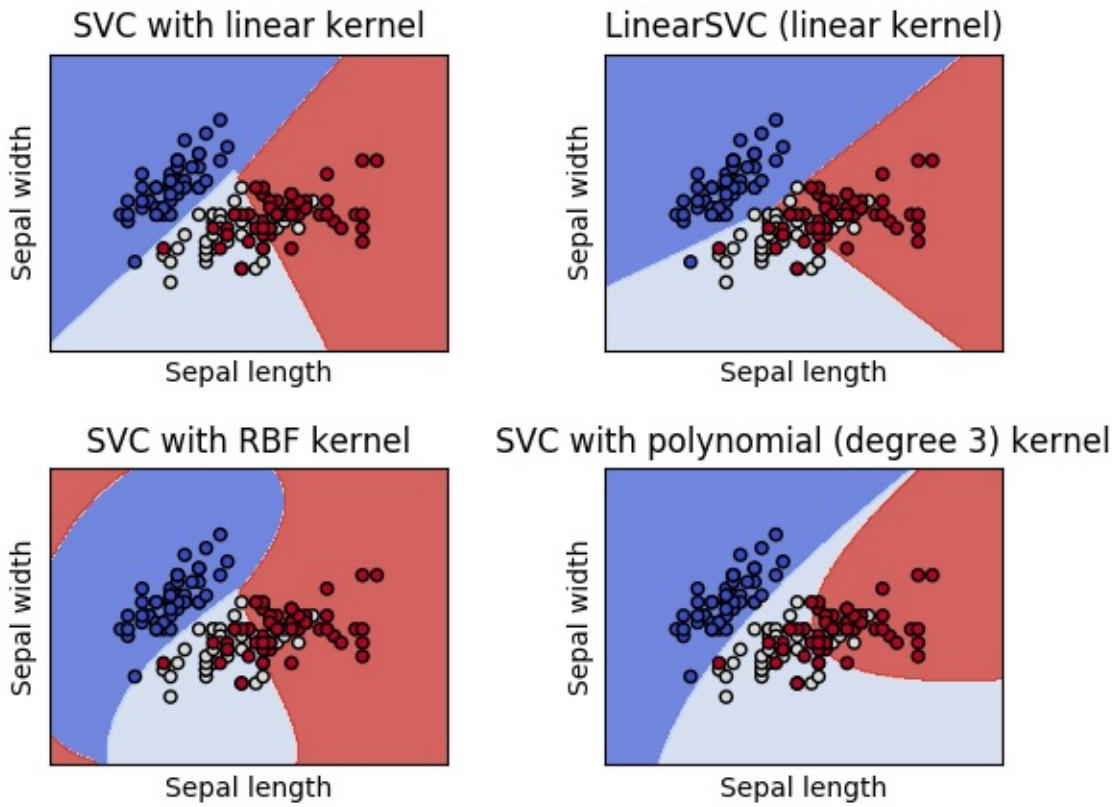
支持向量机的缺点包括：

- 如果特征数量比样本数量大得多，在选择核函数 核函数 时要避免过拟合，而且正则化项是非常重要的.
- 支持向量机不直接提供概率估计，这些都是使用昂贵的五次交叉验算计算的.（详情见 得分和概率）.

在 scikit-learn 中，支持向量机提供 dense(`numpy.ndarray` ，可以通过 `numpy.asarray` 进行转换）和 sparse（任何 `scipy.sparse` ）样例向量作为输出.然而，要使用支持向量机来对 sparse 数据作预测，它必须已经拟合这样的数据.使用行优先存储（C-order）的 `numpy.ndarray` （dense）或者带有 `dtype=float64` 的 `scipy.sparse.csr_matrix` （sparse）来优化性能.

1.4.1. 分类

`SVC` ， `NuSVC` 和 `LinearSVC` 能在数据集中实现多元分类.



`SVC` 和 `NuSVC` 是相似的方法，但是接受稍许不同的参数设置并且有不同的数学方程(在这部分看 数学公式)．另一方面，`LinearSVC` 是另一个实现线性核函数的支持向量分类．记住 `LinearSVC` 不接受关键词 `kernel`，因为它被假设为线性的．它也缺少一些 `SVC` 和 `NuSVC` 的成员(members) 比如 `support_` ．

和其他分类器一样，`SVC`，`NuSVC` 和 `LinearSVC` 将两个数组作为输入：`[n_samples, n_features]` 大小的数组 X 作为训练样本，`[n_samples]` 大小的数组 y 作为类别标签(字符串或者整数)：

```
>>> from sklearn import svm
>>> X = [[0, 0], [1, 1]]
>>> y = [0, 1]
>>> clf = svm.SVC(gamma='scale')
>>> clf.fit(X, y)
SVC(C=1.0, cache_size=200, class_weight=None, coef0=0.0,
 decision_function_shape='ovr', degree=3, gamma='scale', kernel='rbf',
 max_iter=-1, probability=False, random_state=None, shrinking=True,
 tol=0.001, verbose=False)
```

在拟合后，这个模型可以用来预测新的值：

```
>>> clf.predict([[2., 2.]])
array([1])
```

SVMs 决策函数取决于训练集的一些子集，称作支持向量．这些支持向量的部分特性可以在 `supportvectors` ，`support_` 和 `n_support` 找到：

```
>>> # 获得支持向量
>>> clf.support_vectors_
array([[ 0.,  0.],
 [ 1.,  1.]])
>>> # 获得支持向量的索引
>>> clf.support_
array([0, 1]...)
>>> # 为每一个类别获得支持向量的数量
>>> clf.n_support_
array([1, 1]...)
```

1.4.1.1. 多元分类

`SVC` 和 `NuSVC` 为多元分类实现了 “one-against-one” 的方法 (Knerr et al., 1990) 如果 `n_class` 是类别的数量, 那么 `n_class * (n_class - 1) / 2` 分类器被重构, 而且每一个从两个类别中训练数据. 为了提供与其他分类器一致的接口, `decision_function_shape` 选项允许聚合 “one-against-one” 分类器的结果成 `(n_samples, n_classes)` 的大小到决策函数:

```
>>> X = [[0], [1], [2], [3]]
>>> Y = [0, 1, 2, 3]
>>> clf = svm.SVC(gamma='scale', decision_function_shape='ovo')
>>> clf.fit(X, Y)
SVC(C=1.0, cache_size=200, class_weight=None, coef0=0.0,
 decision_function_shape='ovo', degree=3, gamma='scale', kernel='rbf',
 max_iter=-1, probability=False, random_state=None, shrinking=True,
 tol=0.001, verbose=False)
>>> dec = clf.decision_function([[1]])
>>> dec.shape[1] # 4 classes: 4*3/2 = 6
6
>>> clf.decision_function_shape = "ovr"
>>> dec = clf.decision_function([[1]])
>>> dec.shape[1] # 4 classes
4
```

另一方面, `LinearSVC` 实现 “one-vs-the-rest” 多类别策略, 从而训练 n 类别的模型. 如果只有两类, 只训练一个模型.:

```
>>> lin_clf = svm.LinearSVC()
>>> lin_clf.fit(X, Y)
LinearSVC(C=1.0, class_weight=None, dual=True, fit_intercept=True,
 intercept_scaling=1, loss='squared_hinge', max_iter=1000,
 multi_class='ovr', penalty='l2', random_state=None, tol=0.0001,
 verbose=0)
>>> dec = lin_clf.decision_function([[1]])
>>> dec.shape[1]
4
```

参见 数学公式 查看决策函数的完整描述.

记住 `LinearSVC` 也实现了可选择的多类别策略，通过使用选项 `multi_class='crammer_singer'` ，所谓的多元 SVM 由 Crammer 和 Singer 明确表达. 这个方法是一致的，对于 one-vs-rest 是不正确的. 实际上，one-vs-rest 分类通常受到青睐，因为结果大多数是相似的，但是运行时间却显著减少.

对于 “one-vs-rest” `LinearSVC` ，属性 `coef` 和 `intercept` 分别具有 `[n_class, n_features]` 和 `[n_class]` 尺寸. 系数的每一行符合 `n_class` 的许多 one-vs-rest 分类器之一，并且就以这一类的顺序与拦截器(intercepts)相似.

至于 one-vs-one `SVC` ，属性特征的布局(layout)有少多些复杂. 考虑到有一种线性核函数， `coef` 和 `intercept` 的布局(layout)与上文描述成 `LinearSVC` 相似，除了 `coef_` 的形状 `[n_class * (n_class - 1) / 2, n_features]` ，与许多二元的分类器相似. 0到n的类别顺序是 “0 vs 1”，“0 vs 2” ，… “0 vs n”，“1 vs 2”，“1 vs 3”，“1 vs n”，… “n-1 vs n”.

`dualcoef` 的形状是 `[n_class-1, n_SV]` ，这个结构有些难以理解. 对应于支持向量的列与 `n_class * (n_class - 1) / 2` “one-vs-one” 分类器相关. 每一个支持向量用于 `n_class - 1` 分类器中.对于这些分类器,每一行的 `n_class - 1` 条目对应于对偶系数(dual coefficients).

通过这个例子更容易说明:

考虑一个三类的问题,类0有三个支持向量 v_0^0, v_0^1, v_0^2 而类 1 和 2 分别有 如下两个支持向量 v_1^0, v_1^1 and v_2^0, v_2^1 .对于每个支持 向量 v_i^j ，有两个对偶系数.在类别 i 和 k $\alpha_{i,k}^j$ 中，我们将支持向量的系数记录为 v_i^j 那么 `dualcoef` 可以表示为:

$\alpha_{0,1}^0$	$\alpha_{0,2}^0$	第0类的SVs系数
$\alpha_{0,1}^1$	$\alpha_{0,2}^1$	
$\alpha_{0,1}^2$	$\alpha_{0,2}^2$	
$\alpha_{1,0}^0$	$\alpha_{1,2}^0$	第1类的SVs系数
$\alpha_{1,0}^1$	$\alpha_{1,2}^1$	
$\alpha_{2,0}^0$	$\alpha_{2,1}^0$	第2类的SVs系数
$\alpha_{2,0}^1$	$\alpha_{2,1}^1$	

(空单元格表示与上方单元格合并)

1.4.1.2. 得分和概率

`SVC` 方法的 `decision_function` 给每一个样例每一个类别分值(scores)(或者在一个二元类中每一个样例一个分值). 当构造器(constructor)选项 `probability` 设置为 `True` 的时候，类成员可能性评估开启.(来自

`predict_proba` 和 `predict_log_proba` 方法）在二元分类中，概率使用 Platt scaling 进行标准化：在 SVM 分数上的逻辑回归，在训练集上用额外的交叉验证来拟合。在多类情况下，这可以扩展为 per Wu et al.(2004)

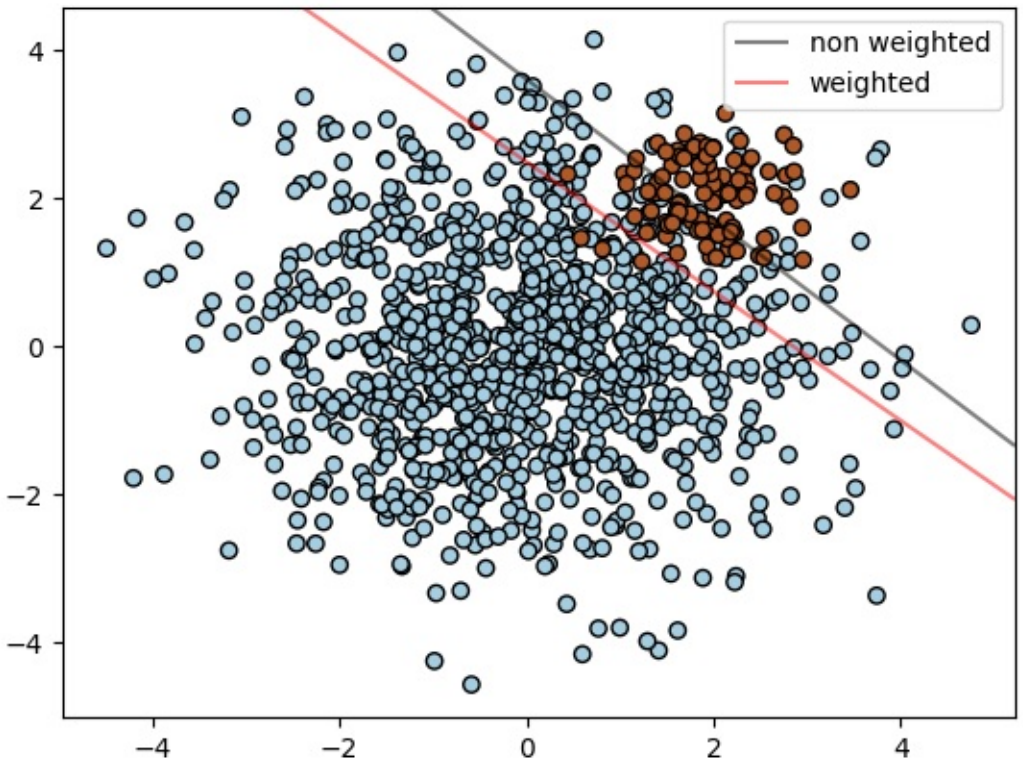
不用说，对于大数据集来说，在 Platt scaling 中进行交叉验证是一项昂贵的操作。另外，可能性预测可能与 scores 不一致，因为 scores 的“argmax”可能不是可能性的 argmax。（例如，在二元分类中，一个样本可能被标记为一个有可能性的类 `predict` <½ according to `predict_proba` .）Platt 的方法也有理论问题。如果 confidence scores 必要，但是这些没必要是可能性，那么建议设置 `probability=False` 并使用 `decision_function` 而不是 `predict_proba` .

> 参考资料：
>
> - Wu, Lin and Weng, "Probability estimates for multi-class classification by pairwise coupling（成对耦合的多类分类的概率估计）"<http://www.csie.ntu.edu.tw/~cjlin/papers/svmprob/svmprob.pdf> , *JMLR 5:975-1005, 2004.*
> - *Platt "Probabilistic outputs for SVMs and comparisons to regularized likelihood methods（SVMs 的概率输出和与规则化似然方法的比较）"<http://www.cs.colorado.edu/~mozer/Teaching/syllabi/6622/papers/Platt1999.pdf> .*

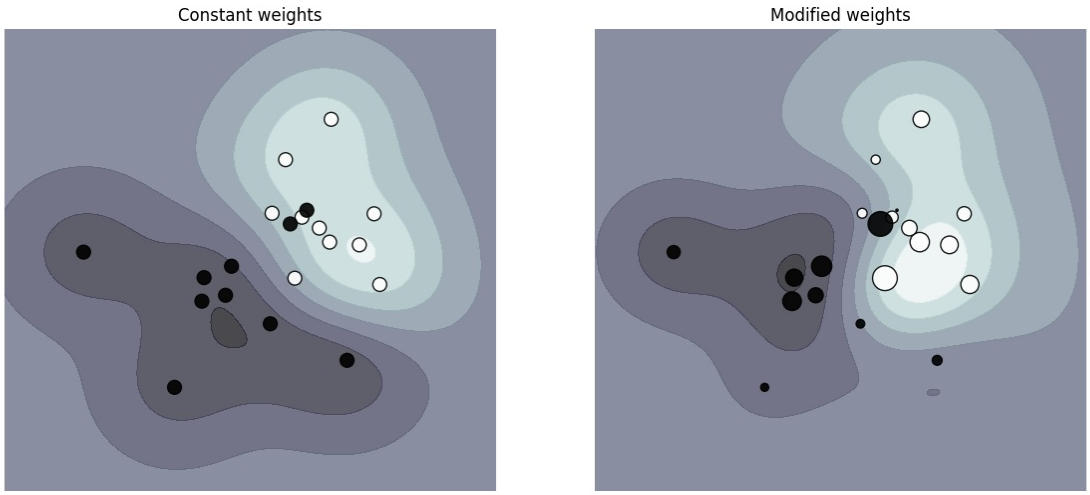
1.4.1.3. 非均衡问题

这个问题期望给予某一类或某个别样例能使用的关键词 `class_weight` 和 `sample_weight` 提高权重 (importance).

`SVC` （而不是 `NuSVC` ）在 `fit` 方法中生成了一个关键词 `class_weight` . 它是形如 `{class_label : value}` 的字典，value 是浮点数大于 0 的值，把类 `class_label` 的参数 `C` 设置为 `C * value` .



`SVC` , `NuSVC` , `SVR` , `NuSVR` 和 `OneClassSVM` 在 `fit` 方法中通过关键词 `sample_weight` 为单一样例实现权重weights.与 `class_weight` 相似，这些把第i个样例的参数 `C` 换成 `C * sample_weight[i]` .



示例：

- Plot different SVM classifiers in the iris dataset,
- SVM: Maximum margin separating hyperplane,
- SVM: Separating hyperplane for unbalanced classes
- SVM-Anova: SVM with univariate feature selection,
- Non-linear SVM
- SVM: Weighted samples,

1.4.2. 回归

支持向量分类的方法可以被扩展用作解决回归问题. 这个方法被称作支持向量回归.

支持向量分类生成的模型(如前描述)只依赖于训练集的子集,因为构建模型的 cost function 不在乎边缘之外的训练点. 类似的,支持向量回归生成的模型只依赖于训练集的子集, 因为构建模型的 cost function 忽略任何接近于模型预测的训练数据.

支持向量分类有三种不同的实现形式: `SVR` , `NuSVR` 和 `LinearSVR` . 在只考虑线性核的情况下, `LinearSVR` 比 `SVR` 提供一个更快的实现形式, 然而比起 `SVR` 和 `LinearSVR` , `NuSVR` 实现一个稍微不同的构思(formulation).细节参见 实现细节.

与分类的类别一样, fit方法会调用参数向量 X, y, 只在 y 是浮点数而不是整数型.:

```
>>> from sklearn import svm
>>> X = [[0, 0], [2, 2]]
>>> y = [0.5, 2.5]
>>> clf = svm.SVR()
>>> clf.fit(X, y)
SVR(C=1.0, cache_size=200, coef0=0.0, degree=3, epsilon=0.1, gamma='auto_deprecated',
  kernel='rbf', max_iter=-1, shrinking=True, tol=0.001, verbose=False)
>>> clf.predict([[1, 1]])
array([ 1.5])
```

示例:

- Support Vector Regression (SVR) using linear and non-linear kernels

1.4.3. 密度估计，异常（novelty）检测

类 `OneClassSVM` 实现了一个用于离群点检测的单类SVM。

有关OneClassSVM的描述和使用，请参见新奇和异常值检测.

1.4.4. 复杂度

支持向量机是个强大的工具，不过它的计算和存储空间要求也会随着要训练向量的数目增加而快速增加。 SVM的核心是一个二次规划问题(Quadratic Programming, QP)，是将支持向量和训练数据的其余部分分离开来。 在实践中(数据集相关)，会根据 libsvm 的缓存有多效，在 $O(n_{features} \times n_{samples}^2)$ 和 $O(n_{features} \times n_{samples}^3)$ 之间基于 libsvm 的缩放操作才会调用这个 QP 解析器。 如果数据是非常稀疏，那 $n_{features}$ 就用样本向量中非零特征的平均数量去替换。

另外请注意，在线性情况下，由 liblinear 操作的 `LinearSVC` 算法要比由它的 libsvm 对应的 `SVC` 更为高效，并且它几乎可以线性缩放到数百万样本或者特征。

1.4.5. 使用诀窍

- 避免数据复制：对于 `SVC` ， `SVR` ， `NuSVC` 和 `NuSVR` ， 如果数据是通过某些方法而不是用行优先存储（C-order）的双精度，那它会在调用底层的 C 命令前先被复制。 您可以通过检查它的 `flags` 属性，来确定给定的 numpy 数组是不是行优先存储（C-order）的。

  对于 `LinearSVC` （和 `LogisticRegression` ）的任何输入，都会以 numpy 数组形式，被复制和转换为 用 liblinear 内部稀疏数据去表达（双精度浮点型 float 和非零部分的 int32 索引）。 如果您想要一个适合大规模的线性分类器，又不打算复制一个密集的行优先存储（C-order）双精度 numpy 数组作为输入， 那我们建议您去使用 `SGDClassifier` 类作为替代。目标函数可以配置为和 `LinearSVC` 模型差不多相同的。

- 内核的缓存大小：在大规模问题上，对于 `SVC` ， `SVR` ， `nuSVC` 和 `NuSVR` ，内核缓存的大小会特别影响到运行时间。如果您有足够可用的 RAM，不妨把它的 `缓存大小` 设得比默认的 200(MB) 要高，例如为 500(MB) 或者 1000(MB)。

- 惩罚系数**C**的设置:在合理的情况下， `C` 的默认选择为 `1` 。如果您有很多混杂的观察数据， 您应该要去调小它。 `C` 越小，就能更好地去正规化估计。

当C值较大时， `LinearSVC` 和 `LinearSVR` 对C值较不敏感，即当C值大于特定阈值后，模型效果将会停止提升。同时，较大的C值将会导致较长的训练时间，Fan et al.(2008)的论文显示，训练时间的差距有时会达到10倍。

- 支持向量机算法本身不能够很好地支持非标准化的数据，所以 我们强烈建议您将数据标准化。 举个例子，对于输入向量 X， 规整它的每个数值范围为 [0, 1] 或 [-1, +1] ，或者标准化它的为均值为0方差为1的数据分布。请注意， 相同的缩放标准必须要应用到所有的测试向量，从而获得有意义的结果。 请参考章节 预处理数据 ，那里会提供到更多关于缩放和规整。

- 在 `NuSVC` / `OneClassSVM` / `NuSVR` 内的参数 `nu` ， 近似是训练误差和支持向量的比值。
- 在 `SVC` ，，如果分类器的数据不均衡（例如，很多正例很少负例），设置 `class_weight='balanced'` 与/或尝试不同的惩罚系数 `C` 。
- 底层实现的随机性： `SVC` 和 `NuSVC` 的底层实现仅使用随机数生成器来打乱数据顺序进行概率估计(当 `probability` 被设置为True时)。这种随机性可以用 `random_state` 参数来控制。如果将 `probability` 设为False，这些估计器就不是随机的，random_state对结果没有影响。底层的 `OneClassSVM` 实现类似于 `SVC` 和 `NuSVC` 的实现。由于 `OneClassSVM` 没有提供概率估计，所以它不是随机的。底层的 `LinearSVC` 实现使用随机数生成器来选择特征，当用双坐标下降(当 `dual` 被设置为True)。因此，对于相同的输入数据，结果略有不同并不罕见。如果发生这种情况，尝试使用较小的 `tol` 参数。这种随机性也可以通过 `random_state` 参数来控制。当 `dual` 设置为False时， `LinearSVC` 的底层实现不是随机的， `random_state` 对结果没有影响。
- 使用由 `LinearSVC(loss='l2', penalty='l1', dual=False)` 提供的 L1 惩罚去产生稀疏解，也就是说，特征权重的子集不同于零，这样做有助于决策函数。 随着增加 `C` 会产生一个更复杂的模型（有更多的特征被选择）。可以使用 `l1_min_c` 去计算 `C` 的数值，去产生一个"null" 模型（所有的权重等于零）。

参考资料：

- Fan, Rong-En, et al., "LIBLINEAR: A library for large linear classification.", Journal of machine learning research 9.Aug (2008): 1871-1874.

1.4.6. 核函数

核函数 可以是以下任何形式：：

- 线性： $\langle x, x' \rangle$.
- 多项式： $(\gamma \langle x, x' \rangle + r)^d$. d 是关键词 `degree` ， r 指定 `coef0` 。
- rbf： $\exp(-\gamma ||x - x'||^2)$. γ 是关键词 `gamma` ， 必须大于 0。
- sigmoid ($\tanh(\gamma \langle x, x' \rangle + r)$)，其中 r 指定 `coef0` 。

初始化时，不同内核由不同的函数名调用：

```
>>> linear_svc = svm.SVC(kernel='linear')
>>> linear_svc.kernel
'linear'
>>> rbf_svc = svm.SVC(kernel='rbf')
>>> rbf_svc.kernel
'rbf'
```

1.4.6.1. 自定义核

您可以自定义自己的核，通过使用python函数作为内核或者通过预计算 Gram 矩阵。

自定义内核的分类器和别的分类器一样，除了下面这几点：

- 空间 `supportvectors` 现在是空的，只有支持向量的索引被存储在 `support_`
- `fit()` 方法的第一个参数的引用（不是副本）将被存储，并作为将来的引用。 如果在 `fit()` 和 `predict()` 之间有数组发生改变，您将会碰到意料外的结果。

1.4.6.1.1. 使用 python 函数作为内核

在构造时，您同样可以通过一个函数传递到关键词 `kernel` ，来使用您自己定义的内核。

您的内核必须要以两个矩阵作为参数，大小分别是 `(n_samples_1, n_features)` ， `(n_samples_2, n_features)` 和返回一个内核矩阵，shape 是 `(n_samples_1, n_samples_2)` .

以下代码定义一个线性核，和构造一个使用该内核的分类器例子：

```
>>> import numpy as np
>>> from sklearn import svm
>>> def my_kernel(X, Y):
...     return np.dot(X, Y.T)
...
>>> clf = svm.SVC(kernel=my_kernel)
```

> 示例：
>
> - 自定义核的SVM

1.4.6.1.2. 使用 Gram 矩阵

在适应算法中，设置 `kernel='precomputed'` 和把 X 替换为 Gram 矩阵。 此时，必须要提供在 所有 训练矢量和测试矢量中的内核值。

```
>>> import numpy as np
>>> from sklearn import svm
>>> X = np.array([[0, 0], [1, 1]])
>>> y = [0, 1]
>>> clf = svm.SVC(kernel='precomputed')
>>> # 线性内核计算
>>> gram = np.dot(X, X.T)
>>> clf.fit(gram, y)
SVC(C=1.0, cache_size=200, class_weight=None, coef0=0.0,
 decision_function_shape='ovr', degree=3, gamma='auto_deprecated',
 kernel='precomputed', max_iter=-1, probability=False,
 random_state=None, shrinking=True, tol=0.001, verbose=False)
>>> # 预测训练样本
>>> clf.predict(gram)
array([0, 1])
```

1.4.6.1.3. RBF 内核参数

当用 径向基（RBF）内核去训练 SVM，有两个参数必须要去考虑： `C` 惩罚系数和 `gamma` 。参数 `C` ，

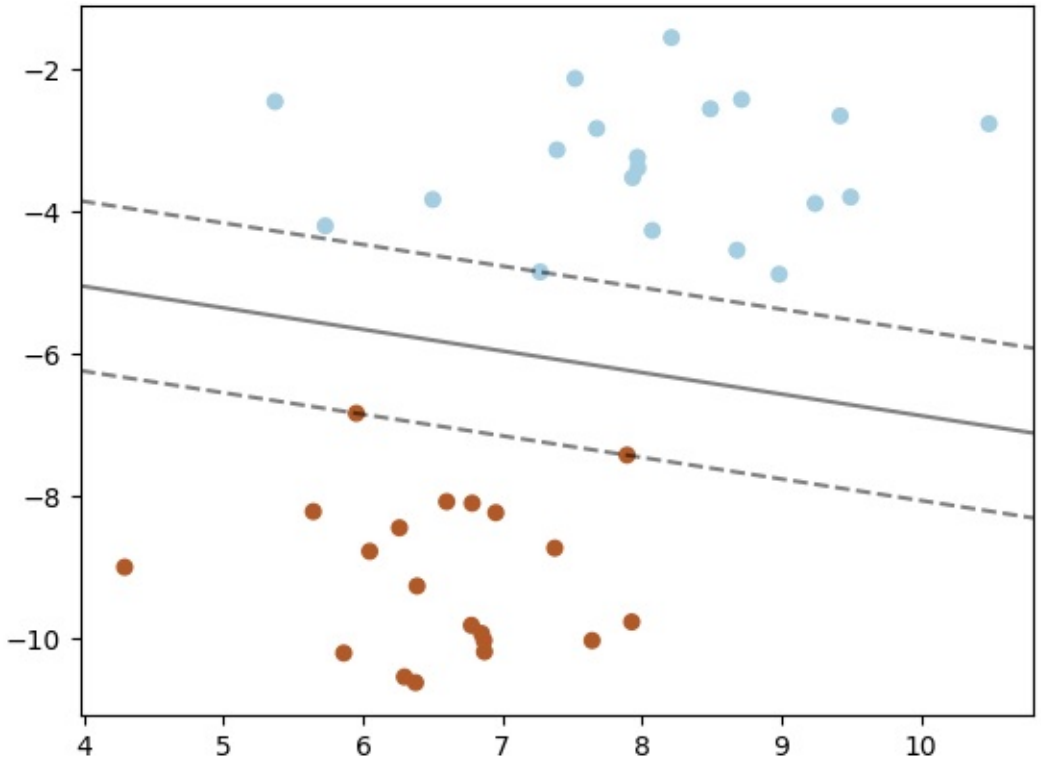
通用在所有 SVM 内核，与决策表面的简单性相抗衡，可以对训练样本的误分类进行有价转换。 较小的 `C` 会使决策表面更平滑，同时较高的 `C` 旨在正确地分类所有训练样本。 `Gamma` 定义了单一 训练样本能起到多大的影响。较大的 `gamma` 会更让其他样本受到影响。

选择合适的 `C` 和 `gamma` ，对SVM的性能起到很关键的作用。建议一点是 使用 `sklearn.model_selection.GridSearchCV` 与 `C` 和 `gamma` 相隔 成倍差距从而选择到好的数值。

> 示例：
>
> - RBF SVM parameters

1.4.7. 数学公式

支持向量机在高维度或无穷维度空间中，构建一个超平面或者一系列的超平面，可以用于分类、回归或者别的任务。直观地看，借助超平面去实现一个好的分割， 能在任意类别中使最为接近的训练数据点具有最大的间隔距离（即所谓的函数余量），这样做是因为通常更大的余量能有更低的分类器泛化误差。



1.4.7.1. SVC

在两类中，给定训练向量 $x_i \in \mathbb{R}^p$ ， i=1,…, n， 和一个向量 $y \in \{1, -1\}^n$ ， SVC能解决 如下主要问题：

$$\min_{w,b,\zeta} \frac{1}{2} w^T w + C \sum_{i=1}^{n} \zeta_i$$
$$\text{subject to } y_i(w^T \phi(x_i) + b) \geq 1 - \zeta_i,$$
$$\zeta_i \geq 0, i = 1, ..., n$$

它的对偶是

$$\min_{\alpha} \frac{1}{2} \alpha^T Q \alpha - e^T \alpha$$
$$\text{subject to } y^T \alpha = 0$$
$$0 \leq \alpha_i \leq C, i = 1, ..., n$$

其中 e 是所有的向量， $C > 0$ 是上界， Q 是一个 n 由 n 个半正定矩阵， 而 $Q_{ij} \equiv y_i y_j K(x_i, x_j)$ ，其中 $K(x_i, x_j) = \phi(x_i)^T \phi(x_j)$ 是内核。所以训练向量是通过函数 ϕ ，间接反映到一个更高维度的（无穷的）空间。

决策函数是：

$$\text{sgn}(\sum_{i=1}^{n} y_i \alpha_i K(x_i, x) + \rho)$$

> 注意：
>
> 虽然这些SVM模型是从 `libsvm` 和 `liblinear` 中派生出来，使用了 `C` 作为调整参数，但是大多数的 攻击使用了 `alpha` 。两个模型的正则化量之间的精确等价，取决于模型优化的准确目标函数。举 个例子，当使用的估计器是 `sklearn.linear_model.Ridge` 做回归时，他们之间的相关性是 $C = \frac{1}{alpha}$ 。

这些参数能通过成员 `dualcoef` 、 `supportvectors` 、 `intercept_` 去访问，这些成员分别控制了输出 $y_i \alpha_i$ 、支持向量和无关项 ρ ：

> 参考资料：
>
> - “Automatic Capacity Tuning of Very Large VC-dimension Classifiers”, I. Guyon, B. Boser, V. Vapnik - Advances in neural information processing 1993.
> - “Support-vector networks”, C. Cortes, V. Vapnik - Machine Learning, 20, 273-297 (1995).

1.4.7.2. NuSVC

我们引入一个新的参数 ν 来控制支持向量的数量和训练误差。参数 $\nu \in (0, 1]$ 是训练误差分数的上限和支持向量分数的下限。

可以看出， ν-SVC 公式是 C-SVC 的再参数化，所以数学上是等效的。

1.4.7.3. SVR

给定训练向量 $x_i \in \mathbb{R}^p$，i=1,…, n，向量 $y \in \mathbb{R}^n$ ε-SVR 能解决以下的主要问题：

$$\min_{w,b,\zeta,\zeta^*} \frac{1}{2} w^T w + C \sum_{i=1}^{n} (\zeta_i + \zeta_i^*)$$

$$\text{subject to } y_i - w^T \phi(x_i) - b \leq \varepsilon + \zeta_i,$$
$$w^T \phi(x_i) + b - y_i \leq \varepsilon + \zeta_i^*,$$
$$\zeta_i, \zeta_i^* \geq 0, i = 1, ..., n$$

它的对偶是

$$\min_{\alpha,\alpha^*} \frac{1}{2} (\alpha - \alpha^*)^T Q (\alpha - \alpha^*) + \varepsilon e^T (\alpha + \alpha^*) - y^T (\alpha - \alpha^*)$$

$$\text{subject to } e^T (\alpha - \alpha^*) = 0$$
$$0 \leq \alpha_i, \alpha_i^* \leq C, i = 1, ..., n$$

其中 e 是所有的向量，$C > 0$ 是上界，Q 是一个 n 由 n 个半正定矩阵，而 $Q_{ij} \equiv K(x_i, x_j) = \phi(x_i)^T \phi(x_j)$ 是内核。所以训练向量是通过函数 ϕ，间接反映到一个更高维度的（无穷的）空间。

决策函数是：

$$\sum_{i=1}^{n} (\alpha_i - \alpha_i^*) K(x_i, x) + \rho$$

这些参数能通过成员 `dualcoef`、`supportvectors`、`intercept_` 去访问，这些 成员分别控制了不同的 $\alpha_i - \alpha_i^*$、支持向量和无关项 ρ：

> 参考资料：
>
> - “A Tutorial on Support Vector Regression”, Alex J. Smola, Bernhard Schölkopf - Statistics and Computing archive Volume 14 Issue 3, August 2004, p. 199-222.

1.4.8. 实现细节

在底层里，我们使用 libsvm 和 liblinear 去处理所有的计算。这些库都使用了 C 和 Cython 去包装。

> 参考资料：
>
> 有关实现的描述和使用算法的细节，请参考
>
> - LIBSVM: A Library for Support Vector Machines.
> - LIBLINEAR – A Library for Large Linear Classification.

我们一直在努力

apachecn/sklearn-doc-zh



1.5. 随机梯度下降

校验者：@A@HelloSilicat@Loopy@qinhanmin2014翻译者：@L

随机梯度下降**(SGD)** 是一种简单但又非常高效的方法，主要用于凸损失函数下线性分类器的判别式学习，例如(线性) 支持向量机 和 Logistic 回归 。 尽管 SGD 在机器学习社区已经存在了很长时间，但是最近在 large-scale learning （大规模学习）方面 SGD 获得了相当大的关注。

SGD 已成功应用于在文本分类和自然语言处理中经常遇到的大规模和稀疏的机器学习问题。对于稀疏数据，本模块的分类器可以轻易的处理超过 10^5 的训练样本和超过 10^5 的特征。

Stochastic Gradient Descent （随机梯度下降法）的优势：

- 高效。
- 易于实现 (有大量优化代码的机会)。

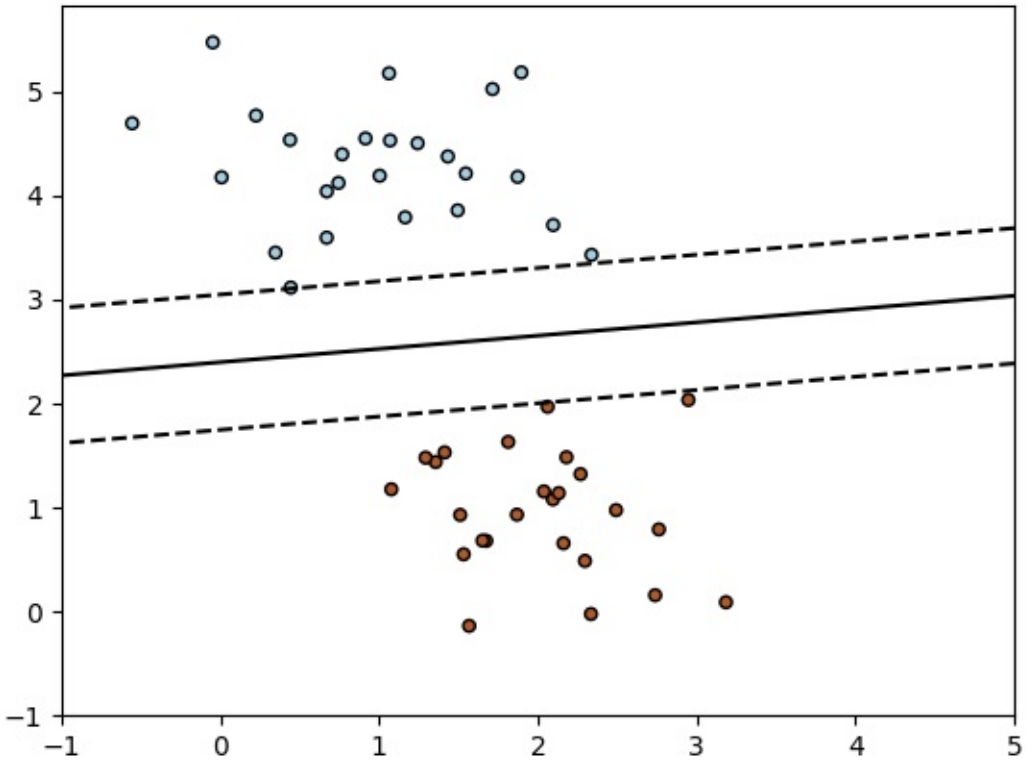
Stochastic Gradient Descent （随机梯度下降法）的劣势：

- SGD 需要一些超参数，例如 regularization （正则化）参数和 number of iterations （迭代次数）。
- SGD 对 feature scaling （特征缩放）敏感。

1.5.1. 分类

> 警告：
>
> 在拟合模型前，确保你重新排列了（打乱）你的训练数据，或者使用 `shuffle=True` 在每次迭代后打乱训练数据。

`SGDClassifier` 类实现了一个简单的随机梯度下降学习例程， 支持分类问题不同的损失函数和正则化方法。



作为另一个 classifier （分类器），拟合 SGD 我们需要两个 array （数组）：保存训练样本的 size 为 [n_samples, n_features] 的数组 X 以及保存训练样本目标值（类标签）的 size 为 [n_samples] 的数组 Y

```
>>> from sklearn.linear_model import SGDClassifier
>>> X = [[0., 0.], [1., 1.]]
>>> y = [0, 1]
>>> clf = SGDClassifier(loss="hinge", penalty="l2")
>>> clf.fit(X, y)
SGDClassifier(alpha=0.0001, average=False, class_weight=None,
              early_stopping=False, epsilon=0.1, eta0=0.0, fit_intercept=True,
              l1_ratio=0.15, learning_rate='optimal', loss='hinge',
              max_iter=1000, n_iter_no_change=5, n_jobs=None, penalty='l2',
              power_t=0.5, random_state=None, shuffle=True, tol=0.001,
              validation_fraction=0.1, verbose=0, warm_start=False)
```

拟合后，我们可以用该模型来预测新值：

```
>>> clf.predict([[2., 2.]])
array([1])
```

SGD 通过训练数据来拟合一个线性模型。成员 `coef_` 保存模型参数：

```
>>> clf.coef_
array([[ 9.9...,  9.9...]])
```

成员 `intercept_` 保存 intercept（截距）（又称作 offset（偏移）或 bias（偏差））：

```
>>> clf.intercept_
array([-9.9...])
```

模型是否使用 intercept（截距），即 a biased hyperplane(一个偏置的超平面)，是由参数 `fit_intercept` 控制的。

使用 `SGDClassifier.decision_function` 来获得到此超平面的 signed distance（符号距离）

```
>>> clf.decision_function([[2., 2.]])
array([ 29.6...])
```

具体的 loss function（损失函数）可以通过 `loss` 参数来设置。 `SGDClassifier` 支持以下的 loss functions（损失函数）：

- `loss="hinge"`: (soft-margin) linear Support Vector Machine（（软-间隔）线性支持向量机），
- `loss="modified_huber"`: smoothed hinge loss（平滑的 hinge 损失），
- `loss="log"`: logistic regression（logistic 回归），
- and all regression losses below（以及所有的回归损失）。

前两个 loss functions（损失函数）是懒惰的，只有一个例子违反了 margin constraint（边界约束），它们才更新模型的参数，这使得训练非常有效率，即使使用了 L2 penalty（惩罚）我们仍然可能得到稀疏的模型结果。

使用 `loss="log"` 或者 `loss="modified_huber"` 来启用 `predict_proba` 方法，其给出每个样本 x 的概率估计 $P(y|x)$ 的一个向量：

```
>>> clf = SGDClassifier(loss="log").fit(X, y)
>>> clf.predict_proba([[1., 1.]])
array([[ 0.00...,  0.99...]])
```

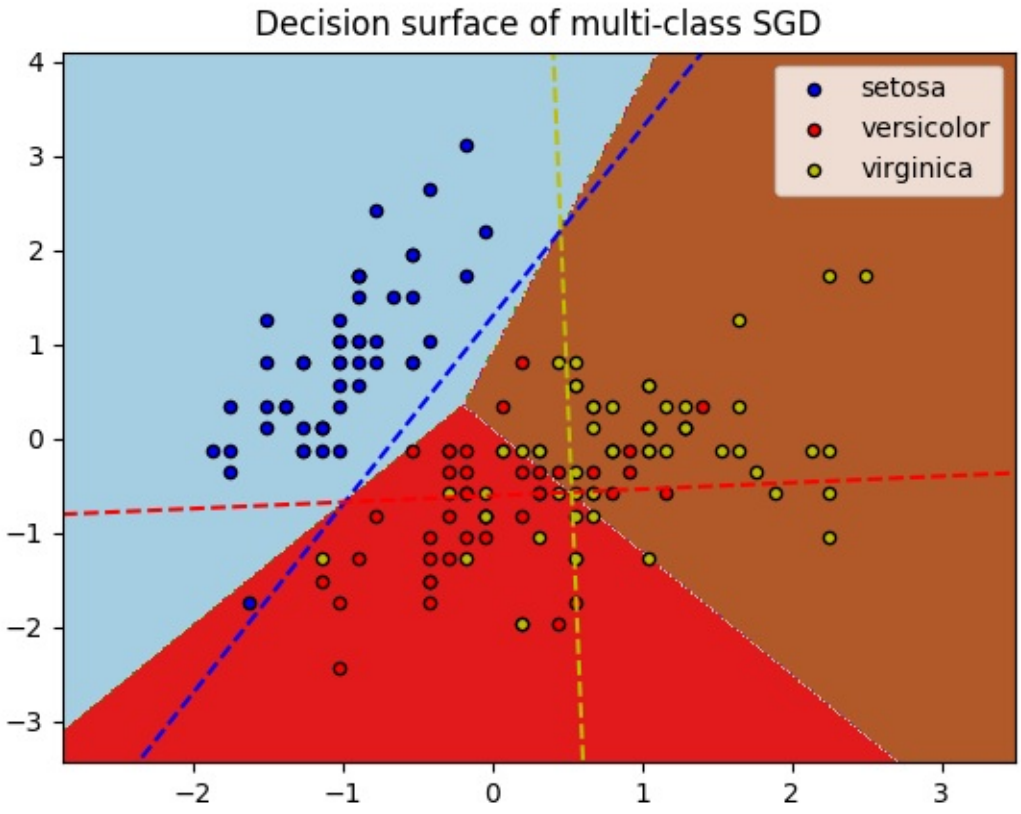
具体的惩罚方法可以通过 `penalty` 参数来设定。 SGD 支持以下 penalties（惩罚）：

- `penalty="l2"`: L2 norm penalty on `coef_`.
- `penalty="l1"`: L1 norm penalty on `coef_`.
- `penalty="elasticnet"`: Convex combination of L2 and L1（L2 型和 L1 型的凸组合）；`(1 - l1_ratio) *L2* + *l1_ratio* L1`.

默认设置为 `penalty="l2"`。 L1 penalty（惩罚）导致稀疏解，使得大多数系数为零。 Elastic Net（弹性网）解决了在特征高相关时 L1 penalty（惩罚）的一些不足。参数 `l1_ratio` 控制了 L1 和 L2 penalty（惩罚）的 convex combination（凸组合）。

`SGDClassifier` 通过利用 “one versus all”（OVA）方法来组合多个二分类器，从而实现多分类。对于每一个 K 类，可以训练一个二分类器来区分自身和其他 $K-1$ 个类。在测试阶段，我们计算每个分类器的

confidence score（置信度分数）（也就是与超平面的距离），并选择置信度最高的分类。下图阐释了基于 iris（鸢尾花）数据集上的 OVA 方法。虚线表示三个 OVA 分类器；不同背景色代表由三个分类器产生的决策面。



在 multi-class classification （多类分类）的情况下， `coef` 是 `shape=[n_classes, n_features]` 的一个二维数组， `intercept` 是 `shape=[nclasses]` 的一个一维数组。 `coef` 的第 i 行保存了第 i 类的 OVA 分类器的权重向量；类以升序索引 （参照属性 `classes_` ）。 注意，原则上，由于它们允许创建一个概率模型，所以 `loss="log"` 和 `loss="modified_huber"` 更适合于 one-vs-all 分类。

`SGDClassifier` 通过拟合参数 `class_weight` 和 `sample_weight` 来支持 weighted classes （加权类）和 weighted instances（加权实例）。更多信息请参照下面的示例和 `SGDClassifier.fit` 的文档。

示例：

- SGD: Maximum margin separating hyperplane,
- Plot multi-class SGD on the iris dataset
- SGD: Weighted samples
- Comparing various online solvers
- SVM: Separating hyperplane for unbalanced classes (参见 "注意")

`SGDClassifier` 支持 averaged SGD (ASGD)。Averaging（均值化）可以通过设置 `average=True` 来启用。AGSD 工作原理是在普通 SGD 的基础上，对每个样本的每次迭代后的系数取均值。当使用 ASGD 时，学习速率可以更大甚至是恒定，在一些数据集上能够加速训练过程。

对于带 logistic loss（logistic 损失）的分类，在 `LogisticRegression` 中提供了另一个采取 averaging strategy（平均策略）的 SGD 变体，其使用了随机平均梯度 (SAG) 算法。

1.5.2. 回归

`SGDRegressor` 类实现了一个简单的随机梯度下降学习例程，它支持用不同的损失函数和惩罚来拟合线性回归模型。`SGDRegressor` 非常适用于有大量训练样本（>10.000)的回归问题，对于其他问题，我们推荐使用 `Ridge` ， `Lasso` ，或 `ElasticNet` 。

具体的损失函数可以通过 `loss` 参数设置。 `SGDRegressor` 支持以下的损失函数：

- `loss="squared_loss"` : Ordinary least squares（普通最小二乘法），
- `loss="huber"` : Huber loss for robust regression（Huber回归），
- `loss="epsilon_insensitive"` : linear Support Vector Regression（线性支持向量回归）.

Huber 和 epsilon-insensitive 损失函数可用于 robust regression（鲁棒回归）。不敏感区域的宽度必须通过参数 `epsilon` 来设定。这个参数取决于目标变量的规模。

`SGDRegressor` 支持 ASGD（平均随机梯度下降） 作为 `SGDClassifier` 。 均值化可以通过设置 `average=True` 来启用。

对于利用了 squared loss（平方损失）和 l2 penalty（l2惩罚）的回归，在 `Ridge` 中提供了另一个采取 averaging strategy（平均策略）的 SGD 变体，其使用了随机平均梯度 (SAG) 算法。

1.5.3. 稀疏数据的随机梯度下降

> 注意
>
> 由于在截距部分收敛学习速率的差异，稀疏实现与密集实现相比产生的结果略有不同。

在 scipy.sparse 支持的格式中，任意矩阵都有对稀疏数据的内置支持方法。但是，为了获得最高的效率，请使用 scipy.sparse.csr_matrix 中定义的 CSR 矩阵格式.

> 示例：
>
> - Classification of text documents using sparse features

1.5.4. 复杂度

SGD 主要的优势在于它的高效性，对于不同规模的训练样本，处理复杂度基本上是线性的。假如 X 是 size 为 (n, p) 的矩阵，训练成本为 $O(knp)$，其中 k 是迭代次数， $\bar{p}$ 是每个样本非零特征的平均数。

但是，最近的理论结果表明，得到期望优化精度的运行时间并不会随着训练集规模扩大而增加。

1.5.5. 停止判据

`SGDClassifier` 和 `SGDRegressor` 类提供了两个判据，当达到给定的收敛水平时停止算法：

- `early_stopping = True` 时，输入数据分为训练集和验证集。该模型在训练集拟合，停止判据是基于验证集上的预测分数。改变参数 `validation_fraction` 可以调整验证集的大小。

- `early_stop = False` 时，模型对整个输入数据进行拟合，停止判据基于整个输入数据上的目标函数来计算。

在这两种情况下，判据在每个 `epoch` 计算一次，当判据没有改变的次数超过参数 `n_iter_no_change` 的值时，算法将停止。改进是用一个容错参数 `tol` 来评估的，并且算法在最大迭代次数 `max_iter` 之后停止。

1.5.6. 实用小贴士

- 随机梯度下降法对 feature scaling （特征缩放）很敏感，因此强烈建议您缩放您的数据。例如，将输入向量 X 上的每个特征缩放到 [0,1] 或 [- 1，+1]， 或将其标准化，使其均值为 0，方差为 1。请注意，必须将 相同 的缩放应用于对应的测试向量中，以获得有意义的结果。使用 `StandardScaler` 能很容易做到这一点：

```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
scaler.fit(X_train) # Don't cheat - fit only on training data
X_train = scaler.transform(X_train)
X_test = scaler.transform(X_test) # apply same transformation to test data
```

假如你的 attributes （属性）有一个固有尺度（例如 word frequencies （词频）或 indicator features（指标特征））就不需要缩放。

- 最好使用 `GridSearchCV` 找到一个合理的 regularization term （正则化项） α ， 它的范围通常在 `10.0**-np.arange(1,7)` 。
- 经验表明，SGD 在处理约 10^6 训练样本后基本收敛。因此，对于迭代次数第一个合理的猜想是 `n_iter = np.ceil(10**6 / n)`，其中 `n` 是训练集的大小。
- 假如将 SGD 应用于使用 PCA 提取的特征，我们发现通过某个常数 c 来缩放特征值是明智的，比如使训练数据的 L2 norm 平均值为 1。
- 我们发现，当特征很多或 eta0 很大时， ASGD（平均随机梯度下降） 效果更好。

> 参考资料：
>
> - “Efficient BackProp” Y. LeCun, L. Bottou, G. Orr, K. Müller - In Neural Networks: Tricks of the Trade 1998.

1.5.7. 数学描述

给定一组训练样本 $(x_1, y_1), \ldots, (x_n, y_n)$ ，其中 $x_i \in \mathbf{R}^m$ ， $y_i \in \{-1, 1\}$ ， 我们的目标是一个线性 scoring function（评价函数） $f(x) = w^T x + b$ ，其中模型参数 $w \in \mathbf{R}^m$ ，截距 $b \in \mathbf{R}$ 。为了做预测， 我们只需要看 $f(x)$ 的符号。找到模型参数的一般选择是通过最小化由以下式子给出的正则化训练误差

$$E(w, b) = \frac{1}{n} \sum_{i=1}^{n} L(y_i, f(x_i)) + \alpha R(w)$$

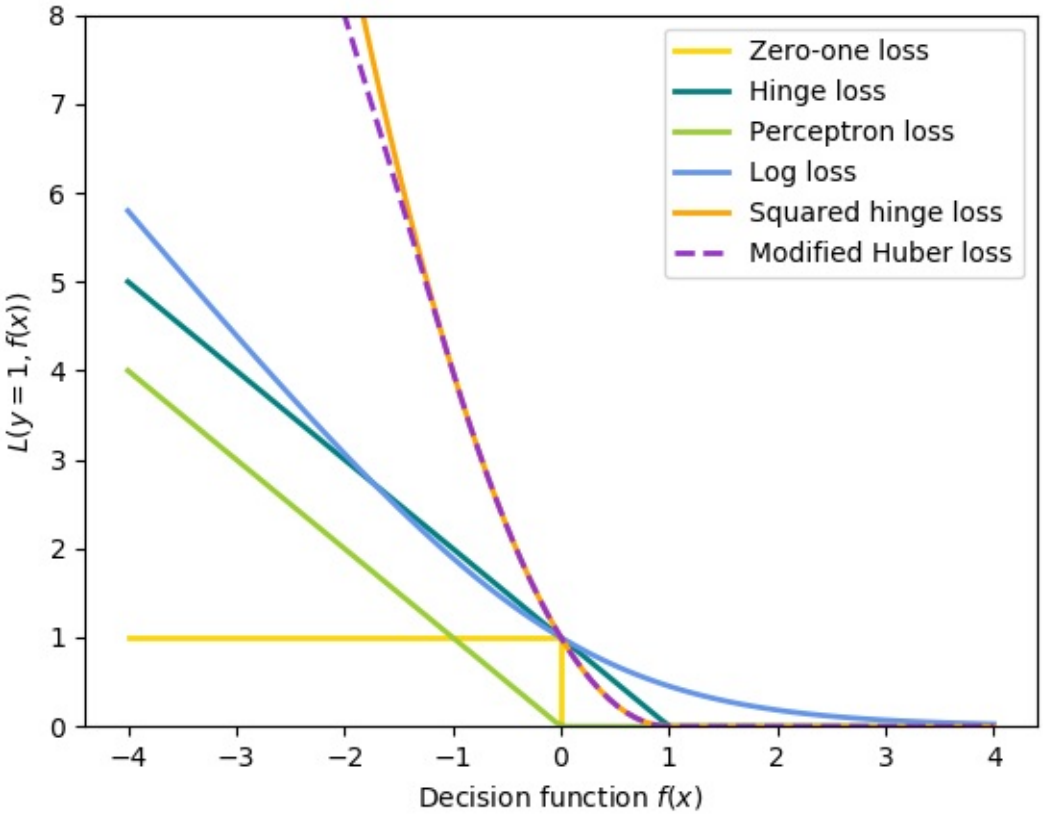
其中 L 衡量模型(mis)拟合程度的损失函数， R 是惩罚模型复杂度的正则化项（也叫作惩罚）； $\alpha > 0$ 是一个非

负超平面。

L 的不同选择产生不同的分类器，例如

- Hinge: (软-间隔) 支持向量机。
- Log: Logistic 回归。
- Least-Squares: 岭回归。
- Epsilon-Insensitive: (软-间隔) 支持向量回归。

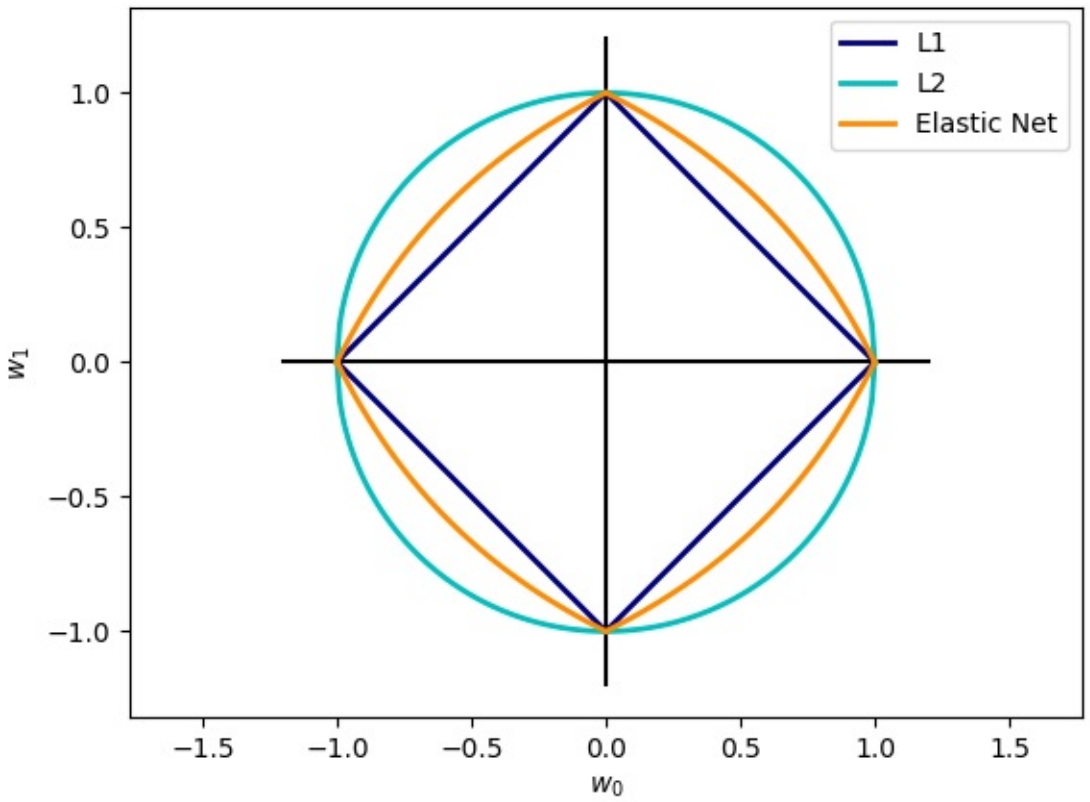
所有上述损失函数可以看作是错误分类误差的上限（0 - 1损失），如下图所示。



比较流行的正则化项 R 包括：

- L2 norm: $R(w) := \frac{1}{2}\sum_{i=1}^{n} w_i^2$，
- L1 norm: $R(w) := \sum_{i=1}^{n} |w_i|$， 这导致了稀疏解。
- Elastic Net: $R(w) := \frac{\rho}{2}\sum_{i=1}^{n} w_i^2 + (1-\rho)\sum_{i=1}^{n} |w_i|$， l2和l1的凸组合， 其中 ρ 由 `1 - l1_ratio` 给出.

下图显示当 $R(w) = 1$ 时参数空间中不同正则项的轮廓。



1.5.7.1. SGD

随机梯度下降法是一种无约束优化问题的优化方法。与（批量）梯度下降法相反，SGD 通过一次只考虑单个训练样本来近似 $E(w,b)$ 真实的梯度。

`SGDClassifier` 类实现了一个 first-order SGD learning routine （一阶 SGD 学习程序）。 算法在训练样本上遍历，并且对每个样本根据由以下式子给出的更新规则来更新模型参数。

$$w \leftarrow w - \eta(\alpha\frac{\partial R(w)}{\partial w} + \frac{\partial L(w^T x_i + b, y_i)}{\partial w})$$

其中 η 是在参数空间中控制步长的 learning rate （学习速率）。 intercept（截距） b 的更新类似但不需要正则化。

学习率 η 可以恒定或者逐渐减小。对于分类来说， 默认的学习率设定方案 （ `learning_rate='optimal'` ）由下式给出。

$$\eta^{(t)} = \frac{1}{\alpha(t_0 + t)}$$

其中 t 是时间步长（总共有 (n_samples * n_iter)时间步长），t_0 是由 Léon Bottou 提出的启发式算法决定的，比如预期的初始更新可以设定为权重的期望大小（假设训练样本的范数近似1）。在 `BaseSGD` 中的

`_init_t` 中可以找到确切的定义。

对于回归来说，默认的学习率是反向缩放（ `learning_rate='invscaling'` ），由下式给出

$$\eta^{(t)} = \frac{eta_0}{t^{power_t}}$$

其中 eta_0 和 $power_t$ 是用户通过 `eta0` 和 `power_t` 分别选择的超参数。

使用固定的学习速率则设置 `learning_rate='constant'` ，或者设置 `eta0` 来指定学习速率。

模型参数可以通过成员 `coef` 和 *`intercept`* 来获得：

- 成员 `coef_` holds the weights（控制权重） w
- 成员 `intercept_` holds b

> 参考资料：
>
> - “Solving large scale linear prediction problems using stochastic gradient descent algorithms” T. Zhang - In Proceedings of ICML ‘04.
> - “Regularization and variable selection via the elastic net” H. Zou, T. Hastie - Journal of the Royal Statistical Society Series B, 67 (2), 301-320.
> - “Towards Optimal One Pass Large Scale Learning with Averaged Stochastic Gradient Descent” Xu, Wei

1.5.8. 实现细节

SGD 的实现受到了 Léon Bottou Stochastic Gradient SVM 的影响。类似于 SvmSGD，权重向量表达为一个标量和一个向量的内积，这样保证在使用L2正则项时可以高效更新权重。 在 sparse feature vectors （稀疏特征向量）的情况下， intercept （截距）是以更小的学习率（乘以 0.01）更新的，这导致了它的更新更加频繁。训练样本按顺序选取并且每处理一个样本就要降低学习速率。我们采用了 Shalev-Shwartz 等人2007年提出的的学习速率设定方案。 对于多类分类，我们使用了 “one versus all” 方法。 我们在 L1 正则化（和 Elastic Net ）中使用 Tsuruoka 等人2009年提出的 truncated gradient algorithm （截断梯度算法）。代码是用 Cython 编写的。

> 参考资料：
>
> - “Stochastic Gradient Descent” L. Bottou - Website, 2010.
> - “The Tradeoffs of Large Scale Machine Learning” L. Bottou - Website, 2011.
> - “Pegasos: Primal estimated sub-gradient solver for svm” S. Shalev-Shwartz, Y. Singer, N. Srebro - In Proceedings of ICML ‘07.
> - “Stochastic gradient descent training for l1-regularized log-linear models with cumulative penalty” Y. Tsuruoka, J. Tsujii, S. Ananiadou - In Proceedings of the AFNLP/ACL ‘09.

我们一直在努力

apachecn/sklearn-doc-zh

加入QQ群

1.6. 最近邻

校验者：@DataMonk2017@Veyron C@舞空@Loopy@qinhanmin2014翻译者：@那伊抹微笑

`sklearn.neighbors` 提供了 neighbors-based（基于邻居的）无监督学习以及监督学习方法的功能。 无监督的最近邻是许多其它学习方法的基础，尤其是 manifold learning（流形学习）和 spectral clustering（谱聚类）。 neighbors-based（基于邻居的）监督学习分为两种： classification （分类）针对的是具有离散标签的数据，regression （回归）针对的是具有连续标签的数据。

最近邻方法背后的原理是从训练样本中找到与新点在距离上最近的预定数量的几个点，然后从这些点中预测标签。 这些点的数量可以是用户自定义的常量（K-最近邻学习）， 也可以根据不同的点的局部密度（基于半径的最近邻学习）确定。距离通常可以通过任何度量来衡量： standard Euclidean distance（标准欧式距离）是最常见的选择。 Neighbors-based（基于邻居的）方法被称为 非泛化 机器学习方法， 因为它们只是简单地”记住”了其所有的训练数据（可能转换为一个快速索引结构，如 Ball Tree 或 KD Tree）。

尽管它简单，但最近邻算法已经成功地适用于很多的分类和回归问题，例如手写数字或卫星图像的场景。 作为一个 non-parametric（非参数化）方法，它经常成功地应用于决策边界非常不规则的分类情景下。

`sklearn.neighbors` 可以处理 Numpy 数组或 `scipy.sparse` 矩阵作为其输入。 对于密集矩阵，大多数可能的距离度量都是支持的。对于稀疏矩阵，支持搜索任意的 Minkowski 度量。

许多学习路径/方法都是依赖最近邻作为核心。 一个例子是 核密度估计 ， 在 密度估计 章节中有讨论。

1.6.1. 无监督最近邻

`NearestNeighbors` （最近邻）实现了 unsupervised nearest neighbors learning（无监督的最近邻学习）。 它为三种不同的最近邻算法提供统一的接口： `BallTree` ， `KDTree` ， 还有基于 `sklearn.metrics.pairwise` 的 brute-force 算法。算法的选择可通过关键字 `'algorithm'` 来控制， 并必须是 `['auto', 'balltree', 'kd_tree', 'brute']` 其中的一个。当设置为默认值 `'auto'` 时，算法会尝试从训练数据中确定最佳方法。有关上述每个选项的优缺点，参见 *[* `Nearest Neighbor Algorithms`](#id11) 。

> 警告
>
> 关于最近邻算法，如果邻居 $k+1$ 和邻居 k 具有相同的距离，但具有不同的标签， 结果将取决于训练数据的顺序。

1.6.1.1. 找到最近邻

为了完成找到两组数据集中最近邻点的简单任务， 可以使用 `sklearn.neighbors` 中的无监督算法：

```
>>> from sklearn.neighbors import NearestNeighbors
>>> import numpy as np
>>> X = np.array([[-1, -1], [-2, -1], [-3, -2], [1, 1], [2, 1], [3, 2]])
>>> nbrs = NearestNeighbors(n_neighbors=2, algorithm='ball_tree').fit(X)
>>> distances, indices = nbrs.kneighbors(X)
>>> indices
array([[0, 1],
```

```
  [1, 0],
  [2, 1],
  [3, 4],
  [4, 3],
  [5, 4]]...)
>>> distances
array([[0.        , 1.        ],
       [0.        , 1.        ],
       [0.        , 1.41421356],
       [0.        , 1.        ],
       [0.        , 1.        ],
       [0.        , 1.41421356]])
```

因为查询集匹配训练集，每个点的最近邻点是其自身，距离为0。

还可以有效地生成一个稀疏图来标识相连点之间的连接情况：

```
>>> nbrs.kneighbors_graph(X).toarray()
array([[ 1.,  1.,  0.,  0.,  0.,  0.],
 [ 1.,  1.,  0.,  0.,  0.,  0.],
 [ 0.,  1.,  1.,  0.,  0.,  0.],
 [ 0.,  0.,  0.,  1.,  1.,  0.],
 [ 0.,  0.,  0.,  1.,  1.,  0.],
 [ 0.,  0.,  0.,  0.,  1.,  1.]])
```

我们的数据集是结构化的，因此按索引顺序的相邻点就在参数空间相邻，从而生成了近似 K-nearest neighbors（K-近邻）的块对角矩阵。 这种稀疏图在各种的利用点之间的空间关系进行无监督学习的情况下都很有用：特别地可参见 `sklearn.manifold.Isomap` ， `sklearn.manifold.LocallyLinearEmbedding` ，和 `sklearn.cluster.SpectralClustering` 。

1.6.1.2. KDTree 和 BallTree 类

另外，我们可以使用 `KDTree` 或 `BallTree` 来找最近邻。 这是上文使用过的 `NearestNeighbors` 类所包含的功能。 `KDTree` 和 `BallTree` 具有相同的接口； 我们将在这里展示使用 `KDTree` 的例子：

```
>>> from sklearn.neighbors import KDTree
>>> import numpy as np
>>> X = np.array([[-1, -1], [-2, -1], [-3, -2], [1, 1], [2, 1], [3, 2]])
>>> kdt = KDTree(X, leaf_size=30, metric='euclidean')
>>> kdt.query(X, k=2, return_distance=False)
array([[0, 1],
 [1, 0],
 [2, 1],
 [3, 4],
 [4, 3],
 [5, 4]]...)
```

对于近邻搜索中选项的更多信息，包括各种距离度量的说明和策略的说明等，请参阅 `KDTree` 和 `BallTree` 类文档。 关于可用度量距离的列表，请参阅 `DistanceMetric` 类。

1.6.2. 最近邻分类

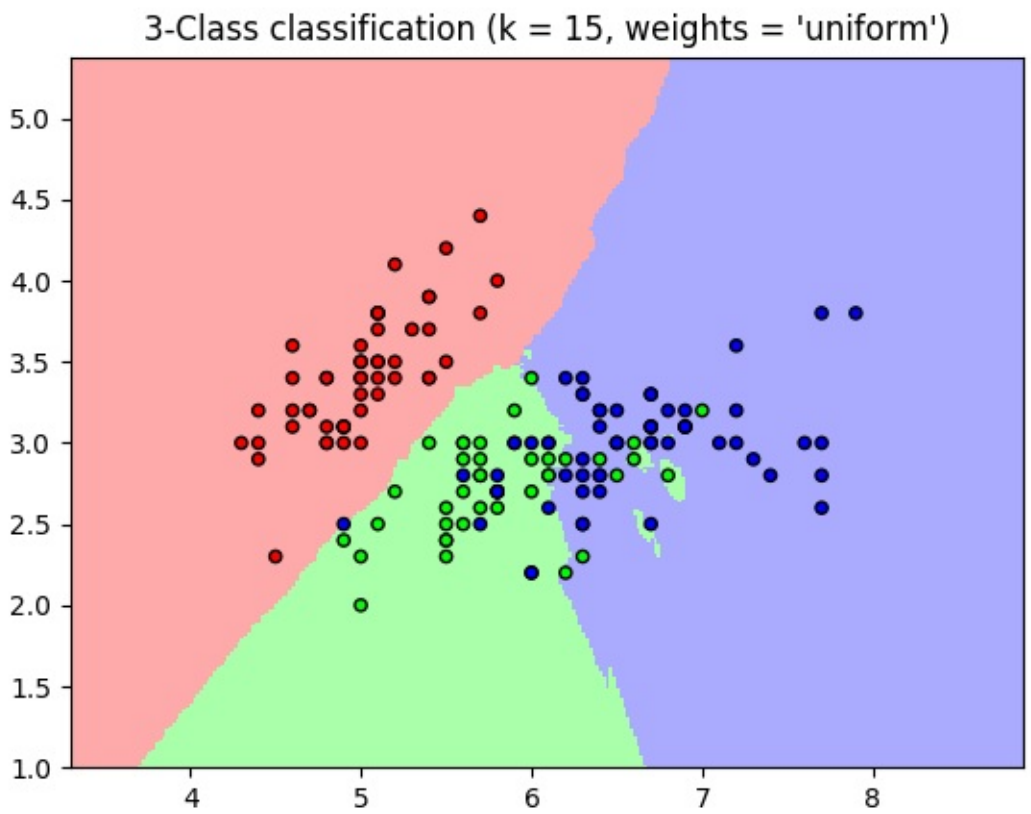
最近邻分类属于 基于实例的学习 或 非泛化学习 ：它不会去构造一个泛化的内部模型，而是简单地存储训练数据的实例。 分类是由每个点的最近邻的简单多数投票中计算得到的：一个查询点的数据类型是由它最近邻点中最具代表性的数据类型来决定的。

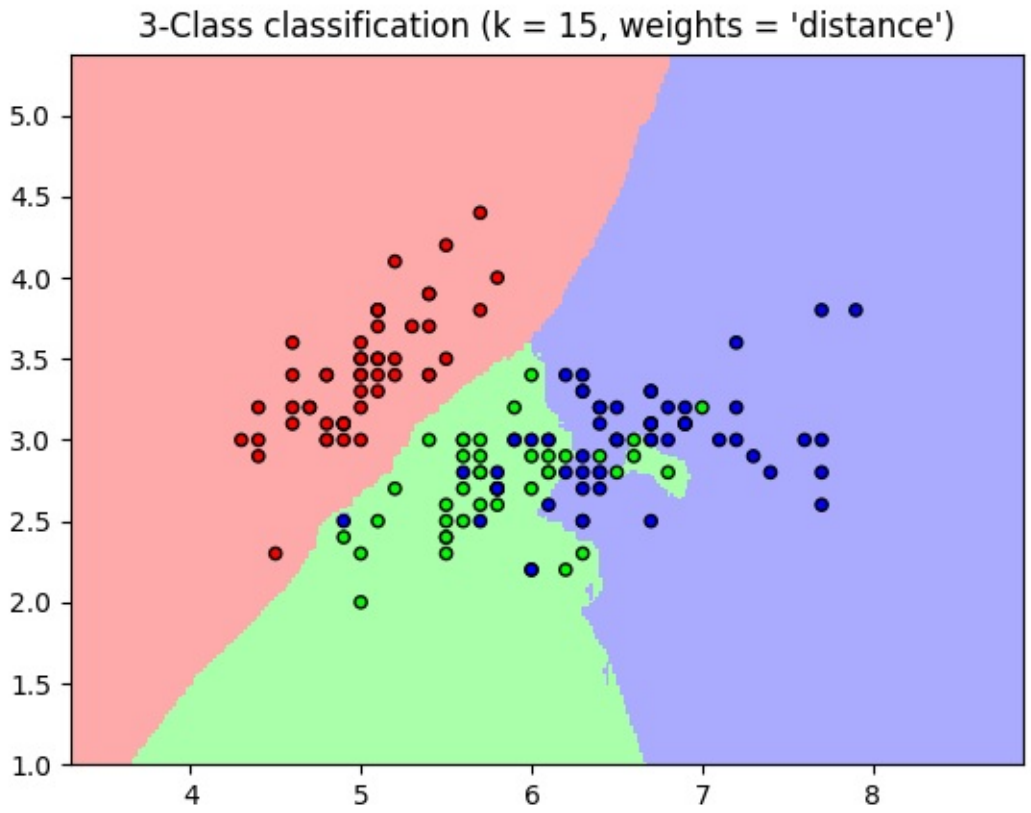
scikit-learn 实现了两种不同的最近邻分类器： 基于每个查询点的 k 个最近邻实现，其中 k 是用户指定的整数值。 `RadiusNeighborsClassifier` 基于每个查询点的固定半径 r 内的邻居数量实现， 其中 r 是用户指定的浮点数值。

k -邻居分类是KNeighborsClassifie的技术中比较常用的一种。 值的最佳选择是高度依赖数据的：通常较大的 k 是会抑制噪声的影响，但是使得分类界限不明显。

如果数据是不均匀采样的，那么 `RadiusNeighborsClassifier` 中的基于半径的近邻分类可能是更好的选择。用户指定一个固定半径 ，使得稀疏邻居中的点使用较少的最近邻来分类。对于高维参数空间，这个方法会由于所谓的 “维度灾难” 而变得不那么有效。

基本的最近邻分类使用统一的权重：分配给查询点的值是从最近邻的简单多数投票中计算出来的。 在某些环境下，最好对邻居进行加权，使得更近邻更有利于拟合。可以通过 `weights` 关键字来实现。默认值 `weights = 'uniform'` 为每个近邻分配统一的权重。而 `weights = 'distance'` 分配权重与查询点的距离成反比。 或者，用户可以自定义一个距离函数用来计算权重。





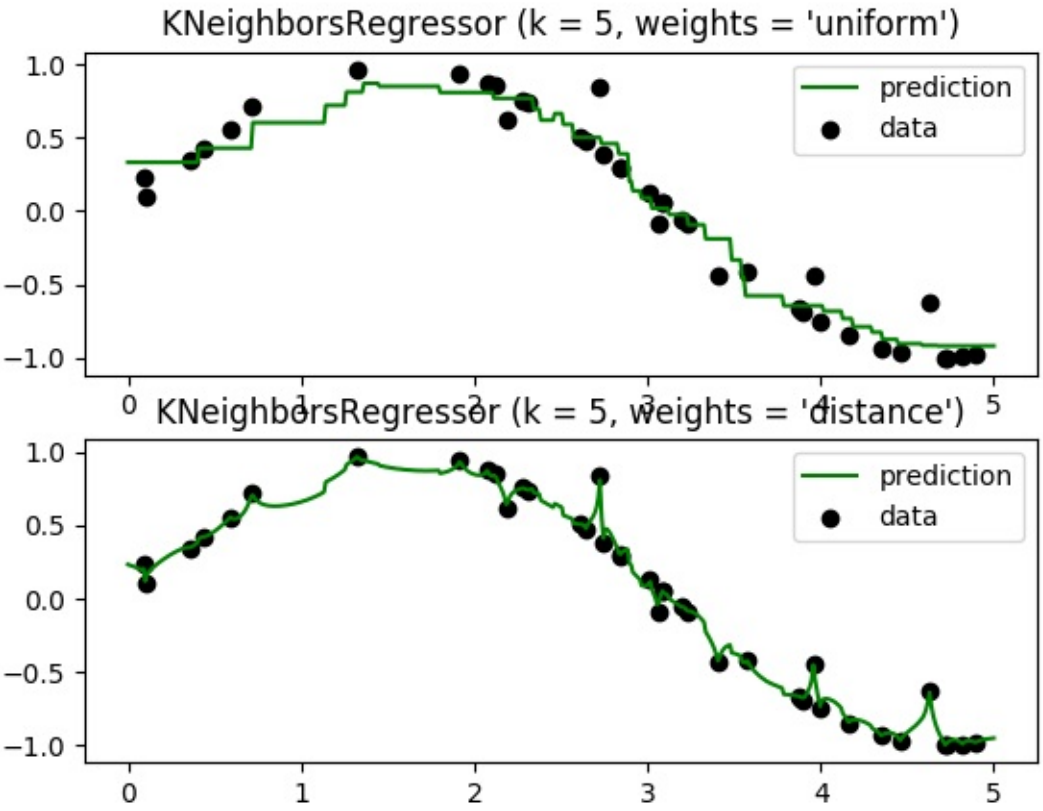
> 示例：
>
> - Nearest Neighbors Classification：使用最近邻进行分类的示例。

1.6.3. 最近邻回归

最近邻回归是用在数据标签为连续变量，而不是离散变量的情况下。分配给查询点的标签是由它的最近邻标签的均值计算而来的。

scikit-learn 实现了两种不同的最近邻回归：`KNeighborsRegressor` 基于每个查询点的 k 个最近邻实现，其中 k 是用户指定的整数值。`RadiusNeighborsRegressor` 基于每个查询点的固定半径 r 内的邻点数量实现，其中 r 是用户指定的浮点数值。

基本的最近邻回归使用统一的权重：即，本地邻域内的每个邻点对查询点的分类贡献一致。在某些环境下，对邻点加权可能是有利的，使得附近点对于回归所作出的贡献多于远处点。这可以通过 `weights` 关键字来实现。默认值 `weights = 'uniform'` 为所有点分配同等权重。而 `weights = 'distance'` 分配的权重与查询点距离呈反比。或者，用户可以自定义一个距离函数用来计算权重。



使用多输出的最近邻进行回归分析 `Face completion with a multi-output estimators`。 |

利用多输出估计器，演示了多输出最近邻回归方法在人脸补全中的应用。在这个例子中，输入 `X` 是脸上半部分像素，输出 `Y` 是脸下半部分像素。



示例：

- `Nearest Neighbors regression`：使用最近邻进行回归的示例。
- `Face completion with a multi-output estimators`：使用最近邻进行多输出回归的示例。

1.6.4. 最近邻算法

1.6.4.1. 暴力计算

最近邻的快速计算是机器学习中一个活跃的研究领域。最简单的近邻搜索的实现涉及数据集中所有成对点之间距离的暴力计算： 对于 D 维度中的 N 个样本来说， 这个方法的复杂度是 $O[DN^2]$ 。 对于小数据样本，高效的暴力近邻搜索是非常有竞争力的。 然而，随着样本数 N 的增长，暴力方法很快变得不切实际了。在 `sklearn.neighbors` 类中， 暴力近邻搜索通过关键字 `algorithm = 'brute'` 来指定，并通过 `sklearn.metrics.pairwise` 中的例程来进行计算。

1.6.4.2. K-D 树

为了解决效率低下的暴力计算方法，已经发明了大量的基于树的数据结构。总的来说， 这些结构试图通过有效地编码样本的 aggregate distance（聚合距离）信息来减少所需的距离计算量。 基本思想是，若 A 点距离 B 点非常远， B 点距离 C 点非常近， 可知 A 点与 C 点很遥远，不需要明确计算它们的距离。 通过这样的方式，近邻搜索的计算成本可以降低为 $O[DN\log(N)]$ 或更低。 这是对于暴力搜索在大样本数 N 中表现的显著改善。

利用这种聚合信息的早期方法是 *KD tree* 数据结构（ *K-dimensional tree* 的简写），它将二维 *Quad-trees* 和三维 *Oct-trees* 推广到任意数量的维度. KD 树是一个二叉树结构，它沿着数据轴递归地划分参数空间，将其划分为嵌入数据点的嵌套的各向异性区域。 KD 树的构造非常快：因为只需沿数据轴执行分区， 无需计算 D-dimensional 距离。 一旦构建完成， 查询点的最近邻距离计算复杂度仅为 $O[\log(N)]$ 。 虽然 KD 树的方法对于低维度 ($D < 20$) 近邻搜索非常快， 当 D 增长到很大时， 效率变低：这就是所谓的 “维度灾难” 的一种体现。 在 scikit-learn 中， KD 树近邻搜索可以使用关键字 `algorithm = 'kd_tree'` 来指定， 并且使用类 `KDTree` 来计算。

> 参考资料：
>
> - “Multidimensional binary search trees used for associative searching”, Bentley, J.L., Communications of the ACM (1975)

1.6.4.3. Ball 树

为了解决 KD 树在高维上效率低下的问题， *ball* 树 数据结构就被研发出来了. 其中 KD 树沿笛卡尔轴（即坐标轴）分割数据， ball 树在沿着一系列的 hyper-spheres 来分割数据. 通过这种方法构建的树要比 KD 树消耗更多的时间， 但是这种数据结构对于高结构化的数据是非常有效的， 即使在高维度上也是一样.

ball 树将数据递归地划分为由质心 C 和半径 R 定义的节点,使得节点中的每个点位于由 r 和 C 定义的 hyper-sphere 内. 通过使用 *triangle inequality*（三角不等式） 减少近邻搜索的候选点数：

$$|x+y| \leq |x| + |y|$$

通过这种设置， 测试点和质心之间的单一距离计算足以确定距节点内所有点的距离的下限和上限. 由于 ball 树节点的球形几何， 它在高维度上的性能超出 *KD-tree*， 尽管实际的性能高度依赖于训练数据的结构. 在 scikit-learn 中， 基于 ball 树的近邻搜索可以使用关键字 `algorithm = 'ball_tree'` 来指定， 并且使用类 `sklearn.neighbors.BallTree` 来计算. 或者， 用户可以直接使用 `BallTree` 类.

> 参考资料：
>
> - “Five balltree construction algorithms”, Omohundro, S.M., International Computer Science Institute Technical Report (1989)

1.6.4.4. 最近邻算法的选择

对于给定数据集的最优算法是一个复杂的选择，并且取决于多个因素：

- 样本数量 N (i.e. `n_samples`) 和维度 D (例如. `nfeatures`).
  - *Brute force* 查询时间以 $O[DN]$ 增长
  - *Ball tree* 查询时间大约以 $O[D\log(N)]$ 增长
  - KD tree 的查询时间 的变化是很难精确描述的.对于较小的 D (小于20) 的成本大约是 $O[D\log(N)]$，并且 KD 树更加有效.对于较大的 D 成本的增加接近 $O[DN]$，由于树结构引起的开销会导致查询效率比暴力还要低.

对于小数据集 (n小于30)，log(N)相当于N，暴力算法比基于树的算法更加有效. `KDTree` 和 `BallTree` 通过提供一个 *leaf size* 参数来解决这个问题：这控制了查询切换到暴力计算样本数量. 使得两种算法对较小的 N的效率都能接近于暴力计算.

- 数据结构：数据的 *intrinsic dimensionality* (本征维数) 和/或数据的 *sparsity* (稀疏度). 本征维数是指数据所在的流形的维数 $d \leq D$，在参数空间可以是线性或非线性的. 稀疏度指的是数据填充参数空间的程度(这与“稀疏”矩阵中使用的概念不同，数据矩阵可能没有零项，但是从这个意义上来讲,它的 **structure** 仍然是 “稀疏” 的)。
  - *Brute force* (暴力查询)时间不受数据结构的影响。
  - *Ball tree* 和 *KD tree* 的数据结构对查询时间影响很大. 一般地，小维度的 sparser (稀疏) 数据会使查询更快. 因为 KD 树的内部表现形式是与参数轴对齐的，对于任意的结构化数据它通常不会表现的像 ball tree 那样好.在机器学习中往往使用的数据集是非常结构化的，而且非常适合基于树结构的查询。
- query point (查询点) 所需的近邻数 k 。

> Brute force 查询时间几乎不受 k 值的影响.> __*Ball tree* 和 *KD tree* 的查询时间会随着 k 的增加而变慢. 这是由于两个影响：首先，k 的值越大在参数空间中搜索的部分就越大. 其次，使用 $k > 1$ 进行树的遍历时，需要对内部结果进行排序.

当 k 相比 N 变大时，在基于树的查询中修剪树枝的能力是减弱的. 在这种情况下，暴力查询会更加有效.

- query points（查询点）数. ball tree 和 KD Tree 都需要一个构建阶段. 在许多查询中分摊时，这种结构的成本可以忽略不计。 如果只执行少量的查询，可是构建成本却占总成本的很大一部分. 如果仅需查询很少的点，暴力方法会比基于树的方法更好.

一般地，如果 $k >= N/2$ ，或输入是稀疏矩阵，或 `'effectivemetric'` 不在 `'kdtree'` 或 `'ball_tree'` 的 `'VALID_METRICS'` 列表中，那么 `algorithm = 'auto'` 选择 `'brute'` 。 如果 $k < N/2$ 并且 `'effective_metric '` 在 `'kdtree'` 的 `'VALID_METRICS'` 列表中，那么 `algorithm = 'auto'` 选择 `'kd_tree'` 。 如果 $k < N/2$ 并且 `'effective_metric '` 在 `'ball_tree'` 的 `'VALID_METRICS'` 列表中，那么 `algorithm = 'auto'` 选择 `'ball_tree'` 。 这种选择基于以下假设：查询点的数量与训练点的数量至少在相同的数量级，并且 `leaf_size` 接近其默认值 `30` .

1.6.4.5. leaf_size 的影响

如上所述，对于小样本暴力搜索是比基于树的搜索更有效的方法. 这一事实在 ball 树和 KD 树中被解释为在叶节点内部切换到暴力搜索. 该开关的级别可以使用参数 `leaf_size` 来指定. 这个参数选择有很多的效果：

构造时间：更大的 `leaf_size` 会导致更快的树构建时间，因为需要创建更少的节点.

查询时间:一个大或小的 `leaf_size` 可能会导致次优查询成本. 当 `leaf_size` 接近 1 时，遍历节点所涉及的开销大大减慢了查询时间. 当 `leaf_size` ，接近训练集的大小，查询变得本质上是暴力的. 这些之间的一个很好的妥协是 `leaf_size = 30` ，这是该参数的默认值.

内存:随着leaf_size的增加，存储树结构所需的内存减少。 对于存储每个节点的D维质心的ball tree，这点至关重要。 针对 `BallTree` 所需的存储空间近似于 `1 / leaf_size` 乘以训练集的大小.

`leaf_size` 不被 brute force queries（暴力查询）所引用.

1.6.5. 最近质心分类

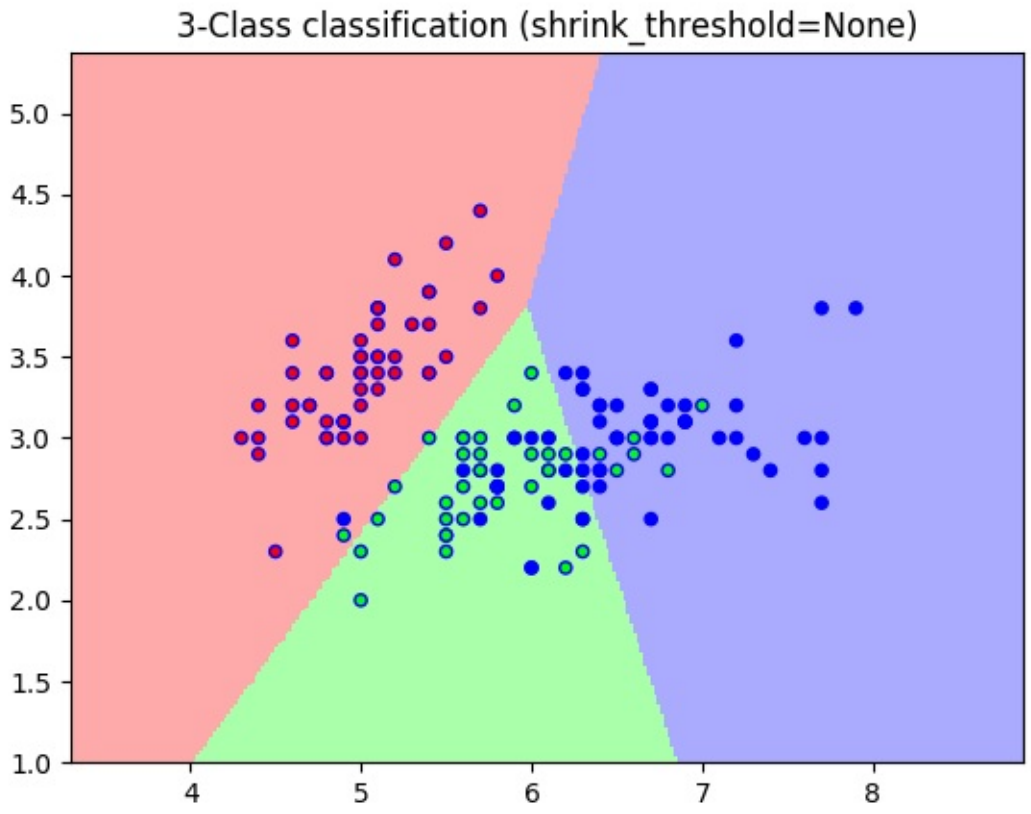
该 `NearestCentroid` 分类器是一个简单的算法，通过其成员的质心来表示每个类。 实际上，这使得它类似于 `sklearn.KMeans` 算法的标签更新阶段. 它也没有参数选择，使其成为良好的基准分类器. 然而，它确实受到非凸类的影响，即当类有显著不同的方差时。所以这个分类器假设所有维度的方差都是相等的。 对于没有做出这个假设的更复杂的方法，请参阅线性判别分析（ `sklearn.discriminant_analysis.LinearDiscriminantAnalysis` ）和二次判别分析（ `sklearn.discriminant_analysis.QuadraticDiscriminantAnalysis` ）. 默认的 `NearestCentroid` 用法示例如下：

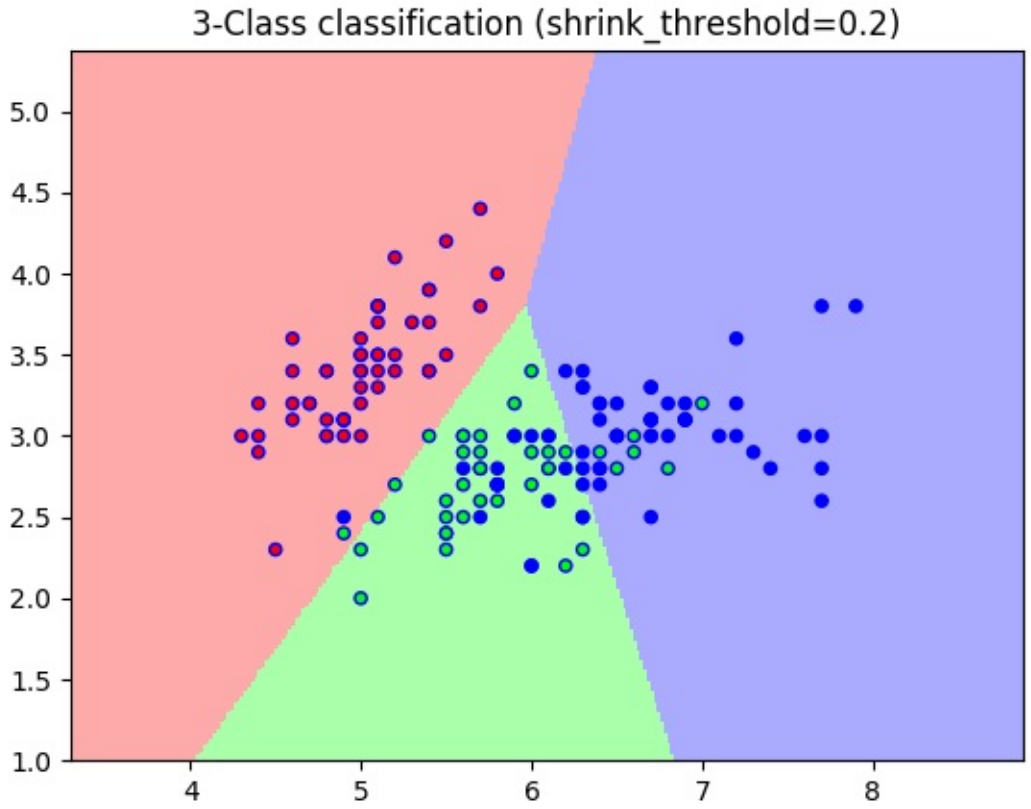
```
>>> from sklearn.neighbors.nearest_centroid import NearestCentroid
>>> import numpy as np
>>> X = np.array([[-1, -1], [-2, -1], [-3, -2], [1, 1], [2, 1], [3, 2]])
>>> y = np.array([1, 1, 1, 2, 2, 2])
>>> clf = NearestCentroid()
>>> clf.fit(X, y)
NearestCentroid(metric='euclidean', shrink_threshold=None)
>>> print(clf.predict([[-0.8, -1]]))
[1]
```

1.6.5.1. 最近缩小质心

该 `NearestCentroid` 分类器有一个 `shrink_threshold` 参数，它实现了 nearest shrunken centroid 分类器. 实际上，每个质心的每个特征的值除以该特征的类中的方差. 然后通过 `shrink_threshold` 来减小特征值. 最值得注意的是，如果特定特征值过0，则将其设置为0. 实际上，这个方法移除了影响分类器的特征。 这很有用，例如，去除噪声特征.

在以下例子中，使用一个较小的 shrink 阈值将模型的准确度从 0.81 提高到 0.82.



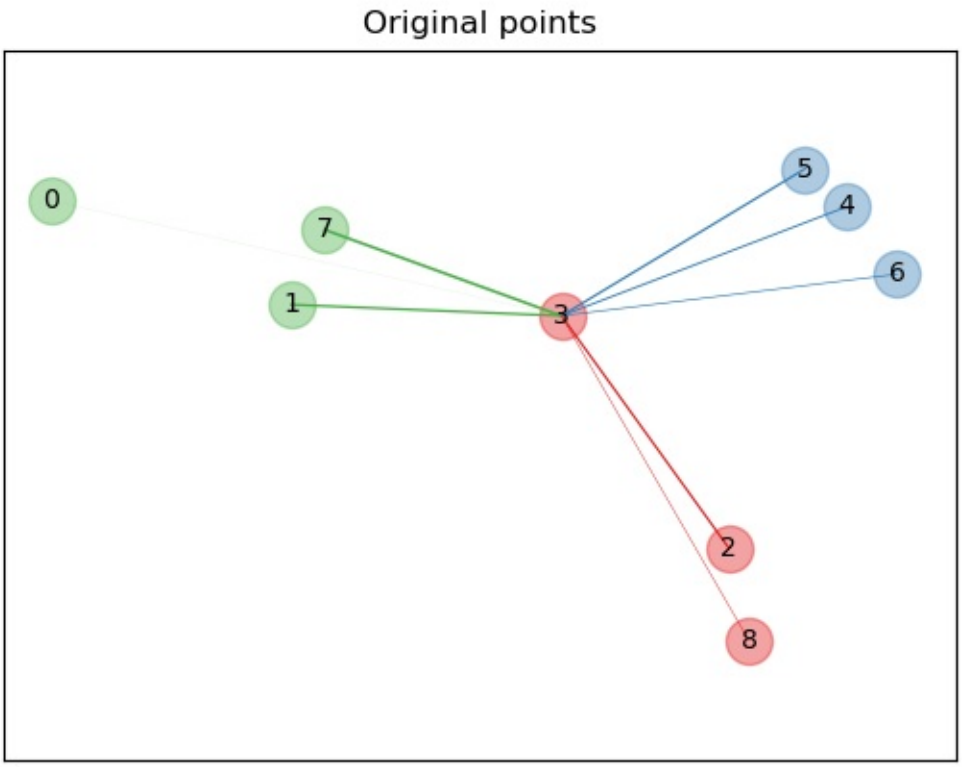


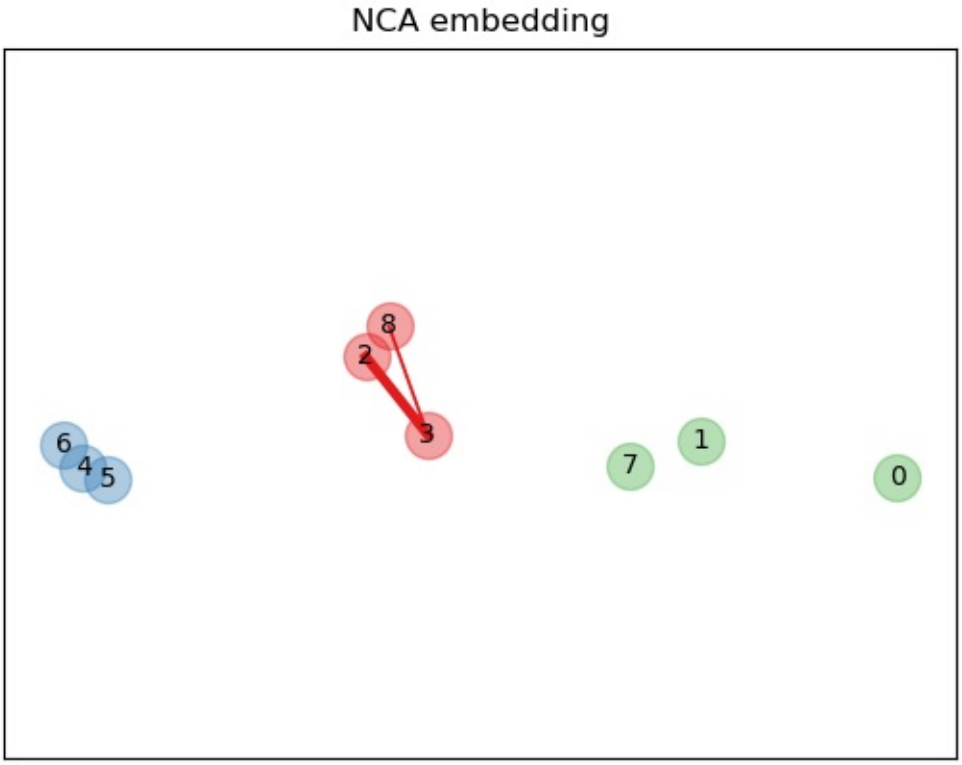
示例：

- Nearest Centroid Classification: 一个分类的例子，它使用了不同 shrink 阀值的最近质心.

1.6.6 邻域成分分析

邻域成分分析(NCA, NeighborhoodComponentsAnalysis)是一种距离度量学习算法，其目的是提高最近邻分类相对于标准欧氏距离的准确性。该算法直接最大化训练集上k近邻(KNN)得分的随机变量，还可以拟合数据的低维线性投影，用于数据可视化和快速分类。





在上图中，我们考虑随机生成的数据集中的一些点。重点研究了 3 号样本点的随机KNN分类问题．样本3和另一个点之间的链路厚度与它们之间的距离成正比，可以看作是随机最近邻预测规则分配给该点的相对权重(或概率)。在原始空

间中，样本3有许多来自不同类的随机邻居，因此正确的分类不太可能。然而，在NCA学习的投影空间中，唯一权重不可忽略的随机邻域与样本3属于同一类，保证了样本3的分类良好。有关详细信息，请参阅数学公式。

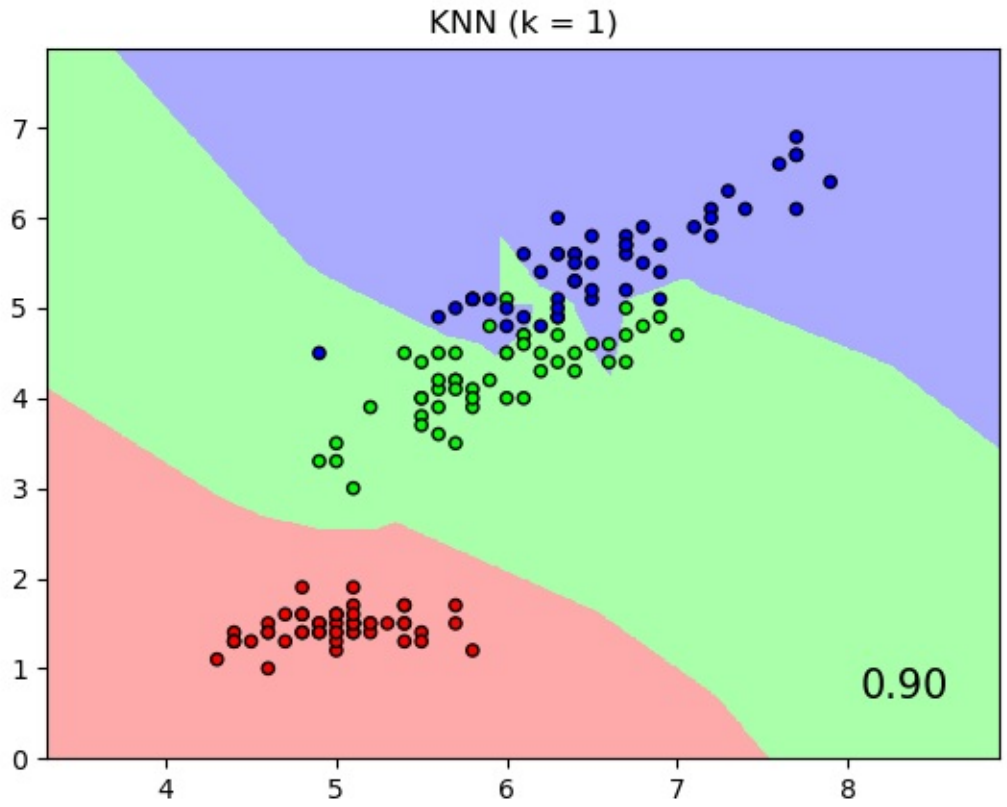
1.6.6.1. 分类

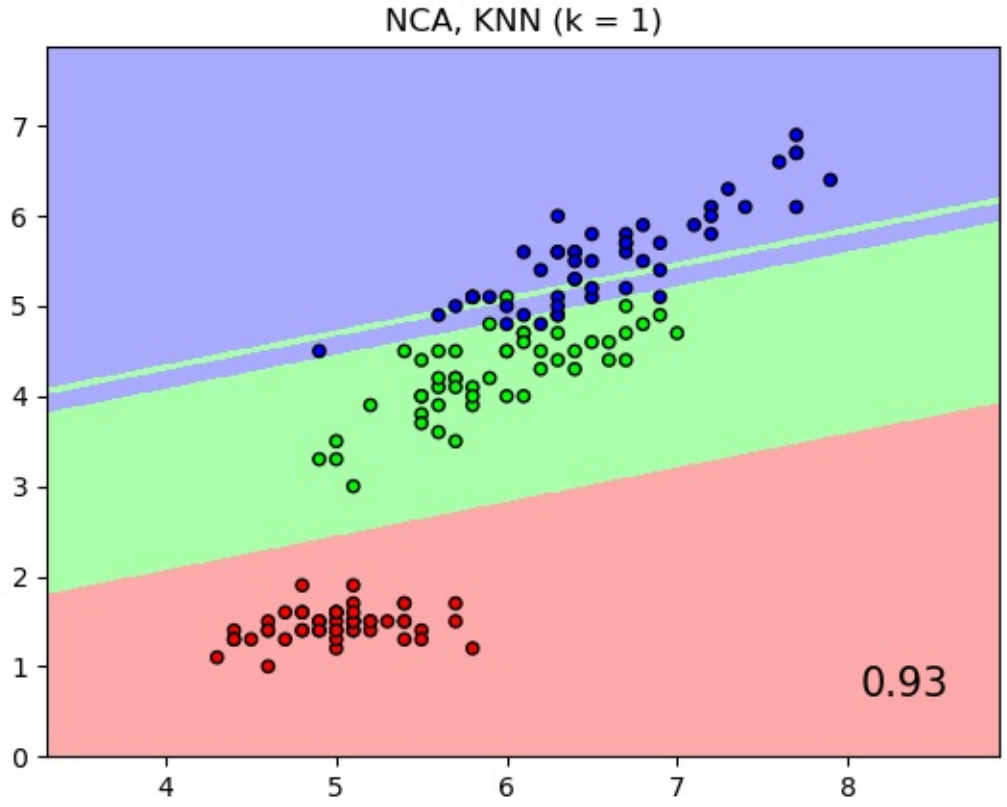
与最近邻分类器(`KNeighborsClassifier`)相结合，NCA是一种有效的分类算法，因为它可以自然地处理多类问题，而不需要增加模型的大小，并且不引入需要用户进行微调的额外参数。

NCA分类在不同规模和难度的数据集的实际应用中显示出良好的效果。与线性判别分析等相关方法相比，NCA没有对类的分布做任何假设。而最近邻分类自然会产生高度不规则的决策边界。

要使用这个模型进行分类，需要将一个`NeighborhoodComponentsAnalysis`实例与一个`KNeighborsClassifier`实例结合起来，`NeighborhoodComponentsAnalysis`实例拟合最优转换，`KNeighborsClassifier`实例在投影空间中执行分类。下面是一个使用这两个类的例子：

```
>>> from sklearn.neighbors import (NeighborhoodComponentsAnalysis,
... KNeighborsClassifier)
>>> from sklearn.datasets import load_iris
>>> from sklearn.model_selection import train_test_split
>>> from sklearn.pipeline import Pipeline
>>> X, y = load_iris(return_X_y=True)
>>> X_train, X_test, y_train, y_test = train_test_split(X, y,
... stratify=y, test_size=0.7, random_state=42)
>>> nca = NeighborhoodComponentsAnalysis(random_state=42)
>>> knn = KNeighborsClassifier(n_neighbors=3)
>>> nca_pipe = Pipeline([('nca', nca), ('knn', knn)])
>>> nca_pipe.fit(X_train, y_train)
Pipeline(...)
>>> print(nca_pipe.score(X_test, y_test))
0.96190476...
```

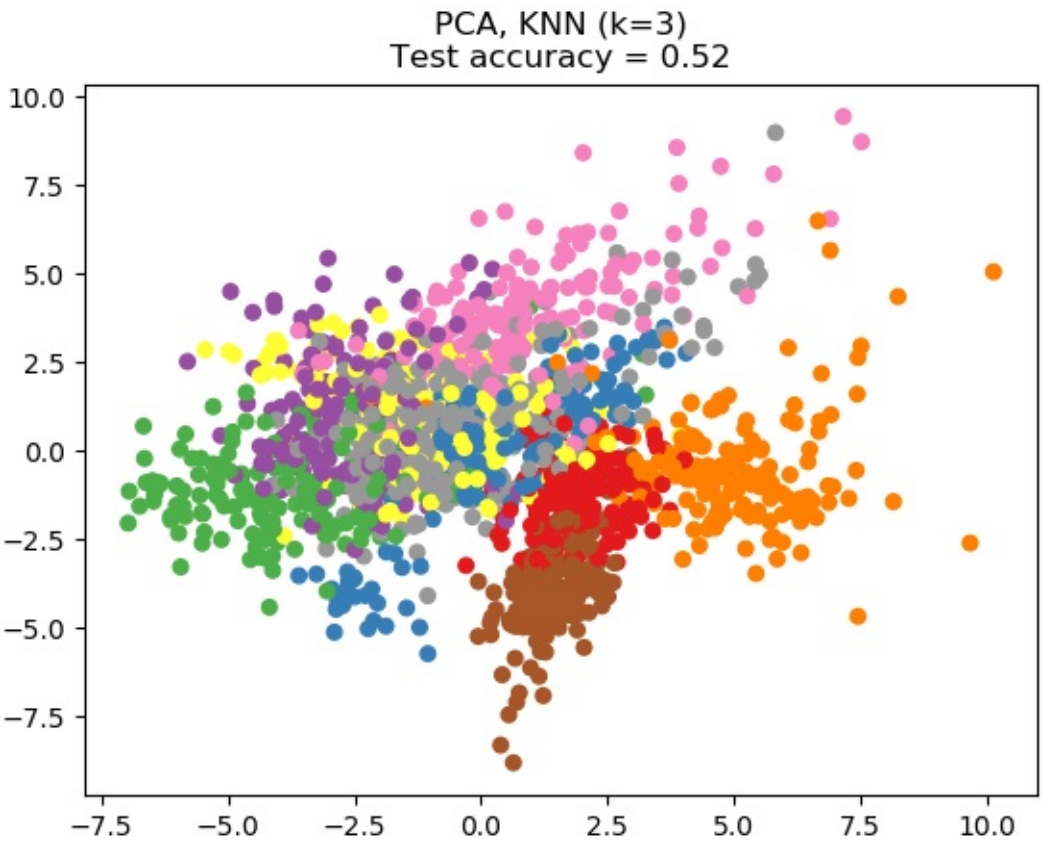


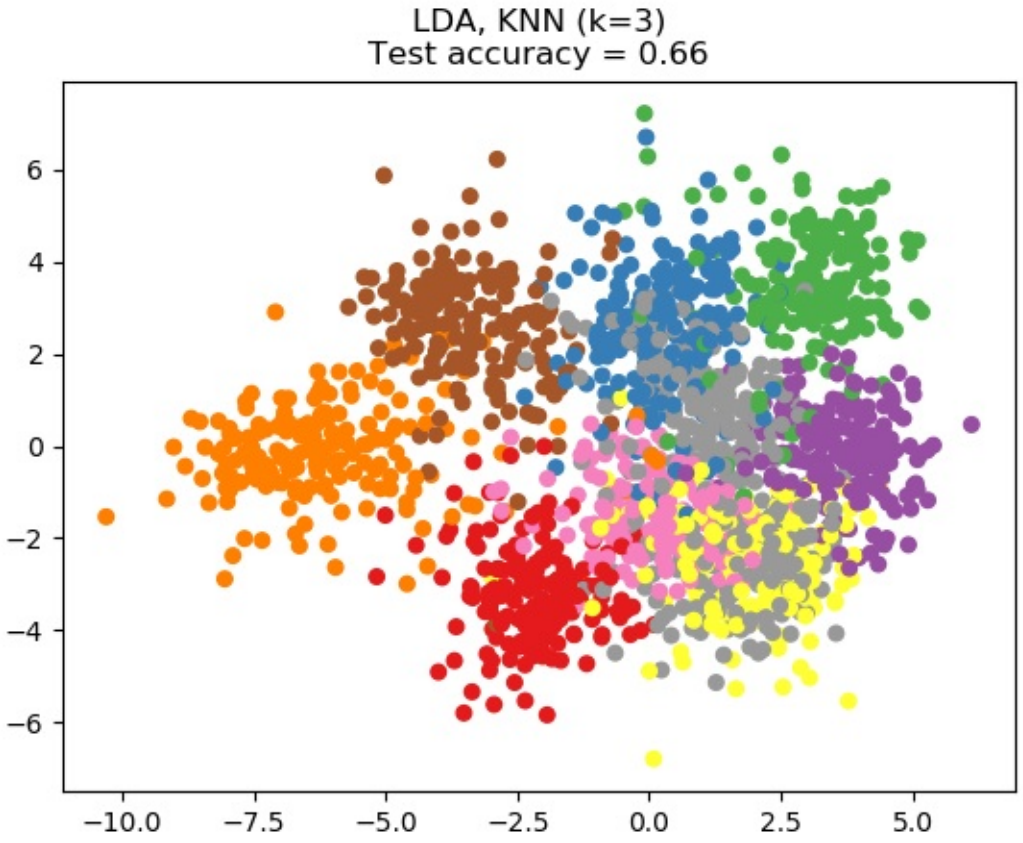


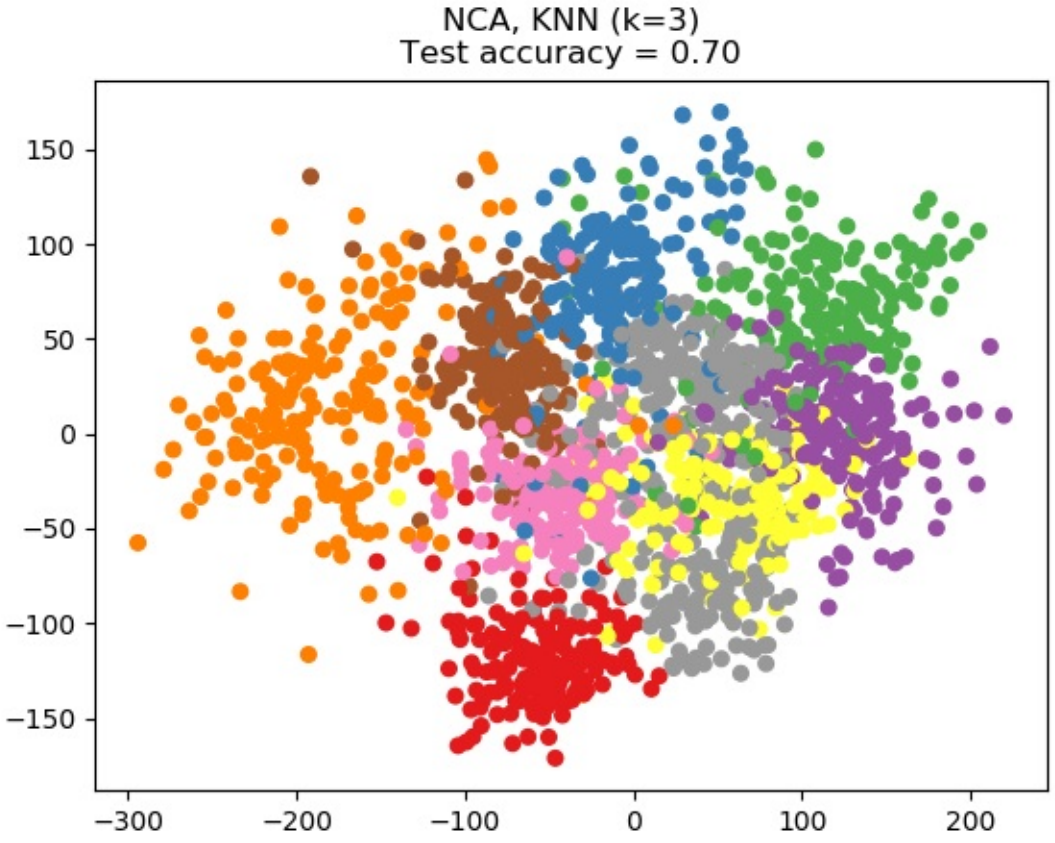
图中仅对鸢尾花数据集上的两个特征进行训练和评分，显示最近邻分类和邻域成分分析分类的决策边界，可以直观的看到差异。

1.6.6.2. 降维

NCA可用于进行监督降维。输入数据被投影到一个由最小化NCA目标的方向组成的线性子空间上。可以使用参数 `n_components` 设置所需的维数。例如，下图显示了主成分分析的降维(`sklearn.decomposition.PCA`)，线性判别分析(`sklearn.discriminant_analysis.LinearDiscriminantAnalysis`)和邻域成分分析(`NeighborhoodComponentsAnalysis`)在一个64个特征，1797个样本的数字数据集的降维结果。数据集被划分为大小相同的训练集和测试集，然后进行标准化。为了评价该方法的分类精度，对每种方法找到的二维投影点进行了3-最近邻分类精度的计算。每个数据样本属于10个类中的一个。







> 示例：
>
> - Comparing Nearest Neighbors with and without Neighborhood Components Analysis

- Dimensionality Reduction with Neighborhood Components Analysis
- Manifold learning on handwritten digits: Locally Linear Embedding, Isomap…

1.6.6.3 数学公式

NCA的目标是拟合一个尺寸为(n_components, n_features)的最优线性变换矩阵，使所有被正确分类的概率样本的和最大，即：

$$\arg\max_{L} \sum_{i=0}^{N-1} p_i$$

其中N是样本数，p_i是第i个样本在学习的嵌入空间中，根据随机近邻规则正确分类的可能性：

$$p_i = \sum_{j \in C_i} p_{ij}$$

其中C_i是第i个样本被分类到的点集，P_{ij}为嵌入空间中欧氏距离上的归一化指数（softmax）值：

$$p_{ij} = \frac{\exp(-||Lx_i - Lx_j||^2)}{\sum_{k \neq i} \exp -(||Lx_i - Lx_k||^2)}, \quad p_{ii} = 0$$

1.6.6.3.1 Mahalanobis距离

NCA可以看作是拟合一个(平方)Mahalanobis距离矩阵：

其中$||L(x_i - x_j)||^2 = (x_i - x_j)^T M (x_i - x_j),$

$M = L^T L$是大小为(特征数，特征数)对称正半定矩阵.

1.6.6.4 实现

该实现遵循了原始论文[1]中解释的内容。对于优化方法，目前使用的是scipy的L-BFGS-B，每次迭代都进行全梯度计算，以避免调整学习速度，提供稳定的拟合。

请参阅下面的示例和NeighborhoodComponentsAnalysis的文档字符串。以获取更多信息。

1.6.6.5 复杂度

1.6.6.5.1 训练

NCA存储一对距离矩阵，占用了（n_samples ** 2）的内存。时间复杂度取决于优化算法的迭代次数。但是，可以使用参数 `max_iter` 设置迭代的最大次数。对于每个迭代，时间复杂度为 `O(n_components x n_samples x min(n_samples, n_features))` 。

1.6.6.5.2 变形

这里变形操作返回值为LX^T，因此它的时间复杂度等于 `n_components x n_features x n_samples_test` 。操作中没有增加空间复杂度。

> 参考资料：
>
> - [1] “Neighbourhood Components Analysis”. Advances in Neural Information”, J. Goldberger, G. Hinton, S. Roweis, R. Salakhutdinov, Advances in Neural Information Processing Systems, Vol. 17, May 2005, pp. 513-520.
> - [2] Wikipedia entry on Neighborhood Components Analysis

我们一直在努力

apachecn/sklearn-doc-zh



1.7. 高斯过程

校验者：@glassy@Trembleguy@Loopy翻译者：@AI追寻者

高斯过程（GP）是一种常用的监督学习方法，旨在解决回归问题和概率分类问题。

高斯过程模型的优点如下：

- 预测内插了观察结果（至少对于正则核）。
- 预测结果是概率形式的（高斯形式的）。这样的话，人们可以计算得到经验置信区间并且据此来判断是否需要修改（在线拟合，自适应）在一些区域的预测值。
- 通用性： 可以指定不同的:内核(kernels)。虽然该函数提供了常用的内核，但是也可以指定自定义内核。

高斯过程模型的缺点包括：

- 它们不稀疏，例如，模型通常使用整个样本/特征信息来进行预测。
- 高维空间模型会失效，高维也就是指特征的数量超过几十个。

1.7.1. 高斯过程回归（GPR）

`GaussianProcessRegressor` 类实现了回归情况下的高斯过程(GP)模型。 为此，需要实现指定GP的先验。当参数 `normalizey=False` 时，先验的均值 通常假定为常数或者零; 当 `normalize_y=True` 时，先验均值通常为训练数据的均值。而先验的方差通过传递 *内核(kernel)* 对象来指定。通过 最大化基于传递 `optimizer` 的对数边缘似然估计*(LML)*，内核的超参可以在 *GaussianProcessRegressor* 类执行拟合过程中被优化。由于 *LML* 可能会存在多个 局部最优解，因此优化过程可以通过指定 `n_restarts_optimizer` 参数进行 多次重复。通过设置内核的超参初始值来进行第一次优化的运行。后续的运行 过程中超参值都是从合理范围值中随机选取的。如果需要保持初始化超参值， 那么需要把优化器设置为 *None* 。

目标变量中的噪声级别通过参数 `alpha` 来传递并指定，要么全局是常数要么是一个数据点。 请注意，适度的噪声水平也可以有助于处理拟合期间的数字问题，因为它被有效地实现为吉洪诺夫正则化(Tikhonov regularization)， 即通过将其添加到核心矩阵的对角线。明确指定噪声水平的替代方法是将 WhiteKernel 组件包含在内核中， 这可以从数据中估计全局噪声水平（见下面的示例）。

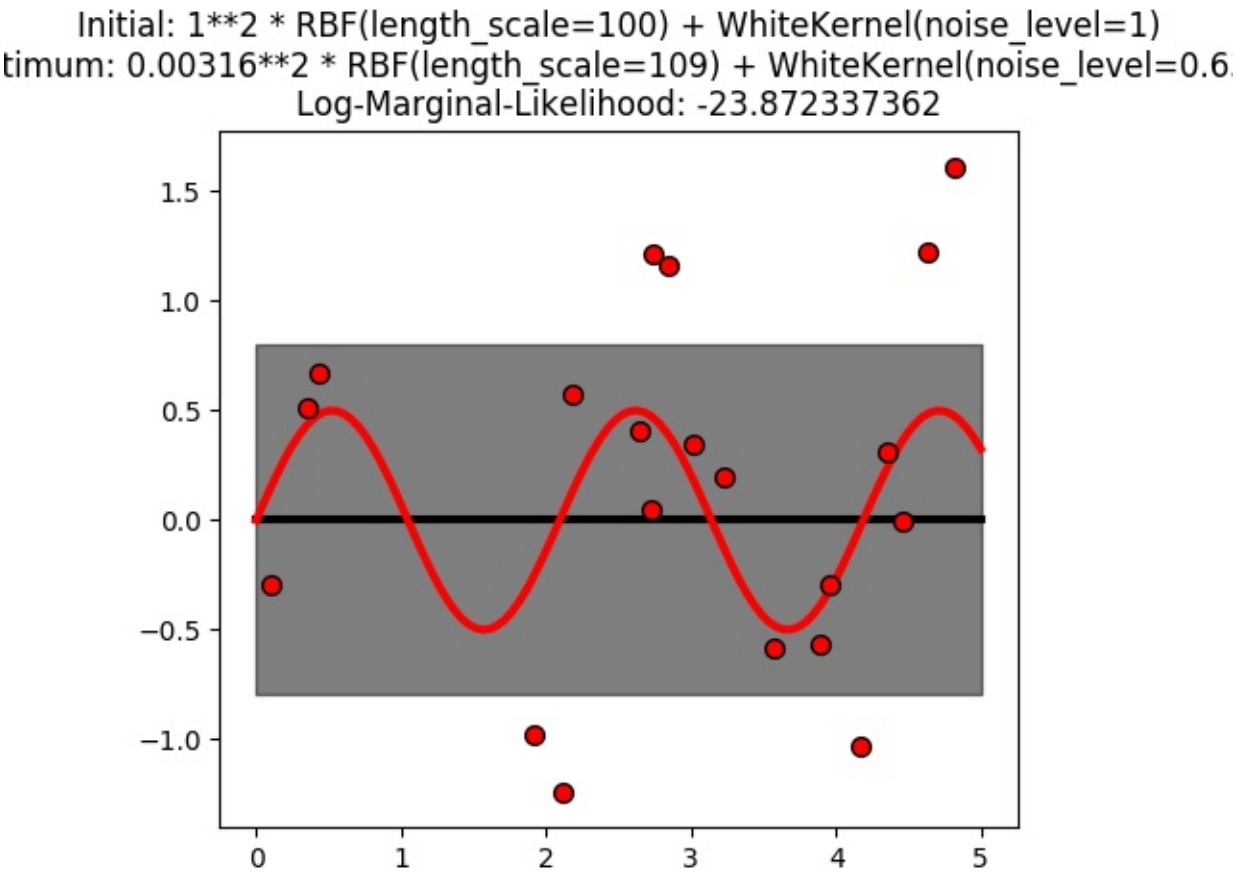
算法实现是基于[RW2006]中的算法 2.1 。除了标准 scikit learn 估计器的 API 之外，GaussianProcessRegressor 的作用还包括：

- 允许预测，无需事先拟合（基于GP先验）
- 提供了一种额外的方法 `sample_y(X)` ，其评估 在给定输入处从 GPR （先验或后验）绘制的样本
- 公开了一种方法 `log_marginal_likelihood(theta)` ，可以在外部使用其他方式选择超参数，例如通过马尔科夫链蒙特卡罗链(Markov chain Monte Carlo)。

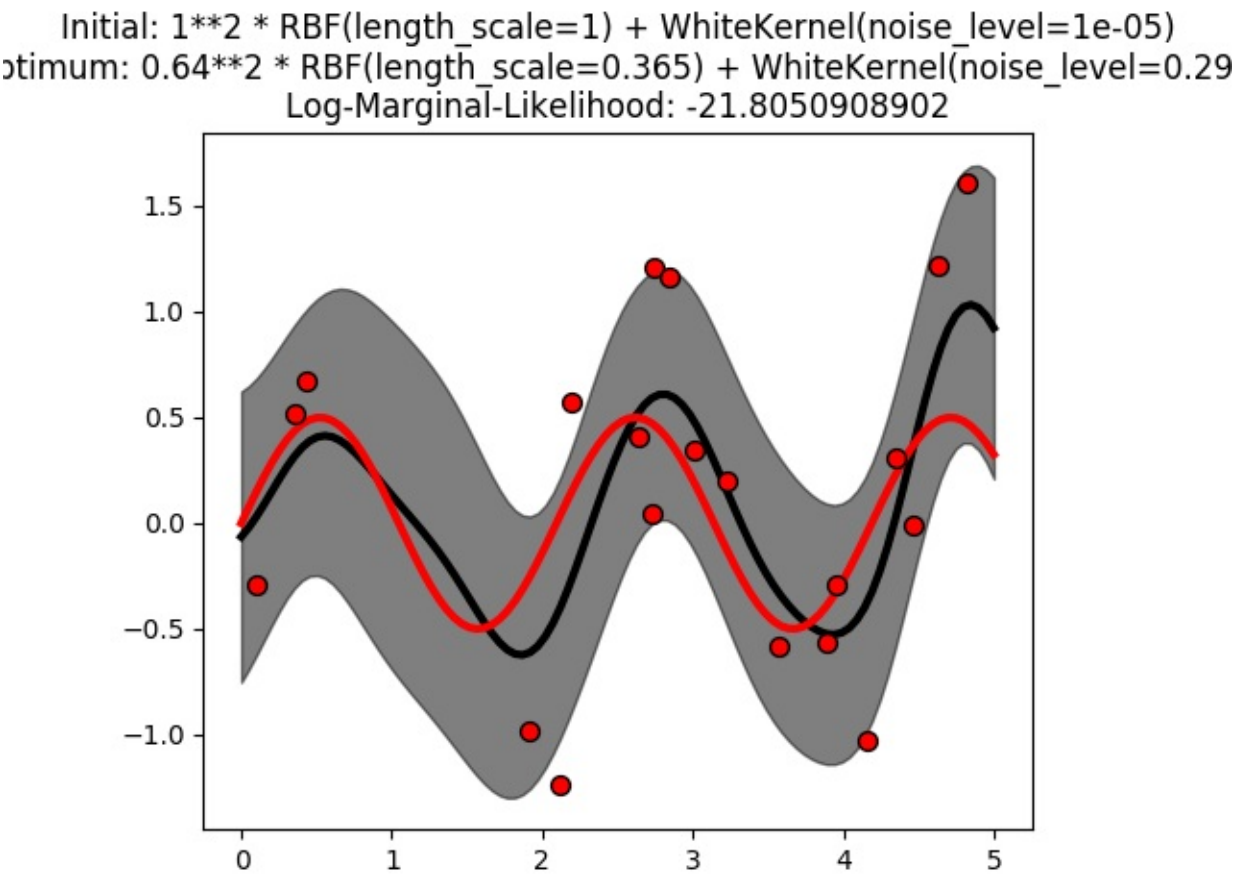
1.7.2. GPR 示例

1.7.2.1. 具有噪声级的 GPR 估计

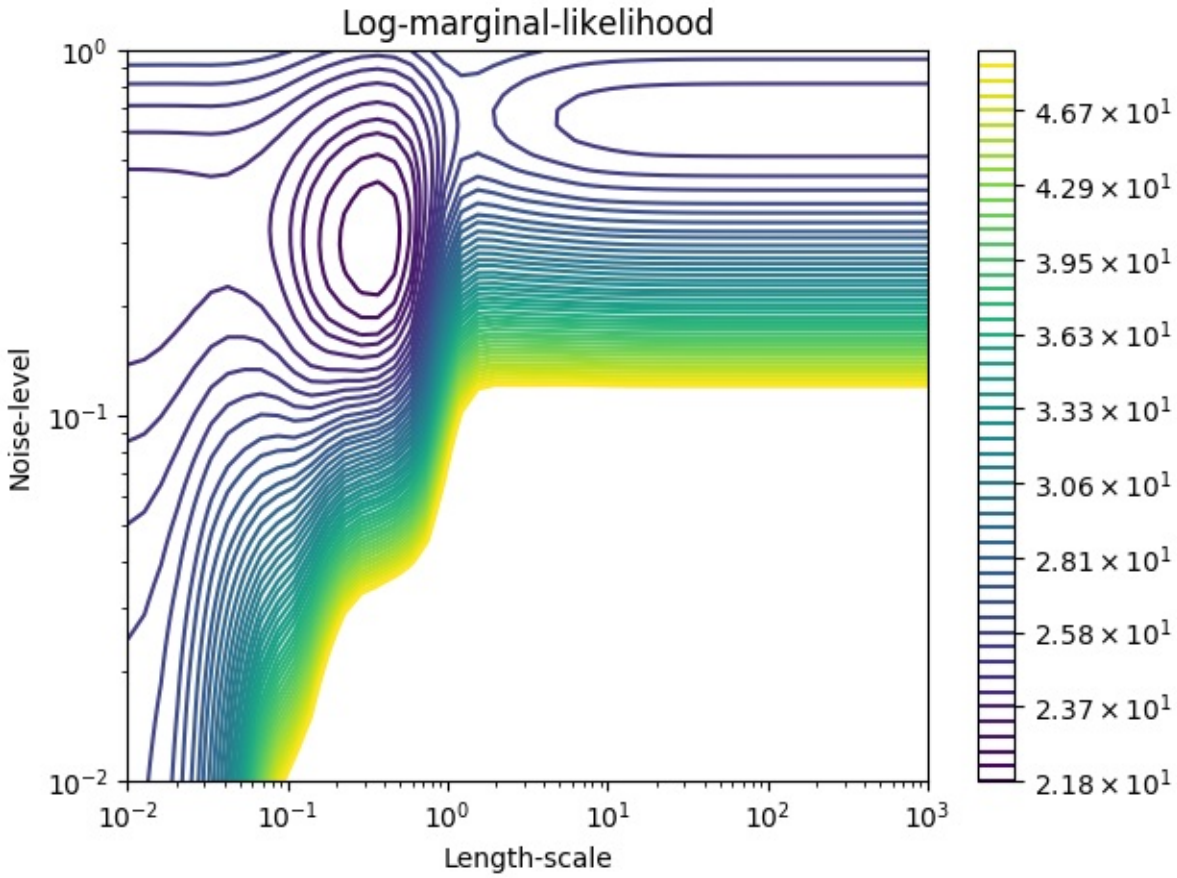
该示例说明具有包含 WhiteKernel 的和核(sum-kernel)的 GPR 可以估计数据的噪声水平。 对数边缘似然（LML）景观的图示表明存在 LML 的两个局部最大值。



第一个对应于具有高噪声电平和大长度尺度的模型，其解释数据中噪声的所有变化。



第二个具有较小的噪声水平和较短的长度尺度，这解释了无噪声功能关系的大部分变化。 第二种模式有较高的可能性； 然而，根据超参数的初始值，基于梯度的优化也可能会收敛到高噪声解。 因此，对于不同的初始化，重复优化多次是很重要的。

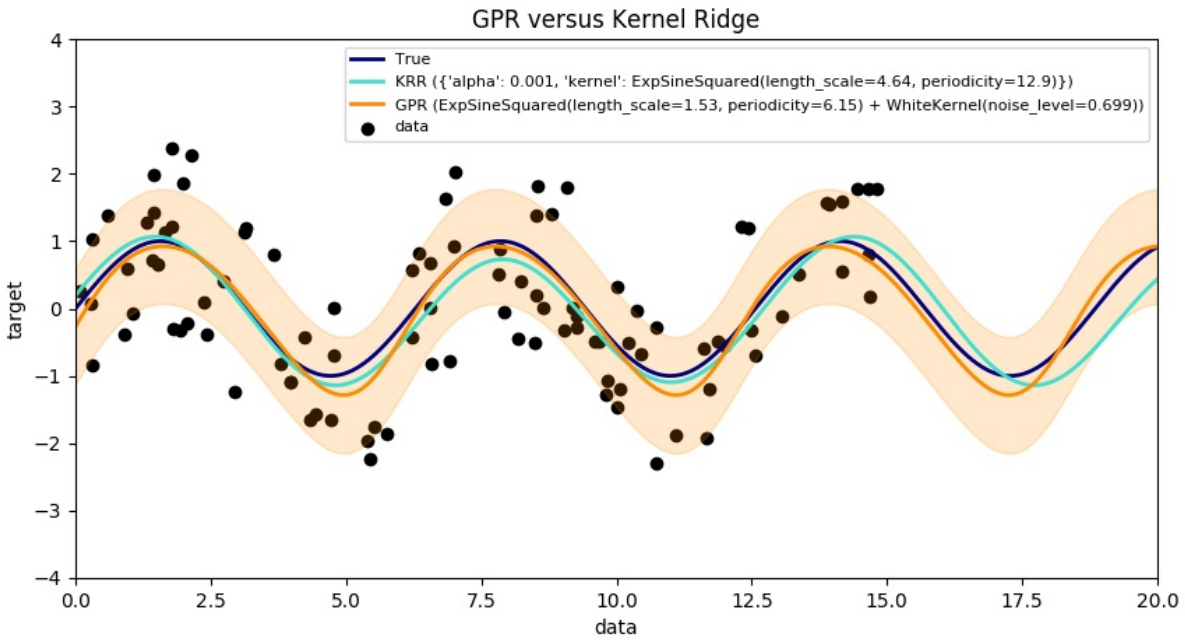


1.7.2.2. GPR 和内核岭回归（Kernel Ridge Regression）的比较

内核脊回归（KRR）和 GPR 通过内部使用 “kernel trick(内核技巧)” 来学习目标函数。 KRR学习由相应内核引起的空间中的线性函数，该空间对应于原始空间中的非线性函数。 基于平均误差损失与脊正弦化，选择内核空间中的线性函数。 GPR使用内核来定义先验分布在目标函数上的协方差，并使用观察到的训练数据来定义似然函数。 基于贝叶斯定理，定义了目标函数上的（高斯）后验分布，其平均值用于预测。

一个主要区别是，GPR 可以基于边际似然函数上的梯度上升选择内核的超参数， 而KRR需要在交叉验证的损失函数（均方误差损失）上执行网格搜索。 另一个区别是，GPR 学习目标函数的生成概率模型，因此可以提供有意义的置信区间和后验样本以及预测值， 而KRR仅提供预测。

下图说明了人造数据集上的两种方法，其中包括正弦目标函数和强噪声。 该图比较了基于 ExpSineSquared 内核的 KRR 和 GPR 的学习模型，适用于学习周期函数。 内核的超参数控制内核的平滑度（length_scale）和周期性（周期性）。 此外，数据的噪声水平由 GPR 通过内核中的另外的 WhiteKernel 组件和 KRR 的正则化参数 α 明确地学习。



该图显示，两种方法都可以学习合理的目标函数模型。 GPR将函数的周期正确地识别为 $2 * \pi$ （6.28），而 KRR 选择倍增的周期为 $4 * \pi$ 。 此外，GPR 为 KRR 不可用的预测提供了合理的置信区间。 两种方法之间的主要区别是拟合和预测所需的时间： 原则上KRR的拟合速度较快，超参数优化的网格搜索与超参数（ “curse of dimensionality(维度诅咒)” ）呈指数级关系。 GPR中的参数的基于梯度的优化不受此指数缩放的影响，因此在具有三维超参数空间的该示例上相当快。 预测的时间是相似的； 然而，生成 GPR 预测分布的方差需要的时间比生成平均值要长。

1.7.2.3. Mauna Loa CO2 数据中的 GRR

该示例基于[RW2006]的第 5.4.3 节。 它演示了使用梯度上升的对数边缘似然性的复杂内核工程和超参数优化的示例。 数据包括在 1958 年至 1997 年间夏威夷 Mauna Loa 天文台收集的每月平均大气二氧 化碳浓度（以百万分之几（ppmv）计）。目的是将二氧化碳浓度建模为时间t的函数。

内核由几个术语组成，负责说明信号的不同属性：

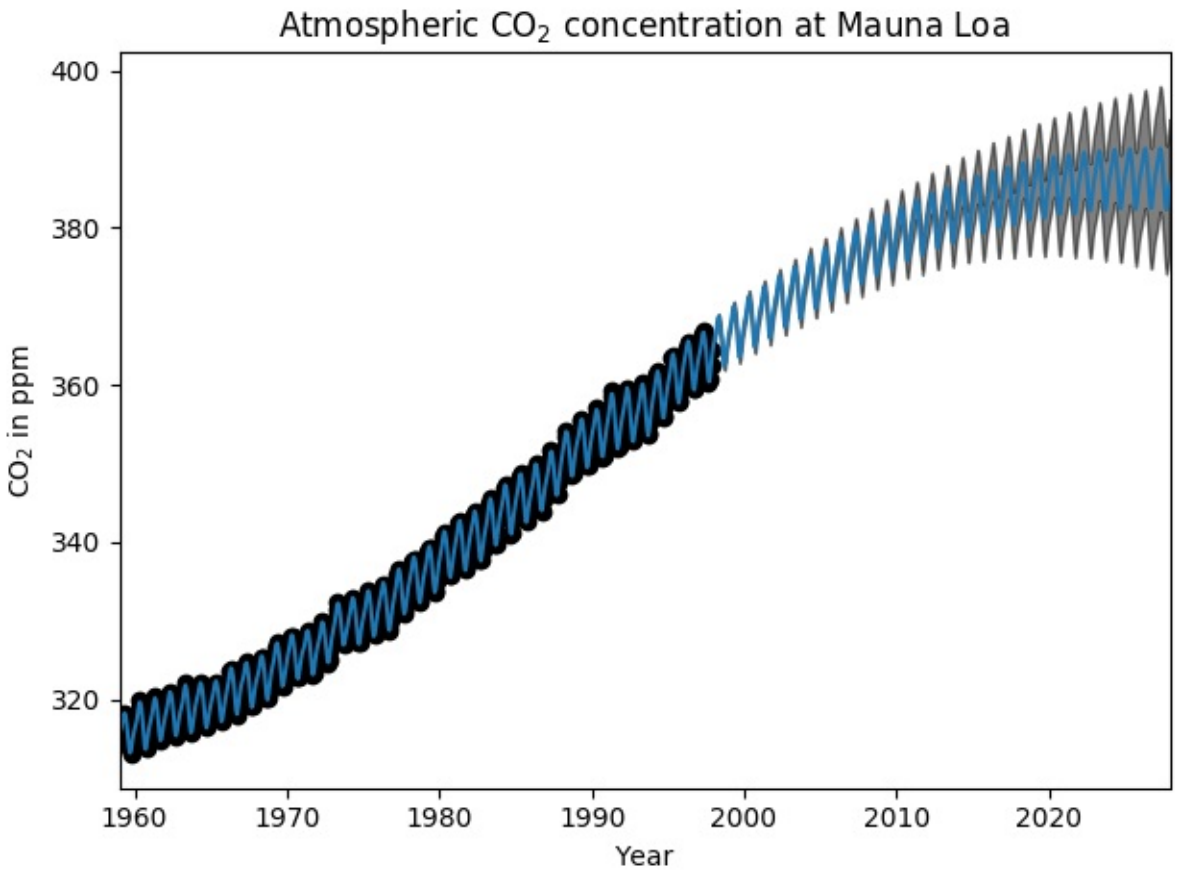
- 一个长期的，顺利的上升趋势是由一个 RBF 内核来解释的。 具有较大长度尺寸的RBF内核将使该分量平滑；没有强制这种趋势正在上升，这给 GP 带来了这个选择。 具体的长度尺度和振幅是自由的超参数。
- 季节性因素，由定期的 ExpSineSquared 内核解释，固定周期为1年。 该周期分量的长度尺度控制其平滑度是一个自由参数。 为了使准确周期性的衰减，采用带有RBF内核的产品。 该RBF组件的长度尺寸控制衰减时间，并且是另一个自由参数。
- 较小的中期不规则性将由 RationalQuadratic 内核组件来解释， RationalQuadratic 内核组件的长度尺度和 alpha 参数决定长度尺度的扩散性。 根据[RW2006]，这些不规则性可以更好地由 RationalQuadratic 来解释， 而不是 RBF 内核组件，这可能是因为它可以容纳几个长度尺度。
- “noise(噪声)” 一词，由一个 RBF 内核贡献组成，它将解释相关的噪声分量，如局部天气现象以及 WhiteKernel 对白噪声的贡献。 相对幅度和RBF的长度尺度是进一步的自由参数。

在减去目标平均值后最大化对数边际似然率产生下列内核，其中LML为-83.214：

```
1. 34.4**2 * RBF(length_scale=41.8)
```

```
    + 3.27**2 * RBF(length_scale=180) * ExpSineSquared(length_scale=1.44,
                                                       periodicity=1)
    + 0.446**2 * RationalQuadratic(alpha=17.7, length_scale=0.957)
    + 0.197**2 * RBF(length_scale=0.138) + WhiteKernel(noise_level=0.0336)
```

因此，大多数目标信号（34.4ppm）由长期上升趋势（长度为41.8年）解释。 周期分量的振幅为3.27ppm，衰减时间为180年，长度为1.44。 长时间的衰变时间表明我们在当地非常接近周期性的季节性成分。 相关噪声的幅度为0.197ppm，长度为0.138年，白噪声贡献为0.197ppm。 因此，整体噪声水平非常小，表明该模型可以很好地解释数据。 该图还显示，该模型直到2015年左右才能做出置信度比较高的预测



1.7.3. 高斯过程分类（GPC）

所述 `GaussianProcessClassifier` 器实现了用于分类目的的高斯过程（GP），当测试的预测采用类概率的形式，更能够用于概率分类。 GaussianProcessClassifier 在隐函数 f 之前设置GP先验，然后通过链接函数进行压缩以获得概率分类。 隐函数 f 因此就是所谓的干扰函数(nuisance function)，其值不能被观测到，并且自身不具有相关性。 其目的是允许模型的表达形式更加简便，并且 f 在预测过程中被去除（整合）。 GaussianProcessClassifier 实现了逻辑链接函数， 对于该逻辑，积分不能在分析上计算，但在二进制情况下很容易近似。

与回归设置相反，即使设置了高斯过程先验，隐函数 f 的后验也不符合高斯分布， 因为高斯似然不适用于离散类标签。相反，使用的是与逻辑链接函数（logit）对应的非高斯似然。 GaussianProcessClassifier 通过拉普拉

斯近似(Laplace approximation)来估计非高斯后验分布。 更多详细信息，请参见[RW2006]的第 3 章。

GP先验平均值假定为零。先验的协方差是通过传递 内核(kernel) 对象来指定的。 在通过最大化基于传递的对数边缘似然（LML）的 GaussianProcessRegressor 拟合期间， 优化内核的超参数 `optimizer` 。由于LML可能具有多个局部最优值， 所以优化器可以通过指定重复启动 `nrestarts_optimizer` 。 第一次运行始终从内核的初始超参数值开始执行；从已经从允许值的范围中随机选择超参数值来进行后续运行。 如果初始超参数需要保持固定，*None* 可以传递作为优化器。

`GaussianProcessClassifier` 通过执行基于OvR(one-versus-rest)或 OvO(one-versus-one)策略的训练和预测来支持多类分类。 在OvR(one-versus-rest)策略中，每个类都配有一个二进制高斯过程分类器，该类别被训练为将该类与其余类分开。 在 “one_vs_one” 中，对于每对类拟合一个二进制高斯过程分类器，这被训练为分离这两个类。 这些二进制预测因子的预测被组合成多类预测。更多详细信息，请参阅 多类别分类 。

在高斯过程分类的情况下，”one_vs_one” 策略可能在计算上更廉价， 因为它必须解决涉及整个训练集的每一个子集的许多问题， 而不是整个数据集的较少的问题。由于高斯过程分类与数据集的大小相互立方，这可能要快得多。但是，请注意，”one_vs_one” 不支持预测概率估计，而只是简单的预测。 此外，请注意，`GaussianProcessClassifier` 在内部还没有实现真正的多类 Laplace 近似， 但如上所述，在解决内部二进制分类任务的基础上，它们使用OvR或OvO的组合方法。

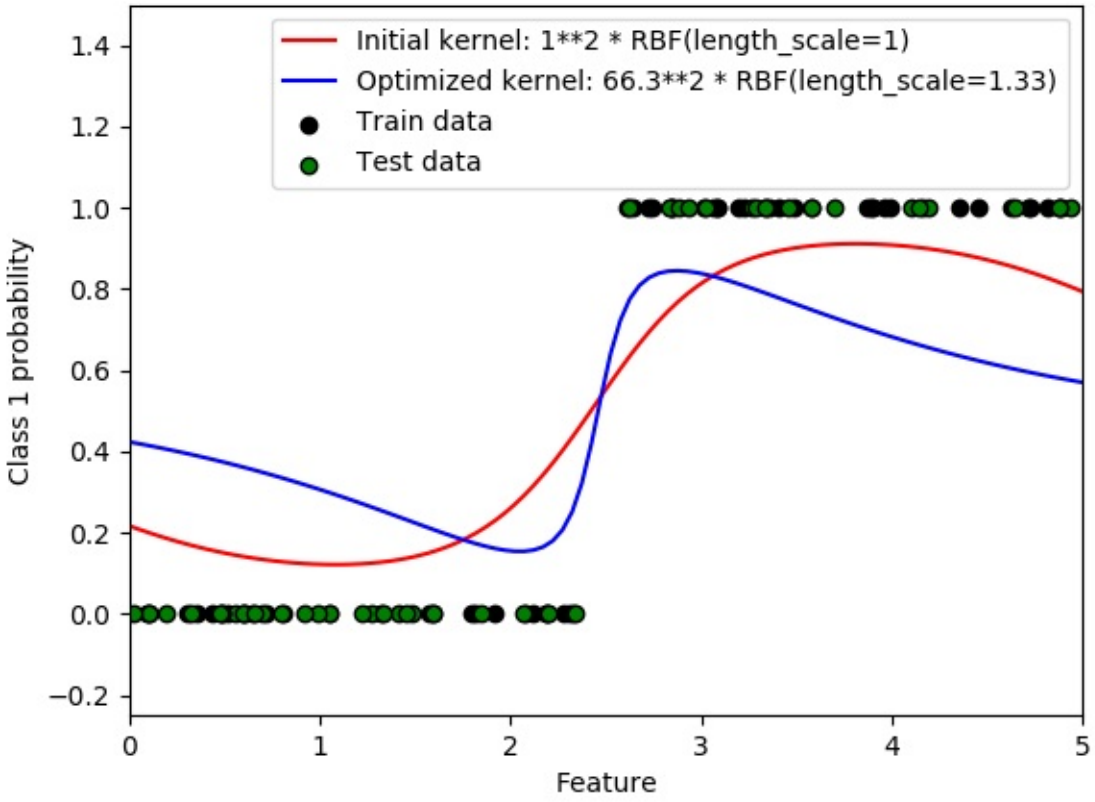
1.7.4. GPC 示例

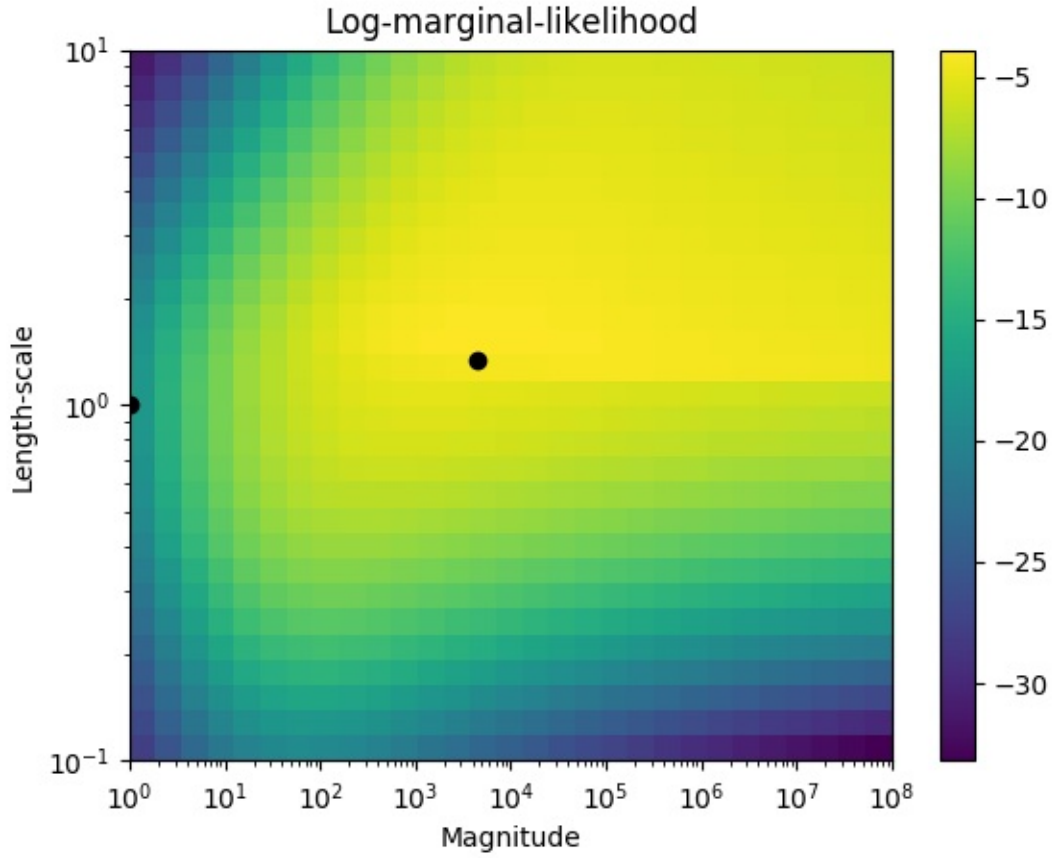
1.7.4.1. GPC 概率预测

该示例说明了对于具有不同选项的超参数的RBF内核的GPC预测概率。 第一幅图显示GPC具有任意选择的超参数的预测概率，以及对应于最大LML（对数边缘似然）对应的超参数。

虽然通过优化LML选择的超参数具有相当大的LML，但是根据测试数据的对数损失，它们的表现更差。 该图显示，这是因为它们在阶级边界（这是好的）表现出类概率的急剧变化， 但预测概率接近0.5远离类边界（这是坏的）这种不良影响是由于GPC内部使用了拉普拉斯逼近。

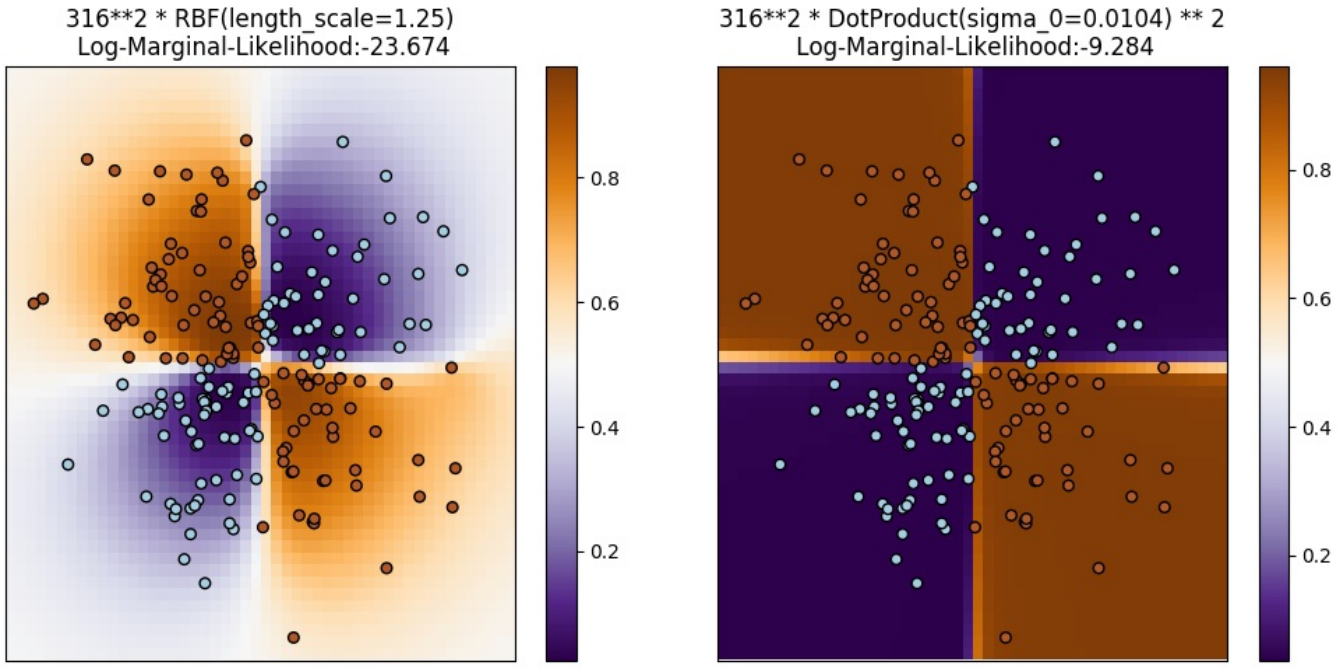
第二幅图显示了内核超参数的不同选择的LML（对数边缘似然），突出了在第一幅图中使用的通过黑点（训练集）选择的两个超参数。





