Near deduplication

Yiqiu Wu

yw3101

1 Introduction

Nowadays, web search engine needs to handle billions of data every day. However, not every web page is useful for a web search engine to index, especially when we take completely useless spam web pages into consideration. PageRank is a way to detect spam web pages. But we also don't want several web pages with like identical news to appear again and again in our data. Near deduplication is a way to get rid of such problems. If we detect and remove all the near duplicates from our data, we can save not only a lot of space required to store index in our servers, but also a lot of time wasting on useless computation. What's more, we can make use of detection result to improve the crawler to skip those websites which always provide some useless copies or spams.

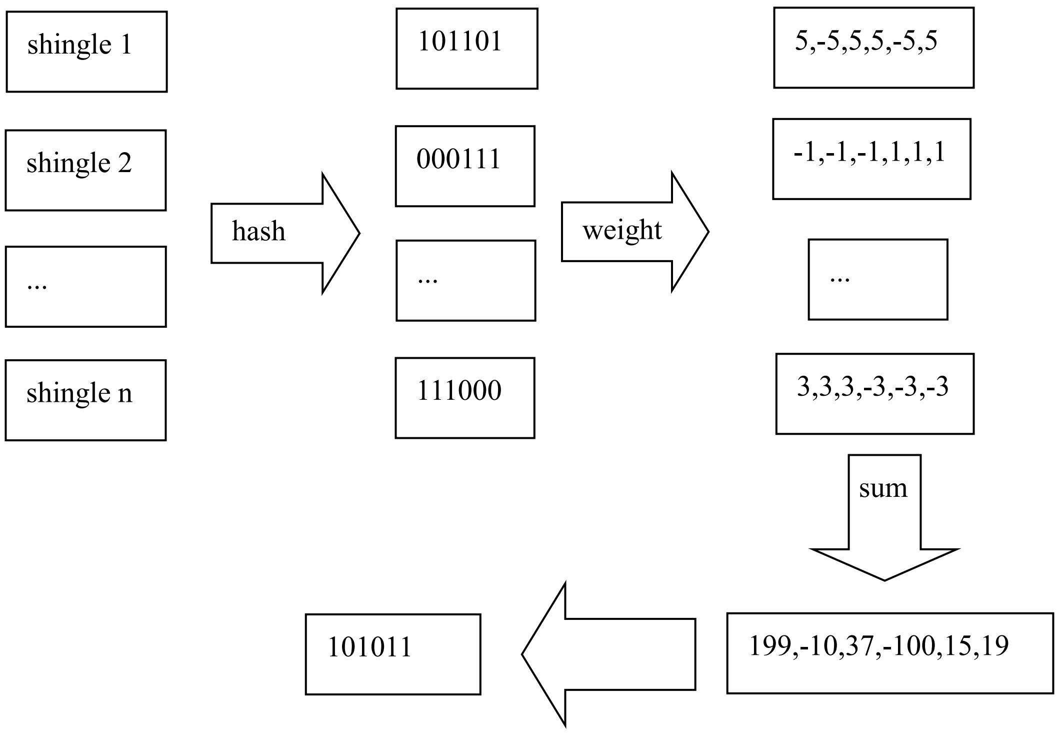
In this study, we introduce two mainly used methods to detect near duplicates under a mass amount of datasets, which is SimHash and MinHash. This two methods are both used by Google for their web crawling and other algorithms. They are some sort of approximation algorithms trading some precision and recall for speed, but are still a promising technique for use on large collections. A main challenge is that how to quickly get the difference of all pairs without simply computing all of them. In this project, we mainly focus on near duplicates detection, because deduplication can be done once we get which pages are duplicates. We will realize both methods in c++ project, compare the result between them based on same data, and maybe decide which one has more advantages over the other.

Overall, we explain the basic principle of two methods and show some actual implementation details in code. How to run the project is provided after that. We compare the result between those two methods based on same data and discuss pros and cons of each method.

2 SimHash

When it comes to duplicate detection, a simple idea is that we can hash the whole text and check whether two different texts share a same hash value. But the issue is that the hash values will be totally different even when one text has a different word. So we use locality-sensitive hashing to detect near duplicates. SimHash is such a kind of LSH. It is a special hash function that differs from conventional and cryptographic hash functions because it aims to maximize the probability of a 'collision' for similar items. In other words, SimHash attempts to make sure that documents that look the same get very similar hashes.

