Plagiarism detection by similarity join

Version of August 11, 2009

"Dear Mr Trent: Since you only pretended to write this paper, I only pretended to grade it!"

R. Schellenberger
Plagiarism detection by similarity join

THESIS

submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

in

Media and Knowledge Engineering

by

Rolf Schellenberger
born in Amsterdam, the Netherlands
Abstract

Since the internet is so big and most of its content is public, it is very hard to find out where the information came from originally. There are many websites that publish news articles, so people and organizations can easily lose track of where their articles are reused with or without their permission. This thesis presents a plagiarism detection algorithm that allows us to quickly compare online news articles with a collection of personal news articles and detect plagiarized passages with the same quality as a human. The algorithm uses a basic shingle index and a Signature Tree as a more advanced pre-filtering step to narrow down the viable documents to a query. The algorithm achieves a score of 0.96 precision and 0.94 recall but is too resource intensive to be considered scalable. When only the pre-filtering step is used, it achieves 0.85 precision and recall creating a speedup of nearly one order of magnitude.

Thesis Committee:

Prof Dr. Ir. A.P. de Vries, Faculty EEMCS, TU Delft
Ir. S.T.J. de Bruijn, TEEZIR, Utrecht
Dr. E.A. Hendriks, Faculty EEMCS, TU Delft
Dr. P. Serdyukov, Faculty EEMCS, TU Delft
Dr. P. Cimiano, Faculty EEMCS, TU Delft
Preface

During my Master Thesis, a lot of people stood by me and supported me along the way. I would like to thank all of them and some people in particular. I would like to start with Arjen de Vries for being my supervisor during my Master Thesis and supporting me along the way by giving advice and a lot of positive input where needed. I would like to thank Pavel Serdyukov for his support and feedback during Arjen’s absence. Besides the people from the Delft University of Technology, I would like to say thanks to Wouter Alink for sharing his work on suffix arrays, which I was unfortunately unable to use. I would like to thank my colleagues at TEEZIR; Arthur, Bianca, Krispijn, Thijs, Victor and especially my supervisor, Stefan de Bruijn. I would like to thank you for giving me advice, showing me the company, its software and new possibilities, let me discover the field of information retrieval and focused content, being part of the team, making me laugh and letting me win during a game of Wii tennis after lunch break. Finally, I would like to thank my family, my father, mother, sister and Maaike who always listen to my stories, encourage and support me. I could not have done it without you.

Rolf Schellenberger
Delft, the Netherlands
August 11, 2009
# Contents

1 Introduction ................................................. 1  
   1.1 Define plagiarism .................................. 1  
   1.2 Problem statement ................................ 2  

2 Related work ............................................... 3  
   2.1 Fingerprinting ..................................... 3  
   2.2 Bag of words model ............................... 3  
      2.2.1 Shingles ..................................... 4  
   2.3 Measuring similarity .............................. 5  
   2.4 Sentence retrieval ................................ 6  

3 Scientific paper .......................................... 7  
   3.1 Introduction ....................................... 7  
      3.1.1 Define plagiarism .......................... 7  
   3.2 Related work ....................................... 8  
      3.2.1 Fingerprinting .............................. 8  
      3.2.2 Bag of words model ....................... 8  
      3.2.3 Shingles ..................................... 9  
   3.3 Contribution ......................................... 9  
      3.3.1 Training and test set ..................... 9  
      3.3.2 Tokenizer .................................. 10  
      3.3.3 Shingles .................................... 10  
      3.3.4 Pre-filtering .............................. 10  
   3.4 Results ............................................. 13  
   3.5 Conclusion ........................................... 15  

4 Implementation ............................................ 19  
   4.1 Application design ................................ 19  
   4.2 Plagiarism detection algorithms .................. 19  
      4.2.1 Shingle indexes ............................ 20  
      4.2.2 Signature tree ............................. 21  
   4.3 Testing ............................................. 22  
   4.4 Data set ............................................. 22  
   4.5 Hardware setup ..................................... 24  

5 Additional results and conclusions ...................... 27  
   5.1 Proposed solution .................................. 27  
   5.2 Demands and wishes ................................ 27  
      5.2.1 Demands .................................... 27  
      5.2.2 Wishes ..................................... 27  
   5.3 Planning ............................................ 27  

6 Future work .................................................. 29  

A Class diagrams ............................................. 33  

B Sphinx search configuration .......................... 37  

C RSSFeedCrawler class diagram ....................... 39  

D Definition and planning .............................. 41  
   D.1 Planning ........................................... 41  

vii
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>E Program of demands and wishes</strong></td>
<td></td>
</tr>
<tr>
<td>E.1 Introduction</td>
<td>43</td>
</tr>
<tr>
<td>E.2 Goal</td>
<td>43</td>
</tr>
<tr>
<td>E.3 Reason</td>
<td>43</td>
</tr>
<tr>
<td>E.4 Adjustments</td>
<td>43</td>
</tr>
<tr>
<td>E.5 Demands</td>
<td>43</td>
</tr>
<tr>
<td>E.5.1 Speed</td>
<td>43</td>
</tr>
<tr>
<td>E.5.2 Flexibility</td>
<td>44</td>
</tr>
<tr>
<td>E.5.3 Scalability</td>
<td>44</td>
</tr>
<tr>
<td>E.5.4 Quality</td>
<td>44</td>
</tr>
<tr>
<td>E.6 Wishes</td>
<td>44</td>
</tr>
<tr>
<td>E.6.1 Robustness</td>
<td>45</td>
</tr>
<tr>
<td>E.6.2 Security</td>
<td>45</td>
</tr>
<tr>
<td><strong>F N-grams and Suffix arrays</strong></td>
<td></td>
</tr>
<tr>
<td>F.1 Suffix Arrays</td>
<td>47</td>
</tr>
<tr>
<td>F.2 NGrams</td>
<td>47</td>
</tr>
<tr>
<td>F.3 Compression</td>
<td>47</td>
</tr>
<tr>
<td>F.4 Conclude</td>
<td>48</td>
</tr>
<tr>
<td><strong>G Proposal of a feasible solution</strong></td>
<td>49</td>
</tr>
</tbody>
</table>
Introduction

When you publish a document on your website, how do you know which websites are using your information as their own? Since the internet is so big and most of its content is public, it is very hard to find out where the information came from originally. Companies (like Google) are trying to detect near-duplicate documents on the internet to save storage space and to prevent the same information from showing multiple times. Near-duplicates can be shared content between different sources or altered copies, like redacted or quoted text. When near-duplicates result from copyrighted material and the original source is not mentioned, it is called plagiarism. Document-by-document comparison is sufficient to detect plagiarized documents that are copied from someone else and adjusted to make it look like your own. Documents can be compared with documents based on their content and structure [13, 20, 22]. To improve efficiency of this process, plagiarism detection can then be viewed as a similarity self-join over the corpus of interest, or as the similarity join between a database of copyrighted material and the corpus of interest.

This thesis will focus on detecting online plagiarized news articles. Its goal is to detect near-duplicate news article passages on various websites, so the corpus of interest for this thesis is a collection of online news articles and a database collection of (plain text) news articles. The plagiarism detection program should be able to report the URL's of web pages that contain altered or copied articles. The number of articles that can be processed must be higher than the number of articles that are crawled (downloaded) online per day. Finally, the publishers are informed about the plagiarized content.

Because there are so many websites that publish news articles, people and organizations can easily lose track of the locations where their articles are reused with or without their permission. Some websites will have an agreement with the original source to use a certain amount of articles per day, other websites do not have such an agreement. To keep track of your own online articles and their reuse frequency, a near-duplicate detection application is required. TEEZIR, an information retrieval company located in Utrecht, The Netherlands, is looking for such an application. TEEZIR is specialized in collecting, analyzing and searching (online) documents, therefore this graduation project is done in cooperation with TEEZIR and the Technical University Delft.

1.1 Define plagiarism

When someone is deliberately duplicating copyrighted material to present it as their own, they will try to cover this up by changing the original text, through adding, removing, editing, or reordering text. These modifications make it more difficult to detect plagiarism. In order to detect near-duplicate content, plagiarism must be defined and the similarity between web articles and the source articles should be measured. According to a dictionary, the definition of the word 'plagiarism' is as follows:

'The copying of another person’s ideas, text, or other creative work, and presenting it as one’s own, especially without permission’

Besides a dictionary, the Dutch author law and jurisprudence define plagiarism, but none of them provides exact numbers or thresholds. Therefore it is very difficult to give a mathematical definition. Humans, on the other hand, are quite good at detecting plagiarism in small corpora, but this is a boring, time-consuming, expensive and impossible task when the collection is large (which is the case when crawling the internet). This thesis presents a Signature Tree pre-filter and shingle index approach to detect near-duplicate passages from news articles on any type of website as good as a human can while keeping the computational time low to be able to deal with large and scalable document collections. Unique about this method is that it detects a copy of the text within a larger web page.

---

1 TEEZIR: "Forget about structure, deal with chaos!", http://www.teezir.com
1.2 Problem statement

The problem statement for this thesis is:

'How to detect plagiarized online news articles within acceptable time and quality by using similarity join?'

This can be divided in smaller problems:

1. 'What is plagiarism?'
2. 'What are the usable similarity join functions that meet the requirements?'
3. 'What is the acceptable time?'
4. 'How can we measure quality?'

