텍스트 데이터 전처리

데이터 전처리

✓ 데이터 랭글링(data wrangling)

- 광범위한 의미의 원본 데이터를 정제하고 사용 가능한 형태로 구성하기 위한 변환 과정
- 데이터 랭글링에 사용되는 가장 일반적인 데이터 구조 - 데이터프레임.

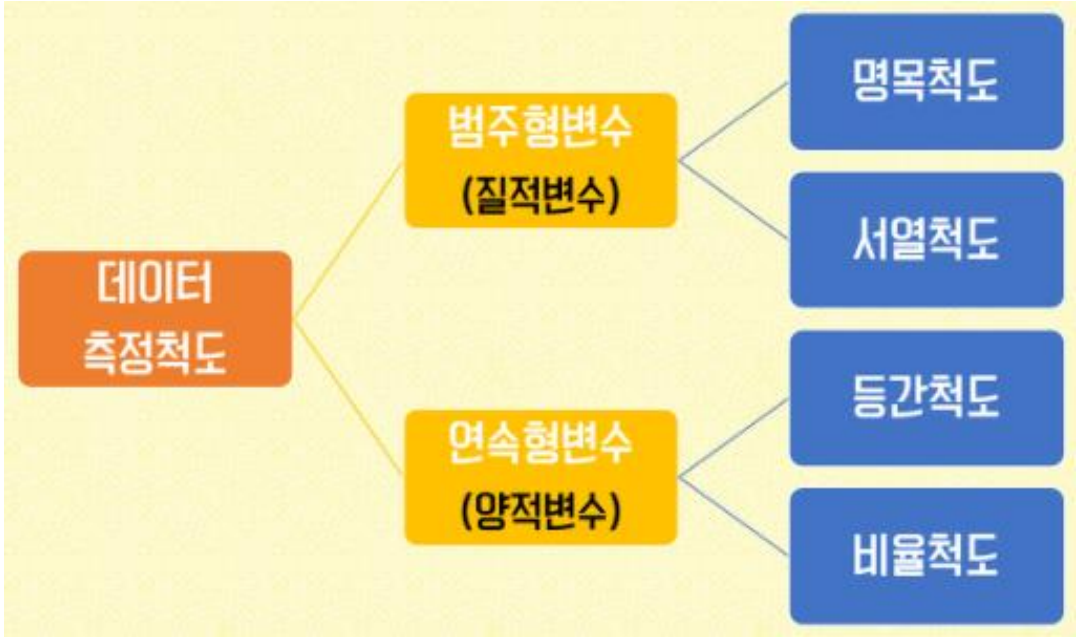
✓ 데이터 스케일링 (scaling)

- 데이터 전처리 과정의 하나
- 데이터 스케일링을 해주는 이유 - **데이터의 값이 너무 크거나 혹은 작은 경우에 모델 알고리즘 학습과정에서 0으로 수렴하거나 무한으로 발산** 해버릴 수 있기 때문
- 데이터 전처리 과정에서 굉장히 중요한 과정

데이터 전처리

변수 분류

- 사이킷런은 문자열 값을 입력 값으로 처리 하지 않기 때문에 숫자 형으로 변환해야 한다
- 척도 - 관측 대상의 속성을 측정 -> 숫자로 나타나도록 일정한 규칙을 적용하여 변환



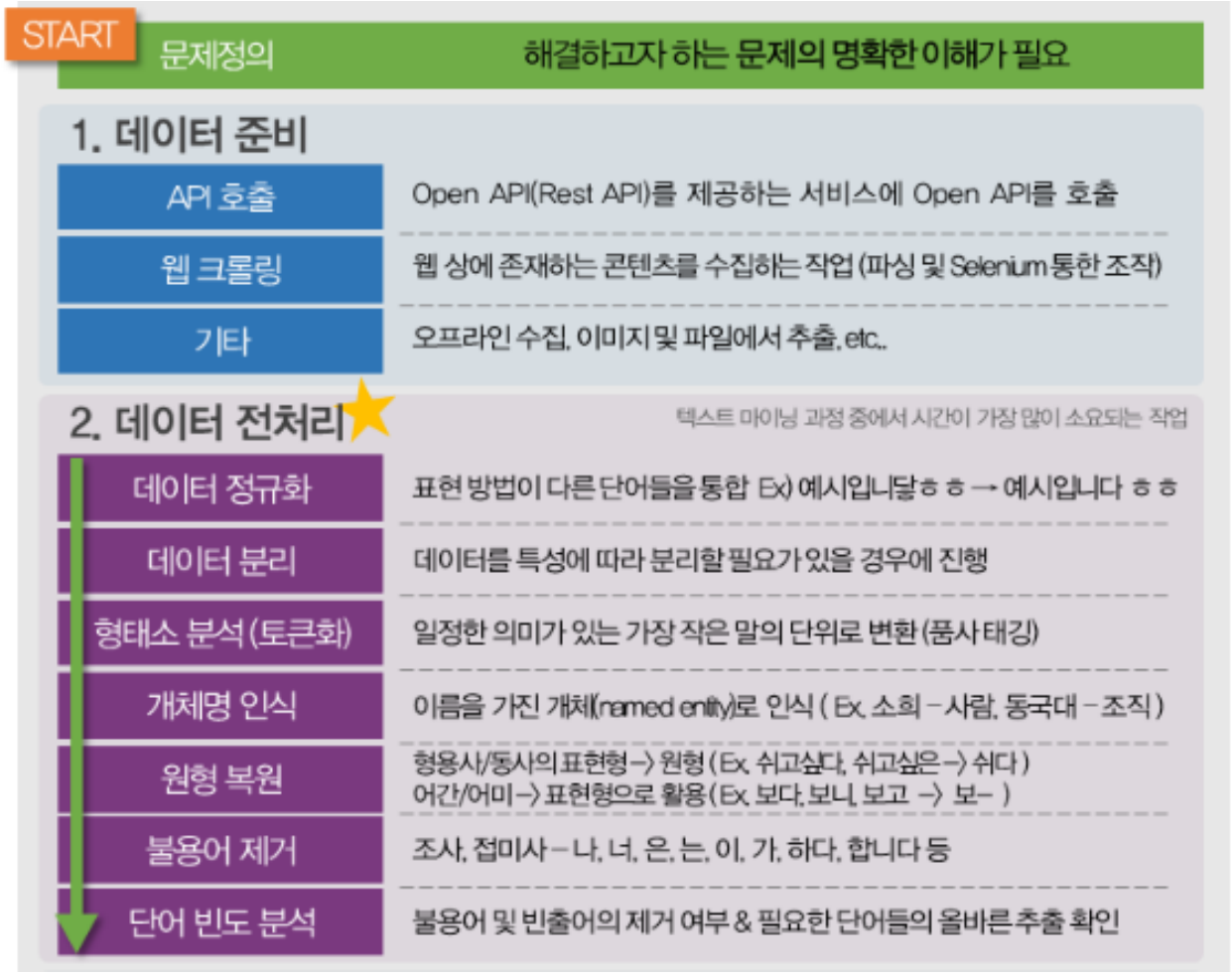
데이터 전처리

변수 분류

명목척도	서열척도	등간척도	비율척도
구분, 분류 목적으로 관측 대상을 숫자를 사용하여 변환 숫자 자체가 가지는 의미는 없음	관측 대상의 속성을 측정해서 값을 순위로 나타낸 것	관측 대상의 속성을 측정해서 상대적인 크기로 나타낸 것	절대적 기준이 있는 영점이 존재, 사칙연산이 가능 대상을 분류 할 수 있고, 차이도 비교할 수 있고, 순위도 만들 수 있음 (명목, 서열, 등간 척도의 성격을 다 가지고 있음)
상호배타적인 특성	값의 크기 비교 가능		
ex) 남자는 1, 여자는 2로 구분	Ex) 성적에 대해 1등, 2등, 3등	ex) 만족도 1점 ~5점	ex) 온도(기온)

