TF – IDF 방법을 이용한 텍스트 검색 구현

2013130874
영어영문학과
한석희

1 & 2. 구현 방법 + 코드 (적용 원리 및 구현 방식)

과정을 두괄식으로 나타내자면,

#1. TF 값 계산을 위한 다큐먼트: 텀프리퀀시 사전생성

```
# 1. import modules

import nltk
from nltk.corpus import stopwords
from nltk.stem.snowball import SnowballStemmer
import collections

# 2. Iterate over 60 files and form a term frequency dictionary for each documents

    # 2-1). make an empty dict doc to term-freq counter
doc2tf = {}

    # 2-2). format : data/
for i in range(60):

    file_path = 'data/{}.txt'.format(i)

    # save text name for dict key
    text_name = '{}.txt'.format(i)

    # process text into freq counter

        # [1]. prepocess tokens - tokenize, remove stopwords, stem => token lists
    text = open(file_path).read()
    text = text.lower()
    tokens = nltk.word_tokenize(text)

    stop = set(stopwords.words('english'))
    tokens = [i for i in tokens if i not in stop]

    snowball_stemmer = SnowballStemmer('english')
    tokens = [snowball_stemmer.stem(i) for i in tokens]

        # [2]. form a counter dictionary and match it with doc name
    count_obj = collections.Counter(tokens)

        # [3]. put it in a doc2tf dict
    doc2tf[text_name] = count_obj
```

ex). doc : term-freq 사전

{
다큐먼트 : {term: freq, term2: freq, …},
다큐먼트 2 : {term: freq, term2: freq, …},
…
}

#2. IDF 값 계산을 위한 {term: 해당 term 이 언급된 다큐먼트 리스트}생성

```
# import modules
from collections import defaultdict

# make empty term2doc_list dict.
term2doc = defaultdict(list)

# iterate over 60 docs make docs list for each terms
for i in range(60):

    doc_name = '{}.txt'.format(i)
    term_set = set(doc2tf[doc_name])

    for term in term_set:
        term2doc[term].append(doc_name)
```

ex). term : doc_list 사전
{
'mill' : ['0.txt',
'2.txt',
'9.txt',
'13.txt',
'18.txt',
'20.txt'],
…
}

#3. Inverted vector 의 row 가 되는 단어들 리스트 생성

4820 개의 토큰이 inverted matrix 의 row 를 형성

```
# 1. 도큐먼트 별 inverted vector를 만들때 사용할 row단어 순서 생성

dim_terms = list(term2doc.keys())
print(dim_terms)
print(len(dim_terms))
```

```
['follow', 'deni', 'trend', 'speak', 'miss', 'attack', 'enough', 'perfect', 'got',
onth', 'frantic', 'accord', 'media', 'feminist', 'tuesday', 'noth', 'all-too-famil
'hijack', 'supermodel', 'miley', 'barb', '!', 'mtv', 'awards.mind', 'retweet', 'tw
aylor', 'hip-hop', 'first', 'movi', 'jump', 'interact', 'top', 'user', 'wide', 'aw
'point', 'person', 'least', 'like', 'photo', 'rememb', 'turn', 'problem', 'want',
'minaj.it', 'racial', 'year', 'prais', 'middl', 'beef', 'thousand', 'taken', 'conn
'show', 'insensit', 'snub', 'launch', 'dig', 'wear it', 'stage', 'tell', 're', 'ra
```

```
4842
```

#4. document: inverted vector 를 생성하여 도큐먼트 별로 inverted matrix 내에서 도큐먼트에 해당하는 벡터를 사전으로 맵핑 (tf*idf 값 계산)

```
# ex). dict key '1.txt'
# ex). [tf.idf(d,t), ] ____ 4840 in dim_terms order.
import math
import numpy as np

# make empty inverted vector dictionary for each doc
doc_inv_vec = defaultdict(list)

for i in range(60):

    doc = '{}.txt'.format(i)

    for term in dim_terms:

        weight = ( math.log10(doc2tf[doc][term]+1) ) * ( math.log10( 60 / (len(term2doc[term])) ))

        doc_inv_vec[doc].append(weight)
```

```
: 1 doc_inv_vec['1.txt']
```

```
: [0.0,
   0.0,
   0.0,
   0.0,
   0.0,
   0.0,
   0.0,
   0.0,
   0.1999465006780432,
   0.0,
   0.0,
   0.0,
   0.2078953618263567,
   0.0,
   0.0,
   0.0,
   0.0,
   0.0,
```

