Working Paper Series
ISSN 1177-777X

Text Categorization and Similarity Analysis: Implementation and Evaluation

Michael Fowke[1], Annika Hinze[1], Ralf Heese[2]

Working Paper: 10/2013
December 2013

[1]Department of Computer Science
The University of Waikato
Private Bag 3105
Hamilton, New Zealand
[2] Pingar International Ltd.
152 Quay St, Auckland, New Zealand

Text Categorization and Similarity Analysis: Implementation and Evaluation

Michael Fowke[1], Annika Hinze[1], Ralf Heese[2]

[1]University of Waikato, Hamilton, New Zealand
[2]Pingar International Ltd, Auckland, New Zealand

Executive Summary

This report covers the implementation of software that aims to identify document versions and semantically related documents. This is important due to the increasing amount of digital information. Key criteria were that the software was fast and required limited disk space. Previous research determined that the Simhash algorithm was the most appropriate for this application so this method was implemented. The structure of each component was well defined with the inputs and outputs constant and the result was a software system that can have interchangeable parts if required.

The software was tested on three document corpuses to try and identify the strengths and weaknesses of the calculations used. Initial modifications were made to parameters such as the size of shingles and the length of hash value to ensure hash values were unique and unrelated phrases were not hashed to similar values. Not surprisingly longer hash values gave more accuracy and the runtime was not increased significantly. Increasing shingle size also gave a better reflection of the uniqueness of each input phrase. The naive implementation performed moderately on a custom made document version corpus but the similarity values were low for documents with only a few word changes. Using a similarity measure based on the Jaccard Index was more accurate. The software was able to successfully identify most document versions correctly and only had issues with the merging and separating of paragraphs. A theoretical solution was described for how this issue could be resolved.

Testing for semantically similar documents was limited compared to the testing for versions as finding document versions was identified as the focus. Initial testing showed that hashing the extracted entities for each paragraph returned values with limited information for each paragraph. Future work should analyse the entities at document level rather than paragraph level.

1. Introduction

Due to the increasing amount of digital information, it is now necessary for software to group documents based on topic and to identify duplicates. The software should ideally find the similarity of documents as well as identify the history of each. This project is in collaboration with Pingar who is company based in Auckland who aims to organise the increasing amount of digital information.

A literature study was carried out [1] as well as research into the specific similarity measure to use for best accuracy in finding related documents [2]. The Simhash algorithm was identified as the best measure to use for document versions and this report covers the implementation of the software. As well as coding the algorithm, the entire software system was developed including the file I/O, document storage and processing method.

The software was tested on three document corpuses and modified to give the most accurate classification results. During the report suggestions are made for improvements to the software and occasionally it was not practical to implement the changes so they are reasoned theoretically so the software can be improved further in the future.

2. Background

To understand the solution some important concepts need to be first introduced. This section covers some important terms that are used throughout the report followed by the results of research carried out earlier in the project.

2.1. Semantic technology

Pingar have provided the Pingar API for extracting entities from a document collection and a Taxonomy generator for finding relationships between extracted entities. These two technologies will be used to generate more accurate classifications than using the document text alone. The Pingar API would extract Bill Clinton and Barak Obama as entities from a document and the Taxonomy generator identifies these are people. The entities are the objects and the taxonomy gives the semantic relationships between them.

2.2. Initial software structure

A literature review [1] was carried out that looked at existing software solutions and analysed each based on their appropriateness for use in this software. At the end of the study the overall structure was decided as shown in Figure 1.

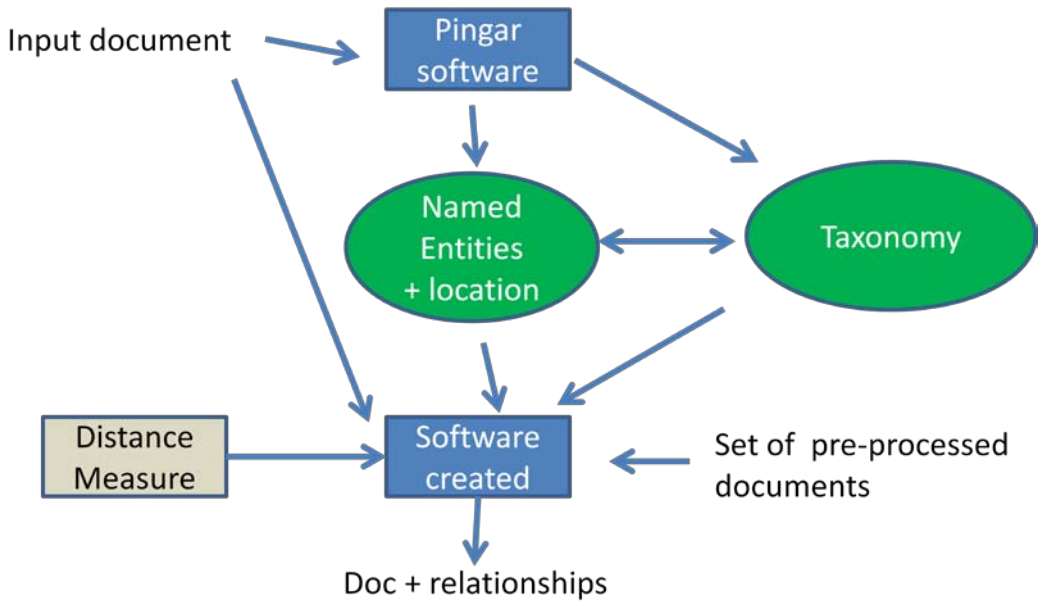

Figure 1: Overall software structure

2.3. Research into Similarity Measure

Research was carried out on the distance measure to identify which algorithm is the most appropriate for this document classification problem. Three similarity measures were considered including a Simhash approach, a word frequency approach and finally a clustering algorithm. Each of the algorithms was assessed on their performance on five key criteria. Word frequency and Simhash were able to accurately find related documents but Simhash is much better in terms of speed of comparisons and amount of disk space used. For this reason it is used as the similarity measure.

The rest of the research was concerned with finding the best way to combine the Simhash algorithm with the output from the provided Pingar API and taxonomy generator to accurately find semantically related documents as well as versions of documents. Hashing algorithms take an input phrase and perform a calculation on it to output a single numerical value to represent the input. The hash value is many times more compact than the original text. Simhash is a type of hashing algorithm and is unique in that it will output similar hash values if the input phrases are similar.

It was decided that using the Simhash algorithm on the original document text could accurately find versions of documents and the Pingar API was not required for this part of the solution. The document text is analysed in chunks with the Simhash value calculated for each paragraph. Documents are compared on the number of paragraphs they share that are similar enough to be versions (few bits different in hash values).

For finding semantically similar documents, the Simhash algorithm is combined with the output from the Pingar API to generate accurate results. The key concepts or entities are extracted from each paragraph and concatenated into a single string then input into the Simhash algorithm. Paragraphs on similar topics have similar hash values and documents with a certain number of related paragraphs are considered semantically related.

3. Solution

This section covers the solution that was implemented that best met the goals of the project. The system design is broken into components and each one has its implementation details described.

3.1. Components of solution

The architecture of the overall system was developed, in which the key components are: Text provider, chunker, pre-processing, Simhash algorithm, comparisons component and the storage components for processed documents as well as document relationships. In the following each of the components are described separately. The components are designed based on interfaces with fixed inputs and outputs so different implementations of components can be created and interchanged with existing components.