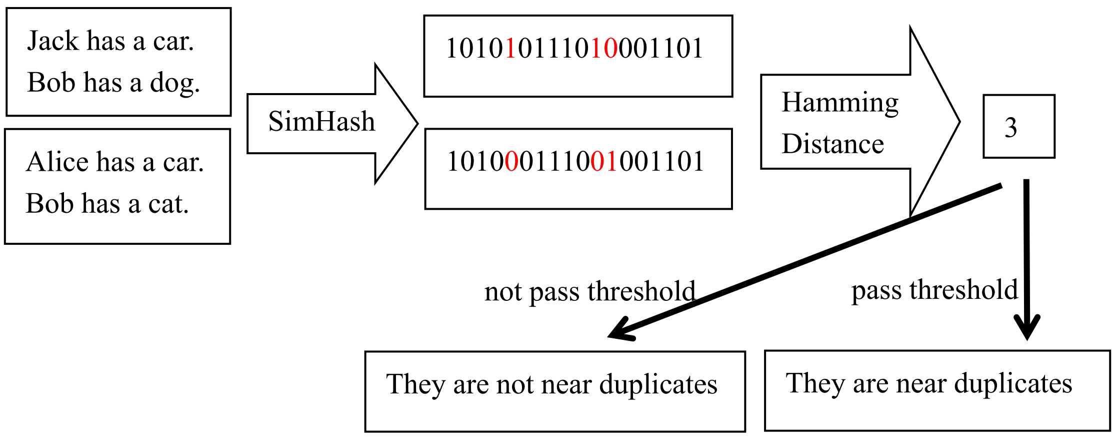
The following graph 1.0 shows the working procedure that applies SimHash to generate a document to a fingerprint.



graph 1.0

Initially, a document is pre-processed to extract a set of keywords from its content. A sequence of like 3 words forms a shingle which is the basic unit in SimHash. We initialize an k-dimensional vector with each dimension as zero. Here, for each shingle of document, it is hashed to a k-bit hash value. The increase or decrease in the k components from these k bits depends on the weight of that shingle. In the next step, the components' sign determines the corresponding bits of the final fingerprint(signature) of document. Generally, we use 64-bit SimHash. The advantage of SimHash is that even if we remove a word from the document, only a few bits will be different.

Hamming distance is usually used to represent the similarity between two documents. In this case, Hamming distance is the number of positions at which the corresponding bits are different. The following graph 2.0 shows how we detect near duplicates between two documents.



graph 2.0

In graph 2.0, we have two similar texts. The SimHash value of each text is computed. The result of two texts have 3 different bits, which means they have a Hamming distance of 3. In my project, a score of 64-3=61 is generated. Then, we check whether this score is over a threshold we set. Pairs passing the threshold are near duplicates.

2 Implement SimHash

The basic idea of SimHash is that for each document, we hash every basic unit into 64-bits multiplying their weight. We sum up all the 64-bits result and check whether the final number in each bit is positive or negative. A positive bit creates ‘1’ in corresponding bit of the final fingerprint. The following pseudo code shows how to get a fingerprint of a particular document.

```
pseudo code:(for document D)
Int fingerprint[64] = 0;
For each term Ti in D
{
    Hi = hash(Ti);
    For each bit Bj in Hi
    {
        if(Bj == 1)
        {
            fingerprint[j] += weight of Ti;
        }
        else
        {
            fingerprint[j] -= weight of Ti;
        }
    }
}
For i from 0 to 63
{
    if(fingerprint[i]>0)
    {
        fingerprint[i]=1;
    }
```

```
        else
        {
                fingerprint[i]=0;
        }
}
```

The following pseudo code shows how a score of a pair is computed with SimHash.

```
pseudo code:(for pair(doc1,doc2))
int score=0;
For i from 0 to 63
{
        if(doc1.fingerprint[i]==doc2.fingerprint[i])
        {
                score++;
        }
}
```

After getting fingerprints of all the documents. We can probably finish our duplicate detection by simply comparing fingerprints between each pair of documents. However, comparing pairs after pairs is not very wise because we have millions of documents to handle with, the cost would be very high if we don't have a more efficient way to compare the fingerprints. Later on, we will discuss how to overcome this problem when we get a mass amount of data. But first, let's introduce another important method.

