Implementation of Locality Sensitive Hashing Techniques

Project Report submitted in partial fulfillment of the requirement for the degree of

Bachelor of Technology.

in

Computer Science & Engineering

under the Supervision of

Dr. Nitin Chanderwal

By

Srishti Tomar(111210)

to

Jaypee University of Information and TechnologyWaknaghat, Solan – 173234, Himachal Pradesh

Certificate

This is to certify that project report entitled “Implementaion of Locality Sensitive Hashing Techniques”, submitted by Srishti Tomar in partial fulfillment for the award of degree of Bachelor of Technology in Computer Science & Engineering to Jaypee University of Information Technology, Waknaghat, Solan has been carried out under my supervision.

This work has not been submitted partially or fully to any other University or Institute for the award of this or any other degree or diploma.

Date:

Supervisor’s Name: Dr. Nitin Chanderwal

Designation : Associate Professor

Acknowledgement

I am highly indebted to Jaypee University of Information Technology for their guidance and constant supervision as well as for providing necessary information regarding the project & also for their support in completing the project.

I would like to express my gratitude towards my parents & Project Guide for their kind co-operation and encouragement which help me in completion of this project.

I would like to express my special gratitude and thanks to industry persons for giving me such attention and time.

My thanks and appreciations also go to my colleague in developing the project and people who have willingly helped me out with their abilities

Date: Name of the student: Srishti Tomar

Table of Content

List of Figures

1.Abstract

Locality-sensitive hashing (**LSH**) is a method of performing probabilistic dimension reduction of high-dimensional data. The basic idea is to hash the input items so that similar items are mapped to the same buckets with high probability (the number of buckets being much smaller than the universe of possible input items). The hashing used in LSH is different from conventional hash functions, such as those used in cryptography, as in the LSH case the goal is to maximize probability of "collision" of similar items rather than avoid collisions. Locality-sensitive hashing, in many ways, mirrors data clustering and Nearest neighbor search.

The idea behind LSH is to construct a family of functions that hash objects into buckets such that objects that are similar will be hashed to the same bucket with high probability. Here, the type of the objects and the notion of similarity between them determine the particular hash function family. Typical instances include the Jaccard coefficient as similarity when the underlying objects are sets and the ℓ-2 norm as distance (i.e., dissimilarity) or the cosine/angle as similarity when the underlying objects are vectors.
LSH in nearest-neighbor applications can improve performance by significant amounts.

Locality Sensitive Hashing (LSH) is widely recognized as one of the most promising approaches to similarity search in high-dimensional spaces. Based on LSH, a considerable number of nearest neighbor search algorithms have been proposed in the past, with some of them having been used in many real-life applications. Apart from their demonstrated superior performance in practice, the popularity of the LSH algorithms is mainly due to their provable performance bounds on query cost, space consumption and failure probability.

2.Motivation

Locality-sensitive hashing helps to deal with the Curse of Dimensionality. The curse of dimensionality refers to various phenomena that arise when analyzing and organizing data in high-dimensional spaces (often with hundreds or thousands of dimensions) that do not occur in low-dimensional settings such as the three-dimensional physical space of everyday experience. There are multiple phenomena referred to by this name in domains such as numerical analysis, sampling, combinatorics, machine learning, data mining and databases. The common theme of these problems is that when the dimensionality increases, the volume of the space increases so fast that the available data become sparse. This sparsity is problematic for any method that requires statistical significance. In order to obtain a statistically sound and reliable result, the amount of data needed to support the result often grows exponentially with the dimensionality. Also organizing and searching data often relies on detecting areas where objects form groups with similar properties; in high dimensional data however all objects appear to be sparse and dissimilar in many ways which prevents common data organization strategies from being efficient. Thus the application of Locality-sensitive hashing can be useful in all of the above fields.

3.Introduction

The idea behind LSH is to construct a family of functions that hash objects into buckets such that objects that are similar will be hashed to the same bucket with high probability. Here, the type of the objects and the notion of similarity between them determine the particular hash function family. Typical instances include the Jaccard coefficient as similarity when the underlying objects are sets and the ℓ-2 norm as distance (i.e., dissimilarity) or the cosine/angle as similarity when the underlying objects are vectors. LSH in nearest-neighbor applications can improve performance by significant amounts.

3.1 Definition:

An LSH family F is defines for a metric space M= (M,d) , a threshold R>0 and an approximation factor c>1. This family F is a family of functions h : M → S which map elements from the metric space to a bucket s ϵ S. The LSH family of functions satisfies the following conditions for any two points p,q ϵ M, using the function h ϵ F which is chosen uniformly at random:

- if d(p,q) ≤ R, then h(p)=h(q) (ie, p and q collide) with probablity at least P1,
- if d(p,q) ≥ cR, then h(p)=h(q) with probability at most P2.

A family is interesting when P1> P2. Such a family F is called (R,cR,P1,P2)-sensitive.

Alternatively ,it is defined with respect to a universe of items U that have a similarity function Φ : U x U → [0,1]. An LSH scheme is a family of hash functions H coupled with a probability distribution D over the functions such that a function h ϵ H chosen according to D satisfies the property that Pr [h(a)=h(b)] = Φ(a,b) for any a,b ϵ U.

3.1.1Amplification:

Given a (d1,d2,p1,p2)-sensitive family F , we can construct new families G by either the AND-construction or OR-construction of F.

To create an AND-construction,we define a new family G of hash functions g, where each function g is constructed from k random functions h_1,...., h_k from F. We then say that for a hash function g ϵ G, g(x)=g(y) if and only if $h_i(x)=h_i(y)$ for i = 1,2,...,k.

Since the members of F are independently chosen for any g ϵ G , G is a (d_1,d_2,p_1^k,p_2^k)-sensitive family.

To create an OR-construction,we define a new family G of hash functions g, where each function g is constructed from k random functions h_1,...., h_k from F. We then say that for a hash function g ϵ G, g(x)=g(y) if and only if $h_i(x)=h_i(y)$ for one or more values of I.

Since the members of F are independently chosen for any g ϵ G , G is a $(d_1,d_2,(1-p_1)^k,(1-p_2)^k)$-sensitive family.

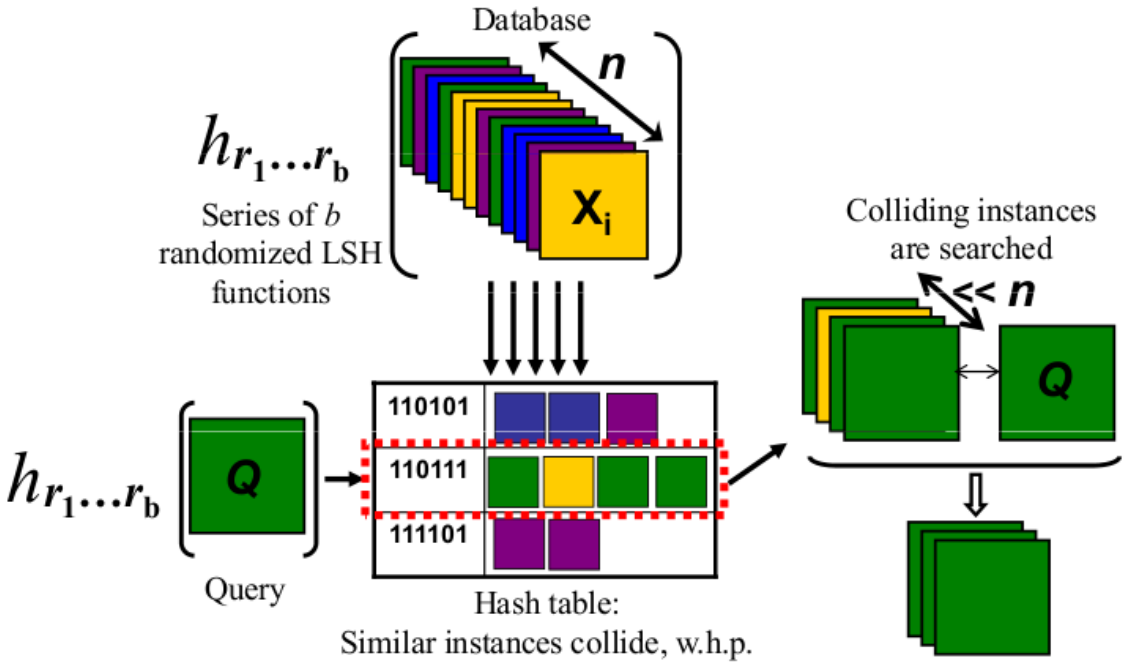

Figure 1. Querying using Hash Functions

3.2 Minhash:

MinHash (or the **min-wise independent permutations** locality sensitive hashing scheme) is a technique for quickly estimating how similar two sets are. The scheme was invented by Andrei Broder in (1997), and initially used in the AltaVista search engine to detect duplicate web pages and eliminate them from search results, It has also been applied in large-scale clustering problems, such as clustering documents by the similarity of their sets of words.

Suppose U is composed of subsets of some ground set of enumerable items S and the similarity function of interest is the Jaccard index J. If Π is a permutation on the indices of S, for A (subset of S) let $h(A) = \min_{a \in A} \{\Pi(a)\}$. Each possible choice of Π defines a single hash function h mapping input sets to elements of S.

Define the function family H to be the set of all such functions and let D be the uniform distribution. Given two sets A,B (both subsets of S) the event that h(A)=h(B) corresponds exactly to the event that the minimizer of Π over AUB lies inside intersection of the sets A,B. As h was chosen uniformly at random, $P_r[h(A)=h(B)]= J(A,B)$ and (H,D) define an LSH scheme for Jaccard Index.

