TF-IDF演算法解析與Python實現方法詳解

TF-IDF（term frequency–inverse document frequency）是一種用於資訊檢索（information retrieval）與文字挖掘（text mining）的常用加權技術。比較容易理解的一個應用場景是當我們手頭有一些文章時，我們希望計算機能夠自動地進行關鍵詞提取。而TF-IDF就是可以幫我們完成這項任務的一種統計方法。它能夠用於評估一個詞語對於一個文集或一個語料庫中的其中一份文件的重要程度。

在一份給定的檔案裡，詞頻 (term frequency, TF) 指的是某一個給定的詞語在該檔案中出現的次數。這個數字通常會被歸一化（分子一般小於分母 區別於IDF），以防止它偏向長的檔案。（同一個詞語在長檔案裡可能會比短檔案有更高的詞頻，而不管該詞語重要與否。）

逆向檔案頻率 (inverse document frequency, IDF) 是一個詞語普遍重要性的度量。某一特定詞語的IDF，可以由總檔案數目除以包含該詞語之檔案的數目，再將得到的商取對數得到。

某一特定檔案內的高詞語頻率，以及該詞語在整個檔案集合中的低檔案頻率，可以產生出高權重的TF-IDF。因此，TF-IDF傾向於過濾掉常見的詞語，保留重要的詞語。

TFIDF的主要思想是：如果某個詞或短語在一篇文章中出現的頻率TF高，並且在其他文章中很少出現，則認為此詞或者短語具有很好的類別區分能力，適合用來分類。TFIDF實際上是：TF * IDF，TF詞頻(Term Frequency)，IDF反文件頻率(Inverse Document Frequency)。TF表示詞條在文件d中出現的頻率（另一說：TF詞頻(Term Frequency)指的是某一個給定的詞語在該檔案中出現的次數）。IDF的主要思想是：如果包含詞條t的文件越少，也就是n越小，IDF越大，則說明詞條t具有很好的類別區分能力。如果某一類文件C中包含詞條t的文件數為m，而其它類包含t的文件總數為k，顯然所有包含t的文件數n=m k，當m大的時候，n也大，按照IDF公式得到的IDF的值會小，就說明該詞條t類別區分能力不強。（另一說：IDF反文件頻率(Inverse Document Frequency)是指果包含詞條的文件越少，IDF越大，則說明詞條具有很好的類別區分能力。）但是實際上，如果一個詞條在一個類的文件中頻繁出現，則說明該詞條能夠很好代表這個類的文字的特徵，這樣的詞條應該給它們賦予較高的權重，並選來作為該類文字的特徵詞以區別與其它類文件。這就是IDF的不足之處.

為了演示在Python中實現TF-IDF的方法，一些基於自然語言處理的預處理過程也會在本文中出現。如果你對NLTK和Scikit-Learn兩個庫還很陌生可以參考如下文章：

Python程式設計使用NLTK進行自然語言處理詳解

Python自然語言處理之詞幹,詞形與最大匹配演算法程式碼詳解

必要的預處理過程

首先，我們給出需要引用的各種包，以及用作處理物件的三段文字。

```
import nltk
import math
import string
from nltk.corpus import stopwords
from collections import Counter
from nltk.stem.porter import *
from sklearn.feature_extraction.text import TfidfVectorizer
text1 = "Python is a 2000 made-for-TV horror movie directed by Richard \
Clabaugh. The film features several cult favorite actors, including William \
Zabka of The Karate Kid fame, Wil Wheaton, Casper Van Dien, Jenny McCarthy, \
Keith Coogan, Robert Englund (best known for his role as Freddy Krueger in the \
A Nightmare on Elm Street series of films), Dana Barron, David Bowe, and Sean \
Whalen. The film concerns a genetically engineered snake, a python, that \
escapes and unleashes itself on a small town. It includes the classic final\
girl scenario evident in films like Friday the 13th. It was filmed in Los Angeles, \
California and Malibu, California. Python was followed by two sequels: Python \
II (2002) and Boa vs. Python (2004), both also made-for-TV films."
text2 = "Python, from the Greek word (πύθων/πύθωνας), is a genus of \
nonvenomous pythons[2] found in Africa and Asia. Currently, 7 species are \
recognised.[2] A member of this genus, P. reticulatus, is among the longest \
snakes known."
text3 = "The Colt Python is a .357 Magnum caliber revolver formerly \
manufactured by Colt's Manufacturing Company of Hartford, Connecticut. \
It is sometimes referred to as a \"Combat Magnum\".[1] It was first introduced \
in 1955, the same year as Smith & Wesson's M29 .44 Magnum. The now discontinued \
Colt Python targeted the premium revolver market segment. Some firearm \
collectors and writers such as Jeff Cooper, Ian V. Hogg, Chuck Hawks, Leroy \
Thompson, Renee Smeets and Martin Dougherty have described the Python as the \
finest production revolver ever made."
```

TF-IDF的基本思想是：詞語的重要性與它在檔案中出現的次數成正比，但同時會隨著它在語料庫中出現的頻率成反比下降。 但無論如何，統計每個單詞在文件中出現的次數是必要的操作。所以說，TF-IDF也是一種基於 bag-of-word 的方法。

