8장. 텍스트 분석

NLP이냐 텍스트 분석이냐?

- NLP: 머신이 인간의 언어를 이해하고 해석하는데 중점
  - 언어 해석을 위한 기계 번역, 자동으로 해석하고 답주는 질의응답 시스템
  - 텍스트 분석을 향상하게 하는 기반 기술
- 텍스트 분석(텍스트 마이닝): 비정형 텍스트에서 의미 있는 정보 추출에 중점
  - 머신러닝, 언어 이해, 통계 등을 활용해 모델 수립하고 정보 추출해 비즈니스 인텔리전스나 예측 분석 등의 분석 작업 주로 수행
  - 텍스트 분류(문서의 특정 분류나 카테고리 예측)

- 감성 분석(텍스트의 주관적 요소 분석 - 리뷰 분석 등)
- 텍스트 요약(텍스트의 주제나 중심 사상 추출)
- 텍스트 군집화와 유사도 측정(비슷한 유형의 문서에 대해 군집화 수행)

01. 텍스트 분석의 이해

텍스트 분석은 비정형 데이터인 텍스트 분석하는 것. 머신러닝 알고리즘은 숫자형의 피처 기반 데이터만 입력받을 수 있어, 텍스트를 머신러닝에 적용하기 위해서는 비정형 텍스트 데이터를 어떻게 피처 형태로 추출하고 추출된 피처에 의미 있는 값을 부여하는가 하는 것이 매우 중요한 요소이다.

텍스트를 word 기반의 다수의 피처로 추출하고 이 피처에 단어 빈도수와 같은 숫자 값을 부여하면 텍스트는 단어의 조합인 벡터값으로 표현될 수 있는데, 이렇게 텍스트를 변환하는 것을 피처 벡터화 또는 피처 추출이라고 한다.

대표적으로 텍스트를 피처 벡터화해서 변환하는 방법에는 BOW와 Word2Vec 방법이 있다. 이 책에서는 BOW만 설명함. 텍스트를 벡터값을 가지는 피처로 변환하는 것은 머신러닝 모델을 적용하기 전에 수행해야 할 매우 중요한 요소이다.

텍스트 분석 수행 프로세스

1. 텍스트 사전 준비작업(텍스트 전처리): 텍스트를 피처로 만들기 전에 미리 클렌징, 대/소문자 변경, 특수문자 삭제 등의 클렌징 작업, 단어 등의 토큰화 작업, 의미 없는 단어 제거 작업, 어근 추출 등의 텍스트 정규화 작업 수행하는 것을 통칭함
2. 피처 벡터화/추출: 사전 준비 작업으로 가공된 텍스트에서 피처를 추출하고 여기에 벡터 값을 할당함. 대표적인 방법은 BOW와 Word2Vec이 있으며, BOW는 대표적으로 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

NTLK 오래전부터 대표적인 파이썬 NLP 패키지였지만, Genism과 SpaCy는 이걸 보완하면서 실제 업무에서 자주 활용되는 패키지.

- Genism: 토픽 모델링 분야에서 가장 두각. Word2Vec 구현 등의 다양한 신기능 제공.

- SpaCy: 뛰어난 수행 성능으로 최근 가장 주목받는 NLP 패키지.

02. 텍스트 사전 준비 작업(텍스트 전처리) - 텍스트 정규화

텍스트 자체를 바로 피처로 만들 수 없음. 이를 위해 사전에 텍스트 가공하는 준비 작업 필요함. 텍스트 정규화는 텍스트를 머신러닝 알고리즘이나 NLP 애플리케이션에 입력 데이터로 사용하기 위해 클렌징, 정제, 토큰화, 어근화 등의 다양한 텍스트 데이터의 사전 작업을 수행하는 것을 의미함.

- 클렌징
- 토큰화
- 필터링/스톱 워드 제거/철자 수정
- Stemming
- Lemmatization

클렌징

텍스트 분석에 방해되는 불필요한 문자, 기호 등을 사전에 제거. HTML, XML 태크나 특정 기호 등을 사전에 제거함.

텍스트 토큰화

토큰화의 유형은 문서에서 문장을 분리하는 문장 토큰화와 문장에서 단어를 토큰으로 분리하는 단어 토큰화로 나눌 수 있음.

문장 토큰화

문장 토큰화는 문장의 마침표, 개행문자(\n) 등 문장의 마지막을 뜻하는 기호에 따라 분리하는 것이 일반적임. 또한 정규 표현식에 따른 문장 토큰화도 가능함. NTLK에서 일반적으로 많이 쓰이는 sent_tokenize를 이용해 토큰화 수행하겠음.

다음은 3개의 문장으로 이뤄진 텍스트 문서를 문장으로 각각 분리하는 예제.

```
from nltk import sent_tokenize
import nltk
nltk.download('punkt')  # 마침표, 개행 문자등의 테이터 세트 다운로드
```

```
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)
```

→ sent_tokenize가 반환하는 것은 각각의 문장으로 구성된 list 객체.

