Near Duplicate Detection Using Simhash

Sumon Biswas

December 13, 2017

Abstract

Near duplicate detection in a large collection of files is a well-studied problem in data science. Many Locality Sensitive Hashing (LSH) algorithms have been recently developed to solve this problem. Among them simhash is a very efficient LSH algorithm that uses probabilistic method to generate similar fingerprints for similar objects. We can store only the fingerprints of a large collection of objects and use those fingerprints to evaluate their similarity. In this report, we will discuss the details of simhash algorithm and its implementation. Hamming distance problem is another well studied problem that is closely related to simhash. We can find near duplicate objects solving Hamming distance problem efficiently on simhash fingerprints. In this report, we will also show how we can reduce the time to search low distant fingerprints without comparing all possible pairs.

1 Introduction

With the advancement of information technology, we have to store huge number of documents, web pages, images etc. Among all of these objects there are a substantial amount which are almost identical to each other. In many applications, we need to find the near duplicates and often the collection we want to search from is pretty big. A few popular applications of near duplicate detection are as follows.

Documents clustering: Searching similar documents is a very common application today. Suppose, we want to cluster published papers from ACM library based on the topic of interest. Or we want to find the nearly duplicate news published in last one month. In addition, plagiarism checking and spam detection applications are entirely dependent upon finding very similar documents.

Web mining: Suppose, we want to develop a focused crawler that will crawl through the web pages that are pertinent to a search query or keywords. To get this done, we have to find only similar web pages related to that topic. Several other web mining applications also use the same idea.

Recommendation system: E-commerce websites generally group their users based on the interest and purchase behavior of the users.

Computer vision: Near duplicate detection is not only applicable for text documents. We can also use this to find similar images while doing an image search.

In this report, we will introduce one popular LSH algorithm called simhash to find near duplicates from a large collection of text documents. We will also analyze the Hamming distance problem that is closely related to simhash.

2 Problem Formulation and Existing Solution

Suppose, we have millions of documents and given a new document we have to find all the near duplicates (e.g., 95% or more similar) from the collection in a reasonable amount of time. We can divide the problem into two parts: how to measure similarity between two documents and how to find the similar documents form a large collection efficiently? Therefore, our goal is to solve the following problems:

1. Given two documents D_a and D_b, what is the similarity measure between them?
2. Given a document D_a, find all the documents that are similar to D_a.
3. Identify all the pairs in the collection that are near duplicate of each other.

There are a few challenges related to the above problems. First, our algorithm should be designed for millions of documents. Second, the files should be compressed enough to fit in memory. Finally, the algorithm must be efficient to find near duplicate in small amount of time.

Different similarity measures have been used in the literature. Among them Euclidean similarity [SM08], Cosine similarity [HK06] and Jaccard similarity [CGGR08] are widely used to similarity measure. Suppose, A and B are two documents represented by set of features. Then the Jaccard similarity measures is given by:

$$\text{sim}(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

This similarity captures how many features are common in both of the documents. Cosine similarity measure is almost the same which captures the angle between two vectors:

$$\text{sim}(A, B) = \frac{|A \cap B|}{\sqrt{|A|.|B|}}$$

Locality sensitive Hashing (LSH): The general idea of hashing is that it maps all items in the collection to another uniformly distributed space. Therefore, two documents with a very small difference will be mapped to entirely different and random hash value. LSH uses the opposite idea. It maps similar items to similar hash value so that we can compare only the hash values to obtain the similarity measure between documents. A locality sensitive hashing scheme is a distribution on a family $\mathcal{F}$ of hash functions operating on a collection of objects, such that for two objects x and y,

$$\mathbf{Pr}_{h \in \mathcal{F}}[h(x) = h(y)] = \text{sim}(x, y)$$

Minhash and simhash are two popular LSH algorithms. Minhash resemblances the Jaccard similarity measure. In minhash, after converting the documents to set of integers, we take multiple permutation of those features. For each permutation we only select the minimum value of that permutation. Thus, by taking all minimum values we get the signature of that document. To compute similarity with other document we just compare the matches of values in their signature. The documents having many features in common will have many matches in their minwise permutation and it can be shown that the probability is exactly same as Jaccard similarity.

