Real-Time Tf-Idf Clustering Using Simhash, Approximate Nearest Neighbors, and DBSCAN

Acknowledgments

I would like to express my sincere gratitude to my thesis advisor Professor Barbara Grosz, for continuously giving me feedback and advice during my research, and for being a wonderful mentor during my last two years at Harvard.

I am also deeply indebted to Professor Stratos Idreos, for his invaluable guidance throughout this semester. I could not have carried out this research without his help.

I would also like to thank Professor H.T. Kung, for graciously agreeing to be my thesis reader, and Professor Jelani Nelson for generously agreeing to meet with me to answer some of my questions.

Lastly, I would like to thank my friends for their support, my parents for their unconditional love, and my brother for constantly encouraging me and checking up on my progress throughout my research.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

As the amount of text and unstructured data generated by people and businesses continues to proliferate at an unprecedented rate, computer scientists have increasingly been focused on finding scalable approaches to making sense of these data. Some of the most successful approaches involve clustering the data. For example, clustering may allow for the detection and elimination of redundancy in a dataset [1]; it may also provide an efficient summarization of a dataset [2]; and may also introduce structure to the dataset that allows for faster search algorithms [3], etc. Clustering is traditionally regarded as a computationally intensive offline process that takes place once and whose results may be repeatedly used in the future. However, there are applications for which it is desirable for clustering to be implemented online in real time, each time a result is needed. For example, news aggregators and content providers such as Facebook and LinkedIn digest millions of news articles every day and serve up a personalized subset of them to each of their hundreds of millions of users' news feeds.

There are two important characteristics about the news feeds that such news aggregators provide:

1. A user's news feed must be reasonably diverse.
2. A news feed is highly personalized and contains targeted news articles according to the user's revealed and inferred preferences.

The first characteristic implies that a news feed must be clustered into categories. The second characteristic implies that each user will be attributed a unique set of news articles, which means the results of a clustering assignment cannot be reused across users. Therefore, these two requirements mean that companies like Facebook and LinkedIn have to cluster every single user's news feed before they can show it to them. This clustering could be done through a periodic offline pass over all members, or it could be done in real time whenever a user logs in. The latter seems a much better alternative because it would cut down on largely unnecessary computation and save massive amounts of energy and money, but it presupposes the existence of efficient clustering algorithms that allow for potentially tens of thousands of news articles to be clustered in seconds. This thesis is primarily focused on describing precisely such an algorithm.

A very common algorithm for text document clustering is term frequency - inverse document frequency (tf-idf) followed by a cosine similarity computation between every pair of documents [4]. However, this approach presents two important drawbacks that prevent it from being a viable candidate for a real-time clustering algorithm:

1. A cosine similarity computation over two very large tf-idf vectors is very computationally intensive.

2. The computation of a quadratic number of similarities, one between every pair of documents, is very computationally intensive.

There have been many attempts recently at overcoming these two drawbacks. The first has largely been overcome by dimensionality reduction techniques such as the Simhash algorithm, which reduces a large vector into a very compact bit representation at the cost of some accuracy. The second drawback is more difficult to overcome, and there have recently been many attempts at tackling the problem. For example, smart bit-flipping algorithms have greatly reduced the number of similarity computations needed for applications of near-duplicate detection, (i.e. cases in which only extremely similar documents are deemed to be in the same cluster). For more general clustering applications (i.e. cases in which documents in the same cluster may be somewhat different but still related), there have been sub-quadratic randomized algorithms proposed, but these algorithms introduce a significant loss of accuracy and leave many important hyper-parameter decisions to the user's discretion.
This thesis is motivated by these more general clustering applications of moderately sized datasets, i.e. datasets on the order of tens of thousands of text documents, that we might be able to cluster in real time.

This thesis presents one major and one minor contribution:

1. The major contribution is a novel randomized algorithm capable of producing an approximately exact clustering of tens of thousands of tf-idf vectors in real-time. The algorithm consists of three successive procedures.

   (a) First, an efficient Locality Sensitive Hashing technique called Simhash that reduces high-dimensional tf-idf vectors to 64-bit fingerprints.

(b) Second, two alternative and novel approximate nearest neighbor procedures that return a desired fraction of all pairs of fingerprints within a distance h of each other. Both procedures attempt to analyze representative samples of their dataset in order to tune their own hyper-parameters, and both exhibit better speed/accuracy tradeoffs than existing randomized algorithms.

(c) Third, a simplified version of the DBSCAN algorithm that takes in a graph of fingerprints within a distance h of each other, and returns a clustering assignment.

2. The minor contribution consists of a series of optimizations to the process of creating tf-idf vectors, which maximize resource usage by leveraging multi-threading, efficient mutex locking, vectorized SIMD instructions, and cache-conscious algorithms.

Chapter 2

Related Work

2.1 Tf-idf and cosine similarity

Term frequency-inverse document frequency (tf-idf), is a statistical technique that reflects how important a word is to a particular document in a text corpus. Tf-idf is very commonly used to cluster text documents [5]. The technique works as follows: Given a corpus of N text documents, we generate one vector of tf-idf weightings for each document in the corpus. Each vector consists of m entries, where m is the number of words in the dictionary, and each entry corresponds to a particular word in the dictionary.

The value in each entry is called a tf-idf weighting, which is the product of a term frequency and an inverse document frequency. The term frequency is proportional to the frequency of the word in the document, and the inverse document frequency is inversely proportional to the frequency of the word in the corpus. The intuition behind the inverse document frequency is that we want to offset the fact that some common words such as articles or prepositions may occur very often in an article but do not contribute much to the semantics

of it.

For any word t in text document d of corpus D containing N documents,

$$\text{tf}(t, d) = f_{t,d}$$

$$\text{idf}(t, D) = \log \frac{N}{1 + |\{d \in D : t \in d\}|}$$

$$\text{tf-idf}(t, d) = \text{tf}(t, d) * \text{idf}(t, D)$$

Having constructed these N tf-idf vectors, we can quantify the similarity between a pair of vectors a and b by using a cosine similarity measure of the angle θ between them.

$$a \cdot b = \|a\|\|b\| \cos(\theta) \Rightarrow \text{similarity}(a, b) = \cos(\theta) = \frac{a \cdot b}{\|a\|\|b\|}$$

Since all tf-idf weights are positive, the cosine similarity will be a number between 0 and 1, with 1 indicating perfect similarity, and 0 indicating no similarity at all.

2.2 Locality Sensitive Hashing and Simhash

Traditional cryptographic hashing functions such as MD5 were designed in order to guarantee that similar but non-identical inputs hash to very different (essentially random) values [6]. Locality Sensitive Hashing (LSH) is a dimensionality reduction technique used often in Nearest Neighbor Search (NNS) problems that proposes to flip that concept on its head.

Instead, LSH techniques guarantee that with high probability, similar inputs will hash to similar values [7]. One of the main benefits of LSH techniques is that they reduce a problem in a high-dimensional space, to an almost equivalent problem in a much lower dimensional space. The problem in the lower dimensional space is much more tractable. For example, high-dimensional vectors may be projected onto a much smaller Hamming space of integers represented by d-bit integers. The distance between two such integers is much simpler to compute: for example, one possible distance measure is the bitwise hamming distance, which is simply the number of bits that differ between the two binary representations.

$$a = 10111010$$

$$b = 11101001$$

Here, the bitwise Hamming distance between a and b is 4.

There is a particular LSH algorithm called Simhash (short for similarity hashing) that was introduced by Charikar in 2002 [8]:
Simhash provably has the property that it reduces high-dimensional vectors u and v to two d-bit integers whose bitwise hamming distance is linearly proportional to the angle between u and v, and thus inversely proportional to their cosine similarity.

Here is a description of the Simhash algorithm:
Given a vector u with N weighted features,

1. Initialize a vector V of d zeros, ($d << N$)

2. For each feature f_i of u with corresponding weight w_i, hash the feature using a d-bit

uniform hash function to obtain a d-bit hash.

3. For each bit j of this hash:
   If j is 1, add w_i to $V[j]$
   If j is 0, subtract w_i from $V[j]$

4. Once this procedure has been completed on all N features of u, construct Simhash $S(u)$, a d-bit integer whose i^{th} bit is 1 if $V[i] > 0$, and 0 otherwise.

5. return $S(u)$

The algorithm has a runtime of $O(dN)$, which is linear in N since d is a small constant. Simhash is very useful for near-duplicate detection as we will see.

2.3 Near-duplicate detection using Simhash

2.3.1 Probabilistic bit flipping

Consider the following problem: Given a d-bit integer a and a collection of N d-bit integers, find all integers from this collection whose bitwise-Hamming distance to a is at most h, meaning that at most h of the d bits may be different between a and such an integer.

If $h = 1$, the following is a possible solution to this problem:

1. Put all of the N integers in a hash table.

2. First check if a has a match in the hash table (distance of 0).

3. For each bit position i from 1 to d of integer a, take a but flip its i^{th} bit and check if the result is in the hash table.

This algorithm runs in $O(d)$ time.
If $h = 2$, we would add a last step to this procedure to check all possible combinations of two simultaneous bit flips.
In general, this type of brute force bit-flipping has a runtime of $O\left(\binom{d}{h}\right)$, which quickly blows up as h increases.

Sood and Loghinov have developed a smarter probabilistic bit-flipping approach [9]. Their approach relies on useful information that is discarded during the Simhash algorithm procedure. Specifically, the d-entry vector V mentioned in section 2.2 above. To explain the intuition behind their approach, consider the example of a Simhash algorithm that when applied to a vector u produces $V = [83, -2, 0.01, 7]$.
This V would lead to a Simhash value of 1011, but a small change in the original u vector is likely to change the third entry of V enough that the eventual third Simhash bit would flip and the Simhash value would become 1001. Indeed, since that scenario is much more likely than any other that would lead to other bits flipping, we say that this third bit of the Simhash is the most "volatile". Sood and Loghinov's algorithm leverages that insight to come up with a smart ordering of the bit flips that should be attempted when solving the aforementioned problem. Specifically, the bits are first sorted in order of decreasing volatility, and an algorithm using a structure called Volatility Ordered Set Heap (VOSH), will generate the optimal bit-flipping order that should be attempted.
With high probability, this approach will find all neighbors h bits away from a after having attempted only a small fraction of the bit flips that the brute force algorithm would. For example, this thesis empirically found that for their data set, for $d = 64$, VOSH takes 17

flips instead of 64 for $h = 1$, 152 flips instead of 2016 for $h = 2$, and 675 flips instead of 41664 for $h = 3$.

While these results are significantly better than the brute force method, there are two reasons why this approach would still not be adequate for our general clustering problem:

1. The number of flips required still grows exponentially with respect to the distance h, so for large h such as 16 of the 64 bits, it would become intractable, and a simple linear search would perform better.

2. In general, if we can afford to, we prefer to have d that is not too small in order to preserve more information when moving from u to its d-bit representation. However, VOSH and bit-flipping are exponentially worse as d grows, whereas a linear search algorithm would take the same asymptotic runtime up to a constant factor when d grows.

These two drawbacks imply that if we increase d and h by the same factor, meaning that we are looking for the same fraction of differing bits but with a more accurate bit representation, then the algorithm would take doubly exponentially longer to compute. Therefore, this bit-flipping approach, while great for near-duplicate detection, is not adequate for larger h distances and general clustering applications.

2.3.2 Block permuted Hamming search

Google researchers have come up with an efficient, exact algorithm for near-duplicate detection in large-scale datasets [10]. Consider the same problem as before: Given a fingerprint f

of d bits and a collection of $N = 2^b$ fingerprints of d bits each, where b is significantly smaller than d, (for example, $d = 64$ and $N = 2^{34}$), find all fingerprints that are within a distance of h from f.

The algorithm is the following (and it is explained quite well in the Sood and Loghinov paper [9]):

1. Split f into $G > h$ blocks of $y = d/G$ consecutive bits.
2. If f and a are at a distance of at most h from each other, then there are at least $G - h$ entire blocks in f and a that match, since the h differing bits can be spread out among at most h blocks. Pick a number g such that $1 \leq g \leq G - h$. For every fingerprint in the collection, group all possible combinations of g blocks at the head of the fingerprint. For a fingerprint that is within distance h of f, one of these combinations must match the first gy bits of f. There are $\binom{G}{g}$ different combinations of these blocks (we do not care about the relative ordering of these g blocks with respect to each other). Each combination is called a block permutation π_i. For each π_i, we end up with a table T_i of permuted fingerprints. We store all T_i tables in memory.
3. Sort each table of permuted fingerprints.
4. Binary search over each of the sorted tables to find matches to the first gy bits of f. The matches (if any) will be consecutive, and it is expected that there will be 2^{b-gy} of them.
5. Linearly scan the list of matches, checking if each match has a Hamming distance within h of f.

Two examples from the Google paper illustrate this process very well, both in the case where $d = 64$, $N = 2^b = 2^{34} = 8$Billion, $h = 3$.

1. Pick $G = 6$ and $g = 3$. So we split f into 6 blocks of 10 or 11 bits each. Then there are $\binom{6}{3} = 20$ ways of building a header of size 3 blocks. For some block permutation π_i, a header of size 31 bits on average will match the first 31 bits of $\pi_i(f)$, and it is expected that $2^{34-31} = 8$ fingerprints will have such a header. We linearly scan these 8 fingerprints checking their distance to $\pi_i(f)$.

2. Pick $G = 4$ and $g = 1$. So we split f into 4 blocks of 16 bits each. Then there are $\binom{4}{1} = 4$ ways of building a header of size 1 block. For some block permutation π_i, a header of size 16 bits will match the first 16 bits of $\pi_i(f)$, and it is expected that $2^{34-16} = 256,000$ fingerprints will have such a header. We linearly scan these 256,000 fingerprints checking their distance to $\pi_i(f)$.

