데이터관리와 분석 Project #3

- Document search engine & Classification and Clustering -

8조

2018- 291 김 재

2017- 610 유 민

2017- 896 이 용

2019- 170 한창진

〈Contents〉

PART I. 문서 검색 엔진

PART II. 문서 분류 및 군집화

PART I. 문서 검색 엔진

1-1. 문서 채점 함수의 변형

Document와 query의 stopwords를 제외한 상태에서, 문서 채점 함수를 BM25로 하였을 때의 성능은 0.120157이었다.

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

우선 CustomScoring.py에 포함된 문서 채점 함수의 기본값을 BM25 함수로 두고, B를 0.1 간격으로, K1을 0.5 간격으로 변경하며 성능을 측정하였다. 이 때, document와 query의 stopwords를 제외한 상태에서 측정을 시행하였다.

그 결과, B=0.4, K1=5일 때 성능이 0.132643으로 가장 높다는 것을 확인하였다.

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

이후 문서 채점 함수를 BM25 함수를 지수로 갖는 함수로 두고, 이 지수함수의 밑수(param)를 0.01 간격으로 변경하며 성능을 측정하였다. 먼저 밑수를 0.1 간격으로 변경해본 뒤, 가장 score가 높을 때의 밑수±0.1의 구간을 0.01 간격으로 변경하면서, 최고 성능 수치 및 이 때의 밑수를 탐색하였다. 마찬가지로 document와 query의 stopwords를 제외한 상태에서 측정을 시행하였다.

그 결과, 밑수가 1.43일 때 성능이 0.149741으로 가장 높다는 것을 확인하였다.

따라서 BM25를 활용한 두 가지의 방식 중 후자의 방식을 택했을 때 성능을 더 많이 끌어올릴 수 있었기 때문에 이하 각 케이스에서의 최적 성능을 탐색할 때에는 후자의 방식을 적용하기로 결정하였다.

추가적으로, BM25와 함께 일반적으로 많이 쓰이는 TF_IDF 함수도 적용해보았지만 BM25보다 항상 낮은 결과를 기록하여 사용하지 않기로 하였다.

1-2. OrGroup의 변수 α 조정

```
parser = QueryParser("contents", schema=ix.schema, group=OrGroup.factory(0))
```

QueryResult.py의 parser에서 OrGroup 함수의 변수인 α를 0부터 1까지 변경하면서 성능을 측정하였다. 이 때, stopwords를 제외한 상태에서 측정을 시행하였다.

α의 값이 변해도 성능에는 거의 영향을 미치지 않음을 확인하였다. 따라서 이하 $\alpha = 0$으로 고정하기로 결정하였다.

1-3. stemming과 lemmatization 기법의 적용

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

```
new_q = ''
for word in q_nomark.split(' '):
    if word.lower() not in stopWords:
        # word_lem = lemmatizer.lemmatize(word.lower())
        word_stem = stemmizer.stem(word.lower())
        new_q += word_stem + ' '
```

Docunemt와 query에 stemming 및 lemmatization 기법을 적용한 뒤 최적 성능을 측정하였다. Stemming을 적용할 때에는 널리 알려진 PorterStemmer, SnowballStemmer, LancasterStemmer 중에서 성능이 가장 좋은 LancasterStemmer를 적용하였다. 이 때, stopwords를 제외하고, $\alpha = 0$인 상태에서 측정을 시행하였다.

정규화 기법 적용 X		Stemming 기법 적용		Lemmatization 기법 적용	
밑수	Avg. BPREF	밑수	Avg. BPREF	밑수	Avg. BPREF
1.44	0.149808	1.57	0.151583	1.54	0.152417

Table 1. 정규화 기법 적용 여부 및 기법 종류에 따른 최적 성능 비교

Stemming과 lemmatization 각각을 적용할 경우 모두 성능이 상승함을 확인할 수 있었다.

1-4. 문장부호의 제거

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

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

Document와 query에서 줄넘김(₩n) 및 물음표, 마침표, 쉼표, 느낌표를 제외한 뒤에 성능을 측정하였다. 이 때, stopwords를 제외하고, $\alpha = 0$이며, 정규화 기법을 적용하지 않은 상태 및 stemming 및 lemmatization 중 하나를 적용한 상태에서 측정을 시행하였다.