The main challenge is to achieve high quality without too much computational time on large and scalable document collections, so to focus on scalability, speed and quality. With growing document collections, the solution to this problem needs to be scalable, because more internet articles will be published per day in the future and new articles will be added to the own document collection over time. Quality and speed are very important, because a computer should be able to detect near-duplicate news articles as good as humans can, but also establish this within acceptable time. This might lead to a trade-off between quality and speed.

The rest of this thesis is as follows. First, the related work is discussed. Second, the work of the thesis is presented as a scientific paper. Finally, the implementation work is described in more detail followed by additional results and future work.
2 Related work

The near-duplicate techniques described in this thesis are based on previous work on text search, information retrieval and plagiarism detection. Document comparison can be done at document, paragraph, sentence and word level. Because sentences are too short and paragraphs can be too long, the level for plagiarism comparison lies in between. In the next sections, the previous work and its relevance to news article plagiarism detection is described. To find out what plagiarism detection approach works best, two types are reviewed: data driven and content based.

2.1 Fingerprinting

A data driven way to detect near-duplicate documents is fingerprinting based on the work by Manber [14], Brin, Davis and Garcia-Molina [3], Garcia-Molina and Shivakumar [19] and Manku et al. [15]. They achieved impressive results on large database collections (>150 Gb) with very low incorrect results (3%), so very useful to retrieve the identical or similar documents. Fingerprinting aims to produce a compact description (fingerprint) for each document in the collection. A typical document fingerprint is a sequence of integers or bits representing the content of that document. The fingerprint granularity is the size of the substring used to generate the fingerprint or also called the minutia. The granularity can be the number of characters, words or sentences. The number of minutiae defines the fingerprint resolution. There are various ways to construct fingerprints, but every good fingerprint should satisfy certain properties:

- The cost of computing a fingerprint must be low.
- It must be deterministic: for a given input, it should always generate the same output.
- A good fingerprint function should map the input values as evenly as possible over the output range, so every output value has the same probability.
- Continuity: two input values that slightly differ should be mapped to nearly the same or the exact same output value.

Several different kind of fingerprinting techniques exist [13]. Hash-value based approaches like [5] use only strings whose hash values are multiples of an integer. Position based schemes [4], on the other hand, select strings based on their offset in a document. Conrad et al. [9] and Chowdhury et al. [8] use strings with high inverse document frequency (IDF [24]). The input range is normally larger than the output range, so collisions occur when building a fingerprint. When the output range decreases, storing the integer values requires less memory space, but the chance of collision increases. This means more different input values map to the same output value, which should be avoided.

A fingerprint, for example, is constructed by using the lexicon information from the document collection, containing all distinct words, where every word has its own unique integer value. The fingerprint of a single document is then constructed by using a bit vector where every word in the document is mapped to an index position of the vector and will set the bit to true. A true bit in the fingerprint means the document contains that word one or multiple times. The bit vector index value is found by retrieving the unique id of the word in the document, modulo the vector length. The construction is illustrated in Table 1.

Because of the speed, consistency and high precision, fingerprinting is a useful technique for detecting near-duplicate documents. On the other hand, Hoad and Zobel [13] find that ranking outperforms fingerprinting when they compare fingerprinting with a variant of the cosine similarity where the difference between two documents is measured by using term frequencies.

2.2 Bag of words model

Instead of using the more data driven fingerprinting techniques, a content based search solution can be used to detect near-duplicate documents. The most common search solution is a full-text search [13]. A full-text index contains a list of references to documents and position for each word.
<table>
<thead>
<tr>
<th>Lexicon</th>
<th>Word</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>&quot;this&quot;</td>
</tr>
<tr>
<td>1</td>
<td>&quot;is&quot;</td>
</tr>
<tr>
<td>2</td>
<td>&quot;a&quot;</td>
</tr>
<tr>
<td>3</td>
<td>&quot;short&quot;</td>
</tr>
<tr>
<td>4</td>
<td>&quot;article&quot;</td>
</tr>
<tr>
<td>5</td>
<td>&quot;another&quot;</td>
</tr>
<tr>
<td>6</td>
<td>&quot;not&quot;</td>
</tr>
<tr>
<td>7</td>
<td>&quot;empty&quot;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Document</th>
<th>Content</th>
<th>Words</th>
<th>Fingerprint</th>
</tr>
</thead>
<tbody>
<tr>
<td>Document 1:</td>
<td>&quot;This is a short article&quot;</td>
<td>0, 1, 2, 3, 4</td>
<td>00011111</td>
</tr>
<tr>
<td>Document 2:</td>
<td>&quot;Another short article&quot;</td>
<td>5, 3, 4</td>
<td>00111000</td>
</tr>
<tr>
<td>Document 3:</td>
<td>&quot;This is not empty&quot;</td>
<td>0, 1, 6, 7</td>
<td>11000011</td>
</tr>
</tbody>
</table>

Table 1: Fingerprinting example by using the lexicon

When performing a search query, the search engine examines all words in every document as it tries to match them with the search words supplied by the user. Table 2 shows a small example of a full-text search. When searching for the phrase "what is it", the index will return (0, 2)(0, 1)(0, 0) and (1, 0)(1, 1)(1, 2), where every tuple contains the document identifier and the offset of the word in that document. Only the documents that contain all words are returned. The results show document 0 and 1 as possible matches where document 1 has the words in the same order as the query. The full-text search tries to match the query words exactly to the documents, so it is not very robust when searching and matching near-duplicate documents, because variations in words, like form, time and synonyms can disrupt the retrieval.

<table>
<thead>
<tr>
<th>Word</th>
<th>(Document, Position)</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>(2, 2)</td>
</tr>
<tr>
<td>banana</td>
<td>(2, 3)</td>
</tr>
<tr>
<td>is</td>
<td>(0, 1), (1, 1), (2, 1)</td>
</tr>
<tr>
<td>it</td>
<td>(0, 0), (1, 2), (2, 0)</td>
</tr>
<tr>
<td>what</td>
<td>(0, 2), (1, 0)</td>
</tr>
</tbody>
</table>

Table 2: Full-text index

2.2.1 Shingles

Shingles, contiguous sequences of characters, sometimes called N-grams, can be used instead of words to search a document collection. All shingles of the word "abracadabra" with \( N = 5 \) are shown in Table 3.

Shingles are successfully applied in related near-duplicate detection work [4, 21]. The index size can be a drawback, that has been addressed in previous work. By keeping only a subset, called a sketch, of all possible shingles, the index size is reduced. Frequently occurring shingles are removed [12] or every 25th shingle is kept [5]. To reduce the processing complexity of shingles on even larger collections, the "Super Shingles" were later proposed by Broder [4]. A Super Shingle is actually a sketch made of a sketch. Hoad and Zobel investigated a variety of approaches for filtering good shingles [13]. Reducing the amount of shingles increases the processing speed, but reduces the precision and effectiveness. Using shingles with a sliding window, co-occurring words can be found and merged to a plagiarized sequence [18]. Shingles can be used to find exact matches, but can also deal with the reordering facets of plagiarism without using too much language analysis logics.
Finding shingles in a large document corpus can be extremely efficient by using an inverted file index [27]. Instead of going through all documents before returning the list of matches, an inverted index does the exact opposite by going over the list of distinct shingles and retrieve per shingle the list of documents in which the shingle occurs.

### 2.3 Measuring similarity

Similarity between documents or passages (sketches) is measured in various ways. A well known measurement is the Jaccard overlap measure:

\[
J(A, B) = \frac{|A \cap B|}{|A \cup B|}
\]

Besides the Jaccard overlap measure, the identity measure [13] is used to find different versions of documents. It is created by a combination of similarity measures [13] and some constraints to expresses the difference (or similarity) between two documents:

- High similarity rating: similar documents should contain similar numbers of occurrences of words.
- Differences in the number of occurrences are penalised.
- Near-duplicate documents should have small term frequency differences.
- Near-duplicate documents should have small difference in length.

A good identity measure function is less sensitive to document lengths differences, ranks documents higher in which the term is common in the document, but rare in the collection and give higher weights to rare terms. Because evaluation time is not substantially affected by the form of the similarity measure [26], it is possible to apply similarity measures to very large collections while maintaining a satisfactory execution time as long as the collection statistics are known. Instead of applying the identity measure to a complete document, it can also be applied to passages. The best identity measure function according to [13] is:

\[
\frac{1}{1 + \log_e (1 + |f_d - f_q|)} \sum_{t \in q \cap d} \log_e \left( \frac{N}{f_t} \right) \frac{1 + |f_d,t - f_q,t|}{1 + |f_d,t - f_q,t|}
\]

Where

\[
\begin{align*}
N &= \text{The number of documents in the collection} \\
f_t &= \text{The number of documents containing term } t \\
f_d &= \text{The number of terms in document } d \\
f_q &= \text{The number of terms in query } q
\end{align*}
\]
\[ f_{d,t} = \text{The number of occurrences of term } t \text{ in document } d \]
\[ f_{q,t} = \text{The number of occurrences of term } t \text{ in query } q \]

2.4 Sentence retrieval

Sentence Retrieval is the task of retrieving a relevant sentence in response to a query, a question or a reference sentence [17] based on text analysis. This technique ranks relevant sentences by their similarity to a query or by the relationship between terms and a term set [1, 6, 7]. There are two possible facets of similarity discussed: lexical and structural. Lexical similarity is an indicator of the degree to which two pieces of information discuss the same topic. It covers exact matches, synonyms, related terms and co-occurrence. Structural similarity defines how much two sentence constructions resemble. It covers identical construction, reordering, matching n-grams and matching patterns. People who plagiarize are expected to use any combination of the text variation techniques above.

