

자연어 전처리

1 자연어 처리란

2 전처리

자연어 처리란

9.1 자연어 처리란

● 자연어 처리란

- ❖ 자연어 처리란 우리가 일상생활에서 사용하는 언어의 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
- ❖ 또한, 자연어 처리를 위해 사용되는 용어들도 낯섬

9.1 자연어 처리란

- 자연어 처리란

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

9.1 자연어 처리란

▼ 그림 9-1 자연어 처리 완성도

완성도 높은 자연어 처리

스팸 처리
(spam detection)

맞춤법 검사
(spell checking)

단어 검색
(keyword search)

객체 인식
(named entity recognition)

완성도 낮은 자연어 처리

질의응답
(question & answering)

요약
(summarization)

유사 단어 바꾸어 쓰기
(paraphrase)

대화
(dialog)

9.1 자연어 처리란

- 자연어 처리 용어 및 과정

- 자연어 처리 관련 용어

- **말뭉치(corpus(코퍼스))**: 자연어 처리에서 모델을 학습시키기 위한 데이터이며, 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

9.1 자연어 처리란

- 자연어 처리 용어 및 과정

- ❖ 불용어(stop words): 문장 내에서 많이 등장하는 단어

분석과 관계없으며, 자주 등장하는 빈도 때문에 성능에 영향을 미치므로 사전에 제거해 주어야 함

불용어 : "a", "the", "she", "he" 등이 있음

- ❖ 어간 추출(stemming): 단어를 기본 형태로 만드는 작업

예를 들어 'consign', 'consigned', 'consigning', 'consignment'가 있을 때 기본 어인 'consign'으로 통일하는 것이 어간 추출

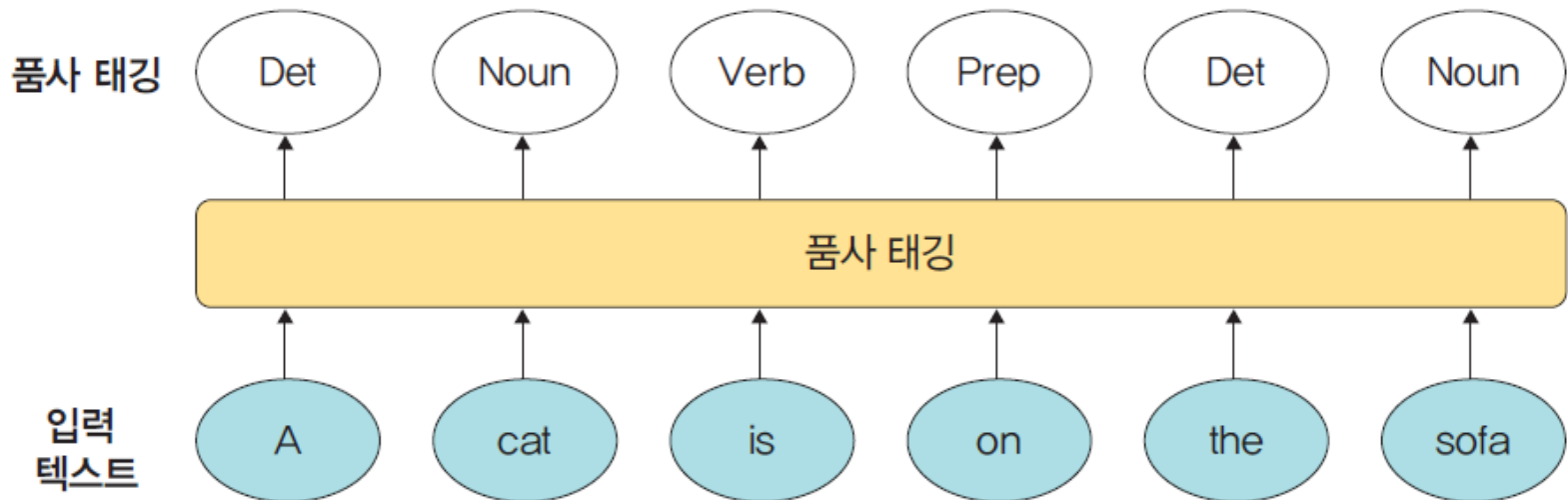
consign	}	consign
consigned		
consigning		
consignment		

9.1 자연어 처리란

● 자연어 처리 용어 및 과정

- ❖ 품사 태깅(part-of-speech tagging): 주어진 문장에서 품사를 식별하기 위해 붙여 주는 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
 - **Noun:** 명사
 - **Verb:** 동사
 - **Prep:** 전치사

- ❖ 품사 태깅은 NLTK를 이용할 수 있음

9.1 자연어 처리란

● 자연어 처리 용어 및 과정

- ❖ NLTK는 아나콘다가 설치되어 있다면 추가적으로 설치할 필요가 없지만, 책에서는 가상 환경에서 실습하므로 다음 명령으로 설치

```
> pip install nltk
```

- ❖ 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

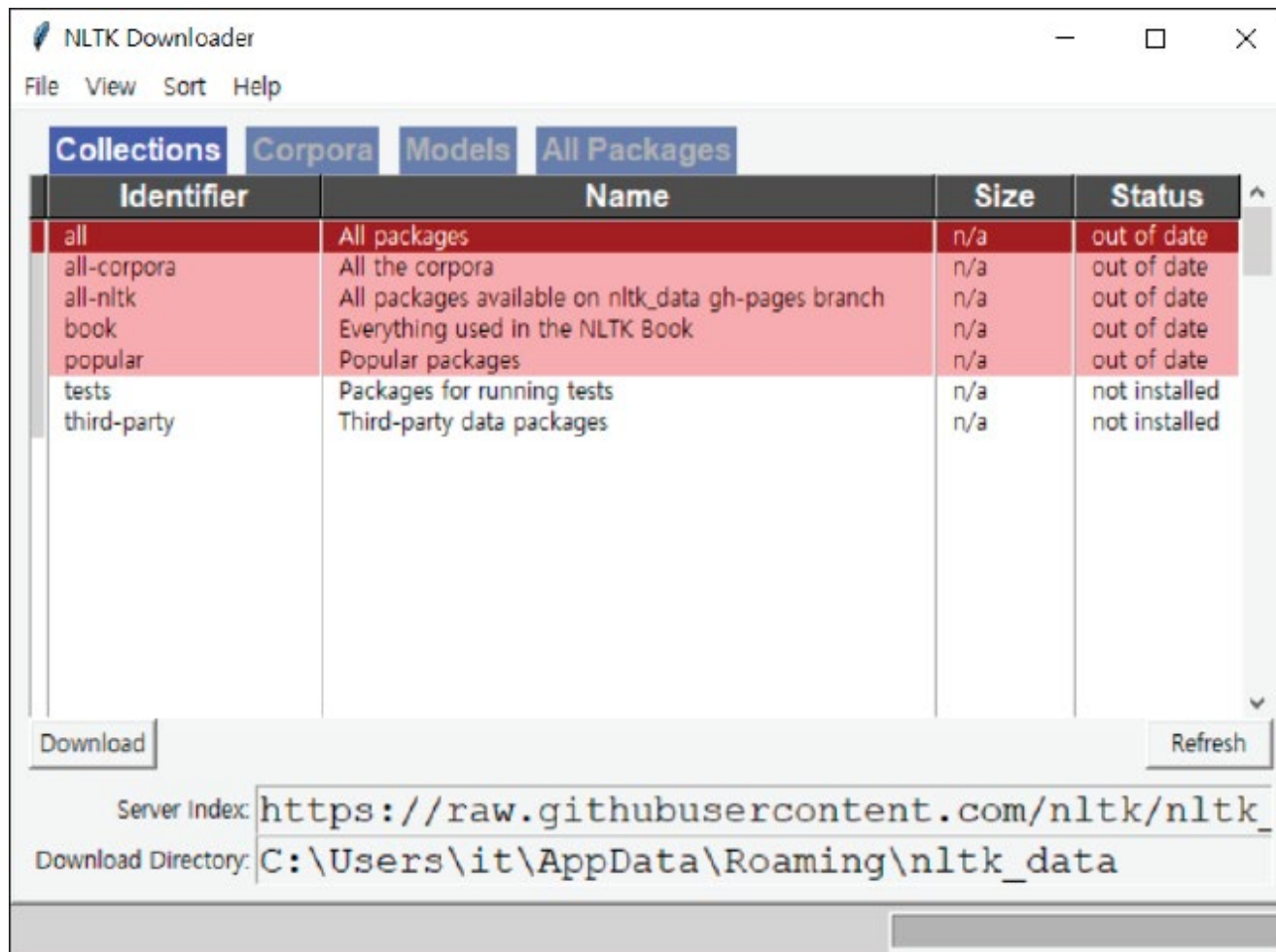
- 자연어 처리 용어 및 과정

NLTK Downloader

- ❖ 주피터 노트북에서 `nltk.download()` 코드를 실행하면 다음과 같이 NLTK Downloader 창이 뜬(윈도에서는 작업 표시줄에  표시로 나타남)
- ❖ 왼쪽 하단의 Download를 눌러야 관련 패키지 등을 내려받을 수 있음
- ❖ 내려받기가 완료된 후에는 File > Exit를 선택해야 다음 단계를 진행할 수 있음

9.1 자연어 처리란

▼ 그림 9-5 NLTK 다운로드



9.1 자연어 처리란

- 자연어 처리 용어 및 과정

- ❖ 다음은 문장 토큰화를 진행한 결과

- ```
['Is', 'it', 'possible', 'distinguishing', 'cats', 'and', 'dogs']
```

- ❖ 태깅에 필요한 자원을 내려받음

코드 9-2 태깅에 필요한 자원 내려받기

```
nltk.download('averaged_perceptron_tagger') ----- 태깅에 필요한 자원 내려받기
```

