605.744: Information Retrieval
Programming Assignment #3: Inverted Files

Sabbir Ahmed

October 16, 2022

1 Introduction

This paper describes the enhancements and features added to the Information Retrieval program started in Assignment 1 and upgraded in Assignment 2. Modifications include improvement in performance and adding support for batch processing and ranking queries.

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:

```
.
├── bin/
├── ir/
│   ├── const.py
│   ├── files.py
│   ├── __init__.py
│   ├── invertedfile.py
│   ├── lexer.py
│   ├── normalize.py
│   ├── packer.py
│   ├── retriever.py
│   └── types.py
├── run.py
├── stats/
└── tmp/
```
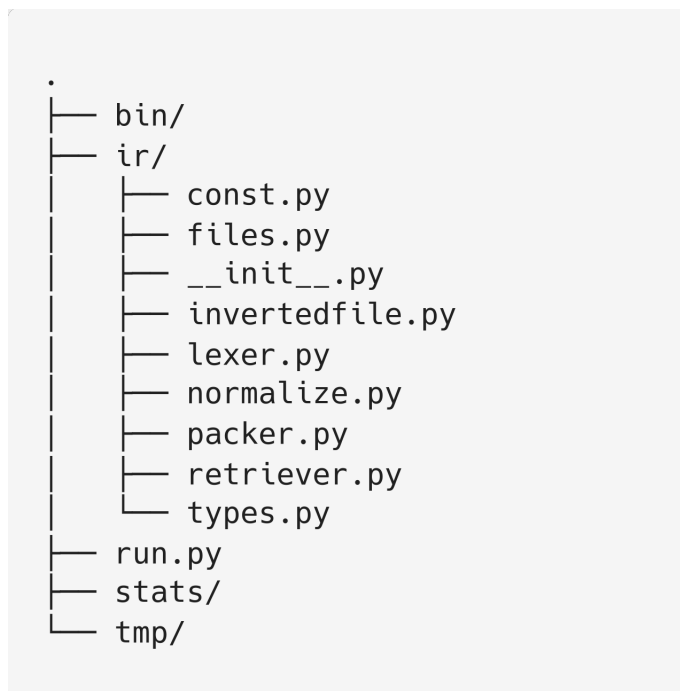
Figure 1: Directory Hierarchy of Assignment 3

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 totals to under 750.
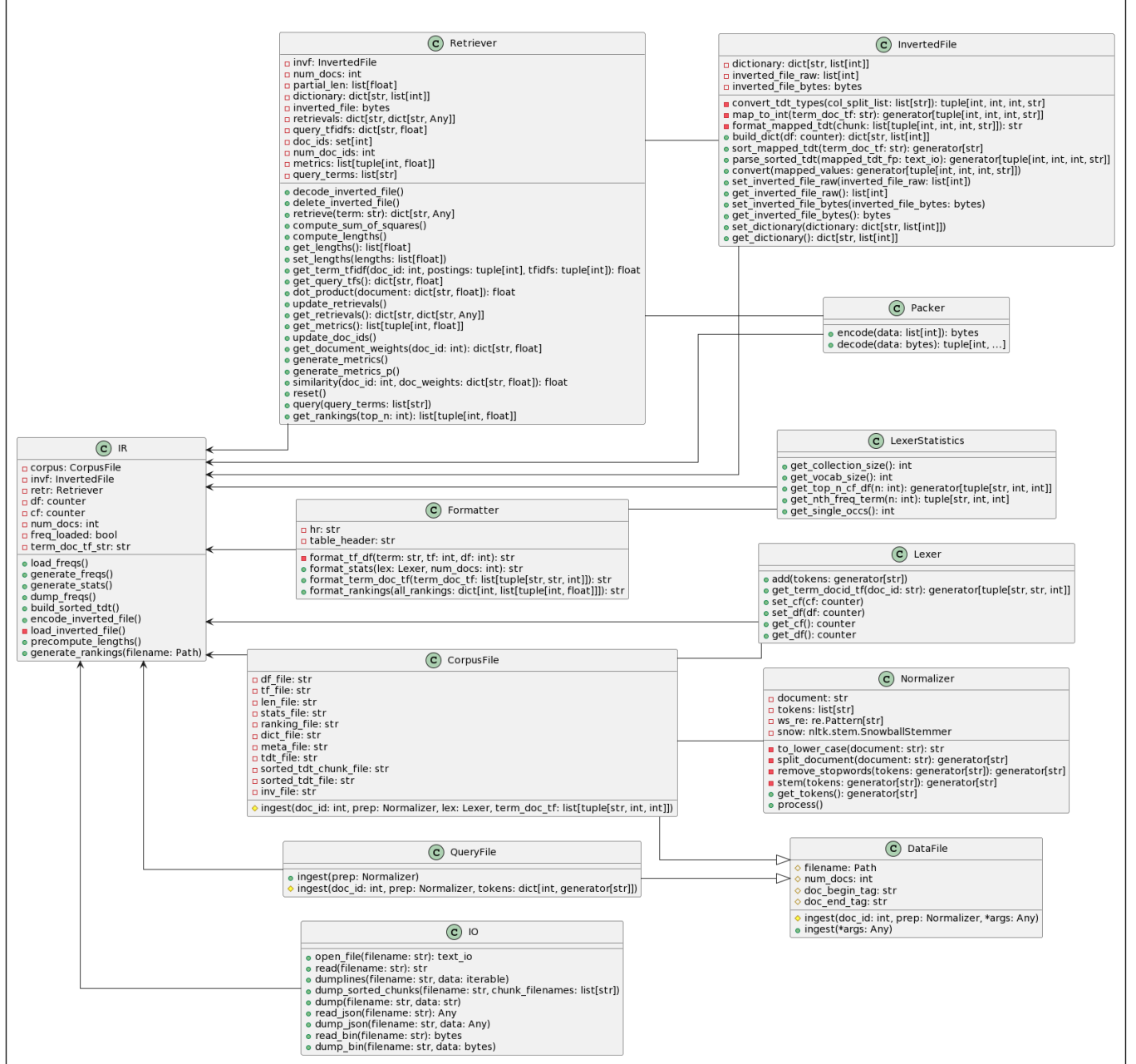
Retriever
- invf: InvertedFile
- num_docs: int
- partial_len: list[float]
- dictionary: dict[str, list[int]]
- inverted_file: bytes
- retrievals: dict[str, dict[str, Any]]
- query_tfidfs: dict[str, float]
- doc_ids: set[int]
- num_doc_ids: int
- metrics: list[tuple[int, float]]
- query_terms: list[str]

