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, 특수 문자 삭제 등의 클렌징 작업, 단어 등의 토큰화 작업, 의미 없는 단어 제거 작업, 어근 추출 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.

05 감성 분석

감성 분석은 문서의 주관적인 감성/의견/감정/기분 등을 파악하기 위한 방법으로 소셜 미디어, 여론조사, 온라인 리뷰, 피드백 등 다양한 분야에서 활용되고 있다.

문서 내 텍스트가 나타내는 여러 가지 주관적인 단어와 문맥을 기반으로 감성수치를 계산하는 방법을 이용한다.

이러한 감성 분석은 지도학습과 비지도학습으로 나눌 수 있다.

- 지도학습은 학습 데이터와 타깃 레이블 값을 기반으로 감성 분석 학습을 수행한 뒤 이를 기반으로 다른 데이터의 감성 분석을 예측하는 방법으로 일반적인 텍스트 기반의 분류와 거의 동일하다.
- 비지도학습은 'Lexicon'이라는 일종의 감성 어휘 사전을 이용한다. Lexicon은 감성 분석을 위한 용어와 문맥에 대한 다양한 정보를 가지고 있으며, 이를 이용해 문서의 긍정적, 부정적 감성 여부를 판단한다.

지도학습 기반 감성 분석 실습 - IMDB 영화평

영화평의 텍스트를 분석해 감성 분석 결과가 긍정 또는 부정적인지를 예측하는 모델을 만들어보자.

 태그르 공백으로 변환하고 `re.sub`를 이용해 영어 문자열이 아닌 문자는 모두 공백으로 변환한다.

학습 데이터와 테스트용 데이터를 분리하고, Pipeline 객체를 이용해 Count 벡터화 및 TF-IDF 벡터화를 적용해보자.

CountVectorizer를 수행했을 때의 결과는 다음과 같다.

```
예측 정확도는 0.8865, ROC_AUC는 0.9506
```

TF-IDF를 수행했을 때의 결과는 다음과 같다.

```
예측 정확도는 0.8932, ROC_AUC는 0.9600
```

TF-IDF의 예측 성능이 조금 더 나아졌음을 확인할 수 있다.

비지도학습 기반 감성 분석 소개

Lexicon은 주로 감성만을 분석하기 위해 지원하는 감성 어휘 사전이다.

감성 사전은 긍정 감성 또는 부정 감성의 정도를 의미하는 수치를 가지고 있으며 이를 **감성 지수**라고 한다.

NLP에서 제공하는 **WordNet 모듈**은 방대한 영어 어휘 사전이다. 이는 **시맨틱 분석을 제공하는 어휘 사전**이다.

'시맨틱(semantic)'이란 '문맥상 의미'를 말한다. '말'이라는 것은 상황에 따라, 문맥에 따라, 화자의 몸짓이나 어조에 따라 다르게 해석될 수 있다. 동일한 단어나 문장이라도 다른 환경

과 문맥에서는 다르게 표현되거나 이해될 수 있다.

영어단어 'Present'는 '선물'이라는 의미도 있지만, '현재'라는 의미도 있다. 언어학에서는 이러한 시멘틱을 표현하기 위해서 여러가지 규칙을 정했으며, NLP 패키지는 시맨틱을 프로그램적으로 인터페이스할 수 있는 다양한 방법을 제공한다.

NLTK의 감성 사전은 다음과 같다.

- SentiWordNet : NLTK 패키지의 WordNet과 유사하게 감성 단어 전용의 WordNet을 구현한 것이다. WordNet의 Synset(각각의 품사로 구성된 개별 단어)별로 3가지 감성 점수를 할당한다. 긍정 감성, 부정 감성, 객관성 지수가 그것이다. 문장별로 단어들의 긍정 감성 지수와 부정 감성 지수를 합산하여 최종 감성 지수를 계산하고 이에 기반해 감성이 긍정인지 부정인지 결정한다.
- VADER : 주로 소셜 미디어의 텍스트에 대한 감성 분석을 제공하기 위한 패키지이다. 뛰어난 감성 분성 결과를 제공하며, 비교적 빠른 수행 시간을 보장해 대용량 텍스트 데이터에 잘 사용되는 패키지이다.
- Pattern : 예측 성능면에서 가장 주목받는 패키지이다.

