데이터관리와 분석 Project #3

Document search engine & Classification and Clustering

11조

2017-13918 황재훈
2017-10854 최의현
2017-18868 조형찬
2017-15751 장준혁
2017-10089 윤성진

<목차>

PART 1. 문서 검색 엔진

1-1. CustomerScoring 함수 변경

CustomerScoring 함수를 변경하기 전, document와 query의 stopwords 제외하고 기본값으로 설정되어 있는 BM25 함수를 사용하였을 때의 성능 값은 0.18430476481540503임을 확인하였다.

```
return idf * ((tf * (K1 + 1)) / (tf + K1 * ((1 - B) + B * fl / avgfl)))
```

이후, BM25 알고리듬에서 경험적으로 결정하는 parameter인 K1과 B를 변경하며 최적의 성능값 결과를 보이는 K1과 B를 설정하였다. 이 때, K1의 변동성이 B의 변동성보다 큰 경향을 보여 K1을 0.01씩 B을 0.01씩 증가시켰고, B=0.4, K1=0.13 일 때, 최댓값 0.19029602494615622을 얻어, B=0.4, K1=0.13으로 설정하였다.

다른 heuristic한 방식으로 customerscoring 함수를 변경해 보고자 document의 각 word의 term frequency를 분석해보았고, 빈도수가 높은 순으로 정렬했을 때, 아래의 그림과 같이 분석 되었다.(아래의 예시는 결과의 일부분이다.)

```
Counter({'': 140037, 'n': 73527, 'the': 7229, 'of': 4955, 'us': 2543, '
circuit': 2348, 'frequ': 2323, 'field': 2317, 'and': 2224, 'electron':
2173, 'vary': 1764, 'magnet': 1758, 'in': 1618, 'is': 1566, 'method': 1
566, 'effect': 1523, 'ampl': 1519, 'giv': 1444, 'two': 1386, 'high': 13
35, 'are': 1307, 'wav': 1295, 'describ': 1294, 'on': 1272, 'result': 12
54, 'observ': 1229, 'radio': 1217, 'to': 1154, 'design': 1143, 'system'
: 1085, 'analys': 1084, 'meas': 1066, 'oscil': 1062, 'ionosph': 1052, '
for': 1039, 'low': 1038, 'ion': 1036, 'with': 991, 'apply': 983, 'show'
: 976, 'typ': 964, 'obtain': 895, 'equ': 878, 'network': 878, 'tim': 87
3, 'gen': 863, 'discuss': 860, 'cur': 857, 'puls': 822, 'lay': 809, 'ma
k': 798, 'sol': 796, 'rel': 795, 'reg': 785, 'bas': 780, 'volt': 775, '
op': 731, 'transist': 725, 'an': 717, 'nois': 708, 'der': 697, 'mod': 6
97, 'by': 696, 'stat': 690, 'energy': 685, 'calc': 680, 'charact': 679,
'expery': 674, 'output': 673, 'sign': 673, 'valu': 668, 'funct': 658, '
pow': 653, 'at': 650, 'rady': 649, 'filt': 648, 'height': 643, 'band':
642, 'distribut': 629, 'diff': 627, 'dens': 626, 'ear': 621, 'also': 61
6, 'elect': 613, 'discussed': 609, 'consid': 607, 'resist': 607, 'deter
min': 605, 'part': 599, 'frequency': 598, 'form': 590, 'described': 582
, 'stabl': 578, 'phas': 578, 'pres': 575, 'reson': 573, 'temp': 572, 'm
ay': 569, 'plasm': 556, 'control': 553, 'emit': 552, 'input': 545, 'atm
```

분석 결과 doc word의 빈도수 그래프는 term frequency=1인 word가 다수를 차지하는, long tail 현상을 보이는 그래프를 띠었고, 이를 통해, idf의 비중을 높이면 결과값이 증가할 것이라고 가정하고, idf^parameter를 해주었고 1부터 0.1씩 증가시키며 확인한 결과 parameter=1.7일 때까지, 결과 값이 증가하는 것을 확인하였고 따라서, 아래의 그림과 같은 경우에서, B=0.4, K1=0.13로 설정된 Scoring 함수를 사용하기로 하였다..

```
return (idf**1.7) * ((tf * (K1 + 1)) / (tf + K1 * ((1 - B) + B * fl / avgfl)))
```

1-2. OrGroup 변수 변경

변수 값을 설정하였을 때, 성능이 상승했으나, [0,1] 사이에서 변수를 증가시켰을 때, 약간의 증가하는 경향을 보이는 것을 확인했고, 따라서 [0.1] 사이 수 중 충분히 큰 수 0.999를 사용하였다.

