8장 텍스트 분석

날짜	@2025년 11월 24일
카테고리	개념 정리

NLP냐 텍스트 분석이냐?

- **NLP(Natural Language Processing)**
  - 기계가 인간 언어를 *이해·해석*하는 데 중점 두고 기술 발전
  - 예: *기계 번역, 질의응답 시스템* 등
- **텍스트 분석(Text Analytics / Text Mining)**
  - 비정형 텍스트에서 *의미 있는 정보 추출*에 초점 두고 기술 발전 (텍스트 토큰화)
  - NLP 기반기술 덕분에 고도화됨
  - 텍스트 분석은 **머신러닝·언어 이해·통계**를 활용하여
    비즈니스 인텔리전스, 예측 분석 등에 널리 사용됨
- 과거 **룰 기반 시스템 → 머신러닝 기반 분석**으로 발전
  (언어 규칙 대신 데이터를 학습해 모델 예측)

텍스트 분석의 주요 기술 영역

- **텍스트 분류 (Text Classification / Categorization)**
  - 지도학습 기반
  - 뉴스 카테고리 분류, 스팸 필터링 등
- **감성 분석 (Sentiment Analysis)**
  - 텍스트 속 *의견·감정·판단* 분석
  - 리뷰/여론 분석 등에 가장 활발히 사용
  - 지도·비지도 모두 적용 가능
- **텍스트 요약 (Summarization)**
  - 중심 주제/핵심 문장 도출
  - 토픽 모델링(Topic Modeling) 활용
- **텍스트 군집화·유사도 측정 (Clustering & Similarity)**
  - 비지도 기반 문서 군집

- 문서 간 유사도 계산해 그룹화

8.1 텍스트 분석 이해

텍스트 → 숫자형 피처로 변환 필요

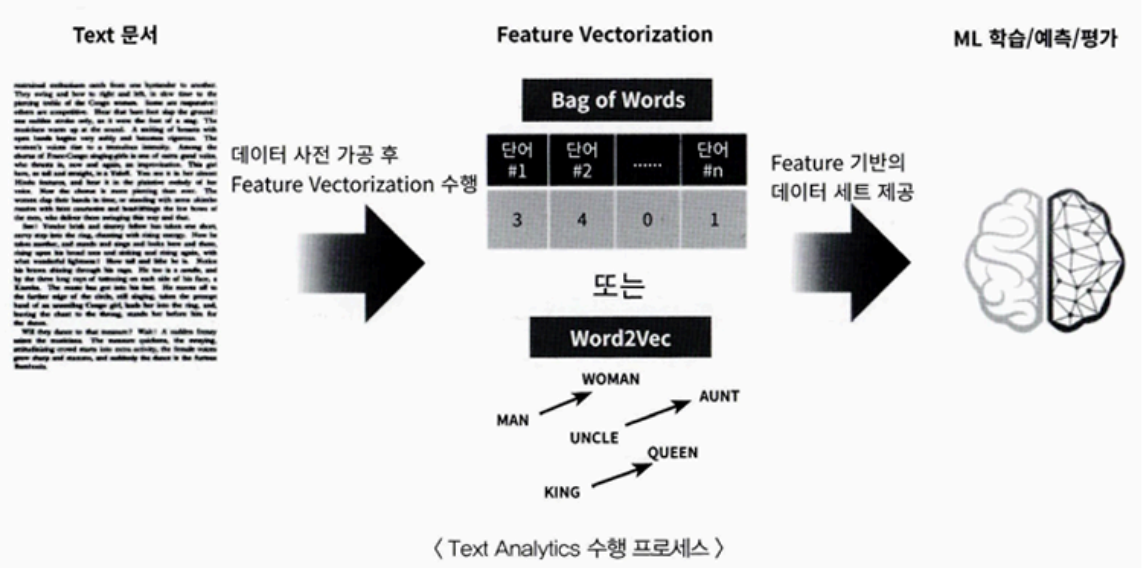
- 머신러닝 모델은 **숫자형 피처**만 입력 가능

  → 비정형 텍스트를 **벡터 형태**로 변환해야 함
- 변환 과정: **피처 벡터화(Feature Vectorization)** / **피처 추출**
- 대표적 벡터화 방법
  - **BOW(Bag of Words)**
  - **Word2Vec**

