Document similarity detection using hasing

Carlos Bergillos, Antoni Rambla, Adrià Cabeza
Departament de Computació

December 16, 2018

Abstract

Our goal is to identify similarities between documents. We have used the Jaccard Similarity theorem, *Local-Sensitive Hashing* algorithm and a *k-shingles* and *minhash signatures* representation of documents to evaluate the effectivity of the similarity computed and the time of computation. We have introduced three different hash functions to see its differences in performance. Also once determined the best parameters, we will give a conclusion about the best way to identify the more similar documents.

Contents

1 Introduction

Our goal is to identify similarities between documents. We say that two documents are similar if they contain a significant number of common substrings that are not too small.

The problem of computing the similarity between two files has been studied extensively and many programs have been developed to solve it. Algorithms for the problem have numerous applications, including spelling correction systems, file comparison tools or even the study of genetic evolution.

Existing approaches can also include a brute force approach of comparing all sub-strings of pair of documents. However, such and approach is computationally prohibitive.

In our case we have represented each document using a k-shingles set of strings, and implemented algorithms to calculate the Jaccard Similarity and an approximation of it using a *Local-Sensitive Hashing* algorithm based on *minhash signatures*.

2 Concept of similarity

First we have to focus into the definition of similarity, when we talk about the "Jaccard similarity",which is calculated by looking at the relative size of their intersection.

The Jaccard similarity, also known as Jaccard index is a statistical measure of similarity of sets. For two sets, it is defined as the size of the intersection divided by the size of the union of the sample sets. Mathematically,

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|} = \frac{|A \cap B|}{|A| + |B| - |A \cap B|}$$

Calculating the similarity estimation using this approach could be solved using k independent repetitions of the MinHash algorithm, however this would require $O(k \times |A|)$ running time.

Let's see an example of how the Jaccard Similarity can be calculated:

$A = \{0, 1, 2, 5, 6\}$,
$B = \{0, 2, 3, 4, 5, 7, 9\}$
$J(A,B) = |A \cap B| / |A \cup B| = |0, 2, 5| / |0, 1, 2, 3, 4, 5, 6, 7, 9| = 3/9 = 0.33$

The complexity of our implementation is $O(n)$. To calculate the intersection of both sets we use two iterators that iterate through both sets. When an element of a set is smaller than the other we increment its iterator and when they are equal we iterate both and a counter. Finally we use the value of the counter to apply the formula.

Pseudocode Jaccard Similarity

```
Intersection(A,B){
  result = 0;
  auto it = A.begin(), it2 = B.begin();
  while(it!= A.end() and it2 != B.end()){
    if(*it < *it2){
      ++it;
    }
    else if(*it > *it2){
      ++it2;
    }
    else {
      ++it;
      ++it2;
      ++result;
    }
  }
  return result;
}

Jaccard(A, B){
  intersection = intersection(A,B);
  result = intersection / (A.size() + B.size() - intersection);
}
```

The source code for this section can be found in '*jaccard.cc*'.

If we take in account the cost of building a set we should observe that for unsorted sequences our cost would be incremented to $O(n \times log(N))$.

3 Representation of documents

To identify lexically similar documents we need a proper way to represent documents as sets and the most effective way is to construct from the document the set of short strings that appear within it. If we do so, even if the documents have different sizes or those sentences appear in different order we will find several common elements. In the next section we will introduce some of the approaches of shingling and its variations.

3.1 *k*-Shingles

A k-shingle (or word-k-gram) is a sequence of consecutive words of size k. Intuitively, two documents A and B are similar if they share enough k-shingles. By performing union and intersection operations between the k-shingles, we can find the Jaccard similarity coefficient between A and B.
There are some variations regarding on how white space (blank, tab, newline, etc) is treated. Also there is a variation that works with a bag of shingles instead of a set to keep the number of appearances of a shingle.

