The Similarity Index v2.0

M. Scott Roth

May 2013

Abstract

This paper advances the concepts for detecting similarity among documents developed in The Similarity Index *(July 2011), and demonstrates a much faster and more accurate approach. The end goal of the paper is to demonstrate that it is possible to capture the salient features of documents in hash values such that these values can be compared to indicate similarity. This index of similarity can be implemented in a content management system to enhance the searching and mining of content.*

1 Introduction

In July 2011, I published a paper titled *The Similarity Index* on my blog (msroth.wordpress.com) that detailed my search for a process that could reduce the essence of a document to a number (the Similarity Index, or SI) so that it could be compared to other such numbers to determine similarity among documents. I was interested in finding a process that produced a number that could be queried using a percent differential (e.g., 90%), to find "similar[1]" documents in a collection. That paper chronicled my experimentation with numerous algorithms and procedures for producing such a number. I will refer to that paper, experiment, and results as SIv1.0.

The goal of the SIv1.0 was to affix the SI value to an object in a content repository as metadata. Then, whenever an object was selected, the repository could easily identify similar objects in the repository by querying for SI values that were within a range of the selected object's SI. The capability to identify similar content in this manner could augment a content management system's ability to search, mine, suggest, classify, and de-duplicate content, among other things.

In SIv1.0, I determined that it was possible to use a sliding *k-gram* of size 40 and the Java *String.hashCode()* as a hash method to produce a set of numeric *shingles* for a document. This set of *shingles* was reduced to a set of *fingerprints*, or a *sketch*, by choosing only those *shingles* that were evenly divisible by 25. The *fingerprints* were then summed to produce the SI. (This experiment relied heavily on the work of Andrei Broder, et al; see Selected Bibliography). For a more thorough discussion of *k-grams*, *fingerprints*, and *sketches*, see the SIv1.0 paper. Though SIv1.0 proved that the concept was viable, the methodology suffered from some limitations and flaws. Addressing those flaws has led to this second experiment, SIv2.0.

SIv1.0 had three flaws I hoped to correct in SIv2.0:

1. Some of the *shingles* produced – and ultimately some the SI values – were negative numbers. This should not have been possible since there were no subtraction operations used in the process. The presence of these negative values meant only one thing: the data type used to hold the *shingle* hash values overflowed. Negative numbers threw off the SI value (because the *fingerprints* were summed to compute the SI), thus masking the real results and inserting uncertainty in the process.

   For SIv2.0, I needed to find a hash that was guaranteed not to overflow, or would handle an overflow in such a way as to not artificially affect the SI.

2. The data corpus used to test SIv1.0 was too small, consisting of only 15 documents. Though SIv1.0 produced favorable results on this data set, I was concerned that it would not produce similar results on a larger data set. In fact, I later tested this theory with a corpus of 10,000

[1] Similar documents are those which share essentially the same words in the same sequence (syntactic similarity). These documents do not necessarily share similar meanings. For the purposes of this paper, the format of documents was inconsequential, the experiment operated only on the text of the documents.

documents and was justified in my concern: SIv1.0 did not perform well against this larger corpus.

For SIv2.0, I assembled a corpus of 13,620 documents ranging in size from 1KB to 4.3 MB on a diverse set of topics. This corpus should prove whether or not SIv2.0 was applicable to a large, diverse set of files.

3. Because the compilation of the *shingle* value involved generating relatively large numbers for each *k-gram* and summing them together to calculate the SI, the SI values grew in proportion to the size of the document being processed. Ultimately, these numbers overflowed the data type used to hold them (Java long).

   The shingle hash was roughly:

   ```
   s[0]*31^(n-1) + s[1]*31^(n-2) + ... + s[n-1]
   ```

   where `s` is the *k-gram*, and `n` is the length of the *k-gram*)

   For SIv2.0, I needed to find a way to consistently produce a 64-bit hash value regardless of the size of the document. I settled on a 64-bit value because I wanted most computers and OSes to be able to handle the comparison of SI values as primitive numeric types.

2 SI v2.0 Experiment

The goals of the SIv2.0 experiment were to address the shortcomings of the SIv1.0 experiment, and make improvements in speed, accuracy, and scalability of the process. Solving the data corpus size was simple enough; solving the hash algorithm and data type issues required a lot more research and experimentation.

