基于 LSA 的文本索引技术初探

2021 年 1 月 19 日

1 引言

给定一个或若干个关键词 (terms)，在一组文章 (Documents) 中，寻找意义最接近这几个关键词的文章，就是文本索引问题. 当我们在各个网站的搜索栏键入短语或者关键词并且期待能够搜出想要的结果时，利用的正是文本检索的技术. 包括搜索引擎、各大社区网站，还包括电脑上的操作系统，都用到这项技术. 简而言之，有了它，我们能够从海量的信息中快速找到想要的.

2 概念

2.1 Term-Document-Matrix

设有一组文章，我们统计每一篇文章中，各个单词 (term) 出现的频数，并且，给每一篇文章和每一个不同的 term 标上序号，做完这些后，假设前后总共遇到 n 篇文章，以及 m 个不同的单词，那么我们可以得到一个 $n \times m$ 的矩阵

$$D = \begin{bmatrix} d_{1,1} & d_{1,2} & \cdots & d_{1,m} \\ d_{2,1} & d_{2,2} & \cdots & d_{2,m} \\ \vdots & \vdots & \ddots & \vdots \\ d_{n,1} & d_{n,2} & \cdots & d_{n,m} \end{bmatrix} \tag{2.1}$$

那么这个矩阵 D 的第 i 行第 j 列元素 $D[i,j]$ 正是在第 i 篇文章中，第 j 个 term 出现的频数.

2.2 TF-IDF

让矩阵 D 的每一个元素分别除以所在行的行和，就得到了 TF 矩阵，TF 也就是 Term Frequency 的意思，其实也就是对 D 的每一行做归一化处理，使得每一行加起来等于 1，从而可被称为频率.

所谓 IDF 就是 Inverse Document Frequency 的意思，我们可以这么样来理解，一个 term 如果它在每一篇文章中都出现，那么它本身所携带的信息量就很少，换句话说，看到这个 term 出现在一篇文章中并没有什么稀奇的，可是如果一个 term 出现得很少，突然在一篇文章当中被发现了，那还是有一点信息量的. 具体地，设 $\mathcal{D}$ 是文章的集合，设 t 是一个 term，我们定义

$$\mathrm{IDF}(t, \mathcal{D}) = \ln \frac{|\mathcal{D}|}{|\{d : t \in d, d \in \mathcal{D}\}|} \tag{2.2}$$

式中，分母部分表示：出现了 t 这个 term 的文章的数量，而分子表示总共有多少篇文章.

于是

$$\mathrm{TF\_IDF}(t, d, \mathcal{D}) = TF(t, d) \cdot IDF(t, \mathcal{D}) \tag{2.3}$$

就成为了 Term-Document-Matrix 的加强版本——因为它更准确地表达了「权重」.

向量空间模型 (Vector Space Model, VSM) 正是借助 TF-IDF 来对文章（或者广义上的文本数据集）来进行建模．算出来了 TF-IDF 矩阵，那么每一篇文章也就能用一组有序数对，也就是向量来表述，也就能够方便地在计算机中进行处理，并且用向量距离来衡量文章之间的相似性，这就是 VSM 的基本思想．

2.3 隐含层语义分析

在实际的场景中，Apple 既可以指代一种水果，又可以指代一家公司，由于 TF-IDF 仅仅考虑的是 term 自己的加权频率，所以它不具备区分 Apple 和 Apple 的能力，换句话说，如果仅仅看 TF-IDF 矩阵中的坐标，就会认为 Apple 和 Apple 是相似的．并且在实际的场景中，Car, Vehicle, Automobile 都指代汽车，可是从 TF-IDF 矩阵看来，Car 是 Car, Vehicle 是 Vehicle，所以在 TF-IDF 矩阵中，同义词无法从相似度上得到体现．