- ❖ 모두 내려받으면 다음과 같이 출력

True

## 9.1 자연어 처리란

- 자연어 처리 용어 및 과정

- ❖ 내려받은 자료를 이용하여 품사를 태깅

코드 9-3 품사 태깅

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

- ❖ 다음은 품사 태깅에 대한 출력 결과

```
[('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

### ● 자연어 처리 용어 및 과정

#### 자연어 처리 과정

- ❖ 자연어는 인간 언어
- ❖ 인간 언어는 컴퓨터가 이해할 수 없기 때문에 컴퓨터가 이해할 수 있는 언어로 바꾸고 원하는 결과를 얻기까지 크게 네 단계를 거침
- ❖ 첫 번째로 인간 언어인 자연어가 입력 텍스트로 들어오게 됨
- ❖ 이때 인간 언어가 다양하듯 처리 방식이 조금씩 다르며, 현재는 영어에 대한 처리 방법들이 잘 알려져 있음
- ❖ 두 번째로는 입력된 텍스트에 대한 전처리 과정이 필요함
- ❖ 세 번째로 전처리가 끝난 단어들을 임베딩

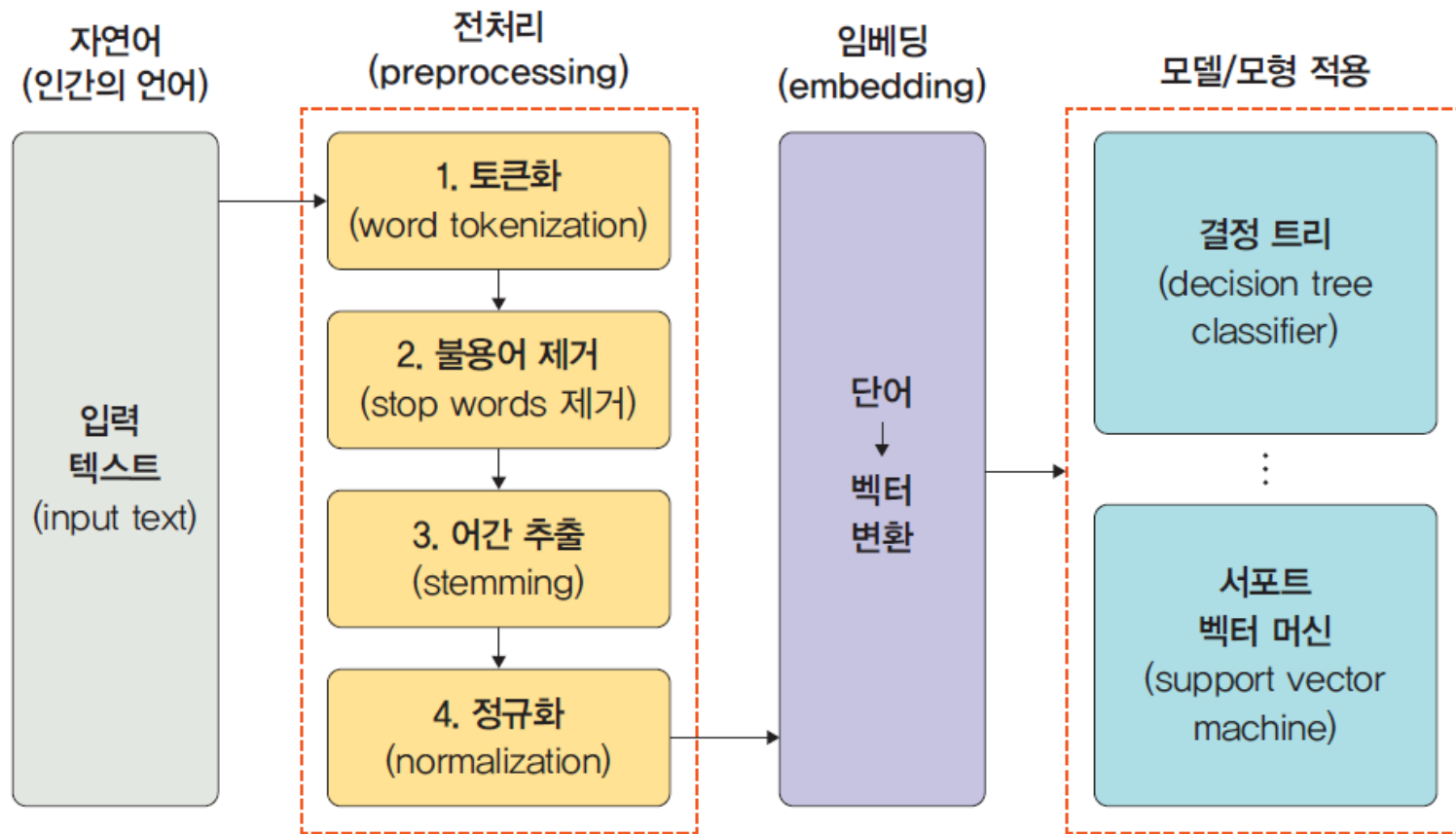
## 9.1 자연어 처리란

- 자연어 처리 용어 및 과정

- ❖ 마지막으로 컴퓨터가 이해할 수 있는 데이터가 완성되었기 때문에 모델/모형(예 결정 트리)을 이용하여 데이터에 대한 분류 및 예측을 수행
- ❖ 이때 데이터 유형에 따라 분류와 예측에 대한 결과가 달라짐

## 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 살펴보겠음

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)
```