Because the symmetric group on n elements has size n!, choosing a truly random permutation from the full symmetric group is infeasible for even moderately sized n. Because of this fact, there has been significant work on finding a family of permutations that is "min-wise independent" - a permutation family for which each element of the domain has equal probability of being the minimum under a randomly chosen Π . It has been established that a min-wise independent family of permutations is at least of size $\text{lcm}(1,2,...,n) \geq e^{n-o(n)}$ and that this bound is tight.

Because min-wise independent families are too big for practical applications, two variant notions of min-wise independence are introduced: restricted min-wise independent permutations families, and approximate min-wise independent families. Restricted min-wise independence is the min-wise independence property restricted to certain sets of cardinality at most k. Approximate min-wise independence differs from the property by at most a fixed ϵ .

3.3 Simhash:

SimHash is an algorithm created by Moses Charikar, from Google. It is an effective tool to compare easily and very fast two datasets. Computers are very pragmatic. They can find very fast if two elements are differents or not. It's a binary world: equal or different. Imagine now that we would like to have an idea of the similarity between these two elements. That would be a much bigger problem: a computer is not designed to do such comparison by nature. It's difficult, so it takes resources. o compare fingerprints, we need an algorithm that generate them using a bigger dataset. The firs idea would be to use hash (md5, sha1). However, these hash change if the input change a bit. We need another algorithm that change a bit if the text does not change a lot. Simhash does that.

3.4Nilsimsa Hash:

Nilsimsa is an anti-spam focused locality-sensitive hashing algorithm originally proposed the cmeclax remailer operator in 2001 and then reviewed by Damiani et al. in their 2004 paper titled, "An Open Digest-based Technique for Spam Detection". The goal of Nilsimsa is to generate a hash digest of an email message such that the digests of two similar messages are similar to each other. In comparison with cryptographic hash functions such as SHA-1 or MD5, making a small modification to a document does not substantially change the resulting hash of the document. Nilsimsa satisfies three requirements outlined by the paper's authors:

3.4.1 The digest identifying each message should not vary significantly (sic) for changes that can be produced automatically.

3.4.2 The encoding must be robust against intentional attacks.

3.4.3 The encoding should support an extremely low risk of false positives.

Nilsimsa similarity matching was taken in consideration by Jesse Kornblum when developing the fuzzy hashing in 2006, that used the algorithms of spamsum by Andrew Tridgell (2002).

Several implementations of Nilsimsa exist as open-source software in languages like c and python.

3.5 Random Projection:

The random projection method of LSH (termed arccos by Andoni and Indyk) is designed to approximate the cosine distance between vectors. The basic idea of this technique is to choose a random hyperplane (defined by a normal unit vector) at the outset and use the hyperplane to hash input vectors.

3.6 TLSH:

TLSH is locality-sensitive hashing algorithm designed for a range of security and digital forensic applications.The goal of TLSH is to generate a hash digest of document such that if two digests have a low distance between them, then it is likely that the messages are similar to each other.

Testing performed in the paper demonstrates that on a range of file types identified the Nilsimsa hash as having a significantly higher false positive rate when compared to other similarity digest schemes such as TLSH, Ssdeep and Sdhash.

An implementations of TLSH is available as open-source software.

3.7Applications:

LSH has been applied to several problem domains including

- Near-duplicate detection

- Hierarchical clustering
- Genome-wide association study
- Image similarity identification
- Gene expression similarity identification
- Audio similarity identification
- Nearest neighbor search
- Audio fingerprint
- Digital video fingerprinting

4.Minhash

MinHash (or the **min-wise independent permutations** locality sensitive hashing scheme) is a technique for quickly estimating how similar two sets are. The scheme was invented by Andrei Broder (1997), and initially used in the AltaVista search engine to detect duplicate web pages and eliminate them from search results. It has also been applied in large-scale clustering problems, such as clustering documents by the similarity of their sets of words.

Jaccard Similarity is a number between 0 and 1; it is 0 when the two sets are disjoint, 1 when they are equal, and strictly between 0 and 1 otherwise. It is a commonly used indicator of the similarity between two sets: two sets are more similar when their Jaccard index is closer to 1, and more dissimilar when their Jaccard index is closer to 0.

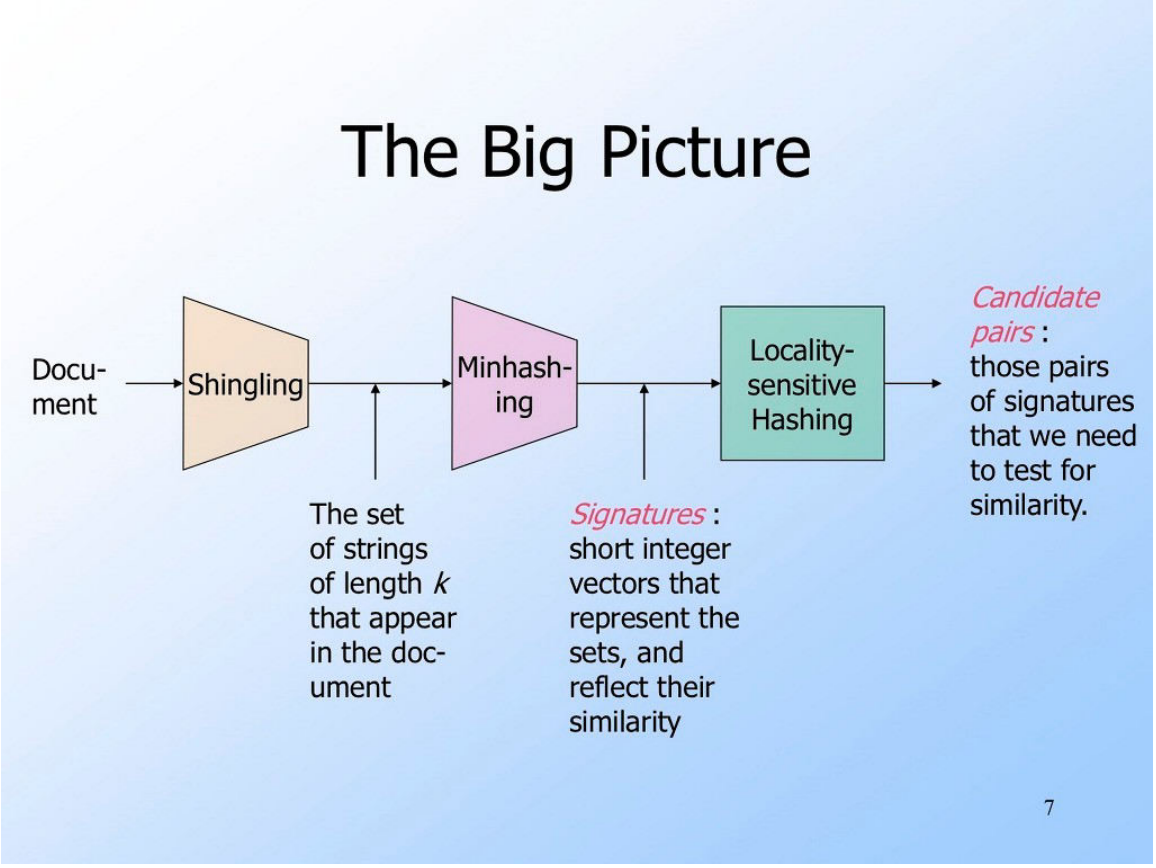

Figure 2: Hashing: Big Picture

Let h be a hash function that maps the members of A and B to distinct integers, and for any set S define $h_{\min}(S)$ to be the member x of S with the minimum value of $h(x)$. Then $h_{\min}(A) = h_{\min}(B)$ exactly when the minimum hash value of the union $A \cup B$ lies in the intersection $A \cap B$. Therefore,

