Boğaziçi University
Software Engineering MS Program

Project Report, Fall 2011
Supervisor: A. Taylan Cemgil

Large-scale Document Similarity Search Application using Minhash

Final Report

January 16, 2012

Ertuğ Karamatlı

Revision History

Revision	Date	Explanation
1.0	11/28/2011	Draft version
2.0	01/09/2012	Presented version
3.0	01/16/2012	Final version

Table of Contents

List of Figures

List of Tables

1 Introduction

Finding similar documents have many applications such as detecting duplicates, detecting plagiarized work, making recommendations, and clustering. For example, search engines can use it to detect duplicate web pages in order to remove them from search results [3], an e-commerce site can implement a "more like this" feature, or a more advanced approach is to cluster users by their interests to make recommendations [5].

Finding similar documents in a large data set is a challenging task. One way to estimate similarity of two documents is calculating their Jaccard similarity by their set of words. Jaccard similarity is a commonly used set similarity metric. However, calculating Jaccard similarity pairwise in a large data set is not feasible due to its time complexity. Instead of calculating Jaccard similarity pairwise, it is possible to generate signatures for each of the sets in linear time and calculate their Jaccard similarity by using an estimator which also have linear time complexity. This technique is called Minhash. It is a locality sensitive hashing scheme to estimate Jaccard similarity of a pair of sets efficiently. In this report, Minhash technique is described, an application which can identify similar documents in a large data set is presented, and lastly, performance of the application is evaluated.

1.1 Locality Sensitive Hashing

Hashing is commonly used for tasks such as accelerating table lookup and checking data integrity. Hash functions are used to map large data sets to smaller "keys". Locality sensitive hashing (LSH) [11, 7, 4, 6, 2, 9] requires hash functions to map similar items to the same key. It is a method to perform probabilistic dimension reduction of high-dimensional data. Using this method, similar items can be found in linear time. It has many applications in large-scale learning such as near-duplicate detection, clustering, and similarity identification.

1.2 Jaccard Similarity

Jaccard similarity [8] is a measure of set similarity which is given in equation 1.

$$\mathrm{J}(A,B) = \frac{|A \cap B|}{|A \cup B|} \tag{1}$$

As an example, if $A = \{1,2,3\}$ and $B = \{1,2,4\}$ then $\mathrm{J}(A,B) = \frac{2}{4} = 0.5$.

2 Minhashing

Minhashing (min-wise independent permutations) [3, 5, 12, 10] is a locality sensitive hashing scheme for estimating similarity of a pair of sets. This technique allows generation of compressed signatures of sets. Instead of calculating Jaccard similarity for each pair, we generate signatures for each set. Similarity calculation is done in real-time by using only the signatures.

To describe how minhashing works, we need to visualize several sets as their characteristic matrix. The columns correspond to the sets and the rows correspond to elements of the universal set from which elements of the sets are selected. A value of 1 denotes set has the element and 0 indicates set does not contain the element. Characteristic matrix for sets $S_1 = \{b\}$, $S_2 = \{c\}$, and $S_3 = \{a, b, d\}$ are shown in Table 1.

$Element$	S_1	S_2	S_3
a	0	0	1
b	1	0	1
c	0	1	0
d	0	0	1

Table 1: A characteristic matrix

To calculate minhash of a set represented by a column of the characteristic matrix, pick a permutation of the rows. The minhash value of any column is the number of the first row, in the permuted order, in which the column has a 1.

$Element$	S_1	S_2	S_3
c	0	1	0
a	0	0	1
b	1	0	1
d	0	0	1

Table 2: A permutation of characteristic matrix rows in Table 1

For example, let us calculate minhash h of sets in Table 1. First, we pick a permutation of the characteristic matrix as shown in Table 2. Then, we look for the values of h by scanning from the top until we see a 1. Thus, we conclude that $h(S_1) = b$, $h(S_2) = c$, and $h(S_3) = a$.

2.1 Minhashing and Jaccard Similarity

It is very impractical to calculate permutations for sets and do the calculations described in the previous section. Fortunately, there is an interesting connection between Minhashing and Jaccard similarity which allows a very efficient computation.

