텍스트 데이터 처리

텍스트 데이터 처리

➢ 텍스트 정제

- 비정형 텍스트 데이터에 기본적인 정제 작업 - strip, replace, split 등 기본 문자열 메서드를 사용하여 텍스트를 바꿉니다.
- 대부분의 텍스트 데이터는 특성으로 만들기 전에 정제되어야 합니다.

```
text_data = ["    Interrobang. By Aishwarya Henriette      ",
             "Parking And Going. By Karl Gautier",
             "     Today Is The night. By Jarek Prakash    "]   # 텍스트 데이터 생성

strip_whitespace = [string.strip() for string in text_data]  # 공백 문자 제거
strip_whitespace  # 텍스트 확인

remove_periods = [string.replace(".", "") for string in strip_whitespace]  # 마침표 제거
remove_periods    # 텍스트 확인

def capitalizer(string: str) -> str:       #함수 정의
    return string.upper()
[capitalizer(string) for string in remove_periods]    # 함수 적용
# 정규 표현식을 사용하여 문자열 치환
import re
def replace_letters_with_X(string: str) -> str:      #함수 정의
    return re.sub(r"[a-zA-Z]", "X", string)
[replace_letters_with_X(string) for string in remove_periods]  # 함수 적용
```

텍스트 데이터 처리

➢ HTML 파싱과 정제

- BeautifulSoup는 HTML 스크레이핑을 위한 강력한 파이썬 라이브러리로서 HTML에 들어 있는 텍스트 데이터를 추출하는데 사용합니다.

```
from bs4 import BeautifulSoup

html = """
        <div class='full_name'><span style='font-weight:bold'>
        Masego</span> Azra</div>"              # 예제 HTML 코드 생성
        """
soup = BeautifulSoup(html, "lxml")  # html을 파싱

# "full_name" 이름의 클래스를 가진 div를 찾아 텍스트를 출력합니다.
soup.find("div", { "class" : "full_name" }).text
```

텍스트 데이터 처리

➢ 구두점 삭제

- 구두점 글자의 딕셔너리를 만들어 translate()에 적용합니다.

```
# 모든 유니코드 구두점을 키로 하고 값은 None인 punctuation 딕셔너리를 만듭니다.
# 문자열로부터 punctuation에 있는 모든 문자를 None으로 바꿉니다 (구두점을 삭제하는 효과)
import unicodedata
import sys

text_data = ['Hi!!!! I. Love. This. Song....',
             '10000% Agree!!!! #LoveIT',
             'Right?!?!']      # 구둣점 포함 텍스트 데이터 생성

# 구두점 문자로 이루어진 딕셔너리를 만듭니다.
punctuation = dict.fromkeys(i for i in range(sys.maxunicode)
                            if unicodedata.category(chr(i)).startswith('P'))

# 문자열의 구두점을 삭제합니다.
[string.translate(punctuation) for string in text_data]
```

유니코드에서 P로 시작하는 카테고리는 구두점을 의미합니다. (https://bit.ly/2vNA2of)
아스키 코드의 구두점은 import string; string.punctuation로 얻을 수 있습니다.

텍스트 데이터 처리

➢ 텍스트 토큰화

- 텍스트를 개별 단어로 나눕니다.
- 자연어 처리 툴킷인 NLTK는 단어 토큰화를 비롯해 강력한 텍스트 처리 기능을 제공

```
# 구두점 데이터를 다운로드합니다.
import nltk
nltk.download('punkt')

from nltk.tokenize import word_tokenize

string = "The science of today is the technology of tomorrow"   # 텍스트 데이터 생성
word_tokenize(string)  # 단어를 토큰으로 나눕니다.

from nltk.tokenize import sent_tokenize

string = "The science of today is the technology of tomorrow. Tomorrow is today."
sent_tokenize(string)   # 문장으로 나눕니다.
```