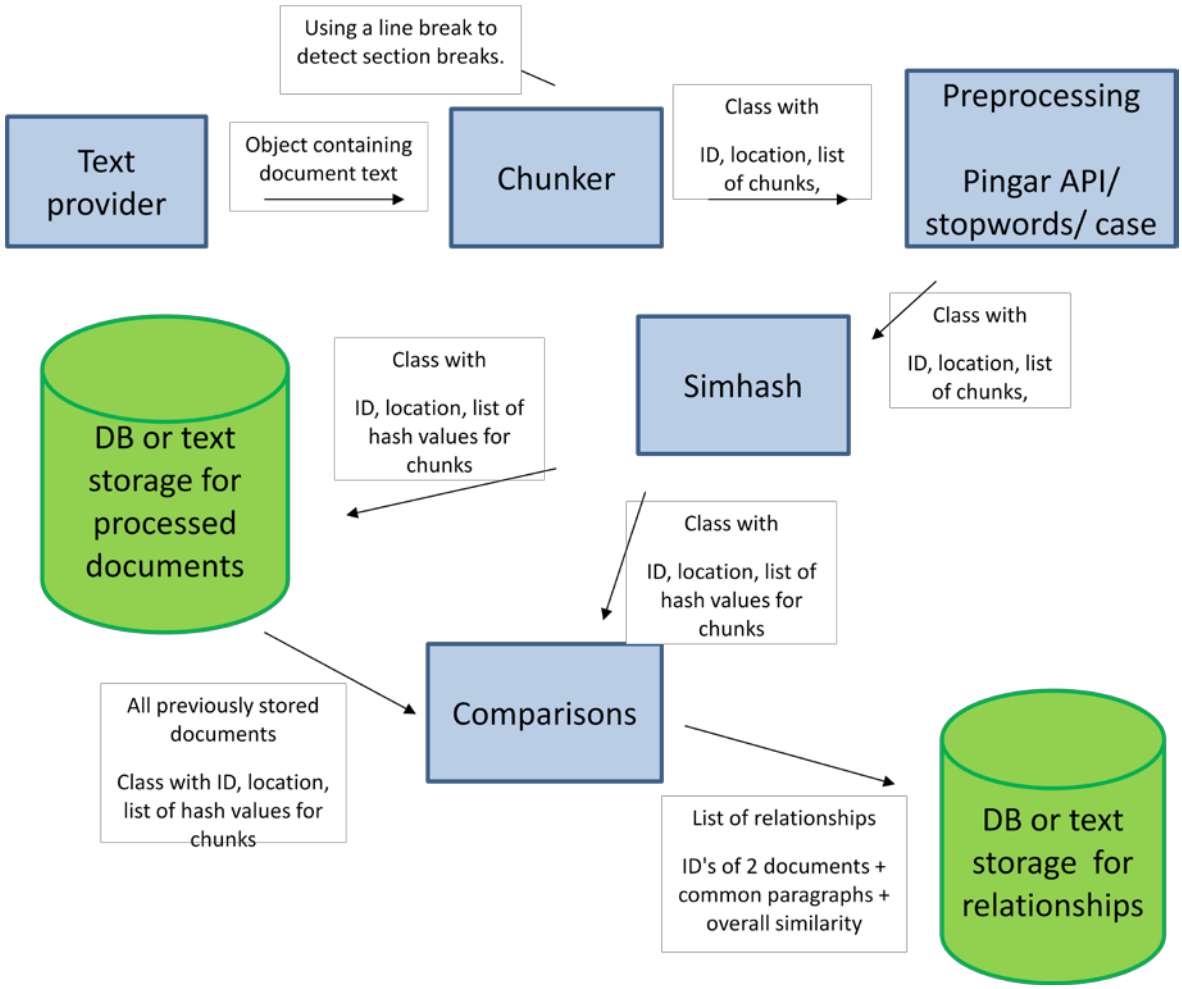

Figure 2: Components of software solution

3.2. Text provider

The text provider must be able to process an input file and output an object that can be analysed by the chunker component.

The text provider is responsible for reading text files from the secondary storage. It creates a data structure representing the document throughout the analysis and the processing. The data structure contains a unique document ID and the plain text of the document. The document ID is a non-negative integer value and is assigned automatically to each document.

Users can specify the location of the documents on the command line or the system will prompt for it in a file dialogue.

3.3. Document Chunker

The next component to implement was the document chunker. The chunker receives a document object and must separate it into the chunks or paragraphs it contains. The chunker has a character as a parameter which specifies the symbol to use for the end of each section. The output from the document chunker is a chunked document which extends the document class so it has the same ID and location but also has an arraylist of chunks of text from the document.

An implementation was created that required the text files to have a certain symbol between paragraphs to represent the end of a section. The chunker was then modified so it was able to detect chunks from normal documents as long as paragraphs were separated by a single line. This is less specific than using the character delimiter but still requires the documents to be in the same format. This chunking technique is fine for testing of the similarity measure but for the system to be developed further, the chunker would need to be more robust.

3.4. Preprocessing

Pre-processing involves adjustments that can be made to documents in order to improve accuracy. One possible pre-processing option is entity extraction where the main concepts are extracted from each paragraph. Entity extraction is a required pre-processing step to find semantically related documents. Pre-processing is not an entirely necessary part of the overall system and it can function without this step as the input and output of pre-processing is a chunked document.

3.4.1. Entity extraction

Entity extraction involves the use of the Pingar API for extracting the key concepts from a document. The entity extraction receives a chunked document and returns a chunked document with an extra array list with each element being the concatenation of the entities for a chunk. Each call to the Pingar API can take 10 input strings to extract entities from so if a document has longer than 10 paragraphs, more than 1 call to the API is required. The entity extraction class is written specifically for this software with chunks passed to the extractor and then the responses concatenated but the bulk of the class is based on Pingar API example classes. Issues arose in testing with a limit on the number of API calls possible per month to the Pingar API.

3.4.2. Stop words

Stop words were left in the document text for identifying document versions. These are the words in a document that are very common and do not contribute to the overall topic and include "the", "and", "was", and "than". Testing was not carried out but the removal of stop words considered detrimental as it is an approach taken when finding semantics rather than version. Darling [3] also states that stop word lists must often be domain-dependent, and there are inevitably cases where filtering results in under-coverage or over coverage. If a document has the same key words but different stop words then this information is important in determining document versions.

3.5. Simhash

This is a major part of the system. The Simhash component must be able to receive a text phrase and output the calculated Simhash value. Simhash is implemented in two steps. A Simhash algorithm is first written to hash an input phrase. The Simhash component also needs to take in a

chunked document and perform this Simhash algorithm on each of the sections and output a hashed document. Hashed document extends Document so it has the same document ID and location.

3.5.1. Simhash algorithm

The implementation of the Simhash algorithm is based on a method described on the Matpalm site [4]. The difference in calculated Simhash values is measured by the number of bits different between two Simhash values. Figure 3 shows this difference in bits.

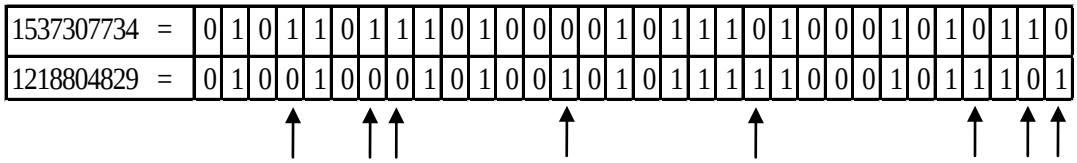

Figure 3: Bit difference between two Simhash values

Break the input into phrases

This part of the algorithm involves breaking the input phrases into smaller chunks called shingles containing a few letters. The example given broke the text into two letter shingles and this was the method created and tested initially. Two letter shingles include spaces and no duplicate shingle is included. An example phrase used from a test document is "Tiger Woods has reportedly divorced his wife Elin Nordegren". Figure 4 shows this phrase broken into shingles. Testing was done in the implementation to find the optimum shingle size.

ti	ig	ge	er	r␣	␣w	w	oo	od	ds	s␣	␣h	ha	as	␣r	re	ep	po	or	rt
te	ed	dl	ly	y␣	␣d	di	iv	vo	or	rc	ce	d␣	hi	is	wi	if	fe	e␣	␣e
el	in	n␣	␣n	no	rd	de	eg	gr	re	en									

Figure 4: Shingles from input phrase

Hash each feature

The example implementation used 32 bit hash values and this was the size tested initially. The length should be long enough so that clashes did not occur with different input being hashed to the same output. The hash size must also be short enough to be computationally efficient. The Java [5] hashing function gives small hash values even though it was a 32-bit hashing algorithm. This is possibly due to the input being small. As a result some of the phrases are found to be very similar even if they are in fact completely different. A new hashing algorithm was written to resolve this. The hashing method gets the lowest 32 bits when the byte value of the two letters in the shingle are multiplied by a large prime number. The prime number used was 27644437 [6] and it produced good quality hash values. The number is a bell prime number and easily proved as prime. The algorithm works much better with this new hashing process shown in Figure 5. Testing was done with different hash sizes to find the balance between unique hash values and fast run time.

Input 2 letter shingle	= “ti”
Numerical value	= byte value of t * byte value of i
	= 116 * 105
	= 12180
Hash value	= lowest 32 bits of (12180 * 27644437)
	= 1701793572

Figure 5: Hashing algorithm implemented

Keep an array to modify

For each of the hashed shingles described above, if a bit *i* is set then add 1 to the value at position *i* in the array. If bit *i* is not set in the hash value, then subtract one from the value at position *i* in the array. This component has little room for different implementations and the only variation is whether the array has size 32 or another length depending on the hash size.

Calculate Simhash value

Set bit *i* to 1 if the value at position *i* in the array above is > 0. Set the value to 0 otherwise. Again there is little room for variation with the length of the hash being determined by the user specified length.

3.5.2. Simhash operation

The Simhash component takes in a chunked document and outputs a hashed document. Each chunk within the document needs to have the Simhash operation described above applied to it. This involves passing each of the chunks within the array list to the hashing function. If the user has chosen to also look for semantically similar documents, then there will also be a list of entities per chunk which will need hashing. The hashing algorithm has customizable parameters such as the size of the hash value produced.