Let h be a hash function. For any set S, x is a member of S with minimum value of h(x). Then, $h_{min}(S) = h(x)$. If $h_{min}(A) = h_{min}(B)$ then the item with minimum hash value x is contained in both sets. Interestingly, the probability of h_{min} values of two sets being equal is the same as their Jaccard similarity:

$$\Pr[h_{min}(A) = h_{min}(B)] = \frac{|A \cap B|}{|A \cup B|} \tag{2}$$

To understand why there is a connection, we need to restrict ourselves to the columns of two sets S_1 and S_2, then rows can be viewed as of one of the three types:

1. Type X rows have 1 in both columns.
2. Type Y rows have 1 in one of the columns and 0 in the other.
3. Type Z rows have 0 in both columns.

Since the characteristic matrix is usually sparse, i.e., universal set contains many other different elements, most rows are of type Z. Thus, what determines both $SIM(S_1, S_2)$ and the probability $h(S_1) = h(S_2)$ is the ratio of the numbers of type X and type Y rows. Let there be x rows of type X and y rows of type Y. Then $SIM(S_1, S_2) = x/(x + y)$. The reason is that x is the size of $S_1 \cap S_2$ and $x + y$ is the size of $S_1 \cup S_2$.

Let us consider the probability of $h(S_1) = h(S_2)$. If we think that the rows are permuted randomly, and we start scanning from the top, the probability that we shall see a type X row before we see a type Y row is $x/(x + y)$. But if the first row from the top other than a type Z row is a type X row, then $h(S_1) = h(S_2)$. Conversely, if the first row other than a type Z row is a type Y row, then the set with a 1 gets that row as its minhash value. However the set with a 0 in that row surely gets some row further down the permuted list. Thus, we know $h(S_1) = h(S_2)$ if we first see a type Y row. We conclude the probability that $h(S_1) = h(S_2)$ is $x/(x + y)$, which is also the Jaccard similarity of S_1 and S_2. Then using this probability, it is possible to construct an estimator to approximate the Jaccard similarity. Although, the variance is too high to compare only a single minhash value, when several minhash values are calculated and averaged, it provides a reasonable approximation.

2.2 Generating Signatures

In order to calculate similarity efficiently, we need smaller representations of large sets which is called “signatures”. The crucial property of signatures is that by only comparing the signatures of two sets, we should be able to estimate the Jaccard similarity of the respective sets. Although, it is not possible to calculate exact similarity from signatures, the estimates they provide are close and increasing the signature size leads to more accurate estimates.

We describe two algorithms to calculate set signatures. One uses many hash functions, while the latter uses a single hash function.

2.2.1 Algorithm with multiple hash functions

Let $H = h_1, h_2, ...h_n$ be a family of hash functions. In order to generate signatures for a set S, minimum hash values should be calculated for each hash function in H. To find minimum hash value for a hash function, calculate $h(s)$ for each element s in S and take the minimum value.

Algorithm 1 GenerateSignaturesWithMultipleHashFunctions

```
for i = 0 → |H| do
  hmin_i ← hmaxval
  for j = 0 → |S| do
    hval ← H_i(S_j)
    if hval < hmin_i then
      hmin_i ← hval
    end if
  end for
end for
return hmin
```

2.2.2 Algorithm with a single hash function

Let h be a hash function. To calculate signatures for a set S, calculate $h(s)$ for each element s in S and take n_h minimum $h(s)$ values.

Algorithm 2 GenerateSignaturesWithSingleHashFunction

```
for i = 0 → |S| do
  hval_i ← h(S_i)
end for
hmin ← n_h minimum hval values
return hmin
```

2.3 Approximating Similarity using Signatures

Due to the fact that the variance of using only a single signature (n_h) is too high to be useful, it is necessary to calculate many signatures and use their mean value. Increasing n_h allows more accurate approximations. Let Z_A and Z_B be signatures for sets A and B with cardinality n_h. Then similarity can be estimated as $\frac{|Z_A \cap Z_B|}{n_h}$.

3 System Description

An application which can identify similar documents in a large data set is designed and implemented. The application is implemented in Java programming language. We named the application *Benzer*.

