DMA 2021-1

# TEAM PROJECT 3 보고서

team 06
김세찬 오재원 이승아 표승희 홍지현

## [ PART 1 ] 검색 엔진(Search Engine)

### 1. 이론적 배경

검색 엔진이란 사용자가 정보를 요구했을 때 검색 도구를 활용하여 관련 정보를 찾아 제공하는 시스템을 이른다. 다시 말해 사용자가 원하는 정보(query)와 관련된 문서(documents)를 찾는 것을 의미한다.

검색 엔진은 먼저 사용자가 필요로 하는 정보를 수집하고 변환하는 과정을 수행한다. text 문서의 경우, stopword 제거, stemming 등의 과정을 통하여 text를 정리하고 index를 생성 및 저장한다. 이후 사용자로부터 query가 들어오면 query를 text와 유사한 방식으로 정리하고, 저장된 index와 비교하여 적절한 정보를 사용자에게 제공한다. 검색 순위를 결정하는 데에는 각 검색 엔진이 중요시하는 기준이 반영된다. 따라서 scoring 하는 방법은 다양하게 존재하며, 예시로는 query의 단어 출현 빈도에 가중치를 두거나, 웹사이트 검색의 경우 Pagerank를 고려하는 등의 방식이 있다.

### 2. SE Process별 구현 방법

#### 1) Document Pre-processing & Indexing

본 프로젝트에서는 baseline에서 제공한 index가 아닌 새로운 index를 만들었다. 이는 검색 엔진의 성능향상을 위하여 document의 전처리 방식이 달라졌기 때문이다. 본 프로젝트에서 수행한 document의 전처리는 크게 stopwords 제거, pos-tagging을 통한 tag 수정, 수정된 tag를 이용한 lemmanize의 방식으로 이루어졌다.

##### (1) Stopwords 제거

전체 document를 분석하였을 때, 전체 문서에서 100번 이상 나오는 단어들은 총 45개로, 전체 corpus와 비교하였을 때 상당히 작은 수였으며, 단순히 조사나 관사 등의 의미를 갖지 않는 단어 외에도 flow, pressure, boundary, method, heat와 같은 명사에 속하는 단어들이 18개가 들어있음을 확인하였다. 위와 같은 document 분석을 통해 얻은 단어들을 document의 stopword로 사용한 경우, nltk의 stopword를 사용했을 때보다 성능이 낮았다.

이는 전체 document의 수에 비해 stopword를 선정한 빈도를 낮게 설정했기 때문으로 보인다. 즉, 전체 document가 1400개인데 stopword들은 단어 빈도로 100번 이상 나온 단어들을 선정했기 때문에 stopword에 포함된 단어들이 실제로는 문서에서 중요한 역할을 했을 수 있다고 생각된다. 특히, 앞서 이야기한 heat나 pressure, flow의 경우 검색엔진이 아닌 사람이 읽었을 때 해당하는 문서가 유체역학이나 물리학 관련 주제를 가지고 있다고 추측할 수 있기 때문에 이러한 단어들을 stopword로 사용하는 것은 옳지 않다고 생각된다.

추가적으로 단어 빈도를 1000번 이상으로 설정하여 stopword를 구하면 nltk의 stopword에 속하는 접속사나 조사 등만 남게 되는 것을 확인하였고, 문서 기반 stopword보다 nltk의

stopword를 사용하는 것이 낫다고 판단하여 해당 라이브러리를 사용하였다.

**(2) Pos-tagging을 통한 tag 수정 & lemmanize**

본 프로젝트에서 구현한 검색엔진은 언어학적인 방식으로 query의 의미를 이해하고자 하였다. 즉, query의 품사에 따른 가중치를 다르게 주는 방식을 기반으로 하고 있다. 따라서 query와 document 모두 nltk에서 제공하는 pos-tag를 이용하여 각 term의 품사를 추가하였다. 또한, 뒤의 query processing에서 자세하게 언급하겠지만, 명사에 속하는 품사들과 형용사 혹은 수식어에 해당하는 품사들의 경우 가중치를 더 주는 방식을 사용하였기 때문에, lemmanize를 통해 미리 해당하는 tag를 가진 단어들을 동일한 품사를 가진 단어로 변경하였다. 예를 들어 words(tag: 'NNS)는 lemmanize를 통해 word('NN')으로 변경하였다.

## 2) Query Pre-processing & Query Expansion

**(1) Query Pre-processing**

whoosh에서 제공하는 검색방식은 query와 document 내의 동일한 단어를 매칭하여 text의 통계적인 값들(term frequency, inversed document frequency 등)을 만들어준다. 따라서 query에 대한 전처리는 1)에서 document에 적용한 것과 동일한 방식을 사용하였다.

**(2) Term Weighting based on Linguistics**

query에 대한 전처리 후 query의 term에 대한 가중치를 주는 작업을 진행하였다. 본 프로젝트에서는 질의어에 대한 분석이 금지되었기 때문에 언어학적인 접근방식을 선택하여 진행하였다. Reference [1]을 통해, 한국어 질의응답 데이터에 대하여 품사를 추출하고, 품사 간의 위치에 따라 서로 다른 가중치를 주어 질의응답의 효과를 높이는 작업이 가능하다는 것을 확인할 수 있었다. 따라서 본 프로젝트의 경우 영어로 된 query가 주어졌지만 위의 Reference를 참고하여 영어 의문문에서 중요하다고 생각되는 부분으로 '명사'와 '명사 주위를 수식하는 형용사'에 가중치를 주는 것을 선택하였으며, 가중치는 직접 튜닝하면서 다음과 같은 코드로 최종 구현하였다.

```
for i, pos in enumerate(tagged_list):

    tagged = get_wordnet_pos(pos[1])
    word_lem = lem.lemmatize(pos[0], pos=tagged[0])

    if pos[0].lower() not in stopWords:
        lemmed_list.append([word_lem, tagged])

    if word_lem.lower() not in stopWords:
        if tagged == ('n','t'):
            weighted_token.append(word_lem+'^1.1') #+'^1.01')

        elif tagged[0] == 'a' or tagged[0] == 'j':
            if get_wordnet_pos(tagged_list[i-1][1]) == ('n','t') or get_wordnet_pos(tagged_list[i+1][1]) == ('n','t'):
                weighted_token.append(word_lem+'^1.1') #+'^1.1')
            elif get_wordnet_pos(tagged_list[i-2][1]) == ('n','t'):
                weighted_token.append(word_lem + '^1.06')  # +'^1.1')
            else:
                weighted_token.append(word_lem)

        else:
            weighted_token.append(word_lem)
```

