Data Management and Analysis Project3

Team03 오태양, 김해찬, 하창우, 김민영

I. INTRODUCTION

II. PART I: DOCUMENT SEARCH ENGINE

A. Check Document.txt

검색 엔진 구현에 앞서, document들이 어떻게 형성되어 있는지 확인했다. document들은 약 1000자 이내이자 8줄 이하의 영문 문장으로 구성되어 있으며, 소문자로 작성되어 있다. document 끝에는 /를 사용하여 document간을 구별했다. 모든 document의 첫 문장은 title로 확인되었다. 모든 문장은 마침표(.)로 종결되어 있으며, 중간 중간 다른 특수문자 또한 확인할 수 있었다. 숫자 또한 본문에 일부 포함되어 있었으며, 일반적인 내용도 있지만 상당수는 전문용어를 사용한 전문지식에 대한 내용으로 구성되어 있었다.

B. Check query.txt

모든 쿼리는 한문장으로 구성되어있으며, 대부분의 경우 의문문으로 물음표로 문장을 마쳤다. document와 마찬가지로 /를 사용하여 쿼리 간을 구별했다.

C. make_index.py - data processing

검색엔진 구현을 위해 document indexing에서 시도한 것들은 다음과 같다. Stemming, Lemmatizing, Stopword remove, Special characeter remove, spell check, Ngram. 각각의 시도는 또 다양한 방식을 통해 구현해 보았고그 중 가장 성능이 좋은 것으로 채택했다. Ngram의 경우 document에서만 구현한 것이 아니라, 쿼리와 같이 이루어졌기 때문에 Query part에서 종합적으로 설명하겠다.

1. Stemming Lemmatizing

document word들을 stemming, lemmatizing 해주기위해 여러가지 방법을 시도했다.

```
if get_wordnet_pos(B[1])==None: word=Lstem.stem(B[0])
```

```
else: word=lemma.lemmatize(B[0],get_wordnet_pos(B[1]))
```

먼저 pos_tag를 사용해 추출한 품사를 사용하여 lemmatizing해주는 것이 가장 정확할 것이라 판단해, 위와같이 주된 품사들에 대해서는 lemmatizing해주고 예외들에 대해서 PorterStemmer, LancasterStemmer를 사용해 그 결과들을 비교했지만 오히려 성능이 감소하는 결과가 나타났다.

```
schema = Schema(docID=NUMERIC(stored=True),contents=TEXT(analyzer=
    StemmingAnalyzer()))
```

이에 따라 처음 스키마를 형성할때에 StemmingAnalyzer를 설정하여 stemming하는 방법을 시도했고, 성능향상을 확인하여 채택했다. 이 코드는 추후 queryresult파일에서 parser를 정의할때 ix.schema에서 형식을 따오기 때문에 자동으로 query의 stemming까지 구현됨을 확인했다.

2. Remove Stopword

```
schema = Schema(docID=NUMERIC(stored=True),contents=TEXT(analyzer=
    StemmingAnalyzer(stoplist=STOP_WORDS)))
```

query의 기본코드와 같이 stopword를 제거해보았으나 효과가 뚜렷하지 않아 위와 같이 StemmingAnalyzer 내에 stoplist를 지정해주어 stopwords를 제거해주었다. 이 때 stopwrod list 별로 효과가 조금씩 달라져서, whoosh와 nltk.corpus에 내장된 stopword list를 둘다 시도해본 결과 whoosh에 내장된 list를 사용했을 때 결과가 가장 좋아 이를 채택했다.

3. Remove Special Character

본문에 사용된 특수문자가 검색엔진의 성능에 부정적인 영향을 끼치는지 확인하기위해 위와같이 특수문자 혹은 Enter를 제거하고자 했다.

```
# Change special character  to whitespace
doc_text=doc_text.replace(".", " ")
```

이를 같이 구현하였는데, 마침표를 제외하고 효과가 없거나 오히려 제거했을때 성능이 감소하였다. 하이픈의 경우 앞뒤의 단어를 묶어 tokenize해주거나 분리하는 시도를 모두해보았고 앞뒤의 단어가 더 중요할 것이라 판단하고 가중치를 주었으나 효과가 없어 모두 기각했다. 결론적으로 마침표만 제거해주는 모델을 채택했다.

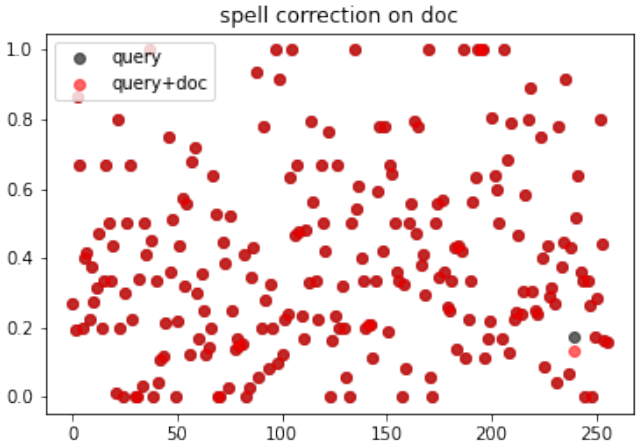

FIG. 1: The value indicates BPREF score and x is query id.

4. *document spell check*

```
schema = Schema(docID=NUMERIC(stored=True), contents=TEXT(analyzer=
    StemmingAnalyzer(stoplist=STOP_WORDS), spelling=True))
```

위와 같이 document 속 오탈자를 보정하는 코드를 구현하여 검색엔진 성능을 비교하였다. Fig 1에서 보면 오탈자를 보정한 결과, 특정 query 를 제외하고는 score 값의 변화가 없었으며, 특정 query 에 대해서 score 가 하락하였다.

D. QueryResult.py 구현

검색엔진 구현을 위해 쿼리 processing에 시도한 방식은 다음과 같다. stopword special character remove, stemming lemmatizing, give weightness, spell check, ngram. 다양한 방식으로 이들을 구현, 비교해 가장 성능이 좋은 모델을 채택했다.

