03.01.03 sklearn 문서 전처리 기능

1. BOW (Bag of Words)

- corpus : 전체 문서 (d1,d2,d3, ...)
- corpus 를 구성하는 고정된 단어장 생성 (단어장 내 단어 : t1, t2, t3, ...)

$X_{i,j}$ = 문서 d_i 내 단어 t_j의 출현 빈도

2. sklearn 문서 전처리 패키지 및 클래스

Scikit-Learn의 feature_extraction 서브패키지와 feature_extraction.text 서브패키지는 다음과 같은 문서 전처리용 클래스를 제공한다.

1. DictVectorizer:
각 단어의 수를 세어놓은 사전에서 BOW 인코딩 벡터를 만든다.

2. CountVectorizer:
문서 집합에서 단어 토큰을 생성하고 각 단어의 수를 세어 BOW 인코딩 벡터를 만든다.

*단어장 만들어놓고, count해서 BOW 만드는 방식은 단어장이 커지면 실제로 사용하기 어렵다는 단점
 => HashingVectorizer 사용

3. TfidfVectorizer:
CountVectorizer와 비슷하지만 TF-IDF 방식으로 단어의 가중치를 조정한 BOW 인코딩 벡터를 만든다.

*CountVectorizer는 단어장에 있는 숫자를 세주기만 함 (빈도수)
*TfidVectorizer는 빈도수 + 중요도에 따라 그 숫자를 줄이거나 늘리는 역할

4. HashingVectorizer:
해시 함수(hash function)을 사용하여 적은 메모리와 빠른 속도로 BOW 인코딩 벡터를 만든다.

*실무에서 우리가 쓰는 단어장이 너무 클 때, 메모리등의 문제로 처리가 힘든 경우가 있음. 이럴 때 사용

2-1) CountVectorizer

1. 문서를 토큰 리스트로 변환
2. 각 문서에서 토큰의 출현 빈도 count
3. 각 문서를 BOW 인코딩 벡터로 변환

<클래스 사용법>
1. 클래스 객체 생성
2. 말뭉치 넣고 fit 메서드 실행
3. vocabulary_ 속성에 단어장이 자동 생성됨
4. transform 메서드로 다른 문서를 BOW 인코딩
5. BOW 인코딩 된 결과는 Sparse 행렬로 만들어지므로, toarray메서드로 보통 행렬로 변환
    *sparse 행렬 : 0이 아닌 데이터만 행렬로 만드는 것 (메모리 효율화 차원)
    *toarray 메서드 : 우리가 아는 보통의 numpy array 로 바뀜

<클래스 사용 code>
```
0. corpus = ['AAA','BBB', ..]
1. vect = CountVectorizer()
2. vect.fit(corpus)
3. vect.vocabulary_
4-5. vect.transform(['AAA']).toarray()
```

<CountVectorizer() 인수>
CountVectorizer는 이러한 작업을 하기 위한 다음과 같은 인수를 가질 수 있다.

[단어장 생성 관련]
    stop_words : 문자열 {‘english’}, 리스트 또는 None (디폴트)
        stop words 목록.‘english’이면 영어용 스탑 워드 사용. stop_words는 단어장에 들어가지 않음
    ngram_range : (min_n, max_n) 튜플, *N그램 : 단어장 생성에 사용할 토큰의 크기
        n-그램 범위
    max_df : 정수 또는 [0.0, 1.0] 사이의 실수. 디폴트 1
        단어장에 포함되기 위한 최대 빈도
    min_df : 정수 또는 [0.0, 1.0] 사이의 실수. 디폴트 1
        단어장에 포함되기 위한 최소 빈도

[tokenizer 관련]
    analyzer : 문자열 {‘word’, ‘char’, ‘char_wb’} 또는 함수
        단어 n-그램, 문자 n-그램, 단어 내의 문자 n-그램
    token_pattern : string
        토큰 정의용 정규 표현식
    tokenizer : 함수 또는 None (디폴트)
        토큰 생성 함수 .

2-2) TF-IDF

Tf : term frequency (빈도수) idf : inverse Document frequency

- 단어장에서 그대로 카운트하지 않고, 중요도로 가중 !
  *중요도 = idf = 모든 문서에 공통적으로 들어있는 단어는 중요하지 않은 것(문서 구별능력이 떨어짐) = 가중치 축소

2-3) Hashing Trick

- Hash function
  : 문자를 받으면 숫자를 출력하는 함수 (속도가 매우 빠름)
  : BOW로 vectorize 시, 단어장에서 찾는 대신(Count, TF-IDF) 해시함수를 이용해 속도를 높인다)

<CountVectorizer>
- Count방식은 모든 작업을 메모리 상에서 수행
- 처리할 문서의 크기가 커지면, 단어장 딕셔너리가 커진다.
- 실행 속도가 느려지거나, 실행이 불가능해짐

