Report Group 05

Exercise 1

WS 2021 - 188.977 Grundlagen des Information Retrieval

Tokenization

Our tokenization is realized as a pipeline that consists of multiple *Python generators*. Each generator modifies the input string and yields the result to the next processing step. This architecture allows us to easily extend the pipeline by simply adding new generators and thus makes the code better maintainable and understandable.

For the parser we used the class `HTMLParser`, which provides a skeleton for an event-based HTML parser. Using the parser, we read a document line by line and parse the items contained in it into named tuples. At this point we have also already made some optimizations: every callback to `handle_data()` appends its data to a list and not a string, which turned out to be notably faster.

To check whether the generators behave like intended in a reproduceable fashion we use Python unit tests and several test cases which showcase the behaviour. The testcases can be found in the `TestTextPreProcessing()` class.

In our implementation we have several steps to obtain tokens from the input text which are described in the following:

1. `split_at_whitespaces()`: This generator splits the input string at every whitespace to separate the individual terms for further processing.
2. `remove_apostrophes()`: This generator removes every character after an apostrophe (one of the defined Unicode representations in APOSTROPHES_REGEX) is encountered including the apostrophe itself. This is done by taking the first index of a `re.split()` call. Example testcases can be found in `test_remove_apostrophes()`.
3. `remove_hyphen()`: This generator breaks up hyphenated sequences into several words using again `re.split()`. To catch every Unicode representation of hyphens we use the HYPHEN_REGEX which includes several different dash symbols like haypehn, ndash, mdash, horbar and more. Again, testcases are available in `test_remove_hyphen()`.
4. `remove_punctuation()`: This generator removes all punctation from strings such as “:.,()” but not from numbers to preserve decimal numbers. To achieve this, we use the `PUNCTUATION_REGEX` with negative lookbehind assertions. Several testcases can be found in `test_remove_punctuation()`.

After these tokenization process we lowercase the resulting tokens using our `lowercase()` method.

Inverted Index

Our inverted index consists of two Python dictionaries: *tokens* and *articles*:

- Tokens stores a mapping of a single token (as `str`) to an array of postings (as `array.array`). Postings is a one-dimensional array containing the article (as `int`)[1] and the term frequency (as `int`) of the tokens in the article’s `<title>` and `<bdy>` entities. A complete example tokens may look like:
  `{‘apple’: [808, 1, 389, 15, …], …}`.

[1] the textContent of the `<id>` entity prepended by `<title>`

- Articles stores a mapping of articles (as `int`)[2] to an one-dimensional array (as `array.array`) containing the document (as `int`)[3] where the article is contained as well as the count of tokens of the corresponding `<title>` and `<bdy>` entities. A complete example may look like:
  `{808: [1, 43432], …}`.

To store the posting, we originally used an ordinary NumPy array. Since NumPy's `.append()` does not occur in-place, but a new array is allocated, and the contents copied, this turned out to be very inefficient and thus slowing down the indexing process. Next, we implemented a dynamic array using `ndarray.resize` and over-allocated the underlying array with a growth factor of 2. This was already fast, but we found out, that Python's native `array.array` allows efficient appending out-of-the-box, though using the same technique essentially. Comparing these two options, we noticed similar performance, but since `array.array` required less memory (especially when dumping the index) and no dependencies are needed, we decided to use `array.array`. As type code we use `I` which stores values using the C Type `unsigned int`.

For dumping and loading the index we use `pickle.dump()` and `pickle.load()`.

Stemming

For stemming we use the NLTK SnowballStemmer for English to stem the tokens we receive from our tokenization pipeline. Again, we designed our `stem()` function as a generator which allows us to seamlessly integrate it in our preprocessing pipeline. To speed up the stemming, we use a LRU (Least Recently Used) Cache to cache the calls to the NLTK Stemmer. Since it is a LRU cache, especially the tokens that occur more frequently in the documents are kept in the cache and speed up the stemming dramatically. On our test system we archived 340.02 articles/s without the cache and 791.06 articles/s with the cache.

For the implementation part we used the LRU Cache from python functools which resides in the Python standard library and therefore should be allowed to use in this exercise. By simply decorating our `lookup_stem()` function with `@lru_cache()` we can cache the most recent calls to the function. How we determined a suitable size for the cache is explained in the Optional Section.

For stop word removal we use the NLTK Stopword List and discard every token that occurs in this list with our `remove_stop_words()` function. To also speed up this process we converted the `stop_words` list into a set to have lookup time in O(1) instead of O(n). This allowed us to achieve another impressive improvement from 791.06 articles/second to 1036.21 articles/second.