How large k should be depends on how long typical documents are and how large the set of typical characters is. For example if we pick k = 4 there are $27^4 = 531441$ possible k-shingles. However, the calculation can be a little bit more subtle because all the characters do not appear with equal probability. A good rule of thumb is to imagine that there are only 20 characters and estimate the number of k-shingles as 20^k. For large documents, choice k = 9 is considered safe.

3.1.1 Hashing Shingles

Instead of using substrings directly as shingles, we can pick a hash function that maps strings of length k to some number of buckets and treat the resulting bucket number as the shingle. That process compacts our data and lets us manipulate shingles by single-word machine operations.

4 MinHash

A minhash function on sets is based on a permutation of the universal set. Given any such permutation, the minhash value for a set is that element of the set that appears first in the permuted order.

This algorithm provides us with a fast approximation to the Jaccard Similarity. The concept is to condense the large sets of unique shingles into a much smaller representations called "signatures". We will then use these signatures to measure the similarity between document, the signature won't give us the exact similarity but we will get a close estimate (the larger the number of signatures you choose, the more accurate the estimate). In this case we define the similarity like:

$$sim(a,b) = \frac{1}{t}\sum_{i=1}^{t}\{1 \quad if \quad a_i = b_i \quad or \quad 0 \quad if \quad a_i \neq b_i\}$$

To implement the idea of generating randomly permuted rows, we don't actually generate the random numbers, since it is not feasible to do so for large datasets, e.g. For a million items you will have to generate a million integers..., not to mention you have to do this for each signatures that you wish to generate. One way to avoid having to generate n permuted rows is to pick n hash functions in the form of :

$$h(x) = (ax + b)\ mod(c)$$

Where:

- x is the row numbers of your original characteristic matrix
- a and b are any random numbers smaller or equivalent to the maximum number of x
- c is a prime number slightly larger than the total number of shingle sets.

4.1 Getting the similarity

To compute the similarity between two documents we only have to traverse the signature matrix once and do an operation similar to the jaccard similarity. The complexity of this algorithm is $O(h)$ where h is the number of hash functions desired.

Pseudocode Similarity Between Two Documents

```
sim(signatureMatrix,a,b)
    float simil = 0;
    for i = 0 to signatureMatrix.size() do:
        if(signatureMatrix[i][a] == signatureMatrix[i][b]) then
          ↪ ++simil;

    return simil / signatureMatrix.size();
```

The source code for this section can be found in '*jaccardaprox.cc*'.

5 Locality-Sensitive Hashing for Documents

This technique allows us to avoid computing the similarity of every pair of sets or their minhash signatures. If we are given signatures for the sets, we may divide them into bands, and only measure the similarity of a pair of sets if they are identical in at least one band. By choosing the size of bands appropriately, we can eliminate from consideration most of the pairs that do not meet our threshold of similarity.

The source code for all subsections can be found in '*jaccardaprox.cc*'.

5.1 Hash function used to hash a vector

This is the code used to hash a row. It consists on a series of XOR operations on each element of the vector together with a random hexadecimal number and the shifting the value of "seed" from the previous iteration some bits to the left and to the right. The time complexity of this algorithm is $O(n)$ being n the size of the vector.

Pseudocode Vector Hashing

```
hash_vec(V) {
  seed = V.size();
  for each i in V do:
    seed = seed XOR (i + 0x9e3779b9 + (seed << 6) + (seed >> 2));
    //operands << and >> are shifting left and right respectively
       ↪ x bits
  return seed;
}
```

5.2 Filling the characteristic matrix

The characteristic matrix is the one we use to represent the shingles that each document has (see section 3). The complexity of this algorithm is $O(n^2)$ where n is the total number of shingles that we get as input.

- characMatrix is the characteristic matrix
- shingles is a set containing all the shingles we have in total
- docShing is a vector of sets in which in position i we have the set of shingles of document i.

Pseudocode To Fill The Characteristic Matrix

```
fill(characMatrix, shingles,docShing){
    iterator it = shingles.begin();
    for i = 0 to characMatrix.size() do:
        shingle = *(it++);
        for j = 0 to characMatrix[0].size() do:
            if(docShing[j] contains(shingle)) then characMatrix[i
               ↪ ][j] = 1;
            else characMatrix[i][j] = 0;
```

5.3 Computing the signature matrix

The signature matrix is the one resulting from finding for each hash function that we have applied and for every document, the first row where there appears a 1 in its permutation. To compute the signature matrix we use one of the three algorithms.