We notice that there is a tradeoff between the number of tables that we store and the number of linear scans we have to perform. However, in both cases, if we apply this algorithm to find the neighbors of all N fingerprints in a collection, we get a much better performance than the $O(N^2 = 2^{68})$ runtime for trivial search.

Unfortunately, this algorithm does not perform so well for combinations of moderate-sized N (in the tens of thousands) and relatively large h (no longer just for near-duplicate detection algorithms). For example, let $d = 64$, $N = 2^{16} = 65,536$, and $h = 16$. Two cases will illustrate why this algorithm is limited in such a setting:

1. If $G = 21$ and $g = 5$, then we have $\binom{21}{5} = 20349$ possible block permutations to build, and one is expected to return on average $2^{16-64/21*5} = 1.7$ matches, whose Hamming

distance to f we can immediately check.

2. If $G = 17$ and $g = 1$, then we have 17 possible block permutations to build, and one is expected to return on average $2^{16-64/17*1} = 4,822$ matches, which we will have to check linearly.

In either case, we notice that in these settings, the benefits of this algorithm do not clearly stand out as much as in the settings of near-duplicate detection over a very large corpus.
Notice too that this algorithm expects the N fingerprints to be distributed randomly among all possible 2^d fingerprints. We cannot generally assume this about a real world dataset of personalized news articles, and so we cannot effectively pick the tradeoff between number of permutations and number of linear scans ahead of time without having an idea of the data's distribution.
Furthermore, this algorithm attempts to efficiently find an exact solution where all near-duplicates are found, presumably in order to eliminate the redundant fingerprints and trim the dataset. In order to do so, we notice that for each query, the algorithm produces many permutation tables, only one of which may actually be useful.
For applications of general clustering however, we care about actually splitting up the dataset into clusters. In these situations, it is actually not so necessary to go through an exact nearest neighbor solution like this algorithm does. As Indyk and Motwani say, "Since the selection of features and the use of a distance metric in the applications are rather heuristic and merely an attempt to make mathematically precise what is after all an essentially aesthetic notion of similarity, it seems like an overkill to insist on the absolute nearest neighbor, in fact determining an ϵ-approximate nearest neighbor for a reasonable value of ϵ, say a small constant, should suffice for most practical purposes" [7]. If we adopt this point of view on general clustering, then instead of insisting on producing all permutation tables in order to

guarantee that one of them will produce all matches in the data, we could attempt a more approximate probabilistic approach, depending on how much accuracy we are willing to give up for the sake of gains in performance.

2.4 Randomized approximation for general clustering

Ravichandran et al. describe a randomized algorithm that helps reduce the time complexity of calculating a similarity matrix between n elements of d bits from n^2d to practically $nd + n \log n$ [11]. An example will best illustrate the intuition behind their approach: Consider the following 8-bit fingerprints a and b:

$$a = 10111010$$

$$b = 00111010$$

These two integers have a Hamming distance of 1. If they were part of a collection of N fingerprints and we sorted that collection in lexicographic order, b is expected to be in the first half of the sorted list, whereas a is expected to be in the second half of the sorted list, very far away from b. This is because even though these two fingerprints differ only in 1 bit, this differing bit happens to be the most significant bit.
If we apply a permutation function π_i to a and b that swaps the first and last bits of each, we get:

$$\pi_i(a) = 00111011$$

$$\pi_i(b) = 00111010$$

These two fingerprints still have the same Hamming distance of 1, but if we apply this permutation to all of the N fingerprints in the collection and sorted this list of permuted fingerprints, we would find that $\pi_i(a)$ and $\pi_i(b)$ would be very close to each other in the sorted list.

The algorithm proposed in this paper extends the search algorithm PLEB (Point Location in Equal Balls) introduced by Indyk and Motwani [7]. Here is the algorithm:

1. Choose a large prime p, and randomly pick integers a and b, $a, b < p$. For each fingerprint x in the collection, produce a permuted fingerprint

$$x' = (ax + b) \mod p$$

   This gives us a list of permuted fingerprints. Repeat this process q times, each time picking different random numbers a and b to produce q different lists of permuted fingerprints.

2. For each of these q lists, lexicographically sort the list.

3. For each of the q sorted lists, calculate the Hamming distance from every fingerprint to its B closest neighbors in the sorted list (using the original un-permuted fingerprints). If this distance is less than the desired h, output the pair of original un-permuted fingerprints.

After having reduced the n vectors of k features to d bits, the above algorithm takes $O(q(n \log n + nBd)$ time to produce a list of pairs of similar fingerprints. Since B and q are constants, the time for this algorithm is therefore $O(nd + n \log n)$, and if we consider d to be a constant too, the runtime becomes $O(n \log n)$.

Note: It is not clear which values of q and B to pick ahead of time.
Moreover, this algorithm focuses on returning lists of similar fingerprints for each fingerprint, so it does not actually return a clustering of the fingerprints.

As an aside, it is important to understand why the authors are not performing their nearest neighbor search with a data structure such as a k-d tree (and why this thesis is not either). K-d trees can be thought of as generalized binary trees that work in k dimensions instead of just one. K-d trees are traditionally used for nearest neighbor searches [12] because a query over a set of N elements has an average runtime of $O(\log N)$. However, the performance of k-d trees provably degenerates into a linear search when the number of dimensions k is comparable to the number of elements N [13].
That is why we first reduce the number of dimensions using Locality Sensitive Hashing techniques such as Simhash.

2.5 DBSCAN

DBSCAN (short for Density-Based Spatial Clustering of Applications with Noise) was introduced in 1996 by Ester et al. as an efficient data-clustering algorithm [14]. Given a set of points, DBSCAN produces clusters of points that are densely packed together. DBSCAN is characterized by two parameters:

- Eps: The minimum distance below which two points are considered close.
- MinPts: The minimum number of of points that have to be within Eps of point A in order for point A and its neighbors to define a cluster or be added to an existing one, otherwise A is (temporarily) considered noise.

DBSCAN classifies the given points as either core points, border points, or noise.

- A core point is a point that has a distance of less than Eps to at least MinPts.
- A border point (also called a density-reachable point) is a point that can be directly reached (has a distance of less than Eps) from a core point
- Any point that is not directly reached by a core point is considered noise

Here is a description of DBSAN:
Start at a point p in the set of points. Perform a "regionQuery" to get all the points that are density-reachable from p. If p is not a core point, label it as noise for now and stop, otherwise, p and its neighbors define a cluster. In that case, for each of p's neighbors, check if it is a core point and if so add its neighbors to the clusters. Move on to the next point in the set and repeat this process.

Note: If two points p and q are in the same cluster, then there exists a path from p to q such that every point along the path is a core point.

Complexity: Each point in the database will be visited, and for each one, we have to find its neighbors at a distance of less than Eps. DBSCAN can be given a precomputed matrix of distances of size N^2, to avoid distance re-computations, but that approach would require $O(N^2)$ space, not to mention the computation required to produce the N^2 matrix in the first place.

Chapter 3

Algorithmic Approach

First, we present the major contribution of this thesis: a real-time algorithm that produces a clustering of text documents out of their tf-idf vectors. Then we move on to describing a series of optimizations to speed up the process of creating tf-idf vectors.

3.1 Real-time algorithm for clustering tf-idf vectors

In this section, we outline this thesis's main contribution: An online algorithm to cluster the collection of tf-idf vectors using a variant of Simhash, followed by an approximate nearest neighbors computation, followed by a slightly modified DBSCAN run.

3.1.1 Parallelized Simhash reduction

As previously noted, a cosine similarity computation between two large vectors is very time consuming. Instead, a dimensionality reduction technique such as Simhash can reduce the problem to comparing d-bit integers. In addition, Simhash produces fingerprints that can be sorted, which may present many gains for search algorithms.

Here, we describe the Simhash reduction from a tf-idf vector to a 64-bit Simhash fingerprint. The algorithm is inspired from [8].

Algorithm 1 Simhash(v)

Data: v: tf-idf vector
dct: a dictionary mapping words to indices
words: list of stems of document corresponding to v
Result: simhash: a 64-bit integer

```
simhash ← 0
W ← array of 64 zero floats
for word in words do
    φ ← UniformHash(word).toInt()              ▷ Use the hash as a 64-bit int
    counter ← 63 while φ > 0 do
        bit ← φ  mod 2
        if bit then
            W[counter] ← W[counter] + v[dct(word)]      ▷ Add tf-idf weight
        end
        else
            W[counter] ← W[counter] - v[dct(word)]      ▷ Subtract tf-idf weight
        end
        φ ← φ >> 1                                     ▷ Right shift
        counter ← counter - 1
    end
end
for i = 0 to 63 do
    if W[i] ≥ 0 then
        simhash← simhash += 1                  ▷ positive weight, set bit to 1
    end
    if i < 63 then
        simhash ← simhash << 1                 ▷ while not the last bit, left shift
    end
end
return simhash
```

We can parallelize this process by running multiple threads in parallel, each of which reduces one document to a Simhash.

3.1.2 Approximate nearest neighbors computation

Given N Simhash fingerprints, we would like to create an adjacency list where each fingerprint is only connected to fingerprints it is similar to (within a distance of h or less from). For a reasonable value of h, the adjacency list we obtain will have a size much smaller than N^2, but a basic approach to creating the list would involve comparing every fingerprint to every other one, which is a $O(N^2)$ algorithm. Intuitively, we would instead like a method that involves comparing each fingerprint only to other fingerprints that are most likely to be similar to it. To achieve this, we can draw inspiration from the permutation algorithms presented in subsection 2.3.2 and section 2.4. Unfortunately, in both those sections, the algorithms described do not make recommendations about the number of permutations q to create, or the search parameter B to be used, and do not discuss the effect that different initial data distributions can have on the choice of these parameters.

In this section, we present two randomized algorithms that achieve high accuracy in a fraction of the time required by the basic quadratic approach. In a later section, we first show that both algorithms perform extremely well. Then, we examine the performance tradeoffs between the two algorithms to make specific recommendations about when to use each.

Both algorithms rely on the following intuition: Instead of making guesses about the data distribution and the mapping between parameter values and achieved level of accuracy, we sample from the data at random and tailor the parameter values to achieve the desired level of accuracy on the sample. These tailored parameters will perform extremely well on the original larger problem.

3.1.2.1 "Fixed Rounds" and "Fixed Ratio" algorithms

Both algorithms begin with the same subroutine, which we call "the sampling phase". Given a list L of N Simhash fingerprints, a beam search parameter B (which we explain how to select an optimal value for in subsubsection 4.2.2.2), a distance h, a table T, and a desired level of accuracy p:

1. Create a sample S of $M = s * N$ fingerprints, where $0 < s < 1$ by selecting from the N given fingerprints at random (we explain how to select a value for s in later sections).

2. Using two nested for loops, create an $M \times M$ matrix where $M_{i,j}$ represents the Hamming distance between S_i and S_j.

3. Count how many pairs (i, j) have a distance of at most h. Call this number the total number of pairs.

4. Set the number of pairs seen so far to 0.

5. Create a list of M new permuted fingerprints by applying a random permutation function π_i (the permutation function is discussed in detail next).

6. Sort the new list.

7. For each fingerprint in the sorted list, compute the Hamming distance between its original (un-permuted) version and each of the original fingerprints of its next $s * B$ neighbors (an efficient Hamming distance computation is outlined later).

8. For every pair of fingerprints considered, if the pair has not already been added to table T and if its Hamming distance is at most h, then add the pair to T and increment the number of pairs seen so far.

9. At the end of step 8, if the number of pairs seen so far is less than a fraction p of the total number of pairs from step 3, go back to step 5.

Both algorithms begin with this subroutine. However, here are the differences between the two:

- **Algorithm 1**: While running the sampling subroutine, this algorithm keeps track of the number of times steps 5 to 9 are repeated. At the end of the routine, it uses that number as q, the number of permutation rounds, on the initial problem with N fingerprints and a beam search parameter B. We refer to this algorithm as "Fixed Rounds".

- **Algorithm 2**: While running the sampling subroutine, this algorithm keeps track of ϵ, the ratio between the number of new pairs found in the last round, over the number of new pairs found in the first round. At the end of the routine, this algorithm moves on to the original problem with N fingerprints and beam search parameter B, and keeps performing permutation rounds while keeping track of the ratio of the number of new pairs found in the last round over the number of new pairs found in the first round. It stops when this ratio is less than or equal to ϵ. We refer to this algorithm as "Fixed Ratio".

