**预处理数据**

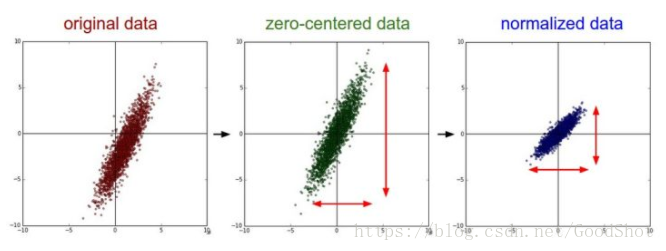
sklearn.preprocessing 包提供了几个常见的实用功能和变换器类型，用来将原始特征向量更改为更适合机器学习模型的形式。

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](http://sklearn.apachecn.org/cn/0.19.0/auto_examples/preprocessing/plot_all_scaling.html#sphx-glr-auto-examples-preprocessing-plot-all-scaling-py) 中有着重说明。

在一些实际问题中，我们得到的样本数据都是多个维度的，即一个样本是用多个特征来表征的。比如在预测房价的问题中，影响房价的因素有房子面积、卧室数量等，我们得到的样本数据就是这样一些样本点，这里的、又被称为特征。很显然，这些特征的量纲和数值得量级都是不一样的，在预测房价时，如果直接使用原始的数据值，那么他们对房价的影响程度将是不一样的，而通过标准化处理，可以使得不同的特征具有相同的尺度（Scale）。简言之，当原始数据不同维度上的特征的尺度（单位）不一致时，需要标准化步骤对数据进行预处理。

下图中以二维数据为例：左图表示的是原始数据；中间的是中心化后的数据，数据被移动大原点周围；右图将中心化后的数据除以标准差，得到为标准化的数据，可以看出每个维度上的尺度是一致的（红色线段的长度表示尺度）。

其实，在不同的问题中，中心化和标准化有着不同的意义，



比如在训练神经网络的过程中，通过将数据标准化，能够加速权重参数的收敛。

对数据进行中心化预处理，这样做的目的是要增加基向量的正交性。

中心化（又叫零均值化）和标准化（又叫归一化）概念及目的？

在回归问题和一些机器学习算法中，以及训练神经网络的过程中，通常需要对原始数据进行中心化（Zero-centered或者Mean-subtraction（subtraction表示减去））处理和标准化（Standardization或Normalization）处理

数据标准化（归一化）处理是数据挖掘的一项基础工作，不同评价指标往往具有不同的量纲和量纲单位，这样的情况会影响到数据分析的结果，为了消除指标之间的量纲影响，需要进行数据标准化处理，以解决数据指标之间的可比性。原始数据经过数据标准化处理后，各指标处于同一数量级，适合进行综合对比评价。

意义：数据中心化和标准化在回归分析中是取消由于量纲不同、自身变异或者数值相差较大所引起的误差。

原理：数据标准化：是指数值减去均值，再除以标准差；

数据中心化：是指变量减去它的均值。

目的：通过中心化和标准化处理，得到均值为0，标准差为1的服从标准正态分布的数据。

（1）中心化（零均值化）后的数据均值为零

（2）z-score 标准化后的数据均值为0，标准差为1（方差也为1）

**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:

>>> from sklearn import preprocessing

>>> import numpy as np

>>> X\_train = np.array([[ 1., -1., 2.],

... [ 2., 0., 0.],

... [ 0., 1., -1.]])

>>> X\_scaled = preprocessing.scale(X\_train)

>>> X\_scaled

array([[ 0. ..., -1.22..., 1.33...],

[ 1.22..., 0. ..., -0.26...],

[-1.22..., 1.22..., -1.06...]])

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。

**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0和1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 。

**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](http://sklearn.apachecn.org/cn/0.19.0/modules/generated/sklearn.preprocessing.MaxAbsScaler.html#sklearn.preprocessing.MaxAbsScaler) 以及 [maxabs\_scale](http://sklearn.apachecn.org/cn/0.19.0/modules/generated/sklearn.preprocessing.maxabs_scale.html#sklearn.preprocessing.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](http://sklearn.apachecn.org/cn/0.19.0/modules/generated/sklearn.preprocessing.scale.html#sklearn.preprocessing.scale) 和 [StandardScaler](http://sklearn.apachecn.org/cn/0.19.0/modules/generated/sklearn.preprocessing.StandardScaler.html#sklearn.preprocessing.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](http://sklearn.apachecn.org/cn/0.19.0/modules/generated/sklearn.preprocessing.RobustScaler.html#sklearn.preprocessing.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，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。

