



sklearn中的特征提取

🕒 Reading time ~5 minutes

1.介绍

sklearn.feature_extraction模块，可以用于从包含文本和图片的数据集中提取特征，以便支持机器学习算法使用。

注意：**Feature extraction**与**Feature Selection**是完全不同的：前者将专有数据（文本或图片）转换成机器学习中可用的数值型特征；后者则是用在这些特征上的机器学习技术。

2. 从字典中load特征

类DictVectorizer可以用于将各列使用标准的python dict对象表示的特征数组，转换成sklearn中的estimators可用的NumPy/SciPy表示的对象。

python的dict的优点是，很方便使用，稀疏，可以存储feature名和值。

DictVectorizer实现了一个称为one-of-K或者”one-hot”编码的类别特征。类别特征是“属性-值”对，它的值严格对应于一列无序的离散概率（比如：topic id, 对象类型，tags, names...）

下例中，”city”是类别的属性，而”temperature”是一个传统的数值型feature：

```
>>> measurements = [
...   {'city': 'Dubai', 'temperature': 33.},
...   {'city': 'London', 'temperature': 12.},
...   {'city': 'San Fransisco', '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 Fransisco', 'temperature']
```

对于在NLP模型中的训练序列型分类器（sequence classifier）来说，DictVectorizer是一个很有用的转换器。它通常会围绕一个感兴趣词进行feature窗口抽取。

例如，我们的第一个算法是抽取词性（Part os Speech：PoS）标签，我们希望用互补标签来训练一个序列分类器（比如：chunker）。下面的dict就是这样一个特征窗口，它围绕着句子“The cat sat on the mat”中的单词“sat”进行抽取：

```
>>> 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
... ]
```

这个描述可以被向量化成一个稀疏的2维矩阵，适合用在分类器中（或者通过text.TfidfTransformer 进行归一化后再进行）

```
>>> 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']
```

你可以想像下，如果从一个文档语料中围绕单个词抽取这样的上下文，产生的矩阵将会十分宽（许多one-hot特征），大多数的值为0。为了让生成的数据结构可以在适配内存，缺省情况下，DictVectorizer类使用一个scipy.sparse矩阵，而非numpy.ndarray。

3.Feature hashing

FeatureHasher类是一个高速、低内存的vectorizer，它使用的技术是：feature hashing，或者是“hashing trick”。对比于以往的方式：在构建一个在训练时遇到的feature的hash表（类似于vectorizer），FeatureHasher会对feature使用一个hash函数来直接决定在样本矩阵中它们的列索引。结果将会大大提速，并降低内存使用；这种hasher不会记得输入feature长得啥样，它没有inverse_transform方法进行可逆转换。

由于hash函数会在不同的feature之间引起冲突，我们使用一个有符号的hash函数，hash值的符号决定了在feature的输出矩阵中的值的符号。这种方式下，冲突很可能会被消除，而非累加错误，输出feature的期望均值为0。

如果在构造器中传入参数：non_negative=True，将使用绝对值。这会undo对一些冲突处理，但是允许输出传给这样的estimators：sklearn.naive_bayes.MultinomialNB或者sklearn.feature_selection.chi2 这样的feature selectors希望非负输入。

FeatureHasher接受其它的mappings（比如：python的dict和它在collections模块中的变种），(feature,value)pair，或者string，这完全取决于构造器中的input_type参数。Mapping被当成是(feature,value)pair的一个列表，而单个strings则具有显式的值1，因此['feat1', 'feat2', 'feat3']被解释成[(‘feat1’, 1), (‘feat2’, 1), (‘feat3’, 1)]。如果单个feature在同一个样本中出现多次，那么相应的值可以加和下：(‘feat’, 2)和(‘feat’, 3.5) 相加等于(‘feat’, 5.5)。FeatureHasher总是使用scipy.sparse的CSR格式。

Feature hashing可以用于文档分类，但不同于text.CountVectorizer，除了做Unicode-to-UTF8外，FeatureHasher不会做分词，或者任何预处理。详见下面说到的[Vectorizing a large text corpus with the hashing trick]。

下面的示例，会考虑一个词级别的NLP任务，它需要从(token, part_of_speech)pair中提取特征。一个可以使用一个python generator函数来抽取features：

```
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)
```