3 MinHash

The similarity of two word sets A and B can be measured by Jaccard similarity, which is defined as: $J(A,B)=\frac{|A\cap B|}{|A\cup B|}$ (the intersection of the sets divided by their union)

The following graph 3.0 table is a simple example of Jaccard similarity.

	A	B
word1	0	0
word2	1	1
word3	0	1
word4	1	1

graph 3.0

'1' represents the existence of a word in one particular file, while '0' represents the opposite situation. In this case, we easily get the Jaccard similarity between A and B: J(A,B)=2/4=0.5. In fact, we will never do the conjunction and disjunction computation to get the Jaccard similarity because the size of words set can be very large. Instead, we look for the first '1' from top to bottom for each set and record the word ID. For this example, we get result '2' from both A and B. The same number means the two sets have a common word 'word2'.

When we randomly shuffle the rows of the table, the result are as graph 4.0.

	A	B

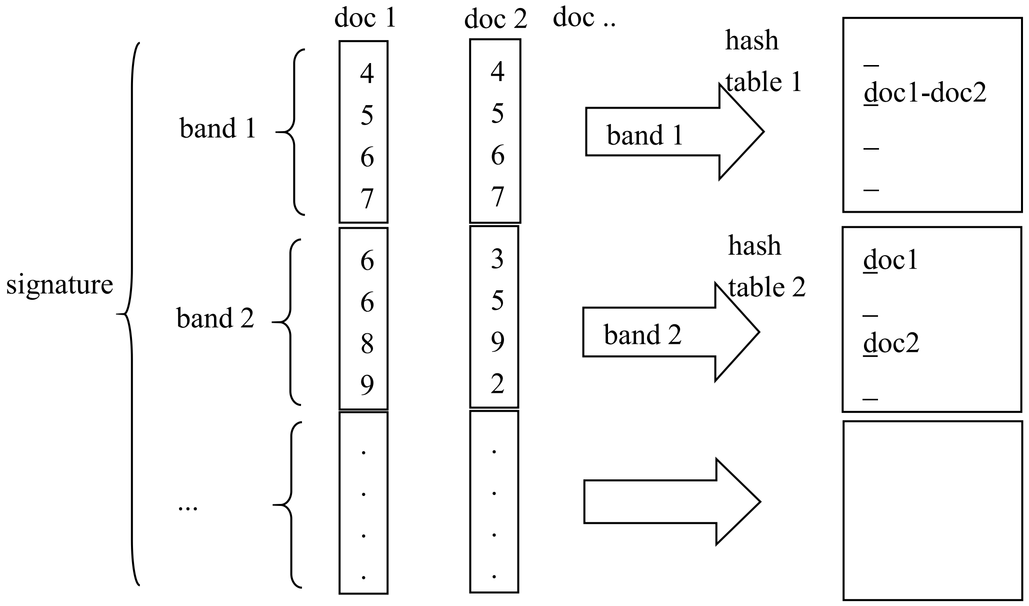
word4	1	1
word3	0	1
word1	0	0
word2	1	1

graph 4.0

We get result ‘1’ from both A and B, which means they have another word ‘word4’ in common. All of these basically generate the theory of MinHash. MinHash of a set is the number of the (row) element with first none-zero in the permuted order. Whether getting a same MinHash is equivalent to randomly drawing elements of the union A ∪ B and checking if they are a member of the intersection A ∩ B. The probability of this being true is exactly the Jaccard similarity. $\Pr[h_{\min}(A) = h_{\min}(B)] = J(A,B)$

Now we try like shuffling 20 times, then we will get 20 MinHash values for each set. We call this sequence of values a signature of each set. Apparently, if we get two sets with exactly the same signature, the probability of them being similar is very high.

We can find all of the pairs of documents that are near-duplicates of each other by simply comparing signatures between each pair of documents. The same problem we met before now appears again. If we have millions of documents, the comparing process will take huge amount of time. We decide to divide signature into b bands, each of which contains r values. We hash each band to a bucket. The probability that two documents are hashed into same bucket by all bands is $1 - (1 - p^r)^b$. As the final result, instead of comparing all the documents, we just output documents in the same bucket which greatly reduce the time cost. The following example shows how we decompose the signature and hash it to buckets to decrease the total number of comparison times.



graph 5.0

In graph 5.0, signature of doc1 and doc2 is split into several bands. After hashing the first band of doc1 and doc2, they are in a same bucket, just because they have exactly the same band value. But after hashing the second band, doc1 and doc2 are separated in different buckets. When it comes to the process of comparison, we scan all the buckets and luckily get a bucket containing both doc1 and doc2. After that, we compare the whole signature between doc1 and doc2, and find out whether they are truly near duplicates. This band method can also be applied to SimHash by seperating fingerprint to 4 parts.

