CW1: Document indexing – by Vladislav Yotkov

Task 1: Inverted index

Index term choices

The system I have implemented supports normalized & stemmed:

- **Unigrams** – single words which are treated as the smallest morphological units (after pre-processing). Trivially, every n-gram is building on top of unigrams, hence they are the baseline of expressions and are necessary for indexing. E.g., '*Elementary'* becomes '*elementari*'.
- **Multi-word expressions (MWE)** – these are the combinations of words provided in the knowledge base (*location & character spreadsheets*), which I assume the proficient user would already know. Under this assumption, they could **easily access documents containing the exact entity of interest**. E.g., '*Springfield Elementary School*' becoming '*springfield_elementari_school'* (pre-processing is applied to each word in a WME).
- **Numbers** – these would be anything but currency expressions because TV shows usually refer to specific time periods (e.g., years) or episodes.

Critique of the system:

- **Multi-word expressions (MWE)** – Firstly, **the knowledge base is insufficient**, and a considerable number of entities are not recognized, which could be dealt with either by providing **a more comprehensive database** of MWEs. On the other hand, it must be also noted that "*storing longer phrases has the potential to greatly expand the vocabulary size*" (Manning & Raghavan, n.d.), which is a trade-off worth considering generally. Nevertheless, in the context of Simpsons, the number of possible locations and characters should not be that daunting.
  Secondly, because the system encodes the absolute term position in each document, the chosen data structure does not allow the storage of both the multi-word expression and the individual words that comprise it (e.g., '*Springfield Elementary School*' being both mapped to '*springfield_elementari_school' and {'springfield', 'elementari', 'school'}).* This **results in a fast retrieval for specialised queries** (assuming these are indeed correct, and the user knows what they are looking for), as the term is already indexed (memory access is O(1)) and there is no need to merge posting lists (which takes O(n)). However, **this could reduce the recall significantly** since our MWE query could be over-specified and miss the true positive expressions. E.g., the following two examples will illustrate the system's misclassification:
  - '*Springfield Elementary'*
  - *'Springfield Elementary School Prison'*

  will not be recognized in an index containing only '*springfield_elementari_school'.*

Pre-processing choices

The system's pre-processing steps are:

1. *Removal of Wikipedia words & characters* – There are a few expressions and symbols which repeat in every document and are obviously related not to the content of the Simpson's episode but rather on the Wikipedia functionality instead. Hence, they are dropped instantly.
   E.g., From '*Wikipedia, the free encyclopedia*', '*Jump to search*', etc.
2. *Normalization by lowercasing* – To **optimize on index size** (memory requirements) and **index creation & querying** (computational requirements), all characters are converted to lowercase. If there was an entity-recognition embedded system in place, this normalization technique would have to be carefully situated afterwards, to extract the capitalized entities in the middle of the sentence (or apply a supervised method to recognise the expression). In the context of the current implementation, no advanced entity recognition is conducted. Therefore, it seems perfectly reasonable in that context.
3. *Removal of redundant characters & sequences* – currency expressions, citation references, file extensions, etc. are unnecessary to be included in the index as they do not carry any inherent meaning to the TV show.
4. *Stop words are retained* – to **ensure consistency in word positioning and proximity searching** we could restrain from removing the *stop words*. Furthermore, "*most modern IR systems, the additional cost of including stop words is not that high – either in terms of index size or in terms of query processing time*" (Manning & Raghavan, n.d.).
5. *Tokenize terms using NLTK's recommended word tokenizer* – While the system uses a combination of the Tree Bank and the Punkt Sentence Tokenizers, **negative auxiliary verbs are expanded** to their respective unabbreviated form (**without apostrophes**) to further optimize the querying & memory complexity. Additionally, the default **word tokenizer supports lexical hyphens**, so complex expressions like '*voice-over'* are successfully tokenized & stemmed into *'voice-ov'*.
6. *Stem the tokens using the Snowball method* – Firstly, either of stemming or lemmatization is necessary to break down a term into a simpler form. Nevertheless, while the system uses **stemming** because it is **less computationally intensive**, it must also be admitted that "[it] **increases recall while harming precision**" (Manning & Raghavan, n.d.). Additionally, the system uses the **Snowball stemmer**, as it is said to be **the upgraded** version of the classical **Porter's stemmer**.
7. *Tokenize multiple-word expressions (MWEs)* – As discussed in the critique of the *Index term choices* above, our system supports MWE to allow for **faster specialised queries**, but **due to the positional encoding per word and used data structure**, the information of the 'building words' is lost, which results in **reduced recall**. A way to optimize the system would be to have representations for both the MWEs from our knowledge base and the words comprising them (with the term position as well). In that way, advanced users can perform specialised searches, while inexperienced users could benefit from either the **posting lists merging** or **proximity search**. It would also be reasonable to setup a pipeline to 'learn' entities that are MWEs before they are taken as single word expressions during tokenization, especially in those cases where our database with external knowledge is insufficient.
8. *Punctuation removal* – An essential step to reduce the tokens list size.

