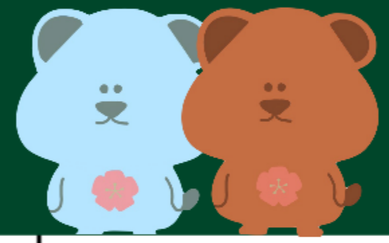


08 텍스트 분석

3팀 민소연, 최은빈

EWHA EURON

8. 텍스트 분석

NLP vs 텍스트 분석

- NLP(National Language Processing) : 머신이 인간의 언어를 이해하고 해석하는 데 중점
- 텍스트 분석(Text Analytics, TA) : **비정형 텍스트**에서 의미 있는 정보를 추출하는 것에 중점

→ NLP는 텍스트 분석을 향상하게 하는 기반 기술.

→ NLP 기술이 발전함에 따라 텍스트 분석도 더욱 정교하게 발전.

NLP	기계 번역, 질의 응답 시스템
텍스트 분석	머신러닝, 언어 이해, 통계 등을 활용한 모델을 수립하고 정보를 추출해 비즈니스 인텔리전스나 예측 분석 등의 분석 작업 수행

8. 텍스트 분석

머신러닝에 기반한 텍스트 분석 기술 영역

- 텍스트 분류, 감성 분석, 텍스트 요약, 텍스트 군집화

텍스트 분류	• 문서가 특정 분류 또는 카테고리에 속하는 것을 예측하는 기법 • Ex) 특정 신문 기사 내용이 연예/정치/사회/문화 중 어떤 카테고리에 속하는지 자동으로 분류, 스팸 메일 검출 • 지도학습 적용
감성 분석	• 텍스트에서 나타나는 감정/판단/믿음/의견/기분 등의 주관적인 요소를 분석하는 기법 • 소셜 미디어 감정 분석, 영화나 제품에 대한 긍정 또는 리뷰, 여론조사 의견 분석 등의 다양한 영역에서 활용 • Text Analytics에서 가장 활발하게 사용되는 분야 • 지도학습 뿐만 아니라 비지도학습을 이용해 적용 가능

8. 텍스트 분석

머신러닝에 기반한 텍스트 분석 기술 영역

- 텍스트 분류, 감성 분석, 텍스트 요약, 텍스트 군집화

텍스트 요약	• 텍스트 내에서 중요한 주제나 중심 사상을 추출하는 기법 • 대표적으로 토픽 모델링(Topic Modeling)이 있음
텍스트 군집화 & 유사도 측정	• 비슷한 유형의 문서에 대해 군집화를 수행하는 기법 • 텍스트 분류를 비지도학습으로 수행하는 방법의 일환으로 사용됨 • 유사도 측정 역시 문서들간의 유사도를 측정해 비슷한 문서끼리 모으는 방법

8.1 텍스트 분석 이해

EWHA EURON

8.1 텍스트 분석 이해

- 텍스트 분석: 비정형 데이터인 텍스트를 분석하는 것
- 지금까지 ML 모델 : 주어진 정형 데이터 기반에서 모델 수립/예측 수행
- 머신러닝 알고리즘은 숫자형의 피처기반 데이터만 입력받을 수 있기 때문에 **'비정형 텍스트 데이터를 어떻게 피처 형태로 추출하고 추출된 피처에 의미있는 값을 부여하는가'** 하는 것이 매우 중요한 요소

피처 벡터화(Feature Vectorization)

- 피처 추출(Feature Extraction)이라고도 함.
- 텍스트를 벡터값을 가지는 피처로 변환하는 것.
- 텍스트를 word(또는 word의 일부분) 기반의 다수의 피처로 추출하고 이 피처에 단어 빈도수와 같은 숫자 값을 부여하면 텍스트는 단어의 조합인 벡터값으로 표현될 수 있는데, 이렇게 텍스트를 변환하는 것을 말함.
- 대표적으로 **BOW(Bag of Words)**와 **Word2Vec** 방법이 있음.

