Chapter 09

텍스트 분석

텍스트 분석의 이해 – NLP와 텍스트 분석

NLP

(Natural Language Processing)

인간의 언어를 이해하고 해석하는데 더 중점을 두고
기술이 발전해 옴
NLP 기술의 발전으로 텍스트 분석도 더욱 정교하게 발전

텍스트 분석

텍스트 분석은 머신러닝, 언어 이해, 통계 등을 활용해
모델을 수립하고 정보를 추출해 비즈니스 인텔리전스
(Business Intelligence)나 예측 분석 등의 분석 작업을
주로 수행합니다

텍스트 분석 주요 영역

텍스트 분류 (Text Classification)

문서가 특정 분류 또는 카테고리에 속하는 것을 예측하는 기법을 통칭합니다. 예를 들어 특정 신문 기사 내용이 연애/정치/사회/문화 중 어떤 카테고리에 속하는지 자동으로 분류하거나 스팸 메일 검출 같은 프로그램이 이에 속합니다. 지도학습을 적용합니다.

감성 분석 (Sentiment Analysis)

텍스트에서 나타나는 감정/판단/믿음/의견/기분 등의 주관적인 요소를 분석하는 기법을 총칭합니다. 소셜 미디어 감정 분석, 영화나 제품에 대한 긍정 또는 리뷰, 여론조사 의견 분석 등의 다양한 영역에서 활용됩니다. 지도학습 방법뿐만 아니라 비지도학습을 이용해 적용할 수 있습니다.

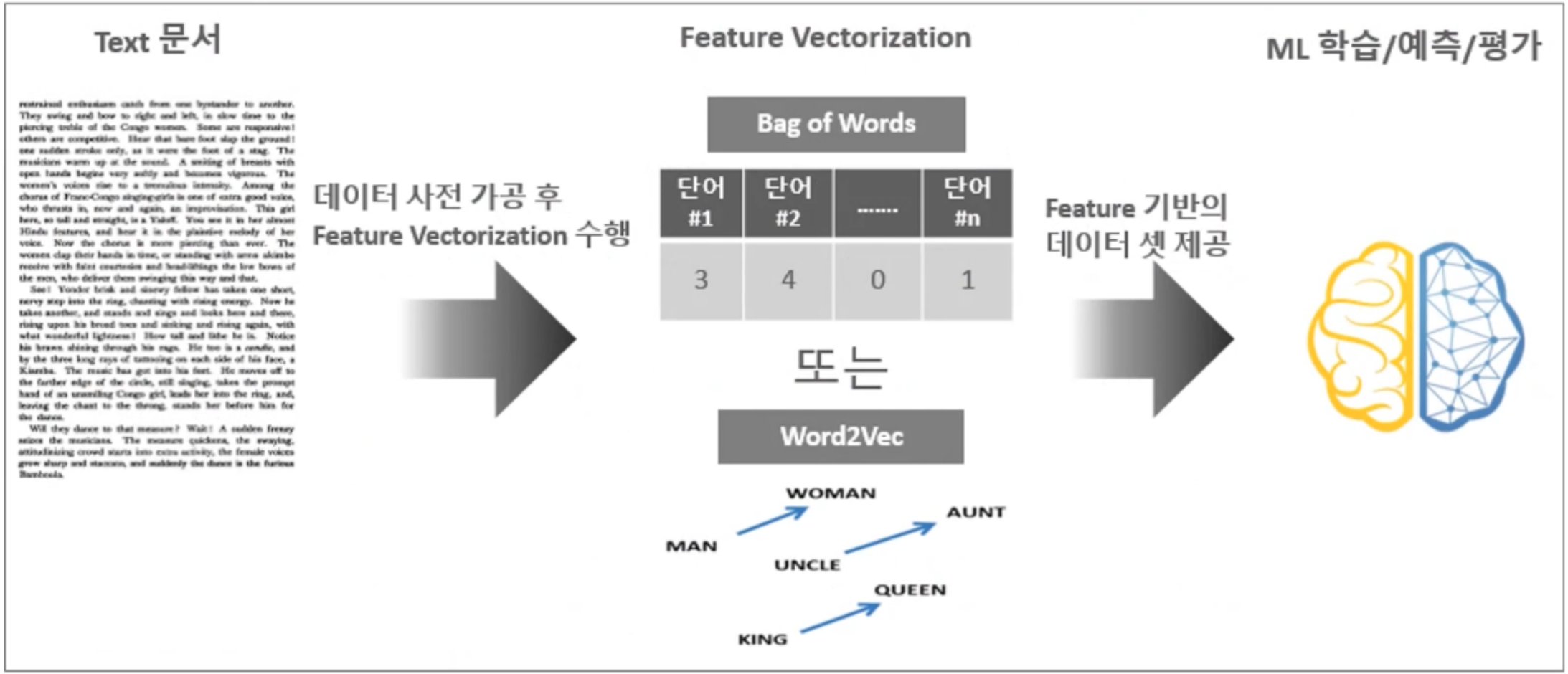
텍스트 요약 (Summarization)

텍스트 내에서 중요한 주제나 중심 사상을 추출하는 기법을 말합니다. 대표적으로 토픽 모델링(Topic Modeling)이 있습니다.

텍스트 군집화와 유사도 측정

비슷한 유형의 문서에 대해 군집화를 수행하는 기법을 말합니다. 텍스트 분류를 비지도학습으로 수행하는 방법의 일환으로 사용될 수 있습니다. 유사도 측정 역시 문서들간의 유사도를 측정해 비슷한 문서끼리 모을 수 있는 방법입니다.

텍스트 분석 프로세스


Text 문서
Feature Vectorization
ML 학습/예측/평가
데이터 사전 가공 후
Feature Vectorization 수행
Bag of Words
단어 #1
단어 #2
.....
단어 #n
3
4
0
1
또는
Word2Vec
WOMAN
AUNT
MAN
UNCLE
QUEEN
KING
Feature 기반의
데이터 셋 제공

파이썬 기반의 NLP, 텍스트 분석 패키지

- **NLTK(National Language Toolkit for Python)**: 파이썬의 가장 대표적인 NLP 패키지입니다. 방대한 데이터 세트와 서브 모듈을 가지고 있으며 NLP의 거의 모든 영역을 커버하고 있습니다. 많은 NLP 패키지가 NLTK의 영향을 받아 작성되고 있습니다. 수행 속도 측면에서 아쉬운 부분이 있어서 실제 대량의 데이터 기반에서는 제대로 활용되지 못하고 있습니다.
- **Gensim**: 토픽 모델링 분야에서 가장 두각을 나타내는 패키지입니다. 오래전부터 토픽 모델링을 쉽게 구현할 수 있는 기능을 제공해 왔으며, Word2Vec 구현 등의 다양한 신기능도 제공합니다. SpaCy와 함께 가장 많이 사용되는 NLP 패키지입니다.
- **SpaCy**: 뛰어난 수행 성능으로 최근 가장 주목을 받는 NLP 패키지입니다. 많은 NLP 애플리케이션에서 SpaCy를 사용하는 사례가 늘고 있습니다.