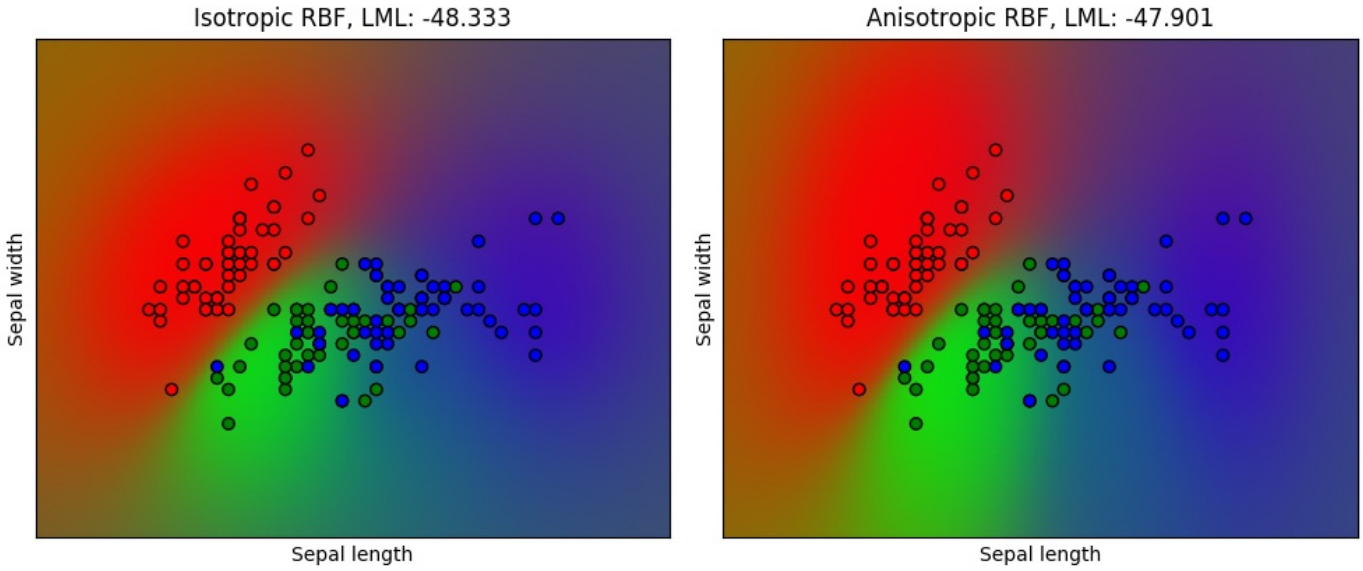
1.7.4.2. GPC 在 XOR 数据集上的举例说明

此示例说明了在XOR数据上的GPC。各向同性的核（ `RBF` ）和非固定的核（ `DotProduct` ）对比固定性。 在这个特定的数据集上， *DotProduct* 内核获得了更好的结果，因为类边界是线性的，与坐标轴重合。 然而，实际上，诸如 `RBF` 这样的固定内核经常获得更好结果。



1.7.4.3. iris 数据集上的高斯过程分类（GPC）

该示例说明了用于虹膜数据集的二维版本上各向同性和各向异性RBF核的GPC的预测概率。 这说明了GPC对多类分类的适用性。 各向异性RBF内核通过为两个特征维度分配不同的长度尺度来获得稍高的LML（对数边缘似然）。



1.7.5. 高斯过程内核

内核（也可以叫做GPs上下文中的”协方差函数”） 是决定高斯过程（GP）先验和后验形状的关键组成部分。 它们通过定义两个数据点的“相似性”，并结合相似的 数据点应该具有相似的目标值的假设，对所学习的函数进行编码。 内核可以分为两类：固定内核，只取决于两个数据点的距离， 不依赖于它们的绝对值 $k(x_i, x_j) = k(d(x_i, x_j))$ ，因此它们对于输入空间中的转换是不变的；非固定的内核，取 决于数据点的具体值。固定内核可以进一步细分为各向同性和各向 异性内核，其中各向同性内核不会在输入空间中旋转。想要了解 更多细节，请参看 [RW2006] 的第四章。

1.7.5.1. 高斯过程内核 API

`Kernel` 主要是用来计算数据点之间的高斯过程协方差。 为此，内核中 `call` 方法会被调用。该方法可以用于计算 2d阵列X中所有数据点对的“自动协方差”，或二维阵列X的数据点 与二维阵列Y中的数据点的所有组合的“互协方差”。以下论断对于 所有内核k（除了 `WhiteKernel` ）都是成立的： `k(X) == K(X, Y=X)` 。 如果仅仅是自协方差的对角线元素被使用，那么内核的方法 `diag()` 将会被调用， 该方法比等价的调用 `call` ： `np.diag(k(X, X)) == k.diag(X)` 具有更高的计算效率。

内核通过超参数向量 θ 进行参数化。这些超参数可以 控制例如内核的长度或周期性（见下文）。通过设置 `call` 方法的参数 `eval_gradient=True` ，所有的内核支持计算解析 内核自协方差对于 θ 的解析梯度。该梯度被用来在高斯过程中（不论是回归型还是分类型的）计算LML（对数边缘似然）函数 的梯度，进而被用来通过梯度下降的方法极大化LML（对数边缘似然）函数 从而确定 θ 的值。对于每个超参数，当对内核的实例 进行赋值时，初始值和边界值需要被指定。通过内核对象属性 `theta` ， θ 的当前值可以被获取或者设置。更重要的是， 超参的边界值可以被内核属性 `bounds` 获取。需要注意的是， 以上两种属性值(theta和bounds)都会返回内部使用值的日志转换值， 这是因为这两种属性值通常更适合基于梯度的优化。每个超参数的 规范 `Hyperparameter` 以实例形式被存储在相应内核中。 请注意使用了以”x”命名的超参的内核必然具有self.x和self.x_bounds这两种属性。

所有内核的抽象基类为 `Kernel` 。Kernel 基类实现了 一个相似的接口 `Estimator` ，提供了方法 `getparams()` ， `setparams()` 以及 `clone()` 。这也允许通过诸如 `Pipeline` 或者 `GridSearch` 之类的元估计来设置内核值。 需要注意的是，由于内核的嵌套结构（通过内核操作符，如下所见）， 内核参数的名称可能会变得相对复杂些。通常来说，对于二元内核操作， 参数的左运算元以 `k1__` 为前缀，而右运算元以 `k2` 为前缀。 一个额外的便利方法是 `clone_with_theta(theta)` ， 该方法返回克隆版本的内核，但是设置超参数为 `theta` 。 示例如下：

```
>>> from sklearn.gaussian_process.kernels import ConstantKernel, RBF
>>> kernel = ConstantKernel(constant_value=1.0, constant_value_bounds=(0.0, 10.0)) * RBF(length_scale=0.5,
length_scale_bounds=(0.0, 10.0)) + RBF(length_scale=2.0, length_scale_bounds=(0.0, 10.0))
>>> for hyperparameter in kernel.hyperparameters: print(hyperparameter)
Hyperparameter(name='k1__k1__constant_value', value_type='numeric', bounds=array([[ 0., 10.]]), n_elements=1,
fixed=False)
Hyperparameter(name='k1__k2__length_scale', value_type='numeric', bounds=array([[ 0., 10.]]), n_elements=1,
fixed=False)
Hyperparameter(name='k2__length_scale', value_type='numeric', bounds=array([[ 0., 10.]]), n_elements=1,
fixed=False)
>>> params = kernel.get_params()
>>> for key in sorted(params): print("%s : %s" % (key, params[key]))
k1 : 1**2 * RBF(length_scale=0.5)
k1__k1 : 1**2
k1__k1__constant_value : 1.0
```

```
k1__k1__constant_value_bounds : (0.0, 10.0)
k1__k2 : RBF(length_scale=0.5)
k1__k2__length_scale : 0.5
k1__k2__length_scale_bounds : (0.0, 10.0)
k2 : RBF(length_scale=2)
k2__length_scale : 2.0
k2__length_scale_bounds : (0.0, 10.0)
>>> print(kernel.theta)  # Note: log-transformed
[ 0.         -0.69314718  0.69314718]
>>> print(kernel.bounds)  # Note: log-transformed
[[      -inf 2.30258509]
 [      -inf 2.30258509]
 [      -inf 2.30258509]]
```

所有的高斯过程内核操作都可以通过 `sklearn.metrics.pairwise` 来进行互操作，反之亦然。 `Kernel` 的子类实例可以通过 `metric` 参数传给 `sklearn.metrics.pairwise` 中的 `pairwise_kernels` 。更重要的是，超参数的梯度不是分析的，而是数字，所有这些内核只支持各向同性距离。该参数 `gamma` 被认为是一个超参数，可以进行优化。其他内核参数在初始化时直接设置， 并保持固定。

1.7.5.2. 基础内核

`ConstantKernel` 内核类可以被用作 `Product` 内核类的一部分， 在它可以对其他因子（内核）进行度量的场景下或者作为更改高斯过程均值的 `Sum` 类的一部分。这取决于参数 $constant_value$ 的设置。该方法定义为：

$$k(x_i, x_j) = constant_value \;\forall\; x_1, x_2$$

`WhiteKernel` 内核类的主要应用实例在于当解释信号的噪声部分时 可以作为内核集合的一部分。通过调节参数 $noise_level$， 该类可以用来估计噪声级别。具体如下所示：

$$k(x_i, x_j) = noise_level \text{ if } x_i == x_j \text{ else } 0$$

1.7.5.3. 内核操作

内核操作是把1~2个基内核与新内核进行合并。内核类 `Sum` 通过 $k_{sum}(X, Y) = k1(X, Y) + k2(X, Y)$ 相加来合并 $k1$ 和 $k2$ 内核。内核类 `Product` 通过 $k_{product}(X, Y) = k1(X, Y) * k2(X, Y)$ 把 $k1$ 和 $k2$ 内核进行合并。内核类 `Exponentiation` 通过 $k_{exp}(X, Y) = k(X, Y)^{\text{exponent}}$ 把基内核与 常量参数 $exponent$ 进行合并。

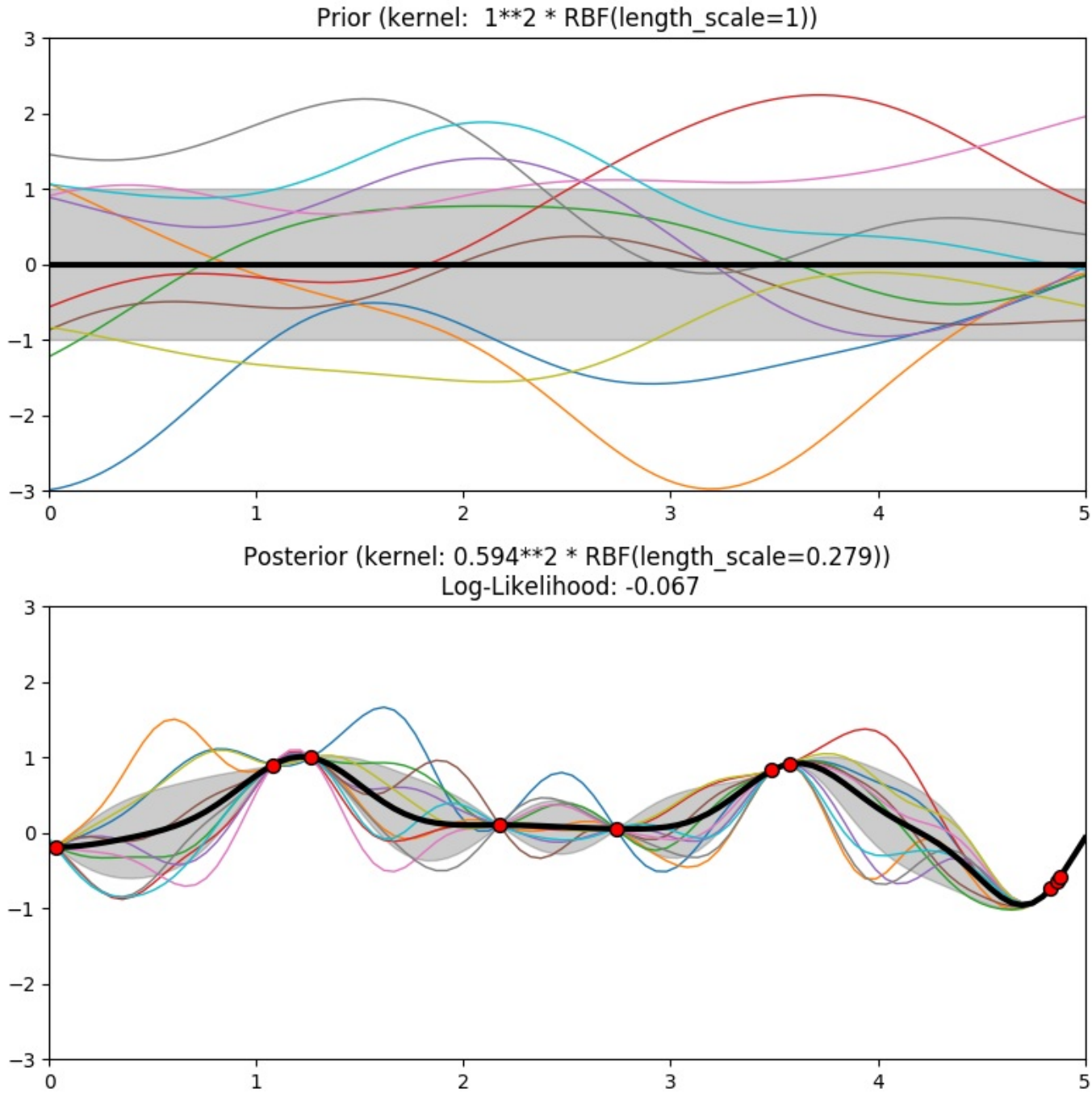
1.7.5.4. 径向基函数内核

`RBF` 内核是一个固定内核，它也被称为“平方指数”内核。它通过定长的参数 $l > 0$ 来对内核进行参数化。该参数

既可以是标量（内核的各向同性变体）或者与输入 x （内核的各向异性变体） 具有相同数量的维度的向量。该内核可以被定义为：

$$k(x_i, x_j) = \exp\left(-\frac{1}{2} d(x_i/l, x_j/l)^2\right)$$

这个内核是无限可微的，这意味着这个内核作为协方差函数的 GP 具有所有阶数的均方差导数， 因此非常平滑。由RBF内核产生的GP的先验和后验示意图如下所示：



1.7.5.5. Matérn 内核

`Matern` 内核是一个固定内核，是 `RBF` 内核的泛化。它有一个额外的参数 ν， 该参数控制结果函数的平滑程

度。它由定长参数 $l > 0$ 来实现参数化。该参数既可以是标量 （内核的各向同性变体）或者与输入 x （内核的各向异性变体）具有相同数量的维度的向量。 该内核可以被定义为：

$$k(x_i, x_j) = \sigma^2 \frac{1}{\Gamma(\nu)2^{\nu-1}} \left(\gamma\sqrt{2\nu} d(x_i/l, x_j/l) \right)^{\nu} K_{\nu} \left(\gamma\sqrt{2\nu} d(x_i/l, x_j/l) \right),$$

因为 $\nu \to \infty$ ，Matérn 内核收敛到 RBF 内核。 当 $\nu = 1/2$ 时，Matérn 内核变得与绝对指数内核相同时，即

$$k(x_i, x_j) = \sigma^2 \exp \left(- \gamma d(x_i/l, x_j/l) \right) \qquad \nu = \tfrac{1}{2}$$

特别的，当 $\nu = 3/2$ 时：

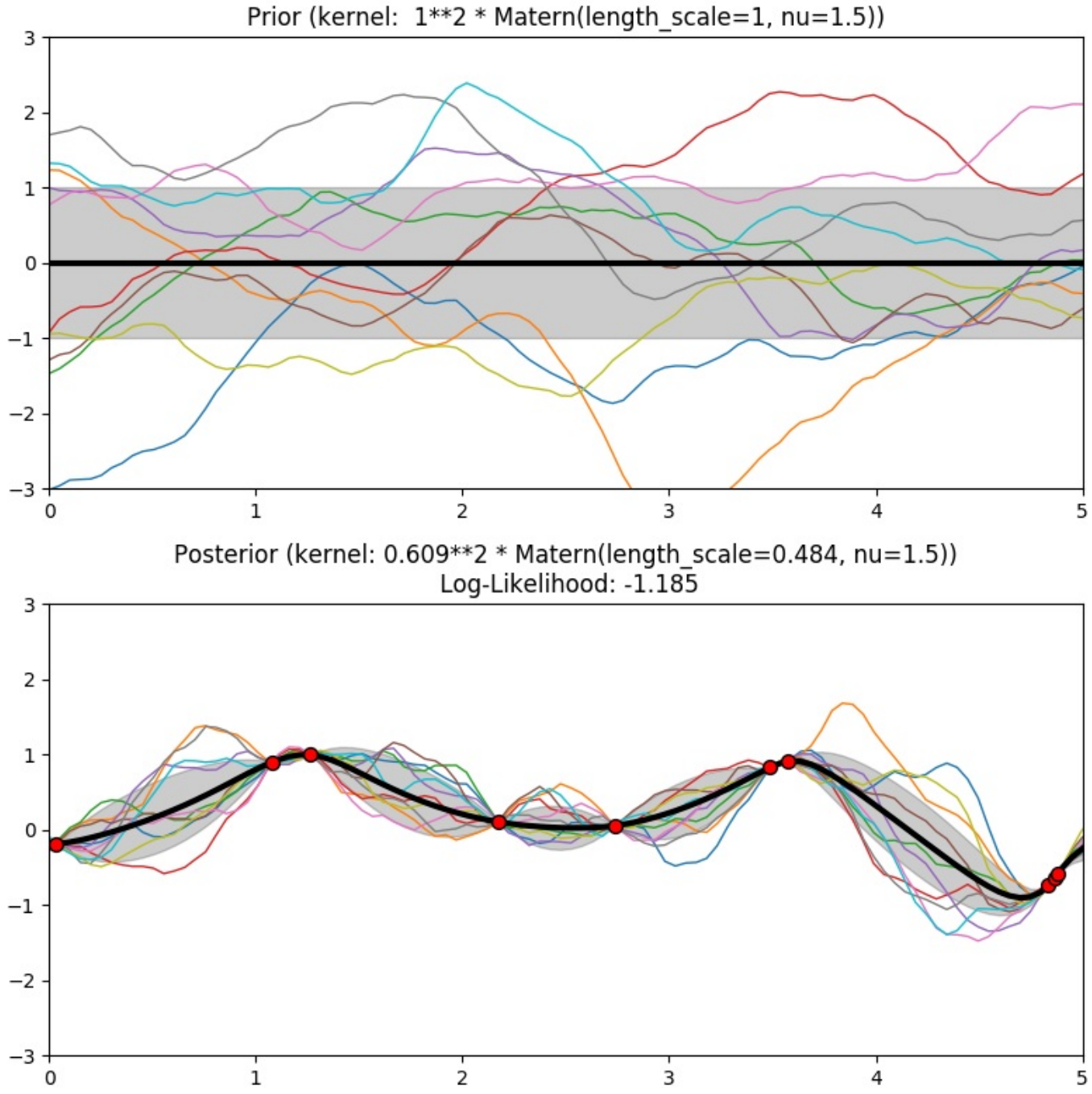
$$k(x_i, x_j) = \sigma^2 \left(1 + \gamma\sqrt{3} d(x_i/l, x_j/l) \right) \exp \left(- \gamma\sqrt{3} d(x_i/l, x_j/l) \right) \qquad \nu = \tfrac{3}{2}$$

和 $\nu = 5/2$ ：

$$k(x_i, x_j) = \sigma^2 \left(1 + \gamma\sqrt{5} d(x_i/l, x_j/l) + \frac{5}{3}\gamma^2 d(x_i/l, x_j/l)^2 \right) \exp \left(- \gamma\sqrt{5} d(x_i/l, x_j/l) \right) \qquad \nu = \tfrac{5}{2}$$

是学习函数的常用选择，并且不是无限可微的（由 RBF 内核假定） 但是至少具有一阶($\nu = 3/2$)或者二阶($\nu = 5/2$)可微性。

通过 ν 灵活控制学习函数的平滑性可以更加适应真正的底层函数关联属性。 通过 Matérn 内核产生的高斯过程的先验和后验如下图所示：



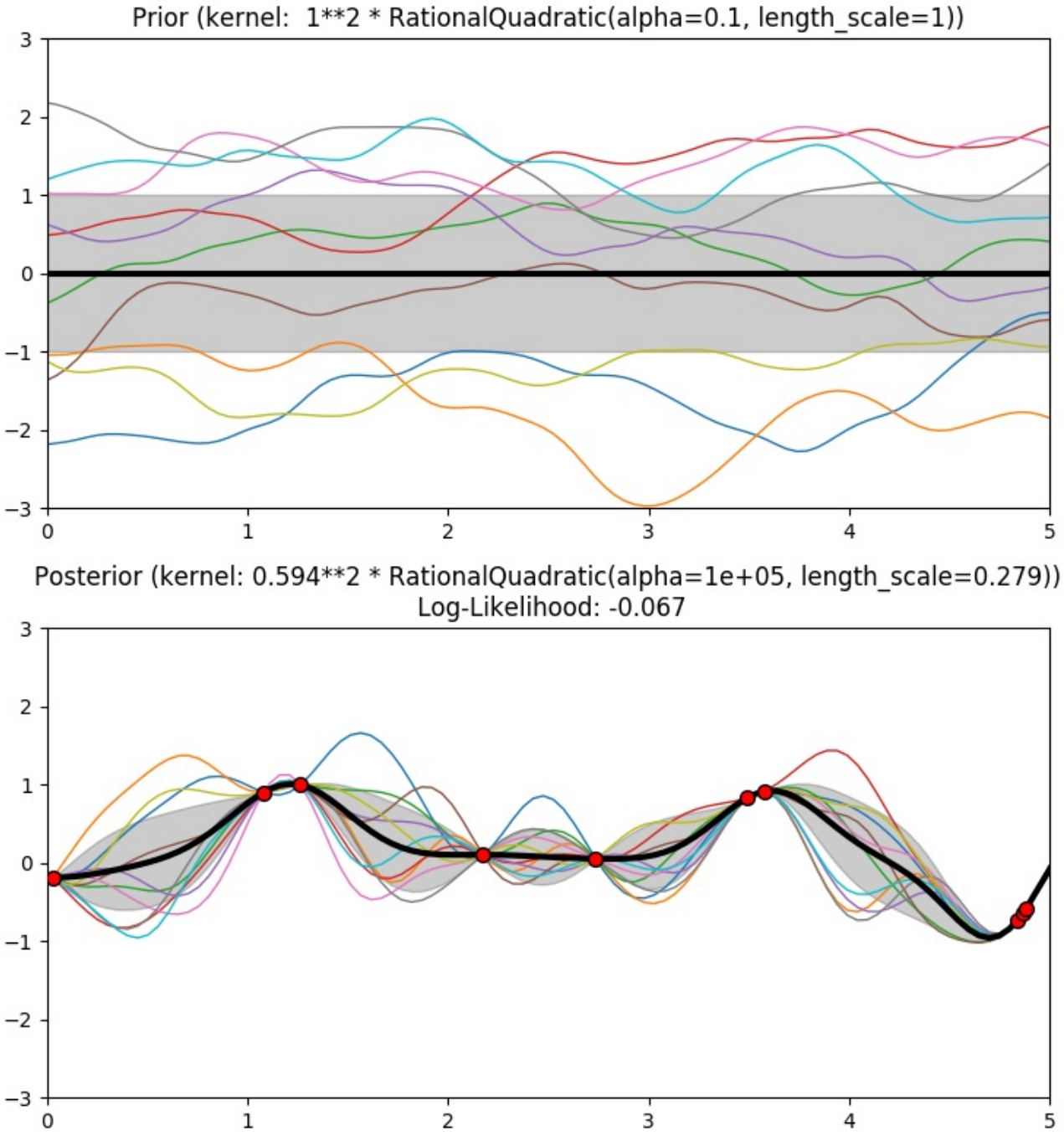
想要更进一步地了解不同类型的Matérn内核请参阅 [RW2006] , pp84。

1.7.5.6. 有理二次内核

`RationalQuadratic` 内核可以被看做不同特征尺度下的 `RBF` 内核的规模混合（一个无穷和） 它通过长度尺度参数 $l > 0$ 和比例混合参数 $\alpha > 0$ 进行参数化。 此时仅支持 l 标量的各向同性变量。内核公式如下：

$$k(x_i, x_j) = \left(1 + \frac{d(x_i, x_j)^2}{2\alpha l^2}\right)^{-\alpha}$$

从 RBF 内核中产生的高斯过程的先验和后验如下图所示：

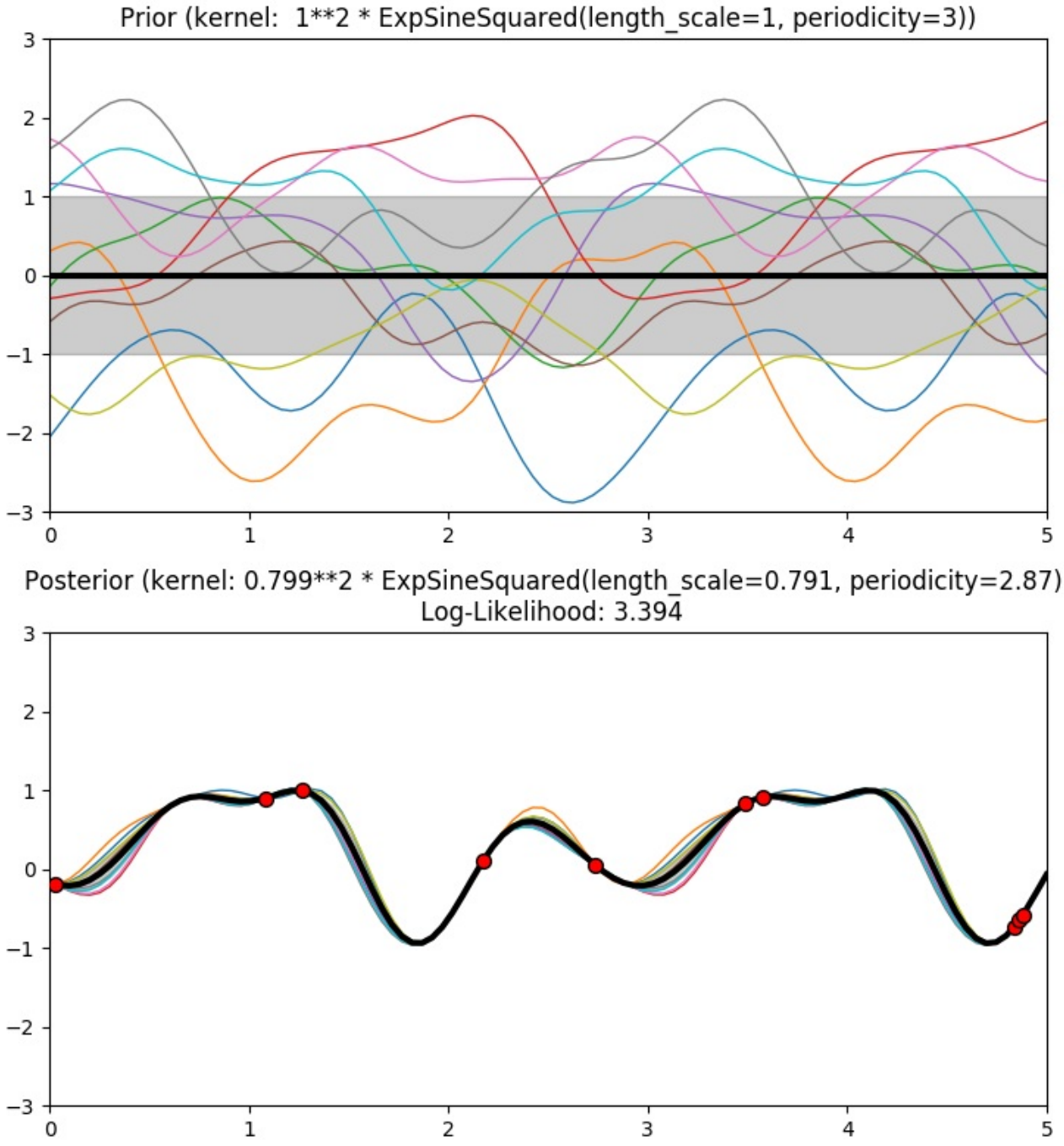


1.7.5.7. 正弦平方内核

`ExpSineSquared` 内核可以对周期性函数进行建模。它由定长参数 $l > 0$ 以及周期参数 $p > 0$ 来实现参数化。此时仅支持 l 标量的各向同性变量。内核公式如下：

$$k(x_i, x_j) = \exp\left(-2\left(\sin(\pi/p * d(x_i, x_j))/l\right)^2\right)$$

从ExpSineSquared内核中产生的高斯过程的先验和后验如下图所示：

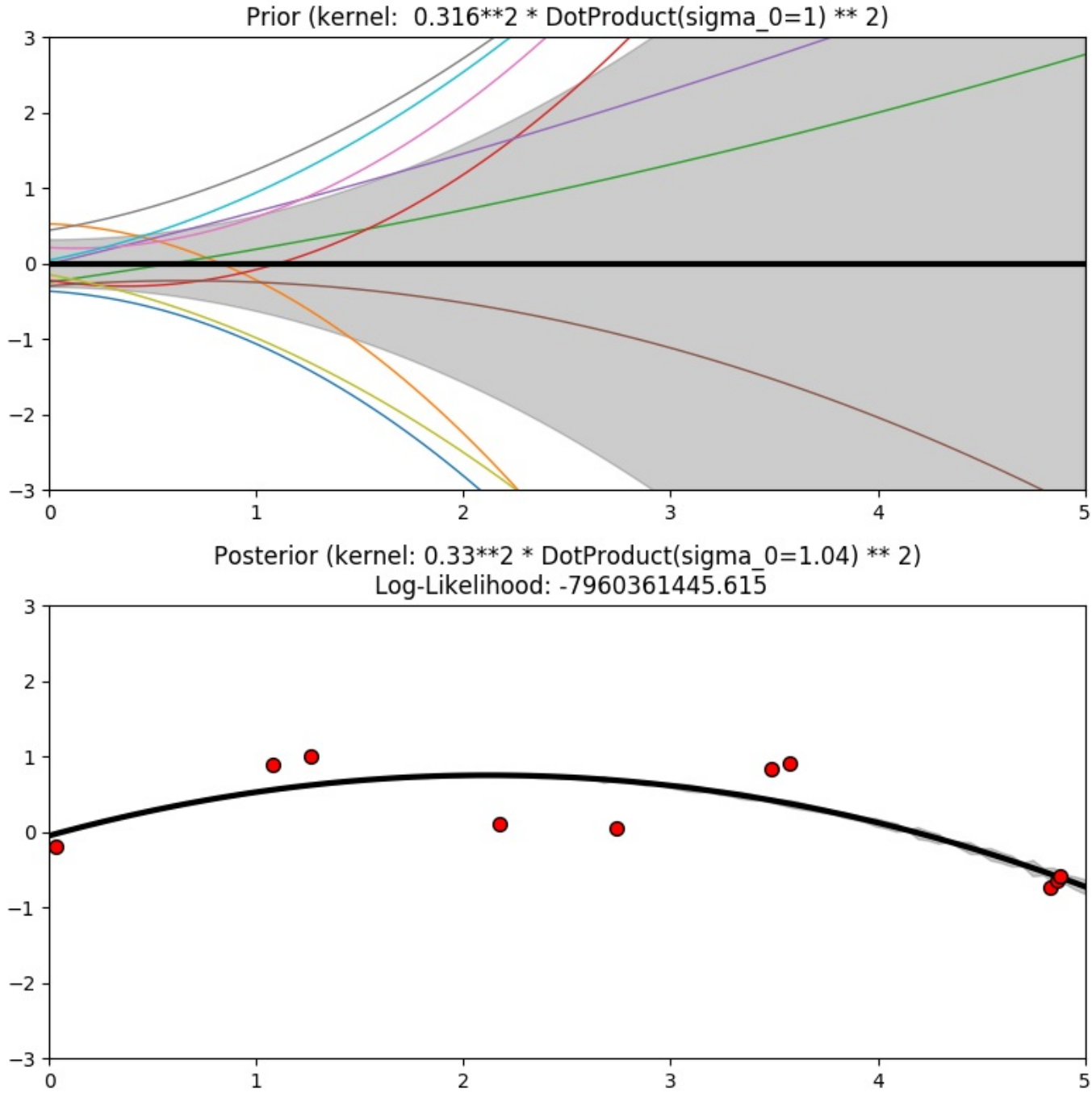


1.7.5.8. 点乘内核

`DotProduct` 内核是非固定内核，它可以通过在线性回归的 $x_d(d = 1, ..., D)$ 的相关系数上加上 服从于 $N(0, 1)$ 的先验以及在线性回归的偏置上加上服从于 $N(0, \sigma_0^2)$ 的先验来获得。 该 `DotProduct` 内核对于原点坐标的旋转是不变的，因此不是转换。它通过设置参数 σ_0^2 来进行参数化。 当 $\sigma_0^2 = 0$ 时，该内核叫做同质线性内核；否则该内核是非同质的。内核公式如下：

$$k(x_i, x_j) = \sigma_0^2 + x_i \cdot x_j$$

`DotProduct` 内核通常和指数分布相结合。实例如下图所示：



1.7.5.9. 参考文献

[RW2006] Carl Eduard Rasmussen and Christopher K.I. Williams, “Gaussian Processes for Machine Learning”, MIT Press 2006, Link to an official complete PDF version of the book here .

我们一直在努力

apachecn/sklearn-doc-zh

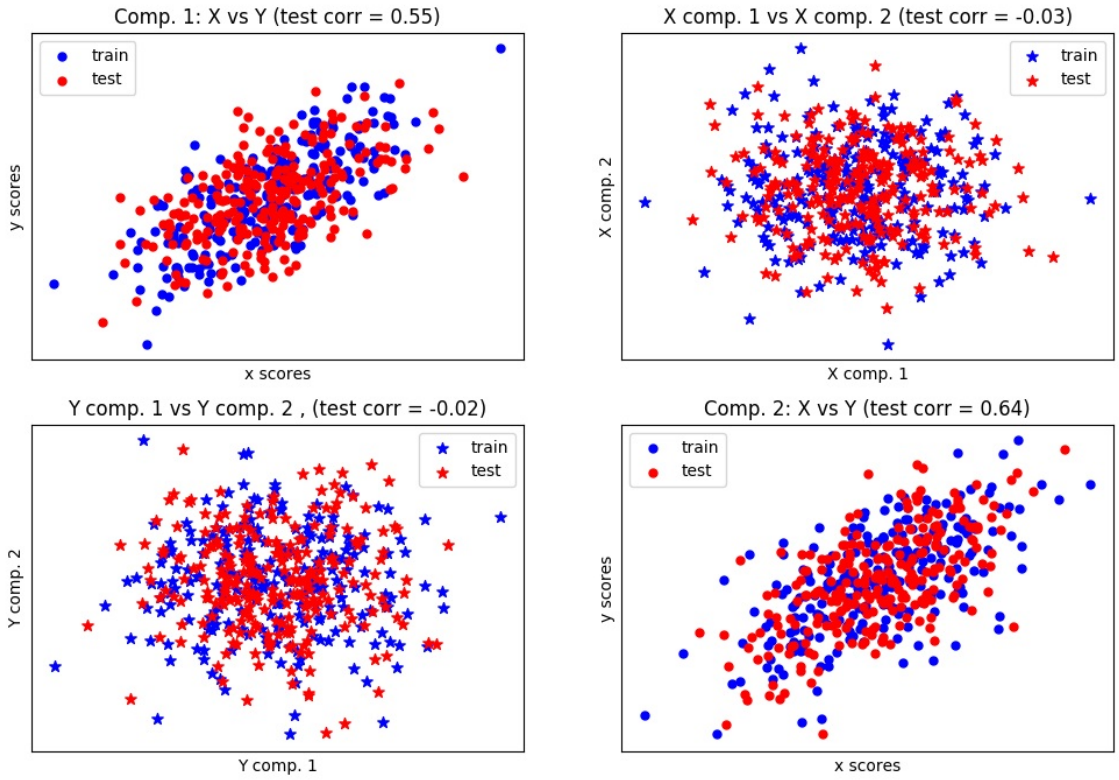
加入QQ群

1.8. 交叉分解

校验者：@peels@qinhanmin2014翻译者：@Counting stars

交叉分解模块主要包含两个算法族：偏最小二乘法（PLS）和典型相关分析（CCA）。

这些算法族具有发现两个多元数据集之间的线性关系的用途： `fit` method （拟合方法）的参数 `X` 和 `Y` 都是 2 维数组。



交叉分解算法能够找到两个矩阵（X 和 Y）的基础关系。它们是对在两个空间的协方差结构进行建模的隐变量方法。它们将尝试在X空间中找到多维方向，该方向能够解释Y空间中最大多维方差方向。 PLS回归特别适用于当预测变量矩阵具有比观测值更多的变量以及当X值存在多重共线性时。相比之下，在这些情况下，标准回归将失败。

包含在此模块中的类有： `PLSRegression` , `PLSCanonical` , `CCA` , `PLSSVD`

> 参考资料：
>
> - JA Wegelin A survey of Partial Least Squares (PLS) methods, with emphasis on the two-block case
>
> 示例：
>
> - Compare cross decomposition methods

我们一直在努力

apachecn/sklearn-doc-zh



1.9. 朴素贝叶斯

校验者：@Kyrie@Loopy@qinhanmin2014翻译者：@TWITCH

朴素贝叶斯方法是基于贝叶斯定理的一组有监督学习算法，即“简单”地假设每对特征之间相互独立。 给定一个类别 y 和一个从 x_1 到 x_n 的相关的特征向量， 贝叶斯定理阐述了以下关系：

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

使用简单(naive)的假设-每对特征之间都相互独立：

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

对于所有的 :i 都成立，这个关系式可以简化为

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

由于在给定的输入中 $P(x_1, \dots, x_n)$ 是一个常量，我们使用下面的分类规则：

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

$$\Downarrow$$

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

我们可以使用最大后验概率(Maximum A Posteriori, MAP) 来估计 $P(y)$ 和 $P(x_i \mid y)$ ； 前者是训练集中类别 y 的相对频率。

各种各样的的朴素贝叶斯分类器的差异大部分来自于处理 $P(x_i \mid y)$ 分布时的所做的假设不同。

尽管其假设过于简单，在很多实际情况下，朴素贝叶斯工作得很好，特别是文档分类和垃圾邮件过滤。这些工作都要求 一个小的训练集来估计必需参数。(至于为什么朴素贝叶斯表现得好的理论原因和它适用于哪些类型的数据，请参见下面的参考。)

相比于其他更复杂的方法，朴素贝叶斯学习器和分类器非常快。 分类条件分布的解耦意味着可以独立单独地把每个特征视为一维分布来估计。这样反过来有助于缓解维度灾难带来的问题。

另一方面，尽管朴素贝叶斯被认为是一种相当不错的分类器，但却不是好的估计器(estimator)，所以不能太过于重视从 `predict_proba` 输出的概率。

> 参考资料：
>
> - H. Zhang (2004). The optimality of Naive Bayes. Proc. FLAIRS.

1.9.1. 高斯朴素贝叶斯

`GaussianNB` 实现了运用于分类的高斯朴素贝叶斯算法。特征的可能性(即概率)假设为高斯分布：

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

参数 σ_y 和 μ_y 使用最大似然法估计。

```
>>> from sklearn import datasets
>>> iris = datasets.load_iris()
>>> from sklearn.naive_bayes import GaussianNB
>>> gnb = GaussianNB()
>>> y_pred = gnb.fit(iris.data, iris.target).predict(iris.data)
>>> print("Number of mislabeled points out of a total %d points : %d"
...       % (iris.data.shape[0],(iris.target != y_pred).sum()))
Number of mislabeled points out of a total 150 points : 6
```

1.9.2. 多项分布朴素贝叶斯

`MultinomialNB` 实现了服从多项分布数据的朴素贝叶斯算法，也是用于文本分类(这个领域中数据往往以词向量表示，尽管在实践中 `tf-idf` 向量在预测时表现良好)的两大经典朴素贝叶斯算法之一。 分布参数由每类 y 的 $\theta_y = (\theta_{y1}, \ldots, \theta_{yn})$ 向量决定， 式中 n 是特征的数量(对于文本分类，是词汇量的大小) θ_{yi} 是样本中属于类 y 中特征 i 概率 $P(x_i \mid y)$ 。

参数 θ_y 使用平滑过的最大似然估计法来估计，即相对频率计数：

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

$N_{yi} = \sum_{x \in T} x_i$ 是 训练集T中特征i在类 y 中出现的次数， $N_y = \sum_{i=1}^{n} N_{yi}$ 是类 y 中出现所有特征的计数总和。

先验平滑因子 $\alpha \geq 0$ 为在学习样本中没有出现的特征而设计，以防在将来的计算中出现0概率输出。 把 $\alpha = 1$

被称为拉普拉斯平滑(Lapalce smoothing)，而 $\alpha < 1$ 被称为Lidstone平滑方法(Lidstone smoothing)。

1.9.3. 补充朴素贝叶斯

`ComplementNB` 实现了补充朴素贝叶斯(CNB)算法。CNB是标准多项式朴素贝叶斯(MNB)算法的一种改进，特别适用于不平衡数据集。具体来说，CNB使用来自每个类的补数的统计数据来计算模型的权重。CNB的发明者的研究表明，CNB的参数估计比MNB的参数估计更稳定。此外，CNB在文本分类任务上通常比MNB表现得更好(通常有相当大的优势)。计算权重的步骤如下：

$$\hat{\theta}_{ci} = \frac{\alpha_i + \sum_{j:y_j \neq c} d_{ij}}{\alpha + \sum_{j:y_j \neq c} \sum_k d_{kj}}$$

$$w_{ci} = \log \hat{\theta}_{ci}$$

$$w_{ci} = \frac{w_{ci}}{\sum_j |w_{cj}|}$$

其中对不在类c中的所有记录j求和，d_{ij}可以是文档 j 中词语i的计数或tf-idf值，α_i是就像MNB中一样的平滑超参数，同时 $\alpha = \sum_i \alpha_i$ 。第二个归一化解决了长记录主导MNB参数估计的问题。分类规则为：

$$\hat{c} = \arg\min_c \sum_i t_i w_{ci}$$

即将记录分配给补充匹配度最低的类。

1.9.4. 伯努利朴素贝叶斯

`BernoulliNB` 实现了用于多重伯努利分布数据的朴素贝叶斯训练和分类算法，即有多个特征，但每个特征 都假设是一个二元 (Bernoulli, boolean) 变量。 因此，这类算法要求样本以二元值特征向量表示；如果样本含有其他类型的数据， 一个 `BernoulliNB` 实例会将其二值化(取决于 `binarize` 参数)。

伯努利朴素贝叶斯的决策规则基于：

$$P(x_i \mid y) = P(i \mid y)x_i + (1 - P(i \mid y))(1 - x_i)$$

与多项分布朴素贝叶斯的规则不同 伯努利朴素贝叶斯明确地惩罚类 y 中没有出现作为预测因子的特征 i ，而多项分布分布朴素贝叶斯只是简单地忽略没出现的特征。

在文本分类的例子中，统计词语是否出现的向量(word occurrence vectors)(而非统计词语出现次数的向量(word count vectors))可以用于训练和使用这个分类器。 `BernoulliNB` 可能在一些数据集上表现得更好，特别是那些更短的文档。 如果时间允许，建议对两个模型都进行评估。

参考资料：

- C.D. Manning, P. Raghavan and H. Schütze (2008). Introduction to Information Retrieval. Cambridge University Press, pp. 234-265.
- A. McCallum and K. Nigam (1998). A comparison of event models for Naive Bayes text classification. Proc. AAAI/ICML-98 Workshop on Learning for Text Categorization, pp. 41-48.
- V. Metsis, I. Androutsopoulos and G. Paliouras (2006). Spam filtering with Naive Bayes – Which Naive Bayes? 3rd Conf. on Email and Anti-Spam (CEAS).

1.9.5. 基于外存的朴素贝叶斯模型拟合

朴素贝叶斯模型可以解决整个训练集不能导入内存的大规模分类问题。 为了解决这个问题， `MultinomialNB` ， `BernoulliNB` ，和 `GaussianNB` 实现了 `partial_fit` 方法，可以动态的增加数据，使用方法与其他分类器的一样，使用示例见 `Out-of-core classification of text documents` 。所有的朴素贝叶斯分类器都支持样本权重。

与 `fit` 方法不同，首次调用 `partial_fit` 方法需要传递一个所有期望的类标签的列表。

对于 scikit-learn 中可用方案的概览，另见 `out-of-core learning` 文档。

注意：所有朴素贝叶斯模型调用 `partial_fit` 都会引入一些计算开销。推荐让数据块越大越好，其大小与 RAM 中可用内存大小相同。

我们一直在努力

apachecn/sklearn-doc-zh



1.10. 决策树

校验者：@文谊@皮卡乒的皮卡乓@Loopy翻译者：@I Remember

Decision Trees (DTs) 是一种用来 `classification` 和 `regression` 的无参监督学习方法。其目的是创建一种模型从数据特征中学习简单的决策规则来预测一个目标变量的值。

例如，在下面的图片中，决策树通过if-then-else的决策规则来学习数据从而估测数一个正弦图像。决策树越深入，决策规则就越复杂并且对数据的拟合越好。



决策树的优势：

- 便于理解和解释。树的结构可以可视化出来。
- 训练需要的数据少。其他机器学习模型通常需要数据规范化，比如构建虚拟变量和移除缺失值，不过请注意，这种模型不支持缺失值。
- 由于训练决策树的数据点的数量导致了决策树的使用开销呈指数分布(训练树模型的时间复杂度是参与训练数据点的对数值)。
- 能够处理数值型数据和分类数据。其他的技术通常只能用来专门分析某一种变量类型的数据集。详情请参阅算法。
- 能够处理多路输出的问题。
- 使用白盒模型。如果某种给定的情况在该模型中是可以观察的，那么就可以轻易的通过布尔逻辑来解释这种情

况。相比之下，在黑盒模型中的结果就是很难说明清 楚地。
- 可以通过数值统计测试来验证该模型。这对事解释验证该模型的可靠性成为可能。
- 即使该模型假设的结果与真实模型所提供的数据有些违反，其表现依旧良好。

决策树的缺点包括：

- 决策树模型容易产生一个过于复杂的模型，这样的模型对数据的泛化性能会很差。这就是所谓的过拟合。一些策略像剪枝、设置叶节点所需的最小样本数或设置数的最大深度是避免出现 该问题最为有效地方法。
- 决策树可能是不稳定的，因为数据中的微小变化可能会导致完全不同的树生成。这个问题可以通过决策树的集成来得到缓解
- 在多方面性能最优和简单化概念的要求下，学习一棵最优决策树通常是一个NP难问题。因此，实际的决策树学习算法是基于启发式算法，例如在每个节点进 行局部最优决策的贪心算法。这样的算法不能保证返回全局最优决策树。这个问题可以通过集成学习来训练多棵决策树来缓解，这多棵决策树一般通过对特征和样本有放回的随机采样来生成。
- 有些概念很难被决策树学习到，因为决策树很难清楚的表述这些概念。例如XOR，奇偶或者复用器的问题。
- 如果某些类在问题中占主导地位会使得创建的决策树有偏差。因此，我们建议在拟合前先对数据集进行平衡。

1.10.1. 分类

`DecisionTreeClassifier` 是能够在数据集上执行多分类的类，与其他分类器一样， `DecisionTreeClassifier` 采用输入两个数组：数组X，用 `[n_samples, n_features]` 的方式来存放训练样本。整数值数组Y，用 `[n_samples]` 来保存训练样本的类标签：

```
>>> from sklearn import tree
>>> X = [[0, 0], [1, 1]]
>>> Y = [0, 1]
>>> clf = tree.DecisionTreeClassifier()
>>> clf = clf.fit(X, Y)
```

执行通过之后，可以使用该模型来预测样本类别：

```
>>> clf.predict([[2., 2.]])
array([1])
```

另外，也可以预测每个类的概率，这个概率是叶中相同类的训练样本的分数：

```
>>> clf.predict_proba([[2., 2.]])
array([[ 0.,  1.]])
```

`DecisionTreeClassifier` 既能用于二分类（其中标签为[-1,1]）也能用于多分类（其中标签为[0,…,k-1]）。使用Lris数据集，我们可以构造一个决策树，如下所示：

```
>>> from sklearn.datasets import load_iris
>>> from sklearn import tree
>>> iris = load_iris()
>>> clf = tree.DecisionTreeClassifier()
>>> clf = clf.fit(iris.data, iris.target)
```

经过训练，我们可以使用 `export_graphviz` 导出器以 Graphviz 格式导出决策树. 如果你是用 conda 来管理包，那么安装 graphviz 二进制文件和 python 包可以用以下指令安装

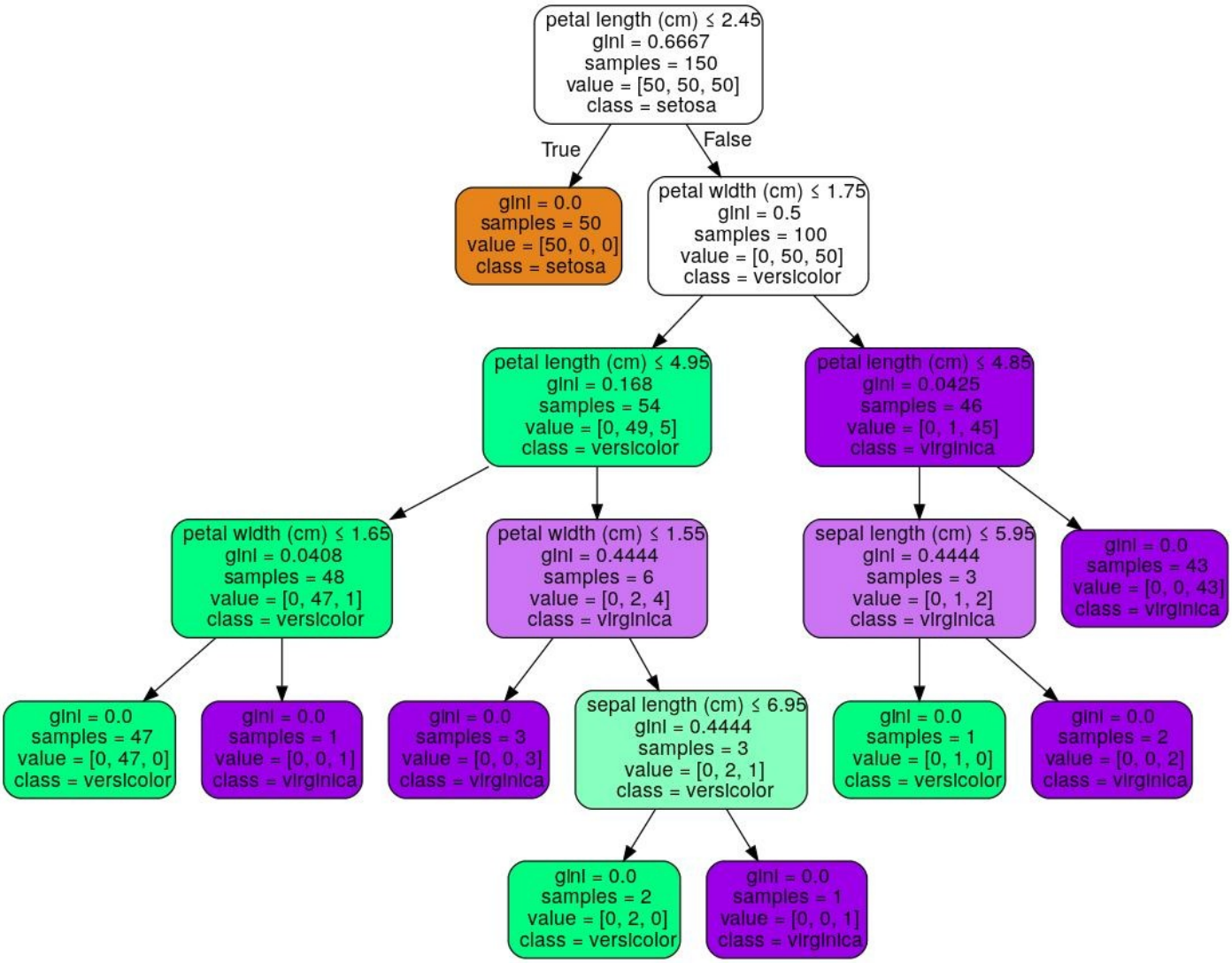
```
conda install python-graphviz
```

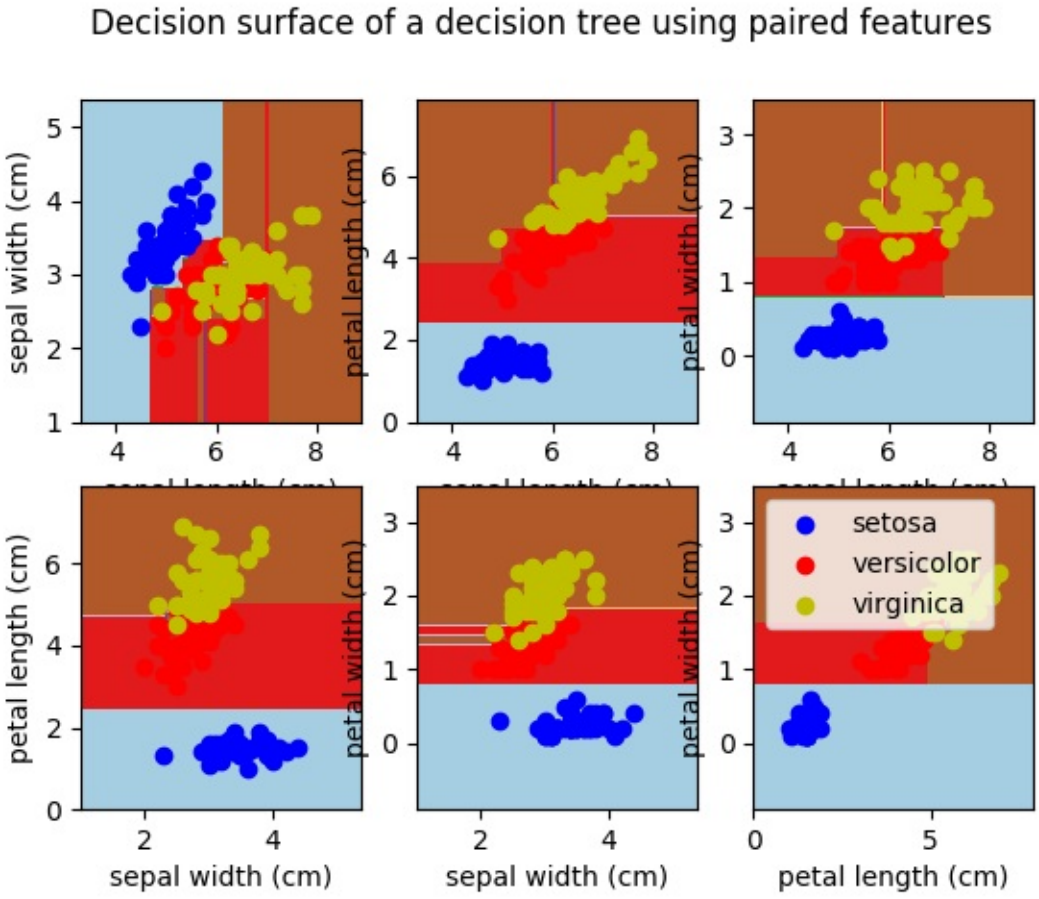
或者，可以从 graphviz 项目主页下载 graphviz 的二进制文件，并从 pypi 安装：Python 包装器，并安装 `pip install graphviz` .以下是在整个 iris 数据集上训练的上述树的 graphviz 导出示例；其结果被保存在 *iris.pdf* 中：

```
>>> import graphviz
>>> dot_data = tree.export_graphviz(clf, out_file=None)
>>> graph = graphviz.Source(dot_data)
>>> graph.render("iris")
```

`export_graphviz` 还支持各种美化，包括通过他们的类着色节点（或回归值），如果需要，还能使用显式变量和类名。Jupyter notebook也可以自动内联式渲染这些绘制节点：

```
>>> dot_data = tree.export_graphviz(clf, out_file=None,
...                                 feature_names=iris.feature_names,
...                                 class_names=iris.target_names,
...                                 filled=True, rounded=True,
...                                 special_characters=True)
>>> graph = graphviz.Source(dot_data)
>>> graph
```

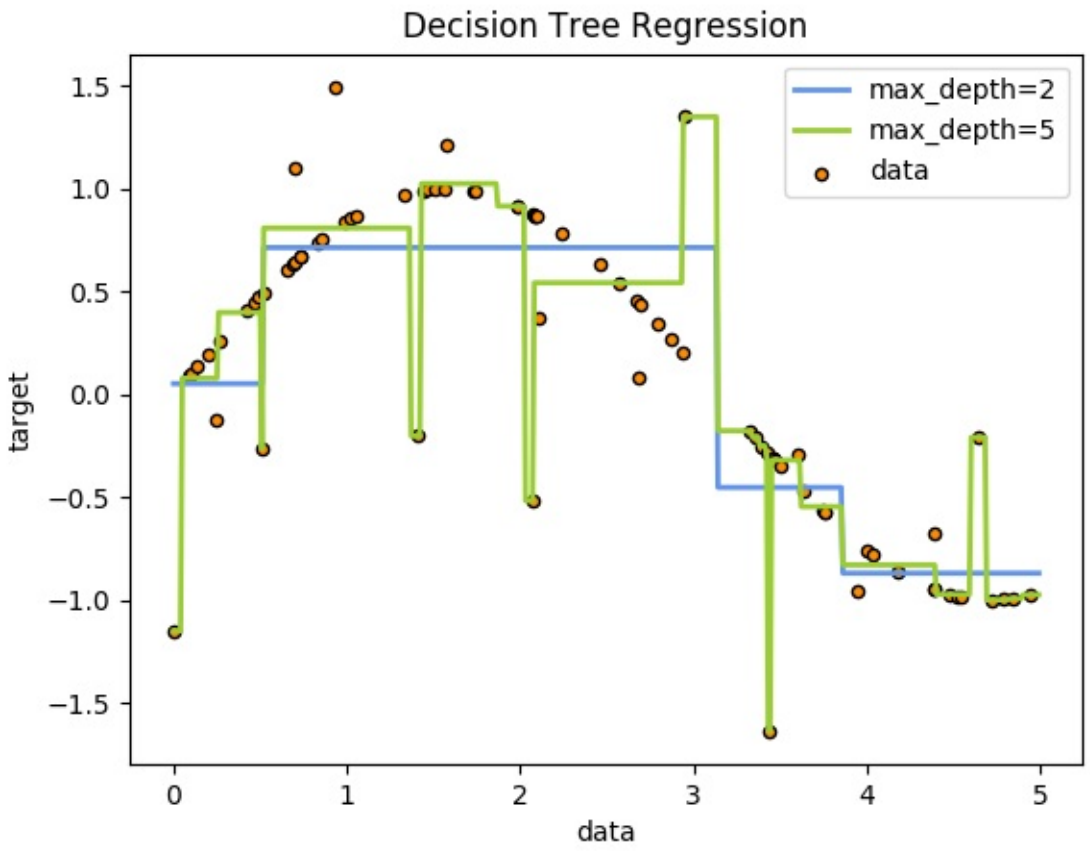

petal length (cm) ≤ 2.45
gini = 0.6667
samples = 150
value = [50, 50, 50]
class = setosa
True
False
gini = 0.0
samples = 50
value = [50, 0, 0]
class = setosa
petal width (cm) ≤ 1.75
gini = 0.5
samples = 100
value = [0, 50, 50]
class = versicolor
petal length (cm) ≤ 4.95
gini = 0.168
samples = 54
value = [0, 49, 5]
class = versicolor
petal length (cm) ≤ 4.85
gini = 0.0425
samples = 46
value = [0, 1, 45]
class = virginica
petal width (cm) ≤ 1.65
gini = 0.0408
samples = 48
value = [0, 47, 1]
class = versicolor
petal width (cm) ≤ 1.55
gini = 0.4444
samples = 6
value = [0, 2, 4]
class = virginica
sepal length (cm) ≤ 5.95
gini = 0.4444
samples = 3
value = [0, 1, 2]
class = virginica
gini = 0.0
samples = 43
value = [0, 0, 43]
class = virginica
gini = 0.0
samples = 47
value = [0, 47, 0]
class = versicolor
gini = 0.0
samples = 1
value = [0, 0, 1]
class = virginica
gini = 0.0
samples = 3
value = [0, 0, 3]
class = virginica
sepal length (cm) ≤ 6.95
gini = 0.4444
samples = 3
value = [0, 2, 1]
class = versicolor
gini = 0.0
samples = 1
value = [0, 1, 0]
class = versicolor
gini = 0.0
samples = 2
value = [0, 0, 2]
class = virginica
gini = 0.0
samples = 2
value = [0, 2, 0]
class = versicolor
gini = 0.0
samples = 1
value = [0, 0, 1]
class = virginica



示例：

- Plot the decision surface of a decision tree on the iris dataset

1.10.2. 回归



决策树通过使用 `DecisionTreeRegressor` 类也可以用来解决回归问题。如在分类设置中，拟合方法将数组X和数组y作为参数，只有在这种情况下，y数组预期才是浮点值：

```
>>> from sklearn import tree
>>> X = [[0, 0], [2, 2]]
>>> y = [0.5, 2.5]
>>> clf = tree.DecisionTreeRegressor()
>>> clf = clf.fit(X, y)
>>> clf.predict([[1, 1]])
array([ 0.5])
```

示例：

- Decision Tree Regression

1.10.3. 多值输出问题

一个多值输出问题是一个类似当 Y 是大小为 `[n_samples, n_outputs]` 的2d数组时，有多个输出值需要预测的监督学习问题。

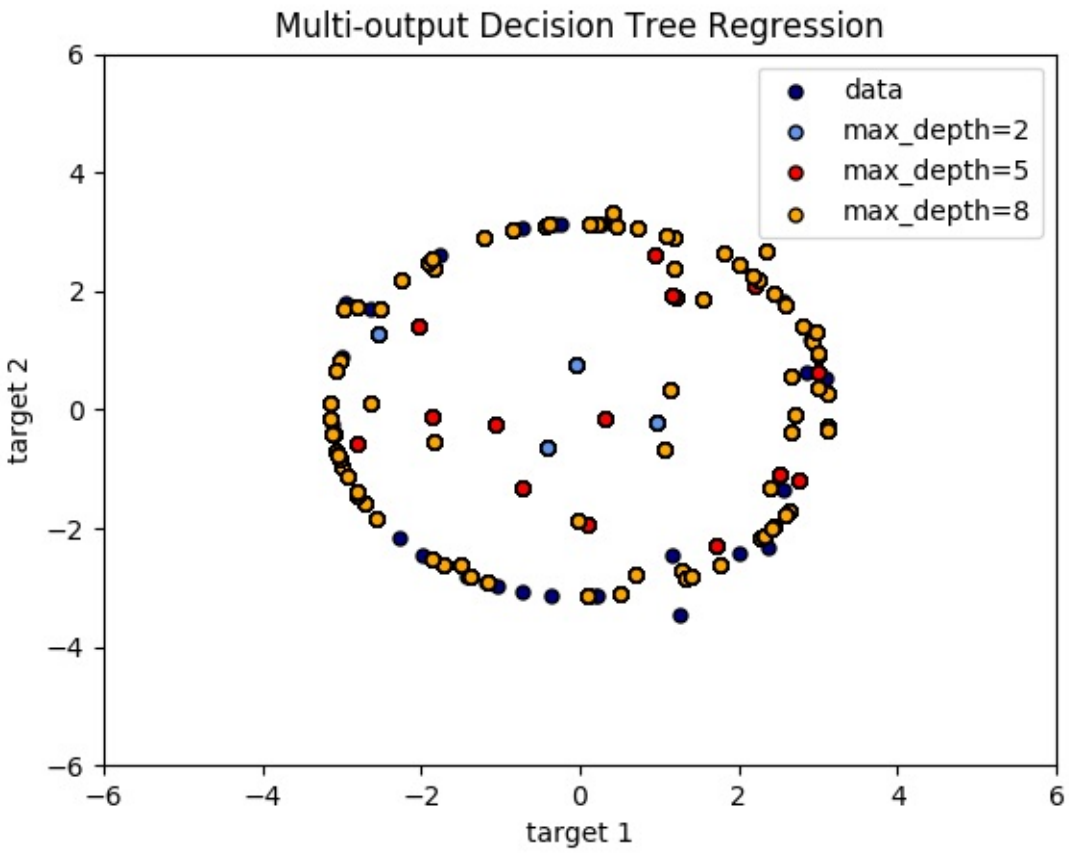
当输出值之间没有关联时，一个很简单的处理该类型的方法是建立一个n独立模型，即每个模型对应一个输出，然后使用这些模型来独立地预测n个输出中的每一个。然而，由于可能与相同输入相关的输出值本身是相关的，所以通常更好的方法是构建能够同时预测所有n个输出的单个模型。首先，因为仅仅是建立了一个模型所以训练时间会更短。第二，最终模型的泛化性能也会有所提升。对于决策树，这一策略可以很容易地用于多输出问题。 这需要以下更改：

- 在叶中存储n个输出值，而不是一个；
- 通过计算所有n个输出的平均减少量来作为分裂标准.

该模块通过在 `DecisionTreeClassifier` 和 `DecisionTreeRegressor` 中实现该策略来支持多输出问题。如果决策树与大小为 `[n_samples, n_outputs]` 的输出数组Y向匹配，则得到的估计器：

- `predict` 是输出n_output的值
- 在 `predict_proba` 上输出 n_output 数组列表

用多输出决策树进行回归分析 Multi-output Decision Tree Regression 。 在该示例中，输入X是单个实数值，并且输出Y是X的正弦和余弦。



使用多输出树进行分类，在 Face completion with a multi-output estimators 中进行了演示。 在该示例中，输入X是面的上半部分的像素，并且输出Y是这些面的下半部分的像素。



示例：

- Multi-output Decision Tree Regression
- Face completion with a multi-output estimators

参考资料：

- M. Dumont et al, Fast multi-class image annotation with random subwindows and multiple output randomized trees, International Conference on Computer Vision Theory and Applications 2009

1.10.4. 复杂度分析

总体来说，用来构建平衡二叉树的运行时间为 $O(n_{samples}n_{features}\log(n_{samples}))$ 查询时间为 $O(\log(n_{samples}))$ 。尽管树的构造算法尝试生成平衡树，但它们并不总能保持平衡。假设子树能大概保持平衡，每个节点的成本包括通过 $O(n_{features})$ 时间复杂度来搜索找到提供熵减小最大的特征。每个节点的花费为 $O(n_{features}n_{samples}\log(n_{samples}))$ ，从而使得整个决策树的构造成本为 $O(n_{features}n^2_{samples}\log(n_{samples}))$ 。

Scikit-learn提供了更多有效的方法来创建决策树。初始实现（如上所述）将重新计算沿着给定特征的每个新分割点的类标签直方图（用于分类）或平均值（用于回归）。与分类所有的样本特征，然后再次训练时运行标签计数，可将每个节点的复杂度降低为 $O(n_{features}\log(n_{samples}))$ ，则总的成本花费为 $O(n_{features}n_{samples}\log(n_{samples}))$ 。这是一种对所有基于树的算法的改进选项。默认情况下，对于梯度提升模型该算法是打开的，一般来说它会让训练速度更快。但对于所有其他算法默认是关闭的，当训练深度很深的树时往往会减慢训练速度。

1.10.5. 实际使用技巧

- 对于拥有大量特征的数据决策树会出现过拟合的现象。获得一个合适的样本比例和特征数量十分重要，因为在高维空间中只有少量的样本的树是十分容易过拟合的。
- 考虑事先进行降维(PCA , ICA ，使您的树更好地找到具有分辨性的特征。
- 通过 `export` 功能可以可视化您的决策树。使用 `max_depth=3` 作为初始树深度，让决策树知道如何适应您的数据，然后再增加树的深度。
- 请记住，填充树的样本数量会增加树的每个附加级别。使用 `max_depth` 来控制输的大小防止过拟合。
- 通过使用 `min_samples_split` 和 `min_samples_leaf` 来控制叶节点上的样本数量。当这个值很小时意味着生成的决策树将会过拟合，然而当这个值很大时将会不利于决策树的对样本的学习。所以尝试 `min_samples_leaf=5` 作为初始值。如果样本的变化量很大，可以使用浮点数作为这两个参数中的百分比。两者之间的主要区别在于 `min_samples_leaf` 保证叶结点中最少的采样数，而 `min_samples_split` 可以创建任意小的叶子，尽管在文献中 `min_samples_split` 更常见。
- 在训练之前平衡您的数据集，以防止决策树偏向于主导类.可以通过从每个类中抽取相等数量的样本来进行类平衡，或者优选地通过将每个类的样本权重（ `sample_weight` ）的和归一化为相同的值。还要注意的是，基于权重的预修剪标准（ `min_weight_fraction_leaf` ）对于显性类别的偏倚偏小，而不是不了解样本权重的标准，如 `min_samples_leaf` 。
- 如果样本被加权，则使用基于权重的预修剪标准 `min_weight_fraction_leaf` 来优化树结构将更容易，这确保叶节点包含样本权重的总和的至少一部分。
- 所有的决策树内部使用 `np.float32` 数组 ，如果训练数据不是这种格式，将会复制数据集。
- 如果输入的矩阵X为稀疏矩阵，建议您在调用fit之前将矩阵X转换为稀疏的 `csc_matrix` ,在调用predict之前将 `csr_matrix` 稀疏。当特征在大多数样本中具有零值时，与密集矩阵相比，稀疏矩阵输入的训练时间可以快几个数量级。

1.10.6. 决策树算法: ID3, C4.5, C5.0 和 CART

所有种类的决策树算法有哪些以及它们之间的区别？scikit-learn 中实现何种算法呢？

ID3（Iterative Dichotomiser 3）由 Ross Quinlan 在1986年提出。该算法创建一个多路树，找到每个节点（即以贪心的方式）分类特征，这将产生分类目标的最大信息增益。决策树发展到其最大尺寸，然后通常利用剪枝来提高树对未知数据的泛华能力。

C4.5 是 ID3 的后继者，并且通过动态定义将连续属性值分割成一组离散间隔的离散属性（基于数字变量），消除了特征必须被明确分类的限制。C4.5 将训练的树（即，ID3算法的输出）转换成 if-then 规则的集合。然后评估每个规则的这些准确性，以确定应用它们的顺序。如果规则的准确性没有改变，则需要决策树的树枝来解决。

C5.0 是 Quinlan 根据专有许可证发布的最新版本。它使用更少的内存，并建立比 C4.5 更小的规则集，同时更准确。

CART（Classification and Regression Trees （分类和回归树））与 C4.5 非常相似，但它不同之处在于它支持数值目标变量（回归），并且不计算规则集。CART 使用在每个节点产生最大信息增益的特征和阈值来构造二叉树。

scikit-learn 使用 CART 算法的优化版本。

1.10.7. 数学表达

给定训练向量 $x_i \in R^n$，i=1,…, l 和标签向量 $y \in R^l$。决策树递归地分割空间，例如将有相同标签的样本归为一组。

将 m 节点上的数据用 Q 来表示。每一个候选组 $\theta = (j, t_m)$ 包含一个特征 j 和阈值 t_m 将，数据分成 $Q_{left}(\theta)$ 和 $Q_{right}(\theta)$ 子集。

$$Q_{left}(\theta) = (x, y)|x_j <= t_m$$
$$Q_{right}(\theta) = Q \setminus Q_{left}(\theta)$$

使用不纯度函数 $H()$ 计算 m 处的不纯度，其选择取决于正在解决的任务（分类或回归）

$$G(Q, \theta) = \frac{n_{left}}{N_m} H(Q_{left}(\theta)) + \frac{n_{right}}{N_m} H(Q_{right}(\theta))$$

选择使不纯度最小化的参数

$$\theta^* = \operatorname{argmin}_\theta G(Q, \theta)$$

在 $Q_{left}(\theta^*)$ 和 $Q_{right}(\theta^*)$ 上递归运算，直到达到最大允许深度，$N_m < \min_{samples}$ 或 $N_m = 1$。

1.10.7.1. 分类标准

对于节点 m ，表示具有 N_m 个观测值的区域 R_m ，如果分类结果采用值是 0,1,…,K-1 的值，让

$$p_{mk} = 1/N_m \sum_{x_i \in R_m} I(y_i = k)$$

是节点 m 中k类观测的比例通常用来处理杂质的方法是Gini

$$H(X_m) = \sum_k p_{mk}(1 - p_{mk})$$

Cross-Entropy （交叉熵）

$$H(X_m) = -\sum_k p_{mk} \log(p_{mk})$$

和 Misclassification （错误分类）

$$H(X_m) = 1 - \max(p_{mk})$$

在 X_m 训练 m 节点上的数据时。

1.10.7.2. 回归标准

如果目标是连续性的值，那么对于节点 m ,表示具有 N_m 个观测值的区域 R_m ，对于以后的分裂节点的位置的决定常用的最小化标准是均方差和平均绝对误差，前者使用终端节点处的平均值来最小化L2误差，后者使用终端节点处的中值来最小化 L1 误差。

Mean Squared Error （均方误差）：

$$c_m = \frac{1}{N_m} \sum_{i \in N_m} y_i$$
$$H(X_m) = \frac{1}{N_m} \sum_{i \in N_m} (y_i - c_m)^2$$

Mean Absolute Error（平均绝对误差）：

$$\bar{y_m} = \frac{1}{N_m} \sum_{i \in N_m} y_i$$
$$H(X_m) = \frac{1}{N_m} \sum_{i \in N_m} |y_i - \bar{y_m}|$$

在 X_m 训练 m 节点上的数据时。

> 示例：
>
> - https://en.wikipedia.org/wiki/Decision_tree_learning
> - https://en.wikipedia.org/wiki/Predictive_analytics
> - L. Breiman, J. Friedman, R. Olshen, and C. Stone. Classification and Regression Trees. Wadsworth, Belmont, CA, 1984.
> - J.R. Quinlan. C4. 5: programs for machine learning. Morgan Kaufmann, 1993.
> - T. Hastie, R. Tibshirani and J. Friedman. Elements of Statistical Learning, Springer, 2009.

我们一直在努力

apachecn/sklearn-doc-zh



1.11. 集成方法

校验者: @Dream on dreamer.@zehuichen123@JanzenLiu@小瑶@\S^R^Y/@Loopy@qinhanmin2014翻译者: @StupidStalker@文谊@t9UhoI

> 注意:在本文中 bagging 和 boosting 为了更好的保留原文意图，不进行翻译 estimator:估计器 base estimator:基估计器

集成方法 的目标是把多个使用给定学习算法构建的基估计器的预测结果结合起来，从而获得比单个估计器更好的泛化能力/鲁棒性。

集成方法通常分为两种:

- 平均方法，该方法的原理是构建多个独立的估计器，然后取它们的预测结果的平均。一般来说组合之后的估计器是会比单个估计器要好的，因为它的方差减小了。

示例:Bagging 方法 ， 随机森林 ， …

- 相比之下，在 **boosting** 方法 中，基估计器是依次构建的，并且每一个基估计器都尝试去减少组合估计器的偏差。这种方法主要目的是为了结合多个弱模型，使集成的模型更加强大。

示例:AdaBoost ， 梯度提升树 ， …

1.11.1. Bagging meta-estimator（Bagging 元估计器）

在集成算法中，bagging 方法会在原始训练集的随机子集上构建一类黑盒估计器的多个实例，然后把这些估计器的预测结果结合起来形成最终的预测结果。 该方法通过在构建模型的过程中引入随机性，来减少基估计器的方差(例如，决策树)。 在多数情况下，bagging 方法提供了一种非常简单的方式来对单一模型进行改进，而无需修改背后的算法。 因为 bagging 方法可以减小过拟合，所以通常在强分类器和复杂模型上使用时表现的很好（例如，完全生长的决策树，fully developed decision trees），相比之下 boosting 方法则在弱模型上表现更好（例如，浅层决策树，shallow decision trees）。

bagging 方法有很多种，其主要区别在于随机抽取训练子集的方法不同:

- 如果抽取的数据集的随机子集是样例的随机子集，我们叫做粘贴 (Pasting) [B1999] 。
- 如果样例抽取是有放回的，我们称为 Bagging [B1996] 。
- 如果抽取的数据集的随机子集是特征的随机子集，我们叫做随机子空间 (Random Subspaces) [H1998] 。
- 最后，如果基估计器构建在对于样本和特征抽取的子集之上时，我们叫做随机补丁 (Random Patches) [LG2012]。

在 scikit-learn 中，bagging 方法使用统一的 `BaggingClassifier` 元估计器（或者 `BaggingRegressor` ），基估计器和随机子集抽取策略由用户指定。 `max_samples` 和 `max_features` 控制着子集的大小（对于样例和特征），`bootstrap` 和 `bootstrap_features` 控制着样例和特征的抽取是有放回还是无放回的。 当使用样本子集时，通过设置 `oob_score=True` ，可以使用袋外(out-of-bag)样本来评估泛化精度。下面的代码片段说明了如何构造一个 `KNeighborsClassifier` 估计器的 bagging 集成实例，每一个基估计器都建立在 50% 的样本随机子集和 50% 的特征随机子集上。

```
>>> from sklearn.ensemble import BaggingClassifier
>>> from sklearn.neighbors import KNeighborsClassifier
>>> bagging = BaggingClassifier(KNeighborsClassifier(),
...                             max_samples=0.5, max_features=0.5)
```

示例：

- Single estimator versus bagging: bias-variance decomposition

参考资料

- [B1999] L. Breiman, “Pasting small votes for classification in large databases and on-line”, Machine Learning, 36(1), 85-103, 1999.
- [B1996] L. Breiman, “Bagging predictors”, Machine Learning, 24(2), 123-140, 1996.
- [H1998] T. Ho, “The random subspace method for constructing decision forests”, Pattern Analysis and Machine Intelligence, 20(8), 832-844, 1998.
- [LG2012] G. Louppe and P. Geurts, “Ensembles on Random Patches”, Machine Learning and Knowledge Discovery in Databases, 346-361, 2012.

1.11.2. 由随机树组成的森林

`sklearn.ensemble` 模块包含两个基于 随机决策树 的平均算法： RandomForest 算法和 Extra-Trees 算法。这两种算法都是专门为树而设计的扰动和组合技术（perturb-and-combine techniques） [B1998] 。 这种技术通过在分类器构造过程中引入随机性来创建一组不同的分类器。集成分类器的预测结果就是单个分类器预测结果的平均值。

与其他分类器一样，森林分类器必须拟合（fit）两个数组： 保存训练样本的数组（或稀疏或稠密的）X，大小为 `[n_samples, n_features]` ，和 保存训练样本目标值（类标签）的数组 Y，大小为 `[n_samples]` ：

```
>>> from sklearn.ensemble import RandomForestClassifier
>>> X = [[0, 0], [1, 1]]
>>> Y = [0, 1]
>>> clf = RandomForestClassifier(n_estimators=10)
>>> clf = clf.fit(X, Y)
```

同 决策树 一样，随机森林算法（forests of trees）也能用来解决 多输出问题（如果 Y 的大小是 `[n_samples, n_outputs]` ）。

1.11.2.1. 随机森林

在随机森林中（参见 `RandomForestClassifier` 和 `RandomForestRegressor` 类）， 集成模型中的每棵树构建时的样本都是由训练集经过有放回抽样得来的（例如，自助采样法-bootstrap sample，这里采用西瓜书中的译法）。

另外，在构建树的过程中进行结点分割时，选择的分割点是所有特征的最佳分割点，或特征的大小为 `max_features` 的随机子集的最佳分割点。

这两种随机性的目的是降低估计器的方差。的确，单棵决策树通常具有高方差，容易过拟合。随机森林构建过程的随机性能够产生具有不同预测错误的决策树。通过取这些决策树的平均，能够消除部分错误。随机森林虽然能够通过组合不同的树降低方差，但是有时会略微增加偏差。在实际问题中，方差的降低通常更加显著，所以随机森林能够取得更好地效果。

与原始文献[B2001]不同的是，scikit-learn 的实现是取每个分类器预测概率的平均，而不是让每个分类器对类别进行投票。

1.11.2.2. 极限随机树

在极限随机树中（参见 `ExtraTreesClassifier` 和 `ExtraTreesRegressor` 类），计算分割点方法中的随机性进一步增强。 与随机森林相同，使用的特征是候选特征的随机子集；但是不同于随机森林寻找最具有区分度的阈值，这里的阈值是针对每个候选特征随机生成的，并且选择这些随机生成的阈值中的最佳者作为分割规则。 这种做法通常能够减少一点模型的方差，代价则是略微地增大偏差：

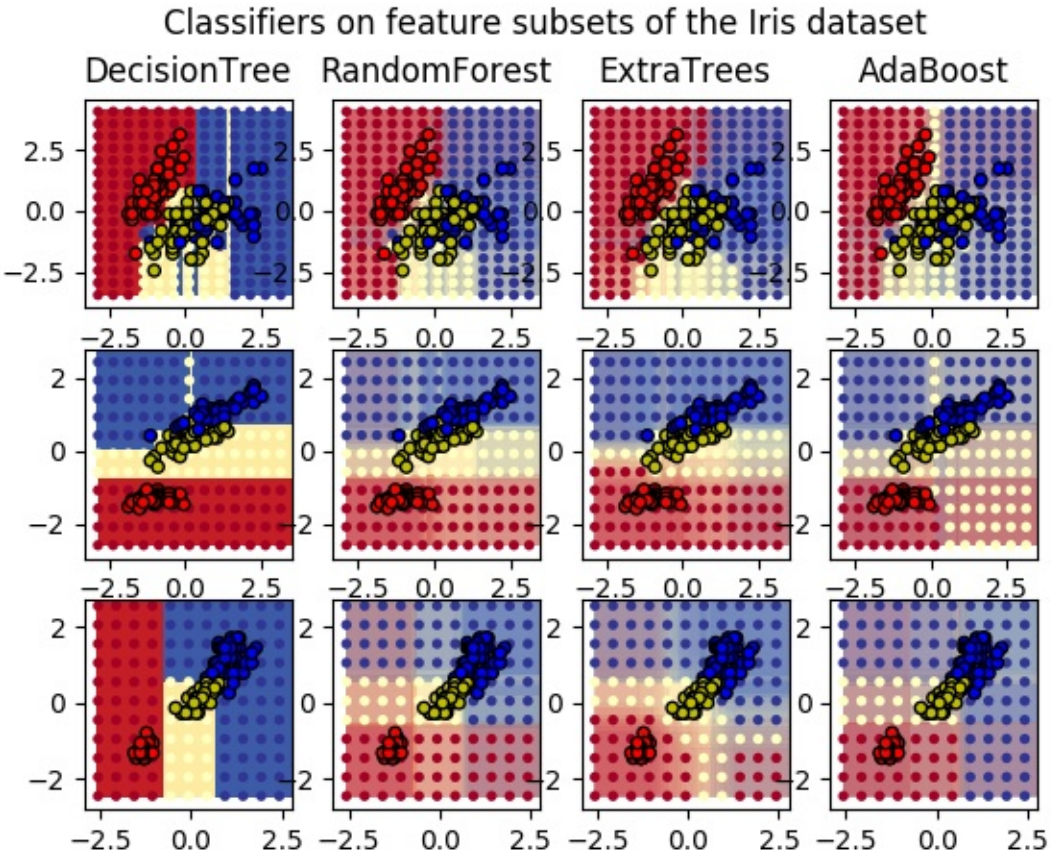
```
>>> from sklearn.model_selection import cross_val_score
>>> from sklearn.datasets import make_blobs
>>> from sklearn.ensemble import RandomForestClassifier
>>> from sklearn.ensemble import ExtraTreesClassifier
>>> from sklearn.tree import DecisionTreeClassifier

>>> X, y = make_blobs(n_samples=10000, n_features=10, centers=100,
...     random_state=0)

>>> clf = DecisionTreeClassifier(max_depth=None, min_samples_split=2,
...     random_state=0)
>>> scores = cross_val_score(clf, X, y, cv=5)
>>> scores.mean()
0.98...

>>> clf = RandomForestClassifier(n_estimators=10, max_depth=None,
...     min_samples_split=2, random_state=0)
>>> scores = cross_val_score(clf, X, y, cv=5)
>>> scores.mean()
0.999...

>>> clf = ExtraTreesClassifier(n_estimators=10, max_depth=None,
...     min_samples_split=2, random_state=0)
>>> scores = cross_val_score(clf, X, y, cv=5)
>>> scores.mean() > 0.999
True
```



1.11.2.3. 参数

使用这些方法时要调整的参数主要是 `n_estimators` 和 `max_features`。 前者（n_estimators）是森林里树的数量，通常数量越大，效果越好，但是计算时间也会随之增加。 此外要注意，当树的数量超过一个临界值之后，算法的效果并不会很显著地变好。 后者（max_features）是分割节点时考虑的特征的随机子集的大小。 这个值越低，方差减小得越多，但是偏差的增大也越多。 根据经验，回归问题中使用 `max_features = None`（总是考虑所有的特征）， 分类问题使用 `max_features = "sqrt"`（随机考虑 `sqrt(n_features)` 特征，其中 `n_features` 是特征的个数）是比较好的默认值。 `max_depth = None` 和 `min_samples_split = 2` 结合通常会有不错的效果（即生成完全的树）。 请记住，这些（默认）值通常不是最佳的，同时还可能消耗大量的内存，最佳参数值应由交叉验证获得。 另外，请注意，在随机森林中，默认使用自助采样法（`bootstrap = True`）， 然而 extra-trees 的默认策略是使用整个数据集（`bootstrap = False`）。 当使用自助采样法方法抽样时，泛化精度是可以通过剩余的或者袋外的样本来估算的，设置 `oob_score = True` 即可实现。

> 注意：
>
> 默认参数下模型复杂度是：`O(MNlog(N))`，其中 `M` 是树的数目，`N` 是样本数。 可以通过设置以下参数来降低模型复杂度：`min_samples_split`，`max_leaf_nodes`，`max_depth` 和 `min_samples_leaf`。

1.11.2.4. 并行化

最后，这个模块还支持树的并行构建和预测结果的并行计算，这可以通过 `n_jobs` 参数实现。 如果设置 `n_jobs = k`，则计算被划分为 `k` 个作业，并运行在机器的 `k` 个核上。 如果设置 `n_jobs = -1`，则使用机器的所有核。 注意由于进程间通信具有一定的开销，这里的提速并不是线性的（即，使用 `k` 个作业不会快 `k` 倍）。 当然，在建立大量的树，或者构建单个树需要相当长的时间（例如，在大数据集上）时，（通过并行化）仍然

可以实现显著的加速。

示例：

- Plot the decision surfaces of ensembles of trees on the iris dataset
- Pixel importances with a parallel forest of trees
- Face completion with a multi-output estimators

参考资料

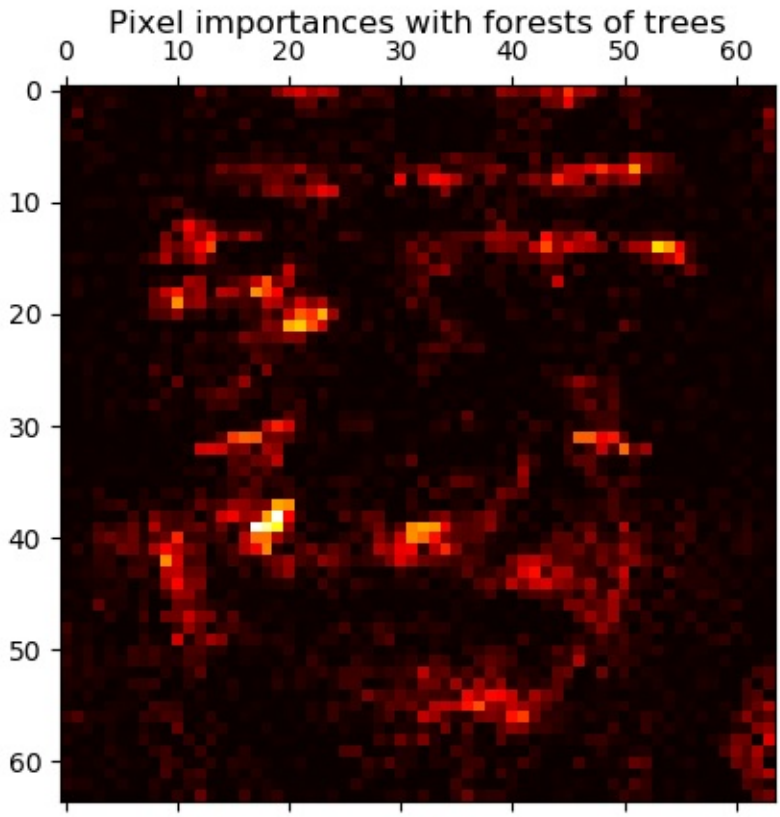
- [B2001] Breiman, “Random Forests”, Machine Learning, 45(1), 5-32, 2001.
- [B1998] Breiman, “Arcing Classifiers”, Annals of Statistics 1998.
- P. Geurts, D. Ernst., and L. Wehenkel, “Extremely randomized trees”, Machine Learning, 63(1), 3-42, 2006.

1.11.2.5. 特征重要性评估

特征对目标变量预测的相对重要性可以通过（树中的决策节点的）特征使用的相对顺序（即深度）来进行评估。 决策树顶部使用的特征对更大一部分输入样本的最终预测决策做出贡献；因此，可以使用接受每个特征对最终预测的贡献的样本比例来评估该 特征的相对重要性 。scikit-learn通过将特征贡献的样本比例与纯度减少相结合得到特征的重要性。

通过对多个随机树中的 预期贡献率 （expected activity rates） 取平均，可以减少这种估计的 方差 ，并将其用于特征选择。这被称作平均纯度减少，或MDI。关于MDI以及随机森林特征重要性的更多信息，请参考[L2014]。

下面的例子展示了一个面部识别任务中每个像素的相对重要性，其中重要性由颜色（的深浅）来表示，使用的模型是 `ExtraTreesClassifier` 。



实际上，对于训练完成的模型这些估计值存储在 `feature_importances` 属性中。 这是一个大小为 `(n_features,)` 的数组，其每个元素值为正，并且总和为 1.0。一个元素的值越高，其对应的特征对预测函数的贡献越大。

示例：

- Pixel importances with a parallel forest of trees
- Feature importances with forests of trees

参考资料：

- [L2014] G. Louppe, “Understanding Random Forests: From Theory to Practice”, PhD Thesis, U. of Liege, 2014.

1.11.2.6. 完全随机树嵌入

`RandomTreesEmbedding` 实现了一个无监督的数据转换。 通过由完全随机树构成的森林， `RandomTreesEmbedding` 使用数据最终归属的叶子节点的索引值（编号）对数据进行编码。 该索引以 one-of-K 方式编码，最终形成一个高维的稀疏二进制编码。 这种编码可以被非常高效地计算出来，并且可以作为其他学习任务的基础。 编码的大小和稀疏度可以通过选择树的数量和每棵树的最大深度来确定。对于集成中的每棵树的，每个样本对应其中的一个叶节点。 编码的大小（维度）最多为 `n_estimators * 2 ** max_depth` ，即森林中的叶子节点的最大数。

由于相邻数据点更可能位于树的同一叶子中，该变换可以作为一种隐式地非参数密度估计。

示例：

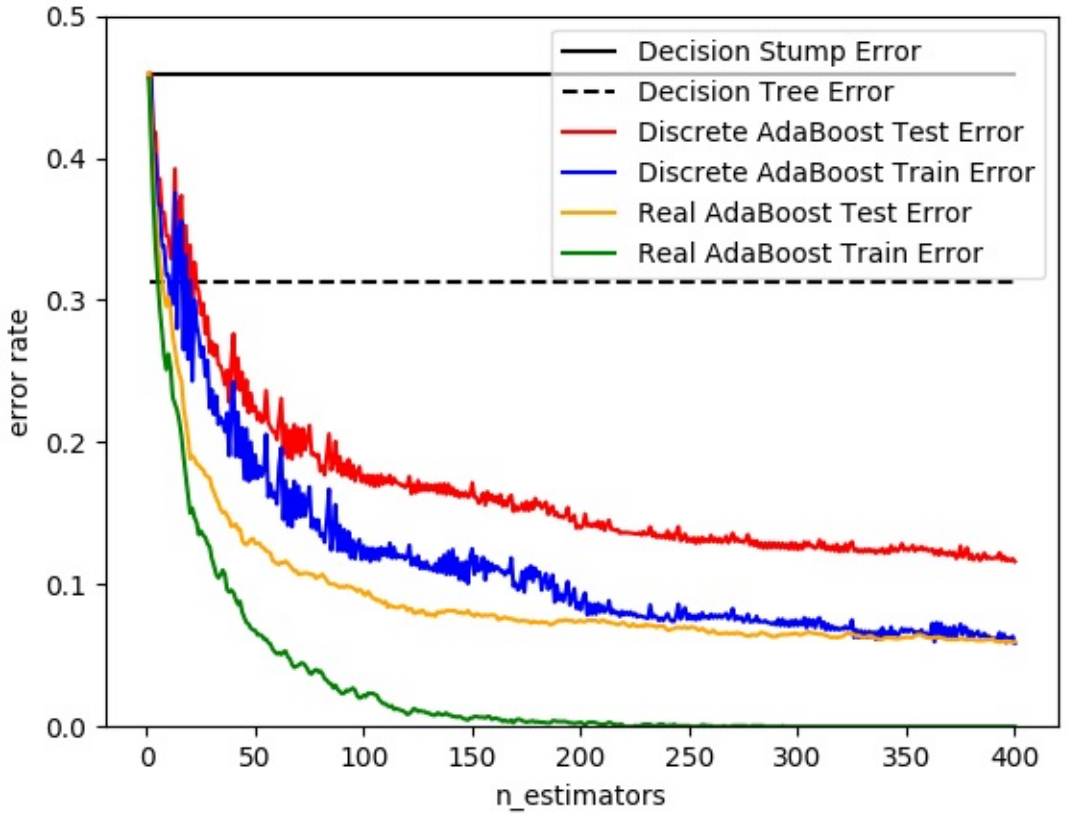
- Hashing feature transformation using Totally Random Trees
- Manifold learning on handwritten digits: Locally Linear Embedding, Isomap… 比较了手写体数字的非线性降维技术。
- Feature transformations with ensembles of trees 比较了基于树的有监督和无监督特征变换.

See also 流形学习 方法也可以用于特征空间的非线性表示，以及降维.

1.11.3. AdaBoost

模型 `sklearn.ensemble` 包含了流行的提升算法 AdaBoost，这个算法是由 Freund and Schapire 在 1995 年提出来的 [FS1995].

AdaBoost 的核心思想是用反复修改的数据（校对者注：主要是修正数据的权重）来训练一系列的弱学习器(一个弱学习器模型仅仅比随机猜测好一点，比如一个简单的决策树)，由这些弱学习器的预测结果通过加权投票(或加权求和)的方式组合，得到我们最终的预测结果。在每一次所谓的提升（boosting）迭代中，数据的修改由应用于每一个训练样本的（新） 的权重 w_1, w_2, …, w_N 组成（校对者注：即修改每一个训练样本应用于新一轮学习器的权重）。 初始化时，将所有弱学习器的权重都设置为 $w_i = 1/N$，因此第一次迭代仅仅是通过原始数据训练出一个弱学习器。在接下来的 连续迭代中，样本的权重逐个地被修改，学习算法也因此要重新应用这些已经修改的权重。在给定的一个迭代中， 那些在上一轮迭代中被预测为错误结果的样本的权重将会被增加，而那些被预测为正确结果的样本的权 重将会被降低。随着迭代次数的增加，那些难以预测的样例的影响将会越来越大，每一个随后的弱学习器都将 会被强迫更加关注那些在之前被错误预测的样例 [HTF].



AdaBoost 既可以用在分类问题也可以用在回归问题中：

- 对于 multi-class 分类， `AdaBoostClassifier` 实现了 AdaBoost-SAMME 和 AdaBoost-SAMME.R [ZZRH2009].
- 对于回归， `AdaBoostRegressor` 实现了 AdaBoost.R2 [D1997].

1.11.3.1. 使用方法

下面的例子展示了如何训练一个包含 100 个弱学习器的 AdaBoost 分类器：

```
>>> from sklearn.model_selection import cross_val_score
>>> from sklearn.datasets import load_iris
>>> from sklearn.ensemble import AdaBoostClassifier

>>> iris = load_iris()
>>> clf = AdaBoostClassifier(n_estimators=100)
>>> scores = cross_val_score(clf, iris.data, iris.target)
>>> scores.mean()
0.9...
```

弱学习器的数量由参数 `n_estimators` 来控制。 `learning_rate` 参数用来控制每个弱学习器对 最终的结果的贡献程度（校对者注：其实应该就是控制每个弱学习器的权重修改速率）。 弱学习器默认使用决策树。不同的弱学习器可以通过参数 `base_estimator` 来指定。 获取一个好的预测结果主要需要调整的参数是 `n_estimators` 和 `base_estimator` 的复杂度（例如：对于弱学习器为决策树的情况，树的深度 `max_depth` 或叶子节点的最小样本数 `min_samples_leaf` 等都是控制树的复杂度的参数）

示例：

- Discrete versus Real AdaBoost 使用 AdaBoost-SAMME 和 AdaBoost-SAMME.R 比较 decision stump， decision tree（决策树）和 boosted decision stump（增强决策树）的分类错误。
- Multi-class AdaBoosted Decision Trees 展示了 AdaBoost-SAMME 和 AdaBoost-SAMME.R 在 multi-class （多类）问题上的性能。
- Two-class AdaBoost 展示了使用 AdaBoost-SAMME 的非线性可分两类问题的决策边界和决策函数值。
- Decision Tree Regression with AdaBoost 使用 AdaBoost.R2 算法证明了回归。

参考资料：

- [FS1995] Y. Freund, and R. Schapire, “A Decision-Theoretic Generalization of On-Line Learning and an Application to Boosting”, 1997.
- [ZZRH2009] J. Zhu, H. Zou, S. Rosset, T. Hastie. “Multi-class AdaBoost”, 2009.
- [D1997] Drucker. “Improving Regressors using Boosting Techniques”, 1997.
- [HTF] T. Hastie, R. Tibshirani and J. Friedman, “Elements of Statistical Learning Ed. 2”, Springer, 2009.

1.11.4. Gradient Tree Boosting（梯度树提升）

Gradient Tree Boosting 或梯度提升回归树（GBRT）是对于任意的可微损失函数的提升算法的泛化。 GBRT 是一个准确高效的现有程序， 它既能用于分类问题也可以用于回归问题。梯度树提升模型被应用到各种领域，包括网页搜索排名和生态领域。

GBRT 的优点：

- 对混合型数据的自然处理（异构特征）
- 强大的预测能力
- 在输出空间中对异常点的鲁棒性(通过具有鲁棒性的损失函数实现)

GBRT 的缺点：

- 可扩展性差（校对者注：此处的可扩展性特指在更大规模的数据集/复杂度更高的模型上使用的能力，而非我们通常说的功能的扩展性；GBRT 支持自定义的损失函数，从这个角度看它的扩展性还是很强的！）。由于提升算法的有序性(也就是说下一步的结果依赖于上一步)，因此很难做并行.

模块 `sklearn.ensemble` 通过梯度提升树提供了分类和回归的方法.

注意：在LightGBM的启发下，Scikit-learn 0.21引入了两种新的梯度提升树的实验实现，即 HistGradientBoostingClassifier和HistGradientBoostingRegressor。这些快速估计器首先将输入样本X放入整数值的箱子(通常是256个箱子)中，这极大地减少了需要考虑的分裂点的数量，并允许算法利用基于整数的数据结构(直方图)，而不是依赖于排序后的连续值。

当样本数量大于数万个样本时，基于直方图的新估计值可以比连续估计值快几个数量级。这些新的估计器的API略有不同，目前还不支持GradientBoostingClassifier和GradientBoostingRegressor的一些特性。

这些新的评估器目前仍处于试验阶段:它们的预测和API可能会在没有任何弃用周期的情况下发生变化。要使用它们，您需要显式地导入 `enable_hist_gradient_boost` ：

```
>> # explicitly require this experimental feature>> from sklearn.experimental import
enable_hist_gradient_boosting  # noqa>> # now you can import normally from ensemble>> from
sklearn.ensemble import HistGradientBoostingClassifier
```

下面的指南只关注GradientBoostingClassifier和GradientBoostingRegressor，这可能是小样本量的首选，因为在这个设置中，装箱可能会导致分割点过于接近。

1.11.4.1. 分类

`GradientBoostingClassifier` 既支持二分类又支持多分类问题。 下面的例子展示了如何训练一个包含 100 个决策树弱学习器的梯度提升分类器：

```
>>> from sklearn.datasets import make_hastie_10_2
>>> from sklearn.ensemble import GradientBoostingClassifier

>>> X, y = make_hastie_10_2(random_state=0)
>>> X_train, X_test = X[:2000], X[2000:]
>>> y_train, y_test = y[:2000], y[2000:]

>>> clf = GradientBoostingClassifier(n_estimators=100, learning_rate=1.0,
...     max_depth=1, random_state=0).fit(X_train, y_train)
>>> clf.score(X_test, y_test)
0.913...
```

弱学习器(例如:回归树)的数量由参数 `n_estimators` 来控制；每个树的大小可以通过由参数 `max_depth` 设置树的深度，或者由参数 `max_leaf_nodes` 设置叶子节点数目来控制。 `learning_rate` 是一个在 (0,1] 之间的超参数，这个参数通过 shrinkage(缩减步长) 来控制过拟合。

> 注意:超过两类的分类问题需要在每一次迭代时推导 `n_classes` 个回归树。因此，所有的需要推导的树数量等于 `n_classes * n_estimators` 。对于拥有大量类别的数据集我们强烈推荐使用 `RandomForestClassifier` 来代替 `GradientBoostingClassifier` 。

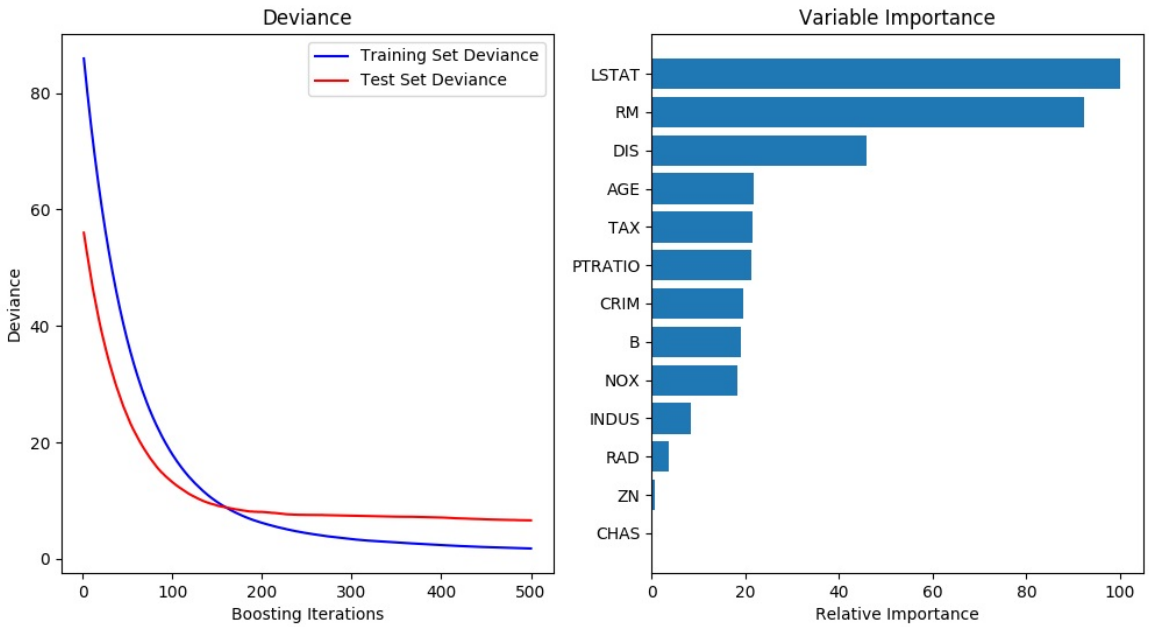
1.11.4.2. 回归

对于回归问题 `GradientBoostingRegressor` 支持一系列 different loss functions ，这些损失函数可以通过参数 `loss` 来指定；对于回归问题默认的损失函数是最小二乘损失函数(`'ls'`)。

```
>>> import numpy as np
>>> from sklearn.metrics import mean_squared_error
>>> from sklearn.datasets import make_friedman1
>>> from sklearn.ensemble import GradientBoostingRegressor

>>> X, y = make_friedman1(n_samples=1200, random_state=0, noise=1.0)
>>> X_train, X_test = X[:200], X[200:]
>>> y_train, y_test = y[:200], y[200:]
>>> est = GradientBoostingRegressor(n_estimators=100, learning_rate=0.1,
...     max_depth=1, random_state=0, loss='ls').fit(X_train, y_train)
>>> mean_squared_error(y_test, est.predict(X_test))
5.00...
```

下图展示了应用损失函数为最小二乘损失，基学习器个数为 500 的 `GradientBoostingRegressor` 来处理 `sklearn.datasets.load_boston` 数据集的结果。左图表示每一次迭代的训练误差和测试误差。每一次迭代的训练误差保存在提升树模型的 `trainscore` 属性中，每一次迭代的测试误差能够通过 `staged_predict` 方法获取，该方法返回一个生成器，用来产生每一 个迭代的预测结果。类似下面这样的图表，可以用于决定最优的树的数量，从而进行提前停止。右图表示每个特征的重要性，它 可以通过 `featureimportances` 属性来获取.



示例：

- Gradient Boosting regression
- Gradient Boosting Out-of-Bag estimates

1.11.4.3. 训练额外的弱学习器

`GradientBoostingRegressor` 和 `GradientBoostingClassifier` 都支持设置参数 `warm_start=True` ，这样设置允许我们在已经训练的模型上面添加更多的估计器。

```
>>> _ = est.set_params(n_estimators=200, warm_start=True)  # set warm_start and new nr of trees
>>> _ = est.fit(X_train, y_train) # fit additional 100 trees to est
>>> mean_squared_error(y_test, est.predict(X_test))
3.84...
```

1.11.4.4. 控制树的大小

回归树基学习器的大小定义了可以被梯度提升模型捕捉到的变量（即特征）相互作用（即多个特征共同对预测产生影响）的程度。 通常一棵深度为 `h` 的树能捕获到秩为 `h` 的相互作用。这里有两种控制单棵回归树大小的方法。

如果你指定 `max_depth=h` ，那么将会产生一个深度为 `h` 的完全二叉树。这棵树将会有（至多） `2h` 个叶子节点和 `2 h - 1` 个切分节点。

另外，你能通过参数 `max_leaf_nodes` 指定叶子节点的数量来控制树的大小。在这种情况下，树将会使用最优优先搜索来生成，这种搜索方式是通过每次选取对不纯度提升最大的节点来展开。一棵 `max_leaf_nodes=k` 的树拥有 `k - 1` 个切分节点，因此可以模拟秩最高达到 `max_leaf_nodes - 1` 的相互作用（即 `max_leaf_nodes - 1` 个特征共同决定预测值）。

我们发现 `max_leaf_nodes=k` 可以给出与 `max_depth=k-1` 品质相当的结果，但是其训练速度明显更快，同时也会

以多一点的训练误差作为代价。参数 `max_leaf_nodes` 对应于文章 [F2001] 中梯度提升章节中的变量 `J` ，同时与 R 语言的 gbm 包的参数 `interaction.depth` 相关，两者间的关系是 `max_leaf_nodes == interaction.depth + 1` 。

1.11.4.5. Mathematical formulation（数学公式）

GBRT 可以认为是以下形式的可加模型：

$$F(x) = \sum_{m=1}^{M} \gamma_m h_m(x)$$

其中 $h_m(x)$ 是基本函数，在提升算法场景中它通常被称作 *weak learners* 。梯度树提升算法（Gradient Tree Boosting）使用固定大小 的 decision trees 作为弱分类器，决策树本身拥有的一些特性使它能够在提升过程中变得有价值， 即处理混合类型数据以及构建具有复杂功能模型的能力.

与其他提升算法类似， GBRT 利用前向分步算法思想构建加法模型：

$$F_m(x) = F_{m-1}(x) + \gamma_m h_m(x)$$

在每一个阶段中，基于当前模型 F_{m-1} 和拟合函数 $F_{m-1}(x_i)$ 选择合适的决策树函数 $h_m(x)$ ，从而最小化损失函数 L 。

$$F_m(x) = F_{m-1}(x) + \arg\min_h \sum_{i=1}^{n} L(y_i, F_{m-1}(x_i) - h(x))$$

初始模型 F_0 根据不同的问题指定，对于最小二乘回归，通常选择目标值的平均值.

> 注意：初始化模型也能够通过 `init` 参数来指定，但传递的对象需要实现 `fit` 和 `predict` 函数。

梯度提升（Gradient Boosting）尝试通过最速下降法以数字方式解决这个最小化问题.最速下降方向是在当前模型 F_{m-1} 下的损失函数的负梯度方向，其中模型 F_{m-1} 可以计算任何可微损失函数：

$$F_m(x) = F_{m-1}(x) + \gamma_m \sum_{i=1}^{n} \nabla_F L(y_i, F_{m-1}(x_i))$$

其中步长 γ_m 通过如下方式线性搜索获得：

$$\gamma_m = \arg\min_{\gamma} \sum_{i=1}^{n} L(y_i, F_{m-1}(x_i) - \gamma \frac{\partial L(y_i, F_{m-1}(x_i))}{\partial F_{m-1}(x_i)})$$

该算法处理分类和回归问题不同之处在于具体损失函数的使用。

1.11.4.5.1. Loss Functions（损失函数）

以下是目前支持的损失函数，具体损失函数可以通过参数 `loss` 指定：

- 回归 (Regression)
  - Least squares (`'ls'`): 由于其优越的计算性能，该损失函数成为回归算法中的自然选择。 初始模型 （校对者注：即损失函数的初始值，下同） 通过目标值的均值给出。
  - Least absolute deviation (`'lad'`): 回归中具有鲁棒性的损失函数，初始模型通过目 标值的中值给出。
  - Huber (`'huber'`): 回归中另一个具有鲁棒性的损失函数，它是最小二乘和最小绝对偏差两者的结合. 其利用 `alpha` 来控制模型对于异常点的敏感度(详细介绍请参考 [F2001]).
  - Quantile (`'quantile'`): 分位数回归损失函数.用 `0 < alpha < 1` 来指定分位数这个损 失函数可以用来产生预测间隔。（详见 Prediction Intervals for Gradient Boosting Regression ）。
- 分类 (Classification)
  - Binomial deviance (`'deviance'`): 对于二分类问题(提供概率估计)即负的二项 log 似然损失函数。模型以 log 的比值比来初始化。
  - Multinomial deviance (`'deviance'`): 对于多分类问题的负的多项log似然损失函数具有 `n_classes` 个互斥的类。提供概率估计。 初始模型由每个类的先验概率给出.在每一次迭代中 `n_classes` 回归树被构建，这使得 GBRT 在处理多类别数据集时相当低效。
  - Exponential loss (`'exponential'`): 与 `AdaBoostClassifier` 具有相同的损失函数。与 `'deviance'` 相比，对被错误标记的样本的鲁棒性较差，仅用于在二分类问题。

1.11.4.6. Regularization（正则化）

1.11.4.6.1. 收缩率 （Shrinkage）

[F2001]提出一个简单的正则化策略，通过一个因子 ν 来衡量每个弱分类器对于最终结果的贡献：

$$F_m(x) = F_{m-1}(x) + \nu\gamma_m h_m(x)$$

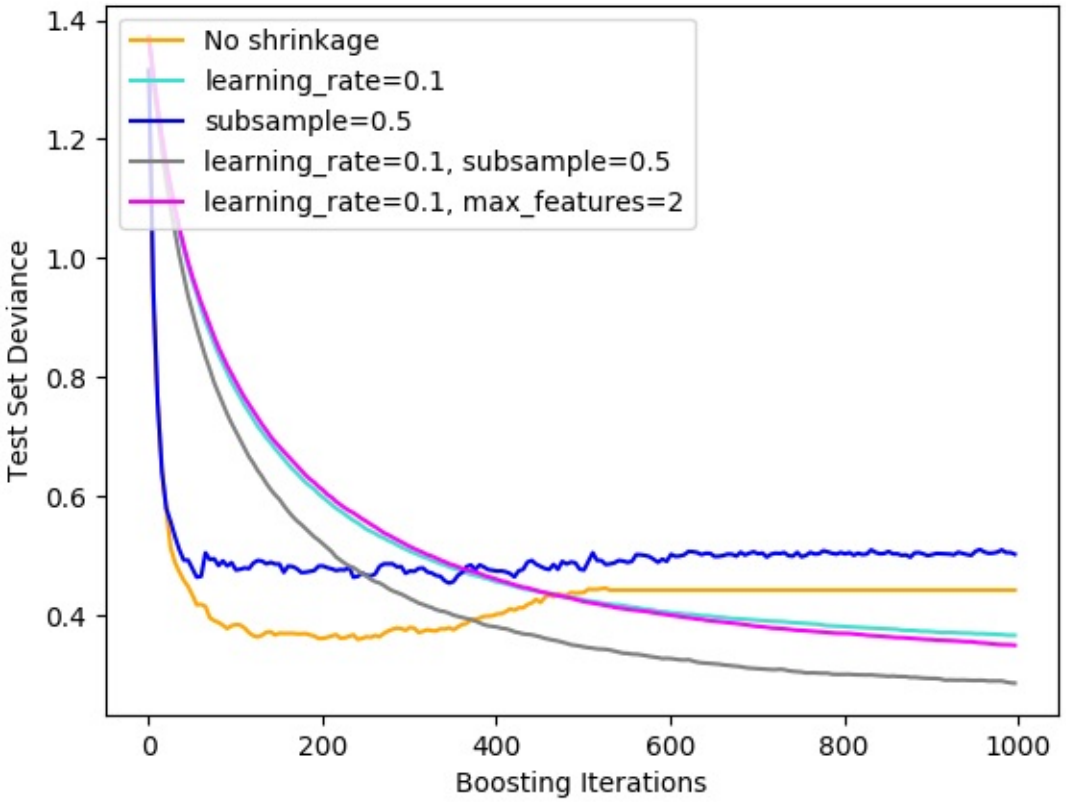
参数 ν 由于它可以控制梯度下降的步长， 因此也叫作 **learning rate** ，它可以通过 `learning_rate` 参数来设置.

在训练一定数量的弱分类器时，参数 `learning_rate` 和参数 `n_estimators` 之间有很强的制约关系。 较小的 `learning_rate` 需要大量的弱分类器才能维持训练误差的稳定。经验表明数值较小的 `learning_rate` 将会得到更好的测试误差。 [HTF2009]推荐把 `learning_rate` 设置为一个较小的常数（例如： `learning_rate <= 0.1` ）同时通过提前停止策略来选择合适的 `n_estimators` . 有关 `learning_rate` 和 `n_estimators` 更详细的讨论可以参考 [R2007]

1.11.4.6.2. 子采样 （Subsampling）

[F1999]提出了随机梯度提升，这种方法将梯度提升（gradient boosting）和 bootstrap averaging(bagging) 相结合。在每次迭代中，基分类器是通过抽取所有可利用训练集中一小部分的 `subsample` 训练得到的子样本采用无放回的方式采样。 `subsample` 参数的值一般设置为 `0.5` 。

下图表明了收缩与否和子采样对于模型拟合好坏的影响。我们可以明显看到指定收缩率比没有收缩拥有更好的表现。而将子采样和收缩率相结合能进一步的提高模型的准确率。相反，使用子采样而不使用收缩的结果十分糟糕。



另一个减少方差的策略是特征子采样，这种方法类似于 `RandomForestClassifier` 中的随机分割。子采样的特征数可以通过参数 `max_features` 来控制。

> 注意：采用一个较小的 `max_features` 值能大大缩减模型的训练时间。

随机梯度提升允许计算测试偏差的袋外估计值（Out-of-bag），方法是计算那些不在自助采样之内的样本偏差的改进。这个改进保存在属性 `oobimprovement` 中 `oobimprovement[i]` 如果将第 i 步添加到当前预测中，则可以改善 OOB 样本的损失。袋外估计可以使用在模型选择中，例如决定最优迭代次数。 OOB 估计通常都很悲观，因此我们推荐使用交叉验证来代替它，而当交叉验证太耗时时我们就只能使用 OOB 了。

> 示例：
>
> - Gradient Boosting regularization
> - Gradient Boosting Out-of-Bag estimates
> - OOB Errors for Random Forests

1.11.4.7. Interpretation（解释性）

通过简单地可视化树结构可以很容易地解释单个决策树，然而对于梯度提升模型来说，一般拥有数百棵/种回归树，将每一棵树都可视化来解释整个模型是很困难的。幸运的是，有很多关于总结和解释梯度提升模型的技术。

1.11.4.7.1. Feature importance（特征重要性）

通常情况下每个特征对于预测目标的影响是不同的.在很多情形下大多数特征和预测结果是无关的。当解释一个模型时，第一个问题通常是：这些重要的特征是什么？他们如何在预测目标方面产生积极的影响的？

单个决策树本质上是通过选择最佳切分点来进行特征选择.这个信息可以用来评定每个特征的重要性。基本思想是：在树的分割点中使用的特征越频繁，特征越重要。 这个特征重要性的概念可以通过简单地平均每棵树的特征重要性来扩展到决策树集合。（详见 特征重要性评估 ）。

对于一个训练好的梯度提升模型，其特征重要性分数可以通过属性 `featureimportances` 查看：

```
>>> from sklearn.datasets import make_hastie_10_2
>>> from sklearn.ensemble import GradientBoostingClassifier

>>> X, y = make_hastie_10_2(random_state=0)
>>> clf = GradientBoostingClassifier(n_estimators=100, learning_rate=1.0,
...     max_depth=1, random_state=0).fit(X, y)
>>> clf.feature_importances_
array([0.10..., 0.10..., 0.11..., ...
```

> 示例：
>
> - Gradient Boosting regression

1.11.5. Voting Classifier（投票分类器）

`VotingClassifier` （投票分类器）的原理是结合了多个不同的机器学习分类器,并且采用多数表决（majority vote）（硬投票） 或者平均预测概率（软投票）的方式来预测分类标签。 这样的分类器可以用于一组同样表现良好的模型,以便平衡它们各自的弱点。

1.11.5.1. 多数类标签 （又称为 多数/硬投票）

在多数投票中，对于每个特定样本的预测类别标签是所有单独分类器预测的类别标签中票数占据多数（模式）的类别标签。

例如，如果给定样本的预测是

- classifier 1 -> class 1
- classifier 2 -> class 1
- classifier 3 -> class 2

类别 1 占据多数,通过 `voting='hard'` 参数设置投票分类器为多数表决方式，会得到该样本的预测结果是类别 1 。

在平局的情况下,投票分类器（VotingClassifier）将根据升序排序顺序选择类标签。 例如，场景如下：

- classifier 1 -> class 2
- classifier 2 -> class 1

这种情况下， class 1 将会被指定为该样本的类标签。

1.11.5.1.1. 用法

以下示例显示如何训练多数规则分类器：

```
>>> from sklearn import datasets
>>> from sklearn.model_selection import cross_val_score
>>> from sklearn.linear_model import LogisticRegression
>>> from sklearn.naive_bayes import GaussianNB
>>> from sklearn.ensemble import RandomForestClassifier
>>> from sklearn.ensemble import VotingClassifier

>>> iris = datasets.load_iris()
>>> X, y = iris.data[:, 1:3], iris.target

>>> clf1 = LogisticRegression(solver='lbfgs', multi_class='multinomial',
...                           random_state=1)
>>> clf2 = RandomForestClassifier(n_estimators=50, random_state=1)
>>> clf3 = GaussianNB()

>>> eclf = VotingClassifier(estimators=[('lr', clf1), ('rf', clf2), ('gnb', clf3)], voting='hard')

>>> for clf, label in zip([clf1, clf2, clf3, eclf], ['Logistic Regression', 'Random Forest', 'naive Bayes',
'Ensemble']):
...     scores = cross_val_score(clf, X, y, cv=5, scoring='accuracy')
...     print("Accuracy: %0.2f (+/- %0.2f) [%s]" % (scores.mean(), scores.std(), label))
Accuracy: 0.95 (+/- 0.04) [Logistic Regression]
Accuracy: 0.94 (+/- 0.04) [Random Forest]
Accuracy: 0.91 (+/- 0.04) [naive Bayes]
Accuracy: 0.95 (+/- 0.04) [Ensemble]
```

1.11.5.2. 加权平均概率 （软投票）

与多数投票（硬投票）相比，软投票将类别标签返回为预测概率之和的 `argmax` 。

具体的权重可以通过权重参数 `weights` 分配给每个分类器。当提供权重参数 `weights` 时，收集每个分类器的预测分类概率， 乘以分类器权重并取平均值。然后将具有最高平均概率的类别标签确定为最终类别标签。

为了用一个简单的例子来说明这一点，假设我们有 3 个分类器和一个 3 类分类问题，我们给所有分类器赋予相等的权重：`w1 = 1,w2 = 1,w3 = 1` 。

样本的加权平均概率计算如下：

分类器	类别 1	类别 2	类别 3
分类器 1	w1 *0.2*	w1 0.5	w1 *0.3*
分类器 2	w2 0.6	w2 *0.3*	w2 0.1
分类器 3	w3 *0.3*	w3 0.4	w3 * 0.3
加权平均的结果	0.37	0.4	0.23

这里可以看出，预测的类标签是 2，因为它具有最大的平均概率.

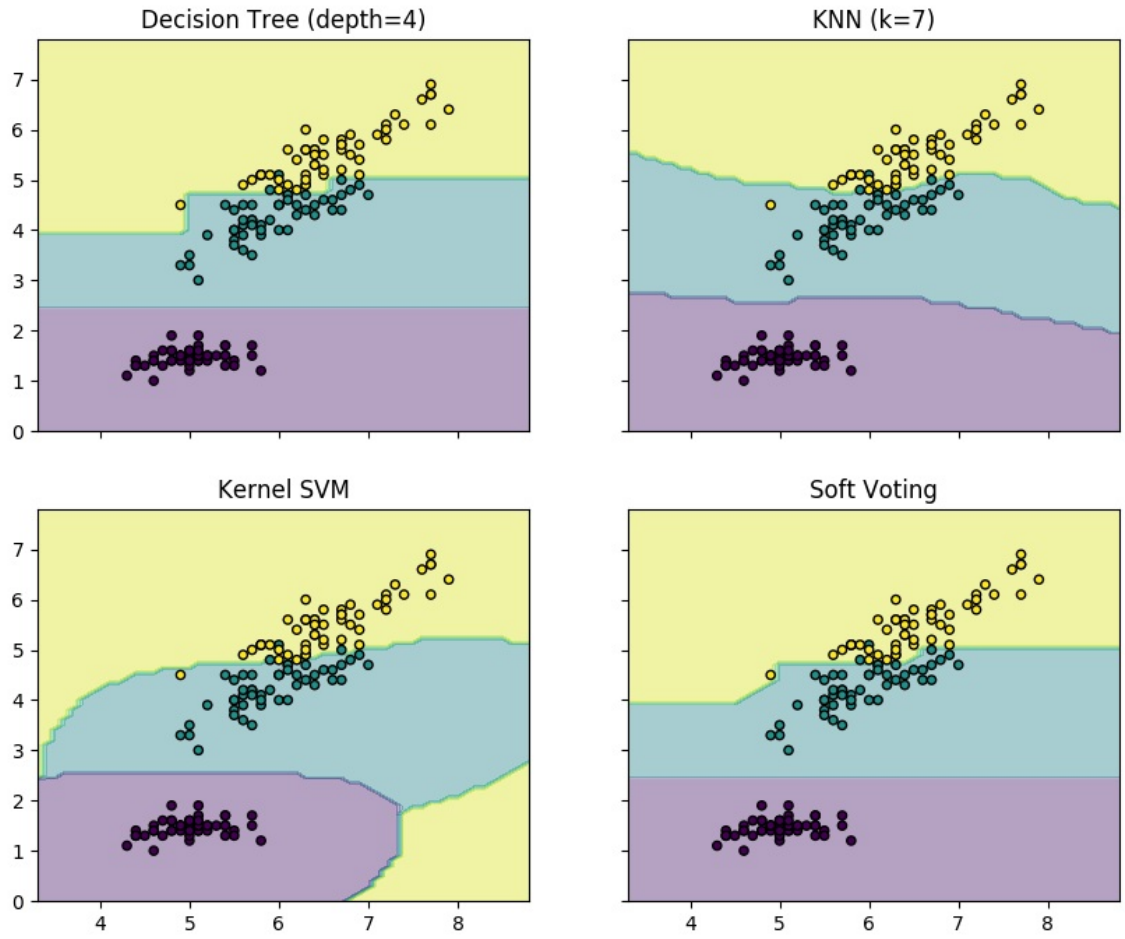
下边的示例程序说明了当软投票分类器（soft VotingClassifier）是基于线性支持向量机（linear SVM）、决策树（Decision Tree）、K 近邻（K-nearest）分类器时，决策域可能的变化情况：

```
>>> from sklearn import datasets
>>> from sklearn.tree import DecisionTreeClassifier
>>> from sklearn.neighbors import KNeighborsClassifier
>>> from sklearn.svm import SVC
>>> from itertools import product
>>> from sklearn.ensemble import VotingClassifier

>>> # Loading some example data
>>> iris = datasets.load_iris()
>>> X = iris.data[:, [0, 2]]
>>> y = iris.target

>>> # Training classifiers
>>> clf1 = DecisionTreeClassifier(max_depth=4)
>>> clf2 = KNeighborsClassifier(n_neighbors=7)
>>> clf3 = SVC(gamma='scale', kernel='rbf', probability=True)
>>> eclf = VotingClassifier(estimators=[('dt', clf1), ('knn', clf2), ('svc', clf3)],
...                         voting='soft', weights=[2, 1, 2])

>>> clf1 = clf1.fit(X, y)
>>> clf2 = clf2.fit(X, y)
>>> clf3 = clf3.fit(X, y)
>>> eclf = eclf.fit(X, y)
```



1.11.5.3. 投票分类器（VotingClassifier）在网格搜索（GridSearchCV）应用

为了调整每个估计器的超参数，`VotingClassifier` 也可以和 `GridSearchCV` 一起使用：

```
>>> from sklearn.model_selection import GridSearchCV
>>> clf1 = LogisticRegression(random_state=1)
>>> clf2 = RandomForestClassifier(random_state=1)
>>> clf3 = GaussianNB()
>>> eclf = VotingClassifier(estimators=[('lr', clf1), ('rf', clf2), ('gnb', clf3)], voting='soft')

>>> params = {'lr__C': [1.0, 100.0], 'rf__n_estimators': [20, 200],}

>>> grid = GridSearchCV(estimator=eclf, param_grid=params, cv=5)
>>> grid = grid.fit(iris.data, iris.target)
```

1.11.5.3.1. 用法

为了通过预测的类别概率来预测类别标签(投票分类器中的 scikit-learn estimators 必须支持

`predict_proba` 方法)：

```
>>> eclf = VotingClassifier(estimators=[('lr', clf1), ('rf', clf2), ('gnb', clf3)], voting='soft')
```

可选地，也可以为单个分类器提供权重：

```
>>> eclf = VotingClassifier(estimators=[('lr', clf1), ('rf', clf2), ('gnb', clf3)], voting='soft', weights=
[2,5,1])
```

1.11.6. 投票回归器(Voting Regressor)

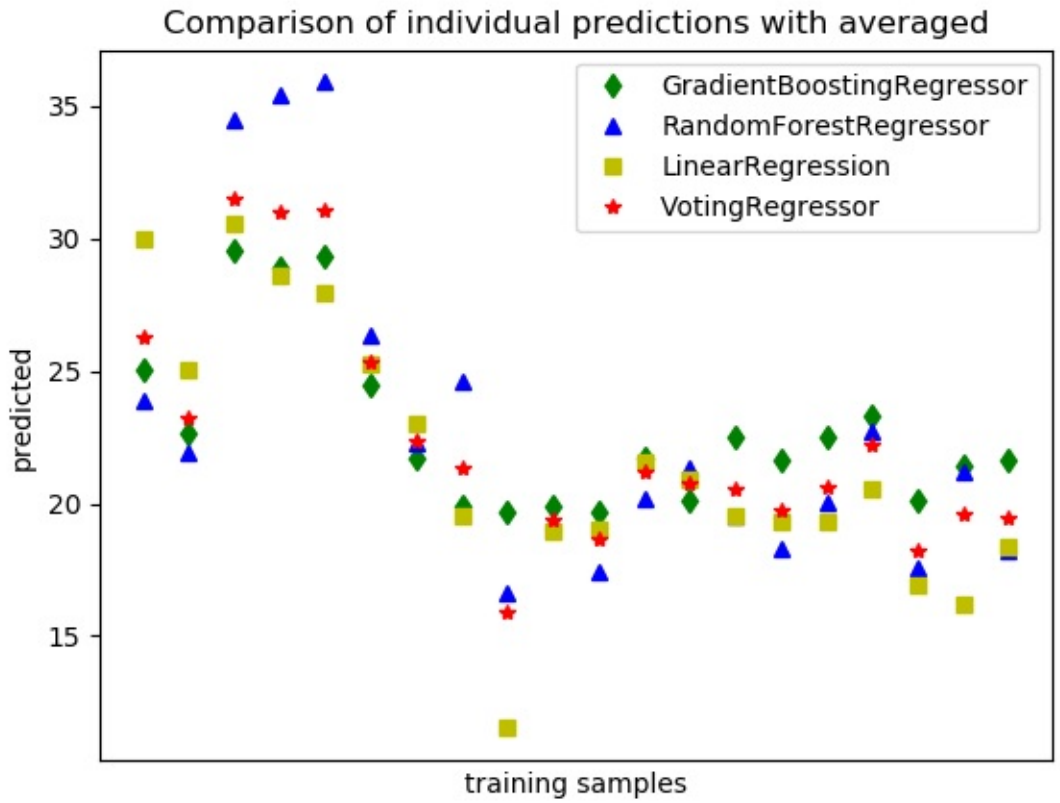
投票回归器背后的思想是将概念上不同的机器学习回归器组合起来，并返回平均预测值。这样一个回归器对于一组同样表现良好的模型是有用的，以便平衡它们各自的弱点。

下面的例子展示了如何匹配投票回归器：

```
>>> from sklearn import datasets
>>> from sklearn.ensemble import GradientBoostingRegressor
>>> from sklearn.ensemble import RandomForestRegressor
>>> from sklearn.linear_model import LinearRegression
>>> from sklearn.ensemble import VotingRegressor

>>> # Loading some example data
>>> boston = datasets.load_boston()
>>> X = boston.data
>>> y = boston.target

>>> # Training classifiers
>>> reg1 = GradientBoostingRegressor(random_state=1, n_estimators=10)
>>> reg2 = RandomForestRegressor(random_state=1, n_estimators=10)
>>> reg3 = LinearRegression()
>>> ereg = VotingRegressor(estimators=[('gb', reg1), ('rf', reg2), ('lr', reg3)])
>>> ereg = ereg.fit(X, y)
```



> 示例：
>
> - Plot individual and voting regression predictions

我们一直在努力

apachecn/sklearn-doc-zh



1.12. 多类和多标签算法

校验者：@溪流-十四号@大魔王飞仙@Loopy翻译者：@v

> 警告 scikit-learn中的所有分类器都可以开箱即用进行多类分类。除非您想尝试不同的多类策略，否则无需使用 `sklearn.multiclass` 模块。

`sklearn.multiclass` 模块采用了 元评估器 ，通过把 `多类` 和 `多标签` 分类问题分解为 二元分类问题去解决。这同样适用于多目标回归问题。

- **Multiclass classification**多类分类** 意味着一个分类任务需要对多于两个类的数据进行分类。比如，对一系列的橘子，苹果或者梨的图片进行分类。多类分类假设每一个样本有且仅有一个标签：一个水果可以被归类为苹果，也可以 是梨，但不能同时被归类为两类。

- **Multilabel classification**多标签分类** 给每一个样本分配一系列标签。这可以被认为是预测不相互排斥的数据点的属性，例如与文档类型相关的主题。一个文本可以归类为任意类别，例如可以同时为政治、金融、 教育相关或者不属于以上任何类别。

- **Multioutput regression**多输出分类** 为每个样本分配一组目标值。这可以认为是预测每一个样本的多个属性，比如说一个具体地点的风的方向和大小。

- **Multioutput-multiclass classification** and **multi-task classification**多输出-多类分类和多任务分类** 意味着单个的评估器要解决多个联合的分类任务。这是只考虑二分类的 multi-label classification 和 multi-class classification 任务的推广。 此类问题输出的格式是一个二维数组或者一个稀疏矩阵

每个输出变量的标签集合可以是各不相同的。比如说，一个样本可以将“梨”作为一个输出变量的值，这个输出变 量在一个含有“梨”、“苹果”等水果种类的有限集合中取可能的值；将“蓝色”或者“绿色”作为第二个输出变量的值， 这个输出变量在一个含有“绿色”、“红色”、“蓝色”等颜色种类的有限集合中取可能的值…

这意味着任何处理 multi-output multiclass or multi-task classification 任务的分类器，在特殊的情况下支持 multi-label classification 任务。Multi-task classification 与具有不同模型公式 的 multi-output classification 相似。详细情况请查阅相关的分类器的文档。

所有的 scikit-learn 分类器都能处理 multiclass classification 任务， 但是 `sklearn.multiclass` 提供的元评估器允许改变在处理超过两类数据时的方式，因为这会对分类器的性能产生影响 （无论是在泛化误差或者所需要的计算资源方面）

下面是按照 scikit-learn 策略分组的分类器的总结，如果你使用其中的一个，则不需要此类中的元评估器，除非你想要自定义的多分类方式。

- 固有的多类分类器：
  - `sklearn.naive_bayes.BernoulliNB`
  - `sklearn.tree.DecisionTreeClassifier`
  - `sklearn.tree.ExtraTreeClassifier`
  - `sklearn.ensemble.ExtraTreesClassifier`
  - `sklearn.naive_bayes.GaussianNB`
  - `sklearn.neighbors.KNeighborsClassifier`
  - `sklearn.semi_supervised.LabelPropagation`

    - `sklearn.semi_supervised.LabelSpreading`
    - `sklearn.discriminant_analysis.LinearDiscriminantAnalysis`
    - `sklearn.svm.LinearSVC` (setting multi_class="crammer_singer")
    - `sklearn.linear_model.LogisticRegression` (setting multi_class="multinomial")
    - `sklearn.linear_model.LogisticRegressionCV` (setting multi_class="multinomial")
    - `sklearn.neural_network.MLPClassifier`
    - `sklearn.neighbors.NearestCentroid`
    - `sklearn.discriminant_analysis.QuadraticDiscriminantAnalysis`
    - `sklearn.neighbors.RadiusNeighborsClassifier`
    - `sklearn.ensemble.RandomForestClassifier`
    - `sklearn.linear_model.RidgeClassifier`
    - `sklearn.linear_model.RidgeClassifierCV`
- **1对1的多类分类器:**
    - `sklearn.svm.NuSVC`
    - `sklearn.svm.SVC`.
    - `sklearn.gaussian_process.GaussianProcessClassifier` (setting multi_class = "one_vs_one")
- **1对多的多类分类器:**
    - `sklearn.ensemble.GradientBoostingClassifier`
    - `sklearn.gaussian_process.GaussianProcessClassifier` (setting multi_class = "one_vs_rest")
    - `sklearn.svm.LinearSVC` (setting multi_class="ovr")
    - `sklearn.linear_model.LogisticRegression` (setting multi_class="ovr")
    - `sklearn.linear_model.LogisticRegressionCV` (setting multi_class="ovr")
    - `sklearn.linear_model.SGDClassifier`
    - `sklearn.linear_model.Perceptron`
    - `sklearn.linear_model.PassiveAggressiveClassifier`
- **支持多标签分类的分类器:**
    - `sklearn.tree.DecisionTreeClassifier`
    - `sklearn.tree.ExtraTreeClassifier`
    - `sklearn.ensemble.ExtraTreesClassifier`
    - `sklearn.neighbors.KNeighborsClassifier`
    - `sklearn.neural_network.MLPClassifier`
    - `sklearn.neighbors.RadiusNeighborsClassifier`
    - `sklearn.ensemble.RandomForestClassifier`
    - `sklearn.linear_model.RidgeClassifierCV`
- **支持多类-多输出分类的分类器:**
    - `sklearn.tree.DecisionTreeClassifier`
    - `sklearn.tree.ExtraTreeClassifier`
    - `sklearn.ensemble.ExtraTreesClassifier`
    - `sklearn.neighbors.KNeighborsClassifier`
    - `sklearn.neighbors.RadiusNeighborsClassifier`
    - `sklearn.ensemble.RandomForestClassifier`

> 警告:目前, `sklearn.metrics` 中没有评估方法能够支持多输出多类分类任务。

1.12.1. 多标签分类格式

在 multilabel learning 中，二元分类任务的合集表示为二进制数组：每一个样本是大小为 (n_samples, n_classes) 的二维数组中的一行二进制值，比如非0元素，表示为对应标签的 子集。 一个数组 `np.array([[1, 0, 0], [0, 1, 1], [0, 0, 0]])` 表示第一个样本属于第 0 个标签，第二个样本属于第一个和第二个标签，第三个样本不属于任何标签。

通过一系列的标签来产生多标签数据可能更为直观。 `MultiLabelBinarizer` 转换器可以用来在标签接口和格式指示器接口之间进行转换。

```
>>> from sklearn.preprocessing import MultiLabelBinarizer
>>> y = [[2, 3, 4], [2], [0, 1, 3], [0, 1, 2, 3, 4], [0, 1, 2]]
>>> MultiLabelBinarizer().fit_transform(y)
array([[0, 0, 1, 1, 1],
       [0, 0, 1, 0, 0],
       [1, 1, 0, 1, 0],
       [1, 1, 1, 1, 1],
       [1, 1, 1, 0, 0]])
```

1.12.2. 1对其余

这个方法也被称为 **1**对多， 在 `OneVsRestClassifier` 模块中执行。 这个方法在于每一个类都将用一个分类器进行拟合。 对于每一个分类器，该类将会和其他所有的类有所区别。除了它的计算效率之外 (只需要 *n_classes* 个分类器)， 这种方法的优点是它具有可解释性。 因为每一个类都可以通过有且仅有一个分类器来代表，所以通过检查一个类相关的分类器就可以获得该类的信息。这是最常用的方法，也是一个合理的默认选择。

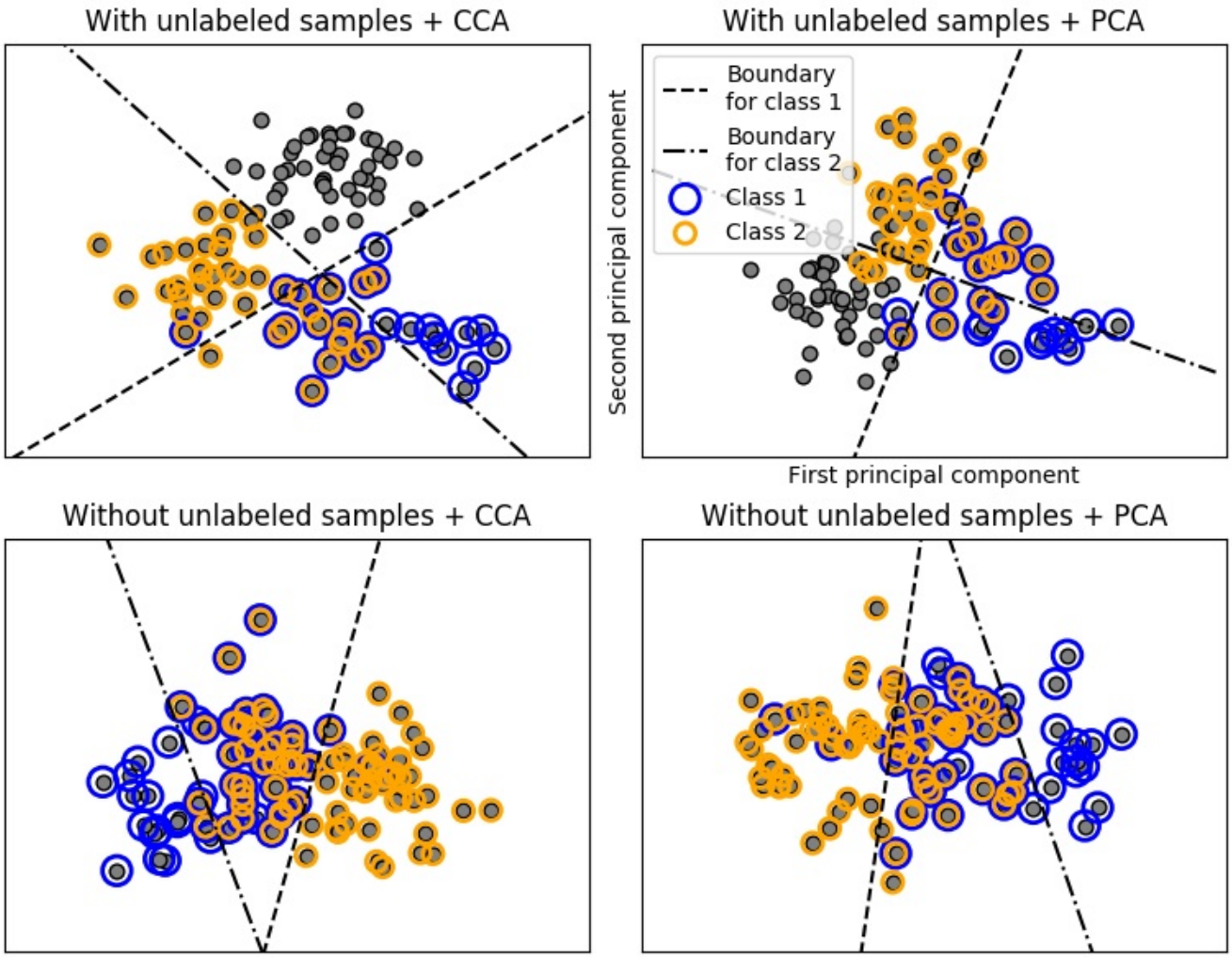
1.12.2.1. 多类学习

下面是一个使用 OvR 的一个例子:

```
>>> from sklearn import datasets
>>> from sklearn.multiclass import OneVsRestClassifier
>>> from sklearn.svm import LinearSVC
>>> iris = datasets.load_iris()
>>> X, y = iris.data, iris.target
>>> OneVsRestClassifier(LinearSVC(random_state=0)).fit(X, y).predict(X)
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 1, 2, 2, 2, 1, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2])
```

1.12.2.2. 多标签学习

`OneVsRestClassifier` 1对其余分类器 也支持 multilabel classification. 要使用该功能，给分类器提供一个指示矩阵，比如 [i,j] 表示第i个样本中的第j个标签。



示例：

- Multilabel classification

1.12.3. 1对1

`OneVsOneClassifier` 1对1分类器 将会为每一对类别构造出一个分类器，在预测阶段，收到最多投票的类别将会被挑选出来。 当存在结时（两个类具有同样的票数的时候），1对1分类器会选择总分类置信度最高的类，其中总分类置信度是由下层的二元分类器 计算出的成对置信等级累加而成。

因为这需要训练出 `n_classes * (n_classes - 1) / 2` 个分类器，由于复杂度为 O(n_classes^2)，这个方法通常比 one-vs-the-rest 慢。然而，这个方法也有优点，比如说是在没有很好的缩放 `n_samples` 数据的核方法中。 这是由于每个单独的学习问题只涉及一小部分数据，而 one-vs-the-rest 将会使用 `n_classes` 次完整的数据。

1.12.3.1. 多类别学习

下面是一个使用OvO进行多类别学习的例子：

```
>>> from sklearn import datasets
>>> from sklearn.multiclass import OneVsOneClassifier
```

```
>>> from sklearn.svm import LinearSVC
>>> iris = datasets.load_iris()
>>> X, y = iris.data, iris.target
>>> OneVsOneClassifier(LinearSVC(random_state=0)).fit(X, y).predict(X)
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 2, 1, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2])
```

参考资料：

- “Pattern Recognition and Machine Learning. Springer”, Christopher M. Bishop, page 183, (First Edition)

1.12.4. 误差校正输出代码

基于Output-code的方法不同于 one-vs-the-rest 和 one-vs-one。使用这些方法，每一个类将会被映射到欧几里得空间，每一个维度上的值只能为0或者1。另一种解释它的方法是，每一个类被表示为二进制 码（一个 由0 和 1 组成的数组）。保存 location （位置）/ 每一个类的编码的矩阵被称为 code book。编码的大小是前面提到的欧几里得空间的纬度。直观上来说，每一个类应该使用一个唯一的编码，同时，好的 code book 应该能够优化分类的精度。 在实现上，我们使用随机产生的 code book，正如在 [3] 提倡的方式，即使更加详尽的方法可能会在未来被加入其中。

在训练时，code book 每一位的二分类器将会被训练。在预测时，分类器将映射到类空间中选中的点的附近。

在 `OutputCodeClassifier` ， `code_size` 属性允许用户设置将会用到的分类器的数量。 它是类别总数的百分比。

在 0 或 1 之中的一个数字会比 one-vs-the-rest 使用更少的分类器。理论上 `log2(n_classes) / n_classes` 足以明确表示每个类。然而在实际情况中，这也许会导致不太好的精确度，因为 `log2(n_classes)` 小于 n_classes.

比 1 大的数字比 one-vs-the-rest 需要更多的分类器。在这种情况下，一些分类器在理论上会纠正其他分类器的错误，因此命名为 “error-correcting” 。然而在实际上这通常不会发生，因为许多分类器的错误通常意义上来说是相关的。error-correcting output codes 和 bagging 有一个相似的作用效果。

1.12.4.1. 多类别学习

下面是一个使用Output-Codes进行多类别学习的例子：

```
>>> from sklearn import datasets
>>> from sklearn.multiclass import OutputCodeClassifier
>>> from sklearn.svm import LinearSVC
>>> iris = datasets.load_iris()
>>> X, y = iris.data, iris.target
>>> clf = OutputCodeClassifier(LinearSVC(random_state=0),
...                            code_size=2, random_state=0)
>>> clf.fit(X, y).predict(X)
```

```
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1, 1, 2, 1, 1,
 1, 2, 1, 1, 1, 1, 1, 1, 2, 1, 1, 1, 1, 1, 2, 2, 2, 1, 1, 1, 1, 1, 1,
 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
 2, 2, 2, 2, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 1, 2, 2, 2, 1, 1, 2, 2, 2,
 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2])
```

参考资料：

- “Solving multiclass learning problems via error-correcting output codes”, Dietterich T., Bakiri G., Journal of Artificial Intelligence Research 2, 1995.
- [3] “The error coding method and PICTs”, James G., Hastie T., Journal of Computational and Graphical statistics 7, 1998.
- “The Elements of Statistical Learning”, Hastie T., Tibshirani R., Friedman J., page 606 (second-edition) 2008.

1.12.5. 多输出回归

多输出回归支持 `MultiOutputRegressor` 可以被添加到任何回归器中。这个策略包括对每个目标拟合一个回归器。因为每一个目标可以被一个回归器精确地表示，通过检查对应的回归器，可以获取关于目标的信息。 因为 `MultiOutputRegressor` 对于每一个目标可以训练出一个回归器，所以它无法利用目标之间的相关度信息。

以下是 multioutput regression（多输出回归）的示例：

```
>>> from sklearn.datasets import make_regression
>>> from sklearn.multioutput import MultiOutputRegressor
>>> from sklearn.ensemble import GradientBoostingRegressor
>>> X, y = make_regression(n_samples=10, n_targets=3, random_state=1)
>>> MultiOutputRegressor(GradientBoostingRegressor(random_state=0)).fit(X, y).predict(X)
array([[-154.75474165, -147.03498585,  -50.03812219],
       [   7.12165031,    5.12914884,  -81.46081961],
       [-187.8948621 , -100.44373091,   13.88978285],
       [-141.62745778,   95.02891072, -191.48204257],
       [  97.03260883,  165.34867495,  139.52003279],
       [ 123.92529176,   21.25719016,   -7.84253   ],
       [-122.25193977,  -85.16443186, -107.12274212],
       [ -30.170388  ,  -94.80956739,   12.16979946],
       [ 140.72667194,  176.50941682,  -17.50447799],
       [ 149.37967282,  -81.15699552,   -5.72850319]])
```

1.12.6. 多输出分类

Multioutput classification 支持能够被添加到任何带有 `MultiOutputClassifier` 标志的分类器中. 这种方法为每一个目标训练一个分类器。 这就允许产生多目标变量分类器。这种类的目的是扩展评估器用于评估一系列目标函数 (f1,f2,f3…,fn) ，这些函数在一个单独的预测矩阵上进行训练以此来预测一系列的响应 (y1,y2,y3…,yn)。

下面是多输出分类的一个例子：

```
>>> from sklearn.datasets import make_classification
>>> from sklearn.multioutput import MultiOutputClassifier
>>> from sklearn.ensemble import RandomForestClassifier
>>> from sklearn.utils import shuffle
>>> import numpy as np
>>> X, y1 = make_classification(n_samples=10, n_features=100, n_informative=30, n_classes=3, random_state=1)
>>> y2 = shuffle(y1, random_state=1)
>>> y3 = shuffle(y1, random_state=2)
>>> Y = np.vstack((y1, y2, y3)).T
>>> n_samples, n_features = X.shape # 10,100
>>> n_outputs = Y.shape[1] # 3
>>> n_classes = 3
>>> forest = RandomForestClassifier(n_estimators=100, random_state=1)
>>> multi_target_forest = MultiOutputClassifier(forest, n_jobs=-1)
>>> multi_target_forest.fit(X, Y).predict(X)
array([[2, 2, 0],
       [1, 2, 1],
       [2, 1, 0],
       [0, 0, 2],
       [0, 2, 1],
       [0, 0, 2],
       [1, 1, 0],
       [1, 1, 1],
       [0, 0, 2],
       [2, 0, 0]])
```

1.12.7. 链式分类器

Classifier chains（查看 `ClassifierChain` ）是一种集合多个二分类器为一个单独的多标签模型的方法，这种方法能够发掘目标之间的相关性信息。

对于有 N 个类的多标签分类问题，为 N 个二元分类器分配 0 到 N-1 之间的一个整数。这些整数定义了模型在 chain 中的顺序。将每个分类器拟合可用的训练数据与真实的类别标签，标签数字相对较小。

当进行预测时，真正的标签将无法使用。相反，每一个模型的预测结果将会传递给链上的下一个模型作为特征来进行使用。

很明显，链的顺序是十分重要的。链上的第一个模型没有关于其他标签的信息，而链上的最后一个模型将会具有所有其他标签的信息。 在一般情况下，我们并不知道链上模型最优的顺序，因此通常会使用许多随机的顺序，将他们的预测求平均。

> 参考资料：
>
> - Jesse Read, Bernhard Pfahringer, Geoff Holmes, Eibe Frank,“Classifier Chains for Multi-label Classification”, 2009.

