08 텍스트 분석

NLP는 머신이 인간의 언어를 이해하고 해석하는 데 더 중점을 두고 기술이 발전해 왔다.

텍스트 분석은 비정형 텍스트에서 의미 있는 정보를 추출하는 것에 좀 더 중점을 두고 기술이 발전해 왔다.

NLP는 기계 번역, 질의응답 시스템의 영역에서 텍스트 분석과 차별점이 있다. **NLP는 텍스트 분석을 향상하게 하는 기반 기술**이라고 볼 수 있다.

텍스트 분석의 영역은 다음과 같다.

- **텍스트 분류** : Text Categorization이라고도 한다. 문서가 특정 분류 또는 카테고리에 속하는 것을 예측하는 기법을 통칭한다. 예를 들어 특정 신문 기사 내용이 연애/정치/사회/문화 중 어떤 카테고리에 속하는 자동으로 분류하거나 스팸 메일 검출 같은 프로그램이 이에 속한다. 지도학습을 적용한다.
- **감성 분석** : 텍스트에서 나타나는 감정/판단/믿음/의견/기분 등의 주관적인 요소를 분석하는 기법을 총칭한다. 소셜 미디어 감정 분석, 영화나 제품에 대한 긍정 또는 리뷰, 여론조사 의견 분석 등의 다양한 영역에서 활용된다. Text Analytics에서 가장 활발하게 사용되고 있는 분야다. 지도학습 방법 뿐만 아니라 비지도학습을 이용해 적용할 수 있다.
- **텍스트 요약** : 텍스트 내에서 중요한 주제나 중심 사상을 추출하는 기법을 말한다. 대표적으로 토픽 모델링이 있다.
- **텍스트 군집화와 유사도 측정** : 비슷한 유형의 문서에 대해 군집화를 수행하는 기법을 말한다. 텍스트 분류를 비지도학습으로 수행하는 방법의 일환으로 사용될 수 있다.

01 텍스트 분석 이해

텍스트 분석은 **비정형 데이터인 텍스트를 분석하는 것**이다.

비정형 텍스트 데이터를 어떻게 피처 형태로 추출하고 추출된 피처에 의미 있는 값을 부여하는가 하는 것이 **매우 중요한 요소**이다.

피처 벡터화 또는 피처 추출은 **텍스트를 word 기반의 다수의 피처로 추출하고 이 피처에 단어 빈도수와 같은 숫자 값을 부여하면 텍스트는 단어의 조합인 벡터값으로 표현하는 것을 말한다.**

텍스트 분석 수행 프로세스

1. **텍스트 사전 준비작업** : 텍스트를 피처로 만들기 전에 미리 클렌징, 대/소문자 변경, 특수문자 삭제 등의 클렌징 작업, 단어 등의 토큰화 작업, 의미 없는 단어 제거 작업, 어근 추

출 등의 텍스트 정규화 작업을 수행하는 것을 통칭한다.

2. **피처 벡터화/추출** : 사전 준비 작업으로 가공된 텍스트에서 피처를 추출하고 여기에 벡터 값을 할당한다. 대표적인 방법은 BOW와 Word2Vec이 있으며, BOW는 대표적으로 Count 기반과 TF-IDF 기반 벡터화가 있다.
3. **ML 모델 수립 및 학습/예측/평가** : 피처 벡터화된 데이트 세트에 ML 모델을 적용해 학습/예측 및 평가를 수행한다.

02 텍스트 사전 준비 작업(텍스트 전처리) - 텍스트 정규화

텍스트 자체를 바로 피처로 만들 수는 없다. 사전에 텍스트를 가공하는 준비 작업이 필요하다.

텍스트 정규화 작업은 다음과 같이 분류할 수 있다.

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

클렌징

텍스트에서 분석에 오히려 방해가 되는 불필요한 문자, 기호 등을 사전에 제거하는 작업이다. 예를 들어 HTML, XML 태그나 특정 기호 등을 사전에 제거한다.

텍스트 토큰화

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

문장 토큰화

문장 토큰화는 문장의 마침표, 개행문자 등 문장의 마지막을 뜻하는 기호에 따라 분리하는 것이 일반적이다.

```
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)
```

```
[nltk_data] Downloading package punkt to
[nltk_data]     C:₩Users₩leech₩AppData₩Roaming₩nltk_data...
```

```
<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.']
```

```
[nltk_data]   Unzipping tokenizers₩punkt.zip.
```

`sent_tokenize()` 가 반환하는 것은 각각의 문장으로 구성된 `list` 객체이다. 반환된 `list` 객체가 3개의 문장으로 된 문자열을 가지고 있는 것을 알 수 있다.

