605.744: Information Retrieval
Programming Assignment #5: Near Duplicate Detection

Sabbir Ahmed

November 13, 2022

Contents

1 Introduction

This paper describes detecting near-duplications (plagiarism) within documents in datasets of various sample sizes.

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.

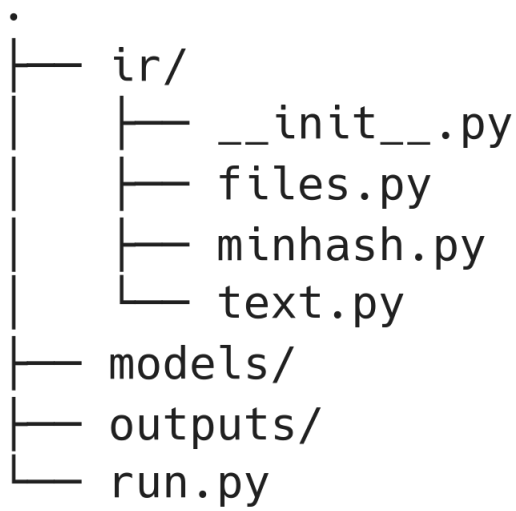

Figure 1: Directory Hierarchy of Assignment 5

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 285.

2.1 Classes

Some classes from Assignment 3 were used in this project:

- the driver script `run.py` was modified with the relevant flags
- the `files.CorpusFile` class was modified to process TSV files and read and write clusters to disk
- the `text.Normalizer` class was used to normalize documents into lists of stemmed words

2.2 `text.Shingle`

The `text.Shingle` class was added to create shingles of n-grams of the documents. Each of the n-grams were stored as unique integers, $-2^{64} - 1 \leq x \leq 2^{64} - 1$, generated by Python's built-in `hash()` function.