接着，将得到的raw_X传给FeatureHasher.transform：

```
raw_X = (token_features(tok, pos_tagger(tok)) for tok in corpus)
```

传给hasher：

```
hasher = FeatureHasher(input_type='string')
X = hasher.transform(raw_X)
```

来得到一个scipy.sparse稀疏矩阵X。

注意：generator的使用，在feature extraction中引入了延迟（laziness）：从hasher按需要处理token。

3.1 实现细节

FeatureHasher使用有符号的32-bit的MurmurHash3变种。因而（由于scipy.sparse的限制），支持feature的最大数目为：📄

由Weinberger提出的hashing trick的原始公式，使用两个独立的hash函数📄和📄各自决定列索引和feature的符号。当前实现的假设是：使用MurmurHash3的有符号位与其它位是独立的。

由于将hash函数转换成一个列索引时需要使用了一个简单的模运算，建议使用参数n_features的平方；否则feature将不会被映射到相应列。

参考：

- [Feature hashing for large scale multitask learning](#)
- [MurmurHash3](#)

4.文本feature抽取

4.1 词袋表示（BOW）

文本分析是机器学习算法最重要的应用领域。对于原始数据，一串符号不能直接传给算法，必须将它们表示成使用固定size的数值型的feature向量，而非变长的原始文档。

为了这个表示，sklearn提供了相应的工具类来从文本内容中抽取数值型feature，对应的功能名为：

- **tolenizing**：对strings进行token化，给每个可能的token一个integer id，例如：使用空格或逗豆做作token分割符。
- **counting**: 统计在每个文档中token的出现次数。
- **normalizing**和**weighting**：对在大多数样本/文档中出现的重要的token进行归一化和降权。

在这种scheme下，features和samples的定义如下:

- 每个独立的token出现率（不管是否归一化），都被当成一个feature。
- 对于一个给定的文档，所有的token频率的向量都被认为是一个多元样本（**multivariate sample**）。

一个文档型语料可以被表示成一个矩阵：每个文档一行，每个在语料中出现的token一列（word）。

我们将文本文档转成数值型feature向量的过程称为向量化（**vectorization**）。这种特定的策略（tokenization/counting/normalization）被称为词袋（Bag of Words）或者”Bag of n-grams”表示。通过单词出现率文档描述的文档会完全忽略文档中单词的相对位置。

4.2 Sparsity

大多数文档通常使用一个非常小的词料子集，产生的矩阵许多feature值都为0（通常99%以上）

例如：一个10000个短文本文档（比如email）的集合，会使用一个100000个唯一词汇表，每个文档将独自使用100到1000个唯一的单词。

为了能够在内存中存储这样一个矩阵，也同时为了加速matrix/vector的代数运算，该实现通常使用一个使用scipy.sparse包的稀疏表示。

4.3 常用的Vectorizer

CountVectorizer同时实现了tokenization和counting：

```
>>> from sklearn.feature_extraction.text import CountVectorizer
```

该模块有许多参数，缺省值是相当合理的，详见。

```
>>> vectorizer = CountVectorizer(min_df=1)
>>> 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)
```

使用它进行tokenize和count:

```
>>> 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>
```

缺省配置对对至少两个字母以上的单词string进行token化时。显示：

```
>>> analyze = vectorizer.build_analyzer()
>>> analyze("This is a text document to analyze.") == (
...   ['this', 'is', 'text', 'document', 'to', 'analyze'])
True
```

在analyzer中发现的每个term，在fit期间会根据在结果矩阵中的列分配一个唯一的integer索引。这个过程如下：

```
>>> 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名转换到列索引的映射关系，会保存在vectorizer的vocabulary_属性上：

```
>>> vectorizer.vocabulary_.get('document')
1
```

在训练语料中没有看到的词，在后续的调用和转换中，会被完全忽略：

```
>>> vectorizer.transform(['Something completely new.']).toarray()
...
array([[0, 0, 0, 0, 0, 0, 0, 0, 0]]...)
```

注意，在之前的语料中，第一个和最后一个文档具有一样的单词组成，因而被编码成相同的vector。假如最后一个文档是个问句，我们会丢失信息。为了保留一些有序的信息，我们可以抽取2-grams的词汇，而非使用1-grams：

```
>>> 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
```

通过该vectorizer抽取的词汇表，比之前的方式更大，可以以local positioning patterns进行模糊编码：