设 $n \times m$ 的矩阵 T 是从某组文章 $\mathcal{D}$ 计算得来的 TF-IDF 矩阵，则可以对它做奇异值分解，得到

$$T = U\Sigma V^\star \tag{2.4}$$

式中，U 是 $n \times r$ 酋矩阵，Σ 是 $r \times r$ 对角阵，V 是 $r \times n$ 酋矩阵，并且 $r = \min\{n, m\}$，所谓的酋矩阵是指自身与自身的共轭转置做矩阵乘法所得运算结果为单位阵的矩阵．现在考虑 T 的第 i 行，矩阵乘法的知识告诉我们

$$\begin{bmatrix} T[i,1] & T[i,2] & \cdots & T[i,m] \end{bmatrix} = \begin{bmatrix} U[i,1] & U[i,2] & \cdots & U[i,r] \end{bmatrix} \begin{bmatrix} d_1\boldsymbol{v}_1 \\ d_2\boldsymbol{v}_2 \\ \vdots \\ d_r\boldsymbol{v}_r \end{bmatrix} \tag{2.5}$$

如此看来，不仅矩阵 T 的第 i 行行向量可作为第 i 篇文章的「坐标」，矩阵 U 的第 i 行也可，差别在于，拿 U 的行向量做坐标，维数只有 r，而拿 T 的行向量做坐标，维数有 m，m 一般都是很大的，而 r 则要小得多，所以 LSA 首先可以有效降低计算成本，且不说 SVD 方法另外还有降维效果．其次，注意到最右边的那个列向量，我们知道做 SVD 得到的中间那个对角阵的对角元，其值一定可以是递减的，所以这里的 $d_1, d_2, \cdots, d_r$ 也是递减的，恰好，将 $\boldsymbol{v}_1, \boldsymbol{v}_2, \cdots, \boldsymbol{v}_3$ 视作一组基，$d_1, d_2, \cdots, d_r$ 则衡量了各自的「重要程度」；并且，每一个 $\boldsymbol{v}_i$ 是 m 维的，可以看做是 m 个 term 的坐标的一个组合，这样一来，每篇文章不再由一个个孤立的 term 的加权词频 (TF-IDF) 来表示，而是用特征空间中的坐标来表示，由此 LSA 开始具有了一定的抽象能力．

3 实现

3.1 分词

若是不做分词呢，一篇文章在内存中就是一个字符串，就是一个字符数组，不是说一定无法从中分析出什么，而是如果把一篇文章看做是字符 (char) 的数组的话，字符太多了，或者说维数太多了，可以想想，一篇文章得有多少个字啊？分词操作，实际上就是「断句」加「数 (shǔ) 数 (shù)」，断句就是将字符串切成一个个的单词 (terms)，而数数就是数每个 term 都出现了多少次，当然了，更加高级的分词软件还具备词性标注功能，不过我们暂时不用到．做了分词之后呢，一篇文章如果表成一个向量，每个分量是每个单词出现的个数，那么这个向量的长度，肯定要比原来字符串的长度短很多，所以，分词首先有效地降低了数据的维数．

我们采用开源项目 lancopku/pkuseg-python[1] 所提供的 Python 软件包来对文章 (document) 和查询输入 (input query) 进行分词，在具体实现中，我们用一个 Python 数组存储每篇文章的文章名，用另一个 Python 数组存储每篇文章的单词列表形式．

3.2 行稀疏矩阵介绍

一个行稀疏矩阵用三个变量来确定, 分别是索引指针 `indptr`, 索引列表 `indices`, 以及数据 `data`, 行标号 `i` 从 `0` 开始,对于原始矩阵的第 `i` 行,Python 数组切片 `indices[indptr[i]:indptr[i+1]]` 表示列标号, 而 `indices[indptr[i]:indptr[i+1]]` 表示第 `i` 行各列对应的元素. 下面, 我们来进行示范:

```
indptr = [0, 2, 5]
indices = [0, 4, 3, 2, 5]
data = [6, 5, 2, 3, 5]
```

那么原矩阵的第 `0` 行是多少呢? 首先我们要知道第 `0` 行哪些列非空, 答案是 `indices[0:2]` 非空, 也就是 `[0, 4]` 非空, 而相应的数据就是 `[6, 5]`, 这下我们知道了: 第 `0` 行的第 `0` 列和第 `4` 列的值分别是 `[6, 5]`, 所以原矩阵的第 `0` 行是:

```
[6, 0, 0, 0, 5]
```