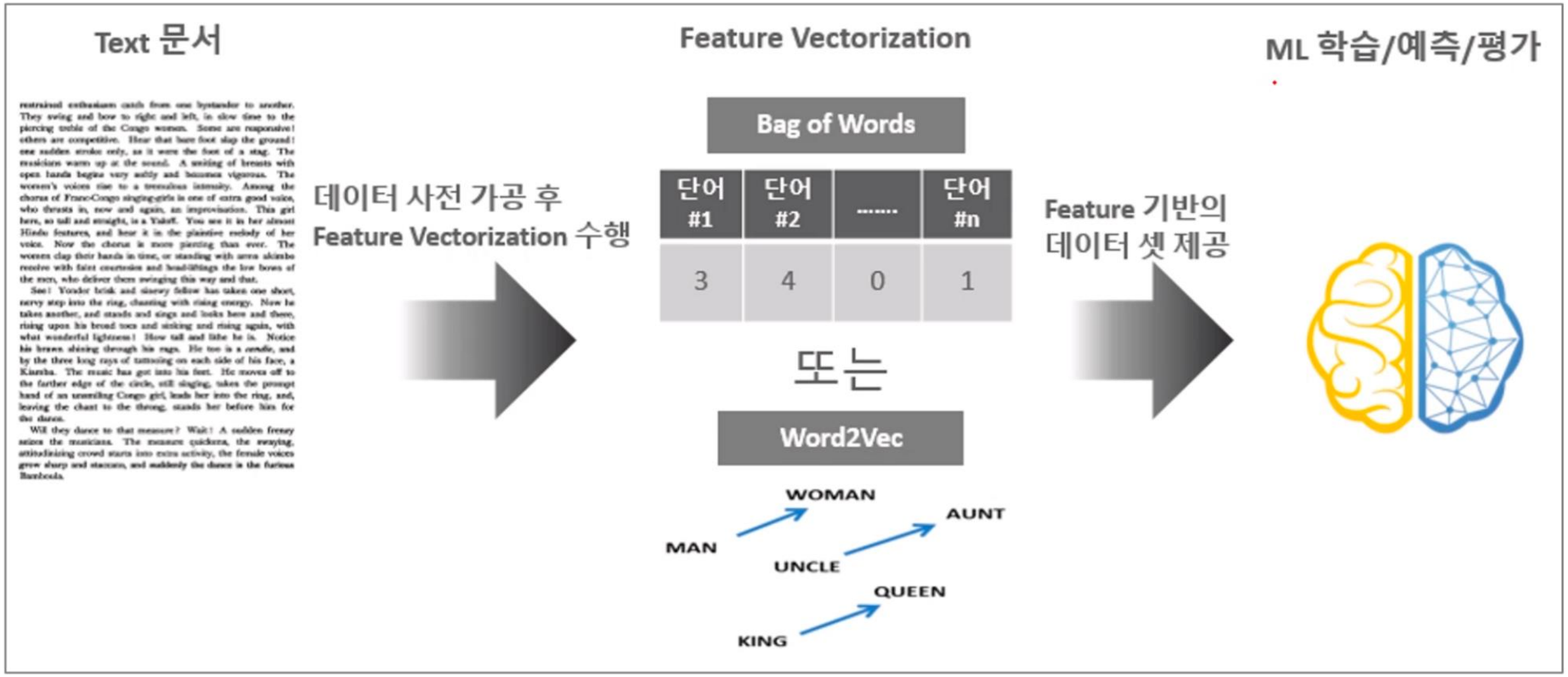
8.1 텍스트 분석 이해

텍스트 분석 수행 프로세스

1. **텍스트 사전 준비작업(텍스트 전처리)**
   - 텍스트를 피처로 만들기 전에 미리 클렌징 작업(대/소문자 변경, 특수문자 삭제), 단어(Word) 등의 토큰화 작업, 어근 추출 등의 텍스트 정규화 작업을 통칭
2. **피처 벡터화 / 추출**
   - 사전 준비 작업으로 가공된 텍스트에서 피처를 추출하고 벡터값 할당
   - 대표적인 방법 BOW, Word2Vec
   - BOW는 대표적으로 Count 기반과 TF-IDF 기반 벡터화가 있음
3. **ML 모델 수립 및 학습/예측/평가**
   - 피처 벡터화된 데이터 세트에 ML 모델 적용해 학습/예측 및 평가 수행

8.1 텍스트 분석 이해

텍스트 분석 수행 프로세스



8.1 텍스트 분석 이해

파이썬 기반의 NLP, 텍스트 분석 패키지

- NTLK(Natural Language Toolkit for Python)
  - 파이썬의 가장 대표적인 NLP 패키지
  - 방대한 데이터 세트와 서브 모듈을 가지고 있으며 NLP의 거의 모든 영역을 커버함
  - 많은 NLP 패키지가 NLTK의 영향을 받아 작성되고 있음
  - 수행속도 측면에서 아쉬운 부분이 있어 실제 대량의 데이터 기반에서는 제대로 활용되지 못하고 있음
- Gensim
  - 토픽 모델링 분야에서 가장 두각을 나타내는 패키지
  - 오래전부터 토픽 모델링을 쉽게 구현할 수 있는 기능 제공
  - SpaCy와 함께 가장 많이 사용되는 NLP 패키지
- SpaCy
  - 뛰어난 수행 성능으로 최근 가장 주목을 받는 NLP 패키지
  - NLP 애플리케이션에서 SpaCy를 사용하는 사례가 늘고 있음

8.1 텍스트 분석 이해

파이썬 기반의 NLP, 텍스트 분석 패키지

- 사이킷런은 머신러닝 위주의 라이브러리여서 NLP를 위한 ‘어근 처리 ' 와 같은 NLP 패키지에 특화된 다양한 라이브러리가 없음.
- 하지만 텍스트를 일정 수준으로 가공하고 머신러닝 알고리즘에 텍스트 데이터를 피처로 처리하기 위한 편리한 기능을 제공하고 있어 사이킷런으로도 충분히 텍스트 분석 수행 가능.
- 더 다양한 텍스트 분석이 적용되어야 하는 경우, 보통 NLTK/Gensim/SpaCy와 같은 NLP 전용 패키지와 결합해 애플리케이션을 작성하는 경우가 많다.

텍스트 사전 준비작업

8.2 텍스트 정규화

EWHA EURON

8.2 텍스트 정규화

텍스트 정규화란?

- 텍스트를 머신러닝 아록리즘이나 NLP 애플리케이션에 입력 데이터로 사용하기 위해 클렌징, 정제, 토큰화, 어근화 등의 다양한 텍스트 데이터의 사전 작업을 수행하는 것을 의미한다

- 클렌징(Cleansing)
- 토큰화(Tokenization)
- 필터링 / 스톱 워드 제거 / 철자 수정
- Stemming
- Lemmatization

8.2 텍스트 정규화

클렌징

- 텍스트에서 분석에 방해가 되는 불필요한 문자, 기호 등을 사전에 제거하는 작업.
- ex) HTML, XML 태그나 특정 기호 등

텍스트 토큰화

- 문서에서 문장을 분리하는 문장 토큰화, 문장에서 단어를 토큰으로 분리하는 단어 토큰화
- NLTK는 이를 위해 다양한 API 제공

▶ 문장 토큰화

- 문장의 마침표(.), 개행문자(\n) 등 문장의 마지막을 뜻하는 기호에 따라 분리
- 정규 표현식에 따른 문장 토큰화도 가능
- NLTK의 sent_tokenize() : 각각의 문장으로 구성된 list 객체 반환
  - NLTK의 경우 단어 사전과 같이 참조가 필요한 데이터 세트의 경우 인터넷으로 다운로드 가능

8.2 텍스트 정규화

▶ 문장 토큰화

- 문장의 마침표(.), 개행문자(\n) 등 문장의 마지막을 뜻하는 기호에 따라 분리
- 정규 표현식에 따른 문장 토큰화도 가능
- 각 문장이 가지는 시맨틱적인 의미가 중요한 요소로 사용될 때 사용
- NLTK의 sent_tokenize() : 각각의 문장으로 구성된 list 객체 반환
  - NLTK의 경우 단어 사전과 같이 참조가 필요한 데이터 세트의 경우 인터넷으로 다운로드 가능

```
from nltk import sent_tokenize
import nltk
nltk.download('punkt')

text_sample = 'The Matrix is everywhere its all around us, here even in this room. ₩
               You can see it out your window or on your television. ₩
               You feel it when you go to work, or go to church or pay your taxes.'
sentences = sent_tokenize(text=text_sample)
print(type(sentences),len(sentences))
print(sentences)
```

