**Scikit-learn学习系列 | 3. sklearn特征提取方法汇总（包含字典、文本、图像的特征提取）**

**1.**[特征工程](https://so.csdn.net/so/search?q=%E7%89%B9%E5%BE%81%E5%B7%A5%E7%A8%8B&spm=1001.2101.3001.7020)**的介绍**

现实世界中数据大体上都是不完整，不一致的脏数据，而数据预处理的目的就是去除噪音，让数据适应模型，匹配模型的需求。数据预处理位于现实问题和模型训练之间，在实际的工程中，数据预处理是多变的，而模型是相对稳定的。因此，是实际的工程应用中，大量的工作是花在数据预处理上，而不是模型算法上。

* **特征工程的分类**  
  1）[特征提取](https://so.csdn.net/so/search?q=%E7%89%B9%E5%BE%81%E6%8F%90%E5%8F%96&spm=1001.2101.3001.7020)（特征编码）：把非数值数据转换成适合模型训练的数值数据  
  2）特征创造（特征升维）：对现有的少量特征进行组合得到新的特征  
  3）特征选择（特征降维）：在众多的特征中选择部分特征，用来表征特定业务应用的样本
* **scikit-learn提供的工具：**  
  1）模块preprocessing：几乎包含数据预处理的所有内容  
  2）模块Impute：填补缺失值专用  
  3）模块feature\_selection：包含特征选择的各种方法的实践  
  4）模块decomposition：包含降维算法
* **特征选择的常见思路与方法**  
  1）根据相同特征，不同样本的特征值自身的特性进行过滤：需要过滤掉不同样本，特征数值差异性较小的特征  
  2）根据特征与标签之间的相关性进行过滤：需要过滤掉特征与标签之间相关性不大的特征  
  3）根据特征与特征之间的相关性进行过滤：需要过滤掉特征与特征之间相关性较大的特征，这说明这部分的特征是相似的  
  4）根据样本与样本之间的相关性进行过滤：如果样本和样本之间也存在相关性，对于相关性强的样本，就可以进行合并来降低样本的数量

简单来说，**就是特征之间比较相似需要过滤，特征与标签没啥关系需要过滤，样本之间比较相似也需要过滤。**

**2. 特征提取的方法**

比较容易混淆的特征提取和特征选择这两个概念：

**特征提取**：将原始特殊数据转换为一组具有明显物理意义（Gabor、几何特征[角点、不变量]、纹理[LBP HOG]）或者统计意义或核的特征数据  
**特征选择**：从特征集合中挑选一组最具统计意义和典型性的特征，而去掉非典型性、不重要的特征，达到特征的降维。用马克思哲学的观点来看：就是抓住主要矛盾、以及矛盾的主要方面，这就是特征选择

* **特征提取的方法主要有三种：**  
  1）从字典中提取特征：DictVectorizer  
  2）从文本中提取特征：词频向量(CountVectorizer)、TF-IDF向量(TfidfVectorizer,TfidfTransformer)、特征哈希向量(HashingVectorizer)  
  3）从图像中提取特征：提取像素矩阵提取边缘和兴趣点
* **特征提取的介绍：**  
  [机器学习](https://edu.csdn.net/cloud/sd_summit?utm_source=glcblog&spm=1001.2101.3001.7020)的模型是基于样本的特征来进行了，而不是基于原始的数据来进行的，所以需要特征工程来对原始的数据进行数据的预处理与特征提取等相关操作。机器学习的模型，只能够识别经过认为提取后的特征，而不是能识别原始数据，这个过程就是特征提取，比如从文本数据中提取每个单词的出现的频率值，并转换成词频向量。  
  对于一些比较复杂的任务，原始数据的特征是不明确的，人无法从中提取都有用的特征，这时候就需要交给深度学习来自动的学习特征。也就是说，原始图片中物体的轮廓、颜色，大小、位置等信息，是不太可能通过人来为某个具体的图片来提取他们的特征，然后在提供给模型的，只能让模型自己去发现原始图片（像素）中物体的特征。所以，大多数的情况下，发现原始数据内在特征的模型比分类或拟合算法模型更加的重要。  
  **以计算机视觉来举例子，为什么需要搭建出如何复杂的卷积神经网络，设置如此多的技巧与各种的注意力机制，其实本质上就是为了更好地提取到图像中的特征，这个过程是机器抽象与复杂的，人类无法完成所有需要交给神经网络来自行学习到相关的特征。所有，卷积神经网络的大部分是为了挖掘出潜在的特征的**，而其后的全连接层表简单，它是用来分类的。