SentiWordNet을 이용한 감성 분석

WordNet Synset과 SentiWordNet SentiSynset 클래스의 이해

```
synsets() 반환 type: <class 'list'>
synsets() 반환 값 개수: 18
synsets() 반환 값 [Synset('present.n.01'), Synset('present.n.02'), Synset('present.
n.03'), Synset('show.v.01'), Synset('present.v.02'), Synset('stage.v.01'), Synset
('present.v.04'), Synset('present.v.05'), Synset('award.v.01'), Synset('give.v.0
8'), Synset('deliver.v.01'), Synset('introduce.v.01'), Synset('portray.v.04'), Syns
et('confront.v.03'), Synset('present.v.12'), Synset('salute.v.06'), Synset('presen
t.a.01'), Synset('present.a.02')]
```

Synset('present.n.01')에서 present는 의미, n은 명사 품사, 01은 present의 명사 의미가 다양하기 때문에 이를 구분하는 인덱스이다.

```
##### Synset name:  present.n.01 #####
POS : noun.time
Definition:  the period of time that is happening now; any continuous stretch of ti
me including the moment of speech
Lemmas: ['present', 'nowadays']
##### Synset name:  present.n.02 #####
POS : noun.possession
Definition:  something presented as a gift
Lemmas: ['present']
##### Synset name:  present.n.03 #####
POS : noun.communication
Definition:  a verb tense that expresses actions or states at the time of speaking
Lemmas: ['present', 'present_tense']
##### Synset name:  show.v.01 #####
POS : verb.perception
Definition:  give an exhibition of to an interested audience
Lemmas: ['show', 'demo', 'exhibit', 'present', 'demonstrate']
##### Synset name:  present.v.02 #####
POS : verb.communication
Definition:  bring forward and present to the mind
Lemmas: ['present', 'represent', 'lay_out']
##### Synset name:  stage.v.01 #####
```

POS, 정의, 부명제 등으로 시멘틱적인 요소를 표현할 수 있으며, sysset은 하나의 단어가 가질 수 있는 여러 가지 시맨틱 정보를 개별 클래스로 나타낸 것이다.

```
# synset 객체를 단어별로 생성합니다.
tree = wn.synset('tree.n.01')
lion = wn.synset('lion.n.01')
tiger = wn.synset('tiger.n.02')
cat = wn.synset('cat.n.01')
dog = wn.synset('dog.n.01')

entities = [tree , lion , tiger , cat , dog]
similarities = []
entity_names = [ entity.name().split('.')[0] for entity in entities]

# 단어별 synset 들을 iteration 하면서 다른 단어들의 synset과 유사도를 측정합니다.
for entity in entities:
    similarity = [ round(entity.path_similarity(compared_entity), 2)  for compared_entity in entities ]
    similarities.append(similarity)

# 개별 단어별 synset과 다른 단어의 synset과의 유사도를 DataFrame형태로 저장합니다.
similarity_df = pd.DataFrame(similarities , columns=entity_names,index=entity_names)
similarity_df
```

	tree	lion	tiger	cat	dog
tree	1.00	0.07	0.07	0.08	0.12
lion	0.07	1.00	0.33	0.25	0.17
tiger	0.07	0.33	1.00	0.25	0.17
cat	0.08	0.25	0.25	1.00	0.20
dog	0.12	0.17	0.17	0.20	1.00

synset 객체는 단어 간의 유사도를 나타내기 위해서 `path_similarity()` 메서드를 제공한다.

