# *Report of Prototype 3*

3 test runs on server: /home/ns1077/benchmarkY1TestRuns
Github: https://github.com/yuanph4ever/TREC-Complex-answer-retrieval-Track

## *Peihao*

a. Kmeans Clustering by 20,000 paragraphs.

For last prototype, I was using 10,000 paragraphs to clustering. This time, I used 20,000 paragraphs (I tried 30k, 40k, and 50k. But they all run out of the memory). The methodology is the same as last time. First, taking 20,000 paragraphs from corpus. Second, converting paragraphs to vectors (tfidf as value) and then running k-means on them (I tried k = 3 and k = 5) to group clusters. Last, doing paragraphs retrieval by using the clusters.

The way to use the clusters is, getting two ranks of clusters for a pair of query and retrieved document. Use the top 1 cluster as the true cluster of query and find the position of that cluster in the rank of document's clusters (pos). Then use this algorithm to get the final score for ranking.

Score (query, paragraph) = BM25_score (q, p) + 1 / pos

Sometimes we cannot find query's true cluster in the retrieved clusters of document. In this case, I provide this algorithm to get the final score for ranking.

Score (query, paragraph) = BM25_score (q, p) * 1/2

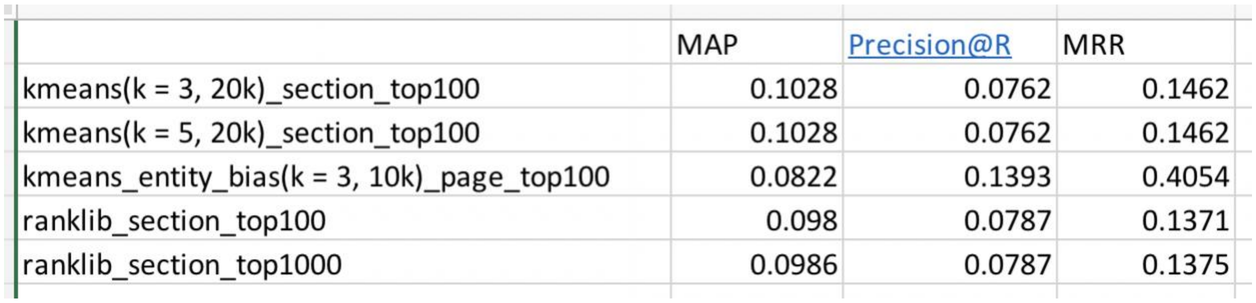
b. "Entities bias" Kmeans Clustering

There are three changes for this method:

1. Give bias to entities: use DBpedia Spotlight to identify entities, give entities 2*tfidf as values in vectors of paragraphs. This way gives entities a bias.
2. Get rid of "junk" paragraphs: if DBpedia Spotlight cannot identify any entity from a paragraph, then see it as "junk" and do not use it for clustering. For example, there are some paragraphs like, "[[[ 1992 , 1994" in corpus. They should not be used for clustering.
3. Randomly collect paragraphs: take one paragraph from every 100 paragraphs in corpus to collect them for clustering. This way keeps variance of topics.

c. Learn to rank

1. Use makeRanklibFile.py to generate feature files for training data and test data.
2. Use Ranklib to train the feature file for training data and generate the model file.
3. Use the model file to rank the feature file of test data and get the score file.
2. Use reRankByML.py to generate the run file by using the score file.



| | MAP | Precision@R | MRR |
|---|---|---|---|
| kmeans(k = 3, 20k)_section_top100 | 0.1028 | 0.0762 | 0.1462 |
| kmeans(k = 5, 20k)_section_top100 | 0.1028 | 0.0762 | 0.1462 |
| kmeans_entity_bias(k = 3, 10k)_page_top100 | 0.0822 | 0.1393 | 0.4054 |
| ranklib_section_top100 | 0.098 | 0.0787 | 0.1371 |
| ranklib_section_top1000 | 0.0986 | 0.0787 | 0.1375 |

***Sangeeta***

**a. Re-rank by DBpedia Type/ Re-Rank by DBpedia Type and BM25 similarity with weight**

Introduction
DBpedia spotlight, is a system that automatically annotate text documents with DBpedia URIs. It gives a wide range of entity and type relationship. This relationship is used in this method to check the relevance of a paragraph.