<코드 1-1> Term weighting with pos-tags

Reference [1]에서와 같이 조사를 제외한 형태소들에 가중치를 주는 경우 높은 성능을 보이는데, 여러 가지 조합을 시도했을 때, 위의 조합이 가장 점수를 높게 받는 것을 확인할 수 있었다. 이를 설명하자면, 일반적으로 명사는 질문의 대상이기는 하지만, 해당 명사를 수식하는

형용사로부터 더 자세한 정보를 얻을 수 있으므로, 명사와 그 주위의 형용사에 가중치를 두면 더 relevant한 문서들이 검색된다고 볼 수 있다.

**(3) Query Expansion**

본 프로젝트에서 구현한 검색엔진은 두 가지 query exapnsion을 사용하였다.

첫 번째 query expansion은 query와 연관있는 keyword를 query에 추가하는 방법이다. 이는 강의에서 배운 PRF(pseudo relevance feedback)과 유사한 방식으로 구현되었다. PRF에서 query likelyhood model로 먼저 검색을 하는 것처럼 본 프로젝트에서 구현한 검색 엔진은 두 번의 검색을 하게 되는데, 자세한 검색과정은 4) Searching & Displying Result에서 다루기로 하고, keyword를 추출하고 query에 추가하는 방법에 대해서만 설명하겠다. 앞서 전처리 및 가중치를 준 query를 사용하여 첫 번째 검색을 통해 수행하고, 상위 document들을 구한다. 해당 document들 중 자주 나오는 단어들을 keyword로 추출한다. 이를 위해 whoosh 라이브러리에서 제공하는 key_terms 함수를 사용하였으며, 사용할 상위 문서의 수와 자주 나오는 단어를 설정할 수 있었다. 따라서 두 가지 변수를 튜닝하면서 가장 높은 점수를 얻을 수 있는 상위 10개 문서에서 자주 나오는 단어 7개를 선정하였다. 그러나 이 방식은 첫 번째 검색결과가 relevant함을 단언할 수 없으므로 각 keyword를 query에 추가하기 전에 가중치를 작게 주는 과정을 포함하였다.

두 번째 query expansion은 bi-gram을 사용한 query expansion이다. 이는 query에서 연속된 term이 있고, 해당하는 phrase가 문서에 존재하는 경우 가중치를 주기 위해 사용한 방식이다. 이 방식을 채택하기 전 query를 uni-gram, bi-gram, tri-gram 및 여러 가지 조합에 따른 성능을 비교했을 때 다음과 같은 결과를 얻을 수 있었다.

| type | score |
|---|---|
| unigram | 0.3699976150960649 |
| bigram | 0.28644325928144615 |
| trigram | 0.16368008603698192 |
| unigram + bigram | 0.34733665544845316 |
| unigram^1.5 + bigram | 0.3563802002219599 |
| unigram^3 + bigram | 0.37181867273084646 |
| unigram^3.5 + bigram | 0.3727205642890792 |
| unigram^4 + bigram | 0.37185477314507837 |
| unigram^5 + bigram | 0.3700769526692274 |
| unigram^7 + bigram | 0.368968747121309 |

<표 1-1> n-gram을 사용하여 query expansion한 경우 성능 비교

위 결과에 따라 unigram과 bigram을 혼합하여 사용하는 경우 가장 높은 성능을 얻을 수 있을 것이라고 판단 하였다. 또한, 첫 번째 검색결과에 추가하여 사용할 수도 있었으나 이 경우보다 두 번째 검색을 위한 query에 추가한 경우 더 높은 점수를 얻을 수 있었기 때문에 keyword 검색과 함께 사용하였다.

그러나 대부분의 phrase는 앞서 형용사와 명사에 대한 가중치를 줄 때 사용된 방식에서 이미 가중치가 부여되었을 수 있으므로, bi-gram을 적용할 때 1보다 작은 가중치를 주었으며 첫 번째 검색에서 찾아내지 못한 phrase에 가중치를 주기 위해 사용되었다.

두 가지 query expansion은 다음과 같은 코드를 통해 구현하였다.

```
keywords = [keyword for keyword, score
            in results.key_terms("contents", docs=10, numterms=7)]
add_keyword_query = new_q

for word in keywords:
    add_keyword_query += ' '+ word+'^0.35'

new_bi = ''
bigram = [word[0] for word in lemmed_list]
bigram_2 = list(zip(bigram, bigram[1:]))
bigram_3 = ['\"' + ' '.join(word) + '\"' for word in bigram_2]

for word in bigram_3:
    new_bi += ' '+word+'^0.3'

add_keyword_query += ' ' + new_bi
```

<코드 1-2> query expansion_key word 추가 & bi-gram 추가

### 3) Scoring

앞선 과정을 통해 얻은 각 query에 대해 document의 score를 매기고, 이를 순차적으로 나열하여 연관성이 높은 document를 반환하는 과정이다. 현재 document, query term에 대한 정보가 주어져 있고, 이를 통해 문서 내 단어 빈도, 역문서 빈도, 전체 데이터셋 내 단어 빈도 등을 계산할 수 있다. 이러한 정보들을 바탕으로 tf-idf 기반의 scoring function을 활용해보고자 했다.

tf*idf 기반의 함수식은 문서 내 단어 빈도와 역문서 빈도를 곱하여 score를 계산하고, 문서 내의 단어 빈도수가 높을수록, 전체 문서에서 단어 등장 문서 수가 적을수록 그 값이 커진다. 다만 score가 tf에 비례하여 값이 커지므로, 문서 내에 단어가 많이 등장할수록 지속하여 증가하게 되며 이는 검색 결과에 부정적인 영향을 미칠 수 있다. 이를 보완하기 위해 사용하는 함수식이 BM25이다. BM25는 일정 수준 이상의 tf을 가질 경우, score에 미치는 영향에 큰 차이가 없도록 tf term을 조정하여, tf-idf에 비해 tf의 영향력을 줄였다. 또한 fl/avgfl term을 추가하여 길이가 짧은 문서의 영향력을 높였다. 따라서 본 프로젝트에서는 tf-idf에 비하여 보다 나은 성능을 가진 BM25를 적용하고자 하였고, 이에 Kamphuis C.(2020)에서 제시하는 BM25의 다양한 변형 꼴에 대해 검색 엔진을 수행해보고, 평균 BPREF을 비교하여 가장 높은 값을 갖는 함수식을 선택하였다.