Here is pseudo-code for Fixed Ratio: (Fixed Rounds is similar to the algorithm in section 2.4)

Algorithm 2 Fixed Ratio

Data: fingerprints: List of N fingerprints
B: Beam search width
h: Maximum Hamming distance between similar fingerprints
p: Desired level of accuracy
T: Table
Result: Adjacency list of similar fingerprints

```
ε = sampling_phase(N, B, h, p, T)
Hashtable table
iteration ← 1
counter1 ← 0
counteri ← 0
while counteri < ε*counter1 do
    permuted_list ← SORT(permute_all(fingerprints))
    counteri ← 0                                          ▷ Re-initialize counteri
    for i from 1 to N do
        for b from 1 to B do
            if i + b < N and haven't seen this pair in table before then
                if distance(ORIG(permuted_list[i]), ORIG(permuted_list[i+b]))
                < h then
                    counteri ← counteri + 1
                    table.add(ORIG(permuted_list[i], ORIG(permuted_list[i+b]))
                    table.add(ORIG(permuted_list[i+b], ORIG(permuted_list[i]))
                end
            end
        end
    end
    if iteration = 1 then
        counter1 ← counteri                  ▷ Set counter1 for future comparisons
    end
    iteration ← iteration + 1
end
adj_lst ← empty list                          ▷ Create adjacency list from table
for (a1, a2) in table do
    if a1 not in table then
        adj_lst[a1] ← empty list                          ▷ Create a list for a1
    end
    adj_lst[a1].append(a2)                          ▷ Add a2 as a neighbor to a1
end
return adj_lst
```

3.1.2.2 Better permutation function

Ravichandran et al. [11] proposed an approximation of a permutation function $\pi_i(x) = (ax + b) \mod p$ with p some large number and $a, b < p$ chosen randomly.
However, this approximation, while fast to compute, produces approximately pairwise-independent fingerprints. Therefore, when this permutation is applied on x_1 and x_2, even though we know the relationship between x_1 and x_2 (for example their Hamming distance), we still cannot predict anything about the relationship between $\pi_i(x_1)$ and $\pi_i(x_2)$ [15]. Therefore, sorting a list of fingerprints obtained through such a permutation would not do a good job of grouping together fingerprints with low Hamming distances.

We propose a permutation function that requires a d-iteration for-loop but conserves Hamming distances and therefore has a high likelihood of grouping together similar fingerprints. The function is based on the Fisher-Yates shuffle [16], which guarantees that every permutation is equally likely to be chosen.

Algorithm 3 Create Permutation

Data: d: Number of bits in a fingerprint
ordering: List of d integers
Result: Permuted ordering

for *i from 1 to d* **do**
  ordering[i] $\leftarrow i$ ▷ Initialize ordering from 1 to d
end

for *i from 1 to* $d-1$ **do**
  $j \leftarrow$ random integer $1 \leq j \leq d-i$
  swap(ordering[i], ordering[$i+j$])
end
return ordering

Algorithm 4 Permute Fingerprint

Data: d: Number of bits in a fingerprint
permuted_ordering: List of d out of order integers
x: d-bit fingerprint to permute
Result: Permuted fingerprint according to permuted_ordering

ret $\leftarrow 0$
for *i from 1 to d* **do**
  if *not not (x & (1LL << permuted_ordering[i]))* **then**
    ret $\leftarrow$ ret | (1LL << i)
  end
  return ret
end

3.1.2.3 Efficient Hamming distance computation

The Hamming distance between fingerprints a and b is defined as the number of set bits in the result of performing a XOR b.

A basic way of counting the set bits of a d-bit integer is to repeatedly left bit-shift the num-

ber and mod by 2 in a d-iteration loop, while counting the number of loop iterations that yielded remainders of 1.

Fortunately, we can even do better and get a constant-time algorithm. We use Java's Integer.bitCount method [17] and extended it for 64-bit integers: (This method is extensible to any power of 2 value of d)

Algorithm 5 Hamming Distance

Data: a, b : Two 64-bit fingerprints
Result: The Hamming distance between a and b

$x \leftarrow a$ XOR b
$x \leftarrow x$ - ((x >> 1LL) & 0x5555555555555555)
$x \leftarrow (x$ & 0x3333333333333333) + ((x >> 2LL) & 0x3333333333333333)
$x \leftarrow ((x + (x >> 4LL))$ & 0x0F0F0F0F0F0F0F0F)
$x \leftarrow (x * (0x0101010101010101)) >> 56LL$
return x

3.1.2.4 Optimizations

We explore a few potential optimizations that are not core to the theoretical algorithms but might lead to significant performance gains.

Cache optimization

In the Fixed Rounds algorithm, after running the sampling phase, we get a value of q, the number of sorted lists of permuted fingerprints to create. We could build the permuted lists more cache-consciously than simply permuting all the fingerprints once, then permuting all of them again, etc. q times. We could instead select a block of fingerprints the size of our desired cache level, and permute all the fingerprints in that block repeatedly q times, then

move on to the next block, until all fingerprints have been permuted q times. By doing so, each fingerprint would only need to be loaded into memory once instead of q times, which could lead to a considerable speedup.

Parallelism

We can use multithreading in Fixed Rounds and Fixed Ratio when performing the beam search such that thread i can compare permuted_fingerprint i to its next B neighbors, then permuted_fingerprint i+number of threads to its next B neighbors, etc. We have to be careful about race conditions because threads may interfere with each other when they probe the table or increment counters. That is why we must put mutex locks over the table.

3.1.3 Simplified DBSCAN

Having created the adjacency list of fingerprints within a distance of h from each other, we run a slightly modified version of DBSCAN on this adjacency list.

There is no longer a notion of Eps, since this has effectively been replaced by the h constraint in the previous steps. We avoid recomputing or even computing distances because each regionQuery of a point immediately returns its list of neighbors in the adjacency list. We do not need an N^2-sized matrix either.

At the end of the DBSCAN run, we return a clustering of the fingerprints, which corresponds to a clustering of the original text documents.

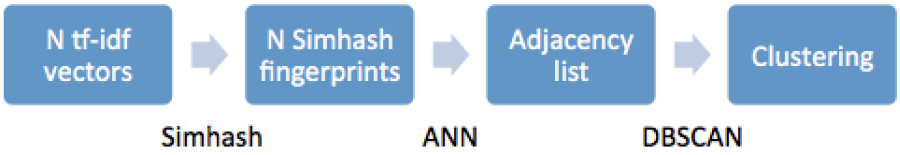

Figure 3.1: Flow of data through the algorithm

3.1.4 Intuition behind algorithm

There are 5 reasons why we expect this algorithm to perform well:

1. Just like the Indyk and Motwani proof of their PLEB algorithm as well as the Ravichandran et al. paper's proof: as q and B increase, it becomes exponentially more unlikely that there are two similar fingerprints that we never compare.

2. We use an exact and unbiased permutation function instead of an approximation that leads to nearly pairwise-independent outputs.

3. **Self-tuning, no need for domain expertise about data**
   The algorithm in section 2.4 requires two hyper-parameters B and q that greatly impact performance, and that are to be chosen by the programmer based on domain expertise about the dataset, as well as advanced knowledge about the mathematical properties of the algorithm. Both of these requirements on the part of the programmer are problematic in the real world. Similarly, the Google algorithm in [10] does not make a recommendation for the number of permutation tables to compute, in part because the optimal number would depend on the distribution of the dataset. In contrast, the Fixed Ratio and Fixed Rounds algorithms in this thesis only require a desired level of accuracy, which is a concept that any programmer has a good intuition about. Both algorithms then leverage sampling in order to fine-tune their own parameters, q in the

case of Fixed Rounds and ϵ in the case of Fixed Ratio, and have empirically been shown to work really well on a range of different data distributions. The reason why we can afford to compute a quadratic time algorithm for a representative sample is because the representative sample does not have to be large, since our datasets are moderately sized, whereas the Google algorithm was designed for extremely large datasets.

4. **Incremental improvement instead of "All or Nothing"**

   The Google algorithm described in subsection 2.3.2 returns an exact answer of all matching pairs. In that algorithm, we first create a number of permutation tables, then binary search them to find a match to a query. Only one of the searched tables will contain matches, which we will then have to linearly scan one at a time. During this linear scan, we have no knowledge of the accuracy achieved so far and therefore cannot reliably decide to stop scanning for matches when we are satisfied. In our proposed Fixed Rounds and Fixed Ratio algorithms, each list of permuted fingerprints returns matches, and the algorithm is able to stop at any point depending on the desired level of accuracy. Controlling the trade-off between speed and accuracy is especially important in real-time operations where the user may not appreciate 100% accuracy if it came at the cost of longer loading times

5. Our modified DBSCAN will make N calls to regionQuery but since regionQuery is now made to return immediately, DBSCAN avoids computing and recomputing distances and does not require any precomputed matrix of size N^2 either.

6. If the clustering we return differs from the optimal clustering, it means that there exist fingerprints i and j that should be labeled in the same cluster but that we somehow miss. In order for that to be true, we have to not only miss the comparison between i and j, but also miss at least one edge on every path consisting of core nodes between i

and j, which becomes increasingly unlikely as h grows beyond near-duplicate detection levels and as the number of pairs we have already found grows.

3.2 Speeding up the creation of tf-idf vectors

We maintain a dictionary that maps each possible word to an index. We can obtain this dictionary from previous runs on corpora from similar sources such that words are included in the dictionary if they appear more than some minimum ϵ number of times. There are 3 steps involved in creating tf-idf vectors. First, creating a term frequency vector for every document in the corpus. Second, creating one global inverse document frequency vector for the entire corpus. Third, multiplying each term frequency vector by the global inverse document frequency vector. We propose a way to parallelize each of the three steps.

3.2.1 Parallelized computation of term frequencies

The serial version of the algorithm for term frequencies would be to simply have two nested loops: For each word of each text document, increment the tf vector for that text document at the index that the word maps to (according to the dictionary).
If we want to parallelize our code to maximize resource usage, we can have each thread loop over the words of a document, but run multiple threads in parallel, each looking at a particular document. Since our text corpus is represented in memory as a list of lists of stems, it is interesting for caching efficiency to experiment with the "chunksize" of the parallelism, i.e. the number of consecutive documents in memory that each thread should

run on. If there are t threads, a chunksize of 1 means that the i^{th} thread will run on the i^{th} document, then on the $(i+t)^{th}$ document, etc. A chunksize of $\frac{\text{number of questions}}{\text{number of threads}}$ means that the i^{th} thread will run on the $(i*t)^{th}$ document, then on the $(i*t+1)^{th}$ document, etc. all the way until the $(i*t+t-1)^{th}$ document.

3.2.2 Parallelized computation of inverse document frequencies

The serial version of the algorithm for inverse document frequency would be to simply have two nested loops: For each word of each text document, if the word has not occurred in this text document before, increment the global idf vector at the index that the word maps to (according to the dictionary).
If we want to parallelize our code to maximize resource usage, we can have each thread loop over the words of a document, but run multiple threads in parallel, each looking at a particular document. Again, we experiment with different chunksizes.
However, we have to be careful about race conditions because two or more threads may attempt to increment the global idf vector at the same word index (if they encounter the same word for the first time in a document at the same time). Therefore, we must add mutex locks to overcome this problem. There are a few different strategies for locking that we can choose from:

1. Coarse-grained locking: One lock for the entire idf vector, meaning that every time any thread wants to increment any entry of the idf vector, it must acquire a lock over the entire vector, and no other thread may modify the vector until the previous thread has released the lock. Therefore, this approach might cause threads to wait on each other for too long and would force serial execution of code.

2. Fine-grained locking: One lock for each of the entries in the idf vector, meaning that a thread only needs to acquire a lock if it needs to modify the entry for a specific word, which should minimize thread contention. However, this approach may come at the cost of significant locking overhead [18].

3. Medium-grained locking: One lock for each block of b entries in the idf vector, meaning that a thread acquires a lock if it needs to modify the entry of one of the words in the block corresponding to that lock. This approach attempts to achieve a balance between minimizing thread contention and minimizing locking overhead.

3.2.3 Parallelized multiplication of tfs by idf

Once we have N tf vectors and one global idf vector, we need to produce N tf-idf vectors by multiplying the i^{th} entry of each tf vector by the i^{th} entry of the idf vector. This can be achieved serially with two for loops: For each entry of each tf vector, multiply that entry by the corresponding entry in the idf vector. We can parallelize this code by having multiple threads, each responsible for its own tf vector multiplication. Again we can experiment with different chunksizes for increased caching benefits.

Fortunately, there is an additional way to add parallelism to this process: Single Instruction, Multiple Data (SIMD) instructions. Most new processors support SIMD instructions, which allow instructions to be applied simultaneously on a vectorized block of data. Such instructions are very useful for image processing for example [19]. In our case, instead of multiplying the i^{th} entry of a tf vector by the i^{th} entry of the idf vector, we can simultaneously multiply the i, $i+1$, $i+2$, ..., $i+v-1$ entries of a tf vector by the i, $i+1$, $i+2$, ..., $i+v-1$ entries of the idf vector. In theory, we can speed up the computation by a factor of v, where v is the size of a SIMD vector.

Chapter 4

Experimental Analysis

In this chapter, we first describe the setup of our experiments, then we describe a set of experiments and analyze and interpret their results.

4.1 Setup and environment

4.1.1 Dataset

Stack Overflow is a collaboratively-edited question and answer site for programmers. A corpus of Stack Overflow question pages is appropriate for our purposes because each question's page makes up a text document and many questions are related to similar topics and can

be clustered in a meaningful way. In fact, some work has already been done on duplicate detection of Stack Overflow questions using (offline) latent topic models [20]. We use Stack Exchange's free API to download an anonymized set of questions from Stack Overflow, we then limit the dataset to only pages written in English. Since part of our focus is achieving real-time clustering, we would like a moderately sized dataset. We randomly select a subset of the data of size N documents, where $N = 2^{16} = 65,536$.

4.1.2 Stemming and tokenizing

Before we can make sense of our text data, we need to perform some pre-processing. We use a Python library called Natural Language Toolkit (NLTK) [21] to tokenize each document, then the NLTK "Porter Stemmer" to stem each token . As an aside, stemming a word means removing morphological affixes from it, leaving only the word stem, which can greatly impact the resulting tf-idf vectors [22]. For example, the stem of both "running" and "runner" is "run". Tokenizing and stemming allow us to represent a text document by a list of its stems, which can then be compared to the stems of another text document. In our experiment, our text corpus will therefore be represented as a list of lists of stems.