3.6. Processed document storage

The only requirement for this component was that a document can be stored once it has had its sections hashed and then loaded again from the storage at a later time.

There were two options for storing the results of the processing: as a file or in a database. Although storing in a database provides the advantages of easier field separation and more elegant reloading of data from storage the implementation is based on a file based data storage. This approach has been chosen because the focus of this thesis is the quality and the performance of the similarity measure.

File readers and writers are used to store the documents ID, location and each of the hash values calculated for its chunks (document text and entities extracted). The files are output with a user specified character between each field. The files can be reloaded by looking for the same delimiter. Each document requires a single line of storage in a text file so it is very efficient.

3.7. Comparisons component

This is the crucial component in the overall system and involves using the Simhash values calculated to determine the overall similarity between two documents. This component consists of two subcomponents. The first component looks to find the hash values that are similar in number of bits

and their positions. The second component looks at the difference in hash values to determine overall similarity.

3.7.1. Finding closely related chunks

This step involves finding the chunks in documents whose hash value is similar to the hash values of chunks in other documents. A method was implemented and some extra analysis was carried out into a potentially more efficient but restricted method.

Implemented method (method 1)

The method implemented involves every chunk having its hash value compared with every chunk in the other documents. The actual comparison of two hash values is efficient as it involves bit masking to determine whether a certain bit is set. Using the "bitwise and" operation on a hash value with the value n^2-1 returns whether the hash value has a 1 or a 0 in position n of its binary representation.

Total comparisons required

For 100 documents with 20 paragraphs each.

Between two documents = 20^2 = 400 comparisons

Number of document comparisons = nCr = 100C2 = 4950

Total chunk comparisons = 4950 * 400 = 1980000

Theoretical method (method 2)

A second option was considered which would likely be faster than the implemented method. This method takes advantage of the numerical properties of hash values by ordering them to find other hash values that are similar in value. The hashes are ordered by numerical value and adjacent values checked for bit difference. Values that are not similar will never need to be checked for the number of bits different. This method is essentially a heuristic that uses previous knowledge of numerical properties where as the first method uses blunt force to check every possible pair of hash values.

Although it is faster it required all chunks to be in memory which is not an option because the software is required to process at least 100,000 documents and the space required is far too much for memory. This method can only be used on small document corpuses. The method is based on a speed up algorithm in Matpalm [4] but adapted for this software. The difference between these two methods is purely in speed performance and there is no difference in classification accuracy.

The method implemented (method 1) involves analysing one document at a time and comparing each chunk with each chunk in every document in the already processed collection of documents. The second method involves comparing all of the chunks from every document at once and some of the comparisons can be avoided. The first method stores each document once it is processed and only considers one document at a time. The new method would need all of the chunks in memory at the same time so obviously it would break if the number of documents were too large for memory.

Suggested implementation of method 2

The following outlines method 2 and how it would work on a smaller document corpus.

1) Process single document

Read in a single document and break the document into chunks (using the chunker described earlier). Calculate the hash value for each of the chunks. Figure 6 shows a new class that is required for this method with fields for a hash value, document ID and location of the paragraph in the document. Create an instance of the chunk class for every chunk in the document.

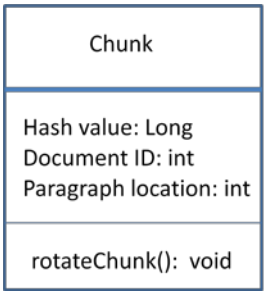

Figure 6: Chunk class

2) Repeat for all documents in the corpus

Process every document in the corpus as specified above. Keep a list containing all the chunks from the entire corpus.

3) Create a 2D array with each element being a list of relationships

The array is created with each element representing the comparison between two documents in the collection (Figure 7). The array is initially empty and will be filled as chunks are compared against each other in the following steps. The objects added to each element are relationship objects specifying which 2 paragraphs from the documents are determined as versions of each other. The purpose of a relationship object is to store the strength of relationship between two paragraphs for analysis at a later stage. The similarity measure is symmetric so half of the array will contain no values as the location (2, 4) is the same as (4, 2). The main diagonal will not contain any values as that would be a document compared against itself.

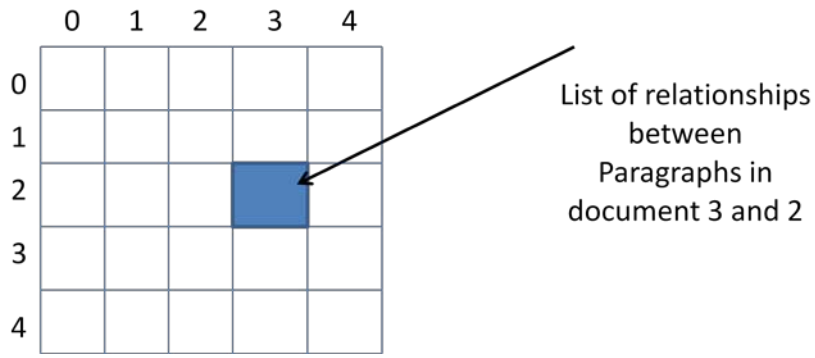

Figure 7: Illustration of 2D relationship array.

4) Order the entire list of chunks based on their hash value

The chunks are ordered based on their numerical value. If two hash values differ in the low order bits then they will appear close together when ordered numerically. Figure 8 shows an ordered list of hash values and phrases (3, 6) and (8, 5) have ended up together and both have a small bit difference. Chunk needs to implement the comparator interface so the chunks can be easily sorted.

Phrase number	value	Bit value	Bit difference from phrase above
4	934	0000001110100110	
3	2648	0000101001011000	9
6	2650	0000101001011010	1
1	37586	1001001011010010	5
8	40955	1001111111111011	6
5	40957	1001111111111101	2
2	50086	1100001110100110	9
7	64475	1111101111011011	9

Figure 8: Ordered hash values

5) Find neighbouring chunks in list within similarity threshold

Neighbouring chunks are chunks that can possibly be within the similarity threshold as their numerical values are similar. Start with the chunk at the first position in the list and compare its hash value with the hash value of the chunk in the second position. If the chunks are within the specified similarity threshold then create a relationship object containing how many bits differ between the two chunks and the paragraph number for each of the chunks. Add this relationship to the list at the appropriate position in the 2D array. If chunks 1 and 2 in the document are within the threshold, then chunks 1 and 3 must also be compared as they could also be within the required threshold. Continue this until finding two chunks that are outside the threshold. At this point begin comparing chunk 2 with chunk 3, then chunk 2 with chunk 4 and so on.

6) Rotate each of the chunks 1 bit to the left

Each of the chunks is rotated 1 bit to the left so that hash values that differ in the highest order bit now differ in the lowest bit. Rotating the hash values keeps the bit difference intact and when the hash values are reordered these values will now appear next to each other. The rotateChunk() method within each chunk will handle the rotation. By rotating all the values one bit to the left, the difference between each of the values remains constant. Figure 9 shows the rotated hash values with the bit difference the same as in the previous figure.

Phrase number	value	Bit value	Bit difference from phrase above
4	3736	0000111010011000	
3	10592	0010100101100000	9
6	10600	0010100101101000	1
1	19274	0100101101001010	5
8	32750	0111111111101110	6
5	32758	0111111111110110	2
2	3739	0000111010011011	9
7	61295	1110111101101111	9

Figure 9: Rotated hash values with bit difference intact

7) Reorder the chunks in the list based on the hash value.

Put the list of chunks back into order.

8) Repeat steps 5, 6 and 7

Find the chunks that are within the specified threshold. Then rotate each chunk one bit to the left and then reorder the list again. The total number of bit rotations required is equal to the number of bits in the hash value. So a 32 bit hash value requires 32 rotations.

By this point the 2D array will contain in each position a list of paragraphs that are versions of each other between the two documents. The similarity between two documents is then calculated as described in section 3.7.2.

Number of comparisons required

It is difficult to put a figure on the number of comparisons required as there is a lot of variation possible. If two documents are completely unrelated then very few of the chunks next to each other will be within the similarity threshold. This will mean for example chunk 1 and 2 will need comparing but not 1 and 3 as the difference between 1 and 2 is already too great. If the documents are more similar then more hash values will need comparing. The main idea is that a large number of comparisons can be avoided as chunks that are very different will never end up next to each other in the list of ordered chunks.

3.7.2. Determining similarity

Once the chunks with similar hash values have been identified, the similarity of two documents is calculated using the common chunks information. There is a lot of information that can be considered when determining this value and the key is to derive a measure that gives a realistic value using all the available information. The information available includes the chunks that are related and exactly how related each chunk is. At this point related work was analysed to see the measures used by other software and whether they could be applied to this situation.