$$\Pr[h_{\min}(A) = h_{\min}(B)] = J(A,B).$$

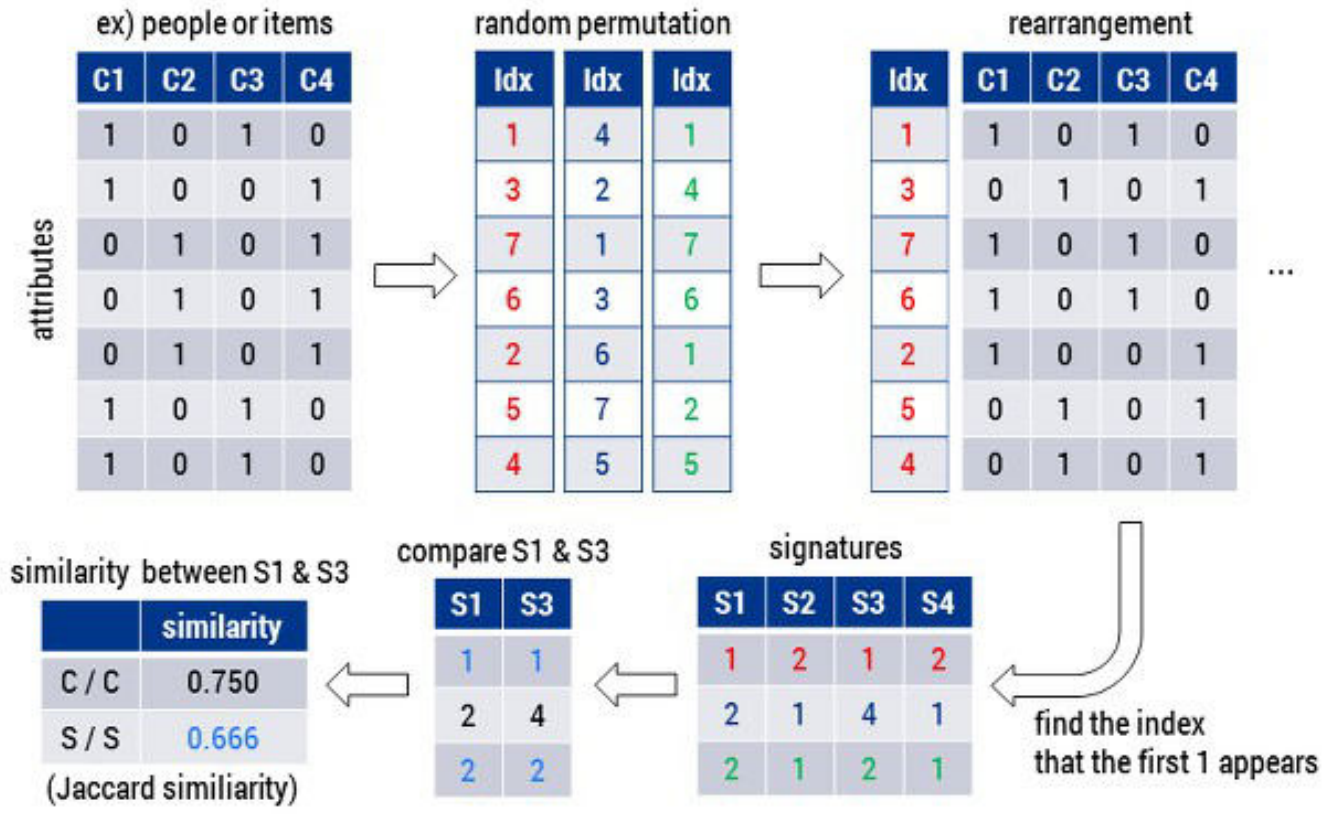

Figure 3. Minwise Hashing

In other words, if r is a random variable that is one when $h_{\min}(A) = h_{\min}(B)$ and zero otherwise, then r is an unbiased estimator of $J(A,B)$, although it has too high a variance to be useful on its own. The idea of the MinHash scheme is to reduce the variance by averaging together several variables constructed in the same way.

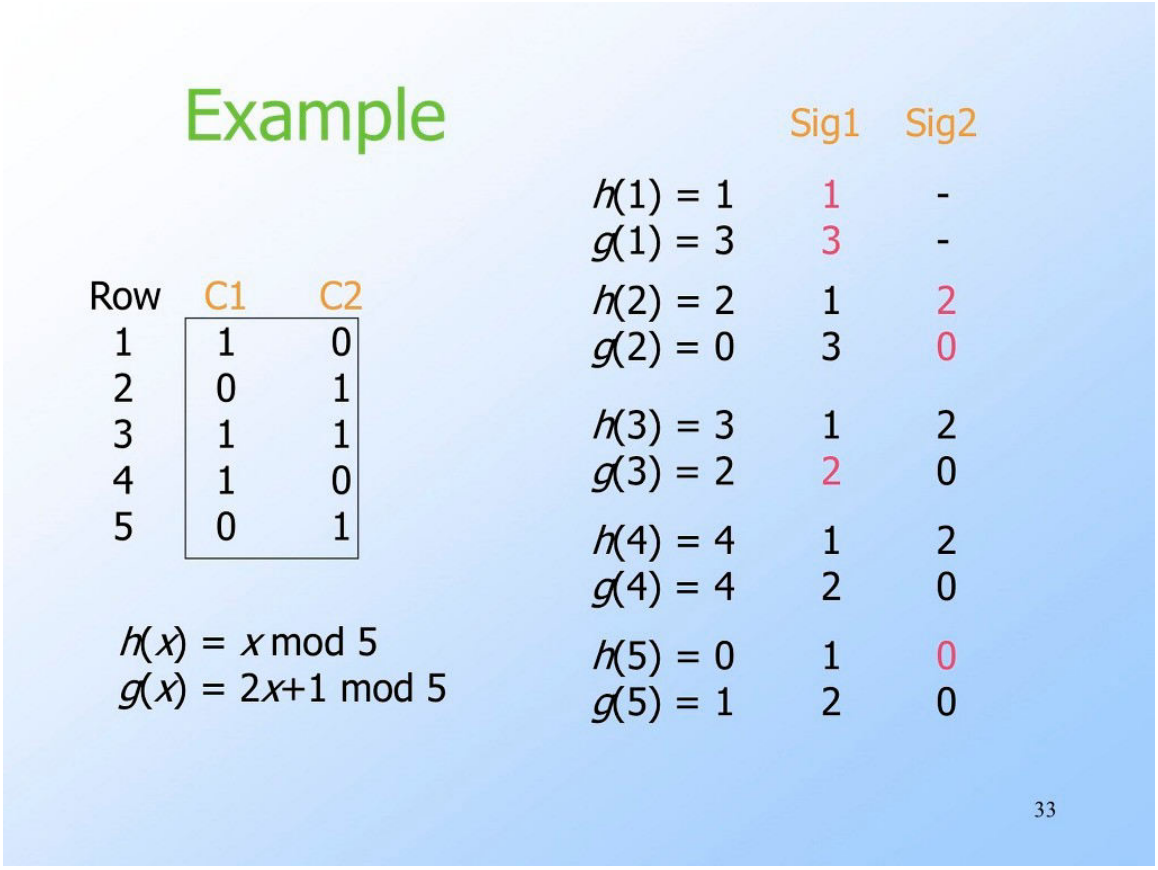

Figure 4. Minhash Example

4.1Variants:

4.1.1Many hash functions

The simplest version of the minhash scheme uses k different hash functions, where k is a fixed integer parameter, and represents each set S by the k values of $h_{\min}(S)$ for these k functions.

To estimate $J(A,B)$ using this version of the scheme, let y be the number of hash functions for which $h_{\min}(A) = h_{\min}(B)$, and use y/k as the estimate. This estimate is the average of k different 0-1 random variables, each of which is one when $h_{\min}(A) = h_{\min}(B)$ and zero otherwise, and each of which is an unbiased estimator of $J(A,B)$. Therefore, their average is also an unbiased estimator, and by standard Chernoff bounds for sums of 0-1 random variables, its expected error is $O(1/\sqrt{k})$.

Therefore, for any constant $\varepsilon > 0$ there is a constant $k = O(1/\varepsilon^2)$ such that the expected error of the estimate is at most ε. For example, 400 hashes would be required to estimate $J(A,B)$ with an expected error less than or equal to .05.

4.1.2Single hash function

It may be computationally expensive to compute multiple hash functions, but a related version of MinHash scheme avoids this penalty by using only a single hash function and uses it to select multiple values from each set rather than selecting only a single minimum value per hash function. Let h be a hash function, and let k be a fixed integer. If S is any set of k or more values in the domain of h, define $h_{(k)}(S)$ to be the subset of the k members of S that have the smallest values of h. This subset $h_{(k)}(S)$ is used as a *signature* for the set S, and the similarity of any two sets is estimated by comparing their signatures.

Specifically, let A and B be any two sets. Then $X = h_{(k)}(h_{(k)}(A) \cup h_{(k)}(B)) = h_{(k)}(A \cup B)$ is a set of k elements of $A \cup B$, and if h is a random function then any subset of k elements

is equally likely to be chosen; that is, X is a simple random sample of $A \cup B$. The subset $Y = X \cap h_{(k)}(A) \cap h_{(k)}(B)$ is the set of members of X that belong to the intersection $A \cap B$. Therefore, $|Y|/k$ is an unbiased estimator of $J(A,B)$. The difference between this estimator and the estimator produced by multiple hash functions is that X always has exactly k members, whereas the multiple hash functions may lead to a smaller number of sampled elements due to the possibility that two different hash functions may have the same minima. However, when k is small relative to the sizes of the sets, this difference is negligible.

By standard Chernoff bounds for sampling without replacement, this estimator has expected error $O(1/\sqrt{k})$, matching the performance of the multiple-hash-function scheme.

In order to implement the MinHash scheme as described above, one needs the hash function h to define a random permutation on n elements, where n is the total number of distinct elements in the union of all of the sets to be compared. But because there are $n!$ different permutations, it would require $\Omega(n \log n)$ bits just to specify a truly random permutation, an infeasibly large number for even moderate values of n. Because of this fact, by analogy to the theory of universal hashing, there has been significant work on finding a family of permutations that is "min-wise independent", meaning that for any subset of the domain, any element is equally likely to be the minimum. It has been established that a min-wise independent family of permutations must include at least different permutations, and therefore that it needs $\Omega(n)$ bits to specify a single permutation, still infeasibly large.

Because of this impracticality, two variant notions of min-wise independence have been introduced: restricted min-wise independent permutations families, and approximate min-wise independent families. Restricted min-wise independence is the min-wise independence property restricted to certain sets of cardinality at most k. Approximate min-wise independence has at most a fixed probability ε of varying from full independence.

The original applications for MinHash involved clustering and eliminating near-duplicates among web documents, represented as sets of the words occurring in those documents. Similar techniques have also been used for clustering and near-duplicate elimination for other types of data, such as images: in the case of image data, an image can be represented as a set of smaller sub-images cropped from it, or as sets of more complex image feature descriptions.

In data mining, Cohen et al. use MinHash as a tool for association rule learning. Given a database in which each entry has multiple attributes (viewed as a 0-1 matrix with a row per database entry and a column per attribute) they use MinHash-based approximations to the Jaccard index to identify candidate pairs of attributes that frequently co-occur, and then compute the exact value of the index for only those pairs to determine the ones whose frequencies of co-occurrence are below a given strict threshold.

The MinHash scheme may be seen as an instance of locality sensitive hashing, a collection of techniques for using hash functions to map large sets of objects down to smaller hash values in such a way that, when two objects have a small distance from each other, their hash values are likely to be the same. In this instance, the signature of a set may be seen as its hash value. Other locality sensitive hashing techniques exist for Hamming distance between sets and cosine distance between vectors; locality sensitive hashing has important applications in nearest neighbor search algorithms. For large distributed systems, and in particular MapReduce, there exist modified versions of MinHash to help compute similarities with no dependence on the point dimension.