The system is able to store documents with multiple field types such as text, boolean attributes, and images. A typical flow is as follows: First, a set of documents is indexed (signatures calculated and stored in the database), then, a document is given to the system to find similar documents. The system does a similarity search in the indexed documents and returns a set of similar documents.

3.1 Software Architecture

The application consists of three main components as given in Figure 1:

1. Client Application: An application that uses the API of Benzer.
2. Benzer: The main application.
3. Database: A database implementation.

The main application *Benzer* exposes an API (application programming interface) to a client application. Benzer and client application communicates via method calls. Benzer communicates to a database via method calls or via network, as defined by the specific database implementation.

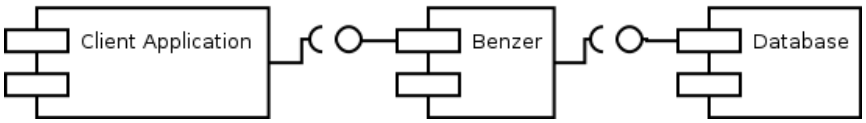


Figure 1: Component diagram of the application

3.2 Software Design

The application is designed using object-oriented design. Many parts of the system is pluggable thus allowing to switch between several possible algorithms. Strategy pattern is used to make algorithms configurable. It proved effective when evaluating performance of several different algorithms. Class diagram is given in Figure 2.

The application exposes a simple API consisting of 2 methods. *indexDocuments* adds documents to the database. *findSimilarDocuments* finds most similar documents to the given document.

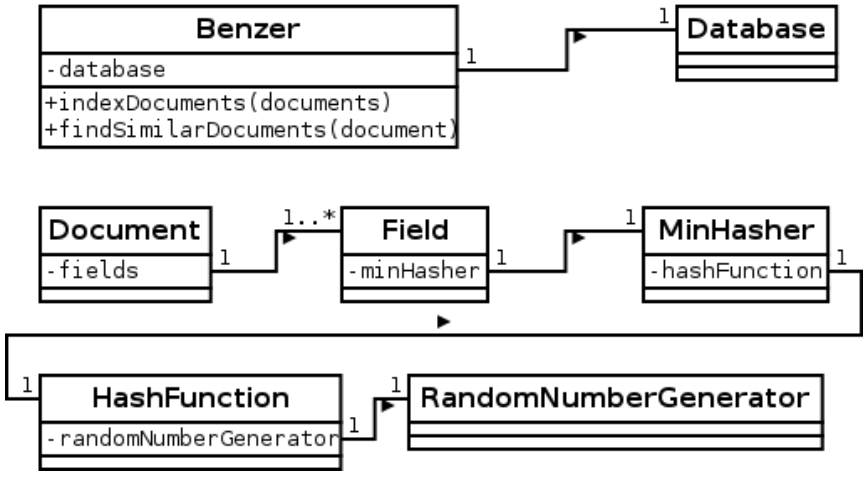


Figure 2: Class diagram of the application

3.3 Document Model

The document model is a representation of a document. Every document has an unique identifier and it can contain different types of fields where each field has a name associated with it. Fields help to change the document similarity problem into a set similarity problem which minhashing can be applied. In current implementation, a document can contain a set of values, text, and images but it is extendable and different field types can be defined. Each field can process its content differently. For example, text field can eliminate stop words and/or apply shingling to text. The only requirement for fields is to generate a set of hashes using a *minhasher*. Document model is given in Figure 3.

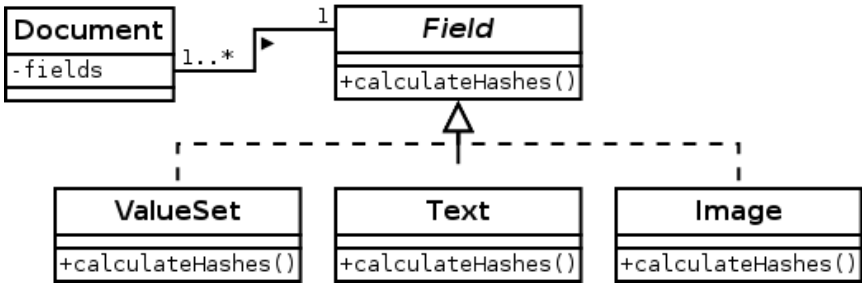


Figure 3: Document Model

3.3.1 ValueSet field

This is the most basic field type. It stores a set of strings and doesn't do any pre-processing. It can represent boolean attributes of the document. It is also useful for debugging and performance evaluations.