텍스트 전처리

➢ 텍스트 마이닝 프로세스



텍스트 전처리

➢ 텍스트 분석

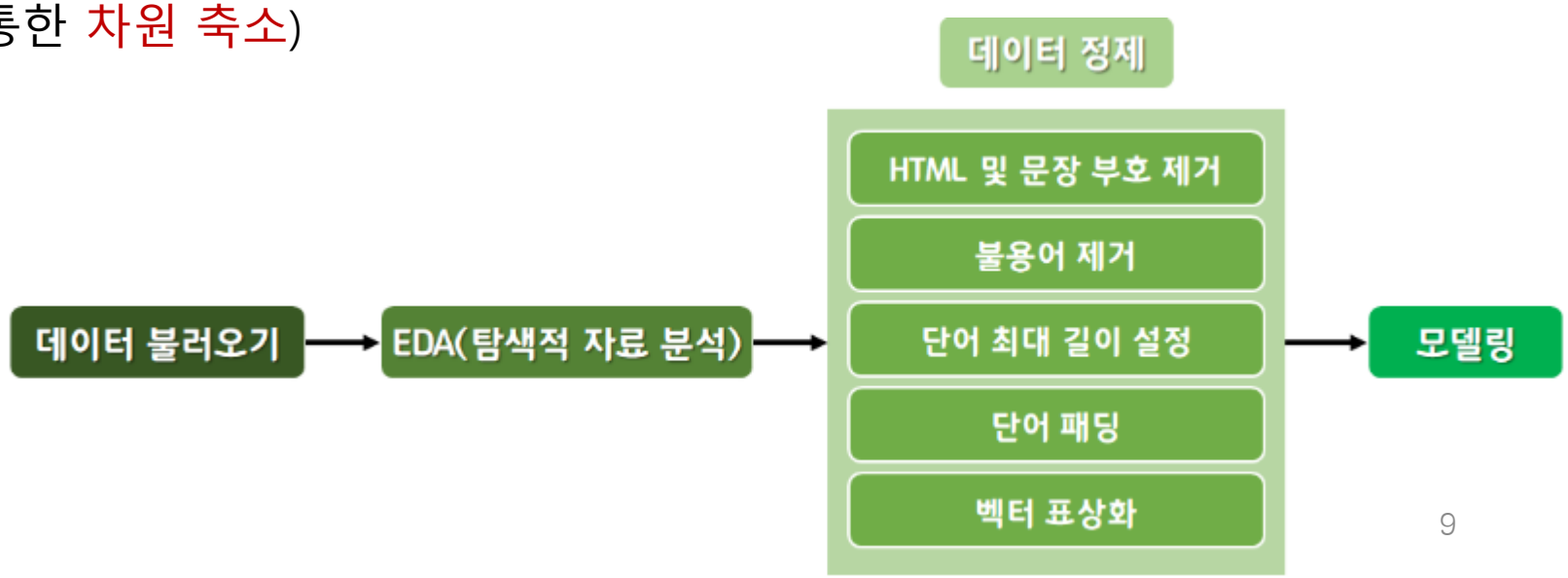
- 비정형 텍스트에서 의미 있는 정보를 추출하는 것
- 문서 분류 (문서가 특정 분류 또는 카테고리에 속하는 것을 예측하는 기법)
- 문서 생성
- 감성 분석 (텍스트에서 나타나는 감정/판단/믿음/의견 등의 주관적인 요소 분석하는 기법)
- 문서 요약 - 텍스트 내에서의 중요한 주제나 중심 사상 추출(Topic Modeling)
- 질의 응답 – 챗봇의 주요 미래 기술
- 기계 번역 – 자연어 이해, sequence to sequence 모형
- 텍스트 군집화(Clustering)와 유사도 측정 - 비슷한 유형의 문서에 대해 군집화를 수행하는 기법
- 토픽 모델링 – 여러 문서에서 공통으로 등장하는 주제를 추출

NLP(Natural Language Processing, 자연어 처리) : 컴퓨터와 인간 언어 간의 상호 작용과 관련된 언어학, 컴퓨터 과학 및 인공 지능의 하위 분야로, 특히 대량의 자연어 데이터를 처리하고 분석할 수 있도록 컴퓨터를 프로그래밍 하는 방법 - Wikipedia

텍스트 전처리

➢ 텍스트 분석을 위한 중요 전처리 단계

- 머신러닝 알고리즘은 텍스트를 의미 있는 숫자로 변환해서 학습시켜야 하므로 핵심이 되는 처리 단계
- 텍스트 정규화 : 클렌징,
  대/소문자 변경,
  특수문자 삭제,
  토큰화 작업,
  의미 없는 단어(Stop word) 제거 작업,
  어근 추출(Stemming/Lemmdatization)
- 텍스트를 number data feature로 벡터화 - Bag of Words (Count 기반 or TF-IDF 기반 벡터화)
- 피처 추출 (특성 선택, 추출을 통한 차원 축소)



텍스트 전처리

➢ 텍스트 전처리 (Text Preprocessing)

- 주어진 텍스트에서 노이즈와 같이 불필요한 부분을 제거하고, 문장을 표준 단어들로 분리한 후에, 각 단어의 품사를 파악하는 것
- 정제(cleaning) : 갖고 있는 코퍼스로부터 분석에 불필요한 노이즈(불필요한 단어) 데이터의 제거
  토큰화 작업에 방해가 되는 분석하고자 하는 목적에 맞지 않는 불필요 단어들을 배제시킴
  등장 빈도가 적은 단어
  길이가 짧은 단어 - 영어 단어의 평균 길이는 6~7 정도이며, 한국어 단어의 평균 길이는 2~3 정도로 추정
  불용어 제거(stopword removal)
- 토큰화 (Tokenization) : 텍스트를 원하는 단위로 나누는 작업, 문장 토큰화(sentence tokenization), 단어 토큰화(word tokenization)
- 정규화 : 같은 의미를 가진 동일한 단어임에도 불구하고 다른 형태로 쓰여진 단어들을 통일시켜서 표준 단어로 만드는 작업
- 품사 Tagging : 단어를 문법적인 기능에 따라 분류한 것

Tag : 중의적인 문제를 해결하기 위해서 말뭉치에 붙이는 중의적인 언어 정보

Tagger : Tagging을 수행하는 프로그램

Tagged Corpus : 태그가 부착된 말뭉치

텍스트 전처리

➢ 텍스트 정제

- 텍스트 데이터는 특성(feature)으로 만들기 전에 정제되어야 합니다
- 텍스트 데이터 정제 - strip, replace, split 등 기본 문자열 메서드를 사용하여 처리

```
text_data = [ "    Interrobang. By Aishwarya Henriette      ",
              "Parking And Going. By Karl Gautier",
              "    Today Is The night. By Jarek Prakash    "]      # 텍스트 데이터 생성

strip_whitespace = [string.strip() for string in text_data]         # 공백 문자 제거
strip_whitespace                                                    # 텍스트 확인
remove_periods = [string.replace(".", "") for string in strip_whitespace]      # 마침표 제거
remove_periods                                                  # 텍스트 확인

def capitalizer(str2 ) :                                        #함수 정의
    return str2.upper()
[capitalizer(string) for string in remove_periods]   # 함수 적용
```