텍스트 전처리(텍스트 정규화)

클렌징 (Cleansing)	텍스트에서 분석에 오히려 방해가 되는 불필요한 문자, 기호 등을 사전에 제거하는 작업입니다. 예를 들어 HTML, XML 태그나 특정 기호 등을 사전에 제거합니다.
토큰화 (Tokenization)	문장 토큰화, 단어 토큰화, n-gram
필터링/스톱워드 제거/철자 수정	불필요한 단어나 분석에 큰 의미가 없는 단어(a, the, is, will등) 그리고 잘못된 철자 수정
Stemming/ Lemmatization	어근(단어 원형) 추출, Lemmatization이 Stemming보다 정교하고 의미론적 기반에서 단어 원형을 찾아줌

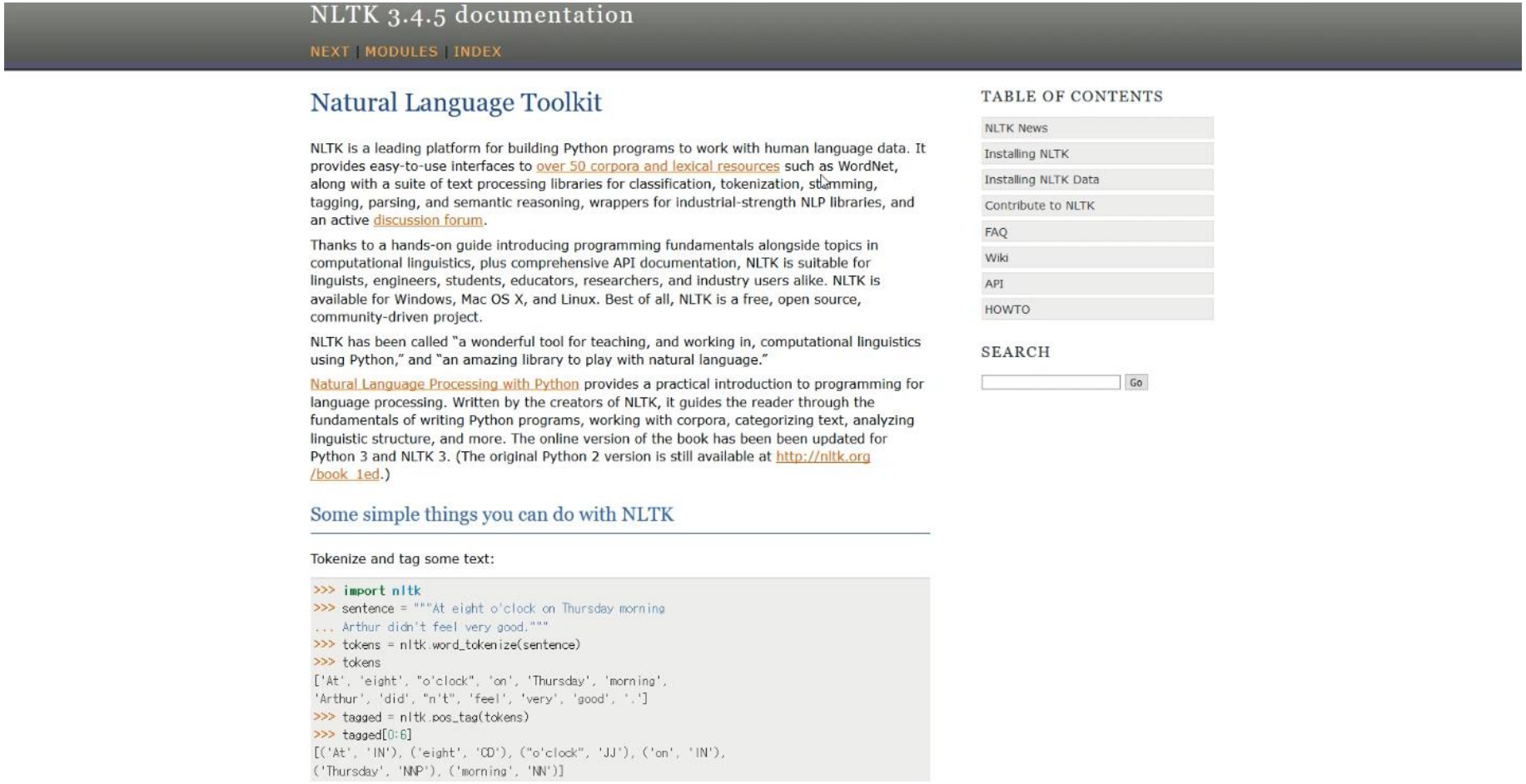
N-gram

- 문장을 개별 단어 별로 하나씩 토큰화 할 경우 문맥적인 의미는 무시될 수 밖에 없습니다. 이러한 문제를 조금이라도 해결해 보고자 도입 된 것이 n-gram 입니다.
- n-gram은 연속된 n개의 단어를 하나의 토큰화 단위로 분리해 내는 것입니다. n개 단어 크기 윈도우를 만들어 문장의 처음부터 오른쪽으로 움직이면서 토큰화를 수행합니다.
- 예를 들어 "Agent Smith knocks the door"를 2-gram(bigram)으로 만들면 (Agent, Smith), (Smith, knocks), (knocks, the), (the, door) 와 같이 연속적으로 2개의 단어들을 순차적으로 이동하면서 단어들을 토큰화 합니다.

실습 : 텍스트 데이터 전처리

nltk 패키지를 이용해서 텍스트 전처리를 해보자

- nltk 패키지 : 딥러닝 이전에 주로 텍스트 분석을 수행했던 패키지
  설치 : pip install nltk



NLTK 3.4.5 documentation

NEXT | MODULES | INDEX

Natural Language Toolkit

NLTK is a leading platform for building Python programs to work with human language data. It provides easy-to-use interfaces to over 50 corpora and lexical resources such as WordNet, along with a suite of text processing libraries for classification, tokenization, stemming, tagging, parsing, and semantic reasoning, wrappers for industrial-strength NLP libraries, and an active discussion forum.