- ❖ 다음은 string1에 대해 문장을 단어로 쪼갠 결과

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

## 9.1 자연어 처리란

- 자연어 처리를 위한 라이브러리

- ❖ 이번에는 string2를 nltk를 이용해서 단어 단위로 분리해 보겠음

코드 9-5 단어 단위로 분리

```
nltk.word_tokenize(string2)
```

## 9.1 자연어 처리란

- 자연어 처리를 위한 라이브러리

- ❖ 다음은 string2에 대해 문장을 단어로 분리시킨 결과

```
['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 분석기
- ❖ 기존에 공개된 꼬꼬마(Kkma), 코모란(Komoran), 한나눔(Hannanum), 트위터(Twitter), 메카브(Mecab) 분석기를 한 번에 설치하고 동일한 방법으로 사용할 수 있도록 해 줌



## 9.1 자연어 처리란

- 자연어 처리를 위한 라이브러리

- 원도 환경에서 KoNLPy 설치 방법

- 1단계. Oracle JDK 설치

- 1. KoNLPy를 설치하기 전에 Oracle JDK를 설치해야 함(KoNLPy 공식 사이트에서는 Oracle JDK를 설치하는 것을 권고하고 있으며, 해당 파일을 내려받을 수 있는 URL을 제시)

- Oracle JDK는 오라클 웹 사이트에 가입해야 내려받을 수 있음

- 오라클 웹 사이트에 접속

- 이미 계정이 있다면 **3**으로 이동

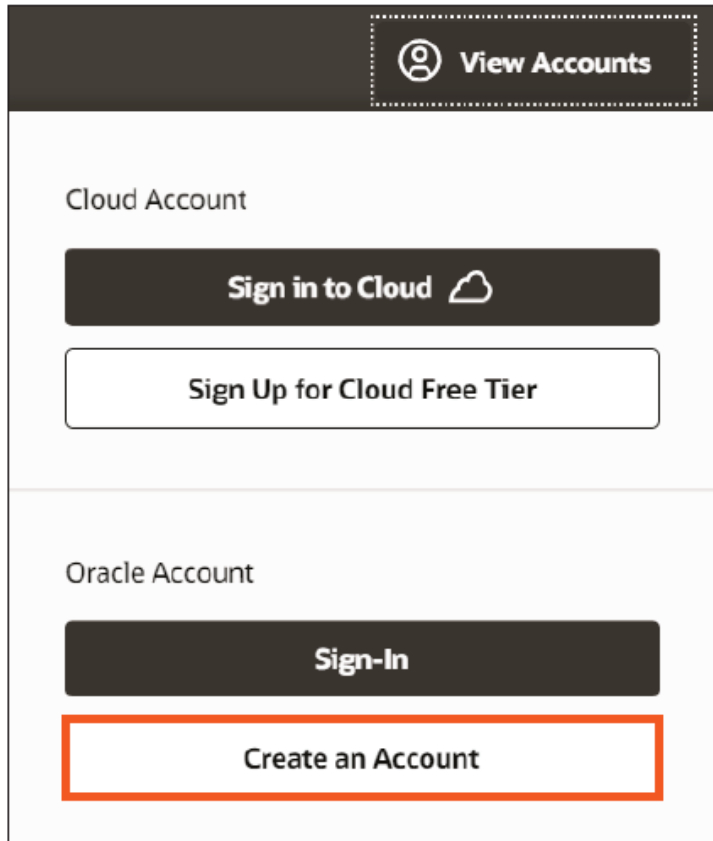
- <https://www.oracle.com/java/technologies/javase-downloads.html>

## 9.1 자연어 처리란

- 자연어 처리를 위한 라이브러리

2. 이메일 화면에서 **View Accounts > Create an Account**를 클릭하여 계정을 생성

▼ 그림 9-7 오라클 웹 사이트에서 계정 생성



## 9.1 자연어 처리란







### ● 자연어 처리를 위한 라이브러리

3. 회원 가입이 완료되었다면 다음 URL에 접속하여 자신의 운영 체제에 맞는 Oracle JDK를 선택

필자는 jdk-8u271-windows-x64.exe를 선택

<https://www.oracle.com/java/technologies/javase/javase-jdk8-downloads.html>

### ▼ 그림 9-8 Oracle JDK 버전 선택

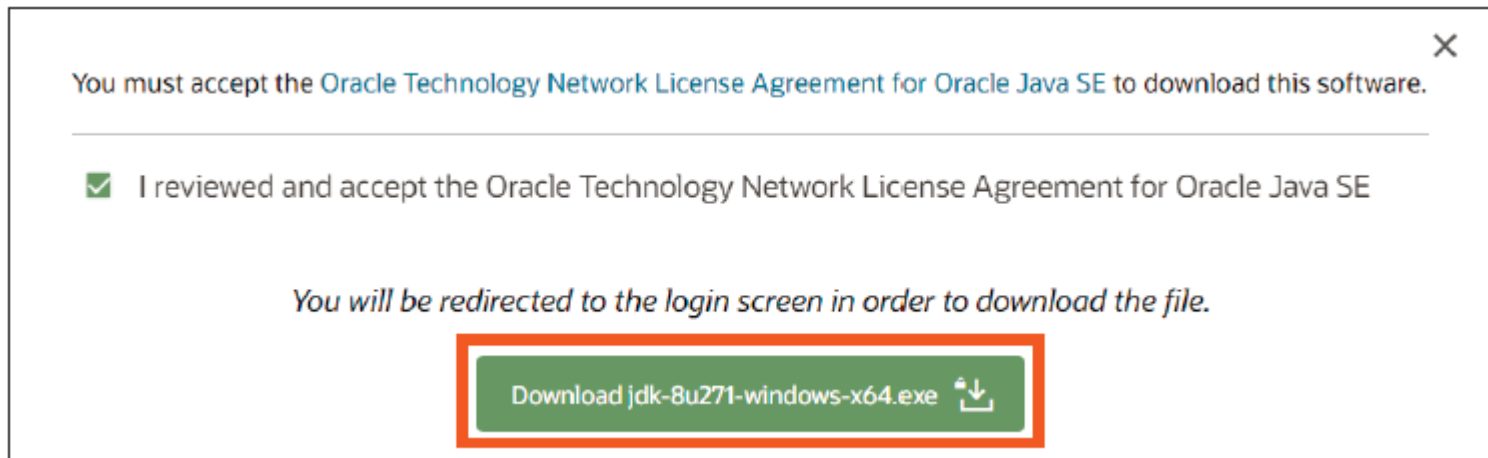
|                                     |           |                                                                                                                                      |
|-------------------------------------|-----------|--------------------------------------------------------------------------------------------------------------------------------------|
| Solaris SPARC 64-bit (SVR4 package) | 125.94 MB |  <a href="#">jdk-8u271-solaris-sparcv9.tar.Z</a>  |
| Solaris SPARC 64-bit                | 88.75 MB  |  <a href="#">jdk-8u271-solaris-sparcv9.tar.gz</a> |
| Solaris x64 (SVR4 package)          | 134.42 MB |  <a href="#">jdk-8u271-solaris-x64.tar.Z</a>     |
| Solaris x64                         | 92.52 MB  |  <a href="#">jdk-8u271-solaris-x64.tar.gz</a>   |
| Windows x86                         | 154.48 MB |  <a href="#">jdk-8u271-windows-i586.exe</a>     |
| Windows x64                         | 166.79 MB |  <a href="#">jdk-8u271-windows-x64.exe</a>      |

## 9.1 자연어 처리란

- 자연어 처리를 위한 라이브러리

4. 다음과 같이 알림 창이 뜨면 체크박스를 선택한 후 **Download**를 눌러 선택한 파일을 내려받음

▼ 그림 9-9 Oracle JDK 내려받기



## 9.1 자연어 처리란

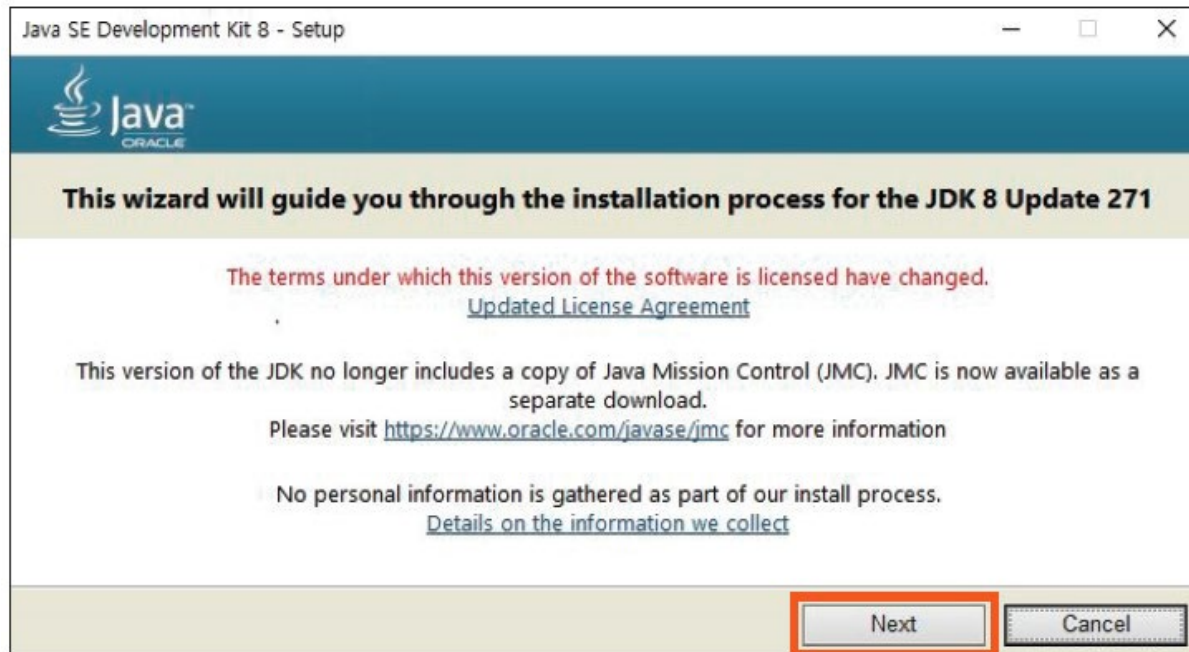
- 자연어 처리를 위한 라이브러리

- 5. 내려받은 파일을 더블클릭하여 설치

JDK 파일을 설치하는 단계

**Next**를 누름

- ▼ 그림 9-10 JDK 설치 시작

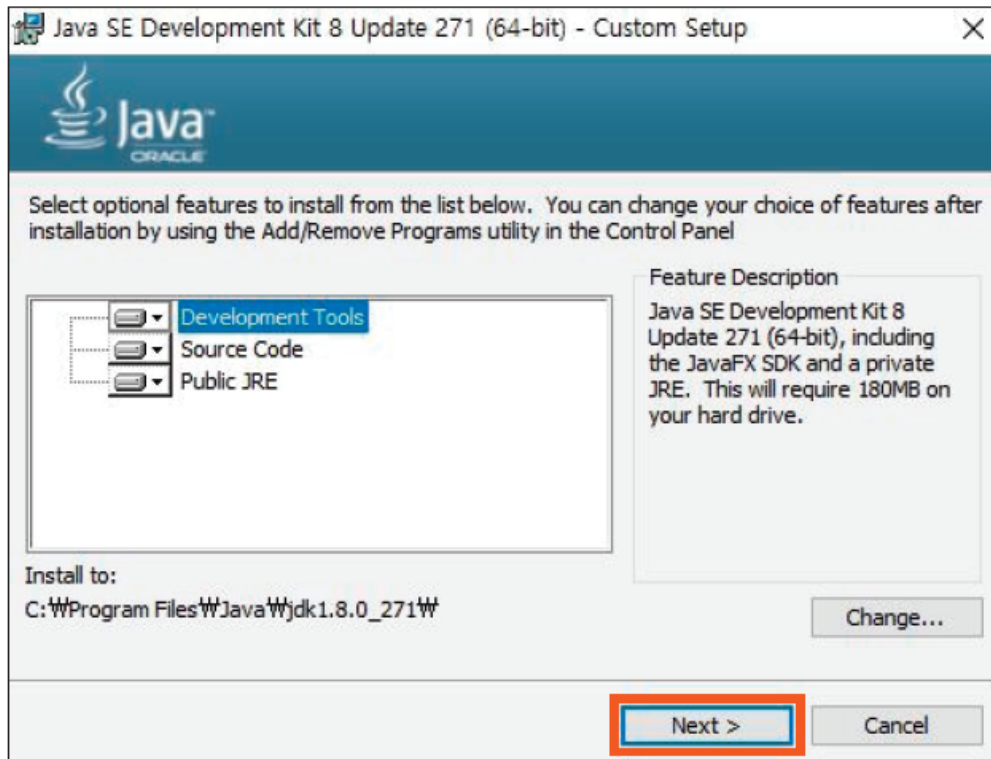


## 9.1 자연어 처리란

- 자연어 처리를 위한 라이브러리

6. 추가적인 기능 설치와 설치 경로를 지정하는 단계  
기본값을 그대로 두고 **Next**를 누름

▼ 그림 9-11 기능과 설치 경로 지정



## 9.1 자연어 처리란

- 자연어 처리를 위한 라이브러리

7. 자바를 설치할 폴더를 지정하는 단계로, 기본값을 그대로 두고 **다음**을 누름

▼ 그림 9-12 자바를 설치할 폴더 지정

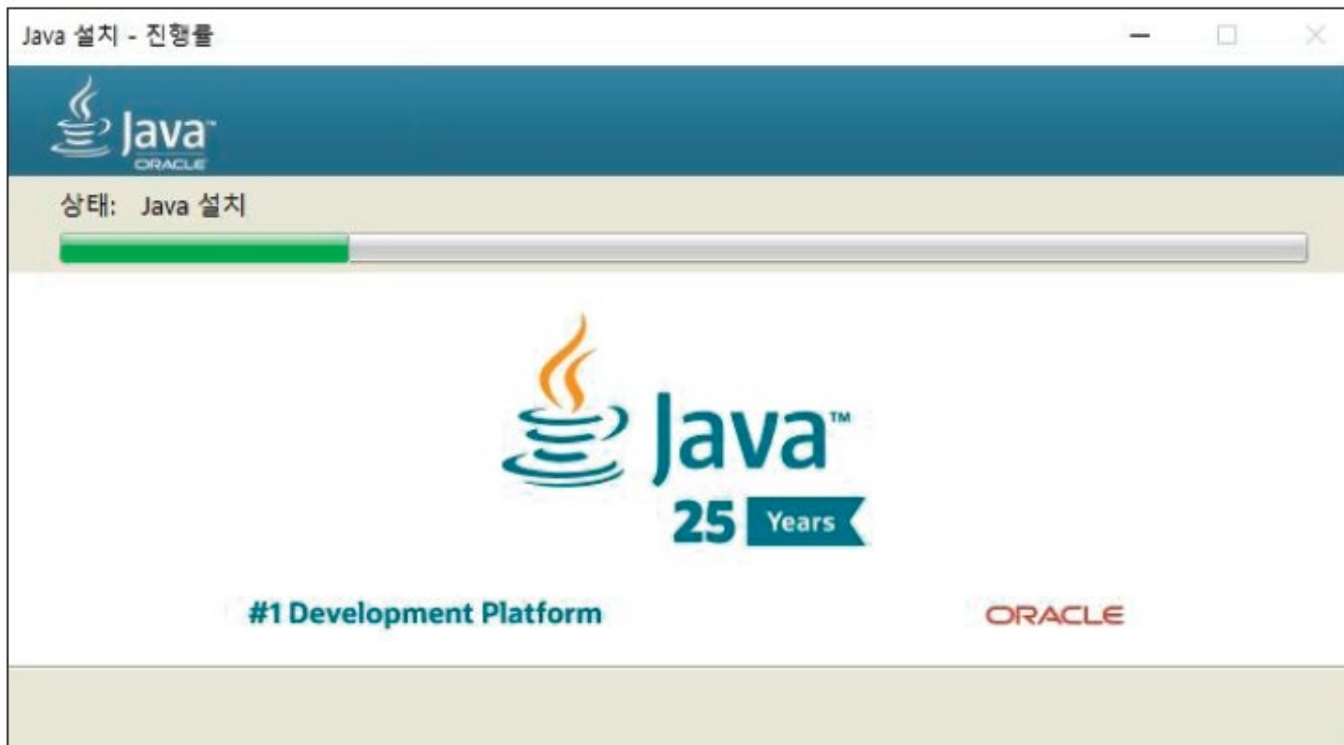


## 9.1 자연어 처리란

- 자연어 처리를 위한 라이브러리

8. 설치가 진행되고 있는 화면

▼ 그림 9-13 설치 중





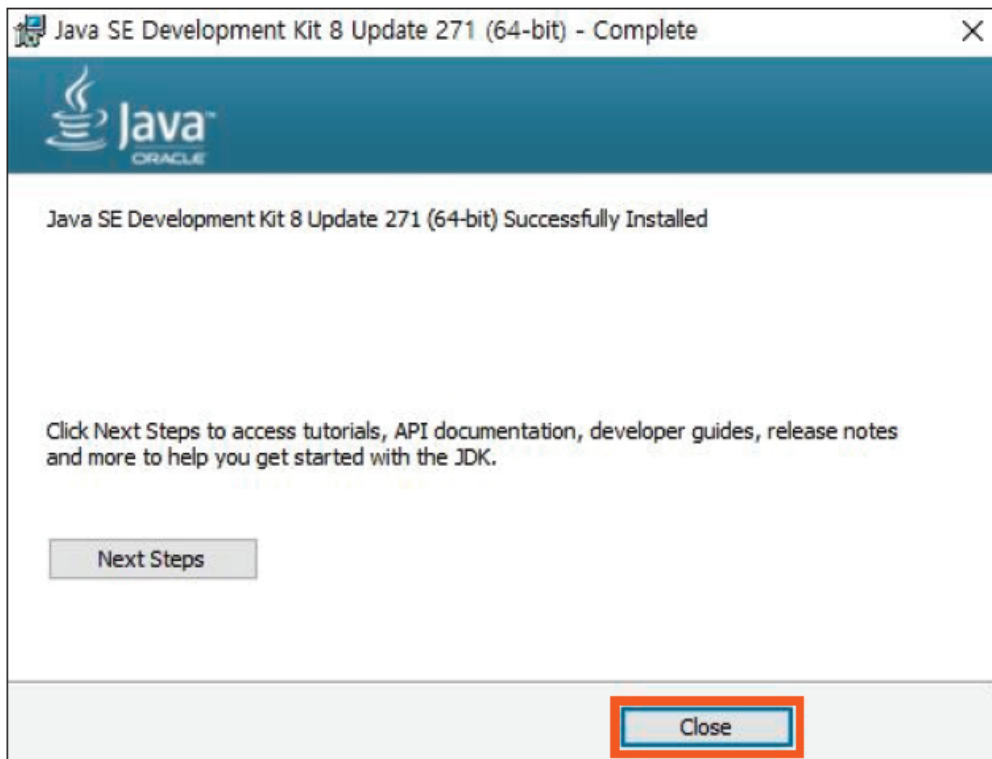
## 9.1 자연어 처리란

- 자연어 처리를 위한 라이브러리

- 9. 설치가 완료

- Close**를 누름

- ▼ 그림 9-14 설치 완료



## 9.1 자연어 처리란

- 자연어 처리를 위한 라이브러리

- 2단계. JType1 설치

- 1. 다음 URL에서 JType1을 내려받아 설치

- 이때 64비트 윈도는 win-amd64, 32비트 윈도는 win32라고 표시된 파일을 내려받아야 함

- 필자는 JType1-1.1.2-cp38-cp38-win\_amd64.whl 버전을 내려받았음

- <https://www.lfd.uci.edu/~gohlke/pythonlibs/#jtype>

## 9.1 자연어 처리란

### ▼ 그림 9-15 JPyPe1 버전 선택

**JPyPe**: allows full access to Java class libraries.

[JPyPe1-1.1.2-cp39-cp39-win\\_amd64.whl](#)

[JPyPe1-1.1.2-cp39-cp39-win32.whl](#)

[JPyPe1-1.1.2-cp38-cp38-win\\_amd64.whl](#)

[JPyPe1-1.1.2-cp38-cp38-win32.whl](#)

[JPyPe1-1.1.2-cp37-cp37m-win\\_amd64.whl](#)

[JPyPe1-1.1.2-cp37-cp37m-win32.whl](#)

[JPyPe1-1.1.2-cp36-cp36m-win\\_amd64.whl](#)

[JPyPe1-1.1.2-cp36-cp36m-win32.whl](#)

[JPyPe1-0.7.1-cp35-cp35m-win\\_amd64.whl](#)

[JPyPe1-0.7.1-cp35-cp35m-win32.whl](#)

[JPyPe1-0.7.1-cp27-cp27m-win\\_amd64.whl](#)

[JPyPe1-0.7.1-cp27-cp27m-win32.whl](#)

[JPyPe1-0.6.3-cp34-cp34m-win\\_amd64.whl](#)

[JPyPe1-0.6.3-cp34-cp34m-win32.whl](#)

## 9.1 자연어 처리란

- 자연어 처리를 위한 라이브러리

2. 아나콘다 프롬프트에서 tf2\_book 가상 환경으로 접속한 후 내려받은 JPytype1 파일을 설치

이때 내려받은 파일의 경로까지 모두 적어야 함

```
> pip install JPytype1-0.5.7-cp27-none-win_amd64.whl
```

## 9.1 자연어 처리란

- 자연어 처리를 위한 라이브러리

JType1을 설치할 때 오류가 발생한다면

- ❖ 먼저 다음 명령으로 pip를 업그레이드
- ❖ 그리고 나서 다시 JType1을 설치해 보자