3 Simhash

Charikar's simhash [Cha02] is a dimensionality reduction technique which maps high dimensional documents to very small sized fingerprints. We can compute the Hamming distance of two fingerprints to measure the cosine similarity. Suppose, we have a collection of vectors in R^d on which we will develop the family of hash functions. We randomly select a random vector $\vec{r}$ from d dimensional Gaussian distribution. Let, we want to measure the similarity of two vectors $\vec{u}$ and $\vec{v}$. We define a hash function as follows:

$$h_{\vec{r}}(\vec{u}) = \begin{cases} 1, & \text{if } \vec{r}.\vec{u} \geq 0 \\ 0, & \text{if } \vec{r}.\vec{u} < 0 \end{cases}$$

Now, the probability of two hash values are equal can be given by:

$$\mathbf{Pr}[h_{\vec{r}}(\vec{u}) = h_{\vec{r}}(\vec{v})] = 1 - \frac{\theta(\vec{u}, \vec{v})}{\pi}$$

The above was used by Goemans and Williamson [GW95] in their rounding scheme for relaxation of MAX-CUT. We will choose a few random vectors $\vec{r}$ to generate a family of hash functions. Using this technique we are getting a slightly different measure of similarity that is correlated with cosine similarity. Suppose, $\vec{u}$ and $\vec{v}$ are vectors represented by the features of documents A and B respectively. Then the above LSH scheme is as follows:

$$\mathbf{Pr}[h(A) = h(B)] = 1 - \frac{\theta}{\pi}$$

$$\text{and } \theta = \cos^{-1}\left(\frac{|A \cap B|}{\sqrt{|A|.|B|}}\right)$$

Here, $\theta(\vec{u}, \vec{v})$ refers to the angle between two vectors $\vec{u}$ and $\vec{u}$. The function $1 - \frac{\theta(\vec{u},\vec{v})}{\pi}$ is closely related to the function $cos(\theta)$. In fact, it is always within a factor of 0.88 from actual cosine distance. In later part, we will empirically show that when the similarity measure is pretty high (e.g., 80% or more) we get a very good correlation between the above similarity measure and exact cosine similarity. Moreover, since we are getting the dot products of random vector and feature vector, it is easy to incorporate the weight of the features with the above similarity measure.

3.1 Algorithm

The basic sketch of using simhash algorithm to measure similarity is:

1. **Step 1:** Convert the document into set of features associated with weights.
2. **Step 2:** Create f-bit fingerprint for each document
3. **Step 3:** Calculate Hamming distance between two fingerprints to measure similarity between corresponding documents.

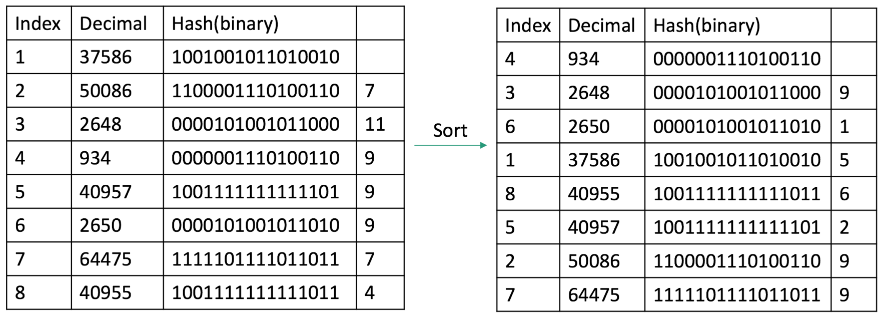
First, we have to choose features for each document. Feature selection also depends on the application. Word is a very obvious choice as features. We can also choose shingle as feature. A k-shingle is every k-length adjacent set of characters. Consider the sentence - "The earth is moving." The set of k-shingles for $k = 5$: {The_e, he_ea, e_ear, _eart, ...}. If the application requires such similarity measure that demands the order of appearance then shingle can be a good choice as feature. In this project, we used word as our feature. There are some preprocessing before converting them as set of features. We converted the whole document to lowercase as case sensitivity does not contribute to similarity score. Then we have removed the stop words (e.g., a, an, the etc.) and punctuation symbols which are common in every document. Next, the weight of each word is calculated. There are several ways to calculate the weight of each feature. The feature with more weight will contribute more to the similarity score. A very intuitive weight measure is the frequency of each term. The term which appears more in a document carries more weight. We can also use TF-IDF (Term Frequency-Inverse Document Frequency) as weight.

