9장
자연어 전처리

9장 자연어 전처리

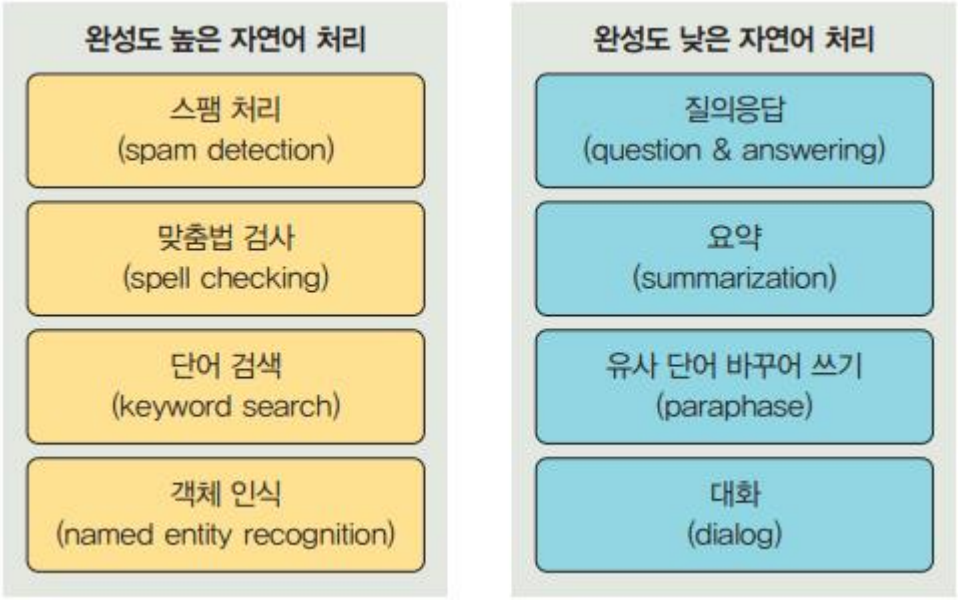
9.1 자연어 처리란

- **자연어 처리란**
  - 자연어 처리란 우리가 일상생활에서 사용하는 언어 의미를 분석하여 컴퓨터가 처리할 수 있도록 하는 과정
  - 자연어 처리는 딥러닝에 대한 이해도 필요하지만, 그에 앞서 인간 언어에 대한 이해도 필요하기 때문에 접근하기 어려운 분야
  - 또한, 언어 종류가 다르고 그 형태가 다양하기 때문에 처리가 매우 어려움
  - 예를 들어 영어는 명확한 띄어쓰기가 있지만, 중국어는 띄어쓰기가 없기 때문에 단어 단위의 임베딩이 어려움

9.1 자연어 처리란

- **자연어 처리란**
  - 다음 그림은 자연어 처리가 가능한 영역과 발전이 필요한 분야
  - 예를 들어 스팸 처리 및 맞춤법 검사는 완성도가 높은 반면, 질의응답 및 대화는 아직 발전이 더 필요한 분야

▼ **그림 9-1 자연어 처리 완성도**



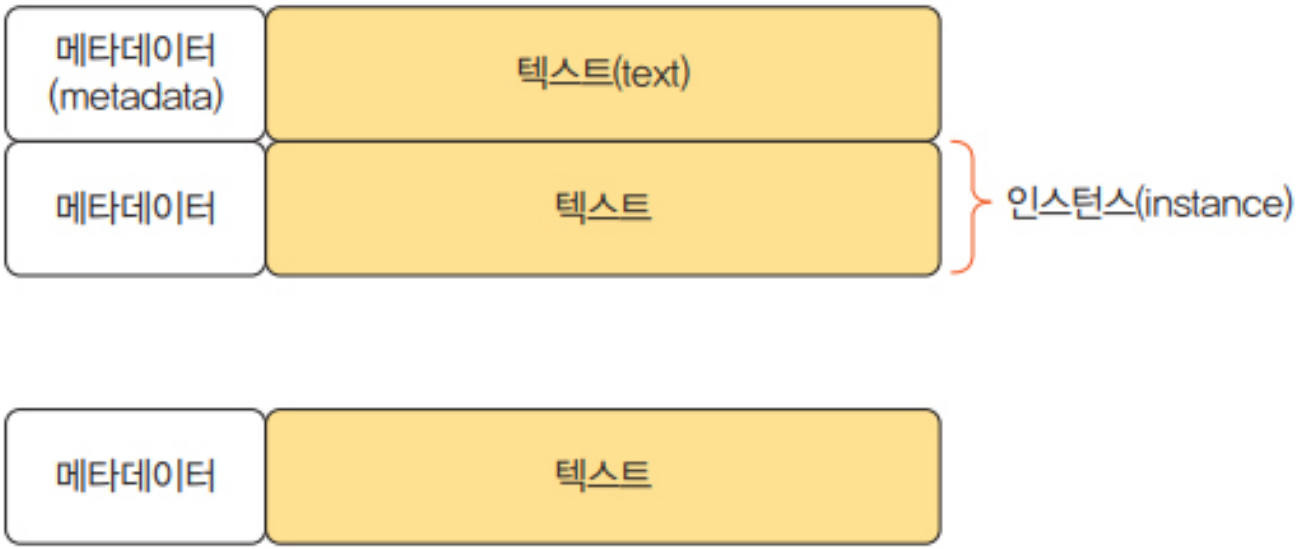
9.1 자연어 처리란

- **자연어 처리 용어 및 과정**

  자연어 처리 관련 용어

  - **말뭉치(corpus(코퍼스))**: 자연어 처리에서 모델을 학습시키기 위한 데이터. 자연어 연구를 위해 특정한 목적에서 표본을 추출한 집합

▼ **그림 9-2 말뭉치(corpus)**



9.1 자연어 처리란

- **자연어 처리 용어 및 과정**
  - **토큰**(token): 자연어 처리를 위한 문서는 작은 단위로 나누어야 하는데, 이때 문서를 나누는 단위가 토큰
    문자열을 토큰으로 나누는 작업을 토큰 생성(tokenizing)이라고 하며,
    문자열을 토큰으로 분리하는 함수를 토큰 생성 함수라고 함
  - **토큰화**(tokenization): 텍스트를 문장이나 단어로 분리하는 것을 의미
    토큰화 단계를 마치면 텍스트가 단어 단위로 분리
  - **불용어**(stop words): 문장 내에서 많이 등장하는 단어
    분석과 관계없으며, 자주 등장하는 빈도 때문에 성능에 영향을 줌. 사전에 제거해 주어야 함. 불용어 예로 “a”, “the”, “she”, “he” 등이 있음

