Document retrieval & Classification and Clustering

팀: 9조

이름: 강명훈 김원재 김영헌

Part I: 문서 검색 엔진

1. 문제 설명

Part I 에서는 검색 엔진 모듈을 python의 whoosh 라이브러리를 사용하여 구현하여야 한다. 구현하는 검색 엔진은 특정 환자에 대한 진료기록에 대해 관련이 높은 순서대로 의학적 내용을 담은 문서들을 나열할 수 있어야 한다.

2. 문서 분석

(1) 개요

문제 조건을 보면 쿼리를 분석을 할 수는 없고, 문서 분석은 허용되는 조건에 따라, 문서 파일을 분석해 본다. 먼저, 문서에서의 단어 빈도를 분석하여 문서에 어떤 단어가 가장 많은지를 파악한다.

(2) 단어 빈도 분석

아래 코드를 작성 후, SE 폴더의 동일 위치에 넣은 다음에 컴파일하면 문서에 사용된 용어들을 내림차순으로 100개를 나열할 수 있다.

```
#%%
from nltk.corpus import stopwords
from nltk.tokenize import RegexpTokenizer
from nltk import Text


stopwords = set(stopwords.words('english'))
tokenizer = RegexpTokenizer(r"\w+")

with open('doc/document.txt', 'r', encoding='utf-8') as f:
    doc = f.read()
    doc = tokenizer.tokenize(doc)
    docwords = [word.lower() for word in doc if word.lower() not in stopwords]
    nostop = Text(docwords)
    a = nostop.vocab()
    print(a.most_common(200))
```

위 코드를 작동하면 아래와 같은 출력이 나오게 된다.

[('patients', 10694), ('0', 5523), ('1', 4198), ('2', 3465), ('treatment', 2891), ('p', 2777), ('clinical', 2714), ('study', 2712), ('patient', 2682), ('acute', 2442), ('risk', 2428), ('results', 2419), ('3', 2401), ('disease', 2327), ('case', 2259), ('5', 2130), ('associated', 2023), ('diagnosis', 1974), ('may', 1784), ('methods', 1769), ('group', 1764), ('6', 1743), ('4', 1666), ('year', 1575), ('years', 1507), ('cases', 1505), ('pain', 1449), ('therapy', 1444), ('high', 1422), ('background', 1373), ('coronary', 1373), ('7', 1344), ('8', 1328), ('mortality', 1305), ('two', 1279), ('pregnancy', 1273), ('abdominal', 1272), ('age', 1262), ('significant', 1230), ('conclusion', 1223), ('symptoms', 1223), ('old', 1216), ('95', 1211), ('one', 1208), ('care', 1199), ('management', 1193), ('9', 1192), ('common', 1160), ('blood', 1154), ('hospital', 1152), ('non', 1151), ('af', 1151), ('diabetes', 1133), ('factors', 1094), ('also', 1093), ('women', 1093), ('present', 1078), ('use', 1073), ('compared', 1071), ('used', 1066), ('infection', 1047), ('however', 1042), ('severe', 1040), ('performed', 1037), ('stroke', 1026), ('ci', 1018), ('rate', 1018), ('data', 1002), ('using', 1002), ('10', 1001), ('atrial', 979), ('report', 978), ('studies', 973), ('increased', 967), ('rare', 965), ('ct', 951), ('levels', 943), ('first', 941), ('complications', 941), ('artery', 940), ('showed', 939), ('days', 934), ('conclusions', 932), ('chronic', 929), ('significantly', 923), ('cause', 922), ('bleeding', 921), ('higher', 913), ('analysis', 912), ('without', 909), ('due', 907), ('cancer', 902), ('surgery', 897), ('early', 897), ('control', 889), ('syndrome', 836), ('lung', 831), ('well', 828), ('reported', 826), ('presented', 821), ('history', 818), ('mean', 815), ('treated', 809), ('time', 803), ('failure', 803), ('outcome', 788), ('type', 785), ('bowel', 784), ('based', 781), ('diagnostic', 779), ('presentation', 774), ('n', 772), ('medical', 771), ('total', 769), ('12', 766), ('groups', 758), ('heart', 756), ('review', 755), ('cardiac', 753), ('outcomes', 748), ('including', 745), ('fibrillation', 745), ('findings', 738), ('among', 738), ('small', 738), ('related', 735), ('pulmonary', 734), ('found', 732), ('positive', 731), ('revealed', 720), ('months', 719), ('chest', 717), ('left', 716), ('day', 715), ('low', 713), ('injury', 706), ('included', 705), ('lower', 700), ('surgical', 700), ('vs', 699),

('emergency', 696), ('primary', 692), ('pancreatitis', 678), ('incidence', 668), ('renal', 664), ('weeks', 663), ('obstruction', 662), ('follow', 661), ('normal', 649), ('insulin', 645), ('30', 642), ('diagnosed', 641), ('health', 641), ('respectively', 640), ('level', 630), ('evidence', 626), ('life', 623), ('gastrointestinal', 620), ('three', 618), ('important', 609), ('respiratory', 600), ('long', 600), ('following', 598), ('imaging', 597), ('serum', 586), ('developed', 581), ('pressure', 576), ('within', 575), ('new', 573), ('001', 572), ('glucose', 571), ('term', 568), ('underwent', 567), ('mg', 567), ('period', 565), ('population', 559), ('test', 552), ('infections', 545), ('although', 544), ('urinary', 542), ('effects', 541), ('15', 524), ('function', 522), ('11', 522), ('right', 521), ('score', 517), ('drug', 516), ('literature', 515), ('major', 509), ('events', 508), ('multiple', 507), ('rates', 506), ('tract', 499), ('condition', 497), ('number', 497), ('considered', 496), ('20', 495), ('diseases', 492), ('cell', 488), ('18', 485)]

내림 차순으로 200개의 word list가 출력되었다. 숫자를 나타내는 표현이 (7,8,two 등등) 확인됨을 알 수 있다. 다만 이 숫자는 환자의 나이를 의미하는지, 증상이 있었거나 치료 기간이 며칠이었는지를 나타내는 것일 수도 있기에 중복이 있을 수 있다.
또한 의학적 문서를 다룰 때는 환자의 나이, 성별, 기존 증상도 중요하지만 의학적 증상 및 발병 부위를 위주로 다루는 것이기 때문에 위의 숫자 요소들을 잘 다루어야 할 필요가 있다. 이는 추후 QueryResult.py의 TODO 부분의 stopWords나 Query handling에서 각종 성능 향상을 위해 사용될 수 있다.

