11-441 Homework 2

Uzi Smadja – usmadja

Experiments and analysis

Corpus exploration:

I used Java for the corpus exploration. Source code is in hw2 folder(including data files).

```
Development Set:
Number of documents:942
Number of words:164466
Number of unique words:14063
Average number of unique words per document:14.928874734607218


Test Set:
Number of documents:942
Number of words:163316
Number of unique words:13924
Average number of unique words per document:14.781316348195329


First Document in Development set:
Unique words:161
Word ids that occured twice:
2 5 10 18 23 27 28 30 32 42 44 45 46 50 52 60 62 69 79 87 91 99 102 107 114 141
```

Baseline approach:

runClusterSize.py runs the baseline algorithm with different numbers of doc clusters and different number of word clusters in a grid algorithm. After looking at the data, and taking into account the time, internal and external metrics I have decided to choose the number of word clusters to be 150 and the number of document clusters to be 50.
After a second thought, **I decided to change to 150 word clusters and 75 document clusters** and got the following results, convincing me to stay with these parameters:

F1	Time	Doc cos sum	Word cos sum
0.55	30.7	644	4213

Grid search for number of word clusters and document clusters:

Word clusters	Doc clusters	Doc F1	Time	Doc Cos Sum	Word Cos Sum
50	50	0.43	14.3	551	3790
50	100	0.41	19.2	553	4311
50	150	0.44	23.9	547	4593
50	200	0.45	27.7	551	4900
100	50	0.53	20.8	605	3881
100	100	0.55	33.4	609	4487
100	150	0.54	41.1	612	5023
100	200	0.49	48.2	603	5242
150	50	0.56	32.6	641	3870
150	100	0.56	41.9	645	4584
150	150	0.56	52.3	646	5008
150	200	**0.58**	60.7	649	**5392**
200	50	0.52	39.7	664	3865
200	100	0.57	51.2	674	4677
200	150	0.57	62.0	**676**	5079
200	200	0.54	**71.0**	675	5401

runCriteria.py runs the baseline algorithm after choosing 150 word clusters and 75 document clusters. runCriteria tries different thresholds for the kmean algorithm.
I used the stopping criteria as a percentage of clusters that havent changed since the last iteration or a maximum number of 20 iterations(just to be safe). **I chose to set the threshold at 0.01** in order to maximize F1 and not gain too much computation time.

Threshold	Doc F1	Time	Doc Cos Sum	Word Cos Sum
1	0.49	30.8	635	4111
0.1	0.53	30.6	638	4153
0.01	**0.58**	34.1	641	4318
0.001	0.57	**39.6**	**644**	**4363**

After tuning the parameters, I will run the code 10 times with run.py:

Doc F1	Doc Cosine sum	Word cosine sum
0.546	640	4339
0.531	641	4314
0.557	640	4385
0.500	639	4189
0.560	643	4320
0.555	645	4309
0.556	**647**	**4456**
0.599	**647**	4443
0.575	**647**	4352
0.523	643	4347

Some statistics about the data:

- Doc F1 Mean: 0.55
- Doc F1 Variance: 0.00076
- Best Doc F1: 0.599
- F1 for best sum of cosine: 0.556 or 0.599 or 0.575
- Word cosine sum mean: 4345
- Word cosine sum variance: 5647.82
- Best word cosine sum: 4456

How well do the internal evaluation results reflect the external evaluation results?

It can be easily seen that the internal evaluation and external evaluations are strongly and positively correlated, the lowest F1 score is together with the lowest cosine sum and the highest F1 is together with the highest cosine sum. Though there are obviously some lower F1 scores with higher cosine sums, they correlation between them is pretty strong.

Do word clusters with better internal evaluation results seem to have more intuitive word groupings?

When I looked at the results with lower internal evaluations, I noticed that pairs of First name and Last name were not always classified together like ‘carole simpson’,’ Madeleine Albright’ but they tended to be together in the results with higher word cosine sum.

Do these results match your expectaions?

These results match my expectation because there seems to be a positive correlation between good internal evaluation and external evaluation like we were hoping for. Sometimes there are is a good internal evaluation with good external evaluation (or the other way around), that are caused by bad random seeds in the beginning.

The custom algorithm:

I chose to use TF-IDF instead of normal TF for the weighting of douments in the matrix, in order to reflect the importance of cetrain words compared to others. Some words may appear in a lot of documents and therefore be less significant when calculating the weight of that word. For example, the word ‘good’ does not give information about the content of a document and therefore does not contribute a lot in calculation of distance between documents.

I chose not to change from cosine similarity to euclidian in order not to lose the order relevance that we have in the cosine similiarity calculation.

I compared between results starting with a Doc2Word or Word2Doc matrix. I received slightly higher results when starting with a Word2Doc matrix and decided to stay with it.

Doc F1	Doc Cosine sum	Word cosine sum
0.641	613	4707
0.586	611	4487
0.567	610	4525
0.573	611	4529
0.590	615	4483
0.594	612	4629
0.584	613	4633
0.587	616	4608
0.543	609	4468
0.608	616	4624

Customized algorithm document F1 mean: 0.587 (0.55 in the baseline algorithm)
Customized algorithm doc cosine sum mean: 612 (643 in the baseline algorithm)
Customized algorithm word cosine sum mean: 4569 (4345 in the baseline algorithm)

I will conclude that the customized algorithm is significantly more effective in external evaluation of documents and in internal word evaluation but less effective in internal document evaluation.

I am not entirely sure, but I believe the decrease in doc cosine sum is due to the change from TF to TF-IDF which helped the word clustering and did the opposite with doc clustering.

The software implementation & data preprocessing:

run.py is the main script that calls the bipartite clustering algorithm
bpc.py contains all the algorithm including kmean,bipartite clustering
runClusterSize.py was used to test different cluster sizes
runCriteria.py was used to test different stopping criteria threshold(kmean)

bpc.py:

- bipartite_clustering(…) - receives Doc2Word matrix, and parameters for the algorithm and returns a Doc2DocCluster and Word2WordCluster mapping.
- Kmean(...)/kmean_body(…) - receives a matrix,number of clusters and threshold and calculates the kmean algorithm.
- Random_numbers(S,N) – generate S different numbers from 0 to N in order to choose initial seeds
- Get_new_centroids(X,closest_cluster) – receive a matrix and the mapping for each row and its closest cluster and recalculates the new centroids.
- Get_df – calculate the IDF of every word in the documen
- Main() – strip the docVectors file and construct the matrices.

Libraries:

- Numpy
- Sklearn.metrics.pairwise – pairwise_distances
- Sklearn.preprocessing – normalize (for cosine sum)
- Scipy.sparse – csr_matrix,lil_matrix (sparse matrices. lil matrix faster for calculating mean)
- Math – in order to use log function for IDF calculation.
- Random – for initial seeds

I started the corpus exploration in Java and then moved to Python. The decision to code in Python came because there are no easy to use libraries (I tried Jeigen and Jama) in Java for manipulation of matrices and vectors. Python's numpy and scipy libraries provide most of the functions necessary and are easy to use and intuitive(like pairwise_distances). Moreover, Scipy provides sparse matrix representation that is easy and simple to use. Python might not be as effective as Java or C++.

I encountered problem when calculating the mean of clusters every step in the algorithm. Whenever a cluster was empty, the mean function would divide by zero and crash. I decided to add a verification before the calculation. I didn't try to remove the empty clusters or assign a random point to the empty clusters in order to get a feel of how many clusters were empty. After a few tests I discovered the number of empty clusters was between 0 and 5 and decided that the impact it would have on my results is minor (after choosing 150/75 clusters).