Evaluating Near-Duplicate Detection in Twitter

Damilare D. Fagbemi

7th September 2014

A Thesis submitted in part fulfilment of the

MSc in Advanced Software Engineering

Supervised by: Professor Mark Keane

UCD School of Computer Science and Informatics

College of Science

University College Dublin

To gain your own voice, you have to forget about having it heard.

- Allen Ginsberg, WD

Abstract

Twitter has become a very popular micro blogging tool used for the expression of views and to broadcast news or events. As people post to Twitter in real time, millions of microblogs (tweets) are generated every second for major events. Near-duplicate detection in Twitter is of increasing importance due to the primary role it plays in first story detection, spam detection, and many other clustering processes. This thesis evaluates how different combinations of similarity measurement techniques, clustering strategies, and tweet representations affect the speed and accuracy of the near-duplicate clustering process in Twitter. It achieves this by building a clustering system which generates and validates near-duplicate clusters using different techniques and algorithms. The objective is to deduce what techniques or algorithms are necessary to achieve optimal speed in the clustering process while producing accurate results. The design of the system and the algorithms it comprises are described, as well as the design of experiments based on the clustering system. Also, the results of the experiments are analysed and certain deductions are made.

I not only use all the brains that I have, but all that I can borrow.

- Woodrow Wilson

Acknowledgements

I want to express my heartfelt thanks to my supervisor, Professor Mark Keane, who has been a source of inspiration to me personally and professionally. Prof. Mark has been the best supervisor I could hope for. His skill in identifying interesting research problems and efficiently organizing the research process ensured that my work proceeded seamlessly from start to finish. His open and down to earth approach also meant that the entire process was not just interesting, but loads of fun. Working with Prof. Mark has taught me how research can be interesting, effective, and enjoyable.

I am very grateful to the Director of the Advanced Software Engineering program, Dr. M. O Cinneide, who has been a constant source of support and advice throughout the entire masters program. His coordination of the masters program has been stellar. A good example is how efficiently students were introduced and mapped to potential thesis supervisors in various research areas. None of my fellow masters students outside the Advanced Software Engineering program had any similar experience and they usually went green with envy whenever I described the process. Once, I was actually accused of boasting.

Thanks also to Fengpan Zhao and Aisling Connoly of the Text Analytics research group at the Insight Centre for Data Analytics who helped me to understand and make use of the Twitter data available at Insight.

Finally, I would like to dedicate this thesis to my parents, who have made many sacrifices to allow me to be where I am today and without whom my education would not be possible.

Contents

List of Figures

Oh that my words were now written! Oh that they were printed in a book! That they were graven with an iron pen and lead in the rock forever!

- Job 19

1 Introduction

1.1 From Cuneiform Tablets to the Gutenberg Bible

A defining difference between humans and other species is our ability to weave complex thought patterns and express them efficiently through language. This ability in language and communication is a cornerstone of our civilization and evolution. The first forms of communication were verbal and while those were certainly useful, they were limited in reach and longevity. A major leap in human evolution was development of writing systems that allowed us to store communication in physical forms. Not only did text make it possible for us to communicate accurately with persons thousands of miles away, we could also archive text and communicate with generations yet to be born. One of the earliest examples of this are Cuneiform tablets dated to around 4th millenium BC which are still being studied today.

The earliest forms of writing were done by hand and as you can imagine it was a painstakingly long process which meant there weren't many actual books in circulation and those that existed were expensive. The next major evolution was the development of the movable type which was a machine that used movable components to reproduce elements of a document. The first of these was dated to around 1040AD and was built in China. Around 1450, Johannes Gutenberg built an improved movable type in Europe and the limited number of characters required for European language was an important factor in its success. This lead to first major book printed in Europe which is the Gutenberg bible in 1455. The high quality and relatively low price of the Gutenberg Bible (1455) established the superiority of movable type in Europe and the use of printing presses spread rapidly[1].

The printing press may be regarded as one of the key factors fostering the Renaissance and due to its effectiveness, its use spread around the globe. The availability of paper and the invention of metal movable type sped the dissemination of ideas from the later 15th century.

1.2 Children of the Web and Twitter

Sometime in the 20th century the world went soft. With the evolution of computing and the personal computer, focus shifted to software, electronic media, and electronic text/books as the primary means of communication. The arrival of the Internet in the same century took things some steps further. With the creation of an international network of computers, a Global Village has evolved which allows us to communicate seamlessly and efficiently with friends, family and associates across the world.

The Internet and the world wide web has had a major impact on the evolution of civilization across the world as information and ideas are transmitted at the speed of light. Although this impact is usually taken for granted by the children of the web and we may forgive them. You see, I believe there are two generations of humans today, the Children of the Web who were born during or after the 1980s and can't imagine a world without computers and the Internet, and all others, the Ancients. To their credit, it must be said that the Ancients are keeping up quite well.

If the 20th century and the emergence of the Internet was a wild ride, the 21st century so far can be described as a roller coaster that only few could have envisaged. The arrival of novel social media platforms such as Twitter, and the dynamism in the evolution of their use and application has seen the world connected in a way that never existed previously.

Twitter allows users to share information about themselves, their activities, and what is happening around them. Users are able to share at most 140 characters. The limitation on the amount of information that can be shared at any one time helped to entrench twitter as a service used to share little bits of information on events as they happen. As a result, text is constantly been shared by several users every second. This leads to a situation where news events are able to break

on twitter, but there is also a challenge of identifying those events quickly due to the share volume of twitter data constantly.

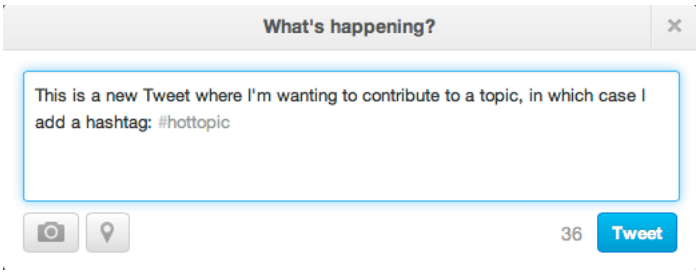

Figure 1.1: Twitter web form for composing a tweet

1.3 Near-Duplicate Identification in Twitter

Twitter has become a very popular micro blogging tool used for the expression of views and to broadcast news or events. As people post to Twitter in real time, millions of microblogs (tweets) are generated every second for major events. Twitter is not only interesting because of this real time response, but also because it is sometimes ahead of traditional news media. For example, the Boston Marathon bombing on April 15, 2013 was first shared on Twitter.

Over 500 million tweets are posted daily in twitter. From the comical, to the bizarre, to serious news, they all pass through twitter. Soon people might tweet when they breathe, because the air on that road is so fresh that the whole world must know about it. That is fine and we are more than happy to encourage multiple tweets since the more tweets are sent, the more data we have to analyse! Our analysis causes us to provide solutions to multiple problems that arise due to the sheer volume of data to be processed.

Near-duplicate detection is essential to many clustering processes as it helps with the grouping of items or elements. For instance, first-story detection deals with the detection of new events on web-scale corpora. In twitter this involves clustering tweets into separate stories or events. Near-duplicate detection is closely associated with that process as it helps with the identification of similar

tweets. The detection and elimination of spam is also heavily reliant on near-duplicate detection. We will focus on near-duplicate detection and some of the techniques and processes that can be used to achieve it.

We investigate the problem of near-duplicate detection in twitter by building a framework which uses and combines several text similarity measurement techniques with different clustering strategies and tweet text representations.

1.4 Research Objectives and Approach

Our aim is to compare the accuracy and efficiency of combinations of text similarity measurement algorithms, clustering strategies and text representations.

My thesis statement is: A framework for near-duplicate text clustering and analysis of generated clusters. In defending this claim my thesis makes the following contributions:

- A software system that generates near-duplicate clusters from tweets stored in a database.
- A system that allows users to make a selection from a range of text similarity measurement techniques, clustering strategies, and text representations. Users are also able to specify the similarity thresholds and the amount of tweets to be clustered.
- Storage of clusters, cluster analytics and clustering time to text files.
- Analysis, comparison, and validation of clusters generated by different combinations of similarity measurement techniques, clustering strategies, and text representations.

1.5 Things to Come

The rest of this dissertation explains the details of the near-duplicate clustering system. A survey of related work in near-duplicate detection is presented in Section 2. A detailed description of the the system, its design as well the algorithms and methodologies used is given Section 3. We bore into the gritty details of the

setup and composition of our experiments in Section 4. Our results are presented and analysed in Section 5. Finally, in Section 6 we summarize the work done, what worked and what didn't then discuss possible future work.

If you don't know history, then you don't know anything. You are a leaf that doesn't know it is part of a tree.

- Michael Crichton

2 Related Work

In this Section, we discuss prior work in near-duplicate detection and analysis of social media as well as how it relates to our research and our system.

2.1 Detecting Near-Duplicates in Text

Our research is concerned with the detection of near-duplicates in tweets. A techniques that has been used previously for near-duplicate similarity search is the nearest neighbour algorithm in which the goal is to find the nearest neighbors of a given point query[2]. The nearest neighbour algorithm is one of the approximation methods that aim to reduce the dimensionality and/or size of the input vectors.

One of the first approaches to solving the approximate-nearest neighbour problem in sublinear time made use of a new method called Multi-Index Locality Sensitive Hashing (LSH) [4]. This method relied on hashing each query point into buckets in such a way that the probability of collision was much higher for points that are nearby. When a new point arrived, it would be hashed into a bucket and the points that were in the same bucket were inspected and the nearest one returned. LSH reduces the candidate pairs for similarity search by ensuring that buckets contain items that are likely to be similar.

LSH has been used for twitter research in a streaming setting [7]. In that work, a modified version of LSH was implemented that addressed certain suggested variance in results provided by applying traditional LSH for near neighbour search. They also introduced a new strategy such that when the LSH scheme declares a document to be new (i.e., sufficiently different from all others), a search is started through the inverted index, but only compares the query with a fixed number of most recent documents.

The near-duplicate similarity search problem has also been directly addressed in the context of identifying near-duplicate web pages[6]. In this work, the problem is solved approximately by applying a sketching function based on min-wise independent permutations in order to compress document vectors whose dimensions correspond to distinct word n-grams (called shingles).

Other researchers have investigated an approach to similarity search which does not rely heavily on approximation techniques [5]. In this case, their work was focused on the problem of finding all pairs of vectors whose similarity score (as determined by a function such as cosine distance) is above a given threshold. They utilized an inverted list based approach to the problem to avoid the need to build a complete inverted index over the vector input. They also ordered the vectors to reduce the search space.

There has been recent work on near-duplicates in Twitter [Ke tao et. al]. Their research focused on the measurement of similarity between tweets by examining syntactic, semantic and contextual information of the tweets. This involved the use of syntactic similarity measurement techniques like Leveshtein distance and Jaccards coefficient for measuring overlaps in words, hashtags and URLs. Semantic similarity was measured by monitoring overlaps in entities or entity types using Named Entity Recognition (NER) services. NER services are provided by specific software systems that locate and classify elements in text into pre-defined categories such as the names of persons, organizations, locations, expressions of times, quantities, monetary values, percentages, etc. Their research also monitored overlaps in WordNet concepts using the lexical database, WordNet. WordNet is a lexical database for the English Language. It groups English words into sets of synonyms, provides definitions and usage examples, and records a number of relations among these synonym sets or their members. WordNet can be seen to provide the functionality of a dictionary and a thesaurus[22]. Finally, contextual similarity was evaluated by measuring temporal difference of tweets, similarity between users and checking the clients used to send tweets. As explained in the next section, our work went a step further than investigating similarity between two tweets as we explored the methods of clustering and nearest neighbour generation to hasten this process.

Our work utilized LSH as a candidate approach for similarity search in a static setting where tweets are stored in a relational database. Our approach combined

our modified version of an existing LSH algorithm [4] with various representation of tweets. The similarity search algorithms and experiments carried out were based on in-memory scenarios.

2.2 Analysis of Social Media

In recent years, analysis of social media platforms and the usage of those platforms has attracted a lot attention from the research community. Earlier research on the Twitter platform was focused on the intrinsic properties of Twitter [9, 10].

Our work utilized LSH as a means of nearest neighbour search for duplicates in a static setting. It provides an alternative means of clustering that could be combined effectively with text similarity measurement techniques to detect near-duplicates.

Some of the recent research on Twitter employed text similarity measurement techniques like Levenshtein distance while measuring overlaps in words and twitter specific characteristics like hashtags and URLs [8]. We decided to treat tweets as more conventional text pieces. Our idea being the clustering of tweets based on pure text analytics. To this end we utilized various tweet representations where certain components aspects of the tweets were filtered away to check the effect on duplicate detection. We were also able to go a few steps above regular words by using shingles.

Some other approaches in near-duplicate detection such as SpotSigs have used alternative approaches to regular word shingles [12]. A shingle is a contiguous subsequence of tokens in text is referred to as a shingle. Given a document D we define its w-shingling S(D,w) as the set of all unique shingles of size w contained in D . So for instance the 4-shingling of (a,rose,is,a,rose,is,a,rose) is the set{ (a,rose,is,a), (rose,is,a,rose), (is,a,rose,is) } [6].The SpotSigs system utilizes shingles close to a stop word-antecedent.

Figure 2.1: SpotSigs Shingle

Shingles are interesting as they potentially convey more information and context than is obtainable using a feature based approach to data analysis. Also, shingles implicitly encode the sequence of data. In our research, some of the text representations used were based on regular 3-shingling across tweets. That is to say we did not use any specifically styled shingles as was the case with SpotSigs.

The details are not the details. They make the design.

-Charles Eames

3 System Description and Design

In this section, we describe the tools and technologies used in building our clustering system. We also present the various algorithms used.

3.1 System Overview

The clustering system was built using the Java programming language. Our choice of Java was motivated by the availability of numerous supporting libraries for text analytics. The Twitter data used by the system is stored in a MySQL relational database and it consists of tweets selected from the Insight on Ireland twitter crawl. The system is built as a lean command line tool.

The basic functionality provided by the system includes:

- Generation of near-duplicate clusters.
- Validation of clusters by selecting the most frequent clusters discovered by our experiments.
- Comparison of the near-duplicate clusters found by using different text similarity measurement techniques.
- Compilation of statistics on clusters found using experiments.

3.2 Application Parameters

Parameter_1: This dictates the technique used to compare tweets for similarity. The actual options are Levenshtein distance, Jaccards coefficient, Minhashing, and Hamming distance.

Parameter_2: The clustering strategies: Naive Pairwise (NPW) comparison, Exact Duplicate Filtering (EDF), and Multi-Index Locality Sensitive Hashing (LSH).