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

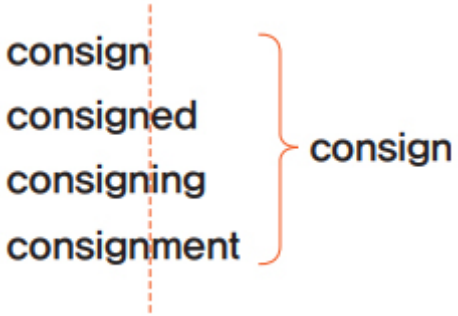
불용어 제거

```
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize
from konlpy.tag import Okt

example = "Family is not an important thing. It's everything."
stop_words = set(stopwords.words('english'))

word_tokens = word_tokenize(example)

result = []
for word in word_tokens:
    if word not in stop_words:
        result.append(word)

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

9.1 자연어 처리란

- **자연어 처리 용어 및 과정**
  - **어간 추출**(stemming): 단어를 기본 형태로 만드는 작업

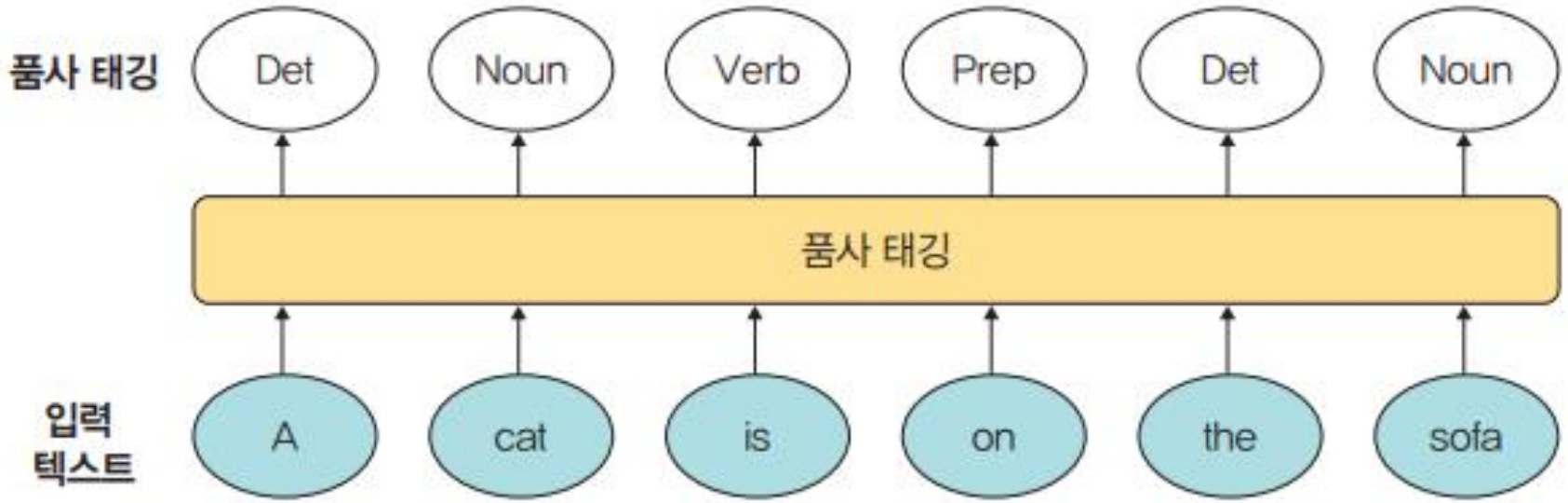
▼ **그림 9-3 어간 추출**



9.1 자연어 처리란

- **자연어 처리 용어 및 과정**
  - **품사 태깅**(part-of-speech tagging): 주어진 문장에서 품사를 식별하기 위한 태그(식별 정보)를 의미

▼ **그림 9-4 품사 태깅**



9.1 자연어 처리란

- **자연어 처리 용어 및 과정**
  - 품사 태깅을 위한 정보는 다음과 같음

    - **Det**: 한정사 (determiner)
    - **Noun**: 명사
    - **Verb**: 동사
    - **Prep**: 전치사 (Preposition)

  - 품사 태깅은 NLTK를 이용할 수 있음

```
> pip install nltk
```

  - Preposition, conjunction, interjection

9.1 자연어 처리란

- **자연어 처리 용어 및 과정 - anaconda**
  - 품사 태깅을 위해 주어진 문장에 대해 토큰화를 먼저 진행
  - 다음 코드를 실행하면 NLTK Downloader 창이 뜸
  - **Download**를 눌러 내려받음

코드 9-1 문장 토큰화

```
import nltk
nltk.download()
text = nltk.word_tokenize("Is it possible distinguishing cats and dogs")
text
```

9.1 자연어 처리란

```
[10] !pip install nltk
```

```
[13] import nltk
     nltk.download('treebank')
```

```
[nltk_data] Downloading package treebank to /root/nltk_data...
[nltk_data]   Unzipping corpora/treebank.zip.
True
```

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

```
[nltk_data] Downloading package averaged_perceptron_tagger to
[nltk_data]     /root/nltk_data...
[nltk_data]   Unzipping taggers/averaged_perceptron_tagger.zip.
True
```

9.1 자연어 처리란

- **자연어 처리 용어 및 과정**
  - 내려받은 자원을 이용하여 품사를 태깅

코드 9-3 품사 태깅

```
nltk.pos_tag(text)
```

```
[('Is', 'VBZ'),
 ('it', 'PRP'),
 ('possible', 'JJ'),
 ('distinguishing', 'VBG'),
 ('cats', 'NNS'),
 ('and', 'CC'),
 ('dogs', 'NNS')]
```

9.1 자연어 처리란

- **자연어 처리 용어 및 과정**
  - 다음은 품사 태깅에 대한 출력 결과

각 단어의 품사는 다음과 같습니다:
•"Is": verb (동사)
•"it": pronoun (대명사)
•"possible": adjective (형용사)
• " cats " : noun (명사)
•"and": conjunction (접속사)
•"dogs": noun (복수형)

