Table of Contents

This book is translated from official user guide of scikit-learn.

1.1. 广义线性模型

英文原文

以下介绍的方法均是用于求解回归问题，其目标值预计是输入变量的一个线性组合。写成数学语言为：假设$\hat{y}$是预测值，则有

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

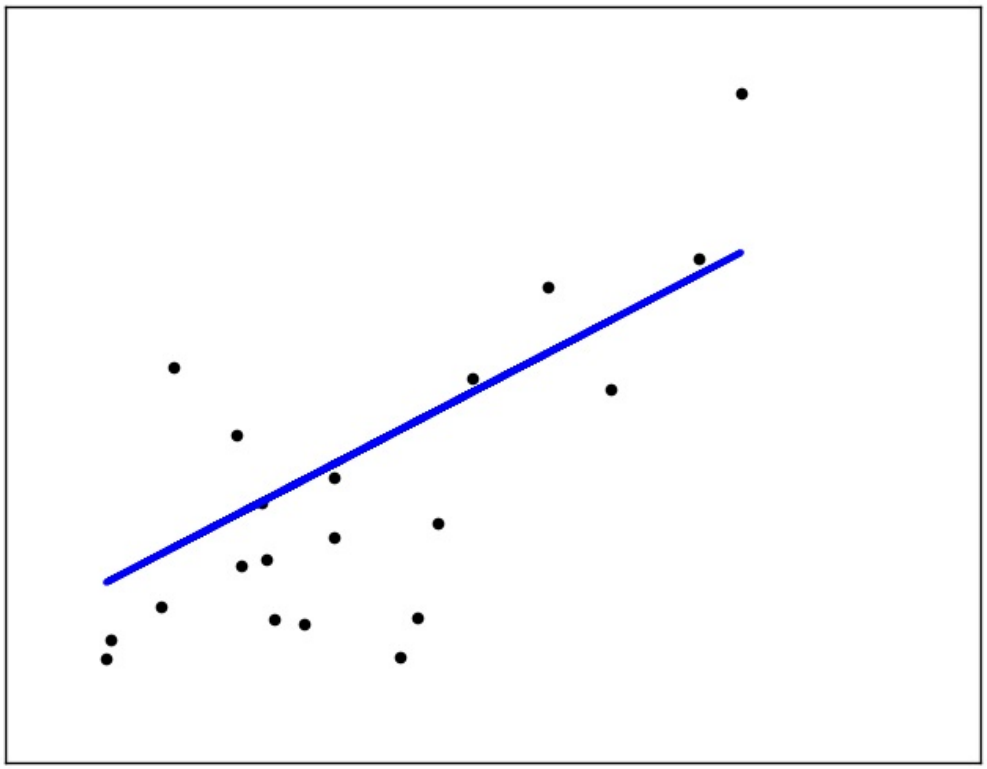
在本节中，称向量$w = (w_1, \ldots, w_p)$为`coef_`，{% math %}w0{% endmath %}为`intercept`

若要将通用的线性模型用于分类问题，可参见Logistic回归

1.1.1 普通最小二乘法

`LinearRegression`使用系数$w = (w_1, \ldots, w_p)$拟合一个线性模型。拟合的目标是要将线性逼近预测值（Xw）和数据集中观察到的值（y）两者之差的平方和尽量降到最小。写成数学公式，即是要解决以下形式的问题

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



`LinearRegression` 的 `fit` 方法接受数组X和y作为输入，将线性模型的系数w存在成员变量 `coef_` 中：

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

需要注意的是，普通最小二乘法的系数预测取决于模型中各个项的独立性。假设各个项相关，矩阵X的列总体呈现出线性相关，那么X就会很接近奇异矩阵，其结果就是经过最小二乘得到的预测值会对原始数据中的随机误差高度敏感，从而每次预测都会产生比较大的方差。这种状况称为重共线性。例如，在数据未经实验设计就进行收集时就会发生重共线性。

线性回归的例子

1.1.1.1 普通最小二乘法的复杂度

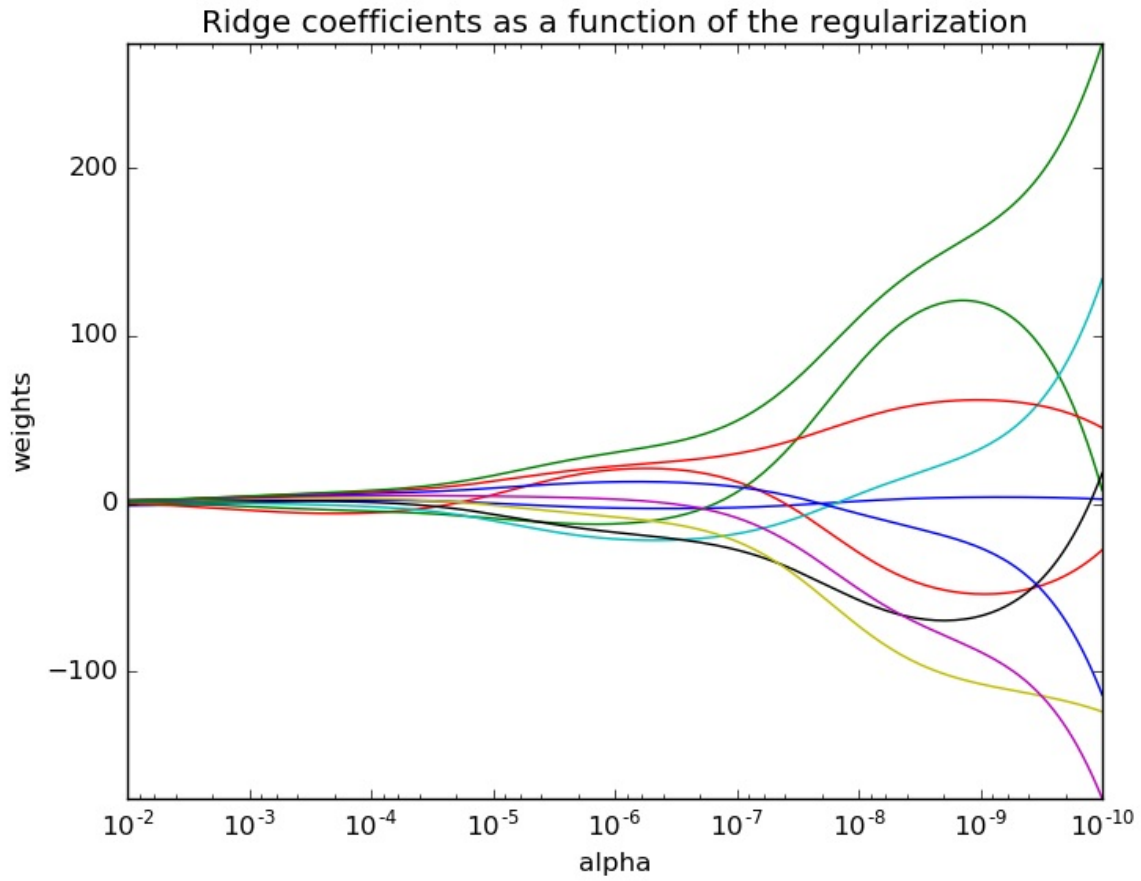
此方法使用X的奇异值分解来求解最小二乘。如果X是$n \times p$矩阵，则算法的复杂度为$O(np^2)$，假设$n \geq p$。

1.1.2 岭回归

岭回归（Ridge regression）引入了一种对系数大小进行惩罚的措施，来解决普通最小二乘可能遇到的某些问题。岭回归最小化带有惩罚项的残差平方和：

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

这里，$\alpha \geq 0$是一个复杂的参数，用以控制系数的缩减量。α值越大，系数缩减得越多，因而会对共线性更加鲁棒。



和其它线性模型类似，`Ridge` 将数组X和y作为 `fit` 方法的参数，将线性模型的系数w存在成员变量 `coef_` 中：

```
>>> from sklearn import linear_model
>>> clf = linear_model.Ridge (alpha = .5)
>>> clf.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)
>>> clf.coef_
array([ 0.34545455,  0.34545455])
>>> clf.intercept_
0.13636...
```

例子：

- 岭回归系数图像：一种正则化方法
- 使用稀疏特征进行文本文档分类

岭回归的时间复杂度

岭回归的复杂度与普通最小二乘法一样

使用广义交叉验证设置正则化参数

`RidgeCV` 使用内置的交叉验证方法选择参数α，进而实现了岭回归。该对象和GridSearchCV的原理类似，只不过RidgeCV默认使用广义交叉验证方法（留一交叉验证的一种高效形式）：

```
>>> from sklearn import linear_model
>>> clf = linear_model.RidgeCV(alphas=[0.1, 1.0, 10.0])
>>> clf.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)
>>> clf.alpha_
0.1
```

参考文献

- “Notes on Regularized Least Squares”, Rifkin & Lippert （技术报告，课程胶片）

1.1.3. Lasso

`Lasso` 是一种预测稀疏系数的线性模型。在某些情况下，Lasso非常有用，因为其倾向于选择拥有更少参数数量的模型，从而有效减少了最终解决方案所依赖的变量个数。这样的特性使得Lasso和它的变种成为了压缩感知（compressed sensing）这一领域的基石。在特定情况下，它可以恢复非零权重的确切集合（it can recover the exact set of non-zero weights）（参见Compressive sensing: tomography reconstruction with L1 prior (Lasso)）

用数学语言讲，Lasso是一个增加了ℓ_1正则项的线性模型。需要优化的目标函数为

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

因此，Lasso是要将带有$\alpha||w||_1$这一项的最小二乘误差降到最小。这里α是一个常数，$||w||_1$是参数向量的ℓ_1范数

`Lasso` 类的实现使用了坐标下降算法来拟合系数。Least Angle Regression使用了另一种实现方法。

```
>>> from sklearn import linear_model
>>> clf = linear_model.Lasso(alpha = 0.1)
>>> clf.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)
>>> clf.predict([[1, 1]])
array([ 0.8])
```

对于一些底层的任务来说，函数 `lasso_path` 也很有效。该函数沿所有可能值的全部路径计算系数

例子：

- Lasso and Elastic Net for Sparse Signals
- Compressive sensing: tomography reconstruction with L1 prior (Lasso)

注释

- 使用**Lasso**进行特征选择：由于Lasso回归可以给出稀疏模型，因此该方法可以用于特征选择。参见“基于L1的特征选择”了解更多细节
- 随机化稀疏：对于特征选择或稀疏性恢复，可以考虑使用随机化稀疏模型

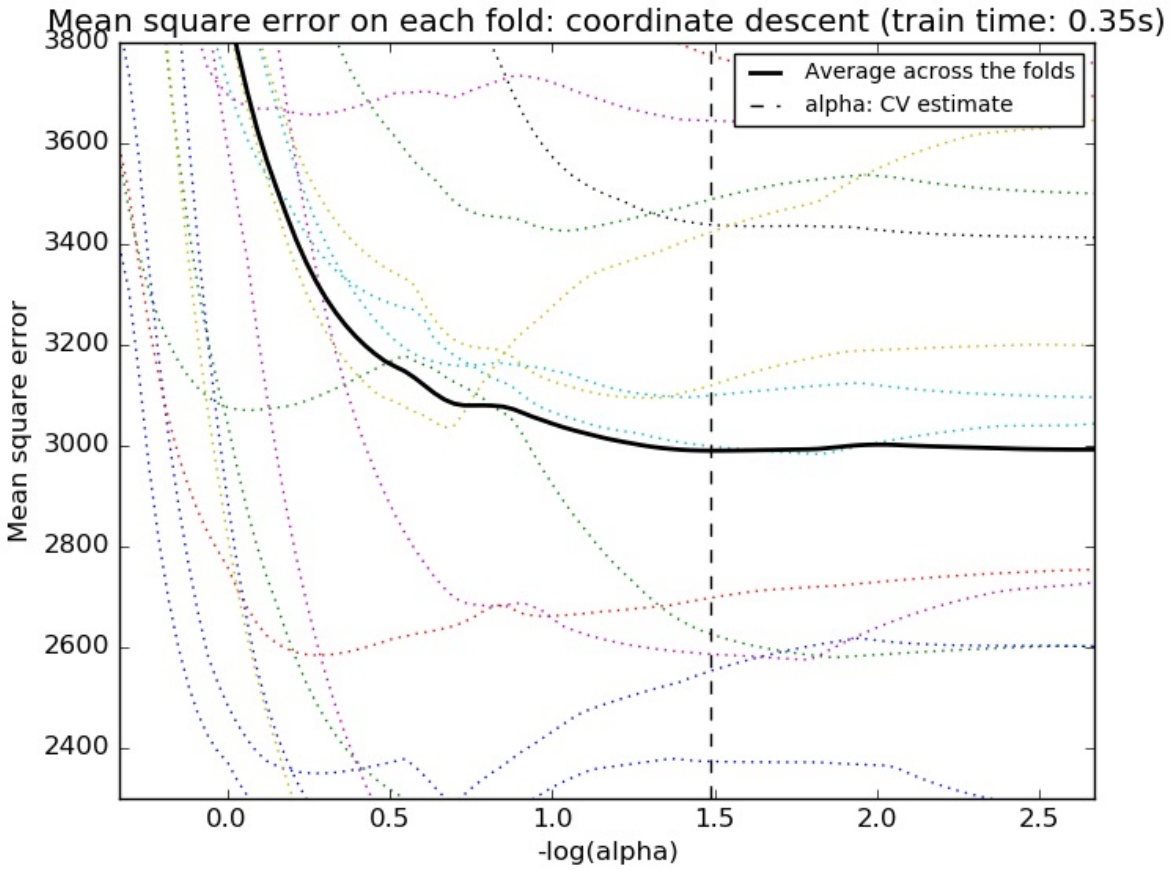
1.1.3.1. 设定正则化参数

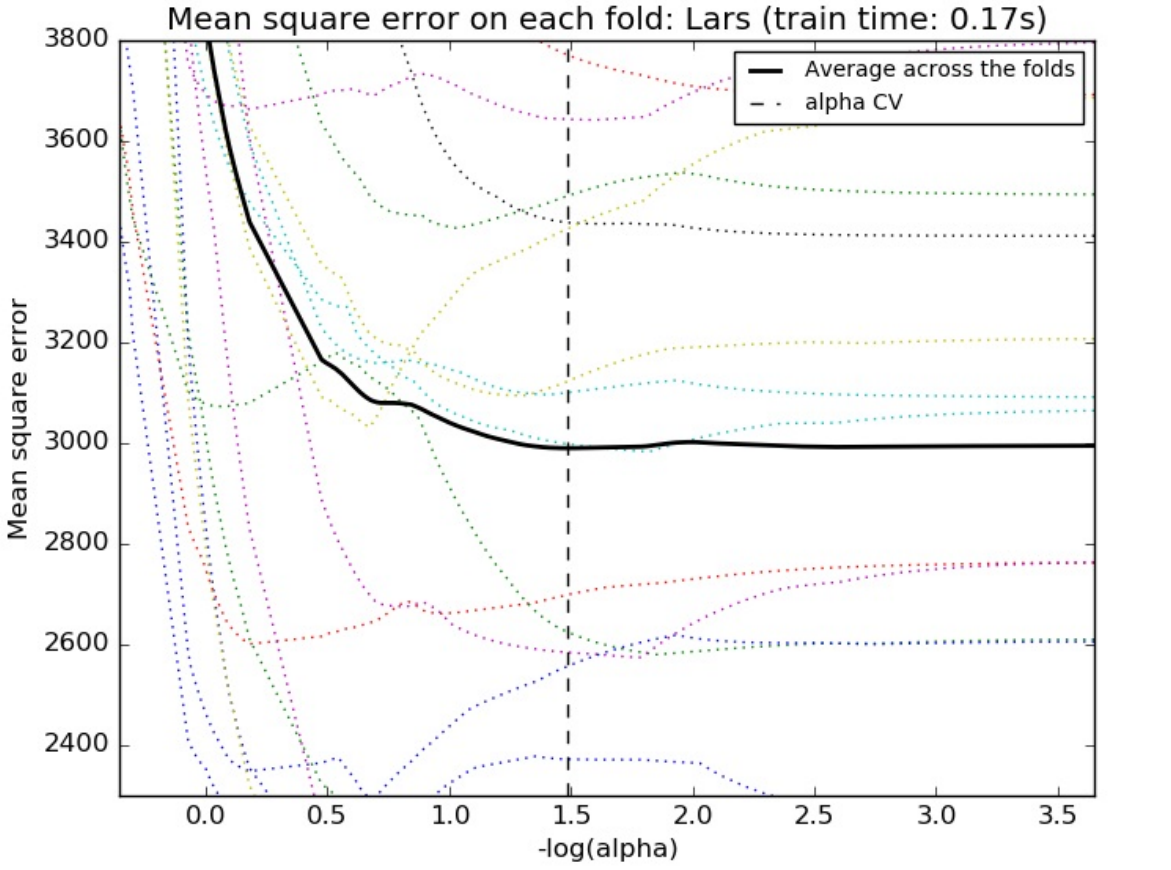
参数 `alpha` 用来控制所得到模型系数的系数程度

1.1.3.1.1. 使用交叉验证

使用scikit-learn提供的一些类，可以通过交叉验证的方式设置Lasso的 `alpha` 变量。这些类包括 `LassoCV` 和 `LassoLarsCV` 。其中 `LassoLarsCV` 基于下面介绍的最小角回归算法。

