605.744: Information Retrieval
Programming Assignment #1: Corpus Statistics

Sabbir Ahmed

September 6, 2022

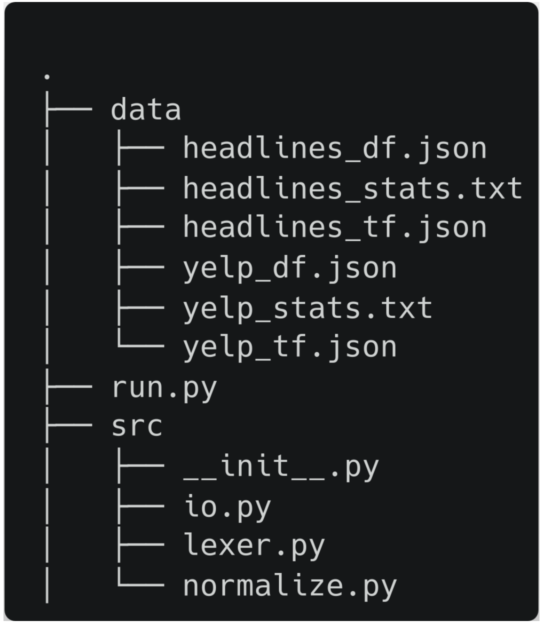
1 Introduction

This paper describes a collection of scripts written to compute the term-frequency, document-frequency, and other statistics from a pre-generated corpus.

2 Technical Background

All of the source code is in Python 3.10. The program is split into several modules and follows an object oriented structure. The following is the directory structure of the source code:

Figure 1: Directory Hierarchy of Assignment 1

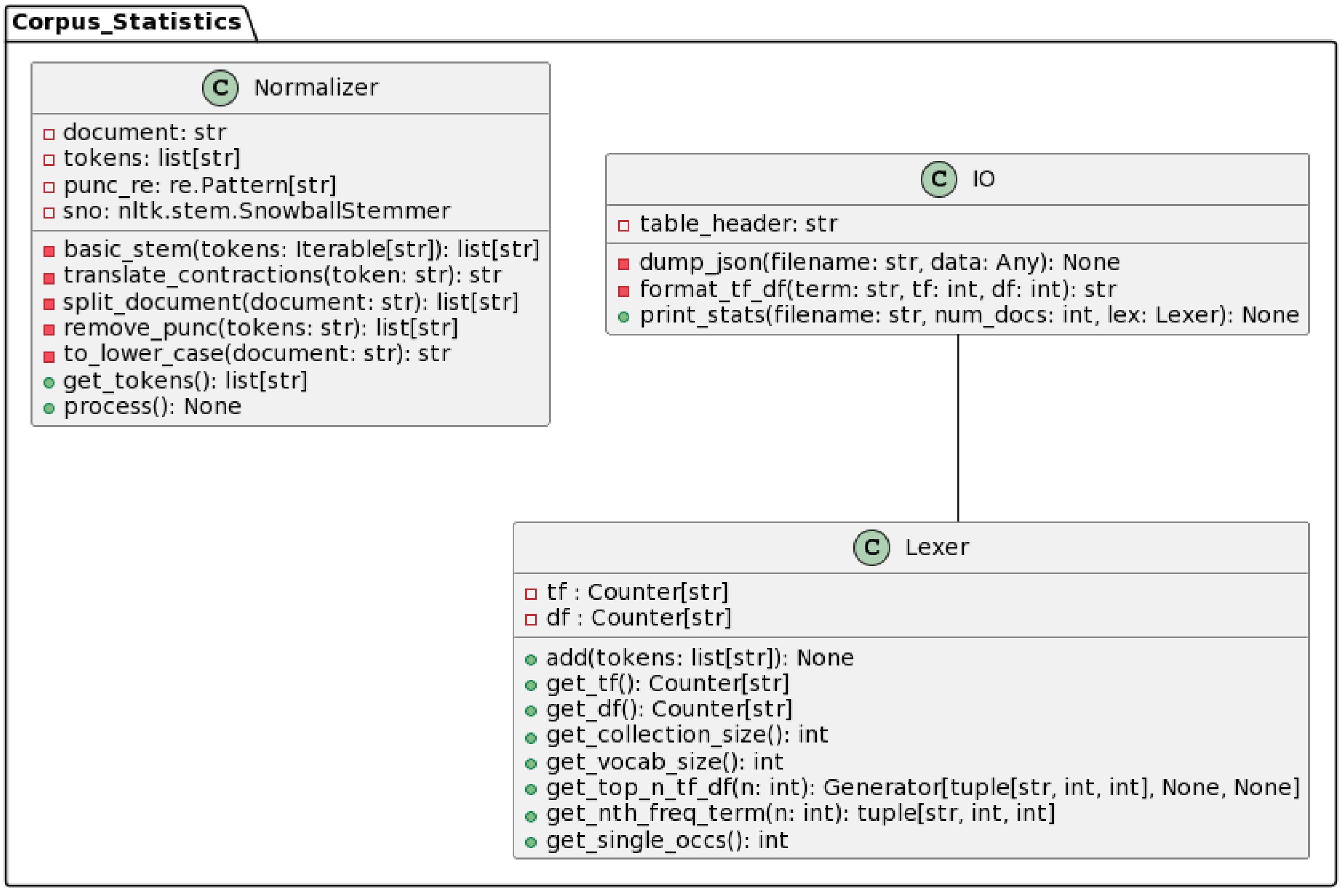


```
.
├── data
│   ├── headlines_df.json
│   ├── headlines_stats.txt
│   ├── headlines_tf.json
│   ├── yelp_df.json
│   ├── yelp_stats.txt
│   └── yelp_tf.json
├── run.py
├── src
│   ├── __init__.py
│   ├── io.py
│   ├── lexer.py
│   └── normalize.py
```

