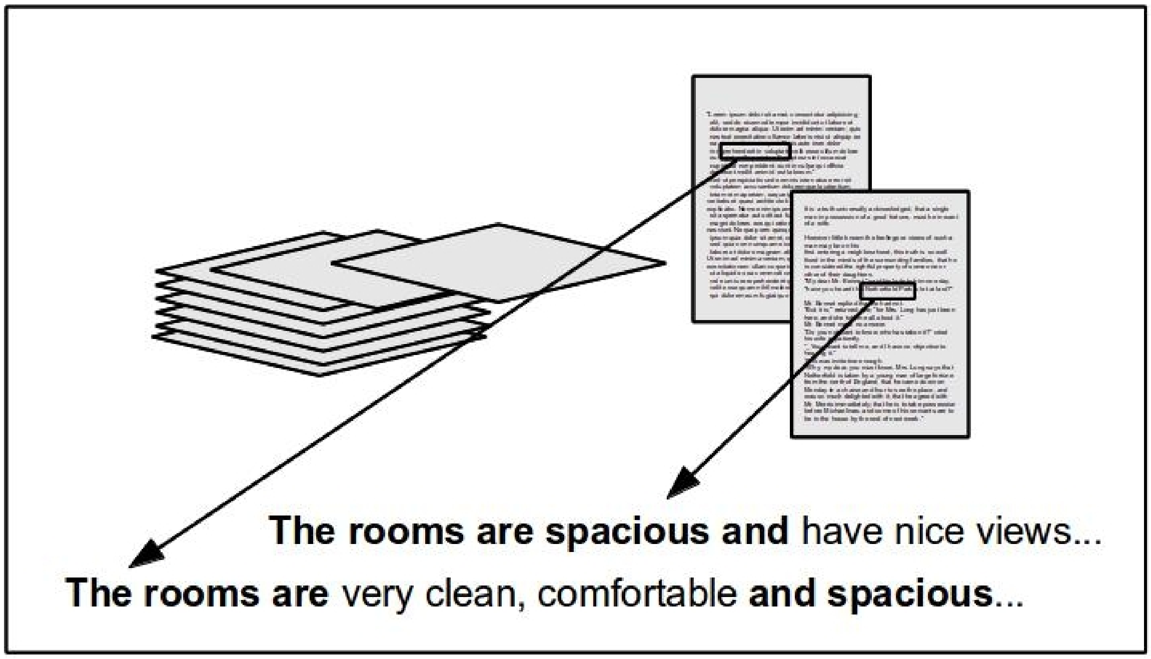

The rooms are spacious and have nice views...
The rooms are very clean, comfortable and spacious...

Finding Top-k Similar Document Pairs
- speeding up a multi-document summarization approach

Master of Science Thesis in the Programme Computer Science - Algorithms, Languages, and Logic

Emma Bogren
Johan Toft

Chalmers University of Technology
Department of Computer Science and Engineering
Göteborg, Sweden, June 2014

The Authors grant to Chalmers University of Technology the non-exclusive right to publish the Work electronically and in a non-commercial purpose make it accessible on the Internet. The Authors warrant that they are the authors to the Work, and warrant that the Work does not contain text, pictures or other material that violates copyright law.
The Authors shall, when transferring the rights of the Work to a third party (for example a publisher or a company), acknowledge the third party about this agreement. If the Authors have signed a copyright agreement with a third party regarding the Work, the Authors warrant hereby that they have obtained any necessary permission from this third party to let Chalmers University of Technology store the Work electronically and make it accessible on the Internet.

Finding Top-k Similar Document Pairs
- speeding up a multi-document summarization approach

EMMA BOGREN,
JOHAN TOFT

© EMMA BOGREN, JOHAN TOFT, June 2014.
Examiner: PETER DAMASCHKE

Chalmers University of Technology
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Cover: Investigating Overlap similarity between pairs of documents

Department of Computer Science and Engineering
Göteborg, Sweden June 2014

Abstract

Today there exist many approaches to multi-document summarization, which is the task of automatically creating a summary from multiple sources. This can be a complicated problem, but in this thesis we have focused on a simple approach that is currently being researched. The idea is to find the pairs of documents with the largest overlaps in words and use these to produce a summary. Originally, the algorithm used for this was very naive, comparing each word of every possible pair of documents, and the aim of this project has been to make it more efficient. We have reviewed existing algorithms used for similar problems, and found two useful ones: TOP-MATA and Top-k Join. We have also created a new algorithm which we call the Segment Bounding Algorithm (SBA). The approaches were evaluated on two data sets – TREC and Opinosis – and the experimental results showed that SBA was the most efficient on the documents of TREC, while Top-k Join performed slightly better than SBA on the shorter documents of Opinosis. SBA was in the end proposed as an improvement of the simple summarization approach.

Keywords: summarization, multi-document summarization, similarity join, top-k similarity join, overlap similarity, similar pairs, similar documents

Acknowledgements

I would like to thank our examiner Prof. Peter Damaschke at Computer Science and Engineering department of Chalmers University for the opportunity to do this master thesis, and for all different ideas discussed during the writing of this thesis.

Johan Toft, Chalmers 2014

The first person I want to express gratitude to is our examiner at Chalmers - Peter Damaschke. He provided us with this interesting task, and spent time discussing ideas with us. Then I want to thank Johan for a great collaboration during this project. I also wish to thank *all* the friends and kind people who I have met during these five last years. Last but not least I want to thank my family and Markus; without them this would never have been possible for me.

Emma Bogren, Chalmers 2014

Contents

Chapter 1

Introduction

Large amounts of information are today available through the Internet, and one can only imagine what the situation will look like in the future. It can be unfeasible to process everything that is published online concerning a topic of interest, and a lot of time may have to be spent on going through redundant content. To deal with this problem, *automatic summarization* is a popular solution that has been researched since the 50's [6], and new and improved approaches are still being developed.

1.1 Automatic summarization

All of us are familiar with summaries; they are texts created from one or multiple sources and contain the most important and relevant information. A summary should preferably be rather short, and it can spare the reader quite a lot of work. It can be tedious for a human to create one by hand, and it would be satisfying to hand over this task to computers. This is the aim of automatic summarization, and today there already exist numerous algorithms for this.

There are many application areas in which this method can be of significant use: when researching scientific articles, when one quickly wants a summary of the latest news collected from different sources, or to obtain an overall opinion about a product or service. Automatic summarization can even be applied to multimedia data such as images, video, and sound. It can save time and effort in the everyday life of people in today's modern society, where large quantities of data are encountered everywhere.

Automatic summarization should sometimes be used with caution though, and not be trusted blindly, since it cannot be guaranteed that all relevant information will be included in the summary produced. In some cases it is necessary to process all the existing data to be sure nothing important is omitted. Examples can be when dealing with contracts, testimonies, or different types of documents where one deviating opinion or statement missed could prove disastrous. However, with this in mind, in most cases it is useful to quickly be able to obtain a summary.

1.1.1 Single- vs multi-document summarization

Automatic summarization can be separated into two categories: *single*- and *multi-document* summarization. The task of single-document summarization is to distinguish the relevant parts of one single document and to create a shorter version of it. In multi-document summarization, the input instead consists of many documents and the overall or most important statements or opinions are sought. This poses some extra challenges since sources might contradict each other, and contradicting opinions might not be presented accurately by the summarization tool [13].

1.1.2 Approaches

There are many ways in which we can approach the problem of creating a summary automatically, but most fit one of two main categories, *extraction*- or *abstraction-based* methods [14]. Extraction-based algorithms aim to find important parts of text(s) and reuse them to create a summary. This is often done by weighting words or sentences by looking at their position in the text, frequency, etc [14]. When extracting parts of documents and putting them together it is important to consider their relations to each other, or otherwise the resulting summary can turn out unreadable or even contain contradicting parts.

In abstraction-based summarization, the summary is a text that describes the document(s) in new words. This is more complicated, but gives more sophisticated summaries [14]. It usually includes some natural language processing, for example parsing sentences syntactically and then performing some pruning [14]. Today, it is also quite common to make use of machine learning algorithms to create summaries, both using extractive - and abstractive approaches. They can be trained to extract sentences by looking at the documents and pre-written summaries [14]. Many algorithms that use machine learning have been shown to produce satisfying summaries, but they can also be very complex. A thorough survey over machine learning approaches can be found in [6].

1.1.3 Evaluation

Evaluation of a summary system is difficult since no summaries are ideal, so what should be used as a baseline? Different methods for measuring the quality of a summary are available; it can be evaluated both by machines and humans. The latter can be both time- and money consuming though. *ROUGE* [18] is a set of metrics that has become standard for automatic evaluation. The metrics compare in various ways summaries created by an algorithm with human-written ones.

