텍스트 벡터화 (Text Vectorization)

- 텍스트 데이터를 머신러닝 모델이 처리할 수 있는 숫자 형태의 벡터로 변환하는 과정
- 컴퓨터는 텍스트 자체를 직접 이해할 수 없으므로, 텍스트의 의미와 특징을 숫자로 표현하는 것이 텍스트 마이닝 및 자연어 처리(NLP)의 핵심 단계

적용 가능한 상황

- 텍스트 분류 (스팸 메일 분류, 감성 분석 등)
- 문서 유사도 측정 및 정보 검색
- 토픽 모델링
- 텍스트 데이터를 입력으로 받는 머신러닝 모델의 전처리 단계

주의사항

- **희소 행렬 (Sparse Matrix)**
  - 텍스트 벡터화 결과는 대부분의 값이 0인 희소 행렬 형태를 보유
  - 이는 메모리 효율성을 위해 `scipy.sparse` 형식으로 저장
- **단어 사전 크기**
  - 단어 사전의 크기가 너무 커지면 벡터의 차원도 커져 계산 비용이 증가하고 과적합 위험이 존재
  - `max_features`, `min_df`, `max_df` 등의 파라미터를 통해 단어 사전을 적절히 제어해야 함
- **전처리 중요성**
  - 벡터화 전에 텍스트 전처리(토큰화, 불용어 제거, 표제어/어간 추출 등)를 얼마나 잘 수행했느냐에 따라 벡터화의 품질과 모델의 성능이 크게 달라짐

예제 데이터

```
import pandas as pd
from sklearn.feature_extraction.text import CountVectorizer, TfidfVectorizer

# 텍스트 데이터 준비
documents = [
    "The quick brown fox jumps over the lazy dog.",
    "Never jump over the lazy dog.",
    "The dog is very lazy.",
    "A quick brown fox is quick."
]
```

Bag of Words (BoW)

- 텍스트를 단어들의 가방(Bag)으로 간주하여, 단어의 순서는 무시하고 각 단어가 문서에 얼마나 많이 나타나는지에만 집중하는 모델
- **동작 방식**:
  1. 전체 문서 집합에서 고유한 단어들을 모아 단어 사전(Vocabulary)을 만듭니다.
  2. 각 문서를 단어 사전의 크기만큼의 벡터로 표현합니다.
     벡터의 각 차원은 단어 사전에 있는 특정 단어에 해당하며, 해당 단어가 문서에 나타나는 횟수

(Count)를 값으로 가집니다.
- **장점**: 간단하고 구현하기 쉽습니다.
- **단점**: 단어의 순서 정보를 잃어버려 문맥을 파악하기 어렵습니다. 희소 행렬(Sparse Matrix) 문제가 발생할 수 있습니다.

CountVectorizer

- `scikit-learn`에서 BoW 모델을 구현하는 데 사용되는 클래스
- 텍스트 문서 컬렉션을 토큰(단어) 발생 횟수의 행렬로 변환
- **주요 파라미터**:
  - `stop_words`: 불용어 리스트를 지정하여 제거합니다.
  - `ngram_range`: 단어 하나(unigram)뿐만 아니라 여러 단어의 조합(n-gram)을 토큰으로 사용할 수 있도록 합니다. (e.g., `(1, 2)`는 unigram과 bigram을 모두 사용)
  - `max_features`: 생성할 특성(단어)의 최대 개수를 제한합니다.
  - `min_df`, `max_df`: 특정 빈도수 이하/이상의 단어를 제거합니다.

```
# CountVectorizer (Bag of Words)
# CountVectorizer 하이퍼파라미터
    # stop_words: 불용어 제거. 'english' 또는 사용자 정의 리스트.
    # ngram_range: (min_n, max_n) n-gram 범위.
    # max_features: 최대 특성(단어) 개수.
    # min_df, max_df: 단어의 최소/최대 문서 빈도.
count_vectorizer = CountVectorizer(stop_words='english', ngram_range=(1, 1))
count_matrix = count_vectorizer.fit_transform(documents)

print("--- CountVectorizer 결과 ---")
print("단어 사전 (Vocabulary):", count_vectorizer.get_feature_names_out())
print("변환된 행렬 (Sparse Matrix):\n", count_matrix.toarray())
print("변환된 행렬 shape:", count_matrix.shape)
'''
단어 사전 (Vocabulary): ['brown', 'dog', 'fox', 'jump', 'jumps', 'lazy', 'quick']
변환된 행렬 (Sparse Matrix):
[[1 1 1 0 1 1 1]
 [0 1 0 1 0 1 0]
 [0 1 0 0 0 1 0]
 [1 0 1 0 0 0 2]]
변환된 행렬 shape: (4, 7)
'''
```