**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](http://sklearn.apachecn.org/cn/0.19.0/modules/generated/sklearn.preprocessing.robust_scale.html#sklearn.preprocessing.robust_scale) 以及 [RobustScaler](http://sklearn.apachecn.org/cn/0.19.0/modules/generated/sklearn.preprocessing.RobustScaler.html#sklearn.preprocessing.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:

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?](http://www.faqs.org/faqs/ai-faq/neural-nets/part2/section-16.html)

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](http://sklearn.apachecn.org/cn/0.19.0/modules/generated/sklearn.decomposition.PCA.html#sklearn.decomposition.PCA) 或 [sklearn.decomposition.RandomizedPCA](http://sklearn.apachecn.org/cn/0.19.0/modules/generated/sklearn.decomposition.RandomizedPCA.html#sklearn.decomposition.RandomizedPCA) 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

[scale](http://sklearn.apachecn.org/cn/0.19.0/modules/generated/sklearn.preprocessing.scale.html#sklearn.preprocessing.scale) 以及 [StandardScaler](http://sklearn.apachecn.org/cn/0.19.0/modules/generated/sklearn.preprocessing.StandardScaler.html#sklearn.preprocessing.StandardScaler) 可以直接处理一维数组。在回归中，缩放目标/相应变量时非常有用。

**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](http://sklearn.apachecn.org/cn/0.19.0/modules/generated/sklearn.preprocessing.KernelCenterer.html#sklearn.preprocessing.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。

**2. 非线性转换**

类似于缩放， [QuantileTransformer](http://sklearn.apachecn.org/cn/0.19.0/modules/generated/sklearn.preprocessing.QuantileTransformer.html#sklearn.preprocessing.QuantileTransformer) 类将每个特征缩放在同样的范围或分布情况下。但是，通过执行一个秩转换能够使异常的分布平滑化，并且能够比缩放更少地受到离群值的影响。但是它的确使特征间及特征内的关联和距离失真了。

[QuantileTransformer](http://sklearn.apachecn.org/cn/0.19.0/modules/generated/sklearn.preprocessing.QuantileTransformer.html#sklearn.preprocessing.QuantileTransformer) 类以及 [quantile\_transform](http://sklearn.apachecn.org/cn/0.19.0/modules/generated/sklearn.preprocessing.quantile_transform.html#sklearn.preprocessing.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...])

也可以通过设置 output\_distribution='normal' 将转换后的数据映射到正态分布:

>>> 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...],

[ 4.31..., 2.02..., 1.01..., 0.1...],

[ 4.32..., 2.05..., 1.02..., 0.1...],

...,

[ 7.84..., 4.34..., 6.84..., 2.5...],

[ 7.87..., 4.37..., 6.87..., 2.5...],

[ 7.9..., 4.4..., 6.9..., 2.5...]])

这样，输入的中值称为输出的平均值，并且以0为中心。正常输出被剪切，使得输入的最小和最大值分别对应于1e-7和1-1e-7分位数——在变换下不会变得无限大。

**3. 归一化**

两个优点：

1）归一化后加快了梯度下降求最优解的速度；

2）归一化有可能提高精度。

1、 归一化为什么能提高梯度下降法求解最优解的速度？

如果机器学习模型使用梯度下降法求最优解时，归一化往往非常有必要，否则很难收敛甚至不能收敛。

2、归一化有可能提高精度

一些分类器需要计算样本之间的距离（如欧氏距离），例如KNN。如果一个特征值域范围非常大，那么距离计算就主要取决于这个特征，从而与实际情况相悖（比如这时实际情况是值域范围小的特征更重要）。

3、以下是两种常用的归一化方法：

1）min-max标准化（Min-MaxNormalization）

也称为离差标准化，是对原始数据的线性变换，使结果值映射到[0 - 1]之间。转换函数如下：

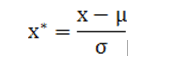
http://images2015.cnblogs.com/blog/323808/201603/323808-20160307152657304-486322487.png

其中max为样本数据的最大值，min为样本数据的最小值。这种方法有个缺陷就是当有新数据加入时，可能导致max和min的变化，需要重新定义。

2）Z-score标准化（0-1标准化）方法

这种方法给予原始数据的均值（mean）和标准差（standard deviation）进行数据的标准化。经过处理的数据符合标准正态分布，即均值为0，标准差为1。

转化函数为：



其中为所有样本数据的均值，为所有样本数据的标准差。

**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)](https://en.wikipedia.org/wiki/Vector_Space_Model) ，经常在文本分类和内容聚类中使用.

