Your Name: Yan Zhao

Your Andrew ID: yanzhao2

Homework 2

1 Corpus Exploration

1.1 Development Set

- The total number of documents: 942
- The total number of words: 254852
- The total number of unique words: 14063
- The average number of unique words per document: 174.59

1.2 Test Set

- The total number of documents: 942
- The total number of words: 249516
- The total number of unique words: 13924
- The average number of unique words per document: 173.37

1.3 First Document in Development Set

- The total number of unique words: 161
- all of the word ids that occurred exactly twice in the document:
  [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]

2 Experiments

2.1 Baseline

In my baseline approach, I set three parameters for tuning: number of cluster, number of iterations and the difference between two adjacent iterations.

One thing needs to mention here is that at each iteration, if empty cluster (which means there are no data points assigned to it), I will remove it from the cluster set. My approach is different from assignment FAQ suggestions. The intuition is simple, if no data points are assigned to a center, that means the cluster is useless and we don't need to update in the following iterations as a result. Also, removing empty cluster can reduce iteration times and thus speed up the system.

Based on my implementation, when I perform experiments, results show clusters will converge within 20 iterations, and the difference between current and previous clusters would be exactly 0 (which means data points assignment is the same). Thus, there is no need to adjusting stop criteria for my system, as it will get to convergence within a short time.

Here I designed my experiments to adjust number of clusters to be 60, 70, 100, 150 and 200. For each setting, I run the algorithm 10 times and calculate the average F1 score. Table below shows results of my experiments, and shows I can get best performance on average using K-means with number of clusters equal to 150.

Experiments/ No. Clusters	1	2	3	4	5	6	7	8	9	10	**Avg**
60	0.4911	0.4844	0.5271	0.4818	0.5126	0.5061	0.4572	0.5594	0.4809	0.4938	**0.4994**
70	0.5098	0.5239	0.5634	0.4962	0.4894	0.5607	0.5218	0.4937	0.5400	0.5421	**0.5241**
100	0.5546	0.5785	0.6020	0.5318	0.5478	0.5831	0.5333	0.5234	0.5402	0.5443	**0.5539**
150	**0.5886**	**0.5914**	**0.6161**	**0.5857**	**0.5540**	**0.5849**	**0.5613**	**0.5543**	**0.5604**	**0.5324**	**0.5729**
200	0.5534	0.5213	0.5504	0.5736	0.5381	0.5583	0.5247	0.5540	0.5425	0.5792	**0.5496**

2.2 K-means++

The only difference between K-means++ and baseline is on choosing initial clusters. For K-means algorithm I choose initial clusters uniformly from all data points, while for K-means++ I choose them randomly based on their distance from existing clusters. Since K-means algorithm will only find local minimum by iterations, using K-means++ can optimize initials so that we can get global minimum after a limited number of iterations.

Here I designed my experiments on K-means++ to adjust number of clusters to be 70, 100 and 150. For each setting, I run the algorithm 10 times and calculate the average F1 score. Table below shows results of my experiments. I can get best performance on average using K-means++ with number of clusters equal to 150. Also as is shown, on average, the performance of K-means++ is better than baseline K-means algorithm.

Experiments/ No. Clusters	1	2	3	4	5	6	7	8	9	10	**Avg**
70	0.5448	0.5053	0.4969	0.5690	0.4990	0.5845	0.5211	0.5098	0.5211	0.5746	**0.5326**
100	0.5853	0.5269	0.5668	0.5341	0.5825	0.5765	0.5575	0.5792	0.5856	0.5752	**0.5670**
150	**0.5909**	**0.5771**	**0.5810**	**0.5756**	**0.5476**	**0.5831**	**0.5839**	**0.5702**	**0.5743**	**0.5846**	**0.5769**

2.3 Custom Approach

I designed my custom approach based on K-means++ from two aspects. One is to do feature engineering, and the other one is to modify distance methodology.

1. **Feature Engineering:**

For this part, my intuition is that there are three aspects that could have influence on representing a document. The first one is obvious term frequency, which has already been represented in feature vectors of documents. The second one is document frequency of each term. The reason is one term with high document frequency should be considered less useful than a term with low document frequency. One example would be stopping words like "the", they appear in every document of a corpus, so although with high term frequency, they should not be considered as a feature for a document. The third one is document length. The reason is that for a long document, the average term frequency would be of course larger than that of a short document, so a term appearing k times in a short document should be more representative than a term appearing same times in a long document.