반환된 list 객체가 3개의 문장으로 된 문자열을 가지고 있는 것을 알 수 있음

```
<class 'list'> 3
['The Matrix is everywhere its all around us, here even in this room.', 'You can see it out your window or on your television.', 'You feel i
t when you go to work, or go to church or pay your taxes.']
```

8.2 텍스트 정규화

▶ 단어 토큰화

- 문장을 단어로 토큰화하는 것
- 기본적으로 공백, 콤마(,), 마침표(.), 개행문자 등으로 단어 분리
- 정규 표현식을 이용하여 다양한 유형으로 토큰화 수행 가능
- Bag of Word와 같이 단어의 순서가 중요하지 않은 경우 문장 토큰화를 사용하지 않고 단어 토큰화만 사용해도 충분함
- NTLK의 word_tokenize( )

```
from nltk import word_tokenize

sentence = "The Matrix is everywhere its all around us, here even in this room."
words = word_tokenize(sentence)
print(type(words), len(words))
print(words)
```

각 단어와 콤마, 마침표 등으로 단어가 분리되어 토큰화된 것을 확인할 수 있음

```
<class 'list'> 15
['The', 'Matrix', 'is', 'everywhere', 'its', 'all', 'around', 'us', ',', 'here', 'even', 'in', 'this', 'room', '.']
```

8.2 텍스트 정규화

문장별로 단어 토큰화하기

- 문서를 먼저 문장으로 나누고, 개별 문장을 다시 단어로 토큰화하는 tokenize_text( ) 함수 생성

```
[3]  from nltk import word_tokenize, sent_tokenize

     #여러개의 문장으로 된 입력 데이터를 문장별로 단어 토큰화 만드는 함수 생성
     def tokenize_text(text):

         # 문장별로 분리 토큰
         sentences = sent_tokenize(text)
         # 분리된 문장별 단어 토큰화
         word_tokens = [word_tokenize(sentence) for sentence in sentences]
         return word_tokens

     #여러 문장들에 대해 문장별 단어 토큰화 수행.
     word_tokens = tokenize_text(text_sample)
     print(type(word_tokens),len(word_tokens))
     print(word_tokens)
```

3개 문장을 문장별로 먼저 토큰화했으므로 word_tokens 변수는 3개의 리스트 객체를 내포하는 리스트이다.
내포된 개별 리스트 객체는 각각 문장별로 토큰화된 단어를 요소로 가진다.

```
<class 'list'> 3
[['The', 'Matrix', 'is', 'everywhere', 'its', 'all', 'around', 'us', ',', 'here', 'even', 'in', 'this', 'room', '.'], ['You', 'ca
n', 'see', 'it', 'out', 'your', 'window', 'or', 'on', 'your', 'television', '.'], ['You', 'feel', 'it', 'when', 'you', 'go', 't
o', 'work', ',', 'or', 'go', 'to', 'church', 'or', 'pay', 'your', 'taxes', '.']]
```

8.2 텍스트 정규화

n-gram

- 문장을 단어별로 하나씩 토큰화 할 경우 문맥적인 의미는 무시될 수밖에 없음.
- 이러한 문제를 해결하기 위해 도입된 것이 n-gram

- n-gram : 연속된 n개의 단어를 하나의 토큰화 단어로 분리해 내는 것
  - n개 단어 크기 윈도우를 만들어 문장의 처음부터 오른쪽으로 움직이면서 토큰화를 수행한다.

"Agent Smith knocks the door"

(Agent, Smith) (Smith, knocks) (knocks, the) (the, door)

- 위의 문장을 2-gram(bigram)으로 만들면 예시와 같이 연속적으로 2개의 단어들을 순차적으로 이동하면서 단어들을 토큰화한다.

8.2 텍스트 정규화

스톱 워드(stop word) 제거

- 스톱 워드 : 분석에 큰 의미가 없는 단어
- ex) 영어에서 is, the, a, will 등의 문장을 구성하는 필수 문법 요소지만 문맥적으로 큰 의미가 없는 단어가 해당
- 문법적인 특성으로 인해 스톱 워드가 텍스트에 빈번하게 나타나 사전에 제거하지 않으면 중요한 단어로 인지될 수 있음
- NLTK에서 가장 다양한 언어의 스톱 워드를 제공

```
import nltk
nltk.download('stopwords')
```

```
print('영어 stop words 갯수:',len(nltk.corpus.stopwords.words('english')))
print(nltk.corpus.stopwords.words('english')[:20])
```

```
영어 stop words 갯수: 179
['i', 'me', 'my', 'myself', 'we', 'our', 'ours', 'ourselves', 'you', "you're", "you've", "you'll", "you'd", 'your', 'yours', 'yourself', 'yo
urselves', 'he', 'him', 'his']
```

8.2 텍스트 정규화

스톱 워드(stop word) 제거

```
import nltk

stopwords = nltk.corpus.stopwords.words('english')
all_tokens = []
# 위 예제의 3개의 문장별로 얻은 word_tokens list 에 대해 stop word 제거 Loop
for sentence in word_tokens:
    filtered_words=[]
    # 개별 문장별로 tokenize된 sentence list에 대해 stop word 제거 Loop
    for word in sentence:
        #소문자로 모두 변환합니다.
        word = word.lower()
        # tokenize 된 개별 word가 stop words 들의 단어에 포함되지 않으면 word_tokens에 추가
        if word not in stopwords:
            filtered_words.append(word)
    all_tokens.append(filtered_words)

print(all_tokens)
```

is, this와 같은 스톱 워드가 필터링을 통해 제거됨

```
[['The', 'Matrix', 'is', 'everywhere', 'its', 'all', 'around', 'us', ',', 'here', 'even', 'in', 'this', 'room', '.'], ['You', 'ca
n', 'see', 'it', 'out', 'your', 'window', 'or', 'on', 'your', 'television', '.'], ['You', 'feel', 'it', 'when', 'you', 'go', 't
o', 'work', ',', 'or', 'go', 'to', 'church', 'or', 'pay', 'your', 'taxes', '.']]
```



```
[['matrix', 'everywhere', 'around', 'us', ',', 'even', 'room', '.'], ['see', 'window', 'television', '.'], ['feel', 'go', 'work', ',', 'go',
'church', 'pay', 'taxes', '.']]
```

