AIchemist Session

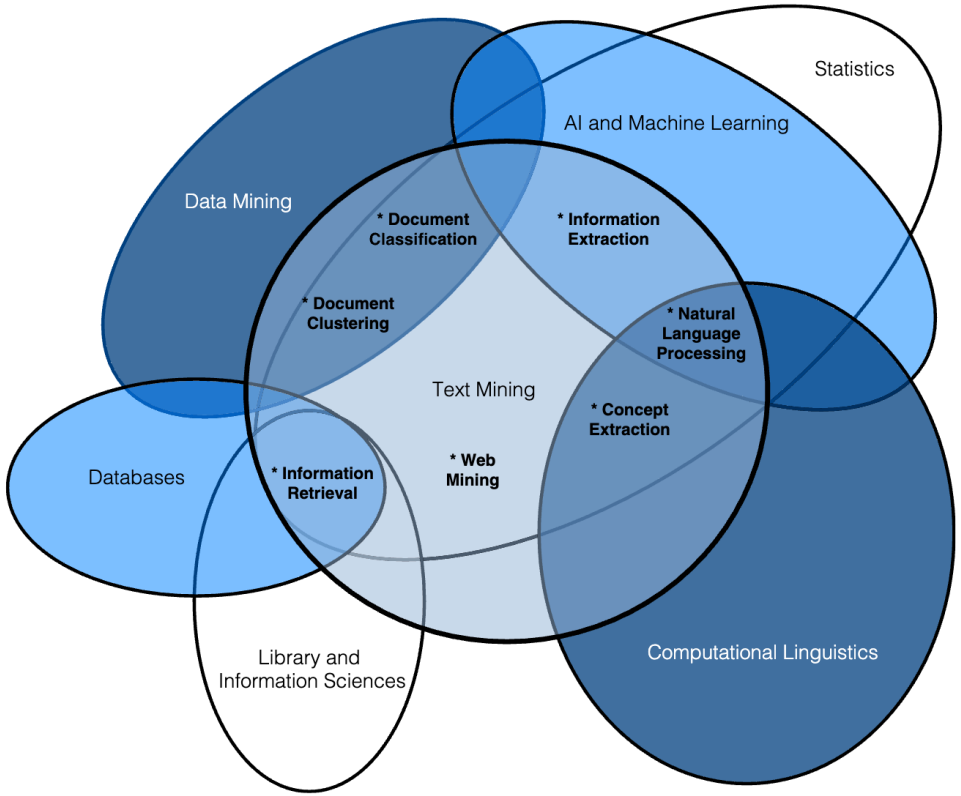
CHAP 08 텍스트 분석

CONTENTS

목차

서론

텍스트 분석(마이닝) vs NLP

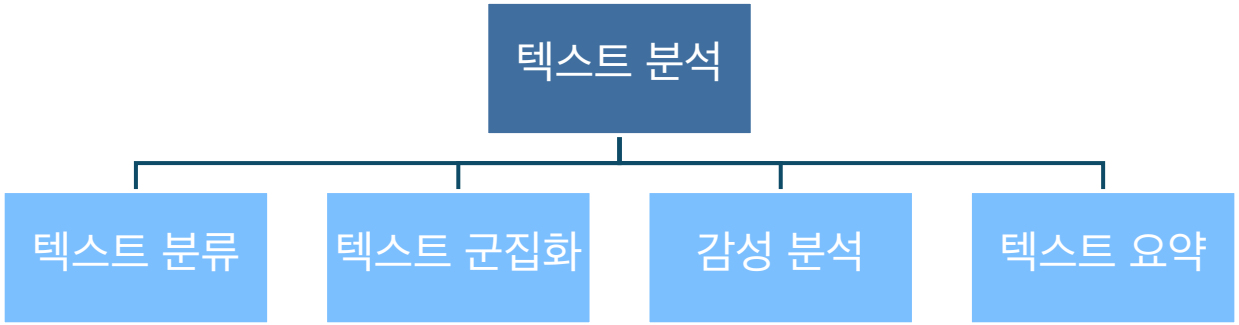


- 자연어처리(NLP)

머신이 인간의 언어를 이해하고 해석하는데 더 중점을 두고 기술을 발전하기 위해 통계학과 다양한 딥러닝 기술들을 적용하여 연구하는 분야

- 텍스트마이닝(Text-mining)