    (본 교재는 BOW 중심)

텍스트 분석 프로세스

1. **텍스트 전처리 (Preprocessing)**
   - 클렌징(대소문자 통일, 특수문자 제거)
   - 토큰화 (문장/단어)
   - 스톱워드(의미 없는 단어) 제거
   - 어근화(Stemming / Lemmatization) - 어근 추출 ... 텍스트 정규화 작업
2. **피처 벡터화 / 추출**
   - 텍스트를 word 기반의 다수의 피처로 추출하고 이 피처에 단어 빈도수와 같은 숫자 값을 부여하여 텍스트를 단어의 조합인 벡터값으로 표현하는 것
   - BOW (Bag of Words)
     - Count 기반 BOW
     - TF-IDF 기반
   - Word2Vec(단어 임베딩)
3. **ML 모델 학습·예측·평가**
   - 벡터화된 데이터를 이용해 모델 구축



〈Text Analytics 수행 프로세스〉

파이썬 기반 NLP·텍스트 분석 패키지

- **NLTK**
  - 가장 오래된 대표 NLP 패키지
  - 방대한 데이터셋 & 모듈 보유
  - 단점: 속도·정확도·대량 데이터 처리 한계
- **Gensim**
  - **토픽 모델링** 특화
  - Word2Vec 구현 등 실무 기능 다수
  - SpaCy와 함께 가장 많이 사용되는 NLP 패키지
- **SpaCy**
  - 매우 빠르고 정확한 최신 NLP 라이브러리
  - 실제 업무 환경에서 활용 증가
- **scikit-learn**
  - 전용 NLP 기능은 부족
  - 하지만 **텍스트 전처리·벡터화 기능**이 잘 갖춰져 있어
    머신러닝 기반 텍스트 분석 수행에 충분히 활용 가능
  - 실무에서는 보통 **NLTK/Gensim/SpaCy와 조합**해서 사용

8.2 텍스트 사전 준비 작업 (텍스트 전처리) - 텍스트 정규화

텍스트 정규화
= 머신러닝/NLP 입력 데이터로 사용하기 위해 **텍스트를 깨끗하고 일정한 구조로 가공하는 과정**

텍스트 정규화 주요 구성

- **클렌징(Cleansing)**
  - HTML/XML 태그, 불필요한 특수기호 제거
- **토큰화(Tokenization)**
  - 문장 토큰화(sentence-level)
  - 단어 토큰화(word-level)
- **스톱워드 제거 / 필터링**
- **어근화(Stemming)**
- **표제어 추출(Lemmatization)**

클렌징

- 텍스트에서 분석에 불필요한 요소를 제거

  (HTML 태그, 공백, 특수 문자 등)

토큰화(Tokenization)

문장 토큰화

- 마침표(.), 개행(\n) 등을 기준으로 문장 분리
- NLTK의 `sent_tokenize()` 자주 사용
- 출력: 문장 단위로 나뉜 문자열 리스트

단어 토큰화

- 단어 단위로 텍스트를 분리
- 공백, 콤마, 마침표 등을 기준
- `word_tokenize()` 사용
- 정규 표현식을 활용해 다양한 커스터마이징 가능

문장 + 단어 토큰화 조합

- 문서를 문장별로 나누고
- 각 문장을 다시 단어 단위로 토큰화
- 리스트 안에 리스트 형태의 구조 생성

  (각 문장별 단어 리스트 포함)