8.2 텍스트 정규화

Stemming과 Lemmatization

- 많은 언어에서 문법적인 요소에 따라 단어가 다양하게 변화하기 때문에
- 문법적 또는 의미적으로 변화하는 단어의 원형을 찾는 방법
- ex) work(동사 원형), worked(과거형), works(3인칭 단수), working(진행형)

- 두 기능 모두 원형 단어를 찾는다는 목적은 유사하다.
- **Lemmatization**
  - Stemming보다 정교하며, 의미론적인 기반에서 단어의 원형을 찾음
  - 품사와 같은 문법적인 효소와 더 의미적인 부분을 감안해 정확한 철자로 된 어근 단어를 찾아줌
  - Stemming보다 변환에 더 오랜 시간이 걸림
- **Stemming**
  - 원형 단어로 변환 시 일반적인 방법을 적용하거나 더 단순화된 방법을 적용
  - 원래 단어에서 일부 철자가 훼손된 어근 단어를 추출하는 경향

8.2 텍스트 정규화

Stemming과 Lemmatization

- NLTK는 다양한 Stemmer를 제공
  - Porter, Lancaster, Snowball Stemmer
- Lemmatization
  - WordNetLemmatizer 제공

<LencasterStemmer( )를 이용한 Stemming>

- 진행형, 3인칭 단수, 과거형에 따른 동사, 그리고 비교, 최상에 따른 형용사의 변화에 따라 Stemming은 더 단순하게 원형 단어를 찾아준다.
- NTLK에서는 LencasterStemmer()와 같이 필요한 Stemmer 객체를 생성한 뒤 이 객체의 stem('단어') 메서드를 호출하면 원하는 '단어' 의 Stemming이 가능함.

8.2 텍스트 정규화

<LencasterStemmer( )를 이용한 Stemming>

- **Stemming**
  - 원형 단어로 변환 시 일반적인 방법을 적용하거나 더 단순화된 방법을 적용
  - 원래 단어에서 일부 철자가 훼손된 어근 단어를 추출하는 경향

```
from nltk.stem import LancasterStemmer
stemmer = LancasterStemmer()

print(stemmer.stem('working'),stemmer.stem('works'),stemmer.stem('worked'))
print(stemmer.stem('amusing'),stemmer.stem('amuses'),stemmer.stem('amused'))
print(stemmer.stem('happier'),stemmer.stem('happiest'))
print(stemmer.stem('fancier'),stemmer.stem('fanciest'))
```

```
work work work
amus amus amus
happy happiest
fant fanciest
```

- work의 경우 모두 기본 단어인 work + ing, s, ed가 붙는 단순한 변화이므로 원형단어 work 인식
- amuse의 경우 각 변화가 기본 단어 amuse가 아닌 amus + ing, s, ed => amus를 원형 단어로 인식
- happy, fancy도 비교형, 최상급형으로 변형된 단어의 정확한 원형을 찾지 못하고
- 원형 단어에서 철자가 다른 어근 단어로 인식하는 경우 발생

8.2 텍스트 정규화

<WordNetLemmatizer를 이용한 Lemmatization>

- Lemmatization은 보다 정확한 원형 단어 추출을 위해 단어의 '품사'를 입력해줘야 함
- lemmatize()의 파라미터로 동사의 경우 'v', 형용사의 경우 'a' 입력

```
from nltk.stem import WordNetLemmatizer
import nltk
nltk.download('wordnet')

lemma = WordNetLemmatizer()
print(lemma.lemmatize('amusing','v'),lemma.lemmatize('amuses','v'),lemma.lemmatize('amused','v'))
print(lemma.lemmatize('happier','a'),lemma.lemmatize('happiest','a'))
print(lemma.lemmatize('fancier','a'),lemma.lemmatize('fanciest','a'))
```

```
[nltk_data] Downloading package wordnet to /root/nltk_data...
amuse amuse amuse
happy happy
fancy fancy
```

Stemmer보다 정확하게 원형 단어를 추출해줌

8.3 Bag of Words

8.3 Bag of Words - BOW

Bag of Words 모델

- 문서가 가지는 모든 단어를 문맥이나 순서를 무시하고 일괄적으로 단어에 대해 빈도 값을 부여해 피처 값을 추출하는 모델
- 장점
  - 쉽고 빠른 구축
  - 단순히 단어의 발생 횟수에 기반하고 있지만, 예상보다 문서의 특징을 잘 나타낼 수 있는 모델이어서 전통적으로 여러 분야에서 활용도가 높음
- 단점
  - 문맥 의미 반영 부족
  - 희소 행렬 문제(희소성, 희소 행렬)

8.3 Bag of Words - BOW

Bag of Words 모델

- 단점
  - 문맥 의미 반영 부족
    - 단어의 순서를 고려하지않기 때문에 문장 내에서 단어의 문맥적인 의미가 무시됨
    - n_gram 기법을 통해 보완할 수 있지만 제한적인 부분에 그치므로 문맥적인 해석을 처리하지 못함
  - 희소 행렬 문제(희소성, 희소 행렬)
    - 매우 많은 문서에서 단어의 총 개수는 수만 ~ 수십만 개가 될 수 있는데, 하나의 문서에 있는 단어는 이 중 극히 일부분임. 따라서 대규모의 칼럼으로 구성된 행렬에서 대부분의 값이 0으로 채워져 ML 알고리즘의 수행시간, 예측 성능을 떨어뜨림 => 희소 행렬을 위한 특별한 기법이 마련되어 있음

- 희소 행렬 : 대규모의 칼럼으로 구성된 행렬에서 대부분의 값이 0으로 채워지는 행렬
- 밀집 행렬 : 대규모의 칼럼으로 구성된 행렬에서 대부분의 값이 0이 아닌 의미 있는 값으로 채워져 있는 행렬

8.3 Bag of Words - BOW

BOW 피처 벡터화

- 피처 벡터화 : 머신러닝 알고리즘은 일반적으로 숫자형 피처를 데이터로 입력받아 동작하기 때문에 텍스트와 같은 데이터는 머신러닝 알고리즘에 바로 입력할 수 X
- 따라서 텍스트는 특정 의미를 가진 숫자형 값인 벡터 값으로 변환해야 하는데, 이러한 변환을 피처 벡터화라고 함.

