COMP472 – Project 2 Report

Rhina Kim – 40130779

October 8, 2022

Overview

The purpose of this project is to implement a naïve indexer, single term query processing, and lossy dictionary compression indexer. Given the Reuter's Corpus "Reuters-21578", both naïve indexer and compressed indexer extract the raw text of each article from the corpus, tokenize the text for all articles, and compose an inverted index.

The difference between naïve indexer and compressed indexer is that naïve indexer only sorts and removes the duplicates of the postings list given, whereas compressed indexer additionally goes through term reduction processes via (1) removing numbers, (2) case folding, (3) removing stop words, (4) applying porter stemmer. It is called "lossy dictionary compression indexer" for the reason of reducing more and more dictionary (distinct terms) to make a compact index.

Different results of inverted index were generated using naïve indexer and using compressed indexer with 5 reduction process. The results are specified in the DEMO file.

Moreover, single term query processing is implemented to perform query search on both naïve indexer and compressed indexer, using sample queries provided by our professor *Ms. Sabine Bergler*. This result outputs are also specified in the DEMO file.

Technologies

Language

- Python 3.8

Interpreter

- Pypy3

Python packages

- Beautiful Soup 4.9
- NLTK 3.7
    - *nltk.word_tokenize*
    - *nltk.stem.PorterStemmer*
    - *nltk.corpus.stopwords*
- PrettyTable

Python 3.8 is used as programming language for this project since it supports natural language processing with Python's NLTK package. Moreover, pypy3 is used as a replacement for original Python 3 interpreter since its runtime interpreter is faster. As this project requires iteration of massive number of large files, using pypy3 as an alternative interpreter seemed to be an optimal choice.

NLTK is brought to this project to tokenize, to apply Porter Stemmer, and to remove stop words from the corpus. Beautiful Soup is given for extracting the text data from *.sgm* files which is composed of markup-languages like HTML and XML. Finally, PrettyTable is used to make a lossy dictionary compression statistics table which supports Subproject 3.

Program Design

Step 0 – Read, Extract raw texts from Reuter's corpus, and Tokenize (Inherited from Project 1)

At beginning, the following operations are performed prior to composing an inverted index through naïve indexer.

```
    for filename in sgm_files:
        # open the file to be read
        sgm_file = open(filename, 'r', encoding='utf-8', errors='ignore')
        # Read Reuter's collection, extract the raw text of each article
        documents = extract_documents_from_corpus(sgm_file)
        # Tokenize each article
        postings = tokenize(documents)
        # Append postings to F
        F.extend(postings)
```

After extracting the Reuter's packaged folder and only collecting SGM format datafiles, the above process extracts raw texts from those datafiles via iteration, performs article tokenization, and all postings results for all datafiles are appended back to the list called "*F*" which generates lists of *tuple(term, documentID)* pairs that will be later used in Step 1 (Subproject 1) section.

The extracting and tokenizing operations are defined in Project1 phase. Please refer to the Project 1 report for more details.

Step 1 – Implement Naïve Indexer (Subproject 1)

The following simplified code explains the implementation of Naïve indexer:

```
########## Subproject 1 ##########
def S1_naive_indexer(F):
    F = sort_postings(F) # Sort postings
    F = remove_duplicates(F) # remove duplicates from postings
    index = inverted_index(F)#CComplete inverted index

def sort_postings(F):
    # sort listings based on the term in ascending alphabetical order
    F.sort(key = lambda posting: posting[1])
    return F

def remove_duplicates(F):
    new_F = list(set(F))
    # sort again since list got messy after removing duplicates
    new_F = sort_postings(new_F)
    return new_F

def inverted_index(F):
    inverted_index = {} # empty inverted index
    for docID, term in F:
        if term in inverted_index:
            # append docIDs to the existing term
```

```
            inverted_index[term].append(docID)
        else:
            inverted_index[term] = [docID]
    return inverted_index
```

Naïve Indexer have two operations:

(1) Sorting posting lists: The posting lists "*F*" returned from Step 0 are in the tuple form (*documentID, term*). The *sort_postings(F)* function takes the posting lists and sort the *term* attribute in ascending alphabetical order.

(2) Removing duplicate term-postings pair from the postings list: The posting lists "*F*" returned after the sorting algorithm then goes through remove_duplicates(F) which removes all identical *terms* residing in same *documentID* using *set()* operation.

Ex) Term: DocumentID → Term: DocumentID

A: 2000
A: 2000 → A: 2002
A: 2002

After completing two operations, the function proceeds to forming Naïve Indexer via *inverted_index(F)* which pairs the unique term with list of *documentID*s. The format of this returned index is in dictionary where its key refers to distinct terms and its value refers to all document positions of that term.

