The Similarity Index

M. Scott Roth

July 1, 2011

Abstract

Is it possible to calculate a hash value for a document that captures its salient characteristics, such that a repository can be queried for like values and retrieve all "similar" documents? If so, similar documents could be easily identified by a simple SQL query without the need for a full text search engine. Such a value would allow systems to quickly identify duplicate or similar content before it is checked into a repository, introduced to an index, or returned in a query result. Additionally, this value could assist with identifying other content a user might be interested in, though they did not explicitly query for it.

This paper endeavors to answer this question by exploring the corpus of existing research in this and related areas, and reporting the results of experimentation. This investigation was conducted with the intent of implementing such a solution in a Documentum environment.

1 Introduction

Recently, an intriguing question was raised on the EDN. The gist of the question was, during the checkin of a document, is there a quick and easy way to determine if a similar document already exists in the repository?[18] If the question had been, is there a way to identify exact duplicate documents, the answer would be "yes". This could be accomplished using a cryptographic hash such as MD5 or SHA-1 [5] [8]. However, because the questioner was interested in "similar" documents, cryptographic hashes were not the solution, since cryptographic hash functions are designed to detect the slightest perturbation of the content [16]. This notion of identifying similar documents sparked an idea in my mind.

The ability to find similar documents exists in the Lucene `MoreLikeThis` API [7], but would require some customization to make it work in Webtop and xPlore. The `MoreLikeThis` API essentially extracts salient words from the current document and uses them to construct a full text query against the Lucene indexes. I was more interested in determining if it was possible to calculate a number that would represent the primary characteristics of a document. If it was, I could query a repository for all documents within +/- 30%[1] of a known value and call those documents "similar". The use of such a number would allow the detection of similar documents in repositories not using xPlore (or Lucene), and could easily be implemented with an Aspect. This value would be the product of a different kind of hash, one where slight differences in content were ignored. I call this value the similarity index, SI.

2 Background

I discovered that there is a vast field of academic research related to the idea of detecting near-duplicate documents, for example: [1] [3] [5] [6] [9] [10] [17] [18]. Closely related to this field of research is the notion of document clustering (putting similar documents in the same indexing bin) [14] and content analytics (generating statistics about documents to improve search and retrieval) [22]. It's pretty heady stuff but fun to read. I included a select bibliography at the end of this paper if you are interested.

Most of the cited research was conducted with Internet search engines in mind (e.g., AltaVista, Google) to reduce redundant results and keep near-duplicate documents out of the indexes. In these scenarios, a collection of numeric *fingerprints* of a document (called a *sketch*) was stored in a database and similarity was determined by how many fingerprints two documents shared. The more fingerprints two documents shared, the more similar they were deemed to be. This idea and basic process has applicability beyond just Internet search engines. These same concepts and processes have been employed to detect duplicated, copied, and refactored source code [10] [11], plagiarism in academic papers [13], copyright infringement [13], articles derived from the same source material [5], and even to suggest products consumers might be interested in based upon the contents of their electronic shopping carts [5].

[1] This is a fairly arbitrary value, but one that seems to work well; some authors declared documents similar if they contained only 50% [9] common fingerprints, others 90% [6].

My idea was to take this concept one step further and reduce the sketch to a single value, the Similarity Index (SI), thus eliminating the need for a database to store the sketch. If the sketch could be reduced to a single value, this value could easily be stored as a property of a document or as a single-valued attribute in Documentum. The idea of reducing the sketch to a single value was discussed and implemented in passing by [9] as a way to quickly determine the accuracy with which their methodology detected duplicate or near-duplicate documents.

One thing to note is what I (and the other authors [1] [5] [9] [17]) mean by *similar documents*. Similar documents are those which share essentially the same words in the same sequence (syntactic similarity). These documents do not necessarily share similar meanings. Detection of this sort would require the analysis of each word, its meaning, sentence constructs, etc. For the purposes of this paper, the format of documents is inconsequential; the experiment operates on only canonical (i.e., text) forms of documents.

The rest of this paper will discuss the experiment I conducted, the methods I used to produce the Similarity Index, the results of the experiment, and the utility of the computed value in identifying similar documents.

3 The Experiment

To conduct the experiment, I wrote a Java class that implemented the process depicted in Figure 1 and used several freely-available libraries to implement some of the hairier mathematics and encoding [19] [20]. The class implemented fourteen different methods for producing the SI value. These methods are discussed in Section 3.3. In addition to these different methods, I used three different input streams: characters, words and Soundex [15] tokens. These input streams are discussed in Section 3.4 and 3.5. Finally, I assembled a collection of test documents that I expected to give a variety of results. These documents are discussed in Section 3.2.

3.1 Overview

There are various techniques for analyzing content for similarity [5] [12] [17]. Most of these techniques begin by slicing and dicing content into *k-grams* to produce *shingles*, *fingerprints*, and *sketches* (see Figure 1). A k-gram is an overlapping sequence of characters or words of *k* length. Shingles are numeric representations of the k-grams produced by hashing the k-grams. Some researches [9] simply used the Java *String.hashCode()* method to produce the hash for each k-gram, while others employed a Rabin hash [2]. Creating shingles produces roughly as many shingles as there are characters or words in the original content. To be efficient, this set must be reduced by a filtering function to a set of fingerprints (i.e., hashed shingle values that represent the salient features of the content). Various methods are proposed for conducting this set reduction including: keeping only those shingles evenly divisible by 25 [1], using a Winnowing function [3] [4], or keeping only the 400 highest valued shingles [3]. The filtered collection of fingerprints is called a sketch.