**3. 字典中提取特征**

用python中的字典存储原始数据（可能是字符串数据）是一种常用的做法，但sklearn的模型的输入特征必须是numpy或scipy数组，存放的必须是数字，这时候就需要把字典中的字符型特征数据转换成numpy或scipy数组的数字。

这时候可以使用**DictVectorizer**，其包含两个作用：  
1）从字典加载特征时可以转换为numpy格式的数组，也就是可以直接喂入模型；  
2）对于字典中含有字符串的特征自动采用独热编码（one-hot）表示；

from sklearn.feature\_extraction import DictVectorizer

# 定义字典，信息格式为 key：value

# 如下有三个样本，每个样本2个特征值：城市、温度

X = [

{'city': 'Dubai', 'temperature': 33.},

{'city': 'London', 'temperature': 12.},

{'city': 'San Francisco', 'temperature': 18.},

]

# 创建一个字典转换特征的对象

vectorizer = DictVectorizer()

# 把measurements字典中的value值（字符串）转换成oneHot编码值

vectorizer.fit\_transform(X)

# 输出：

# array([[ 1., 0., 0., 33.],

# [ 0., 1., 0., 12.],

# [ 0., 0., 1., 18.]])

# 打印相关信息

vectorizer.feature\_names\_

# 输出：['city=Dubai', 'city=London', 'city=San Francisco', 'temperature']

vectorizer.vocabulary\_

# 输出：{'city=Dubai': 0, 'temperature': 3, 'city=London': 1, 'city=San Francisco': 2

# 以上numpy输出的第一列，如果标为1表示城市为Dubai

# 以上numpy输出的第二列，如果标为1表示城市为London

# 以上numpy输出的第三列，如果标为1表示城市为San Francisco

# 参数sort表示是否为特征排序，如下所示：

# DictVectorizer(sort=True): {'city=Dubai': 0, 'temperature': 3, 'city=London': 1, 'city=San Francisco': 2}

# DictVectorizer(sort=False): {'city=Dubai': 0, 'temperature': 1, 'city=London': 2, 'city=San Francisco': 3}

# 还可以获取对象的设置参数

vectorizer.get\_params()

# 输出：{'dtype': numpy.float64, 'separator': '=', 'sort': True, 'sparse': False}

# 设置sparse参数为False同样可以直接获得编码矩阵，不需要toarray转换，如下所示这下面代码作业是等价的

vectorizer = DictVectorizer(sparse=True)

vectorizer.fit\_transform(X).toarray()

# 输出：

# array([[ 1., 0., 0., 33.],

# [ 0., 1., 0., 12.],

# [ 0., 0., 1., 18.]])

vectorizer = DictVectorizer(sparse=False)

vectorizer.fit\_transform(X)

# 输出：

# array([[ 1., 0., 0., 33.],

# [ 0., 1., 0., 12.],

# [ 0., 0., 1., 18.]])

https://csdnimg.cn/release/blogv2/dist/pc/img/newCodeMoreWhite.png

说明：由于对于城市这个特征来说，其有字符串表示所有需要将字符串编码表示；而对于温度这个特征来说，因为其本来就是一个数值，所有不需要进行编码表示，可以直接转换为numpy的数值格式。

**4. 文本中提取特征**

给定的样本数据以一串字符串文本，字符串文本本身是无法进行模型训练的，这就需要对所有样本中的所有字符串单词进行统一编码，然后用编码后的数据表示原先的字符串文本。需要注意，这时候与字符串编码的情况已经不同了，因为现在需要处理的是文本，需要对整个文本进行编码成特征向量，而不是简单的对某一个特征的字符串内容进行编码。