```
> pip install --upgrade pip
```

- ❖ pip를 업그레이드하는 데 다음과 같은 오류 메시지가 표시되었다면 권한이 부족하다는 이야기이므로 아나콘다 프롬프트를 관리자 권한(아나콘다 프롬프트 메뉴에서 마우스 오른쪽 버튼을 눌러 관리자 권한 선택)으로 실행한 후 다시 시도해 보자

```
ERROR: Could not install packages due to an EnvironmentError: [WinError 5] 액세스가 거부되었습니다: 'C:\\Users\\it\\AppData\\Local\\Temp\\pip-uninstall-xb8elb6e\\pip.exe'
Consider using the `--user` option or check the permissions.
```

## 9.1 자연어 처리란

- 자연어 처리를 위한 라이브러리

3단계. KoNLPy 설치

❖ KoNLPy를 설치

```
> pip install konlpy
```

## 9.1 자연어 처리란

- 자연어 처리를 위한 라이브러리

- 우분투에서 KoNLPy 설치 방법

- ❖ 우분투에서는 다음 명령으로 Oracle JDK와 JType1을 바로 설치할 수 있음

```
$ sudo apt-get install g++ openjdk-8-jdk
```

```
$ sudo apt-get install python3-dev; pip3 install konlpy
```

- ❖ 설치가 완료되었으니, 예제를 살펴보자

코드 9-6 라이브러리 호출 및 문장을 형태로 변환

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

- ❖ 다음은 문장을 형태로 변환한 출력 결과

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

## 9.1 자연어 처리란

### ● 자연어 처리를 위한 라이브러리

- ❖ 이번에는 문장을 형태소로 변환한 후 품사를 태깅해 보겠음

코드 9-7 품사 태깅

```
print(komoran.pos('소파 위에 있는 것이 고양이인가요? 강아지인가요?')) ----- 텍스트에서 품사를
태깅하여 반환
```

- ❖ 다음은 문장을 형태로 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



## 9.1 자연어 처리란

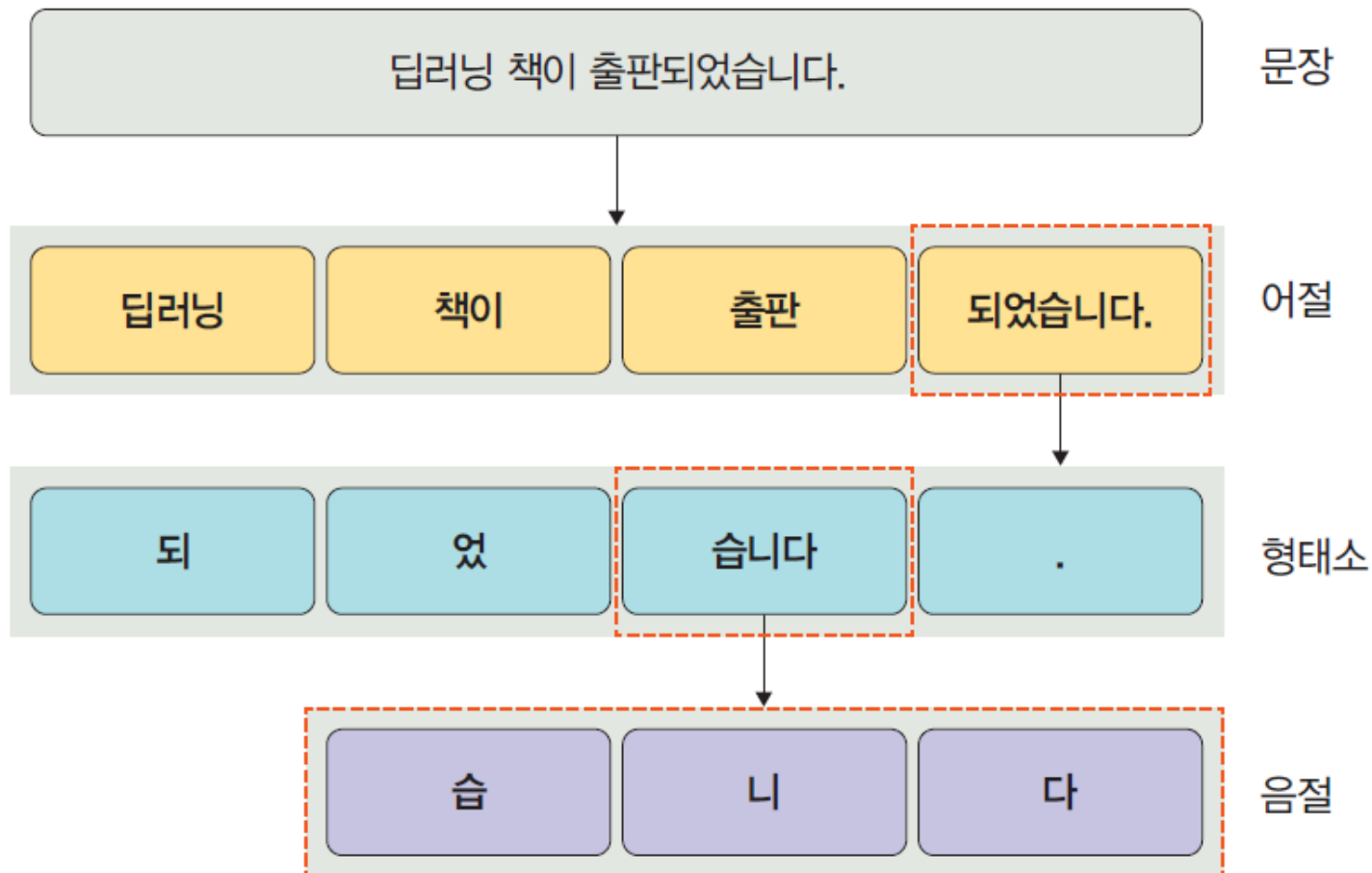
- 자연어 처리를 위한 라이브러리

### 형태소

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

## 9.1 자연어 처리란

### ▼ 그림 9-16 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)

## 9.1 자연어 처리란

- 자연어 처리를 위한 라이브러리

- ❖ Gensim을 사용하려면 다음 명령으로 먼저 설치해야 함
- ❖ 9.2절에서 사용하므로 여기에서 설치

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

## 9.1 자연어 처리란

### ● 자연어 처리를 위한 라이브러리

#### 사이킷런

- ❖ 사이킷런(scikit-learn)은 파이썬을 이용하여 문서를 전처리할 수 있는 라이브러리를 제공
- ❖ 특히 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

# 전처리

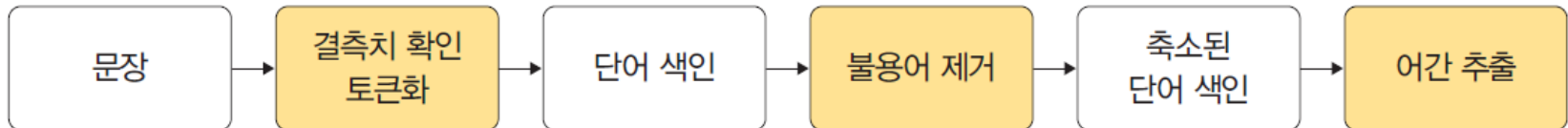
---

## 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
- ❖ 앞서도 살펴보았지만, 전처리 과정은 다음 그림과 같음

#### ▼ 그림 9-17 전처리 과정



## 9.2 전처리

### ● 결측치 확인

- ❖ 결측치는 다음 표의 성춘향에 대한 '몸무게'처럼 주어진 데이터셋에서 데이터가 없는(NaN) 것
- ❖ 결측치 확인 및 처리는 다음 방법을 이용

▼ 표 9-1 결측치

| ID | 이름  | 몸무게 | 키   |
|----|-----|-----|-----|
| 1  | 홍길동 | 76  | 177 |
| 2  | 성춘향 | NaN | 155 |
| 3  | 이도령 | 65  | 170 |



## 9.2 전처리

- 결측치 확인

### 결측치 확인하기

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

코드 9-8 결측치를 확인할 데이터 호출

```
import pandas as pd
df = pd.read_csv('../chap9\data\class2.csv')
df ----- 주어진 데이터를 확인
```

---

## 9.2 전처리

### ● 결측치 확인

❖ 다음과 같이 class2.csv 데이터셋을 확인할 수 있음

▼ 그림 9-18 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

## 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

- ❖ 다음은 결측치 개수에 대한 출력 결과

```
Unnamed: 0 0
id 0
tissue 0
class 0
class2 0
x 2
y 2
r 2
dtype: int64
```

- ❖ 결측치는 x, y, r 각각 두 개씩 존재

## 9.2 전처리

### ● 결측치 확인

- ❖ 전체 데이터 대비 결측치 비율을 확인해 보자

코드 9-10 결측치 비율

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

- ❖ 다음은 결측치 비율에 대한 출력 결과

```
Unnamed: 0 0.000000
id 0.000000
tissue 0.000000
class 0.000000
class2 0.000000
x 0.333333
y 0.333333
r 0.333333
dtype: float64
```

## 9.2 전처리

- 결측치 확인

### 결측치 처리하기

❖ 다음은 모든 행에 결측치가 존재한다면(모든 행이 NaN일 때) 해당 행을 삭제하는

코드 9-11 결측치 삭제 처리

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

```
print(df) ----- 데이터 확인(삭제 유무 확인)
```

---

## 9.2 전처리

- 결측치 확인

- ❖ 다음은 결측치를 삭제 처리하여 출력된 결과
- ❖ 모든 행에 NaN이 있는 것이 아니라서 삭제된 행이 없음

|   | 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  |

## 9.2 전처리

### ● 결측치 확인

- ❖ 다음은 결측치가 하나라도 존재한다면(데이터가 하나라도 NaN 값이 있을 때) 해당 행을 삭제하는 처리 방법

코드 9-12 결측치 삭제 처리

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

- ❖ 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 보겠음

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 보겠음

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
- ❖ NLTK를 이용하여 문장 토큰화를 구현해 보겠음

#### 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)
```

## 9.2 전처리

### ● 토큰화

- ❖ 다음은 문장 토큰화를 실행한 결과
- ❖ 정확하게 문장 단위로 구분되는 것을 확인할 수 있음

```
['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

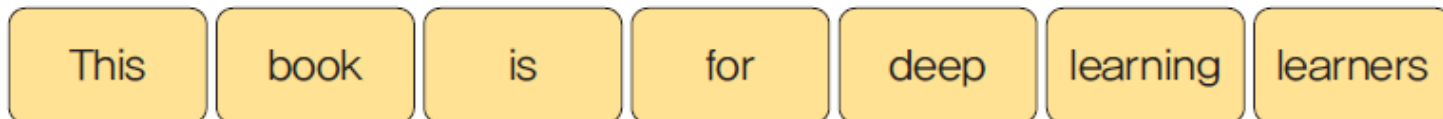
- 단어 토큰화

- ❖ 단어 토큰화는 다음과 같이 띄어쓰기를 기준으로 문장을 구분

- ▼ 그림 9-19 단어 토큰화

“This book is for deep learning learners”

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 보겠음

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)
```

---

## 9.2 전처리

### ● 토큰화

❖ 다음은 아포스트로피가 포함된 문장에서 단어 토큰화를 실행한 결과

```
['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

- ❖ 마지막으로 NLTK가 아닌 케라스를 이용하여 주어진 문장을 구분해 보겠음
- ❖ 케라스에서는 text\_to\_word\_sequence를 이용

코드 9-18 케라스를 이용한 단어 토큰화

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

- ❖ 다음은 케라스를 이용한 단어 토큰화를 실행한 결과

```
['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 9-19 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 9-20 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])
 r1 = (" ".join(r)).strip() ----- 형태소 사이에 공백 " "을 넣고, 양쪽 공백은 삭제
 result.append(r1)
print(r1)
```

## 9.2 전처리

### ● 토큰화

❖ 다음은 형태소 분석 결과

document

아 더빙 진짜 짜증나다 목소리

흙 포스터 보고 초딩 영화 줄 오버 연기 가볍다 앓다

너 무재 밉았 다그 래서 보다 추천 다

교도소 이야기 구면 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

### ● 토큰화

- ❖ 앞서 생성했던 형태소를 별도 파일로 저장
- ❖ 이 부분은 한국어 토큰화와 관련성은 없으나 사용 방법을 소개하기 위해 포함

코드 9-21 형태소 저장