SentiWordNet은 Senti_Synset 클래스를 가지고 있으며 `senti_synsets()` 는 Senti_Synset 클래스를 리스트 형태로 변환한다.

```
senti_synsets() 반환 type : <class 'list'>
senti_synsets() 반환 값 갯수: 11
senti_synsets() 반환 값 : [SentiSynset('decelerate.v.01'), SentiSynset('slow.v.02'), SentiSynset('slow.v.03'), SentiSynset('slow.a.01'), S
entiSynset('slow.a.02'), SentiSynset('dense.s.04'), SentiSynset('slow.a.04'), SentiSynset('boring.s.01'), SentiSynset('dull.s.08'), SentiS
ynset('slowly.r.01'), SentiSynset('behind.r.03')]
```

또한 SentiSynset 객체는 단어의 감성을 나타내는 감성 지수와 객관성 지수를 가지고 있다.

```
import nltk
from nltk.corpus import sentiwordnet as swn

father = swn.senti_synset('father.n.01')
print('father 긍정감성 지수: ', father.pos_score())
print('father 부정감성 지수: ', father.neg_score())
print('father 객관성 지수: ', father.obj_score())
print('₩n')
fabulous = swn.senti_synset('fabulous.a.01')
print('fabulous 긍정감성 지수: ',fabulous .pos_score())
print('fabulous 부정감성 지수: ',fabulous .neg_score())
```

```
father 긍정감성 지수:  0.0
father 부정감성 지수:  0.0
father 객관성 지수:  1.0


fabulous 긍정감성 지수:  0.875
fabulous 부정감성 지수:  0.125
```

father는 객관적인 단어로 객관성 지수가 1.0이고 fabulous는 감성 단어로서 긍정 감성 지수가 0.875, 부정 감성 지수가 0.125이다.

SentiWordNet을 이용한 영화 감상평 감성 분석

SentiWordNet을 이용해 감성 분석을 수행하는 개략적인 순서는 다음과 같다.

1. 문서를 문장 단위로 분해
2. 다시 문장을 단어 단위로 토큰화하고 품사 매김
3. 품사 태깅된 단어 기반으로 synset 객체와 senti_synset 객체를 생성
4. Senti_synset에서 긍정/부정 지수를 구하고 이를 모두 합산해 특정 임계치 값 이상일 때 긍정 감성으로, 그렇지 않을 때는 부정 감성으로 결정

SentiWordNet을 이용하기 위해서 WordNet을 이용해 문서를 다시 토큰화한 뒤 어근 추출과 품사 태깅을 적용해야한다.

품사 태깅을 수행하는 내부 함수를 생성하자.

```
from nltk.corpus import wordnet as wn

def penn_to_wn(tag):
    if tag.startswith('J'):
        return wn.ADJ
    elif tag.startswith('N'):
        return wn.NOUN
    elif tag.startswith('R'):
        return wn.ADV
    elif tag.startswith('V'):
        return wn.VERB
```

이제 문서를 문장 → 단어 토큰 → 품사 태깅 후에 SentiSynset 클래스를 생성하고 Polarity Score를 합산하는 함수를 생성하자.

```
from nltk.stem import WordNetLemmatizer
from nltk.corpus import sentiwordnet as swn
from nltk import sent_tokenize, word_tokenize, pos_tag

def swn_polarity(text):
    sentiment = -0.0
    tokens_count = 0

    lemmatizer = WordNetLemmatizer()
    raw_sentences = sent_tokenize(text)
    # 분해된 문장별로 단어 토큰 -> 품사 태깅 후에 SentiSynset 생성 -> 감성 지수 합산
    for raw_sentence in raw_sentences:
        tagged_sentence = pos_tag(word_tokenize(raw_sentence))
        for word, tag in tagged_sentence:

            wn_tag = penn_to_wn(tag)
            if wn_tag not in (wn.NOUN,wn.ADJ,wn.ADV):
                continue
            lemma = lemmatizer.lemmatize(word,pos=wn_tag)
            if not lemma:
                continue
            synsets = wn.synsets(lemma,pos=wn_tag)
            if not synsets:
                continue
            synset = synsets[0]
            swn_synset =swn.senti_synset(synset.name())
            sentiment += (swn_synset.pos_score() - swn_synset.neg_score())
            tokens_count += 1
    if not tokens_count:
        return 0
    if sentiment >= 0:
        return 1

    return 0
```