3. Make_index.py 수정

(1) stopWords 처리

document.txt를 파싱해서 넣는 경우, 위에서 본 것 중에서 필요 없는 단어는 제외한다. 기존 있던 이 코드는 불용어가 없는데, 미리 Make_index.py에서도 불용어 처리를 추가해서 검색 성능을 향상시키기 위한 토큰화 작업을 수행한다.
이 문제에서 환자의 상태에 맞는 의학 문헌을 검색하는 성능을 좋게 하기 위해서는 2의 문서 분석의 (2) 단어 빈도 분석 자료를 활용한다. 자주 나오는 용어들 중에서 diseases, risk, patients 등이 있다. 이는 특정한 아픈 신체 부위나 환자의 특정 나이나 특정한 치료법에 대해서 언급하는 것이 아니다. 또한 숫자의 경우도 처리해야 한다. 환자의 나이인지, 투약한 약물의 약 횟수 혹은 양이 될 수도 있다. 빈도 순으로 나열했을 경우에 모호한 단어를 포함하면 검색 성능이 떨어질 것으로 보인다. 이를 해결하기 위해서, 일반적인 증상에 대한 단어, 여러 맥락에 사용될 수 있는 단어는 제외하였다. 아래는 그 목록이다.

```
stopWords = set(stopwords.words('english'))
stopWords.update(['patients', 'treatment', 'p', 'clinical', 'study', 'patient', 'results'])
stopWords.update(['case','associated', 'diagnosis', 'may', 'methods', 'group', 'diseases'])
stopWords.update(['year', 'years', 'cases', 'pain', 'therapy', 'background', 'two', 'age'])
stopWords.update(['significant', 'conclusion', 'old', '95', 'one', 'care', 'management', 'common'])
stopWords.update(['hospital', 'non', 'af', 'factors', 'also', 'present', 'use'])
stopWords.update(['compared', 'used', 'however', 'severe', 'performed', 'rate', 'data', 'using',])
stopWords.update(['report', 'studies', 'rare', 'showed', 'days', 'conclusions', 'significantly', 'cause', 'higher', 'analysis'])
stopWords.update(['without', 'due', 'well', 'reported', 'presented', 'history', 'mean', 'treated', 'time'])
stopWords.update(['failure', 'outcome', 'type', 'based', 'presentation', 'n', 'medical', 'total', 'groups'])
stopWords.update(['review', 'outcomes', 'including', 'findings', 'among', 'small', 'related', 'found', 'positive', 'revealed'])
stopWords.update(['months', 'day', 'low', 'injury', 'included', 'lower', 'vs', 'primary', 'incidence'])
stopWords.update(['weeks', 'follow', 'normal', 'within', 'new', 'term', 'underwent'])
stopWords.update(['period', 'population', 'test', 'although', 'effects', 'function', 'right', 'score'])
stopWords.update(['literature', 'major', 'events', 'multiple', 'rates', 'condition', 'number', 'considered'])
```

(2) 스키마 설정 및 analyzer 변경

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

불용어 처리를 할 수 있도록 analyzer를 stemmingAnalyser를 활용하였다. 이 작업을 수행하여 검색의 성능이 좋아짐을 확인하였다.

(3) POS tagging 및 WordNetLemmatizer() 활용

```
def tagToPos(tag):
    if 'JJ' in tag:
        return 'a'
    elif 'NN' in tag:
        return 'n'
    elif 'VB' in tag:
        return 'v'
    elif 'RB' in tag:
        return 'r'
    else:
        return None

def importantTag(tag):
    if 'JJ' in tag:
        return True
    elif 'NN' in tag:
        return True
    elif 'VB' in tag:
        return True
    elif 'RB' in tag:
        if tag == 'WRB':
            return False
        return True
    return False
```

품사 태그를 간단한 품사 범주(형용사, 명사, 동사, 부사)로 변환한다. 예를 들어, 태그에 'JJ'가 포함되어 있으면 형용사로 간주하고 'a'를 반환한다. 마찬가지로, 'NN'이 포함되어 있으면 명사로 간주하고 'n'을 반환한다.

```
index_dir = "index"

if not os.path.exists(index_dir):
    os.makedirs(index_dir)

ix = create_in(index_dir, schema)
writer = ix.writer()

lm = WordNetLemmatizer()
```

```
with open('./doc/document.txt', 'r', encoding='UTF-8') as f:
    text = f.read()
    docs = text.split('////\n')[:-1]

    for doc in docs:
        br = doc.find('\n')
        docID = int(doc[:br])
        doc_text = doc[br+1:]

        tagged_list = pos_tag(word_tokenize(doc_text))
        new_doc_text = ''
        for (word, tag) in tagged_list:
            if not importantTag(tag):
                new_doc_text += '//////////' + ' '
                continue
            word = lm.lemmatize(word, pos=tagToPos(tag))
            if '-' in word:
                for w in word.split('-'):
                    new_doc_text += w + ' '
            else:
                new_doc_text += word + ' '
```

Tokenizing 하는데 있어서 Lemmatizing을 활용하였다. 또한 위에서 설정한 pos_tag를 갖고 반복을 돌면서 nltk로 import한 word_tokenize()을 활용하여 tagged list에 넣은 뒤에, 각 단어와 태그를 확인하면서, 중요하지 않은 pos tag를 갖고 있는 것은 빗금 처리를 하고, 중요한 단어에 대해서는 lemmatize() 함수를 활용하였다. 또한 '-' (하이픈) 처리도 진행하였다. 이렇게 하면 전체적인 쿼리에 대한 검색 성능이 향상됨을 확인하였다.