Parameter_3: The tweet representations to be used.

Parameter_4: The similarity threshold to be used in adjudging if two tweets are near-duplicates.

Parameter_5: The dataset size or the number of tweets to be selected from the database for clustering.

The following diagram displays the process flow of the system.

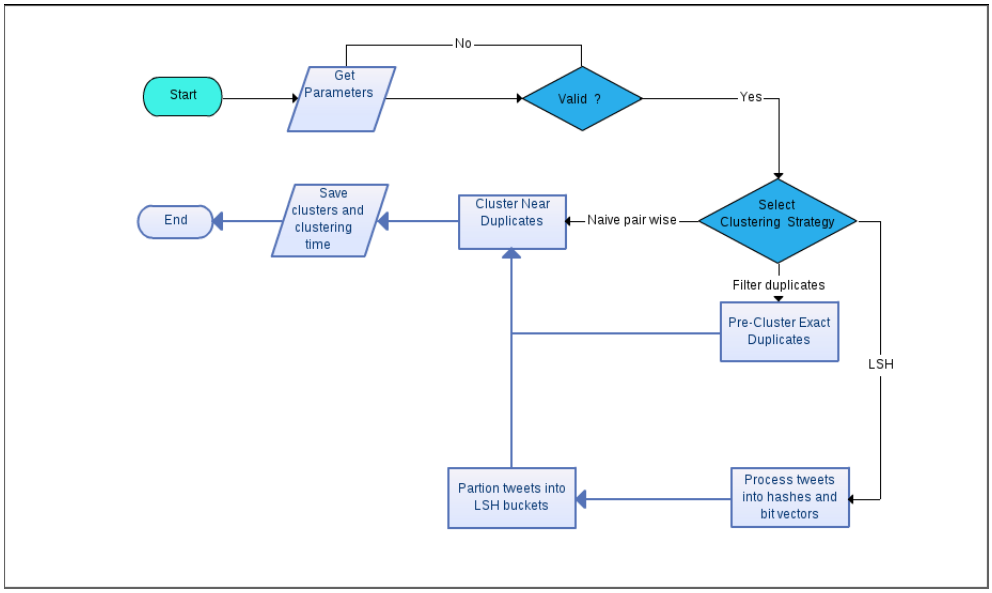

Figure 3.1: System Process Flow

3.3 Algorithms and Data Structures

Throughout the course of our research, we will be evaluating the effects of different combinations of similarity measurement techniques, clustering strategies, and tweet representations on near-duplicate detection. We wish to explore the specific effects on the speed and accuracy of the clustering process. In this section, we will be introducing the actual techniques, strategies, or algorithms used.

We will also give a brief description of the data structures we used for our experiments.

3.3.1 Similarity Measurement Techniques

In this section we explain the various text similarity measurement techniques used in our system.

Levenshtein Distance This is the least number of single-character edits (i.e. insertions, deletions or substitutions) that are necessary for changing one word into another.

For example, the Levenshtein distance between "kitten" and "sitting" is 3, since the following three edits change one into the other, and there is no way to do it with fewer than three edits:

kitten - > sitten (substitution of "s" for "k")
sitten - > sittin (substitution of "i" for "e")
sittin - > sitting (insertion of "g" at the end)

Figure 3.2: Levenshtein Example

We normalized the Levenshtein distance into a similarity metric by dividing the distance with the maximum length of the supplied strings :

$L_s = 1 - (\frac{L_d}{\text{maxlength}(S_1,S_2)}) \ldots$ where L_s = Levenstein similarity, L_d = Levenshtein distance, while S_1 and S_2 are the supplied strings. L_s is always a value between 0 and 1. Our Java implementation is shown in Appendix B.1.

Jaccards Coefficient It is a statistic that measures similarity between finite sample sets, and is defined as the size of the intersection divided by the size of the union of the sample sets. In our case this could be sets of tokens or groups of tokens.

Given two strings, S_1 = “I will go the gym”, S_2= “I will be at the gym”. Both strings could be tokenized into Sets A ={1, will, go, to, the, gym} and B = {I, will, be, at, the, gym}.

Jaccards coefficient can be used to measure the similarity of the strings by using the formula below:

$$J(A,B) = \frac{|A \cup B|}{|A \cap B|}. \text{ Clearly, } 0 \leq J(A,B) \leq 1.$$

Figure 3.3: Jaccards Coefficient

Our Java implementation is shown in Appendix B.2

Minhashing A problem that is encountered when running algorithms like Jaccards coefficient is that the scale of the value and the problem changes with the size (number of tokens) of the tweet. The Jaccards coefficient has a time complexity of $O(N^2M^2)$, where N is the number of tokens per tweet and M is the number of tweets. That is inefficient.

To address this, we can transform our tokens with a method known as minhash. The difference here is that the word or word groups are hashed into a finite set of integer values. Hence each text or tweet in our case is assigned a minhash signature consisting of N number of hashes.

With minhash signatures (and the LSH process which uses it), we have to select the approriate value of N in order to reduce the error factor created by reducing our tweets to a set of integers. N is usually selected empirically. The higher, the value of N, the smaller the error factor. However, a higher value of N could affect the speed of the program.

The formula below enables us to select N based on the error factor we are able accommodate.

$N = \frac{1}{(errorfactor \times errorfactor)}$

Hence, selecting N as 100, gives and error factor of 10%.Our tests showed us that by selecting an error factor of 7%, we were able to achieve accuracy in similarity comparison via Minhashing without losing speed due to the 200 hash values in the resulting Minhash signatures. In a subsequent section we will explain how the number of hash values can be modified to suit the size of the dataset to analysed.

The algorithms below show how the minhash signatures are generated:

Hash (i,t)

Input: a seed integer value i and a token t
Output: a hash value h, computed using hash functions like LINEAR, MURMUR128, MD5 ... here we'll use LINEAR
$h \leftarrow LINEAR(i, t)$
return h

Figure 3.4: The hash function used in the Minhash Algorithms

Minhash(T,N)

Input: a set of tokens T = {t_1,t_2,...,t_n} and N number of hash values to be generated
Output: a list M comprising of integer hash values
$M \leftarrow \emptyset$
for all $I = 1 \rightarrow N$ **do**
    $M_i \leftarrow \emptyset$
    for all $i = 1 \rightarrow n$ **do**
        $h \leftarrow Hash(i, t_i)$
        $M_i \leftarrow M_i \cup h$
    end for
    min_hash= min (M_i)
    $M \leftarrow M \cup min\_hash$
end for
return M

Figure 3.5: The Minhash Algorithm

Using minhash signatures the Jaccards coefficient can be approximated as described by this algorithm:

MinHashSimilarity (H_1,H_2,N)

```
Input: two Minhash signatures H1 and H2 of size N
Output: a similarity score s ∈ [0,1]count = 0
for all i = 1 → N do
    if H1i = H2i then
        count = count + 1
    end if
end for
s = count/N
return s
```

Figure 3.6: The Minhash similarity score algorithm

Minhashing has time complexity of $O(NM^2)$.

Our Java implementation consisted of a wrapper class built on a Minhash class written by Thomas Jungblot [23]. The wrapper class is shown in Appendix B.3.

Hamming Distance between Bit Vectors This is the number of positions at which corresponding symbols are different in strings of equal length. The Levenshtein distance which was described previously is a more sophisticated technique that is able to handle strings of different length, but it is more computationally intensive. Furthermore, in an interesting daisy-chain like sequence, we reduced the minhash signatures generated via Minhashing to smaller bit vectors of equal length which can be compared using Hamming Distance.

For example, the Hamming distance between: 1011101 and 1001001 is 2. We also normalized the hamming distance into a similarity metric thus:

$H_s = 1 - (\frac{H_d}{N})\ldots$ where H_s = Hamming similarity, H_d = Hamming distance, and N is the length of both input strings.

Our implementation of this similarity measurement technique was done in the LSH class which is shown in Appendix C.2.

In a subsequent section on LSH, we will explain how bit vectors are created, why, and how they are used.

3.3.2 Tweet Representation

In near-duplicate detection, the appropriate choice of text representation is often dependent on the context. For instance, near-duplicate detection in documents usually involves aggressive filtering, stemming, or stopping of text. However, using the same methods with tweets might be detrimental to the clustering process since the tweets are already so small in size. Hence, it is important for us to investigate the use of different representations of tweets in order to explore their effects on the speed and accuracy of near-duplicate detection in Twitter. 8 different tweet representations were used in the experiments:

Full Text Original tweet text without any form of filtering.

Filtered Punctuation All non alphanumeric characters are removed.

Stopped and Stemmed Text We filtered English language stop words such as “the” or “and” which help build ideas but do not carry significance themselves [13]. We went one step further by reducing inflected words to their stem, base or root form in a process known as stemming. For example, stemming reduces the words "fishing", "fished", and "fisher" to the root word, "fish".

Full Shingles First, what are shingles and why use them?A contiguous subsequence of tokens in text is referred to as a shingle. Given a document D we define its w-shingling S(D,w) as the set of all unique shingles of size w contained in D . So for instance the 4-shingling of (a,rose,is,a,rose,is,a,rose) is the set{ (a,rose,is,a), (rose,is,a,rose), (is,a,rose,is) } [6]. Shingles theoretically take a more human approach to similarity matching. This is due to the fact that similar items within documents are not just characters or tokens, but matching chunks of the document. Shingles potentially convey more information and context than is obtainable using a feature based approach to data analysis. Also, shingles implicitly encode the sequence of data.

In our work we use 3-shingling since tweets contain much smaller than larger documents where 4-shingling or 5-shingling might be used. Also, we believe 2-shingling is too close to the unary tokens. Given two tweets, A and B, with sets

of shingles, S(A) and S(B) respectively then using the Jaccards coefficient, the similarity S_{sh} of A and B can be defined as:

$$S_{sh}(A,B) = \frac{|S(A) U S(B)|}{|S(A) n S(B)|}. \text{ Also, } 0 \leq S_{sh}(A,B) \leq 1$$

Figure 3.7: Shingle similarity calculation based on Jaccards Coefficient

For the purposes of our experiments, full shingling denotes shingles based on unfiltered tweet text.

Filtered Punctuation Shingles This representation comprises shingles based on tweets which have punctuation removed. Hence, we filter non alphanumeric characters then run 3-shingling on the remaining text.

Stopped and Stemmed Shingles First, we pre-process the tweet text by running stemming and stopping algorithms over the it then we run 3-shingle on the pre-processed text.

Minhash Signatures A set of integer values representing the tweet text. This concept was explained in the previous section.

Bit Vectors This is a set that stores the least significant bits of each of the integer values in a Minhash signature. The reason for this and how it is achieved is explained in a subsequent section of this section on LSH.

3.3.3 Tweet Processing

The system processes and stores tweets using different data structures depending on the similarity measurement technique and text representation to be used. The table below describes the data structures used in tweet content manipulation:

Similarity Technique	Text Representation	Shingles Representation
Levenshtein Distance	String	List of Shingles (strings)
Jaccards Coefficient	Set (consisting of strings representing individual words or Tokens)	Set of shingles
Minhashing	Set (consisting of integer hash values created from Tokens)	Set (of integer hash values created from Shingles)
Hamming Distance	Bit vectors created from Minhash signatures (set of integers created from words/ token)	Bit vectors created from Minhash signatures (set of integers created from shingles)

Figure 3.8: Data structures used in storing text representations for similarity matching using different text similarity techniques

3.3.4 Clustering Strategies

In a nutshell, our clustering strategies refer to the means by which we identify what tweets are similar and group them together. 3 different clustering strategies were employed in our experiments. Our Java implementation of clustering strategies is shown in Appendix C.

Naive Pairwise (NPW) With this method all tweets are compared with all other tweets in a brute manner. The algorithm is displayed below

NaivePairWise(R,t)

Input: a set of records R = {r_1,r_2,... r_n}, a similarity function sim(.,.) and a similarity threshold $t \in [0,1]$
Output: A super set C consisting of sub sets ci ... cn such that each c has records, {x,y,...z}, such that for any pair (x,y) in c, $sim(x,y) \geq t$
$S \leftarrow \emptyset$
for all $i = 1 \rightarrow n$ **do**
    $c_i \leftarrow \emptyset$
    $c_i \leftarrow \{r_i\}$
    $R \leftarrow R \setminus \{r_i\}$
    for all $j = 1 \rightarrow n$ **do**
        if $r_i \neq r_j$ **then**
            if $sim(r_i, r_j) \geq t$ **then**
                $c_i \leftarrow c_i \cup \{r_j\}$
                $R \leftarrow R \setminus \{r_j\}$
            end if
        end if
    end for
    $C \leftarrow C \cup c_i$
end for
return C

Figure 3.9: NaivePairWise Algorithm

Exact Duplicate Filtering This method goes a step further than the Naive pair through some pre-processing. It achieves this by checking for tweets that are exactly the same or equal and clusters them appropriately in order to avoid the overhead of using text similarity algorithms for such tweets.

Multi-Index Locality Sensitivity Hashing (LSH) To determine that two tweets might be near-duplicates, we can check their minhash signatures. However, we do not need to know the exact value of each of the hash values, only that they are equal at the same positions in the hash vector. Hence, we can take the least significant bit of each hash value in a minhash signature to form a bit vector

for each tweet. With bit vectors, LSH is able to rapidly cluster tweets into different buckets. This ensures that checks for near-duplicates are only performed on tweets within the same bucket.

Let's take a detailed look at the various logical pieces and algorithms used in our implementation of LSH. Previously, we explained minhash signatures and showed the algorithm that enables us to generate those signatures. However, even with minhash signatures, we are still comparing many tweet pairs which gives us a running time of $O(NM^2)$ where N is the size of the minhash signatures and M is the number of tweets .

You will remember that we selected N to be 200 which is even more than the 140 characters existent in a tweet. Hence, for the purposes of our work, Minhashing and the use of bit vectors are not especially aimed at reducing the effort used in directly comparing tweet pairs. However, they serve as building blocks for our implementation of LSH which is used to reduce the number of tweets pair comparisons that take place. So let us dive in.

Bit Vectors Our bit vectors are based on the concept of bit sampling which stems from the fact that we do not need to know exact hash values. All we want to know is if the hash values are equal at their respective position in each hash vector built from the hash values. Hence, we will only look at the least significant bit (LSB) of each hash value. When comparing two tweets, if the hashes are equal in the same position in the Minhash vector, then the bits in the equivalent position after bit sampling should be also equal. So, we can emulate the Jaccards similarity of two minhashes by taking the Hamming distance of the bit vectors. Efficient implementations of the hamming distance function run in near O(1) time for reasonable sizes of N [14]. This means that when comparing M tweets to each other, we have been able to reduce the time complexity to $O(M^2)$.