4.2Algorithm Used for comparing minhashes:

```
for each row r do
 begin
 compute h(r)
 for each column c do
  if c has 1 in row r
  if h(r) is smaller than M(i,c) then
   M(i,c)=h(r)
```

5. Simhash

A hash function usually hashes different values to totally different hash values simhash is one where similiar items are hashed to similiar hash values(by similar we mean the bitwise hamming distance between hash values).

Simhash is useful because if the simhash bitwise hamming distance of two phrases is low then their jaccard coefficient is high. In the case that two numbers have a low bitwise hamming distance and the difference in their bits are in the lower order bits then it turns out that they will end up close to each other if the list is sorted.

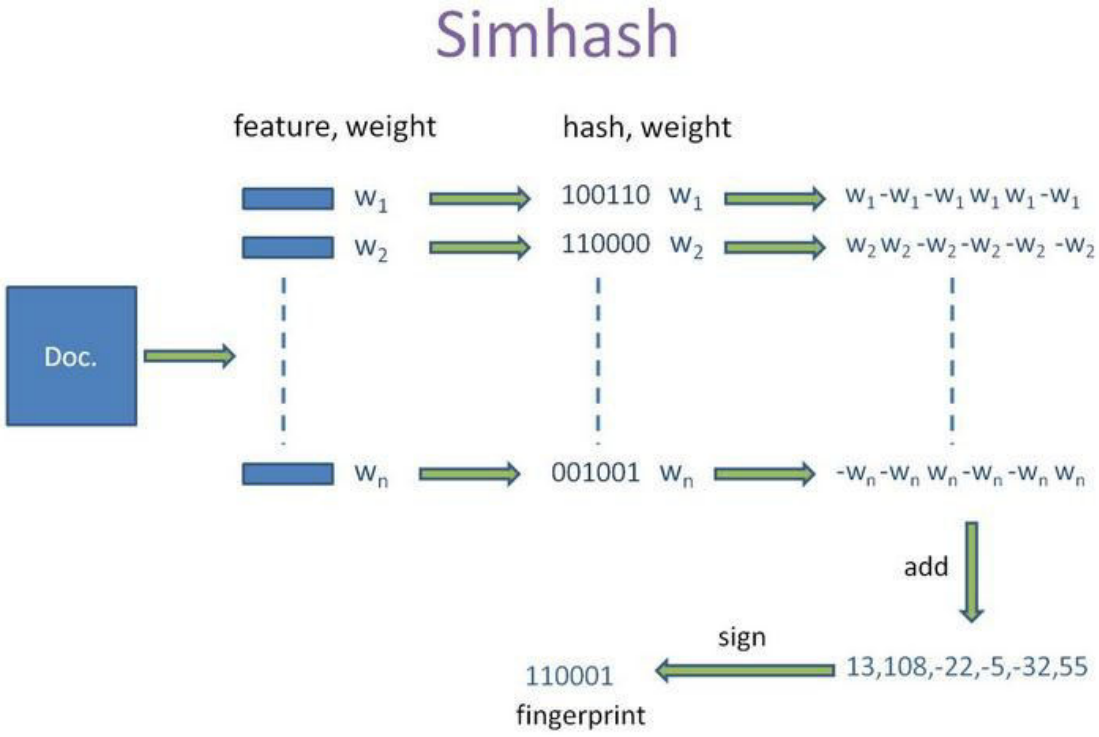

Figure 5. Simhash

SimHash is an algorithm created by Moses Charikar, from Google. It is an effective tool to compare easily and very fast two datasets. Computers are very pragmatic. They can find very fast if two elements are differents or not. It's a binary world: equal or different.

Imagine now that we would like to have an idea of the similarity between these two elements. That would be a much bigger problem: a computer is not designed to do such comparison by nature. It's difficult, so it takes resources.

The computer will find if the texts are equals (strictly) by browsing the first text and see if the letter in its current position is the same in the other text.

But the point here is that if the computer find a difference, it stops. It does not use anymore resources. It knows that the strings are strictly different.

That's where similarity is a challenge. If we check for strict equality, we only need to stop on the first difference. If we want a similarity estimation, we have to check the entire text. In math, similarity is:

$$similarity(A,B)=A \cap BA \cup B$$

For big datasets, the part

$$A \cap B$$

take some time to compute. That's where SimHash is useful.

With SimHash, we will create a **fingerprint** that will replace the datasets `A` and `B`:

$$simhash(A) \cap simhash(A)$$

Thus we will compare much smaller elements, the comparison time will be dramatically reduced.

To compare fingerprints, we need an algorithm that generate them using a bigger dataset. The firs idea would be to use hash (md5, sha1). However, these hash change if the input

change a bit. We need another algorithm that change a bit if the text does not change a lot. Simhash does that.

The official SimHash algorithm is:

- Define a fingerprint size (for instance 32 bits)
- Create an array `V[]` filled with this size of zeros
- For each element in the dataset, we create a unique hash with md5, sha1 of any other hash algorithm that give same-sized results
- For each hash, for each bit i in this hash
    - If the bit is 0, we add 1 to `V[i]`
    - If the bit is 1, we take 1 to `V[i]`
- For each i
    - If `V[i]` > 0, i = 1
    - If `V[i]` < 0, i = 0

It gives us a fingerprint characterizing our text, an approximation of the text data.

A fingerprint is in fact a binary number, for instance:
`10101011100010001010000101111100`. Now, to find

$$simhash(A) \cap simhash(A)$$

we only have to use a XOR operation, by example:

```
--- 10101011100010001010000101111100
XOR 10101011100010011110000101111110
  = 00000000000000001010000000000010
```

Here, the `1` in the XOR result are the differences between the two fingerprints. To get an idea of the difference between the original texts, we juste have to count the number of `1` and divide it by the total size.

We have 3 ones for 32 characters, so we have 3 differences per 32 elements : the estimation of the difference is `3 / 32 = 0,09375`.

And for the similarity: `1 - 3 / 32 = 0,90625`, a bit more than 90%. We have a similarity index!

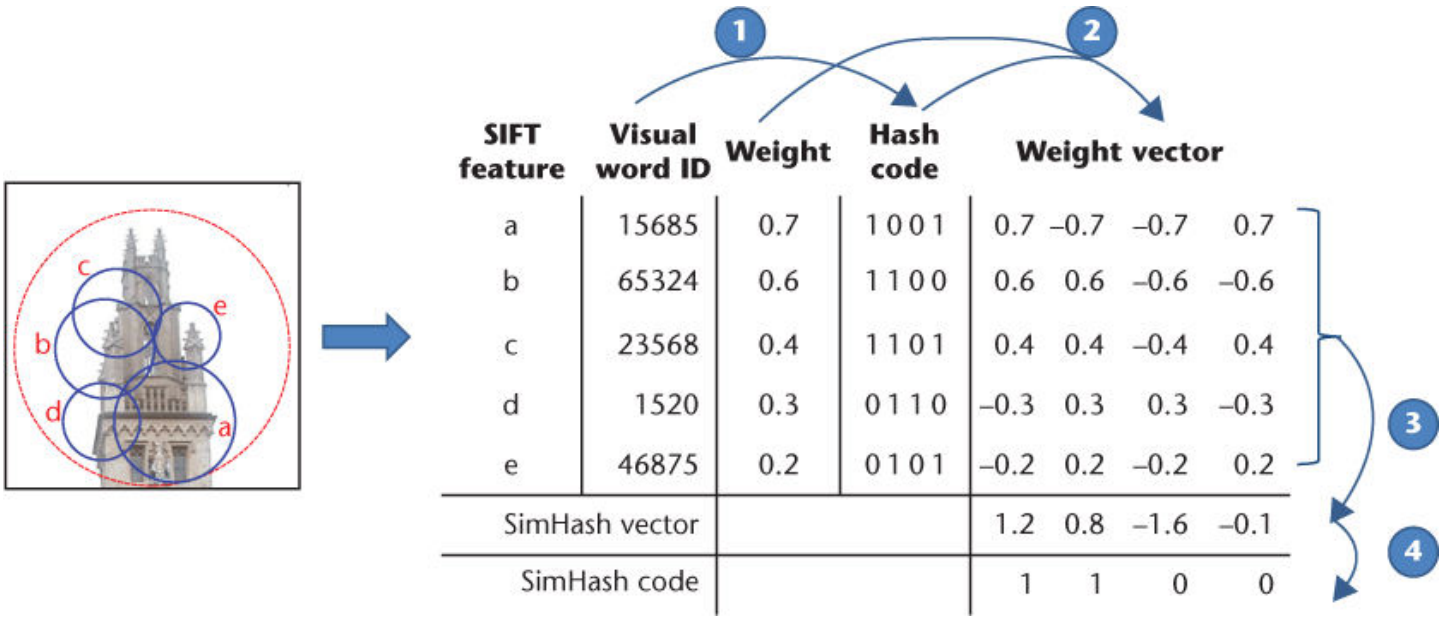

Figure 6. Simhash Example

Algorithm *simhash(doc, n)*
doc: document for which simhash is computed
n: length of the desired hash size in bits

1. *doc* is split into tokens (words for example) or super-tokens (word tuples)
2. weights are associated with tokens (for example: frequency count)
3. *v* = vector of size n, initialized to 0
4. **for each** *token* **in** *doc*
5. *token_hash* = make_n-bit_simple_hash (token)
6. **for** i = 1 **to** n **do**
7. **if**(i^{th} bit of *token_hash* == 1)
8. *v[i]* = *v[i]* + weight(token)
9. **else**
10. *v[i]* = *v[i]* - weight(token)
11. *bit_vector* = vector of size n, initialized to 0
12. **for** i = 1 **to** n **do**
13. **if**(*v[i]* > 0)
14. *bit_vector [i]* = 1