此时，每一个字符串文本称为一个样本，样本字符串中的字符的个数有长有短，而不是一个单词是一个样本，也不是整篇文章是一个样本。由于需要对这个文本编码为特征向量，那么直接使用ASCII编码是没有作用的，因为其只是简单的翻译，没有提取到文本的特征信息，比如词频，词的重要性等。

**4.1 特征提取方法：词频向量编码(CountVectorizer)**

**英语词频向量**

词频就是单词出现的频率，单词出现的频率，一定程度上反应了文本内在的特征。文本是一篇文章或一段话，或一句话等等，它是样本的最小单位。对于英文的词频特征编码就是根据文本中单词出现的次数进行向量编码，其具有几个性质：

1）向量中每个分量的值反应了单词出现的频次  
2）向量的长度，反应了所有样本中单词的个数  
3）每个向量位置代表的单词，依据OneHot编码

**CountVectorizer的工作流程：**

* 预处理：CountVectorizer 类会将文档（文本）全部转换成小写。
* 预处理：词块大多是单词，但是他们也可能是一些短语，字母长度小于2的词块(如 I, a)被略去。可以用stop\_words选项排除一些常用但没有太多意义的助词(如is,are,in)，eg：stop\_words = ‘english’
* 分词：把句子分割成词块（token）或有意义的字母序列。
* 词频统计：统计它们出现的次数。
* 构建字典：把文本转换成字典，如：{‘this’: 8, ‘is’: 3, ‘the’: 6, ‘first’: 2, ‘document’: 1, ‘second’: 5, ‘and’: 0, ‘third’: 7, ‘one’: 4}
* 词向量编码：把字典转换成词向量编码（类似oneHot编码）

from sklearn.feature\_extraction.text import CountVectorizer

# 构建样本数据，这里设置一句话表示一个样本

X = [

'This is the first document.',

'This document is the second document.',

'And this is the third one.',

'Is this the first document?',

]

# 构建一个词频向量转换器

# 参数说明：

# stop\_words: 停用词，可以用.get\_feature\_names()函数看见选择哪个文本

vectorizer = CountVectorizer()

vectorizer.fit\_transform(X).toarray()

# 输出：

# array([[0, 1, 1, 1, 0, 0, 1, 0, 1],

# [0, 2, 0, 1, 0, 1, 1, 0, 1],

# [1, 0, 0, 1, 1, 0, 1, 1, 1],

# [0, 1, 1, 1, 0, 0, 1, 0, 1]])

# 获取使用的特征单词

vectorizer.get\_feature\_names\_out()

# 输出：array(['and', 'document', 'first', 'is', 'one', 'second', 'the', 'third', 'this'], dtype=object)

vectorizer.get\_feature\_names()

# 输出：['and', 'document', 'first', 'is', 'one', 'second', 'the', 'third', 'this']

# 获取位置编码信息

vectorizer.vocabulary\_

# 输出：

# {'this': 8, 'is': 3, 'the': 6, 'first': 2, 'document': 1, 'second': 5, 'and': 0, 'third': 7, 'one': 4}

# 单词编码后，转换了key:value字典，value表示在oneHot编码中的位置

# X 中具有非零条目的每个文档的术语

Y = vectorizer.fit\_transform(X).toarray()

vectorizer.inverse\_transform(Y)

# 输出:

# [array(['document', 'first', 'is', 'the', 'this'], dtype='<U8'),

# array(['document', 'is', 'second', 'the', 'this'], dtype='<U8'),

# array(['and', 'is', 'one', 'the', 'third', 'this'], dtype='<U8'),

# array(['document', 'first', 'is', 'the', 'this'], dtype='<U8')]

https://csdnimg.cn/release/blogv2/dist/pc/img/newCodeMoreWhite.png

**分析：**

这里以第二个向量编码来解释说明，可以看见样本 'This document is the second document.'被编码为了一个向量：[0, 2, 0, 1, 0, 1, 1, 0, 1]。对于着vectorizer.vocabulary\_输出的位置编码信息，从0位开始说明：