Two measures were analysed on their appropriateness for this software. The first was the Jaccard index [7] which is used by Charikar in the Simhash paper [8] and is used for finding the similarity between two sets. Each set would be a document and the elements of the set would be the paragraphs contained in the document. The second measure is the Euclidean distance [9] which determines the similarity of two objects based on the square root of the sum of squares similar to how Pythagoras works. The objects would be documents and each point in the object would be a paragraph. The Jaccard index was chosen due to it being the one identified by Charikar as the most appropriate for the Simhash algorithm.

During implementation a naive similarity measure was tested and also a similarity measure using the Jaccard index to ensure it gave the improved accuracy required. The two similarity measures are covered in the following paragraphs. The final paragraph in this section covers the combining of semantic and version similarity into a single score and is the same whether using the naive solution or the Jaccard similarity measure.

Initial similarity measure

The first step is to find a value for the version relationship between two documents. The following formula calculates this and considers both the number of chunks that are considered versions and exactly how close in version the chunks are. The first part of the formula is looking at what percent-

age of the chunks are shared between the documents. This value is divided by the number of chunks in the longer document. The values were tested using the number of chunks in the smaller document but the values were far too high. A document with a single paragraph which is also in a document with 100 paragraphs would show 100% of the paragraphs are shared which is not a useful value. The second part of the equation considers exactly how similar each of the chunks are that are within the required closeness threshold. This part of the formula is important as two documents may have many common paragraphs within the closeness threshold but each is slightly altered. This second part of the formula will give high similarity values for documents with paragraphs having little or no alterations.

Version similarity = (# similar chunks in version / # chunks in longer document)

* (1 - (average bit difference of chunks within closeness required / closeness required)

The next step is a similar calculation but for the semantic relationship between two documents. The only difference is that chunks are related by topic rather than in version.

Semantic similarity= (# chunks with similar topic / # chunks in longer document)

* (1 - (average bit difference of chunks within closeness required / closeness required)

Obviously the similarity values calculated above will vary with different closeness thresholds used. The closeness threshold is used directly in the second line of each formula, and also will affect the number of chunks that are determined as similar in the top line. This is fine and the software user needs to be aware of this and the closeness threshold kept constant while testing the entire document corpus.

Similarity measure incorporating Jaccard index

Research suggested using the Jaccard index [7] which was similar in some ways to the naive method but should give a more realistic estimate of the similarity between two documents. The Jaccard measure states that the similarity between two sets is given by the intersection of the sets divided by the union of the sets. The following formula represents this measure.

$$sim(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

To apply this to document versions, the similarity would be the number of paragraphs that are versions of each other divided by the sum of paragraphs from the two documents less the number of paragraphs that are versions. This is fairly similar to the initial naive measure which divides the number of similar paragraphs by the number of paragraphs in the longer document. The Jaccard measure should improve accuracy as it effectively averages the lengths of the documents.

The part of the similarity measure that needed finalizing was how to incorporate the degree of similarity between each paragraph. The Jaccard index works with the number of related paragraphs but does not consider how related each paragraph is. It was decided that the Jaccard index could also be applied to find the degree of similarity between two paragraphs as it is still a comparison of two sets. The naive implementation used the average number of bits different divided by the threshold for required bit difference. It was adjusted to also use the Jaccard index so the number of bits com-

mon in the hash values for two paragraphs was divided be the union of bits in hash values. So if only five bits were different between two hash values. The Jaccard value would be 59 / (64 + 64 - 59) = 49/ 69. This value is higher than the values calculated using the naive formula so paragraphs will receive higher scores.

The final question was whether to only consider the degree of similarity in paragraphs that are found to be versions (as used in the naive approach) or to consider the similarity for all paragraphs in the document. It was decided that the only paragraphs of interest were the ones that were considered versions and the measure did not consider how different the non-version paragraphs were. As a result the threshold chosen to identify paragraph versions becomes crucial in calculating the overall similarity between two documents. Testing of the naive measure was done using a threshold of five. The testing for the Jaccard similarity measure was done using the value seven which was determined as the highest value that could be used without introducing false positives. Even though the bit difference threshold is adjustable, this is not an issue as it lets the user find out the similarity of two documents based on how similar they consider similar to be. Seven is a good value to use for consistency but it is up to the user.

Overall similarity value

The final calculation considers both the semantic similarity and the similarity in version to output an overall similarity value. If the version similarity is greater or equal to 0.1, then this relationship is considered high enough and should be returned. Version scores are shown between 0.5-1 so the score is divided by two and added to 0.5. If the version score is less than 0.1 then it is considered unimportant so the semantic score is used instead. This is a value between 0-0.5 so the semantic value calculated above is divided by two. Figure 10 shows the calculation for overall similarity.

```
if(similarityVersion >= 0.1) {
similarityTotal = 0.5 + (similarityVersion/2);
}
else {
similarityTotal = .5 * similaritySemantic;
}
```

Figure 10: Calculation for overall similarity

3.8. Results output and storage

The final stage is to display and store the document similarity information. The results should be stored in a way so they can be later recalled back into the software.

This component is similar to the processed document storage and has the option of text file storage or database storage. As stated earlier the accuracy of the similarity measure is the main focus of the thesis so the text file storage is used. Each relationship is displayed with the ID's of the two documents being compared as well as the overall similarity value. Statistics are shown for how the similarity value was calculated. Figure 11 shows the output for two input documents with ID's of 2 and 1. These two documents have almost no relationship. Two paragraphs were found to have a very small version relationship but this was too small a percentage of the overall document. No two paragraphs were found to be similar in topic.

```
Relationship between documents 2 and 1. Paragraphs similar in version are:(13,8 with BD: 10)(13,9 with BD: 10)
                    paragraphs similar in topic are:
                    length of paragraphs = (20,14)
                    number similar in version = 2 average distance = 10.0
                    number similar in topic = 0 average distance =0.0
                    total similarity is 0.0
```

Figure 11: Relationship output for documents 2 and 1

3.9. System Control

The system control component controls the entire operation of the system. A class was written whose role is to instantiate each component and to pass the information between each. The control determined how the system handles the input of many files. The components of the system are written to handle a single input file and compare it with any documents in the corpus. It was decided that the system would pass files in one at a time and each would be compared against each document already stored. This method only requires one call per document to the to the process that finds relationships on a single input document. Each document has a chance to be compared against every other using this method. A single input directory is required as a parameter and the system loads in one document at a time until all of the documents in the specified directory have been compared.

Parameters are required at different places within the software and changing the parameters within the code can be slow for testing. Consequently the parameters have been extracted into a single text file that is required as a parameter to the main control class. A different text file can be easily specified to make the software operate with different parameters.

During testing, a number of settings arose that would be useful for testing the system fully and these were added to the input text file. By the end the software could be run in many different forms with only altering this text file with parameters as shown in Figure 12.

Parameter	Description
Doc Storage	Location to store the processed documents
Rel Storage	Location to store the relationships between documents
Input	Input directory for documents
Bit Difference	Number of bits difference allowed for versions
Hash Size	Number of bits to use for hash size
Shingle Size	Size in letters of the shingles for simhash
Delim	Symbol to use for separating fields in output text file
Statistics	Display statistics as well as the overall similarity measure
Only Version	Only look for versions of documents (no semantic relationships)
Close documents	Only return relationships for close documents (ignore documents with no relationship)

Figure 12: Parameters set in input file

4. Evaluation of preferred solution

This section covers testing and evaluation of the software system on real document corpuses.

The first part covers the document corpuses used in testing. The evaluation includes two parts: parameter testing and evaluating document versions. The evaluation of the parameter settings tested

different configurations of the similarity measure to achieve reliable results. The evaluation on document versions was first done using the naive similarity measure on the controlled document corpus and the issues identified. The Jaccard similarity measure was analysed on its ability to correct these issues using the controlled document corpus. The most accurate measure was then tested on the duplicate document corpus to ensure the software could handle most real life situations. Finally the most accurate parameter settings and similarity measure are summarised.

4.1. Document collections and characteristics

Three document corpuses are used to test the software created. Two of the document corpuses are designed to test the reliability of the software when finding document versions. The final corpus is designed for testing semantically related documents but also used in parameter evaluation. The controlled version corpus is a well-controlled corpus created from scratch with various changes made. This corpus is useful to isolate each of the possible changes that occur between document versions. The duplicate document corpus is a real corpus with text files that have been extracted from word and PDF files. The final corpus contains news articles on a range of topics. The articles are brief with short paragraphs.

4.1.1. Possible version changes

A version document is one that develops from another document and a number of alterations are common. This list identifies the changes that were tested within the corpus and hopefully the software is able to identify them.

- Adding/removing/replacing a word in a sentence.
- Changing the word order in a sentence.
- Adding/removing/replacing a sentence in a paragraph.
- Changing sentence order in a paragraph.
- Adding/removing/replacing paragraphs within the document.
- Changing paragraph order in the document.
- Splitting a paragraph into two paragraphs or merging two paragraphs.
- A combination of the changes above.

