# Locality Sensitive Hashing (LSH) via MinHash

Your Name(s): Sicong Chang, Tianyi Tang

September 5, 2020

## 1 Application

Find all similar string document pairs with similarity difference equal to or smaller than a given threshold from a tremendous string document collection. A real life example is plagiarism detection. This is a significant challenge, as traditional way of comparing every pair in the collection would give exact answers but is totally infeasible. Our goal here is to find similar documents with high probability and reasonable amount of work.

## 2 Algorithm Pseudocode

LSH takes in a list of txt files and a threshold s. It returns the pairs of files whose Jaccard similarity is at least s. Let $C_1$ and $C_2$ be the shingles set of two documents. Jaccard similarity of the two files is defined as $\frac{|c_1 \cap c_2|}{|c_1 \cup c_2|}$.

LSH via minHash has 4 steps.

The first step is shingling. It takes in $n$ text files and a parameter $k$, then output a map docToShingle from each document to a set of shingles and the union of all these sets allShingles. The set of shingles of a document is a set of all consecutive $k$ characters in a text file. Let's call the size of the allShingles $m$.

The second step is to turn the set we created to boolean matrices. It takes in docToShingle, the shingle map we created, and output an $m \times n$ matrix $M$. The rows represent m elements of allShingles. The columns represent each text file. $M[i][j] = 1$ means the ith shingle exists in jth file, $M[i][j] = 0$ otherwise.

The third step is minhash. It takes in the Matrix $M$ we created and a parameter $k$. It first generates $k$ permutations of the rows of $M$: $\pi_1, \pi_2, \ldots, \pi_k$. Then generate $k$ hash functions $h_1, h_2, \ldots, h_k$. $h_i(C)$ is the number of the first row in the order of $\pi_i$ in which column $C$ has a value 1. Then we apply the hash functions to all columns and output a $k \times n$ signature $S$. $S[i][j] = h_i(M[:, j])$. Then we output $S$.

The last step is LSH. It takes in the signature $S$ we created and band number $b$. For

each band, hash its portion of each column to a hash table with $k$ buckets($k$ is large). Output the pairs of columns that hash to the same bucket for $\geq$ one band. Those pairs are the candidate pairs we need to check similarity.

**Algorithm 1** LSH

**function** SHINGLE(files, $k$) ▷ Turn files to shingle sets
  Initialize a map docToShingle and a set allShingles.
  **for** each file in files **do**
    Initialize a set S
    loop over the file and add all k-shingles to S
    add (file, S) to docToShingle
    allShingles = allShingles $\cup$ S
  **return** docToShingle, allShingles

**function** MINHASH($M, k$) ▷ Hash boolean matrix of shingling to signature
  Initialize $k$ permutations of rows of $M$ $\pi_1$, $\pi_2$, ..., $\pi_k$
  Initialize $k \times$ n signature matrix S
  **for** each $\pi_i$ **do**
    **for** each column $C_j$ **do**
      S[i][j] = the first index p of $\pi_i$ such that $C_j[p] = 1$
  **return** S

**function** LSH($S, b$) ▷ Hash the signature matrix
  Initialize b list of buckets of size k(set k as large as possible)
  **for** $i = 0, 1, 2, ..., b1$ **do**
    currentBand = $M[ir : (i+1)r, :]$
    Hash each colomn of currentBand to the ith list of buckets
  **return** all pairs in one of the same buckets

The expected runtime is $O(n)$ since we need to minhash each document to a signature matrix and hash each column of signature matrix to a bucket. All these operations are $O(n)$.
The expected space complexity is $O(n)$. We need to store a boolean matrix, a signature matrix, and several list of buckets. We don't actually store a boolean matrix. Since it is sparse, we use a map from document to all indices with 1 instead. Thus it's in $O(n)$. Signature matrix is in $O(n)$ since its column number is n and row number is a constant. The buckets are also in $O(n)$ because although there are many buckets available, at most n of them is non empty and those are the only ones we store.
Thus the space complexity is $O(n)$.
The algorithm is not guaranteed to produce the correct answer since when we are doing LSH, we only know the probability that 2 document are considered not similar by algorithm(they don't hash to the same bucket for all bands) are small. However, there is no guarantee that they will hash to the same bucket at least once. Thus the algorithm is not guaranteed to produce the correct answer.