Multi-Index Optimization - Chunking Bit vectors helped us to reduce the running time of our tweet comparisons to $O(M^2)$, but we would like to utilize a divide and conquer strategy to increase efficiency further.

Let us use a simple example to show how bit vectors help, lets set N=32, and we want to have a bit similarity of 0.9: We need 28 of the 32 bits to be equal, or 4

bits unequal. The number of unequal bits are the radius, R of the bit vectors; i.e. if two bit vectors are within a certain radius of bits, then they are similar. If we split up XOR_mask into 4 chunks of 8 bits, then at least one chunk will have exactly zero or exactly one of the bit differences[16]. To explain further, if we split XOR_mask of size N into K chunks, with an expected radius R, then at least one chunk is guaranteed to have $floor(\frac{R}{K})$ or less bits unequal. We will choose all the parameters (N, K, and R) such that $floor(\frac{R}{K})$= 1 [15]. For our system, we have N= 200, K = 10, and R = 10 .

Multi-Index Optimization - LSH Table We can now design a data structure LSH Table using HashMaps to index the bit vectors to reduce the number of bit similarity comparisons drastically (but increase memory consumption in O(M))[17]. The algorithms are described below.

GetBitChunks (b,K)

Input: bit set b and K number of chunks required
Output: A set of bit sets B, comprising bit sets bi... bn such that the size of B is K
$B \leftarrow \emptyset$
chunkSize = $\frac{\text{size}(b)}{k}$
for all $i = 1 \rightarrow \text{size}\,(b)$ **do**

    $bi \leftarrow$bits of b from i to chunksize
    $B \leftarrow B \cup b_i$
    i = i + chunkSize

end for
return B

Figure 3.10: Splitting bit vectors into bit Chunks

InitLSH(B,N,R,K)

```
Input: a set of bit sets B = {b1,b2,... br}, where the each
b1...br represents bit sets for each tweet. N = number of bits,
K = number of chunks, R = radius
Output: A hash table H which stores other hash tables which
map bit set chunks to Lists L comprising actual bit sets.
H ← Ø
for all i → 1 to k do
    hi ← Ø //create hash maps
    H ← H ∪ hi
end for
for all i = 1 → r do
    chunks = GetBitChunks(bi,K)
    for all j = 1 → k do
        if chunksj ∉ hj then
            hj{chunksj} ← Ø
        end if
        L =hj{chunksj}
        L ← L ∪ bi
        hj{chunksj}= L
    end for
end for
return H
```

Figure 3.11: Initializing the data structure for LSH

CreateLSHBuckets(H,B,R,K)

```
Input: A hash map H = {h1,h2,... hk} created via the previous
algorithm, InitLSH. A set of bit sets B = {b1,b2,... br},
where the each b1...br represents bit sets for each tweet.
K = number of chunks, R = radius
Output: A List L = {l1, l2,... ln} such that each l is an lsh
bucket with candidate near-duplicates.
for all i = 1 → r do
    l ← 0
    l ← l ∪ {bi}
    chunks = GetBitChunks(bi,K)
    for all j = 1 → K do
        hash_table = hj
        chunk = chunksj
        for all hash_table_val ∈ hash_table do
            if count({hashval} ∖ {chunk}) ≤ R then
                lx ← Ø
                lx ←hashval{chunk}
                l ← l ∪ lx
            end if
        end for
    end for
    L ← L ∪ l
end for
return L
```

Figure 3.12: Creating LSH buckets consisting of candidate pairs

In InitLSH, we create K hash tables for each K chunks then split the bit vectors into K chunks. For each of the K chunks, we add the original bit vector into the associated hash table under the index chunk.

In CreateLSHBuckets, once more we split the bit vectors for each tweet into chunks and for each chunk look up the associated hash table for a chunk that's close depending on the defined radius, R. The returned list is a set of candidate bit vectors to check for bit similarity via Hamming distance. We also checked the tweets associated with those candidate bit vectors for similarity using the other

similarity techniques employed in our research.

To compare every M tweet to every other tweet we insert its bit vector into an LSH Table (that is an O(K) operation, where K is constant). Afterwards, to find similar tweets, we do a lookup from the LSH Table (another O(K) operation), and then check bit similarity for each of the candidates returned. The number of candidates to check is usually on the order of $\frac{M}{2^{\frac{N}{K}}}$, if at all. Therefore, the time complexity to compare all M tweets to each other is $O(\frac{M \times M}{2^{\frac{N}{K}}})$. In practice, N and K are empirically chosen such that $2^{\frac{N}{K}} \geq M$, so the final time complexity is O(M) [17].

3.4 Summary

We covered a bit of ground in this section, taking a high-level and an in-depth look at the clustering system and its various components. We started by giving a quick system overview before describing the process flow of the clustering system. The parameters to be supplied when running the system were also presented. Next, we described the similarity techniques employed: Levenshtein distance, Jaccards coefficient, Minhashing, and Hamming distance. We went on to describe the eight tweet representations were used in our work. Finally, we provided a detailed look at the various clustering algorithms we implemented.

The best way to show that a stick is crooked is not to argue about it or to spend time denouncing it, but to lay a straight stick alongside it.

- D.L. Moody

4 Experimental Design

After designing and building the clustering system, it was time for the all important experiments. We conducted several experiments to investigate the various algorithms we implemented, their performance, the validity of the results produced as well as how the various techniques and algorithms could be combined. In this section, we describe how the various pieces of our experiments were put together and how the experiments were conducted.

4.1 Objectives of Our Experiments

Here is a more formal outline of the specific problems we set out to investigate and possibly solve:

- Speed:
  - What text similarity techniques, clustering strategies, or text representations give the quickest clustering time?
- Accuracy:
  - What text similarity technique is most efficient at identifying near-duplicate clusters?
- How can the parameters of our algorithms be tuned for optimal results?
- What are the effects of tuning the parameters of our algorithms in different ways?

Clustering Strategy	Upper Bound on Data Sizes Clustered
NPW	100,000
EDF	100,000
LSH	1,000,000

Figure 4.1: Clustering Strategies and Data Sizes Clustered

4.2 Datasets and Data Sizes

The data used for our experiments was selected from the Insight on Ireland Twitter crawl maintained by the Insight Centre for Data Analytics at the University College Dublin, Ireland. The tweets used covered a few days in October, 2013. For ease of access during clustering, the data which was stored in JSON files was imported into the MySQL relational database management system.

Initially, as the algorithms were developed and/ or implemented, we utilized a dataset sized at 25,000 while clustering 1,000 to 10,000 records. Our final experiments were conducted on a dataset sized at 3 million records and we the created clusters using 50,000 to 1 million records.

The clustering strategy to be used in a particular experiment determined the maximum number of records that we could cluster given the time constraints for our work and its submission.

4.3 The Memory Limitation

Our experiments were run on a machine with a dual-core processor and 4GB memory. During our initial experiments we encountered memory errors as we approached data sizes of 100,000. The memory errors caused the clustering program to crash.

The system was built around Java's Object Oriented Programming capabilities. As tweets were retrieved from the database, tweet objects were created. To make the subsequent tweet clustering process faster, we had initialized the tweets to contain the tokens, shingles, and Minhash signatures based on the tweet text. We realized that this caused the program to create many weakly referenced objects that used up memory that could not be freed by the Java Garbage Collector. Not only did that cause the program to run out of memory, it did not speed up

the clustering process since the clustering time should take into cognisance the amount of time it takes to initialize tweet objects for the clustering process.

Furthermore, depending on the user supplied parameters to the system, a run of the program might require a set of tokens created from the tweet without requiring the creation of shingles or minhash signatures. We modified the clustering system such that tokens, shingles, and minhash signatures were created only if and when necessary. For instance, if the user wishes to run the clustering system using full text representations of the tweets, at the point in the program process flow that involves the comparison of two tweets, sets of tokens are created for both tweets and those tokens are stored in two variables. The same two variables are used to store the sets of tokens for comparing the next two tweets.

After the modifications described above, we were able to run the clustering process on data sizes ranging from 100,000 to 1,000,000.

4.4 Mix and Match

One of the first things to be done is to determine how the similarity techniques, clustering algorithms, and tweet representations are to be combined. There are many possible combinations. With more combinations, there are many more experiments to run. We arrived at a set of combinations that could be run and analysed with the required time while ensuring we have robust experiments and results. Figure 4.2 describes this:

Similarity Measurement Technique	**Text**	**Shingles**	**Minhash signatures**	**Bit vectors**
Levenshtein (NPW, EDF, LSH)	X	X		
Jaccards (NPW, EDF, LSH)	X	X		
Minhashing (NPW, EDF, LSH)			X	
Hamming distance (LSH)				X

Figure 4.2: Mixing and matching similarity technique with tweet representations representation

Points of note about the figure above:

- The table also depicts how the different clustering strategies are to be used; NPW: Naive Pairwise, EDF: Exact Duplicate Filtering, LSH: Multi-Index Locality Sensitivity Hashing.
- Both text and shingles representations can be split into: full, filtered punctuation characters, and filtered stop and stem words.
- Minhash signatures can be created based on text and shingles depending on the input supplied to the program.
- The bit vectors are based solely on minhash signatures built from shingles.

4.5 When is Similar Similar?

No matter the choice of clustering strategy or tweet representation, eventually it boils down to comparing two tweets and making judgement calls on what is similar and what is not. This means that we require a score or value that helps us make that call. We refer to that score as the Similarity Threshold. Comparing two tweets gives us a Similarity Score which is compared with the Similarity Threshold to determine if the if the two tweets are near-duplicates. We discovered that different combinations of tweet representations and similarity measurement required different similarity thresholds.

For instance, when comparing two tweets X and Y using full text representations with similarity technique Levenshtein, we could obtain a similarity score of SC1. However, comparing the same tweets using full shingle representations and the Jaccards similarity technique might yield a similarity score SC2 which is less than SC1. Hence, it is not possible to use one threshold for all tweet comparisons.

In order to determine the appropriate thresholds for different combinations of tweet representations and similarity techniques, we ran a series of similarity scoring experiments on existing near-duplicate tweet pairs. The experiments were conducted using different representations of such tweets. This helped us to identify what similarity threshold was necessary to correctly identify near-duplicates given a specific combination of tweet representation and similarity measurement technique. Figure 4.3 depicts the various similarity thresholds that were chosen:

Similarity Technique	Similarity Threshold	
Levenshtein	Text	Shingles
	0.50	0.25
Jaccards	Text	Shingles
	0.50	0.35
Minhashing	0.10	
Hamming Distance	0.65	

Figure 4.3: Similarity thresholds

It is worthy of note that in order to improve the robustness of our experiments, we considered 3 sets of similarity thresholds per combination of similarity technique and tweet representation. These are:

UpperBound: This is the high similarity threshold at which there is an expected risk of false positives.

Exact: An optimal threshold that minimizes false positives and negatives.

LowerBound: At this, we expect that there is an increased possibility of false negatives.

However, the three thresholds per experiments would triple the number of experiments and results. Hence, for simplicity we only used the exact thresholds which were displayed in Figure 4.3.

4.6 The Algorithm/Parameter Tuning Conundrum

The performance of most algorithms depend on configurable parameters that can be tuned for various scenarios. In our work, this was particularly the case with LSH. Also, the manner of implementation of Levenshtein affects its performance and was a factor in our selection of the library used to incorporate it into our clustering system.

4.6.1 The Case of Levenshtein Distance

The initial version of the Levenshtein algorithm has a time complexity and a space complexity of O(NM) where, N and M are the lengths of the two strings

being compared. This is because it creates a matrix of $n \times m$ which is used for the distance computation [18]. Hence, if both strings are 1,000 characters long, the resulting matrix is 1MB and if the strings are all integers, the size of the matrix is 4×1MB = 4MB. This simple and straightforward method takes exponential time.

We require a more efficient means of implementing Levenshtein. Luckily for us, there is an algorithm that does just that via the use of dynamic programming [19]. It is based on the realization that at any point in time, Levenshtein only requires 2 rows (or 2 columns) of the matrix in memory. This reduces the space complexity to O(min(N,M)).

A further improvement describes a variant that takes two strings and maximum edit distance of K [20]. This means that we are only interested in the edit distance if it is below a threshold K. This algorithm computes and stores a part of the dynamic programming table around its diagonal. This algorithm has a time complexity O(K× min(N,M)) where N and M are the lengths of the strings. Its space complexity is O (2K + 1).

Having found out ways to improve the performance of Levenshtein distance in our experiments, it was time to implement or use an existing implementation. In continuance of our lucky streak. The Apache Commons Lang[21] java programming library implements the Levenshtein algorithm with the modifications just described and we were able to use it for our experiments.

4.6.2 Tweaking LSH

The LSH algorithm makes use of three parameters whose values are chosen empirically:

N: The number of hash values in the Minhash signature.

K: The number of bit chunks built created from the bit vector (which is built using the Minhash signature).

R: The radius of two bit vectors which is used to measure the difference or similarity between them.

In the previous section, we explained how our LSH algorithm works. Here, we can start by making a rather obvious statement that the selection of N, R, and K affects the performance of LSH. Those 3 parameters should also be chosen in conjunction with the size of the dataset on which LSH is to be run. Let us see why this is the case.

The value of K means the bit vector of size N is split into K chunks. This means each bit chunk has a size of$\frac{N}{K}$. The Radius R is the allowed difference in bits between any 2 bit vectors. The actual value used to select candidates for possible similarity is floor($\frac{R}{K}$ which is the allowed bit difference for each bit chunk. Hence, with N = 100, K = 10, and R= 10, there are 10 bit chunks which contain 10 bits each. Each of those chunks are compared with the 10 bit chunks of other bit vectors and if the difference in corresponding chunks are between 0 and 1 (floor($\frac{R}{K}$) bits then the tweets representing each of those bit vectors might be similar and are placed in the same LSH bucket. Tweets in the same LSH bucket are checked for similarity using the NaivePairWise algorithm described in the previous Section.

Remember that, as explained in the previous Section, when using LSH the number of candidates to check is usually on the order of $\frac{M}{2^{\frac{N}{K}}}$ Fast search in hamming space. Therefore, the time complexity to compare all M messages to each other is $O(\frac{M \times M}{2^{\frac{N}{K}}})$. In practice, N and K are empirically chosen such that $2^{\frac{N}{K}} \geq M$.