텍스트 전처리

➢ 토큰화

- NLTK : 교육용으로 개발된 자연어 처리 및 문서 분석용 파이썬 패키지

```
import nltk
nltk.download('punkt')
nltk.download('webtext')
nltk.download('wordnet')
nltk.download('stopwords')
nltk.download('averaged_perceptron_tagger')
nltk.download('omw-1.4')
```

```
# 문장 토큰화(sentence tokenize)
para = "Hello everyone. It's good to see you. Let's start our text mining class!"
from nltk.tokenize import sent_tokenize
print(sent_tokenize(para)) #주어진 text를 sentence 단위로 tokenize함. 주로 . ! ? 등을 이용

para_kor = "안녕하세요, 여러분. 만나서 반갑습니다. 이제 텍스트마이닝 클래스를 시작해봅시다!"
print(sent_tokenize(para_kor)) #한국어에 대해서도 sentence tokenizer는 잘 동작함
```

텍스트 전처리

➢ 토큰화

- 자연어 문서를 분석하기 위해서는 긴 문자열을 분석을 위한 작은 단위로 나누어야 한다.
- 토큰(token) : 문자열 단위
- 토큰 생성(tokenizing) : 문자열을 토큰으로 나누는 작업
- 문장 토큰화 - 문장의 마침표, 개행문자(₩n) 등 문장의 마지막을 뜻하는 기호에 따라 분리하는 것
- 단어 토큰화 - 문장을 단어로 토큰화하는 것, 기본적으로 공백, 콤마(,), 마침표(.), 개행문자 등으로 단어를 분리

```
#단어 토큰화 (word tokenize)
from nltk.tokenize import word_tokenize
print(word_tokenize(para)) #주어진 text를 word 단위로 tokenize함

from nltk.tokenize import WordPunctTokenizer
print(WordPunctTokenizer().tokenize(para))

print(word_tokenize(para_kor))
```

텍스트 전처리

➢ 정규표현식을 이용한 토큰화

- NLTK를 사용하지 않고도 다양한 조건에 따라 정규표현식을 이용한 토큰화

```
import re
re.findall("[abc]", "How are you, boy?")  #a, b, c 중에 해당하는 모든 문자를 찾아서 반환
re.findall("[0123456789]", "3a7b5c9d")   #숫자만 반환
re.findall("[₩w]", "3a 7b_ '.^&5c9d")    #[a-zA-Z0-9]의 줄임표현 w
re.findall("[_]+", "a_b, c__d, e___f")     #+는 한번이상 반복
re.findall("[₩w]+", "How are you, boy?")
re.findall("[o]{2,4}", "oh, hoow are yoooou, booooooy?") #{min, max} 지정된 반복 횟수
```

```
from nltk.tokenize import RegexpTokenizer
tokenizer = RegexpTokenizer("[₩w']+") #regular expression(정규식)을 이용한 tokenizer
print(tokenizer.tokenize("Sorry, I can't go there.")) # can't를 하나의 단어로 인식

tokenizer = RegexpTokenizer("[₩w]+")
print(tokenizer.tokenize("Sorry, I can't go there."))

text1 = "Sorry, I can't go there."
tokenizer = RegexpTokenizer("[₩w']{3,}")
print(tokenizer.tokenize(text1.lower()))
```

텍스트 전처리

➢ 노이즈와 불용어 제거

- 정규표현식을 이용한 치환을 통해 원하는 패턴의 노이즈를 제거할 수 있다
- 불용어 : 분석에는 필요가 없는 단어

```
from nltk.corpus import stopwords
english_stops = set(stopwords.words('english'))

text1 = "Sorry, I couldn't go to movie yesterday."

tokenizer = RegexpTokenizer("[₩w']+")
tokens = tokenizer.tokenize(text1.lower())

#stopwords를 제외한 단어들만으로 list를 생성
result = [word for word in tokens if word not in english_stops]
print(result)

#nltk가 제공하는 영어 stopword를 확인
print(english_stops)
```

텍스트 전처리

➢ 노이즈와 불용어 제거

- NLTK의 stopwords : 불용어 리스트, 179개, 토큰화된 단어가 소문자라고 가정함)
- sklearn 라이브러리 : 영어 불용어 리스트를 제공 (318개), frozenset 객체이기 때문에 인덱스를 사용할 수 없다.
- 한국어 불용어 삭제 - 토큰화 후에 조사, 접속사 등을 제거
- 사용자가 정의한 불용어 사전으로부터 불용어를 제거할 수 있다

```
from konlpy.tag import Okt
okt = Okt()
example = "고기를 아무렇게나 구우려고 하면 안 돼. 고기라고 다 같은 게 아니거든. 예컨대 삼겹살을 구울 때는 중요한 게 있지."
stop_words = "를 아무렇게나 구우려고 안 돼 같은 게 구울 때 는"
stop_words = set(stop_words.split(' '))
word_tokens = okt.morphs(example)
result = [word for word in word_tokens if not word in stop_words]

print('불용어 제거 전 :',word_tokens)
print('불용어 제거 후 :',result)
```

텍스트 전처리

➢ 한국어 불용어

-

불용어가 많은 경우에는 코드 내에서 직접 정의하지 않고 txt 파일이나 csv 파일로 정리해놓고 이를 불러와서 사용하기도 합니다.
https://www.ranks.nl/stopwords/korean
https://bit.ly/2vOg4Lu - 영어 전체 불용어
https://bit.ly/2Vs05lN , https://bit.ly/2VKOUnF , https://bit.ly/2J912sv - 한글 불용어

텍스트 전처리

➢ 구두점 삭제

- 구두점 글자들의 딕셔너리를 만들어 translate()에 적용

```
import unicodedata
import sys

text_data = [ 'Hi!!!! I. Love. This. Song....',
              '10000% Agree!!!! #LoveIT',
              'Right?!?!']
# 구두점 문자로 이루어진 딕셔너리를 생성.
punctuation = dict.fromkeys( i for i in range(sys.maxunicode) if
unicodedata.category(chr(i)).startswith('P'))
print(punctuation)

# 문자열의 구두점을 삭제
import string
[string.translate(punctuation) for string in text_data]
```

유니코드에서 P로 시작하는 카테고리는 구두점을 의미합니다. (https://bit.ly/2vNA2of)
아스키 코드의 구두점은 import string; string.punctuation로 얻을 수 있습니다.

텍스트 전처리

➢ 정규화(Normalization)

- 같은 의미를 가진 동일한 단어이면서 다른 형태로 쓰여진 단어들을 통일해 표준 단어로 만드는 작업
- 자연어 처리에서 전처리, 더 정확히는 정규화의 지향점은 갖고 있는 코퍼스로부터 복잡성을 줄이는 일입니다
- 어간 추출(stemming) : 어형이 변형된 단어로부터 접사 등을 제거하고 그 단어의 어간을 분리해내는 작업
- 표제어 추출(lemmatizaiton)
- 단어의 개수를 줄이는 정규화 - 영어권 언어에서 대, 소문자를 통합하는 것 (대부분 대문자를 소문자로 변환)
  대문자와 소문자가 구분되어야 하는 경우 : US, 회사 이름(General Motors)나, 사람 이름(Bush) 등

```
import re
text = "I was wondering if anyone out there could enlighten me on this car."

# 길이가 1~2인 단어들을 정규 표현식을 이용하여 삭제
shortword = re.compile(r'₩W*₩b₩w{1,2}₩b')
print(shortword.sub('', text))
```

텍스트 전처리

➢ 정규화 : 어간 추출(stemming)

- 변화된 단어의 접미사나 어미를 제거하여 같은 의미를 가지는 형태소의 기본형을 찾는 방법
- 형태학적 분석을 단순화한 버전 (예 : tradition과 traditional의 어간은 tradit)
- NLTK 패키지에서 PorterStemmer LancasterStemmer 클래스 제공
- Stemmer 알고리즘은 규칙 기반의 접근을 하고 있으므로 어간 추출 후의 결과에는 사전에 없는 단어들이 포함될 수 있다
- 텍스트 데이터에서 어간을 추출하면 읽기는 힘들어지지만 기본 의미에 가까워지고 샘플 간에 비교하기가 좋다.
- NLTK의 PorterStemmer - 단어의 어미를 제거하여 어간을 바꿀 수 있다.

```
from nltk.stem import PorterStemmer, LancasterStemmer

stemmer1 = PorterStemmer()
stemmer2 =  LancasterStemmer()

words = ["fly", "flies", "flying", "flew", "flown"]
print("Porter Stemmer   :", [stemmer1.stem(w) for w in words])
print("Lancaster Stemmer:", [stemmer2.stem(w) for w in words])
```