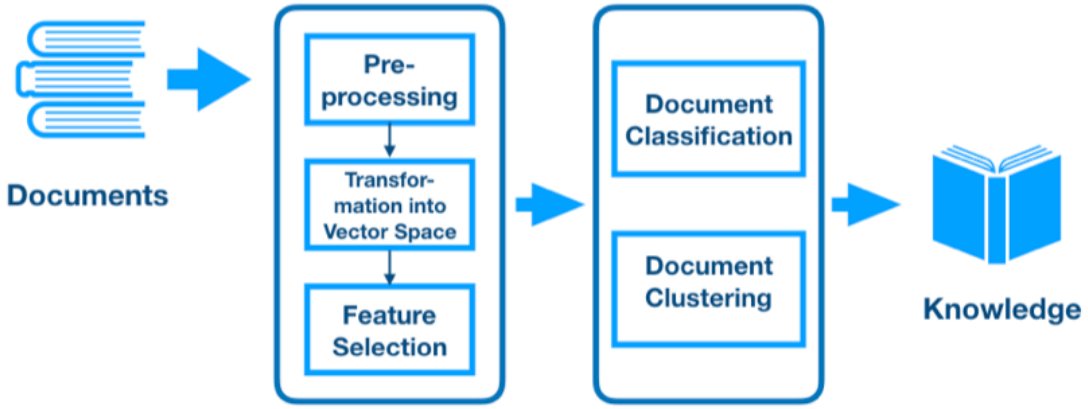
자연어처리(NLP)의 결과물인 언어모델(Language Model)을 활용하여 배달앱 리뷰나 쇼핑몰 검색 데이터와 같은 텍스트 형태로된 비정형 텍스트에서 고객의 경향성이나 선호도 등 유의미한 정보를 얻어내기 위한 분석 기법



01. 텍스트 분석 이해

텍스트 분석 이해

머신러닝 알고리즘은 숫자형의 피처 기반 데이터만 입력받을 수 있음. 비정형 데이터에서 피처 형태로 추출하고 추출된 피처에 숫자값을 부여할 수 있어야 됨



1) **텍스트 사전 준비 작업 (텍스트 전처리)** : 클렌징, 대/소문자 변경, 특수분자 삭제 등의 클렌징 작업, 단어 등의 토큰화 작업, stop word 제거 작업, 어근 추출 등의 텍스트 정규화 작업 수행

2) **피처 벡터화/추출** : 가공된 텍스트에 피처를 추출하고 벡터값을 할당 (BOW, Word2Vec)

3) **ML 모델 수립 및 학습/예측/평가** : 피처 벡터화 된 데이터 세트에 ML 모델을 적용

텍스트 분석 이해

텍스트 분석 패키지

NLTK	파이썬의 가장 대표적인 NLP 패키지. 많은 NLP 패키지가 NTLK의 영향을 받아 작성되고 있음. 수행 속도 측면에서 아쉬운 부분이 있음.
Genism	토픽 모델링 분야에서 가장 두각을 나타내는 패키지.
SpaCy	뛰어난 수행 성능으로 최근 가장 주목을 받는 NLP 패키지. 많은 NLP 어플리케이션에서 SpaCy를 사용하는 사례 증가

02. 텍스트 전처리

텍스트 정규화

텍스트 전처리

사전에 텍스트를 가공하는 준비 작업 필요

- 클렌징
- 토큰화
- 필터링/스톱워드 제거/철자 수정
- Stemming
- Lemmatization

클렌징

텍스트 분석에 방해가 되는 불필요한 문자, 기호 등을 사전에 제거

1. HTML 태그 : `<br>`, `<p>`, `<div>`
2. URL
3. 이메일 주소
4. 특수 문자: 텍스트 분석에는 일반적으로 특수 문자(`@`, `#`, `$`, `%`, `^`, `&`, `*`, `(`, `)`, `-`, `_`, `=`, `+`, `[`, `]`, `{`, `}`, `;`, `:`, `,`, `.`, `<`, `>`, `/`, `?`, `~`, `|`, `\` 등)가 필요 없습니다.
5. 숫자: 날짜나 시간 정보는 분석에 방해가 될 수 있습니다.
6. 이모티콘, 이모지
7. 불필요한 공백
8. 언어에 따라 필요 없는 문자: 예를 들어, 한국어 텍스트 분석에는 한자나 외국어 문자가 필요 없을 수 있습니다.
9. 문장 부호: 콤마, 마침표, 물음표, 느낌표 등의 문장 부호

텍스트 전처리

테스트 토큰화

토큰

텍스트 토큰화에서 '토큰'은 일반적으로 문장을 구성하는 최소 단위를 의미. (단어, 문장, 문단 등)
텍스트 토큰화는 주어진 텍스트를 이러한 토큰으로 분리하는 과정 (예를 들어, 문장을 단어 단위로 분리하거나, 문단을 문장 단위로 분리하는 것)

