Report 1: Similarity Detection

Group 150: Lorenzo Deflorian, Riccardo Fragale

November 10, 2025

1 Data

We decided to use a dataset from Kaggle (`https://www.kaggle.com/datasets/shubchat/1002-short-stories-from-project-guttenberg`). It contains a collection of short stories extracted from the well-known Project Gutenberg portal, featuring works from famous authors in literary history.

To extract the files, we used a script that employs the KaggleHub API to download and store them in the repository for this lab. For the sake of our project, we decided to analyze and compare texts from a single author, as this increases the likelihood of finding similarities between documents.

Each time the test is executed, we collect all data related to the specified author, including the book title, book number, and the full content of each short story.

2 Methods

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

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

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

The pipeline begins by converting each document into a set of shingles and hashing them to avoid duplicates. Shingle size plays an important role;

we use a size of 9 for large files and 5 for smaller texts. Before shingling, we preprocess the documents by collapsing consecutive whitespace into a single space, converting text to lowercase, and removing punctuation, since punctuation rarely contributes to meaningful text similarity.

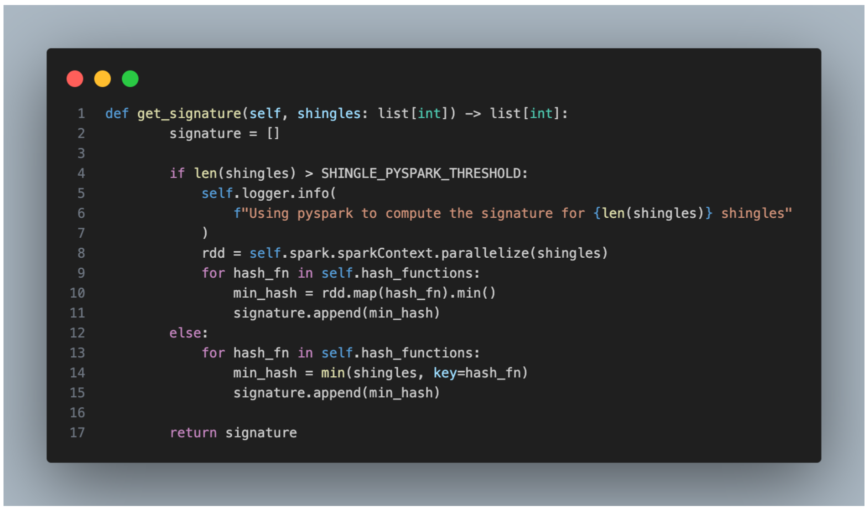
Shingling splits the text into substrings (shingles), which we hash using `mmh3` and mask to 32 bits. To ensure reproducible results across machines, we fix the `mmh3` seed. The hashed shingles are then collected into a list of integers, one for each document.

The second step in the pipeline, and a very important one, is MinHashing. Through MinHashing, we convert large sets of shingles into short signatures while preserving similarity. The key theoretical idea is as follows: take $k = 100$ independent hash functions and apply them to the elements (row numbers with value 1) to compute a vector of k minHash values for a set S. The resulting column vector is the signature of the set.

We generate a family of 100 hash functions defined as:

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

where p is a prime number 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. The following function is used to compute the signatures for each set of shingles:

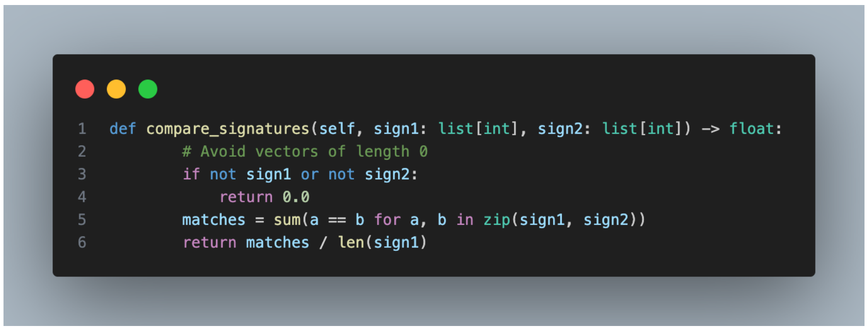


```python
def get_signature(self, shingles: list[int]) -> list[int]:
        signature = []

        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 shown above, the procedure is quite straightforward to implement.

Closely related to the signatures is the **CompareSignatures** class, which becomes very useful later on. It calculates the fraction of the minHash functions in which two signatures agree, in other words, an estimate of the Jaccard similarity between the two sets of signatures.



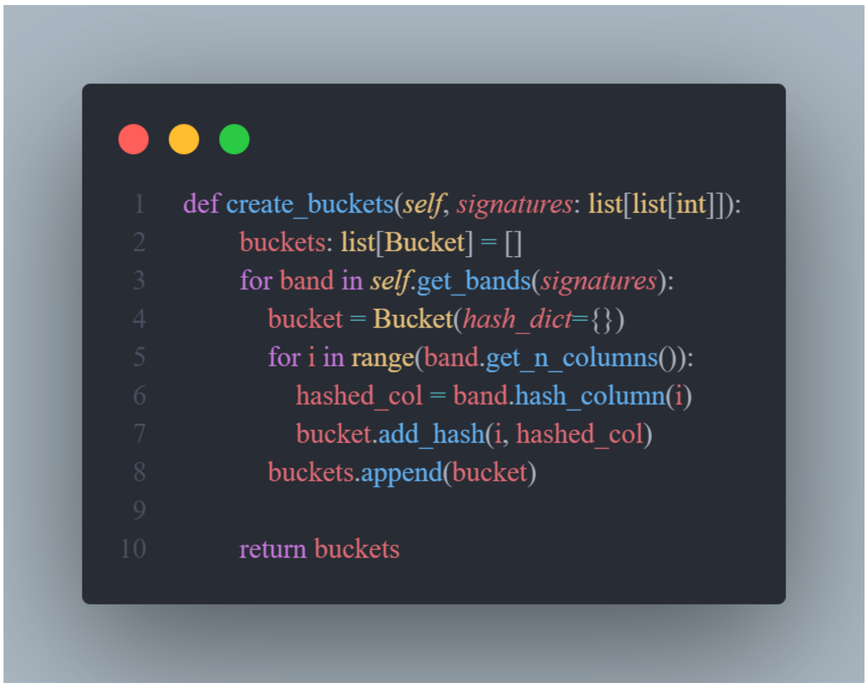
```python
def compare_signatures(self, sign1: list[int], sign2: list[int]) -> float:
        # Avoid vectors of length 0
        if not sign1 or not sign2:
            return 0.0
        matches = sum(a == b for a, b in zip(sign1, sign2))
        return matches / len(sign1)
```

Figure 2: Computation of the comparison between signatures.

The final step in our pipeline is Locality-Sensitive Hashing (LSH), which allows us to restrict our analysis to pairs of signatures that are likely to be similar. From a computational perspective, comparing all signatures (especially for long texts) would be prohibitively time-consuming.

The idea behind LSH is to split the signature matrix into b bands of r rows each (so that $b \times r = n$, where n is the number of hash functions). The choice of b and r is crucial, as it influences the probability of finding similar documents. In particular, using fewer rows per band (r) increases the likelihood of identifying similar documents by allowing more flexibility in matching parts of the signatures. In our case, we experimented with different values of r and found that using $r = 4$ or $r = 3$ provides a good balance between accuracy and efficiency.

This result makes sense for our task, as we are analyzing short stories (texts that are not extremely long) and authors tend to maintain a consistent writing style without copying large sections verbatim. Processing each band separately, column by column, we hash the portion of the signature corresponding to that band and we store it in a bucket. (Note that each band uses a different bucket). If different documents hash to the same value in the same band, we consider them as candidate pairs for similarity comparison.



```python
def create_buckets(self, signatures: list[list[int]]):
    buckets: list[Bucket] = []
    for band in self.get_bands(signatures):
        bucket = Bucket(hash_dict={})
        for i in range(band.get_n_columns()):
            hashed_col = band.hash_column(i)
            bucket.add_hash(i, hashed_col)
        buckets.append(bucket)

    return buckets
```

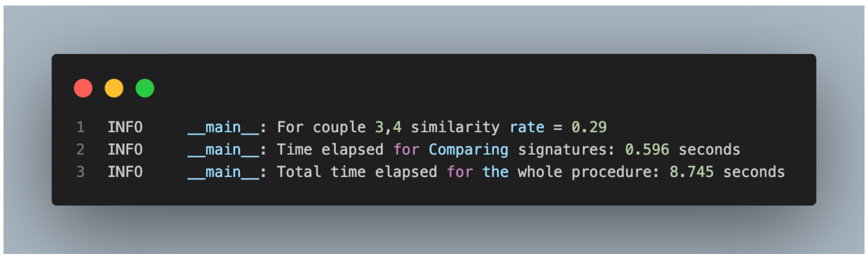
Figure 3: Bucket creation using LSH

3 Results

As mentioned earlier, we focused on short stories by Charles Dickens and Edgar Allan Poe. In both cases, we obtained a reasonable number of samples: six stories for Dickens and seven for Poe. Since their writing styles and themes differ significantly, we decided to compare each author's stories only against other works by the same author.

We expected to observe some degree of similarity, as authors typically reuse similar words and syntactic structures across their works.

The figure below shows the results for Dickens' short stories using four rows per band:



```
INFO     __main__: For couple 3,4 similarity rate = 0.29
INFO     __main__: Time elapsed for Comparing signatures: 0.596 seconds
INFO     __main__: Total time elapsed for the whole procedure: 8.745 seconds
```

Figure 4: Results for the Dickens test.

Translating the pair (3,4) into the corresponding story titles, we found that *Reprinted Pieces* and *Some Christmas Stories* have a similarity rate of 0.29. While not particularly high, this result aligns with our expectations.

We also added logging messages to measure the execution time of each step in our pipeline. We observed that the longest stages are shingling and MinHashing, while LSH and the final signature comparison are relatively fast. This implies that optimizing the first two steps (possibly by leveraging Spark) can significantly improve performance, while the remaining ones can be efficiently executed locally without parallelization.

Depending on the test configuration, the total computation time ranges from 8 to 15 seconds. Given these small-scale experiments, we believe that using PySpark is not strictly necessary, since the analyzed texts are relatively short. However, in a real-world application involving a much larger dataset, Spark would become essential to speed up the entire process.

4 Conclusions

This assignment helped us thoroughly understand the steps required to identify similarities between texts. We also realized how important these techniques are, as they can be used to detect plagiarism or duplicate web pages, both key aspects in literature and information retrieval.

Moreover, they can reveal stylistic features of authors (as we partially did), such as how frequently certain textual patterns appear in their works. Our next step would be to apply the developed strategy to a much larger dataset in order to evaluate the scalability and performance of our approach.