Sourav Mangla
Netid: souravmangla

Jeopardy System Report 583

1. Instruction to run the source code: -

a. Clone the repository from one of the link:
http://www.github.com/ManglaSourav/Project583
https://drive.google.com/file/d/1GrTKsJxsg5uXQ16ut2ijnT1g87Nm_JnL/view?usp=sharing

b. Download the index from this link:
https://drive.google.com/file/d/1LEyqlrwPI8CCtFCnBi2ICKH7PFEEoKUK/view?usp=sharing

c. Download glove.6B pre-trained embeddings from the given link and put "glove.6B.100d.txt" file into resources:
https://huggingface.co/stanfordnlp/glove/resolve/main/glove.6B.zip

d. My resources directory:

e. To install dependencies: - "*mvn compile"*

f. Test index building:
   i. Copy the wiki data pages in resource folder (directory name should be "*wiki-data*")
   ii. Run the main method of "*IndexBuilder*" file.

g. Test the Engine:
   i. Copy the downloaded index and "*glove.6B.100d.txt"* file in resource folder (index directory name should be "*wiki-index*")
   ii. Run the main method of "**Main**" file/class.

h. I have also made a sanity check test case (Which check how many docs our index has), to run the test: "*mvn test*".

2. Source Code Explanation: -

My code has major 5 classes. These are the following:

a. ***IndexBuilder :-*** This is helpful in building our index and has following methods
   i. ***buildIndex***() :- This method is a configuration method; configuring the Lucene according to the project's need. Also, this method loads every wiki-data file and process them one by one by calling a helper method "*putFileToIndex*()".
   ii. ***putFileToIndex()*** *:-* This method process every file and does pre-process on the data like cleaning tags, special characters. Also, this method is finding wiki page starting point, categories inside a page and sub topics in the page. When everything is done, I'm adding this page to Lucene index by calling a helper method "*addDoc*()".
   iii. ***addDoc()*** :- This method is simply adding the wiki page data to Lucene by doing conditional stemming and lemmatization.

b. Engine :- This class measures the performance of my index by using different performance measures like P@1, MRR(mean reciprocal rank) and different scoring functions like BM25 and Boolean similarity.
   i. ***Engine():-*** It is a constructor and It loads the pre-built index and it also loads all 100 questions in the main memory from the file.
   ii. ***Pa1():-*** This method implements the P@1 performance measure.
   iii. ***MRR():-*** This method implements the MRR performance measure.
   iv. ***searchInLucene() :-*** This method searches our question's answer in the Lucene and return the result.

c. ***LanguageModel :-*** This class is an implementation of 5th part of the project. This class implement a language model on the top 10 documents returned by the Lucene. This model is using *Dirichlet* smoothing with *LMDirichletSimilarity* similarity provided by the Lucene.
   i. ***LanguageModel():-*** It loads the pre-built index and it also loads all 100 questions in the main memory from the file.
   ii. ***applyLM():-*** This method applies the Language model on query one by one and measures performance using *MRR* and I'm using *LMDirichletSimilarity* similarity as a scoring function which is giving me higher performance.
   iii. ***LM_With_Dirichlet_smoothing():-*** This method has actual model and smoothing implementation for a given question.

d. ***Word2Vec :-*** This class is an implementation of the bonus part of the project. This class implement a Word2Vec model on the top 10 documents returned by Lucene. This model is using pre-trained "glove.6B.100d.txt" embeddings to calculate question and document embeddings.
   i. ***Word2Vec():-*** It loads the pre-built index and it also loads all 100 questions in the main memory from the file.
   ii. ***avgEmbeddingForSentence ():-*** This is a helper function to calculate average embedding for a given sentence.
   iii. ***searchInLucene():-*** This method searches our question's answer in the Lucene and rerank them using word2vec embedding vectors for the docs and question and perform a cosine similarity between them.
   iv. ***calculateCosineScore():-*** This method calculates cosine similarity between question and document embedding vectors.

e. ***Utils :-*** This class has utility functions and constant variables like directory name or path which are used throughout the project. All data and index files must be inside the resource directory.