我们一直在努力

apachecn/sklearn-doc-zh


加入QQ群

1.13. 特征选择

校验者：@yuezhao9210@BWM-蜜蜂@Loopy翻译者：@v

在 `sklearn.feature_selection` 模块中的类可以用来对样本集进行 feature selection（特征选择）和 dimensionality reduction（降维），这将会提高估计器的准确度或者增强它们在高维数据集上的性能。

1.13.1. 移除低方差特征

`VarianceThreshold` 是特征选择的一个简单基本方法，它会移除所有那些方差不满足一些阈值的特征。默认情况下，它将会移除所有的零方差特征，即那些在所有的样本上的取值均不变的特征。

例如，假设我们有一个特征是布尔值的数据集，我们想要移除那些在整个数据集中特征值为0或者为1的比例超过80%的特征。布尔特征是伯努利（ Bernoulli ）随机变量，变量的方差为

$$\mathrm{Var}[X] = p(1-p)$$

因此，我们可以使用阈值 ``.8 * (1 - .8)``进行选择：

```
>>> from sklearn.feature_selection import VarianceThreshold
>>> X = [[0, 0, 1], [0, 1, 0], [1, 0, 0], [0, 1, 1], [0, 1, 0], [0, 1, 1]]
>>> sel = VarianceThreshold(threshold=(.8 * (1 - .8)))
>>> sel.fit_transform(X)
array([[0, 1],
       [1, 0],
       [0, 0],
       [1, 1],
       [1, 0],
       [1, 1]])
```

正如预期一样，`VarianceThreshold` 移除了第一列，它的值为 0 的概率为 $p = 5/6 > .8$ 。

1.13.2. 单变量特征选择

单变量的特征选择是通过基于单变量的统计测试来选择最好的特征。它可以当做是评估器的预处理步骤。Scikit-learn 将特征选择的内容作为实现了 transform 方法的对象：

- `SelectKBest` 移除那些除了评分最高的 K 个特征之外的所有特征
- `SelectPercentile` 移除除了用户指定的最高得分百分比之外的所有特征
- 对每个特征应用常见的单变量统计测试：假阳性率（false positive rate） `SelectFpr`，伪发现率（false discovery rate） `SelectFdr`，或者族系误差（family wise error） `SelectFwe`。
- `GenericUnivariateSelect` 允许使用可配置方法来进行单变量特征选择。它允许超参数搜索评估器来选择最好的单变量特征。

例如下面的实例，我们可以使用 χ^2 检验样本集来选择最好的两个特征：

```
>>> from sklearn.datasets import load_iris
>>> from sklearn.feature_selection import SelectKBest
>>> from sklearn.feature_selection import chi2
>>> iris = load_iris()
>>> X, y = iris.data, iris.target
>>> X.shape
(150, 4)
>>> X_new = SelectKBest(chi2, k=2).fit_transform(X, y)
>>> X_new.shape
(150, 2)
```

这些对象将得分函数作为输入，返回单变量的得分和 p 值 （或者仅仅是 `SelectKBest` 和 `SelectPercentile` 的分数）：

- 对于回归： `f_regression` , `mutual_info_regression`
- 对于分类： `chi2` , `f_classif` , `mutual_info_classif`

这些基于 F-test 的方法计算两个随机变量之间的线性相关程度。另一方面，mutual information methods（互信息）能够计算任何种类的统计相关性，但是作为非参数的方法，互信息需要更多的样本来进行准确的估计。

> 稀疏数据的特征选择
>
> 如果你使用的是稀疏的数据（例如数据可以由稀疏矩阵来表示）， `chi2` , `mutual_info_regression` , `mutual_info_classif` 可以处理数据并保持它的稀疏性。
>
> 警告
>
> 不要使用一个回归评分函数来处理分类问题，你会得到无用的结果。
>
> 示例：
>
> - Univariate Feature Selection
> - Comparison of F-test and mutual information

1.13.3. 递归式特征消除

给定一个外部的估计器，可以对特征赋予一定的权重（比如，线性模型的相关系数），recursive feature elimination （ `RFE` ）通过考虑越来越小的特征集合来递归的选择特征。 首先，评估器在初始的特征集合上面训练并且每一个特征的重要程度是通过一个 `coef` 属性 或者 *`feature_importances`* 属性来获得。 然后，从当前的特征集合中移除最不重要的特征。在特征集合上不断的重复递归这个步骤，直到最终达到所需要的特征数量为止。 `RFECV` 在一个交叉验证的循环中执行 RFE 来找到最优的特征数量

> 示例：
>
> - Recursive feature elimination ： 通过递归式特征消除来体现数字分类任务中像素重要性的例子。
> - Recursive feature elimination with cross-validation ： 通过递归式特征消除来自动调整交叉验证中选择的特征数。

1.13.4. 使用 SelectFromModel 选取特征

`SelectFromModel` 是一个 meta-transformer（元转换器），它可以用来处理任何带有 `coef` 或者 *`feature_importances`* 属性的训练之后的评估器。 如果相关的 `coef_` 或者 `featureimportances` 属性值低于预先设置的阈值，这些特征将会被认为不重要并且移除掉。除了指定数值上的阈值之外，还可以通过给定字符串参数来使用内置的启发式方法找到一个合适的阈值。可以使用的启发式方法有 mean 、 median 以及使用浮点数乘以这些（例如，0.1*mean ）。

有关如何使用的例子，可以参阅下面的例子。

> 示例：
>
> - Feature selection using SelectFromModel and LassoCV: 从 Boston 数据中自动选择最重要两个特征而不需要提前得知这一信息。

1.13.4.1. 基于 L1 的特征选取

Linear models 使用 L1 正则化的线性模型会得到稀疏解：他们的许多系数为 0。 当目标是降低使用另一个分类器的数据集的维度， 它们可以与 `feature_selection.SelectFromModel` 一起使用来选择非零系数。特别的，可以用于此目的的稀疏评估器有用于回归的 `linear_model.Lasso` ，以及用于分类的 `linear_model.LogisticRegression` 和 `svm.LinearSVC`

```
>>> from sklearn.svm import LinearSVC
>>> from sklearn.datasets import load_iris
>>> from sklearn.feature_selection import SelectFromModel
>>> iris = load_iris()
>>> X, y = iris.data, iris.target
>>> X.shape
(150, 4)
>>> lsvc = LinearSVC(C=0.01, penalty="l1", dual=False).fit(X, y)
>>> model = SelectFromModel(lsvc, prefit=True)
>>> X_new = model.transform(X)
>>> X_new.shape
(150, 3)
```

在 SVM 和逻辑回归中，参数 C 是用来控制稀疏性的：小的 C 会导致少的特征被选择。使用 Lasso，alpha 的值越大，越少的特征会被选择。

> 示例：
>
> - Classification of text documents using sparse features: 不同算法的比较，当使用 L1 正则化的特征选择在文件分类任务上。
>
> **L1-recovery 和 compressive sensing**（压缩感知）
>
> 当选择了正确的 alpha 值以后， Lasso 可以仅通过少量观察点便恢复完整的非零特征， 假设特定的条件可以被满足的话。特别的，数据量需要 “足够大” ，不然 L1 模型的表现将缺乏保障。 “足够大” 的定义取决于非零系数的个数、特征数量的对数值、噪音的数量、非零系数的最小绝对值、 以及设计矩阵（design maxtrix） X 的结构。特征矩阵必须有特定的性质，如数据不能过度相关。
>
> 关于如何选择 alpha 值没有固定的规则。alpha 值可以通过交叉验证来确定（ `LassoCV` 或者 `LassoLarsCV` ），尽管这可能会导致欠惩罚的模型：包括少量的无关变量对于预测值来说并非致命的。相反的， BIC（ `LassoLarsIC` ）倾向于给定高 alpha 值。
>
> **Reference**（参考文献） Richard G. Baraniuk “Compressive Sensing”, IEEE Signal Processing Magazine [120] July 2007
> http://dsp.rice.edu/sites/dsp.rice.edu/files/cs/baraniukCSlecture07.pdf

1.13.4.2. 基于 Tree（树）的特征选取

基于树的 estimators （查阅 `sklearn.tree` 模块和树的森林 在 `sklearn.ensemble` 模块） 可以用来计算特征的重要性，然后可以消除不相关的特征（当与 `sklearn.feature_selection.SelectFromModel` 等元转换器一同使用时）：

```
>>> from sklearn.ensemble import ExtraTreesClassifier
>>> from sklearn.datasets import load_iris
>>> from sklearn.feature_selection import SelectFromModel
>>> iris = load_iris()
>>> X, y = iris.data, iris.target
>>> X.shape
(150, 4)
>>> clf = ExtraTreesClassifier()
>>> clf = clf.fit(X, y)
>>> clf.feature_importances_
array([ 0.04...,  0.05...,  0.4...,  0.4...])
>>> model = SelectFromModel(clf, prefit=True)
>>> X_new = model.transform(X)
>>> X_new.shape
(150, 2)
```

> 示例：
>
> - Feature importances with forests of trees: 在合成数据上恢复有用特征的示例。
> - Pixel importances with a parallel forest of trees: 在人脸识别数据上的示例。

1.13.5. 特征选取作为 pipeline（管道）的一部分

特征选择通常在实际的学习之前用来做预处理。在 scikit-learn 中推荐的方式是使用 `sklearn.pipeline.Pipeline`：

```
clf = Pipeline([
  ('feature_selection', SelectFromModel(LinearSVC(penalty="l1"))),
  ('classification', RandomForestClassifier())
])
clf.fit(X, y)
```

在这段代码中，我们利用 `sklearn.svm.LinearSVC` 和 `sklearn.feature_selection.SelectFromModel` 来评估特征的重要性并且选择出相关的特征。 然后，在转化后的输出中使用一个 `sklearn.ensemble.RandomForestClassifier` 分类器，比如只使用相关的特征。你也可以使用其他特征选择的方法和可以提供评估特征重要性的分类器来执行相似的操作。 请查阅 `sklearn.pipeline.Pipeline` 来了解更多的实例。

我们一直在努力

apachecn/sklearn-doc-zh


加入QQ群

1.14. 半监督学习

校验者：@STAN,废柴0.1@Loopy翻译者：@那伊抹微笑

半监督学习 适用于在训练数据上的一些样本数据没有贴上标签的情况。 `sklearn.semi_supervised` 中的半监督估计，能够利用这些附加的未标记数据来更好地捕获底层数据分布的形状，并将其更好地类推到新的样本。 当我们有非常少量的已标签化的点和大量的未标签化的点时，这些算法表现均良好。

> *y* 中含有未标记的数据
>
> 在使用 `fit` 方法训练数据时，将标识符与已标签化的数据一起分配给未标签化的点是尤其重要的. 实现该标记的方法是使用整数值 -1.

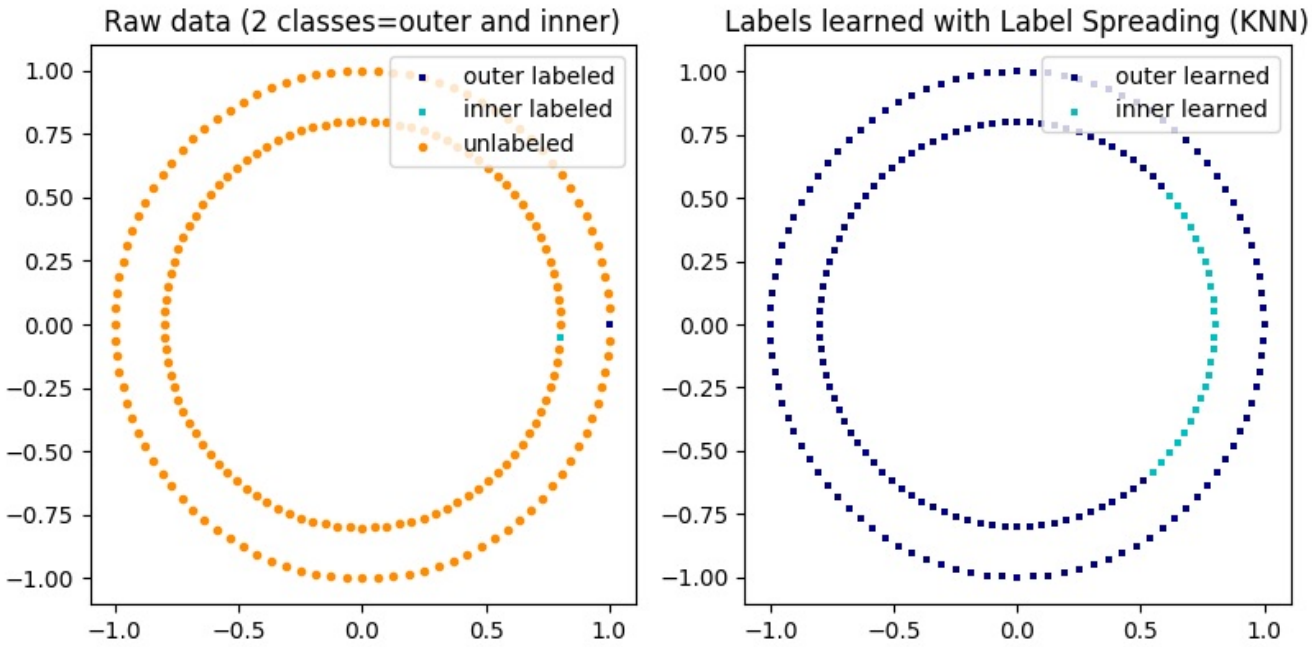
1.14.1. 标签传播

标签传播表示半监督图推理算法的几个变体。

该模型的一些特性如下*:*

- 可用于分类和回归任务
- 使用内核方法将数据投影到备用维度空间

scikit-learn 提供了两种标签传播模型： `LabelPropagation` 和 `LabelSpreading` 。 两者都通过在输入的 dataset（数据集）中的所有 items（项）上构建 similarity graph （相似图）来进行工作。



标签传播说明:未标签化的观察值结构与 *class*（类）结构一致，因此可以将 *class*（类）标签传播到训练集的未标签化的观察值

`LabelPropagation` 和 `LabelSpreading` 在对图形的相似性矩阵，以及对标签分布的 clamping effect（夹持效应）的修改方面不太一样。 Clamping 允许算法在一定程度上改变真实标签化数据的权重。 该

`LabelPropagation` 算法执行输入标签的 hard clamping，这意味着 $\alpha = 0$ 。 这些 clamping factor 可以不是很严格的，例如 $\alpha = 0.2$ ，这意味着我们将始终保留原始标签分配的 80%，但该算法可以将其分布的置信度改变在 20% 以内。

`LabelPropagation` 使用原始相似性矩阵从未修改的数据来构建。 `LabelSpreading` 最小化具有正则化属性的损耗函数，因此它通常更适用于噪声数据。 该算法在原始图形的修改版本上进行迭代，并通过计算 normalized graph Laplacian matrix （归一化图拉普拉斯矩阵）来对边缘的权重进行归一化。 此过程也用于 Spectral clustering 。

标签传播模型有两种内置的 kernel methods（核函数）。 kernel （核）的选择会影响算法的可扩展性和性能。以下是可用的：

- rbf ($\exp(-\gamma|x-y|^2), \gamma > 0$). γ 通过关键字 gamma 来指定。
- knn ($1[x' \in kNN(x)]$). k 通过关键字 n_neighbors 来指定。

RBF 核将产生一个完全连接的图形，它通过密集矩阵在内存中表示。 该矩阵可能非常大，与算法的每次迭代执行全矩阵乘法计算的成本相结合可导致超长的运行时间。 在另一方面，KNN 核将产生更多的内存友好的稀疏矩阵，这样可以大幅度的减少运行时间。

> 示例
>
> - Decision boundary of label propagation versus SVM on the Iris dataset
> - Label Propagation learning a complex structure
> - Label Propagation digits active learning
>
> 参考资料
>
> - [1] Yoshua Bengio, Olivier Delalleau, Nicolas Le Roux. In Semi-Supervised Learning (2006), pp. 193-216
> - [2] Olivier Delalleau, Yoshua Bengio, Nicolas Le Roux. Efficient Non-Parametric Function Induction in Semi-Supervised Learning. AISTAT 2005 http://research.microsoft.com/en-us/people/nicolasl/efficient_ssl.pdf

---

我们一直在努力

apachecn/sklearn-doc-zh



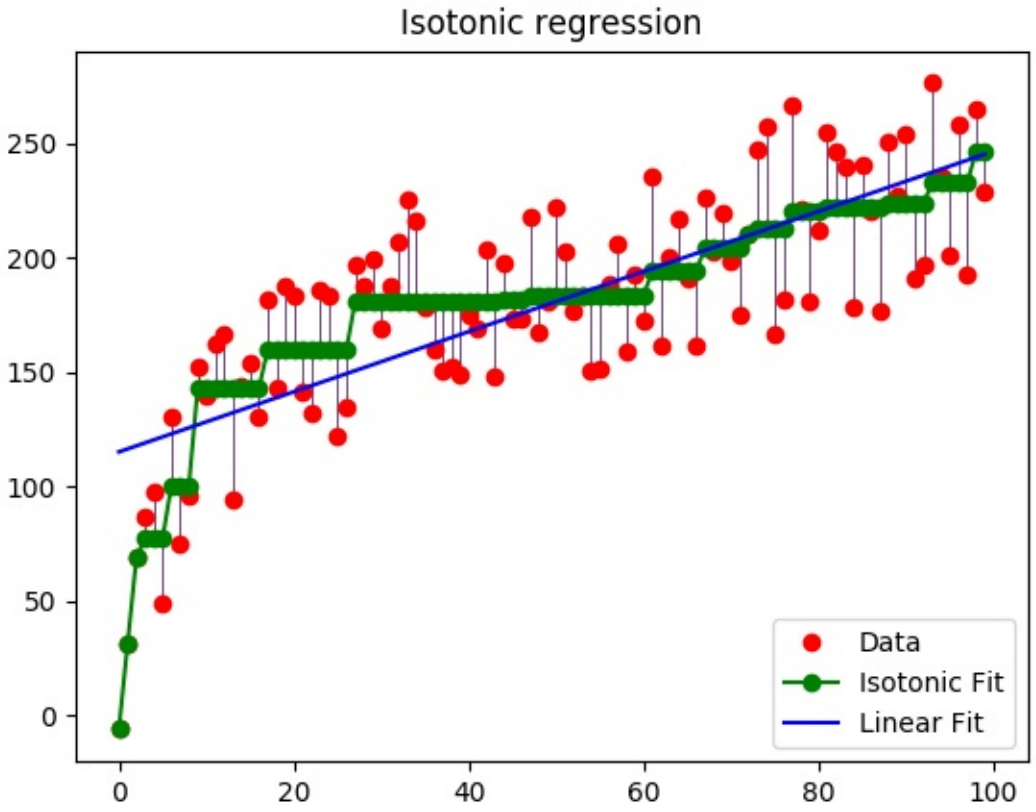
1.15. 等式回归

校验者：@STAN,废柴0.1翻译者：@Damon

`IsotonicRegression` 类对数据进行非降函数拟合. 它解决了如下的问题:

- 最小化 $\sum_i w_i(y_i - \hat{y}_i)^2$
- 服从于 $\hat{y}_{min} = \hat{y}_1 \leq \hat{y}_2 \cdots \leq \hat{y}_n = \hat{y}_{max}$

其中每一个 w_i 是 strictly 正数而且每个 y_i 是任意实 数. 它生成一个由平方误差接近的不减元素组成的向量.实际上这一些元素形成 一个分段线性的函数.



我们一直在努力

apachecn/sklearn-doc-zh

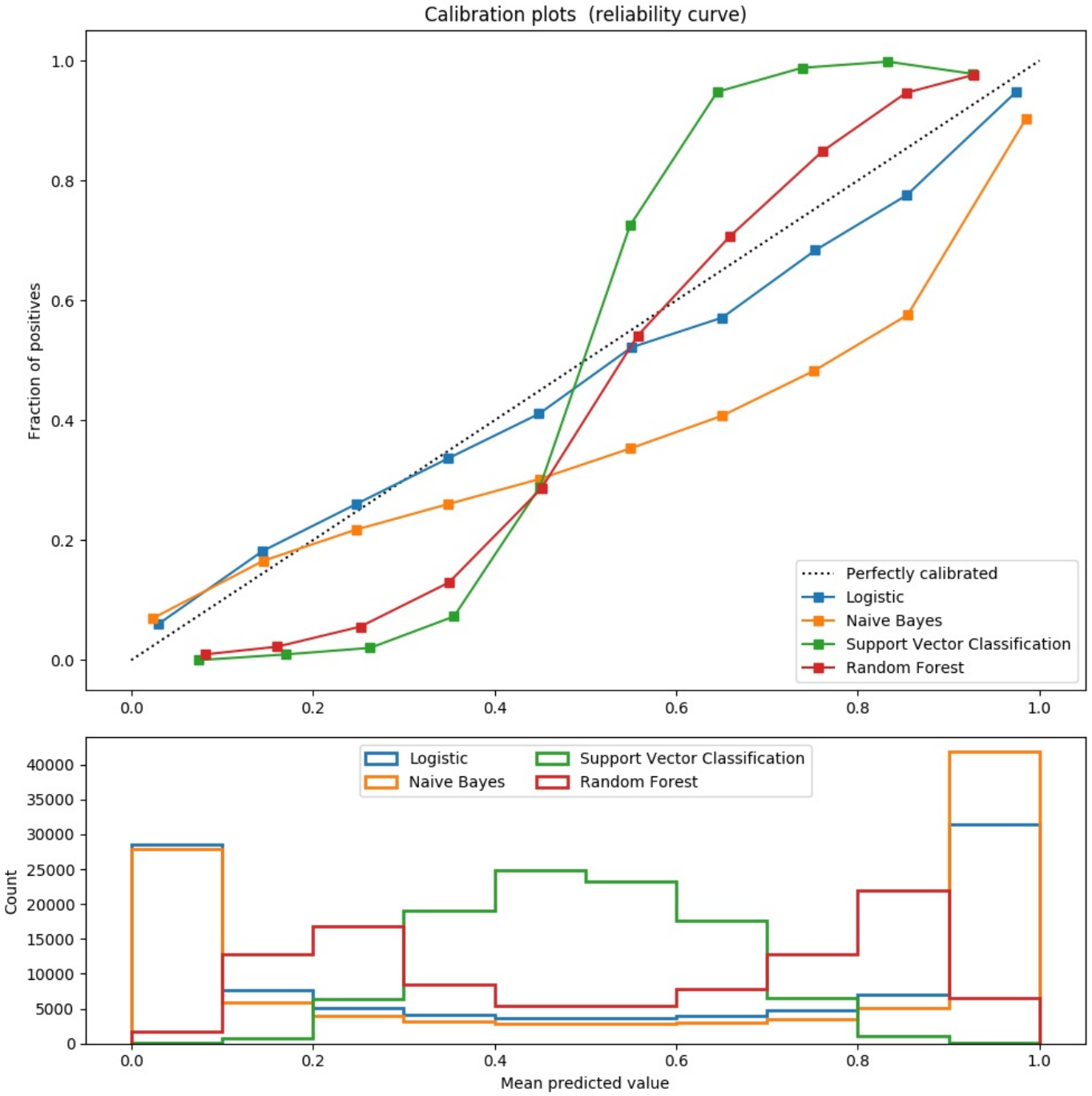
加入QQ群

1.16. 概率校准

校验者：@曲晓峰@小瑶翻译者：@那伊抹微笑

执行分类时，您经常希望不仅可以预测类标签，还要获得相应标签的概率。这个概率给你一些预测的信心。一些模型可以给你贫乏的概率估计，有些甚至不支持概率预测。校准模块可以让您更好地校准给定模型的概率，或添加对概率预测的支持。

精确校准的分类器是概率分类器，其可以将 `predict_proba` 方法的输出直接解释为 `confidence level`（置信度级别）。例如，一个经过良好校准的（二元的）分类器应该对样本进行分类，使得在给出一个接近 `0.8` 的 `prediction_proba` 值的样本中，大约 `80%` 实际上属于正类。以下图表比较了校准不同分类器的概率预测的良好程度：



`LogisticRegression` 默认情况下返回良好的校准预测，因为它直接优化了 log-loss（对数损失）情况. 相反，其他方法返回 biased probabilities（偏倚概率）；每种方法有不同的偏差:

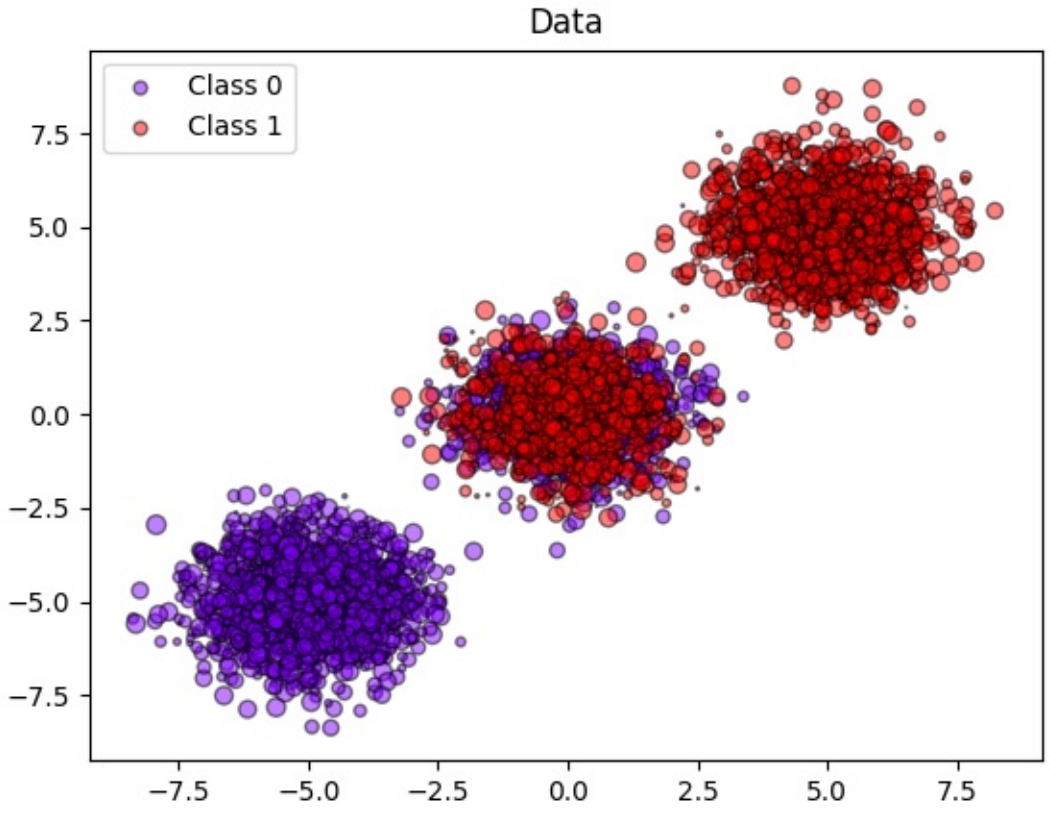
- `GaussianNB` 往往将概率推到 0 或 1（注意直方图中的计数）. 这主要是因为它假设特征在给定类别的条件下是独立的，在该数据集中不包含 2 个冗余特征.

- `RandomForestClassifier` 解释了相反的行为: 直方图在约 0.2 和 0.9 的概率时显示峰值，而接近 0 或 1 的概率非常罕见. Niculescu-Mizil 和 Caruana [4] 给出了一个解释: ”诸如 bagging 和 random forests（随机森林）的方法， 从基本模型的平均预测中可能难以将预测置于 0 和 1 附近，因为基础模型的变化会偏离预测值，它们应该接近于零或偏离这些值，因为预测被限制在 [0,1] 的间隔，由方差引起的误差往往是靠近 0 和 1 的一边， 例如，如果一个模型应该对于一个案例，预测 p = 0，bagging 可以实现的唯一方法是假设所有的 bagging 树预测为零. 如果我们在 bagging 树上增加噪声，这种噪声将导致一些树预测大于 0 的值，因此将 bagging 的平均预测从 0 移开. 我们用随机森林最强烈地观察到这种效应，因为用随机森林训练的 base-level 树由于特征划分而具有相对较高的方差. 因此，校准曲线也被称为可靠性图 (Wilks 1995 [5]) 显示了一种典型的 sigmoid 形状，表明分类器可以更多地信任其 “直觉”，并通常将接近 0 或 1 的概率返回.

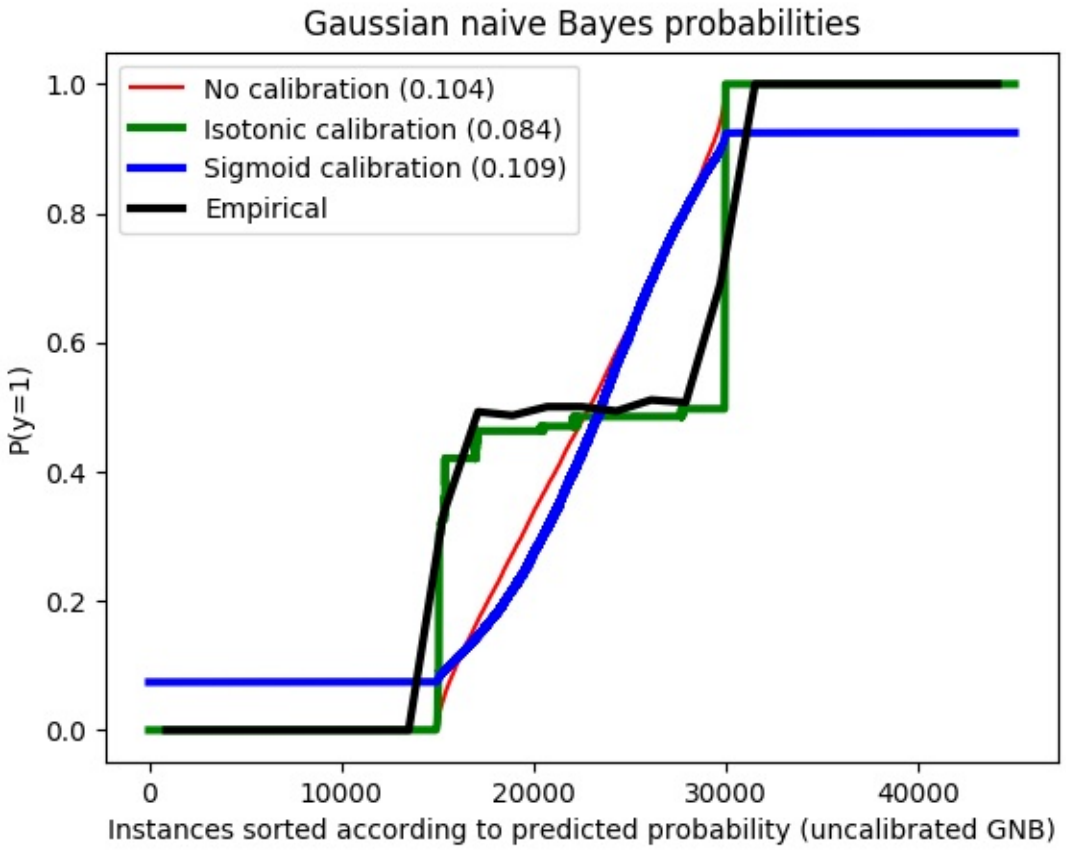
- 线性支持向量分类 （ `LinearSVC` ） 显示了作为 RandomForestClassifier 更多的 Sigmoid 曲线，这是经典的最大边距方法 (compare Niculescu-Mizil and Caruana [4]), 其重点是靠近决策边界的 hard samples（支持向量）.

提供了执行概率预测校准的两种方法: 基于 Platt 的 Sigmoid 模型的参数化方法和基于 isotonic regression（保序回归）的非参数方法 （ `sklearn.isotonic` ）. 对于不用于模型拟合的新数据，应进行概率校准. 类 `CalibratedClassifierCV` 使用交叉验证生成器，并对每个拆分模型参数对训练样本和测试样本的校准进行估计. 然后对折叠预测的概率进行平均. 已经安装的分类器可以通过:class:*CalibratedClassifierCV* 传递参数 cv =”prefit” 这种方式进行校准. 在这种情况下，用户必须手动注意模型拟合和校准的数据是不相交的.

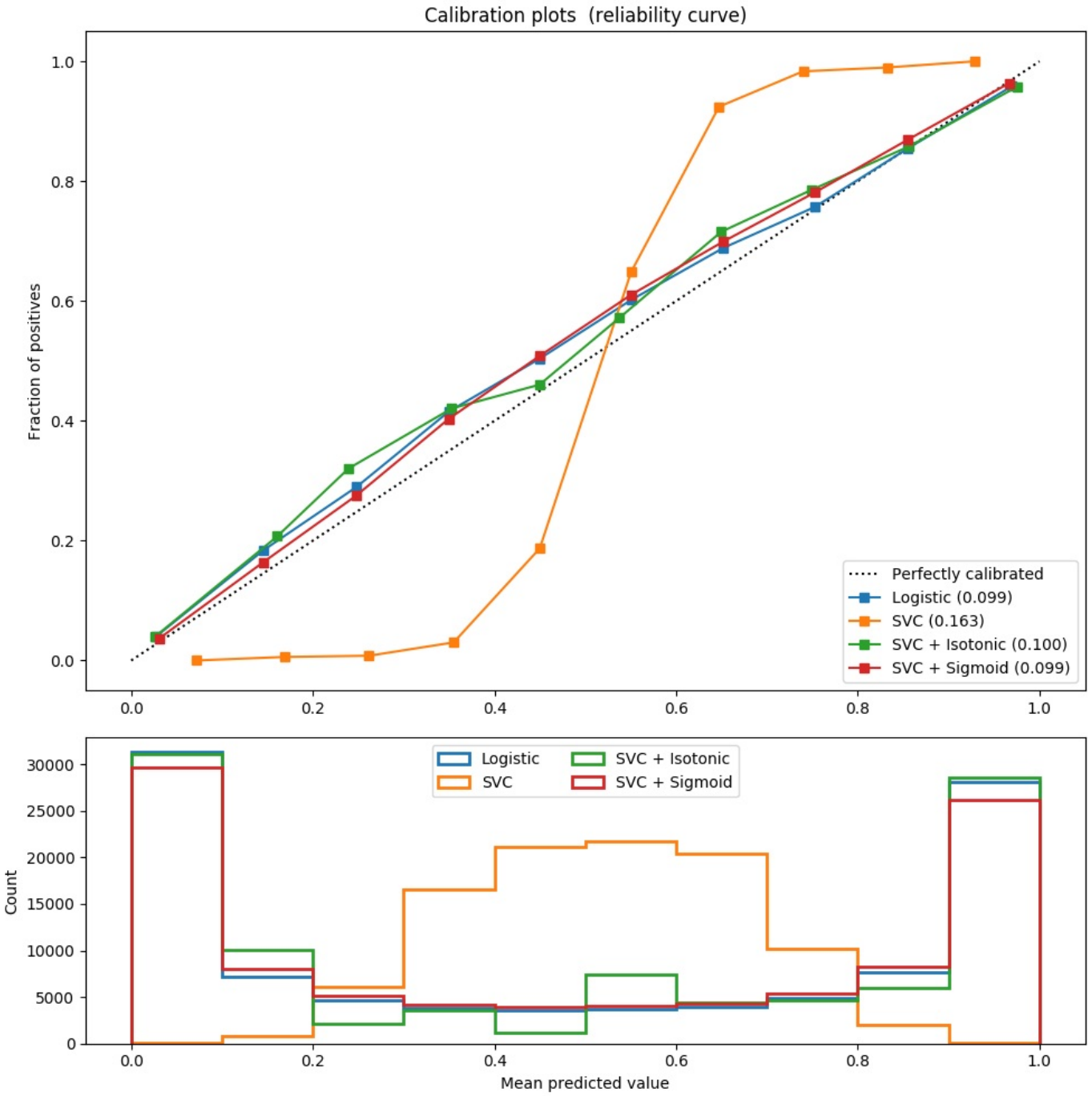
以下图像展示了概率校准的好处. 第一个图像显示一个具有 2 个类和 3 个数据块的数据集. 中间的数据块包含每个类的随机样本. 此数据块中样本的概率应为 0.5.



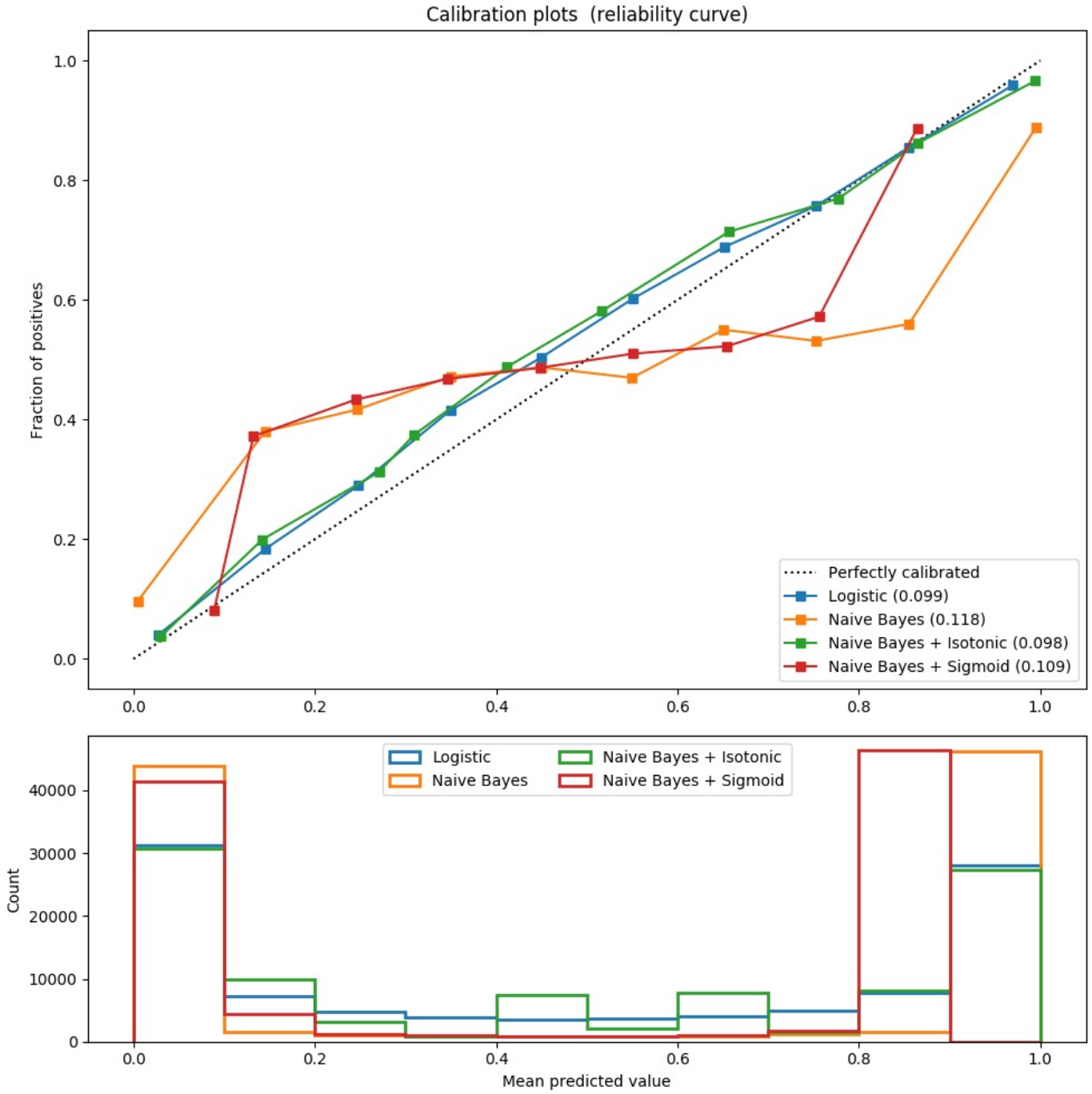
以下图像使用没有校准的高斯朴素贝叶斯分类器，使用 `sigmoid` 校准和非参数的等渗校准来显示上述估计概率的数据. 可以观察到，非参数模型为中间样本提供最准确的概率估计，即`0.5`.



对具有20个特征的100.000个样本（其中一个用于模型拟合）进行二元分类的人造数据集进行以下实验. 在 20个特征中，只有 2 个是信息量，10 个是冗余的. 该图显示了使用逻辑回归获得的估计概率，线性支持向量分类器（SVC）和具有 sigmoid 校准和 sigmoid 校准的线性 SVC. 校准性能使用 Brier score `brier_score_loss` 来计算，请看下面的图例（越销越好）.



这里可以观察到，逻辑回归被很好地校准，因为其曲线几乎是对角线. 线性 SVC 的校准曲线或可靠性图具有 sigmoid 曲线，这是一个典型的不够自信的分类器. 在 LinearSVC 的情况下，这是 hinge loss 的边缘属性引起的，这使得模型集中在靠近决策边界（支持向量）的 hard samples（硬样本）上. 这两种校准都可以解决这个问题，并产生几乎相同的结果. 下图显示了高斯朴素贝叶斯在相同数据上的校准曲线，具有两种校准，也没有校准.

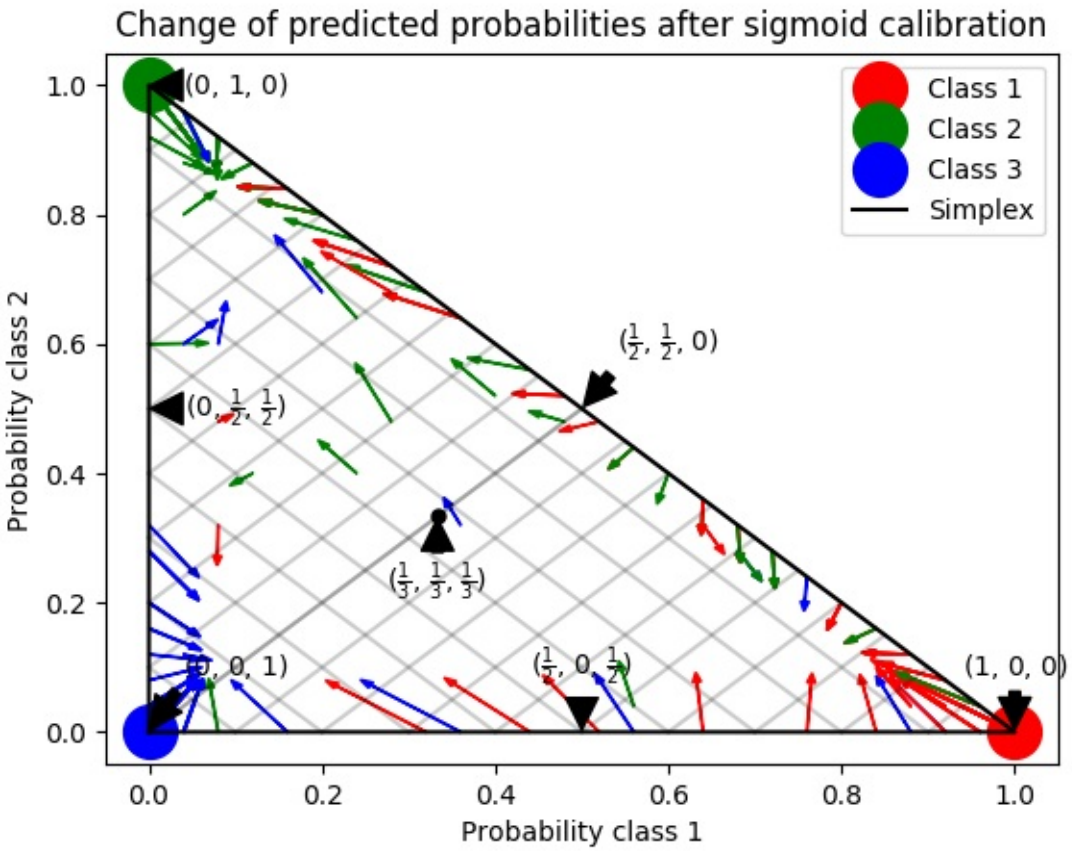


可以看出， 高斯朴素贝叶斯的表现非常差，但是以线性 SVC 的方式也是如此．尽管线性 SVC 显示了 sigmoid 校准曲线，但高斯朴素贝叶斯校准曲线具有转置的 sigmoid 结构．这对于过分自信的分类器来说是非常经典的．在这种情况下，分类器的过度自信是由违反朴素贝叶斯特征独立假设的冗余特征引起的．

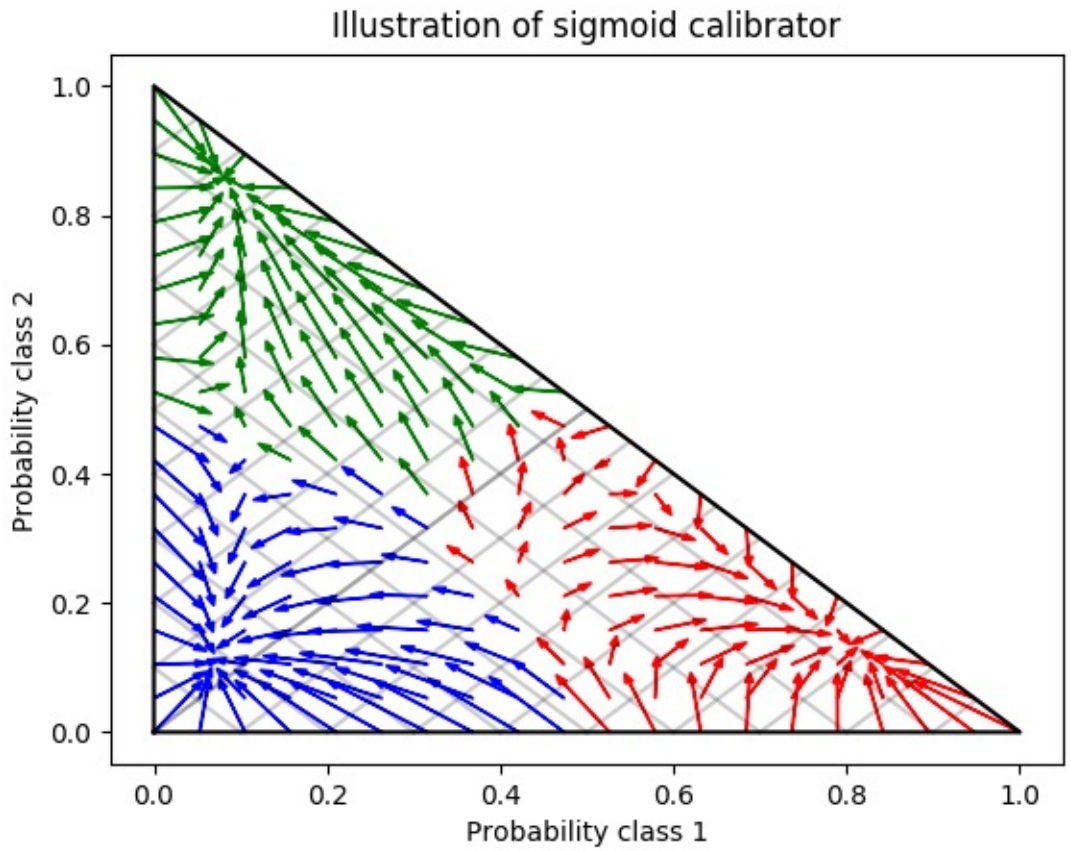
用等渗回归法对高斯朴素贝叶斯概率的校准可以解决这个问题，从几乎对角线校准曲线可以看出．Sigmoid 校准也略微改善了 brier 评分，尽管不如非参数等渗校准那样强烈．这是 sigmoid 校准的固有限制，其参数形式假定为 sigmoid ，而不是转置的 sigmoid 曲线．然而，非参数等渗校准模型没有这样强大的假设，并且可以处理任何形状，只要有足够的校准数据．通常，在校准曲线为 sigmoid 且校准数据有限的情况下，sigmoid 校准是优选的，而对于非 sigmoid 校准曲线和大量数据可用于校准的情况，等渗校准是优选的．

`CalibratedClassifierCV` 也可以处理涉及两个以上类的分类任务，如果基本估计器可以这样做的话．在这种情况下，分类器是以一对一的方式分别对每个类进行校准．当预测未知数据的概率时，分别预测每个类的校准概率．由于这些概率并不总是一致，因此执行后处理以使它们归一化．

下一个图像说明了 Sigmoid 校准如何改变 3 类分类问题的预测概率. 说明是标准的 2-simplex，其中三个角对应于三个类. 箭头从未校准分类器预测的概率向量指向在保持验证集上的 sigmoid 校准之后由同一分类器预测的概率向量. 颜色表示实例的真实类（red: class 1, green: class 2, blue: class 3）.



基础分类器是具有 25 个基本估计器（树）的随机森林分类器. 如果这个分类器对所有 800 个训练数据点进行了训练，那么它的预测过于自信，从而导致了大量的对数损失. 校准在 600 个数据点上训练的相同分类器， 其余 200 个数据点上的 method =’sigmoid’ 减少了预测的置信度， 即将概率向量从单面的边缘向中心移动:



该校准导致较低的 log-loss（对数损失）. 请注意，替代方案是增加基准估计量的数量，这将导致对数损失类似的减少.

> 参考资料：
>
> - Obtaining calibrated probability estimates from decision trees and naive Bayesian classifiers, B. Zadrozny & C. Elkan, ICML 2001
> - Transforming Classifier Scores into Accurate Multiclass Probability Estimates, B. Zadrozny & C. Elkan, (KDD 2002)
> - Probabilistic Outputs for Support Vector Machines and Comparisons to Regularized Likelihood Methods, J. Platt, (1999)
> - [4] Predicting Good Probabilities with Supervised Learning, A. Niculescu-Mizil & R. Caruana, ICML 2005
> - [5] On the combination of forecast probabilities for consecutive precipitation periods. Wea. Forecasting, 5, 640–650., Wilks, D. S., 1990a

---

我们一直在努力

apachecn/sklearn-doc-zh



1.17. 神经网络模型（有监督）

校验者：@tiantian1412@火星@Loopy翻译者：@A

> 警告
>
> 此实现不适用于大规模数据应用。 特别是 scikit-learn 不支持 GPU。如果想要提高运行速度并使用基于 GPU 的实现以及为构建深度学习架构提供更多灵活性的框架，请参阅 Related Projects 。

1.17.1. 多层感知器

多层感知器(MLP) 是一种监督学习算法，通过在数据集上训练来学习函数 $f(\cdot) : R^m \rightarrow R^o$ ，其中 m 是输入的维数，o 是输出的维数。 给定一组特征 $X = x_1, x_2, ..., x_m$ 和标签 y ，它可以学习用于分类或回归的非线性函数。 与逻辑回归不同的是，在输入层和输出层之间，可以有一个或多个非线性层，称为隐藏层。 图1 展示了一个具有标量输出的单隐藏层 MLP。

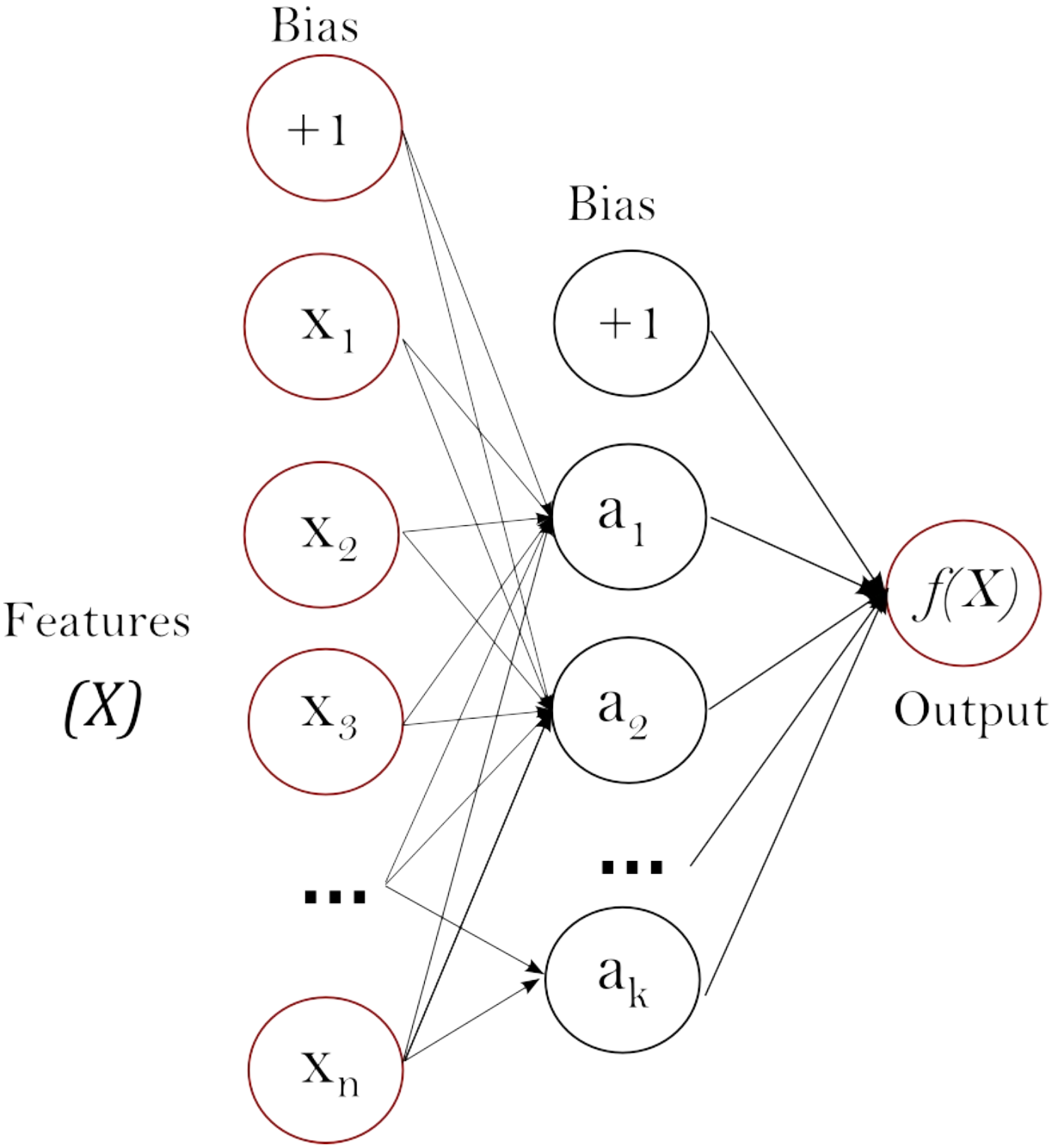


图1：单隐藏层MLP.

最左层的输入层由一组代表输入特征的神经元 $\{x_i|x_1, x_2, ..., x_m\}$ 组成。 每个隐藏层中的神经元将前一层的值进行加权线性求和转换 $w_1x_1 + w_2x_2 + ... + w_mx_m$ ，再通过非线性激活函数 $g(\cdot) : R \rightarrow R$ - 比如双曲正切函数 `tanh` 。 输出层接收到的值是最后一个隐藏层的输出经过变换而来的。

该模块包含公共属性 `coefs` 和 `intercepts` 。 `coefs` 是一系列权重矩阵，其中下标为 i 的权重矩阵表示第 i 层和第 $i+1$ 层之间的权重。 `intercepts` 是一系列偏置向量，其中的下标为 i 的向量表示添加到第

$i+1$ 层的偏置值。

多层感知器的优点：

- 可以学习得到非线性模型。
- 使用 `partial_fit` 可以学习得到实时模型(在线学习)。

多层感知器(MLP)的缺点：

- 具有隐藏层的 MLP 具有非凸的损失函数，它有不止一个的局部最小值。 因此不同的随机权重初始化会导致不同的验证集准确率。
- MLP 需要调试一些超参数，例如隐藏层神经元的数量、层数和迭代轮数。
- MLP 对特征归一化很敏感.

解决这些缺点的方法请参阅 实用使用技巧 部分。

1.17.2. 分类

`MLPClassifier` 类实现了通过 Backpropagation 进行训练的多层感知器（MLP）算法。

MLP 在两个 array 上进行训练:大小为 (n_samples, n_features) 的 array X 储存表示训练样本的浮点型特征向量; 大小为 (n_samples,) 的 array y 储存训练样本的目标值（类别标签）：

```
>>> from sklearn.neural_network import MLPClassifier
>>> X = [[0., 0.], [1., 1.]]
>>> y = [0, 1]
>>> clf = MLPClassifier(solver='lbfgs', alpha=1e-5,
...                     hidden_layer_sizes=(5, 2), random_state=1)
...
>>> clf.fit(X, y)
MLPClassifier(activation='relu', alpha=1e-05, batch_size='auto',
              beta_1=0.9, beta_2=0.999, early_stopping=False,
              epsilon=1e-08, hidden_layer_sizes=(5, 2),
              learning_rate='constant', learning_rate_init=0.001,
              max_iter=200, momentum=0.9, n_iter_no_change=10,
              nesterovs_momentum=True, power_t=0.5, random_state=1,
              shuffle=True, solver='lbfgs', tol=0.0001,
              validation_fraction=0.1, verbose=False, warm_start=False)
```

拟合（训练）后，该模型可以预测新样本的标签：

```
>>> clf.predict([[2., 2.], [-1., -2.]])
array([1, 0])
```

MLP 可以为训练数据拟合一个非线性模型。 `clf.coefs_` 包含了构建模型的权值矩阵：

```
>>> [coef.shape for coef in clf.coefs_]
[(2, 5), (5, 2), (2, 1)]
```

目前， `MLPClassifier` 只支持交叉熵损失函数，通过运行 `predict_proba` 方法进行概率估计。

MLP 算法使用的是反向传播的方式。 更准确地说，它使用了通过反向传播计算得到的梯度和某种形式的梯度下降来进行训练。 对于分类来说，它最小化交叉熵损失函数，为每个样本 x 给出一个向量形式的概率估计 $P(y|x)$

```
>>> clf.predict_proba([[2., 2.], [1., 2.]])
array([[  1.967...e-04,   9.998...-01],
 [  1.967...e-04,   9.998...-01]])
```

`MLPClassifier` 通过应用 Softmax 作为输出函数来支持多分类。

此外，该模型支持 多标签分类 ，一个样本可能属于多个类别。 对于每个类，原始输出经过 logistic 函数变换后，大于或等于 0.5 的值将进为 1，否则为 0。 对于样本的预测输出，值为 1 的索引位置表示该样本的分类类别：

```
>>> X = [[0., 0.], [1., 1.]]
>>> y = [[0, 1], [1, 1]]
>>> clf = MLPClassifier(solver='lbfgs', alpha=1e-5,
...                     hidden_layer_sizes=(15,), random_state=1)
...
>>> clf.fit(X, y)
MLPClassifier(activation='relu', alpha=1e-05, batch_size='auto',
              beta_1=0.9, beta_2=0.999, early_stopping=False,
              epsilon=1e-08, hidden_layer_sizes=(15,),
              learning_rate='constant', learning_rate_init=0.001,
              max_iter=200, momentum=0.9, n_iter_no_change=10,
              nesterovs_momentum=True, power_t=0.5,  random_state=1,
              shuffle=True, solver='lbfgs', tol=0.0001,
              validation_fraction=0.1, verbose=False, warm_start=False)
>>> clf.predict([[1., 2.]])
array([[1, 1]])
>>> clf.predict([[0., 0.]])
array([[0, 1]])
```

更多内容请参阅下面的示例和文档 `MLPClassifier.fit` 。

示例：

- Compare Stochastic learning strategies for MLPClassifier
- Visualization of MLP weights on MNIST

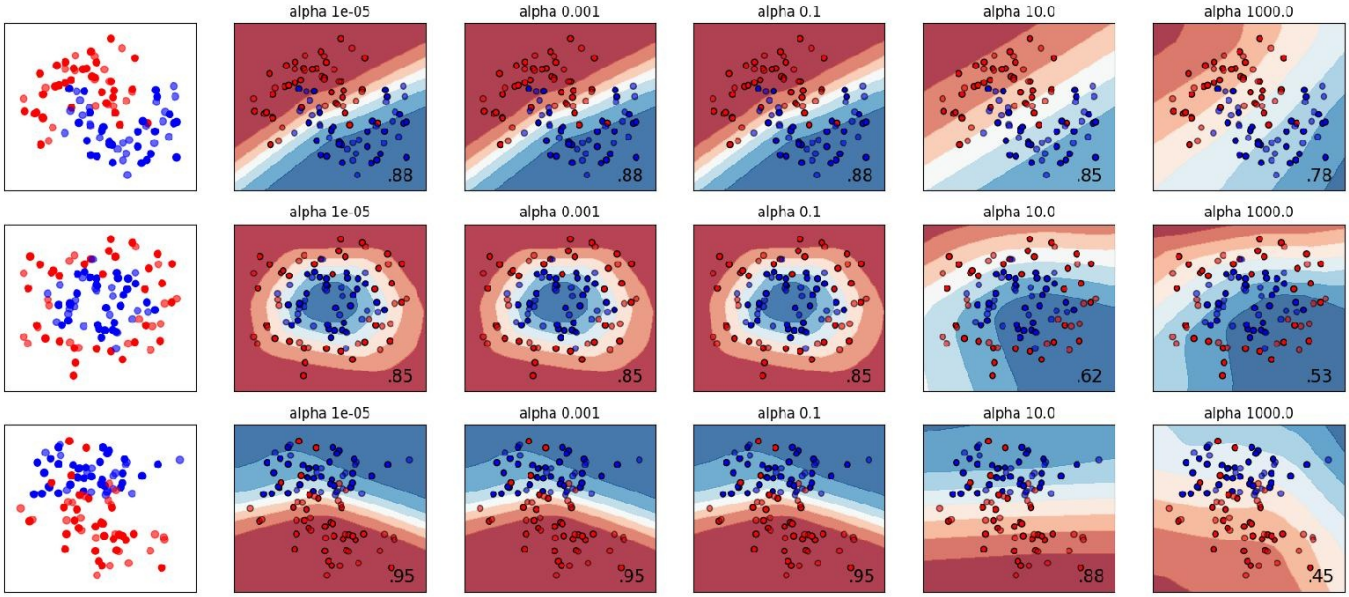
1.17.3. 回归

`MLPRegressor` 类多层感知器（MLP）的实现，在使用反向传播进行训练时的输出层没有使用激活函数，也可以看作是使用恒等函数（identity function）作为激活函数。 因此，它使用平方误差作为损失函数，输出是一组连续值。

`MLPRegressor` 还支持多输出回归，其中一个样本可以有多个目标值。

1.17.4. 正则化

`MLPRegressor` 类和 `MLPClassifier` 类都使用参数 `alpha` 作为正则化(L2 正则化)系数，正则化通过惩罚大数量级的权重值以避免过拟合问题。 下面的图表展示了不同的 alpha 值下的决策函数的变化。



详细信息，请参阅下面的示例。

> 示例：
>
> - Varying regularization in Multi-layer Perceptron

1.17.5. 算法

MLP 使用 Stochastic Gradient Descent（随机梯度下降）(SGD), Adam, 或者 L-BFGS 进行训练。 随机梯度下降（SGD） 使用关于需要适应的一个参数的损失函数的梯度来更新参数，即

$$w \leftarrow w - \eta(\alpha \frac{\partial R(w)}{\partial w} + \frac{\partial Loss}{\partial w})$$

其中 η 是控制训练过程参数更新步长的学习率（learning rate）。 $Loss$ 是损失函数（loss function）。

更多细节可以在这个文档中找到 SGD 。

Adam 类似于 SGD，因为它是 stochastic optimizer （随机优化器），但它可以根据低阶矩的自适应估计自动调整参数更新的量。

使用 SGD 或 Adam ，训练过程支持在线模式和小批量学习模式。

L-BFGS 是利用 Hessian 矩阵来近似函数的二阶偏导数的求解器，它使用 Hessian 的逆矩阵来近似进行参数更

新。 该实现使用 Scipy 版本的 L-BFGS。

如果所选择的方法是 'L-BFGS'，训练过程不支持在线学习模式和小批量学习模式。

1.17.6. 复杂度

假设有 n 个训练样本， m 个特征， k 个隐藏层，每个包含 h 个神经元 - 为简单起见， o 个输出神经元。

反向传播的时间复杂度是 $O(n \cdot m \cdot h^k \cdot o \cdot i)$ ，其中 i 是迭代次数。 由于反向传播具有高时间复杂性，最好以较少数量的隐藏层神经元和较少的隐藏层个数开始训练。

1.17.7. 数学公式

给出一组训练样本 $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$ 其中 $x_i \in \mathbf{R}^n$ ， $y_i \in \{0, 1\}$ ，一个单隐藏层单神经元 MLP 学习到的函数是 $f(x) = W_2 g(W_1^T x + b_1) + b_2$ ，其中 $W_1 \in \mathbf{R}^m$ 和 $W_2, b_1, b_2 \in \mathbf{R}$ 是模型参数. W_1, W_2 分别是输入层与隐藏层之间和隐藏层与输出层之间的权重， b_1, b_2 分别是隐藏层和输出层的偏置值. $g(\cdot) : R \rightarrow R$ 是激活函数，默认为双曲正切函数。 具体形式如下，

$$g(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

对于二分类， $f(x)$ 经过 logistic 函数 $g(z) = 1/(1 + e^{-z})$ 得到 0 到 1 之间的输出值。 阈值设置为 0.5 ，输出大于等于 0.5 的样本分到 positive class （正类），其他的分为 negative class （负类）。

如果多于两类，则 $f(x)$ 本身将是一个大小为 (n_classes,) 的向量。 它需要经过 softmax 函数而不是 logistic 函数进行变换，具体形式如下，

$$\text{softmax}(z)_i = \frac{\exp(z_i)}{\sum_{l=1}^{k} \exp(z_l)}$$

其中 z_i 表示 softmax 函数的第 i 个输入的元素，它对应于第 i 类， K 是类别的数量。 计算结果是样本 x 属于每个类别的概率的向量。 最终输出的分类结果是具有最高概率的类别。

在回归问题中，输出依然是 $f(x)$;因此，输出激活函数就是恒等函数。

MLP 根据特定问题使用不同的损失函数。 二分类问题的损失函数的是交叉熵，具体形式如下，

$$Loss(\hat{y}, y, W) = -y \ln \hat{y} - (1 - y) \ln (1 - \hat{y}) + \alpha ||W||_2^2$$

其中 $\alpha||W||_2^2$ 是 L2 正则化的模型复杂度惩罚项；$\alpha > 0$ 这个非负的超参数控制惩罚的程度。

对于回归问题，MLP 使用平方误差损失函数，具体形式如下，

$$Loss(\hat{y}, y, W) = \frac{1}{2}||\hat{y} - y||_2^2 + \frac{\alpha}{2}||W||_2^2$$

从随机初始化权重开始，多层感知器（MLP）不断更新这些权重值来最小化损失函数。计算完损失之后，从输出层到前面各层进行反向传播，更新权重参数的值，旨在减小损失函数。

在梯度下降中，计算得到损失函数关于每个权重的梯度 $\nabla Loss_W$ 并从权重 W 中减掉。用公式表示为，

$$W^{i+1} = W^i - \epsilon \nabla Loss_W^i$$

其中 i 是当前迭代步数，ϵ 是大于 0 学习率。

算法停止的条件或者是达到预设的最大迭代次数，或者是损失函数低于某个特定值。

1.17.8. 实用技巧

- 多层感知器对特征的缩放是敏感的，所以它强烈建议您归一化你的数据。 例如，将输入向量 X 的每个属性放缩到到 [0, 1] 或 [-1，+1] ，或者将其标准化使它具有 0 均值和方差 1。注意，为了得到有意义的结果，您必须对测试集也应用 相同的 尺度缩放。 您可以使用 `StandardScaler` 进行标准化。

```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
# Don't cheat - fit only on training data
scaler.fit(X_train)
X_train = scaler.transform(X_train)
# apply same transformation to test data
X_test = scaler.transform(X_test)
```

另一个推荐的方法是在 `Pipeline` 中使用的 `StandardScaler` 。

- 最好使用 `GridSearchCV` 找到一个合理的正则化参数 α ，通常范围是在 `10.0 ** -np.arange(1, 7)` 。
- 据经验可知，我们观察到 *L-BFGS* 收敛速度是更快的并且是小数据集上更好的解决方案。对于规模相对比较大的数据集，*Adam* 是非常鲁棒的。 它通常会迅速收敛，并得到相当不错的表现。另一方面，如果学习率调整得正确， 使用 momentum 或 nesterov’s momentum 的 *SGD* 可以比这两种算法更好。

1.17.9. 使用 warm_start 的更多控制

如果您希望更多地控制 SGD 中的停止标准或学习率，或者想要进行额外的监视，使用 `warm_start=True` 和 `max_iter=1` 并且自身迭代可能会有所帮助：

```
>>> X = [[0., 0.], [1., 1.]]
```

```
>>> y = [0, 1]
>>> clf = MLPClassifier(hidden_layer_sizes=(15,), random_state=1, max_iter=1, warm_start=True)
>>> for i in range(10):
...     clf.fit(X, y)
...     # additional monitoring / inspection
MLPClassifier(...
```

参考资料：

- “Learning representations by back-propagating errors.” Rumelhart, David E., Geoffrey E. Hinton, and Ronald J. Williams.
- “Stochastic Gradient Descent” L. Bottou - Website, 2010.
- “Backpropagation” Andrew Ng, Jiquan Ngiam, Chuan Yu Foo, Yifan Mai, Caroline Suen - Website, 2011.
- “Efficient BackProp” Y. LeCun, L. Bottou, G. Orr, K. Müller - In Neural Networks: Tricks of the Trade 1998.
- “Adam: A method for stochastic optimization.” Kingma, Diederik, and Jimmy Ba. arXiv preprint arXiv:1412.6980 (2014).

我们一直在努力

apachecn/sklearn-doc-zh



2. 无监督学习

- 2.1. 高斯混合模型
  - 2.1.1. 高斯混合
  - 2.1.2. 变分贝叶斯高斯混合
- 2.2. 流形学习
  - 2.2.1. 介绍
  - 2.2.2. Isomap
  - 2.2.3. 局部线性嵌入
  - 2.2.4. 改进型局部线性嵌入（MLLE）
  - 2.2.5. 黑塞特征映射（HE）
  - 2.2.6. 谱嵌入
  - 2.2.7. 局部切空间对齐（LTSA）
  - 2.2.8. 多维尺度分析（MDS）
  - 2.2.9. t 分布随机邻域嵌入（t-SNE）
  - 2.2.10. 实用技巧
- 2.3. 聚类
  - 2.3.1. 聚类方法概述
  - 2.3.2. K-means
  - 2.3.3. Affinity Propagation
  - 2.3.4. Mean Shift
  - 2.3.5. Spectral clustering
  - 2.3.6. 层次聚类
  - 2.3.7. DBSCAN
  - 2.3.8. OPTICS
  - 2.3.9. Birch
  - 2.3.10. 聚类性能度量
- 2.4. 双聚类
  - 2.4.1. Spectral Co-Clustering
  - 2.4.2. Spectral Biclustering
  - 2.4.3. Biclustering 评价
- 2.5. 分解成分中的信号（矩阵分解问题）
  - 2.5.1. 主成分分析（PCA）
  - 2.5.2. 截断奇异值分解和隐语义分析
  - 2.5.3. 词典学习
  - 2.5.4. 因子分析
  - 2.5.5. 独立成分分析（ICA）
  - 2.5.6. 非负矩阵分解(NMF 或 NNMF)
  - 2.5.7. 隐 Dirichlet 分配（LDA）
- 2.6. 协方差估计
  - 2.6.1. 经验协方差
  - 2.6.2. 收缩协方差
  - 2.6.3. 稀疏逆协方差
  - 2.6.4. 鲁棒协方差估计

- 2.7. 新奇点和离群点检测
  - 2.7.1. 离群点检测方法一览
  - 2.7.2. Novelty Detection（新奇点检测）
  - 2.7.3. Outlier Detection（离群点检测）
  - 2.7.4. 使用LOF进行新奇点检测
- 2.8. 密度估计
  - 2.8.1. 密度估计：直方图
  - 2.8.2. 核密度估计
- 2.9. 神经网络模型（无监督）
  - 2.9.1. 限制波尔兹曼机

我们一直在努力

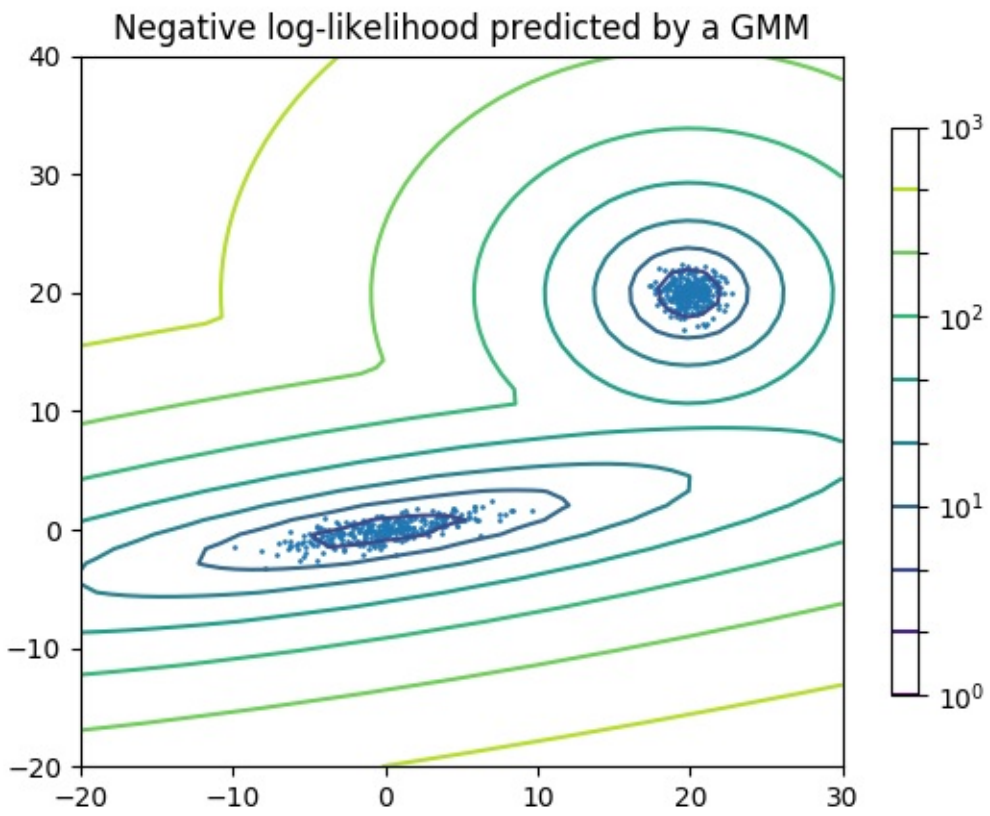
apachecn/sklearn-doc-zh



2.1. 高斯混合模型

校验者: @why2lyj@Shao Y.@Loopy@barrycg翻译者: @glassy

`sklearn.mixture` 是一个应用高斯混合模型进行非监督学习的包(支持 diagonal, spherical, tied, full 四种协方差矩阵), (注: *diagonal* 指每个分量有各自独立的对角协方差矩阵, *spherical* 指每个分量有各自独立的方差(再注:*spherical*是一种特殊的 *diagonal*, 对角的元素相等), *tied* 指所有分量共享一个标准协方差矩阵, *full* 指每个分量有各自独立的标准协方差矩阵), 它可以对数据进行抽样, 并且根据数据来估计模型。同时该包也支持由用户来决定模型内混合的分量数量。 (译注: 在高斯混合模型中, 我们将每一个高斯分布称为一个分量, 即 *component*)



二分量高斯混合模型:数据点, 以及模型的等概率线。

高斯混合模型是一个假设所有的数据点都是生成于有限个带有未知参数的高斯分布所混合的概率模型。 我们可以将这种混合模型看作是 k-means 聚类算法的一种扩展使用 (注:*kmeans*必须要求*cluster*的模型是圆形的,不能是椭圆的), 它包含了数据的协方差结构以及隐高斯模型的中心信息。

对应不同的估算策略,Scikit-learn 实现了不同的类来估算高斯混合模型。详细描述如下:

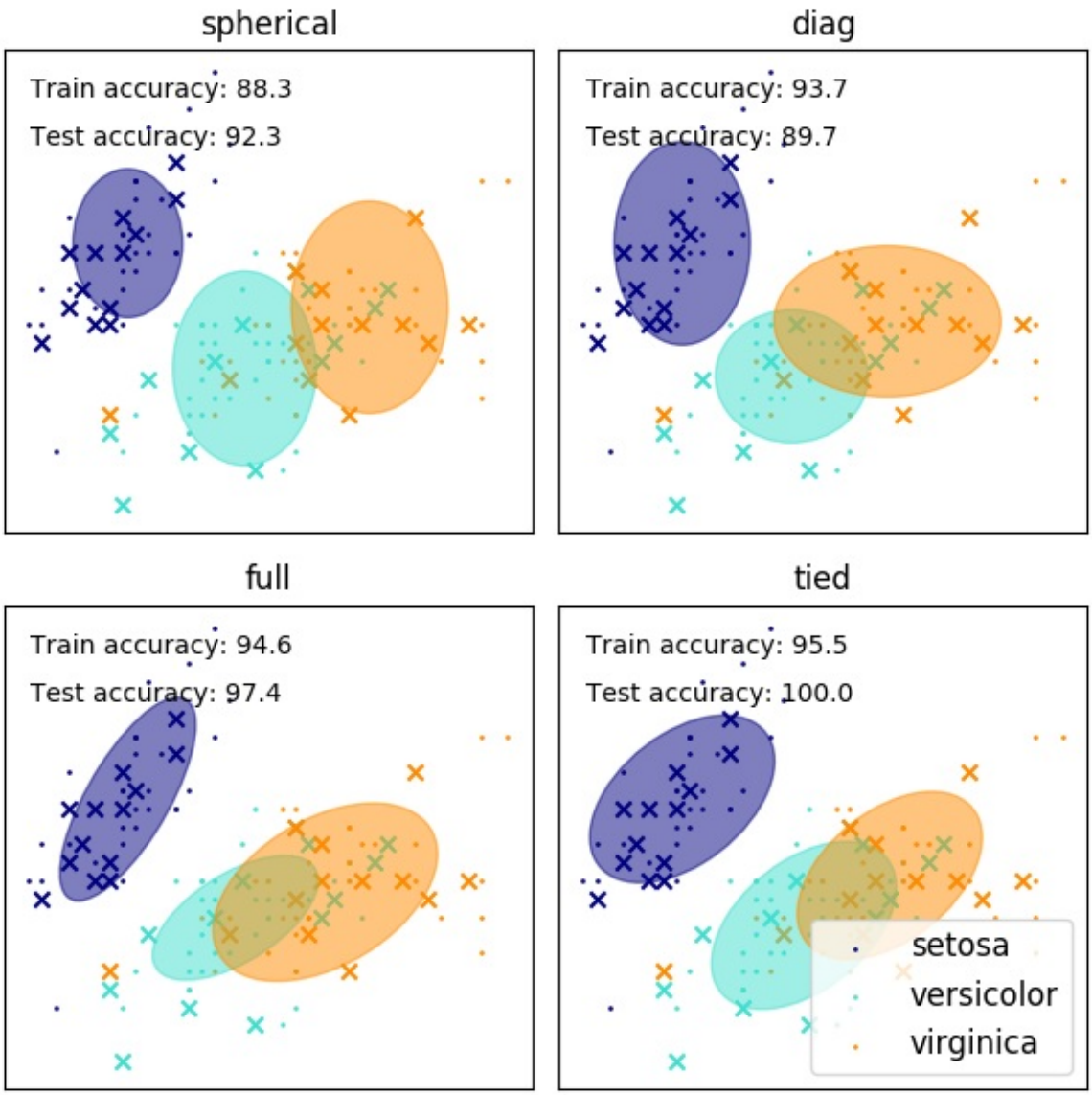
2.1.1. 高斯混合

`GaussianMixture` 对象实现了用来拟合高斯混合模型的 期望最大化 (EM) 算法。它还可以为多变量模型绘制置信椭圆体, 同时计算 BIC (Bayesian Information Criterion, 贝叶斯信息准则) 来评估数据中聚类的数量。而

其中的 `GaussianMixture.fit` 方法可以从训练数据中拟合出一个高斯混合模型。

如果给定测试数据，通过使用 `GaussianMixture.predict` 方法，可以为每个样本分配最适合的高斯分布模型。

`GaussianMixture` 方法中自带了不同的选项来约束不同估类的协方差：`spherical`，`diagonal`，`tied` 或 `full` 协方差。



示例：

- 一个利用高斯混合模型在鸢尾花卉数据集（IRIS 数据集）上做聚类的协方差实例，请查阅 GMM covariances
- 一个绘制密度估计的例子，请查阅 Density Estimation for a Gaussian mixture

2.1.1.1. 优缺点 GaussianMixture

2.1.1.1.1. 优点

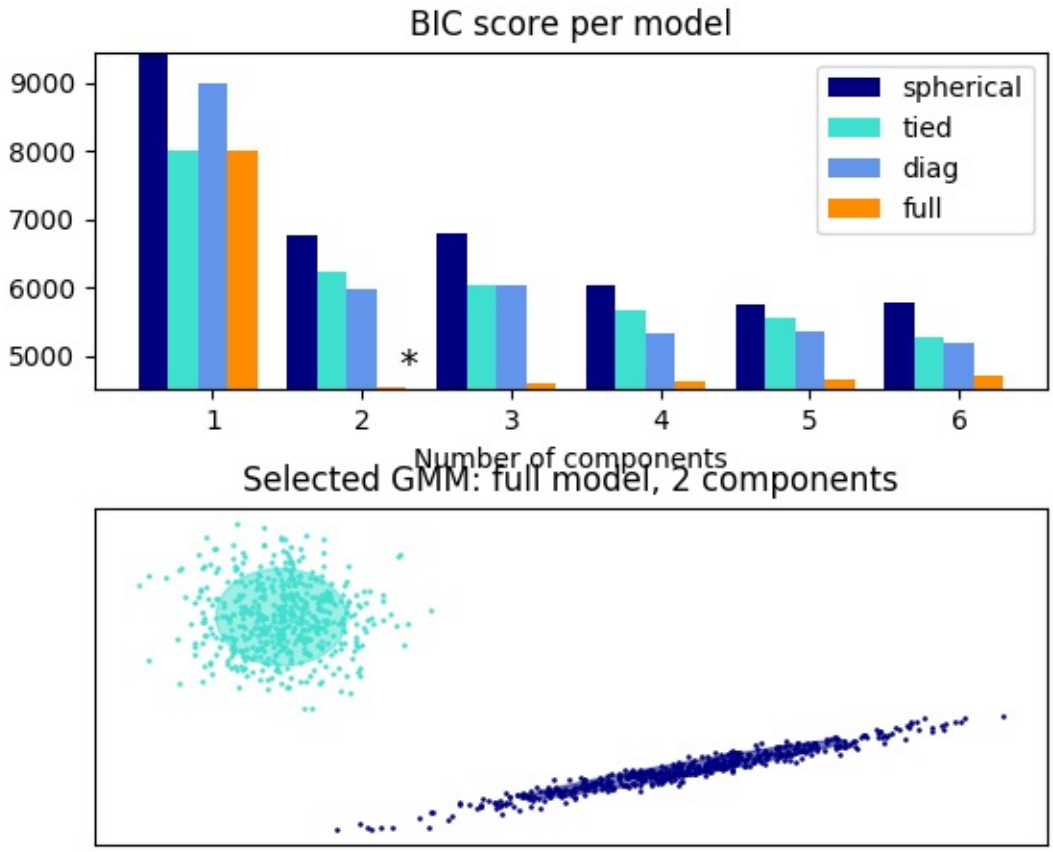
- 速度： 是混合模型学习算法中最快的算法.
- 无偏差性： 这个算法仅仅只是最大化可能性，并不会使均值偏向于0，或是使聚类大小偏向于可能适用或者可能不适用的特殊结构。

2.1.1.1.2. 缺点

- 奇异性：当每个混合模型没有足够多的点时，会很难去估算对应的协方差矩阵，同时该算法会发散并且去寻找具有无穷大似然函数值的解，除非人为地正则化相应的协方差。
- 分量的数量：这个算法总是会使用它所能用的全部分量，所以在缺失外部线索的情况下，需要留存数据或者信息理论标准来决定用多少个分量。

2.1.1.2. 选择经典高斯混合模型中分量的个数

通过BIC（贝叶斯信息准则）来选择高斯混合的分量数是一种高效的方法。 理论上，它仅当在近似状态下可以恢复正确的分量数（即如果有大量数据可用，并且假设这些数据（具有独立同分布属性）实际上是一个混合高斯模型生成的）。注意：使用 变分贝叶斯高斯混合 可以避免高斯混合模型中分量数量的选择。



> 示例：
>
> - 一个用典型的高斯混合进行的模型选择的例子，请查阅 `Gaussian Mixture Model Selection`

2.1.1.3. 估计算法期望最大化（EM）

在从无标记的数据中应用高斯混合模型主要的困难在于：通常不知道哪个点来自哪个隐分量 （如果可以获取到这些信息，就可以很容易通过相应的数据点，拟合每个独立的高斯分布）。 期望最大化（`Expectation-maximization, EM`） 是一个理论完善的统计算法，其通过迭代方式来解决这个问题。首先，假设一个随机分量（随机地选择一个中心点，该点可以由 `k-means` 算法得到，或者甚至可以在原点周围随意选取一个点），并为每个点分别计算由该混合模型内的每个分量生成的概率。然后，调整模型参数以最大化模型生成这些参数的可能性。重复

这个过程，该算法保证该过程内的参数总会收敛到一个局部最优解。

2.1.2. 变分贝叶斯高斯混合

`BayesianGaussianMixture` 对象实现了具有变分推理算法的高斯混合模型的变体。 这个对象的API和 `GaussianMixture` 所定义的API类似。

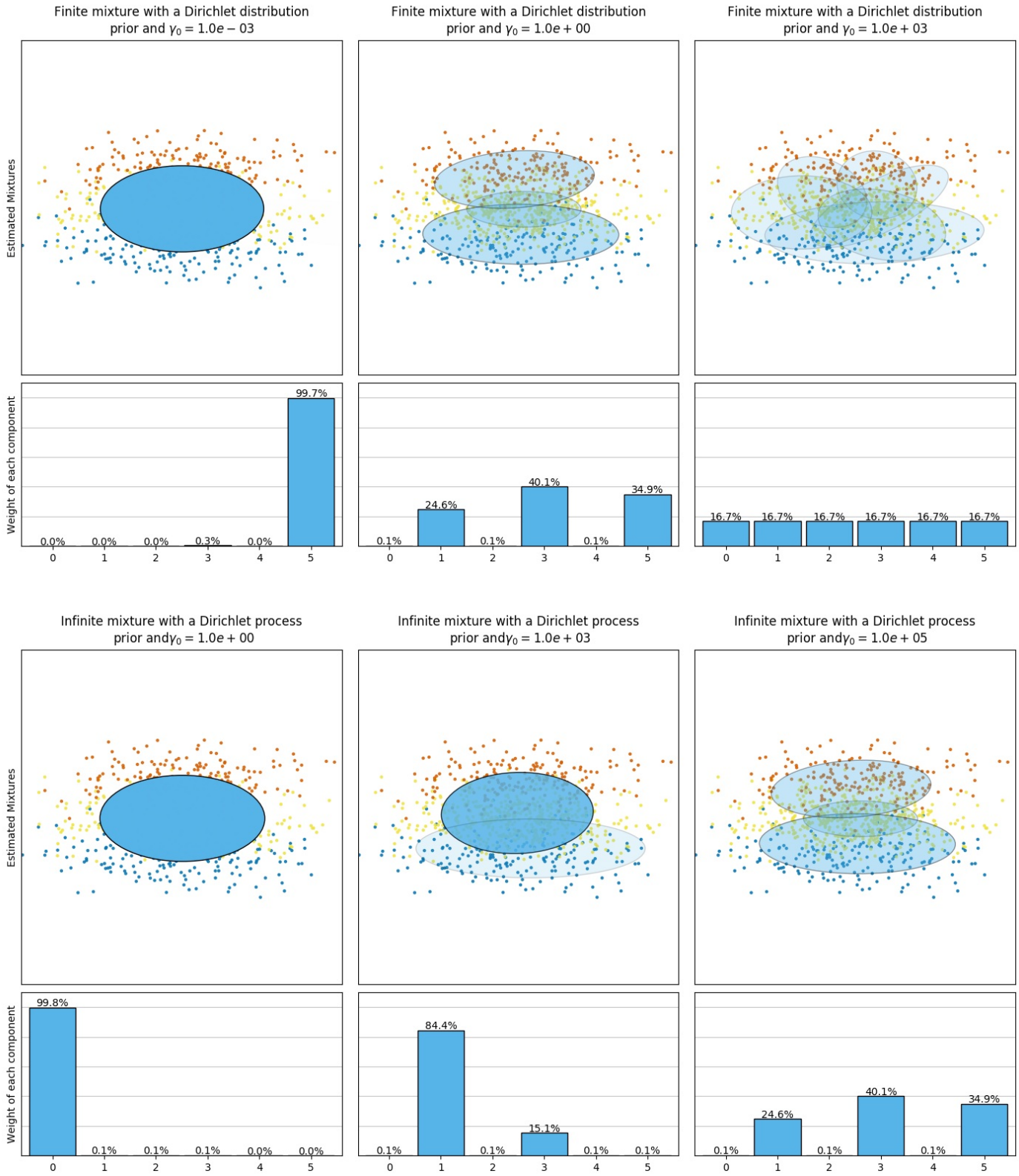
2.1.2.1. 估计算法：变分推断（variational inference）

变分推断是期望最大化（EM）的扩展，它最大化了模型证据（包括先验）的下界，而不是数据的似然性。 变分方法的原理与期望最大化相同（二者都是迭代算法，在寻找由每种模型的混合所产生的每个点的概率和根据所分配的点拟合之间两步交替），但是变分方法通过整合先验分布信息来增加正则化限制。 这避免了期望最大化解决方案中常出现的奇异性，但是也给模型带来了微小的偏差。变分的计算过程通常比较慢，但不会慢到无法使用。

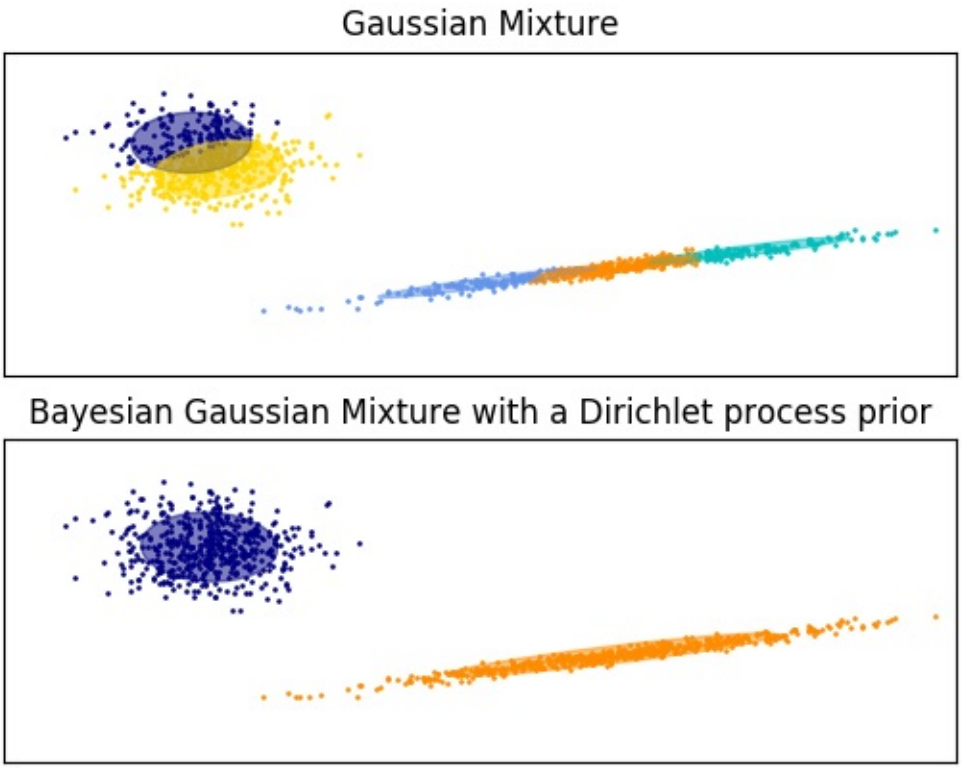
由于它的贝叶斯特性，变分算法比预期最大化（EM）需要更多的超参数（即先验分布中的参数），其中最重要的就是浓度参数 `weight_concentration_prior` 。指定一个低浓度先验，将会使模型将大部分的权重分配到少数分量上，而其余分量的权重则趋近 0。而高浓度先验将会使混合模型中的更多的分量在混合模型中都有相当比列的权重。

`BayesianGaussianMixture` 类的参数实现提出了两种权重分布先验： 一种是利用 Dirichlet distribution（狄利克雷分布）的有限混合模型，另一种是利用 Dirichlet Process（狄利克雷过程）的无限混合模型。在实际应用中，狄利克雷过程推理算法是近似的，并且使用具有固定最大分量数的截断分布（称之为 Stick-breaking representation）， 而其使用的分量数实际上总是依赖数据。

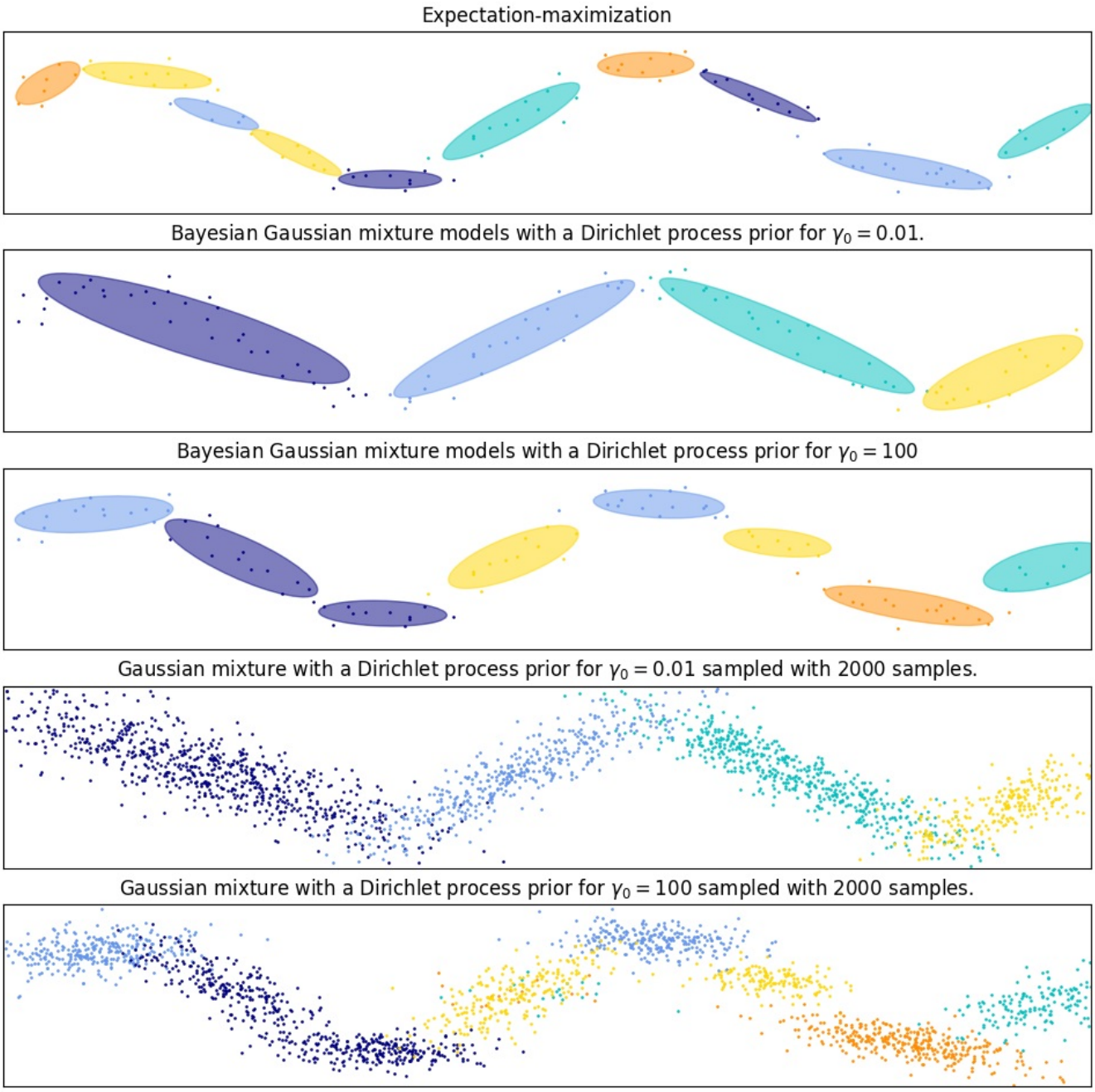
下图比较了不同类型的权重浓度先验（参数 `weight_concentration_prior_type` ）组合不同的 `weight_concentration_prior` 的值时，所获得的结果。在此处，我们可以发现 `weight_concentration_prior` 参数的值对获得的有效激活的分量数（即权重较大的分量的数量）有很大影响。 我们也能注意到当先验类型是 ‘dirichlet_distribution’ 时，大的浓度权重先验会导致更均匀的权重，然而这种情况却并不必然适用于 ‘dirichlet_process’ 类型（默认类型）。



下面的例子将具有固定数量分量的高斯混合模型与具有Dirichlet process prior（狄利克雷过程先验）的变分高斯混合模型进行了比较。此处，一个经典高斯混合模型被指定由 5 个分量（其中两个是聚类）在某个数据集上拟合而成。 我们可以看到，具有狄利克雷过程的变分高斯混合模型 在相同的数据上进行拟合时,可以将自身分量数量制在 2 。在该例子中，用户选择了 `n_components=5` ，这与真正的试用数据集（toy dataset）的生成分量数量不符。很容易注意到， 狄利克雷过程先验的变分高斯混合模型可以采取保守的策略，仅拟合生成一个分量。



在下图中，我们将拟合一个并不能被高斯混合模型很好描述的数据集。 调整 `BayesianGaussianMixture` 类的参数 `weight_concentration_prior` ，这个参数决定了用来拟合数据的分量数量。我们在最后两个图上展示了从两个混合结果中产生的随机抽样。



示例：

- 一个用 `GaussianMixture` 和 `BayesianGaussianMixture` 绘制置信椭圆体的例子， 请查阅 `Gaussian Mixture Model Ellipsoids`
- `Gaussian Mixture Model Sine Curve` 这个例子展示了用 `GaussianMixture` 和 `BayesianGaussianMixture` 来拟合正弦波。
- 一个使用不同的 `weight_concentration_prior_type` 用以不同的 `weight_concentration_prior` 参数值的 *BayesianGaussianMixture* 来绘制置信椭圆体的例子。 请查阅 `Concentration Prior Type Analysis of Variation Bayesian Gaussian Mixture`

2.1.2.2. BayesianGaussianMixture 下的变分推理的优缺点

2.1.2.2.1. 优点

- 自动选择：当 `weight_concentration_prior` 足够小以及 `n_components` 比模型需要的数量更大时，变分贝叶

斯混合模型计算的结果可以让一些混合权重值趋近 0。 这让模型可以自动选择合适的有效分量数量。这过程仅仅需要提供分量的数量上限。但是请注意，“理想” 的激活分量数量具有应用独特性，在数据挖掘参数设置中通常并不明确。

- 对参数数量的敏感度较低:和总是使用全部能用分量，因而由不同数量的分量来组合过量解决方案的有限模型不同， 带有狄利克雷过程先验的变分推理（ `weight_concentration_prior_type='dirichlet_process'` ）的输出并不总拥有和参数一致的变化， 因此该变分推理更加稳定和需要更少的调优。
- 正则化： 由于结合了先验信息，变分的解比期望最大化（EM）的解有更少的病理特征（pathological special cases）。

2.1.2.2.2. 缺点

- 速度： 变分推理所需要的额外参数化使推理速度变慢，尽管慢得不多。
- 超参数： 这个算法需要一个额外的可能需要交叉验证进行实验调优的超参数。
- 偏差： 在推理算法存在许多隐含的偏差（如果用到狄利克雷过程也会有偏差）， 每当这些偏差和数据之间不匹配时，用有限模型可能会拟合出更好的模型。

2.1.2.3. The Dirichlet Process（狄利克雷过程）

此处，我们描述了基于狄利克雷过程混合的变分推理算法。狄利克雷过程是在 具有无限大，无限制的分区数的聚类上的先验概率分布。相比于有限高斯混合模型，变分技术让我们在推理时间几乎没有惩罚（penalty）的情况下纳入了高斯混合模型的先验结构。

一个重要的问题是狄利克雷过程是如何实现用无限的，无限制的聚类数，并且结果仍然是一致的。 本文档不做出完整的解释，但是你可以看这里 stick breaking process 来帮助你理解它。折棍（stick breaking）过程是狄利克雷过程的衍生。我们每次从一个单位长度的 stick 开始，且每一步都折断剩余部分的一部分。每次，我们把每个 stick 的长度联想成所有点里落入一组混合的点的比例。 最后，为了表示无限混合，我们把 stick 的最后剩余部分联想成没有落入其他所有组的点的比例。每段的长度是一个随机变量， 与浓度参数成概率比例。较小的浓度值将单位长度分成较大的 stick 段（即定义更集中的分布）。较高的浓度值将生成更小的 stick 段（即增加非零权重的分量数量）。

狄利克雷过程的变分推理技术，在对该无限混合模型进行有限近似的情形下，仍然可以运用。用户不必事先指定想要的分量数量，只需要指定浓度参数和混合分量数的上界（假定上界高于“真实”的分量数，这仅仅影响算法复杂度，而不是实际使用的分量数量）。

我们一直在努力

apachecn/sklearn-doc-zh

加入QQ群

2.2. 流形学习

校验者：@XuJianzhi@RyanZhiNie@羊三@Loopy@barrycg翻译者：@XuJianzhi@羊三

```
Look for the bare necessities

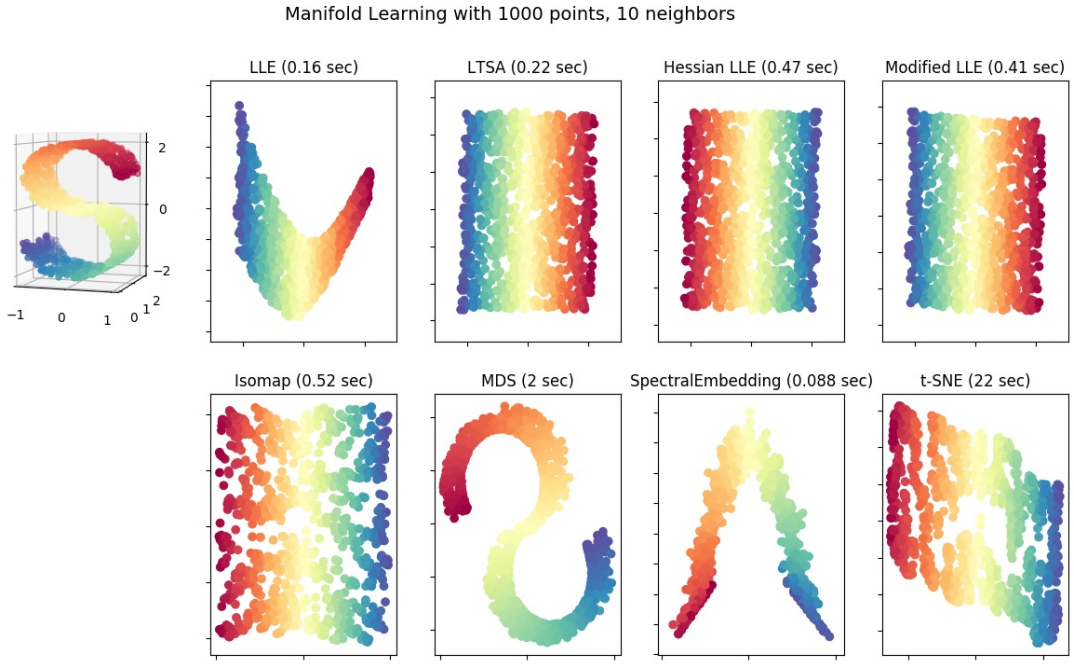
The simple bare necessities

Forget about your worries and your strife

I mean the bare necessitiesOld Mother Nature’s recipes

That bring the bare necessities of life

– Baloo的歌 [奇幻森林]
```

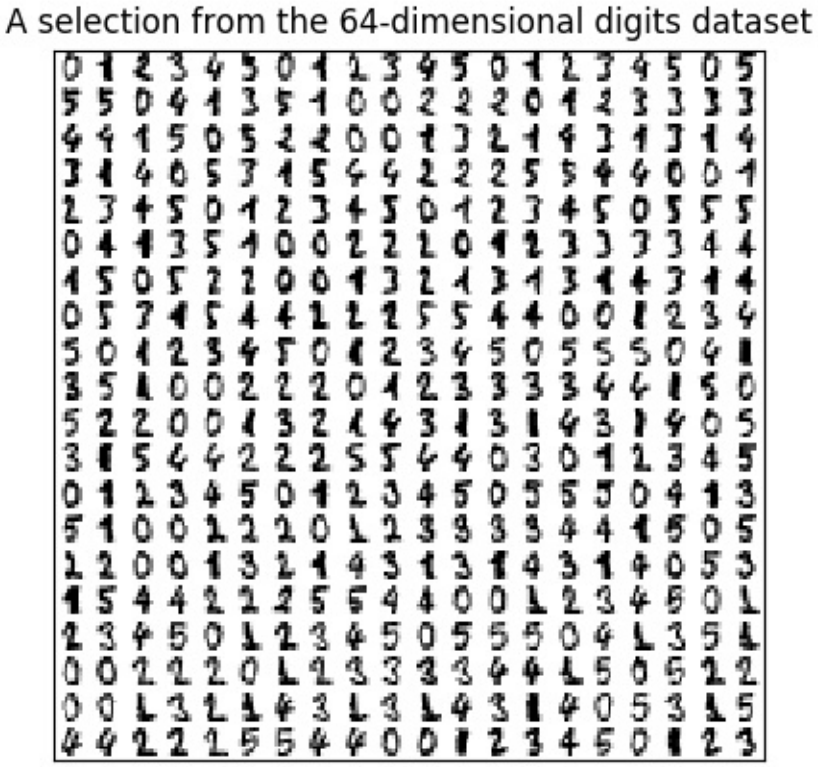


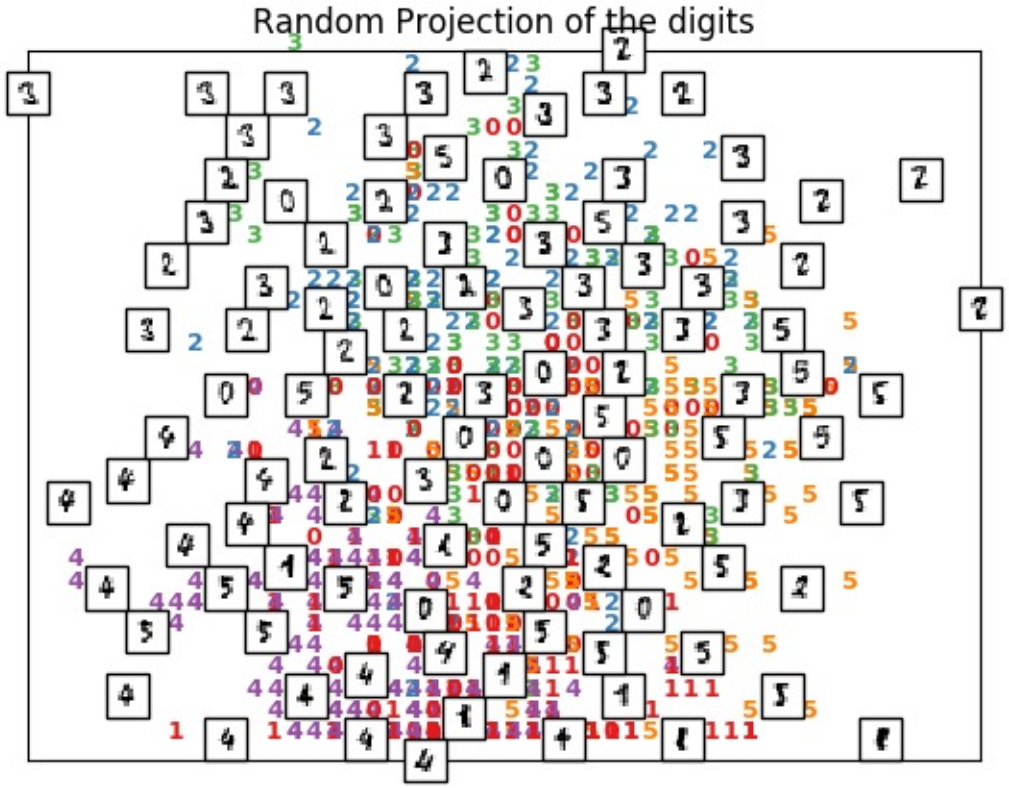
流形学习是一种非线性降维方法。其算法基于的思想是：许多数据集维度过高的现象完全是人为导致得。

2.2.1. 介绍

高维数据集通常难以可视化。虽然，可以通过绘制两维或三维的数据来显示高维数据的固有结构，但与之等效的高维图不太直观。为了促进高维数据集结构的可视化，必须以某种方式降低维度。

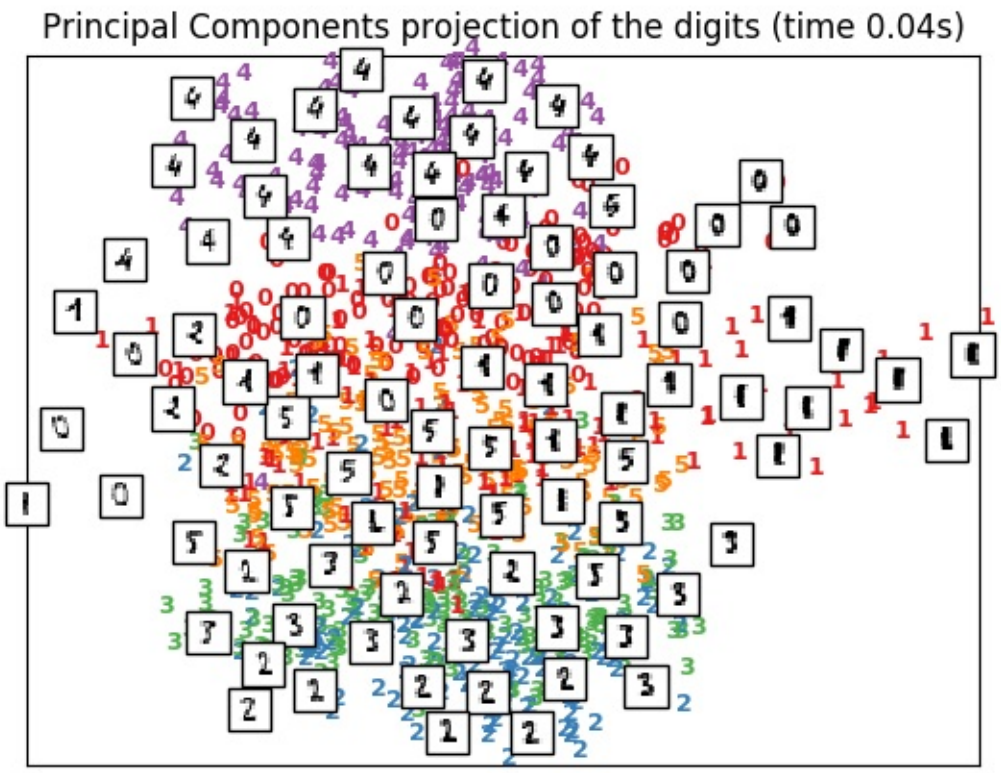
通过对数据的随机投影来实现降维是最简单的方法。虽然这样做能实现数据结构一定程度的可视化，但这种随机选择方式仍有许多有待改进之处。在随机投影过程中，数据中更有趣的结构很可能会丢失。

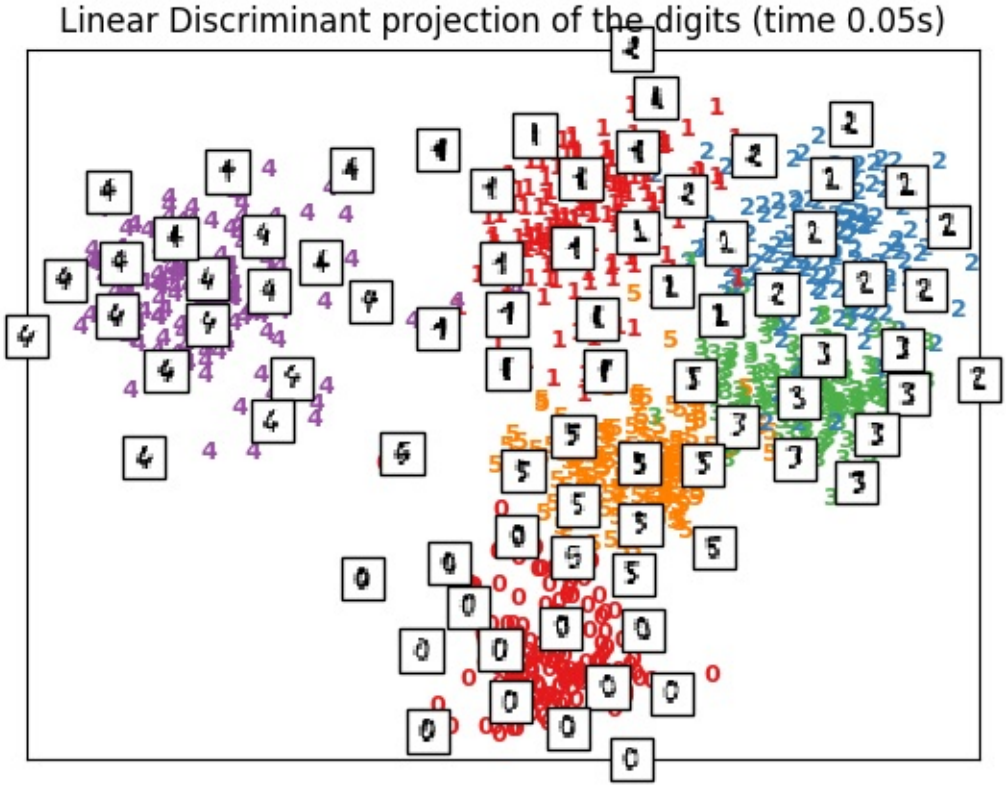




为了解决这一问题，一些监督和无监督的线性降维框架被设计出来，如主成分分析（PCA），独立成分分析以及线性判别分析等。这些算法明确规定了如何来选择数据的“有趣的”线性投影。它们虽然高效，但是经常错失数据中重要的非

线性结构。





流形学习可以被认为是将线性框架（如 PCA ）推广到对数据中非线性结构敏感的一次尝试。虽然存在监督变量，但

是典型的流形学习问题是无监督的：它从数据本身学习数据的高维结构，而不使用预定的分类。

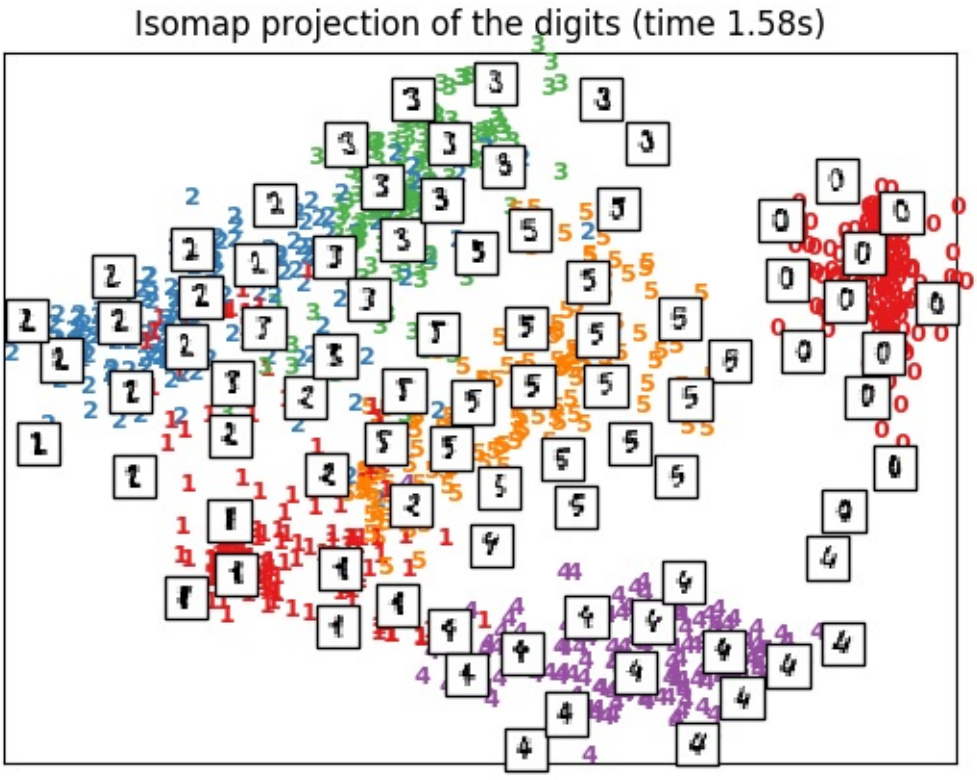
示例：

- 参见 Manifold learning on handwritten digits: Locally Linear Embedding, Isomap… ，手写数字降维的例子。
- 参见 Comparison of Manifold Learning methods ，玩具 “S曲线” 数据集降维的例子。

以下概述了 scikit-learn 中可用的流形学习实现

2.2.2. Isomap

流形学习的最早方法之一是 Isomap 算法，等距映射（Isometric Mapping）的缩写。Isomap 可以被视为多维缩放（Multi-dimensional Scaling: MDS）或核主成分分析（Kernel PCA）的扩展。Isomap 寻求一个较低维度的嵌入(译注：嵌入*(embedding)*，在此处，可以理解为高维数据到低维数据的一种映射转换，数据间的固有结构不变化)，它保持了所有点之间的原有的测地距离(译注:测地距离（*geodesic distance*）是指在图中连接某两个顶点的最短距离*(shortest path)*)。Isomap 可以通过 `Isomap` 对象执行。



2.2.2.1. 复杂度

Isomap 算法包括三个阶段：

- 最近邻搜索. Isomap 使用 `sklearn.neighbors.BallTree` 进行有效的近邻搜索。 对于 D 维中 N 个点的 k 个最近邻，代价约为 $O[D \log(k) N \log(N)]$

- 最短路径图搜索．该类算法中已知最有效的算法是 Dijkstra 算法或 Floyd-Warshall 算法，其复杂度分别是约 $O[N^2(k + \log(N))]$ 和 $O[N^3]$ 。 用户可通过使用 isomap 的 path_method 关键字来选择该算法。 如果未指定，则代码自行尝试为输入数据选择最佳算法。
- 部分特征值分解．对应于 $N \times N$ isomap核中 d 个最大特征值的特征向量，进行嵌入编码。 对于密集求解器，代价约为 $O[dN^2]$ 。 通常可以使用 ARPACK 求解器来减少代价。 用户可通过使用 isomap 的 path_method 关键字指定特征求解器。 如果未指定，则代码自行尝试为输入数据选择最佳算法。Isomap 的整体复杂度是 $O[D \log(k) N \log(N)] + O[N^2(k + \log(N))] + O[dN^2]$.

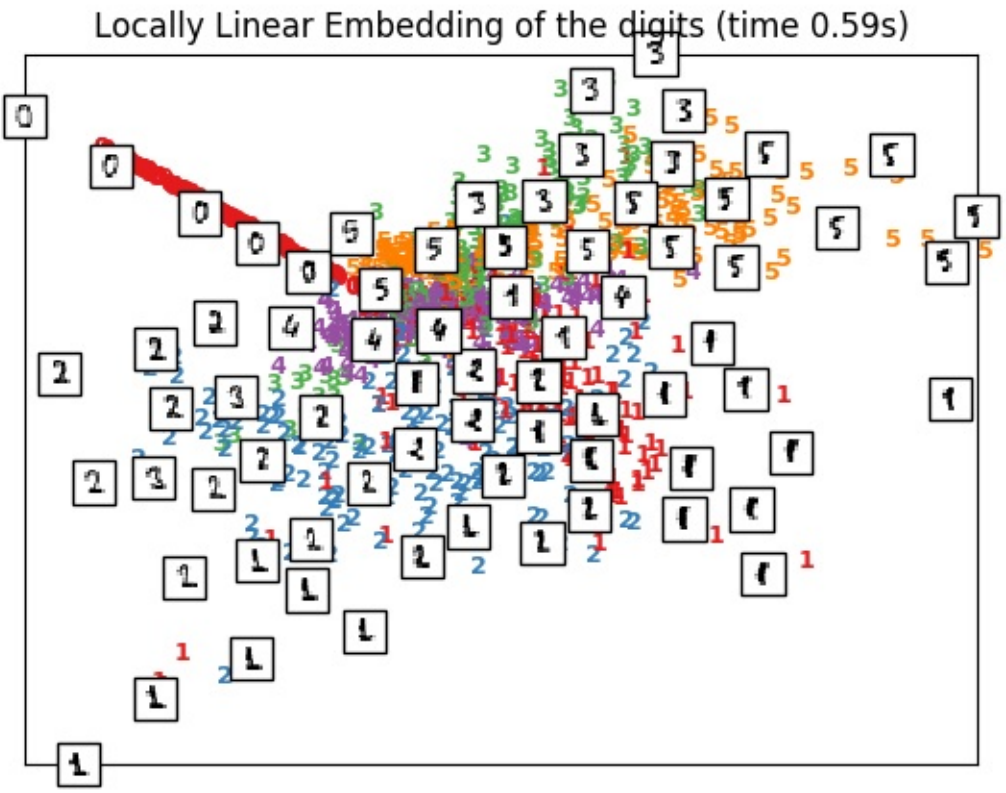
- N ： 训练数据点的个数
- D ： 输入维度
- k ： 最近邻的个数
- d ： 输出维度

> 参考资料：
>
> - “A global geometric framework for nonlinear dimensionality reduction” Tenenbaum, J.B.; De Silva, V.; & Langford, J.C. Science 290 (5500)

2.2.3. 局部线性嵌入

局部线性嵌入（LLE）通过保留局部邻域内的距离来寻求数据的低维投影。 它可以被认为是一系列的局部主成分分析在全局范围内的相互比较，找到最优的局部非线性嵌入。

局部线性嵌入可以使用 `locally_linear_embedding` 函数或其面向对象的等效方法 `LocallyLinearEmbedding` 来实现。



2.2.3.1. 复杂度

标准的 LLE 算法包括三个阶段：

- 最邻近搜索。参见上述 Isomap 讨论。
- 构造权重矩阵。$O[DNk^3]$。LLE 权重矩阵的构造包括每个 N 局部邻域的 $k \times k$ 线性方程的解。
- 部分特征值分解。参见上述 Isomap 讨论。标准 LLE 的整体复杂度是

$$O[D \log(k) N \log(N)] + O[DNk^3] + O[dN^2].$$

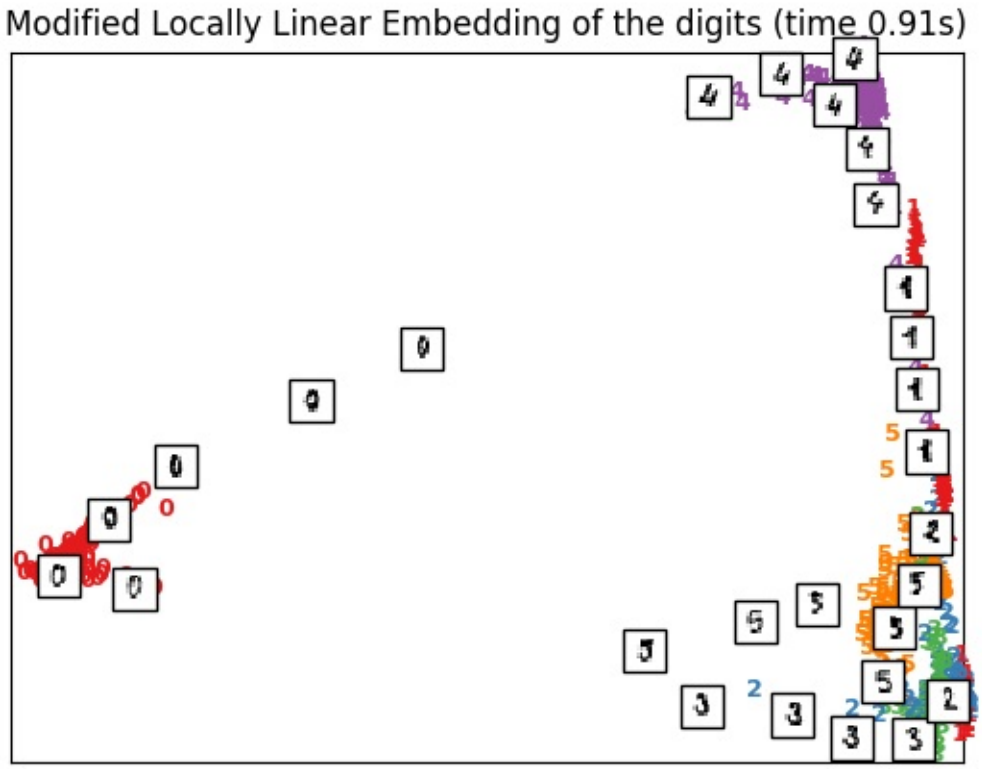
- N ： 训练数据点的个数
- D ： 输入维度
- k ： 最近邻的个数
- d ： 输出维度

> 参考资料：
>
> - “Nonlinear dimensionality reduction by locally linear embedding” Roweis, S. & Saul, L. Science 290:2323 (2000)

2.2.4. 改进型局部线性嵌入（MLLE）

局部线性嵌入（LLE）的一个众所周知的问题是正则化问题。当 neighbors（邻域）的数量多于输入的维度数量时，定义每个局部邻域的矩阵是不满秩的。为解决这个问题，标准的局部线性嵌入算法使用一个任意正则化参数 r ，它的取值受局部权重矩阵的迹的影响。虽然我们普遍认为，当正则化参数 $r \to 0$ ，解向目标嵌入情况收敛，但是当 $r > 0$ 时，并不保证得到最优解。以上说明了，在嵌入过程中的正则化问题会扭曲流形的内部几何形状，使其失真。

解决正则化问题的一种方法是对邻域使用多个权重向量。这就是改进型局部线性嵌入（MLLE）算法的精髓。MLLE 可被执行于函数 `locally_linear_embedding` ，或者面向对象的副本 `LocallyLinearEmbedding` ，附带关键词 `method = 'modified'` 。它需要满足 `n_neighbors > n_components` 。



2.2.4.1. 复杂度

MLLE 算法分为三部分：

- 近邻搜索。与标准 LLE 的相同。
- 权重矩阵构造。大约是 $O[DNk^3] + O[N(k-D)k^2]$ 。该式第一项恰好等于标准 LLE 算法的复杂度。该式第二项与由多个权重来构造权重矩阵相关。在实践中，构造 MLLE 权重矩阵所增加的损耗（就复杂度而言），比其它两部分要小。
- 部分特征值分解。与标准 LLE 的相同。综上，MLLE 的复杂度为 $O[D\log(k)N\log(N)] + O[DNk^3] + O[N(k-D)k^2] + O[dN^2]$ 。

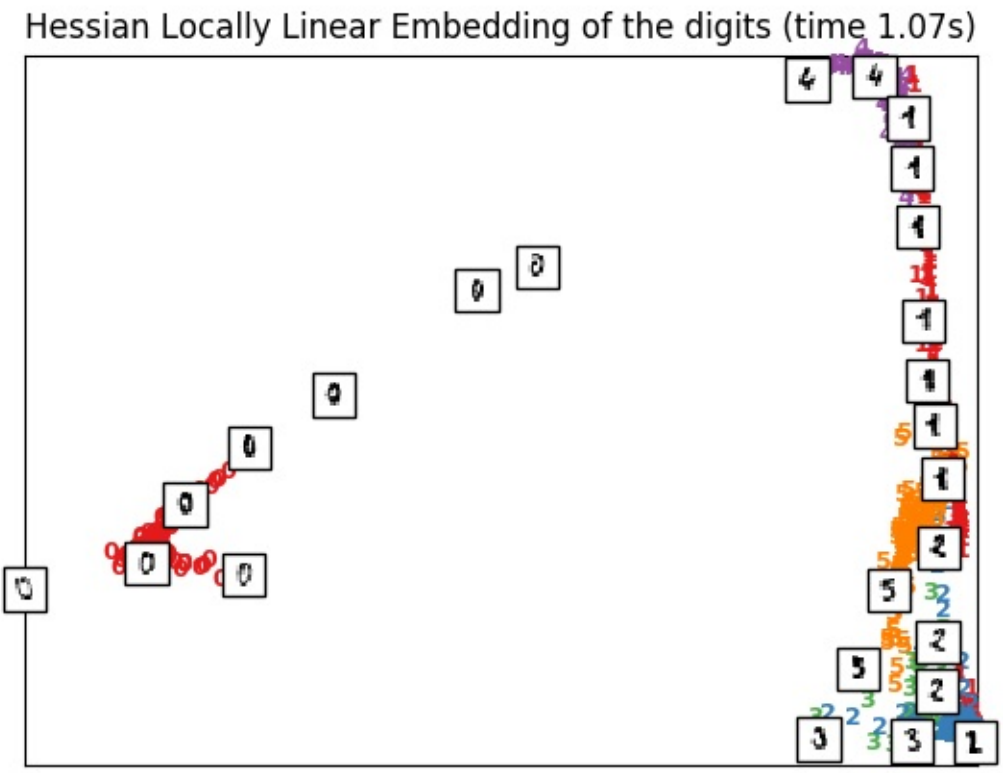
- N ： 训练集数据点的个数

- D ： 输入维度
- k ： 最近邻域的个数
- d ： 输出的维度

> 参考资料：
>
> - “MLLE: Modified Locally Linear Embedding Using Multiple Weights” Zhang, Z. & Wang, J.

2.2.5. 黑塞特征映射（HE）

黑塞特征映射（也称作基于黑塞的 LLE：HLLE ）是解决 LLE 正则化问题的另一种方法。在每个用于恢复局部线性结构的邻域内，它会围绕一个基于黑塞的二次型展开。虽然其它的实现表明它对数据大小进行缩放的能力较差，但是 sklearn 实现了一些算法改进，使得在输出低维度时它的损耗可与其他 LLE 变体相媲美。HLLE 可实现为函数 `locally_linear_embedding` 或其面向对象的形式 `LocallyLinearEmbedding` ，附带关键词 `method = 'hessian'` 。它需满足 `n_neighbors > n_components * (n_components + 3) / 2` 。



2.2.5.1. 复杂度

HLLE 算法分为三部分：

- 近邻搜索。与标准 LLE 的相同。

- 权重矩阵构造．大约是 $O[DNk^3] + O[Nd^6]$ 。其中第一项与标准 LLE 相似。第二项来自于局部黑塞估计量的一个 QR 分解。
- 部分特征值分解。与标准 LLE 的相同。综上，HLLE 的复杂度为 $O[D \log(k) N \log(N)] + O[DNk^3] + O[Nd^6] + O[dN^2]$ 。

- N ： 训练集数据点的个数
- D ： 输入维度
- k ： 最近邻域的个数
- d ： 输出的维度

参考资料：

- “Hessian Eigenmaps: Locally linear embedding techniques for high-dimensional data” Donoho, D. & Grimes, C. Proc Natl Acad Sci USA. 100:5591 (2003)

2.2.6. 谱嵌入

谱嵌入是计算非线性嵌入的一种方法。scikit-learn 执行拉普拉斯特征映射，该映射是用图拉普拉斯的谱分解的方法把数据进行低维表达。这个生成的图可认为是低维流形在高维空间里的离散近似值。基于图的代价函数最小化确保了流形上彼此临近的点被映射后在低维空间也彼此临近，低维空间保持了局部距离。谱嵌入可执行为函数 `spectral_embedding` 或它的面向对象的对应形式 `SpectralEmbedding` 。

2.2.6.1. 复杂度

谱嵌入（拉普拉斯特征映射）算法含三部分：

- 加权图结构。把原始输入数据转换为用相似（邻接）矩阵表达的图表达。
- 图拉普拉斯结构。非规格化的图拉普拉斯是按 $L = D - A$ 构造，并按 $L = D^{-\frac{1}{2}}(D - A)D^{-\frac{1}{2}}$ 规格化的。
- 部分特征值分解。在图拉普拉斯上进行特征值分解。综上，谱嵌入的复杂度是 $O[D \log(k) N \log(N)] + O[DNk^3] + O[dN^2]$ 。

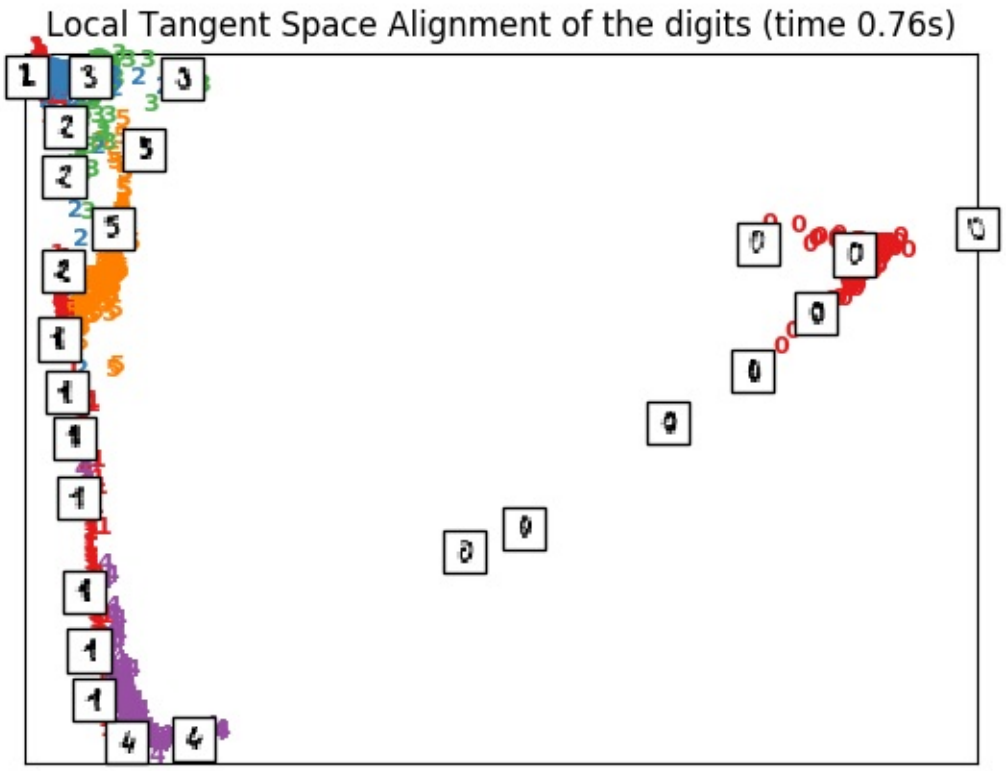
- N ： 训练集数据点的个数
- D ： 输入维度
- k ： 最近邻域的个数
- d ： 输出的维度

参考资料：

- “Laplacian Eigenmaps for Dimensionality Reduction and Data Representation” M. Belkin, P. Niyogi, Neural Computation, June 2003; 15 (6):1373-1396

2.2.7. 局部切空间对齐（LTSA）

尽管，严格意义上来说，局部切空间对齐(LTSA) 并不是LLE的变体，但是，从算法角度来说，它们俩又是足够接近的，所以把它放在该目录下。与 LLE 算法关注于保持临点距离不同，LTSA 寻求通过切空间来描述局部几何形状，并（通过）实现全局最优化来对其这些局部切空间，从而得知对应的嵌入。 LTSA 可执行为函数 `locally_linear_embedding` 或它的面向对象的对应形式 `LocallyLinearEmbedding` ，附带关键词 `method = 'ltsa'` 。



2.2.7.1. 复杂度

LTSA 算法含三部分：

- 近邻搜索。与标准 LLE 的相同。
- 加权矩阵构造。大约是 $O[DNk^3] + O[k^2d]$ 。其中第一项与标准 LLE 相似。
- 部分特征值分解。同于标准 LLE 。综上，复杂度是 $O[D \log(k) N \log(N)] + O[DNk^3] + O[k^2d] + O[dN^2]$ 。
- N ： 训练集数据点的个数

- D ： 输入维度
- k ： 最近邻域的个数
- d ： 输出的维度

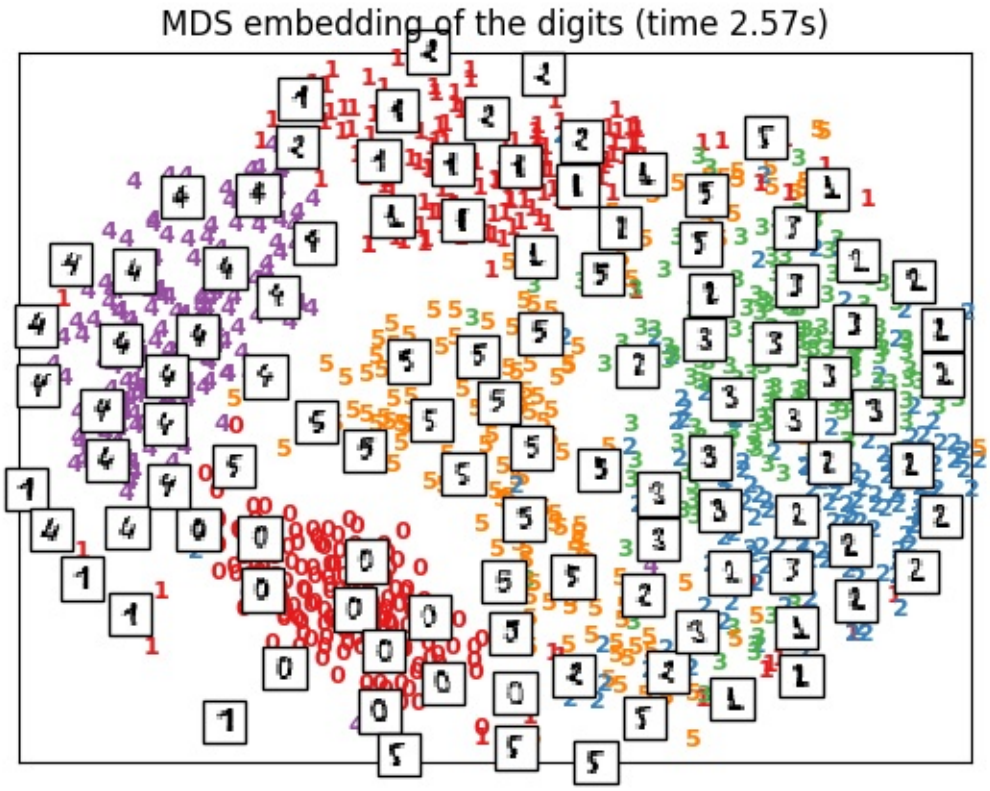
> 参考资料：
>
> - “Principal manifolds and nonlinear dimensionality reduction via tangent space alignment” Zhang, Z. & Zha, H. Journal of Shanghai Univ. 8:406 (2004)

2.2.8. 多维尺度分析（MDS）

多维尺度分析 Multidimensional scaling （ `MDS` ） 寻求数据的低维表示，而这些低维数据间的距离保持了它们在初始高维空间中的距离。

一般来说，（MDS）是一种用来分析在几何空间距离相似或相异数据的技术。MDS 尝试在几何空间上将相似或相异的数据进行建模。这些数据可以是物体间的相似等级，也可是分子的作用频率，还可以是国家简单贸易指数。

MDS算法有2类：度量和非度量。在 scikit-learn 中， `MDS` 类具有上述两者的实现。在度量 MDS 中，输入相似度矩阵源自度量(并因此遵从三角形不等式)，输出两点之间的距离被设置为尽可能接近相似度或相异度的数据。在非度量版本中，算法尝试保持距离的控制，并因此寻找在所嵌入空间中的距离和相似/相异之间的单调关系。



设 S 是相似度矩阵，X 是 n 个输入点的坐标。差异 $\hat{d}_{ij}$ 是某些最优方式所选择的相似度转换。然后，通过

$\sum_{i<j} d_{ij}\left(X\right) - \hat{d}_{ij}\left(X\right)${i < j} d{ij}(X) - \hat{d}_{ij}(X)∑i<jdij(X)−d^ij(X) 定义称为 Stress （应力值）的对象。

2.2.8.1. 度量 MDS

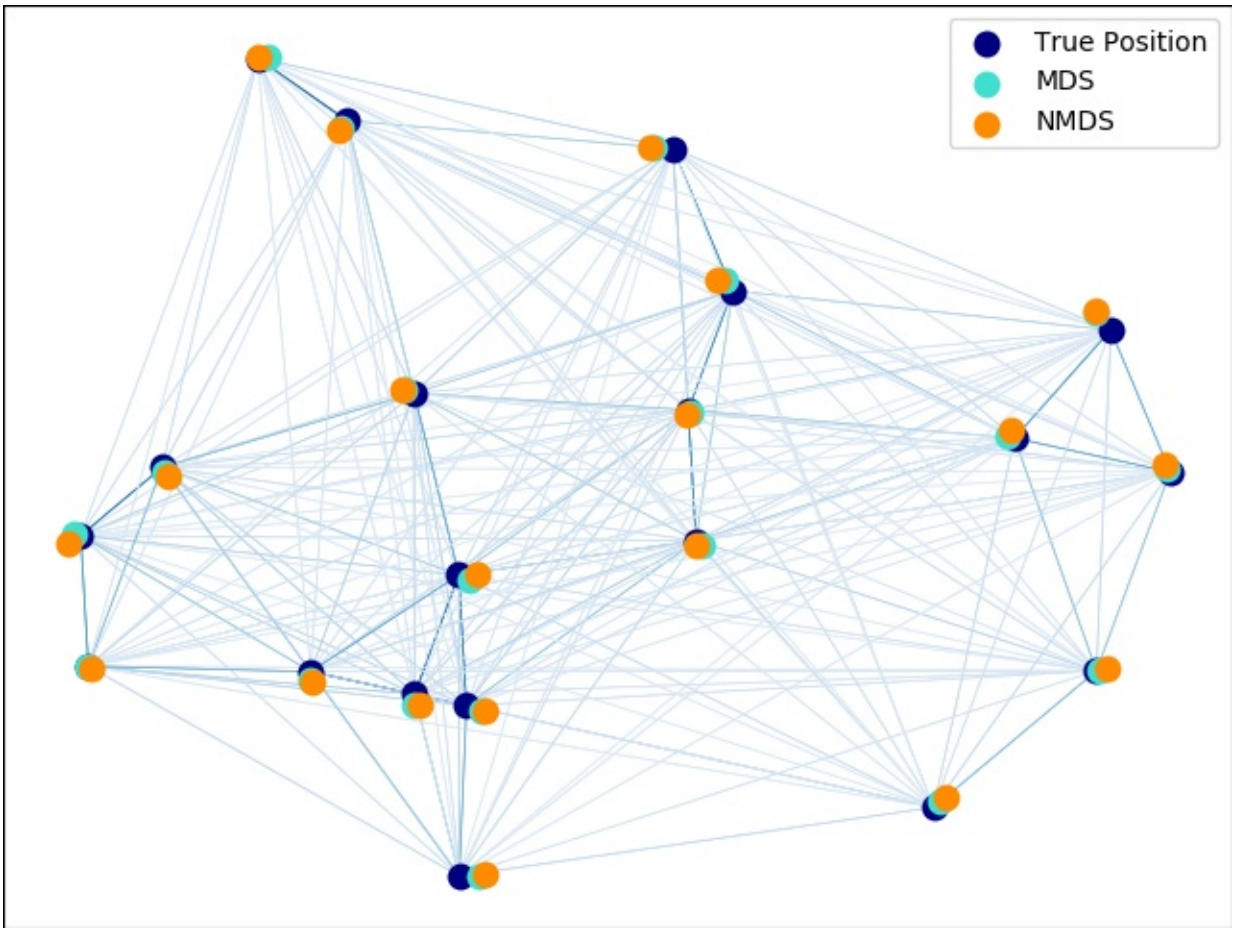
最简单的度量 MDS 模型称为 *absolute MDS*（绝对*MDS*），差异由 $\hat{d}_{ij} = S_{ij}$ 定义。对于绝对 MDS，值 S_{ij} 应精确地对应于嵌入点的点 i 和 j 之间的距离。

大多数情况下，差异应设置为 $\hat{d}_{ij} = bS_{ij}$ 。

2.2.8.2. 非度量 MDS

非度量 MDS 关注数据的排序。如果 $S_{ij} < S_{jk}${ij} < S{jk}Sij<Sjk ，则嵌入应执行 $d_{ij} < d_{jk}$ 。这样执行的一个简单算法是在 S_{ij} 上使用 d_{ij} 的单调回归，产生与 S_{ij} 相同顺序的差异 $\hat{d}_{ij}$ 。

此问题的 a trivial solution（一个平凡解）是把所有点设置到原点上。为了避免这种情况，将差异 S_{ij} 标准化。



参考资料：

- “Modern Multidimensional Scaling - Theory and Applications” Borg, I.; Groenen P. Springer Series in Statistics (1997)
- “Nonmetric multidimensional scaling: a numerical method” Kruskal, J. Psychometrika, 29 (1964)
- “Multidimensional scaling by optimizing goodness of fit to a nonmetric hypothesis” Kruskal, J. Psychometrika, 29, (1964)

2.2.9. t 分布随机邻域嵌入（t-SNE）

t-SNE（ `TSNE` ）将数据点的相似性转换为概率。原始空间中的相似性表示为高斯联合概率，嵌入空间中的相似性表示为 “学生” 的 t 分布。这允许 t-SNE 对局部结构特别敏感，并且有超过现有技术的一些其它优点：

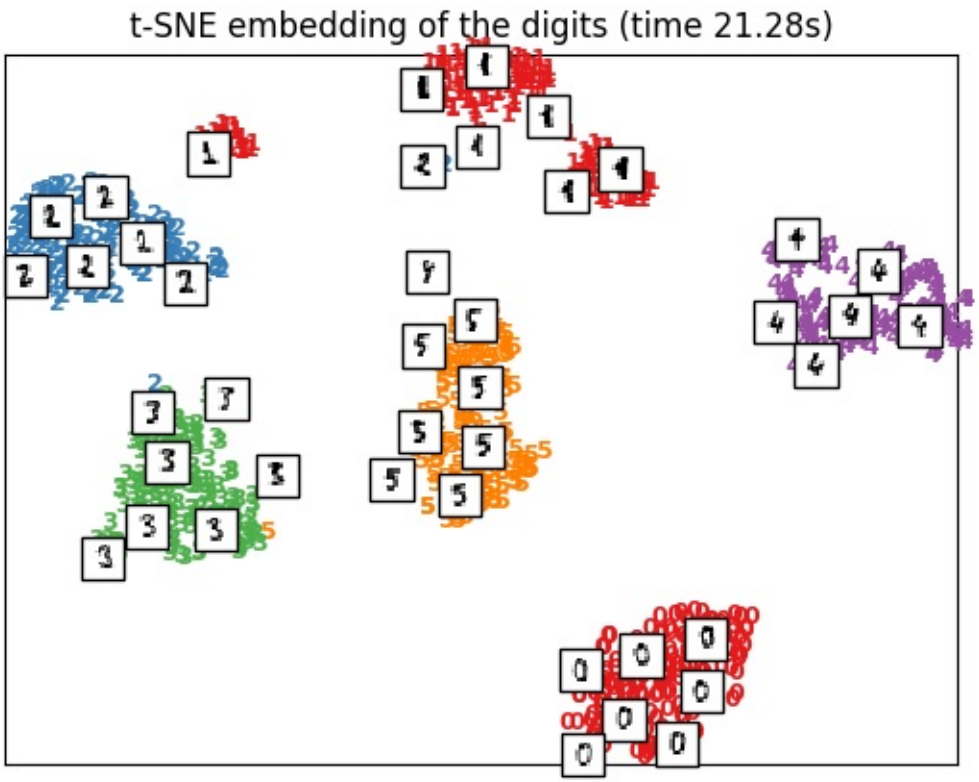
- 在一个单一映射上按多种比例显示结构
- 显示位于多个、不同的流形或聚类中的数据
- 减轻在中心聚集的趋势

Isomap、LLE 和其它变体最适合展开单个连续低维流形，而 t-SNE 将侧重于数据的局部结构，并倾向于提取聚类的局部样本组，就像S曲线示例中突出显示的那样。这种基于局部结构对样本进行分组的能力可能有助于在视觉上同时解开包含多个流形的数据集，如数字数据集中的情况。

原始空间和嵌入空间中的联合概率的 Kullback-Leibler（KL） 散度将通过梯度下降而最小化。注意，KL 发散不是凸的，即具有不同初始化的多次重新开始将以KL发散的局部最小值结束。因此，有时候，通过尝试不同的开始值来选择具有最低KL散度的嵌入是非常有用的。

使用 t - SNE 的缺点大致如下：

- t-SNE 的计算成本很高，在百万样本数据集上可能需要几个小时，而PCA将在几秒或几分钟内完成同样工作。
- Barnes-Hut t-SNE 方法仅限于二维或三维嵌入。
- 该算法是随机的，不同种子的多次重新开始可以产生不同的嵌入。然而，以最小的误差选择嵌入是完全合理的。
- 未明确保留全局结构。用PCA初始化点(使用 *init='pca'*)，可以减轻此问题。



2.2.9.1. 优化 t-SNE

t-SNE 的主要目的是实现高维数据的可视化。因此，当数据将嵌入到二维或三维时，它效果最好。

优化KL发散有时可能有点棘手。有五个参数控制 t-SNE 的优化，因此可能也控制最终嵌入的质量：

- 困惑度
- 早期增长因子
- 学习率
- 最大迭代次数
- 角度（不在精确方法中使用）

困惑度（perplexity 译注:有时也作混乱度）定义为 $k = 2^{(S)}$ ，其中 S 是条件概率分布的香农熵。k 面色子的复杂度是 k ，因此 k 实际上是生成条件概率时 t-SNE 考虑的最近邻域的个数。复杂度越大导致有越多的近邻域，则对小结构越不敏感。相反地，越低的困惑度考虑越少的邻域，并因此忽略越多的全局信息而越关注局部邻域。当数据集的大小变大时，需要更多的点来获得局部邻域的合理样本，因此可能需要更大的困惑度。类似地，噪声越大的数据集需要越大的困惑度来包含足够的局部邻域，以超出背景噪声。

最大迭代次数通常足够高，并不需要任何调整。优化分为两个阶段:早期增长阶段和最终优化阶段。在早期增长阶段，原始空间中的联合概率将通过与给定因子相乘而被人为地增加。越大的因子导致数据中的自然聚类之间的差距越大。如果因子过高，KL 发散可能在此阶段增加。通常不需要对其进行调谐。学习率是一个关键参数。如果梯度太低，下降会陷入局部极小值;如果过高，KL发散将在优化阶段增加。可以在 Laurens van derMaaten 的常见问题解答中找到更多提示(见参考资料)。最后一个参数角度是性能和精度之间的折衷。角度越大意味着我们可以通过单个点来近似的区域越大，从而导致越快的速度，但结果越不准确。

“如何高效使用 t-SNE” 提供了一个关于各种参数效果的讨论，以及用来探索不同参数效果的交互式绘图。

2.2.9.2. Barnes-Hut t-SNE

在此实现的 Barnes-Hut t-SNE 通常比其他流形学习算法慢得多。优化是很困难的，梯度的计算是 $O[dNlog(N)]$ ，其中 d 是输出维数，N 是样本个数。Barnes-Hut 方法在 t-SNE 复杂度为 $O[dN^2]$ 的精确方法之上进行了改进，但有其他几个显著区别：

- Barnes-Hut 仅在目标维度为3或更小时，才有效。构建可视化时，基于2-D的案例是典型的案例。
- Barnes-Hut 仅适用于密集的输入数据。稀疏数据矩阵只能用精确方法嵌入，或者可以通过密集的低阶投影来近似，例如使用 `sklearn.decomposition.TruncatedSVD`
- Barnes-Hut 是精确方法的一种近似。这种近似方法使用 angle 作为参数，因此当参数 method=”exact” 时，angle 参数无效。
- Barnes-Hut 的拓展性很高。Barnes-Hut 可用于嵌入数十万个数据点，而精确方法只能处理数千个样本，再多就很困难了。

出于可视化目的（ t-SNE 的主要使用情况），强烈建议使用 Barnes-Hut 方法。精确的 t-SNE 方法可用于检验高维空间中嵌入的理论性质，但由于计算能力的约束而仅限于小数据集。

同时可以注意到，数字 label 与 t-SNE 发现的自然聚类大致匹配，而 PCA 模型的线性 2D 投影产生了标签区域在很大程度上重叠的表示。这是一个强有力的线索，表明该数据可以通过关注局部结构的非线性方法（例如，具有高斯 RBF 核的 SVM ）很好地分离。然而，如果不能在二维中用 t-SNE 来可视化分离良好的均匀标记组，并不一定意味着数据不能被监督模型正确地分类。可能是因为 2 维还不够低，无法准确表示数据的内部结构。

> 参考资料：
>
> - “Visualizing High-Dimensional Data Using t-SNE” van der Maaten, L.J.P.; Hinton, G. Journal of Machine Learning Research (2008)
> - “t-Distributed Stochastic Neighbor Embedding” van der Maaten, L.J.P.
> - “Accelerating t-SNE using Tree-Based Algorithms.” L.J.P. van der Maaten. Journal of Machine Learning Research 15(Oct):3221-3245, 2014.