- modular hashing
- multiplicative hashing
- murmur hashing

The time complexity on this algorithm is tricky as we have to take into account the cost of the hashing algorithm used. Usually the time will be $O(n^2 * h * \zeta)$ where n is the number of shingles of the characteristic matrix, h is the number of hashing algorithms that we use and ζ is the cost of computing the hash value. Albeit we use the algorithms only to compute the hash to do row permutations, we follow a general schema to fill it that goes as it follows:

Pseudocode To Compute The Signature Matrix

```
initialize signature matrix with infinity values
for i = 0 to characterisicMatrix.size() do:
  for j = 0 to characteristicMatrix[0].size() do:
    if(characteristicMatrix[i][j] == 1) then
      for k = 0 to h do:  //h is the number of hash functions
        ↪ that we need
        rowPermutated <- value of hash algorithm designed for row
          ↪ i
        //note that we always use the same algorithm
        if(rowPermutated < signatureMatrix[k][j]) signatureMatrix
          ↪ [k][j] = rowPermutated;
```

5.4 LSH and candidate pairs

As we have mentioned in the beginning of this section, by choosing the size of bands and rows appropriately we augment the number of false positives, documents that have very little in common, or the number of false negatives, documents that have a lot of common but are not checked. The time complexity of this algorithm is $O(h * m * h/r * m) = O(h^2/r * m^2)$ (h/r is the cost of of making the rows and the last m is the worst case cost of the last for for all documents being the same(no hauria de ser m^2 ja que es sum de l=1 fins a m de l)) where:

- h is the number of hash functions
- m is the number of documents
- r is the number of rows

Pseudocode To Do LSH

```
LSH(signatureMatrix,r,h)
    candidats;
    bucket;
    for i = 0 to h adding each time r to i (i+=r) do:
        bucket.clear();
        for j = 0 to signatureMatrix[0].size() do:
            compute the row for document j;
            doc1 = hash_vec(row);
            if(bucket contains doc1)
                for l = 0 to size of the bucket that stores all
                    ↪ documents with value doc1
                    insert in candidats every possible pair that
                        ↪ can be done with j and all documents
                        ↪ that were here beforehand.
                we add j to the bucket that represents the value
                    ↪ doc1
            else
                we create another place for rows with value doc1
                    ↪ and we store j there

    return candidats;
```

We choose to return them as a set to avoid having to check a and b and then maybe later b and a again.

Function that orders a pair placing the smallest number as the first. The complexity is trivial here $O(1)$

```
parella_inc(int a, int b){
  if(a < b ) then return pair(a,b);
  else return pair(b,a);
```

6 Hashing

Hashing is a technique for dimensionality reduction. It uses a hash function that is any function that can be used to map data (called a key) of arbitrary size to data of a fixed size (called a hash value or hash). That hash is a sum up of everything that is in the data. You can never make it backwards from the hash to the data.

Hashing is done for indexing and locating items in databases because it is easier to find the shorter hash value than the longer string.

The hash functions that we want to use need to be:

- **Really fast**
  Dimensionality reduction is often a time bottle-neck and using a fast basic hash function to implement it may improve running times significantly.
- **Avoid hash collisions**
  We do not want to get the same hash using different pieces of data
- **Uniform distribution**
  The hash values are uniformly distributed.

In our project we have not chosen hash cryptographic functions (i.e. sha-1 or md5) because they are too slow for our purpose.

6.1 Modular Hashing

In the modular hashing (also called the division method), we map a key k into one of m slots by taking the reminder of k divided by m. It takes the following form.

$$h(k) = (ak + b)\ (mod\ m))$$

Where:

- a and b are any random numbers smaller or equivalent to the maximum number of k

When using the division method, we usually avoid certain values of m. For example, $m = 2^p$ for some integer p, then $h(k)$ would be just the p-lowest-order bits of k. A prime value is often a good choice of m.