Not all sentences are relevant to plagiarism detection, because if a sentence contains no 'useful information', it cannot be plagiarized [17], because the sentence can be used by anybody. An example of such a sentence would be: "If I didn’t tell Woodrow Wilson, why would I tell you?". When there are many relevant sentences between two documents, it can suggest plagiarism. Using sentence retrieval techniques has the advantage that you can detect almost any variation of a sentence. This would make it very hard for a person to plagiarize without being detected. The drawback of this approach is its inefficiency and language dependency.

\[ \text{http://www.myfavoritegames.com/dragonball-z/Info/ChatLogs/} \]
\[ \text{Dragonball-Z-GT-ChatLog-BriceArmstrong.htm, August 2006} \]
3 Scientific paper

The next view pages include the scientific paper about this Master Thesis project including the introduction, relevant previous work, contribution, results and conclusion.

Since the internet is so big and most of its content is public, it is very hard to find out where the information came from originally. There are many websites that publish news articles, so people and organizations can easily lose track of where their articles are reused with or without their permission. This paper presents a plagiarism detection algorithm that allows us to quickly compare online news articles with a collection of personal news articles and detect plagiarized passages with the same quality as a human. The algorithm uses a basic shingle index and a Signature Tree as a more advanced pre-filtering step to narrow down the viable documents to a query. The algorithm achieves a score of 0.96 precision and 0.94 recall but is too resource intensive to be considered scalable. When only the pre-filtering step is used, it achieves 0.85 precision and recall creating a speedup of nearly one order of magnitude.

3.1 Introduction

When you publish a document on your website, how do you know which websites are using your information as their own? Since the internet is so big and most of its content is public, it is very hard to find out where the information came from originally. Near-duplicates can be shared content between different sources or altered copies, like redacted or quoted text. When near-duplicates result from copyrighted material and the original source is not mentioned, it is called plagiarism. Document-by-document comparison is sufficient to detect plagiarized documents that are copied from someone else and adjusted to make it look like your own. Documents can be compared with documents based on their content and structure [13, 20, 22]. To improve efficiency of this process, plagiarism detection can then be viewed as a similarity self-join over the corpus of interest, or as the similarity join between a database of copyrighted material and the corpus of interest.

The corpus of interest for this paper is the collection of online news articles and a database collection of (plain text) news articles. Many of the news articles that are published online today can easily be read and copied to use as your own. Publishers would like to keep track of the people (or websites) that use (parts of) their articles, because a fee must be paid when people are reusing copyrighted articles. There are many websites that publish news articles, so people and organizations can easily lose track of where their articles are reused with or without their permission.

3.1.1 Define plagiarism

When someone is deliberately duplicating copyrighted material to present it as their own, they will try to cover this up by changing the original text, through adding, removing, editing, or reordering text. These modifications make it more difficult to detect plagiarism. In order to detect near-duplicate content, plagiarism must be defined and the similarity between online articles and the original articles should be measured. The definition according to a dictionary is as follows:

‘The copying of another person’s ideas, text, or other creative work, and presenting it as one’s own, especially without permission’

Besides a dictionary, the Dutch author law and jurisprudence define plagiarism, but none of them provides exact numbers or thresholds. Therefore it is very difficult to give a mathematical definition. Humans, on the other hand, are quite good at detecting plagiarism in small corpora, but this is a boring, time-consuming, expensive and impossible task when the collection is large (which is the case when crawling the internet). This paper presents a Signature Tree pre-filter and shingle index approach to detect near-duplicate passages from news articles on any type of website as good as a human can while keeping the computational time low to be able to deal with large

4Auteurswet 1912, http://nl.wikisource.org/wiki/Auteurswet_1912_-Hoofdstuk_I#Artikel_15
and scalable document collections. Unique about this method is that it detects a copy of the text within a larger web page.

The rest of this paper is as follows. First, the related work is discussed. Next, the contribution of this paper is explained by going over the used techniques. In chapter 3.4, the results are presented and finally conclusions are drawn.

3.2 Related work

The near-duplicate techniques described in this paper are based on previous work on text search, information retrieval and plagiarism detection. Document comparison can be done at document, paragraph, sentence and word level. Because sentences are too short and paragraphs can be too long, the level for plagiarism comparison lies in between. In the next sections, the previous work and its relevance to news article plagiarism detection is described. To find out what plagiarism detection approach works best, two types are reviewed: data driven and content based.

3.2.1 Fingerprinting

A data driven way to detect near-duplicate documents is fingerprinting based on the work by Manber [14], Brin, Davis and Garcia-Molina [3], Garcia-Molina and Shivakumar [19] and Manku et al. [15]. They achieved impressive results on large database collections (>150 Gb) with very low incorrect results (3%), so very useful to retrieve the identical or similar documents. Fingerprinting aims to produce a compact description (fingerprint) for each document in the collection. A typical document fingerprint is a sequence of integers or bits representing the content of that document. The fingerprint granularity is the size of the substring used to generate the fingerprint or also called the minuti. The granularity can be the number of characters, words or sentences. The number of minuti defines the fingerprint resolution. There are various ways to construct fingerprints, but every good fingerprint should satisfy certain properties:

- The cost of computing a fingerprint must be low.
- It must be deterministic: for a given input, it should always generate the same output.
- A good fingerprint function should map the input values as evenly as possible over the output range, so every output value has the same probability.
- Continuity: two input values that slightly differ should be mapped to nearly the same or the exact same output value.

Several different kind of fingerprinting techniques exist [13]. Hash-value based approaches like [5] use only strings whose hash values are multiples of an integer. Position based schemes [4], on the other hand, select strings based on their offset in a document. Conrad et al. [9] and Chowdhury et al. [8] use strings with high inverse document frequency (IDF [24]). The input range is normally larger than the output range, so collisions occur when building a fingerprint. When the output range decreases, storing the integer values requires less memory space, but the chance of collision increases. This means more different input values map to the same output value, which should be avoided.

Because of the speed, consistency and high precision, fingerprinting is a useful technique for detecting near-duplicate documents. On the other hand, Hoad and Zobel [13] find that ranking outperforms fingerprinting when they compare fingerprinting with a variant of the cosine similarity where the difference between two documents is measured by using term frequencies.

3.2.2 Bag of words model

Instead of using the more data driven fingerprinting techniques, a content based search solution can be used to detect near-duplicate documents. The most common search solution is a full-text search [13]. A full-text index contains a list of references to documents and position for each word. When performing a search query, the search engine examines all words in every document as it
tries to match them with the search words supplied by the user. The full-text search tries to match the query words exactly to the documents, so it is not very robust when searching and matching near-duplicate documents, because variations in words, like form, time and synonyms can disrupt the retrieval.

3.2.3 Shingles

Shingles, contiguous sequences of characters, sometimes called N-grams, can be used instead of words to search a document collection. Shingles are successfully applied in related near-duplicate detection work [4, 21]. The index size can be a drawback, that has been addressed in previous work. By keeping only a subset, called a sketch, of all possible shingles, the index size is reduced. Frequently occurring shingles are removed [12] or every 25th shingle is kept [5]. To reduce the processing complexity of shingles on even larger collections, the “Super Shingles” were later proposed by Broder [4]. A Super Shingle is actually a sketch made of a sketch. Hoad and Zobel investigated a variety of approaches for filtering good shingles [13]. Reducing the amount of shingles increases the processing speed, but reduces the precision and effectiveness. Using shingles with a sliding window, co-occurring words can be found and merged to a plagiarized sequence [18]. Shingles can be used to find exact matches, but can also deal with the reordering facets of plagiarism without using too much language analysis logics.

Finding shingles in a large document corpus can be extremely efficient by using an inverted file index [27]. Instead of going through all documents before returning the list of matches, an inverted index does the exact opposite by going over the list of distinct shingles and retrieve per shingle the list of documents in which the shingle occurs.

3.3 Contribution

The challenge and contribution of this paper is to detect plagiarism in rather small (news) documents and prevent the approach from marking documents on the same topic as near-duplicates. The method proposed can be used for large document collections and processes many documents at high speed. It uses the shingle index for near-duplicate detection and a new pre-filtering step to minimize the running time and performs almost as good as humans. The main goal is not to apply (shallow) text parsing or natural language modelling, but to focus on similarity, quality, scalability and speed.

Two article collections are used: the original source articles containing plain text news articles and the web articles downloaded by an RSS feed crawler containing potentially copyright inflicting HTML content. Because every query on the index is relatively slow, the number of queries should be minimized by using the smallest document collection or the collection with the least content as query. Therefore one index is build using the web articles and queried with the source articles. Because the index is build on the web article collection, the detection is run as a batch process using all source articles and downloaded web articles since the last run. To decrease the number index comparisons even more, a pre-filter step is introduced where every source article is queried again on the web article collection to filter out irrelevant source articles.

In the next subsections, the approach will be explained in more detail starting by introducing the training and test set, followed by the plagiarism detection step and the pre-filtering step.

3.3.1 Training and test set

Some data collections for plagiarism detection are available online, like the new collection from the 1st International Competition on Plagiarism Detection 2009\textsuperscript{5}, but there are none containing only news articles.