텍스트 데이터 처리

➢ 불용어 삭제

- 불용어는 작업 전에 삭제해야 하는 일련의 단어를 의미하기도 하지만 유용한 정보가 거의 없는 매우 자주 등장하는 단어를 의미합니다.
- NLTK의 stopwords(불용어 리스트, 179개)를 사용하여 토큰화된 단어에서 불용어를 찾고 삭제할 수 있습니다
- NLTK의 stopwords는 토큰화된 단어가 소문자라고 가정합니다.
- 사이킷런도 영어 불용어 리스트를 제공합니다. (318개)
- 사이킷런의 불용어는 frozenset 객체이기 때문에 인덱스를 사용할 수 없습니다.

```
import nltk
nltk.download('stopwords')   # 불용어 데이터를 다운로드

from nltk.corpus import stopwords
# 단어 토큰을 만듭니다.
tokenized_words = ['i', 'am', 'going', 'to', 'go', 'to', 'the', 'store', 'and', 'park']

stop_words = stopwords.words('english')  # 불용어 로드
[word for word in tokenized_words if word not in stop_words]  # 불용어 삭제
stop_words[:5]  # 불용어를 확인
```

```
from sklearn.feature_extraction.text import ENGLISH_STOP_WORDS

len(ENGLISH_STOP_WORDS), len(stop_words)
list(ENGLISH_STOP_WORDS)[:5]
```

텍스트 데이터 처리

- **불용어 삭제**
  - https://bit.ly/2vOg4Lu - 영어 전체 불용어
  - https://bit.ly/2Vs05lN - 한글 불용어
  - https://bit.ly/2VKOUnF - 한글 불용어
  - https://bit.ly/2J912sv - 한글 불용어

텍스트 데이터 처리

➢ 어간 추출

- 어간 추출은 단어의 어간을 구분하여 기본 의미를 유지하면서 어미를 제거합니다.
- (예 : tradition과 traditional은 어간 tradit을 가집니다.)
- 텍스트 데이터에서 어간을 추출하면 읽기는 힘들어지지만 기본 의미에 가까워지고 샘플 간에 비교하기에 더 좋습니다.
- NLTK의 PorterStemmer는 단어의 어미를 제거하여 어간을 바꿀 수 있습니다.

```
from nltk.stem.porter import PorterStemmer

# 단어 토큰을 만듭니다.
tokenized_words = ['i', 'am', 'humbled', 'by', 'this', 'traditional', 'meeting']

# 어간 추출기를 만듭니다.
porter = PorterStemmer()

# 어간 추출기를 적용합니다.
[porter.stem(word) for word in tokenized_words]
```

텍스트 데이터 처리

➢ 품사 태깅

- 텍스트 데이터에서 단어나 문자의 품사를 태깅
- NLTK는 품사 태그를 위해 구문 주석 말뭉치인 펜 트리뱅크(Eenn Treebank)를 사용합니다.

```
import nltk
nltk.download('averaged_perceptron_tagger')  # 태거를 다운로드

from nltk import pos_tag
from nltk import word_tokenize

text_data = "Chris loved outdoor running"   # 샘플 텍스트 데이터

# 사전 훈련된 품사 태깅을 사용합니다.
text_tagged = pos_tag(word_tokenize(text_data))
text_tagged  # 품사 확인 (단어와 품사 태그로 이루어진 튜플의 리스트 출력)
```

태그	품사
NNP	고유 명사, 단수
NN	명사, 단수 또는 불가산 명사
RB	부사
VBD	동사, 과거형
VBG	동사, 동명사 또는 현재 분사
JJ	형용사
PRP	인칭 대명사

- 텍스트가 태깅되면 태그를 사용해 특정 품사를 찾을 수 있습니다.

```
# 단어를 필터링
[word for word, tag in text_tagged if tag in ['NN','NNS','NNP','NNPS'] ]
```

텍스트 데이터 처리

➢ 품사 태깅

- 트윗 문장을 각 품사에 따라 특성으로 변환 (명사가 있을 경우 1, 그렇지 않으면 0)
- classes를 사용하면 각 특성이 어떤 품사를 나타내는지 알 수 있습니다..

```
#샘플의 트윗 문장을 각 품사에 따라 특성으로 변환
from sklearn.preprocessing import MultiLabelBinarizer

tweets = ["I am eating a burrito for breakfast",
          "Political science is an amazing field",
          "San Francisco is an awesome city"]
tagged_tweets = []

# 각 단어와 트윗을 태깅합니다.
for tweet in tweets:
    tweet_tag = nltk.pos_tag(word_tokenize(tweet))
    tagged_tweets.append([tag for word, tag in tweet_tag])

# 원-핫 인코딩을 사용하여 태그를 특성으로 변환
one_hot_multi = MultiLabelBinarizer()
one_hot_multi.fit_transform(tagged_tweets)
# classes_를 사용하면 각 특성이 어떤 품사를 나타내는지 알 수 있습니다.
one_hot_multi.classes_
```

