Kirill Semenov

Report on Assignment 2

For the second assignment, I used the Terrier IR Platform and, specifically, the PyTerrier package as a Python wrapper for it. Although PyTerrier provides a range of preprocessing functions and allows to work directly with the files of TREC format, some of its parameters (such as case preservation or different variants of tokenization) are easier to control beforehand. Thus, the general organization of the algorithm is the following:

1. Collection preprocessing. At this step, all necessary files are transformed into pandas dataframes. During this step, various preprocessing parameters can be tweaked, namely: case preservation, tokenization, lemmatization and stop word removal.
2. Indexing. Fully implemented with the help of PyTerrier. At this step, the indexing functions can be tweaked (for instance, TF, TF-IDF, BM25 or language models).
3. Query preprocessing. Implemented outside PyTerrier (by the same functions and with the same parameters as at step 1). At this step, we can enhance the query construction (not only by titles, but also by descriptions).
4. Retrieval process. Implemented with PyTerrier with the dataframes of the index and the queries. At this step, the query expansion can be applied.
5. Saving the results to the .res file.

For steps 1, 3 and 5, the main part of the code was used from the Assignment #1.

Run 0

I found the preprocessing steps of PyTerrier indexer quite uneasy to specify; thus, the tokenization and case preservation steps were made by me. The trickiest part was case preservation, as by default PyTerrier is case insensitive. I solved it by adding the alphabetic prefixes “TITLE” to the words starting with capital letter, and “UPPER” for the words consisting of only capital letters. I could not use any specific symbols, as PyTerrier tokenization tends to delete all non-alphanumeric symbols.

The performance of the algorithm on English data was 0.2678, and 0.2056 on the Czech data, which is similar to the results of run-1 from my first assignment.

Experiments

I have tried to tweak several parameters. The general table with result is represented below.

Experiment group	**Run/experiment**	**MAP (EN)**	**MAP (CS)**
	Run-0	0.2678	0.2056
Preprocessing: casing	*Case-insensitive*	*0.3602*	*0.2542*
	Case-sensitive + bm25	0.1496	0.1600
Preprocessing: tokenization	Morphodita	0.1047	0.0728
	Nltk	0.3602	0.2542
	Clear-stopwords	0.3588	0.2518
Indexing	BM25	0.3497	0.2517
	TF	0.1234	0.0605
	PL2	0.3355	0.2405
	Dirichlet LM	0.3498	0.2654
	Run 1: LGD	**0.3887**	**0.2735**
	NLTK+LGD	**0.3887**	**0.2735**
	Hiemstra LM	0.3018	0.2339
Query expansion	Tf-idf + query expansion	0.3669	0.3102

	Bm25 + query expansion	0.3613	0.3084
	Run 2: LGD + query expansion	**0.3686**	**0.3111**
	DPH + query expansion	0.3626	0.3033
Enhanced query construction	Lgd+ enhanced	0.3084	0.2488
	Bm25 + enhanced	0.2945	0.2488

Table 1: MAP values of different ablations on the train data. The scores in bold are the best results; the scores in italics are the main "breaking point" compared to Run-0.
The algorithms highlighted green are Run-1 and Run-2.

Case sensitivity

The first experiment was to relax the case preservation and use the default configuration of PyTerrier (which does not take casing into account). This only action increased the performance in English by 10 percent and in Czech by 5 per cent. Interestingly, the Czech outputs were less numerous (approx. 21,000 documents instead of almost 25,000 documents with case preservation), which suggests that case insensitivity can be beneficial for Czech with respect to the efficiency of the algorithm.

As this comparison (and several other analogous comparisons, for instance, with BM25 indexing parameter) showed the importance of case insensitivity, hereinafter all considered variants by default would be case insensitive.

Tokenization

Similar to the 1st Assignment, I have tried NLTK package for tokenization, as well as morphodita for lemmatization and NLTK + (list-based) stopword removal. All these algorithms showed comparably similar quality to the basic (case-insensitive) algorithm; the only exception was morphodita, which showed a poor performance. I can hypothesize that this might be because of the abundant additional tags that morphodita is uses to disambiguate lemmas (and which are usually written with non-alphabetic values). I tried to compress them only to alphabetic symbols, but this may not have been enough. Notably, NLTK tokenization, although being more profound than the default whitespace-based PyTerrier default tokenization, showed exactly the same result as the default one, but was working three times longer (approx. 10 min compared to 3-4 minutes with the pyterrier "naïve" tokenization). Thus, we decided not to test NLTK package for the sake of time optimization.

Indexing

PyTerrier provides a convenient interface to test various formulas and approaches for indexing. Thus, I have tried even more options than I could implement in Assignment 1. The tested models are:

1. TF – basic term frequency;
2. TF-IDF – with basic parameters;
3. PL2 – one of the Divergence-from-randomness models – family of models that try to rank terms depending on how different their distributions in the collection is from the random distribution;
4. Dirichlet LM – smoothing model based on the collection language model: assigns a negative score to documents that contain the term, but with fewer occurrences than predicted by the collection language model;
5. LGD – combined smoothing based on Divergence-from-randomness and Language modelling approaches;
6. Hiemstra_LM – another language model specified for IR task;
7. DPH – was used only together with query expansion (as recommended by the Terrier developers).

The worst performance (twice as worse as run-0) was demonstrated by the TF model. The majority of other models showed a comparable quality as the case-insensitive version of run-0 (usually – slightly worse). The only notable outlier (+2% to the case-insensitive baseline) is the LGD model. This observation shows us that combining the language modelling and the probabilistic approaches to the task of term-document weighting is a rewarding activity.

As it would be seen later, the record of the LGD performance would not be beaten. Thus, run-1 is the LGD weighting model with case insensitivity, otherwise with the same parameters as run-0.

Query Construction

In this homework, I tried expanding queries only by description (<desc> tag). I used it with several indexing models, and none of them surpassed the corresponding results without query construction. This is a surprising behavior for me, because in the previous homework it did help significantly.

Query Expansion

Finally, I tested the query expansion functionality in the PyTerrier module. I tried the Bo1 algorithm. The results showed significant improvements for the Czech data, while relatively similar performance in English. This may be interpreted as follows: on the data that we have (and given no additional resources such as user logs or thesauri), the main "expansion" happens with finding different forms of the same lemmas, or different grammatical variants of the same idioms, but not the "real" synonyms or other similar words. Thus, it is more crucial to have such system for Czech than for English.

Among all variants of indexing algorithms together with query expansion, LGD again showed its best performance. Thus, this combination was used for the parameters of run-2.

Results and Limitations

In general, the main improvements were brought by case insensitivity (which is, again, a default parameter for Terrier already), choosing an appropriate weighting model (ideally – not the vector space one), and query expansion. Thus, Run-1 is based on case insensitivity and LGD, and run-2 is based on run-1 plus query expansion.

However, I have not tried all features of Terrier. Among these options, two important ones were left: tweaking the parameters and models for query expansion (including some extra sources such as thesauri), and the neural IR algorithms.