Implementation at hierarchal level (Improvement over Prototype 2)
This method was implemented for three run files:

- Section_runfile(Heading weigth)
- Lowest_Heading_Section_Runfile
- lucene1--paragraph-section--sectionPath-ql-none--Text-std-k1000-benchmarkY1test.v201.cbor. outlines (Top 100 paragraphID)

Evaluation:
Train Data:

| Method | MAP | MRR | Rprec |
|---|---|---|---|
| Re-rank by DBpedia type CK section | 0.1044 | 0.1478 | 0.0735 |
| Re-rank by DBpedia Weight CK section | 0.0062 | 0.0106 | 0.0011 |
| Re-rank by DBpedia type HW section | 0.1042 | 0.1483 | 0.0769 |
| Re-rank by DBpedia Weight HW section | 0.0062 | 0.0011 | 0.0100 |
| Re-rank by DBpedia type Lowest HW section | 0.0585 | 0.0802 | 0.0426 |
| Re-rank by DBpedia Weight Lowest HW section | 0.0095 | 0.0125 | 0.0045 |

Conclusion:
This method gives better result for section (100 Paragraph ID) than pages (20 paragraph IDs). It is improvement over Prototype 2.

**b. Re-rank by TagMe Type**

Introduction
Tagme, an entity linking system that is able to efficiently and judiciously augment a plain-text with pertinent hyperlinks to Wikipedia pages. The specialty of Tagme with respect to known

systems is that it may annotate texts which are short and poorly composed. This annotation is extremely informative, so any task that is currently addressed using the bag-of-words paradigm could benefit from using this annotation to draw upon (the millions of) Wikipedia pages and their inter-relations.

Implementation

Pre-requisite for this method:

- Register to tagme.d4science.org.
- The organization will provide a secret token number, which is used in the code to extract entities when given a text.
- For every text one need to hit https://tagme.d4science.org/tagme/tag?lang=en&gcube-token=3d193770-ce77-41a7-a433-91f8074d6d28-843339462 to get the entities where Token = 3d193770-ce77-41a7-a433-91f8074d6d28-843339462.

This task uses TagMe variation and the frequency of its type in each paragraph to re-rank the run file.
1)A search method was implemented, "BM25" similarity of "lucene" was used to compute similarity between query and paragraph to generate a baseline run file(Section).
2) The paragraph IDs of the baseline run file were compared with the "dedup.articlesparagraphs.cbor" to get the text content.
3) For each paragraph ID, entities and its respective type was gathered using TagMe spotlight.
4) The frequency of TagMe entities was clustered for each paragraph ID in the run file.
5) The run file was then re-ranked as per the descending order of the frequency of type in a paragraph given a query. This method was implemented for top 100 paragraph IDs.
6) This method was implemented for three run files:

- Section_runfile(Heading weigth)
- Lowest_Heading_Section_Runfile
- lucene1--paragraph-section--sectionPath-ql-none--Text-std-k1000-benchmarkY1test.v201.cbor. outlines (Top 100 paragraphID)

**c. Re-Rank by TagMe Type and BM25 similarity with weight**

Implementation

This task uses TagMe variation for the frequency of its type and and BM25 similarity score in each paragraph to re-rank the run file.
1)A search method was implemented, " BM25" similarity of "lucene" was used to compute similarity between query and paragraph to generate a baseline run file.
2) The paragraph IDs of the baseline run file were compared with the "dedup.articlesparagraphs.cbor" to get the text content.
3) For each paragraph ID, entities and its respective type was gathered using TagMe spotlight.
4) The frequency of TagMe type was clustered for each paragraph ID in the run file.
5) The run file was then re-ranked as per the descending order of the frequency of type in a paragraph given a query.
6) The run file generated was further processed to get a better score were both the rank and BM25 score could be used to get an appropriate ranking.