	정규화 기법 적용 X		Stemming 기법 적용		Lemmatization 기법 적용	
	밑수	Avg. BPREF	밑수	Avg. BPREF	밑수	Avg. BPREF
문장부호 유지	1.44	0.149808	1.57	0.151583	1.54	0.152417
문장부호 삭제	1.42	0.153704	1.53	0.163499	1.59	0.157892

Table 2. 문장부호의 삭제 여부에 따른 최적 성능 비교

세 경우 모두 문장부호를 제거했을 때 성능이 상승함을 확인할 수 있었다. 따라서, 이하 문장부호를 항상 제거하기로 하였다. 또한, 정규화 기법을 적용할 때가 성능이 더욱 높게 상승하였으므로, 이하 정규화 기법을 적용하지 않는 경우는 성능 측정 대상에서 배제하였다.

1-5. pos_tagging을 이용한 품사 필터링

```
doc_text_postag = pos_tag(doc_text_nomark.split(), tagset='universal')
doc_text_part = ''
for word, tag in doc_text_postag:
    if tag in ['NOUN']:
        doc_text_part += word + ' '
```

```
q_postag = pos_tag(q_nomark.split(), tagset='universal')
q_part = ''
for word, tag in q_postag:
    if tag in ['NOUN']:
        q_part += word + ' '
```

문장의 의미를 구성할 때 가장 중요한 역할을 하는 명사와 동사만을 남기고 검색을 수행하면 검색 성능이 좋아질 것으로 예상하여, pos_tagging을 이용하여 명사만을 남겼을 때, 명사와 동사만을 남겼을 때의 성능을 측정하였다. 이 때, stopwords를 제외하고, $\alpha = 0$이며, 문장부호를 제거하고, stemming 및 lemmatization 중 하나를 적용한 상태에서 측정을 시행하였다.

	Stemming 기법 적용		Lemmatization 기법 적용	
	밑수	Avg. BPREF	밑수	Avg. BPREF
필터링 X	1.53	0.163499	1.59	0.157892
명사만 필터링	1.55	0.129238	1.44	0.135006
명사, 동사 필터링	1.53	0.142076	1.55	0.137147

Table 3. 품사 필터링 여부에 따른 최적 성능 비교

품사를 필터링했을 때의 최적 성능값은 필터링하지 않았을 때의 최적 성능값보다 작음을 확인하였다. 따라서 이하 품사를 필터링하지 않기로 결정하였다.

1-6. word2vec 알고리즘의 적용

```
model = Word2Vec(sentences=result_list, size=1, window=3, min_count=5, workers=4, sg=0)
model.wv.save_word2vec_format('word2vec_size1_window3')
```

```
new_q = ''
for word in q_nomark.split(' '):
    if word.lower() not in stopWords:
        word_lem = lemmatizer.lemmatize(word.lower())
        word_stem = stemmizer.stem(word.lower())

        loaded_model = KeyedVectors.load_word2vec_format(word2vec_model)  # 모델 로드
        if word_stem in loaded_model.vocab:
            model_result = loaded_model.most_similar(word_stem)
            most_sim_word = model_result[0][0]
            # print(most_sim_word)
            new_q += word_stem + ' ' + most_sim_word + ' '  # 가장 가까운 단어도 함께 query에 넣어줌!
        else:
            new_q += word_stem + ' '
```

Query에 포함된 단어 각각에 대해, 이들과 가장 가까운 단어를 query에 함께 포함시켜주면 검색 성능이 좋아질 것으로 예상하여, word2vec 알고리즘의 similarity를 이용하여 이를 구현한 뒤 성능을 측정하였다. Word2vec의 size 변수는 1, window 변수는 3으로 두고 테스트하였다. 이 때, stopwords를 제외하고, $\alpha = 0$이며, 문장부호를 제거하고, stemming 및 lemmatization 중 하나를 적용한 상태에서 측정을 시행하였다.

	Stemming 기법 적용		Lemmatization 기법 적용	
	밑수	Avg. BPREF	밑수	Avg. BPREF
Word2vec 적용X	1.53	0.163499	1.59	0.157892
Word2vec 적용	1.53	0.163499	1.43	0.156018

Table 4. word2vec의 적용 여부에 따른 최적 성능 비교

Stemming을 적용할 경우에는 word2vec을 적용하기 전후의 최적 성능이 같았다. stemming을 수행하게 되면 단어의 원형이 보존이 되지 않아서, word2vec이 관련 단어를 잘 찾지 못하기 때문에 이러한 현상이 발생하였을 것이라고 추측하였다.