For purposes of this explanation, assume the incoming document stream is the first sentence from Lincoln’s Gettysburg Address:

Four score and seven years ago our fathers brought forth on this continent, a new nation, conceived in Liberty, and dedicated to the proposition that all men are created equal.

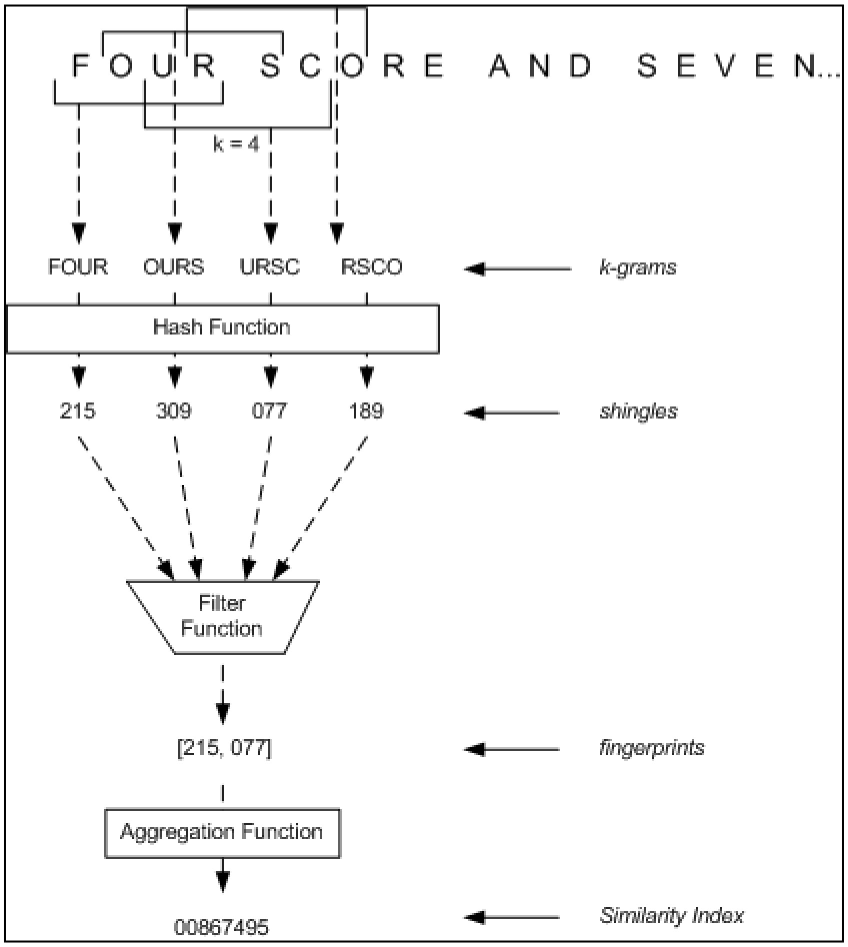

Figure 1 k-grams, Shingles, Fingerprints and Sketches

3.1.1 k-grams of Letters and Words

k-grams were constructed by reading either the incoming character or word streams and simply appending successive characters (words) to a String until it equaled *k* length. The length, *k*, was controlled by a process parameter (see Section 3.6) and varied across runs of the experiment.

In Figure 1 , the first nine character k-grams (*k* = 4) for the beginning of Lincoln's Gettysburg Address are:

```
{ (four), (ours), (ursc), (rsco), (scor),
      (core), (orea), (rean), (eand) }
```

The first four word k-grams (*k*=4) look like this:

```
{ (four score and seven), (score and seven years),
  (and seven years ago), (seven years ago our) }
```

This overlapping (i.e., shingle effect) provides some insulation against inserted and deleted characters (words) in the text. Because the shingles are generated from a sliding window of *k* length, eventually any inserted or deleted letters will be overcome, and the shingles produced will resemble those of the original document. A good filtering algorithm will pick up the same shingles from both an original and a near duplicate document, thus eliminating the difference the inserted or deleted characters (words) made.

3.1.2 Hash Function

The purpose of the hash function was to encode each shingle into a unique fingerprint. I used Java's *String.hashCode()* method to hash k-grams into fingerprints. [9] submits that this hash is sufficiently collision-resistant for this purpose. More details about Java's *String.hashCode()* method can be found in [21].

3.1.3 Filter Function

The filter function proved to be a key aspect of the process. As mentioned previously, the filter function determined which fingerprints were included in the sketch and which were not. The crux of the function was that it must be selective enough to retain the major characteristics of a document without creating a sketch that was so liberal there was no differentiation among documents. Filtering techniques are discussed in more detail in Section 3.3.

3.1.4 Aggregation Function

The aggregation function endeavored to reduce the sketch produced by the filter function to a single number. Like the filter function discussed in Section 3.1.3, the aggregation function needed to produce a value that represented the character of the sketch without being too generic or restrictive. I took three approaches to aggregation. The first was to simply sum the fingerprints in the sketch [9]. The second was to recursively produce k-grams from the fingerprints in the sketch until only one k-gram remained. This has been referred to as a *super shingle* by [9] [14] [23]. The third approach was to produce a Rabin hash of the sketch.