1. *Remove Stopword*

Stopword를 제거할때, 어떤 stopword list를 사용하여 제거할 것인지 비교했다. whoosh와 nltk.corpus에 내장된 stopword를 사용해 비교했다.

```
for word in q.split(' '):
    if word.lower() not in stopWords: new_q += word + ' '
    # Remove stopWords
```

이 때, 기본코드에 주어진 대로 nltk.corpus에 내장된 stopword list를 사용했을 때 가장 성능이 좋아 채택했다.

2. Stemming Lemmatizing

make_index.py에서와 같이 word 별로 *pos_tag*를 사용해 품사별로 다양한 Stemmizer & Lemmatizer를 시도했지만 성능이 하락해 기각했다. 이는 처음 스키마생성에 StemAnalyzer를 사용해서 서칭할 수 있게 하는 알고리즘의 성능이 각각 stemming해주는 것 대비 성능이 좋기 때문으로 보인다. 다만 stopword의 경우 Analyzer에서 설정해주었음에도 기본코드에서 추가적으로 제거했을 때 효과가 좋았기 때문에 유지했다.

3. Remove Special Character

쿼리가 주어졌을 때, 쿼리의 내용이 중요하며 특수문자는 중요하지 않다고 판단하였다. 특히 쿼리 특성상 ? 와 같은 질문형태인 것이 많을 것으로 예상하여 이를 제거하는 코드를 작성하여 검색엔진을 구현하였으며 성능이 유의미하게 향상되는 것을 확인했다. '?' 외에도 특수문자가 검색에 미치는 영향을 알아보기 위해 쿼리에 존재하는 특수문자를 제거하며 그 성능을 비교했다.

```
q=q.replace("?", " ")
q=q.replace(",", " ")
q=q.replace(".", " ")
```

그 결과 하이픈은 제거했을 때 오히려 성능이 떨어지는 결과를 보였고 몇몇은 변화가 없었다. 이에 따라 위의 3개만 제거하는 방식을 채택했다.

4. Give Weight to Token

더 중요한 word 들에 대해서 가중치를 주고, 상대적으로 중요하지않은 word들에 대해선 가중치를 줄이는 방식을 구현하고자 했다. 이 때 중요한 것은 그 기준을 설정하는 것이었다. 여러 서칭을 통해 최종적으로 품사와 특수한 위치의 단어들, 그리고 서로다른단어들에 대해서 가중치를 주고자 했다.
먼저 단어의 품사에 따라 가중치를 주어 비교했다. 가중치는 처음 1.02와 0.98을 시도하여 증가시킬지 감소시킬지 판단하고 0.1 단위로 변화시킨후, 최대 성능을 구현하는 값을 찾고 다시한번 0.01단위로 세부조정했다. 이 때 0.001단위의 변화는 검색엔진 성능에 영향이 없음을 확인했다.

```
if B[1]=="MD":word="" # best value : 0.0
elif B[1]=="NN":word=word+"^1.08" # best value : 1.08
elif B[1]=="JJ":word=word # best value : 1
elif B[1]=="RB":word=word+"^0.78" # best value : 0.78
elif B[1]=="VB":word=word # best value : 1
else: word=word+"^0.76" # best value : 0.76
```

그 결과 구현한 best model은 위와 같다. 'modal'word의 경우 활용되지 않는 부정사 동사로 can, should와 같은 것들을 포함하는데 이 word는 아예 제거했을 때 가장 효과가 좋았다. 명사들의 경우 1.08의 가중치를 주었고, 부사의 경우 가중치를 0.78로 낮추었다. 동사와 형용사는 가중치를 유지시키고 그 외의 모든 단어의 가중치를 0.76으로 낮춰주었을 때 가장 성능이 좋았다.

두 번째로, stopword 사이에 있는 단어에 가중치를 주어 비교했다. 이를 구현한 결과, 오히려 성능이 감소했는데 이는 stopword 사이의 단어들을 ngram처럼 구현하기 위해 편법으로 word들을 붙여버려서라고 보였고 이에 따라 이 모델은 기각했다.

세번째로 TA Session에서와 같이 쿼리 중 한 단어의 높은 빈도수보단 여러 단어가 포함되어 있는 경우에 더 가중하여 비교하는 코드를 아래와 같이 구현하였다.

```
parser = QueryParser("contents", schema=ix.schema, group=OrGroup.factory
    (0.9))
```

하지만 성능향상이 없었으며, 오히려 하락하는 결과를 보여주어 제거하였다. 보통의 경우 성능이 향상되지만 이 상황에서는 문서의 길이, 개수, 쿼리 개수 등이 작은 상황이기에 이러한 결과를 반환한 것이라 결론내렸다.

마지막으로 쿼리에 괄호가 포함되어 있다면 괄호 안의 단어는 특히 더 중요할 수 있을 것이라 생각해, 괄호 속 마지막 단어에 가중치를 부여해보았다.

```
# Give Weight to Word in ()
q=q.replace(")", "^1.2") # best value : 1.2
```

그 결과 성능이 향상되어 채택했다.

5. Query spell check

```
correct = searcher.correct_query(query,new_q)
```

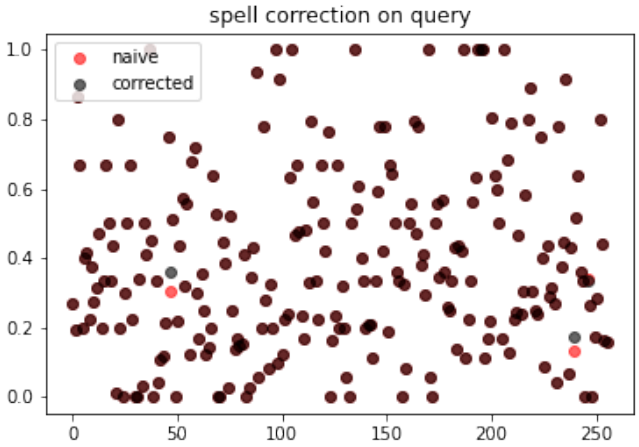

FIG. 2: The value indicates BPREF score and x is query id.

```
results = searcher.search(correct.query, limit=None)
```

위와 같이 쿼리 속 오탈자를 보정하는 코드를 구현하여 검색엔진 성능을 비교한 결과 성능이 향상되었다. Fig 2에서 보면 오탈자를 고쳐주고 오탈자를 보정한 회색 점이 기존의 빨간 점보다 서칭 성능이 더 좋음을 확인할 수 있다.