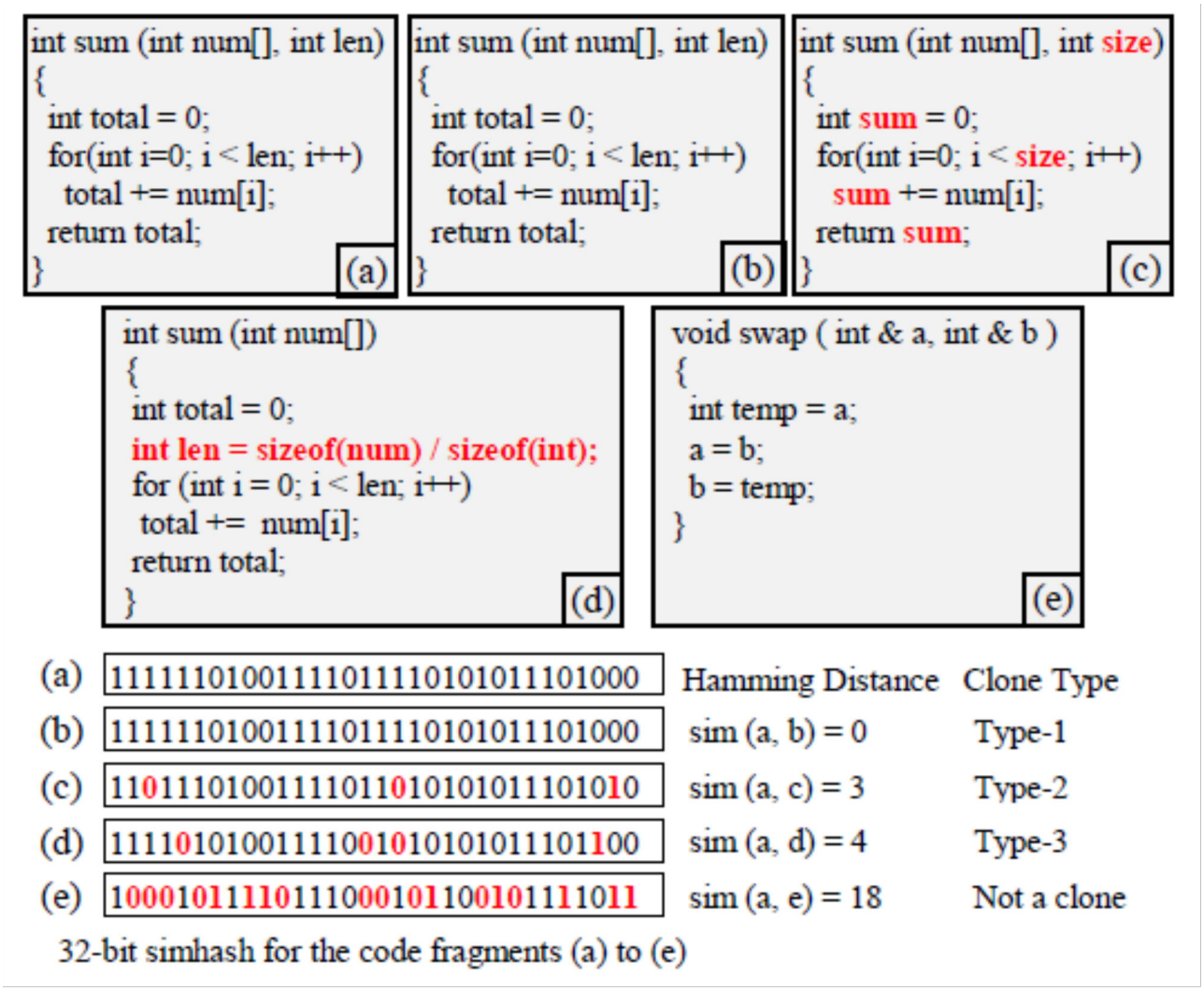

Figure 7. 32-bit Simhash for the code fragments

SimHash is currently used by Google to compare page with its database, to avoid duplicate contents.

But the main usage of SimHash is to compare things in a database. For instance, let's imagine we want to find the most similar articles of the one we are currently reading. It appears complicated at the first sight using only SQL. However with SimHash it's not that difficult: we just have to store a fingerprint for each aticle, and use the XOR operation in SQL to count the 1s in the binary result.

Most hash functions are used to separate and obscure data, so that similar data hashes to very different keys. We propose to use hash functions for the opposite purpose: to detect similarities between data. Detecting similar and classifying documents is a well-studied problem, but typically involves complex heuristics and/or O(n2) pair-wise comparisons. Using a hash function that hashed similar to similar values, similarity could be determined simply by comparing pre-sorted hash key values. The challenge is to find a similarity hash that minimizes false positives. We have implemented a family of similarity hash functions with this intent. We have further enhanced their performance by storing the auxiliary data used to compute our hash keys. This data is used as a second lteration after a hash key comparison indicates that two are potentially similar. We use these tests to explore the notion of “similarity”.

As storage capacities become larger it is increasingly difficult to organize and manage growing file systems. Identical copies or older versions of file often become separated and scattered across a directory structure. Consolidating or removing multiple versions of a file becomes desirable. However, duplication technologies do not extend well to the case where file are not identical. Techniques for identifying similar file could also be useful for classification purposes and as an aid to search. A standard technique in similarity detection is to map features of a file into some high dimensional space, and then use distance within this space as a measure of similarity. Unfortunately, this typically involves computing the distance between all pairs of file, which leads to O(n2)

similarity detection algorithms. If these file-to-vector mappings could be reduced to a one-dimensional space, then the data points could be sorted in O(n log n) time, greatly increasing detection speed. Typically, hash functions are designed to minimize collisions (where two different inputs map to the same key value). With cryptographic hash functions, collisions should be nearly impossible, and nearly identical data should hash to very different keys."Similarity" is a vague word, and can have numerous meanings in the context of computer files. We take the view that in order for two files to be similar they must share content. However, there are different ways to define that sharing. For example, the content of a file, Take a text file encoded in rtf as an example. Content could refer to the entire file, just the text portion of the file (not including rtf header information), or the semantic meaning of the text portion of the file (irrespective of the actual text).Many previous attempts at file similarity detection have focused on detecting similarity on the text level. We decided to use binary similarity as our metric. Two file are similar if only a small percentage of their raw bit patterns are different. This often fails to detect other types of similarity. For example, adding a line to source code might shift all line numbers within the compiled code. The two source file would be detected as similar under our metric; the compiled results would not. We decided on binary similarity because we did not want to focus on one particular file type (e.g. text documents) or structure. Another issue we do not explore is that of semantic similarity. For example, two text files may use different words but contain the same content in terms of meaning. Or, two MP3 files of the same song with different encodings may result in completely different binary content. The focus is on syntactic, not semantic, similarity. In the words of Udi Manber, no effort is made to understand the contents of the files. Broder made clear the distinction between resemblance (when two files resemble each other) and containment (when one file is contained inside of another). As an example of a containment relationship, take the case where one file consists of repeated copies of another smaller file. The focus of SimHash has been on resemblance detection. Two file with a size disparity (as in the example above) are implicitly different; containment relationships between files do not necessarily make two files 'similar' under our metric.

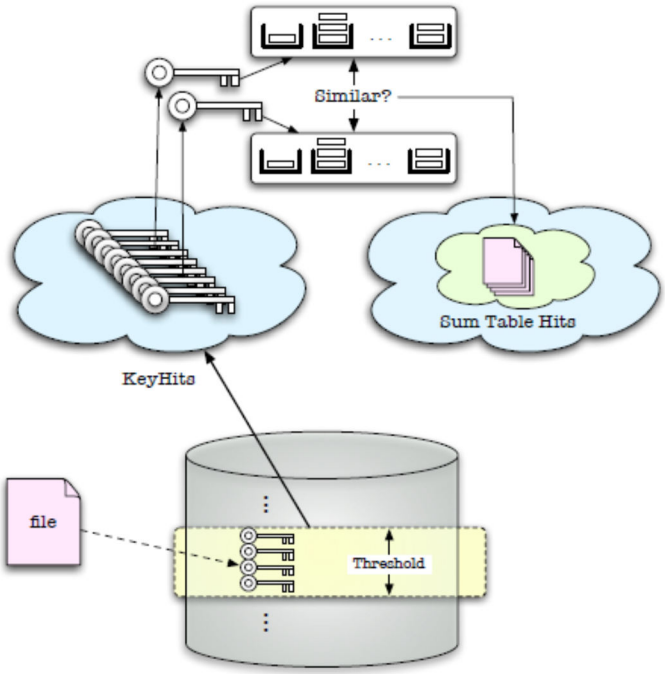

Figure 8. Simhash in databases

In order for files to be similar under our type of metric, they must contain a large number of common pieces. Another dividing point of techniques is the granularity and coverage of these pieces. SimHash operates at a very fine granularity, specifically byte or word level. Complete coverage is not attempted; only care about the portions of the file which match our set of bit patterns. Given some similarity metric, there needs to be a threshold to determine how close within that metric files need to be to count as similar. Focus is on files which have a strong degree of similarity,ideally within 1-2% of each other. Another issue is whether a form of similarity detection is meant to operate on a relative or absolute level. In other words, is the focus retrieving a set of similar to a given file, or retrieving all pairs of similar files. SimHash does both.The problem of identifying similarity is not a new one, although no one seems to have discovered a consistently good general solution. The most relevant paper to SimHash is over ten years old . There has also been a body of research focusing on redundancy elimination or deduplication. Much of the research on similarity detection since then has focused on very specific applications and filetypes. This includes:

technical documentation

software systems

plagiarism detection

music

web pages

In most cases, the main goal of redundancy elimination is a reduction in either bandwidth or storage. Redundancy elimination can focus on eliminating multiple copies of the same

file, or else preventing repeats of specific blocks shared between files. The standard way to identify duplicate blocks is by hashing each block.
As files are modified, new copies of modified blocks are written to disk, without changing references to unmodified blocks. Shared or unmodified blocks are identified by comparing hashes of the blocks within a file before writing to disk.