2.1 Data Corpus

For this experiment, I assembled 13,620 files (343 MB) from various online sources (e.g., `textfiles.com` and Project Gutenberg). These files ranged in size from 1KB to 4.3 MB and covered a wide variety to topics[2]. Though some of the files inherently contained a lot of similarity (e.g., hard drive specification files in the Computer collection), I randomly introduced variations into the data set to increase its size and control similarity.

After downloading the files, I ran a Java program that implemented the following logic to introduce controlled similarity into the corpus:

1. Copy each file in the corpus to the workspace for the experiment. Each file had a 50% chance of selection for modification.

[2] Appendix 1 contains a list of the file collections used for testing.

2. If a file was selected for modification, it had a 50% chance of being selected for augmentation, and a 50% chance of being selected for reduction.

3. If a file was selected for augmentation, a random string from the file – up to 80% of the size of the file – was selected for addition to the file. This string was randomly inserted into the file.

4. If a file was selected for reduction, a random string from the file – up to 80% of the size of the file – was selected for removal from the file. The location of the deletion was randomly selected.

5. The files were saved with new file names to indicate whether the file was augmented or reduced, and by what amount. For example:

   - `feder15.txt` - the Federalist Papers (1.2 MB)
   - `feder15_i_43.txt` - the Federalist Papers with 43% additional text (2.1 MB)[3]

   This naming convention assisted in quickly determining how similar files were based upon the modifications made to them.

This technique effectively increased the data corpus to 20,476 files (523 MB).

2.2 The Hashing Algorithm

Developing a new hashing algorithm that did not overflow its data type and produced consistent 64-bit hashes regardless of the input size proved to be the most difficult part of this experiment. I wanted an algorithm that hashed "similar" files into the same "bucket". As humans, we can do this fairly easily. We can sort objects based upon some attribute and put all of the objects that share similar properties into the same group, or bucket. In computer algorithm parlance, this is called local clustering, and there are numerous examples of computer systems that implement this capability, for example: de-duplication engines, plagiarism detection systems, copyright violation systems, and recommendation engines. These systems (in general), create sets of *fingerprints* for each object in their universe and systematically compare each *fingerprint* in the sets to detect similarity. The difference between these implementations and what I was seeking was that I wanted to reduce the set of *fingerprints* to a single value that could easily be stored and compared to determine similarity, without the need for a database to store the *fingerprints*.

I surmised a hashing algorithm that worked similar to the MD5 or SHA-1 algorithms that processed fixed-sized blocks and produced fixed-sized output fields would be best. In these algorithms, the output fields are concatenated to produce the final hash. There were several problems with using an approach like this. First, I couldn't use an algorithm that included an avalanche effect (this is the property of the algorithm that ensures the hash is sufficiently changed based upon changing input). Instead, I wanted an algorithm that ignored minor changes in input, and consistently produced the same or similar hash values. The second problem was that because these algorithms generally produced four 16-bit fields

[3] Incidentally, despite the rather large amount of duplication inserted into this file, the experiment found these files to be similar.

that were concatenated to produce the final hash, I wasn't able to compare them using a simple +/-% approach as I did in SIv1.0.

After many attempts at creating my own algorithm and reading many research papers (see the Selected Bibliography if you are interested), I was led to the SimHash algorithm developed by Moses Charikar.

2.2.1 SimHash

The SimHash[4] algorithm was exactly what I was looking for; eloquently simple compared to other algorithms and techniques I reviewed, though not so simple that I could have devised it.

SimHash works by breaking the input string into *k-grams* and producing a fixed sized *shingle* for each *k-gram*. The size of the *shingle* is the same size as the final hash (e.g., 64-bits). Each bit position of each *shingle* is reviewed[5]. If the bit at `shingle[i]` is set (i.e., 1), then the same bit position in a temporary vector (e.g., `V[i]`) is incremented by 1. If the bit at `shingle[i]` is not set (i.e., 0), then `V[i]` is decremented by 1. Once the entire input string has been evaluated, the SimHash is calculated by reviewing the temporary vector, V. If the bit at `V[i]` is greater than 0, then the bit at `simhash[i]` is set to 1, else it is set to 0. The result of this process is a 64-bit binary number.

