数据特征处理之特征哈希（Feature Hashing）

标签：　#机器学习#　　#特征工程#　　#预处理#　　时间：2018/09/26 16:56:23　　作者：小木

一、特征哈希（Feature Hashing/Hashing Trick）简介

大多数机器学习算法的输入要求都是实数矩阵，将原始数据转换成实数矩阵就是所谓的特征工程（Feature Engineering），而特征哈希（feature hashing，也称哈希技巧，hashing trick）就是一种特征工程技术。它的目标就是将一个数据点转换成一个向量。

我们先看一下对分类数据（categorical data）和文本数据（text data）进行特征工程处理的一般方法。

分类变量（category variable）就是一组有有限值（finite number of values）的变量。如身份证号、广告类别等。最常见的对分类变量的处理是使用独热模型（one-hot encoding）：创建 N 个二元变量，其中 N 是该分类变量所有可能的取值数量。

而对于文本数据的特征处理，最简单的方法是词袋模型（bag-of-word model）：创建 N 个二元变量，其中N是词汇的数量（即不同单词的数量）。对于每个文档来说，创建一个 N 维向量，文档中包含的某个词汇的数量即是这个向量中词汇对应的索引的值。

可以看到，这两种方法非常类似，都创建了高维稀疏的矩阵。而特征哈希是以哈希表（hash table）的方式来实现这两种转换方法。下面简要介绍一下哈希表。

二、哈希表（Hash Table）

哈希表是一种数据结构，它是根据键值（key）来直接访问内存存储位置的数据结构。每个哈希表都是用一个哈希函数（也叫散列函数，hash function）来实现键-值（key-value）对的映射。这种函数可以将任何一种数据或者消息压缩成摘要（即散列值），使得其数据量变小且格式固定。理想的散列函数会把不同的键散列到不同的块中，但是大多数哈希表都存在哈希碰撞（hashing collision）的可能，即不同的键可能会被映射到相同的值上（后面会解释，这一点不影响机器学习模型的效果）。

在运用哈希表的时候，通常我们需要定义输出的范围，例如假设我们希望将输出范围定义在0-N之间，那么我们就可以使用一个函数，可以将输入数据散列到[0,n-1]之间即可。假设我们创建如下的哈希函数，可以将单词映射成五种类别，即0-4索引：

```
h(the) mod 5 = 0
h(quick) mod 5 = 1
h(brown) mod 5 = 1
h(fox) mod 5 = 3
```

那么对于某句话：

`the quick brown fox`

来说，其使用哈希特转换的向量就是：

`(1,2,0,1,0)`

哈希表有如下特性：

- 相同的输入可能有相同的输出（一般情况下比例不高）
- 不同的输出一定对应不同的输入
- 正向计算很简单，反向计算很困难
- 根据输入查找输出效率很高

三、简单的案例

我们以垃圾邮件检测（spam）为例（这属于文本分类的一个应用），假设有如下两封邮件，第一封邮件是垃圾邮件，第二封邮件不是垃圾邮件：

```
i make ten thousand dollars per week just surfing the web! (spam)
are you free for a meeting early next week? (not spam)
```

使用词袋模型，我们构造如下的索引表：

```
i: 0
make: 1
ten: 2
thousand: 3
dollars: 4
per: 5
week: 6
just: 7
surfing: 8
the: 9
web: 10
are: 11
you: 12
free: 13
for: 14
a: 15
meeting: 16
early: 17
next: 18
```

总共19个词汇量，我们创建一个19维的向量，得到如下结果：

```
i make ten thousand dollars per week just surfing the web! (spam)
-> [1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0]
are you free for a meeting early next week? (not spam)
-> [0 0 0 0 0 0 1 0 0 0 0 1 1 1 1 1 1 0]
```

接下来我们就可以使用分类模型来训练，预测标记垃圾邮件，并过滤垃圾邮件了。但是，有个很简单的方法来规避这种审查，如某封邮件如下：

```
ii mayke are you th0usands of free for a $$$s surf1ing teh webz meeting early next week
-> [0 0 0 0 0 0 1 0 0 0 0 1 1 1 1 1 1 0]
```