- decode_inverted_file()
- delete_inverted_file()
- retrieve(term: str): dict[str, Any]
- compute_sum_of_squares()
- compute_lengths()
- get_lengths(): list[float]
- set_lengths(lengths: list[float])
- get_term_tfidf(doc_id: int, postings: tuple[int], tfidfs: tuple[int]): float
- get_query_tfs(): dict[str, float]
- dot_product(document: dict[str, float]): float
- update_retrievals()
- get_retrievals(): dict[str, dict[str, Any]]
- get_metrics(): list[tuple[int, float]]
- update_doc_ids()
- get_document_weights(doc_id: int): dict[str, float]
- generate_metrics()
- generate_metrics_p()
- similarity(doc_id: int, doc_weights: dict[str, float]): float
- reset()
- query(query_terms: list[str])
- get_rankings(top_n: int): list[tuple[int, float]]

InvertedFile
- dictionary: dict[str, list[int]]
- inverted_file_raw: list[int]
- inverted_file_bytes: bytes

- convert_tdt_types(col_split_list: list[str]): tuple[int, int, int, str]
- map_to_int(term_doc_tf: str): generator[tuple[int, int, int, str]]
- format_mapped_tdt(chunk: list[tuple[int, int, int, str]]): str
- build_dict(df: counter): dict[str, list[int]]
- sort_mapped_tdt(term_doc_tf: str): generator[str]
- parse_sorted_tdt(mapped_tdt_fp: text_io): generator[tuple[int, int, int, str]]
- convert(mapped_values: generator[tuple[int, int, int, str]])
- set_inverted_file_raw(inverted_file_raw: list[int])
- get_inverted_file_raw(): list[int]
- set_inverted_file_bytes(inverted_file_bytes: bytes)
- get_inverted_file_bytes(): bytes
- set_dictionary(dictionary: dict[str, list[int]])
- get_dictionary(): dict[str, list[int]]

Packer
- encode(data: list[int]): bytes
- decode(data: bytes): tuple[int, ...]

LexerStatistics
- get_collection_size(): int
- get_vocab_size(): int
- get_top_n_cf_df(n: int): generator[tuple[str, int, int]]
- get_nth_freq_term(n: int): tuple[str, int, int]
- get_single_occs(): int

IR
- corpus: CorpusFile
- invf: InvertedFile
- retr: Retriever
- df: counter
- cf: counter
- num_docs: int
- freq_loaded: bool
- term_doc_tf_str: str

- load_freqs()
- generate_freqs()
- generate_stats()
- dump_freqs()
- build_sorted_tdt()
- encode_inverted_file()
- load_inverted_file()
- precompute_lengths()
- generate_rankings(filename: Path)

Formatter
- hr: str
- table_header: str

- format_tf_df(term: str, tf: int, df: int): str
- format_stats(lex: Lexer, num_docs: int): str
- format_term_doc_tf(term_doc_tf: list[tuple[str, str, int]]): str
- format_rankings(all_rankings: dict[int, list[tuple[int, float]]]): str

Lexer
- add(tokens: generator[str])
- get_term_docid_tf(doc_id: str): generator[tuple[str, str, int]]
- set_cf(cf: counter)
- set_df(df: counter)
- get_cf(): counter
- get_df(): counter

CorpusFile
- df_file: str
- tf_file: str
- len_file: str
- stats_file: str
- ranking_file: str
- dict_file: str
- meta_file: str
- tdt_file: str
- sorted_tdt_chunk_file: str
- sorted_tdt_file: str
- inv_file: str

- ingest(doc_id: int, prep: Normalizer, lex: Lexer, term_doc_tf: list[tuple[str, int, int]])

Normalizer
- document: str
- tokens: list[str]
- ws_re: re.Pattern[str]
- snow: nltk.stem.SnowballStemmer

- to_lower_case(document: str): str
- split_document(document: str): generator[str]
- remove_stopwords(tokens: generator[str]): generator[str]
- stem(tokens: generator[str]): generator[str]
- get_tokens(): generator[str]
- process()

QueryFile
- ingest(prep: Normalizer)
- ingest(doc_id: int, prep: Normalizer, tokens: dict[int, generator[str]])

DataFile
- filename: Path
- num_docs: int
- doc_begin_tag: str
- doc_end_tag: str

- ingest(doc_id: int, prep: Normalizer, *args: Any)
- ingest(*args: Any)

IO
- open_file(filename: str): text_io
- read(filename: str): str
- dumplines(filename: str, data: iterable)
- dump_sorted_chunks(filename: str, chunk_filenames: list[str])
- dump(filename: str, data: str)
- read_json(filename: str): Any
- dump_json(filename: str, data: Any)
- read_bin(filename: str): bytes
- dump_bin(filename: str, data: bytes)

Figure 2: UML of Information Retrieval

2.1 Existing Classes

2.1.1 Driver

The driver script for the program is `run.py`. The script uses command line options to process corpus files to perform the following operations:

1. generate document and relative term frequencies

2. generate statistics on corpus frequencies

3. save frequencies to file

4. extract term-docID-term frequency records and save to temporary files

5. generate, encode and save inverted file

6. compute document weights and save for repeated use

7. process query files and generate rankings

2.1.2 `lexer.Lexer`

The `Lexer` class had its methods on generating statistics separated into the `LexerStatistics` class.

