Information Retrieval

Non-factoid Answer Passage Retrieval

1. Introduction

The goal of this project is to implement a Factoid Question Answering system that answers questions given in a natural language. The dataset is a collection of documents taken from Wikipedia further separated into multiple passage. The system tries to find the passage that will best answer the given question.

Our system is based fully on classical information retrieval techniques.

1. Preprocessing

The indexer iterates over all the documents in the corpus. Each document passages are fused into one text and is further processed.

All stop words are removed from the text along with any characters that are not alphabetic or numeric.

Next, the text is split into tokens separated by spaces and the tokens are then stemmed.

Technology used include, PyStemmer, nltk stopwords , nltk word\_toknizer

3 Document Indexing

This phase includes creating the inverse indexing and corpus TF-IDF values.

After the preprocessing step, we have the full corpus available for processing.

The inverse indexing is stored in a trie-tree that is implemented in python module marisa-tree. We go over each token in each document and save its document id and position in the corresponding node.

For TF-IDF we have used python sk-learn TfidfVectorizer. This implementation accepts documents corpus and calculate each token corresponding value.

4 Document Ranking

Ranking is done with the help of TF-IDF scoring and sliding proximity window.

The first step is to filter the relevant documents that need to be further examined. Such filtration is done using the inverse index we have prepared.

First, we process the query input as we did in the preprocessing step, removing the stop words and stemming the tokens. We must perform this operation so the inverse index and the query tokens will be matched correctly.

After we have processed the query tokens, we find the tokens average TF-IDF values. We proceed with the query tokens that have TF-IDF value higher than the mean of all the query tokens.

We use the inverse index to extract the documents id that contains these tokens.

with the filtered documents in hands we base the scoring on 2 criteria

TF-IDF Scoring: We create a vector of the TF-IDF value for each document an use cosine similarity between the document and processed query tokens. The similarity is in the range of 0-1.0 and will be used as a score with the weight if 0.8.

Sliding proximity Window: Remember that the inverse index contains the position of the words in each document. We traverse all the positions that the query tokens appear in the filtered documents and in a window of 5 after and 5 before position search for the other query tokens and increase the counter by 1. We normalize this score by dividing in the number of query tokens.

We do that for a window of 6, 10 and 40 with weights of 0.5 0.4 0.1 respectively.

The score is in the range of 0-1.0 and will be used as a score with the weight if 0.2.

Once we have found the top documents we proceed with the 2 documents with the highest score to passage processing.

5 Passage processing

This is mostly similar to the preprocessing step. However, we have found that TF-IDF is not very effective because all passages are about the same topic, hence using the same terms.

What we do however is building an n-gram structure for each passage by utilizing sk-learn CountVectorizer

6 Passage Ranking

The scoring is based upon 2 criteria again

Ngrams: The query tokens are turned into ngrams and vectorized. We again find the cosine similarity of each document with the query tokens n-grams vector. (up to n=7)

The similarity is in the range of 0-1.0 and will be used as a score with the weight of 0.8.

Query Expansion: we expand the query tokens by using NLTK WordNet API. This API returns for each token its synonyms. The synonyms are tokenized similarity to previous steps.

Like we did with the ngrams, this time we only use unigram for the synonyms. So essentially, we check if the synonyms appear in the passage tokens.

The score is in the range of 0-1.0 and will be used as a score with the weight of 0.2.

7 Conclusion

We have found an accuracy of 25% using this method.

REFERENCES

[1] Project GitHub: <https://github.com/lichguard/APR>.