```
[('Is', 'VBZ'),
 ('it', 'PRP'),
 ('possible', 'JJ'),
 ('distinguishing', 'VBG'),
 ('cats', 'NNS'),
 ('and', 'CC'),
 ('dogs', 'NNS')]
```

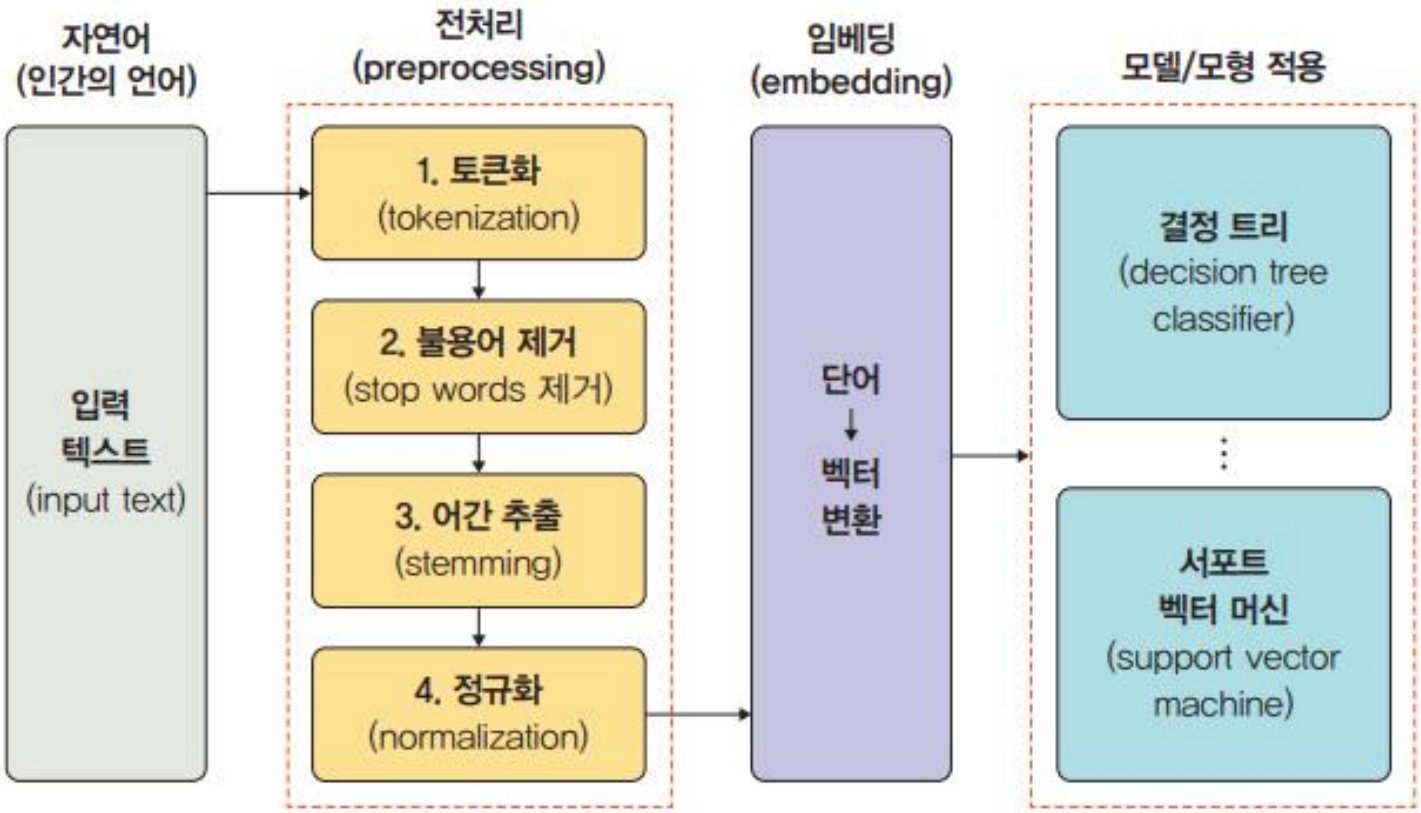
9.1 자연어 처리란

- **자연어 처리 용어 및 과정**
  - 여기에서 사용되는 품사 의미는 다음과 같음

    - **VBZ**: 동사, 동명사 또는 현재 분사
    - **PRP**: 인칭 대명사(PP)
    - **JJ**: 형용사
    - **VBG**: 동사, 동명사 또는 현재 분사
    - **NNS**: 명사, 복수형
    - **CC**: 등위 접속사

9.1 자연어 처리란

▼ 그림 9-6 자연어 처리 과정



9.1 자연어 처리란

- **자연어 처리를 위한 라이브러리**

  NLTK

  - NLTK(Natural Language ToolKit)는 교육용으로 개발된 자연어 처리 및 문서 분석용 파이썬 라이브러리
  - 다양한 기능 및 예제를 가지고 있으며 실무 및 연구에서도 많이 사용되고 있음
  - 다음은 NLTK 라이브러리가 제공하는 주요 기능

    - 말뭉치
    - 토큰 생성
    - 형태소 분석
    - 품사 태깅

9.1 자연어 처리란

- **자연어 처리를 위한 라이브러리**
  - 설치한 NLTK 라이브러리를 이용하여 예제를 살펴보자

코드 9-4 nltk 라이브러리 호출 및 문장 정의

```
import nltk
nltk.download('punkt') ------ 문장을 단어로 쪼개기 위한 자원 내려받기
string1 = "my favorite subject is math"
string2 = "my favorite subject is math, english, economic and computer science"
nltk.word_tokenize(string1)
```

9.1 자연어 처리란

- **자연어 처리를 위한 라이브러리**
  - 다음은 string1에 대해 문장을 단어로 쪼갠 결과