Ex) indexer = {dictionary: posting lists}

Assays: 5933 → 5958

Assembly: 11323 → 11355 → 10030 → 11742 → 4502

Step 2 – Implement Single Term Query Search (Subproject 2)

This step implements the query search operation via *S2_naive_query_search(index*) or *S3_compressed_index_search(index)* where the function takes list of sample queries (previously defined) as input and search for each query in the given inverted index.

The list of the sample queries given by our professor:

```
SAMPLE_QUERIES = ["copper", "Samjens", "Carmark", "Bundesbank"]
```

The following is the simplified implementation of single query search:

```
########## Subproject 2 ##########
def S2_naive_query_search(index):
    for query in SAMPLE_QUERIES:
        # do query search
        postings, num_postings = query_search(query, index)

        # if query exists in postings
        if postings:
            print(Documents mentioning "' + query + '" exists based on the index.')
            print(Document position: ")
            for posting in postings:
                print(posting)
            print(Term frequency: " + str(num_postings))
```

```
        # if no queries in postings
        else:
            print('No documents mentioning "' + query + '" has found.)
```

The *S2_naive_query_search(index)* and *S3_compressed_query_search*(*index*) both have same implementation. The only difference is its function's argument where one takes naive indexer whereas the other takes compressed indexer which will be later brought from Step 3 (Subproject 3). Despite the same implementation, I have separated the query search functions into two, S2 and S3, for the better execution flow (subproject 1 naïve indexer → subproject 2 query search with naïve indexer → subproject 3 compressed indexer → subproject 2 query search with compressed indexer)

One thing to note from this query search process is that I have made two versions of query processing within the function: (1) input sample queries as it is, or (2) make all the sample queries lowercase before the query search.

```
# With lowercasing queries
def S2_naive_query_search(index):
    for query in SAMPLE_QUERIES:
        if query.isalpha:
            query = query.lower()

        … REST IS SAME PROCESS …
```

I have considered these two cases since there will be "case folding" process in Subproject 3 where all alphabetic term in the inverted index will be converted to lowercase letters. Since some of the sample queries were provided in Capital letter, I have considered this other case to take account of possible "if" scenarios.

Step 3 – Implement Lossy Dictionary Compression Indexer (Subproject 3)

This step implements another type of indexer: Lossy Dictionary Compression Indexer which adds more functionalities to the naive indexer. The purpose of this indexer is to perform index compression – reduce number of distinct terms as much as possible – for the reason of aiming increased performance (conserve disk space, augment performing speed).

The indexer goes through four steps:

(1) Removing numeric terms (as well as its postings list)
(2) Make all terms lowercase (upper case term and lowercase term will turn into one distinct term, and its posting lists will get merged)
(3) Remove 30 stop words OR 150 stop words from the index
(4) Apply porter stemmer

The following is the simplified implementation of compressed indexer:

```
def S3_compressed_indexer(F):
    F = sort_postings(F)
    F = remove_duplicates(F)
    index = inverted_index(F)
    # Default Case
    num_terms_default, num_postings_default, index = default(index)
    # Remove numbers
```

```
    num_terms_no_numbers, num_postings_no_numbers, index = no_numbers(index)
    # Case folding
    num_terms_lowercase, num_postings_lowercase, index = case_folding(index)
    # 30 stop words
    num_terms_30_stopwords, num_postings_30_stopwords, index = 30_stop_words(index)
    # 150 stop words
    num_terms_150_stopwords, num_postings_150_stopwords, index = 150_stop_words(index)
    # Porter stemming
    num_terms_stemmer, num_postings_stemmer, index = stemmer(index)

def compute_distinct_terms(index):
    num_terms = len(index)
    return num_terms

def compute_nonpositional_postings(index):
    num_postings = 0
    for postings in index.values():
        num_postings += len(postings)
    return num_postings

def compute_reduction(start, final):
    # Reduction = (Starting value − Final value) / (Starting value) × 100
    return round((start - final) / start * 100, 2)
```

Every reduction function takes inverted index as an argument, performs reduction process such as removing stop words, applying porter stemmer, etc. (this was already completed in Project 1 phase) and computes the number of distinct terms, number of non-positional postings (total postings count), reduction value (%Δ) and cumulative reduction value (%T).
(*) Remember, inverted index variable in the program is a type of dictionary which is composed of distinct term as its keys and postings lists as its values:
Ex) indexer = {dictionary: posting lists [docId, docId, …]
Assays: [5933, 5958]
Assembly: [11323, 11355, 10030, 11742, 4502]
}

Number of distinct terms are computed by taking the size of the index which is in type *dict*: *len(index)* since its keys represent distinct terms.

Number of non-positional listings are computed by taking all values inside the dictionary "*index*" via *postings = index.values()* and iterate again those *postings* to retrieve all the listed *documentID*s. In other words, the total number of *documentIDs* inside the index represents the non-positional listings.

Using all the return values from each reduction functions, it constructs the dictionarty compression table which is showcased in DEMO file.