4. QueryResult.py

(1) Tokenizer, lemmatizer 선택

```
result_dict = {}
ix = index.open_dir("./index")
retokenize = RegexpTokenizer(r'\w+')
lemmatizer = WordNetLemmatizer()
```

RegexpTokenizer는 정규 표현식을 사용하여 문자열을 토큰화한다. 이를 사용하면 공백뿐만 아니라 특수 문자나 구두점 등도 쉽게 제거할 수 있다. 이렇게 하면 텍스트 데이터가 더 깔끔해지고, 불필요한 토큰이 제거되므로 검색 엔진의 성능이 향상될 수 있다. 이렇게 바꾸면 BPREF를 확인해 보니, 성능이 약간 향상됨을 확인하였다. 또한 Stemming보다는 Lemmatizing을 활용하는 것이 더 나은 성능을 보이는 데다가, 쿼리의 양도 그렇게까지 방대한 양은 아니므로, 처리 시간이 그다지 길지 않으므로 성능에 초점을 맞추고자 했다. lemmatizer을 WordNetLemmatizer()을 활용하여 아래 쿼리 파싱에 사용하기로 하였다. 이 두 기능을 사용하면 검색 엔진의 성능이 향상된다.

(2) stopWords(불용어) 처리와 추가

Make_index.py에서 했던 것처럼 똑같은 리스트로 불용어 처리를 해 주었다. 동일한 stopWords set를 만들어서 입력해 주었다. 이렇게 하여 검색 엔진의 성능을 향상시켰다.

(3) Query handling – Ongroup.factory 값 변경

parser = QueryParser("contents", schema=ix.schema, group=OrGroup.factory(0.9))
Parser의 Orgroup 기본값을 0.9로 수정하여 같은 단어가 중복되어 있어도 점수가 동일하게 나오는 현상을 방지하였고, 이렇게 했을 때 약간의 성능 향상이 있었다.

(4) Pos tagging

Make_index.py에서 했던 것처럼 똑같은 식으로 처리를 해 주었다. 이는 아래 쿼리에 대한 정보를 파싱하는 데 동일하게 사용될 것이다. 이렇게 하여 검색 엔진의 성능이 향상시켰다.

(5) 파싱 작업

이제 Make_index.py에서 document를 파싱했던 것처럼 쿼리 문서도 파싱하도록 한다. Pos tagging, 했던 것처럼 작업을 진행한다. 중요한 단어에 대해서는 lemmatize() 함수를 활용하였다. 또한 '-' (하이픈) 처리도 진행하였다.

```
for qid, q in query_dict.items():
    new_q = ''

    sentence = q.lower()
    tagged_list = pos_tag(retokenize.tokenize(sentence))

    for (word, tag) in tagged_list:
        if not importantTag(tag):
            continue
        if word in stopWords:
            continue
        word = lemmatizer.lemmatize(word, pos=tagToPos(tag))
        if '-' in word:
            for w in word.split('-'):
                new_q += w + ' '
        else: new_q += word + ' '
    query = parser.parse(new_q.lower())

    results = searcher.search(query, limit=None)
    result_dict[qid] = [result.fields()['docID'] for result in results]
```

5. CustomResult.py

(1) 함수 모델 선정

기존 뼈대코드에는 BM25F모델이 있다. 이는 있는 쿼리에 대해, 이미 있는 문서가 맞는지 맞는지 아닌지에 대한 척도를 갖고 점수를 매기고, Boolean model과 벡터 모델보다 훨씬 더 나은 점수가 나오도록 하여야 한다. BM25 모델은 상당히 완성도 있는 모델이기 때문에, 완벽하게 다른 모델을 갖고 와서 활용하는 것보다는 이미 존재하는 BM25 모델을 활용하거나, 거기서 더 나아간 기존 모델(BM25L, BM25+)을 활용하여 거기서 파라미터를 조절하는 것이 더 낫다고 판단하였다. [1][2]

먼저 아래와 같이 코드의 클래스 생성자를 구현하였다.

```
class CustomScoring(WeightLengthScorer):
    def __init__(self, searcher, fieldname, text, qf=1, K1=1.5, B=0.75, eps=0.2):
        parent = searcher.get_parent()  # Returns self if no parent
        self.idf = parent.idf(fieldname, text)
        self.cf = parent.frequency(fieldname, text)
        self.dc = parent.doc_count_all()
        # self.fl = parent.field_length(fieldname)
        self.avgfl = parent.avg_field_length(fieldname)

        #self.param = param
        self.qf = qf

        self.K1 = K1
        self.B = B
        self.eps = eps


        self.setup(searcher, fieldname, text)

    def _score(self, weight, length):
        # return bm25_plus(self.idf, weight, length, self.avgfl, self.K1, self.B, self.eps)
        return bm25L(self.idf, weight, length, self.avgfl, self.K1, self.B, self.eps)
        # return intappscorer(weight, self.idf, self.cf, self.qf, self.dc, length, self.avgfl, self.param)
```

추가로 BM25L, BM25+을 둬 놓고, 기존 것과 비교했을 때 더 높은 성능을 보이는 함수를 찾는다.

```
def bm25L(idf, tf, fl, avgfl, K1, B, eps):
    m = tf/(1 - B + B * fl/avgfl)
    return idf * (K1 + 1) * (m + eps) / (K1 + m + eps)

def bm25_plus(idf, tf, fl, avgfl, K1, B, eps):
    return idf * (tf * (K1 + 1) / (K1 * (1 - B + B * (fl / avgfl)) + tf) + eps)
```

함수를 비교하여 보니 BM25L이 가장 나은 성능을 보였기에, 거기서 파라미터 조절을 해 보기로 했다.

(2) 파라미터 조절 – customscoring.py에서 진행

BM25L 식을 돌려 본 결과 smoothing에 활용하는 eps는 0으로 하여도 오류가 없는 함수여서, eps는 0으로 설정하였다. 성능을 향상시키기 위해 B와 K1을 경험적으로 조절하여서 문제를 해결해야 한다. 이 함수에서 사용되는 두 가지 매개변수는 b와 K1다. B는 0에 가까워질수록 문서 길이가 무시되며, K1은 tf의 포화 수준을 결정한다. 이 매개변수는 토큰이 문서 점수에 미치는 영향을 제한한다. 쿼리 토큰의 tf가 k1 이하일 경우, 해당 키워드가 문서에 등장할 때 BM25L 점수가 급격히 증가하지만, tf가 k1을 초과하면 키워드의 영향은 크게 줄어든다. 즉, k1은 특정 수치 이상의 용어 빈도가 점수에 미치는 영향을 억제하는 역할을 한다. 여러 값을 테스트한 결과 K1 = 3, B = 0.7, eps = 0 일 때 제일 좋은 성능을 보였다.