텍스트 데이터 처리

➢ 품사 태깅

- 특별한 주제(예: 의료)에 대한 영어 텍스트가 아니라면 사전 훈련된 NLTK의 품사 태깅을 사용합니다
- pos_tag의 정확도가 매우 낮다면 NLTK를 사용하여 자신만의 태그 모델을 훈련시킬 수 있습니다.
- 태그 모델을 훈련하는데 가장 어려운 점 : 각 단어를 태깅한 많은 양의 텍스트 문서 필요
- 백오프 n-그램 태그 모델 : n-그램 태그 모델의 n은 한 단어의 품사를 예측하기 위해 고려할 이전 단어의 수
- TrigramTagger(이전 두 단어를 고려), BigramTagger(이전 한단어를 고려), UnigramTagger (그 단어 자체만 참고)

```
import nltk
nltk.download('brown')            # 브라운 코퍼스를 다운로드
from nltk.corpus import brown
from nltk.tag import UnigramTagger
from nltk.tag import BigramTagger
from nltk.tag import TrigramTagger
# 브라운 코퍼스에서 텍스트를 추출한 다음 문장으로 나눕니다.
sentences = brown.tagged_sents(categories='news')

train = sentences[:4000]  # 4,000개의 문장은 훈련용
test = sentences[4000:]  #623개는 테스트용으로 나눕니다

unigram = UnigramTagger(train)  # 백오프 태그 객체 생성
bigram = BigramTagger(train, backoff=unigram)
trigram = TrigramTagger(train, backoff=bigram)

trigram.evaluate(test)  # 정확도를 확인
```

말뭉치 또는 코퍼스는 자연언어 연구를 위해 특정한 목적을 가지고 언어의 표본을 추출한 집합

텍스트 데이터 처리

➢ 한글 품사 태깅

- KoNLPy(https://konlpy-ko.readthedocs.io) - 한글 품사 태깅을 위한 도구
- pip install konlpy
- KoNLPy는 기존 태거(tagger)들을 손쉽게 사용하도록 도와줍니다.
- 태거 - Hannanum, Kkma, Komoran, Mecab, Okt(Open Korean Text)
- pos() - KoNLPy의 태거는 품사 태깅을 위한 메서드.

```
from konlpy.tag import Okt
okt = Okt()

text = '태양계는 지금으로부터 약 46억 년 전, 거대한 분자 구름의 일부분이 중력 붕괴를 일으키면서 형성되었다'
okt.pos(text)

okt.morphs(text)  #형태소 추출

okt.nouns(text)  # 명사 추출
```

텍스트 데이터 처리

➢ 텍스트를 BoW로 인코딩하기

- 텍스트 데이터에서 특정 단어의 등장 횟수를 나타내는 특성을 만들 수 있습니다.
- BoW(bag of word) 모델 - 텍스트 데이터 있는 고유한 단어마다 하나의 특성(각 단어가 샘플에 등장하는 횟수를 담고 있음)을 만듦
- BoW 특성 행렬의 특징을 사용하면 데이터 저장 공간을 줄일 수 있습니다.
- CountVectorizer - 텍스트 데이터에서 특정 단어의 등장 횟수를 나타내는 특성을 희소행렬(대부분 0으로 구성)로 출력
- get_feature_names() - 각 특성에 연결된 단어를 확인

```
import numpy as np
from sklearn.feature_extraction.text import CountVectorizer

text_data = np.array(['I love Brazil. Brazil!',
                      'Sweden is best',
                      'Germany beats both'])

count = CountVectorizer()                    # BoW 특성 행렬을 만듭니다.
bag_of_words = count.fit_transform(text_data)
bag_of_words                                 # 특성 행렬을 확인
bag_of_words.toarray()                       #단어 카운트 행렬 확인
count.get_feature_names()                    # 특성 이름을 확인
```