실험해본 함수식은 총 5가지이며, 해당 식은 아래와 같다. 식은 모두 tf-idf을 기반으로 하는 BM25와 유사한 꼴을 띄며, tf와 idf를 가공하는 방식에 약간의 차이를 보였다.

| | |
|---|---|
| BM25 | $\left(\log\left(\frac{dc}{df+1}\right)+1\right)\cdot\frac{tf\cdot(K1+1)}{K_1\cdot\left(1-B+B\cdot\left(\frac{fl}{avgfl}\right)\right)+tf}$ |
| Robertson et al. | $\log\left(\frac{dc-df+0.5}{df+0.5}\right)\cdot\frac{tf}{K_1\cdot\left(1-B+B\cdot\left(\frac{fl}{avgfl}\right)\right)+tf}$ |
| Lucene (accurate) | $\log\left(1+\frac{dc-df+0.5}{df+0.5}\right)\cdot\frac{tf}{K_1\cdot\left(1-B+B\cdot\left(\frac{fl}{avgfl}\right)\right)+tf}$ |
| ATIRE | $\log\left(\frac{dc}{df}\right)\cdot\frac{tf\cdot(K1+1)}{K_1\cdot\left(1-B+B\cdot\left(\frac{fl}{avgfl}\right)\right)+tf}$ |
| $TF_{l\circ\delta\circ p}\times IDF$ | $\log\left(\frac{dc+1}{df}\right)\cdot\left(1+\log\left(1+\log\left(\frac{tf}{\left(1-B+B\cdot\left(\frac{fl}{avgfl}\right)\right)}+\delta\right)\right)\right)$ |

<그림 1-1> BM25 유사 함수식

위의 함수식에서 사용된 변수의 설명은 아래와 같다.

| | |
|---|---|
| dc | 전체 document 수 |
| df | term이 등장한 document 수 |
| tf | document 내 term 빈도 |
| fl | document 내 단어 개수 |
| avgfl | document 당 평균 단어 개수 |

<표 1-2> 함수식 변수 설명

제공된 CustomScoring에서 idf의 식이 log(dc / df + 1)) + 1 인 것을 토대로 df를 계산하여 추가 parameter로 사용하였다. 위의 다섯 가지 식을 custom scoring function으로 적용하여 1), 2)로 얻어진 document, query에 대하여 scoring을 진행하였다. 상수 K1, B는 각각 2, 0.75로 두고 계산하였고, 결과로 나온 평균 BPREF는 아래와 같다.

| Scoring Function | 평균 BPREF |
|---|---|
| BM25 | 0.3914657606323226 |
| Robertson et al. | 0.3900867397076411 |
| Lucene (accurate) | 0.3903928308101211 |
| ATIRE | 0.39062767154954253 |
| $TF_{l\circ\delta\circ p}\times IDF$ | 0.3955755654517156 |
| TF*IDF | 0.20818680479923612 |

<표 1-3> Function 별 평균 BPREF

실험 결과, 함수식에 따라 평균 BPREF값이 큰 차이를 보이지는 않았으나, BM25 변형 식이 모두 tf-idf 방식보다는 큰 값을 가짐을 알 수 있었다. 간소한 차이기는 하나, $TF_{l\circ\delta\circ p}\times IDF$ 함수식의 평균 BPREF가 가장 높은 것으로 확인되었고, 따라서 custom scoring function으로 해당 함수식을 선택하여 scoring을 진행하였다.

### 4) Searching

앞서 2)-(3)에서 언급했던 것과 같이 본 프로젝트에서 구현한 검색엔진은 두 번의 검색을 통해 결과를 보여준다. 첫 번째 검색의 경우 PRF에서 query likelihood model을 사용하여 검색결과를 먼저 도출하는 것처럼 전처리 및 품사에 따른 weighting이 된 첫 번째 query를 사

용하여 첫 번째 결과를 도출한다. 해당 결과를 바로 사용하지 않고, 이 결과로부터 keyword를 추출하여 query에 추가한다. 또한, query의 bi-gram또한 query에 추가하여 expansion된 query를 얻는다. 두 번째 검색의 경우, expansion된 query를 사용하여 전체 문서에 대해 검색을 진행하고, 이로부터 얻은 결과를 검색결과로 보여준다. 해당 방식을 도입했을 때의 성능 차이를 확인하기 위하여, 첫 번째 검색만 사용하여 검색결과를 나타내는 경우와 두 번째 검색까지 사용하여 검색결과를 나타내는 경우의 점수와 score가 0인 문서들의 수를 다음 표와 같이 정리해 볼 수 있다.

| | 평균 BPREF | score가 0인 문서 |
|---|---|---|
| 첫 번째 검색만 사용 | 0.3681165190164179 | 15개 |
| 두 번째 검색까지 사용 | 0.3955755654517156 | 18개 |

<표 1-4> query expansion을 사용한 검색에서의 성능비교

두 번의 검색과정을 거치므로 시간적 효율은 떨어지나, 성능 면에서는 첫 번째 검색보다 약 3%의 성능이 높아진 것을 볼 수 있었다. 그러나 score가 0인 문서가 늘어난 것을 확인할 수 있었는데, 해당 검색 결과들의 경우 첫 번째 검색에서 score가 낮았던 문서들에 대해서 query expansion을 진행하면서 더 엄격한 기준이 도입된 것과 같은 효과가 나타난 것으로 보인다. 이 문제점에 대한 개선방안은 첫 번째 검색에서 relevant하다고 판단된 문서들을 우선적으로 결과에 포함하고, 이를 기준으로 검색을 한다면 score가 0인 문서를 줄일 수 있을 것이라고 생각된다. 그러나 본 프로젝트에서는 relevant한 문서 정보를 가져와서 사용할 수 없으므로, 평균 BPREF를 증가시킬 수 있는 두 번째 방식을 선택하여 구현하였다.