4.1.2. Controlled version corpus

It was difficult to find an appropriate corpus to use to test the software for finding document versions so a corpus was created from scratch. This has the benefit that the variations between documents can be controlled so every kind of variation in document versions can be tested. In the end a corpus was found for testing versions but the bulk of the testing was done on this controlled corpus. Figures 13-18 show how the documents were created by applying modifications as described in the previous section. Each document under a bold document originated from the bold document but with the changes described.

ID 0: Business related news article	
ID 1:	Deleted two words from each paragraph.
ID 2:	Added two word in each paragraph
ID 3:	Replaced one word in each paragraph with a constant word
ID 4:	Swapped position of two groups containing two words in each paragraph

Figure 13: Summary of documents 0-4 in controlled version corpus.

ID 5: Entertainment article with long paragraphs	
ID 6:	Added an extra sentence at different positions in each paragraph
ID 7:	Deleted one sentence from each paragraph
ID 8:	Changed the sentence order within each paragraph.
ID 9:	Replaced one sentence in each paragraph

Figure 14: Summary of documents 5-9 in controlled version corpus.

ID 10: Lifestyle article with long paragraphs.	
ID 11:	Removed two paragraphs from middle of document
ID 12:	Removed two paragraphs from end of document
ID 13:	Added two paragraphs to the end of the document
ID 14:	Added two paragraphs to middle of the document

Figure 15: Summary of documents 10-14 in controlled version corpus.

ID 15: National news article	
ID 16:	Added two words and removed two words from each paragraph
ID 17:	Added a word to each paragraph and deleted 3 paragraphs
ID 18:	Joined a number of paragraphs into single paragraphs
ID 19:	Removed 2 words to each paragraph and deleted 3 paragraphs.

Figure 16: Summary of documents 15-19 in controlled version corpus.

ID 20: Sports article	
ID 21:	Removed two non-adjacent words from each paragraph
ID 22:	Added two words non-adjacent in each paragraph
ID 23:	Added two paragraphs and removed two paragraphs
ID 24:	Deleted two paragraphs and joined paragraphs together

Figure 17: Summary of documents 20-24 in controlled version corpus.

ID 25: News article on technology	
ID 26:	Twice the original document
ID 27:	Changed the paragraph order
ID 28:	Separated some of the paragraphs into multiple paragraphs.

Figure 18: Summary of documents 25-28 in controlled version corpus.

4.1.3. Duplicate version corpus

Pingar provided a corpus to test the accuracy for finding document versions [10]. The documents were created by extracting text files from word and PDF documents. The corpus was generated using existing software developed by Pingar and the corpus represents real-world documents. The documents have a blank line between paragraphs so that they can be easily processed by the software. A few issues arose in the preparation of the corpus and some of the files have unusual lines such as *" . It also"* on a single line. The corpus contains duplicate documents as well as version documents and plenty of documents with no relationship at all.

4.1.4. Semantic document corpus

The final document corpus used was another from Pingar [11]. Figure 19 shows the topics of the documents in the corpus. The overall topic section is shown but just because two documents are both in the same section does not mean that they should be found as related semantically, they may be completely different.

Documents	Topic
0-12	Business
13-26	Entertainment
27-35	Lifestyle
36-45	Masthead
46-60	National
61-72	Sport
73-85	Technology
86-91	Travel
92-99	World

Figure 19: Summary of document topics in semantic document corpus.

4.2. Testing software parameters

The software was written with a number of adjustable parameters that needed to be set to achieve best performance. The algorithm is tested with different parameters to ensure that the test for version does not find documents to be versions of each other if they are not versions. This means testing the similarity measure with different parameters to see which setup gives the most unique hash values and highest chance of different input phrases being hashed to different values. The algorithm setup determined from this testing is then used in the evaluation of the similarity measure for version similarity. To do this the software is run on the semantic document corpus. No documents in the corpus are versions so this test ensures that no documents should be found as versions that are not versions. Initial tests using a hash size of 32-bit produced poor results. No two documents were shown as closely related in version but there were over 100 relationships generated showing that at least one paragraph in a document is a version of a paragraph in another document. Upon close investigation it was discovered that only one of these paragraphs should be versions. Two documents in the entertainment section share a paragraph that is identical in each document. The rest of the relationships found are false positives. The validity of any relationship found is meaningless if the software also finds many paragraphs to be related that should not be.

4.2.1. Modifying bit difference

The first tests are done using a bit difference of 5, so paragraphs are considered as versions if the hash values calculated are within 5 bits of each other. It is then considered what happens when the bit difference is reduced to 2. This should mean that only very closely related paragraphs are found as versions. Testing is carried out but the problem is not fixed. Many of the relationships found have identical hash values despite the paragraphs being very different. Also the very small bit difference means that some sentences that should be found as related could be missed.

4.2.2. Modifying length of shingles

Testing was carried out on the length of the shingles used in the Simhash algorithm. The initial implementation used a shingle size of two, so each input to Simhash (a paragraph) was broken into

unique two letter shingles. If a longer shingle is used, there are more variations possible in shingle value and less duplicate shingles. The idea was that the longer shingle should better represent the uniqueness of each paragraph. If the words "there" and "threw" are hashed using a shingle size of two, then "th" and "re" would be common shingles between the words. If four letter shingles are used then there are no common shingles so the words would appear less similar. This hypothesis still was not successful as a number of paragraphs were still shown to be versions with this new shingle size of four. The paragraphs related were different to the ones found previously but still incorrect paragraph relationships were found. If the required bit difference is set to zero when using four letter shingles, then no incorrect relationships are found. This means that using four letter shingles the paragraphs are all identified uniquely, but some are still considered very similar which does not help. This is an improvement on two letter shingles as many paragraphs give identical Simhash values using two letters. This still is not helpful though as using a bit difference of zero would not find genuine versions of paragraphs with a few word differences.

It is then considered whether the software could be run two times with different shingle sizes used each time as no two paragraphs were found as related when using both four letter shingles and two letter shingles. This is a slightly more complicated approach but would give correct results.

4.2.3. Modifying Hash Size

Using different length shingles is useful and a method could be created to give good accuracy, but other possibilities also need to be tested. The number of bits in the hash values is changed from 32 to 64 and the results tested. It is assumed that 64 bit values would likely give more accurate results as the larger hash size means less chance of two difference phrases being hashed to similar values as far more hash values are available. This increase in hash size gives significantly better results. Using a hash size of 64 and looking for paragraphs with 2 bits difference returned only the one paragraph relation that it should. Testing is then done on what bit difference can be specified to still only find this single correct relationship. If the bit difference can be increased then it improves the chance of finding genuine paragraph versions while still not finding false positives. It was found that the bit difference could be increased to 4 and only the single relationship returned. When the difference is increased to 5 then 5 paragraphs are found as related that should not be. This was useful as the higher the bit difference, the more likely that paragraphs that are genuinely related will be found.

After finding that a hash size of 64 is more accurate, it was then considered what shingle size to use for best accuracy. Earlier it was discovered that a longer hash size gave more accurate predictions but it is necessary to test exactly which shingle size works best with a 64 bit hash value. With a 64 bit hash and shingle size of four, the bit difference could be increased from five to seven and only the one correct paragraph relationship is returned. This is better than two letter shingles as there is greater margin for the software to find genuinely related paragraphs. When the shingle size is increased even further to six, the results are not as good as more incorrect paragraphs are returned. It was settled that four was the most accurate shingle size to use.

As stated previously, a very accurate algorithm can be created using a combination of shingle sizes. If two paragraphs have similar hash values when using two, three and four letters in shingles then the chance they are related is very high. This was not implemented in the time available but would be a useful improvement to test.

4.3. Testing naive measure

The following testing is done on the controlled version corpus. The corpus contains a range of different adjustments made to documents with the aim of finding which break the similarity measure and in which areas the measure performs well. The values for similarity are between 0.5-1 as this is the range that has been allocated for version relationships.

Figure 20 shows the similarity for documents 1-4 as compared to document 0 and all 4 documents have been identified as versions of document 0. The scores given for all four documents are similar and low. As all the documents in this group have had minimal changes made, the similarity measure should be adjusted to give higher readings. Deleting, adding or replacing two words gave similar scores as expected (documents 1, 2, 3). Swapping groups of words (document 4) gives a similar score but perhaps it would be higher with a smaller shingle size. A smaller shingle size has more weighting on the words themselves rather than the order as no shingle will contain letters from two words in the one shingle. Testing is done on document 4 compared to document 0 using a shingle size of 2 and as expected the similarity measure is much higher (0.8725). When the shingle size is smaller, the order of the words has much less impact.