6. Ngram method

마지막으로 Ngram tokenizer 방식을 시도해보았다. 서칭결과, Ngram의 구현방식은 alphabet 단위와 word단위 2종류가 있었다. alphabet 단위는 딥러닝 수준의 알고리즘과 데이터량이 필요할 것이라 추측했지만 구현자체는 어렵지 않아 두 방식 모두 시도해보았다.

```
ngt=analysis.NgramTokenizer(minsize=1,maxsize=12)
tokens=ngt(q)
new_q=""
for token in tokens: new_q+=token.text+" "
```

위와 같이 알파벳 단위로 ngram을 시도해본결과 maxsize를 키울수록 성능이 높아지기는 했으나, ngram을 쓰지 않았을 때에 비해는 낮은 성능이었기에 기각했다.

```
def extract_ngrams(data, num):
    n_grams = ngrams(word_tokenize(data), num)
    return [''.join(grams) for grams in n_grams]
    # return [' '.join(grams) for grams in n_grams]
```

```
##################################################
tokens =extract_ngrams(new_q,1)
tokens+=extract_ngrams(new_q,2)
tokens+=extract_ngrams(new_q,3) # ... to 8
##################################################
for token in tokens: new_q+=token+" "
```

위와 같이 extract_ngram이란 함수를 정의하고 tokenize를 해주었다. 처음에는 주석 코드와 같이 token 안의 여러 단어들 사이에 공백을 두었는데, parser에서는 공백을 기준으로 단어를 구분했기 때문에 ngram이 무효화 되었다. 이에 따라 편법으로 공백을 제거하고 시도하였으나 성능이 감소했고, ngram의 range를 증가시킴에 따라 소폭 향상되었으나 여전히 기존 모델의 성능이 좋아 기각했다.

```
# defining part schema in make_index.py
contents=TEXT(analyzer=StemmingAnalyzer(stoplist=STOP_WORDS)|NgramFilter(
    minsize=1,maxsize=18))
# (minsize=1,maxsize=18,at='start'), (minsize=1,maxsize=18,at='end')
schema = Schema(docID=NUMERIC(stored=True),  title=NGRAMWORDS(minsize=2,
    maxsize=10, stored=True, field_boost=1.0, tokenizer=None, at='start',
    queryor=False, sortable=False),contents=TEXT(analyzer=StemmingAnalyzer(
    stoplist=STOP_WORDS)))
```

queryresult.py에서 word단위의 ngram을 정석적으로 주는 방법을 찾지 못했다. 자료조사 끝에 shema 형성시 NgramFilter를 주거나 NgramWord 설정을 줄 수 있음을 확인했고 parameter(range and at)를 변화시켜가며 결과를 비교했다. 그 결과 ngram range를 키울수록 성능이 증가했으나 약 18 이상의 사이즈부터는 성능의 향상은 없었다.

E. CustomScoring.py 구현

강의 내용에 따라 scoring 으로 활용할 수 있는 것들은 tf.idf, binary independence model, querylikelihood model 등이 있다. 하지만 주어진 query 에 대해 relevant document set 을 아는 것은 대부분 불가능하다. 이때, 사용할 수 있는 scoring 으로 tf.idf, BM25(r_i, R = 0) 이 있다. 따라서 scoring 분석은 tf.idf, k1 값을 다르게 한 BM25를 사용하여 실험하였다.

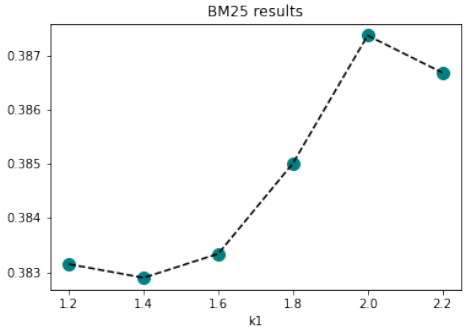

FIG. 3: BM25 results with various k1

1. *tf.idf*

term frequency 와 inverse document frequency 를 활용한 scoring 은 다음과 같이 계획하였다. 모두 0.2 0.3 사이 score 를 보였으며, BM25 보다 성능이 낮았다.
1) tf×idf 2)tf/fl×idf 3)(log(tf)+1)×log(idf) 4) log(tf×idf) 5) tf/cf
6) (tf/cf)×(tf/fl)/idf×avgfl
각각에 대한 BPREF score 는 표I과 같다.

TABLE I: results for tf.idf

tf×idf	tf/fl×idf	(log(tf)+1)×log(idf)
0.2111437285710863	0.2986156646228551	0.2978252843784924
log(tf×idf)	tf/cf	(tf/cf)×(tf/fl)/idf×avgfl
0.2822899523613001	0.23232100734359537	0.2275473813510324

2. *BM25*

BM25 의 경우 B,K1 은 pre-trained 되는 parameter 값으로 B=0.75 이며 k1∈[1.2,2] 의 값이다. 따라서 k1 값을 변화시키며 score 를 확인하였고 결과는 Fig 3 와 같다. k1 값이 [1.2,2] 이지만 이 사이에서 계속 증가하여 2.2 까지 실험하였고, 2에서 최대값을 가지는 것을 알 수 있었다.

F. Search Engine - Result Discussion

결과적으로, 본 연구에서 구현한 best model은 약 38.7%의 정답률을 보였다. 이는 초기 query에 stopword 처리만 해주었을 때 정답률인 32%대비 약 6.7% 향상된 결과이다. 배운 바

와 같이 가중치, stopword 제거, spellcheck, stemming, Ngram을 모두 구현해 시도해보았다. 아쉬운 점은 best model에 Ngram이 채택되지 않은 것인데, 이는 query 혹은 document의 data 갯수가 Ngram 방식을 쓰기에 부족했던 것으로 추측했다.

III. PART II : CLASSIFICATION CLUSTERING

A. Classification - Naïve Bayes Classifier

나이브 베이즈 분류 모델을 파이프라인을 사용하여 구축하였다. 사용한 파이프라인 내의 원소(변환기)는 Couuntvectorizer, Tfidftransformer, MultinomialNB이다. 각 변환기의 조정한 parameter는 다음과 같다.