`review_df`에 새로운 칼럼인 '`preds`'를 추가해 이 칼럼에 `swn_polarity(text)`로 반환된 감성 평가를 담고 실제 감성 평가인 '`sentiment`'칼럼과 비교해서 정확도, 정밀도, 재현율 값을 모두 측정해보자.

```
review_df['preds'] = review_df['review'].apply( lambda x : swn_polarity(x) )
y_target = review_df['sentiment'].values
preds = review_df['preds'].values
```

```
from sklearn.metrics import accuracy_score, confusion_matrix, precision_score
from sklearn.metrics import recall_score, f1_score, roc_auc_score
import numpy as np

print(confusion_matrix( y_target, preds))
print("정확도:", np.round(accuracy_score(y_target , preds), 4))
print("정밀도:", np.round(precision_score(y_target , preds),4))
print("재현율:", np.round(recall_score(y_target, preds), 4))
```

```
[[7668 4832]
 [3636 8864]]
정확도: 0.6613
정밀도: 0.6472
재현율: 0.7091
```

정확도가 약 66.13%, 재현율이 70.91%이다. 전반적인 성능 평가 지표는 만족스러울 만한 수치는 아니다.

VADER를 이용한 감정 분석

VADER는 소셜 미디어의 감성 분석 용도로 만들어진 룰 기반의 Lexicon이다. VADER는 `SentimentIntensityAnalyer` 클래스를 이용해 쉽게 감성 분석을 제공한다.

```
from nltk.sentiment.vader import SentimentIntensityAnalyzer

senti_analyzer = SentimentIntensityAnalyzer()
senti_scores = senti_analyzer.polarity_scores(review_df['review'][0])
print(senti_scores)
```

```
{'neg': 0.13, 'neu': 0.743, 'pos': 0.127, 'compound': -0.7943}
```

`polarity_scores()` 메서드를 호출해 감성 점수를 구한 뒤, 해당 문서의 감성점수가 임계값 이상이면 긍정, 그렇지 않으면 부정으로 판단한다. 'neg'는 부정, 'neu'는 중립, 'pos'는 긍정, 'compund'는 neg, neu, pos score를 적절히 조합해 -1에서 1 사이의 감성 지수를 표현한 값이다.

```
def vader_polarity(review,threshold=0.1):
    analyzer = SentimentIntensityAnalyzer()
    scores = analyzer.polarity_scores(review)

    # compound 값에 기반하여 threshold 입력값보다 크면 1, 그렇지 않으면 0을 반환
    agg_score = scores['compound']
    final_sentiment = 1 if agg_score >= threshold else 0
    return final_sentiment

# apply lambda 식을 이용하여 레코드별로 vader_polarity( )를 수행하고 결과를 'vader_preds'에 저장
review_df['vader_preds'] = review_df['review'].apply( lambda x : vader_polarity(x, 0.1) )
y_target = review_df['sentiment'].values
vader_preds = review_df['vader_preds'].values

print(confusion_matrix( y_target, vader_preds))
print("정확도:", np.round(accuracy_score(y_target , vader_preds),4))
print("정밀도:", np.round(precision_score(y_target , vader_preds),4))
print("재현율:", np.round(recall_score(y_target, vader_preds),4))
```

```
[[ 6747  5753]
 [ 1858 10642]]
정확도: 0.6956
정밀도: 0.6491
재현율: 0.8514
```

정확도가 SentiWordNet보다 향상됐고, 특히 재현율은 약 85%로 매우 향상되었음을 알 수 있다.

06 토픽 모델링

토픽 모델링이란 문서 집합에 숨어있는 주제를 찾아내는 것이다. 숨겨진 주제를 효과적으로 표현할 수 있는 중심 단어를 함축적으로 추출한다.

머신러닝 기반의 토픽 모델링에 자주 사용되는 기법은 **LSA**와 **LDA**이다.

LDA 기반의 토픽 모델링을 20 뉴스그룹 데이터 세트를 이용해서 적용해보자.

모토사이클, 야구, 그래픽스, 윈도우, 중동, 기독교, 전자공학, 의학의 8개의 주제를 추출하고 이들 텍스트에 LDA 기반의 토픽 모델링을 적용해보자.