	Document 0
Document 1	0.575
Document 2	0.5525
Document 3	0.5875
Document 4	0.595

Figure 20: Version testing involving document 0

Figure 21 shows the similarity for documents 6-9 compared to document 5. Documents 6 and 7 have had a sentence added or deleted from each paragraph and receive similar scores. Again the score is relatively low but this is fine as adding/removing an entire sentence is a large change to a paragraph. Document 8 involves sentence reordering and is expected to receive a high score as the content is still the same and testing confirmed this hypothesis. Simhash is not concerned with the order of the text but rather the presence of the small shingles. As stated above, this measure would be higher again if 2 letter shingles were used rather than 4 letter shingles. Document 9 has a single sentence altered in each paragraph so the score should be similar to documents 6 and 7 which proved to be true.

	Document 5
Document 6	0.633
Document 7	0.6
Document 8	0.76667
Document 9	0.5833

Figure 21: Version testing involving document 5

Figure 22 shows testing done on documents 11-14 as compared to document 10. These tests involve paragraphs adjusted as a whole rather than the content within each paragraph. The naive similarity score is designed such that documents sharing exact paragraphs score highly so these documents were expected to score well. Testing shows that all 4 documents received similar high scores. Documents 11 and 12 remove paragraphs from different locations and the score shows that the posi-

tion of removal has no impact. The same applies for documents 13 and 14 which involve the addition of paragraphs. The high scores awarded are justified as two documents are close in version if one only has 2 paragraphs added/removed. The Jaccard index will hopefully award similar scores for documents 1 and 11. Removing words from every paragraph should be similar to removing an entire paragraph and currently they are quite different. The score for adding paragraphs is very slightly higher than removal as the fraction of similar paragraphs for 11 and 12 is for example 14/16 whereas the fraction for 13 and 14 is 16/18. The two paragraphs that are different make up a smaller percentage of the document when there is a greater number of paragraphs.

	Document 10
Document 11	0.875
Document 12	0.875
Document 13	0.9
Document 14	0.9

Figure 22: Version testing involving document 10

A concern with the naive document version similarity measure is that none of documents 16-19 are found to be versions of document 15. These four documents all have a combination of changes made to them and it seems that the changes are too much for the naive measure to handle. This is not necessarily a bad thing and the level of changes could be enough to justify that the documents are no longer versions. Document 16 has two words removed from each paragraph as well as two words added and the change is too great for the software. A single paragraph is found as related which is not a big enough percentage of the document. The paragraphs are all fairly short and longer paragraphs would likely be shown as versions with the same modification as the changes will be a smaller percentage of each paragraph. Document 17 has one word added to each paragraph and 3 entire paragraphs deleted which is a fairly major change. Eight paragraphs are found to be common between this document and document 15 but this is still not a high enough percentage of common paragraphs and the bit difference is also high for each paragraph. Document 19 involves deleting words and paragraphs and again testing shows the adjustments to be too great to be considered versions.

Document 18 has a number of paragraphs joined and is the modification that the software will likely struggle with. The software works with the similarity of each paragraph and separating paragraphs in two creates two quite different paragraphs. The software did struggle with this document and it is found to have no similarity to document 15. This is a major concern and the similarity measure should be able to identify this. The obvious approach is to analyse the document on a sentence level as well as paragraph level. When the document is analysed by sentence (chunks separated by full stops) then splitting paragraphs in two does not affect the sentences present in the document. This is another modification which is only described in theory and not implemented as it would require a lot of changes and also run slower. If the document is analysed both on sentence and paragraph level then the software can get a good representation of the modifications made between two documents. Analysing the document on a sentence level will increase the accuracy but also increase the run time as the data to process increases. If each paragraph has on average 5 sentences then analysing at sentence level means five times as many hash values to calculate and many more comparisons. The comparisons involves each chunk being checked against every other chunk so this would increase the number of comparisons by a factor of 25.

Figure 23 shows documents 21-24 compared to documents 20. Document 21 has two non-adjacent words removed from each paragraph which should still be found as a version. Testing found this document to be a version and the interesting part is that the relationship is similar but not quite as strong as for document 1 which involves two adjacent words being removed from each paragraph. This score is very slightly lower as removing two non-adjacent words affects 4 neighbouring words rather than two when the words are adjacent thus more shingles are affected. Document 22 adds two non-adjacent words to each paragraph and is found as a version for similar reasons as document 21. Document 23 has two paragraphs added and two paragraphs removed and a high score is expected as previous testing shows modifications at paragraph level to score highly. Testing confirms this as the similarity measure uses a multiplier involving the average bit difference between paragraphs and if many paragraphs are unchanged, the bit difference is 0 so the multiplier is maximum. As stated previously the similarity measure using the Jaccard index should hopefully ensure that word removal from each paragraph is closer in value to removing an entire paragraph. Document 24 involves joining paragraphs which as discussed earlier the software struggles with. A suggested fix was mentioned above.

	Document 20
Document 21	0.564285714
Document 22	0.55
Document 23	0.785714286
Document 24	0

Figure 23: Version testing involving document 20

Figure 24 shows documents 26-28 compared to document 25 in version. Document 26 is document 25 twice and scores a similarity of 1. This is too high as although all the content of one document is in the other, the other contains that content twice. This is perhaps not important as it is an unlikely situation. Document 27 involves reordering the paragraphs in the document and is expected to score very highly. Testing gave a score of 1.0 but perhaps the algorithm should consider reordering in some form as it is obvious that the documents are not duplicates. Document 28 tests whether paragraphs could be split into two and still be found as versions. Previous tests show that the software struggles when paragraphs are merged but it appears to handle the situation when paragraphs are split. In fact merging and splitting paragraphs should give identical results as the similarity analysis is symmetrical and it does not matter if the first document is compared to the second or the second is compared to the first. Upon closer inspection it becomes clearer. Documents 18 and 15 are found as related as a single paragraph is found as related between the documents that has not been merged with anything else. The document has 20 paragraphs so this is 1/20 of the document. Documents 28 and 25 also have a single common paragraph but the documents are much shorter so the proportion of similar paragraphs is greater so a version relationship was found.

	Document 25
Document 26	1
Document 27	1
Document 28	0.55

Figure 24: Version testing involving document 25

4.3.1. Issues with preferred solution

Most of the document variations are picked up as required by the software. When documents have a small number of word additions/ removals/replaces to each paragraph then they are correctly identified as versions. When the number of changes becomes too high (such as deleting words and deleting paragraphs) then documents are not identified as versions. This is as expected as they no longer are versions with too many adjustments. It is somewhat unexpected that adding/removing entire sentences scores better than adding/removing two words from each paragraph. This is possibly because of document structure and the results of tests on different documents should not be compared. When testing adding/removing sentences, the paragraphs were long and contained many sentences. When testing adding/removing words the paragraphs tended to contain single sentences.

Removing/adding paragraphs is handled well and all document versions identified correctly. The score for changing paragraphs is slightly too high and this will need to be changed so that word additions/removals score similarly to paragraph adjustments. Issues arose with joining paragraphs or separating paragraphs. The calculation will not find any paragraphs to be related as the software analyses the documents on a paragraph level and the paragraphs are now much different. This is not such a big problem as usually entire paragraphs will not be merged but part of a paragraph moved to another in which case each of the paragraphs will still be identified as versions. This situation was tested by the documents that have entire sentences added/removed from paragraphs.

Figure 25 summarises the changes made and their effect on the similarity score.

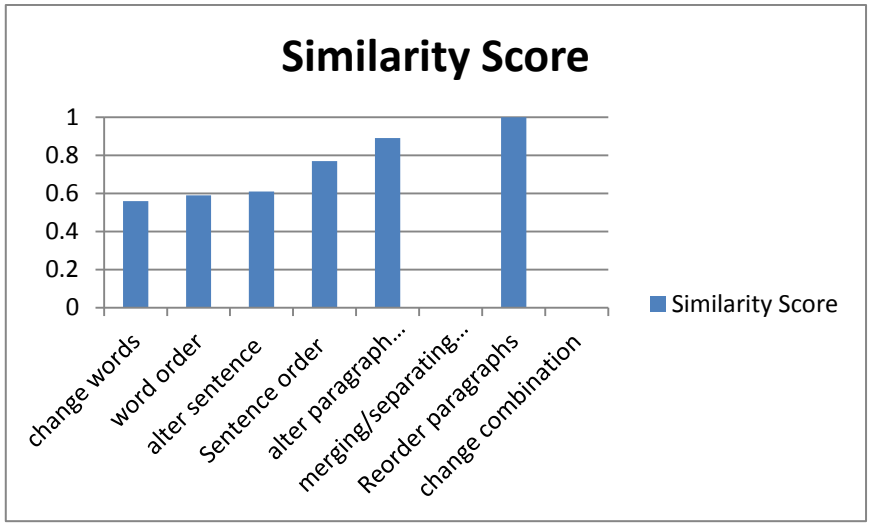

Figure 25: Summary of document adjustments

4.3.2. Key criteria for next stage of testing

Testing of the naive similarity measure identified three main criteria that the Jaccard similarity measure should meet. The measure should give similar scores for removing/replacing words as for removing/replacing paragraphs. Replacing words should score higher than replacing sentences as currently it is the other way around. Finally the measure can hopefully handle the merging and separating of paragraphs.