After the preprocessing is done, we create f-bit binary fingerprint of each document using simhash algorithm. The value of f is 32 or 64 in practice. First, each feature is converted to a f-bit binary hash value using a uniformly distributed hash function (e.g., MD5, FNV, Murmur). Then we define a vector of length f, initially with all zero values. Now, we iterate through each bit position (1 to f). If the bit position is 1 then we add the weight and if the bit position is 0 then we subtract the weight. After all the iterations, we get a vector of real values of length f. Finally, if the i_{th} value is negative we convert it to 0, otherwise we convert it to 1. Thus, we get f-bit fingerprint of a document [CS10].

Algorithm 1 Simhash (u)

```
 1: $V \leftarrow$ array of $b$ zeros
 2: for $i$ in $F(u)$ do
 3:     $\sigma_i \leftarrow$ UniformHash($i$)
 4:     for $j = 1$ to $b$ do
 5:         if $\sigma_{ij} \leftarrow 1$ then
 6:             $V[j] \leftarrow V[j] + w_i$
 7:         else
 8:             $V[j] \leftarrow V[j] + w_i$
 9: for $j = 1$ to $b$ do
10:     if $V[j] \geq 0$ then
11:         $V[j] \leftarrow 1$
12:     else$V[j] \leftarrow 0$
    return $V$
```

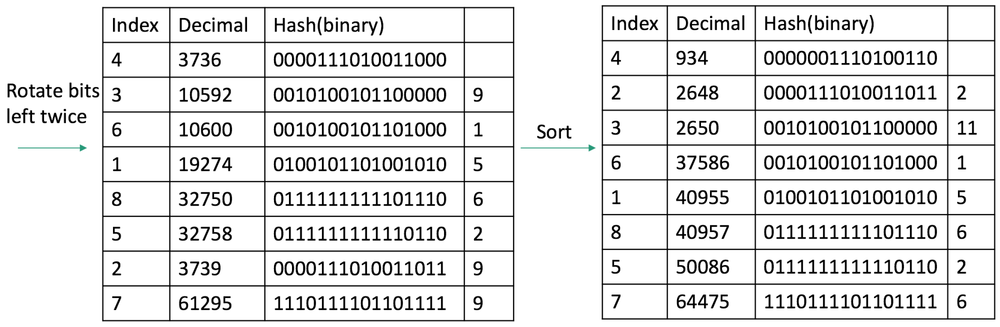
The runtime of the algorithm is $\bigodot(bn)$ where b is length of fingerprint. Note that for any length document we are generating binary fingerprint of length b which preserves the similarity measurement. If we have 1 billion web pages as a collection, we need less than 8 GB memory. Therefore, we can copy the entire collection in RAM and compare them to find similar ones.



Index	Decimal	Hash(binary)	
1	37586	1001001011010010	
2	50086	1100001110100110	7
3	2648	0000101001011000	11
4	934	0000001110100110	9
5	40957	1001111111111101	9
6	2650	0000101001011010	9
7	64475	1111101111011011	7
8	40955	1001111111111011	4

Index	Decimal	Hash(binary)	
4	934	0000001110100110	
3	2648	0000101001011000	9
6	2650	0000101001011010	1
1	37586	1001001011010010	5
8	40955	1001111111111011	6
5	40957	1001111111111101	2
2	50086	1100001110100110	9
7	64475	1111101111011011	9

Figure 1: Fingerprints are sorted and matched with the adjacent ones.


Rotate bits left twice
Sort

Index	Decimal	Hash(binary)	
4	3736	0000111010011000	
3	10592	0010100101100000	9
6	10600	0010100101101000	1
1	19274	0100101101001010	5
8	32750	0111111111101110	6
5	32758	0111111111110110	2
2	3739	0000111010011011	9
7	61295	1110111101101111	9

Index	Decimal	Hash(binary)	
4	934	0000001110100110	
2	2648	0000111010011011	2
3	2650	0010100101100000	11
6	37586	0010100101101000	1
1	40955	0100101101001010	5
8	40957	0111111111101110	6
5	50086	0111111111110110	2
7	64475	1110111101101111	6

Figure 2: Fingerprints are left-shifted twice and matched with the adjacent ones.