3.1.5 Similarity Index

The Similarity Index was the final result of the experiment. Its goal, as stated earlier, was to represent the salient features of a document as a single value. With the SI, it was possible to determine if two documents were similar by comparing the difference between their SI values. If the difference between the values was < 30%, the documents were deemed similar.

3.2 The Document Collection

Table 1 describes the document collection used in the experiment.

Table 1 Document Collection

File Name	Chars	Words	Remarks
`AddToClipboardAction.java`	3,632	213	The source code for a Java class file
`DeleteDocument.class`	18,468	2,711	A compiled Java class file
`Gettysburg.txt`	1,473	270	The text to Lincoln's Gettysburg Address
`Proverbs.txt`	91,384	19,157	The book of Proverbs from the Bible

File Name	Chars	Words	Remarks
`Proverbs1-16.txt`	42,069	8,376	The first 16 chapters of the book of Proverbs -- I expected this file to have some similarity with `Proverbs.txt` since this text represents a subset of the whole, and should be a good test for detecting deletions at the end of a file.
`Proverbs1-24.txt`	69,016	13,736	The first 24 chapters of the book of Proverbs -- I expected this file to have strong similarity with `Proverbs.txt` since this text represents a subset of the whole. I also expected it to have some similarity with `Proverbs1-16.txt` since it contains those chapters also.
`Proverbs25-31.txt`	22,368	4,421	The last seven chapters of the book of Proverbs -- I expected this file to have low similarity with `Proverbs.txt` since this text represents a subset (25%) of the whole. I expected it to not be similar to `Proverbs1-16.txt` or `Proverbs1-24.txt` since their contents are disjoint.
`TargetSetup.Result.txt`	3,115	347	A log file
`usConstitution(copy).txt`	48,067	7,669	An exact copy of `usConstitution.txt` -- I expected this file to have perfect similarity with `usConstitution.txt`.
`usConstitution-BillofRights.txt`	32,006	5,039	The text of the US Constitution without the Bill of Rights or any subsequent amendments -- I expected this file to have some similarity with the `usConstitution.txt` since this file represents a subset of the whole.
`usConstitution-noPreamble.txt`	47,728	7,617	The text of the US Constitution without the Preamble -- I expected this file to have strong similarity with the `usConstitution.txt` since this file represents a subset of the whole. It should be a good test for detecting deletions at the beginning of a file.
`usConstitution.txt`	48,067	7,669	The text of the US Constitution -- I expected this file to have perfect similarity with `usConstitution(copy).txt` and some similarity with `usConstitution-BillofRights.txt` and `usConstitution-noPreamble.txt` since those files represented subsets of this file's content.
`usDOI-1.txt`	8,147	1,337	The text of the US Declaration of Independence less one sentence near the middle of the file -- I expected this file to have strong similarity with `usDOI.txt` since they were nearly identical, and some similarity with `usDOI-Grievances.txt` since that file was a subset of this one. This should also be a good test for detecting a small deletion in the middle of the document.
`usDOI-Grievances.txt`	4,093	682	The text of the US Declaration of Independence less the grievances against the King of England -- I expected this file to have low similarity with `usDOI.txt` and `usDOI-1.txt` since there is a 50% difference in their contents.

File Name	Chars	Words	Remarks
`usDOI.txt`	8,179	1,341	The text of the US Declaration of Independence -- I expected this file to have strong similarity with `usDOI-1.txt` since they were nearly identical and some similarity with `usDOI-Grieveances.txt` since that file contained a subset of this file's content.

3.3 Similarity Index Methodologies

This section briefly describes each methodology used to calculate the SI. These methodologies were further affected by the type of input stream (Section 3.4) and variations in processing the input stream (Section 3.5).

1. Sum of Java *hashCode()* fingerprints – Used Java *String.hashCode()*[2] to generate hashes of k-grams and summed all of the hashes [9].
2. Rabin hash fingerprints - Used Java *String.hashCode()* to generate hashes of k-grams and produced a Rabin hash of the entire set of fingerprints.
3. Sum 0 mod 25 sketch – Filtered fingerprints that were 0 mod 25 (i.e., evenly divisible by 25) to produce a sketch and summed the sketch [9] [23].
4. Rabin hash 0 mod 25 sketch – Created a Rabin hash of the 0 mod 25 sketch.
5. Sum of Winnowing sketch – Used a Winnowing algorithm [4] to produce the sketch. Summed all the fingerprints in the sketch.
6. Rabin hash of Winnowing sketch – Created a Rabin hash of the Winnowed sketch.
7. Sum low 400 sketch - Summed the 400 lowest value fingerprints. 400 was chosen to be similar to the process described in [9] [14].
8. Rabin hash of low 400 sketch – Created Rabin hash of the 400 lowest value fingerprints.
9. Sum high 400 sketch - As a variation on approach #7, summed the 400 highest value fingerprints.
10. Rabin hash high 400 sketch – Created Rabin hash of the 400 highest value fingerprints.
11. Sum random 1/3 sketch - Randomly created a sketch of 1/3 of the fingerprints and summed the sketch. This was not expected to provide good results since an inconsistent set of fingerprints is chosen from each document [13].
12. Rabin hash random 1/3 sketch – Created a Rabin hash of the random 1/3 sketch of the fingerprints.
13. Sum 200 most frequent words sketch – Created fingerprints of the 200 most frequently used words in each document and summed the sketch. This is similar to the Textract method used in [9].
14. Super shingle of all fingerprints – Reduce the entire set of fingerprints to a single value by repeatedly producing k-grams of the fingerprint set [9] [14].