The following scenarios help to describe some experiments conducted to choose optimal N, R, K:

- **Using 100,10,10 (N,R,K) with a dataset of size 100k:** This test was not completed in 60 minutes and was aborted. Here, the chunk size becomes 10. We iterate normally through bit chunks while initializing/creating the LSH data structure, but many bit vectors are mapped to the same bucket. This is because with 100,000 records and N =100, we have a bit vector with 100 least significant bits selected from the 100 integer hash values in the Minhash signature. There is an increased likelihood that within 100k records, there will be records with similar least significant bits making up their bit vector. Hence, we have larger LSH buckets containing more items and a lot of time is spent running the Naive Pairwise algorithm in those larger buckets.

- **Using 200,10,10 (N,R,K) with a dataset of size 100k:** This test was completed in 25 minutes. Fewer bit vectors or tweets are mapped into a single LSH bucket as now we have more variety in the values contained in the bit buckets. As a result after initializing the LSH data structure and mapping tweets to LSH buckets, the naive clustering process within those buckets is much faster than the prior. Also, there was no major difference noticed between 100,10, 10 and 200,10,10 for 100k in the time it took to initialize and map tweets to bit buckets. The major difference was the clustering time within those buckets.

- **Using 500,25,25 with 100k:** This test took about 34 minutes. Hence, even though 500,10,10 has the same results with 200,10,10 in terms of the $2^{\frac{N}{K}} \geq M$ rule, we have larger bit vectors and more bit chunks in this scenario which possibly led to the increased overall LSH clustering time. Hence for 100k, 500,25,25 is not optimal as 200,10,10 provided valid near-duplicate clusters in a shorter time.

- **Using 200,10,20 with 100k:** This test took about 14 minutes, 11 minutes less than 200,10, 10. This was an interesting and somewhat unexpected result which occurred consistently over multiple test runs. We believe that given the bit chunk radius of 2, there were a few more tweets mapped into the same LSH bucket, but much fewer LSH buckets in total.

- **Increasing the dataset size to 1M:** When we increase the data set size to 1 million, 200,10,10 is not optimal as we have a lot more records to process. Hence, many records have similar bit chunks leading to many records mapped to the same bucket which causes longer clustering time in those buckets We tried:

  - 200,10,10: It was not completed after 48 hours and was aborted.
  - 300,10,30: The process was completed in 47 hours and 13 minutes.
  - 500,20,20: The program ran out of memory and crashed within 3 hours.

In conclusion, when increasing chunk size ($\frac{N}{K}$), we should increase R such that floor $\frac{R}{K} > 1$. For instance, 200,10,10 means we want an overall bit similarity of .95 since 200$\times$.95 = 190 bits should be equal and the radius becomes 200 – 190

= 10 so for each chunk, the radius_per_chunk is floor ($\frac{R}{K}$ = $\frac{10}{10}$ =1). If we want an overall bit similarity of 0.90, then 200$\times$.90 = 180 so radius is 20 and we'll have 200, 10, 20 which gives the radius_per_chunk as 2. Hence, if we increase N and cause chunk size ($\frac{N}{K}$) to increase, we should increase the radius in such a way that we preserve the overall bit similarity we are aiming for which could be 0.90 or 0.95 etc. It is also interesting to note that the manipulation of the radius could also affect the size of the records eventually clustered into in buckets... and could affect false positives/false negatives.

4.7 Summary

In this section, we addressed the design of our experiments. First, we outlined the objectives of the experiments. Afterward, we described how the similarity measurement techniques could be combined with the clustering strategies and tweet representations. Also, the different combinations require specific similarity thresholds for effective near duplicate detection as we described. We studied the configurable parameters of clustering algorithms and examined the effects of the parameters. In particular, we explored how the LSH parameters (N, R, K) should be chosen to achieve optimal clustering results. The experiments cannot run without data and we described the data used as well as the dataset sizes. Finally, we looked at the memory requirement of the clustering process.

It is a capital mistake to theorize before one has data. Insensibly one begins to twist facts to suit theories, instead of theories to suit facts.

- Sir Arthur Conan Doyle

5 Results and Analysis

To recap, our research deals with the identification of near-duplicates in Twitter. Specifically, we wish to find out how different combinations of similarity measurement techniques, clustering strategies, and tweet representations affect the speed and accuracy of the near-duplicate clustering process. Here is a quick look at the various elements of our experiments:

Similarity Measurement Techniques: Levenshtein distance, Jaccards coefficient, Minhashing, Hamming distance

Clustering Strategies: Naive Pairwise(NPW), Exact Duplicate Filtering (EDF), and Multi-Index Locality Sensitive Hashing (LSH)

Tweet Representations: 8 different representations of tweets which are Full text, Filtered punctuation text, Stopped and stemmed text, Full shingles, Filtered punctuation shingles, Stopped and stemmed shingles, minhash signatures, and Bit vectors.

In this section, we seek to make clear deductions from our experiments by answering these research questions:

1. Speed

    (a) Effect of the clustering strategies on clustering speed:

        i. Which clustering strategy provides the fastest clustering speed?
        ii. How does the running time of each clustering strategy grow as the data size grows?

(b) Effect of text representation on clustering speed: Which text representation gives us the fastest speed and how does it affect the behavior of each clustering strategy?

(c) How does the fastest clustering strategy behave as we increase the data size using a staggered approach?

2. Accuracy: Which text similarity measurement technique is best at identifying near-duplicate Twitter tweets?

The rest of this section is organized into sub sections with each subsection catering to one of our research questions.

5.1 Speed

5.1.1 Effect of Clustering Strategies on Clustering Speed

In order to the evaluate the speed of the three clustering strategies built into our clustering system, we clustered different sizes of tweets using all three strategies and two similarity measurement techniques, Levenshtein distance and Jaccards Coefficient. We chose those two techniques because they operate directly on the tweet text and as such give us clearer information on the behavior of the clustering strategies. This set of experiments used the full tweet text that had not been filtered or modified in any way. As described in Figure 4.3, different similarity thresholds were required for different combinations of similarity measurement techniques and tweet representations. Interestingly Levenshtein distance and Jaccards coefficient required the same similarity threshold, 0.5, when run on plain tweet text representations (filtered or unfiltered).

What Clustering Strategy Provides the Best Speed? In this experiment, 100,000 tweets were clustered using Naive Pairwise (NPW), Exact Duplicate Filtering (EDF) and Multi-Index Locality Sensitivity Hashing (LSH).

Figure 5.1 shows that LSH is the fastest clustering strategy by some distance. No real surprises there. However, it was surprising to discover that EDF was noticeable slower than NPW. We expected the opposite. It is worthy of note at

this point that the clustering time for EDF included the time it took to filter exact duplicates. We believe that this increased the overall clustering time. We also believe that this indicates that there weren't many exact duplicates. Hence, after filtering exact duplicates, a considerable amount of time was still necessary to cluster near-duplicates.

Another interesting discovery is that while Jaccards coefficient is significantly faster than Levenshtein distance for NPW and EDF, both techniques have the same running time with LSH. How could this be? We discovered that with appropriately selected parameters as displayed in Fig 4.3, LSH could be used to effectively cluster near duplicates into buckets such that very little naive comparison is necessary within those buckets. To achieve this, all three parameters have to be chosen empirically with the size of the dataset in consideration. For this set of experiments, we set, N=200, K=10, and R=20. To be able to maintain such efficient clustering of near-duplicates using mostly LSH, we need to increase the number of hash values, N, as the dataset increases. This would require an increase in memory requirements as a set of hash values is stored for every tweet that is to be clustered. Also, the values of R and K should also be modified depending on the desired outcomes.

Observing the Growth in Running Time of the Clustering Strategies We went a step further by exploring how the running time of the clustering process was affected as we steadily increased the size of the dataset. The focus shifted to the top two clustering strategies as we eliminated EDF which performed worst in the previous experiment.

The maximum dataset size used here was 100,000 since with NPW it took almost 2000 minutes to cluster that number of tweets using Levenshtein distance. We generated clusters based on 4 different dataset sizes: 25,000; 50,000; 75,000; and the 100,000 maximum size.

Once again, LSH outperforms NPW on all dataset sizes. Even more interesting is that the running time of LSH grows linearly as we increase the dataset size. This proves our statement in Section 3.3.4, that it takes O(M) time to cluster M tweets using LSH. However, we must remember that to sustain O(M) clustering time for LSH, the LSH parameters have to be tweaked as the dataset size increases or indeed decreases.

The growth in clustering time of NPW depends on the similarity technique in use. We can see in Fig 5.2 that the growth in clustering time for NPW & Jaccards though not exactly linear, is quite close. On the other hand, clustering using NPW & Levenshtein definitely occurs in exponential time as we increase the size of our dataset.

5.1.2 The Effect of Tweet Representation on Clustering Speed

On this leg of our adventure, we set out to find what tweet representation provided the fastest results in terms of clustering speed while still producing valid clusters. Afterward, we would run the same experiments presented in section 5.1.1 on the same Twitter datasets using the best tweet representation this time around to observe its effects on the clustering process, if any.

To determine the best tweet representation, we clustered 50,000 tweets using NPW with Levenshtein distance, Jaccards coefficient, and Minhashing. We ran those tests using all tweet representations described in Section 3.3.2. We discovered that using plain text representations was faster than shingles or minhash signatures for clustering tweets. When clustering documents or web resources, minhash signatures are used to reduce the amount of values to be compared for each document. However, in the case of the tweets, this does not apply because even though the tweets are already very short in length, we still require reasonably large minhash signatures (sometimes lengthier than the tweet) to effectively represent tweets during clustering. Shingles [6]are used to cluster near-duplicates in a more human-like manner, but it was also slower than plain text. It was also important to check that even though text representations were faster during clustering, they still gave identified near-duplicates effectively. We saw no major difference in the clusters identified using plain text, shingles, minhash signatures, or bit vectors. The key to this was the identification and use of appropriate similarity thresholds for different tweet text representations.

Once, we identified plain text representations as superior, it was necessary to identify which plain text representation was the best. Our experiments showed that the Filtered Punctation tweet text representation performed better than the other plain text representations, Full Text, and Stopped and Stemmed Text. This is easily explained since unlike the other two text representations, Fil-

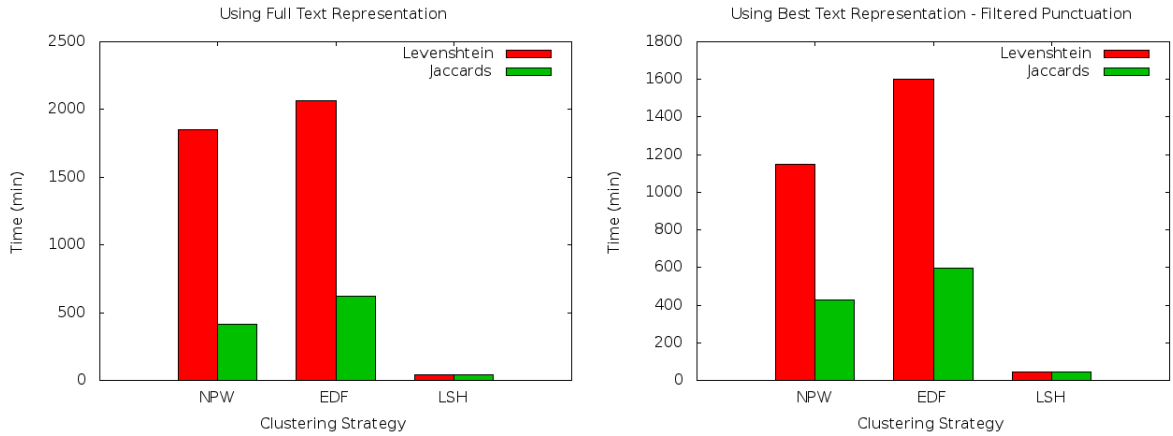

Figure 5.1: Evaluating clustering speed using Levenshtein, jaccards with NPW, EDF, LSH on 100k tweets

tered Punctuation removed several punctuation characters thereby shortening the text. Also, since we still obtained valid clusters, we can conclude that the punctuations characters were not useful for identifying near-duplicate tweets.

In subsequent sections we will run the same experiments described in Section 5.1 using the best tweet representation we have identified.

Speed of Clustering Strategies using the Best Tweet Representation As Figure 5.1 shows, the effect of the best tweet representation is best observed when clustering using Levenshtein. On the other hand, Jaccards coefficient seems indifferent to the change in tweet representation. We should note that this representations are in the top 3 for speed according to our research. Hence, we can expect Jaccards coefficient to show a noticeable change in clustering speed if we were to use shingling or Minhashing.

Growth in Running Time using the Best Tweet Representation Once again the most noticeable effect of the best tweet representation is seen when clustering using Levenshtein and NPW. As we explained in Section 5.1, using LSH with 100,000 tweets, we were able to set LSH parameters such that it was difficult to distinguish between similarity measurement techniques regarding clustering speed. We can also see that the change in tweet representation has no effect on LSH. This is because the filtered text has little or know effect on the

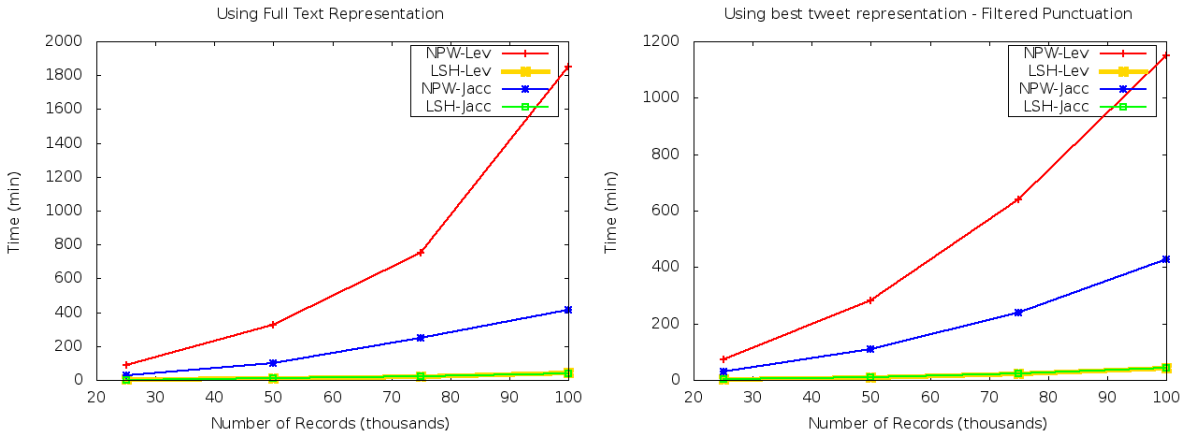

Figure 5.2: The behaviour of clustering strategies as the size of the dataset increased

LSH algorithms described in Section 3.3.4.

