TTDS CW1 Report
Nik Peleshatyi - s2150635

Tokenization and Stemming

I have split and streamlined text preprocessing into a pipeline that consists of 3 functions that are applied sequentially:

1. `tokenize_text` takes in a string of text and using regex replaces any non-alphanumeric (except underscores and hyphens) characters into newline chars. Empty string entries or None's are `filter`'ed out, abd the list is returned. Tokenizationis controlled using a predefined `regex`, which can be modified without affecting the rest of the code.
2. `remove_stopwords` takes in a list of tokenized (list of strings) text filters out the ones that are in the stopword list, which is loaded from the file when the script first starts. Done using `filter` and cast back to `list`. For efficiency, the list is loaded into a `set`, which allows much faster lookups through hashing, rather than going over the **entire** list every time.
3. `normalize` takes in a list of text tokens and applies the Porter Stemmer to each item. Done using `map` and cast back to `list`.

Inverted Index

Implemented as a class, which has methods that allow indexing, loading from a file, saving to a file, as well as performing several types of searches over the index. The class defines variables, such as parsing regexes, and text processing sequence that govern how those functions are executed and allow modifications without modifying the methods directly.

1. Indexing

Loading and indexing process is decoupled and split into stages: load and parse document collection, preprocess the document text, and for each term of a document, add a record of term's position. For that reason, I decided to implement the index as a dictionary (hashmap) of dictionaries of lists. The top level dict is for terms, the next is document IDs and the values are positions of occurrence. `{ term: { doc_id: [1, 4, 6]  } }`. This is efficient as dict lookups are constant time, regardless of the collection size.

`load_and_index_collection` - given an XML file, it is parsed by a built-in python XML library, then, every child `<DOC>` of the tree is converted to an instance of a `Document`, which essentiallay converts an XML object into an object, where items, such as DOCNO, TEXT and HEADLINE are accessible as attributes. Once loaded, the headline is combined with text and the document is sent for processing and adding to the index. All document ids are recorded into a set for ease of negation operations.

`add_document_to_index` - sends the text through the processing pipeline achieving a list of normalized, free of stopwords, tokens that can be used as keys for the index map. Enumerated iterator provides position, which is added to the `{term : {doc_id : [..., pos]}}` list.

2. Serialization

To allow faster initialisation, the index provides import and export methods to avoid indexing the collection every time.

`save_to_file` - given a file to write into, goes over every term in the `terms index` and writes a line that lists a term, document ids and comma-separated positions on new lines. At the end adds a **newline** to differentiate from next record. Writes one term at a time to avoid creating a huge string and using memory.

`load_from_file` - the inverse process of saving, initialises from the file, reads the file line by line to avoid loading the entire thing in memory at once. If newline is encountered, the term is reset and the next line will be the term, and all the following lines will be document ids with positions, which are extracted using regex and added to the index.

3. Search functionality

Based on the coursework requirements, the system works on the assumption that no more than one AND/OR will be present in a single query, this greatly reduces the complexity.

The query processing has 2 stages: parsing and evaluation. Parsing in `search` takes care of AND/OR operators and their effect on the final result implementing the backbone of the **Boolean search**. The evaluation handles the subqueries, including NOT processing.

`evaluate_expression` checks for negation and based on regex for available operations (proximity search, exact and nonexact **Phrase Search**), performs the search. I implemented `phrase_search` as a **particular case of proximity search with distance 1** for each pair of adjacent query tokens (after processing) and by applying set union or intersection to the intermediate results. I also

accounted for edge cases, where there are no tokens after preprocessing - returns empty set, and if there's 1 token, performs `term_search` - returns the lookup of the index map for that token. Exact and nonexact are differentiated by the presence of `""` marks, regex is used to extract content.

Explicit proximity search uses regex to extract the distance and the terms to perform the search.

Proximity search - given 2 terms and a distance N, goes over the documents, in which both terms are present and returns those that match the constraints. `check_terms_close_in_document` provides a nice abstraction of the logic so it can be called for selection (True/False) on each document.

The actual check logic involves 2 pointers and a doubly nested loop:

- The top loop (j) goes over the 2nd list while within list 2 bounds
- The inner loop (i) keeps going over the 1st list while **ALL** apply:

1. Within list 1 bounds.
2. Value of list 1 at the pointer `i` is <= than that of list 2 at `j`
3. Difference of values at respective pointers is > N.

- At each top loop (j) iteration, check if the distance between values at current pointers is <= N, if so, return `True` as the terms are close.
- If the bounds were reached, return `False`.

Ranked Retrieval (TFIDF-based) - using the class pipeline, preprocesses the query into a list of normalized tokens. Then collects a union of all documents, where at least one of the query tokens appeared. For each document a tf-idf score (as shown in the lecture) is computed over ALL documents present in the index. In case the token does not appear in a document, set the weigth to 0 to avoid math errors when computing a log. (doc_id, score) tuple pairs are combined into a list and returned.

What I learned

- Tokenization and stemming are very important for NLP in search engines.
- Efficient implementation can bring execution time from minutes to subseconds.
- Modular and reusable code makes development faster and consistent.
- Version controlling tools allow for resilient experimentation and work between different devices.

Challenges I faced

My first implementation was taking around 4-5 minutes to run indexing and and searching on 10 queries, which was frustratingly slow. I took to debugging and learned that stopword removal and normalization take a long time. Stopword removal, by using a `set`, reduced runtime from 30 seconds to <1. Stemming initally used the NLTK library, which was slow so I switched to PyStemmer, which was written in C, but my computer had a version mismatch, which took a long time to fix and eventually I just worked on DICE machines. The low-level library brought down execution from 2.5 minutes to ~3 seconds.

How to improve and scale

- The search function should implement a Shunting Yard algorithm or another parsing method or a state machine to allow processing arbitrarily complex Boolean queries.
- Instead of loading the entire collection of documents at once, it could be processed line-by-line, but that would require implementing a more complex parser, which would be similar to what I did for loading the serialized index.
- Serialized index could be in a database, to allow for efficient segmentation since DBMS are built with indexing optimizations.
- Implement Delta Encoding for position arrays, which would require less memory for large differences in positions.
- Implementing compute-intensive tasks, such as indexing and processing in a low-level language like C or C++ to make it faster and more memory efficient.