텍스트 전처리

- **정규화 : 어간 추출(stemming)**

```
from nltk.tokenize import word_tokenize

para = "Hello everyone. It's good to see you. Let's start our text mining class!"
tokens = word_tokenize(para) #토큰화 실행
print(tokens)

stemmer = PorterStemmer()
result = [stemmer.stem(token) for token in tokens]
print(result)

from nltk.stem import LancasterStemmer
stemmer2 = LancasterStemmer()
result2 = [stemmer2.stem(token) for token in tokens]
print(result2)
```

텍스트 전처리

➢ 정규화 : 표제어 추출(Lemmatization)

- 의미적 관점에서 '단어의 기본형 ' 을 찾는 작업
- 원형 복원(lemmatizing) - 같은 의미를 가지는 여러 단어를 사전형으로 통일하는 작업
- 품사(part of speech)를 지정하는 경우 좀 더 정확한 원형을 찾을 수 있다
- WordNetLemmatizer

```
from nltk.stem import WordNetLemmatizer

lemmatizer = WordNetLemmatizer()
print(lemmatizer.lemmatize('cooking'))
print(lemmatizer.lemmatize('cooking', pos='v')) #품사를 지정
print(lemmatizer.lemmatize('cookery'))
print(lemmatizer.lemmatize('cookbooks'))
from nltk.stem import PorterStemmer
stemmer = PorterStemmer()
print('stemming result:', stemmer.stem('believes'))
print('lemmatizing result:', lemmatizer.lemmatize('believes'))
print('lemmatizing result:', lemmatizer.lemmatize('believes', pos='v'))
```

텍스트 전처리

➢ 정규화 : 표제어 추출(Lemmatization)

표제어(Lemma)는 한글로는 '표제어' 또는 '기본 사전형 단어' 정도의 의미를 갖습니다.
단어들이 다른 형태를 가지더라도, 그 뿌리 단어를 찾아가서 단어의 개수를 줄일 수 있는지 판단합니다.
ex) am, are, is는 서로 다른 스펠링이지만 그 뿌리 단어는 be, 이 단어들의 표제어는 be

규칙에 기반한 알고리즘은 종종 제대로 된 일반화를 수행하지 못 할 수 있습니다.
어간 추출을 하고 나서 일반화가 지나치게 되거나, 또는 덜 되거나 하는 경우입니다.

Stemming	Lemmatization
am → am	am → be
the going → the go	the going → the going
having → hav	having → have

텍스트 전처리

➢ 품사 태깅

- 형태소 : 의미를 가진 가장 작은 말의 단위
- 영어는 각 단어와 형태소가 비교적 명확해서 결과도 정확한 편이나, 한글은 동음이의어가 많고 품사 자체도 복잡해서 매우 어려운 작업에 속하며, 정확한 결과를 기대하기 어렵다
- 품사 : 명사, 대명사, 수자, 조사, 동사, 형용사, 관형사, 부사, 감타사와 같이 공통된 성질을 지닌 낱말끼리 모아 놓은 낱말의 갈래

	품사	설명
체언	명사	이름을 나타내는 낱말
	대명사	이름을 대신해 가리키는 낱말
	수사	수량이나 순서를 가리키는 낱말
관계언	조사	도와주는 낱말
용언	동사	움직임을 나타내는 낱말
	형용사	상태나 성질을 나타내는 낱말
수식언	관형사	체언을 꾸며주는 낱말
	부사	주로 용언을 꾸며 주는 낱말
독립언	감탄사	놀란, 느낌, 부름, 대답을 타나내는 낱말

텍스트 전처리

➢ 품사 태깅

- NLTK는 품사 태그를 위해 구문 주석 말뭉치인 펜 트리뱅크(Eenn Treebank)를 제공
- https://www.ling.upenn.edu/courses/Fall_2003/ling001/penn_treebank_pos.html

```
import nltk
nltk.download('averaged_perceptron_tagger')                    # 태거 다운로드
from nltk import pos_tag
from nltk import word_tokenize

tokens = word_tokenize("Hello everyone. It's good to see you. Let's start our text mining class!")
print(nltk.pos_tag(tokens))
```

- 텍스트가 태깅되면 태그를 사용해 특정 품사를 찾을 수 있습니다.

```
# 태깅된 단어들로부터 태그를 활용해서 명사종류의 단어 필터링
[word for word, tag in text_tagged if tag in ['NN','NNS','NNP','NNPS'] ]
```

태그	품사
NNP	고유 명사, 단수
NN	명사, 단수 또는 불가산 명사
RB	부사
VBD	동사, 과거형
VBG	동사, 동명사 또는 현재 분사
JJ	형용사
PRP	인칭 대명사

텍스트 전처리

➢ 품사 태깅

- nltk.help.upenn_tagset( ) : 품사 약어의 의미와 설명을 볼 수 있다

```
nltk.help.upenn_tagset('CC')
```

- 태그를 사용해 특정 품사의 단어만 추출

```
tokens = word_tokenize("Hello everyone. It's good to see you. Let's start our text mining class!")
my_tag_set = ['NN', 'VB', 'JJ']
my_words = [word for word, tag in nltk.pos_tag(tokens) if tag in my_tag_set]
print(my_words)
```

```
words_with_tag = ['/'.join(item) for item in nltk.pos_tag(tokens)]
print(words_with_tag)
```

텍스트 전처리

➢ 한글 형태소 분석과 품사 태깅

- 음절 : 하나의 종합된 음의 느낌을 주는 말소리의 단위
- 형태소 : '뜻 ' 을 가진 가장 작은 말의 단위
- 단어 : 홀로 쓸 수 있는 말
- 어절 : 문장을 구성하는 각각의 마디로, 띄어쓰기 단위
- 한국어는 5언 9품사의 구조를 가지고 있습니다.
- '동사'와 '형용사'는 어간(stem)과 어미(ending)의 결합으로 구성됩니다.
- 한글 형태소 분석 라이브러리 KoNLPy : Hannanum, Kkma, Komoran, Twitter, Mecab
- http://konlpy.org/ko/latest/install

언	품사
체언	명사, 대명사, 수사
수식언	관형사, 부사
관계언	조사
독립언	감탄사
용언	**동사, 형용사**

morphs(phrase)	텍스트를 형태소 단위로 분리 , 형태소의 리스트 반환
nouns(phrase)	텍스트를 형태소 단위로 분리해서 명사만 반환
pos(phrase)	텍스트를 형태소 단위로 분리하고, 각 형태소에 품사를 부착해 반환

텍스트 전처리

➢ 한글 형태소 분석과 품사 태깅

```
import nltk
from nltk.tokenize import word_tokenize
sentence = '''절망의 반대가 희망은 아니다.
어두운 밤하늘에 별이 빛나듯
희망은 절망 속에 싹트는 거지
만약에 우리가 희망함이 적다면
그 누가 세상을 비출어줄까.
정희성, 희망 공부'''

tokens = word_tokenize(sentence)
print(tokens)
print(nltk.pos_tag(tokens))
```

```
from konlpy.tag import Okt
t = Okt()
print('형태소:', t.morphs(sentence))
print()
print('명사:', t.nouns(sentence))
print()
print('품사 태깅 결과:', t.pos(sentence))
```