Lemmatization을 적용할 경우에는 word2vec을 적용했을 때의 최적 성능이 적용하지 않았을 때의 최적 성능에 비해 낮았다.

따라서 예상과 달리, word2vec 알고리즘의 적용은 성능 개선에 별다른 도움을 주지 못함을 확인할 수 있었다.

1-7. 최적의 검색 엔진 모델 제시

이상의 논의를 종합하여, $\alpha = 0$, document 및 query의 stopwords 제외, 문장부호 제거, stemming 적용, 문서 채점 함수가 1.53^(BM25)인 모델을 제시한다. 이 검색 모델을 주어진 76개의 query에 적용했을 시, 평균 BPREF는 0.163499이다.

PART II. 문서 분류 및 군집화

2-1. 영어 신문 기사 분류

```
categories = ['world', 'us', 'business', 'technology', 'health', 'sports', 'science', 'entertainment']

train_data = load_files(container_path='D:\\CC\\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(stop_words='english', lowercase=True)),
        ('tfidf', TfidfTransformer(use_idf=True, smooth_idf=True)),
        ('clf', MultinomialNB(alpha=.01)),
    ])
clf_nb.fit(train_data.data, train_data.target)
```

뼈대 코드 아래, naïve bayes의 pipeline을 만들어 주었다.

주의할 점은, 여러가지 naïve bayes classifier중 multinomial NB를 사용하여 alpha를 0.01로 맞추어 분류 작업을 실시하였다는 것이다. 즉, 다항분포 나이브베이즈로 알파 가중치를 0 이상으로 하여 다른 샘플 이용 시 확률이 0을 되는 것을 방지하였고, 구체적으로 0.01로 설정하여 Lidstone smoothing방법을 선택하였다.

```
clf_svm = Pipeline([
        ('vect', CountVectorizer(stop_words='english', lowercase=True)),
        ('tfidf', TfidfTransformer(use_idf=True, smooth_idf=True)),
        ('clf', SVC(kernel='linear', C=1.0, random_state=0)),
    ])
clf_svm.fit(train_data.data, train_data.target)

test_data = load_files(container_path='D:\\CC\\text\\test', categories=categories, shuffle=True,
                       encoding='utf-8', decode_error='replace')
docs_test = test_data.data
```

또한 SVM classification으로는 SVM 중 SVC로 C-Support vector classification 방법을 사용하였고, 선형 분류를 채택하였다. 가중치 인수 C에는 1.0을 넣었다.

```
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))
```

Naïve bayes의 정확도와 SVM의 정확도를 계산하였다.

```
NB accuracy : 24 / 26
SVM accuracy: 25 / 26
```

그 결과 Naïve Bayes는 24/26이 나왔고 SVM은 25/26이 나왔다.

```
TEAM = 8

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 pickle형태로 저장하였다.

2-2. 영어 신문 기사 군집화

군집화는 유사한 item들의 group을 파악하는 것이 목적이다. 이번 과제에서는 주어진 8개 카테고리의 영어 신문 기사(world, us, business, technology, health, sports, science, entertainment)에 대해, 신문 기사가 주어졌을 때 k-means clustering을 수행하고 그 성능을 평가/개선/분석한다.

군집화에서는 train data와 test data의 구분이 필요 없으므로 모든 기사들이 합쳐진 text_all의 데이터를 이용한다. 주어진 데이터에 대하여 먼저 디폴트 값으로 군집화를 시행한다.

```
from sklearn.datasets import load_files
from sklearn.feature_extraction.text import CountVectorizer, TfidfTransformer
from sklearn import metrics
from sklearn.cluster import KMeans
```

```
categories = ['world', 'us', 'business', 'technology', 'health', 'sports', 'science', 'entertainment']

data = load_files(container_path='text_all', categories=categories, shuffle=True,
                  encoding='utf-8', decode_error='replace',random_state=0)
```

우선 필요한 라이브러리를 모두 import한 후에 문제에 주어진 카테고리를 선언하고 load_files 함수로 text_all 폴더 내의 신문 기사 데이터를 불러온다. 불러온 데이터에 대해

CountVectorizer와 TfidfTransformer를 이용하여 데이터 전처리를 수행한다.