这封邮件里面包含了某些用户自己创造的单词，这些单词在我们的词汇表中没有，但是实际上我们依然可以识别出来，它是一封垃圾邮件。但是，用上述词袋模型转换的结果却是和前面第二封邮件类似的向量。显然，分类模型会把它归为正常邮件中。因此，上述特征工程显然不能满足要求。

除此之外，使用上述特征工程方法还有一个巨大的问题就是通常会创建非常高维的稀疏向量。假设我们有100万邮件作为训练集，每封邮件平均只有几十个单词，但词汇表可能有数十万，这样创建出来的输入数据是一个高维稀疏矩阵，这对很多机器学习算法来说并不是友好的输入。

如果使用上述的哈希特征方法，就可以将所有的原始数据转换成指定范围内的散列值。这样做有几个好处：

1. 即便对于不在词汇表中的单词，我们依然可以计算出一个散列值，因此不容易被规避，也不需要事先准备词汇表，新特征的转换对输入特征的长度不影响（因为事先已经定义好了散列范围）
2. 只需要散列新来的数据，并不需要重新对所有数据进行哈希处理，所以支持在线学习
3. 经过哈希特征工程之后，原来非常稀疏的向量可能会变得不那么稀疏
4. 尽管有散列冲突，但是研究和实践表明，这种影响很小。

哈希特征工程的比较大的缺点是缺乏可解释性，因为特征被处理成无法解释的散列值了。尽管如此，这个技巧才很多时候非常有用。

特征哈希的使用技巧

使用哈希特征的时候需要选择散列的范围，这个并没有统一的标准。较小的散列范围会导致较多的冲突，影响准确性，较大的范围会占用较高的内存和花费较多的训练时间。因此，在实际情况中，要根据你的目标选择，如果不考虑训练时间的话，可以考虑使用较大范围的散列结果。

Feature hashing(特征哈希)

原创 大师鲁 于 2018-06-11 20:04:58 发布 阅读量1w 收藏 9 点赞数 5

Feature hashing(特征哈希)

在机器学习中，特征哈希也称为哈希技巧（类比于核技巧），是一种快速且空间利用率高的特征 向量化 方法，即将任意特征转换为向量或矩阵中的索引。它通过对特征应用散列函数并直接使用特征的散列值作为索引来工作，而不是在关联数组中查找索引。

例子

在典型的文档分类任务中，机器学习算法（包括学习和分类）的输入是自由文本。 因此，构造了BOW表示：每个单词被抽取并计数，并且在训练集中的每个不同的单词都定义了训练集和测试集中每个文档的一个特征（独立变量）。
但是，机器学习算法通常用数值向量来定义。 因此一个文档集中的单词被认为是一个文档矩阵，其中每行是一个文档，每列是一个特征/单词; 在这个矩阵中的i,j项捕获了文档i中词汇表的第j项的频率（或权重）。根据Zipf定律，这些向量通常极其稀疏。
常用的方法是在训练时或在此之前构建训练集词汇表的字典表示，并用它将单词映射到索引。 例如这三个文档

- John likes to watch movies.
- Mary likes movies too.
- John also likes football.

可以使用字典转换成文档矩阵

Term	Index
John	1
likes	2
to	3
watch	4

Term	Index
movies	5
Mary	6
too	7
also	8
football	9

$$\begin{pmatrix} \text{John} & \text{likes} & \text{to} & \text{watch} & \text{movies} & \text{Mary} & \text{too} & \text{also} & \text{football} \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \end{pmatrix}$$

（和常见的文档分类和聚类一样，标点符号被删除了）
这个过程的问题在于，这些字典占用大量的存储空间并且规模随着训练集的增长而增长。 如果词典保持不变，对手可能会尝试发现不存在于存储词典中的新单词或拼写错误，来绕开机器学习的过滤器。 这就是为什么Yahoo!尝试使用特征哈希进行垃圾邮件过滤的原因。
哈希技巧并不局限于文本分类和类似文档的任务，也可以应用于任何涉及大规模（可能是无限的）特征的问题。

