Report 1: Similarity Detection

Group 150: Lorenzo Deflorian, Riccardo Fragale

November 9, 2025

1 Data

We decided to employ a dataset taken from Kaggle (`https://www.kaggle.com/datasets/shubchat/1002-short-stories-from-project-guttenberg`). It contains a a set of short stories extracted from the wonderful portal of Project Guttenberg. They are very well known short stories from famoous writers in history. In order to extract this files we employed a script where we use kagglehub API to download the files and store them inside the repository for this lab. For the sake of our project, we decided to analyze and compare texts from a single author as there is a higher chance that documents are similar. In particular, every time the test is running we save all the data scraped from the dataset looking only for the author we decide to specify. We collect all the data regarding title of book, book number and the content of the short story.

2 Methods

We split the procedures for finding similar items into separate modules:

- **Shingling.py**
- **Minhashing.py**
- **Compare_Sets.py**
- **Compare_Signatures.py**
- **LSH.py**

Tests are in the **test** folder, and **main.py** runs the experiments for Dickens and Edgar Allan Poe. For each step we provide a plain implementation and a Spark-based variant, to compare any efficiency gains against the overhead of connecting to a Spark cluster and loading data.

The pipeline starts by converting each document into a set of shingles and hashing them to avoid duplicates. Shingle size is important; we use a size

of 9 for large files and 5 for smaller texts. Documents are preprocessed by collapsing consecutive whitespace to a single space, converting to lowercase, and removing punctuation, since punctuation rarely contributes to meaningful text similarity. Shingling splits the text into substrings (shingles), which we hash with mmh3 and mask to 32 bits. To ensure reproducible results across machines, we fix the mmh3 seed. The hashed shingles are collected into a list of integers (and we have one list for each document).

Second step of the pipeline, and also very important is MinHashing. Through minhashing we converted large set of shingles into short signatures preserving similarity. The key theoretical idea is the following: Take k = 100 independent hash functions, then apply the functions to the elements (the row numbers with value 1) to compute a vector of k minHash values for a set S; the resulting column-vector is a signature of the set. We decided to generate a family of 100 hash functions in mathematical form:

$$h_i(x) = (a_i x + b_i) \bmod p,$$

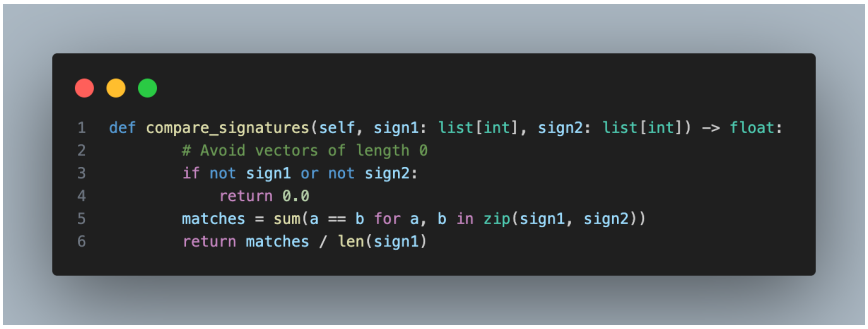
where p is a prime larger than the maximum shingle identifier, and each $a_i \in \{1, \ldots, p-1\}$, $b_i \in \{0, \ldots, p-1\}$ is sampled uniformly at random. Then the following is the function used to compute the signatures for the set of shingles

```python
def get_signature(self, shingles: list[int]) -> list[int]:
        signature = []

        if len(shingles) > SHINGLE_PYSPARK_THRESHOLD:
            self.logger.info(
                f"Using pyspark to compute the signature for {len(shingles)} shingles"
            )
            rdd = self.spark.sparkContext.parallelize(shingles)
            for hash_fn in self.hash_functions:
                min_hash = rdd.map(hash_fn).min()
                signature.append(min_hash)
        else:
            for hash_fn in self.hash_functions:
                min_hash = min(shingles, key=hash_fn)
                signature.append(min_hash)

        return signature
```

Figure 1: Visualization of MinHash signatures for the documents.

As it can be seen the procedure is quite straightforward in terms of implementation.

