MONGOOSE PROTOTYPE 3

Team 3

Prototype submission for partial fulfillment of the requirements of CS980.01

Spring 2018

SUMANTA KASHYAPI
SHUBHAM CHATTERJEE
TARUN PRASAD GANESA PANDIAN
AJESH VIJAYARAGAVAN

CONTENTS

1. Introduction

1.1. Task Definition

Given an article stub Q, retrieve for each of its sections H_i, a ranking of relevant passages. The passage is taken from a provided passage corpus.

1.2. Methods used

a) Candidate set generation using BM25, various language models and Pseudo-Relevance Feedback with Query expansion using KNN.
b) Weighted PageRank Algorithm
c) Personalised Page Rank Algorithm
d) Entity Linking using DBpedia
e) LDA topic models with and without WordNet synonyms
f) Named Entity Recognition

2. Approach

2.1. Candidate set generation

For candidate set generation, we explored several variations of the basic methods. We used BM25 and Language Models such as LM-DS and LM-JM to retrieve top 100 paragraphs for each page title and then applied query expansion using the following methods:

a) KNN with incremental expansion
b) KNN with extended query set
c) Entities from DBpedia
d) Datamuse

We experimented with variations of each of these and found that none of these could beat the basic BM25 model in performance (measured in terms of MAP). However, combining each of these using RankLib leads to better performance. The results for each variation experimented with is shown in table-1. The BM25, LM-DS, LM-JM and RM3 models are standard information retrieval models and hence we refrain from further discussing these models. However, a description of how each method was applied for the task of query expansion follows.

2.1.1. Query Expansion using KNN

Let the given query Q be$\{q_1, \dots, q_m\}$. In this approach, we define the set C of candidate expansion terms as

$$C = \bigcup_{q \in Q} \boldsymbol{NN}(q)$$

where $\boldsymbol{NN}(q)$ is the set of K terms that are nearest to q in the embedding space. We use *GloVe* to obtain the word vectors for each word. For each candidate expansion term t in C, we compute the mean cosine similarity between t and all the terms in Q using the equation:

$$Sim(t, Q) = \frac{1}{|Q|} \sum_{q_i \in Q} \boldsymbol{t}.\boldsymbol{q_i}$$

The terms in C are sorted by this mean score and the top K candidates are chosen as the actual expansion terms. We use a set of pseudo-relevant documents (PRD) that are retrieved at top ranks in response to the initial query to restrict the search domain for the candidate expansion terms. Instead of searching for nearest neighbor within the entire vocabulary of the document collection, we consider only those terms that occur within PRD. The size of PRD may be varied as a parameter.

2.1.1.1. Incremental KNN based approach

Instead of computing the nearest neighbors for each query term in a single step, we follow an incremental procedure. The first assumption in this method is that, the most similar neighbours have comparatively lower drift than the terms occurring later in the list in terms of similarity. Since the most similar terms are the strongest contenders for becoming the expansion terms, it may be assumed that these terms are also similar to each other, in addition to being similar to the query term. Based on the above assumption, we use an iterative process of pruning terms from$\boldsymbol{NN}(q)$, the list of candidates obtained for each term q.

We start with $\boldsymbol{NN}(q)$). Let the nearest neighbours of q in order of decreasing similarity be $t_1, t_2, \dots, t_N$. We prune the K least similar neighbours to obtain $t_1, t_2, \dots, t_{N-K}$. Next, we considert_1, and reorder the terms $t_2, \dots, t_{N-K}$ in decreasing order of similarity with t_1. Again, the K least similar neighbours in the reordered list are pruned to obtain $t'_2, t'_3 \dots, t'_{N-2K}$. Next, we pick t'_2 and repeat the same process. This continues for l iterations. At each step, the nearest neighbours list is reordered based on the nearest neighbour obtained in the previous step, and the set is pruned. Essentially, by following the above procedure, we are constraining the nearest neighbours to be similar to each other in addition to being similar to the query term. A high value of $l \geq 10$ may lead to query drift. A low value of $l \leq 2$ essentially performs similar to the basic pre-retrieval model. We empirically choose $l = 5$ as the number of iterations for this method. Let $NN_l(q)$denote the iteratively pruned nearest neighbour list for q. The expanded query is then constructed as in Section 2.1.1, except that $NN_l(q)$ is used in place of $NN(q)$in the equation. See table 1 and 2 for results.