3.3.2 Text field

This field is intended for storing natural language text. It stores a string value and does some basic text processing such as tokenization.

3.3.3 Image field

This field is intended to store images. It stores an RGB representation of an image. It converts the image to grayscale and scales it down to 16x16 size.

3.4 Database

The application needs a database to store and query hashes. Namely, it defines 3 operations in the interface:

1. addHashes: Adds hashes of a document to the database.
2. getIds: Retrieves ids of the documents which matches at least one of supplied hashes.
3. getHashes: Retrieves all hashes of a document.

Database class diagram is given in Figure 4. There are two implementations of this interface.

3.4.1 In-memory database

It is the fastest database implementation but also it is limited by the available memory in the system which the application is running on. It maintains all data in hash tables.

3.4.2 Redis database

Redis is an open source key-value store which can contain strings, hashes, lists, sets and sorted sets [1]. This database implementation is more scalable than in-memory database because it can store data in a distributed fashion but slower due to network overhead.

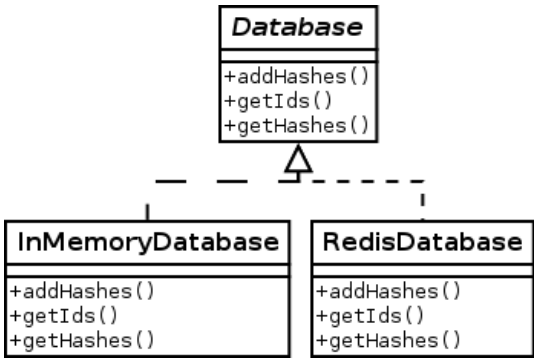


Figure 4: Class diagram of databases

3.5 Minhash Algorithms

Two minhash signature generation algorithms are implemented for signature generation as described in detail in section 2.2. Minhasher class diagram is given in Figure 5.

3.5.1 Multiple hash function minhash algorithm

This minhasher implementation takes a hash function count parameter n_h and initializes n_h hash functions from linear hash family [3]. To calculate minhashes, it returns the minimum hash value for each hash function while iterating through all elements of the set. To calculate similarity of two sets by signatures, it takes two set of signatures, calculates identical element count n_i, and returns n_i/n_h

3.5.2 Single hash function minhash algorithm

This minhasher implementation takes an element count parameter n_h and initializes a single hash functions from linear hash family [3]. To calculate minhashes, it calculates hash value for each element of the set, and returns the n_h minimum values as the signatures. To calculate similarity of two sets by signatures, it takes two set of signatures, calculates identical element count n_i, and returns n_i/n_h

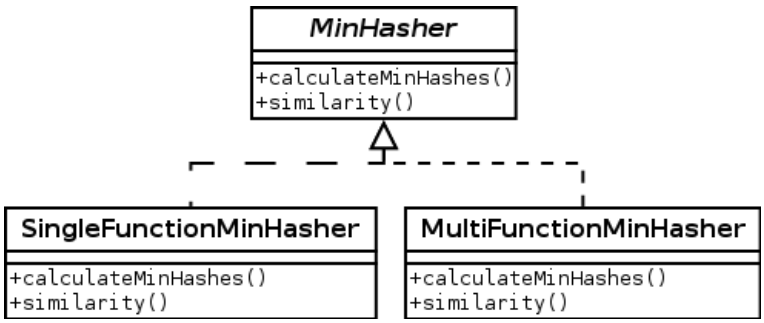


Figure 5: Class diagram of minhash algorithms

4 Evaluation

In order to evaluate the performance of the system, we need a reference to calculate the error rate. The reference is obtained by generating a pair of sets with a known similarity value. Then error was calculated comparing this value to the estimator result. It was calculated using RMS error as shown in equation 3 where s_e and s_r denotes estimated and real similarity, respectively. Each e_{rms} value is computed with 10 different s_e and s_r pairs ($n = 10$).

$$e_{rms} = \sqrt{\frac{\sum_{i=1}^{n}(s_{e,i} - s_{r,i})^2}{n}} \tag{3}$$

4.1 Experiment Setup

Although, the experiment is set up to evaluate the performance of the system, it effectively evaluates the performance of the minhashing technique. ValueSet field type is used to represent sets. Sets A and B with cardinality n are generated conforming to real similarity value s_r.

$$n = |A| = |B| \tag{4}$$

$$s_r = \frac{|A \cap B|}{|A \cup B|} \tag{5}$$

Sets A and B contains n_i identical elements.