<HashingVectorizer>
- 해시함수 사용
- 단어에 대한 인덱스 번호를 수식으로 생성
- 사전 메모리 필요없고, 실행 시간 단축 가능
- 단어의 충돌 가능(매우 드물게)

3. Gensim 패키지

- Bag of Words 인코딩
- TF - IDF 인코딩
- 토픽 모델링

<Gensim의 BOW 인코잉 기능>

1. Dictionary 클래스 이용
   - token2id 속성으로 사전 저장
   - doc2bow 메서드로 bow 저장

2. TfidModel 클래스를 이용하면 TF - IDF 인코딩도 가능

step 1) corpus 만들기

In [2]:

```
corpus = [
    'This is the first document.',
    'This is the second second document.',
    'And the third one.',
    'Is this the first document?',
    'The last document?',
]
```

step 2) 토큰 리스트 생성

In [4]:

```
token_list = [[text for text in doc.split()] for doc in corpus]
token_list
```

Out[4]:

```
[['This', 'is', 'the', 'first', 'document.'],
 ['This', 'is', 'the', 'second', 'second', 'document.'],
 ['And', 'the', 'third', 'one.'],
 ['Is', 'this', 'the', 'first', 'document?'],
 ['The', 'last', 'document?']]
```

step 3) Dictionary 객체 생성

In [6]:

```
from gensim.corpora import Dictionary

dictionary = Dictionary(token_list)
dictionary.token2id    #sklearn 에선 dictionary.vocabulary_ 일 것
```

Out[6]:

```
{'This': 0,
 'document.': 1,
 'first': 2,
 'is': 3,
 'the': 4,
 'second': 5,
 'And': 6,
 'one.': 7,
 'third': 8,
 'Is': 9,
 'document?': 10,
 'this': 11,
 'The': 12,
 'last': 13}
```

step 4) BOW 인코딩

- sparse 행렬로 출력

In [7]:

```
term_matrix = [dictionary.doc2bow(token) for token in token_list]
term_matrix
```

Out[7]:

```
[[(0, 1), (1, 1), (2, 1), (3, 1), (4, 1)],
 [(0, 1), (1, 1), (3, 1), (4, 1), (5, 2)],
 [(4, 1), (6, 1), (7, 1), (8, 1)],
 [(2, 1), (4, 1), (9, 1), (10, 1), (11, 1)],
 [(10, 1), (12, 1), (13, 1)]]
```

step 5) TF-IDF 인코딩

In [15]:

```
from gensim.models import TfidfModel

tfidf = TfidfModel(term_matrix)

for doc in tfidf[term_matrix]:
    print("doc : ")
    for k, v in doc:
        print(k,v)
```

```
doc :
0 0.49633406058198626
1 0.49633406058198626
2 0.49633406058198626
3 0.49633406058198626
4 0.12087183801361165
doc :
0 0.25482305694621393
1 0.25482305694621393
3 0.25482305694621393
4 0.0620568558708622
5 0.8951785160431313
doc :
4 0.07979258234193365
6 0.5755093812740171
7 0.5755093812740171
8 0.5755093812740171
doc :
2 0.3485847413542797
4 0.08489056411237639
9 0.6122789185961829
10 0.3485847413542797
11 0.6122789185961829
doc :
10 0.37344696513776354
12 0.6559486886294514
13 0.6559486886294514
```

4. Topic modeling 실습

1) 텍스트 데이터 다운로드

In [17]:

```
from sklearn.datasets import fetch_20newsgroups

newsgroups = fetch_20newsgroups(categories = ['comp.graphics','rec.sport.baseball','sci.med'])
```

2) 명사추출

In [18]:

```
from nltk.tag import pos_tag
from nltk.tokenize import word_tokenize

tagged_list = [pos_tag(word_tokenize(doc)) for doc in newsgroups.data]
nouns_list = [[t[0] for t in doc if t[1].startswith('N')] for doc in tagged_list]
```

3) 표제어 추출

In [19]:

```
from nltk.stem import WordNetLemmatizer

lm = WordNetLemmatizer()

nouns_list = [[lm.lemmatize(w, pos='n') for w in doc] for doc in nouns_list]
```

4) 불용어 제거

In [21]:

```
import re
token_list = [[text.lower() for text in doc] for doc in nouns_list]
token_list = [[re.sub(r"[^A-Za-z]+",'',word) for word in doc] for doc in token_list]
```

5) Topic 모델링

In [22]:

```
from gensim import corpora

dictionary = corpora.Dictionary(token_list)
doc_term_matrix = [dictionary.doc2bow(tokens) for tokens in token_list]
```