```
['my', 'favorite', 'subject', 'is', 'math']
```

```
['my',
 'favorite',
 'subject',
 'is',
 'math',
 ',',
 'english',
 ',',
 'economic',
 'and',
 'computer',
 'science']
```

9.1 자연어 처리란

- **자연어 처리를 위한 라이브러리**

  KoNLPy

  - KoNLPy(코엔엘파이라고 읽음)는 한국어 처리를 위한 파이썬 라이브러리
  - KoNLPy는 파이썬에서 사용할 수 있는 오픈 소스 형태소 분석기로, 기존에 공개된 꼬꼬마(Kkma), 코모란(Komoran), 한나눔(Hannanum), 트위터(Twitter), 메카브(Mecab) 분석기를 한 번에 설치하고 동일한 방법으로 사용할 수 있도록 해 줌

  - https://konlpy-ko.readthedocs.io/ko/v0.4.3/morph/
  - 출처: 형태소 분석 및 품사 태깅

품사 태깅 클래스 간 비교

이제 *tag Package* 에 있는 태거들의 성능을 확인해볼까요? 실험은 4개의 코어가 있는 인텔 i7 CPU, 파이썬 2.7, KoNLPy 0.4.1을 이용해 수행되었습니다.

Time analysis [1]

1. 로딩 시간: 사전 로딩을 포함하여 클래스를 로딩하는 시간.
    - **Kkma**: 5.6988 *secs*
    - **Komoran**: 5.4866 *secs*
    - **Hannanum**: 0.6591 *secs*
    - **Twitter**: 1.4870 *secs*
    - **Mecab**: 0.0007 *secs*
2. 실행시간: 10만 문자의 문서를 대상으로 각 클래스의 `pos` 메소드를 실행하는데 소요되는 시간.
    - **Kkma**: 35.7163 *secs*
    - **Komoran**: 25.6008 *secs*
    - **Hannanum**: 8.8251 *secs*
    - **Twitter**: 2.4714 *secs*
    - **Mecab**: 0.2838 *secs*

9.1 자연어 처리란

- **자연어 처리를 위한 라이브러리**

  3단계. KoNLPy 설치

  - KoNLPy를 설치

```
> pip install konlpy
```

```
pip install konlpy

Looking in indexes: https://pypi.org/simple, https://u
Collecting konlpy
  Downloading konlpy-0.6.0-py2.py3-none-any.whl (19.4
     ---------------------------------------
Requirement already satisfied: lxml>=4.1.0 in /usr/loc
Requirement already satisfied: numpy>=1.6 in /usr/loca
Collecting JPype1>=0.7.0
  Downloading JPype1-1.4.1-cp310-cp310-manylinux_2_12_
     ---------------------------------------
Requirement already satisfied: packaging in /usr/local
Installing collected packages: JPype1, konlpy
Successfully installed JPype1-1.4.1 konlpy-0.6.0
```

9.1 자연어 처리란

- **자연어 처리를 위한 라이브러리**
  - 설치가 완료되었으니, 예제를 살펴보자

코드 9-6 라이브러리 호출 및 문장을 형태소로 변환[1]

```
from konlpy.tag import Komoran
komoran = Komoran()
print(komoran.morphs('딥러닝이 쉽나요? 어렵나요?')) ------ 텍스트를 형태소로 반환
```

```
['딥러닝이', '쉽', '나요', '?', '어렵', '나요', '?']
```

9.1 자연어 처리란

- **자연어 처리를 위한 라이브러리**
  - 이번에는 문장을 형태소로 변환한 후 품사를 태깅해 보자

코드 9-7 품사 태깅

```
print(komoran.pos('소파 위에 있는 것이 고양이인가요? 강아지인가요?'))
```

------ 텍스트에서 품사를 태깅하여 반환

  - 다음은 문장을 형태소로 분해하여 품사를 태깅한 출력 결과

```
[('소파', 'NNP'), ('위', 'NNG'), ('에', 'JKB'), ('있', 'VV'), ('는', 'ETM'), ('것',
'NNB'), ('이', 'JKS'), ('고양이', 'NNG'), ('이', 'VCP'), ('ㄴ가요', 'EF'), ('?',
'SF'), ('강아지', 'NNG'), ('이', 'VCP'), ('ㄴ가요', 'EF'), ('?', 'SF')]
```

  - 참고로 KoNLPy에서 제공하는 주요 기능은 다음과 같음

    - 형태소 분석
    - 품사 태깅

> KOMORAN

- https://docs.komoran.kr/firststep/postypes.html

대분류	소분류	세분류
체언	명사NN	일반명사NNG
		고유명사NNP
		의존명사NNB
	대명사NP	대명사NP
		수사NR
용언	동사VV	동사VV
	형용사VA	형용사VA
	보조용언VX	보조용언VX
	지정사VC	긍정지정사VCP
		부정지정사VCN
수식언	관형사MM	관형사MM
	부사MA	일반부사MAG
		접속부사MAJ

NNG (nouns used as generic or common nouns)
NNP (nouns used as proper nouns)

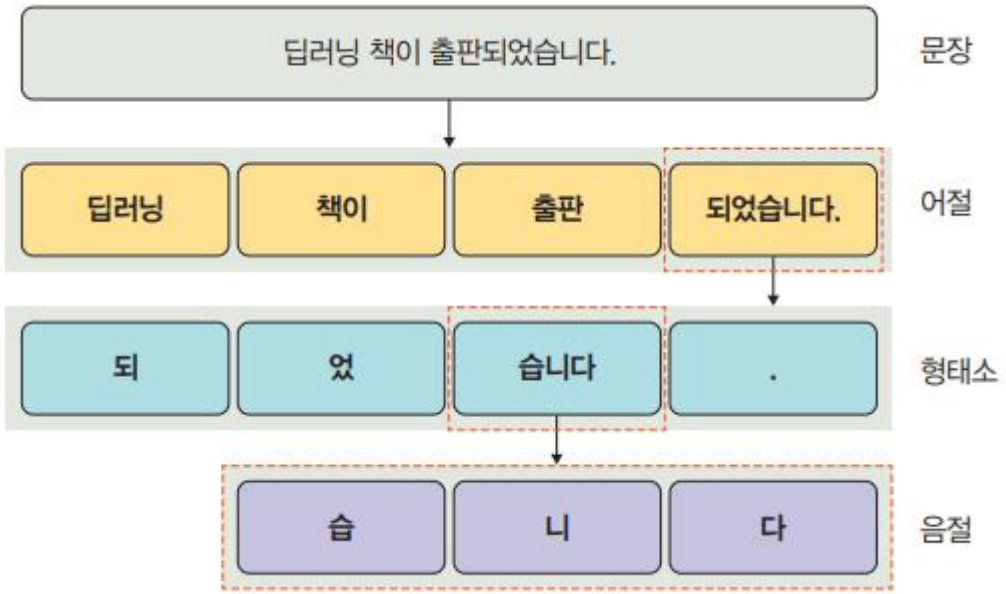
9.1 자연어 처리란

- **자연어 처리를 위한 라이브러리**

  형태소

  - 형태소는 언어를 쪼갤 때 의미를 가지는 최소 단위
  - 다음 그림은 형태소 분석을 위한 단계를 도식화한 것

▼ **그림 9-14 형태소**



9.1 자연어 처리란

- **자연어 처리를 위한 라이브러리**

  Gensim

  - Gensim은 파이썬에서 제공하는 워드투벡터(Word2Vec) 라이브러리
  - 딥러닝 라이브러리는 아니지만 효율적이고 확장 가능하기 때문에 폭넓게 사용하고 있음
  - 다음은 Gensim에서 제공하는 주요 기능
    - **임베딩**: 워드투벡터
    - 토픽 모델링
    - **LDA**(Latent Dirichlet Allocation) **잠재 디리클레 할당**

9.1 자연어 처리란

- **자연어 처리를 위한 라이브러리**
  - Gensim을 사용하려면 다음 명령으로 먼저 설치해야 함
  - 9.2절에서 사용하므로 여기에서 설치