1.1.4 A simple approach to summarization

Many of the existing approaches to automatic summarization for text documents are time consuming and/or complex. A simple approach to multi-document summarization is currently being researched, and it has shown promising results [5]. It has even performed better than some machine learning approaches for the data that it has been evaluated on, which consisted of short human-written product and service reviews [5].

The idea of the approach is to find the pairs of documents that share the most words, counting each word only once per document, which should indicate that they are similar. Then the shortest of these is used as a summary for all the documents. The method is straightforward but effective.

1.2 Problem and aim

A problem with the mentioned approach to multi-document summarization is that it is slow; currently the most similar pairs are generated in a brute-force manner. For every pair of documents, every word in them is checked to calculate how many the documents share. For large collections of documents this naive method is prohibitively expensive, since the running time scales quadratically with increasing number of documents. In order to make the summarization approach usable, it is necessary to find a more efficient way of finding similar pairs.

The aim of this thesis is to research methods to speed up the running time of the approach. Existing solutions to similar problems can be researched, and new algorithms could be developed. The algorithms that we find will have to be evaluated and compared to each other in order to find the most appropriate one(s). We wish to be able to present a method that makes it feasible to create summaries from large sets of documents. The solution has to be exact, since we want to be able to guarantee good summaries.

1.3 Method

The project started out with a background research of existing methods for the problem and similar ones, since we were uncertain whether or not the task had been approached before. Some methods were found early, but turned out to be inapplicable. Later, two useful solutions called *Top-k Join* and *TOP-MATA* were found and modified for our purposes. New algorithms were also developed by exploiting properties of the text data and by using efficient data structures, but many of them did not end up as finished solutions because of various problems that we encountered. For example, we used some heuristics that worked well but made the algorithms approximate. We were in the end able to produce an algorithm called the *Segment Bounding Algorithm*, which always returned the correct answer.

The effectiveness and behaviour of the algorithms that were regarded as applicable were evaluated experimentally on two data sets after implementation. The sets were the *Opinosis* data set, which consisted of short extracts from reviews of products and services, and the *TREC* data set, containing somewhat longer abstracts from medical journals. We compared running times and other behaviours and could come to a conclusion about which algorithm to use to improve the running time of the summarization approach.

1.4 Limitations

In our research, we cannot guarantee that we will find all existing relevant approaches to our problem, but we intend to cover the most famous ones, and touch most methods in some way. The data that will be used for evaluation will consist of text documents of human utterances, in English, since this is the kind of data that the summarization approach has been evaluated on and aims to create summaries for. We will not be able to guarantee that the algorithms will work well on other types of data - especially not if it cannot be approximated with *Zipf's law*, an empirical law that we will look at later in the report. We will not evaluate the quality of the produced summaries or the viability of the suggested approach. The main concern is only to find an effective algorithm for finding the most similar pairs of documents.

1.5 Outline

We start off by introducing the reader to the background theory and we deal with matters that are important before starting to discuss solutions. For example, how to represent documents, and we look at how we can define similarity. We also take a look at the naive algorithm in detail and discuss properties of our data. In Chapter 3 we investigate existing approaches to our problem and similar ones, and in Chapter 4 we develop a new algorithm. Then we present the results from our evaluations of the approaches and discuss them in Chapter 5. In Chapter 6 we conclude and suggest a solution to the problem of efficiently finding the most similar pairs of text documents.

Chapter 2

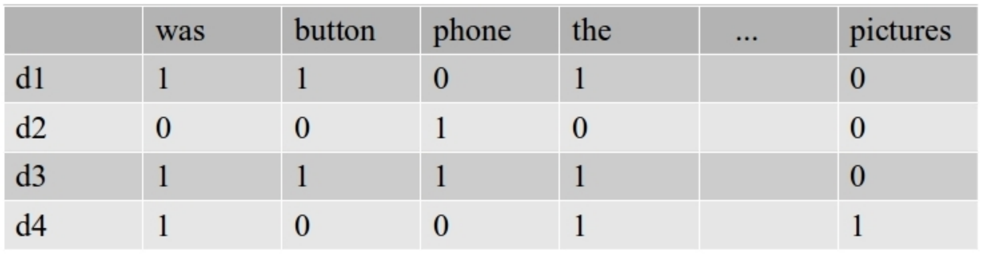
Background

In this chapter we will explain the details of the simple approach to multi-document summarization which this thesis aims to improve, and we will discuss some minor modifications that we can perform right away. To be able to do this we first need to deal with the issue of how to represent documents, and we also give an introduction to similarity measures. Then we take a look at the data that we will be using and show some important properties that might be exploitable or need consideration when we create an improved algorithm for the approach.

2.1 Representation of documents

Which representation of documents to select depends in what way our algorithms will use them and what information we need to preserve. In our case we are interested in *which* words occur in a document as opposed to *how many times* and *where* in the documents they occur. We will choose a commonly used representation called a *term-document matrix*, and we will also show how we will modify this to be more space efficient.

A term-document matrix is used to represent an entire document collection and the terms in them. Each row corresponds to a document, and each column to a word, and in our case an entry (i, j) will contain the value 1 if document i contains term j, and 0 otherwise. It is also common to let the entries store the number of occurrences of a word in a document, but this is not of interest for us. As an example of how we create a term-document matrix, we can imagine the following short documents, which are made-up



	was	button	phone	the	...	pictures
d1	1	1	0	1		0
d2	0	0	1	0		0
d3	1	1	1	1		0
d4	1	0	0	1		1

Figure 2.1: Term-document matrix representing the documents

product reviews of a smartphone:

There was no button for the number five (d1)
Great phone! (d2)
Who created this phone? The five button was missing... (d3)
The design was nice, and it took high quality pictures. (d4)

The documents could produce the term-document matrix illustrated in Figure 2.1, where the words have been given an arbitrary ordering, and some of them have been left out for simplicity. To represent a single document we can use the *Vector Space Model*, and view each document as a vector in a vector space. Such a vector will contain one element (dimension) for each word in the term-document matrix, which will be either 0 or 1 depending on the values in the row of the corresponding document. From the matrix in Figure 2.1, we would get the following vectors:

$$\begin{aligned}
\mathbf{d1} &: < 1, 1, 0, 1, \ldots, 0 > \\
\mathbf{d2} &: < 0, 0, 1, 0, \ldots, 0 > \\
\mathbf{d3} &: < 1, 1, 1, 1, \ldots, 0 > \\
\mathbf{d4} &: < 1, 0, 0, 1, \ldots, 1 >
\end{aligned}$$

The problem with the term-document matrix is that it will be very sparse; most of the entries will contain the value 0. To save space, we will use another representation in our implementations which is also more practical. We will use a list for each document that will consist of the column indexes in the term-document matrix in which the row corresponding to the document contains the value 1. The indexes in each list will always be sorted by ascending order. To illustrate this, we assign column indexes to the words of the documents from our earlier example from 1 to 22, assuming that there are 22 unique words in total. This gives us the following index lists:

$$\begin{aligned}\mathbf{d1}\ &:[1, 2, 4, \ldots]\\ \mathbf{d2}\ &:[3, \ldots]\\ \mathbf{d3}\ &:[1, 2, 3, 4, \ldots]\\ \mathbf{d4}\ &:[1, 4, \ldots, 22]\end{aligned}$$

We will use the the Vector Space Model to represent our documents in the next section about similarity measures, since they make the examples simpler, but in our implementations we will always use the index lists that we just discussed. It saves space since we remove all the entries that were filled with 0's, and is also often more practical.

The algorithms that we will be looking at in this project will all take as input the index lists of the documents, but we will refer to them as a term-document matrix. The index lists are created by first scanning all the documents and splitting them up into words. Characters which are not letters are removed, and the rest are converted to lower case letters. A global word vocabulary of every word and its respective frequency is kept.

Some algorithms will require that the words are ordered by ascending or descending frequency; in these cases the vocabulary is sorted. If we do not require any specific ordering, we will simply assign indexes to the words as they are scanned from the documents. The index lists are then created by iterating the (potentially) sorted version of the vocabulary, and by appending an index to all vectors which corresponding documents contain the word. To be able to do this, we use *inverted indexes*, that we have constructed during the scanning of the documents. An inverted index is a data structure mapping a word to the documents that it occurs in.

2.2 Similarity measures

The idea of the automatic summarization algorithm that we work with in this project is, as stated before, to find the most similar pairs of documents and create a summary from these. In the next section we will explain the details of this method, but first we need to give a more exact definition of *similar* pairs. Similarity can be described as "[...] a mean to quantify the common information shared by two objects." [3]. There are many ways in which we can measure it, but we are interested in one particular measure that is often referred to as the *Overlap similarity*. We will now present some common measures for similarity, but also *dissimilarity* (or distance) which can be thought of as an inverse of the former, that will be of use later in the report and we will also explain the Overlap similarity more thoroughly.