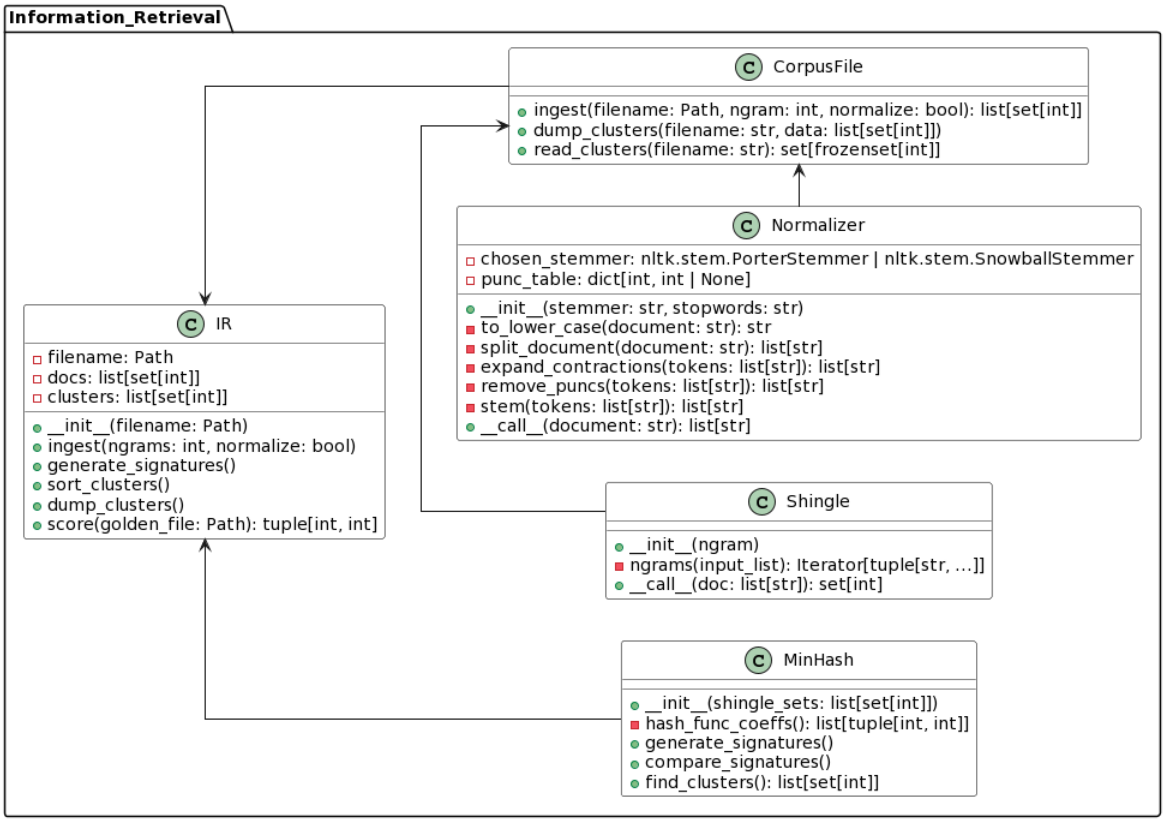

Figure 2: UML of Information Retrieval

2.3 `minhash.MinHash`

The `minhash.MinHash` class was added to generate the signatures from the shingles and determine clusters of near-duplicate documents. The algorithm for generating and comparing the min-hash values was based on the column-row permutations described in *Mining Massive Datasets* [1]. A min hash, Φ_s, is generated by hash functions of the form $h(x) = (a \times x + b)\ mod\ p$, where p is prime and the random coefficients, $\{a, b \in \mathbb{N} : 0 \leq a, b \leq 2^{32} - 1\}$. For this function, p is chosen to be $4,294,967,295$, the first prime number larger than $2^{32} - 1$.

The signatures are stored in a $N \times N$ matrix.

2.4 External Libraries

The following external libraries were used to implement portions of the assignment:

- Natural Language Toolkit (NLTK) [2]
- NetworkX [3]

Table 1: Distribution of *Assessment* in the training dataset

N-Gram	Normalized	$G - O$	$O - G$
1	True	2	1
1	False	2	2
2	True	1	2
2	False	3	2
3	True	2	4
3	False	2	4
4	True	3	6
4	False	3	6
5	True	3	6
5	False	4	9
6	True	10	12
6	False	10	12

3 Scoring

References

[1] J. Leskovec, A. Rajaraman, and J. D. Ullman, *3.3.5 Computing Minhash Signatures*, pp. 83–86. Cambridge University Press, 2022.

[2] S. Bird, E. Klein, and E. Loper, *Natural language processing with Python: analyzing text with the natural language toolkit*. " O'Reilly Media, Inc.", 2009.

[3] A. A. Hagberg, D. A. Schult, and P. J. Swart, "Exploring network structure, dynamics, and function using networkx," in *Proceedings of the 7th Python in Science Conference* (G. Varoquaux, T. Vaught, and J. Millman, eds.), (Pasadena, CA USA), pp. 11–15, 2008.

A Source Code

Code Listing 1: ir/__init__.py

```
from pathlib import Path

from .files import CorpusFile
from .minhash import MinHash


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

        self.filename: Path = filename
        self.docs: list[set[int]] = []
        self.clusters: list[set[int]] = []

    def ingest(self, ngrams: int, normalize: bool) -> None:

        print(
            f"Shingling documents with {ngrams}-grams and normalization={normalize
                                                                        }..."
```

```
        )
        self.docs = CorpusFile().ingest(self.filename, ngrams, normalize)

    def generate_signatures(self) -> None:

        min_hash = MinHash(self.docs)
        print("Generating signatures...")
        min_hash.generate_signatures()

        print("Comparing signatures...")
        min_hash.compare_signatures()

        print("Finding clusters...")
        self.clusters = min_hash.find_clusters()

    def sort_clusters(self) -> None:
        self.clusters.sort(key=len, reverse=True)

    def dump_clusters(self) -> None:

        self.sort_clusters()

        CorpusFile().dump_clusters(self.filename.stem, self.clusters)

    def score(self, golden_file: Path) -> tuple[int, int]:

        golden_data = CorpusFile().read_clusters(golden_file.stem)
        output = CorpusFile().read_clusters(self.filename.stem)

        gold_diff_out = golden_data.difference(output)
        out_diff_gold = output.difference(golden_data)

        return (len(gold_diff_out), len(out_diff_gold))
```