Based on the intuition above, for each term frequency, I give a penalty based on its document frequency and document length. The formula is shown below:

$$\text{Term Weight} = \mathrm{tf} * \log\left(\frac{N-df+0.5}{df+0.5}\right) * \frac{1}{k*\frac{doclen}{avg\_len}}$$

Here in my formula, *Term Weight* is my modified feature value; *tf* represents for term weight; *N* is the total number of documents in corpus, which is the same as I calculated in part one; *df* is the document frequency for each term; *doclen* is the length of document; *avg_len* is the average document length for whole corpus; ***k*** is parameter I set myself, which I will adjust in following experiments.

As is seen from the formula, I design the feature for each term in a document from three aspects, as stated above. The first part is term frequency, which I read directly from *docVector* files. The second part is my *idf* calculation, here I use RSJ weight intuited by BM25 score in Search Engine, it is a smoothing version of *log(N/df)*. The third part is my penalty on document length, and I use parameter ***k*** to adjust the penalty scale.

2. Distance Methodology:

Except for using Cosine similarity to group data points into clusters, I also use Euclidean distance. The reason I use another distance method is Cosine similarity will group data points based on their angle to the original, while Euclidean distance can group data points based on their absolute distance. Since we have no idea about the distribution of data points before experiments, we should try different distance method to better understand the data set.

However, after experiments, I find that using Euclidean distance for K-means, the cluster will not converge, and the final F1 score is less than 0.5, which means a random guess. There are some guesses for this phenomenon that I will state in part 3. As a result, I remove this modification from my custom method.

Here I designed my experiments on custom algorithm based on K-means++. According to the results of K-means++, it has best performance on average when number of clusters equal to 150. Thus in experiments of this section, I set number of clusters equal to 150, and tune on parameter k. For each setting, I run the algorithm 10 times and calculate the average F1 score. Table below shows results of my experiments. I can get best performance on average using custom algorithm with number of clusters equal to 150, *k* equals to 0.6. Also as is shown, on average, the performance of custom algorithm is better than baseline and K-means++ algorithm.

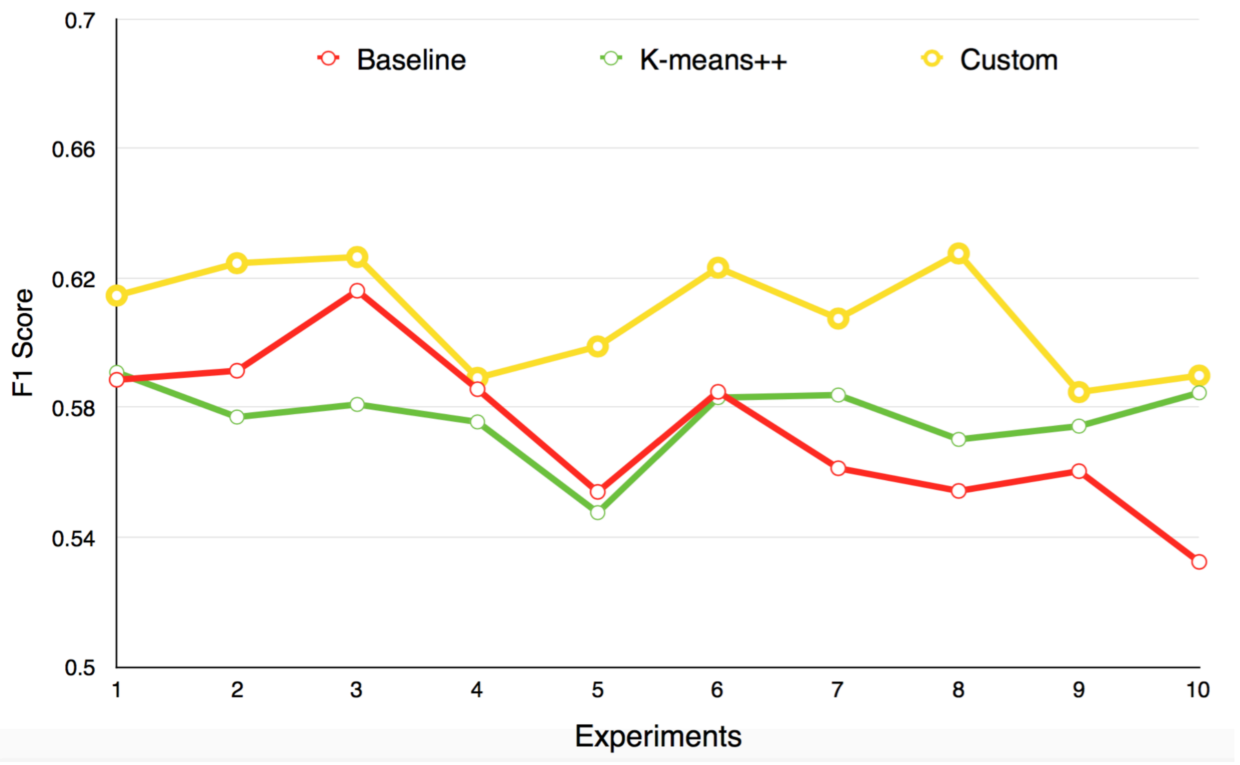
Experiments/ No. Clusters, k	1	2	3	4	5	6	7	8	9	10	**Avg**
150, 0.6	**0.6146**	**0.6246**	**0.6265**	**0.5892**	**0.5989**	**0.6232**	**0.6075**	**0.6276**	**0.5848**	**0.5899**	**0.6087**
150, 0.8	0.6172	0.5591	0.6341	0.5860	0.5695	0.6011	0.6235	0.5818	0.6027	0.5936	**0.5970**
150, 1.0	0.6174	0.5960	0.5630	0.5635	0.5568	0.5929	0.5670	0.5796	0.6438	0.6270	**0.5907**

