Prototype Report: 3

Team No:1
GitHub Repository: https://github.com/ba1044/DataScience_FinalProject
Prototype 3 Release: https://github.com/ba1044/DataScience_FinalProject/releases/tag/Prototype3

Benchmark Test Runfiles:

The benchmark public test run files are located in results_prototype_3/public_test

Method Description:

Topic Retrieval

In the embedding methods below, I try to embed paragraphs/queries into spaces of topics. I define a topic as a collection of related paragraphs. In particular, I use abstracts of Wikipedia pages that are related by some category. For example, Medicine might contain abstracts from pages such as Medicine, Hospitals, Surgery, etc. In total, each topic is a collection of 50 abstracts.

I try to filter the abstracts by using vrank (see http://www.6020peaks.com/vRank/), such that the 50 abstracts are the top 50 abstracts according to vrank (that match the dbc category of the topic, such as Medicine).

Abstracts are stored in the paragraphs/ directory, where each subdirectory is a topic. The sparql_downloader command is used to download all of this, but I have the resources on the server (see last page).

Embedding of Paragraphs Using Perturbation

To embed paragraphs in a space of topics, I devised the following:

- Represent a paragraph, P, and each topic, T_i, as distributions (typically unigram frequency or k-mer frequency).
- Create N copies of P and perturb each copy, P_n, by additive Gaussian noise.
- For each topic T_i, define a measure of distance to each P_n, restricted to the support of P (I use the L2 norm).
- Treat the measure of a topic T_i over all of P_n as a distribution.
- Using gradient descent, find a convex combination of these distributions that minimizes KLD to the uniform distribution (over P_n).
- Treat the set of weights obtained by gradient descent as the coordinates for a point on a simplex (dimension equal to the number of topics used).

This method is demonstrated in the perturbation_embedding method (in the table below). Unfortunately, it's really slow, so I had to significantly reduce the number of perturbations evaluated for each paragraph (which makes the final solutions more unstable / unlikely to be correct). How I use the method is to embed queries and paragraphs score them according to manhattan distance.

Embedding Using Hierarchical Methods

In the previous method, we can think of the final set of weights as a description of which topics are "most important" in generating a given paragraph (i.e. what are the causal factors). Using this method, I devised a way to decompose each topic in the following way:

- A topic is a collection of paragraphs (pertaining to the topic). Treat each paragraph in a topic as a distribution (over words or k-mers). Treat the topic itself as a distribution (concatenate all of its paragraphs and calculate unigram/k-mer frequency).
- Using the previous method, find the convex combination of paragraphs that best describes the topic.
- A paragraph is a collection of sentences, and a sentence is a collection of words. Repeat the previous steps to obtain the most important sentences in a paragraph and the most important words in a sentence.

Using this method, each topic is expressed as a hierarchical distribution. Let T(P) be probability of paragraph given topic, P(S) be probability of sentence given paragraph, and S(W) be probability of word given sentence. Then the probability of a word given a topic is (I'm probably going to mess this up): T(P(S(W))).

However, what I am after is a pullback using the inverse, such that if you have a probability over words, you can describe a probability over topics. I approximate this using the Normalized Levenshtien similarity metric from words in a query/paragraph to the most iportant words in these topics (I say a word is "present" in a topic model if any of the words in a query/paragraph have a similarity >= 0.8 to the word in question). In retrospect, I should have just used a unigram/k-mer frequency model...

Finally, I describe a distribution over topics with respect to these scores and treat this as the "embedding" in a space of topics. I then take the Manhattan distance between the embedded query and paragraphs and rank the paragraphs according to how close they are to the query.

My methods in the table below explore variations of this process:

- **hier_ascent**: Taking a similarity measure "backwards" equates to ascendy the hierarchy. In this method, I explore variations in how I transfer the measure backwards (from words to sentences, from sentences to paragraphs, and finally from paragraphs to topics). Normally I just sum up the measures over words in a sentence (multiplying by their probabilities in a hierarchy) to get a measure of a sentence, and repeat this process for paragraphs and topics. I explore alternatives to this process (only take the max / only take the values that meet a minimum threshold)
- **Hier_clusters**: I have too many topics (21!) so I tried to create separate features by partitioning them into clusters of topics.
- **Hier_subclusters**: Normally I am taking the distance between a query and paragraphs based on a distribution over topics. In this method, I instead do it based on a distribution over a topic's paragraphs.
- **Hier_metrics**: Normally I use manhattan distance between embedded queries / paragraphs. In this method, I explore using Euclidean / Manhattan / Minkowski / Cosine distance.

- **Hier_query_variations**: Normally, I just use the words in a query to embed a query. In this method, I try to alternative version (query expansion of query terms / also tried concatenating top 100 documents).
- **Hier_reduction**: These methods are variations in how I sum up my distances to words in a topic from words in a query/paragraph: take the max, take the average, add smoothing (similarity scores for small queries/paragraphs are "boosted").

Note:

In all of the embedding methods, I use learning to rank with BM25. The scores you see are with respect to choosing weights for the paragraph embedding features and BM25 and then running trec eval on the output. Unfortunately, I saw no evidence that my features made any significant contributions to MAP this time.