1-3. 문장 부호 제거

document term frequency 분석에서 ' '의 빈도수가 가장 많은 것을 확인하였고, Stemming, Lemmaztizing 등의 다양한 processing을 하기 전에 문장 부호를 제거하였을 때, 성능이 개선된다면 문장 부호를 제거한 후 processing을 진행하였을 때에도 긍정적인 결과 값을 얻을 수 있을 것이라는 가정하에 문장 부호 제거를 실행해주었다. 이 때, query와 document 모두에 같은 처리를 해주어야 긍정적인 결과를 얻을 수 있을 것이라고 가정하였고 그 결과 성능이 0.21589834835284644로 상승하였고 이후 과정에서 문장 부호를 제거한 후 추가적인 processing을 실행하였다. 아래의 그림은 document와 query의 문장 부호를 제거 해주는 코드이다.

```
table = str.maketrans('\n?.,!', '      ')
doc_text = doc_text.translate(table)

new_doc_text = ''
for word in doc_text.split(' '):
    if word.lower() not in stopWords:
        # word = n.lemmatize(word)
        # word_stem = stemmizer.stem(word.lower())
        # new_doc_text += word_stem + ' '
        new_doc_text += word + ' '
```

```
for qid, q in query_dict.items():
    table = str.maketrans('\n?.,!', '      ')
    q = q.translate(table)
    new_q = ''
```

1-4. Stemming 및 Lemmatizing 적용

먼저 Stemming을 실행하였다. stemming 함수는 널리 쓰이는 PorterStemmer, SnowballStemmer, LancasterStemmer 3가지를 이용하였다. stemming 역시, 각 stemmer 함수를 사용할 때, document와 query에 동일하게 적용해 주었고, 문장 부호 제거를 사전에 실시한 후 진행하였다. 그 결과 3가지 모두의 경우에서 성능이 개선되는 것을 확인할 수 있었고, 그 중 LancasterStemmer를 사용한 경우가 가장 성능 값이 좋아, LancasterStemmer를 사용하기로 결정하였다.

이후, Lemmatizing을 진행하였다. Lemmatizing을 위해서 WorldNetLemmatizer를 이용하였다. 이 경우, 사전에 문장 부호 처리를 실시해준 뒤, stemming을 하지 않은 경우, 3가지의 stemming을 한 경우, 총 4가지 경우에 각각 lemmatizing을 document와 query에 동일하게

해준 뒤, 결과를 비교해 보았다. 그 결과 LancasterStemmer와 WorldNetLemmatizer를 같이 사용한 경우가 성능 값이 가장 좋게 나오는 것을 확인하였고 LancasterStemmer와 WorldNetLemmatizer를 같이 사용하기로 결정하였다. 아래의 그림은 Stemming 과 Lemmatizing을 처리해주는 코드이다.

```
n = WordNetLemmatizer()
st3 = LancasterStemmer()
```

```
new_doc_text = ''
for word in doc_text.split(' '):
    if word.lower() not in stopWords:
        word = n.lemmatize(word.lower())
        word_stem = st3.stem(word)
        new_doc_text += word_stem + ' '
```

```
n = WordNetLemmatizer()
#s1 = PorterStemmer()
#s2 = SnowballStemmer('english')
s3 = LancasterStemmer()
```

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

        word = n.lemmatize(word.lower())
        #new_q += word + ' '
        new_q += s3.stem(word) + ' '
```

1-5. Query Expansion

Stemming 과 Lemmatizing 처리 후 query expansion을 적용하여 보았다. query expansion 은 강의 자료에 소개되어있는 유사어 사전인 nltk의 wordnet을 사용해보았다. 휴리스틱하게 가장 유사한 단어 한 개, 가장 유사한 단어 3개, 유사 순위 3,4,5위 추가 등 여러 가지 옵션으로 실험해보았지만 성능의 향상은 없었다. 따라서 query expansion 기능은 적용하지 않도록 하였다.