2.1.1.2. Extended Query Term Set

Considering NNs of individual query word makes a generalization towards the process of choosing expansion terms since a single term may not reflect the information need properly. For example, consider the TREC query *Orphan Drugs* where the respective terms may have multiple associations, not related to the actual information need. The conceptual meaning of composition of two or more words can be achieved by simple addition of the constituent vectors.

Given a query Q consisting of m terms $\{q_1, \dots, q_m\}$., we first construct Q_C, the set of query word bigrams.

$$Q_C = \{\langle q_1, q_2 \rangle, \langle q_2, q_3 \rangle, \dots, \langle q_{m-1}, q_m \rangle\}$$

We define the embedding for a bigram $\langle q_i, q_{i+1} \rangle$as simply $\boldsymbol{q}_i + \boldsymbol{q_{i+1}}$, where $\boldsymbol{q_i}$ and $\boldsymbol{q_{i+1}}$ are the embeddings of words q_i and q_{i+1}. Next, we define an extended query term set (EQTS) Q' as

$$Q' = Q \cup Q_C$$

For the proposed approaches, the effect of compositionality can be integrated by considering Q' of in place of Q in section 2.1.1. See table 1 and 2 for results.

2.1.2. Query Expansion using entities from DBpedia

This method takes in the page titles from the cbor file, and uses the indexed paragraph corpus to query and generate the candidate set. The first step of the process is to create the expanded query. We use the page title as the initial query, and retrieve the top 5 paragraphs from the indexed paragraph corpus using BM25 similarity. This is stored in a Hash Map with the page title as the key and the top 5 paragraphs as the value. The second step is to walk through the map, and extract entities from the paragraph text using DBPedia. This produces a map of page titles and entities under each of them. As we retrieve the top 5 paragraphs, there is a good chance we end up with repeating entities. Thus, for each of the pages, unique entities are extracted. We concatenate the page title and the related entities together to form the query string. We search the indexed paragraph corpus with this query, and retrieve the top 200 paragraphs, which is written into a run file. The output obtained is a list of paragraphs (up to 200) for each of the page title's, which is used as the candidate set for other methods. See table 2 for results.

Pseudocode:

```
for every page:
    retrieve top 5 paragraphs using BM25
    store as (page : para1, para2, .. , para5)
for every page:
    for every paragraph:
        extract entities using DBPedia
        store as (entities : e1, e2, .. , en)
    store as (page : entities)
for every page:
    generate query q = page title + entity1 + entity2 + … + entity n
    retrieve top 200 paragraphs using BM25 with query q
    store as (page : para1, para2, … , para N)
```

2.1.3. Query Expansion using DataMuse API

The method takes in Page names as input. To have maximum precision value for the candidate set, the query used to search and retrieve paragraph ids needs to be expanded to get most relevant paragraph ids with high priority. The query should be expanded with terms on different contexts including synonyms, hyponym, spelling, sound and vocabulary. ***Datamuse*** is a web based collection that has 10 million indexed terms that can be queried on different contexts such as similar to, related to, all words in hierarchical mapping of relations, explanation based on a term including additional term suggestions depending on user's query. On querying the API with different combination of terms, the JSON response array has related terms on the contexts. See table 2 for results.

/words?ml : Provides synonyms,

rel_[code]: Related word constraints.

?sl: Sound.

?sp: with same Spelling.

?topics: Topic words.

?lc,? rc: Left and right context.

?md: metadata.

Pseudocode:

```
For each page in deserialized iterable annotations of train.pages.cbor-outlines.cbor file:
    Pass page and section path values to Query string builder to get Page name query
    Add query string to datamuse api
    Get request to retrieve response in JSON array
    For JSON array length:
        Extract terms from JSON object and string.
        Add terms to list.
    Expand query with retrieved terms
    Searcher with query string for 200 relevant paragraphs
    For score document length: number of top paragraphs:
        Retreive ids and write to run file.
```

2.2. Combining Personalized PageRank scores with other methods

We obtained a candidate set using various methods (BM25, LM-DS, LM-JM) using page titles as queries and applied the Personalized PageRank algorithm to the graph of paragraphs obtained. The graph is defined as a set of nodes (which consist of paragraphs) and a set of edges (entities shared by the paragraphs). An edge is added between two nodes, if the paragraphs representing the nodes have at least one entity in common. The seed set for the PPR algorithm is obtained by considering the top 10 paragraphs retrieved. This score for each paragraph is combined with the score obtained by other methods as a weighted score to obtain the final score as follows:

$$final\ score = (\ 0.8 * score1\ ) + (\ 0.2 * score2\ )$$

where $score1$ is the score obtained from some other method and $score2$ is the Personalized PageRank score. See table 2 in section 3.6 for results.