New score = BM25 similarity score + 1/(Rank of the para id as per frequency of type)
7) The New score was then used to re-rank the run file. This method was implemented for top 20 paragraph IDs.

Evaluation:

| Method | MAP | MRR | Rprec |
|---|---|---|---|
| Re-rank by TagMe type CK section | 0.1040 | 0.1475 | 0.0735 |
| Re-rank by TagMe Weight CK section | 0.0090 | 0.0025 | 0.0148 |
| Re-rank by TagMe type HW section | 0.1039 | 0.1481 | 0.0769 |
| Re-rank by TagMe Weight HW section | 0.0098 | 0.0166 | 0.0039 |
| Re-rank by TagMe type Lowest HW section | 0.0561 | 0.0801 | 0.0426 |
| Re-rank by TagMe Weight Lowest HW section | 0.0091 | 0.0123 | 0.0046 |

Test Data:

| Method | MAP | MRR | Rprec |
|---|---|---|---|
| Re-rank by TagMe type HW section | 0.1194 | | 0.0916 |
| Re-rank by TagMe Weight HW section | 0.0090 | | 0.0014 |

Conclusion:
The first method was implemented to count the frequency of entities in a paragraph and rank it in descending order of frequency of entities. The second method was implemented to showcase the importance of the knowledge base categorization considering BM25 score with help of weights. It was assumed that higher the frequency of the entities in a paragraph, more relevant. But at the same time the BM25 score cannot be ignored. So, weights were provided it was re-ranked according to the formula mentioned. This also provides score like baseline method. There is no significant improvement but there is no decrease in the measures
For both the variation, the first method performed better than second.

***Nithin***

## Implementation

Steps includes

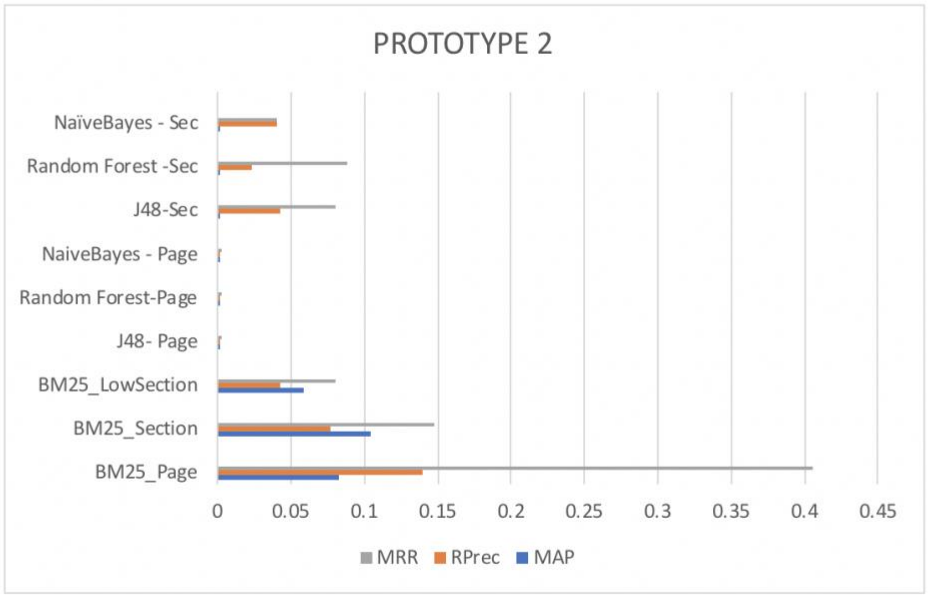
1. We have used train v2.0 of the TREC -car dataset to train the classifier.
2. To train the classifier we considered base.train.cbor-paragraphs.cbor of the train data.
3. Here the paragraph is the text and the heading corresponding to the paragraph are the class values.
4. The training set is created.
5. Now feed these training set to the above-mentioned classifiers.
6. It takes lot of time to build the classifier for the training set is generated because of the huge file size. This time I took 30,000 paragraphs which is still less when compared to the whole train v2.0.
7. Now the classifier is trained with the training set of 30000 paragraphs,
8. The base line method of bm25 is run and the results are evaluated.
9. The baseline method is again run and the each paragraph retrieved from the method are feed to the test set as instance and each instance is classified.