CountVectorizer	Stop_words, ngram_range, max_df, min_df, max_features
TfidfTranformer	-
MultinomialNB	Alpha

TABLE II: Classifier parameter

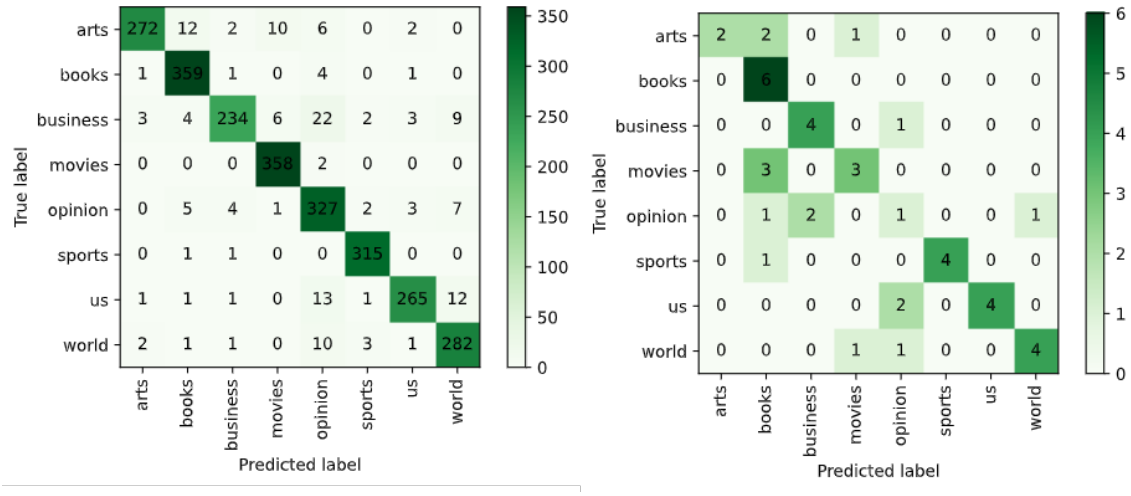

FIG. 4: Left is train data and the right is the test data

Fig 4에서 주목할 만한 점은 첫 번째로 training의 정확도에 비해 test의 정확도가 매우 낮다는 것, 두 번째로 양쪽 모두에서 opinion 그룹의 오답률이 다른 그룹에 비해 높은 편이었다는 것이다. 전자의 경우, ngram_range를 조절하여 충분히 많은 데이터를 제공하고자 했다.

Ngram_range의 min값을 올리면 과적합 현상이 더 극명히 드러났다. 실제로 ngram_range를 (2,2)나 (1,3)으로 설정하자, train_data의 정확도는 98%이상으로 올라가는 반면, 맞춘 test data의 수는 23개까지 하락했다. 따라서 이 분석에서는 ngram_range를 (1, 1), (1, 2)만을 비교해 보았다. 또, 여기서 발생하는 과적합은 alpha값을 조절하며 방지하기로 했다.

후자는 정확한 조절 방법을 찾기 어려워 주어진 train data의 내용을 알아보았다.

우선 frequency가 높은 단어들을 보았다. 'said', 'new', 'mr' 같은 단어들이 그 순위가 높았는데, 이런 단어들 중 일부는 다수의 그룹에 영향이 커 분류하는데 큰 효과가 없을 수 있다. 덧붙여 사용되는 vectorize된 단어들의 대부분이 그 빈도수가 전체에서 1 ∼ 2인 단어들이고, 단어들의 총 개수가 unigram에서는 32,431개, unigram ∼bigram에서는 286,455개로, 이 역시 많은 값이었다. max_df, min_df값을 바꾸며 이를 개선하기로 했다.
위와 같은 방법으로 좋은 성능을 나타내는 최적의 매개변수를 다음과 같이 찾았다.
Fig 5에 따르면 test data에서의 성능이 70 ∼ 75%으로 올라갔다. 여전히 opinion그룹을

Parameter	value-Only Unigram	value-Unigram∼Bigram
Stop_words	'english'	'english'
Ngram_range	(1, 1)	(1, 2)
Max_df	700	700
Min_df	2	2
alpha	0.4	0.4
Test 정확도	31/44	30/44

TABLE III: parameter of model

맞춘 케이스는 여전히 적었는데, 이는 다른 parameter나 모델을 사용할 때도 계속 나타나는 현상이었다. 위의 경우 opinion그룹을 정확히 맞춘 횟수가 각각 0, 1이었다.

어떤 단어들로 인해 Naïve bayes 분류기가 판단을 하는지를 찾기 위해 카테고리별로 높은 확률을 가지게 하는 단어들을 조사해보았다. 위에서 성능이 좋았던 case가 unigram인 점을 고려해 unigram 모델을 계속 사용하였다.
전체적으로 각 확률이 0.4이상의 값으로 그 구분이 명확히 되는 단어들이었다. 다만 opinion과 같이 test data에서 성능이 떨어지는 카테고리의 경우, 주요 단어들을 의미적으로 주목해보아도 opinion인지 모호한 경우도 있었다. 예를 들면, 'editor'는 충분히 books 그룹에, 'infrastructure'은 business 그룹에 들어갈 만한 단어들이고, 실제로 이 단어들을 각각 books