2.2.10. 实用技巧

- 确保对所有特征使用相同的缩放。因为流形学习方法是基于最近邻搜索的，否则算法的性能可能很差。有关缩放异构数据的方便方法，请参阅 StandardScaler 。
- 由每个例程计算的重构误差可用于选择最佳输出维度。对于嵌入在 D 维参数空间中的 d 维流形，重构误差将随着 `n_components` 的增加而减小，直到 `n_components == d` 。
- 注意，噪声数据可以对流形造成“短路”，其实质上充当了一个桥梁，用于连接流形的不同部分，否则（没有这样的“桥梁”）这些流形将被很好地划分开。基于噪声和/或不完全数据的流形学习是一个活跃的研究领域。
- 某些输入配置可能导致奇异加权矩阵，例如，当数据集中的两个以上点完全相同时，或者当数据被分割成不连续的组时。在这种情况下， `solver='arpack'` 将无法找到零空间(null space)。解决这一问题的最简单方法是使用 `solver='dense'` ，它将在一个奇异矩阵上进行，尽管它可能因为输入点的数量而非常缓慢。或者，人们可以尝试理解奇异性的来源：如果它是由于不相交的集合，增加 `n_neighbors` 可能有所帮助；如果是由于数据集中的相同点，则删除这些点可能有所帮助。

> See also:完全随机树嵌入 也可以用于得到特征空间的非线性表示，另外它不用降维。

我们一直在努力

apachecn/sklearn-doc-zh



2.3. 聚类

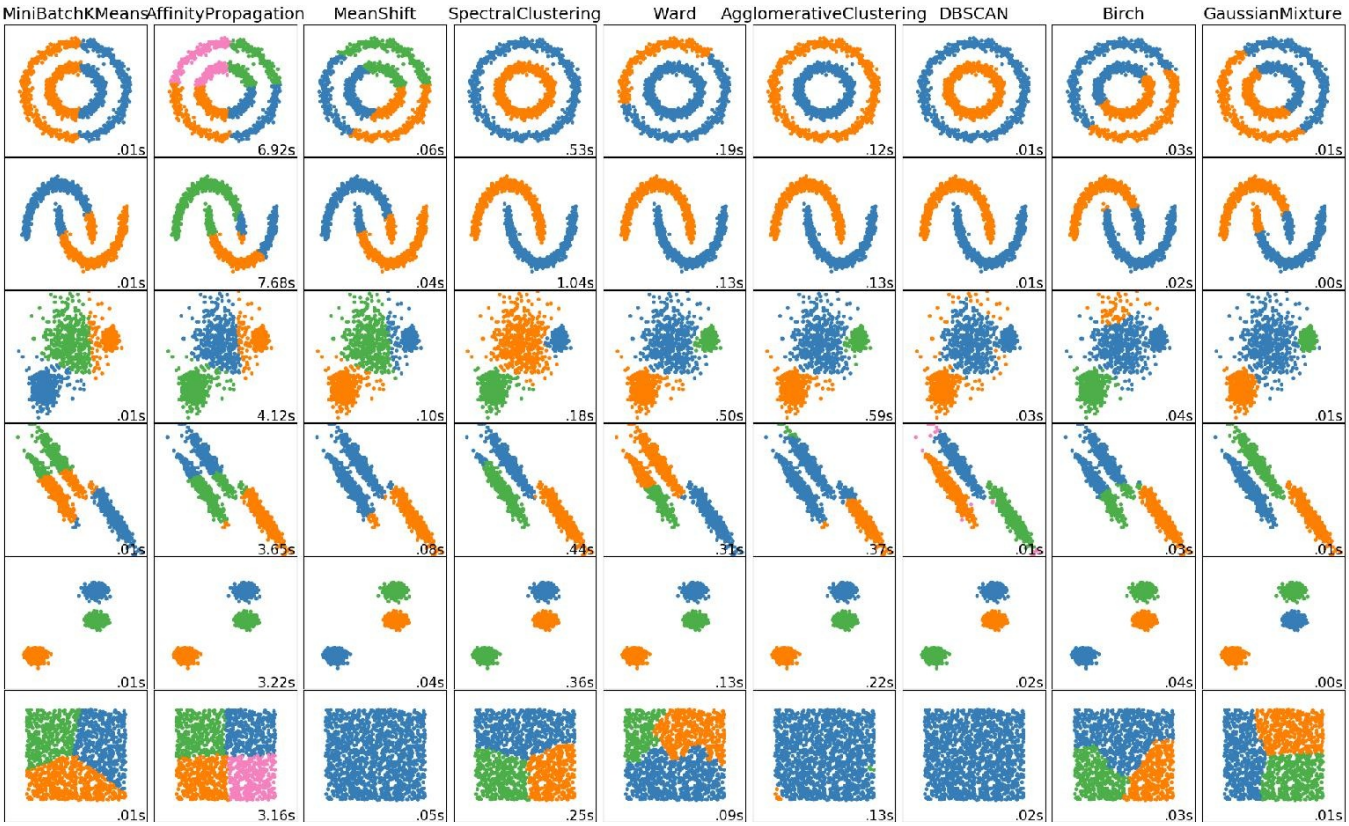
校验者：@花开无声@小瑶@Loopy@barrycg翻译者：@小瑶@krokyin

未标记的数据的 聚类(Clustering) 可以使用模块 `sklearn.cluster` 来实现。

每个聚类算法(clustering algorithm)都有两个变体：一个是 类（class），它实现了 `fit` 方法来学习训练数据的簇（cluster），还有一个 函数（function），当给定训练数据，返回与不同簇对应的整数标签数组（array）。对于类来说，训练数据上的标签可以在 `labels_` 属性中找到。

> 输入数据
>
> 需要注意的一点是，该模块中实现的算法可以采用不同种类的矩阵作为输入。所有算法的调用接口都接受 shape `[n_samples, n_features]` 的标准数据矩阵。这些矩阵可以通过使用 `sklearn.feature_extraction` 模块中的类获得。对于 `AffinityPropagation`，`SpectralClustering` 和 `DBSCAN` 也可以输入 shape `[n_samples, n_samples]` 的相似矩阵。这些可以通过 `sklearn.metrics.pairwise` 模块中的函数获得。

2.3.1. 聚类方法概述



scikit-learn 中的 聚类算法 的比较

Method name（方法名称）	Parameters（参数）	Scalability（可扩展性）	Usecase（使用场景）	Geomet (metr used) (图形（公用）
			通用，均匀的	

K-Means（K-均值）	number of clusters（聚类形成的簇的个数）	非常大的 n_samples，中等的 n_clusters 使用 MiniBatch 代码）	cluster size（簇大小），flat geometry（平面几何），不是太多的 clusters（簇）	Distanc between points（间的距离
Affinity propagation	damping（阻尼），sample preference（样本偏好）	Not scalable with n_samples（n_samples 不可扩展）	Many clusters, uneven cluster size, non-flat geometry（许多簇，不均匀的簇大小，非平面几何）	Graph distanc (e.g. nearest neighbo graph)（离（例如近邻图）
Mean-shift	bandwidth（带宽）	Not scalable with n_samples（n_samples 不可扩展）	Many clusters, uneven cluster size, non-flat geometry（许多簇，不均匀的簇大小，非平面几何）	Distanc between points（间的距离
Spectral clustering	number of clusters（簇的个数）	中等的 n_samples，小的 n_clusters	Few clusters, even cluster size, non-flat geometry（几个簇，均匀的簇大小，非平面几何）	Graph distanc (e.g. nearest neighbo graph)（离（例如邻图））
Ward hierarchical clustering	number of clusters（簇的个数）	大的 n_samples 和 n_clusters	Many clusters, possibly connectivity constraints（很多的簇，可能连接限制）	Distanc between points（间的距离
Agglomerative clustering	number of clusters（簇的个数），linkage type（链接类型），distance（距离）	大的 n_samples 和 n_clusters	Many clusters, possibly connectivity constraints, non Euclidean distances（很多簇，可能连接限制，非欧氏距离）	Any pairwis distanc 意成对距
DBSCAN	neighborhood size（neighborhood 的大小）	非常大的 n_samples，中等的 n_clusters	Non-flat geometry, uneven cluster sizes（非平面几何，不均匀的簇大小）	Distanc between nearest points（点之间的距离）
Gaussian mixtures（高斯混合）	many（很多）	Not scalable（不可扩展）	Flat geometry, good for density estimation（平面几何，适用于密度估计）	Mahalan distanc to centers 中心的马离）
Birch	branching factor（分支因子），threshold（阈值），optional global clusterer（可选全局	大的 n_clusters 和 n_samples	Large dataset, outlier removal, data reduction.（大型数据集，异常值	Euclide distanc between points（间的欧氏距

	簇）.		去除，数据简化）	离）

当簇具有特殊的形状，即非平面流体（译注：即该流体的高斯曲率非*0*），并且标准欧氏距离不是正确的度量标准（metric）时，非平面几何聚类(Non-flat geometry clustering)是非常有用的。这种情况出现在上图的两个顶行中。

用于聚类（clustering）的高斯混合模型（Gaussian mixture models），专用于混合模型描述在 文档的另一章节 。可以将 KMeans 视为具有每个分量的协方差(equal covariance per component)相等的高斯混合模型的特殊情况。

2.3.2. K-means

`KMeans` 算法通过把样本分离成 n 个具有相同方差的类的方式来聚集数据，最小化称为 惯量(inertia) 或 簇内平方和(within-cluster sum-of-squares)的标准（criterion）。该算法需要指定簇的数量。它可以很好地扩展到大量样本(large number of samples)，并已经被广泛应用于许多不同领域的应用领域。

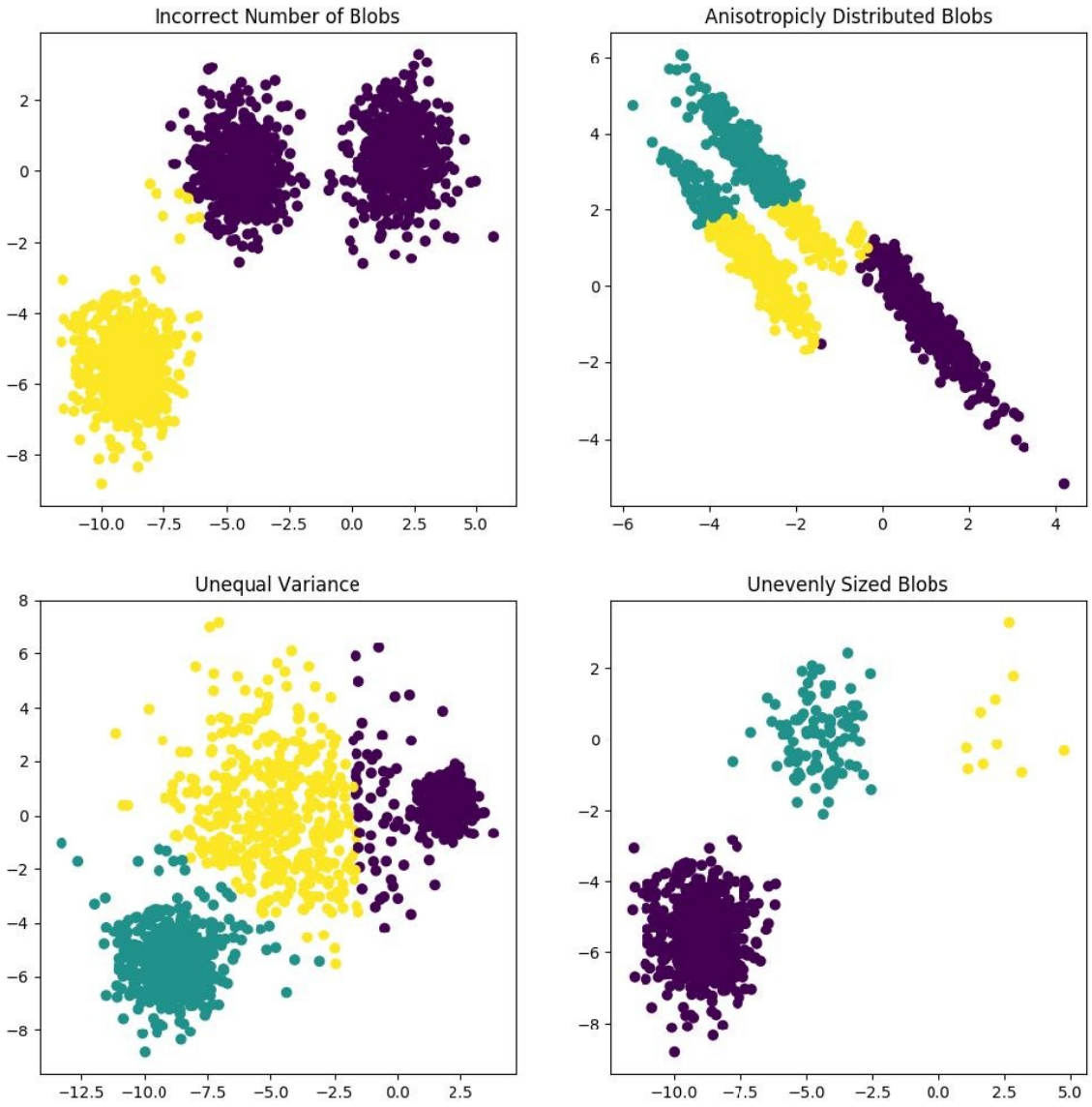
k-means 算法将一组 N 样本 X 划分成 K 不相交的簇 C，每个都用该簇中的样本的均值 μ_j 描述。 这个均值(means)通常被称为簇的 “质心(centroids)”; 注意，它们一般不是从 X 中挑选出的点，虽然它们是处在同一个空间。

K-means（K-均值）算法旨在选择一个质心， 能够最小化惯性或簇内平方和的标准：

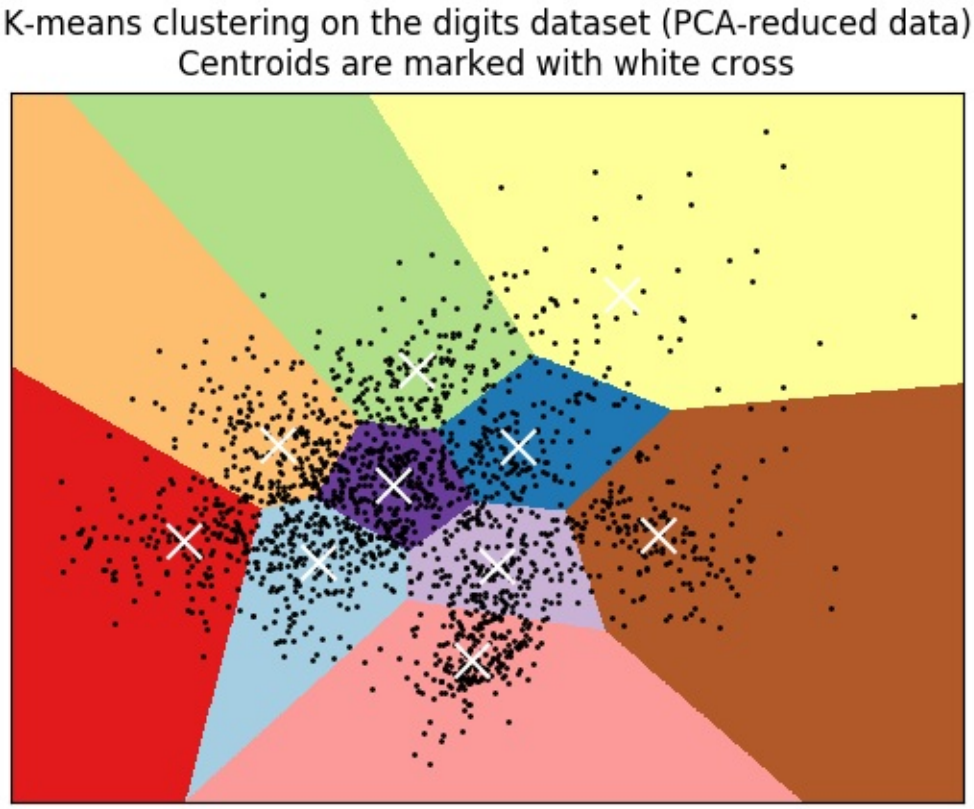
$$\sum_{i=0}^{n} \min_{\mu_j \in C} (||x_j - \mu_i||^2)$$

惯性被认为是测量簇内聚程度的度量(measure)。 它有各种缺点：

- 惯性假设簇是凸(convex)的和各项同性(isotropic)，这并不是总是对的。它对 细长的簇或具有不规则形状的流行反应不佳。
- 惯性不是一个归一化度量(normalized metric): 我们只知道当惯量的值较低是较好的，并且零是最优的。但是在非常高维的空间中，欧氏距离往往会膨胀（这就是所谓的 “维度诅咒/维度惩罚”(curse of dimensionality)）。在 k-means 聚类算法之前运行诸如 PCA 之类的降维算法可以减轻这个问题并加快计算速度。



K-means 通常被称为劳埃德算法(Lloyd's algorithm)。简而言之，该算法可分为三个步骤。第一步是选择初始质心，最基本的方法是从 X 数据集中选择 k 个样本。初始化完成后，K-means 由接下来两个步骤之间的循环组成。 第一步将每个样本分配到其最近的质心。第二步通过取分配给每个先前质心的所有样本的平均值来创建新的质心。计算旧的和新的质心之间的差异，并且算法重复这些最后的两个步骤，直到该值小于阈值。换句话说，算法重复这个步骤，直到质心不再显著移动。



K-means 相当于具有小的全对称协方差矩阵(small, all-equal, diagonal covariance matrix)的期望最大化算法(expectation-maximization algorithm)。

该算法也可以通过 `Voronoi diagrams (Voronoi图)` 的概念来理解。首先,使用当前质心计算点的 Voronoi 图。Voronoi 图中的每个段(segment)都成为一个单独分离的簇。其次，质心被更新为每个段的平均值。然后，该算法重复此操作，直到满足停止条件。 通常情况下，当迭代之间的目标函数(objective function)的相对减小小于给定的公差值(tolerance value)时，算法停止。在此实现中不是这样：当质心移动小于公差时，迭代停止。

给定足够的时间，K-means 将总是收敛的，但这可能是局部最小。这很大程度上取决于质心的初始化。 因此，通常会进行几次初始化不同质心的计算。帮助解决这个问题的一种方法是 k-means++ 初始化方案，它已经在 scikit-learn 中实现（使用 `init='k-means++'` 参数）。 这将初始化质心（通常）彼此远离，导致比随机初始化更好的结果，如参考文献所示。

该算法支持样本权重功能，该功能可以通过参数 `sample_weight` 实现。该功能允许在计算簇心和惯性值的过程中，给部分样本分配更多的权重。 例如，给某个样本分配一个权重值2，相当于在dataset X 中增加一个该样本的拷贝。

存在一个参数，以允许 K-means 并行运行，称为 `n_jobs` 。给这个参数赋予一个正值指定使用处理器的数量（默认值：1）。值 -1 使用所有可用的处理器，-2 使用全部可用处理器减一，等等。并行化(Parallelization)通常以内存的代价(cost of memory)加速计算（在这种情况下，需要存储多个质心副本，每个作业(job)使用一个副本）。

> 警告
>
> K-Means 的并行版本不能在 OS X 上运行，当内部的 *numpy* 使用 *Accelerate* 框架。这是预期的行为：*Accelerate* 可以在 fork 之后调用，但是您需要使用 Python binary 来执行子进程（posix 并不支持多进程处理 ）。

K-means 可用于矢量量化(vector quantization)。这是使用以下类型的训练模型的变换方法实现的 `KMeans` 。

示例：

- Demonstration of k-means assumptions: 演示 k-means 是否直观地执行(performs intuitively)，何时不执行
- A demo of K-Means clustering on the handwritten digits data: 聚类手写数字

参考资料：

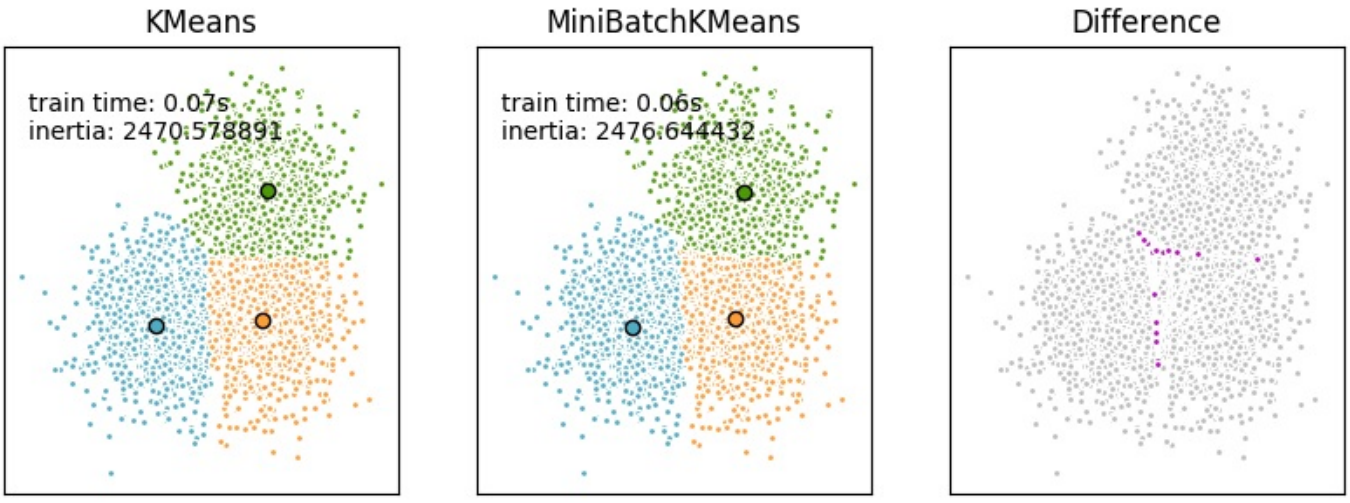
- “k-means++: The advantages of careful seeding” Arthur, David, and Sergei Vassilvitskii, *Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*, Society for Industrial and Applied Mathematics (2007)

2.3.2.1. 小批量 K-Means

`MiniBatchKMeans` 是 `KMeans` 算法的一个变种，它使用小批量(mini-batches)来减少计算时间，而这多个批次仍然尝试优化相同的目标函数。小批量是输入数据的子集，在每次训练迭代中随机抽样。这些小批量大大减少了收敛到局部解所需的计算量。 与其他降低 k-means 收敛时间的算法不同，小批量 k-means 产生的结果通常只比标准算法略差。

该算法在两个步骤之间进行迭代，类似于 vanilla k-means 。在第一步， b 样本是从数据集中随机抽取的，形成一个小批量。然后将它们分配到最近的质心。 在第二步，质心被更新。与 k-means 不同， 该变种算法是基于每个样本(per-sample)。对于小批量中的每个样本，通过取样本的流平均值(streaming average)和分配给该质心的所有先前样本来更新分配的质心。 这具有随时间降低质心的变化率(rate of change)的效果。执行这些步骤直到达到收敛或达到预定次数的迭代。

`MiniBatchKMeans` 收敛速度比 `KMeans` 快 ，但是结果的质量会降低。在实践中，质量差异可能相当小，如示例和引用的参考。



示例：

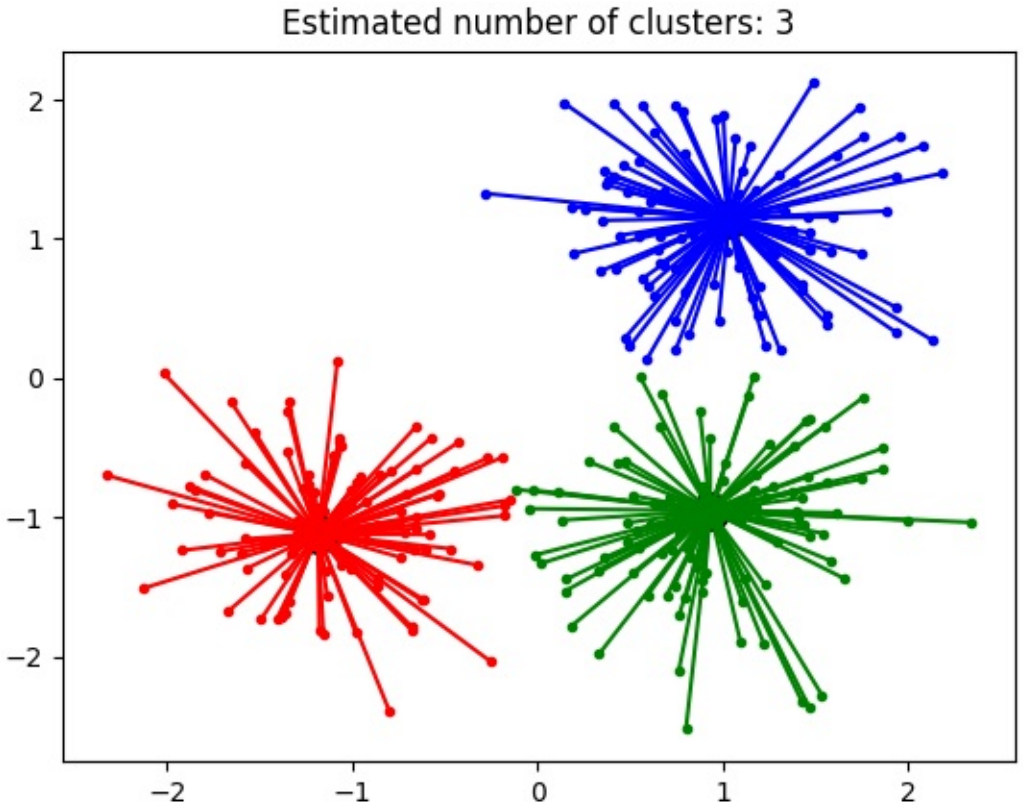
- Comparison of the K-Means and MiniBatchKMeans clustering algorithms: KMeans 与 MiniBatchKMeans 的比较
- Clustering text documents using k-means: 使用 稀疏 MiniBatchKMeans的文档聚类
- Online learning of a dictionary of parts of faces

参考资料：

- “Web Scale K-Means clustering” D. Sculley, *Proceedings of the 19th international conference on World wide web* (2010)

2.3.3. Affinity Propagation

`AffinityPropagation` AP聚类是通过在样本对之间发送消息直到收敛的方式来创建聚类。然后使用少量模范样本作为聚类中心来描述数据集，而这些模范样本可以被认为是最能代表数据集中其它数据的样本。在样本对之间发送的消息表示一个样本作为另一个样本的模范样本的 适合程度，适合程度值在根据通信的反馈不断更新。更新迭代直到收敛，完成聚类中心的选取，因此也给出了最终聚类。



Affinity Propagation 算法比较有趣的点是可以根据提供的数据决定聚类的数目。 因此有两个比较重要的参数*preference*, 决定使用多少个模范样本 阻尼因子*(damping factor)* 减少吸引信息和归属信息以减少更新这些信息时的数据振荡。

AP聚类算法主要的缺点是算法的复杂度．AP聚类算法的时间复杂度是 $O(N^2T)$，其中 N 是样本的个数 ， T 是收敛之前迭代的次数．如果使用密集的相似性矩阵空间复杂度是 $O(N^2)$ 如果使用稀疏的相似性矩阵空间复杂度可以降低。 这使得AP聚类最适合中小型数据集。

示例：

- Demo of affinity propagation clustering algorithm: Affinity Propagation on a synthetic 2D datasets with 3 classes.
- Visualizing the stock market structure Affinity Propagation on Financial time series to find groups of companies

Algorithm description(算法描述): 样本之间传递的信息有两种。 第一种是吸引信息 (responsibility) $r(i,k)$, 样本 k 适合作为样本 i 的聚类中心(译注:原文中*exemplar*,即前面所示的模范样本)的程度。

第二种是 归属信息(availability) $a(i,k)$, 样本 i 选择样本 k 作为聚类中心的适合程度,并且考虑其他所有样本选取 k 做为聚类中心的合适程度。 通过这个方法，选取示例样本作为聚类中心， 如果 (1) 该样本与其许多样本相似，并且 (2) 被许多样本选取为可以代表它们自身的模范样本。

样本 k 对样本 i 吸引度计算公式:

$$r(i,k) \leftarrow s(i,k) - max[a(i,k') + s(i,k') \forall k' \neq k]$$

其中 $s(i,k)$ 是样本 i 和样本 k 之间的相似度。样本 k 作为样本 i 的示例样本的合适程度:

$$a(i,k) \leftarrow min[0, r(k,k) + \sum_{i' \ s.t. \ i' \notin \{i,k\}} r(i',k)]$$

算法开始时 r 和 a 都被置 0,然后开始迭代计算直到收敛。 为了防止更新数据时出现数据振荡，在迭代过程中引入阻尼因子 λ :

$$r_{t+1}(i,k) = \lambda \cdot r_t(i,k) + (1-\lambda) \cdot r_{t+1}(i,k)$$

$$a_{t+1}(i,k) = \lambda \cdot a_t(i,k) + (1-\lambda) \cdot a_{t+1}(i,k)$$

其中 t是迭代的次数。

2.3.4. Mean Shift

`MeanShift` 算法旨在于发现一个样本密度平滑的 *blobs* 。均值漂移(Mean Shift)算法是基于质心的算法，通过更新质心的候选位置，这些侯选位置通常是所选定区域内点的均值。然后，这些候选位置在后处理阶段被过滤以消除近似重复，从而形成最终质心集合。

给定第 t 次迭代中的候选质心 x_i ， 候选质心的位置将被按照如下公式更新:

$$x_i^{t+1} = x_i^t + m(x_i^t)$$

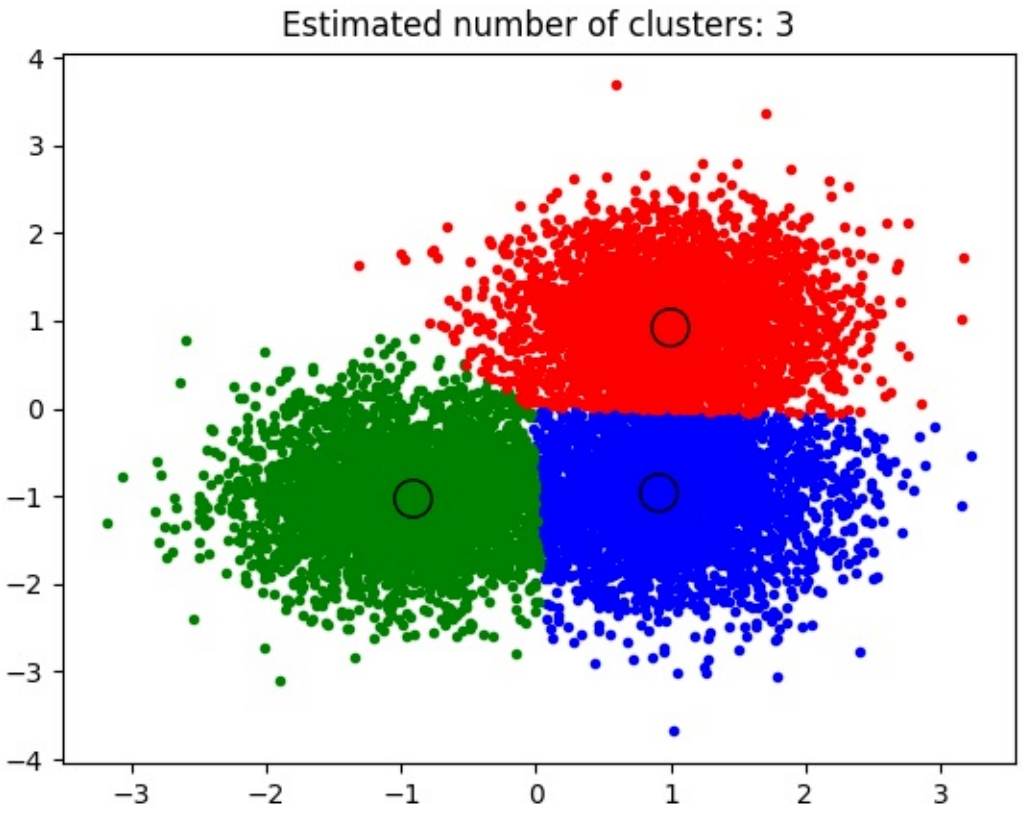
其中 $N(x_i)$ 是围绕 x_i 周围一个给定距离范围内的样本邻域， m 是均值偏移向量(mean shift vector)， 该向量是所有质心中指向 点密度增加最多的区域的偏移向量。使用以下等式计算，有效地将质心更新为其邻域内样本的平均值:

$$m(x_i) = \frac{\sum_{x_j \in N(x_i)} K(x_j - x_i) x_j}{\sum_{x_j \in N(x_i)} K(x_j - x_i)}$$

算法自动设定聚类的数目，而不是依赖参数 `带宽（bandwidth）`，带宽是决定搜索区域的size的参数。 这个参数可以手动设置，但是如果没有设置，可以使用提供的函数 `estimate_bandwidth` 获取 一个估算值。

该算法不是高度可扩展的，因为在执行算法期间需要执行多个最近邻搜索。 该算法保证收敛，但是当 质心的变化较小时，算法将停止迭代。

通过找到给定样本的最近质心来给新样本打上标签。



示例：

- A demo of the mean-shift clustering algorithm: Mean Shift clustering on a synthetic 2D datasets with 3 classes.

参考资料：

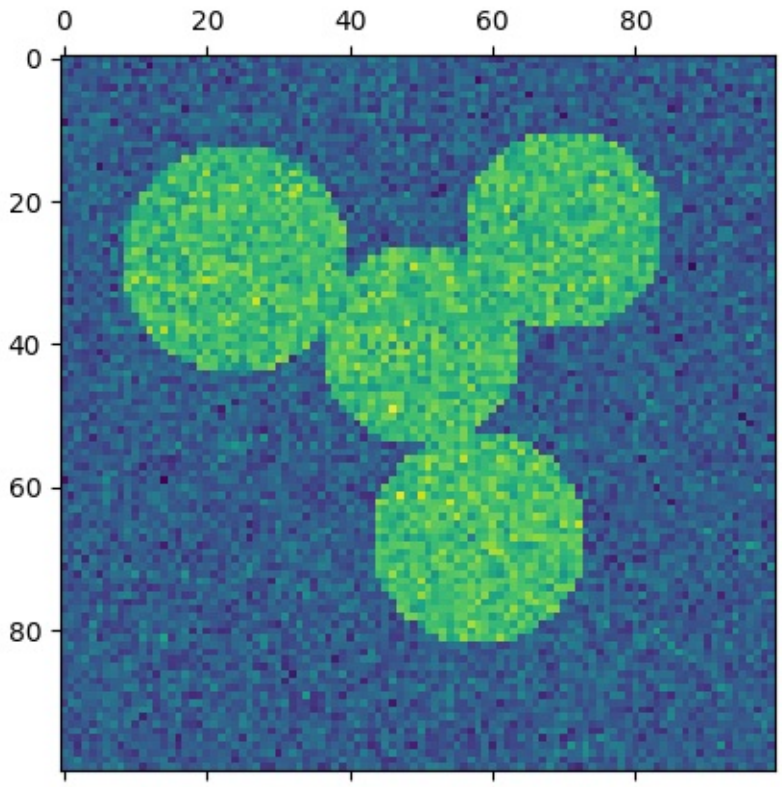
- “Mean shift: A robust approach toward feature space analysis.” D. Comaniciu and P. Meer, *IEEE Transactions on Pattern Analysis and Machine Intelligence* (2002)

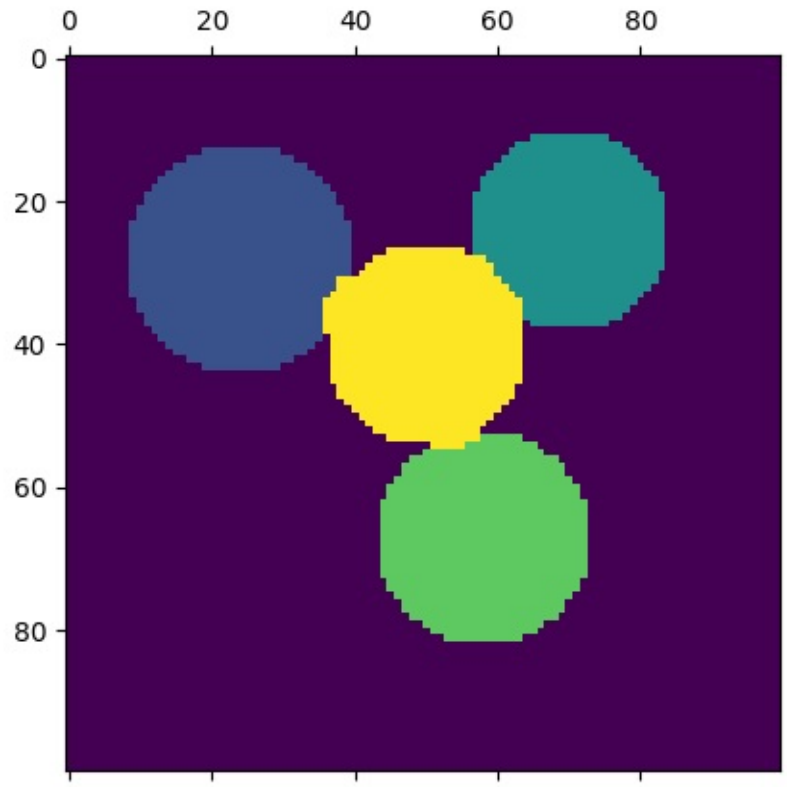
2.3.5. Spectral clustering

`SpectralClustering(谱聚类)` 是在样本之间进行关联矩阵的低维度嵌入，然后在低维空间中使用 KMeans 算法。 如果关联矩阵稀疏并且 pyamg 模块已经被安装，则这是非常有效的。 谱聚类 需要指定簇的数量。这个算法适用于簇

数量少时，在簇数量多时是不建议使用。

对于两个簇，它解决了相似图形上的 归一化切割(normalised cuts)的凸松弛问题：将图形切割成两个，使得切割的边缘的权重比每个簇内的边缘的权重小。在图像处理时，这个标准是特别有趣的：图像的顶点是像素，相似图形的边缘是图像的渐变函数。





警告:将距离转换为良好的相似性

请注意，如果你的相似矩阵的值分布不均匀，例如:存在负值或者距离矩阵并不表示相似性， spectral problem 将会变得奇异，并且不能解决。 在这种情况下，建议对矩阵的 entries 进行转换。比如在有向距离矩阵上应用 heat kernel:

```
similarity = np.exp(-beta * distance / distance.std())
```

请看这样一个应用程序的例子。

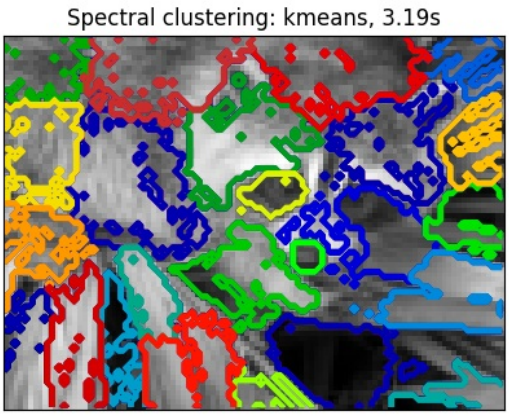
示例:

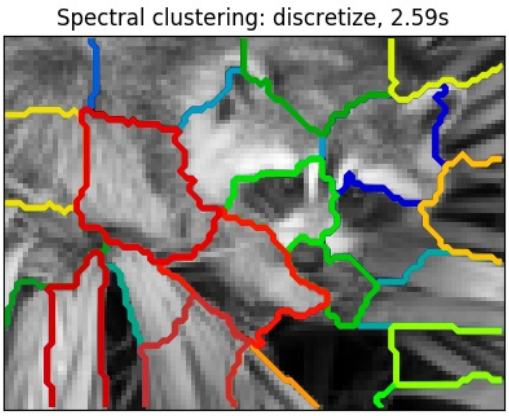
- Spectral clustering for image segmentation: Segmenting objects from a noisy background using spectral clustering.
- Segmenting the picture of a raccoon face in regions: Spectral clustering to split the image of the raccoon face in regions.

2.3.5.1. 不同的标记分配策略

`SpectralClustering` 的 `assign_labels` 参数代表着可以使用不同的分配策略。 `"kmeans"` 可以匹配更精细的数据细节，但是可能更加不稳定。 特别是，除非你可以控制 `random_state` 否则可能无法复现运行的结果 ，因为它取决于随机初始化。另一方， 使用 `"discretize"` 策略是 100% 可以复现的，但是它往往会产生相当均匀的几何形状的闭合块。

assign_labels="kmeans"	assign_labels="discretize"





2.3.5.2. 谱聚类用于图聚类问题

谱聚类还可以通过谱嵌入对图进行聚类。在这种情况下，关联矩阵(affinity matrix) 是图形的邻接矩阵，谱聚类可以由 affinity='precomputed' 进行初始化。

```
>>> from sklearn.cluster import SpectralClustering
>>> sc = SpectralClustering(3, affinity='precomputed', n_init=100,
...                         assign_labels='discretize')
>>> sc.fit_predict(adjacency_matrix)
```

参考资料：

- "A Tutorial on Spectral Clustering" Ulrike von Luxburg, 2007
- "Normalized cuts and image segmentation" Jianbo Shi, Jitendra Malik, 2000
- "A Random Walks View of Spectral Segmentation" Marina Meila, Jianbo Shi, 2001
- "On Spectral Clustering: Analysis and an algorithm" Andrew Y. Ng, Michael I. Jordan, Yair Weiss, 2001

2.3.6. 层次聚类

层次聚类(Hierarchical clustering)代表着一类的聚类算法，这种类别的算法通过不断的合并或者分割内置聚类来构建最终聚类。 聚类的层次可以被表示成树（或者树形图(dendrogram)）。树根是拥有所有样本的唯一聚类，叶子是仅有一个样本的聚类。 请参照 Wikipedia page 查看更多细节。

The `AgglomerativeClustering` 使用自下而上的方法进行层次聚类：开始是每一个对象是一个聚类， 并且聚类别相继合并在一起。 连接标准(linkage criteria 译注: 该词似乎尚未有公认的翻译标准) 决定用于合并策略的度量：

- **Ward** 最小化所有聚类内的平方差总和。这是一种方差最小化(variance-minimizing)的优化方向， 这是与k-means 的目标函数相似的优化方法，但是用 凝聚分层（agglomerative hierarchical）的方法处理。
- **Maximum** 或 **complete linkage** 最小化成对聚类间最远样本距离。
- **Average linkage** 最小化成对聚类间平均样本距离值。

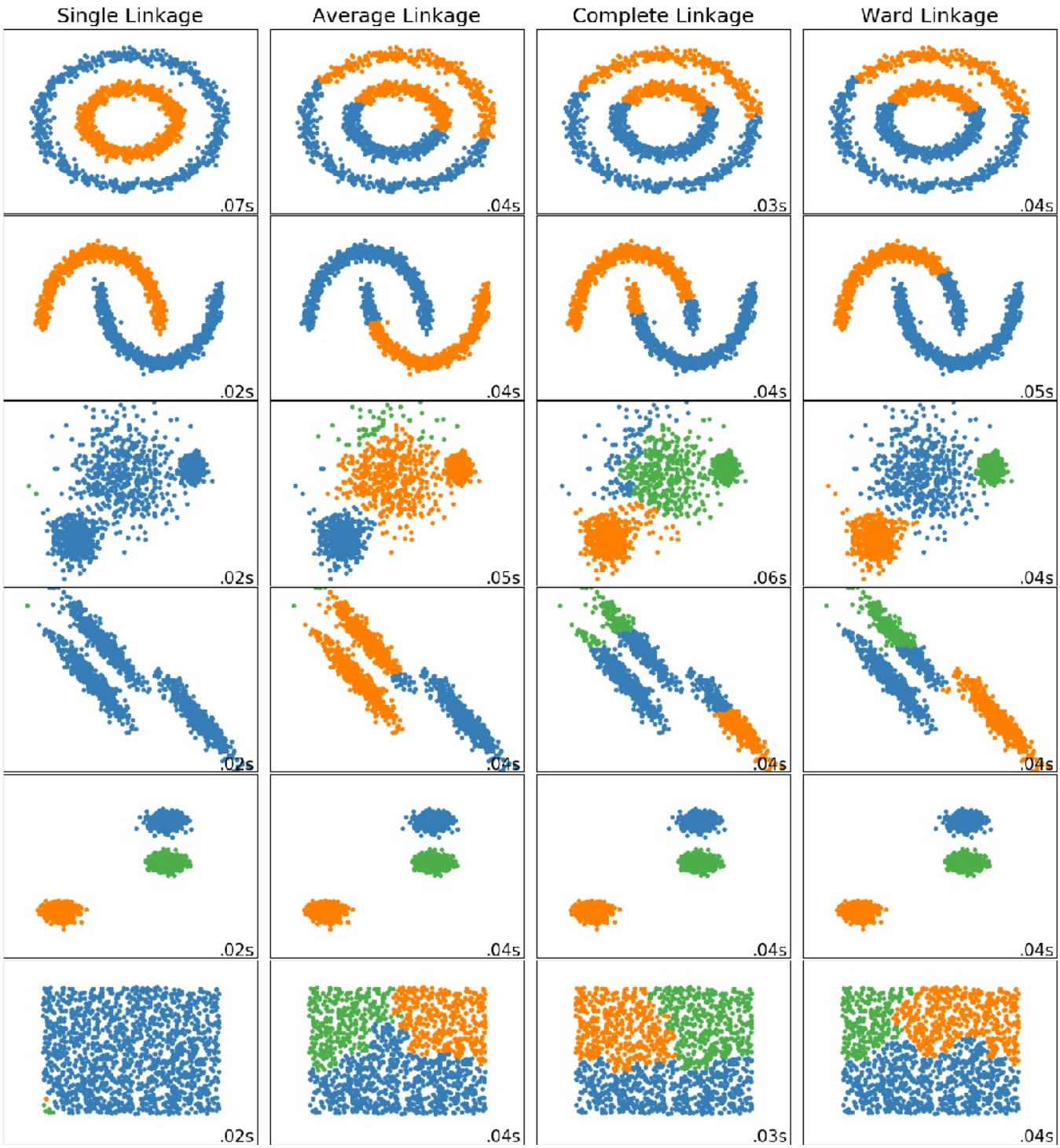
- **Single linkage** 最小化成对聚类间最近样本距离值。

`AgglomerativeClustering` 在联合使用同一个连接矩阵(connectivity matrix)时，也可以扩大到大量的样本，但是在样本之间没有添加连接约束时，计算代价很大:每步都要考虑所有可能的合并。

> **FeatureAgglomeration**
>
> `FeatureAgglomeration` 使用 凝聚聚类(agglomerative clustering) 将看上去相似的 特征组合在一起，从而减少特征的数量。这是一个降维工具，请参照 无监督降维。

2.3.6.1. 不同连接类型: Ward, complete,average and single linkage

`AgglomerativeClustering` 支持 Ward, single, average, and complete linkage 策略.



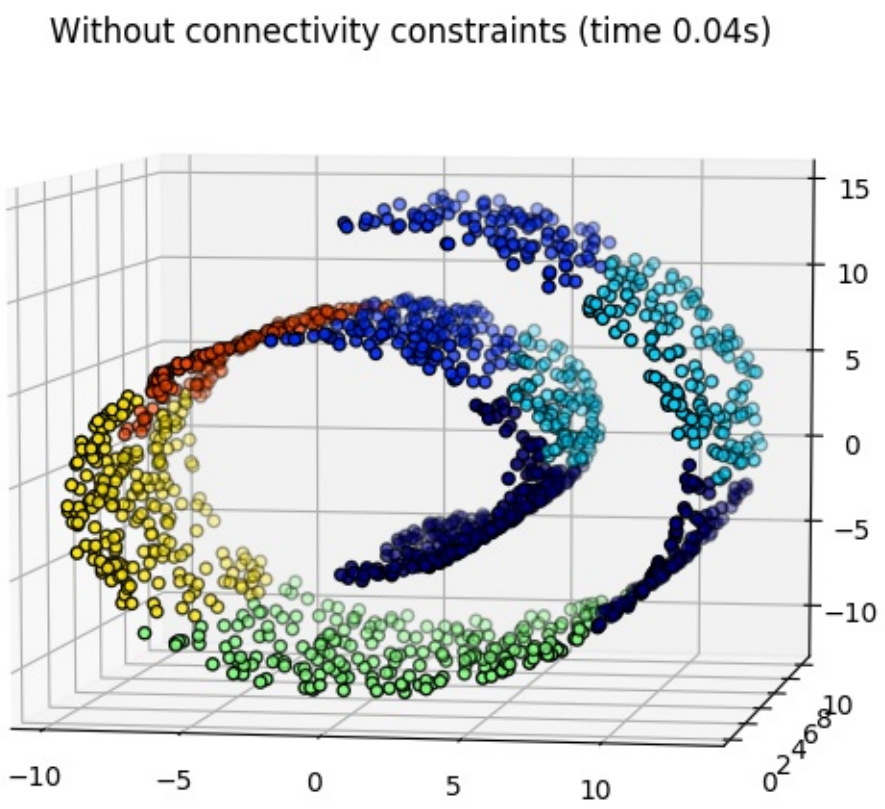
Agglomerative cluster 存在“rich get richer”现象导致聚类大小不均匀。这方面 Single linkage 是最坏的策略，Ward 给出了最规则的大小。然而，在 Ward 中 affinity (or distance used in clustering) 不能被改变，对于 非欧氏度量(non Euclidean metrics) 来说，average linkage 是一个好的选择。Single linkage 虽然对于噪声数据的鲁棒性并不强，但是对规模较大的dataset提供非常有效的层次聚类算法。 Single linkage 同样对非全局数据有很好的效果。

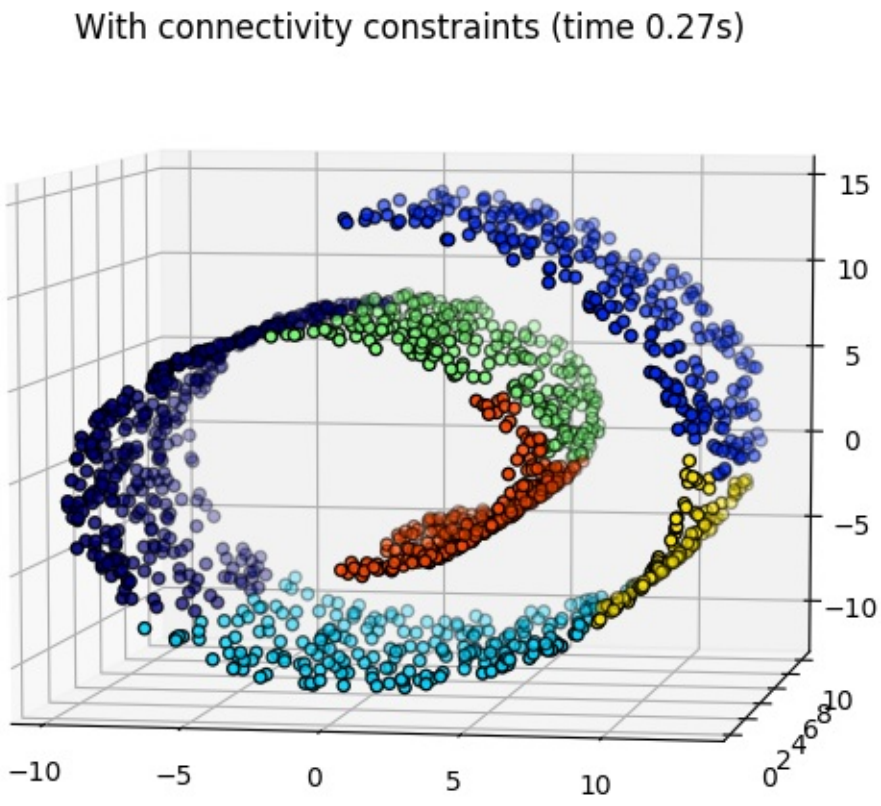
示例：

- Various Agglomerative Clustering on a 2D embedding of digits: exploration of the different linkage strategies in a real dataset.

2.3.6.2. 添加连接约束

`AgglomerativeClustering` 中一个有趣的特点是可以使用连接矩阵(connectivity matrix)将连接约束添加到算法中（只有相邻的聚类可以合并到一起），连接矩阵为每一个样本给定了相邻的样本。 例如，在 swiss-roll 的例子中，连接约束禁止在不相邻的 swiss roll 上合并，从而防止形成在 roll 上重复折叠的聚类。

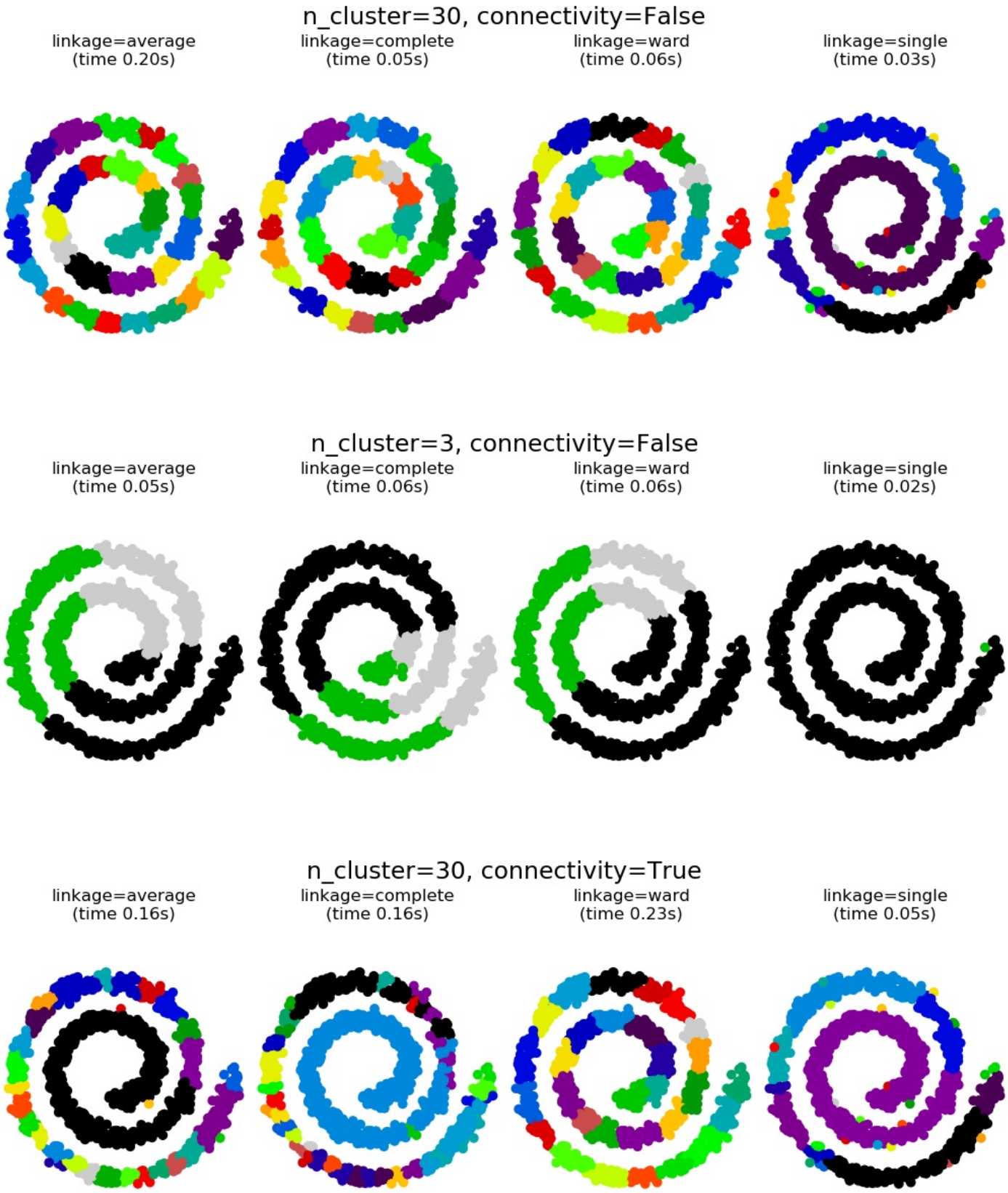


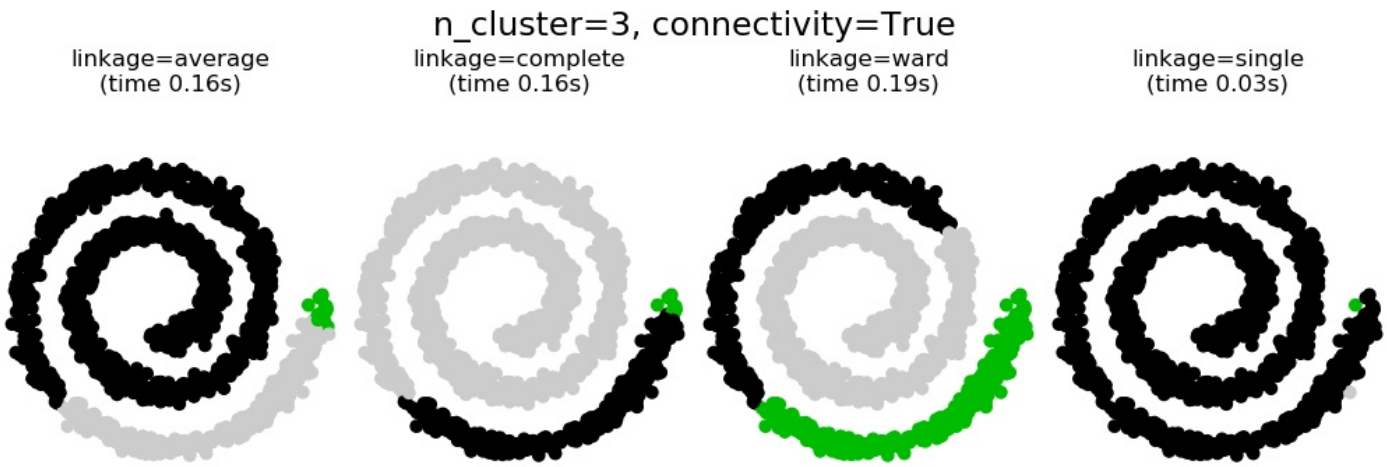


这些约束对塑造局部结构是很有用的，而且这也使得算法更快，特别是当样本数量巨大时。

连通性的限制(connectivity constraint)是通过连接矩阵来实现的: scipy sparse matrix 仅仅在行列的交界处有元素存在，而这些行列记录相连接的数据集的索引。这个矩阵可以通过先验信息构建:例如，你可能通过仅仅将从一个连接指向另一个的链接合并页面来聚类页面。也可以从数据中学习到，例如使用 `sklearn.neighbors.kneighbors_graph` 限制与最近邻的合并， 就像 这个例子 中那样，或者使用 `sklearn.feature_extraction.image.grid_to_graph` 仅合并图像上相邻的像素点， 就像 这个 coin 例子 。

> 示例:
>
> - A demo of structured Ward hierarchical clustering on a raccoon face image: Ward clustering to split the image of a raccoon face in regions.
> - Hierarchical clustering: structured vs unstructured ward: Example of Ward algorithm on a swiss-roll, comparison of structured approaches versus unstructured approaches.
> - Feature agglomeration vs. univariate selection: Example of dimensionality reduction with feature agglomeration based on Ward hierarchical clustering.
> - Agglomerative clustering with and without structure
>
> 警告: **single, average and complete linkage**的连接约束
>
> single, average and complete linkage的连接约束可以增强聚合聚类中的 ‘rich getting richer’ 现象，特别是当它们使用函数 `sklearn.neighbors.kneighbors_graph` 进行构建时。 在少量聚类的限制中，更倾向于给出一些 macroscopically occupied clusters 并且几乎是空的 (讨论内容请查看 Agglomerative clustering with and without structure)。在这个问题上，single是最脆弱的 linkage 选项。


n_cluster=30, connectivity=False
linkage=average
(time 0.20s)
linkage=complete
(time 0.05s)
linkage=ward
(time 0.06s)
linkage=single
(time 0.03s)
n_cluster=3, connectivity=False
linkage=average
(time 0.05s)
linkage=complete
(time 0.06s)
linkage=ward
(time 0.06s)
linkage=single
(time 0.02s)
n_cluster=30, connectivity=True
linkage=average
(time 0.16s)
linkage=complete
(time 0.16s)
linkage=ward
(time 0.23s)
linkage=single
(time 0.05s)



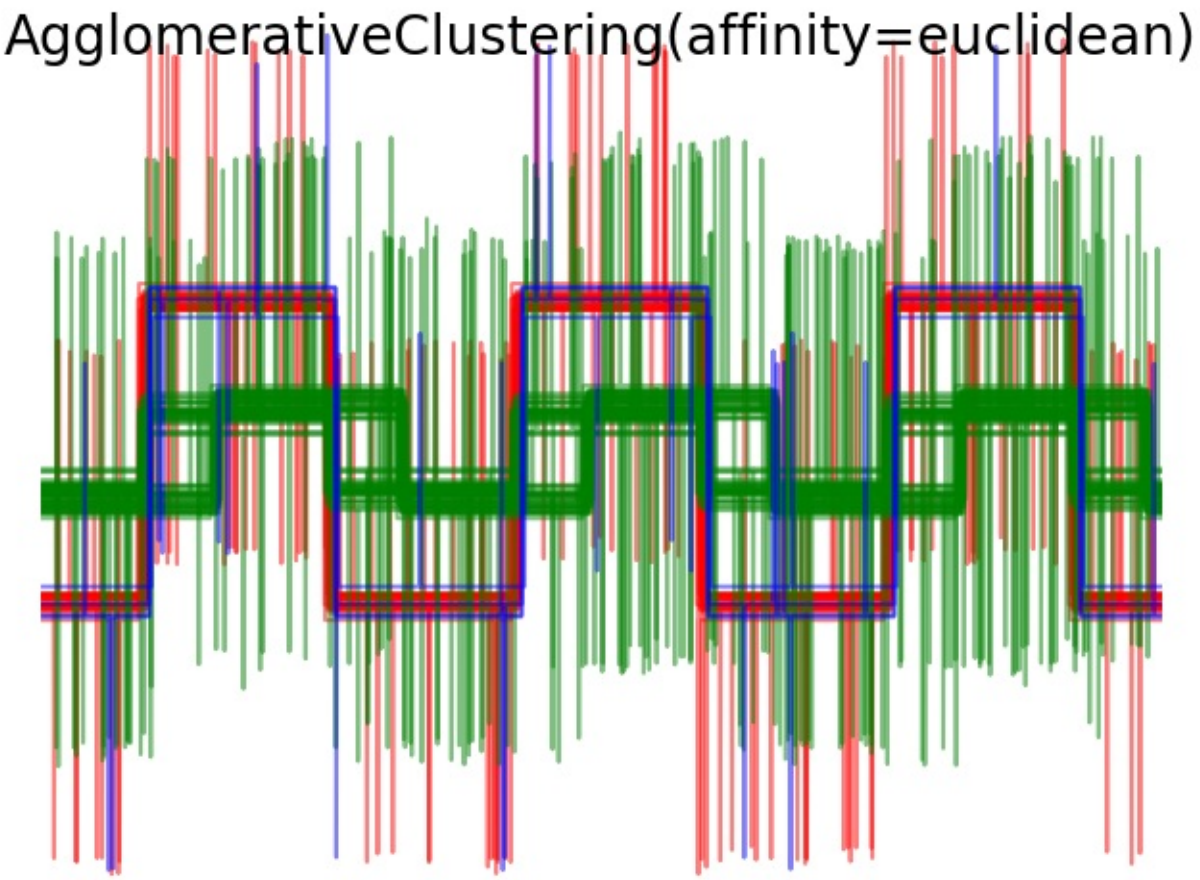
2.3.6.3. Varying the metric

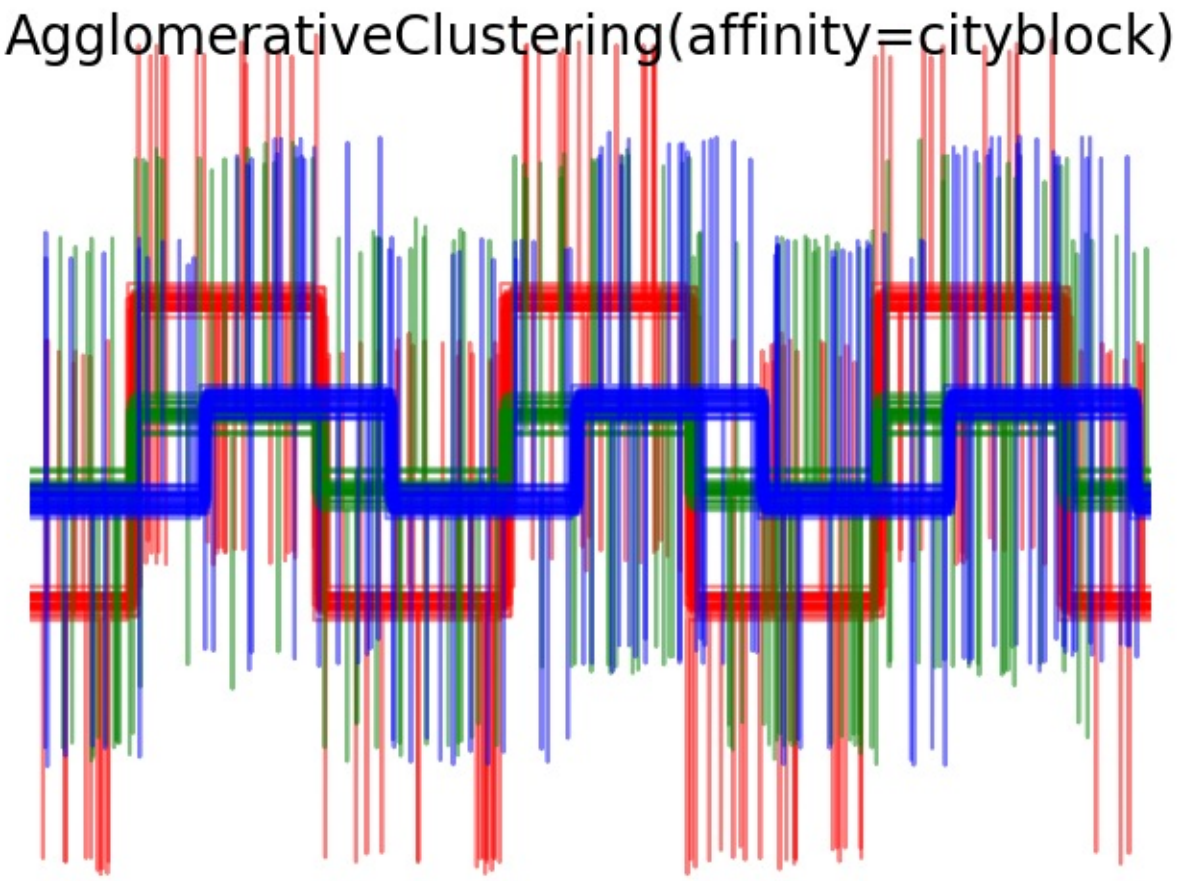
Single, verage and complete linkage 可以使用各种距离 (or affinities), 特别是欧氏距离 (Euclidean distance *l2* 距离), 曼哈顿距离 (Manhattan distance) (or 城市区块距离(Cityblock), or *l1* 距离), 余弦距离(cosine distance), 或者任何预先计算的关联矩阵(affinity matrix).

- *l1* 距离有利于稀疏特征或者稀疏噪声: 例如很多特征都是0，就像在文本挖掘中使用 rare words 一样。
- 余弦 距离是非常有趣的，因为它对全局放缩是一样的。

选择度量标准的方针是使得不同类样本之间距离最大化，并且最小化同类样本之间的距离。







示例：

- Agglomerative clustering with different metrics

2.3.7. DBSCAN

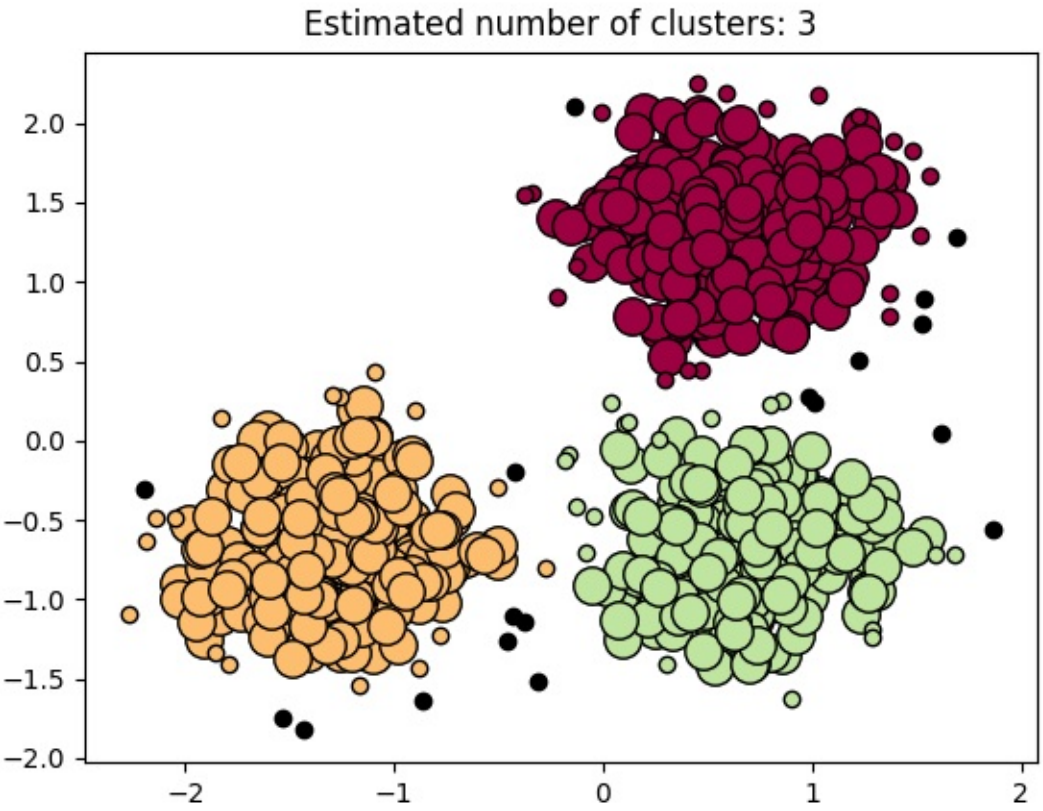
The `DBSCAN` 算法将簇视为被低密度区域分隔的高密度区域。由于这个相当普遍的观点， DBSCAN发现的簇可以是任何形状的，与假设簇是凸的 K-means 相反。 DBSCAN 的核心概念是 *core samples*，是指位于高密度区域的样本。 因此一个簇是一组核心样本，每个核心样本彼此靠近（通过某个距离度量测量） 和一组接近核心样本的非核心样本（但本身不是核心样本）。算法中的两个参数， `min_samples` 和 `eps` ，正式的定义了我们所说的 稠密（dense）_。较高的 `min_samples` 或者较低的 `eps` 都表示形成簇所需的较高密度。

更正式的，我们定义核心样本是指数据集中的一个样本的 `eps` 距离范围内，存在 `min_samples` 个其他样本，这些样本被定为为核心样本的邻居(neighbors) 。这告诉我们，核心样本在向量空间的稠密区域。一个簇是一个核心样本的集合，可以通过递归来构建，选取一个核心样本，查找它所有的邻居样本中的核心样本，然后查找新获取的核心样本的邻居样本中的核心样本，递归这个过程。 簇中还具有一组非核心样本，它们是簇中核心样本的邻居的样本，但本身并不是核心样本。 显然，这些样本位于簇的边缘。

根据定义，任何核心样本都是簇的一部分，任何不是核心样本并且和任意一个核心样本距离都大于 `eps` 的样本将被视为异常值。

当参数 `min_samples` 主要表示算法对噪声的容忍度(当处理大型噪声数据集时， 需要考虑增加该参数的值)， 针对具体地数据集和距离函数，参数 `eps` 如何进行合适地取值是非常关键，这通常不能使用默认值。参数 `eps` 控制了点地领域范围。如果取值太小,大部分地数据并不会被聚类(被标注为 `-1` 代表噪声)； 如果取值太大，可能会 导致相近 的多个簇被合并成一个,甚至整个数据集都被分配到一个簇。一些启发式（heuristics）参数选择方法已经在一些文献上讨论过了，例如，在最近邻距离图种，基于Knee的参数选择方式（在下面引用的资料里已讨论过了）.

在下图中，颜色表示簇成员属性，大圆圈表示算法发现的核心样本。较小的圈子表示仍然是簇的一部分的非核心样本。 此外，异常值由下面的黑点表示。



示例：

- Demo of DBSCAN clustering algorithm

实现

DBSCAN 算法具有确定性的，当以相同的顺序给出相同的数据时总是形成相同的簇。 然而，当以不同的顺序提供数据时，聚类的结果可能不相同。首先，即使核心样本总是被分配给相同的簇，这些簇的标签将取决于数据中遇到这些样本的顺序。第二个更重要的是，非核心样本的簇可能因数据顺序而有所不同。当一个非核心样本距离两个核心样本的距离都小于 `eps` 时，就会发生这种情况。 通过三角不等式可知，这两个核心样本距离一定大于 `eps` 或者处于同一个簇中。 非核心样本将被分配到首先查找到该样本的簇，因此结果也将取决于数据的顺序。

当前版本使用 ball trees 和 kd-trees 来确定领域，这样避免了计算全部的距离矩阵 （0.14 之前的 scikit-learn 版本计算全部的距离矩阵）。保留使用 自定义指标(custom metrics)的可能性。更多细节请参照 `NearestNeighbors` 。

在大规模样本上运行时的内存消耗

默认的实现方式并没有充分利用内存，因为在不使用 kd-trees 或者 ball-trees 的情况下构建一个完整的相似度矩阵（e.g. 使用稀疏矩阵）。这个矩阵将消耗 n^2 个浮点数。 解决这个问题的几种机制：

- 使用 OPTICS 聚类算法联合 `extract_dbscan` 方式。 OPTICS 聚类算法同样需要计算全结对矩阵(full pairwise matrix)，但每次仅需要保持一行在内存中(内存复杂度为 n)。
- 稀疏半径邻域图(A sparse radius neighborhood graph)(其中缺少的样本被假定为距离超出eps) 可以以高效的方式预先编译，并且可以使用 `metric='precomputed'` 来运行 dbscan。
- 数据可以压缩，当数据中存在准确的重复时，可以删除这些重复的数据，或者使用BIRCH。 任何。然后仅需要使用相对少量的样本来表示大量的点。当训练DBSCAN时，可以提供一个 `sample_weight` 参数。

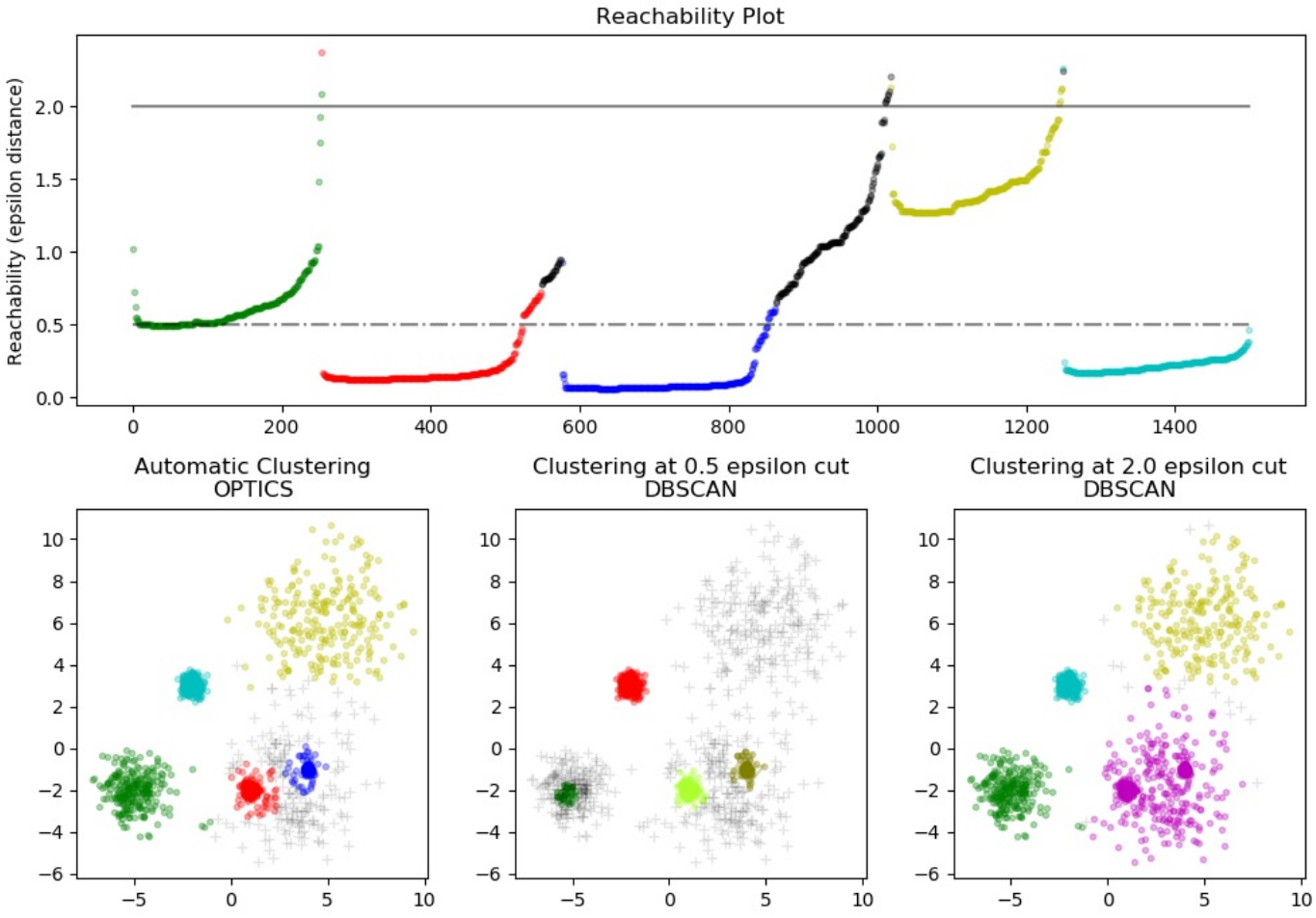
引用：

- “A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise” Ester, M., H. P. Kriegel, J. Sander, and X. Xu, In Proceedings of the 2nd International Conference on Knowledge Discovery and Data Mining, Portland, OR, AAAI Press, pp. 226–231. 1996
- “DBSCAN revisited, revisited: why and how you should (still) use DBSCAN. Schubert, E., Sander, J., Ester,

> M., Kriegel, H. P., & Xu, X. (2017). In ACM Transactions on Database Systems (TODS), 42(3), 19.

2.3.8. OPTICS

OPTICS算法与DBSCAN算法有许多相似之处，可以认为是DBSCAN算法将 `eps` 要求从一个值放宽到一个值范围的推广。OPTICS与DBSCAN的关键区别在于OPTICS算法建立了一个可达性图，它为每个样本分配了一个 `reachability` (可达性距离)和一个簇 `ordering` 属性内的点(spot)；这两个属性是在模型拟合时分配的，用于确定簇的成员关系。如果运行OPTICS时 `max_eps` 设置为默认值 `inf` ，则可以使用 `cluster_optics_dbscan` 方法对任意给定的 `eps` 值在线性时间内重复执行 `DBSCAN` 样式的簇提取。将 `max_eps` 设置为一个较低的值将导致较短的运行时间，并可以视为从每个点到找到其他潜在可达点的最大邻域半径。



OPTICS生成的可达性距离允许在单个数据集内对进行可变密度的簇提取。如上图所示，结合距离可达性和数据集 `ordering` 产生一个可达性图，其中点密度在Y轴上表示，并且点被排序以使得附近点相邻。在单个值处“切割”可达性图会产生类似 `DBSCAN` 的结果；“切割”上方的所有点都被归类为噪声，每次从左到右读取时都表示新的簇。使用*OPTICS*的默认簇提取查看图中的陡峭斜率以查找簇，用户可以使用参数 `xi` 定义什么算作陡峭斜率。对图表本身进行分析还有其他可能性，例如通过可达性树形图生成数据的层次表示，并且可以通过 `cluster_hierarchy` 参数访问算法检测到的簇的层次结构。上图是彩色编码的，因此平面空间中的簇颜色与可达性图的线性段簇相匹配。请注意，蓝色和红色聚类在可达性图中是相邻的，并且可以分层次地表示为较大父簇的子代。

> 示例
>
> - Demo of OPTICS clustering algorithm
>
> 与DBSCAN相比

OPTICS `cluster_optics_dbscan` 方法和DBSCAN的结果非常相似，但并不总是相同；具体而言，是在标记离群点和噪声点方面。这部分是因为由OPTICS处理的每个密集区域的第一个样本具有大的可达性值，使得接近其区域中的其他点，因此有时将被标记为噪声而不是离群点。当它们被视为被标记为离群点或噪声的候选点时，这会影响相邻点的判断。

请注意，对于任何单个值eps，DBSCAN的运行时间往往比OPTICS短；但是，对于不同eps 值的重复运行，单次运行OPTICS可能需要比DBSCAN更少的累积运行时间。同样重要的是要注意的是OPTICS的输出接近DBSCAN的只有eps和max_eps接近。

计算复杂度

采用空间索引树以避免计算整个距离矩阵，并允许在大量样本上有效地使用内存。可以通过 `metric` 关键字提供不同的距离度量。

对于大型数据集，可以通过 `HDBSCAN` 获得类似（但不相同）的结果 。HDBSCAN实现是多线程的，并且具有比OPTICS更好的算法运行时间复杂性，代价是更高的内存代价。对于使用HDBSCAN将耗尽系统内存的极大数据集，OPTICS将维持n（而不是n^2）内存缩放；然而，可能需要使用 `max_eps` 参数的调优来在合理的时间内给出解。

参考资料

- “OPTICS: ordering points to identify the clustering structure.” Ankerst, Mihael, Markus M. Breunig, Hans-Peter Kriegel, and Jörg Sander. In ACM Sigmod Record, vol. 28, no. 2, pp. 49-60. ACM, 1999.

2.3.9. Birch

The `Birch` 为给定数据构建一棵 Characteristic Feature Tree (CFT，聚类特征树)。 数据实质上是被有损压缩成一组 Characteristic Feature nodes (CF Nodes，聚类特征节点)。 CF Nodes 有许多称为 Characteristic Feature subclusters (CF Subclusters) 的子簇，并且这些位于非终结位置的CF Subclusters 可以拥有 CF Nodes 作为子节点。

CF Subclusters 保存用于簇的必要信息，防止将整个输入数据保存在内存中。 这些信息包括：

- Number of samples in a subcluster（子簇中样本数）.
- Linear Sum - A n-dimensional vector holding the sum of all samples（保存所有样本和的n维向量）
- Squared Sum - Sum of the squared L2 norm of all samples（所有样本的L2 norm的平方和）.
- Centroids - To avoid recalculation linear sum / n_samples（为了防止重复计算 linear sum / n_samples）.
- Squared norm of the centroids（质心的 Squared norm ）.

Birch 算法有两个参数，即阈值(threshold)和分支因子(branching factor)。分支因子限制了一个节点中的子簇的数量 ，阈值限制了新加入的样本和存在与现有子簇中样本的最大距离。

该算法可以视为将数据简化的一种方法，因为它将输入的数据简化到可以直接从CFT的叶子结点中获取的一组子簇。被简化的数据可以通过将其集合到全局簇(global clusterer)来进一步处理。全局簇可以 通过 `n_clusters` 来设置。

如果 `n_clusters` 被设置为 None，将直接读取叶子结点中的子簇，否则， 一个名为全局聚类的处理步骤将这些子簇全部标记为全局簇，这些样本将被打上距离它们最近的子簇的全局簇的标签。

算法描述：

- 一个新的样本作为一个CF Node 被插入到 CF Tree 的根节点。然后将其合并到根节点的子簇中去，使得合并后子簇拥有最小的半径，子簇的选取受阈值和分支因子的约束。如果子簇也拥有孩子节点，则重复执行这个步骤直到到达叶子结点。在叶子结点中找到最近的子簇以后，递归的更新这个子簇及其父簇的属性。

- 如果合并了新样本和最近的子簇获得的子簇半径大于阈值的平方(square of the threshold)， 并且子簇的数量大于分支因子，则将为这个样本分配一个临时空间。 最远的两个子簇被选取，剩下的子簇按照之间的距离分为两组作为被选取的两个子簇的孩子节点。
- 如果拆分的节点有一个父级子簇(parent subcluster)，并且有足够容纳一个新的子簇的空间，那么父簇拆分成两个。如果没有空间容纳一个新的簇，那么这个节点将被再次拆分，依次向上检查父节点是否需要分裂，如果需要， 则按叶子节点方式相同分裂。

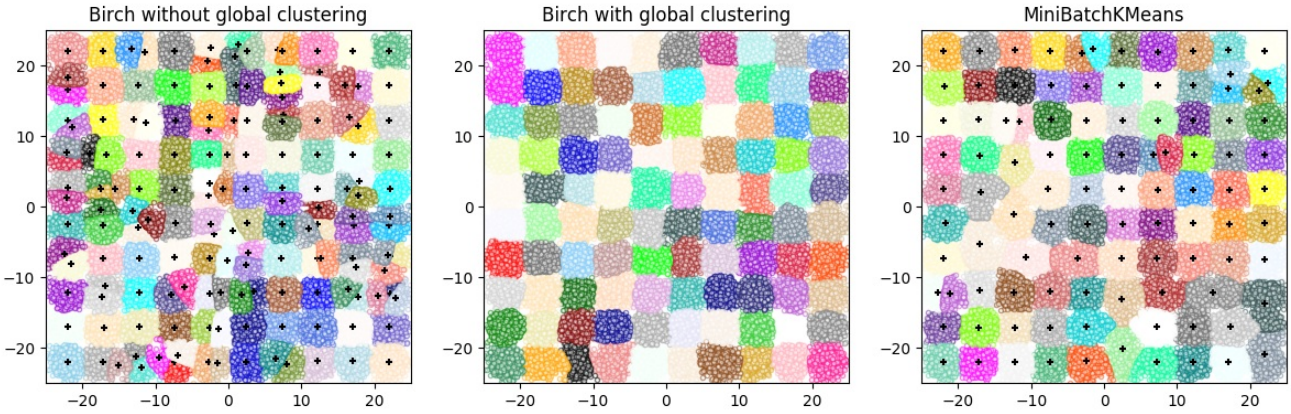
Birch or MiniBatchKMeans?

- Birch 在高维数据上表现不好。按经验来说，如果 `n_features` 大于20，通常使用 MiniBatchKMeans 更好。
- 如果需要减少数据实例的数量，或者如果需要大量的子聚类作为预处理步骤或者其他， Birch 比 MiniBatchKMeans 更有用。

How to use partial_fit?

为了避免对 global clustering 的计算，每次调用 `partial_fit` ，建议进行如下操作:

- 初始化 `n_clusters=None` 。
- 通过多次调用 `partial_fit` 训练所有数据。
- 通过使用 `brc.set_params(n_clusters=n_clusters)` ,设置 `n_clusters` 为所需值。
- 最后调用无参的 `partial_fit` ， 例如 `brc.partial_fit()` 执行全局聚类。



参考资料:

- Tian Zhang, Raghu Ramakrishnan, Maron Livny BIRCH: An efficient data clustering method for large databases. http://www.cs.sfu.ca/CourseCentral/459/han/papers/zhang96.pdf
- Roberto Perdisci JBirch - Java implementation of BIRCH clustering algorithm https://code.google.com/archive/p/jbirch

2.3.10. 聚类性能度量

度量聚类算法的性能不是简单的统计错误的数量或计算监督分类算法中的 准确率(precision)和 召回率(recall)。 特别地，任何度量指标（evaluation metric）不应该考虑到簇标签的绝对值，而是如果这个聚类方式所分离数据类似于部分真实簇分类 (ground truth set of classes 译注: *gorund truth*指的是真实值，在这里理解为标准答案)或者满足某些假设，在同于一个相似性度量（similarity metric）之下,使得属于同一个类内的成员比不同类的成员更加类似。

2.3.10.1. 调整后的兰德指数

在已知真实簇标签分配(the ground truth class assignment) `labels_true` 和我们的聚类算法基于相同样本所得到的 `labels_pred`，调整后的兰德指数**(adjusted Rand index)** 是一个函数，用于测量两个簇标签分配的值的 相似度 ，忽略排列(permutations)和 **with chance normalization**:

```
>>> from sklearn import metrics
>>> labels_true = [0, 0, 0, 1, 1, 1]
>>> labels_pred = [0, 0, 1, 1, 2, 2]

>>> metrics.adjusted_rand_score(labels_true, labels_pred)
0.24...
```

在预测的标签列表中重新排列 0 和 1，把 2 重命名为 3，得到相同的得分:

```
>>> labels_pred = [1, 1, 0, 0, 3, 3]
>>> metrics.adjusted_rand_score(labels_true, labels_pred)
0.24...
```

另外， `adjusted_rand_score` 是 对称的**(symmetric)** ：交换参数不会改变得分。它可以作为 共识度量**(consensus measure)**:

```
>>> metrics.adjusted_rand_score(labels_pred, labels_true)
0.24...
```

完美的标签的得分为 1.0

```
>>> labels_pred = labels_true[:]
>>> metrics.adjusted_rand_score(labels_true, labels_pred)
1.0
```

不良标签 (e.g. 无相关标签(independent labelings))的得分是负数 或 接近于 0.0 分:

```
>>> labels_true = [0, 1, 2, 0, 3, 4, 5, 1]
>>> labels_pred = [1, 1, 0, 0, 2, 2, 2, 2]
>>> metrics.adjusted_rand_score(labels_true, labels_pred)
-0.12...
```

2.3.10.1.1. 优点

- **Random (uniform) label assignments have a ARI score close to 0.0**（随机（统一）标签分配的 **ARI** 得分接近于 **0.0**） 适用于 `n_clusters` 和 `n_samples` 取任何值（这并不适用于原始的 Rand index 或者 V-measure 的情况）。
- **Bounded range**（有界范围） **[-1, 1]**: 负值是不良标签(例如:无关联标签)的得分，类似的聚类结果有一个正的 ARI 值， 1.0 是完美的匹配得分。
- **No assumption is made on the cluster structure**（对簇的结构不需作出任何假设）: 可以用于比较不同类的聚类算法，例如: k-means 和 谱聚类算法 间的比较。虽然，前者的簇是isotropic blob

shapes，后者的簇是 folded shapes。

2.3.10.1.2. 缺点

- 与惯量相反，**ARI** 要求先得知真实簇信息**(ground truth classes)**，而在实践中几乎不可用，或者需要人工标注者手动分配（如在监督学习环境中）。

然而，ARI 在纯粹的非监督学习的环境下，作为聚类模型选择过程中共识索引(Consensus Index)的一个构建模块是非常有用的（TODO）。

> 示例：
>
> - Adjustment for chance in clustering performance evaluation: 分析数据集大小对随机分配聚类度量值的影响。

2.3.10.1.3. 数学表达

如果 C 是一个真实簇的标签分配， K 是簇的个数，我们定义 a 和 b 如：

- a，在 C 中的相同集合的与 K 中的相同集合中的元素对数
- b，在 C 中的不同集合与 K 中的不同集合中的元素对数

原始（未排序）的 Rand index 则由下式给出：

$$\mathrm{RI} = \frac{a + b}{C_2^{n_{samples}}}$$

其中 $C_2^{n_{samples}}$ 是数据集中可能的数据对(pairs)的总数（不排序）。

然而，RI 得分不能保证 随机标签分配(random label assignments)会获得接近零的值（特别是如果簇的数量与样本数量有着相同的规模排序）。

为了抵消这种影响，我们可以通过定义 adjusted Rand index 来低估(discount)随机标签的预期 RI $E[\mathrm{RI}]$,如下所示：

$$\mathrm{ARI} = \frac{\mathrm{RI} - E[\mathrm{RI}]}{\max(\mathrm{RI}) - E[\mathrm{RI}]}$$

> 参考资料
>
> - Comparing Partitions L. Hubert and P. Arabie, Journal of Classification 1985
> - Wikipedia entry for the adjusted Rand index

2.3.10.2. 基于互信息(mutual information)的得分

在已知 真实簇的标签分配 `labels_true` 和我们的聚类算法基于相同样本所得到的 `labels_pred`， 互信息 是测

量两个标签分配的 一致性**(agreement)**，忽略排列(permutations)。这种测量方案有两个不同的标准化版本可用，**Normalized Mutual Information(NMI)** 和 **Adjusted Mutual Information(AMI)**。NMI 经常在文献中使用，而 AMI 最近被提出，并且 **normalized against chance** ：

```
>>> from sklearn import metrics
>>> labels_true = [0, 0, 0, 1, 1, 1]
>>> labels_pred = [0, 0, 1, 1, 2, 2]

>>> metrics.adjusted_mutual_info_score(labels_true, labels_pred)
0.22504...
```

在预测的标签列表中重新排列 0 和 1，把 2 重命名为 3，得到相同的得分：

```
>>> labels_pred = [1, 1, 0, 0, 3, 3]
>>> metrics.adjusted_mutual_info_score(labels_true, labels_pred)
0.22504...
```

全部的， `mutual_info_score` ， `adjusted_mutual_info_score` 和 `normalized_mutual_info_score` 是对称的：交换参数不会更改分数。因此，它们可以用作 共识度量：

```
>>> metrics.adjusted_mutual_info_score(labels_pred, labels_true)
0.22504...
```

完美标签得分是 1.0：

```
>>> labels_pred = labels_true[:]
>>> metrics.adjusted_mutual_info_score(labels_true, labels_pred)
1.0

>>> metrics.normalized_mutual_info_score(labels_true, labels_pred)
1.0
```

这对于 `mutual_info_score` 是不正确的，因此更难判断：

```
>>> metrics.mutual_info_score(labels_true, labels_pred)
0.69...
```

不良标签（例如，无相关标签）具有非正分数：

```
>>> labels_true = [0, 1, 2, 0, 3, 4, 5, 1]
>>> labels_pred = [1, 1, 0, 0, 2, 2, 2, 2]
>>> metrics.adjusted_mutual_info_score(labels_true, labels_pred)
-0.10526...
```

2.3.10.2.1. 优点

- **Random (uniform) label assignments have a AMI score close to 0.0**（随机（统一）标签分

配的**AMI**评分接近**0.0**） 适用于 `n_clusters` 和 `n_samples` 取任何值（这并不适用于原始的 Rand index 或者 V-measure 的情况）。

- **Upper Bound of 1**: 数值趋近于零，是说明两个标签分配之间是独立，无关联；而数值趋近于一时， 表示两者之间有着极高的一致性(significant agreement)。 甚至，当AMI的取值正好是 1 时， 说明两个标签分配时完全相等的(无论是否排列过)。

2.3.10.2.2. 缺点

- 与惯量相反，**MI-based measures** 需要先得知 **ground truth classes** ，而在实践中几乎不可用，或者需要人工标注者手动分配（如在监督学习环境中）。

然而，基于 MI 的测量方式(MI-based measures)也可用于纯无人监控的环境， 作为聚类模型选择过程中共识索引的一个构建模块。

- NMI 和 MI 不能进行使得随机标签度量的得分为0的调整。

> 示例：
>
> - Adjustment for chance in clustering performance evaluation: 分析数据集大小对随机分配聚类度量值的影响。 此示例还包括 Adjusted Rand Index。

2.3.10.2.3. 数学公式

假设两个标签分配（在相同的 N 个对象中进行），U 和 V。 它们的 熵(entropy)是一个分区集合(partition set)的不确定性量，定义如下：

$$H(U) = -\sum_{i=1}^{|U|} P(i)\log(P(i))$$

其中 $P(i) = |U_i|/N$ 是从 U 中随机选取的对象， 选取对象落入到类 U_i 的概率。同样对于 V：

$$H(V) = -\sum_{j=1}^{|V|} P'(j)\log(P'(j))$$

使用 $P'(j) = |V_j|/N$. U 和 V 之间的 mutual information (MI) 由下式计算：

$$\mathrm{MI}(U,V) = \sum_{i=1}^{|U|}\sum_{j=1}^{|V|} P(i,j)\log\left(\frac{P(i,j)}{P(i)P'(j)}\right)$$

其中 $P(i,j) = |U_i \cap V_j|/N$ 是随机选择的对象同时落入两个类的概率 U_i 和 V_j。

也可以用设定的基数表达式表示：

$$\mathrm{MI}(U,V) = \sum_{i=1}^{|U|}\sum_{j=1}^{|V|}\frac{|U_i \cap V_j|}{N}\log\left(\frac{N|U_i \cap V_j|}{|U_i||V_j|}\right)$$

归一化(normalized) mutual information 被定义为

$$\mathrm{NMI}(U,V) = \frac{\mathrm{MI}(U,V)}{\sqrt{H(U)H(V)}}$$

mutual information 的值以及 归一化变量(normalized variant)的值都不会为随机标签度量而调整，但是会随着不同的簇标签数量的增加，不管标签分配之间的 “mutual information” 的实际数量如何，都会趋向于增加。

mutual information 的期望值可以用 Vinh, Epps 和 Bailey,(2009) 的以下公式来计算。在这个方程式中，$a_i = |U_i|$ (U_i 中元素的数量) 和 $b_j = |V_j|$ (V_j 中元素的数量).

$$E[\mathrm{MI}(U,V)] = \sum_{i=1}^{|U|}\sum_{j=1}^{|V|}\sum_{n_{ij}=(a_i+b_j-N)^+}^{\min(a_i,b_j)}\frac{n_{ij}}{N}\log\left(\frac{N.n_{ij}}{a_i b_j}\right)\frac{a_i!b_j!(N-a_i)!(N-b_j)!}{N!n_{ij}!(a_i-n_{ij})!(b_j-n_{ij})!(N-a_i-b_j+n_{ij})!}$$

使用期望值，然后可以使用与 adjusted Rand index 相似的形式来计算调整后的 mutual information:

$$\mathrm{AMI} = \frac{\mathrm{MI} - E[\mathrm{MI}]}{\max(H(U),H(V)) - E[\mathrm{MI}]}$$

对于归一化互信息和调整互信息，归一化值通常是各聚类熵的某种广义均值。存在各种广义的方法，没有确定的规则来确定哪种方法优于其他方法。这个决定很大程度上是以领域为基础(即不同的领域中，有不同的最优方法);例如，在社区发现算法（Community Detection）中，算术平均是最常见的。每一种归一化方法都提供了“qualitatively similar behaviours”[YAT2016]。在我们的实现中，这是由 `average_method` 参数控制的。

Vinh等(2010)用平均法命名NMI和AMI的变体[VEB2010]。它们的“平方根”和“和”平均值是几何和算术平均值;我们使用了这些更广泛的通用名称。

参考资料

- Strehl, Alexander, and Joydeep Ghosh (2002). “Cluster ensembles – a knowledge reuse framework for combining multiple partitions”. Journal of Machine Learning Research 3: 583–617. doi:10.1162/153244303321897735.
- Wikipedia entry for the (normalized) Mutual Information
- Wikipedia entry for the Adjusted Mutual Information
- [VEB2009] Vinh, Epps, and Bailey, (2009). “Information theoretic measures for clusterings comparison”. Proceedings of the 26th Annual International Conference on Machine Learning - ICML ‘09. doi:10.1145/1553374.1553511. ISBN 9781605585161.
- [VEB2010] Vinh, Epps, and Bailey, (2010). “Information Theoretic Measures for Clusterings Comparison: Variants, Properties, Normalization and Correction for Chance”. JMLR http://jmlr.csail.mit.edu/papers/volume11/vinh10a/vinh10a.pdf
- [YAT2016] Yang, Algesheimer, and Tessone, (2016). “A comparative analysis of community detection algorithms on artificial networks”. Scientific Reports 6: 30750. doi:10.1038/srep30750.

2.3.10.3. 同质性，完整性和 V-measure

已知真实簇标签分配，可以使用 条件熵(conditional entropy)分析来定义一些直观的度量（intuitive metric）。

特别是 Rosenberg 和 Hirschberg (2007) 为任何簇分配定义了以下两个理想的目标：

- 同质性(**homogeneity**)： 每个簇只包含一个类的成员
- 完整性(**completeness**)： 给定类的所有成员都分配给同一个簇。

我们可以把这些概念作为分数 `homogeneity_score` 和 `completeness_score` 。两者均在 0.0 以下 和 1.0 以上（越高越好）：

```
>>> from sklearn import metrics
>>> labels_true = [0, 0, 0, 1, 1, 1]
>>> labels_pred = [0, 0, 1, 1, 2, 2]

>>> metrics.homogeneity_score(labels_true, labels_pred)
0.66...

>>> metrics.completeness_score(labels_true, labels_pred)
0.42...
```

称为 **V-measure** 的调和平均数(harmonic mean)由以下函数计算 `v_measure_score` ：

该函数公式如下：

$$v = \frac{(1+\beta) \times \text{homogeneity} \times \text{completeness}}{(\beta \times \text{homogeneity} + \text{completeness})}$$

`beta` 默认值为1.0，但如果beta值小于1，则为：

```
>>> metrics.v_measure_score(labels_true, labels_pred, beta=0.6)
0.54...
```

更大的 `beta` 权重将提高同质性，当使用大于1的 `beta` 值时：

```
>>> metrics.v_measure_score(labels_true, labels_pred, beta=1.8)
0.48...
```

更大的 `beta` 权重将提高完整性。

V-measure 实际上等于上面讨论的 mutual information (NMI) ，仅仅是聚合函数（aggregation function）是算术均值[B2011]。

同质性， 完整性 and V-measure 可以使用 `homogeneity_completeness_v_measure` 进行计算 如下：

```
>>> metrics.homogeneity_completeness_v_measure(labels_true, labels_pred)
...
(0.66..., 0.42..., 0.51...)
```

以下聚类分配稍微好一些，因为它是同质但并不完整：

```
>>> labels_pred = [0, 0, 0, 1, 2, 2]
>>> metrics.homogeneity_completeness_v_measure(labels_true, labels_pred)
...
(1.0, 0.68..., 0.81...)
```

注意：

- `v_measure_score` 是 对称的：它可以用于评估同一数据集上两个 无相关标签分配(independent assignments)的 一致性。
- `completeness_score` 和 `homogeneity_score` 却不适用于上述情况：两者间存在下述约束：

```
homogeneity_score(a, b) == completeness_score(b, a)
```

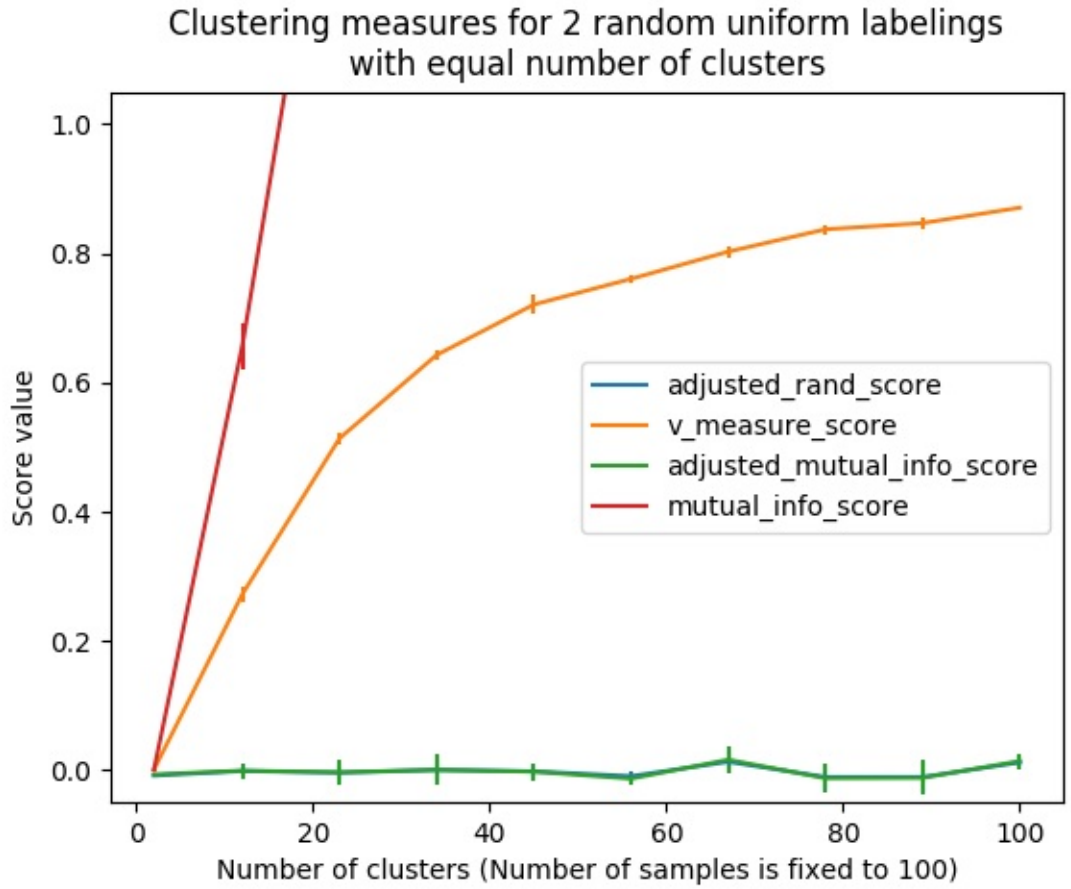
2.3.10.3.1. 优点

- **Bounded scores**（有界的分数）：`0.0` 是最坏的，`1.0` 是一个完美的分数.
- `Intuitive interpretation`（直观解释）：具有不良 `V-measure` 的聚类可以在 在同质性和完整性方面进行定性分析 以更好地感知到标签分配过程中的错误类型。
- **No assumption is made on the cluster structure**（对簇的结构没有作出任何假设）：可以用于比较不同类的聚类算法，例如：`k-means` 和 `spectral clustering algorithms`（谱聚类算法）间的比较。虽然，前者的簇是`isotropic blob shapes`，后者的簇是 `folded shapes`。

2.3.10.3.2. 缺点

- 之前介绍的度量标准(`metrics`)并不针对随机标记进行标准化**(not normalized with regards to random labeling)**：这意味着，依赖样本数量，簇数量和 标定过的真实数据类数量，完全随机的标注方式并不总是产生 同质性，完整性 和 `v-measure` 的相同值。特别是 随机标注不会产生零分，特别是当簇数量大时。

当样本数量超过 `1000`，簇的数量小于 `10` 时，可以安全地忽略此问题。对于较小的样本数量或者较大数量的簇，使用 **adjusted index** 例如 **Adjusted Rand Index (ARI)**）。



- 这些度量标准 要求先得知真实数据簇，而在实践中几乎不可用，或需要人工标注者人工分配（如在受监督的学习环境中）。

示例：

- `Adjustment for chance in clustering performance evaluation`: 分析数据集大小对随机分配聚类度量值的影响。

2.3.10.3.3. 数学表达

同质性 和 完整性 的得分由下面公式给出：

$$h = 1 - \frac{H(C|K)}{H(C)}$$

$$c = 1 - \frac{H(K|C)}{H(K)}$$

其中 $H(C|K)$ 是 给定簇分配的类的条件熵 ，由下式给出：

$$H(C|K) = -\sum_{c=1}^{|C|}\sum_{k=1}^{|K|}\frac{n_{c,k}}{n}\cdot\log\left(\frac{n_{c,k}}{n_k}\right)$$

并且 $H(C)$ 是 已聚合的类的熵 **(entropy of the classes)**，并且由下式给出：

$$H(C) = -\sum_{c=1}^{|C|} \frac{n_c}{n} \cdot \log\left(\frac{n_c}{n}\right)$$

n 个样本总数， n_c 和 n_k 分别属于 c 类和簇 k 的样本数，最后 $n_{c,k}$ 分配给簇 k 的类 c 的样本数。

给定类分配的簇的条件熵（**conditional entropy of clusters given class**）$H(K|C)$ 和 簇的熵 $H(K)$ 以 对称的形式 定义。

Rosenberg 和 Hirschberg 进一步定义 **V-measure** 作为 同质性和完整性的调和均值：

$$v = 2 \cdot \frac{h \cdot c}{h + c}$$

> 参考资料
>
> - V-Measure: A conditional entropy-based external cluster evaluation measure Andrew Rosenberg and Julia Hirschberg, 2007
> - [B2011] Identication and Characterization of Events in Social Media, Hila Becker, PhD Thesis.

2.3.10.4. Fowlkes-Mallows 得分

当样本的已标定的真实类分配已知时，可以使用 Fowlkes-Mallows 指数（`sklearn.metrics.fowlkes_mallows_score`）。Fowlkes-Mallows 得分 FMI 被定义为 成对的准确率和召回率的几何平均值：

$$\text{FMI} = \frac{\text{TP}}{\sqrt{(\text{TP} + \text{FP})(\text{TP} + \text{FN})}}$$

其中的 `TP` 是 真正例**(True Positive)** 的数量（即，真实标签组和预测标签组中属于相同簇的点对数），`FP` 是 假正例**(False Positive)** （即，在真实标签组中属于同一簇的点对数，而不在预测标签组中），`FN` 是 假负例**(False Negative)** 的数量（即，预测标签组中属于同一簇的点对数，而不在真实标签组中）。

得分范围为 0 到 1。较高的值表示两个簇之间的良好相似性。

```
>>> from sklearn import metrics
>>> labels_true = [0, 0, 0, 1, 1, 1]
>>> labels_pred = [0, 0, 1, 1, 2, 2]
```

```
>>> metrics.fowlkes_mallows_score(labels_true, labels_pred)
0.47140...
```

在预测的标签列表中重新排列 0 和 1，把 2 重命名为 3，得到相同的得分：

```
>>> labels_pred = [1, 1, 0, 0, 3, 3]

>>> metrics.fowlkes_mallows_score(labels_true, labels_pred)
0.47140...
```

完美的标签得分是 1.0：

```
>>> labels_pred = labels_true[:]
>>> metrics.fowlkes_mallows_score(labels_true, labels_pred)
1.0
```

不良标签（例如,无相关标签）的得分为 0：

```
>>> labels_true = [0, 1, 2, 0, 3, 4, 5, 1]
>>> labels_pred = [1, 1, 0, 0, 2, 2, 2, 2]
>>> metrics.fowlkes_mallows_score(labels_true, labels_pred)
0.0
```

2.3.10.4.1. 优点

- **Random (uniform) label assignments have a AMI score close to 0.0**（随机（统一）标签分配的**AMI**评分接近**0.0**） 适用于 `n_clusters` 和 `n_samples` 取任何值（这并不适用于原始的 Rand index 或者 V-measure 的情况）。
- **Upper Bound of 1**: 数值趋近于零，是说明两个标签分配之间是独立，无关联；而数值趋近于一时， 表示两者之间有着极高的相似度(significant agreement)。 甚至，当AMI的取值正好是 1 时， 说明两个标签分配时完全相等的(无论是否排列过)。
- **No assumption is made on the cluster structure**（对簇的结构没有作出任何假设）： 可以用于比较不同类的聚类算法，例如： k-means 和 spectral clustering algorithms（谱聚类算法）间的比较。虽然，前者的簇是isotropic blob shapes， 后者的簇是 folded shapes。

2.3.10.4.2. 缺点

- 与惯量相反，基于 **FMI** 的测量方案需要先了解已标注的真数据类 ，而在实践中几乎不可用，或者需要人工标注者手动分配（如在监督学习的学习环境中）。

> 参考资料
>
> - E. B. Fowkles and C. L. Mallows, 1983. “A method for comparing two hierarchical clusterings”. Journal of the American Statistical Association. http://wildfire.stat.ucla.edu/pdflibrary/fowlkes.pdf
> - Wikipedia entry for the Fowlkes-Mallows Index

2.3.10.5. Silhouette 系数

如果不知道真实簇标签，则必须使用模型本身进行度量。Silhouette 系数（ `sklearn.metrics.silhouette_score` ）是一个这样的评估例子，其中较高的 Silhouette 系数得分和能够更好定义的聚类的模型相关联。Silhouette 系

数 是依据每个样本进行定义，由两个得分组成：

- **a**: 样本与同一类别中所有其他点之间的平均距离。
- **b**: 样本与 下一个距离最近的簇 中的所有其他点之间的平均距离。

然后将单个样本的 Silhouette 系数 *s* 给出为：

$$s = \frac{b - a}{max(a, b)}$$

一组样本的 Silhouette 系数可以作为其中每个样本的 Silhouette 系数的平均值。

```
>>> from sklearn import metrics
>>> from sklearn.metrics import pairwise_distances
>>> from sklearn import datasets
>>> dataset = datasets.load_iris()
>>> X = dataset.data
>>> y = dataset.target
```

在正常使用情况下，将 Silhouette 系数应用于聚类结果的分析。

```
>>> import numpy as np
>>> from sklearn.cluster import KMeans
>>> kmeans_model = KMeans(n_clusters=3, random_state=1).fit(X)
>>> labels = kmeans_model.labels_
>>> metrics.silhouette_score(X, labels, metric='euclidean')
...
0.55...
```

> 参考资料
>
> - Peter J. Rousseeuw (1987). “Silhouettes: a Graphical Aid to the Interpretation and Validation of Cluster Analysis”. Computational and Applied Mathematics 20: 53–65. doi:10.1016/0377-0427(87)90125-790125-7).

2.3.10.5.1. 优点

- 对于不正确的聚类，分数为 -1 ，高密度聚类为 +1 。零点附近的分数表示 重叠的聚类。
- 当簇密集且分离较好时，分数更高，这关联到了簇的标准概念。

2.3.10.5.2. 缺点

- 凸簇的 Silhouette 系数通常比其他类型的簇更高，例如通过 DBSCAN 获得的基于密度的簇。

> 示例：
>
> - Selecting the number of clusters with silhouette analysis on KMeans clustering : 在这个例子中，silhouette 分析用于为 n_clusters 选择最佳值.

2.3.10.6. Calinski-Harabaz 指数

如果不知道真实簇标签，则可以使用 Calinski-Harabaz 指数 (`sklearn.metrics.calinski_harabaz_score`)-或被称为方差比准则(Variance Ratio Criterion)-来评估模型，其中较高的 Calinski-Harabaz 的得分和能够更好定义的聚类的模型相关联。

对于 k 簇，Calinski-Harabaz 得分 s 是通过簇间色散平均值(between-clusters dispersion mean)与簇内色散之间(within-cluster dispersion)的比值给出的:

$$s(k) = \frac{\mathrm{Tr}(B_k)}{\mathrm{Tr}(W_k)} \times \frac{N - k}{k - 1}$$

其中 B_K 是组间色散矩阵(between group dispersion matrix)， W_K 是由以下定义的簇内色散矩阵(within-cluster dispersion matrix):

$$W_k = \sum_{q=1}^{k} \sum_{x \in C_q} (x - c_q)(x - c_q)^T$$

$$B_k = \sum_{q} n_q (c_q - c)(c_q - c)^T$$

N 为数据中的点数， C_q 为簇 q 中的点集， c_q 为簇 q 的中心， c 为 E 的中心， n_q 为簇 q 中的点数。

```
>>> from sklearn import metrics
>>> from sklearn.metrics import pairwise_distances
>>> from sklearn import datasets
>>> dataset = datasets.load_iris()
>>> X = dataset.data
>>> y = dataset.target
```

在正常使用情况下，将 Calinski-Harabaz指数应用于聚类结果的分析。

```
>>> import numpy as np
>>> from sklearn.cluster import KMeans
>>> kmeans_model = KMeans(n_clusters=3, random_state=1).fit(X)
>>> labels = kmeans_model.labels_
>>> metrics.calinski_harabaz_score(X, labels)
560.39...
```

2.3.10.6.1. 优点

- 当簇密集且分离较好时，分数更高，这关联到了簇的标准概念。
- 得分计算很快。

2.3.10.6.2. 缺点

- 凸簇的 Calinski-Harabaz 指数通常高于其他类型的簇，例如通过 DBSCAN 获得的基于密度的簇。

参考资料

- Caliński, T., & Harabasz, J. (1974). “A dendrite method for cluster analysis”. Communications in Statistics-theory and Methods 3: 1-27. doi:10.1080/03610926.2011.560741.

2.3.10.7. Davies-Bouldin Index

如果不知道真实簇标签，则可以使用 Davies-Bouldin Index（简称： DBI）(sklearn.metrics.davies_bouldin_score) 去度量模型， 这里，一个较低的DBI值总是意味着一个具有更好簇间分割表现的模型。该指数(Index) 每个簇 C_iCi ，其中$i = 0, ..., k$i=0,…,k和最大相似值的C_jCj之间的相似度均值。在该指数的上下文里，相似度被定义一个用于协调的度量：

- Si S_i Si ， 簇 i i i 内每个点和簇心(centroid of cluster)的平均距离 — 也称簇直径。
- dij d_{ij} dij， 簇心 i i i 和 j j j 之间的距离。

以下是一个简单构建 Rij R_{ij} Rij 的方法，使得它具有非负性和对称性：

Rij=si+sjdij R*{ij} = \frac{s_i + s_j}{d*{ij}} Rij=dijsi+sj

接着， DBI 就可以如下定义：

DB=1k∑i=1kmaxi≠jRij DB = \frac{1}{k} \sum*{i=1}^k \max*{i \ne j} R_{ij}
DB=k1∑i=1kmaxi≠jRij

零是最低得得分. 当 该值越接近零，意味着一个更好的分割表现。

在一般的情况下， DBI 总是应用于如下得簇分析结果：

```
>>> from sklearn import datasets
>>> iris = datasets.load_iris()
>>> X = iris.data
>>> from sklearn.cluster import KMeans
>>> from sklearn.metrics import davies_bouldin_score
>>> kmeans = KMeans(n_clusters=3, random_state=1).fit(X)
>>> labels = kmeans.labels_
>>> davies_bouldin_score(X, labels)
0.6619...
```

2.3.10.7.1. 优点

- BDI 的计算 比 Silhouette scores 要简单。
- 该指数仅仅只计算数据集内的数量和特征。

2.3.10.7.2. 缺点

- 凸簇(convex clusters)的 DBI 通常比其他类型的簇更高，例如通过 DBSCAN 获得的基于密度的簇。
- 在一般情况下， 质心距离只能使用欧氏空间内的距离度量来衡量。
- 该方法得出的一个好的值并不意味着这是最好的信息检测技术。

参考资料

- Davies, David L.; Bouldin, Donald W. (1979). “A Cluster Separation Measure” IEEE Transactions on Pattern Analysis and Machine Intelligence. PAMI-1 (2): 224-227. doi:10.1109/TPAMI.1979.4766909.
- Halkidi, Maria; Batistakis, Yannis; Vazirgiannis, Michalis (2001). “On Clustering Validation Techniques” Journal of Intelligent Information Systems, 17(2-3), 107-145. doi:10.1023/A:1012801612483.
- Wikipedia entry for Davies-Bouldin index.

2.3.10.8. Contingency Matrix

列联矩阵（Contingency Matrix）(sklearn.metrics.cluster.contingency_matrix) 记录了每个真实/预测簇对之间的交叉基数。 列联矩阵为所有的聚合度量提供了足量的统计数据，而其中样本都是独立和相同分布，并不考虑没有被聚合的实例。

一个实例如下：

```
>>> from sklearn.metrics.cluster import contingency_matrix
>>> x = ["a", "a", "a", "b", "b", "b"]
>>> y = [0, 0, 1, 1, 2, 2]
>>> contingency_matrix(x, y)
array([[2, 1, 0],
       [0, 1, 2]])
```

输出array的第一行 意味着有三个样本的真实簇是'a'. 而在这三个样本中， 两个的预测簇是 0,一个是 1， 没有一个是 2。而第二行意味着有三个样本的真实簇 是'b'。 而其中，没有一个样本的预测簇是 0， 一个是 1， 两个是 2。

这种 分类方式的混淆矩阵(confusion matrix) 是一个方块(行数和列数相等)列联矩阵，该矩阵的行列分别对应样本的分类。

2.3.10.8.1. 优点

- 允许 测试每个真实 簇在预测簇上的散布,反之亦然。
- 列联矩阵的计算利用了两个聚类方式之间的相似度统计数据（就像本文档中记录其它方式一样）

2.3.10.8.2. 缺点

- 列联矩阵易用于描述小数量的簇，但是当簇的数量变得很大时，矩阵的描述就会变得异常艰难。
- 该方法并不能用于聚类优化的一个度量手段。

参考资料

- Wikipedia entry for contingency matrix

我们一直在努力

apachecn/sklearn-doc-zh

加入QQ群

2.4. 双聚类

校验者：@udy@barrycg翻译者：@程威

Biclustering(双向聚类) 的实现模块是 `sklearn.cluster.bicluster` 。 双向聚类算法对数据矩阵的行列同时进行聚类。而这些行列的聚类称之为 双向簇(biclusters)。每一次聚类都会基于原始数据矩阵确定一个子矩阵， 并且这些子矩阵具有一些需要的属性。

例如， 给定一个矩阵 `(10, 10)` ， 如果对其中三行二列进行双向聚类，就可以获得一个子矩阵 `(3, 2)` 。

```
>>> import numpy as np
>>> data = np.arange(100).reshape(10, 10)
>>> rows = np.array([0, 2, 3])[:, np.newaxis]
>>> columns = np.array([1, 2])
>>> data[rows, columns]
array([[ 1,  2],
 [21, 22],
 [31, 32]])
```

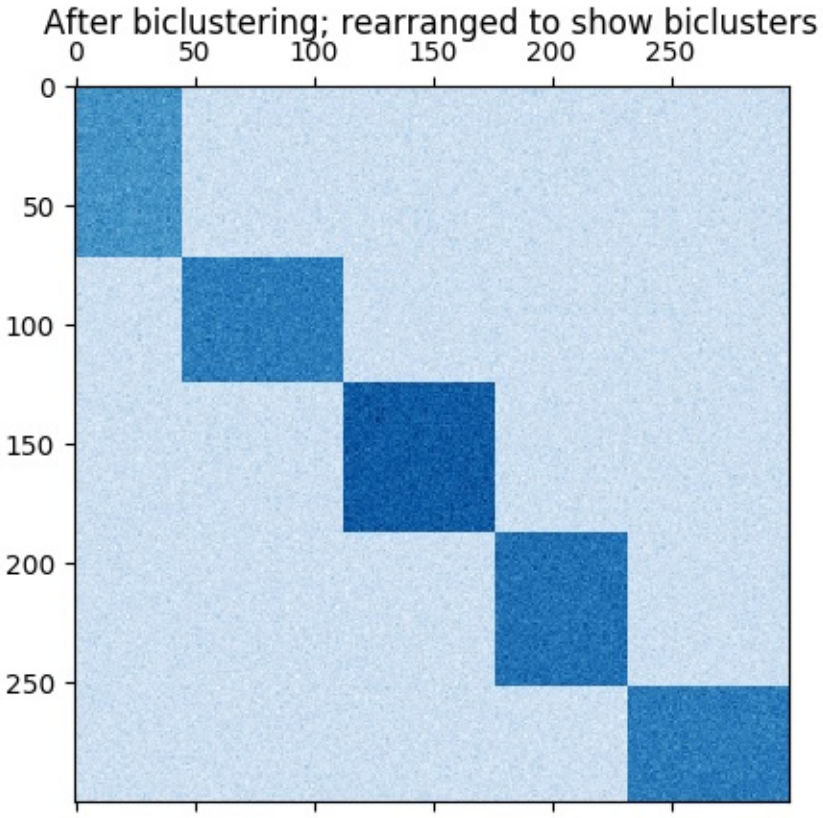
为了可视化，给定一个双向簇，数据矩阵的行列可以重新分配，使得该双向簇是连续的。

不同的双向聚类算法在如何定义双向簇方面有所不同，但其中通用类型包括：

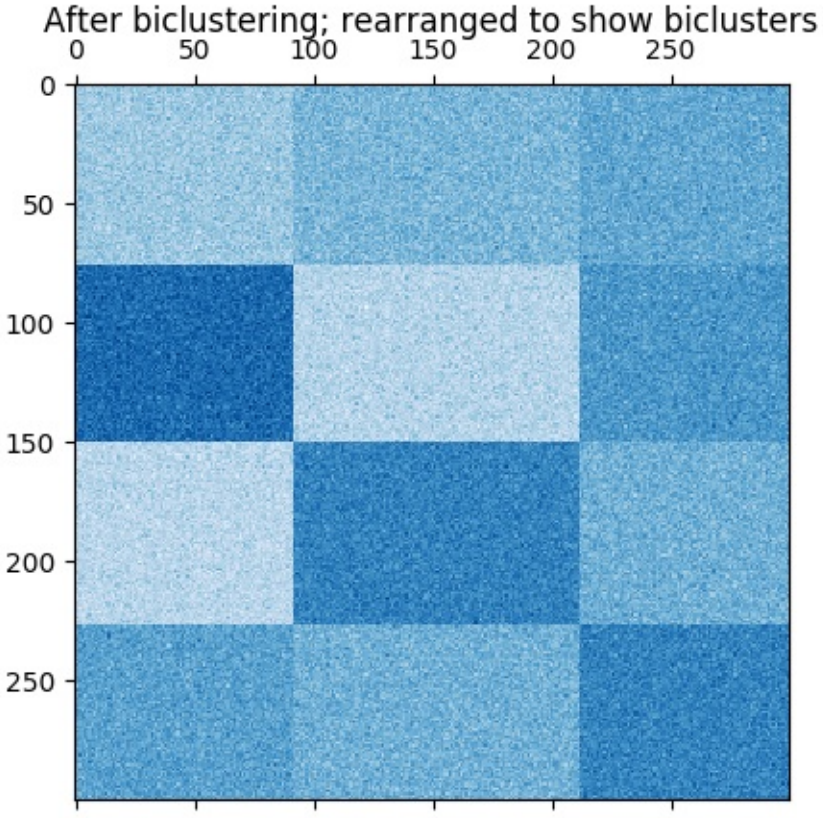
- 常量， 常量行或常量列。
- 异常高的或者低的值。
- 低方差的子矩阵。
- 相互关联的行列。

算法在给双向簇分配行列的方式不同， 会导致不同的双向聚类结构。当行和列分成区块时，会出现块对角或者棋盘结构。

如果每一行和每一列仅属于一个双向簇，重新排列数据矩阵的行和列，会使得双向簇出现在对角线上。下面是一个例子，此结构的双向簇具有比其他行列更高的平均值：



在棋盘结构的例子中， 每一行属于所有的列簇， 每一列属于所有的行簇。下面是一个例子，每个双向簇内的值差异较小：



在拟合模型之后， 可以在 `rows` 和 `columns` 属性中找到行簇和列簇的归属信息(membership)。 `rows[i]` 是一个二元向量， 其中非零元素表示属于双向簇 `i` 的行。 同样的， `columns` `[i]` 就表示属于双向簇 `i` 的列。

一些模块也有 `rowlabels` 和 `columnlabels` 属性。 这些模块可以对行列进行分区， 例如在块对角或者棋盘双向簇结构。

> 注意双向聚类在不同的领域有很多其他名称，包括 co-clustering, two-mode clustering, two-way clustering, block clustering, coupled two-way clustering 等.有一些算法的名称，比如 Spectral Co-Clustering algorithm，反应了这些备用名称。

2.4.1. Spectral Co-Clustering

`SpectralCoclustering(联合谱聚类)` 算法找到的双向簇的值比其它的行和列更高。每一个行和列都只属于一个双向簇，所以重新分配行和列，使得分区连续显示对角线上的高值：

> 注意算法将输入的数据矩阵看做成二分图：该矩阵的行和列对应于两组顶点，每个条目对应于行和列之间的边，该算法近似的进行归一化，对图进行切割，找到更重的子图。

2.4.1.1. 数学公式

找到最优归一化剪切的近似解，可以通过图形的 Laplacian 的广义特征值分解。 通常这意味着直接使用 Laplacian 矩阵. 如果原始数据矩阵 A 的形状 $m \times n$， 则对应的二分图(bipartite graph)的 Laplacian 矩阵具有形状 $(m+n) \times (m+n)$。 但是， 在这种情况下， 直接使用 A ，因为它更小，更有效率。

输入矩阵 A 被预处理如下：

$$A_n = R^{-1/2} A C^{-1/2}$$

R 是对角线矩阵， 其中元素 i 等于 $\sum_j A_{ij}$ ， C 是 对角矩阵，其中元素 j 等于 $\sum_i A_{ij}$。

奇异值分解， $A_n = U \Sigma V^\top$ ，产生了 A 行列的分区. 左边奇异向量的子集给予行分区，右边的奇异向量的子集给予列分区。

奇异向量 $\ell = \lceil \log_2 k \rceil$ 从第二个开始， 提供所需的分区信息。这些用于形成矩阵 Z：

$$Z = \begin{bmatrix} R^{-1/2} U \\ C^{-1/2} V \end{bmatrix}$$

U 的列是 $u_2, \ldots, u_{\ell+1}$， 和 V 的列也具有相似特性。

然后 Z 的所有行通过使用 k-means 进行聚类. 第一个 `n_rows` 标签提供行分区信息, 剩下的 `n_columns` 标签提供列分区信息。

> 示例:
>
> - A demo of the Spectral Co-Clustering algorithm: 如何用双向簇产生一个数据矩阵并应用。
> - Biclustering documents with the Spectral Co-clustering algorithm:一个在 20 个新闻组数据集中发现双向簇的例子
>
> 参考资料:
>
> - Dhillon, Inderjit S, 2001. Co-clustering documents and words using bipartite spectral graph partitioning.

2.4.2. Spectral Biclustering

`SpectralBiclustering(双向谱聚类)` 算法假设输入的数据矩阵具有隐藏的棋盘结构。具有这种结构的矩阵的行列可能被分区，使得在笛卡尔积中的大部分双向簇的列簇和行簇是近似恒定的。

例如，如果有两个行分区和三个列分区，每一行属于三个双向簇，每一列属于两个双向簇。

这个算法对矩阵的行和列进行分区，以至于提供一个相应的块状不变的棋盘矩阵，近似于原始矩阵。

2.4.2.1. 数学表示

输入矩阵 A 先归一化，使得矩阵的棋盘模式更明显。这里有三种方法:

- 独立的行列归一化, 如联合谱聚类中所示. 这个方法使得所有行进行行内相加得到一个相同常量，所有列相加得到另一个相同常量。
- **Bistochastization**: 重复行和列归一化直到收敛。该方法使得行和列相加得到一个相同的常数。
- 对数归一化: 数据矩阵的对数是 $L = \log A$. 列对数就是 $\overline{L_{i\cdot}}$, 行 对数就是 $\overline{L_{\cdot j}}$, $\overline{L_{\cdot\cdot}}$ 是 L 的整体平均. 最终矩阵通过下面的公式计算 $K_{ij} = L_{ij} - \overline{L_{i\cdot}} - \overline{L_{\cdot j}} + \overline{L_{\cdot\cdot}}$ 归一化后，第一个的奇异向量被计算，就如同联合谱聚类算法一样。

如果使用对数归一化，则所有的奇异向量都是有意义的。但是, 如果是独立归一化或Bistochastization 被使用, 第一个奇异向量, u_1 和 v_1。 会被丢弃。 从现在开始, “第一个” 奇异向量指的是 $u_2 \dots u_{p+1}$ 和 $v_2 \dots v_{p+1}$ ，除了对数归一化的情况。

给定这些奇异向量，按照分段常数向量的最佳近似程度,将他们排序。使用一维 k-means 找到每个向量的近似值并使用欧氏距离进行评分。最好的左右奇异向量的某个子集被选择。下一步, 数据将被投影到奇异向量的最佳子集并进行聚类。

例如，如果已计算得到 p 个奇异向量, q 个 最佳得奇异向量可以被找出, 因为![q 。让 U 为列是 q 个最佳左奇异向量的矩阵, 并且 V 是用右奇异向量组成的矩阵. 为了划分行, 将 A 投影到 q 维空间: $A * V$。把 $m \times q$ 矩阵的 m 行作为样本, 然后使用 k-means 的聚类处理产生行标签。类似地，将列投影到 $A^\top * U$ ，并且对

$n \times q$ 矩阵进行聚类得到列标签。

> 示例：
>
> - A demo of the Spectral Biclustering algorithm: 一个简单的例子显示如何生成棋盘矩阵和对它进行双向聚类。
>
> 参考资料：
>
> - Kluger, Yuval, et. al., 2003. Spectral biclustering of microarray data: coclustering genes and conditions.

2.4.3. Biclustering 评价

有两种评估双聚类结果的方法：内部和外部。内部评估，如聚类稳定性， 依赖于数据和结果本身。目前在scikit-learn中没有内部的双向簇评估。外部评估是指外部信息来源，例如真实解。当处理真实数据时，真实解通常是未知的，但是，由于真实解是已知，因此人造数据的双向聚类可能对于评估算法非常有用。

为了将一组已发现的双向簇与一组真实的双向簇行比较，需要两个相似性度量：单个双向簇的相似性度量以及将这些个体相似度结合到总分中的度量。

为了比较单个双向簇，可以采用了几种措施。现在，只有Jaccard索引已被实现：

$$J(A, B) = \frac{|A \cap B|}{|A| + |B| - |A \cap B|}$$

其中 A 和 B 是双向簇， | A∩B | 是交叉点的元素的数量。

Jaccard 索引 达到最小值0，当biclusters完全不同时，Jaccard指数最小值为0;当biclusters完全相同时，Jaccard指数最大值为1。

一些方法已经开发出来，用来比较两个双向簇的集合(set)。从现在开始， 仅 `consensus_score` (Hochreiter et. al., 2010) 是可以用：

- 使用 Jaccard 索引或类似措施， 为每个集合中的一个双向簇对计算簇之间的相似性。
- 以一对一的方式将双向簇从一个集合分配给另一个集合，以最大化其相似性的总和。该步骤使用匈牙利算法(Hungarian algorithm)执行。
- 相似性的最终总和除以较大集合的大小。当所有双向簇对都完全不相似时，最小共识得分为0。当两个双向簇集合相同时，最大得分为1。

> 参考资料：
>
> - Hochreiter, Bodenhofer, et. al., 2010. FABIA: factor analysis for bicluster acquisition.