Thanks to a hands-on guide introducing programming fundamentals alongside topics in computational linguistics, plus comprehensive API documentation, NLTK is suitable for linguists, engineers, students, educators, researchers, and industry users alike. NLTK is available for Windows, Mac OS X, and Linux. Best of all, NLTK is a free, open source, community-driven project.

NLTK has been called "a wonderful tool for teaching, and working in, computational linguistics using Python," and "an amazing library to play with natural language."

Natural Language Processing with Python provides a practical introduction to programming for language processing. Written by the creators of NLTK, it guides the reader through the fundamentals of writing Python programs, working with corpora, categorizing text, analyzing linguistic structure, and more. The online version of the book has been been updated for Python 3 and NLTK 3. (The original Python 2 version is still available at http://nltk.org/book_1ed.)

Some simple things you can do with NLTK

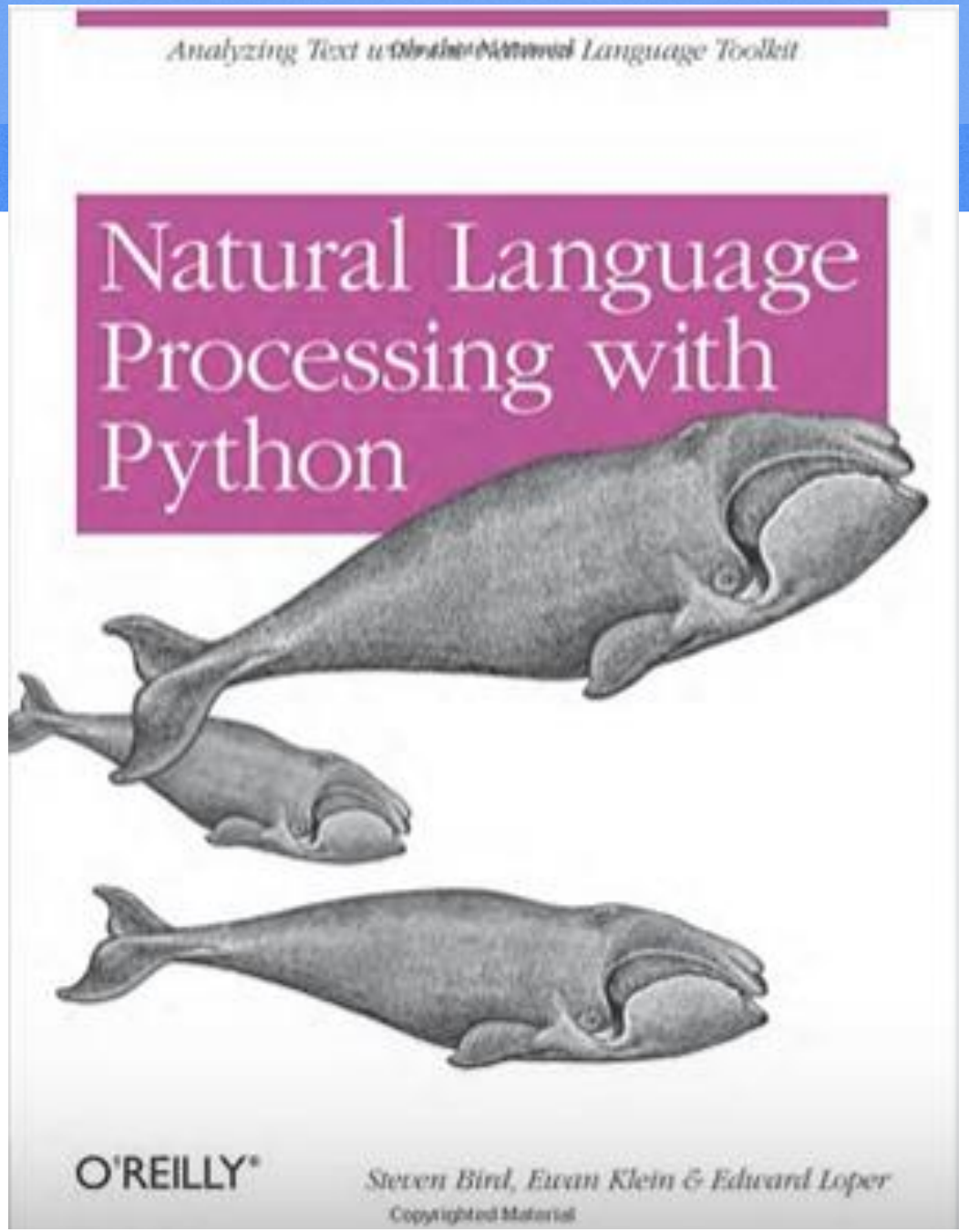
Tokenize and tag some text:

```
>>> import nltk
>>> sentence = """At eight o'clock on Thursday morning
... Arthur didn't feel very good."""
>>> tokens = nltk.word_tokenize(sentence)
>>> tokens
['At', 'eight', "o'clock", 'on', 'Thursday', 'morning',
'Arthur', 'did', "n't", 'feel', 'very', 'good', '.']
>>> tagged = nltk.pos_tag(tokens)
>>> tagged[0:6]
[('At', 'IN'), ('eight', 'CD'), ("o'clock", 'JJ'), ('on', 'IN'),
('Thursday', 'NNP'), ('morning', 'NN')]
```

TABLE OF CONTENTS

SEARCH

Go


Analyzing Text with the Natural Language Toolkit
Natural Language Processing with Python
O'REILLY
Steven Bird, Ewan Klein & Edward Loper
Copyrighted Material

실습 : 텍스트 데이터 전처리

토큰화

```
# 문장 토큰화 : sent_tokenize
from nltk import sent_tokenize

text_sample = 'The Matrix is everywhere its all around us, here even in this room. \
               You can see it out your window or on your television. \
               You feel it when you go to work, or go to church or pay your taxes.'

sentences = sent_tokenize(text=text_sample)

print(type(sentences), len(sentences))
print(sentences)
```

```
<class 'list'> 3
['The Matrix is everywhere its all around us, here even in this room.', 'You can see it out your window or on your te
levision.', 'You feel it when you go to work, or go to church or pay your taxes.']
```

```
# 단어 토큰화 : word_tokenize
from nltk import word_tokenize

sentence = "The Matrix is everywhere its all around us, here even in this room."

words = word_tokenize(sentence)
print(type(words), len(words))
print(words)
```