```
syns = wordnet.synsets(s3.stem(word).lower())
if len(syns) > 4:
    # new_q += syns[].lemmas()[0].name() + ' '
    # new_q += syns[2].lemmas()[0].name() + ' '
    new_q += s3.stem(n.lemmatize(syns[1].lemmas()[0].name())) + ' '
    new_q += s3.stem(n.lemmatize(syns[2].lemmas()[0].name())) + ' '
    new_q += s3.stem(n.lemmatize(syns[3].lemmas()[0].name())) + ' '
```

1-6 결과

앞에서의 내용을 정리한 최종 결과로, document와 query에 stopword 및 문장 부호를 제거한 뒤, LancasterStemmer와 WorldNetLemmatizer를 이용한 processing을 거친 후, OrGroup 변수를 0.999로 설정하고 아래의 그림과 같은 BM25 함수의 idf에 parameter를 제곱하여 idf의 비중을 높게한 함수를 이용한 모델을 사용한다.

```
return ((idf**parameter) * ((tf * (K1 + 1)) / (tf + K1 * ((1 - B) + B * fl / avgfl))))
```

이 때, processing을 거치며, 최적 성능을 산출하는 기존에 설정한 함수의 변수 idf의 parameter, B, K1이 달라졌을 수 있기 때문에, 한번 더 변수의 숫자를 변경하며 조정해 주었고, idf의 parameter=1.5, B=0.5 ,K1=0.13인 모델을 최종 사용하여 그 때의 성능 값이 0.25908098603535284로 최대임을 확인하였다.

PART 2. 문서 분류 및 군집화

2-1. 영어 신문 기사 분류

2-1-1. naive bayes

```
categories = ['Business', 'Entertainment', 'Living', 'Metro', 'Shopping', 'Sports', 'Tech']

train_data = load_files(container_path='text/train', categories=categories, shuffle=True,
                        encoding='utf-8', decode_error='replace')

# TODO - 2-1-1. Build pipeline for Naive Bayes Classifier
clf_nb = Pipeline([
                ('vect', CountVectorizer(lowercase=True, stop_words='english', ngram_range=(1,1))),
                ('tfidf', TfidfTransformer(use_idf=True, smooth_idf=True, sublinear_tf=False)),
                ('clf', MultinomialNB(alpha=0.01))
                ])
clf_nb.fit(train_data.data, train_data.target)
```

Naive bayes의 pipeline이다.

CountVectorize에서 lowercase는 True로 default값으로 사용하였다. stop_words는 'english'로 stopping하였으며, ngarme_range=(1,1)로, unigram을 사용하였다. tokenization도 none으로 사용하지 않았다. 기타 min_df, max_df 등도 default값으로 유지하였다.

TfidTransformation에서 us_idf, smooh_if, sublinear_tf 모두 default값을 유지하였다.

성능 개선을 위해 다음과 같은 것들을 시도해 보았다.
-Naive bayers Classifier에서 –Gaussian NB, Bernoulli NB를 각각 적용하여 성능을 비교해보았다.
-Multinomial NB에서는 alpha 값을 바꾸어가며 성능을 비교해보았다.

결과적으로 Multinomial NB, alpha=0.01에서 33/40개를 올바르게 분류하는 것을 보았으며, 정확도가 가장 높았다.

2-1-2. svm