To collect representative data from online news sources, an RSS Feed crawler (downloader) collected 19,900 Dutch news articles from 15 different sources from the 9\textsuperscript{th} till the 15\textsuperscript{th} of March.

\textsuperscript{5}SEPLN09 Workshop PAN. Uncovering Plagiarism, Authorship and Social Software Misuse, http://www.webis.de/pan-09/
One source (and its 819 articles) is selected as the 'original source' and stripped down to plain text articles. Next, 1,000 articles are randomly selected from all remaining sources and called the web articles. These articles are split up into 500 training and 500 testing documents. The dataset with copyrighted material is filled with 819 copyrighted plain text news articles and the corpus of interest consists of 500 news articles from other online news sources including their HTML mark-up. The final step is manually tagging the training and test set collections, to be able to compare the quality to a human treated ground truth.

### 3.3.2 Tokenizer

A tokenizer is a tool to clean up the document content and split it into smaller parts to query the index. Text on web pages is surrounded by website content, i.e. menu’s, banners and other links. This surrounding content, or noise, is not relevant to the similarity between documents, especially when the text appears on various websites with different style and menu structures. To ignore some of the noise and focus on the relevant text blocks, HTML mark-up is removed. The content is cleaned even more by removing all unsupported characters and normalizing characters like é, í and ö to e, i and o. The final step is to break the clean content into smaller character sequences. These sequences are used to query the index and retrieve documents that contain the same sequence.

### 3.3.3 Shingles

All tokenized shingles from a source document are used to find near-duplicate passages in the web document collection by using substrings with a fixed length. Working with shingles with a fixed length has the advantage that the length of multiple shingles can easily be calculated and that they are less sensitive to stemmed text. Now a query document is split up into \( \frac{2 \times \text{content length}}{\text{shingle length}} \) equally distributed shingles to have some overlap between them. Next, every query shingle is fired at the index and the document results are returned. Keeping the complete shingle index in memory would be possible for the rather small training and test set, but to keep the results more realistic and scalable, the index is stored on disk.

To find near-duplicate passages, the order and offset of the retrieved shingles is saved. Multiple shingles from the same retrieved article that are close to each other are merged into a shingle block. These blocks can contain so called gaps. A gap can occur when two shingles are not exact copies, but the shingles around them are. Gaps are caused by stemming, altering, removing and adding. This is visualized in Figure 1. When the block length is as least as long as the threshold set, it is found plagiarized. The pseudo code for finding plagiarized shingle blocks is shown below. The document containing a plagiarized passage (with or without gaps) is returned as a true positive result.

```sql
SELECT shingleA, shingleB
FROM sourceArticles, webArticles
WHERE shingleA = shingleB
GROUP BY webArticles.documentId
HAVING blockLength >= plagiarismThreshold
```

### 3.3.4 Pre-filtering

Normally, when comparing two documents and the similarity drops below a certain threshold, the analysis can terminate early. But when finding near-duplicate passages, early termination is not possible, because even the last passage can be a possible hit. Therefore, comparing every query shingle with every index shingle is a quite expensive operation. To minimize the amount of comparisons and keep the approach scalable and fast, irrelevant articles could be filtered out, called pre-filtering.

Various existing pre-filtering solutions use classification [25, 2], edit distance, prefix filtering and positional filtering [23]. Here, a straightforward implementation of a tree structure that uses
The minimum computational time to filter out irrelevant document candidates based on fingerprint distances is introduced. This pre-filter is called the Signature Tree and was originally introduced as a text indexing methodology [10, 11].

The Signature Tree is constructed and used as follows. First, a balanced empty tree is build with a given depth, branch factor, signature bit vector length and compression factor. Second, all web articles are transformed into signature bit vectors and added to the tree. The signature bit vector is constructed by using the lexicon information from the web article collection, containing all distinct words, where every word has its own unique integer value. The fingerprint of a single document is then constructed by using a bit vector where every word in the document is mapped to an index position of the vector and will set the bit to true. A true bit in the fingerprint means the document contains that word one or multiple times. The bit vector index value is found by retrieving the unique id of the word in the document modulo the vector length. The construction is illustrated in Table 4. Short words and too frequently used words are left out when constructing the fingerprints, since almost all documents will have those words in common.

<table>
<thead>
<tr>
<th>Lexicon</th>
<th>Word</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>“this”</td>
</tr>
<tr>
<td>1</td>
<td>”is”</td>
</tr>
<tr>
<td>2</td>
<td>”a”</td>
</tr>
<tr>
<td>3</td>
<td>”short”</td>
</tr>
<tr>
<td>4</td>
<td>”article”</td>
</tr>
<tr>
<td>5</td>
<td>”another”</td>
</tr>
<tr>
<td>6</td>
<td>”not”</td>
</tr>
<tr>
<td>7</td>
<td>”empty”</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Document</th>
<th>Content</th>
<th>Words</th>
<th>Fingerprint</th>
</tr>
</thead>
<tbody>
<tr>
<td>Document 1:</td>
<td>“This is a short article”</td>
<td>0, 1, 2, 3, 4</td>
<td>000111111</td>
</tr>
<tr>
<td>Document 2:</td>
<td>”Another short article”</td>
<td>5, 3, 4</td>
<td>001110000</td>
</tr>
<tr>
<td>Document 3:</td>
<td>”This is not empty”</td>
<td>0, 1, 6, 7</td>
<td>11000011</td>
</tr>
</tbody>
</table>

Table 4: Fingerprinting example by using the lexicon

All web articles are added to the leaf nodes without the use of any clustering, classification or distance calculations, but a round robin approach. This speeds up the tree building process. Every node of the tree maintains the signature of all document signatures that are added to this node or any of its leaf nodes. The node signatures are merged by using the OR operand. See Figure 2.

When the tree is constructed and filled with web articles, the source articles are also transformed into signature bit vectors by using the same lexicon information. Irrelevant web articles can easily be filtered out by comparing the overlap between the source document signature and
the node signatures starting from the root node down to all leaf nodes. When there is not enough overlap, the web articles below that node can be left out for further processing. This process is visualized in Figure 3. Only the possible relevant web articles returned by the Signature Tree have to be looked up in the shingle index.

To reduce storage space, an attempt was made to compress the signatures by introducing the compression factor. The compression factor is how much the signature is compressed every level down the tree. For example, if the compression factor is 2 and the signature length is 128, the signatures at the first level will have a length equal to 128, but at the second level, the length is equal to 64 where the 65th bit is mapped to the 1st bit, the 66th bit is mapped to the 2nd bit, etc. When two bits are merged and they are different, collision occurs. With fewer documents per node at a lower level in the tree, collisions are assumed to occur less. With low collisions and less bits per document signature, comparing two signatures is done faster and it uses less memory.

A good Signature Tree retrieves possible relevant documents with a very high recall score and the highest possible precision score, because all relevant documents should be found and as many irrelevant documents should be filtered out. In order to construct a good Signature Tree, the following parameters require training:

1. frequently used words: the percentage of the most frequent terms in the document collection that were not used for the constructed signatures. This parameter removes too frequent words, like web page menu items (home, contact, help, etc) and stop words.

2. short words: the minimum word length to use for the signatures. This parameter removes very short words, like numbers and dates.
3. **depth**: the depth of the signature tree.

4. **branch factor**: the number of branches at every node in the tree.

5. **signature length**: the length (number of bits) of the signature. When the lexicon size is larger than the signature length, collision will occur.

6. **compression factor**: the compression factor of the signatures when stepping down the tree. When the compression is 2 and the signature length of a node is 10, all its children will have a signature length of 5. When the compression increases, the number of collisions will increase.

7. **overlap**: the minimum overlap between two signatures (number of true bits) to find the node and its documents still relevant. If the overlap is less, documents are no longer relevant and not returned by the tree (see Figure 6 where the minimum overlap is 3). The overlap value is independent of the level of the node in the tree. Future work can show whether more or less overlap is required between signatures down the tree.

---

**3.4 Results**

First, the document-level Signature Tree and token-level shingle index are individually tested and trained to achieve the highest quality possible. Next, they are combined to achieve high quality and speed. The speed is measured as the time required to process a single document. The quality is measured by precision and recall. Precision is the fraction of retrieved documents relevant.
Recall is the fraction of relevant documents retrieved. These two values can be calculated by using the true \( tp \) and false positives \( fp \) and the false negatives \( fn \):

\[
\text{Precision} = \frac{tp}{tp + fp} \quad \text{Recall} = \frac{tp}{tp + fn}
\]

The first results are from the shingle index runs. To see what parameter values give the best results, a parameter sweep is done on all three parameters: shingle length, minimum passage text length to call it plagiarism and the allowed gap between shingles. The shingle length parameter is chosen between 5 and 100 characters, the minimum passage text length between 10 and 100 characters and the gap between 0 and 10 shingles. The five best results are presented in Table 5. The results show that a gap of 6 shingles works the best in all cases. The shingle length influences the speed the most as run A4 and A5 show. The minimum passage length to call something plagiarism, lies around 50 characters, which is close to the threshold of 60 found by [16] and has the most influence on the precision and recall scores. A1 is chosen as the best run, because the precision and recall scores are almost equal and both very high. Table 5 also shows that the shingle index requires a lot of time per document to process. The Signature Tree helps to speed this up.