A natural question when classifying blocks is how to identify block boundaries. The options for this include fixed size chunking (for example, filesystem blocks), fixed size chunking over a sliding window, or some form of dynamic content-based chunking . Content-defined chunking consistently outperforms fixed sized chunking at identifying redundancies, but involves larger time and space overheads .Instead of coalescing repeated blocks, delta encoding works at a finer granularity. Essentially, it uses the difference (or delta) between two files to represent the second one. This is only effective when the two files resemble each other closely Different versions in a version control system is a good example.

6.Nilsimsa Hash

Nilsimsa is an anti-spam focused locality-sensitive hashing algorithm originally proposed the cmeclax remailer operator in 2001 and then reviewed by Damiani et al. in their 2004 paper titled, "An Open Digest-based Technique for Spam Detection". The goal of Nilsimsa is to generate a hash digest of an email message such that the digests of two similar messages are similar to each other. In comparison with cryptographic hash functions such as SHA-1 or MD5, making a small modification to a document does not substantially change the resulting hash of the document. Nilsimsa satisfies three requirements outlined by the paper's authors:

1. The digest identifying each message should not vary significantly (sic) for changes that can be produced automatically.

2. The encoding must be robust against intentional attacks.

3. The encoding should support an extremely low risk of false positives.

6.1Origin of requirements:

If a generic hash function like MD5 (or SHA-1) is used to produce the digests of a message, the spammer can easily fool this protection measure by inserting into each message, in an arbitrary position, a few random characters(called hash busters) that will immediately make the message unique, with practically no impact on the user perception of the message. This observation gives birth to the first requirement given above which is:

The digest identifying each message should not vary significantly (sic) for changes that can be produced automatically.

What is needed is a localized hashing function such as those applied in information retrieval systems. However, the techniques designed for information retrieval have to be carefully adapted since they were not designed to tolerate malicious behavior which must

instead be considered in our environment. For instance , MIME encoding is often used by spammers to disguise message content while it is in transit, thereby allowing it to sneak past content-based spam filtering.

Less often, spammers will use HTML character entity codes to disguise selected characters in an HTML body.This originates the second requirement mentioned above which is :

The encoding must be robust against intentional attacks.

A solution to the spam problem must assume that spammers are technically competent and after an analysis of the characteristics of the filtering solutions they may spend resources to design tools that are able to automatically produce spam that bypasses the checks put in place.

Finally an encoding suitable for use in the framework of an anti-spam filter should not put the user at risk of accidental deletion of important messages.Therefore, we have a third requirement:

The encoding should support an extremely low risk of false positives.

It is worth noting that the risk of classifying a legitimate email as spam (false positive), is far more costly than the risk of missing one spam (false positive).

Nilsimsa similarity matching was taken in consideration by Jesse Kornblum when developing the fuzzy hashing in 2006,[that used the algorithms of spam by Andrew Tridgell (2002).

A promising anti-spam technique consists in collecting users opinions that given email messages are spam and using this collective judgment to block message propagation to other users .To be effective, this strategy requires away to identify similarity among

email messages, even if the program used by the spammer to generate the messages may try to obfuscate their common origin.

Nilsimsa operates by using a window of 5 characters that slides along the text of the message one character at a time. When a new character enters the window, the algorithm generates the trigrams associated with the window and passes each of them to a hash function h().The hash function h() computes a value i = h(trigram) between 0 and 255 that corresponds to the ith counter in an array of integers of size 255, called accumulator ,and whose value is increased by 1. After the text analysis , the accumulator will present in the i-th cell the number of trigrams that have been found in the text producing i by the application of the hash function. The relative frequency of each bucket is compared with the average bucket frequency observed for a large collection of messages and the value representing this ratio is associated with the bucket.

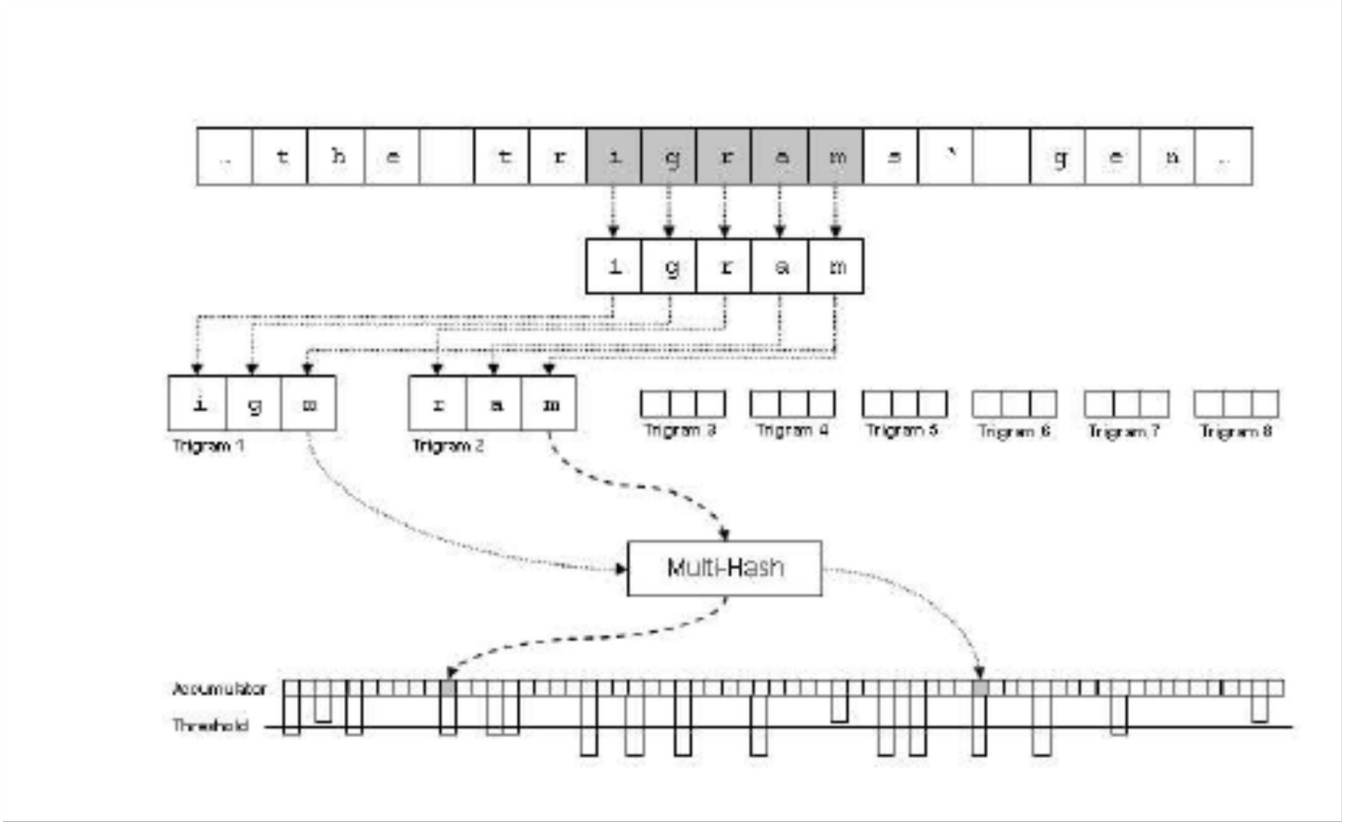

Figure 9. Nilsimsa Algorithm

Then, the ratio of each bucket is considered and if the i-th ratio is greater than the median, the i-th bit of the nilsimsa code is set to 1; it is set to 0 otherwise.In this way a 32-byte code is produced.

To determine if two messages present the same textual content, their Nilsimsa digests are compared, checking the number of bits in the same position that have the same value. The Nilsimsa Compare Value is the number of bits that are equal minus 128. The maximum value of Nilsimsa Compare Value is 128, for two identical digests.

The use of internet has been extensively increasing over the past decade and it continues to be on the ascent. Hence the Internet is gradually becoming an integral part of everyday life. Internet usage is expected to continue growing and email has become a powerful tool intended for idea and information exchange. Negligible time delay during transmission,

security of the data being transferred, low costs are few of the multifarious advantages that email enjoys over other physical methods. However there are few issues that spoil the efficient usage of emails. Spam email is one among them. The term spam is used to describe any “unwanted” thing.

Email spam is a set of unwanted electronic spam mail that contains nearly identical messages sent to huge number of recipients. Spam mail can be not only annoying but also dangerous to recipients. Clicking on links contained in spam emails may send users to phishing and malware .It also may include malware as scripts or other risky executable file attachments.

The problem of spam or Unsolicited Bulk Email (UBE) is becoming a pressing issue. Spam email characterized by three main features:

•Anonymity: The address and identity of the sender are concealed.

•Mass Mailing: The email is sent to large groups of people.

•Unsolicited: The email is not requested by the recipients. While no effective and complete solution to the spam problem is currently available,several moderately successful anti-spam techniques have been proposed, each operating along a different line.

6.2Filters:

List Based Filters: List-based filters attempt to stop spam by categorizing senders as spammers or trusted users, and blocking or allowing their messages accordingly.Senders in blacklist are considered spammers and all mails sent by them are blocked, where senders in whitelist are trustees and all mails sent by them are allowed.

Content-Based Filters: Rather than enforcing across the board policies for all messages from a particular email or IP address, content-based filters evaluate words or phrases found in each individual message to determine whether an email is spam or legitimate.

A word-based spam filter is the simplest type of content-based filter. Generally speaking, word-based filters simply block any email that contains certain terms.

Heuristic (or rule-based) filters like Spam Assassin take things a step beyond simple word-based filters. Rather than blocking messages that contain a suspicious word, heuristic filters take multiple terms found in an email into consideration.