1) 문장 토큰화

텍스트 문서를 문장으로 각각 분리

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

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

```
<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 when you go to work, or go to church or pay your taxes.']
```

텍스트 전처리

2) 단어 토큰화

문장을 단어로 토큰화

- 공백, 마침표, 개행문자 로 분리
- 정규 표현식을 이용해 분리
- 단어의 순서가 중요하지 않은 경우 문장 토큰화를 사용하지 않고 단어 토큰화만 사용해도 충분
- 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',
'.']
```

텍스트 전처리

3) 문장 토큰화 + 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', 'when', 'you', 'go', 'to', 'work', ',', 'or', 'go', 'to', 'church', 'or', 'pay', 'your', 'taxes', '.']]
```

*n-gram
문장을 단어별로 토큰화 할 경우 문맥적인 의미가 무시된다는 한계가 존재.

'N-gram'은 텍스트 토큰화에서 사용되는 기법으로, **연속된 'N'개의 토큰으로 구성된 시퀀스를 의미**

예를 들어, '2-gram' 또는 'bigram'은 연속된 두 개의 토큰(보통 단어)으로 이루어진 시퀀스를 나타냄. 이를 통해 단어나 문장의 문맥을 보다 잘 이해하고, 예측하거나 분석하는 데 도움을 줌.

텍스트 전처리

스톱워드 제거 분석에 큰 의미가 없는 단어를 지칭 (예. Is, the, a, will)

NTLK에 언어별로 목록화된 스톱워드 제공
Stop words를 필터링으로 제거 예제

```
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)
```

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

텍스트 전처리

Stemming과 Lemmatization 문법적 또는 의미적으로 변화하는 단어의 원형을 찾는 과정을 의미합니다.

1. Stemming

- "working"의 어간: "work"
- "worked"의 어간: "work"
- "works"의 어간: "work"

Stemming에서는 단순히 어미를 제거하기 때문에, "working", "worked", "works" 모두 "work"라는 어간을 가집니다.

2. Lemmatization:

- "am", "are", "is"의 표제어: "be"
- "car", "cars"의 표제어: "car"
- "better"의 표제어: "good"

Lemmatization에서는 단어의 의미와 문법적 특성을 고려하여 표제어를 찾기 때문에, "am", "are", "is"는 모두 "be"라는 표제어를, "car", "cars"는 "car"라는 표제어를, "better"는 "good"이라는 표제어를 가집니다.

Stemming	Lemmatization
• 원형 단어로 변환 시 일반적인 방법을 적용하거나 더 단순화된 방법을 적용 • 원래 단어에서 일부 철자가 훼손된 어근 단어 추출	• 문법적인 요소와 의미적인 부분을 감안해 정확한 철자로 된 어근 단어 찾음 • Stemming보다 더 오랜 시간 소요

텍스트 전처리

Stemming 예제

```
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
```

Lemmatization 예제

```
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'))
```

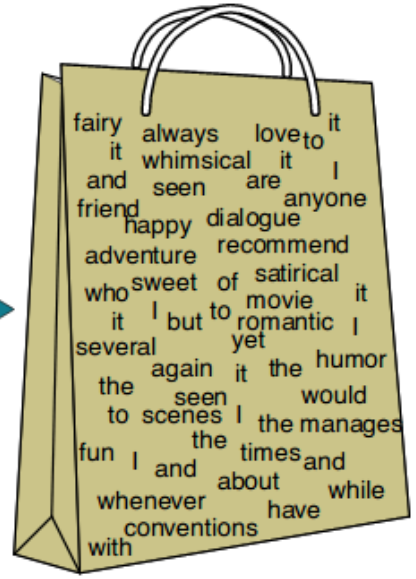
```
[nltk_data] Downloading package wordnet to
[nltk_data]     C:\Users\yong\AppData\Roaming\nltk_data...
[nltk_data]   Package wordnet is already up-to-date!
amuse amuse amuse
happy happy
fancy fancy
```

03. Bag of Words - BOW

Bag of Words - BOW

모든 단어를 문맥이나 순서를 무시하고 일괄적으로 단어에 대해 빈도값 을 부여해 피처 값을 추출하는 모델

I love this movie! It's sweet, but with satirical humor. The dialogue is great and the adventure scenes are fun... It manages to be whimsical and romantic while laughing at the conventions of the fairy tale genre. I would recommend it to just about anyone. I've seen it several times, and I'm always happy to see it again whenever I have a friend who hasn't seen it yet!