3.2 Hamming distance problem:

The final step of finding near duplicate is to calculate the Hamming distance of two fingerprints. Suppose, we have 1 million documents, hence 1 million fingerprints. We are given a particular fingerprint and we need to find the near duplicate of that document. Using trivial search, we need $\bigodot(n)$ comparisons. And if we need to find all near duplicate pairs from the whole collection, we need $\bigodot(n^2)$ comparisons. For a large volume of files, this is nearly impossible. So, researchers try to solve the Hamming distance problem in sublinear time in simhash fingerprint space.

The basic idea is to sort the fingerprints and only compare the adjacent ones. The fingerprints that only differs in their lower order bits will be adjacent. However, the fingerprints which differs mostly in their higher order bits are fall apart. We can combat the issue by taking multiple permutation of bit positions and sort them. We illustrate the solution by sketching the following example.

Let, we have 8 fingerprints in our collection and each fingerprint is 8-bit long. Our goal is to find all pairs having Hamming distance at most 3.

The rightmost column in figure 1 denotes the Hamming distance of current fingerprint with the immediately previous fingerprint. In this collection, pairs having Hamming distance 3 or less are: $H(3,6) = 1, H(8,5) = 2$ and $H(4,2) = 2$. By sorting the fingerprints, we got (3, 6) and (8, 5) adjacent. However, the pair (4, 2) has fallen apart. We will now permute the bit position and sot them again. In this case, we are left shifting all bits twice.

In figure 2, the pair (4, 2) are placed adjacently. Note that in this permutation the other two near duplicates are also placed adjacently. Therefore, by taking k permutation of bits we can find almost all the near duplicates in $\bigodot(kn)$ time.

Gurmeet et al proposed a technique called Block Permuted Hamming Search (BPHS) for online queries as well as batch queries [MS07]. The goal is to search over all fingerprints to match the query fingerprint within Hamming distance h. Now we divide the b-bits into G blocks so that each block has $\gamma = \frac{b}{G}$ bits. Now, select an integer g between 1 and $G - h$. To have distance h or less, two fingerprints must match g most significant blocks. We will use different choices of g blocks as header from the total G blocks. The rest of the blocks are tail blocks. The order of blocks in the

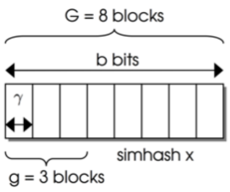


Figure 3: Block permuted Hamming Search.

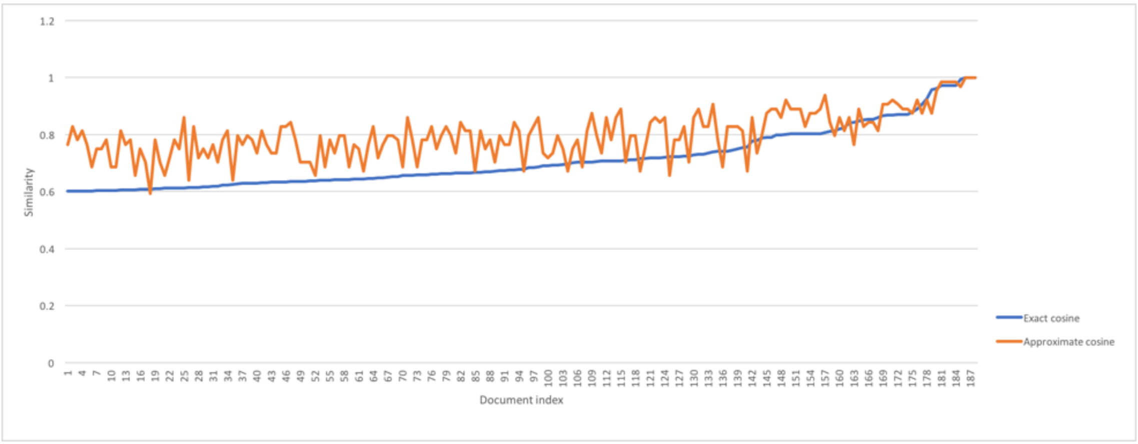


Figure 4: Corelation between actual cosine similarity and approximate cosine similarity calculated using simhash.

header and in the tail does not matter. So, we have $T = \binom{G}{g}$ sorted copies of fingerprints. We create T tables of fingerprints and perform Hamming search on them. To search near duplicate of fingerprint x, we perform binary search to find the numerically lowest fingerprint which has the same header as x. Now, we iterate one by one and calculate hamming distance only in the tail part. We stop when the first mismatch happens in the header. Thus, we can reduce the time of Hamming search and it can be showed that for appropriate choice of G and g, we can reduce the false negatives significantly [SL11].