对于有很多共线性回归项的高维数据集来说，LassoCV更为适用。但是，LassoLarsCV的优势在于，它可以找出更多alpha的相关值，而且如果样本数量显著小于变量数量，它比LassoCV更快。


Mean square error on each fold: coordinate descent (train time: 0.35s)
Average across the folds
alpha: CV estimate
Mean square error
-log(alpha)
3800
3600
3400
3200
3000
2800
2600
2400
0.0
0.5
1.0
1.5
2.0
2.5


Mean square error on each fold: Lars (train time: 0.17s)
Average across the folds
alpha CV
Mean square error
-log(alpha)
3800
3600
3400
3200
3000
2800
2600
2400
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5

1.1.3.1.2 基于信息准则的模型选择

作为另一种可选方案，`LassoLarsIC`使用赤池信息量准则（AIC）和贝叶斯信息量准则（BIC）进行预测。在使用k折交叉验证方法时，该预测器只计算一遍正则化路径来寻找alpha的最优值，而不像其它预测其那样计算k+1遍。因此这种方法所需的计算资源更少。然而，这种准则需要对最终结果的自由度有一个正确预测。它是由大量样本（渐进结果）得来，而且假设模型是正确的（也就是说，数据就是由这个模型产生的）。如果所解问题badly conditioned（特征数多于样本数），那么这种方法也容易失败。

例子

- Lasso模型选择：交叉验证/AIC/BIC

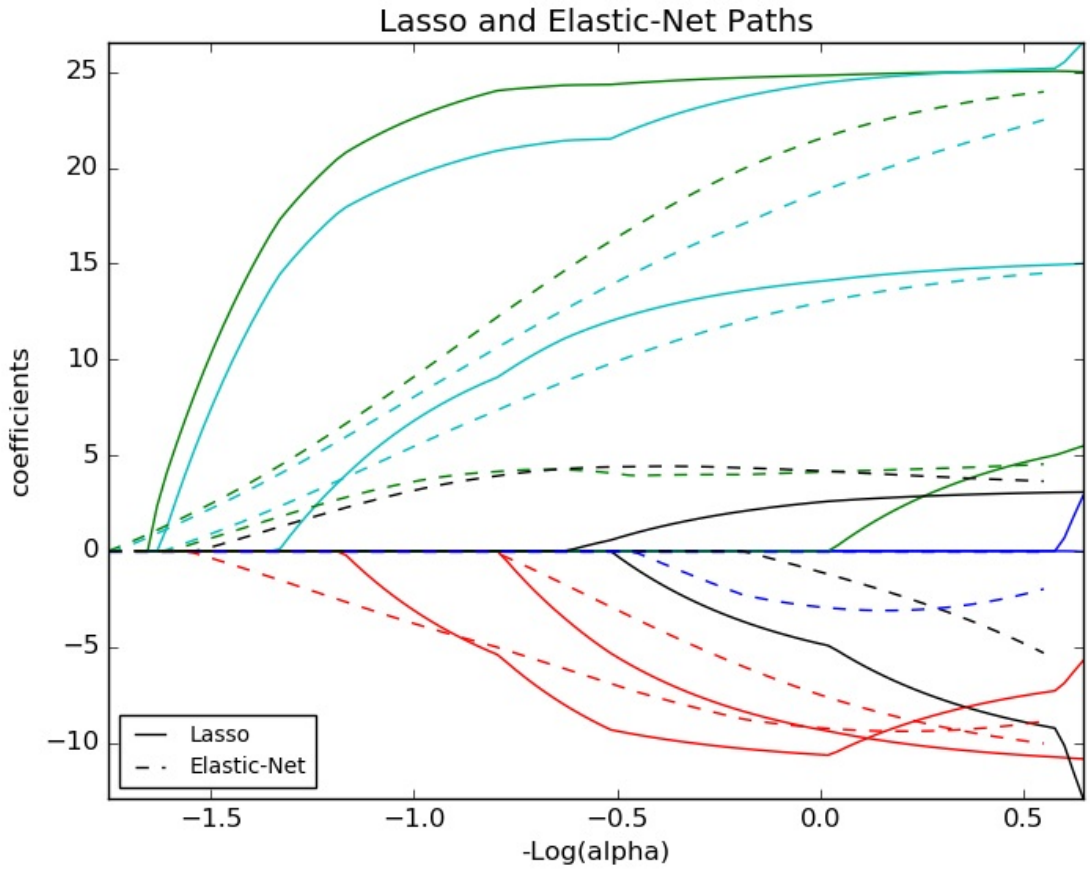
1.1.4. 弹性网

弹性网（`ElasticNet`）同时使用L1和L2正则化项来训练线性回归模型。使用这样的组合可以得到一个稀疏模型，也就是像Lasso那样只有少量权重是非零的，同时这样的模型还能够保持Ridge的正则化特性。我们使用参数`l1_ratio`来控制L1和L2的凸组合。

当多个特征互相相关时，弹性网就可以派上用场。面对这样的情况，Lasso可能会随机选择一个特征，而弹性网会两个都选。

在Lasso和岭回归中选择一个平衡点的应用优势在于，它允许弹性网在轮转情况（rotation）下继承岭回归的某些稳定性。其实质是最小化如下目标函数：

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



`ElasticNetCV` 类可以用来设置参数 `alpha` ({% math %}\alpha{% endmath %}) 和 `l1_ratio` ({% math %}\rho{% endmath %})，使用的方法仍然是交叉验证方法。

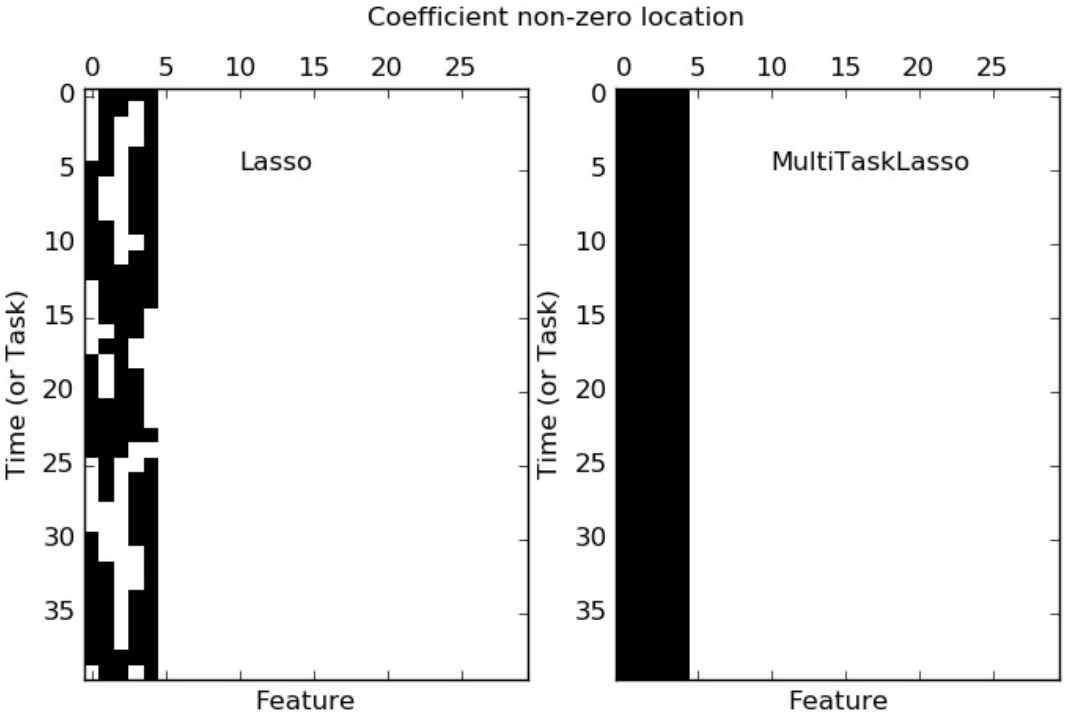
例子

- 用于稀疏信号的Lasso和弹性网
- Lasso和弹性网

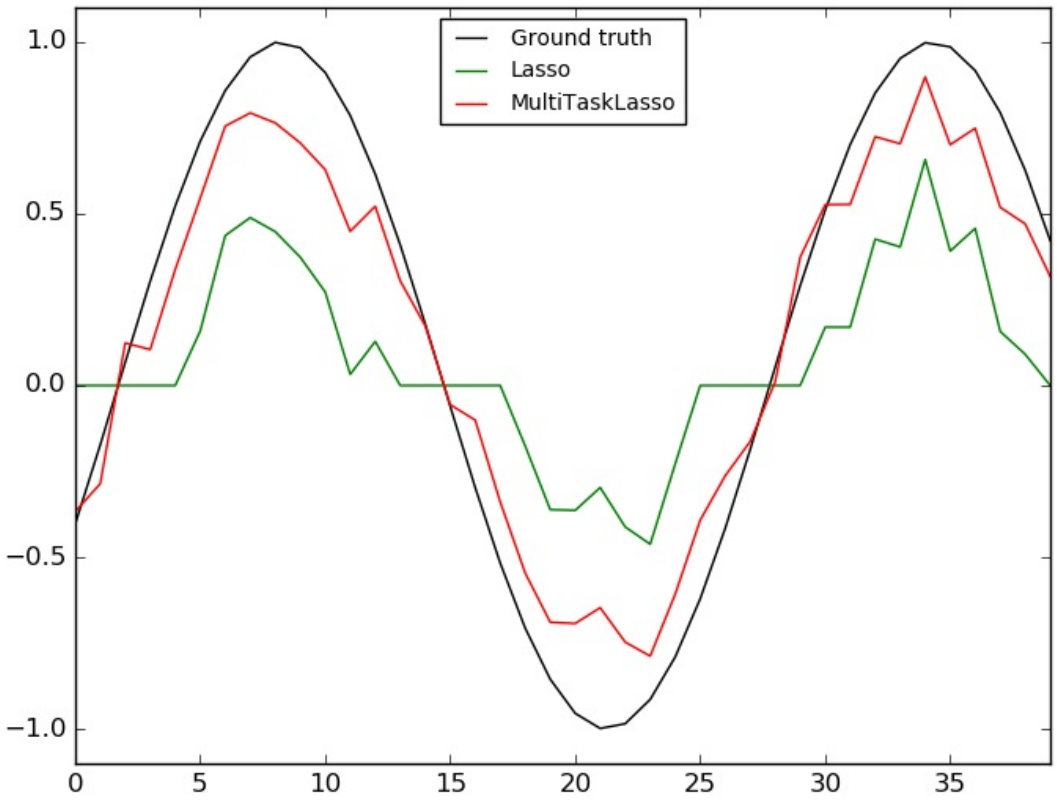
1.1.5 多任务Lasso

多任务Lasso（ `MultiTaskLasso` ）对多元联合回归问题给出一个系数稀疏的线性模型。这里 `y` 是一个n_samples * n_tasks的二维数组。其限制是选出的特征对所有回归问题（也叫作任务）都一样。

下图比较了使用简单Lasso和多任务Lasso得到的权重中非零系数的位置。Lasso得到的是离散的非零点，而多任务Lasso得到的是一整列。



下图给出了两种模型对时间序列模型的拟合比较（？Fitting a time-series model, imposing that any active feature be active at all times.）



例子：

- 使用多任务Lasso进行联合特征选择

数学上讲，该模型使用了混合的{% math %}\ell*1{% endmath %}和*{% math %}\ell_2{% endmath %}正则项来训练线性模型，即求解下述最优函数的最小值： *{% math %}* \min_w *frac{1}{2n*{\mathrm{samples}}}||XW-Y||^22 + *alpha* ||*W*||{21} {% endmath %} 其中 {% math %} ||W||*{21} = *sum_i* *sqrt{*sum_j* w{ij}^2} {% endmath %} `MultiTaskLasso` 类的实现使用了坐标下降算法来拟合系数。

1.1.6. 最小角回归（LARS）

最小角回归（Least-angle regression (LARS)）是一种用于高维数据的回归算法，由Bradley Efron, Trevor Hastie, Iain Johnstone和Robert Tibshirani共同提出。

LARS的优点在于

- 当p>>n时（即数据集维度远大于数据数量时），计算效率很高。
- 计算速度与前向选择算法（forward selection）一样快，和普通最小二乘算法有同阶的复杂度。
- 产生的是完整的，逐步的线性求解路径，在交叉验证或其他类似模型调优时尤其有用。
- 如果两个自变量与因变量（响应变量）的相关性近乎等同，那么它们的系数应该以大致相同的速率增加。算法过程正与此直觉相符，而且更加稳定。
- 可以很容易地修改来为其它预测器（如Lasso）产生结果。

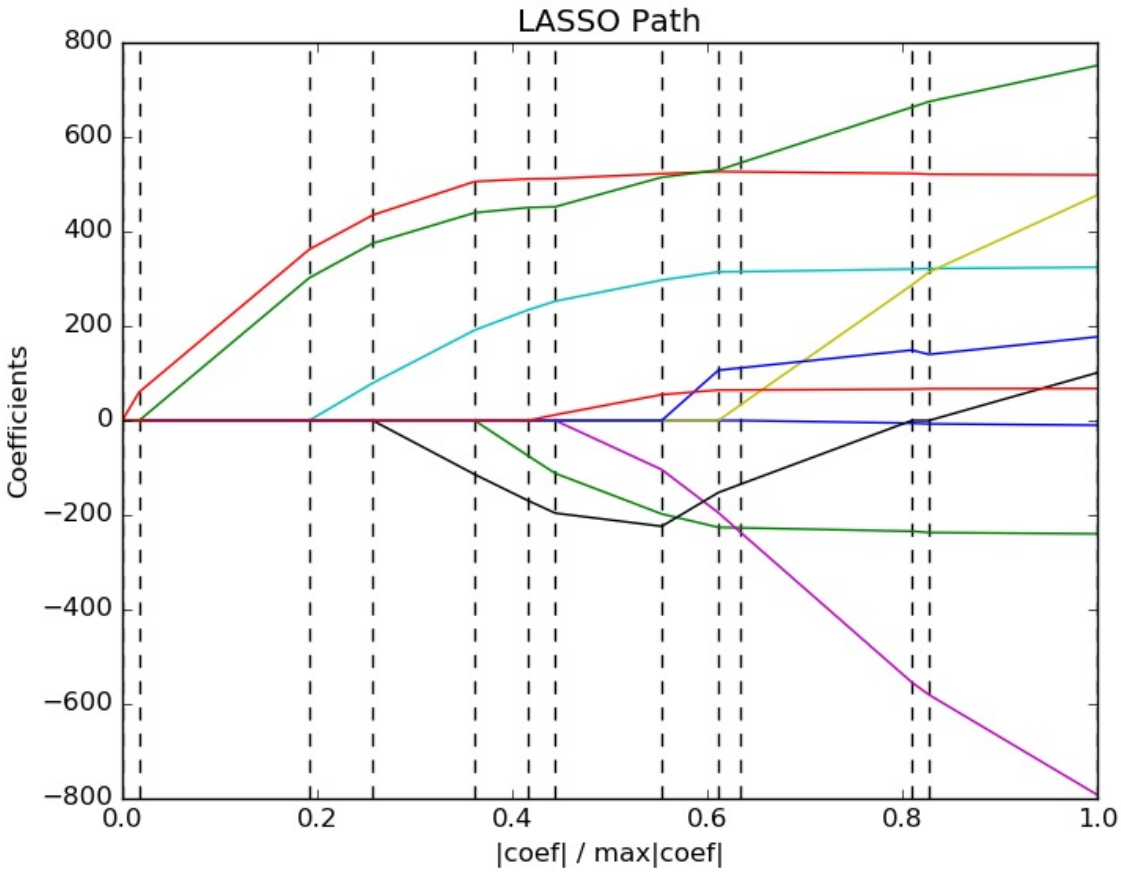
LARS的缺点包括

- LARS是对残差项迭代重拟合，因此对数据中的噪声尤其敏感。详细讨论可见Efron et al. (2004) Annals of Statistics article一文中的讨论部分（由Weisberg所写）。