再来看第 `1` 行,实际上是第二行,只不过标号是 `1`,它有哪些字段有值呢?它有值的位置是 `indices[2:5]`, 也就是 `[3, 2, 5]`, 对应的数据是多少呢? 对应的数据是 `data[2:5]`, 也就是 `[2, 3, 5]`, 那么这一行的值就是

```
[0, 0, 3, 2, 0, 5]
```

第一行和第二行的「长度」不一致, 因为是稀疏矩阵表示, 所以没有表示的就是 `0`, 所以两行拼在一起就得到原矩阵:

```
[
    [6, 0, 0, 0, 5, 0],
    [0, 0, 3, 2, 0, 5]
]
```

也有可能遇到 `indices` 有重复值的情况, 例如

```
indptr = [0:5]
indices = [1, 1, 1, 2, 2]
data = [1, 1, 1, 1, 1]
```

那么我们从这些数据中知道, 原矩阵只有一行, 并且第一行位置 `1` 有三个 `1`, 所以第一行位置 `1` 的值就是这些值加起来, 就是 `3`, 类似的第一行位置 `2` 的值是 `2`:

```
[[0, 3, 2]]
```

行稀疏矩阵用来在计算机中表示和存储 Term-Document-Matrix 非常有用, 那是因为人类的各种语言中的词汇量都非常多, 而每一篇文章一般只用到其中的很少一部分词汇.

3.3 Term-Document-Matrix

假设 `articles` 是一个二阶列表, 它的第 `i` 个元素也就是 `articles[i]` 存储着编号为 `i` 的文章的单词, 那么 Term-Document-Matrix 将会被构建成为行稀疏矩阵的形式, `indptr`, `indices`, 与 `data` 这三个变量与上文中的含义一样, 用来表示这个 Term-Document-Matrix 的行稀疏矩阵:

```
indptr = [0]
indices = list()
data = list()
vocabulary = dict()

```

```
for i in range(len(articles)):
    terms = articles[i]
    for term in terms:
        term_index = 0
        if term in vocabulary:
            term_index = vocabulary[term]
        else:
            term_index = len(vocabulary)
            vocabulary[term] = term_index

        indices.append(term_index)
        data.append(1)

    indptr.append(len(indices))
```

对照着§ 3.2对行稀疏矩阵的介绍，我们能够理解上列代码.

3.4 TF-IDF

首先是计算 TF 矩阵，TF 矩阵的第 i 行第 j 列的值就是第 j 个 term 在第 i 篇文章中出现的频数除以这篇文章总共的单词数（当然包括重复出现的），设 `doc_matrix` 是 Term-Document-Matrix, 那么 TF 矩阵是这样计算的

```
tf = scipy.divide(doc_matrix, scipy.sum(doc_matrix, axis=1))
```

IDF 矩阵也可以计算，它的第 i 行第 j 列表示第 j 个 term 的 IDF 值:

```
# (n_x[0], n_y[0]), (n_x[1], n_y[2]), ... 表示非零元素坐标
n_x, n_y = doc_matrix.nonzero()

# 去重
s = set()
for j in range(len(n_x)):
    s.add((n_x[j], n_y[j],))

n_y = list()
for x, y in s:
    n_y.append(y)

col_indexes, non_zeros_count = np.unique(n_y, return_counts=True)
idf = np.log(doc_matrix.shape[0]/non_zeros_count)
```

那么，TF-IDF 矩阵就是 TF 矩阵和 IDF 矩阵按元素相乘了:

```
tf_idf = tf.multiply(idf)
```

然后我们会在 TF-IDF 矩阵的基础上做进一步分析.

3.5 LSA 的实现

设 T 是 TF-IDF 矩阵，它和 Term-Document-Matrix 的行数、列数都相同，设 T 是 n 行 m 列的，那么这里 n 就是文章的个数，m 是所有不同的 term 的个数，对于一般的情形，m 远大于 n，对于大数据情形，n 和 m 都非常大，这里 m 可能少则一两万，多则数万. 在 GPU 上已经有人实现了高效的 SVD