TF-IDF (Term Frequency-Inverse Document Frequency)

- BoW 모델의 단점을 보완하여, 단순히 단어의 등장 횟수뿐만 아니라 단어의 중요도를 반영하는 가중치 모델
- **TF (Term Frequency, 단어 빈도)**:
  - 특정 단어가 문서 내에 나타나는 횟수.
  - $TF(t, d) = \frac{단어 t가 문서 d에 나타나는 횟수}{문서 d의 총 단어 수}$
- **IDF (Inverse Document Frequency, 역문서 빈도)**:
  - 단어가 전체 문서 집합에서 얼마나 희귀하게 나타나는지를 나타내는 지표.

    - 희귀한 단어일수록 중요도가 높다고 판단.
    - $IDF(t, D) = \log(\frac{전체 문서 수}{단어 t를 포함하는 문서 수 + 1})$
- **TF-IDF**: $TF-IDF(t, d, D) = TF(t, d) \times IDF(t, D)$
- **의미**: 특정 문서에서 자주 나타나지만, 전체 문서에서는 희귀하게 나타나는 단어일수록 높은 TF-IDF 값을 가집니다. 이는 해당 단어가 문서를 특징짓는 데 중요한 역할을 한다는 의미입니다.

TfidfVectorizer

- `scikit-learn`에서 TF-IDF 모델을 구현하는 데 사용되는 클래스
- `CountVectorizer`와 유사하게 텍스트를 토큰화하고 단어 빈도를 계산한 후, TF-IDF 가중치를 적용하여 벡터로 변환
- **주요 파라미터**: `CountVectorizer`와 유사하며, `use_idf`, `smooth_idf` 등 IDF 계산과 관련된 파라미터가 추가됩니다.

```
# TfidfVectorizer
# TfidfVectorizer 하이퍼파라미터
# CountVectorizer와 유사하며, use_idf, smooth_idf 등 IDF 관련 파라미터 추가.
tfidf_vectorizer = TfidfVectorizer(stop_words='english', ngram_range=(1, 1))
tfidf_matrix = tfidf_vectorizer.fit_transform(documents)

print("\n--- TfidfVectorizer 결과 ---")
print("단어 사전 (Vocabulary):", tfidf_vectorizer.get_feature_names_out())
print("변환된 행렬 (Sparse Matrix):\n", tfidf_matrix.toarray())
print("변환된 행렬 shape:", tfidf_matrix.shape)
'''
단어 사전 (Vocabulary): ['brown', 'dog', 'fox', 'jump', 'jumps', 'lazy', 'quick']
변환된 행렬 (Sparse Matrix):
[[0.41101031  0.33274827  0.41101031  0.          0.52131446  0.33274827
0.41101031]
 [0.          0.47380449  0.          0.74230628  0.          0.47380449  0.
]
 [0.          0.70710678  0.          0.          0.          0.70710678  0.
]
 [0.40824829  0.          0.40824829  0.          0.          0.
0.81649658]]
변환된 행렬 shape: (4, 7)
'''

# TF-IDF 값 확인 (예시: 'quick' 단어의 TF-IDF)
# 'quick'의 인덱스 찾기
quick_idx = tfidf_vectorizer.vocabulary_['quick']
print(f"\n'quick' 단어의 IDF: {tfidf_vectorizer.idf_[quick_idx]:.3f}") # 1.511
```

결과 해석 방법

- `get_feature_names_out()`: 벡터화에 사용된 단어 사전(Vocabulary)을 확인할 수 있습니다.
    - `get_feature_names()`: scikit-learn 구버전에서는 `_out`은 제외하고 작성
- `toarray()`: 희소 행렬을 일반 배열로 변환하여 각 문서의 단어 빈도 또는 TF-IDF 값을 확인할 수 있습니다.

- shape: 변환된 행렬의 크기를 통해 문서의 개수와 단어 사전의 크기를 알 수 있습니다.
- **TF-IDF 값**: 특정 단어가 문서 내에서 얼마나 중요하게 다루어지는지 나타냅니다. 값이 높을수록 해당 단어가 문서를 특징짓는 데 더 중요하다고 해석할 수 있습니다.