可以使用 `Lars` 或者更底层的实现 `lars_path` 来训练LARS模型。

1.1.7. LARS Lasso （本小节翻译蹩脚，需要优化）

`LassoLars` 是一种使用LARS算法实现的lasso模型。和基于坐标下降的实现不同，该模型产生确切解，是其系数范数函数的逐步线性 (which is piecewise linear as a function of the norm of its coefficients)。




```
>>> from sklearn import linear_model
>>> clf = linear_model.LassoLars(alpha=.1)
>>> clf.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)
>>> clf.coef_
array([ 0.717157...,  0.        ])
```

例子

- 使用LARS产生的Lasso路径

Lars算法给出了系数沿着正则化参数的完整路径（almost for free？），因此通常可以使用函数 `lars_path` 获得该路径。

1.1.7.1 数学表述

算法与前向逐步回归类似，但是LARS Lasso算法并不在每一步都包含变量，而是将预测到的参数沿与残差相关度同角度的方向增加。

LARS Lasso给出的不是一个向量解，而是一组解的曲线。每个解都对应于一个参数向量的L1范数。完整的系数路径存储在数组 `coef_path_` 中，大小为n_features * max_features + 1。第一列恒为0。

参考文献

- 原始算法的细节可参考Hastie等人所著的《最小角回归》一文。

1.1.8. 正交匹配追踪（Orthogonal Matching Pursuit, OMP）

`OrthogonalMatchingPursuit` 和 `orthogonal_mp` 实现了OMP算法。该算法使用非零系数数目（即L0伪范数）作为限定条件，来拟合线性模型。

和最小角回归一样，OMP也是一个前向特征选择模型。在非零元素个数确定的情况下，该算法可以逼近最优解向量：

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

OMP也可以针对某个给定的误差值，而不用非零元素个数作为目标进行优化

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

OMP是一种贪心算法，在每一步都选择和当前残差最相关的项。它与简单匹配追踪方法（MP）相似，但是在每步迭代中的表现都更好一些。因为每步中残差都使用到之前选择的字典元素所组成空间的正交投影重新计算。

例子

- 正交匹配追踪

参考文献

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

1.1.9. 贝叶斯回归

使用贝叶斯回归技术，可以在预测阶段就将正则项参数包括进来：正则化参数不再是一个被写死的固定值，而是可以根据手头数据调优。

达到这一效果的方法是引入模型超越参数上的无信息先验。岭回归中使用的ℓ_2正则项所起到的作用在于，在参数集w上的高斯先验下寻找一个准确度为λ^{-1}的最大化后验的解。这个解可以被看作是可从数据中预测的随机变量，而不用手动去设置λ。

为了得到一个完整的概率模型，我们假设输出y是关于Xw的高斯分布：

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

其中α是从数据中预测的随机变量。

贝叶斯回归的优点在于：

- 适应于手头数据
- 可以用来在预测阶段就将正则参数包含进来

其缺点在于

- 模型推导会比较花时间

参考文献

- Bishop的经典著作PRML中有关于贝叶斯方法的很好介绍
- 原始算法细节可见Radford M. Neal所著*Bayesian learning for neural networks*一书

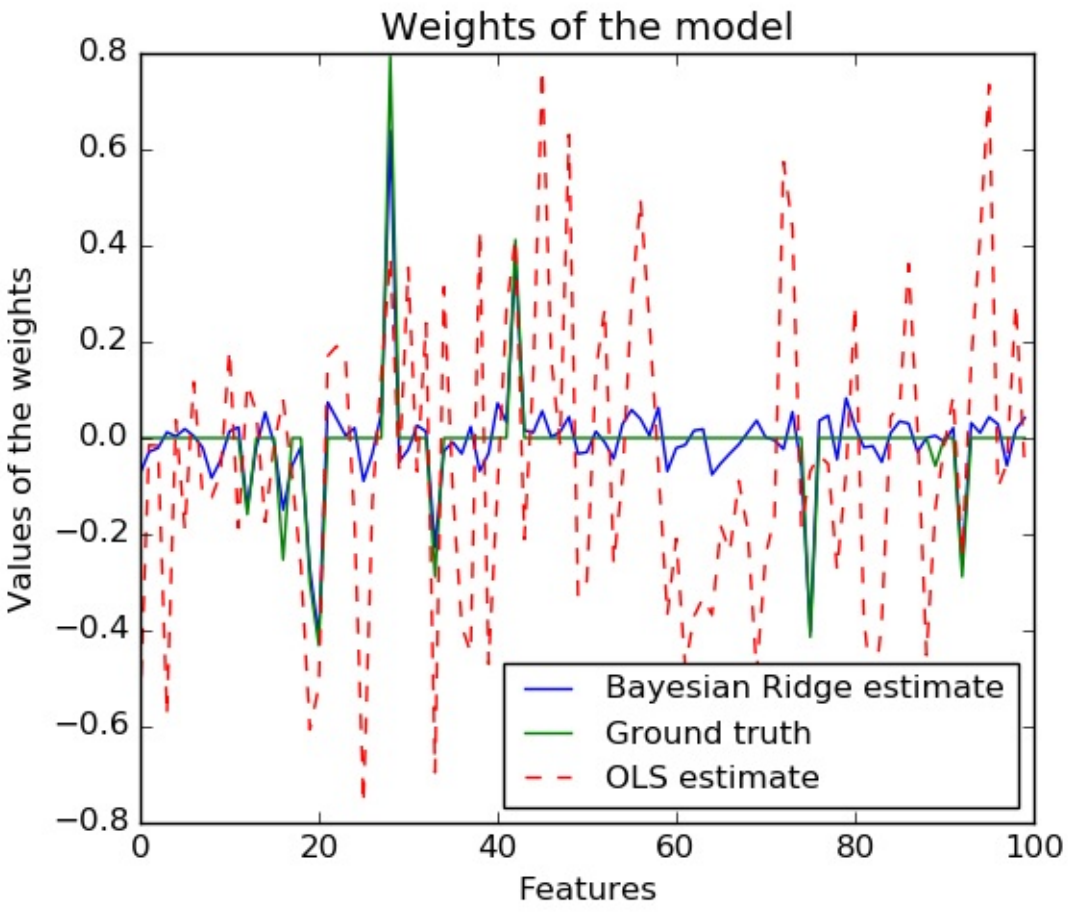
1.1.9.1. 贝叶斯岭回归

`BayesianRidge` 使用上述原理对给定的回归问题建立概率模型。参数w的先验分布由球面高斯分布（spherical Gaussian）给出：

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

α和λ上的分布由Γ-分布给出。

得到的模型称为贝叶斯岭回归，其与经典的岭回归模型 `Ridge` 类似。参数w, α和λ在拟合模型的过程中联合求得。剩余的超越参数是α和λ上Γ分布的参数，通常被选作是无信息的。这些参数通过最大化边际对数似然（marginal log likelihood）得到。默认值是
$\alpha_1 = \alpha_2 = \lambda_1 = \lambda_2 = 1 \times e^{-6}$



贝叶斯岭回归可以被用于回归分析：

```
>>> from sklearn import linear_model
>>> X = [[0., 0.], [1., 1.], [2., 2.], [3., 3.]]
>>> Y = [0., 1., 2., 3.]
>>> clf = linear_model.BayesianRidge()
>>> clf.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)
```

训练后，模型可以用来预测新值：

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

通过变量 `coef_` 获取系数w：

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

由于贝叶斯框架的原因，该方法得到的权重与通过普通最小二乘得到的参数略有差别。不过贝叶斯岭回归对ill-posed问题有更强的鲁棒性。

例子

- 贝叶斯岭回归

参考文献

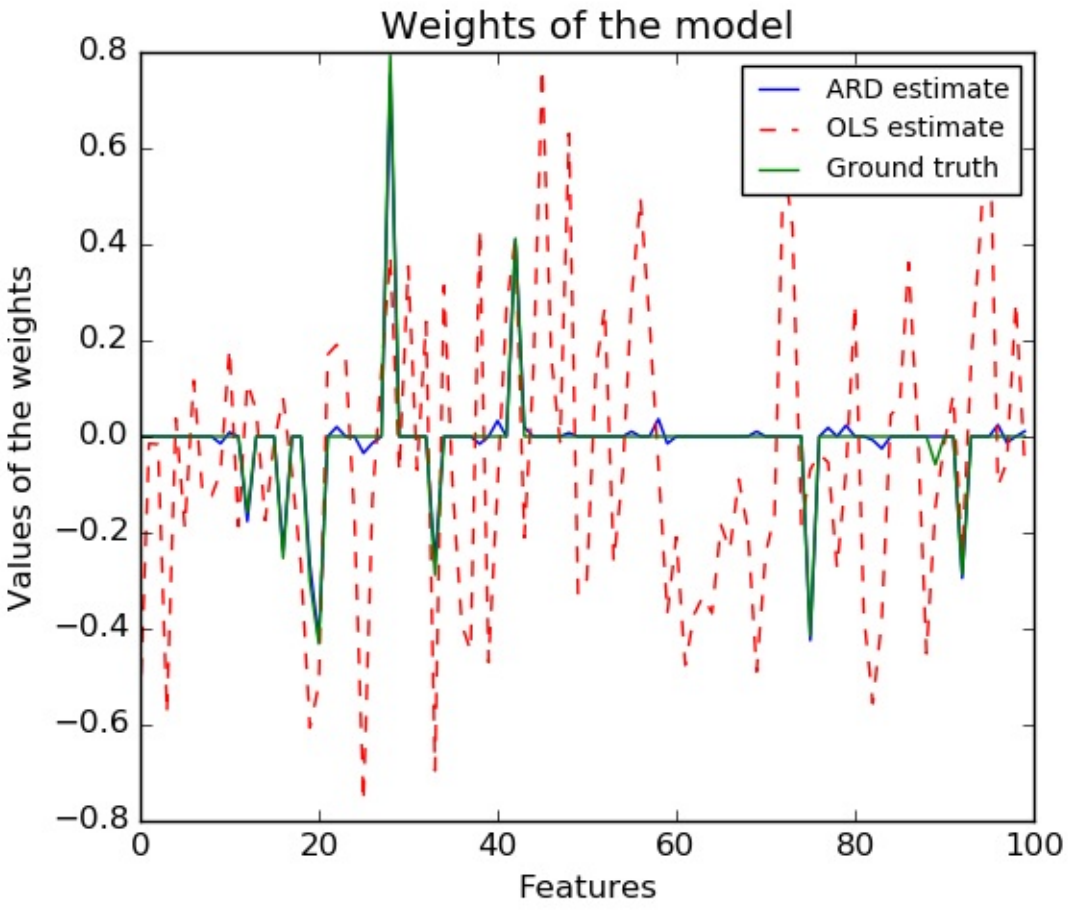
- 更多细节可参考MacKay, David J. C.所著*Bayesian Interpolation*一文

1.1.9.2. 自动相关判定（Automatic Relevance Determination, ARD）

`ARDRegression` 和贝叶斯岭回归很像，但是会得到更稀疏的w[1][2]。ARDRegression使用了一个不同的w的先验分布：它不再假设高斯分布是球面的，而是认为是平行于坐标轴的椭球形高斯分布。这意味着每个从高斯分布得到的权重w_i都以0位中心，精确度为λ_i。

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

与贝叶斯岭回归不同的是，每个权重w_i都有自己的标准差λ_i。所有λ_i上的先验分布与给定超越参数λ_1和λ_2的Γ分布相同。



例子

- 自动相关判定回归(ARD)

参考文献

- [1] Christopher M. Bishop: *Pattern Recognition and Machine Learning*, Chapter 7.2.1
- [2] David Wipf & Srikantan Nagarajan: *A new view of autonomic relevance determination*

1.1.10. Logistic回归

虽然名字中带着“回归”两字，但实际上Logistic回归一般是用来解决分类问题，而不是回归问题。在文献中，该方法也被称为“Logit回归”、“最大熵分类（MaxEnt）”或者“对数-线性分类器”。该模型使用logistic函数对给定样本属于不同类别的概率进行建模。

Scikit-learn使用 `LogisticRegression` 实现Logistic回归。可以通过该实现拟合一个多类别（一对多）Logistic回归模型。拟合时还可以根据需要选择是否增加L1或者L2惩罚项。

从优化问题的角度看，带有L2正则项的二元Logistic回归是要求解下式

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

类似的，如果是使用L1正则项，则是求解

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

LogisticRegresssion实现的求解器有“liblinear”（C++库LIBLINEAR）的封装、“newton-cg”、“lbfgs“和”sag“。其中“newton-cg”和“lbfgs“只支持L2惩罚项，这两个求解器在某些高维数据上收敛更快。使用L1惩罚项会得到稀疏的预测系数。

求解器“liblinear”在Liblinear的基础上实现了坐标下降（CD）算法。对于L1惩罚项，可以使用sklearn.svm.l1_min_c来计算C的一个更深的下界，以得到一个非“空”（这里“空”的意思是所有特征的权重都为0）模型。该实现得益于底层出色的LIBLINEAR库。然而，liblinear实现的坐标下降算法不能够学到一个真正的多项（多类别）模型；而是把优化问题分解为多个一对多问题，所以会对每个类别各自训练一个二元分类器。不过这些分解都是在底层进行，对用户透明，因此 `LogisticRegression` 使用这个求解器作为多类别分类器（？）。

使用 `LogisticRegression` 时，将 `solver` 设成“lbfgs”或者“newton-cg”，并将 `multi_class` 设为“multinomial”，可以学习到一个真正的多项logistic回归模型。这意味着使用该模型预测得到的概率应该会比默认的一对多方法得到的结果准确度更高。然而这两个求解器（以及“sag”）不能优化带有L1惩罚项的模型，因此“multinomial”不能学到稀疏模型。

“sag”求解器使用了随机平均梯度下降算法[3]。它不能处理多类别问题，而且只能使用带有L2惩罚项的模型。但是在大数据集上（样本量和特征数都很大的情况下）它比其它求解器要快。

简而言之，下表给出了一个粗略的求解器选择标准

情况	求解器
小数据集，或需要L1惩罚项	“liblinear”
对多类别问题误差敏感	“lbfgs”或者“newton-cg”
大数据集	“sag”

对于大数据集，也可以考虑使用 `SGDClassifier` ，该分类器使用了“对数”损失函数。

例子

- Logistic回归中的L1惩罚项与稀疏性
- L1-线性回归的路径

与liblinear的不同

如果将 `fit_intercept` 设为 `False` ，而且拟合出的 `coef_` （或）被预测的数据为0，则设置了 `solver=liblinear` 的 `LogicRegression` 或 `LinearSVC` 得到的得分与直接使用底层外部库得到的得分会有差别。因为对于 `decision_function` 值为0的样本， `LogicRegression` 和 `LinearSVC` 将其划分为负类，而liblinear库划分为正类。需要注意的是，如果出现了这种情况（即设置 `fit_intercept` 为 `False` 且很多样本的 `decision_function` 值为0）通常意味着产生了欠拟合，得到的模型不是一个好模型。在这种情况下，建议将 `fit_intercept` 设为 `True` 并增大 `intercept_scaling` 。

注解：使用稀疏**Logistic**回归进行特征选择

使用L1惩罚项的logistic回归会得到稀疏模型，因此可以用来进行特征选择。细节可参见基于L1的特征选择。

`LogisticRegressionCV` 同样实现了Logistic回归。该类是使用了自带的交叉验证方法来找到最优的C。由于“暖启动”机制，“newton-cg”、“sag”和“lbfgs”求解器在高维密集数据上速度更快。对于多类别问题，如果参数 `multi_class` 被设成ovr，是对每个类别获得一个最优的C；如果被设成Multinomial，则最优的C使得交叉熵损失（cross-entropy loss）最小。

参考文献： [3]. Mark Schmidt, Nicolas Le Roux, and Francis Bach: Minimizing Finite Sums with the Stochastic Average Gradient.