# 3 Explanation and Intuition

The algorithm works because two theorems:

$P(h(d1) = h(d2)) = sim(d1, d2)$
This means that the similarity of two documents is equal to the expectation of fraction of the hash functions in which they agree in the signature matrix. The probability of them agreeing on one hash function is equal to the similarity between them.

Recall that b is the number of band, k is the number of hash functions we applied on boolean matrix, and s is the threshold. The probability of at least one band of signature column hashed to same bucket is $1 - (1 - s^{\frac{k}{b}})^b$.
Thus we can tune b to make this probability large so that we include all the pairs we want in our candidate list at a large probability.

It is a Monte Carlo algorithm since we can only output all correct results at a very high probability. However, we are not guaranteed to find all of them.

# 4 Advantage(s) over Deterministic Counterpart(s)

Let $M$ be the map docidToSignature we generated, $t$ be the threshold.

**Algorithm 2** Deterministic

```
function NEARPAIRS(M, t)
    Initialize a set S
    for every file pair d1, d2 do
        if Jaccard similarity(d1,d2) ≥ t then
            add (d1,d2) to S
```

A deterministic way to find all similar pairs is to compute the similarity of every possible pair of column of the shingling boolean matrix, and then filter out the pairs whose similarity is larger or equal to threshold. However, since there are $\frac{n(n-1)}{2}$ pairs, the time complexity is $O(n^2)$, which is worse than $O(n)$. Considering space complexity:

For deterministic solution, we need to keep referencing shingling sets of documents. The size of these sets is not deterministic. However, for LSH, we only need to refer to signature matrix. Thus the space conplexity is constant of $O(kn)$. Besides, since we are storing ints in signature matrix and strings in shingling sets, we are using less space.

# 5 Implementation in Python

```
from sys import getsizeof
import sys
import os
import string
import numpy as np
from sympy import symbols, solve
import time
import matplotlib.pyplot as plt

# path: path of a txt file
# k: the length of a shingle
# num_of_docs: the max documents the function process
# create a k-shingles set of a txt file
def shingle_one_doc(path, k):
    shingle_for_one_doc = set()
    with open(path, encoding="latin-1") as file:
        # read a file
        text = file.read()
        text = str(text)
        trimed = text.replace(' ', '').replace('\n', '')
        for i in range(len(trimed) - k + 1):
            # loop through shingles in a file
            shingle_for_one_doc.add(trimed[i: i+k])
    return shingle_for_one_doc

# num_of_docs: the max documents the function process
# path: the path to a directory of files
# k: the length of a shingle
# return three things in a tuple:
# 1.a map from file id (start from 0) to its shringling set.
# 2.a numpy array with the union of all the shingles sets of files.
# 3.a map from file id to filename.
def shingle(num_of_docs, path, k=8):
    docid = 0
    count = num_of_docs
    print(count)
    docid_to_shingling_set = dict()
    docid_to_docname = dict()
    all_tokens = set()
    for filename in os.listdir(path):
        if filename.endswith(".txt") and count > 0:
            file_path = os.path.join(path, filename)
            words = shingle_one_doc(file_path, k)
```

```
            docid_to_shingling_set[docid] = words
            all_tokens = all_tokens | words
            docid_to_docname[docid] = file_path
            docid += 1
            count = count - 1
    return docid_to_shingling_set, np.array(list(all_tokens)), docid_to_docname

# docid_to_shingling_set: a map from file id (start from 0) to its shringling set.
# all_tokens: a numpy array with the union of all the shingles sets of files.
# return a map from docid to a set of all the indexes of shingles it contains
def create_doc_signature(docid_to_shingling_set, all_tokens):
    docid_to_signature = dict()
    for docid in docid_to_shingling_set:
        #loop through each doc
        words = docid_to_shingling_set[docid]
        wordsSignature = set()
        for i in range(len(all_tokens)):
            # loop through all the tokens
            if all_tokens[i] in words:
                wordsSignature.add(i)
        docid_to_signature[docid] = wordsSignature
    return docid_to_signature

# docid_to_signature: a map from user_id to a set of all the
# indexes of shingles it contains
# a permutation of all the shingles
# return an numpy array which stores signature
# for all files for a specific permutation
def min_hash(docid_to_signature, permutation):
    res = np.zeros(len(docid_to_signature))
    for i in range(len(docid_to_signature)):
        # for each doc
        doc_sig = docid_to_signature[i]
        for j in range(len(permutation)):
            # we store j
            if permutation[j] in doc_sig:
                res[i] = j
                break
    return res

# curr: current band of signature
# k: number of buckets
# dict: the bucket we want to modify
# hash each files in the band to k buckets
def create_buckets_for_one_band(curr,k, dict):
```

```
    # for each column hash
    # print(curr.shape[1])
    for i in range(curr.shape[1]):
        column = curr[:,i]
        # hash value is just the bucket number
        hash_value = hash(column.tostring()) % k
        if hash_value not in dict:
            dict[hash_value] = set()
        dict[hash_value].add(i)

# m: number of rows of the matrix
# n: number of columns of the matrix
# num_hashes: number of hash functions we use when doing min hash
# docid_to_signature: a map from user_id to a set of
# all the indexes of shingles it contains
# return a signature matrix of documents
def generate_signature_matrix(m, n, num_hashes, docid_to_signature):
    sig_matrix = np.full((num_hashes, len(docid_to_signature)), np.inf)
    # generate hash function
    rand_matrix = np.random.randint(0,2^32,(num_hashes,2))

    for i in range(m):
        # for each column
        for j in range(n):
            if i in docid_to_signature[j]:
                for k in range(num_hashes):
                    # apply min hash algorithm
                    sig_matrix[k][j] = min(sig_matrix[k][j],
                    (rand_matrix[k][0] * i + rand_matrix[k][1]) % m)
    return sig_matrix

# sig_matrix: signature matrix of the documents
# num_hashes: number of hash functions we use when doing min hash
# b: number of bands
# k: number of buckets
def lsh(sig_matrix, num_hashes, b, k=sys.maxsize):
    # LSH process
    r = int(num_hashes / b)
    # k = sys.maxsize
    # num of buckets
    dicts = [dict() for i in range(b)]
    for i in range(b):
        # for each band
        curr = sig_matrix[i*r:(i+1)*r,:]
        # a bucket is a dict from bucketid to set of docids
```

