Report 1: Finding Similar Items: Textually Similar Documents

Group 34: Sofia Krylova, Iosif Koen

November 13, 2022

1 Introduction

A fundamental data-mining problem is to examine data for "similar" items. The naive approach to finding pairs of similar items requires us to look at every pair of items. When we are dealing with a large dataset, looking at all pairs of items may be prohibitive, even given an abundance of hardware resources.

Luckily, there is a family of techniques called locality-sensitive hashing, or LSH, that allows us to focus on pairs that are likely to be similar, without having to look at all pairs. Thus, we can avoid the quadratic growth in computation time but encountering false negative pairs.

This, the aim of this home assignment is to implement the stages of finding textually similar documents based on Jaccard similarity using the shingling, minhashing, and locality-sensitive hashing (LSH) techniques and corresponding algorithms.

We did not use any dedicated data frameworks. The task is implemented in Python.

2 Data source

We used 2 sets of data for the testing purposes: Christmas songs as a small dataset and books as a voluminous dataset. Both of the datasets were combined by ourselves, and we also created a few mixed files.

3 Solutions and Evaluation

3.1 Shingling

The most effective way to represent documents as sets, for the purpose of identifying lexically similar documents is to construct from the document the set of short strings that appear within it.

How large a shingle size k should depend on how long typical documents are and how large the set of typical characters is. The important thing to remember is that k should be picked large enough that the probability of any given shingle appearing in any given document is low.

In our implementation, we used *k=5* for small texts and *k=10* for large documents as a default size of a k-shingle. Also, we cleaned up the text by removing punctuation, double spaces and new line charecters.

Further, instead of using substrings directly as shingles, we picked a built-in hash function that maps strings of length k to a hash value, which improves memory usage.

To compare these sets, we used the Jaccard similarity. The Jaccard similarity of sets S and T is the ratio of the size of the intersection of S and T to the size of their union:

$$\frac{|S \cap T|}{|S \cup T|} \tag{1}$$

3.2 Minhashing

Even with hashing, a sole shingling approach is extremely memory consuming. Here *"signatures"* comes in handy. A signature is a smaller representation of shingles. We can compare the signatures of two sets and estimate the Jaccard similarity of the underlying sets from the signatures alone. It is not possible that the signatures give the exact similarity of the sets they represent, but the estimates they provide are close, and the larger the signatures the more accurate the estimates.

For minhashing, we used 100 hash functions

$$\frac{(a * x + b)}{c}, \tag{2}$$

where a and b - random values, and c is constant.

The signature similarity is calculated as follows:

$$\frac{\sum_{n=1}^{N}[h_{min}(S^1) = h_{min}(S^2)]}{N}, \tag{3}$$

3.3 Extra effort: Locality-Sensitive Hashing

General approach to LSH is to "hash" items several times, in such a way that similar items are more likely to be hashed to the same bucket than dissimilar items are. We then consider any pair that hashed to the same bucket for any of the hashings to be a candidate pair. We check only the candidate pairs for similarity. The hope is that most of the dissimilar pairs will never hash to the same bucket, and therefore will never be checked.

The probability that the signatures agree in all the rows r of at least one band b, and therefore become a candidate pair, is

$$1 - 1(1 - s^r)^b. \tag{4}$$

In our implementation, we used the following values: $b = 20$ and *threshold = 0.8*. Once we defined candidate pairs, we compare them the same way we compared signatures from the previous approach.

4 How to run

To run, you can access *test.py* class in our project. It only contains a main method, that calls a runner method with required arguments. You would need Python 3+ (we used 3.9) to start a program.

```
if __name__ == '__main__':
    Runner().run('./data/', 5)
    Runner().run('voluminous-data/', 10)
```

5 Evaluation

Method	Small data	Big data
Shingling	0.024	13.336
Minhashing	0.554	227.469
LSH	0.001	0.205

Table 1: Benchmark results

Output for all approaches with a similarity threshold of *0.8* for a small dataset.

```
Jaccard similarity for documents
    ./data/Mr Grinch v3.txt and ./data/Mr Grinch v2.txt: 0.99
Jaccard similarity for documents
    ./data/Jingle Bell v2.txt and ./data/Jingle Bell v3.txt: 0.91

Jaccard signature similarity for documents
    ./data/Jingle Bell v2.txt and ./data/Jingle Bell v3.txt: 0.9
Jaccard signature similarity for documents
    ./data/Mr Grinch v3.txt and ./data/Mr Grinch v2.txt: 1.0

LSH signature similarity for documents
    ./data/Jingle Bell v2.txt and ./data/Jingle Bell v3.txt: 0.9
LSH signature similarity for documents
    ./data/Mr Grinch v3.txt and ./data/Mr Grinch v2.txt: 1.0
```

Output for all approaches with a similarity threshold of *0.8* for a big dataset.

```
Jaccard similarity for documents
    voluminous-data/Short-Stories v2.txt and voluminous-data/Short-Stories.txt:
         0.89

Jaccard signature similarity for documents
    voluminous-data/Short-Stories v2.txt and voluminous-data/Short-Stories.txt:
        0.98

LSH signature similarity for documents
    voluminous-data/Short-Stories v2.txt and voluminous-data/Short-Stories.txt:
        0.98
```

6 Conclusions

In this homework assignment, we learned three essential techniques for finding similar documents and managed to implement them.

According to the benchmark results, we can see that minhashing approach is ultimately ineffective for bigger dataset, while the LSH gives the same similarity accuracy and the fastest results. K-shingling is less time-consuming as a minhasing technique, however, it has the highest memory requirements.