```
In [12]: count_vect=CountVectorizer()
         data_counts=count_vect.fit_transform(data.data)

In [13]: data_tf=TfidfTransformer().fit_transform(data_counts)
```

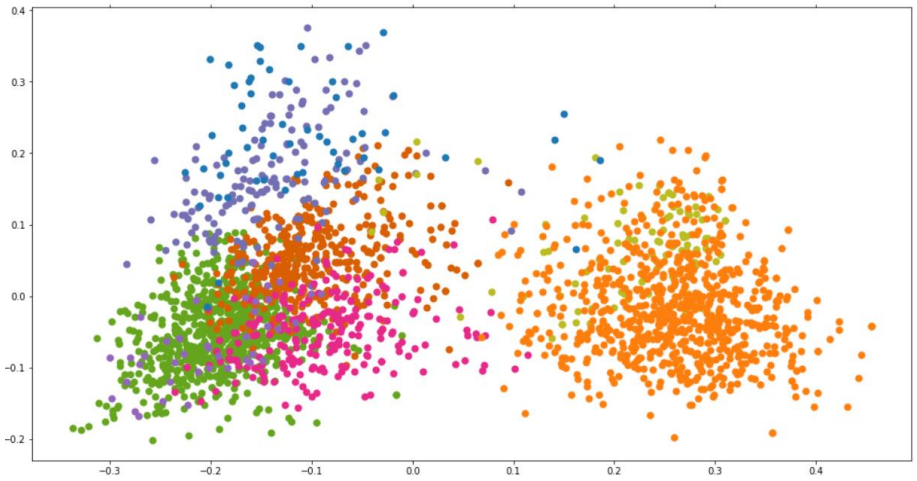
마지막으로 cluster의 개수를 카테고리의 개수와 동일하게 8개로 지정하고 전처리를 마친 데이터 data_tf를 fit 시킨다.

```
In [14]: clst = KMeans(n_clusters=8,random_state=0)
         clst.fit(data_tf)

Out[14]: KMeans(random_state=0)
```

이 과정을 수행한 군집화 결과에 대해 성능을 평가할 수 있다. 성능에 대한 지표로 'v-measure'를 이용한다. 이는 평가지표의 일종인 'homogeneity'와 'completeness'의 조화평균이다. 'homogeneity'는 각 군집에 하나의 클래스 내의 데이터를 포함하는지를 평가하며, 'completeness'는 주어진 class 내의 데이터가 하나의 군집으로 형성되는지를 평가한다. 수행한 군집화에 대한 v-measure, homogeneity, completenss는 차례대로 아래와 같이 출력된다.

```
In [15]: from sklearn.metrics import v_measure_score
         v_measure_score(data.target, clst.labels_)

Out[15]: 0.1938339039084002

In [17]: from sklearn.metrics import homogeneity_score
         homogeneity_score(data.target, clst.labels_)

Out[17]: 0.17562399326493303

In [19]: from sklearn.metrics import completeness_score
         completeness_score(data.target, clst.labels_)

Out[19]: 0.21625691721867848
```

v-measure의 최소가 0, 최대가 1인 점을 감안했을 때 군집화 결과에 대한 v-measure 0.1938은 군집화 성능이 좋다고 평가할 수 없으며 차원 축소를 통해 시각화해보면 아래와 같다. 위의 성능 지표에서 homogeneity보다 completeness의 값이 조금 더 큰 것을 확인할 수 있다, 같은 색의 점들이 무작위로 퍼져있지 않고 대략적으로 근접하게 위치하고 있긴 하지만(completeness 결정), 다른 색들과 많이 겹쳐져 있기 때문이다(homogeneity 결정). 따라서 이를 각각의 구별되는 cluster로 분류하기 매우 어렵다.



지금까지 주어진 데이터를 디폴트 값으로 군집화 해보았다. 이제 input parameter를 변화시켜 보면서 성능을 개선할 수 있다. 성능 개선을 거친 군집화 과정은 아래와 같다. 위의 군집화 과정에서 CountVectorizer와 K-means의 파라미터를 변화시켜보면서 성능을 개선시켜볼 수 있었다.

먼저 CountVectorizer의 input parameter를 변화시켜 본 것이다. Countvectorizer는 문서 집합에서 단어 토큰을 생성하고 각 단어의 수를 세어서 벡터로 인코딩한다.