Pseudocode Modular Hashing

```
ModularHashing(k)
   x ← the maximum value possible of k
   a ← random number mod (x)
   b ← random number mod (x)
   m ← a prime number ≥ x
   value ← a × k + b mod (m)
```

The source code for this section can be found in '*ModularHash.cc*'.

6.2 Multiplicative Hashing

The multiplicative method for creating hash functions operates in two steps. Firstly, we multiply the key k by a constant A in the range $0 < A < 1$ and extract the fractional part of kA. Then, we multiply this value by m and take the floor the result. To sum up:

$$\lfloor m\ kA \times mod(1) \rfloor$$

Where:

- $kA \times mod(1)$ is the fractional part of kA, that is, $kA - \lfloor kA \rfloor$

An advantage of the multiplication method is that the value of m is not critical. We typically choose it to be a power of 2 ($m = 2^r$ for some integer r), since we can then easily implement the function on most computers.

Supposing that the word size of the machine is w bits, that k fits in a single word and p is the number of bits that you want for the size of your hash value.

We restrict A to be a fraction of the form, where s is an integer in the range $0 < s < 2^w$. We first mulitply k by the w-bit integer $s = A \times 2^w$. The result is a $|2 \times w|$-bit value. The desired p-bit hash value consists of the p most significant bits of r_0.

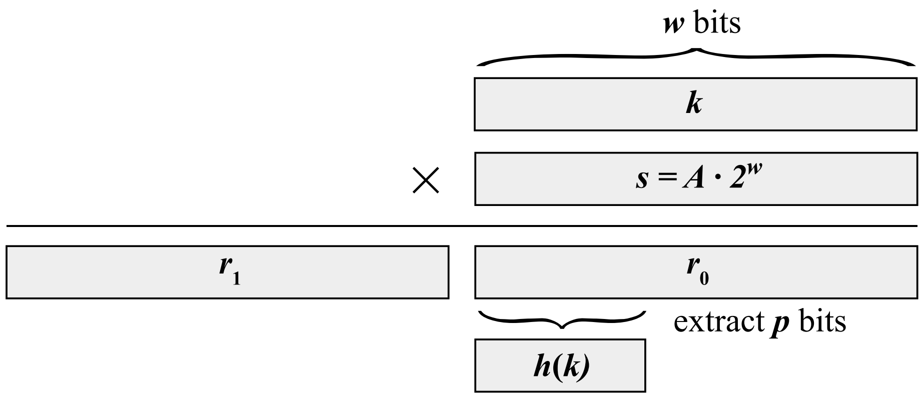

Figure 1: Multiplicative Hashing

Although this method works with any value of the constant A, it works better with some values than with others. The optimal choice depends on the characteristics of the data being hashed. Donald Knuth suggest that...

$A = (\sqrt{5} - 1)/2 \simeq 0.6180339887$

...is likely to work well. So that's why we used this value as an starting point to generate different multiplicative hash functions.

Pseudocode Multiplicative Hashing

```
MultiplicativeHashing(k,p)
```

$A \leftarrow (\sqrt{5} - 1)/2 \simeq 0.6180339887$
$s \leftarrow A \times 2^w$
$r \leftarrow k \times s$
$r_0 \leftarrow r \; mod \; (2^w)$
$value \leftarrow r_0 >> (w - p)$

The source code for this section can be found in '*MultiplicativeHash.cc*'.