Highly connected to the signatures is the **CompareSignatures** class which be very useful later on. It is calculated as the the fraction of the minHash functions in which they agree. Actually it's an estimate of the Jaccard similarity between the two sets of signatures.



```
def compare_signatures(self, sign1: list[int], sign2: list[int]) -> float:
        # Avoid vectors of length 0
        if not sign1 or not sign2:
            return 0.0
        matches = sum(a == b for a, b in zip(sign1, sign2))
        return matches / len(sign1)
```

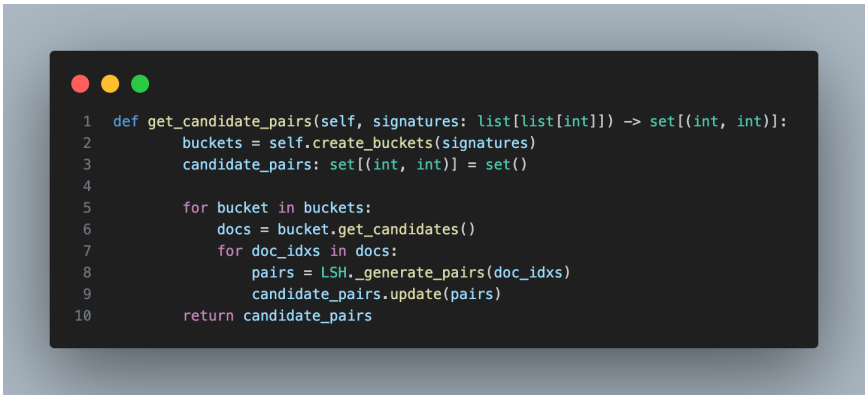
Figure 2: Compute the comparison between signatures.

Now, coming back to the pipeline the final step is Locally Sensitive Hashing, which is necessary to restrict our analysis only to the pairs of signatures where we have a high probability that there is a similarity. On a complexity point of view checking and comparing all signatures, especially for longer texts would be really too long. Two key parameters for this procedure are the number of bands and the number of rows per band. We decided to keep the product b * r = n = 100. We also saw that changing a bit the number of rows per band changes a bit the result obtained. In particular, the higher is the number of rows per band, the lower is the similarity between the signatures of the different texts. This is in our opinion reasonable as increasing the number of rows per band implies that a longer portion of the text of the short story should be almost identical to another one. This is less probable as the authors tend to use similar phrase structures but they do not entirely copy a list of words inside their texts.

3 Results

As I anticipated before, we decided to focus on the short stories by Charles Dickens and Edgar Allan Poe. In both cases we have a reasonable number of contents; 6 stories for the first and seven for the latter. Since their writing style is different, and also the contents and the themes of their short stories, we decided to compare the writing of a single author to the ones of the same writer. We expected to find good similarity as generally authors tend to use similar words and adopt a certain syntactical set of structures to build phrases.

This picture below shows the result for the Dickens short stories regarding the case with 4 rows per band.



```python
def get_candidate_pairs(self, signatures: list[list[int]]) -> set[(int, int)]:
        buckets = self.create_buckets(signatures)
        candidate_pairs: set[(int, int)] = set()

        for bucket in buckets:
            docs = bucket.get_candidates()
            for doc_idxs in docs:
                pairs = LSH._generate_pairs(doc_idxs)
                candidate_pairs.update(pairs)
        return candidate_pairs
```

Figure 3: result of Dickens test

Translating the expression couple (3,4) into the title of the short story we found out that *Reprinted Pieces* and *Some Christmas Stories* have a similarity rate = 0.29 which is not very very high but still inside our expectations.

We decided to add logging messages regarding the length of all the steps of our pipeline. We found out that the longest parts are shingling and minhashing while LSH and a the final comparison of the signatures are quite fast. This implies that it is crucial to reduce as much as possible the complexity of the first two procedures (also by using Spark) while the other two are less time consuming and can be done directly on our machine without parallelizing.

4 Conclusions

This homework let us correctly understand all the steps needed to correctly identify similarities inside similar texts. We also realized that they are very important tools as they can be used to identify possible plagiarism which is a key aspect in the literature markets. Moreover, they can be employed to find stylical aspects of authors (such as we in part did) since we could now how much parts of a text are always used by a writer inside their operas.