```
stop_words = set(stopwords.words('english'))

vect=CountVectorizer(stop_words=stop_words,min_df=20, max_df=730)
data_counts=vect.fit_transform(data.data)
```

1. Stopwords : 'english'

토큰을 생성할 때 유의미한 단어 토큰만 생성하기 위한 작업으로, stopwords로 지정된 단어들은 토큰화되지 않는다. NLTK 라이브러리에서는 'english' 라는 stopwords 리스트를 정의해두었다. 이를 이용하면 영어의 관사, 대명사 등을 stopwords로 쉽게 처리할 수 있다.

2. min_df=20

문서 20개 미만에 나오는 단어를 삭제한다는 조건이다. 같은 카테고리 내에 있는 문서들은 최대한 가깝게, 다른 카테고리에 있는 문서들과는 최대한 멀리 떨어져야 하는데 너무 드물게 나오는 단어들은 같은 카테고리 내에 있는 문서들의 거리도 멀리 떨어뜨릴 수 있다고 판단했다. 적절하게 숫자를 바꿔가며 성능을 확인했다.

3. max_df=480

480개 이상의 문서에서 나오는 단어를 삭제한다는 조건이다. Min_df를 선정한 것과 마찬가지로 같은 카테고리 내에 있는 문서들은 최대한 가깝게, 다른 카테고리에 있는 문서들과는 최대한 멀리 떨어져야 하는데 너무 자주 나오는 단어는 카테고리와 카테고리를 구별하기 어렵게 만들 수 있다. Text_all의 폴더에서 8개의 카테고리 모두, 대략 300개 전후의 신문기사를 갖고 있다. 따라서 max_df도 이와 가까운 수로 적절하게 값을 바꿔가며 높은 성능을 갖

는 값을 선정했다.

4. N-gram

n-gram range(1,1), 즉 unigram일 때 가장 성능이 좋았으며 이는 디폴트 값이므로 따로 변경하지 않았다.

다음으로는 k-means clustering의 파라미터 설정을 바꿔볼 수 있다.