5.1.3 Scaling up the Dataset Size for the Fastest Clustering Strategy

Here, we wish to explore the growth in running time of the best clustering process (LSH), and the best tweet representation (Filtered Punctuation text) as we progressively scale up the dataset size to a maximum of 1 million records. For this experiment we will be comparing all similarity measurement techniques: Levenshtein distance, Jaccards coefficient, Minhashing, and Hamming Distance.

Our results here show that Jaccards coefficient gives the quickest clustering speed as the dataset size increases. Also, while Jaccards coefficient, Levenshtein distance, and Minhashing seem to have almost linear growth in running time in relation to increased dataset sizes, the running time growth for Hamming distance is more exponential. As the dataset size is steadily increased, Hamming distance progressively gives poor clustering speed results in comparison with the other similarity measurement techniques.

To ensure that we had the best clustering speeds, as we increased the dataset size, we modified the LSH parameters appropriately. Figure 5.3 shows that we ran this experiment for 3 dataset sizes, 340000, 680000, and 1,000,000. For the first two dataset sizes, our values for the LSH parameters N, R, K were 200,10,20 respectively while for the largest dataset size of 1,000,000, the LSH parameters were 300,10,30.

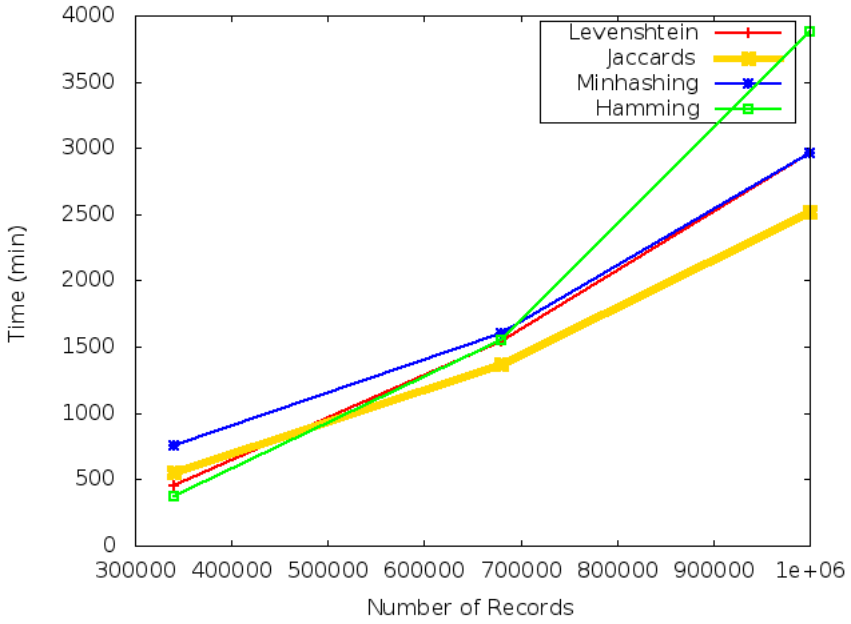

Figure 5.3: Scaling up LSH

5.2 Accuracy

All previous experiments focused on the speed of the clustering process. Here, we sought to gauge how effective each similarity measurement technique is in identifying near-duplicate clusters. To accomplish this, we generated clusters using each of the 4 similarity measurement techniques. Afterward, 1000 random clusters were selected from the 4 cluster sets. This was done by randomly selecting a similarity technique and randomly selecting one of its clusters until we obtained a random cluster set sized at 1000. Finally, the cluster sets produced by each technique were checked against the random cluster set. The aim being to select clusters as randomly as we possibly could before checking how many of those random clusters were found by each of the similarity techniques. To adjudge whether two clusters were the same we employed an error factor of 5%. Hence, if there was a 0 - 5% difference in the content of two clusters, those clusters were considered to be equal.

Similarity Technique	Accuracy
Levenshtein Distance	78.5%
Jaccards Coefficient	80.2%
Minhashing	55.2%
Hamming Distance	54.8%

Figure 5.4: Evaluating the Accuracy of the Similarity Measurement Techniques using NPW and 20,000 records

5.2.1 Using NPW

For this experiment,we utilized the NPW clustering strategy and dataset sizes of 20,000 records. As depicted in Figure 5.4, Jaccards coefficient performed best with Levenshtein distance coming a very close second. Minhashing and Hamming distance were a distant third and fourth respectively. Our results were not surprising since the minhash signatures used for Minhashing reduce the actual tweets to a set of integer values while the bit vectors using with Hamming distance reduces the minhash signatures further to a set comprising the least significant bits of each hash value in a minhash signature. This helps to speed up the clustering process via LSH as described in section 3.3.4.

5.2.2 Using LSH

All techniques performed much better when run with LSH against dataset sizes of 20,000 as depicted in Figure 5.5. We assumed that with that number of records not much clustering was performed using the similarity techniques since LSH had effectively partitioned potential near-duplicates into separate LSH buckets. We decided to run out tests using 680,000 records. As we increased the sample, we obtained better accuracy results across all similarity measurement techniques as shown in Figure 5.6. This is due to the fact that given the right LSH parameters, even as the dataset sizes increases, very little NPW type clustering should be done in LSH buckets. Still, even with LSH we can see Levenshtein distance and Jaccards coefficient have the edge. This leads us to conclude that the most accurate near-duplicate clustering results can be achieved by combining LSH with Levenshtein distance or Jaccards coefficient.

Similarity Technique	Accuracy
Levenshtein Distance	99.9%
Jaccards Coefficient	99.8%
Minhashing	98.7%
Hamming Distance	99.5%

Figure 5.5: Accuracy of the Similarity Measurement Techniques using LSH and 20,000 records

Similarity Technique	Accuracy
Levenshtein Distance	99.9%
Jaccards Coefficient	99.9%
Minhashing	99.2%
Hamming Distance	99.8%

Figure 5.6: Accuracy of the Similarity Measurement Techniques using LSH and 680,000 records

5.3 Summary

This section analyzed the results of different experiments which answered our four research questions. We examined the speed of our clustering strategies and while LSH was the unsurprising winner, EDF performed worse than was expected. To observe the growth in running time of the best two clustering strategies, we steadily increased the size of the dataset from 25,000 to 100,000. We noticed that running time of LSH grew linearly and that proved that it takes O(M) time to cluster M tweets using LSH. To explore the effect of tweet representation on clustering, we sought and found the best tweet representation for clustering, Filtered Punctuation, and explored its impact on the clustering strategies and similarity techniques. Since LSH had been performing well, we scaled up the dataset size significantly while clustering with LSH to observe the effect on LSH and the different similarity techniques. Finally, we studied the accuracy of the similarity measurement techniques by putting them to the test in the identification of a set of random clusters.

Success is not a place at which one arrives but rather the spirit with which one undertakes and continues the journey.

- Alex Noble

6 Conclusion

The objective of our work is the evaluation of different approaches and algorithms for near-duplicate detection with a focus on the Twitter microblogging tool. Near-duplicate detection in Twitter is of increasing importance due to the primary role it plays in first story detection, spam detection, and many other clustering processes.

Specifically, we sought to understand how different combinations of similarity measurement techniques, clustering strategies, and tweet representations affect the speed and accuracy of the near-duplicate clustering process. To achieve this, we built a clustering system that includes the following functionality:

- Generation of near-duplicate clusters.
- Validation of clusters by selecting the most frequent clusters discovered by our experiments.
- Comparison of the near-duplicate clusters found by using different text similarity measurement techniques.
- Compilation of statistics on clusters found using experiments.

The clustering system incorporates 4 similarity measurement techniques namely Levenshtein distance, Jaccards coefficient, Minhashing, and Hamming distance. We also designed and/or implemented 3 clustering strategies: Naive Pairwise (NPW), Exact Duplicate Filtering (EDF), and Multi-Index Locality Sensitive Hashing (LSH). Finally, to discover the optimal text representation for detecting near-duplicates in the context of Twitter, we utilized 8 tweet representations namely

Full text, Filtered punctuation text, Stopped and stemmed text, Full shingles, Filtered punctuation shingles, Stopped and stemmed shingles, minhash signatures, and Bit vectors.

Speed We designed a variety of experiments as described in Figure 4.1 and 4.2 to ascertain what combination of similarity measurement techniques, clustering strategies and tweet representations provide us with the quickest speed in the clustering process. Our results analysis showed us the following:

- Jaccards coefficient similarity measurement technique offered the quickest clustering speed.
- Due to the small size of tweets, plain text representations gave the best speed as opposed to shingles, minhash signatures or bit vectors. Also, we were able to select Filtered punctuation text as the best tweet representation in terms of clustering speed.
- In terms of clustering strategies, LSH gives the fastest clustering speed followed by NPW and EDF respectively.
- Given the right LSH parameters, the running time of the LSH based clustering process grows linearly as we increase the dataset size. This proves our statement in Section 3.3.4, that it takes O(M) time to cluster M tweets using LSH.

Accuracy Lastly, we measured the accuracy of the similarity measurement techniques in identifying near-duplicates. We discovered that:

- When clustering using the Naive Pair Wise strategy,Jaccards coefficient performed best with Levenshtein distance coming a very close second. Minhashing and Hamming distance were a distant third and fourth respectively.
- Using LSH, the most accurate similarity measurement techniques were still Levenshtein distance and Jaccards coefficient with their results being almost identical. However, the accuracy results of all similarity measurement techniques were substantially better when combined with LSH.

Minhashing and Hamming distance were a much closer third and fourth compared to their results with NPW.

- Levenshtein distance or Jaccards coefficient combined with LSH provides the most accurate near-duplicate clustering results.

Appendix A

Implementation of Building Blocks

In this section we present some of the central Java code in our clustering system. The Java classes presented here are essential to the understanding of the code presented in Appendix B and C.

A.1 Tweet

This class is used to create an object to store data for a single tweet. Some of the implementation code is shown here. The code displayed is restricted to the initial declaration of member variables since this conveys just enough information about how the class functions.

```
package ie.ucd.nearduplicates.utils;
import ie.ucd.nearduplicates.main.Main;
import ie.ucd.nearduplicates.techniques.MinHashWrapper;
import java.util.ArrayList;
import java.util.BitSet; import java.util.Set;
public class Tweet
{

    private String tweetId;
    private String text;
    private String filteredPunctuationText;
    private String filteredStopStemText;
    private boolean processed;
    private int clusterNumber;
    private ArrayList<String> shinglesList;
    private Set<String> shinglesSet;
    private Set<String> tokens;
    private int[] minHashFromTokens;
    private int[] minHashFromShingles;
```

```
    private BitSet bitSet;
    /* the rest of the class is not shown here*/

}
```

A.2 Clustering Details

This class is used to store the clustering preferences entered by the user when starting the clustering program. The code shown is restricted to the declaration of member variables and the constructor.

```
package ie.ucd.nearduplicates.main;
public class ClusteringDetails
{

    private int technique;
    private int methodology;
    private int representation;
    private double similarityThreashold;
    private int dataLimit;

public ClusteringDetails(int technique, int methodology,
int representation, double similarityThreashold, int dataLimit)

    {
       super();
       this.technique = technique;
       this.methodology = methodology;
       this.representation = representation;
       this.similarityThreashold = similarityThreashold;
       this.dataLimit = dataLimit;
    }
    /* the rest of the class is not shown here*/

}
```

A.3 Main

This is the entry point to our clustering system.

```
package ie.ucd.nearduplicates.main;
import ie.ucd.nearduplicates.model.TweetModel;
import ie.ucd.nearduplicates.techniques.LSH;
import ie.ucd.nearduplicates.utils.FileUtils;
import ie.ucd.nearduplicates.utils.Tweet;
import java.util.ArrayList;
import java.util.BitSet;
import java.util.HashMap; import java.util.Scanner;
public class Main
{


    //constants that define the various user selectable options
    //techniques
    public static final int LEVENSHTEIN = 1;
    public static final int JACCARDS = 2;
    public static final int MINHASHING = 3;
    public static final int HAMMING = 4;

//clustering methodology

    public static final int UNPROCESSED = 1;
    public static final int DUPLICATES_PROCESSED = 2;
    public static final int LSH = 3;

//tweet representation

    public static final int FULL_TEXT = 1;
    public static final int FILTERED_PUNCTUATION_TEXT = 2;
    public static final int FILTERED_STOP_STEM_TEXT = 3;
    public static final int FULL_SHINGLES = 4;
    public static final int FILTERED_PUNCTUATION_SHINGLES = 5;
    public static final int FILTERED_STOP_STEM_SHINGLES = 6;
    static String technique_text;
```

```
    static String methodology_text;
    static String representation_text;
    static String threshold_text;
    static String data_limit_text;
    static int technique_val;
    static int methodology_val;
    static int representation_val;
    static double threshold_val;
    static int data_limit_val;
    public static ClusteringDetails clusteringDetails;

public static int currentDataSize;
public static void main(String[] args)

    {
       if(args.length == 5)
       {

          technique_text = args[0];
          methodology_text = args[1];
          representation_text = args[2];
          threshold_text = args[3];
          data_limit_text = args[4];
          processInput();
       }
       else
       {
          getInput();
          processInput();
       }
       clusteringDetails = new ClusteringDetails(technique_val,
       methodology_val, representation_val,threshold_val,
       data_limit_val);
       TweetModel tweetModel = new TweetModel();
       if(methodology_val == UNPROCESSED)
```

```
{

   HashMap<String,Tweet> tweets = tweetModel.getTweets();
   ClusterMaker clusterMaker = new ClusterMaker(tweets,
   clusteringDetails);
   long startTime   = System.nanoTime();
   clusterMaker.generateNearDuplicateClusters();
   long endTime   = System.nanoTime();
   double totalTime =
   (double)(endTime - startTime)/1000000000;
   String timeSpent = "\n"+ Main.currentDataSize +
   "\t"+ totalTime;
   FileUtils.saveClusteringTime(timeSpent,
     clusteringDetails);
}
else if(methodology_val == DUPLICATES_PROCESSED)
{

   HashMap<String,Tweet> tweets = tweetModel.getTweets();
   ClusterMaker clusterMaker = new ClusterMaker(tweets,
   clusteringDetails);
   long startTime   = System.nanoTime();
   clusterMaker.generateNearDuplicateClusters();
   long endTime   = System.nanoTime();
   double totalTime =
     (double)(endTime - startTime)/1000000000;
   String timeSpent =
     "\n"+ Main.currentDataSize + "\t"+ totalTime;
   FileUtils.saveClusteringTime(timeSpent,
     clusteringDetails);
}
else if (methodology_val == LSH)
{
```

```
        HashMap<BitSet,ArrayList<Tweet>> tweetsAndBits =
          tweetModel.getTweetsAndBits();
        LSH lsh =
          new LSH(tweetsAndBits,300,10,30,clusteringDetails);
        long startTime   = System.nanoTime();
        ArrayList<HashMap<BitSet,ArrayList<BitSet>>>
          initLSHMap = lsh.createLSHInitMap();
        HashMap<Integer,ArrayList<BitSet>> lshBuckets
          = lsh.createLSHBuckets(initLSHMap);
        lsh.generateClustersFromLSH(lshBuckets);
        long endTime   = System.nanoTime();
        double totalTime =
          (double)(endTime - startTime)/1000000000;
        String timeSpent =
          "\n"+ Main.currentDataSize + "\t"+ totalTime;
        FileUtils.saveClusteringTime(timeSpent,
          clusteringDetails);
    }
}
public static void getInput()
{
    //handling the input of the technique
    System.out.println("Enter Technique : ");
    Scanner scanIn = new Scanner(System.in);
    technique_text = scanIn.nextLine();
    System.out.println("Enter Cluster Methodology: ");
    methodology_text = scanIn.nextLine();
    System.out.println("Enter Tweet Representation choice:");
    representation_text = scanIn.nextLine();
    System.out.println("Enter Similarity Threshold value:");
    threshold_text = scanIn.nextLine();
    System.out.println("Enter Data Set Size Limit:");
    data_limit_text = scanIn.nextLine();
    scanIn.close();
```

```
}
public static void processInput()
{
   //handling the input of the technique
   try
   {

      technique_val = Integer.parseInt(technique_text);
      if(technique_val < 1 || technique_val > 4)
      {

        System.out.println("Incorrect input of technique");
        System.exit(0);
      }
      methodology_val = Integer.parseInt(methodology_text);
      if(methodology_val < 1 || methodology_val > 3)
      {

        System.out.println("Incorrect input of" +
          "clustering methodology");
        System.exit(0);
      }
      representation_val =
        Integer.parseInt(representation_text);
      if(representation_val < 1 || representation_val > 6)
      {

        System.out.println("Incorrect input of"+
          "representation");
        System.exit(0);
      }
      threshold_val = Double.parseDouble(threshold_text);
      if(threshold_val < 0 || threshold_val > 1)
      {
```

```
            System.out.println("Incorrect input of threshold");
            System.exit(0);
          }
          data_limit_val = Integer.parseInt(data_limit_text);
          if(data_limit_val <= 0)
          {

            System.out.println("Incorrect input of data limit");
          }
        } catch(NumberFormatException e)
        {

          System.out.println("Incorrect input");
          System.exit(0);
        }
    }// end process input
}
```