6. 최종 결과

Evaluate.py로 기존 뼈대코드에 저장된 형태인 random seed(6)으로 돌려 본 결과, 아래와 같은 결과가 나온다


```
test_query = random.sample(sorted(query_dict.keys()), 30)
```

위와 같이 코드를 바꿔서 30개의 모든 쿼리에 대한 점수 평균을 낸 결과 다음과 같이 나온다. (그리고 어차피 모든 쿼리는 30개이기 때문에, 모든 쿼리의 평균적인 점수를 알고자 실시하였다.)

Part 1의 참고 문헌

[1] Trotman, Andrew, Antti Puurula, and Blake Burgess. "Improvements to BM25 and language models examined." *Proceedings of the 19th Australasian Document Computing Symposium*. 2014.

[2] Lv, Yuanhua, and ChengXiang Zhai. "Lower-bounding term frequency normalization." *Proceedings of the 20th ACM international conference on Information and knowledge management*. 2011.

PART2: 문서 분류 및 군집화

PART2에서는 6가지 카테고리의 논문 초록을 분류, 군집화하는 모델을 구현하는 것을 목표로 한다

(P2-1) 논문 초록 분류

1. Naïve Bayes Classifier
   1) NB 분류기로 GaussianNB, MultinomialNB, BernoulliNB, CategoricalNB, ComplementNB 5가지가 있다. 이중 CategoricalNB는 범주형 데이터에 적합하므로 텍스트 데이터에는 바로 적용할 수 없기 때문에 여기서는 생략하고 비교를 진행한다. Sklearn 라이브러리를 이용하여 데이터를 로드하고 pipeline을 이용하여 각각의 분류기에 대해 features을 추출하여 성능을 비교한다.
      - Gaussian Naive Bayes

```
# Gaussian Naive Bayes
clf_gnb = Pipeline([
    ('vect', CountVectorizer()),
    ('tfidf', TfidfTransformer()),
    ('to_dense', FunctionTransformer(lambda x:
np.asarray(x.todense()), accept_sparse=True)),
    ('clf', GaussianNB()),
])
```

GaussianNB는 연속형 데이터를 처리하는 데 사용된다. 이 분류기는 데이터의 각 feature가 정규분포를 따른다고 가정한다.

pipeline에서 countvectorizer(), tfidftransformer()를 사용하여 text data를 vector로 변환할 때 대부분의 값이 0이기 때문에 희소행렬로 변환한다. GaussianNB는 희소행렬을 직접 처리할 수 없기 때문에 FunctionTransformer()를 사용하여 희소 행렬을 밀집 행렬로 변환하여 사용한다.

GaussianNB 성능평가: 정확도 (Accuracy): 71% (339/480)

Metric	algebraic geometry	computer vision	general economics	quantitative biology	quantum physics	statistics theory	Overall
Precision	0.92	0.81	0.67	0.62	0.66	0.57	0.71
Recall	0.78	0.74	0.79	0.71	0.70	0.51	0.71
F1-Score	0.85	0.77	0.72	0.66	0.68	0.54	0.71
Support (샘플 개수)	88	80	80	75	80	77	480

      - MultinomialNB

```
# Multinomial Naive Bayes
clf_mnb = Pipeline([
    ('vect', CountVectorizer()),
    ('tfidf', TfidfTransformer()),
    ('clf', MultinomialNB()),
])
```

MultinomialNB는 이산형 데이터 특히, 단어 빈도와 같은 카운트 데이터를 터리하는 데 특화된 분류기이다. 주로 text분류에 사용되며 각 클래스에 대해 단어의 출현빈도를 기반으로 확률을 계산한다.

MultinomialNB 성능평가: 정확도 (Accuracy): 81% (391/480)

Metric	algebraic geometry	computer vision	general economics	quantitative biology	quantum physics	statistics theory	Overall
Precision	0.95	0.82	0.90	0.85	0.83	0.61	0.81
Recall	0.93	0.91	0.70	0.69	0.81	0.82	0.81
F1-Score	0.94	0.86	0.79	0.76	0.82	0.70	0.81
Support (샘플 개수)	88	80	80	75	80	77	480

- BernoulliNB

```
# Bernoulli Naive Bayes
clf_bnb = Pipeline([
    ('vect', CountVectorizer()),
    ('tfidf', TfidfTransformer()),
    ('clf', BernoulliNB()),
])
```

BernoulliNB는 주로 이진 데이터를 처리하는 NB분류기이다. bool값처럼 각 특징이 존재하거나 존재하지 않는 것을 기준으로 분류한다. 이 분류기는 단어의 등장여부와 같이 이진분류에서 성능이 좋지만 등장 횟수와 같이 다른 특징은 사용하지 않는다.

BernoulliNB 성능평가: 정확도 (Accuracy): 78% (373/480)

Metric	algebraic geometry	computer vision	general economics	quantitative biology	quantum physics	statistics theory	Overall
Precision	0.82	0.85	0.86	0.85	0.77	0.58	0.78
Recall	0.97	0.84	0.75	0.63	0.71	0.74	0.78
F1-Score	0.89	0.84	0.80	0.72	0.74	0.65	0.78
Support (샘플 개수)	88	80	80	75	80	77	480

- ComplementNB

```
# Complement Naive Bayes
clf_cnb = Pipeline([
    ('vect', CountVectorizer()),
    ('tfidf', TfidfTransformer()),
    ('clf', ComplementNB()),
])
```

ComplementNB는 위의 MultinomialNB의 변형된 버전이다. 특히, 불균형한 data에 대해 더 좋은 성능을 보인다. 이 분류기는 각 클래스의 보완 정보를 사용해 확률을 계산한다. 이를 통해 클래스 간의 연관도를 줄인다.