4 Implement MinHash

In order to get the shuffling effect, we randomly generate several hash functions applying on word ID instead of actually rearranging the rows of the table. These hash functions have the format of $h(x) = (ax + b)\%c$ which produce a different random mapping of the values for different choices of a and b.

For one document, we scan all the words and apply every hash functions on them. The minimum hash values of each hash function form the final signature. The following pseudo code shows how to get a signature of a particular document.

```
pseudo code:(for document D)
generate 20 different hash functions;
int signature[20] = INT_MAX;
For each term Ti in D
{
    For each hash function Fj
    {
        if(Fj(Ti) < signature[j])
        {
            signature[j] = Fj(Ti);
        }
    }
}
```

The following pseudo code shows how a score of a pair is computed with MinHash.

```
pseudo code:(for pair(doc1,doc2))
int score=0;
For i from 0 to 19
{
    if(doc1.signature[i]==doc2.signature[i])
    {
        score++;
    }
}
```

5 How to run the project

This program is written in C++ on Visual Studio 2017 on Windows 10. It is recommended to use Visual Studio to open this whole project.

Before running this program, please check if the program is running in Release Mode, which will greatly improve the running speed of this program. Also please make sure there is enough stack for this project (The stack reserve size is already set to be a big enough number in project setting).

Original pages files must be uncompressed WET files from “http://commoncrawl.org”.

There is no limit on the number and the name of those files. To begin deduplication, need to put those files into “CreateIndex/pages/” directory.

When start running the program, input the hash mode you want. 1 means MinHash, 2 means SimHash.

After the program finishes its job, there will be a near duplicates detection result in “CreateIndex/result/” based on which hash you chose. Each page pair with their similarity score will be listed in the txt file.

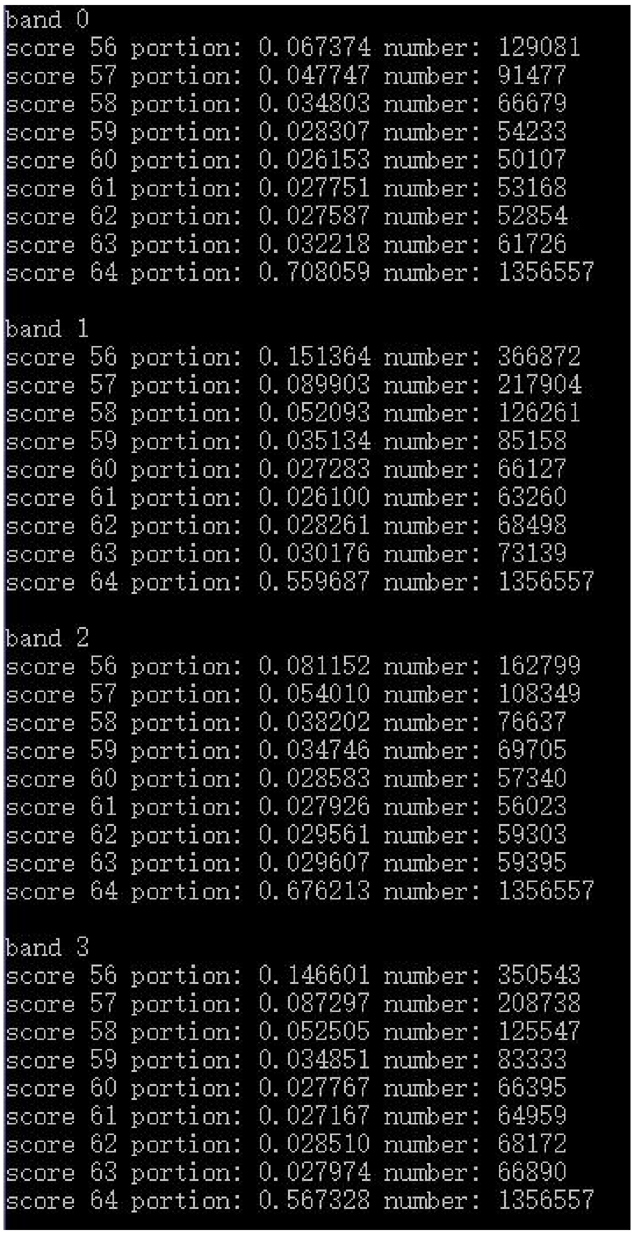
6 About the result

Original files: 60 WET files containing 2 million documents with total size of nearly 18 GB.