3 Analysis

3.1 Performance

As is shown in the experiments, on average, my custom algorithm has a best performance, while baseline approach has a worst performance.

Moreover, below is a chart describing the stability of each approach. For every approach, I choose the settings with the best performance (Baseline, K-means++: number of clusters = 150; Custom: number of clusters = 150, k = 0.6), and run 10 times. As is shown, my custom approach has the best performance on average, while baseline K-means has the worst. Also, the chart shows all three approaches are stable with a little fluctuation within 10%, while K-means++ and Custom approach are even more stable than baseline algorithm.



3.2 Custom Algorithm

For feature engineering, after setting *idf* and document length penalties to term frequency, I can improve F1 score significantly. As is stated in part 2, one term with high document frequency should be considered less useful than a term with low document frequency. Like stopping words would appear in every document of a corpus, with high term frequency, while they are useless in representing the topic of documents. Also, a term appears frequently in short documents seem to be more representative of their topics than in long documents. Thus, given two penalties on document frequency and document length can make term frequencies be more representative of a document, and make document vector more reasonable as a result.

For distance methodology, after using Euclidean distance, K-means algorithm cannot get to convergence, and the F1 score drops significantly. I think the reason is that Cosine similarity measure data points difference by their angle to original, while Euclidean distance measure by their absolute values. Thus, for document vectors in our experiment, which are highly sparse, Cosine similarity can better represent the difference of each vector. Moreover, true label in our experiment is based on topics of each document, which may be originally grouped by angle to the origin, not by their absolute distance.

4 Software Implementation & data Processing

4.1 Design decisions and software architecture

I create two java classes, matrixIO.java is used for reading document from files or writing document labels to files. It can also read document frequency. Kmeans.java is used for calculation of three clustering approaches.

For the work flow, I read document vectors from files, and store them as a sparse matrix. Then I use different initialization methods to initial clusters from document matrix, and store clusters as matrix. I can then perform basic K-means algorithm iteratively based on the document matrix and cluster matrix, and give final labels of each document. In the end, I write these labels to a file for evaluation.

4.2 Major data structure

For each the document vector and cluster centroid, I represent them as Vector from mahout-math, and I store and calculate them by using methods of Matrix from mahout-math.

4.3 Programming tools and libraries

I use mahout-math as recommended in assignment FAQ, for sparse matrix calculation.

4.4 Strengths and weaknesses

I think the strength of my design is I can adjust different parameters, choose different approaches or perform experiments on development/test data with just changing a few Boolean arguments, I don't need to modify any function when doing experiments.

For example, when I perform the algorithm, the only function I need to call is

"findCluster(Boolean isDev, Boolean isKmeanPlus, Boolean isCosine, Boolean isIDF, Boolean isIDocLen)"

Then I can set different experiment by just modifying arguments of this function, and all parameters used for the algorithm, like number of clusters, can be initialized when instantiate java class.