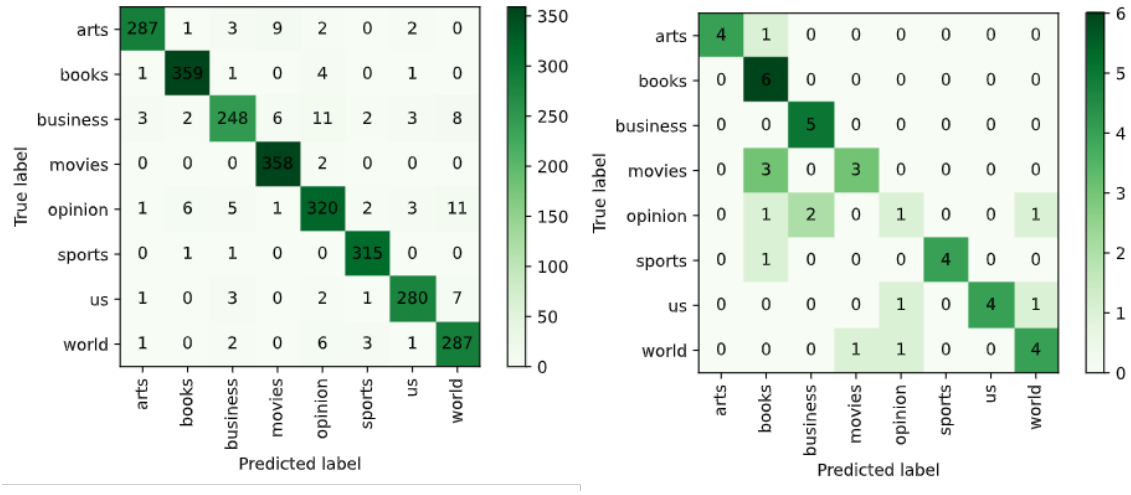

FIG. 5: The left is the train data and the right is the test data.

Category	주요 단어(상위 5개)	상위 20개 단어의 가능성 평균
Arts	'lets' 'museum' 'music' 'album' 'rundown'	0.54
Books	'novel' 'book' 'books' 'pp' 'writes'	0.62
Business	'stock' 'airlines' 'market' 'gamestop' 'tesla'	0.46
Movies	'film' 'movie' 'oscar' 'globes' 'directed'	0.62
Opinion	'bret' 'editor' 'gail' 'housing' 'infrastructure'	0.40
Sports	'tournament' 'league' 'players' 'game' 'season'	0.67
Us	'floyd' 'chauvin' 'impeachment' 'democrat' 'senate'	0.44
world	'israel' 'minister' 'israeli' 'vaccine' 'gaza'	0.46

TABLE IV: Top word

와 business로 있을 확률은 각각 0.23, 0.19 정도이다. 'museum'와 같은 다른 단어들의 확률들이 대개 다른 그룹에 분류될 확률이 0.1이하라는 점을 고려하면 그 설명력이 부족한 단어들이라고 생각할 수 있다.

성능 개선을 위해서는 이와 같은 단어들을 명확히 하는 것이 중요하다고 보인다. 각 단어(unigram)가 group에 배정될 확률이 고르게 분포한다면, 그 단어는 분류 작업에 큰 영향을 미치지 못할 것이라 판단했다. 따라서 구분 짓는 확률들의 분산이 일정값 이하인 경우를 stopwords를 통해 제외하기로 했다. 다양한 값으로 분석한 결과 분산이 0.0005미만인 경우를 제외했고, 이 때 test data에서의 성능은 Fig 6과 같았다.

여전히 opinion을 분류하는데 취약하지만, precision에서 비교적 좋은 성능을 보였다. 위의

	precision	recall	f1-score
arts	1	0.8	0.89
books	0.5	1	0.67
business	0.71	1	0.83
movies	0.75	0.5	0.6
opinion	1	0.2	0.33
sports	1	0.8	0.89
us	1	0.83	0.91
world	0.71	0.83	0.77
accuracy			0.75(33/44)

TABLE V: Rate

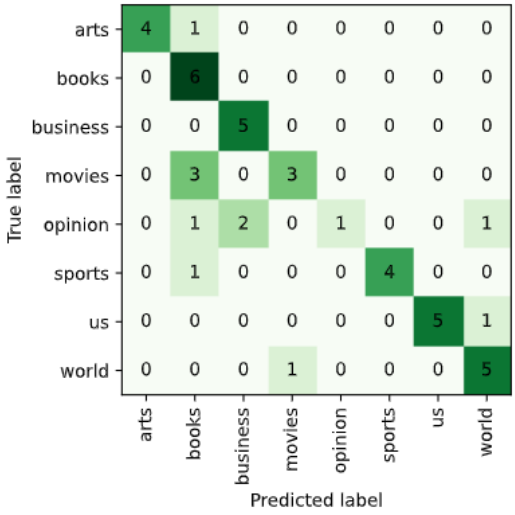

FIG. 6: confusion matrix of test data

실험을 바탕으로 다음과 같은 최종 모델을 선정했다.

B. Classification - Support Vector Machine

SVM모델 역시 비슷한 구조의 파이프라인을 사용하였다. Countvectorizer와 Tfidftransformer에서 사용되는 파라미터의 종류는 같다. gridseachCV로 여기서 최적의 parameter을 찾은 결과, max_df=950, min_df=4, stop_words='english'에서 좋은 성능을 나타냈다.

SVC에서 주로 조절한 파라미터는 kernel, C, gamma, coefficient이다. SVC에서는 이

Parameter	값
Stop_words	'english' + (별도 설정)
Ngram_range	(1, 1)
Max_df	700
Min_df	2
alpha	0.4
Test 정확도	33/44

TABLE VI: 최종 Naïve bayes 모델

파라미터들에 영향을 많이 받는다는 사실이 알려져 있으므로, 이 파라미터들로 인해 생기는 변화들에 더욱 주목하려 했다.

먼저 kernel을 default인 rbf로 했을 때이다. Stopword를 제외하고 별다른 파라미터를 설정하지 않았는데도 train set과 test set에서 각각 정확도 99%(2565/2573), 75%(33/44)으로 앞선 나이브 베이즈 분류기보다 높은 성능을 보였다. 다만, 이 경우 test에 비해 train의 정확도가 너무 높으므로 과적합이 일어난 경우라고 생각할 수 있었다. 그 외에 Kernel을 다른 방법으로 설정해보았다. 아래는 그 결과이다.

Kernel이 비선형인 rbf나 poly 같은 경우 train data를 더 잘 학습했으나 test data에는

kernel	Train accuracy	Test accuracy
rbf(default)	2565	33
linear	2553	34
Poly(2)	2573	34
sigmoid	2211	29

TABLE VII: kernel -accuracy

그에 비해 큰 향상을 보이지 않았다. 이는 다양한 parameter를 설정해도 나타났다. Sigmoid는 training accuracy 역시 낮은 값이었다. 따라서 kernel은 linear하게 두어 성능의 향상을 알아보기로 했다.