1. SimHash result

Processing time: 546 s

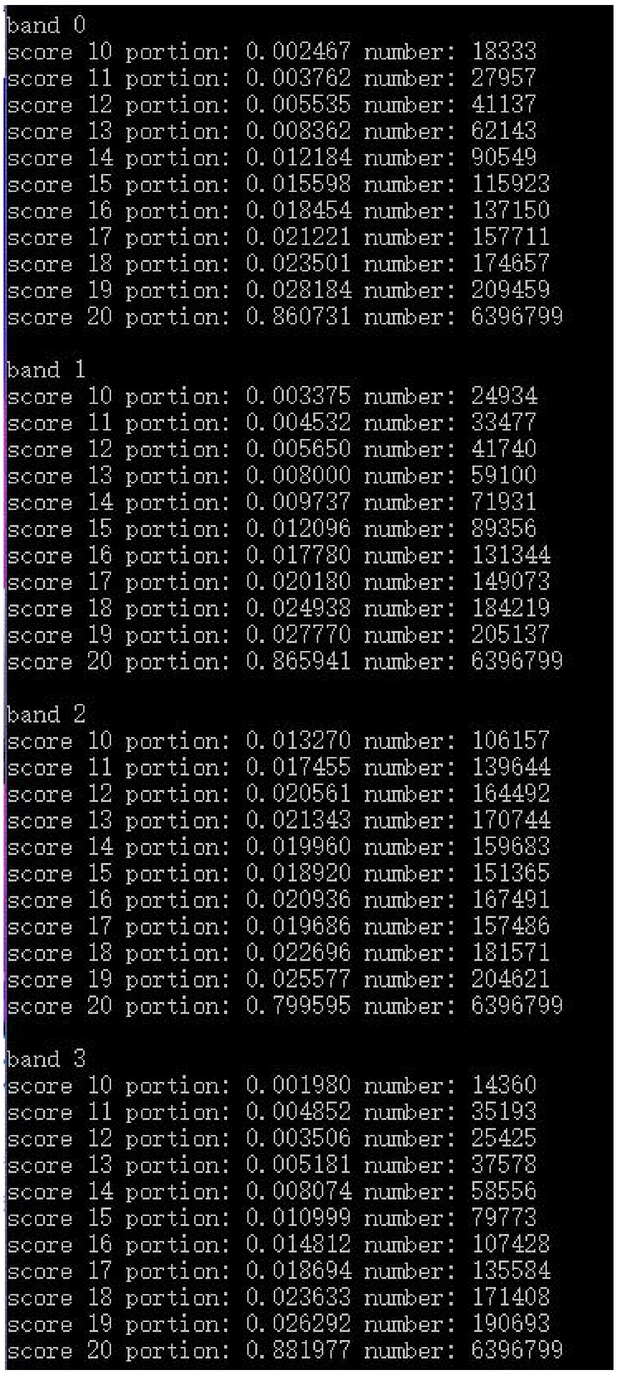
result file size: 1.5GB



```
band 0
score 56 portion: 0.067374 number: 129081
score 57 portion: 0.047747 number: 91477
score 58 portion: 0.034803 number: 66679
score 59 portion: 0.028307 number: 54233
score 60 portion: 0.026153 number: 50107
score 61 portion: 0.027751 number: 53168
score 62 portion: 0.027587 number: 52854
score 63 portion: 0.032218 number: 61726
score 64 portion: 0.708059 number: 1356557

band 1
score 56 portion: 0.151364 number: 366872
score 57 portion: 0.089903 number: 217904
score 58 portion: 0.052093 number: 126261
score 59 portion: 0.035134 number: 85158
score 60 portion: 0.027283 number: 66127
score 61 portion: 0.026100 number: 63260
score 62 portion: 0.028261 number: 68498
score 63 portion: 0.030176 number: 73139
score 64 portion: 0.559687 number: 1356557

band 2
score 56 portion: 0.081152 number: 162799
score 57 portion: 0.054010 number: 108349
score 58 portion: 0.038202 number: 76637
score 59 portion: 0.034746 number: 69705
score 60 portion: 0.028583 number: 57340
score 61 portion: 0.027926 number: 56023
score 62 portion: 0.029561 number: 59303
score 63 portion: 0.029607 number: 59395
score 64 portion: 0.676213 number: 1356557

band 3
score 56 portion: 0.146601 number: 350543
score 57 portion: 0.087297 number: 208738
score 58 portion: 0.052505 number: 125547
score 59 portion: 0.034851 number: 83333
score 60 portion: 0.027767 number: 66395
score 61 portion: 0.027167 number: 64959
score 62 portion: 0.028510 number: 68172
score 63 portion: 0.027974 number: 66890
score 64 portion: 0.567328 number: 1356557
```