먼저 `CountVectorizer` 를 적용해서 데이터세트를 피처 벡터화한다.

```
from sklearn.datasets import fetch_20newsgroups
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.decomposition import LatentDirichletAllocation

# 모토사이클, 야구, 그래픽스, 윈도우즈, 중동, 기독교, 의학, 우주 주제를 추출.
cats = ['rec.motorcycles', 'rec.sport.baseball', 'comp.graphics', 'comp.windows.x',
        'talk.politics.mideast', 'soc.religion.christian', 'sci.electronics', 'sci.med'  ]

# 위에서 cats 변수로 기재된 category만 추출. featch_20newsgroups( )의 categories에 cats 입력
news_df= fetch_20newsgroups(subset='all',remove=('headers', 'footers', 'quotes'),
                            categories=cats, random_state=0)

#LDA 는 Count기반의 Vectorizer만 적용합니다.
count_vect = CountVectorizer(max_df=0.95, max_features=1000, min_df=2, stop_words='english', ngram_range=(1,2))
feat_vect = count_vect.fit_transform(news_df.data)
print('CountVectorizer Shape:', feat_vect.shape)
```

```
CountVectorizer Shape: (7862, 1000)
```

`LatentDirichletAllocation` 클래스의 `n_components` 파라미터를 이용해 토픽 개수를 조정한다.

```
lda = LatentDirichletAllocation(n_components=8, random_state=0)
lda.fit(feat_vect)
```

```
LatentDirichletAllocation(n_components=8, random_state=0)
```

`components_` 는 개별 토픽별로 각 word 피처가 얼마나 많이 그 토픽에 할당됐는지에 대한 수치를 가지고 있다. `components_`의 형태와 속성값을 확인해보자.

```
print(lda.components_.shape)
lda.components_
```

```
(8, 1000)

array([[3.60992018e+01, 1.35626798e+02, 2.15751867e+01, ...,
        3.02911688e+01, 8.66830093e+01, 6.79285199e+01],
       [1.25199920e-01, 1.44401815e+01, 1.25045596e-01, ...,
        1.81506995e+02, 1.25097844e-01, 9.39593286e+01],
       [3.34762663e+02, 1.25176265e-01, 1.46743299e+02, ...,
        1.25105772e-01, 3.63689741e+01, 1.25025218e-01],
       ...,
       [3.60204965e+01, 2.08640688e+01, 4.29606813e+00, ...,
        1.45056650e+01, 8.33854413e+00, 1.55690009e+01],
       [1.25128711e-01, 1.25247756e-01, 1.25005143e-01, ...,
        9.17278769e+01, 1.25177668e-01, 3.74575887e+01],
       [5.49258690e+01, 4.47009532e+00, 9.88524814e+00, ...,
        4.87048440e+01, 1.25034678e-01, 1.25074632e-01]])
```

`lda_model.components_` 값만으로는 각 토픽별 word 연관도를 보기가 어렵다.

`display_topics()` 함수를 만들어서 각 토픽별로 연관도가 높은 순으로 word를 나열해 보자.

```
def display_topics(model, feature_names, no_top_words):
    for topic_index, topic in enumerate(model.components_):
        print('Topic #',topic_index)

        topic_word_indexes = topic.argsort()[::-1]
        top_indexes = topic_word_indexes[:no_top_words]

        feature_concat = ' '.join([feature_names[i] for i in top_indexes])
        print(feature_concat)
feature_names = count_vect.get_feature_names()

display_topics(lda,feature_names, 15)
```

```
Topic # 0
year 10 game medical health team 12 20 disease cancer 1993 games years patients good
Topic # 1
don just like know people said think time ve didn right going say ll way
Topic # 2
image file jpeg program gif images output format files color entry 00 use bit 03
Topic # 3
like know don think use does just good time book read information people used post
Topic # 4
armenian israel armenians jews turkish people israeli jewish government war dos dos turkey arab armenia 000
Topic # 5
edu com available graphics ftp data pub motif mail widget software mit information version sun
Topic # 6
god people jesus church believe christ does christian say think christians bible faith sin life
Topic # 7
use dos thanks windows using window does display help like problem server need know run
```

모토사이클, 야구 주제의 경우 명확한 주제어가 추출되진 않았고, Topic 1,3,5가 주로 애매한 주제어가 추출되었고, 나머지 주제에선 관련된 주제어가 추출되었음을 확인할 수 있다.