<피처 벡터화>

1. 각 문서의 텍스트를 단어로 추출해 피처로 할당
2. 각 단어의 발생 빈도와 같은 값을 해당 피처에 부여해 벡터로 생성

8.3 Bag of Words - BOW

BOW 피처 벡터화

- BOW 모델에서의 피처 벡터화
  - 모든 문서에서 모든 단어를 칼럼 형태로 나열하고 각 문서에서 해당 단어의 횟수나 정규화된 빈도를 값으로 부여하는 데이터 세트 모델로 변경하는 것
  - M개의 텍스트 문서가 있고 이 문서에서 모든 단어를 추출해 나열했을 때 N개의 단어가 있다고 가정하면 문서의 피처 벡터화를 수행하면 M개의 문서는 각각 N개의 값이 할당된 피처의 벡터 세트가 됨
  - => M x N개의 단어 피처로 이뤄진 행렬 구성

- BOW 피처 벡터화 방식
  - 카운트 기반의 벡터화
  - TF-IDF(Term Frequency – Inverse Document Frequency) 기반의 벡터화

8.3 Bag of Words - BOW

카운트 벡터화, TF-IDF 벡터화

- **카운트 벡터화**
  - 단어 피처에 값을 부여할 때 각 문서에서 해당 언어가 나타나는 횟수, 즉 Count를 부여하는 경우
  - 카운트 값이 높을수록 중요한 단어로 인식됨
  - 하지만 카운트만 부여할 경우 그 문서의 특징을 나타내기보다는 언어의 특성상 문장에서 자주 사용될 수밖에 없는 단어까지 높은 값을 부여하게 됨

- **TF-IDF 벡터화**
  - 카운트 벡터화의 문제점을 보완할 수 있음
  - 개별 문서에서 자주 나타나는 단어에 높은 가중치를 주되, 모든 문서에서 전반적으로 자주 나타나는 단어에 대해서는 페널티를 주는 방식으로 값을 부여해서 단어에 대한 가중치의 균형을 맞춤
  - 문서마다 텍스트가 길고 문서의 개수가 많은 경우 카운트 방식보다는 **TF-IDF 방식을 사용하는 것이 더 좋은 예측 성능을 보장할 수 있음**

8.3 Bag of Words - BOW

- 여러 가지 뉴스의 문서 중
- ‘분쟁’, ‘종교 대립’, ‘유혈 사태’ 등의 단어 자주 등장

⇒ 지역 분쟁과 관련한 뉴스일 가능성이 높음. 해당 단어는 그 문서의 특징을 잘 나타냄.

- ‘많은’, ‘빈번하게’, ‘당연히’, ‘조직’, ‘업무’ 등의 단어

⇒ 문서의 특징과 관련성이 적지만 보편적으로 많이 사용되기 때문에 문서에 반복적으로 사용될 가능성이 높음

⇒ 이러한 단어가 단순히 등장하는 횟수에 따라 중요도를 평가받는다면 문서를 특정짓기 어려워짐

$$TFIDF_i = TF_i * log\frac{N}{DF_i}$$

TF_i = 개별 문서에서의 단어 i 빈도

DF_i = 단어 i 를 가지고 있는 문서 개수

N = 전체 문서 개수

8.3 Bag of Words - BOW

사이킷런의 Count 및 TF-IDF 벡터화 구현

- **CountVectorizer 클래스**
  - 카운트 기반의 벡터화를 구현한 클래스
  - 단지 피처 벡터화만 수행하지는 않으며 소문자 일괄 변환, 토큰화, 스톱 워드 필터링 등의 텍스트 전처리도 함께 수행함
  - fit()과 transform()을 통해 피처 벡터화된 객체를 반환함

- **TfidfVectorizer 클래스**
  - TF-IDF 벡터화를 구현한 클래스
  - 파라미터와 변환 방법은 CountVectorizer와 동일

8.3 Bag of Words - BOW

CountVectorizer 입력 파라미터

파라미터 명	파라미터 설명
max_df	전체 문서에 걸쳐서 너무 높은 빈도수를 가지는 단어 피처를 제외하기 위한 파라미터 너무 높은 빈도수를 가지는 단어는 문법적인 특성으로 반복적인 단어일 가능성이 높아 제거하기 위해 사용 - max_df = 100과 같이 정수 값: 전체 문서에 걸쳐 100개 이하로 나타나는 단어만 피처로 추출 - max_df = 0.95와 같이 부동소수점 값(0.0~1.0): 전체 문서에 걸쳐 빈도수 0~95% 까지의 단어만 피처로 추출하고 나머지 상위 5%는 피처로 추출하지 X
min_df	전체 문서에 걸쳐서 너무 낮은 빈도수를 가지는 단어 피처를 제외하기 위한 파라미터 특정 단어가 min_df에 설정된 값보다 적은 빈도수를 가진다면 이 단어는 크게 중요하지 않거나 가비지(garbage)성 단어일 확률이 높음 - min_df = 2와 같이 정수 값: 전체 문서에 걸쳐서 2번 이하로 나타나는 단어는 피처로 추출 X - min_df = 0.02와 같이 부동소수점 값(0.0~1.0): 전체 문서에 걸쳐서 하위 2% 이하의 빈도수를 가지는 단어는 피처로 추출하지 X

8.3 Bag of Words - BOW

CountVectorizer 입력 파라미터

파라미터 명	파라미터 설명
max_features	추출하는 피처의 개수를 제한하며 정수로 값을 지정 max_features = 2000 : 가장 높은 빈도를 가지는 단어 순으로 정렬해 2000개까지만 피처로 추출
stop_words	‘english’ 로 지정하면 영어의 스톱 워드로 지정된 단어는 추출에서 제외한다
n_gram_range	Bag of Words 모델의 단어 순서를 어느 정도 보강하기 위한 n_gram 범위를 설정한다 튜플 형태로 (범위 최솟값, 범위 최댓값)을 지정 (1,1)로 지정: 토큰화된 단어를 1개씩 피처로 추출, (1,2)로 지정: 토큰화된 단어를 1개씩(minimum 1), 그리고 순서대로 2개씩(maximum 2) 묶어서 피처로 추출
analyzer	피처 추출을 수행한 단위를 지정. 디폴트는 ‘word’ . Word가 아니라 character의 특정 범위를 피처로 만드는 특정한 경우 등을 적용할 때 사용됨