2.1.3 `files` Classes

The `DataFile` class was renamed to `CorpusFile` and the `QueryFile` class was added to ingest query files. Both of these classes have been derived from the `DataFile` class.

The `Formatter` class has a new method to format rankings.

The `IO` class has a new method for writing chunks of data to files to be merged again. This method is only used for dumping chunks of sorted term-docID-term frequency records.

2.1.4 `invertedfile.InvertedFile`

The `InvertedFile` class received the following modifications:

- the dictionary now only contains the term index, offset, length, and document frequency for each terms.

- the term-docID-term frequency records were sorted and saved to files in chunks. The chunk files are later merged to a single sorted file. This approach allows for the program to process large corpuses without stressing the memory.

2.2 New Classes

2.2.1 `retriever.Retriever`

The`Retriever` class was added to compute document weights and rank queries.

The class precomputes document weights to save to a file for repeated use.

3 Statistics and Observations

With the modifications made to the text normalization process, the statistics generated from the input file have changed. The new top 10 more frequent tokens now include "market", "announc", and "report", which are terms that expected in headlines.

In terms of the inverted files and dictionaries, the space they occupy on disk combined are significantly lower than the original document.

File	Size (in bytes)	Description
headlines.txt	39381610	Input corpus file
headlines_dict.json	4275427	Generated dictionary JSON file
headlines_if.bin	27774312	Inverted binary file

Table 1: Sizes of Files Computed Through the `stat` Command on a Debian Based Linux

4 Testing

The tests described by the prompt were performed via the `test.py` driver script. The tokens to look up had to first be normalized through the `Normalizer` class.

Code Listing 1: Test 1: Document frequency and postings list for the terms: "Heidelberg" "cesium" "Trondheim" "crustacean"

```python
# NOTE: the normalization stems "crustacean" to "crustacea"
# and "crustaceans" to "crustacean"
tokens1 = normalize_test_terms(
    prep, ("Heidelberg", "cesium", "Trondheim", "crustaceans")
)
results1 = read_inverted_file(
    invf,
    tokens1,
    ("term", "postings", "postings_len"),
)
```

Code Listing 2: Test 2: Document frequency for the words: "Hopkins" "Stanford" "Brown" and "college"

```python
tokens2 = normalize_test_terms(
    prep, ("Hopkins", "Stanford", "Brown", "college")
)
results2 = read_inverted_file(invf, tokens2, ("term", "postings_len"))
```

Code Listing 3: Test 3: docids for documents that have both "Elon" and "Musk"

```python
# NOTE: the normalization stems "Musks" to "Musk"
tokens3 = normalize_test_terms(prep, ("Elon", "Musks"))
results3 = read_inverted_file(invf, tokens3, ("postings",))
elon, musk = results3
elon_postings, musk_postings = set(elon["postings"]), set(musk["postings"])
elon_musk_postings = elon_postings & musk_postings
```

A Source Code

Code Listing 4: ../ir/const.py

```python
DOC_PROC = 10_000
CHUNK_SIZE = 1_000_000

BYTE_FMT_CHAR = "I"
BYTE_FMT_SIZE = 4

QUERY_DOC_ID = 0
```

```python
# arbitrary - 75,000 for keywords and 90,000 for questions
PARALLEL_THRESH = 90_000


class IDX:

    # shared indices
    TID = 0

    class DICT:

        # dictionary indices
        OF = 1   # offset
        WID = 2   # length
        DF = 3   # document frequency

    class INVF:

        # inverted file indices
        DID = 1   # document ID
        TF = 2   # term count
        STR = 3   # term string



JHED = "sahmed80"
```