```
> pip install -U gensim
```

  - -U 옵션은 이미 설치된 패키지를 업그레이드하는 것을 의미

9.1 자연어 처리란

- **자연어 처리를 위한 라이브러리**

  사이킷런

  - 사이킷런(scikit-learn)은 파이썬을 이용하여 문서를 전처리할 수 있는 라이브러리를 제공
  - 자연어 처리에서 특성 추출 용도로 많이 사용
  - 다음은 사이킷런에서 제공하는 주요 기능
    - **CountVectorizer**: 텍스트에서 단어의 등장 횟수를 기준으로 특성을 추출
    - **Tfidfvectorizer**: TF-IDF 값을 사용해서 텍스트에서 특성을 추출
    - **HashingVectorizer:** CountVectorizer와 방법이 동일하지만 텍스트를 처리할 때 해시 함수를 사용하기 때문에 실행 시간이 감소

참고: . "Term Frequency-Inverse Document Frequency Vectorizer"의 약자 .

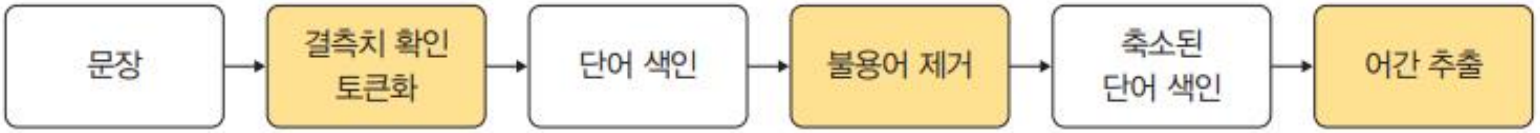
빈도(TF)는 용어가 문서에 나타나는 빈도를 측정하는 반면, 역 문서 빈도(IDF)는 말뭉치의 모든 문서에서 용어가 얼마나 중요한지를 측정합니다

9.2 전처리

9.2 전처리

- **전처리**
  - 머신 러닝이나 딥러닝에서 텍스트 자체를 특성으로 사용할 수는 없음
  - 텍스트 데이터에 대한 전처리 작업이 필요한데, 이때 전처리를 위해 토큰화, 불용어 제거 및 어간 추출 등 작업이 필요함

▼ **그림 9-15 전처리 과정**



9.2 전처리

- **결측치 확인**

  결측치 확인하기

  - 결측치를 확인하기 위해 내려받은 예제 파일의 data 폴더에 있는 class2.csv 파일을 사용

코드 9-8 결측치를 확인할 데이터 호출

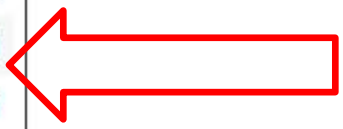
```
import pandas as pd
df = pd.read_csv('..\chap09\data\class2.csv')
df ------ 주어진 데이터를 확인
```

9.2 전처리

- **결측치 확인**
  - 다음과 같이 class2.csv 데이터셋을 확인할 수 있음

▼ **그림 9-16 class2.csv 데이터셋**

	Unnamed: 0	id	tissue	class	class2	x	y	r
0	0	mdb000	C	CIRC	N	535.0	475.0	192.0
1	1	mdb001	A	CIRA	N	433.0	268.0	58.0
2	2	mdb002	A	CIRA	I	NaN	NaN	NaN
3	3	mdb003	C	CIRC	B	NaN	NaN	NaN
4	4	mdb004	F	CIRF	I	488.0	145.0	29.0
5	5	mdb005	F	CIRF	B	544.0	178.0	26.0



  - 여기에서 주어진 데이터 중 NaN으로 표시된 부분들이 결측치
  - NaN(Not A Number)

9.2 전처리

- **결측치 확인**
  - isnull() 메서드를 사용하여 결측치 개수를 확인

코드 9-9 결측치 개수 확인

```
df.isnull().sum()  ------ isnull() 메서드를 사용하여 결측치가 있는지 확인한 후,
                          sum() 메서드를 사용하여 결측치가 몇 개인지 합산하여 보여 줍니다.
```

9.2 전처리

- **결측치 확인**
  - 전체 데이터 대비 결측치 비율을 확인해 보자

코드 9-10 결측치 비율

```
df.isnull().sum() / len(df)
```

9.2 전처리

- **결측치 확인**

  결측치 처리하기

  - 다음은 모든 행에 결측치가 존재한다면(모든 행이 NaN일 때) 해당 행을 삭제하는 처리 방법

코드 9-11 결측치 삭제 처리

```
df = df.dropna(how='all') ------ 모든 행이 NaN일 때만 삭제
print(df) ------ 데이터 확인(삭제 유무 확인)
```

코드 9-12 결측치 삭제 처리