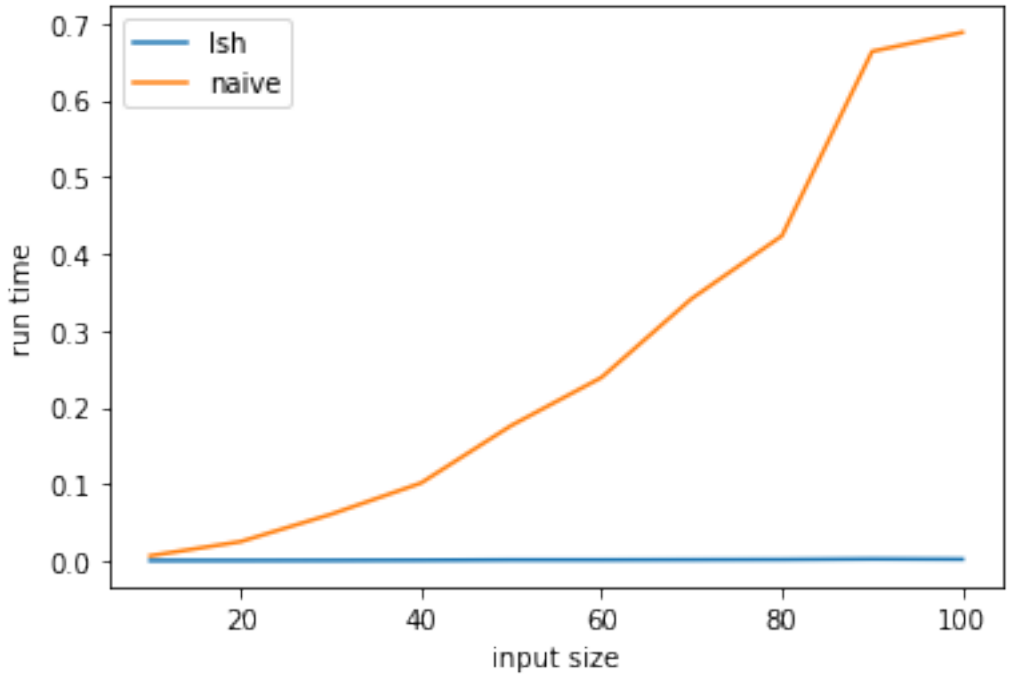
```
            create_buckets_for_one_band(curr, k, dicts[i])
        return show_result(dicts)

    # return all the similar pairs in dicts
    def show_result(dicts):
        similar_pairs = set()
        for word_dict in dicts:
            for key in word_dict:
                if len(word_dict[key]) > 1:
                    similars = word_dict[key]
                    for doc1 in similars:
                        for doc2 in similars:
                            if doc1 < doc2:
                                similar_pairs.add((doc1, doc2))
        return similar_pairs

    # naively return all the similar pairs
    def show_actual_result(docid_to_signature, threshold):
        similar_pairs = set()
        for doc1 in docid_to_signature:
            for doc2 in docid_to_signature:
                overlap = docid_to_signature[doc1].intersection(docid_to_signature[doc2])
                union = docid_to_signature[doc1] | docid_to_signature[doc2]
                if (len(overlap) / len(union) >= threshold and doc1 < doc2):
                    similar_pairs.add((doc1, doc2))
        return similar_pairs

    # compute similarity data of given pairs of documents
    def compute_threshold(docid_to_signature, similar_pairs, threshold):
        similarity_list = list()
        actual_similar = set()
        for doc1, doc2 in similar_pairs:
            overlap = docid_to_signature[doc1].intersection(docid_to_signature[doc2])
            union = docid_to_signature[doc1] | docid_to_signature[doc2]
            # print(overlap)
            # print(union)
            # print(matching_size)
            # print(all_tokens)
            similarity = len(overlap) / len(union)
            similarity_list.append(similarity)
            if similarity >= threshold:
                actual_similar.add((doc1,doc2))
        return np.mean(similarity_list), actual_similar

    # take in a set of document id pairs and return the corresponding document name pairs
```

```
def id_to_name(similar_pairs, docid_to_docname):
    name_set = set()
    for doc1, doc2 in similar_pairs:
        name_set.add((docid_to_docname[doc1], docid_to_docname[doc2]))
    return name_set
```