我们一直在努力

apachecn/sklearn-doc-zh

加入QQ群

2.5. 分解成分中的信号（矩阵分解问题）

校验者：@武器大师一个挑俩@png@barrycg翻译者：@柠檬@片刻

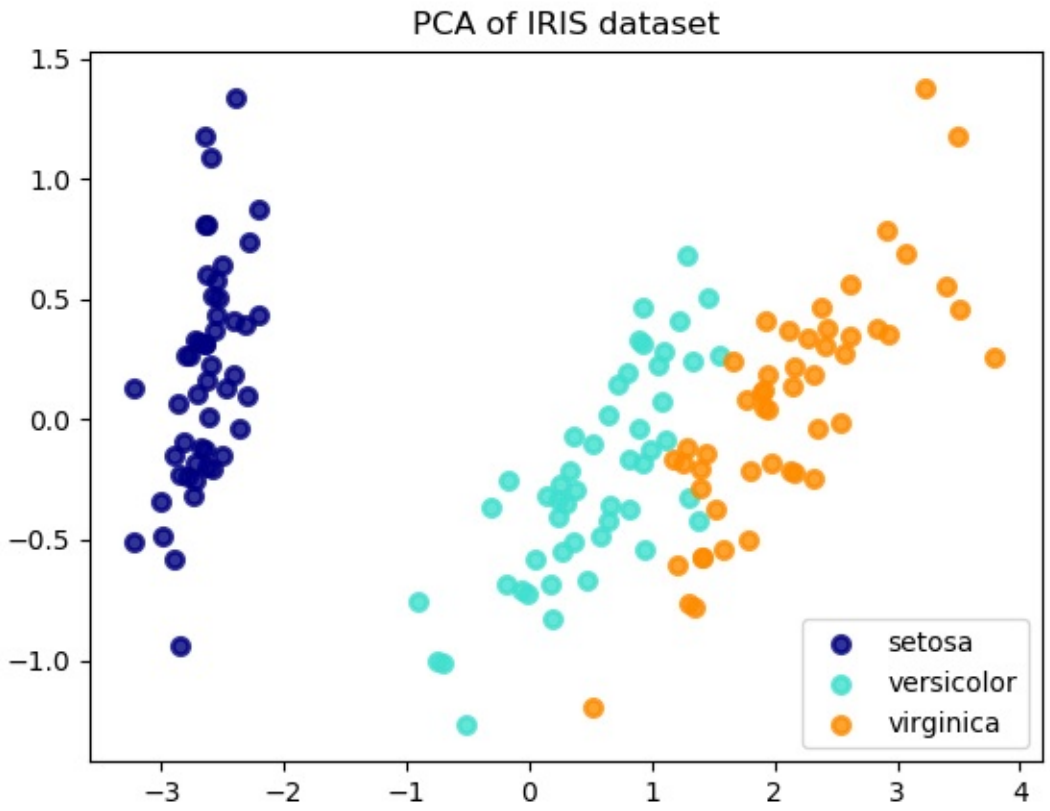
2.5.1. 主成分分析（PCA）

2.5.1.1. 准确的PCA和概率解释（Exact PCA and probabilistic interpretation）

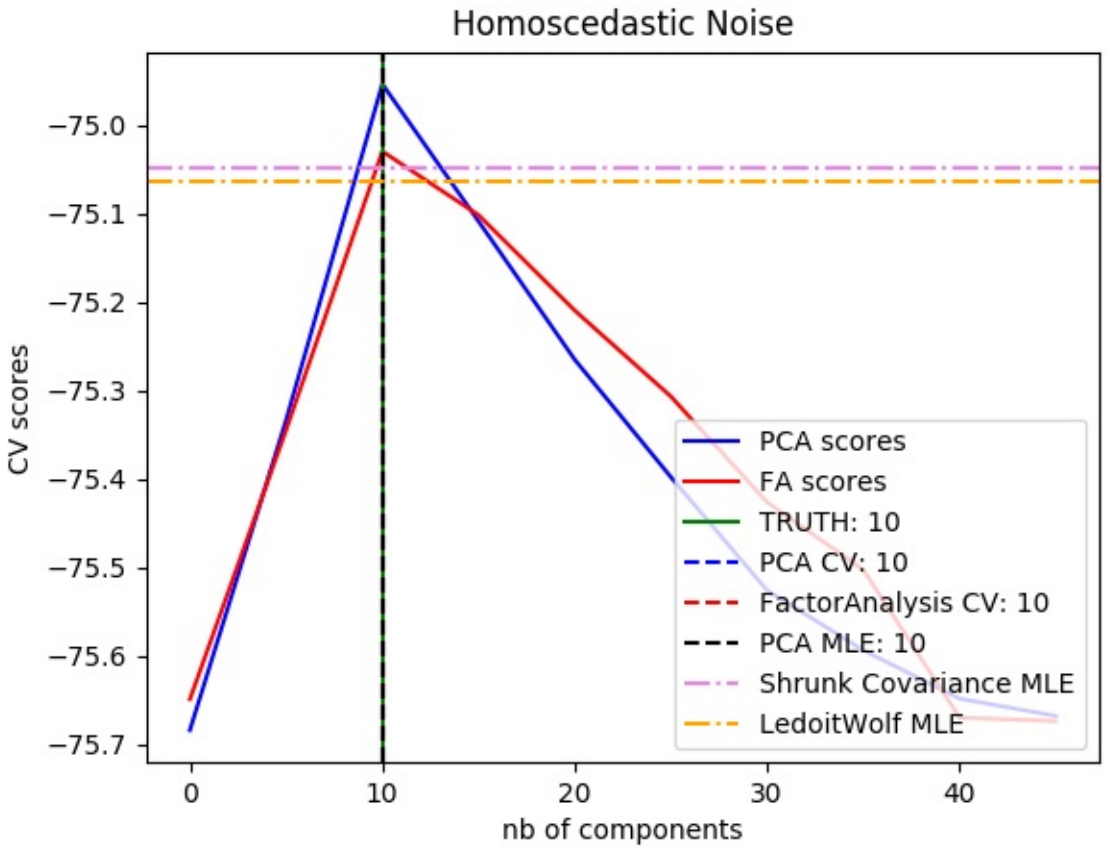
PCA 用于对具有一组连续正交分量(Orthogonal component 译注: 或译为正交成分,下出现 成分 和 分量 是同意词)的多变量数据集进行方差最大化的分解。 在 scikit-learn 中， `PCA` 被实现为一个变换器对象， 通过 `fit` 方法可以拟合出 *n* 个成分， 并且可以将新的数据投影(project, 亦可理解为分解)到这些成分中。

在应用SVD(奇异值分解) 之前， PCA 是在为每个特征聚集而不是缩放输入数据。可选参数 `whiten=True` 使得可以将数据投影到奇异（singular）空间上，同时将每个成分缩放到单位方差。 如果下游模型对信号的各向同性作出强假设，这通常是有用的，例如，使用RBF内核的 SVM 算法和 K-Means 聚类算法。

以下是iris数据集的一个示例，该数据集包含4个特征，通过PCA降维后投影到方差最大的二维空间上：



`PCA` 对象还提供了 PCA 算法的概率解释，其可以基于可解释的方差量给出数据的可能性。PCA对象实现了在交叉验证（cross-validation）中使用 *score* 方法：



> 示例：
>
> - Comparison of LDA and PCA 2D projection of Iris dataset
> - Model selection with Probabilistic PCA and Factor Analysis (FA)

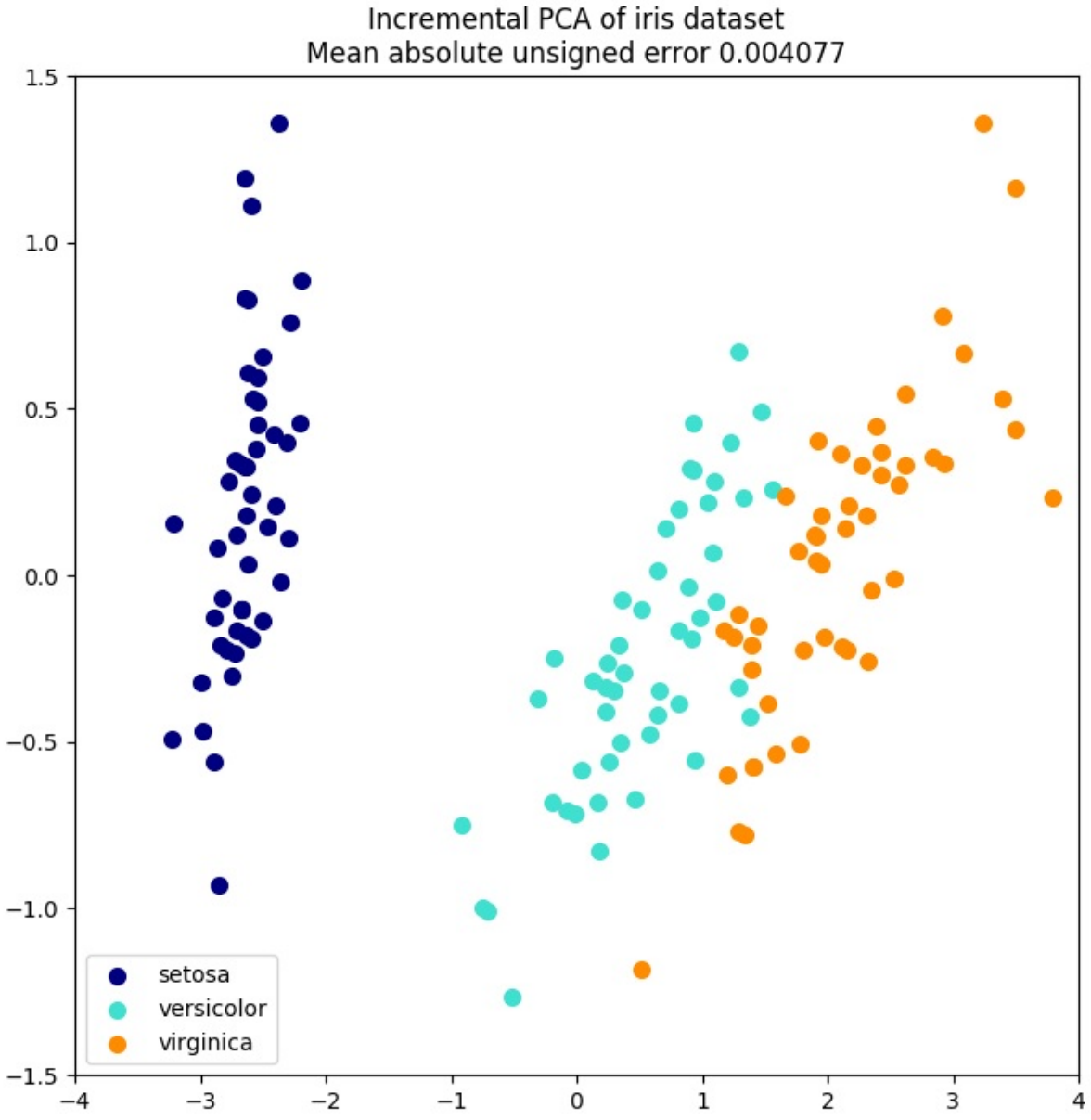
2.5.1.2. 增量PCA (Incremental PCA)

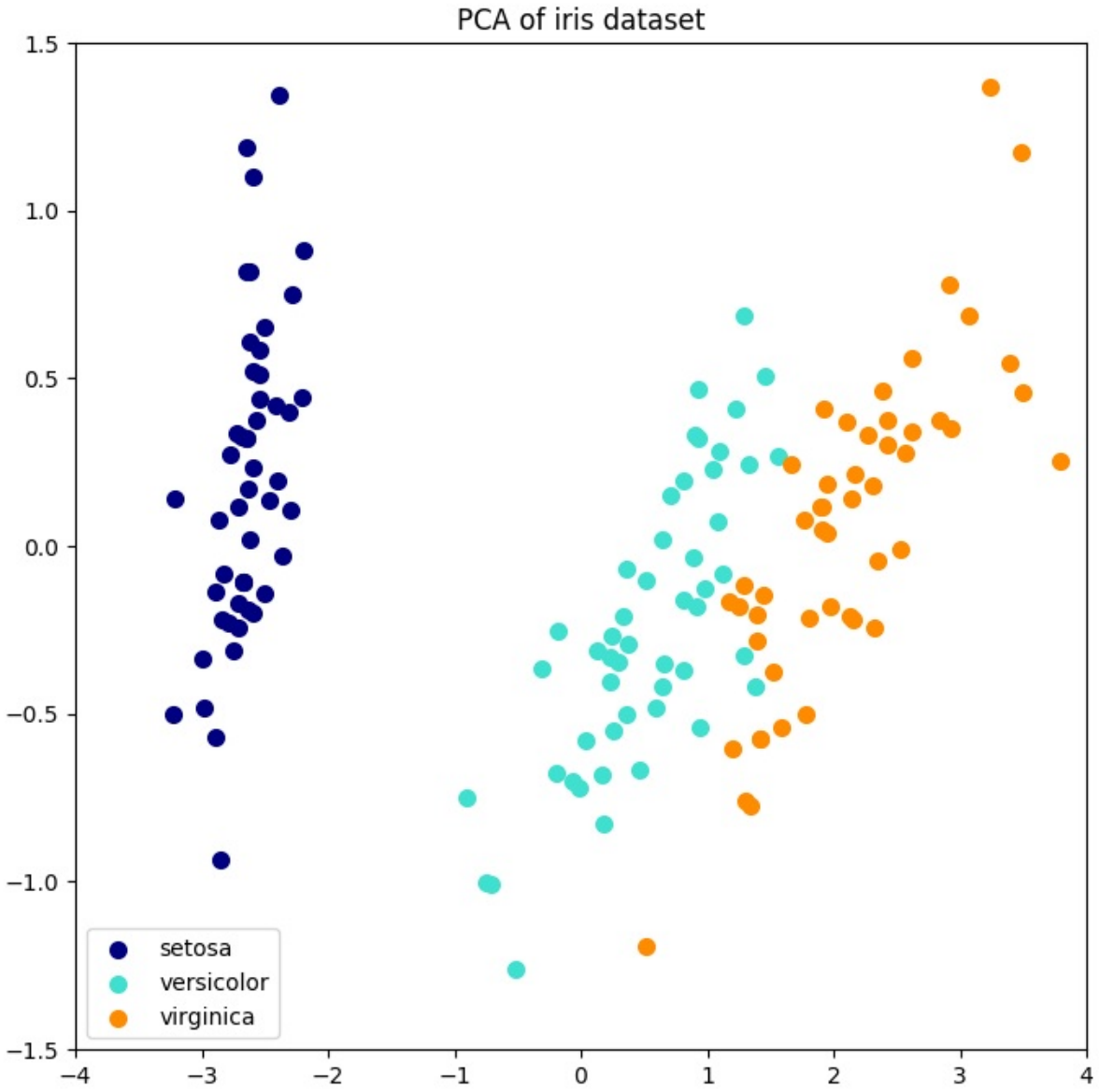
`PCA` 对象非常有用，但针对大型数据集的应用，仍然具有一定的限制。最大的限制是 `PCA` 仅支持批处理，这意味着所有要处理的数据必须放在主内存。`IncrementalPCA` 对象使用不同的处理形式，即允许部分计算以小型批处理方式处理数据的方法进行，而得到和 `PCA` 算法差不多的结果。`IncrementalPCA` 可以通过以下方式实现核外（out-of-core）主成分分析：

- 基于从本地硬盘或网络数据库中连续获取的数据块之上，使用 `partial_fit` 方法。
- 在 memory mapped file（通过 `numpy.memmap` 创建）上使用 fit 方法。

`IncrementalPCA` 类为了增量式的更新 explained*variance_ratio* ，仅需要存储估计出的分量和噪声方差。这就是为什么内存使用量依赖于每个批次的样本数量，而不是数据集中需要处理的样本总量。

在应用SVD之前，`IncrementalPCA` 就像 `PCA` 一样，为每个特征聚集而不是缩放输入数据。


Incremental PCA of iris dataset
Mean absolute unsigned error 0.004077
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−4
−3
−2
−1
0
1
2
3
4
setosa
versicolor
virginica



示例

- Incremental PCA

2.5.1.3. 基于随机化SVD的PCA

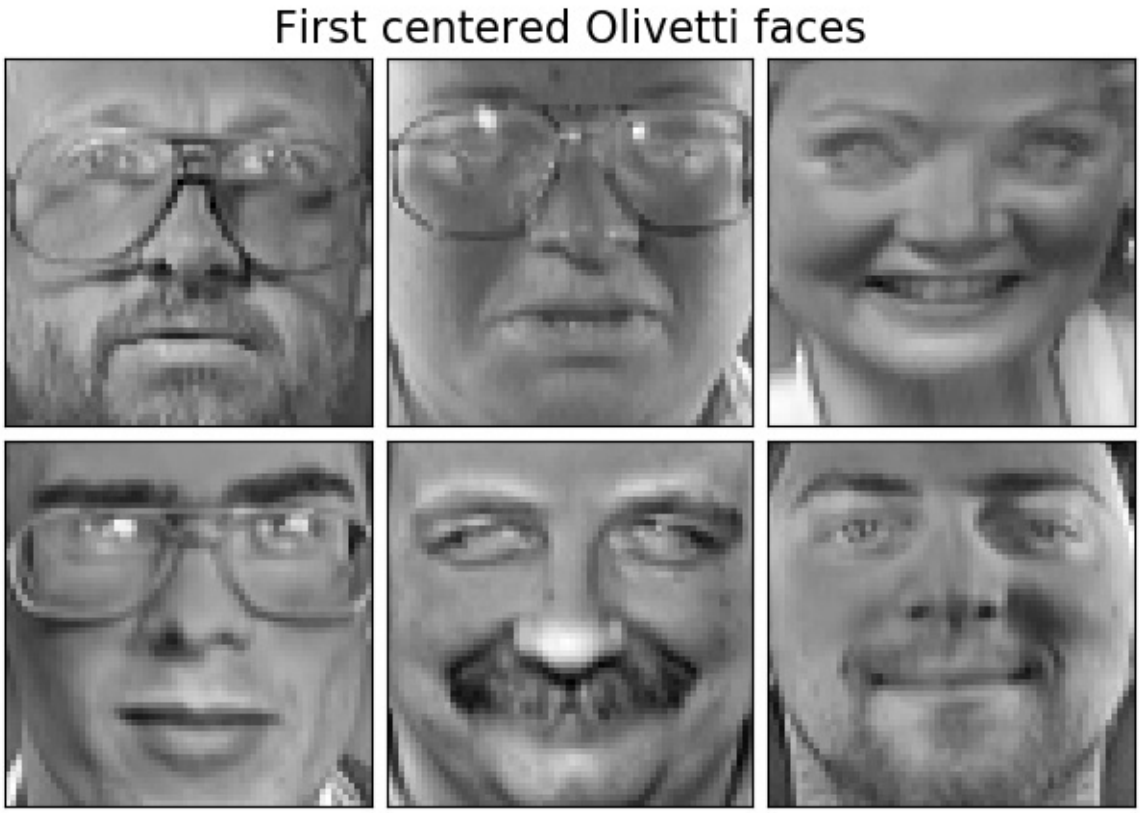
通过丢弃具有较低奇异值的奇异向量的分量，将数据降维到低维空间并保留大部分方差信息是非常有意义的。

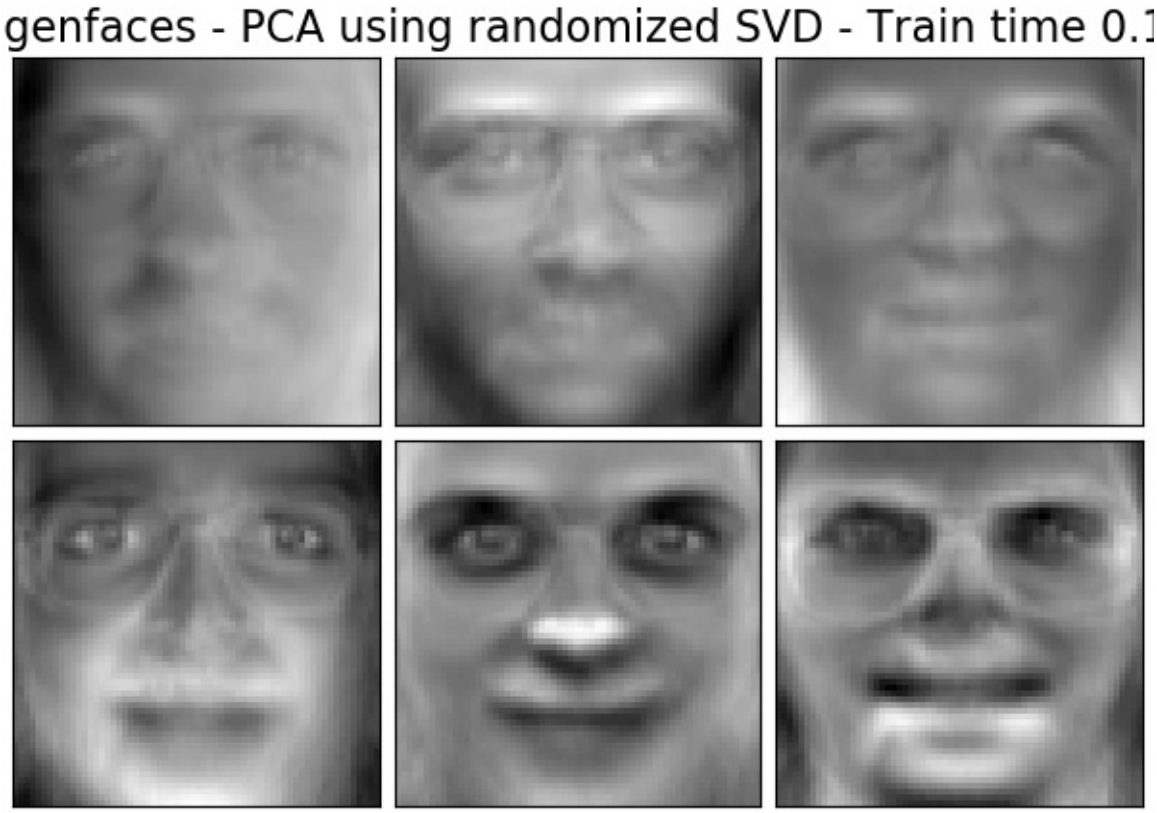
例如，如果我们使用64x64像素的灰度级图像进行人脸识别，数据的维数为4096，在这样大的数据上训练含RBF内核的支持向量机是很慢的。此外我们知道这些数据固有维度远低于4096，因为人脸的所有照片都看起来有点相似。样本位于较低维度的流体上（例如约200维）。 PCA算法可以用于线性变换数据，同时降低维数并同时保留大部分可描述的方差信息。

在这种情况下，使用可选参数 `svd_solver='randomized'` 的 `PCA` 是非常有用的。既然我们将要丢弃大部分奇异

值，那么仅仅就实际转换中所需的奇异向量进行计算就可以使得 PCA 计算过程变得异常有效。

例如：以下显示了来自 Olivetti 数据集的 16 个样本肖像（以 0.0 为中心）。右侧是前 16 个奇异向量重画的肖像。因为我们只需要使用大小为 $n_{samples} = 400$ 和 $n_{features} = 64 \times 64 = 4096$ 的数据集的前 16 个奇异向量，使得计算时间小于 1 秒。





注意：使用可选参数 `svd_solver='randomized'` ，在 `PCA` 中我们还需要给出输入低维空间大小 `n_components` 。

我们注意到，如果 $n_{\max} = \max(n_{\text{samples}}, n_{\text{features}})$ 且 $n_{\min} = \min(n_{\text{samples}}, n_{\text{features}})$ ，对于PCA中的实现，随机 `PCA` 的时间复杂度是： $O(n_{\max}^2 \cdot n_{\text{components}})$ ，而不是 $O(n_{\max}^2 \cdot n_{\min})$ 。

就内部实现的方法而言，随机 `PCA` 的内存占用量和 $2 \cdot n_{\max} \cdot n_{\text{components}}$ ，而不是 $n_{\max} \cdot n_{\min}$ 成正比。

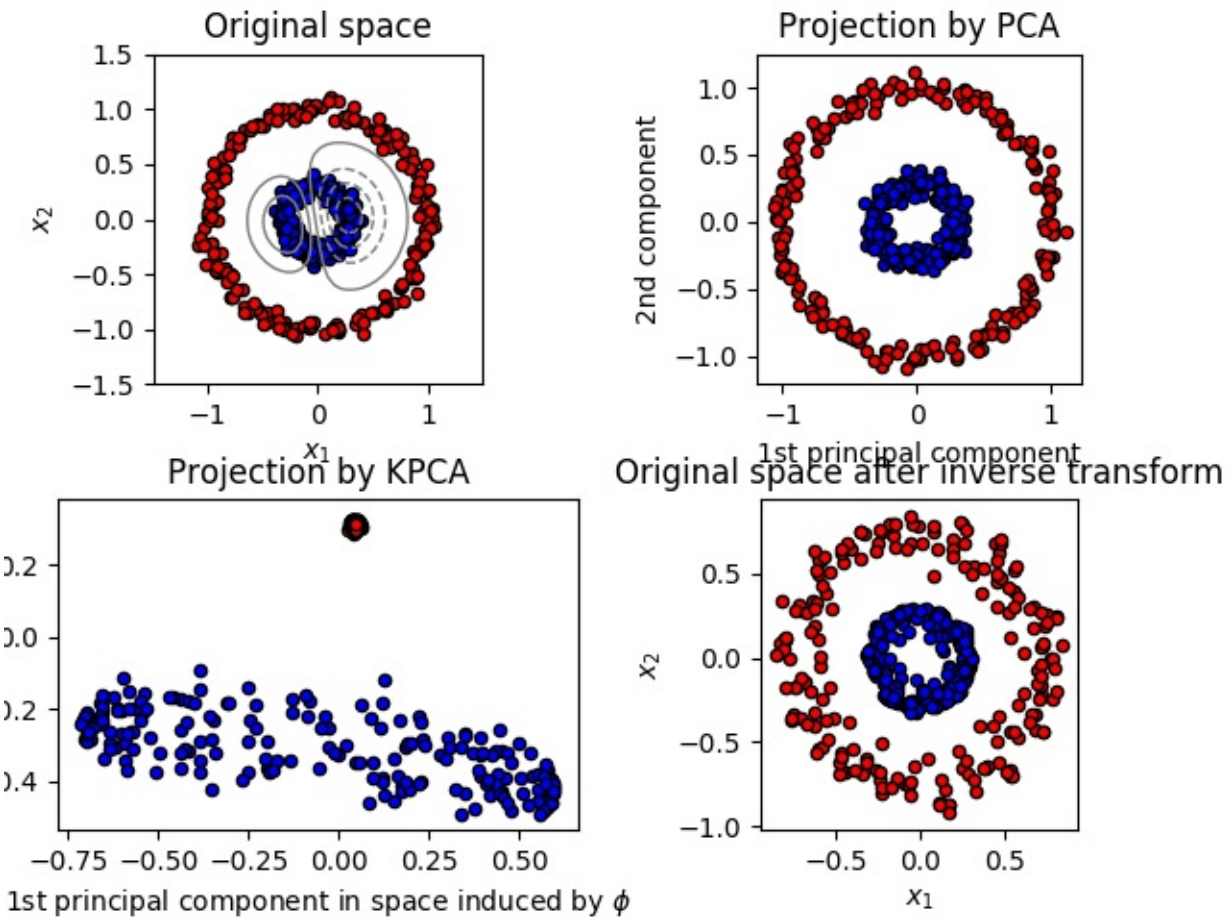
注意：选择参数 `svd_solver='randomized'` 的 `PCA` 的 `inverse_transform` 的实现，并不是对应 `transform` 的逆变换（即使 参数设置为默认的 `whiten=False` ）

> 示例：
>
> - Faces recognition example using eigenfaces and SVMs
> - Faces dataset decompositions
>
> 参考资料：
>
> - “Finding structure with randomness: Stochastic algorithms for constructing approximate matrix decompositions” Halko, et al., 2009

2.5.1.4. 核 PCA

`KernelPCA` 是 PCA 的扩展，通过使用核方法实现非线性降维（dimensionality reduction）（参阅 成对的矩阵，类别和核函数）。 它具有许多应用，包括去噪，压缩和结构化预测（ structured prediction ）

(kernel dependency estimation（内核依赖估计）)。 `KernelPCA` 支持 `transform` 和 `inverse_transform` 。



> 示例：
>
> - Kernel PCA

2.5.1.5. 稀疏主成分分析（ SparsePCA 和 MiniBatchSparsePCA ）

`SparsePCA` 是 PCA 的一个变体，目的是提取能最大程度得重建数据的稀疏分量集合。

小批量稀疏 PCA（ `MiniBatchSparsePCA` ）是一个 `SparsePCA` 的变体，它速度更快但准确度有所降低。对于给定的迭代次数，通过迭代该组特征的小块来达到速度的增加。

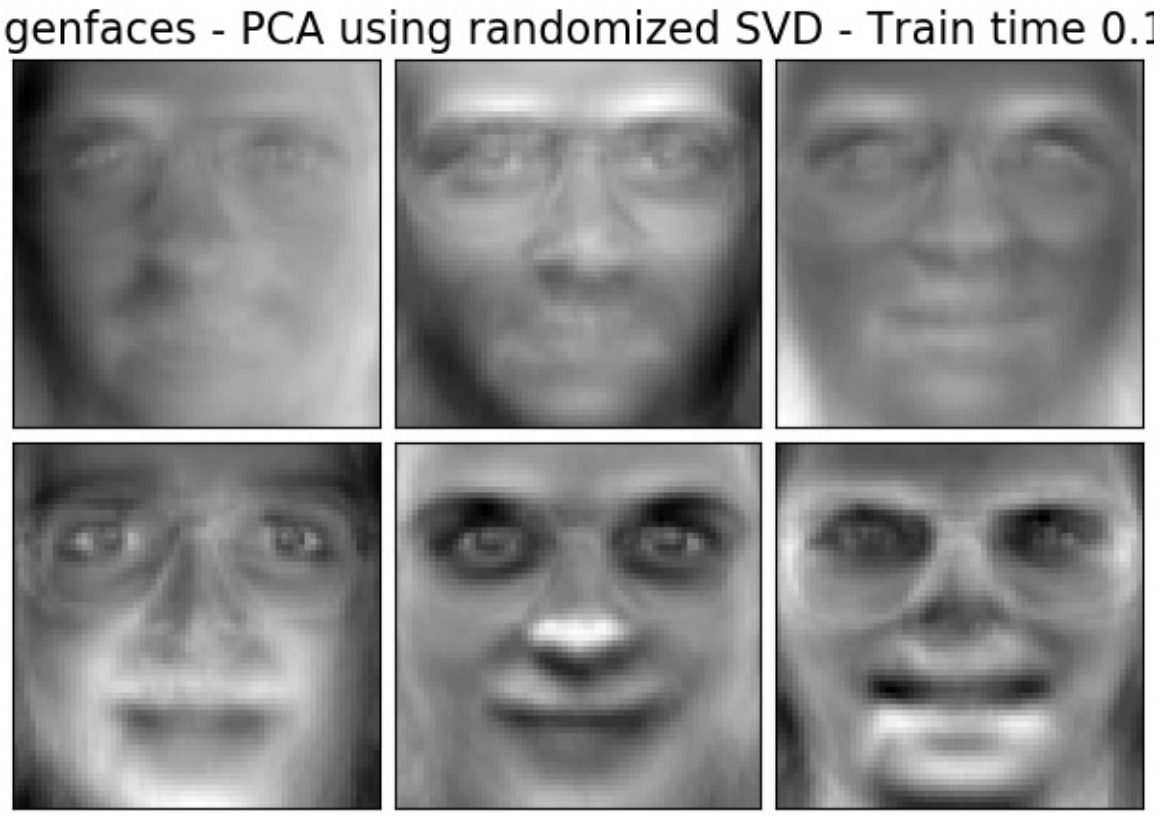
Principal component analysis（主成分分析）（ `PCA` ）的缺点在于，通过该方法提取的成分具有独占的密度表达式，即当表示为原始变量的线性组合时，它们具有非零系数，使之难以解释。在许多情况下，真正的基础分量可以被更自然地想象为稀疏向量；例如在面部识别中，每个分量可能自然地映射到面部的某个部分。

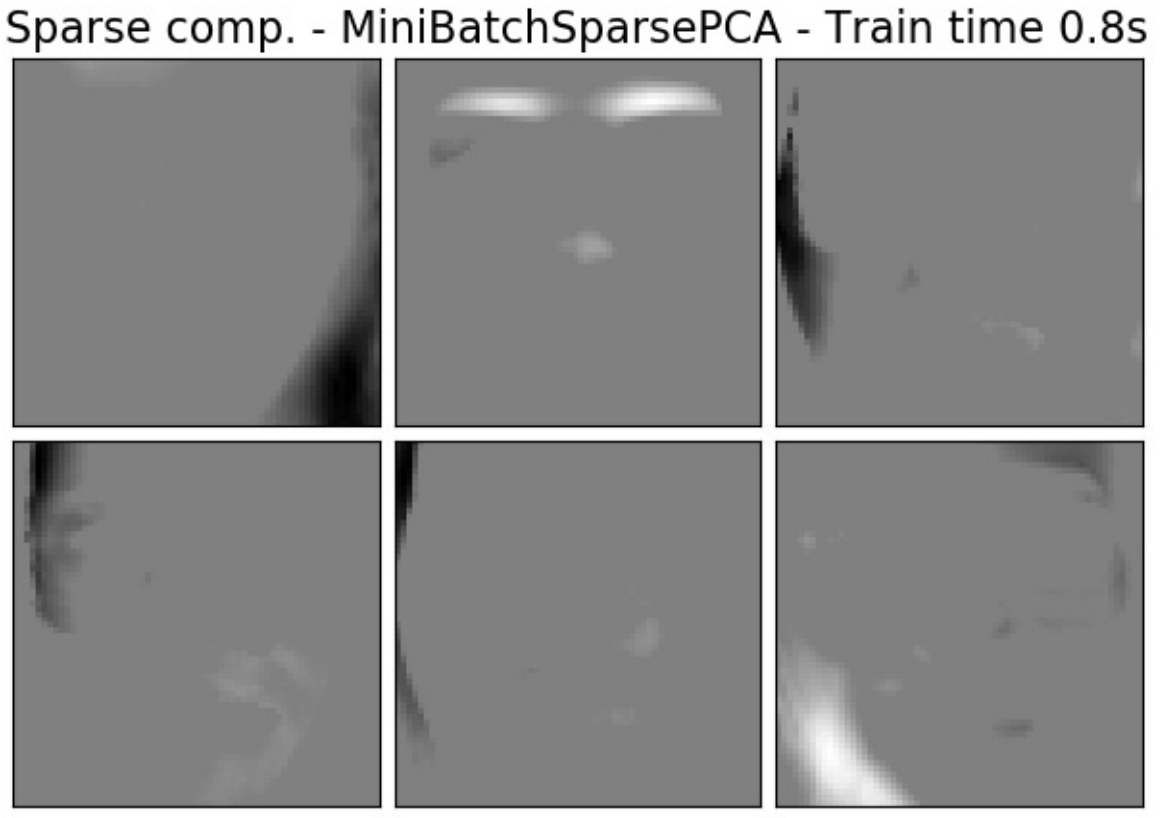
稀疏的主成分产生更节约、可解释的表达式，明确强调了样本之间的差异性来自哪些原始特征。

以下示例说明了使用稀疏 PCA 提取 Olivetti 人脸数据集中的 16 个分量。可以看出正则化项产生了许多零。此外，数据的自然结构导致了非零系数垂直相邻（vertically adjacent）。该模型并不具备纯数学意义的执行：

每个分量都是一个向量 $h \in \mathbf{R}^{4096}$ ，除非人性化地的可视化为 64x64 像素的图像，否则没有垂直相邻性的概念。

下面显示的分量看起来局部化（appear local)是数据的内在结构的影响，这种局部模式使重建误差最小化。有一种考虑到邻接性和不同结构类型的导致稀疏的规范（sparsity-inducing norms），参见 [Jen09] 对这种方法进行了解。有关如何使用稀疏 PCA 的更多详细信息，请参阅下面的示例部分。





请注意，有多种不同的计算稀疏PCA 问题的公式。 这里使用的方法基于 [Mrl09] 。对应优化问题的解决是一个带

有惩罚项（L1范数的）ℓ_1 的 PCA 问题（dictionary learning（字典学习））：

$$(U^*, V^*) = \arg\min_{U,V} \frac{1}{2}||X - UV||_2^2 + \alpha||V||_1$$
$$\text{subject to } ||U_k||_2 = 1 \text{ for all } 0 \leq k < n_{components}$$

稀疏推导（sparsity-inducing）ℓ_1 规范也可以当训练样本很少时，避免从噪声中拟合分量。可以通过超参数 `alpha` 来调整惩罚程度（或称稀疏度）。值较小会导致温和的正则化因式分解，而较大的值将许多系数缩小到零。

> 注意虽然本着在线算法的精神，`MiniBatchSparsePCA` 类不实现 `partial_fit`，因为在线算法是以特征为导向，而不是以样本为导向。
>
> 示例：
>
> - Faces dataset decompositions
>
> 参考资料：
>
> - [Mrl09] “Online Dictionary Learning for Sparse Coding” J. Mairal, F. Bach, J. Ponce, G. Sapiro, 2009
> - [Jen09] “Structured Sparse Principal Component Analysis” R. Jenatton, G. Obozinski, F. Bach, 2009

2.5.2. 截断奇异值分解和隐语义分析

`TruncatedSVD` 实现了一个奇异值分解（SVD）的变体，它只计算 k 个最大的奇异值，其中 k 是用户指定的参数。

当截断的 SVD被应用于 term-document矩阵（由 `CountVectorizer` 或 `TfidfVectorizer` 返回）时，这种转换被称为 latent semantic analysis (LSA)，因为它将这样的矩阵转换为低维度的"语义"空间。特别地是 LSA 能够抵抗同义词和多义词的影响（两者大致意味着每个单词有多重含义），这导致 term-document 矩阵过度稀疏，并且在诸如余弦相似性的度量下表现出差的相似性。

> 注意LSA 也被称为隐语义索引 LSI，尽管严格地说它是指用于信息检索的持久索引（persistent indexes）。

从数学角度来说，训练样本 X 用截断的SVD 会产生一个低秩的（ low-rank）近似值 X：

$$X \approx X_k = U_k \Sigma_k V_k^\top$$

在这个操作之后，$U_k \Sigma_k^\top$ 是转换后的训练集，其中包括 k 个特征（在 API 中被称为 `n_components` ）。

还需要转换一个测试集 X，我们乘以 V_k：

$$X' = X V_k$$

> 注意自然语言处理(NLP) 和信息检索(IR) 文献中的 LSA 的大多数处理方式是交换矩阵 X 的坐标轴，使其具有 `n_features` × `n_samples` 的形状。我们在 scikit-learn API 使用不同方式呈现 LSA，找到的奇异值是相同的。

`TruncatedSVD` 非常类似于 `PCA` ，但不同之处在于它应用于样本矩阵 X 而不是它们的协方差矩阵。当从特征值中减去 X 的每列（每个特征per-feature）的均值时，在得到的矩阵上应用 truncated SVD 相当于 PCA 。实际上，这意味着 `TruncatedSVD` 转换器（transformer）接受 `scipy.sparse` 矩阵，而不需要对它们进行密集（densify），因为即使对于中型大小文档的集合，密集化 （densifying）也可能填满内存。

虽然 `TruncatedSVD` 转换器（transformer）可以在任何（稀疏的）特征矩阵上工作，但还是建议在 LSA/document 处理背景中，在 tf–idf 矩阵上的原始频率计数使用它。特别地，应该打开子线性缩放（sublinear scaling）和逆文档频率（inverse document frequency）（ `sublinear_tf=True, use_idf=True` ）以使特征值更接近于高斯分布，补偿 LSA 对文本数据的错误假设。

> 示例：
>
> - Clustering text documents using k-means
>
> 参考资料：
>
> - Christopher D. Manning, Prabhakar Raghavan and Hinrich Schütze (2008), *Introduction to Information Retrieval*, Cambridge University Press, chapter 18: Matrix decompositions & latent semantic indexing

2.5.3. 词典学习

2.5.3.1. 带有预计算词典的稀疏编码

`SparseCoder` 对象是一个估计器 （estimator），可以用来将信号转换成一个固定的,预计算的词典内原子（atoms）的稀疏线性组合（sparse linear combination），如离散小波基（ discrete wavelet basis）。 因此，该对象不实现 `fit` 方法。该转换相当于一个稀疏编码问题：将数据的表示为尽可能少的词典原子的线性组合。 词典学习的所有变体实现以下变换方法，可以通过 `transform_method` 初始化参数进行控制：

- Orthogonal matching pursuit(追求正交匹配) (正交匹配追踪（OMP）)
- Least-angle regression (最小角度回归)(最小角回归)
- Lasso computed by least-angle regression(最小角度回归的Lasso 计算)
- Lasso using coordinate descent （ 使用坐标下降的Lasso)(Lasso)
- Thresholding(阈值)

阈值方法速度非常快，但是不能产生精确的重建。 它们在分类任务的文献中已被证明是有用的。对于图像重建任务，追求正交匹配可以产生最精确、无偏的重建。

词典学习对象通过 `split_code` 参数提供稀疏编码结果中的正值和负值分离的可能性。当使用词典学习来提取将用于监督学习的特征时，这是有用的，因为它允许学习算法将不同的权重从正加载（loading）分配给相应的负加载的特定原子。

单个样本的分割编码具有长度 `2 * n_components` ，并使用以下规则构造：首先，计算长度为 `n_components` 的常规编码。然后， `split_code` 的第一个 `n_components` 条目将用常规编码向量的正部分填充。分割编码的第二部分用编码向量的负部分填充，只有一个正号。因此， split_code 是非负的。

> 示例：
>
> - Sparse coding with a precomputed dictionary

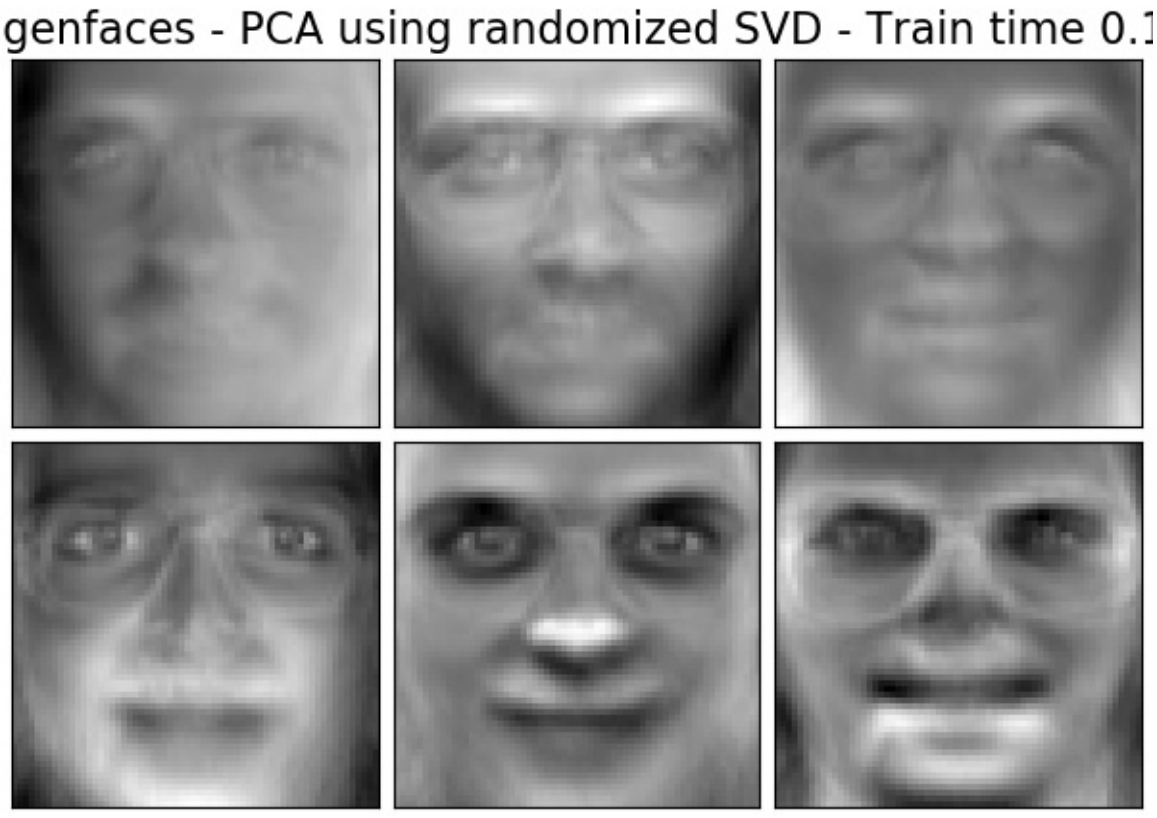
2.5.3.2. 通用词典学习

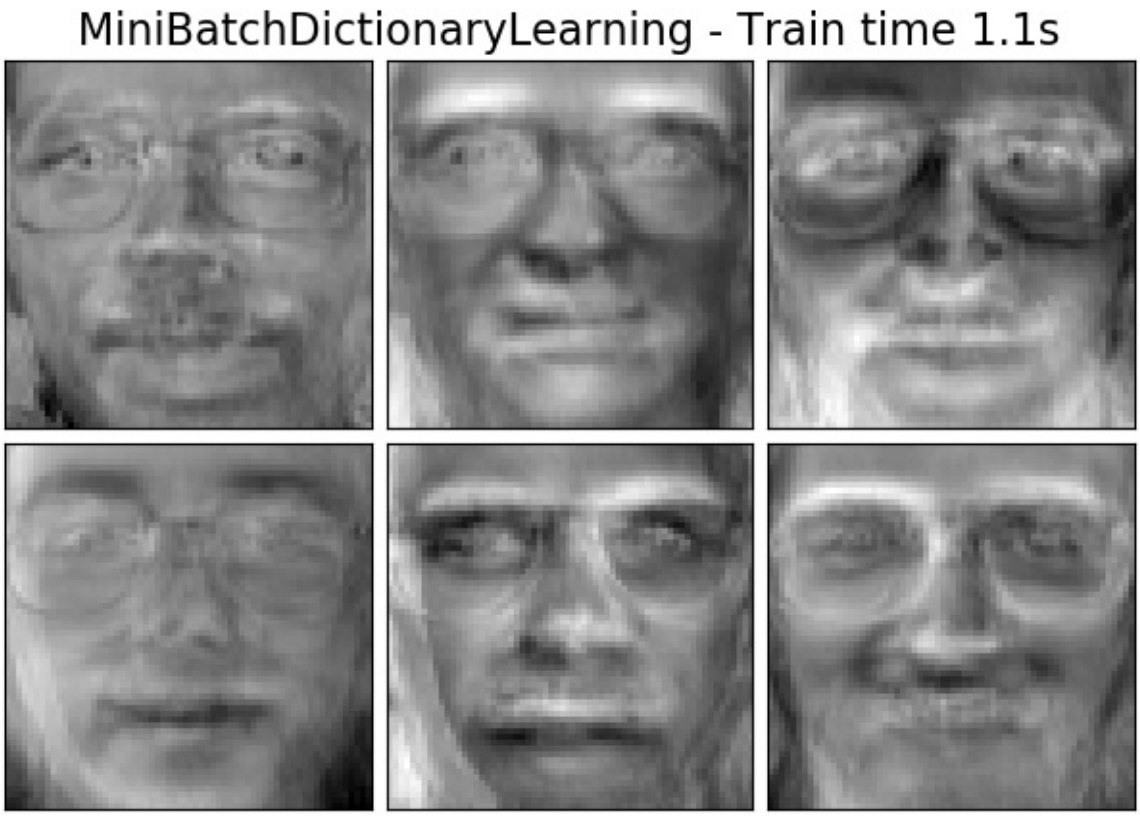
词典学习(`DictionaryLearning`) 是一个矩阵因式分解问题，相当于找到一个在拟合数据的稀疏编码中表现良好的（通常是过完备的（`overcomplete`））词典。

使用过完备词典的原子的稀疏组合来表示数据被认为是哺乳动物初级视觉皮层的工作方式。 因此，应用于图像补丁的词典学习已被证明在诸如图像完成、修复和去噪，以及有监督的识别图像处理任务中表现良好的结果。

词典学习被视为通过交替更新稀疏编码来解决的优化问题，也作为解决多个 `Lasso` 问题的一个解决方案，考虑到字典固定，可以通过更新字典以最大程度的拟合出稀疏编码。

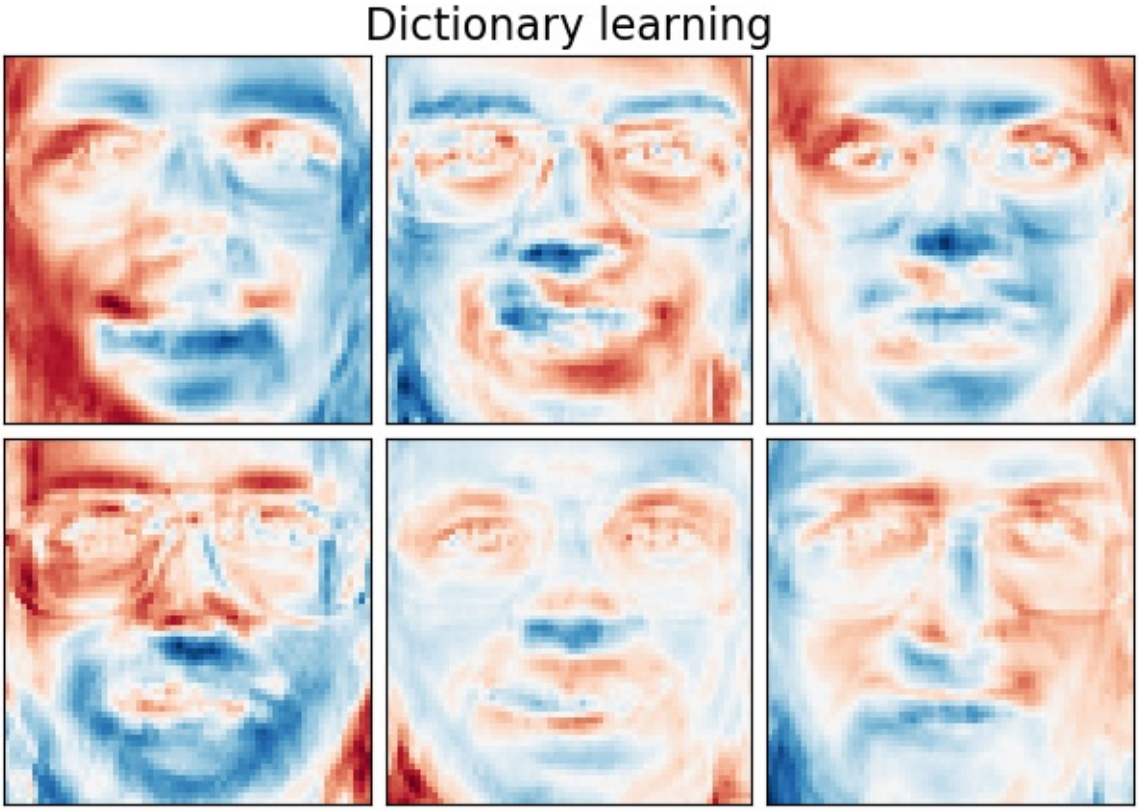
$$(U^*, V^*) = \arg\min_{U,V} \frac{1}{2}||X - UV||_2^2 + \alpha||U||_1$$
$$\text{subject to} ||V_k||_2 = 1 \text{ for all } 0 \leq k < n_{\text{atoms}}$$

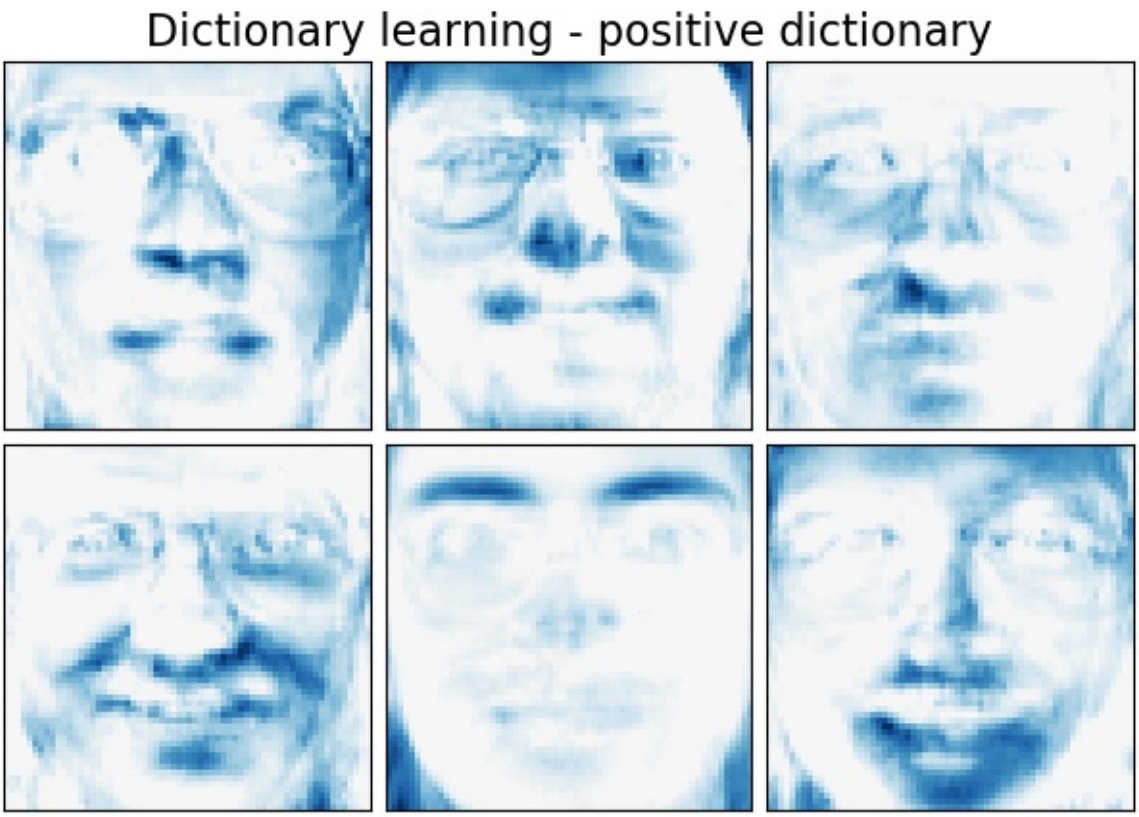


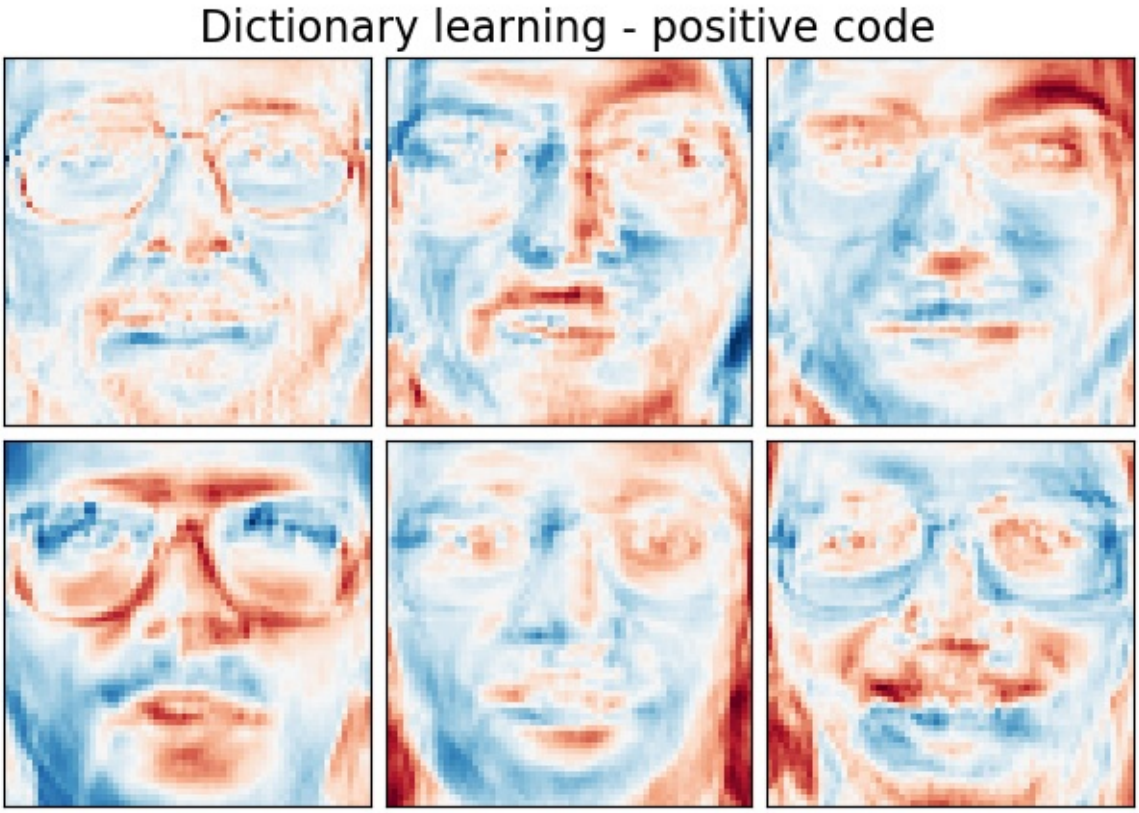


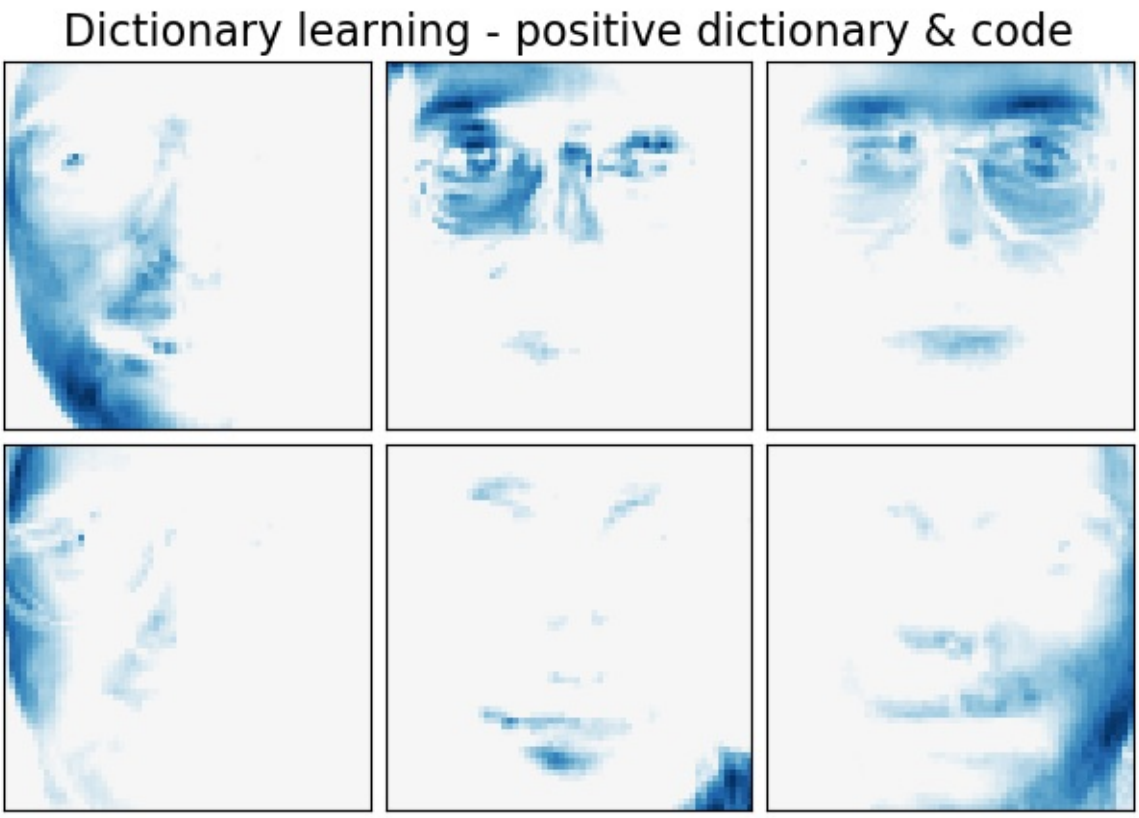
在使用这样一个过程来拟合词典之后，变换只是一个稀疏编码的步骤，与所有的词典学习对象共享相同的实现。（参见带有预计算词典的稀疏编码）。

也可以通过限制字典和/或编码为正来匹配数据中可能表现的约束。以下是正约束应用的脸部图像。红色表示负值， 蓝色表示正值，白色表示零值。

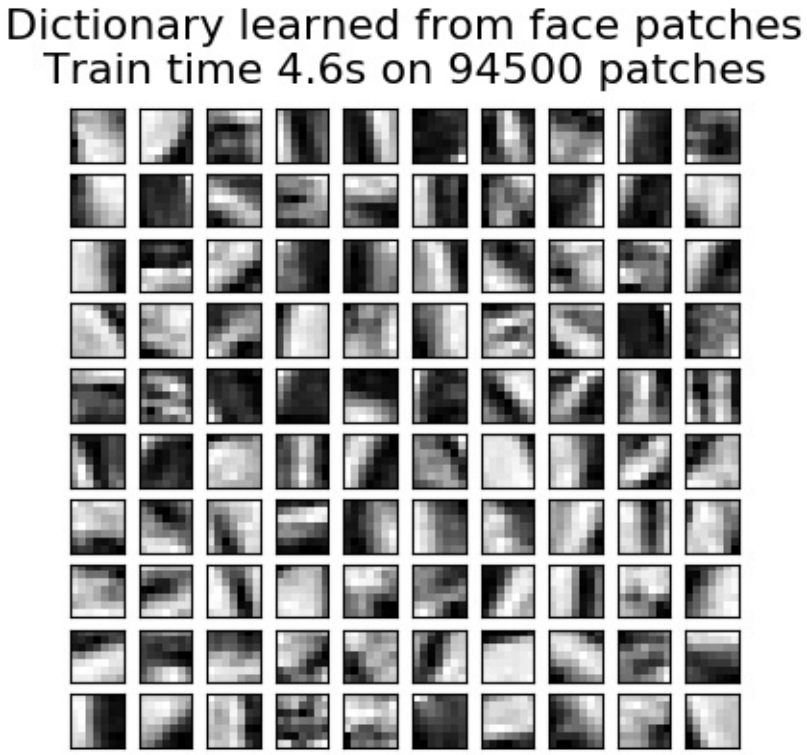

Dictionary learning


Dictionary learning - positive dictionary





以下图像显示了字典学习是如何从浣熊脸部的部分图像中提取的4x4像素图像补丁中进行词典学习的。



示例：

- Image denoising using dictionary learning

参考资料：

- “Online dictionary learning for sparse coding” J. Mairal, F. Bach, J. Ponce, G. Sapiro, 2009

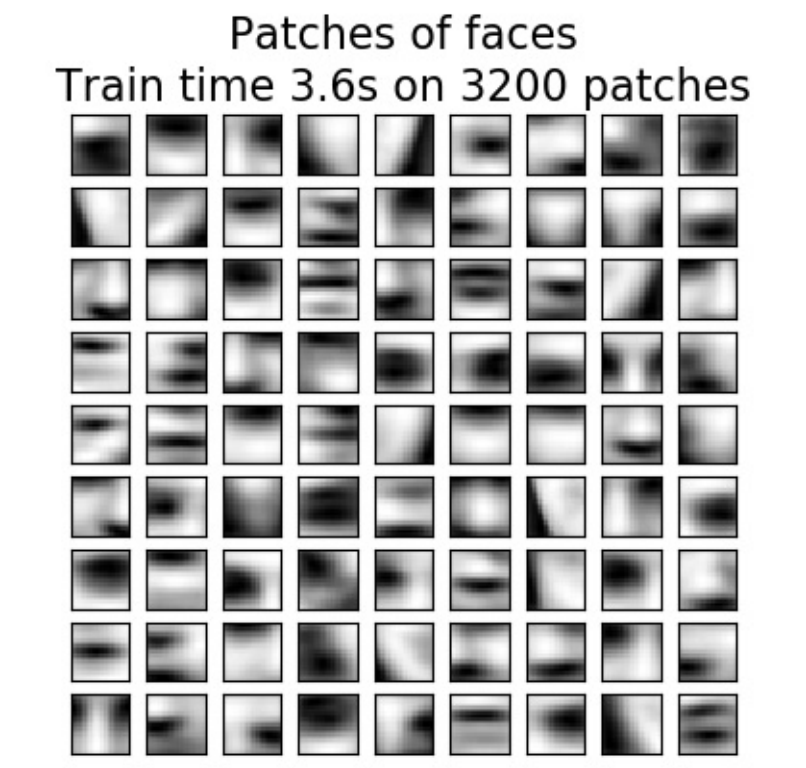
2.5.3.3. 小批量字典学习

`MiniBatchDictionaryLearning` 实现了更快、更适合大型数据集的字典学习算法，其运行速度更快，但准确度有所降低。

默认情况下， `MiniBatchDictionaryLearning` 将数据分成小批量，并通过在指定次数的迭代中循环使用小批量，以在线方式进行优化。但是，目前它没有实现停止条件。

估计器还实现了 `partial_fit` ，它通过在一个小批处理中仅迭代一次来更新字典。 当在线学习的数据从一开始就不容易获得，或者数据超出内存时，可以使用这种迭代方法。

字典学习聚类



注意，当使用字典学习来提取表示（例如，用于稀疏编码）时，聚类可以是学习字典的良好中间方法。 例如， `MiniBatchKMeans` 估计器能高效计算并使用 `partial_fit` 方法实现在线学习。

示例：在线学习面部部分的字典 Online learning of a dictionary of parts of faces

2.5.4. 因子分析

在无监督的学习中，我们只有一个数据集 $X = \{x_1, x_2, \ldots, x_n\}$. 这个数据集如何进行数学描述？ X 的一个非常简单的 连续隐变量模型

$$x_i = W h_i + \mu + \epsilon$$

矢量 h_i 被称为“隐性的”，因为它是不可观察的。 ϵ 被认为是符合高斯分布的噪声项，平均值为 0，协方差为 Ψ（即 $\epsilon \sim \mathcal{N}(0, \Psi)$），$\mu$ 是偏移向量。 这样一个模型被称为“生成的”，因为它描述了如何从 h_i 生成 x_i。如果我们使用所有的 x_i 作为列来形成一个矩阵 $\mathbf{X}$，并将所有的 h_i 作为矩阵 $\mathbf{H}$ 的列， 那么我们可以写（适当定义的 $\mathbf{M}$ 和 $\mathbf{E}$）：

$$\mathbf{X} = W\mathbf{H} + \mathbf{M} + \mathbf{E}$$

换句话说，我们 分解 矩阵 $\mathbf{X}$. 如果给出 h_i，上述方程自动地表示以下概率解释：

$$p(x_i|h_i) = \mathcal{N}(W h_i + \mu, \Psi)$$

对于一个完整的概率模型，我们还需要隐变量 h 的先验分布。 最直接的假设（基于高斯分布的良好性质）是 $h \sim \mathcal{N}(0,\mathbf{I})$ 。这产生一个高斯分布作为 x 的边际分布：

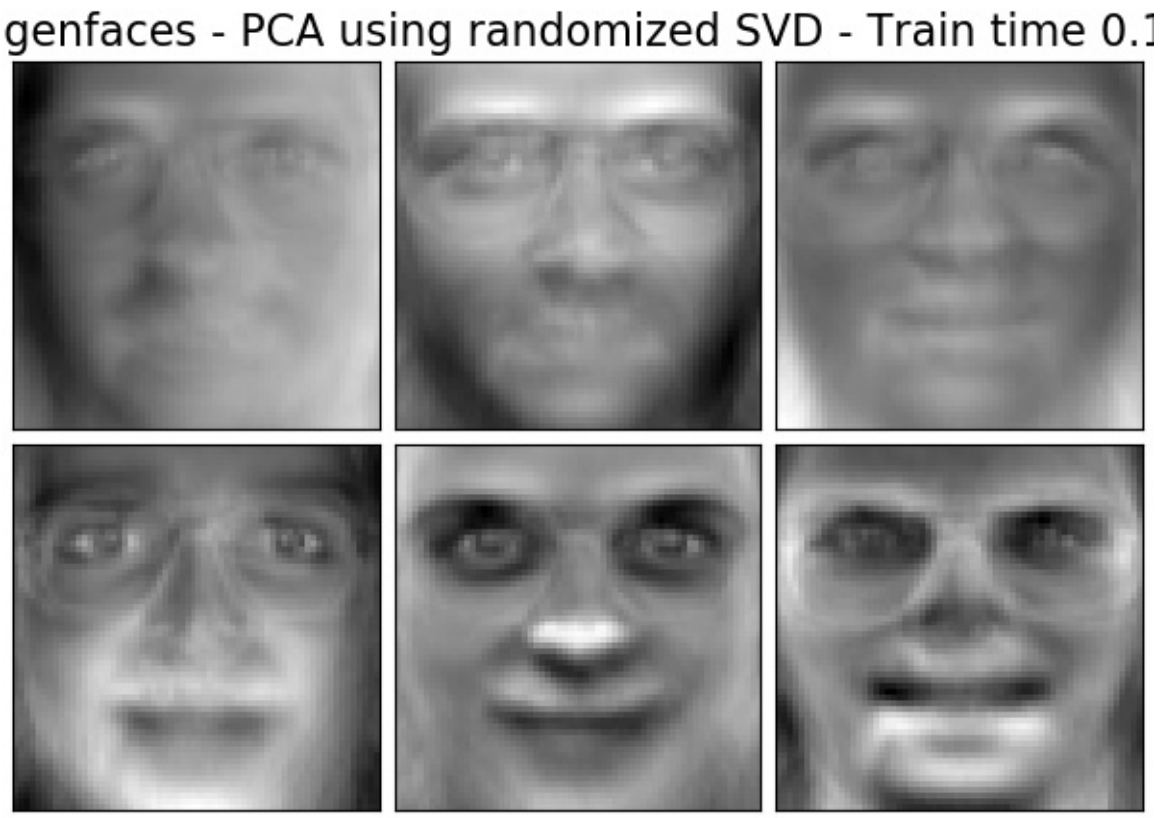
$$p(x) = \mathcal{N}(\mu, WW^T + \Psi)$$

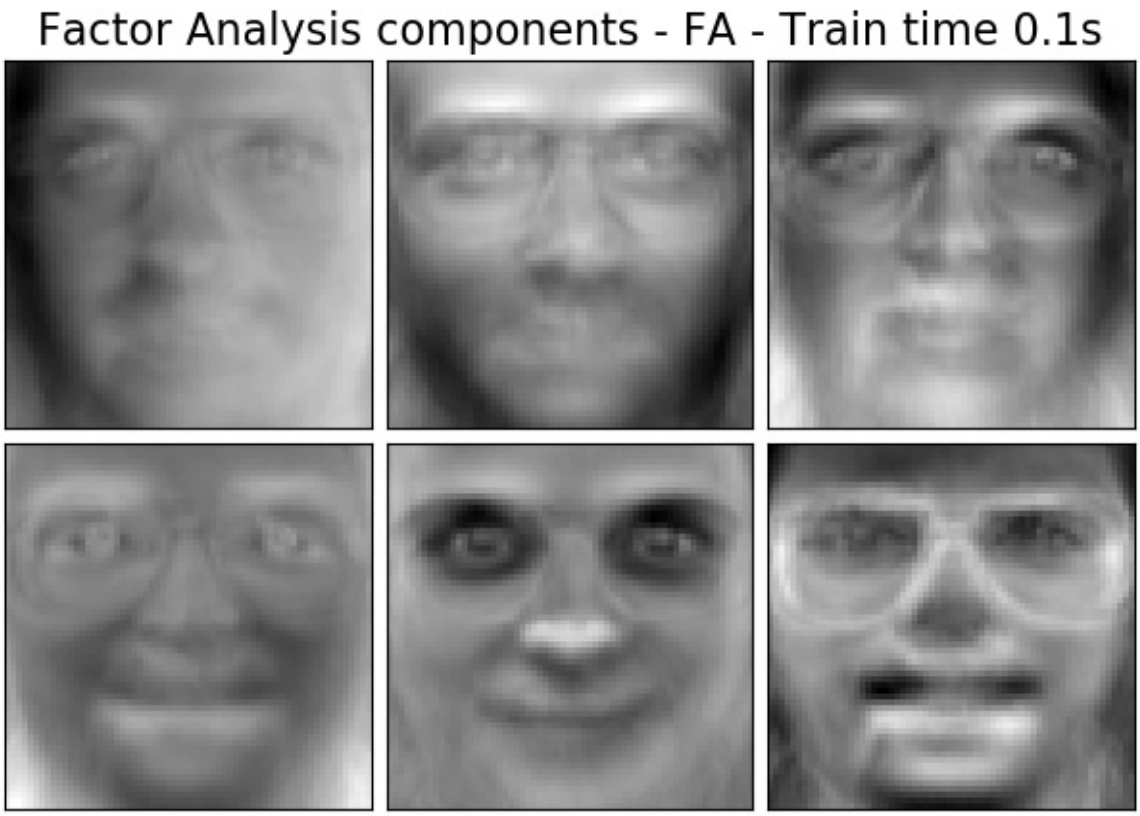
现在，在没有任何进一步假设的前提下，隐变量 h 是多余的 - x 完全可以用均值和协方差来建模。 我们需要对这两个参数之一进行更具体的构造。 一个简单的附加假设是将误差协方差 Ψ 构造成如下：

- $\Psi = \sigma^2\mathbf{I}$ ：这个假设能推导出 `PCA` 的概率模型。
- $\Psi = \mathrm{diag}(\psi_1, \psi_2, \ldots, \psi_n)$ ：这个模型称为 `FactorAnalysis` ，一个经典的统计模型。 矩阵W有时称为“因子加载矩阵”。

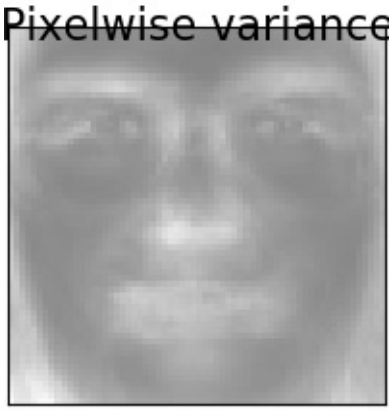
两个模型基都可以使用低阶协方差矩阵来估计一个高斯分布。因为这两个模型都是概率性的，所以它们可以集成到更复杂的模型中，例如因子分析器的混合。如果隐变量基于非高斯分布的先验，则得到完全不同的模型（例如， `FastICA` ）。

因子分析 可以 产生与 `PCA` 类似的成分（例如其加载矩阵的列）。 然而，这些成分没有通用的性质（例如它们是否是正交的）：

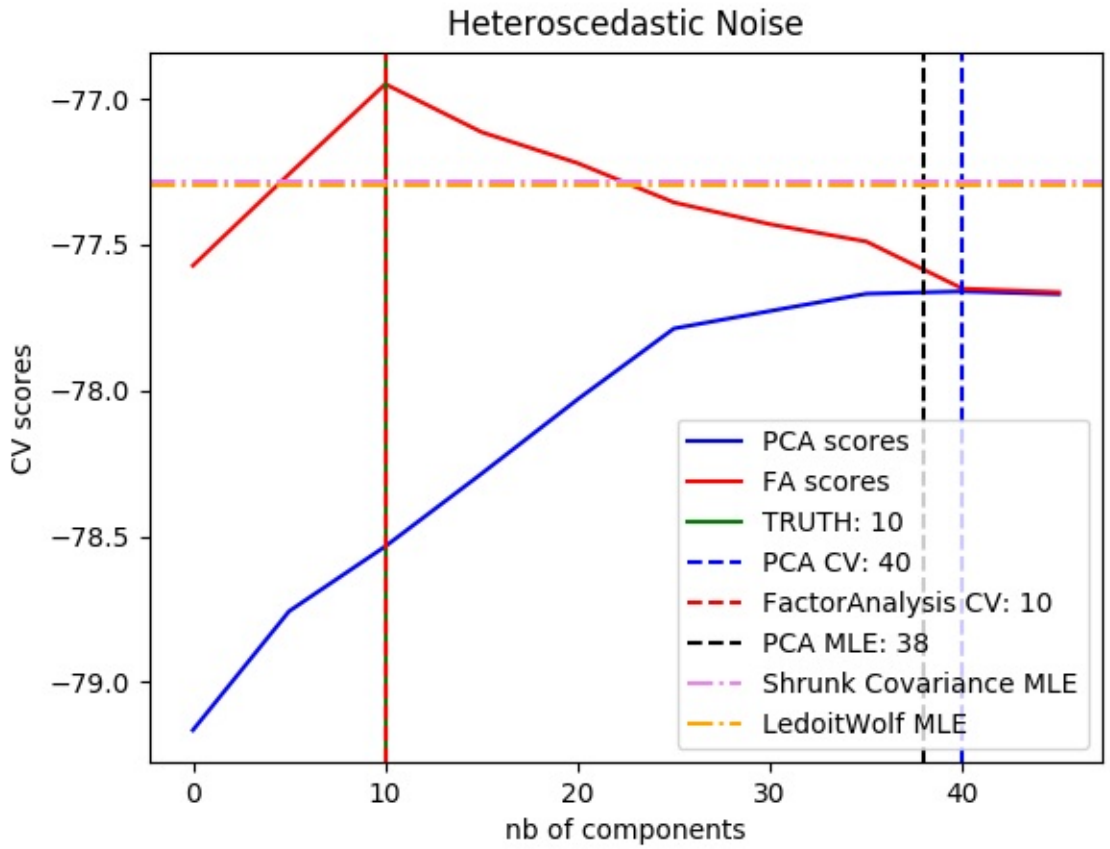




因子分析(PCA) 的主要优点是可以独立地对输入空间的每个方向（异方差噪声）的方差进行建模：



在异方差噪声存在的情况下，这可以比概率 PCA 作出更好的模型选择：



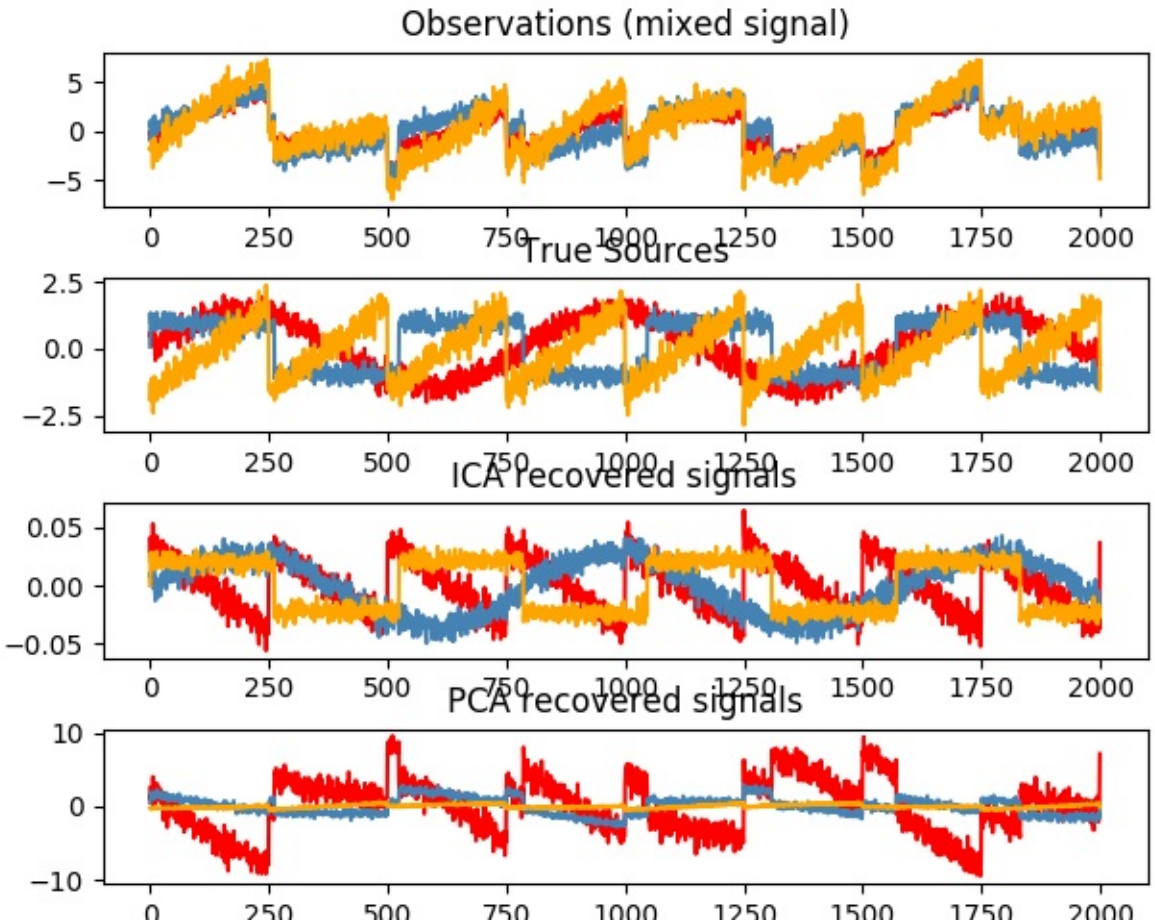
示例：

- Model selection with Probabilistic PCA and Factor Analysis (FA)

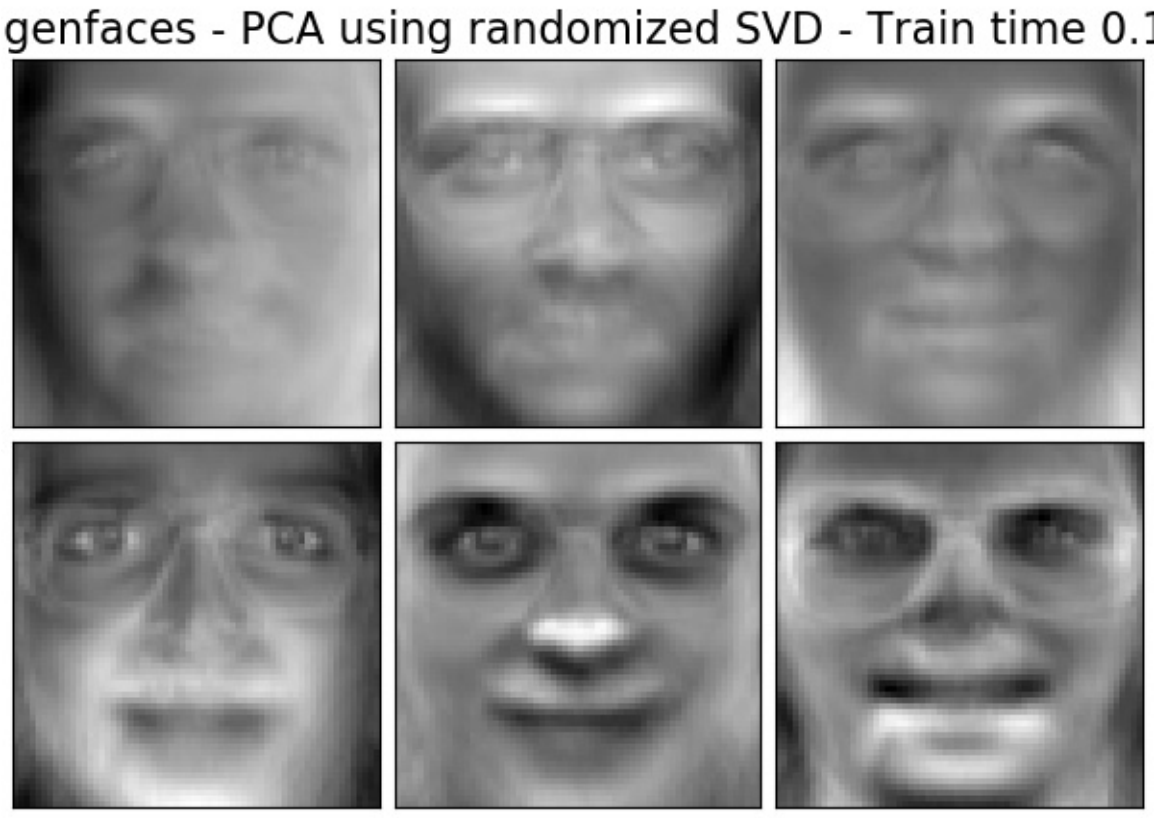
2.5.5. 独立成分分析（ICA）

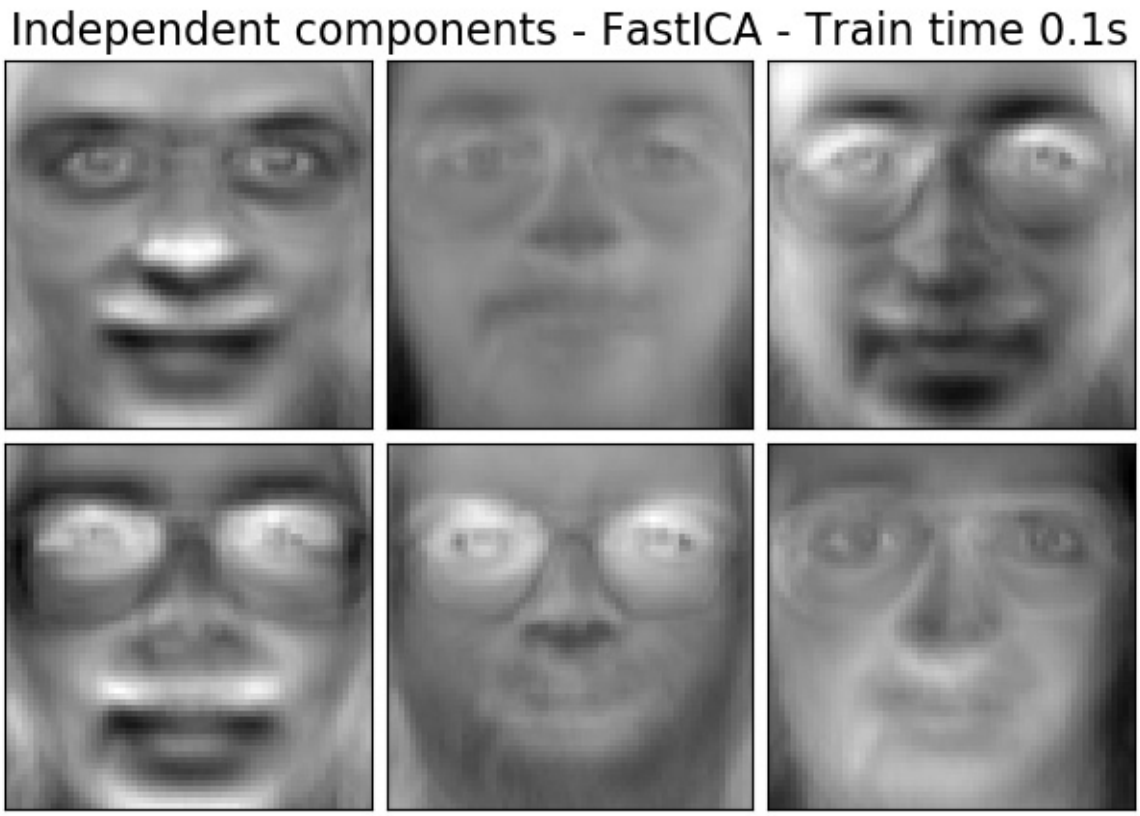
独立分量分析将多变量信号分解为独立性最强的加性子组件。 它通过 `Fast ICA` 算法在 scikit-learn 中实现。 ICA 通常不用于降低维度，而是用于分离叠加信号。 由于 ICA 模型不包括噪声项，因此要使模型正确，必须使用白化(whitening)。这可以在内部使用 whiten 参数或手动使用 PCA 的一种变体。

ICA 通常用于分离混合信号（称为 盲源分离 的问题），如下例所示：



ICA 也可以用于寻找具有稀疏性的分量的非线性分解：





示例：

- Blind source separation using FastICA
- FastICA on 2D point clouds
- Faces dataset decompositions

2.5.6. 非负矩阵分解(NMF 或 NNMF)

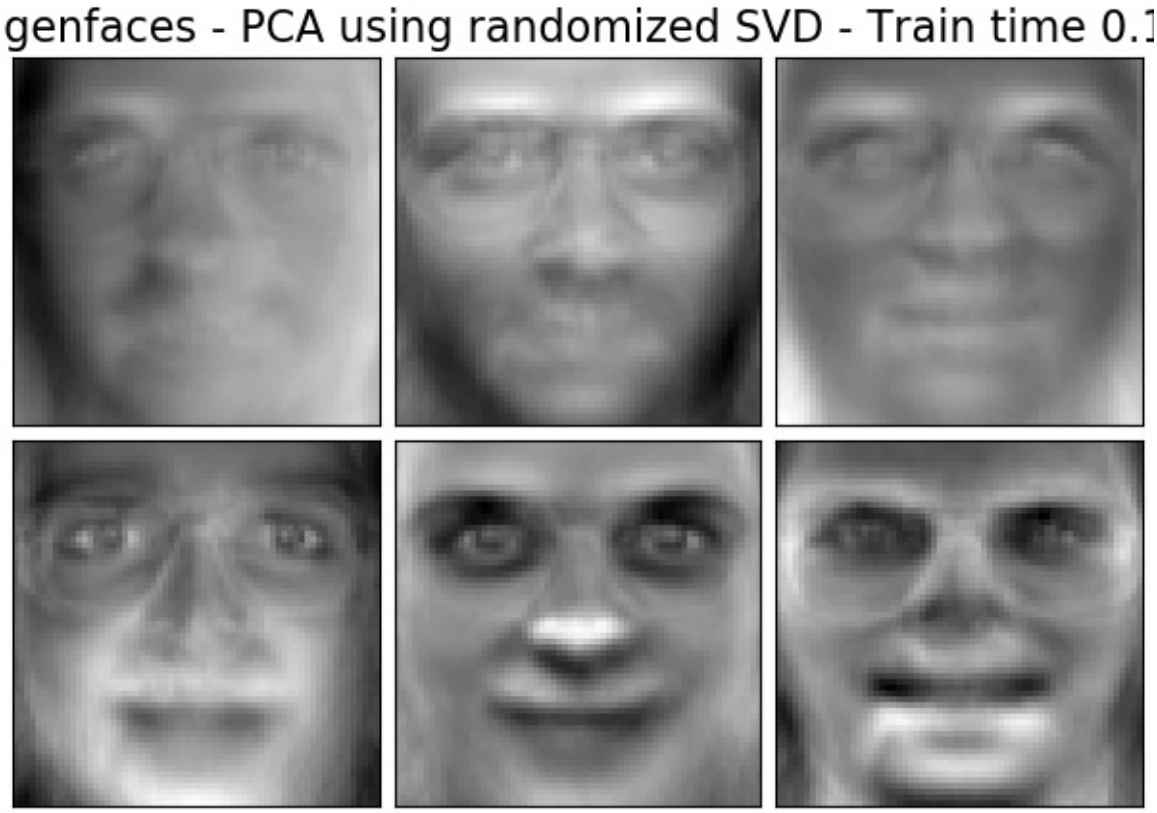
2.5.6.1. NMF 与 Frobenius 范数

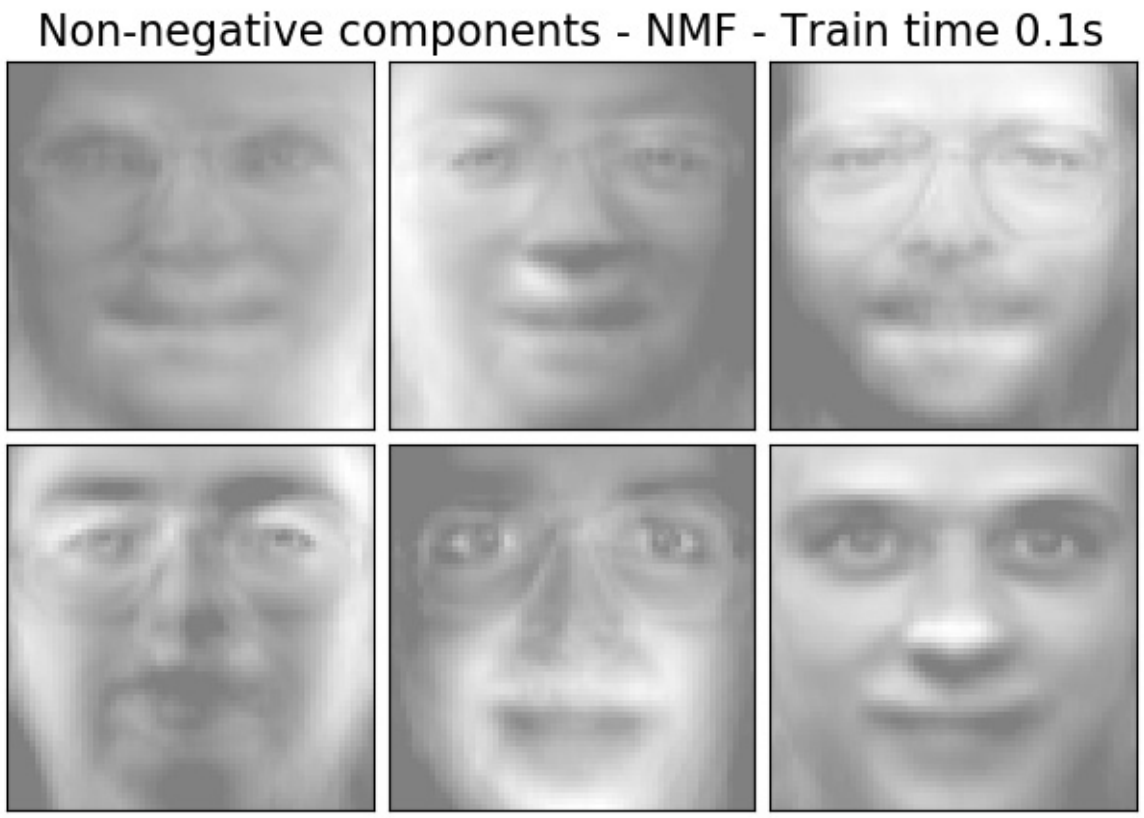
NMF [1] 是在数据和分量是非负情况下的另一种降维方法。 在数据矩阵不包含负值的情况下，可以插入 NMF 而不是 PCA 或其变体。 通过优化 X 与矩阵乘积 WH 之间的距离 d ，可以将样本 X 分解为两个非负矩阵 W 和 H。 最广泛使用的距离函数是 Frobenius 平方范数，它是欧几里德范数到矩阵的推广：

$$d_{\mathrm{Fro}}(X, Y) = \frac{1}{2} ||X - Y||_{\mathrm{Fro}}^2 = \frac{1}{2} \sum_{i,j} (X_{ij} - Y_{ij})^2$$

与 PCA 不同，通过叠加分量而不减去，以加法方式获得向量的表示。这种加性模型对于表示图像和文本是有效的。

[Hoyer, 2004] [2] 研究表明，当处于一定约束时， NMF 可以产生数据集基于某子部分的表示，从而获得可解释的模型。以下示例展示了与 PCA 特征面相比， NMF 从 Olivetti 面部数据集中的图像中发现的16个稀疏分量。





`init` 属性确定了应用的初始化方法，这对方法的性能有很大的影响。 `NMF` 实现了非负双奇异值分解方法。 NNDSVD [4] 基于两个 SVD 过程，一个估算数据矩阵， 另一个使用单位秩矩阵的代数性质估算的部分SVD因子的正部分。基本的 NNDSVD 算法更适合稀疏分解。其变体 NNDSVDa（全部零值替换为所有元素的平均值）和 NNDSVDar（零值替换为比数据平均值除以100小的随机扰动）在稠密情况时推荐使用。

请注意，乘法更新（ Multiplicative Update ‘mu’）求解器无法更新初始化中存在的零，因此当与引入大量零的基本 NNDSVD 算法联合使用时，会导致较差的结果；在这种情况下，应优先使用 NNDSVDa 或 NNDSVDar。

也可以通过设置 `init="random"` ，使用正确缩放的随机非负矩阵来初始化 `NMF` 。 整数种子或 `RandomState` 也可以传递给 `random_state` 以控制重现性。

在 `NMF` 中，L1 和 L2 先验可以被添加到损失函数中以使模型正则化。 L2 先验使用 Frobenius 范数，而L1 先验使用 L1 范数。与 `ElasticNet` 一样， 我们通过 `l1_ratio` （ρ）参数和正则化强度参数 `alpha` （α）来控制 L1 和 L2 的组合。那么先验项是：

$$\alpha\rho||W||_1 + \alpha\rho||H||_1 + \frac{\alpha(1-\rho)}{2}||W||_{\text{Fro}}^2 + \frac{\alpha(1-\rho)}{2}||H||_{\text{Fro}}^2$$

正则化目标函数为：

$$d_{\text{Fro}}(X, WH) + \alpha\rho||W||_1 + \alpha\rho||H||_1 + \frac{\alpha(1-\rho)}{2}||W||_{\text{Fro}}^2 + \frac{\alpha(1-\rho)}{2}||H||_{\text{Fro}}^2$$

`NMF` 正则化 W 和 H . 公共函数 `non_negative_factorization` 允许通过 `regularization` 属性进行更精细的控制，将 仅W ，仅H 或两者正则化。

2.5.6.2. 具有 beta-divergence 的 NMF

如前所述，最广泛使用的距离函数是 Frobenius 平方范数，这是欧几里得范数到矩阵的推广：

$$d_{\text{Fro}}(X, Y) = \frac{1}{2}||X - Y||_{Fro}^2 = \frac{1}{2}\sum_{i,j}(X_{ij} - Y_{ij})^2$$

其他距离函数可用于 NMF，例如（广义） Kullback-Leibler(KL) 散度，也称为 I-divergence：

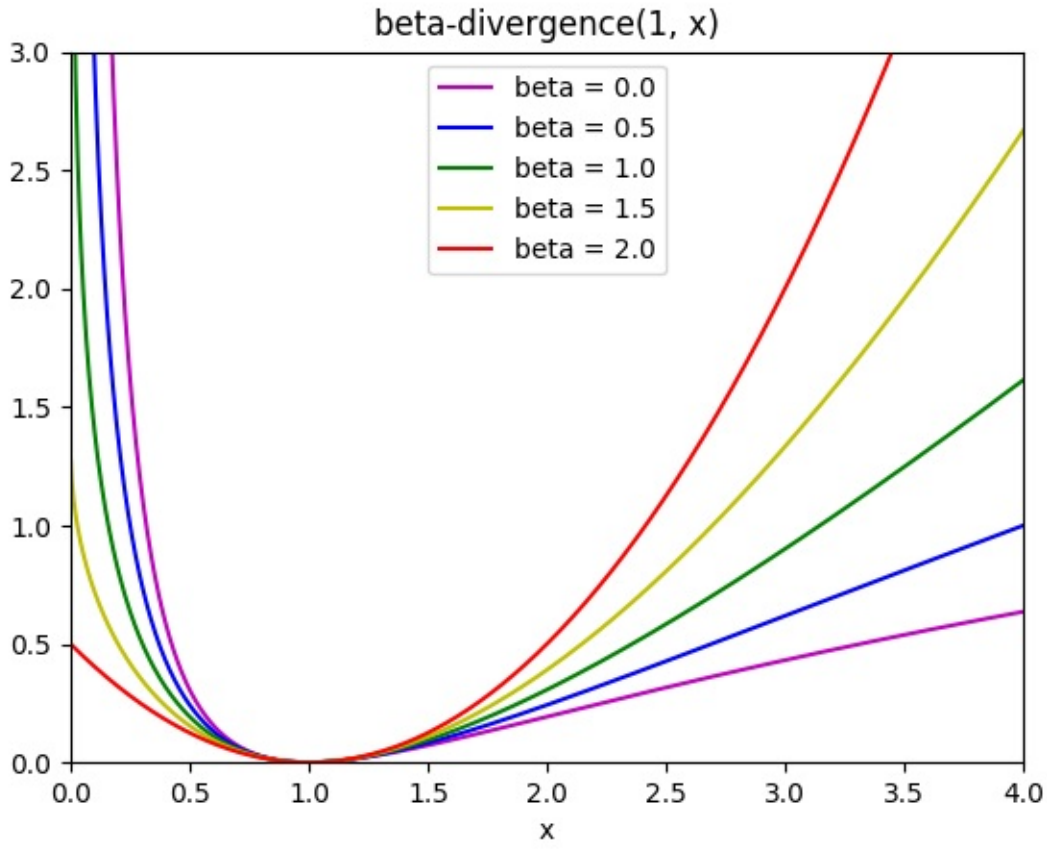
$$d_{KL}(X, Y) = \sum_{i,j}(X_{ij}\log(\frac{X_{ij}}{Y_{ij}}) - X_{ij} + Y_{ij})$$

或者， Itakura-Saito(IS) divergence：

$$d_{IS}(X, Y) = \sum_{i,j}(\frac{X_{ij}}{Y_{ij}} - \log(\frac{X_{ij}}{Y_{ij}}) - 1)$$

这三个距离函数是 beta-divergence 函数族的特殊情况，其参数分别为 $\beta = 2, 1, 0$ [6] 。 beta-divergence 定义如下：

$$d_\beta(X, Y) = \sum_{i,j}\frac{1}{\beta(\beta-1)}(X_{ij}^\beta + (\beta-1)Y_{ij}^\beta - \beta X_{ij}Y_{ij}^{\beta-1})$$



请注意，在 $\beta \in (0; 1)$ 上定义无效，仅仅分别连续扩展到 d_{KL} 和 d_{IS} 。

`NMF` 使用 坐标下降 （ Coordinate Descent ‘cd’） [5] 和乘法更新(Multiplicative Update ‘mu’) [6] 来实现两个求解器。 ‘mu’ 求解器可以被用于优化每个 beta-divergence，包括 Frobenius 范数 ($\beta = 2$) ， （广义） Kullback-Leibler divergence ($\beta = 1$) 和Itakura-Saito divergence（beta = 0）)。请注意，对于 $\beta \in (1; 2)$ ，’mu’ 求解器明显快于 β 的其他值。还要注意，使用负数（或0，即 ‘itakura-saito’ ） β ，输入矩阵不能包含零值。

‘cd’ 求解器只能优化 Frobenius 范数。由于 NMF 的潜在非凸性，即使优化相同的距离函数，不同的求解器也可能会收敛到不同的最小值。

NMF最适用于 *fittransform* 方法，该方法返回矩阵*W*.矩阵 *H* 被存储到拟合模型中的 *components* 属性；方法 `transform` 将基于这些存储的分量,分解新的矩阵 X_new：

```
>>> import numpy as np
>>> X = np.array([[1, 1], [2, 1], [3, 1.2], [4, 1], [5, 0.8], [6, 1]])
>>> from sklearn.decomposition import NMF
>>> model = NMF(n_components=2, init='random', random_state=0)
>>> W = model.fit_transform(X)
>>> H = model.components_
>>> X_new = np.array([[1, 0], [1, 6.1], [1, 0], [1, 4], [3.2, 1], [0, 4]])
>>> W_new = model.transform(X_new)
```

示例：

- Faces dataset decompositions
- Topic extraction with Non-negative Matrix Factorization and Latent Dirichlet Allocation
- Beta-divergence loss functions

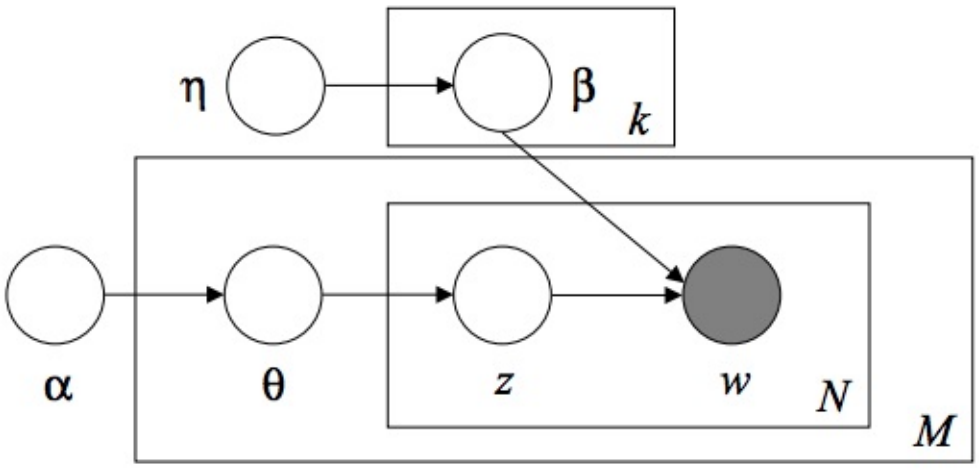
参考资料：

- [1] “Learning the parts of objects by non-negative matrix factorization” D. Lee, S. Seung, 1999
- [2] “Non-negative Matrix Factorization with Sparseness Constraints” P. Hoyer, 2004
- [4] “SVD based initialization: A head start for nonnegative matrix factorization” C. Boutsidis, E. Gallopoulos, 2008
- [5] “Fast local algorithms for large scale nonnegative matrix and tensor factorizations.” A. Cichocki, P. Anh-Huy, 2009
- [6] “Algorithms for nonnegative matrix factorization with the beta-divergence” C. Fevotte, J. Idier, 2011

2.5.7. 隐 Dirichlet 分配（LDA）

隐 Dirichlet 分配是离散数据集（如文本语料库）的集合的生成概率模型。 它也是一个主题模型，用于从文档集合中发现抽象主题。

LDA 的图形模型是一个三级生成模型：



关于上述图形模型中出现的符号的说明，可以在Hoffman et al.(2013)中找到：

- 语料库是D个文档的集合。
- 文档是由N个单词组成的序列。
- 语料库中有K个主题。
- 这些框表示重复采样。

在图形模型中，每个节点都是一个随机变量，在生成过程中起着一定的作用。阴影节点表示观察到的变量，未阴影节点表示隐藏的(潜在的)变量。在这种情况下，语料库中的单词是我们观察到的唯一数据。潜在变量决定了语料库中主题的随机混合和文档中单词的分布。LDA的目标是利用观察到的词语来推断隐含的主题结构。

当建模文本语料库时，该模型假设具有 D 文档和 K 主题的语料库的以下生成过程，其中K 对应 API 中的 `n_components` ：

- 对于每个主题 k∈Kk\in Kk∈K，绘制 βk Dirichlet(η)\beta_k \thicksim Dirichlet(\eta) βk Dirichlet(η)。提供一个基于单词的分布，比如，一个单词在 topic kkk 中出现的概率。 η\etaη 对应 topic_word_prior。
- 对于每个文档 d∈Dd\in Dd∈D，绘制主题概率 θk Dirichlet(α)\theta_k \thicksim Dirichlet(\alpha) θk Dirichlet(α)。α\alphaα 对应 doc_topic_prior。
- 对于文档 d 中的每个单词 i：
  - 绘制主题索引 $z_{di} \sim \text{Multinomial}(\theta_d)$
  - 绘制观察词 $w_{ij} \sim \text{Multinomial}(beta_{z_{di}.})$ 对于参数估计，后验分布为：

$$p(z, \theta, \beta|w, \alpha, \eta) = \frac{p(z, \theta, \beta|\alpha, \eta)}{p(w|\alpha, \eta)}$$

由于后验分布难以处理，变体贝叶斯方法使用更简单的分布 $q(z, \theta, \beta|\lambda, \phi, \gamma)$ 去近似估算， 并且优化了这些变体参数 λ, ϕ, γ 以最大化Evidence Lower Bound (ELBO):

$$\log\ P(w|\alpha, \eta) \geq L(w, \phi, \gamma, \lambda) \stackrel{\triangle}{=} E_q[\log\ p(w, z, \theta, \beta|\alpha, \eta)] - E_q[\log\ q(z, \theta, \beta)]$$

最大化 ELBO 相当于最小化 $q(z, \theta, \beta)$ 和后验 $p(z, \theta, \beta|w, \alpha, \eta)$ 之间的 Kullback-Leibler(KL) 散度。

`LatentDirichletAllocation` 实现在线变体贝叶斯算法，支持在线和批量更新方法。 批处理方法在每次完全传递数据后更新变体变量，在线方法从小批量数据点中更新变体变量。

> 注意虽然在线方法保证收敛到局部最优点，最优点的质量和收敛速度可能取决于与小批量大小和学习率相关的属性。

当 `LatentDirichletAllocation` 应用于 “document-term” 矩阵时，矩阵将被分解为 “topic-term” 矩阵和 “document-topic” 矩阵。 虽然 “topic-term” 矩阵在模型中被存储为 `components_` ，但是可以通过 `transform` 方法计算 “document-topic” 矩阵。

`LatentDirichletAllocation` 还实现了 `partial_fit` 方法。这可用于当数据被顺序提取时。

> 示例：
>
> - Topic extraction with Non-negative Matrix Factorization and Latent Dirichlet Allocation
>
> 参考资料：
>
> - “Latent Dirichlet Allocation” D. Blei, A. Ng, M. Jordan, 2003
> - “Online Learning for Latent Dirichlet Allocation” M. Hoffman, D. Blei, F. Bach, 2010
> - “Stochastic Variational Inference” M. Hoffman, D. Blei, C. Wang, J. Paisley, 2013

参考 Neighborhood Components Analysis 的降维方式.

我们一直在努力

apachecn/sklearn-doc-zh


加入QQ群

2.6. 协方差估计

校验者：@李昊伟@小瑶@Loopy@barrycg翻译者：@柠檬

许多统计问题需要估计一个总体的协方差矩阵，这可以看作是对数据集散点图形状的估计。大多数情况下，必须对某个样本进行这样的估计，当它的属性(如尺寸，结构，均匀性)对估计质量有重大影响时。*sklearn.covariance* 包的目的是提供一个能在各种设置下准确估计总体协方差矩阵的工具。

我们假设观察是独立的，相同分布的 (i.i.d.)。

2.6.1. 经验协方差

总所周知，数据集的协方差矩阵可以被经典 *maximum likelihood estimator*(最大似然估计) （或 “经验协方差”） 很好地近似，条件是与特征数量（描述观测值的变量）相比，观测数量足够大。 更准确地说，样本的最大似然估计是相应的总体协方差矩阵的无偏估计。

样本的经验协方差矩阵可以使用该包的函数 `empirical_covariance` 计算 ， 或者使用 `EmpiricalCovariance.fit` 方法将对象 `EmpiricalCovariance` 与数据样本拟合 。 要注意，根据数据是否聚集，结果会有所不同，所以可能需要准确地使用参数 `assume_centered` 。更准确地说，如果要使用 `assume_centered=False` ， 测试集应该具有与训练集相同的均值向量。 如果不是这样，两者都应该被用户聚集， 然后再使用 `assume_centered=True` 。

> 示例：
>
> - See Shrinkage covariance estimation: LedoitWolf vs OAS and max-likelihood for an example on how to fit an `EmpiricalCovariance` object to data.

2.6.2. 收缩协方差

2.6.2.1. 基本收缩

尽管是协方差矩阵的无偏估计， 最大似然估计不是协方差矩阵的特征值的一个很好的估计， 所以从反演(译注：矩阵的求逆过程)得到的精度矩阵是不准确的。 有时，甚至出现因矩阵元素地特性，经验协方差矩阵不能求逆。 为了避免这样的反演问题，引入了经验协方差矩阵的一种变换方式： `shrinkage` 。

在 scikit-learn 中，该变换（使用用户定义的收缩系数） 可以直接应用于使用 `shrunk_covariance` 方法预先计算的协方差。 此外，协方差的收缩估计可以通过 `ShrunkCovariance` 对象的 `ShrunkCovariance.fit` 方法拟合到数据中。再次，根据数据是否聚集，结果会不同，所以可能要准确使用参数 `assume_centered` 。

在数学上，这种收缩在于减少经验协方差矩阵的最小和最大特征值之间的比率 可以通过简单地根据给定的偏移量移动每个特征值来完成，这相当于找到协方差矩阵的l2惩罚的最大似然估计器（l2-penalized Maximum Likelihood Estimator）。在实践中，收缩归结为简单的凸变换：$\Sigma_{\text{shrunk}} = (1-\alpha)\hat{\Sigma} + \alpha\frac{\text{Tr}\hat{\Sigma}}{p}\text{Id}$.

选择收缩量， α 相当于设置了偏差/方差权衡，下面将就此进行讨论。

示例：

- See Shrinkage covariance estimation: LedoitWolf vs OAS and max-likelihood for an example on how to fit a `ShrunkCovariance` object to data.

2.6.2.2. Ledoit-Wolf 收缩

在他们的 2004 年的论文 [1] 中， O.Ledoit 和 M.Wolf 提出了一个公式， 用来计算最优的收缩系数 α ，它使得估计协方差和实际协方差矩阵之间的均方误差(Mean Squared Error)进行最小化。

可以使用sklearn.covariance包中 `ledoit_wolf` 函数来计算样本的协方差的Ledoit-Wolf 估计， 或者拟合一个 `LedoitWolf` 对象对相同的样本来获取该估计。

注意:协方差矩阵为各向同性(isotropic)的情形

值得注意的是，当样本数量远远大于特征数量时，人们会认为不需要收缩。这背后的直觉是，如果总体协方差是满秩的， 当样本数量增加时，样本协方差也会变为正定。因此，没有收缩的必要，该方法应该自动做到这一点。

然而， 上述情况并不适用于总体协方差恰好是恒等矩阵(identity matrix)的倍数时， 即Ledoit-Wolf过程。在这种情况下，Ledoit-Wolf收缩估计值随着样本数的增加而接近1。这表明，Ledoit-Wolf意义下协方差矩阵的最优估计是恒等式的倍数(multiple of the identity)。 由于总体协方差已经是恒等矩阵的倍数，Ledoit-Wolf解确实是一个合理的估计。

示例：

- See Shrinkage covariance estimation: LedoitWolf vs OAS and max-likelihood 关于如何将 `LedoitWolf` 对象与数据拟合， 并将 Ledoit-Wolf 估计器的性能进行可视化的示例。

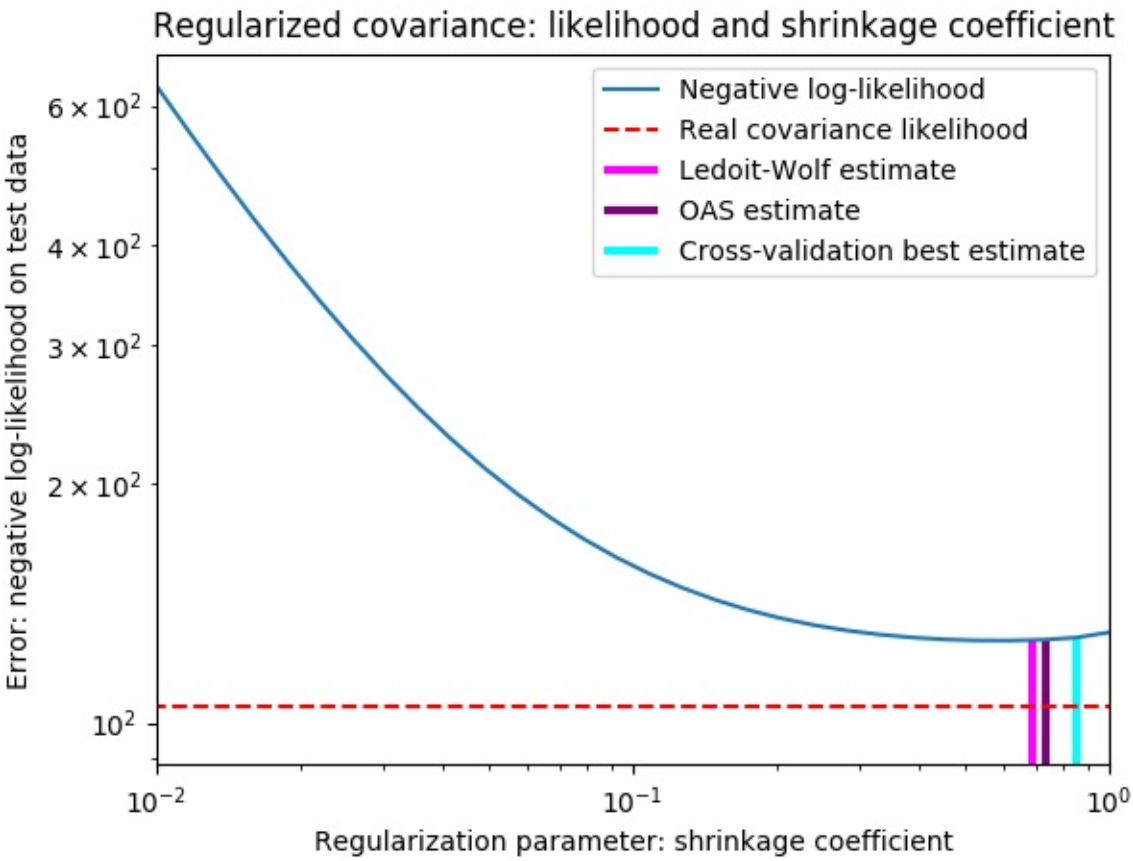
参考资料：

- [1] O. Ledoit and M. Wolf, “A Well-Conditioned Estimator for Large-Dimensional Covariance Matrices”, Journal of Multivariate Analysis, Volume 88, Issue 2, February 2004, pages 365-411.

2.6.2.3. Oracle 近似收缩

在数据为高斯分布的假设下，Chen et al. 等 [2] 推导出了一个公式，旨在 产生比 Ledoit-Wolf 公式具有更小均方误差的收缩系数。 所得到的估计器被称为协方差的 Oracle 收缩近似估计器。

可以使用sklearn.covariance 包中函数 `oas` 计算样本协方差的OAS估计，或者可以通过将 `OAS` 对象拟合到相同的样本来获得该估计。



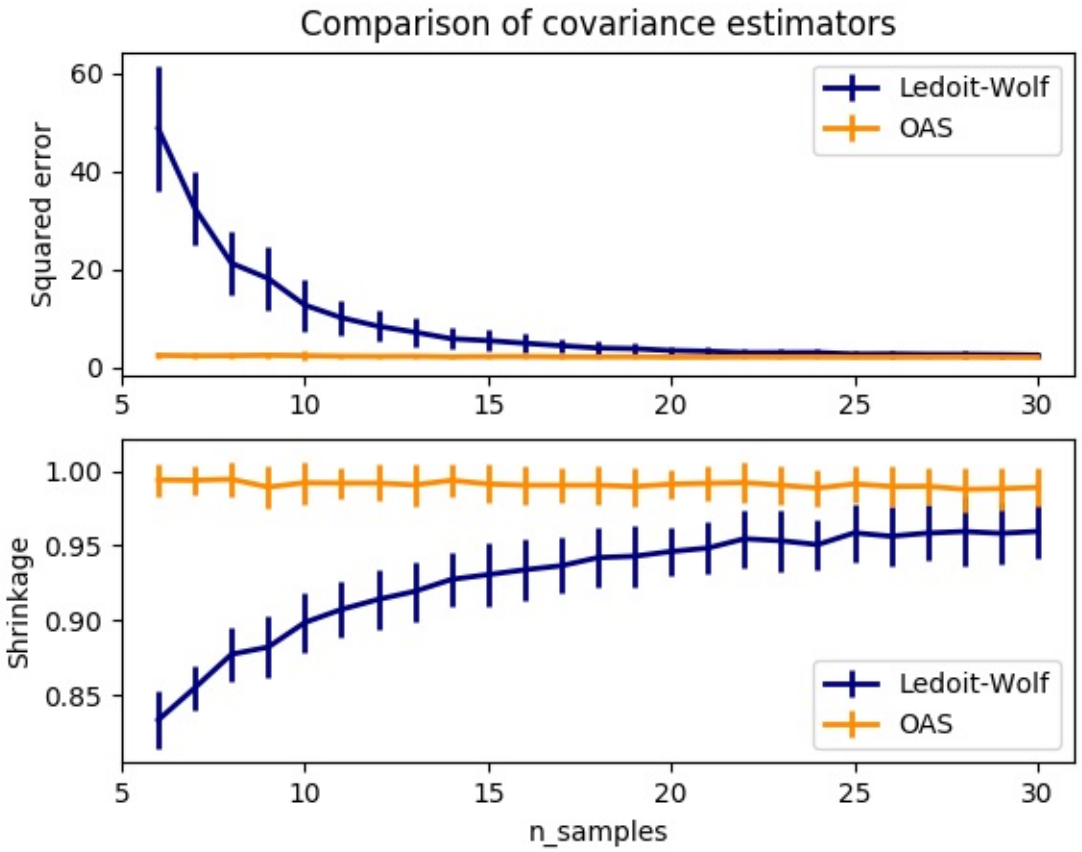
设定收缩时的偏差方差权衡：比较 Ledoit-Wolf 和 OAS 估计的选择

参考资料：

- [2]Chen et al., “Shrinkage Algorithms for MMSE Covariance Estimation”, IEEE Trans. on Sign. Proc., Volume 58, Issue 10, October 2010.

示例：

- See Shrinkage covariance estimation: LedoitWolf vs OAS and max-likelihood for an example on how to fit an `OAS` object to data.
- See Ledoit-Wolf vs OAS estimation to visualize the Mean Squared Error difference between a `LedoitWolf` and an `OAS` estimator of the covariance.

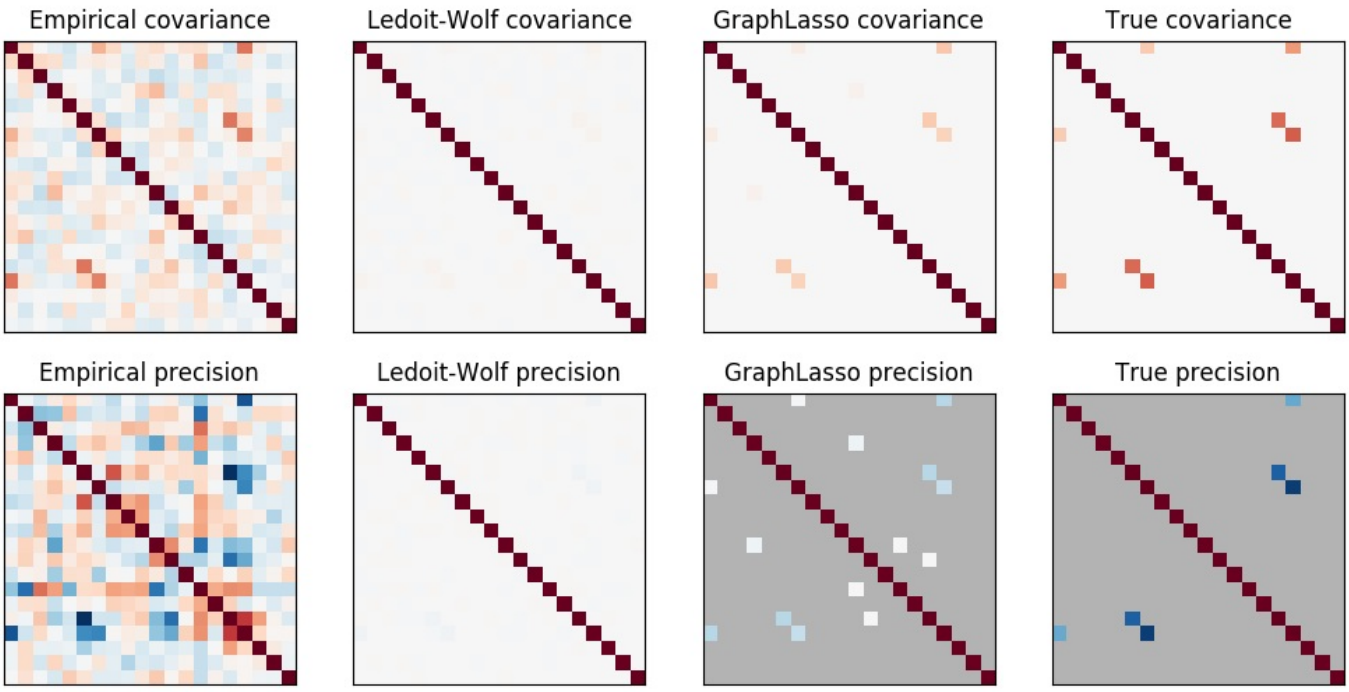


2.6.3. 稀疏逆协方差

协方差矩阵的逆矩阵，通常称为精度矩阵（precision matrix），它与部分相关矩阵（partial correlation matrix）成正比。它给出部分独立性关系。换句话说，如果两个特征与其他特征有条件地独立， 则精度矩阵中的对应系数将为零。这就是为什么估计一个稀疏精度矩阵是有道理的：通过从数据中学习独立关系， 可以得到更好的协方差矩阵估计。这被称为 协方差选择*(covariance selection)*。

在小样本的情况，即 `n_samples` 和 `n_features` 相等或更小， 稀疏的逆协方差估计往往比收缩的协方差估计更好。然而，在相反的情况下，或者对于非常相关的数据，它们可能在数值上不稳定。 此外，与收缩估算不同，稀疏估计器能够恢复非对角线结构（off-diagonal structure）。

`GraphLasso` 估计器使用 L1 惩罚确立精度矩阵的稀疏性： `alpha` 参数越高，精度矩阵的稀疏性越大。 相应的 `GraphLassoCV` 对象使用交叉验证来自动设置 `alpha` 参数。



协方差矩阵和 精度矩阵基于最大似然度估计，收缩估计和稀疏估计的比较_

> 注意：结构恢复
>
> 从数据中的相关性恢复图形结构是一个具有挑战性的事情。如果您对这种恢复感兴趣，请记住：
>
> - 相关矩阵的恢复比协方差矩阵更容易：在运行 `GraphLasso` 前先标准化观察值
> - 如果底层图具有比平均节点更多的连接节点，则算法将错过其中一些连接。
> - 如果您的观察次数与底层图形中的边数相比不大，则不会恢复。
> - 即使您具有良好的恢复条件，通过交叉验证（例如使用GraphLassoCV对象）选择的 Alpha 参数将导致选择太多边。然而，相关的边会比不相关的边有更大的权重。

数学公式如下：

$$\hat{K} = \operatorname{argmin}_K \left(\operatorname{tr} SK - \log\det K + \alpha \|K\|_1 \right)$$

其中：K 是要估计的精度矩阵（precision matrix），S 是样本的协方差矩阵。$\|K\|_1$ 是非对角系数 K（off-diagonal coefficients）的绝对值之和。 用于解决这个问题的算法是来自 Friedman 2008 Biostatistics 论文的 GLasso 算法。 它与 R 语言 `glasso` 包中的算法相同。

> 示例：
>
> - Sparse inverse covariance estimation：合成数据的示例，显示结构的一些恢复，并与其他协方差估计器进行比较。
> - Visualizing the stock market structure：真实股票市场数据的示例，查找哪些信号相关度最强。
>
> 参考资料：
>
> - Friedman et al, “Sparse inverse covariance estimation with the graphical lasso”, Biostatistics 9, pp 432, 2008

2.6.4. 鲁棒协方差估计

实际数据集通常是会有测量或记录错误。常规但不常见的观察也可能出于各种原因。 每个不常见的观察称为异常值(outliers)。上面提出的经验协方差估计器和收缩协方差估计器对数据中异常值非常敏感。 因此，应该使用鲁棒协方差估计（robust covariance estimators）来估算其真实数据集的协方差。 或者，可以使用鲁棒协方差估计器来执行异常值检测， 并根据数据的进一步处理：丢弃/降低某些观察值。

`sklearn.covariance` 包实现了 robust estimator of covariance， 即 Minimum Covariance Determinant [3]。

2.6.4.1. 最小协方差决定

最小协方差决定（Minimum Covariance Determinant）估计器是 由 P.J. Rousseeuw 在 [3] 中引入的数据集协方差的鲁棒估计 (robust estimator)。 这个想法是找出一个给定比例（h）的 “好” 观察值，它们不是离群值， 且可以计算其经验协方差矩阵。 然后将该经验协方差矩阵重新缩放以补偿所执行的观察选择（”consistency step(一致性步骤)”）。计算完最小协方差决定估计器后，可以根据其马氏距离（Mahalanobis distance）给出观测值的权重， 得到数据集的协方差矩阵的重新加权估计（”reweighting step(重新加权步骤)”）。

Rousseeuw 和 Van Driessen [4] 开发了 FastMCD 算法，以计算最小协方差决定因子（Minimum Covariance Determinant）。在 scikit-learn 中，在将 MCD 对象拟合到数据时， 使用该算法。FastMCD 算法同时计算数据集位置的鲁棒估计。

原始估计(Raw estimates)可通过 `MinCovDet` 对象的 `raw*location*` 和 `raw*covariance*` 属性获得。

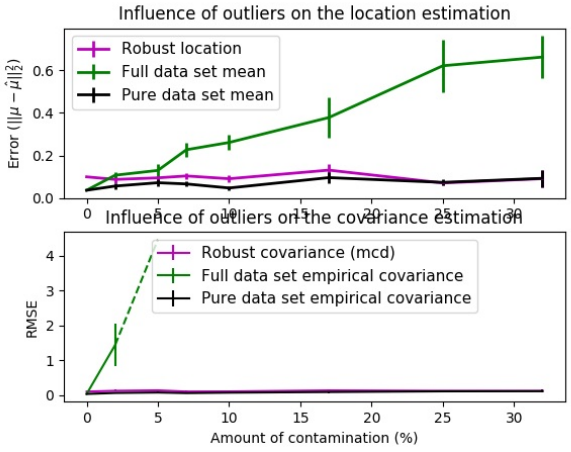
参考资料：

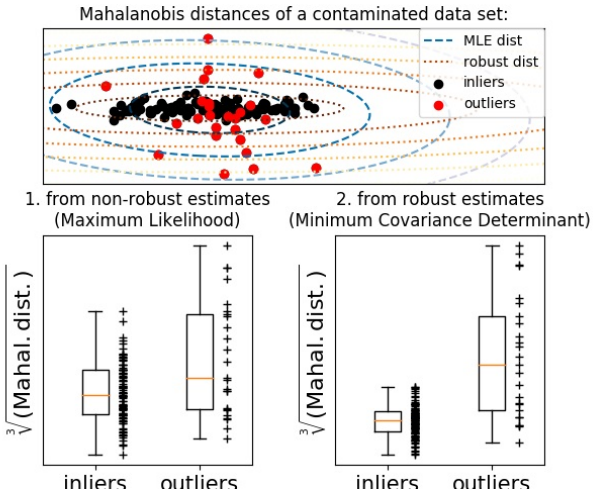
- [3] P. J. Rousseeuw. Least median of squares regression. J. Am Stat Ass, 79:871, 1984.
- [4] A Fast Algorithm for the Minimum Covariance Determinant Estimator, 1999, American Statistical Association and the American Society for Quality, TECHNOMETRICS.

示例：

- See Robust vs Empirical covariance estimate 关于如何将对象 `MinCovDet` 与数据拟合的示例， 尽管存在异常值，但估计结果仍然比较准确。
- See Robust covariance estimation and Mahalanobis distances relevance 依据马氏距离（Mahalanobis distance），对协方差估计器 `EmpiricalCovariance` 和 `MinCovDet` 之间的差异进行可视化。（所以我们得到了精度矩阵的更好估计）

异常值对位置和协方差估计的影响	使用Mahalanobis距离分离内围点和离群点





我们一直在努力

apachecn/sklearn-doc-zh

加入QQ群

2.7. 新奇点和离群点检测

校验者：@RyanZhiNie@羊三@Loopy@barrycg翻译者：@羊三

许多应用程序需要能够对新观测值(observation 译注:观测到的样本的值)进行判断，判断其是否与现有观测值服从同一分布(即新观测值为内围点(inlier))，相反则被认为不服从同一分布(即新观测值为离群点(outlier))。通常，这种能力被用于清理真实数据集， 但它有两种重要区分：

- 离群点检测： 训练数据包含离群点,即远离其它内围点。离群点检测估计器会尝试拟合出训练数据中内围点聚集的区域， 会忽略有偏离的观测值。
- 新奇点检测： 训练数据未被离群点污染，我们对新观测值是否为离群点感兴趣。在这个语境下，离群点被认为是新奇点。

离群点检测 和 新奇点检测 都被用于异常检测， 所谓异常检测就是检测反常的观测值或不平常的观测值。离群点检测 也被称之为 无监督异常检测； 而 新奇点检测 被称之为 半监督异常检测。 在离群点检测的语境下， 离群点/异常点 不能够形成一个稠密的聚类簇，因为可用的估计器都假定了 离群点/异常点 位于低密度区域。相反的，在新奇点检测的语境下，新奇点/异常点 是可以形成 稠密聚类簇的，只要它们在训练数据的一个低密度区域，这被认为是正常的。

scikit-learn项目提供了一套可用于新奇点或离群点检测的机器学习工具。该策略是以无监督的方式学习数据中的对象来实现的：

```
1. estimator.fit(X_train)
```

然后可以使用 *predict* 方法将新观测值归为内围点或离群点：

```
1. estimator.predict(X_test)
```

内围点被标记为1，而离群点被标记为-1。 预测方法在估计器计算出的原始评分函数上使用一个阈值。这个评分函数可以通过方法 `score_samples` 进行访问，而且这个阈值可以由参数 `contamination` 控制。

`decision_function` 方法也是使用评分函数定义的，这样的话，得分为负值的就是 离群点， 得分为非负的就是 内围点：

```
1. estimator.decision_function(X_test)
```

请注意 neighbors.LocalOutlierFactor 类默认不支持 `predict` , `decisionfunction` 和 *score_samples* 方法，而只支持 *fit_predict* 方法， 因为这个估计器从一开始就是被用到 离群点检测中去的。训练样本的异常性得分可以通过访问 *negative_outlier_factor* 属性来获取。

如果你真的想用 neighbors.LocalOutlierFactor 类进行新奇点检测， 即：对新的未见过的样本 预测其标签或计算其异常性得分， 你可以在实例化这个估计器的时候将其 `novelty` 参数设为 True ，这一步必须要在拟合之前做。这样的话， `fit_predict` 方法就不可用了。

> 警告 ：使用局部离群因子(Local Outlier Factor,LOF)进行新奇点检测

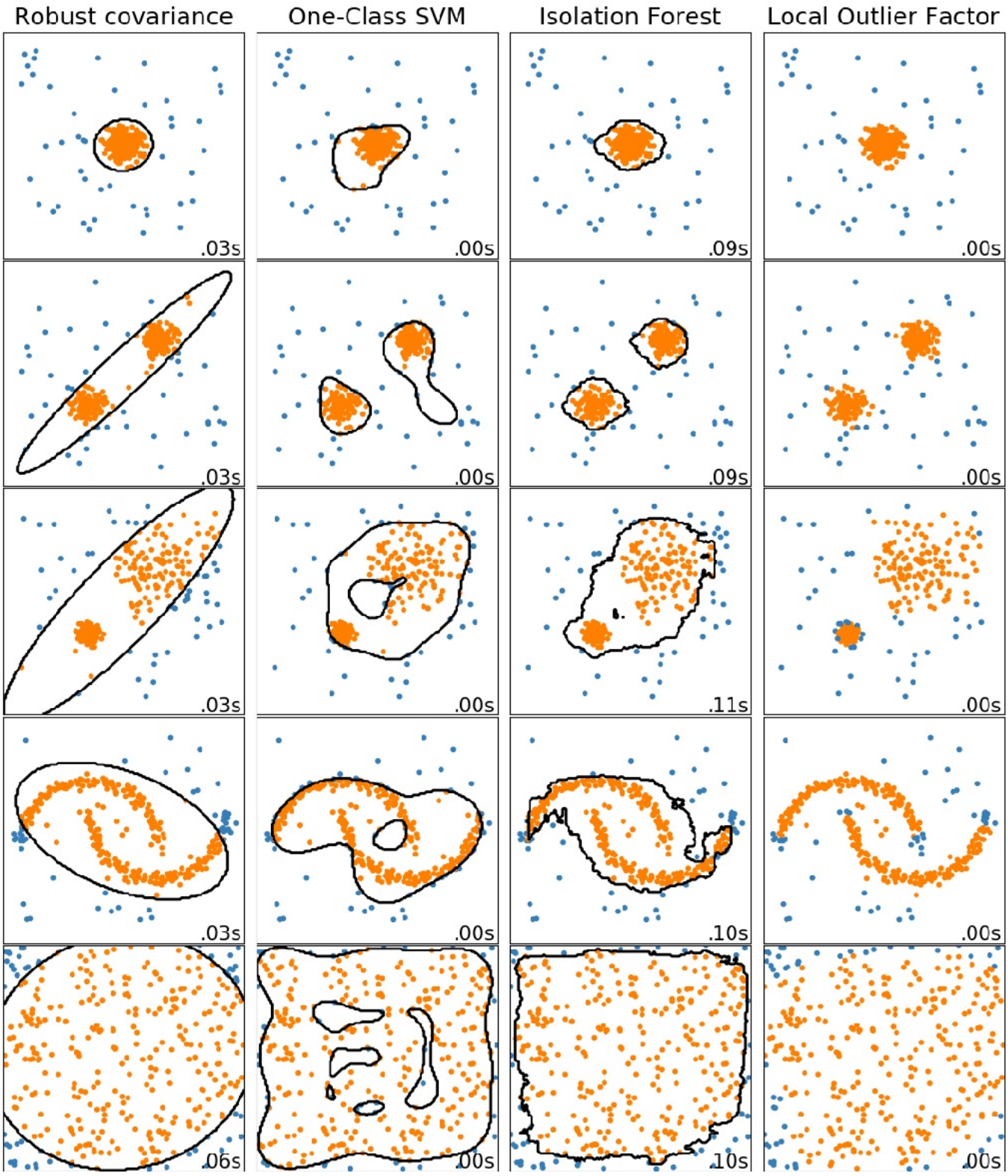
> 当 `novelty` 参数被设为 `True` 时，在新的未见过的数据上，你只能使用 `predict`，`decisionfunction` 和 `score_samples`，而不能把这几个函数用在训练数据上， 否则会导致错误的结果。训练样本的异常性得分总是可以通过访问 `negative_outlier_factor` 属性来获取。

`neighbors.LocalOutlierFactor` 类在离群点检测和新奇点检测中的行为被总结在下面的表中。

方法	离群点检测	新奇点检测
`fitpredict`	可用	不可用
`predict`	不可用	只能用于新数据
`decision_function`	不可用	只能用于新数据
`score_samples`	用 `negative_outlier_factor`	只能用于新数据

2.7.1. 离群点检测方法一览

下面这个示例针对scikit-learn 中的所有离群点检测算法进行了对比。 局部离群因子(LOF, Local Outlier Factor) 没有在下图画出黑色的决策边界，因为在离群点检测中使用LOF时 它没有 predict 方法可以用在新数据上（见上面表格）。



`ensemble.IsolationForest` 和 `neighbors.LocalOutlierFactor` 在这里所用的数据集上表现得相当好。`svm.OneClassSVM` 类对离群点本来就很敏感，因此在离群点检测中表现的不是很好(译注：这里的敏感 应该指的是会把离群点划到决策边界内)。最后，`covariance.EllipticEnvelope` 类 假定了数据是服从高斯分布的且要学习一个椭圆(ellipse)。关于这个对比试验中各种estimators的更多详细信息,请参考 `Comparing anomaly detection algorithms for outlier detection on toy datasets`和后续小节。

> 示例：
>
> - 查看 `Comparing anomaly detection algorithms for outlier detection on toy datasets`以对比 `svm.OneClassSVM`, `ensemble.IsolationForest`, `neighbors.LocalOutlierFactor` 和 `covariance.EllipticEnvelope` 的表现.

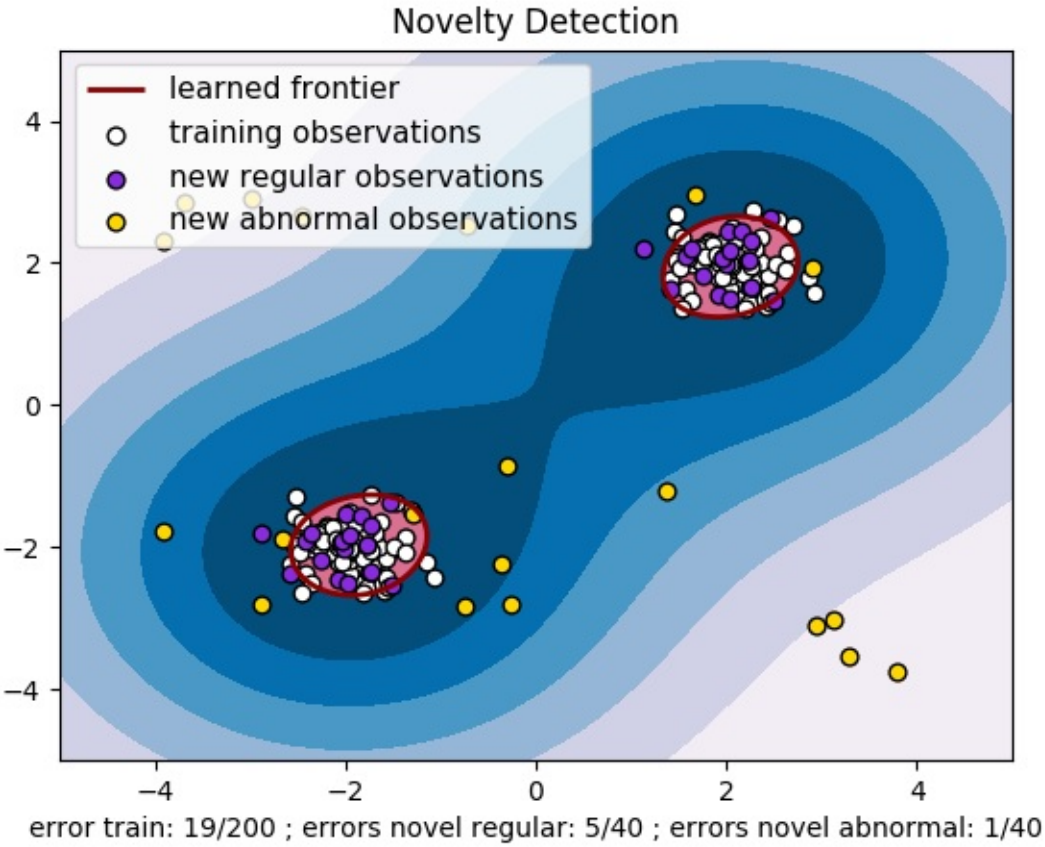
2.7.2. Novelty Detection（新奇点检测）

考虑一个以 p 个特征描述、包含 n 个有着相同分布的观测值的数据集。 现在考虑我们再往该数据集中添加一个观测值。 如果新观测与原有观测有很大差异，我们就可以怀疑它是否是内围值(regular 译注：应是常规值,这里用已使用过的内围值)吗？ （即是否来自同一分布？）或者相反，如果新观测与原有观测很相似，我们就无法将其与原有观测区分开吗？ 这就是新奇点检测工具和方法所解决的问题。

一般来说，它将要学习出一个粗略且紧密的边界，界定出初始观测分布的轮廓，绘制在嵌入的 p 维空间中。那么，如果后续的观测值都落在这个边界划分的子空间内，则它们被认为来自与初始观测值相同的总体。 否则，如果它们在边界之外，我们可以说就我们评估中给定的置信度而言，它们是异常的。

One-Class SVM（单类支持向量机）已经由 Schölkopf 等人采用以实现新奇检测，并在 支持向量机 模块的 `svm.OneClassSVM` 对象中实现。 需要选择 kernel 和 scalar 参数来定义边界。 通常选择 RBF kernel，即使没有确切的公式或算法来设置其带宽(bandwidth)参数。这是 scikit-learn 实现中的默认值。 ν 参数，也称为单类支持向量机的边沿，对应于在边界之外找到新的但常规的观测值的概率。

> 参考资料：
>
> - Estimating the support of a high-dimensional distribution Schölkopf, Bernhard, et al. Neural computation 13.7 (2001): 1443-1471.
>
> 示例：
>
> - 参见 One-class SVM with non-linear kernel (RBF)，通过 `svm.OneClassSVM` 对象学习一些数据来将边界可视化。
> - Species distribution modeling



2.7.3. Outlier Detection（离群点检测）

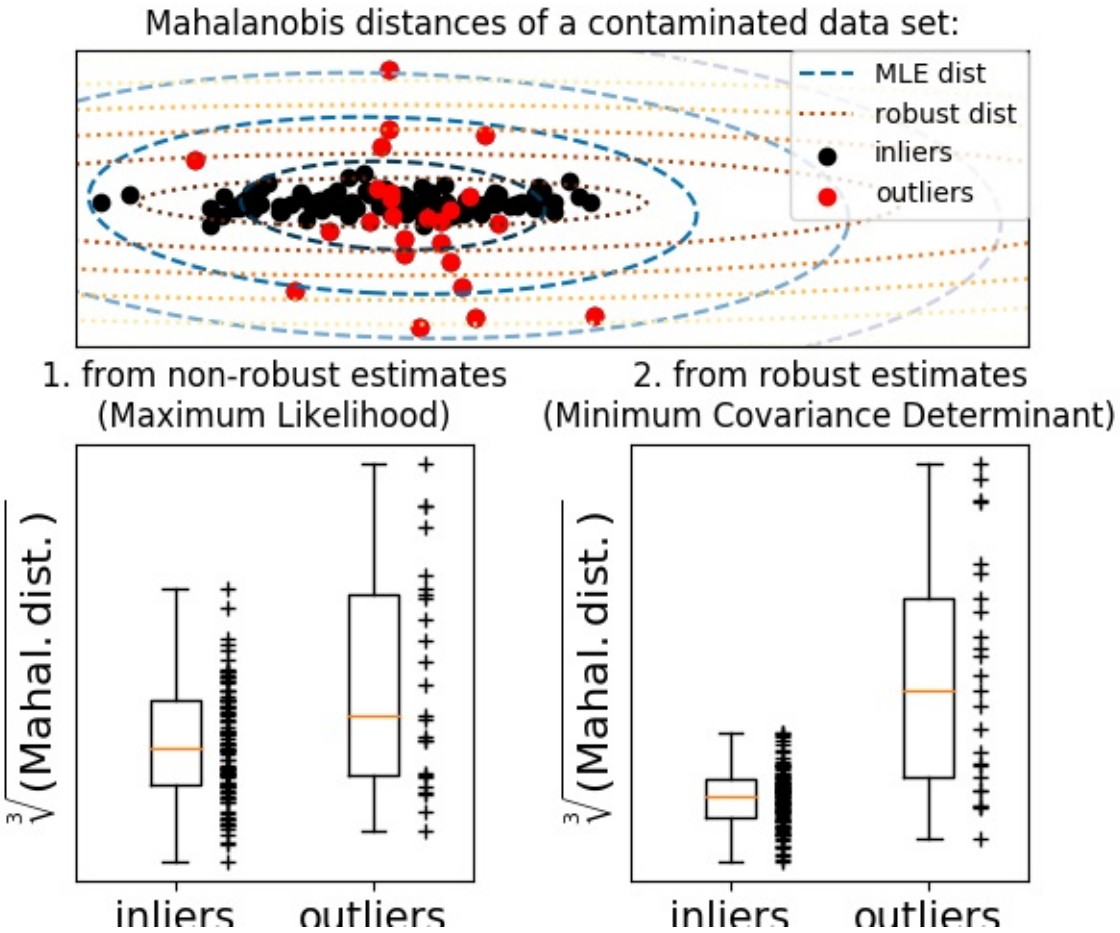
离群点检测类似于新奇点检测，其目的是将内围观测(regular observation)的中心与一些被称为 “离群点” 的污染数据进行分离。 然而，在离群点检测的情况下，我们没有干净且适用于训练任何工具的数据集来代表内围观测的总体。

2.7.3.1. Fitting an elliptic envelope（椭圆模型拟合）

实现离群点检测的一种常见方式是假设常规数据来自已知分布（例如，数据服从高斯分布）。 从这个假设来看，我们通常试图定义数据的 “形状”，并且可以将偏远观测(outlying observation)定义为足够远离拟合形状的观测。

scikit-learn 提供了 `covariance.EllipticEnvelope` 对象，它能拟合出数据的稳健协方差估计，从而为中心数据点拟合出一个椭圆，忽略不和该中心模式相关的点。

例如，假设内围数据服从高斯分布，它将稳健地（即不受异常值的影响）估计内围位置和协方差。 从该估计得到的马氏距离用于得出偏远性(outlyingness)度量。 该策略如下图所示。



示例：

- 参见 Robust covariance estimation and Mahalanobis distances relevance 说明对位置和协方差使用标准估计（ covariance.EmpiricalCovariance ）或稳健估计（ covariance.MinCovDet ）来评估观测值的偏远性的差异。

参考资料：

- Rousseeuw, P.J., Van Driessen, K. “A fast algorithm for the minimum covariance determinant estimator” Technometrics 41(3), 212 (1999)

2.7.3.2. Isolation Forest（隔离森林）

在高维数据集中实现离群点检测的一种有效方法是使用随机森林。 ensemble.IsolationForest 通过随机选择一个特征，然后随机选择所选特征的最大值和最小值之间的分割值来"隔离"观测。

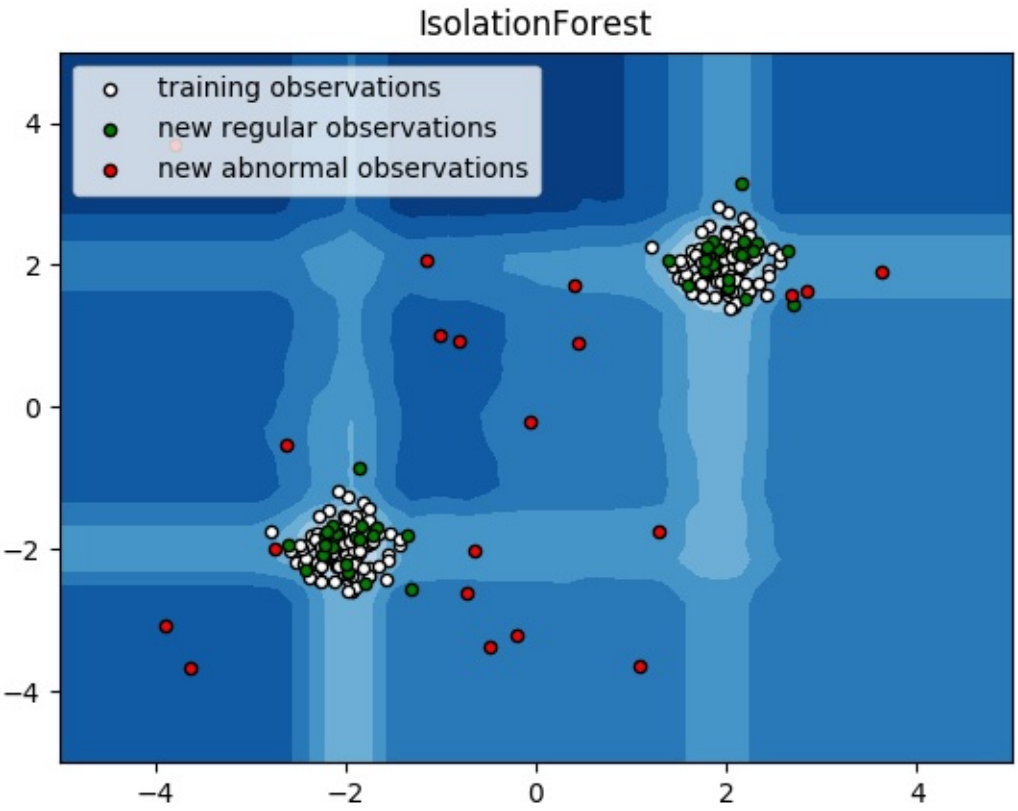
由于递归划分可以由树形结构表示，因此隔离样本所需的分割次数等同于从根节点到终止节点的路径长度。

在这样的随机树的森林中取平均的路径长度是数据正态性和我们的决策功能的量度。

随机划分能为异常观测产生明显的较短路径。 因此，当随机树的森林共同地为特定样本产生较短的路径长度时，这些样本就很有可能是异常的。

ensemble.IsolationForest的实现，是基于tree.ExtraTreeRegressor全体.按照隔离森林的原始论文，每棵树的最大深度设置为$\log_2(n)$,其中n是构建树所用的样本数量（详见: (Liu et al., 2008) ）

该算法在已下示例中进行了说明。



`ensemble.IsolationForest` 支持 `warm_start=True` ，这让你可以添加更多的树到一个已拟合好的模型中：

```
>>> from sklearn.ensemble import # 2.7.4Forest
>>> import numpy as np
>>> X = np.array([[-1, -1], [-2, -1], [-3, -2], [0, 0], [-20, 50], [3, 5]])
>>> clf = IsolationForest(n_estimators=10, warm_start=True)
>>> clf.fit(X)  # fit 10 trees
>>> clf.set_params(n_estimators=20)  # add 10 more trees
>>> clf.fit(X)  # fit the added trees
```

示例：

- 参见 IsolationForest example 说明隔离森林的用法。
- 参见 Outlier detection with several methods. 比较 `ensemble.IsolationForest` 与 `neighbors.LocalOutlierFactor` ， `svm.OneClassSVM` （调整为执行类似离群点检测的方法）和基于协方差使用 `covariance.EllipticEnvelope` 进行离群点检测。

参考资料：

- Liu, Fei Tony, Ting, Kai Ming and Zhou, Zhi-Hua. “Isolation forest.” Data Mining, 2008. ICDM‘08. Eighth IEEE International Conference on.

2.7.3.3. Local Outlier Factor（局部离群因子）

对轻度高维数据集(即维数勉强算是高维)实现异常值检测的另一种有效方法是使用局部离群因子（LOF）算法。

`neighbors.LocalOutlierFactor` （LOF）算法计算出反映观测异常程度的得分（称为局部离群因子）。 它测量给定数

据点相对于其邻近点的局部密度偏差。 算法思想是检测出具有比其邻近点明显更低密度的样本。

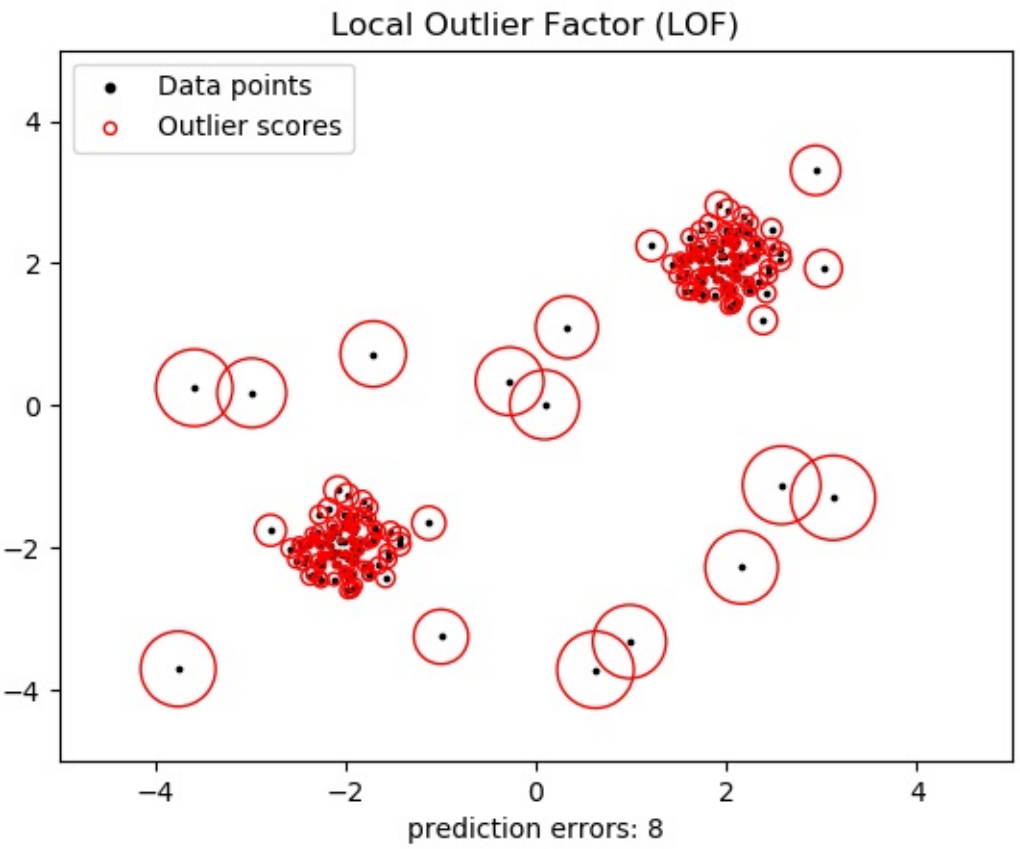
实际上，局部密度从 k 个最近邻得到。 观测数据的 LOF 得分等于其 k 个最近邻的平均局部密度与其本身密度的比值：正常情况预期与其近邻有着类似的局部密度，而异常数据则预计比近邻的局部密度要小得多。

考虑的k个近邻数（别名参数 n_neighbors ）通常选择 1) 大于一个聚类簇必须包含对象的最小数量，以便其它对象可以成为该聚类簇的局部离散点，并且 2) 小于可能成为聚类簇对象的最大数量，减少这K个近邻成为离群点的可能性。在实践中，这样的信息通常不可用，并且使 n_neighbors = 20 似乎通常都能使得算法有很好的表现。 当离群点的比例较高时（即大于 10% 时，如下面的示例），n_neighbors 应该较大（在下面的示例中，n_neighbors = 35）。

LOF 算法的优点是考虑到数据集的局部和全局属性：即使在具有不同潜在密度的离群点数据集中，它也能够表现得很好。 问题不在于样本是如何被分离的，而是样本与周围近邻的分离程度有多大。

当使用 LOF 进行离群点检测的时候，不能使用 predict, decision*function* 和 *score_samples* 方法， 只能使用 *fit_predict* 方法。训练样本的异常性得分可以通过 *negative_outlier_factor* 属性来获得。 注意 当使用LOF算法进行新奇点检测的时候(novelty 设为 True)， predict, decision_function 和 score_samples 函数可被用于新的未见过数据。请查看使用LOF进行新奇点检测.