it	6
I	5
the	4
to	3
and	3
seen	2
yet	1
would	1
whimsical	1
times	1
sweet	1
satirical	1
adventure	1
genre	1
fairy	1
humor	1
have	1
great	1
…	…

장점	단점
쉽고 빠른 구축 문서의 특징을 잘 나타낼 수 있는 모델	문맥 의미 반영 부족 희소 행렬 문제

*희소 행렬 문제

많은 문서에서 단어를 추출하면 매우 많은 단어가 칼럼으로 만들어짐. 대부분의 데이터는 0값으로 채워지게됨. 대규모의 칼럼으로 구성된 행렬에서 대부분의 값이 0으로 채워지는 행렬을 희소 행렬이라고 함.

Bag of Words - BOW

Bow 피처벡터화

BOW 모델

단어 빈도 값을 피처에 값으로 부여해 각 문서를 단어 피처의 발생 빈도 값으로 구성된 벡터로 만드는 기법

Document D1	*The child makes the dog happy* the: 2, dog: 1, makes: 1, child: 1, happy: 1
Document D2	*The dog makes the child happy* the: 2, child: 1, makes: 1, dog: 1, happy: 1

↓

	child	dog	happy	makes	the	BoW Vector representations
D1	1	1	1	1	2	**[1,1,1,1,2]**
D2	1	1	1	1	2	**[1,1,1,1,2]**

피처 벡터화 방식

- 카운트 기반 벡터화
- TF-IDF 기반 벡터화

카운트 기반 벡터화에서는 중요하지 않지만 자주 사용되는 단어에도 높은 중요도 값을 부여하는 문제 발생
이 문제를 보완하기 위해 TF-IDF 출현

Bag of Words - BOW

TF-IDF

개별 문서에서 자주 사용되는 단어에 높은 가중치를 주되, 모든 문서에서 전반적으로 자주 나타나는 단어에 대해서는 패널티를 주는 방식으로 값을 부여

	text	tf	idf
0	Eddard Stark is a king in the north.	1	3
1	A king but one king : kings are everywhere.	2	3
2	Hodor was different : he was not a king .	1	3
3	But the North could not change without him.	0	3

	king	was	the	not	a		he	one	north	kings	is	in	him	everywhere	A	different	could	change	but	are	Stark	North	Hodor	Eddard
0	0.333333	0.0	0.5	0.0	0.5	0.0	0.0	0.0	1.0	0.0	1.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	1.0
1	0.666667	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	1.0	0.0	0.0	0.0	1.0	1.0	0.0	0.0	0.0	1.0	1.0	0.0	0.0	0.0	0.0
2	0.333333	2.0	0.0	0.5	0.5	1.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0
3	0.000000	0.0	0.5	0.5	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	1.0	1.0	0.0	0.0	0.0	1.0	0.0	0.0