~~n-gram~~

- ~~단순 단어 토큰화의 단점(문맥 소실)을 보완~~
- ~~연속된 n개의 단어를 하나의 토큰으로 묶음~~
- ~~예: “Agent Smith knocks the door” (2-gram)~~
  - ~~(Agent, Smith)~~
  - ~~(Smith, knocks)~~
  - ~~(knocks, the)~~
  - ~~(the, door)~~

Stemming & Lemmatization

- 영어 단어는 시제, 수, 형태에 따라 다양한 변형이 일어남

  (work → works, worked, working /

  happy → happier, happiest 등)
- Stemming과 Lemmatization은 이런 변형을

  원래의 ‘기본 단어 형태’(Root or Lemma) 로 되돌리는 작업임

Stemming

- 단순 규칙 기반 방식으로 단어의 어근을 추출하는 방법
- 빠르지만 **철자가 훼손된 비정상적인 형태**가 나올 수 있음
- 정확도는 상대적으로 낮음
- NLTK는 Porter, Lancaster, Snowball Stemmer 등을 제공함

  특히 Lancaster는 변환이 공격적이라 어근 손실이 잦음

예시

- working, works, worked → work
- amusing, amused, amuses → amus (철자 손실 발생)
- happier, fanciest → happi, fanci (정확한 원형이 아님)

Lemmatization

- 단어의 **품사 정보**를 기반으로

  정확한 원형(Lemma) 을 찾아주는 방식
- 의미 기반 분석을 하기 때문에 Stemming보다 더 정확함

- 변환 속도는 더 느리지만 결과 품질이 높음
- NLTK에서는 WordNetLemmatizer를 사용함

  원형을 제대로 찾기 위해 품사 지정이 일반적임
  - 동사: lemmatize(word, 'v')
  - 형용사: lemmatize(word, 'a')

예시

- amusing, amused, amuses → amuse
- happier, fanciest → happy, fancy

8.3 Bag of Words - BOW

Bag of Words 모델

- 문서가 갖는 모든 단어를 문맥/순서 무시하고 일괄적으로 단어에 대해 빈도 값 부여해 피처 값 추출하는 모델
- 문서 내 모든 단어 한꺼번에 봉투(Bag) 안에 넣은 뒤 흔들어서 섞는다는 의미로 Bag of Words(BOW) 모델이라고 함

다음과 같은 2개의 문장이 있다고 가정하고 이 문장을 Bag of words의 단어 수(Word Count) 기반으로 피처를 추출해 보겠습니다.

문장 1:

'My wife likes to watch baseball games and my daughter likes to watch baseball games too'

문장 2:

'My wife likes to play baseball'

1. 문장 1과 문장 2에 있는 모든 단어에서 중복을 제거하고 각 단어(feature 또는 term)를 칼럼 형태로 나열합니다. 그러고 나서 각 단어에 고유의 인덱스를 다음과 같이 부여합니다.

   'and': 0, 'baseball': 1, 'daughter': 2, 'games': 3, 'likes': 4, 'my': 5, 'play': 6, 'to': 7, 'too': 8, 'watch': 9, 'wife': 10

2. 개별 문장에서 해당 단어가 나타나는 횟수(Occurrence)를 각 단어(단어 인덱스)에 기재합니다. 예를 들어 baseball은 문장 1, 2에서 총 2번 나타나며, daughter는 문장 1에서만 1번 나타납니다.

	Index 0	Index 1	Index 2	Index 3	Index 4	Index 5	Index 6	Index 7	Index 8	Index 9	Index 10
	and	baseball	daughter	games	likes	my	play	to	too	watch	wife
문장 1	1	2	1	2	2	2		2	1	2	1
문장 2		1			1	1	1	1			1

문장 1에서 baseball은 2회 나타남

장점

- 쉽고 빠른 구축
- 단순히 단어의 발생 횟수에 기반하고 있음에도 문서의 특징 잘 나타낼 수 있는 모델
  → 전통적으로 여러 분야에서 활용도가 높다

