Chap 8. 텍스트 분석

NLP (National Language Processing) : 머신이 인간의 언어를 이해하는 데 중점텍스트 분석 (Text Analytics) : (텍스트 마이닝) 비정형 텍스트에서 의미 있는 정보를 추출하는 것에 중점

- 텍스트 분류 (Text Classification) : 문서가 특정 분류 또는 카테고리에 속하는 것을 예측하는 기법을 통칭한다.
- 감성 분석 (Sentiment Analysis) : 텍스트에서 나타나는 감정/판단/믿음/의견/기분 등의 주관적인 요소를 분석하는 기법을 총칭한다.
- 텍스트 요약 (Summaraization) : 텍스트 내에서 중요한 주제나 중심 사상을 추출하는 기법을 말한다. (ex. 토픽 모델링)
- 텍스트 군집화와 유사도 측정 : 비슷한 유형의 문서에 대해 군집화를 수행하는 기법을 말한다.

1. 텍스트 분석 이해

텍스트 분석은 비정형 데이터인 텍스트를 분석하는 것이다. 지금까지 머신러닝 모델은 주어진 정형 데이터 기반에서 모델을 수립하고 예측을 수행하였는데, 텍스트를 머신러닝에 적용하기 위해서는 머신러닝 알고리즘이 숫자형의 피처 기반 데이터만 입력받을 수 있기 때문에 word 기반의 다수의 피처로 추출하고 이 피처에 단어 빈도수와 같은 숫자 값을 부여한 벡터 값으로 표현하는 텍스트 변환 방법인 피처 벡터화 (피처 추출)를 거쳐야 한다. 대표적인 방법으로 BOW (Bag of Words)와 Word2Vec 방법이 있다.

텍스트 분석 수행 프로세스

1. 텍스트 사전 준비작업 (텍스트 전처리) : 텍스트를 피처로 만들기 전에 미리 대/소문자 변경, 특수문자 삭제 등의 클렌징 작업, 단어 등의 토큰화 작업, 의미 없는 단어 (Stop Word) 제거 작업, 어근 추출 (Stemming / Lemmatization) 등의 텍스트 정규화 작업을 수행하는 것을 말한다.
2. 피처 벡터화 / 추출 : 사전 준비 작업으로 가공된 텍스트에서 피처를 추출하고 여기에 벡터 값을 할당한다. 대표적인 방법으로는 BOW 와 Word2Vec이 있으며 BOW는 대표적

으로 Count 기반과 TF-IDF 기반 벡터화가 있다.

3. ML 모델 수립 및 학습/예측/평가 : 피처 벡터화된 데이터 세트에 ML 모델을 적용해 학습/예측 및 평가를 수행한다.

파이썬 기반의 NLP,텍스트 분석 패키지

- NLTK (Natural Language Toolkit for Python) : 파이썬의 가장 대표적인 NLP 패키지로 방대한 데이터 세트와 서브 모듈을 가지고 있으며 NLP의 거의 모든 영역을 커버하고 있다. 하지만 수행 속도 면에서는 아쉬운 부분이 있어 실제 대량의 데이터 기반에서는 제대로 활용되지 못하고 있다.
- Gensim : 토픽 모델링 분야에서 가장 두각을 나타내는 패키지로 오래전부터 토픽 모델링을 쉽게 구현할 수 있는 기증을 제공해 왔으며 Word2Vec 구현 등의 다양한 신기능도 제공한다.
- SpaCy : 뛰어난 수행 성능으로 최근 가장 주목을 받는 NLP 패키지이다.

2. 텍스트 사전 준비 작업(텍스트 전처리) - 텍스트 정규화

텍스트 자체를 바로 피처를 만들 수 없기 때문에 사전에 텍스트를 가공하는 준비 작업이 필요하다.

- 클렌징 (Cleansing)
- 토큰화 (Tokenization)
- 필터링 / 스톱 워드 제거 / 철자 수정
- Stemming
- Lemmatization

클렌징

텍스트에서 분석에 방해가 되는 불필요한 문자, 기호 등을 사전에 제거하는 작업 (ex. HTML, XML 태그나 특정 기호 등을 사전에 제거)

텍스트 토큰화

- **문장 토큰화**

문장의 마침표, 개행문자 등 문장의 마지막을 뜻하는 기호에 따라 분리하는 것이 일반적이다. 또한 정규 표현식에 따른 문장 토큰화도 가능하다.

```
# 각각의 문장으로 구성된 list 객체 반환
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)
```

- **단어 토큰화**

```
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)
```

```
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)
```

- stopword 제거

스톱 워드 (Stop Word)는 분석에 큰 의미가 없는 단어를 지칭함.

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

```
print('영어 stop words 갯수:',len(nltk.corpus.stopwords.words('english')))
print(nltk.corpus.stopwords.words('english')[:20])

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)
```

3. Bag of Words - BOW

머신러닝 알고리즘은 일반적으로 숫자형 피처를 데이터로 입력받아 동작하기 때문에 텍스트 자체를 바로 피처로 머신러닝 알고리즘에 입력할 수가 없다. 따라서 텍스트를 특정 의미를 가지는 숫자형 값인 벡터 값으로 변환해야 하는데, 이러환 변환을 피처 벡터화라고 한다. BOW 모델에서 피처 벡터화는 모든 문서에서 모든 단어를 칼럼 형태로 나열하고 각 문서에서 해당 단어의 횟수나 정규화된 빈도를 값으로 부여하는 데이터 세트 모델로 변경하는 것이다. BOW의 피처 벡터화 방식에는 2가지가 존재한다.