<table>
<thead>
<tr>
<th>Run</th>
<th>Shingle length</th>
<th>Passage length</th>
<th>Shingle gap</th>
<th>Precision</th>
<th>Recall</th>
<th>Speed (ms/doc)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Run A1</td>
<td>25</td>
<td>50</td>
<td>6</td>
<td>0.95</td>
<td>0.96</td>
<td>404</td>
</tr>
<tr>
<td>Run A2</td>
<td>25</td>
<td>45</td>
<td>6</td>
<td>0.91</td>
<td>0.98</td>
<td>401</td>
</tr>
<tr>
<td>Run A3</td>
<td>25</td>
<td>55</td>
<td>6</td>
<td>0.96</td>
<td>0.90</td>
<td>424</td>
</tr>
<tr>
<td>Run A4</td>
<td>20</td>
<td>50</td>
<td>6</td>
<td>0.93</td>
<td>0.96</td>
<td>503</td>
</tr>
<tr>
<td>Run A5</td>
<td>30</td>
<td>50</td>
<td>6</td>
<td>0.93</td>
<td>0.95</td>
<td>311</td>
</tr>
</tbody>
</table>

Table 5: Shingle index parameters versus quality and speed

During the Signature Tree step, high recall is more important than high precision, because the second step will increase the precision by examining the viable article candidates more thoroughly, but cannot increase the recall. Therefore our goal is to pre-filter with high recall and the highest possible precision score. Individual plots of runs where only one parameter value is changed, show the relation between that parameter and the precision, recall and speed. The three most influential parameters are the frequently used words, short words and overlap. Their plots are shown in Figure 4 (a), (b) and (c).

When the ignore frequency increases, more frequently used words are left out, so only the more specific words remain. This increases the precision, because less mistakes are made with articles containing mainly frequently used words. Ignoring some of the most frequent terms will give the algorithm a big speed-up as shown in Figure 4 (a), because many stop words can be left out for comparison.

Figure 4 (b) shows that the use of long words increases the precision and reduces the recall. Copying longer, more unique words makes plagiarism more obvious. On the other hand, many articles are no longer marked as plagiarized, because they do not contain so many long words to have enough overlap. The speed is not influenced very much by the minimum token length.

Finally, Figure 4 (c) shows that when the overlap threshold increases, only the articles which share many of the same words are retrieved (exact copies), meaning the precision increases and the recall decreases. The drop in precision score after an overlap of 55 words, can be explained by some incorrectly tagged true positives. Some plagiarized articles are incorrectly tagged as a false positive, but retrieved by the algorithm as a true positive result. When the overlap is 80 words, half of the retrieved true positive results is incorrectly tagged.

The recall and precision scores from the most interesting signature tree runs are shown in Figure 4 (d). Every dot in the graph presents the precision and recall scores of a single run where every run was done with different parameter values. Four of the best results and their parameter values are displayed in Table 6. The speed per document does not include the approximately 17 seconds it takes to build the Signature Tree.
Table 6: Signature Tree parameters versus quality and speed

Table 6 shows that the depth of the tree, the number of branches, the signature length, and the compression factor use the same parameter value in all 4 runs, so these values work best. The depth is tested between 1 and 10, the number of branches between 10 and 50, the signature length between 16384 and 262144 and the compression between 1 and 8. The frequently used words, short words and overlap have more influence on the precision and recall scores, but do not influence the speed very much. Run B2 shows a very nice overall result where the precision and recall are both 0.85. The other runs show a much higher difference between the recall and precision score. Run B1 and B4 show the best results, because their recall scores are very high and have a much higher precision score compared to B3. Meaning more irrelevant documents are filtered out and less time is spent querying the shingle index.

Finally, the shingle index and Signature Tree are combined. The parameters from Run A1 + B1 and A1 + B4 are used for the final runs. Table 7 shows the quality and speed results, where the speed is the average over 10 runs.

Table 7: Final results: the average speed over 10 runs.

3.5 Conclusion

The final results show that the Signature Tree pre-filtering is a very fast way to detect near-duplicate articles. It does its analysis almost 20 times faster than the shingle index approach and still produces acceptable quality. As a pre-filtering step, it also does a nice job, because it finds almost all viable documents (93%) and filters out a lot of irrelevant articles by achieving a precision of 66%. Even on its own, the Signature Tree can detect plagiarism very fast with acceptable quality: 85% precision and recall (Run B2).

When these results are compared with the more advanced and in depth shingle text comparison approach, the shingles are the best approach in terms of quality. It can detect plagiarism almost as good as a human can. Also, it is a lot faster than a human, but when dealing with large document
collections, 400 milliseconds per document can be too slow. Therefore, combining the two methods
gives us the advantage from both: very high quality and still 4 times faster.

The combined run shows a slightly lower recall, because very short news articles that were
tagged as plagiarism by the human are missed, because the content and number of shared words
are less than the best working threshold of 18 words and 50 characters. The shingle run does not
score 100 percent precision, because the human made some mistakes when building the training
and test set. He had to compare the 819 articles with the other 1000 articles and did not find all
plagiarised documents by hand. This is the human error. The results that show up as incorrect
(false positives) results, are all plagiarized, so the algorithm was able to find the human error.
These results were not corrected in the training and test set, because than it should also be
corrected for the overlooked true negatives.

The methods are not thoroughly tested on large collections (>20,000 articles), but the results
look very promising. When working with even larger collections, the current approach would be
too slow. Making this work seems possible when taking the news article date into account. A
news article from today never needs a comparison with news articles from last month or maybe
even last week. If this is applied on the shingle indexes, online news article passage plagiarism
detection might be possible on news articles from all over the web.
Figure 4: Signature Tree results
4 Implementation

The implementation starts by defining the (functional) requirements for the plagiarism detection application:

1. The application must detect near-duplicate plagiarized passages from online news articles in a fast and scalable way to deal with large data sets. Therefore it should support multi-core cpu’s to make full use of the parallel processing time.

2. The detection quality must be high, preferably the same as a human would achieve when searching for plagiarized passages, meaning among the retrieved plagiarized articles are not many incorrect results and the number of undetected plagiarized articles is kept low.

3. The application must support multiple plagiarism detection implementations. Switching between the implemented solutions must be flexible, by changing a configuration parameter.

4. The source articles, web articles and the plagiarism detection results are all stored in a database system, so new articles can easily be added and the results evaluated.

5. The application runs on Windows, is written in C# and uses a Microsoft SQL database.

The next sections give a more detailed description of the application realization. First, the system architecture is chosen and the class diagram drawn. Second, the plagiarism detection algorithms are selected and implemented. Third, the speedup is created by implementing the Signature Tree. Finally, the characteristics of the dataset and the hardware setup are discussed.

4.1 Application design

The application is divided into three subsystems: data, search and plagiarism system. Their functionality and interaction is described in more detail.

The data system (Appendix A Figure 7) contains all data objects used and connects to the database to retrieve and save documents. A singleton manager class (DataManager) is used by any other subsystem to interact with the data system. Hereby the data stream is controlled by this manager.

The search system (Appendix A Figure 8) implements all the required search functionality. The system retrieves all documents from the data system that match a query document by using a given search algorithm. The search system is very flexible and can contain many search algorithm implementations. Switching between these implementations is done by simply changing a configuration parameter. The search system also has a manager class (SearchManager) to make itself easily accessible for any other subsystem.

The plagiarism system (Appendix A Figure 9) is the main application. When started, it loads all system parameters and starts a number of plagiarism detection threads. Every thread queries the search system by accessing the SearchManager to retrieve a list of plagiarized documents matching the query documents. The results are stored in the database for evaluation. Doubling the number of threads almost halves the retrieval time, but using more threads than CPU cores is useless. The recommended number of threads is equal to the number of CPU cores.

The class diagram of the whole system and its subsystem relations is displayed in Appendix A Figure 10.

4.2 Plagiarism detection algorithms

The application design is meant to support multiple plagiarism detection algorithms, because when implementing these algorithms, small implementation adjustments are made to find the best solution. To keep the code clean and compare the various algorithms, every algorithm is saved to a different class file. By adjusting an application configuration parameter, a specific class implementation is specified by using reflection.
Without reflection
 Foo foo = new Foo();
 foo.Hello();

With reflection
 Type t = this.GetType("Namespace.Foo");
 object foo = Activator.CreateInstance(t);
 t.InvokeMember("Hello", BindingFlags.InvokeMethod, null, foo, null);

The main goal here is to implement a Shingle index and use this to detect plagiarism. The Shingle index can be implemented by using either the TEEZIR framework, some external (open source) tool or build something from scratch. To make a good decision, three major factors are examined for every approach: time, functionality and flexibility. The shingle index still needs to compare \( N \times M \) documents, so this requires a pre-filter to filter out irrelevant documents and speed up the detection process. This is done by implementing the Signature Tree. Three plagiarism detection implementations are built: the shingle index, the signature tree pre-filter and those two combined.

### 4.2.1 Shingle indexes

Working at a company like TEEZIR, with a lot of experience in search techniques, has the advantage of being able to use their framework containing many search solutions. Their framework can easily set up a simple shingle index application supporting large document collections. Scalability can sometimes be an issue when the collections become too large. Also, every bug fix and update they apply to their framework is automatically applied to the shingle index code. Because the framework allows you to use default classes or build your own custom code, it gives you a lot of flexibility when building the shingle index. A major drawback of using the extensive framework can be the time required to get acquainted with the framework and build your customized index. So using the framework has the advantage being flexible and having most of the required functionalities, but it might take some time before everything is correctly set up.