텍스트 전처리

➢ 한글 형태소 분석과 품사 태깅

- **활용(conjugation)** : 용언의 어간(stem)이 어미(ending)를 가지는 일
- **어간(stem)** : 용언(동사, 형용사)을 활용할 때, 원칙적으로 모양이 변하지 않는 부분. 활용에서 어미에 선행하는 부분. 때론 어간의 모양도 바뀔 수 있음 (예: 긋다, 긋고, 그어서, 그어라).
- **어미(ending)** : 용언의 어간 뒤에 붙어서 활용하면서 변하는 부분이며, 여러 문법적 기능을 수행
- 활용은 어간이 어미를 취할 때, 어간의 모습이 일정하다면 규칙 활용, 어간이나 어미의 모습이 변하는 불규칙 활용으로 나뉩니다.
- **규칙 활용** : 어간이 어미가 붙기전의 모습과 어미가 붙은 후의 모습이 같으므로, 규칙 기반으로 어미를 단순히 분리해주면 어간 추출이 됩니다.
- **불규칙 활용** : 어간이 어미를 취할 때 어간의 모습이 바뀌거나 취하는 어미가 특수한 어미

잡/어간 + 다/어미

어간의 형식이 달라지는 경우 : '듣-, 돕-, 곱-, 잇-, 오르-, 노랗-' 등이 '듣/들-, 돕/도우-, 곱/고우-, 잇/이-, 올/올-, 노랗/노라-'
일반적인 어미가 아닌 특수한 어미를 취하는 경우 : '오르+ 아/어→올라, 하+아/어→하여, 이르+아/어→이르러, 푸르+아/어→푸르러'

텍스트 전처리

➢ 한글어 형태소 분석

형태소 분석	설명
MeCab	일본어를 위한 엔진 언어, 사전 코퍼스에 의존하지 않는 범용적인 설계 품사 독립적 설계
Hannanum	띄어쓰기가 되지 않은 문장은 분석을 못함 정제된 언어가 사용되지 않는 문서에 대한 형태소 분석 정확도는 낮음 69개의 확장된 카이스트 태그셋을 기본으로 사용 6개의 상위태그와 20개의 태그로 세분화
Kkma	공개 소프트웨어 , 세종 태그셋을 기본으로 사용 띄어쓰기 오류에 덜 민감한 형태소 분석기 분석 시간이 5개의 형태소 분석기 중 가장 길다 정제된 언어가 사용되지 않는 문서에 대한 형태소 분석 정확도는 낮음
Komoran	여러 어절을 하나의 품사로 분석 가능 적용 분야에 따라 공백이 포함된 고유명사를 더 정확하게 분석 자소가 분리된 문장이나 오탈자에 대해서도 괜찮은 분석 품석 품질 다른 형태소 분석기보다 로닝 시간이 길다, 분석 속도 빠름 띄어쓰기 없는 문장 분석 취약함, 미등록어 처리 문제와 동음이의어 처리 문제가 존재함
Okt	띄어쓰기에 가장 좋은 성능을 보여준다 stemming 가능, 토큰화, 정규화 이모티콘이나 해쉬태그등 인터넷 텍스트에 특화된 범주가 추가 되어있다. 이모티콘, 비표준어, 비속어 등이 많이 포함되어 있는 정제되지 않는 데이터에 대해 강점

텍스트 전처리

➢ 텍스트 정수 인코딩(Integer Encoding)

- 단어를 고유한 정수에 맵핑(mapping)시키는 전처리 작업
- BOW(Bag of Words) :
  단어 등장 빈도수를 기준으로 정렬한 뒤에 고유한 정수 인덱스를 부여하는 방법
  문서에서 단어의 사용 여부만 표시하는 방법
  단어가 문서에 나타난 수를 바녕해 보정하는 방법

텍스트 전처리

➢ NLTK(Natural Language Toolkit) 말뭉치

- 말뭉치 (corpus) : 자연언어 연구를 위해 특정한 목적을 가지고 언어의 표본을 추출한 집합
  소설, 신문 등의 문서를 모아놓은 것
  형태소, 등의 보조적 의미를 추가하고 쉬운 분석을 위해 구조적인 형태로 정리해 놓은 것을 포함
- corpus 서브패키지에서는 다양한 연구용 말뭉치를 제공한다.
  설치시에 제공되지 않고 download 명령으로 사용자가 다운로드 받아야 한다

```
import nltk
nltk.download("book", quiet=True)
from nltk.book import *

#저작권이 말소된 문학작품을 포함하는 gutenberg 말뭉치
nltk.corpus.gutenberg.fileids()
```

텍스트 전처리

➢ NLTK 영화 Review Corpus

- movie_reviews

함수	설명
fileids()	영화 리뷰 문서들의 id(fileid)를 반환한다. 매개변수 categories를 이용하면 특정 분류에 속하는 문서들의 id만 가져올 수 있다
categories()	리뷰 문서들에 대한 분류, 즉 라벨을 보여준다. 여기서는 감성을 표현하는 긍정('pos')과 부정('neg') 값을 갖는다
raw()	리뷰 문서의 원문을 문자열의 리스트 형태로 반환한다. 인수로 fileid를 주면 특정 문서만 가져올 수 있다.
sents()	리뷰 문서의 원문에 대해 NLTK의 sent_tokenize로 토큰화한 문장들을 다시 word_tokenize로 토큰화한 결과를 반환한다 인수로 fileid를 주면 특정 문서에 대한 토큰화 결과를 가져올 수 있다
words()	리뷰 문서의 원문에 대해 NLTK의 word_tokenize로 토큰화한 결과를 반환한다. 인수로 fileid를 주면 특정 문서에 대한 토큰화 결과를 가져올 수 있다

텍스트 전처리

➢ NLTK 영화 Review Corpus

- movie_reviews

```
import nltk
nltk.download('movie_reviews')
nltk.download('punkt')
from nltk.corpus import movie_reviews
print('#영화 리뷰 문서 개수 : ', len(movie_reviews.fileids()))
print('#samples of file ids:', movie_reviews.fileids()[:10]) #id를 10개까지만 출력
print('#영화 리뷰 분류값 :', movie_reviews.categories())
print('#label이 부정인 영화 리뷰 문서 수 :', len(movie_reviews.fileids(categories='neg')))
print('# label이 긍정인 영화 리뷰 문서 수 :', len(movie_reviews.fileids(categories='pos')))
fileid = movie_reviews.fileids()[0]   #첫번째 문서의 id를 반환
print('#id of the first review:', fileid)
print('# 첫번째 문서의 내용 200자 : ₩n', movie_reviews.raw(fileid)[:200])
print()
# 첫번째 문서를 sentence tokenize한 결과 중 앞 두 문장
print('#sentence tokenization result:', movie_reviews.sents(fileid)[:2])
#첫번째 문서를 word tokenize한 결과 중 앞 스무 단어
print('#word tokenization result:', movie_reviews.words(fileid)[:20])
```

텍스트 전처리

➢ BOW 기반의 특성 벡터 추출

1. 문서 집합(말뭉치) 토큰화
2. 불용어 제거
3. 정규화
4. 품사 태깅
5. 정한 기준에 따라 단어들을 선택, 특성 집합 생성
6. 각 문서별로 특성 추출 대상 단어들에 대해 단어의 빈도를 계산, 특성 벡터 생성

텍스트 전처리

➢ BOW 기반의 특성 벡터 추출

```
documents = [list(movie_reviews.words(fileid)) for fileid in movie_reviews.fileids()]
print(documents[0][:50]) #첫째 문서의 앞 50개 단어를 출력

# 단어별 말뭉치 전체에서의 빈도를 계산, 빈도가 높은 단어부터 정렬한 후에 상위 10개 단어 출력
word_count = {}
for text in documents:
    for word in text:
        word_count[word] = word_count.get(word, 0) + 1

sorted_features = sorted(word_count, key=word_count.get, reverse=True)
for word in sorted_features[:10]:
    print(f"count of '{word}': {word_count[word]}", end=', ')
```