```
with open("NaverMovie.nlp", 'w', encoding='utf-8') as fp:
 fp.write("\n".join(result))
```

- ❖ Word2Vec 모델을 생성한 후 저장

코드 9-22 Word2Vec 모델 생성

```
mData = word2vec.LineSentence("NaverMovie.nlp")
mModel = word2vec.Word2Vec(mData, size=200, window=10, hs=1, min_count=2, sg=1)
mModel.save("NaverMovie.model") ----- 모델 저장
```

- ❖ 한글에 대한 토큰화도 크게 다르지 않은 것을 확인할 수 있었음
- ❖ 토큰화를 왜 해야 하고 어떻게 하는지에 대한 방법만 알면 언어에 관계없이 수행할 수 있음

## 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

코드 9-23 불용어 제거

```
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)
```

## 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

## 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 목적이지만, 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 보겠음
- ❖ 먼저 포터 알고리즘을 적용해 보겠음

코드 9-24 포터 알고리즘

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

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 포터 알고리즘을 실행한 결과

obsess obsess

standard standard

nation nation

absent absent

tribal tribal

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



## 9.2 전처리

### ● 어간 추출

❖ 이번에는 랭커스터 알고리즘을 적용해 보겠음

코드 9-25 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 정밀도가 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 문장내에서 단어 의미도 고려하기 때문에 성능이 좋음
    - ❖ 어간 추출보다 시간이 더 오래 걸리는 단점이 있음

## 9.2 전처리

### ● 어간 추출

❖ 표제어 추출은 WordNetLemmatizer를 주로 사용

코드 9-26 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

- ❖ 다음은 표제어 추출을 실행한 결과

obsesses obsessed

standardizes standardization

national nation

absentness absently

tribalical tribalicalized

## 9.2 전처리

### ● 어간 추출

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

코드 9-27 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'))
```

## 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

- ❖ 정규화(normalization)는 표현 방법이 다른 단어들을 통합시켜서 같은 단어로 만들어 주는 것
- ❖ 예를 들어 USA와 US는 의미가 같으므로, 같은 의미로 해석되도록 만들어 주는 과정
- ❖ 머신 러닝/딥러닝은 데이터 특성들을 비교하여 패턴을 분석
- ❖ 이때 각각의 데이터가 갖는 스케일 차이가 크면 어떤 결과가 나타날까?
- ❖ 예를 들어 다음과 같은 데이터셋이 있다고 가정해보자
- ❖ MonthlyIncome은 0~10000의 범위를 갖지만, RelationshipSatisfaction은 0~5의 범위를 가짐
- ❖ 즉, MonthlyIncome과 RelationshipSatisfaction은 상당히 다른 값의 범위를 갖는데, 이 상태에서 데이터를 분석하면 MonthlyIncome 값이 더 크기 때문에 상대적으로 더 많은 영향을 미치게 됨
- ❖ 중요한 것은 값이 크다고 해서 분석에 더 중요한 요소라고 간주 할 수 없기 때문에 정규화가 필요한 것



## 9.2 전처리

▼ 표 9-2 정규화

| Monthly Income | Age | PercentSalary Hike | Relationship Satisfaction | TrainingTimes LastYear | YearsInCurrent Role |
|----------------|-----|--------------------|---------------------------|------------------------|---------------------|
| 5993           | 23  | 11                 | 1                         | 0                      | 4                   |
| 5130           | 55  | 23                 | 4                         | 3                      | 7                   |
| 2090           | 45  | 15                 | 2                         | 3                      | 0                   |
| 2909           | 60  | 11                 | 3                         | 3                      | 7                   |
| 3468           | 47  | 12                 | 4                         | 3                      | 2                   |
| 3068           | 51  | 13                 | 3                         | 2                      | 7                   |
| 2670           | 19  | 20                 | 1                         | 3                      | 0                   |
| 2693           | 33  | 22                 | 2                         | 2                      | 0                   |
| 9526           | 37  | 21                 | 2                         | 2                      | 7                   |
| 5237           | 59  | 13                 | 2                         | 3                      | 7                   |

## 9.2 전처리

### ● 정규화

- ❖ 좀 더 자세한 내용을 살펴보기 위해 예제 두 개를 진행해 보자
- ❖ 동일한 데이터셋을 이용하여 하나의 예제는 정규화를 진행하지 않았을 때의 정확도를 알아보고, 또 다른 예제는 정규화를 진행했을 때의 정확도를 알아보겠음
- ❖ 먼저 정규화를 진행하지 않았을 때의 예제를 살펴보는 데 필요한 라이브러리를 호출

코드 9-28 라이브러리 호출

```
import pandas as pd
from sklearn.model_selection import train_test_split
import tensorflow as tf
from tensorflow.python.data import Dataset
from tensorflow.keras.utils import to_categorical
from tensorflow.keras import models
from tensorflow.keras import layers
```

---

## 9.2 전처리

### ● 정규화

- ❖ 내려받은 예제 파일의 data 폴더에 있는 covtype.csv 파일을 메모리로 로딩
- ❖ covtype.csv 파일은 지역 4곳에 대한 환경과 나무들의 상태에 대해 정리한 데이터셋

코드 9-29 데이터셋 로딩 및 모델 훈련

```
df = pd.read_csv('../chap9/data/covtype.csv')
x = df[df.columns[:54]]
y = df.Cover_Type ----- 정답(레이블)을 Cover_Type 칼럼으로 지정

x_train, x_test, y_train, y_test = train_test_split(x, y, train_size=0.7,
 random_state=90) ----- 훈련과 검증 데이터셋으로
 분리하며, 전체 데이터셋 중
 70%를 훈련용으로 사용

model = tf.keras.Sequential([
 tf.keras.layers.Dense(64, activation='relu',
 input_shape=(x_train.shape[1],)),
 tf.keras.layers.Dense(64, activation='relu'),
 tf.keras.layers.Dense(8, activation='softmax')
]) ----- 출력층은 소프트맥스 활성화 함수 사용
```

## 9.2 전처리

### ● 정규화

```
model.compile(optimizer=tf.keras.optimizers.Adam(0.001),
 loss='sparse_categorical_crossentropy',
 metrics=['accuracy']) ----- y가 다중 분류가 가능한 값일 것이기 때문에
 sparse_categorical_crossentropy 손실 함수 사용
```

```
history1 = model.fit(
 x_train, y_train,
 epochs=26, batch_size=60,
 validation_data=(x_test, y_test)) ----- 모델 훈련
```

---

## 9.2 전처리

### ● 정규화

❖ 다음은 모델 훈련에 대한 출력 결과

Train on 406708 samples, validate on 174304 samples

Epoch 1/26

406708/406708 [=====] - 9s 23us/step - loss: 3.2084 -  
accuracy: 0.5765 - val\_loss: 1.1794 - val\_accuracy: 0.6811

Epoch 2/26

406708/406708 [=====] - 10s 23us/step - loss: 1.1763 -  
accuracy: 0.6387 - val\_loss: 0.8447 - val\_accuracy: 0.6538

Epoch 3/26

406708/406708 [=====] - 10s 25us/step - loss: 0.8390 -  
accuracy: 0.6687 - val\_loss: 0.7099 - val\_accuracy: 0.7046

Epoch 4/26

406708/406708 [=====] - 10s 23us/step - loss: 0.6936 -  
accuracy: 0.7037 - val\_loss: 0.6578 - val\_accuracy: 0.7269

Epoch 5/26

406708/406708 [=====] - 10s 25us/step - loss: 0.6383 -

## 9.2 전처리

- 정규화

```
accuracy: 0.7252 - val_loss: 0.5917 - val_accuracy: 0.7517
```

```
...(중간 생략)...
```

```
Epoch 21/26
```

```
406708/406708 [=====] - 11s 27us/step - loss: 0.5133 -
```

```
accuracy: 0.7821 - val_loss: 0.5160 - val_accuracy: 0.7827
```

```
Epoch 22/26
```

```
406708/406708 [=====] - 10s 25us/step - loss: 0.5144 -
```

```
accuracy: 0.7819 - val_loss: 0.4919 - val_accuracy: 0.7941
```

```
Epoch 23/26
```

```
406708/406708 [=====] - 11s 27us/step - loss: 0.5103 -
```

```
accuracy: 0.7838 - val_loss: 0.4969 - val_accuracy: 0.7932
```

```
Epoch 24/26
```

```
406708/406708 [=====] - 11s 27us/step - loss: 0.5086 -
```

```
accuracy: 0.7847 - val_loss: 0.4857 - val_accuracy: 0.7963
```

## 9.2 전처리

- 정규화

Epoch 25/26

406708/406708 [=====] - 10s 25us/step - loss: 0.5065 -  
accuracy: 0.7852 - val\_loss: 0.5235 - val\_accuracy: 0.7809

Epoch 26/26

406708/406708 [=====] - 10s 26us/step - loss: 0.5052 -  
accuracy: 0.7856 - val\_loss: 0.5003 - val\_accuracy: 0.7881

## 9.2 전처리

### ● 정규화

- ❖ 훈련 결과 검증 데이터셋에 대한 정확도가 78%이지만 손실도 50%로 상당히 높음
- ❖ 또한, 정확도가 거의 변화되지 않았는데 이것은 26 에포크가 진행되는 동안 모델의 학습이 진행되지 않았음을 의미
- ❖ 즉, 칼럼들이 비슷한 값의 범위를 갖지 않기 때문에 기울기가 앞뒤로 진동하거나 전역·지역 최소값에 도달하기까지 오랜 시간이 걸림
- ❖ 이러한 문제를 해결하려고 정규화를 진행
- ❖ 경사 하강법을 이용하여 빠르게 전역·지역 최소점을 찾기 위해 칼럼의 범위를 비슷한 값으로 취하도록 하는 방법



## 9.2 전처리

### ● 정규화

- ❖ 이번에는 정규화를 진행했을 때의 정확도에 대해 알아보자
- ❖ 먼저 데이터에 대한 정규화를 진행

코드 9-30 데이터 정규화

```
from sklearn import preprocessing
df = pd.read_csv('../chap9/data/covtype.csv')
x = df[df.columns[:55]]
y = df.Cover_Type
x_train, x_test, y_train, y_test = train_test_split(x, y, train_size=0.7,
 random_state=90)

train_norm = x_train[x_train.columns[0:10]] ----- 훈련 데이터셋에서 정규화가 필요한 칼럼 선택
test_norm = x_test[x_test.columns[0:10]] ----- 검증 데이터셋에서 정규화가 필요한 칼럼 선택

std_scale = preprocessing.StandardScaler().fit(train_norm) ----- ①
x_train_norm = std_scale.transform(train_norm)
```

## 9.2 전처리

### ● 정규화

```
training_norm_col = pd.DataFrame(x_train_norm, index=train_norm.index,
 columns=train_norm.columns) ----- 넘파이(numpy) 배열을 데이터프레임
 (DataFrame)으로 변환
x_train.update(training_norm_col)
print(x_train.head())

x_test_norm = std_scale.transform(test_norm) ----- 검증 데이터셋 정규화
testing_norm_col = pd.DataFrame(x_test_norm, index=test_norm.index,
 columns=test_norm.columns)
x_test.update(testing_norm_col)
print(x_test.head())
```

---

## 9.2 전처리

## ● 정규화

- ❖ ① 정규화 방법은 예제에서 구현한 StandardScaler() 외에도 세 가지가 더 있음
  - ㉠ StandardScaler(): 각 특성의 평균을 0, 분산을 1로 변경하여 특성의 스케일을 조정

StandardScaler()를 구하는 공식은 다음과 같음

$$\text{StandardScaler}() = \frac{x - \mu}{\sigma}$$

( $x$ : 입력 데이터,  $\mu$ : 평균,  $\sigma$ : 표준편차)