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

실습 : 텍스트 데이터 전처리

문장 별 단어 토큰화

```
# 여러 문장들에 대한 단어 토큰화
from nltk import word_tokenize, sent_tokenize

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

    # 문장별로 분리 토큰
    sentences = sent_tokenize(text)

    # 분리된 문장별 단어 토큰화
    word_tokens = [word_tokenize(sentence) for sentence in sentences]
    return word_tokens

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

```
<class 'list'> 3
[['The', 'Matrix', 'is', 'everywhere', 'its', 'all', 'around', 'us', ',', 'here', 'even', 'in', 'this', 'room', '.'],
['You', 'can', 'see', 'it', 'out', 'your', 'window', 'or', 'on', 'your', 'television', '.'], ['You', 'feel', 'it', 'w
hen', 'you', 'go', 'to', 'work', ',', 'or', 'go', 'to', 'church', 'or', 'pay', 'your', 'taxes', '.']]
```

stopwords 제거

영어 스탑워즈 확인해보기

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

```
[nltk_data] Downloading package stopwords to /Users/aiden/nltk_data...
[nltk_data]   Unzipping corpora/stopwords.zip.
```

```
True
```

```
# 영어 stopwords 확인
print('영어 stop words 갯수:',len(nltk.corpus.stopwords.words('english')))

# 영어 stopwords 중 10개만 출력해보자
print(nltk.corpus.stopwords.words('english')[:20])
```

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

```
# 위 예제 3개 문장에서 얻은 단어 토큰에 대해 stopwords 제거해보자
import nltk

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

print(all_tokens)
```

the, is 등 stopwords가 제거된 것 확인 가능

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

Stemming과 Lemmatization

Stemmer

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

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

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

Lemmatizer

```
from nltk.stem import WordNetLemmatizer

import nltk
# nltk.download('wordnet')

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

```
[nltk_data] Downloading package wordnet to /Users/aiden/nltk_data...
[nltk_data]   Unzipping corpora/wordnet.zip.
```

```
amuse amuse amuse
happy happy
fancy fancy
```

피처 벡터화 : Bag of Words – BOW

BOW

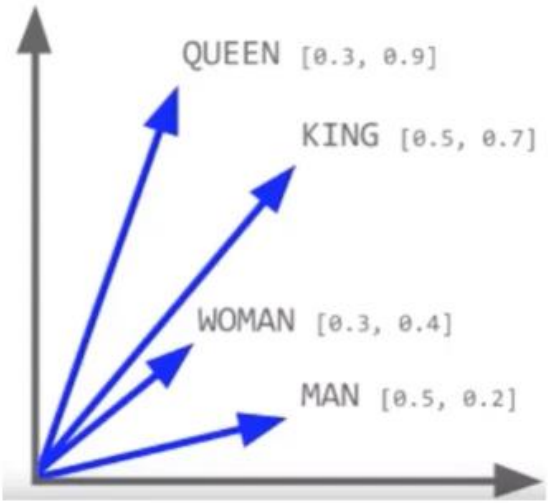
단어 feature 들

	단어 #1	단어 #2	단어 #3	.....	단어 #n
Doc #1 **History of Football**	3	4	3	0	1
.....	..	..	..	..	..
Doc #m **Premier League News**	0	5	0	4	3

각 문서에서 해당 단어의 횟수나 정규화 변환된 횟수

Document Term Matrix: 개별 문서(또는 문장)을 단어들의 횟수나 정규화 변환된 횟수로 표현

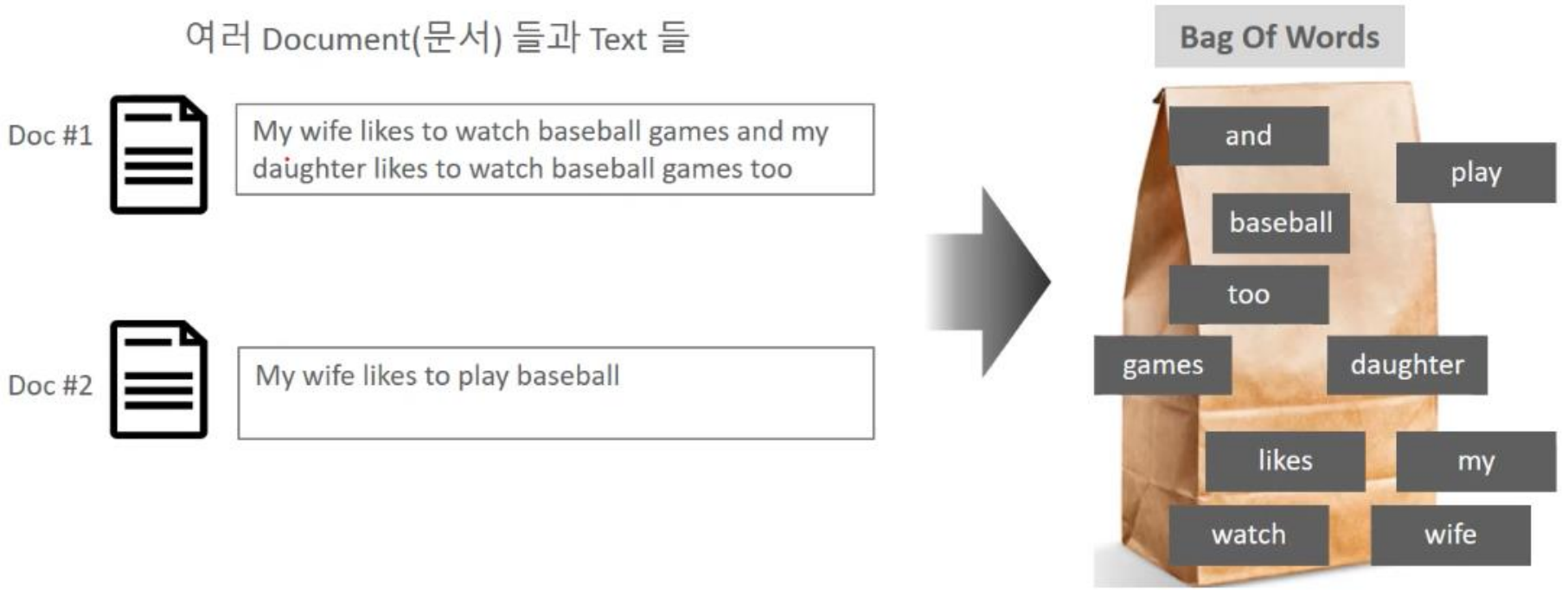
Word Embedding (Word2Vec)



개별 단어를 문맥을 가지는 N차원 공간에 벡터로 표현

BOW – Bag of Words

Bag of Words 모델은 문서가 가지는 모든 단어(Words)를 문맥이나 순서를 무시하고 일괄적으로 단어에 대해 빈도 값을 부여해 피처 값을 추출하는 모델입니다. 문서 내 모든 단어를 한꺼번에 봉투(Bag)안에 넣은 뒤에 흔들어서 섞는다는 의미로 Bag of Words(BOW) 모델이라고 합니다



BOW 구조

문장 1: 'My wife likes to watch baseball games and my daughter likes to watch baseball games too'
문장 2: 'My wife likes to play baseball'

1. 문장 1과 문장 2에 있는 모든 단어에서 중복을 제거하고 각 단어(feature 또는 term)를 컬럼 형태로 나열합니다. 그러고 나서 각 단어에 고유의 인덱스를 다음과 같이 부여합니다.