Code Listing 2: ir/files.py

```
from pathlib import Path

from .text import Normalizer, Shingle

JHED = "sahmed80"


class CorpusFile:
    @staticmethod
    def ingest(
        filename: Path, ngram: int = 3, normalize: bool = True
    ) -> list[set[int]]:

        shingle = Shingle(ngram)
        norm = Normalizer()
        docs: list[set[int]] = []

        with open(filename) as fp:
            for line in fp:
                if line:
                    _, doc = line[:-1].split("\t")
                    if normalize:
                        doc = norm(doc)
```

```
                    else:
                        doc = doc.split(" ")
                    doc_shingles = shingle(doc)
                    docs.append(doc_shingles)

        return docs

    @staticmethod
    def dump_clusters(filename: str, data: list[set[int]]) -> None:

        filepath = f"outputs/{JHED}-{filename}.txt"
        with open(filepath, "w") as fp:
            output = ""
            for c in data:
                output += " ".join(str(i) for i in c) + "\n"
            fp.write(output)
        print(f"Dumped to '{filepath}'")

    @staticmethod
    def read_clusters(filename: str) -> set[frozenset[int]]:

        data: set[frozenset[int]] = set()
        filepath = f"outputs/{JHED}-{filename}.txt"
        with open(filepath) as fp:
            cluster = frozenset[int]
            for line in fp:
                if line:
                    cluster = line[:-1].split(" ")
                    cluster = frozenset(int(i) for i in cluster)
                    data.add(cluster)

        return data
```

Code Listing 3: ir/minhash.py

```
import random

import networkx as nx

NEXT_PRIME = 4294967311
MAX_VAL = 4294967295


class MinHash:
    def __init__(self, shingle_sets: list[set[int]]) -> None:

        random.seed(0)
        self.shingle_sets = shingle_sets
        self.n_docs = len(shingle_sets)
        self.n_hashes = 200
        self.threshold = 0.5 * self.n_hashes

        self.signatures: list[list[int]] = []
        self.sim: list[list[int]] = [
            [0] * self.n_docs for _ in range(self.n_docs)
        ]

    def __hash_func_coeffs(self) -> list[tuple[int, int]]:
```

```
        return [
            (
                random.randint(0, MAX_VAL),
                random.randint(0, MAX_VAL),
            )
            for _ in range(self.n_hashes)
        ]

    def generate_signatures(self) -> None:

        hash_func_coeffs = self.__hash_func_coeffs()

        for shingle_set in self.shingle_sets:
            signature: list[int] = []

            for a, b in hash_func_coeffs:
                phi_s = NEXT_PRIME + 1
                for shingle in shingle_set:
                    hash_code = (a * shingle + b) % NEXT_PRIME

                    if hash_code < phi_s:
                        phi_s = hash_code

                signature.append(phi_s)

            self.signatures.append(signature)

        self.shingle_sets.clear()

    def compare_signatures(self) -> None:

        for first in range(self.n_docs):

            if not first % 1000:
                print(f"\t{first}/{self.n_docs}")

            for second in range(first + 1, self.n_docs):

                self.sim[first][second] = sum(
                    self.signatures[first][k] == self.signatures[second][k]
                    for k in range(self.n_hashes)
                )

        print(f"\t{self.n_docs}/{self.n_docs}")
        self.signatures.clear()

    def find_clusters(self) -> list[set[int]]:

        clusters = nx.utils.UnionFind(list(range(1, self.n_docs + 1)))

        for first in range(self.n_docs):
            for second in range(first + 1, self.n_docs):
                sim = self.sim[first][second]
                if sim > self.threshold:
                    clusters.union(first + 1, second + 1)

        return list(clusters.to_sets())
```