다음은 StandardScaler()를 구현하는 예시 코드

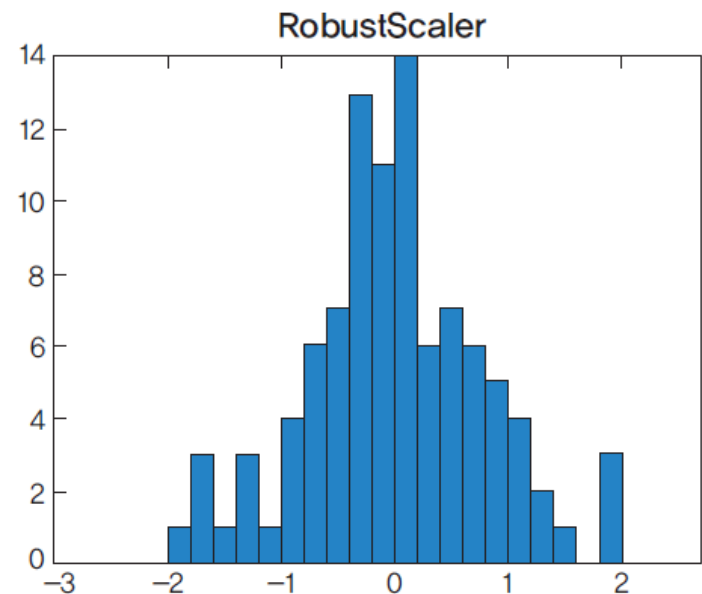
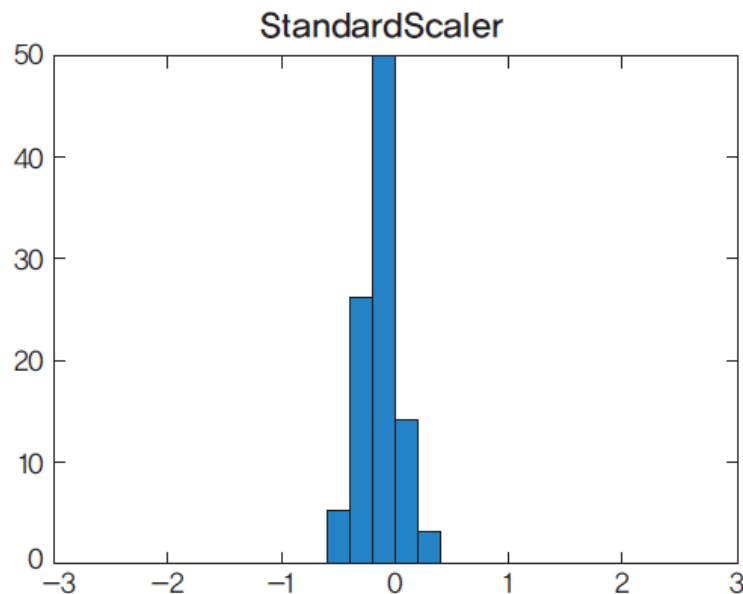
```
from sklearn.preprocessing import StandardScaler
standardScaler = StandardScaler() ----- StandardScaler 객체 생성
print(standardScaler.fit(train_data)) ----- fit() 메서드를 사용하여 훈련 데이터셋을 적용
train_data_standardScaled = standardScaler.transform(train_data) -----
transform() 메서드를 사용하여 훈련 데이터셋을 적용
```

## 9.2 전처리

### ● 정규화

- ㉞ RobustScaler(): 평균과 분산 대신 중간 값(median)과 사분위수 범위 (InterQuartile Range, IQR)를 사용 StandardScaler()와 비교하면 다음 그림과 같이 정규화 이후 동일한 값이 더 넓게 분포되어 있는 것을 확인할 수 있음

▼ 그림 9-20 StandardScaler와 RobustScaler 비교



## 9.2 전처리

- 정규화

- ❖ 다음은 RobustScaler()를 구현하는 예시 코드

```
from sklearn.preprocessing import RobustScaler
robustScaler = RobustScaler()
print(robustScaler.fit(train_data))
train_data_robustScaled = robustScaler.transform(train_data)
```

## 9.2 전처리

### ● 정규화

#### 사분위수 범위(IQR)

- ❖ 사분위수란 전체 관측 값을 오름차순으로 정렬한 후 전체를 사등분하는 값을 나타냄
- ❖ 다음과 같이 표현할 수 있음
  - 제1사분위수 =  $Q1$  = 제25백분위수
  - 제2사분위수 =  $Q2$  = 제50백분위수
  - 제3사분위수 =  $Q3$  = 제75백분위수
- ❖ 이때 제3사분위수와 제1사분위수 사이 거리를 자료의 흩어진 정도의 척도로 사용할 수 있는데, 이 수치를 사분위수 범위(IQR)라고 함
- ❖ 사분위수 범위는 다음과 같이 표현할 수 있음

사분위수 범위:  $IQR = \text{제3사분위수} - \text{제1사분위수} = Q3 - Q1$

## 9.2 전처리

### ● 정규화

- ㉔ MinMaxScaler(): 모든 특성이 0과 1 사이에 위치하도록 스케일을 조정 이때 이상치의 데이터가 있을 경우 반환된 값이 매우 좁은 범위로 압축될 수 있음 즉, 이상치에 매우 민감할 수 있기 때문에 주의해야 함

MinMaxScaler()를 구하는 공식은 다음과 같음

$$\text{MinMaxScaler}() = \frac{x - x_{\min}}{x_{\max} - x_{\min}}$$

( $x$ : 입력 데이터)

다음은 MinMaxScaler()를 구현하는 예시 코드

```
from sklearn.preprocessing import MinMaxScaler
minMaxScaler = MinMaxScaler()
print(minMaxScaler.fit(train_data))
train_data_minMaxScaled = minMaxScaler.transform(train_data)
```

## 9.2 전처리

### ● 정규화

④ MaxAbsScaler(): 절댓값이 0~1 사이가 되도록 조정

즉, 모든 데이터가 -1~1의 사이가 되도록 조정하기 때문에 양의 수로만 구성된 데이터는 MinMaxScaler()와 유사하게 동작  
또한, 큰 이상치에 민감하다는 단점이 있음

다음은 MaxAbsScaler()를 구현하는 예시 코드

```
from sklearn.preprocessing import MaxAbsScaler
maxAbsScaler = MaxAbsScaler()
print(maxAbsScaler.fit(train_data))
train_data_maxAbsScaled = maxAbsScaler.transform(train_data)
```



## 9.2 전처리

### ● 정규화

- ❖ 다음은 코드 9-30으로 데이터 정규화를 진행한 후 x\_train과 x\_test에 대한 head() 정보를 출력한 결과

|        | Elevation | Aspect    | Slope     | Horizontal_Distance_To_Hydrology | \ |
|--------|-----------|-----------|-----------|----------------------------------|---|
| 152044 | 0.222366  | -0.228639 | -0.412503 | 0.148486                         |   |
| 363373 | 1.980490  | -0.469989 | 0.255453  | 3.018822                         |   |
| 372733 | -1.081933 | 0.271939  | 0.389044  | -0.867895                        |   |
| 572846 | -1.164122 | -0.157128 | -0.278912 | -1.267860                        |   |
| 114145 | -0.052787 | 0.861906  | 0.255453  | -0.279711                        |   |

|        | Vertical_Distance_To_Hydrology | Horizontal_Distance_To_Roadways | \ |
|--------|--------------------------------|---------------------------------|---|
| 152044 | 0.149095                       | 1.336119                        |   |
| 363373 | 4.443372                       | 0.168073                        |   |
| 372733 | -0.160093                      | -0.241801                       |   |
| 572846 | -0.795646                      | -0.461170                       |   |
| 114145 | -0.125739                      | 1.811419                        |   |

## 9.2 전처리

- 정규화

|        | Hillshade_9am | Hillshade_Noon | Hillshade_3pm | \ |
|--------|---------------|----------------|---------------|---|
| 152044 | 1.002687      | 0.539776       | -0.510339     |   |
| 363373 | 1.227001      | -0.270132      | -1.190275     |   |
| 372733 | 0.292357      | 1.349684       | 0.378807      |   |
| 572846 | 0.965301      | 0.641014       | -0.431885     |   |
| 114145 | -1.090917     | 1.299065       | 1.581770      |   |

|        | Horizontal_Distance_To_Fire_Points | ... | Soil_Type32 | Soil_Type33 | \ |
|--------|------------------------------------|-----|-------------|-------------|---|
| 152044 | -0.111226                          | ... | 0           | 0           |   |
| 363373 | -0.703030                          | ... | 0           | 0           |   |
| 372733 | 0.038235                           | ... | 0           | 0           |   |
| 572846 | -1.450334                          | ... | 0           | 0           |   |
| 114145 | -0.328623                          | ... | 0           | 0           |   |

## 9.2 전처리

- 정규화

|        | Soil_Type34 | Soil_Type35 | Soil_Type36 | Soil_Type37 | Soil_Type38 | \ |
|--------|-------------|-------------|-------------|-------------|-------------|---|
| 152044 | 0           | 0           | 0           | 0           | 0           |   |
| 363373 | 0           | 0           | 0           | 0           | 0           |   |
| 372733 | 0           | 0           | 0           | 0           | 0           |   |
| 572846 | 0           | 0           | 0           | 0           | 0           |   |
| 114145 | 0           | 0           | 0           | 0           | 0           |   |

|        | Soil_Type39 | Soil_Type40 | Cover_Type |
|--------|-------------|-------------|------------|
| 152044 | 0           | 0           | 2          |
| 363373 | 0           | 1           | 1          |
| 372733 | 0           | 0           | 3          |
| 572846 | 0           | 0           | 2          |
| 114145 | 0           | 0           | 2          |

[5 rows x 55 columns]

## 9.2 전처리

- 정규화

|        | Elevation | Aspect    | Slope     | Horizontal_Distance_To_Hydrology | \ |
|--------|-----------|-----------|-----------|----------------------------------|---|
| 204886 | 0.783394  | -1.310245 | -0.946867 | 0.233185                         |   |
| 116027 | -0.903262 | -1.006323 | -0.679685 | -1.267860                        |   |
| 328145 | -0.270766 | -1.095711 | -0.278912 | 0.379054                         |   |
| 579670 | -1.139108 | -0.961628 | -0.412503 | 0.454342                         |   |
| 41341  | 0.265247  | 0.736762  | -1.347641 | 2.708261                         |   |

|        | Vertical_Distance_To_Hydrology | Horizontal_Distance_To_Roadways | \ |
|--------|--------------------------------|---------------------------------|---|
| 204886 | -1.465554                      | 0.093026                        |   |
| 116027 | -0.795646                      | -0.611906                       |   |
| 328145 | 0.200626                       | -0.948657                       |   |
| 579670 | 0.389574                       | -0.772905                       |   |
| 41341  | 2.072931                       | 2.321998                        |   |

## 9.2 전처리

- 정규화

|        | Hillshade_9am | Hillshade_Noon | Hillshade_3pm | \ |
|--------|---------------|----------------|---------------|---|
| 204886 | -0.006729     | 0.084203       | 0.221899      |   |
| 116027 | 0.367128      | -0.168893      | -0.274977     |   |
| 328145 | 0.217585      | -0.472609      | -0.301128     |   |
| 579670 | 0.441900      | -0.421989      | -0.510339     |   |
| 41341  | -0.006729     | 1.045968       | 0.718775      |   |

|        | Horizontal_Distance_To_Fire_Points | ... | Soil_Type32 | Soil_Type33 | \ |
|--------|------------------------------------|-----|-------------|-------------|---|
| 204886 | 0.253368                           | ... | 0           | 0           |   |
| 116027 | 0.226194                           | ... | 0           | 0           |   |
| 328145 | -0.330133                          | ... | 0           | 0           |   |
| 579670 | -0.882685                          | ... | 0           | 0           |   |
| 41341  | 1.243735                           | ... | 0           | 0           |   |

## 9.2 전처리

- 정규화

|        | Soil_Type34 | Soil_Type35 | Soil_Type36 | Soil_Type37 | Soil_Type38 | \ |
|--------|-------------|-------------|-------------|-------------|-------------|---|
| 204886 | 0           | 0           | 0           | 0           | 0           |   |
| 116027 | 0           | 0           | 0           | 0           | 0           |   |
| 328145 | 0           | 0           | 0           | 0           | 0           |   |
| 579670 | 0           | 0           | 0           | 0           | 0           |   |
| 41341  | 0           | 0           | 0           | 0           | 0           |   |

|        | Soil_Type39 | Soil_Type40 | Cover_Type |
|--------|-------------|-------------|------------|
| 204886 | 0           | 0           | 1          |
| 116027 | 0           | 0           | 2          |
| 328145 | 0           | 0           | 2          |
| 579670 | 0           | 0           | 3          |
| 41341  | 0           | 0           | 2          |