第0位的0：第0位表示and单词的OneHot编码的位置；0表示and在该单个样本中的次数为0；  
第1位的2：第1位表示document单词的OneHot编码的位置；2表示document在该单个样本中的次数为2；  
第2位的0：第2位表示first单词的OneHot编码的位置；0表示first在该单个样本中的次数为0；  
第3位的1：第3位表示is单词的OneHot编码的位置；1表示is在该单个样本中的次数为1；  
第4位的0：第4位表示one单词的OneHot编码的位置；0表示one在该单个样本中的次数为0；  
第5位的1：第5位表示second单词的OneHot编码的位置；1表示second在该单个样本中的次数为1；  
第6位的1：第6位表示the单词的OneHot编码的位置；1表示the在该单个样本中的次数为1；  
第7位的0：第7位表示third单词的OneHot编码的位置；0表示third在该单个样本中的次数为0；  
第8位的1：第8位表示this单词的OneHot编码的位置；1表示this在该单个样本中的次数为1；

也就就说，词频向量只考虑了样本中单词出现的频次，频次越高，向量中该单词对应分量位置的数值越大。但是没有考虑单词的语义和上下文关系。

**中文词频向量**

英文有26个单字母，26个字母组成单词，如mine，单词之间通过空格分开，因此，因为的分词是比较简单的，通过空格就可以区分开。中文的单字和单词之间都没有空格分离，只有句子之间才有分割，这就给中文的分词带来了极大的麻烦。中文的分词有点类似英文的词组，比如北京，这是两个中文字，但实际是一个中文单词。

所以，中文分词需要专门的库，相当于在文本中对中文词与词之间增加一个类似英文的“空格”的过程。经过中文分词后的文本，就可以给CountVectorizer进行处理了。jieba库是一款优秀的 Python 第三方中文分词库，jieba 支持三种分词模式：精确模式、全模式和搜索引擎模式，下面是三种模式的特点。

1）精确模式：试图将语句最精确的切分，不存在冗余数据，适合做文本分析。  
2）全模式：将语句中所有可能是词的词语都切分出来，速度很快，但是存在冗余数据  
3）搜索引擎模式：在精确模式的基础上，对长词再次进行切分。

安装jieba中文分词库：! pip install jieba

Collecting jieba

Downloading jieba-0.42.1.tar.gz (19.2 MB)

━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 19.2/19.2 MB 5.6 MB/s eta 0:00:0000:0100:01m

Preparing metadata (setup.py) ... done

Building wheels for collected packages: jieba

Building wheel for jieba (setup.py) ... done

Created wheel for jieba: filename=jieba-0.42.1-py3-none-any.whl size=19314476 sha256=0e45ceb6ce813e0b435498c1122f032ff843dac7d1339006dc59f700e9e47480

Stored in directory: /home/fs/.cache/pip/wheels/7d/74/cf/08c94db4b784e2c1ef675a600b7b5b281fd25240dcb954ee7e

Successfully built jieba

Installing collected packages: jieba

Successfully installed jieba-0.42.1

使用jieba分词后，与英文词频编码类似的：

from sklearn.feature\_extraction.text import CountVectorizer

import jieba

X = [

'朋友，小红是我的好朋友',

'小明对小红说：“小红，我们还是不是朋友”',

'小明与小红是朋友'

]

# 对中文文本进行分词

cutX = ["|".join(jieba.cut(x)) for x in X]