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

2. 문장에서 해당 단어가 나타나는 횟수(Occurrence)를 각 단어(단어 인덱스)에 기재합니다. 예를 들어 baseball은 문장 1, 2에서 총 2번 나타나며, daughter는 문장 1에서만 1번 나타납니다

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

문장 1에서 baseball 은 2회 나타남.

BOW 장단점

장점

- 쉽고 빠른 구축
- 예상보다 문서의 특징을 잘 나타내어 전통적으로 여러분야에서 활용도가 높음

VS

단점

- 문맥 의미(Semantic Context) 반영 문제 질의응답에는 x
- 희소 행렬 문제 메모리 문제

BOW 피처 벡터화

M 개의 Text 문서들

Doc #1

제목 : History of Football

The contemporary history of the world's favourite game spans more than 100 years.

.....

Doc #M

제목 : Premier League News

Premier League gets another chance to size up to La Liga on the pitch as England continues to dominate Spain off it

M x N 피처 벡터화

단어 feature 들

	단어 #1	단어 #2	단어 #3	.....	단어 #n
Doc #1 **History of Football**	3	4	3	0	1
.....	..	..	..	..	..
Doc #m **Premier League News**	0	5	0	4	3

각 문서에서 해당 단어의 횟수나 정규화 변환된 횟수

BOW 피처 벡터화 유형

단순 카운트 기반의 벡터화

단어 피처에 값을 부여할 때 각 문서에서 해당 단어가 나타나는 횟수, 즉 Count를 부여하는 경우를 카운트 벡터화라고 합니다. 카운트 벡터화에서는 카운트 값이 높을수록 중요한 단어로 인식됩니다.

증권 문서 : 주가, 시가, 주식, 기업 같은 단어는 너무 자주 등장하는데 큰 의미가 없다
서킷 브레이크 같은 단어는 자주 안 나오는 단어이므로 중요하다

TF-IDF 벡터화

카운트만 부여할 경우 그 문서의 특징을 나타내기보다는 언어의 특성상 문장에서 자주 사용될 수밖에 없는 단어까지 높은 값을 부여하게 됩니다. 이러한 문제를 보완하기 위해 TF-IDF(Term Frequency Inverse Document Frequency) 벡터화를 사용합니다.

TF-IDF는 개별 문서에서 자주 나타나는 단어에 높은 가중치를 주되, 모든 문서에서 전반적으로 자주 나타나는 단어에 대해서는 페널티를 주는 방식으로 값을 부여합니다.

TF-IDF

TF-IDF(Term Frequency Inverse Document Frequency)

- 특정 단어가 다른 문서에는 나타나지 않고 특정 문서에서만 자주 사용된다면 해당 단어는 해당 문서를 잘 특징짓는 중요 단어일 가능성이 높음
- 특정 단어가 매우 많은 여러 문서에서 빈번히 나타난다면 해당 단어는 개별 문서를 특징짓는 정보로서의 의미를 상실

TF(Term Frequency)

문서에서 해당 단어가 얼마나 나왔는지를 나타내는지표

DF(Document Frequency)

해당 단어가 몇 개의 문서에서 나타났는지를 나타내는 지표

IDF(Inverse Document Frequency)

DF의 역수로서 전체 문서수/DF

한 개의 문서(Document)

모든 문서들(Corpus)

Term Frequency

The	Matrix	is	nothing	but	an	advertising	gimmick
40	5	50	12	20	45	3	2

Document Frequency

The	Matrix	is	nothing	but	an	advertising	gimmick
2000	190	2300	500	1200	3000	52	12

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

TF_i = 개별 문서에서의 단어 i 빈도

DF_i = 단어 i를 가지고 있는 문서 개수

N = 전체 문서 개수

사이킷런 CountVectorizer 파라미터

파라미터 명	파라미터 설명
max_df	전체 문서에 걸쳐서 너무 높은 빈도수를 가지는 단어 피처를 제외하기 위한 파라미터입니다. 너무 높은 빈도수를 가지는 단어는 스톱 워드와 비슷한 문법적인 특성으로 반복적인 단어일 가능성이 높기에 이를 제거하기 위해 사용됩니다. max_df = 100과 같이 정수 값을 가지면 전체 문서에 걸쳐 100개 이하로 나타나는 단어만 피처로 추출합니다. max_df = 0.95와 같이 부동소수점 값(0.0 ~ 1.0)을 가지면 전체 문서에 걸쳐 빈도수 0~95%까지의 단어만 피처로 추출하고 나머지 상위 5%는 피처로 추출하지 않습니다.
min_df	전체 문서에 걸쳐서 너무 낮은 빈도수를 가지는 단어 피처를 제외하기 위한 파라미터입니다. 수백~ 수천 개의 전체 문서에서 특정 단어가 min_df 에 설정된 값보다 적은 빈도수를 가진다면 이 단어는 크게 중요하지 않거나 가비지(garbage)성 단어일 확률이 높습니다. min_df = 2와 같이 정수 값을 가지면 전체 문서에 걸쳐서 2번 이하로 나타나는 단어는 피처로 추출하지 않습니다. min_df = 0.02와 같이 부동소수점 값(0.0 ~ 1.0)을 가지면 전체 문서에 걸쳐서 하위 2% 이하의 빈도수를 가지는 단어는 피처로 추출하지 않습니다.
max_features	피처로 추출하는 피처의 개수를 제한하며 정수로 값을 지정합니다. 가령 max_features = 2000으로 지정할 경우 가장 높은 빈도를 가지는 단어 순으로 정렬해 2000개까지만 피처로 추출합니다.
stop_words	'english'로 지정하면 영어의 스톱 워드로 지정된 단어는 추출에서 제외합니다.