8.3 Bag of Words - BOW

CountVectorizer 입력 파라미터

파라미터 명	파라미터 설명
token_pattern	토큰화를 수행하는 정규 표현식 패턴을 지정 디폴트 값은 '\b\w\w+\b' 로, 공백 또는 개행 문자 등으로 구분된 단어 분리자(\b) 사이의 2문자(문자 또는 숫자, 즉 영숫자) 이상의 단어(word)를 토큰으로 분리 analyzer= 'word' 로 설정했을 때만 변경 가능하나, 디폴트 값을 변경할 경우는 거의 발생 X
tokenizer	토큰화를 별도의 커스텀 함수로 이용 시 적용 일반적으로 CountTokenizer 클래스에서 어근 변환 시 이를 수행하는 별도의 함수를 tokenizer 파라미터에 적용하면 됨

8.3 Bag of Words - BOW

CountVectorizer를 이용한 피처 벡터화 과정

사전 데이터 가공

- 모든 문자를 소문자로 변환하는 등의 사전 작업 수행
- (Default로 lowercase = True임)

토큰화

- 디폴트는 단어 기준(analyzer = True)
- n_gram_range를 반영해 각 단어를 토큰화

텍스트 정규화

- stop_words 파라미터가 주어진 경우 스톱 워드 필터링만 가능
- 어근 변환은 자체 지원 X, tokenizer 파라미터에 커스텀 어근 변환 함수를 적용하여 어근 변환 수행 가능 (Stemming, Lemmatization)

피처 벡터화

- max_df, min_df, max_features 등의 파라미터를 반영하여 Token된 단어들을 피처로 추출 후 단어 빈도수 벡터 값(vectorization) 적용

8.3 Bag of Words - BOW

BOW 벡터화를 위한 희소행렬

1) BOW 벡터화 방식

문장1: “My wife likes to watch baseball games and my daughter likes to watch baseball games too”

문장2: “My wife likes to play baseball”

	Index 0	Index 1	Index 2	Index 3	Index 4	Index 5	Index 6	Index 7	Index 8	Index 9	Index 10
	and	baseball	daughter	games	likes	my	play	to	too	watch	wife
문장 1	1	2	1	2	2	2	0	2	1	2	1
문장 2	0	1	0	0	1	1	1	1	0	0	1

문장 1에서 baseball은 2회 나타남

8.3 Bag of Words - BOW

BOW 벡터화를 위한 희소행렬

1) BOW 벡터화 방식

1-Gram	2-Gram	3-Gram
The	The Margherita	The Margherita pizza
Margherita	Margherita pizza	Margherita pizza is
pizza	pizza is	pizza is not
is	is not	is not bad
not	not bad	not bad taste
bad	bad taste	
taste		

8.3 Bag of Words - BOW

2) COO 형식

: 0이 아닌 데이터만 별도의 array에 저장하고,
그 데이터가 가리키는 행과 열의 위치를 별도의 배열로 저장하는 방식

```
#행렬 만들기
import numpy as np

dense = np.array( [ [3, 0, 1], [0, 2, 0] ] )
```

8.3 Bag of Words - BOW

2) COO 형식

```
from scipy import sparse

#0이 아닌 데이터를 추출
data = np.array( [3, 1, 2] )

#'3'의 위치를 행렬로 나타내면 (0,0)
#'1'의 위치를 행렬로 나타내면 (0,2)
#'2'의 위치를 행렬로 나타내면 (1,1)

#행 위치와 열 위치를 각각 배열로 생성
row_pos = np.array( [ 0, 0, 1 ])
col_pos = np.array( [ 0, 2, 1 ])

#sparse 패키지의 coo_matrix를 이용해 COO 형식으로 희소 행렬 생성
sparse_coo = sparse.coo_matrix((data, (row_pos, col_pos)))
```

8.3 Bag of Words - BOW

2) COO 형식

Dense 형식의 원본 데이터

[[0,0,1,0,0,5],
[1,4,0,3,2,5],
[0,6,0,3,0,0],
[2,0,0,0,0,0],
[0,0,0,7,0,8],
[1,0,0,0,0,0]]

0이 아닌 데이터 값 배열

[1, 5, 1, 4, 3, 2, 5, 6, 3, 2, 7, 8, 1]

0이 아닌 데이터 값의 행과 열 위치

(0, 2), (0,5)
(1, 0), (1, 1), (1, 3), (1, 4), (1, 5)
(2, 1), (2, 3)
(3, 0)
(4, 3), (4, 5)
(5, 0)

행 위치 배열

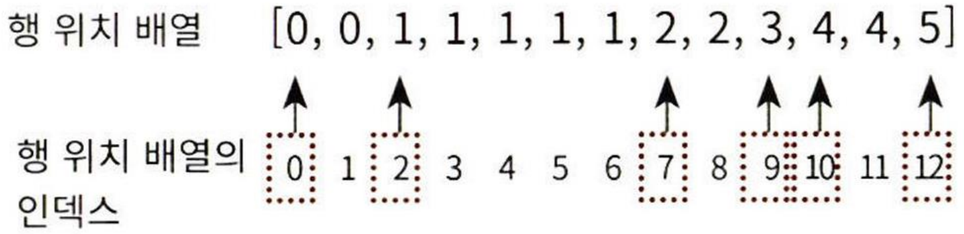
[0, 0, 1, 1, 1, 1, 1, 2, 2, 3, 4, 4, 5]

열 위치 배열

[2, 5, 0, 1, 3, 4, 5, 1, 3, 0, 3, 5, 0]

8.3 Bag of Words - BOW

3) CSR 형식



행 위치 배열의 고유값 시작 인덱스 배열 [0, 2, 7, 9, 10, 12]

\+

총 항목 개수 배열 [13]

행 위치 배열의 고유값 시작 인덱스 배열 최종 [0, 2, 7, 9, 10, 12, 13]