## [ PART 2 ] 문서 분류 및 군집화

### 1. R2-1: 영어 신문 기사 분류

The New York Times의 영어 신문 기사를 opinion, business, world, us, arts, sports, books, movies 총 8개의 카테로리로 분류하고자 한다. train set에는 각 카테고리 별로 300개의 기사가 제공 되어 있으며, 이 데이터를 활용하여 학습을 시킨 후, 추후에 기사 30개에 대하여 각 카테고리 별로 분류하는 것을 목표로 한다.

#### 1) 데이터 전처리 과정

① 특수 문자 제거 (특수 문자는 중요한 정보가 없는 경우가 많기 때문에 제거)
② 길이가 3이하인 단어는 제거 (영어의 경우, 길이가 짧은 단어는 대부분 불용어에 해당하기 때문)
③ 전체 단어에 대한 소문자 변환 (대문자는 문장의 맨 앞과 같은 특정 상황에서만 쓰이고, 대부분의 글은 소문자로 작성되기 때문)

```python
import pandas as pd
import nltk

def preProcess(documents):
    news_df = pd.DataFrame({'document':documents})
    # 특수 문자 제거
    news_df['clean_doc'] = news_df['document'].str.replace("[^a-zA-Z]", " ")
    # 길이가 3이하인 단어는 제거 (길이가 짧은 단어 제거)
    news_df['clean_doc'] = news_df['clean_doc'].apply(lambda x: ' '.join([w for w in x.split() if len(w)>3]))
    # 전체 단어에 대한 소문자 변환
    news_df['clean_doc'] = news_df['clean_doc'].apply(lambda x: x.lower())

    return news_df['clean_doc']
```

<코드 2-1> 데이터 전처리 코드

### 2) Naïve Bayes Classifier

파이프라인 생성 과정:

데이터에서 영어의 기본적인 stop_words를 제거하여 벡터화 하였고, TF.IDF 가중치를 부여하였다. Multinomial Naive Bayes classifier에서 alpha값은 0.01 ~ 0.5의 값을 0.01 단위로 테스트를 한 후 적합한 파라미터를 적용하였다.

```python
# TODO - 2-1-1. Build pipeline for Naive Bayes Classifier
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.feature_extraction.text import TfidfTransformer
from sklearn.naive_bayes import MultinomialNB
clf_nb = Pipeline([('vect', CountVectorizer(stop_words = 'english')),
                   ('tfidf', TfidfTransformer(use_idf=True)),
                   ('clf', MultinomialNB(alpha=0.05))])
clf_nb.fit(preProcess(train_data.data), train_data.target)
docs_test = test_data.data

predicted = clf_nb.predict(docs_test)
print("NB accuracy : %d / %d" % (np.sum(predicted==test_data.target), len(test_data.target)))
print(metrics.classification_report(test_data.target, predicted, target_names=test_data.target_names))
print(metrics.confusion_matrix(test_data.target, predicted))
```

<코드 2-2> Multinomial Naive Bayes classifier 사용한 코드 작성

데이터를 train시킨 후, test 해본 결과 아래와 같은 결과를 얻을 수 있었다. 44개의 test 데이터 중에서 32개를 정확하게 분류하였으며, 73%의 정확도를 기록하였다. 본 코드는 books와 opinion 카테고리 분류에 있어서 약세를 보였다.

```
NB accuracy : 32 / 44
              precision    recall  f1-score   support

        arts       1.00      0.80      0.89         5
       books       0.50      0.83      0.62         6
    business       0.71      1.00      0.83         5
      movies       0.75      0.50      0.60         6
     opinion       0.50      0.20      0.29         5
      sports       1.00      0.80      0.89         5
          us       0.83      0.83      0.83         6
       world       0.71      0.83      0.77         6

    accuracy                           0.73        44
   macro avg       0.75      0.73      0.72        44
weighted avg       0.75      0.73      0.71        44

[[4 1 0 0 0 0 0 0]
 [0 5 0 0 1 0 0 0]
 [0 0 5 0 0 0 0 0]
 [0 3 0 3 0 0 0 0]
 [0 0 2 0 1 0 1 1]
 [0 1 0 0 0 4 0 0]
 [0 0 0 0 0 0 5 1]
 [0 0 0 1 0 0 0 5]]
```

<그림 2-1> Multinomial Naive Bayes classifier 결과

### 3) SVM Classifier

파이프라인 생성 과정:

데이터에서 영어의 기본적인 stop_words를 제거하여 벡터화 하였고, TF.IDF 가중치를 부여하였다. 이번에는 SVM을 활용하였고, C = 1, gamma = 1, kernel = 'linear'로 설정을 하였다. C는 [0.1, 0.5, 1, 2, 5, 10, 100, 1000], gamma는 [1, 0.75, 0.5, 0.25, 0.1, 0.01, 0.001, 0.0001], kernel은 ['rbf', 'linear', 'poly']로 grid_search하여 최적의 파라미터 값을 찾았다.

```python
from sklearn.model_selection import GridSearchCV
param_grid = {'C': [0.1, 0.5, 1, 2, 5, 10, 100, 1000],
              'gamma': [1, 0.75, 0.5, 0.25, 0.1, 0.01, 0.001, 0.0001],
              'kernel': ['rbf', 'linear', 'poly']}

grid = GridSearchCV(SVC(), param_grid, refit = True, verbose = 3)
```