6.2.1 Murmur Hash

Consists in applying some multiplications (MU) and rotations (R) to the entry bytes to obtaint the hash. It uses multiple constants which are decided to make it a good hash function by passing 2 basic tests, the Avalanche Test that evaluates how the output changes if the input is slightly modified and the statistical Chi-Squared Test. We have based the implementation of this hash function on Austin Appleby's implementation. Source : `https://sites.google.com/site/murmurhash/`

Pseudocode extracted from Wikipedia

```
Murmur3(key, len, seed)
    c1 ← 0xcc9e2d51
    c2 ← 0x1b873593
    r1 ← 15
    r2 ← 13
    m ← 5
    n ← 0xe6546b64

    hash ← seed

    for each fourByteChunk of key
        k ← fourByteChunk

        k ← k × c1
        k ← (k ROL r1)
        k ← k × c2

        hash ← hash XOR k
        hash ← (hash ROL r2)
        hash ← hash × m + n

    with any remainingBytesInKey
        remainingBytes ← remainingBytes × c1
        remainingBytes ← (remainingBytes ROL r1)
        remainingBytes ← remainingBytes × c2

        hash ← hash XOR remainingBytes

    hash ← hash XOR len
    hash ← hash XOR (hash >> 16)
    hash ← hash × 0x85ebca6b
    hash ← hash XOR (hash >> 13)
```

```
hash ← hash × 0xc2b2ae35
hash ← hash XOR (hash >> 16)
```

The source code for this section can be found in '*MurmurHash3.cc*'.

7 Data

7.1 Real-world data

- **Harry Potter and the Sourcerer Stone**: All the text from the first Harry Potter novel.
- **The Lord of The Rings: The return of the King**: The entire script from the last Lord of the Rings movie.
- **Star Wars: Heir to the Empire**: An Star Wars novel by Timothy Zahn.

7.2 Generating the data

To generate the data for our experiments we have used the *random_ shuffle* function inside the C++ STL which rearranges the elements randomly of a vector. The function swaps the value of each element with that of some other randomly picked element. Its complexity is $O(n)$ with n being the distance between first and last minus one.

The algorithm we implemented to generate our data consists in traversing all the document once to generate a vector of strings containing all the words, then, as many times as permutations we want, we shuffle the vector and write the result in a document.

8 Experiments

We designed some experiments in order to test our algorithms, and discover how some of their variables affect the outcome.

8.1 *k*-shingles Size

The source code for this experiment can be found in '*jocProvesJaccard.cc*'.

We tested different sizes of k for our k-shingles. For us, k defines the number of characters for each k-shingle.

Using the algorithm explained in subsection 7.2, we generated 20 permutations of a 50 words document, thus obtaining 20 new documents.

Then we calculated the Jaccard Similarity (section section 2) for all the possible pairs of documents from this set.

The number of possible pair combinations for a set of n elements (where order is not important) is given by $\binom{n}{2}$.

In our case, with a set of 20 documents we have:

$$\binom{20}{2} = 190$$

So we computed the Jaccard Similarity for these 190 pairs of documents, for different sizes of k, ranging from 2 to 20. The results can be seen in Figure 2.

As we expected, the similarity decreases as the size of the k-shingles increases.

For smaller values of k that allow k-shingles to remain within a word, we see that a lot of similarities are found, as the documents share the same words. On the other hand, for values of k that span more than one word, the similarity between documents is very small (because the documents don't share the word order, so the probability of identical k-shingles decreases). In fact, we observe that for values of k larger 12, the similarity obtained is mostly 0.

As mentioned in subsection 3.1, a good and reasonable value for k is 9. For $k = 9$, the similarity found between all our random documents is less than 0.1, which corresponds with the fact that the documents don't really share sentences or many contiguous words.

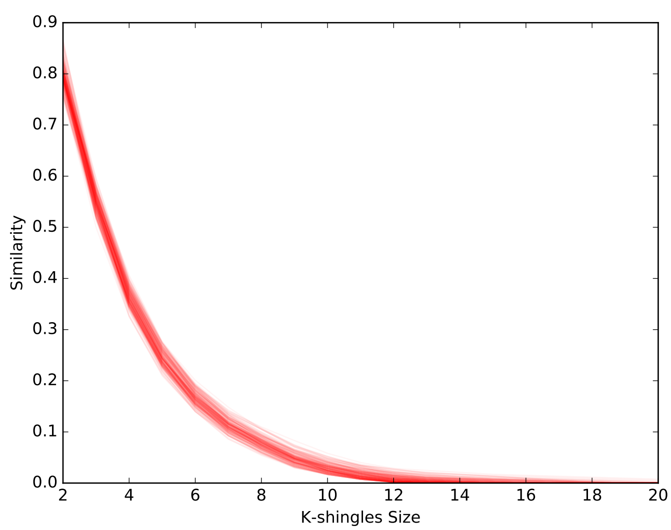

Figure 2: Relation between the K-shingles size and the similarity (for all of the possible pairwise combinations of 20 random permutations of a 50 words document)