函数 [normalize](http://sklearn.apachecn.org/cn/0.19.0/modules/generated/sklearn.preprocessing.normalize.html#sklearn.preprocessing.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](http://sklearn.apachecn.org/cn/0.19.0/modules/generated/sklearn.preprocessing.Normalizer.html#sklearn.preprocessing.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](http://sklearn.apachecn.org/cn/0.19.0/modules/generated/sklearn.pipeline.Pipeline.html#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](http://sklearn.apachecn.org/cn/0.19.0/modules/generated/sklearn.preprocessing.normalize.html#sklearn.preprocessing.normalize) 以及类 [Normalizer](http://sklearn.apachecn.org/cn/0.19.0/modules/generated/sklearn.preprocessing.Normalizer.html#sklearn.preprocessing.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。

**4. 二值化**

**4.1. 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)](https://en.wikipedia.org/wiki/Bernoulli_distribution) 。例如这个例子 [sklearn.neural\_network.BernoulliRBM](http://sklearn.apachecn.org/cn/0.19.0/modules/generated/sklearn.neural_network.BernoulliRBM.html#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](http://sklearn.apachecn.org/cn/0.19.0/modules/generated/sklearn.preprocessing.Normalizer.html#sklearn.preprocessing.Normalizer) ，实用程序类 [Binarizer](http://sklearn.apachecn.org/cn/0.19.0/modules/generated/sklearn.preprocessing.Binarizer.html#sklearn.preprocessing.Binarizer) 也被用于 [sklearn.pipeline.Pipeline](http://sklearn.apachecn.org/cn/0.19.0/modules/generated/sklearn.pipeline.Pipeline.html#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](http://sklearn.apachecn.org/cn/0.19.0/modules/generated/sklearn.preprocessing.StandardScaler.html#sklearn.preprocessing.StandardScaler) 和 [Normalizer](http://sklearn.apachecn.org/cn/0.19.0/modules/generated/sklearn.preprocessing.Normalizer.html#sklearn.preprocessing.Normalizer) 类的情况，预处理模块提供了一个相似的函数 [binarize](http://sklearn.apachecn.org/cn/0.19.0/modules/generated/sklearn.preprocessing.binarize.html#sklearn.preprocessing.binarize) ，以便不需要转换接口时使用。

稀疏输入

[binarize](http://sklearn.apachecn.org/cn/0.19.0/modules/generated/sklearn.preprocessing.binarize.html#sklearn.preprocessing.binarize) 以及 [Binarizer](http://sklearn.apachecn.org/cn/0.19.0/modules/generated/sklearn.preprocessing.Binarizer.html#sklearn.preprocessing.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. 分类特征编码**

在机器学习中，特征经常不是数值型的而是分类型的。举个例子，一个人可能有 ["male", "female"] ， ["from Europe", "from US", "from Asia"] ， ["uses Firefox", "uses Chrome", "uses Safari", "uses Internet Explorer"] 等分类的特征。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] 。

这个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：浏览器的类别数据则是任意排序的)

一种将分类特征转换为能够被scikit-learn中模型使用的编码是one-of-K或one-hot编码，在 [OneHotEncoder](http://sklearn.apachecn.org/cn/0.19.0/modules/generated/sklearn.preprocessing.OneHotEncoder.html#sklearn.preprocessing.OneHotEncoder) 中实现。这个类使用 m 个可能值转换为 m 值化特征，将分类特征的每个元素转化为一个值。

考虑如下例子:

>>> enc = preprocessing.OneHotEncoder()

>>> enc.fit([[0, 0, 3], [1, 1, 0], [0, 2, 1], [1, 0, 2]])

OneHotEncoder(categorical\_features='all', dtype=<... 'numpy.float64'>,

handle\_unknown='error', n\_values='auto', sparse=True)

>>> enc.transform([[0, 1, 3]]).toarray()

array([[ 1., 0., 0., 1., 0., 0., 0., 0., 1.]])