#5. 함수 생성 :

input : query string
output : top-5-cosine-similarity text names

쿼리문을 받아 ->
전처리(tokenizing, stop-words removal, stemming) 하고 ->
쿼리를 tfd-id 벡터화하여,
그것과 도큐먼트들의 코사인 유사도를 계산한 후,
정렬하여 높은 유사도를 가지는 상위 5 개의
(상위 도큐먼트 이름, 유사도) 튜플을 리턴하는 함수

```
def query_five_docs_1(query):

    # make empty list (doc, cos_sim)
    doc_cossim = []

    # tokenize, stem query
    query = query.lower()
    query_tokens = nltk.word_tokenize(query)

    stop = set(stopwords.words('english'))
    query_tokens = [i for i in query_tokens if i not in stop]

    snowball_stemmer = SnowballStemmer('english')
    query_tokens = [snowball_stemmer.stem(i) for i in query_tokens]
    query_counter = collections.Counter(query_tokens)

    # make query tf-idf inverted vector

    query_inv_vec = [(( math.log10(query_counter[term]+1) ) * ( math.log10( 60 / (len(term2doc[term])) )))
                     if term in query_tokens else 0 for term in dim_terms]

    # normalize the query-tf-idf-vector

    query_inv_vec = np.asarray(query_inv_vec, dtype=np.float)
    query_vec = normalize(query_inv_vec[:,np.newaxis], axis=0).ravel()

    # iterate over other 60 doc vectors
    for k in range(60):

        # normalize 60 vectors
        doc_name = '{}.txt'.format(k)
        doc_vec = np.asarray(doc_inv_vec[doc_name], dtype=np.float)
        doc_vec = normalize(doc_vec[:,np.newaxis], axis=0).ravel()

        # calculate cosine similarity between query-tf-idf vector and the documents-tf-idf vector
        cos_sim = dot(query_vec, doc_vec)/(norm(query_vec)*norm(doc_vec))

        # append (doc_name, cossim) to doc_cossim
        doc_cossim.append((doc_name, cos_sim))

    # sort doc_cossim by 2nd element of tuples
    doc_cossim = list(doc_cossim)
    doc_cossim = sorted(doc_cossim, key=lambda x: x[1], reverse = True)

    # return the front 5 with doc names
    return [tuple for tuple in doc_cossim[:5]]
```

6. 위의 함수의 일부분을 변형하여 쿼리를 tf*idf vector 화 하는 방법 이외에, 쿼리를 one-hot vector 인 채로 유사도를 계산한 것의 performance 를 비교

```
    query_tokens = [snowball_stemmer.stem(i) for i in query_tokens]

    # make query as one-hot vector normalize and make it into numpy vector

    query_inv_vec = [1 if term in query_tokens else 0 for term in dim_terms]
```

=> query 쿼리를 원핫벡터화 하고 (추후에 정규화하여 사용)

3. 실행 결과

쿼리를 tf*idf vector 화 했을 때, 코사인 유사도가 높은 도큐먼트는 다음과 같이 다섯개로 나타났다.

[('54.txt', 0.13722324592747573),
 ('46.txt', 0.09101079348476551),
 ('58.txt', 0.08296495035560224),
 ('48.txt', 0.07407843089173001),
 ('50.txt', 0.06678607218427488)]

=> 54, 46, 58, 48, 50

반면, 쿼리를 원핫벡터화하여 사용할 경우 상위 5 개 도큐먼트는 다음과 같았다.

[('54.txt', 0.14139883628916960),
 ('58.txt', 0.09126754306628192),
 ('46.txt', 0.08423511884791456),
 ('48.txt', 0.07018769377562273),
 ('49.txt', 0.06454224270594335)]

=> 54, 58, 46, 48, 49

반면, 쿼리를 tf*idf 화 했을 때는, obama 의 idf 스코어가 상대적 희소성으로 인해 높게 산출되어 query 의 inverted matrix vector 에서
obama 의 weight 가 높게 산출되어,
[표 1] 에서 보는 바와 같이, obama 의 언급수가 높은 도큐먼트가 상위 유사도 도큐먼트로 산출된다는 것을 알 수 있다.

	Tf - idf	One hot
RANK_1	46	58
P* / O*	2 / 10	8 / 1
RANK_5	50	49
P / O	5 / 8	9 / 2

[표 1]

*P : president / *O : obama