使用哈希技巧进行特征向量化

和维护一个字典不同，使用哈希技巧的特征向量化器可以把哈希函数应用于特征（例如单词）来构建一个预定义长度的向量，然后将哈希值直接用作特征索引并更新结果向量。

```
def hashing_vectorizer(features, N):
     x = N * [0]
     for f in features:
         h = hash(f)
         idx = h % N
         x[idx] += 1
     return x
```

因此，如果我们的特征向量是[“cat”,”dog”,”cat”], hash function 是，如果是“cat”，结果是1，如果是“dog”，结果是2。我们设定输出特征向量的维度(N)是4，那么输出就是[0,2,1,0]。建议使用第二个单比特输出的 hash function 来确定更新值的符号，以抵消hash冲突的影响。如果使用这样的hash函数，则该算法变为

```
def hashing_vectorizer(features, N):
     x = N * [0]
     for f in features:
         h = hash(f)
         idx = h % N
         if ξ(f) == 1:
             x[idx] += 1
         else:
             x[idx] -= 1
     return x
```

上面的代码实际上将每个样本转换为一个向量。 优化后的版本会生成一个对的流，并让学习和预测算法消耗这些流。

tips:

- 一般来说，我们使用向量的维数可能非常大，例如2^{25}或类似大小。
- 向量会非常稀疏，因此使用一种节省空间的格式进行存储，例如，一个向量存储不为0的值，另一个向量存储这些值所在的列号。
- 可以使用第二个hash函数，它返回+1或-1来决定是否对向量进行加减。这将使冲突最小化，使向量的维度发挥巨大的价值。

References:

Wikipedia Feature hashing

Feature Hashing for Large Scale Multitask Learning.

机器学习sklearn（十四）： 特征工程（五）特征编码（二）特征哈希(二) 转载

mob604756ef35df 2021-06-19 18:14:00

文章标签 ico 向量化 字符串 sed 散列函数 文章分类 机器学习 人工智能

特征哈希（相当于一种降维技巧）

类 FeatureHasher 是一种高速，低内存消耗的向量化方法，它使用了特征散列技术 ，或可称为 “散列法” （hashing trick）的技术。 代替在构建训练中遇到的特征的哈希表，如向量化所做的那样 FeatureHasher 将哈希函数应用于特征，以便直接在样本矩阵中确定它们的列索引。 结果是以牺牲可检测性为代价，提高速度和减少内存的使用; 哈希表不记得输入特性是什么样的，没有 inverse_transform 办法。

由于散列函数可能导致（不相关）特征之间的冲突，因此使用带符号散列函数，并且散列值的符号确定存储在特征的输出矩阵中的值的符号。 这样，碰撞可能会抵消而不是累积错误，并且任何输出要素的值的预期平均值为零。 默认情况下，此机制将使用 alternate_sign=True 启用，对于小型哈希表大小（ n_features < 10000 ）特别有用。 对于大的哈希表大小，可以禁用它，以便将输出传递给估计器，如 sklearn.naive_bayes.MultinomialNB 或 sklearn.feature_selection.chi2 特征选择器，这些特征选项器可以使用非负输入。

类 FeatureHasher 接受映射（如 Python 的 dict 及其在 collections 模块中的变体），使用键值对 (feature, value) 或字符串，具体取决于构造函数参数 input_type 。 映射被视为 (feature, value) 对的列表，而单个字符串的隐含值为1，因此 ['feat1', 'feat2', 'feat3'] 被解释为 [('feat1', 1), ('feat2', 1), ('feat3', 1)] 。 如果单个特征在样本中多次出现，相关值将被求和（所以 ('feat', 2) 和 ('feat', 3.5) 变为 ('feat', 5.5) ）。 FeatureHasher 的输出始终是 CSR 格式的 scipy.sparse 矩阵。

特征散列可以在文档分类中使用，但与 text.CountVectorizer 不同， FeatureHasher 不执行除 Unicode 或 UTF-8 编码之外的任何其他预处理; 请参阅下面的哈希技巧向量化大文本语料库，用于组合的 tokenizer/hasher。