Linear에서 gridSearchCV를 사용하여 C값을 구한 결과는 다음과 같다. C=1에서 score가 가장 높게 나타났다.

위의 C를 1로 결정하고, 다음으로 ovo와 ovr을 비교를 해보았다. 그러나 둘 사이에 유의미한 변화는 나타나지 않았다. 두 모델의 train data와 test data에서 confusion matrix가 정확히

C	2	1.5	1	0.5	0.25
Mean_test_score	0.795	0.798	0.800	0.790	0.742

TABLE VIII: Mean test score

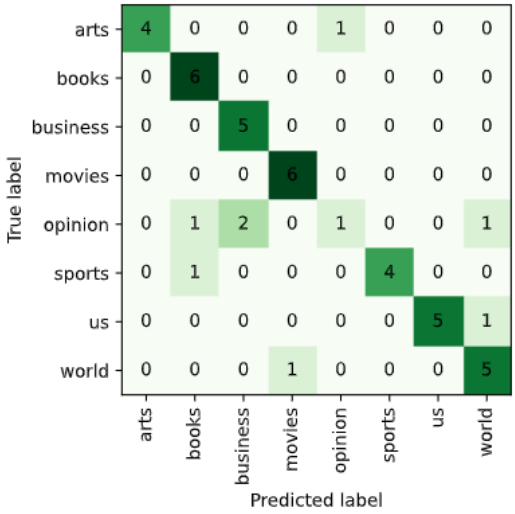

FIG. 7: 최종 모델의 test confusion matrix

일치했기 때문이다. 따라서 최종 선택된 모델의 변수와 그 테스트 결과는 Fig 7과 같다.

Parameter	값
Stop_words	'english'
Ngram_range	(1, 2)
Max_df	950
Min_df	4
kernel	Linear
Decision_function_shape	'ovr'
Test 정확도	36/44

TABLE IX: Last SVM model

C. Clustering - Greed searching in given pipeline

1. Countvectorizer

sklearn의 클래스인 Countvectorizer는 다큐먼트를 받아서 preprocessing, stemming, tokenization을 통해 문서를 토큰들로 분해하고, 이 중 feature로 사용할 적절한 토큰들을 골라 등장 회수로 벡터를 만든다. 조정가능한 내부 파라미터들은 다음과 같다.
fi|0#0 Kmeans clustering을 위해선 우선 벡터화가 의미 있는 feature들로 구성되어야 하기 때문에 Countvectorizer의 파라미터들부터 수정했다. Analyzer, max_features, binary 등은 feature 생성에 의미가 있는 파라미터들이지만 default값이 최적임을 몇 번의 테스트를 통해 확인할 수 있었다. KMeans clustering의 v_measure 평가값에 직접적으로 영향을 준 파라미터들은 다음과 같다.

2. Countvectorizer-stop_words

기본값은 None이며, 그 외에 기본 내장된 'english'를 사용할 수 있다. 'english' stopwords를 사용 시 None에 비해서 높은 v_measure. 상승을 확인할 수 있었다. 자주 등장하지만 특이성이 부족한 feature들은 idf로 그 중요도가 보정됨에도 불구하고 많은 노이즈를 생성하고 있었음을 확인할 수 있었다. Stopword의 목록이 군집화의 정확도에 영향을 준다는 점에 착안하여 여러 stopword들을 테스트해보았다. 각각에 대한 수행 결과의 평균은 다음 그래프로 확인 가능하다. 기본 내장 stopword인 'english'를 포함하여, 'postgresql.txt', 'lingpipe.txt', 'nltk.txt', 'okapi_sample.txt', 'onix.txt'에서 낮은 표준편차와 높은 평균치를 가진 v_measure를 확인했으며 이후 다른 파라미터 조정 시에 위의 stop_word들에 대해 반복문을 시행해 가장 높은 v_measure를 가지는 모델을 찾았다.

3. Countvectorizer - ngram range

ngram은 토큰 생성 시 몇 개의 단어를 연속으로 하여 토큰을 생성할지에 대한 값이다. (n, m)으로 설정 시, n개 이상, m개 이하 단어의 sequence들을 토큰들로 생성하게 된다. 주어진 데이터셋은 신문기사이므로 구문보다는 단어로 비교적 주제가 명확하게 구분되는 데이터일 것이라 예상했으며 n-gram의 (n,m) 조합을 (1,1)부터 (5,5)까지 총 10가지를 확인해본 결과, (1,1)의 unigram feature가 가장 안정적으로 높은 v_measure를 보이는 예측과 일관된 결과를 확인했다.

Stopword	평균 v_measure	최대 v_measure
Onix.txt	0.421244	0.459918
Okapi_sample.txt	0.402374	0.409524
Lexisnexis.txt	0.397162	0.451394
postgresql.txt	0.3980842	0.447895
nltk.txt	0.383623	0.455236
'english'	0.406156	0.448173
Total stopword	0.37068369	0.459918

TABLE X: Confusion Matrix

4. Countvectorizer - min_df, max_df

생성된 토큰들 중 너무 적게 등장하거나 너무 많은 문서에서 등장하는 토큰들을 feature 선택 시 제외하는 파라미터이다. min_df의 경우 여러 값에 대한 반복 시행 결과, 20 ∼30 사이 값에서 최적의 v_measure를 보임을 확인했다. 이 사이의 값들 중에 서로 다른 모델마다 반복하여 v_measure가 가장 높은 값을 선택했다. max_df의 경우 너무 낮은 값에선 의미 있는 토큰들마저 제외해버려 군집화 정확도를 큰 폭으로 낮추었고, 적당한 값 이상이 되면 큰 변화를 주지 못했다. 높은 df를 가진 단어 중 feature로서 의미가 적은 것들은 이미 stop_words로 대부분이 제외되기 때문으로 이해했다.

5. tfidfTransformer

tfidfTransformer는 feature로 정해진 단어들의 등장 회수를 나타내는 Countvectorizer의 반환값을 tf(term frequency) 및 idf(inverse document frequency) 값으로 보정하며 필요할 시 log 등으로 스케일링을 해주는 클래스이다. 조정가능한 내부 파라미터들은 다음과 같다.