算法，由此我们可以认为 SVD 是快的，于是在合理的时长内，我们能够找到 $n \times r$ 酋矩阵 U，$r \times r$ 酋矩阵 Σ，以及 $n \times r$ 酋矩阵 V，使得

$$T = U\Sigma V^\star \tag{3.1}$$

式中，$r = \min\{m, n\}$．设 U_k 表示 U 只取前 k 列，设 Σ_k 表示 Σ 只取前 k 行前 k 列，设 V_k 表示 V 只取前 r 行，在某些较好的情况下，不大的 k 就能取得较好的近似效果，表示为：

$$T \approx U_k\Sigma_k(V_k)^\star \stackrel{\text{def}}{=} T_k \tag{3.2}$$

当然，如果 k 和 r 再接近些，近似效果还有好，不过提升就不那么明显了．这里我们说的主要是 SVD，可见 LSA 主要是基于 SVD 的．

设 $\boldsymbol{t}_i$ 是 T 的第 i 行的行向量，它还表示第 i 篇文章的 TF-IDF 坐标，并且我们知道 $\boldsymbol{t}_i$ 是 m 维的，m 可能有好几万．设 $\boldsymbol{s}_i$ 是矩阵 $U_k\Sigma_k$ 的行向量，那这样一来，我们有：

$$\boldsymbol{t}_i = \boldsymbol{s}_i(V_k)^\star \tag{3.3}$$

利用酋矩阵的性质，假设 V 是实的，那么

$$\boldsymbol{t}_iV_k = \boldsymbol{s}_i \tag{3.4}$$

式 3.4其实揭示了，当得到了 TF-IDF 矩阵的一行之后，怎样计算出 $U_k\Sigma_k$ 矩阵对应的那一行．这也是搜索的原理：当收到一个查询请求，就把查询字符串做分词，然后把这个查询看做是一篇「文章」也就是一个 Document，去计算 Term-Document-Matrix 新的一行，得到 $\boldsymbol{t}_{n+1}$，然后利用式式 3.4去计算 $\boldsymbol{s}_{n+1}$，然后在 $\boldsymbol{s}_i, i = 1, 2, \cdots, n$ 这 n 个坐标中，找出与 $\boldsymbol{s}_{n+1}$ 余弦相似度最高的那几个．

下面是实现的节选，注意 $V^\star$ 用 vh 表示：

```
# v^\star 用 vh 表示
# 满足 u @ np.diag(s) * vh = tf_idf
u, s, vh = svds(tf_idf, k = k)

doc_coords = u @ np.diag(s)

# feature_coord 就是 \boldsymbol{s}_{n+1}
def to_feature_coord(origin_coord):

    origin_coord = np.reshape(
        origin_coord,
        newshape=(1, vh.T.shape[0],)
    )

    feature_coord = np.matmul(origin_coord, vh.T)

    return feature_coord[0, :]

# 寻找与 input_feature 余弦相似度最高的
def find_nearest(input_feature: np.ndarray):

    cosine_values = list()

    for d in doc_coords:
        cos_value = cos_of_two_vector(d, input_feature)
        cosine_values.append(cos_value)
```

```
27
28      indexes = list(range(len(doc_coords)))
29      indexes.sort(
30          reverse = True,
31          key = lambda i: cosine_values[i]
32      )
33
34      cosine_values = list(map(lambda i: cosine_values[i], indexes))
35
36      return ( indexes, cosine_values, )
```

那么函数 `find_nearest` 给出的 `indexes` 就是余弦相似度 (和 `input_feature`) 逐渐递减的下标列表. 比如说返回的是:

```
indexes = [3, 2, 0, 1]
```

那么我就知道标号为 3 的文章最可能是我要找的, 标号为 2 的文章对于我的搜索的匹配程度又低一些.

要查看完整的代码请点击这里.

4 总结

首先通过分词, 使词向量的维数大幅降低, 然后通过计算 TF-IDF, 使得权重更为准确, 然后通过对 TF-IDF 矩阵做奇异值分解, 使得比较两个高维向量以及计算两个高维向量之间的相似性成为可能, 降维技术的应用可真广泛.

参考文献

[1] Ruixuan Luo, Jingjing Xu, Yi Zhang, Xuancheng Ren, and Xu Sun. Pkuseg: A toolkit for multi-domain chinese word segmentation. *CoRR*, abs/1906.11455, 2019.