```
# TODO - 2-1-2. Build pipeline for SVM Classifier
clf_svm = Pipeline([
                    ('vect', CountVectorizer(lowercase=True, stop_words='english', ngram_range=(1,1))),
                    ('tfidf', TfidfTransformer(use_idf=True, smooth_idf=True, sublinear_tf=False)),
                    ('clf', LinearSVC(C=1.0, random_state=0))
                    ])
clf_svm.fit(train_data.data, train_data.target)
```

clf_svm의 pipeline이다

결과적으로, LinearSVC, C=1.0에서 35/40개를 올바르게 분류하여 이 classfier를 선택하게 되었다.

성능 개선을 위해 다음과 같은 것들을 시도해 보았다.

-SVC, NuSVC, LinearSVC 각각 적용하여 성능을 비교해보았다.
-SVC에서 C(Regularization parameter)값과 kernel(linear, poly, rbf, sigmoid)를 바꾸어가며 성능을 비교해보았다.
-NuSVC에서 kernel을 바꾸어가며 성능을 비교해 보았다.
-LinearSVC에서 C 값을 바꾸어가며 성능을 비교해 보았다.

이런 것들을 모두 시도해봤을 때, 결과적으로 가장 성능이 좋은 모델은 위에서 언급한 LinearSVC, C=1.0이었다.

모델을 선택할 때 추가적으로 고려한 상황은 다음과 같다.

-정확도가 35/40개로 평가된 방식은 이외에도 몇 개의 모델이 더 있었지만, 다른 기사에 대해서도 좀 더 일관적으로 높은 정확도를 가질 수 있을 것 같은 방식을 채택하였다.
-TruncatedSVD를 적용하여 보았으나 정확도가 35/40보다 높아지는 것들을 찾지 못하였다.
(차원 축소를 시도했지만, 성능 평가에서 성능이 유의미하게 향상되지는 않았다.)

```
test_data = load_files(container_path='text/test', categories=categories, shuffle=True,
                       encoding='utf-8', decode_error='replace')
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))

predicted1 = clf_svm.predict(docs_test)
print("SVM accuracy: %d / %d" % (np.sum(predicted1 == test_data.target), len(test_data.target)))
# print(metrics.classification_report(test_data.target, predicted1, target_names=test_data.target_names))
# print(metrics.confusion_matrix(test_data.target, predicted1))
```

위 모델을 토대로 Naive bayes의 정확도와 SVM 정확도를 계산해보았다.

```
TEAM = 11

with open('DMA_project3_team%02d_nb.pkl' % TEAM, 'wb') as f1:
    pickle.dump(clf_nb, f1)

with open('DMA_project3_team%02d_svm.pkl' % TEAM, 'wb') as f2:
    pickle.dump(clf_svm, f2)
```

마지막으로 이를 pkl형태로 저장하였다.

2-2. 영어 신문 기사 군집화

2-2-1. 기본 설명

영어 신문 기사를 군집화 하는데 있어서는 K-means Clustering을 사용할 것이다. K-means Clusltering은 n 개의 데이터가 주어졌을 때 이를 k개의 그룹으로 나눈 후에 이 각각 그룹을 클러스터로 형성하는 방식이다. 그룹은 비용함수를 minimize 하는 방식으로 이루어진다.

이 프로젝트에서 성능을 판단하는데 있어서 v-measure를 사용할 것이다. v-measure은 homogeneity와 compeleteness의 조화 평균값이다.

여기서, homegeneity는 클러스터 내의 동질성을 나타내는 지표이고, completeness는 범주의 데이터 점들이 같은 클러스터 내에 있는 것을 나타낸다.

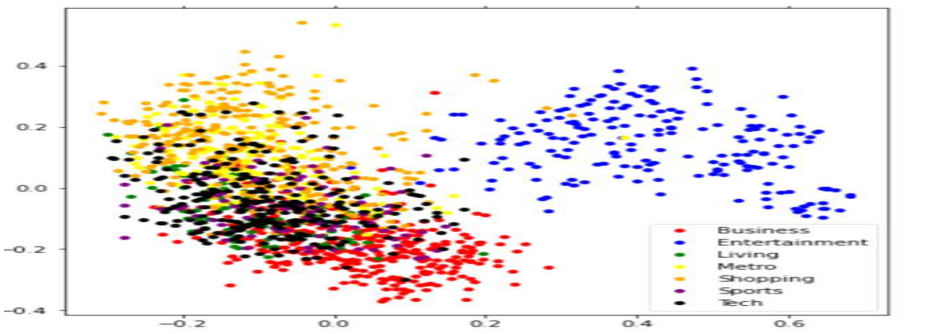
2-2-2. 초기 모델 분석