# 6 Visual Results and Analysis

## 6.1 runtime vs input

Plot:



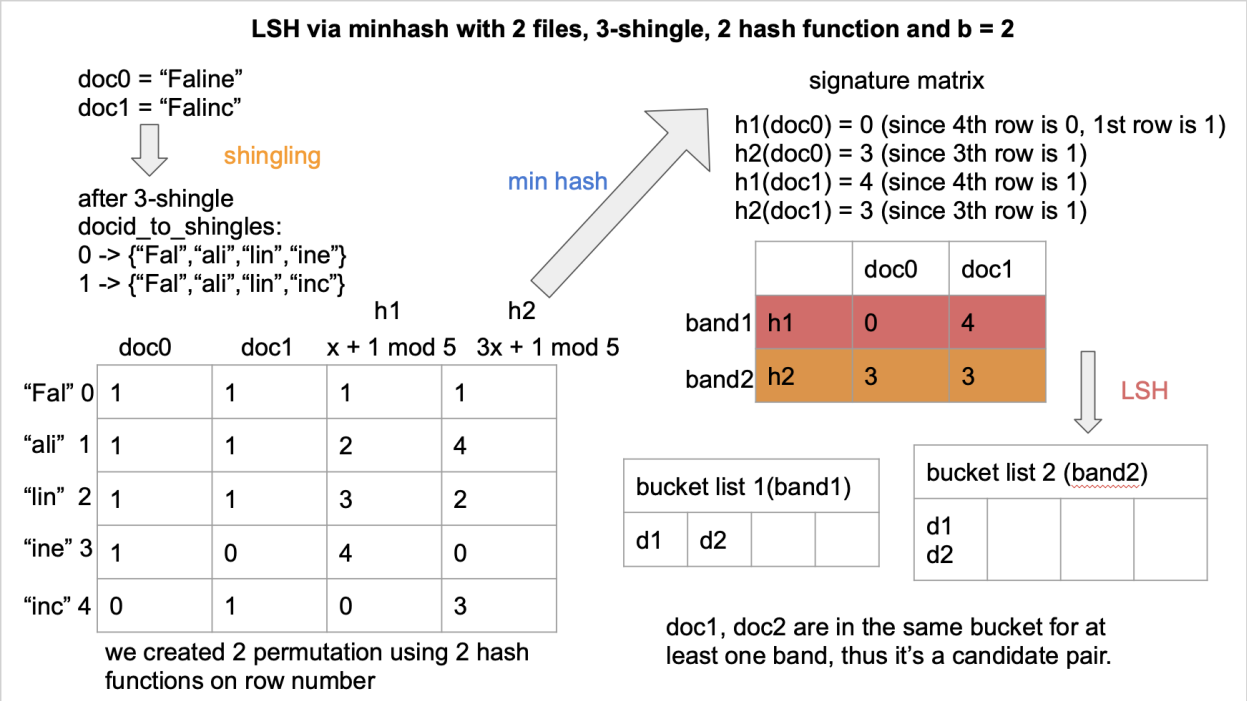

It's clear that deterministic algorithm's run time grows in $O(n^2)$, which is significantly larger than LSH's run time every time we find near pairs.

Code:

```
num_hashes = 100
path = "corpus-20090418"
num_of_docs_list = np.arange(10,101,10)
lsh_time_list = list()
naive_time_list = list()
for num_of_docs in num_of_docs_list:
    print(num_of_docs)
    docid_to_shingling_set, all_tokens, docid_to_docname = shingle(num_of_docs, path, k=
    docid_to_signature = create_doc_signature(docid_to_shingling_set, all_tokens)
```

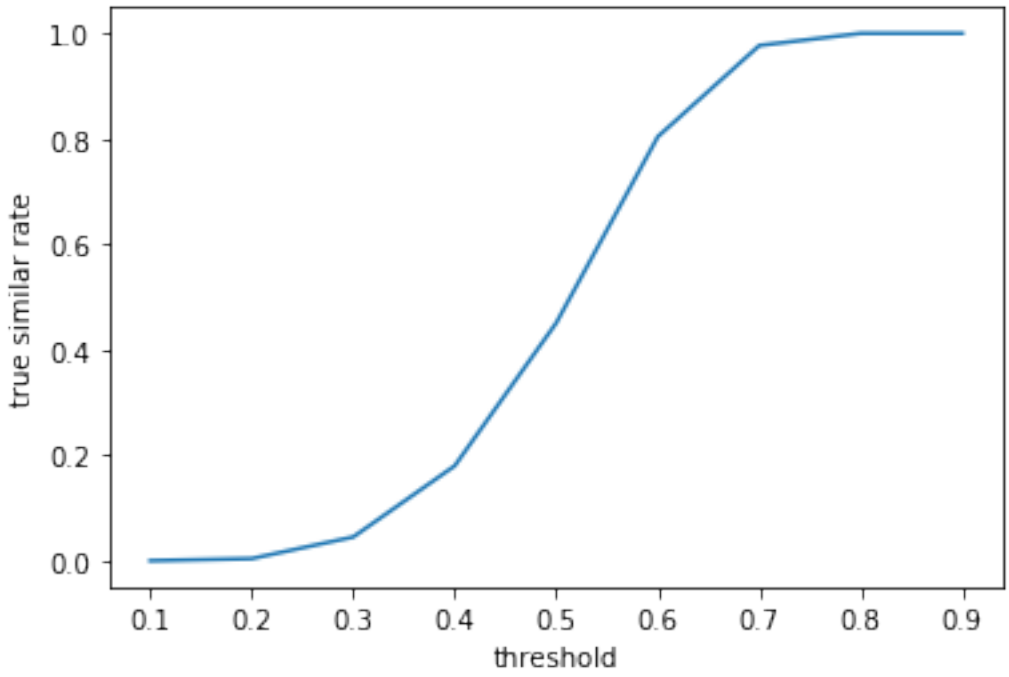
```
    threshold = 0.6
    x = symbols('x')
    expr = (1/x)**(x/num_hashes)-threshold
    b = int(solve(expr)[0])

    start_time = time.time()
    sig_matrix = generate_signature_matrix(len(all_tokens), len(docid_to_signature), num

    similar_pairs = lsh(sig_matrix, b=20, num_hashes=num_hashes)
    lsh_finish = time.time()
    actual_similar_pairs = show_actual_result(docid_to_signature, threshold)
    naive_finish = time.time()

    lsh_time_list.append(lsh_finish - start_time)
    naive_time_list.append(naive_finish - lsh_finish)
plt.plot(num_of_docs_list, lsh_time_list, label="lsh")
plt.plot(num_of_docs_list, naive_time_list, label = "naive")
plt.xlabel("input size")
plt.ylabel("run time")
plt.legend()
```

## 6.2 Example of algorithm




An example of LSH via min hash for 2 files, 3-shingle, 2 hash functions and b = 2.
The 2 documents are hashed to the same bucket for one band so they are considered similar.