Contextual Similarity with Query Expansion:

Using Lucene to retrieve abstract text of entities in query (Created by Jordan), and use Spotlight API to annotate the abstract to get entities relevant to query. Then rank the entities to expand the query, finally run expanded query against index using BM25. Using Dependency parser to get the parsed tree of the paragraph and for ranking traverse the paragraph from root to top five tree level and find out the query word or query expansion word and ranked the paragraph based on the BM25 similarity. In this method, I propose a method for measuring contextual similarity, an important type of semantic relationship, between entities. we can locate the page relevant to an entity. Using their tree, the semantic similarity between entities can be measured in different dimensions. Currently I am using on the Page Queries, However My belief is it should work well on section level.

Contextual Similarity using Top level dependency parser tree:

In this method, Using Dependency parser to get the parsed tree of the paragraph of section path and for ranking traverse the paragraph from root to top three tree level and find out the root query word and rank the paragraph based on root query word with the BM25 similarity. In this method, I propose a method for measuring contextual similarity, an important type of semantic relationship, between entities. we can locate the page relevant to an entity. Using their tree, the semantic similarity between entities can be measured in different dimensions.

Similarity using Top words dependency Parser:

In this we are choosing some words from dependency parser. Dependency parser gives a nice tree and it also tells about the semantic structure of the tree.Which shows a hierarchical structure of the sentence. Instead of considering all the word of the paragraph, consider only, nsubj,cc,compound,root,nummod. Instead of taking all the paragraph in this method only focusing on the above word and based on the word doing ranking with BM25 Similarity.

Natural Language Processor with entities relationships:

This method will get a initial rank based on BM25, and then re-rank the documents with entities relationships. It will use DBpedia Spotlight API to recognise the entities in query, and use Natural Language Processor to retrieve all relationships in each document content from the initial run. If the entities in query participate in the relationship, then the score of this document will increased.
Object_count: how many relationships that entities participate in as object.

Subject_count: how many relationships that entities participate in as subject.

Final_score = initial_score + object_count * object_factor + subject_count * subject_factor
Finally, the method will create a re-ranked documents.

Contributions:

Jordan: Explored different kinds of methods for embedding paragraphs/queries into spaces of topics (through "hierarchical" methods and through the perturbation method I described previously).

Kevin: Update the Natural Language Processor for dependency tree and relation extraction.

Bindu: Explored Contextual similarity with Query expansion, with top level trees. I also choose different way to find highest ranking paragraph with only focusing on some part of dependency parser tree like nsubj, cc, compound, root, nummod because it makes more sense to find the related Query.

Teamwork:

Jordan: Other than organizing the repository, generating the run.sh file, and making sure everyone's code merged and ran smoothly… I wasn't a very good teammate this time. :(

Bindu: Using lucene index of entities and their associated method which was created by Jordan and used Kevin query expansion method which was useful for my contextual similarity with query expansion.

Kevin: Spotlight API method and natural language processing method for Jordan and Bindu.

Evaluation Results on BechmarkY1train:

Method	**MAP**	**MRR**	**Rprec**	**Author**
hier_ascent	0.1056	0.1502	0.0787	Jordan
hier_clusters	0.1081	0.1529	0.0823	Jordan
hier_metrics	0.1053	0.1506	0.0777	Jordan
hier_query_variations	0.1060	0.1510	0.0791	Jordan
hier_reduction	0.1047	0.1491	0.0774	Jordan
hier_subclusters	0.1061	0.1494	0.0799	Jordan
perturbation_embedding	0.1050	0.1488	0.0799	Jordan
Contextual_Similarity_ with_Query_Expansion	0.0676	0.3213	0.1135	Bindu+kaixin

Full Para Dependency Parser of section queries	0.0043	0.00351	0.0085	Bindu
Top_Para_Dep_Parser	0.0547	0.3036	0.1059	Bindu
NLP entities relation method - Page queries	0.0621	0.4152	0.1000	Kevin
NLP entities relation method - Section path queries	0.0968	0.1389	0.0753	Kevin

Required Files:

Precompiled prototype 3 jar
/trec_data/team_1/program.jar

Location to a compiled copy of trec_eval (used by run.sh)
/trec_data/trec_eval

Location to paragraph corpus (used by index command to create Lucene index)
/trec_data/paragraphCorpus/dedup.articles-paragraphs.cbor

Location to unprocessedAllButBenchmark (used by hyperlink_indexer and abstract_indexer)
/trec_data/unprocessedAllButBenchmark/unprocessedAllButBenchmark.cbor

Premade abstract Lucene index (made with abstract_indexer command)
/trec_data/team_1/abstract

Collection of Abstracts related to Topics (made with sparql_downloader)
/trec_data/team_1/paragraphs

Topics that have been decomposed into hierarchies (made with topic_decomposer)
/trec_data/team_1/descent_data

Premade Lucene index for Paragraph Corpus (made with index command)
/trec_data/team_1/myindex

Spotlight jar and required models
/trec_data/team_1/spotlight_server