This pre-processing takes some time to perform and cannot be done in real time. However, if we consider the application of clustering a LinkedIn or Facebook member's feed of documents, each document only needs to be tokenized and stemmed once and can then be used across all members. Therefore, this pre-processing step is usually performed when a text document is first ingested and the resulting list of stems is included as metadata. The same can be said about the process of creating tf-idf vectors, and even reducing a tf-idf vector to a Simhash fingerprint: a tf-idf vector for a text document and its corresponding Simhash fingerprint can both be created offline at the time the document is ingested into the database, and may

be reused across all users of the service.

In this experiment, we build the dictionary that maps stems of words to integer indices from a random subset of the Stack Overflow corpus of size 100,000 documents. We get a dictionary containing around 58,000 unique words. A tf-idf vector will thus be a vector of 58000 entries, where entry i corresponds to the word whose stem maps to i in the dictionary.

4.1.3 Using Cython and C++

Cython is a language that allows Python programs to call C and C++ code natively. Cython is compiled down to C so it has C-like performance [23], which is very desirable for us since minimizing runtime and overhead is one of our primary concerns. In addition, Cython provides convenient support for multi-threading. Finally, since NLTK is a Python library, the resulting data we obtain must be compressed as a Python "pickle" file, and can only be opened in Python.

Therefore, in our experiment, we use Python to tokenize and stem our text data, and then open the resulting compressed data. Then we use Cython to build tf-idf vectors and reduce them to Simhash fingerprints using C-like code with C-performance that can be called from our Python files. Once we have the Simhash fingerprints, we write them to disk, and load them into memory in a C++ program, and perform the clustering in a C++ environment.

4.1.4 Hardware used

We ran all experiments on an Intel Core i7 2.7Ghz quad-core machine running OS X El Capitan, with 16GB of RAM, 500GB of disk space, L1 Cache of 32KB, L2 Cache of 256KB, L3 Cache of 6MB, and support for AVX SIMD instructions with 256-bit registers.

4.2 Results and analysis

4.2.1 Performance of parallelized Simhash

The Simhash reduction allows us to compute efficient Hamming distances between 64-bit fingerprints instead of expensive cosine similarities over large tf-idf vectors. However, if we want to cluster a collection of tf-idf vectors in real time, we must be able to perform this reduction in real time as well. In this experiment, we show that by running 8 threads in parallel, we can reduce the 65K tf-idf vectors into Simhash fingerprints in just 0.35 seconds, which keeps our hopes of achieving a real-time algorithm alive.
Here is the performance of the Simhash 64-bit reduction for different levels of parallelism:

We immediately notice that the Simhash reduction is extremely fast, especially when combined with multi-threading. However, we notice that the speedup factor decreases as the number of threads grows, especially between 4 and 8 threads. This result is probably due to an expensive scheduling overhead [24], especially given that we are testing the limits of our quad core machine, and because we know that hyper-threading is typically slower than having additional physical cores. [25].

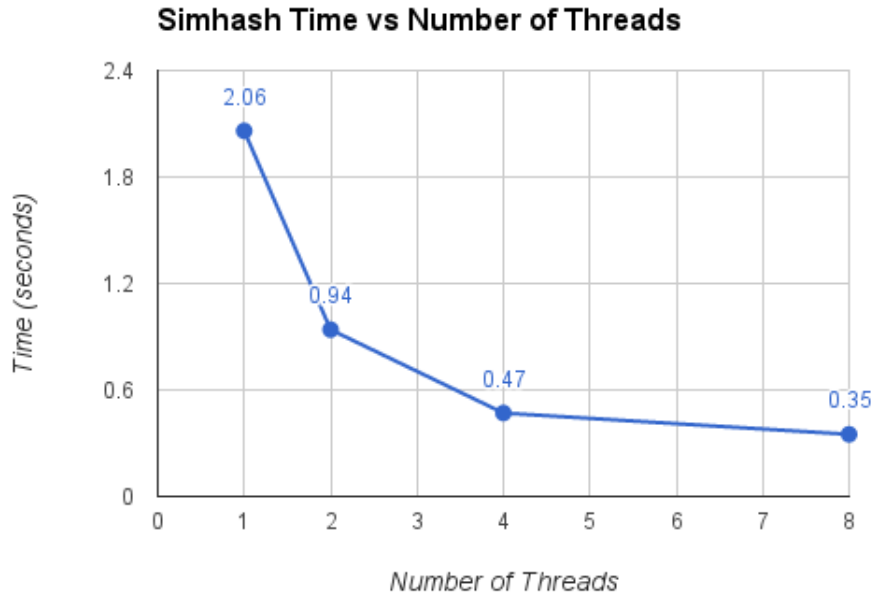

Figure 4.1: Time for Simhash reduction as a function of number of threads

4.2.2 Performance of Approximate Nearest Neighbors

For the rest of this analysis, we will use "matching pair" to refer to a pair of fingerprints (i, j) that are within Hamming distance h of each other, where $h = 16$.

4.2.2.1 Comparison of proposed ANN to previous work

Comparison to basic quadratic algorithm

There exists a basic quadratic runtime approach for a nearest neighbor computation: Create a matrix of $N \times N$ entries, and compare each pair of fingerprints inside two nested for loops. It is important to perform this computation for two reasons: first, to see if the basic approach

is fast enough to yield a runtime algorithm, and second, to store the total number of matches found and use it as the gold standard to which we compare the number of matches found by our approximation. We show that this quadratic algorithm completely breaks our real-time requirement and expends a massive amount of resources and energy.

Since $N = 2^{16}$, this algorithm has to create a matrix of $2^{32} > 4$ Billion entries, and compute the distance between $\binom{2^{16}}{2} > 2$ Billion pairs.
Empirically, this quadratic algorithm takes an average of 31 seconds to run, which obviously breaks our real time algorithm requirement. In addition, the computation spills over to disk, since the matrix cannot entirely fit in main memory. We contrast that algorithm's performance with our ANN approach on $q = 100$, $B = 160$. At the end of each round, we compute the accuracy achieved by each algorithm:

$$\text{accuracy} = \frac{\text{\# of matching pairs found so far}}{\text{total \# of matching pairs}}$$

The results are shown in Figure 4.2.

We immediately notice that there is an exponential drop in the marginal fraction of pairs found in a round as the number of rounds increases. This is to be expected because for large enough B, most matching pairs may be found immediately, and will be encountered repeatedly in ensuing rounds and not counted again. Therefore, it is relatively easy to reach a moderate accuracy level such as 70% (in this case it took 12 permutation rounds), but much more difficult to reach high levels of accuracy such as 95% (34 rounds in this experiment). In most clustering applications though, a 90% accuracy rate may be enough for practical purposes. For 90% accuracy, it took 25 permutation rounds in this experiment. After 25 rounds of comparing each fingerprint to its 160 successors, the total number of comparisons

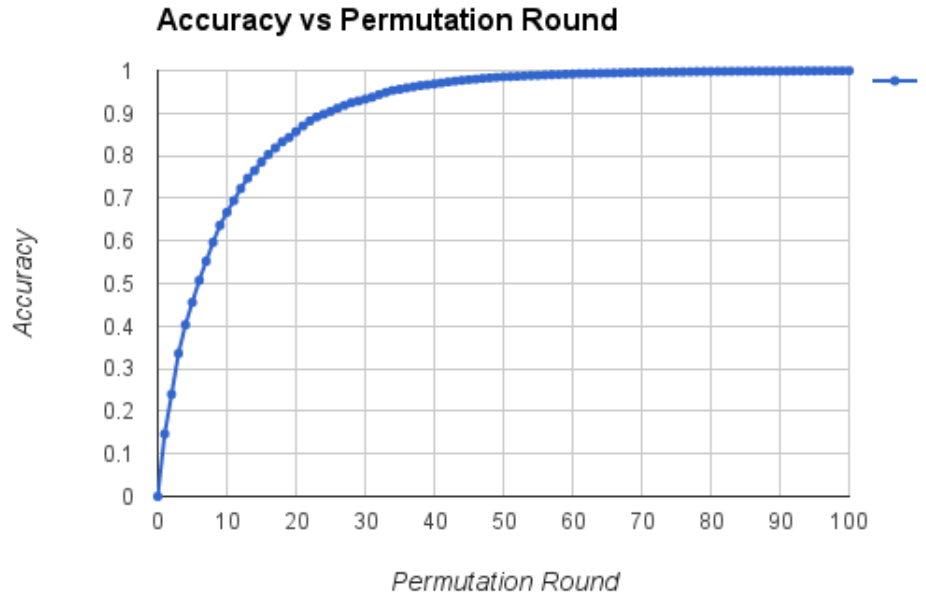

Figure 4.2: Accuracy achieved at the end of each permutation round

we would have performed is:

$$2^{16} * 160 * 25 = 0.12 * \binom{2^{16}}{2}$$

So we can achieve 90% accuracy after performing only 12% of the number of comparisons required by the basic quadratic approach. In addition, our ANN procedure takes only 2.4 seconds to run for 25 rounds, which is much faster than the 31 seconds taken by the basic quadratic approach.

Comparison to the Ravichandran et. al algorithm [11]

We now compare our ANN's performance to that of the randomized approach described in section 2.4 [11].

One of the main advantages of the ANN procedure that we propose is that it does not require the programmer to "guess" optimal values of hyper-parameters because its sampling phase can fine-tune its hyper-parameters to the dataset. However, for the sake of argument, we will ignore this important advantage and choose to isolate the effect of our Fisher-Yates permutation function on accuracy and speed relative to that of the approximate permutation $(ax + b) \mod p$ proposed by [11]. In order to do so, we run both algorithms for 500 rounds with $B = 160$, and at the end of each round, we report the accuracy achieved by each algorithm and we compare the time taken by each algorithm to achieve the same level of accuracy. The results are plotted in Figure 4.3.

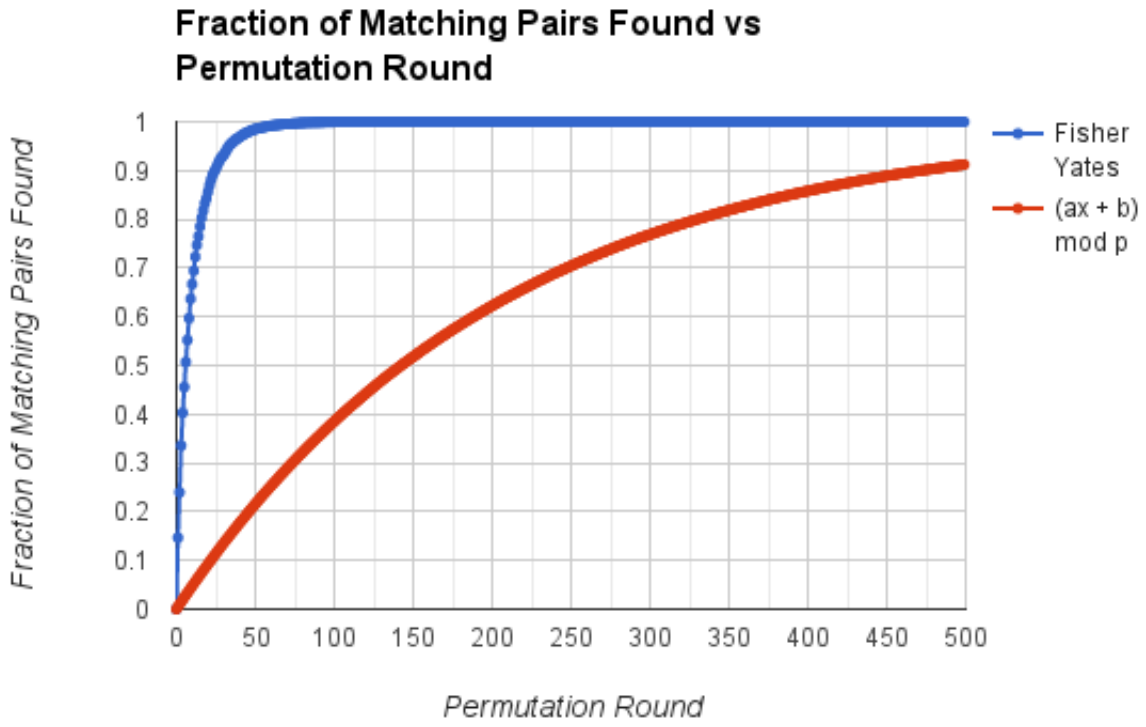

Figure 4.3: Comparison of accuracy between the Fisher Yates shuffle and the approximation function

We notice that while it takes our Fisher-Yates-based shuffle 25 rounds to reach an accuracy

of 90%, it takes the $(ax + b) \mod p$) approximation 472 rounds to reach 90%.
In addition, even though each round of the approximation terminates faster, it still takes 19.2 seconds to reach 90% using the approximation function, whereas it only takes 2.4 seconds to reach 90% using the Fisher-Yates-based permutation. From these results, we conclude that as expected, our permutation function is more accurate than the approximation because it is guaranteed to maintain the same Hamming distance between fingerprints i and j as between permuted fingerprints $\pi(i)$ and $\pi(j)$. More importantly though, we conclude that even though our permutation function was slower to compute because it involved a for loop over all 64 bits of a fingerprint, its accuracy was enough to offset the approximation function's speed. Therefore, even after taking away one of our ANN algorithm's main advantages by setting the same parameters for it as for [11], our algorithm still greatly outperforms [11].