사이킷런 CountVectorizer 파라미터

파라미터 명	파라미터 설명
ngram_range	Bag of Words 모델의 단어 순서를 어느 정도 보강하기 위한 n-gram 범위를 설정합니다. 튜플 형태로 (범위 최솟값, 범위 최댓값)을 지정합니다. 예를 들어 (1, 1)로 지정하면 토큰화된 단어를 1개씩 피처로 추출합니다. (1, 2)로 지정하면 토큰화된 단어를 1개씩(minimum 1), 그리고 순서대로 2개씩(maximum 2) 묶어서 피처로 추출합니다.
analyzer	피처 추출을 수행한 단위를 지정합니다. 당연히 디폴트는 'word' 입니다. Word가 아니라 character의 특정 범위를 피처로 만드는 특정한 경우 등을 적용할 때 사용됩니다.
token_pattern	토큰화를 수행하는 정규 표현식 패턴을 지정합니다. 디폴트 값은 '\b\w\w+\b로, 공백 또는 개행 문자 등으로 구분된 단어 분리자(\b) 사이의 2문자(문자 또는 숫자, 즉 영숫자) 이상의 단어(word)를 토큰으로 분리합니다. analyzer= 'word'로 설정했을 때만 변경 가능하나 디폴트 값을 변경할 경우는 거의 발생하지 않습니다. 어근 추출시 외부 함수를 사용할 경우 해당 외부 함수를 token_pattern의 인자로 사용합니다.
lower_case	모든 문자를 소문자로 변경할 것인지를 설정. 기본은 True입니다.

CountVectorizer로 피처 벡터화

CountVectorizer 를 이용한 피처 벡터화

- 사전 데이터 가공

모든 문자를 소문자로 변환하는 등의 사전 작업 수행.
(Default 로 **lowercase = True** 임)

↓

토큰화

Default는 단어 기준(analyzer = True) 이며 n_gram_range를 반영하여 토큰화 수행

↓

텍스트 정규화

Stop Words 필터링만 수행
Stemmer , Lemmatize 는 CountVectorizer 자체에서는 지원되지 않음. 이를 위한 함수를 만들거나 외부 패키지로 미리 Text Normalization 수행 필요

↓

피처 벡터화

max_df , min_df , max_features 등의 파라미터를 반영하여 Token된 단어들을 feature extraction 후 vectorization 적용.

희소 행렬

수십만개의 컬럼

수천 ~ 수만개 레코드

	단어 1	단어 2	단어 3	...	단어 1000	...	단어 2000	...	단어 10000	....	단어 20000	.....	단어 100000
문서1	1	2	2	0	0	0	0	0	0	0	0	0	0
문서2	0	0	1	0	0	1	0	0	1	0	0	0	1
문서 ...	....	....	....	....	....	....	....	....	....	....	....	....	....
문서 10000	0	1	3	0	0	0	0	0	0	0	0	0	0

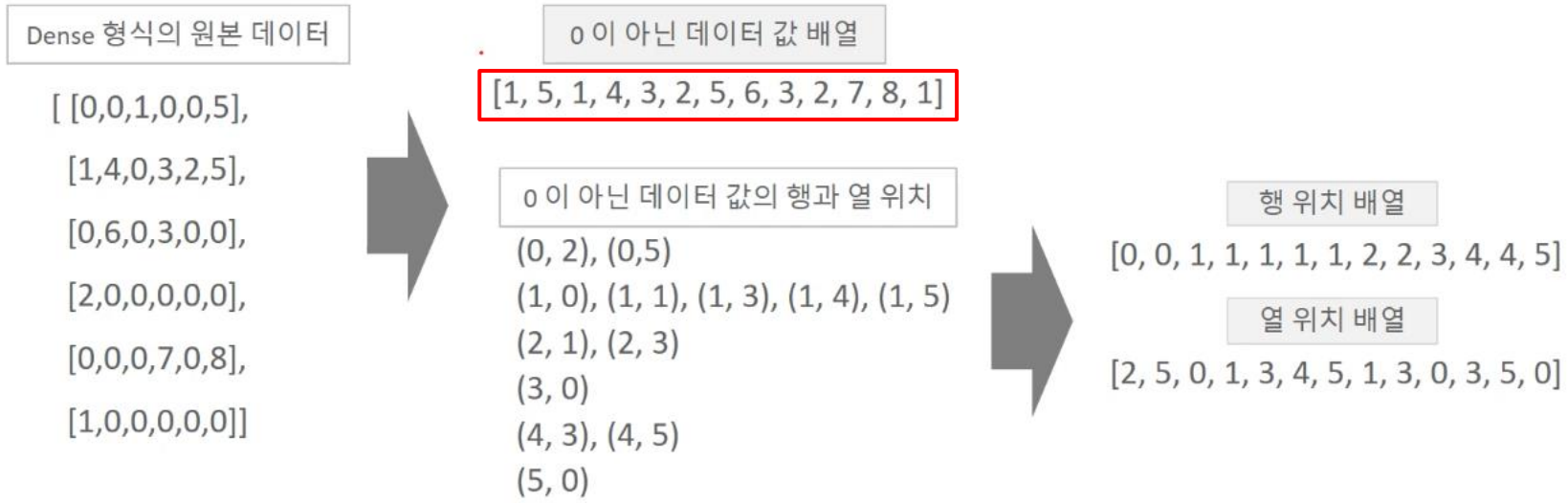
BOW 의 Vectorization 모델은 너무 많은 0 값이 메모리 공간에 할당되어 많은 메모리 공간이 필요하며 또한 연산 시에도 데이터 액세스를 위한 많은 시간이 소모 됩니다.

희소 행렬의 저장 변환 형식

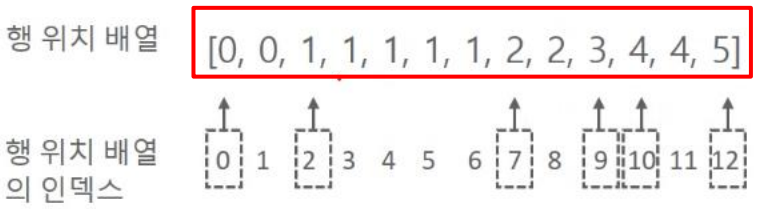
- COO 형식: Coordinate(좌표) 방식을 의미하며 0이 아닌 데이터만 별도의 배열(Array)에 저장하고 그 데이터를 가리키는 행과 열의 위치를 별도의 배열로 저장하는 방식
- CSR 형식: COO 형식이 위치 배열값을 중복적으로 가지는 문제를 해결한 방식. 일반적으로 CSR형식이 COO보다 많이 사용됨.