텍스트 데이터 처리

➢ 텍스트를 BoW로 인코딩하기

- 희소행렬의 모든 원소를 저장하는 대신 0이 아닌 값만 저장하고 나머지 원소는 모두 0이라고 가정하면 매우 큰 특성 행렬을 다룰 때 메모리를 절약할 수 있습니다.
- CountVectorizer는 BoW 특성 행렬을 쉽게 만들 수 있는 편리한 매개변수를 제공
- ngram-range 매개변수 - 기본적으로 모든 특성은 단어 하나이지만, 각 특성을 단어 두개(2-gram)나 단어 세 개(3-gram)로 만들 수 있습니다.
- stop_words 매개변수 - 내장된 리스트나 사용자가 지정한 리스트에 포함된 유용하지 않은 단어를 제거
- vocabulary 매개변수 - 대상 단어나 구를 제한 할 수 있다
- 어휘 사전 - 텍스트에서 고유한 단어를 추출하여 순서대로 번호를 매긴 것
- 어휘 사전은 CountVectorizer의 vocabulary_ 속성에 딕셔너리로 저장

```
# 옵션을 지정하여 특성 행렬을 만듭니다.
count_2gram = CountVectorizer( ngram_range=(1,2),
                               stop_words="english",
                               vocabulary=['brazil'])
bag = count_2gram.fit_transform(text_data)

bag.toarray()                  # 특성 행렬을 확인

count_2gram.vocabulary_        # 1-그램과 2-그램을 확인
```

텍스트 데이터 처리

➢ 텍스트를 BoW로 인코딩하기

- max_df 매개변수는 단어가 등장할 문서의 최대 개수를 지정, 너무 자주 등장하는 단어를 제외할 때 사용
- min_df 매개변수는 단어가 등장하는 문서의 최소 개수를 지정, 드물게 등장하는 단어를 제외할 때 사용
- max_df와 min_df에서 0~1사이의 실숫갓을 지정하면 전체 문서 개수에 대한 비율이 됩니다.
- max_features 매개변수는 어휘 사전 크기를 제한할때 지정하며, 전체 문서에서 빈도순으로 최상위 max_features개의 단어가 추출됩니다.

텍스트 데이터 처리

➢ 단어 중요도에 가중치 부여하기

- TfidVectorizer : tf-idf(단어 빈도-역문서 빈도)를 사용해 트윗, 영화 리뷰, 연설문 등 하나의 문서에 등장하는 단어의 빈도와 다른 모든 문서에 등장하는 빈도를 비교합니다.
- tf(단어 빈도) - 한 문서에 어떤 단어가 많이 등장할수록 그 문서에 더 중요한 단어
- df(문서 빈도) - 한 단어가 많은 문서에 나타나면 이는 어떤 특정 문서에 중요하지 않은 단어
- tf와 df 두 통계치를 연결하여 각 문서가 문서에 얼마나 중요한 단어인지를 점수로 할당
- tf를 idf(역문서 빈도)에 곱합니다.

$$tf\text{-}idf(t,d) = tf(t,d) \times idf(t)$$

```
import numpy as np
from sklearn.feature_extraction.text import TfidfVectorizer

text_data = np.array(['I love Brazil. Brazil!',
                      'Sweden is best',
                      'Germany beats both'])

tfidf = TfidfVectorizer()          # tf-idf 특성 행렬을 만듭니다.
feature_matrix = tfidf.fit_transform(text_data)
feature_matrix                     # tf-idf 특성 행렬을 확인
feature_matrix.toarray()           # tf-idf 특성 행렬을 밀집 배열로 확인

tfidf.vocabulary_                  # 특성 이름을 확인
```

텍스트 데이터 처리

➢ 단어 중요도에 가중치 부여하기

- 사이킷런은 tf-idf 벡터를 정규화할 때 기본적으로 유클리드 노름을 사용합니다.
- smooth_idf 매개변수 - 기본값 True , 로그 안의 분모와 분자에 1을 더하면 모든 단어가 포함된 가상의 문서가 있는 것 같은 효과를 내고 분모가 0이 되는 것을 막아줍니다.
- 모든 문서에 포함된 단어가 있으면 로그 값이 0이되므로 전체 tf-idf 값이 0이 되는 것을 막기 위해 idf 공식 마지막에 1을 더합니다.
- smooth_idf =False로 지정하면 분모와 분자에 1을 더하지 않는 공식을 사용합니다.

$$\mathrm{idf}(t) = \log \frac{1 + n_d}{1 + \mathrm{df}(d,t)} + 1$$

n_d는 문서의 개수이고 $\mathrm{df}(d,t)$는 단어 t의 문서 빈도

- TfidVectorizer는 ngram_range, max_df, min_df, max_features 등의 매개변수를 지원