3.4 Input Streams

Each method of generating the SI was tested using three different input streams:

[2] Unless explicitly stated, the Java *String.hashCode()* method was used to create fingerprints of k-grams.

1. The input was treated as a stream of characters.
2. The input was treated as a stream of words; all whitespace was removed except for one space between each word.
3. The input was treated as a stream of Soundex [15] tokens. All words were converted to Soundex tokens in an effort to introduce fuzziness into the content.

As each method listed in Section 3.3 was executed using a different input stream it was assigned a new method number to aid in tracking. The methods were numbered as follows:

```
tracking method number = method number + offset number
```

Table 2 Tracking Method Numbers.

Method Number	Tracking Offset	Input Stream	Tracking Method Number
1 – 14	0	Character-based stream	1 - 14
1 – 14	20	Word-based stream	21 – 34
1 – 14	40	Soundex token-based stream	41 - 54

Note: Method 13 (*Sum 200 most frequent words sketch*) does not have a character stream-based equivalent.

3.5 Input Stream Processing Variations

In addition to each input stream, each method was executed using four different input stream processing variations. The purpose for varying the processing on the input stream was to further distill the salient characteristics of each document.

3.5.1 Methods 1 – 14

Methods 1-14 used a character stream as inputs. Table 3 lists the input stream processing variations applied to each character stream.

Table 3 Input Stream Process Variations for Methods 1 – 14.

Process Variation	With Vowels	With Stop Words	Comments
A	Yes	Yes	This variation represented the unadulterated character stream.
B	Yes	No	All of the stop words were removed from the input stream of characters since they are words common to all files and provided no differentiation.
C	No	Yes	All vowels were removed from the input stream of characters since they were common to all content and provided no differentiation.
D	No	No	All stop words and vowels were removed from the input stream of characters. This represented the most unique version of the content.

3.5.2 Methods 21 – 34 and Methods 41 - 54

Methods 21 – 54 used word-based input streams. Table 4 lists the input stream process variations applied to each word stream.

Table 4 Input Stream Process Variations for Methods 21 – 54.

Process Variation	With Stop Words	With Duplicate Words	Comments
A	Yes	Yes	This variation represented the unadulterated word stream.
B	No	Yes	All stop words were removed from the input stream, but duplicate words were allowed to stay.
C	Yes	No	Stop words were allowed to stay in the input stream, but all duplicate words were removed.
D	No	No	All stop words and duplicate words were removed from the input stream. This represented the most unique version of the content.

3.6 Process Parameters

Each run of the experiment executed each of the methods using each input stream variation and input stream processing variation discussed in Sections 3.3, 3.4, 3.5 with the following process parameters:

- *k* – the size of the k-gram.
- *w* – the size of the sliding window used in the Winnowing process.

Four runs of the experiment were conducted with the following process parameters:

Table 5 k and w process parameters

Run	k	W
1	4	5
2	8	12
3	16	25
4	40	100

As discussed in [5] [6], a *k* that was too large resulted in unrelated documents having too much commonality (false positives), while a *k* that was too small exaggerated minor differences and resulted in similar documents having divergent fingerprints. [6] suggested that a *k* in the range of 3 – 10 would give the best results. Given this knowledge, I expected the run with *k* = 8 to produce the most accurate results.

4 Analysis

One of the most difficult parts of the experiment was determining how to score the results and determine whether a combination of methodology, input stream, input stream process variation, and process parameters produced a good SI.

Each run of the experiment produced a large comma separated value (CSV) file that was loaded into Excel for analysis. To begin, I build a set of matrices for each run that compared each document's SI to every other document's SI in the collection. I focused my attention on three *zones* in these matrices; namely where the documents I knew were similar were compared. Section 4.1 discusses these zones and their expected similarity.

4.1 Expected Results

Table 6 , Table 7 , and Table 8 depict what I expected the scores to reveal, expressed as similarity. The justifications for these expectations were discussed in Section 3.2.

Table 6 Zone 1 – Proverbs Collection

	Proverbs.txt	Proverbs1-16.txt	Proverbs1-24.txt	Proverbs25-31.txt
Proverbs.txt	Exact	Similar	Similar	Low Similarity
Proverbs1-16.txt	Similar	Exact	Similar	None
Proverbs1-24.txt	Similar	Similar	Exact	None
Proverbs25-31.txt	Low Similarity	None	None	Exact

Table 7 Zone 2 – Constitution Collection

	usConstitution (copy).txt	usConstitution-BillofRights.txt	usConstitution-noPreamble.txt	usConstitution.txt
usConstitution (copy).txt	Exact	Similar	Similar	Exact
usConstitution-BillofRights.txt	Similar	Exact	Similar	Medium
usConstitution-noPreamble.txt	Similar	Similar	Exact	Similar
usConstitution.txt	Exact	Similar	Similar	Exact