Appendix B

Implementation of Similarity Measurement Techniques

B.1 Levenstein Distance

The Java class below as used to calculate Levenshtein distance and similarity based on the Apache Commons Lang library[21].

```
package ie.ucd.neardуplicates.techniques;
import java.util.ArrayList;
import ie.ucd.neardуplicates.utils.MathUtils;
import org.apache.commons.lang3.StringUtils;
public class Levenshtein {


    public static float getLevenshteinSimText(String text1,
    String text2)
    {
       int text1length = text1.length();
       int text2length = text2.length();
       //non-normalized distance
       int distance = StringUtils.getLevenshteinDistance(text1,
       text2);
       //normalized distance
       float normalizedDistance =
         (float)distance / MathUtils.max(text1length, text2length);
       float similarity = 1 - normalizedDistance;
       return similarity;
    }
    /*this implementation operates on text and checks that
     *the levenshtein similarity is within a threshold*/
    public static float getLevenshteinSimTextV2(String text1,
```

```
String text2, int thresholdDistance)
{
   int text1length = text1.length();
   int text2length = text2.length();
   //non-normalized distance
   int distance =
   StringUtils.getLevenshteinDistance(text1,
     text2, thresholdDistance);
   if(distance == -1)
   {

      //then its not similar so just return 0
      return 0;
   }
   //normalized distance
   float normalizedDistance =
     (float)distance / MathUtils.max(text1length,
     text2length);
   float similarity = 1 - normalizedDistance;
   return similarity;
}
public static float getLevenshteinSimShingles
(ArrayList<String> list1, ArrayList<String> list2)
{
   int list1Size = list1.size();
   int list2Size = list2.size();
   float collatedDistance;
   float miniDistance;
   int nosComparisons = 0;
   String shingle1 = "";
   String shingle2 = "";
   int counter;
long startTime2= System.currentTimeMillis();
   if (list1Size >= list2Size)
```

```
{
   counter = 0;
   miniDistance = 0;
   collatedDistance = 0;
   for (int i=0; i< list1Size; i++)
   {

   //if we haven't gotten to the end of list2
     if(i<list2Size)
     {

       shingle1 = list1.get(i);
       shingle2 = list2.get(i);
       miniDistance =
         StringUtils.getLevenshteinDistance(shingle1,
         shingle2);
       collatedDistance +=
         (float)miniDistance /
            MathUtils.max(shingle1.length(),
            shingle2.length());
     }
     else
     {

       collatedDistance += 1;
     }
   }
   nosComparisons = list1.size();
}
else //list2 is greater than list 1
{

   //mainly, we go through list 2
```

```
            counter = 0;
            miniDistance = 0;
            collatedDistance = 0;
            for (int i=0; i< list2Size; i++)
            {

              //if we haven't gotten to the end of list1
              if(i<list1Size)
              {

                shingle1 = list1.get(i);
                shingle2 = list2.get(i);
                miniDistance +=
                  StringUtils.getLevenshteinDistance(shingle1,
                  shingle2);
                collatedDistance +=
                  (float)miniDistance /
                    MathUtils.max(shingle1.length(),
                    shingle2.length());
              }
              else
              {

                collatedDistance += 1;
              }
            }
            nosComparisons = list2.size();
          }
          float shingledDistance =
            (float)collatedDistance/nosComparisons;
          float shingledSimilarity = 1 - shingledDistance;
          return shingledSimilarity;
      }
  }
```

B.2 Jaccards Coefficient

```
package ie.ucd.nearduplicates.techniques;
import java.util.Set;
import com.google.common.collect.Sets;
public class Jaccard
{
public static double getJaccardSimilarity(
      Set<String> theSet1, Set<String> theSet2)

    {
       Set<String> set1 = theSet1;
       Set<String> set2 = theSet2;
       Set<String> intersection = Sets.intersection(set1, set2);
       Set<String> union = Sets.union(set1, set2);
       return (double)intersection.size()/union.size();

}
}
```

B.3 Minhashing

We implemented a minhash wrapper that used the minhashing code written by Thomas Jungblot [23].

```
package ie.ucd.nearduplicates.techniques;
import java.util.Set;
public class MinHashWrapper
{

    static double errorFactor = 0.07;
    static int numOfHashes =
      (int)(1 / (errorFactor * errorFactor));
    minHash2 = new MinHash<String>(errorFactor,setA, setB);

static long seed = 87887978;
```

```
    static int defNumHashes = 300;
    static MinHash minHash = MinHash.create(defNumHashes,seed);

public static int[] getMinHash(Set<String> set)

    {

int[] minHashes = minHash.minHashSet(set);

        return minHashes;
    }

public static double measureSimilarity(
      int[] minHashes1, int[] minHashes2)

    {

       double similarity =
         minHash.measureSimilarity(minHashes1, minHashes2);
       return similarity;

    }

}
```

B.4 Hamming Distance and Bit Similarity

We made use of hamming distance when comparing bit vectors create from the minhash signatures of tweets. The specific Java methods are show here.

```
public static int getHammingDistance(
  BitSet bitSetA, BitSet bitSetB)
{

    int distance = 0;
    BitSet clonedBitSetA =  (BitSet)bitSetA.clone();
    clonedBitSetA.xor(bitSetB);
    for(int i=0;i<clonedBitSetA.size();i++)
    {
```

```
        Boolean bitValue = clonedBitSetA.get(i);
        if(bitValue)
        {
           distance++;
        }
}
return distance;
}
public static double getBitSimilarity(
  BitSet bitSetA, BitSet bitSetB, int N)
{

    int distance = getHammingDistance(bitSetA, bitSetB);
    double similarity = 1 - ((double)distance/N);
return similarity;
}
```

Appendix C

Implementation of Clustering Strategies

C.1 Naive Pair Wise (NPW) and Exact Duplicate Filtering (EDF)