텍스트 전처리

➢ BOW 기반의 특성 벡터 추출

```
from nltk.tokenize import RegexpTokenizer
from nltk.corpus import stopwords
tokenizer = RegexpTokenizer("[₩w']{3,}")  # 정규포현식으로 토크나이저를 정의
english_stops = set(stopwords.words('english'))
documents = [movie_reviews.raw(fileid) for fileid in movie_reviews.fileids()] #원문을 가져옴
# stopwords의 적용과 토큰화를 동시에 수행.
tokens = [[token for token in tokenizer.tokenize(doc) if token not in english_stops] for doc in documents]

word_count = {}
for text in tokens:
    for word in text:
        word_count[word] = word_count.get(word, 0) + 1

sorted_features = sorted(word_count, key=word_count.get, reverse=True)

print('num of features:', len(sorted_features))
for word in sorted_features[:10]:
    print(f"count of '{word}': {word_count[word]}", end=', ')
```

텍스트 전처리

➢ BOW 기반의 특성 벡터 추출

- 카운트 벡터는 말뭉치 전체의 단어 집합 혹은 선별한 단어 집합에 대한 단어들의 빈도로 이루어져 있으며, 매우 희소하다는 특징이 있다

```
def document_features(document, word_features):
    word_count = {}
    for word in document: #document에 있는 단어들에 대해 빈도수를 먼저 계산
        word_count[word] = word_count.get(word, 0) + 1

    features = []
    for word in word_features: #word_features의 단어에 대해 계산된 빈도수를 feature에 추가
        features.append(word_count.get(word, 0)) #빈도가 없는 단어는 0을 입력
    return features
```

```
word_features_ex = ['one', 'two', 'teen', 'couples', 'solo']
doc_ex = ['two', 'two', 'couples']
print(document_features(doc_ex, word_features_ex))
```

텍스트 전처리

➢ BOW 기반의 특성 벡터 추출

- Movie_reviews 카운트 벡터 생성

```
#영화 리뷰 토큰화된 데이터의 빈도가 높은 상위 1000개의 단어만 추출하여 features를 구성
word_features = sorted_features[:1000]

feature_sets = [document_features(d, word_features) for d in tokens]

# 첫째 feature set의 내용을 앞 20개만 word_features의 단어와 함께 출력
for i in range(20):
    print(f'({word_features[i]}, {feature_sets[0][i]})', end=', ')

print(feature_sets[0][-20:])
```

텍스트 전처리

➢ 사이킷런으로 카운트 벡터 생성

- sklearn.feature_extraction 모듈
- CountVectorizer 클래스

주요 매개변수	설명
tokenizer	함수 형태로 외부 토크나이저를 지정, 지정하지 않으면 자체 토크나이저를 사용한다
stop_words	리스트 형태로 불용어 사전을 지정한다. 'English'로 값을 주면 자체 영어 불용어 사전을 사용한다.
ngram_range	(min_n, max_n)의 튜플 형태로 ngram 범위를 지정한다. 기본값(1, 1)
max_df	단어로 특성을 구성할 때, 문서에 나타난 빈도(document frequency)가 max_df보다 크면 제외한다. 비율 혹은 문서의 수로 지정 가능하다.
min_df	단어로 특성을 구성할 때, 문서에 나타난 빈도(document frequency)가 min_df보다 작으면 제외한다. 비율 혹은 문서의 수로 지정 가능하다
max_features	최대 특성의 수를 지정한다. 지정하지 않으면 전체 단어를 다 사용한다
vocabulary	특성으로 사용할 단어들을 직접 지정한다.
binary	True 값을 주면 빈도 대신 단어의 유무(1/0)로 특성 값을 생성한다.

텍스트 전처리

➢ 사이킷런으로 카운트 벡터 생성

- sklearn은 자체적인 토크나이저를 지원

CountVectorizer 주요 메서드	설명
fit(raw_documents)	인수로 주어진 문서 집합(raw_documents)에 대해 토큰화를 수행하고 특성 집합을 생성한다
transform(raw_documents)	fit()에서 생성한 특성 집합을 이용해 인수로 주어진 문서 집합 (raw_documents)에 대해 카운트벡터로 변환해 반환한다
fit_transform(raw_documents)	인수로 주어진 문서집합(raw_documents)에 대해 fit과 transform을 동시에 수행한다
get_feature_names_out()	특성 집합에 있는 특성의 이름, 즉 단어를 순서대로 반환한다.

텍스트 전처리

➢ 사이킷런으로 카운트 벡터 생성

```
reviews = [movie_reviews.raw(fileid) for fileid in movie_reviews.fileids()]

from sklearn.feature_extraction.text import CountVectorizer
cv = CountVectorizer(vocabulary=word_features)
#cv = CountVectorizer(max_features=1000) #특성 집합을 지정하지 않고 최대 특성의 수를 지정하는 경우
print(cv) #객체에 사용된 인수들을 확인

reviews_cv = cv.fit_transform(reviews) #reviews를 이용하여 count vector를 학습하고, 변환
print(cv.get_feature_names_out()[:20]) # count vector에 사용된 feature 이름을 반환
print(word_features[:20]) # 비교를 위해 출력

print('#type of count vectors:', type(reviews_cv))
print('#shape of count vectors:', reviews_cv.shape)
print('#sample of count vector:')
print(reviews_cv[0, :10])
print(reviews_cv)
```

텍스트 전처리

➢ 사이킷런으로 카운트 벡터 생성

```
print(feature_sets[0][:20]) #절 앞에서 직접 계산한 카운트 벡터
print(reviews_cv.toarray()[0, :20]) #변환된 결과의 첫째 feature set 중에서 앞 20개를 출력

for word, count in zip(cv.get_feature_names_out()[:20], reviews_cv[0].toarray()[0, :20]):
    print(f'{word}:{count}', end=', ')
```

텍스트 전처리

➢ 한국어 텍스트의 카운트 벡터 변환

- 네이버 영화 리뷰 (텍스트 파일로 제공)
  https://github.com/e9t/nsmc
- 네이버 뉴스 분류, NNST 파이썬 라이브러리 형태로 제공
  https://github.com/jason9693/NNST-Naver-News-for-Standard-and-Technology-Database
- 네이버 뉴스 분류, 데이터 파일로 제공
  https://github.com/cranberryai/todak_todak_python/raw/master/machine_learning_text/naver_news/newData.zip

```
import pandas as pd
df = pd.read_csv('./data/daum_movie_review.csv')
df.head(10)

from sklearn.feature_extraction.text import CountVectorizer
daum_cv = CountVectorizer(max_features=1000)

daum_DTM = daum_cv.fit_transform(df.review) #review를 이용하여 count vector를 학습하고, 변환
print(daum_cv.get_feature_names_out()[:100]) # count vector에 사용된 feature 이름을 반환
```

텍스트 전처리

➢ 한국어 텍스트의 카운트 벡터 변환

```
from konlpy.tag import Okt #konlpy에서 Twitter 형태소 분석기를 import
twitter_tag = Okt()

print('#전체 형태소 결과:', twitter_tag.morphs(df.review[1]))
print('#명사만 추출:', twitter_tag.nouns(df.review[1]))
print('#품사 태깅 결과', twitter_tag.pos(df.review[1]))

def my_tokenizer(doc):
    return [token for token, pos in twitter_tag.pos(doc) if pos in ['Noun', 'Verb', 'Adjective']]

print("나만의 토크나이저 결과:", my_tokenizer(df.review[1]))
```

