Text Technologies for Data Science

Qais Patankar - s1620208 - 20th October 2019

There were several tasks involved in building this simple IR tool:
- Preprocessing of text (tokenisation, stopping, and stemming)
- Building a loadable index
- Parsing queries
- Performing searches

Preprocessing

The preprocessing function is defined as `tokenize`, but also takes care of removing stopwords, and stemming tokens.

Tokenization

Initially, this code was used to case fold, tokenize, and strip all punctuation from text:

```
toks = text.lower().translate(str.maketrans('', '', string.punctuation)).split()
```

If we deconstruct that line, the code performs the following operations:
1. converts the entire text to lowercase
2. replaces punctuation with the empty string
3. split on sequences of whitespace characters

The second item there proved troublesome with the following test search query:

```
#1(san, francisco)
```

A document containing “San Francisco-based” was not included in the query, and this was because (pre-stem) the above quote would be tokenized to [“san”, “franciscobased”] instead of [“san”, “francisco”, “based”].

This doesn’t make sense as “francisco” and “based” are clearly two distinct words that should be treated separately. This was resolved by adjusting the second step to replace punctuation with spaces instead:

```
spaces = len(string.punctuation) * " "
toks = text.lower().translate(str.maketrans(string.punctuation, spaces)).split()
```

Once the overall task matured, it was decided that **only letters** should be retained — no numbers or symbols. This resulted in the final approach of mapping every letter to a space:

```
toks = "".join(map(non_alpha_to_space, text.lower())).split
```

This handles situations like “Francisco-based” well, but results in “Adam’s Apple” tokenizing to [“adam”, “s”, “apple”] (pre-stem). The addition of the extra “s” should not affect search results in any major way, so a special case to handle this was not introduced.

Stripping numbers has the benefit of returning documents containing “BBC2” when searching for “BBC”. Unfortunately, this also means that searching “BBC2” will also return results containing “BBC3”.

Stopping

A utility function `get_file_lines` was defined to read in all lines in a given file. This was used with englishST.txt to read all the stopwords into a simple `stopwords` list. This is passed to the preprocessing function as a `filter_set`.

Stopping is achieved by a simple filter with a lambda function:

```
toks = filter(lambda t: t not in filter_set, toks)
```

Stemming

To stem words the **stemming** Python library was used — this library is not preinstalled on DICE machines. It was initially difficult to use this library as its latest version (1.0.1) does not include any documentation, only **stemming 1.0**.

Many documents in the trec.sample.xml sample collection contained duplicate words, and stemming words is an expensive process, so it was clear that memoisation would be an effective optimisation technique. To achieve this, the `stemming.porter2.stem` library function was wrapped with a `memoized_stem` function, backed by `functool’s` `lru_cache` annotation. Various LRU cache sizes were tested, and a cache size of 4096 was settled upon once satisfactory performance was achieved.

```
toks = map(memoized_stem, toks)
```

Finally, the tokenize function returns `list(toks)` to ensure that the generator returned by `map` above is consumed fully into a `list`, allowing it to be reused multiple times.

Building a loadable index

The tool initially checks to see if there is an existing file with the following format: `$collection_filename.index` (i.e. the collection filename with suffix “dot index”). If the file exists, the file is read and the `docmap` and `index` is loaded using `pickle.load`, otherwise the collection file is processed.

The collection file (in XML format) is parsed using the Python standard library and translated into a list of **`Doc`** instances. Each instance contains fields for the document number, the headline, the text content and the list of tokens.

To facilitate quick accessing of documents, a `docmap` is generated. This is a trivial task and is the initial representation of the input files in memory (other than the initial XML string).

The `docmap` is a `dict` mapping document numbers to the corresponding instance of `Doc`.

`map[docnum]Doc`

Using the `docmap`, an `index` is generated. This is a mapping from each term to a `docposmap`. Each `docposmap` is a `dict` that maps document numbers to a list of positions.

This is done by iterating through each **`doc`** and performing the following computations:
- loop through each **`token`** in **`doc.tokens`**, adding their position to a list
- loop through each list of positions (for each **`token`** seen in this **`doc`**), and assign **`index[token][doc.num]`** the list of positions.

If the type of `index` is expanded a little, it looks like this:

`map[term](map[docnum]positions])`

(where `positions` is a simple list of integers)

This leaves the application with two key objects: `docmap` and `index`. Note that throughout this document and the code itself the terms "**term**" and "**token**" are used interchangeably.

These two objects are dumped to the `$collection_filename.index` as a tuple `(docmap, index)` using `pickle.dump`. Additionally, the index is converted to string representation and written to `$collection_filename.index.txt`.

In the `index.txt` file, keys are sorted using `safe_int` as the "key" for Python's `sorted` function. The `safe_int(`**`x`**`)` function is defined as returning `int(`**`x`**`)` where possible, and **x** otherwise. This is so that lists like `[“200”, “300”, “25”]` are correctly sorted to `[“25”, “200”, “300”]` instead of `[“200”, “25”, “300”]`, without crashing for non-integer entries.