As shown in the Main class of Appendix A, the processing performed before the usage of this cluster maker class depends on the selection of NPW and EDF by the user.

```
package ie.ucd.nearduplicates.main;
import ie.ucd.nearduplicates.techniques.Jaccard;
import ie.ucd.nearduplicates.techniques.Levenshtein;
import ie.ucd.nearduplicates.techniques.MinHashWrapper;
import ie.ucd.nearduplicates.utils.ArrayUtils;
import ie.ucd.nearduplicates.utils.MathUtils;
import ie.ucd.nearduplicates.utils.TextUtils;
import ie.ucd.nearduplicates.utils.Tweet;
import java.io.BufferedWriter;
import java.io.File;
import java.io.FileWriter;
import java.io.IOException;
import java.util.ArrayList;
import java.util.BitSet;
import java.util.HashMap;
import java.util.Iterator;
import java.util.Map;
import java.util.Set;
/*This class is used to generate clusters using
  *CLusterDetails and write to a necessary data file*/
public class ClusterMaker
{
private HashMap<String, Tweet> tweets;
```

```
    private ClusteringDetails clusteringDetails;
private ArrayList<String> comparingShinglesList;
    private Set<String> comparingShinglesSet;
    private Set<String> comparingTokens;
    private int[] comparingMinHashFromTokens;
    private int[] comparingMinHashFromShingles;
    private BitSet comparingBitSet;
    private ArrayList<String> comparedShinglesList;
    private Set<String> comparedShinglesSet;
    private Set<String> comparedTokens;
    private int[] comparedMinHashFromTokens;
    private int[] comparedMinHashFromShingles;
    private BitSet comparedBitSet;
    public ClusterMaker(HashMap<String, Tweet> tweets,
      ClusteringDetails clusteringDetails)
    {
       super();
       this.tweets = tweets;
       this.clusteringDetails = clusteringDetails;
    }
public void generateNearDuplicateClusters()
{
    long startTime = System.nanoTime();
    System.out.println("\n*****Clustering near duplicates...");
    int lastClusterNumber = 0;
    HashMap<Integer, ArrayList<Tweet>> clusteredTweets
      = new HashMap<Integer,ArrayList<Tweet>>();
    HashMap<Integer, Integer> nosClusterTweets
      = new HashMap<Integer, Integer>();
    for (Map.Entry<String, Tweet> entry : tweets.entrySet())
    {

       String comparingTweetId = entry.getKey();
```

```
String comparingTweetText = entry.getValue().getText();
Tweet comparingTweet = entry.getValue();
if(comparingTweet.isProcessed())
   continue;
HashMap<String, Tweet> comparedTweets = tweets;
int tweetsFound = 1;
lastClusterNumber++;
comparingTweet.setClusterNumber(lastClusterNumber);
comparingTweet.setProcessed(true);
ArrayList<Tweet> newCluster = new ArrayList<Tweet>();
newCluster.add(comparingTweet);
clusteredTweets.put(lastClusterNumber, newCluster);
for (Map.Entry<String, Tweet> comparedEntry :
  comparedTweets.entrySet())
{

   String comparedTweetId = comparedEntry.getKey();
   String comparedTweetText
     = comparedEntry.getValue().getText();
   Tweet comparedTweet = comparedEntry.getValue();
   if(comparedTweet.isProcessed())
     continue;
   if(comparingTweetId.equals(comparedTweetId))
   {
     continue;
   }
   else if(comparingTweet.getClusterNumber() != 0 &&

     comparingTweet.getClusterNumber()
       == comparedTweet.getClusterNumber())
     continue;
   else
   {
```

```
/*here we compare tweets based on
  the clustering details*/
if(clusteringDetails.getTechnique()
  == Main.LEVENSHTEIN)
{
  double similarity = 0;
if(clusteringDetails.getRepresentation()
      == Main.FULL_TEXT ||
      clusteringDetails.getRepresentation()
      == Main.FILTERED_PUNCTUATION_TEXT ||
      clusteringDetails.getRepresentation()
      == Main.FILTERED_STOP_STEM_TEXT)
    {
       //trying out threshold for levenshtein
      int thresholdDistance =
      (int)
       (clusteringDetails.getSimilarityThreashold() *
      MathUtils.max(comparingTweetText.length(),
      comparedTweetText.length()));
 similarity =
       Levenshtein.getLevenshteinSimTextV2(
       comparingTweetText,comparedTweetText,
       thresholdDistance);
  }
  else if(clusteringDetails.getRepresentation()
    == Main.FULL_SHINGLES
   || clusteringDetails.getRepresentation()
    == Main.FILTERED_PUNCTUATION_SHINGLES
   || clusteringDetails.getRepresentation()
   == Main.FILTERED_STOP_STEM_SHINGLES)
  {
    comparingShinglesList
      = TextUtils.getShinglesList(comparingTweetText,
```

```
        3, 3);
      comparedShinglesList
        = TextUtils.getShinglesList(comparedTweetText,
        3, 3);
      similarity =
      Levenshtein.getLevenshteinSimShingles(
        comparingShinglesList,comparedShinglesList);
    }

    if(similarity >=
     clusteringDetails.getSimilarityThreashold())
    {
      tweetsFound++;
      comparedTweet.setClusterNumber(
       comparingTweet.getClusterNumber());
      comparedTweet.setProcessed(true);
      ArrayList<Tweet> existingCluster =
      clusteredTweets.get(
       comparingTweet.getClusterNumber());
      existingCluster.add(comparedTweet);
    }//end if
  }// end if technique is levenshtein
  else if(clusteringDetails.getTechnique()
    == Main.JACCARDS)
  {
    double similarity = 0;
    if(clusteringDetails.getRepresentation()
    == Main.FULL_TEXT||
    clusteringDetails.getRepresentation()
    == Main.FILTERED_PUNCTUATION_TEXT||
    clusteringDetails.getRepresentation()
    == Main.FILTERED_STOP_STEM_TEXT)
    {
      comparingTokens
```

```
    = TextUtils.getTokens(comparingTweetText);
  comparedTokens
    = TextUtils.getTokens(comparedTweetText);
  similarity
    = Jaccard.getJaccardSimilarity(comparingTokens,
    comparedTokens);
}
else if(clusteringDetails.getRepresentation()
== Main.FULL_SHINGLES||
clusteringDetails.getRepresentation()
== Main.FILTERED_PUNCTUATION_SHINGLES||
clusteringDetails.getRepresentation()
== Main.FILTERED_STOP_STEM_SHINGLES)
{
  comparingShinglesSet
    = TextUtils.getShinglesSet(comparingTweetText,
     3, 3);
  comparedShinglesSet
    = TextUtils.getShinglesSet(comparedTweetText,
     3, 3);
  similarity
    = Jaccard.getJaccardSimilarity(
    comparingShinglesSet,comparedShinglesSet);
}
if(similarity
   >= clusteringDetails.getSimilarityThreashold())
{
  tweetsFound++;
  comparedTweet.setClusterNumber(
    comparingTweet.getClusterNumber());
  comparedTweet.setProcessed(true);
  ArrayList<Tweet> existingCluster =
    clusteredTweets.get(
    comparingTweet.getClusterNumber());
```

```
    existingCluster.add(comparedTweet);
  }//end if
}// end if technique is jaccards
else if(clusteringDetails.getTechnique()
  == Main.MINHASHING)
{
  double similarity = 0;
  if(clusteringDetails.getRepresentation()
    == Main.FULL_TEXT
  || clusteringDetails.getRepresentation()
    == Main.FILTERED_PUNCTUATION_TEXT||
  clusteringDetails.getRepresentation()
    == Main.FILTERED_STOP_STEM_TEXT)
  {
    comparingTokens
      = TextUtils.getTokens(comparingTweetText);
    comparedTokens
      = TextUtils.getTokens(comparedTweetText);
    comparingMinHashFromTokens
     = MinHashWrapper.getMinHash(comparingTokens);
    comparedMinHashFromTokens
     = MinHashWrapper.getMinHash(comparedTokens);
    similarity
     = MinHashWrapper.measureSimilarity(
      comparingMinHashFromTokens,
      comparedMinHashFromTokens);
  }
  else if(clusteringDetails.getRepresentation()
    == Main.FULL_SHINGLES||
    clusteringDetails.getRepresentation()
    == Main.FILTERED_PUNCTUATION_SHINGLES
    || clusteringDetails.getRepresentation()
    == Main.FILTERED_STOP_STEM_SHINGLES)
  {
```

```
        comparingShinglesSet
          = TextUtils.getShinglesSet(comparingTweetText,
          3, 3);
        comparedShinglesSet
          = TextUtils.getShinglesSet(comparedTweetText,
          3, 3);
        comparingMinHashFromShingles
          = MinHashWrapper.getMinHash(
          comparingShinglesSet);
        comparedMinHashFromShingles
          = MinHashWrapper.getMinHash(
          comparedShinglesSet);
       similarity
         = MinHashWrapper.measureSimilarity(
         comparingMinHashFromShingles,
         comparedMinHashFromShingles);
     }
     if(similarity >=
       clusteringDetails.getSimilarityThreashold())
     {
       tweetsFound++;
       comparedTweet.setClusterNumber(
         comparingTweet.getClusterNumber());
       comparedTweet.setProcessed(true);
       ArrayList<Tweet> existingCluster =
         clusteredTweets.get(
           comparingTweet.getClusterNumber());
       existingCluster.add(comparedTweet);
     }//end if
}// end if technique is minhashing

else if(clusteringDetails.getTechnique()
  == Main.HAMMING)
  {
```

```
double similarity = 0;
if(clusteringDetails.getRepresentation()
== Main.FULL_TEXT||
clusteringDetails.getRepresentation()
== Main.FILTERED_PUNCTUATION_TEXT||
clusteringDetails.getRepresentation()
== Main.FILTERED_STOP_STEM_TEXT)
{
   comparingTokens
     = TextUtils.getTokens(comparingTweetText);
  comparedTokens
    = TextUtils.getTokens(comparedTweetText);
  comparingMinHashFromTokens
    = MinHashWrapper.getMinHash(comparingTokens);
  comparedMinHashFromTokens
    = MinHashWrapper.getMinHash(comparedTokens);
  comparingBitSet
    = ArrayUtils.createBitVector(
      comparingMinHashFromTokens);
  comparedBitSet
    = ArrayUtils.createBitVector(
      comparedMinHashFromTokens);
  similarity =
      ArrayUtils.getBitSimilarity(comparingBitSet,
      comparedBitSet,
      comparingMinHashFromTokens.length);
}
else if(clusteringDetails.getRepresentation()
 == Main.FULL_SHINGLES||
 clusteringDetails.getRepresentation()
 == Main.FILTERED_PUNCTUATION_SHINGLES||
clusteringDetails.getRepresentation()
== Main.FILTERED_STOP_STEM_SHINGLES)
{
```

```
    comparingShinglesSet
      = TextUtils.getShinglesSet(comparingTweetText,
      3, 3);
    comparedShinglesSet
      = TextUtils.getShinglesSet(comparedTweetText,
      3, 3);
    comparingMinHashFromShingles
      = MinHashWrapper.getMinHash(
        comparingShinglesSet);
    comparedMinHashFromShingles
      = MinHashWrapper.getMinHash(
        comparedShinglesSet);
    comparingBitSet
      = ArrayUtils.createBitVector(
        comparingMinHashFromShingles);
    comparedBitSet
      = ArrayUtils.createBitVector(
        comparedMinHashFromShingles);
    similarity
      = ArrayUtils.getBitSimilarity(comparingBitSet,
       comparedBitSet,
       comparingMinHashFromShingles.length);
  }
  if(similarity
    >= clusteringDetails.getSimilarityThreashold())
  {
    tweetsFound++;
    comparedTweet.setClusterNumber(
     comparingTweet.getClusterNumber());
    comparedTweet.setProcessed(true);
    ArrayList<Tweet> existingCluster =
     clusteredTweets.get(
     comparingTweet.getClusterNumber());
      existingCluster.add(comparedTweet);
```

```
            }//end if
          }// end if technique is hamming
          }
        }//end for that goes through compared tweet
        if(nosClusterTweets.containsKey(tweetsFound))
        {
          int nosClusters =
            nosClusterTweets.get(tweetsFound);
          nosClusters++;
          nosClusterTweets.put(tweetsFound,nosClusters);
        }
        else
        {

          nosClusterTweets.put(tweetsFound,1);
        }
    } //end for that goes through comparing tweets
    System.out.println("*****Near duplicates clustered”+
      “successfully...");
    long endTime   = System.nanoTime();
    double totalTime =
      (double)(endTime - startTime)/1000000000;
    String timeSpent =
      "\n"+ Main.currentDataSize + "\t"+ totalTime;
    saveClusters(clusteredTweets, nosClusterTweets,timeSpent);

} //end function that generates cluster
public void saveClusters(HashMap<Integer, ArrayList<Tweet>>
  clusteredTweets,HashMap<Integer, Integer>
  nosClusterTweets,String timeSpent)
{

    System.out.println("\n*****Saving clusters to text”+
      ” file...");
```

```
    String filenamePrefixCluster =
      clusteringDetails.getTechnique() +"_" +
      clusteringDetails.getMethodology() + "_"+
      clusteringDetails.getRepresentation() + "_" +
      clusteringDetails.getSimilarityThreashold() + "_" +
      clusteringDetails.getDataLimit();

String filenamePrefixTime
      = clusteringDetails.getTechnique() +"_" +
        clusteringDetails.getMethodology() + "_"+
        clusteringDetails.getRepresentation() + "_" +
        clusteringDetails.getSimilarityThreashold();

    File fileClusters =
      new File(filenamePrefixCluster+"-clusters.txt");
    File fileNosClusters =
      new File(filenamePrefixCluster+
    "-nosclusters_nostweets.txt");
    FileWriter fwClusters;
    BufferedWriter bwClusters = null;
    FileWriter fwNosClusters;
    BufferedWriter bwNosClusters = null;
    try
    {
       if (!fileClusters.exists())
          fileClusters .createNewFile();
       if (!fileNosClusters.exists())

          fileNosClusters .createNewFile();
       String headerClusters =
         "#Cluster Number \tCluster";
       String headerNosClusters =
         "#Number of Clusters \tNumber of Tweets";
       fwClusters =
         new FileWriter(fileClusters.getAbsoluteFile());
```

```
bwClusters =
  new BufferedWriter(fwClusters);
fwNosClusters =
  new FileWriter(fileNosClusters.getAbsoluteFile());
bwNosClusters =
  new BufferedWriter(fwNosClusters);
bwClusters.write(headerClusters);
bwNosClusters.write(headerNosClusters);
//loop through hashmap of clusters and write to file
for (Map.Entry<Integer, ArrayList<Tweet>> entry :
  clusteredTweets.entrySet()) {
   int clusterNos = entry.getKey();
   //loop through tweets for this cluster
   ArrayList<Tweet> tweets =
     clusteredTweets.get(clusterNos);
   String tweetIds = "";
   for (Tweet tweet : tweets)
   {

     tweetIds += tweet.getTweetId() + ",";
   }
   String lineClusters=
     "\n"+clusterNos + "\t"+ tweetIds;
   bwClusters.write(lineClusters);
}
for (Map.Entry<Integer, Integer> entry :
   nosClusterTweets.entrySet())
{
   int  nosTweets = entry.getKey();
   //loop through tweets for this cluster
   int nosClusters = nosClusterTweets.get(nosTweets);
   String lineNosClusters=
     "\n"+nosClusters+ "\t"+ nosTweets;
   bwNosClusters.write(lineNosClusters);
```

```
        }
        bwClusters.close();
        bwNosClusters.close();
        System.out.println("*****Clusters written to text”+
          ” files successfully...");
    }
    catch (IOException e) {
        e.printStackTrace();
    }
} //end saveClusters method
} //end class ClusterMaker
```