# cutX输出：

# ['朋友|，|小红|是|我|的|好|朋友',

# '小明|对|小红|说|：|“|小红|，|我们|还|是不是|朋友|”',

# '小明|与|小红|是|朋友']

# 对于jieba分词后的数据是不能直接显示的，print也不行，不过可以转换为list格式输出

# [list(jieba.cut(x)) for x in X]

# 输出：

# [['朋友', '，', '小红', '是', '我', '的', '好', '朋友'],

# ['小明', '对', '小红', '说', '：', '“', '小红', '，', '我们', '还', '是不是', '朋友', '”'],

# ['小明', '与', '小红', '是', '朋友']]

# 生成一个词频转换器对象, 去除一些不意义的词“好的是的”

vectorizer = CountVectorizer(stop\_words = ['好的','是的'])

vectorizer.fit\_transform(cutX).toarray()

# 输出：

# array([[0, 1, 0, 0, 2],

# [1, 2, 1, 1, 1],

# [1, 1, 0, 0, 1]])

# 获取位置编码信息

vectorizer.vocabulary\_

# 输出：{'朋友': 4, '小红': 1, '小明': 0, '我们': 2, '是不是': 3}

**4.2 特征提取方法：权重向量编码(CountVectorizer)**

在上面提到的词频编码中，词频向量中存放的是出现的绝对数值，这有两个缺点：  
1）不同文章，长短不同，单词的数量自然不同，这样导致不同文章的比较就存现了不公平现象；  
2）有些词，虽然出现的次数比较大，但它本身没有任何含义，这样的词频实际上是没有意义的；

为了解决这样的问题，相比于出现的次数，选择词汇在文中的比例权重更具备特征性。也就是说，可以用词汇在文中所有单词中的权重比例进行编码。使用TfidfTransformer可以把普通的词频向量转换成Tf-idf权重向量

from sklearn.feature\_extraction.text import CountVectorizer, TfidfTransformer

X = [

'This is the first document.',

'This document is the second document.',

'And this is the third one.',

'Is this the first document?',

]

# 构建词频向量转换器

vectorizer = CountVectorizer()

Y = vectorizer.fit\_transform(X).toarray()

# 输出：

# array([[0, 1, 1, 1, 0, 0, 1, 0, 1],

# [0, 2, 0, 1, 0, 1, 1, 0, 1],

# [1, 0, 0, 1, 1, 0, 1, 1, 1],

# [0, 1, 1, 1, 0, 0, 1, 0, 1]])

# 获取位置编码信息

vectorizer.vocabulary\_

# 输出：{'this': 8, 'is': 3, 'the': 6, 'first': 2, 'document': 1, 'second': 5, 'and': 0, 'third': 7, 'one': 4}

# 构建词频权重转换器，其可以将词频转换成对于的权重

transformer = TfidfTransformer()

transformer.fit\_transform(Y).toarray()

# 输出：

# array([[0. , 0.46979139, 0.58028582, 0.38408524, 0. , 0. , 0.38408524, 0. , 0.38408524],

# [0. , 0.6876236 , 0. , 0.28108867, 0. , 0.53864762, 0.28108867, 0. , 0.28108867],

# [0.51184851, 0. , 0. , 0.26710379, 0.51184851, 0. , 0.26710379, 0.51184851, 0.26710379],

# [0. , 0.46979139, 0.58028582, 0.38408524, 0. , 0. , 0.38408524, 0. , 0.38408524]])

https://csdnimg.cn/release/blogv2/dist/pc/img/newCodeMoreWhite.png

**分析：**

对于第一个样本，'This is the first document.'，如果是普通的词频向量，那么这里每个单词出现的次数都是1而已，体现不出出现的单词哪个是更重要的；相比之下，在权重向量中，可以看见1,2位置上的权重都比较大，对于的词汇是’first’: 2, ‘document’: 1，很直观的理解，对于第一个样本来说这两个单词确实是比较重要的，所以权重向量比词频向量更能表示特征。

1）**TfidfTransformer的输入**：文本对应的普通的词频向量  
2）**TfidfTransformer的输出**：二范数的输出，每个向量分量的值，都被限制到[0,1]，这样就消除了文章长度不同造成的词频数量的不同造成的两个文本之间的差异  
3）**TfidfTransformer的转换**：与普通二范数转换的区别是，TfidfTransformer还考虑了所有文本中相同词的影响。TfidfTransformer转换后，代表一篇文章的词向量更加的合理