장단점 및 대안

- **장점**:
  - **BoW/CountVectorizer**: 간단하고 직관적이며, 구현이 쉽습니다.
  - **TF-IDF/TfidfVectorizer**: 단어의 중요도를 반영하여 BoW보다 더 의미 있는 벡터를 생성합니다.
- **단점**:
  - **단어의 순서 정보 손실**: 단어의 순서를 고려하지 않으므로 문맥 정보를 잃어버립니다.
  - **희소 행렬**: 단어 사전의 크기가 커지면 벡터의 차원이 매우 커지고, 대부분의 값이 0인 희소 행렬이 되어 메모리 비효율적일 수 있습니다.
  - **의미론적 유사성 부족**: 'apple'과 '사과'처럼 의미는 같지만 형태가 다른 단어를 구분하지 못합니다.
- **대안**:
  - **Word Embeddings (단어 임베딩)**: Word2Vec, GloVe, FastText 등 단어를 고정된 크기의 밀집 벡터로 표현하며, 단어 간의 의미론적 유사성(semantic similarity)을 벡터 공간에 반영합니다.
  - **Contextual Embeddings (문맥 임베딩)**: ELMo, BERT, GPT 등 단어의 의미를 주변 문맥에 따라 다르게 표현하는 최신 임베딩 기법으로, NLP 모델의 성능을 크게 향상시켰습니다.

유사도 측정

- 코사인 유사도 : 두 벡터값에서 코사인 각도를 구해서 판단, -1~1 사이의 값을 가진다!
- 유클리디언 유사도 : 두 벡터 간의 거리로 유사도를 판단 (유클리디언 거리판단)
- 맨하탄 유사도 : 두 벡터 간의 거리로 유사도를 판단 (맨하탄 거리판단)
- 자카드 유사도 : 두 문장을 각각 단어의 집합으로 만든 뒤, 두 집합을 통해 유사도 측정
  - 유사도 측정법 : A/B
    - A : 두 집합의 교집합인 공통된 단어의 개수
    - B : 집합이 가지는 단어의 개수
  - 자카드 유사도는 0과 1 사이의 값을 가짐

```
from sklearn.metrics.pairwise import cosine_similarity, euclidean_distances,
manhattan_distances
from scipy.spatial.distance import jaccard

# 코사인 유사도 (기본)
cosine_sim = cosine_similarity(tfidf_matrix)
# 개별 비교 시 아래처럼 작성 가능 ([[0.31531524]])
print(cosine_similarity(tfidf_matrix[0], tfidf_matrix[1]))

# 유클리디언 거리 → 유사도로 변환 (1 / (1 + 거리))
euclidean_dist = euclidean_distances(tfidf_matrix)
euclidean_sim = 1 / (1 + euclidean_dist)

# 맨하탄 거리 → 유사도로 변환
manhattan_dist = manhattan_distances(tfidf_matrix)
manhattan_sim = 1 / (1 + manhattan_dist)
```

```
# 자카드 유사도 (TF-IDF는 연속값이므로 주로 binary로 변환 후 사용)
binary_matrix = (tfidf_matrix > 0).astype(int)
n_docs = len(documents)
jaccard_sim = np.zeros((n_docs, n_docs))

for i in range(n_docs):
    for j in range(n_docs):
        jaccard_sim[i, j] = 1 - jaccard(binary_matrix[i], binary_matrix[j])  # 1 -
거리 = 유사도

# 결과 출력
print("=== Cosine Similarity ===\n", np.round(cosine_sim, 3))
'''
[[1.    0.315 0.471 0.671]
 [0.315 1.    0.67  0.   ]
 [0.471 0.67  1.    0.   ]
 [0.671 0.    0.    1.   ]]
'''
print("\n=== Euclidean Similarity ===\n", np.round(euclidean_sim, 3))
'''
[[1.    0.461 0.493 0.552]
 [0.461 1.    0.552 0.414]
 [0.493 0.552 1.    0.414]
 [0.552 0.414 0.414 1.   ]]
'''
print("\n=== Manhattan Similarity ===\n", np.round(manhattan_sim, 3))
'''
[[1.    0.265 0.285 0.385]
 [0.265 1.    0.453 0.231]
 [0.285 0.453 1.    0.247]
 [0.385 0.231 0.247 1.   ]]
'''
print("\n=== Jaccard Similarity ===\n", np.round(jaccard_sim, 3))
'''
[[0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]]
'''
```