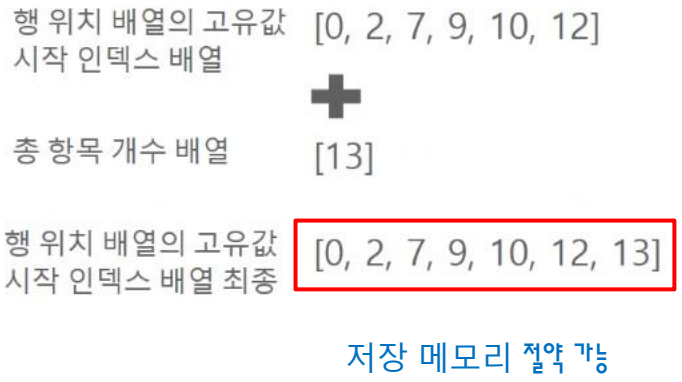
파이썬에서는 희소 행렬을 COO, CSR 형식으로 변환하기 위해서 Scipy의 coo_matrix(), csr_matrix() 함수를 이용합니다.

COO형식


Dense 형식의 원본 데이터
[[0,0,1,0,0,5],
[1,4,0,3,2,5],
[0,6,0,3,0,0],
[2,0,0,0,0,0],
[0,0,0,7,0,8],
[1,0,0,0,0,0]]
0 이 아닌 데이터 값 배열
[1, 5, 1, 4, 3, 2, 5, 6, 3, 2, 7, 8, 1]
0 이 아닌 데이터 값의 행과 열 위치
(0, 2), (0,5)
(1, 0), (1, 1), (1, 3), (1, 4), (1, 5)
(2, 1), (2, 3)
(3, 0)
(4, 3), (4, 5)
(5, 0)
행 위치 배열
[0, 0, 1, 1, 1, 1, 1, 2, 2, 3, 4, 4, 5]
열 위치 배열
[2, 5, 0, 1, 3, 4, 5, 1, 3, 0, 3, 5, 0]

CSR 형식





저장 메모리 절약 가능

COO

```
import numpy as np

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

```
from scipy import sparse

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

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

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

```
sparse_coo.toarray()
```

```
array([[3, 0, 1],
       [0, 2, 0]])
```

CSR

```
from scipy import sparse

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

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

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

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

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

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

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

```
COO 변환된 데이터가 제대로 되었는지 다시 Dense로 출력 확인
[[0 0 1 0 0 5]
 [1 4 0 3 2 5]
 [0 6 0 3 0 0]
 [2 0 0 0 0 0]
 [0 0 0 7 0 8]
 [1 0 0 0 0 0]]
CSR 변환된 데이터가 제대로 되었는지 다시 Dense로 출력 확인
[[0 0 1 0 0 5]
 [1 4 0 3 2 5]
 [0 6 0 3 0 0]
 [2 0 0 0 0 0]
 [0 0 0 7 0 8]
 [1 0 0 0 0 0]]
```

간단하게 표현

```
dense3 = np.array([[0,0,1,0,0,5],
                   [1,4,0,3,2,5],
                   [0,6,0,3,0,0],
                   [2,0,0,0,0,0],
                   [0,0,0,7,0,8],
                   [1,0,0,0,0,0]])

coo = sparse.coo_matrix(dense3)
csr = sparse.csr_matrix(dense3)
```

실습 :
20 Newsgroup 분류

20 Newsgroup 분류하기 문제

18846 개의 뉴스 문서를 20개의 뉴스 그룹 카테고리로 분류

comp.graphics comp.os.ms-windows.misc comp.sys.ibm.pc.hardware comp.sys.mac.hardware comp.windows.x	rec.autos rec.motorcycles rec.sport.baseball rec.sport.hockey	sci.crypt sci.electronics sci.med sci.space
misc.forsale	talk.politics.misc talk.politics.guns talk.politics.mideast	talk.religion.misc alt.atheism soc.religion.christian

텍스트 정규화 → 피처 벡터화 → 머신러닝 학습/예측/평가 → Pipeline 적용 → GridSearchCV 최적화

데이터 확인

```
: from sklearn.datasets import fetch_20newsgroups

news_data = fetch_20newsgroups(subset='all',random_state=156)
news_data
```

```
Downloading 20news dataset. This may take a few minutes.
Downloading dataset from https://ndownloader.figshare.com/files/5975967 (14 MB)
```

```
: {'data': ['From: egreen@east.sun.com (Ed Green - Pixel Cruncher)\nSubject: Re: Observation re: helmets\nOrganizati
on: Sun Microsystems, RTP, NC\nLines: 21\nDistribution: world\nReply-To: egreen@east.sun.com\nNNTP-Posting-Host: l
aser.east.sun.com\n\nIn article 211353@mavenry.altcit.eskimo.com, maven@mavenry.altcit.eskimo.com (Norman Hamer) w
rites:\n> \n> The question for the day is re: passenger helmets, if you don\'t know for \n>certain who\'s gonna ri
de with you (like say you meet them at a .... church \n>meeting, yeah, that\'s the ticket)... What are some guidel
ines? Should I just \n>pick up another shoei in my size to have a backup helmet (XL), or should I \n>maybe get an
inexpensive one of a smaller size to accomodate my likely \n>passenger? \n\nIf your primary concern is protecting
the passenger in the event of a\ncrash, have him or her fitted for a helmet that is their size.  If your\nprimary
concern is complying with stupid helmet laws, carry a real big\nspare (you can put a big or small head in a big he
lmet, but not in a\nsmall one).\n\n---\nEd Green, former Ninjaite |I was drinking last night with a biker,\n  Ed.G
reen@East.Sun.COM    |and I showed him a picture of you.  I said,\nDoD #0111  (919)460-8302  |"Go on, get to know h
er, you\'ll like her!"\n (The Grateful Dead) -->  |It seemed like the least I could do...\n\n',
  "From: jlevine@rd.hydro.on.ca (Jody Levine)\nSubject: Re: insect impacts\nOrganization: Ontario Hydro - Research
Division\nLines: 64\n\nI feel childish.\n\nIn article <1ppvds$92a@seven-up.East.Sun.COM> egreen@East.Sun.COM write
```

```
print(news_data.keys())
```

```
dict_keys(['data', 'filenames', 'target_names', 'target', 'DESCR'])
```

18846개 뉴스문서	파일 이름	타겟의 뉴스그룹명	타겟 값: 0~19

타겟값 별 분포도 확인

```
import pandas as pd

print('target 클래스의 값과 분포도 \n',pd.Series(news_data.target).value_counts().sort_index())
print('target 클래스의 이름들 \n',news_data.target_names)
len(news_data.target_names), pd.Series(news_data.target).shape
```