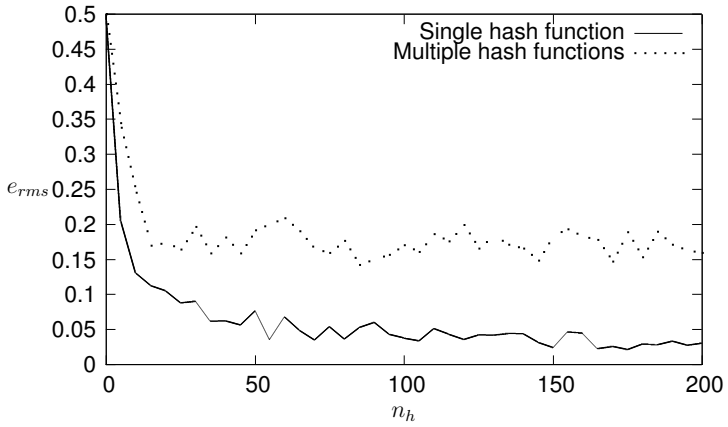
$$n_i = s_r n \tag{6}$$

$$n_d = n - n_i \tag{7}$$

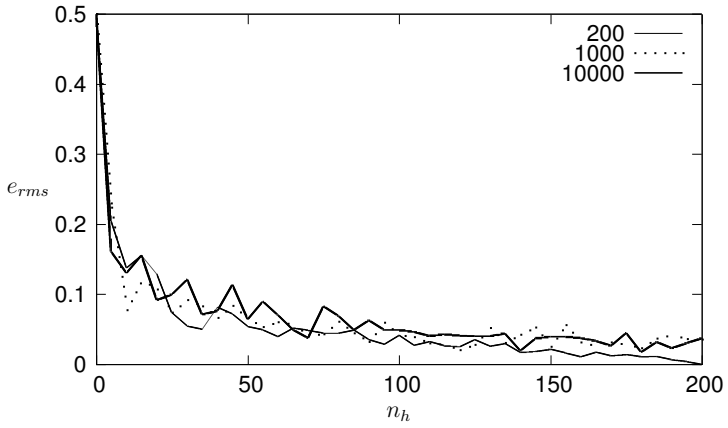
First, elements of A are generated randomly. Then, n_i of the elements of A are added to B. Lastly, n_d unique elements are generated randomly and added to B.

4.2 Results

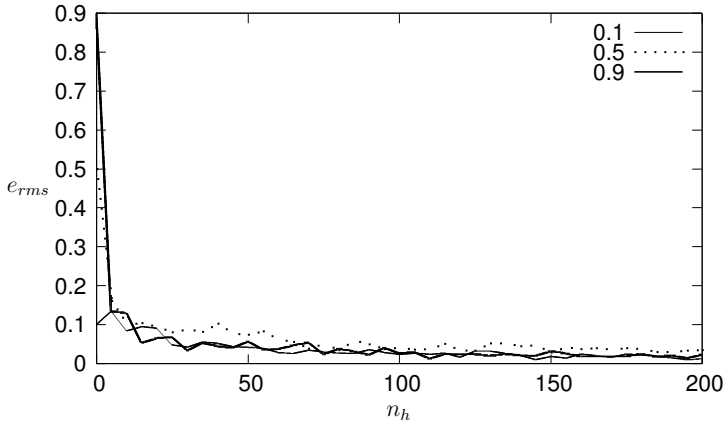
Variations of several parameters in the implementation are evaluated. As expected, when hash count n_h is increased, the estimation gets more accurate. Comparison of two minhash algorithms are shown in figure 6a. Version with a single hash function performs better than the version that uses multiple hash functions. Comparison of element counts n (200, 1000, 10000) are shown in figure 6b. When $n = h_n$, estimator reaches $e_{rms} = 0$. Variation of real similarities are shown in 6c but there is no considerable difference.