Code Listing 5: ../ir/files.py

```python
import heapq
import json
from pathlib import Path

from .const import DOC_PROC, JHED
from .lexer import Lexer, LexerStatistics
from .normalize import Normalizer
from .types import Any, iterable, text_io, generator


class IO:
    @staticmethod
    def open_file(filename: str) -> text_io:

        return open(f"{filename}.txt")

    @staticmethod
    def read(filename: str) -> str:

        with open(f"{filename}.txt") as fp:
            return fp.read()

    @staticmethod
    def dumplines(filename: str, data: iterable) -> None:

        with open(f"{filename}.txt", "w") as fp:
            fp.writelines(data)
        print(f"Dumped to '{filename}.txt'")

    @staticmethod
    def dump_sorted_chunks(filename: str, chunk_filenames: list[str]) -> None:
```

```python
        chunks: list[text_io] = []
        for chunk_filename in chunk_filenames:
            chunks.append(IO.open_file(chunk_filename))

        with open(f"{filename}.txt", "w") as fp:
            fp.writelines(heapq.merge(*chunks, key=lambda k: int(k.split()[0])))

        for chunk_files in chunks:
            chunk_files.close()

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

        with open(f"{filename}.txt", "w") as fp:
            fp.write(data)
        print(f"Dumped to '{filename}.txt'")

    @staticmethod
    def read_json(filename: str) -> Any:

        with open(f"{filename}.json") as fp:
            return json.loads(fp.read())

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

        with open(f"{filename}.json", "w") as fp:
            json.dump(data, fp)
        print(f"Dumped json to '{filename}.json'")

    @staticmethod
    def read_bin(filename: str) -> bytes:

        with open(f"{filename}.bin", "rb") as fp:
            return fp.read()

    @staticmethod
    def dump_bin(filename: str, data: bytes) -> None:

        with open(f"{filename}.bin", "wb") as fp:
            fp.write(data)
        print(f"Dumped binary to '{filename}.bin'")


class Formatter:

    hr: str = "------------------------------\n"
    table_header: str = f"{'Word':<12} | {'TF':<6} | {'DF':<6}\n{hr}"

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

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

    @staticmethod
    def format_stats(lex_stats: LexerStatistics, num_docs: int = 0) -> str:

        contents: str = ""
```

```python
        contents += f"{Formatter.hr}"
        contents += f"{num_docs} documents.\n"

        contents += f"{Formatter.hr}"
        contents += f"Collections size: {lex_stats.get_collection_size()}\n"
        contents += f"Vocabulary size: {lex_stats.get_vocab_size()}\n"
        contents += f"\n{Formatter.hr}"

        contents += "Top 100 most frequent words:\n"
        contents += Formatter.table_header
        for term in lex_stats.get_top_n_cf_df(100):
            contents += Formatter.__format_tf_df(*term)

        contents += f"\n{Formatter.hr}"
        contents += "500th word:\n"
        contents += Formatter.table_header
        contents += Formatter.__format_tf_df(*lex_stats.get_nth_freq_term(500))

        contents += f"\n{Formatter.hr}"
        contents += "1000th word:\n"
        contents += Formatter.table_header
        contents += Formatter.__format_tf_df(*lex_stats.get_nth_freq_term(1000))

        contents += f"\n{Formatter.hr}"
        contents += "5000th word:\n"
        contents += Formatter.table_header
        contents += Formatter.__format_tf_df(*lex_stats.get_nth_freq_term(5000))

        contents += f"\n{Formatter.hr}"
        single_occs: int = lex_stats.get_single_occs()
        contents += "Number of words that occur in exactly one document:\n"
        contents += f"{single_occs} ({round(single_occs / lex_stats.get_vocab_size
                                            () * 100, 2)}%)\n"

        return contents

    @staticmethod
    def format_term_doc_tf(term_doc_tf: list[tuple[str, int, int]]) -> str:

        contents: str = ""
        for line in term_doc_tf:
            contents += " ".join(str(i) for i in line) + "\n"

        return contents

    @staticmethod
    def format_rankings(
        all_rankings: dict[int, list[tuple[int, float]]]
    ) -> str:

        contents: str = ""
        for query_id, rankings in all_rankings.items():
            contents += (
                "\n".join(
                    f"{query_id} Q0 {doc_id} {rank + 1} {score:.6f} {JHED}"
                    for rank, (doc_id, score) in enumerate(rankings)
                )
                + "\n"
```

```python
            )

        return contents


class DataFile:
    def __init__(self, filename: Path) -> None:

        self.filename: Path = filename
        self.num_docs: int = 0
        self.doc_begin_tag: str = ""
        self.doc_end_tag: str = ""

    def _ingest(self, doc_id: int, prep: Normalizer, *args: Any) -> None:
        raise NotImplementedError

    def ingest(self, *args: Any) -> None:

        prep = Normalizer()
        doc_id: int = -1
        doc: str = ""

        with open(self.filename) as fp:
            for line in fp:

                if line:
                    if self.doc_begin_tag in line:
                        doc_id = int(line[6:-2])
                        self.num_docs += 1

                    elif self.doc_end_tag in line:

                        prep.set_document(doc)
                        prep.process()
                        self._ingest(doc_id, prep, *args)

                        doc = ""

                        if self.num_docs % DOC_PROC == 0:
                            print("Normalized", self.num_docs, "documents")

                    else:
                        doc += line

        print("Normalized", self.num_docs, "documents")


class QueryFile(DataFile):
    def __init__(self, filename: Path) -> None:

        super().__init__(filename)

        self.doc_begin_tag: str = "<Q ID="
        self.doc_end_tag: str = "</Q>"

    def _ingest(
        self, doc_id: int, prep: Normalizer, tokens: dict[int, generator[str]]
    ) -> None:
```

```python
        tokens[doc_id] = prep.get_tokens()


class CorpusFile(DataFile):
    def __init__(self, filename: Path) -> None:

        super().__init__(filename)

        self.doc_begin_tag: str = "<P ID="
        self.doc_end_tag: str = "</P>"

        self.df_file: str = f"outputs/stats/{filename.stem}_df"
        self.tf_file: str = f"outputs/stats/{filename.stem}_tf"
        self.len_file: str = f"outputs/stats/{filename.stem}_len"
        self.stats_file: str = f"outputs/stats/{filename.stem}_summary"
        self.ranking_file: str = f"outputs/stats/{filename.stem}_rank"
        self.dict_file: str = f"outputs/stats/{filename.stem}_dict"

        self.meta_file: str = f"outputs/tmp/{filename.stem}_ndocs"
        self.tdt_file: str = f"outputs/tmp/{filename.stem}_tdt"
        self.sorted_tdt_chunk_file: str = f"outputs/tmp/{filename.stem}_chunk_"
        self.sorted_tdt_file: str = f"outputs/tmp/{filename.stem}_sort"

        self.inv_file: str = f"outputs/bin/{filename.stem}_if"

    def _ingest(
        self,
        doc_id: int,
        prep: Normalizer,
        lex: Lexer,
        term_doc_tf: list[tuple[str, int, int]],
    ) -> None:

        # add processed tokens to the lexer
        lex.add(prep.get_tokens())

        # save records of term-DocID-tf
        term_doc_tf.extend(lex.get_term_docid_tf(doc_id))
```