A very suited and powerful open source search tool is Sphinx\(^6\). It can connect to many types of databases, like MySQL and Microsoft SQL, to collect and index the information retrieved from the database. Sphinx is suitable for very large document collections and supports distributed search. Sphinx supports prefix and infix searching where infix is suitable to retrieve all documents containing some query shingle. The infix search does not require to rebuild the complete index when the shingle length is changed, so this can save some time when tuning the plagiarism detection algorithms. I worked with Sphinx before, so it takes little time to set up a proper index. Sphinx is an open source tool and actively being developed, so updates and bug fixes are continuously available.

The last option is to build a shingle index from scratch. There are many things to take care of when building a search tool and index, like processing and compressing the content and how it is stored on disk, caching the results, keeping the index scalable en preferably distributed and building a proper interface to query the index. It will take a lot of time to build something like this. The functionality and flexibility can be very high, because it is all custom made and should exactly fit the requirements.

<table>
<thead>
<tr>
<th>Approach</th>
<th>Functionality</th>
<th>Flexibility</th>
<th>Time</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>TEEZIR framework</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+++</td>
</tr>
<tr>
<td>Sphinx search</td>
<td>+</td>
<td>+</td>
<td>+++</td>
<td>+ + ++</td>
</tr>
<tr>
<td>Custom index</td>
<td>++</td>
<td>++</td>
<td>--</td>
<td>++</td>
</tr>
</tbody>
</table>

Table 8: Possible Shingle index implementation approaches compared

\(^6\)Sphinx search, http://www.sphinxsearch.com
The three approaches are compared and the result is shown in Table 8. The Sphinx search index (v0.9.9-rc) is used with the configuration file parameters from Appendix B. To query the index, Sphinx can be accessed like a MySQL server by using the MySQLDriverCS dll for C#.

4.2.2 Signature tree

The Signature tree implementation was intended to be used as a pre-filter for the shingle index search, but can also used as a standalone plagiarism detection algorithm. The implementation is completely done in C# and added to the search subsystem of the application.

The Signature Tree is constructed and used as follows. First, a balanced empty tree is build with a given depth, branch factor, signature bit vector length and compression factor. Second, all web articles are transformed into signature bit vectors and added to the tree. The signature bit vector is constructed by using the lexicon information from the web article collection, containing all distinct words, where every word has its own unique integer value. The fingerprint of a single document is then constructed by using a bit vector where every word in the document is mapped to an index position of the vector and will set the bit to true. A true bit in the fingerprint means the document contains that word one or multiple times. The bit vector index value is found by retrieving the unique id of the word in the document modulo the vector length. The construction is illustrated in Table 1. Short words and too frequently used words are left out when constructing the fingerprints, since almost all documents will have those words in common.

To filter out too frequent terms, the lexicon knows how many times a single term it is used in all documents in the collection. This list is sorted from most to least frequent terms and the top K percent frequent terms are selected as 'too frequent'. The implementation finding the too frequent terms is as follows:

```csharp
// Retrieve the terms ordered by frequency
List<Term> frequentTerms = lexicon.FrequentTerms;

// Ignore the top K% frequent terms when finding viable documents
int minIndex = (int)(frequentTerms.Count * IGNORE_FREQUENCY_PERCENTAGE);
Term tooFrequentTerm = frequentTerms[minIndex];
```

To reduce storage space, an attempt was made to compress the signatures by introducing the compression factor. The compression factor is how much the signature is compressed every level down the tree. For example, if the compression factor is 2 and the signature length is 128, the signatures at the first level will have a length equal to 128, but at the second level, the length is equal to 64 where the 65th bit is mapped to the 1st bit, the 66th bit is mapped to the 2nd bit, etc. When two bits are merged and they are different, collision occurs. With fewer documents per node at a lower level in the tree, collisions are assumed to occur less. With low collisions and less bits per document signature, comparing two signatures is done faster and it uses less memory. A small block of code follows to show how signatures are created when the list of terms of a document are known and the size of the signature are given:

```csharp
/// Make a signature print from the terms.
private Signature MakeSignature(List<Term> terms, int size) {
    Signature signature = new Signature(size);
    foreach (Term term in terms) {
        if (term != null) {
            signature.Set(term.ID % signature.Size, true);
        }
    }
    return signature;
```
All web articles are added to the leaf nodes without using any clustering, classification or distance calculations. A simple round robin approach is used. This speeds up the tree building process. Every node of the tree maintains the signature of all document signatures that are added to this node or any of its leaf nodes. The node signatures are merged by using the OR operand. See Figure 5.

When the tree is constructed and filled with web articles, the source articles are also transformed into signature bit vectors by using the same lexicon information. Irrelevant web articles can easily be filtered out by comparing the overlap between the source document signature and the node signatures starting from the root node down to all leaf nodes. When there is not enough overlap, the web articles below that node can be left out for further processing. This process is visualized in Figure 6. Only the possible relevant web articles returned by the Signature Tree have to be looked up in the shingle index. The code to calculate the overlap is displayed below.

```java
// Construct the overlap signature
Signature query = MakeSignature(terms, child.Signature.Size);
Signature and = query.Clone();
and.And(child.Signature);

// Enough overlap?
int overlap = and.NumberOfTrue;
if (overlap >= MIN_OVERLAP) {
    // Found viable documents
}
```

### 4.3 Testing

Testing is needed to detect incorrect code to achieve high quality software. Testing is not only done afterwards, when the application is finished, but already starts during the implementation phase. Unit tests are very suitable to test during the implementation phase individual blocks of code, like classes or subsystems. Therefore, before every class is implemented, a unit test set is build to detect possible failures of every block of code. Failures like null pointer exceptions, connection timeouts, calling a function with the incorrect preconditions, returning the function with incorrect postconditions, and many more can be detected early during the creation of the software. When the classes are adjusted in a later state of the implementation, all unit tests can be rerun to see if the pre- and postconditions still hold and no tests fail. Every block of code is tested with a unit test. Especially the search system is thoroughly tested, because it contains the most vital and difficult blocks of code. A small example of a unit test is:

```java
Signature signature = new Signature(64);
Assert.AreEqual(signature.NumberOfTrue, 0);

// Signature: 11100000001
signature.Set(0, true);
signature.Set(1, true);
signature.Set(2, true);
signature.Set(9, true);

Assert.AreEqual(signature.NumberOfTrue, 4);
```

### 4.4 Data set

Because there is no dataset available on online news articles, one had to be build. Collecting online news articles is done by subscribing to 150 RSS feeds from 15 different news websites containing
the most recently published news articles. When new articles are found, they are downloaded and added to the dataset. This requires an RSS feed crawler application to monitor all 150 feeds. The requirements for the RSS feed crawler are:

1. The application must be able to monitor RSS, Atom and RDF feeds and easily extendable with more types of feeds.

2. Depending on how frequently a feed is updated, the application should increase or decrease the interval time between monitoring that feed not to lose any information.

3. The solution must be very robust, because it is run as as service and should never stop, otherwise it will lose vital information which cannot be retrieved, because most feeds only contain up to approximately 20 articles and within a few hours, articles are too old and can no longer be retrieved through a feed.

4. Adding, editing and removing feeds must be easy and the application will detect the changes automatically. No reboot is required.

5. Only valid news articles, containing a title, date and a link to full text web page are supported.

6. To ignore RSS feed commercials or skip certain web pages, a blacklist of URL’s is used.

7. The application runs on Windows, is written in C# and uses a Microsoft SQL database.
Figure 6: Querying the Signature Tree to find relevant near-duplicate candidates. The red signatures are the overlap between the query and the node signature. When the number of true bits in the overlap signature is equal or higher than 3 in this example, the algorithm continues walking down the tree. This example uses a compression factor of 1.

The RSS feed crawler was not available from the TEEZIR framework, but custom-built for this project and TEEZIR. The class diagram in Appendix C Figure 11 shows its three main sub functionalities:

- Monitor the list of feeds and add, remove or update feeds whenever the information changes.
- Monitor a single feed and retrieve new RSS feed items to add them to the download queue.
- Monitor the download queue and download per item its HTML page and store this information in the database.

To collect representative data from online news sources, the RSS feed crawler (downloader) collected 19,900 Dutch news articles from 15 different sources from the 9th till the 15th of March 2009. One source (and its 819 articles) is selected as the ‘original source’ and stripped down to plain text articles. Next, 1,000 articles are randomly selected from all remaining sources and called the web articles. These articles are split up into 500 training and 500 testing documents. The dataset with copyrighted material is filled with 819 copyrighted plain text news articles and the corpus of interest consists of 500 news articles from other online news sources including their HTML mark-up. The final step is manually tagging the training and test set collections, to be able to compare the quality to a human treated ground truth.

4.5 Hardware setup
The three applications, RSS feed crawler, Plagiarism detection tool and Sphinx search were run on different hardware and operating systems. The RSS feed crawler had to run 24/7 for over at
least one week, so the TEEZIR server was used. This machine runs Windows 2003 Server and has a dual quad core CPU and 16GB memory. The plagiarism detection application runs on a dual core machine at 2.26 GHz with 2GB memory running Windows Vista. Sphinx is not very stable running under Windows, so it runs in a virtual environment (Ubuntu server 8.04 on Sun VirtualBox using 384MB memory) on the same dual core machine. To improve the retrieval speed, the virtual machine should be moved to another physical computer.
5 Additional results and conclusions

Early in the process, a number of documents were written describing the thesis definition, its planning, the demands and wishes and a proposal of a feasible solution. This section will discuss these documents and discuss the differences between the targets and the achieved goals.