텍스트 전처리

➢ 한국어 텍스트의 카운트 벡터 변환

- DTM(Document Term Matrix) : 문서를 행으로, 단어를 열로 해서 단어의 빈도를 나타낸 행렬
- TDM(Term Document Matrix) : DTM의 전치행렬(행과 열을 바꾼 것으로 대각선을 축으로 반사시킨 결과)

```
from sklearn.feature_extraction.text import CountVectorizer

daum_cv = CountVectorizer(max_features=1000, tokenizer=my_tokenizer)
#명사만 추출하고 싶은 경우에는 tokenizer에 'twitter_tag.nouns'를 바로 지정해도 됨

daum_DTM = daum_cv.fit_transform(df.review) #review를 이용하여 count vector를 학습하고, 변환
print(daum_cv.get_feature_names_out()[:100]) # count vector에 사용된 feature 이름을 반환

print(repr(daum_DTM))
print(110800/(14725*1000))

for word, count in zip(daum_cv.get_feature_names_out(), daum_DTM[1].toarray()[0]):
    if count > 0:
        print(word, ':', count, end=', ')
```

텍스트 전처리

➢ 카운트 벡터 활용

- 문서의 특성을 표현하는 벡터는 문서 간의 유사도를 측정하는데 사용할 수 있다
- 코사인 유사도(cosine similarity) : 두 벡터가 이루는 각도의 코사인값으로 정의
- 벡터의 방향성만 비교
- 두 벡터가 가장 가까우면 유사도1, 가장 먼 경우 유사도 0
- sklearn.metrics.pairwise.cosine_similarity() : 다수 벡터와 다수 벡터 간의 유사도를 한 번에 계산하고 그 결과를 행렬로 반환

```
from sklearn.metrics.pairwise import cosine_similarity

start = len(reviews[0]) // 2 #첫째 리뷰의 문자수를 확인하고 뒤 절반을 가져오기 위해 중심점을 찾음
source = reviews[0][-start:] #중심점으로부터 뒤 절반을 가져와서 비교할 문서를 생성
source_cv = cv.transform([source]) #코사인 유사도는 카운트 벡터에 대해 계산하므로 벡터로 변환
#transform은 반드시 리스트나 행렬 형태의 입력을 요구하므로 리스트로 만들어서 입력
print("#대상 특성 행렬의 크기:", source_cv.shape) #행렬의 크기를 확인, 문서가 하나이므로 (1, 1000)
sim_result = cosine_similarity(source_cv, reviews_cv) #변환된 count vector와 기존 값들과의 similarity 계산
print("#유사도 계산 행렬의 크기:", sim_result.shape)
print("#유사도 계산결과를 역순으로 정렬:", sorted(sim_result[0], reverse=True)[:10])
```

텍스트 전처리

➢ 카운트 벡터 활용

```
import numpy as np
print('#가장 유사한 리뷰의 인덱스:', np.argmax(sim_result[0]))

print('#가장 유사한 리뷰부터 정렬한 인덱스:', (-sim_result[0]).argsort()[:10])
```

텍스트 전처리

➢ TF-IDF (Term Frequency – Inverse Document Frequency)

- 단어의 빈도와 역 문서 빈도(문서의 빈도에 특정 식을 취함)를 사용하여 DTM 내의 각 단어들마다 중요한 정도를 가중치로 주는 방법
- TF-IDF는 주로 문서의 유사도를 구하는 작업, 검색 시스템에서 검색 결과의 중요도를 정하는 작업, 문서 내에서 특정 단어의 중요도를 구하는 작업 등에 쓰일 수 있습니다
- DTM을 만든 후, TF-IDF 가중치를 부여합니다.
- TF-IDF는 TF와 IDF를 곱한 값을 의미
- sklearn.feature_extraction.text.TfidfVectorizer

```
from sklearn.feature_extraction.text import TfidfTransformer
transformer = TfidfTransformer()

reviews_tfidf = transformer.fit_transform(reviews_cv)
#TF-IDF 행렬의 모양과 카운트 행렬의 모양이 일치하는 것을 확인
print('#shape of tfidf matrix:', reviews_tfidf.shape )
print('#20 count score of the first review:', reviews_cv[0].toarray()[0][:20])
print('#20 tfidf score of the first review:', reviews_tfidf[0].toarray()[0][:20])
```

텍스트 전처리

➢ TF-IDF (Term Frequency – Inverse Document Frequency)

$$idf(t) = log(\frac{n}{1+df(t)})$$

```
from sklearn.feature_extraction.text import TfidfVectorizer

tf = TfidfVectorizer(vocabulary=word_features)
reviews_tf = tf.fit_transform(reviews)

source_tf = tf.transform([source]) #코사인 유사도는 카운트 벡터에 대해 계산하므로 벡터로 변환
#transform은 반드시 리스트나 행렬 형태의 입력을 요구하므로 리스트로 만들어서 입력

sim_result_tf = cosine_similarity(source_tf, reviews_tf) #변환된 count vector와 기존 값들과의
similarity 계산

print('#가장 유사한 리뷰의 인덱스:', np.argmax(sim_result_tf[0]))

print('#카운트 벡터에 대해 가장 유사한 리뷰부터 정렬한 인덱스:', (-sim_result[0]).argsort()[:10])
print('#TF-IDF 벡터에 대해 가장 유사한 리뷰부터 정렬한 인덱스:', (-sim_result_tf[0]).argsort()[:10])
```

텍스트 전처리

➢ 품사 태깅

- 특별한 주제(예: 의료)에 대한 영어 텍스트가 아니라면 사전 훈련된 NLTK의 품사 태깅을 사용합니다
- pos_tag의 정확도가 매우 낮다면 NLTK를 사용하여 자신만의 태그 모델을 훈련시킬 수 있다.
- 문맥적인 의미가 중요한 경우 단어의 순서를 가지도록 토큰화 방법 : n-gram
- 백오프 n-그램 태그 모델 : n은 한 단어의 품사를 예측하기 위해 고려할 이전 단어의 수
- BigramTagger(이전 한단어를 고려), TrigramTagger(이전 두 단어를 고려), UnigramTagger (그 단어 자체만 참고)

```
import nltk
nltk.download('brown')            # 브라운 코퍼스를 다운로드
from nltk.corpus import brown
from nltk.tag import UnigramTagger , BigramTagger,  TrigramTagger
# 브라운 코퍼스에서 텍스트를 추출한 다음 문장으로 나눕니다.
sentences = brown.tagged_sents(categories='news')
train = sentences[:4000]                    # 4,000개의 문장은 훈련용
test = sentences[4000:]                      #623개는 테스트용으로 나눕니다
unigram = UnigramTagger(train)  # 백오프 태그 객체 생성
bigram = BigramTagger(train, backoff=unigram)
trigram = TrigramTagger(train, backoff=bigram)
trigram.evaluate(test)  # 정확도를 확인
```

텍스트 전처리

➢ 텍스트 정수 인코딩(Integer Encoding)

- MultiLabelBinarizer는 class의 종류(개수) 인식과 multi-label encoding을 수행한다.
- 트윗 문장을 각 품사에 따라 특성으로 변환 (명사가 있을 경우 1, 그렇지 않으면 0)
- classes_ - 각 특성의 품사 확인

```
from sklearn.preprocessing import MultiLabelBinarizer

tweets = [ "I am eating a burrito for breakfast",
           "Political science is an amazing field",
           "San Francisco is an awesome city"]
tagged_tweets = []

# 각 단어와 트윗을 태깅합니다.
for tweet in tweets:
    tweet_tag = nltk.pos_tag(word_tokenize(tweet))
    tagged_tweets.append([tag for word, tag in tweet_tag])

# 원-핫 인코딩을 사용하여 태그를 특성으로 변환
one_hot_multi = MultiLabelBinarizer()
one_hot_multi.fit_transform(tagged_tweets)
one_hot_multi.classes_
```

텍스트 전처리

➢ 텍스트 정수 인코딩(Integer Encoding)