```
>>> 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]]...)
```

特别的，这种interrogative的形式“Is this”只保存在最后的文档中：

```
>>> feature_index = bigram_vectorizer.vocabulary_.get('is this')
>>> X_2[:, feature_index]
array([0, 0, 0, 1]...)
```

4.4 TF-IDF term weighting

在大文本语料中，一些词语出现非常多（比如：英语中的“the”，“a”，“is”），它们携带着很少量的信息量。我们不能在分类器中直接使用这些词的频数，这会降低那些我们感兴趣但是频数很小的term。

我们需要对feature的count频数做进一步re-weight成浮点数，以方便分类器的使用，这一步通过tf-idf转换来完成。

tf表示词频（term-frequency），idf表示inverse document-frequency，tf-idf表示tf * idf。它原先适用于信息检索（搜索引擎的ranking），同时也发现在文档分类和聚类上也很好用。

使用TfidfTransformer 进行归一化。

```
>>> from sklearn.feature_extraction.text import TfidfTransformer
>>> transformer = TfidfTransformer()
>>> transformer
TfidfTransformer(norm='l2', smooth_idf=True, sublinear_tf=False,
                  use_idf=True)
```

可以详见此。

示例：第1个term每次都100%出现，因而最不感兴趣。其它两个feature的出现率<50%，因而获得更多的文档内容概率表示。

```
>>> 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([[ 0.85...,  0. ...,  0.52...],
       [ 1. ...,  0. ...,  0. ...],
       [ 1. ...,  0. ...,  0. ...],
       [ 1. ...,  0. ...,  0. ...],
       [ 0.55...,  0.83...,  0. ...],
       [ 0.63...,  0. ...,  0.77...]])
```

每行都要归一化成一个单元欧拉范式。每个feature的权重通过fit方法计算，并被保存到模型属性中：

4.6 应用与示例

BOW表示法在实际应用中相当简单并且很有用。

实际上，在**supervised setting**时它与线性模型相结合，可以很快地训练文档分类器：

- [Classification of text documents using sparse features](#)

在**unsupervised setting**中，可以用于分组相似文本：

- [Clustering text documents using k-means](#)

最后，可能发现语料的主要主题，通过使用非负矩阵分解（NMF或NNMF）：

- [Topic extraction with Non-negative Matrix Factorization and Latent Dirichlet Allocation](#)

4.7 BOW表示法的限制

unigrams集（BOW）不能捕获句字和多个词的表示，会丢失掉词的顺序依存。另外，BOW模型不能解释可能的误拼（misspellings）或者词派生（word derivations）。

N-grams可以搞定！不同于unigrams (n=1), 你可以使用bigrams(n=2)。

例如：当我们处理一个包含两个文档的语料：['words','wprds']。第二个文档包含了一个误拼的单词。如果使用bow模型，它将认为两个完全不同文档。而2-gram表示法，则会发现两个文档在8个feature中的4个是匹配的，因而可以更好地进行分类决策：

```
>>> ngram_vectorizer = CountVectorizer(analyzer='char_wb', ngram_range=(2, 2), min_df=1)
>>> 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, 0, 1, 1, 1, 0],
       [1, 1, 0, 1, 1, 0, 1]])
```

在上面的示例中，使用‘char_wb’分析器，它可以在字符边界内创建n-grams的字符（两边使用空格补齐）。而‘char’分析器则可以基于词来创建n-grams。