What is stored in the inverted index

Once the document has been cleaned and tokenized, the system would **iteratively build up the inverted index** per document and per token in that document. What the inverted index stores is:

- the term (be it a single-word or a multi-word expression)
- the document IDs of the documents it occurs in
- the term frequency in a document
- the positions of the term in each document

The data structure implemented (*Figure 1*) for that purpose is based on Python's dictionaries and lists. Interestingly, there is no need for any sort operations for two reasons. Firstly, by inserting terms per document, the system **retains the natural document order** defined by the document filenames. Secondly, Python's dictionaries are based on HashMap, which has as **average time complexity of O(1) per key access**.

Furthermore, by supporting **term frequency** – i.e., the number of times a term was found in a document (*Figure 2*) and **document frequency** – i.e., the number of documents the term was found in the index (*Figure 3*), our system is ready for embedding the TF-IDF ranking functionality.

A way to improve the current system would be to **restrict the inverted index to only single-word expression** (SWE) terms and implement a **phrase index** to keep all **multi-word expressions** (MWEs). In this manner, we would not cause a reduction to the **recall**. Further discussion on optimisation could be found in the Discussion section of Task 2.

Demo

For a comprehensive demonstration of the system's details, please refer to *Figures 6-9*.

Task 2: Positional indexing (proximity search)

Discussion

System Assumption. Query equivalence

Under the current system implementation, the proximity search for terms *A* and *B* in a window *W* does not assume any order (e.g., *A -> B*, or *B -> A*) and thus, it can provide all the documents where terms ***A* and *B* are in any position within the window**. In that manner, we **relax the querying process** and ensure that the all the relevant documents are retrieved, while also keeping in mind that **the user might not know the exact order of the terms beforehand**. For example, searching for both ("*Bart"* AND "*Marge*") and ("*Marge"* AND "Bart") in a window *W* would result in the same documents and positions per document which is reasonable because it is unreasonable to expect the user to know the order of terms in that case (*Figure 4* and *Figure 5* illustrate the **query equivalence**).

Multi-word terms search

Although, the inverted index includes all the multi-word expressions (MWEs) contained in the knowledge base, as discussed in the previous sections this results in:

- **faster information retrieval** for specialised queries
- **reduced recall** for variants (shorter or longer) of the MWEs
- **higher memory load** when storing MWE entities

In the context of Simpsons, what is most concerning is the **reduced recall** which under the current implementation of the inverted index we cannot do much to relax.
Nevertheless, if we decide to restrain from inserting the MWEs into the index and instead break them into single-word pieces, we could end up with a more compact index size. Furthermore, we could **substitute the multi-word querying with a multi-word proximity search** with a window size of 1. This is a feasible optimization to the current system, and it would theoretically result in:

- **slightly slower information retrieval** as we will be always calculating proximity instead of direct memory access to the inverted index
- **improved recall** since extra character/location information will not cause the searching to skip an MWE
- **lower memory load** as phrases will no longer be stored in the index

One issue which will remain unresolved by taking this approach is that we will still encounter **the same low precision** as our stemming will reduce both "*operational functionality*" and "*operative functioning*" to "*['oper', 'function']*".

Appendix A: Supplementary figures

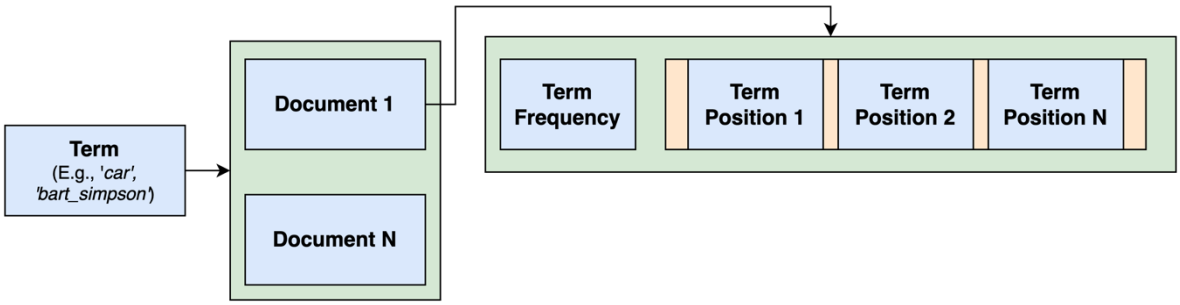

Figure 1. Data Structure of Inverted Index. It contains a mapping between a term and all documents that it is found in. Another mapping is stored for each document to the term frequency and the positions of the term in the document.

```
index.inverted_index["voice-ov"]

{0: [1, [2509]],
 5: [2, [339, 1414]],
 7: [1, [1422]],
 11: [1, [1458]],
 13: [1, [1532]],
 15: [2, [1416, 1623]],
 17: [1, [1004]],
 32: [2, [217, 1894]],
 57: [1, [1146]]}
```