除了可以使用TfidfTransformer可以把普通的词频向量转换成Tf-idf权重向量外，还可以直接使用TfidfVectorizer，**TfidfVectorizer相当于把CountVectorizer与TfidfTransformer的功能集成了在一起，一步到位。**

from sklearn.feature\_extraction.text import TfidfVectorizer

# 构建样本数据，这里设置一句话表示一个样本

X = [

'This is the first document.',

'This document is the second document.',

'And this is the third one.',

'Is this the first document?',

]

# 设置打印的每行的输出长度，并且取消科学计数法打印

np.set\_printoptions(suppress=True, linewidth=300)

# 构建一个词频向量转换器

# 参数说明：

# stop\_words: 停用词，可以用.get\_feature\_names()函数看见选择哪个文本

vectorizer = TfidfVectorizer()

vectorizer.fit\_transform(X).toarray()

# 输出：每个数值表示单词在文本中的权重

# array([[0. , 0.46979139, 0.58028582, 0.38408524, 0. , 0. , 0.38408524, 0. , 0.38408524],

# [0. , 0.6876236 , 0. , 0.28108867, 0. , 0.53864762, 0.28108867, 0. , 0.28108867],

# [0.51184851, 0. , 0. , 0.26710379, 0.51184851, 0. , 0.26710379, 0.51184851, 0.26710379],

# [0. , 0.46979139, 0.58028582, 0.38408524, 0. , 0. , 0.38408524, 0. , 0.38408524]])

# 获取位置编码信息

vectorizer.vocabulary\_

# 输出：

# {'this': 8, 'is': 3, 'the': 6, 'first': 2, 'document': 1, 'second': 5, 'and': 0, 'third': 7, 'one': 4}

https://csdnimg.cn/release/blogv2/dist/pc/img/newCodeMoreWhite.png

**补充分析：**

这里可以看出，直接使用TfidfVectorizer来文本样本编码构建权重向量，与先使用CountVectorizer将文本样本编码成词频向量，再利用TfidfTransformer将词频向量转换为权重向量。最后得到的结果，二者是相同的，这也进一步说明了TfidfVectorizer是CountVectorizer与TfidfTransformer的集成。

**4.3 特征提取方法：哈希向量编码(HashingVectorizer)**

上述模型的方法在文本中单词数量较小时很好用，也很直接，但在有些场景下很难使用，比如分词后的词汇字典表非常大，达到100万+词（不是单字上百万个），**此时如果直接使用词频向量或Tf-idf权重向量的方法，那么文本的特征向量的分量太多，占用的内存太大**。有可能将内存撑爆。

在这种情况下可以应用哈希向量技巧进行降维（哈希向量是一种特征降维的技术手段），因为对于词汇字典太大的情况，显然对于特征是冗余的，只需要保留主要的有用的特征即可，该合并的合并，该去掉的去掉。

* **散列表（Hash表）**

在我很早的时候，接触过数据结构，曾经也做过一些笔记，当时比较详细的记录了散列表的各方面内容，见链接：[散列表的查找算法（开放地址法，链地址法）](https://blog.csdn.net/weixin_44751294/article/details/104472438)，下面就只是简单介绍一下。

**散列表（Hash table，也叫哈希表）是一种多对1的数据结构，它通过把关键码值映射到表中一个位置来访问记录，以加快查找的速度**。这个映射函数叫做散列函数，存放记录的数组叫做散列表。既然是多对一，就会出现多个输入关键值key，映射到相同的位置，这就是函数冲突。

对于降维的应用来说，既然这些不同的特征最后映射到相同的位置，那么可以认识到他们或许是相似的。那么，这些映射到相同位置的输入，就可以合并在一起，不需要链表区分具备相同映射索引的输入，映射后具备相同索引的所有输入，合并为同一类型，起到降维的目的。文本hash编码就采用的这种策略。

