THE UNIVERSITY *of* EDINBURGH

# Text Technologies for Data Science

## Coursework 1

Marina Potsi
s1958674

October 20, 2019

# Contents

# 1 Introduction

This report explains the steps that were followed while implementing an Information Retrieval (IR) system, in which we preprocess large documents, create a positional inverted index and apply boolean, phrase and proximity search as well as ranked IR based on Term Frequency – Inverse Document Frequency (TFIDF). The programming language Python was used exclusively for this coursework.

Given a TREC xml file (trec.5000.xml) containing 5000 text documents along with their headlines and other relevant information, we load it to the memory using the function **load_xml**, which calls the ElementTree XML API. We then automatically wrap the whole file in a <ROOT> tag, so that it is parsed correctly by the ElementTree object and we store the headlines that are merged with the document text in a Python dictionary as key - value pairs, with documents ids being the keys and the preprocessed documents being the values.

# 2 Text Preprocessing

Preprocessing of documents is essential for information retrieval, allowing us to transform terms in their simplest form and have a common index for multiple forms of the same term. This way we can achieve better retrieval in terms of performance and time. Our preprocessing includes tokenisation, stopping and stemming.

## 2.a Tokenisation

Tokenisation is the process of splitting the text into smaller parts of the initial one, which are called tokens. Here, before we split it on any whitespace, we apply the following regular expressions:

- *reg_1*: `(^FT\s{2})|([^\w\s])|(\_)`
- *reg_2*: `\s+`

*reg_1* removes the initials "FT" from the beginning of the headline and any punctuation marks including underscore, since they don't give more information about each document and it won't help us distinguish between any of the documents. *reg_2* removes multiple new lines and spaces. Following, we convert the text to its lowercase form and split it having the space as a separator.

```
def tokenise(text):
    """Removes punctuation, new lines, multiple whitespaces and the initial
    FT in front of the headlines and then splits it into tokens
    Args:
        text (string): The text provided to tokenise
    Returns:
        tokenised (list): List of tokens
    """
    reg_1 = r'(^FT\s{2})|([^\w\s])|(\_)'
    reg_2 = r'\s+'

    # Replace punctuation marks and the abbreviation FT at the beginning of the headline with an empty
      string
    no_punctuation_text = re.sub(reg_1, ' ', text, flags=re.MULTILINE)
    # Replace new lines and multiple spaces with empty string
    no_spaces_text = re.sub(reg_2, ' ', no_punctuation_text, flags=re.MULTILINE)
    tokenised = no_spaces_text.lower().strip().split(' ')  # Lowecase and split text
    return tokenised
```

Listing 1: Tokenisation function

### 2.b Stopping

For the removal of English stop words, we download the file from the link provided, save it in the data directory and save it in a list in memory. We then call the function **remove_stop_words** which creates a lambda function and excludes any of the document words that also belong to the *stop_words* list.

### 2.c Stemming

For stemming, we used the Python implementation of the Porter2 stemming algorithm for English, which removes suffixes in order to remove the morphological variations of a word and only have the root.

## 3 Positional Inverted Index

For the positional inverted index, we create a mapping of all unique terms to the documents they belong and their exact positions in each document. We used a Python dictionary data structure named *inverted_index*, with terms as keys and another dictionary of documents and positions as values. If a word already exists in the dictionary, we update the positions list, otherwise we update the key-value pair. The inverted index dictionary is stored in the

required format in **index.txt** and as a binary file (**index.pkl**) in order to save some space.

```
def create_inverted_index(tokenised_docs):
    """Creates the positional inverted incdex from a list of tokenised documents
    Args:
        tokenised_docs (dict): Tokenised documents for each document number
    """
    print('Create inverted index...')
    inverted_index = dict()

    for doc_no, token_doc_list in tokenised_docs.items():
        for word in token_doc_list:
            word_indices = find_indices_of_word(token_doc_list, word)
            doc_indices_dict = {}
            doc_indices_dict[doc_no] = word_indices
            if word in inverted_index:
                inverted_index[word].update(doc_indices_dict)
            else:
                inverted_index.setdefault(word, doc_indices_dict)

    # Save inverted index in txt file in the required format and binary file
    save_inverted_index_txt(inverted_index, INVERTED_INDEX_FILE)
    save_file_binary(inverted_index, INVERTED_INDEX_FILE)
```

Listing 2: create_inverted_index function

# 4 Search

After the inverted index, we create a term-document incident collection that shows which documents each term belongs to. We create a boolean matrix, having as rows the terms of the inverted index and columns the unique document numbers. For each word we create a boolean vector, having True if the word appears in a specific document and False if it doesn't. We will use this boolean matrix to make boolean operations between two or more words or a combination of searches.

## 4.a Boolean, Phrase and Proximity Search

We have created a function **boolean_search_queries** which takes the queries from *queries.boolean.txt* and depending on the type of query, it calls a different search function. If a query starts with # then we know that it's a proximity search and we call the **phrase_proximity_search** with the right parameters. Consequently, if the query is an exact phrase search such as "term1, term2", we treat it as a proximity search with distance 1. The only difference

is that for the phrase search the order matters, so the results for the query above should be different from the results of the query "term2, term1", while for a proximity search we care about the distance of the words being less or equal to the number in front of the query, regardless of their order.

In addition, we split each query into subparts in order to create a boolean numpy array for each one of them and therefore be able to do a search regardless of the number of terms included in it or the combination of searches needed to be done in one query. If the boolean search query includes only two terms connected with the logical operators AND, OR, NOT, we create a boolean array for each term, map each logical operator to one of &, | or ~ and pass the query string into the *eval* method which also returns a boolean array for the whole query. The operation is the same if the query includes both a phrase search and a boolean search, so it offers flexibility for any combination of queries. The results for the boolean queries can be found in *queries.boolean.txt*.

### 4.b Ranked IR based on TFIDF

For the ranked retrieval, we are interested in the documents that contain at least one of the terms in the query, so we create a boolean OR search which returns a boolean vector of the relevant documents. For each document we retrieved, we compute the retrieval score using the TFIDF (term frequency - inverse document frequency) formula, which assigns a weight to each term that belongs to a document. The higher the weight, the more relevant is the document for a specific query. The results for the ranked queries can be found in *queries.ranked.txt*, which includes at most 1000 documents per query.

## 5 Challenges and comments

The challenges faced on this coursework were mainly related to the variability of the queries for the boolean search. Queries can include multiple terms connected with the logical operators AND, OR, NOT, as well as phrases. Therefore we needed a way to compute the search without having to worry about corner cases, limitations related to the extendability of the function or the processing time. The Python eval method seemed to handle these issues, since in can accept any number of boolean numpy arrays.

Last but not least, we could improve the tokenisation by retaining some punctuation marks that are parts of numbers, such as dots, commas, or the ones that are parts of a URL or an email. The tokenisation could also be

extended to support accents and/or other languages, as well as documents from another source, medical text for example, where the structure and vocabulary can be different.