C.2 Multi-Index Locality Sensitive Hashing (LSH)

```
package ie.ucd.nearduplicates.techniques;
import ie.ucd.nearduplicates.main.ClusteringDetails;
import ie.ucd.nearduplicates.main.Main;
import ie.ucd.nearduplicates.utils.TextUtils;
import ie.ucd.nearduplicates.utils.Tweet;
import java.io.BufferedWriter;
import java.io.File;
import java.io.FileWriter;
import java.io.IOException;
import java.util.ArrayList;
import java.util.BitSet;
import java.util.HashMap;
import java.util.Iterator;
import java.util.Map;
import java.util.Set;
public class LSH
{

    private HashMap<BitSet, ArrayList<Tweet>> tweets;
```

```
    private int N; //number of hashes
    private int K; //k bands for lsh
    int R;//radius
    private ClusteringDetails clusteringDetails;

private ArrayList<String> comparingShinglesList;

    private Set<String> comparingShinglesSet;
    private Set<String> comparingTokens;
    private int[] comparingMinHashFromTokens;
    private int[] comparingMinHashFromShingles;
    private ArrayList<String> comparedShinglesList;
    private Set<String> comparedShinglesSet;
    private Set<String> comparedTokens;
    private int[] comparedMinHashFromTokens;
    private int[] comparedMinHashFromShingles;
    /*this specifies how many bits can be different
      *between similar bit vector chunks*/
    int chunkRadius;
    public LSH(HashMap<BitSet, ArrayList<Tweet>> tweets,
      int N, int K, int R,
      ClusteringDetails clusteringDetails)
    {
       this.tweets = tweets;
       this.K = K;
       this.R = R;
       this.clusteringDetails = clusteringDetails;
       this.N = N;
       chunkRadius = (int)Math.floor(R/K);

}


    public static int getHammingDistance(BitSet bitSetA,
      BitSet bitSetB)
    {
       int distance = 0;
```

```
   BitSet clonedBitSetA =  (BitSet)bitSetA.clone();
   clonedBitSetA.xor(bitSetB);
   for(int i=0;i<clonedBitSetA.size();i++)
   {

      Boolean bitValue = clonedBitSetA.get(i);
      if(bitValue)
      {
        distance++;
      }
}
return distance;
}
public static double getBitSimilarity(BitSet bitSetA,
  BitSet bitSetB, int N)
{

   int distance = getHammingDistance(bitSetA, bitSetB);
   double similarity = 1 - ((double)distance/N);
return similarity;
}
//creating the LSH Table data structure
public ArrayList<HashMap<BitSet,ArrayList<BitSet>>>
  createLSHInitMap()
{

   ArrayList<HashMap<BitSet,ArrayList<BitSet>>>
     lshBuckets =
      new ArrayList<HashMap<BitSet,ArrayList<BitSet>>>();
   //create the k hashmaps
   for(int i=0;i<K;i++)
   {

      HashMap<BitSet,ArrayList<BitSet>> band =
```

```
        new HashMap<BitSet,ArrayList<BitSet>>();
      lshBuckets.add(band);
   }
   //dividing the bitsets into chunks
   int x =1;
   for (Map.Entry<BitSet, ArrayList<Tweet>> tweetEntry :
      tweets.entrySet())
   {

      BitSet bitSet = tweetEntry.getKey();
      ArrayList<Tweet> comparingTweetList =
      tweets.get(bitSet);
      //split the bit set into K chunks
      int chunkNos = 0;
      int chunkSize = N/K; //i
      int index2 = chunkSize;
      for(int i=0;i<N;i+=chunkSize)
      {

        BitSet bitSetChunk = bitSet.get(i, index2);
HashMap<BitSet,ArrayList<BitSet>> currentBand
          = lshBuckets.get(chunkNos);
        if (currentBand.containsKey(bitSetChunk)) {

  ArrayList<BitSet> bitSetList = currentBand.get(bitSetChunk);
           bitSetList.add(bitSet);
           currentBand.put(bitSetChunk, bitSetList);
        }
        else
        {

       ArrayList<BitSet> bitSetList =
           new ArrayList<BitSet>();
         bitSetList.add(bitSet);
```

```
            currentBand.put(bitSetChunk, bitSetList);
          }
        chunkNos++;
        index2 += chunkSize;
        }
    }
    return lshBuckets;
}//end method
public HashMap<Integer,ArrayList<BitSet>>
  createLSHBuckets(ArrayList<HashMap<BitSet,
  ArrayList<BitSet>>> lshInitList )
{

    HashMap<Integer,ArrayList<BitSet>> lshBuckets = new HashMap<Int
    ArrayList<BitSet> processedBitSets =
      new ArrayList<BitSet>();
    int lastBucketNos = 0;
    for (Map.Entry<BitSet, ArrayList<Tweet>> tweetEntry :
      tweets.entrySet())
    {

        //Map.Entry<String, Tweet> comparedEntry =
          comparedIt.next();
        BitSet bitSet = tweetEntry.getKey();
        if(processedBitSets.contains(bitSet))
          continue;
        ArrayList<BitSet> initBucket =
          new ArrayList<BitSet>();
        initBucket.add(bitSet);
        lshBuckets.put(lastBucketNos,initBucket);
        //add this BitSet to the list of the processed
        processedBitSets.add(bitSet);
        //split the bit set into K chunks
        int chunkNos = 0;
```

```
int chunkSize = N/K;
int index2 = chunkSize;
for(int i=0;i<N;i+=chunkSize)
{

  BitSet comparingBitSetChunk = bitSet.get(i, index2);
  index2 += chunkSize;
  HashMap<BitSet,ArrayList<BitSet>> currentBand =
    lshInitList.get(chunkNos);

  for (Map.Entry<BitSet,ArrayList<BitSet>>
    currentBandEntry : currentBand.entrySet()) {

    BitSet comparedBitSetChunk =
      currentBandEntry.getKey();
    ArrayList<BitSet> bitSetList =
      currentBandEntry.getValue();
    BitSet clonedComparedBitSet =
      (BitSet)comparedBitSetChunk.clone();
    clonedComparedBitSet
      .xor(comparingBitSetChunk);
    int differentBits = 0;
    for(int j=0;
      j<clonedComparedBitSet.length();j++)
    {
      Boolean bitValue =
        clonedComparedBitSet.get(j);
      if(bitValue)
      {
        differentBits++;
      }
      if(differentBits > chunkRadius)
       break;
      if(differentBits <= chunkRadius)
```

```
        {
          ArrayList<BitSet> checkedBitSetList =
            new ArrayList<BitSet>();
          for(BitSet checkedBitSet: bitSetList)
          {
            if(!processedBitSets.
              contains(checkedBitSet))
            {
              processedBitSets.add(checkedBitSet);
              checkedBitSetList.add(checkedBitSet);
            }
          }

        if(lshBuckets.containsKey(lastBucketNos))
            {
              ArrayList<BitSet> bucket =
                lshBuckets.get(lastBucketNos);
              bucket.addAll(checkedBitSetList);
            }
            else
            {
             lshBuckets.put(lastBucketNos,
               checkedBitSetList);
            }
          }
        }
        chunkNos++;
      } //end for
      lastBucketNos++;
    }
    return lshBuckets;
}//end method
public void generateClustersFromLSH(
  HashMap<Integer, ArrayList<BitSet>> lshBuckets)
```

```
{

   long startTime = System.nanoTime();
   System.out.println("\n*****Clustering near "+
   "duplicates via LSH...");
   int lastClusterNumber = 0;
   int tweetsFound;
   HashMap<Integer, ArrayList<Tweet>>
     clusteredTweets = new HashMap<Integer,
     ArrayList<Tweet>>();
   HashMap<Integer, Integer> nosClusterTweets=
     new HashMap<Integer, Integer>();
   ArrayList<BitSet> processedBitSetList =
     new ArrayList<BitSet>();
   for(Iterator<Map.Entry<Integer,
     ArrayList<BitSet>>> it =
     lshBuckets.entrySet().iterator();
     it.hasNext(); )
   {

     Map.Entry<Integer, ArrayList<BitSet>> entry =
       it.next();
     ArrayList<BitSet> comparingBitSetList =
       entry.getValue();
     ArrayList<BitSet> comparedBitSetList =
       comparingBitSetList;
     tweetsFound = 0;
     for(int i=0;
       i<comparingBitSetList.size();i++)
     {
       BitSet comparingBitSet =
         comparingBitSetList.get(i);
       if(processedBitSetList.
         contains(comparingBitSet))
```

```
        continue;
      else
      {
        ArrayList<Tweet> tweetsForBitSet
          = tweets.get(comparingBitSet);
        ArrayList<Tweet> newCluster
          = new ArrayList<Tweet>();
        newCluster.addAll(tweetsForBitSet);
        tweetsFound = 1;
        lastClusterNumber++;
        tweetsFound = tweetsForBitSet.size();
        clusteredTweets.put(lastClusterNumber, newCluster);

        for(int j=0;j<comparedBitSetList.size();j++)
        {
          BitSet comparedBitSet = comparedBitSetList.get(j);
          if(processedBitSetList.contains(comparedBitSet))
            continue;
          else if(i==j)
            continue;
          else
          {
            ArrayList<Tweet> comparingTweetList =
              tweets.get(comparingBitSet);
            ArrayList<Tweet> comparedTweetList =
              tweets.get(comparedBitSet);
            Tweet comparingTweet =
              comparingTweetList.get(0);
            Tweet comparedTweet =
              comparedTweetList.get(0);

//set the number of tweets found double similarity = 0;

 if(clusteringDetails.getTechnique()
```

```
    == Main.HAMMING)
   {
    similarity = getBitSimilarity(comparingBitSet,
      comparedBitSet);
   }
  else if(clusteringDetails.getTechnique()
    == Main.LEVENSHTEIN)
  {
   if(clusteringDetails.getRepresentation() ==
     Main.FULL_TEXT||
     clusteringDetails.getRepresentation() ==
     Main.FILTERED_PUNCTUATION_TEXT||
     clusteringDetails.getRepresentation() ==
     Main.FILTERED_STOP_STEM_TEXT)
   {
     similarity =
       Levenshtein.getLevenshteinSimText(
       comparingTweet.getText(),
       comparedTweet.getText());
   }
 else if(clusteringDetails.getRepresentation()
  ==Main.FULL_SHINGLES||
  clusteringDetails.getRepresentation() ==
  Main.FILTERED_PUNCTUATION_SHINGLES||
  clusteringDetails.getRepresentation() ==
  Main.FILTERED_STOP_STEM_SHINGLES)
 {
   comparingShinglesList =
     TextUtils.getShinglesList(
     comparingTweet.getText(),3, 3);
   comparedShinglesList =
     TextUtils.getShinglesList(
     comparedTweet.getText(),3, 3);
   similarity =
```

```
            Levenshtein.getLevenshteinSimShingles(
            comparingShinglesList,
            comparedShinglesList);
      }
    }//end if technique is levenshtein
    else if(clusteringDetails.getTechnique()
      == Main.JACCARDS)
    {
      if(clusteringDetails.getRepresentation()
        == Main.FULL_TEXT||
        clusteringDetails.getRepresentation() ==
        Main.FILTERED_PUNCTUATION_TEXT||
        clusteringDetails.getRepresentation() ==
        Main.FILTERED_STOP_STEM_TEXT)
      {
        comparingTokens =
          TextUtils.getTokens(
          comparingTweet.getText());
        comparedTokens =
          TextUtils.getTokens(
          comparedTweet.getText());
        similarity =
          Jaccard.getJaccardSimilarity(
          comparingTokens, comparedTokens);
      }
      else if(clusteringDetails.getRepresentation() ==
        Main.FULL_SHINGLES||
        clusteringDetails.getRepresentation() ==
        Main.FILTERED_PUNCTUATION_SHINGLES||

    clusteringDetails.getRepresentation() ==
       Main.FILTERED_STOP_STEM_SHINGLES)
    {
      comparingShinglesSet =
```

```
      TextUtils.getShinglesSet(
      comparingTweet.getText(),3, 3);
    comparedShinglesSet =
      TextUtils.getShinglesSet(
      comparedTweet.getText(),3, 3);
    similarity =
      Jaccard.getJaccardSimilarity(
      comparingShinglesSet,comparedShinglesSet);
  }
 }//end if technique is jaccards

else if(clusteringDetails.getTechnique() ==
  Main.MINHASHING)
{
   if(clusteringDetails.getRepresentation() ==
     Main.FULL_TEXT||
     clusteringDetails.getRepresentation() ==
     Main.FILTERED_PUNCTUATION_TEXT||
     clusteringDetails.getRepresentation() ==
     Main.FILTERED_STOP_STEM_TEXT)
   {
     comparingTokens =
       TextUtils.getTokens(
         comparingTweet.getText());
     comparedTokens =
         TextUtils.getTokens(
          comparedTweet.getText());
     comparingMinHashFromTokens
         = MinHashWrapper.getMinHash(
           comparingTokens);
     comparedMinHashFromTokens
         = MinHashWrapper.getMinHash(
          comparedTokens);
     similarity =
```

```
            MinHashWrapper.measureSimilarity(
            comparingMinHashFromTokens,
             comparedMinHashFromTokens);
    }
    else if(clusteringDetails.getRepresentation() ==
      Main.FULL_SHINGLES||
      clusteringDetails.getRepresentation() ==
      Main.FILTERED_PUNCTUATION_SHINGLES||
      clusteringDetails.getRepresentation() ==
      Main.FILTERED_STOP_STEM_SHINGLES)
    {
      comparingShinglesSet =
        TextUtils.getShinglesSet(
        comparingTweet.getText(),3, 3);
      comparedShinglesSet =
        TextUtils.getShinglesSet(
        comparedTweet.getText(),3, 3);
      comparingMinHashFromShingles =
        MinHashWrapper.getMinHash(
        comparingShinglesSet);
      comparedMinHashFromShingles =
        MinHashWrapper.getMinHash(
        comparedShinglesSet);
      similarity =
        MinHashWrapper.measureSimilarity(
        comparingMinHashFromShingles,
        comparedMinHashFromShingles);
    }
  } //end if technique is minhashing
  if(similarity >=
    clusteringDetails.getSimilarityThreashold())
  {
    tweetsFound += comparedTweetList.size();
    newCluster.addAll(comparedTweetList);
```

```
                clusteredTweets.put(lastClusterNumber,
                  newCluster);
                processedBitSetList.add(comparedBitSet);
              }
             }
            }
        }

        if(nosClusterTweets.containsKey(tweetsFound))
        {

           int nosClusters = nosClusterTweets.get(tweetsFound);
           nosClusters++;
           nosClusterTweets.put(tweetsFound,nosClusters);
        }
        else
        {

           nosClusterTweets.put(tweetsFound,1);
        }
    }
}


System.out.println("*****Near duplicates clustered via”+
  ” LSH successfully...");
long endTime   = System.nanoTime();
double totalTime =
  (double)(endTime - startTime)/1000000000;
String timeSpent =
  "\n"+ Main.currentDataSize + "\t"+ totalTime;
saveClusters(clusteredTweets, nosClusterTweets,timeSpent);
} //end method
```

```
public void saveClusters(HashMap<Integer,
  ArrayList<Tweet>> clusteredTweets,
  HashMap<Integer, Integer> nosClusterTweets,
  String timeSpent)
{
    /*implementation not shown here since it is
    similar to Appendix C.1*/
}
}
```

References

[1] The Harvard College Library. http://hcl.harvard.edu/libraries/houghton/ collections/widener/gutenberg.cfm

[2] M. S. Charikar. Similarity Estimation Techniques from Rounding Algorithms. In Proc. of the 34th Annual Symposium on Theory of Computing, pages 380-388, 2002.

[3] Mayur Datar, Nicole Immorlica, Piotr Indyk, and Vahab Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In SCG '04: Proceedings of the twentieth annual symposium on Computational geometry, pages 253–262, New York, NY, USA. ACM, 2004.

[4] Piotr Indyk and Rajeev Motwani. Approximate nearest neighbors: towards removing the curse of dimensionality. In STOC '98: Proceedings of the thirtieth annual ACM symposium on Theory of computing, pages 604–613, New York, NY, USA. ACM, 1998.

[5] Roberto J. Bayardo, Yiming Ma,and Ramakrishnan Srikant. Scaling Up All Pairs Similarity Search. In Proc. of the 16th Int'l Conf. on World Wide Web, Banff, Alberta, Canada, 131-140, 2007.

[6] A. Z. Broder, S. C. Glassman, M. S. Manasse, & G. Zweig. Syntactic clustering of the Web. In Proc. of the 6th Int'l World Wide Web Conference, 391-303, 1997.

[7] Sasa Petrovic, Miles Osborne, and Victor Lavrenko. Streaming First Story Detection with Application to Twitter. In Proc. of the the 10th Annual Conference of the North American Chapter of the Association for Computational Linguistics, pages 181-189, 2010.

[8] Ke Tao et. al. Groundhog Day: Near-Duplicate Detection on Twitter. In Proc. of the 22nd international conference on World Wide Web, pages 1273-1284, 2013.

[9] Akshay Java, Xiaodan Song, Tim Finin, and Belle Tseng. Why we twitter: understanding microblogging usage and communities. In WebKDD/SNA-

KDD '07: Proceedings of the 9th WebKDD and 1st SNA-KDD 2007 workshop on Web mining and social network analysis, pages 56–65, New York, NY, USA. ACM, 2007.

[10] Balachander Krishnamurthy, Phillipa Gill, and Martin Arlitt. A few chirps about twitter. In WOSP '08: Proceedings of the first workshop on Online social networks, pages 19–24, New York, NY, USA. ACM, 2008.

[11] Gang Luo, Chunqiang Tang, and Philip S. Yu. Resource-adaptive real-time new event detection. In SIGMOD '07: Proceedings of the 2007 ACM SIGMOD international conference on Management of data, pages 497–508, New York, NY, USA. ACM, 2007.

[12] Martin Theobald, Jonathan Siddharth, and Andreas Paepcke. SpotSigs: robust and efficient near duplicate detection in large web collections. In Proc. of the 31st annual international ACM SIGIR conference on Research and development in information retrieval, pages 563-570, 2008.

[13] Anand Rajaraman and Jeffrey Ullman. Mining of Massive Datasets. ISBN:1107015359 9781107015357. Cambridge University Press New York, NY, USA, pages 71-124, 2011.

[14] Sean E. Anderson, Bit Twiddling Hacks. https://graphics.stanford.edu/~seander/bithacks.html, 2014.

[15] Jay Chan. Multi-Index Locality Sensitive Hashing for Fun and Profit. https://engineering.eventbrite.com/multi-index-locality-sensitive-hashing-for-fun-and-profit/, 2014.

[16] Herstein, I. N. Topics In Algebra. ISBN 978-1114541016. Waltham: Blaisdell Publishing Company, page 90, 1964

[17] Ali Punjani. Fast Search in Hamming Space with Multi-index Hashing. In Proc. of the 2012 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), pages 3108-3115, 2012.

[18] Vladimir Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals vladimir levenshtein. In Soviet Physics Doklady, pages 707-710, 1966.

[19] Robert A. Wagner and Michael J. Fischer. The String-to-String Correction Problem. In the Journal of the ACM, volume 21, issue 1, pages 168-173, 1974.

[20] Esko Ukkonen. On Approximate String Matching. In proc. of the 1983 International FCT-Conference on Fundamentals of Computation Theory, pages 487-495, 1983.

[21] Apache Commons Lang. http://commons.apache.org/proper/commons-lang/, 2014

[22] G. Miller et al. WordNet: A Lexical Database for English. Communications of the ACM, 38(11), pages 39 to 41, 1995.

[23] Minhash Java code by Thomas Jungblot. https://github.com/thomasjungblut/thomasjungblut-common/blob/master/src/de/jungblut/nlp/MinHash.java