Table 8 Zone 3 – Declaration of Independence Collection

	usDOI-1.txt	usDOI-Grievances .txt	usDOI.txt
usDOI-1.txt	Exact	Low Similarity	Similar
usDOI-Grievances .txt	Low Similarity	Exact	Low Similarity
usDOI.txt	Similar	Low Similarity	Exact

4.2 Evaluation of Results

To determine if the documents were similar, I calculated an upper and lower bound for each document's SI, and then compared each document's SI to this interval. I chose a threshold of 30% to calculate the bounds. Figure 2 depicts this process graphically.

$$\text{if } (SI_{lower} < SI_2 < SI_{upper}) \text{ then } SI_2 \text{ is similar to } SI$$

Any SI falling between the bounds was regarded as a *hit* on a similar document and scored as a 1. Any *hit* outside of a zone discussed in section 4.1 was deemed a *false positive*, meaning the SI indicated the documents were similar though I knew they were not. Any *miss* (scored as a 0) inside a zone (the SI indicated the documents were not similar though I knew they were) was deemed a *false negative*.

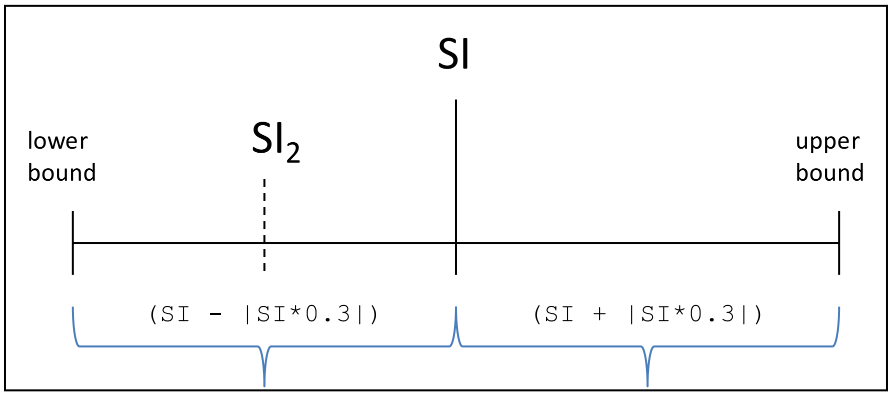

Figure 2 SI Upper and Lower Bounds

Using the notion of false negatives, false positives, and hits, I constructed a score for each combination of methods, input streams, input stream process variations, and process parameters by taking the ratio of correct indicators to incorrect indicators.

```
score = correct indicators / (false negatives + false positives)
```

Using this methodology, I looked for combinations of methods, input variation, process variations, and process parameters with the highest scores.

5 Results

Using the evaluation methodology discussed in section 4.2, I found 104 (out of 672) candidate combinations with scores > 1.0. From this set, I chose the 5 highest scoring combinations of method, input stream, input stream processing variations, and process parameters for further investigation. These top five candidates are listed in Table 9 .

Table 9 Results

Method	Input Stream Process Variation	k	w	False Positives	False Negatives	Correct	Score
43	B	40	100	0	10	35	3.50
47	C	40	100	14	5	40	2.11
43	B	4	5	3	13	32	2.00
3	B	16	25	8	10	35	1.94
21	B	40	100	2	14	31	1.94

5.1 Observations

- Input stream process variation B (stop words removed, but duplicate words kept) was used by 4/5 candidates.

- Method 43 produced two of the top five scores using the same input stream process variation but different process parameters.
- The *w* process parameter can be ignored since none of the candidate methods used it.
- Methods 3 and 43 used essentially the same process (sum the 0 mod 25 sketch) to arrive at the SI. The difference between them was that Method 3 operated on strings and Method 43 on words.
- No methods that used a *k* = 8 appear in the list. In fact, no method using *k* = 8 made the top ten. Based upon the literature [6], I expected this process parameter to yield the best results.

5.2 Analysis of Results

To really determine the method and combination of input stream, input stream process variations, and process parameters that produced the best results required examining each method's output meticulously. In particular, I was looking for places where false negatives were acceptable (e.g., where some documents were subsets of one another). False positives were also acceptable in some cases since the idea was that these documents would be suggested to an end user who would make the final decision on similarity. The following sections examine each of the top five candidate methods in greater detail.

5.2.1 Method 43B (w=40, k=100) : 3.50

This method produced results in complete agreement with the expected values. Recall that a value of "1" indicated the two documents were similar (within the +/- 30% threshold), while a "0" indicated they were not.

Table 10 Method 43B (k=40, w=100), Zone 1 – Proverbs Collection

	Proverbs.txt	Proverbs1-16.txt	Proverbs1-24.txt	Proverbs25-31.txt
Proverbs.txt	1	1	1	0
Proverbs1-16.txt	1	1	1	0
Proverbs1-24.txt	1	1	1	0
Proverbs25-31.txt	0	0	0	1

Table 11 Method 43B (k=40, w=100), Zone 2 – Constitution Collection

	usConstitution (copy).txt	usConstitution-BillofRights.txt	usConstitution-noPreamble.txt	usConstitution.txt
usConstitution (copy).txt	1	1	1	1
usConstitution-BillofRights.txt	1	1	1	1
usConstitution-noPreamble.txt	1	1	1	1
usConstitution.txt	1	1	1	1

Table 12 **Method 43B (k=40, w=100), Zone 3 – Declaration of Independence Collection**

	usDOI-1.txt	usDOI-Grievances .txt	usDOI.txt
usDOI-1.txt	1	0	1
usDOI-Grievances .txt	0	1	0
usDOI.txt	1	0	1

5.2.2 Method 47C (k=40, w=100) : 2.11

This method produced very good results within the zones; however, it also produced 14 false positives that hurt its overall score.