Code Listing 4: ir/text.py

```
from typing import Iterator

import nltk

# fmt: off
CONTRACTIONS = {
    "aren't": "are not", "ain't": "is not", "can't": "can not",
    "could've": "could have", "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 would",
    "he'll": "he will", "he's": "he is", "i'd": "i would",
    "i'll": "i will", "i'm": "i am", "i've": "i have", "isn't": "is not",
    "it'll": "it will", "it'd": "it would", "it's": "it is", "let's": "let us",
    "mightn't": "might not", "might've'": "might have", "mustn't": "must not",
    "must've'": "must have", "shan't": "sha not", "she'd": "she would",
    "she'll": "she will", "she's": "she is", "should've": "should have",
    "shouldn't": "should not", "that'll": "that will", "that's": "that is",
    "there's": "there is", "they'd": "they would", "they'll": "they will",
    "they're": "they are", "they've": "they have", "wasn't": "was not",
    "we'd": "we would", "we'll": "we will", "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 would", "who'll": "who will",
    "who're": "who are", "who's": "who is", "who've": "who have",
    "won't": "wo not", "wouldn't": "would not", "would've": "would have",
    "y'all": "you all", "you'd": "you would", "you'll": "you will",
    "you're": "you are", "you've": "you have",
}
# fmt: on


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

        self.stem = nltk.stem.SnowballStemmer("english")
        self.punc_table: dict[int, int | None] = str.maketrans(
            "", "", """!"#$%&'()*+,-./:;<=>?@[\\]^_`{|}~"""
        )

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

        return document.lower()

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

        return document.split(" ")

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

        temp_tokens: list[str] = []
        for token in tokens:
            if token in CONTRACTIONS:
                temp_tokens.extend(CONTRACTIONS[token].split(" "))
            else:
                temp_tokens.append(token)

        return temp_tokens
```

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

        return [i.translate(self.punc_table) for i in tokens]

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

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

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

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

        # split the document on its whitespace
        tokens: list[str] = self.__split_document(doc_lc)

        # expand contraction words
        tokens = self.__expand_contractions(tokens)

        # remove punctuations
        tokens = self.__remove_puncs(tokens)

        # stem tokens
        tokens = self.__stem(tokens)

        return tokens


class Shingle:
    def __init__(self, ngram) -> None:
        self.ngram = ngram

    def __ngrams(self, input_list) -> Iterator[tuple[str, ...]]:
        return zip(*(input_list[i:] for i in range(self.ngram)))

    def __call__(self, doc: list[str]) -> set[int]:

        shingles: set[int] = set()
        for shingle in self.__ngrams(doc):

            hashed_shingle = hash(" ".join(shingle))
            shingles.add(hashed_shingle)

        return shingles
```

Code Listing 5: run.py

```
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="perform all operations",
    )
    parser.add_argument(
        "-r",
        "--read",
        type=int,
        nargs="?",
        default=None,
        const=3,
        help="read and ingest file",
    )
    parser.add_argument(
        "-n",
        "--norm",
        default=False,
        action="store_true",
        help="normalize documents",
    )
    parser.add_argument(
        "-g",
        "--gen",
        action=argparse.BooleanOptionalAction,
        help="perform min-hashing",
    )
    parser.add_argument(
        "-d",
        "--dump",
        action=argparse.BooleanOptionalAction,
        help="dump clusters to disk",
    )
    parser.add_argument(
        "-s", "--score", type=str, help="score clusters against golden file"
    )
    parser.add_argument(
        "-t",
        "--test",
        action=argparse.BooleanOptionalAction,
        help="perform all operations",
    )

    args = vars(parser.parse_args())

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

    if args["all"]:
        ir_obj.ingest(ngrams=3, normalize=True)
        ir_obj.generate_signatures()
        ir_obj.dump_clusters()

    elif args["test"]:

        scores: list[tuple[int, bool, tuple[int, int]]] = []
        for ngram in range(1, 7):
            for normalize in (True, False):
                ir_obj.ingest(ngrams=ngram, normalize=normalize)
                ir_obj.generate_signatures()
                ir_obj.dump_clusters()
```

```
                scores.append(
                    (ngram, normalize, ir_obj.score(Path(args["score"])))
                )

        print(scores)

    else:

        if args["read"]:
            ir_obj.ingest(ngrams=args["read"], normalize=args["norm"])

        if args["gen"]:
            ir_obj.generate_signatures()

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

        if args["score"]:
            print(ir_obj.score(Path(args["score"])))
```