07 문서 군집화 소개와 실습(Opinion Review 데이터 세트)

문서 군집화 개념

문서 군집화는 비슷한 텍스트 구성의 문서를 군집화하는 것이다. 문서 군집화는 동일한 군집에 속하는 문서를 같은 카테고리 소속으로 분류할 수 있으며 비지도학습 기반으로 동작한다.

Opinion Review 데이터 세트를 이용한 문서 군집화 수행하기

10 텍스트 분석 실습 - 캐글 Mercari Price Suggestion Challenge

제공되는 데이터 속성

- train_id : 데이터 id
- name : 제품명
- item_condition_id : 판매자가 제공하는 제품 상태
- category_name : 카테고리 명

- brand_name : 브랜드 이름
- price : 제품 가격, 예측을 위한 타깃 속성
- shipping : 배송비 무료 여부, 1이면 무료(판매자가 지불), 0이면 유료(구매자 지불)
- item_description : 제품에 대한 설명

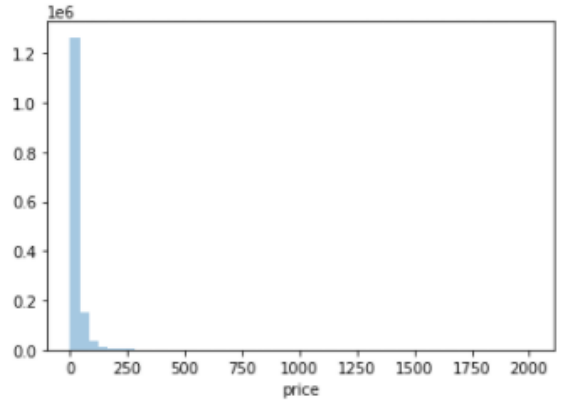
이 중에서 price가 예측해야 할 타깃이다.

```
import matplotlib.pyplot as plt
import seaborn as sns
%matplotlib inline

y_train_df = mercari_df['price']
plt.figure(figsize=(6,4))
sns.distplot(y_train_df,kde=False)
```

```
C:₩Users₩leech₩anaconda3₩lib₩site-packages₩seaborn₩distributions.py:2557: FutureWarning: `distplot` is a deprecated function and will be removed in a future version. Please adapt your code to use either `displot` (a figure-level function with similar flexibility) or `histplot` (an axes-level function for histograms).
  warnings.warn(msg, FutureWarning)
```

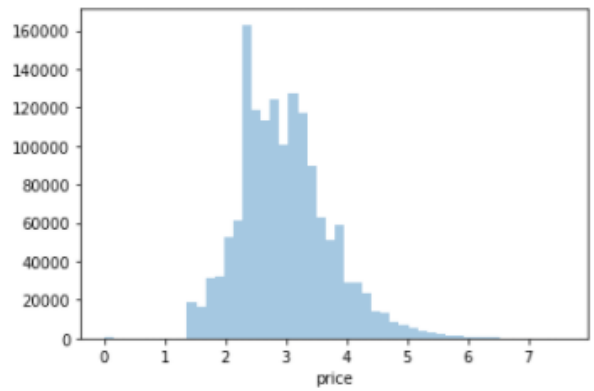
```
<AxesSubplot:xlabel='price'>
```




```
import numpy as np

y_train_df = np.log1p(y_train_df)
sns.distplot(y_train_df,kde=False)
```

```
<AxesSubplot:xlabel='price'>
```



price 값이 적은 가격을 가진 데이터 값에 왜곡 분포되어 있음을 확인할 수 있으며, 로그 값으로 변환하여 price 값을 정규분포에 가깝게 수정하자.