2.2.1 Jaccard similarity

The first measure that we will be looking at is a similarity measure called the *Jaccard similarity.* The Jaccard similarity of two vectors $d1$ and $d2$, representing two documents, is defined according to equation 2.1, which in words is the intersection size between $d1$ and $d2$ divided by their union size.

$$J(d1, d2) = \frac{|d1 \cap d2|}{|d1 \cup d2|} \tag{2.1}$$

Each word is included in each of $d1$ and $d2$ at most once, so the similarity will be the number of unique words the documents share divided by the total number of unique words in them, and this will give a value between 0 and 1. The Jaccard similarity gives us the percentage of all unique words that both documents include, which means that two documents that together contain many words need to have a larger intersection compared to two documents that contain fewer words in order to get a large similarity.

2.2.2 Hamming distance

Another measure is the *Hamming distance*, which does not measure similarity, but dissimilarity. It applies to strings, including binary strings that we can create from the vectors representing our documents. For two strings, that have to be of the same length, the measure is defined as the number of positions at which they differ. For instance, assume we have two vectors corresponding to documents called $d1$ and $d2$:

$$d1 =< 0, 1, 0, 0, 1, 1 >$$

$$d2 =< 1, 0, 0, 1, 1, 0 >$$

We can see that the vectors have different values on positions 1, 2, 4 and 6, which means that they have a Hamming distance of 4. The longer two documents are, the larger can their Hamming distance be.

2.2.3 Cosine similarity

The *Cosine similarity* of two vectors is defined as the cosine of the angle between them. The angle of two vectors $d1$ and $d2$ can be obtained through equation 2.2, which is their inner product divided by their magnitudes multiplied together.

$$COS(d1, d2) = \frac{d1 * d2}{\|d1\| \|d2\|} \tag{2.2}$$

The similarity will range between 0 and 1 for positive elements, which is our case, and the larger value the more similar they are. If two vectors get a value of 1, it would mean that their angle would be 0 degrees and they are one and the same vector. If they instead got the value 0 it would indicate that they are orthogonal since their angle would be 90 degrees.

2.2.4 Overlap similarity

When we state that we want to find the most similar pairs of documents in this thesis, we mean that they should be similar as defined by a measure called the *Overlap similarity.* The Overlap similarity of two vectors $d1$ and $d2$ is simply defined according to equation 2.3.

$$Overlap(d1, d2) = |d1 \cap d2| \tag{2.3}$$

It is the number of words that the document vectors share, which is similar to the Jaccard similarity, but we do not divide by the total number of words. This implies that the similarity values will not lie between 0 and 1, but instead between 0 and min($|d1|$, $|d2|$). The more words two documents share, the more similar they are. This means that longer documents have the potential of achieving a larger similarity than shorter ones.

The Overlap similarity as we define it should not be confused with another common measure that goes under the same name, and is defined as equation 2.4 [26]. Here we also divide the size of the intersection by the length of the shorter document.

$$\frac{|d1 \cap d2|}{min(|d1|, |d2|)} \tag{2.4}$$

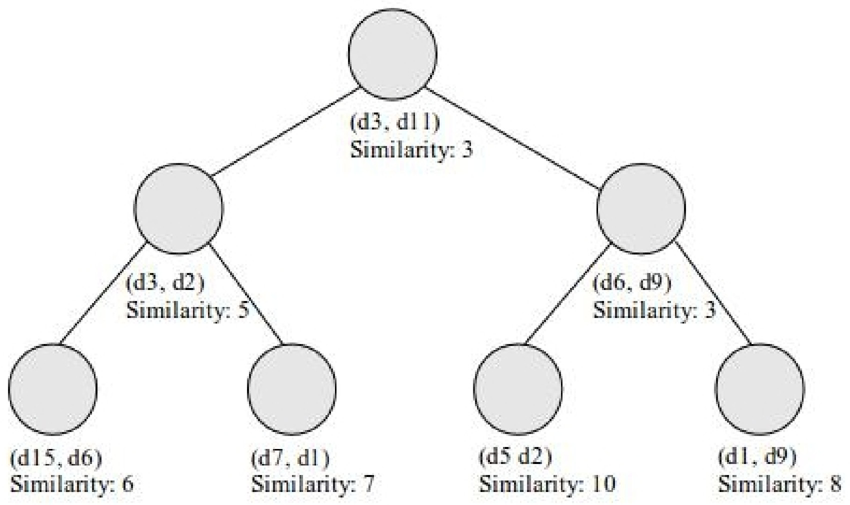


Figure 2.2: An example of the min-heap at execution

2.3 Finding similar document pairs

We now have all the knowledge essential to be able to discuss the simple but naive algorithm for summarization, and we will motivate the need for improvement. The algorithm takes as input a term-document matrix, and outputs the k pairs that have the highest similarity values (ties broken arbitrarily). From these, a summary will be created.

The current version of the algorithm considers each and every word (column in the term-document matrix) for every pair of documents that can be created. It adds 1 to the similarity if both entries in the rows corresponding to the two documents in the pair contain the value 1. To keep track of the k calculated pairs that currently have scored the highest, we store them in a *min-heap*. A min-heap is a tree-based data structure in which the pair with the lowest similarity can be efficiently accessed – $O(1)$ – since the heap always satisfy the property that each node (pair) has a greater or equal similarity value than its parent. This is important since this is the similarity a pair must beat in order to be inserted to the heap. An example of what it could look like during execution of the algorithm is illustrated in Figure 2.2.

The time complexity of the algorithm is bad; if we denote the number of documents or rows in the term-document matrix as n and the total number of words or columns as m, it will be $O(m \cdot n^2)$, since we have $\frac{n(n-1)}{2}$ pairs of documents, which is $O(n^2)$, and m words to go through for each. In the application area of summarization, both n and m can be large and this can lead to running times making the approach impracticable. A person who desire a summary in order to save some effort and time not having to read all the documents does probably not want to spend hours waiting for one to be produced.

In Section 2.1 we mentioned that we could represent the original term-document matrix more efficiently in order to save space. We can use lists consisting of the indexes of the words that are in the documents. If we do this, it is not necessary to iterate all m words, but merely the words included in any of the two documents. The indexes occur in the vectors in ascending order, and therefore we can process them in that order and if we reach the last index of one of the vectors then we do not need to continue comparing them, since they will share no more indexes.

Another important thing to observe is that many words might only occur in a single document. These words cannot possibly contribute to the similarity score of any pair, so we might as well get rid of them before we hand the index lists to the algorithm. This can be done at the same time as we construct the term-document matrix. In Section 2.1 we mentioned keeping a word vocabulary of words and their frequencies. After scanning all the documents, this vocabulary can be used to prune these words.

Later on in the report, our algorithms will take a term-document matrix as input. We will then, as stated earlier, assume that it is already on the index list-format, and that all the words occurring only in one document have been removed. This will apply to both the naive algorithm, and to the more efficient methods that we may find - if nothing else is stated.

2.4 Properties of text data

Before we start working on finding an efficient solution to our problem, we take a closer look at our data. This will consist of text documents written in natural language, and more specifically it will be product reviews and medical abstracts written in English. This type of data, and many others, has a property that is probably important if we should be able to find a more efficient solution to our problem. It follows a principle called *Zipf's law*.

Zipf's law is a law based on the results of extensive experiments, and it states that many types of data can be approximated with a power law distribution called a *Zipfian distribution* [28]. Given a set of data, if $x(k)$ is the total number of occurrences of the kth most frequent object, and x_M is the number of occurrences of the most common object, Zipf's law is defined by equation 2.5 [4].

$$x(k) = x_M/k \tag{2.5}$$

Zipf's law applies well to English texts [17], and states that the frequency of a word is related to its frequency rank. The word that appears most times in the text will occur approximately twice as often as the second most frequent word, three times as often as the third most frequent one, and so on.

We represent our documents using index lists in which the index of a word can be contained only once. In addition, we remove the words that only occur in one document. This alters our data, and the question is whether or not Zipf's law still applies to it. Zipf's law is actually said not to hold for *subsets* of Zipfian data [4], and especially not when the elements that are missing are very frequent ones.

