Data Mining - ID2222

Homework 1

Finding Similar Items: Textually Similar Documents

André Silva - Jérémy Navarro

November 9, 2020

1 Overview

We chose *Python* as the programming language and an OOP approach for the development of the project.

Information about the structure of the project, as well as instructions on how to run it follow:

```
$ unzip homework1.zip
$ cd homework1/
$ python3 main.py
```

The project is composed by the following files:

- `testFiles/`: Directory containing example documents and the SMS Spam Collection [2].
- `classes.py`: Contains the classes implementing the several stages of finding textually similar documents.
- `main.py`: The executable script which performs experimentation on the SMS Spam Collection [2], and demonstrates the usage of the classes.

2 Classes

2.1 Shingling

The class **Shingling** takes 3 arguments:

- `document`: Path of the document or document
- `k`: Size of each shingle
- `isFile`: `True` if `document` is a path to a file. Default is `True`

The first step consists in parsing the file's content into a string, if `document` is a file path.

After this, we compute the shingles of size **k**, which are put in an ordered set, after being hashed with **CRC32** [1]. We use this hash function as it allows us to represent each shingle as a 4B integer.

2.2 CompareSets

The class **CompareSets** takes 2 arguments:

- `set1`: Set of ordered hashed shingles
- `set2`: Set of ordered hashed shingles

The class computes the **Jaccard Similarity** of the two sets. For that, we iterate over the values of `set1` and check if they are present in `set2`. We use the result of this computation in the formula of the **Jaccard Similarity**.

2.3 MinHashing

The class **MinHashing** takes 3 arguments:

- `maxVal`: Maximum value of any hashed shingle. Default value is 2^{32}
- `k`: Number of hash functions to be generated. Default value is 100

We start off by generating the `k` independent hash functions. To do this, we need to generate `k` values of `a`, `b` and compute `c`. `a` and `b` are random integers with a lower value than `maxVal`, while `c` is a prime number slightly higher than `maxVal`, as to reduce colisions.

With the hash functions generated, we can now compute the minimum hash values for each of them, given a set. The output of this is a document signature.

2.4 CompareSignatures

The class **CompareSignatures** takes 2 arguments:

- `sig1`: Signature of a document
- `sig2`: Signature of a document

This class performs a simple task, computing the percentage of elements of the signatures that are equal. This results in an approximation of the **Jaccard Similarity** computed in 2.2.

2.5 (*Bonus*) LSH

The class **LSH** takes 3 arguments:

- **b** Integer of the number of bands
- **r** Integer of the number of rows
- **listMinHash** List of min-hashed signatures

First, we populate each band with the hash values of the corresponding parts of the signatures. The hash function used is the built in hash function for tuples.

Then, for every row in the band we check if there are equal hash values in other rows. As we are working with hash values, this is the equivalent of placing them in buckets.

The pairs of rows found in this last step represent the pairs of possible candidates.

3 Results

In Figure 1, we can see partial results of running out experiment.

We can notice that the number of candidate pairs generated at the end of **LHR** is very small when compared to the number of possible pairs of documents.

We can also see that, for the example of pair showed, the estimated similarity using **MinHash** is close to the **Jaccard Similarity** of the same documents.

```
Possible pairs: 27966
Candidate pairs: 70
Similar documents: 6


"You are a winner U have been specially selected
) " is similar to "You are a winner U have been
0871277810810"
Set1 : 112
Set2 : 106
Similarity 0.8793 = 87.93%
Estimated Similarity 0.8200 = 82.00%
Possible pairs: 61075
Candidate pairs: 247
Similar documents: 9
```

Figure 1: partial results of running `main.py`.

In Figure 2, the results of our experiment are shown.

We can see that the relation between the size of the dataset and the execution time is, approximately, linear. This means the algorithm scales relatively well as we increase the size of the dataset, something that wouldn't happen if we tried to compute the **Jaccard Similarity** between all possible pairs of documents.

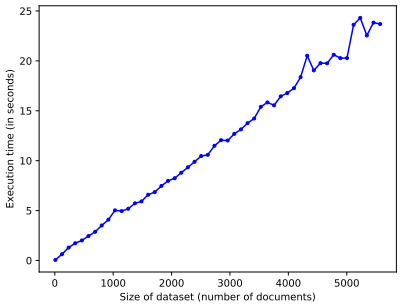

Figure 2: Execution time (in seconds) versus the size of the dataset [2].

References

[1] *CRC32*. `https://docs.python.org/3/library/zlib.html#zlib.crc32`. accessed: 2020-11-08.

[2] *SMS Spam Collection*. `https://archive.ics.uci.edu/ml/datasets/sms+spam+collection`. accessed: 2020-11-08.