1.1.11. 随机梯度下降（SGD）

随机梯度下降是一种简单但是非常有效的拟合线性模型的方法。当样本数（以及特征数）非常大时该方法尤其有效。`partial_fit`方法支持在线/核外学习。

`SGDClassifier`和`SGDRegressor`允许用户使用不同的（凸）损失函数和不同的惩罚项，分别针对分类问题和回归问题训练线性模型。例如将`loss`设为“log”，则SGDClassifier会拟合一个logistic回归模型；设成“hinge”则拟合一个线性支持向量机（SVM）。

参考文献

- 随机梯度下降

1.1.12. 感知机

`Perceptron`是另一种适用于大规模学习的简单算法。默认情况下：

- 该算法不需要学习率
- 该算法不被正则化
- 该算法只有在遇到错误的情况下才更新模型

最后一个特征表明感知机比使用合页损失函数（hinge loss）的SGD训练起来要快，而且得到的模型更稀疏。

1.1.13. 被动攻击算法（Passive Aggressive Algorithm）

被动攻击算法是一种适用于大规模学习的算法。它们与感知机类似的地方是也不需要设置学习率。但是不同在于这些算法会包含一个正则项参数`C`。

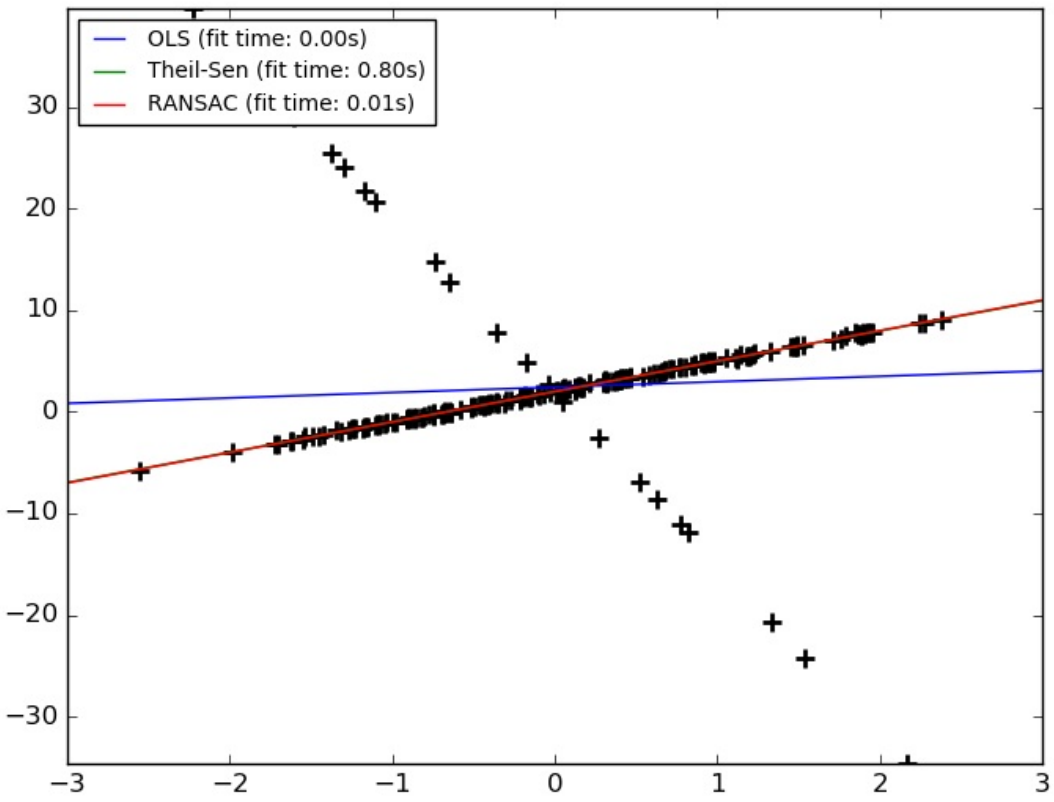
对于分类问题，可以使用`PassiveAggressiveClassifier`，其中损失函数`loss`可以设为“hinge”（PA-I）或者“squared_hinge”（PA-II）。对于回归问it，可以使用`PassiveAggressiveRegressor`，其中损失函数`loss`可以设为“epsilon_insensitive”（PA-I）或者“squared_epsilon_insensitive”（PA-II）。

参考文献

- K. Crammer, O. Dekel, J. Keshat, S. Shalev-Shwartz, Y. Singer, "*Online Passive-Aggressive Algorithms*", JMLR 7, 2006

1.1.14. 稳健回归（鲁棒回归）：异常值与模型误差

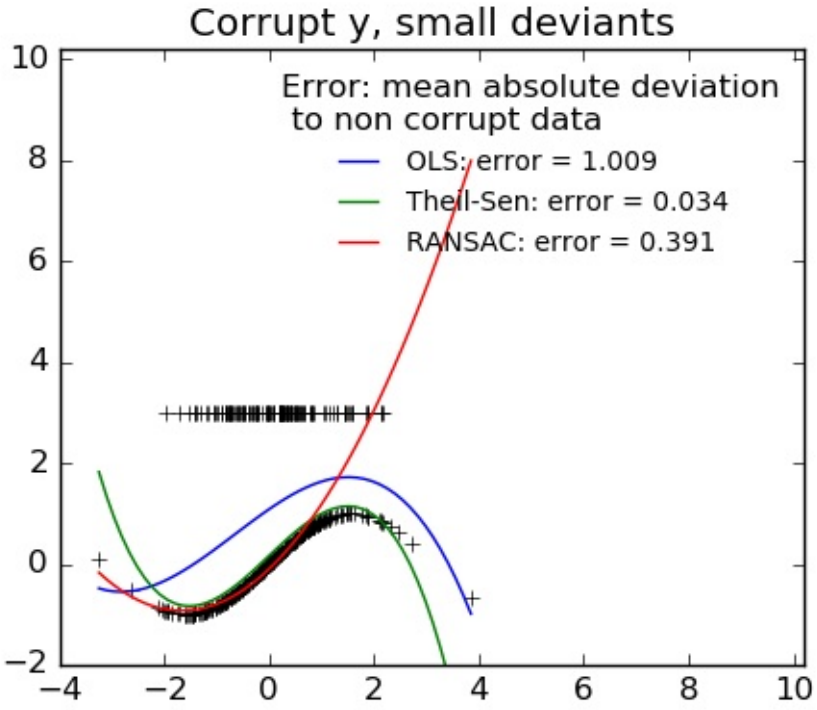
稳健回归所着力解决的问题是，如果数据中有坏点（或是异常值或是模型中的误差），如何拟合回归模型。

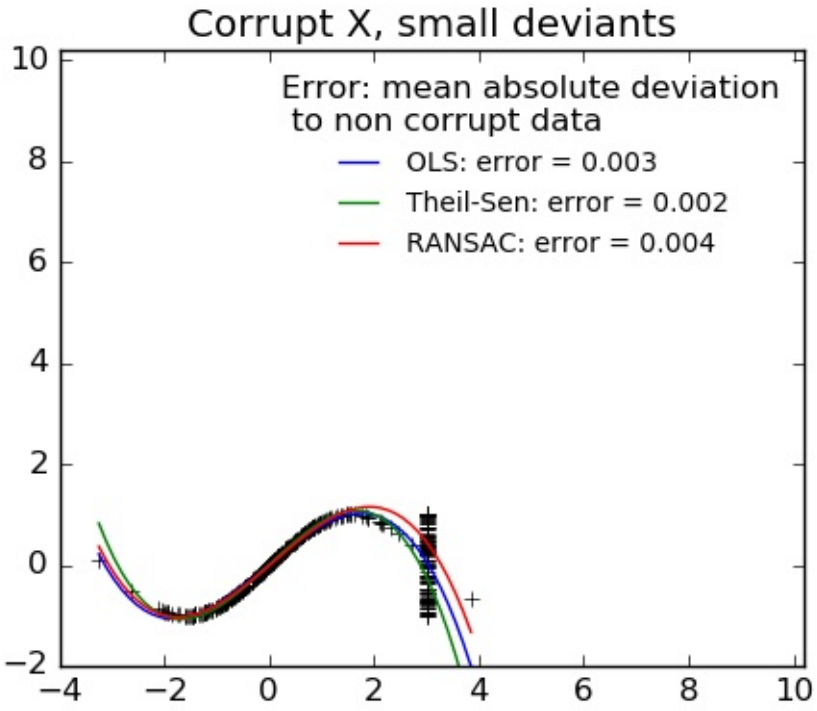


1.1.14.1. 不同的场景和一些有用的概念

处理带异常值的数据时，有些事情要谨记于心

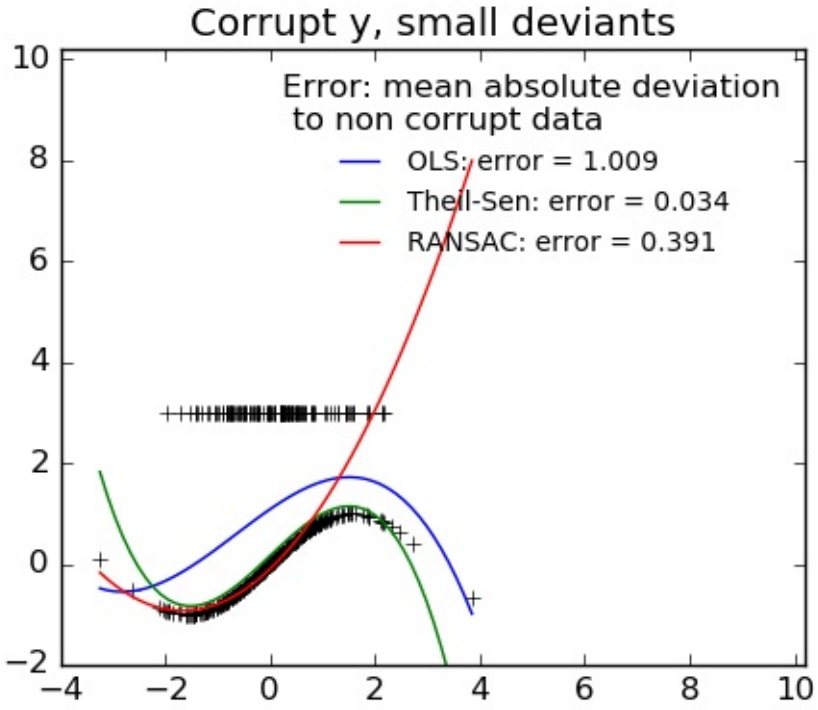
- 异常值出现在**X**中还是在**y**中？

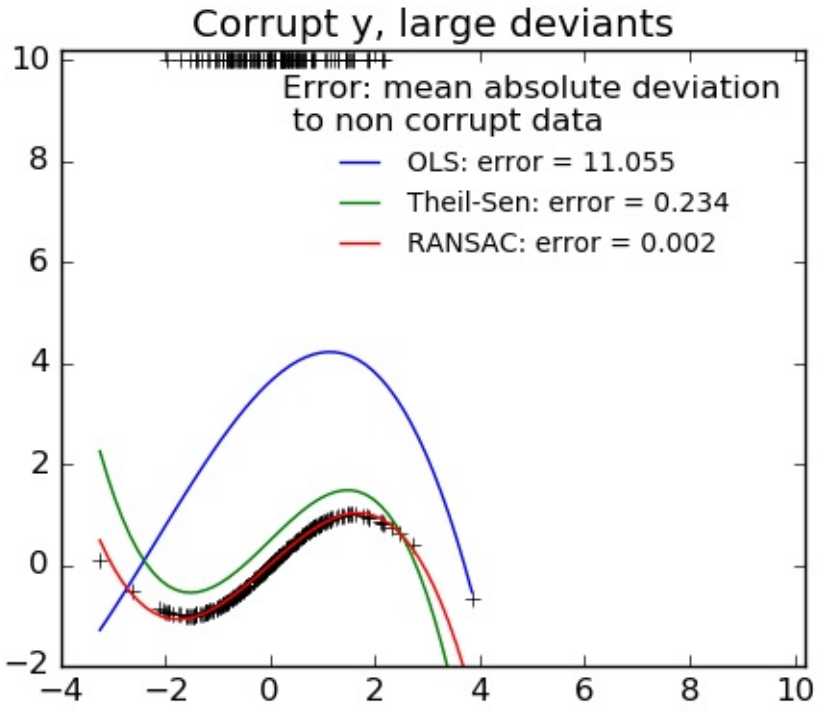




- 异常值比例 **vs.** 误差偏移量

异常点个数对模型影响很大，但是偏差多少也很重要





An important notion of robust fitting is that of breakdown point: the fraction of data that can be outlying for the fit to start missing the inlying data.

需要注意的是，稳健拟合在高维度数据集（特征个数很大的数据集）中很难实现。稳健模型在这些情况下可能不会工作。

综合考虑：使用哪个预测器？

Scikit-learn提供两种稳健回归预测器：RANSAC和Theil Sen

- RANSAC更快，样本数量变大时规模可扩展性更好（译者认为，可以看作是对更大的样本量表现更好）。
- 如果在X轴方向出现了中等大小的异常值，则Theil Sen表现更好。但是如果数据维度太高，这样的优势就没有了。

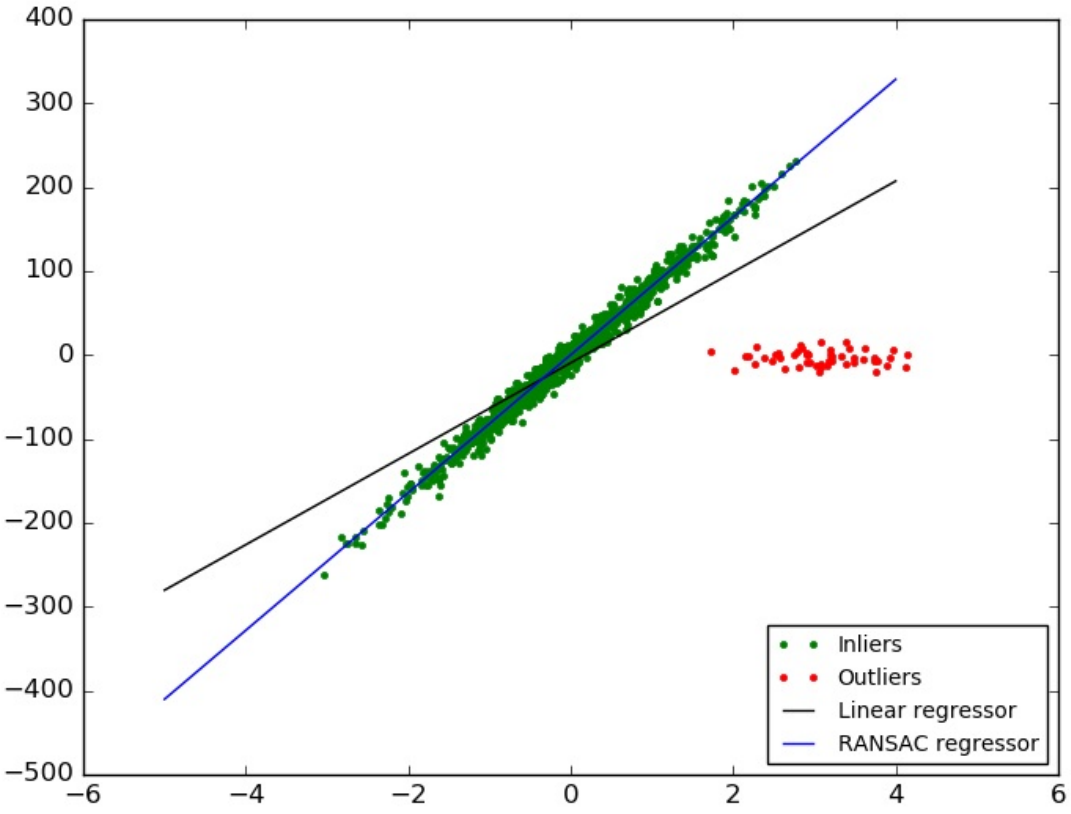
如果不知道该用哪个，就用RANSAC。

1.1.14.2. RANSAC: 随机抽样一致性算法

RANSAC（随机抽样一致性算法）从完整的数据集中随机选择一个由正常点组成的子集，并在该子集上拟合模型。

RANSAC是一种非确定性算法，仅以特定概率（依赖于迭代次数，即参数 `max_trials` ）产生一个比较合理的结果。该算法对线性回归和非线性回归问题都适用，在摄影测量计算机视觉（photogrammetric computer vision）这个领域中尤其受欢迎。