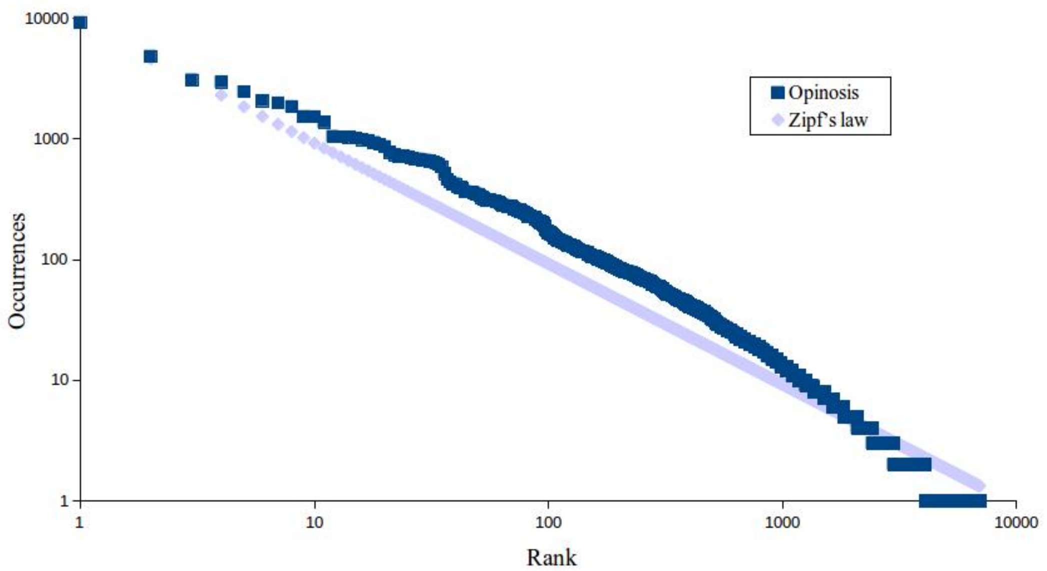


Figure 2.3: Word frequencies of terms in Opinosis

Figure 2.3 compares the behaviour of the unaltered version of the Opinosis data to the prediction of Zipf's law. The light line shows the approximation and the darker line shows the actual frequencies. The data does not follow the approximation exactly but it clearly exhibits a Zipf-like behaviour.

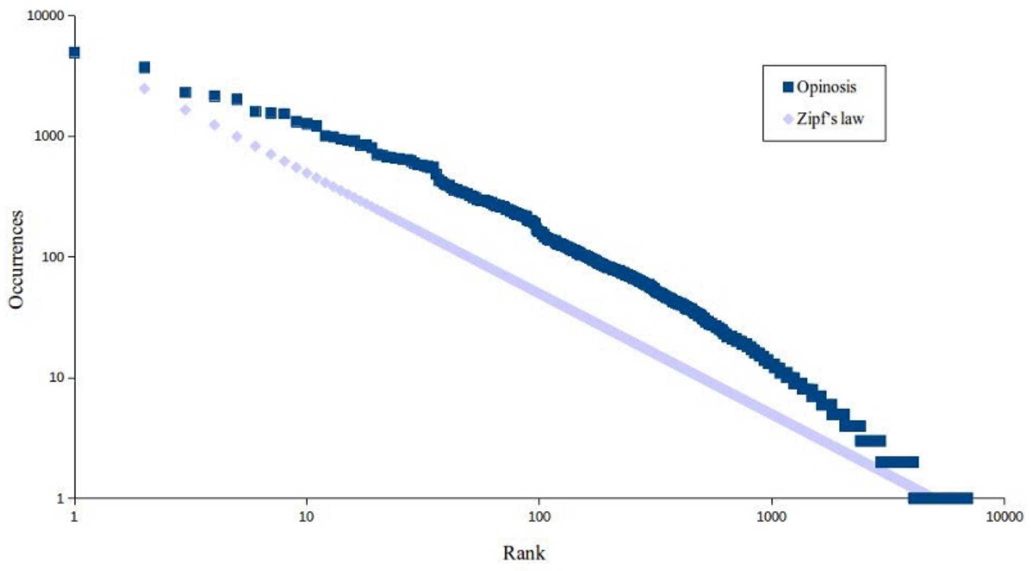


Figure 2.4: Word frequencies of terms in Opinosis, counting each word once per document

We then look at Figure 2.4, in which we display the frequencies of the words of the Opinosis data when we only count each word once per document, to capture the behaviour of the data when representing it with index lists. This data can be approximated by Zipf's law as well, even though it is a bit less accurate. If we also should remove the words only occurring in one document, the lines would end at a lower rank value, but the behaviour of the rest of the words would still exhibit the same behaviour. We do not include the same test for the TREC - data set, since the behaviour should be similar.

Chapter 3

Previous Work

We now have all the knowledge needed to be able to discuss solutions to our problem. We will begin by introducing the most relevant of the previous work that we have found. A few of the methods are close to what we are searching for, but can still not be applied for different reasons that we will discuss, but some are also found useful (after some modifications).

3.1 The similarity join problem

The problem of finding the most similar pairs of documents is closely related to the well-known *similarity join problem* [23, 24]. Similarity join is the operation of returning all pairs created from the records (representing documents in our case) of two given sets which have a similarity larger than or equal to a specified threshold [2]. One can also simply create the pairs from one set - a special case called *self-join* [23], which is also *our* case since we have a single set of documents. There are many existing algorithms for the similarity join problem, and we will soon look at an example called the *All-Pairs* algorithm.

A common method to reduce the number of pairs needed to be checked for similarity when approaching the similarity join problem is to apply the *prefix filtering principle* [1, 24, 23]. Prefix filtering makes use of the fact that if the elements in the records are ordered, for example by frequency, then some parts of two sets must intersect in order for them to have a similarity equal to or larger than a specified threshold. As Xiao et. al puts it: "Consider an ordering $\mathcal{O}$ of token universe $\mathcal{U}$ and a set of records, each with tokens sorted in the order of $\mathcal{O}$. Let the p-prefix of a record x be the first p tokens of x.

If $O(x, y) \geq \alpha$, then $(|x| - \alpha + 1)$-prefix of x and the $(|y| - \alpha + 1)$-prefix of y must share at least one token" [23]. Here, $O(x, y)$ refers to the Overlap similarity of x and y. In the simple case of Overlap similarity, α will be the same as the similarity threshold, and this is what we will focus on. As an example, assume that we have two records (lists of word-indexes) $d1$ of length 5 and $d2$ of length 4. In order for them to have an overlap of at least 3 elements (words or tokens), the (5-3+1)-prefix of $d1$ and the (4-3+1)-prefix of $d2$ must share at least one element. Otherwise they could not possibly share 3 elements in total. In the example lists of indexes below, the 3-prefix in $d1$ and the 2-prefix in $d2$ are highlighted.

$$d1 : [\mathbf{0}, \mathbf{1}, \mathbf{3}, 5, 6]$$
$$d2 : [\mathbf{1}, \mathbf{2}, 4, 6]$$

The prefix filtering principle is useful when generating pairs to be checked for exact similarity; given a value on α, it is only necessary to calculate pairs which share a token in their prefixes if they should be able to reach this threshold [24]. The pairs that satisfy this condition are called *candidate pairs.*

3.1.1 All-Pairs algorithm

The *All-Pairs algorithm*[1] is an algorithm which solves the similarity join problem. It uses the prefix-filtering principle to efficiently reduce the search space. It creates inverted indexes, which are built up gradually and are used to generate candidate pairs. The elements of each record d are sorted in order of frequency, and we access the ones occurring in their $(|d| - \alpha + 1)$-prefix one by one. For each element in the prefix of each record, we iterate over the records in its inverted index. These will share an element with d in their prefixes. We calculate the exact similarity of the pairs created using each such record and d together. Then d is added to each inverted index of every element being processed in the prefix. When we have iterated over all records, we return the pairs that turned out to have a similarity larger than or equal to the threshold.

The drawback with the algorithms for the similarity join problem is the similarity threshold that needs to be specified. If we knew the exact threshold needed to produce k pairs, this algorithm would be useful to us, but as it is, we would have to test different thresholds until we returned the number of pairs that we wanted. This would be an inefficient solution.

3.2 Top-k similarity join

In the previous section we introduced a problem called the *similarity join problem* which is similar to our problem, but requires the specification of a similarity threshold. Xiao et al. [23] realized that for some applications it could be useful not having to decide on a threshold before running the algorithm, and simply return the k most similar pairs of records. They created a new algorithm called the *Top-k Join algorithm*, which is based on the All-Pairs algorithm, but does not require a threshold to be defined.

The algorithm can be adapted for different similarity measures, and one of them is Overlap similarity [23]. However, this measure is only briefly mentioned in [23], and no test results are included. We therefore implement the algorithm using the Overlap similarity, and we also adapt it to work as well as possible for this measure in particular. We will now explain the algorithm in detail, whilst continuing to consider the self-join case in which we only have one set to create pairs from. We also assume in our examples that the records are index lists representing documents, since that is what we are interested in.

The Top-k Join algorithm is just like the All-Pairs algorithm based on prefix filtering, and we will only consider pairs which share a word in their prefixes. The difference from the problem discussed in the previous section is that the threshold will not be constant in this case - it will be gradually increased. We keep a min-heap, called the *result heap*, in which we will store the k currently most similar pairs that have been calculated exactly. The heap is initialized with k arbitrarily chosen pairs. In order for a new pair to be put on the heap, it needs to be more similar than the pair that is currently the kth best, and we will denote this similarity s_k. This value corresponds to the threshold in the similarity join problem.