8.2 Performance of Different Hashing Algorithms

The source code for this experiment can be found in '*jocProvesHashTimes.cc*'.

We have tested different k sizes for our k-shingles and three different hash functions in the process of filling the signature matrix to check their different performance. The process of filling the signature matrix is explained in the section 7 and our hash functions are MurmurHash3, Multiplicative Hash and Modular Hash and can be seen more deeply in the section 8.

As we expected initially, the time increases as the size of the k-shingle increases. This behavior can be explained using the fact that the number of different shingles we would have to work with would be bigger. It is trivial to observe that the number of different k-shingles is going to have a growing tendency based on their size.

Also we have been able to observe that the performance of MurmurHash and MultiplicativeHash are almost the same and ModularHash is the one that has performed the best, performing almost always with values smaller than the half of the others two.

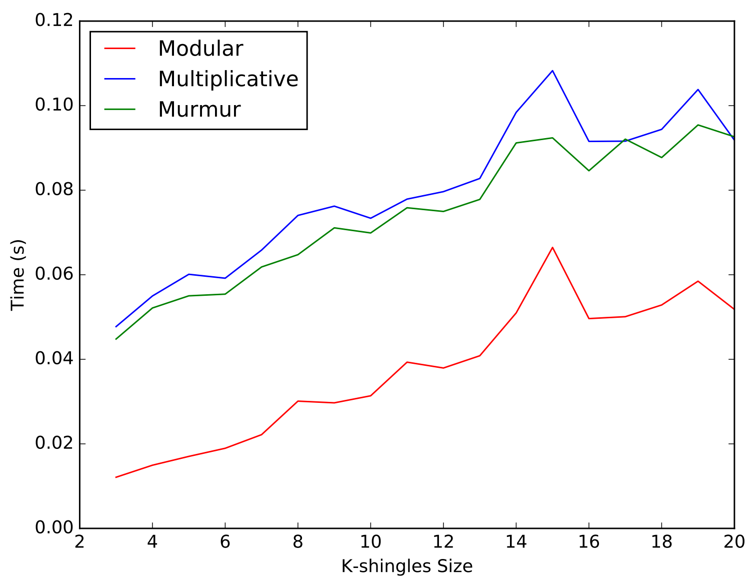

Figure 3: Difference between the time used by different Hashing Functions

8.3 Performance of Different Document Similarity Approaches

The source code for this experiment can be found in '*jocProvesJaccSim.cc*'.

We tested three different approaches to find similarities in a set of documents.

- Calculating the Jaccard similarity for all the possible pair combinations in the document set.
- Calculating a Jaccard similarity approximation via a computed signature matrix.
- Calculating a Jaccard similarity approximation via a computed signature matrix with a locality-sensitive algorithm.

Each approach is supposed to perform respectively faster.

We created a set of 410 documents (result of random permutations of a 50-words document, as explained in subsection 7.2).

We then independently applied these 3 approaches to subsets of gradually increasing size, ending with the whole set of 410 documents, while recording the times of each process. The plotted results can be found in Figure 4.

All the tests in this experiment were done with k-shingles of size 5, and for the second and third approach, we used 200 hashing functions (resulting in a signature matrix of 200 rows). Additionally, the third approach divided the 200 rows in 20 bands of 10 rows. These parameters were chosen to simulate a plausible real application.