```
target 클래스의 값과 분포도
 0     799
1     973
2     985
3     982
4     963
5     988
6     975
7     990
8     996
9     994
10    999
11    991
12    984
13    990
14    987
15    997
16    910
17    940
18    775
19    628
dtype: int64
target 클래스의 이름들
 ['alt.atheism', 'comp.graphics', 'comp.os.ms-windows.misc', 'comp.sys.ibm.pc.hardware', 'comp.sys.mac.hardware', 'co
mp.windows.x', 'misc.forsale', 'rec.autos', 'rec.motorcycles', 'rec.sport.baseball', 'rec.sport.hockey', 'sci.crypt',
'sci.electronics', 'sci.med', 'sci.space', 'soc.religion.christian', 'talk.politics.guns', 'talk.politics.mideast',
'talk.politics.misc', 'talk.religion.misc']

(20, (18846,))
```

학습/테스트용 데이터 생성

학습과 테스트용 데이터 생성

```
from sklearn.datasets import fetch_20newsgroups

# subset='train'으로 학습용(Train) 데이터만 추출, remove=('headers', 'footers', 'quotes')로 내용만 추출
train_news= fetch_20newsgroups(subset='train', remove=('headers', 'footers', 'quotes'), random_state=156)
X_train = train_news.data
y_train = train_news.target
print(type(X_train))

# subset='test'으로 테스트(Test) 데이터만 추출, remove=('headers', 'footers', 'quotes')로 내용만 추출
test_news= fetch_20newsgroups(subset='test',remove=('headers', 'footers','quotes'),random_state=156)
X_test = test_news.data
y_test = test_news.target
print('학습 데이터 크기 {0} , 테스트 데이터 크기 {1}'.format(len(train_news.data) , len(test_news.data)))
```

```
<class 'list'>
학습 데이터 크기 11314 , 테스트 데이터 크기 7532
```

피처 벡터화 변환과 머신러닝 학습/예측

CountVectorizer로 피처 벡터화 변환과 머신러닝 모델 학습/예측/평가

```
from sklearn.feature_extraction.text import CountVectorizer

# Count Vectorization으로 feature extraction 변환 수행.
cnt_vect = CountVectorizer()
cnt_vect.fit(X_train , y_train)
X_train_cnt_vect = cnt_vect.transform(X_train)

# 학습 데이터로 fit( )된 CountVectorizer를 이용하여 테스트 데이터를 feature extraction 변환 수행.
X_test_cnt_vect = cnt_vect.transform(X_test)

print('학습 데이터 Text의 CountVectorizer Shape:',X_train_cnt_vect.shape)
print('테스트 데이터 Text의 CountVectorizer Shape:',X_test_cnt_vect.shape)
```

```
학습 데이터 Text의 CountVectorizer Shape: (11314, 101631)
테스트 데이터 Text의 CountVectorizer Shape: (7532, 101631)
```

```
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score

# LogisticRegression을 이용하여 학습/예측/평가 수행. 로지스틱 회귀는 희소행렬에 잘 동작한다.
lr_clf = LogisticRegression()
lr_clf.fit(X_train_cnt_vect , y_train)
pred = lr_clf.predict(X_test_cnt_vect)
print('CountVectorized Logistic Regression 의 예측 정확도는 {0:.3f}'.format(accuracy_score(y_test,pred)))
```

```
CountVectorized Logistic Regression 의 예측 정확도는 0.608
```

테스트 데이터의 뉴스 카테고리 예측 성공률이 60%정도이다

TF-IDF, stopwords 제거를 적용한 학습/예측

TF-IDF 벡터화 적용해서 학습/예측/평가 수행

```
# TF-IDF Vectorization 적용하여 학습 데이터셋과 테스트 데이터 셋 변환.
from sklearn.feature_extraction.text import TfidfVectorizer

tfidf_vect = TfidfVectorizer()
tfidf_vect.fit(X_train)
X_train_tfidf_vect = tfidf_vect.transform(X_train)
X_test_tfidf_vect = tfidf_vect.transform(X_test)

# LogisticRegression을 이용하여 학습/예측/평가 수행.
lr_clf = LogisticRegression()
lr_clf.fit(X_train_tfidf_vect , y_train)
pred = lr_clf.predict(X_test_tfidf_vect)
print('TF-IDF Logistic Regression 의 예측 정확도는 {0:.3f}'.format(accuracy_score(y_test ,pred)))
```

```
TF-IDF Logistic Regression 의 예측 정확도는 0.674
```

정확도 상승

```
# 실행시간 5분
# stop words 필터링을 추가하고 ngram을 기본(1,1)에서 (1,2)로 변경하여 Feature Vectorization 적용.
tfidf_vect = TfidfVectorizer(stop_words='english', ngram_range=(1,2), max_df=300 ) # bigram
tfidf_vect.fit(X_train)
X_train_tfidf_vect = tfidf_vect.transform(X_train)
X_test_tfidf_vect = tfidf_vect.transform(X_test)

# 로지스틱 회귀 적용
lr_clf = LogisticRegression()
lr_clf.fit(X_train_tfidf_vect , y_train)
pred = lr_clf.predict(X_test_tfidf_vect)
print('TF-IDF Vectorized Logistic Regression 의 예측 정확도는 {0:.3f}'.format(accuracy_score(y_test ,pred)))
```

```
TF-IDF Vectorized Logistic Regression 의 예측 정확도는 0.692
```

정확도 상승

GridSearchCV

```
from sklearn.model_selection import GridSearchCV

# 최적 C 값 도출 튜닝 수행. CV는 3 Fold셋으로 설정.
params = { 'C':[0.01, 0.1, 1, 5, 10]}
grid_cv_lr = GridSearchCV(lr_clf ,param_grid=params , cv=3 , scoring='accuracy' , verbose=1 )
grid_cv_lr.fit(X_train_tfidf_vect , y_train)
print('Logistic Regression best C parameter :',grid_cv_lr.best_params_ )

# 최적 C 값으로 학습된 grid_cv로 예측 수행하고 정확도 평가.
pred = grid_cv_lr.predict(X_test_tfidf_vect)
print('TF-IDF Vectorized Logistic Regression 의 예측 정확도는 {0:.3f}'.format(accuracy_score(y_test ,pred)))
```

```
Fitting 3 folds for each of 5 candidates, totalling 15 fits

Logistic Regression best C parameter : {'C': 10}
TF-IDF Vectorized Logistic Regression 의 예측 정확도는 0.704
```

정확도 상승

Thank You