4.4. Changes to the preferred solution

A similarity measure using the Jaccard Index is tested on how well it handles the situations that the naive measure struggled with. One of the parts that needed improvement was the ability to find

documents that have paragraphs merged/separated. This part will only be solved theoretically using the suggested improvement above analysing the document at sentence level as well as paragraph level. The other improvements can hopefully be solved by this improved similarity measure.

4.5. Retesting of preferred solution

The software is tested on the same controlled version corpus using the Jaccard similarity measure to consider the number of related paragraphs as well as the degree of similarity between each paragraph. The bit difference is also increased to seven. The overall consequence should be higher similarity values. In the previous testing it was found that often the similarity output was too low, particularly for word changes and hopefully this new measure performs better. The similarities are shown in Figure 26 and the similarity values are higher than those when using the naive measure. Of particular note is that the similarities are much higher for documents 1-4 which only change a few words in each paragraph. This is one of the required criteria for the Jaccard index measure. Another good feature is that changing a few words scores higher than changing an entire sentence which makes much more sense. A result of the higher scores is that documents 17 and 19 which were previously identified as not versions of document 15 are now found as versions. These two documents should be found as versions and now the software has found a good balance of what is a version and what is not as documents 16 and 18 contained fairly major changes and are not identified as versions.

	1	2	3	4	6	7	8	9	11	12	13	14
Naive measure	0.58	0.55	0.59	0.6	0.63	0.6	0.77	0.58	0.88	0.88	0.9	0.9
Jaccard measure	0.76	0.78	0.74	0.82	0.72	0.65	0.96	0.59	0.88	0.88	0.9	0.9

	16	17	18	19	21	22	23	24	26	27	28
Naive measure	0	0	0	0	0.56	0.55	0.79	0	1	1	0.55
Jaccard measure	0	0.63	0	0.61	0.65	0.74	0.72	0.56	1	1	0.57

Figure 26: Similarity values for naive measure and measure incorporating Jaccard index

4.6. Testing on duplicate version corpus

The similarity measure using the Jaccard index performs much better on the controlled version corpus. The next step is to test the software on a real document corpus. The corpus used is the duplicate version corpus [10]. It is important to test the software on this corpus to ensure the software does not break as soon as it is used on a corpus that was not created specifically for this software. The duplicate version corpus was created using a new line character to separate paragraphs so the software could process it correctly. The corpus highlighted areas where the software is fragile which will need improvement if the software is to be relied on.

No absolute value is available for how similar two documents should be so document similarity is checked manually (often the name is a giveaway if one is called document1_old and one is document1_new) and the software run to see how accurate it is. Two of the documents in the corpus are exact copies and the software identified this by giving the relationship a similarity score of 1.0. One important use of the software is to find exact duplicate documents so it is good that the software is able to carry out this task.

One area where the software struggles is with very short paragraphs. Some of the documents are unusual and one has "the" as the only word on a line so the chunker has identified this as a para-

graph. Another document has a full stop alone on a line and the full stop seems to have hashed to a similar value to "the" so these two paragraphs are found as versions. There are two solutions to this issue. Firstly the software would work fine if the input documents were cleaned so paragraphs appeared on single lines. A more robust method would be to introduce a weighting to each paragraph so its effect on the overall similarity value is proportional to the length of the paragraph. This way the very short "mistake" paragraphs will not stop the software being accurate.

Another issue that surfaced while testing on the duplicate version corpus is common paragraphs that match more than one paragraph from another document. One document contains the words "Open Issue" on a single line 4 times and the chunker determines each to be a paragraph. Another document also had 4 sections with "open issue" as a header. Each of the paragraphs is mapped to all of the paragraphs in the other document so 16 relationships are introduced. The number of common paragraphs between the documents is higher than the number of paragraphs in a single document. This could be fixed however it will take a lot of coding and will only be described here and not implemented. It is important that each paragraph in a document is only determined as a version of a single paragraph in another document. When a paragraph is compared against every other paragraph in another document, only the paragraph with the highest similarity is the one that it remembers as a version. The reason it will not be implemented yet is that it gets more complicated still. Initially paragraph 1 from document A may be found to be a version of paragraph 2 from document B. The next step is searching for versions using paragraph 2 of document A. It may found that paragraph 2 of document A is a closer match to paragraph 2 of document B than paragraph 1 was. This situation is shown in Figure 27. Each paragraph should only be matched once per document and this situation would mean that the relationship found between paragraph 1 of document A and paragraph 2 of document B would need to be removed. This could all be implemented with a list of relationships present for each paragraph and at the end an algorithm will choose the relationship with the highest similarity and ensure each paragraph is only mapped once.

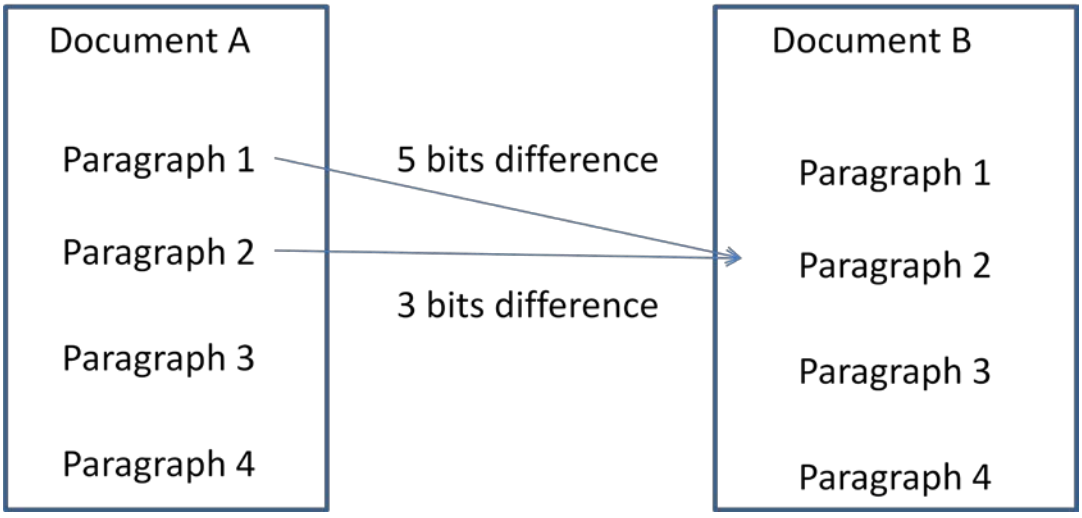

Figure 27: Finding paragraph versions between documents A and B

Many documents contain common lines such as "input and output" which are treated as paragraphs so the documents are found to be versions. This will always be difficult to avoid as many documents will use common headings for structure. The solution mentioned above with a lesser weighting for shorted paragraphs should fix this. Again this will not be coded currently but is an important fix.

The next two documents are very similar with one being a newer version of the old one. The newer document has a combination of changes including new paragraphs, extra words in paragraphs and extra sentences in paragraphs. The documents receive a similarity score of 0.87 to show the documents are very similar so the software worked successfully.

Difficulties with the corpus

Some of the issues mentioned here are not particularly important as the documents in the collection have been automatically formatted to the required input and contain some errors. Figure 28 shows the first few lines from one of the text files. The automated tool used for formatting the file has produced some errors so the similarity results are unusual. The similarity between a document containing this section cannot be relied on as very unusual results occur which will not happen with normal documents.

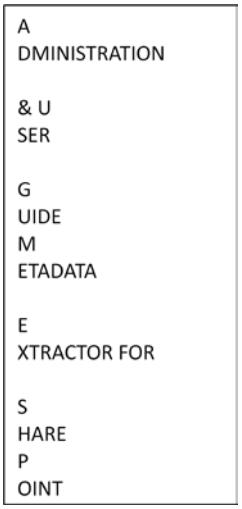

Figure 28: First section of input document.

As well as the unusual sentences, the documents in the corpus tend to have short paragraphs. This means that modifications to paragraphs have a large impact on the hash values and are usually not identified as versions. The software is most effective when the paragraphs are long therefore changes made will be correctly identified as the majority of the paragraph remains unchanged.

Even though some of the documents are dodgy, the main thing to note from the testing is that when documents truly are versions of each other, the similarity score given is much higher than the scores given for documents that are found as versions due to unusual lines in the input documents. Any time two documents are found as versions when they should not, it was due to unusual input rather than a situation that will occur commonly in normal documents.