The source code for all of the files are attached in Appendix A.

The total number of non-empty lines of code for the program comes to 200, with an average execution time of 40 seconds to process both of the sample files.

Figure 2: UML of Corpus Statistics


Corpus_Statistics
C Normalizer
document: str
tokens: list[str]
punc_re: re.Pattern[str]
sno: nltk.stem.SnowballStemmer
basic_stem(tokens: Iterable[str]): list[str]
translate_contractions(token: str): str
split_document(document: str): list[str]
remove_punc(tokens: str): list[str]
to_lower_case(document: str): str
get_tokens(): list[str]
process(): None
C IO
table_header: str
dump_json(filename: str, data: Any): None
format_tf_df(term: str, tf: int, df: int): str
print_stats(filename: str, num_docs: int, lex: Lexer): None
C Lexer
tf : Counter[str]
df : Counter[str]
add(tokens: list[str]): None
get_tf(): Counter[str]
get_df(): Counter[str]
get_collection_size(): int
get_vocab_size(): int
get_top_n_tf_df(n: int): Generator[tuple[str, int, int], None, None]
get_nth_freq_term(n: int): tuple[str, int, int]
get_single_occs(): int

2.1 Driver

The driver script for the program is `run.py`. The file is responsible for opening up the input files, reading in all of the documents contained in those files, processing them into the lexicon, and generating and saving statistics on them. Since the sample files are relatively small, the entire content may be loaded into the memory of a modern machine. However, to account for possible larger files, the lines containing the documents are saved in memory, normalized, and added to the lexicon before moving to the next document.

This method of reading the files imposes a limitation, where every document is assumed to be on the 2nd line out of 4 (line_num % 4 == 1). The algorithm will need to be modified if the files in the future have different formats.

2.2 `normalize.Normalizer`

The `Normalizer` class normalizes the documents. The class is instantiated once and it provides a method to set the document for normalization. The normalization pipeline includes the following processes, in order:

1. translating the entire document to lower case. The document is unconditionally transformed into lower case, which can lead to confusion with words that are considered proper nouns.

2. splitting the document on whitespace into tokens.
3. expanding word contractions into their components. The contractions are pre-generated into a Python dict in `normalize.CONTRACTIONS` for constant-time look-ups.
4. splitting the tokens on all punctuations except "’". The "’" is preserved for the stemmer to process later. Not splitting on this character also avoids including incorrect tokens into the lexicon. For example, "food’s" will get tokenized into ["food", "s"].
5. rejoining and splitting the tokens to remove empty tokens.
6. stemming tokens with nltk.stem.SnowballStemmer.stem(word).

The tokens are available as a list of string to be added into the lexicon before being overwritten by the next document.

2.2.1 External Libraries

The external library `nltk` was used for its stemming capabilities. The stemmers `PorterStemmer` and `SnowballStemmer` were tested over a sample of the documents, and the latter algorithm appeared to yield subjectively better-looking results.