Default parameters
norm='l2', use_idf=True, smooth_idf=True, sublinear_tf=False

위 파라미터들은 대체로 v_measure값에 유의미한 상승을 보이지 못했다. 특히 Norm, Use_idf, Smooth_idf 값들은 기본값이 최적임을 간단히 확인할 수 있었다. 군집화 형태에 눈에 띄는 변화를 보이는 파라미터는 sublinear_tf였다.

6. tfidfTransformer -Sublinear_tf

이를 True로 설정할 시 tf값을 1+log(tf)로 보정했다. 이 경우 결과 분포는 다음과 같이 바뀌었다. Log scale로 변화하며 결과 값에 변화는 주지만 군집의 상대적인 복잡도에는 큰 변화를 주지 못하므로 v_measure 상승에 영향을 주지 못했다.

7. KMeans Clustering

KMeans Clustering은 K개의 initial point를 군집의 중심으로 지정 후, 그 점들을 중심으로 유클리드 거리를 계산하여 각 개체들의 군집을 할당, 그리고 각 군집의 중심을 재계산하여 위의 과정을 반복하여 K개의 주어진 군집을 생성했다. 조정가능한 파라미터들은 다음과 같다.

위 파라미터들 중 군집화 성능에 직접적인 영향을 주는 것들은 init, random_state, n_init, tol, algorithm이었으며 이 중 init은 K-Means++ 알고리즘으로, 초기 centroid가 분산되어 나타나도록 점의 선택 확률을 조정해주며 기본 내장 방식이다. Tolerance(tol)값은 군집화의 변화를 확인하여 수렴 여부를 결정하는 값으로, 반복 시행 결과 적당히 낮은 값에서 미미한 성능 개선을 보임을 확인하여 1e-6으로 충분히 작게 설정해주었다.
-random_state : random_state는 군집화 함수 제출 시 결과를 일정하게 하기 위해 서로 다른 방법으로 모델을 개선할 때마다 random_state 0 ∼42에 대해 최적 시작값을 고정하기 위해 반복문을 시행했다.
-n_init : n_init의 경우 적당히 큰 값으로 설정하면 모델의 성능 개선에 큰 영향이 없음을 확인했다. n_init이 클수록 여러 차례 반복하여 수행 시간이 선형으로 길어졌기 때문에 수행 시간과 타협하여 기본값인 10을 사용했다.
-algorithm의 경우 scikit-learn의 KMeans 클래스에선 EM-style(기댓값 최대화) 알고리즘인 'full'과 Lloyd 알고리즘의 변형인 Elkan 알고리즘 'elkan'을 지원했는데, 여러 차례 실험을 통해 이 두 값에 평균엔 큰 차이가 없음을 확인했고 수행시간이 적어 반복 수행이 빠른 'full' 알고리즘을 채택했다.

D. Clustering - Trying Other Scoring Methods

1. Vectorize-Tokenization

CountVectorizer 기본 토큰화는 특수 문자 제거, 온점 및 낱개 문자 제거를 수행한다. 다른 토큰화 방식을 시도하기 위하여 nltk 패키지를 사용했다. 이 중 callable로 사용가능한 word_tokenize, wordpunct_tokenize를 시도해보았지만 v_measure의 상승은 확인하지 못했다.

2. Vectorize-Stemmer, Lemmatizer

CountVectorizer에서는 어근을 찾는 stemming 과정이 포함되지 않기 때문에 stemming으로 토큰의 의미가 같을 경우 같은 feature의 수의 포함되도록 하였다. nltk 패키지에서 제공되는 Stemmer와 Lemmatizer에 대해 CountVectorizer의 analyzer 함수를 재정의한 새 클래스를 생성하여 Vectorizer로 사용해보았다.
-The LancasterStemmer class: v_measure avg = 0.438076911 max=0.46716645
-The SnowballStemmer class: v_measure avg = 0.438052360 max=0.45759459
-The WordNetLemmatizer class : v_measure avg = 0.44875163, max= 0.48838172
세 가지 경우 모두 기존 최댓값을 갱신하지 못하여 채택하지 않았다.

3. Vectorize-Word2Vec

CountVectorizer는 One-hot 벡터를 생성하는 반면 Word2Vec은 워드 임베딩을 하며 단어 간 유사도를 반영하여 단어의 의미를 벡터화 할 수 있다. 이런 Word2Vec 모델 문서를 이용해 문서를 임베딩 하는 것을 시도했으나 유의미한 변화를 발견하지 못했다. 조사해본 결과, Word2Vec의 단어 간 유사도를 학습하는 기능은 큰 자료에서 더 좋은 성능을 보인다고 한다.

E. Best Model 파라미터 및 결과 분석

더 이상의 모델 개선은 어렵다 판단하여 기존 모델에서 더 엄밀하게 그리드 서칭을 진행하여 다음과 같은 최종 모델을 얻어냈다. (미표기시 기본값 사용)
- CountVectorizer : stop_words='okapi_samplel.txt', max_df=0.5, min_df=24
- TfidfTransformer : norm='l2', sublinear_tf=True
- KMeans : n_cluster=8, n_init=10, tol=6e-1, algorithm='full', random_state=17
- v_measure : 0.51146056

1. PCA를 통한 차원 축소 비교

PCA를 통한 시각화 이미지는 Fig 8와 같다.

KMeans Clustering은 유클리드 거리를 사용하는 특성 상 각 군집이 구 형태로 밀집되어 있을 때 잘 작동한다. 군집화 결과인 Figure 1을 확인해보면 낮은 차원이지만 투사된 개체들의 군집이 특정 위치에 타원 형태로 모여 있음을 확인할 수 있다. 반면 target 분포의 경우 밀집된 부분을 제외하고는 보다 넓게 산개해 있거나 타원을 벗어난 복잡한 형태를 띄는 것을

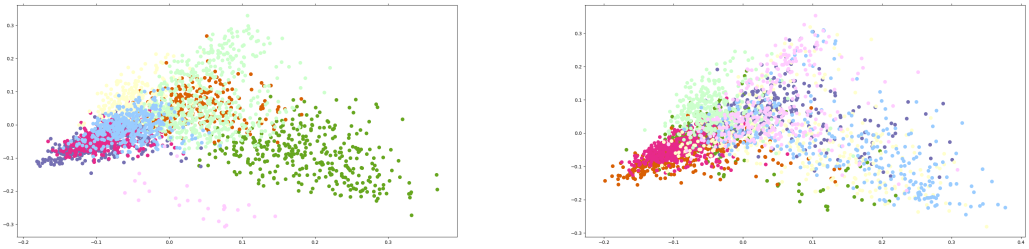

FIG. 8: The left image is cluster and the right image is target.