ComplementNB 성능평가: 정확도 (Accuracy): 83% (400/480)

Metric	algebraic geometry	computer vision	general economics	quantitative biology	quantum physics	statistics theory	Overall
Precision	0.90	0.82	0.88	0.87	0.83	0.70	0.83
Recall	0.97	0.91	0.79	0.72	0.85	0.74	0.83
F1-Score	0.93	0.86	0.83	0.79	0.84	0.72	0.83
Support (샘플 개수)	88	80	80	75	80	77	480

4가지의 분류기를 실행해본 text를 분류하는 데 특화된 MultinomialNB와 ComplementNB가 좋은 성능을 보였다. 특히 그 중 ComplementNB가 정확도 측면에서 약 2% 높은 결과를 보였다. 따라서 MultinomialNB, ComplementNB를 사용하며 feature extraction을 개선시켜 최적의 결과를 도출하고자 한다.

2) CountVectorizer()에는 다양한 파라미터가 있다. 성능을 개선시키기 위해 파라미터들의 최고의 조합을 찾는다. Stop word는 불용어 제거 여부를 정한다. default값은 None이며 stop_words="english"로 설정할 경우 지정된 영어 불용어 목록을 제거한다. 예를 들어 영어의 경우 "the", "is", "in", "and"와 같은 단어들이 불용어에 해당한다. 불용어를 제거하면 모델이 중요한 정보에 더 집중할 수 있어 성능 향상에 도움이 될 수 있다. Ngram_range는 연속된 단어 중 함께 분석할 단어의 개수를 정한다. 이를 활용하면 text의 문맥을 보다 더 정확하게 파악할 수 있다. max_df와 min_df는 문서 빈도가 특정 임계값 이상, 이하인 단어를 제외하는 파라미터이다. 이를 통해 중요한 단어의 비중을 높이고 노이즈나 오타로 인한 단어를 제외하여 성능을 개선시킬 수 있다. max_feature는 text단어를 벡터화할 때 사용할 최대 단어 수를 지정한다. 이를 통해 계산 효율성을 향상시키고 불필요하게 많은 특징을 사용하는 것을 방지할 수 있다.

```
min_df_list = np.array([i for i in range(3, 8)])
max_df_list = [i/10 for i in range(1, 11)]
stop_word_list = {1: None, 2: 'english'}
max_feature_list = [500, 1000, 1500, 2000, 2500]
ngram_range_list = [(1, 1), (1, 2)]
classifiers = {'MultinomialNB': MultinomialNB(), 'ComplementNB':
ComplementNB()}
```

각 파라미터에 다양한 값을 검사하고 MultinomialNB, ComplementNB중 최고의 성능을 찾는다. Min_df값은 3~8, Max_df는 0.1부터 1까지, stop word는 default값과 'english'불용어를 사용해 파악한다. Max feature값은 500부터 2500까지 500단위로 파악한다. Ngram은 default값인

(1,1)을 이용해 각 단어의 분석을 하고 (1,2)를 통해 연속된 2단어까지의 분석을 실행한다.

```
for clf_name, clf in classifiers.items():
    for stop_word in stop_word_list:
        for max_feature in max_feature_list:
            for min_df in min_df_list:
                for max_df in max_df_list:
                    for ngram_range in ngram_range_list:
                        count_vect =
CountVectorizer(stop_words=stop_word_list[stop_word],
                                                max_features=max_feature,
                                                min_df=min_df, max_df=max_df,
                                                ngram_range=ngram_range)
                        clf_pipeline = Pipeline([
                            ('vect', count_vect),
                            ('tfidf', TfidfTransformer()),
                            ('clf', clf)
                        ])
                        clf_pipeline.fit(train_data.data, train_data.target)
                        predicted = clf_pipeline.predict(docs_test)
                        accuracy = np.mean(predicted == test_data.target)
                        if accuracy > max_accuracy:
                            max_accuracy = accuracy
                            max_accuracy_case = [(clf_name, stop_word, max_feature,
min_df, max_df, ngram_range)]
                        elif accuracy == max_accuracy:
                            max_accuracy_case.append((clf_name, stop_word,
max_feature, min_df, max_df, ngram_range))


print("Max accuracy:", max_accuracy)
print("Max accuracy cases:", max_accuracy_case)
```

실행 결과 max accuracy는 0.833, max accuracy cases는 다음과 같다.

Classifier	Stop Words	Max Features	Min DF	Max DF	Ngram Range
ComplementNB	2	1500	4	0.4	(1, 1)
ComplementNB	2	1500	4	0.5	(1, 1)
ComplementNB	2	1500	4	0.6	(1, 1)
ComplementNB	2	1500	4	0.7	(1, 1)
ComplementNB	2	1500	4	0.8	(1, 1)
ComplementNB	2	1500	4	0.9	(1, 1)
ComplementNB	2	1500	4	1.0	(1, 1)
ComplementNB	2	2500	7	0.3	(1, 1)
ComplementNB	2	2500	7	0.4	(1, 1)
ComplementNB	2	2500	7	0.5	(1, 1)
ComplementNB	2	2500	7	0.6	(1, 1)
ComplementNB	2	2500	7	0.7	(1, 1)
ComplementNB	2	2500	7	0.8	(1, 1)
ComplementNB	2	2500	7	0.9	(1, 1)
ComplementNB	2	2500	7	1.0	(1, 1)

2. SVM

1) SVM분류기로 SVC, NuSVC, LinearSVC 3가지가 있다. SVC는 다양한 커널 트릭을 사용해 데이터를 분류하는 분류기이다. NuSVC는 SVC와 유사하지만 새로운 파라미터가 추가되어 예측 오류의 상한선과 서포트 벡터 비율의 하한선을 제한하는 분류기이다. LinearSVC는 선형 커널을 사용하는 SVC이다. 선형 데이터에 특화 되어있고 빠른 속도를 가진다. SVC와 NuSVC는 1:1 방법론을 사용하고 LinearSVC는 1:n 방법론을 사용한다.

```
classifiers = {
    'SVC': {
        'model': SVC(),
        'params': {
            'C': [0.1, 1, 10],
            'kernel': ['linear', 'rbf'],
            'gamma': ['scale', 'auto']
        }
    },
    'NuSVC': {
        'model': NuSVC(),
        'params': {
            'nu': [0.1, 0.5, 0.9],
            'kernel': ['linear', 'rbf'],
            'gamma': ['scale', 'auto']
        }
    },
    'LinearSVC': {
        'model': LinearSVC(max_iter=10000),
        'params': {
            'C': [0.1, 1, 10]
        }
    }
}
```