```
from sklearn.datasets import load_files
from sklearn.feature_extraction.text import CountVectorizer, TfidfTransformer
from sklearn import metrics
from sklearn.cluster import KMeans
from nltk.corpus import stopwords
from sklearn.decomposition import TruncatedSVD
from sklearn.preprocessing import Normalizer
from sklearn.pipeline import make_pipeline

categories = ['Business', 'Entertainment', 'Living', 'Metro', 'Shopping', 'Sports', 'Tech']

data1 = load_files(container_path='C:/Users/User/Downloads/DMA_project3/DMA_project3/CC/text_all', categories=categories, shuffle=True,
                   encoding='utf-8', decode_error='replace')

# TODO - Data preprocessing and clustering
data_trans=TfidfTransformer().fit_transform(CountVectorizer().fit_transform(data1.data))

clst1=KMeans(n_clusters=7,random_state=0)
clst1.fit(data_trans)
print(metrics.v_measure_score(data.target, clst1.labels_))
```

우선 초기 모델이다. 초기 모델의 v-measure를 계산하였더니 다음과 같은 결과가 나왔다..

0.43577680136701025

다음은 2차원 평면상에 clustering한 결과를 확인해 보자.



위 그림은 초기모델을 pca를 통해 차원을 축소 시킨 후 2차원 평면상에 clustering을 표현한 것이다. pca 함수는 sparse matrix를 input를 받지 않기 때문에 기존의 data_trans 객체에 대해 svd를 집행하였다.

v-measure를 측정하는데 있어서 중요한 척도인 completeness와 homogeneity두 가지 요소로 위의 결과를 분석해보면, competeness에 있어서는, 같은 색들의 데이터들이 대략적으로 근접하고 있는 모습을 볼 수가 있었다. 하지만, homogeneity 측면에서 결과를 분석해보면, 색들이 매우 많이 겹치고 있는 모습을 볼 수가 있었다. 따라서, 군집 간 구별이 잘 되지 않은 모습을 볼 수 있었다.

이제 다양한 요소들을 사용하여서 모델을 개선 시킬 것이다.

2-2-3. 모델 개선

1. CountVectorizer 수정

```
from nltk.corpus import stopwords

stop_words=set(stopwords.words('english'))
count_vec=CountVectorizer(stop_words=stop_words,min_df=10, max_df=600)
vec_data =count_vec.fit_transform(data.data)
data_tf  =TfidfTransformer().fit_transform(vec_data)
```

첫 번째로는, CountVectorizer부분에서 성능을 개선 시킨 것이다.

-stopwords-'english'로 설정

영어에서 대명사, 관사와 같은 단어들은 문서의 구분과 상관없이 많이 나온다. 따라서 이러한 단어들은 유의미하지 않으며 토큰화시키지 않는 것이 좋다. NTLK 라이브러리에서 'english'를 이용하면 대명사, 관사를 stopwords로 처리하기 때문에 위와 같이 설정하였다.

-min_df=10, max_df=600

위 설정은 문서 10개 미만에서 나오는 단어와, 문서 600개 이상에서 나오는 단어를 삭제시키는 것이다. 텍스트 파일을 직접 확인해본 결과를 바탕으로, 텍스트가 잘리는 경우를 최대한 없애기 위하여 휴리스틱하게 범위를 넓게 잡았다.

-tf, idf vectorize

마지막으로- tf,idf vectorize과정을 거쳤다. tf는 (term frequency)로 해당 단어가 몇 번 나오는지 count하는 것이다. idf는 희귀하게 나오는 단어(더 의미 있는 단어)에 가중치를 주는 것으로, 이 두 장치를 도입하였다.

2. svd normalizer 사용

```
data_trans
```

tf, idf를 도입한 후 data_trans를 이용하여 tf, idf벡터 matrix를 확인해 보았다.