The prefixes of the documents will initially consist of only one term. Each document will also be associated with a *similarity upper bound*, which is the maximum similarity any pair containing that document can have, if the currently last term in its prefix is the first one it shares with the other document in the pair. For a document d with prefix length p_d, the similarity upper bound will be defined according to equation 3.1. For the Overlap measure it will simply be the number of words that have not yet been added to the prefix of the document, plus 1 since we know that the documents share the last term in the prefix.

$$|d| - p_d + 1 \tag{3.1}$$

The algorithm keeps a max-heap which we will call the *event heap*, in which it stores one entry per document on the following form:

$$< id_d, p_d, sp_d >$$

id_d is the document id, p_d is the index of the last word in the current prefix, and sp_d is the similarity upper bound. The entry on top will be the one with the largest upper bound, which will initially be the one containing the longest document (in case of ties, the positions will be arbitrary). To see this, consider the very short index lists below. We create an entry for each, including their initial similarity upper bounds. For $d1$ this bound will be $|d1| - p_{d1} + 1 = 2 - 1 + 1 = 2$, for $d2$ we get $|d2| - p_{d2} + 1 = 3 - 1 + 1 = 3$, and for $d3$ the value will be $|d3| - p_{d3} + 1 = 4 - 1 + 1 = 4$. The values are equal to the lengths of the documents.

$$d1 = [0, 2]$$
$$d2 = [1, 2, 5]$$
$$d3 = [0, 1, 3, 5]$$

Top-k Join keeps inverse indexes for the words, just like the All-Pairs algorithm. It iteratively removes the top-element from the event heap and processes the inverted index of the last word in the prefix of the document in the entry. It will calculate the exact similarity between the document and each other document in the inverted index. If a similarity is larger than current s_k, the result heap will be updated. After that, a new entry is created where the prefix will be extended by one word, and the similarity upper bound will be decreased by 1. This is then inserted in the event heap, and we also insert the document to the inverted index. We can stop removing entries from the event heap either when it is empty or when the next similarity upper bound is smaller than or equal to s_k.

A pair of documents could share more than one word, and therefore we want to make sure that we do not calculate the exact similarity of a pair more than once. For this we keep all pairs which similarities have been calculated in a hash table. To avoid storing too many pairs in the hash table, we could make sure to only put pairs in it that would actually be calculated a second time. To check this we can figure out if the prefixes of maximum length of the documents have any words in common. The maximum length for a document d, when s_k is the currently kth largest similarity value, is given by equation 3.2. If it would be longer there would not be enough words in it to be able to beat s_k.

$$|d| - s_k + 1 \tag{3.2}$$

Worth noticing is that [23] sorts the elements in their records by *inverse frequency*, i.e. the least frequent element comes first. It is stated that this should decrease the number of candidate pairs, since if we consider pairs that share rare words then they should have a high chance of getting a large similarity value. As we discussed in Section 2.4, for data following Zipf's law, the frequent elements will occur much more often than the rare ones. The intuition is that if we look at the frequent elements first it would not say much about the total similarity of the records, since so many of them share these elements. But if we consider the rare elements first we can assume that pairs sharing these are also likely to share frequent elements, making them good candidates.

The pseudocode in Algorithm 1 describes the Top-k Join algorithm. The code includes our modifications of the algorithm which adapt it for the Overlap similarity. Lines 2-5 is initialization of the data structures that we have mentioned. Then between lines 7-11 we check whether the similarity upper bound of d - the document in the entry on top of the event heap - is larger than s_k. If it is, the algorithm will terminate, but otherwise we look at the word at position p_d in d.

In lines 14 to 23 we iterate the documents in the inverted index of the word $d[p_d]$, and in lines 16 to 18 we calculate the similarity of a pair (d, d') if it has not already been done earlier. Then we save the pair in the hash table H, containing all previously calculated pairs. The result heap is then updated if needed. When we are done with the inverted index we store the document d in it, and then we create and insert a new, updated entry in the event heap in line 25. In the end the algorithm will return R - a heap containing the k most similar pairs.

Algorithm 1 Top-k Join

Input: *tdm* - term-document matrix sorted by inverse frequency
Input: *k* - number of pairs to be returned
Output: *k* most similar pairs

```
1: function TOPKJOIN($tdm, k$)
2:     $I \leftarrow$ empty inverted indices
3:     $E \leftarrow$ event heap with one entry per document
4:     $R \leftarrow$ result heap initialized with $k$ arbitrarily chosen pairs
5:     $H \leftarrow$ empty hash table of calculated pairs
6:     while $|E| > 0$ do
7:         $(id_d,\ p_d,\ sp\_d\ ) \leftarrow E$.pop()
8:         $s_k \leftarrow$ similarity of $R$.top()
9:         if $sp\_d \leq s_k$ then
10:            break
11:        end if
12:        $d \leftarrow tdm[id_d]$
13:        $w \leftarrow d[p_d]$
14:        for $(id_{d'}, p_{d'})$ in $I[w]$ do
15:            $d' \leftarrow tdm[id_{d'}]$
16:            if $(d, d')$ not in $H$ then
17:                $s = \text{overlap}(d, d')$
18:                $H$.add$(d, d')$
19:                if $s_k < s$ then
20:                    $R$.update$(s)$
21:                end if
22:            end if
23:        end for
24:        $I[w]$.append$(id_d)$
25:        $E$.push$(id_d,\ pd + 1,\ sp_d - 1)$
26:    end while
27:    return $R$
28: end function
```

3.2.1 Suffix filtering

[23] mentions an implementation detail that one can also optimize the Top-k Join algorithm using *suffix filtering* and *positional filtering*. These kinds of filtering are described more in detail in [24]. Positional filtering does not make the algorithm faster for the Overlap similarity though, but suffix filtering may be beneficial [24].

The suffix filtering step is intended to filter out pairs before calculating their exact similarity by investigating the suffixes of the documents. The suffixes are simply the parts of the documents that does not yet belong to the prefixes. For the suffixes we will not yet have calculated any partial similarity, and we will not have added the words in them to inverted indexes. Still, we want to be able to estimate whether the similarity of the suffixes can make the total similarity large enough to beat s_k. We do this by calculating a lower bound on the *Hamming distance* of the suffixes of pair, which we then compare with a maximum allowed Hamming distance based on s_k. The Hamming distance measure was described in Section 2.2.2.

For each candidate pair, we convert the current value of s_k into a maximum allowed Hamming distance threshold. We then create lower bound of the Hamming distance of the suffixes of the documents and compare with the threshold. If the distance is higher than the threshold it means that we can skip calculating the exact Overlap similarity of the pair since we know that it will not be more similar than s_k. The maximum allowed Hamming distance can be created according to equation 3.3 in the Overlap case [24]. Here, x and y are documents, and i and j are their prefix indexes respectively.

$$H_{max}(x, y) = |x| + |y| - 2s_k - (i + j - 2) \tag{3.3}$$

This is the maximum distance allowed for the suffixes in order for the pair to be able to be more similar than s_k. A simple estimation of the lower bound on the Hamming distance we can get by comparing the lengths of S_x and S_y according to equation 3.4, where S_x is x's suffix.

$$H_{low}(S_x, S_y) = \Big||S_x| - |S_y|\Big| \tag{3.4}$$

In order to change one document into the other it is necessary to at least add the number of more words included in the longer document. This is quite a weak lower bound though, and to improve it [24] suggest to first choose any word w in S_x, and then to split S_x into two lists S_{xr} and S_{xl} at this word.

Then the word or its ordered successor is found in S_y, splitting it into S_{yr} and S_{yl}. We can then estimate a more accurate lower bound using equation 3.5.

$$H_{low}(S_x, S_y) \geq \Big| |S_{xl}| - |S_{yl}| \Big| + \Big| |S_{xr}| - |S_{yr}| \Big| \tag{3.5}$$

The lower bound of equation 3.4 is applied to the left parts and the right parts independently, and the results are then added together. By considering smaller parts of the documents we may get results closer to the real Hamming distance.

We can also choose more than one token w and repeat the test several times to get more strict filtering. The results from previous tests can be used to give more accurate results for the upcoming ones. This is done recursively, by picking a word w and computing the lower bounds of both the left and right parts of the documents independently. For each part we will check if we are still within the allowed distance and in that case we continue on the part by splitting it up into two new parts.