(a) Minhash algorithms



(b) Element counts (n)



(c) Real similarities (s_r)

Figure 6: RMS errors of different parameters with respect to hash count n_h

5 Conclusion and Future Work

Minhash is a locality sensitive hashing scheme for estimating similarity of a pair of sets. This technique allows calculating Jaccard similarity of sets efficiently. An application which can identify similar documents in a large data set is presented which implements minhashing. Finally, performance of the application is evaluated. Minhash implementation performed as expected, successfully approximating the Jaccard similarity of sets. When hash count n_h is increased, the estimation gets more accurate. The minhash algorithm with a single hash function for signature generation performed better than the multiple hash function version. This should be investigated in detail why there is a difference occured. By looking at the results of the experiment, it can be said that using the algorithm with the single hash function and a hash count n_h around 50 is a good choice, but it can change due to specific application requirements.

As a future work, it is possible to implement shingling for text field which will improve accuracy of text similarity detection. An advanced fingerprinting technique such as SIFT (Scale-invariant feature transform) for image field can be implemented. Additionally, performance evaluation of other field types such as text and image may be done. Also, the system can be improved to have a REST (Representational state transfer) API so that external 3rd party applications can be supported.

References

[1] Redis. Available from: `http://redis.io/`.

[2] A. Andoni and P. Indyk. Near-Optimal Hashing Algorithms for Approximate Nearest Neighbor in High. In *47th Annual IEEE Symposium on Foundations of Computer Science*, 2006.

[3] a. Broder. Min-Wise Independent Permutations. *Journal of Computer and System Sciences*, 60(3):630–659, June 2000. Available from: `http://linkinghub.elsevier.com/retrieve/pii/S0022000099916902`, `doi:10.1006/jcss.1999.1690`.

[4] M. S. Charikar. Similarity estimation techniques from rounding algorithms. *Proceedings of the thiry-fourth annual ACM symposium on Theory of computing - STOC '02*, page 380, 2002. Available from: `http://portal.acm.org/citation.cfm?doid=509907.509965`, `doi:10.1145/509961.509965`.

[5] A. Das, M. Datar, S. Rajaram, and A. Garg. Google News Personalization: Scalable Online. *Track: Industrial Practice and Experience*, pages 271–280, 2007. Available from: `http://www2007.org/papers/paper570.pdf`.

[6] M. Datar, N. Immorlica, P. Indyk, and V. S. Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. *Proceedings of the twentieth annual symposium on Computational geometry - SCG '04*, page 253, 2004. Available from: `http://portal.acm.org/citation.cfm?doid=997817.997857`, `doi:10.1145/997817.997857`.

[7] P. Indyk and R. Motwani. Approximate Nearest Neighbors : Towards Removing the Curse of Dimensionality NNS definition. In *Proceedings of the thirtieth annual ACM symposium on Theory of computing*, 1998.

[8] P. Jaccard. Étude comparative de la distribution florale dans une portion des Alpes et des Jura. *Bulletin de la Société Vaudoise des Sciences Naturelles*, 37:547–579, 1901.

[9] L. Paulevé, H. Jégou, and L. Amsaleg. Locality sensitive hashing: A comparison of hash function types and querying mechanisms. *Pattern Recognition Letters*, 31(11):1348–1358, Aug. 2010. Available from: `http://linkinghub.elsevier.com/retrieve/pii/S0167865510001169`, `doi:10.1016/j.patrec.2010.04.004`.

[10] A. Rajaraman and J. Ullman. *Mining of Massive Datasets*. 2010. Available from: `http://infolab.stanford.edu/~ullman/mmds.html`.

[11] Wikipedia. Locality-sensitive hashing. Available from: `http://en.wikipedia.org/wiki/Locality-sensitive_hashing`.

[12] Wikipedia. MinHash. Available from: `http://en.wikipedia.org/wiki/MinHash`.