단어 토큰화

단어 토큰화는 문장을 단어로 토큰화하는 것이다. 기본적으로 공백, 콤마, 마침표, 개행문자 등으로 단어를 분리하지만, 정규 표현식을 이용해 다양한 유형으로 토큰화를 수행할 수 있다.

```
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', '.']
```

NLTK에서 기본으로 제공하는 `word_tokenize()` 를 이용해 단어로 토큰화할 수 있다.

`sent_tokenize` 와 `word_tokenize` 를 조합해 문서에 대해서 모든 단어를 토큰화해보자.

```
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', 'se
e', 'it', 'out', 'your', 'window', 'or', 'on', 'your', 'television', '.'], ['You', 'feel', 'it', 'when', 'you', 'go', 'to', 'work', ',',
'or', 'go', 'to', 'church', 'or', 'pay', 'your', 'taxes', '.']]
```

문장을 단어별로 하나씩 토큰화 할 경우 문맥적인 의미는 무시될 수 밖에 없다.

이를 위해 n-gram을 도입해보자. n-gram은 연속된 n개의 단어를 하나의 토큰화 단위로 분리해 내는 것이다.

스톱 워드 제거

스톱 워드는 분석에 큰 의미가 없는 단어를 지칭한다.

```
import nltk

stopwords = nltk.corpus.stopwords.words('english')
all_tokens = []
for sentence in word_tokens:
    filtered_words = []
    for word in sentence:
        word = word.lower()
        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', '.'], ['feel', 'go', 'work', ',', 'g
o', 'church', 'pay', 'taxes', '.']]
```

Stemming과 Lemmatization

Stemming은 원형 단어로 변환 시 일반적인 방법을 적용하거나 더 단순화된 방법을 적용해 원래 단어에서 일부 철자가 훼손된 어근 단어를 추출하는 경향이 있다.

진행형, 3인칭 단수, 과거형에 따른 동사, 그리고 비교, 최상에 따른 형용사의 변화에 따라 Stemming은 더 단순하게 원형 단어를 찾아준다.

```
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
```

Lemmatization은 품사와 같은 문법적인 요소와 더 의미적인 부분을 감안해 정확한 철자로 된 어근 단어를 찾는다.

일반적으로 Lemmatization은 보다 정확한 원형 단어 추출을 위해 단어의 '품사'를 입력해줘야 한다.