Following is a pseudo-code representation of the process:

1. Produce a set of *shingles* (S) for the input.
2. Initialize a temporary vector (V), 64-bits in size, containing all zeros.
3. For each *shingle* (`s`) in set S, if `s[i]` is `1` (where `i` = bit position), then increment `V[i]`. If `s[i]` is `0`, decrement `V[i]`.
4. Initialize the SimHash vector (H), 64-bits in size, containing all zeros.
5. After processing all of the *shingles* in S, evaluate the temporary vector, V: if `V[i] > 0`, then `H[i] = 1`, else `H[i] = 0`.
6. The resulting binary number represented by vector H is the SimHash value.

[4] The SimHash algorithm was developed by Moses Charikar in 2002, and described in his paper; Similarity Estimation Techniques from Rounding Algorithms (see Selected Bibliography). The algorithm is also patented in US Patent 7158961.

[5] The input is converted to a binary string for comparison.

Table 1 contains a simplified example of how the SimHash algorithm works. To conserver space, I used a 16-bit V and H, and a 3-bit *k-gram*. To produce the *shingles*, I simply summed the ASCII values of the characters in the *k-grams*. This is not a viable hashing technique for real world usage, but served well for this example. Each row in the table represents a successive loop in the algorithm (i.e., step 3 in the pseudo-code above). The input string was "Hello world".

Table 1: Sample SimHash Demonstration

k-gram	shingle	shingle (binary)	V															
			0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
			0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Hel	72 + 101 + 108 = 281	0000000100011001	-1	-1	-1	-1	-1	-1	-1	1	-1	-1	-1	1	1	-1	-1	1
ell	101 + 108 + 108 = 317	0000000100111101	-2	-2	-2	-2	-2	-2	-2	2	-2	-2	0	2	2	0	-2	2
lo_	108 + 111 + 32 = 251	0000000011111011	-3	-3	-3	-3	-3	-3	-3	1	-1	-1	1	3	3	-1	-1	3
o_w	111 + 32 + 119 = 262	0000000100000110	-4	-4	-4	-4	-4	-4	-4	2	-2	-2	0	2	2	0	0	2
_wo	32 + 119 + 111 = 262	0000000100000110	-5	-5	-5	-5	-5	-5	-5	3	-3	-3	-1	1	1	1	1	1
wor	119 + 111 + 114 = 344	0000000101011000	-6	-6	-6	-6	-6	-6	-6	4	-4	-2	-2	2	2	0	0	0
orl	111 + 114 + 108 = 333	0000000101001101	-7	-7	-7	-7	-7	-7	-7	5	-5	-1	-3	1	3	1	-1	1
rld	114 + 108 + 100 = 322	0000000101000010	-8	-8	-8	-8	-8	-8	-8	6	-6	0	-4	0	2	0	0	0
			H															
			0	0	0	0	0	0	0	1	0	0	0	0	1	0	0	0

The resulting SimHash value was: H = 0000000100001000 = 264 (decimal)

2.2.2 Evaluating SimHash Output

Hashes produced by SimHash can be compared as integer numbers or as binary numbers to determine similarity. Because the SimHash algorithm could place a 1 in the left-most bit of the binary hash number (forcing the decimal representation of that number to be negative), most authors preferred to compare the binary form of the hash using a Hamming Distance. Simply comparing these numbers based upon their decimal values could misrepresent their similarity due to differing cardinality caused by that first bit. This is similar to the negative numbers caused by overflow encountered with SIv1.0. This problem does not exist using bit-wise Hamming Distances.

For the purposes of this experiment, the Hamming Distance between two SimHash values was determined by simply identifying the number of bits that differed between the binary representations of the hashes. For example:

```
File 1 hash:      1010001111011011111100110100111101110110111010100000010100010000
File 2 hash:      1010001110011011111110000101011110111011011101010000010100010000
Bit differences:          1         1  11   1                         1
```

The Hamming Distance between File 1 and File 2 is 6. The lower the Hamming Distance, the more similar the files are.

2.3 The SI Index and Determining Similarity

The SimHash value is the Similarity Index I was searching for. Similarity between two files can be determined by comparing the Hamming Distance between their SI values (SimHashes), as demonstrated in Section 2.2.2.