Table 13 **Method 47C (k=40, w=100), Zone 1 – Proverbs Collection**

	Proverbs.txt	Proverbs1-16.txt	Proverbs1-24.txt	Proverbs25-31.txt
Proverbs.txt	1	1	1	0
Proverbs1-16.txt	0	1	1	1
Proverbs1-24.txt	1	1	1	1
Proverbs25-31.txt	0	1	0	1

Table 14 **Method 47C (k=40, w=100), Zone 2 – Constitution Collection**

	usConstitution (copy).txt	usConstitution-BillofRights.txt	usConstitution-noPreamble.txt	usConstitution.txt
usConstitution (copy).txt	1	1	1	1
usConstitution-BillofRights.txt	1	1	0	1
usConstitution-noPreamble.txt	1	1	1	1
usConstitution.txt	1	1	1	1

Table 15 **Method 47C (k=40, w=100), Zone 3 – Declaration of Independence Collection**

	usDOI-1.txt	usDOI-Grievances .txt	usDOI.txt
usDOI-1.txt	1	1	1
usDOI-Grievances .txt	1	1	1
usDOI.txt	1	1	1

5.2.3 Method 43B (w=4, k=5) : 2.00

This method produced results in conformance with the expected values in the Proverbs and Declaration of Independence collections, but did not do as well with the US Constitution collection (3 false negatives). These misses in conjunction with 3 false positives hurt its score.

Table 16 Method 43B (k=4, w=5), Zone 1 – Proverbs Collection

	Proverbs.txt	Proverbs1-16.txt	Proverbs1-24.txt	Proverbs25-31.txt
Proverbs.txt	1	1	1	0
Proverbs1-16.txt	1	1	1	0
Proverbs1-24.txt	1	1	1	0
Proverbs25-31.txt	0	0	0	1

Table 17 Method 43B (k=4, w=5), Zone 2 – Constitution Collection

	usConstitution (copy).txt	usConstitution-BillofRights.txt	usConstitution-noPreamble.txt	usConstitution.txt
usConstitution (copy).txt	1	1	1	1
usConstitution-BillofRights.txt	0	1	0	0
usConstitution-noPreamble.txt	1	1	1	1
usConstitution.txt	1	1	1	1

Table 18 Method 43B (k=4, w=5), Zone 3 – Declaration of Independence Collection

	usDOI-1.txt	usDOI-Grievances .txt	usDOI.txt
usDOI-1.txt	1	0	1
usDOI-Grievances .txt	0	1	0
usDOI.txt	1	0	1

5.2.4 Method 3B (w=16, k=25) : 1.94

This method performed perfectly with respect to the expected values in each zone. However, it produced 8 false positives (it confused the `DeleteDocument.class` binary file with the US Constitution collection). These false positives degraded its overall score; however, false positives such as these could be easily dismissed by a user upon an initial inspection of the results.

Table 19 Method 3B (k=16, w=25), Zone 1 – Proverbs Collection

	Proverbs.txt	Proverbs1-16.txt	Proverbs1-24.txt	Proverbs25-31.txt
Proverbs.txt	1	1	1	0
Proverbs1-16.txt	1	1	1	0
Proverbs1-24.txt	1	1	1	0
Proverbs25-31.txt	0	0	0	1

Table 20 Method 3B (k=16, w=25), Zone 2 – Constitution Collection

	usConstitution (copy).txt	usConstitution-BillofRights.txt	usConstitution-noPreamble.txt	usConstitution.txt
usConstitution (copy).txt	1	1	1	1
usConstitution-BillofRights.txt	1	1	1	1
usConstitution-noPreamble.txt	1	1	1	1
usConstitution.txt	1	1	1	1

Table 21 Method 3B (k=16, w=25), Zone 3 – Declaration of Independence Collection

	usDOI-1.txt	usDOI-Grievances .txt	usDOI.txt
usDOI-1.txt	1	0	1
usDOI-Grievances .txt	0	1	0
usDOI.txt	1	0	1

5.2.5 Method 21B (w=40, k=100) : 1.94

This method also performed well, except for within the US Constitution collection, thus accounting for its degraded score.