3.2.2 Our implementation

No test results for the Overlap similarity were included in [23], and the details for this measure are not described very thoroughly. We found several steps that we considered unnecessary due to the properties of the Overlap measure. First of all, we did not need to include the step referred to as "optimization for indexing". In this step we would stop inserting documents into the inverted indexes if the current document on the event heap had an *indexing upper bound* smaller than s_k. For the Overlap similarity this bound will be equal to the similarity upper bound of the document, which would make such a check redundant. If the similarity upper bound would be smaller than s_k, the algorithm would already have stopped since it is the first check we do after removing the document from the event heap.

[23] also creates an *accessing upper bound*, which is used to limit the access to the inverted indexes. It is defined in equation 3.6. They state that we do not need to continue considering documents in an index if this bound is smaller than s_k, for some document y with similarity upper bound sp_y in the index. Since we process the documents on the event heap by monotonously decreasing similarity upper bound, it is also said that we can remove all the entries inserted earlier than y. We find this accessing optimization to be redundant though. sp_x can never be smaller than s_k, since then we would

already have stopped, and neither can sp_y - since y was inserted earlier than x, which means that its similarity upper bound will be larger than sp_x.

$$min(sp_x, sp_y) \tag{3.6}$$

[23] also includes a step of size filtering to their algorithm which lets us skip calculating the exact similarity between x and a document y in an inverted index list if the length of y should be smaller than s_k. This would mean that the pair could not have a similarity large enough to be inserted to the result heap. This check is also redundant since if the length of y would be smaller than s_k, then so would the length of x. We are accessing the documents from the event heap in monotonously decreasing length, since their similarity upper bounds are exactly the number of remaining words in their suffixes. If x would be shorter than s_k we would already have stopped the algorithm, since its similarity upper bound would be too small.

3.3 An algorithm for Cosine similarity

Most algorithms for finding similar pairs using the *Cosine similarity* – described in Section 2.2.3 – requires a threshold to be defined in a way similar to the algorithms for the similarity join problem described in Section 3.1. This was something that [27] noticed, and they instead created two algorithms for this measure that found the top-k pairs. The first one, TOP-DATA[27], is based on diagonal traversal of matrices, inspired by and a refinement of the TOP-COP algorithm [25]. We will here consider the second one, since it is an improvement of the TOP-DATA algorithm and has shown a better performance [27]. This algorithm is called TOP-MATA, and it basically creates upper bounds for pairs of documents and employs a max-first strategy based on the bounds during the search for similar pairs.

Using the Vector Space Model for representing a document d, [27] defines the *support* according to equation 3.7, where d_i is the value - either 0 or 1 - of the word on position i in the vector. N is the total number of dimensions in d, which is the same as the total number of columns in the term-document matrix that includes d. An upper bound on the Cosine similarity of two documents $d1$ and $d2$ is then created using this definition, and is stated in equation 3.8, where it is assumed that $supp(d1) \geq supp(d2)$.

$$supp(d) = \frac{1}{N}\sum_{i=1}^{N} d_i \tag{3.7}$$

$$COS(d1, d2) \leq \sqrt{\frac{supp(d2)}{supp(d1)}} \tag{3.8}$$

If S is the set of all our documents, the TOP-MATA algorithm starts off by creating a matrix $M = S \times S$ containing n^2 pairs. Since half of these are, in our view, the same pairs as the other half - the pair $(d1, d2)$ is equivalent to $(d2, d1)$ - we can remove the lower triangle matrix [27], as shown in figure 3.1. The documents appear in the rows and columns of the matrix in order of decreasing support. In the example shown in figure 3.1, this would mean that the document $d2$ has the highest support, while $d4$ has the lowest.

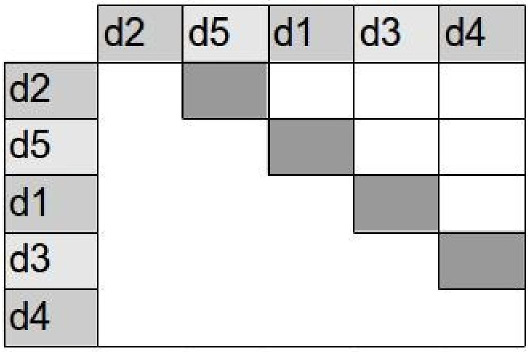


Figure 3.1: Example matrix created by the TOP-MATA algorithm

TOP-MATA keeps a max-heap containing the largest upper bound for each row in the matrix. This will at start be equal to the bounds of the left-most entries in the rows, highlighted in grey color in figure 3.1, since the numerator (the lowest support in the pair) in equation 3.8 will decrease as we look at the columns to the right of this. This means that the whole bound will decrease.

The algorithm removes the top-element from the max-heap and calculates the exact similarity of the pair in that entry, and all other entries, if any, on the same row with the same upper bound. Then a new maximum upper bound for the row is calculated, which will be the bound of the entry in the left-most entry on the same row, still not considered. This new value is inserted into the max-heap.

Similar to other earlier mentioned algorithms, we also keep a heap containing the k best results so far. We call the lowest similarity in this heap s_k, and pairs that we calculate must beat this to be inserted. We will continue removing top-entries from the max-heap containing the upper bounds until we encounter one which is smaller than or equal to s_k.

This algorithm cannot be immediately applied to our problem, since we are interested in the Overlap similarity - not the Cosine similarity. It can be adapted for this measure though, if we define a new upper bound. This upper bound is shown in equation 3.9, where *d1* and *d2* are documents. The Overlap similarity of two documents can at most be the length of the shortest of the two.

$$Overlap(d1, d2) \leq min(|d1|, |d2|) \tag{3.9}$$

We also sort the documents in the matrix by decreasing length instead of decreasing support, and then the algorithm works in the exact same way as for the Cosine measure; we will therefore not include any pseudocode for it.

Zhu et al also mentions the *TKPC*-algorithm [15], which is based on an FP-tree data structure, and they compare their TOP-MATA algorithm with it. Since TOP-MATA is much more efficient when k is small (550 in their experiments) compared to the total number of pairs[27], we will not consider this algorithm, since we are not interested in very large values of k.

3.4 Locality-sensitive hashing

Locality-sensitive hashing (LSH) is a probabilistic approximate approach commonly used for problems when we want to find similar documents in large collections of data [21], [16]. Some applications involve the *Nearest Neighbour problem*, where we want to find the item that is most similar to a given query item [8], and clustering [16]. The approach is related to our problem, but since it only gives approximate solutions and cannot guarantee a correct result, we cannot make use of it. We will still mention it briefly in this section though, since it could be interesting in cases when an approximate solution is enough.

The idea of LSH is to hash items (the representations of our documents) into buckets in a way so that similar items end up in the same bucket with high probability and dissimilar items in the same bucket with low probability [20]. We need to define a family of locality-sensitive hash functions for our similarity (or distance) measure [7]. Such a family is said to be $(\gamma, \phi, \alpha, \beta)$-sensitive if the equations 3.10 and 3.11 hold for any hash function h in the family. $d1$ and $d2$ are two items (documents), d is a distance measure (although we could also give a definition for the case when we have a similarity measure), and γ and ϕ are distance values. The following conditions should also be fulfilled: $\alpha > \beta$ and $\gamma < \phi$ [20], [8].

$$d(d1, d2) \leq \gamma \rightarrow p(h(d1) = h(d2)) \geq \alpha \tag{3.10}$$

$$d(d1, d2) \geq \phi \rightarrow p(h(d1) = h(d2)) \leq \beta \tag{3.11}$$

When the distance is small we want the probability that the items hash into the same bucket to be large, and when it is large we want the probability to be small. When performing the locality sensitive hashing, we can concatenate multiple of the hash functions in order to increase the gap between α and β [8].

There are many different methods for how to construct locality sensitive hash functions, and we will give a simple example mentioned in [20], using the Hamming distance as our distance measure. We will assume that we have documents on the form given by the Vector Space Model. If we have two documents *d1* and *d2* consisting of n dimensions, we create a hash family $\{h_1, ..., h_l\}$, where $h_i(d1)$ is the ith element of *d1*. Then equation 3.12 will hold for any hash function h_i in the family. For instance, if *d1* and *d2* have 10 dimensions and differ in 5 positions, the probability that the hash function will hash them into the same bucket will be $1 - 5/10 = 1/2$. This lets us define α and β according to equations 3.13 and 3.14, for some $\gamma < \phi$, and we have thus found a $(\gamma, \phi, \alpha, \beta)$-sensitive family.

$$p(h_i(d1) = h_i(d2)) = 1 - \frac{d(d1, d2)}{n} \tag{3.12}$$

$$\alpha = 1 - \gamma/n \tag{3.13}$$

$$\beta = 1 - \phi/n \tag{3.14}$$

3.5 A MapReduce approach

MapReduce [9] is a framework introduced by Google for processing large data, and programs using it are based on two types of functions: *mappers* and *reducers*. The problem that the program is trying to solve is divided into subproblems which are distributed over machines which each performs some task on it and returns the result. This is the *map-step* that generates key-value pairs. The *reduce-step* is then to put together all the sub-results. The MapReduce model provides a relatively simple way to do batch processing on big data problems by distributing computations (map and reduce - jobs) across a cluster of computers. The MapReduce runtime takes care of scheduling, communication between machines, etc.