Comparison to the Google algorithm [10]

We now compare our ANN's performance to that of the randomized approach described in subsection 2.3.2 [10], ignoring for now the fact that one of our approach's advantages is a capability to dynamically pick efficient hyper-parameters.
As we saw from our discussion of figure 4.2, our ANN procedure achieves an accuracy of 90% in 2.4 seconds after performing $2^{16} * 160 * 25$ total comparisons, which means that each fingerprint was compared 160*25 = 4,000 times. If we look at the last part of our discussion from subsection 2.3.2, we notice that the expected number of linear scans performed by [10] for $G = 17$ and $g = 1$ assuming that the data are randomly distributed was 4,822. Similarly, the number of permutation tables computed by [10] for $G = 21$ and $g = 5$ was 2,0349.
This is a qualitative comparison, but since the operations being performed in both algorithms are fundamentally the same (Hamming distance computations and bit permutation), we can intuitively compare these numbers to reach a qualitative assessment of our procedure's

performance relative to that of [10]. It is important to note that the number of linear scans performed by [10] assumes that the data are completely randomly distributed, which is not true in real world applications. To see the importance of this fact, consider the extreme case where all fingerprints fall in the block that we linearly scan. In that case, the number of linear scans performed by [10] would be all the fingerprints in the dataset.

4.2.2.2 Performance of the sampling phase

Existence of a small enough, representative enough sample

The success of our algorithm is predicated on our ability to select a sample large enough to be truly representative of the entire dataset, but small enough that we can perform the quadratic algorithm in order to get the total number of matches in the sample and estimate the parameters q and ϵ to be used by Fixed Rounds and Fixed Ratio. We want to show that a sample of size $M = 4,096$ has these two properties.

We explore different values of the fraction s such that our sample of size M consists of $s * N$ randomly selected fingerprints. For each value of s, we compute the basic quadratic algorithm on the randomly selected sample and report the runtimes in Figure 4.4.

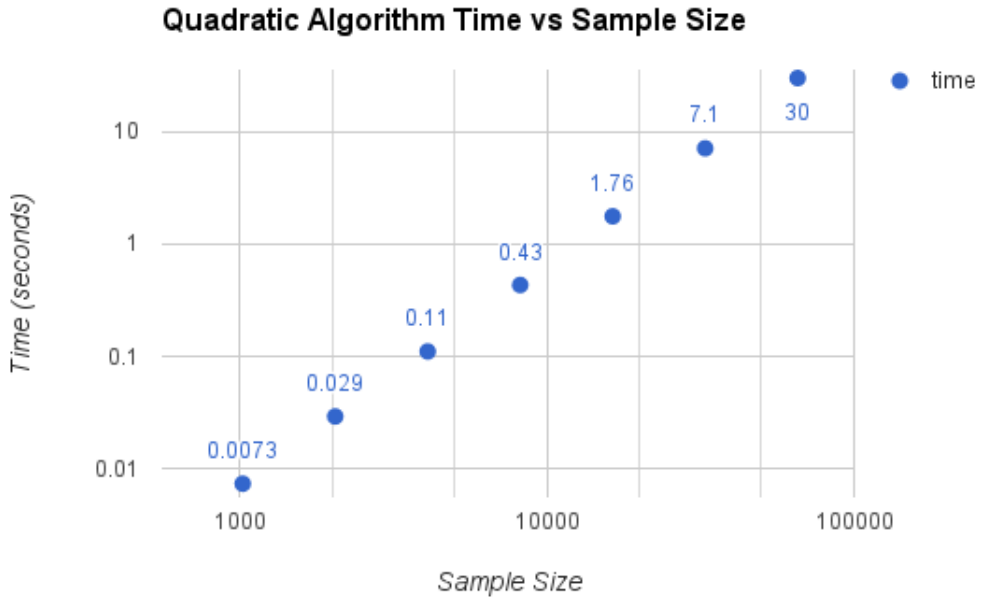

Figure 4.4: Time for basic quadratic algorithm as a function of sample size

We notice that for $M = 4,096$, the quadratic algorithm runs in a relatively inexpensive 0.11 seconds. This corresponds to a fraction $s = \frac{1}{16}$.

We run the permutation algorithm over a sample of size $M = 4096$, for 100 rounds, and with a beam search parameter of 10 (which corresponds to 160/16). In Figure 4.5, we visualize the scatter plot obtained next to the one obtained from the original problem. The two lines we obtain are virtually identical, thus confirming that the sample is representative of the initial dataset.

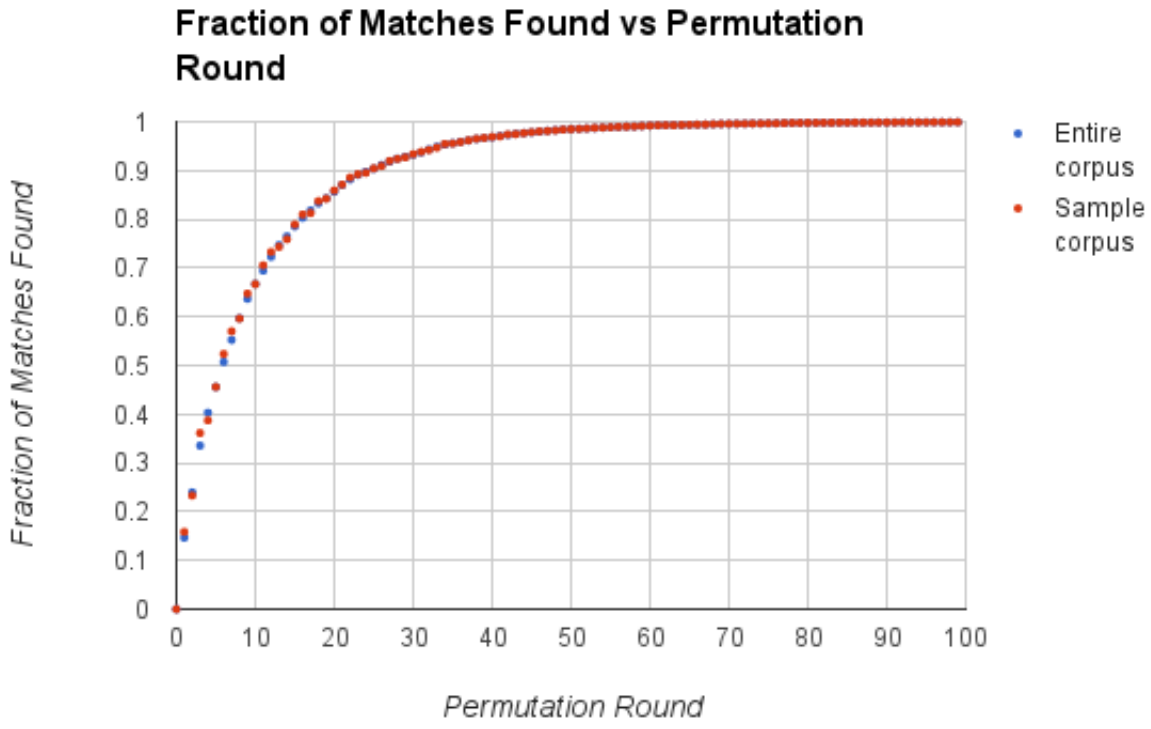

Figure 4.5: Comparison of accuracy between the representative sample and the entire corpus

Optimal table data structure

We explore different possible data structures that could be used during the sampling phase, in order to avoid duplicates when counting the number of new pairs found in each round and when adding an edge between a pair of fingerprints. For the sampling phase, we achieve the best empirical results by using a 2D $M \times M$ distances array initialized with flags indicating that the corresponding pair has not yet been found to be a match.

For a desired accuracy of 90%, the sampling phase takes on average 0.2 seconds, this includes the basic quadratic computation as well as the repeated permutations.

4.2.2.3 Performance of Fixed Rounds and Fixed Ratio

Optimal table data structure

For Fixed Rounds, we need to use a table that keeps track of all matching pairs found. For Fixed Ratio, we need to use a table for the same reason but also in order to count the number of new matching pairs found at the end of each round (in order to make a decision about the stopping condition). For both algorithms, we cannot afford to build $N \times N$ distance arrays like we did in the sampling case because it would require GBs of memory. Instead, we achieve the best empirical results using N C++ unordered_set structures, one for each fingerprint, where a set corresponding to a fingerprint contains the fingerprints that are similar to it. An added benefit of using this data structure is that the list of sets we obtain is also the adjacency list we have to feed into DBSCAN, so we do not have to compute an adjacency list from the table at the end of the algorithm. However, at the end of each round, we do have to count the number of pairs found by aggregating the counts of all the sets (which is fortunately fast because each C++ set keeps metadata around and returns its element count immediately).

Gains from caching in fixed rounds

For fixed rounds, we have the luxury of knowing the number of permutation rounds q ahead of time. Therefore, we attempt to reduce I/O latency by building our permutation lists all at once by permuting each block of b fingerprints q times so that each block is only loaded into cache memory once instead of q times.

Unfortunately, we observe very little speedup (less than 1%). This can be attributed to two reasons:

First, having to malloc q lists of N entries is slower than one list that keeps getting recycled.

Second, and more importantly, I/O latency is most greatly reduced when reducing the num-

ber of loads from main memory into the L3 cache, but not as much for loads into the L2 and L1 caches. In this case, the size of the list of N fingerprints we are permuting is well below the size of the L3 cache, which explains why we do not observe significant latency reduction. However, for N on the order of millions, this technique would be significantly more beneficial.

Gains from parallelism

We can introduce multithreading into both algorithms by running t threads in parallel: Thread i is responsible for comparing the j^{th} permuted fingerprint to its B successors, for all j such that $(j - i) \mod t = 0$.

We have to be careful to prevent race conditions, since multiple threads inserting elements into the same unordered set would overwrite each other's work, and would actually cause segmentation faults because inserts involve delete operations. We place a mutex lock on each of the N unordered sets. The performance of both algorithms for $B = 160$ and a desired accuracy of 85%, as a function of the number of threads run in parallel is reported in Figure 4.6.

We notice that there is a linear speedup between 1 and 2 threads, but locking contention and scheduling overhead greatly reduce the speedup when going from 2 to 4 and 4 to 8 threads.

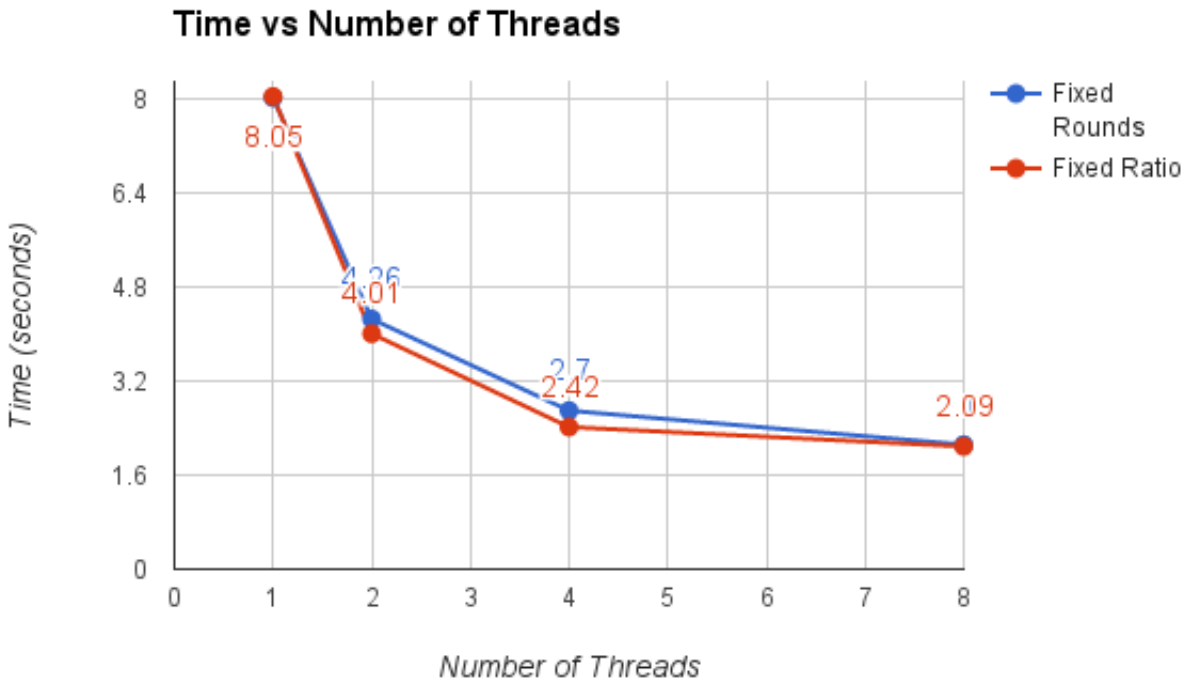

Figure 4.6: Gains from parallelism to both Fixed Rounds and Fixed Ratio

Comparison of Fixed Rounds and Fixed Ratio

We experiment with different values of B and different values of the desired accuracy. For each combination of parameters, we run each algorithm 100 times. The averages are reported in tables 4.1, 4.2, 4.3 and 4.4.

Note: Where applicable, results for Fixed Rounds are reported in blue and to the left, results for Fixed Ratio are reported in red and to the right.