4.7. Summary of issues and changes

A number of issues have been identified in previous sections along with strategies to fix them that have either been implemented or reasoned theoretically. Figure 29 summarises the issues identified and further information on each of the fixes is covered in the previous section.

Issue	Fix	Coded	Theory
Simhash clashes	Increased hash size and shingle size	✓	
Many comparisons required between documents	More efficient comparisons algorith		✓
Altering words gave low scores	Introduced Jaccard index	✓	
Modifying sentences scored higher than modifying words	Introduced Jaccard index	✓	
Couldn't handle merging/joining paragraphs	Analyse document at sentence level		✓
Paragraph reordering scores 1.0	To award 1.0 the paragraphs must align perfectly		✓
Short paragraphs hurting calculations	Attach a weighting to each paragraph based on length		✓
Paragraphs being mapped to more than one paragraph	Algorithm introduced to map each paragraph to only the most similar paragraph		✓

Figure 29: Issues and resolution strategies

Final configuration and untested improvements

The final configuration of the parameters and implementation for the similarity measure is now finalised. The parameters used are a Simhash length of 64 bits and shingle size of 4. This combination of parameters best represents the uniqueness of each input phrase to output non clashing hash values. The similarity measure uses the Jaccard index both for calculating the number of common paragraphs between documents as well as the degree of similarity between each paragraph.

Three untested improvements should be included in the final configuration. The software should analyse the document at sentence level as well as paragraph level to cope with paragraph separations. The similarity measure should assign a weighting to each paragraph based on its length. Finally the software should ensure that each paragraph is mapped to at most one other paragraph.

4.8. Performance summary

The main goal is to find document similarity in version. This was tested thoroughly and gives accurate results.

The speed and disk space required for the software are major strengths. When working with the semantic document corpus of size 100 (average 15 paragraphs), the documents could be stored in a text file of size 40KB which is very small. If the same document size is assumed then 100,000 documents would require 43 MB in space. The documents used are fairly small but 100,000 documents should not require more than 1 GB of storage. The algorithm is fast due to the comparisons involving only the analysis of two 64 bit numbers. The run time could be reduced further using the speed up method described in Section 3.7.1. assuming the document corpus fits in memory. The time taken to test for versions on the semantic document corpus of size 100 was 48 seconds. Brief testing for finding semantic similarity found that the same corpus took 4 minutes so 80% of this time was finding semantically related documents. This took longer due to the time taken to retrieve entities from the Pingar API.

The similarity measure using the Jaccard index along with a few small changes gave improved accuracy. 21 of the 23 files in the controlled document corpus were identified as versions and the remaining 2 had large modifications made from the original. The naive similarity measure found 19 of the 23 to be versions. The values returned were also more realistic than for the naive measure such as word modifications in each paragraph scoring 0.77 compared to 0.58. Except for a few very unusual situations the software can be relied upon to accurately find document versions in a corpus.

One of the issues encountered was the struggle to find a good real life corpus for testing document versions. In the end a duplicate version corpus was found but it was inconsistent in its formatting. This is partly due to the software needing fairly specific input forcing the documents to be prepared

in a certain way. The controlled version corpus works fine for testing and every imaginable variation could be tested but it would be good to do further testing on real document corpuses.

4.9. Future work

This section covers work that can be done in the future to further improve the software. Further testing is possible for document versions by considering the Euclidean distance as a similarity measure. The other major section of future work is to evaluate the ability of the software to find semantically similar documents.

4.9.1. Further testing for document versions

Testing of document versions was fairly thorough. One thing that could be interesting is to test the effect of using the Euclidean distance as the similarity measure. Charikar [8] states that the Jaccard index will work the best with the Simhash algorithm but perhaps a similarity measure using the Euclidean distance should be tested as well. This change would be possible as the software would be the same except for the final calculation. Any of the theoretical solutions could also be implemented to further improve the software. Two theoretical changes would appear to give the greatest improvement. Testing should be done on analysing the document at sentence level as well as paragraph level. Finally testing should be done giving each paragraph a weighting based on the length of the paragraph.

4.9.2. Testing for semantically related documents

Testing for semantic relationships was done on the semantic document corpus described in Section 4.1.2. The corpus has news articles in 9 sections such as sport and health. A gold standard of semantically similar documents has been created through manual inspection. The testing was limited initially due to the number of calls possible to the Pingar API per month. This issue was resolved but the majority of the testing left for future work. On the first testing run the software worked as expected and the entities were extracted from each paragraph and then hashed and stored.

Issues arose in the testing as not as many entities were extracted from each paragraph as expected so it was difficult to match the topics between paragraphs. This is due to the paragraphs containing few key words and possibly because test sentences used previously were rich in content as they were often titles of articles. It could also be possible to alter the settings in the API to extract more entities. The semantic document corpus has fairly short paragraphs (often one sentence) and most of the paragraphs have 1 or 2 extracted entities with many having no entities at all. As a result the hash values for each paragraph contain little or no information. This means very few paragraphs are found to have the same topic and no documents are found as semantically related. A paragraph with a single entity would only be related to another paragraph if they share the same extracted entity. If a paragraph has 5 or 6 entities extracted then the chance of similar paragraphs being found is higher as another paragraph would only need to have 2 or 3 of the same concepts.

Documents will usually have longer paragraphs meaning more entities will be extracted from each so the software may already perform better on another corpus. There are three other possibilities to investigate that should improve the accuracy for finding semantic documents.

1. Analyse the extracted entities from larger chunks of the document. Entities could be extracted from the entire document or perhaps sections. If the document is broken into introduction, middle and conclusion then the introduction and conclusion will likely contain

many of the important concepts. There will be plenty of entities to hash so more documents will be found as related.

2. Adjust the similarity measure so fewer paragraphs are required to be related for the software to identify documents as semantically related. This is due to it being very difficult for documents to share many paragraphs with similar topics. The existing method of calculating the score based on the percentage of paragraphs that are related gives a very low score.
3. Introduce synonyms for the extracted entities. If few entities are extracted per paragraph then this will not fix the problem but it should improve the accuracy slightly if documents use synonyms for the same concept.

5. Conclusion

The aim was to create software that is able to find document versions. A piece of software was created and tested that could accurately find document versions. The software could be even more accurate using some of the theoretical improvements that are suggested in the report. A more efficient comparisons algorithm was also described which would reduce the run time as long as the corpus fits into memory.

The software has a few issues with stability in that the corpus needs to be prepared in a certain way for the software to effectively find related documents. When the Simhash algorithm uses a hash size of 64 bits and a shingle size of 4 then it will uniquely identify input phrases. The similarity measure is able to accurately identify document versions when adjustments made to documents include word alterations, paragraph alterations and a combination of both. It struggles with the merging of paragraphs but using the suggested fix of analysing at the sentence level this issue will be resolved. The software has some problems with very short paragraphs in the duplicate test corpus but with an adjustment to weight the paragraph based on length then the software would be able to accurately find document versions in most document corpuses.

The software has a feature to find semantically related documents but it is not well-tested. In the testing it struggled due to the small size of each paragraph in the corpus and the lack of entities extracted for each paragraph. This can be improved by analysing the document as a whole or adjusting the calculation for the amount of related paragraphs required for documents to be considered related.

6. References

[1] Fowke, M., Hinze. A., Heese, R. (2013). Text Categorization and Similarity Analysis: Literature review. Working paper 11/2013, Computer Science Department, University of Waikato.

[2] Fowke, M., Hinze. A., Heese, R. (2013) Text Categorization and Similarity Analysis: Similarity measure, Architecture and Design. Working paper 12/2013, Computer Science Department, University of Waikato.

[3] Darling, W. M. (2011, December). A Theoretical and Practical Implementation Tutorial on Topic Modeling and Gibbs Sampling. In Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies (pp. 642-647).

[4] Matpalm. The Simhash algorithm. Retrieved from http://matpalm.com/resemblance/simhash; accessed on 15 October 2013.

[5] Java. (2013). What is Java. Retrieved from www.java.com

[6] Wolfram Math World. (2013) Bell Number. Retrieved from http://mathworld.wolfram.com

[7] Real, R., & Vargas, J. M. (1996). The probabilistic basis of Jaccard's index of similarity. Systematic biology, 45(3), 380-385.

[8] Charikar, M. S. (2002, May). Similarity estimation techniques from rounding algorithms. In Proceedings of the thirty-fourth annual ACM symposium on Theory of computing (pp. 380-388). ACM.

[9] Danielsson, P. E. (1980). Euclidean distance mapping. Computer Graphics and image processing, 14(3), 227-248.

[10] Duplicate document corpus. Provided by Pingar. http://www.pingar.com/about-us

[11] Document Semantics corpus. Provided by Pingar. http://www.pingar.com/about-us