默认情况下，每个特征使用几维的数值由数据集自动推断。当然，你也可以通过使用参数``n\_values``来精确指定。 在我们的例子数据集中，有两个可能得性别类别，三个洲，四个网络浏览器。接着，我们训练编码算法，并用来对一个样本数据进行转换。 在结果中，前两个数值是性别编码，紧接着的三个数值是洲编码，最后的四个数值是浏览器编码

注意，如果训练集中有丢失的分类特征值，必须显式地设置 n\_values ，举个例子，

>>> enc = preprocessing.OneHotEncoder(n\_values=[2, 3, 4])

>>> # 注意到第二、三个特征是不全的

>>> # features

>>> enc.fit([[1, 2, 3], [0, 2, 0]])

OneHotEncoder(categorical\_features='all', dtype=<... 'numpy.float64'>,

handle\_unknown='error', n\_values=[2, 3, 4], sparse=True)

>>> enc.transform([[1, 0, 0]]).toarray()

array([[ 0., 1., 1., 0., 0., 1., 0., 0., 0.]])

参见 [从字典类型加载特征](http://sklearn.apachecn.org/cn/0.19.0/modules/feature_extraction.html#dict-feature-extraction) ，它对于分类特征代表一个dict，而不是整数。

**6. 缺失值插补**

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。

[Imputer](http://sklearn.apachecn.org/cn/0.19.0/modules/generated/sklearn.preprocessing.Imputer.html#sklearn.preprocessing.Imputer) 类提供了估算缺失值的基本策略，使用缺失值所在的行/列中的平均值、中位数或者众数来填充。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. ]]

[Imputer](http://sklearn.apachecn.org/cn/0.19.0/modules/generated/sklearn.preprocessing.Imputer.html#sklearn.preprocessing.Imputer) 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，缺失值被编码为0，因此隐式地存储在矩阵中。当缺失值比可观察到的值多的时候，这种格式是合适的。

[Imputer](http://sklearn.apachecn.org/cn/0.19.0/modules/generated/sklearn.preprocessing.Imputer.html#sklearn.preprocessing.Imputer) 可以在 Pipeline 中用作构建支持插补的合成模型。参见 [Imputing missing values before building an estimator](http://sklearn.apachecn.org/cn/0.19.0/auto_examples/plot_missing_values.html#sphx-glr-auto-examples-plot-missing-values-py) 。

**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](http://sklearn.apachecn.org/cn/0.19.0/modules/generated/sklearn.preprocessing.PolynomialFeatures.html#sklearn.preprocessing.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 [:ref:`svm\_kernels`时 ，多项式特征被隐式地使用在 `核函数(kernel methods) <https://en.wikipedia.org/wiki/Kernel\_method>`\_](http://sklearn.apachecn.org/cn/0.19.0/modules/preprocessing.html#id14) 中(比如， [sklearn.svm.SVC](http://sklearn.apachecn.org/cn/0.19.0/modules/generated/sklearn.svm.SVC.html#sklearn.svm.SVC) ， [sklearn.decomposition.KernelPCA](http://sklearn.apachecn.org/cn/0.19.0/modules/generated/sklearn.decomposition.KernelPCA.html#sklearn.decomposition.KernelPCA) )。

创建并使用多项式特征的岭回归实例请见 [Polynomial interpolation](http://sklearn.apachecn.org/cn/0.19.0/auto_examples/linear_model/plot_polynomial_interpolation.html#sphx-glr-auto-examples-linear-model-plot-polynomial-interpolation-py) 。

**.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](http://sklearn.apachecn.org/cn/0.19.0/modules/generated/sklearn.preprocessing.FunctionTransformer.html#sklearn.preprocessing.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)

>>> X = np.array([[0, 1], [2, 3]])

>>> transformer.transform(X)

array([[ 0. , 0.69314718],

[ 1.09861229, 1.38629436]])

使用一个 [FunctionTransformer](http://sklearn.apachecn.org/cn/0.19.0/modules/generated/sklearn.preprocessing.FunctionTransformer.html#sklearn.preprocessing.FunctionTransformer) 类来做定制化特征选择的例子，请见 [Using FunctionTransformer to select columns](http://sklearn.apachecn.org/cn/0.23.0/auto_examples/preprocessing/plot_function_transformer.html#sphx-glr-auto-examples-preprocessing-plot-function-transformer-py) 。