在文本分类中，特征的基本单元是“单词”，每个单词是一个特征，该单词出现的词频是特征值。**Hash函数可以将任意个数的单词，一个个映射到一个固定长度的散列数字中去。这个映射的过程，就完成了不同单词的合并，即不同特征的合**并，并给出了合并后的特征的编号。比如有100万个单词的文本，如果没有hash，这就有100万种特征，假设Hahs表的长度N=1万，经过Hash之后，这些100万个特征就被合并成了1万特征，合并的方法就是Hash函数。

* **Hash函数的特征**

Hash函数是一种多对一的映射关系，其具备以下的几个特性：  
1）正向快速：给定明文和 hash 算法，在有限时间和有限资源内能计算出 hash 值。  
2）逆向困难：给定（若干） hash 值，在有限时间内很难（基本不可能）逆推出明文。  
3）**输入敏感**：原始输入信息修改一点信息，产生的 hash 值看起来应该都有很大不同。  
4）碰撞避免：对于任意两个不同的数据块，其hash值相同的可能性极小，既对于一个给定的数据块，找到和它hash值相同的数据块极为困难。  
5）**输出长度任意指定**：**可以指定任意指定输出长度，比如指定N=10或N=1000.**

目前流行的Hash函数, 包括MD4,MD5,SHA等。Hash映射的本质是特征提取，就原始的单词数据中，提取能够用于机器学习模型的特征。每个索引代表一种特征，并采用OneHot编码，作为特征所在的位置；该位置上的数值，就是特征该特征的特征值，就得到了特征向量。

from sklearn.feature\_extraction.text import HashingVectorizer

X = [

'This is the first document.',

'This document is the second document.',

'And this is the third one.',

'Is this the first document?',

]

# 构建Hash向量转换器

vectorizer = HashingVectorizer()

vectorizer.fit\_transform(X).toarray().shape

# 输出: (4, 1048576)

# 可以看见，这里默认的特征列有10w， 因为这里参数n\_features默认设置default=(2 \*\* 20)

# 但是对于我们的简单文本并不需要如此庞大的阵列，可以通过参数n\_features来设置得小一些

# 再次构建Hash向量转换器，这里进行增维，实现特征的扩充

vectorizer = HashingVectorizer(n\_features=10)

vectorizer.fit\_transform(X).toarray() # 与vectorizer.transform(X)等价

# 输出：

# array([[ 0. , 0. , 0. , 0. , 0. , 0. , -0.57735027, 0.57735027, -0.57735027, 0. ],

# [ 0. , 0. , -0.5 , 0. , 0. , 0.5 , 0. , 0.5 , -0.5 , 0. ],

# [ 0. , 0.40824829, 0.40824829, 0. , -0.40824829, -0.40824829, 0. , 0.40824829, -0.40824829, 0. ],

# [ 0. , 0. , 0. , 0. , 0. , 0. , -0.57735027, 0.57735027, -0.57735027, 0. ]])

# 再次构建Hash向量转换器，这里降维，实现hash的合并

vectorizer = HashingVectorizer(n\_features=3)

vectorizer.transform(X).toarray()

# 输出：

# array([[ 0. , 0.4472136 , -0.89442719],

# [ 0. , 0.70710678, -0.70710678],

# [ 0.40824829, 0.40824829, -0.81649658],

# [ 0. , 0.4472136 , -0.89442719]])

https://csdnimg.cn/release/blogv2/dist/pc/img/newCodeMoreWhite.png

可以看见，这里无论样本的长度多长多短，总可以缩放到一个固定长度的编码向量中，而且每个向量的分量值都会被限制在[0,1]之间，也就是进行了归一化处理。Hash表达长度N越小，Hash冲突的可能性越大

**5. 图像中提取特征**

scikit中也提供了图像的特征提取，不过这里一般还是搭专门的网络结构来设计，这里只是简单的看一看吧。**这里提供了一个图片补丁输出函数extract\_patches\_2d，就是可以将一张图像切分成很多个补丁，也就是小的图片，这个思想是在vision transofermer中首次提出来的，而已的目的是可以将图片转换为序列信息**。然后可以用序列模型来处理图像，提取图像的信息。