```
df1 = df.dropna() ------ 데이터에 하나라도 NaN 값이 있으면 행을 삭제
print(df1)
```

9.2 전처리

- **결측치 확인**
  - 다음은 결측치를 삭제 처리하여 출력된 결과

	Unnamed: 0	id	tissue	class	class2	x	y	r
0	0	mdb000	C	CIRC	N	535.0	475.0	192.0
1	1	mdb001	A	CIRA	N	433.0	268.0	58.0
4	4	mdb004	F	CIRF	I	488.0	145.0	29.0
5	5	mdb005	F	CIRF	B	544.0	178.0	26.0

9.2 전처리

- **결측치 확인**
  - 다음은 결측치를 다른 값으로 채우는 방법
  - 결측치를 '0'으로 채워 보자

코드 9-13 결측치를 0으로 채우기

```
df2 = df.fillna(0)
print(df2)
```

9.2 전처리

- **결측치 확인**
  - 다음은 결측치를 0으로 채운 출력 결과
  - NaN이 0으로 채워진 것을 확인할 수 있음

	Unnamed: 0	id	tissue	class	class2	x	y	r
0	0	mdb000	C	CIRC	N	535.0	475.0	192.0
1	1	mdb001	A	CIRA	N	433.0	268.0	58.0
2	2	mdb002	A	CIRA	I	0.0	0.0	0.0
3	3	mdb003	C	CIRC	B	0.0	0.0	0.0
4	4	mdb004	F	CIRF	I	488.0	145.0	29.0
5	5	mdb005	F	CIRF	B	544.0	178.0	26.0

9.2 전처리

- **결측치 확인**
  - 다음으로 결측치를 해당 열의 평균값으로 채워 보자

코드 9-14 결측치를 평균으로 채우기

```
df['x'].fillna(df['x'].mean(), inplace=True)
print(df)
```

9.2 전처리

- **결측치 확인**
  - 다음은 결측치를 평균으로 채운 출력 결과
  - x열에 대해 평균값(500.0)으로 NaN 값이 채워져 있는 것을 확인할 수 있음

	Unnamed: 0	id	tissue	class	class2	x	y	r
0	0	mdb000	C	CIRC	N	535.0	475.0	192.0
1	1	mdb001	A	CIRA	N	433.0	268.0	58.0
2	2	mdb002	A	CIRA	I	500.0	NaN	NaN
3	3	mdb003	C	CIRC	B	500.0	NaN	NaN
4	4	mdb004	F	CIRF	I	488.0	145.0	29.0
5	5	mdb005	F	CIRF	B	544.0	178.0	26.0

9.2 전처리

- **결측치 확인**
  - 이외에도 다음 방법들로 결측치를 처리할 수 있음

    - 데이터에 하나라도 NaN 값이 있을 때 행 전체를 삭제
    - 데이터가 거의 없는 특성(열)은 특성(열) 자체를 삭제
    - 최빈값 혹은 평균값으로 NaN 값을 대체

9.2 전처리

- **토큰화**
  - 토큰화(tokenization)는 주어진 텍스트를 단어/문자 단위로 자르는 것을 의미
  - 토큰화는 문장 토큰화와 단어 토큰화로 구분
  - 예를 들어 'A cat is on the sofa'라는 문장이 있을 때 단어 토큰화를 진행하면 각각의 단어인 'A', 'cat', 'is', 'on', 'the', 'sofa'로 분리

9.2 전처리

- 토큰화

  문장 토큰화

  - 주어진 문장을 토큰화한다는 것은 마침표(.), 느낌표(!), 물음표(?) 등 문장의 마지막을 뜻하는 기호에 따라 분리하는 것

9.2 전처리

- 토큰화
  - NLTK를 이용하여 문장 토큰화를 구현해 보자

코드 9-15 문장 토큰화

```
from nltk import sent_tokenize
text_sample = 'Natural Language Processing, or NLP, is the process of extracting the
meaning, or intent, behind human language. In the field of Conversational artificial
intelligence (AI), NLP allows machines and applications to understand the intent of
human language inputs, and then generate appropriate responses, resulting in a natural
conversation flow.'
tokenized_sentences = sent_tokenize(text_sample)
print(tokenized_sentences)
```

```
['Natural Language Processing, or NLP, is the process of extracting the meaning,
or intent, behind human language.', 'In the field of Conversational artificial
intelligence (AI), NLP allows machines and applications to understand the intent of
human language inputs, and then generate appropriate responses, resulting in a natural
conversation flow.']
```

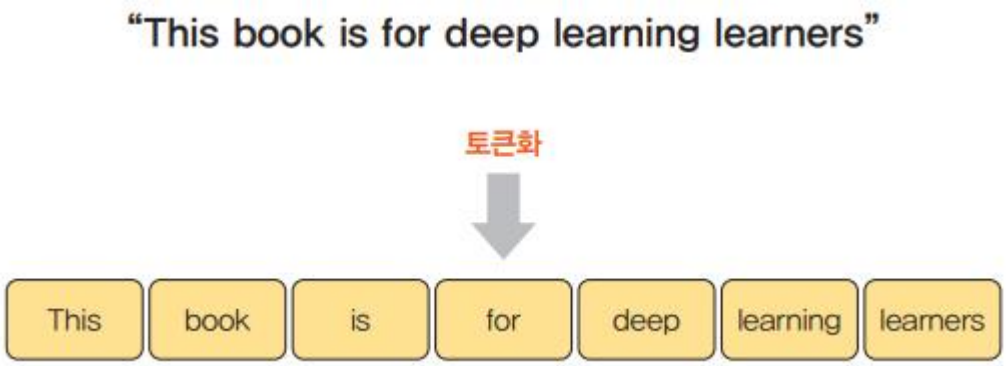
9.2 전처리

- 토큰화

단어 토큰화

- 단어 토큰화는 다음과 같이 띄어쓰기를 기준으로 문장을 구분

▼ **그림 9-17 단어 토큰화**



9.2 전처리

- 토큰화
  - 한국어는 띄어쓰기만으로 토큰을 구분하기 어려운 단점이 있음(한글 토큰화는 뒤에서 학습할 KoNLPy를 사용)
  - 역시 NLTK 라이브러리를 이용하여 주어진 문장을 단어 단위로 토큰화해 보자

코드 9-16 단어 토큰화