We immediately notice three important results:

1. Both algorithms achieve the required level of accuracy, and both do it within a reason-

B \ Desired Accuracy	75%	80%	85%	90%	95%
160	1.57, 1.52	1.88, 1.75	2.12, 2.09	2.57, 2.51	3.34, 3.19
320	1.69, 1.68	2.07, 1.90	2.32, 2.28	2.42, 2.35	3.29, 3.27
640	1.79, 1.59	1.94, 1.94	2.27, 2.31	2.88, 2.87	3.21, 3.19
960	1.69, 1.90	2.41, 2.08	2.24, 2.38	3.15, 3.37	3.88, 3.75

Table 4.1: Time in seconds for both algorithms

B \ Desired Accuracy	75%	80%	85%	90%	95%
160	78.9, 74.4	83.4, 79.2	87.9, 84.5	92.2, 89.4	96.4, 94.3
320	77.7, 76.5	82.4, 80.0	87.2, 84.9	91.6, 89.9	96.0, 94.7
640	78.2, 78.3	82.3, 82.1	86.7, 86.3	91.4, 91.2	95.9, 94.6
960	76.5, 80.2	81.4, 83.0	85.8, 86.4	90.7, 91.5	95.4, 95.6

Table 4.2: % Accuracy achieved for both algorithms

able amount of time.

2. A smaller beam size seems to be faster. This is not surprising because as the beam size increases, a fingerprint is compared to a larger number of other fingerprints, a lot of which aren't likely to be matches since the likeliest matches are supposed to be in the very near vicinity of the fingerprint (since that is the whole point of sorting the permuted fingerprints in the first place).

3. For a small beam size, Fixed Rounds overshoots the required level of accuracy, and by doing so, it takes a longer time than necessary. Fixed Ratio on the other hand gives nearly exactly the desired level of accuracy on average and performs fewer rounds as a result and is faster. However, Fixed Ratio also has a larger variance than Fixed

B \ Desired Accuracy	75%	80%	85%	90%	95%
160	3.39%	7.18%	1.83%	2.26%	4.79%
320	0.59%	8.64%	1.53%	2.98%	0.58%
640	12.18%	0.11%	-1.73%	0.11%	0.78%
960	-11.04%	15.81%	3.09%	-6.40%	3.57%

Table 4.3: % Time difference (positive values indicate Fixed Ratio is faster)

B \ Desired Accuracy	75%	80%	85%	90%	95%
160	1.07, 4.62	1.12, 4.29	0.92, 3.37	0.81, 2.34	0.50, 1.75
320	1.47, 4.38	1.54, 4.41	1.15, 3.62	0.01, 0.03	1.79, 3.98
640	1.68, 5.27	1.79, 3.98	1.57, 3.20	1.20, 2.40	0.65, 1.87
960	1.87, 4.31	2.09, 3.41	1.71, 3.60	1.24, 2.41	0.88, 1.26

Table 4.4: Standard deviation of accuracy measures

Rounds so even though it returns the desired accuracy on average, it is much more likely than Fixed Rounds to go well below the desired accuracy.

Why is Fixed Ratio more likely to behave correctly on average for small beam sizes? The running ratio is a little volatile as we can see from Figure 4.7. On occasion, the ratio even goes up between two rounds. When the beam size is small, there are more permutation rounds needed to achieve the desired level of accuracy, each accounting for a small step in the ratio. This fact coupled with the ratio's intrinsic volatility makes it more likely for the running ratio to dip beneath the target ratio and cause the algorithm to terminate before the expected number of rounds. For example, in Figure 4.7, if the sampling phase reached the desired accuracy at the 28th permutation round, then Fixed Ratio would actually terminate

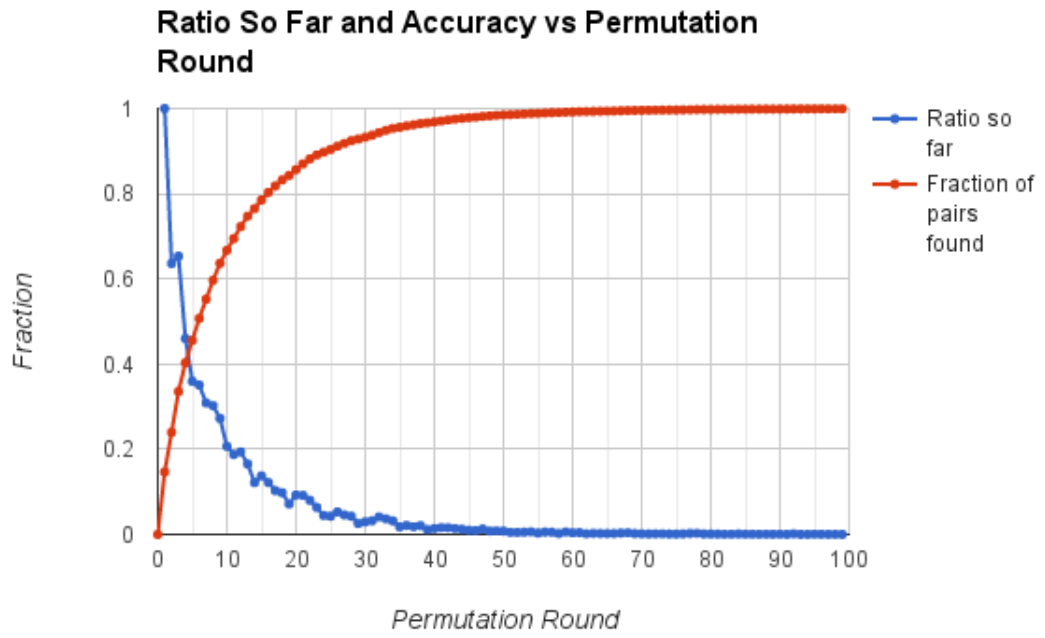

Figure 4.7: Running ratio at the end of each permutation round

at 23 permutation rounds because the ratio at that point would equal the target ratio. In this case, Fixed Ratio terminates more quickly and achieves a lower accuracy than Fixed Rounds, but since Fixed Rounds empirically tends to achieve a higher accuracy than required, Fixed Ratio will tend to return the required level of accuracy. However, this volatility of the ratio is also the reason why Fixed Ratio's achieved accuracy has a higher standard deviation than Fixed Rounds'.

Therefore, it seems like whatever the desired level of accuracy, a beam size of 160 performs best. For $B = 160$, Fixed ratio achieves the desired level of accuracy 4% faster than Fixed Rounds on average, but with an additional average standard error of 2.39%.

Note: Even though it seems like we could do even better with a smaller value of B, we have

to make sure to use a beam parameter of $B/16$ during the sampling phase, so there is not much room to decrease B.

Generality of results

We should be cautious about using the above algorithm and parameter recommendation in every situation because these results may be very specific to the particular data set we were looking at. Therefore, we performed the same tests with different corpora that were distributed differently.

We tried 10 different corpora of N text documents each that we pulled from Stack Overflow, we obtained the same general results and tradeoffs, even for different values of h.

However, this might be a staple of Stack Overflow corpora where there might naturally be the same amount of clusters distributed in generally the same way across the entire website, which would lead to the same empirical results. However, to test this theory, we use a measure of density that is indicative of the distribution of fingerprints in a set, and compare this measure across the different corpora we tested. The measure is the following ratio, which we call the density ratio:

$$\rho = \frac{\text{Number of matches in the dataset}}{\text{Expected number of matches if the fingerprints were drawn from a uniform distribution}}$$

First, we have to compute the denominator:

If we have N fingerprints that are all drawn from a uniform distribution between 0 and $2^{64}-1$, we first find the probability p that for a given fingerprint x, there is another fingerprint within Hamming distance h of it.

There are $\binom{64}{i}$ ways for a fingerprint to be at a distance of exactly i bits from x, we simply

choose i of x's 64 bits to flip. Therefore, there are $\sum_{i=0}^{h} \binom{64}{i}$ ways that a fingerprint can be within distance h of x. Since there are 2^{64} total possible values that a fingerprint can take,

$$p = \frac{\sum_{i=0}^{h} \binom{64}{i}}{2^{64}}$$

Given a pair of fingerprints, we can treat the occurrence of a match between them as a Bernoulli random variable X with probability p. Since X is distributed Bernoulli, we know that $\mathbb{E}(X) = p$. Therefore, by the linearity of expectation, we know that since there are $\binom{N}{2}$ total pairs, the expected number of matches between them will be $\binom{N}{2} * p$.
Therefore, the number of expected matches within distance h in a set of N uniformly distributed 64 bit fingerprints is:

$$D = \binom{N}{2} \frac{\sum_{i=0}^{h} \binom{64}{i}}{2^{64}}$$

In particular, for $N = 2^{16}$ and $h = 16$, we get $D \approx 83,000$ matches.

In our experiment, the first corpus had 923,000 matches, so its density ratio was

$$\rho = \frac{923000}{83000} \approx 11$$

The other corpora had density ratios ranging between 8 and 13, but as we said earlier, the empirical results were the same. Interestingly, if we assume the 923,000 are distributed evenly among the fingerprints, each fingerprint would have $\frac{923000}{2^{16}} \approx 14$, which is extremely close to $\log N$ and confirms why DBSCAN expects each regionQuery to return on the order of $\log N$ neighborPts on average [14].
However, density ratio does not tell the whole story, since a corpus can have many docu-

ments that are extremely similar in one large cluster, while the rest of the documents could conceivably be uniformly distributed. We could not find such a corpus, so we generated a few contrived ones using C++'s arc4random function:
We created a very densely packed subset of the N fingerprints by modding uniformly random fingerprints by 2^{16}, which ensured that for all of these fingerprints, they had the same first 48 bits and could only differ in the next 16. For the rest of the fingerprints, we generated them uniformly randomly between 0 and $2^{64} - 1$.
After running the same experiments, we observed the following 3 results:

1. The smallest B of 160 still performs best for both algorithms and any desired accuracy.

2. Fixed Rounds performs exactly the same, it returns slightly higher accuracy than required, for any desired level of accuracy, in the same amount of time.

3. Fixed Ratio achieves slightly less than desired accuracy on average but terminates faster. However, the drop in accuracy is slightly larger on average than for the real corpora we experimented with, and the standard deviation of the accuracy is significantly larger (up to 10 and 15% at its highest).

4. The larger the tightly packed subset, the lower the accuracy of Fixed Ratio and the higher its standard deviation.

Based on this analysis, we can say with some degree of confidence that Fixed Rounds is always safe to use, and should be the go-to algorithm in cases where the underlying distribution of the data is unknown or suspected to be greatly skewed. However, it is important to note that in these contrived examples, we have forced thousands of fingerprints to have thousands of matches each, instead of the expected $\log N$ matches for each fingerprints in real world

corpora, so we might not have much reason to worry about Fixed Ratio failing in practice after all.

4.2.3 Performance of simplified DBSCAN

Having obtained an adjacency list from the approximate nearest neighbors procedure, we run DBSCAN using the adjacency list as a graph, with each call to regionQuery immediately returning a list of NeighborPts. We experiment with different values of MinPts and use MinPts=100 in the analysis below. This modified DBSCAN only takes around 0.10 seconds to complete on average.

In order to assess the accuracy of the clustering that DBSCAN returns, we must compare it to the clustering returned by running DBSCAN on a precomputed $N \times N$ matrix of distances, which we can compute using the basic quadratic approach. In order to compare the two clusterings, we can use the Adjusted Rand Index (ARI) measure:
The Rand Index is a measure of similarity between two clusterings. It is basically the ratio of the number of pairs on which the two clusterings agree, over the total number of pairs. Two clusterings agree on a pair of fingerprints either if both fingerprints are assigned the same label in both clusterings, or if they are assigned different labels in both clusterings [26]. The Rand Index for two clusterings is a value between 0 and 1, where 0 means the two clusterings agree on no pairs at all, and 1 means they agree on all pairs.
We use a slightly modified version of the Rand Index called the Adjusted Rand Index (ARI). ARI is corrected for chance so that an ARI of 0 indicates that the clusterings are completely random with respect to each other, and an ARI of 1 indicates that the two clusterings agree on all pairs.

In order to test the accuracy of our entire clustering procedure, we first calculate the distances between all pairs in a $N \times N$ matrix, and feed it into DBSCAN to return a clustering that we regard as the gold standard. Then, we run the basic permutation algorithm with $B = 160$, and at the end of each round, we feed the adjacency list we obtain into DBSCAN and obtain a clustering. We compute the ARI between that clustering and the gold standard. Here is a graph of the ARI by round:

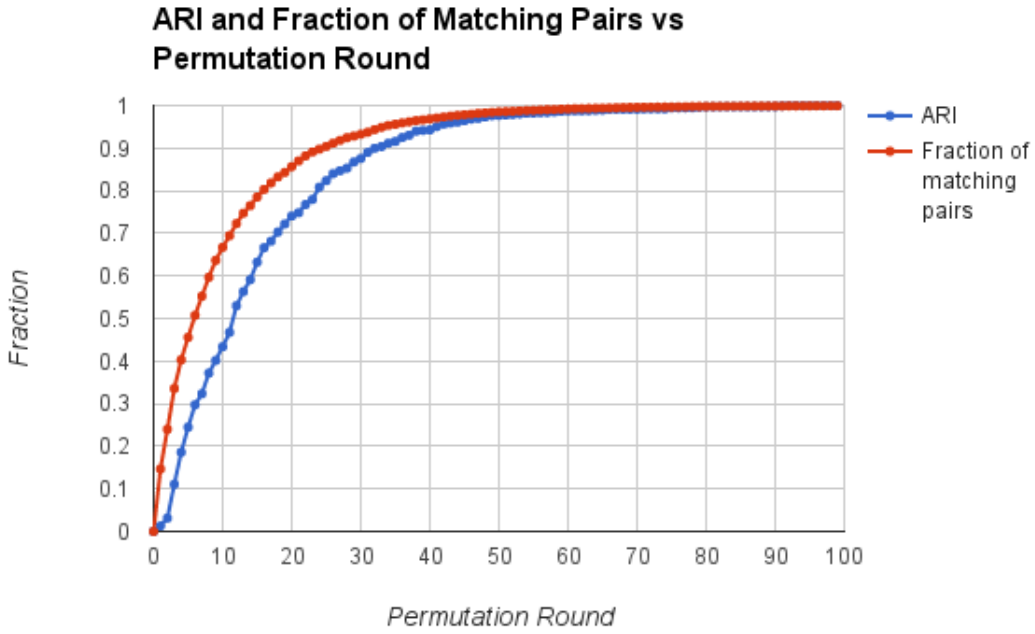

Figure 4.8: ARI and fraction of matching pairs found at the end of each round

We immediately notice that the two lines mirror each other, with ARI starting off close to 0, but gradually catching up to the line representing the fraction of matches found. For example, when the fraction of matches found reaches 90%, ARI is already at 85%. This confirms our intuition that for a highly accurate adjacency list, DBSCAN will perform well because in order for two similar documents i and j to not be assigned the same cluster, we not only have to miss out on comparing them, but we also have to miss out on comparing

at least one pair on every path from i to j. Therefore, the fraction of pairs found (especially at high accuracy levels), is a good proxy for the ARI of the corresponding clustering.