We can observe that the first approach, while being the only one that computes the real Jaccard similarity (and not an approximation), is exponential in nature, and thus, is not feasible for large document sets. In fact, while doing this test, we had to limit the execution of this approach to only sets with less than 100 documents, as it becomes unmanageable otherwise.

The other two approaches are way better alternatives, as they are asymptotically faster. And, while they seem to both be asymptotically linear, we don't think that's the case. We don't have the time or computing power to do tests with larger datasets, but the way the second approach is constructed, it iterates through all the $\binom{n}{2}$ possible pairs of documents, and this has cost $O(n^2)$, with n being the number of documents. We believe that most of the computation time is spent constructing the k-shingles set, and constructing the signature matrix, with a cost of $O(n)$, and that the similarity calculation for all the possible document pairs represents only a small fraction of the total time for relatively small datasets, thus making it seem of linear behavior. At the same time, the third approach is also faster than the second one, we can see in Figure 4 how they diverge as the size of the datasets increases. In this case, we believe the cost is still quadratic, but with a way smaller scaling factor, as only a small fraction of the possible $\binom{n}{2}$ document pairs are selected as candidates for similarity checking.

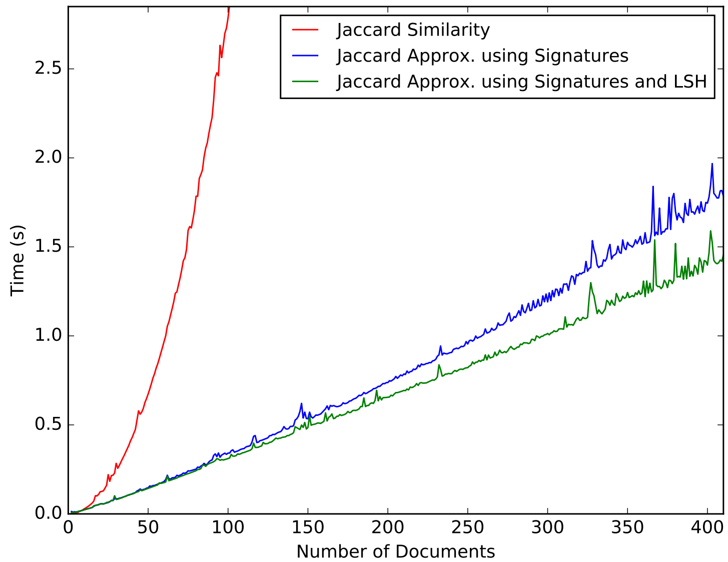

Figure 4: Time spent by the three approaches for different sets of documents of increasing size

8.4 Precision of Jaccard Similarity Approximations

The source code for this experiment can be found in '*jocProvesJaccSimLsh.cc*'.

In the following experiment we wanted to test how precise the outputed similarity for the MinHash method is.

In the experiment we have observed how the similarity changes depending on the number of hash functions used. Also we have plotted the Jaccard Similarity value in order to see which is the real similarity for the pair of documents analysed and check how the precision changes. All the tests in this experiment were done with a pair of documents with 60% similarity and with k-shingles of size 9, which is a good value for k as we argued in subsection 3.1.

As we expected when the number of hash functions is small the difference between the real similarity and the approximation differs a lot. Moreover, as the number of hash functions increases the difference of similarity decreases significantly. The reason we believe this happens is because albeit the hash functions are not perfect and create collisions, the number of "good" permutations outnumbers the ones that collide. Having more functions means that overall we have more "good" permutations and thus the impact that the "bad" ones have is less significant.