### 6.3 True similar rate




The figure above shows the true similar rate vs threshold when we use 100 hash functions and set b = 20. True similar rate is:
$\frac{\text{number close pairs we get}}{\text{number close pairs in the file set}}$
This is consistent with results in the resources we refer to.
Code:

```
def min_hash(docid_to_signature, permutation):
    res = np.zeros(len(docid_to_signature))
    for i in range(len(docid_to_signature)):
        # for each doc
        doc_sig = docid_to_signature[i]
        for j in range(len(permutation)):
            # we store j
            if permutation[j] in doc_sig:
                res[i] = j
                break
    return res

num_hashes = 100
num_of_docs = 2
thresholdlist = np.arange(0.1,1.0,0.1)
false_postive = list()
for i in range(len(thresholdlist)):
```

```
    threshold = thresholdlist[i]-0.1
    path = "files/" + str((i+1)/10)
    docid_to_shingling_set, all_tokens, docid_to_docname =
    shingle(num_of_docs, path, k=3)
    docid_to_signature = create_doc_signature(docid_to_shingling_set, all_tokens)

    lsh_res = 0
    for i in range (1000):
        sig_matrix = np.zeros((num_hashes, len(docid_to_signature)))
        for i in range(num_hashes):
            # create permutation
            permutation = np.arange(len(all_tokens))
            np.random.shuffle(permutation)
            sig_matrix[i] = min_hash(docid_to_signature, permutation)
        similar_pairs = lsh(sig_matrix, b=20, num_hashes=num_hashes)
        mean, lsh_similar = compute_threshold(docid_to_signature,
        similar_pairs, threshold)
        lsh_res += len(lsh_similar)
    false_postive.append(lsh_res / 1000)
plt.plot(thresholdlist, false_postive)
plt.xlabel("threshold")
plt.ylabel("false postive")
```

# 7 Probabilistic Analysis

Define minhash function as $h$ and Jaccard similarity as $sim$, we'll prove the following claim: $sim(d1, d2) = P(h(d1) = h(d2))$, where $d1, d2$ are 2 documents.

Define a, b, c, d as 4 kinds of shingles, where among $d1, d2$ they form the following table (1 means the document contains the shingle, and 0 otherwise):

| | d1 | d2 |
|---|---|---|
| a | 1 | 1 |
| b | 1 | 0 |
| c | 0 | 1 |
| d | 0 | 0 |

Define A, B, C, D as the number of shingles of type-a, b, c, d, respectively.
By definition, the Jaccard similarity $sim(d1, d2) = \frac{|d1 \cap d2|}{|d1 \cup d2|}$, which also equals to $\frac{A}{A+B+C}$.
By definition, $h(d)$ = the serial number of the first (in permuted order) shingle that exits in document $d$.
Thus, the total number of shingles that $h(d1)$ and $h(d2)$ may refer to is $A + B + C$.
Since $h(d1) = h(d2)$ means both sides refer to the same shingle that exists in both $d1$ and $d2$, the single is a type-a shingle.

Thus, $P(h(d1) = h(d2)) = \frac{A}{A+B+C} = sim(d1, d2)$

Recall that b is the number of band, k is the number of hash functions we applied on boolean matrix, and s is the threshold. We want to prove that the probability of at least one band of signature column hashed to same bucket is $1 - (1 - s^{\frac{k}{b}})^b$.
The first property we proved tells us that the probability of signature of 2 column matches on one row is s.
We pick any band. Each band has $\frac{k}{b}$ rows. Thus the probability of all signature matches in the band is $s^{\frac{k}{b}}$. Thus we know that the probability of the two columns in the band hash to different bucket is $1 - s^{\frac{k}{b}}$. Then the probability of all bands hash to different buckets is $(1 - s^{\frac{k}{b}})^b$. Then the probability of at two column hash to same bucket for at least 1 band is $1 - (1 - s^{\frac{k}{b}})^b$. With this property, we can evaluate our result and choose proper $b$ and $k$ to ensure this probability is large.

# References


1. Algorithm
2. Pseudocode 1
3. Pseudocode 2
4. Code of a conference paper search engine