```
from nltk import word_tokenize
sentence = "This book is for deep learning learners"
words = word_tokenize(sentence)
print(words)
```

  - 다음은 단어 토큰화를 실행한 결과

```
['This', 'book', 'is', 'for', 'deep', 'learning', 'learners']
```

9.2 전처리

- 토큰화
  - 아포스트로피(')가 있는 문장은 어떻게 구분할까?
  - 아포스트로피에 대한 분류는 NLTK에서 제공하는 WordPunctTokenizer를 이용
  - 예를 들어 it's는 it, ', s로 구분했고, don't는 don, ', t로 구분
  - 다음 코드는 아포스트로피가 포함된 문장을 구분

코드 9-17 아포스트로피가 포함된 문장에서 단어 토큰화

```
from nltk.tokenize import WordPunctTokenizer
sentence = "it's nothing that you don't already know except most people aren't aware
of how their inner world works."
words = WordPunctTokenizer().tokenize(sentence)
print(words)
```

```
['it', ''', 's', 'nothing', 'that', 'you', 'don', ''', 't', 'already', 'know',
'except', 'most', 'people', 'aren', ''', 't', 'aware', 'of', 'how', 'their', 'inner',
'world', 'works', '.']
```

9.2 전처리

- 토큰화

한글 토큰화 예제

- 한국어 토큰화는 앞서 배운 KoNLPy 라이브러리를 사용
- 9장 예제 data 폴더의 ratings_train.txt 데이터 파일을 사용

코드 9-18 라이브러리 호출 및 데이터셋 준비

```
import csv
from konlpy.tag import Okt
from gensim.models import word2vec

f = open(r'..\data\ratings_train.txt', 'r', encoding='utf-8')
rdr = csv.reader(f, delimiter='\t')
rdw = list(rdr)
f.close()
```

9.2 전처리

- 토큰화
  - 한글 형태소 분석을 위해 오픈 소스 한글 형태소 분석기(Twitter(Okt))를 사용

코드 9-19 오픈 소스 한글 형태소 분석기 호출

```
twitter = Okt()

result = []

for line in rdw: ------ 텍스트를 한 줄씩 처리
    malist = twitter.pos(line[1], norm=True, stem=True) ------ 형태소 분석
    r = []
    for word in malist:
        if not word[1] in ["Josa","Eomi","Punctuation"]: ------ 조사, 어미, 문장 부호는 제외하고 처리
            r.append(word[0])
    rl = (" ".join(r)).strip() ------ 형태소 사이에 " "(공백)을 넣고, 양쪽 공백은 삭제
    result.append(rl)
    print(rl)
```

9.2 전처리

- 토큰화
  - 다음은 형태소 분석 결과

```
document
아 더빙 진짜 짜증나다 목소리
흠 포스터 보고 초딩 영화 줄 오버 연기 가볍다 않다
너 무재 밓었 다그 래서 보다 추천 다
교도소 이야기 구먼 솔직하다 재미 없다 평점 조정
... 중간 생략 ...
이 뭐 한국인 거들다 먹거리 필리핀 혼혈 착하다
청춘 영화 최고봉 방황 우울하다 날 들 자화상
한국 영화 최초 수간 하다 내용 담기다 영화
```

9.2 전처리

- 불용어 제거
  - 불용어(stop word)란 문장 내에서 빈번하게 발생하여 의미를 부여하기 어려운 단어들을 의미
  - 예를 들어 'a', 'the' 같은 단어들은 모든 구문(phrase)에 매우 많이 등장하기 때문에 아무런 의미가 없음
  - 특히 불용어는 자연어 처리에 있어 효율성을 감소시키고 처리 시간이 길어지는 단점이 있기 때문에 반드시 제거가 필요함

9.2 전처리

- 불용어 제거
  - 다음은 NLTK 라이브러리를 이용한 코드

```
import nltk
from nltk.corpus import stopwords
nltk.download('stopwords')
nltk.download('punkt')
from nltk.tokenize import word_tokenize

sample_text = "One of the first things that we ask ourselves is what are the pros and
cons of any task we perform."
text_tokens = word_tokenize(sample_text)

tokens_without_sw = [word for word in text_tokens if not word in stopwords.words(
                    'english')]
print("불용어 제거 미적용:", text_tokens, '\n')
print("불용어 제거 적용:", tokens_without_sw)
```

[40]
```
import nltk
from nltk.corpus import stopwords
nltk.download('stopwords')
nltk.download('punkt')
from nltk.tokenize import word_tokenize
```

```
[nltk_data] Downloading package stopwords to /root/nltk_data...
[nltk_data]   Package stopwords is already up-to-date!
[nltk_data] Downloading package punkt to /root/nltk_data...
[nltk_data]   Package punkt is already up-to-date!
```

```
sample_text='One of the first things that we ask ourselves is what are the pros and cons of any task we perform'
text_tokens=word_tokenize(sample_text)

token_without_sw=[word for word in text_tokens if not word in stopwords.words('english')]

print(text_tokens)
print(token_without_sw)
```

```
['One', 'of', 'the', 'first', 'things', 'that', 'we', 'ask', 'ourselves', 'is', 'what', 'are', 'the', 'pros', 'and', 'cons', 'of', 'any',
['One', 'first', 'things', 'ask', 'pros', 'cons', 'task', 'perform']
```

9.2 전처리

- 불용어 제거
  - 다음은 불용어 제거와 미제거에 대한 실행 결과

```
불용어 제거 미적용: ['One', 'of', 'the', 'first', 'things', 'that', 'we', 'ask', 'ourselves',
'is', 'what', 'are', 'the', 'pros', 'and', 'cons', 'of', 'any', 'task', 'we', 'perform',
'.']