例如，有一个词级别的自然语言处理任务，需要从 `(token, part_of_speech)` 键值对中提取特征。可以使用 Python 生成器函数来提取功能:

```
def token_features(token, part_of_speech):
    if token.isdigit():
        yield "numeric"
    else:
        yield "token={}".format(token.lower())
        yield "token,pos={},{}".format(token, part_of_speech)
    if token[0].isupper():
        yield "uppercase_initial"
    if token.isupper():
        yield "all_uppercase"
    yield "pos={}".format(part_of_speech)
```

然后， `raw_X` 为了可以传入 `FeatureHasher.transform` 可以通过如下方式构造:

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

并传入一个 hasher:

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

得到一个 `scipy.sparse` 类型的矩阵 `X` 。

注意使用发生器的理解，它将懒惰引入到特征提取中：词令牌（token）只能根据需要从哈希值进行处理。

实现细节

类 FeatureHasher 使用签名的 32-bit 变体的 MurmurHash3。 因此导致（并且由于限制 scipy.sparse ），当前支持的功能的最大数量 $2^{31} - 1$.
特征哈希的原始形式源于Weinberger et al，使用两个单独的哈希函数，h 和 ξ 分别确定特征的列索引和符号。 现有的实现是基于假设：MurmurHash3的符号位与其他位独立。
由于使用简单的模数将哈希函数转换为列索引，建议使用2次幂作为 n_features 参数; 否则特征不会均匀的分布到列中。

> 参考资料:
> - Kilian Weinberger, Anirban Dasgupta, John Langford, Alex Smola and Josh Attenberg (2009). 用于大规模多任务学习的特征散列. Proc. ICML.
> - MurmurHash3.

class sklearn.feature_extraction. FeatureHasher (*n_features=1048576*, *, *input_type='dict'*, *dtype=<class 'numpy.float64'>*, *alternate_sign=True*)
Implements feature hashing, aka the hashing trick.
This class turns sequences of symbolic feature names (strings) into scipy.sparse matrices, using a hash function to compute the matrix column corresponding to a name. The hash function employed is the signed 32-bit version of Murmurhash3.
Feature names of type byte string are used as-is. Unicode strings are converted to UTF-8 first, but no Unicode normalization is done. Feature values must be (finite) numbers.
This class is a low-memory alternative to DictVectorizer and CountVectorizer, intended for large-scale (online) learning and situations where memory is tight, e.g. when running prediction code on embedded devices.
Read more in the User Guide.
New in version 0.13.

Parameters

n_featuresint, default=220**

The number of features (columns) in the output matrices. Small numbers of features are likely to cause hash collisions, but large numbers will cause larger coefficient dimensions in linear learners.

input_type{“dict”, “pair”, “string”}, default=”dict”

Either “dict” (the default) to accept dictionaries over (feature_name, value); “pair” to accept pairs of (feature_name, value); or “string” to accept single strings. feature_name should be a string, while value should be a number. In the case of “string”, a value of 1 is implied. The feature_name is hashed to find the appropriate column for the feature. The value’s sign might be flipped in the output (but see non_negative, below).

dtypenumpy dtype, default=np.float64

The type of feature values. Passed to scipy.sparse matrix constructors as the dtype argument. Do not set this to bool, np.boolean or any unsigned integer type.

alternate_signbool, default=True

When True, an alternating sign is added to the features as to approximately conserve the inner product in the hashed space even for small n_features. This approach is similar to sparse random projection.

Changed in version 0.19: `alternate_sign` replaces the now deprecated `non_negative` parameter.

Examples

```
>>> from sklearn.feature_extraction import FeatureHasher
>>> h = FeatureHasher(n_features=10)
>>> D = [{'dog': 1, 'cat':2, 'elephant':4},{'dog': 2, 'run': 5}]
>>> f = h.transform(D)
>>> f.toarray()
array([[ 0.,  0., -4., -1.,  0.,  0.,  0.,  0.,  0.,  2.],
       [ 0.,  0.,  0., -2., -5.,  0.,  0.,  0.,  0.,  0.]])
```