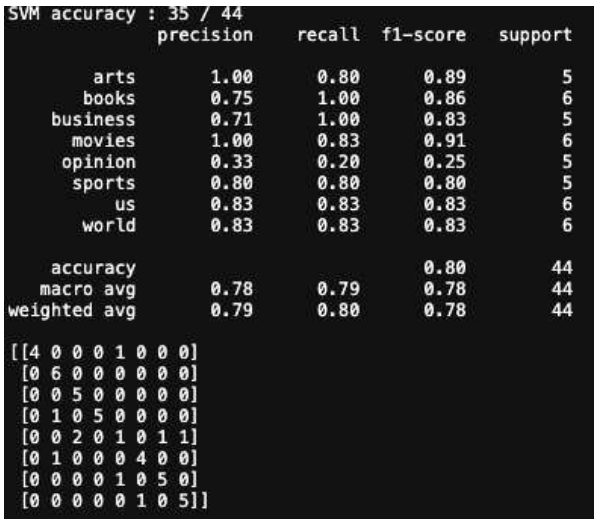
<코드 2-3> GridSearch 사용하여 최적 parameter 찾기

```python
# TODO - 2-1-2. Build pipeline for SVM Classifier
from sklearn.svm import SVC
from sklearn.svm import LinearSVC
clf_svm = Pipeline([('vect', CountVectorizer(stop_words='english')),
                    ('tfidf', TfidfTransformer()),
                    ('clf', SVC(C = 1, gamma = 1, kernel = 'linear'))])
clf_svm.fit(preProcess(train_data.data), train_data.target)

predicted = clf_svm.predict(docs_test)
print("SVM accuracy : %d / %d" % (np.sum(predicted==test_data.target), len(test_data.target)))
print(metrics.classification_report(test_data.target, predicted, target_names=test_data.target_names))
print(metrics.confusion_matrix(test_data.target, predicted))
```

<코드 2-4> SVM Classifier 사용한 코드 작성

데이터를 train시킨 후, test 해본 결과 아래와 같은 결과를 얻을 수 있었다. 44개의 test 데이터 중에서 35개를 정확하게 분류하였으며, 80%의 정확도를 기록하였다. SVM classifier는 Multinomial Naive Bayes classifier보다 정확도가 높았다.



```
SVM accuracy : 35 / 44
              precision    recall  f1-score   support

        arts       1.00      0.80      0.89         5
       books       0.75      1.00      0.86         6
    business       0.71      1.00      0.83         5
      movies       1.00      0.83      0.91         6
     opinion       0.33      0.20      0.25         5
      sports       0.80      0.80      0.80         5
          us       0.83      0.83      0.83         6
       world       0.83      0.83      0.83         6

    accuracy                           0.80        44
   macro avg       0.78      0.79      0.78        44
weighted avg       0.79      0.80      0.78        44

[[4 0 0 0 1 0 0 0]
 [0 6 0 0 0 0 0 0]
 [0 0 5 0 0 0 0 0]
 [0 1 0 5 0 0 0 0]
 [0 0 2 0 1 0 1 1]
 [0 1 0 0 0 4 0 0]
 [0 0 0 0 1 0 5 0]
 [0 0 0 0 0 1 0 5]]
```

<그림 2-2> SVM Classifier 결과

### 2. R2-2: 영어 신문 기사 군집화

본 part 2-2 는 영어 신문 기사를 군집화하는 것을 목적으로 하며, 텍스트 전처리 혹은 parameter 조정 등을 통해 그 군집화 성능을 향상시키고자 한다.

Requirements 는 아래 <표 3-1>과 같다.

| PART | Requirements |
|---|---|
| R2-2 | - Algorithm : K-means Clustering<br>- data : 'text_all' folder<br>- 평가 : V-measure<br>- label : arts, books, business, movies, opinion, sports, us, world |

<표 3-1>

#### 1) Variable 설정

성능 향상을 위해 설정한 variable의 종류는 아래 <표 3-2>와 같다.

| Index | variables | 과정 |
|---|---|---|
| V1 | Title weight | Vectorize 이전 Preprocessing |
| V2 | Tokenizer | |
| V3 | Vectorizer parameter | Vectorize >> Clustering |
| V4 | TfidfTransformer parameter | |
| V5 | Kmeans parameter | |

<표 3-2>

Variable을 설정하는 코드 내용은 아래 <코드 3-1>, <코드 3-2>와 같다.

```
dataframe2['total'] = (dataframe2['title'] + ' ')*2 + ' ' + dataframe2['body'] ## V1 title weight

# nomalization (tokenize, lemmatize)
w_tokenizer = nltk.tokenize.WhitespaceTokenizer() ## V2 NLTKWordTokenizer(), TreebankWordTokenizer()
lemmatizer = nltk.stem.WordNetLemmatizer()
```

<코드 3-1> V1, V2

```
# vectorize
vectorizer = CountVectorizer(stop_words='english',  analyzer='word', min_df=9, max_df=0.8) ## V3 / CountVectorizer, TfidfVectorizer / ngram_range=, max_feature=,
data_trans1 = vectorizer.fit_transform((dataframe2['le_st']))
data_trans2 = TfidfTransformer(smooth_idf=False).fit_transform(data_trans1) ## V4 / norm=, use_idf=, sublinear_tf=

# cluster
clst = KMeans(n_clusters=8, random_state=10) ## V5 / init=, n_init=, max_iter=, precompute_distance=, algorithm=
clst.fit(data_trans2)

print(metrics.v_measure_score(data.target, clst.labels_))
```

<코드 3-2> V3, V4, V5

**2) 성능 향상 요인**

**(1) Text-Preprocessing (V1, V2)**

- Vectorize 이전에 text의 전처리 과정을 미리 진행하고, 그 결과값을 vectorize 하였다.
- 전처리 과정은 다음 <표 3-3> 과 같은 순서로 진행하였다.

| 순서 | 내용 | 세부내용 |
|---|---|---|
| ① | Dataframe | - load한 파일을 dataframe으로 생성함<br>- label과 data를 column으로 부름 |
| ② | Text Lower | - data를 모두 소문자로 전환함 |
| ③ | Title Weight | - data 중 title에 가중치를 부여함 |
| ④ | Token > P1,P2 > lemma | - tokenize 후 P1이나 P2를 거쳐 lemmatize함<br>(P1, P2는 아래 <표3-4>에 서술함) |
| ⑤ | Stopwords | - NLTK 패키지의 stopwords를 이용함 |

<표 3-3>

- 위 ④ 는 <표 3-3>과 같이 크게 두 가지로 구분하여 진행하였다.

| Index | Preprocessing | 결과 |
|---|---|---|
| P1 | Tokenize >> Pos_tagging >> Lemmatize | 비교적 성능 우수 |
| P2 | Tokenize >> Stemming >> Lemmatize | |