```
<1501x32932 sparse matrix of type '<class 'numpy.float64'>'
        with 346257 stored elements in Compressed Sparse Row format>
```

위와 같은 결과가 나왔으며, tf, idf벡터는 굉장히 sparse한 matrix인 것을 확인할 수 있었다.

- sparse한 matrix를 다룰 때는 svd가 효과적이다. python에서는 조각난 svd(truncated svd)를 이용하여 연산량과 성능 사이에서 trade off 하는 모습을 보여준다.

tf, idf는 단어의 빈도수를 이용한다. 따라서, 단어에 내재되어 있는 잠재적인 의미를 고려하지 못한다는 단점이 있다. 그 단점을 커버하기 위하여 lsa(latent semantic analysis)를 사용하였다. lsa는 의미가 비슷한 것들은 유사하다고 판단할 수 있게 도와주어 토픽 모델링에 적합하다.

pipeline에 집어넣을 때는 truncated svd를 normalize해서 학습시키면 좀 더 나은 성능을 보이기 때문에 normalizer를 사용하였다.

```
from sklearn.decomposition import TruncatedSVD
from sklearn.preprocessing import Normalizer
from sklearn.pipeline import make_pipeline

svd = TruncatedSVD(n_components=100, random_state=44)
normalizer = Normalizer(copy=False)
lsa = make_pipeline(svd,normalizer)
data_lsa=lsa.fit_transform(data_tf)
```

3. random_state값 조정

Svd를 생성할 때 random_state 값에 따라 성능 값이 다르게 나오는 것을 알고 있다. (random_state는 데이터를 training과 test data로 나누기 때문이다.) 따라서 휴리스틱하게 random_state의 값을 바꿔가며 성능 확인을 하였다. 그 결과 random_state=44일 때, v-measure이 65.1퍼센트가 되었고 가장 좋은 성능을 보이는 것을 알 수 있었고, random_state=44를 사용하였다.

```
from sklearn.datasets import load_files
from sklearn.feature_extraction.text import CountVectorizer, TfidfTransformer
from sklearn import metrics
from sklearn.cluster import KMeans
from nltk.corpus import stopwords
from sklearn.decomposition import TruncatedSVD
from sklearn.preprocessing import Normalizer
from sklearn.pipeline import make_pipeline

categories = ['Business', 'Entertainment', 'Living', 'Metro', 'Shopping', 'Sports', 'Tech']

data = load_files(container_path='C:/Users/User/Downloads/DMA_project3/DMA_project3/CC/text_all', categories=categories, shuffle=True,
                  encoding='utf-8', decode_error='replace')

# TODO - Data preprocessing and clustering
stop_words=set(stopwords.words('english'))
count_vec=CountVectorizer(stop_words=stop_words,min_df=10, max_df=600)
vec_data =count_vec.fit_transform(data.data)
data_tf  =TfidfTransformer().fit_transform(vec_data)