$$w_{x,y} = tf_{x,y} \times \log\left(\frac{N}{df_x}\right)$$

TF-IDF

Term x within document y

$tf_{x,y}$ = frequency of x in y

df_x = number of documents containing x

N = total number of documents

*TF-IDF가 count기반 벡터화보다 더 높은 예측 성능 보장

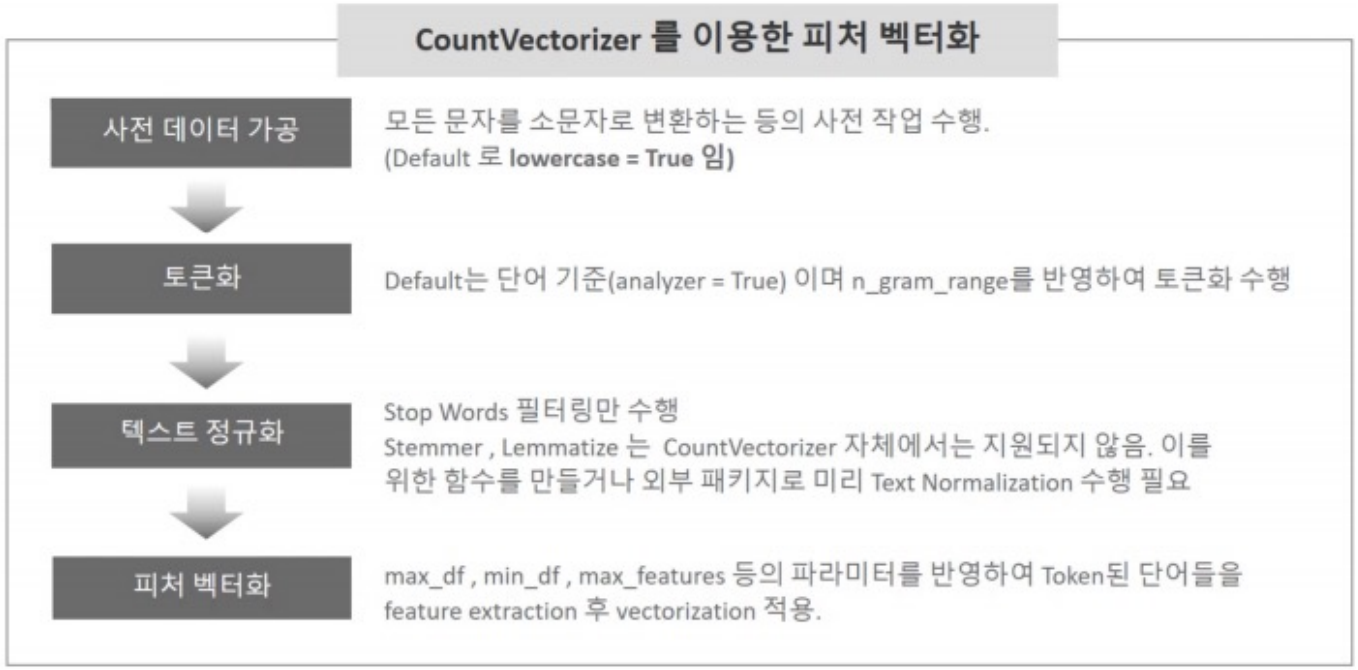
Bag of Words - BOW

CountVectorizer, Tf-idfVectorizer

- **CountVectorizer**

카운트 기반의 벡터화 구현 클래스, 텍스트의 전처리도 함께 수행

max_df	너무 높은 빈도수를 가지는 단어 피처를 제외하기 위한 파라미터
min_df	너무 낮은 빈도수를 가지는 단어 피처를 제외하기 위한 파라미터
max_features	추출하는 피처의 개수를 정수로 제한
stop_words	'english ' 라고 지정하면 영어의 스톱워드로 지정된 단어는 추출에서 제외
n_gram_range	n_gram범위를 설정
analyzer	피처 추출을 수행할 단위를 선택 (default = word)
token_pattern	토큰화를 수행하는 정규 패턴을 지정
tokenizer	토큰화를 별도의 커스텀 함수로 이용시 적용



Bag of Words - BOW

희소 행렬

대규모의 행렬이 생성시 레코드의 각 문서가 가지는 단어의 수는 제한적이기 때문에 이 행렬의 값은 대부분 0이 차지

Dense Matrix

1	2	31	2	9	7	34	22	11	5
11	92	4	3	2	2	3	3	2	1
3	9	13	8	21	17	4	2	1	4
8	32	1	2	34	18	7	78	10	7
9	22	3	9	8	71	12	22	17	3
13	21	21	9	2	47	1	81	21	9
21	12	53	12	91	24	81	8	91	2
61	8	33	82	19	87	16	3	1	55
54	4	78	24	18	11	4	2	99	5
13	22	32	42	9	15	9	22	1	21

Sparse Matrix

1	.	3	.	9	.	3	.	.	.
11	.	4	.	.	.	.	.	2	1
.	.	1	.	.	.	4	.	1	.
8	.	.	.	3	1	.	.	.	.
.	.	.	9	.	.	1	.	17	.
13	21	.	9	2	47	1	81	21	9
.	.	.	.	.	.	.	.	.	.
.	.	.	.	19	8	16	.	.	55
54	4	.	.	.	11	.	.	.	.
.	.	2	.	.	.	.	22	.	21