Code Listing 6: ../ir/invertedfile.py

```python
from .const import IDX, CHUNK_SIZE
from .types import generator, counter, text_io


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

        self.dictionary: dict[str, list[int]] = {}
        self.inverted_file_raw: list[int] = []
        self.inverted_file_bytes: bytes

    def build_dict(self, df: counter) -> dict[str, list[int]]:

        for idx, term in enumerate(sorted(df.keys())):
            # term: [term index, offset, length, df]
            self.dictionary[term] = [idx, 0, 0, df[term]]

        return self.dictionary
```

```python
def __convert_tdt_types(
    self, col_split_list: list[str]
) -> tuple[int, int, int, str]:
    return (
        self.dictionary[col_split_list[IDX.TID]][IDX.TID],  # term ID
        int(col_split_list[IDX.INVF.DID]),  # doc ID
        int(col_split_list[IDX.INVF.TF]),  # term frequency,
        col_split_list[IDX.TID],  # term
    )

def __map_to_int(
    self, term_doc_tf: str
) -> generator[tuple[int, int, int, str]]:

    return (
        self.__convert_tdt_types(i.split(" "))
        for i in term_doc_tf.split("\n")[:-1]
    )

@staticmethod
def __format_mapped_tdt(chunk: list[tuple[int, int, int, str]]) -> str:

    return "\n".join(" ".join(map(str, s)) for s in chunk) + "\n"

def sort_mapped_tdt(self, term_doc_tf: str) -> generator[str]:

    mapped_values = self.__map_to_int(term_doc_tf)

    chunk: list[tuple[int, int, int, str]] = []
    cur_cs: int = 0

    for mapped_value in mapped_values:
        cur_cs += 1
        chunk.append(mapped_value)
        if cur_cs % CHUNK_SIZE == 0:
            chunk.sort()
            yield self.__format_mapped_tdt(chunk)
            chunk = []

    if chunk:
        chunk.sort()
        yield self.__format_mapped_tdt(chunk)

def parse_sorted_tdt(
    self, mapped_tdt_fp: text_io
) -> generator[tuple[int, int, int, str]]:

    with mapped_tdt_fp:
        for line in mapped_tdt_fp:
            if line:
                mapped_tdt = line[:-1].split(" ")
                yield (
                    int(mapped_tdt[IDX.TID]),
                    int(mapped_tdt[IDX.INVF.DID]),
                    int(mapped_tdt[IDX.INVF.TF]),
                    str(mapped_tdt[IDX.INVF.STR]),
                )
```

```python
    def convert (
        self , mapped_values : generator [ tuple [int , int , int , str ]]
    ) -> None :

        cur : int = -1
        offset : int = 0

        for val in mapped_values :

            term_str : str = val [IDX.INVF.STR]

            if cur != val [IDX.TID]:

                cur = val [IDX.TID]

                # update offset
                self . dictionary [ term_str ][ IDX.DICT.OF] = offset

                # update width between the current term and the next
                self . dictionary [ term_str ][ IDX.DICT.WID] = (
                    self . dictionary [ term_str ][ IDX.DICT.DF] * 2
                )

            offset += 2
            self . inverted_file_raw . extend ( val [IDX.INVF.DID : IDX.INVF.STR])
    def set_inverted_file_raw ( self , inverted_file_raw : list [int ]) -> None :

        self . inverted_file_raw = inverted_file_raw

    def get_inverted_file_raw ( self ) -> list [int ]:

        return self . inverted_file_raw

    def set_inverted_file_bytes ( self , inverted_file_bytes : bytes ) -> None :

        self . inverted_file_bytes = inverted_file_bytes

    def get_inverted_file_bytes ( self ) -> bytes :

        return self . inverted_file_bytes

    def set_dictionary ( self , dictionary : dict [str , list [int ]]) -> None :

        self . dictionary = dictionary

    def get_dictionary ( self ) -> dict [str , list [int ]]:

        return self . dictionary
```

Code Listing 7: ../ir/lexer.py

```python
from collections import Counter

from .types import generator , counter


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

11

```python
        self.cf: counter = Counter ()
        self.df: counter = Counter ()
        self.tf: counter = Counter ()
        self.dictionary: dict[str, list[int]] = {}

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

        # create a Counter for the document
        self.tf.clear()
        self.tf.update(tokens)

        # update the total term-frequency values with the Counter
        self.cf.update(self.tf)

        # increment the document-frequency values
        self.df.update(self.tf.keys())

    def get_term_docid_tf(self, doc_id: int) -> generator[tuple[str, int, int]]:

        for term in self.tf:
            yield term, doc_id, self.tf[term]

    def set_cf(self, cf: counter) -> None:

        self.cf = cf

    def set_df(self, df: counter) -> None:

        self.df = df

    def get_cf(self) -> counter:

        return self.cf

    def get_df(self) -> counter:

        return self.df


class LexerStatistics:
    def __init__(self, lex: Lexer) -> None:
        self.cf = lex.cf
        self.df = lex.df

    def get_collection_size(self) -> int:

        return self.cf.total()

    def get_vocab_size(self) -> int:

        return len(self.cf)

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

        top_n_cf = self.cf.most_common(n)
        for cf in top_n_cf:
            term, freq = cf
            yield term, freq, self.df[term]
```

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

        term, freq = self.cf.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 8: ../ir/normalize.py