5.1 Proposed solution

The 'Definition' and the 'Proposal of a feasible solution' documents (Appendix G and D) both state that the plagiarism detection application should be able to detect plagiarized passages instead of complete articles. This criterion is successfully fulfilled, although different then first described. Initially two algorithms were compared: N-grams (shingles) and suffix arrays. Appendix F describes both methods in more detail. Although suffix arrays are very powerful to search for exact matches, when searching for near-duplicate passages, suffix arrays must be split up into smaller sequences to query the index, which is not very different from the shingle approach. Instead of implementing the Suffix array, the shingle index is enriched with the Signature Tree to speed-up the retrieval process.

5.2 Demands and wishes

During the setup, the demands and wishes are described to define what criteria will make this thesis project a success. The full document is included in Appendix E.

5.2.1 Demands

Almost all essential criteria are met. The speed criterion is met, because the number of online Dutch articles published per day is lower than the time required to examine all articles. If TEEZIR wants to run the application on all Dutch web pages instead of all Dutch news pages, they need more throughput. In that case the speed per article might still be too high and some improvement steps must follow.

The flexibility and quality criteria are also met, because users can select plagiarized sets of articles based on the similarity score. This score is stored in the database and expresses the highest passage similarity between two articles. The quality of the retrieved results is high: 0.96 precision and 0.94 recall, close to human perception.

The scalability of the plagiarism detection application is not extensively tested, but with better hardware and the possibility to make full use of multi-core processors, it will be possible to scale up to larger data sets.

5.2.2 Wishes

The thesis project and implementation did not focus on any of the wishes described in Appendix E during the implementation phase. However, thanks to a good design and testing, the application did not crash during any of the runs. Therefore the robustness of the plagiarism detection application is high. The security wishes are not met in any way.

5.3 Planning

The planning described in Appendix D was followed, except for the implementation phase, which took 5 instead of 4 weeks due to extra research for the Signature Tree implementation.
6 Future work

This section describes the topics for further research: plagiarism detection improvements and additions.

The plagiarism detection application is not extensively tested on large document collections, so very little is known about its scalability. To find the data collection size threshold where the application can no longer process articles fast enough because it is running out of resources, can be found by using various data collection sizes. This threshold value gives a good indication about the scalability of the current application and its maximum capabilities on a standard desktop computer. Improving the scalability can be achieved by using a distributed search, where a single query is not run on one computer, but distributed to multiple servers. These servers simultaneously retrieve the results to a subset of the query. All subsets are merged and returned as one result to the user.

The current solution uses the source articles to query the web article index. Since the source article collection is a very static collection and its index does not require a frequent rebuild, it can be more efficient to swap the collections and use the web articles to query the source article index. An even more optimized solution is to build indexes on both article collections and match one index with the other.

According to Hoad and Zobel [13], ranking outperforms fingerprinting, so instead of the Signature Tree, an inverted file index can be used to retrieve the number of distinct words two articles share. When they have enough distinct words in common, the index article is returned as a viable article. The Signature Tree is very useful when trying to find the best parameter values to pre-filter your document collection. When these parameters are found, tests on different collection sizes can tell whether a Signature Tree or an inverted file index performs better with this set of parameters.

As an addition to the application, results can be made more accessible to its end users by building a Graphical User Interface (GUI) to retrieve all plagiarism results or only the results with a similarity value above a user-defined threshold. A simple web interface can sustain.
References


A Class diagrams

Figure 7: The data system, containing all used data objects.
Figure 8: The search system, used to query by example.
Figure 9: The plagiarism system, a multi-threaded plagiarism detection application.
Figure 10: The complete class diagram showing the relations between the subsystems.
B Sphinx search configuration

# minimum indexed word length
# default is 1 (index everything)
min_word_len = 1

# charset definition and case folding rules "table"
# optional, default value depends on charset_type
#
# defaults are configured to include English and Russian characters only
# you need to change the table to include additional ones
# this behavior MAY change in future versions
#
# charset_table = 0..9, A..Z->a..z, a..z, U+021, U+023..U+02C, U+02F,
# U+03C..U+040, U+05B..U+05E, U+060, U+07B..U+07E,
# U+080..U+09F, U+0A1..U+0AC, U+0AE..U+0FF

# minimum word prefix length to index
# optional, default is 0 (do not index prefixes)
#
min_prefix_len = 0

# minimum word infix length to index
# optional, default is 0 (do not index infixes)
#
min_infix_len = 1

# enable star-syntax (wildcards) when searching prefix/infix indexes
# known values are 0 and 1
# optional, default is 0 (do not use wildcard syntax)
#
enable_star = 1

# n-gram length to index, for CJK indexing
# only supports 0 and 1 for now, other lengths to be implemented
# optional, default is 0 (disable n-grams)
#
ngram_len = 1

# whether to strip HTML tags from incoming documents
# known values are 0 (do not strip) and 1 (do strip)
# optional, default is 0
html_strip = 1

# what HTML elements contents to strip
# optional, default is empty (do not strip element contents)
#
html_remove_elements = style, script
C RSSFeedCrawler class diagram

Figure 11: The complete class diagram of the RSSFeedCrawler build to collect online news articles.
D Definition and planning

When you search on the internet, you will find many near-duplicate pages that contain almost the same information. It is easy to copy-paste information from a webpage to your own. Many companies (like Google) are trying to detect near-duplicate documents on the internet to save storage space and to prevent the same information from showing multiple times. Near-duplicates can be exact copies, like mirror pages, different formatted versions (PDF, HTML, etc.) or shared content between different sources or altered copies, like redacted or quoted content. When near-duplicates result from copyrighted material and the original source is not specified, this is called plagiarism.

In most of the cases found in the literature, documents are compared with documents based on their content and structure. Document-by-document comparison is sufficient to detect plagiarized documents that are copied from someone else and adjusted to make it look like your own. To improve efficiency of this process, plagiarism detection can then be viewed as a similarity self-join over the corpus of interest, or as the similarity join between a database of copyrighted material and the corpus of interest.

A problem with the current approaches is that people also plagiarize by 'borrowing' small passages from other documents without mentioning the original source. To be able to detect this kind of plagiarism, new similarity join techniques should be developed and tested. Performance and quality are the two important issues to optimize for, as we should be able to handle large datasets with documents (e.g., a crawl of the Dutch internet). Instead of comparing N x N documents in the self-join, the technique should be able to handle a comparison of M x M passages. To guarantee high quality results, many plagiarized passages should be found while the amount of false positives should be kept low.

Eventually, this plagiarism detection application will be implemented in the framework of TEEZIR, an innovative information retrieval company in Utrecht. The tool will get a stream of online news articles as input and will detect plagiarism on a local online database, consisting of thousands of news articles.

D.1 Planning

<table>
<thead>
<tr>
<th>Week no.</th>
<th>Weeks</th>
<th>Action</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>5</td>
<td>Planning</td>
<td>Literature research, define the thesis assignment and make a schedule.</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>Collecting content</td>
<td>Collecting online articles to use as a test set.</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
<td>Initial experiments</td>
<td>Test early prototype implementation of candidate and baseline algorithms (using the NGrams and Suffix array). Get acquainted with these algorithms.</td>
</tr>
<tr>
<td>12</td>
<td>1</td>
<td>Application design</td>
<td>Design of the application. Requirements, functional requirements, class diagram.</td>
</tr>
<tr>
<td>13</td>
<td>4</td>
<td>Implementation</td>
<td>Implementation of the NGrams and Suffix array algorithms.</td>
</tr>
<tr>
<td>17</td>
<td>3</td>
<td>Testing and tuning</td>
<td>Testing and tuning the application. Tune the parameters and measure the performance and quality.</td>
</tr>
<tr>
<td>20</td>
<td>6</td>
<td>Evaluation</td>
<td>Evaluate and optimize the results.</td>
</tr>
<tr>
<td>26</td>
<td>5</td>
<td>Writing</td>
<td>Write the report.</td>
</tr>
</tbody>
</table>
E Program of demands and wishes

E.1 Introduction

Many text documents are published on the internet every day. Articles are published on blogs, personal websites, news websites, etc. Most of these documents are public and can be read by anybody. When documents are public, it is easy to copy the content and use it anywhere else on the internet as if it were your own. When copyrighted documents are (partly) copied and the original author or source is not mentioned, it is called plagiarism.

The need for automatic plagiarism detection for online news websites is growing. Publishers would like to keep track of the people (or websites) that use (parts of) their articles, because a fee must be paid when people are reusing copyrighted articles. To be able to detect plagiarism, similarity between online articles and the original articles should be measured.

TEEZIR, an information retrieval company located in Utrecht, The Netherlands, is looking for a solution to detect plagiarized online news articles. This document will make clear what the demands and wishes are to successfully complete this project.

E.2 Goal

The goal of this thesis project is to detect near-duplicate news articles on various websites. The program should be able to report the URL’s of web pages that contain altered or copied articles against a known personal database with news articles. Finally, the publishers can be informed about the plagiarized content. The amount of articles that can be processed should be higher than the amount of articles that are crawled (downloaded) online per day.