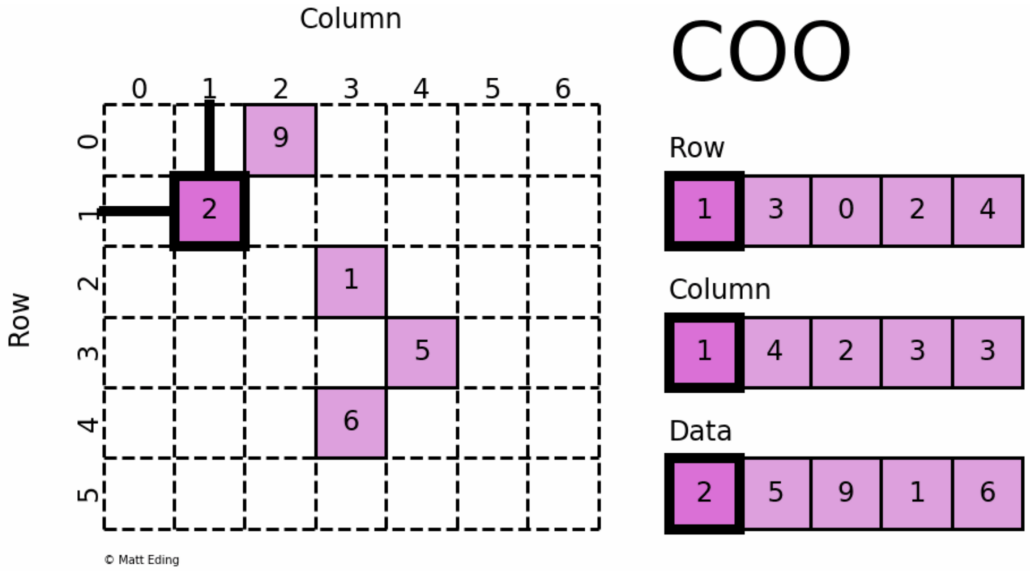
- COO 형식
- CSR 형식

⇢ 적은 메모리 공간을 차지하도록 변환

Bag of Words - BOW

1) COO

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



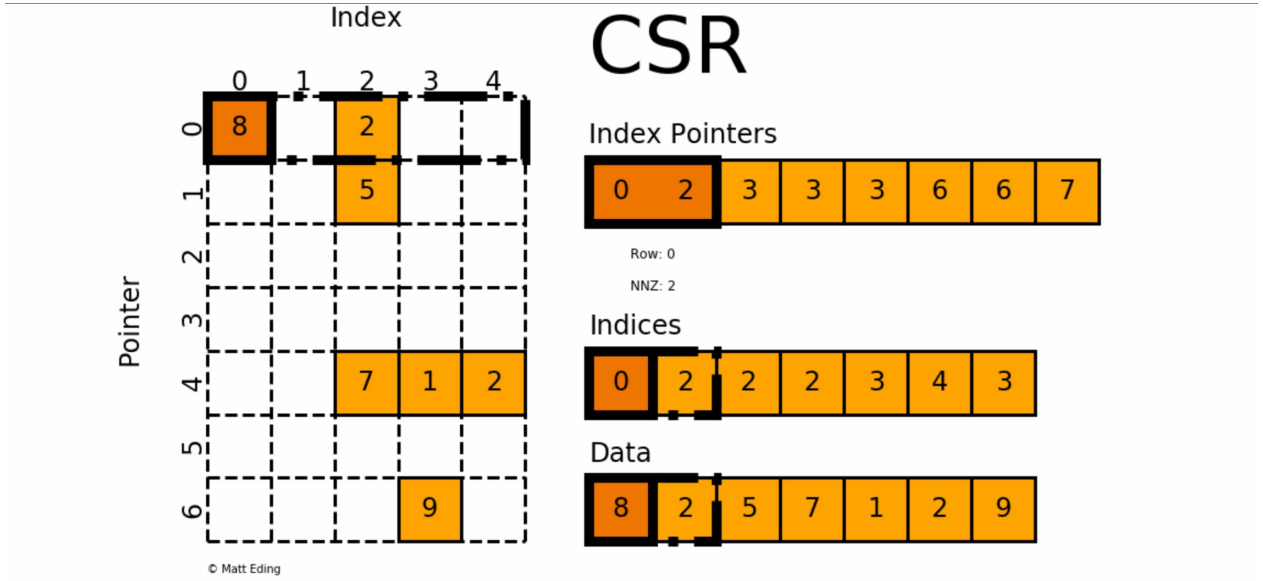
*희소행렬 변환 시 Scipy 사용

```
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)))
```

Bag of Words - BOW

2) CSR 방식

행렬의 비-0 요소들을 행의 순서대로 나열하고, 이들의 열 인덱스를 따로 저장합니다. 또한, 각 행의 첫 번째 요소가 저장된 인덱스 위치를 별도의 배열에 저장하여, 특정 행을 참조하는 데 필요한 정보를 제공합니다.