svd = TruncatedSVD(n_components=100, random_state=44)
normalizer = Normalizer(copy=False)
lsa = make_pipeline(svd,normalizer)
data_lsa=lsa.fit_transform(data_tf)
clst = KMeans(n_clusters=7, random_state=0)
clst.fit(data_lsa)
print(metrics.v_measure_score(data.target, clst.labels_))
```

(random_state=44의 모습)

4. KMeans 여러 attribute 조정

KMeans cluster를 생성한 후, lsa로 학습시킨 data로 비교해 학습을 진행하였다. KMeans함수의 여러 가지 attribute를 조정해봤으나 기본값이 가장 좋은 성능을 보여주었다.

2-2-4. 수정한 모델 확인

따라서 수정한 모델은 stopword removal로 문서를 전처리 한 후, svd를 거쳐서 lsa를 하였다.

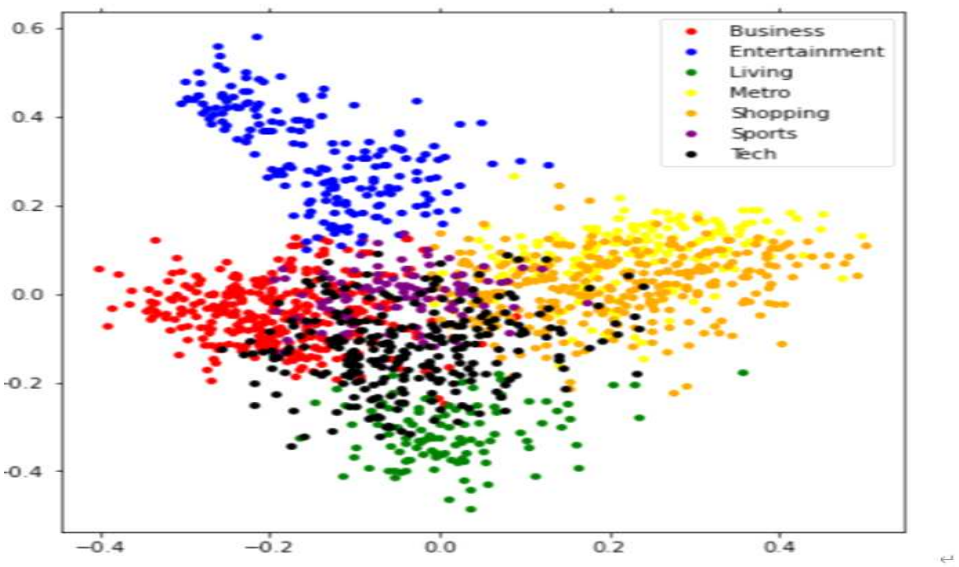
우선 v-measure관점에서 보면 초기값 43.6%에서 65.1%로 개선 시킨 것을 확인할 수 있

었다. 대략적으로 49.3%정도가 개선된 것을 확인할 수 있었다.

다음으로는, 2차원 평면상에서 clustering을 확인해보자.

pca를 통해 차원 축소 후 2차원 평면상에 clustering를 표현한 것이다. (위에서 언급했듯이, 이것은 초기모델과 우리가 수정한 모델 모두에 적용하였다.)

```
from sklearn.decomposition import PCA
pca = PCA(2)
pca.fit(data_lsa1)
X_PCA = pca.transform(data_lsa1)
X_PCA.shape

x, y = X_PCA[:, 0], X_PCA[:, 1]

colors = {0: 'red', 1: 'blue',2: 'green', 3: 'yellow', 4: 'orange',  5:'purple',6:'black'}
names = {0: 'Business', 1: 'Entertainment', 2: 'Living', 3: 'Metro', 4: 'Shopping',5:'Sports', 6:'Tech'}
df = pd.DataFrame({'x': x, 'y':y, 'label':labels})
groups = df.groupby('label')

fig, ax = plt.subplots(figsize=(7, 7))

for name, group in groups:
    ax.plot(group.x, group.y, marker='o', linestyle='', ms=5,
            color=colors[name],label=names[name], mec='none')
    ax.set_aspect('auto')
    ax.tick_params(axis='x',which='both',bottom='off',top='off',labelbottom='off')
    ax.tick_params(axis= 'y',which='both',left='off',top='off',labelleft='off')

ax.legend()

plt.show()
```



cluster한 결과가 위와 같이 나오는 것을 볼 수가 있었다. 초기모델과 비교해보았을 때 확연히 좋아진 모습을 볼 수가 있었다.

우선, completeness 부분에서 분석해보았을 때, 같은 색들이 밀접하게 분포되어있는 것을 볼 수가 있었다.

특히 이 모델은 초기의 모델과 비교하여 보았을 때, homogeneity 척도에서 큰 개선을 이루었다. 초기모델은 다양한 색깔들의 점들이 매우 근접하게 있어서 분류가 잘되지 않은 모습을 보여주었다. 하지만, 개선된 모델은 그 이전 모델보다 훨씬 더 색깔들이 분류가 잘 되어 있는 모습을 볼 수가 있었다.

따라서 우리는 일련의 과정을 거쳐서 모델을 개선하였고 그 모델은 초기모델보다 훨씬 더 성능이 좋은 것을 확인할 수 있었다.