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

lemma = WordNetLemmatizer()
print(lemma.lemmatize('amusing','v'),lemma.lemmatize('amuses','v'),lemma.lemmatize('a
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₩yarnp₩AppData₩Roaming₩nltk_data...
[nltk_data]   Unzipping corpora₩wordnet.zip.
```

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

03 Bag of Words - BOW

Bag of Words 모델은 문서가 가지는 모든 단어를 문맥이나 순서를 무시하고 일괄적으로 단어에 대해 빈도 값을 부여해 피처 값을 추출하는 모델이다.

BOW 모델의 장점

- 쉽고 빠른 구축이 가능하다
- 단어의 발생 횟수에 기반하고 있지만, 예상보다 문서의 특징을 잘 나타낼 수 있는 모델이어서 전통적으로 여러 분야에서 활용도가 높다

BOW 모델의 단점

- **문맥 의미 반영 부족** : 단어의 순서를 고려하지 않기 때문에 문장 내에서 단어의 문맥적인 의미가 무시된다. 물론 이를 보완하기 위해 n_gram 기법을 활용할 수 있지만, 제한적인 부분에 그치므로 언어의 많은 부분을 차지하는 문맥적인 해석을 처리하지 못하는 단점이 있다.
- **희소 행렬 문제 :** 피처 벡터화를 수행하면 희소 행렬 형태의 데이터 세트가 만들어지기 쉽다. 많은 문서에서 단어를 추출하면 매우 많은 단어가 칼럼으로 만들어진다. 그러나, 극히 일부분의 단어만이 존재하고 대부분의 데이터는 0 값으로 채워지게 된다. 이처럼 **대규모 칼럼으로 구성된 행렬에서 대부분의 값이 0으로 채워지는 행렬을 희소행렬**이라고 한다. 이와는 반대로 **대부분의 값이 0이 아닌 의미 있는 값으로 채워져 있는 행렬을 밀집행렬**이라고 한다. 희소 행렬은 일반적으로 ML 알고리즘의 수행 시간과 예측 성능을 떨어뜨리기 때문에 희소 행렬을 위한 특별한 기법이 마련되어 있다.

BOW 피처 벡터화

텍스트는 특정 의미를 가지는 숫자형 값인 벡터 값으로 변환해야 하는데, 이러한 변환을 피처 벡터화라고 한다. **피처 벡터화는 각 문서의 텍스트를 단어로 추출해 피처로 할당하고, 각 단어의 발생 빈도와 같은 값을 이 피처에 값으로 부여해 각 문서를 이 단어 피처의 발생 빈도 값으로 구성된 벡터로 만드는 기법**이다.

BOW의 피처 벡터화 방식

- 카운트 기반의 벡터화
- TF-IDF 기반의 벡터화

단어 피처에 값을 부여할 때 각 문서에서 해당 단어가 나타나는 횟수, 즉 count를 부여하는 경우를 카운트 벡터화라고 한다.

TF-IDF 는 개별 문서에서 자주 나타나는 단어에 높은 가중치를 주되, 모든 문서에서 전반적으로 자주 나타나는 단어에 대해서는 패널티를 주는 방식을 말한다.

즉, 모든 문서에서 반복적으로 자주 발생하는 단어에 대해서는 패널티를 부여하는 방식으로 단어에 대한 가중치의 균형을 맞추는 것이다.

사이킷런의 Count 및 TF-IDF 벡터화 구현

`CountVectorizer` 클래스를 통해 카운트 기반의 피처 여러 개의 문서로구성된 텍스트의 피처 벡터화 방법

1. 모든 문자를 소문자로 변경
2. 디폴트로 단어 기준으로 `n_gram_range` 를 반영해 각 단어를 토큰화
3. 텍스트 정규화를 수행
4. 토큰화된 단어를 피처로 추출하고 단어 빈도수 벡터 값을 적용

BOW 벡터화를 위한 희소 행렬

- **COO 형식**

  0이 아닌 데이터에만 별도의 데이터 배열에 저장하고, 그 데이터가 가리키는 행과 열의 위치를 별도의 배열로 저장하는 방식이다.

- **CSR 형식**

  COO 형식이 행과 열의 위치를 나타내기 위해서 반복적인 위치 데이터를 사용해야 하는 문제점을 해결한 방식이다.

  행 위치 배열 내에 있는 고유한 값의 시작 위치만 다시 별도의 위치 배열로 가지는 변환 방식을 말한다.

04 텍스트 분류 실습 - 20 뉴스그룹 분류

텍스트 분류는 특정 문서의 분류를 학습 데이터를 통해 학습해 모델을 생성한 뒤 이 학습 모델을 이용해 다른 문서의 분류를 예측하는 것이다.

텍스트 정규화

`fetch_20newsgroups()` 는 인터넷에서 로컬 컴퓨터로 데이터를 먼저 내려받은 후에 메모리로 데이터를 로딩한다.

피처 벡터화 변환과 머신러닝 모델 학습/예측/평가

- `CountVectorizer` 를 이용해 학습 데이터의 텍스트를 피처 벡터화한다.
- **테스트 데이터에서 `CountVectorizer` 를 적용할 때는 반드시 학습 데이터를 이용해 `fit()` 이 수행된 `CountVectorizer` 객체를 이용해 테스트 데이터를 변환해야 한다는 것이다.**

Count 기반으로 피처 벡터화가 적용된 데이터 세트에 대한 **로지스틱 회귀**의 예측 정확도는 약 0.617이다.

In [19]:
```
from sklearn.datasets import fetch_20newsgroups

train_news = fetch_20newsgroups(subset = 'train', remove =('headers','footers','quotes'),random_state = 156)
X_train = train_news.data
y_train = train_news.target

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)))
```

```
학습 데이터 크기 11314, 테스트 데이터 크기 7532
```

In [20]:
```
from sklearn.feature_extraction.text import CountVectorizer

cnt_vect = CountVectorizer()
cnt_vect.fit(X_train)
X_train_cnt_vect = cnt_vect.transform(X_train)

X_test_cnt_vect = cnt_vect.transform(X_test)
print('학습 데이터 텍스트의 CountVectorizer Shape:',X_train_cnt_vect.shape)
```

```
학습 데이터 텍스트의 CountVectorizer Shape: (11314, 101631)
```

In [23]:
```
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score

lr_clf = LogisticRegression()
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 0.605417
```

```
C:₩Users₩leech₩anaconda3₩lib₩site-packages₩sklearn₩linear_model₩_logistic.py:763: ConvergenceWarning: lbfgs failed to converge (status=1):
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression
  n_iter_i = _check_optimize_result(
```

TF-IDF 기반으로 벡터화를 변경해 예측 모델을 수행하자.