단어 토큰화

단어 토큰화는 문장을 단어로 토큰화 하는 것. 기본적으로 공백, 콤마, 마침표, 개행문자 등으로 단어를 분리하지만, 정규 표현식을 이용해 다양한 유형으로 토큰화를 수행할 수 있음.

마침표나 개행문자와 같이 문장을 분리하는 구분자를 이용해 단어를 토큰화할 수 있으므로 Bag of Word와 같이 단어의 순서가 중요하지 않은 경우 문장 토큰화를 사용하지 않고 단어 토큰화만 사용해도 충분함. 일반적으로 문장 토큰화는 각 문장이 가지는 시맨틱적인 의미가 중요한 요소로 사용될 때 사용함.

```
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)
```

이번에는 문장 토큰화와 단어 토큰화를 조합해 문서에 대해서 모든 단어를 토큰화해 보겠음. 문서를 먼저 문장으로 나누고, 개별 문장을 다시 단어로 토큰화하는 tokenize_text() 함수를 생성.

```
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)
```

문장을 단어별로 하나씩 토큰화하면 문맥적인 의미는 무시됨. 이를 해결해보고자 도입된 것이 n-gram. n-gram은 연속된 n개의 단어를 하나의 토큰화 단위로 분리해 내는 것. n개 단어 크기 윈도우를 만들어 문장의 처음부터 오른쪽으로 움직이면서 토큰화를 수행함.

스톱 워드 제거

스톱 워드는 분석에 큰 의미가 없는 단어를 지칭함. is, the a, will 같은… 이 단어의 경우 텍스트에 빈번하기 나타나기 때문에 사전에 제거하지 않으면 중요단어로 인지될 수 있음. 이 의미 없는 단어 제거하는 것이 중요한 전처리 작업이다.

언어별로 이러한 스톱워드가 목록화 되어 있음.

```
# 먼저 NLTK의 stopwords 목록 내려받음
import nltk
nltk.download('stopwords')
```

영어에 몇개의 stopwords 있는지 보고, 20개만 확인.

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

stopword를 필터링으로 제거해 분석을 위한 의미 있는 단어만 추출

```
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)
```

Stemming과 Lemmatization

언어에서 문법적 요소에 따라 단어 다양하게 변함. 과거, 현재, 3인칭 단수 여부 등.

Stemming과 Lemmatization은 문법적 또는 의미적으로 변화하는 단어의 원형을 찾는 것. Lemmatization이 Stemming보다 정교하며 의미론적인 기반에서 단어의 원형을 찾음. Stemming은 원형 단어로 변환 시 일반적인 방법을 적용하거나 더 단순화된 방법을 적용해 원래 단어에서 일부 철자가 훼손된 어근 단어를 추출하느 ㄴ경향이 있음. L은 품사와 같은 문법적인 요소와 더 의미적인 부분 감안해 정확한 철자로 된 어근 단어 찾아줌.

먼저, LancasterStemmer를 이용해 Stemmer부터 살펴보겠음.

```
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'))
```

이번에는 WordNetLemmatizer를 이용해 L 수행하겠음.

```
from nltk.stem import WordNetLemmatizer
import nltk
nltk.download('wordnet')

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

03. Bag of Words - BOW

BOW 모델은 문서가 가지는 모든 단어를 문맥이나 순서 무시하고 일괄적으로 단어에 대해 빈도값을 부여해 피처 값을 추출하는 모델. 문서 내 모든 단어를 한꺼번에 봉투 안에 넣은 뒤에 흔들어서 섞는다는 의미…

BOW 피처 벡터화

텍스트는 특정 의미를 가지는 숫자형 값인 벡터 값으로 변환해야 하는데, 이런 변환을 피처 벡터화라고 한다.

- 카운트 기반의 벡터화
- TF-IDF 기반의 벡터화

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

사전 데이터 가공 → 토큰화 → 텍스트 정규화 → 피처 벡터화

Bow 벡터화를 위한 희소 행렬

```
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()
```

```
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())
```

```
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)
```

05. 감성 분석

문서의 주관적인 감성/의견/감정/기분 등을 파악하기 위한 방법으로 소셜 미디어, 여론조사, 온라인 리뷰, 피드백 등 다양한 분야에서 활용됨. 감성분석은 문서 내 텍스트가 나타내는 여러 가지 주관적인 단어와 문맥을 기반으로 감성 수치를 계산하는 방법을 이용한다. 감성 지수는 긍정 감성 지수와 부정 감성 지수로 구성되며 이들 지수를 합산해 긍정 감성 또는 부정 감성을 결정함.

지도학습 기반 감성 분석 실습

유명한 imdb의 영화 사이트의 영화평을 이용하겠음.

비지도학습 기반 감성 분석 소개

비지도 학습은 lexicon을 기반으로 하는 것이다. 위의 지도 감성 분석은 데이터 세트가 레이블 값을 가지고 있었는데, 많은 감성 분석용 데이터는 결정된 레이블 값을 가지고 있지 않음. 이 경우 lexicon은 유용하게 사용됨.