Bayesian filters employ the laws of mathematical probability to determine which messages are legitimate and which are spam. In order for a Bayesian filter to effectively block spam,the end user must initially "train" it by manually flagging each message as either junk or legitimate. Over time, the filter takes words and phrases found in legitimate emails and adds them to a list; it does the same with terms found in spam.

7.Conclusion

Locality-sensitive hashing (**LSH**) reduces the dimensionality of high-dimensional data. LSH hashes input items so that similar items map to the same "buckets" with high probability (the number of buckets being much smaller than the universe of possible input items). LSH differs from conventional and cryptographic hash functions because it aims to maximize the probability of a "collision" for similar items. Locality-sensitive hashing has much in common with data clustering and nearest neighbor search.

Locality-sensitive hashing can be used to calculate similarities between files to various degrees of accuracy.

There has been considerable research and use of similarity digests and Locality Sensitive Hashing (LSH) schemes - those hashing schemes where small changes in a file result in small changes in the digest. These schemes are useful in security and forensic applications. We examine how well three similarity digest schemes (Ssdeep, Sdhash and TLSH) work when exposed to random change. Various file types are tested by randomly manipulating source code, Html, text and executable files. In addition, we test for similarities in modified image files that were generated by cybercriminals to defeat fuzzy hashing schemes (spam images). The experiments expose shortcomings in the Sdhash and Ssdeep schemes that can be exploited in straight forward ways. The results suggest that the TLSH scheme is more robust to the attacks and random changes considered.

Similarity digest schemes exhibit the property that small changes to the file being hashed results in a small change to the hash. The similarity between two files can be determined by comparing the digests of the original files.

8.Tools and Technologies

Eclipse IDE for Java:

Version: Eclipse Luna(4.4)

Eclipse is an integrated development environment (IDE). It contains a base workspace and an extensible plug-in system for customizing the environment. Written mostly in Java, Eclipse can be used to develop applications. By means of various plug-ins, Eclipse may also be used to develop applications in other programming languages: Ada, Java etc. can also be used to develop packages for the software Mathematica. Development environments include the Eclipse Java development tools (JDT) for Java and Scala, Eclipse CDT for C/C++ and Eclipse PDT for PHP, among others.

The initial codebase originated from IBM VisualAge.The Eclipse software development kit (SDK), which includes the Java development tools, is meant for Java developers. Users can extend its abilities by installing plug-ins written for the Eclipse Platform, such as development toolkits for other programming languages, and can write and contribute their own plug-in modules.

Released under the terms of the Eclipse Public License, Eclipse SDK is free and open source software (although it is incompatible with the GNU General Public License). It was one of the first IDEs to run under GNU Classpath and it runs without problems under IcedTea.

9.References

1.Moses S. Charikar, “Similarity Estimation Techniques”

2.Locality-sensitive hashing wikipedia

3.Kernelized Locality-Sensitive Hashing -Brian Kulis and Kristen Grauman(2009)

4.Locality sensitive hashing: a comparison of hash function types and querying mechanisms- Loic Pauleve, Herve Jegou, Laurent Amsaleg (2010)

5.Damiani et. al (2004). "An Open Digest-based Technique for Spam Detection"

6.Qin Liv (2007).”Multi-Probe LSH”

7.Nilsimsa Hash vs. SimHash

https://outofthemine.wordpress.com/2015/01/10/nilsimsa-vs-simhash/

8.Nilsimsa hash http://en.wikipedia.org/wiki/Nilsimsa_Hash

9.G. Forman, K. Eshghi, and S. Chiocchetti. Finding similar in large document repositories.Conference on Knowledge Discovery in Data, pages 394{400, 2005.[2] T.C. Hoad and J. Zobel. Methods for identifying versioned and plagiarized documents. Journal of the American Society for Information Science and Technology, 54(3):203{215, 2003.

10.U. Manber. Finding similar system. Proceedings of the USENIX Winter 1994 Technical Conference on USENIX Winter 1994 Technical Conference table of contents, pages 2{ 2, 1994.

11.A. Broder. On the resemblance and containment of documents. Proceedings of the Compression and Complexity of Sequences, page 21, 1997.

12. T. Yamamoto, M. Matsusita, T. Kamiya, and K. Inoue. Measuring Similarity of Large Software Systems Based on Source Code Correspondence. Proceedings of the 6th

International Conference on Product Focused Software Process Improvement (PROFES05), 2005.
13. M. Welsh, N. Borisov, J. Hill, R. von Behren, and A. Woo. Querying large collections of music for similarity, 1999.
14. D. Buttler. A Short Survey of Document Structure Similarity Algorithms. International Conference on Internet Computing, 2004.
15. B. S. Baker. “A Program for Identifying Duplicated Code”. Proc.CSS Interface,1999, Vol. 24, pp. 49-57.
16. M. Ester, H.-P. Kriegel, J. Sander, and X. Xu, “A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise”, Proc. KDD, 1996, pp. 226-231.
17. C. Gong, Y. Huang, X. Cheng and S. Bai. “Detecting near-duplicates in large-scale short text databases”, Proc.PAKDD, 2008, pp. 877-883.
18. M. Henzinger, “Finding near-duplicate web pages: a large-scale evaluation of algorithms”, Proc. SIGIR, 2006, pp. 284-291

10.Appendices

Appendix A
Minhash:

Class: Minhash1.java:

Imported Files:

```
import java.io.*;
import java.util.HashMap;
import java.util.HashSet;
import java.util.Map;
import java.util.Random;
import java.util.Set;
```

Code:

```
public class minhash1 {
   Set<String> set1 = new HashSet<>();
   Set<String> set2 = new HashSet<>();
   Map<String, boolean[]> bitArray = new HashMap<String, boolean[]>();
   int setnum=2,hashNum;
   int minHashValues[][];
   int hash[];
    public void initialize()
         for (int i = 0; i < setnum; i++)
           for (int j = 0; j < hashNum; j++)

                 minHashValues[i][j] = Integer.MAX_VALUE;

       System.out.println("minHashValuesArray");
       for(int i = 0; i < setnum; i++)
           for (int j = 0; j < hashNum; j++)

          System.out.print( minHashValues[i][j]+" ");
       System.out.println();
       hash = new int[hashNum];

        Random r = new Random(11);

        for (int i = 0; i <hashNum; i++)
           int a = (int) r.nextInt();

           int b = (int) r.nextInt();

           int c = (int) r.nextInt();
```

```
            int x = hash(a * b * c, a, b, c);

            hash[i] = x;


}
   }
public static void main(String args[])
{


}
public void buildSetsFromFiles()
{
   // Build sets from the two files to be compared for similarity

 try{
   File myFile = new File("C:\\Users\\x-Srishti-
x\\Documents\\NetBeansProjects\\JavaApplication1\\src\\javaapplication1\\TextFile1.txt"
);
   FileReader fileReader = new FileReader(myFile);
   BufferedReader br = new BufferedReader(fileReader);
   String line;
   while((line = br.readLine())!=null)
   {
      String[] result = line.split(" ");
      for(String token: result)
      {
         set1.add(token);

      }

   }
   br.close();
     System.out.println(set1);




   File myFile1 = new File("C:\\Users\\x-Srishti-
x\\Documents\\NetBeansProjects\\JavaApplication1\\src\\javaapplication1\\TextFile2.txt"
);
```

```
    FileReader fileReader1 = new FileReader(myFile1);
    BufferedReader br1 = new BufferedReader(fileReader1);

    while((line = br1.readLine())!=null)
    {
       String[] result = line.split(" ");
       for(String token: result)
       {
          set2.add(token);

       }

    }
    br.close();
      System.out.println(set2);
      hashNum=set1.size()+set2.size();
      System.out.println(hashNum);
      minHashValues = new int[setnum][hashNum];
      initialize();

  buildBitmapsFromSets();
}
catch(Exception ex)
{
}

}
  private static int hash(int x, int a, int b, int c)

    {

       int hashValue = (int) ((a * (x >> 4) + b * x + c)&131071)+5;


       return Math.abs(hashValue);

    }
public void buildBitmapsFromSets()
{



       for (String t : set1)
```

```
        {
            bitArray.put(t, new boolean[] { true, false });
        }
        for (String t : set2)
        {
            if (bitArray.containsKey(t))
            {

                bitArray.put(t, new boolean[] { true, true });
            }
            else if (!bitArray.containsKey(t))
            {

                bitArray.put(t, new boolean[] { false, true });
            }

        }
        System.out.println(bitArray);

}
public void sim()
{

    //calculating minhash signatures for the sets
        minhashSig(set1,0);
        minhashSig(set2,1);

    int similarMinHash = 0;
```

```
    double similarity;

        for (int i = 0; i < hashNum; i++)

        {

            if (minHashValues[0][i] == minHashValues[1][i])

            {
                System.out.println(minHashValues[0][i]);
                similarMinHash++;

            }

        }
        System.out.println("similarMinhashes"+similarMinHash+"hashNum"+hashNum);

        similarity =(double) similarMinHash/hashNum;


        System.out.println("Similarity between the two documents is"+similarity
+"or"+similarity*100+"%" );

}


public void minhashSig(Set<String> set, int in)
{
     int index = 0;
      System.out.println("MINSIG");
       for (String element : bitArray.keySet())

       {

          // for every element

          for (int i = 0; i < hashNum; i++)

          {

             // for every hash value
```

```
            if (set.contains(element))
            {
               int hindex = hash[index];
               if (hindex < minHashValues[in][index])
               {
                 System.out.println(hindex);
                  minHashValues[in][index] = hindex;//if the hash is smaller, replace it
               }
            }
         }
         index++;
      }
}

}
```

Minhash2.java:

```
package javaapplication1;

import java.util.HashSet;
import java.util.Set;
public class minhash2 {
   public static void main(String args[])
   {
   minhash1 min = new minhash1();
   min.buildSetsFromFiles();
   min.sim();

   }
}
```