[5 rows x 55 columns]

## 9.2 전처리

### ● 정규화

- ❖ 훈련과 검증 데이터셋의 정규화도 마쳤으니 모델 훈련을 진행해 보자
- ❖ 다음 코드는 앞서 사용했던 코드와 동일하며, 단지 데이터셋의 정규화 유무만 다름

코드 9-31 데이터셋 로딩 및 모델 훈련

```
model = tf.keras.Sequential([
 tf.keras.layers.Dense(64, activation='relu',
 input_shape=(x_train.shape[1],)),
 tf.keras.layers.Dense(64, activation='relu'),
 tf.keras.layers.Dense(8, activation='softmax')
])

model.compile(optimizer=tf.keras.optimizers.Adam(0.001),
 loss='sparse_categorical_crossentropy',
 metrics=['accuracy'])

history2 = model.fit(
 x_train, y_train,
 epochs=26, batch_size=60,
 validation_data=(x_test, y_test))
```

## 9.2 전처리

### ● 정규화

❖ 다음은 모델을 훈련시킨 결과

Train on 406708 samples, validate on 174304 samples

Epoch 1/26

406708/406708 [=====] - 10s 24us/step - loss: 0.0293 -  
accuracy: 0.9922 - val\_loss: 0.0015 - val\_accuracy: 0.9994

Epoch 2/26

406708/406708 [=====] - 10s 26us/step - loss: 3.5381e-04 -  
accuracy: 0.9999 - val\_loss: 1.5352e-05 - val\_accuracy: 1.0000

Epoch 3/26

406708/406708 [=====] - 10s 26us/step - loss: 3.6593e-04 -  
accuracy: 0.9999 - val\_loss: 1.3790e-04 - val\_accuracy: 1.0000

Epoch 4/26

406708/406708 [=====] - 10s 25us/step - loss: 1.4982e-04 -  
accuracy: 1.0000 - val\_loss: 3.2008e-06 - val\_accuracy: 1.0000

Epoch 5/26

406708/406708 [=====] - 10s 25us/step - loss: 1.4265e-04 -



## 9.2 전처리

### ● 정규화

```
accuracy: 1.0000 - val_loss: 5.1801e-06 - val_accuracy: 1.0000
```

```
...(중간 생략)...
```

```
Epoch 21/26
```

```
406708/406708 [=====] - 11s 27us/step - loss: 1.9403e-08 -
```

```
accuracy: 1.0000 - val_loss: 4.8524e-06 - val_accuracy: 1.0000
```

```
Epoch 22/26
```

```
406708/406708 [=====] - 11s 27us/step - loss: 1.8890e-04 -
```

```
accuracy: 1.0000 - val_loss: 7.9978e-06 - val_accuracy: 1.0000
```

```
Epoch 23/26
```

```
406708/406708 [=====] - 11s 27us/step - loss: 6.7900e-08 -
```

```
accuracy: 1.0000 - val_loss: 8.8818e-08 - val_accuracy: 1.0000
```

```
Epoch 24/26
```

```
406708/406708 [=====] - 11s 28us/step - loss: 1.1787e-04 -
```

```
accuracy: 1.0000 - val_loss: 5.3235e-04 - val_accuracy: 0.9999
```

## 9.2 전처리

- 정규화

Epoch 25/26

406708/406708 [=====] - 11s 28us/step - loss: 8.3980e-07 -  
accuracy: 1.0000 - val\_loss: 7.2145e-08 - val\_accuracy: 1.0000

Epoch 26/26

406708/406708 [=====] - 11s 27us/step - loss: 1.3516e-08 -  
accuracy: 1.0000 - val\_loss: 1.1551e-09 - val\_accuracy: 1.0000

## 9.2 전처리

- 정규화

- ❖ 정규화 이후 검증 데이터셋을 사용한 정확도가 100%가 되었음(이미 두 번째 에포크에서 검증 데이터셋에 대한 정확도가 100%가 되었기 때문에 예제에서 사용한 데이터셋은 이미 어느 정도 정규화가 되어 있다고 생각하면 되고 일반적인 데이터셋에서는 한두 번의 에포크로 정확도 100%를 얻는 것은 쉽지 않음)

# 자연어 처리를 위한 임베딩

---

- 1 임베딩
- 2 트랜스포머 어텐션
- 3 한국어 임베딩

# 임베딩

---

## 10.1 임베딩

### ● 임베딩

- ❖ 임베딩(embedding)은 사람이 사용하는 언어(자연어)를 컴퓨터가 이해할 수 있는 언어(숫자) 형태인 벡터(vector)로 변환한 결과 혹은 일련의 과정을 의미
- ❖ 임베딩 역할은 다음과 같음
  - 단어 및 문장 간 관련성 계산
  - 의미적 혹은 문법적 정보의 함축(예 왕-여왕, 교사-학생)
- ❖ 임베딩 방법에 따라 희소 표현 기반 임베딩, 횡수 기반 임베딩, 예측 기반 임베딩, 횡수/예측 기반 임베딩이 있음

## 10.1 임베딩

### ● 희소 표현 기반 임베딩

- ❖ 희소 표현(sparse representation)은 대부분의 값이 0으로 채워져 있는 경우로, 대표적으로 원-핫 인 코딩이 있음

### ● 원-핫 인코딩

- ❖ 원-핫 인코딩(one-hot encoding)이란 주어진 텍스트를 숫자(벡터)로 변환해 주는 것
- ❖ 다시말해 단어 N개를 각각 N차원의 벡터로 표현하는 방식으로, 단어가 포함되어 있는 위치에 1을 넣고 나머지는 0 값을 채움
- ❖ 예를 들어 딕셔너리에 [calm, fast, cat] 같은 값이 있다면 fast를 표현하는 벡터는 [0, 1, 0]이 됨

## 10.1 임베딩

### ▼ 그림 10-1 원-핫 인코딩





## 10.1 임베딩

### ● 희소 표현 기반 임베딩

- ❖ 사이킷런을 이용하여 원-핫 인코딩을 적용한 예제를 살펴보자
- ❖ 9장에서 사용한 class2.csv 파일을 사용하여 예제를 진행

코드 10-1 원-핫 인코딩 적용

```
import pandas as pd
class2 = pd.read_csv("../chap10\data\class2.csv") ----- 데이터셋을 메모리로 로딩

from sklearn import preprocessing
label_encoder = preprocessing.LabelEncoder() ----- 데이터를 인코딩하는 데 사용하며, 다음의
onehot_encoder = preprocessing.OneHotEncoder() ----- OneHotEncoder()와 함께 사용
----- 데이터를 숫자 형식으로 표현

train_x = label_encoder.fit_transform(class2['class2'])
train_x
```

---

## 10.1 임베딩

- 희소 표현 기반 임베딩

- ❖ 다음은 원-핫 인코딩을 적용한 결과

```
array([2, 2, 1, 0, 1, 0])
```

## 10.1 임베딩

### ● 희소 표현 기반 임베딩

- ❖ 원-핫 인코딩에는 치명적인 단점이 있음
- ❖ 첫째, 수학적 의미에서 원-핫 벡터들은 하나의 요소만 1 값을 갖고 나머지는 모두 0인 희소 벡터(sparse vector)를 가짐
  - 이때 두 단어에 대한 벡터의 내적(inner product)을 구해 보면 0 값을 갖게 되므로 직교(orthogonal)를 이룸
  - 단어끼리 관계성(유의어, 반의어) 없이 서로 독립적(independent)인 관계가 됨
- ❖ 둘째, '차원의 저주(curse of dimensionality)' 문제가 발생
  - 하나의 단어를 표현하는 데 말뭉치(corpus)에 있는 수만큼 차원이 존재하기 때문에 복잡해짐
  - 예를 들어 단어 10만 개를 포함한 데이터셋에 원-핫 인코딩 배열을 구성한다면 그 차원 개수는 10만 개에 이르게 됨

## 10.1 임베딩

- 희소 표현 기반 임베딩

- ❖ 원-핫 인코딩에 대한 대안으로 신경망에 기반하여 단어를 벡터로 바꾸는 방법론들이 주목을 받고 있음
- ❖ 예를 들어 워드투벡터(Word2Vec), 글로브(GloVe), 패스트텍스트(FastText) 등이 대표적인 방법론

## 10.1 임베딩

- **횟수 기반 임베딩**

- ❖ 횟수 기반은 단어가 출현한 빈도를 고려하여 임베딩하는 방법
- ❖ 대표적으로 카운터 벡터와 TF-IDF가 있음

## 10.1 임베딩

### ● 횡수 기반 임베딩

#### 카운터 벡터

- ❖ 카운터 벡터(counter vector)는 문서 집합에서 단어를 토큰으로 생성하고 각 단어의 출현 빈도수를 이용하여 인코딩해서 벡터를 만드는 방법
- ❖ 즉, 토큰나이징과 벡터화가 동시에 가능한 방법
- ❖ 카운터 벡터는 사이킷런의 CountVectorizer()를 사용하여 코드로 구현할 수 있음
- ❖ CountVectorizer()는 다음 작업이 가능

1. 문서를 토큰 리스트로 변환
2. 각 문서에서 토큰의 출현 빈도를 셈
3. 각 문서를 인코딩하고 벡터로 변환

## 10.1 임베딩

### ● 횃수 기반 임베딩

- ❖ 다음은 사이킷런을 이용한 예제
- ❖ 코퍼스를 정의하고 CountVectorizer() 객체를 생성

코드 10-2 코퍼스에 카운터 벡터 적용

```
from sklearn.feature_extraction.text import CountVectorizer
corpus = [
 'This is last chance.',
 'and if you do not have this chance.',
 'you will never get any chance.',
 'will you do get this one?',
 'please, get this chance',
]
vect = CountVectorizer()
vect.fit(corpus)
vect.vocabulary_
```

## 10.1 임베딩

- **횃수 기반 임베딩**

- ❖ 다음은 코퍼스에 카운터 벡터를 적용한 결과

```
{'this': 13,
 'is': 7,
 'last': 8,
 'chance': 2,
 'and': 0,
 'if': 6,
 'you': 15,
 'do': 3,
```



## 10.1 임베딩

- 횃수 기반 임베딩

```
'not': 10,
'have': 5,
'will': 14,
'never': 9,
'get': 4,
'any': 1,
'one': 11,
'please': 12}
```

## 10.1 임베딩

- **횃수 기반 임베딩**

- ❖ 이번에는 CountVectorizer() 적용 결과를 배열로 변환해 보자

코드 10-3 배열 변환

```
vect.transform(['you will never get any chance.']).toarray()
```

- ❖ 다음은 배열로 변환한 출력 결과

```
array([[0, 1, 1, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 1]], dtype=int64)
```

## 10.1 임베딩

- **횟수 기반 임베딩**

- ❖ 이번에는 불용어를 제거한 카운터 벡터를 확인해 보자

코드 10-4 불용어를 제거한 카운터 벡터

```
stop_words를 사용하여 is, not, an 같은 불용어 제거
vect = CountVectorizer(stop_words=["and", "is", "please", "this"]).fit(corpus)
vect.vocabulary_
```

## 10.1 임베딩

- **횃수 기반 임베딩**

- ❖ 불용어를 제거한 카운터 벡터가 다음과 같이 출력

```
{'last': 6,
 'chance': 1,
 'if': 5,
 'you': 11,
 'do': 2,
 'not': 8,
 'have': 4,
 'will': 10,
 'never': 7,
 'get': 3,
 'any': 0,
 'one': 9}
```

## 10.1 임베딩

- 횡수 기반 임베딩

### TF-IDF

- ❖ TF-IDF(Term Frequency-Inverse Document Frequency)는 정보 검색론(Information Retrieval, IR)에서 가중치를 구할 때 사용되는 알고리즘