- NLTK에서 빈도수 계산 도구인 FreqDist()를 지원

```
from nltk import FreqDist
import numpy as np

# np.hstack으로 문장 구분을 제거
vocab = FreqDist(np.hstack(preprocessed_sentences))
print(vocab["barber"])          # 'barber'라는 단어의 빈도수 출력

vocab_size = 5
vocab = vocab.most_common(vocab_size)          # 등장 빈도수가 높은 상위 5개의 단어만 저장
print(vocab)
word_to_index = {word[0] : index + 1 for index, word in enumerate(vocab)}
print(word_to_index)
```

```
# enumerate()는 순서가 있는 자료형(list, set, tuple, dictionary, string)을 입력으로 받아 인덱스를 순차적으로 함께 리턴한다
test_input = ['a', 'b', 'c', 'd', 'e']
for index, value in enumerate(test_input):          # 입력의 순서대로 0부터 인덱스를 부여함.
    print("value : {}, index: {}".format(value, index))
```

텍스트 전처리

➢ 텍스트 정수 인코딩(Integer Encoding)

- 케라스(Keras)의 텍스트 전처리

```
from tensorflow.keras.preprocessing.text import Tokenizer

preprocessed_sentences = [['barber', 'person'], ['barber', 'good', 'person'], ['barber', 'huge', 'person'], ['knew', 'secret'], ['secret', 'kept', 'huge', 'secret'], ['huge', 'secret'], ['barber', 'kept', 'word'], ['barber', 'kept', 'word'], ['barber', 'kept', 'secret'], ['keeping', 'keeping', 'huge', 'secret', 'driving', 'barber', 'crazy'], ['barber', 'went', 'huge', 'mountain']]
tokenizer = Tokenizer()
# fit_on_texts()안에 코퍼스를 입력으로 하면 빈도수를 기준으로 단어 집합을 생성 (빈도수가 높은 순으로 낮은 정수 인덱스를 부여)
tokenizer.fit_on_texts(preprocessed_sentences)
print(tokenizer.word_index)  #단어별 인덱스
Print(tokenizer.word_counts)  # 단어별 빈도수
print( tokenizer.texts_to_sequences(preprocessed_sentences) )  #문장을 단어의 인덱스로 정수 변환
```

텍스트 전처리

➢ 패딩(Padding)

- 여러 문장의 길이가 전부 동일한 문서들에 대해서는 하나의 행렬로 보고, 한꺼번에 묶어서 병렬 연산 처리할 수 있습니다.

```
import numpy as np
from tensorflow.keras.preprocessing.text import Tokenizer
preprocessed_sentences = [['barber', 'person'], ['barber', 'good', 'person'], ['barber', 'huge', 'person'], ['knew',
'secret'], ['secret', 'kept', 'huge', 'secret'], ['huge', 'secret'], ['barber', 'kept', 'word'], ['barber', 'kept', 'word'],
['barber', 'kept', 'secret'], ['keeping', 'keeping', 'huge', 'secret', 'driving', 'barber', 'crazy'], ['barber', 'went',
'huge', 'mountain']]
tokenizer = Tokenizer()
tokenizer.fit_on_texts(preprocessed_sentences)
encoded = tokenizer.texts_to_sequences(preprocessed_sentences)
print(encoded)
max_len = max(len(item) for item in encoded)
print('최대 길이 :',max_len)
for sentence in encoded:
    while len(sentence) < max_len:
        sentence.append(0)                    #제로 패딩(zero padding)

padded_np = np.array(encoded)
padded_np
```

텍스트 전처리

➢ 패딩(Padding)

- 케라스에서는 패딩을 위해 pad_sequences()를 제공
- pad_sequences는 기본적으로 문서의 앞에 0을 채움
- 숫자 0이 아니라 다른 숫자를 패딩을 위한 숫자로 사용 가능합니다.

```
from tensorflow.keras.preprocessing.sequence import pad_sequences

encoded = tokenizer.texts_to_sequences(preprocessed_sentences)
print(encoded)
padded = pad_sequences(encoded)
print(padded)
## 뒤에 0을 채우고 싶다면 인자로 padding='post'를 주면됩니다.
padded = pad_sequences(encoded, padding='post')
print(padded)
(padded == padded_np).all() Numpy를 이용하여 패딩을 했을 때와 결과가 동일한지 두 결과를 비교합니다.
#길이에 제한을 두고 패딩 - maxlen의 인자로 정수를 주면, 해당 정수로 모든 문서의 길이를 동일하게 합니다.
padded = pad_sequences(encoded, padding='post', maxlen=5)
print(encoded)     #길이가 5보다 짧은 문서들은 0으로 패딩되고, 기존에 5보다 길었다면 데이터가 손실됩니다.
#데이터가 손실될 경우에 앞의 단어가 아니라 뒤의 단어가 삭제되도록 하고싶다면 truncating='post'를 사용
padded = pad_sequences(encoded, padding='post', truncating='post', maxlen=5)
print(encoded)
```

텍스트 전처리

➢ 원-핫 인코딩(One-Hot Encoding)

- 서로 다른 단어들의 단어 집합(vocabulary)에서는 기본적으로 book과 books와 같이 단어의 변형 형태도 다른 단어로 간주합니다.
- 원-핫 벡터(One-Hot vector) : 단어 집합의 크기를 벡터의 차원으로 하고, 표현하고 싶은 단어의 인덱스에 1의 값을 부여하고, 다른 인덱스에는 0을 부여하는 단어의 벡터 표현 방식
  첫째, 정수 인코딩을 수행 (각 단어에 고유한 정수를 부여합니다.)
  둘째, 표현하고 싶은 단어의 고유한 정수를 인덱스로 간주하고 해당 위치에 1을 부여하고, 다른 단어의 인덱스의 위치에는 0을 부여합니다.

```
from konlpy.tag import Okt
okt = Okt()

tokens = okt.morphs("나는 자연어 처리를 배운다")
print(tokens)
word_to_index = {word : index for index, word in enumerate(tokens)}
print('단어 집합 :',word_to_index)
def one_hot_encoding(word, word_to_index):
    one_hot_vector = [0]*(len(word_to_index))
    index = word_to_index[word]
    one_hot_vector[index] = 1
    return one_hot_vector
one_hot_encoding("자연어", word_to_index)
```

텍스트 전처리

➢ 케라스(Keras)를 이용한 원-핫 인코딩(One-Hot Encoding)

- 케라스는 정수 인코딩 된 결과로부터 원-핫 인코딩을 수행하는 to_categorical()를 지원합니다.

```
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.utils import to_categorical

text = "나랑 점심 먹으러 갈래 점심 메뉴는 햄버거 갈래 갈래 햄버거 최고야"

tokenizer = Tokenizer()
tokenizer.fit_on_texts([text])
print('단어 집합 :',tokenizer.word_index)

sub_text = "점심 먹으러 갈래 메뉴는 햄버거 최고야"
encoded = tokenizer.texts_to_sequences([sub_text])[0]
# 단어 집합(vocabulary)에 있는 단어들로만 구성된 텍스트가 있다면, texts_to_sequences()를 통해서 이를 정수 시퀀스로 변환가능합니다.
print(encoded)
one_hot = to_categorical(encoded)
print(one_hot)
```

텍스트 전처리

➢ 원-핫 인코딩(One-Hot Encoding)의 한계

- 단어의 개수가 늘어날 수록, 벡터를 저장하기 위해 필요한 공간이 계속 늘어난다
  원 핫 벡터는 단어 집합의 크기가 곧 벡터의 차원 수가 됩니다.
- 단어의 유사도를 표현하지 못한다
- 단어의 잠재 의미를 반영하여 다차원 공간에 벡터화 하는 기법 :
  => 카운트 기반의 벡터화 방법인 LSA(잠재 의미 분석), HAL
  => 예측 기반으로 벡터화하는 NNLM, RNNLM, Word2Vec, FastText 등