该算法将完整的输入样本集分成一系列正常值（可能受到噪声影响）和异常值（因错误的测量手段或对数据的非法假设而产生）。最终只使用得到的正常值拟合模型。



1.1.14.2.1. 算法细节

在每次迭代中都完成以下工作

1. 从原始数据中随机选择 `min_samples` 个样本，检查数据集是否合法（参看 `is_data_valid` 。
2. 在这个随机得到的数据集上拟合模型，检查得到的模型是否合法（参看 `is_model_valid` ）。
3. 对每条数据，计算对应y与模型预测值的残差（ `base_estimator.predict(X)-y` ），判断数据是正常点还是异常点。所有残差绝对值小于阈值 `residual_threshold` 的数据都被认为是正常点。
4. 如果当前得到的正常值数量是开始迭代以来最高的（严格好于之前的最大值），则把该模型设为当前最佳模型。

迭代停止的条件为达到最大迭代次数（ `max_trials` ）或者满足其它指定的终止条件（见 `stop_n_inliers` 和 `stop_score` ）。最终模型使用之前得到的最佳模型里的正常值（一致集）进行拟合。

函数 `is_data_valid` 和 `is_model_valid` 允许开发人员发现和舍弃随机子样本的退化组合。如果预测模型不需要找出退化情况，则仍然要使用 `is_data_valid` 。因为这个函数在模型拟合之前被调用，可以提高计算性能。

例子

- 使用RANSAC进行稳健线性模型预测
- 稳健线性预测器拟合

参考文献

- http://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)

1.1.14.3. Theil-Sen预测器：基于广义中位数的预测器

`TheilSenRegressor` 模型使用了高维空间中的广义中位数，因此对多变量异常值是鲁棒的。需要注意的是，即便如此，该模型的鲁棒性还是会随着求解问题维度的增加而快速减少。在高维空间中，它的鲁棒性会越来越差，最后不比普通最小二乘强多少。

例子：

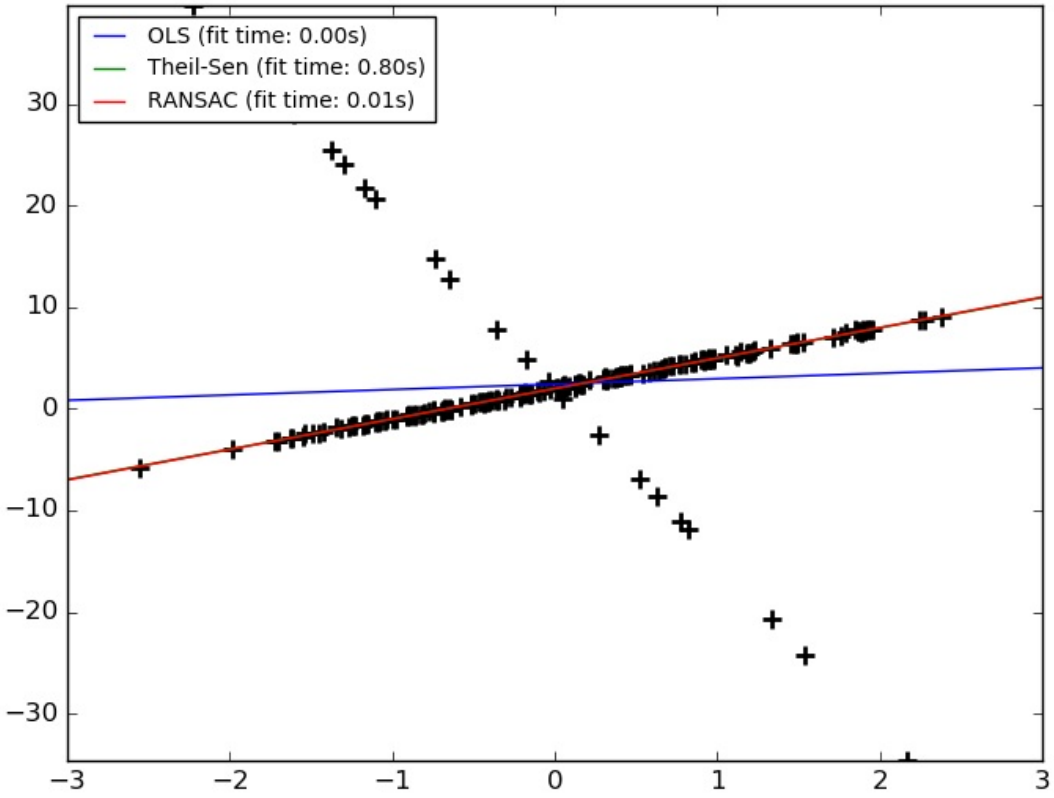
- Theil-Sen回归
- 稳健线性预测器拟合

参考文献

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

1.1.14.3.1. 理论支持

`TheilSenRegressor` 在逼近效率上可与普通最小二乘（OLS）媲美，而且是一种无偏模型。和OLS不同的是，Theil-Sen是一种非参数方法，这意味着它并不对数据服从的分布做任何假设。由于Theil-Sen是基于中位数的，因此它对坏点（异常点）更加鲁棒。在单变量情况下，Theil-Sen和简单线性回归相比能多容忍29.3%的坏点。



在Scikit-learn的实现中，`TheilSenRegressor` 使用了一种对多变量线性回归模型的泛化算法。该算法使用了稀疏中位数（在多元空间中对中位数概念的一个推广）。

Theil-Sen算法的时间复杂度和空间复杂度随

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

变化。这说明当有大量数据和特征时该算法会不再使用。在这种情况下，可以只考虑所有可能组合的一个子集来控制复杂度。

例子

- Theil-Sen回归

参考文献

- [4] Xin Dang, Hanxiang Peng, Xueqin Wang and Heping Zhang: Theil-Sen Estimators in a Multiple Linear Regression Model.
- [5] T. Kärkkäinen and S. Äyrämö: On Computation of Spatial Median for Robust Data Mining.

1.1.15. 多项式回归：使用基本函数扩展线性模型

在机器学习领域中有一个普遍现象，就是用数据的非线性函数来训练线性模型。这种方法既保留了线性方法在速度上的优势，又拓宽了模型所拟合的数据的范围。

比如，可以通过构造系数的多项式特征来扩展简单的线性回归模型。对于标准线性回归，如果输入数据只有二维，那么构造的模型可能会像这样：

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

假设我们不想拟合出来一个平面了，而是要拟合出来一个抛物面，那么我们可以加入特征的二次多项式，得到这样的模型

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

让人略感意外的是，得到的模型其实仍然是一个线性模型：为了验证这一点，假设我们创建一个新的变量：

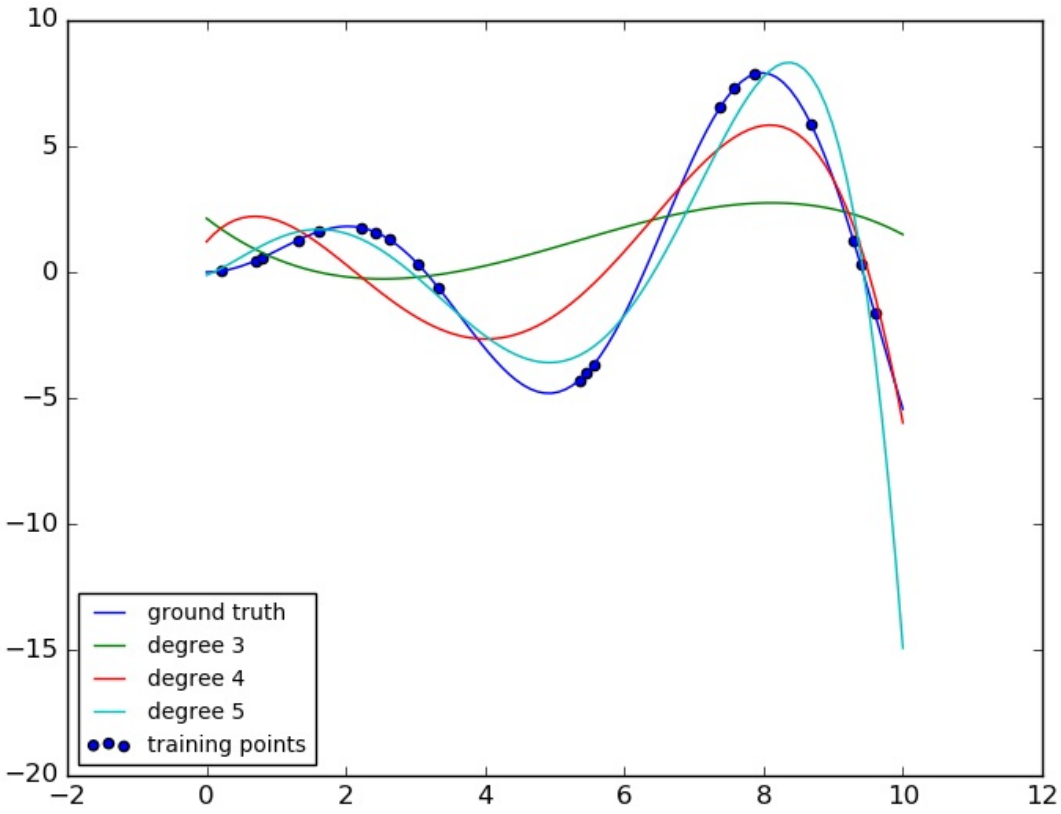
$$z = [x_1, x_2, x_1 x_2, x_1^2, x_2^2]$$

使用这种被重新标注的数据，得到的模型可以重写为

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

可以看到，得到的多项式回归与之前所述的线性回归是同一类的（即模型对w仍然是线性的），可以用同样的技术求解。这样一来，线性方法在由这些基本函数构建的更高维空间中仍然适用，模型的灵活性得到了增强，可以适用于更广的数据范围。

这里给出了一个例子，将上述思想应用到一维数据中，通过增加变量的次数来构造多项式特征：



图像创建之前使用了 `PolynomialFeatures` 进行预处理。该类将输入矩阵转化为给定度数的新矩阵，用法如下

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

特征 `X` 从$[x_1, x_2]$转化成了$[1, x_1, x_2, x_1^2, x_1 x_2, x_2^2]$，可以被任何线性模型所使用。

这种预处理可以经Pipeline这种工具流水线化。表示简单多项式回归的对象可以如下创建使用：

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

使用多项式特征训练出来的线性模型可以被精确转回至输入多项式系数上（？）

在有些情况下，没必要包含某个单独特征的高阶形式，只需要加入所谓的交互特征，即之多d个不同特征的乘积。这时可以通过将`PolynomialFeatures`的`interaction_only`设为“True”得到。

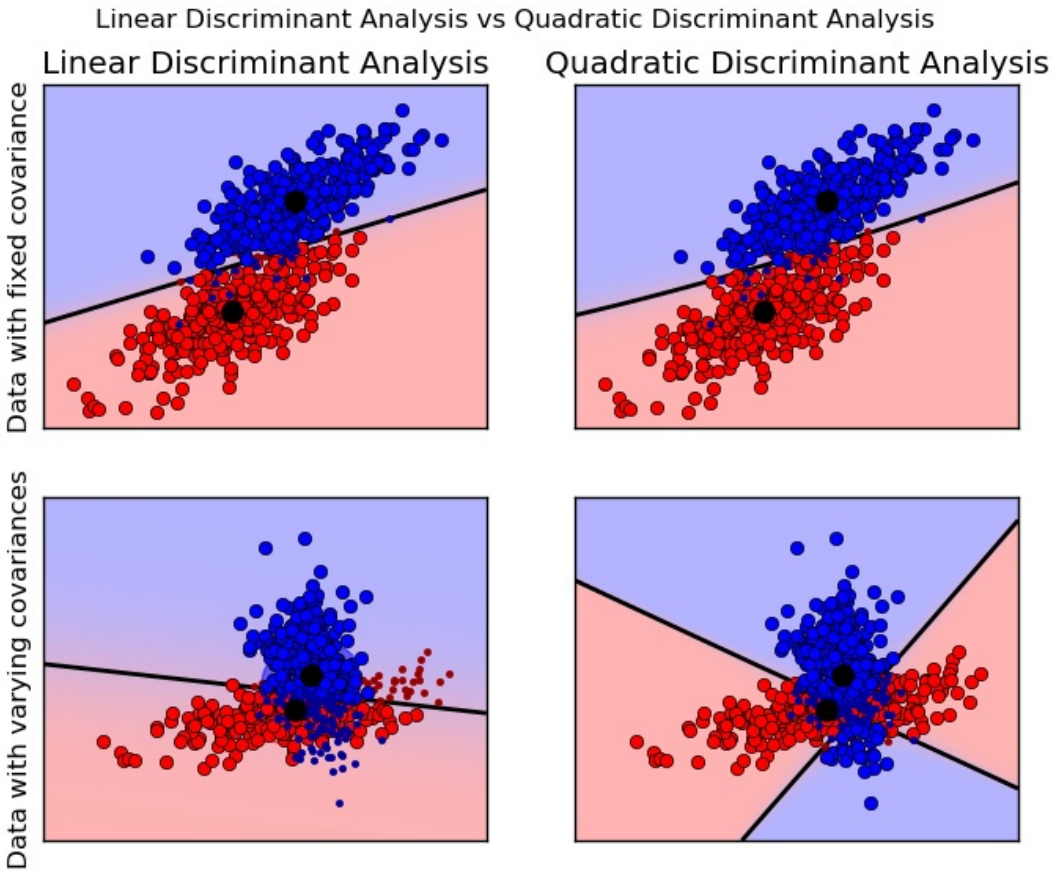
例如，如果特征是布尔值，则对所有n都有$x_i^n = x_i$，这样的特征是没用的。但是$x_i x_j$代表了两个布尔值的合取。这种情况下，可以使用线性分类器解决异或问题：

```
>>> from sklearn.linear_model import Perceptron
>>> from sklearn.preprocessing import PolynomialFeatures
>>> import numpy as np
>>> X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
>>> y = X[:, 0] ^ X[:, 1]
>>> X = PolynomialFeatures(interaction_only=True).fit_transform(X)
>>> X
array([[ 1.,  0.,  0.,  0.],
       [ 1.,  0.,  1.,  0.],
       [ 1.,  1.,  0.,  0.],
       [ 1.,  1.,  1.,  1.]])
>>> clf = Perceptron(fit_intercept=False, n_iter=10, shuffle=False).fit(X, y)
>>> clf.score(X, y)
1.0
```

1.2. 线性与二次判别分析

线性判别分析（`discriminant_analysis.LinearDiscriminantAnalysis`）和二次判别分析（`discriminant_analysis.QuadraticDiscriminantAnalysis`）是两种经典的分类器。这两种模型如它们的名字所揭示的那样，分别用作线性和二次决策面。

这些分类器的优势在于，它们有封闭形式解，而且便于计算。其与生俱来就支持多类别分类，在实际应用中表现出色，而且没有超参数需要调优。



上图给出了线性判别分析和二次判别分析的决策平面。其中底下两幅图表明，线性判别分析只能学习到线性边界，而二次判别分析可以学到二次边界，所以更加灵活。

例子

- Linear and Quadratic Discriminant Analysis with confidence ellipsoid 给出了在人造数据集上对LDA和QDA的比较。