```python
import re

from nltk import stem

from .types import generator

# fmt: off
STOPWORDS: set[str] = {
    # contractions
    "aren't", "ain't", "can't", "could've", "couldn't", "didn't", "doesn't", "don'
                                      t",
    "hadn't", "hasn't", "haven't", "he'd", "he'll", "he's", "i'd", "i'll",
    "i'm", "i've", "isn't", "it'll", "it'd", "it's", "let's", "mightn't",
    "might've'", "mustn't", "must've'", "shan't", "she'd", "she'll", "she's", "
                                      should've",
    "shouldn't", "that'll", "that's", "there's", "they'd", "they'll", "they're", "
                                      they've",
    "wasn't", "we'd", "we'll", "we're", "we've", "weren't", "what'll", "what're",
    "what's", "what've", "where's", "who'd", "who'll", "who're", "who's", "who've"
                                      ,
    "won't", "wouldn't", "would've", "y'all", "you'd", "you'll", "you're", "you've
                                      ",
    # NLTK stopwords
    "a", "all", "am", "an", "and", "any", "are", "as",
    "at", "be", "because", "been", "being", "but", "by", "can",
    "cannot", "could", "did", "do", "does", "doing", "for", "from",
    "had", "has", "have", "having", "he", "her", "here", "hers",
    "herself", "him", "himself", "his", "how", "i", "if", "in",
    "is", "it", "its", "itself", "just", "let", "may", "me",
    "might", "must", "my", "myself", "need", "no", "nor", "not",
    "now", "o", "of", "off", "on", "once", "only", "or",
    "our", "ours", "ourselves", "shall", "she", "should", "so", "some",
    "such", "than", "that", "the", "their", "theirs", "them", "themselves",
    "then", "there", "these", "they", "this", "those", "to", "too",
    "very", "was", "we", "were", "what", "when", "where", "which",
    "who", "whom", "why", "will", "with", "would", "you", "your",
    "yours", "yourself", "yourselves",
}
# fmt: on


class Normalizer:
```

```python
    def __init__(self) -> None:

        self.document: str = ""
        self.tokens: generator[str]

        self.ws_re: re.Pattern[str] = re.compile(r"([A-Za-z]+'?[A-Za-z]+)")
        self.snow: stem.SnowballStemmer = stem.SnowballStemmer("english")

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

        self.document = document

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

        return document.lower()

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

        return (x.group(0) for x in self.ws_re.finditer(document))

    def __remove_stopwords(self, tokens: generator[str]) -> generator[str]:
        return (word for word in tokens if word not in STOPWORDS)

    def __stem(self, tokens: generator[str]) -> generator[str]:

        return (self.snow.stem(token) for token in tokens)

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

        return self.tokens

    def process(self) -> None:

        # convert the entire document to lower-case
        doc_lc: str = self.__to_lower_case(self.document)

        # split the document on its whitespace
        self.tokens = self.__split_document(doc_lc)

        # remove contractions and stopwords
        self.tokens = self.__remove_stopwords(self.tokens)

        # stem tokens
        self.tokens = self.__stem(self.tokens)
```

Code Listing 9: ../ir/packer.py

```python
from struct import pack, unpack

from .const import BYTE_FMT_CHAR, BYTE_FMT_SIZE


class Packer:
    @staticmethod
    def encode(data: list[int]) -> bytes:

        return pack(BYTE_FMT_CHAR * len(data), *data)

    @staticmethod
```

```python
    def decode ( data : bytes ) -> tuple [ int , ...]:

        return unpack ( BYTE_FMT_CHAR * ( len ( data ) // BYTE_FMT_SIZE ) , data )
```

Code Listing 10: ../ir/retriever.py

```python
from math import log2 , sqrt
from multiprocessing import Pool

from .const import IDX , QUERY_DOC_ID , PARALLEL_THRESH
from .invertedfile import InvertedFile
from .packer import Packer
from .types import Any


Decoded_Inverted_File : tuple [ int , ...]


class Retriever :
    def __init__ ( self , invf : InvertedFile , num_docs : int ) -> None :

        self . invf = invf
        self . num_docs = num_docs

        self . partial_len : list [ float ] = [ 0.0 ] * ( self . num_docs + 1 )

        self . dictionary : dict [ str , list [ int ]] = invf . get_dictionary ()
        self . inverted_file : bytes = invf . get_inverted_file_bytes ()

        self . retrievals : dict [ str , dict [ str , Any ]] = {}
        self . query_tfidfs : dict [ str , float ] = {}
        self . doc_ids : set [ int ] = set ()
        self . num_doc_ids : int = 0

        self . metrics : list [ tuple [ int , float ] | None ] = []
        self . query_terms : list [ str ] = []

    def decode_inverted_file ( self ) -> None :

        global Decoded_Inverted_File
        Decoded_Inverted_File = Packer . decode ( self . inverted_file )

    def delete_inverted_file ( self ) -> None :

        global Decoded_Inverted_File
        Decoded_Inverted_File = tuple ()

    def retrieve ( self , term : str ) -> dict [ str , Any ]:

        invf_data : dict [ str , Any ] = { "term" : term }

        # term not in dictionary
        if term not in self . dictionary :
            return invf_data

        of : int = self . dictionary [ term ][ IDX . DICT . OF ]
        width : int = self . dictionary [ term ][ IDX . DICT . WID ]
        decoded_chunk : tuple [ int , ...] = Decoded_Inverted_File [
            of : of + width + 1
```