4.2.4 Aggregate performance of real-time clustering algorithm

On a dataset of $N = 2^{16} = 65,536$ tf-idf vectors, our algorithm runs a parallelized implementation of the Simhash algorithm and returns 65,536 fingerprints in 0.35 seconds instead of 2 seconds.

For a desired accuracy of 90%, both Fixed Rounds and Fixed Ratio compute an adjacency list of matching fingerprints in around 2.5 seconds, instead of the algorithm in section 2.4 which achieves the same accuracy in 19 seconds (even after we optimally choose its parameters and add parallelism to it), and the basic quadratic algorithm which returns an exhaustive adjacency list in 31 seconds.

From the adjacency list, the simplified DBSCAN produces an actual clustering of all fingerprints in 0.10 seconds and achieves an accuracy rate of 85% relative to the optimal exhaustive clustering assignment.

Therefore, the entire procedure is done in less than 3 seconds, which is still reasonably within the bounds of real-time requirements, but could even be sped up on a more performant machine. In fact, as we mentioned earlier, the Simhash reduction could also be performed offline and stored in a database, which cuts the online portion of the algorithm down to around 2.6 seconds.

In summary, here is an example screenshot of three Stack Overflow question pages that were assigned the same cluster label:

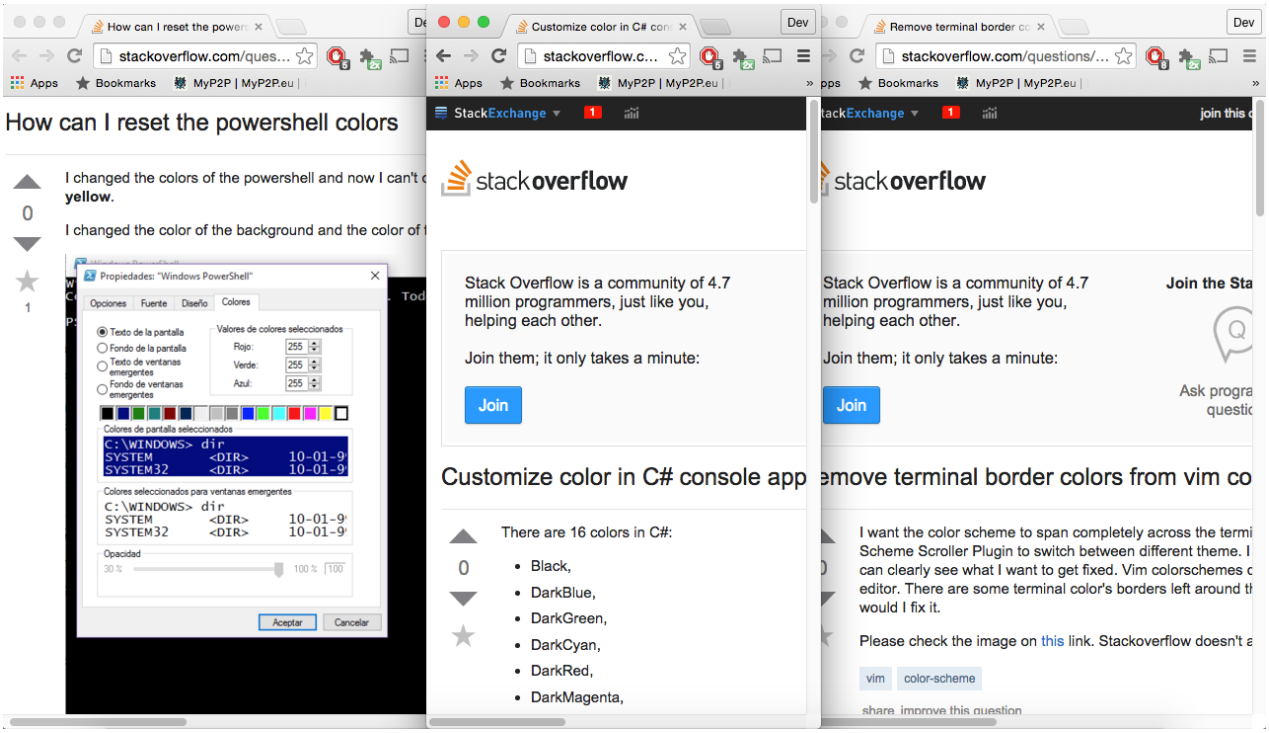

Figure 4.9: Example of three questions assigned to the same cluster

4.2.5 Performance of parallelized computation of tf vectors

We attempt to parallelize the computation of term frequencies by running multiple threads in parallel. At any moment, each thread will be assigned a list of stemmed words corresponding to a Stack Overflow question, and it will have to count the number of occurrences of each stem and increment the appropriate index counter in the corresponding tf vector. We observe the following performance depending on the number of threads run in parallel:

As expected, our best performance was achieved with 8 threads, 0.79 seconds, which is a 2.6x speed increase over the serial version. However, the speedup does not increase linearly with the number of threads, and that is probably because of scheduling overhead [24], which is especially noticeable for such a relatively small computation time.

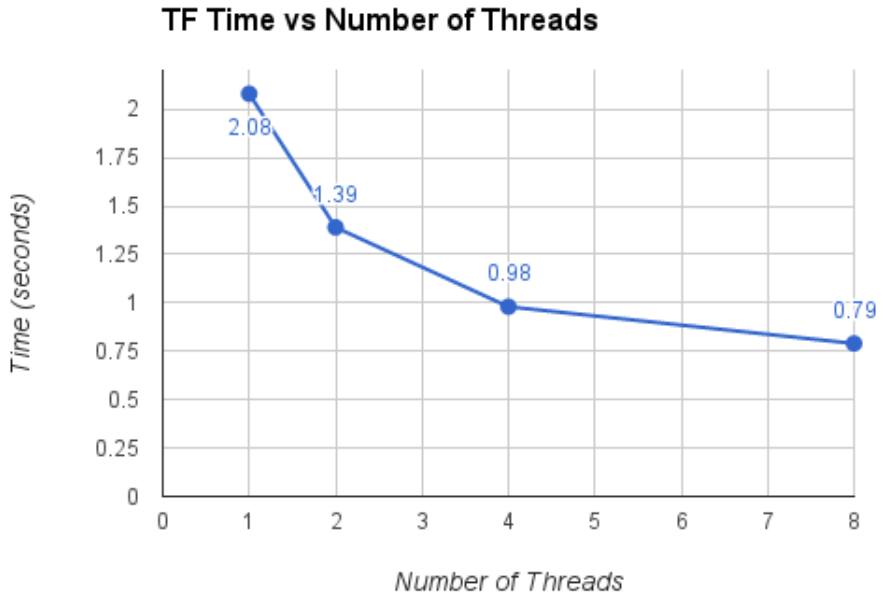

Figure 4.10: Time for tf computation as a function of number of threads

4.2.6 Performance of parallelized computation of idf vector

We would like to parallelize the computation of the global inverse document frequency vector, but we must avoid thread contention since in this situation, threads are all writing to the same vector. Keeping the number of mutex locks fixed at 20000 (approximately one lock for every 3 words), and varying the number of threads, we obtain the following results:

Again, we notice a significant speedup due to parallelism. The parallelism speedup is not linear with the number of threads though, again probably due to scheduling overhead, but also due to increased thread contention, i.e. the fact that more threads are now more likely to wait on each other for locks.

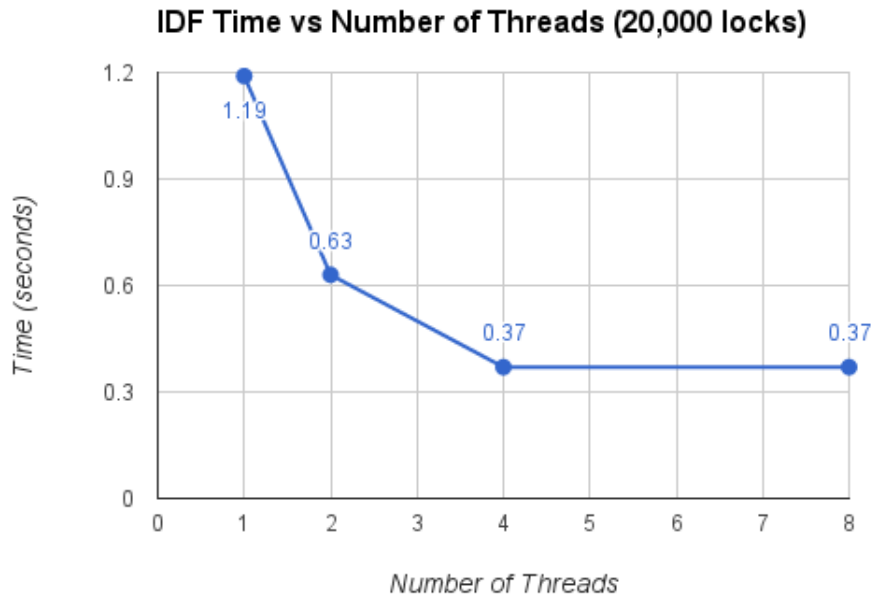

Figure 4.11: Time for idf computation as a function of number of threads

We would like to experiment with different locking granularities to find a balance between minimizing thread contention on the one hand and minimizing locking overhead on the other. We keep the number of threads constant at 8 and vary the granularity of our locks, we observe the following performance:

As the number of locks increases from 1 to 10,000, we notice a steady increase in performance. This is due to the fact that the more locks we have, the less likely it is that threads will be idly waiting on each other. However, after 20,000 locks, we start to notice a dip in performance. This is because of locking overhead. Our best result came with 8 threads and 20,000 locks: 0.37 seconds, instead of the serial 0.65 seconds.

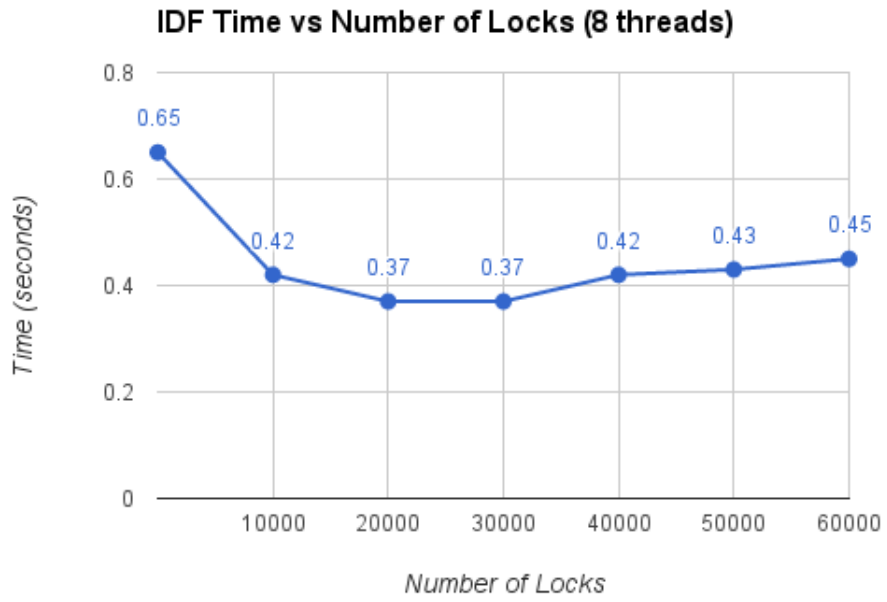

Figure 4.12: Time for inverse document frequency computation as a function of number of locks

4.2.7 Performance of parallelized multiplication of tfs by idf

The multiplication of tf and idf is a great candidate for SIMD instructions: Instead of performing multiplications of pairs of floats one at a time, we multiply each group of 8 32-bit floats of a tf vector by the corresponding group of 8 32-bit floats of the idf vector. Here is our performance as a function of the number of threads with and without SIMD instructions:

We notice that multithreading has the most pronounced effect when moving from 1 thread to two threads, after which scheduling overhead begins to outweigh its benefits. We also notice that SIMD instructions improve performance slightly for one thread, but that improvement is amplified as the number of threads increases. Still, SIMD instructions only give a speedup

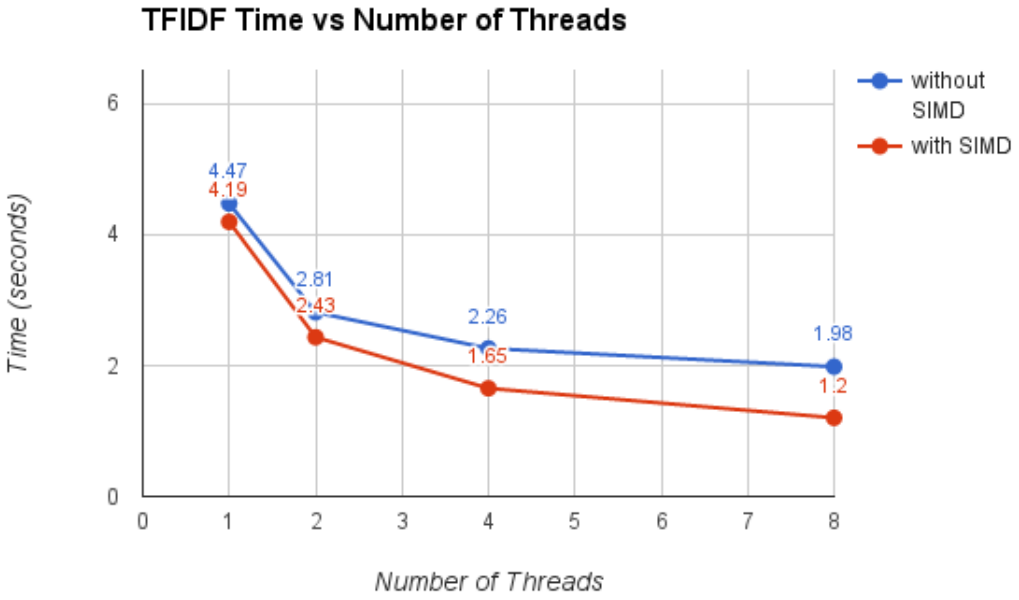

Figure 4.13: Comparison of time for tf x idf multiplication as a function of number of threads, with and without SIMD instructions

of around 2x for 8 threads. Most likely, this is partly because of the cost of loading the data into blocks of 8 32-bit floats that go into a SIMD register. If we performed more than one mathematical operation after having loaded the data into registers, we might observe a greater speedup.