Those pairs scoring 64 are mostly error pages or identical pages from the same website.
Those pairs scoring near 62 are usually pages having small different content but with same layout from the same website, while a small amount of pairs are not related.
Those pairs scoring under 60 have very poor similarity, most of them are totally different.

2. MinHash result
Processing time: 501 s
result file size: 5GB



```
band 0
score 10 portion: 0.002467 number: 18333
score 11 portion: 0.003762 number: 27957
score 12 portion: 0.005535 number: 41137
score 13 portion: 0.008362 number: 62143
score 14 portion: 0.012184 number: 90549
score 15 portion: 0.015598 number: 115923
score 16 portion: 0.018454 number: 137150
score 17 portion: 0.021221 number: 157711
score 18 portion: 0.023501 number: 174657
score 19 portion: 0.028184 number: 209459
score 20 portion: 0.860731 number: 6396799

band 1
score 10 portion: 0.003375 number: 24934
score 11 portion: 0.004532 number: 33477
score 12 portion: 0.005650 number: 41740
score 13 portion: 0.008000 number: 59100
score 14 portion: 0.009737 number: 71931
score 15 portion: 0.012096 number: 89356
score 16 portion: 0.017780 number: 131344
score 17 portion: 0.020180 number: 149073
score 18 portion: 0.024938 number: 184219
score 19 portion: 0.027770 number: 205137
score 20 portion: 0.865941 number: 6396799

band 2
score 10 portion: 0.013270 number: 106157
score 11 portion: 0.017455 number: 139644
score 12 portion: 0.020561 number: 164492
score 13 portion: 0.021343 number: 170744
score 14 portion: 0.019960 number: 159683
score 15 portion: 0.018920 number: 151365
score 16 portion: 0.020936 number: 167491
score 17 portion: 0.019686 number: 157486
score 18 portion: 0.022696 number: 181571
score 19 portion: 0.025577 number: 204621
score 20 portion: 0.799595 number: 6396799

band 3
score 10 portion: 0.001980 number: 14360
score 11 portion: 0.004852 number: 35193
score 12 portion: 0.003506 number: 25425
score 13 portion: 0.005181 number: 37578
score 14 portion: 0.008074 number: 58556
score 15 portion: 0.010999 number: 79773
score 16 portion: 0.014812 number: 107428
score 17 portion: 0.018694 number: 135584
score 18 portion: 0.023633 number: 171408
score 19 portion: 0.026292 number: 190693
score 20 portion: 0.881977 number: 6396799
```

Those pairs scoring 20 are mostly error pages or identical pages from the same website.
Those pairs scoring near 16 are mostly pages having small different content but with same layout from the same website.

Those pairs scoring under 12 have poor similarity, sometimes they don’t share anything in common.

3. Analysis on result

In my experiment result, unfortunately MinHash seems to be superior to SimHash on every aspects. MinHash can get more near duplicate candidates than SimHash. MinHash’s score is more credible than the one of SimHash. MinHash even runs a little faster than SimHash. The only disadvantage of MinHash is that it requires a lot more space than SimHash because a signature which is a sequence of number is stored with each document. When the number of documents reaches a high value, space used for MinHash will be very huge.

References

[1] wikipedia

[2] Caitlin Sadowski and Greg Levin. Hash-based Similarity Detection, 2007.

[3] Phuc-Tran Ho, Hee-Sun Kim and Sung-Ryul Kim. Application of Sim-Hash Algorithm and Big Data Analysis in Spam Email Detection System, 2014.

[4] Qunyan Zhang, Haixin Ma, Weining Qian and Aoying Zhou. Duplicate Detection for Identifying Social Spam in Microblogs, 2013.

[5] Ping Li. In Defense of MinHash Over SimHash, 2014.