확인할 수 있다. 이를 Vectorizer와 scoring function을 수정함으로써 어느 정도 범위에서는 분포의 변형이 가능하지만 어느 모델이든 tf(Term Frequency)에 근거한 벡터를 생성하는 이상, 스케일의 변화를 주거나 노이즈를 줄일 수 있을 뿐 복잡한 형태의 분포를 단순한 형태로 수정하기는 어렵다. 즉 K-Means 군집화로 비지도 학습을 하는 이상 주어진 데이터의 복잡한 형태를 모두 반영하는 성공적인 군집화는 어렵다고 진단하였고, 여러 파라미터 테스트, tokenization, stemming, scoring 함수의 시도에도 군집화 평가지표인 v_measure가 0.5 근방에서 벗어나지 않는 이유로 해석하였다.

	1	2	3	4	5	6	7	8
Opinion	8	18	16	22	0	5	**215**	**25**
Business	6	12	**310**	4	1	18	21	0
World	30	**175**	6	6	2	40	29	0
Us	0	3	8	**330**	0	2	23	0
Arts	**125**	45	31	2	1	**114**	36	0
Sports	1	8	2	2	**252**	26	31	0
Books	**158**	11	2	0	1	**118**	10	0
movies	9	9	1	5	2	**254**	26	0

TABLE XI: Confusion Matrix

2. Metrics 시각화

해당 모델을 Metrics의 confusion_matrix로 라벨에 따른 분포를 나타낸 결과는 다음과 같다. 각 행이 같은 타겟 카테고리, 세로가 같게 분류된 군집 라벨이다. 우선 군집화에서는 라벨의 의미에 관계없이 각 군집에 대해 번호로 라벨을 지정한 것을 확인할 수 있다. 군집화에서 특정

카테고리에 대한 벡터 정보가 전혀 주어지지 않았기 때문인데, 군집에 대한 적절한 라벨 지정을 위해선 군집의 평균 feature 구성과 학습된 라벨의 feature를 비교하며 분류하는 과정이 필요함을 알 수 있었다. 이제 군집을 분석해보겠다. 한 군집은 하나의 카테고리로 구성되며 총 8개의 군집이 균일하게 생겨야 성공적인 군집이다. 이 기준으로 카테고리 기준으로 먼저 보면, 'opinion', 'business', 'world', 'us', 'sports', movies' 카테고리의 경우 각 카테고리가 하나의 라벨에 대부분이 포함된 군집을 가지고 있다. 다른 라벨에 얼마나 퍼져서 분표하는지 여부는 각각 다르며, 그 정도는 주제의 커버 범위의 다양성 지표로 이해할 수 있다. 또한 군집에서 빠져나간 개체들도 소규모로 다른 군집의 일부분에 모여 있는 것을 확인 가능하다. 'sports'의 경우 5번 군집이 주를 이루며, 6, 7번 군집에 소규모 군집을 이룬 것을 볼 수 있다. 반면 'arts', 'books' 카테고리에서는 두 큰 군집으로 나뉘고 'arts'의 경우 꽤 큰 소규모 군집을 2, 3, 7번 라벨에서 이룬 것을 볼 수 있다. 군집 라벨 기준으로 분석해보면 2, 3, 4, 5번 라벨은 단일 카테고리로 구성되며 성공적이게 카테고리에 따른 군집이 이루어 졌음을 알 수 있다. 7번, 8번의 경우 하나의 카테고리가 두 군집에 걸쳐 있는데, 특히 8번 군집에 해당하는 문서들이 다른 문서들과 확연히 구분되는 feature를 가졌음을 알 수 있다. 두 군집을 점유하지만 둘을 하나의 군집으로 묶어 볼 경우 단일 카테고리로 이루어지므로 두 중심으로 퍼져 있을 뿐 나쁘지 않은 군집이라고 볼 수 있다. 1번 군집은 'arts' 및 'books'가 혼재되어 있다. 'arts' 및 'books'에서 공통적이게 사용하는 문화 관련 용어가 있어 구분이 명확치 못했음을 추측하였다. 또한 6번 군집에서는 'movie' 문서들과 'arts', 'books' 문서들이 혼재해 있는데, 이 또한 인문 분야 용어의 혼재로 인해 분류가 어려웠음을 알 수 있다. 7번 군집에는 'opinion' 카테고리가 주를 이루지만 다른 카테고리에서도 일정 개수만큼 소규모 군집을 이뤄 참가해 있다. 'opinion' 성격의 문서들이 다른 여러 카테고리에 분산되어 분포하고 있음을 알 수 있다.

F. Clustering-discussion

PCA의 차원 축소 시각화를 통해 주어진 문서 집합의 분포가 꽤 복잡함을 알 수 있었다. Countvectorizer와 tfidfTransformer의 여러 파라미터를 조정하며 그 분포를 어느 정도 퍼뜨리며 군집의 정확도를 향상시킬 수 있었으며 그에 가장 큰 영향을 미친 것은 stopwords, min_df, normalization, n_init 파라미터들이었다. 그 외에도 sublinear_tf, max_df 등이 미미한 영향을 미치는 것을 확인했다. 기본 내장 방식 외에 다른 tokenizer, stemmer, Lemmatizer를 시도했으나 유의미한 결과를 발견하지 못했다. 결과적으로 random_state를 바꿔가며 최고 v_measure를 보이는 결과를 찾는 데 가장 큰 시간을 쏟았다. 그 결과를 분석해본 결과 군집화 성능이 낮은 부분은 실제 문서 카테고리에서도 혼동될 여지가 있는 카테고리들임을 정성적이게 확인할 수 있었다.