1.2.1. 使用线性判别分析进行维度缩减

`discriminant_analysis.LinearDiscriminantAnalysis` 可以用来进行有监督的维度缩减。其原理是把输入数据投影到一个线性子空间中，而该子空间的方向将不同类别之间的距离最大化（更准确的数学语言表示请见下文）。输出数据的维度一定小于类别的个数，因此总体来讲这是一个比较强的维度缩减方法，而且只在多类别问题中有意义。

该过程在 `discriminant_analysis.LinearDiscriminantAnalysis.transform` 中实现。可以通过 `n_components` 来设置目标维度数。该参数不影响 `discriminant_analysis.LinearDiscriminantAnalysis.fit` 或 `discriminant_analysis.LinearDiscriminantAnalysis.predict` 。

例子

- Comparison of LDA and PCA 2D projection of Iris dataset：比较LDA和PCA在Iris数据集上的维度缩减性

1.2.2. LDA与QDA分类器的数学表达

无论是LDA还是QDA，都可以从简单的条件概率模型中推导出。这里的概率模型指的是给定类别k这一条件下$P(X|y=k)$这一条件概率模型。这样，可以通过贝叶斯定理求出预测值：

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

最后选择能够最大化该条件概率的k值。

更准确地说，对于线性和二次判别分析，$P(X|y)$的模型为多变量高斯分布，分布密度为：

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

为了把这个模型用作分类器，我们只需要从训练数据中估计以下值

- 所属类别的概率$P(y=k)$。该值可以通过求属于类别k的数据所占比例获得。
- 每个类别的期望μ_k。该值可以通过求属于类别k的数据的均值获得。
- 协方差矩阵Σ_k。该值可以通过分析样本数据得出，也可以通过正则化的估计函数得出。可参考下面关于“收缩（shrinkage）”的章节。

对于LDA，我们假设每个类别的高斯分布都有相同的协方差矩阵，即对所有k有$\Sigma_k = \Sigma$。这样才能得出类别之间的线性决策平面（可以通过不同类的对数概率比得出）。即

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

对于QDA，我们对高斯分布的协方差矩阵Σ_k不做任何假设，可以得到二次决策平面。更多细节可参看[3]。

注：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矩阵（即假设类别之间条件独立），则得到的分类器与高斯朴素贝叶斯分类器 `naive_bayes.GaussianNB` 等价。

1.2.3. LDA维度缩减的数学形式

为了理解LDA在维度缩减中的应用，这里有必要从几何的角度上重新回顾一下上面说明的LDA分类原理。记K为所有目标类别的总个数。由于在LDA中假设所有类别都有相同的估计协方差$\sum$，因此可以对数据进行重新缩放，使得协方差矩阵是单位矩阵：

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

由此可得，在数据缩放以后，若要对某个数据点进行分类，则等价于找到与该点欧几里得距离最近的类别均值μ_k^*。但是这个分类过程也可以在把原数据点投影到所有类别的所有μ_k^*生成的$K-1$维仿射子空间后得到。这说明LDA分类器隐含着一个思想，就是通过把数据线性投影到$K-1$维空间进行维度缩减。

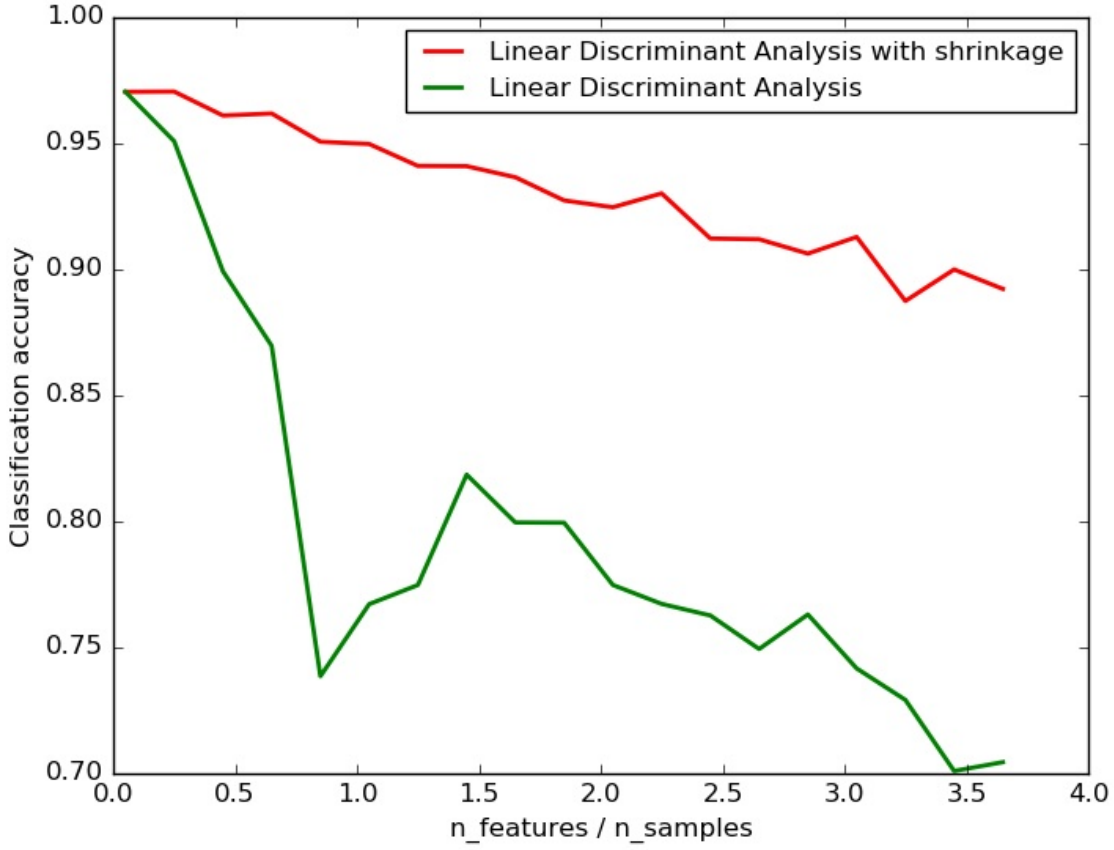
甚至我们可以设定投影过后数据的新维度L，使缩减量不止为1维。即将数据投影到L维线性子空间H_L，该子空间将投影后得到的μ_k^*的方差最大化（实际上可以看作是在为转换后的类别均值μ_k^*做主成分分析）。这个L对应于 `discriminant_analysis.LinearDiscriminantAnalysis.transform` 方法中用到的 `n_components` 参数。更详细的解释参看[3]。

1.2.4. 收缩

在训练样本数小于特征总数时，可以借助“收缩”这一工具来提高协方差矩阵估计的准确度。在这种情况下，把按照经验得到的样本协方差作为估计量效果会很差。
把 `discriminant_analysis.LinearDiscriminantAnalysis` 类的 `shrinkage` 参数设为“auto”就可以使用收缩LDA。这样，根据Ledoit和Wolf提出的引理[4]，sklearn可以通过分析的方式自动确定最优的收缩参数。需要注意的是，只有把 `solver` 参数设为“lsqr”或者“eigen”才能让当前的收缩起作用。

也可以手动把参数 `shrinkage` 设为一个0到1中间的值。特别的，当该值为0时，不进行收缩（即使用经验得到的协方差矩阵）；该值为1时，使用完全收缩（即使用对角的方差矩阵作为协方差矩阵的估计值）。不走极端则会得到协方差矩阵的收缩估计版本。

.inear Discriminant Analysis vs. shrinkage Linear Discriminant Analysis (1 discriminative feature



1.2.5. 估计算法

默认的求解器为“svd”，在分类和转换问题上都可用，而且不依赖协方差矩阵的计算。当特征数量比较大时，该方法表现不错。但是它不能和收缩一起使用。

“lsqr”求解器是一种高效的算法，只能用于回归，支持收缩。

“eigen”求解器基于optimization of the between class scatter to within class scatter ratio，在分类和转换问题上都可用。不过该求解器需要计算协方差矩阵，所以在特征数目比较多时不太适合。

例子

Normal and Shrinkage Linear Discriminant Analysis for classification 比较了使用和不用收缩的LDA分类器

参考文献

[3] “The Elements of Statistical Learning”, Hastie T., Tibshirani R., Friedman J., Section 4.3, p.106-119, 2008.

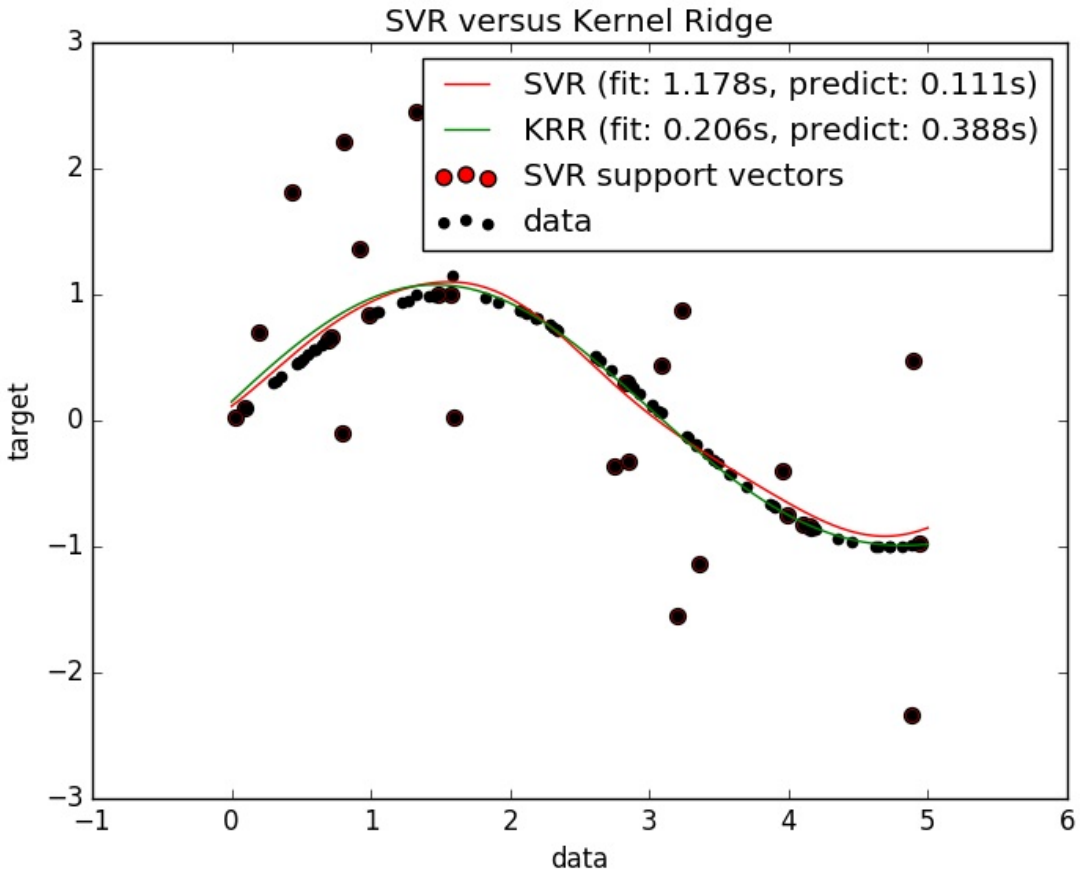
[4] Ledoit O, Wolf M. Honey, I Shrunk the Sample Covariance Matrix. The Journal of Portfolio Management 30(4), 110-119, 2004.

1.3. 核岭回归

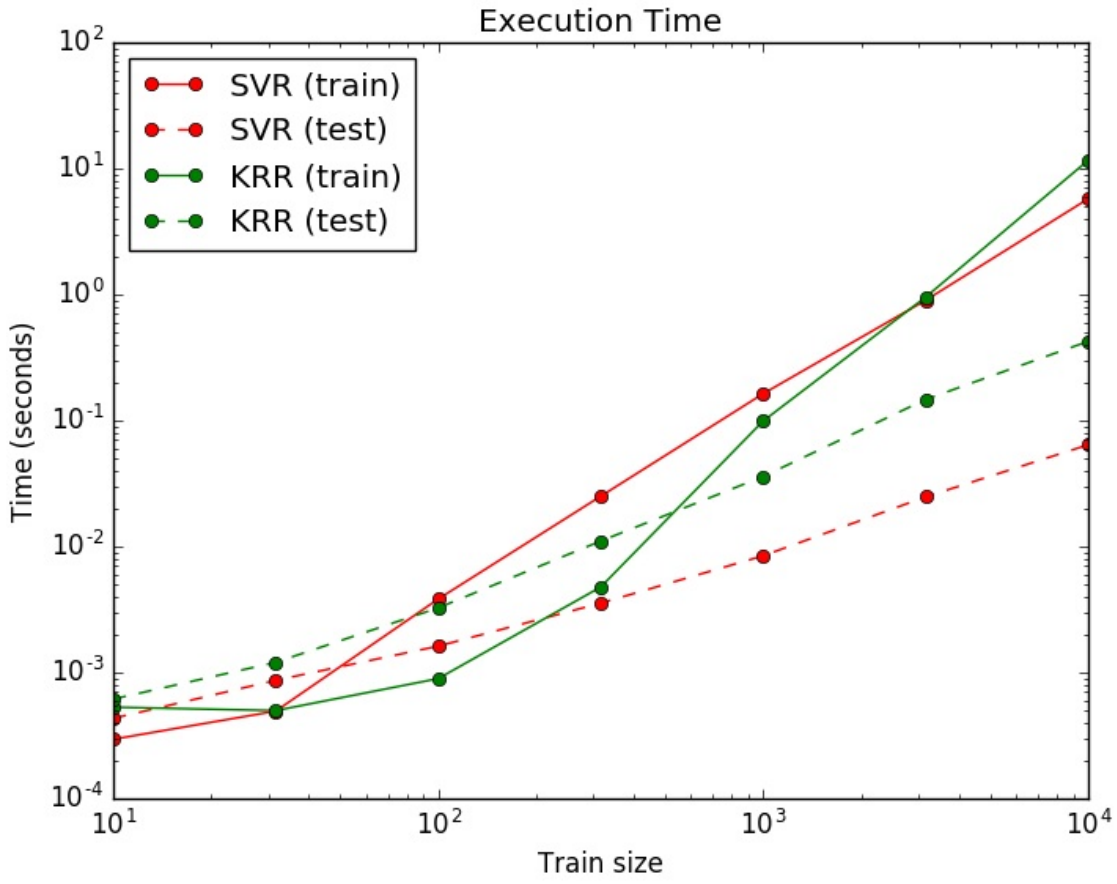
核岭回归（KRR）[M2012]将岭回归（带有ℓ_2范数正则化的线性最小二乘）和核组合了起来。因此它是在由对应核和数据所导出的空间中学习线性函数。对于非线性核，则对应于原始空间中的一个非线性函数。

`KernelRidge` 学到的模型与支持向量回归（SVR）学到的模型有着相同的形式。然而两者用到的损失函数有所不同。KRR用的是平方误差损失函数，而SVR用到的是ϵ-不敏感损失函数——当然，两者都结合了ℓ_2正则化。与SVR不同，`KernelRidge` 可以使用闭合形式拟合，而且在中等大小的数据集上通常拟合更快。另一方面，`KernelRidge` 学习到的模型是非稀疏的，因此在预测时会比SVR慢，因为后者在预测是对$\epsilon > 0$学习一个稀疏模型。

下图在一个人工构造的数据集上对 `KernelRidge` 和SVR进行了对比。其中，数据集通过如下方法构造：总的目标函数是一个正弦函数，而对每五个数据点会增加一个很强的噪声。图中给出了学习到的 `KernelRidge` 模型和SVR模型，两个模型的复杂度/正则度和RBF核函数的带宽都通过网格搜索进行了优化。两者学习到的曲线非常接近，然而拟合 `KernelRidge` 模型所花的时间大约只有拟合SVR模型所花时间的七分之一。但是，在对十万个目标值进行预测的时间这个指标上，SVR只用了 `KernelRidge` 所花时间的三分之一。原因是SVR学习到的是一个稀疏模型，只把100个训练点中大约三分之一的点作为支持向量。



