Homework 1 - Data Mining

Group 11 - Perttu Jääsekläinen & Gabriele Morello

November 23, 2022

Contents

1 Introduction

In this homework we constructed a pipeline to find similarities between data among a set of documents. To achieve this, we used three techniques - each of them using the result of the previous one to operate on it and produce a new result.

We started by using shingling to decompose the documents in a set of strings, followed by MinHashing to reduce the number of comparisons and size of shingles, finished by locality-sensitive hashing to create candidate pairs of documents to compare - further optimizing the amount of comparisons done between documents.

We implemented our program in Scala using Apache Spark for shingling operations and distributed data sets, where the data sets used testing were composed of paragraphs from the Wikipedia pages of some countries.

Each of the three operations belongs to a class, and the main procedure invokes methods from them to produce the results.

2 Shingling

The first part of our implementation was building shingles of documents - where we used a k = 5 shingle size to look for similarities in Wikipedia articles for countries. We used a total set of 8 files, where 1 of the files was duplicated. Below is examples of shingles of size k = 5 for the word "hello world".

```
(hello), (ellow), (llowo), (lowor), (oworl), (world)
```

To create shingles we started reading files with Spark and stored them on a resilient distributed data set. We filtered out spaces and grouped all the shingles by the document they belong to. To save space, we hashed every shingle, which resulted in an RDD of tuples in the format (filename, set of hash of shingles). In the example below, we use a set of strings for example purposes - in practice, they were 32 bit integers.

```
[(filepath1, Set((ellow), (llowo), ...)),
    (filepath2, Set((lowor),(oworl), ...)), ...]
```

To compare the similarities between documents, we calculated the Jaccard similarity to get an accurate result (intersection of sets divided by the union of distinct elements).

When dealing with larger documents, comparing all shingles is not feasible - since the set of shingles is simply too large (would take way too long to compute and compare all documents and shingles). To improve this, we implementing MinHashing, introduced in the next section.

3 MinHashing

To compare every single shingle of every single document is very costly - especially when dealing with large amounts of files. To improve performance, we implemented MinHashing signatures for documents - which allows us to perform less comparisons and make an estimate for similarities rather than having an exact value.

To compute MinHashes, we made k = 100 permutations for all the distinct shingles - and looked for the first occurrence of shingles in each document. The signature for the document would be the row number of the first shingle that appears in the specific document

For example, a permutation for A,B,C,D,E could be C,D,A,B,E - and the signature for a column (document) would be 1 if it contains C, 2 if it contains D, or 3 if it contains A etc. When doing this multiple times, there is a high probability that similar documents will have the same signature for the different permutations.

When done with the permutations - we have a much smaller set of values to compare (one row for each permutation, where each row contains an entry for each document - rather than all shingles for all documents). This allows us to deal with larger files and compute similarities faster - for the trade off of being

slightly less accurate - since we might do too few permutations and run into edge cases.

To compare the signatures, we look for the amount of rows where documents have the same values, divided by the total amount of rows (permutations).

The more permutations we do, the more accurate result we get. If we do a low amount of permutations, for example 10, we could have scenarios where documents with 10k+ elements are determined to be 100% similar - even if they both only contain the same 10 elements and the permutations happen to be in a certain order.

MinHashing is an improvement from Shingling, but we still have the problem of doing way too many comparisons than necessary - instead, we might want to only compare documents *that might be similar* - which is what we do in Locality Sensitive Hashing.

4 Locality Sensitive Hashing (LSH)

MinHashing is very convenient to avoid having to compare all shingles - but it is still very costly if we want to compare a lot of rows and make lots of permutations for more accurate results.

To improve the performance and be able to compute similarities for larger files, we also implemented Locality Sensitive Hashing (LSH) to speed up the comparison process.

LSH works by using bands with a certain amount of permutations of rows (i.e. grouped together), which are hashed into a bucket. If multiple documents within the same band hash to the same bucket - we determine them to be a "candidate pair", which should be examined further for similarities.

Since we're only comparing bands and not every single pair of documents and permutations, we can avoid a lot of comparisons - only comparing the ones that actually hash to the same bucket.

To determine if documents are similar, at the end of LSH; we check in how many bands documents hashed to the same bucket - divided by the total amount of bands. If the ratio is above some threshold, t, we can estimate the documents to be similar.

As with MinHashing, the accuracy of LSH depends on the band size and the amount of rows in each band. We used a permutation size of $k = 1000$, with $b = 200$ bands and $r = 5$ rows for the similarities, which seemed to give about the same result as when using Jaccard similarity.

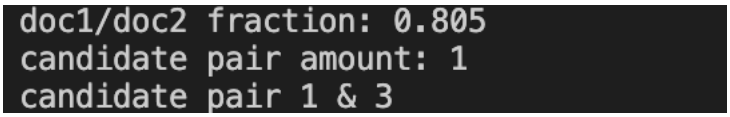

Figure 1: *The LSH candidate pairs, which is an estimate for similarities between documents. We determine that document 1 and 3 are similar (russia and russia copy) with a similiarity rate of 0.805 between 200 bands of 5 rows (k = 1000 permutations).*

5 Discussion

This homework gave us an introduction to working with Spark, which proved to be more difficult than we thought - since it requires another way of thinking when coding (since the different blocks of code aren't necessarily executed in the same environment). We also thought we'd learn Scala at the same time - which, in hindsight, probably wasn't the best idea.

We also had some struggles with understanding the concepts of MinHashing and LSH - but once it clicked, the implementation was fairly straight forward. We also attempted to parallellize the entire process - but didn't fully succeed, since we'd have to refactor the code on some level to account for the necessary changes. Since it wasn't a requirement for the assignment, we chose to not implement it. The overall functionality would be the same, but as it is coded right now, the program isn't optimized for larger data sets.

Now that we're familiar with Spark and Scala, we are both happy that we chose to learn them and can see ourselves using the same tools for other assignments in the future! :)

Figures for LSH 1 and Jaccard 2 contain example runs of our code, where we compared the Jaccard similarity of our file copy (with some rows removed). The LSH similairty is 0.805, while the Jaccard is 0.96 - which is an acceptable difference, since the LSH is an estimate.

Figure 2: *Example run of comparisons between Wikipedia articles with Jaccard similarities. The articles are very different, apart from the certain duplicate documents we've created.*