Appendix B
Nilsimsa Hash:

Nilsimsa.java:

```
import java.io.*;
import java.util.Random;


public class Nilsimsa{
        int g,g1,g2;//hash functions
         Random r = new Random();

         int r1=r.nextInt(2)+1;
         String[] trigrams=
{"the","and","ing","her","hat","his","tha","ere","for","ent","ion","ter","was","you","ith","
ver","all","wit","thi","tio","eth","dth","men","sth","oft","tis","edt","has","nde","ent"};
    //list of the most common trigrams
        int[] avgfreq= new int[256];
        int[] accum= new int[256];
        int[] ratios= new int[256];
        int[] nilsimsaCode=new int[256];

        int[] Code1=new int[256];
        int[] Code2=new int[256];
        int flag=0,check;

        //float avgfreq=2;
        float median=0;


        public Nilsimsa()
        {
                for(int i=0;i<=255;i++)
                {
                        accum[i]=0;
                }
        }
         public static void main(String[] args)throws IOException
           {

           }
         public void setFlag()
```

```
{
        flag=1;
}
 public int h1(String x) {
          char ch[];
          ch = x.toCharArray();
          int i, sum;
          for (sum=0, i=0; i < x.length(); i++)
            sum += ch[i];
          i= (255)*(sum-33000)/(123000);
          return  i;
         }
public int h2(String x)
{
         int a=x.hashCode();
        int b= (255)*(a-33000)/(123000);

        return b;
}
public void hashgen(String x)
{
        //System.out.println("r1:"+r1);

         if(r1==1)

                 g=h1(x)+2*h2(x);

         if(r1==2)

                 g=h1(x)+h2(x)*3;

         if(r1==3)

                 g=h1(x)+h2(x)%5;

          if(g<0)
                  g=-g;
          if(g>255)
                  g=(255);


          acc(g);
}
public void slidewindow()
```

```
 {


}
 public void makeTrigrams(String s)
 {
   char a,b,c;
   int i=0;



   while(i<3)
   {

                    a=s.charAt(i);
                    b=s.charAt(i+1);
                    c=s.charAt(i+2);
                   //System.out.println(a+""+b+""+c);
                   /* if(a>=65&&a<=90){
                            int x=a+32;
                            a=(char)x;
                    }
                    if(b>=65 && b<=90){
                            int x=b+32;
                            b=(char)x;
                    }
                    if(c>=65&& c<=90){
                            int x=c+32;
                            c=(char)x;
                    }

                    if((int)a>(int )b)
        {
                    temp=a;
                    a=b;
                    b=temp;
        }
        if((int)a>(int)c)
        {
                    temp=a;
                    a=c;
                    c=temp;
        }
        if((int)b>(int)c)
```

```
        {
                temp=b;
                b=c;
                c=temp;
        }
        //System.out.println(a+" "+b+" "+c);
        */
                String x=""+a+b+c;
                //System.out.println(x);
        check=checkTrigram(x);
        if(check==1)
        hashgen(x);
         i++;
   }
   //System.out.println("Accumulator");
 for(int f=0;f<256;f++)
 {
      //System.out.print(accum[f]+" ");
 }
 //System.out.println();

}
 public int checkTrigram(String x)
 {
       check=1;
       for(int i=0;i<30;i++)
       {
               if(x.equals(trigrams[i])==true)
               {
                       check=0;
               }

       }
       return check;
 }
/* public void multihash(String x)
 {
       //using three hash functions
       int a=x.hashCode();
       int b= (255)*(a-33000)/(123000);

       //System.out.println(b);
       acc(b);
 }*/
```

```
public void acc(int a)
{
accum[a]++;
}
public void calcRatio()
{
	System.out.println();
	System.out.println("Accumulator");

	for(int i=0;i<256;i++)
	{
		System.out.print(accum[i]+" ");

	}
	System.out.println();
	//System.out.println("ratios");
	for(int i=0;i<256;i++)
	{
		if(avgfreq[i]!=0)
		{
	ratios[i]=(accum[i]/avgfreq[i]);

		}
		else
			ratios[i]=0;
		median+=ratios[i];
		//System.out.print(ratios[i]);
}
	System.out.println();
	median/=256;
	System.out.println();
	System.out.println("median:"+median);
}
public void genByteCode()
{
	System.out.println();
	System.out.println("Nilsimsa Code");
	for(int i=0;i<256;i++)
	{
	 if(ratios[i]>median)
		nilsimsaCode[i]=1;
	 else
		nilsimsaCode[i]=0;
	 if(flag==0)
```

```
                    Code1[i]=nilsimsaCode[i];
            else
                    Code2[i]=nilsimsaCode[i];
           System.out.print(nilsimsaCode[i]);
        }

          System.out.println();

    }
    public void compValue()
    {
          int ctr=0,cmpvalue;
          for(int i=0;i<256;i++)
          {
                  if(Code1[i]==Code2[i])
                          ctr++;
          }
          cmpvalue=ctr-128;
          System.out.println("Nilsimsa Compare Value:"+cmpvalue);
    }
}
```

NilsimsaObject.java:

```
import java.io.FileReader;

import java.io.IOException;

import java.util.Random;

public class NilsimsaObject {

        public static void main(String[] args) throws IOException  {
                //AvgBucketFrequency abf= new AvgBucketFrequency();

                Nilsimsa nim=new Nilsimsa(); //creating an object of nilsimsa class
                 String win="";
      int ctr=0;
      Random r = new Random();
      String[] file1=
{"E://EclipseLunaWorkspace//final//bin//Spam_Catalog//Spam1.txt","E://EclipseLunaW
orkspace//final//bin//Spam_Catalog//Spam2.txt","E://EclipseLunaWorkspace//final//bin//
Spam_Catalog//Spam3.txt","E://EclipseLunaWorkspace//final//bin//Spam_Catalog//Spa
m4.txt","E://EclipseLunaWorkspace//final//bin//Spam_Catalog//Spam5.txt","E://EclipseL
unaWorkspace//final//bin//Spam_Catalog//Spam6.txt","E://EclipseLunaWorkspace//final/
/bin//Spam_Catalog//Spam7.txt","E://EclipseLunaWorkspace//final//bin//Spam_Catalog//
Spam8.txt","E://EclipseLunaWorkspace//final//bin//Spam_Catalog//Spam9.txt","E://Ecli
pseLunaWorkspace//final//bin//Spam_Catalog//Spam10.txt","E://EclipseLunaWorkspace/
/final//bin//Spam_Catalog//Spam11.txt","E://EclipseLunaWorkspace//final//bin//Spam_C
atalog//Spam12.txt","E://EclipseLunaWorkspace//final//bin//Spam_Catalog//Spam13.txt"
,"E://EclipseLunaWorkspace//final//bin//Spam_Catalog//Spam14.txt","E://EclipseLunaW
orkspace//final//bin//Spam_Catalog//Spam15.txt","E://EclipseLunaWorkspace//final//bin/
/Spam_Catalog//Spam16.txt","E://EclipseLunaWorkspace//final//bin//Spam_Catalog//Sp
am17.txt","E://EclipseLunaWorkspace//final//bin//Spam_Catalog//Spam18.txt","E://Eclip
seLunaWorkspace//final//bin//Spam_Catalog//Spam19.txt","E://EclipseLunaWorkspace//
final//bin//Spam_Catalog//Spam20.txt"};
          //file1 contains the addresses of all the files in the spam catalog
          int r2=r.nextInt(20);

      FileReader inputStream = null;

      try {

         inputStream = new
FileReader("E://EclipseLunaWorkspace//final//bin//Spam.txt"); //Reading the file that
contains 100 spam emails
```

```
    //outputStream = new FileWriter("characteroutput.txt");

    int c;

    while ((c = inputStream.read()) != -1) {
       //outputStream.write(c);
            //System.out.print((char)c);
            ctr++;
            win+=(char)c;
            if(ctr==5)
            {
                    ctr=0;
                    //System.out.println(win);
                    nim.makeTrigrams(win);
                    win="";
            }
    }
}
finally {

    if (inputStream != null)
    {
       inputStream.close();
    }


}
System.out.println("Average Bucket Frequency");
for(int i=0;i<256;i++)
{

    nim.avgfreq[i]=(nim.accum[i]/100);
// System.out.print(nim1.accum[i]+" ");

System.out.print(nim.avgfreq[i]+" ");
nim.accum[i]=0;

}


try {
    inputStream = new FileReader(file1[r2]);
    //outputStream = new FileWriter("characteroutput.txt");
```

```
            int c;
            while ((c = inputStream.read()) != -1) {
               //outputStream.write(c);
                     //System.out.print((char)c);
                     ctr++;
                     win+=(char)c;
                     if(ctr==5)
                     {
                              ctr=0;
                              //System.out.println(win);
                              nim.makeTrigrams(win);
                              win="";
                     }
            }
         }

         finally {

            if (inputStream != null)
            {
               inputStream.close();
            }


         }
         nim.calcRatio();
         nim.genByteCode();
         for(int i=0;i<=255;i++)
         {
             nim.accum[i]=0;
         }
         System.out.println();
         try {

            inputStream = new
FileReader("E://EclipseLunaWorkspace//final//bin//MyFile2.txt");
            //outputStream = new FileWriter("characteroutput.txt");

            int c;
            while ((c = inputStream.read()) != -1)
            {
               //outputStream.write(c);
                     //System.out.print((char)c);
                     ctr++;
```

```
                win+=(char)c;

                if(ctr==5)
                {
                        ctr=0;
                        //System.out.println(win);
                        nim.makeTrigrams(win);
                        win="";
                }
            }
        }

        finally {

            if (inputStream != null)
            {
                inputStream.close();
            }


        }
        nim.calcRatio();
        nim.genByteCode();
        nim.compValue();

            }

}
```