2.3. Ranking paragraphs in each cluster using Weighted PageRank

The method takes in clusters of paragraphs, and ranks the paragraphs inside the clusters using the Weighted PageRank algorithm. This algorithm is a variation on the simple PageRank Algorithm where the edges are now weighted by the number of entities they share or the number of words they have in common. Using the candidate input set, the clustering algorithms produce a set of clusters within each page. Based on the clustering algorithm used, we get different number of clusters inside each page. This is the input to the PageRank algorithm. The input is structured as follows: we have multiple pages identifiable by their page ID. Each of these pages have single/multiple clusters. Each of the clusters have single/multiple paragraphs, identifiable by their paragraph ID.

The paragraphs within the clusters are not ranked by any order. We create a graph of these paragraphs, using the paragraphs as nodes. The edges between them are derived using the two variants described below. Each variant produces a different graph structure, leading to different ranking of the paragraphs inside the cluster.

a) DBPedia Spotlight

This process is done for every cluster inside the page. Using the paragraph ID, the paragraph text is extracted from the Lucene Index, which is passed as the input to DBPedia Spotlight. The output from DBPedia Spotlight is a list of entities found in the paragraph. This is stored, and the process is repeated for all the paragraphs inside the cluster. The list of paragraphs and their entities are analyzed. Paragraphs having same entities are linked with each other. The edges are weighed by the number of entities common.

b) Common Words

This process is also done for every cluster of paragraphs inside the page. The words from each of the paragraphs are extracted, and stored along with their paragraph IDs. The number of common words between each of the paragraphs is computed, and stored in a matrix. The average number of common words between paragraphs in the cluster is calculated. The paragraph pairs which have a higher number of common words than this average value, are linked with each other and the edges are weighed by the number of words common.

Using this graph structure, we apply the PageRank algorithm. The score for each of the paragraph is taken from the PageRank vector, which is also used in ranking these paragraphs. A pseudocode describing the method is given below. The results are shown in table 3.

```
Cluster Ranking
```

```
for every page:
    for every cluster:
        for every paragraph:
            (1) extract list of entities
            (2) extract list of words
            store as (P : <list>)
        store as (C : (P:<list>, P:<list>, P:<list>, .. ))
        apply weighted page rank
        store as (C : P1, P2, .. Pn)
        return ranked clusters
write the result into run file
```

Weighted PageRank

```
for every paragraph:
    for every paragraph:
        count common entity/words
        initialize weight matrix
        initialize transition probability matrix
    initialize pagerank vector
compute pagerank using weight matrix, transition probability matrix
and pagerank vector
get pagerank score for each node and store as (P1 : score, P2 : score,
.. , Pn : score)
return pagerank scores
```

2.4. LDA TOPIC MODELS

1. For each page, we generate topic distribution for each paragraph i in candidate set tp_i.
2. (i) For each page, we generate the topic distribution for each section headings j in candidate set ts_j.

   (ii) For each page, we generate topic distribution for each section headings j in candidate set but this time expanding it with the heading word's synonyms from WordNet tss_j .

3. Obtain ranking for each section using
   i) $KLDivergence(ts_j, tp_i)$

ii) $KLDivergence(tss_j, tp_i)$

as rank score.

See the graph below for results.

2.5. NAMED ENTITY RECOGNITION

Adding weight to Named Entities such as person, organization, place that are extracted using a NER api from a retrieved paragraph which are related to the Query. These added weights will change the ranking of paragraphs to provide high priority to relevant paragraphs.

Pseudocode:

```
For each paragraph in paragraphs file:
    If collected list of candidate paragraph list contains the paragraph id
        Retrieve text and add to list
For each element in collected list of paragraph text:
    Concatenate with url for NER api with token
For each complete url in the list:
    HTTP request and get response as JSON object
    Extract derived named entity count
    Insert in hashmap for that paragraph id to rerank with added weight.
```

2.6. PARAGRAPH SIMILARITY EXPERIMENTS

This study is carried out to examine similarity between all paragraph pairs so that it can help in our clustering methods. The algorithm in pseudocode is given below. The results are shown in table 4.