下图比较了在不同大小训练集上拟合 `KernelRidge` 和SVR并使之预测所花的时间。在中等大小训练集（少于1000条数据）上 `KernelRidge` 的拟合要比SVR的快；然而在大数据集上SVR有更强的规模可扩展性。在预测时间这一方面，SVR总是比 `KernelRidge` 快，因为SVR学到的是稀疏模型。注意SVR的稀疏程度（以及由此产生的预测时间）依赖于参数ϵ和C；$\epsilon = 0$则学到的模型是密集模型。



参考文献

[M2012] *Machine Learning: A Probabilistic Perspective*. Murphy, K. P. - chapter 14.4.3, pp. 492-493, The MIT Press, 2012

1.4. 支持向量机

支持向量机（**SVM**）是一系列可用于分类、回归和异常值检测的有监督学习方法。

支持向量机的优点包括：

- 在高维空间中行之有效。
- 当维数大于样本数时仍然可用。
- 在决策函数中只使用训练点的一个子集（称为支持向量），大大节省了内存开销。
- 用途广泛：决策函数中可以使用不同的核函数。提供了一种通用的核，但是也可以指定自定义的核。

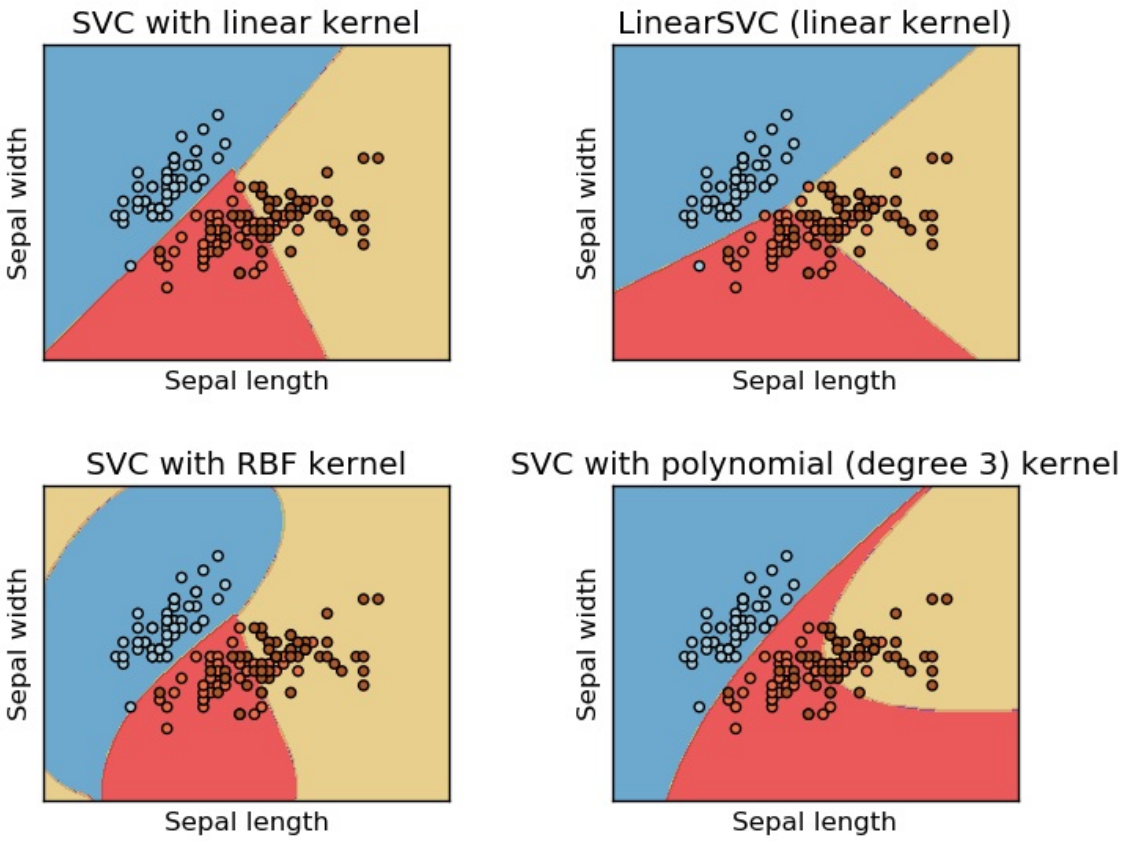
而其劣势在于

- 如果特征数量远大于样本数量，则表现会比较差。
- SVM不直接提供概率估计。这个值通过五折交叉验证计算，代价比较高（见下面"跑分与概率"一节）。

Scikit-learn中的支持向量机同时支持密集样本向量（`numpy.ndarray`和可通过`numpy.asarray`转化的数据类型）和稀疏样本向量（任何`scipy.sparse`对象）。但是如果想用SVM对稀疏数据进行预测，则必须先在这些数据上拟合。为了优化性能，应该使用C阶（C-Ordered）`numpy.ndarray`（密集的）或`scipy.sparse.csr_matrix`（稀疏的），并指定`dtype=float64`。

1.4.1. 分类

要在数据集上进行多类别分类，可以使用`SVC`，`NuSVC`和`LinearSVC`这三个类。



`SVC` 和 `NuSVC` 两种方法类似，但是接受的参数有细微不同，而且底层的数学原理不一样（见“数学原理”一节）。另一方面，`LinearSVC` 是对支持向量分类的另一种实现，使用了线性核。注意 `LinearSVC` 不接受关键字 `kernel` ，因为核被预设为是线性的。其与 `SVC` 和 `NuSVC` 相比还缺少了一些成员，如 `support_` 。

和其它分类器一样， `SVC` ， `NuSVC` 和 `LinearSVC` 接受两个数组：大小为 `[n_samples, n_features]` 的数组X，包含训练样本；以及大小为 `[n_samples]` 的数组y，包含类别标签（以字符串类型或整型存储）：

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

拟合以后就可以用得到的模型预测新值

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

SVM的决策函数依赖于训练数据的一个子集，称为支持向量。它们的一些属性可以在成员变量 `support_vectors_` ， `support_` 和 `n_support` 中找到

```
>>> # get support vectors
>>> clf.support_vectors_
array([[ 0.,  0.],
       [ 1.,  1.]])
>>> # get indices of support vectors
>>> clf.support_
array([0, 1]...)
>>> # get number of support vectors for each class
>>> clf.n_support_
array([1, 1]...)
```

1.4.1.1. 多类别分类

`SVC` 和 `NuSVC` 使用“一对多”方法（Knerr et al., 1990）来实现多类别分类。如果 `n_class` 是类别的数目，则该实现会构造 `n_class * (n_class - 1) / 2` 个分类器，每个分类器针对两个类别对数据进行训练。为了提供一个和其它分类器一致的接口，选项 `decision_function_shape` 允许调用者将所有“一对一”分类器的结果聚合进一个 `(n_samples, n_classes)` 的决策函数

```
>>> X = [[0], [1], [2], [3]]
>>> Y = [0, 1, 2, 3]
>>> clf = svm.SVC(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='auto', 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` 实现了“一对多”分类法，因此会训练n_class个模型。如果只有两个类别，那么只会得到一个模型：

```
>>> 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
```

决策函数的完整描述可见“数学原理”一节。

需要注意的是，`LinearSVC` 还实现了另一种分类策略，即所谓的多类别SVM（由Crammer和Singer提出）。通过指定 `multi_class='crammer_singer'` ，可以使用该方法。它得到的结果总是一致的，但是一对多分类法并不能保证这一点。然而在实际应用中，还是一对多分类法用的更多，因为通常来讲该方法得到的结果变化不会太大（mostly similar），而且用的时间显著的短。

一对多 `LinearSVC` 得到的属性中， `coef_` 是 `n_class * n_features` 维矩阵，而 `intercept_` 是 `n_class * 1` 维的。系数中的每一行都对应于n_class个“一对多”分类器中的一个（截距也是类似），按照每个与其它类别进行比较的类别顺序排序（in the order of the "one" class）。

而在“一对一” `SVC` 中，属性的格式略微有些复杂难懂。在有线性核的情况下， `coef_` 和 `intercept_` 的格式与上面所述 `LinearSVC` 的类似，只不过 `coef` 的大小为 `[n_class * (n_class - 1) / 2, n_features]` （对应于二元分类器的个数）。类别0到n的顺序是“0 vs 1”, “0 vs 2” , ... “0 vs n”, “1 vs 2”, “1 vs 3”, “1 vs n”, . . . “n-1 vs n”。

`dual_coef` 的大小是[n_class-1, n_SV]，不过格式有些难以描述。列对应于 `n_class * (n_class - 1) / 2` 个“一对一”分类器中出现的支持向量。每个支持向量被用于 `n_class - 1` 个分类器中。每一行中 `n_class - 1` 个项目对应于这些分类器的对偶系数（dual coefficient）。

举个例子可能能说清楚一点：

考虑一个三元分类问题。类别0的支持向量为v_0^0, v_0^1, v_0^2。类别1和2有两个支持向量，分别为v_1^0, v_1^1和v_2^0, v_2^1。对每个支持向量v_i^j，有两个对偶系数。称类别i和k之间分类器支持向量v_i^j的系数为$\alpha_{i,k}^j$，则 `dual_coef_` 如下表所示：

[表格暂缺]

1.4.1.2. 得分与概率

`SVC` 中的 `decision_function` 方法对每个样本都会给出在各个类别上的分数（在二元分类问题中，是对每个样本给出一个分数）。如果构造函数的 `probability` 被设为 `True` ，则可以得到属于每个类别的概率估计（通过 `predict_proba` 和 `predict_log_proba` 方法）。在二元分类中，概率使用Platt缩放进行调整：通过在训练机上做额外的交叉检验来拟合一个在SVM分数上的Logistic回归。在多元分类中，这种方法被Wu et al. (2004)扩展了。

显而易见的是，Platt缩放中的交叉检验在大数据集上是一个代价很高的操作。此外，概率估计与实际得分可能会不一致，即使得分取得了最大值，概率并不一定也能取到最大值。（例如在二元分类中，某个样本经由 `predict` 方法得到的分类标签，如果使用 `predict_proba` 计算可能概率小于1/2。）Platt的方法在理论上也有一些问题。如果需要拿到置信分数，而这些分数又不一定非得是概率，则建议把 `probability` 置为 `False` ，并且使用 `decision_function` ，而不是 `predict_proba` 。

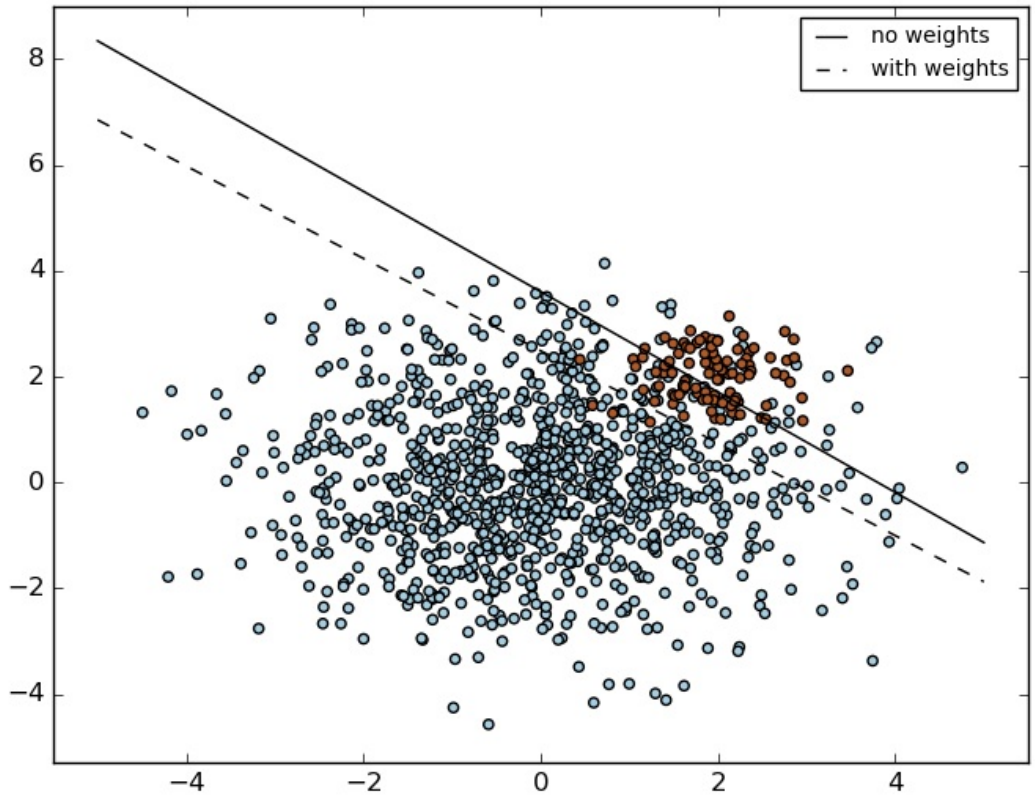
参考文献

- Wu, Lin and Weng, “*Probability estimates for multi-class classification by pairwise coupling*”. JMLR 5:975-1005, 2004.

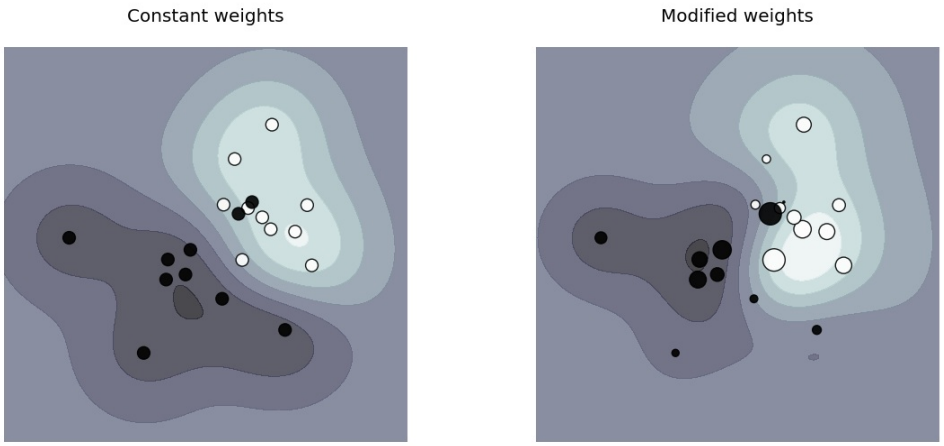
1.4.1.3. 不平衡问题

在某些情况下，一些指定的类别或某几个样本关键字可能更加重要，这时可以使用 `class_weight` 和 `sample_weight` 。

`SVC` （不是 `NuSVC` ）在 `fit` 方法中实现了关键字 `class_weight` 。该关键字是字典类型，形式为 `{class_label : value}` ，这里 `value` 是一个正浮点数，将类别 `class_label` 的参数 `C` 设为 `C * value` 。



`SVC`，`NuSVC`，`SVR`，`NuSVR`和`OneClassSVM`同样在`fit`方法中通过关键字`sample_weight`实现了对单独样本赋予特殊权重的功能。与`class_weight`类似，它们把第i个样本的参数`C`设为`C * sample_weight[i]`。



例子

- 在iris数据集中试验不同的SVM分类器，并作图比较
- SVM：最大化间隔分离超平面
- SVM：不平衡类别的分离超平面
- SVM-Anova：带有单变量特征选择的SVM

- 非线性SVM
- SVM：带权重问题的例子

1.4.2. 回归

支持向量分类这样的方法可以经扩展用在回归问题上，称作支持向量回归。

支持向量分类产生的模型，如上所述，只依赖于训练数据的一个子集。其原因在于，构造模型时用到的代价函数并不关心那些不在边界上的数据点。类似的，支持向量回归所生成的模型也只依赖于训练数据的一个自己，因为构造模型时用到的代价函数用不上那些与预测值很接近的训练数据。

支持向量回归有三种不同实现：`SVR`，`NuSVR`和`LinearSVR`。`LinearSVR`提供的实现比`SVR`快，但是只使用线性核。`NuSVR`则是使用了一个略不同的数学原理。细节见后面“实现细节”部分。

与分类问题类似，`fit`方法接受向量X和y作为参数，不过这里y应该是浮点型而不是整型：

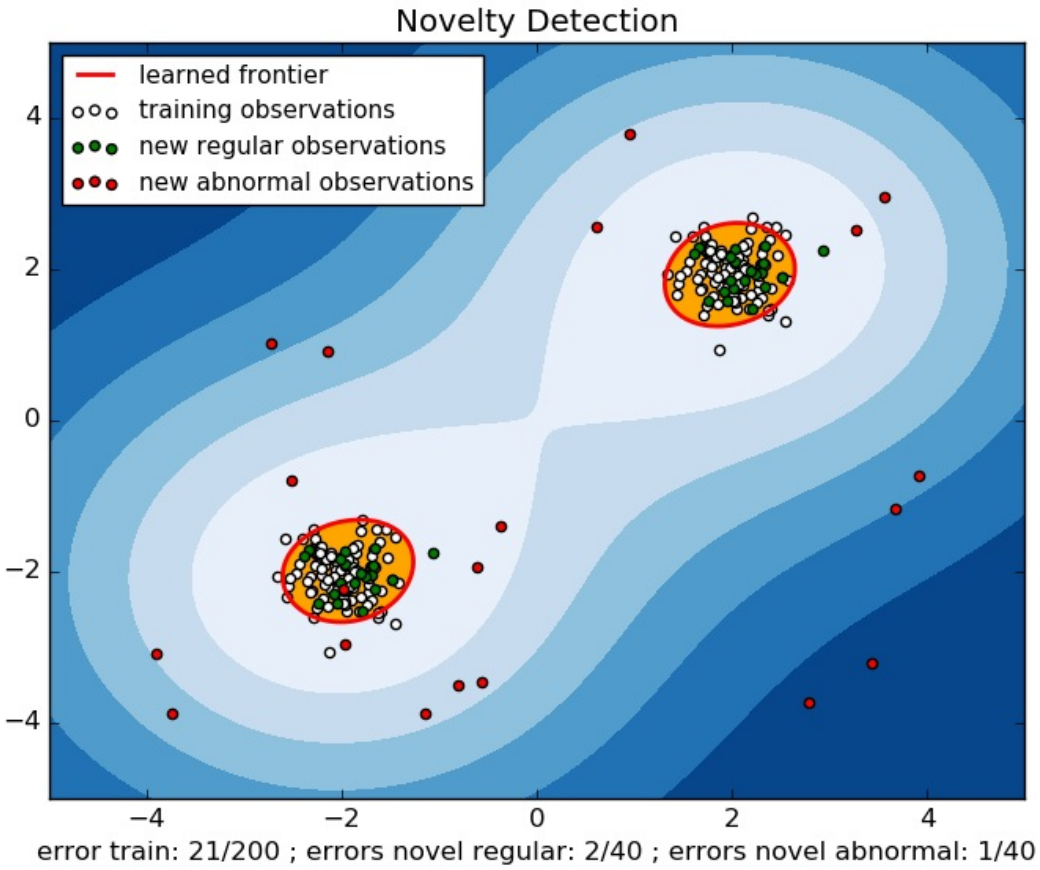
```
>>> from sklearn import svm
>>> X = [[0, 0], [2, 2]]
>>> y = [0.5, 2.5]
>>> clf = svm.SVR()
>>> clf.fit(X, y)
SVR(C=1.0, cache_size=200, coef0=0.0, degree=3, epsilon=0.1, gamma='auto',
    kernel='rbf', max_iter=-1, shrinking=True, tol=0.001, verbose=False)
>>> clf.predict([[1, 1]])
array([ 1.5])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