首先我們來做分詞，其中比較值得注意的地方是我們設法剔除了其中的標點符號（顯然，標點符號不應該成為最終的關鍵詞）。

```
def get_tokens(text):
lowers = text.lower()
#remove the punctuation using the character deletion step of translate
remove_punctuation_map = dict((ord(char), None) for char in string.punctuation)
no_punctuation = lowers.translate(remove_punctuation_map)
tokens = nltk.word_tokenize(no_punctuation)
return tokens
```

下面的程式碼用於測試上述分詞結果，Counter() 函式用於統計每個單詞出現的次數。

```
tokens = get_tokens(text1)
count = Counter(tokens)
print (count.most_common(10))
```

執行上述程式碼後可以得到如下結果，我們輸出了其中出現次數最多的10個詞。

```
[('the', 6), ('python', 5), ('a', 5), ('and', 4), ('films', 3), ('in', 3),
('madefortv', 2), ('on', 2), ('by', 2), ('was', 2)]
```

顯然，像 the, a, and 這些詞儘管出現的次數很多，但是它們與文件所表述的主題是無關的，所以我們還要去除“詞袋”中的“停詞”（stop words），程式碼如下：

```
def stem_tokens(tokens, stemmer):
stemmed = []
for item in tokens:
stemmed.append(stemmer.stem(item))
return stemmed
```

同樣，我們來測試一下上述程式碼的執行效果。

```
tokens = get_tokens(text1)
filtered = [w for w in tokens if not w in stopwords.words('english')]
count = Counter(filtered)
print (count.most_common(10))
```

從下面的輸出結果你會發現，之前那些缺乏實際意義的 the, a, and 等詞已經被過濾掉了。

```
[('python', 5), ('films', 3), ('film', 2), ('california', 2), ('madefortv', 2),
('genetically', 1), ('horror', 1), ('krueger', 1), ('filmed', 1), ('sean', 1)]
```

但這個結果還是不太理想，像 films, film, filmed 其實都可以看出是 film，而不應該把每個詞型都分別進行統計。這時就需要要用到我們在前面文章中曾經介紹過的 Stemming 方法。程式碼如下：

```
tokens = get_tokens(text1)
filtered = [w for w in tokens if not w in stopwords.words('english')]
stemmer = PorterStemmer()
stemmed = stem_tokens(filtered, stemmer)
```

類似地，我們輸出計數排在前10的詞彙（以及它們出現的次數）：

```
count = Counter(stemmed)
print(count)
```

上述程式碼執行結果如下：

```
Counter({'film': 6, 'python': 5, 'madefortv': 2, 'california': 2, 'includ': 2, '2004': 1,
'role': 1, 'casper': 1, 'robert': 1, 'sequel': 1, 'two': 1, 'krueger': 1,
'ii': 1, 'sean': 1, 'lo': 1, 'clabaugh': 1, 'finalgirl': 1, 'wheaton': 1,
'concern': 1, 'whalen': 1, 'cult': 1, 'boa': 1, 'mccarthi': 1, 'englund': 1,
'best': 1, 'direct': 1, 'known': 1, 'favorit': 1, 'movi': 1, 'keith': 1,
'karat': 1, 'small': 1, 'classic': 1, 'coogan': 1, 'like': 1, 'elm': 1,
'fame': 1, 'malibu': 1, 'sever': 1, 'richard': 1, 'scenario': 1, 'town': 1,
'friday': 1, 'david': 1, 'unleash': 1, 'vs': 1, '2000': 1, 'angel': 1, 'nightmar': 1,
'zabka': 1, '13th': 1, 'jenni': 1, 'seri': 1, 'horror': 1, 'william': 1,
'street': 1, 'wil': 1, 'escap': 1, 'van': 1, 'snake': 1, 'evid': 1, 'freddi': 1,
'bow': 1, 'dien': 1, 'follow': 1, 'engin': 1, 'also': 1})
```

至此，我們就完成了基本的預處理過程。

TF-IDF的演算法原理

預處理過程中，我們已經把停詞都過濾掉了。如果只考慮剩下的有實際意義的詞，前我們已經講過，顯然詞頻（TF，Term Frequency）較高的詞之於一篇文章來說可能是更為重要的詞（也就是潛在的關鍵詞）。但這樣又會遇到了另一個問題，我們可能發現在上面例子中，madefortv、california、includ 都出現了2次（madefortv其實是原文中的made-for-TV，因為我們所選分詞法的緣故，它被當做是一個詞來看待），但這顯然並不意味著“作為關鍵詞，它們的重要性是等同的”。

因為”includ”是很常見的詞（注意 includ 是 include 的詞幹）。相比之下，california 可能並不那麼常見。如果這兩個詞在一篇文章的出現次數一樣多，我們有理由認為，california 重要程度要大於 include ，也就是說，在關鍵詞排序上面，california 應該排在 include 的前面。