3가지의 분류기 중 최고의 성능을 내는 분류기와 파라미터를 찾는다. 사용한 파라미터는 다음과 같다. SVC()분류기는 c, kernel, gamma를 사용하였다. c는 학습 오류와 모델 복잡도 간의 균형을 조절하는 파라미터이다. 이 파라미터는 값이 커질수록 더 엄격해져서 학습오류를 줄이는 방향으로 작용한다. C로 0.1, 1, 10을 사용한다. Kernel는 사용할 커널의 유형을 지정한다. 커널로 linear, rbf를 사용한다. Gamma는 커널의 계수를 정한다. 이 값이 클수록 결정 결계가 더 복잡해진다. NuSVC의 파라미터로는 nu, kernel, gamma를 사용한다. Nu는 c와 유사하지만 서포트 벡터의 비율과 예측 오류를 제한한다. LinearSVC의 파라미터로는 c, max_iter을 사용한다. Max_iter는 허용되는 반복횟수를 제한한다.

```
for clf_name, clf_dict in classifiers.items():
    model = clf_dict['model']
    params = clf_dict['params']


    if clf_name == 'SVC':
        for C in params['C']:
            for kernel in params['kernel']:
                for gamma in params['gamma']:
                    clf_params = {'C': C, 'kernel': kernel, 'gamma': gamma}
                    clf_pipeline = Pipeline([
```

```
                    ('vect', CountVectorizer()),
                    ('tfidf', TfidfTransformer()),
                    ('clf', model.set_params(**clf_params))
                ])
                clf_pipeline.fit(train_data.data, train_data.target)
                predicted = clf_pipeline.predict(docs_test)
                accuracy = np.mean(predicted == test_data.target)
                if accuracy > max_accuracy:
                    max_accuracy = accuracy
                    max_accuracy_case = [(clf_name, clf_params)]
                elif accuracy == max_accuracy:
                    max_accuracy_case.append((clf_name, clf_params))
    elif clf_name == 'NuSVC':
        for nu in params['nu']:
            for kernel in params['kernel']:
                for gamma in params['gamma']:
                    clf_params = {'nu': nu, 'kernel': kernel, 'gamma': gamma}
                    clf_pipeline = Pipeline([
                        ('vect', CountVectorizer()),
                        ('tfidf', TfidfTransformer()),
                        ('clf', model.set_params(**clf_params))
                    ])
                    clf_pipeline.fit(train_data.data, train_data.target)
                    predicted = clf_pipeline.predict(docs_test)
                    accuracy = np.mean(predicted == test_data.target)
                    if accuracy > max_accuracy:
                        max_accuracy = accuracy
                        max_accuracy_case = [(clf_name, clf_params)]
                    elif accuracy == max_accuracy:
                        max_accuracy_case.append((clf_name, clf_params))
    elif clf_name == 'LinearSVC':
        for C in params['C']:
            clf_params = {'C': C}
            clf_pipeline = Pipeline([
                ('vect', CountVectorizer()),
                ('tfidf', TfidfTransformer()),
                ('clf', model.set_params(**clf_params))
            ])
            clf_pipeline.fit(train_data.data, train_data.target)
            predicted = clf_pipeline.predict(docs_test)
            accuracy = np.mean(predicted == test_data.target)
            if accuracy > max_accuracy:
                max_accuracy = accuracy
                max_accuracy_case = [(clf_name, clf_params)]
            elif accuracy == max_accuracy:
                max_accuracy_case.append((clf_name, clf_params))


print("\nMax accuracy:", max_accuracy)
print("Max accuracy cases:", max_accuracy_case)
```

실행 결과 max accuracy는 0.8458, max accuracy cases는 다음과 같다.

Classifier	C	Nu	Kernel	Gamma
SVC	10		Linear	Scale
SVC	10		Linear	Auto
NuSVC		0.1	Linear	Scale
NuSVC		0.1	Linear	auto

SVC(c=10, kernel=linear), NuSVC(Nu=0.1, kernel=linear)이 가장 좋은 성능을 보인다. 이 두가지를 countvectorize, transformer의 다양한 파라미터를 변환시키며 가장 좋은 성능을 찾는다.

```
min_df_list = np.array([i for i in range(3, 8)])
max_df_list = [i/10 for i in range(1, 11)]
stop_word_list = {1: None, 2: 'english'}
max_feature_list = [500, 1000, 1500, 2000, 2500]
ngram_range_list = [(1, 1), (1, 2)]
```

NB와 동일하게 각 파라미터에 다양한 값을 부여하고 SVC(C=10, kernel='linear'), NuSVC(nu=0.1, kernel='linear')의 최고의 성능을 찾는다.

```
for clf_name, clf in classifiers.items():
    for stop_word in stop_word_list:
        for max_feature in max_feature_list:
            for min_df in min_df_list:
                for max_df in max_df_list:
                    for ngram_range in ngram_range_list:
                        count_vect =
CountVectorizer(stop_words=stop_word_list[stop_word],
                                                max_features=max_feature,
                                                min_df=min_df, max_df=max_df,
                                                ngram_range=ngram_range)
                        clf_pipeline = Pipeline([
                            ('vect', count_vect),
                            ('tfidf', TfidfTransformer()),
                            ('clf', clf)
                        ])
                        clf_pipeline.fit(train_data.data, train_data.target)
                        predicted = clf_pipeline.predict(docs_test)
                        accuracy = np.mean(predicted == test_data.target)
                        accuracies.append((clf_name, stop_word_list[stop_word],
max_feature, min_df, max_df, ngram_range, accuracy))
                        if accuracy > max_accuracy:
                            max_accuracy = accuracy
                            max_accuracy_case = [(clf_name,
stop_word_list[stop_word], max_feature, min_df, max_df, ngram_range)]
                        elif accuracy == max_accuracy:
                            max_accuracy_case.append((clf_name,
stop_word_list[stop_word], max_feature, min_df, max_df, ngram_range))

print("\nMax accuracy:", max_accuracy)
print("Max accuracy cases:", max_accuracy_case)
```

실행 결과 max accuracy는 0.83125, max accuracy cases는 다음과 같다.