Pseudocode

```
For all wordnet similarity:
For all page in pages:
   for all section in top level sections of page:
       for all paragraphs p1 in sections:
             1) compute the similarity between p1 and any paragraph p2
on the page
```

```
            2) interpreting p1 as "query" rank all paragraphs p2 by
the similarity

            3) interpreting all paragraphs in the section as "true"
and all other paragraphs as "false",compute

                 3.1) Mean average precision (MAP) of the ranking.

                 3.2) Precision@5 of the ranking

                 3.3) Precision@R of the ranking
```

3. Evaluation

3.1. Data set

We use the ground truth files from the benchmarkY1-train for evaluation and the page-level and hierarchical section level training files for k-fold cross validation and training our methods.

3.2. Corpus

The *paragraphCorpus* is used as the corpus.

3.3. Queries

The section-path names are used as queries for sections whereas the page names are used as queries for articles.

3.4. Ground truth

The files *train.pages.cbor-hierarchicall.qrels* is used as the ground truth for section name queries and *train.pages.cbor-article.qrels* is used as the ground truth for page level queries.

3.5. Evaluation measures used

We use the following three evaluation measures:

- Mean Average Precision (MAP)
- Precision at R (RPrec)
- Reciprocal rank (recip_rank)

3.6. RESULTS

3.6.1. Evaluation results for candidate set generation using page names as queries

Features	MAP	Rprec	recip_rank
BM25 **BM25+KNN-INC** **BM25+KNN-EXT**	0.0877	0.01449	0.04501
LM-DS **LM-DS+KNN-INC** **LM-DS+KNN-EXT**	0.0933	0.1520	0.4349
LM-JM **LM-JM+KNN-INC** **LM-JM+KNN-EXT**	0.0699	0.1202	0.3397
BM25 **BM25+KNN-INC** **BM25+KNN-EXT** **LM-DS** **LM-DS+KNN-INC** **LM-DS+KNN-EXT**	0.0981	0.1575	0.4865
BM25 **BM25+KNN-INC** **BM25+KNN-EXT** **LM-DS** **LM-DS+KNN-INC** **LM-DS+KNN-EXT** **LM-JM** **LM-JM+KNN-INC** **LM-JM+KNN-EXT**	0.0985	0.1578	0.4996
BM25 **BM25+KNN-INC** **BM25+KNN-EXT** **BM25+KNN-PRF** **BM25+RM3**	0.0884	0.1473	0.4956
LM-DS **LM-DS+KNN-INC** **LM-DS+KNN-EXT** **LM-DS+KNN-PRF** **LM-DS+RM3**	0.0942	0.1530	0.4943
LM-JM **LM-JM+KNN-INC** **LM-JM+KNN-EXT** **LM-JM+KNN-PRF** **LM-JM+RM3**	0.0677	0.1161	0.3455

BM25 LM-DS	0.0977	0.1564	0.5065

Table 1: RankLib Combined Results

Variation	MAP	Rprec	recip_rank
BM25+KNN-INC+PRF	0.0760	0.1236	0.3493
BM25+KNN-EXT+PRF	0.0817	0.1281	0.4208
LM-DS+KNN-INC+PRF	0.0665	0.1075	0.3044
LM-DS+KNN-EXT+PRF	0.0650	0.1109	0.3306
LM-JM+KNN-INC+PRF	0.0601	0.0986	0.3038
LM-JM+KNN-EXT+PRF	0.0620	0.1009	0.3336
BM25+DBpedia	0.0567	0.0985	0.3478
BM25+DataMuse	0.0792	0.0183	0.0332
BM25+PPR	0.0040	0.0058	0.0303
LM-DS+PPR	0.0044	0.0049	0.0235
LM-JM+PPR	0.0027	0.0024	0.0070

Table 2: Individual Results

3.6.4. Evaluation results for ranking within clusters

Method	Using DBpedia			Using common words		
	MAP	Rprec	recip_rank	MAP	Rprec	recip_rank
HAC + WordNet Similarity	0.0072	0.0044	0.0146	0.0083	0.0054	0.0167

Table 3: Ranking within clusters

3.6.5. Evaluation results for experiments in section 2.4

Method	MAP	P@5	P@R
Word2Vec Similarity	0.0104	0.0128	0.108
Random (baseline)	0.0069	0.0061	0.0064

Table 4

3.6.6. Evaluation results for running prototype on *benchmarkY1-train* for hierarchical queries after combining with RankLib