```
clst = KMeans(n_clusters=8, init='random', max_iter=1000, n_init=5,random_state=0)
```

1. init='random'

k-menas clustering의 성능은 초기 중심 좌표를 설정하는 것에 영향을 받는다. 초기 중심의 좌표를 설정하는 방법은 대표적으로 k-means++와 random 두 가지가 있다. K-means++은 주어진 데이터 포인트 중 하나를 첫번째 초기 중심으로 설정하고 이 점과 나머지 데이터 사이의 거리를 모두 측정하여 거리비례 확률에 따라 다른 초기중심을 결정한다. Random은 말 그대로 무작위로 초기 중심을 설정하는 방법이다. 두 가지 중에 random의 성능이 더 좋았다.

2. max_iter=1000

이 파라미터는 최대 몇 번의 iteration을 허용할 것인지에 관한 제약이다. K-means clustering은 여러 번의 iteration을 반복하면서 중심점이 수렴될 때까지 중심점을 갱신해나간다. Kmeans의 제약에 (verbose=1)을 추가하면 몇 번의 iteration만에 clutering이 종료되는지 확인해볼 수 있다. 실제로 수행해보니 대략 20번 내외에서 clustering이 종료되었지만 값을 늘려도 성능에 영향이 없으므로 충분하게 늘려놓았다.

3. n_init=5

앞서 언급했듯이, k-means clustering은 초기 중심 설정에 따라 성능이 달라질 수 있다. Random의 방식 같은 경우는 각 실행마다 어떤 점을 무작위로 선택하는지에 따라 성능이 달라질 수 있는 것이다. 따라서 반복 수행을 거치게 된다. 해당 값에서 가장 좋은 성능을 보였다.

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

Out[54]: 0.580774676043078

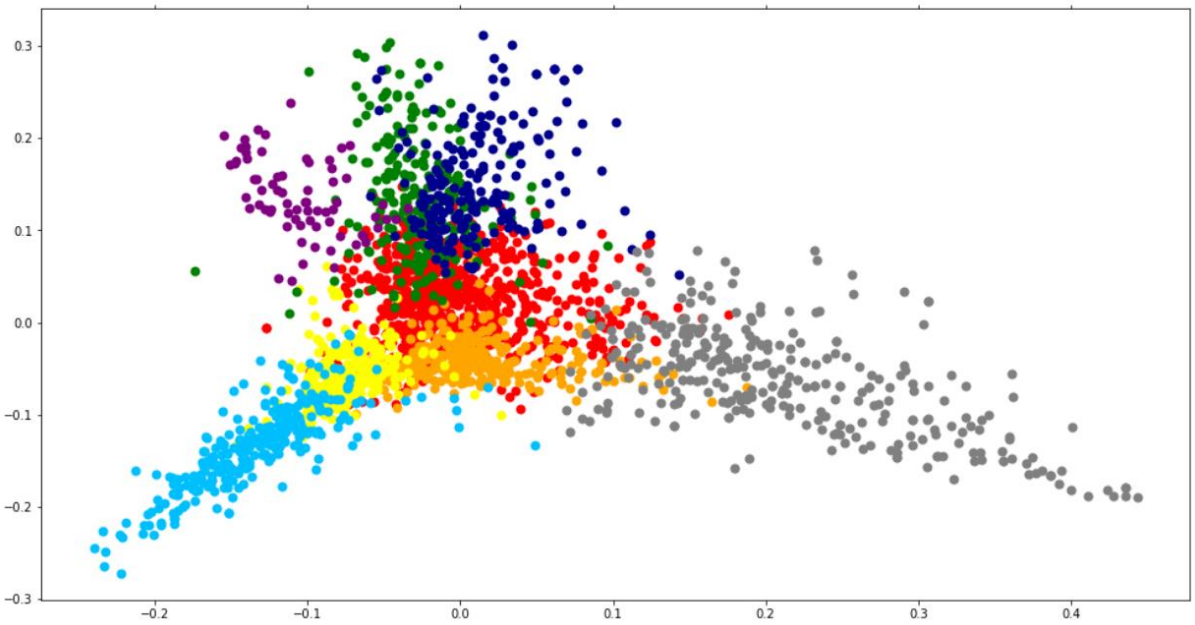
```
In [55]: from sklearn.metrics import homogeneity_score
         homogeneity_score(data.target, clst.labels_)
```

Out[55]: 0.5601211979731063

```
In [56]: from sklearn.metrics import completeness_score
         completeness_score(data.target, clst.labels_)
```

Out[56]: 0.6030095875914545

이 과정을 통해 수행한 clustering의 v_measure 값은 약 0.58으로, 디폴트값으로 수행했을 때의 v-measure 값인 0.19보다 약 3배의 값으로 개선된 것을 확인할 수 있다. 개선된 cluster에 대해서도 가시화해볼 수 있다.



입력 파라미터가 디폴트일 때와 달리, 같은 색의 점들은 더욱 근접하게 위치하고 있으며 다른 색의 점들이 겹치는 정도가 줄어들었음을 확인할 수 있다.

앞서, k means를 정의했던 변수 clst에 대해 labels_ 를 적용하면 array가 출력되는데 이는 각 인덱스에 해당하는 문서들이 어떤 군집에 속해 있는지 군집의 인덱스를 저장하고 있다. 따라서 for문과 if문을 사용하여 2472개의 문서 중에서 clst.labels_가 같은 값을 갖는 인덱스들을 모두 출력해줄 수 있다. 오른쪽에 첨부한 코드는 0번째 cluster로 군집화된 문서들의 인덱스이다.