於是，我們需要一個重要性權值調整引數，來衡量一個詞是不是常見詞。如果某個詞比較少見，但是它在某篇文章中多次出現，那麼它很可能就反映了這篇文章的特性，它就更有可能揭示這篇文字的話題所在。這個權重調整引數就是“逆文件頻率”（IDF，Inverse Document Frequency），它的大小與一個詞的常見程度成反比。

知道了 TF 和 IDF 以後，將這兩個值相乘，就得到了一個詞的TF-IDF值。某個詞對文章的重要性越高，它的TF-IDF值就越大。如果用公式來表示，則對於某個特定檔案中的詞語 ti 而言，它的 TF 可以表示為：

其中 $n_{i,j}$是該詞在檔案 d_j中出現的次數，而分母則是檔案 d_j 中所有詞彙出現的次數總和。如果用更直白的表達是來描述就是，

某一特定詞語的IDF，可以由總檔案數目除以包含該詞語之檔案的數目，再將得到的商取對數即可：

其中，|D| 是語料庫中的檔案總數。 $|\{j:t_i \in d_j\}|$ 表示包含詞語 ti 的檔案數目（即 $n_{i,j} \neq 0$ 的檔案數目）。如果該詞語不在語料庫中，就會導致分母為零，因此一般情況下使用 $1 + |\{j:t_i \in d_j\}|$。同樣，如果用更直白的語言表示就是

最後，便可以來計算 TF-IDF(t)=TF(t)×IDF(t)。
下面的程式碼實現了計算TF-IDF值的功能。

```
def tf(word, count):
return count[word] / sum(count.values())
def n_containing(word, count_list):
return sum(1 for count in count_list if word in count)
def idf(word, count_list):
return math.log(len(count_list) / (1   n_containing(word, count_list)))
def tfidf(word, count, count_list):
return tf(word, count) * idf(word, count_list)
```

再給出一段測試程式碼：

```
countlist = [count1, count2, count3]
for i, count in enumerate(countlist):
print("Top words in document {}".format(i   1))
scores = {word: tfidf(word, count, countlist) for word in count}
sorted_words = sorted(scores.items(), key=lambda x: x[1], reverse=True)
for word, score in sorted_words[:3]:
print("\tWord: {}, TF-IDF: {}".format(word, round(score, 5)))
```

輸出結果如下：

```
Top words in document 1
Word: film, TF-IDF: 0.02829
Word: madefortv, TF-IDF: 0.00943
Word: california, TF-IDF: 0.00943
Top words in document 2
Word: genu, TF-IDF: 0.03686
Word: 7, TF-IDF: 0.01843
Word: among, TF-IDF: 0.01843
Top words in document 3
Word: revolv, TF-IDF: 0.02097
Word: colt, TF-IDF: 0.02097
Word: manufactur, TF-IDF: 0.01398
```

利用Scikit-Learn實現的TF-IDF

因為 TF-IDF 在文字資料探勘時十分常用，所以在Python的機器學習包中也提供了內建的TF-IDF實現。主要使用的函式就是TfidfVectorizer()，來看一個簡單的例子。

```
>>> corpus = ['This is the first document.',
'This is the second second document.',
'And the third one.',
'Is this the first document?',]
>>> vectorizer = TfidfVectorizer(min_df=1)
>>> vectorizer.fit_transform(corpus)
<4x9 sparse matrix of type '<class 'numpy.float64'>'
with 19 stored elements in Compressed Sparse Row format>
>>> vectorizer.get_feature_names()
['and', 'document', 'first', 'is', 'one', 'second', 'the', 'third', 'this']
>>> vectorizer.fit_transform(corpus).toarray()
array([[ 0.    , 0.43877674, 0.54197657, 0.43877674, 0.    ,
0.    , 0.35872874, 0.    , 0.43877674],
[ 0.    , 0.27230147, 0.    , 0.27230147, 0.    ,
0.85322574, 0.22262429, 0.    , 0.27230147],
[ 0.55280532, 0.    , 0.    , 0.    , 0.55280532,
0.    , 0.28847675, 0.55280532, 0.    ],
[ 0.    , 0.43877674, 0.54197657, 0.43877674, 0.    ,
0.    , 0.35872874, 0.    , 0.43877674]])
```

最終的結果是一個 4×9 矩陣。每行表示一個文件，每列表示該文件中的每個詞的評分。如果某個詞沒有出現在該文件中，則相應位置就為 0 。數字 9 表示語料庫裡詞彙表中一共有 9 個（不同的）詞。例如，你可以看到在文件1中，並沒有出現 and，所以矩陣第一行第一列的值為 0 。單詞 first 只在文件1中出現過，所以第一行中 first 這個詞的權重較高。而 document 和 this 在 3 個文件中出現過，所以它們的權重較低。而 the 在 4 個文件中出現過，所以它的權重最低。

最後需要說明的是，由於函式 TfidfVectorizer() 有很多引數，我們這裡僅僅採用了預設的形式，所以輸出的結果可能與採用前面介紹的（最基本最原始的）演算法所得出之結果有所差異（但數量的大小關係並不會改變）。有興趣的讀者可以參考這裡來了解更多關於在Scikit-Learn中執行 TF-IDF 演算法的細節。

總結