该策略如下图所示。



示例：

- 参见 Anomaly detection with Local Outlier Factor (LOF) 是 neighbors.LocalOutlierFactor 使用示例。
- 参见 Outlier detection with several methods. 与其它异常检测方法进行比较。

参考资料：

- Breunig, Kriegel, Ng, and Sander (2000) LOF: identifying density-based local outliers. Proc. ACM SIGMOD

2.7.4. 使用LOF进行新奇点检测

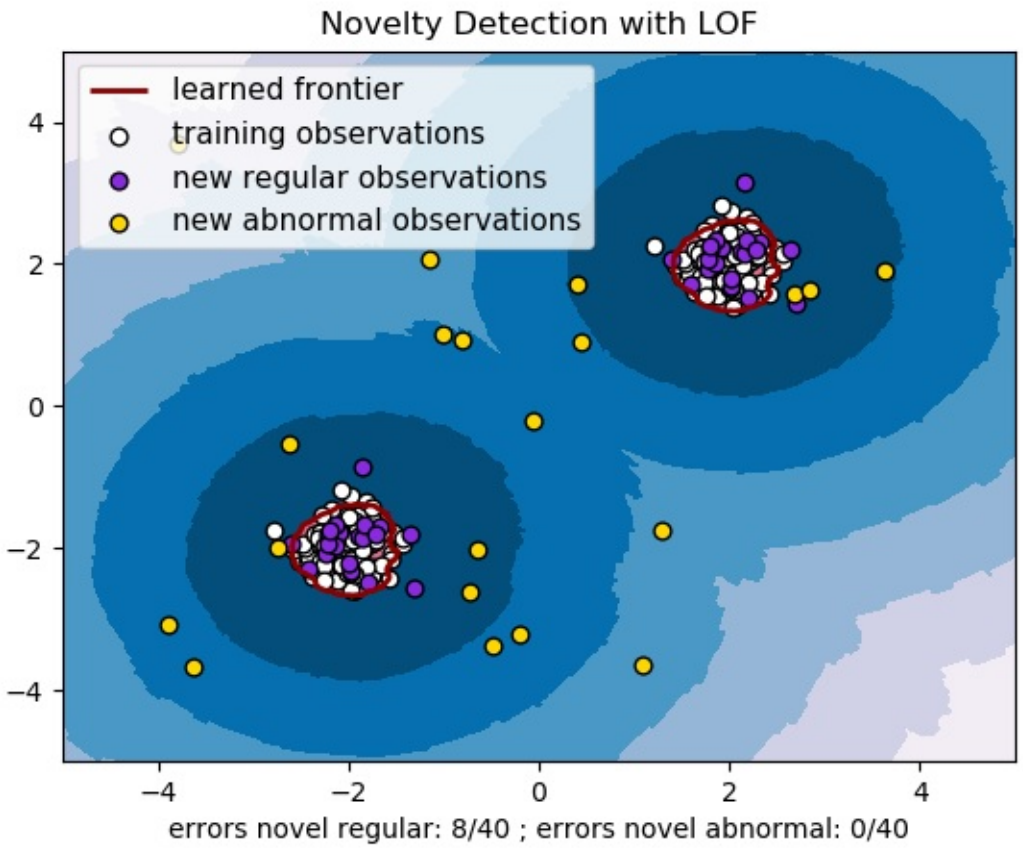
为了使用 `neighbors.LocalOutlierFactor` 类进行新奇点检测，即对新的未见过的样本 预测其标签或计算其异常性得分，你必须在实例化估计器时，将其 `novelty` 参数设为 `True`，这一步必须要在拟合之前完成：

```
lof = LocalOutlierFactor(novelty=True)
lof.fit(X_train)
```

请注意 `fit_predict` 方法在这情况下就不可用了。

> 警告：使用局部离群因子(Local Outlier Factor,LOF)进行新奇点检测
>
> 当 `novelty` 参数被设为 True 时，要当心。在新的未见过的数据上，你只能使用 `predict`，`decisionfunction` 和 `score_samples`，而不能把这几个函数用在训练数据上，否则会导致错误的结果。训练样本的异常性得分总是可以通过 `negative_outlier_factor` 属性来访问获取。

使用局部离群因子(LOF)进行新奇点检测的示例见下图。



我们一直在努力

apachecn/sklearn-doc-zh


加入QQ群

2.8. 密度估计

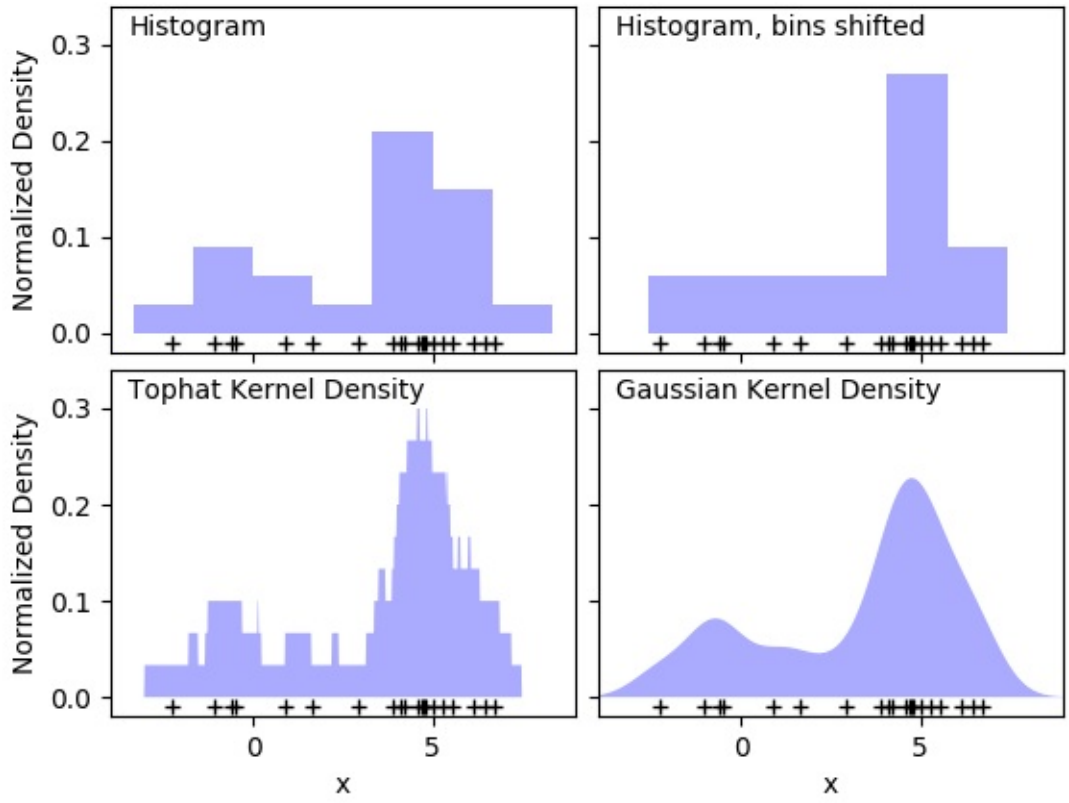
校验者：@不将就@barrycg翻译者：@Xi

密度估计涉及了无监督学习，特征工程和数据建模这三个不同的领域。一些最流行和最有用的密度估计方法是混合模型，如高斯混合(`sklearn.mixture.GaussianMixture`)，和基于邻近的方法，如核密度估计 `sklearn.neighbors.KernelDensity` 。 `clustering` 一节中更充分地讨论了高斯混合，因为此方法也用作为一种无监督聚类方案。

密度估计是一个非常简单的概念，大多数人已经熟悉了其中一种常用的密度估计技术：直方图。

2.8.1. 密度估计：直方图

直方图是一种简单的数据可视化方法，其中定义了组(bins)，并且统计了每个组(bin)中的数据点的数量。在下图的左上角中可以看到一个直方图的例子：



然而，直方图的一个主要问题是分组(binning)的选择可能会对得到的可视化结果造成不相称的影响。考虑上图中右上角的图，它显示了相同数据下,组(bins)向右移动后的直方图。这两个可视化的结果看起来完全不同，可能会导致对数据作出不同的解释。

直观地说，你也可以把一个直方图看成由一堆块组成，每个点上放一个块. 通过在合适的网格空间中堆积这些块，我们就可以得到直方图。但是，如果不是把这些块堆叠在一个规则的网格上，而是把每个块定位在它所代表的点上，然后把每个位置的总高度相加呢?这样可以得到如上图左下角所示的可视化.它可能不像直方图那样整洁，但是由数据决

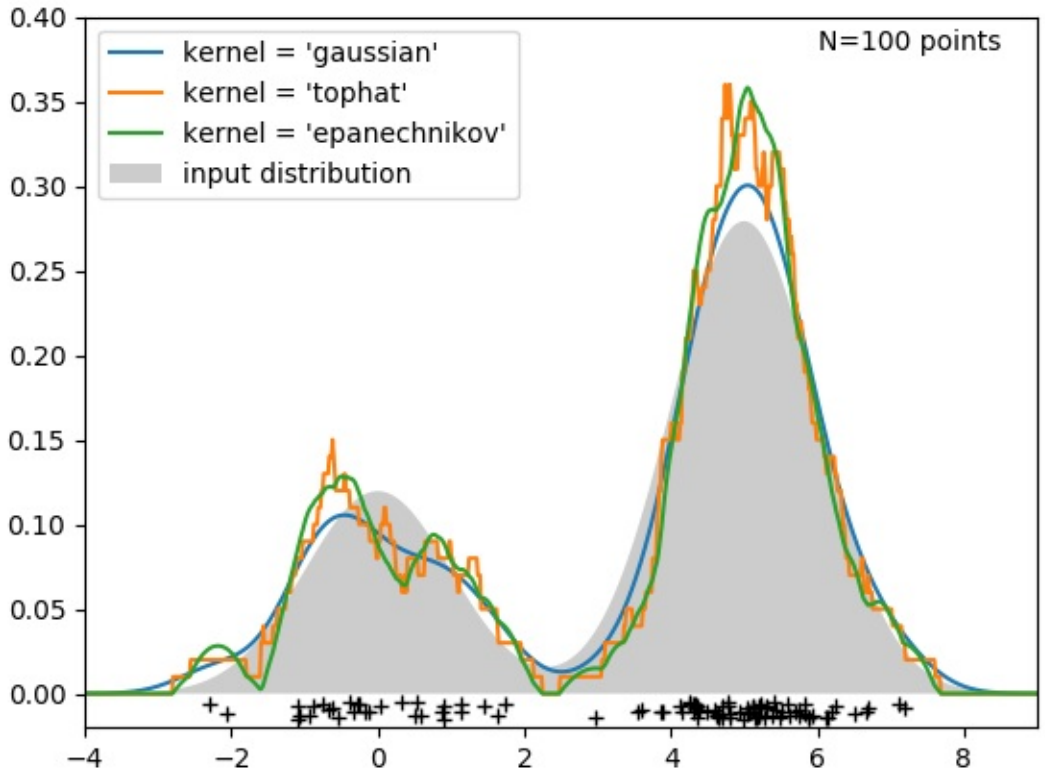
定块的位置意味着它能更好地表示基本的数据。

这个可视化是核密度估计的一个例子，该例中用的是一种顶帽核(top-hat kernel)（即每个点上放一个方块）。我们可以通过使用一种更平滑的核来得到一个更平滑的分布。上图右下角展示了一个高斯核密度估计，其中每个点都给总的分布贡献一条高斯曲线。结果是从数据中得到了一个平滑的密度估计，并且可作为一个强大的非参数的点分布模型。

2.8.2. 核密度估计

`sklearn.neighbors.KernelDensity` 实现了 scikit-learn 中的核密度估计，它使用 Ball Tree 或 KD Tree 来进行高效查询（有关这些讨论请参见 最近邻 ）。尽管为了简单起见上述示例采用的是一维数据集，但核密度估计能够用在任意维度上，不过在实际应用中,维数灾难(the curse of dimensionality)会导致其在高维上的性能降低。

如下图所示，从双峰分布中绘制了100个点，并展示了选用三个不同核的核密度估计：



图中可以很明显地看到核的形状如何影响结果分布的平滑度. 使用 scikit-learn 核密度估计的方法如下所示：

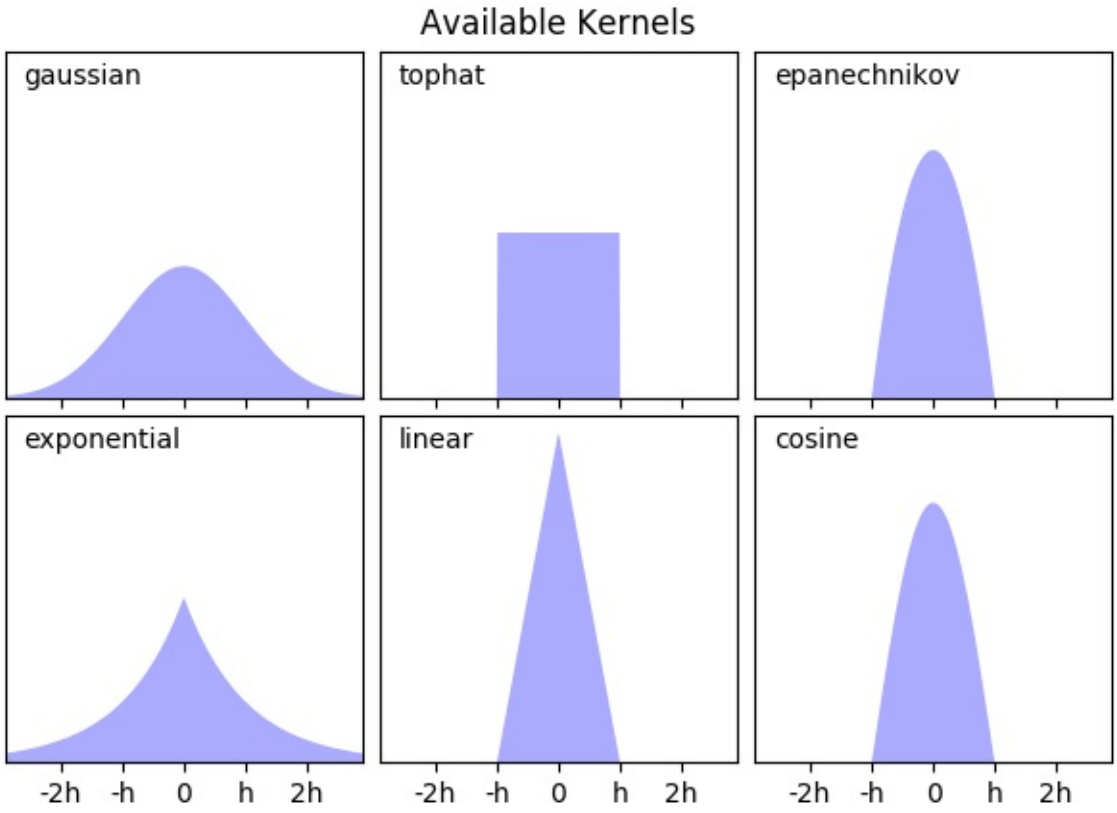
```
>>> from sklearn.neighbors.kde import KernelDensity
>>> import numpy as np
>>> X = np.array([[-1, -1], [-2, -1], [-3, -2], [1, 1], [2, 1], [3, 2]])
>>> kde = KernelDensity(kernel='gaussian', bandwidth=0.2).fit(X)
>>> kde.score_samples(X)
array([-0.41075698, -0.41075698, -0.41076071, -0.41075698, -0.41075698,
 -0.41076071])
```

如上所示,这里我们选用的是高斯核 `kernel='gaussian'` .在数学上，核是由带宽参数 h 控制的正值函数 $K(x;h)$. 给定核的形状后,在一组点 $x_i; i = 1 \cdots N$ 内的 y 点处的密度估计由下式给出:

$$\rho_K(y) = \sum_{i=1}^{N} K((y - x_i)/h)$$

这里的带宽作为平滑参数，用来平衡结果中偏差和方差的值。 大的带宽会导致非常平滑（即高偏差）密度分布,而小的带宽则导致不平滑（即高方差）密度分布。

`sklearn.neighbors.KernelDensity` 实现了一些常见形状的核， 如下图所示:



这些核的形式如下所示:

- Gaussian kernel (`kernel = 'gaussian'`)

$$K(x;h) \propto \exp(-\frac{x^2}{2h^2})$$

- Tophat kernel (`kernel = 'tophat'`)

$$K(x;h) \propto 1 \text{ if } x < h$$

- Epanechnikov kernel (`kernel = 'epanechnikov'`)

$K(x; h) \propto 1 - \frac{x^2}{h^2}$

- Exponential kernel (`kernel = 'exponential'`)

$K(x; h) \propto \exp(-x/h)$

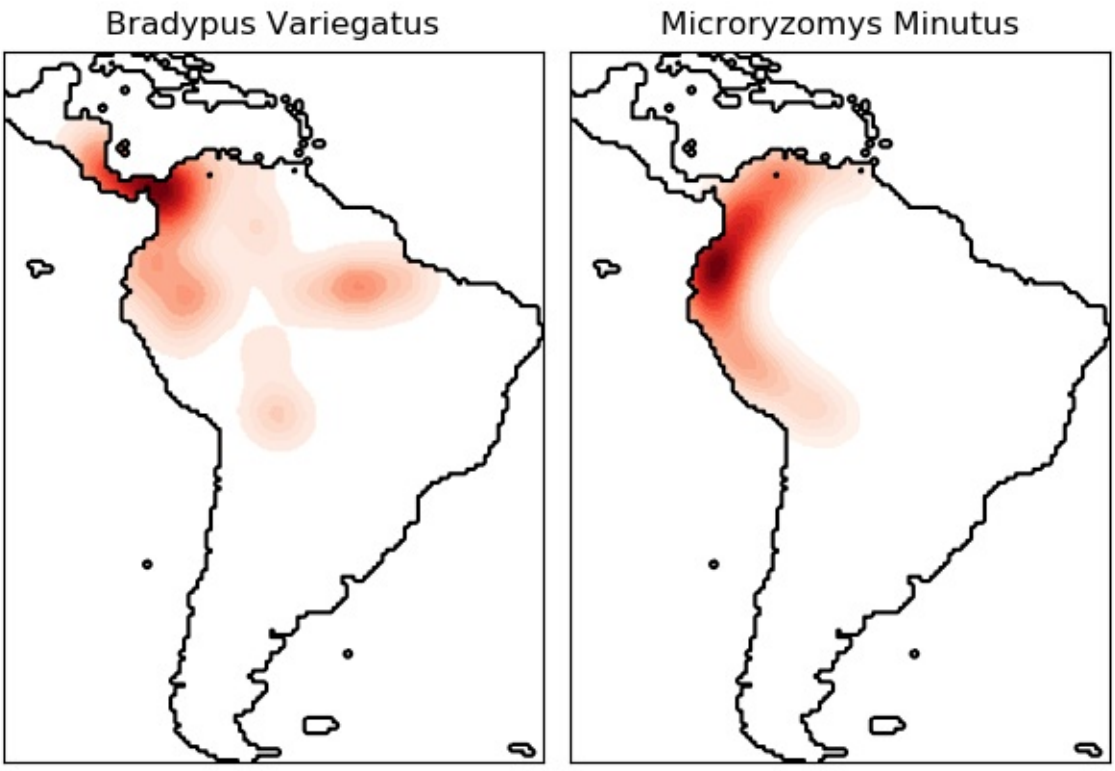
- Linear kernel (`kernel = 'linear'`)

$K(x; h) \propto 1 - x/h$ if $x < h$

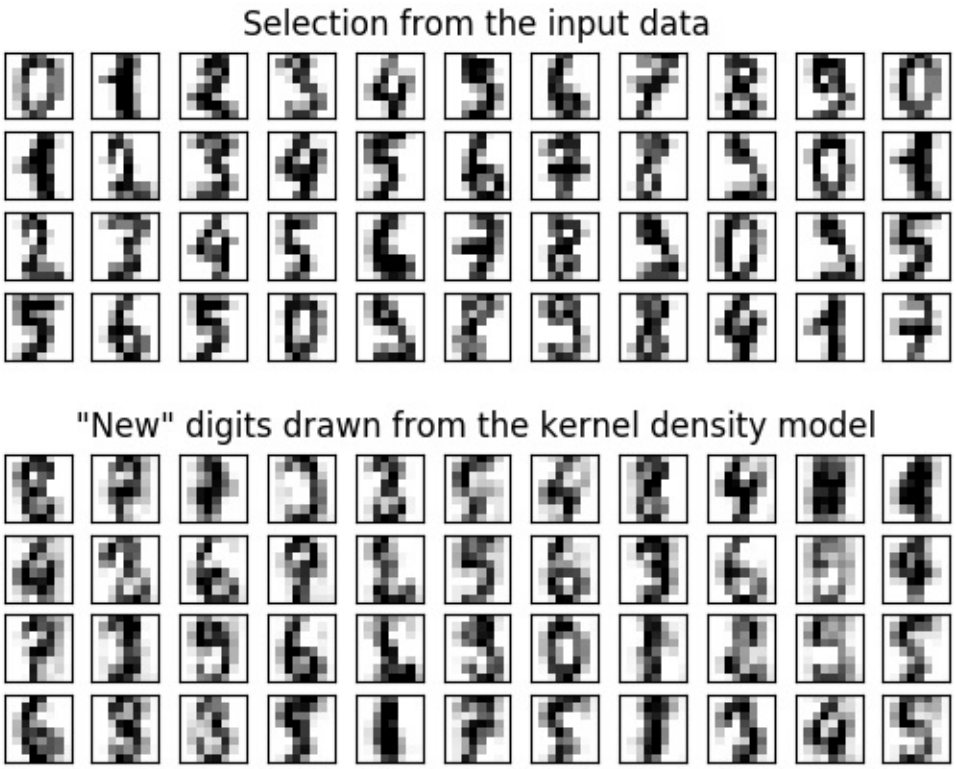
- Cosine kernel (`kernel = 'cosine'`)

$K(x; h) \propto \cos(\frac{\pi x}{2h})$ 如果 $x < h$

核密度估计可以与任何有效的距离度量一起使用（可用度量列表请参见 `sklearn.neighbors.DistanceMetric` ），但其结果只能被欧氏度量进行合适地归一化。 一个特别有用的度量是测量球体上的点与点之间角距离 的 Haversine distance 。 下面是使用核密度估计来对地理空间数据进行可视化的示例，本例中南美大陆两种不同物种的观测分布如图：



核密度估计的另一个有用的应用是从数据集中学习出一个非参数生成模型，以便有效地从该生成模型中绘制新的样本。 以下是使用此过程创建一组新的手写数字的示例，使用的是高斯核对数据的 PCA 投影进行学习：



“新”数据由输入数据线性组合而成，其权重根据 KDE 模型按概率给出。

> 示例：
>
> - Simple 1D Kernel Density Estimation: 一维简单核密度估计的计算。
> - Kernel Density Estimation: 使用核密度估计来学习手写数字数据生成模型，以及使用该模型绘制新样本的示例
> - Kernel Density Estimate of Species Distributions: 使用Haversine距离度量来显示地理空间数据的核密度估计示例.

我们一直在努力

apachecn/sklearn-doc-zh

加入QQ群

2.9. 神经网络模型（无监督）

校验者：@不将就@Loopy@barrycg翻译者：@夜神月

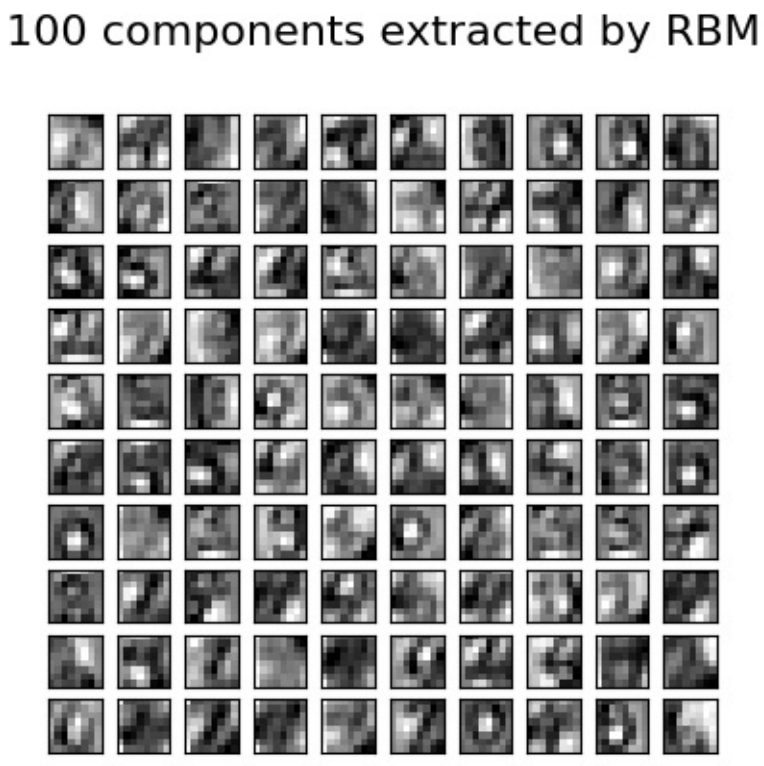
2.9.1. 限制波尔兹曼机

限制玻尔兹曼机 (Restricted Boltzmann machines, 简称RBM)是基于概率模型的无监督非线性特征学习器。当用 RBM 或 多层次结构的RBMs 提取的特征在馈入线性分类器（如线性支持向量机或感知机）时通常会获得良好的结果。

该模型对输入的分布作出假设。目前，scikit-learn 只提供了 `BernoulliRBM` ，它假定输入是二值(binary values)的，或者是 0 到 1 之间的值，每个值都编码特定特征被激活的概率。

RBM 尝试使用特定图形模型最大化数据的可能性(the likelihood of the data)。所使用的参数学习算法（Stochastic Maximum Likelihood （随机最大似然））防止特征表示偏离输入数据，这使得它们能学习到有趣的特征，但使得该模型对于小数据集不太有用且通常对于密度估计无效。

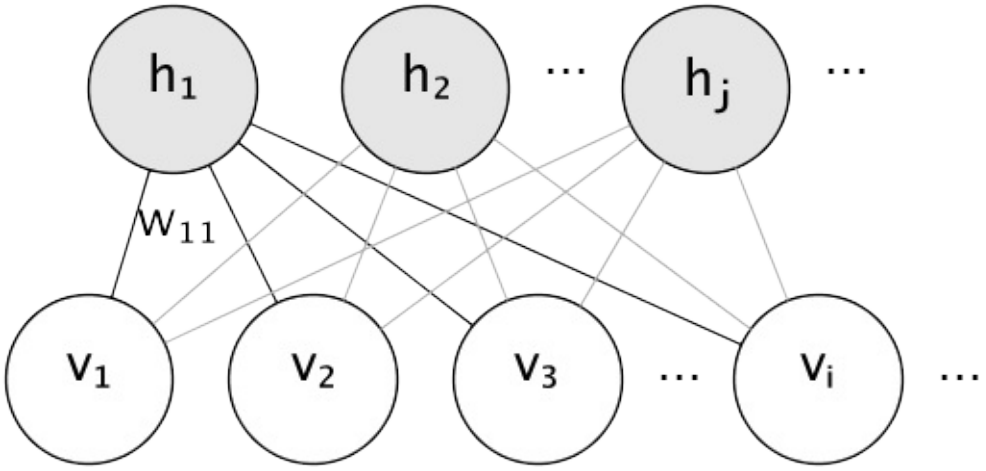
该方法因为深层神经网络需要独立RBM的权重来初始化而普及。这种方法被称为无监督的预训练（unsupervised pre-training）。



> 示例：
>
> - Restricted Boltzmann Machine features for digit classification

2.9.1.1. 图形模型和参数化

RBM 的图形模型是一个全连接的二分图（fully-connected bipartite graph）。



节点是随机变量，其状态取决于它连接到的其他节点的状态。这个模型可通过连接的权重值进行参数化，同时每个可见或隐藏单元都有一个偏置项(biased term)，为了简单起见，上图中的偏置项被省略了。

用能量函数衡量联合概率分布的质量：

$$E(\mathbf{v}, \mathbf{h}) = \sum_i \sum_j w_{ij} v_i h_j + \sum_i b_i v_i + \sum_j c_j h_j$$

在上面的公式中，$\mathbf{b}$ 和 $\mathbf{c}$ 分别是可见层和隐藏层的偏置向量。 模型的联合概率是根据能量来定义的：

$$P(\mathbf{v}, \mathbf{h}) = \frac{e^{-E(\mathbf{v}, \mathbf{h})}}{Z}$$

“限制”是指模型的二分图结构，它禁止隐藏单元之间或可见单元之间的直接交互。 这代表以下条件独立性成立：

$$h_i \perp h_j | \mathbf{v}$$
$$v_i \perp v_j | \mathbf{h}$$

二分图结构允许使用高效的块吉比斯采样(block Gibbs sampling)进行推断。

2.9.1.2. 伯努利限制玻尔兹曼机

在 `BernoulliRBM` 中，所有单位都是二进制随机单元。 这意味着输入数据应该是二值，或者是在 0 和 1 之间的实数值，其表示可见单元活跃或不活跃的概率。 这是一个很好的字符识别模型，其中的关注点是哪些像素是活跃的，哪些不是。 对于自然场景的图像，它不再适合，因为背景，深度和相邻像素的趋势取相同的值。

每个单位的条件概率分布由其接收的输入的 logistic sigmoid函数给出：

$$P(v_i = 1 | \mathbf{h}) = \sigma(\sum_j w_{ij} h_j + b_i)$$
$$P(h_i = 1 | \mathbf{v}) = \sigma(\sum_i w_{ij} v_i + c_j)$$

其中 σ 是 logistic sigmoid函数:

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

2.9.1.3. 随机最大似然学习

在 `BernoulliRBM` 函数中实现的训练算法被称为随机最大似然（Stochastic Maximum Likelihood (SML)）或持续对比发散（Persistent Contrastive Divergence (PCD)）。由于数据的似然函数的形式，直接优化最大似然是不可行的:

$$\log P(v) = \log \sum_h e^{-E(v,h)} - \log \sum_{x,y} e^{-E(x,y)}$$

为了简单起见，上面的等式是针对单个训练样本所写的。相对于权重的梯度由对应于上述的两个项构成。根据它们的符号，它们通常被称为正梯度和负梯度。在这种实现中，按照小批量样本（mini-batches of samples ）对梯度进行计算。

在最大化对数似然度(maximizing the log-likelihood)的情况下，正梯度使模型更倾向于与观察到的训练数据兼容的隐藏状态。由于 RBM 的二分体结构，可以高效地计算。然而，负梯度是棘手的。其目标是降低模型偏好的联合状态的能量，从而使数据保持真实。可以通过马尔可夫链蒙特卡罗近似，使用块吉比斯采样，通过迭代地对每个 v 和 h 进行交互采样，直到链混合。以这种方式产生的样本有时被称为幻想粒子。这是无效的，很难确定马可夫链是否混合。

对比发散方法建议在经过少量迭代后停止链，迭代数 k 通常为 1.该方法快速且方差小，但样本远离模型分布。

持续对比发散(PCD)解决了这个问题。而不是每次需要梯度时都启动一个新的链，并且只执行一个吉比斯采样步骤，

在 PCD 中，我们保留了多个链（幻想粒子）,每个 链,在每个权重更新之后， 执行k个吉比斯采样步骤。这使得粒子能更彻底地探索空间.

> 参考资料:
>
> - “A fast learning algorithm for deep belief nets” G. Hinton, S. Osindero, Y.-W. Teh, 2006
> - “Training Restricted Boltzmann Machines using Approximations to the Likelihood Gradient” T. Tieleman, 2008

我们一直在努力

apachecn/sklearn-doc-zh

加入QQ群

3. 模型选择和评估

- 3.1. 交叉验证：评估估算器的表现
  - 3.1.1. 计算交叉验证的指标
  - 3.1.2. 交叉验证迭代器
  - 3.1.3. A note on shuffling
  - 3.1.4. 交叉验证和模型选择
- 3.2. 调整估计器的超参数
  - 3.2.1. 网格追踪法-穷尽的网格搜索
  - 3.2.2. 随机参数优化
  - 3.2.3. 参数搜索技巧
  - 3.2.4. 暴力参数搜索的替代方案
- 3.3. 模型评估：量化预测的质量
  - 3.3.1. `scoring` 参数：定义模型评估规则
  - 3.3.2. 分类指标
  - 3.3.3. 多标签排名指标
  - 3.3.4. 回归指标
  - 3.3.5. 聚类指标
  - 3.3.6. 虚拟估计
- 3.4. 模型持久化
  - 3.4.1. 持久化示例
  - 3.4.2. 安全性和可维护性的局限性
- 3.5. 验证曲线：绘制分数以评估模型
  - 3.5.1. 验证曲线
  - 3.5.2. 学习曲线

我们一直在努力

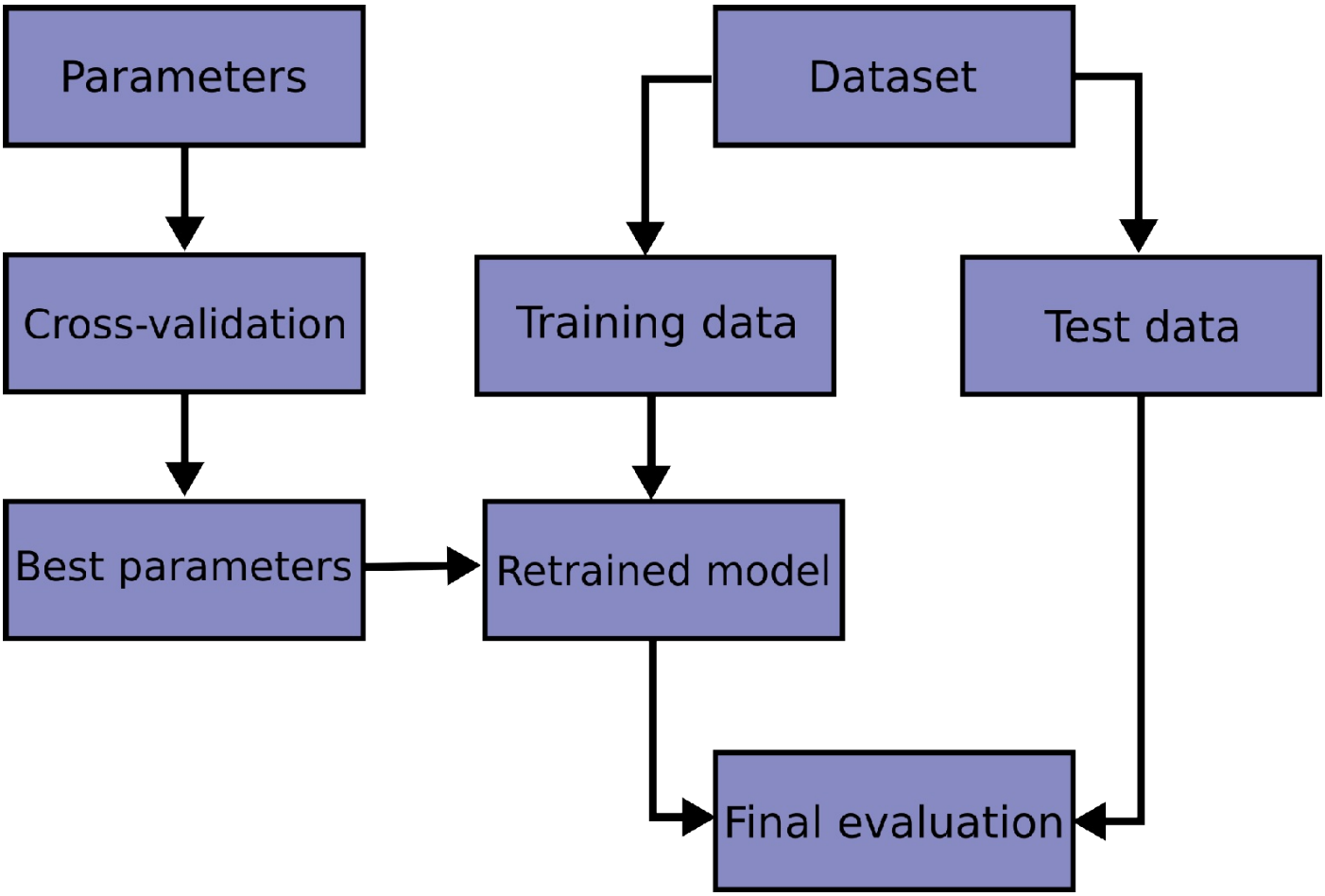
apachecn/sklearn-doc-zh



3.1. 交叉验证：评估估算器的表现

校验者：@想和太阳肩并肩@樊雯@Loopy翻译者：@\S^R^Y/

学习预测函数的参数，并在相同数据集上进行测试是一种错误的做法：一个仅给出测试用例标签的模型将会获得极高的分数，但对于尚未出现过的数据它则无法预测出任何有用的信息。这种情况称为 **overfitting**（过拟合）. 为了避免这种情况，在进行（监督）机器学习实验时，通常取出部分可利用数据作为 **test set**（测试数据集） `X_test, y_test` 。需要强调的是这里说的 “experiment(实验)” 并不仅限于学术（academic），因为即使是在商业场景下机器学习也往往是从实验开始的。下面是模型训练中典型的交叉验证工作流流程图。通过网格搜索可以确定最佳参数。



利用 scikit-learn 包中的 `train_test_split` 辅助函数可以很快地将实验数据集划分为任何训练集（training sets）和测试集（test sets）。下面让我们载入 iris 数据集，并在此数据集上训练出线性支持向量机：

```
>>> import numpy as np
>>> from sklearn.model_selection import train_test_split
>>> from sklearn import datasets
>>> from sklearn import svm

>>> iris = datasets.load_iris()
>>> iris.data.shape, iris.target.shape
((150, 4), (150,))
```

我们能快速采样到原数据集的 40% 作为测试集，从而测试（评估）我们的分类器：

```
>>> X_train, X_test, y_train, y_test = train_test_split(
...     iris.data, iris.target, test_size=0.4, random_state=0)

>>> X_train.shape, y_train.shape
((90, 4), (90,))
>>> X_test.shape, y_test.shape
((60, 4), (60,))

>>> clf = svm.SVC(kernel='linear', C=1).fit(X_train, y_train)
>>> clf.score(X_test, y_test)
0.96...
```

当评价估计器的不同设置（”hyperparameters(超参数)”）时，例如手动为 SVM 设置的 `C` 参数， 由于在训练集上，通过调整参数设置使估计器的性能达到了最佳状态；但 在测试集上 可能会出现过拟合的情况。 此时，测试集上的信息反馈足以颠覆训练好的模型，评估的指标不再有效反映出模型的泛化性能。 为了解决此类问题，还应该准备另一部分被称为 “validation set(验证集)” 的数据集，模型训练完成以后在验证集上对模型进行评估。 当验证集上的评估实验比较成功时，在测试集上进行最后的评估。

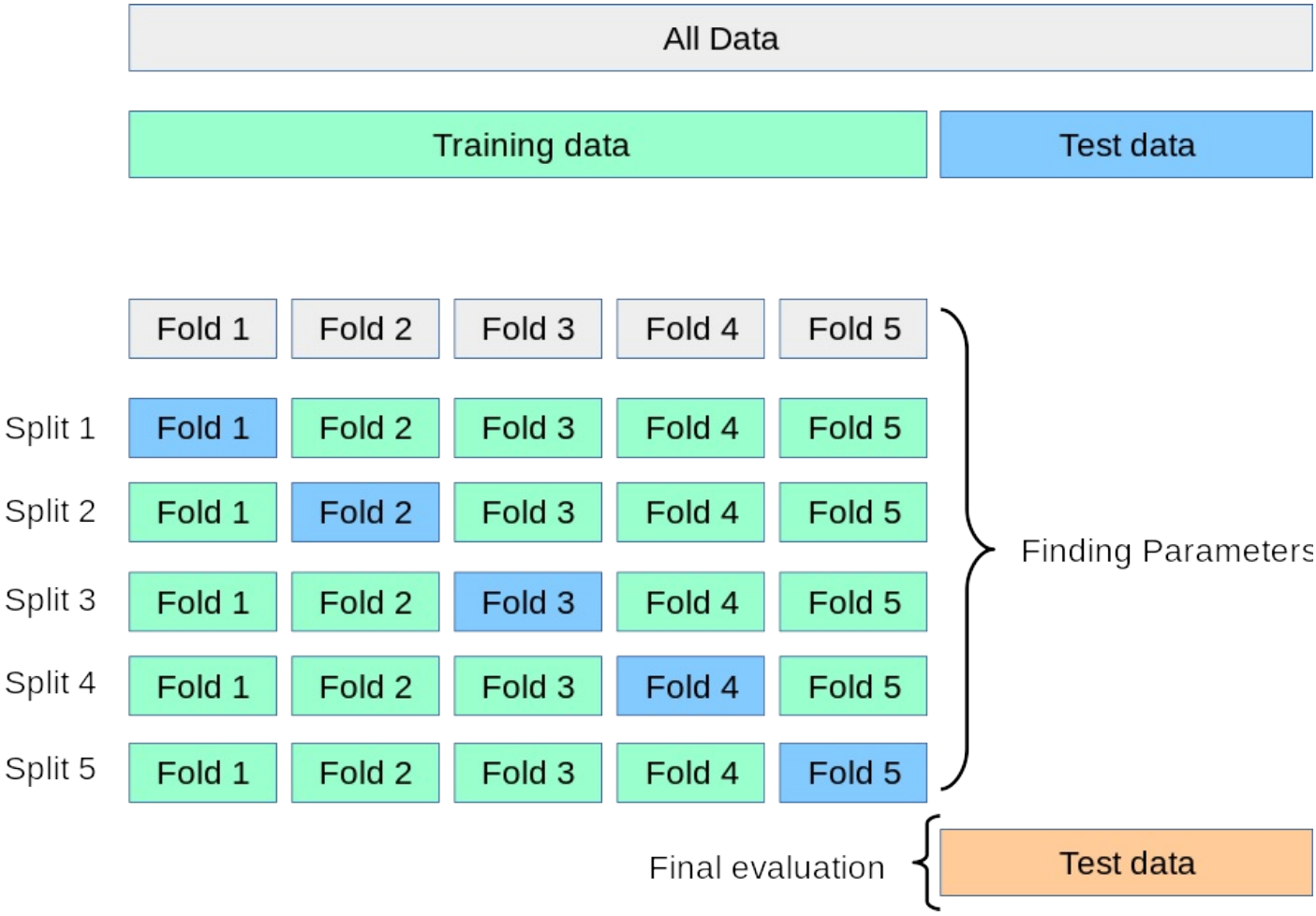
然而，通过将原始数据分为3个数据集合，我们就大大减少了可用于模型学习的样本数量， 并且得到的结果依赖于集合对（训练，验证）的随机选择。

这个问题可以通过 交叉验证（CV ） 来解决。 交叉验证仍需要测试集做最后的模型评估，但不再需要验证集。

最基本的方法被称之为，*k*-折交叉验证 。 k-折交叉验证将训练集划分为 k 个较小的集合（其他方法会在下面描述，主要原则基本相同）。 每一个 *k* 折都会遵循下面的过程：

- 将 $k-1$ 份训练集子集作为 training data （训练集）训练模型，
- 将剩余的 1 份训练集子集用于模型验证（也就是把它当做一个测试集来计算模型的性能指标，例如准确率）。

k-折交叉验证得出的性能指标是循环计算中每个值的平均值。 该方法虽然计算代价很高，但是它不会浪费太多的数据（如固定任意测试集的情况一样）， 在处理样本数据集较少的问题（例如，逆向推理）时比较有优势。



3.1.1. 计算交叉验证的指标

使用交叉验证最简单的方法是在估计器和数据集上调用 `cross_val_score` 辅助函数。

下面的例子展示了如何通过分割数据，拟合模型和计算连续 5 次的分数（每次不同分割）来估计 linear kernel 支持向量机在 iris 数据集上的精度：

```
>>> from sklearn.model_selection import cross_val_score
>>> clf = svm.SVC(kernel='linear', C=1)
>>> scores = cross_val_score(clf, iris.data, iris.target, cv=5)
>>> scores
array([0.96..., 1.  ..., 0.96..., 0.96..., 1.        ])
```

评分估计的平均得分和 95% 置信区间由此给出：

```
>>> print("Accuracy: %0.2f (+/- %0.2f)" % (scores.mean(), scores.std() * 2))
Accuracy: 0.98 (+/- 0.03)
```

默认情况下，每个 CV 迭代计算的分数是估计器的 `score` 方法。可以通过使用 scoring 参数来改变计算方式如下：

```
>>> from sklearn import metrics
>>> scores = cross_val_score(
```

```
...     clf, iris.data, iris.target, cv=5, scoring='f1_macro')
>>> scores
array([0.96..., 1.  ..., 0.96..., 0.96..., 1.        ])
```

详情请参阅 scoring 参数：定义模型评估规则 。 在 Iris 数据集的情形下，样本在各个目标类别之间是平衡的，因此准确度和 F1-score 几乎相等。

当 `cv` 参数是一个整数时， `cross_val_score` 默认使用 `KFold` 或 `StratifiedKFold` 策略，后者会在估计器派生自 `ClassifierMixin` 时使用。

也可以通过传入一个交叉验证迭代器来使用其他交叉验证策略，比如：

```
>>> from sklearn.model_selection import ShuffleSplit
>>> n_samples = iris.data.shape[0]
>>> cv = ShuffleSplit(n_splits=5, test_size=0.3, random_state=0)
>>> cross_val_score(clf, iris.data, iris.target, cv=cv)
array([0.977..., 0.977..., 1.  ..., 0.955..., 1.        ])
```

另外一种可选方案是使用一个可迭代生成器作为索引数组产生(train, test) 划分，比如：

```
>>> def custom_cv_2folds(X):
...     n = X.shape[0]
...     i = 1
...     while i <= 2:
...         idx = np.arange(n * (i - 1) / 2, n * i / 2, dtype=int)
...         yield idx, idx
...         i += 1
...
>>> custom_cv = custom_cv_2folds(iris.data)
>>> cross_val_score(clf, iris.data, iris.target, cv=custom_cv)
array([1.        , 0.973...])
```

保留数据的数据转换

正如在训练集中保留的数据上测试一个 predictor （预测器）是很重要的一样，预处理（如标准化，特征选择等）和类似的 data transformations 也应该从训练集中学习，并应用于预测数据以进行预测：

```
>> from sklearn import preprocessing>> X_train, X_test, y_train, y_test = train_test_split(  …
iris.data, iris.target, test_size=0.4, random_state=0)>> scaler =
preprocessing.StandardScaler().fit(X_train)>> X_train_transformed = scaler.transform(X_train)>> clf =
svm.SVC(C=1).fit(X_train_transformed, y_train)>> X_test_transformed = scaler.transform(X_test)>>
clf.score(X_test_transformed, y_test)  0.9333…
```

`Pipeline` 可以更容易地组合估计器，在交叉验证下使用如下：

```
>> from sklearn.pipeline import make_pipeline>> clf = make_pipeline(preprocessing.StandardScaler(),
svm.SVC(C=1))>> cross_val_score(clf, iris.data, iris.target, cv=cv)  …  array([ 0.97…,  0.93…,  0.95…])
```

可以参阅 Pipeline（管道）和 FeatureUnion（特征联合）：合并的评估器.

3.1.1.1. cross_validate 函数和多度量评估

`cross_validate` 函数与 `cross_val_score` 在下面的两个方面有些不同 -

- 它允许指定多个指标进行评估.
- 除了测试得分之外，它还会返回一个包含训练得分，拟合次数， score-times （得分次数）的一个字典。It returns a dict containing training scores, fit-times and score-times in addition to the test score.

对于单个度量评估，其中 scoring 参数是一个字符串，可以调用或 None ， keys 将是 - `['test_score', 'fit_time', 'score_time']`

而对于多度量评估，返回值是一个带有以下的 keys 的字典 - `['test<scorer1_name>', 'test<scorer2name>', 'test<scorer…>', 'fit_time', 'score_time']`

`return_train_score` 默认设置为 `True` 。 它增加了所有 scorers(得分器) 的训练得分 keys 。如果不需要训练 scores ，则应将其明确设置为 `False` 。

你还可以通过设置 `return_estimator=True` 来保留在所有训练集上拟合好的估计器。

可以将多个测度指标指定为list，tuple或者是预定义评分器(predefined scorer)的名字的集合

```
>>> from sklearn.model_selection import cross_validate
>>> from sklearn.metrics import recall_score
>>> scoring = ['precision_macro', 'recall_macro']
>>> clf = svm.SVC(kernel='linear', C=1, random_state=0)
>>> scores = cross_validate(clf, iris.data, iris.target, scoring=scoring,
...                         cv=5)
>>> sorted(scores.keys())
['fit_time', 'score_time', 'test_precision_macro', 'test_recall_macro']
>>> scores['test_recall_macro']
array([0.96..., 1.  ..., 0.96..., 0.96..., 1.        ])
```

或作为一个字典 mapping 得分器名称预定义或自定义的得分函数：

```
>>> from sklearn.metrics.scorer import make_scorer
>>> scoring = {'prec_macro': 'precision_macro',
...            'rec_macro': make_scorer(recall_score, average='macro')}
>>> scores = cross_validate(clf, iris.data, iris.target, scoring=scoring,
...                         cv=5, return_train_score=True)
>>> sorted(scores.keys())
['fit_time', 'score_time', 'test_prec_macro', 'test_rec_macro',
 'train_prec_macro', 'train_rec_macro']
>>> scores['train_rec_macro']
array([0.97..., 0.97..., 0.99..., 0.98..., 0.98...])
```

这里是一个使用单一指标的 `cross_validate` 的例子：

```
>>> scores = cross_validate(clf, iris.data, iris.target,
...                         scoring='precision_macro', cv=5,
...                         return_estimator=True)
```

```
>>> sorted(scores.keys())
['estimator', 'fit_time', 'score_time', 'test_score']
```

3.1.1.2. 通过交叉验证获取预测

除了返回结果不同，函数 `cross_val_predict` 具有和 `cross_val_score` 相同的接口， 对于每一个输入的元素，如果其在测试集合中，将会得到预测结果。交叉验证策略会将可用的元素提交到测试集合有且仅有一次（否则会抛出一个异常）。

> 警告 ：交叉预测可能使用不当
>
> `cross_val_predict` 函数的结果可能会与 `cross_val_score` 函数的结果不一样，因为在这两种方法中元素的分组方式不一样。函数 `cross_val_score` 在所有交叉验证的折子上取平均。但是，函数 `cross_val_predict` 只是简单的返回由若干不同模型预测出的标签或概率。因此， `cross_val_predict` 不是一种适当的泛化错误的度量。

函数 `cross_val_predict` 比较适合做下列事儿：

- 从不同模型获得的预测结果的可视化。
- 模型混合： 在集成方法中，当一个有监督估计量的预测被用来训练另一个估计量时

可用的交叉验证迭代器在下面的章节将提到。

> 示例
>
> - Receiver Operating Characteristic (ROC) with cross validation,
> - Recursive feature elimination with cross-validation,
> - Parameter estimation using grid search with cross-validation,
> - Sample pipeline for text feature extraction and evaluation,
> - 绘制交叉验证预测图,
> - Nested versus non-nested cross-validation.

3.1.2. 交叉验证迭代器

接下来的部分列出了一些用于生成索引标号，用于在不同的交叉验证策略中生成数据划分的工具。

3.1.2.1. 交叉验证迭代器-循环遍历数据

假设一些数据是独立的和相同分布的 (i.i.d) 假定所有的样本来源于相同的生成过程，并假设生成过程没有记忆过去生成的样本。

在这种情况下可以使用下面的交叉验证器。

> 注意
>
> 尽管 i.i.d 数据是机器学习理论中的一个常见假设，但在实践中很少成立。如果知道样本是使用时间相关的过程生成的，则使用 time-series aware cross-validation scheme 更安全。 同样，如果我们知道生成过程具有 group structure （群体结构）（从不同 subjects（主体），experiments（实验）， measurement devices （测量设备）收集的样本），则使用 group-wise cross-validation 更安全。

3.1.2.1.1. K 折

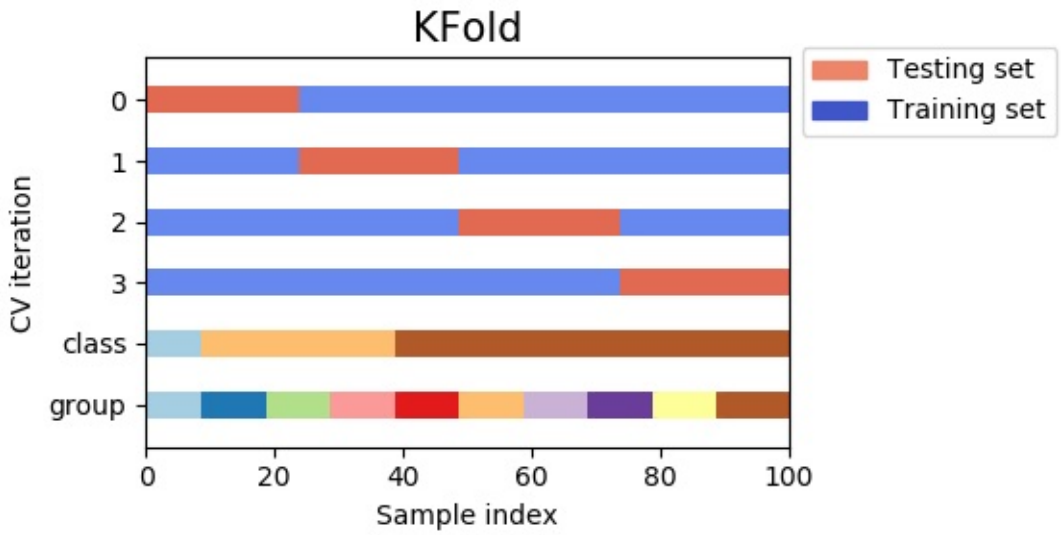
`KFold` 将所有的样例划分为 k 个组，称为折叠 (fold) (如果 $k = n$， 这等价于 *Leave One Out*(留一)策略)，都具有相同的大小(如果可能)。预测函数学习时使用 $k - 1$ 个折叠中的数据，最后一个剩下的折叠会用于测试。

在 4 个样例的数据集上使用 2-fold 交叉验证的例子：

```
>>> import numpy as np
>>> from sklearn.model_selection import KFold

>>> X = ["a", "b", "c", "d"]
>>> kf = KFold(n_splits=2)
>>> for train, test in kf.split(X):
...     print("%s  %s" % (train, test))
[2 3] [0 1]
[0 1] [2 3]
```

这个例子是关于交叉验证的可视化的。请注意KFold不被分类所影响.



每个折叠由两个 arrays 组成，第一个作为 *training set* ，另一个作为 *test set* 。 由此，可以通过使用 numpy 的索引创建训练/测试集合：

```
>>> X = np.array([[0., 0.], [1., 1.], [-1., -1.], [2., 2.]])
>>> y = np.array([0, 1, 0, 1])
>>> X_train, X_test, y_train, y_test = X[train], X[test], y[train], y[test]
```

3.1.2.1.2. 重复 K-折交叉验证

`RepeatedKFold` 重复 K-Fold n 次。当需要运行时可以使用它 `KFold` n 次，在每次重复中产生不同的分割。

2折 K-Fold 重复 2 次的示例：

```
>>> import numpy as np
>>> from sklearn.model_selection import RepeatedKFold
```

```
>>> X = np.array([[1, 2], [3, 4], [1, 2], [3, 4]])
>>> random_state = 12883823
>>> rkf = RepeatedKFold(n_splits=2, n_repeats=2, random_state=random_state)
>>> for train, test in rkf.split(X):
...     print("%s  %s" % (train, test))
...
[2 3] [0 1]
[0 1] [2 3]
[0 2] [1 3]
[1 3] [0 2]
```

类似地， `RepeatedStratifiedKFold` 在每个重复中以不同的随机化重复 n 次分层的 K-Fold 。

3.1.2.1.3. 留一交叉验证（LOO）

`LeaveOneOut` （或 LOO）是一个简单的交叉验证。每个学习集都是通过除了一个样本以外的所有样本创建的，测试集是被留下的样本。 因此，对于 n 个样本，我们有 n 个不同的训练集和 n 个不同的测试集。这种交叉验证程序不会浪费太多数据，因为只有一个样本是从训练集中删除掉的：

```
>>> from sklearn.model_selection import LeaveOneOut

>>> X = [1, 2, 3, 4]
>>> loo = LeaveOneOut()
>>> for train, test in loo.split(X):
...     print("%s  %s" % (train, test))
[1 2 3] [0]
[0 2 3] [1]
[0 1 3] [2]
[0 1 2] [3]
```

LOO 潜在的用户选择模型应该权衡一些已知的警告。 当与 k 折交叉验证进行比较时，可以从 n 样本中构建 n 模型，而不是 k 模型，其中 $n > k$ 。 此外，每个在 $n - 1$ 个样本而不是在 $(k-1)n/k$ 上进行训练。在两种方式中，假设 k 不是太大，并且 $k < n$ ， LOO 比 k 折交叉验证计算开销更加昂贵。

就精度而言， LOO 经常导致较高的方差作为测试误差的估计器。直观地说，因为 n 个样本中的 $n - 1$ 被用来构建每个模型，折叠构建的模型实际上是相同的，并且是从整个训练集建立的模型。

但是，如果学习曲线对于所讨论的训练大小是陡峭的，那么 5- 或 10- 折交叉验证可以泛化误差增高。

作为一般规则，大多数作者和经验证据表明， 5- 或者 10- 交叉验证应该优于 LOO 。

参考资料：

- http://www.faqs.org/faqs/ai-faq/neural-nets/part3/section-12.html;
- T. Hastie, R. Tibshirani, J. Friedman, The Elements of Statistical Learning, Springer 2009
- L. Breiman, P. Spector Submodel selection and evaluation in regression: The X-random case, International Statistical Review 1992;
- R. Kohavi, A Study of Cross-Validation and Bootstrap for Accuracy Estimation and Model Selection, Intl. Jnt.

> Conf. AI
> - R. Bharat Rao, G. Fung, R. Rosales, On the Dangers of Cross-Validation. An Experimental Evaluation, SIAM 2008;
> - G. James, D. Witten, T. Hastie, R Tibshirani, An Introduction to Statistical Learning, Springer 2013.

3.1.2.1.4. 留 P 交叉验证 (LPO)

`LeavePOut` 与 `LeaveOneOut` 非常相似，因为它通过从整个集合中删除 p 个样本来创建所有可能的 训练/测试 集。对于 n 个样本，这产生了 $\binom{n}{p}$ 个 训练-测试 对。与 `LeaveOneOut` 和 `KFold` 不同，当 $p > 1$ 时，测试集会重叠。

在有 4 个样例的数据集上使用 Leave-2-Out 的例子：

```
>>> from sklearn.model_selection import LeavePOut

>>> X = np.ones(4)
>>> lpo = LeavePOut(p=2)
>>> for train, test in lpo.split(X):
...     print("%s  %s" % (train, test))
[2 3] [0 1]
[1 3] [0 2]
[1 2] [0 3]
[0 3] [1 2]
[0 2] [1 3]
[0 1] [2 3]
```

3.1.2.1.5. 随机排列交叉验证 a.k.a. Shuffle & Split

`ShuffleSplit`

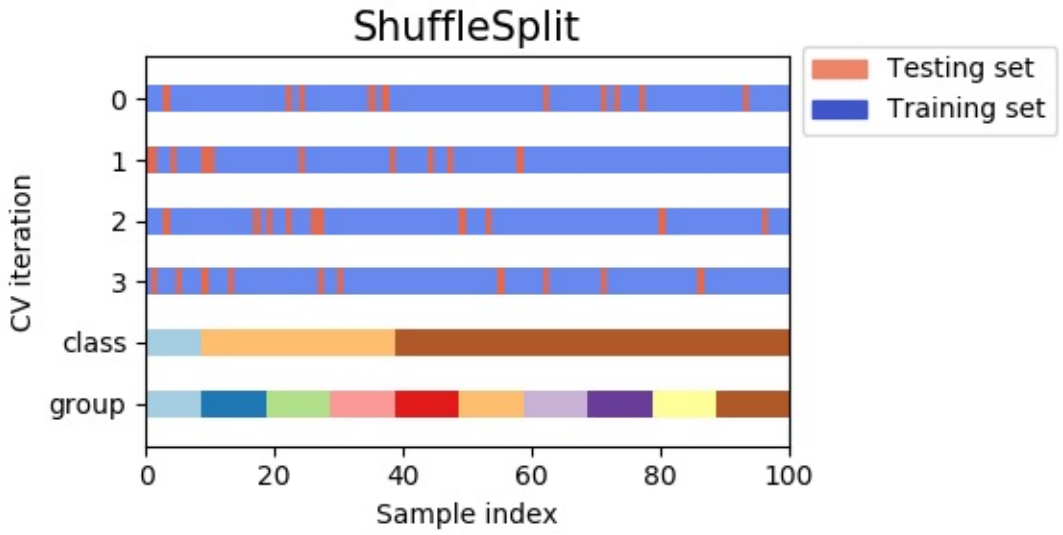
`ShuffleSplit` 迭代器 将会生成一个用户给定数量的独立的训练/测试数据划分。样例首先被打散然后划分为一对训练测试集合。

可以通过设定明确的 `random_state` ，使得伪随机生成器的结果可以重复。

这是一个使用的小例子：

```
>>> from sklearn.model_selection import ShuffleSplit
>>> X = np.arange(5)
>>> ss = ShuffleSplit(n_splits=3, test_size=0.25,
...     random_state=0)
>>> for train_index, test_index in ss.split(X):
...     print("%s  %s" % (train_index, test_index))
...
[1 3 4] [2 0]
[1 4 3] [0 2]
[4 0 2] [1 3]
```

下面是交叉验证行为的可视化。注意ShuffleSplit不受分类的影响。



`ShuffleSplit` 可以替代 `KFold` 交叉验证，因为其提供了细致的训练 / 测试划分的 数量和样例所占的比例等的控制。

3.1.2.2. 基于类标签、具有分层的交叉验证迭代器

一些分类问题在目标类别的分布上可能表现出很大的不平衡性：例如，可能会出现比正样本多数倍的负样本。在这种情况下，建议采用如 `StratifiedKFold` 和 `StratifiedShuffleSplit` 中实现的分层抽样方法，确保相对的类别频率在每个训练和验证 折叠 中大致保留。

3.1.2.2.1. 分层 k 折

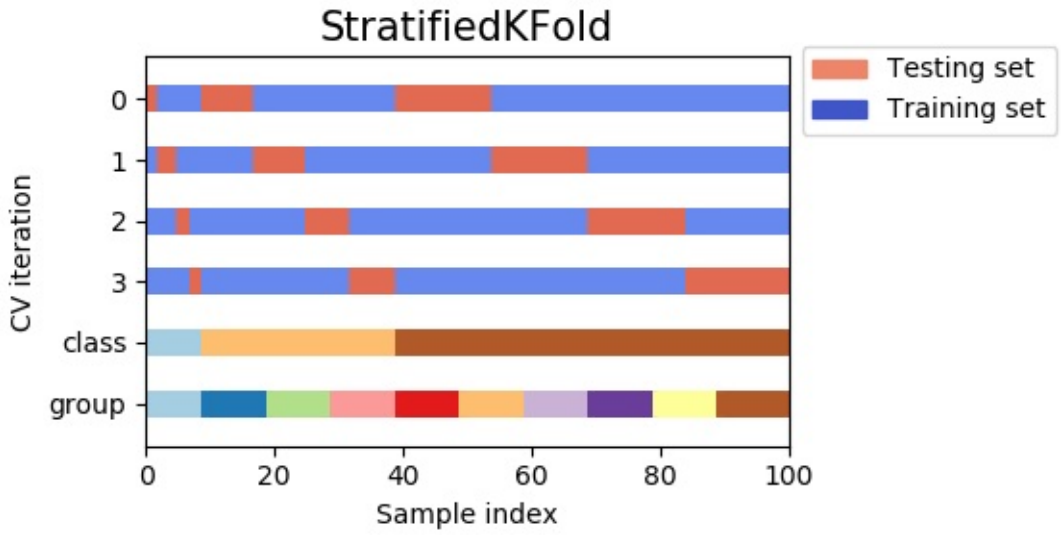
`StratifiedKFold` 是 *k-fold* 的变种，会返回 *stratified*（分层） 的折叠：每个小集合中， 各个类别的样例比例大致和完整数据集中相同。

在有 10 个样例的，有两个略不均衡类别的数据集上进行分层 3-fold 交叉验证的例子：

```
>>> from sklearn.model_selection import StratifiedKFold

>>> X = np.ones(10)
>>> y = [0, 0, 0, 0, 1, 1, 1, 1, 1, 1]
>>> skf = StratifiedKFold(n_splits=3)
>>> for train, test in skf.split(X, y):
...     print("%s  %s" % (train, test))
[2 3 6 7 8 9] [0 1 4 5]
[0 1 3 4 5 8 9] [2 6 7]
[0 1 2 4 5 6 7] [3 8 9]
```

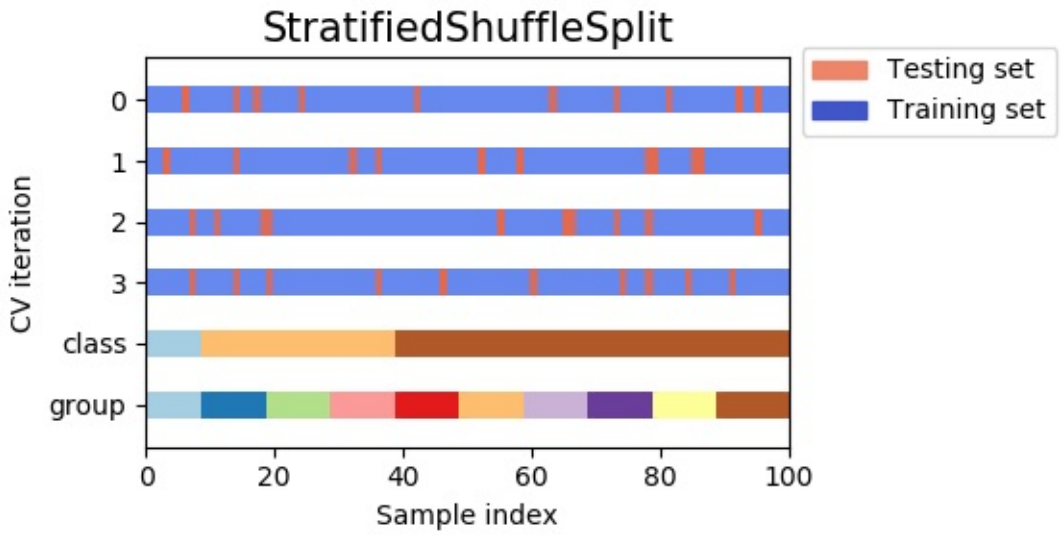
下面是该交叉验证方法的可视化。



`RepeatedStratifiedKFold` 可用于在每次重复中用不同的随机化重复分层 K-Fold n 次。

3.1.2.2.2. 分层随机 Split

下面是该交叉验证方法的可视化。



`StratifiedShuffleSplit` 是 *ShuffleSplit* 的一个变种，会返回直接的划分，比如： 创建一个划分，但是划分中每个类的比例和完整数据集中的相同。

3.1.2.3. 用于分组数据的交叉验证迭代器

如果潜在的生成过程产生依赖样本的 groups ，那么 i.i.d. 假设将会被打破。

这样的数据分组是特定于域的。一个例子是从多个患者收集医学数据，从每个患者身上采集多个样本。而这样的数据很可能取决于个人群体。 在我们的例子中，每个样本的患者 ID 将是其 group identifier （组标识符）。

在这种情况下，我们想知道在一组特定的 groups 上训练的模型是否能很好地适用于看不见的 group 。为了衡量这一点，我们需要确保验证对象中的所有样本来自配对训练折叠中完全没有表示的组。

下面的交叉验证分离器可以用来做到这一点。 样本的 grouping identifier （分组标识符） 通过 `groups` 参数指定。

3.1.2.3.1. 组 k-fold

`GroupKFold` 是 k-fold 的变体，它确保同一个 group 在测试和训练集中都不被表示。 例如，如果数据是从不同的 subjects 获得的，每个 subject 有多个样本，并且如果模型足够灵活以高度人物指定的特征中学习，则可能无法推广到新的 subject 。 `GroupKFold` 可以检测到这种过拟合的情况。 Imagine you have three subjects, each with an associated number from 1 to 3:

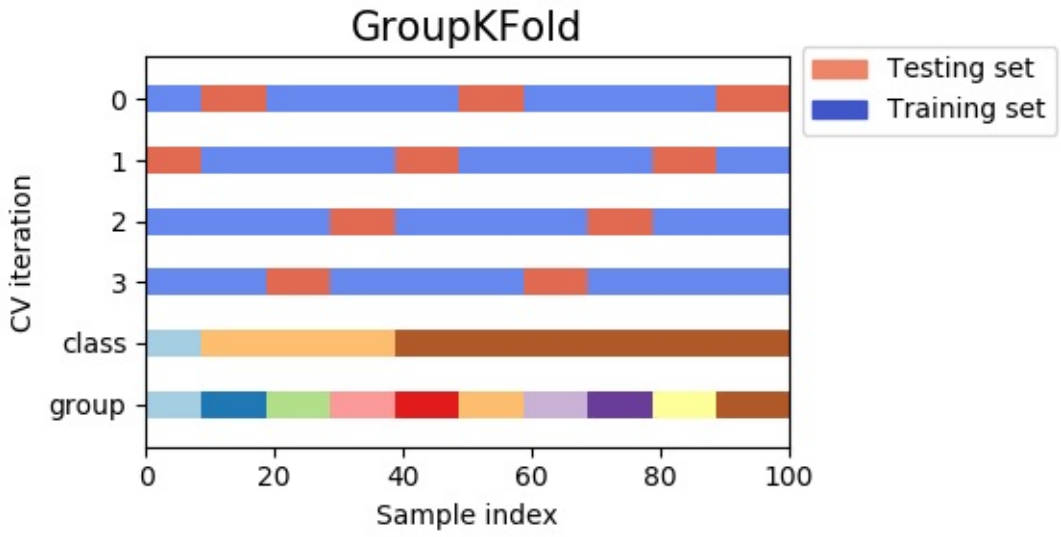
```
>>> from sklearn.model_selection import GroupKFold

>>> X = [0.1, 0.2, 2.2, 2.4, 2.3, 4.55, 5.8, 8.8, 9, 10]
>>> y = ["a", "b", "b", "b", "c", "c", "c", "d", "d", "d"]
>>> groups = [1, 1, 1, 2, 2, 2, 3, 3, 3, 3]

>>> gkf = GroupKFold(n_splits=3)
>>> for train, test in gkf.split(X, y, groups=groups):
...     print("%s  %s" % (train, test))
[0 1 2 3 4 5] [6 7 8 9]
[0 1 2 6 7 8 9] [3 4 5]
[3 4 5 6 7 8 9] [0 1 2]
```

每个 subject 都处于不同的测试阶段，同一个科目从来没有在测试和训练过程中。请注意，由于数据不平衡，折叠的大小并不完全相同。

下面是该交叉验证方法的可视化。



3.1.2.3.2. 留一组交叉验证

`LeaveOneGroupOut` 是一个交叉验证方案，它根据第三方提供的 array of integer groups （整数组的数组）来提供样本。这个组信息可以用来编码任意域特定的预定义交叉验证折叠。

每个训练集都是由除特定组别以外的所有样本构成的。

例如，在多个实验的情况下，`LeaveOneGroupOut` 可以用来根据不同的实验创建一个交叉验证：我们使用除去一个实验的所有实验的样本创建一个训练集：

```
>>> from sklearn.model_selection import LeaveOneGroupOut

>>> X = [1, 5, 10, 50, 60, 70, 80]
>>> y = [0, 1, 1, 2, 2, 2, 2]
>>> groups = [1, 1, 2, 2, 3, 3, 3]
>>> logo = LeaveOneGroupOut()
>>> for train, test in logo.split(X, y, groups=groups):
...     print("%s  %s" % (train, test))
[2 3 4 5 6] [0 1]
[0 1 4 5 6] [2 3]
[0 1 2 3] [4 5 6]
```

另一个常见的应用是使用时间信息：例如，组可以是收集样本的年份，从而允许与基于时间的分割进行交叉验证。

3.1.2.3.3. 留 P 组交叉验证

`LeavePGroupsOut` 类似于 `LeaveOneGroupOut` ，但为每个训练/测试集删除与 P 组有关的样本。

Leave-2-Group Out 的示例：

```
>>> from sklearn.model_selection import LeavePGroupsOut

>>> X = np.arange(6)
>>> y = [1, 1, 1, 2, 2, 2]
>>> groups = [1, 1, 2, 2, 3, 3]
>>> lpgo = LeavePGroupsOut(n_groups=2)
>>> for train, test in lpgo.split(X, y, groups=groups):
...     print("%s  %s" % (train, test))
[4 5] [0 1 2 3]
[2 3] [0 1 4 5]
[0 1] [2 3 4 5]
```

3.1.2.3.4. Group Shuffle Split

`GroupShuffleSplit` 迭代器是 `ShuffleSplit` 和 `LeavePGroupsOut` 的组合，它生成一个随机划分分区的序列，其中为每个分组提供了一个组子集。

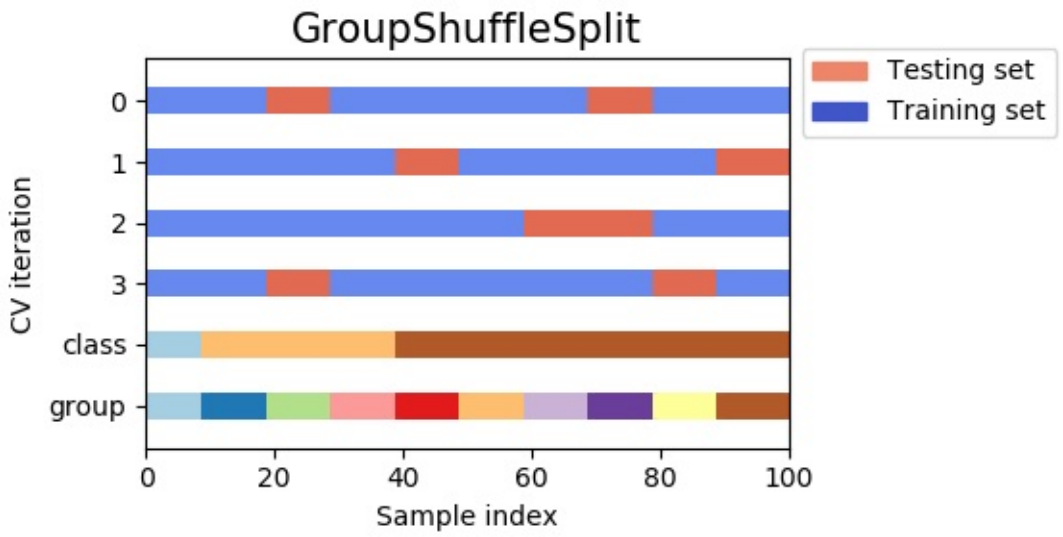
这是使用的示例：

```
>>> from sklearn.model_selection import GroupShuffleSplit

>>> X = [0.1, 0.2, 2.2, 2.4, 2.3, 4.55, 5.8, 0.001]
>>> y = ["a", "b", "b", "b", "c", "c", "c", "a"]
>>> groups = [1, 1, 2, 2, 3, 3, 4, 4]
>>> gss = GroupShuffleSplit(n_splits=4, test_size=0.5, random_state=0)
>>> for train, test in gss.split(X, y, groups=groups):
```

```
...     print("%s  %s" % (train, test))
...
[0 1 2 3] [4 5 6 7]
[2 3 6 7] [0 1 4 5]
[2 3 4 5] [0 1 6 7]
[4 5 6 7] [0 1 2 3]
```

下面是该交叉验证方法的可视化。



当需要 `LeavePGroupsOut` 的操作时，这个类的信息是很有必要的，但是 组 的数目足够大，以至于用 P 组生成所有可能的分区将会花费很大的代价。在这种情况下， `GroupShuffleSplit` 通过 `LeavePGroupsOut` 提供了一个随机（可重复）的训练 / 测试划分采样。

3.1.2.4. 预定义的折叠 / 验证集

对一些数据集，一个预定义的，将数据划分为训练和验证集合或者划分为几个交叉验证集合的划分已经存在。 可以使用 `PredefinedSplit` 来使用这些集合来搜索超参数。

比如，当使用验证集合时，设置所有验证集合中的样例的 `test_fold` 为 `0`，而将其他样例设置为 `-1` 。

3.1.2.5. 交叉验证在时间序列数据中应用

时间序列数据的特点是时间 (*autocorrelation(*自相关性*)*) 附近的观测之间的相关性。 然而，传统的交叉验证技术，例如 `KFold` 和 `ShuffleSplit` 假设样本是独立的且分布相同的，并且在时间序列数据上会导致训练和测试实例之间不合理的相关性（产生广义误差的估计较差）。 因此，对 “future(未来)” 观测的时间序列数据模型的评估至少与用于训练模型的观测模型非常重要。为了达到这个目的，一个解决方案是由 `TimeSeriesSplit` 提供的。

3.1.2.5.1. 时间序列分割

`TimeSeriesSplit` 是 *k-fold* 的一个变体，它首先返回 k 折作为训练数据集，并且 $(k+1)$ 折作为测试数据集。 请注意，与标准的交叉验证方法不同，连续的训练集是超越前者的超集。 另外，它将所有的剩余数据添加到第一个训练分区，它总是用来训练模型。

这个类可以用来交叉验证以固定时间间隔观察到的时间序列数据样本。

对具有 6 个样本的数据集进行 3-split 时间序列交叉验证的示例：

```
>>> from sklearn.model_selection import TimeSeriesSplit

>>> X = np.array([[1, 2], [3, 4], [1, 2], [3, 4], [1, 2], [3, 4]])
>>> y = np.array([1, 2, 3, 4, 5, 6])
>>> tscv = TimeSeriesSplit(n_splits=3)
>>> print(tscv)
TimeSeriesSplit(max_train_size=None, n_splits=3)
>>> for train, test in tscv.split(X):
...     print("%s  %s" % (train, test))
[0 1 2] [3]
[0 1 2 3] [4]
[0 1 2 3 4] [5]
```

3.1.3. A note on shuffling

如果数据的顺序不是任意的（比如说，相同标签的样例连续出现），为了获得有意义的交叉验证结果，首先对其进行打散是很有必要的。然而，当样例不是独立同分布时打散则是不可行的。例如：样例是相关的文章，以他们发表的时间 进行排序，这时候如果对数据进行打散，将会导致模型过拟合，得到一个过高的验证分数：因为验证样例更加相似（在时间上更接近） 于训练数据。

一些交叉验证迭代器， 比如 `KFold` ，有一个内建的在划分数据前进行数据索引打散的选项。注意：

- 这种方式仅需要很少的内存就可以打散数据。
- 默认不会进行打散，包括设置 `cv=some_integer` （直接）k 折叠交叉验证的 `cross_val_score` ， 表格搜索等。注意 `train_test_split` 会返回一个随机的划分。
- 参数 `random_state` 默认设置为 `None` ，这意为着每次进行 `KFold(…, shuffle=True)` 时，打散都是不同的。 然而， `GridSearchCV` 通过调用 `fit` 方法验证时，将会使用相同的打散来训练每一组参数。
- 为了保证结果的可重复性（在相同的平台上），应该给 `random_state` 设定一个固定的值。

3.1.4. 交叉验证和模型选择

交叉验证迭代器可以通过网格搜索得到最优的模型超参数，从而直接用于模型的选择。 这是另一部分 调整估计器的超参数 的主要内容。

我们一直在努力

apachecn/sklearn-doc-zh



3.2. 调整估计器的超参数

校验者：@想和太阳肩并肩翻译者：@\S^R^Y/

超参数，即不直接在估计器内学习的参数。在 scikit-learn 包中，它们作为估计器类中构造函数的参数进行传递。典型的例子有：用于支持向量分类器的 `C` 、 `kernel` 和 `gamma` ，用于Lasso的 `alpha` 等。

搜索超参数空间以便获得最好 交叉验证 分数的方法是可能的而且是值得提倡的。

通过这种方式，构造估计器时被提供的任何参数或许都能被优化。具体来说，要获取到给定估计器的所有参数的名称和当前值，使用：

```
1. estimator.get_params()
```

搜索包括：

- 估计器(回归器或分类器，例如 `sklearn.svm.SVC()`)
- 参数空间
- 搜寻或采样候选的方法
- 交叉验证方案
- 计分函数

有些模型支持专业化的、高效的参数搜索策略， 描述如下 。在 scikit-learn 包中提供了两种采样搜索候选的通用方法:对于给定的值， GridSearchCV 考虑了所有参数组合；而 RandomizedSearchCV 可以从具有指定分布的参数空间中抽取给定数量的候选。介绍完这些工具后，我们将详细介绍适用于这两种方法的 最佳实践 。

注意，通常这些参数的一小部分会对模型的预测或计算性能有很大的影响，而其他参数可以保留为其默认值。 建议阅读估计器类的相关文档，以更好地了解其预期行为，可能的话还可以阅读下引用的文献。

3.2.1. 网格追踪法–穷尽的网格搜索

`GridSearchCV` 提供的网格搜索从通过 `param_grid` 参数确定的网格参数值中全面生成候选。例如，下面的 `param_grid` ：

```
1. param_grid = [
2.   {'C': [1, 10, 100, 1000], 'kernel': ['linear']},
3.   {'C': [1, 10, 100, 1000], 'gamma': [0.001, 0.0001], 'kernel': ['rbf']},
4.  ]
```

探索两个网格的详细解释： 一个具有线性内核并且C在[1,10,100,1000]中取值； 另一个具有RBF内核，C值的交叉乘积范围在[1,10，100,1000]，gamma在[0.001，0.0001]中取值。

`GridSearchCV` 实例实现了常用估计器 API：当在数据集上“拟合”时，参数值的所有可能的组合都会被评估，从而计算出最佳的组合。

> 示例：

- 有关在数字数据集上的网格搜索计算示例，请参阅 基于交叉验证的网格搜索参数估计。
- 有关来自文本文档特征提取器（n-gram计数向量化器和TF-IDF变换器）的网格搜索耦合参数与分类器（这里是使用具有弹性网格的SGD训练的线性SVM 或L2惩罚）使用 *pipeline.Pipeline* 示例,请参阅 用于文本特征提取和评估的示例管道。
- 有关iris数据集的交叉验证循环中的网格搜索示例， 请参阅 嵌套与非嵌套交叉验证。
- 有关用于同时评估多个指标的GridSearchCV示例，请参阅 cross_val_score 与 GridSearchCV 多指标评价的实证研究。

3.2.2. 随机参数优化

尽管使用参数设置的网格法是目前最广泛使用的参数优化方法， 其他搜索方法也具有更有利的性能。 RandomizedSearchCV 实现了对参数的随机搜索， 其中每个设置都是从可能的参数值的分布中进行取样。 这对于穷举搜索有两个主要优势：

- 可以选择独立于参数个数和可能值的预算
- 添加不影响性能的参数不会降低效率

指定如何取样的参数是使用字典完成的， 非常类似于为 `GridSearchCV` 指定参数。 此外， 通过 `n_iter` 参数指定计算预算， 即取样候选项数或取样迭代次数。 对于每个参数， 可以指定在可能值上的分布或离散选择的列表 （均匀取样）：

```
{'C': scipy.stats.expon(scale=100), 'gamma': scipy.stats.expon(scale=.1),
  'kernel': ['rbf'], 'class_weight':['balanced', None]}
```

本示例使用 `scipy.stats` 模块， 它包含许多用于采样参数的有用分布， 如 `expon` ， `gamma` ， `uniform` 或者 `randint` 。 原则上， 任何函数都可以通过提供一个 `rvs` （随机变量样本）方法来采样一个值。 对 `rvs` 函数的调用应在连续调用中提供来自可能参数值的独立随机样本。

警告

scipy 0.16之前的版本不允许指定随机状态，而是使用 `np.random` 来产生随机状态，或是使用 `np.random.set` 来指定状态. 然而在scikit-learn 0.18以后，如果scipy版本大于0.16，则sklearn.model_selection可以通过用户指定来获得随机状态

对于连续参数 （如上面提到的 `C` ），指定连续分布以充分利用随机化是很重要的。这样，有助于 `n_iter` 总是趋向于更精细的搜索。

示例：

- 随机搜索和网格搜索的使用和效率的比较： 有关随机搜索和网格搜索超参数估计的对比

引用：

- Bergstra, J. and Bengio, Y., Random search for hyper-parameter optimization, The Journal of Machine Learning Research (2012)

3.2.3. 参数搜索技巧

3.2.3.1. 指定目标度量

默认情况下， 参数搜索使用估计器的评分函数来评估（衡量）参数设置。 比如 sklearn.metrics.accuracy_score 用于分类和 sklearn.metrics.r2_score 用于回归。 对于一些应用，

其他评分函数将会更加适合（例如在不平衡的分类，精度评分往往是信息不足的）。 一个可选的评分功能可以通过评分参数指定给 `GridSearchCV` ， `RandomizedSearchCV` 和许多下文将要描述的、专业化的交叉验证工具。 有关详细信息，请参阅 评分参数:定义模型评估规则。

3.2.3.2. 为评估指定多个指标

`GridSearchCV` 和 `RandomizedSearchCV` 允许为评分参数指定多个指标。

多指标评分可以被指定为一个预先定义分数名称字符串列表或者是一个得分手名字到得分手的函数或预先定义的记分员名字的映射字典。 有关详细信息，请参阅 多指标评估。

在指定多个指标时，必须将 `refit` 参数设置为要在其中找到 `best_params` ，并用于在整个数据集上构建 `best_estimator` 的度量标准（字符串）。 如果搜索不应该 `refit`，则设置 `refit=False` 。在使用多个度量值时，如果将 `refit` 保留为默认值，不会导致结果错误。

有关示例用法，请参见 `cross_val_score` 与 `GridSearchCV` 多指标评价的实证研究。

3.2.3.3. 复合估计和参数空间

管道：链式评估器 描述了如何使用这些工具搜索参数空间构建链式评估器。

3.2.3.4. 模型选择：开发和评估

通过评估各种参数设置，可以将模型选择视为使用标记数据训练网格参数的一种方法。

在评估结果模型时，重要的是在网格搜索过程中未看到的 `held-out` 样本数据上执行以下操作：建议将数据拆分为开发集（**`development set`**，供 `GridSearchCV` 实例使用）和评估集（**`evaluation set`**）来计算性能指标。

这可以通过使用效用函数 `train_test_split` 来完成。

3.2.3.5. 并行机制

`GridSearchCV` 和 `RandomizedSearchCV` 可以独立地评估每个参数设置。如果您的OS支持，通过使用关键字 `n_jobs=-1` 可以使计算并行运行。 有关详细信息，请参见函数签名。

3.2.3.6. 对故障的鲁棒性

某些参数设置可能导致无法 `fit` 数据的一个或多个折叠。 默认情况下，这将导致整个搜索失败，即使某些参数设置可以完全计算。 设置 `error_score=0` （或 `=np.NaN` ）将使程序对此类故障具有鲁棒性，发出警告并将该折叠的分数设置为0（或 `NaN` ），但可以完成搜索。

3.2.4. 暴力参数搜索的替代方案

3.2.4.1. 模型特定交叉验证

某些模型可以与参数的单个值的估计值一样有效地适应某一参数范围内的数据。 此功能可用于执行更有效的交叉验

证， 用于此参数的模型选择。

该策略最常用的参数是编码正则化矩阵强度的参数。在这种情况下， 我们称之为， 计算估计器的正则化路径 (**regularization path**)。

以下是这些模型的列表：

模型	描述
linear_model.ElasticNetCV([l1_ratio, eps, …])	Elastic Net model with iterative fitting along a regularization path
linear_model.LarsCV([fit_intercept, …])	Cross-validated Least Angle Regression model
linear_model.LassoCV([eps, n_alphas, …])	Lasso linear model with iterative fitting along a regularization path
linear_model.LassoLarsCV([fit_intercept, …])	Cross-validated Lasso, using the LARS algorithm
linear_model.LogisticRegressionCV([Cs, …])	Logistic Regression CV (aka logit, MaxEnt) classifier.
linear_model.MultiTaskElasticNetCV([…])	Multi-task L1/L2 ElasticNet with built-in cross-validation.
linear_model.MultiTaskLassoCV([eps, …])	Multi-task L1/L2 Lasso with built-in cross-validation.
linear_model.OrthogonalMatchingPursuitCV([…])	Cross-validated Orthogonal Matching Pursuit model (OMP)
linear_model.RidgeCV([alphas, …])	Ridge regression with built-in cross-validation.
linear_model.RidgeClassifierCV([alphas, …])	Ridge classifier with built-in cross-validation.

3.2.4.2. 信息标准

一些模型通过计算一个正则化路径（代替使用交叉验证得出数个参数），可以给出正则化参数最优估计的信息理论闭包公式。

以下是从 Akaike 信息标准（AIC）或贝叶斯信息标准（可用于自动选择模型）中受益的模型列表：

模型	描述
linear_model.LassoLarsIC([criterion, …])	Lasso model fit with Lars using BIC or AIC for model selection

3.2.4.3. 出袋估计

当使用基于装袋的集合方法时，即使用具有替换的采样产生新的训练集，部分训练集保持不用。 对于集合中的每个分类器，训练集的不同部分被忽略。

这个省略的部分可以用来估计泛化误差，而不必依靠单独的验证集。 此估计是”免费的”，因为不需要额外的数据，可以用于模型选择。

目前该方法已经实现的类以下几个：

方法	描述
`ensemble.RandomForestClassifier` ([…])	A random forest classifier.
`ensemble.RandomForestRegressor` ([…])	A random forest regressor.
`ensemble.ExtraTreesClassifier` ([…])	An extra-trees classifier.
`ensemble.ExtraTreesRegressor` ([n_estimators, …])	An extra-trees regressor.
`ensemble.GradientBoostingClassifier` ([loss, …])	Gradient Boosting for classification.
`ensemble.GradientBoostingRegressor` ([loss, …])	Gradient Boosting for regression.

我们一直在努力

apachecn/sklearn-doc-zh



3.3. 模型评估： 量化预测的质量

校验者： @飓风@小瑶@FAME@v@Loopy翻译者： @小瑶@片刻@那伊抹微笑

有 3 种不同的 API 用于评估模型预测的质量：

- **Estimator score method**（估计器得分的方法）： Estimators（估计器）有一个 `score（得分）` 方法，为其解决的问题提供了默认的 evaluation criterion （评估标准）。 在这个页面上没有相关讨论，但是在每个 estimator （估计器）的文档中会有相关的讨论。
- **Scoring parameter**（评分参数）： Model-evaluation tools （模型评估工具）使用 cross-validation （如 `model_selection.cross_val_score` 和 `model_selection.GridSearchCV`）依靠 internal *scoring* strategy （内部 *scoring*（得分） 策略）。这在 scoring 参数： 定义模型评估规则 部分讨论。
- **Metric functions**（指标函数）： `metrics` 模块实现了针对特定目的评估预测误差的函数。这些指标在以下部分部分详细介绍 分类指标, 多标签排名指标, 回归指标 和 聚类指标 。

最后， 虚拟估计 用于获取随机预测的这些指标的基准值。

> See also:对于 “pairwise（成对）” metrics（指标）, *samples*（样本） 之间而不是 estimators （估计量）或者 predictions（预测值）, 请参阅 成对的矩阵, 类别和核函数 部分。dr

3.3.1. scoring 参数： 定义模型评估规则

Model selection （模型选择）和 evaluation （评估）使用工具，例如 `model_selection.GridSearchCV` 和 `model_selection.cross_val_score` ，采用 `scoring` 参数来控制它们对 estimators evaluated （评估的估计量）应用的指标。

3.3.1.1. 常见场景： 预定义值

对于最常见的用例， 您可以使用 `scoring` 参数指定一个 scorer object （记分对象）； 下表显示了所有可能的值。 所有 scorer objects （记分对象）遵循惯例 **higher return values are better than lower return values**（较高的返回值优于较低的返回值） 。因此，测量模型和数据之间距离的 metrics （度量），如 `metrics.mean_squared_error` 可用作返回 metric （指数）的 negated value （否定值）的 neg_mean_squared_error 。

Scoring（得分）	Function（函数）	Comment（注解）
Classification（分类）		
‘accuracy’	`metrics.accuracy_score`	
‘average_precision’	`metrics.average_precision_score`	
‘f1’	`metrics.f1_score`	for binary targets（用于二进制目标）
‘f1_micro’	`metrics.f1_score`	micro-averaged（微平均）

‘f1_macro’	`metrics.f1_score`	macro-averaged（宏平均）
‘f1_weighted’	`metrics.f1_score`	weighted average（加权平均）
‘f1_samples’	`metrics.f1_score`	by multilabel sample（通过 multilabel 样本）
‘neg_log_loss’	`metrics.log_loss`	requires `predict_proba` support（需要 `predict_proba` 支持）
‘precision’ etc.	`metrics.precision_score`	suffixes apply as with ‘f1’（后缀适用于 ‘f1’）
‘recall’ etc.	`metrics.recall_score`	suffixes apply as with ‘f1’（后缀适用于 ‘f1’）
‘roc_auc’	`metrics.roc_auc_score`	
Clustering（聚类）		
‘adjusted_mutual_info_score’	`metrics.adjusted_mutual_info_score`	
‘adjusted_rand_score’	`metrics.adjusted_rand_score`	
‘completeness_score’	`metrics.completeness_score`	
‘fowlkes_mallows_score’	`metrics.fowlkes_mallows_score`	
‘homogeneity_score’	`metrics.homogeneity_score`	
‘mutual_info_score’	`metrics.mutual_info_score`	
‘normalized_mutual_info_score’	`metrics.normalized_mutual_info_score`	
‘v_measure_score’	`metrics.v_measure_score`	
Regression（回归）		
‘explained_variance’	`metrics.explained_variance_score`	
‘neg_mean_absolute_error’	`metrics.mean_absolute_error`	
‘neg_mean_squared_error’	`metrics.mean_squared_error`	
‘neg_mean_squared_log_error’	`metrics.mean_squared_log_error`	
‘neg_median_absolute_error’	`metrics.median_absolute_error`	
‘r2’	`metrics.r2_score`	

使用案例：

```
>>> from sklearn import svm, datasets
>>> from sklearn.model_selection import cross_val_score
>>> iris = datasets.load_iris()
>>> X, y = iris.data, iris.target
>>> clf = svm.SVC(probability=True, random_state=0)
>>> cross_val_score(clf, X, y, scoring='neg_log_loss')
array([-0.07..., -0.16..., -0.06...])
```

```
>>> model = svm.SVC()
>>> cross_val_score(model, X, y, scoring='wrong_choice')
Traceback (most recent call last):
ValueError: 'wrong_choice' is not a valid scoring value. Valid options are ['accuracy',
'adjusted_mutual_info_score', 'adjusted_rand_score', 'average_precision', 'completeness_score',
'explained_variance', 'f1', 'f1_macro', 'f1_micro', 'f1_samples', 'f1_weighted', 'fowlkes_mallows_score',
'homogeneity_score', 'mutual_info_score', 'neg_log_loss', 'neg_mean_absolute_error', 'neg_mean_squared_error',
'neg_mean_squared_log_error', 'neg_median_absolute_error', 'normalized_mutual_info_score', 'precision',
'precision_macro', 'precision_micro', 'precision_samples', 'precision_weighted', 'r2', 'recall',
'recall_macro', 'recall_micro', 'recall_samples', 'recall_weighted', 'roc_auc', 'v_measure_score']
```

> 注意
>
> ValueError exception 列出的值对应于以下部分描述的 functions measuring prediction accuracy （测量预测精度的函数）。 这些函数的 scorer objects （记分对象）存储在 dictionary `sklearn.metrics.SCORERS` 中。

3.3.1.2. 根据 metric 函数定义您的评分策略

模块 `sklearn.metrics` 还公开了一组 measuring a prediction error （测量预测误差）的简单函数，给出了基础真实的数据和预测：

- 函数以 `_score` 结尾返回一个值来最大化，越高越好。
- 函数 `_error` 或 `_loss` 结尾返回一个值来 minimize （最小化），越低越好。当使用 `make_scorer` 转换成 scorer object （记分对象）时，将 `greater_is_better` 参数设置为 False（默认为 True; 请参阅下面的参数说明）。

可用于各种机器学习任务的 Metrics （指标）在下面详细介绍。

许多 metrics （指标）没有被用作 `scoring（得分）` 值的名称，有时是因为它们需要额外的参数，例如 `fbeta_score` 。在这种情况下，您需要生成一个适当的 scoring object （评分对象）。生成 callable object for scoring （可评估对象进行评分）的最简单方法是使用 `make_scorer` 。该函数将 metrics （指数）转换为可用于可调用的 model evaluation （模型评估）。

一个典型的用例是从库中包含一个非默认值参数的 existing metric function （现有指数函数），例如 `fbeta_score` 函数的 `beta` 参数：

```
>>> from sklearn.metrics import fbeta_score, make_scorer
>>> ftwo_scorer = make_scorer(fbeta_score, beta=2)
>>> from sklearn.model_selection import GridSearchCV
>>> from sklearn.svm import LinearSVC
>>> grid = GridSearchCV(LinearSVC(), param_grid={'C': [1, 10]}, scoring=ftwo_scorer)
```

第二个用例是使用 `make_scorer` 从简单的 python 函数构建一个完全 custom scorer object （自定义的记分对象），可以使用几个参数 ：

- 你要使用的 python 函数（在下面的例子中是 `my_custom_loss_func` ）
- python 函数是否返回一个分数 （ `greater_is_better=True` ，默认值）或者一个 loss （损失） （ `greater_is_better=False` ）。 如果是一个 loss （损失），scorer object （记分对象）的 python 函数的输出被 negated （否定），符合 cross validation convention （交叉验证约定），scorers 为更好的模型返回更高的值。

- 仅用于 classification metrics （分类指数）： 您提供的 python 函数是否需要连续的 continuous decision certainties （判断确定性）（ `needs_threshold=True` ）。默认值为 False 。
- 任何其他参数，如 `beta` 或者 `labels` 在 函数 `f1_score` 。

以下是建立 custom scorers （自定义记分对象）的示例，并使用 `greater_is_better` 参数：

```
>>> import numpy as np
>>> def my_custom_loss_func(y_true, y_pred):
...     diff = np.abs(y_true - y_pred).max()
...     return np.log1p(diff)
...
>>> # score will negate the return value of my_custom_loss_func,
>>> # which will be np.log(2), 0.693, given the values for X
>>> # and y defined below.
>>> score = make_scorer(my_custom_loss_func, greater_is_better=False)
>>> X = [[1], [1]]
>>> y = [0, 1]
>>> from sklearn.dummy import DummyClassifier
>>> clf = DummyClassifier(strategy='most_frequent', random_state=0)
>>> clf = clf.fit(X, y)
>>> my_custom_loss_func(clf.predict(X), y)
0.69...
>>> score(clf, X, y)
-0.69...
```

3.3.1.3. 实现自己的记分对象

您可以通过从头开始构建自己的 scoring object （记分对象），而不使用 `make_scorer` factory 来生成更加灵活的 model scorers （模型记分对象）。 对于被叫做 scorer 来说，它需要符合以下两个规则所指定的协议：

- 可以使用参数 `(estimator, X, y)` 来调用它，其中 `estimator` 是要被评估的模型， `X` 是验证数据， `y` 是 `X` （在有监督情况下）或 `None` （在无监督情况下）已经被标注的真实数据目标。
- 它返回一个浮点数，用于对 `X` 进行量化 `estimator` 的预测质量，参考 `y` 。 再次，按照惯例，更高的数字更好，所以如果你的 scorer 返回 loss ，那么这个值应该被 negated 。

注意:在n_jobs > 1的函数中使用自定义评分器

虽然在调用函数的旁边定义自定义计分函数应该使用默认的joblib后端(loky)，但是从另一个模块导入它将是一种更健壮的方法，并且独立于joblib后端。

例如，在下面的例子中，要使用大于1的 `n_jobs` ， `custom_scoring_function` 函数保存在用户创建的模块中(custom_scorer_module.py)并导入：

```
>> from custom_scorer_module import custom_scoring_function>> cross_val_score(model, … X_train, … y_train, … scoring=make_scorer(custom_scoring_function, greater_is_better=False), … cv=5, … n_jobs=-1)
```

3.3.1.4. 使用多个指数评估

Scikit-learn 还允许在 `GridSearchCV` , `RandomizedSearchCV` 和 `cross_validate` 中评估 multiple metric （多个指数）。

为 `scoring` 参数指定多个评分指标有两种方法：

- 作为 string metrics 的迭代：

```
>>> scoring = ['accuracy', 'precision']
```

- 作为 dict ，将 scorer 名称映射到 scoring 函数：

```
>>> from sklearn.metrics import accuracy_score
>>> from sklearn.metrics import make_scorer
>>> scoring = {'accuracy': make_scorer(accuracy_score),
...            'prec': 'precision'}
```

请注意， dict 值可以是 scorer functions （记分函数）或者 predefined metric strings （预定义 metric 字符串）之一。

目前，只有那些返回 single score （单一分数）的 scorer functions （记分函数）才能在 dict 内传递。不允许返回多个值的 Scorer functions （Scorer 函数），并且需要一个 wrapper 才能返回 single metric（单个指标）：

```
>>> from sklearn.model_selection import cross_validate
>>> from sklearn.metrics import confusion_matrix
>>> # A sample toy binary classification dataset
>>> X, y = datasets.make_classification(n_classes=2, random_state=0)
>>> svm = LinearSVC(random_state=0)
>>> def tn(y_true, y_pred): return confusion_matrix(y_true, y_pred)[0, 0]
>>> def fp(y_true, y_pred): return confusion_matrix(y_true, y_pred)[0, 1]
>>> def fn(y_true, y_pred): return confusion_matrix(y_true, y_pred)[1, 0]
>>> def tp(y_true, y_pred): return confusion_matrix(y_true, y_pred)[1, 1]
>>> scoring = {'tp': make_scorer(tp), 'tn': make_scorer(tn),
...            'fp': make_scorer(fp), 'fn': make_scorer(fn)}
>>> cv_results = cross_validate(svm.fit(X, y), X, y,
...                             scoring=scoring, cv=5)
>>> # Getting the test set true positive scores
>>> print(cv_results['test_tp'])
[10  9  8  7  8]
>>> # Getting the test set false negative scores
>>> print(cv_results['test_fn'])
[0 1 2 3 2]
```

3.3.2. 分类指标

`sklearn.metrics` 模块实现了几个 loss, score, 和 utility 函数来衡量 classification （分类）性能。 某些 metrics （指标）可能需要 positive class （正类），confidence values（置信度值）或 binary decisions values （二进制决策值）的概率估计。 大多数的实现允许每个样本通过 `sample_weight`

参数为 overall score （总分）提供 weighted contribution （加权贡献）。

其中一些仅限于二分类案例：

调用	功能
precision_recall_curve(y_true, probas_pred)	Compute precision-recall pairs for different probability thresholds
roc_curve (y_true, y_score[, pos_label, …])	Compute Receiver operating characteristic (ROC)

其他也可以在多分类案例中运行：

调用	功能
cohen_kappa_score(y1, y2[, labels, weights, …])	Cohen’s kappa: a statistic that measures inter-annotator agreement.
confusion_matrix (y_true, y_pred[, labels, …])	Compute confusion matrix to evaluate the accuracy of a classification
hinge_loss (y_true, pred_decision[, labels, …])	Average hinge loss (non-regularized)
matthews_corrcoef (y_true, y_pred[, …])	Compute the Matthews correlation coefficient (MCC)

有些还可以在 multilabel case （多重案例）中工作：

调用	功能
accuracy_score(y_true, y_pred[, normalize, …])	Accuracy classification score.
classification_report (y_true, y_pred[, …])	Build a text report showing the main classification metrics
f1_score (y_true, y_pred[, labels, …])	Compute the F1 score, also known as balanced F-score or F-measure
fbeta_score (y_true, y_pred, beta[, labels, …])	Compute the F-beta score
hamming_loss (y_true, y_pred[, labels, …])	Compute the average Hamming loss.
jaccard_similarity_score (y_true, y_pred[, …])	Jaccard similarity coefficient score
log_loss (y_true, y_pred[, eps, normalize, …])	Log loss, aka logistic loss or cross-entropy loss.
precision_recall_fscore_support (y_true, y_pred)	Compute precision, recall, F-measure and support for each class
precision_score(y_true, y_pred[, labels, …])	Compute the precision
recall_score (y_true, y_pred[, labels, …])	Compute the recall
zero_one_loss(y_true, y_pred[, normalize, …])	Zero-one classification loss.

一些通常用于 ranking:

调用	功能
`dcg_score`(y_true, y_score[, k])	Discounted cumulative gain (DCG) at rank K.
`ndcg_score` (y_true, y_score[, k])	Normalized discounted cumulative gain (NDCG) at rank K.

有些工作与 binary 和 multilabel （但不是多类）的问题:

调用	功能
`average_precision_score`(y_true, y_score[, …])	Compute average precision (AP) from prediction scores
`roc_auc_score` (y_true, y_score[, average, …])	Compute Area Under the Curve (AUC) from prediction scores

在以下小节中，我们将介绍每个这些功能，前面是一些关于通用 API 和 metric 定义的注释。

3.3.2.1. 从二分到多分类和 multilabel

一些 metrics 基本上是为 binary classification tasks （二分类任务）定义的（例如 `f1_score` ，`roc_auc_score` ）。在这些情况下，默认情况下仅评估 positive label （正标签），假设默认情况下，positive label （正类）标记为 `1` （尽管可以通过 `pos_label` 参数进行配置）。

将 binary metric （二分指标）扩展为 multiclass （多类）或 multilabel （多标签）问题时，数据将被视为二分问题的集合，每个类都有一个。 然后可以使用多种方法在整个类中 average binary metric calculations （平均二分指标计算），每种类在某些情况下可能会有用。 如果可用，您应该使用 `average` 参数来选择它们。

- `"macro（宏）"` 简单地计算 binary metrics （二分指标）的平均值，赋予每个类别相同的权重。在不常见的类别重要的问题上，macro-averaging （宏观平均）可能是突出表现的一种手段。另一方面，所有类别同样重要的假设通常是不真实的，因此 macro-averaging （宏观平均）将过度强调不频繁类的典型的低性能。
- `"weighted（加权）"` 通过计算其在真实数据样本中的存在来对每个类的 score 进行加权的 binary metrics （二分指标）的平均值来计算类不平衡。
- `"micro（微）"` 给每个 sample-class pair （样本类对）对 overall metric （总体指数）（sample-class 权重的结果除外） 等同的贡献。除了对每个类别的 metric 进行求和之外，这个总和构成每个类别度量的 dividends （除数）和 divisors （除数）计算一个整体商。 在 multilabel settings （多标签设置）中，Micro-averaging 可能是优先选择的，包括要忽略 majority class （多数类）的 multiclass classification （多类分类）。
- `"samples（样本）"` 仅适用于 multilabel problems （多标签问题）。它 does not calculate a per-class measure （不计算每个类别的 measure），而是计算 evaluation data （评估数据）中的每个样本的 true and predicted classes （真实和预测类别）的 metric （指标），并返回 (`sample_weight`-weighted) 加权平均。
- 选择 `average=None` 将返回一个 array 与每个类的 score 。

虽然将 multiclass data （多类数据）提供给 metric ，如 binary targets （二分类目标），作为 array of class labels （类标签的数组），multilabel data （多标签数据）被指定为 indicator

matrix（指示符矩阵），其中 cell `[i, j]` 具有值 1，如果样本 `i` 具有标号 `j` ，否则为值 0 。

3.3.2.2. 精确度得分

`accuracy_score` 函数计算 accuracy，正确预测的分数（默认）或计数 (normalize=False)。

在 multilabel classification （多标签分类）中，函数返回 subset accuracy（子集精度）。如果样本的 entire set of predicted labels （整套预测标签）与真正的标签组合匹配，则子集精度为 1.0; 否则为 0.0 。

如果 $\hat{y}_i$ 是第 i 个样本的预测值，y_i 是相应的真实值，则 n_{samples} 上的正确预测的分数被定义为

$$\text{accuracy}(y, \hat{y}) = \frac{1}{n_{\text{samples}}} \sum_{i=0}^{n_{\text{samples}}-1} 1(\hat{y}_i = y_i)$$

其中 $1(x)$ 是 indicator function（指示函数）.

```
>>> import numpy as np
>>> from sklearn.metrics import accuracy_score
>>> y_pred = [0, 2, 1, 3]
>>> y_true = [0, 1, 2, 3]
>>> accuracy_score(y_true, y_pred)
0.5
>>> accuracy_score(y_true, y_pred, normalize=False)
2
```

In the multilabel case with binary label indicators（在具有二分标签指示符的多标签情况下）：

```
>>> accuracy_score(np.array([[0, 1], [1, 1]]), np.ones((2, 2)))
0.5
```

> 示例：
>
> - 参阅 Test with permutations the significance of a classification score 例如使用数据集排列的 accuracy score （精度分数）。

3.3.2.3. Balanced accuracy score

此 `balanced_accuracy_score` 函数计算 balanced accuracy，它可以避免在不平衡数据集上作出夸大的性能估计。它是每个类的召回分数的宏观平均，或者，等价地，原始准确度，其中每个样本根据其真实类的样本占比率加权。因此，对均衡数据集，该函数的得分与准确率得分是相等的。

在二分类情况下， `balanced accuracy` 等价于sensitivity(true positive rate)和 specificity(真负率:true negative rate)的算术平均值，或者ROC曲线下具有二元预测值的面积，而不是分数。

如果分类器在两个类上都表现的一样好，该函数就会退化为传统的准确率(即正确预测数量除以总的预测数量).

作为对比，如果传统的准确率(conventional accuracy)比较好，仅仅是因为分类器利用了一个不均衡测试集，此时 `balanced_accuracy` 将会近似地掉到$\frac{1}{n_classes}$

得分的范围是0到1，或者当设置参数 `adjusted` =True 时，得分被缩放到从$\frac{1}{1 - \text{n_classes}}$到1,包括边界的，随机条件下得分为0.

如果y_i是第i个样本的真值，并且w_i是对应的样本权重，然后我们调整样本权重到：

$$\hat{w}_i = \frac{w_i}{\sum_j 1(y_j = y_i) w_j}$$

其中 `1(x)` 是Indicator_function,给定样本中，如果$\hat{y}_i$是第i个样本的真值,则 `balanced_accuracy` 表示为：

$$\text{balanced-accuracy}(y, \hat{y}, w) = \frac{1}{\sum \hat{w}_i} \sum_i 1(\hat{y}_i = y_i) \hat{w}_i$$

当设置参数 `adjusted` =True时， `balanced_accuracy` 反映

$$\text{balanced-accuracy}(y, \mathbf{0}, w) = \frac{1}{n_classes}$$的相对增加,在二分类情况下，这也被称为 Youden's J statistic或者informedness

> 注意:这里的multiclass定义似乎是二进制分类中使用的度量的最合理的扩展，尽管在文献中没有达成一定的共识:
>
> - 我们定义:来自[Mosley2013], [Kelleher2015]和[Guyon2015], [Guyon2015]调整后的版本,以确保随机预测得分为0,完美的预测得分为1 .
> - 如[Mosley2013]所述:计算每个类的精度与召回量之间的最小值。然后将这些值平均到类的总数上，以获得平衡的精度。
> - 如[Urbanowicz2015]所述:计算每个类的敏感性和特异性的平均值，然后在总类数上取平均值。
>
> 参考资料:
>
> - [Guyon2015] (1, 2) I. Guyon, K. Bennett, G. Cawley, H.J. Escalante, S. Escalera, T.K. Ho, N. Macià, B. Ray, M. Saeed, A.R. Statnikov, E. Viegas, Design of the 2015 ChaLearn AutoML Challenge, IJCNN 2015.
> - [Mosley2013] (1, 2) L. Mosley, A balanced approach to the multi-class imbalance problem, IJCV 2010.
> - [Kelleher2015] John. D. Kelleher, Brian Mac Namee, Aoife D'Arcy, Fundamentals of Machine Learning for Predictive Data Analytics: Algorithms, Worked Examples, and Case Studies, 2015.
> - [Urbanowicz2015] Urbanowicz R.J., Moore, J.H. ExSTraCS 2.0: description and evaluation of a scalable learning classifier system, Evol. Intel. (2015) 8: 89.

3.3.2.4. Cohen's kappa

函数 `cohen_kappa_score` 计算 Cohen's kappa statistic（统计）。 这个 measure （措施）旨在比较不同人工标注者的标签，而不是 classifier （分类器）与 ground truth （真实数据）。

kappa score （参阅 docstring ）是 -1 和 1 之间的数字。 .8 以上的 scores 通常被认为是很好的 agreement （协议）; 0 或者 更低表示没有 agreement （实际上是 random labels （随机标签））。

Kappa scores 可以计算 binary or multiclass （二分或者多分类）问题，但不能用于 multilabel problems （多标签问题）（除了手动计算 per-label score （每个标签分数）），而不是两个以上的 annotators （注释器）。

```
>>> from sklearn.metrics import cohen_kappa_score
>>> y_true = [2, 0, 2, 2, 0, 1]
>>> y_pred = [0, 0, 2, 2, 0, 2]
>>> cohen_kappa_score(y_true, y_pred)
0.4285714285714286
```

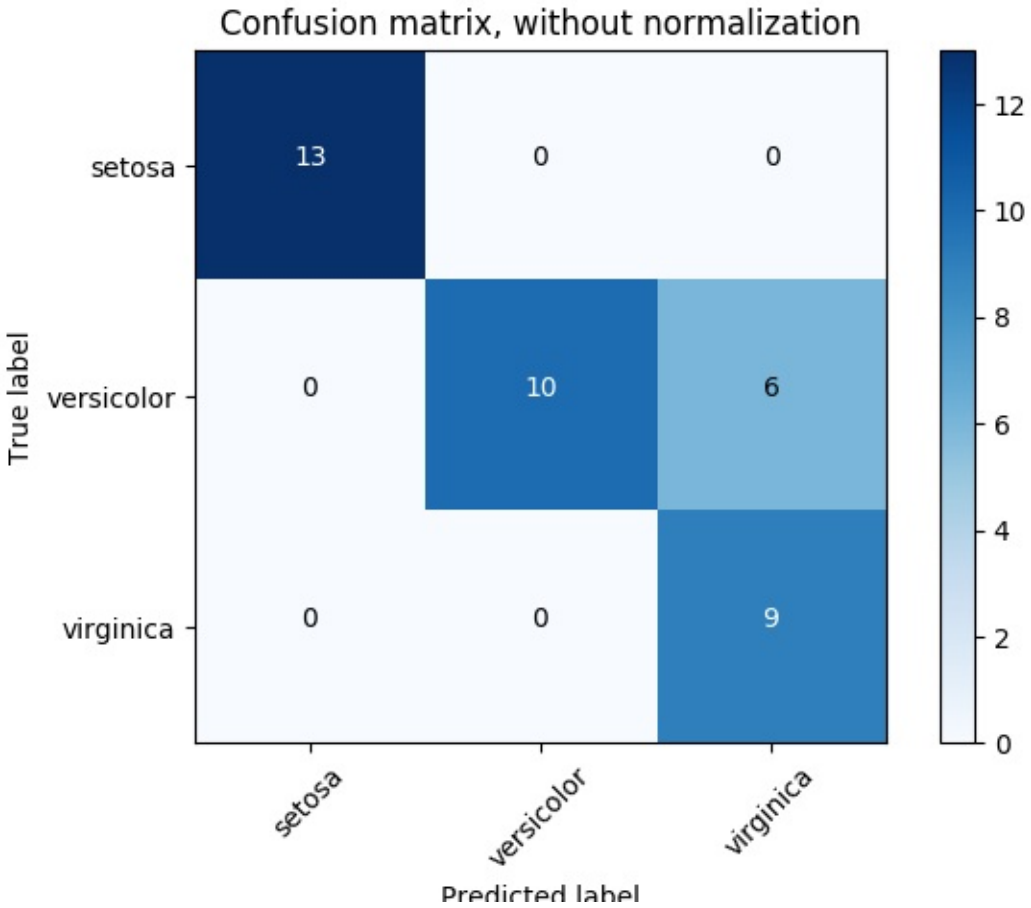
3.3.2.5. 混淆矩阵

`confusion_matrix` 函数通过计算 confusion matrix（混淆矩阵） 来 evaluates classification accuracy （评估分类的准确性）。

根据定义，confusion matrix （混淆矩阵）中的 entry（条目） i, j，是实际上在 group i 中的 observations （观察数），但预测在 group j 中。这里是一个示例：

```
>>> from sklearn.metrics import confusion_matrix
>>> y_true = [2, 0, 2, 2, 0, 1]
>>> y_pred = [0, 0, 2, 2, 0, 2]
>>> confusion_matrix(y_true, y_pred)
array([[2, 0, 0],
 [0, 0, 1],
 [1, 0, 2]])
```

这是一个这样的 confusion matrix （混淆矩阵）的可视化表示 （这个数字来自于 Confusion matrix）：



对于 binary problems （二分类问题），我们可以得到 true negatives（真 negatives），false positives（假 positives），false negatives（假 negatives） 和 true positives（真 positives） 的数量如下：

```
>>> y_true = [0, 0, 0, 1, 1, 1, 1, 1]
>>> y_pred = [0, 1, 0, 1, 0, 1, 0, 1]
>>> tn, fp, fn, tp = confusion_matrix(y_true, y_pred).ravel()
>>> tn, fp, fn, tp
(2, 1, 2, 3)
```

示例：

- 参阅 Confusion matrix 例如使用 confusion matrix （混淆矩阵）来评估 classifier （分类器）的输出质量。
- 参阅 Recognizing hand-written digits 例如使用 confusion matrix （混淆矩阵）来分类手写数字。
- 参阅 Classification of text documents using sparse features 例如使用 confusion matrix （混淆矩阵）对文本文档进行分类。

3.3.2.6. 分类报告

`classification_report` 函数构建一个显示 main classification metrics （主分类指标）的文本报告。这是一个小例子，其中包含自定义的 `target_names` 和 inferred labels （推断标签）：

```
>>> from sklearn.metrics import classification_report
>>> y_true = [0, 1, 2, 2, 0]
>>> y_pred = [0, 0, 2, 1, 0]
```

```
>>> target_names = ['class 0', 'class 1', 'class 2']
>>> print(classification_report(y_true, y_pred, target_names=target_names))
              precision    recall  f1-score   support

     class 0       0.67      1.00      0.80         2
     class 1       0.00      0.00      0.00         1
     class 2       1.00      0.50      0.67         2

    accuracy                           0.60         5
   macro avg       0.56      0.50      0.49         5
weighted avg       0.67      0.60      0.59         5
```

示例:

- 参阅 Recognizing hand-written digits 作为手写数字的分类报告的使用示例。
- 参阅 Classification of text documents using sparse features 作为文本文档的分类报告使用的示例。
- 参阅 Parameter estimation using grid search with cross-validation 例如使用 grid search with nested cross-validation （嵌套交叉验证进行网格搜索）的分类报告。

3.3.2.7. 汉明损失

`hamming_loss` 计算两组样本之间的 average Hamming loss （平均汉明损失）或者 Hamming distance（汉明距离） 。

如果 $\hat{y}_j$ 是给定样本的第 j 个标签的预测值，则 y_j 是相应的真实值，而 n_{labels} 是 classes or labels （类或者标签）的数量，则两个样本之间的 Hamming loss （汉明损失） $L_{Hamming}$ 定义为:

$$L_{Hamming}(y, \hat{y}) = \frac{1}{n_{\text{labels}}} \sum_{j=0}^{n_{\text{labels}}-1} 1(\hat{y}_j \neq y_j)$$

其中 $1(x)$ 是 indicator function（指标函数）.

```
>>> from sklearn.metrics import hamming_loss
>>> y_pred = [1, 2, 3, 4]
>>> y_true = [2, 2, 3, 4]
>>> hamming_loss(y_true, y_pred)
0.25
```

在具有 binary label indicators （二分标签指示符）的 multilabel （多标签）情况下:

```
>>> hamming_loss(np.array([[0, 1], [1, 1]]), np.zeros((2, 2)))
0.75
```

注意

在 multiclass classification （多类分类）中， Hamming loss （汉明损失）对应于 `y_true` 和 `y_pred` 之间的 Hamming

> distance（汉明距离），它类似于 零一损失 函数。然而， zero-one loss penalizes （0-1损失惩罚）不严格匹配真实集合的预测集，Hamming loss （汉明损失）惩罚 individual labels （独立标签）。因此，Hamming loss（汉明损失）高于 zero-one loss（0-1 损失），总是在 0 和 1 之间，包括 0 和 1;预测真正的标签的正确的 subset or superset （子集或超集）将给出 0 和 1 之间的 Hamming loss（汉明损失）。

3.3.2.8. 精准，召回和 F-measures

直观地来理解，precision 是 the ability of the classifier not to label as positive a sample that is negative （classifier （分类器）的标签不能被标记为正的样本为负的能力），并且 recall 是 classifier （分类器）查找所有 positive samples （正样本）的能力。

F-measure （F_β 和 F_1 measures）可以解释为 precision （精度）和 recall （召回）的 weighted harmonic mean （加权调和平均值）。F_β measure 值达到其最佳值 1 ，其最差分数为 0 。与 $\beta = 1$，F_β 和 F_1 是等价的， recall （召回）和 precision （精度）同样重要。

`precision_recall_curve` 通过改变 decision threshold （决策阈值）从 ground truth label （被标记的真实数据标签） 和 score given by the classifier （分类器给出的分数）计算 precision-recall curve （精确召回曲线）。

`average_precision_score` 函数根据 prediction scores （预测分数）计算出 average precision (AP) （平均精度）。该分数对应于 precision-recall curve （精确召回曲线）下的面积。该值在 0 和 1 之间，并且越高越好。通过 random predictions （随机预测）， AP 是 fraction of positive samples （正样本的分数）。

$$\text{AP} = \sum_n (R_n - R_{n-1}) P_n$$

其中P_n和R_n是第n个阈值处的precision和recall。对于随机预测，AP是正样本的比例。

参考文献 [Manning2008] 和 [Everingham2010] 提出了AP的两种可替代变体对precision-recall曲线进行内插。 当前，函数 `average_precision_score` 还没有实现任何具备内插的变体版本。 参考文献 [Davis2006] 和 [Flach2015] 描述了为什么precision-recall曲线上的点的线性内插提供了一个过于乐观(overly-optimistic)的分类器性能度量。 在函数 `auc` 中使用梯形规则(trapezoidal rule)计算曲线下面积的时候，这个线性内插(linear interpolation)会被使用。

几个函数可以让您实现 analyze the precision （分析精度），recall（召回） 和 F-measures 得分：

调用	功能
`average_precision_score` (y_true, y_score[, …])	Compute average precision (AP) from prediction scores
`f1_score` (y_true, y_pred[, labels, …])	Compute the F1 score, also known as balanced F-score or F-measure
`fbeta_score` (y_true, y_pred, beta[, labels, …])	Compute the F-beta score
`precision_recall_curve` (y_true, probas_pred)	Compute precision-recall pairs for different probability thresholds

precision_recall_fscore_support(y_true, y_pred)	Compute precision, recall, F-measure and support for each class
precision_score(y_true, y_pred[, labels, …])	Compute the precision
recall_score(y_true, y_pred[, labels, …])	Compute the recall

请注意， precision_recall_curve 函数仅限于 binary case （二分情况）。 average_precision_score 函数只适用于 binary classification and multilabel indicator format （二分类和多标签指示器格式）。

示例：

- 参阅 Classification of text documents using sparse features 例如 f1_score 用于分类文本文档的用法。
- 参阅 Parameter estimation using grid search with cross-validation 例如 precision_score 和 recall_score 用于 using grid search with nested cross-validation （使用嵌套交叉验证的网格搜索）来估计参数。
- 参阅 Precision-Recall 例如 precision_recall_curve 用于 evaluate classifier output quality（评估分类器输出质量）。

参考资料

- [Manning2008] C.D. Manning, P. Raghavan, H. Schütze, Introduction to Information Retrieval, 2008.
- [Everingham2010] M. Everingham, L. Van Gool, C.K.I. Williams, J. Winn, A. Zisserman, The Pascal Visual Object Classes (VOC) Challenge, IJCV 2010.
- [Davis2006] J. Davis, M. Goadrich, The Relationship Between Precision-Recall and ROC Curves, ICML 2006.
- [Flach2015] P.A. Flach, M. Kull, Precision-Recall-Gain Curves: PR Analysis Done Right, NIPS 2015.

3.3.2.8.1. 二分类

在二分类任务中，术语 ''positive（正）'' 和 ''negative（负）'' 是指 classifier's prediction （分类器的预测），术语 ''true（真）'' 和 ''false（假）'' 是指该预测是否对应于 external judgment （外部判断）（有时被称为 ''observation（观测值）''）。给出这些定义，我们可以指定下表：

	Actual class (observation)	Actual class (observation)
Predicted class (expectation)	tp (true positive) Correct result	fp (false positive) Unexpected result
Predicted class (expectation)	fn (false negative) Missing result	tn (true negative) Correct absence of result

在这种情况下，我们可以定义 precision（精度）， recall（召回） 和 F-measure 的概念：

$$\text{precision} = \frac{tp}{tp + fp},$$

$$\text{recall} = \frac{tp}{tp + fn},$$

$$F_\beta = (1 + \beta^2) \frac{\text{precision} \times \text{recall}}{\beta^2 \text{precision} + \text{recall}}.$$

以下是 binary classification （二分类）中的一些小例子：

```
>>> from sklearn import metrics
>>> y_pred = [0, 1, 0, 0]
>>> y_true = [0, 1, 0, 1]
>>> metrics.precision_score(y_true, y_pred)
1.0
>>> metrics.recall_score(y_true, y_pred)
0.5
>>> metrics.f1_score(y_true, y_pred)
0.66...
>>> metrics.fbeta_score(y_true, y_pred, beta=0.5)
0.83...
>>> metrics.fbeta_score(y_true, y_pred, beta=1)
0.66...
>>> metrics.fbeta_score(y_true, y_pred, beta=2)
0.55...
>>> metrics.precision_recall_fscore_support(y_true, y_pred, beta=0.5)
(array([0.66..., 1.        ]), array([1. , 0.5]), array([0.71..., 0.83...]), array([2, 2]))


>>> import numpy as np
>>> from sklearn.metrics import precision_recall_curve
>>> from sklearn.metrics import average_precision_score
>>> y_true = np.array([0, 0, 1, 1])
>>> y_scores = np.array([0.1, 0.4, 0.35, 0.8])
>>> precision, recall, threshold = precision_recall_curve(y_true, y_scores)
>>> precision
array([0.66..., 0.5       , 1.        , 1.        ])
>>> recall
array([1. , 0.5, 0.5, 0. ])
>>> threshold
array([0.35, 0.4 , 0.8 ])
>>> average_precision_score(y_true, y_scores)
0.83...
```

3.3.2.8.2. 多类和多标签分类

在 multiclass and multilabel classification task（多类和多标签分类任务）中，precision（精度），recall（召回），and F-measures 的概念可以独立地应用于每个标签。 有以下几种方法 combine results across labels （将结果跨越标签组合），由 `average` 参数指定为 `average_precision_score`（仅用于 multilabel），`f1_score`，`fbeta_score`，`precision_recall_fscore_support`，`precision_score` 和 `recall_score` 函数，如上 above 所述。请注意，对于在包含所有标签的多类设置中进行 “micro”-averaging （”微”平均），将产生相等的 precision（精度），recall（召回）和 F，而 “weighted（加权）” averaging（平均）可能会产生 precision（精度）和 recall（召回）之间的 F-score 。

为了使这一点更加明确，请考虑以下 notation （符号）：

- $y_{predicted}$（预测）$(sample, label)$ 对
- $\hat{y}_{true}$（真）$(sample, label)$ 对

- L labels 集合
- S samples 集合
- $y_s y$ 的子集与样本 s, 即 $y_s := \{(s', l) \in y | s' = s\}$
- $y_l y$ 的子集与 label l
- 类似的, $\hat{y}_s$ 和 $\hat{y}_l$ 是 $\hat{y}$ 的子集
- $P(A, B) := \frac{|A \cap B|}{|A|}$
- $R(A, B) := \frac{|A \cap B|}{|B|}$ (Conventions （公约）在处理 $B = \emptyset$ 有所不同; 这个实现使用 $R(A, B) := 0$, 与 P 类似.)
- $F_\beta(A, B) := (1 + \beta^2) \frac{P(A,B) \times R(A,B)}{\beta^2 P(A,B) + R(A,B)}$

然后将 metrics （指标）定义为:

average	Precision	Recall	Fbeta
"micro"	$P(y, \hat{y})$	$R(y, \hat{y})$	![F\beta(y, \hat{y})](/projects/sklearn-0.21.3/93f9d89ff86616780639491c938dd9ca.jpeg)
"samples"	$\frac{1}{\lvert S \rvert} \sum_{s \in S} P(y_s, \hat{y}_s)$	$\frac{1}{\lvert S \rvert} \sum_{s \in S} R(y_s, \hat{y}_s)$	$\frac{1}{\lvert S \rvert} \sum_{s \in S} F_\beta(y_s, \hat{y}_s)$
"macro"	$\frac{1}{\lvert L \rvert} \sum_{l \in L} P(y_l, \hat{y}_l)$	$\frac{1}{\lvert L \rvert} \sum_{l \in L} R(y_l, \hat{y}_l)$	$\frac{1}{\lvert L \rvert} \sum_{l \in L} F_\beta(y_l, \hat{y}_l)$
"weighted"	$\frac{1}{\sum_{l \in L} \lvert \hat{y}_l \rvert} \sum_{l \in L} \lvert \hat{y}_l \rvert P(y_l, \hat{y}_l)$	$\frac{1}{\sum_{l \in L} \lvert \hat{y}_l \rvert} \sum_{l \in L} \lvert \hat{y}_l \rvert R(y_l, \hat{y}_l)$	$\frac{1}{\sum_{l \in L} \lvert \hat{y}_l \rvert} \sum_{l \in L} \lvert \hat{y}_l \rvert F_\beta(y_l, \hat{y}_l)$
None	$\langle P(y_l, \hat{y}_l) \lvert l \in L \rangle$	$\langle R(y_l, \hat{y}_l) \lvert l \in L \rangle$	$\langle F_\beta(y_l, \hat{y}_l) \lvert l \in L \rangle$

```
>>> from sklearn import metrics
>>> y_true = [0, 1, 2, 0, 1, 2]
>>> y_pred = [0, 2, 1, 0, 0, 1]
>>> metrics.precision_score(y_true, y_pred, average='macro')
0.22...
>>> metrics.recall_score(y_true, y_pred, average='micro')
...
0.33...
>>> metrics.f1_score(y_true, y_pred, average='weighted')
0.26...
>>> metrics.fbeta_score(y_true, y_pred, average='macro', beta=0.5)
0.23...
>>> metrics.precision_recall_fscore_support(y_true, y_pred, beta=0.5, average=None)
...
```

```
    (array([0.66..., 0.        , 0.        ]), array([1., 0., 0.]), array([0.71..., 0.        , 0.        ]),
15. array([2, 2, 2]...))
```

For multiclass classification with a “negative class”, it is possible to exclude some labels:

```
1. >>> metrics.recall_score(y_true, y_pred, labels=[1, 2], average='micro')
2. ... # excluding 0, no labels were correctly recalled
3. 0.0
```

Similarly, labels not present in the data sample may be accounted for in macro-averaging.

```
1. >>> metrics.precision_score(y_true, y_pred, labels=[0, 1, 2, 3], average='macro')
2. ...
3. 0.166...
```

3.3.2.9. Jaccard 相似系数 score

`jaccard_similarity_score` 函数计算 pairs of label sets （标签组对）之间的 Jaccard similarity coefficients 也称作 Jaccard index 的平均值（默认）或总和。

将第 i 个样本的 Jaccard similarity coefficient 与 被标注过的真实数据的标签集 y_i 和 predicted label set （预测标签集）：$\hat{y}_i$ 定义为

$$J(y_i, \hat{y}_i) = \frac{|y_i \cap \hat{y}_i|}{|y_i \cup \hat{y}_i|}.$$

`jaccard_score` 就像 `precision_recall_fscore_support` 中的设定方法，本身应用于二分类，并通过使用从二分类扩展到多标记和多类（见从二分到多分类和multilabel）。

二分类时：

```
1. >>> import numpy as np
2. >>> from sklearn.metrics import jaccard_score
3. >>> y_true = np.array([[0, 1, 1],
4. ...                    [1, 1, 0]])
5. >>> y_pred = np.array([[1, 1, 1],
6. ...                    [1, 0, 0]])
7. >>> jaccard_score(y_true[0], y_pred[0])
8. 0.6666...
```

在具有二分类标签指示符的多标签案例中：

```
1. >>> jaccard_score(y_true, y_pred, average='samples')
2. 0.5833...
```

```
>>> jaccard_score(y_true, y_pred, average='macro')
0.6666...
>>> jaccard_score(y_true, y_pred, average=None)
array([0.5, 0.5, 1. ])
```

将多类问题二进制化并像对应的多标签问题一样处理：

```
>>> y_pred = [0, 2, 1, 2]
>>> y_true = [0, 1, 2, 2]
>>> jaccard_score(y_true, y_pred, average=None)
...
array([1. , 0. , 0.33...])
>>> jaccard_score(y_true, y_pred, average='macro')
0.44...
>>> jaccard_score(y_true, y_pred, average='micro')
0.33...
```

3.3.2.10. Hinge loss

`hinge_loss` 函数使用 hinge loss 计算模型和数据之间的 average distance （平均距离），这是一种只考虑 prediction errors （预测误差）的 one-sided metric （单向指标）。（Hinge loss 用于最大边界分类器，如支持向量机）

如果标签用 +1 和 -1 编码，则 y：是真实值，并且 w 是由 `decision_function` 输出的 predicted decisions （预测决策），则 hinge loss 定义为：

$$L_{\text{Hinge}}(y, w) = \max\left\{1 - wy, 0\right\} = \left|1 - wy\right|_+$$

如果有两个以上的标签， `hinge_loss` 由于 Crammer & Singer 而使用了 multiclass variant （多类型变体）。 Here 是描述它的论文。

如果 y_w 是真实标签的 predicted decision （预测决策），并且 y_t 是所有其他标签的预测决策的最大值，其中预测决策由 decision function （决策函数）输出，则 multiclass hinge loss 定义如下：

$$L_{\text{Hinge}}(y_w, y_t) = \max\left\{1 + y_t - y_w, 0\right\}$$

这里是一个小例子，演示了在 binary class （二类）问题中使用了具有 svm classifier （svm 的分类器）的 `hinge_loss` 函数：

```
>>> from sklearn import svm
>>> from sklearn.metrics import hinge_loss
>>> X = [[0], [1]]
>>> y = [-1, 1]
>>> est = svm.LinearSVC(random_state=0)
>>> est.fit(X, y)
LinearSVC(C=1.0, class_weight=None, dual=True, fit_intercept=True,
     intercept_scaling=1, loss='squared_hinge', max_iter=1000,
```

```
     multi_class='ovr', penalty='l2', random_state=0, tol=0.0001,
     verbose=0)
>>> pred_decision = est.decision_function([[-2], [3], [0.5]])
>>> pred_decision
array([-2.18...,  2.36...,  0.09...])
>>> hinge_loss([-1, 1, 1], pred_decision)
0.3...
```

这里是一个示例，演示了在 multiclass problem （多类问题）中使用了具有 svm 分类器的 `hinge_loss` 函数：

```
>>> X = np.array([[0], [1], [2], [3]])
>>> Y = np.array([0, 1, 2, 3])
>>> labels = np.array([0, 1, 2, 3])
>>> est = svm.LinearSVC()
>>> est.fit(X, Y)
LinearSVC(C=1.0, class_weight=None, dual=True, fit_intercept=True,
     intercept_scaling=1, loss='squared_hinge', max_iter=1000,
     multi_class='ovr', penalty='l2', random_state=None, tol=0.0001,
     verbose=0)
>>> pred_decision = est.decision_function([[-1], [2], [3]])
>>> y_true = [0, 2, 3]
>>> hinge_loss(y_true, pred_decision, labels)
0.56...
```

3.3.2.11. Log 损失

Log loss，又被称为 logistic regression loss（logistic 回归损失）或者 cross-entropy loss（交叉熵损失） 定义在 probability estimates （概率估计）。它通常用于 (multinomial) logistic regression （（多项式）logistic 回归）和 neural networks （神经网络）以及 expectation-maximization （期望最大化）的一些变体中，并且可用于评估分类器的 probability outputs （概率输出）（ `predict_proba` ）而不是其 discrete predictions （离散预测）。

对于具有真实标签 $y \in \{0, 1\}$ 的 binary classification （二分类）和 probability estimate （概率估计） $p = \Pr(y = 1)$，每个样本的 log loss 是给定的分类器的 negative log-likelihood 真正的标签：

$$L_{\log}(y, p) = -\log \Pr(y|p) = -(y \log(p) + (1 - y) \log(1 - p))$$

这扩展到 multiclass case （多类案例）如下。 让一组样本的真实标签被编码为 1-of-K binary indicator matrix Y，即 如果样本 i 具有取自一组 K 个标签的标签 k，则 $y_{i,k} = 1$。令 P 为 matrix of probability estimates （概率估计矩阵），$p_{i,k} = \Pr(t_{i,k} = 1)$。那么整套的 log loss 就是

$$L_{\log}(Y, P) = -\log \Pr(Y|P) = -\frac{1}{N} \sum_{i=0}^{N-1} \sum_{k=0}^{K-1} y_{i,k} \log p_{i,k}$$

为了看这这里如何 generalizes （推广）上面给出的 binary log loss （二分 log loss），请注意，在 binary case （二分情况下），$p_{i,0} = 1 - p_{i,1}$ 和 $y_{i,0} = 1 - y_{i,1}$，因此扩展 $y_{i,k} \in \{0, 1\}$ 的 inner sum （内部和），给出 binary log loss （二分 log loss）。

`log_loss` 函数计算出一个 a list of ground-truth labels （已标注的真实数据的标签的列表）和一个 probability matrix （概率矩阵） 的 log loss，由 estimator （估计器）的 `predict_proba` 方法返回。

```
>>> from sklearn.metrics import log_loss
>>> y_true = [0, 0, 1, 1]
>>> y_pred = [[.9, .1], [.8, .2], [.3, .7], [.01, .99]]
>>> log_loss(y_true, y_pred)
0.1738...
```

`y_pred` 中的第一个 `[.9, .1]` 表示第一个样本具有标签 0 的 90% 概率。log loss 是非负数。

3.3.2.12. 马修斯相关系数

`matthews_corrcoef` 函数用于计算 binary classes （二分类）的 Matthew's correlation coefficient

(MCC) 引用自 Wikipedia:

> “Matthews correlation coefficient（马修斯相关系数）用于机器学习，作为 binary (two-class) classifications （二分类）分类质量的度量。它考虑到 true and false positives and negatives （真和假的 positives 和 negatives），通常被认为是可以使用的 balanced measure（平衡措施），即使 classes are of very different sizes （类别大小不同）。MCC 本质上是 -1 和 +1 之间的相关系数值。系数 +1 表示完美预测，0 表示平均随机预测， -1 表示反向预测。statistic （统计量）也称为 phi coefficient （phi）系数。”

在 binary (two-class) （二分类）情况下，tp, tn, fp 和 fn 分别是 true positives, true negatives, false positives 和 false negatives 的数量，MCC 定义为

$$MCC = \frac{tp \times tn - fp \times fn}{\sqrt{(tp + fp)(tp + fn)(tn + fp)(tn + fn)}}.$$

在 multiclass case （多类的情况）下， Matthews correlation coefficient（马修斯相关系数） 可以根据 K classes （类）的 `confusion_matrix` C 定义 defined 。为了简化定义，考虑以下中间变量：

- $t_k = \sum_i^K C_{ik}$ 真正发生了 k 类的次数，
- $p_k = \sum_i^K C_{ki}$ 类被预测的次数，
- $c = \sum_k^K C_{kk}$ 正确预测的样本总数，
- $s = \sum_i^K \sum_j^K C_{ij}$ 样本总数.

然后 multiclass MCC 定义为：

$$MCC = \frac{c \times s - \sum_k^K p_k \times t_k}{\sqrt{(s^2 - \sum_k^K p_k^2) \times (s^2 - \sum_k^K t_k^2)}}$$

当有两个以上的标签时， MCC 的值将不再在 -1 和 +1 之间。相反，根据已经标注的真实数据的数量和分布情况，最小值将介于 -1 和 0 之间。最大值始终为 +1 。

这是一个小例子，说明了使用 `matthews_corrcoef` 函数：

```
>>> from sklearn.metrics import matthews_corrcoef
>>> y_true = [+1, +1, +1, -1]
>>> y_pred = [+1, -1, +1, +1]
>>> matthews_corrcoef(y_true, y_pred)
-0.33...
```

3.3.2.13. 多标记混淆矩阵

`multilabel_confusion_matrix` 函数计算分类（默认）或样本（ `samplewise` = True时）的多标记混淆矩阵，以评估分类的准确性。 `multilabel_confusion_matrix` 也将多类数据视为多标记数据，因为这是一种常用于评估二元分类指标

（如精度，召回等）的多类问题的转换。

在计算分类多标记混淆矩阵C时 ，类的真阴性计数i是$C_{i,0,0}$，假阴性是$C_{i,1,0}$，真阳性是$C_{i,1,1}$，虚警率是$C_{i,0,1}$

以下示例演示了使用多标签指标矩阵输入的 `multilabel_confusion_matrix` 函数 ：

```
>>> import numpy as np
>>> from sklearn.metrics import multilabel_confusion_matrix
>>> y_true = np.array([[1, 0, 1],
...                    [0, 1, 0]])
>>> y_pred = np.array([[1, 0, 0],
...                    [0, 1, 1]])
>>> multilabel_confusion_matrix(y_true, y_pred)
array([[[1, 0],
        [0, 1]],

       [[1, 0],
        [0, 1]],

       [[0, 1],
        [1, 0]]])
```

或者可以为每个样本的标签构建一个混淆矩阵：

```
>>> multilabel_confusion_matrix(y_true, y_pred, samplewise=True)
array([[[1, 0],
        [1, 1]],

       [[1, 1],
        [0, 1]]])
```

这是一个演示如何使用多类输入 `multilabel_confusion_matrix` 函数的示例 ：

```
>>> y_true = ["cat", "ant", "cat", "cat", "ant", "bird"]
>>> y_pred = ["ant", "ant", "cat", "cat", "ant", "cat"]
>>> multilabel_confusion_matrix(y_true, y_pred,
...                             labels=["ant", "bird", "cat"])
array([[[3, 1],
        [0, 2]],

       [[5, 0],
        [1, 0]],

       [[2, 1],
        [1, 2]]])
```

以下是一些示例，演示了使用 `multilabel_confusion_matrix` 函数计算多标记指标矩阵输入问题中每个类的recall（或灵敏度），specificity,fall out和miss rate。

计算每个类的recall（也称为真阳性率或sensitivity）：

```
>>> y_true = np.array([[0, 0, 1],
...                    [0, 1, 0],
...                    [1, 1, 0]])
>>> y_pred = np.array([[0, 1, 0],
...                    [0, 0, 1],
...                    [1, 1, 0]])
>>> mcm = multilabel_confusion_matrix(y_true, y_pred)
>>> tn = mcm[:, 0, 0]
>>> tp = mcm[:, 1, 1]
>>> fn = mcm[:, 1, 0]
>>> fp = mcm[:, 0, 1]
>>> tp / (tp + fn)
array([1. , 0.5, 0. ])
```

计算每个类的specificity

```
>>> tn / (tn + fp)
array([1. , 0. , 0.5])
```

计算每个类的fall out

```
>>> fp / (fp + tn)
array([0. , 1. , 0.5])
```

计算每个类的miss rate

```
>>> fn / (fn + tp)
array([0. , 0.5, 1. ])
```

3.3.2.14. Receiver operating characteristic (ROC)

函数 `roc_curve` 计算 receiver operating characteristic curve, or ROC curve. 引用 Wikipedia :

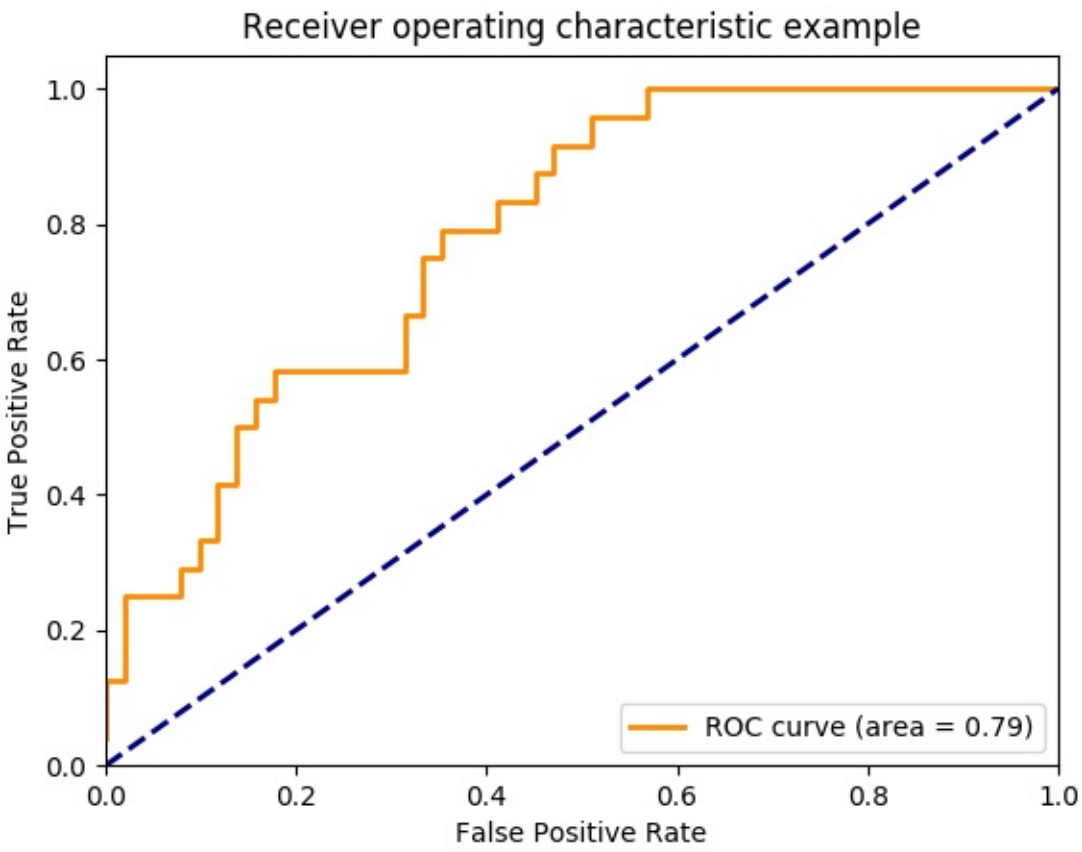
> “A receiver operating characteristic (ROC), 或者简单的 ROC 曲线, 是一个图形图, 说明了 binary classifier （二分分类器）系统的性能, 因为 discrimination threshold （鉴别阈值）是变化的。它是通过在不同的阈值设置下, 从 true positives out of the positives (TPR = true positive 比例) 与 false positives out of the negatives (FPR = false positive 比例) 绘制 true positive 的比例来创建的。 TPR 也称为 sensitivity（灵敏度）, FPR 是减去 specificity（特异性） 或 true negative 比例。”

该函数需要真正的 binar value （二分值）和 target scores（目标分数）, 这可以是 positive class 的 probability estimates （概率估计）, confidence values（置信度值）或 binary decisions（二分决策）。 这是一个如何使用 `roc_curve` 函数的小例子:

```
>>> import numpy as np
>>> from sklearn.metrics import roc_curve
>>> y = np.array([1, 1, 2, 2])
>>> scores = np.array([0.1, 0.4, 0.35, 0.8])
```

```
>>> fpr, tpr, thresholds = roc_curve(y, scores, pos_label=2)
>>> fpr
array([0. , 0. , 0.5, 0.5, 1. ])
>>> tpr
array([0. , 0.5, 0.5, 1. , 1. ])
>>> thresholds
array([1.8 , 0.8 , 0.4 , 0.35, 0.1 ])
```

该图显示了这样的 ROC 曲线的示例：



`roc_auc_score` 函数计算 receiver operating characteristic (ROC) 曲线下的面积，也由 AUC 和 AUROC 表示。通过计算 roc 曲线下的面积，曲线信息总结为一个数字。 有关更多的信息，请参阅 Wikipedia article on AUC .

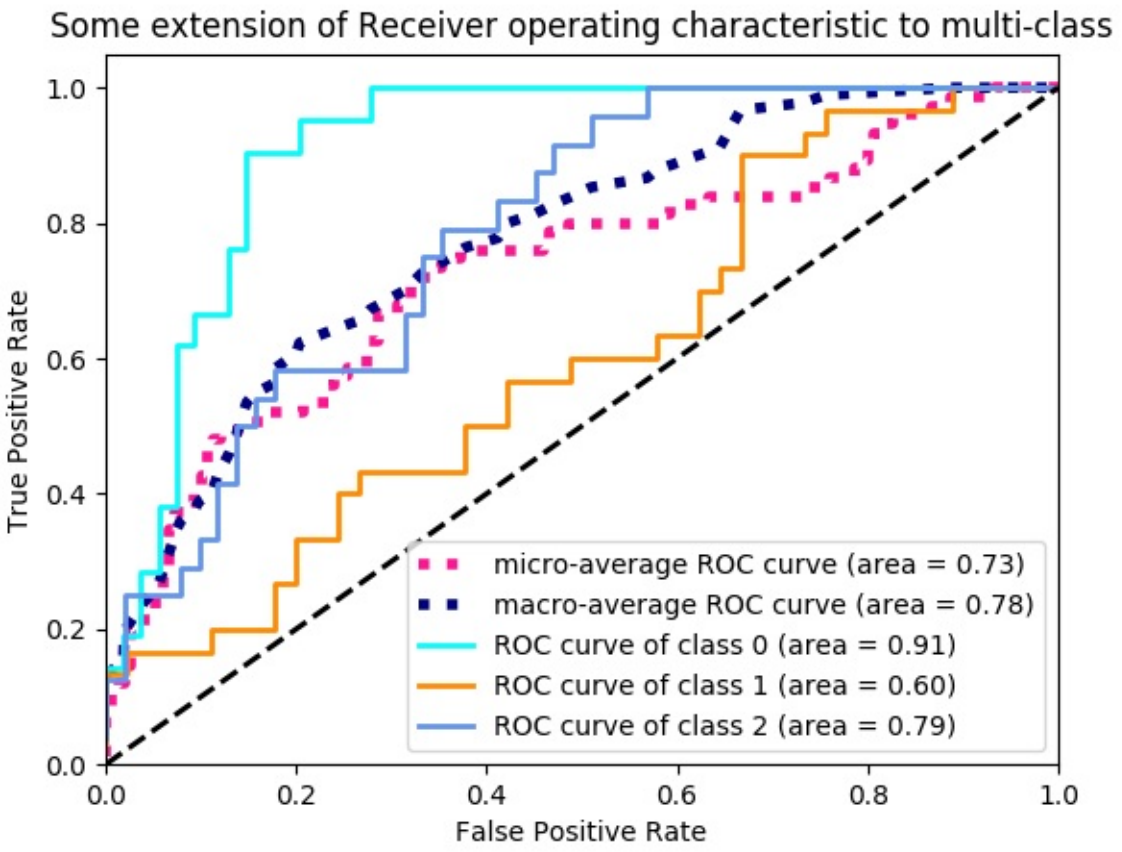
```
>>> import numpy as np
>>> from sklearn.metrics import roc_auc_score
>>> y_true = np.array([0, 0, 1, 1])
>>> y_scores = np.array([0.1, 0.4, 0.35, 0.8])
>>> roc_auc_score(y_true, y_scores)
0.75
```

在 multi-label classification （多标签分类）中， `roc_auc_score` 函数通过在标签上进行平均来扩展 above .

与诸如 subset accuracy （子集精确度），Hamming loss（汉明损失）或 F1 score 的 metrics（指标）

相比， ROC 不需要优化每个标签的阈值。 `roc_auc_score` 函数也可以用于 multi-class classification （多类分类），如果预测的输出被 binarized （二分化）。

在那些高虚警率(false positive rate)不被容忍的情况下，roc_auc_score 函数的参数 max_fpr 可被用来把ROC曲线累加到一个给定的限制。



示例：

- 参阅 Receiver Operating Characteristic (ROC) 例如使用 ROC 来评估分类器输出的质量。
- 参阅 Receiver Operating Characteristic (ROC) with cross validation 例如使用 ROC 来评估分类器输出质量，使用 cross-validation （交叉验证）。
- 参阅 Species distribution modeling 例如使用 ROC 来 model species distribution 模拟物种分布。

3.3.2.15. 零一损失

`zero_one_loss` 函数通过 n_{samples} 计算 0-1 classification loss (L_{0-1}) 的 sum （和）或 average （平均值）。默认情况下，函数在样本上 normalizes （标准化）。要获得 L_{0-1} 的总和，将 `normalize` 设置为 `False` 。

在 multilabel classification （多标签分类）中，如果零标签与标签严格匹配，则 `zero_one_loss` 将一个子集作为一个子集，如果有任何错误，则为零。默认情况下，函数返回不完全预测子集的百分比。为了得到这样的子集的计数，将 `normalize` 设置为 `False` 。

如果 $\hat{y}_i$ 是第 i 个样本的预测值， y_i 是相应的真实值，则 0-1 loss L_{0-1} 定义为：

$$L_{0-1}(y_i, \hat{y}_i) = 1(\hat{y}_i \neq y_i)$$

其中 $1(x)$ 是 indicator function.

```
>>> from sklearn.metrics import zero_one_loss
>>> y_pred = [1, 2, 3, 4]
>>> y_true = [2, 2, 3, 4]
>>> zero_one_loss(y_true, y_pred)
0.25
>>> zero_one_loss(y_true, y_pred, normalize=False)
1
```

在具有 binary label indicators （二分标签指示符）的 multilabel （多标签）情况下，第一个标签集 [0,1] 有错误：

```
>>> zero_one_loss(np.array([[0, 1], [1, 1]]), np.ones((2, 2)))
0.5

>>> zero_one_loss(np.array([[0, 1], [1, 1]]), np.ones((2, 2)),  normalize=False)
1
```

> 示例：
>
> - 参阅 Recursive feature elimination with cross-validation 例如 zero one loss 使用以通过 cross-validation （交叉验证）执行递归特征消除。

3.3.2.16. Brier 分数损失

`brier_score_loss` 函数计算二进制类的 Brier 分数 。引用维基百科：

> “Brier 分数是一个特有的分数函数，用于衡量概率预测的准确性。它适用于预测必须将概率分配给一组相互排斥的离散结果的任务。”

该函数返回的是 实际结果与可能结果 的预测概率之间均方差的得分。 实际结果必须为1或0（真或假），而实际结果的预测概率可以是0到1之间的值。

Brier 分数损失也在0到1之间，分数越低（均方差越小），预测越准确。它可以被认为是对一组概率预测的 “校准” 的度量。

$$BS = \frac{1}{N}\sum_{t=1}^{N}(f_t - o_t)^2$$

其中： N 是预测的总数， f_t 是实际结果 o_t 的预测概率。

这是一个使用这个函数的小例子：

```
>>> import numpy as np
```

```
>>> from sklearn.metrics import brier_score_loss
>>> y_true = np.array([0, 1, 1, 0])
>>> y_true_categorical = np.array(["spam", "ham", "ham", "spam"])
>>> y_prob = np.array([0.1, 0.9, 0.8, 0.4])
>>> y_pred = np.array([0, 1, 1, 0])
>>> brier_score_loss(y_true, y_prob)
0.055
>>> brier_score_loss(y_true, 1-y_prob, pos_label=0)
0.055
>>> brier_score_loss(y_true_categorical, y_prob, pos_label="ham")
0.055
>>> brier_score_loss(y_true, y_prob > 0.5)
0.0
```

示例：

- 请参阅分类器的概率校准 Probability calibration of classifiers ，通过 Brier 分数损失使用示例 来执行分类器的概率校准。

参考资料：

- G.Brier, 以概率表示的预测验证 , 月度天气评估78.1（1950）

3.3.3. 多标签排名指标

在多分类学习中，每个样本可以具有与其相关联的任何数量的真实标签。目标是给予高分，更好地评价真实标签。

3.3.3.1. 覆盖误差

`coverage_error` 函数计算必须包含在最终预测中的标签的平均数，以便预测所有真正的标签。 如果您想知道有多少 top 评分标签，您必须通过平均来预测，而不会丢失任何真正的标签，这很有用。 因此，此指标的最佳价值是真正标签的平均数量。

注意

我们的实现的得分比 Tsoumakas 等人在2010年的提出的计算方式大1。这扩展了它来处理一个具有0个真实标签实例的退化情况的能力。

正式地，给定真实标签 $y \in \{0,1\}^{n_{\text{samples}} \times n_{\text{labels}}}$ 的二进制指示矩阵和与每个标签 $\hat{f} \in \mathbb{R}^{n_{\text{samples}} \times n_{\text{labels}}}$ 相关联的分数，覆盖范围被定义为

$$coverage(y, \hat{f}) = \frac{1}{n_{\text{samples}}} \sum_{i=0}^{n_{\text{samples}}-1} \max_{j:y_{ij}=1} \text{rank}_{ij}$$

与 $\text{rank}_{ij} = \left|\left\{k : \hat{f}_{ik} \geq \hat{f}_{ij}\right\}\right|$ 。给定等级定义，通过给出将被分配给所有绑定值的最大等级， `y_scores` 中的关系会被破坏。

这是一个使用这个函数的小例子：

```
>>> import numpy as np
>>> from sklearn.metrics import coverage_error
>>> y_true = np.array([[1, 0, 0], [0, 0, 1]])
>>> y_score = np.array([[0.75, 0.5, 1], [1, 0.2, 0.1]])
>>> coverage_error(y_true, y_score)
2.5
```

3.3.3.2. 标签排名平均精度

`label_ranking_average_precision_score` 函数实现标签排名平均精度（LRAP）。 该度量值与 `average_precision_score` 函数相关联，但是基于标签排名的概念，而不是精确度和召回。

标签排名平均精度（LRAP）是分配给每个样本的每个真实标签的平均值，真实对总标签与较低分数的比率。 如果能够为每个样本相关标签提供更好的排名，这个指标就会产生更好的分数。 获得的得分总是严格大于0，最佳值为1。如果每个样本只有一个相关标签，则标签排名平均精度等于 平均倒数等级 。

正式地，给定真实标签 $y \in \mathcal{R}^{n_{\text{samples}} \times n_{\text{labels}}}$ 的二进制指示矩阵和与每个标签 $\hat{f} \in \mathcal{R}^{n_{\text{samples}} \times n_{\text{labels}}}$ 相关联的得分，平均精度被定义为

$$LRAP(y, \hat{f}) = \frac{1}{n_{\text{samples}}} \sum_{i=0}^{n_{\text{samples}}-1} \frac{1}{|y_i|} \sum_{j:y_{ij}=1} \frac{|\mathcal{L}_{ij}|}{\text{rank}_{ij}}$$

与 $\mathcal{L}_{ij} = \left\{k : y_{ik} = 1, \hat{f}_{ik} \geq \hat{f}_{ij}\right\}$, $\text{rank}_{ij} = \left|\left\{k : \hat{f}_{ik} \geq \hat{f}_{ij}\right\}\right|$ 和 $|\cdot|$ 是集合的 l0 范数或基数。

这是一个使用这个函数的小例子：

```
>>> import numpy as np
>>> from sklearn.metrics import label_ranking_average_precision_score
>>> y_true = np.array([[1, 0, 0], [0, 0, 1]])
>>> y_score = np.array([[0.75, 0.5, 1], [1, 0.2, 0.1]])
>>> label_ranking_average_precision_score(y_true, y_score)
0.416...
```

3.3.3.3. 排序损失

`label_ranking_loss` 函数计算在样本上平均排序错误的标签对数量的排序损失，即真实标签的分数低于假标签，由虚假和真实标签的倒数加权。最低可实现的排名损失为零。

正式地，给定真相标签 $y \in \{0, 1\}^{n_{\text{samples}} \times n_{\text{labels}}}$ 的二进制指示矩阵和与每个标签 $\hat{f} \in \mathbb{R}^{n_{\text{samples}} \times n_{\text{labels}}}$ 相关联的得分，排序损失被定义为

$$\text{ranking_loss}(y, \hat{f}) = \frac{1}{n_{\text{samples}}} \sum_{i=0}^{n_{\text{samples}}-1} \frac{1}{|y_i|(n_{\text{labels}} - |y_i|)} \left|\left\{(k, l) : \hat{f}_{ik} < \hat{f}_{il}, y_{ik} = 1, y_{il} = 0\right\}\right|$$

其中 $|\cdot|$ 是 ℓ_0 范数或集合的基数。

这是一个使用这个函数的小例子：

```
>>> import numpy as np
>>> from sklearn.metrics import label_ranking_loss
>>> y_true = np.array([[1, 0, 0], [0, 0, 1]])
>>> y_score = np.array([[0.75, 0.5, 1], [1, 0.2, 0.1]])
>>> label_ranking_loss(y_true, y_score)
0.75...
>>> # With the following prediction, we have perfect and minimal loss
>>> y_score = np.array([[1.0, 0.1, 0.2], [0.1, 0.2, 0.9]])
>>> label_ranking_loss(y_true, y_score)
0.0
```

> 参考资料：
>
> - Tsoumakas, G., Katakis, I., & Vlahavas, I. (2010). 挖掘多标签数据。在数据挖掘和知识发现手册（第667-685页）。美国 Springer.

3.3.4. 回归指标

该 `sklearn.metrics` 模块实现了一些 loss, score 以及 utility 函数以测量 regression（回归）的性能. 其中一些已经被加强以处理多个输出的场景： `mean_squared_error` , `mean_absolute_error` , `explained_variance_score` 和 `r2_score` .

这些函数有 `multioutput` 这样一个 keyword（关键的）参数, 它指定每一个目标的 score（得分）或 loss（损失）的平均值的方式. 默认是 `'uniform_average'` , 其指定了输出时一致的权重均值. 如果一个 `ndarray` 的 shape `(n_outputs,)` 被传递, 则其中的 entries（条目）将被解释为权重，并返回相应的加权平均值. 如果 `multioutput` 指定了 `'raw_values'` , 则所有未改变的部分 score（得分）或 loss（损失）将以 `(n_outputs,)` 形式的数组返回.

该 `r2_score` 和 `explained_variance_score` 函数接受一个额外的值 `'variance_weighted'` 用于 `multioutput` 参数. 该选项通过相应目标变量的方差使得每个单独的 score 进行加权. 该设置量化了全局捕获的未缩放方差. 如果目标变量的大小不一样, 则该 score 更好地解释了较高的方差变量. `multioutput='variance_weighted'` 是 `r2_score` 的默认值以向后兼容. 以后该值会被改成 `uniform_average` .

3.3.4.1. 解释方差得分

该 `explained_variance_score` 函数计算了 explained variance regression score（解释的方差回归得分）.

如果 $\hat{y}$ 是预估的目标输出, y 是相应（正确的）目标输出, 并且 Var is 方差, 标准差的平方, 那么解释的方差预估如下：

$$\text{explained_variance}(y, \hat{y}) = 1 - \frac{Var\{y - \hat{y}\}}{Var\{y\}}$$

最好的得分是 1.0，值越低越差.