E.3 Reason

Because there are so many websites that publish news articles, people and organizations can easily lose track of the locations where their articles are reused with or without their permission. It would be very interesting to know where and how frequently articles from a single source (author or organization) are published. Also, it would be very interesting to know since when articles were copied (published), so the original publisher knows how much they can charge the source. Some websites will have an agreement with the original source to use a certain amount of articles per day, other websites do not have such an agreement. To keep track of your own online articles and their reuse frequency, a near-duplicate detection application is required.

E.4 Adjustments

Testing the application in a real situation with streaming online news articles and a growing own database is very complicated, because the results are very hard to reproduce and building a test set is almost impossible. To generate more consistent environment to test our demands, a fixed personal database and a fixed online article database are used.

E.5 Demands

E.5.1 Speed

The amount of articles that are processed per time unit should be higher than the amount of articles that are crawled in the same time unit. This will ensure that the program can process all retrieved online news articles. The news articles that are crawled are retrieved from different news RSS feeds. RSS feeds will provide the most recent news articles, so the crawled collection of news articles is always incremental.
E.5.2 Flexibility

The amount of resemblance between two documents is expressed by a similarity value. This value can be anything between 0 (no resemblance at all) and 1 (a passage from document A is an exact copy of a passage from document X). This will make it easy for the end-user to select documents with a certain degree of similarity. If the user is looking for documents with exact matches, it should only select the documents with a similarity value of 1.

E.5.3 Scalability

The article collection will grow over time. We expect the amount of online articles will grow as well. Scaling the online collection and the own collection should not affect the performance (speed). Scaling the collections to infinity will never guarantee the speed, but let me do a small calculation to show to what collection size this application should be able to scale to without jeopardizing the speed demands.

- Number of articles published per week by 125 online news RSS feeds: 20,000
- Scaling to 1,000 online news sources per week: 20,000 / 125 * 1,000 = 160,000 articles
- Number of online articles to process per second: 160,000 / (7 * 24 * 60) = 0.26
- Number of articles published per week by a single news source: 1,000
- Using 1 year of own documents to be able to detect recent plagiarized articles: 52 * 1,000 = 52,000 articles

So the application should be able to scale to an index for approximately 52,000 articles and process 1 online article every 4 seconds to keep up with a real situation.

E.5.4 Quality

Parts of articles that are literally copied should be detected as plagiarism. Parts that are edited should be detected as near-duplicates. Matching the human plagiarism detection quality is impossible, but news articles will not differ that much, so most of the plagiarized articles should be detected. Of course, when copied parts are very short or widely spread through the article, detection may fail. To test the quality of the detection, a test set should be constructed by a human. The set exists of 1000 randomly selected online news (including HTML mark-up) from an RSS feed collection during one week. 500 Articles will be used for training and 500 for testing purposes. One online news source is selected as the "original source" and all articles from this source published in the same week are selected. This set will contain approximately 700+ articles containing only the plain news text. This human will read all 1700 articles in the test set and write down what online articles are near-duplicates of which original article. This information is saved and considered the "correct answers". The goal is to perform as good as the human. The quality will be expressed by calculating the precision and recall scores, which are common and accepted measurements in Information Retrieval.

E.6 Wishes

Wishes are not mandatory for success, but implementation would be very nice. Detecting plagiarism is all about near-duplicate detection. But detecting real plagiarism would also mean the program is able to determine whether there is a link to the original source of the copied document or not. Initially, this must be done manually, but it would be convenient to build this into the program and have it done automatically.
E.6.1 Robustness

The program should be quite robust. The program will be running for many days continuously, so whenever there goes something wrong with the internet connection or the local database, the application needs to recover automatically and continue its work. Failures should be logged and reported to the owner, so they can try to fix or prevent them from happening. The application should use a copy of the original article database in case something goes wrong, so no extra backup is required for the database system. Disk failure should be prevented by using a backup system or mirror raid. When the power fails, the detected results should be save and stored to disk, so saving the results frequently is desirable.

E.6.2 Security

The articles that are used during the detection task are (most of the time) copyrighted. Unauthorized people should not be able to access these documents.
F N-grams and Suffix arrays

F.1 Suffix Arrays

A suffix array is basically a lexicographically sorted list of all the suffixes of a document. The suffix array is used to find all instances of string W in document A with N suffixes. Building a suffix array can be done in linear time \(O(N)\), independent on the lexicon size. Building in linear time is not very realistic, because it makes use of a very big constant. Having a sorted list of all suffixes of the text, has the advantage to search the index binary. Any search on the index can be answered in \(O(\log N)\) time in worst case.

The power of a suffix array is that with one traversal over the query text, all matching locations can be retrieved. The longest common prefix can be stored in a second array. For every suffix, the longest common prefix length is stored for its successor.

When looking at the plagiarism detection task, some suffix array properties can be identified as useful. First of all, being able to find a longest common prefix can be useful to detect large blocks of copied text. The speed to search all documents in a collection will be acceptable fast. Suffix arrays by themselves are not able to solve the plagiarism detection. To detect plagiarism, you should not only be able to detect the matching text parts, but also detect parts that are close to each other in the document.

An assumed drawback can be that the index cannot fit into memory, but must be stored on disk. This will cost some I/O time when (binary) searching the files on disk. Maybe we can predict or store where changes in the first letter of the suffixes are located on disk, to save I/O time. Another drawback is that the suffix array cannot be extended when new documents are added to the collection, so documents should be added in batches. This could lower the quality of the plagiarism detection when suffix arrays are build on the database documents. This can be solved or improved by building a suffix array index on the query documents and save up the query documents till some amount before trying to detect plagiarism. This would make the plagiarism detection into a more static approach instead of dynamically streaming.

To query complete documents on the suffix array index, complete words can be used from the query document to match them on the index. Another option would be to build NGrams from the query document and try to find matches of the NGrams in the document index.

F.2 NGrams

NGrams are blocks of tokens (characters) of a fixed size. These blocks can be generated by taking substrings of a fixed size the original text.

NGrams are dependent on the lexicon size. The amount of possibilities are determined by the lexicon size to the power of N. The occurrences of the NGrams in the documents can be looked up in the index.

When using NGrams for plagiarism detection, the amount of possible combinations are lowered, because sequences like ‘aaaaa’ are hardly seen in text documents. This could give the NGram technique the advantage to save the index in memory. The link to what document and at what offset could be saved in memory when there are not too many documents and when there is enough memory available.

A drawback can be, for large collections, that the index cannot be saved in memory. Just as with the suffix arrays, NGams cannot solve the plagiarism detection task by itself. NGram offsets should be compared and when they are close to each other, a sequence is formed to determine whether it is large enough to be plagiarism.

F.3 Compression

To store more tokens on disk or in memory, a compression technique (Huffman encoding) can be used. This technique will compress the most frequently used tokens the best. This can be useful for plagiarism detection, because low compressed text parts can be detected and treated as the
most unique parts of the document. When these unique parts are also found in other documents, it could be a lead to plagiarism.

F.4 Conclude

To conclude, it would still be very interesting to compare suffix arrays and NGrams. Both seem very suitable to build efficient indexes and to search occurrences of word W in document N. I would expect the suffix array to perform better when the query’s become longer. The NGram on the other hand can be very quick when the index can be kept in memory instead of on disk.

Maybe there will be a way to combine the best things of both approaches by using the suffix array index and using the NGram query.
G Proposal of a feasible solution

There are various possible solutions to solve the problem statement. I would like to use a combination of the above discussed techniques. The index used to search the document collection should be able to handle many articles and should be easily scalable to even larger collections. An index by itself cannot determine whether an article is plagiarized or not, because we are dealing with near-duplicate passages and not only exact copies. So besides the index, this problem will require some extra logics based on thresholds or even the law. My main goal is not to apply (shallow) text parsing or natural language modelling, but to focus on quality, scalability and speed.

The n-gram and suffix array indexes are very suitable for this task. The indexes will be used to search for small passages of query articles. The full-text index is unable to retrieve small passages, like substrings or sentences. That is the reason why this type of index is left out. The index search will return a list of documents in which the passages occur. The offset and length of the results will be taken into account to measure how much two documents resemble. The resemblance is expressed as a value between 0 and 1. With 0 meaning: at least one similarity and 1: this is an exact copy.

I want to show which index type is the best in terms of speed and scalability. To optimize this, the design of the application must be well thought-out. Think about multi-threading, memory and i/o management. The (multi-threaded) application design diagram is included in Appendix A. Besides a good design, the fingerprinting and compression technique can decrease memory usage and disk access time to speed up the application.

Different combinations of stopping and stemming will change the index quality. Tests should display what combination is best for this task. Besides stopping and stemming, the mark-up language will be removed, because it is not needed to detect plagiarism. The resemblance logics will influence the retrieval quality too, but also the retrieval speed. When the quality increases, the expected speed will decrease. A good balance must be found between speed and quality.

To test and measure the speed and scalability, I will use a data collection of 20.000 articles or more. The articles are collected from RSS feeds from different Dutch news sources in the last months. A smaller manually tagged test set is extracted to measure the quality. By personally comparing 819 own articles with a 1000 random selected online news articles, the own articles are linked to all near-duplicate online articles. This will generate a set of articles that a human would call plagiarized. Finally, the set is split into 500 training and 500 testing articles. All 819 own articles will be used when training and testing. Precision and recall will express the quality of every test run.