## 10.1 임베딩

### ● 횃수 기반 임베딩

- ❖ TF(Term Frequency)(단어 빈도)는 문서 내에서 특정 단어가 출현한 빈도를 의미
- ❖ 예를 들어 TF에 딥러닝과 신문기사라는 단어가 포함되어 있다고 가정함
- ❖ 이것은 '신문기사'에서 '딥러닝'이라는 단어가 몇 번 등장했는지 의미
- ❖ 즉, '신문기사'에서 '딥러닝'이라는 단어가 많이 등장한다면 이 기사는 딥러닝과 관련이 높다고 할 수 있으며, 다음 수식을 사용
- ❖ 이때  $tf_{t,d}$ 는 특정 문서  $d$ 에서 특정 단어  $t$ 의 등장 횃수를 의미

$$tf_{t,d} = \begin{cases} 1 + \log count(t, d) & count(t, d) > 0 \text{ 일 때} \\ 0 & \text{그 외} \end{cases}$$

- ❖ 혹은

$$tf_{t,d} = \log(count(t, d) + 1)$$

( $t$ (term): 단어,  $d$ (document): 문서 한 개)

## 10.1 임베딩

### ● 횡수 기반 임베딩

- ❖ IDF(Inverse Document Frequency)(역문서 빈도)를 이해하려면 DF(Document Frequency)(문서 빈도)에 대한 개념부터 이해해야 함
- ❖ DF는 한 단어가 전체 문서에서 얼마나 공통적으로 많이 등장하는지 나타내는 값
- ❖ 즉, 특정 단어가 나타난 문서 개수라고 이해하면 됨

$df_t$  = 특정 단어  $t$ 가 포함된 문서 개수

## 10.1 임베딩

### ● 횃수 기반 임베딩

- ❖ 특정 단어  $t$ 가 모든 문서에 등장하는 일반적인 단어(예 a, the)라면, TF-IDF 가중치를 낮추어 줄 필요가 있음
- ❖ DF 값이 클수록 TF-IDF의 가중치 값을 낮추기 위해 DF 값에 역수를 취하는데, 이 값이 IDF
- ❖ 역수를 취하면 전체 문서 개수가 많아질수록 IDF 값도 커지므로 IDF는 로그(log)를 취해야 함
- ❖ 이것을 수식으로 표현하면 다음과 같음

$$idf_t = \log\left(\frac{N}{df_t}\right) = \log\left(\frac{\text{전체 문서 개수}}{\text{특정 단어 } t \text{가 포함된 문서 개수}}\right)$$



## 10.1 임베딩

### ● 횃수 기반 임베딩

- ❖ 이때 중요한 점은 전체 문서에 특정 단어가 발생하는 빈도가 0이라면 분모가 0이 되는 상황이 발생
- ❖ 이를 방지하고자 다음과 같이 분모에 1을 더해 주는 것을 스무딩(smoothing)이라고 함

$$idf_t = \log\left(\frac{N}{1 + df_t}\right) = \log\left(\frac{\text{전체 문서 개수}}{1 + \text{특정 단어 } t \text{가 포함된 문서 개수}}\right)$$

## 10.1 임베딩

- **횟수 기반 임베딩**

- ❖ TF-IDF는 다음 상황에서 사용

- 키워드 검색을 기반으로 하는 검색 엔진
    - 중요 키워드 분석
    - 검색 엔진에서 검색 결과의 순위를 결정

## 10.1 임베딩

### ● 횃수 기반 임베딩

- ❖ 사이킷런의 TfidfVectorizer()를 이용한 TF-IDF 예제를 살펴보겠음
- ❖ 코퍼스를 정의하고 TfidfVectorizer()를 적용한 후 유사도를 계산하여 행렬로 표현

코드 10-5 TF-IDF를 적용한 후 행렬로 표현

```
from sklearn.feature_extraction.text import TfidfVectorizer
doc = ['I like machine learning', 'I love deep learning', 'I run everyday']
tfidf_vectorizer = TfidfVectorizer(min_df=1)
tfidf_matrix = tfidf_vectorizer.fit_transform(doc)
doc_distance = (tfidf_matrix * tfidf_matrix.T)
print('유사도를 위한', str(doc_distance.get_shape()[0]), 'x', str(doc_distance.get_shape()[1]), 행렬을 만들었습니다.')
print(doc_distance.toarray())
```

## 10.1 임베딩

### ● 희수 기반 임베딩

- ❖ 다음은 TF-IDF를 적용한 후 행렬로 표현한 결과

유사도를 위한 3 x 3 행렬을 만들었습니다.

```
[[1. 0.224325 0.]
 [0.224325 1. 0.]
 [0. 0. 1.]]
```

- ❖ TF-IDF 값은 특정 문서 내에서 단어의 출현 빈도가 높거나 전체 문서에서 특정 단어가 포함된 문서가 적을수록 TF-IDF 값이 높음
- ❖ 이 값을 사용하여 문서에 나타나는 흔한 단어(예 a, the)들을 걸러 내거나 특정 단어에 대한 중요도를 찾을 수 있음

## 10.1 임베딩

### ● 예측 기반 임베딩

- ❖ 예측 기반 임베딩은 신경망 구조 혹은 모델을 이용하여 특정 문맥에서 어떤 단어가 나올지를 예측하면서 단어를 벡터로 만드는 방식
- ❖ 대표적으로 워드투벡터가 있음

## 10.1 임베딩

### ● 예측 기반 임베딩

#### 워드투벡터

- ❖ 워드투벡터(Word2Vec)는 신경망 알고리즘으로, 주어진 텍스트에서 텍스트의 각 단어마다 하나씩 일련의 벡터를 출력
- ❖ 워드투벡터의 출력 벡터가 2차원 그래프에 표시될 때, 의미론적으로 유사한 단어의 벡터는 서로 가깝게 표현
- ❖ 이때 '서로 가깝다'는 의미는 코사인 유사도를 이용하여 단어 간의 거리를 측정한 결과로 나타나는 관계성을 의미
- ❖ 즉, 워드투벡터를 이용하면 특정 단어의 동의어를 찾을 수 있음

## 10.1 임베딩

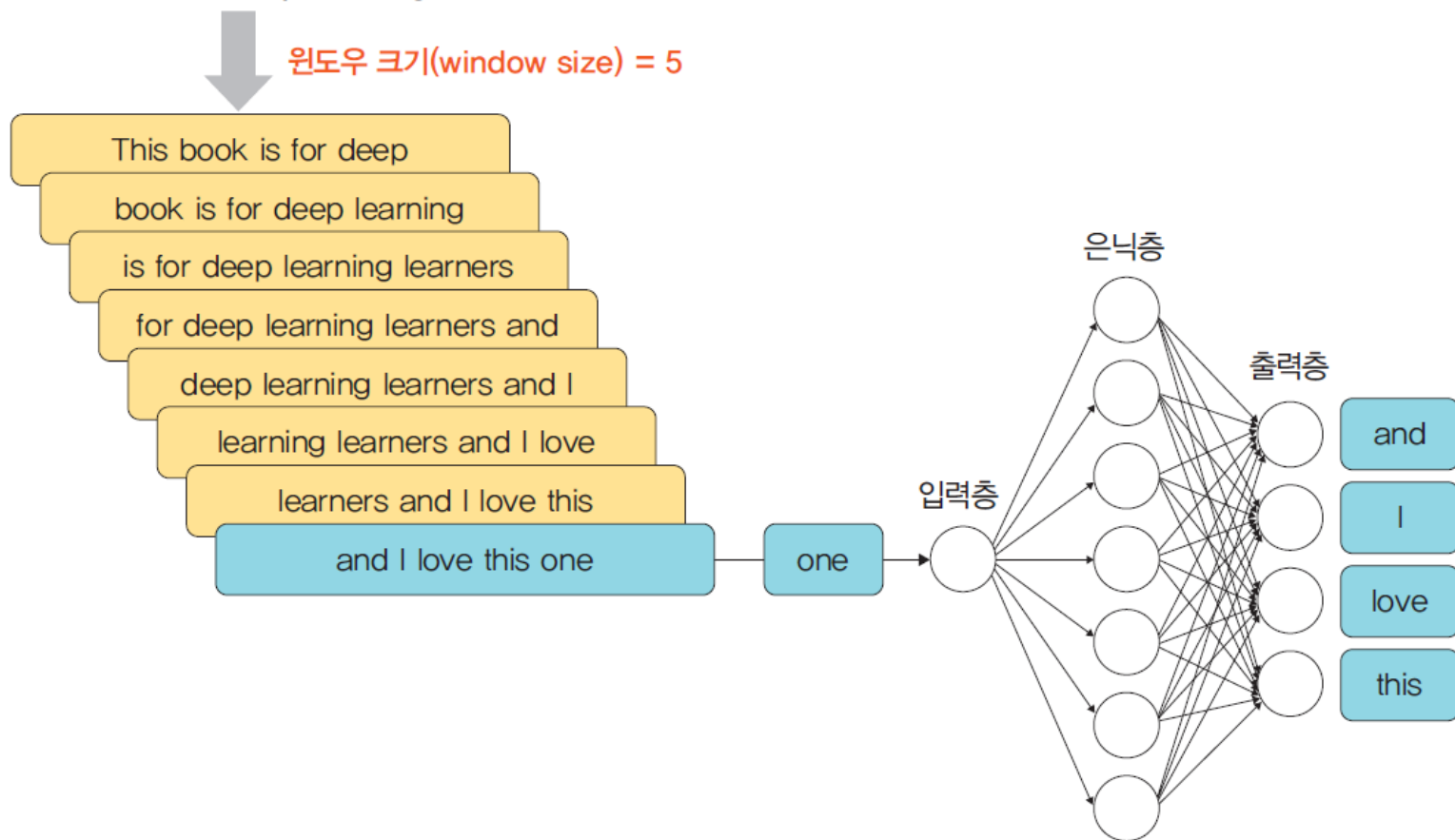
### ● 예측 기반 임베딩

- ❖ 워드투벡터가 수행되는 과정은 다음과 같음
- ❖ 일정한 크기의 윈도우(window)로 분할된 텍스트를 신경망 입력으로 사용
- ❖ 이때 모든 분할된 텍스트는 한 쌍의 대상 단어와 컨텍스트로 네트워크에 공급
- ❖ 다음 그림과 같이 대상 단어는 'one'이고 컨텍스트는 'and', 'I', 'love', 'this' 단어로 구성
- ❖ 또한, 네트워크의 은닉층에는 각 단어에 대한 가중치가 포함되어 있음

## 10.1 임베딩

### ▼ 그림 10-2 워드투벳터

“This book is for deep learning learners and I love this one”





## 10.1 임베딩

### ● 예측 기반 임베딩

- ❖ 워드투벡터를 이용하여 텍스트를 벡터로 변환하는 예제를 살펴보자
- ❖ 처음에 할 일은 필요한 모든 라이브러리를 호출하고 텍스트 데이터셋(peter.txt)을 메모리로 로딩
- ❖ 메모리로 로딩된 데이터셋에 NLTK의 word\_tokenize를 적용하여 토큰화

코드 10-6 데이터셋을 메모리로 로딩하고 토큰화 적용

```
from nltk.tokenize import sent_tokenize, word_tokenize
import warnings
warnings.filterwarnings(action='ignore')
import gensim
from gensim.models import Word2Vec

sample = open("../chap10\data\peter.txt", "r", encoding='UTF8') ----- 피터팬 데이터셋 로딩
s = sample.read()

f = s.replace("\n", " ") ----- 줄바꿈(\n)을 공백(" ")으로 변환
data = []
```

## 10.1 임베딩

### ● 예측 기반 임베딩

```
for i in sent_tokenize(f): ----- 로딩한 파일의 각 문장마다 반복
 temp = []
 for j in word_tokenize(i): ----- 문장을 단어로 토큰화
 temp.append(j.lower()) ----- 토큰화된 단어를 소문자로 변환하여 temp에 저장
 data.append(temp)
```

data

---

## 10.1 임베딩

- 예측 기반 임베딩

- ❖ 다음은 코퍼스에 토큰화를 진행한 결과

```
[['once',
 'upon',
 'a',
 'time',
 'in',
 'london',
 ',',
 'the',
 'darlings',
 'went',
 'out',
```

## 10.1 임베딩

- 예측 기반 임베딩

```
'to',
'a',
'dinner',
'party',
'leaving',
'their',
'three',
'children',
'wendy',
,,
'jhon',
,,
'and',
```

## 10.1 임베딩

- 예측 기반 임베딩

```
'michael',
'at',
'home',
'.'],
...(이하 생략)...
```

- ❖ 출력 결과를 보면 단어 기준으로 토큰화가 되어 있는데, CBOW와 skip-gram을 이용하여 단어간 유사성을 살펴보자