<표 3-4>

- 성능 향상 실험과정에서 여러 stemmer를 사용해보았음에도, P1의 방식이 더 좋은 성능을 보였기에 본 보고서에서는 아래부터 P1에 초점을 맞춰 서술하도록 한다.
- P1과 P2를 활용한 전처리 코드는 아래 <코드 3-3>과 <코드 3-4>와 같다.

```python
# nomalization (tokenize, lemmatize)
w_tokenizer = nltk.tokenize.WhitespaceTokenizer()
lemmatizer = nltk.stem.WordNetLemmatizer()

def get_wordnet_pos(treebank_tag):
    if treebank_tag.startswith('J'):
        return wordnet.ADJ
    elif treebank_tag.startswith('V'):
        return wordnet.VERB
    elif treebank_tag.startswith('N'):
        return wordnet.NOUN
    elif treebank_tag.startswith('R'):
        return wordnet.ADV
    else:
        return wordnet.NOUN

def pos_tag(text):
    pos_tokens1 = [nltk.pos_tag(w_tokenizer.tokenize(text))]
    pos_tokens2 = [[(lemmatizer.lemmatize(word, get_wordnet_pos(pos_tag))) for (word, pos_tag) in pos] for pos in pos_tokens1]
    return pos_tokens2[0]

dataframe2['lemma'] = dataframe2.total.apply(pos_tag)
```

<코드 3-3> P1 코드

```python
# nomalization (tokenize, stemmize, lemmatize)
w_tokenizer = nltk.tokenize.WhitespaceTokenizer()
w_stemmer = nltk.stem.SnowballStemmer('english')
lemmatizer = nltk.stem.WordNetLemmatizer()

def stemmize_text(text):
    return [w_stemmer.stem(w) for w in w_tokenizer.tokenize(text)]

def lemmatize_text(text):
    return [lemmatizer.lemmatize(w) for w in stemmize_text(text)]

dataframe2['lemma'] = dataframe2.total.apply(lemmatize_text)
```

<코드 3-4> P2 코드

(2) Parameter (V3, V4, V5)

- V3, V4, V5의 경우 아래 <표 3-5>의 parameter를 조정하면서 실험하였다.

| Index | parameter | 비고 |
|---|---|---|
| V3 | Vectorizer종류, Stop_words, analyzer, min_df, max_df, ngram_range, max_feature | 7개 |
| V4 | norm, use_idf, smooth_idf, sublinear | 4개 |
| V5 | random_state, init, n_init, max_iter, algorithm, precompute_distance, | 6개 |

<표 3-5>

3) 성능 향상 실험 과정

(1) Base

- 기존에 주어진 코드를 실행하였을 때 성능은 <그림 3-1>와 같다.

```
0.2065207122753286
```

<그림 3-1>

(2) Countvectorizer 기본 설정

- stop_words를 영어에 대해 지정하였다.
- analyzer를 'char', 'char_wb'로 변경해보았으나 default값인 'word'가 가장 좋은 성능을

보여, 기존대로 word 단위로 analyzer가 작동하도록 하였다.
- (2)의 결과, 성능은 <그림 3-2>과 같다.


<그림 3-2>

**(3) Countvectorizer, Kmeans 일부 parameter 변경**

- 실험에서 변경한 parameter와 범위는 아래 <표 3-6>과 같고, 그 코드는 <코드 3-5>이다.
- 총 11*10*15 = 1650 가지의 case를 실험하였다.

| 함수 | parameter | 범위 | 개수 |
|---|---|---|---|
| Countvectorizer | min_df | range(1, 12) | 11 |
| | max_df | np.arange(0.5, 1, 0.05) | 10 |
| Kmeans | random_state | range(0, 15) | 15 |

<표 3-6>

```
# vectorize
for k in range(1, 12):
    for m in list(np.arange(0.5, 1.0, 0.05)):
        for n in range(0, 15):

            V_list = [(k,m,n)]

            for a, b, c in V_list:
                vectorizer = CountVectorizer(stop_words='english',  analyzer='word', min_df=a, max_df=b) ## V3
                data_trans1 = vectorizer.fit_transform((dataframe2['le_st']))
                data_trans2 = TfidfTransformer().fit_transform(data_trans1) ## V4

                # cluster
                clst = KMeans(n_clusters=8, random_state=c) ## V5
                clst.fit(data_trans2)

                print("end %d %f %d" % (a, b, c))
                print(metrics.v_measure_score(data.target, clst.labels_))
```

<코드 3-5>

- 실험 결과, 가장 좋은 성능을 가진 case는 <표 3-7> 이다.

| 함수 | parameter | 범위 내 최적값 |
|---|---|---|
| Countvectorizer | min_df | 9 |
| | max_df | 모든 경우 default(1.0)와 동일 |
| Kmeans | random_state | 10 |

<표 3-7>

**(4) Countvectorizer, Kmeans parameters 추가 변경**

- 위 (3)의 결과값을 고정하고, 다른 parameters를 조정하였다.
- 실험에서 변경한 parameter와 범위는 아래 <표 3-8>과 같고, 그 코드는 <코드 3-6>이다.
- 총 25*9*10 = 2250 가지의 case를 실험하였다.

| 함수 | parameter | 범위 | 개수 |
|---|---|---|---|
| Countvectorizer | max_features | range(500, 1000, 20) | 25 |
| Kmeans | n_init | range(10, 100, 10) | 9 |
| | max_iter | range(300, 800, 50) | 10 |

<표 3-8>

```
    # vectorize
    for k in range(500, 1000, 20):
        for m in range(10, 100, 10):
            for n in range(300, 800, 100):
                V_list = [(k,m,n)]

                for a, b, c in V_list:
                    vectorizer = CountVectorizer(stop_words='english',  analyzer='word', min_df=9, max_df=0.8, max_features=a) ## V3
                    data_trans1 = vectorizer.fit_transform((dataframe2['le_st']))
                    data_trans2 = TfidfTransformer().fit_transform(data_trans1) ## V4

                    # cluster
                    clst = KMeans(n_clusters=8, random_state=10, n_init=b, max_iter=c) ## V5
                    clst.fit(data_trans2)

                    print("end %d, %d, %d" % (a, b, c))
                    print(metrics.v_measure_score(data.target, clst.labels_))
    print("end %d" % (i))
```

<코드 3-6>

- 실험 결과, 위 세 parameters는 default값이 가장 성능이 높았다.
- (3), (4)의 결과를 바탕으로 clustering을 진행하면 최종 성능은 <그림 3-3>과 같다.

```
0.40513428314145183
```

<그림 3-3>

**(5) Text-Preprocessing 과정 추가**

- vectorize 되기 이전 문서를 전처리하는 과정을 추가하였다.
- 전처리 과정은 상술했던 2)-(1)의 내용과 같다.