3. Results & Output: -

Scoring Function	Performance metric	Result
BM25	P@1	22%
BM25	MRR	28%
Boolean	P@1	13%
Boolean	MRR	18%
MultiSimilarity	P@1	19%
MultiSimilarity	MRR	29%
Language Model	MRR	41%
Word2Vec	MRR	25%

Sourav Mangla
Netid: souravmangla

```
It won't take more than 1 minute to run all the models
Applying BM25Similarity
[main] INFO edu.stanford.nlp.tagger.maxent.MaxentTagger - Loading POS tagger from
P@1: 0.22
MRR: 0.28495238095238096

Applying BooleanSimilarity
P@1: 0.13
MRR: 0.18233333333333332

Applying MultiSimilarity : BooleanSimilarity & BM25Similarity & TFIDF_Similarity
P@1: 0.19
MRR: 0.29581746031746026

Part 5 soln
wait....model is calculating the score
Language Model performance 0.4159404761904763

Bonus part soln
Word2vec is processing....
MRR for word2vec : 0.2475714285714285

Process finished with exit code 0
```

4. **Indexing and Retrieval: -** I tested all combinations of terms utilizing lemmas and stem words to construct my index terms, but without lemmatization and stemming, terms performed best for my use case. I'm inserting categories and headings to the page as I parse the content of the wiki page. I'm also adding categories to the index separately.
   I evaluated numerous ways for querying, such as obtaining documents without including clues in the query, however, including clues in the query improves performance. I also trimmed the query's whitespaces and lowered the case.
   I ran into problems with special characters, tags, and a lot of links in the data. I used regular expressions to get rid of most of them.

5. **Measuring Performance: -** Precision at 1 (P@1) and mean reciprocal rank (MRR) were used to evaluate the performance of my Jeopardy system. I didn't utilize MAP (Mean Average Precision) since each question only has one relevant document or answer, and MAP performs better when there are several relevant documents.
   MRR performs better in my use case since it checks for the correct documents at multiple ranks and has a higher likelihood of getting the correct document.

6. **Scoring Function: -** I experimented with many scoring functions, including BM25, Boolean, and TFIDF. I also used Lucene's MultiSimilarity feature, which allows us to aggregate numerous scoring functions into one.
   In my use case, BM25 and MultiSimilarity produced consistently good results, while Boolean and TFIDF have not. BM25 works better because it is combination of both vector space model and Boolean model. Using these models BM25 gives better relevance for the documents.

Sourav Mangla
Netid: souravmangla

7. **Error Analysis: -** In terms of MRR performance metrics, I'm getting 45 questions answered right in the top 10 documents, with 22 of those being correct at the top.

   I think the questions can be answered by such a simple system because of their well-designed wiki data structure and scarcity of noise, simple systems frequently perform effectively. In addition, asked questions have a clue with them which increase the likelihood of better results

   These are the following reasons I think, why my system is not getting other answers correct:-
   a. The length of some questions is small and to get the right answer Lucene needs longer phrases. these are some examples of the question for which I am getting the wrong answer:
      jell-o, post-it notes, 1983: "beat it".
   b. Some answers were incorrect because of proximity and have similar content, question and clue. For example: They're all talking about art and museums
      The Taft Museum of Art, The Georgia O'Keeffe Museum, The Kalamazoo Institute of Arts, The Sun Valley Center for the Arts, The Naples Museum of Art
   c. Lucence is not good at predicting locations on the basis of small information for example:
      The High Kirk of St. Giles, where John Knox was minister, Matthias Church, or Matyas Templom, where Franz Joseph was crowned in 1867, In this Finnish city, the Lutheran Cathedral, also known as Tuomiokirkko.
   d. Lucene is also not good at predicting celebrity names. For example :
      1980: "Rock With You", 1988: "Father Figure", 1988: "Man In The Mirror", 1989: "miss you much"
   e. Many questions have short or shared clues resulting incorrect answers.

   I tested all combinations of terms utilizing lemmas and stem words to construct my index terms, but without lemmatization and stemming, my model is performing better.
   The best configuration of my model is using multisimilarity or BM25 similarity using MRR performance measure with no stemming and no lemmatization and getting near 30% score. I think stemming can be successful in some cases, but not always, like two words that should be stemmed to the same root are not.

8. **Improved IR Model: -** To improve my IR system, I constructed a language model. In this model, I'm applying a language model that uses Dirichlet smoothing with Lucene's LMDirichletSimilarity similarity to rerank the top 10 documents. With an MRR performance score of 41%, this model has the best performance of the models I've tried.

9. **Word2Vec:-** To improve my IR system, I added a word2vec model. Using the word2vec embedding vector, I reranked the top 10 Lucene documents in this model. Using pre-trained embeddings "glove.6B.100d," I constructed embeddings of doc sentences and questions in this manner. I'm reranking those documents using the cosine similarity score between question and document embedding vectors, and I'm receiving about a 25% MRR performance score.