4 Experiments

4.1 Simhash Implementation:

We have implemented a 64-bit simhash fingerprinting algorithm in Java. We have used Murmur hash as the uniformly distributed hash functions which is used in practice to implement simhash. The program takes any two documents as input, compute the fingerprint of each document and calculate Hamming distance between them to get the similarity measure. Suppose, the Hamming distance of two documents A and B is 3. Then the similarity measure is $1 - \frac{3}{64} = 95\%$. Then we design the program to take each pair in the collection and then compute the similarity.

Figure 4 depicts the correlation between exact cosine similarity and the approximate cosine similarity given by simhash algorithm. We can see a lot of deviation from the actual measure when the similarity is low to moderate. However, when the similarity is very high (above 85%) then the two similarity measures are almost same. We are using simhash to find near duplicates in the collection. So, we can ignore the deviation in case of low similarity measure. Moreover, since simhash is a probabilistic similarity measure there are some false positives in the result. We have run our program on thousand text documents and found 0.004% false positives.

Next, we implement the permuted hamming search in simhash space and found a significant improvement of time complexity. For 1000 documents, the pairwise comparisons take 625 seconds whereas for 25 different permutations it took only 2.62 seconds. However, in permuted hamming search we miss a few pairs which were supposed to be in the result. For $k = 25$ we get 11.26% false negatives.

5 Conclusion

In this project, we have implemented simhash algorithm to evaluate approximate cosine similarity between two documents from a large collection of files. We have preprocessed the documents, created word vectors with weight and then implemented simhash algorithm to generate 64-bit fingerprint of each document. Then we have implemented block permuted hamming search in our fingerprint space. Block permuted Hamming search helps us to reduce the time to find similar pairs significantly. However, we have to consider a few false negatives in this result. By designing the block permutation in a better way, we can reduce the false negative rate. The hamming distance solution still takes a substantial amount of time for million o documents which might be infeasible for web crawler. In addition, we need to store multiple copies of the fingerprints in memory. Therefore, Hamming search in simhash space is still an open problem.

References

[CGGR08] Jr. C. G. González, W. Bonventi and A. L. V. Rodrigues. Density of closed balls in real-valued and autometrized boolean spaces for clustering applications. *Proc. 19th Brazilian Symp. Artif. Intell., Savador, Brazil*, pages 8–22, 2008.

[Cha02] Moses S Charikar. Similarity estimation techniques from rounding algorithms. *Proceedings of the thiry-fourth annual ACM symposium on Theory of computing-ACM*, pages 380–388, 2002.

[CS10] Donald Metzler Croft, W. Bruce and Trevor Strohman. Search engines: Information retrieval in practice. *Reading: Addison-Wesley*, 283, 2010.

[GW95] M. X. Goemans and D. P. Williamson. Improved approximation algorithms for maximum cut and satisfiability problems using semidefinite programming. *JACM*, 42(6):1115–1145, 1995.

[HK06] J. Han and M. Kamber. Data mining: Concepts and techniques. *2nd ed. San Francisco, CA, USA: Morgan Kaufmann; Boston, MA, USA: Elsevier*, 2006.

[MS07] Arvind Jain Manku, Gurmeet Singh and Anish Das Sarma. Detecting near-duplicates for web crawling. *Proceedings of the 16th international conference on World Wide Web-ACM*, pages 141–150, 2007.

[SL11] Sadhan Sood and Dmitri Loguinov. Probabilistic near-duplicate detection using simhash. *20th ACM international conference on Information and knowledge management-ACM*, pages 1117–1126, 2011.

[SM08] T. W. Schoenharl and G. Madey. Evaluation of measurement techniques for the validation of agent-based simulations against streaming data. *Proc. ICCS, Kraków*, 2008.