**(6) Title weight 값 조정**

- title에 속하는 단어들을 반복시키는 횟수를 변경해가며 실험하였다.
- 반복 횟수의 범위는 range(1, 4) 이며, (2)~(5) 과정을 각 weight 값에 대하여 적용하였다.
- 실험 코드는 <코드 3-7>과 같다.

```
for i in range(1,4):
    # dataframe, lower
    dataframe = pd.DataFrame(data.data, data.target)
    dataframe.columns=['text']

    dataframe['text'] = dataframe['text'].str.lower()

    # title split & weight
    dataframe['text'] = dataframe['text'].str.split('\n')
    dataframe2 = dataframe['text'].apply(pd.Series)

    dataframe2.columns=['title', 'body', 'NAN']
    del dataframe2['NAN']

    dataframe2['total'] = (dataframe2['title'] + ' ')*i + ' ' + dataframe2['body'] ## V1 title weight
```

<코드 3-7>

- 실험 결과, 반복 횟수 2회가 범위 내 최적값이다.
- (6)의 결과를 적용했을 때 성능은 <그림 3-4>와 같다.

```
0.48766260941645745
```

<그림 3-4>

**(7) 기타 parameter 조정**

- 이전까지의 과정에서 변경하지 않았던 parameters는 <표 3-9>와 같다.

| Index | parameter | 비고 |
|---|---|---|
| V2 | Tokenizer종류 | 1개 |
| V3 | Vectorizer종류, ngram_range | 2개 |
| V4 | norm, use_idf, smooth_idf, sublinear | 4개 |
| V5 | init, algorithm, precompute_distance, | 3개 |

<표 3-9>

- NLTKWordTokenizer를 사용해본 결과, Whitespacetokenizer가 더 좋은 성능을 보였다.
- TfidfVectorizer를 사용해본 결과, CountVectorizer가 더 좋은 성능을 보였다.
- ngram_range를 (1,2), (2,2)로 설정하였으나 default가 더 좋은 성능을 보였다.
- norm, use_idf, sublinear를 각각 default와는 반대값으로 설정해보았으나 default가 더 좋은 성능을 보였다.
- smooth_idf를 False로 설정하자 성능이 향상되었다.
- init를 'random'으로 설정하거나 algorithm을 'full'로 설정하자 성능이 줄어들었다.
- precompute_distance는 변경하여도 성능이 바뀌지 않았다.
- 결과적으로, 최종 성능은 <그림 3-5>와 같다.

```
0.4889165181679361
```

<그림 3-5>

**4) 최종 코드**

- 최종적인 코드는 아래 <코드 3-8>, <그림 3-9>이다.

```python
# TODO - Data preprocessing and clustering
# dataframe, lower
dataframe = pd.DataFrame(data.data, data.target)
dataframe.columns=['text']

dataframe['text'] = dataframe['text'].str.lower()

# title split & weight
dataframe['text'] = dataframe['text'].str.split('\n')
dataframe2 = dataframe['text'].apply(pd.Series)

dataframe2.columns=['title', 'body', 'NAN']
del dataframe2['NAN']

dataframe2['total'] = (dataframe2['title'] + ' ')*2 + ' ' + dataframe2['body']

# nomalization (tokenize, lemmatize)
w_tokenizer = nltk.tokenize.WhitespaceTokenizer()
lemmatizer = nltk.stem.WordNetLemmatizer()

def get_wordnet_pos(treebank_tag):
    if treebank_tag.startswith('J'):
        return wordnet.ADJ
    elif treebank_tag.startswith('V'):
        return wordnet.VERB
    elif treebank_tag.startswith('N'):
        return wordnet.NOUN
    elif treebank_tag.startswith('R'):
        return wordnet.ADV
    else:
        return wordnet.NOUN

def pos_tag(text):
    pos_tokens1 = [nltk.pos_tag(w_tokenizer.tokenize(text))]
    pos_tokens2 = [[(lemmatizer.lemmatize(word, get_wordnet_pos(pos_tag))) for (word, pos_tag) in pos] for pos in pos_tokens1]
    return pos_tokens2[0]

dataframe2['lemma'] = dataframe2.total.apply(pos_tag)

# stopwords remove
sw = stopwords.words('english')
dataframe2['le_st'] = dataframe2['lemma'].apply(lambda x: ' '.join([word for word in x if word not in sw]))
```

<코드 3-8>

```python
# vectorize
vectorizer = CountVectorizer(stop_words='english',  analyzer='word', min_df=9, max_df=0.8)
data_trans1 = vectorizer.fit_transform((dataframe2['le_st']))
data_trans2 = TfidfTransformer(smooth_idf=False).fit_transform(data_trans1)

# cluster
clst = KMeans(n_clusters=8, random_state=10)
clst.fit(data_trans2)

print(metrics.v_measure_score(data.target, clst.labels_))
```

<코드 3-9>

**5) 분석**

**(1) 성능 향상 요인**

- 실험을 진행하는 과정에서 나타나는 성능의 변화는 <표 3-10>과 같다.

| Timeline | 추가 또는 수정된 variables/parameter | 성능 |
|---|---|---|
| 0 | (Base) | 0.20652 |
| 1 | V3 : Stopwords = 'english' | 0.39794 |
| 2 | V3 : min_df = 9<br>V5 : random_state = 10 | 0.40513 |
| 3 | 추가적인 Preprocessing<br>V1 : title*2 (weight) | 0.48766 |
| 4 | V4 : smooth_idf = False | 0.48891 |