[19] realized that the MapReduce model could be used for computing pairwise document similarity. However, to compute exact document similarity they suggest a brute force approach. An *embarrassingly parallel problem* is "a computation [that] consists of a number of tasks that can execute more or less independently, without communication" [11]. These kinds of problems are suitable for the MapReduce model, and this is the case for generating and computing the similarity of all pairs.

The main steps in the algorithm is to first do a mapper-step where a pair is created for each document and word in it. Then these are grouped and reduced so that each word is mapped to a list of documents that contain the word (inverted indexes). Then in another mapper-step, pairs are created for each inverted index out of all the documents in it. Each such pair would in our case with the Overlap similarity be paired up with a score of 1. These are then sorted and then the pairs are reduced to obtain the total score of each pair. This is simple, but the approach still scales in a $O(n^2)$ fashion.

Some suggestions are presented in [19] in order to reduce the time taken to process the document corpus. By removing a certain percent of the terms, namely the most frequent ones, they do achieve a linear scaling [10]. The downside is that the result is not the exact document similarity, which we want, and therefore this approach is not applicable in our case. It is not possible to keep the terms and obtain a good running time, since their results show that as the percentage is decreased the running time approaches $O(n^2)$ behaviour.

Chapter 4

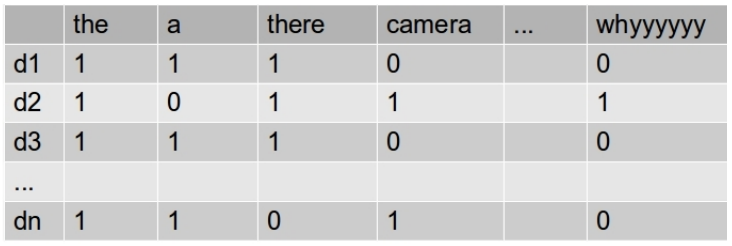
The Segment Bounding Algorithm

The approaches we have discussed until now have been previous work, and we have been able to apply some of them with a few modifications. In this chapter we look at some of the ideas that were developed during this project and we arrive at a new and useful algorithm called the *Segment Bounding Algorithm* (SBA).

4.1 Segmenting the TDM

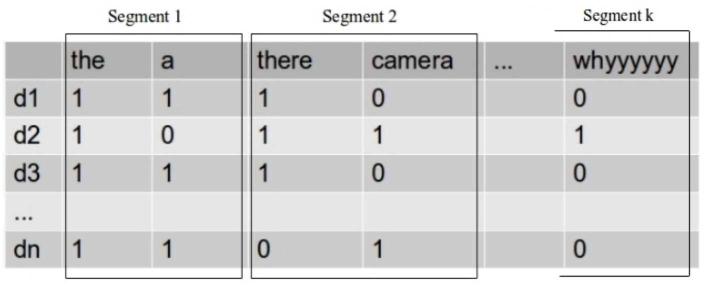
Instead of considering each term-document matrix created from our documents as a whole unit to be searched, we thought of the idea to divide it into smaller segments. Then it could be possible to find similar pairs in these independently, and then somehow combine the results to find the overall winners. Each segment would consist of some words and when comparing the documents to each other within a certain segment, we would only consider these words.

To be able to explain the idea of segmentation in a straight-forward way, we for now assume that the documents are represented by a sparse term-document matrix rather than the more compact index lists. Imagine that we are provided with the matrix in Figure 4.1, in which some columns and rows have been left out for simplicity. In order to split this up into segments, we define a parameter r, which value will determine the length of them. To create the first segment, we add as many words to it as needed for at least



	the	a	there	camera	...	whyyyyyy
d1	1	1	1	0		0
d2	1	0	1	1		1
d3	1	1	1	0		0
...						
dn	1	1	0	1		0

Figure 4.1: Term-frequency matrix example


Segment 1
Segment 2
Segment k

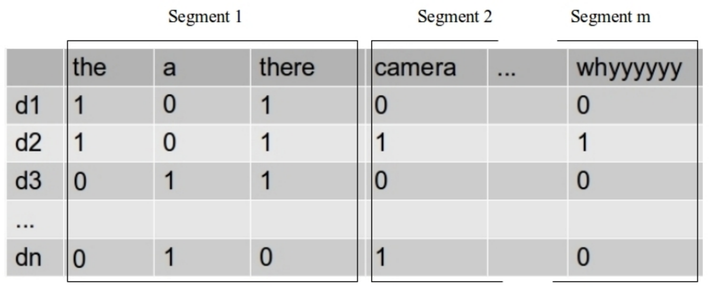
	the	a	there	camera	...	whyyyyyy
d1	1	1	1	0		0
d2	1	0	1	1		1
d3	1	1	1	0		0
...						
dn	1	1	0	1		0

Figure 4.2: Term-frequency matrix divided into segments

one document to have r entries in the matrix with value 1 amongst those columns. We then continue with the second segment, and so on until no document has r more words. The last segment might look a bit different compared to the other ones, since there is no guarantee that any row in the matrix will contain r 1's in it.

If we would create segments for the term-document matrix in Figure 4.1, using $r = 2$, it would give the result shown in Figure 4.2. Some segments have been left out since not all of the columns of the matrix are displayed. We can clearly see that we have at least one document in each of the two first segments with r 1's. If we instead assume that we were provided with the matrix in Figure 4.3, and we imagine that none of the n rows contains two 1's in the two first columns, the first segment would look different. We would need to include the three first words in order to get at least one document with the two words included in it.

When we have created a segment, we would like to find all the pairs of document that have high Overlap similarity inside it. One way to do this is to generate all the subsets of the words of each document and then insert them all into a hash table. When we hash a subset, a collision in the table



	Segment 1			Segment 2		Segment m
	the	a	there	camera	...	whyyyyyy
d1	1	0	1	0		0
d2	1	0	1	1		1
d3	0	1	1	0		0
...						
dn	0	1	0	1		0

Figure 4.3: Term-frequency matrix divided into segments

will indicate that at least two documents share the words in it (unless there is a collision of two different subsets, which should not occur too often).

An entry in the hash table will store all the id's of the documents that contain the subset of words which is hashed to this index. From now on we will refer to the event of multiple documents sharing a subset of words as a *collision*. In Figure 4.4 we can see an example of how such a hash table could look like. The documents called $d3$ and $d7$ share a subset of words with column number 1,2,5, and 7, and therefore they will both be stored at the index of this subset.

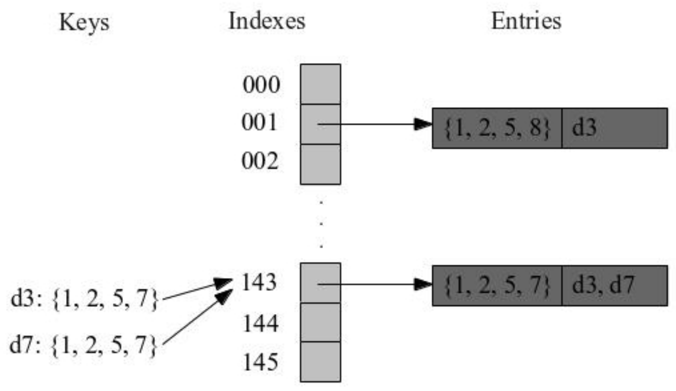


Figure 4.4: Hash table storing subsets of words and their documents

Using this technique, we could calculate not only the large overlaps, but *all* the subsets of words in a segment that are shared by multiple documents. If we kept track of all the collisions in a segment in some way, it would be easy to find the most similar pairs in it, since it would be the ones that can be generated from the documents of the largest subset(s).

All the collisions of a segment can be calculated in $O(n \cdot 2^r)$, since we have n documents and $O(2^r)$ subsets of words to hash for each of them, which in reality could be rather slow - especially if r is large. To solve this, we will in the next chapter discuss ideas for algorithms which does not require all the subset sizes for every segment to be generated and hashed.

4.2 Generating the overall winners

A difficult and important task was to efficiently find the pairs of documents that had the highest overall similarities, using the results from the smaller problems. In Section 2.4 we noted that our data behaves according to an empirical law called *Zipf's law*, and one idea was that this might help us to show that it would not be necessary to process all the segments.

According to Zipf's law, if we sorted the term-document matrix by decreasing frequency of the words, the first segments would experience the most collisions, while the last ones would only give few. Consequently, we thought that it would be enough to consider the early segments, since the later ones would not contribute much to the similarity values. Tests showed that it was indeed true that the overall winners were often the ones that scored highest in the early segments, but this heuristic turned out to be difficult to exploit.