2.3 `lexer.Lexer`

The `Lexer` class utilizes 2 `collections.Counter` objects to store the term-frequency and document-frequency of the lexicon. The bag-type container is a part of the standard library and match in performance compared to default `dicts`, but with additional methods such as `Counter.total()`, `Counter.most_common(n)`, etc. These methods are helpful in generating statistics required in the assignment.

3 Statistics and Observations

Since stopwords were not removed, they were prominent in the 100 most frequent terms in both of the files. "the" and "to" were present in the top 5 most frequent terms in both of the sets of documents. In the Yelp corpus, some of the most commonly occurring words were related to food and restaurant services. This theme is expected due to the source of the documents containing reviews of mostly food service establishments and its quality. The Headlines corpus did not appear to circulate a theme, although some of the frequent words appeared to be numerical or date values. This can be expected due to the source of the documents mostly discussing events with timelines. The numerical values can indicate money, percentage, weather, length of service, etc.

Both of the corpora had an almost identical percentage of singly-occurring terms (48.8%). These are terms that appeared in only one document.

A Source Code

Code Listing 1: ./src/io.py

```
import json
from typing import Any

from .lexer import Lexer
```

```
class IO:

    __table_header: str = (
        f"{'Word':<12} | {'TF':<6} | {'DF':<6}\n----------------------------"
    )

    @staticmethod
    def __dump_json(filename: str, data: Any) -> None:

        with open(f"data/{filename}.json", "w") as fp:
            json.dump(data, fp)

    @staticmethod
    def __format_tf_df(term: str, tf: int, df: int) -> str:

        return f"{term:<12} | {tf:<6} | {df:<6}"

    @staticmethod
    def print_stats(filename: str, num_docs: int, lex: Lexer) -> None:

        print("Processed", num_docs, "documents.")
        IO.__dump_json(filename + "_tf", lex.get_tf())
        IO.__dump_json(filename + "_df", lex.get_df())

        with open(f"data/{filename}_stats.txt", "w") as fp:

            print("----------------------------", file=fp)
            print(num_docs, "documents.", file=fp)

            print("----------------------------", file=fp)
            print("Collections size:", lex.get_collection_size(), file=fp)
            print("Vocabulary size:", lex.get_vocab_size(), file=fp)
            print("\n----------------------------", file=fp)

            print("Top 100 most frequent words:", file=fp)
            print(IO.__table_header, file=fp)
            for term in lex.get_top_n_tf_df(100):
                print(IO.__format_tf_df(*term), file=fp)

            print("\n----------------------------", file=fp)
            print("500th word:", file=fp)
            print(IO.__table_header, file=fp)
            print(IO.__format_tf_df(*lex.get_nth_freq_term(500)), file=fp)

            print("\n----------------------------", file=fp)
            print("1000th word:", file=fp)
            print(IO.__table_header, file=fp)
            print(IO.__format_tf_df(*lex.get_nth_freq_term(1000)), file=fp)

            print("\n----------------------------", file=fp)
            print("5000th word:", file=fp)
            print(IO.__table_header, file=fp)
            print(IO.__format_tf_df(*lex.get_nth_freq_term(5000)), file=fp)

            print("\n----------------------------", file=fp)
            single_occs: int = lex.get_single_occs()
            print(
                "Number of words that occur in exactly one document:", file=fp
            )
            print(
                single_occs,
                f"({round(single_occs / lex.get_vocab_size() * 100, 2)}%)",
                file=fp,
            )
```