```
>>> ngram_vectorizer = CountVectorizer(analyzer='char_wb', ngram_range=(5, 5), min_df=1)
>>> 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), min_df=1)
>>> 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
```

单词边界感敏的变种char_wb，可以处理使用空格分割的句子，它比原始的char具有更小的噪声特征。对于这样的语言来说，它可以增加预测精度，加速分类器训练，保持健壮性，忽略误拼和词派生。

通过抽取n-grams（而非独立的单词），我们还可以保留局部的位置信息。bag of words和bag of n-grams两者间，摧毁了大部分文档内部结构，保留了大多数有意义的信息。

为了NLU（Natural Language Understanding），句子和段落的局部结构是必须的。许多这样的模型被转成“结构化输出（Structured output）”问题，这在sklearn的范围之外。

4.8 使用hashing trick对大文本语料进行向量化

上面的vectorization scheme都很简单，实际上它持有着一个从string token到整型feature索引的内存映射（ vocabulary_属性 ），在处理大数据集时会引起许多问题。

- 对于越大的语料，词汇量也会增长，会使用更多内存。
- fitting需要满足所需的内部数据结构的内存分配
- 构建word-mapping需要一个完整的数据集，它不可能以一个严格的在线方式（ online ）去拟合文本分类器
- 使用一个大的vocabulary_，picking和unpicking vectorizers将会很慢（ 比对相同大小的Numpy中的数组进行picking/unpicking更慢 ）
- 对vectorization任务进行分割不是很容易，vocabulary_属性具有一个共享状态，它使用了一个锁（ fine grained synchronization barrier ）：将string token映射到feature index依赖于每一个token出现的顺序，很难共享，并对并发的workers性能有影响，更慢。

可以通过“hashing trick” (sklearn.feature_extraction.FeatureHasher)技术来克服这个缺点，可以使用CountVectorizer进行文本预处理和token化feature。

```
>>> 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>
```

我们可以看到在相应的输出向量中，有16个非0的feature token被抽取出来：比之前使用CountVectorizer小（ 19个 ）。这是因为hash函数有冲突（ 由于n_features参数值过低 ）。

在现实中，n_features参数可以使用缺省值：2 ** 20（ 足够上千w的features了 ）。如果内存或下游的模型的size的选择一个小点的值：比如2**18，可能就不会在文本分类上引入过多的冲突。

注意，维度不会影响算法的CPU训练时间，它通过CSR矩阵（ LinearSVC(dual=True), Perceptron, SGDClassifier, PassiveAggressive ） ，但它对于CSC矩阵是有影响的（ 比如：LinearSVC(dual=False), Lasso() ）

缺省设置下的示例：

```
>>> hv = HashingVectorizer()
>>> hv.transform(corpus)
...
<4x1048576 sparse matrix of type '<... 'numpy.float64'>'
  with 19 stored elements in Compressed Sparse ... format>
```

我们不再得到碰撞，但它带来更高维的开销。当然，其它terms仍会冲突。

HashingVectorizer 有下面的限制：

- 不能进行逆运算（ 没有inverse_transform方法 ），也不能访问features的原始表示。单向
- 不能提供IDF权重。可以考虑添加TfidfTransformer到pipeline中。

4.9 执行out-of-core的scaling

使用HashingVectorizer的另一个好处是，可以执行基于外存out-of-core的归一化。这意味着你不需要主存也可以学习数据。

实现out-of-core的策略是将数据以mini-batch的方式提供给estimator。每个mini-batch都使用HashingVectorizer进行向量化，以保证estimator的输入空间仍然是相同的维度。任何时候内存的使用都限制在mini-batch的size中。尽管对于数据量没有限制，但从实际来看，学习时间经常受CPU时间限制。

示例：

- Out-of-core classification of text documents

4.10 定制vectorizer类

通过定制一些函数并传给一个vectorizer的回调即可：

```
>>> def my_tokenizer(s):
...     return s.split()
...
>>> vectorizer = CountVectorizer(tokenizer=my_tokenizer)
>>> vectorizer.build_analyzer()(u"Some... punctuation!") == (
...     ['some...', 'punctuation!'])
True
```

- preprocessor:
- tokenizer:
- analyzer:

...

参考：

1.http://scikit-learn.org/stable/modules/feature_extraction.html

LIKE TWEET +1

0 Comments

d0evi1

Login ▾

Recommend

Share

Sort by Best ▾



Start the discussion...

LOG IN WITH



OR SIGN UP WITH DISQUS

Name

Be the first to comment.

ALSO ON D0EVI1

corpora.dictionary - 构建 word id 映射

1 comment • 4 months ago •

鄭宜崴 — 想請問 當我們建立Dictionary或Corpus後要如何知道哪一個id 對應到 哪一個word ...

libfm 1.4.2 manual

1 comment • 7 months ago •

Hongru Liu — 请问libfm学习后的结果在哪里会展示出来(如果不指定out的话)？

我的推荐：机器学习资料

1 comment • a year ago •

Yiyi Liu — 工具篇找不到网页了亲TAT

sklearn中的模型评估

4 comments • a year ago •

法布雷亚戈 — 谢谢。有空我改一下，现在免费可用的图片外链都越来越少了。----- 原始邮件 ...