Table 22 Method 21B (k=40, w=100), Zone 1 – Proverbs Collection

	Proverbs.txt	Proverbs1-16.txt	Proverbs1-24.txt	Proverbs25-31.txt
Proverbs.txt	1	1	1	0
Proverbs1-16.txt	1	1	1	0
Proverbs1-24.txt	1	1	1	0
Proverbs25-31.txt	0	0	0	1

Table 23 Method 21B (k=40, w=100), Zone 2 – Constitution Collection

	usConstitution (copy).txt	usConstitution-BillofRights.txt	usConstitution-noPreamble.txt	usConstitution.txt
usConstitution (copy).txt	1	1	1	1
usConstitution-BillofRights.txt	0	1	0	0
usConstitution-noPreamble.txt	1	0	1	1
usConstitution.txt	1	1	1	1

Table 24 Method 21B (k=40, w=100), Zone 3 – Declaration of Independence Collection

	usDOI-1.txt	usDOI-Grievances .txt	usDOI.txt
usDOI-1.txt	1	0	1
usDOI-Grievances .txt	0	1	0
usDOI.txt	1	0	1

6 Conclusion

After reviewing the detailed results in Section 5, it became clear that the best results were obtained using the following combination of method, input stream, input stream processing variation, and process parameters:

- Method: 43 (sum 0 mod 25)
- Input Stream: Soundex
- Input Stream Processing Variation: B (remove stop words, retain duplicates)
- k: 40 (size of the k-grams sampled)
- w: 100 (not applicable for this method)

This is not a totally unexpected result. [9] used a similar approach to validate their own methodology, and [6] claims this is how Alta Vista has been operating for years; however, I believe my introduction of the Soundex token stream has improved upon their results. A nice – and probably the most important – feature of this method is that it consistently always chooses the same fingerprints from the collection of shingles. Even if text has been inserted or deleted in a document, eventually the shingling process produces the same shingles as the original, and the 0 mod 25 filtering function consistently chooses the same set of fingerprints for the sketch.

Though my methodology was rather crude and brutish, and doesn't carry nearly the mathematical elegance of [1] [2] [4] [6], I have proven that the notion of reducing an entire document to a numeric value and using that value to gauge its similarity among other documents is a viable concept. I have also proven that the introduction fuzziness via the Soundex algorithm has improved the method's accuracy. Soundex introduced fuzziness by coalescing similarly spelled and sounding words into a single token. However, allowing duplicate words to remain in the input stream ensured that the token stream retained enough uniqueness to make the SI values distinguishable. If the addition of Soundex had had no impact, I would have expected to see all of the Method 43 scores (i.e., A, B, C and D) to be the closer.

6.1 Final Observations

A few final observations:

- The same method for calculating the SI must be used across the entire corpus of documents in order for the numbers to properly interact. This seems intuitively obvious and shouldn't be a surprise; the same limitation exists in cryptography. You can't compare MD5 hashes and SHA-1 hashes for the same document and expect them to match.

- This experiment was conducted on a very small set of documents. To really determine the robustness of the process and the theory, it should be run against a much larger collection of documents.
- The most frequently occurring method in the top 104 results was method 9 (including method 9, 29, and 49 – the sum of the highest 200 shingles) at 23 times. The second most frequently occurring method was 3 (3, 23, 43 – sum of 0 mod 25) at 21 times, and then method 7 (7, 27, 47 – sum of the lowest value 200 shingles) at 21 times. Using the lowest valued shingles (and by extension, the highest valued shingles) is discussed in [14]. Figure 3 depicts the frequency of methods in the top 104 results.

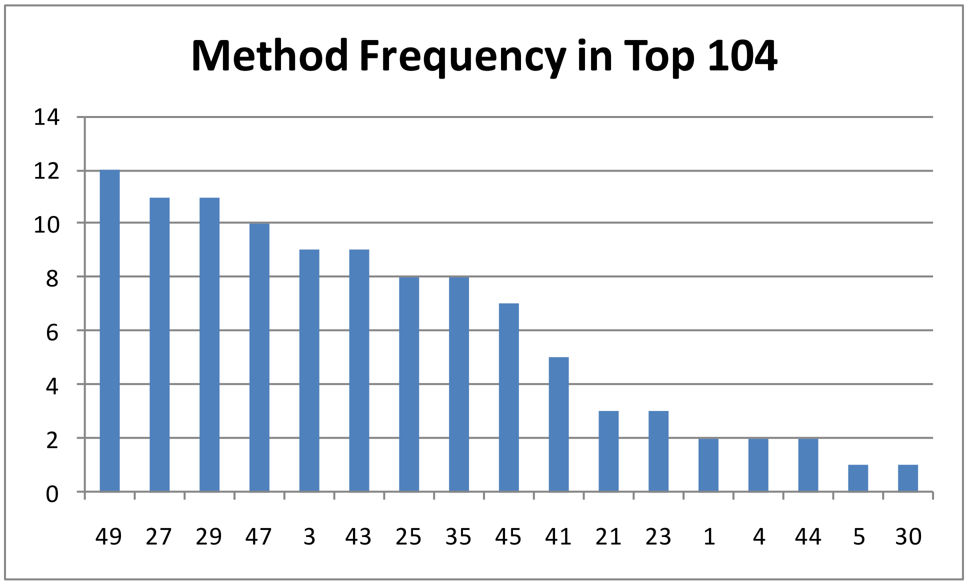

Figure 3 Most Frequent Used Methods in Top 104 Results

- I find it curious that no one has pursued this idea further than [9]. Perhaps they have proven that it is mathematically unsound and there is no reason to pursue it further? Regardless, I have proven that this is relatively inexpensive (effort-, processing- and storage-wise) and a useful way to detect similar or nearly duplicate documents in a corpus.