Code Listing 2: ./src/normalize.py

```
import re
from typing import Iterable

import nltk

CONTRACTIONS: dict[str, str] = {
    "aren’t": "are not",
    "ain’t": "is not",
    "can’t": "cannot",
    "couldn’t": "could not",
    "didn’t": "did not",
    "doesn’t": "does not",
    "don’t": "do not",
    "hadn’t": "had not",
    "hasn’t": "has not",
    "haven’t": "have not",
    "he’d": "he had",
    "he’ll": "he will",
    "he’s": "he is",
    "i’d": "i had",
    "i’ll": "i will",
    "i’m": "i am",
    "i’ve": "i have",
    "isn’t": "is not",
    "it’s": "it is",
    "let’s": "let us",
    "mightn’t": "might not",
    "mustn’t": "must not",
    "shan’t": "shall not",
    "she’d": "she had",
    "she’ll": "she will",
    "she’s": "she is",
    "shouldn’t": "should not",
    "that’s": "that is",
    "there’s": "there is",
    "they’d": "they had",
    "they’ll": "they will",
    "they’re": "they are",
    "they’ve": "they have",
    "wasn’t": "was not",
    "we’d": "we had",
    "we’re": "we are",
    "we’ve": "we have",
    "weren’t": "were not",
    "what’ll": "what will",
    "what’re": "what are",
    "what’s": "what is",
    "what’ve": "what have",
    "where’s": "where is",
    "who'd": "who had",
    "who'll": "who will",
    "who're": "who are",
    "who's": "who is",
    "who've": "who have",
    "won't": "will not",
    "wouldn't": "would not",
    "you'd": "you had",
    "you'll": "you will",
    "you're": "you are",
    "you've": "you have",
}


class Normalizer:
    def __init__(self) -> None:

        self.__document: str = ""
        self.__tokens: list[str] = []

        self.__punc_re: re.Pattern[str] = re.compile(
            '[!"#$%&()*+,-./:;<=>?@[\\]^_`{|}~]'
        )
        self.__sno: nltk.stem.SnowballStemmer = nltk.stem.SnowballStemmer(
            "english"
        )

    def set_document(self, document: str) -> None:

        self.__document = document[:-1]

    def __basic_stem(self, tokens: Iterable[str]) -> list[str]:

        return [self.__sno.stem(token) for token in tokens]

    def __translate_contractions(self, token: str) -> str:

        return CONTRACTIONS.get(token, token)

    def __split_document(self, document: str) -> list[str]:

        return document.split(" ")

    def __remove_punc(self, tokens: str) -> list[str]:

        return self.__punc_re.split(tokens)

    def __to_lower_case(self, document: str) -> str:

        return document.lower()

    def get_tokens(self) -> list[str]:

        return self.__tokens

    def process(self) -> None:

        self.__document = self.__to_lower_case(self.__document)
        self.__tokens = self.__split_document(self.__document)
        temp_str: str = ""
        for token in self.__tokens:
            temp_str += self.__translate_contractions(token) + " "

        no_puncs: list[str] = self.__remove_punc(temp_str)
        no_empty: Iterable[str] = filter(
            None, self.__split_document(" ".join(no_puncs))
        )
        self.__tokens = self.__basic_stem(no_empty)
```

Code Listing 3: ./src/lexer.py

```
from collections import Counter
from typing import Generator


class Lexer:
    def __init__(self) -> None:

        self.__tf: Counter[str] = Counter()
        self.__df: Counter[str] = Counter()

    def add(self, tokens: list[str]) -> None:

        self.__tf.update(tokens)
        self.__df.update(set(tokens))

    def get_tf(self) -> Counter[str]:

        return self.__tf

    def get_df(self) -> Counter[str]:

        return self.__df

    def get_collection_size(self) -> int:

        return self.__tf.total()

    def get_vocab_size(self) -> int:

        return len(self.__tf)

    def get_top_n_tf_df(
        self, n: int
    ) -> Generator[tuple[str, int, int], None, None]:

        top_n_tf = self.__tf.most_common(n)
        for tf in top_n_tf:
            term, freq = tf
            yield term, freq, self.__df[term]

    def get_nth_freq_term(self, n: int) -> tuple[str, int, int]:

        term, freq = self.__tf.most_common(n)[-1]
        return term, freq, self.__df[term]

    def get_single_occs(self) -> int:
        single_occs: int = 0
        for df in self.__df.values():
            if df == 1:
                single_occs += 1

        return single_occs
```

Code Listing 4: ./run.py

```
from src.io import IO
from src.normalize import Normalizer
from src.lexer import Lexer


if __name__ == "__main__":

    def process_document(filename: str) -> None:

        prep = Normalizer()
        lex = Lexer()
        io = IO()

        line_num = 0
        num_docs = 0

        with open(filename) as fp:
            for line in fp:
                match line_num % 4:
                    case 0:
                        num_docs += 1
                    case 1:
                        prep.set_document(line)
                        prep.process()
                        lex.add(prep.get_tokens())
                    case _:
                        pass
                line_num += 1

        io.print_stats(filename[:-4], num_docs, lex)

    process_document("yelp.txt")
    process_document("headlines.txt")
```