Classifier	Stop word	Max feature	Min df	Max df	ngram range
SVC	None	2500	4	1.0	(1,1)
NuSVC	None	2500	4	1.0	(1,1)

(P2-1) 논문 초록 분류

Clustering은 분류기와 같은 데이터를 사용하여 가장 성능이 좋은 군집화 모델을 구현하는 것을 목표로 한다.

여러 방법 중 K-means clustering을 사용하고, 평가 방법으로는 Homogeneity와 completeness의 조화 평균인 V-measure을 사용하도록 한다.

1) 우선 군집의 개수를 설정한다.

```
n_clusters_range = range(3, 10)
n_init_range = [5 * i for i in range(1, 11)]
```

주어진 데이터의 카테고리가 6개이므로 군집의 개수를 3개부터 9개까지, 초기 클러스터 중심을 설정하는 반복 횟수인 n_init을 5부터 50까지 설정하여 최적의 군집 개수와 반복 횟수를 찾는다.

```
for n_clusters, n_init in product(n_clusters_range, n_init_range):
    v_measure_sum = 0
    n_iter = 20
    for i in range(n_iter):
        clst = KMeans(n_clusters=n_clusters, n_init=n_init, init='k-
means++', random_state=42)
        clst.fit(data_trans)
        v_measure = metrics.v_measure_score(data.target, clst.labels_)
        v_measure_sum += v_measure

    avg_v_measure = v_measure_sum / n_iter

    if avg_v_measure > best_v_measure:
        best_v_measure = avg_v_measure
        best_params = {
            'n_clusters': n_clusters,
            'n_init': n_init
        }


print(f"최적의 V-measure: {best_v_measure}")
print(f"최적의 파라미터: {best_params}")
```

KMeans 클러스터링은 초기 중심의 설정에 따라 결과가 달라질 수 있다. 따라서 n_iter을 20 으로 설정해 각 파라미터 값에 따라 20번을 반복하고 평균 V-measure을 구한다.

그 결과 최적의 V-measure: 0.5187804300055878, 최적의 파라미터: {'n_clusters': 6, 'n_init': 10}임을 얻었다.

2) 이제 군집 6개와 n_init: 10인 군집화를 특징 추출기의 파라미터를 바꿔가며 개선시킨다.

```
max_features_range = [500, 1000, 1500, 2000, 2500]
min_df_range = [1, 2, 3, 4, 5, 6]
max_df_range = [0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0]
```

stopword='english'로 설정하고 위의 파라미터로 최고의 성능을 보이는 군집화 모델을 찾는다.

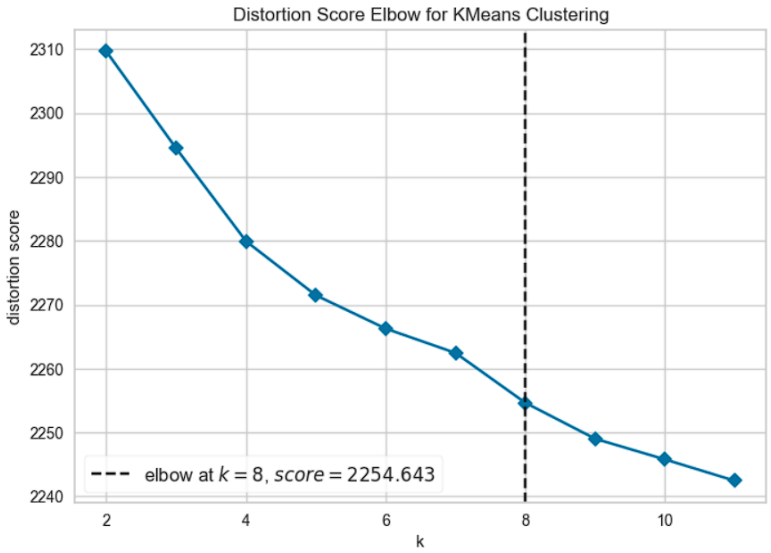
```
n_clusters = 6
n_init = 10


for max_features in max_features_range:
    for min_df in min_df_range:
        for max_df in max_df_range:
            v_measure_sum = 0
            n_iter = 20
            for i in range(n_iter):
                count_vect = CountVectorizer(stop_words='english',
max_features=max_features, min_df=min_df, max_df=max_df)
                data_counts = count_vect.fit_transform(data.data)
                tfidf_transformer = TfidfTransformer()
                data_trans = tfidf_transformer.fit_transform(data_counts)
```

```
            clst = KMeans(n_clusters=n_clusters, n_init=n_init, init='k-
means++', random_state=42)
            clst.fit(data_trans)
            v_measure = metrics.v_measure_score(data.target,
clst.labels_)
            v_measure_sum += v_measure

        avg_v_measure = v_measure_sum / n_iter

        if avg_v_measure > best_v_measure:
            best_v_measure = avg_v_measure
            best_params = {
                'max_features': max_features,
                'min_df': min_df,
                'max_df': max_df
            }


print(f"최적의 V-measure: {best_v_measure}")
print(f"최적의 특징 추출기 파라미터: {best_params}")
```

그 결과 최적의 V-measure: 0.5392807569161527, 최적의 특징 추출기 파라미터: {'max_features': 2000, 'min_df': 3, 'max_df': 0.4}임을 알 수 있다.

3) Elbow Method를 사용해 최적의 클러스터링 수를 찾는다. Elbow Method는 클러스터 수에 따른 비용 함수(WCSS: Within-Cluster Sum of Squares)를 그래프로 나타낸다. 클러스터 수가 증가할수록 WCSS 값은 감소하지만, 어느 시점 이후로 감소율이 급격히 완만해진다. 이 시점을 "elbow"라고 하며, 이는 최적의 클러스터 수를 보인다.