4.2.8 Caching benefits of a small chunksize

We compare the performance of multithreading with a chunksize of 1, (i.e. threads are interleaved with only one element between consecutive threads), to multithreading with a standard chunksize of $c = \frac{\text{num elements}}{\text{num threads}}$, (i.e. each thread runs over a block of c consecutive

elements). Here are the results:

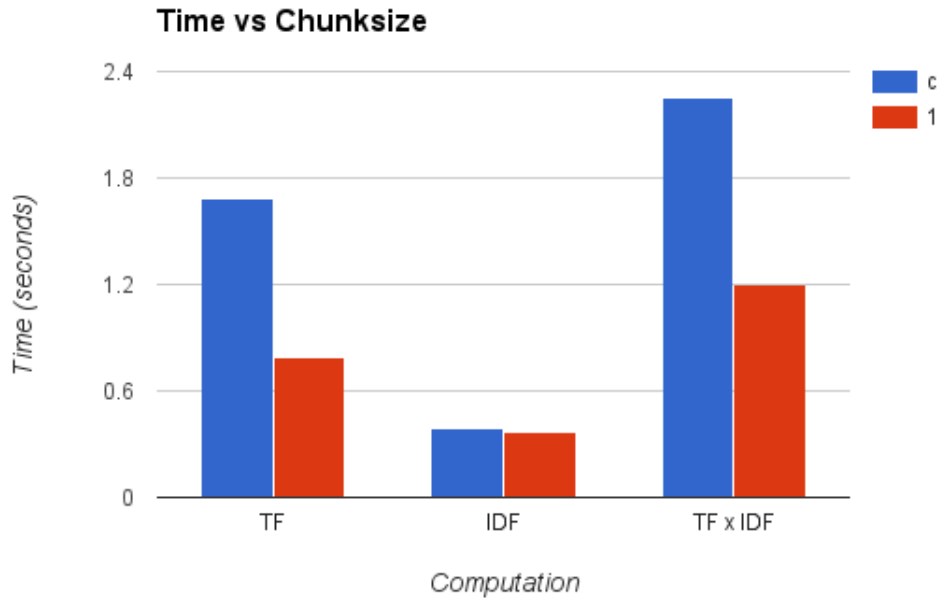

Figure 4.14: Comparison of the impact of chunksize on the time for tf, idf, and tf x idf computations

As expected, we get caching benefits and reduce I/O latency. However, the improvement is negligible for the idf computation. The is because the entire idf vector can fit easily in the L3 cache, whereas for the tf and tf x idf computations, the list of tf vectors does not entirely fit in the L3 cache, meaning that threads will overwrite each other's contents in the L3 cache and will have to load them again. However, the entire idf vector cannot fit in either the L2 or L1 cache, so we would expect to see I/O latency reduction based on caching benefits in those two levels. Therefore, the fact that there were no significant gains in the idf computation confirms that the major latency arises from moving memory to the L3 cache, not from L3 to L2 and L1. This realization supports the fact that the cache-conscious block computation of permutation lists did not yield significant gains in performance.

From all the aforementioned experiments, we conclude that it is possible to perform the entire tf-idf computation for $N = 2^{16}$ documents (each represented as a list of stemmed words) in around 2.4 seconds, by using cache-conscious multi-threading, smart locking, and SIMD instructions. This corresponds to a speedup of 3.3x.

Chapter 5

Conclusion and Future Work

In this thesis, we have focused on the problem of clustering a corpus of tens of thousands of text documents in real time. This thesis contains a major and a minor contribution.

The major contribution is an efficient algorithm that is capable of producing an approximately exact clustering of tens of thousands of tf-idf vectors in real-time. The algorithm consists of three successive procedures. First, a reduction from high-dimensional tf-idf vectors to 64-bit Simhash fingerprints using a parallelized and efficient LSH technique called Simhash. Second, two alternative procedures that return a desired fraction of all pairs of fingerprints within a desired distance h of each other. These procedures use a sampling technique to determine a target number of rounds or a target ratio, then run a series of permutation rounds which involve sorting lists of fingerprints that have been permuted using a uniformly random permutation function, until the target number of rounds or target ratio have been met. The third procedure is a simplified version of the DBSCAN algorithm which

when given as input an adjacency list produced by the second procedure, efficiently returns a clustering without having to perform any distance comparisons. This entire algorithm is implemented and shown to produce a clustering of 65,536 high-dimensional tf-idf vectors accurate to within 85%, in under 3 seconds instead of 31 seconds.
The minor contribution consists of a series of optimizations that speed up the process of creating a tf-idf vector for each text document in the corpus by relying on multithreading, efficient use of mutex locks, vectorized SIMD instructions, and cache conscious algorithms. The first contribution describes a system that takes as input 65,536 lists of stemmed and tokenized pages of questions pulled from Stack Overflow's API, and produces 65,536 tf-idf vectors of 58,000 entries each in approximately 2.4 seconds instead of 8 seconds.

The major contribution of this thesis builds on existing literature that primarily discusses efficient algorithms involving dimensionality reduction and bit-permutation procedures for near-duplicate detection in extremely large corpora. The algorithm for the main contribution described in this thesis attempts to behave efficiently on a given dataset with minimal input from the programmer, as it uses sampling methods in an attempt to make more informed decisions about hyper-parameters that it uses in later steps. Nevertheless, the algorithm does present the programmer with a minor tradeoff between average speed on the one hand, and standard deviation of achieved accuracy on the other. However, if the programmer has no preference or has concerns about a potentially unusual distribution of their data, this thesis recommends an algorithm that has empirically been shown to perform extremely well on datasets with a wide range of differently distributed elements.

This thesis shows that there is still vast room for advancement in the field of approximate

nearest neighbor computations. Indeed, an immediate area of future research would be to create a mathematical model with theoretical guarantees regarding the performance of each of the two alternative algorithms outlined in this thesis, "Fixed Rounds" and "Fixed Ratio", depending on the distribution of the dataset. One possible way of determining this distribution in real-time might be to take a representative sample, compute a quadratic matrix of distances and cluster it using traditional DBSCAN in order to get a sense of the number of clusters in the dataset and the size of each. After we obtain mathematically quantifiable knowledge about the distribution, the mathematical model may be able to pick the appropriate stopping condition. Another area of future research could be a more flexible sampling technique. In this thesis, we have opted for uniform sampling, but research shows that in datasets with skewed cluster distributions, biased sampling produces more representative samples [27]. When data are suspected to have skewed clusters, a possible algorithm might be to try out different sampling techniques, and compare the results of the quadratic algorithm on each sample to reach a conclusion about the more likely data distribution and the better suited hyper-parameters. Another potential area of research would be to use a new and exciting variation on DBSCAN called DBSCAN-M [28] that is based on mutual reinforcement. This algorithm may be able to close the (already small) gap observed between the ARI and the fraction of matching pairs found in Figure 4.8.

Bibliography

[1] K. Tran, S. Oh. UWSNs: A Round-Based Clustering Scheme for Data Redundancy Resolve. *International Journal of Distributed Sensor Networks*, 2014.

[2] B. Patra, S. Nandi. Effective data summarization for hierarchical clustering in large datasets. *Knowledge and Information Systems*, 42(1): 1-20, January 2015.

[3] X. Zhao, W. Lin, J. Hao, X. Zuo, J. Yuan. Clustering and pattern search for enhancing particle swarm optimization with Euclidean spatial neighborhood search. *Neurocomputing* 171: 966-981, 2016.

[4] R.K. Roul, O.R. Devanand, and S.K. Sahay. Web document clustering and ranking using Tf-Idf based apriori approach. *IJCA Proceedings on ICACEA*, 2: 74-78, June 2014.

[5] J. Ramos. Using TF-IDF to determine word relevance in document queries. *Proceedings of the First Instructional Conference on Machine Learning*, 1-4, 2003.

[6] Target Collision Resistant Hash Function, *Encyclopedia of Cryptography and Security*, 2nd Edition, 1279.

[7] P. Indyk and F. Motwani. Approximate nearest neighbors: towards removing the curse of dimensionality. *Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing*, 604-613, 1998.

[8] M. Charikar. Similarity estimation techniques from rounding algorithms. *Proceedings of the Thirty-Fourth Annual ACM Symposium on Theory of Computing*, 380-388, May 2002.

[9] S. Sood and D. Loguinov. Probabilistic near-duplicate detection using simhash. *Proceedings of the Twentieth ACM International Conference on Information and Knowledge Management*, 1117-1126, 2011.

[10] G. Singh Manku, A. Jain, and A. Das Sarma. Detecting near-duplicates for web crawling. *Proceedings of the 16th International Conference on World Wide Web*, 141-149, 2007.

[11] D. Ravichandran, P. Pantel, and E. Hovy. Randomized algorithms and NLP: using locality sensitive hash functions for high speed noun clustering. *Association for Computational Linguistics*, 1-8, 2005.

[12] W. Choi, S. Oh. Fast Nearest Neighbor Search using Approximate Cached k-d tree, *International Conference on Intelligent Robots and Systems*, 4524-4529, 2012.

[13] J.E. Goodman, J. ORourke, and P. Indyk. Chapter 39: Nearest neighbours in high-dimensional spaces. *Handbook of Discrete and Computational Geometry*, 877-892, 2004.

[14] M. Ester, Hans-Peter Kriegel, J. Sander, and X. Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. *KDD-96 Proceedings*, 226-231, 1996.

[15] S. Savage. Lecture 2: Constructing k-wise independent variables. *University of California San Diego CSE 123*, 1-4, 2015.

[16] R.A. Fisher and F. Yates. Statistical tables for biological, agricultural and medical research. *American Journal of Veterinary Research*, 3: 26-27, 1963.

[17] Java.lang.Integer. *Free Software Foundation*, 2005.

[18] P.C. Diniz and M.C. Rinard. Lock coarsening: eliminating lock overhead in automatically parallelized object-based programs. *Journal of Parallel and Distributed Computing*, 49(2):1-29, March 1998.

[19] G. Conte, S. Tommesani, and F. Zanichelli. The long and winding road to high-performance image processing with MMX/SSE. *IEE Computer Society*, 303-310, 2000.

[20] Y. Zhang, D. Lo, X. Xia, J. Sun. Multi-Factor Duplicate Question Detection in Stack Overflow. *Journal of Computer Science and Technology*, 981-997, 2015.

[21] Natural language toolkit. *NLTK Project*, March 2016.
http://www.nltk.org/

[22] M. Kantrowitz, B. Mohit, V. Mittal. Stemming and its effects on TFIDF ranking, *SIGIR*, 357-359, 2000.

[23] S. Behnel, R. Bradshaw, L. Dalcn, M. Florisson, V. Makarov, and D.S. Seljebotn. Cython C-Extensions for Python. *Cython*, March 2016.

[24] M. Holenderski, R. Bril, J. Lukkien. Parallel-Task Scheduling on Multiple Resources. *ECRTS* 233-244, 2012

[25] S. Saini, H. Jin, R. Hood, D. Barker, P. Mehrotra, R. Biswas. The impact of hyper-threading on processor resource utilization in production applications. *High Performance Computing*, 1-10, 2011.

[26] W.M. Rand. Objective criteria for the evaluation of clustering methods. *Journal of the American Statistical Association*, 66(336): 846-850, 1971.

[27] A. Appel, A. Paterlini, E. de Sousa, A. Traina, C. Traina Jr. A Density-Biased Sampling Technique to Improve Cluster Representativeness. *Knowledge Discovery in Databases*, 366-373, 2007.

[28] Y. Li, C. Guo, R. Shi, X. Liu, Y. Mei. DBSCAN-M: An Intelligent Clustering Algorithm Based on Mutual Reinforcement, *International Conference on Algorithms and Architectures for Parallel Processing*, 66-77, December 2015.