例子

- 使用线性核和非线性核进行支持向量回归（SVR）

1.4.3. 密度估计与新奇值检测（novelty detection）

单类别SVM可用于新奇值检测，即给定一组样本，检测出该数据集的软边界，以判断新数据点是否属于该数据集。`OneClassSVM`类实现该方法。在这种情况下，由于这是一种无监督学习，数据没有类别标签，因此`fit`方法只接受数组X作为输入。见新奇值与异常值检测检测一节了解更多用法。



例子

- 使用非线性核（RBF）的单类别SVM
- 种类分布建模

1.4.4. 复杂度

支持向量机是一种能力很强的工具，但是随着支持向量数目的增加，它们对计算和存储资源的需求也会快速增长。SVM的核心是二次规划问题（QP），将支持向量与其他训练数据分开。取决于libsvm缓存在实践中使用的效率（这又是依赖于具体的数据集），基于libsvm的实现所用的QP求解器时间复杂度在$O(n_{\text{features}} \times n_{\text{samples}}^2)$到$O(n_{\text{features}} \times n_{\text{samples}}^3)$不等。如果数据非常稀疏，$n_{\text{features}}$应该被样本向量中非零特征的平均数取代。

还要注意的是，对线性的情况，基于liblinear实现的`LinearSVC`的算法比基于libsvm实现的`SVC`效率要高很多，而且前者在处理以百万计的样本和/或特征时仅以线性增长。

1.4.5. 应用建议

- 避免数据拷贝：对`SVC`、`SVR`、`NuSVC`和`NuSVR`，如果传给特定方法的数据不是以C语言所使用的顺序排列，而且是double精度，那么该数据会在调用底层C实现之前被拷贝一份。通过检查`flag`属性，可以检查给定的numpy数组是否是以C格式的连续存储方式排

列的。对 `LinearSVC` 和 `LogisticRegression` ，任何以numpy数组形式传入的输入都会被拷贝，然后转化为liblinear内部的稀疏数据表示形式（双精度浮点数，对非零元素存储32位整型的索引）。如果你想训练一个大规模的线性分类器，而又不想拷贝一个稠密的numpy C-存储双精度数组，我们建议使用 `SGDClassifier` 。可以对它的目标函数进行配置，使其与 `LinearSVC` 模型所使用的基本相同。

- 核缓存大小：对 `SVC` 、 `SVR` 、 `NuSVC` 和 `NuSVR` ，核缓存的大小对较大问题求解的运行时间有非常强的影响。如果你有足够内存，建议将 `cache_size` 设置为一个高于默认值200（MB）的值，比如500（MB）或1000（MB）。
- 设置**C**：默认情况下 `C` 设为1，这是一个合理的选择。如果样本中有许多噪音观察点，则应该减小这个值。这意味着对估计结果进行更严格的正则化。
- SVM算法会受数据取值范围的影响，所以强烈建议在使用之前对数据进行缩放。例如把输入向量X的每个属性缩放到[0,1]或[-1,+1]内，或者进行标准化使数据的均值为0方差为1。注意在测试向量上也要进行同样的缩放，这样才能得到有意义的结果。关于数据缩放和标准化的更多细节，参见处理数据一节
- `NuSVC` / `OneClassVM` / `NuSVR` 中的参数 `mu` 估计了训练误差和支持向量的比率
- 在 `SVC` 中，如果要分类的数据是不平衡的（如有很多正数据但是很少负数据），应该加选项 `class_weight='balanced'` 然后/或者尝试不同的惩罚项参数 `C` 。
- `LinearSVC` 的底层实现使用了随机数生成器来在拟合模型时选择特征。因此对同样的输入数据有略微不同的结果不是怪事。如果发生了这样的情况，试一个更小的tol参数
- 利用 `LinearSVC(loss='l2', penalty='l1', dual=False)` 来引入L1惩罚项会产生一个稀疏解，即特征权重中只有一少部分不为0，会对决策函数产生贡献。增加 `C` 会产生一个更复杂的模型（有更多特征被选择）。可以通过 `l1_min_c` 来计算产生“空”模型（所有权重都是0）的 `C` 值。

1.4.6. 核函数

核函数可以有以下几种选择：

- 线性核$\langle x, x' \rangle$
- 多项式核$(\gamma \langle x, x' \rangle + r)^d$。其中$d$由选项 `degree` 指定，$r$由 `coef0` 指定
- 径向基函数（RBF）：$\exp(-\gamma |x - x'|^2)$。其中γ由选项 `gamma` 指定，必须大于0
- sigmoid函数（$\tanh(\gamma \langle x, x' \rangle + r)$），其中$r$由选项 `coef0` 指定

在初始化时通过选项 `kernel` 指定用什么核

```
>>> 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

Scikit提供两种方法来自定义核函数：给参数 `kernel` 传入一个python函数，或者提前计算好Gram矩阵。使用自定义核的分类器和其它分类器有类似的行为，不过以下两点除外：

- `support_vectors_` 域为空，只在 `support_` 里面存储支持向量的索引
- 会为 `fit()` 方法的第一个参数存储一个引用（不是拷贝），来为以后引用之做准备。如果在调用 `fit()` 之后，在调用 `predict()` 之前修改这个数组，则会产生一些不可预知的结果。

1.4.6.1.1. 使用Python函数作为核

可以在构造函数中向参数 `kernel` 传进一个函数，来使用自定义的核。该函数必须接受两个大小分别为 `(n_samples_1, n_features)`，`(n_samples_2, n_features)` 的矩阵作为参数，返回一个大小为 `(n_samples_1, n_samples_2)` 的核矩阵。

如下代码定义了一个线性核，并使用该核创建了一个分类器实例

```
>>> 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)
```

例子

- 使用自定义核的SVM

1.4.6.1.2. 使用Gram矩阵

将参数 `kernel` 设为 `precomputed`，就可以把Gram矩阵（而不是X）传给 `fit` 方法。若如此做，需要提供所有训练向量和测试向量之间的核的值。

```
>>> import numpy as np
>>> from sklearn import svm
>>> X = np.array([[0, 0], [1, 1]])
>>> y = [0, 1]
>>> clf = svm.SVC(kernel='precomputed')
>>> # linear kernel computation
>>> 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=None, degree=3, gamma='auto',
    kernel='precomputed', max_iter=-1, probability=False,
    random_state=None, shrinking=True, tol=0.001, verbose=False)
>>> # predict on training examples
>>> clf.predict(gram)
array([0, 1])
```

1.4.6.1.3. RBF核的参数

使用径向基函数（RBF）核训练SVM时，需要考虑 `C` 和 `gamma` 这两个参数。参数 `C` 会被所有SVM核用到，用来在样本误分类和决策平面的简单性之间做出权衡。 `C` 值如果小，得到的决策平面会更光滑；而大的 `C` 则试图将所有训练样本都进行正确的分类。 `gamma` 定义单个训练样本能有多少影响。 `gamma` 越大，与之更近的样本受到的影响越大。

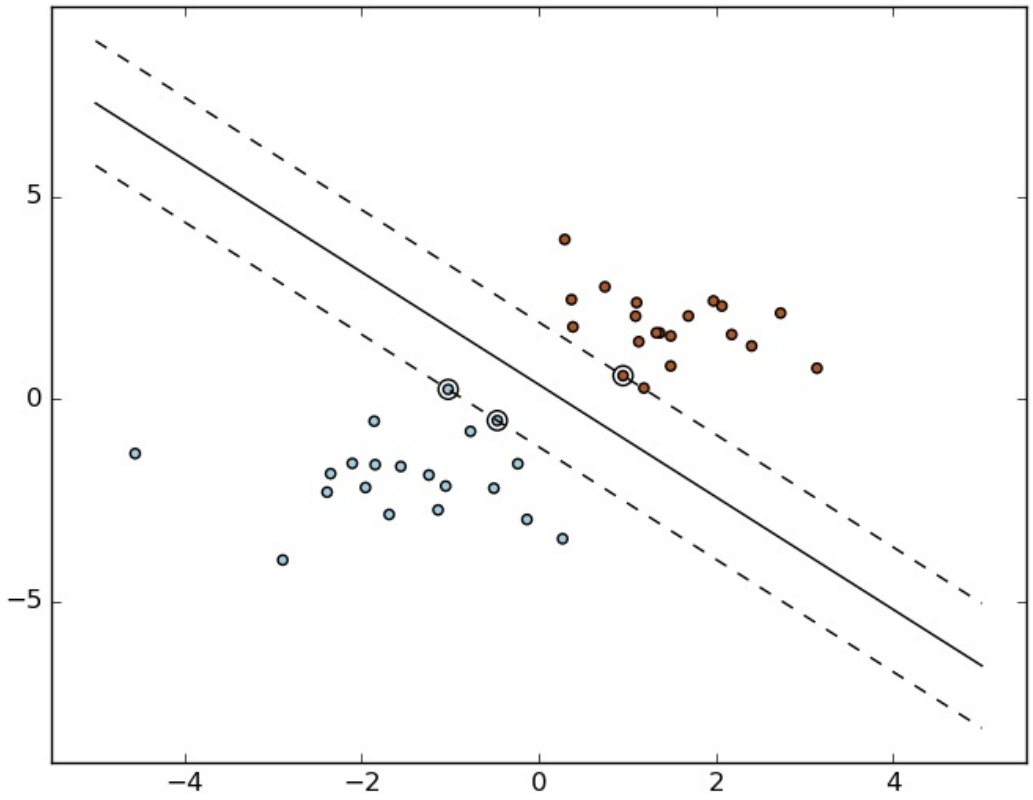
对这两个值的选择会极大影响SVM的性能。建议使用 `sklearn.grid_search.GridSearchCV` 来在 `C` 和 `gamma` 广阔的指数空间里进行选择，得到最合适的值。

例子

- RBF SVM参数

1.4.7. 数学表示

支持向量机的原理是在高维甚至无限维空间中构建一个超平面或者若干超平面组成的集合，并藉此用于分类、回归或其它任务。从直觉上来讲，好的分隔面是由使得函数间隔最大的超平面得到（即该超平面到任何类别最近训练数据点的距离都取得最大值），因为总体上来讲，间隔越大，分类器的泛化误差越小。



1.4.7.1. SVC

给定可能属于某两种类别的训练向量$x_i \in \mathbb{R}^p, i = 1, \ldots, n$和向量$y \in \{1, -1\}^n$，SVC解决的首要问题是

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

其对偶形式为

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

其中e是全1向量，$C > 0$是上界，Q是$n \times n$半正定矩阵，$Q_i j \equiv y_i y_j K(x_i, x_j)$，其中$K(x_i, x_j) = \phi(x_i)^T \phi(x_j)$是核。这里训练向量通过函数$\phi$被隐式映射到一个高维（甚至无限维）空间。

决策函数是

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

注意：虽然从 `libsvm` 和 `liblinear` 导出的SVM模型使用 `C` 作为正则化参数，但实际上其它更多预测器使用的是 `alpha` 。这两个参数之间的关系是 $C = \frac{n_{\text{samples}}}{alpha}$

这些参数可以通过各个成员变量访问： `dual_coef_` 存放乘积 $y_i \alpha_i$ ， `support_vectors` 存放支持向量， `intercept_` 存放独立项 ρ ：

参考文献：

- *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 Leaming, 20, 273-297 (1995)

1.4.7.2. NuSVC

我们提供了一个新的参数 ν ，控制支持向量的个数和训练误差。参数 $\nu \in (0, 1]$ 是训练误差的上限和支持向量的下限。

可以看到 ν -SVC是 C -SVC的一种重参数化形式，因此两者在数学上是等价的。

1.4.7.3. SVR

给定训练向量组 $x_i \in \mathbb{R}^p,\ i = 1, \ldots, n$ 和向量 $y \in \mathbb{R}^n$, ε -SVR要解决的主要问题如下所示：

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

其对偶为

$$\begin{aligned} \min_{\alpha,\alpha^*} \frac{1}{2} (\alpha - \alpha^*)^T Q (\alpha - \alpha^*) + \varepsilon e^T (\alpha + \alpha^*) - y^T (\alpha - \alpha^*) \\ \text{subject to } e^T (\alpha - \alpha^*) = 0 \\ 0 \leq \alpha_i, \alpha_i^* \leq C, i = 1, \ldots, n \end{aligned}$$

其中e是全1向量，$C > 0$是上界。Q是$n \times n$的半正定矩阵，$Q_{ij} \equiv K(x_i, x_j) = \phi(x_i)^T \phi(x_j)$是核。这里训练向量通过函数$\phi$被隐式映射到一个高维（甚至无限维）空间。

决策函数为

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

这些参数可以通过各个成员变量访问。其中变量`dual_coef_`存储$\alpha_i - \alpha_i^*$，`support_vectors_`存储支持向量，`intercept_`存储独立项ρ

参考文献

- “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，我们使用libsvm和liblinear来处理所有计算逻辑。这些库是被C和python包装的。

参考文献：

要了解所用算法的实现细节，可以参考

- LIBSVM: a library for Support Vector Machines
- LIBLINEAR – A Library for Large Linear Classification

1.5. 随机梯度下降

随机梯度下降（Stochastic Gradient Descent, SGD）是一个