Research Challenge - Information Retrieval 2019006752

JOÃO ANTONIO OLIVEIRA PEDROSA, Universidade Federal de Minas Gerais, Brasil

This challenge aims to assess the student's skills as a search engineer based upon the knowledge acquired during the course. In particular, the challenge focuses on the problem of entity search in a knowledge base. Given a query q and an entity corpus D, the goal is to produce a ranking of entities sorted in decreasing order of relevance with respect to q.

1 TECHNIQUES

The 5 most relevant techniques used throughout the development of the challenge will be described in this document. Since code has changed a lot during implementation, and due to the experimental nature of the task, the source code that will be provided will contain many files that were used during the implementation but will not necessarily contain all code necessary to reproduce all the techniques here described. Nevertheless, the most effective submission should be reproducible with the source code provided.

1.1 First Attempt: BM25

Submission name in Kaggle: **16-58.csv**
Average nDCG@100: **0.28853**

This submission has been made using a library that all submissions in this report will use. It's name is **Pyserini** and its documentation can be found here.

This library provides an interface to create an index and search in it. The first step to create the index with custom documents is passing the documents to a JSONL with each line in the following format:

```
{
    "id": "doc1",
    "contents": "this is the contents."
}
```

So we will need to fetch all information of a document inside the *contents* field and set the *id* field to be equal to the *id* in the document. In this attempt what has been done is the fetching of only the *title* and the *text* fields, so the format of each document after processing was:

```
{
    "id": "docId",
    "contents": "docTitle\n docText\n"
}
```

With this done, the index can be created with the library as instructed in the documentation.
The next step is processing each query by searching in the constructed index. This is done using the **LuceneSearcher** class provided by the library. This class returns the best scoring documents

Author's address: João Antonio Oliveira Pedrosa, joao.pedrosa@dcc.ufmg.br, Universidade Federal de Minas Gerais, Belo Horizonte, Minas Gerais, Brasil.

according to BM25.
The CSV submitted to the attempt is a result of the 100 best ranking documents, according to BM25 and processed with the **LuceneSearcher**, for each query.

1.2 Second Attempt: Introducing Keywords and Processing Documents

Submission name in Kaggle: **17-55 (1).csv**
Average nDCG@100: **0.31155**

Since the previous attempt was just to check how well would the BM25 perform without any processing, in this attempt the idea is to make a simple pre-processing in the documents and the queries and, along with that, add the keywords field to the JSONL given to **Pyserini** when creating the index. The JSONL was constructed as follows:

```
{
    "id": "docId",
    "contents": "docTitle\n docText\n docKeyword1\n docKeyword2\n ..."
}
```

The pre-processing step in this attempt was as simple as it can be and consisted of simply uncasing all letters in the query and in the document so it was just a simple *string.lower()* command in *Python*.

1.3 Third Attemp: Stemming and Stop-Word removal

Submission name in Kaggle: **20-45.csv**
Average nDCG@100: **0.33406**

The idea for this submission is to increase pre-processing both in documents and in query. What was made is a combination of Stemming and Stop-Word removal. For this a function named *pre-processing* was implemented, it basically receives a string and pass it through two other functions: *Stemming* and *Stopword-Removal* before returning it.

1.3.1 Stemming. Stemming is made with the *NLTK* library using two classes, the *Porter Stemmer* and the *Word Tokenizer*. Each sentence is divided into tokens with the Word Tokenizer and the stemmed using the Porter Stemmer. A simple code is shown below:

```
def stemSentence(sentence):
    token_words = word_tokenize(sentence)
    stem_sentence=[]
    for word in token_words:
        stem_sentence.append(porter.stem(word))
    return " ".join(stem_sentence)
```

1.3.2 Stop Word Removal. Stop Word Removal is made with the *Spacy* library. The Stop Words set is retrieved from *en_core_web_lg*, a collection of features for the english language. After that, the word **"not"** is removed since it is important to keep it as it has a greater meaning then other stop words. With the set in hands it is possible to break the sentence again using the Word Tokenizer from *NLTK* and keep only the tokens that are different from stop words. A simple code is shown below:

```
def remove_stopwords(sentence):
    text_tokens = word_tokenize(sentence)
   tokens_without_sw = [word for word in text_tokens if not word in all_stopwords]

    return " ".join(tokens_without_sw)
```

1.3.3 *Pre-processing.*
With the pre-processing function defined, all fields of the corpus are pre-processed before assembling the dataset together in the JSONL for the Pyserini indexing function. With the index defined all processing to gather answers for the queries are made in the same way as in previous attempts.

