Kirill Semenov

Assignment 1 Report

System Design

The code is written in Python, so no additional building is required (However, it is necessary to download some files, which will be described later). A number of statistical and ML libraries was used, including numpy, scipy, nltk and ufal.morphodita.

The script is separated into modules, the majority of which corresponds to the steps of the data preprocessing. The crucial steps of the script are:

1. Creating the term-document matrix (`create_matrix_with_keys` function). This is done with the cycle: for each xml file with a set of documents, we extract the id of each document and the text contents of it (both – with help of regular expressions) and tokenize (+preprocess) them. Next, we use dictionaries to rapidly compute term frequencies and document frequencies. Next, we create the vectors of tf and idf values. Then, for each token in each document, we create a triplet of values (tf-idf value, (document id, term id)), based on which we create a sparse tf-idf matrix;
2. Reading the query document (`parse_query_doc` function) and creating the term-query matrix (`create_sparse_matrix_queries` function). Here, for each title, we preprocess it the same way as documents. Then, we strip queries from out-of-vocabulary words (i.e. those not met in document collection). After that, for each query, we create a sparse vector of the same dimension as the vocabulary of the document collection (we also use the idf values from the collection);
3. Computing the cosine similarity vectors for each query-doc pair (`create_cos_sim_vectors` function): We take each query vector from p.2, stack it horizontally so that we obtain the matrix of the same dimension as the tf-idf collection, and compute a cosine similarity of each row of these pairs of matrices (by summing over the document axis the pointwise multiplications of each term);
4. Ranking the documents (`rank_most_similar_documents` function): we sort the results in descending order and putting a maximum of 1000 most relevant values for each query;
5. Saving the best ranked results (`save_to_file` function).

For the sake of saving time, I used the numpy library to parallelize the matrix computations, namely:

- tf values (`compute_tf` function);
- idf values (`compute_idf` function);
- tf-idf matrix;
- query matrix (`create_sparse_matrix_queries` function);
- ranking cosine similarities (`rank_most_similar_documents ` function).

For the sake of saving space, I used the scipy library and different forms of sparse matrices there. They were used on the steps of creating the tf-idf matrices of documents and queries.

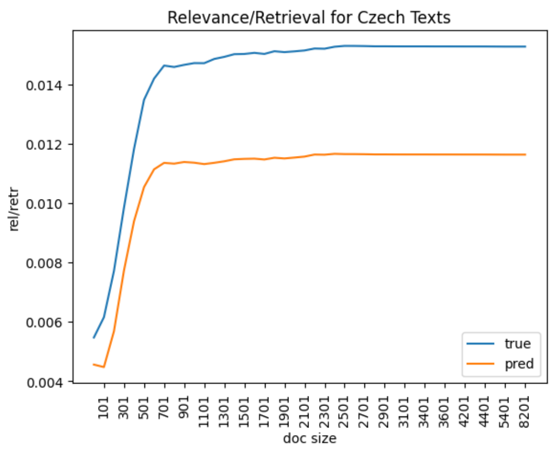
Experiments

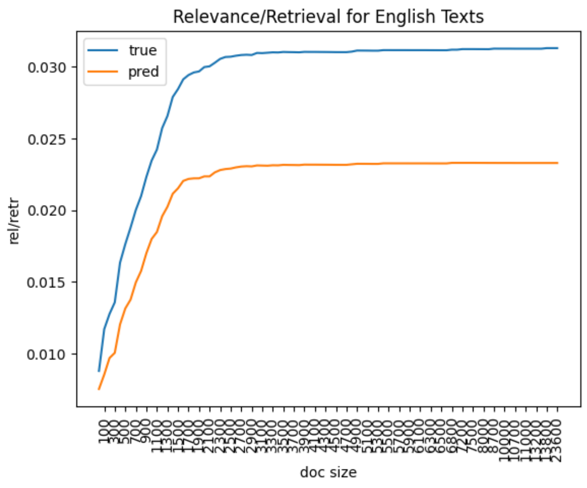
We compared the performance of the algorithm based on the following parameters:

1. tokenization algorithm (-tokenization parameter):
    a. default: the tokens are split by spaces and/or punctuation (i.e. punctuation is omitted);
    b. nltk: using the nltk library for tokenization;

   c. morphodita: using the morphodita library for tokenization + lemmatization (to run, installing the taggers is needed: https://ufal.mff.cuni.cz/morphodita#language_models);
   d. clear_stopwords: using nltk for tokenization + clearing the documents from stopwords (English stopwords are retrieved from the libraries, and Czech stopword set is taken from here: https://github.com/stopwords-iso/stopwords-cs)
   e. nltk_clear_punct: nltk for tokenization + clearing punctuation (because the initial tokenization excludes punctuation and is fairly good)
   f. nltk_clear_punct_case_insensitive: nltk for tokenization + clearing punctuation + case-insensitive
2. preprocessing algorithm (-preprocessing parameter):
   a. default: clearing from html tags;
   b. case_insensitive: switching all symbols to lower case.
3. TF weighting (-tf parameter):
   a. default: logarithmic
   b. boolean;
   c. augmented.
4. IDF weighting (-idf parameter):
   a. default;
   b. none;
   c. prob.
5. Query construction (-query_construction parameter) – only for run-2:
   a. default: only title tag contents
   b. enhanced: title + desc tag contents
   c. enhanced_2: title + desc + narr tag contents

I also tried experimenting on pivot normalization: I computed the relevant/retrieved ratios for the best run (run-2) outputs and the training data; however, the ground truth data were always better than the run-2 outputs. The observations of that are demonstrated below, and the calculations are provided in the `pivot_normalization.ipynb` file.





Results

Below I present results of my ablations; I present the main MAP metric. In bold, the best algorithm variants for each run and their scores are marked.

run	filename prefix	explanation	map	
			en	cs
run-0		**all default values**	**0.0781**	**0.1459**
	case_insensitive	preprocessing: case-insensitive	0.1714	0.1879
	nltk	tokenization: nltk	0.1029	0.1461
run-1	**nltk_case_insensitive**	**tokenization: nltk; preprocessing: case-insensitive**	**0.1984**	**0.1882**
	morphodita_case_insensitive	tokenization(+lemmatization): morphodita; preprocessing: case_insensitive	0.0859	0.0518
	morphodita	tokenization(+lemmatization): morphodita	0.0456	0.0761
	clear_stopwords	tokenization+preprocessing: clear_stopwords	0.1027	0.1462
	nltk_clear_punct	tokenization: nltk; preprocessing: clear punctuation	0.1028	0.1459
	nltk_clear_punct_case_insensitive	tokenization: nltk; preprocessing: clear punctuation, case_insensitive	0.1714	0.1879
	idf_none	idf weighting: none	0.0689	0.124
	idf_prob	idf weighting: prob	0.0782	0.1459
	tf_bool	tf weighting: bool	0.0627	0.0962
	tf_augmented	tf weighting: augmented	0.0628	0.0968
run-2	**enhanced**	**query construction: enhanced; tokenization: nltk; preprocessing: case-insensitive**	0.2641	**0.229**
	enhanced_2	query construction: enhanced_2; tokenization: nltk; preprocessing: case-insensitive	**0.2848**	0.2262

From the table above, we can conclude the following:

1. For the tokenization and other preprocessing steps, the big improvements are obtained by lowercasing the words and tokenizing the texts with nltk (the combination of these two parameters gives the best results for the run-1). Notably, lemmatizing texts with morphodita decreases the performance. Removing of stop-words has some positive effect, but it is significant only for the English data. Notably, while applying case insensitivity, the quality on English data jumps by 10% while only by 4% on Czech data. I can hypothesize that this can happen because the number of unique word forms in Czech is already big and its reduction to lowercase does not influence the vocabulary. The similar thing happens with tokenization: for English, the increase of performance is more significant. I can suppose that this is due to more correct segmentation of the English possessives (‘s) and other combined forms such as “aren’t” etc.
2. For IDF weighting, the probabilistic weighting gives approximately same results. However, the if we look at the number of lines in the resulting documents is lower. This can mean that this algorithm is better at grouping more relevant results in the upper part of the distribution (however, this part is still big and equals to 1000).
3. For TF weighting, the augmented weighting gives a drop in performance for both languages.

4. Using more information from the queries gives a new significant increase in the quality; however, the better results (at least for Czech) are observed only if the ‘desc’ part is included. Maybe this happens because the narratives are too wordy and they start misguiding the cosine similarity algorithms.