```
In [168]: for i in range(2472) :
              if clst.labels_[i]==0 :
                  print (i)
```

```
5
6
13
22
24
27
30
31
34
39
42
43
44
47
51
53
56
59
63
66
```

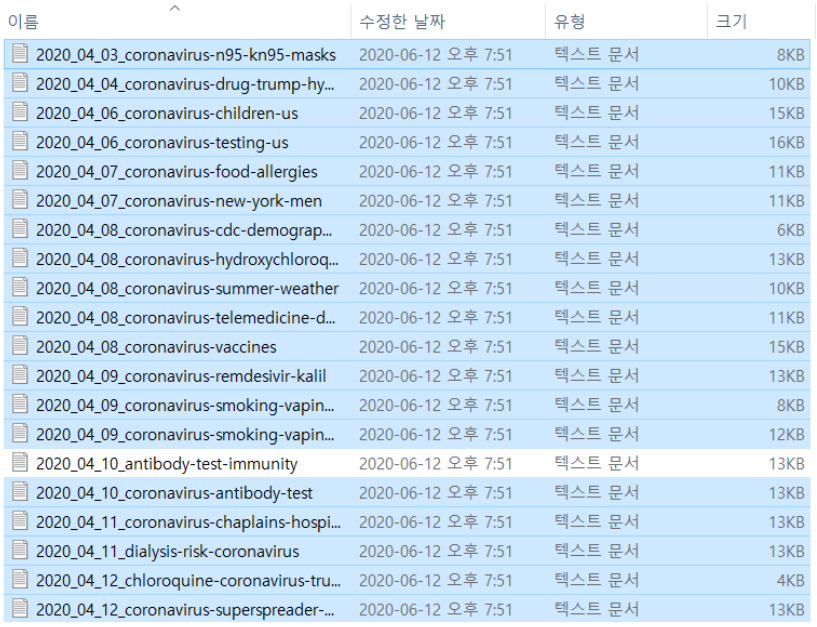
군집화의 성능 지표인 v-measure가 약 0.5인 점을 감안하여, 오른쪽의 결과와 같이 출력된 인덱스에 대해 정답 레이블들을 조사해보면 각 cluster가 어떤 카테고리를 군집화하고자 했는지를 유추할 수 있다.

유추 결과에 따른 cluster 별로 유추한 카테고리와 PCA를 이용한 가시화 상에서의 색은 다음과 같다.

0 : world, 빨강 1 : science, 주황 2 : entertainment, 노랑 3 : technology, 초록

4 : sports, 하늘 5 : business, 남색 6 : us, 보라 7 : health, 회색

Clustering 가시화 결과를 참고하여 분석해볼 수 있다.



이름	수정한 날짜	유형	크기
2020_04_03_coronavirus-n95-kn95-masks	2020-06-12 오후 7:51	텍스트 문서	8KB
2020_04_04_coronavirus-drug-trump-hy...	2020-06-12 오후 7:51	텍스트 문서	10KB
2020_04_06_coronavirus-children-us	2020-06-12 오후 7:51	텍스트 문서	15KB
2020_04_06_coronavirus-testing-us	2020-06-12 오후 7:51	텍스트 문서	16KB
2020_04_07_coronavirus-food-allergies	2020-06-12 오후 7:51	텍스트 문서	11KB
2020_04_07_coronavirus-new-york-men	2020-06-12 오후 7:51	텍스트 문서	11KB
2020_04_08_coronavirus-cdc-demograp...	2020-06-12 오후 7:51	텍스트 문서	6KB
2020_04_08_coronavirus-hydroxychloroq...	2020-06-12 오후 7:51	텍스트 문서	13KB
2020_04_08_coronavirus-summer-weather	2020-06-12 오후 7:51	텍스트 문서	10KB
2020_04_08_coronavirus-telemedicine-d...	2020-06-12 오후 7:51	텍스트 문서	11KB
2020_04_08_coronavirus-vaccines	2020-06-12 오후 7:51	텍스트 문서	15KB
2020_04_09_coronavirus-remdesivir-kalil	2020-06-12 오후 7:51	텍스트 문서	13KB
2020_04_09_coronavirus-smoking-vapin...	2020-06-12 오후 7:51	텍스트 문서	8KB
2020_04_09_coronavirus-smoking-vapin...	2020-06-12 오후 7:51	텍스트 문서	12KB
2020_04_10_antibody-test-immunity	2020-06-12 오후 7:51	텍스트 문서	13KB
2020_04_10_coronavirus-antibody-test	2020-06-12 오후 7:51	텍스트 문서	13KB
2020_04_11_coronavirus-chaplains-hospi...	2020-06-12 오후 7:51	텍스트 문서	13KB
2020_04_11_dialysis-risk-coronavirus	2020-06-12 오후 7:51	텍스트 문서	13KB
2020_04_12_chloroquine-coronavirus-tru...	2020-06-12 오후 7:51	텍스트 문서	4KB