I still liked the notion developed in SIv1.0 that the SI should allow querying for files that are within +/- X% of a known SI value. For example, it would be possible to find files that are 90% similar to a given file, or files that differed by only 10% from a given file. Knowing the SI is 64-bits long, I can locate files that are 90% similar (or only differ by 10%) by finding Hamming Distances between their SI values that are equal to or less than 6.

```
hamming_distance = hash_length – (similarity) * (hash_length)
hamming_distance = 64 – (0.90) * (64)
hamming_distance = 64 – 57.6
hamming_distance = 6.4
```

Therefore, files that are 90% similar, or differ by only 10%, will have a Hamming Distance between their SI values of 6 or less.

2.4 Experiment Execution

The experiment was run in three distinct parts: loading the data corpus, calculating the SI (SimHash) value, and comparing SI values to determine similarity. My experiment used code written completely in

Java and utilized classes from the Common Crawl-crawler project (see Selected Bibliography) for generating *shingles*, producing SimHash values, and calculating Hamming Distances.

2.4.1 Loading the Data Corpus

As explained in Section 2.1, an initial corpus of 13,620 files (343 MB) from various online sources was expanded to a corpus of 20,476 files (523 MB) by randomly introducing variations into selected files. Variations included both insertions and deletions of up to 80% the size of the original file. The modified files were given filenames that identified the size and type of variation introduced to the original file.

Statistics for the final data corpus were:

- insert modifications : 4,619
- delete modifications : 3,057
- unchanged : 12,800
- total files : 20,476

2.4.2 Calculating the SI (SimHash) Value

Step two of the experiment simply calculated the SI (SimHash) value for each file in the corpus. The file's name and SI value were stored in an internal memory array in preparation for the analysis step of the experiment. This step took 160 sec (3.3 MB/sec).

2.4.3 Analyzing SI Values

To analyze the SI values produced in step two, the program calculated the Hamming Distance between every SI value in the memory array described in Section 2.4.2. This required roughly 211 million comparisons. Files with Hamming Distances of 6 or less (90% similarity) were flagged as similar. This step produced results that were further analyzed using a spreadsheet. This step took 106 sec (1.99 million comparisons/sec).

2.5 Results and Observations

In the end, the experiment produced a result file 35,377 lines long. Thanks to some spreadsheet features that allowed for searching and special formatting, matches of similar files were easy to identify and verify.

Here are a few observations from the result set:

- The experiment identified 14,900 files (73% of corpus) as having similarity to at least one other file in the corpus. Of course, some of these files had more than one "match" as they proved similar to numerous files in the corpus, so this number overstates the result somewhat.
- File `st8134k.txt` was identified with the highest number of similar matches: 15. 25 files in the result set had matches of 10 or more to other files in the corpus. File `st8134k.txt` was a hard drive specification sheet and only differed to some of the files in the corpus by as little as a single character.
- Looking at modified files versus their unmodified forms, the algorithm did not match files with deleted material as well as those with inserted material. There were 2,613 files with Hamming

Distances of 7[6] or more when compared to their unmodified forms. Of these 2,613 files, only 141 included insertions (5%), all the rest were modified by deletion. This observation makes sense when you think about it. Absent *shingles* would definitely change the calculation of the SimHash value. Inserted, duplicated material *might* change the hash slightly, but the algorithm smoothed these variations (as it should have).

- Again, looking at modified files versus their unmodified forms, of the 940 pairs with Hamming Distance of 0 (exact matches), only 27 (3%) were of the deletion variety.
- Some files with as much as 80% insertion still compared as identical (Hamming Distance of 0). The greatest variance using the deletion modification that still produced a Hamming Distance of 0 was 33%. Intuitively this value seems high to me and causes me a little concern. This could be the subject of more testing and research.
- Several files were identified as similar that were, in fact, exact duplicates. These files were named differently or were included in different collections in the corpus and were not obviously identifiable as the same file before the experiment. For example, `computes/handles.txt` and `politics/anonymit` are the same file, as are `law/court.law` and `news/neidorf`. Once identified as exact matches by the SI process, they were manually verified. The modified forms of these files were also identified as similar by the SI process.
- The `computers/harddrives` collection contained hardware specification files. These files were essentially identical in content and format, except for the exact specifications they contained for a particular piece of hardware. For example, the difference between the *ST-71P Solid State FlashCard* (`st71p.txt`) and the *ST-71P5 Solid State FlashCard* (`st71p5.txt`) was only a few characters. The SI algorithm properly identified these spec sheets as similar to one another (in addition to the `st72p5.txt`, `st720p5.txt`, `st75p5.txt`, and `st710p5.txt`). However, it was discerning enough to distinguish these from the *ST-72A Solid State FlashCard* (`st72a.txt`) file that contained additional content absent from the other spec sheets.