단점

- 문맥 의미(Semantic Context) 반영 부족
  - BOW 단어의 순서 고려 X
    → 문장 내에서 단어의 문맥적 의미 무시
  - 이를 보완하기 위해 n_gram 기법 활용할 수 있지만 제한적임
- 희소 행렬 문제
  - BOW로 피처 벡터화 수행 시 희소 행렬 형태의 데이터셋 만들어지기 쉬움
  - 많은 문서에서 단어 추출 시 매우 많은 단어가 칼럼으로 만들어짐, 문서마다 서로 다른 단어로 구성돼 단어가 나타나지 않는 경우 많음
  - **희소 행렬(Sparse Matrix)**: 대규모 칼럼으로 구성된 행렬에서 대부분 값이 0으로 채워지는 행렬
    → 일반적으로 ML 알고리즘 수행 시간과 예측 성능 떨어뜨림
  - **밀집 행렬(Dense Matrix):** 대부분 값이 0이 아닌 의미 있는 값으로 채워져 있는 행렬

텍스트는 문자 문자열 그대로는 머신러닝 모델이 이해할 수 없기 때문에 **수치 기반 피처로 변환하는 과정**이 필요

= 이 과정을 **피처 벡터화** 또는 특성 추출(Feature Extraction)

대표적인 벡터화 방식 2가지

- **BOW(Bag of Words) 기반 벡터화**
- **TF-IDF 기반 벡터화**

BOW(Bag of Words) 피처 벡터화

- 모든 단어를 칼럼 형태로 나열 → 각 문서에서 해당 단어의 횟수 / 정규화된 빈도를 값으로 부여하는 데이터셋 모델로 변경하는 것
- ex) M개 텍스트 문서에서 모든 단어를 추출해 나열했을 때, N개 단어 있다고 가정
  → 피처 벡터화 수행 시 M개 문서는 각각 N개 값이 할당된 피처의 데이터 셋이 됨
  ⇒ **M*N개 단어 피처로 이뤄진 행렬** 구성

<카운트 기반 벡터화>

- 단어 피처에 값 부여할 때 각 문서에서 해당 단어가 나타나는 횟수(count)를 부여하는 경우
- 카운트 값 높을 수록 중요단어로 인식
- but, 카운트만 부여할 경우 그 문서의 특징이 나타나기보다는 언어 특성상문장에서 자주 사용될 수밖에 없는 단어까지 높은 값 부여하는 문제 발생

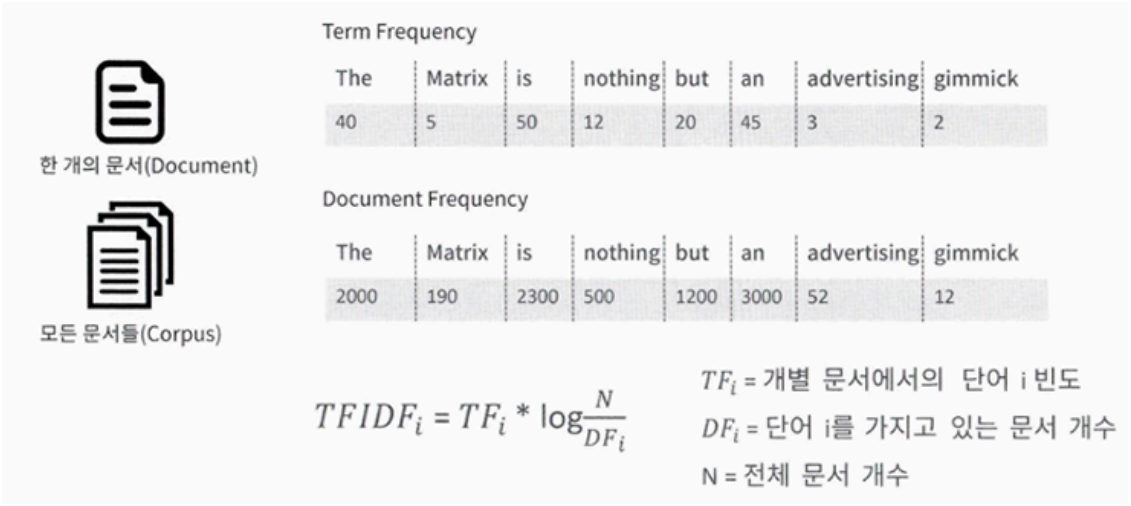
<TF-IDF(Term Frequency - Inverse Document Frequency)>