```
from sklearn.feature_extraction.text import TfidfVectorizer

#TF-IDF 벡터화를 적용해 학습 데이터 세트와 테스트 데이터 세트 반환
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()
lr_clf.fit(X_train_tfidf_vect,y_train)
pred = lr_clf.predict(X_test_tfidf_vect)
print('TF_IDF Logistic Regression의 예측 정확도는 {0:.3f}'.format(accuracy_score(y_test,pred)))
```

```
TF_IDF Logistic Regression의 예측 정확도는 0.674
```

TF-IDF가 단순 카운트 기반보다 훨씬 더 높은 예측도를 제공함을 알 수 있다.

텍스트 분석에서 머신러닝 모델의 성능을 향상시키는 중요한 2가지 방법

- **최적의 ML 알고리즘을 선택하는 것**
- **최상의 피처 전처리를 수행하는 것**

TfidfVectorizer 클래스의 스톱 워드를 기존 'None'에서 'english'로 변경하고, ngram_range는 기존 (1,1)에서 (1,2)로, max_df = 300으로 변경한 뒤 다시 예측 성능을 측정해보자.

```
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)

#LogisticRegression을 이용해 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 Regression의 예측 정확도는 {0:.3f}'.format(accuracy_score(y_test,pred)))
```

```
TF_IDF Logistic Regression의 예측 정확도는 0.692
```

GridSearchCV를 이용해 로지스틱 회귀의 하이퍼파라미터 최적화를 수행해보자.

로지스틱 회귀의 C 파라미터만 변경하면서 최적의 C값을 찾은 뒤 이 C값을 이용해 성능을 평가해보자.

```
Logistic Regression best C parameter : {'C': 10}
TF-IDF Vectorized Logistic Regression 의 예측 정확도는 0.701
```

여기서 C는 규제의 강도를 조절하는 alpha 값의 역수이다.

C가 10일 때 가장 좋은 예측 성능을 나타냈으며, 예측 정확도는 약 0.703으로 향상되었음을 알 수 있다.

사이킷런 파이프라인 사용 및 GridSearchCV와의 결합

Pipeline 클래스를 이용하면 피처 벡터화와 ML 알고리즘 학습/예측을 위한 코드 작성을 한 번에 진행할 수 있다.

머신러닝에서 Pipeline이란 데이터의 가공,변환 등의 전처리와 알고리즘 적용을 마치 '수도관에서 물이 흐르듯' 한꺼번에 스트림 기반으로 처리한다는 의미이다.

이렇게 하면 더 직관적인 ML 모델 코드를 생성할 수 있으며, 수행 시간을 절약할 수 있다.

```
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(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)))
```

```
Pipeline을 통한 Logistic Regression 의 예측 정확도는 0.704
```

머신러닝 모델의 학습과 예측이 Pipeline의 `fit()`과 `predict()`로 통일돼 수행됨을 알 수 있다. Pipeline 방식을 적용하면 머신러닝 코드를 더 직관적이고 쉽게 작성할 수 있다.

GridSearchCV에 Pipeline을 입력하면서 TfidfVectorizer의 파라미터와 Logistic Regressiondml 하이퍼 파라미터를 함께 최적화 한다.

```
from sklearn.pipeline import Pipeline

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

# 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)))
```

```
Fitting 3 folds for each of 27 candidates, totalling 81 fits
[Parallel(n_jobs=1)]: Done  81 out of  81 | elapsed: 32.6min finished
{'lr_clf__C': 10, 'tfidf_vect__max_df': 700, 'tfidf_vect__ngram_range': (1, 2)} 0.755524129397207
Pipeline을 통한 Logistic Regression 의 예측 정확도는 0.702
```

정확도는 크게 개선되지 않았지만, 가장 좋은 검증 세트 성능 수치를 도출할 수 있었다.