- MAP = 0.0811
- Rprec = 0.0660
- Recip_rank = 0.1287

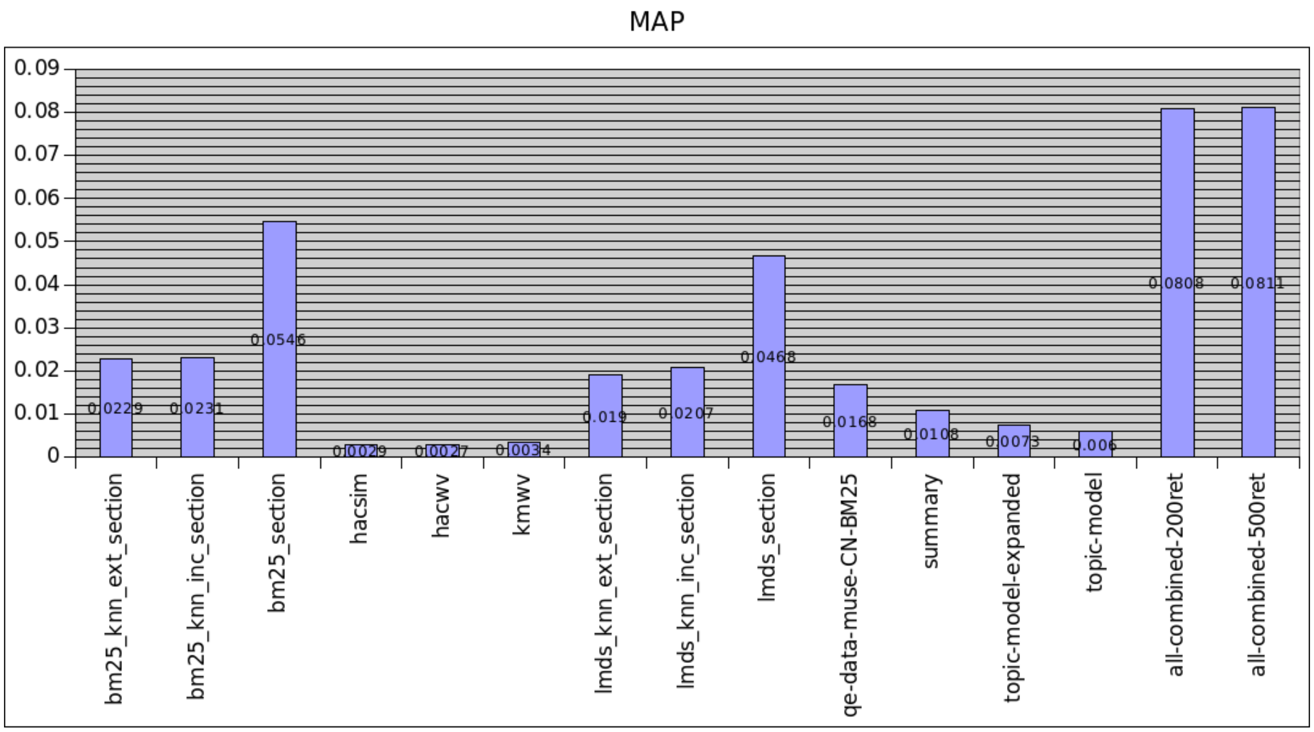

Figure 1: Comparison of MAPs of different methods on training data

APPENDIX A

Installation Instructions

Follow the instructions given below to install and run the prototype:

Step-1: Clone the project from GitHub

```
git clone https://github.com/nihilistsumo/Mongoose.git
```

Step-2: Change to Mongoose directory

```
cd Mongoose
```

Step-3: Change permission of the scripts

```
chmod 755 install.sh run.sh
```

Step-4: Run the installation script

```
./install.sh
```

Step-5: Run the prototype

```
java -jar target/Mongoose-0.0.1-SNAPSHOT-jar-with-dependencies.jar
```

For details, see the README file in the GitHub Repository.

APPENDIX B

Individual Contribution

- **Sumanta Kashyapi**: Topic models, changing toplevel section files to hierarchical, writing scripts, summarization, paragraph similarity study for clustering
- **Shubham Chatterjee**: Candidate set generation, Pseudo-Relevance Feedback, Query Expansion using KNN, Personalized PageRank for paragraphs, document preparation
- **Tarun Prasad Ganesa Pandian**: PageRank for clusters using DBpedia spotlight and common words, weighted PageRank
- **Ajesh Vijayaragavan**: Candidate set generation by combining Query expansion with DataMuse api terms with BM25 similarity, Named Entity Recognizer for Paragraphs.