3 Conclusion

In conclusion, I am extremely satisfied with the SIv2.0 experiment. The SimHash algorithm has proven to work exactly as I had envisioned the SI to work. I feel justified in my original notion that characterizing the content of a file as a single numeric value that could be compared to other such values to determine similarity was possible. Though I would have loved to have developed the algorithm myself, I am grateful to Mr. Charikar for his brilliance in developing SimHash for us. (I'm sure Google is also very pleased with him since SimHash seems to be an integral part of how their search engine suggests related sites to visitors.)

As I discussed previously, and as you can probably imagine, there are numerous applications for the SI: recommendation engines, research, content mining, plagiarism detection, content de-duplication,

[6] Less than 90% match.

content classification, etc. However, my intent was much simpler. I wanted a way to identify similar content in a content management system that did not require the use of a full text search engine. By assigning an SI value as metadata to each content object in a repository, a simple query could be constructed to find content similar to a selected content object. For example, a pseudo-query might look something like this:

```
select content_id from content_object where
   hamming_distance(SI_value,[SI]) <= 6
```

In this case, the `hamming_distance()` is a database function, stored procedure, or function of the content repository that determines the Hamming Distance between all `SI_values` and the `[SI]` value passed to it.

As demonstrated in the next section, I was able to implement a small prototype of this idea. It is my hope that in the future, major content management system vendors might adapt this idea and incorporate the notion of a Similarity Index into their core product offerings. In my mind, the benefits of the ability to quickly search for similar content in a repository – without the need for a full text indexing solution -- far outweigh the additional storage required to persist the SI value (a single 64-bit field), and the overhead to implement the SimHash algorithm and Hamming Distance logic.

4 Sample Implementation

Using a test environment consisting of Documentum 6.6 and SQL Server 2005, I was able to create a relatively simple implementation of SIv2.

First, I created a custom document object type, `sr_document`, that contained a metadata field `siv2` of type `STRING(64)`. I then imported several hundred documents from my test corpus and saved their SIs to the `siv2` metadata field.

I created a hamming distance function, `HamDist()`, in SQL using code posted by Jeff Smith (http://weblogs.sqlteam.com/jeffs/archive/2007/05/09/60197.aspx) that accepted two 64-character strings as inputs, and returned their Hamming Distance as output. This code can be found in Appendix 2.

I was then able to run a query like this from the Microsoft SQL Server Management Studio:

```
select
      s.r_object_id,
      s.object_name,
      m.siv2,
      dbo.HamDist(m.siv2,
      '0011111101011101000101010000111101110110010010010101110010000011
      ') as HD
from
      dm_sysobject_s s,
```

```
      sr_document_s m
where
      m.r_object_id = s.r_object_id
      and
      dbo.HamDist(m.siv2,
      '0011111101011101000101010000111101110110010010010101110010000011
      ') <= 6
```

where the 64-character SI is known from the experiment output for file `sttech.txt`. This query correctly identified the four other files in the repository that were similar to the specified file, according to the experiment results.

r_object_id	object_name	siv2	HD
090000018006e74c	**sttech.txt**	**0011111101011101000101010000111101110110010 01001010111001000011**	**0**
090000018006e74d	treknolo	0001101101000101000101010000111101111110010 01001010111001001011	6
090000018006e74e	warpte_i_6.txt	0001111101011101000101010000111101110110010 01001010111001000011	1
090000018006e764	treknolo_i_25	0001101101000101000101010000111101111110010 01001010111001001011	6
090000018006e765	warpte.txt	0001111101011101000101010000111101110110010 01001010111001000011	1

Unfortunately, DQL (Documentum's query language) does not allow user-defined database functions or stored procedures, so a similar query is not possible directly from Documentum. It is possible run the query as an SQL Pass-through query using the `exec_sql` syntax like this:

```
execute exec_sql with
      query= 'select s.r_object_id,
                  s.object_name,
                  m.siv2,
                  dbo.HamDist(m.siv2,
''0011111101011101000101010000111101110110010010010101110010000011'')
as HD
      from
            dm_sysobject_s s,
            sr_document_s m
      where
            m.r_object_id = s.r_object_id
            and
            dbo.HamDist(m.siv2,
''0011111101011101000101010000111101110110010010010101110010000011'')
<= 6'
```

However, `exec_query` only returns `true` or `false` as its result, not the result of the actual SQL query.