BOW의 단순 '빈도수' 문제를 개선하기 위해 등장한 방식

- 개별 문서에서 자주 나타나는 단어에 높은 가중치 주되 - 모든 문서에서 전반적으로 자주 나타나는 단어 패널티 부여 (카운트 기반 벡터화 한계 보완)
- 문서마다 텍스트 길고, 문서 개수 많은 경우에 더 좋은 예측 성능 보임
- **TF(Term Frequency)**: 문서 내 특정 단어 등장 빈도
- **IDF(Inverse Document Frequency)**: 너무 많은 문서에 흔하게 등장하는 단어는 중요도가 낮다는 가중치
  - 전체 문서에서 많이 등장할수록 IDF 값 감소
  - "the", "is" 같은 단어는 자동으로 중요도가 낮아짐
- 최종 벡터 = **TF × IDF**

장점

- 단순 Count보다 **정보적 가치가 높은 단어에 가중치**를 부여
- 문서별 특징을 더 잘 반영
- 뉴스 분류, 문서 분류에서 매우 빈번히 사용됨



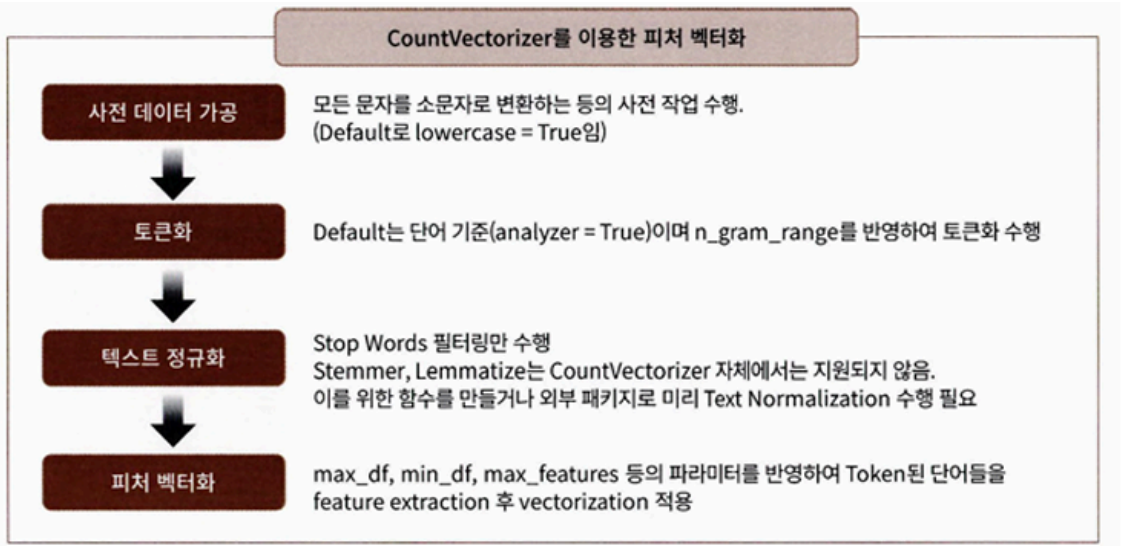
CountVectorizer 클래스 - count & TF-IDF 백터화 구현