This `safe_int` function is used later on, for other parts of the tool.

Parsing queries

The tool supports two logical binary operators: **`AND`** and **`OR`**. Throughout this section "spaced operator" is defined as an operator surrounded by a space character on both sides.

For each operator, until a spaced operator exists inside the query string, we split by the spaced operator.

This results in “San AND Francisco” splitting into `parts=[“San”, “Francisco”]`. This misbehaves if a string query contains tab characters or other forms of whitespace around the operator. *Operands* is a better word to describe that list than `parts`.

If the query string does not contain any operator, the operator is assumed to be **`OR`**. If queries are to be ranked (i.e. from queries.ranked.txt), the query string is split by space and assigned to *parts*.

Otherwise *parts* is assigned a list containing just the query string. This is ensures that phrasal or proximity (with space after comma) queries aren’t prematurely split.

For each operand, whitespace is stripped from both sides. Negation is determined by the existence of a “NOT” prefix, and removed from the operand text accordingly.

Phrase searches are then determined by the double quotes on both sides, followed by the quotes being stripped. The resultant operand text is split by sequences of whitespace. If the phrase contains just one word, then it is treated as a regular single-word query. If it contains two words, the operand “text” is assigned a tuple of both words. Otherwise an error is raised, as a phrase should only contain exactly 1 or 2 words.

If the operand is not a phrase (determined by the initial double quote check), a regular expression is used to check for a proximity match, with the `distance` plucked out, and the two words involved assigned to the operand “text” as a tuple.

If the operand “text” is a tuple, both words are preprocessed. Otherwise the single word is checked to be a stopword. If it’s a stopword, it’s ignored, otherwise it’s preprocessed.

Finally, the operand (with its metadata: negated, quoted, distance) is added to a list of operands, like this:

```
new_parts.append(QueryOperand(s, negated, quoted, distance))
```

Once all operands have been parsed, a tuple (operator, operands) is returned:

```
return (chosen_op, new_parts)
```

What if a query string contains stopwords?

In this case the stopword will just be stripped, resulting in the query “francisco AND and” being computed as almost the same query as “francisco”, shown below:

```
➜ python app.py trec.5000.xml ‘francisco AND and’ --debug
26 documents, query: ('1', ('AND', [QueryOperand(francisco)]))
```

```
➜ python app.py trec.5000.xml ‘francisco’ --debug
26 documents, query: ('1', ('OR', [QueryOperand(francisco)]))
```

Hence, both query strings return precisely the same results.

If the query is “francisco AND NOT and”, the latter half of the query is stripped:

```
➜ python app.py trec.5000.xml ‘francisco AND NOT and’ --debug
26 documents, query: ('1', ('AND', [QueryOperand(francisco)]))
```

However, the tool returns an (unhelpful) error if a *phrase* contains a stopword:

```
➜ python app.py trec.5000.xml '"francisco and"' --debug
queries.QueryError: Odd s 'and', got toks: []
```

It’s good to return an error in this case as it would be disingenuous to return matches where the phrase did not occur. Potential improvements to the tool in the future would also check unstemmed versions of words to ensure a phrasal match.

Performing searches

Searches are performed given operator and list of QueryOperands.

First inclusion and exclusion sets are produced

For each operand we check if the operand is phrasal or proximity based. If so, the operand “text” is a tuple. We continue with this operand if both terms in the tuple are in the index. A set of common documents for each term is produced and iterated, and for each common document:

- A pair of positions is produced for each possible combination of positions between two terms
- For each pair, if phrasal: check that the pos_term_b minus pos_term_a is equal to 1
- For each pair, if proximity, check that the absolute difference between positions is equal to the difference defined in the operand.
- Add each matching document to a *docs* list (for this operand)

If not phrasal or proximity based, we check if the operand text is in the index. If so, the *docs* list is assigned to the documents in the index for that term.

If neither of the above, this operand is skipped. If not skipped, we include *docs* in the inclusion set (or exclusion set, if operand is negated).

Handling the inclusion and exclusion sets

If ranked queries are performed, the usual algorithm is applied. Otherwise, special cases are needed for plain NOT queries as well as AND queries.

For plain NOT queries, the inclusion set is initially empty, but exclusions are not. The inclusion set should in fact contain the entire document set. This is checked and accounted for.

Then, if the operator is an AND, all the inclusion sets are reduced to a single inclusion set using *set.intersection*.

Finally, the returned set is inclusions minus exclusions.

Future fixes here would be to ensure that singular ANDs work well with not:

```
➜ python app.py trec.5000.xml 'NOT francisco AND and' --debug
0 documents, query: ('1', ('AND', [QueryOperand(NOT francisco)]))
```

It works fine if the query is just ‘NOT francisco’.