<표 3-10>

- CountVectorizer 자체에 내장된 stop-word-list를 사용하는 것만으로도 성능이 두 배가량 향상된다. 따라서 Stopword를 활성화하지 않았을 때는 너무 흔하거나 불필요한 단어들로 인해 결과가 왜곡되어 나타났었음을 추측할 수 있다.
- CountVectorizer의 또 다른 parameter 중 유효한 변화를 가져오는 것은 min_df 로, 1씩만 변화시켜도 성능이 예민하게 달라지는 것을 볼 수 있었다. 반면에 max_df는 0.5에서 1까지 계속해서 큰 변화를 주었음에도 불구하고 성능에 전혀 영향을 주지 못하였다. 즉, 본 R2-2에서는 빈도가 높은 단어들보다는 빈도가 낮은 단어들로 인해 clustering 결과가 민감하게 변동했음을 알 수 있다.
- Clustering 초기의 center를 지정해주는 random_state 값에 따라서도 성능이 비교적 크게 좌우되었다. 이는 각 cluster의 처음 center를 어디에 잡느냐에 따라 Clustering 결과값이 크게 달라질 수 있음을 시사한다.
- Text-Preprocessing을 Vectorization 이전에 추가한 것은 10%p 정도의 성능 향상을 가져왔다. CountVectorization은 stemming이나 lemmatize 기능을 포함하고 있지 않기 때문에 추가적인 preprocessing이 상당한 효과를 가져올 수 있었던 것으로 보인다. 또한, CountVectorization에 내장된 stopwords를 적용하기 전에 NLTK의 stopwords 함수를 활용하여 이중으로 불필요한 단어를 제거한 것도 text clustering에 긍정적인 영향을 준 것으로 생각된다.
- data text 내부에서 title과 body를 구분하고, title을 반복하여 text에 재삽입한 과정도 성능 향상에 기여하였다. 이 때 반복 횟수를 조금씩만 변화시켰음에도 불구하고 성능이 민감하게 변동된 점을 보아 weight의 크기가 clustering 결과에 상당한 영향을 주는 것으로 파악된다. 또한, 결과적으로 2회 반복이 가장 큰 성능을 가져왔는데, 실험해본 결과 이보다 weight가 커질수록 성능이 급격하게 저하되는 것을 관찰할 수 있었다. 이는 특정 단어나 구문에 대해 가중치를 부여하는 것이 적정선을 벗어난다면 clustering 결과를 크게 왜곡할 위험성을 가진다는 점을 보인다.
- 큰 변화는 아니지만 smoothing 과정을 거치지 않는 것이 R2-2에서는 성능 향상을 가져왔다. Smoothing을 통해 확률을 조정하는 과정에서 clustering에 미세한 오류를 발생시켰던 것으로 추정된다.
- 추가적으로, P1과 P2를 비교하여 실험을 진행하면서 Lemmatizer와 병행하여 진행하는 작업에는 stemming보다 pos_tagging이 적절함을 확인할 수 있었다.

**(2) 실험 결과1 - 각 Label의 Top-10 Keywords**

```
keywords:
 ['israel', 'israeli', 'gaza', 'palestinian', 'hamas', 'netanyahu', 'rocket', 'jerusalem', 'arab', 'minister']
```

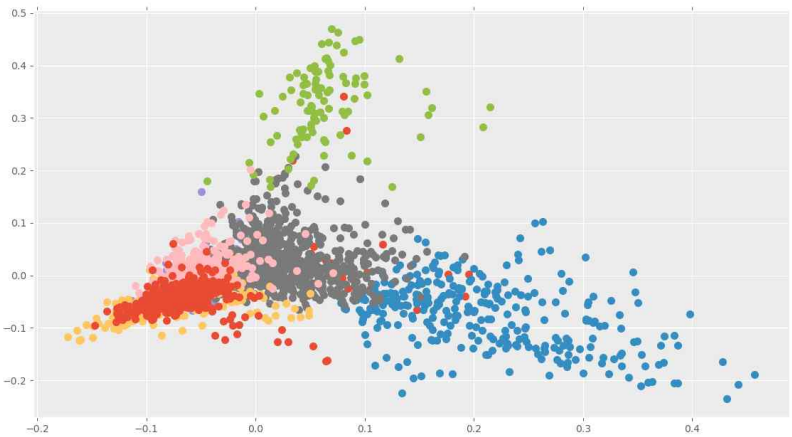
```
keywords:
 ['trump', 'biden', 'president', 'mr', 'republican', 'senate', 'vote', 'election', 'party', 'capitol']
```

```
keywords:
 ['art', 'music', 'museum', 'artist', 'new', 'york', 'work', 'song', 'gallery', 'album']
```

<그림 3-6> Top-10 Keywords 일부

- 각 cluster에서의 keywords를 출력해본 결과, 대략적으로 비슷한 유형의 키워드끼리 묶인 것을 확인할 수 있다.
- 그러나 여전히 'israeli'나 'mr' 과 같이 알 수 없는 단어들이 존재하며, 'new'와 같이 카테고리와 크게 상관없는 단어도 keyword에 들어가 있음을 알 수 있다.
- 또한 'israel'과 'trump'는 "world"라는 카테고리에 함께 있는 것이 clustering의 성능을 더 높일 것으로 보이지만 다른 cluster에 속해 있다.
- 위와 같은 한계점으로 성능이 0.5 미만에 그친 것으로 파악된다.

**(3) 실험 결과2 - 시각화**




<그림 3-7>

- clustering 결과값을 시각화해본 결과, 5개 정도의 cluster는 잘 나뉘어 있으나 좌측 하부 부분 근방은 여러 cluster들이 모여있는 것을 확인할 수 있다.
- 기존에 학습된 모델을 이용하는 등의 방법을 통해 Clustering의 성능을 높혀 각 군집을 차별화한다면, 각 cluster center들 사이가 벌어진 시각화 결과가 나올 것으로 예상된다.

---

[Part 1 참고 문헌]

[1] 박용민; 김보겸; 이재성. 질문 특성을 고려한 커뮤니티 질의응답 시스템 (cQA) 자질 추출 방법. 제 26 회 한글 및 한국어 정보처리 학술대회, 2014, 119-121.

[2] Kamphuis C., de Vries A.P., Boytsov L., Lin J. (2020) Which BM25 Do You Mean? A Large-Scale Reproducibility Study of Scoring Variants. In: Jose J. et al. (eds) Advances in Information Retrieval. ECIR 2020. Lecture Notes in Computer Science, vol 12036. Springer, Cham.