Figure 2. Inverted Index contents for the word 'voice-over' (pre-processed as 'voice-ov'). As shown, for each document ID the system stores the term frequency and term locations.

```
"Elementary" is processed as "elementari"
Document frequency: 6
Document IDs: ['3.22', '4.5', '4.14', '6.2', '6.21', '7.20']
```

Figure 3. Demonstration of document frequency for a queried term

```
Searching for ("run" AND "across")
In document 6.23 pre-processed terms: "run" & "across" are within a window of 2 at positions: [[472, 470]]
In document 6.25 pre-processed terms: "run" & "across" are within a window of 2 at positions: [[76, 77]]
In document 5.3 pre-processed terms: "run" & "across" are within a window of 2 at positions: [[76, 77]]
In document 7.1 pre-processed terms: "run" & "across" are within a window of 2 at positions: [[76, 77]]
"run" and "across" appear within a pre-defined window of 2 in 4 documents. These are:
['6.23', '6.25', '5.3', '7.1']
```

Figure 4. Querying using proximity search for pair of terms ("run" AND "across).

```
Searching for ("across" AND "run")
In document 6.23 pre-processed terms: "across" & "run" are within a window of 2 at positions: [[470, 472]]
In document 6.25 pre-processed terms: "across" & "run" are within a window of 2 at positions: [[77, 76]]
In document 5.3 pre-processed terms: "across" & "run" are within a window of 2 at positions: [[77, 76]]
In document 7.1 pre-processed terms: "across" & "run" are within a window of 2 at positions: [[77, 76]]
"across" and "run" appear within a pre-defined window of 2 in 4 documents. These are:
['6.23', '6.25', '5.3', '7.1']
```

Figure 5. Querying using proximity search for pair of terms ("across" AND "run").

Appendix B: Demo

```
blackboard_set = ['Bart Simpson', 'Bart the Lover', 'Simpsonovi']
blackboard_set_processed = [index.process_document(d) for d in blackboard_set]
blackboard_set_processed
```

```
[['bart_simpson'], ['bart', 'the', 'lover'], ['simpsonovi']]
```

Figure 6. Only multi-word terms that exist in the knowledge base are recognised. E.g., 'Bart Simpson' is while 'Bart the Lover' is not in the database.

```
print(index.dump(['Bart the Lover']))
```

```
"Bart the Lover" is processed as "bart_the_lover"
"bart_the_lover" does not occur in the inverted index

These terms are not in the inverted index: ['bart_the_lover']
Input is not in the inverted index
```

Figure 7. Searching for a multi-word term treated as a single term could be unsuccessful as the entity is not in the database.

```
print(index.dump('Bart the Lover'.split(' ')))
```

```
"Bart" is processed as "bart"
Document frequency: 113
Document IDs: ['3.1', '3.2', '3.3', '3.4', '3.5', '3.6', '3.7', '3.8', '3.9', '3.10', '

"the" is processed as "the"
Document frequency: 118
Document IDs: ['3.1', '3.2', '3.3', '3.4', '3.5', '3.6', '3.7', '3.8', '3.9', '3.10', '

"Lover" is processed as "lover"
Document frequency: 7
Document IDs: ['3.15', '3.16', '3.17', '5.20', '5.21', '5.22', '6.3']

['bart', 'the', 'lover'] are in ['3.15', '3.16', '3.17', '5.20', '5.21', '5.22', '6.3']
['3.15', '3.16', '3.17', '5.20', '5.21', '5.22', '6.3']
```

Figure 8. Searching for a set of single-word terms could successfully retrieve a multi-word entity and the merged posting lists.

```
print(index.dump('Lover nonexistentfiller'.split(' ')))
```

```
"Lover" is processed as "lover"
Document frequency: 7
Document IDs: ['3.15', '3.16', '3.17', '5.20', '5.21', '5.22', '6.3']

"nonexistentfiller" is processed as "nonexistentfil"
"nonexistentfil" does not occur in the inverted index

These terms are not in the inverted index: ['nonexistentfil']
['3.15', '3.16', '3.17', '5.20', '5.21', '5.22', '6.3']
```

Figure 9. In the cases where we query a non-existing word, it is reasonable to still retrieve whatever could be found for the user.

References

1. Manning, C. D.; Raghavan, P. & Schütze, H. (2008), Introduction to Information Retrieval , Cambridge University Press , Cambridge, UK .