1.4 Best Attempt: Leading Zeroes Correction

Submission name in Kaggle: **20-57.csv**
Average nDCG@100: **0.43569**

This section will not be counted towards the 5 different techniques described in this document. It is in fact just a disclaimer and need to be included since it is the best result amongst all.

In all previous submissions there was an error that was the cause of so low scores with techniques that should result in something remotely good. In the construction of the CSV that is submitted to the queries, the *QueryId* field should contain leading zeroes, i.e., all IDs should have three digits. In previous submission IDs such as "002" were being submitted as "2" so all *QueryId* from 2 to 98 were not being taken into account when calculating the score. With this correction, the best technique achieves **0.43569** in score and is a combination of **Stop Words Removal**, **Stemming** and ranking according to the **BM25** function.

1.5 Fourth Attempt: Tok2Vec Classifier

Submission name in Kaggle: **spacy_15-37.csv**
Average nDCG@100: **0.33896**

This submission uses a Neural Model to classify documents into two classes: *Partially Relevant* and *Highly Relevant*. The idea is that, after retrieving the 100 most relevant documents using the previous technique, the developed model could be used to classify documents into these two classes and then make a rank in which all documents classified as *Highly Relevant* would come first, ordered by the BM25 score, followed by all the *Partially Relevant* documents, ordered by this same score.

The model has been developed with *Spacy*, a library that provides a wide variety of Natural Language Processing tools, including text classification models. To develop the model that has been used in this attempt, a Spacy pipeline has been implemented with the *Tok2Vec* and *TextCat* components. Along with that, pre-trained word-embeddings from the *en_core_web_lg* collection have been used.

1.5.1 *Tok2Vec.*
This layer of the model is responsible for the Word Embeddings. In natural language processing, word embedding are a form of representation of words in the form of a real-valued vector that encodes the meaning of the word in a way that the words that are closer in the vector space are expected to be similar in meaning. With this, is expected that words in the document that have similar meaning as the words in queries could contribute to the result, something that is not implemented in the calculation of BM25. In the case of this model, this layers uses the pre-trained weights from the collection of english features that has been used.

1.5.2 TextCat. This layer is responsible for the categorization of each document. It calculates the weights that should result in a categorization that fits the training data.

1.5.3 Gathering Data. Since the corpus is to large, a pre-processing had to been done as the corpus could not fit entirely into memory. For that, we pre-processed each query in the training queries CSV, retrieving the text from the document and the text from the queries from the original datasets. A dataset containing all data necessary to training was assembled and stored into a new CSV.

1.5.4 Final Ranking. With the model implemented and trained, the 100 best documents gathered with the use of BM25 are classified into highly relevant or partially relevant. All the highly relevant documents are displayed first in reverse order of BM25 score. The partially relevant documents follow after that. An attempt using only the confidence of the model to classify documents as highly relevant to define their order of relevance has also been made but achieved a **0.25944** score.

1.6 Fifth Attempt: Bert Base Uncased

Submission name in Kaggle: **spacy-transformer_18-35.csv**
Average nDCG@100: **0.35529**

Following the same idea of the previous submission, another neural model was trained to classify the documents in the same way as the previous one. To develop the model that has been used in this attempt, a Spacy pipeline has been implemented with the *Transformer* and *TextCat* components. The Text Categorizer component is the same as in the previous model and the transformer model used was the *bert_base_uncased* from the Hugging Face repository. This model has a lot of training parameters (110M parameters in total) and took a long time and consumed a huge amount of memory to train. It was trained using a CPU, which is why it took so long (something between 6 and 8 hours) to train, and used 30GB of RAM during the training. It's results where not very good during training, achieving a bad F-score of 0.51 in the validation database. Due to the amount of time to train the model, no parameter adjustment has been made to improve the model but Spacy already tries to do it by itself. Classification of the documents from each query also took some time to be made. In the end, ranking has been assessed in the same way that in the previous method, displaying the Highly Relevant results first and the Partially Relevant after, ordered by BM25 score.

2 FAILED SOLUTIONS

This section will briefly describe some attempted solutions that the author were not able to fully implement.

2.1 Dense Retrieval with Pyserini

The Pyserini library also offers a function to build dense indexes and then perform dense retrieval. An attempt to build a Faiss Index using Pyserini has been made but due to problems with the dependencies of the model

2.2 Multi-field BM25 with Pyserini

Another function offered by Pyserini is to build a sparse index using more than one field instead of only using the field "contents". Again, due to problems with dependencies, this solution has not been fully implemented.