*사이파이의 csr_matrix 클래스 사용

```
# 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))
```

04. 텍스트 분류 실습

20 뉴스그룹 분류

텍스트 분류 실습

텍스트를 기반으로 분류를 수행할 때는 먼저 텍스트를 정규화한 뒤 피처 벡터화를 적용

- **실습 과정**

(1) 카운트기반, TF-IDF 기반 벡터화 진행

(2) 하이퍼 파라미터 튜닝

(3) 사이킷런 Pipeline 객체 수행

텍스트 분류 실습

① 텍스트 정규화

- key값 확인

```
from sklearn.datasets import fetch_20newsgroups

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

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

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

- target 클래스 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)
```

```
target 클래스의 값과 분포도
 0     799
1     973
2     985
3     982
4     963
5     988
6     975
7     990

Name: count, dtype: int64
target 클래스의 이름들
 ['alt.atheism', 'comp.graphics', 'comp.os.ms-windows.misc', 'comp.sys.ibm.pc.hardware', 'comp.sys.mac.hardware',
'comp.windows.x', 'misc.forsale', 'rec.autos', 'rec.motorcycles', 'rec.sport.baseball', 'rec.sport.hockey', 'sci.cr
ypt', 'sci.electronics', 'sci.med', 'sci.space', 'soc.religion.christian', 'talk.politics.guns', 'talk.politics.mid
east', 'talk.politics.misc', 'talk.religion.misc']
```

텍스트 분류 실습

① 텍스트 정규화

- 개별 데이터 확인

```
print(news_data.data[0])
```

```
From: egreen@east.sun.com (Ed Green - Pixel Cruncher)
Subject: Re: Observation re: helmets
Organization: Sun Microsystems, RTP, NC
Lines: 21
Distribution: world
Reply-To: egreen@east.sun.com
NNTP-Posting-Host: laser.east.sun.com

In article 211353@mavenry.altcit.eskimo.com, maven@mavenry.altcit.eskimo.com (Norman Hamer) writes:
>
> The question for the day is re: passenger helmets, if you don't know for
>certain who's gonna ride with you (like say you meet them at a .... church
>meeting, yeah, that's the ticket)... What are some guidelines? Should I just
>pick up another shoei in my size to have a backup helmet (XL), or should I
>maybe get an inexpensive one of a smaller size to accomodate my likely
>passenger?

If your primary concern is protecting the passenger in the event of a
crash, have him or her fitted for a helmet that is their size.  If your
primary concern is complying with stupid helmet laws, carry a real big
spare (you can put a big or small head in a big helmet, but not in a
small one).

---
Ed Green, former Ninjaite |I was drinking last night with a biker,
  Ed.Green@East.Sun.COM   |and I showed him a picture of you.  I said,
DoD #0111  (919)460-8302  |"Go on, get to know her, you'll like her!"
 (The Grateful Dead) -->  |It seemed like the least I could do...
```

텍스트 분석 의도에 벗어나는 제목과 소속, 이메일 주소 등의 헤더와 푸터 정보는 제거

→ 순수 텍스트만으로 구성된 기사 내용으로 어떤 뉴스 그룹에 속하는지 분류

텍스트 분류 실습

② 피처벡터화 변환과 머신러닝 모델 학/예/평

- CountVectorizer 피처 벡터화 변환

```
from sklearn.feature_extraction.text import CountVectorizer

# Count Vectorization으로 피처 벡터화 변환 수행
cnt_vect = CountVectorizer()
cnt_vect.fit(X_train)
X_train_cnt_vect = cnt_vect.transform(X_train)

# 학습 데이터로 fit()된 CountVectorizer를 이용해 테스트 데이터를 피처 벡터화 변환 수행
X_test_cnt_vect = cnt_vect.transform(X_test)

print('학습 데이터 텍스트 CountVectorizer Shape:', X_train_cnt_vect.shape)
```

cnt_vect.transform() 사용

```
학습 데이터 텍스트 CountVectorizer Shape: (11314, 114751)
```

반드시 학습데이터를 이용해 fit()이 수행된 CountVectorizer 객체를 이용해 테스트 데이터 변환(transform)

텍스트 분류 실습

② 피처벡터화 변환과 머신러닝 모델 학/예/평

- 로지스틱 회귀 적용

```
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score
import warnings
warnings.filterwarnings('ignore')

# LogisitcRegression을 이용하여 학습/예측/평가 수행
lr_clf = LogisticRegression(solver='liblinear')
lr_clf.fit(X_train_cnt_vect, y_train)
pred = lr_clf.predict(X_test_cnt_vect)
print('CountVectorized Logistic Regression의 예측 정확도는 {0:.3f}'.format(accuracy_score(y_test, pred)))
```

```
CountVectorized Logistic Regression의 예측 정확도는 0.731
```