8.3 Bag of Words - BOW

3) CSR 형식

```
from scipy import sparse

dense2= np.array( [[0, 0, 1, 0, 0, 5],
                   [1, 4, 0, 3, 2, 5],
                   [0, 6, 0, 3, 0, 0],
                   [2, 0, 0, 0, 0, 0],
                   [0, 0, 0, 7, 0, 8],
                   [1, 0, 0, 0, 0, 0]])

#0이 아닌 데이터 추출
data2= np.array([1,5,1,4,3,2,5,6,3,2,7,8,1])

#행 위치와 열 위치를 각각 array로 생성
row_pos = np.array([0,0,1,1,1,1,1,2,2,3,4,4,5])
col_pos = np.array([2,5,0,1,3,4,5,1,3,0,3,5,0])

#COO 형식으로 변환
sparse_coo = sparse.coo_matrix((data2, (row_pos, col_pos)))

#행 위치 배열의 고유한 값의 시작 위치 인덱스를 배열로 생성
row_pos_ind = np.array([0,2,7,9,10,12,13])

#CSR 형식으로 변환
sparse_csr = sparse.csr_matrix((data2, col_pos, row_pos_ind))

print('COO 변환된 데이터가 제대로 되었는지 다시 Dense로 출력 확인')
print(sparse_coo.toarray())
print('CSR 변환된 데이터가 제대로 되었는지 다시 Dense로 출력 확인')
print(sparse_csr.toarray())
```

8.5 감성 분석

EWHA EURON

감성 분석 실습

1) 지도학습 기반

```
from sklearn.model_selection import train_test_split

class_df = review_df['sentiment']
feature_df = review_df.drop(['id', 'sentiment'], axis = 1, inplace = False)

X_train, X_test, y_train, y_test = train_test_split(feature_df, class_df, test_size = 0.3, random_state= 156)
```

```
from sklearn.feature_extraction.text import CountVectorizer, TfidfVectorizer
from sklearn.pipeline import Pipeline
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, roc_auc_score

#스톱 워드는 English, filtering, ngram은 (1,2)로 설정해 CountVectorization 수행
#LogisticRegression의 C는 10으로 설정
pipeline= Pipeline([
    ('cnt_vect', CountVectorizer(stop_words = 'english', ngram_range = (1,2))),
    ('lr_clf', LogisticRegression(C= 10))])

#Pipeline 객체를 이용해 fit(), predict() 로 학습/예측 수행. predict_proba()는 roc_auc 때문에 수행.
pipeline.fit(X_train['review'], y_train)
pred = pipeline.predict(X_test['review'])
pred_probs= pipeline.predict_proba(X_test['review'])[:,1]

print('예측 정확도는 {0:.4f}, ROC-AUC는 {1:.4f}'.format(accuracy_score(y_test, pred),
                                                roc_auc_score(y_test, pred_probs)))
```

감성 분석 실습

1) 지도학습 기반

```
#TD-IDF 벡터화
#스톱 워드는 English, filtering, ngram은 (1,2)로 설정해 CountVectorization 수행
#LogisticRegression의 C는 10으로 설정
pipeline= Pipeline([
    ('tfidf_vect', TfidfVectorizer(stop_words = 'english', ngram_range = (1,2))),
    ('lr_clf', LogisticRegression(C= 10))])

pipeline.fit(X_train['review'], y_train)
pred = pipeline.predict(X_test['review'])
pred_probs= pipeline.predict_proba(X_test['review'])[:,1]

print('예측 정확도는 {0:.4f}, ROC-AUC는 {1:.4f}'.format(accuracy_score(y_test, pred),
                                                    roc_auc_score(y_test, pred_probs)))
```

```
예측 정확도는 0.8936, ROC-AUC는 0.9598
```

감성 분석 실습

2) 비지도학습 기반

Sentiment Analysis Dictionaries

SentiWordNet, SentiWords, VADER

WORD	POLARITY	SD	INDIVIDUAL SCORES
attractions	1.8	0.87178	[1, 3, 0, 2, 2, 2, 2, 3, 1, 2]
attractive	1.9	0.53852	[2, 2, 2, 1, 3, 2, 1, 2, 2, 2]
attractively	2.2	0.6	[3, 2, 2, 3, 2, 2, 2, 3, 1, 2]
avoided	-1.4	0.4899	[-2, -1, -2, -1, -1, -1, -2, -1, -2, -1]
avoider	-1.8	0.6	[-2, -1, -3, -1, -2, -2, -2, -1, -2, -2]
avoiders	-1.4	0.66332	[-2, -2, -1, -2, -1, -1, 0, -1, -2, -2]

출처: https://www.youtube.com/watch?app=desktop&v=Fv1CrtakXPk

감성 분석 실습

2-1) SentiWordNet

```
#'present'라는 단어에 대한 Sysnset 추출

from nltk.corpus import wordnet as wn

term = 'present'

#'present' 라는 단어로 wordnet의 synsets 생성
synsets = wn.synsets(term)
print('synsets()반환 type:', type(synsets))
print('synsets()반환 값 개수:', len(synsets))
print('synsets()반환 값:', synsets)
```

```
synsets()반환 type: <class 'list'>
synsets()반환 값 개수: 18
synsets()반환 값: [Synset('present.n.01'), Synset('present.n.02'), Synset('present.n.03'), Synset('show.v.01'), Synset('presen
t.v.02'), Synset('stage.v.01'), Synset('present.v.04'), Synset('present.v.05'), Synset('award.v.01'), Synset('give.v.08'), Syn
set('deliver.v.01'), Synset('introduce.v.01'), Synset('portray.v.04'), Synset('confront.v.03'), Synset('present.v.12'), Synset
('salute.v.06'), Synset('present.a.01'), Synset('present.a.02')]
```

감성 분석 실습

2-1) SentiWordNet

```
for synset in synsets :
    print('##### Synset name: ', synset.name(), '#####')
    print('POS:', synset.lexname())
    print('Dfinition:', synset.definition())
    print('Lemmas:', synset.lemma_names())
```

```
##### Synset name:  present.n.01 #####
POS: noun.time
Dfinition: the period of time that is happening now; any continuous stretch of time including the moment of speech
Lemmas: ['present', 'nowadays']
##### Synset name:  present.n.02 #####
POS: noun.possession
Dfinition: something presented as a gift
Lemmas: ['present']
##### Synset name:  present.n.03 #####
POS: noun.communication
Dfinition: a verb tense that expresses actions or states at the time of speaking
Lemmas: ['present', 'present_tense']
##### Synset name:  show.v.01 #####
POS: verb.perception
Dfinition: give an exhibition of to an interested audience
Lemmas: ['show', 'demo', 'exhibit', 'present', 'demonstrate']
##### Synset name:  present.v.02 #####
POS: verb.communication
Dfinition: bring forward and present to the mind
Lemmas: ['present', 'represent', 'lay_out']
```