1. 카운트 기반 벡터화 : 단어 피처에 값을 부여할 때 각 문서에서 해당 단어가 나타나는 횟수에 Count를 부여하는 경우를 카운트 벡터화라고 한다. 카운트 값이 높을수록 중요한 단어로 인식되며, 이에 그 문서의 특징을 나타내는 단어가 아닌 언어의 특성상 문장에서 자주 사용될 수 밖에 없는 단어까지 높은 값을 부여하게 된다.
2. TF-IDF (Term Frequency - Inverse Document Frequency) 기반 벡터화 : 위와 같은 카운트 기반 벡터화의 문제를 보완하기 위해 개별 문서에서 자주 나는 단어에 높은 가중치를 주되, 모든 문서에서 전반적으로 자주 나타나는 단어에 대해서는 페널티를 주는 방식으로 값을 부여한다.

사이킷런의 Count 및 TF-IDF 벡터화 구현 : CountVectorizer, TfidVectorizer

CountVectorizer, TfidVectorizer의 파라미터

- max_df : 너무 높은 빈도수를 가지는 단어 피처를 제외, 정수 값을 가지면 전체 문서에 걸쳐 n개 이하로 나타나는 단어만 피처로 추출하고 부동 소수점 값을 가지면 전체 문서에 걸쳐 빈도수% 까지의 단어만 피처로 추출한다.
- min_df : 너무 낮은 빈도수를 가지는 단어 피처를 제외
- max_features : 높은 빈도를 가지는 단어 순으로 몇 개를 추출할 건지 입력
- stop_words : english로 지정하면 여어의 스톱 워드로 지정된 단어는 추출에서 제외한다.
- n_gram_range : BOW 모델이 문맥의 의미를 반영하지 못한다는 단점을 극복하기 위해 n_gram 범위를 설정한다. 튜플 형태로 (범위 최솟값, 범위 최댓값)을 지정
- analyzer : 피처 추출을 수행한 단위를 지정한다. default = 'word'
- token_pattern : 토큰화를 수행하는 정규 표현식 패턴을 지정합니다. default = '\b\w\w+\b'
- tokenizer : 토큰화를 별도의 커스텀 함수로 이용시 적용한다.

CountVectorizer 클래스를 이용한 피처 벡터화 방법

1. 사전 데이터 가공 : 영어의 경우 모든 문자를 소문자로 변경하는 등의 전처리 작업을 수행한다.
2. 토큰화 : 디폴트로 단어 기준으로 n_gram_range를 반영해 각 단어를 토큰화한다.
3. 텍스트 정규화 : Stop Word 필터링을 수행한다. (Stemmer, Lemmatize는 자체 지원 x)
4. 피처 벡터화 : max_df, min_df, max_features 등의 파라미터를 이용해 토큰화된 단어를 피처로 추출하고 단어 빈도수 벡터 값을 적용한다.

BOW 벡터화를 위한 희소 행렬

BOW 벡터화를 수행할 경우 불필요한 0값이 메모리 공간에 할당되어 메모리 공간이 많이 필요하다. 따라서 적은 메모리 공간을 차지할 수 있도록 변환이 필요하다. 대표적인 방법으로 COO 형식과 CSR 형식이 있다. 일반적으로 큰 희소 행렬을 저장하고 계산을 수행하는 능력이 CSR 형식이 더 뛰어나기 때문에 CSR 형식을 많이 사용한다.

희소 행렬 - COO형식

COO(Coordinate : 좌표) 형식은 0이 아닌 데이터만 별도의 데이터 배열에 저장하고, 그 데이터가 가리키는 행과 열의 위치를 별도의 배열로 저장하는 방식이다.

```
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]])
```

희소 행렬 - CSR형식

CSR(Compressed Sparse Row) 형식은 COO 형식이 행과 열의 위치를 나타내기 위해서 반복적인 위치 데이터를 사용해야 하는 문제점을 해결한 방식이다. COO 형식의 행 위치 배열을 자세히 살펴보면 순차적인 같은 값이 반복적으로 나타나는 것을 확인할 수 있다. 행 위치 배열이 0부터 순차적으로 증가하는 값으로 이루어졌다는 특성을 고려하면 행 위치 배열의 고유한 값의 시작 위치만 표기하는 방법으로 이러한 반복을 제거할 수 있다.

```
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]) # 데이터 값 배열

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

# 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]) # 마지막에 총 항목 개수 배열을 추가로 넣어준다
```

```
# 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]]
```

```
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)
print(coo)
print(csr)
```

```
  (0, 2)  1
  (0, 5)  5
  (1, 0)  1
  (1, 1)  4
  (1, 3)  3
  (1, 4)  2
  (1, 5)  5
  (2, 1)  6
  (2, 3)  3
  (3, 0)  2
  (4, 3)  7
  (4, 5)  8
  (5, 0)  1
```

```
  (0, 2)  1
  (0, 5)  5
  (1, 0)  1
  (1, 1)  4
  (1, 3)  3
  (1, 4)  2
  (1, 5)  5
  (2, 1)  6
  (2, 3)  3
  (3, 0)  2
  (4, 3)  7
  (4, 5)  8
  (5, 0)  1
```