```python
        ]

        postings: tuple[int, ...] = decoded_chunk[:width:2]
        tf: tuple[int, ...] = decoded_chunk[1 : width + 1 : 2]
        df: int = len(postings)
        idf: float = log2(self.num_docs / df)
        tfidf: tuple[float, ...] = tuple(i * idf for i in tf)

        invf_data["postings"] = postings

        invf_data["idf"] = idf
        invf_data["tfidf"] = tfidf

        return invf_data

    def compute_sum_of_squares(self) -> None:

        for val in self.dictionary.values():
            of: int = val[IDX.DICT.OF]
            width: int = val[IDX.DICT.WID]
            decoded_chunk = Decoded_Inverted_File[of : of + width + 1]

            postings: tuple[int, ...] = decoded_chunk[:width:2]
            tfs: tuple[int, ...] = decoded_chunk[1 : width + 1 : 2]

            df: int = len(postings)
            idf: float = log2(self.num_docs / df)
            tfidfs: tuple[float, ...] = tuple(tf * idf for tf in tfs)

            for doc_id, tfidf in zip(postings, tfidfs):
                self.partial_len[doc_id] += tfidf * tfidf

    def compute_lengths(self) -> None:

        for doc_id in range(1, len(self.partial_len)):
            self.partial_len[doc_id] = sqrt(self.partial_len[doc_id])

    def get_lengths(self) -> list[float]:

        return self.partial_len

    def set_lengths(self, lengths: list[float]) -> None:

        self.partial_len = lengths

    @staticmethod
    def get_term_tfidf(
        doc_id: int, postings: tuple[int], tfidfs: tuple[int]
    ) -> float:

        if doc_id in postings:
            return tfidfs[postings.index(doc_id)]

        return 0

    def get_query_tfs(self) -> dict[str, float]:

        tfs: dict[str, float] = {}
        self.partial_len[QUERY_DOC_ID] = 0.0
```

16

```python
        for term in self.query_terms:
            tfs[term] = (
                self.query_terms.count(term) * self.retrievals[term]["idf"]
            )
            self.partial_len[QUERY_DOC_ID] += tfs[term] * tfs[term]

        self.partial_len[QUERY_DOC_ID] = sqrt(self.partial_len[QUERY_DOC_ID])
        return tfs

    def dot_product(self, document: dict[str, float]) -> float:

        return sum(
            self.query_tfidfs[term] * document[term]
            for term in self.query_tfidfs.keys()
        )

    def update_retrievals(self) -> None:

        self.retrievals = {
            term: self.retrieve(term) for term in self.query_terms
        }

    def get_retrievals(self) -> dict[str, dict[str, Any]]:

        return self.retrievals

    def get_metrics(self) -> list[tuple[int, float] | None]:
        return self.metrics

    def update_doc_ids(self) -> None:

        for retr in self.retrievals.values():
            for posting in retr["postings"]:
                if retr["idf"]:
                    self.doc_ids.add(posting)

    def get_document_weights(self, doc_id: int) -> dict[str, float]:

        tfidfs: dict[str, float] = {}
        for term in self.query_terms:
            if term not in tfidfs:
                tfidfs[term] = self.get_term_tfidf(
                    doc_id,
                    self.retrievals[term]["postings"],
                    self.retrievals[term]["tfidf"],
                )
        return tfidfs

    def generate_metrics(self) -> None:

        self.metrics = [None] * self.num_doc_ids
        cur: int = 0

        for doc_id, tfidfs in zip(
            self.doc_ids, map(self.get_document_weights, self.doc_ids)
        ):
            self.metrics[cur] = (doc_id, self.similarity(doc_id, tfidfs))
            cur += 1
```

```python
    def generate_metrics_p(self) -> None:

        self.metrics = [None] * self.num_doc_ids
        cur = 0

        with Pool() as executor:
            for doc_id, tfidfs in zip(
                self.doc_ids,
                executor.map(self.get_document_weights, self.doc_ids),
            ):
                self.metrics[cur] = (doc_id, self.similarity(doc_id, tfidfs))
                cur += 1

    def similarity(self, doc_id: int, doc_weights: dict[str, float]) -> float:

        dot: float = self.dot_product(doc_weights)
        doc_len: float = self.partial_len[doc_id]
        return dot / (doc_len * self.partial_len[QUERY_DOC_ID])

    def reset(self) -> None:

        self.doc_ids.clear()
        self.retrievals.clear()
        self.metrics.clear()
        self.query_tfidfs.clear()

    def query(self, query_terms: list[str]) -> None:

        print(f"Querying '{query_terms}'...")
        self.query_terms = query_terms

        # initialize retrievals with terms from query
        self.update_retrievals()

        # initialize set of all documents with at least one query term
        self.update_doc_ids()
        self.num_doc_ids = len(self.doc_ids)
        print(f"Found {self.num_doc_ids} relevant documents...")

        self.query_tfidfs = self.get_query_tfs()

        # generate metrics of all the retrieved documents
        if len(self.doc_ids) > PARALLEL_THRESH:
            print("Generating metrics in parallel...")
            self.generate_metrics_p()
        else:
            print("Generating metrics...")
            self.generate_metrics()

    def get_rankings(self, top_n: int = 100) -> list[tuple[int, float] | None]:

        return sorted(self.metrics, key=lambda x: x[1], reverse=True)[:top_n]
```

Code Listing 11: ../ir/types.py

```python
from typing import Any, Counter, Generator, Iterable, TextIO, TypeAlias, TypeVar


iterable: TypeAlias = Iterable[Any]
```

```python
counter: TypeAlias = Counter[str]

text_io: TypeAlias = TextIO

T = TypeVar("T")
generator: TypeAlias = Generator[T, None, None]
```

Code Listing 12: ../run.py