下面是一下有关 `explained_variance_score` 函数使用的一些例子：

```
>>> from sklearn.metrics import explained_variance_score
>>> y_true = [3, -0.5, 2, 7]
>>> y_pred = [2.5, 0.0, 2, 8]
>>> explained_variance_score(y_true, y_pred)
0.957...
>>> y_true = [[0.5, 1], [-1, 1], [7, -6]]
>>> y_pred = [[0, 2], [-1, 2], [8, -5]]
>>> explained_variance_score(y_true, y_pred, multioutput='raw_values')
...
array([ 0.967...,  1.        ])
>>> explained_variance_score(y_true, y_pred, multioutput=[0.3, 0.7])
...
0.990...
```

3.3.4.2. 最大误差

`max_error` 函数计算最大残差，该度量捕获预测值和真实值之间的最坏情况误差。在完全拟合的单输出回归模型中，训练集上的 `max_error` 将为0，尽管在现实世界中这是极不可能的，但是这个度量显示了模型在拟合时的误差程度。

如果 $\hat{y}_i$ 是 i-th 样本的预测值，并且 y_i 是对应的真实值，则将最大误差定义为

$$\text{Max Error}(y, \hat{y}) = max(|y_i - \hat{y}_i|)$$

以下是 `max_error` 函数的一个例子：

```
>>> from sklearn.metrics import max_error
>>> y_true = [3, 2, 7, 1]
>>> y_pred = [9, 2, 7, 1]
>>> max_error(y_true, y_pred)
6
```

`max_error` 不支持多输出

3.3.4.3. 平均绝对误差

该 `mean_absolute_error` 函数计算了 平均绝对误差，一个对应绝对误差损失预期值或者 $l1$-norm 损失的风险度量.

如果 $\hat{y}_i$ 是 i-th 样本的预测值，并且 y_i 是对应的真实值，则平均绝对误差 (MAE) 预估的 n_{samples} 定义如下

$$\text{MAE}(y, \hat{y}) = \frac{1}{n_{\text{samples}}} \sum_{i=0}^{n_{\text{samples}}-1} |y_i - \hat{y}_i| .$$

下面是一个有关 `mean_absolute_error` 函数用法的小例子：

```
>>> from sklearn.metrics import mean_absolute_error
>>> y_true = [3, -0.5, 2, 7]
>>> y_pred = [2.5, 0.0, 2, 8]
>>> mean_absolute_error(y_true, y_pred)
0.5
>>> y_true = [[0.5, 1], [-1, 1], [7, -6]]
>>> y_pred = [[0, 2], [-1, 2], [8, -5]]
>>> mean_absolute_error(y_true, y_pred)
0.75
>>> mean_absolute_error(y_true, y_pred, multioutput='raw_values')
array([ 0.5,  1. ])
>>> mean_absolute_error(y_true, y_pred, multioutput=[0.3, 0.7])
...
0.849...
```

3.3.4.4. 均方误差

该 `mean_squared_error` 函数计算了 均方误差，一个对应于平方（二次）误差或损失的预期值的风险度量.

如果 $\hat{y}_i$ 是 i-th 样本的预测值，并且 y_i 是对应的真实值，则均方误差（MSE）预估的 n_{samples} 定义如下

$$\text{MSE}(y, \hat{y}) = \frac{1}{n_{\text{samples}}} \sum_{i=0}^{n_{\text{samples}}-1} (y_i - \hat{y}_i)^2 .$$

下面是一个有关 `mean_squared_error` 函数用法的小例子：

```
>>> from sklearn.metrics import mean_squared_error
>>> y_true = [3, -0.5, 2, 7]
>>> y_pred = [2.5, 0.0, 2, 8]
>>> mean_squared_error(y_true, y_pred)
0.375
>>> y_true = [[0.5, 1], [-1, 1], [7, -6]]
>>> y_pred = [[0, 2], [-1, 2], [8, -5]]
>>> mean_squared_error(y_true, y_pred)
0.7083...
```

> 示例
>
> - 点击 Gradient Boosting regression 查看均方误差用于梯度上升（gradient boosting）回归的使用例子。

3.3.4.5. 均方误差对数

该 `mean_squared_log_error` 函数计算了一个对应平方对数（二次）误差或损失的预估值风险度量.

如果 $\hat{y}_i$ 是 i-th 样本的预测值，并且 y_i 是对应的真实值，则均方误差对数（MSLE）预估的 n_{samples} 定义如下

$$\text{MSLE}(y, \hat{y}) = \frac{1}{n_{\text{samples}}} \sum_{i=0}^{n_{\text{samples}}-1} (\log_e(1 + y_i) - \log_e(1 + \hat{y}_i))^2.$$

其中 $\log_e(x)$ 表示 x 的自然对数. 当目标具有指数增长的趋势时，该指标最适合使用，例如人口数量，跨年度商品的平均销售额等. 请注意，该指标会对低于预测的估计值进行估计.

下面是一个有关 `mean_squared_log_error` 函数用法的小例子:

```
>>> from sklearn.metrics import mean_squared_log_error
>>> y_true = [3, 5, 2.5, 7]
>>> y_pred = [2.5, 5, 4, 8]
>>> mean_squared_log_error(y_true, y_pred)
0.039...
>>> y_true = [[0.5, 1], [1, 2], [7, 6]]
>>> y_pred = [[0.5, 2], [1, 2.5], [8, 8]]
>>> mean_squared_log_error(y_true, y_pred)
0.044...
```

3.3.4.6. 中位绝对误差

该 `median_absolute_error` 函数尤其有趣，因为它的离群值很强. 通过取目标和预测之间的所有绝对差值的中值来计算损失.

如果 $\hat{y}_i$ 是 i-th 样本的预测值，并且 y_i 是对应的真实值，则中位绝对误差（MedAE）预估的 n_{samples} 定义如下

该 `median_absolute_error` 函数不支持多输出.

下面是一个有关 `median_absolute_error` 函数用法的小例子:

```
>>> from sklearn.metrics import median_absolute_error
>>> y_true = [3, -0.5, 2, 7]
>>> y_pred = [2.5, 0.0, 2, 8]
>>> median_absolute_error(y_true, y_pred)
0.5
```

3.3.4.7. R^2 score, 可决系数

该 `r2_score` 函数计算了 computes R^2, 即 可决系数. 它提供了将来样本如何可能被模型预测的估量. 最佳分

数为 1.0，可以为负数（因为模型可能会更糟）. 总是预测 y 的预期值，不考虑输入特征的常数模型将得到 R^2 得分为 0.0.

如果 $\hat{y}_i$ 是 i-th 样本的预测值，并且 y_i 是对应的真实值，则 R² 得分预估的 n_{samples} 定义如下

$$R^2(y, \hat{y}) = 1 - \frac{\sum_{i=0}^{n_{\text{samples}}-1}(y_i - \hat{y}_i)^2}{\sum_{i=0}^{n_{\text{samples}}-1}(y_i - \bar{y})^2}$$

其中 $\bar{y} = \frac{1}{n_{\text{samples}}}\sum_{i=0}^{n_{\text{samples}}-1} y_i$.

下面是一个有关 `r2_score` 函数用法的小例子：

```
>>> from sklearn.metrics import r2_score
>>> y_true = [3, -0.5, 2, 7]
>>> y_pred = [2.5, 0.0, 2, 8]
>>> r2_score(y_true, y_pred)
0.948...
>>> y_true = [[0.5, 1], [-1, 1], [7, -6]]
>>> y_pred = [[0, 2], [-1, 2], [8, -5]]
>>> r2_score(y_true, y_pred, multioutput='variance_weighted')
...
0.938...
>>> y_true = [[0.5, 1], [-1, 1], [7, -6]]
>>> y_pred = [[0, 2], [-1, 2], [8, -5]]
>>> r2_score(y_true, y_pred, multioutput='uniform_average')
...
0.936...
>>> r2_score(y_true, y_pred, multioutput='raw_values')
...
array([ 0.965...,  0.908...])
>>> r2_score(y_true, y_pred, multioutput=[0.3, 0.7])
...
0.925...
```

> 示例：
>
> - 点击 Lasso and Elastic Net for Sparse Signals 查看关于R²用于评估在Lasso and Elastic Net on sparse signals上的使用.

3.3.5. 聚类指标

该 `sklearn.metrics` 模块实现了一些 loss, score 和 utility 函数. 更多信息请参阅 聚类性能度量 部分，例如聚类，以及用于二分聚类的 Biclustering 评测.

3.3.6. 虚拟估计

在进行监督学习的过程中，简单的 sanity check（理性检查）包括将人的估计与简单的经验法则进行比较. `DummyClassifier` 实现了几种简单的分类策略:

- `stratified` 通过在训练集类分布方面来生成随机预测.
- `most_frequent` 总是预测训练集中最常见的标签.
- `prior` 总是给出能够最大化类先验概率的预测（类似于 most_frequent）并且 predict_proba 返回类先验概率.
- `uniform` 随机产生预测.
- `constant` 总是预测用户提供的常量标签.当 positive class（正类）较少时,这种方法的主要动机是 F1-scoring.

请注意，这些所有的策略，`predict` 方法彻底的忽略了输入数据!

为了说明 `DummyClassifier`，首先让我们创建一个不平衡数据集.

```
>>> from sklearn.datasets import load_iris
>>> from sklearn.model_selection import train_test_split
>>> iris = load_iris()
>>> X, y = iris.data, iris.target
>>> y[y != 1] = -1
>>> X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)
```

接下来，让我们比较一下 `SVC` 和 `most_frequent` 的准确性.

```
>>> from sklearn.dummy import DummyClassifier
>>> from sklearn.svm import SVC
>>> clf = SVC(kernel='linear', C=1).fit(X_train, y_train)
>>> clf.score(X_test, y_test)
0.63...
>>> clf = DummyClassifier(strategy='most_frequent',random_state=0)
>>> clf.fit(X_train, y_train)
DummyClassifier(constant=None, random_state=0, strategy='most_frequent')
>>> clf.score(X_test, y_test)
0.57...
```

我们看到 `SVC` 没有比一个 dummy classifier（虚拟分类器）好很多. 现在，让我们来更改一下 kernel:

```
>>> clf = SVC(gamma='scale', kernel='rbf', C=1).fit(X_train, y_train)
>>> clf.score(X_test, y_test)
0.94...
```

我们注意到准确率提升到将近 100%. 建议采用交叉验证策略，以更好地估计精度，如果不是太耗 CPU 的话. 更多信息请参阅 交叉验证：评估估算器的表现 部分. 此外，如果要优化参数空间，强烈建议您使用适当的方法；更多详情请参阅 调整估计器的超参数 部分.

通常来说，当分类器的准确度太接近随机情况时，这可能意味着出现了一些问题：特征没有帮助，超参数没有正确调整，class 不平衡造成分类器有问题等…

`DummyRegressor` 还实现了四个简单的经验法则来进行回归:

- `mean` 总是预测训练目标的平均值.
- `median` 总是预测训练目标的中位数.
- `quantile` 总是预测用户提供的训练目标的 quantile（分位数）.
- `constant` 总是预测由用户提供的常数值.

在以上所有的策略中, `predict` 方法完全忽略了输入数据.

我们一直在努力

apachecn/sklearn-doc-zh



3.4. 模型持久化

校验者：@why2lyj（Snow Wang）@小瑶翻译者：@那伊抹微笑

在训练完 scikit-learn 模型之后，最好有一种方法来将模型持久化以备将来使用，而无需重新训练。 以下部分为您提供了有关如何使用 pickle 来持久化模型的示例。 在使用 pickle 序列化时，我们还将回顾一些安全性和可维护性方面的问题。

pickle的另一种方法是使用相关项目中列出的模型导出工具之一将模型导出为另一种格式。与pickle不同，一旦导出，就不能恢复完整的Scikit-learn estimator对象，但是可以部署模型进行预测，通常可以使用支持开放模型交换格式的工具，如“ONNX”或“PMML”。

3.4.1. 持久化示例

可以通过使用 Python 的内置持久化模型将训练好的模型保存在 scikit 中，它名为 pickle：

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

>>> import pickle
>>> s = pickle.dumps(clf)
>>> clf2 = pickle.loads(s)
>>> clf2.predict(X[0:1])
array([0])
>>> y[0]
0
```

在这个 scikit 的特殊示例中，使用 joblib 来替换 pickle（ `joblib.dump` & `joblib.load` ）可能会更有意思，这对于内部带有 numpy 数组的对象来说更为高效， 通常情况下适合 scikit-learn estimators（预估器），但是也只能是 pickle 到硬盘而不是字符串：

```
>>> from sklearn.externals import joblib
>>> joblib.dump(clf, 'filename.pkl')
```

之后你可以使用以下方式回调 pickled model 可能在另一个 Python 进程中）：

```
>>> clf = joblib.load('filename.pkl')
```

> 注意 `joblib.dump` 和 `joblib.load` 函数也接收类似 file-like 的对象而不是文件名。 更多有关使用 Joblib 来持久化数据的信息可以参阅 这里.

3.4.2. 安全性和可维护性的局限性

pickle（和通过扩展的 joblib），在安全性和可维护性方面存在一些问题。 有以下原因，

- 绝对不要使用未经 pickle 的不受信任的数据，因为它可能会在加载时执行恶意代码。
- 虽然一个版本的 scikit-learn 模型可以在其他版本中加载，但这完全不建议并且也是不可取的。 还应该了解到，对于这些数据执行的操作可能会产生不同及意想不到的结果。

为了用以后版本的 scikit-learn 来重构类似的模型，额外的元数据应该随着 pickled model 一起被保存：

- 训练数据，例如：引用不可变的快照
- 用于生成模型的 python 源代码
- scikit-learn 的各版本以及各版本对应的依赖包
- 在训练数据的基础上获得的交叉验证得分

这样可以检查交叉验证得分是否与以前相同。

由于模型内部表示可能在两种不同架构上不一样，因此不支持在一个架构上转储模型并将其加载到另一个体系架构上。

如果您想要了解更多关于这些问题以及其它可能的序列化方法，请参阅这个 Alex Gaynor 的演讲.

我们一直在努力

apachecn/sklearn-doc-zh



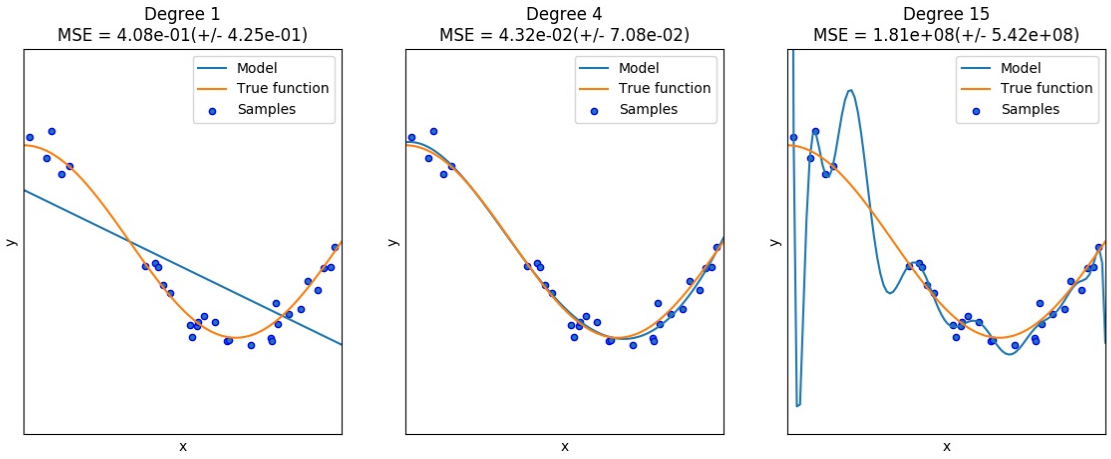
3.5. 验证曲线： 绘制分数以评估模型

校验者： @正版乔@正版乔@小瑶翻译者： @Xi

每种估计器都有其优势和缺陷。它的泛化误差可以用偏差、方差和噪声来分解。估计值的 偏差 是不同训练集的平均误差。估计值的 方差 用来表示它对训练集的变化有多敏感。噪声是数据的一个属性。

$$f(x) = \cos(\frac{3}{2}\pi x)$$

在下面的图中，我们可以看到一个函数 和这个函数的一些噪声样本。 我们用三个不同的估计来拟合函数： 多项式特征为1,4和15的线性回归。我们看到，第一个估计最多只能为样本和真正的函数提供一个很差的拟合 ，因为它太简单了(高偏差），第二个估计几乎完全近似，最后一个估计完全接近训练数据， 但不能很好地拟合真实的函数，即对训练数据的变化（高方差）非常敏感。



偏差和方差是估计所固有的属性，我们通常必须选择合适的学习算法和超参数，以使得偏差和 方差都尽可能的低（参见偏差-方差困境）。 另一种降低方差的方法是使用更多的训练数据。不论如何，如果真实函数过于复杂并且不能用一个方 差较小的估计值来近似，则只能去收集更多的训练数据。

在一个简单的一维问题中，我们可以很容易看出估计值是否存在偏差或方差。然而，在高维空间中， 模型可能变得非常难以具象化。 出于这种原因，使用以下工具通常是有帮助的。

> 示例：
>
> - Underfitting vs. Overfitting
> - Plotting Validation Curves
> - Plotting Learning Curves

3.5.1. 验证曲线

我们需要一个评分函数（参见模型评估：模型评估： 量化预测的质量）来验证一个模型， 例如分类器的准确性。 选择估计器的多个超参数的正确方法当然是网格搜索或类似方法 （参见调优估计的超参数 调整估计器的超参数 ），其选择一个或多个验证集上的分数最高的超参数。 请注意，如果我们基于验证分数优化了超参数，则验证分数就有偏差了，并且不再是一个良好的泛化估计。 为了得到正确的泛化估计，我们必须在另一个测试集上计算得分。

然而，绘制单个超参数对训练分数和验证分数的影响，有时有助于发现该估计是否因为某些超参数的值 而出现过拟合或欠拟合。

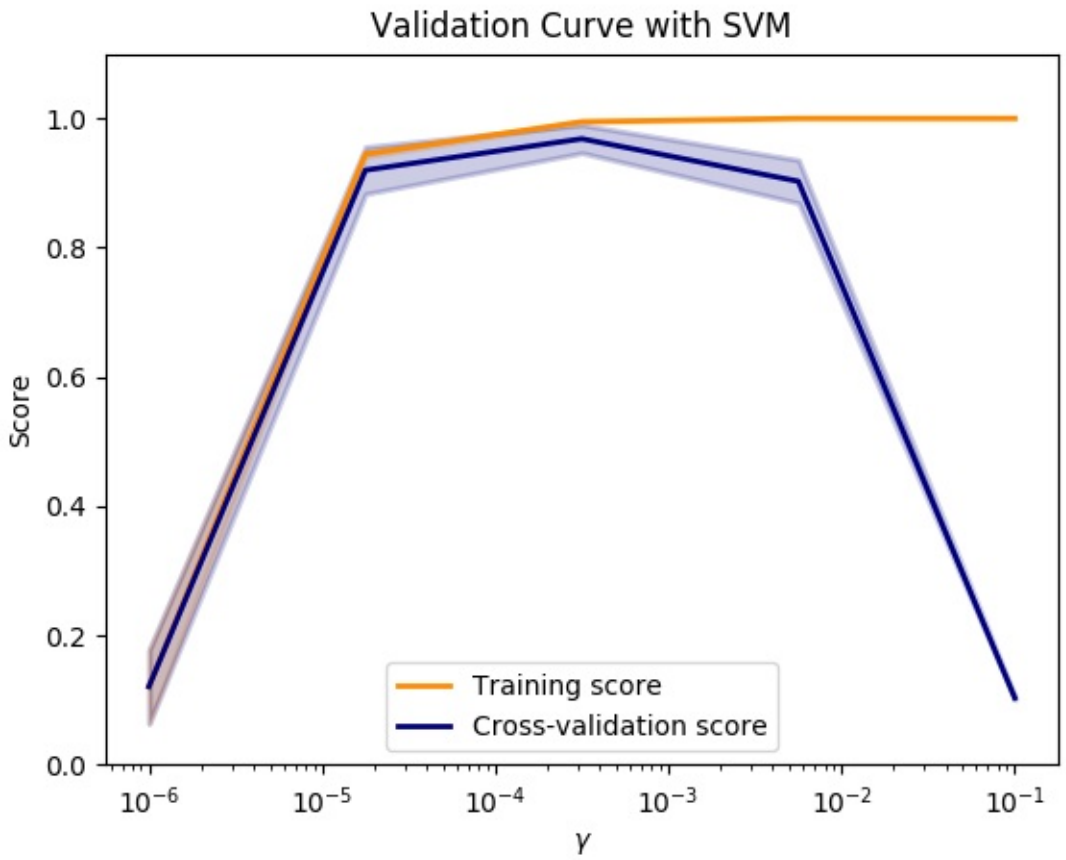
本例中,下面的方程 `validation_curve` 能起到如下作用：

```
>>> import numpy as np
>>> from sklearn.model_selection import validation_curve
>>> from sklearn.datasets import load_iris
>>> from sklearn.linear_model import Ridge

>>> np.random.seed(0)
>>> iris = load_iris()
>>> X, y = iris.data, iris.target
>>> indices = np.arange(y.shape[0])
>>> np.random.shuffle(indices)
>>> X, y = X[indices], y[indices]

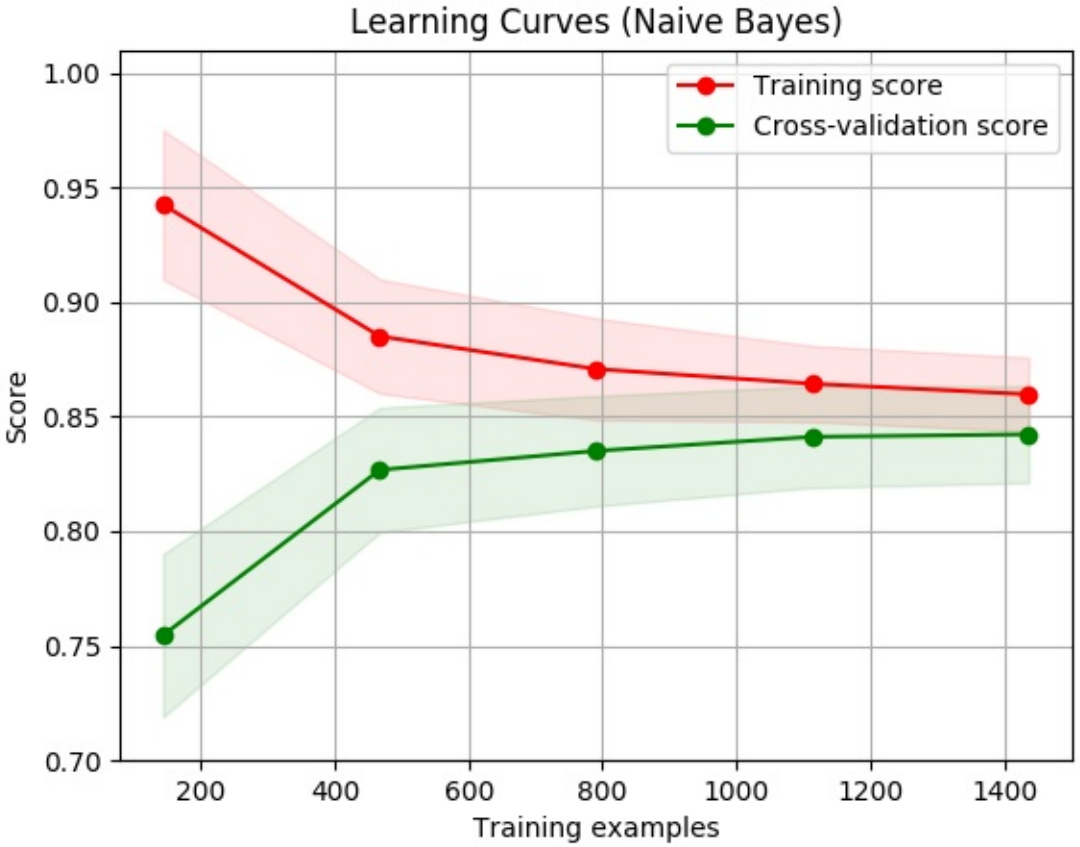
>>> train_scores, valid_scores = validation_curve(Ridge(), X, y, "alpha",
...                                               np.logspace(-7, 3, 3),
...                                               cv=5)
>>> train_scores
array([[0.93..., 0.94..., 0.92..., 0.91..., 0.92...],
       [0.93..., 0.94..., 0.92..., 0.91..., 0.92...],
       [0.51..., 0.52..., 0.49..., 0.47..., 0.49...]])
>>> valid_scores
array([[0.90..., 0.84..., 0.94..., 0.96..., 0.93...],
       [0.90..., 0.84..., 0.94..., 0.96..., 0.93...],
       [0.46..., 0.25..., 0.50..., 0.49..., 0.52...]])
```

如果训练得分和验证得分都很低，则估计器是不合适的。如果训练得分高，验证得分低，则估计器过拟合， 否则估计会拟合得很好。通常不可能有较低的训练得分和较高的验证得分。所有三种情况都可以 在下面的图中找到，其中我们改变了数字数据集上 SVM 的参数 γ 。

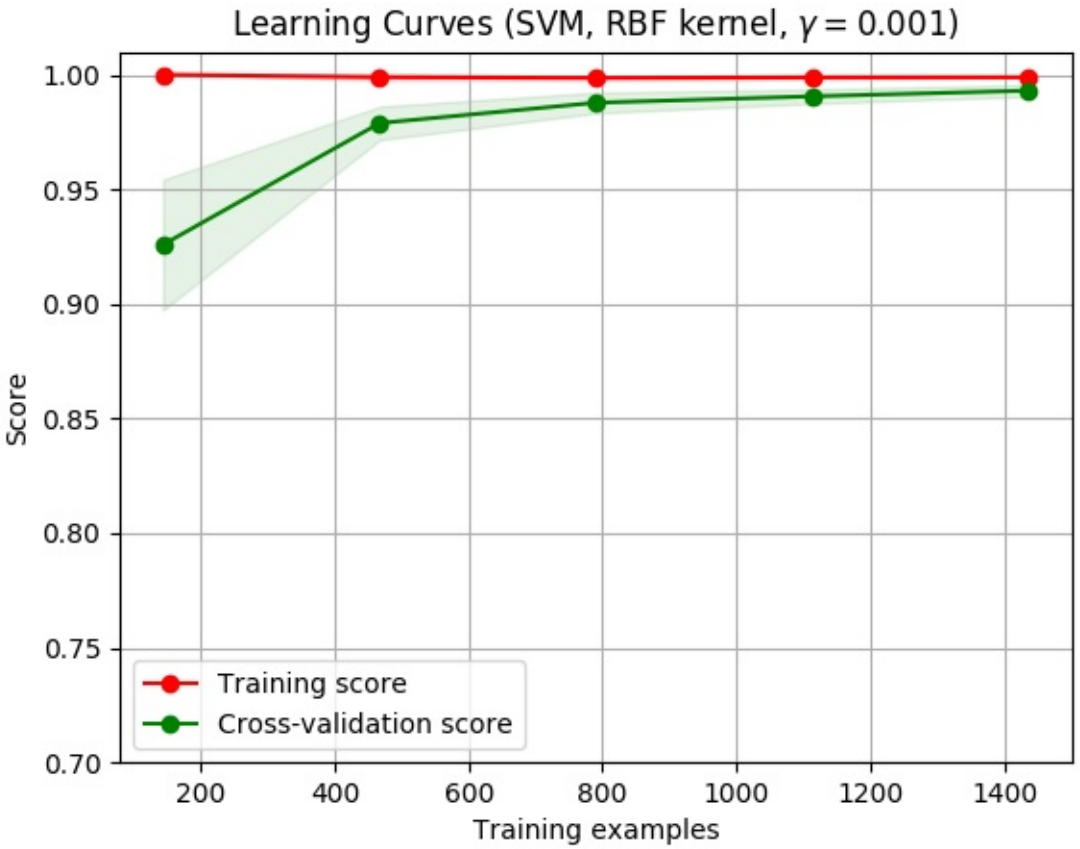


3.5.2. 学习曲线

学习曲线显示了对于不同数量的训练样本的估计器的验证和训练评分。它可以帮助我们发现从增加更多的训 练数据中能获益多少，以及估计是否受到更多来自方差误差或偏差误差的影响。如果在增加训练集大小时，验证分数和训练 分数都收敛到一个很低的值，那么我们将不会从更多的训练数据中获益。在下面的图中看到一个例子：朴素贝叶斯大致收敛到一个较低的分数。



我们可能需要使用评估器或者一个当前评估器的参数化形式来学习更复杂概念（例如有一个较低的偏差）。 如果训练样本的最大时，训练分数比验证分数得分大得多，那么增加训练样本很可能会增加泛化能力。 在下面的图中，可以看到支持向量机（SVM）可以从更多的训练样本中获益。



我们可以使用： `learning_curve` 函数来绘制这样一个学习曲线所需的值（已使用的样本数量，训练集 上的平均分数和验证集上的平均分数）：

```
>>> from sklearn.model_selection import learning_curve
>>> from sklearn.svm import SVC

>>> train_sizes, train_scores, valid_scores = learning_curve(
...     SVC(kernel='linear'), X, y, train_sizes=[50, 80, 110], cv=5)
>>> train_sizes
array([ 50, 80, 110])
>>> train_scores
array([[0.98..., 0.98 , 0.98..., 0.98..., 0.98...],
       [0.98..., 1.   , 0.98..., 0.98..., 0.98...],
       [0.98..., 1.   , 0.98..., 0.98..., 0.99...]])
>>> valid_scores
array([[1. ,  0.93...,  1. ,  1. ,  0.96...],
       [1. ,  0.96...,  1. ,  1. ,  0.96...],
       [1. ,  0.96...,  1. ,  1. ,  0.96...]])
```

我们一直在努力

apachecn/sklearn-doc-zh



4. 检验

- 4.1. 部分依赖图

我们一直在努力

apachecn/sklearn-doc-zh



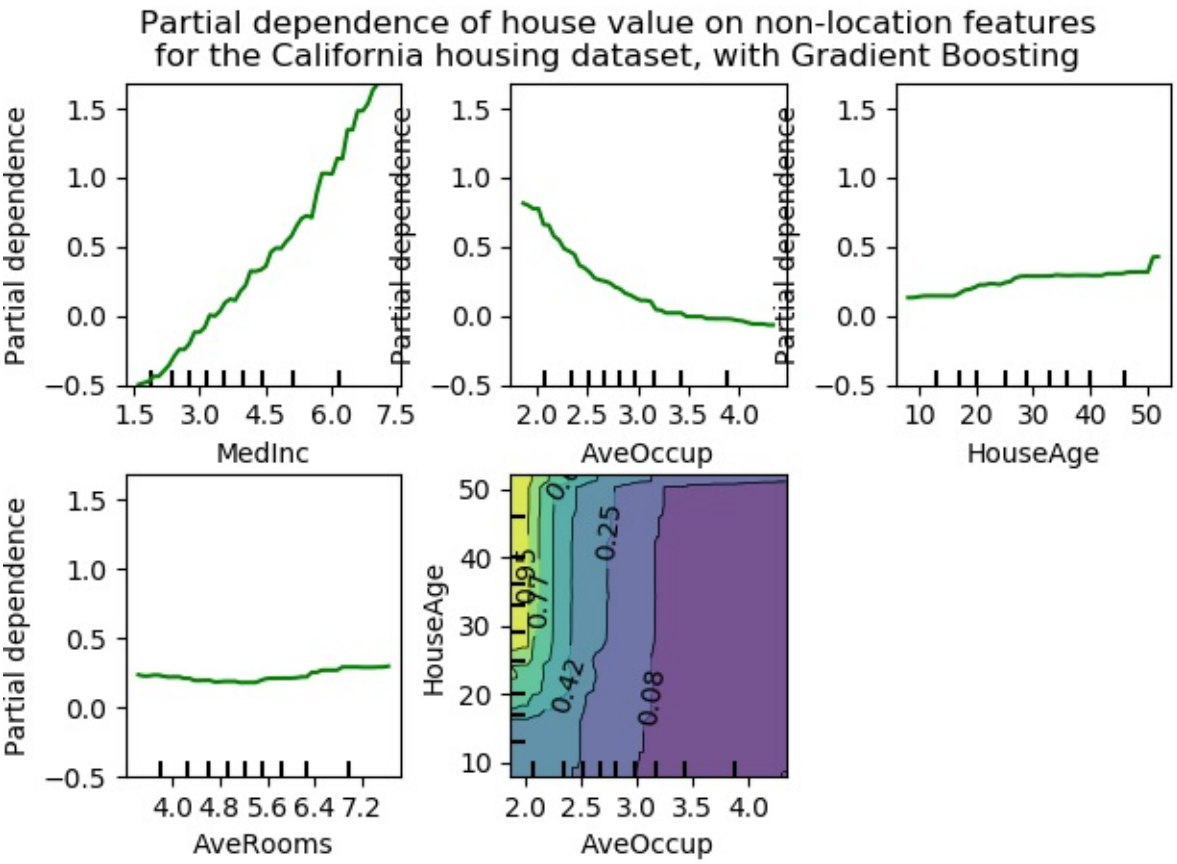
4.1. 部分依赖图

校验者： 待核验翻译者： @Loopy

部分依赖图(以下简称PDP)显示了目标响应[1]和一组“目标”特征之间的依赖关系，并边缘化所有其他特征（“补充”特征）的值。直观地，我们可以将部分依赖关系解释为预期目标响应作为“目标”特征的函数。

由于人类感知的限制，目标特征集的大小必须很小（通常是一个或两个），因此目标特征通常需要从最重要的特征中选择。

下图展示了使用GradientBoostingRegressor实现的，加利福尼亚州住房数据集的四个单向和一个双向PDP：



单向PDP告诉我们目标响应和目标特征(如线性、非线性)之间的相互作用。上图左上方的图表显示了一个地区的收入中位数对房价中位数的影响;我们可以清楚地看到它们之间的线性关系。注意，PDP假设目标特征独立于补体特征，而这一假设在实践中经常被推翻。

具有两个目标特征的PDP显示了这两个特征之间的相互作用。例如，上图中的双变量PDP显示了房价中值与房屋年龄和每户平均居住者的联合值之间的关系。我们可以清楚地看到这两个特征之间的相互作用:对于平均入住率大于2的情况，房价几乎与房屋的年龄无关，而对于数值小于2的情况，则与年龄有很强的依赖关系。

`sklearn.inspection` 提供了一个方便的函数 `plot_partial_dependency` 来创建单向和双向PDP。在下面的例子中，我们展示了如何创建一个局部依赖图的网格:两个用于特征0和1的单向PDP，以及两个特征之间的双向PDP:

```
>>> from sklearn.datasets import make_hastie_10_2
```

```
>>> from sklearn.ensemble import GradientBoostingClassifier
>>> from sklearn.inspection import plot_partial_dependence

>>> X, y = make_hastie_10_2(random_state=0)
>>> clf = GradientBoostingClassifier(n_estimators=100, learning_rate=1.0,
...     max_depth=1, random_state=0).fit(X, y)
>>> features = [0, 1, (0, 1)]
>>> plot_partial_dependence(clf, X, features)
```

您可以使用 `plt.gcf()` 和 `plt.gca()` 访问新创建的figure和axis对象。

对于多类分类，需要通过 `target` 参数设置类标签，来创建PDPs：

```
>>> from sklearn.datasets import load_iris
>>> iris = load_iris()
>>> mc_clf = GradientBoostingClassifier(n_estimators=10,
...     max_depth=1).fit(iris.data, iris.target)
>>> features = [3, 2, (3, 2)]
>>> plot_partial_dependence(mc_clf, X, features, target=0)
```

在多输出回归中，可以使用相同的参数 `target` 指定目标。

如果需要局部依赖函数的原始值而不是图，可以使用 `sklearn.inspection.partial_dependence` ：

```
>>> from sklearn.inspection import partial_dependence

>>> pdp, axes = partial_dependence(clf, X, [0])
>>> pdp
array([[ 2.466...,  2.466..., ...
>>> axes
[array([-1.624..., -1.592..., ...
```

求局部依赖关系的值直接由 `x` 生成。对于双向局部依赖关系，则生成值的二维网格。
由 `sklearn.inspection.partial_dependency` 返回的 `values` 字段会给出每个目标特征在网格中使用的实际值。它们也对应于图的轴线。

对于网格中“目标”特征的每个值，部分依赖函数需要将估计值对“补充”特征的所有可能值的预测边缘化。使用'brute'方法，可以通过用网格中的每个目标特征值替换X中的，并计算平均预测值来完成。

在决策树中，不需要参考训练数据('recursion'方法)就可以有效地评估这一点。对每个网格点执行加权树遍历：如果一个分割节点包含“目标”特征，则跟踪相应的左或右分支，否则同时跟踪两个分支，每个分支由输入该分支的训练样本的比例进行加权。最后，用所有访问叶的加权平均给出了部分依赖关系。注意，使用'recursion'方法时， `X` 只用于生成网格，而不是计算平均预测。平均预测总是根据决策树训练的数据来计算的。

> 脚注
>
> [1] 对于分类问题，目标响应可以是类的概率（二元分类的正类）或决策函数。
>
> 示例
>
> - Partial Dependence Plots

参考文献

- [HTF2009] T. Hastie, R. Tibshirani and J. Friedman, The Elements of Statistical Learning, Second Edition, Section 10.13.2, Springer, 2009.
- [Mol2019] C. Molnar, Interpretable Machine Learning, Section 5.1, 2019.

我们一直在努力

apachecn/sklearn-doc-zh



5. 数据集转换

scikit-learn 提供了一个用于转换数据集的库，它也许会 clean（清理）（请参阅 预处理数据），reduce（减少）（请参阅 无监督降维），expand（扩展）（请参阅 内核近似）或 generate（生成）（请参阅 特征提取） feature representations（特征表示）.

像其它预估计一样，它们由具有 `fit` 方法的类来表示，该方法从训练集学习模型参数（例如，归一化的平均值和标准偏差）以及 `transform` 方法将该转换模型应用于不可见数据. 同时 `fit_transform` 可以更方便和有效地建模与转换训练数据.

将 Pipeline（管道）和 FeatureUnion（特征联合）：合并的评估器 中 transformers（转换）使用并行的或者串联的方式合并到一起. 成对的矩阵，类别和核函数 涵盖将特征空间转换为 affinity matrices（亲和矩阵），而 预测目标（y）的转换 考虑在 scikit-learn 中使用目标空间的转换（例如. 标签分类）.

- 5.1. Pipeline（管道）和 FeatureUnion（特征联合）：合并的评估器
  - 5.1.1. Pipeline：链式评估器
  - 5.1.2. 回归中的目标转换
  - 5.1.3. FeatureUnion（特征联合）：复合特征空间
  - 5.1.4. 用于异构数据的列转换器
- 5.2. 特征提取
  - 5.2.1. 从字典类型加载特征
  - 5.2.2. 特征哈希（相当于一种降维技巧）
  - 5.2.3. 文本特征提取
  - 5.2.4. 图像特征提取
- 5.3 预处理数据
  - 5.3.1 标准化，也称去均值和方差按比例缩放
  - 5.3.2 非线性转换
  - 5.3.3 归一化
  - 5.3.4 类别特征编码
  - 5.3.5 离散化
  - 5.3.6 缺失值补全
  - 5.3.7 生成多项式特征
  - 5.3.8 自定义转换器
- 5.4 缺失值插补
  - 5.4.1 单变量与多变量插补
  - 5.4.2 单变量插补
  - 5.4.4 参考
  - 5.4.5 标记缺失值
- 5.5. 无监督降维
  - 5.5.1. PCA：主成份分析
  - 5.5.2. 随机投影
  - 5.5.3. 特征聚集
- 5.6. 随机投影
  - 5.6.1. Johnson-Lindenstrauss 辅助定理
  - 5.6.2. 高斯随机投影

    - 5.6.3. 稀疏随机矩阵
- 5.7. 内核近似
    - 5.7.1. 内核近似的 Nystroem 方法
    - 5.7.2. 径向基函数内核
    - 5.7.3. 加性卡方核
    - 5.7.4. Skewed Chi Squared Kernel (偏斜卡方核?暂译)
    - 5.7.5. 数学方面的细节
- 5.8. 成对的矩阵, 类别和核函数
    - 5.8.1. 余弦相似度
    - 5.8.2. 线性核函数
    - 5.8.3. 多项式核函数
    - 5.8.4. Sigmoid 核函数
    - 5.8.5. RBF 核函数
    - 5.8.6. 拉普拉斯核函数
    - 5.8.7. 卡方核函数
- 5.9. 预测目标 (`y`) 的转换
    - 5.9.1. 标签二值化
    - 5.9.2. 标签编码

我们一直在努力

apachecn/sklearn-doc-zh



5.1. Pipeline（管道）和 FeatureUnion（特征联合）：合并的评估器

校验者：@程威@Loopy翻译者：@Sehriff

变换器(Transformers)通常与分类器，回归器或其他的学习器组合在一起以构建复合估计器。 完成这件事的最常用工具是 Pipeline。 Pipeline 经常与 FeatureUnion 结合起来使用。 FeatureUnion 用于将变换器(transformers)的输出串联到复合特征空间(composite feature space)中。 TransformedTargetRegressor 用来处理变换 target（即对数变化 y）。 作为对比，Pipelines类只用来变换(transform)观测数据(X)。

5.1.1. Pipeline：链式评估器

`Pipeline` 可以把多个评估器链接成一个。这个是很有用的，因为处理数据的步骤一般都是固定的，例如特征选择、标准化和分类。 `Pipeline` 在这里有多种用途：

- 便捷性和封装性 你只要对数据调用 `fit` 和 `predict` 一次来适配所有的一系列评估器。
- 联合的参数选择 你可以一次 `grid search` 管道中所有评估器的参数。
- 安全性 训练转换器和预测器使用的是相同样本，管道有助于防止来自测试数据的统计数据泄露到交叉验证的训练模型中。

管道中的所有评估器，除了最后一个评估器，管道的所有评估器必须是转换器。（例如，必须有 `transform` 方法）. 最后一个评估器的类型不限（转换器、分类器等等）

5.1.1.1. 用法

5.1.1.1.1. 构造

`Pipeline` 使用一系列 `(key, value)` 键值对来构建，其中 `key` 是你给这个步骤起的名字， `value` 是一个评估器对象：

```
>>> from sklearn.pipeline import Pipeline
>>> from sklearn.svm import SVC
>>> from sklearn.decomposition import PCA
>>> estimators = [('reduce_dim', PCA()), ('clf', SVC())]
>>> pipe = Pipeline(estimators)
>>> pipe
Pipeline(memory=None,
         steps=[('reduce_dim', PCA(copy=True,...)),
                ('clf', SVC(C=1.0,...))], verbose=False)
```

功能函数 `make_pipeline` 是构建管道的缩写； 它接收多个评估器并返回一个管道，自动填充评估器名：

```
>>> from sklearn.pipeline import make_pipeline
>>> from sklearn.naive_bayes import MultinomialNB
```

```
>>> from sklearn.preprocessing import Binarizer
>>> make_pipeline(Binarizer(), MultinomialNB())
Pipeline(memory=None,
         steps=[('binarizer', Binarizer(copy=True, threshold=0.0)),
                ('multinomialnb', MultinomialNB(alpha=1.0,
                                                class_prior=None,
                                                fit_prior=True))],
         verbose=False)
```

5.1.1.1.2. 访问步骤

管道中的评估器作为一个列表保存在 `steps` 属性内，但可以通过索引或名称（`[idx]`）访问管道：

```
>>> pipe.steps[0]
('reduce_dim', PCA(copy=True, iterated_power='auto', n_components=None,
                   random_state=None, svd_solver='auto', tol=0.0,
                   whiten=False))
>>> pipe[0]
PCA(copy=True, iterated_power='auto', n_components=None, random_state=None,
    svd_solver='auto', tol=0.0, whiten=False)
>>> pipe['reduce_dim']
PCA(copy=True, ...)
```

管道的 `named_steps` 属性允许在交互式环境中使用tab补全，以按名称访问步骤：

```
>> pipe.named_steps.reduce_dim is pipe['reduce_dim']
True
```

还可以使用通常用于Python序列（如列表或字符串）的切片表示法提取子管道（尽管只允许步骤1）。这对于只执行一些转换（或它们的逆）是很方便的：

```
>>> pipe[:1]
Pipeline(memory=None, steps=[('reduce_dim', PCA(copy=True, ...))],...)
>>> pipe[-1:]
Pipeline(memory=None, steps=[('clf', SVC(C=1.0, ...))],...)
```

5.1.1.1.3. 嵌套参数

管道中的评估器参数可以通过 `<estimator>__<parameter>` 语义来访问：

```
>>> pipe.set_params(clf__C=10)
Pipeline(memory=None,
         steps=[('reduce_dim', PCA(copy=True, iterated_power='auto',...)),
                ('clf', SVC(C=10, cache_size=200, class_weight=None,...))],
         verbose=False)
```

这对网格搜索尤其重要：

```
>>> from sklearn.model_selection import GridSearchCV
>>> param_grid = dict(reduce_dim__n_components=[2, 5, 10],
...                   clf__C=[0.1, 10, 100])
>>> grid_search = GridSearchCV(pipe, param_grid=param_grid)
```

单独的步骤可以用多个参数替换，除了最后步骤，其他步骤都可以设置为 `passthrough` 来跳过

```
>>> from sklearn.linear_model import LogisticRegression
>>> param_grid = dict(reduce_dim=['passthrough', PCA(5), PCA(10)],
...                   clf=[SVC(), LogisticRegression()],
...                   clf__C=[0.1, 10, 100])
>>> grid_search = GridSearchCV(pipe, param_grid=param_grid)
```

管道的估计量可以通过索引检索：

```
>>> pipe[0]
PCA(copy=True, ...)
```

示例 ：

- Pipeline Anova SVM
- Sample pipeline for text feature extraction and evaluation
- Pipelining: chaining a PCA and a logistic regression
- Explicit feature map approximation for RBF kernels
- SVM-Anova: SVM with univariate feature selection
- Selecting dimensionality reduction with Pipeline and GridSearchCV

也可以参阅：

- 调整估计器的超参数

5.1.1.2. 注意

对管道调用 `fit` 方法的效果跟依次对每个评估器调用 `fit` 方法一样，都是 `transform` 输入并传递给下个步骤。 管道中最后一个评估器的所有方法，管道都有。例如，如果最后的评估器是一个分类器， `Pipeline` 可以当做分类器来用。如果最后一个评估器是转换器，管道也一样可以。

5.1.1.3. 缓存转换器：避免重复计算

适配转换器是很耗费计算资源的。设置了 `memory` 参数， `Pipeline` 将会在调用 `fit` 方法后缓存每个转换器。如果参数和输入数据相同，这个特征用于避免重复计算适配的转换器。典型的例子是网格搜索转换器，该转化器只要适配一次就可以多次使用。

`memory` 参数用于缓存转换器。 `memory` 可以是包含要缓存的转换器的目录的字符串或一个 `joblib.Memory` 对象：

```
>>> from tempfile import mkdtemp
>>> from shutil import rmtree
>>> from sklearn.decomposition import PCA
```

```
>>> from sklearn.svm import SVC
>>> from sklearn.pipeline import Pipeline
>>> estimators = [('reduce_dim', PCA()), ('clf', SVC())]
>>> cachedir = mkdtemp()
>>> pipe = Pipeline(estimators, memory=cachedir)
>>> pipe
Pipeline(...,
         steps=[('reduce_dim', PCA(copy=True,...)),
                ('clf', SVC(C=1.0,...))], verbose=False)
>>> # Clear the cache directory when you don't need it anymore
>>> rmtree(cachedir)
```

> 警告：缓存转换器的副作用
>
> 使用 `Pipeline` 而不开启缓存功能，还是可以通过查看原始实例的，例如：
>
> ```
> >> from sklearn.datasets import loaddigits>> digits = load_digits()>> pca1 = PCA()>> svm1 =
> SVC(gamma='scale')>> pipe = Pipeline([('reduce_dim', pca1), ('clf', svm1)])>> pipe.fit(digits.data,
> digits.target)  Pipeline(memory=None,           steps=[('reduce_dim', PCA(…)), ('clf', SVC(…))],
> verbose=False)>> # The pca instance can be inspected directly>> print(pca1.components)
> [[-1.77484909e-19  … 4.07058917e-18]]
> ```
>
> 开启缓存会在适配前触发转换器的克隆。因此，管道的转换器实例不能被直接查看。 在下面例子中， 访问 `PCA` 实例 `pca2` 将会引发 `AttributeError` 因为 `pca2` 是一个未适配的转换器。 这时应该使用属性 `named_steps` 来检查管道的评估器：
>
> ```
> >> cachedir = mkdtemp()>> pca2 = PCA()>> svm2 = SVC(gamma='scale')>> cached_pipe =
> Pipeline([('reduce_dim', pca2), ('clf', svm2)],memory=cachedir)>> cached_pipe.fit(digits.data,
> digits.target)  …  Pipeline(memory=…,            steps=[('reduce_dim', PCA(…)), ('clf', SVC(…))],
> verbose=False)>> print(cached_pipe.named_steps['reduce_dim'].components
> ) …      [[-1.77484909e-19  … 4.07058917e-18]]>> # Remove the cache directory>> rmtree(cachedir)
> ```
>
> 示例 ：
>
> - Selecting dimensionality reduction with Pipeline and GridSearchCV

5.1.2. 回归中的目标转换

`TransformedTargetRegressor` 在拟合回归模型之前对目标 `y` 进行转换。这些预测通过一个逆变换被映射回原始空间。它以预测所用的回归器为参数，将应用于目标变量的变压器为参数：

```
>>> import numpy as np
>>> from sklearn.datasets import load_boston
>>> from sklearn.compose import TransformedTargetRegressor
>>> from sklearn.preprocessing import QuantileTransformer
>>> from sklearn.linear_model import LinearRegression
>>> from sklearn.model_selection import train_test_split
>>> boston = load_boston()
>>> X = boston.data
>>> y = boston.target
>>> transformer = QuantileTransformer(output_distribution='normal')
>>> regressor = LinearRegression()
>>> regr = TransformedTargetRegressor(regressor=regressor,
```

```
...                                      transformer=transformer)
>>> X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)
>>> regr.fit(X_train, y_train)
TransformedTargetRegressor(...)
>>> print('R2 score: {0:.2f}'.format(regr.score(X_test, y_test)))
R2 score: 0.67
>>> raw_target_regr = LinearRegression().fit(X_train, y_train)
>>> print('R2 score: {0:.2f}'.format(raw_target_regr.score(X_test, y_test)))
R2 score: 0.64
```

对于简单的变换，可以传递一对函数，而不是一个Transformer对象，定义变换及其逆映射：

```
>>> def func(x):
...     return np.log(x)
>>> def inverse_func(x):
...     return np.exp(x)
```

随后，对象被创建为：

```
>>> regr = TransformedTargetRegressor(regressor=regressor,
...                                   func=func,
...                                   inverse_func=inverse_func)
>>> regr.fit(X_train, y_train)
TransformedTargetRegressor(...)
>>> print('R2 score: {0:.2f}'.format(regr.score(X_test, y_test)))
R2 score: 0.65
```

默认情况下，所提供的函数在每次匹配时都被检查为彼此的倒数。但是，可以通过将 `check_reverse` 设置为 `False` 来绕过这个检查：

```
>>> def inverse_func(x):
...     return x
>>> regr = TransformedTargetRegressor(regressor=regressor,
...                                   func=func,
...                                   inverse_func=inverse_func,
...                                   check_inverse=False)
>>> regr.fit(X_train, y_train)
TransformedTargetRegressor(...)
>>> print('R2 score: {0:.2f}'.format(regr.score(X_test, y_test)))
R2 score: -4.50
```

> 注意可以通过设置transformer或函数对func和inverse_func来触发转换。但是，同时设置这两个选项会产生错误。
>
> 例子
>
> - Effect of transforming the targets in regression model

5.1.3. FeatureUnion（特征联合）：复合特征空间

`FeatureUnion` 合并了多个转换器对象形成一个新的转换器，该转换器合并了他们的输出。一个 `FeatureUnion` 可以接收多个转换器对象。在适配期间，每个转换器都单独的和数据适配。 对于转换数据，转换器可以并发使用，且输出的样本向量被连接成更大的向量。

`FeatureUnion` 功能与 `Pipeline` 一样- 便捷性和联合参数的估计和验证。

可以结合：`FeatureUnion` 和 `Pipeline` 来创造出复杂模型。

（一个 `FeatureUnion` 没办法检查两个转换器是否会产出相同的特征。它仅仅在特征集合不相关时产生联合并确认是调用者的职责。）

5.1.3.1. 用法

一个 `FeatureUnion` 是通过一系列 `(key, value)` 键值对来构建的，其中的 `key` 给转换器指定的名字（一个绝对的字符串；他只是一个代号），`value` 是一个评估器对象：

```
>>> from sklearn.pipeline import FeatureUnion
>>> from sklearn.decomposition import PCA
>>> from sklearn.decomposition import KernelPCA
>>> estimators = [('linear_pca', PCA()), ('kernel_pca', KernelPCA())]
>>> combined = FeatureUnion(estimators)
>>> combined
FeatureUnion(n_jobs=None,
             transformer_list=[('linear_pca', PCA(copy=True,...)),
                               ('kernel_pca', KernelPCA(alpha=1.0,...))],
             transformer_weights=None, verbose=False)
```

跟管道一样，特征联合有一个精简版的构造器叫做:func:*make_union* ，该构造器不需要显式给每个组价起名字。

正如 `Pipeline` ， 单独的步骤可能用 `set_params` 替换 ，并设置为 `drop` 来跳过：

```
>>> combined.set_params(kernel_pca='drop')
...
FeatureUnion(n_jobs=None,
             transformer_list=[('linear_pca', PCA(copy=True,...)),
                               ('kernel_pca', 'drop')],
             transformer_weights=None, verbose=False)
```

> 示例 ：
>
> - Concatenating multiple feature extraction methods

5.1.4. 用于异构数据的列转换器

> 警告: compose.ColumnTransformer 还在实验中，它的 API可能会变动的。

许多数据集包含不同类型的特性，比如文本、浮点数和日期，每种类型的特征都需要单独的预处理或特征提取步骤。通常，在应用scikit-learn方法之前，最容易的是对数据进行预处理，例如 pandas。 在将数据传递给scikit-

learn之前处理数据可能会出现问题，原因如下：

- 将来自测试数据的统计信息集成到预处理程序中，使得交叉验证分数不可靠(被称为数据泄露)。 例如，在尺度变换或计算缺失值的情况下。
- 你可能想要在 `parameter search` 中包含预处理器参数。

compose.ColumnTransformer对数据的不同列执行不同的变换，该管道不存在数据泄漏，并且可以参数化。ColumnTransformer可以处理数组、稀疏矩阵和pandas DataFrames。

对每一列，都会应用一个不同的变换， 比如preprocessing或某个特定的特征抽取方法：

```
>>> import pandas as pd
>>> X = pd.DataFrame(
...     {'city': ['London', 'London', 'Paris', 'Sallisaw'],
...      'title': ["His Last Bow", "How Watson Learned the Trick",
...                "A Moveable Feast", "The Grapes of Wrath"],
...      'expert_rating': [5, 3, 4, 5],
...      'user_rating': [4, 5, 4, 3]})
```

对于这些数据，我们可能希望使用 `preprocessing.OneHotEncoder` 将 `city` 列编码为一个分类变量，同时使用 `feature_extraction.text.CountVectorizer` 来处理 `title` 列。由于我们可能会把多个特征抽取器用在同一列上， 我们给每一个变换器取一个唯一的名字，比如“city_category”和“title_bow”。默认情况下，忽略其余的ranking列(`remainder='drop'`)：

```
>>> from sklearn.compose import ColumnTransformer
>>> from sklearn.feature_extraction.text import CountVectorizer
>>> column_trans = ColumnTransformer(
...     [('city_category', CountVectorizer(analyzer=lambda x: [x]), 'city'),
...      ('title_bow', CountVectorizer(), 'title')],
...     remainder='drop')

>>> column_trans.fit(X)
ColumnTransformer(n_jobs=None, remainder='drop', sparse_threshold=0.3,
    transformer_weights=None,
    transformers=...)

>>> column_trans.get_feature_names()
...
['city_category__London', 'city_category__Paris', 'city_category__Sallisaw',
'title_bow__bow', 'title_bow__feast', 'title_bow__grapes', 'title_bow__his',
'title_bow__how', 'title_bow__last', 'title_bow__learned', 'title_bow__moveable',
'title_bow__of', 'title_bow__the', 'title_bow__trick', 'title_bow__watson',
'title_bow__wrath']

>>> column_trans.transform(X).toarray()
...
array([[1, 0, 0, 1, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0],
       [1, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 1, 1, 1, 0],
       [0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
       [0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 1, 1, 0, 0, 1]]...)
```

在上面的例子中， `CountVectorizer` 希望接受一维数组作为输入，因此列被指定为字符串(`'title'`)。然而， `preprocessing.OneHotEncoder` 就像大多数其他转换器一样，期望2D数据，因此在这种情况下，您需要将列指定为字符串列表(`['city']`)。

除了标量或单个项列表外，列选择可以指定为多个项、整数数组、片或布尔掩码的列表。如果输入是DataFrame，则字符串可以引用列，整数总是解释为位置列。

我们可以通过设置 `remainder='passthrough'` 来保留其余的ranking列。这些值被附加到转换的末尾：

```
>>> column_trans = ColumnTransformer(
...     [('city_category', OneHotEncoder(dtype='int'),['city']),
...      ('title_bow', CountVectorizer(), 'title')],
...     remainder='passthrough')

>>> column_trans.fit_transform(X)
...
array([[1, 0, 0, 1, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 5, 4],
       [1, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 1, 1, 1, 0, 3, 5],
       [0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 4, 4],
       [0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 1, 1, 0, 0, 1, 5, 3]]...)
```

可以将 `remainder` 设置为estimator来转换剩余的ranking列。转换后的值被附加到转换的末尾：

```
>>> from sklearn.preprocessing import MinMaxScaler
>>> column_trans = ColumnTransformer(
...     [('city_category', OneHotEncoder(), ['city']),
...      ('title_bow', CountVectorizer(), 'title')],
...     remainder=MinMaxScaler())

>>> column_trans.fit_transform(X)[:, -2:]
...
array([[1. , 0.5],
       [0. , 1. ],
       [0.5, 0.5],
       [1. , 0. ]])
```

函数 `make_column_transformer` 可用来更简单的创建类对象 `ColumnTransformer` 。 特别的，名字将会被自动给出。上面的例子等价于

```
>>> from sklearn.compose import make_column_transformer
>>> column_trans = make_column_transformer(
...     (OneHotEncoder(), ['city']),
...     (CountVectorizer(), 'title'),
...     remainder=MinMaxScaler())
>>> column_trans
ColumnTransformer(n_jobs=None, remainder=MinMaxScaler(copy=True, ...),
         sparse_threshold=0.3,
         transformer_weights=None,
         transformers=[('onehotencoder', ...)
```

> 示例
>
> - Column Transformer with Heterogeneous Data Sources
> - Column Transformer with Mixed Types

我们一直在努力

apachecn/sklearn-doc-zh



5.2. 特征提取

校验者：@if only翻译者：@片刻

模块 `sklearn.feature_extraction` 可用于提取符合机器学习算法支持的特征，比如文本和图片。

> 注意特征特征提取与特征选择有很大的不同：前者包括将任意数据（如文本或图像）转换为可用于机器学习的数值特征。后者是将这些特征应用到机器学习中。

5.2.1. 从字典类型加载特征

类 `DictVectorizer` 可用于将标准的Python字典（dict）对象列表的要素数组转换为 scikit-learn 估计器使用的 NumPy/SciPy 表示形式。

虽然 Python 的处理速度不是特别快，但 Python 的 `dict` 优点是使用方便，稀疏（不需要存储的特征），并且除了值之外还存储特征名称。

类 `DictVectorizer` 实现了 “one-of-K” 或 “one-hot” 编码，用于分类（也称为标称，离散）特征。分类功能是 “属性值” 对，其中该值被限制为不排序的可能性的离散列表（例如主题标识符，对象类型，标签，名称…）。

在下面的例子，”城市” 是一个分类属性，而 “温度” 是传统的数字特征：

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

>>> from sklearn.feature_extraction import DictVectorizer
>>> vec = DictVectorizer()

>>> vec.fit_transform(measurements).toarray()
array([[  1.,   0.,   0.,  33.],
 [  0.,   1.,   0.,  12.],
 [  0.,   0.,   1.,  18.]])

>>> vec.get_feature_names()
['city=Dubai', 'city=London', 'city=San Francisco', 'temperature']
```

类 `DictVectorizer` 也是对自然语言处理模型中训练序列分类器的有用的表示变换，通常通过提取围绕感兴趣的特定的词的特征窗口来工作。

例如，假设我们具有提取我们想要用作训练序列分类器（例如：块）的互补标签的部分语音（PoS）标签的第一算法。以下 dict 可以是在 “坐在垫子上的猫” 的句子，围绕 “sat” 一词提取的这样一个特征窗口：

```
>>> pos_window = [
...     {
...         'word-2': 'the',
```

```
...         'pos-2': 'DT',
...         'word-1': 'cat',
...         'pos-1': 'NN',
...         'word+1': 'on',
...         'pos+1': 'PP',
...     },
...     # in a real application one would extract many such dictionaries
... ]
```

该描述可以被矢量化为适合于呈递分类器的稀疏二维矩阵（可能在被管道 `text.TfidfTransformer` 进行归一化之后）：

```
>>> vec = DictVectorizer()
>>> pos_vectorized = vec.fit_transform(pos_window)
>>> pos_vectorized
<1x6 sparse matrix of type '<... 'numpy.float64'>'
 with 6 stored elements in Compressed Sparse ... format>
>>> pos_vectorized.toarray()
array([[ 1.,  1.,  1.,  1.,  1.,  1.]])
>>> vec.get_feature_names()
['pos+1=PP', 'pos-1=NN', 'pos-2=DT', 'word+1=on', 'word-1=cat', 'word-2=the']
```

你可以想象，如果一个文本语料库的每一个单词都提取了这样一个上下文，那么所得的矩阵将会非常宽（许多 one-hot-features），其中大部分通常将会是0。 为了使结果数据结构能够适应内存，该类 `DictVectorizer` 的 `scipy.sparse` 默认使用一个矩阵而不是一个 `numpy.ndarray` 。

5.2.2. 特征哈希（相当于一种降维技巧）

类 `FeatureHasher` 是一种高速，低内存消耗的向量化方法，它使用了特征散列技术 ，或可称为 “散列法”（hashing trick）的技术。 代替在构建训练中遇到的特征的哈希表，如向量化所做的那样 `FeatureHasher` 将哈希函数应用于特征，以便直接在样本矩阵中确定它们的列索引。 结果是以牺牲可检测性为代价，提高速度和减少内存的使用； 哈希表不记得输入特性是什么样的，没有 `inverse_transform` 办法。

由于散列函数可能导致（不相关）特征之间的冲突，因此使用带符号散列函数，并且散列值的符号确定存储在特征的输出矩阵中的值的符号。 这样，碰撞可能会抵消而不是累积错误，并且任何输出要素的值的预期平均值为零。默认情况下，此机制将使用 `alternate_sign=True` 启用，对于小型哈希表大小（ `n_features < 10000` ）特别有用。 对于大的哈希表大小，可以禁用它，以便将输出传递给估计器，如 `sklearn.naive_bayes.MultinomialNB` 或 `sklearn.feature_selection.chi2` 特征选择器，这些特征选项器可以使用非负输入。

类 `FeatureHasher` 接受映射（如 Python 的 `dict` 及其在 `collections` 模块中的变体），使用键值对 `(feature, value)` 或字符串，具体取决于构造函数参数 `input_type` 。 映射被视为 `(feature, value)` 对的列表，而单个字符串的隐含值为1，因此 `['feat1', 'feat2', 'feat3']` 被解释为 `[('feat1', 1), ('feat2', 1), ('feat3', 1)]` 。 如果单个特征在样本中多次出现，相关值将被求和（所以 `('feat', 2)` 和 `('feat', 3.5)` 变为 `('feat', 5.5)` ）。 `FeatureHasher` 的输出始终是 CSR 格式的 `scipy.sparse` 矩阵。

特征散列可以在文档分类中使用，但与 `text.CountVectorizer` 不同， `FeatureHasher` 不执行除 Unicode 或 UTF-8 编码之外的任何其他预处理； 请参阅下面的哈希技巧向量化大文本语料库，用于组合的 tokenizer/hasher。

例如，有一个词级别的自然语言处理任务，需要从 `(token, part_of_speech)` 键值对中提取特征。可以使用 Python 生成器函数来提取功能：

```
def token_features(token, part_of_speech):
    if token.isdigit():
        yield "numeric"
    else:
        yield "token={}".format(token.lower())
        yield "token,pos={},{}".format(token, part_of_speech)
    if token[0].isupper():
        yield "uppercase_initial"
    if token.isupper():
        yield "all_uppercase"
    yield "pos={}".format(part_of_speech)
```

然后， `raw_X` 为了可以传入 `FeatureHasher.transform` 可以通过如下方式构造：

```
raw_X = (token_features(tok, pos_tagger(tok)) for tok in corpus)
```

并传入一个 hasher：

```
hasher = FeatureHasher(input_type='string')
X = hasher.transform(raw_X)
```

得到一个 `scipy.sparse` 类型的矩阵 `X` 。

注意使用发生器的理解，它将懒惰引入到特征提取中：词令牌（token）只能根据需要从哈希值进行处理。

5.2.2.1. 实现细节

类 `FeatureHasher` 使用签名的 32-bit 变体的 MurmurHash3。 因此导致（并且由于限制 `scipy.sparse` ），当前支持的功能的最大数量 $2^{31} - 1$.

特征哈希的原始形式源于Weinberger et al，使用两个单独的哈希函数，h 和 ξ 分别确定特征的列索引和符号。现有的实现是基于假设：MurmurHash3的符号位与其他位独立。

由于使用简单的模数将哈希函数转换为列索引，建议使用2次幂作为 `n_features` 参数；否则特征不会均匀的分布到列中。

> 参考资料：
>
> - Kilian Weinberger, Anirban Dasgupta, John Langford, Alex Smola and Josh Attenberg (2009). 用于大规模多任务学习的特征散列. Proc. ICML.
> - MurmurHash3.

5.2.3. 文本特征提取

5.2.3.1. 话语表示

文本分析是机器学习算法的主要应用领域。 然而，原始数据，符号文字序列不能直接传递给算法，因为它们大多数要求具有固定长度的数字矩阵特征向量，而不是具有可变长度的原始文本文档。

为解决这个问题，scikit-learn提供了从文本内容中提取数字特征的最常见方法，即：

- 令牌化（**tokenizing**） 对每个可能的词令牌分成字符串并赋予整数形的id，例如通过使用空格和标点符号作为令牌分隔符。
- 统计（**counting**） 每个词令牌在文档中的出现次数。
- 标准化（**normalizing**） 在大多数的文档 / 样本中，可以减少重要的次令牌的出现次数的权重。。

在该方案中，特征和样本定义如下：

- 每个单独的令牌发生频率（标准化或不标准化）被视为一个特征。
- 给定文档中所有的令牌频率向量被看做一个多元sample样本。

因此，文本的集合可被表示为矩阵形式，每行对应一条文本，每列对应每个文本中出现的词令牌(如单个词)。

我们称向量化是将文本文档集合转换为数字集合特征向量的普通方法。 这种特殊思想（令牌化，计数和归一化）被称为 **Bag of Words** 或 “Bag of n-grams” 模型。 文档由单词出现来描述，同时完全忽略文档中单词的相对位置信息。

5.2.3.2. 稀疏

由于大多数文本文档通常只使用文本词向量全集中的一个小子集，所以得到的矩阵将具有许多特征值为零（通常大于99%）。

例如，10,000 个短文本文档（如电子邮件）的集合将使用总共100,000个独特词的大小的词汇，而每个文档将单独使用100到1000个独特的单词。

为了能够将这样的矩阵存储在存储器中，并且还可以加速代数的矩阵/向量运算，实现通常将使用诸如 `scipy.sparse` 包中的稀疏实现。

5.2.3.3. 常见 Vectorizer 使用方法

类 `CountVectorizer` 在单个类中实现了 tokenization （词语切分）和 occurrence counting （出现频数统计）：

```
>>> from sklearn.feature_extraction.text import CountVectorizer
```

这个模型有很多参数，但参数的默认初始值是相当合理的（请参阅 参考文档 了解详细信息）：

```
>>> vectorizer = CountVectorizer()
>>> vectorizer
CountVectorizer(analyzer=...'word', binary=False, decode_error=...'strict',
  dtype=<... 'numpy.int64'>, encoding=...'utf-8', input=...'content',
  lowercase=True, max_df=1.0, max_features=None, min_df=1,
  ngram_range=(1, 1), preprocessor=None, stop_words=None,
```

```
  strip_accents=None, token_pattern=...'(?u)\\b\\w\\w+\\b',
  tokenizer=None, vocabulary=None)
```

我们用它来对简约的文本语料库进行 tokenize（分词）和统计单词出现频数：

```
>>> corpus = [
...     'This is the first document.',
...     'This is the second second document.',
...     'And the third one.',
...     'Is this the first document?',
... ]
>>> X = vectorizer.fit_transform(corpus)
>>> X
<4x9 sparse matrix of type '<... 'numpy.int64'>'
 with 19 stored elements in Compressed Sparse ... format>
```

默认配置通过提取至少 2 个字母的单词来对 string 进行分词。做这一步的函数可以显式地被调用：

```
>>> analyze = vectorizer.build_analyzer()
>>> analyze("This is a text document to analyze.") == (
...     ['this', 'is', 'text', 'document', 'to', 'analyze'])
True
```

analyzer 在拟合过程中找到的每个 term（项）都会被分配一个唯一的整数索引，对应于 resulting matrix（结果矩阵）中的一列。此列的一些说明可以被检索如下：

```
>>> vectorizer.get_feature_names() == (
...     ['and', 'document', 'first', 'is', 'one',
...      'second', 'the', 'third', 'this'])
True

>>> X.toarray()
array([[0, 1, 1, 1, 0, 0, 1, 0, 1],
 [0, 1, 0, 1, 0, 2, 1, 0, 1],
 [1, 0, 0, 0, 1, 0, 1, 1, 0],
 [0, 1, 1, 1, 0, 0, 1, 0, 1]]...)
```

从 feature 名称到 column index（列索引） 的逆映射存储在 `vocabulary_` 属性中：

```
>>> vectorizer.vocabulary_.get('document')
1
```

因此，在未来对 transform 方法的调用中，在 training corpus （训练语料库）中没有看到的单词将被完全忽略：

```
>>> vectorizer.transform(['Something completely new.']).toarray()
...
array([[0, 0, 0, 0, 0, 0, 0, 0, 0]]...)
```

请注意，在前面的 corpus（语料库）中，第一个和最后一个文档具有完全相同的词，因为被编码成相同的向量。 特别是我们丢失了最后一个文件是一个疑问的形式的信息。为了防止词组顺序颠倒，除了提取一元模型 1-grams（个别词）之外，我们还可以提取 2-grams 的单词：

```
>>> bigram_vectorizer = CountVectorizer(ngram_range=(1, 2),
...                                     token_pattern=r'\b\w+\b', min_df=1)
>>> analyze = bigram_vectorizer.build_analyzer()
>>> analyze('Bi-grams are cool!') == (
...     ['bi', 'grams', 'are', 'cool', 'bi grams', 'grams are', 'are cool'])
True
```

由 vectorizer（向量化器）提取的 vocabulary（词汇）因此会变得更大，同时可以在定位模式时消除歧义：

```
>>> X_2 = bigram_vectorizer.fit_transform(corpus).toarray()
>>> X_2
...
array([[0, 0, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 1, 1, 0],
 [0, 0, 1, 0, 0, 1, 1, 0, 0, 2, 1, 1, 1, 0, 1, 0, 0, 0, 1, 1, 0],
 [1, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 1, 1, 1, 0, 0, 0],
 [0, 0, 1, 1, 1, 1, 0, 1, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 1, 0, 1]]...)
```

特别是 “Is this” 的疑问形式只出现在最后一个文档中：

```
>>> feature_index = bigram_vectorizer.vocabulary_.get('is this')
>>> X_2[:, feature_index]
array([0, 0, 0, 1]...)
```

5.2.3.3.1 使用停止词

停止词是像“and”、“the”、“him”这样的词，这些词在表示文本内容时被认为是没有信息的，可以删除它们，以避免它们被理解为预测的信号。然而，有时，类似的词对预测很有用，比如在对写作风格或性格进行分类时。

在我们提供的“英w文”停止词列表中有几个已知的问题。详情参见[NQY18]。

请慎重选择停止词列表。流行的停止词列表可能包括对某些任务(如计算机)具有高度信息性的词。

您还应该确保停止单词列表具有与矢量化器中使用的相同的预处理和标记。单词"we’ve"被 `CountVectorizer` 的默认记号分配器分割成"we"和"ve"，所以如果"we’ve"在停止词列表中，但"ve"不在，"ve"会被保留在转换后的文本中。我们的矢量化器将尝试识别和警告某些类型的不一致性。

> 参考资料
>
> - [NQY18] J. Nothman, H. Qin and R. Yurchak (2018). “Stop Word Lists in Free Open-source Software Packages”. In Proc. Workshop for NLP Open Source Software.

5.2.3.4. Tf–idf 项加权

在一个大的文本语料库中，一些单词将出现很多次（例如 “the”, “a”, “is” 是英文），因此对文档的实际内容没有什么有意义的信息。 如果我们将直接计数数据直接提供给分类器，那么这些频繁词组会掩盖住那些我们关注但很少

出现的词。

为了重新计算特征权重，并将其转化为适合分类器使用的浮点值，因此使用 `tf-idf` 变换是非常常见的。

Tf表示术语频率，而 `tf-idf` 表示术语频率乘以转制文档频率：$\text{tf-idf(t,d)} = \text{tf(t,d)} \times \text{idf(t)}$.

使用 `TfidfTransformer` 的默认设置， `TfidfTransformer(norm='l2', use_idf=True, smooth_idf=True, sublinear_tf=False)` 术语频率，一个术语在给定文档中出现的次数乘以 `idf` 组件， 计算为

$$\text{idf}(t) = log\frac{1+n_d}{1+\text{df}(d,t)} + 1$$,

其中 n_d 是文档的总数， $\text{df}(d,t)$ 是包含术语 t 的文档数。 然后，所得到的 `tf-idf` 向量通过欧几里得范数归一化：

$$v_{norm} = \frac{v}{||v||_2} = \frac{v}{\sqrt{v_1{}^2 + v_2{}^2 + \cdots + v_n{}^2}}$$.

它源于一个词权重的信息检索方式(作为搜索引擎结果的评级函数)，同时也在文档分类和聚类中表现良好。

以下部分包含进一步说明和示例，说明如何精确计算 `tf-idfs` 以及如何在 `scikit-learn` 中计算 `tf-idfs`， `TfidfTransformer` 并 `TfidfVectorizer` 与定义 `idf` 的标准教科书符号略有不同

$$\text{idf}(t) = log\frac{n_d}{1+\text{df}(d,t)}.$$

在 `TfidfTransformer` 和 `TfidfVectorizer` 中 `smooth_idf=False` ，将 “1” 计数添加到 `idf` 而不是 `idf` 的分母：

$$\text{idf}(t) = log\frac{n_d}{\text{df}(d,t)} + 1$$

该归一化由类 `TfidfTransformer` 实现：

```
>>> from sklearn.feature_extraction.text import TfidfTransformer
>>> transformer = TfidfTransformer(smooth_idf=False)
>>> transformer
TfidfTransformer(norm=...'l2', smooth_idf=False, sublinear_tf=False,
 use_idf=True)
```

有关所有参数的详细信息，请参阅 参考文档。

让我们以下方的词频为例。第一个次在任何时间都是100%出现，因此不是很有重要。另外两个特征只占不到50%的比例，因此可能更具有代表性：

```
>>> counts = [[3, 0, 1],
...           [2, 0, 0],
...           [3, 0, 0],
...           [4, 0, 0],
```

```
...           [3, 2, 0],
...           [3, 0, 2]]
...
>>> tfidf = transformer.fit_transform(counts)
>>> tfidf
<6x3 sparse matrix of type '<... 'numpy.float64'>'
    with 9 stored elements in Compressed Sparse ... format>

>>> tfidf.toarray()
array([[0.81940995, 0.        , 0.57320793],
       [1.        , 0.        , 0.        ],
       [1.        , 0.        , 0.        ],
       [1.        , 0.        , 0.        ],
       [0.47330339, 0.88089948, 0.        ],
       [0.58149261, 0.        , 0.81355169]])
```

每行都被正则化，使其适应欧几里得标准：

$$v_{norm} = \frac{v}{||v||_2} = \frac{v}{\sqrt{v_1{}^2 + v_2{}^2 + \cdots + v_n{}^2}}$$

例如，我们可以计算 `计数` 数组中第一个文档中第一个项的 `tf-idf` ，如下所示：

$$n_{d,\text{term1}} = 6$$

$$\text{df}(d,t)_{\text{term1}} = 6$$

$$\text{idf}(d,t)_{\text{term1}} = log\frac{n_d}{\text{df}(d,t)} + 1 = log(1) + 1 = 1$$

$$\text{tf-idf}_{\text{term1}} = \text{tf} \times \text{idf} = 3 \times 1 = 3$$

现在，如果我们对文档中剩下的2个术语重复这个计算，我们得到：

$$\text{tf-idf}_{\text{term2}} = 0 \times log(6/1) + 1 = 0$$

$$\text{tf-idf}_{\text{term3}} = 1 \times log(6/2) + 1 \approx 2.0986$$

和原始 `tf-idfs` 的向量：

$$\text{tf-idf}_raw = [3, 0, 2.0986].$$

然后，应用欧几里德（L2）规范，我们获得文档1的以下 `tf-idfs`：

$$\frac{[3, 0, 2.0986]}{\sqrt{\left(3^2 + 0^2 + 2.0986^2\right)}} = [0.819, 0, 0.573].$$

此外，默认参数 `smooth_idf=True` 将 “1” 添加到分子和分母，就好像一个额外的文档被看到一样包含集合中的每个术语，这样可以避免零分割：

$$\text{idf}(t) = log\frac{1+n_d}{1+\text{df}(d,t)} + 1$$

使用此修改，文档1中第三项的 tf-idf 更改为 1.8473:

$$\text{tf-idf}_{\text{term3}} = 1 \times log(7/3) + 1 \approx 1.8473$$

而 L2 标准化的 tf-idf 变为

$$\frac{[3,0,1.8473]}{\sqrt{\left(3^2+0^2+1.8473^2\right)}} = [0.8515, 0, 0.5243]$$

:

```
>>> transformer = TfidfTransformer()
>>> transformer.fit_transform(counts).toarray()
array([[0.85151335, 0.        , 0.52433293],
       [1.        , 0.        , 0.        ],
       [1.        , 0.        , 0.        ],
       [1.        , 0.        , 0.        ],
       [0.55422893, 0.83236428, 0.        ],
       [0.63035731, 0.        , 0.77630514]])
```

通过 `拟合` 方法调用计算的每个特征的权重存储在模型属性中:

```
>>> transformer.idf_
array([1. ..., 2.25..., 1.84...])
```

由于 tf-idf 经常用于文本特征，所以还有一个类 `TfidfVectorizer` ，它将 `CountVectorizer` 和 `TfidfTransformer` 的所有选项组合在一个单例模型中:

```
>>> from sklearn.feature_extraction.text import TfidfVectorizer
>>> vectorizer = TfidfVectorizer()
>>> vectorizer.fit_transform(corpus)
...
<4x9 sparse matrix of type '<... 'numpy.float64'>'
 with 19 stored elements in Compressed Sparse ... format>
```

虽然tf-idf标准化通常非常有用，但是可能有一种情况是二元变量显示会提供更好的特征。 这可以使用类 `CountVectorizer` 的 `二进制` 参数来实现。 特别地，一些估计器，诸如 伯努利朴素贝叶斯 显式的使用离散的布尔随机变量。 而且，非常短的文本很可能影响 tf-idf 值，而二进制出现信息更稳定。

通常情况下，调整特征提取参数的最佳方法是使用基于网格搜索的交叉验证，例如通过将特征提取器与分类器进行流水线化:

> - 用于文本特征提取和评估的样本管道 Sample pipeline for text feature extraction and evaluation

5.2.3.5. 解码文本文件

文本由字符组成，但文件由字节组成。字节转化成字符依照一定的编码(encoding)方式。 为了在Python中的使用文本文档，这些字节必须被 解码 为 Unicode 的字符集。 常用的编码方式有 ASCII，Latin-1（西欧），KOI8-R（俄语）和通用编码 UTF-8 和 UTF-16。还有许多其他的编码存在

> 注意
>
> 编码也可以称为 ‘字符集’，但是这个术语不太准确：单个字符集可能存在多个编码。

scikit-learn 中的文本提取器知道如何解码文本文件， 但只有当您告诉他们文件的编码的情况下才行， `CountVectorizer` 才需要一个 `encoding` 参数。 对于现代文本文件，正确的编码可能是 UTF-8，因此它也是默认解码方式（ `encoding="utf-8"` ）.

如果正在加载的文本不是使用UTF-8进行编码，则会得到 `UnicodeDecodeError` . 矢量化的方式可以通过设定 `decode_error` 参数设置为 `"ignore"` 或 `"replace"` 来避免抛出解码错误。 有关详细信息，请参阅Python函数 `bytes.decode` 的文档（在Python提示符下键入 `help(bytes.decode)` ）。

如果您在解码文本时遇到问题，请尝试以下操作：

- 了解文本的实际编码方式。该文件可能带有标题或 README，告诉您编码，或者可能有一些标准编码，您可以根据文本的来源来推断编码方式。
- 您可能可以使用 UNIX 命令 `file` 找出它一般使用什么样的编码。 Python `chardet` 模块附带一个名为 `chardetect.py` 的脚本，它会猜测具体的编码，尽管你不能依靠它的猜测是正确的。
- 你可以尝试 UTF-8 并忽略错误。您可以使用 `bytes.decode(errors='replace')` 对字节字符串进行解码，用无意义字符替换所有解码错误，或在向量化器中设置 `decode_error='replace'` . 这可能会损坏您的功能的有用性。
- 真实文本可能来自各种使用不同编码的来源，或者甚至以与编码的编码不同的编码进行粗略解码。这在从 Web 检索的文本中是常见的。Python 包 ftfy 可以自动排序一些解码错误类，所以您可以尝试将未知文本解码为 `latin-1` ，然后使用 `ftfy` 修复错误。
- 如果文本的编码的混合，那么它很难整理分类（20个新闻组数据集的情况），您可以把它们回到简单的单字节编码，如 `latin-1` 。某些文本可能显示不正确，但至少相同的字节序列将始终代表相同的功能。

例如，以下代码段使用 `chardet` （未附带 scikit-learn，必须单独安装）来计算出编码方式。然后，它将文本向量化并打印学习的词汇（特征）。输出在下方给出。

```
>>> import chardet
>>> text1 = b"Sei mir gegr\xc3\xbc\xc3\x9ft mein Sauerkraut"
>>> text2 = b"holdselig sind deine Ger\xfcche"
>>> text3 = b"\xff\xfeA\x00u\x00f\x00 \x00F\x00l\x00\xfc\x00g\x00e\x00l\x00n\x00 \x00d\x00e\x00s\x00
\x00G\x00e\x00s\x00a\x00n\x00g\x00e\x00s\x00,\x00
\x00H\x00e\x00r\x00z\x00l\x00i\x00e\x00b\x00c\x00h\x00e\x00n\x00,\x00 \x00t\x00r\x00a\x00g\x00
\x00i\x00c\x00h\x00 \x00d\x00i\x00c\x00h\x00 \x00f\x00o\x00r\x00t\x00"
>>> decoded = [x.decode(chardet.detect(x)['encoding'])
...            for x in (text1, text2, text3)]
>>> v = CountVectorizer().fit(decoded).vocabulary_
>>> for term in v: print(v)
```

（根据 `chardet` 的版本，可能会返回第一个值错误的结果。） 有关 Unicode 和字符编码的一般介绍，请参阅 Joel Spolsky的 绝对最低要求每个软件开发人员必须了解 Unicode.

5.2.3.6. 应用和实例

词汇表达方式相当简单，但在实践中却非常有用。

特别是在 监督学习的设置 中，它能够把快速和可扩展的线性模型组合来训练 文档分类器， 例如：

> - 使用稀疏特征对文本文档进行分类 Classification of text documents using sparse features

在 无监督的设置 中，可以通过应用诸如 K-means 的聚类算法来将相似文档分组在一起：

> - 使用k-means聚类文本文档 Clustering text documents using k-means

最后，通过松弛聚类的约束条件，可以通过使用非负矩阵分解（ 非负矩阵分解(NMF 或 NNMF) 或NNMF）来发现语料库的主要主题：

> - 主题提取与非负矩阵分解和潜在Dirichlet分配 Topic extraction with Non-negative Matrix Factorization and Latent Dirichlet Allocation

5.2.3.7. 词语表示的限制

一组单词（什么是单词）无法捕获短语和多字表达，有效地忽略任何单词顺序依赖。另外，这个单词模型不包含潜在的拼写错误或词汇导出。

N-grams 可以帮助我们！而不是构建一个简单的unigrams集合 (n=1)，可能更喜欢一组二进制 (n=2)，其中计算连续字对。

还可以考虑一个字符 n-grams 的集合，这是一种对拼写错误和派生有弹性的表示。

例如，假设我们正在处理两个文档的语料库： `['words', 'wprds']` 。第二个文件包含 'words' 一词的拼写错误。一个简单的单词表示将把这两个视为非常不同的文档，两个可能的特征都是不同的。 然而，一个字符 2-gram 的表示可以找到匹配的文档中的8个特征中的4个，这可能有助于优选的分类器更好地决定：

```
>>> ngram_vectorizer = CountVectorizer(analyzer='char_wb', ngram_range=(2, 2))
>>> counts = ngram_vectorizer.fit_transform(['words', 'wprds'])
>>> ngram_vectorizer.get_feature_names() == (
...     [' w', 'ds', 'or', 'pr', 'rd', 's ', 'wo', 'wp'])
True
>>> counts.toarray().astype(int)
array([[1, 1, 1, 0, 1, 1, 1, 0],
 [1, 1, 0, 1, 1, 1, 0, 1]])
```

在上面的例子中，使用 `'char_wb'` 分析器'，它只能从字边界内的字符（每侧填充空格）创建 n-grams。 `'char'` 分析器可以创建跨越单词的 n-grams：

```
>>> ngram_vectorizer = CountVectorizer(analyzer='char_wb', ngram_range=(5, 5))
>>> ngram_vectorizer.fit_transform(['jumpy fox'])
...
<1x4 sparse matrix of type '<... 'numpy.int64'>'
 with 4 stored elements in Compressed Sparse ... format>
>>> ngram_vectorizer.get_feature_names() == (
...     [' fox ', ' jump', 'jumpy', 'umpy '])
True
```

```
>>> ngram_vectorizer = CountVectorizer(analyzer='char', ngram_range=(5, 5))
>>> ngram_vectorizer.fit_transform(['jumpy fox'])
...
<1x5 sparse matrix of type '<... 'numpy.int64'>'
 with 5 stored elements in Compressed Sparse ... format>
>>> ngram_vectorizer.get_feature_names() == (
...     ['jumpy', 'mpy f', 'py fo', 'umpy ', 'y fox'])
True
```

对于使用白色空格进行单词分离的语言，对于语言边界感知变体 `char_wb` 尤其有趣，因为在这种情况下，它会产生比原始 `char` 变体显着更少的噪音特征。 对于这样的语言，它可以增加使用这些特征训练的分类器的预测精度和收敛速度，同时保持关于拼写错误和词导出的稳健性。

虽然可以通过提取 n-grams 而不是单独的单词来保存一些本地定位信息，但是包含 n-grams 的单词和袋子可以破坏文档的大部分内部结构，因此破坏了该内部结构的大部分含义。

为了处理自然语言理解的更广泛的任务，因此应考虑到句子和段落的地方结构。因此，许多这样的模型将被称为 “结构化输出” 问题，这些问题目前不在 scikit-learn 的范围之内。

5.2.3.8. 用哈希技巧矢量化大文本语料库

上述向量化方案是简单的，但是它存在 从字符串令牌到整数特征索引的内存映射 （ `vocabulary_` 属性），在处理大型数据集时会引起几个问题 ：

- 语料库越大，词汇量越大，使用的内存也越大.
- 拟合（fitting）需要根据原始数据集的大小等比例分配中间数据结构的大小.
- 构建词映射需要完整的传递数据集，因此不可能以严格在线的方式拟合文本分类器.
- pickling和un-pickling vocabulary 很大的向量器会非常慢（通常比pickling/un-pickling单纯数据的结构，比如同等大小的Numpy数组）.
- 将向量化任务分隔成并行的子任务很不容易实现，因为 `vocabulary_` 属性要共享状态有一个细颗粒度的同步障碍：从标记字符串中映射特征索引与每个标记的首次出现顺序是独立的，因此应该被共享，在这点上并行worker的性能收到了损害，使他们比串行更慢。

通过组合由 `sklearn.feature_extraction.FeatureHasher` 类实现的 “散列技巧”（特征哈希（相当于一种降维技巧））和 `CountVectorizer` 的文本预处理和标记化功能，可以克服这些限制。

这种组合是在 `HashingVectorizer` 中实现的，该类是与 `CountVectorizer` 大部分 API 兼容的变压器类。`HashingVectorizer` 是无状态的，这意味着您不需要 `fit` 它：

```
>>> from sklearn.feature_extraction.text import HashingVectorizer
>>> hv = HashingVectorizer(n_features=10)
>>> hv.transform(corpus)
...
<4x10 sparse matrix of type '<... 'numpy.float64'>'
 with 16 stored elements in Compressed Sparse ... format>
```

你可以看到从向量输出中抽取了16个非0特征标记：与之前由CountVectorizer在同一个样本语料库抽取的19个非0特征要少。差异来自哈希方法的冲突，因为较低的n_features参数的值。

在真实世界的环境下，n_features参数可以使用默认值2 **20**（将近**100**万可能的特征）。如果内存或者下游模型的大小是一个问题，那么选择一个较小的值比如**2** 18可能有一些帮助，而不需要为典型的文本分类任务引入太多额外的冲突。

注意维度并不影响CPU的算法训练时间，这部分是在操作CSR指标（LinearSVC(dual=True), Perceptron, SGDClassifier, PassiveAggressive），但是，它对CSC matrices (LinearSVC(dual=False), Lasso(), etc)算法有效。

让我们再次尝试使用默认设置：

```
>>> hv = HashingVectorizer()
>>> hv.transform(corpus)
...
<4x1048576 sparse matrix of type '<... 'numpy.float64'>'
 with 19 stored elements in Compressed Sparse ... format>
```

冲突没有再出现，但是，代价是输出空间的维度值非常大。当然，这里使用的19词以外的其他词之前仍会有冲突。

类 `HashingVectorizer` 还具有以下限制：

- 不能反转模型（没有inverse_transform方法），也无法访问原始的字符串表征，因为，进行mapping的哈希方法是单向本性。
- 没有提供了IDF权重，因为这需要在模型中引入状态。如果需要的话，可以在管道中添加 `TfidfTransformer` 。

5.2.3.9. 使用 HashingVectorizer 执行外核缩放

使用 `HashingVectorizer` 的一个有趣的开发是执行外核 out-of-core 缩放的能力。 这意味着我们可以从无法放入电脑主内存的数据中进行学习。

实现核外扩展的一个策略是将数据以流的方式以一小批提交给评估器。每批的向量化都是用HashingVectorizer这样来保证评估器的输入空间的维度是相等的。因此任何时间使用的内存数都限定在小频次的大小。 尽管用这种方法可以处理的数据没有限制，但是从实用角度学习时间受到想要在这个任务上花费的CPU时间的限制。

对于文本分类任务中的外核缩放的完整示例，请参阅文本文档的外核分类 Out-of-core classification of text documents.

5.2.3.10. 自定义矢量化器类

通过将可调用传递给向量化程序构造函数可以定制行为：

```
>>> def my_tokenizer(s):
...     return s.split()
...
>>> vectorizer = CountVectorizer(tokenizer=my_tokenizer)
>>> vectorizer.build_analyzer()(u"Some... punctuation!") == (
...     ['some...', 'punctuation!'])
True
```

特别是我们命名：

- `预处理器`： 可以将整个文档作为输入（作为单个字符串）的可调用，并返回文档的可能转换的版本，仍然是整个字符串。这可以用于删除HTML标签，小写整个文档等。
- `tokenizer`： 一个可从预处理器接收输出并将其分成标记的可调用函数，然后返回这些列表。
- `分析器`： 一个可替代预处理程序和标记器的可调用程序。默认分析仪都会调用预处理器和刻录机，但是自定义分析仪将会跳过这个。 N-gram提取和停止字过滤在分析器级进行，因此定制分析器可能必须重现这些步骤。

（Lucene 用户可能会识别这些名称，但请注意，scikit-learn 概念可能无法一对一映射到 Lucene 概念上。）

为了使预处理器，标记器和分析器了解模型参数，可以从类派生并覆盖 `build_preprocessor` ， `build_tokenizer` 和 `build_analyzer` 工厂方法，而不是传递自定义函数。

一些提示和技巧：

- 如果文档由外部包进行预先标记，则将它们存储在文件（或字符串）中，令牌由空格分隔，并通过 `analyzer=str.split`
- Fancy 令牌级分析，如词干，词法，复合分割，基于词性的过滤等不包括在 scikit-learn 代码库中，但可以通过定制分词器或分析器来添加。这是一个 `CountVectorizer` ，使用 NLTK 的 tokenizer 和 lemmatizer：

```
>>> from nltk import word_tokenize
>>> from nltk.stem import WordNetLemmatizer
>>> class LemmaTokenizer(object):
...     def __init__(self):
...         self.wnl = WordNetLemmatizer()
...     def __call__(self, doc):
...         return [self.wnl.lemmatize(t) for t in word_tokenize(doc)]
...
>>> vect = CountVectorizer(tokenizer=LemmaTokenizer())
```

（请注意，这不会过滤标点符号。）例如，以下例子将英国的一些拼写变成美国拼写：：

```
>>> import re
>>> def to_british(tokens):
...     for t in tokens:
...         t = re.sub(r"(...)our$", r"\1or", t)
...         t = re.sub(r"([bt])re$", r"\1er", t)
...         t = re.sub(r"([iy])s(e$|ing|ation)", r"\1z\2", t)
...         t = re.sub(r"ogue$", "og", t)
...         yield t
...
>>> class CustomVectorizer(CountVectorizer):
...     def build_tokenizer(self):
...         tokenize = super().build_tokenizer()
...         return lambda doc: list(to_british(tokenize(doc)))
...
>>> print(CustomVectorizer().build_analyzer()(u"color colour"))
[...'color', ...'color']
```

用于其他样式的预处理；例子包括 stemming，lemmatization，或 normalizing numerical tokens，后者说明如下：

- Biclustering documents with the Spectral Co-clustering algorithm

在处理不使用显式字分隔符（例如空格）的亚洲语言时，自定义向量化器也是有用的。

5.2.4. 图像特征提取

5.2.4.1. 补丁提取

`extract_patches_2d` 函数从存储为二维数组的图像或沿着第三轴的颜色信息三维提取修补程序。 要从其所有补丁重建图像，请使用 `reconstruct_from_patches_2d` 。例如让我们使用3个彩色通道（例如 RGB 格式）生成一个 4x4 像素的图像：

```
>>> import numpy as np
>>> from sklearn.feature_extraction import image

>>> one_image = np.arange(4 * 4 * 3).reshape((4, 4, 3))
>>> one_image[:, :, 0]  # R channel of a fake RGB picture
array([[ 0,  3,  6,  9],
 [12, 15, 18, 21],
 [24, 27, 30, 33],
 [36, 39, 42, 45]])

>>> patches = image.extract_patches_2d(one_image, (2, 2), max_patches=2,
...     random_state=0)
>>> patches.shape
(2, 2, 2, 3)
>>> patches[:, :, :, 0]
array([[[ 0,  3],
 [12, 15]],

 [[15, 18],
 [27, 30]]])
>>> patches = image.extract_patches_2d(one_image, (2, 2))
>>> patches.shape
(9, 2, 2, 3)
>>> patches[4, :, :, 0]
array([[15, 18],
 [27, 30]])
```

现在让我们尝试通过在重叠区域进行平均来从补丁重建原始图像：

```
>>> reconstructed = image.reconstruct_from_patches_2d(patches, (4, 4, 3))
>>> np.testing.assert_array_equal(one_image, reconstructed)
```

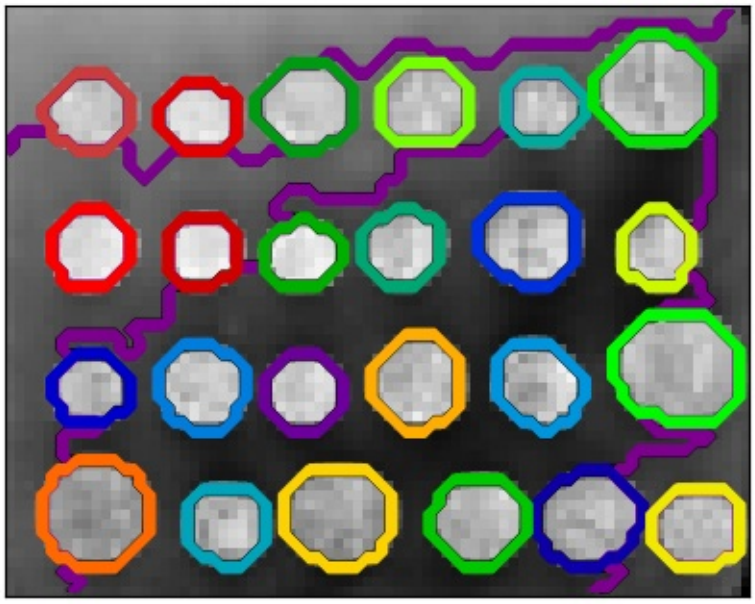
在 `PatchExtractor` 以同样的方式类作品 `extract_patches_2d` ，只是它支持多种图像作为输入。它被实现为一个估计器，因此它可以在管道中使用。看到：

```
>>> five_images = np.arange(5 * 4 * 4 * 3).reshape(5, 4, 4, 3)
>>> patches = image.PatchExtractor((2, 2)).transform(five_images)
```

```
>>> patches.shape
(45, 2, 2, 3)
```

5.2.4.2. 图像的连接图

scikit-learn 中的几个估计可以使用特征或样本之间的连接信息。 例如，Ward聚类（层次聚类 层次聚类 ）可以聚集在一起，只有图像的相邻像素，从而形成连续的斑块：



为此，估计器使用 ‘连接性’ 矩阵，给出连接的样本。

该函数 `img_to_graph` 从2D或3D图像返回这样一个矩阵。类似地， `grid_to_graph` 为给定这些图像的形状的图像构建连接矩阵。

这些矩阵可用于在使用连接信息的估计器中强加连接，如 Ward 聚类（层次聚类 层次聚类 ），而且还要构建预计算的内核或相似矩阵。

> 示例
>
> - A demo of structured Ward hierarchical clustering on a raccoon face image
> - Spectral clustering for image segmentation
> - Feature agglomeration vs. univariate selection

我们一直在努力

apachecn/sklearn-doc-zh



5.3 预处理数据

校验者：@if only翻译者：@Trembleguy

`sklearn.preprocessing` 包提供了几个常见的实用功能和变换器类型，用来将原始特征向量更改为更适合机器学习模型的形式。

一般来说，机器学习算法受益于数据集的标准化。如果数据集中存在一些离群值，那么稳定的缩放或转换更合适。不同缩放、转换以及归一在一个包含边缘离群值的数据集中的表现在 `Compare the effect of different scalers on data with outliers` 中有着重说明。

5.3.1 标准化，也称去均值和方差按比例缩放

数据集的 标准化 对scikit-learn中实现的大多数机器学习算法来说是 常见的要求 。如果个别特征或多或少看起来不是很像标准正态分布(具有零均值和单位方差)，那么它们的表现力可能会较差。

在实际情况中,我们经常忽略特征的分布形状，直接经过去均值来对某个特征进行中心化，再通过除以非常量特征(non-constant features)的标准差进行缩放。

例如，在机器学习算法的目标函数(例如SVM的RBF内核或线性模型的l1和l2正则化)，许多学习算法中目标函数的基础都是假设所有的特征都是零均值并且具有同一阶数上的方差。如果某个特征的方差比其他特征大几个数量级，那么它就会在学习算法中占据主导位置，导致学习器并不能像我们说期望的那样，从其他特征中学习。

函数 `scale` 为数组形状的数据集的标准化提供了一个快捷实现：

```
>>> X_train = np.array([[ 1., -1.,  2.],
...                     [ 2.,  0.,  0.],
...                     [ 0.,  1., -1.]])
...
>>> min_max_scaler = preprocessing.MinMaxScaler()
>>> X_train_minmax = min_max_scaler.fit_transform(X_train)
>>> X_train_minmax
array([[0.5       , 0.        , 1.        ],
       [1.        , 0.5       , 0.33333333],
       [0.        , 1.        , 0.        ]])
```

经过缩放后的数据具有零均值以及标准方差：

```
>>> X_scaled.mean(axis=0)
array([ 0.,  0.,  0.])

>>> X_scaled.std(axis=0)
array([ 1.,  1.,  1.])
```

`预处理` 模块还提供了一个实用类 `StandardScaler` ，它实现了转化器的API来计算训练集上的平均值和标准偏差，以便以后能够在测试集上重新应用相同的变换。因此，这个类适用于 `sklearn.pipeline.Pipeline` 的早期步骤：

```
>>> scaler = preprocessing.StandardScaler().fit(X_train)
>>> scaler
StandardScaler(copy=True, with_mean=True, with_std=True)

>>> scaler.mean_
array([ 1. ...,  0. ...,  0.33...])

>>> scaler.scale_
array([ 0.81...,  0.81...,  1.24...])

>>> scaler.transform(X_train)
array([[ 0.  ..., -1.22...,  1.33...],
 [ 1.22...,  0.  ..., -0.26...],
 [-1.22...,  1.22..., -1.06...]])
```

缩放类对象可以在新的数据上实现和训练集相同缩放操作：

```
>>> X_test = [[-1., 1., 0.]]
>>> scaler.transform(X_test)
array([[-2.44...,  1.22..., -0.26...]])
```

你也可以通过在构造函数 :class:StandardScaler 中传入参数 `with_mean=False` 或者 `with_std=False` 来取消中心化或缩放操作。

5.3.1.1 将特征缩放至特定范围内

一种标准化是将特征缩放到给定的最小值和最大值之间，通常在零和一之间，或者也可以将每个特征的最大绝对值转换至单位大小。可以分别使用 `MinMaxScaler` 和 `MaxAbsScaler` 实现。

使用这种缩放的目的包括实现特征极小方差的鲁棒性以及在稀疏矩阵中保留零元素。

以下是一个将简单的数据矩阵缩放到 `[0, 1]` 的例子：

```
>>> X_train = np.array([[ 1., -1.,  2.],
...                     [ 2.,  0.,  0.],
...                     [ 0.,  1., -1.]])
...
>>> min_max_scaler = preprocessing.MinMaxScaler()
>>> X_train_minmax = min_max_scaler.fit_transform(X_train)
>>> X_train_minmax
array([[ 0.5       ,  0.        ,  1.        ],
 [ 1.        ,  0.5       ,  0.33333333],
 [ 0.        ,  1.        ,  0.        ]])
```

同样的转换实例可以被用与在训练过程中不可见的测试数据：实现和训练数据一致的缩放和移位操作：

```
>>> X_test = np.array([[ -3., -1.,  4.]])
>>> X_test_minmax = min_max_scaler.transform(X_test)
>>> X_test_minmax
array([[-1.5       ,  0.        ,  1.66666667]])
```

可以检查缩放器（scaler）属性，来观察在训练集中学习到的转换操作的基本性质：

```
>>> min_max_scaler.scale_
array([ 0.5       ,  0.5       ,  0.33...])

>>> min_max_scaler.min_
array([ 0.        ,  0.5       ,  0.33...])
```

如果给 `MinMaxScaler` 提供一个明确的 `feature_range=(min, max)` ，完整的公式是：

```
X_std = (X - X.min(axis=0)) / (X.max(axis=0) - X.min(axis=0))

X_scaled = X_std * (max - min) + min
```

类 `MaxAbsScaler` 的工作原理非常相似，但是它只通过除以每个特征的最大值将训练数据特征缩放至 `[-1, 1]` 范围内，这就意味着，训练数据应该是已经零中心化或者是稀疏数据。 例子：:用先前例子的数据实现最大绝对值缩放操作。

以下是使用上例中数据运用这个缩放器的例子：

```
>>> X_train = np.array([[ 1., -1.,  2.],
...                     [ 2.,  0.,  0.],
...                     [ 0.,  1., -1.]])
...
>>> max_abs_scaler = preprocessing.MaxAbsScaler()
>>> X_train_maxabs = max_abs_scaler.fit_transform(X_train)
>>> X_train_maxabs                # doctest +NORMALIZE_WHITESPACE^
array([[ 0.5, -1. ,  1. ],
 [ 1. ,  0. ,  0. ],
 [ 0. ,  1. , -0.5]])
>>> X_test = np.array([[ -3., -1.,  4.]])
>>> X_test_maxabs = max_abs_scaler.transform(X_test)
>>> X_test_maxabs
array([[-1.5, -1. ,  2. ]])
>>> max_abs_scaler.scale_
array([ 2.,  1.,  2.])
```

在 `scale` 模块中进一步提供了方便的功能。当你不想创建对象时，可以使用如 `minmax_scale` 以及 `maxabs_scale` 。

5.3.1.2 缩放稀疏（矩阵）数据

中心化稀疏(矩阵)数据会破坏数据的稀疏结构，因此很少有一个比较明智的实现方式。但是缩放稀疏输入是有意义的，尤其是当几个特征在不同的量级范围时。

`MaxAbsScaler` 以及 `maxabs_scale` 是专为缩放数据而设计的，并且是缩放数据的推荐方法。但是， `scale` 和 `StandardScaler` 也能够接受 `scipy.sparse` 作为输入，只要参数 `with_mean=False` 被准确传入它的构造器。否

则会出现 `ValueError` 的错误，因为默认的中心化会破坏稀疏性，并且经常会因为分配过多的内存而使执行崩溃。 `RobustScaler` 不能适应稀疏输入，但你可以在稀疏输入使用 `transform` 方法。

注意，缩放器同时接受压缩的稀疏行和稀疏列(参见 `scipy.sparse.csr_matrix` 以及 `scipy.sparse.csc_matrix`)。任何其他稀疏输入将会 转化为压缩稀疏行表示 。为了避免不必要的内存复制，建议在上游(早期)选择CSR或CSC表示。

最后，如果已经中心化的数据并不是很大，使用 `toarray` 方法将输入的稀疏矩阵显式转换为数组是另一种选择。

5.3.1.3 缩放有离群值的数据

如果你的数据包含许多异常值，使用均值和方差缩放可能并不是一个很好的选择。这种情况下，你可以使用 `robust_scale` 以及 `RobustScaler` 作为替代品。它们对你的数据的中心和范围使用更有鲁棒性的估计。

> 参考资料：
>
> 更多关于中心化和缩放数据的重要性讨论在此FAQ中提及：Should I normalize/standardize/rescale the data?
>
> **Scaling vs Whitening**
>
> 有时候独立地中心化和缩放数据是不够的，因为下游的机器学习模型能够对特征之间的线性依赖做出一些假设(这对模型的学习过程来说是不利的)。
>
> 要解决这个问题，你可以使用 `sklearn.decomposition.PCA` 并指定参数 `whiten=True` 来更多移除特征间的线性关联。
>
> 在回归中缩放目标变量
>
> 上面介绍过的所有函数(`scale` , `minmax_scale` , `maxabs_scale` , 和 `robust_scale`) 都可以直接处理一维数组。

5.3.1.4 核矩阵的中心化

如果你有一个核矩阵 K ，它计算由函数 phi 定义的特征空间的点积，那么一个 `KernelCenterer` 类能够转化这个核矩阵，通过移除特征空间的平均值，使它包含由函数 phi 定义的内部产物。

5.3.2 非线性转换

有两种类型的转换是可用的：分位数转换和幂函数转换。分位数和幂变换都基于特征的单调变换，从而保持了每个特征值的秩。

分位数变换根据公式$G^{-1}(F(X))$将所有特征置于相同的期望分布中，其中F是特征的累积分布函数，G^{-1}是期望输出值分布G的分位数函数．这个公式基于以下两个事实：

- 如果Ｘ是具有连续累积分布函数Ｆ的随机变量，那么Ｆ(Ｘ)均匀分布在[０， １]
- 如果Ｕ是在[０， １]上的随机分布，那么G-1(Ｕ)有分布Ｇ．通过执行秩变换，分位数变换平滑了异常分布，并且比缩放方法受异常值的影响更小。但是它的确使特征间及特征内的关联和距离失真了。

幂变换则是一组参数变换，其目的是将数据从任意分布映射到接近高斯分布的位置。

5.3.2.1 映射到均匀分布

`QuantileTransformer` 类以及 `quantile_transform` 函数提供了一个基于分位数函数的无参数转换，将数据映射到了零到一的均匀分布上：

```
>>> from sklearn.datasets import load_iris
>>> from sklearn.model_selection import train_test_split
>>> iris = load_iris()
>>> X, y = iris.data, iris.target
>>> X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)
>>> quantile_transformer = preprocessing.QuantileTransformer(random_state=0)
>>> X_train_trans = quantile_transformer.fit_transform(X_train)
>>> X_test_trans = quantile_transformer.transform(X_test)
>>> np.percentile(X_train[:, 0], [0, 25, 50, 75, 100])
array([ 4.3,  5.1,  5.8,  6.5,  7.9])
```

这个特征是萼片的厘米单位的长度。一旦应用分位数转换，这些元素就接近于之前定义的百分位数：

```
>>> np.percentile(X_train_trans[:, 0], [0, 25, 50, 75, 100])
...
array([ 0.00... ,  0.24...,  0.49...,  0.73...,  0.99... ])
```

这可以在具有类似形式的独立测试集上确认：

```
>>> np.percentile(X_test[:, 0], [0, 25, 50, 75, 100])
...
array([ 4.4  ,  5.125,  5.75 ,  6.175,  7.3  ])
>>> np.percentile(X_test_trans[:, 0], [0, 25, 50, 75, 100])
...
array([ 0.01...,  0.25...,  0.46...,  0.60... ,  0.94...])
```

5.3.2.2 映射到高斯分布

在许多建模场景中，需要数据集中的特征的正态化。幂变换是一类参数化的单调变换， 其目的是将数据从任何分布映射到尽可能接近高斯分布，以便稳定方差和最小化偏斜。

类 PowerTransformer 目前提供两个这样的幂变换，`Yeo-Johnson transform` 和 `the Box-Cox transform` 。

Yeo-Johnson transform:

$$x_i^{(\lambda)} = \begin{cases} [(x_i+1)^\lambda - 1]/\lambda & \text{if } \lambda \neq 0, x_i \geq 0, \\ \ln(x_i) + 1 & \text{if } \lambda = 0, x_i \geq 0 \\ -[(-x_i+1)^{2-\lambda} - 1]/(2-\lambda) & \text{if } \lambda \neq 2, x_i < 0, \\ -\ln(-x_i+1) & \text{if } \lambda = 2, x_i < 0 \end{cases}$$

Box-Cox transform:

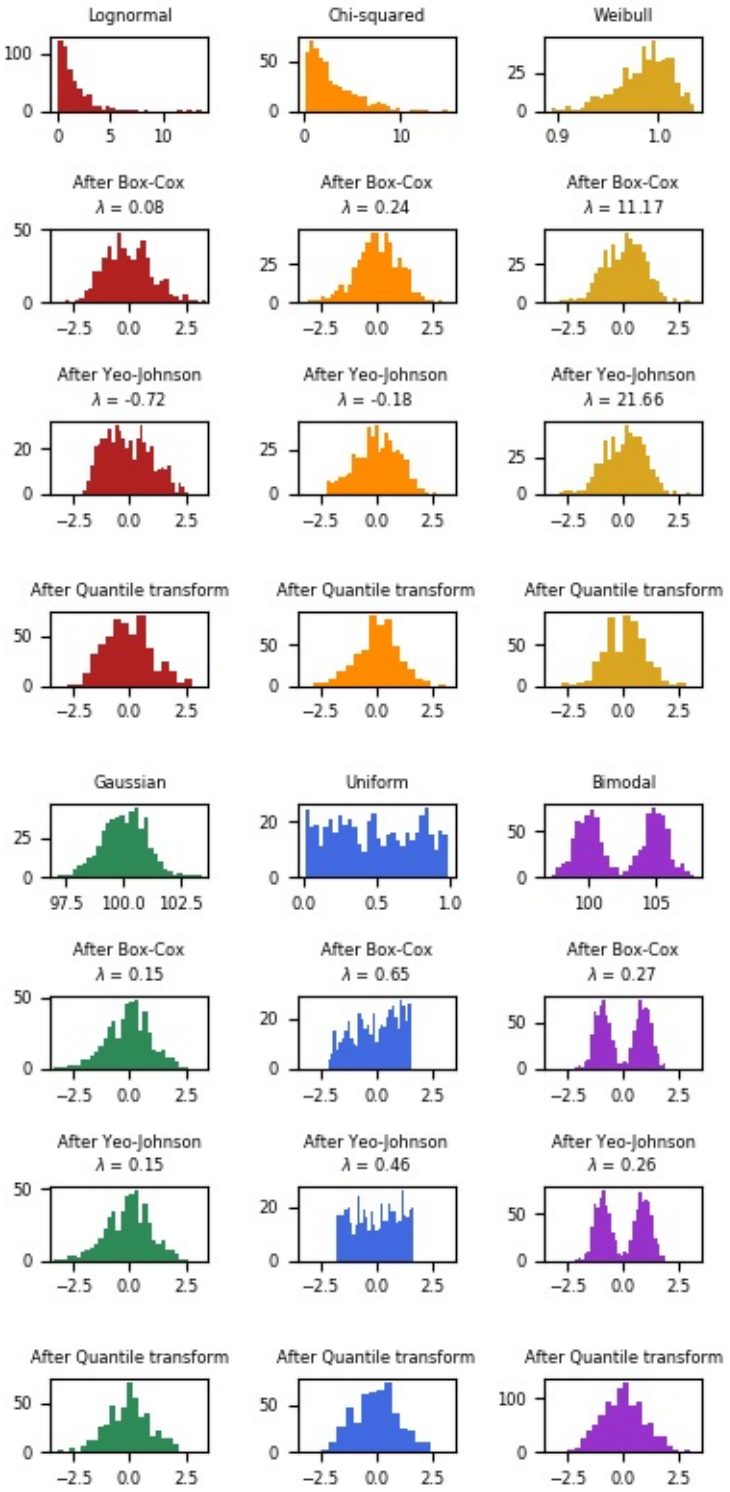
$$x_i^{(\lambda)} = \begin{cases} \dfrac{x_i^{\lambda} - 1}{\lambda} & \text{if } \lambda \neq 0, \\ \ln(x_i) & \text{if } \lambda = 0, \end{cases}$$

Box-Cox只能应用于严格的正数据。在这两种方法中，变换都是参数化的，通过极大似然估计来确定。下面是一个使用Box-Cox将样本从对数正态分布映射到正态分布的例子：

```
>>> pt = preprocessing.PowerTransformer(method='box-cox', standardize=False)
>>> X_lognormal = np.random.RandomState(616).lognormal(size=(3, 3))
>>> X_lognormal
array([[1.28..., 1.18..., 0.84...],
       [0.94..., 1.60..., 0.38...],
       [1.35..., 0.21..., 1.09...]])
>>> pt.fit_transform(X_lognormal)
array([[ 0.49...,  0.17..., -0.15...],
       [-0.05...,  0.58..., -0.57...],
       [ 0.69..., -0.84...,  0.10...]])
```

上述例子设置了参数 `standardize` 的选项为 `False` 。 但是，默认情况下，类`PowerTransformer`将会应用 `zero-mean` ， `unit-variance normalization` 到变换出的输出上。

下面的例子中 将 Box-Cox 和 Yeo-Johnson 应用到各种不同的概率分布上。 请注意 当把这些方法用到某个分布上的时候， 幂变换得到的分布非常像高斯分布。但是对其他的一些分布，结果却不太有效。这更加强调了在幂变换前后对数据进行可视化的重要性。



我们也可以 使用类 `QuantileTransformer` (通过设置 `output_distribution` ='normal')把数据变换成一个正态分布。下面是将其应用到iris dataset上的结果：

```
>>> quantile_transformer = preprocessing.QuantileTransformer(
...     output_distribution='normal', random_state=0)
>>> X_trans = quantile_transformer.fit_transform(X)
>>> quantile_transformer.quantiles_
array([[4.3, 2. , 1. , 0.1],
       [4.4, 2.2, 1.1, 0.1],
       [4.4, 2.2, 1.2, 0.1],
       ...,
```

```
        [7.7, 4.1, 6.7, 2.5],
        [7.7, 4.2, 6.7, 2.5],
        [7.9, 4.4, 6.9, 2.5]])
```

因此，输入的中值变成了输出的均值，以0为中心。正态输出被裁剪以便输入的最大最小值(分别对应于1e-7和1-1e-7)不会在变换之下变成无穷。

5.3.3 归一化

归一化 是 缩放单个样本以具有单位范数 的过程。如果你计划使用二次形式(如点积或任何其他核函数)来量化任何样本间的相似度，则此过程将非常有用。

这个观点基于 向量空间模型(Vector Space Model) ，经常在文本分类和内容聚类中使用.

函数 `normalize` 提供了一个快速简单的方法在类似数组的数据集上执行操作，使用 `l1` 或 `l2` 范式：

```
>>> X = [[ 1., -1.,  2.],
...      [ 2.,  0.,  0.],
...      [ 0.,  1., -1.]]
>>> X_normalized = preprocessing.normalize(X, norm='l2')

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

`preprocessing` 预处理模块提供的 `Normalizer` 工具类使用 `Transformer` API 实现了相同的操作(即使在这种情况下， `fit` 方法是无用的：该类是无状态的，因为该操作独立对待样本).

因此这个类适用于 `sklearn.pipeline.Pipeline` 的早期步骤：

```
>>> normalizer = preprocessing.Normalizer().fit(X)  # fit does nothing
>>> normalizer
Normalizer(copy=True, norm='l2')
```

在这之后归一化实例可以被使用在样本向量中，像任何其他转换器一样：

```
>>> normalizer.transform(X)
array([[ 0.40..., -0.40...,  0.81...],
 [ 1.  ...,  0.  ...,  0.  ...],
 [ 0.  ...,  0.70..., -0.70...]])

>>> normalizer.transform([[-1.,  1., 0.]])
array([[-0.70...,  0.70...,  0.  ...]])
```

> 稀疏(数据)输入
>
> 函数 `normalize` 以及类 `Normalizer` 接收 来自**scipy.sparse**的密集类数组数据和稀疏矩阵 作为输入。

> 对于稀疏输入，在被提交给高效Cython例程前，数据被 转化为压缩的稀疏行形式（参见 `scipy.sparse.csr_matrix` ）。为了避免不必要的内存复制，推荐在上游选择CSR表示。

5.3.4 类别特征编码

在机器学习中，特征经常不是连续的数值型的而是标称型的(categorical)。举个例子，一个人的样本具有特征 `["male", "female"]` ， `["from Europe", "from US", "from Asia"]` ， `["uses Firefox", "uses Chrome", "uses Safari", "uses Internet Explorer"]` 等。 这些特征能够被有效地编码成整数，比如 `["male", "from US", "uses Internet Explorer"]` 可以被表示为 `[0, 1, 3]` ， `["female", "from Asia", "uses Chrome"]` 表示为 `[1, 2, 1]` 。

要把标称型特征(categorical features) 转换为这样的整数编码(integer codes)，我们可以使用 `OrdinalEncoder` 。 这个估计器把每一个categorical feature变换成 一个新的整数数字特征 (0 到 n_categories - 1):

```
>>> enc = preprocessing.OrdinalEncoder()
>>> X = [['male', 'from US', 'uses Safari'], ['female', 'from Europe', 'uses Firefox']]
>>> enc.fit(X)
OrdinalEncoder(categories='auto', dtype=<... 'numpy.float64'>)
>>> enc.transform([['female', 'from US', 'uses Safari']])
array([[0., 1., 1.]])
```

这样的整数特征表示并不能在scikit-learn的估计器中直接使用，因为这样的连续输入，估计器会认为类别之间是有序的，但实际却是无序的。(例如：浏览器的类别数据是任意排序的)。

另外一种将标称型特征转换为能够被scikit-learn中模型使用的编码是one-of-K， 又称为 独热码或dummy encoding。 这种编码类型已经在类`OneHotEncoder`中实现。该类把每一个具有n_categories个可能取值的categorical特征变换为长度为n_categories的二进制特征向量，里面只有一个地方是1，其余位置都是0。

继续我们上面的例子：

```
>>>
>>> enc = preprocessing.OneHotEncoder()
>>> X = [['male', 'from US', 'uses Safari'], ['female', 'from Europe', 'uses Firefox']]
>>> enc.fit(X)
OneHotEncoder(categorical_features=None, categories=None,
       dtype=<... 'numpy.float64'>, handle_unknown='error',
       n_values=None, sparse=True)
>>> enc.transform([['female', 'from US', 'uses Safari'],
...                ['male', 'from Europe', 'uses Safari']]).toarray()
array([[1., 0., 0., 1., 0., 1.],
       [0., 1., 1., 0., 0., 1.]])
```

默认情况下，每个特征使用几维的数值可以从数据集自动推断。而且也可以在属性 `categories_` 中找到：

```
>>>
>>> enc.categories_
[array(['female', 'male'], dtype=object), array(['from Europe', 'from US'], dtype=object), array(['uses
Firefox', 'uses Safari'], dtype=object)]
```

可以使用参数 `categories_` 显式地指定这一点。我们的数据集中有两种性别、四种可能的大陆和四种web浏览器：

```
>>> genders = ['female', 'male']
>>> locations = ['from Africa', 'from Asia', 'from Europe', 'from US']
>>> browsers = ['uses Chrome', 'uses Firefox', 'uses IE', 'uses Safari']
>>> enc = preprocessing.OneHotEncoder(categories=[genders, locations, browsers])
>>> # Note that for there are missing categorical values for the 2nd and 3rd
>>> # feature
>>> X = [['male', 'from US', 'uses Safari'], ['female', 'from Europe', 'uses Firefox']]
>>> enc.fit(X)
OneHotEncoder(categorical_features=None,
       categories=[...], drop=None,
       dtype=<... 'numpy.float64'>, handle_unknown='error',
       n_values=None, sparse=True)
>>> enc.transform([['female', 'from Asia', 'uses Chrome']]).toarray()
array([[1., 0., 0., 1., 0., 0., 1., 0., 0., 0.]])
```

如果训练数据可能缺少分类特性，通常最好指定 `handle_unknown ='ignore'`，而不是像上面那样手动设置类别。当指定handle_unknown='ignore'，并且在转换过程中遇到未知类别时，不会产生错误，但是为该特性生成的一热编码列将全部为零(handle_unknown='ignore'只支持一热编码)：

如果训练数据中可能含有缺失的标称型特征，通过指定 `handle_unknown ='ignore'`比像上面代码那样手动设置 `categories` 更好。 当 `handle_unknown ='ignore'` 被指定并在变换过程中真的碰到了未知的 categories，则不会抛出任何错误,但是由此产生的该特征的one-hot编码列将会全部变成 0 。(这个参数设置选项 `handle_unknown ='ignore'` 仅仅在 one-hot encoding的时候有效)：

```
>>> enc = preprocessing.OneHotEncoder(handle_unknown='ignore')
>>> X = [['male', 'from US', 'uses Safari'], ['female', 'from Europe', 'uses Firefox']]
>>> enc.fit(X)
OneHotEncoder(categorical_features=None, categories=None, drop=None,
       dtype=<... 'numpy.float64'>, handle_unknown='ignore',
       n_values=None, sparse=True)
>>> enc.transform([['female', 'from Asia', 'uses Chrome']]).toarray()
array([[1., 0., 0., 0., 0., 0.]])
```

还可以使用 `drop` 参数将每个列编码为n_categories-1列，而不是n_categories列。此参数允许用户为要删除的每个特征指定类别。这对于避免某些分类器中输入矩阵的共线性是有用的。例如，当使用非正则化回归(线性回归)时，这种功能是有用的，因为共线性会导致协方差矩阵是不可逆的。当这个参数不是None时， `handle_unknown` 必须设置为error：

```
>>> X = [['male', 'from US', 'uses Safari'], ['female', 'from Europe', 'uses Firefox']]
>>> drop_enc = preprocessing.OneHotEncoder(drop='first').fit(X)
>>> drop_enc.categories_
[array(['female', 'male'], dtype=object), array(['from Europe', 'from US'], dtype=object), array(['uses
Firefox', 'uses Safari'], dtype=object)]
>>> drop_enc.transform(X).toarray()
array([[1., 1., 1.],
       [0., 0., 0.]])
```

标称型特征有时是用字典来表示的，而不是标量，具体请参阅从字典中加载特征。

5.3.5 离散化

离散化 (Discretization) (有些时候叫 量化(quantization) 或 装箱(binning)) 提供了将连续特征划分为离散特征值的方法。 某些具有连续特征的数据集会受益于离散化，因为 离散化可以把具有连续属性的数据集变换成只有名义属性(nominal attributes)的数据集。 (译者注： nominal attributes 其实就是 categorical features, 可以译为 名称属性，名义属性，符号属性，离散属性 等)

One-hot 编码的离散化特征 可以使得一个模型更加的有表现力(expressive)，同时还能保留其可解释性(interpretability)。 比如，用离散化器进行预处理可以给线性模型引入非线性。

5.3.5.1 K-bins 离散化

KBinsDiscretizer 类使用k个等宽的bins把特征离散化

```
>>> X = np.array([[ -3., 5., 15 ],
...               [  0., 6., 14 ],
...               [  6., 3., 11 ]])
>>> est = preprocessing.KBinsDiscretizer(n_bins=[3, 2, 2], encode='ordinal').fit(X)
```

默认情况下，输出是被 one-hot 编码到一个稀疏矩阵。(请看类别特征编码)。 而且可以使用参数 `encode` 进行配置。对每一个特征， bin的边界以及总数目在 `fit` 过程中被计算出来，它们将用来定义区间。 因此，对现在的例子，这些区间间隔被定义如下：

- 特征 1: `[-∞,-1],[-1,2),[2,∞)`
- 特征 2: `[-∞,5),[5,∞)`
- 特征 3: `[-∞,14],[14,∞)`

基于这些 bin 区间， X 就被变换成下面这样：

```
>>> est.transform(X)
array([[ 0., 1., 1.],
       [ 1., 1., 1.],
       [ 2., 0., 0.]])
```

由此产生的数据集包含了有序属性(ordinal attributes),可以被进一步用在类 sklearn.pipeline.Pipeline 中。

离散化(Discretization)类似于为连续数据构建直方图(histograms)。 然而，直方图聚焦于统计特征落在特定的bins里面的数量，而离散化聚焦于给这些bins分配特征取值。

`KBinsDiscretizer` 类实现了不同的 binning策略，可以通过参数 `strategy` 进行选择。 ‘uniform’ 策略使用固定宽度的bins。 ‘quantile’ 策略在每个特征上使用分位数(quantiles)值以便具有相同填充的bins。 ‘kmeans’ 策略基于在每个特征上独立执行的k-means聚类过程定义bins。

> 示例
>
> - Using KBinsDiscretizer to discretize continuous features

- Feature discretization
- Demonstrating the different strategies of KBinsDiscretizer

5.3.5.2 特征二值化

特征二值化 是 将数值特征用阈值过滤得到布尔值 的过程。这对于下游的概率型模型是有用的，它们假设输入数据是多值 伯努利分布(Bernoulli distribution) 。例如这个例子 `sklearn.neural_network.BernoulliRBM` 。

即使归一化计数(又名术语频率)和TF-IDF值特征在实践中表现稍好一些，文本处理团队也常常使用二值化特征值(这可能会简化概率估计)。

相比于 `Normalizer` ，实用程序类 `Binarizer` 也被用于 `sklearn.pipeline.Pipeline` 的早期步骤中。因为每个样本被当做是独立于其他样本的，所以 `fit` 方法是无用的：

```
>>> X = [[ 1., -1.,  2.],
...      [ 2.,  0.,  0.],
...      [ 0.,  1., -1.]]

>>> binarizer = preprocessing.Binarizer().fit(X)  # fit does nothing
>>> binarizer
Binarizer(copy=True, threshold=0.0)

>>> binarizer.transform(X)
array([[ 1.,  0.,  1.],
 [ 1.,  0.,  0.],
 [ 0.,  1.,  0.]])
```

也可以为二值化器赋一个阈值：

```
>>> binarizer = preprocessing.Binarizer(threshold=1.1)
>>> binarizer.transform(X)
array([[ 0.,  0.,  1.],
 [ 1.,  0.,  0.],
 [ 0.,  0.,  0.]])
```

相比于 `StandardScaler` 和 `Normalizer` 类的情况，预处理模块提供了一个相似的函数 `binarize` ，以便不需要转换接口时使用。

稀疏输入

`binarize` 以及 `Binarizer` 接收 来自`scipy.sparse`的密集类数组数据以及稀疏矩阵作为输入 。

对于稀疏输入，数据被 转化为压缩的稀疏行形式 (参见 `scipy.sparse.csr_matrix`)。为了避免不必要的内存复制，推荐在上游选择CSR表示。

5.3.6 缺失值补全

关于缺失值补全的方法和工具的讨论，请看章节： 缺失值处理。

5.3.7 生成多项式特征

在机器学习中，通过增加一些输入数据的非线性特征来增加模型的复杂度通常是有效的。一个简单通用的办法是使用多项式特征，这可以获得特征的更高维度和互相间关系的项。这在 `PolynomialFeatures` 中实现：

```
>>> import numpy as np
>>> from sklearn.preprocessing import PolynomialFeatures
>>> X = np.arange(6).reshape(3, 2)
>>> X
array([[0, 1],
 [2, 3],
 [4, 5]])
>>> poly = PolynomialFeatures(2)
>>> poly.fit_transform(X)
array([[  1.,   0.,   1.,   0.,   0.,   1.],
 [  1.,   2.,   3.,   4.,   6.,   9.],
 [  1.,   4.,   5.,  16.,  20.,  25.]])
```

X 的特征已经从 (X_1, X_2) 转换为 $(1, X_1, X_2, X_1^2, X_1X_2, X_2^2)$ 。

在一些情况下，只需要特征间的交互项，这可以通过设置 `interaction_only=True` 来得到：

```
>>> X = np.arange(9).reshape(3, 3)
>>> X
array([[0, 1, 2],
 [3, 4, 5],
 [6, 7, 8]])
>>> poly = PolynomialFeatures(degree=3, interaction_only=True)
>>> poly.fit_transform(X)
array([[   1.,    0.,    1.,    2.,    0.,    0.,    2.,    0.],
 [   1.,    3.,    4.,    5.,   12.,   15.,   20.,   60.],
 [   1.,    6.,    7.,    8.,   42.,   48.,   56.,  336.]])
```

X的特征已经从 (X_1, X_2, X_3) 转换为 $(1, X_1, X_2, X_3, X_1X_2, X_1X_3, X_2X_3, X_1X_2X_3)$ 。

注意，当使用多项的 `Kernel functions` 时，多项式特征被隐式地在核函数中被调用(比如，`sklearn.svm.SVC`，`sklearn.decomposition.KernelPCA`)。

创建并使用多项式特征的岭回归实例请见 `Polynomial interpolation` 。

5.3.8 自定义转换器

在机器学习中，想要将一个已有的 Python 函数转化为一个转换器来协助数据清理或处理。可以使用 `FunctionTransformer` 从任意函数中实现一个转换器。例如，在一个管道中构建一个实现日志转换的转化器，这样做：

```
>>> import numpy as np
>>> from sklearn.preprocessing import FunctionTransformer
>>> transformer = FunctionTransformer(np.log1p, validate=True)
>>> X = np.array([[0, 1], [2, 3]])
>>> transformer.transform(X)
array([[0.        , 0.69314718],
       [1.09861229, 1.38629436]])
```

通过设置 `check_reverse` =True并在转换之前调用 `fit` ，可以确保 `func` 和 `inverse_func` 是彼此的拟过程。请注意，请注意一个 `warning` 会被抛出，并且可以使用 `filterwarnings` 将其转为一个 `error`

使用一个 `FunctionTransformer` 类来做定制化特征选择的例子，请见 Using FunctionTransformer to select columns 。

我们一直在努力

apachecn/sklearn-doc-zh



5.4 缺失值插补

校验者：@if only 待二次校验翻译者：@Trembleguy@Loopy

因为各种各样的原因，真实世界中的许多数据集都包含缺失数据，这类数据经常被编码成空格、NaNs，或者是其他的占位符。但是这样的数据集并不能scikit-learn学习算法兼容，因为大多的学习算法都默认假设数组中的元素都是数值，因而所有的元素都有自己的意义。 使用不完整的数据集的一个基本策略就是舍弃掉整行或整列包含缺失值的数据。但是这样就付出了舍弃可能有价值数据（即使是不完整的 ）的代价。 处理缺失数值的一个更好的策略就是从已有的数据推断出缺失的数值。有关插补(imputation)，请参阅常用术语表和API元素条目。

5.4.1 单变量与多变量插补

一种类型的插补算法是单变量算法，它只使用第i个特征维度中的非缺失值(如 `impute.SimpleImputer`)来插补第i个特征维中的值。相比之下，多变量插补算法使用整个可用特征维度来估计缺失的值(如 `impute.IterativeImputer`)。

5.4.2 单变量插补

`SimpleImputer` 类提供了计算缺失值的基本策略。缺失值可以用提供的常数值计算，也可以使用缺失值所在的行/列中的统计数据(平均值、中位数或者众数)来计算。这个类也支持不同的缺失值编码。

以下代码段演示了如何使用包含缺失值的列(轴0)的平均值来替换编码为 `np.nan` 的缺失值：

```
>>> import numpy as np
>>> from sklearn.preprocessing import Imputer
>>> imp = Imputer(missing_values='NaN', strategy='mean', axis=0)
>>> imp.fit([[1, 2], [np.nan, 3], [7, 6]])
Imputer(axis=0, copy=True, missing_values='NaN', strategy='mean', verbose=0)
>>> X = [[np.nan, 2], [6, np.nan], [7, 6]]
>>> print(imp.transform(X))
[[ 4.          2.        ]
 [ 6.          3.666...]
 [ 7.          6.        ]]
```

`SimpleImputer` 类也支持稀疏矩阵：

```
>>> import scipy.sparse as sp
>>> X = sp.csc_matrix([[1, 2], [0, 3], [7, 6]])
>>> imp = Imputer(missing_values=0, strategy='mean', axis=0)
>>> imp.fit(X)
Imputer(axis=0, copy=True, missing_values=0, strategy='mean', verbose=0)
>>> X_test = sp.csc_matrix([[0, 2], [6, 0], [7, 6]])
>>> print(imp.transform(X_test))
[[ 4.          2.        ]
 [ 6.          3.666...]
 [ 7.          6.        ]]
```

注意，此格式不用于隐式存储矩阵中的缺失值，因为它会在转换时将其密集化。编码为0的缺失值必须与密集输入一起使用。

当使用 `'most_frequent'` 或 `'constant'` 策略时， `SimpleImputer` 类还支持以 string values 或 pandas categoricals 表示的分类数据(categorical data)

```
>>> import pandas as pd
>>> df = pd.DataFrame([["a", "x"],
...                    [np.nan, "y"],
...                    ["a", np.nan],
...                    ["b", "y"]], dtype="category")
...
>>> imp = SimpleImputer(strategy="most_frequent")
>>> print(imp.fit_transform(df))
[['a' 'x']
 ['a' 'y']
 ['a' 'y']
 ['b' 'y']]
```

5.4.3 多变量插补

一种更复杂的方法是使用`IterativeImputer`类，它将每个缺失值的特征建模为其他特征的函数，并使用该估计值进行估算。它以迭代循环方式执行：在每个步骤中，将要素目标列指定为输出y，将其他列视为输入X。使用一个回归器来在已知（未缺失）y 的样本上，对（X，y）进行拟合。然后使用这个回归器来预测缺失的 y 值。这是以迭代的方式对每个特征进行的，然后重复 `max_iter` 轮。最后一轮的计算结果被返回。

> 注意 :这个估计器目前还处于试验阶段:预测和API可能会在没有任何弃用周期的情况下发生变化。要使用它，您需要显式地导入 `enable_iterative_imputer` 。

```
>>> import numpy as np
>>> from sklearn.experimental import enable_iterative_imputer
>>> from sklearn.impute import IterativeImputer
>>> imp = IterativeImputer(max_iter=10, random_state=0)
>>> imp.fit([[1, 2], [3, 6], [4, 8], [np.nan, 3], [7, np.nan]])
IterativeImputer(add_indicator=False, estimator=None,
                 imputation_order='ascending', initial_strategy='mean',
                 max_iter=10, max_value=None, min_value=None,
                 missing_values=nan, n_nearest_features=None,
                 random_state=0, sample_posterior=False, tol=0.001,
                 verbose=0)
>>> X_test = [[np.nan, 2], [6, np.nan], [np.nan, 6]]
>>> # the model learns that the second feature is double the first
>>> print(np.round(imp.transform(X_test)))
[[ 1.  2.]
 [ 6. 12.]
 [ 3.  6.]]
```

`SimpleImputer` 和`IterativeImputer`都可以用来在管道中构建支持计算的复合估计器。在构建估算器之前，请参阅一个估算缺失值的例子。

5.4.3.1 多变量插补的灵活性

在R数据科学生态系统中，有许多成熟的估算包:Amelia, mi, mice, missForest等。misforest是一种很流行的算法，它是不同序列计算算法的一个特殊实例，这些算法都可以使用IterativeImputer来实现，通过传递不同的回归函数来预测缺失的特征值。在misforest的情况下，这个回归因子是一个随机森林。请参见Imputing missing values with variants of IterativeImputer

5.4.3.2 单次与多次插补

在统计学界，通常的做法是执行多次计算，例如，为单个特征矩阵生成m个单独的计算。然后，每一个m的估算都通过后续的分析管道(例如，特征工程、聚类、回归、分类)进行。m个最终分析结果(例如，延迟验证错误)允许数据科学家了解由于缺失值所导致的固有不确定性，分析结果可能会有何不同。上述做法称为多重插补。

我们实现IterativeImputer的灵感来自于R的MICE包[1]，但与之不同的是，我们返回了一个单一的插补，而不是多个插补。然而，当 `sample_posterior` =True时， `IterativeImputer` 也可以通过重复应用于具有不同随机种子的同一数据集来进行多次计算。参见[2]第4章，以获得更多关于多重和单次估算的讨论。

当用户对由于缺失值而导致的测量不确定性不感兴趣时，单次和多次插补在预测和分类上下文中有多大用处，这仍然是一个有待解决的问题。

注意，调用 `IterativeImputer` 的转换方法不允许改变样本的数量。因此，单次调用 `transform` 不能实现多次计算。

5.4.4 参考

- Stef van Buuren, Karin Groothuis-Oudshoorn (2011). “mice: Multivariate Imputation by Chained Equations in R”. Journal of Statistical Software 45: 1-67.
- Roderick J A Little and Donald B Rubin (1986). “Statistical Analysis with Missing Data”. John Wiley & Sons, Inc., New York, NY, USA.

5.4.5 标记缺失值

`MissingIndicator` 转换器用于将数据集转换为相应的二进制矩阵，以指示数据集中缺失值的存在。这个变换与归算结合起来是有用的。当使用插补时，保存关于哪些值丢失的信息可以提供有用的信息。

`NaN` 通常用作缺少值的占位符。但是，它强制数据类型为浮点数。参数 `missing_values` 允许指定其他占位符，如整数。 在以下示例中，我们将使用-1作为缺失值

```
>>> from sklearn.impute import MissingIndicator
>>> X = np.array([[-1, -1, 1, 3],
...               [4, -1, 0, -1],
...               [8, -1, 1, 0]])
>>> indicator = MissingIndicator(missing_values=-1)
>>> mask_missing_values_only = indicator.fit_transform(X)
>>> mask_missing_values_only
array([[ True,  True, False],
       [False,  True,  True],
       [False,  True, False]])
```

参数 `features` 用于选择构造掩码的特征。默认情况下，它是 `'missing-only'`,在 `fit` 时返回包含缺失值的特征的输入掩码

```
>>> indicator.features_
array([0, 1, 3])
```

参数 `features` 可以设置为`'all'`以返回所有特征，无论它们是否包含缺失的值

```
>>> indicator = MissingIndicator(missing_values=-1, features="all")
>>> mask_all = indicator.fit_transform(X)
>>> mask_all
array([[ True,  True, False, False],
       [False,  True, False,  True],
       [False,  True, False, False]])
>>> indicator.features_
array([0, 1, 2, 3])
```

当在 Pipeline 中使用 MissingIndicator时，务必使用 `FeatureUnion` 或 `ColumnTransformer` 来添加 indicator features 到 regular features中. 首先，我们在iris数据集上插补一些缺失值:

```
>>> from sklearn.datasets import load_iris
>>> from sklearn.impute import SimpleImputer, MissingIndicator
>>> from sklearn.model_selection import train_test_split
>>> from sklearn.pipeline import FeatureUnion, make_pipeline
>>> from sklearn.tree import DecisionTreeClassifier
>>> X, y = load_iris(return_X_y=True)
>>> mask = np.random.randint(0, 2, size=X.shape).astype(np.bool)
>>> X[mask] = np.nan
>>> X_train, X_test, y_train, _ = train_test_split(X, y, test_size=100,
...                                                random_state=0)
```

现在我们创建一个FeatureUnion。为了使分类器能够处理这些数据，所有的特征都将使用SimpleImputer进行估算。 此外，它还从 MissingIndicator中添加指示变量。

```
>>> transformer = FeatureUnion(
...     transformer_list=[
...         ('features', SimpleImputer(strategy='mean')),
...         ('indicators', MissingIndicator())])
>>> transformer = transformer.fit(X_train, y_train)
>>> results = transformer.transform(X_test)
>>> results.shape
(100, 8)
```

当然，我们不能用transformer来做任何预测。我们应该用分类器(例如， `DecisionTreeClassifier`)将其封装在pipeline中， 以便能够进行预测。

```
>>> clf = make_pipeline(transformer, DecisionTreeClassifier())
>>> clf = clf.fit(X_train, y_train)
>>> results = clf.predict(X_test)
```

```
>>> results.shape
(100,)
```

我们一直在努力

apachecn/sklearn-doc-zh



5.5. 无监督降维

校验者：@程威翻译者：@十四号

如果你的特征数量很多，在监督步骤之前，可以通过无监督的步骤来减少特征．很多的 无监督学习 方法实现了一个名为 `transform` 的方法，它可以用来降低维度．下面我们将讨论大量使用这种模式的两个具体示例．

> **Pipelining**非监督数据约简和监督估计器可以链接起来。 请看 Pipeline：链式评估器.

5.5.1. PCA：主成份分析

`decomposition.PCA` 寻找能够捕捉原始特征的差异的特征的组合．请参阅 分解成分中的信号（矩阵分解问题）．

> 示例
>
> - Faces recognition example using eigenfaces and SVMs

5.5.2. 随机投影

模块： `random_projection` 提供了几种用于通过随机投影减少数据的工具．请参阅文档的相关部分：随机投影．

> 示例
>
> - The Johnson-Lindenstrauss bound for embedding with random projections

5.5.3. 特征聚集

`cluster.FeatureAgglomeration` 应用 层次聚类 将行为类似的特征分组在一起．

> 示例
>
> - Feature agglomeration vs. univariate selection
> - Feature agglomeration
>
> 特征缩放
>
> 请注意，如果功能具有明显不同的缩放或统计属性，则 `cluster.FeatureAgglomeration` 可能无法捕获相关特征之间的关系．使用一个 `preprocessing.StandardScaler` 可以在这些 设置中使用．

我们一直在努力

apachecn/sklearn-doc-zh

加入QQ群

5.6. 随机投影

校验者：@FontTian@程威翻译者：@Sehriff

`sklearn.random_projection` 模块实现了一个简单且高效率的计算方式来减少数据维度，通过牺牲一定的精度（作为附加变量）来加速处理时间及更小的模型尺寸。 这个模型实现了两类无结构化的随机矩阵：Gaussian random matrix 和 sparse random matrix.

随机投影矩阵的维度和分布是受控制的，所以可以保存任意两个数据集的距离。因此随机投影适用于基于距离的方法。

> 参考资料：
>
> - Sanjoy Dasgupta. 2000. Experiments with random projection. In Proceedings of the Sixteenth conference on Uncertainty in artificial intelligence (UAI'00), Craig Boutilier and Moisés Goldszmidt (Eds.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 143-151.
> - Ella Bingham and Heikki Mannila. 2001. Random projection in dimensionality reduction: applications to image and text data. In Proceedings of the seventh ACM SIGKDD international conference on Knowledge discovery and data mining (KDD '01). ACM, New York, NY, USA, 245-250.

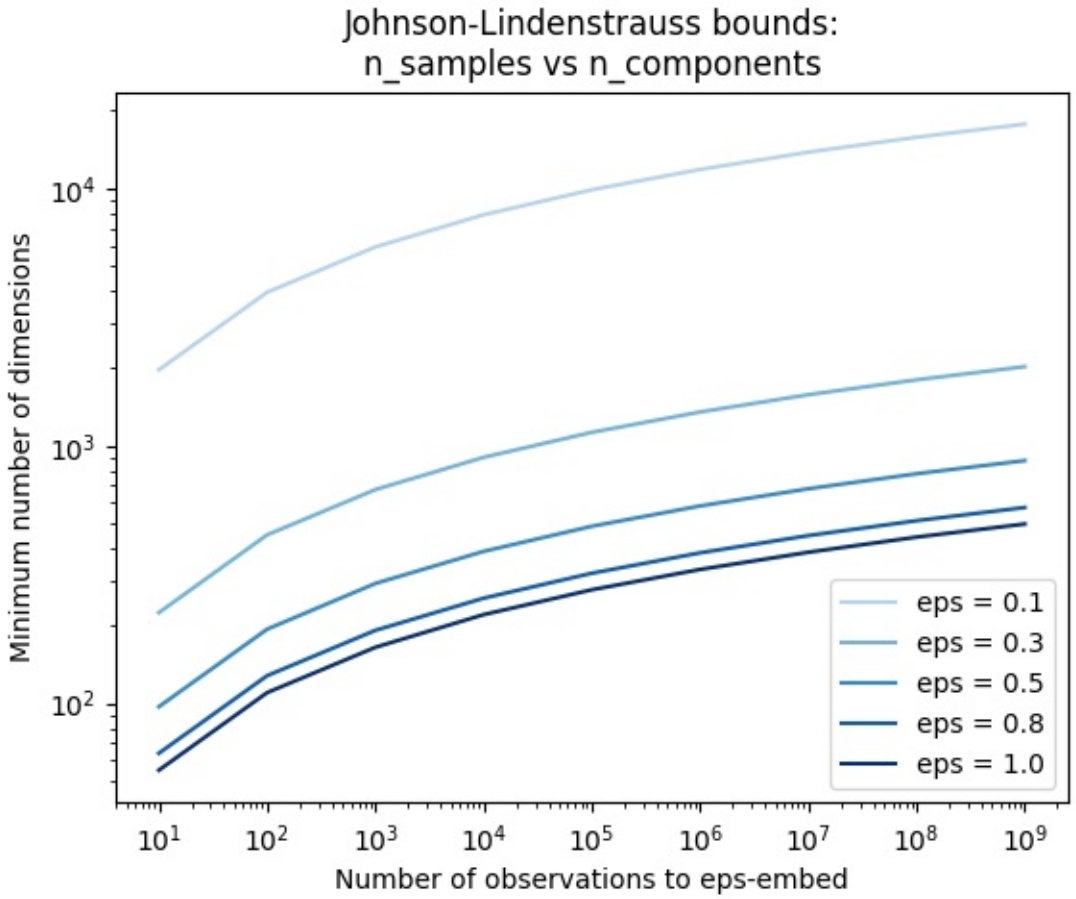
5.6.1. Johnson-Lindenstrauss 辅助定理

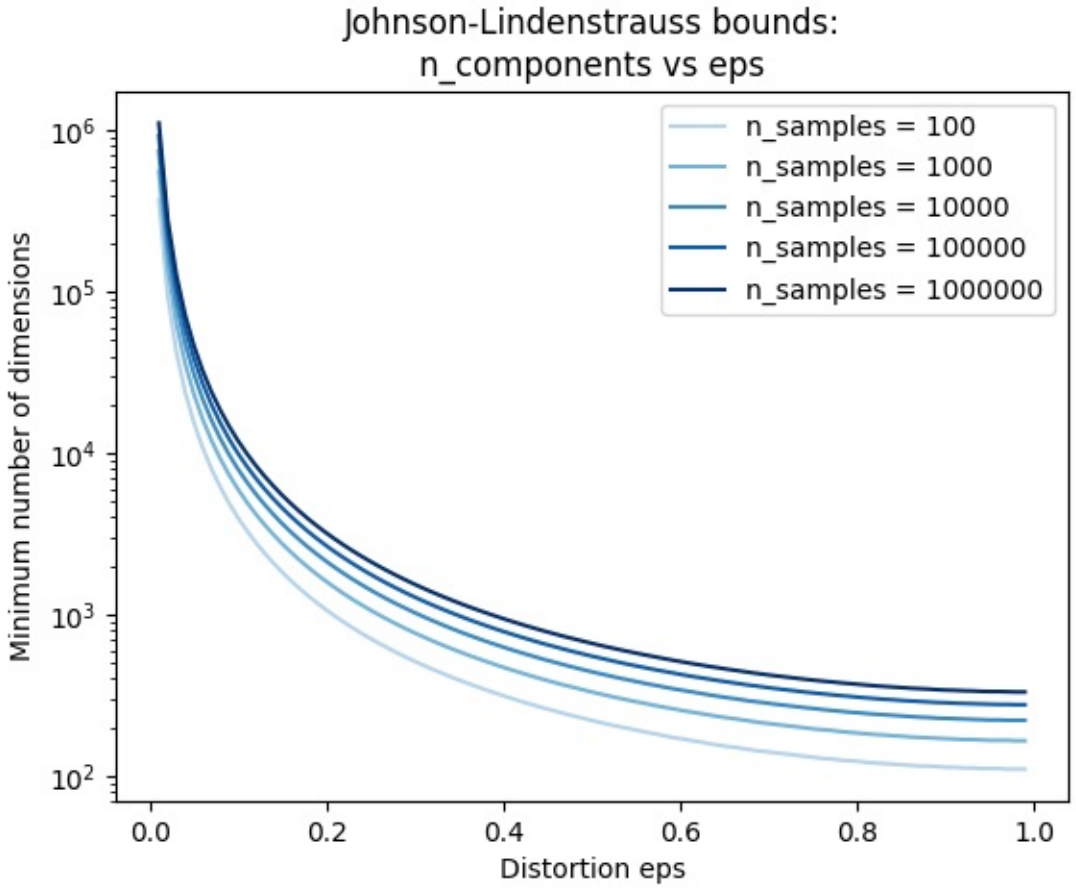
支撑随机投影效率的主要理论成果是Johnson-Lindenstrauss lemma (quoting Wikipedia):

> 在数学中，johnson - lindenstrauss 引理是一种将高维的点从高维到低维欧几里得空间的低失真嵌入的方案。 引理阐释了高维空间下的一小部分的点集可以内嵌到非常低维的空间，这种方式下点之间的距离几乎全部被保留。 内嵌所用到的映射至少符合 Lipschitz 条件，甚至可以被当做正交投影。

有了样本数量， `sklearn.random_projection.johnson_lindenstrauss_min_dim` 会保守估计随机子空间的最小大小来保证随机投影导致的变形在一定范围内：