We also provide Unit Tests for these steps of the pipeline in `test_stemming()` and `test_remove_stop_words()` respectively and for the whole pipeline in `test_text2tokens()`.

Scoring

The scoring methods were implemented as introduced by in the lecture, except for the IDF in the BM25. Here we used the following formula from Wikipedia, which yielded better results:

$$\mathrm{IDF}(q_i) = \ln\left(\frac{N - n(q_i) + 0.5}{n(q_i) + 0.5} + 1\right)$$

[2] the textContent of the `<id>` entity prepended by `<title>`, e. g., 808 for `<title>Google</title><id>808</id>`
[3] the number in its filename, e. g., 41 for `41.xml`

The parameters for BM25 were set to k1=1.25, b=0.75. The calculations are done with the help of NumPy arrays. The scores are computed for every query term, and we implemented two ways to calculate the total score. First, we just summed all the scores and used this to rank the documents secondly the cosine measure can also be used as similarity measure between the query and the document. However, the sum yielded better results therefore we used this method for the evaluation.

Evaluation Results

	mAP	NDCG@10	P@10	P@10
TFIDF	0.1365	0.4837	0.4385	0.4385
BM25	0.2451	0.5780	0.5404	0.5404

The performance tests were conducted on an Intel Core i7-6700K (4 Cores / Threads), CPython 3.8 and Ubuntu 20.04.3 LTS.

- Run time for indexing:
  - No Cache: 0:13:48.730239 (~14 min.); 340.02 articles/s
  - Cache: 0:05:56.216803 (~ 6 min.); 791.06 articles/s
  - Cache and Stopwords as `set`: 0:04:31.943855 (~ 4 min.); 1036.21 articles/s
- Run time for querying:
  - Average query time (over `eval.qrels` queries):
    - BM25: 0:00:00.467624 s
    - TF-IDF: 0:00:00.458287 s
- Number of documents indexed: 281,782
- Peak allocated RAM:
  - Indexing process: ~1.7 GB RAM (w/o pickle)
  - Indexing process: ~3 GB RAM (w/ pickle, for a couple of seconds)
  - Tokenizing processes: ~100 MB per Core

Discussion of Results

As expected, the BM25 scoring method yielded much more relevant results then the TF-IDF and this also can be seen when using the exploration mode. The results returned using the BM25 feel much more relevant to the topic. However, the TF-IDF method is slightly faster than BM25 due to its simpler formula to calculate the score.

How to run your Prototype

1. Unpack the submission file `Group 05_gir.zip` and the dataset file `GIR2021 dataset.zip`:

```
unzip Group\ 05_gir.zip
cd Group\ 05_gir
unzip GIR2021\ dataset.zip -d dataset
```

2. Create a virtual environment, install the needed dependencies, and download stop words list:

```
python3 -m venv venv
. venv/bin/activate
pip3 install -r code/requirements.txt
python3 -m nltk.downloader stopwords
```

3. Change to the code directory (mandatory since we use relative paths!) and execute `createindex.py` to create an `index.obj` file at `../index.obj`.

```
cd code
python3 createindex.py # dumps index at ../index.obj
```

4. For evaluation and exploration execute the corresponding files.

```
python3 evaluation_mode.py # use -h for help, requires index at
../index.obj;
python3 exploration_mode.py # requires index at ../index.obj
```

Optional Section

To **speed up the indexing process** we implemented multi-processing, thus we use all available CPU cores on a multi-core system while populating the index. We have organized this in such a way that there are a variable number of processes[4] that read in new documents and tokenizes them, and a single process managing the index itself and adding the tokens to the index. We've chosen this structure because (a) adding tokens to the index is much faster than tokenizing documents and (b) the performance of sending and receiving tokenized articles via pipes (by using e. g. `ProcessExecutorPool()`) is much faster than managing a dictionary in shared memory (by using e. g. `Manager.dict()`).

Finally, to benchmark our implementation easily and estimate the remaining time until the indexing process is finished, we included a "progress bar":

```
Indexing: 281782/281782 (1036.21 articles/s) [Estimated remaining time:
0:00:00]
Total indexing time: 0:04:31.943855
Bytes written: 954688498
```

To speed up the stemming process we use the Python functools LRU Cache as described in the Stemming section and determined the cache size by examining the used amount of RAM per Core and the throughput in articles/second. Since we want to build a scalable system, we restricted the RAM consumption per core to a maximum of 100MB. We figured out that a cache size of 150,000 entries allows us to archive a throughput nearly as good as if we used an unbounded cache size while at the same time only consuming ~100 MB of RAM/core. We therefore set the cache size to 150,000 entries.

[4] per default this value is equal to the number of available CPU cores