- scikit-learn에서 제공하는 BOW 구현 도구
- 주요 기능
  - **토큰화 + 단어 사전(단어 집합) 생성 + 단어 카운트 벡터화**를 모두 수행
- 동작 방식
  1. 전체 문서에서 등장한 모든 단어를 수집해 **vocabulary(단어 집합)** 생성
  2. 문서별로 단어 등장 횟수를 세어 **문서-단어 행렬** 생성
- 텍스트 전처리 및 피처 벡터화 위한 입력 파라미터 설정
- fit(), transform() 통해 객체 반환

파라미터 명	파라미터 설명
max_df	전체 문서에 걸쳐서 너무 높은 빈도수를 가지는 단어 피처를 제외하기 위한 파라미터입니다. 너무 높은 빈도수를 가지는 단어는 스톱 워드와 비슷한 문법적인 특성으로 반복적인 단어일 가능성이 높기에 이를 제거하기 위해 사용됩니다. max_df = 100과 같이 정수 값을 가지면 전체 문서에 걸쳐 100개 이하로 나타나는 단어만 피처로 추출합니다. Max_df = 0.95와 같이 부동소수점 값(0.0 ~ 1.0)을 가지면 전체 문서에 걸쳐 빈도수 0~95%까지의 단어만 피처로 추출하고 나머지 상위 5%는 피처로 추출하지 않습니다.
min_df	전체 문서에 걸쳐서 너무 낮은 빈도수를 가지는 단어 피처를 제외하기 위한 파라미터입니다. 수백~수천 개의 전체 문서에서 특정 단어가 min_df에 설정된 값보다 적은 빈도수를 가진다면 이 단어는 크게 중요하지 않거나 가비지(garbage)성 단어일 확률이 높습니다. min_df = 2와 같이 정수 값을 가지면 전체 문서에 걸쳐서 2번 이하로 나타나는 단어는 피처로 추출하지 않습니다. min_df = 0.02와 같이 부동소수점 값(0.0 ~ 1.0)을 가지면 전체 문서에 걸쳐서 하위 2% 이하의 빈도수를 가지는 단어는 피처로 추출하지 않습니다.
max_features	추출하는 피처의 개수를 제한하며 정수로 값을 지정합니다. 가령 max_features = 2000으로 지정할 경우 가장 높은 빈도를 가지는 단어 순으로 정렬해 2000개까지만 피처로 추출합니다.
stop_words	'english'로 지정하면 영어의 스톱 워드로 지정된 단어는 추출에서 제외합니다.
n_gram_range	Bag of Words 모델의 단어 순서를 어느 정도 보강하기 위한 n_gram 범위를 설정합니다. 튜플 형태로 (범위 최솟값, 범위 최댓값)을 지정합니다. 예를 들어 (1, 1)로 지정하면 토큰화된 단어를 1개씩 피처로 추출합니다. (1, 2)로 지정하면 토큰화된 단어를 1개씩(minimum 1), 그리고 순서대로 2개씩(maximum 2) 묶어서 피처로 추출합니다.
analyzer	피처 추출을 수행한 단위를 지정합니다. 당연히 디폴트는 'word'입니다. Word가 아니라 character의 특정 범위를 피처로 만드는 특정한 경우 등을 적용할 때 사용됩니다.
token_pattern	토큰화를 수행하는 정규 표현식 패턴을 지정합니다. 디폴트 값은 '\b\w\w+\b로, 공백 또는 개행 문자 등으로 구분된 단어 분리자(\b) 사이의 2문자(문자 또는 숫자, 즉 영숫자) 이상의 단어(word)를 토큰으로 분리합니다. analyzer= 'word'로 설정했을 때만 변경 가능하나 디폴트 값을 변경할 경우는 거의 발생하지 않습니다.
tokenizer	토큰화를 별도의 커스텀 함수로 이용시 적용합니다. 일반적으로 CountTokenizer 클래스에서 어근 변환 시 이를 수행하는 별도의 함수를 tokenizer 파라미터에 적용하면 됩니다.