```python
import argparse
from pathlib import Path

from ir import InformationRetrieval

if __name__ == "__main__":

    parser = argparse.ArgumentParser()
    parser.add_argument("path", type=str, help="path of corpus file")
    parser.add_argument(
        "-a",
        "--all",
        action=argparse.BooleanOptionalAction,
        help="run through the entire list of default pipeline tasks",
    )
    parser.add_argument(
        "-f",
        "--freq",
        action=argparse.BooleanOptionalAction,
        help="generate frequencies",
    )
    parser.add_argument(
        "-s",
        "--stat",
        action=argparse.BooleanOptionalAction,
        help="generate frequency statistics",
    )
    parser.add_argument(
        "-d",
        "--dump",
        action=argparse.BooleanOptionalAction,
        help="save frequencies data to file",
    )
    parser.add_argument(
        "-l",
        "--load",
        action=argparse.BooleanOptionalAction,
        help="load pre-generated frequencies data",
    )
    parser.add_argument(
        "-t",
        "--sort",
        action=argparse.BooleanOptionalAction,
        help="sort term-docID-tf",
    )
    parser.add_argument(
        "-e",
        "--encode",
        action=argparse.BooleanOptionalAction,
```

```python
        help="generate inverted file",
    )
    parser.add_argument(
        "-p",
        "--precompute",
        action=argparse.BooleanOptionalAction,
        help="precompute document lengths",
    )
    parser.add_argument("-q", "--query", type=str, help="path of query file")

    args = vars(parser.parse_args())

    ir_obj = InformationRetrieval(Path(args["path"]))

    if args["all"]:
        ir_obj.generate_freqs()
        ir_obj.generate_stats()
        ir_obj.dump_freqs()
        ir_obj.build_sorted_tdt()
        ir_obj.encode_inverted_file()
        ir_obj.precompute_lengths()

    elif args["query"]:
        ir_obj.generate_rankings(args["query"])

    else:

        if args["freq"]:
            ir_obj.generate_freqs()

        if args["stat"]:
            ir_obj.generate_stats()

        if args["dump"]:
            ir_obj.dump_freqs()

        if args["load"]:
            ir_obj.load_freqs()

        if args["sort"]:
            ir_obj.build_sorted_tdt()

        if args["encode"]:
            ir_obj.encode_inverted_file()

        if args["precompute"]:
            ir_obj.precompute_lengths()
```

B Outputs

Code Listing 13: Statistics of 'cord19.txt'

```
191175 documents.
```

```
Collections size: 33210573
Vocabulary size: 235068
```

```
Top 100 most frequent words:
Word          | TF      | DF

patient       | 267998  | 63993
covid         | 252298  | 74596
infect        | 246033  | 64655
virus         | 232738  | 49802
use           | 214074  | 77178
diseas        | 195507  | 68034
studi         | 187927  | 71649
et            | 153283  | 16190
al            | 150622  | 16209
cell          | 145742  | 30101
sar           | 137020  | 36140
cov           | 130447  | 31096
result        | 123228  | 72512
case          | 120655  | 45202
sever         | 120300  | 55787
coronavirus   | 116961  | 49820
protein       | 115340  | 23943
health        | 111969  | 41252
respiratori   | 108997  | 40993
effect        | 100773  | 46708
viral         | 98581   | 31984
clinic        | 96950   | 44802
human         | 96673   | 33935
includ        | 95935   | 53930
model         | 94548   | 26192
develop       | 91223   | 44230
dure          | 89595   | 48495
also          | 87922   | 47992
report        | 87485   | 44429
high          | 86951   | 45665
time          | 86286   | 43141
other         | 85219   | 47220
pandem        | 84575   | 39876
differ        | 84259   | 42098
method        | 84123   | 49344
data          | 83378   | 37939
associ        | 82262   | 40387
between       | 81910   | 44378
system        | 81817   | 35506
increas       | 79974   | 42047
more          | 79195   | 44747
respons       | 78578   | 34210
activ         | 78488   | 28638
base          | 77826   | 39194
treatment     | 75067   | 34318
risk          | 74859   | 31683
group         | 72682   | 26838
caus          | 72630   | 39896
most          | 69956   | 41638
```

test		69622		28999
control		68199		33363
one		66955		38917
signific		66423		39374
care		65341		27849
two		65164		37088
vaccin		65144		15516
relat		65058		37294
detect		64859		25018
acut		64758		35606
present		64516		39540
provid		62845		38108
after		62302		34266
specif		61915		32569
howev		60715		40308
import		60669		37040
hospit		59890		24929
identifi		59546		34524
rate		58666		28326
number		58115		30099
immun		57606		19052
new		57337		32469
analysi		57058		31677
year		55798		30865
emerg		55491		29429
level		55141		27560
compar		54938		32370
outbreak		54894		25179
review		53990		28865
potenti		53623		33340
show		53132		33893
both		52821		34003
rna		52761		14564
first		52546		33450
influenza		51928		13365
into		51747		31728
measur		51608		26121
factor		51581		25311
syndrom		50655		29203
day		49488		22113
popul		49475		23125
well		49403		32778
function		49129		22275
pathogen		49122		19835
posit		48410		25882
structur		48243		18592
express		47848		16451
follow		47746		30428
perform		47426		27677
type		47384		22660
gene		47318		14183

500th word:

Word	TF	DF
morbid	13447	9512

1000th word:

Word	TF	DF
fold	6588	3904

5000th word:

Word	TF	DF
tat	510	209

Number of words that occur in exactly one document:
126829 (53.95%)