There are numerous alternatives for implementing an SI solution in Documentum (e.g., a Service-Based Object (SBO) or web component that executes the query directly on the database using JDBC). There will be challenges with each alternative, but they could be a fun endeavors and possibly subjects for future work.

5 Selected Bibliography

Following is a list of books, papers, websites, and blogs I used while researching this paper. I found all of these references helpful in understanding this area of information science, if not downright fascinating. Some of these resources I found after I ran my experiments and drew my conclusions, so you will see some similarity between this paper and some of these sources. I encourage you to take advantage of these resources also.

1. Broder, Andrei; Glassman, Steven; Manasse, Mark; Zweig, Geoffrey; *Syntactic Clustering of the Web*; http://www.hpl.hp.com/techreports/Compaq-DEC/SRC-TN-1997-015.pdf.

2. Schleimer, Wilkerson, and Aiken; *Winnowing: Local Algorithms for Document Fingerprinting*, 2003; http://theory.stanford.edu/~aiken/publications/papers/sigmod03.pdf.

3. Pi, Bingfeng; Fu, Shunkai; Wang, Weilei; Han, Song; *SimHash-based Effective and Efficient Detecting of Near-Duplicate Short Messages*; http://www.academypublisher.com/proc/iscsct09/papers/iscsct09p20.pdf.

4. Henzinger, Monika; *Finding Near Duplicate Web Pages: A Large Scale Evaluation of Algorithms*; http://infoscience.epfl.ch/record/99373/files/Henzinger06.pdf.

5. *Java hashCode()*, Wikipedia, the free encyclopedia; http://en.wikipedia.org/wiki/Java_hashCode().

6. *Why does Java's hashCode() in String use 31 as a multiplier?*, stackoverflow.com, http://stackoverflow.com/questions/299304/why-does-javas-hashcode-in-string-use-31-as-a-multiplier.

7. Moulton, Ryan; *Simple Simhashing*; http://moultano.wordpress.com/article/simple-simhashing-3kbzhsxyg4467-6/.

8. Jenkins, Bob; *Hash Functions and Block Ciphers*; http://burtleburtle.net/bob/hash.

9. Arash Partow; *General Purpose Hash Function Algorithms*; http://partow.net/programming/hashfunctions/index.html.

10. Charikar, Moses; *Similarity Estimation Techniques from Rounding Algorithms*; http://www.cs.princeton.edu/courses/archive/spr04/cos598B/bib/CharikarEstim.pdf.

11. Simpliplant; *Calculating similarity between text strings in Python*; http://blog.simpliplant.eu/calculating-similarity-between-text-strings-in-python/.

12. Philipp Braun; *Handling problematic content: A Google Algorithm for detecting near-duplicates;* http://www.philippbraun.net/2011/05/handling-problematic-content-google.html.

13. Charikar, Moses; US Patent 7158961; *Methods and apparatus for estimating similarity;* http://www.google.com/patents?id=tph-AAAAEBAJ&zoom=4&dq=Methods%20and%20apparatus%20for%20estimating%20similarity&pg=PA1#v=onepage&q&f=false.

14. *Metaphone*; Wikipedia, the free encyclopedia; http://en.wikipedia.org/wiki/Metaphone.

15. Zobel, Justin; Dart, Philip; *Phonetic String Matching: Lessons from Information Retrieval;* http://goanna.cs.rmit.edu.au/~jz/fulltext/sigir96.pdf.

16. Manku, Gurmeet; Jain Arvind; Sarma, Anish; *Detecting Near Duplicates for Web Crawling;* http://www2007.org/papers/paper215.pdf.

17. Garcia, Edel; *Term Vector Calculations A Fast Track Tutorial*; http://www.miislita.com/information-retrieval-tutorial/term-vector-fast-track.pdf.