텍스트 분류 실습

② 피처벡터화 변환과 머신러닝 모델 학/예/평

- TF-IDF 적용 후 로지스틱 회귀

```
from sklearn.feature_extraction.text import TfidfVectorizer

# TF-IDF 벡터화를 적용해 학습 데이터 세트와 테스트 데이터 세트 변환
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 이용해 학습/예측/평가 수행.
lr_clf = LogisticRegression(solver='liblinear')
lr_clf.fit(X_train_tfidf_vect, y_train)
pred = lr_clf.predict(X_test_tfidf_vect)
print('TF-IDF Logsitic Regression의 예측 정확도는 {0:3f}'.format(accuracy_score(y_test, pred)))
```

```
TF-IDF Logsitic Regression의 예측 정확도는 0.777748
```

단순 카운트보다 더 높은 예측 정확도 제공

텍스트 분석에서 머신러닝 모델 성능 향상시키기

1. 최적의 ML 알고리즘 선택하기
2. 최상의 피처 전처리를 수행하기

텍스트 분류 실습

③ stopwords 및 GridSearchCV

- stopwords

```
#stop words 필터링을 추가하고 ngrams을 기본 (1,1)에서 (1,2)로 변경해 피처 벡터화 적용
tfidf_vect = TfidfVectorizer(stop_words='english', ngram_range=(1,2), max_df=300)
tfidf_vect.fit(X_train)
X_train_tfidf_vect = tfidf_vect.transform(X_train)
X_test_tfidf_vect = tfidf_vect.transform(X_test)

lr_clf= LogisticRegression(solver='liblinear')
lr_clf.fit(X_train_tfidf_vect, y_train)
pred = lr_clf.predict(X_test_tfidf_vect)
print('TF-IDF Vectorized Logistic Regression의 예측 정확도는 {0:.3f}'.format(
accuracy_score(y_test, pred)))
```

```
TF-IDF Vectorized Logistic Regression의 예측 정확도는 0.794
```

- GridSearchCV

```
from sklearn.model_selection import GridSearchCV

# 최적 C 값 도출 튜닝 수행. CV는 폴드 세트로 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 C값으로 학습된 grid_cv로 예측 및 정확도 평가
pred = grid_cv_lr.predict(X_test_tfidf_vect)
print('TF-IDF Vectorized Logistic Regression의 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 Regression의 예측 정확도는 0.806
```

가장 좋은 예측 성능

텍스트 분류 실습

④ Pipeline 사용

```
from sklearn.pipeline import Pipeline

# TfidfVectorizer 객체를 tfidf_vect 객체명으로, LogisticRegression객체를 lr_clf 객체명으로 생성하는 Pipeline생성
pipeline = Pipeline([
    ('tfidf_vect', TfidfVectorizer(stop_words='english', ngram_range=(1,2), max_df=300)),
    ('lr_clf', LogisticRegression(solver='liblinear', C=10))
])

# 별도의 TfidfVectorizer객체의 fit_transform( )과 LogisticRegression의 fit(), predict( )가 필요 없음.
# pipeline의 fit( ) 과 predict( ) 만으로 한꺼번에 Feature Vectorization과 ML 학습/예측이 가능.
pipeline.fit(X_train, y_train)
pred = pipeline.predict(X_test)
print('Pipeline을 통한 Logistic Regression 의 예측 정확도는 {0:.3f}'.format(accuracy_score(y_test ,pred)))
```

```
from sklearn.pipeline import Pipeline

pipeline = Pipeline([
    ('tfidf_vect', TfidfVectorizer(stop_words='english')),
    ('lr_clf', LogisticRegression(solver='liblinear'))
])

# Pipeline에 기술된 각각의 객체 변수에 언더바(_)2개를 연달아 붙여 GridSearchCV에 사용될
# 파라미터/하이퍼 파라미터 이름과 값을 설정. .
params = { 'tfidf_vect__ngram_range': [(1,1), (1,2), (1,3)],
           'tfidf_vect__max_df': [100, 300, 700],
           'lr_clf__C': [1, 5, 10]
}

# GridSearchCV의 생성자에 Estimator가 아닌 Pipeline 객체 입력
grid_cv_pipe = GridSearchCV(pipeline, param_grid=params, cv=3 , scoring='accuracy',verbose=1)
grid_cv_pipe.fit(X_train , y_train)
print(grid_cv_pipe.best_params_ , grid_cv_pipe.best_score_)

pred = grid_cv_pipe.predict(X_test)
print('Pipeline을 통한 Logistic Regression 의 예측 정확도는 {0:.3f}'.format(accuracy_score(y_test ,pred)))
```

Pipeline 클래스를 이용해 (앞서 진행한) 피처 벡터화와 ML 알고리즘 학습/예측을 위한 코드 작성을 한번에 진행할 수 있음

수고하셨습니다