2020_04_12_coronavirus-superspreader-...	2020-06-12 오후 7:51	텍스트 문서	13KB

마지막 7번 클러스터(회색)의 경우 다른 클러스터와 잘 구별되어 있음을 확인할 수 있다. 그 이유는 올해 초부터 전세계에 퍼진 코로나바이러스에 관련된 뉴스가 health 카테고리를 차지하는 비율이 압도적이었기 때문이었다고 예측할 수 있다. 오른쪽 사진은, health 카테고리 내의 신문기사를 날짜로 정렬하였을 때 4월 초에 해당하는 기사들이다. 한 페이지 내에서 4월 10일의 한 기사를 제외하고 모든 기사가 제목에서 코로나바이러스를 직접적으로 언급하고 있는 것을 확인할 수 있다. 즉, 강력한 키워드 '코로나 바이러스'로 health 카테고리를 잘 군집화했다고 판단한다. 하지만 7번 군집이 0번 군집(health)과 1번 군집(science)과 일부 접점을 보인다. 이 또한 코로나바이러스 키워드의 영향이라고 유추할 수 있다. 코로나 바이러스는 전세계적(world) 문제이며, 실제로 world 카테고리 내에 코로나바이러스 관련 기사가 상당수 존재하는 것을 확인할 수 있다. 또한 science 카테고리 내에서도 코로나바이러스와 관련하여, 바이러스의 유전적 분석이나 치료법 등에 대해 다루고 있으며 이러한 영향이 크게 작용했다고 판단된다.

한 가지 짚고 넘어갈 점은, 앞서 CountVectorizer를 적용할 때 max_df=480을 적용했다는 점이다. 2472개의 기사 중에서 480개 이상의 기사에서 코로나바이러스를 언급한다면 이 단어는 단어장에 추가되지 못한다. 실제로 get_feature_names()를 적용하여 확인해보면 키워드 'coronavirus'는 존재하지 않지만 'corona'와 'coronaviruses'가 존재한다. 또한 'viruse'는 없지만, 'viruses'는 남아있는 것을 확인할 수 있다. Stem 처리를 별도로 안 했기 때문에 480개 미만의 기사에서 등장했으면 비슷한 의미로 남아서 영향을 준 것으로 예상된다.

다음으로 또 눈 여겨 볼 만한 군집은 world 카테고리를 구별하고자 한 0번 군집(빨강)이다. 이 군집의 경우 거의 대부분의 군집과 많은 접점을 가지면서 겹치는 부분이 비교적 많은 것을 볼 수 있다. 이는 이름에서 먼저 예상할 수 있는데, 카테고리의 이름 'world'는 존재하는 거의 대부분의 것을 포함해도 논리적으로 무리가 없기 때문이다. 먼저 코로나바이러스에 대한 다수의 기사로 7번 군집(health)과 6번 군집(science)에 대한 접점을 유추해볼 수 있다. 또한 카테고리 world 내의 기사 몇 개를 예로 들 수 있다. 1월 5일 기사인 'trump-iran-nuclear-agreement', 1월 6일 기사인, 'trump-iran-soleimani-strategy', 1월 9일 기사인 'trump-china-iran'의 경우 단기간 내에 트럼프가 다수 언급되면서 6번 군집(us,보라)과의 접점을 시사했다. 그리고 3번 군집(technology, 초록), 5번 군집(business, 남색)과도 넓게 접하고 있다. 4차 산업혁명 시대에서 기술은, 지역의 경계를 무너뜨리게 되었다. 현시점에서의 기술이 갖는 특징을 고려해본다면 넓은 접

점은 당연할 것이다. business 카테고리 또한 같은 맥락을 갖는다. Business 카테고리 내의 기사들을 살펴보면 주식에 관련된 기사들이 많은 비중을 차지하고 있다. 주식에 아주 조금의 관심이라도 있는 사람들은 미국 증시의 변화가 전세계(world)에 어떻게 작용하는지 쉽게 이해할 것이다. 다양한 요인으로 설명이 가능하겠지만, world 카테고리는 특성 상 넓은 범위를 포괄적으로 다양한 주제를 포함할 수 있기때문에 군집화에도 그러한 영향을 미친 것 같다.