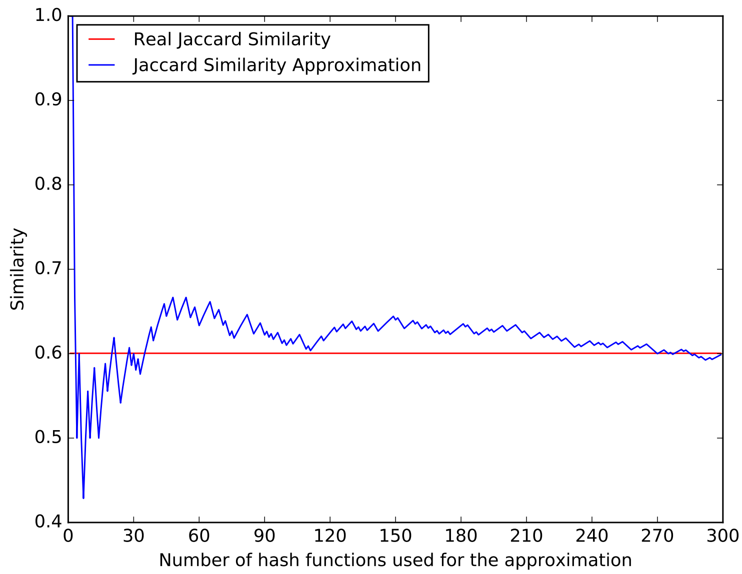

Figure 5: Approximated Jaccard Similarity for a pair of 2 documents with 60% real similarity (K-shingles size = 9)

9 Conclusions

1. Small sizes for k-shingles result in many false positives. And very large sizes for the k-shingles result in very few (if any) similarities being found. 9 can be a good value for general text documents.

2. Larger sizes for the k-shingles result in longer computation times.

3. Of the hashing functions we evaluated, Modular Hashing is the one that performed the best. Resulting in significantly faster computation times than Multiplicative Hashing and Murmur Hashing.

4. Trying to find pairs of similar documents by computing the Jaccard similarity for all the possible pairs is not feasible for large document sets. Jaccard approximation algorithms using minhash signatures produce a very similar result to the real Jaccard similarity, but with a fraction of the cost. And for very large documents sets, locality-sensitive hashing should be considered.

5. Larger number of hash functions result in a better precision for the MinHash method.

References

[1] J. Bank and B. Cole, *Calculating the Jaccard similarity coefficient with map reduce for entity pairs in Wikipedia* (Wikipedia Similarity Team, 2008).

[2] J. Leskovec, A. Rajaraman and J. Ullman *Finding Similar Items. In Mining of Massive Datasets* (Cambridge University Press, 2014).

[3] E. W. Myers *An O(ND) Difference Algorithm and Its Variations* (Department of Computer Science, University of Arizona)

[4] S. Alzahrani and N. Salim, *Fuzzy Semantic-Based String Similarity for Extrinsic Plagiarism Detection* (Taif University, Saudi Arabia and Universiti Teknologi Malaysia, Malaysia)

[5] S. Dahlgaard, M. Tejs Knudsen and M. Thorup, *Practical Hash Functions for Similarity Estimation and Dimensionality Reduction* (University of Copenhagen)

[6] D. Lemire and O. Kaser, *Strongly universal string hashing is fast* (Université du Québec Canadá and University of New Brunswick, Canada)

[7] C++ Standard Template Library, Source: `cplusplus.com/reference/stl/`

[8] *MurmurHash.* Source: `https://en.wikipedia.org/wiki/MurmurHash`

[9] *Hash function.* Source: `https://en.wikipedia.org/wiki/Hash_function`

[10] T. H. Cormen, C. E. Leierson, R. L. Rivest and C. Stein, *Introduction to Algorithms. Third Edition*

[11] D. Knuth, *Sorting and Searching*, volume 3 of *The Art of Computer Programming* (Addison-Wesley, 1997. Third Edition)

[12] S. Yar [Developer's Catalog By Sahib Yar] *Sorting and Searching*, *Murmur Hash - Explained* (Youtube, Jun 23, 2017), Retrieved from: `https://www.youtube.com/watch?v=b8HzEZt0RCQ&t=1s`

[13] T. Scoot [Computerphile] *Hashing Algorithms and Security - Computerphile*, (Youtube, Nov 8, 2013), Retrieved from: `https://www.youtube.com/watch?v=b4b8ktEV4Bg&t=181s`

[14] L. Lamport, *LATEX: a document preparation system: user's guide and reference manual* (Addison-Wesley Pub. Co., cop. 1994)