With regard to the original question that spurred this endeavor: yes, it is possible to determine if similar documents exist in a repository. And yes, this determination can be made at the time of ingestion (i.e., checkin). There are likely numerous ways this could be accomplished (e.g., the Lucene *`MoreLikeThis`* API); however, using the Similarity Index as described here is a quick and easy way to make a better than fair assessment of content, without the need for a full-text index or database.

6.2 Post Script

I created a relatively simple Aspect in Document 6.6 (by repurposing some of the code from the experiment) that generated SI values. I imported my document collection in to Documentum as basic `dm_documents` and applied the Aspect to each document in the collection. I then endeavored to identify similar documents using DQL. I learned that DQL doesn't work well with large numbers, but I was able to find similar documents using a query.

My approach was to find the SI value of my current document, determine what 30% of that value was, and then use that percentage to establish an upper and lower bound for the query. Something like this:

```
SELECT r_object_id
FROM dm_document
WHERE si_aspect.value > [lower bound].0
AND si_aspect.value < [upper bound].0
```

To force DQL to process the upper and lower bounds as double precision variables, I had to tack a "0" onto the end of each boundary number.

This DQL produced results in line with the experiment.

7 Selected Bibliography

Following is a selection of research papers, books, and articles I read to formulate and execute this experiment. If this topic, this paper, or the techniques it discussed are of interest to you, these citations will interest you too. Put on your thinking caps, some of this stuff is really heady.

1. Broder, A., *On the resemblance and containment of documents*, 1997, http://www.cs.princeton.edu/courses/archive/spr05/cos598E/bib/broder97resemblance.pdf.
2. Rabin, M., *Fingerprinting by Random Polynomial*, 1981, http://www.xmailserver.org/rabin.pdf.
3. Elbegbayan, N., *Winnowing, a Document Fingerprinting Algorithm*, 2005, http://www.ida.liu.se/~TDDC03/oldprojects/2005/final-projects/prj10.pdf.
4. Schleimer, Wilkerson, and Aiken, *Winnowing: Local Algorithms for Document Fingerprinting*, 2003, http://theory.stanford.edu/~aiken/publications/papers/sigmod03.pdf.
5. Rajaraman and Ullman, *Mining of Massive Datasets*, 2010, http://infolab.stanford.edu/~ullman/mmds/book.pdf.
6. Broder, A., *Algorithms for duplicate documents*, 2005, http://www.cs.princeton.edu/courses/archive/spr05/cos598E/bib/Princeton.pdf.
7. Johnson, A., *How MoreLikeThis Works in Lucene*, 2008, http://cephas.net/blog/2008/03/30/how-morelikethis-works-in-lucene/.
8. Preneel, B., *Analysis and Design of Cryptographic Hash Functions*, 2003, http://homes.esat.kuleuven.be/~preneel/phd_preneel_feb1993.pdf.
9. Cooper, Coden, and Brown, *Detecting Similar Documents Using Salient Terms*, 2002, http://www.labsoftware.com/flahdo/Papers/CIKMDuplicates.pdf.
10. Huston, C., *Fingerprinting Jar Files Using Winnowing*, 2009, http://www.catehuston.com/files/fingerprinting%20jar%20files%20using%20winnowing.pdf.
11. Dig, Comertoglu, Marinov, and Johnson, *Automatic Detection of Refactorings for Libraries and Frameworks*, 2005, http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.116.654&rep=rep1&type=pdf.
12. Nikkhoo, H., *The Impact of Near-Duplicate Documents on Information Retrieval Evaluation*, 2010, http://uwspace.uwaterloo.ca/bitstream/10012/5750/1/Khoshdel%20Nikkhoo_Hani.pdf.
13. Heintze, N., *Scalable Document Fingerprinting (Extended Abstract)*, 1996, http://www.cs.cmu.edu/afs/cs/user/nch/www/koala/main.html.
14. Broder, Glassman, Manasse, and Zweig, *Syntactic Clustering of the Web*, 1997, http://www.hpl.hp.com/techreports/Compaq-DEC/SRC-TN-1997-015.pdf.
15. Wikipedia, *Soundex*, 2011, http://en.wikipedia.org/wiki/Soundex.
16. Wikipedia, *Cryptographic Hash Functions*, 2011, http://en.wikipedia.org/wiki/Cryptographic_hash.
17. Yang, Peng and Zeng, *The Design and Implementation of Document Similarity Detecting Systems*, 2010, http://www.joics.com/publishedpapers/2010_7_3_739_745.pdf.
18. Saladjiev, *Way to find documents with similar content*, March 2011, https://community.emc.com/message/536386#536386.
19. com.planetj.math.rabinhash.RabinHashFunction64, http://rabinhash.sourceforge.net/.

20. Knuth, D., Soundex Algorithm, http://www.java-forums.org/java-lang/7438-soundex-algorithm-implementation-java.html.
21. Wikipedia*, Java hashCode()*, 2011, http://en.wikipedia.org/wiki/java_hashCode().
22. Garcia, Mi Islita website, http://www.miislita.com/information-retrieval-tutorial/information-retrieval-tutorials.html.
23. Ye, Wen, and Ma*, A systematic study on parameter correlations in large scale duplicate document detection*, 2007, http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.64.3748&rep=rep1&type=pdf

<SDG><