불용어 제거 적용: ['One', 'first', 'things', 'ask', 'pros', 'cons', 'task', 'perform', '.']
```

  - 불용어 제거를 적용한 결과는 'of', 'the' 같은 단어가 삭제된 것을 확인할 수 있음

  - if not word in stopwords.words('english'): 불러온 단어가 영어 불용어 사전에 포함되어 있지 않으면, 즉 불용어가 아니면 token_without_sw 리스트에 추가.

9.2 전처리

- 어간 추출
  - 어간 추출(stemming)과 표제어 추출(lemmatization)은 단어 원형을 찾아 주는 것
  - 예를 들어 '쓰다'의 다양한 형태인 writing, writes, wrote에서 write를 찾는 것
  - 어간 추출은 단어 그 자체만 고려하기 때문에 품사가 달라도 사용 가능
  - 예를 들어 어간 추출은 다음과 같이 사용

    - Automates, automatic, automation → automat

9.2 전처리

- 어간 추출
  - 반면 표제어 추출은 단어가 문장 속에서 어떤 품사로 쓰였는지 고려하기 때문에 품사가 같아야 사용 가능
  - 예를 들어 다음 표제어 추출이 가능
    - am, are, is → be
    - car, cars, car's, cars' → car
  - 즉, 어간 추출과 표제어 추출은 둘 다 어근 추출이 목적
  - 어간 추출은 사전에 없는 단어도 추출할 수 있고 표제어 추출은 사전에 있는 단어만 추출할 수 있다는 점에서 차이가 있음

9.2 전처리

- 어간 추출
  - NLTK의 어간 추출로는 대표적으로 포터(porter)와 랭커스터(lancaster) 알고리즘이 있음
  - 이 둘에 대한 차이를 코드로 확인해 보자
  - 먼저 포터 알고리즘을 적용해 보자

코드 9-23 포터 알고리즘

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

print(stemmer.stem('obesses'), stemmer.stem('obssesed'))
print(stemmer.stem('standardizes'), stemmer.stem('standardization'))
print(stemmer.stem('national'), stemmer.stem('nation'))
print(stemmer.stem('absentness'), stemmer.stem('absently'))
print(stemmer.stem('tribalical'), stemmer.stem('tribalicalized')) ------ 사전에 없는 단어
```

9.2 전처리

- 어간 추출
  - 다음은 포터 알고리즘을 실행한 결과

```
obsess obsess
standard standard
nation nation
absent absent
tribal tribalic
```

  - 포터 알고리즘 수행 결과 단어 원형이 비교적 잘 보존되어 있는 것을 확인할 수 있음

  - tribal 종족의 / tribalic 부족의

9.2 전처리

- 어간 추출
  - 이번에는 랭커스터 알고리즘을 적용해 보자

코드 9-24 랭커스터 알고리즘

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

print(stemmer.stem('obsesses'), stemmer.stem('obsessed'))
print(stemmer.stem('standardizes'), stemmer.stem('standardization'))
print(stemmer.stem('national'), stemmer.stem('nation'))
print(stemmer.stem('absentness'), stemmer.stem('absently'))
print(stemmer.stem('tribalical'), stemmer.stem('tribalicalized')) ------ 사전에 없는 단어
```

9.2 전처리

- 어간 추출
  - 다음은 랭커스터 알고리즘을 실행한 결과

```
obsess obsess
standard standard
nat nat
abs abs
trib trib
```

  - 포터 알고리즘과 다르게 랭커스터 알고리즘은 단어 원형을 알아볼 수 없을 정도로 축소시키기 때문에 정확도가 낮음
  - 일반적인 상황보다는 데이터셋을 축소시켜야 하는 특정 상황에서나 유용함

9.2 전처리

- 어간 추출

  표제어 추출

  - 일반적으로 어간 추출보다 표제어 추출의 성능이 더 좋음
  - 품사와 같은 문법뿐만 아니라 문장 내에서 단어 의미도 고려하기 때문에 성능이 좋음
  - 어간 추출보다 시간이 더 오래 걸리는 단점이 있음

9.2 전처리

- 어간 추출 - 원형(lemmatization)을 추출
  - 표제어 추출은 WordNetLemmatizer를 주로 사용

코드 9-25 표제어 추출

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

from nltk.stem import WordNetLemmatizer ------ 표제어 추출 라이브러리
lemma = WordNetLemmatizer()

print(stemmer.stem('obsesses'), stemmer.stem('obsessed'))
print(lemma.lemmatize('standardizes'), lemma.lemmatize('standardization'))
print(lemma.lemmatize('national'), lemma.lemmatize('nation'))
print(lemma.lemmatize('absentness'), lemma.lemmatize('absently'))
print(lemma.lemmatize('tribalical'), lemma.lemmatize('tribalicalized'))
```

9.2 전처리

- 어간 추출
  - 일반적으로 표제어 추출의 성능을 높이고자 단어에 대한 품사 정보를 추가하곤 함
  - 다음 코드와 같이 두 번째 파라미터에 품사 정보를 넣어 주면 정확하게 어근 단어를 추출할 수 있음

코드 9-26 품사 정보가 추가된 표제어 추출

```
print(lemma.lemmatize('obsesses','v'), lemma.lemmatize('obsessed','a'))
print(lemma.lemmatize('standardizes','v'), lemma.lemmatize('standardization','n'))
print(lemma.lemmatize('national','a'), lemma.lemmatize('nation','n'))
print(lemma.lemmatize('absentness','n'), lemma.lemmatize('absently','r'))
print(lemma.lemmatize('tribalical','a'), lemma.lemmatize('tribalicalized','v'))
```

WordNetLemmatizer 객체인 lemma (표제어)를 사용하여 동사(verb)인 "obeses"와 형용사(adjective)인 "obsessed"를 각각 원형 복원(lemmatization)하는 코드입니다

adverb(부사)로 간주하여 원형을 복원

9.2 전처리

- 어간 추출
  - 다음은 품사 정보가 추가된 표제어 추출을 실행한 결과
  - 몇 개의 단어만 예시로 진행했기 때문에 앞에서 진행했던 결과와 동일하게 나타나지만 수백~수천 단어를 진행할 때는 차이가 크게 나타남

```
obsess obsessed
standardize standardization
national nation
absentness absently
tribalical tribalicalized
```

9.2 전처리

- 정규화
  - 파이토치의 데이터셋과 데이터로더를 이용하면 방대한 양의 데이터를 배치 단위로 쪼개서 처리할 수 있고, 데이터를 무작위로 섞을 수 있기 때문에 효율적으로 데이터를 처리할 수 있음
  - 또한, 여러 개의 GPU를 사용하여 데이터를 병렬로 학습시킬 수도 있음
  - 즉, 데이터양이 많을 때 주로 사용하며 데이터양이 많지 않다면 꼭 사용할 필요는 없음

감사합니다.