<피처 벡터화 방법>

1. 영어) 모든 문자 소문자로 변경 등 전처리 작업
2. 디폴트로 단어 기준 n_gram_range 반영 → 각 단어 토근화
3. 텍스트 정규화 수행
   - stop_words 파라미터 주어진 경우 스톱 워드 필터링만 가능
   - stemming, lemmatization 같은 어근 변환은 직접 지원X
     → tokenizer 파라미터에 커스텀 어근 변환 함수 적용해 수행 가능
4. max_df, min_df, max_features 등 파라미터 적용해 토큰화된 단어 피처로 추출 & 단어 빈도수 벡터값 적용



CountVectorizer vs TF-IDF Vectorizer 비교

특징	Count(BOW)	TF-IDF
기준	단어 등장 횟수	빈도 × 역문서빈도
흔한 단어	중요 단어로 취급됨	중요도가 자동 감소
장점	단순, 빠름	더 정교한 가중치
단점	의미 반영 약함	계산량 조금 증가
사용처	간단한 모델, 기초 분석	문서 분류, 감성 분석 등

BOW 벡터화를 위한 희소 행렬

1) 희소 행렬(Sparse Matrix)이란

- 텍스트를 CountVectorizer/TfidfVectorizer로 벡터화하면 **수만~수십만 개의 피처(단어)**가 생성됨
- 한 문서가 실제로 포함하는 단어는 적기 때문에 **대부분의 값이 0으로 채워진 거대한 행렬**이 만들어짐
- 이렇게 **0이 대부분을 차지하는 행렬**을 *희소 행렬*이라 부름
- 희소 행렬은
  - 불필요한 0 값 때문에 **메모리 낭비**
  - 데이터가 커서 **연산 속도 저하** 발생
- 따라서 희소 행렬을 더 압축된 방식으로 저장해야 하며 대표적인 방식이 **COO 형식**과 **CSR 형식**

2) COO 형식 (Coordinate Format)

- '좌표' 기반으로 **0이 아닌 값만 저장**하는 형식

- 저장 방식
  - 데이터 배열: 0이 아닌 값만 저장
  - 행 위치 배열: 해당 값이 있는 행 번호
  - 열 위치 배열: 해당 값이 있는 열 번호
- 예시
  - 행렬: \[3,0,1],[0,2,0]\[3, 0, 1], [0, 2, 0]\[3,0,1],[0,2,0]
  - 데이터: **[3, 1, 2]**
  - 행 위치: **[0, 0, 1]**
  - 열 위치: **[0, 2, 1]**
- Scipy의 `coo_matrix`로 쉽게 생성 가능
- 단점
  - **행/열 위치 값이 반복적으로 저장됨**

    → 메모리 효율이 떨어짐

    → 보다 큰 행렬에서는 비효율적

3) CSR 형식 (Compressed Sparse Row)

- COO의 단점을 해결하기 위한 방식
- **행 위치 배열에서 중복을 제거하고, 해당 행이 시작되는 인덱스만 저장**
- 예시
  - 기존 행 위치 배열:

    `[0, 0, 1, 1, 1, 1, 1, 2, 2, 3, 4, 4, 5]`
  - 고유 값 시작 위치만 기록 →

    `[0, 2, 7, 9, 10, 12]`
  - 마지막에 데이터 개수(=13)를 붙여 최종 CSR 인덱스 배열 생성

    → `[0, 2, 7, 9, 10, 12, 13]`
- CSR가 효율적인 이유
  - **중복된 행 값 저장이 필요 없음**
  - 메모리 사용량 감소
  - 연산 속도 향상 (특히 행 단위 계산에서 강점)
- Scipy의 `csr_matrix`로 바로 생성 가능
- 밀집 행렬(Numpy array)을 그대로 입력해도 CSR 포맷으로 자동 변환됨

CountVectorizer / TfidfVectorizer와 희소 행렬

- 두 벡터라이저 모두 **CSR 형식**의 희소 행렬을 반환함
- 이는
  - 대규모 텍스트 데이터
  - 고차원 피처(단어)
  - 머신러닝 알고리즘 적용

  에 최적화하기 위해 CSR을 사용하는 것

8.5 감성 분석

개

- 감성 분석은 문서의 **주관적 감정·의견·태도**를 파악하는 기술로,

  소셜미디어·여론·리뷰·피드백 등 다양한 데이터에 적용됨.
- 문서 내 단어·문맥을 기반으로 **감성 수치(긍정/부정)**를 계산하여 최종 감성을 분류함.
- 방법론은 크게 **지도학습(supervised)** 과 **비지도학습(unsupervised)** 으로 나뉨.