감성 분석 실습

2-1) SentiWordNet

```
# synset 객체를 단어별로 생성
tree = wn.synset('tree.n.01')
lion = wn.synset('lion.n.01')
tiger = wn.synset('tiger.n.02')
cat = wn.synset('cat.n.01')
dog = wn.synset('dog.n.01')

# 여러 객체를 하나의 리스트로 생성
entities = [tree , lion , tiger , cat , dog]

# 각 객체의 이름 리스트
entity_names = [ entity.name().split('.')[0] for entity in entities]

# 각 객체별 다른 객체와의 유사도 측정
similarities = []

#단어별 synset을 반복하며 다른 단어의 synset과의 유사도 측정
for entity in entities:
    similarity = [ round(entity.path_similarity(compared_entity), 2) for compared_entity in entities ]
    similarities.append(similarity)

#개별 단어별 synset과 다른 단어의 synset과의 유사도를 df로 저장
similarity_df = pd.DataFrame(similarities , columns=entity_names, index=entity_names)
similarity_df
```

감성 분석 실습

2-1) SentiWordNet

```
# synset 객체를 단어별로 생성
tree = wn.synset('tree.n.01')
lion = wn.synset('lion.n.01')
tiger = wn.synset('tiger.n.02')
cat = wn.synset('cat.n.01')
dog = wn.synset('dog.n.01')

# 여러 객체를 하나의 리스트로 생성
entities = [tree , lion , tiger , cat , dog]

# 각 객체의 이름 리스트
entity_names = [ entity.name().split('.')[0] for entity in entities]

# 각 객체별 다른 객체와의 유사도 측정
similarities = []

#단어별 synset을 반복하며 다른 단어의 synset과의 유사도 측정
for entity in entities:
    similarity = [ round(entity.path_similarity(compared_entity), 2) for compared_entity in entities ]
    similarities.append(similarity)

#개별 단어별 synset과 다른 단어의 synset과의 유사도를 df로 저장
similarity_df = pd.DataFrame(similarities , columns=entity_names, index=entity_names)
similarity_df
```

	tree	lion	tiger	cat	dog
tree	1.00	0.07	0.07	0.08	0.12
lion	0.07	1.00	0.33	0.25	0.17
tiger	0.07	0.33	1.00	0.25	0.17
cat	0.08	0.25	0.25	1.00	0.20
dog	0.12	0.17	0.17	0.20	1.00

감성 분석 실습

2-1) SentiWordNet

```
import nltk
from nltk.corpus import sentiwordnet as swn

#father의 긍/부정 감성지수, 객관성 지수
father = swn.senti_synset('father.n.01')
print('father 긍정감성 지수:', father.pos_score())
print('father 부정감성 지수:', father.neg_score())
print('father 객관성 지수:', father.obj_score())
print('₩n')

#fabulous의 긍/부정 감성지수
fabulous = swn.senti_synset('fabulous.a.01')
print('fabulous 긍정감성 지수:', fabulous.pos_score())
print('fabulous 부정감성 지수:', fabulous.neg_score())
```

```
father 긍정감성 지수: 0.0
father 부정감성 지수: 0.0
father 객관성 지수: 1.0


fabulous 긍정감성 지수: 0.875
fabulous 부정감성 지수: 0.125
```

감성 분석 실습

2-1) SentiWordNet

```
#단어의 긍/부정 예측하기
# 각 문서별로 긍정/부정 예측
review_df['preds'] = review_df['review'].apply( lambda x : swn_polarity(x) )

y_target = review_df['sentiment'].values
preds = review_df['preds'].values
```

```
#예측 성능 평가
from sklearn.metrics import accuracy_score, precision_score, recall_score, confusion_matrix
from sklearn.metrics import f1_score, roc_auc_score

print(confusion_matrix(y_target, preds))
print('정확도:', np.round(accuracy_score(y_target, preds), 4))
print('정밀도:', np.round(precision_score(y_target, preds), 4))
print('재현율:', np.round(recall_score(y_target, preds), 4))
```

```
[[7649 4851]
 [3578 8922]]
정확도: 0.6628
정밀도: 0.6478
재현율: 0.7138
```

감성 분석 실습

2-2) VADER

```
from nltk.sentiment.vader import SentimentIntensityAnalyzer

senti_analyzer = SentimentIntensityAnalyzer()
senti_scores = senti_analyzer.polarity_scores(review_df['review'][0])

# dictionary로 반환
print(senti_scores)
```

```
{'neg': 0.13, 'neu': 0.744, 'pos': 0.126, 'compound': -0.8278}
```

감성 분석 실습

2-2) VADER

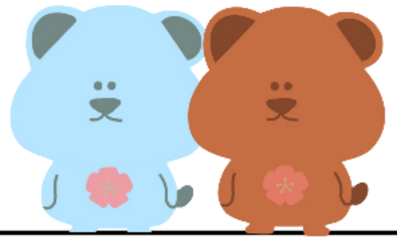
```
def vader_polarity(review, threshold = 0.1):
    from nltk.sentiment.vader import SentimentIntensityAnalyzer

    # VADER 객체로 감성 지수 산출
    analyzer = SentimentIntensityAnalyzer()
    scores = analyzer.polarity_scores(review)

    # 감성 지수가 threshold 보다 크거나 같으면 1, 그렇지 않으면 0
    agg_score = scores['compound']
    final_sentiment = 1 if agg_score >= threshold else 0

    return final_sentiment

# 각 문서별로 긍정/부정 예측
review_df['vader_preds'] = review_df['review'].apply( lambda x : vader_polarity(x, 0.1) )

y_target = review_df['sentiment'].values
vader_preds = review_df['vader_preds'].values
```

```
print(confusion_matrix(y_target, vader_preds))
print("정확도:", np.round(accuracy_score(y_target, vader_preds), 4))
```

```
[[ 6819  5681]
 [ 1936 10564]]
정확도: 0.6953
```

EWHA EURON

THANK YOU



EWHA EURON