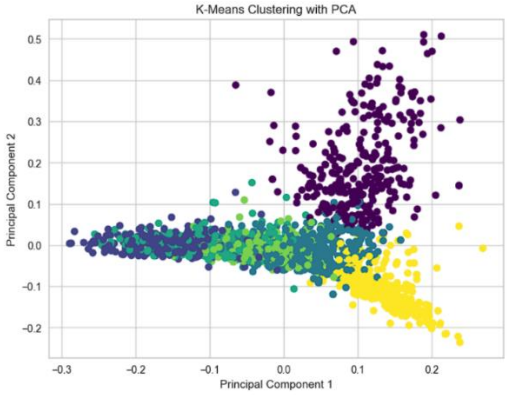
```
# TODO - Find the appropriate number of clusters
model = KMeans(init='k-means++', random_state=42)
visualizer = KElbowVisualizer(model, k=(2, 12), timings=False)

visualizer.fit(data_trans.toarray()) #Fit the data to the visualizer
visualizer.show()
```

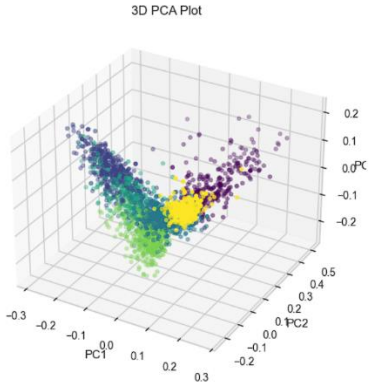


Elbow Method의 k=8 즉, 최적의 클러스터 수가 8개로 식별되었다. 이는 전에 구한 최적의 클러스터 수인 6과 다르다. Elbow Method를 사용하는 것은 하나의 방법이지만, 항상 최적의 성능을 보장하지는 않는다. Elbow Method는 관성(WCSS)을 기준으로 최적의 클러스터 수를 찾으려

는 시도이기 때문에 v-measure을 기준으로 따져 본 6개의 클러스터가 더 좋은 성능을 보인다고 할 수 있다. 실제로 주어진 data의 카테고리가 6개이므로 클러스터의 수가 6개인 것이 합리적이다.

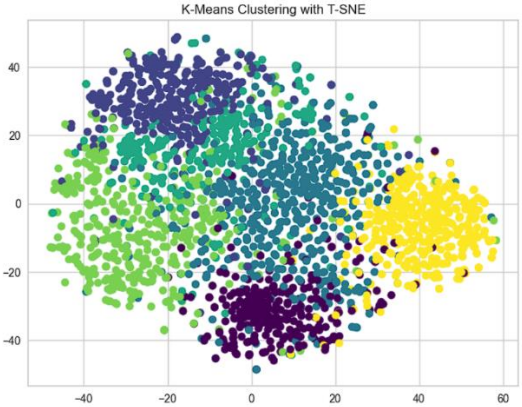
4) 찾은 최적의 클러스터링의 결과를 시각화하기 위해 PCA를 사용한다. 이는 고차원의 데이터를 저차원으로 변환하여 주요 특성은 유지하면서 데이터를 시각화 한다. 이 시각화의 기준은 분산이다. 각 축은 데이터의 가장 큰 분산을 설명한다. 이 시각화는 가능한 많은 분산을 유지하도록 한다.

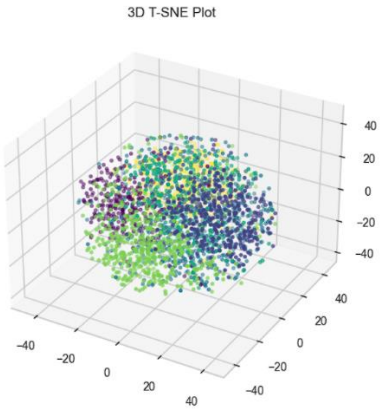




그래프에서 색상별로 명확하게 구분되는 영역은 이 클러스터링이 잘 작동해 유의미하게 군집화를 했음을 알려준다. 그 예로 2차원 시각화의 오른쪽 아래 노란색 클러스터는 다른 클러스터와 명확하게 구분되었음을 알 수 있다. 하지만 2차원 시각화에서 중앙부는 다양한 색이 섞여 있다. 이는 2차원에서는 시각적인 구별이 불가능 하다는 의미이다. 3차원의 시각화를 보면 2차원과는 다르게 그래프의 중앙부분이 구분 가능하다. 즉, 차원 수가 높아지면 분류의 기준이 늘어 데이터의 군집화를 더 잘 관찰할 수 있다. 이 방법은 고차원인 데이터가 저차원으로 축소되었기에 일부 정보가 손실될 수 있다. 하지만 데이터의 주요 분포를 시각적으로 확인할 수 있다.

5) T-SNE는 PCA와 같이 크러스터링의 결과를 시각화해준다. 이 방법은 고차원 데이터의 비선형 구조와 데이터 포인트 간의 국소적인 유사성을 보존하면서 저차원으로 축소한다. 분산을 최대화하는 PCA와는 다르게 데이터간의 유사성을 확률로 표현하고 고차원과 저차원의 이 확률 분포 차이를 최소화하는 방식이다. 즉, 국소적인 구조를 잘 보존한다.





이 시각화 역시 크러스터링의 결과를 잘 보여준다. PCA와 마찬가지로 2차원에서 노란색처럼 확실하게 다른 데이터와 구분되는 부분이 있다. 하지만 3차원은 PCA와 다르게 2차원보다 구분이 어렵다. 이는 두 시각화의 방법 차이 때문이다. PCA는 데이터의 전역적인 구조와 주요 분산을

보존하여 시각화를 하는 반면, t-SNE는 데이터의 국소적인 유사성을 보존하는 방법이다. 따라서 PCA는 특정 데이터끼리의 차이와 유사도보다는 전반적인 데이터들의 구조를 더욱 잘 분류하고 t-SNE는 특정 데이터끼리의 유사성을 더욱 잘 보여준다. 이는 결과로도 잘 나타난다. PCA의 결과는 클러스터마다 시각적으로 구분되는 정도의 차이가 있지만 t-SNE는 모든 클러스터가 비슷하게 구분된다. 따라서 각 기법은 사용 목적과 데이터의 특성에 따라 선택하여 사용하는 것이 중요하다.