  ▷ 지도학습: 레이블이 있는 학습 데이터를 기반으로 다른 텍스트의 감성을 예측

  ▷ 비지도학습: Lexicon(감성 어휘 사전)을 활용하여 단어의 감성 점수를 합산해 문서 감성 결정

지도학습 기반 감성 분석 (IMDB 영화 리뷰)

- **데이터셋**: Kaggle의 *labeledTrainData.tsv* (ID, sentiment, review 칼럼)
- 리뷰 텍스트는 HTML 태그(br>), 숫자·특수문자가 포함되어 있어 **정규 표현식**과

  `str.replace()` , `re.sub()` 를 이용해 모두 제거.
- sentiment 를 y 값으로 분리하고 id 컬럼은 삭제하여 X 데이터셋 구성.
- `train_test_split` 으로 17,500개 train / 7,500개 test 로 분리.
- **Pipeline**을 이용해 한 번에 처리:

  ▷ CountVectorizer + LogisticRegression 성능 평가 (정확도·ROC-AUC)

  ▷ TfidfVectorizer + LogisticRegression 으로 재평가 → TF-IDF가 더 좋은 성능을 보임

비지도학습 기반 감성 분석 소개

- 레이블이 없는 감성 데이터에서 **Lexicon(감성 사전)**을 활용.
- 감성 사전은 단어별 **긍정/부정 점수(Polarity Score)**를 제공하여 문서의 감성을 계산.
- 대표적인 감성 사전:

  ▷ **SentiWordNet**: WordNet의 Synset 단위로 3가지 점수(긍정/부정/객관성) 제공

▷ **VADER**: 소셜미디어 텍스트에 강함, 빠르고 정확도 높음

▷ **Pattern**: 성능 우수하지만 Python 3.x 미지원

WordNet & Synset 개념

- WordNet은 단어의 사전 정의뿐 아니라 **시맨틱(semantic, 문맥 의미)** 정보를 담은 구조
- 단어는 품사별·의미별로 **Synset(Sets of cognitive synonyms)** 단위로 구성됨
- 동일 단어라도 문맥에 따라 여러 Synset을 가질 수 있으며, 이는 감성 사전 구성에서 중요한 요소

SentiWordNet 기반 감성 분석

- WordNet Synset에 대해 각각 **긍정/부정/객관성 점수**를 매긴 사전.
- 문서 내 단어들의 감성 점수를 합산하여 최종 감성 판단.
- 다만 전체적으로 **정확도가 높지 않아 실무 활용도는 낮음**.
- 이해 목적의 예제로 소개되며, 실제 분석에는 VADER가 더 적합.

VADER 기반 감성 분석

- 소셜미디어에 최적화된 감성 분석 도구
- 빠르고 정확도가 높아 **비지도 감성 분석의 대표적 실무 모델**로 사용
- `vader_polarity()` 함수 생성
  - 입력 파라미터로 영화 감성평 텍스트와 긍정/부정 결정하는 임곗값(threshold) 가짐
  - SentimentIntensityAnalyzer 객체의 polarity_scores() 메서드 호출해 감성 결과 반환
- 각 문서별로 감성 결과를 vader_preds 라는 review_df 의 새로운 칼럼으로 저장
- 저장된 감성 분석 결과 기반으로 VADER 예측 성능 측정

⇒ 결과) 정확도 SentiWordNet보다 향상 / 재현율 매우 크게 향상