18. Rivest, Ron; *The MD5 Message-Digest Algorithm*; http://www.ietf.org/rfc/rfc1321.txt.

19. *MD5*; Wikipedia, the free encyclopedia; http://en.wikipedia.org/wiki/MD5.

20. *Hamming Distance*; Wikipedia, the free encyclopedia; http://en.wikipedia.org/wiki/Hamming_distance.

21. *Simhashing (hopefully) made simple*; http://ferd.ca/simhashing-hopefully-made-simple.html.

22. Kelcey, Mat; *part 3: the simhash algorithm*; http://matpalm.com/resemblance/simhash/.

23. Wang, Wei; *Similarity Join Algorithms: An Introduction*; http://www.cse.unsw.edu.au/~weiw/project/tutorial-simjoin-SEBD08.pdf.

24. Karimi, Aref; *The magic behind the Google Search*; http://aspguy.wordpress.com/2012/02/18/the-magic-behind-the-google-search/

25. `SimHash` class, *Common Crawl-crawler* project; https://github.com/commoncrawl/commoncrawl-crawler/tree/master/src/org/commoncrawl/util

26. Smitelli, Scott; *Fun with YouTube's Audio Content ID System*, 2009; http://www.scottsmitelli.com/articles/youtube-audio-content-id

27. MacKay, David J.C; *Information Theory, Inference, and Learning Algorithms*, 2005; Cambridge University Press; http://www.inference.phy.cam.ac.uk/mackay/itila/

28. Wiegand, Thomas and Schwarz, Heiko; *Source Coding: Part I of Fundamentals of Source and Video Coding*, 2011; http://www.stanford.edu/class/ee398a/BookWiegandSchwarz.pdf

29. Smith, Jeff; *Hamming Distance Algorithm in SQL*; http://weblogs.sqlteam.com/jeffs/archive/2007/05/09/60197.aspx.

30. Sadowski, Caitlin and Levin, Greg; *SimHash: Hash-based Similarity Detection*, 2007, http://infolab.stanford.edu/~manku/papers/07www-duplicates.pdf.

6 Appendix 1

Test data corpus.

6.1 Textfiles.com

- Computers (1704 files / 37.4MB)
- Humor (2067 files / 26.9MB)
- Internet (852 files / 36.4MB)
- Law (534 files / 15.4MB)
- Media (167 files / 6.29MB)
- Misc (398 files / 14.2MB)
- News (185 files / 2.32MB)
- Politics (2200 files / 59.4MB)
- Programming (608 files / 22.1MB)
- Science (280 files / 4.94MB)
- SF (636 files / 23.6MB)
- Stories (477 files / 12.5MB)
- UFO (2935 files / 35.8MB)
- Uploads (559 files / 4.72MB)

6.2 Project Gutenberg

- 1mlkd11.txt 799KB
- 2city10.txt 759KB
- 1776-va_rts.mht 7KB
- 2000010.txt 580KB
- Alcott-little-261.txt 1041KB
- Alice.txt 151KB
- Barrie-peter-277.txt 260KB
- Bronte-wuthering-304.txt 660KB
- Bunyan-pilgrims-304.txt 298KB
- Civildis.txt 52KB
- Common_sense.txt 123KB
- Dgray10.txt 442KB
- Dracula.txt 839KB
- Getty11.txt 10KB
- Hamilton-federalist-548.txt 1166KB
- Hdark11.txt 222KB
- Kjv10.txt 4329KB
- World94.txt 2802KB

7 Appendix 2

Hamming Distance function in SQL from
http://weblogs.sqlteam.com/jeffs/archive/2007/05/09/60197.aspx.

```
create function HamDist(@value1 char(64), @value2 char(64))

returns int
as
begin
declare @distance int
declare @i int
declare @len int

select @distance = 0,
@i =1,
@len = case when len(@value1) > len(@value2)
then len(@value1)
else len(@value2) end

if (@value1 is null) or (@value2 is null)
return null

while (@i <= @len)
select @distance = @distance +
case when substring(@value1,@i,1) != substring(@value2,@i,1)
then 1
else 0 end,
@i = @i +1

return @distance
end
```

Example:

```
SQL> select dbo.HamDist('00100','11100')
```

Result:

```
SQL> 2
```

<SDG><