```
>>> from sklearn.random_projection import johnson_lindenstrauss_min_dim
>>> johnson_lindenstrauss_min_dim(n_samples=1e6, eps=0.5)
663
>>> johnson_lindenstrauss_min_dim(n_samples=1e6, eps=[0.5, 0.1, 0.01])
array([    663,   11841, 1112658])
>>> johnson_lindenstrauss_min_dim(n_samples=[1e4, 1e5, 1e6], eps=0.1)
array([ 7894,  9868, 11841])`
```





示例：

- 查看 The Johnson-Lindenstrauss bound for embedding with random projections 里面有Johnson-Lindenstrauss引理的理

论说明和使用稀疏随机矩阵的经验验证。

参考资料：

- Sanjoy Dasgupta and Anupam Gupta, 1999. An elementary proof of the Johnson-Lindenstrauss Lemma.

5.6.2. 高斯随机投影

The `sklearn.random_projection.GaussianRandomProjection` 通过将原始输入空间投影到随机生成的矩阵（该矩阵的组件由以下分布中抽取）：$N(0, \frac{1}{n_{components}})$ 降低维度。

以下小片段演示了任何使用高斯随机投影转换器：

```
>>> import numpy as np
>>> from sklearn import random_projection
>>> X = np.random.rand(100, 10000)
>>> transformer = random_projection.GaussianRandomProjection()
>>> X_new = transformer.fit_transform(X)
>>> X_new.shape
(100, 3947)
```

5.6.3. 稀疏随机矩阵

`sklearn.random_projection.SparseRandomProjection` 使用稀疏随机矩阵，通过投影原始输入空间来降低维度。

稀疏矩阵可以替换高斯随机投影矩阵来保证相似的嵌入质量，且内存利用率更高、投影数据的计算更快。

如果我们定义 `s = 1 / density` ，随机矩阵的元素由下式抽取。

$$\begin{cases} -\sqrt{\frac{s}{n_{\text{components}}}} & 1/2s \\ 0 & \text{with probability } 1 - 1/s \\ +\sqrt{\frac{s}{n_{\text{components}}}} & 1/2s \end{cases}$$

其中 $n_{\text{components}}$ 是投影后的子空间大小。 默认非零元素的浓密度设置为最小浓密度，该值由Ping Li et al.：$N(0, \frac{1}{n_{components}})$ 推荐，根据公式： 计算。

以下小片段演示了如何使用稀疏随机投影转换器：

```
>>> import numpy as np
>>> from sklearn import random_projection
>>> X = np.random.rand(100,10000)
```

```
>>> transformer = random_projection.SparseRandomProjection()
>>> X_new = transformer.fit_transform(X)
>>> X_new.shape
(100, 3947)
```

参考资料：

- D. Achlioptas. 2003. Database-friendly random projections: Johnson-Lindenstrauss with binary coins. Journal of Computer and System Sciences 66 (2003) 671–687
- Ping Li, Trevor J. Hastie, and Kenneth W. Church. 2006. Very sparse random projections. In Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining (KDD '06). ACM, New York, NY, USA, 287-296.

我们一直在努力

apachecn/sklearn-doc-zh



5.7. 内核近似

校验者：@FontTian@numpy@Loopy翻译者：@程威

这个子模块包含与某些 kernel 对应的特征映射的函数，这个会用于例如支持向量机的算法当中(see 支持向量机)。 下面这些特征函数对输入执行非线性转换，可以用于线性分类或者其他算法。

与 kernel trick 相比，近似的进行特征映射更适合在线学习，并能够有效 减少学习大量数据的内存开销。使用标准核技巧的 svm 不能有效的适用到海量数据，但是使用近似内核映射的方法，对于线性 SVM 来说效果可能更好。 而且，使用 `SGDClassifier` 进行近似的内核映射，使得对海量数据进行非线性学习也成为了可能。

由于近似嵌入的方法没有太多经验性的验证，所以建议将结果和使用精确的内核方法的结果进行比较。

也可参阅多项式回归：用基函数展开线性模型 用于精确的多项式变换。

5.7.1. 内核近似的 Nystroem 方法

`Nystroem` 中实现了 Nystroem 方法用于低等级的近似核。它是通过采样 kernel 已经评估好的数据。默认情况下， `Nystroem` 使用 `rbf` kernel，但它可以使用任何内核函数和预计算内核矩阵. 使用的样本数量 - 计算的特征维数 - 由参数 `n_components` 给出.

5.7.2. 径向基函数内核

`RBFSampler` 为径向基函数核构造一个近似映射，又称为 Random Kitchen Sinks [RR2007]. 在应用线性算法（例如线性 SVM ）之前，可以使用此转换来明确建模内核映射：

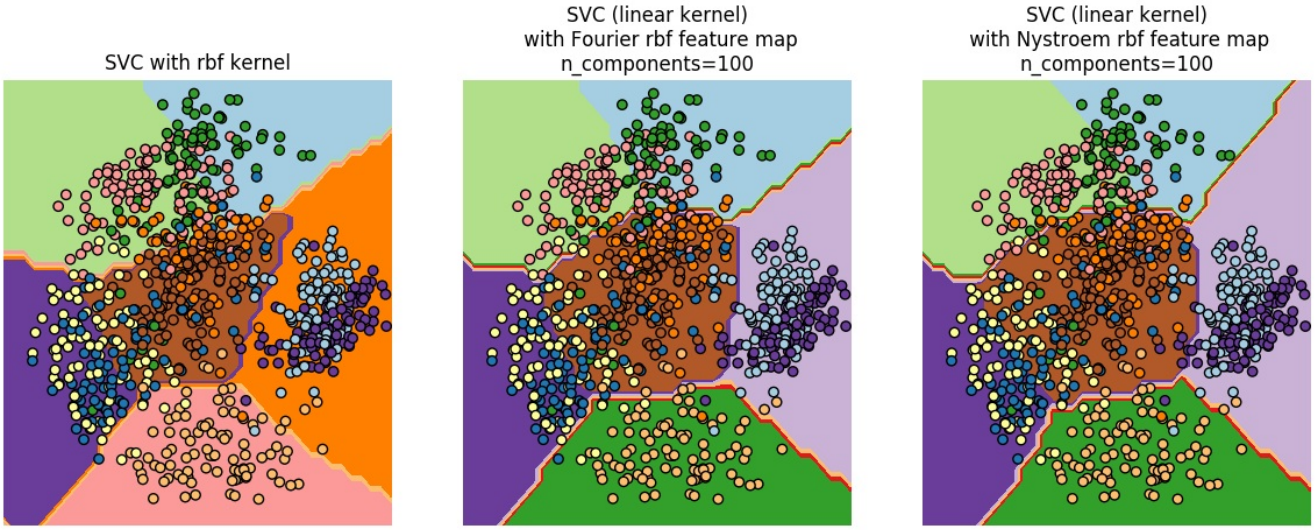
```
>>> from sklearn.kernel_approximation import RBFSampler
>>> from sklearn.linear_model import SGDClassifier
>>> X = [[0, 0], [1, 1], [1, 0], [0, 1]]
>>> y = [0, 0, 1, 1]
>>> rbf_feature = RBFSampler(gamma=1, random_state=1)
>>> X_features = rbf_feature.fit_transform(X)
>>> clf = SGDClassifier()
>>> clf.fit(X_features, y)
SGDClassifier(alpha=0.0001, average=False, class_weight=None, epsilon=0.1,
 eta0=0.0, fit_intercept=True, l1_ratio=0.15,
 learning_rate='optimal', loss='hinge', max_iter=5, n_iter=None,
 n_jobs=1, penalty='l2', power_t=0.5, random_state=None,
 shuffle=True, tol=None, verbose=0, warm_start=False)
>>> clf.score(X_features, y)
1.0
```

这个映射依赖于内核值的 Monte Carlo 近似. `fit` 方法执行 Monte Carlo 采样，而该 `transform` 方法执行 数据的映射.由于过程的固有随机性，结果可能会在不同的 `fit` 函数调用之间变化。

该 `fit` 函数有两个参数： `n_components` 是特征变换的目标维数. `gamma` 是 RBF-kernel 的参数.

`n_components` 越高，会导致更好的内核近似， 并且将产生与内核 SVM 产生的结果更相似的结果。请注意，”拟合” 特征函数实际上不取决于 `fit` 函数传递的数据。只有数据的维数被使用。 详情可以参考 [RR2007].

对于给定的值 `n_components` `RBFSampler` 在 `Nystroem` 中使用通常不太准确， 但是 `RBFSampler` 使用更大的特征空间，更容易计算。



将精确的 RBF kernel (左) 与 approximation (右) 进行比较。

> 示例：
>
> - Explicit feature map approximation for RBF kernels

5.7.3. 加性卡方核

Additive Chi Squared Kernel (加性卡方核)是直方图的核心，通常用于计算机视觉。

这里使用的 Additive Chi Squared Kernel 给出

$$k(x, y) = \sum_i \frac{2x_i y_i}{x_i + y_i}$$

这个和 `sklearn.metrics.additivechi2_kernel` 不完全一样.*[VZ2010]* 的作者喜欢上面的版本，因为它总是积极的。 由于这个 kernel 是可添加的，因此可以分别处理嵌入的 x_i. 这使得在规则的间隔类对傅里叶变换进行性才赢，代替近似的 Monte Carlo 采样。

`AdditiveChi2Sampler` 类实现了这个组件采样方法. 每个组件都被采样 n 次，每一个输入维数都会产生 *2n+1* 维（来自傅立叶变换的实部和复数部分的两个数据段的倍数）. 在文献中， n 经常取为 1 或者 2，将数据集转换为 `n_samples * n_features` 大小（在 $n = 2$ 的情况下 ）.

`AdditiveChi2Sampler` 提供的近似特征映射可以和 `RBFSampler` 提供的近似特征映射合并，得到一个取幂的 chi squared kerne。可以查看 [VZ2010] 和 [VVZ2010] `RBFSampler` 的合并.

5.7.4. Skewed Chi Squared Kernel (偏斜卡方核?暂译)

skewed chi squared kernel 给出下面公式

$$k(x,y) = \prod_i \frac{2\sqrt{x_i + c}\sqrt{y_i + c}}{x_i + y_i + 2c}$$

它有和 指数卡方核 相似的属性，用于计算机视觉.但是允许进行简单的蒙特卡洛近似的特征映射。

`SkewedChi2Sampler` 的使用和之前描述的 `RBFSampler` 一样.唯一的区别是自由参数，称之为 c. 这种映射和数学细节可以参考 [LS2010].

5.7.5. 数学方面的细节

核技巧 像支持向量机，或者 核化 PCA 依赖于 再生核希尔伯特空间（RKHS） 对于任何 核函数 k （叫做 Mercer kernel），保证了 ϕ 进入 希尔伯特空间 $\mathcal{H}$ 的映射，例如:

$$k(x,y) = \langle \phi(x), \phi(y) \rangle$$

$\langle \cdot, \cdot \rangle$ 是在 Hilbert space 中做内积.

如果一个算法，例如线性支持向量机或者 PCA，依赖于数据集的数量级 x_i ，可能会使用 $k(x_i, x_j)$ ， 符合孙发的映射 $\phi(x_i)$. 使用 k 的优点在于 ϕ 永远不会直接计算，允许大量的特征计算（甚至是无限的）.

kernel 方法的一个缺点是，在优化过程中有可能存储大量的 kernel 值 $k(x_i, x_j)$. 如果使用核函数的分类器应用于新的数据 y_j ， $k(x_i, y_j)$ 需要计算用来做预测，训练集中的 x_i 有可能有很多不同的。

这个子模块的这些类中允许嵌入 ϕ ，从而明确的与 $\phi(x_i)$ 一起工作， 这消除了使用 kernel 的需要和存储训练样本.

参考资料:

- [RR2007] “Random features for large-scale kernel machines” Rahimi, A. and Recht, B. - Advances in neural information processing 2007,
- [LS2010] “Random Fourier approximations for skewed multiplicative histogram kernels” Random Fourier approximations for skewed multiplicative histogram kernels - Lecture Notes for Computer Sciencd (DAGM)
- [VZ2010] “Efficient additive kernels via explicit feature maps” Vedaldi, A. and Zisserman, A. - Computer Vision and Pattern Recognition 2010
- [VVZ2010] “Generalized RBF feature maps for Efficient Detection” Vempati, S. and Vedaldi, A. and Zisserman, A. and Jawahar, CV - 2010

我们一直在努力

apachecn/sklearn-doc-zh



5.8. 成对的矩阵， 类别和核函数

校验者：@FontTian@numpy翻译者：@程威

The `sklearn.metrics.pairwise` 子模块实现了用于评估成对距离或样本集合之间的联系的实用程序。

本模块同时包含距离度量和核函数，对于这两者这里提供一个简短的总结。

距离度量是形如 `d(a, b)` 例如 `d(a, b) < d(a, c)` 如果对象 `a` 和 `b` 被认为“更加相似”相比于 `a` 和 `c` . 两个完全相同的目标的距离是零。最广泛使用的例子就是欧几里得距离。 为了保证是‘真实的’度量， 其必须满足以下条件：

1. 对于所有的 a 和 b, d(a, b) >= 0
2. 正定性：当且仅当 a = b时, d(a, b) == 0
3. 对称性：d(a, b) == d(b, a)
4. 三角不等式：d(a, c) <= d(a, b) + d(b, c)

核函数是相似度的标准. 如果对象 `a` 和 `b` 被认为“更加相似”相比对象 `a` 和 `c` ，那么 `s(a, b) > s(a, c)` . 核函数必须是半正定性的.

存在许多种方法将距离度量转换为相似度标准，例如核函数。 假定 `D` 是距离， and `S` 是核函数：

1. `S = np.exp(-D * gamma)` , 其中 `gamma` 的一种选择是 `1 / num_features`
2. `S = 1. / (D / np.max(D))`

X 的行向量和 Y 的行向量之间的距离可以用函数 pairwise_distances 进行计算。 如果 Y 被忽略，则 X 的所有行向量的成对距离就会被计算。 类似的，函数 pairwise.pairwise_kernels 可以使用不同的核函数 (kernel functions)来计算 X 和 Y 之间的 kernel。 请查看API获得更多详情。

```
>>> import numpy as np
>>> from sklearn.metrics import pairwise_distances
>>> from sklearn.metrics.pairwise import pairwise_kernels
>>> X = np.array([[2, 3], [3, 5], [5, 8]])
>>> Y = np.array([[1, 0], [2, 1]])
>>> pairwise_distances(X, Y, metric='manhattan')
array([[ 4.,  2.],
       [ 7.,  5.],
       [12., 10.]])
>>> pairwise_distances(X, metric='manhattan')
array([[0., 3., 8.],
       [3., 0., 5.],
       [8., 5., 0.]])
>>> pairwise_kernels(X, Y, metric='linear')
array([[ 2.,  7.],
       [ 3., 11.],
       [ 5., 18.]])
```

5.8.1. 余弦相似度

`cosine_similarity` 计算L2正则化的向量的点积. 也就是说, if x 和 y 都是行向量,, 它们的余弦相似度 k 定义为:

$$k(x, y) = \frac{xy^\top}{||x||||y||}$$

这被称为余弦相似度, 因为欧几里得(L2) 正则化将向量投影到单元球面内, 那么它们的点积就是被向量表示的点之间的角度。

这种核函数对于计算以tf-idf向量表示的文档之间的相似度是一个通常的选择. `cosine_similarity` 接受 `scipy.sparse` 矩阵. (注意到 `sklearn.feature_extraction.text` 中的tf-idf函数能计算归一化的向量, 在这种情况下 `cosine_similarity` 等同于 `linear_kernel`, 只是慢一点而已.)

> 参考资料:
>
> - C.D. Manning, P. Raghavan and H. Schütze (2008). Introduction to Information Retrieval. Cambridge University Press. http://nlp.stanford.edu/IR-book/html/htmledition/the-vector-space-model-for-scoring-1.html

5.8.2. 线性核函数

函数 `linear_kernel` 是计算线性核函数, 也就是一种在 `degree=1` 和 `coef0=0` (同质化) 情况下的 `polynomial_kernel` 的特殊形式. 如果 `x` 和 `y` 是列向量, 它们的线性核函数是:

$$k(x, y) = x^\top y$$

5.8.3. 多项式核函数

函数polynomial_kernel计算两个向量的d次方的多项式核函数. 多项式核函数代表着两个向量之间的相似度.概念上来说, 多项式核函数不仅考虑相同维度还考虑跨维度的向量的相似度。当被用在机器学习中的时候, 这可以原来代表着特征之间的 相互作用。

多项式函数定义为:

$$k(x, y) = (\gamma x^\top y + c_0)^d$$

其中:

- `x`, `y` 是输入向量
- `d` 核函数维度

如果 $c_0 = 0$ 那么核函数就被定义为同质化的.

5.8.4. Sigmoid 核函数

函数 `sigmoid_kernel` 计算两个向量之间的S型核函数. S型核函数也被称为双曲切线或者 多层感知机(因为在神经

网络领域，它经常被当做激活函数)． S型核函数定义为：

$$k(x, y) = \tanh(\gamma x^\top y + c_0)$$

其中：

- `x`， `y` 是输入向量
- γ 是斜度
- c_0 是截距

5.8.5． RBF 核函数

函数 `rbf_kernel` 计算计算两个向量之间的径向基函数核（RBF）。 其定义为：

$$k(x, y) = \exp(-\gamma ||x - y||^2)$$

其中 `x` 和 `y` 是输入向量．如果 $\gamma = \sigma^{-2}$ 核函数就变成方差为 σ^2 的高斯核函数．

5.8.6． 拉普拉斯核函数

函数 `laplacian_kernel` 是一种径向基函数核的变体，定义为：

$$k(x, y) = \exp(-\gamma ||x - y||_1)$$

其中 `x` 和 `y` 是输入向量 并且 $||x - y||_1$ 是输入向量之间的曼哈顿距离．

已被证明在机器学习中运用到无噪声数据中是有用的． 可见例如 Machine learning for quantum mechanics in a nutshell.

5.8.7． 卡方核函数

在计算机视觉应用中训练非线性支持向量机时，卡方核函数是一种非常流行的选择．它能以 `chi2_kernel` 计算然后将参数 `kernel="precomputed"` 传递到 `sklearn.svm.SVC` ：

```
>>> from sklearn.svm import SVC
>>> from sklearn.metrics.pairwise import chi2_kernel
>>> X = [[0, 1], [1, 0], [.2, .8], [.7, .3]]
>>> y = [0, 1, 0, 1]
>>> K = chi2_kernel(X, gamma=.5)
>>> K
array([[1.        , 0.36787944, 0.89483932, 0.58364548],
       [0.36787944, 1.        , 0.51341712, 0.83822343],
       [0.89483932, 0.51341712, 1.        , 0.7768366 ],
       [0.58364548, 0.83822343, 0.7768366 , 1.        ]])
```

```
11.
12. >>> svm = SVC(kernel='precomputed').fit(K, y)
13. >>> svm.predict(K)
14. array([0, 1, 0, 1])
```

也可以直接使用 `kernel` 变量：

```
1. >>> svm = SVC(kernel=chi2_kernel).fit(X, y)
2. >>> svm.predict(X)
3. array([0, 1, 0, 1])
```

卡方核函数定义为

$$k(x, y) = \exp\left(-\gamma \sum_i \frac{(x[i] - y[i])^2}{x[i] + y[i]}\right)$$

数据假定为非负的，并且已经以L1正则化。 归一化随着与卡方平方距离的连接而被合理化，其是离散概率分布之间的距离。

卡方核函数最常用于可视化词汇的矩形图。

> 参考资料：
>
> - Zhang, J. and Marszalek, M. and Lazebnik, S. and Schmid, C. Local features and kernels for classification of texture and object categories: A comprehensive study International Journal of Computer Vision 2007 http://research.microsoft.com/en-us/um/people/manik/projects/trade-off/papers/ZhangIJCV06.pdf

我们一直在努力

apachecn/sklearn-doc-zh

加入QQ群

5.9. 预测目标（y）的转换

校验者：@FontTian@numpy翻译者：@程威

本章要介绍的这些变换器不是被用于特征的，而是只被用于变换监督学习的目标。 如果你希望变换预测目标以进行学习，但是在原始空间中评估模型，请参考回归中的目标转换 。

5.9.1. 标签二值化

`LabelBinarizer` 是一个用来从多类别列表创建标签矩阵的工具类：

```
>>> from sklearn import preprocessing
>>> lb = preprocessing.LabelBinarizer()
>>> lb.fit([1, 2, 6, 4, 2])
LabelBinarizer(neg_label=0, pos_label=1, sparse_output=False)
>>> lb.classes_
array([1, 2, 4, 6])
>>> lb.transform([1, 6])
array([[1, 0, 0, 0],
 [0, 0, 0, 1]])
```

对于多类别是实例，可以使用 `MultiLabelBinarizer` ：

```
>>> lb = preprocessing.MultiLabelBinarizer()
>>> lb.fit_transform([(1, 2), (3,)])
array([[1, 1, 0],
 [0, 0, 1]])
>>> lb.classes_
array([1, 2, 3])
```

5.9.2. 标签编码

`LabelEncoder` 是一个可以用来将标签规范化的工具类，它可以将标签的编码值范围限定在[0,n_classes-1]. 这在编写高效的Cython程序时是非常有用的. `LabelEncoder` 可以如下使用：

```
>>> from sklearn import preprocessing
>>> le = preprocessing.LabelEncoder()
>>> le.fit([1, 2, 2, 6])
LabelEncoder()
>>> le.classes_
array([1, 2, 6])
>>> le.transform([1, 1, 2, 6])
array([0, 0, 1, 2])
>>> le.inverse_transform([0, 0, 1, 2])
array([1, 1, 2, 6])
```

当然，它也可以用于非数值型标签的编码转换成数值标签（只要它们是可哈希并且可比较的）：

```
>>> le = preprocessing.LabelEncoder()
>>> le.fit(["paris", "paris", "tokyo", "amsterdam"])
LabelEncoder()
>>> list(le.classes_)
['amsterdam', 'paris', 'tokyo']
>>> le.transform(["tokyo", "tokyo", "paris"])
array([2, 2, 1])
>>> list(le.inverse_transform([2, 2, 1]))
['tokyo', 'tokyo', 'paris']
```

我们一直在努力

apachecn/sklearn-doc-zh



scikit-learn 教程 0.21.x

- 使用 scikit-learn 介绍机器学习
- 关于科学数据处理的统计学习教程
- 处理文本数据
- 选择正确的评估器(estimator.md)
- 外部资源，视频和谈话

使用 scikit-learn 介绍机器学习

校验者：@小瑶@hlxstc@BWM-蜜蜂@小瑶@Loopy翻译者：@李昊伟@…

内容提要

在本节中，我们介绍一些在使用 scikit-learn 过程中用到的 机器学习 词汇，并且给出一些例子阐释它们。

机器学习：问题设置

一般来说，一个学习问题通常会考虑一系列 n 个 样本）数据，然后尝试预测未知数据的属性。 如果每个样本是 多个属性的数据 （比如说是一个多维记录），就说它有许多“属性”，或称 **features(特征)** 。

我们可以将学习问题分为几大类：

- 监督学习 ， 其中数据带有一个附加属性，即我们想要预测的结果值（ 点击此处 转到 scikit-learn 监督学习页面）。这个问题可以是：
- 分类 ： 样本属于两个或更多个类，我们想从已经标记的数据中学习如何预测未标记数据的类别。 分类问题的一个例子是手写数字识别，其目的是将每个输入向量分配给有限数目的离散类别之一。 我们通常把分类视作监督学习的一个离散形式（区别于连续形式），从有限的类别中，给每个样本贴上正确的标签。
- 回归 ： 如果期望的输出由一个或多个连续变量组成，则该任务称为 回归 。 回归问题的一个例子是预测鲑鱼的长度是其年龄和体重的函数。
- 无监督学习， 其中训练数据由没有任何相应目标值的一组输入向量x组成。这种问题的目标可能是在数据中发现彼此类似的示例所聚成的组，这种问题称为 聚类 ， 或者，确定输入空间内的数据分布，称为 密度估计 ，又或从高维数据投影数据空间缩小到二维或三维以进行 可视化 （点击此处 转到 scikit-learn 无监督学习页面）。

> 训练集和测试集
>
> 机器学习是从数据的属性中学习，并将它们应用到新数据的过程。 这就是为什么机器学习中评估算法的普遍实践是把数据分割成 训练集 （我们从中学习数据的属性）和 测试集 （我们测试这些性质）。

加载示例数据集

scikit-learn 提供了一些标准数据集，例如 用于分类的 iris 和 digits 数据集 和 波士顿房价回归数据集 .

在下文中，我们从我们的 shell 启动一个 Python 解释器，然后加载 `iris` 和 `digits` 数据集。我们的符号约定是 `$` 表示 shell 提示符，而 `>>>` 表示 Python 解释器提示符：

```
$ python
>>> from sklearn import datasets
>>> iris = datasets.load_iris()
>>> digits = datasets.load_digits()
```

数据集是一个类似字典的对象，它保存有关数据的所有数据和一些元数据。 该数据存储在 `.data` 成员中，它是 `n_samples, n_features` 数组。 在监督问题的情况下，一个或多个响应变量存储在 `.target` 成员中。 有关不同数据集的更多详细信息，请参见 专用数据集部分 。

例如，在数字数据集的情况下， `digits.data` 使我们能够得到一些用于分类的样本特征：

```
>>> print(digits.data)
[[  0.   0.   5. ...,   0.   0.   0.]
 [  0.   0.   0. ...,  10.   0.   0.]
 [  0.   0.   0. ...,  16.   9.   0.]
 ...,
 [  0.   0.   1. ...,   6.   0.   0.]
 [  0.   0.   2. ...,  12.   0.   0.]
 [  0.   0.  10. ...,  12.   1.   0.]]
```

并且 `digits.target` 表示了数据集内每个数字的真实类别，也就是我们期望从每个手写数字图像中学得的相应的数字标记：

```
>>> digits.target
array([0, 1, 2, ..., 8, 9, 8])
```

> 数据数组的形状
>
> 数据总是二维数组，形状 `(n_samples, n_features)` ，尽管原始数据可能具有不同的形状。 在数字的情况下，每个原始样本是形状 `(8, 8)` 的图像，可以使用以下方式访问：
>
> ```
> >> digits.images[0]  array([[  0.,   0.,   5.,  13.,   9.,   1.,   0.,   0.],   [  0.,   0.,  13.,  15.,
> 10.,  15.,   5.,   0.],   [  0.,   3.,  15.,   2.,   0.,  11.,   8.,   0.],   [  0.,   4.,  12.,   0.,
> 0.,   8.,   8.,   0.],   [  0.,   5.,   8.,   0.,   0.,   9.,   8.,   0.],   [  0.,   4.,  11.,   0.,
> 1.,  12.,   7.,   0.],   [  0.,   2.,  14.,   5.,  10.,  12.,   0.,   0.],   [  0.,   0.,   6.,  13.,
> 10.,   0.,   0.,   0.]])
> ```
>
> 该 数据集上的简单示例 说明了如何从原始数据开始调整，形成可以在 scikit-learn 中使用的数据。
>
> 从外部数据集加载
>
> 要从外部数据集加载，请参阅 加载外部数据集 。

学习和预测

在数字数据集的情况下，任务是给出图像来预测其表示的数字。 我们给出了 10 个可能类（数字 0 到 9）中的每一个的样本，我们在这些类上 拟合 一个 估计器 ，以便能够 预测 未知的样本所属的类。

在 scikit-learn 中，分类的估计器是一个 Python 对象，它实现了 `fit(X, y)` 和 `predict(T)` 等方法。

估计器的一个例子类 `sklearn.svm.SVC` ，实现了 支持向量分类 。 估计器的构造函数以相应模型的参数为参数，但目前我们将把估计器视为黑箱即可：

```
>>> from sklearn import svm
```

```
>>> clf = svm.SVC(gamma=0.001, C=100.)
```

> 选择模型的参数
>
> 在这个例子中，我们手动设置 `gamma` 值。不过，通过使用 网格搜索 及 交叉验证 等工具，可以自动找到参数的良好值。

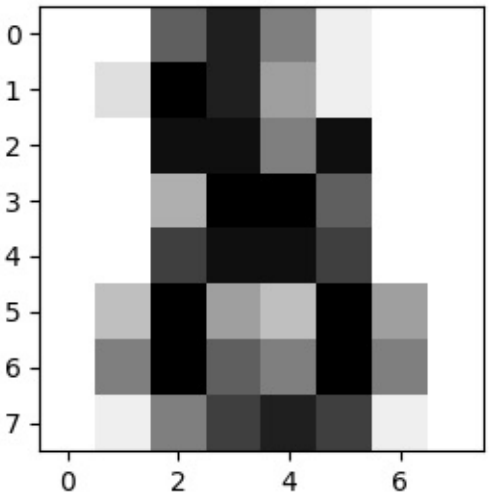
我们把我们的估计器实例命名为 `clf` ，因为它是一个分类器（classifier）。它现在必须拟合模型，也就是说，它必须从模型中 *learn*（学习）。 这是通过将我们的训练集传递给 `fit` 方法来完成的。作为一个训练集，让我们使用数据集中除最后一张以外的所有图像。 我们用 `[:-1]` Python 语法选择这个训练集，它产生一个包含 `digits.data` 中除最后一个条目（entry）之外的所有条目的新数组

```
>>> clf.fit(digits.data[:-1], digits.target[:-1])
SVC(C=100.0, cache_size=200, class_weight=None, coef0=0.0,
  decision_function_shape='ovr', degree=3, gamma=0.001, kernel='rbf',
  max_iter=-1, probability=False, random_state=None, shrinking=True,
  tol=0.001, verbose=False)
```

现在你可以预测新的值，特别是我们可以向分类器询问 `digits` 数据集中最后一个图像（没有用来训练的一条实例）的数字是什么：

```
>>> clf.predict(digits.data[-1:])
array([8])
```

相应的图像如下：



正如你所看到的，这是一项具有挑战性的任务：图像分辨率差。你是否认同这个分类？

这个分类问题的一个完整例子可以作为一个例子来运行和学习： 识别手写数字。 Recognizing hand-written digits.

模型持久化

可以通过使用 Python 的内置持久化模块（即 pickle ）将模型保存：

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

>>> import pickle
>>> s = pickle.dumps(clf)
>>> clf2 = pickle.loads(s)
>>> clf2.predict(X[0:1])
array([0])
>>> y[0]
0
```

在scikit的具体情况下，使用 joblib 替换 pickle（ `joblib.dump` & `joblib.load` ）可能会更有趣，这对大数据更有效，但只能序列化 (pickle) 到磁盘而不是字符串变量：

```
>>> from joblib import dump, load
>>> dump(clf, 'filename.joblib')
```

之后，您可以加载已保存的模型（可能在另一个 Python 进程中）：

```
>>> clf = load('filename.joblib')
```

> 注意
>
> `joblib.dump` 以及 `joblib.load` 函数也接受 file-like（类文件） 对象而不是文件名。有关 Joblib 的数据持久化的更多信息，请 点击此处 。

请注意，pickle 有一些安全性和维护性问题。有关使用 scikit-learn 的模型持久化的更多详细信息，请参阅 模型持久化 部分。

规定

scikit-learn 估计器遵循某些规则，使其行为更可预测。在公共术语表和API元素中有更详细的描述。

类型转换

除非特别指定，输入将被转换为 `float64`

```
>>> import numpy as np
>>> from sklearn import random_projection

```

```
>>> rng = np.random.RandomState(0)
>>> X = rng.rand(10, 2000)
>>> X = np.array(X, dtype='float32')
>>> X.dtype
dtype('float32')

>>> transformer = random_projection.GaussianRandomProjection()
>>> X_new = transformer.fit_transform(X)
>>> X_new.dtype
dtype('float64')
```

在这个例子中，`X` 原本是 `float32` ，被 `fit_transform(X)` 转换成 `float64` 。

回归目标被转换为 `float64` ，但分类目标维持不变：

```
>>> from sklearn import datasets
>>> from sklearn.svm import SVC
>>> iris = datasets.load_iris()
>>> clf = SVC()
>>> clf.fit(iris.data, iris.target)
SVC(C=1.0, cache_size=200, class_weight=None, coef0=0.0,
 decision_function_shape='ovr', degree=3, gamma='auto', kernel='rbf',
 max_iter=-1, probability=False, random_state=None, shrinking=True,
 tol=0.001, verbose=False)

>>> list(clf.predict(iris.data[:3]))
[0, 0, 0]

>>> clf.fit(iris.data, iris.target_names[iris.target])
SVC(C=1.0, cache_size=200, class_weight=None, coef0=0.0,
 decision_function_shape='ovr', degree=3, gamma='auto', kernel='rbf',
 max_iter=-1, probability=False, random_state=None, shrinking=True,
 tol=0.001, verbose=False)

>>> list(clf.predict(iris.data[:3]))
['setosa', 'setosa', 'setosa']
```

这里，第一个 `predict()` 返回一个整数数组，因为在 `fit` 中使用了 `iris.target` （一个整数数组）。 第二个 `predict()` 返回一个字符串数组，因为 `iris.target_names` 是一个字符串数组。

再次训练和更新参数

估计器的超参数可以通过 `sklearn.pipeline.Pipeline.set_params` 方法在实例化之后进行更新。 调用 `fit()` 多次将覆盖以前的 `fit()` 所学到的参数：

```
>>> import numpy as np
>>> from sklearn.datasets import load_iris
>>> from sklearn.svm import SVC
>>> X, y = load_iris(return_X_y=True)

>>> clf = SVC()
```

```
>>> clf.set_params(kernel='linear').fit(X, y)
SVC(C=1.0, cache_size=200, class_weight=None, coef0=0.0,
  decision_function_shape='ovr', degree=3, gamma='auto_deprecated',
  kernel='linear', max_iter=-1, probability=False, random_state=None,
  shrinking=True, tol=0.001, verbose=False)
>>> clf.predict(X[:5])
array([0, 0, 0, 0, 0])

>>> clf.set_params(kernel='rbf', gamma='scale').fit(X, y)
SVC(C=1.0, cache_size=200, class_weight=None, coef0=0.0,
  decision_function_shape='ovr', degree=3, gamma='scale', kernel='rbf',
  max_iter=-1, probability=False, random_state=None, shrinking=True,
  tol=0.001, verbose=False)
>>> clf.predict(X[:5])
array([0, 0, 0, 0, 0])
```

在这里，估计器被 `SVC()` 构造之后，默认内核 `rbf` 首先被改变到 `linear` ，然后改回到 `rbf` 重新训练估计器并进行第二次预测。

多分类与多标签拟合

当使用 `多类分类器` 时，执行的学习和预测任务取决于参与训练的目标数据的格式：

```
>>> from sklearn.svm import SVC
>>> from sklearn.multiclass import OneVsRestClassifier
>>> from sklearn.preprocessing import LabelBinarizer

>>> X = [[1, 2], [2, 4], [4, 5], [3, 2], [3, 1]]
>>> y = [0, 0, 1, 1, 2]

>>> classif = OneVsRestClassifier(estimator=SVC(random_state=0))
>>> classif.fit(X, y).predict(X)
array([0, 0, 1, 1, 2])
```

在上述情况下，分类器使用含有多个标签的一维数组训练模型，由于每个样本只对应一个类别标签，因此 `predict()` 方法可提供相应的多标签预测。分类器也可以通过标签二值化后的二维数组来训练：

```
>>> y = LabelBinarizer().fit_transform(y)
>>> classif.fit(X, y).predict(X)
array([[1, 0, 0],
       [1, 0, 0],
       [0, 1, 0],
       [0, 0, 0],
       [0, 0, 0]])
```

这里， 分类器 `fit()` 方法在 `y` 的二维二元标签表示上执行， 每个样本可同时属于两种类别，同时具有两个种类的标签， 所以要使用 `LabelBinarizer` 将目标向量 `y` 转化成二值化后的二维数组。在这种情况下， `predict()` 返回一个多标签预测相应的 二维 数组。

请注意，第四个和第五个实例返回全零向量，表明它们不能匹配用来训练中的目标标签中的任意一个。使用多标签输出，类似地可以为一个实例分配多个标签：

```
>> from sklearn.preprocessing import MultiLabelBinarizer
>> y = [[0, 1], [0, 2], [1, 3], [0, 2, 3], [2, 4]]
>> y = MultiLabelBinarizer().fit_transform(y)
>> classif.fit(X, y).predict(X)
array([[1, 1, 0, 0, 0],
       [1, 0, 1, 0, 0],
       [0, 1, 0, 1, 0],
       [1, 0, 1, 1, 0],
       [0, 0, 1, 0, 1]])
```

在这种情况下，用来训练分类器的多个向量被赋予多个标记， `MultiLabelBinarizer` 用来二值化多个标签产生二维数组并用来训练。 `predict()` 函数返回带有多个标签的二维数组作为每个实例的结果。

我们一直在努力

apachecn/sklearn-doc-zh



关于科学数据处理的统计学习教程

校验者： 待校验翻译者： @Loopy

> 统计学习
>
> 随着科学实验数据集规模的快速增长，机器学习机器学习技术正变得越来越重要。它能处理的问题主要包括：建立连接不同观测值的预测函数，对观测值进行分类，或者分析未标记数据集中的结构。
>
> 本教程将探讨统计学习。以统计推断为目标，使用机器学习技术，根据手头的数据来得出结论。
>
> Scikit-learn是一个Python模块，它将科学计算的Python包(NumPy, SciPy, matplotlib)集成到了一起。

- 机器学习： scikit-learn 中的设置以及预估对象
  - 数据集
  - 预估对象
- 监督学习：从高维观察预测输出变量
  - 最近邻和维度惩罚
  - 线性模型：从回归到稀疏
  - 支持向量机(SVMs)
- 模型选择：选择估计量及其参数
  - 分数和交叉验证分数
  - 交叉验证生成器
  - 网格搜索和交叉验证估计量
- 无监督学习： 寻求数据表示
  - 聚类： 对样本数据进行分组
  - 分解： 将一个信号转换成多个成份并且加载
- 把它们放在一起
  - 模型管道化
  - 用特征面进行人脸识别
  - 开放性问题： 股票市场结构
- 寻求帮助
  - 项目邮件列表
  - 机器学习从业者的 Q&A 社区

我们一直在努力

apachecn/sklearn-doc-zh



机器学习: scikit-learn 中的设置以及预估对象

校验者: @Kyrie@片刻翻译者: @冰块

数据集

Scikit-learn可以从一个或者多个数据集中学习信息，这些数据集合可表示为2维阵列，也可认为是一个列表。列表的第一个维度代表 样本 ，第二个维度代表 特征 （每一行代表一个样本，每一列代表一种特征）。

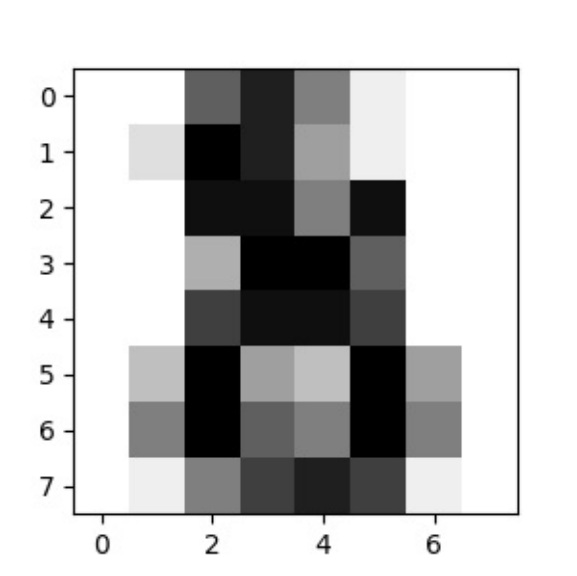
样例: iris 数据集（鸢尾花卉数据集）

```
>> from sklearn import datasets>> iris = datasets.load_iris()>> data = iris.data>> data.shape(150, 4)
```

这个数据集包含150个样本，每个样本包含4个特征: 花萼长度，花萼宽度，花瓣长度，花瓣宽度，详细数据可以通过 `iris.DESCR` 查看。

如果原始数据不是 `(n_samples, n_features)` 的形状时，使用之前需要进行预处理以供scikit-learn使用。

数据预处理样例:digits数据集(手写数字数据集)



digits数据集包含1797个手写数字的图像，每个图像为8*8像素

```
>> digits = datasets.load_digits()>> digits.images.shape(1797, 8, 8)>> import matplotlib.pyplot as plt>>
plt.imshow(digits.images[-1], cmap=plt.cm.gray_r)<matplotlib.image.AxesImage object at …>
```

为了在scikit中使用这一数据集，需要将每一张8×8的图像转换成长度为64的特征向量

```
>> data = digits.images.reshape((digits.images.shape[0], -1))
```

预估对象

拟合数据: scikit-learn实现最重要的一个API是 `estimator` 。estimators是基于数据进行学习的任何对象，它可以是一个分类器，回归或者是一个聚类算法，或者是从原始数据中提取/过滤有用特征的变换器。

所有的拟合模型对象拥有一个名为 `fit` 的方法，参数是一个数据集（通常是一个2维列表）：

```
>>> estimator.fit(data)
```

拟合模型对象构造参数: 在创建一个拟合模型时，可以设置相关参数，在创建之后也可以修改对应的参数:

```
>>> estimator = Estimator(param1=1, param2=2)
>>> estimator.param1
1
```

拟合参数: 当拟合模型完成对数据的拟合之后，可以从拟合模型中获取拟合的参数结果，所有拟合完成的参数均以下划线(_)作为结尾:

```
>>> estimator.estimated_param_
```

我们一直在努力

apachecn/sklearn-doc-zh

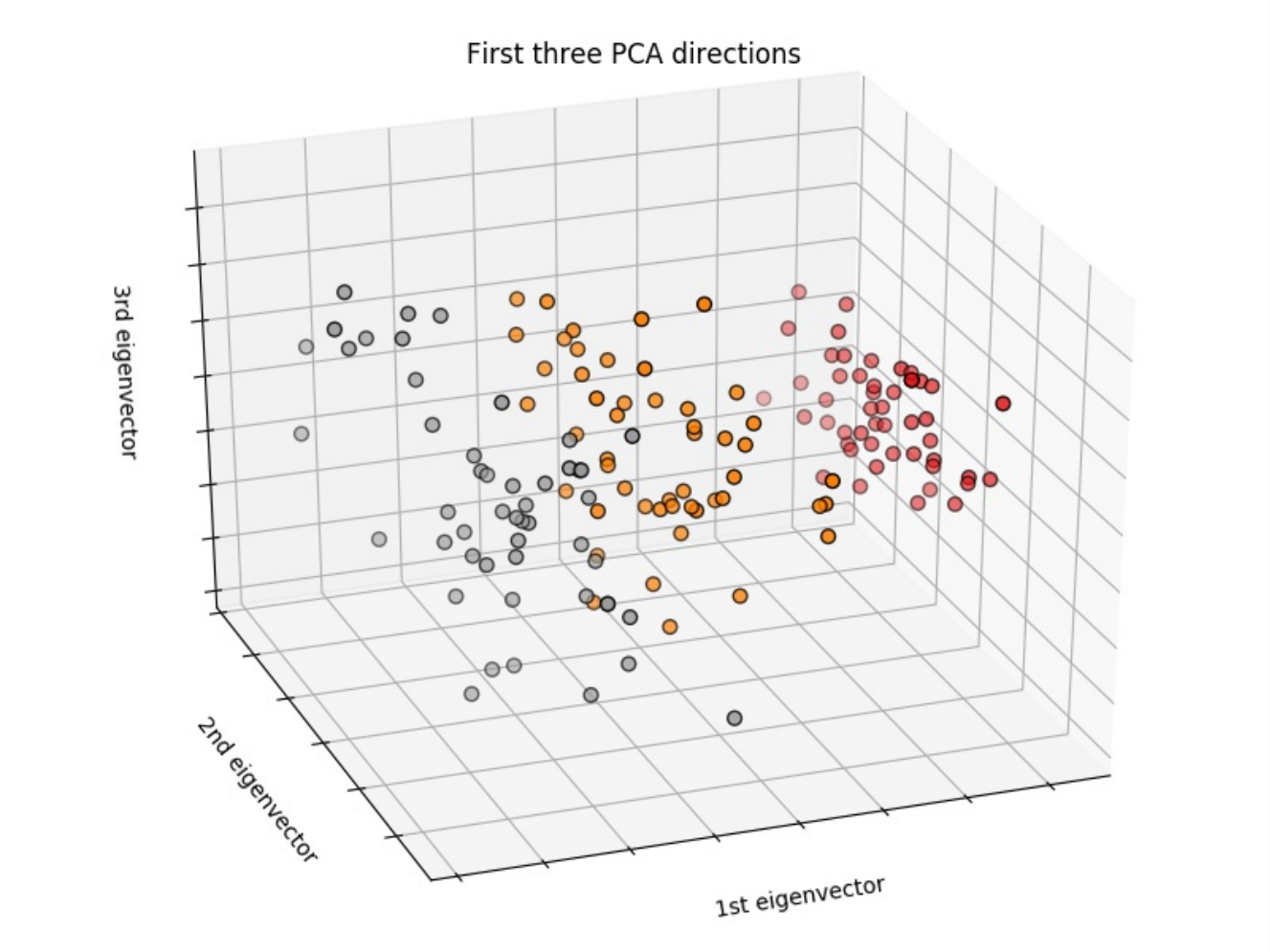


监督学习：从高维观察预测输出变量

校验者：@Kyrie@片刻@Loopy翻译者：@森系

> 监督学习解决的问题
>
> 监督学习 在于学习两个数据集的联系：观察数据 `X` 和我们正在尝试预测的额外变量 `y` （通常称“目标”或“标签”），而且通常是长度为 `n_samples` 的一维数组。
>
> scikit-learn 中所有监督的估计量 都有一个用来拟合模型的 `fit(X, y)` 方法，和根据给定的没有标签观察值 `X` 返回预测的带标签的 `y` 的 `predict(X)` 方法。
>
> 词汇：分类和回归
>
> 如果预测任务是为了将观察值分类到有限的标签集合中，换句话说，就是给观察对象命名，那任务就被称为 分类 任务。另外，如果任务是为了预测一个连续的目标变量，那就被称为 回归 任务。
>
> 当在 scikit-learn 中进行分类时，`y` 是一个整数或字符型的向量。
>
> 注：可以查看用 scikit-learn 进行机器学习介绍 快速了解机器学习中的基础词汇。

最近邻和维度惩罚

> 鸢尾属植物分类
>
> 
>

> 鸢尾属植物数据集是根据花瓣长度、花瓣度度、萼片长度和萼片宽度4个特征对3种不同类型的鸢尾属植物进行分类：

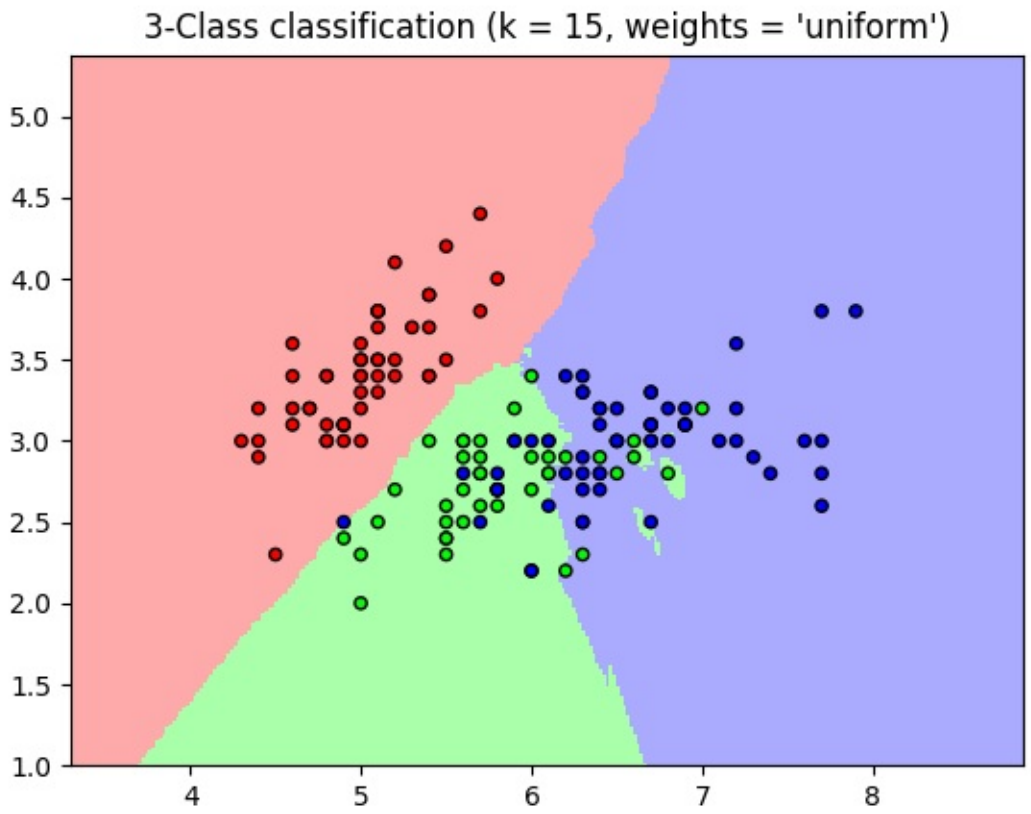
```
>> import numpy as np>> from sklearn import datasets>> iris = datasets.load_iris()>> iris_X = iris.data>> iris_y = iris.target>> np.unique(iris_y)array([0, 1, 2])
```

K近邻分类器

最近邻：也许是最简单的分类器：给定一个新的观察值 `X_test`，用最接近的特征向量在训练集(比如，用于训练估计器的数据)找到观察值。(请看 Scikit-learn 在线学习文档的 最近邻章节 获取更多关于这种分类器的信息)

> 训练集和测试集
>
> 当用任意的学习算法进行实验时，最重要的就是不要在用于拟合估计器的数据上测试一个估计器的预期值，因为这不会评估在 新数据 上估计器的执行情况。这也是数据集经常被分为 训练 和 测试 数据的原因。

KNN(k 最近邻)分类器例子：




```
>>> # 将鸢尾属植物数据集分解为训练集和测试集
>>> # 随机排列，用于使分解的数据随机分布
>>> np.random.seed(0)
>>> indices = np.random.permutation(len(iris_X))
>>> iris_X_train = iris_X[indices[:-10]]
>>> iris_y_train = iris_y[indices[:-10]]
>>> iris_X_test  = iris_X[indices[-10:]]
>>> iris_y_test  = iris_y[indices[-10:]]
>>> # 创建和拟合一个最近邻分类器
```

```
>>> from sklearn.neighbors import KNeighborsClassifier
>>> knn = KNeighborsClassifier()
>>> knn.fit(iris_X_train, iris_y_train)
KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='minkowski',
 metric_params=None, n_jobs=1, n_neighbors=5, p=2,
 weights='uniform')
>>> knn.predict(iris_X_test)
array([1, 2, 1, 0, 0, 0, 2, 1, 2, 0])
>>> iris_y_test
array([1, 1, 1, 0, 0, 0, 2, 1, 2, 0])
```

维度惩罚

为了使一个估计器有效，你需要邻接点间的距离小于一些值：d，这取决于具体问题。在一维中，这需要平均 *n sim 1/d* 点。在上文 k-NN 例子中，如果数据只是由一个0到1的特征值和 n 训练观察值所描述，那么新数据将不会超过 $1/n$。因此，最近邻决策规则会很有效率，因为与类间特征变量范围相比， $1/n$ 很小。

如果特征数是 p，你现在就需要 $n \sim 1/d^p$ 点。也就是说我们在一维 $[0, 1]$ 空间里需要10个点，在 p 维里就需要 10^p 个点。当 p 增大时，为了得到一个好的估计器，相应的训练点数量就需要成倍增大。

比如，如果每个点只是单个数字(8个字节)，那么一个 k-NN 估计器在一个非常小的 $p \sim 20$ 维度下就需要比现在估计的整个互联网的大小(±1000 艾字节或更多)还要多的训练数据。

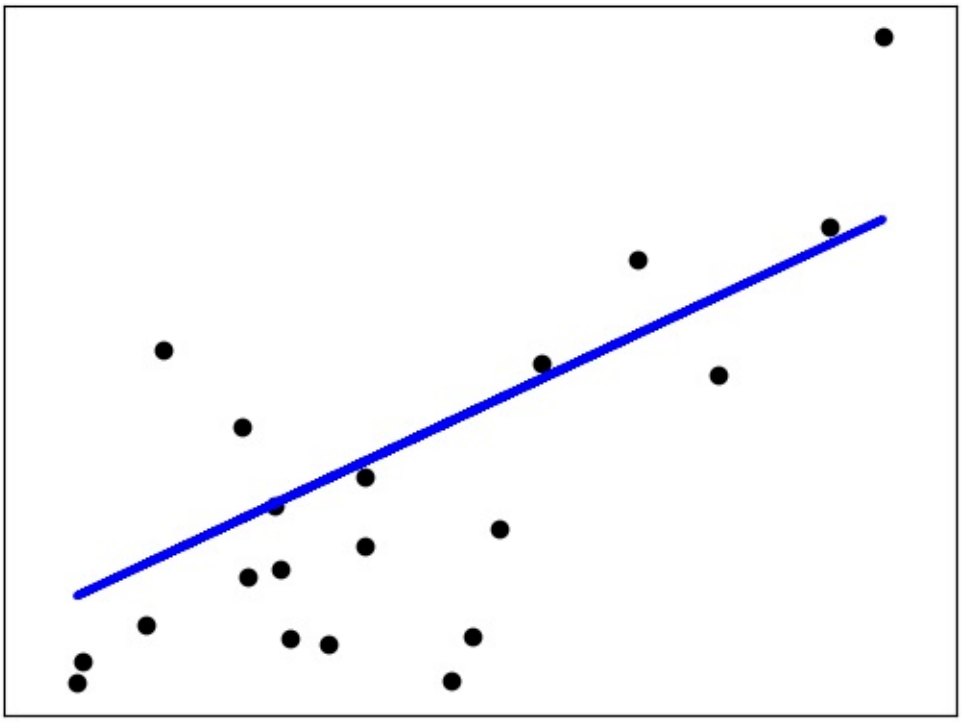
这叫 维度惩罚，是机器学习领域的核心问题。

线性模型：从回归到稀疏

> 糖尿病数据集
>
> 糖尿病数据集包括442名患者的10个生理特征(年龄，性别，体重，血压)，和一年后的疾病级别指标：
>
> ```
> >> diabetes = datasets.load_diabetes()>> diabetes_X_train = diabetes.data[:-20]>> diabetes_X_test  =
> diabetes.data[-20:]>> diabetes_y_train = diabetes.target[:-20]>> diabetes_y_test  =
> diabetes.target[-20:]
> ```
>
> 任务是使用生理特征来预测疾病级别。

线性回归

`LinearRegression`，最简单的拟合线性模型形式，是通过调整数据集的一系列参数令残差平方和尽可能小。



Linear models: $y = X\beta + \epsilon$

- X：数据
- y：目标变量
- β：回归系数
- ϵ：观察噪声

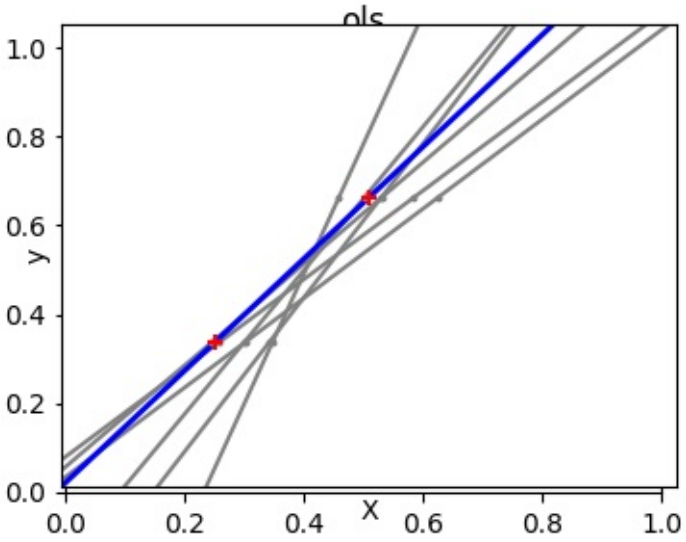
```
>>> from sklearn import linear_model
>>> regr = linear_model.LinearRegression()
>>> regr.fit(diabetes_X_train, diabetes_y_train)
LinearRegression(copy_X=True, fit_intercept=True, n_jobs=1, normalize=False)
>>> print(regr.coef_)
[   0.30349955 -237.63931533  510.53060544  327.73698041 -814.13170937
 492.81458798  102.84845219  184.60648906  743.51961675   76.09517222]

>>> # 均方误差
>>> np.mean((regr.predict(diabetes_X_test)-diabetes_y_test)**2)
2004.56760268...

>>> # 方差分数：1 是完美的预测
>>> # 0 意味着 X 和 y 之间没有线性关系。
>>> regr.score(diabetes_X_test, diabetes_y_test)
0.5850753022690...
```

收缩

如果每个维度的数据点很少，观察噪声就会导致很大的方差：

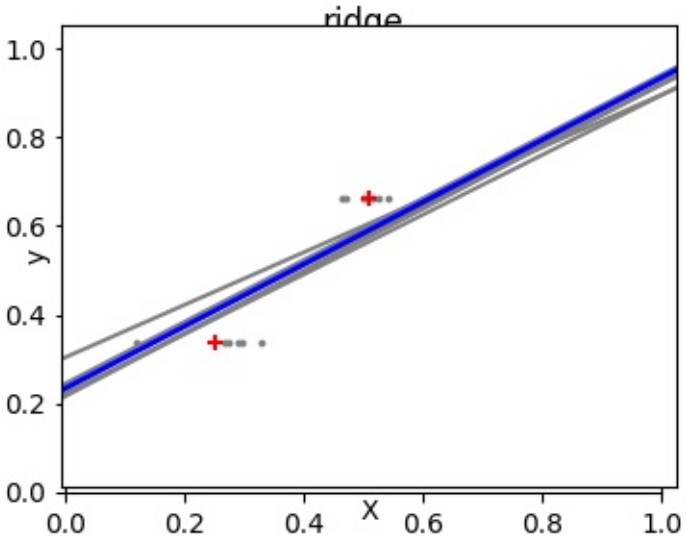



```
>>> X = np.c_[ .5, 1].T
>>> y = [.5, 1]
>>> test = np.c_[ 0, 2].T
>>> regr = linear_model.LinearRegression()

>>> import matplotlib.pyplot as plt
>>> plt.figure()

>>> np.random.seed(0)
>>> for _ in range(6):
...     this_X = .1*np.random.normal(size=(2, 1)) + X
...     regr.fit(this_X, y)
...     plt.plot(test, regr.predict(test))
...     plt.scatter(this_X, y, s=3)
```

高纬统计学习中的一个解决方法是 *收缩* 回归系数到0：任何两个随机选择的观察值数据集都很可能是不相关的。这称为岭回归：




```
>>> regr = linear_model.Ridge(alpha=.1)

>>> plt.figure()

>>> np.random.seed(0)
>>> for _ in range(6):
...     this_X = .1*np.random.normal(size=(2, 1)) + X
...     regr.fit(this_X, y)
...     plt.plot(test, regr.predict(test))
...     plt.scatter(this_X, y, s=3)
```

这是 **bias/variance tradeoff** 中的一个例子：岭参数 `alpha` 越大，偏差越大，方差越小。

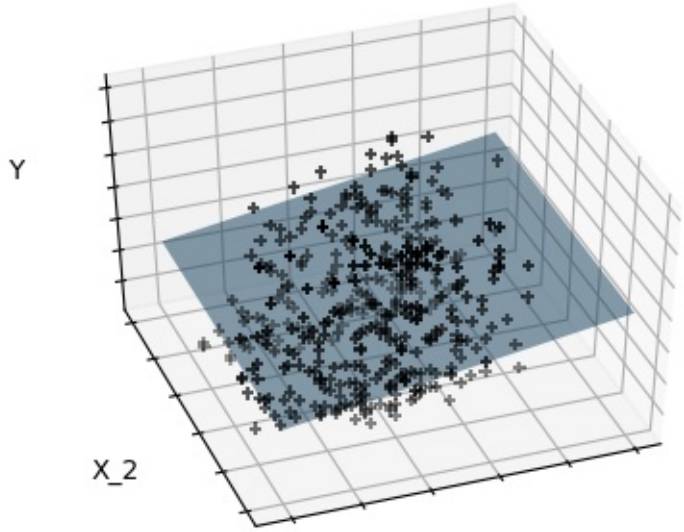
我们可以选择 `alpha` 来最小化排除错误，这里使用糖尿病数据集而不是人为数据：

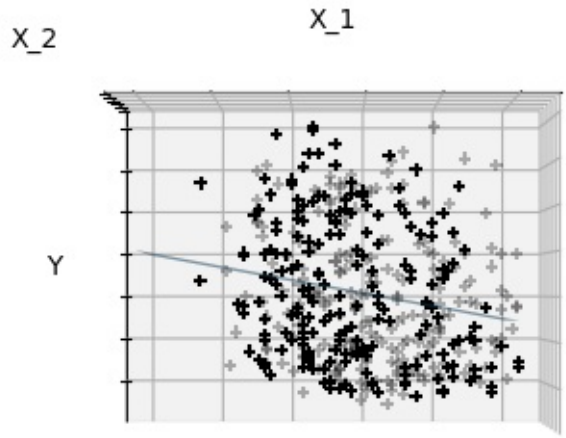
```
>>> alphas = np.logspace(-4, -1, 6)
>>> from __future__ import print_function
>>> print([regr.set_params(alpha=alpha
...             ).fit(diabetes_X_train, diabetes_y_train,
...             ).score(diabetes_X_test, diabetes_y_test) for alpha in alphas])
[0.5851110683883..., 0.5852073015444..., 0.5854677540698..., 0.5855512036503..., 0.5830717085554...,
0.57058999437...]
```

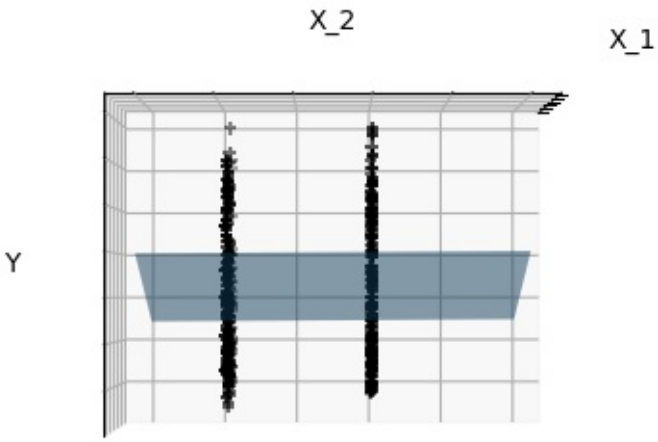
> 注意：捕获拟合参数噪声使得模型不能归纳新的数据称为过拟合。岭回归产生的偏差被称为 正则化。

稀疏

只拟合特征1和2







> 注意：整个糖尿病数据集包括11个维度(10个特征维度和1个目标变量)。很难直观地表示出来，但是记住那是一个比较 空 的空间可能比较有用。

我们可以看到，尽管特征2在整个模型占有一个很大的系数，但是当考虑特征1时，其对 `y` 的影响就较小了。

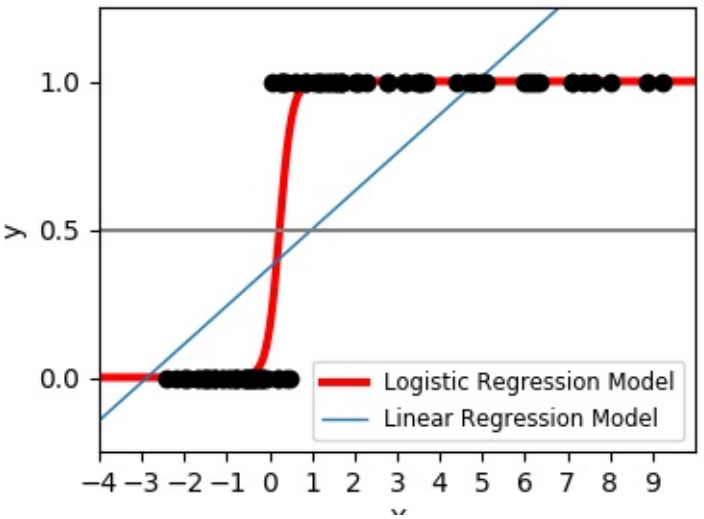
为了提高问题的条件(比如，缓解维度惩罚)，只选择信息特征和设置无信息时就会变得有趣，比如特征2到0。岭回归

会减小他们的值，但不会减到0.另一种抑制方法，称为 Lasso （最小绝对收缩和选择算子），可以把一些系数设为0。这些方法称为 稀疏法，稀疏可以看作是奥卡姆剃刀的应用：模型越简单越好。

```
>>> regr = linear_model.Lasso()
>>> scores = [regr.set_params(alpha=alpha
...             ).fit(diabetes_X_train, diabetes_y_train
...             ).score(diabetes_X_test, diabetes_y_test)
...        for alpha in alphas]
>>> best_alpha = alphas[scores.index(max(scores))]
>>> regr.alpha = best_alpha
>>> regr.fit(diabetes_X_train, diabetes_y_train)
Lasso(alpha=0.025118864315095794, copy_X=True, fit_intercept=True,
 max_iter=1000, normalize=False, positive=False, precompute=False,
 random_state=None, selection='cyclic', tol=0.0001, warm_start=False)
>>> print(regr.coef_)
[   0.         -212.43764548  517.19478111  313.77959962 -160.8303982    -0.
 -187.19554705   69.38229038  508.66011217   71.84239008]
```

> 同一个问题的不同算法
>
> 不同的算法可以用于解决同一个数学问题。比如在 scikit-learn 里 `Lasso` 对象使用 coordinate descent 方法解决 lasso 回归问题，对于大型数据集很有效。但是，scikit-learn 也提供了使用 *LARS* 算法 的: `LassoLars` 对象，对于处理带权向量非常稀疏的数据非常有效(比如，问题的观察值很少)。

分类



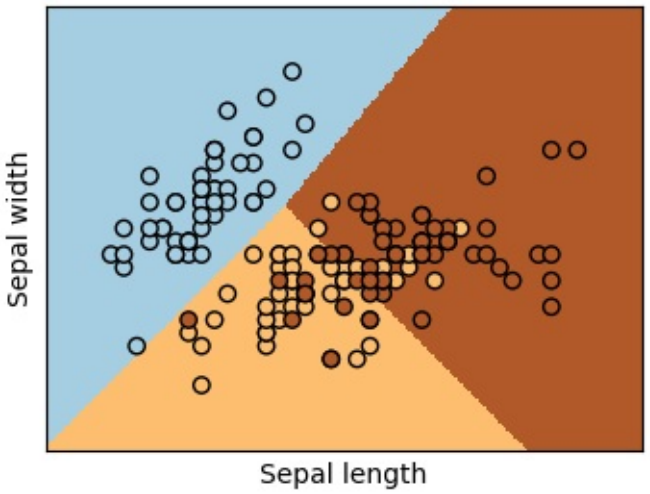
对于分类，比如标定 鸢尾属植物 任务，线性回归就不是好方法了，因为它会给数据很多远离决策边界的权值。一个线性方法是为了拟合 sigmoid 函数 或 **logistic** 函数:

$$y = \text{sigmoid}(X\beta - \text{offset}) + \epsilon = \frac{1}{1 + \exp(-X\beta + \text{offset})} + \epsilon$$

```
>>> logistic = linear_model.LogisticRegression(C=1e5)
>>> logistic.fit(iris_X_train, iris_y_train)
```

```
LogisticRegression(C=100000.0, class_weight=None, dual=False,
 fit_intercept=True, intercept_scaling=1, max_iter=100,
 multi_class='ovr', n_jobs=1, penalty='l2', random_state=None,
 solver='liblinear', tol=0.0001, verbose=0, warm_start=False)
```

这就是有名的： `LogisticRegression`



> 多类分类
>
> 如果你有很多类需要预测，一种常用方法就是去拟合一对多分类器，然后使用根据投票为最后做决定。
>
> 使用 **logistic** 回归进行收缩和稀疏
>
> `LogisticRegression` 对象中的 `C` 参数控制着正则化数量： `C` 值越大，正则化数量越小。 `penalty="l2"` 提供收缩(比如，无稀疏系数)，同时 penalty="l1" 提供稀疏化。
>
> 练习
>
> 尝试用最近邻和线性模型分类数字数据集。留出最后 10%的数据，并测试观察值预期效果。

```
from sklearn import datasets, neighbors, linear_modeldigits = datasets.load_digits()X_digits =
digits.datay_digits = digits.target
```

> 解决方法

支持向量机(SVMs)

线性 SVMs

支持向量机 属于判别模型家族：它们尝试通过找到样例的一个组合来构建一个两类之间最大化的平面。通过 `C` 参数进行正则化设置： `C` 的值小意味着边缘是通过分割线周围的所有观测样例进行计算得到的(更正则化)； `C` 的值大意味着边缘是通过邻近分割线的观测样例计算得到的(更少正则化)。

> 例子：

- Plot different SVM classifiers in the iris dataset

SVMs 可以用于回归：*SVR*（支持向量回归）-，或者分类 *SVC*（支持向量分类）。

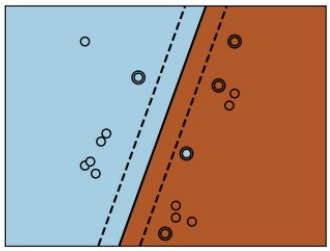
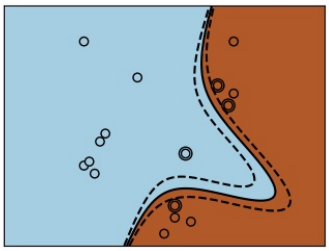
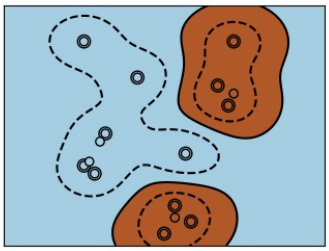
```
>>> from sklearn import svm
>>> svc = svm.SVC(kernel='linear')
>>> svc.fit(iris_X_train, iris_y_train)
SVC(C=1.0, cache_size=200, class_weight=None, coef0=0.0,
 decision_function_shape='ovr', degree=3, gamma='auto', kernel='linear',
 max_iter=-1, probability=False, random_state=None, shrinking=True,
 tol=0.001, verbose=False)
```

警告：规格化数据

对很多估计器来说，包括 SVMs，为每个特征值使用单位标准偏差的数据集，是获得好的预测重要前提。

使用核

在特征空间类并不总是线性可分的。解决办法就是构建一个不是线性的但能是多项式的函数做代替。这要使用 核技巧 *(kernel trick)*，它可以被看作通过设置 *kernels* 在观察样例上创建决策力量：

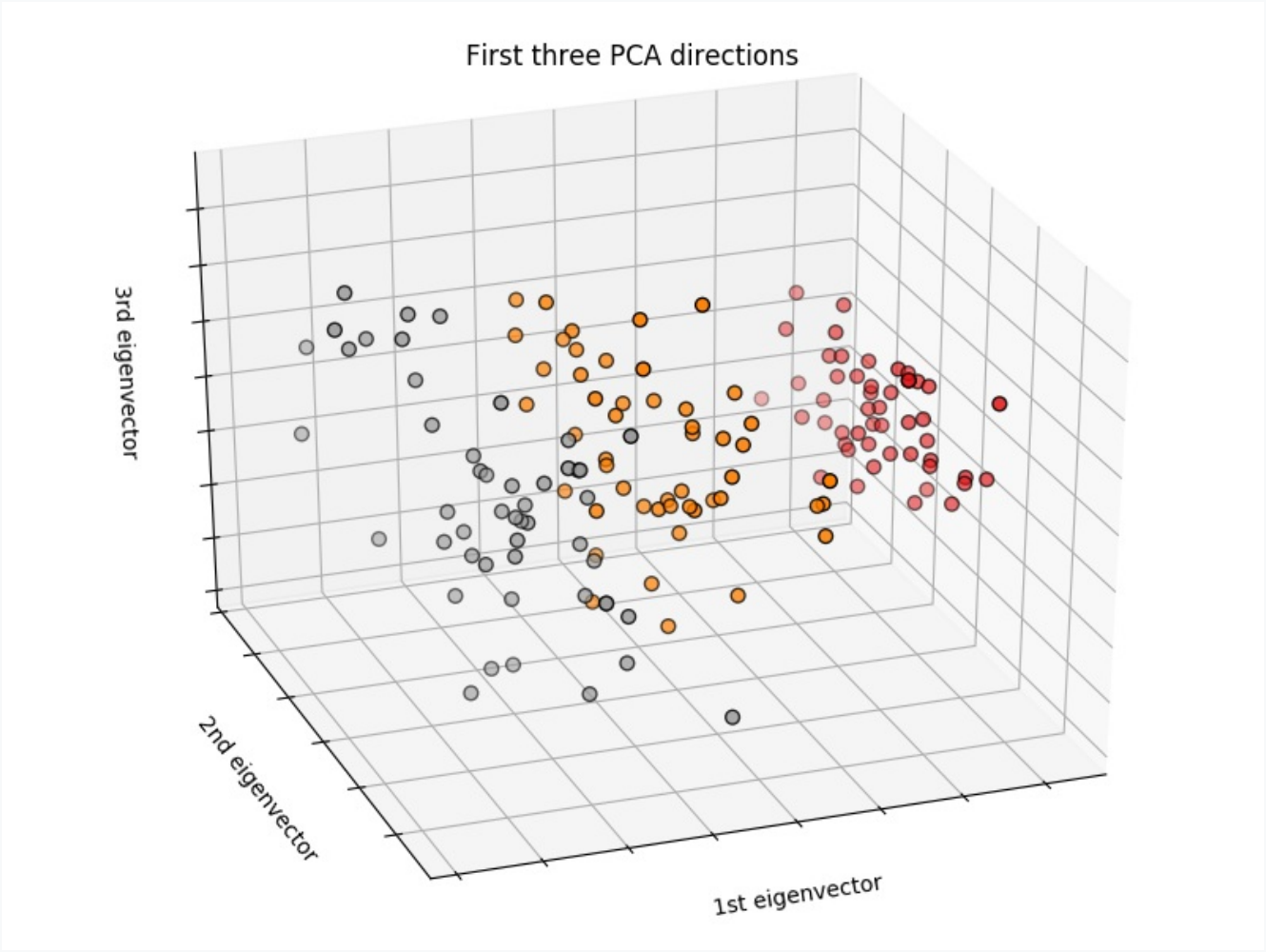
线性核	多项式核	**RBF** 内核(径向基函数)
		
`>>> svc = svm.SVC(kernel='linear')`	`>>> svc = svm.SVC(kernel='poly',degree=3)`	`>>> svc = svm.SVC(kernel='rbf')`

交互例子

查看 SVM GUI 来下载 `svm_gui.py`；通过左右按键添加两类数据点，拟合模型并改变参数和数据。

练习

根据特征1和特征2，尝试用 SVMs 把1和2类从鸢尾属植物数据集中分出来。为每一个类留下10%，并测试这些观察值预期效果。



警告：类是有序的，不要留下最后10%，不然你只能测试一个类了。提示：为了直观显示，你可以在网格上使用 `decision_function` 方法。

```
iris = datasets.load_iris()X = iris.datay = iris.targetX = X[y != 0, :2]y = y[y != 0]
```

解决方法

我们一直在努力

apachecn/sklearn-doc-zh



模型选择：选择估计量及其参数

校验者：@片刻翻译者：@森系

分数和交叉验证分数

如我们所见，每一个估计量都有一个可以在新数据上判定拟合质量(或预期值)的 `score` 方法。越大越好.

```
>>> from sklearn import datasets, svm
>>> digits = datasets.load_digits()
>>> X_digits = digits.data
>>> y_digits = digits.target
>>> svc = svm.SVC(C=1, kernel='linear')
>>> svc.fit(X_digits[:-100], y_digits[:-100]).score(X_digits[-100:], y_digits[-100:])
0.98
```

为了更好地预测精度(我们可以用它作为模型的拟合优度代理)，我们可以连续分解用于我们训练和测试用的 折叠数据。

```
>>> import numpy as np
>>> X_folds = np.array_split(X_digits, 3)
>>> y_folds = np.array_split(y_digits, 3)
>>> scores = list()
>>> for k in range(3):
...     # We use 'list' to copy, in order to 'pop' later on
...     X_train = list(X_folds)
...     X_test = X_train.pop(k)
...     X_train = np.concatenate(X_train)
...     y_train = list(y_folds)
...     y_test = y_train.pop(k)
...     y_train = np.concatenate(y_train)
...     scores.append(svc.fit(X_train, y_train).score(X_test, y_test))
>>> print(scores)
[0.934..., 0.956..., 0.939...]
```

这被称为 `KFold` 交叉验证.

交叉验证生成器

`scikit-learn` 有可以生成训练/测试索引列表的类，可用于流行的交叉验证策略。

类提供了 `split` 方法，方法允许输入能被分解的数据集，并为每次选择的交叉验证策略迭代生成训练/测试集索引。

下面是使用 `split` 方法的例子。

```
>>> from sklearn.model_selection import KFold, cross_val_score
>>> X = ["a", "a", "b", "c", "c", "c"]
>>> k_fold = KFold(n_splits=3)
>>> for train_indices, test_indices in k_fold.split(X):
...      print('Train: %s | test: %s' % (train_indices, test_indices))
Train: [2 3 4 5] | test: [0 1]
Train: [0 1 4 5] | test: [2 3]
Train: [0 1 2 3] | test: [4 5]
```

然后就可以很容易地执行交叉验证了：

```
>>> [svc.fit(X_digits[train], y_digits[train]).score(X_digits[test], y_digits[test])
...  for train, test in k_fold.split(X_digits)]
[0.963..., 0.922..., 0.963..., 0.963..., 0.930...]
```

交叉验证分数可以使用 `cross_val_score` 直接计算出来。给定一个估计量，交叉验证对象，和输入数据集，`cross_val_score` 函数就会反复分解出训练和测试集的数据，然后使用训练集和为每次迭代交叉验证运算出的基于测试集的分数来训练估计量。

默认情况下，估计器的 `score` 方法被用于运算个体分数。

可以参考 `metrics 模块` 学习更多可用的评分方法。

```
>>> cross_val_score(svc, X_digits, y_digits, cv=k_fold, n_jobs=-1)
array([ 0.93489149,  0.95659432,  0.93989983])
```

`n_jobs=-1` 意味着运算会被调度到所有 CPU 上进行。

或者，可以提供 `scoring` 参数来指定替换的评分方法。

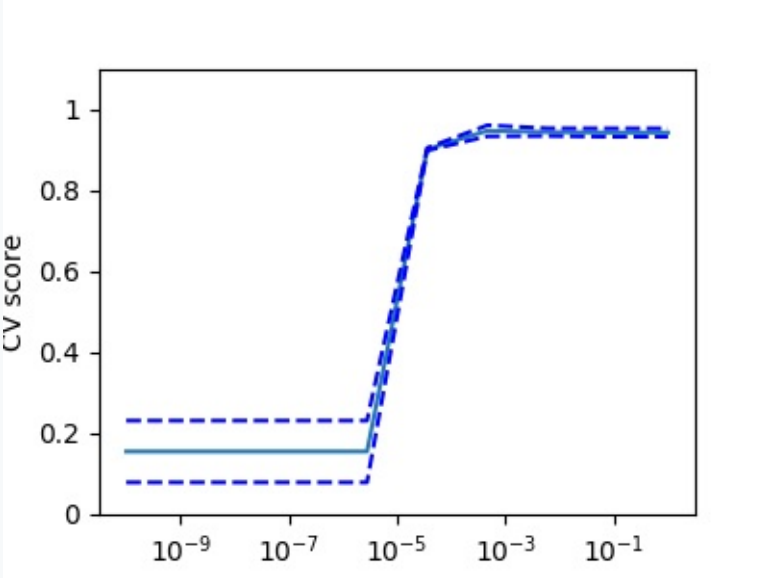
```
>>> cross_val_score(svc, X_digits, y_digits, cv=k_fold,
...                 scoring='precision_macro')
array([0.96578289, 0.92708922, 0.96681476, 0.96362897, 0.93192644])
```

交叉验证生成器

调用	功能
`KFold` **(n_splits, shuffle, random_state)**	将其分解为 K 个折叠，在 K-1 上训练，然后排除测试。
`StratifiedKFold` **(n_splits, shuffle, random_state)**	和 K-Fold 一样，但会保留每个折叠里的类分布。
`GroupKFold` **(n_splits)**	确保相同组不会在测试和训练集里。
`ShuffleSplit` **(n_splits, test_size, train_size, random_state)**	生成基于随机排列的训练/测试索引。
`StratifiedShuffleSplit`	和 shuffle 分解一样，但会保留每个迭代里的类分布。
`GroupShuffleSplit`	确保相同组不会在测试和训练集里。

`LeaveOneGroupOut()`	使用数组分组来给观察分组。
`LeavePGroupsOut(n_groups)`	忽略 P 组。
`LeaveOneOut()`	忽略一个观察。
`LeavePOut(p)`	忽略 P 观察。
`PredefinedSplit`	生成基于预定义分解的训练/测试索引。

练习



在数字数据集中，用一个线性内核绘制一个 `SVC` 估计器的交叉验证分数来作为 `C` 参数函数(使用从1到10的点对数网格).

```
    import numpy as npfrom sklearn.model_selection import cross_val_scorefrom sklearn import datasets,
    svmdigits = datasets.load_digits()X = digits.datay = digits.targetsvc = svm.SVC(kernel='linear')C_s =
1.  np.logspace(-10, 0, 10)
```

方法： Cross-validation on Digits Dataset Exercise

网格搜索和交叉验证估计量

网格搜索

`scikit-learn` 提供了一个对象，在给定数据情况下，在一个参数网格，估计器拟合期间计算分数，并选择参数来最大化交叉验证分数。这个对象在构建过程中获取估计器并提供一个估计器 API。

```
1.  >>> from sklearn.model_selection import GridSearchCV, cross_val_score
2.  >>> Cs = np.logspace(-6, -1, 10)
3.  >>> clf = GridSearchCV(estimator=svc, param_grid=dict(C=Cs),
4.  ...                    n_jobs=-1)
5.  >>> clf.fit(X_digits[:1000], y_digits[:1000])
6.  GridSearchCV(cv=None,...
7.  >>> clf.best_score_
8.  0.925...
9.  >>> clf.best_estimator_.C
```

```
0.0077...

>>> # Prediction performance on test set is not as good as on train set
>>> clf.score(X_digits[1000:], y_digits[1000:])
0.943...
```

默认情况下， `GridSearchCV` 使用一个三倍折叠交叉验证。但是，如果它检测到分类器被传递，而不是回归，它就会使用分层的三倍。

> 嵌套交叉验证
>
> ```
> >> cross_val_score(clf, X_digits, y_digits)…array([ 0.938…,  0.963…,  0.944…])
> ```
>
> 两个交叉验证循环并行执行：一个由 `GridSearchCV` 估计器设置 `gamma` ，另一个 `cross_val_score` 则是测量估计器的预期执行情况。结果分数是对新数据上的预期分数的无偏估计。
>
> 警告你不可以并行运算嵌套对象(`n_jobs` 不为1)。

交叉验证估计量

设置参数的交叉验证可以更有效地完成一个基础算法。这就是为什么对某些估计量来说，scikit-learn 提供了 交叉验证 估计量自动设置它们的参数。

```
>>> from sklearn import linear_model, datasets
>>> lasso = linear_model.LassoCV()
>>> diabetes = datasets.load_diabetes()
>>> X_diabetes = diabetes.data
>>> y_diabetes = diabetes.target
>>> lasso.fit(X_diabetes, y_diabetes)
LassoCV(alphas=None, copy_X=True, cv=None, eps=0.001, fit_intercept=True,
 max_iter=1000, n_alphas=100, n_jobs=1, normalize=False, positive=False,
 precompute='auto', random_state=None, selection='cyclic', tol=0.0001,
 verbose=False)
>>> # 估计器自动选择它的 lambda:
>>> lasso.alpha_
0.01229...
```

这些估计量和它们的副本称呼类似，在名字后加 ‘CV’。

> 练习
>
> 在糖尿病数据集中，找到最优正则化参数 α。
>
> 另外： 你有多相信 α 的选择？
>
> ```
> from sklearn import datasetsfrom sklearn.linear_model import LassoCVfrom sklearn.linear_model import
> Lassofrom sklearn.model_selection import KFoldfrom sklearn.model_selection import GridSearchCVdiabetes =
> datasets.load_diabetes()
> ```
>
> 方法： Cross-validation on diabetes Dataset Exercise

我们一直在努力

apachecn/sklearn-doc-zh



无监督学习： 寻求数据表示

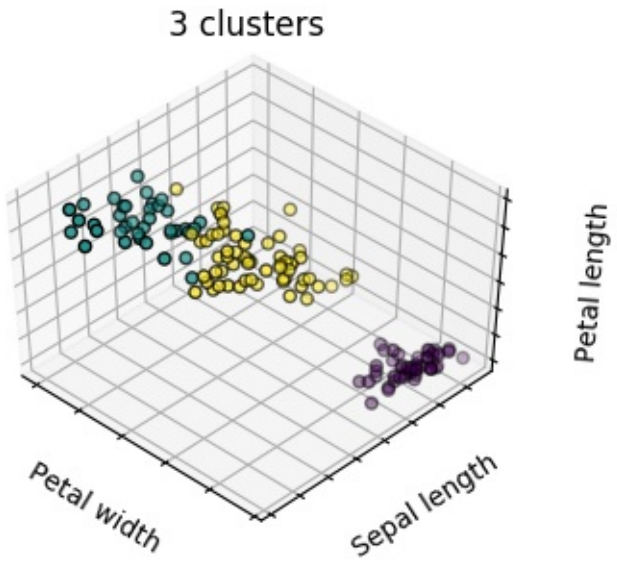
校验者： @片刻翻译者： @X

聚类： 对样本数据进行分组

> 可以利用聚类解决的问题
>
> 对于 iris 数据集来说，我们知道所有样本有 3 种不同的类型，但是并不知道每一个样本是那种类型：此时我们可以尝试一个 **clustering task**（聚类任务） 聚类算法：将样本进行分组，相似的样本被聚在一起，而不同组别之间的样本是有明显区别的，这样的分组方式就是 *clusters*（聚类）

K-means 聚类算法

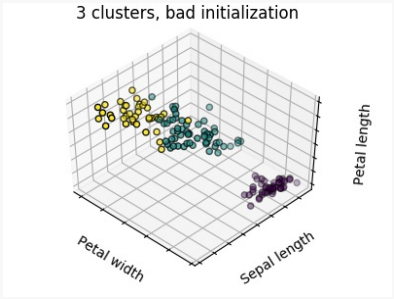
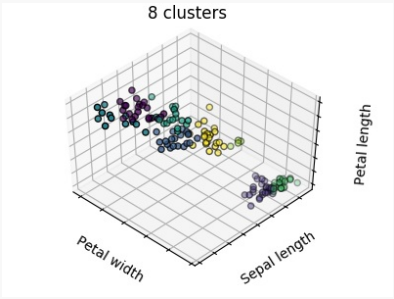
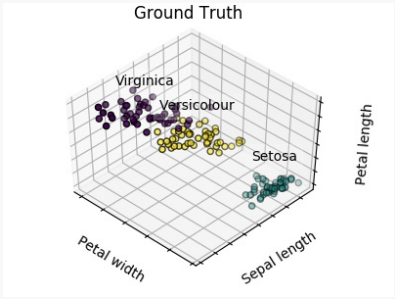
关于聚类有很多不同的聚类标准和相关算法，其中最简便的算法是 K-means 。




```
>>> from sklearn import cluster, datasets
>>> iris = datasets.load_iris()
>>> X_iris = iris.data
>>> y_iris = iris.target

>>> k_means = cluster.KMeans(n_clusters=3)
>>> k_means.fit(X_iris)
KMeans(algorithm='auto', copy_x=True, init='k-means++', ...
>>> print(k_means.labels_[::10])
[1 1 1 1 1 0 0 0 0 0 2 2 2 2 2]
>>> print(y_iris[::10])
[0 0 0 0 0 1 1 1 1 1 2 2 2 2 2]
```

> 警告
>
> *kmeans* 算法无法保证聚类结果完全绝对真实的反应实际情况。首先，选择正确合适的聚类数量不是一件容易的事情，第二，该算法对初始值的设置敏感，容易陷入局部最优。尽管 *scikit-learn* 采取了不同的方式来缓解以上问题，目前仍没有完美的解决方案。

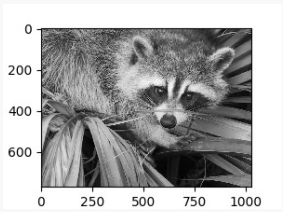
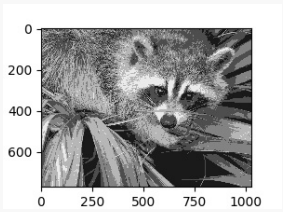
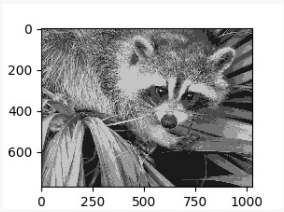
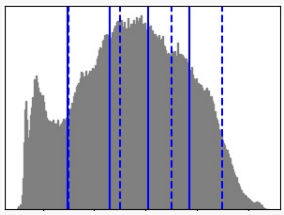
		
Bad initialization	**8 clusters**	**Ground truth**

Don't over-interpret clustering results（不要过分解读聚类结果）

Application example: vector quantization（应用案例:向量量化*(vector quantization)*）

一般来说聚类，特别是 *K_means* 聚类可以作为一种用少量样本来压缩信息的方式。这种方式就是 *vector quantization* 。例如，*K_means* 算法可以用于对一张图片进行色调分离:

```
>> import scipy as sp>> try:…    face = sp.face(gray=True)… except AttributeError:…    from scipy import
misc…    face = misc.face(gray=True)>> X = face.reshape((-1, 1)) # We need an (n_sample, n_feature)
array>> k_means = cluster.KMeans(n_clusters=5, n_init=1)>> k_means.fit(X)KMeans(algorithm='auto',
copy_x=True, init='k-means++', …>> values = k_means.cluster_centers
.squeeze()>> labels = kmeans.labels>> face_compressed = np.choose(labels, values)>>
face_compressed.shape = face.shape
```

			
Raw image	K-means quantization	Equal bins	Image histogram

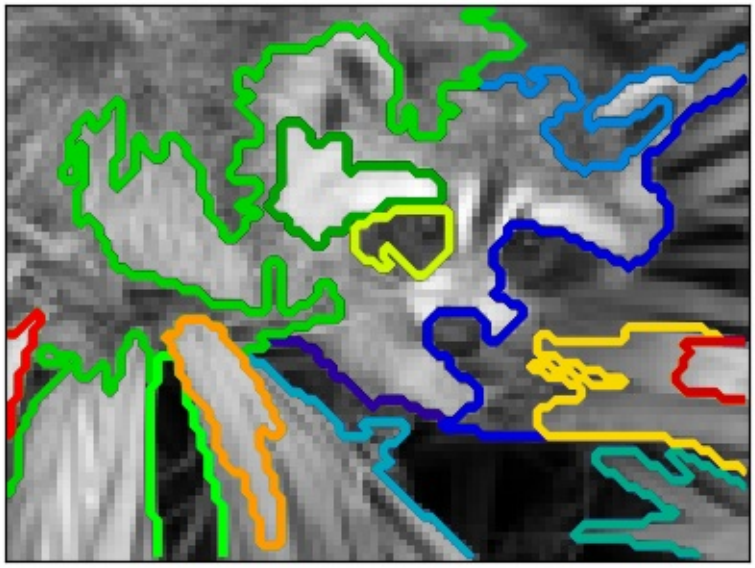
分层聚类算法： 谨慎使用

分层聚类算法是一种旨在构建聚类层次结构的分析方法，一般来说，实现该算法的大多数方法有以下两种:

- **Agglomerative**（聚合） - 自底向上的方法: 初始阶段，每一个样本将自己作为单独的一个簇，聚类的簇以最小化距离的标准进行迭代聚合。当感兴趣的簇只有少量的样本时，该方法是很合适的。如果需要聚类的 簇数量很大，该方法比K_means算法的计算效率也更高。
- **Divisive**（分裂） - 自顶向下的方法: 初始阶段，所有的样本是一个簇，当一个簇下移时，它被迭代的进行分裂。当估计聚类簇数量较大的数据时，该算法不仅效率低(由于样本始于一个簇，需要被递归的进行 分裂)，而且从统计学的角度来讲也是不合适的。

连接约束聚类

对于逐次聚合聚类，通过连接图可以指定哪些样本可以被聚合在一个簇。在 scikit 中，图由邻接矩阵来表示，通常该矩阵是一个稀疏矩阵。这种表示方法是非常有用的，例如在聚类图像时检索连接区域(有时也被称为连接要素):



```
from scipy.ndimage.filters import gaussian_filter

import matplotlib.pyplot as plt

import skimage
from skimage.data import coins
from skimage.transform import rescale

from sklearn.feature_extraction.image import grid_to_graph
from sklearn.cluster import AgglomerativeClustering

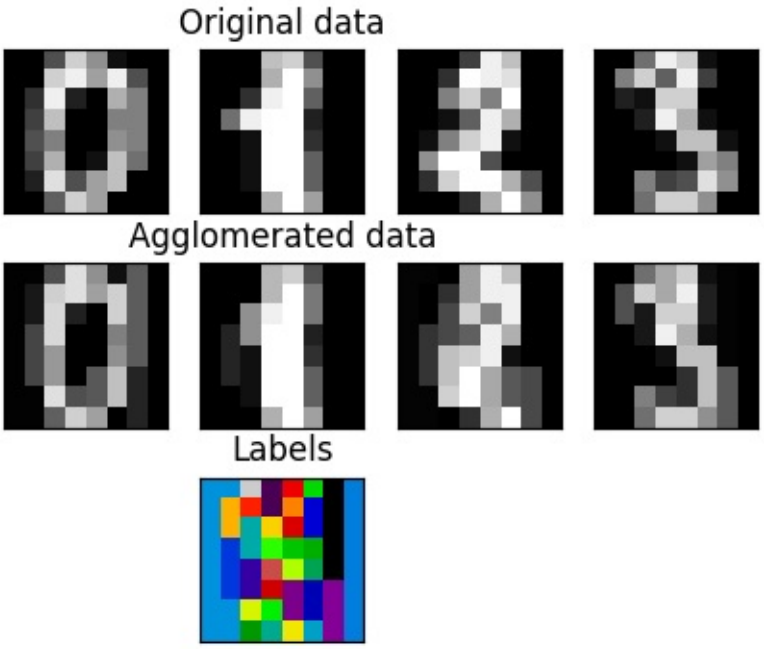
# these were introduced in skimage-0.14
if LooseVersion(skimage.__version__) >= '0.14':
    rescale_params = {'anti_aliasing': False, 'multichannel': False}
else:
    rescale_params = {}

# ###############################################################################
# Generate data
orig_coins = coins()

# Resize it to 20% of the original size to speed up the processing
# Applying a Gaussian filter for smoothing prior to down-scaling
# reduces aliasing artifacts.
smoothened_coins = gaussian_filter(orig_coins, sigma=2)
```

特征聚集

我们已经知道，稀疏性可以缓解特征维度带来的问题，i.e 即与特征数量相比，样本数量太少。 另一个解决该问题的方式是合并相似的维度：**feature agglomeration**（特征聚集）。该方法可以通过对特征聚类来实现。换 句话说，就是对样本数据转置后进行聚类。




```
>>> digits = datasets.load_digits()
>>> images = digits.images
>>> X = np.reshape(images, (len(images), -1))
>>> connectivity = grid_to_graph(*images[0].shape)

>>> agglo = cluster.FeatureAgglomeration(connectivity=connectivity,
...                                      n_clusters=32)
>>> agglo.fit(X)
FeatureAgglomeration(affinity='euclidean', compute_full_tree='auto',...
>>> X_reduced = agglo.transform(X)

>>> X_approx = agglo.inverse_transform(X_reduced)
>>> images_approx = np.reshape(X_approx, images.shape)
```

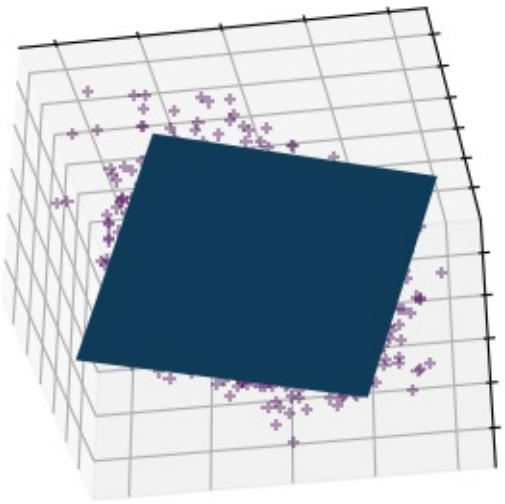
> `transform` and `inverse_transform` methods
>
> Some estimators expose a `transform` method, for instance to reduce the dimensionality of the dataset.

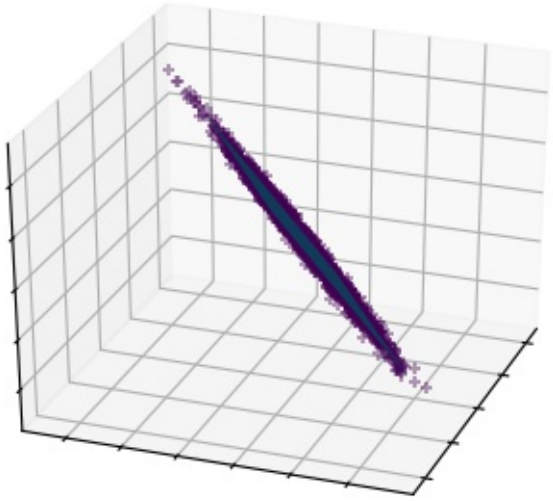
分解： 将一个信号转换成多个成份并且加载

> **Components and loadings**（成分和载荷）
>
> 如果 X 是多维数据，那么我们试图解决的问题是在不同的观察基础上对数据进行重写。我们希望学习得到载荷 L 和成分 C 使得 X = L C 。提取成分 C 有多种不同的方法。

主成份分析： PCA

主成分分析（PCA） 将能够解释数据信息最大方差的的连续成分提取出来





上图中样本点的分布在一个方向上是非常平坦的：即三个单变量特征中的任何一个都可以有另外两个特征来表示。主成分分析法(PCA)可以找到使得数据分布不 *flat* 的矢量方向(可以反映数据主要信息的特征)。

当用主成分分析(PCA)来 *transform*（转换） 数据时，可以通过在子空间上投影来降低数据的维数。

```
>>> # Create a signal with only 2 useful dimensions
>>> x1 = np.random.normal(size=100)
>>> x2 = np.random.normal(size=100)
>>> x3 = x1 + x2
>>> X = np.c_[x1, x2, x3]

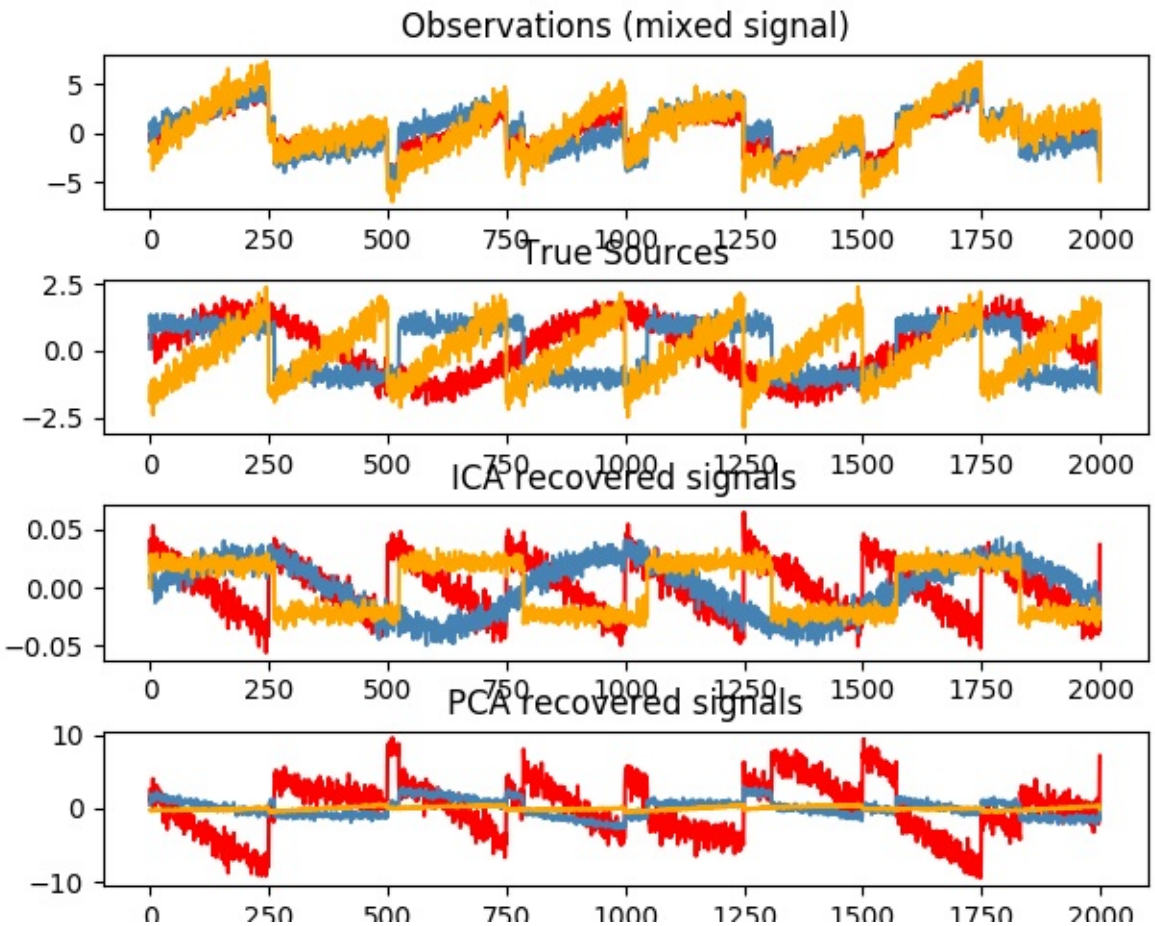
>>> from sklearn import decomposition
>>> pca = decomposition.PCA()
>>> pca.fit(X)
PCA(copy=True, iterated_power='auto', n_components=None, random_state=None,
  svd_solver='auto', tol=0.0, whiten=False)
```

```
>>> print(pca.explained_variance_)
[  2.18565811e+00   1.19346747e+00   8.43026679e-32]

>>> # As we can see, only the 2 first components are useful
>>> pca.n_components = 2
>>> X_reduced = pca.fit_transform(X)
>>> X_reduced.shape
(100, 2)
```

独立成分分析： ICA

独立成分分析（ICA） 可以提取数据信息中的独立成分，这些成分载荷的分布包含了最多的 的独立信息。该方法能够恢复 **non-Gaussian**（非高斯） 独立信号：




```
>>> # Generate sample data
>>> import numpy as np
>>> from scipy import signal
>>> time = np.linspace(0, 10, 2000)
>>> s1 = np.sin(2 * time)  # Signal 1 : sinusoidal signal
>>> s2 = np.sign(np.sin(3 * time))  # Signal 2 : square signal
>>> s3 = signal.sawtooth(2 * np.pi * time)  # Signal 3: saw tooth signal
>>> S = np.c_[s1, s2, s3]
>>> S += 0.2 * np.random.normal(size=S.shape)  # Add noise
>>> S /= S.std(axis=0)  # Standardize data
>>> # Mix data
>>> A = np.array([[1, 1, 1], [0.5, 2, 1], [1.5, 1, 2]])  # Mixing matrix
```

```
>>> X = np.dot(S, A.T)  # Generate observations

>>> # Compute ICA
>>> ica = decomposition.FastICA()
>>> S_ = ica.fit_transform(X)  # Get the estimated sources
>>> A_ = ica.mixing_.T
>>> np.allclose(X,  np.dot(S_, A_) + ica.mean_)
True
```

我们一直在努力

apachecn/sklearn-doc-zh

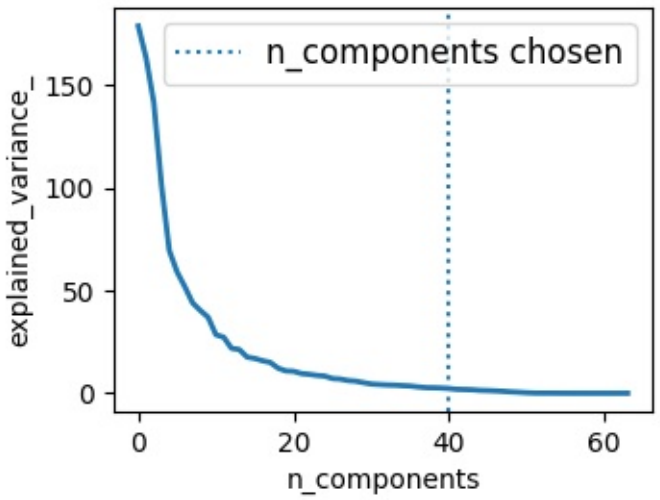


把它们放在一起

校验者：@片刻翻译者：@X

模型管道化

我们已经知道一些模型可以做数据转换，一些模型可以用来预测变量。我们可以建立一个组合模型同时完成以上工作：




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

from sklearn import datasets
from sklearn.decomposition import PCA
from sklearn.linear_model import SGDClassifier
from sklearn.pipeline import Pipeline
from sklearn.model_selection import GridSearchCV


# Define a pipeline to search for the best combination of PCA truncation
# and classifier regularization.
logistic = SGDClassifier(loss='log', penalty='l2', early_stopping=True,
                         max_iter=10000, tol=1e-5, random_state=0)
pca = PCA()
pipe = Pipeline(steps=[('pca', pca), ('logistic', logistic)])

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

# Parameters of pipelines can be set using '__' separated parameter names:
param_grid = {
    'pca__n_components': [5, 20, 30, 40, 50, 64],
```

```
    'logistic__alpha': np.logspace(-4, 4, 5),
}
search = GridSearchCV(pipe, param_grid, iid=False, cv=5)
search.fit(X_digits, y_digits)
print("Best parameter (CV score=%0.3f):" % search.best_score_)
print(search.best_params_)

# Plot the PCA spectrum
pca.fit(X_digits)

fig, (ax0, ax1) = plt.subplots(nrows=2, sharex=True, figsize=(6, 6))
ax0.plot(pca.explained_variance_ratio_, linewidth=2)
ax0.set_ylabel('PCA explained variance')

ax0.axvline(search.best_estimator_.named_steps['pca'].n_components,
            linestyle=':', label='n_components chosen')
```

用特征面进行人脸识别

该实例用到的数据集来自 LFW_(Labeled Faces in the Wild)。数据已经进行了初步预处理

> http://vis-www.cs.umass.edu/lfw/lfw-funneled.tgz (233MB)

```
"""
```

Faces recognition example using eigenfaces and SVMs

The dataset used in this example is a preprocessed excerpt of the"Labeled Faces in the Wild", aka LFW_:

http://vis-www.cs.umass.edu/lfw/lfw-funneled.tgz (233MB)

.. _LFW: http://vis-www.cs.umass.edu/lfw/

Expected results for the top 5 most represented people in the dataset:

================== ============ ======= ========== ======= precision recall f1-score support================== ============ ======= ========== ======= Ariel Sharon 0.67 0.92 0.77 13 Colin Powell 0.75 0.78 0.76 60 Donald Rumsfeld 0.78 0.67 0.72 27 George W Bush 0.86 0.86 0.86 146Gerhard Schroeder 0.76 0.76 0.76 25 Hugo Chavez 0.67 0.67 0.67 15 Tony Blair 0.81 0.69 0.75 36

```
    avg / total       0.80      0.80      0.80       322
```

================== ============ ======= ========== =======

"""from time import timeimport loggingimport matplotlib.pyplot as plt

from sklearn.model_selection import train_test_splitfrom sklearn.model_selection import GridSearchCVfrom sklearn.datasets import fetch_lfw_peoplefrom sklearn.metrics import classification_reportfrom sklearn.metrics import confusion_matrixfrom

```
sklearn.decomposition import PCAfrom sklearn.svm import SVC

print(doc)
```

Display progress logs on stdout

```
logging.basicConfig(level=logging.INFO, format='%(asctime)s %(message)s')
```

# #

Download the data, if not already on disk and load it as numpy arrays

```
lfw_people = fetch_lfw_people(min_faces_per_person=70, resize=0.4)
```

introspect the images arrays to find the shapes (for plotting)

```
n_samples, h, w = lfw_people.images.shape
```

for machine learning we use the 2 data directly (as relative pixel

positions info is ignored by this model)

```
X = lfw_people.datan_features = X.shape[1]
```

the label to predict is the id of the person

```
y = lfw_people.targettarget_names = lfw_people.target_namesn_classes = target_names.shape[0]

print("Total dataset size:")print("n_samples: %d" % n_samples)print("n_features: %d" % n_features)print("n_classes: %d" % n_classes)
```

# #

Split into a training set and a test set using a stratified k fold

split into a training and testing set

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

# #

Compute a PCA (eigenfaces) on the face dataset (treated as unlabeled

dataset): unsupervised feature extraction / dimensionality reduction

```
n_components = 150

print("Extracting the top %d eigenfaces from %d faces" % (n_components, X_train.shape[0]))t0 = time()pca =
PCA(n_components=n_components, svd_solver='randomized', whiten=True).fit(X_train)print("done in %0.3fs" % (time() - t0))

eigenfaces = pca.components_.reshape((n_components, h, w))

print("Projecting the input data on the eigenfaces orthonormal basis")t0 = time()X_train_pca =
pca.transform(X_train)X_test_pca = pca.transform(X_test)print("done in %0.3fs" % (time() - t0))
```

# #

Train a SVM classification model

```
print("Fitting the classifier to the training set")t0 = time()param_grid = {'C': [1e3, 5e3, 1e4, 5e4, 1e5], 'gamma':
[0.0001, 0.0005, 0.001, 0.005, 0.01, 0.1], }clf = GridSearchCV(SVC(kernel='rbf', class_weight='balanced'), param_grid,
cv=5, iid=False)clf = clf.fit(X_train_pca, y_train)print("done in %0.3fs" % (time() - t0))print("Best estimator found by
grid search:")print(clf.best_estimator_)
```

# #

Quantitative evaluation of the model quality on the test set

```
print("Predicting people's names on the test set")t0 = time()y_pred = clf.predict(X_test_pca)print("done in %0.3fs" %
(time() - t0))

print(classification_report(y_test, y_pred, target_names=target_names))print(confusion_matrix(y_test, y_pred,
labels=range(n_classes)))
```

# #

Qualitative evaluation of the predictions using matplotlib

```
def plot_gallery(images, titles, h, w, n_row=3, n_col=4): """Helper function to plot a gallery of portraits""" plt.figure(figsize=(1.8 n_col, 2.4 n_row)) plt.subplots_adjust(bottom=0, left=.01, right=.99, top=.90, hspace=.35) for i in range(n_row * n_col): plt.subplot(n_row, n_col, i + 1) plt.imshow(images[i].reshape((h, w)), cmap=plt.cm.gray) plt.title(titles[i], size=12) plt.xticks(()) plt.yticks(())
```

plot the result of the prediction on a portion of the test set

```
def title(y_pred, y_test, target_names, i): pred_name = target_names[y_pred[i]].rsplit(' ', 1)[-1] true_name = target_names[y_test[i]].rsplit(' ', 1)[-1] return 'predicted: %s\ntrue: %s' % (pred_name, true_name)

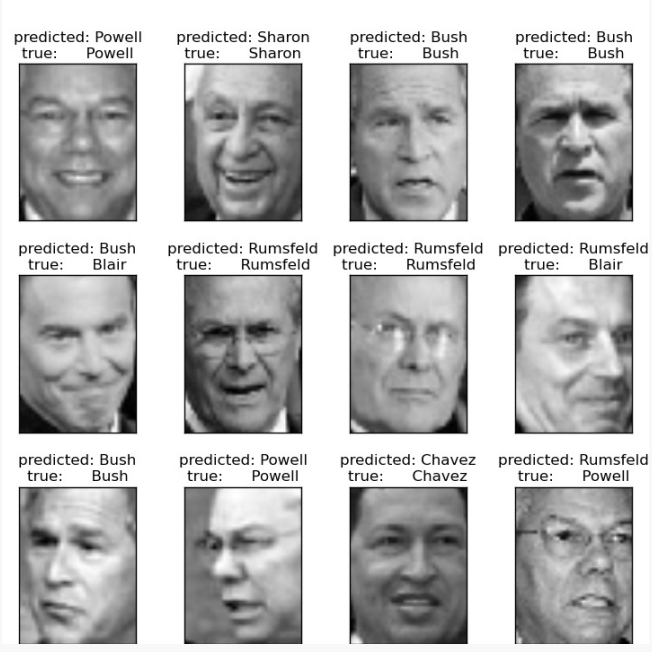
prediction_titles = [title(y_pred, y_test, target_names, i) for i in range(y_pred.shape[0])]

plot_gallery(X_test, prediction_titles, h, w)
```

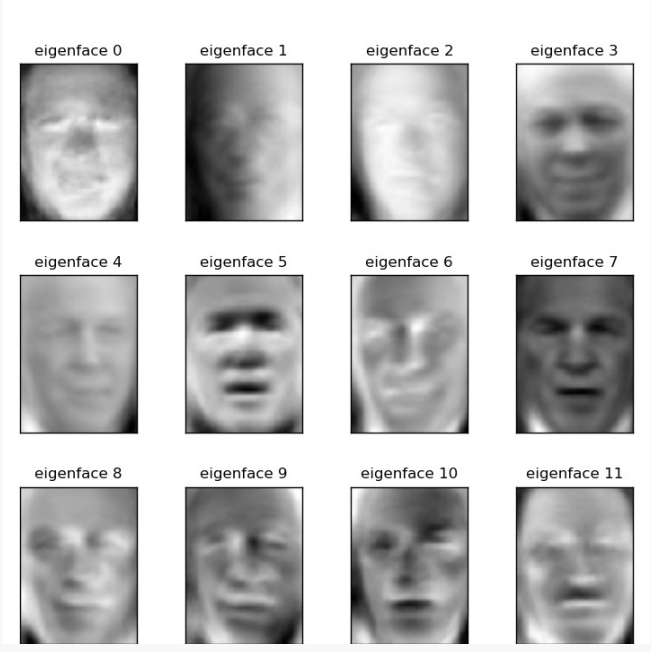
plot the gallery of the most significative eigenfaces

```
eigenface_titles = ["eigenface %d" % i for i in range(eigenfaces.shape[0])]plot_gallery(eigenfaces, eigenface_titles, h, w)

plt.show()
```



Prediction



Eigenfaces

数据集中前5名最有代表性样本的预期结果：

```
                   precision    recall  f1-score   support

Gerhard_Schroeder       0.91      0.75      0.82        28
   Donald_Rumsfeld       0.84      0.82      0.83        33
```

```
      Tony_Blair       0.65      0.82      0.73        34
     Colin_Powell      0.78      0.88      0.83        58
   George_W_Bush       0.93      0.86      0.90       129

      avg / total      0.86      0.84      0.85       282
```

开放性问题： 股票市场结构

我们可以预测 Google 在特定时间段内的股价变动吗？

Learning a graph structure

我们一直在努力

apachecn/sklearn-doc-zh



寻求帮助

校验者：@片刻翻译者：@X

项目邮件列表

如果您在使用 scikit 的过程中发现错误或者需要在说明文档中澄清的内容，可以随时通过 Mailing List 进行咨询。

机器学习从业者的 Q&A 社区

Quora.com:	Quora有一个和机器学习问题相关的主题以及一些有趣的讨论：https://www.quora.com/topic/Machine-Learning
Stack Exchange:	Stack Exchange 系列网站包含 multiple subdomains for Machine Learning questions（机器学习问题的多个分支）。

- '斯坦福大学教授 Andrew Ng 教授的机器学习优秀免费在线课程'：https://www.coursera.org/learn/machine-learning
- '另一个优秀的免费在线课程，对人工智能采取更一般的方法'：https://www.udacity.com/course/intro-to-artificial-intelligence--cs271

我们一直在努力

apachecn/sklearn-doc-zh



处理文本数据

校验者：@NellyLuo@那伊抹微笑@微光同尘翻译者：@Lielei

本指南旨在一个单独实际任务中探索一些主要的 `scikit-learn` 工具：分析关于 20 个不同主题的一个文件集合（新闻组帖子）。

在本节中，我们将会学习如何：

- 读取文件内容以及所属的类别
- 提取合适于机器学习的特征向量
- 训练一个线性模型来进行分类
- 使用网格搜索策略找到特征提取组件和分类器的最佳配置

教程设置

开始这篇教程之前，你必须首先安装 *scikit-learn* 以及所有其要求的库。

更多信息和系统安装指导请参考 安装说明 。

这篇入门教程的源代码可以在你的 scikit-learn 文件夹下面找到：

```
scikit-learn/doc/tutorial/text_analytics/
```

这个入门教程包含以下的子文件夹：

- `*.rst files` - 用 sphinx 编写的该教程的源代码
- `data` - 用来存放在该教程中用到的数据集的文件夹
- `skeletons` - 用来练习的未完成的示例脚本
- `solutions` - 练习的答案

你也可以将这个文件结构拷贝到您的电脑的硬盘里名为 `sklearn_tut_workspace` 的文件夹中来编辑你自己的文件完成练习，同时保持原有文件结构不变：

```
% cp -r skeletons work_directory/sklearn_tut_workspace
```

机器学习算法需要数据。 进入每一个 `$TUTORIAL_HOME/data` 子文件夹，然后运行 `fetch_data.py` 脚本（需要您先读取这些文件）。

例如：

```
% cd $TUTORIAL_HOME/data/languages
% less fetch_data.py
% python fetch_data.py
```

加载这 20 个新闻组的数据集

该数据集名为 “Twenty Newsgroups” 。 下面是这个数据集的官方介绍，引自 网站:

> Twenty Newsgroups 数据集是一个包括近 20,000 个新闻组文档的集合，（几乎）平均分成了 20 个不同新闻组。 据我们所知，这最初是由 Ken Lang 收集的 ，很可能是为了他的论文 “Newsweeder: Learning to filter netnews,” 尽管他没有明确提及这个集合。 这 20 个新闻组集合已成为一个流行的数据集，用于机器学习中的文本应用的试验中，如文本分类和文本聚类。

接下来我们会使用 scikit-learn 中的这个内置数据集加载器来加载这 20 个新闻组。 或者，您也可以手动从网站上下载数据集，使用 `sklearn.datasets.load_files` 功能，并将其指向未压缩文件夹下的 `20news-bydate-train` 子文件夹。

在第一个示例中，为了节约时间，我们将使用部分数据：从 20 个类别的数据集中选出 4 个来进行训练:

```
>>> categories = ['alt.atheism', 'soc.religion.christian',
...               'comp.graphics', 'sci.med']
```

如下所示，我们现在能够加载对应这些类别的文件列表:

```
>>> from sklearn.datasets import fetch_20newsgroups
>>> twenty_train = fetch_20newsgroups(subset='train',
...     categories=categories, shuffle=True, random_state=42)
```

返回的数据集是一个 `scikit-learn` “bunch”: 一个简单的包含多个 “field” 的存储对象，可以方便的使用 python 中的 `dict` keys 或 `object` 属性来读取，比如 `target_names` 包含了所请求的类别名称:

```
>>> twenty_train.target_names
['alt.atheism', 'comp.graphics', 'sci.med', 'soc.religion.christian']
```

这些文件本身被读进内存的 `data` 属性中。 另外，这些文件名称也可以容易获取到:

```
>>> len(twenty_train.data)
2257
>>> len(twenty_train.filenames)
2257
```

让我们打印出所加载的第一个文件的前几行:

```
>>> print("\n".join(twenty_train.data[0].split("\n")[:3]))
From: sd345@city.ac.uk (Michael Collier)
Subject: Converting images to HP LaserJet III?
Nntp-Posting-Host: hampton

>>> print(twenty_train.target_names[twenty_train.target[0]])
comp.graphics
```

监督学习需要让训练集中的每个文档对应一个类别标签。 在这个例子中，类别是每个新闻组的名称，也刚好是每个储存文本文件的文件夹的名称。

由于速度和空间上效率的原因 `scikit-learn` 加载目标属性为一个整型数列， 它与 `target_names` 列表中类别名称的 index（索引）相对应。 每个样本的类别的整数型 id 存放在 `target` 属性中：

```
>>> twenty_train.target[:10]
array([1, 1, 3, 3, 3, 3, 3, 2, 2, 2])
```

也可以通过如下方式取得类别名称：

```
>>> for t in twenty_train.target[:10]:
...     print(twenty_train.target_names[t])
...
comp.graphics
comp.graphics
soc.religion.christian
soc.religion.christian
soc.religion.christian
soc.religion.christian
soc.religion.christian
sci.med
sci.med
sci.med
```

你可以发现所有的样本都被随机打乱（使用了修正的 RNG 种子）： 当你在重新训练整个数据集之前，这样可以帮助你只选取前几个样本来快速训练一个模型以及获得初步结果。

从文本文件中提取特征

为了在文本文件中执行机器学习算法， 我们首先要做的是将文本内容转化成数值形式的特征向量。

词袋

最直观的方法就是用词袋来表示：

- 在训练集中每一个出现在任意文中的单词分配一个特定的整数 id（比如，通过建立一个从单词到整数索引的字典）。
- 对于每个文档 `#i`，计算每个单词 `w` 的出现次数并将其存储在 `X[i, j]` 中作为特征 `#j` 的值，其中 `j` 是在字典中词 `w` 的索引。在这种方法中 `n_features` 是在整个文集（文章集合的缩写，下同）中不同单词的数量： 这个值一般来说超过 100,000 。

如果 `n_samples == 10000` ， 存储 `X` 为 “float32” 型的 numpy 数组将会需要 10000 x 100000 x 4 bytes = **4GB**内存 ，在当前的计算机中非常不好管理的。

幸运的是， **X** 数组中大多数的值为 **0** ，是因为特定的文档中使用的单词数量远远少于总体的词袋单词个数。 因此我们可以称词袋模型是典型的 **high-dimensional sparse datasets**（高维稀疏数据集） 。 我们可以通过只在内存中保存特征向量中非 0 的部分以节省大量内存。

`scipy.sparse` 矩阵正是能完成上述操作的数据结构，同时 `scikit-learn` 有对这样的数据结构的内置支持。

使用 scikit-learn 来对文本进行分词

文本的预处理， 分词以及过滤停用词都被包含在一个可以构建特征字典和将文档转换成特征向量的高级组件中

```
>>> from sklearn.feature_extraction.text import CountVectorizer
>>> count_vect = CountVectorizer()
>>> X_train_counts = count_vect.fit_transform(twenty_train.data)
>>> X_train_counts.shape
(2257, 35788)
```

`CountVectorizer` 支持单词或者连续字符的 N-gram 模型的计数。 一旦拟合， 向量化程序就会构建一个包含特征索引的字典：

```
>>> count_vect.vocabulary_.get(u'algorithm')
4690
```

在词汇表中一个单词的索引值对应的是该单词在整个训练的文集中出现的频率。

从出现次数到出现频率

出现次数的统计是非常好的开始，但是有个问题：长的文本相对于短的文本有更高的单词平均出现次数，尽管他们可能在描述同一个主题。

为了避免这些潜在的差异，只需将各文档中每个单词的出现次数除以该文档中所有单词的总数：这些新的特征称之为词频 `tf` (Term Frequencies)。

另一个在词频的基础上改良是，降低在该训练文集中的很多文档中均出现的单词的权重，从而突出那些仅在该训练文集中在一小部分文档中出现的单词的信息量。

这种方法称为 tf-idf ，全称为 “Term Frequency times Inverse Document Frequency” 。

tf 和 **tf-idf** 都可以按照下面的方式计算：

```
>>> from sklearn.feature_extraction.text import TfidfTransformer
>>> tf_transformer = TfidfTransformer(use_idf=False).fit(X_train_counts)
>>> X_train_tf = tf_transformer.transform(X_train_counts)
>>> X_train_tf.shape
(2257, 35788)
```

在上面的样例代码中，我们首先使用了 `fit(..)` 方法来拟合对数据的 estimator（估算器），接着使用 `transform(..)` 方法来把我们的计数矩阵转换成 *tf-idf* 型。 通过跳过冗余处理，这两步可以结合起来，来更快地得到同样的结果。这种操作可以使用上节提到过的 `fit_transform(..)` 方法来完成，如下：

```
>>> tfidf_transformer = TfidfTransformer()
>>> X_train_tfidf = tfidf_transformer.fit_transform(X_train_counts)
>>> X_train_tfidf.shape
(2257, 35788)
```

训练分类器

现在我们有了我们的特征，我们可以训练一个分类器来预测一个帖子所属的类别。 让我们从 朴素贝叶斯 分类器开始. 该分类器为该任务提供了一个好的基准（baseline）. `scikit-learn` 包含了该分类器的若干变种；最适用在该问题上的变种是多项式分类器：

```
>>> from sklearn.naive_bayes import MultinomialNB
>>> clf = MultinomialNB().fit(X_train_tfidf, twenty_train.target)
```

为了尝试预测新文档所属的类别，我们需要使用和之前同样的步骤来抽取特征。 不同之处在于，我们在transformer调用 `transform` 而不是 `fit_transform` ，因为这些特征已经在训练集上进行拟合了：

```
>>> docs_new = ['God is love', 'OpenGL on the GPU is fast']
>>> X_new_counts = count_vect.transform(docs_new)
>>> X_new_tfidf = tfidf_transformer.transform(X_new_counts)

>>> predicted = clf.predict(X_new_tfidf)

>>> for doc, category in zip(docs_new, predicted):
...     print('%r => %s' % (doc, twenty_train.target_names[category]))
...
'God is love' => soc.religion.christian
'OpenGL on the GPU is fast' => comp.graphics
```

构建 Pipeline（管道）

为了使得 向量化（vectorizer） => 转换器（transformer） => 分类器（classifier） 过程更加简单，`scikit-learn` 提供了一个 `Pipeline` 类，操作起来像一个复合分类器：

```
>>> from sklearn.pipeline import Pipeline
>>> text_clf = Pipeline([('vect', CountVectorizer()),
...                      ('tfidf', TfidfTransformer()),
...                      ('clf', MultinomialNB()),
... ])
```

名称 `vect` ， `tfidf` 和 `clf` （分类器）都是任意的。 我们将会在下面的网格搜索（grid search）小节中看到它们的用法。 现在我们可以使用下面的一行命令来训练模型：

```
>>> text_clf.fit(twenty_train.data, twenty_train.target)
Pipeline(...)
```

在测试集上的性能评估

评估模型的预测准确度同样简单：

```
>>> import numpy as np
>>> twenty_test = fetch_20newsgroups(subset='test',
...     categories=categories, shuffle=True, random_state=42)
>>> docs_test = twenty_test.data
>>> predicted = text_clf.predict(docs_test)
>>> np.mean(predicted == twenty_test.target)
0.834...
```

那就是，我们模型的准确度为 83.4%. 我们使用线性分类模型 支持向量机（SVM），是公认的最好的文本分类算法之一（尽管训练速度比朴素贝叶斯慢一点）。仅需要在 Pipeline（管道）中插接入不同的分类器对象，就可以改变我们的学习器：

```
>>> from sklearn.linear_model import SGDClassifier
>>> text_clf = Pipeline([
...     ('vect', CountVectorizer()),
...     ('tfidf', TfidfTransformer()),
...     ('clf', SGDClassifier(loss='hinge', penalty='l2',
...                           alpha=1e-3, random_state=42,
...                           max_iter=5, tol=None)),
... ])

>>> text_clf.fit(twenty_train.data, twenty_train.target)
Pipeline(...)
>>> predicted = text_clf.predict(docs_test)
>>> np.mean(predicted == twenty_test.target)
0.9101...
```

此外，`scikit-learn` 提供了更加细致的模型性能评估工具：

```
>>> from sklearn import metrics
>>> print(metrics.classification_report(twenty_test.target, predicted,
...     target_names=twenty_test.target_names))
...
                        precision    recall  f1-score   support

           alt.atheism       0.95      0.80      0.87       319
         comp.graphics       0.87      0.98      0.92       389
               sci.med       0.94      0.89      0.91       396
soc.religion.christian       0.90      0.95      0.93       398

              accuracy                           0.91      1502
             macro avg       0.91      0.91      0.91      1502
          weighted avg       0.91      0.91      0.91      1502


>>> metrics.confusion_matrix(twenty_test.target, predicted)
array([[256,  11,  16,  36],
       [  4, 380,   3,   2],
       [  5,  35, 353,   3],
       [  5,  11,   4, 378]])
```

正如所期望的， confusion matrix（混淆矩阵）表明 atheism 和 christian 两个类别的新闻帖子会比它们中任一类与 computer graphics 类别的新闻贴子更容易彼此混淆。

使用网格搜索进行调参

我们已经接触了类似于 `TfidfTransformer` 中 `use_idf` 这样的参数，分类器也有许多参数； 比如， `MultinomialNB` 包含了平滑参数 `alpha` 以及 `SGDClassifier` 有惩罚参数 `alpha` 和目标函数中的可设置的损失以及惩罚因子（请翻阅该模组说明，或者更多信息请使用 python 的 `help` 文档）。

不是调整 chain（链）的各种组件的参数，相反，通过构建巨大的可能值的网格从而对最佳参数的穷尽搜索是可能的。 我们尝试所有情况的分类器：使用词袋或者二元模型，使用或者不使用 idf ，在线性 SVM 上设置 0.01 或者 0.001 的惩罚参数：

```
>>> from sklearn.model_selection import GridSearchCV
>>> parameters = {'vect__ngram_range': [(1, 1), (1, 2)],
...               'tfidf__use_idf': (True, False),
...               'clf__alpha': (1e-2, 1e-3),
... }
```

很明显，如此地穷举是非常耗时的。 如果我们有多个 CPU，通过设置 `n_jobs` 参数可以进行并行处理。 如果我们将该参数设置为 `-1` ， 该方法会使用机器的所有 CPU 核心：

```
>>> gs_clf = GridSearchCV(text_clf, parameters, n_jobs=-1)
```

网格搜索的操作跟 `scikit-learn` 中常见的模型的操作是类似的。 让我们来选择训练集中的一小部分进行搜索来加速计算：

```
>>> gs_clf = gs_clf.fit(twenty_train.data[:400], twenty_train.target[:400])
```

在 `GridSearchCV` 中调用 `fit` ，我们得到了用来 `predict` 的分类器：

```
>>> twenty_train.target_names[gs_clf.predict(['God is love'])[0]]
'soc.religion.christian'
```

该对象的 `best_score_` 和 `best_params_` 属性存放了最佳的平均分以及其所对应的参数配置：

```
>>> gs_clf.best_score_
0.900...
>>> for param_name in sorted(parameters.keys()):
...     print("%s: %r" % (param_name, gs_clf.best_params_[param_name]))
...
clf__alpha: 0.001
tfidf__use_idf: True
vect__ngram_range: (1, 1)
```

更详细的搜索总结可以在 `gs_clf.cv_results_` 中得到。

`cvresults` 参数能够轻易地以 `DataFrame` 格式被导入到 pandas 中，以供以后观察。

练习

为了做这个练习，请拷贝 ‘skeletons’ 文件夹到新的文件夹，并将其命名为 ‘workspace’:

```
1. % cp -r skeletons workspace
```

这时候可以任意更改练习的代码而不会破坏原来的练习指导。

然后启动 ipython 交互环境，并运行以下 python 脚本:

```
1. [1] %run workspace/exercise_XX_script.py arg1 arg2 arg3
```

如果出现错误，请使用 `%debug` 来启动 ipdb 调试环境。

迭代更改答案直到练习完成。

在每个练习中，**skeleton** 文件提供了所有必需的**import**语句，加载数据的模板代码以及评估模型准确度的样例代码.

练习 1：语言识别

- 请使用以维基百科中的文章作为训练集的自定义的预处理器和 `CharNGramAnalyzer` ， 来编写一个文本分类的 Pipeline（管道）。
- 评估某些测试集的性能.

ipython command line:

```
1. %run workspace/exercise_01_language_train_model.py data/languages/paragraphs/
```

练习 2：电影评论的情感分析

- 编写一个文本分类 Pipeline（管道）来将电影评论分类为积极的（positive）还是消极的（negative）。
- 使用网格搜索来找到好的参数配置。
- 使用测试集进行性能评估。

ipython 命令行:

```
1. %run workspace/exercise_02_sentiment.py data/movie_reviews/txt_sentoken/
```

练习 3：CLI 文本分类实用程序

使用前面的练习结果以及标准库的 `cPickle 模块，编写一个命令行工具来检测由 stdin` 输入的文本语言。如果输入的文本是英文的话，请评估该文本的极性（积极的还是消极的）。

加分项：该工具对其预测给出置信水平。

快速链接

当你完成这个章节时，下面是几个建议来帮助你对 scikit-learn 有进一步的理解：

- 尝试使用 `CountVectorizer` 类下的 `analyzer` 以及 `token normalisation` 。
- 如果你没有标签，尝试使用 聚类 来解决你的问题。
- 如果每篇文章有多个标签，请参考 多类别和多标签部分 _ 。
- 尝试使用 Truncated SVD 解决 隐语义分析.
- 使用 Out-of-core Classification 来学习不会存入计算机的主存储器的数据。
- 使用 Hashing Vectorizer 来节省内存，以代替 `CountVectorizer` 。

我们一直在努力

apachecn/sklearn-doc-zh



选择正确的评估器(estimator)

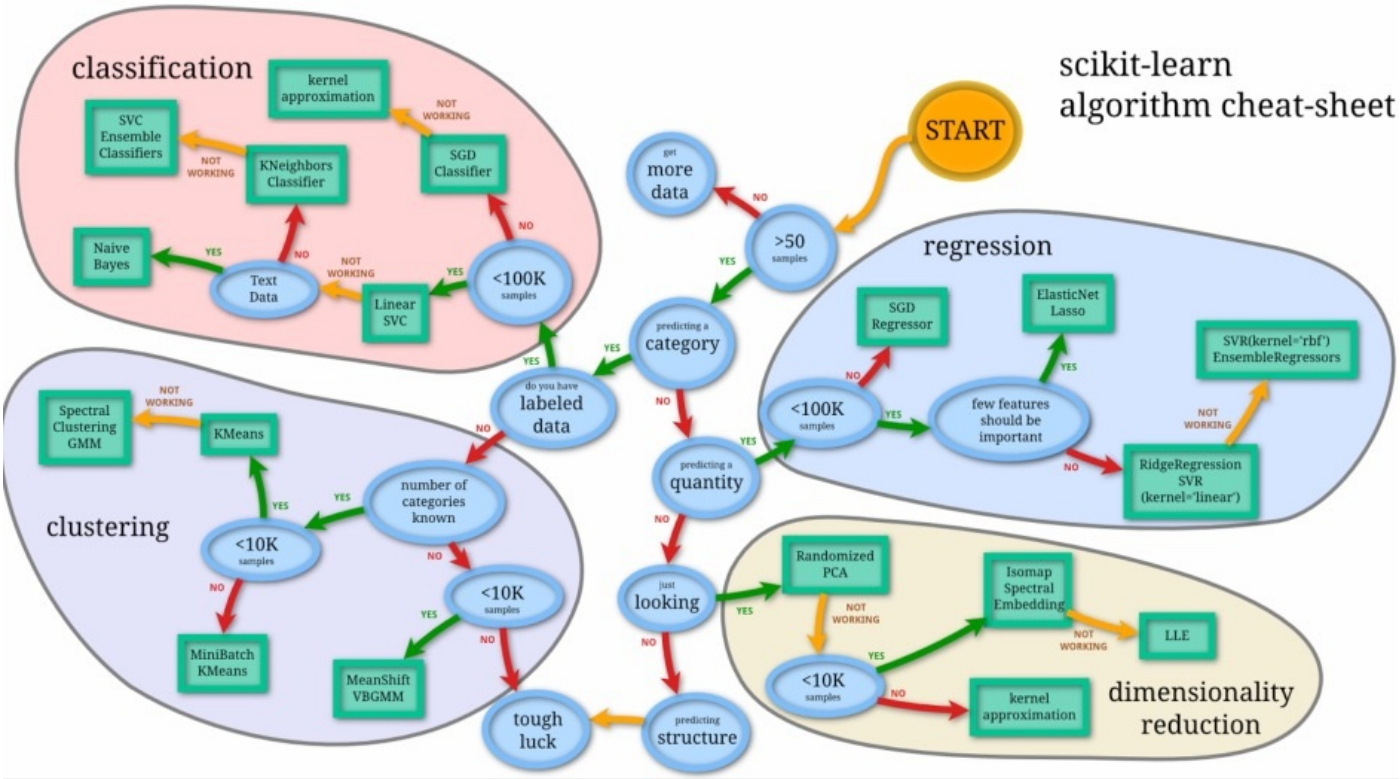
校验者:翻译者: @李孟禹

通常，解决机器学习问题的最困难的部分可能是找到恰当的的评估器(estimator)。

不同的评估器更适合不同类型的数据和不同的问题。

下面的流程图是一些粗略的指导，可以让用户根据自己的数据来选择应该尝试哪些评估器。

点击下图的任何评估器，查看其文档。



我们一直在努力

apachecn/sklearn-doc-zh

加入QQ群

外部资源，视频和谈话

校验者：

翻译者：@巴黎ㄍメの雨季

For written tutorials, see the Tutorial section of the documentation.

Scientific Python 的新手？

For those that are still new to the scientific Python ecosystem, we highly recommend the Python Scientific Lecture Notes. This will help you find your footing a bit and will definitely improve your scikit-learn experience. A basic understanding of NumPy arrays is recommended to make the most of scikit-learn.

外部教程

There are several online tutorials available which are geared toward specific subject areas:

- Machine Learning for NeuroImaging in Python
- Machine Learning for Astronomical Data Analysis

视频

- An introduction to scikit-learn Part I and Part II at Scipy 2013 by Gael Varoquaux, Jake Vanderplas and Olivier Grisel. Notebooks on github.

- Introduction to scikit-learn by Gael Varoquaux at ICML 2010

> A three minute video from a very early stage of the scikit, explaining the basic idea and approach we are following.

- Introduction to statistical learning with scikit-learn by Gael Varoquaux at SciPy 2011

> An extensive tutorial, consisting of four sessions of one hour. The tutorial covers the basics of machine learning, many algorithms and how to apply them using scikit-learn. The material corresponding is now in the scikit-learn documentation section 关于科学数据处理的统计学习教程.

- Statistical Learning for Text Classification with scikit-learn and NLTK (and slides) by Olivier Grisel at PyCon 2011

> Thirty minute introduction to text classification. Explains how to use NLTK and

scikit-learn to solve real-world text classification tasks and compares against cloud-based solutions.

- Introduction to Interactive Predictive Analytics in Python with scikit-learn by Olivier Grisel at PyCon 2012

> 3-hours long introduction to prediction tasks using scikit-learn.

- scikit-learn - Machine Learning in Python by Jake Vanderplas at the 2012 PyData workshop at Google

> Interactive demonstration of some scikit-learn features. 75 minutes.

- scikit-learn tutorial by Jake Vanderplas at PyData NYC 2012

> Presentation using the online tutorial, 45 minutes.

我们一直在努力

apachecn/sklearn-doc-zh



常见问题

校验者：@Mysry翻译者：@STAN,废柴0.1

在这里，我们试着给出一些经常出现在邮件列表上的问题的答案。

项目名称是什么(很多人弄错)?

scikit-learn，不是scikik、SciKit、sci-kit learn，也不是我们曾使用的scikits.learn和scikits-learn。

如何称呼scikit-learn?

sy-kit learn。sci代表着科学!

选择 scikit的理由 ？

scikit拥有很多围绕Scipy构建的科学工具箱。你可以在 <*https://scikits.appspot.com/scikits>* 查找工具列表。 *scikit-image* <*http://scikit-image.org/>* 和 scikit-learn_一样受欢迎 。

如何才能为 scikit-learn 贡献自己的力量?

在添加一个通常是重要冗长的新算法前，推荐你观看 known issues 。 关于 scikit-learn 贡献，请不要直接和 scikit-learn 的贡献者联系。

获得scikit-learn用法的最佳方法？

对于一般的机器学习问题，请使用 交叉验证 和 `[machine-learning]` 。

对于**scikit-learn**使用的问题，请点击 Stack Overflow 中带有 `[scikit-learn]` 和 `[python]` 标签的部分。 你也可以使用 联系列表 .

请确保包您的代码段较小(最好少于10行)，并且可以突出显示玩具数据集上的问题（例如从 `sklearn.datasets` 或者是用固定随机数种子 `numpy.random` 函数生成）. 请删除任何不需要重现您的问题的代码行。该问题因该可以在安装了scikit-learn的python命令行中简单地通过复制粘贴您的代码重现. 并且不要忘了import语句.

编写能够重现的代码的更多指南可以在以下网址找到：http://stackoverflow.com/help/mcve

你的代码依旧引起了你即使Google过也不明白的异常，请确保你在运行复制的脚本时包含完整的回溯。

关于错误报告或者功能请求，请参阅 issue tracker on Github 。

你还可以在 scikit-learn Gitter channel 找到一些用户与开发人员。

请不要直接给任何作者发邮件请求帮助，报告**bug**或其他与**scikit-learn**相关的问题。

如何创建一个 bunch 对象?

不要创建 bunch 对象，他们不是 scikit-learn API 的一部分. Bunch 对象只是打包一些 numpy 数组的方式. numpy 数组只是为scikit-learn 用户的模型提供数据的工具。

例如，训练一个分类器，你只需要一个2D的输入变量数组 `X` 和一个1D目标变量数组 `y` 。 `X` 数组将特征作为列，样本保存为行。 `y` 数组包含用于对每个样本的类成员资格编码的整型数值 `X` .

如何将我自己的数据集加载到 scikit-learn 可用的格式?

一般来说,scikit-learn 可以在诸如 numpy 数组或者 scipy 稀疏矩阵这样的数字数据上运行。其他格式的如 pandas Dataframe 的数组也是可以的。

有关将数据文件加载到可用数据结构中的更多信息，参阅 加载外部数据集 。

新算法的纳入标准是什么 ？

我们仅考虑添加已经完善的算法。通常的标准是发布3年以上，被引用超过 200 次，而且被广泛使用。对广泛使用的方法提供了明确改进的技术（如增强型数据结构或更有效的近似技术）也将被考虑纳入。

在满足上述标准的算法或技术中,只有这些能够 `适合` 现在 scikit-learn API 的， `预测/转换` 接口通常具有 numpy 阵列或稀疏矩阵的输入/输出。

贡献者应该支持通过研究论文或其他类似软件包中的实现来增加提出算法技术的重要性，通过常见的用例或应用程序证明其有用性，并通过基准和/或图证实性能改进（如果有的话）。预计所提出的算法应该在某些领域优于已经在 scikit-learn 中实现的方法.

还要注意，您的实现不需要在 scikit-learn 中与 scikit-learn 工具一起使用。您可以用 scikit-learn 兼容的方式实现您最喜欢的算法，将其上传到 github 便于我们获知。我们将在 Related Projects 列出。

为什么你对 scikit-learn 中的算法如此讲究?

代码是需要维护的， 我们需要平衡我们的团队规模和代码量(并且: 复杂性与特征的数量非线性相关). 该软件包依赖于核心开发人员利用他们的空闲时间修复错误，维护代码和审查贡献。 添加的任何算法都需要开发人员的密切，但此时原作者可能早已失去兴趣。 也可以在 该链接 查看。

为什么从 scikit-learn 中删除 HMMS ？

未来 scikit-learn 中会添加图形模型或序列预测吗? 。

未来 scikit-learn 中会添加图形模型或序列预测吗?

目前可能性不大 scikit-learn 尝试为机器学习中的基本任务提供统一的 API，使用管道和元算法（如网格搜索）将所有内容都集中在一起。 结构化学习所需的概念，API ，算法和专业知识与 scikit learn 所提供的不同。如果我们开始进行随意的结构化学习，那么我们需要重新设计整个软件包，这个项目可能在自身的负担下崩溃。

这里有两个类似于 scikit-learn 的做结构化预测的 API:

- pystruct 处理一般结构化学习

(专注于具有近似推理的任意图形结构上的 SSVMs ； 将样本的概念定义为图形结构的一个实例)

- seqlearn 仅处理序列（专注于精确推断;主要是为了完整性附带了 HMMs ;将特征向量作为样本，并对特征向量之间的依赖使用偏移编码）

你会添加 GPU 支持吗?

目前不会，主要在于 GPU 支持将引入许多软件依赖关系并引入平台特定的问题。scikit-learn 旨在轻松安装在各种平台上。除了神经网络，GPU 在当今的机器学习中不起重要作用，通常我们可以通过仔细选择算法来获得更大的速度增益。

你支持 PyPy 吗?

如果您不知道 PyPy 它是个新的，快速，及时的编译 Python 实现，但是我们不支持。若 PyPy 中的 NumPy support 已经完善或接近完善，并且 SciPy 也被移植时，我才会考虑移植。 scikit-learn 使用了太多的 NumPy 所以不能完成部分实现。

如何处理字符串数据（或树，图...）？

scikit-learn 估计器假设您将为他们提供实值特征向量。这个假设在几乎所有的库都是硬编码的。但是，您可以通过多种方式将非数字输入馈送到估计器。

如果您有文本文档，可以使用术语频率特征； 参阅内置 文本向量化器 的 文本特征提取 。 对于从任何类型的数据更一般的特征提取，见 从字典类型加载特征 和 特征哈希（相当于一种降维技巧） 。

另一个常见的情况是当您对这些数据有非数字数据和自定义距离（或相似度）指标时。示例包括具有编辑距离的字符串（也称为 Levenshtein 距离;例如 DNA 或 RNA 序列）。这些可以编码为数字，但这样做通常很麻烦也容易出错。使用任意数据的距离度量可以通过以下两种方式完成。

首先，许多估计器采用预计算的距离/相似矩阵，因此如果数据集不太大，可以计算所有输入对的距离。如果数据集很大，您可以使用仅具有一个“特征”的特征向量，该特征是单独数据结构的索引，并提供在该数据结构中查找实际数据的自定义度量函数。 例如，使用 DBSCAN 与 Levenshtein 距离:

>

```
>>> from leven import levenshtein
>>> import numpy as np
>>> from sklearn.cluster import dbscan
```

```
>>> data = ["ACCTCCTAGAAG", "ACCTACTAGAAGTT", "GAATATTAGGCCGA"]
>>> def lev_metric(x, y):
...     i, j = int(x[0]), int(y[0])     # extract indices
...     return levenshtein(data[i], data[j])
...
>>> X = np.arange(len(data)).reshape(-1, 1)
>>> X
array([[0],
 [1],
 [2]])
>>> dbscan(X, metric=lev_metric, eps=5, min_samples=2)
([0, 1], array([ 0,  0, -1]))
```

(这里使用了第三方编辑距离包 `leven`)

类似的技巧也可以在树形内核、图形内核等使用。

为什么我有时会在 OSX 或 Linux 下遇到 n_jobs > 1 崩溃/冻结?

一些例如 `GridSearchCV` 和 `crossval_score` 的*scikit-learn*工具，它们可以依靠 *Python* 的内置 *_multiprocessing* 模块，通过 `n_jobs > 1` 作为参数，将执行并行化到多个 Python 进程。

问题是 Python 由于性能原因 `multiprocessing` 会执行 `fork` 系统调用，而不是 `exec` 系统调用。许多库如 OSX 下的（某些版本的）Accelerate / vecLib，（某些版本的）MKL，GCC 的 OpenMP 运行时,nvidia 的 Cuda （可能还有一些其他的),都是自行管理自己的内部线程池。在调用 *fork* 时，子进程中的线程池状态已损坏：线程池认为它有许多线程，而只有主线程状态已被 fork。有可能更改库，使它们在发生 fork 时检测，并在该情况下重新初始化线程池：我们对 OpenBLAS 执行了此操作（从 0.2.10 开始在 master 中合并），并且我们向 GCC 的 OpenMP 运行时提供了一个 补丁 （尚未审查)。

但最终，真正的罪魁祸首是 Python 的 `multiprocessing` ，执行 `fork` 而不是执行 `exec` 来减少开始的和新使用的并行计算的 Python 进程的开销。但这这违反了 POSIX 标准，因而被一些软件编辑器（如苹果）拒绝认为在 Accelerate / vecLib 中缺乏 fork 安全是一个 bug。

在 Python 3.4 或以上版本中，现在可以配置 `multiprocessing` 决定使用 ‘forkserver’ 或者 ‘spawn’ 启动方法(而不是默认的 ‘fork’)来管理进程池。若要使用 scikit-learn 来解决此问题，你可以将 JOBLIB_START_METHOD 的环境变量设为 ‘forkserver’ 。但是用户应该意识到使用 ‘forkserver’ 方法会阻止 joblib.Parallel 调用在 shell 会话中交互定义的函数。

如果你直接使用 `multiprocessing` 的自定义代码而非通过调用 joblib 使用，你可以为你的程序全局启用 ‘forkserver’ 模式： 在主脚本中插入以下说明：

```
import multiprocessing

# other imports, custom code, load data, define model...

if __name__ == '__main__':
    multiprocessing.set_start_method('forkserver')

```

```
    # call scikit-learn utils with n_jobs > 1 here
```

你可以在 multiprocessing文档 上找到更多新启动方法的默认值。

为什么不支持深度学习或强化学习/scikit-learn 中将会支持深度学习或强化学习吗?

深度学习和强化学习需要丰富的词汇来定义一个架构，深度学习还需要 GPU 来进行有效的计算。然而，这些都不符合 scikit-learn 的设计限制。因此，深度学习和强化学习目前已经超出了 scikit-learn 寻求实现的范围。

你可以找到更多关于gpu支持的信息 `Will you add GPU support?` _.

为什么我的pull请求没有得到注意?

scikit-learn 审查过程需要大量的时间，因此贡献者不应该因为 `pull` 请求缺乏回应或没有被审查而沮丧。我们非常关心第一次正确的使用，因为维护和以后的更改成本高昂。我们不会发布 “实验性” 代码， 所以我们所有的贡献将会立即得到大量使用，并且在最初的时候就应该是最高的质量。

除此之外，scikit-learn 在审查能力方面是有限的; 许多审稿人和核心开发人员都是利用自己的时间在 scikit-learn 工作。如果您的 `pull` 请求进展缓慢，可能是因为审阅者很忙，希望您能理解，并希望您不要因为这个原因而关闭您的 `pull` 请求或停止您的工作。

如何为整个执行设置一个统一的 random_state ?

一般对于测试和复制，更为重要的是让整个执行由具有随机组件的算法中使用的伪随机数生成器的单个种子进行控制。Scikit-learn 不使用自己的全局随机状态;每当 RandomState 实例或整数随机种子不作为参数提供时，它依赖于类似的用法 `numpy.random.seed` numpy 全局随机数种子。例如，要将执行的 numpy 全局随机状态设置为 42，可以在相应的脚本中执行以下操作:

```
import numpy as np
np.random.seed(42)
```

然而，全局随机状态在执行期间容易被其他代码修改。因此，确保可复制性的唯一方法是在每个地方传递 `RandomState` 实例，并确保估算器和交叉验证分隔符都具有其 `random_state` 参数集。

我们一直在努力

apachecn/sklearn-doc-zh



时光轴详情

该页面展示了本 scikit-learn 中文文档项目随时间变化，而发生的重大事情，特在该页面记录下来。

贡献者

衷心感谢给位参与的贡献者，具体的贡献者列表，请参阅【0.19.X】贡献者名单.

项目角色

有关该项目的角色角色信息，比如： 负责人，发起人，支持者，翻译者，校验者 。。。等等信息，请参阅项目角色结构.

时光轴

2019-06-29: 更新用户指南和教程到0.21.3版本，修正格式问题，文档链接为:https://sklearn.apachecn.org/docs/0.21.3

2017-11-17: 更新校验完成的页面，修改文档样式，在对应的页面加上对应的贡献者，文档链接为:https://sklearn.apachecn.org/docs/0.19.x.

2017-10-20: 发起 scikit-learn 0.19 中文文档 的第一期校验活动，又来了一些新的大佬，参与到该活动中来，具体的校验详情请参阅校验进度.

2017-09-29: 发起 scikit-learn 0.19 中文文档 的翻译活动，有很多无私的贡献者愿意参与，具体的翻译详情请参阅翻译进度.

我们一直在努力

apachecn/sklearn-doc-zh