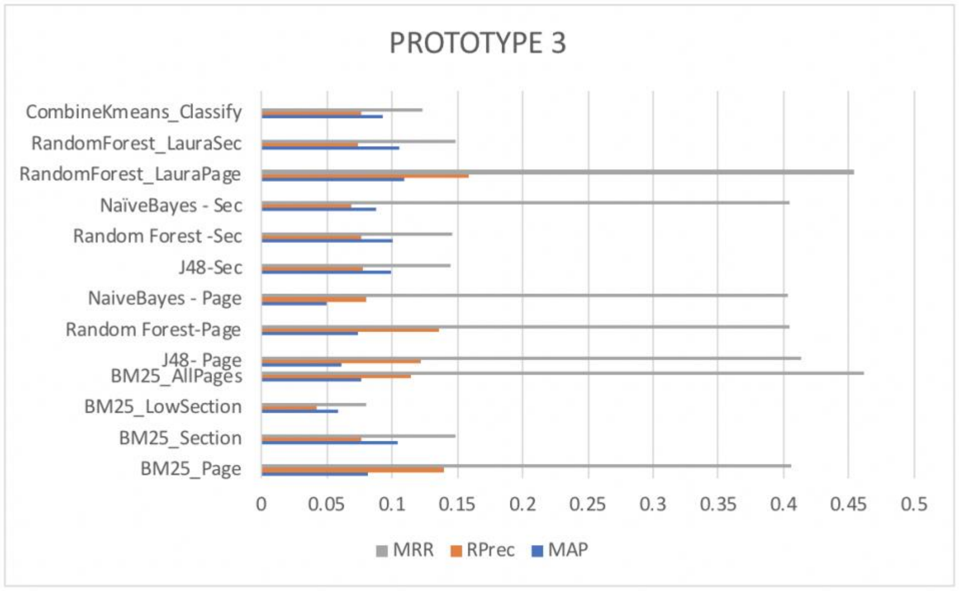
10. If the prediction level of the instance is greater than 0.5 then the prediction result is added to the run file by adding relevance score of 1. Otherwise the paragraph retrieved from the Lucene result is written to the file with the similarity score.
11. 
12. All the results are written to the run file and evaluated against the train qrel files.

Evaluation
My methods are evaluated with respect to MAP, RPrec, and MRR. All these below methods are evaluated against train ground truth qrel files.

| Method # | Method | MAP | RPrec | MRR |
|---|---|---|---|---|
| 1. a | BM25_Page | 0.0821 | 0.1393 | 0.4054 |
| 1. b | BM25_Section | 0.1039 | 0.0769 | 0.1481 |
| 1. c | BM25_LowSection | 0.0585 | 0.0426 | 0.0802 |
| 2. | BM25_AllPages | 0.0767 | 0.1149 | 0.4616 |
| 3. a | J48- Page | 0.0612 | 0.1222 | 0.4129 |
| 3. b | Random Forest-Page | 0.0743 | 0.1365 | 0.4045 |
| 3. c | NaiveBayes - Page | 0.0501 | 0.0805 | 0.4027 |
| 4. a | J48-Sec | 0.0988 | 0.0775 | 0.1445 |
| 4. b | Random Forest -Sec | 0.1011 | 0.0767 | 0.1458 |
| 4. c | NaïveBayes - Sec | 0.0881 | 0.0685 | 0.4041 |
| 5. a | RandomForest_LauraPage | 0.1093 | 0.1591 | 0.4533 |
| 5. b | RandomForest_LauraSec | 0.1050 | 0.0735 | 0.1482 |
| 6. | CombineKmeans_Classify | 0.0928 | 0.0762 | 0.1233 |

Charts:
Here I am providing the performance of the same classifier in prototype 2 and prototype. We can see the drastic change of the results.





Results

The evaluation results for the classifier are better when the model is bigger (ie) more training data.