**单通道补丁提取**

from sklearn.feature\_extraction.image import extract\_patches\_2d, reconstruct\_from\_patches\_2d

# 首先模拟单通道为例：

image = np.arange(4\*4).reshape((4,4))

# 输出：

# array([[ 0, 1, 2, 3],

# [ 4, 5, 6, 7],

# [ 8, 9, 10, 11],

# [12, 13, 14, 15]])

# 这里可以清楚的看见，这里的补丁提取是每移动一个像素就提取为一个补丁

patches = extract\_patches\_2d(image, patch\_size=(2, 2))

# 输出：

# array([[[ 0, 1],

# [ 4, 5]],

# [[ 1, 2],

# [ 5, 6]],

# [[ 2, 3],

# [ 6, 7]],

# [[ 4, 5],

# [ 8, 9]],

# [[ 5, 6],

# [ 9, 10]],

# [[ 6, 7],

# [10, 11]],

# [[ 8, 9],

# [12, 13]],

# [[ 9, 10],

# [13, 14]],

# [[10, 11],

# [14, 15]]])

# 通过对重叠区域进行平均来从补丁重建原始图像

reconstruct\_from\_patches\_2d(patches, (4, 4))

# 输出：

# array([[ 0., 1., 2., 3.],

# [ 4., 5., 6., 7.],

# [ 8., 9., 10., 11.],

# [12., 13., 14., 15.]])

# 判断重建后的图像是否与原图像一致

(image == reconstructed).all()

# 输出：True，所以是一致的

这里可以通过PIL来直观的查看提取出来的补丁的图像：

from sklearn.feature\_extraction.image import extract\_patches\_2d

import cv2

from PIL import Image

# 读取图像

image = cv2.imread('./photo/cow.jpg')

# 获取补丁，这里说明一下参数的：

# patch\_size：设置补丁的大小

# max\_patches：设置提取补丁的数量

# random\_state：设置随机种子

patches = extract\_patches\_2d(image, patch\_size=(300, 300), max\_patches=10, random\_state=42)

# patches.shape： (10, 300, 300, 3)

# 对图像矩阵转换为图片格式输出

Image.fromarray(patches[0])

* 原图像为：



* 这里选择的补丁输出图像为：



**多通道补丁提取**

from sklearn.feature\_extraction.image import extract\_patches\_2d, reconstruct\_from\_patches\_2d

# 首先模拟单通道为例：

image = np.arange(3\*4\*4).reshape(4, 4, 3)

# 输出：

# array([[[ 0, 1, 2],

# [ 3, 4, 5],

# [ 6, 7, 8],

# [ 9, 10, 11]],

# [[12, 13, 14],

# [15, 16, 17],

# [18, 19, 20],

# [21, 22, 23]],

# [[24, 25, 26],

# [27, 28, 29],

# [30, 31, 32],

# [33, 34, 35]],

# [[36, 37, 38],

# [39, 40, 41],

# [42, 43, 44],

# [45, 46, 47]]])

# 这里可以清楚的看见，这里的补丁提取是每移动一个像素就提取为一个补丁

extract\_patches\_2d(image, patch\_size=(3,3), max\_patches=2)

# 输出：

# array([[[[ 0, 1, 2],

# [ 3, 4, 5],

# [ 6, 7, 8]],

# [[12, 13, 14],

# [15, 16, 17],

# [18, 19, 20]],

# [[24, 25, 26],

# [27, 28, 29],

# [30, 31, 32]]],

# [[[ 3, 4, 5],

# [ 6, 7, 8],

# [ 9, 10, 11]],

# [[15, 16, 17],

# [18, 19, 20],

# [21, 22, 23]],

# [[27, 28, 29],

# [30, 31, 32],

# [33, 34, 35]]]])

**参考资料：**

1. 博主“**文火冰糖的硅基工坊**”的专栏——[机器学习与scikit-learn](https://blog.csdn.net/hiwangwenbing/category_11649697.html)