Our implementation of the idea did not generate all the segments at once, but instead started with the one containing the most frequent words. The thought was that we could find the winning pairs and subsets of the segment, and then somehow combine this with the results of the next segment. A problem was that we did not know how many segments we would have to go through before we could be sure that those left would not alter the result.

We might have been able to solve the problem with the segments by creating some kind of bound on how much the similarity could be changed in the remaining segments, but we also had another problem with this approach. There was no guarantee that the overall winners would actually be the winners of a certain segment - it was just very likely. To be sure that the result returned was not an approximate one, we needed to consider more subset sizes than the largest ones. We could not prove how many we would need, and thus we had to generate and hash them all. Then we merged them with the result of the next segment, and the algorithm became very inefficient.

We did not find a good way of making use of our original idea, so instead we started to think in another direction. We thought that it was a bit naive to calculate the exact Overlap similarity of a pair only because they shared one rare word in their prefixes, as in the Top-k Join algorithm, and at the same time we had seen that it could be difficult only to consider the frequent words. Perhaps it would be better to look at all the segments at the same time in some way?

We tested to start off creating all of the segments at once, but only hashing the subsets of the largest possible size, which was r. When we had done this for each segment, we wanted to work on the one that seemed most likely to contain the overall most similar pairs in its collisions for the hashed subsets. We used the idea to pick the segment with the fewest number of pairs that could be created from its documents sharing the subsets of size r.

As a very simple example, assume that we had three segments, and that r=5. For the first segment, we go through each document and generate all subsets of the words it has in this segment which have size 5. We hash these, and get for example the following collisions:

Subset	Documents
{1,2,3,5,7}	$d2$,$d5$,$d6$,$d10$,$d44$
{1,3,4,6,7}	$d2$,$d4$,$d26$

The number of pairs that we could create from k elements is given by equation 4.1, and when we set k to the number of documents in each subset we get 10 and 3 pairs for the collisions respectively, which in total is 13.

$$\frac{k!}{2(k-2)!} \tag{4.1}$$

Say that we now did this for the other two segments as well and achieved 15 pairs for the second and 10 for the third. This would mean that we would pick segment 3. We would then calculate the similarity of all of these pairs exactly and store the best results in a min-heap, in the same way as the naive algorithm. For the same segment we then hashed all the subsets of size $r-1$ for the documents, to find the second most similar pairs in the segment. Then the process started over again, and we searched for the new segment with the fewest number of pairs.

The heuristic we used was based on the intuition that if there were few pairs that had the highest similarity in a segment, these would be likely to beat the scores of many other pairs. Also, we would spend as little time as possible calculating exact Overlap similarities. We could easily create an upper bound on the similarity of the unseen pairs by summing up the current sizes of the subsets being hashed of every segment. For example, assume that we once again have three segments with the following subsets sizes currently being hashed:

Segment	Subset size
1	5
2	4
3	2

A pair which similarity has not been calculated yet could at most have an overlap of 5+4+2=11 words, since we are considering the largest possible subset sizes that we have not already processed. We could stop looking for more pairs when this value was lower than or equal to the smallest similarity in the result heap.

An important thing that we noted was that in order for the algorithm to be efficient, we could never know the exact number of pairs included in the collisions of a segment. This is because when we choose a certain segment, some of its pairs might already have been calculated - although they were not when we created the collisions. It would probably take too much time to go through each segment each time we have calculated new pairs and remove the pairs that should not be counted. We decided to settle with a good approximation rather than adding that much extra work for the algorithm.

To find the segment with fewest number of pairs quickly, we kept a min-heap called the *segment heap* with entries consisting of the id of a segment, its current subset size, the collisions for this size, and the number of pairs that could be created from the documents in the collisions. The entry on the top would be the one with the least pairs. We iteratively removed the top-entry from the heap and calculated the pairs of its collisions exactly. Then we updated the result heap and decreased the subset size of the entry by 1. We then generated all the collisions of the new size, and inserted an entry with the updated values in the segment heap. If the bound of the overall similarity for unseen pairs was now too small, we would stop and return the pairs we had found. Otherwise we would again remove the top-entry from the heap.

We also let the algorithm keep a hash table to store the pairs that had already been calculated, so that we did not calculate the similarity of a pair more than once. When counting the number of pairs for a segment, we would exclude the pairs that were already in this hash table. We also realised that we could perform some *size filtering*; it was unnecessary to hash the subsets of a document which length was smaller than the lowest similarity value in the result heap. Such a document could not possibly occur in a pair with an Overlap similarity large enough to be returned by the algorithm. After each update of the result heap we therefore remove the documents that were too short.

The pseudo-code for the algorithm is presented in Algorithm 2, and we call it the *Segment Bounding Algorithm.* Lines 1 to 8 are initialization, where we for example create segments for the term-document matrix, and add the initial entries for the segment heap. Then we loop while the upper bound of the similarity of the unseen pairs, here referred to as $total\_max$, is smaller or equal to the worst similarity in the result heap.

We remove the top-entry from the segment heap in line 10, and then we iterate over the document lists for its collisions and create pairs from these. We calculate the exact similarity of a pair in line 16 if we have not already done so. Then we check if we should update the result heap. In line 24 we filter out the documents that are now to short to be able to be included in a winning pair. We then decrease the subset size for the segment and in line 28 we calculate the collisions of this new size. A new entry is created with the same id but with the updated subset size, the new collisions, and the number of pairs that can be created from these. We insert it in the segment heap in line 30.

Algorithm 2 Segment Bounding Algorithm

Input: *tdm* - term document matrix
Input: k - the number of pairs with highest similarity value to return
Input: r - largest size of subsets to use for segmentation of *tdm*
Output: k most similar pairs

```
 1: $R \leftarrow$ empty heap for storing top pairs
 2: $S \leftarrow$ empty segment heap
 3: $H \leftarrow$ empty hash table for computed pairs
 4: $tdm\_segmented \leftarrow$ segment $tdm$ according to $r$
 5: $total\_max \leftarrow r \cdot$ number of segments
 6: $S \leftarrow$ initialize_segment_heap($S$, $r$, $tdm\_segmented$)
 7: $s_k \leftarrow 0$
 8: $filtered\_docs \leftarrow$ empty list of size filtered documents
 9: while $s_k < total\_max$ and length($S$) $> 0$ do
10:     ($n$, $i$, $subset\_size$, $collisions$) $\leftarrow S$.pop()
11:     for $docs$ in $collisions$ do
12:         for ($d1$,$d2$) in pairs($docs$) do
13:             $d1$,$d2 \leftarrow$ min($d1$,$d2$), max($d1$,$d2$)
14:             if (($d1$,$d2$)) not in $H$ then
15:                 $H$.add(($d1$,$d2$))
16:                 $score \leftarrow$ overlap($d1, d2$)
17:                 if $score > s_k$ then
18:                     $R$.push(($score$, ($d1$,$d2$))
19:                     $s_k$ = similarity of $R$.top()
20:                 end if
21:             end if
22:         end for
23:     end for
24:     $filtered\_docs \leftarrow$ documents with length shorter than or equal to $s_k$
25:     $segment \leftarrow tdm\_segmented[i]$
26:     $total\_max \leftarrow total\_max - 1$
```

```
        if ($subset\_size$ - 1) $\neq$ 0 then
            $n$, $collisions$ ← Segment_Collisions($segment$, $subset\_size$-1,
                                                  $filtered\_docs$, $H$)
            $S$.push(($n$, $i$, $subset\_size$-1, $collisions$))
        end if
    end while
    return $R$
```

The function that produces the collisions for a segment is *Segment_Collisions*, and is defined in Algorithm 3. It iterates over all of the documents from line 5 to 19, and if they have not been filtered and if they do not have fewer words in the segment than the subset size we are looking for, it generates all its subsets of the correct size (line 8). Each of the subsets is hashed in lines 9 to 16, and if we get a collision, we save this in a the set called C.

When we have generated and hashed all subsets of size *subset_size* we calculate the number of pairs we can create from the ones that were involved in collisions. We do not create a pair that we have already calculated exact similarity for, and lines 24-26 handle this by checking whether it occurs in the hash table of calculated pairs, H. We save the document lists for each collisions in *collision_docs* and in line 31 we return this and the number of pairs for the segment.

Algorithm 3 Segment_Collisions

Input: *segment* - segment to scan for collisions
Input: *subset_size* - subsets in collisions should be of this size
Input: *filtered_docs* - removed documents
Input: H - hash table of calculated pairs
Output: n - number of pairs for next collisions
Output: *collisions* - a list of lists of documents that collides with subsets of size equal to *subset_size*

```
1:  $D \leftarrow$ empty hash table for subsets
2:  $C \leftarrow$ empty set of subsets that have collisions
3:  $n \leftarrow 0$
4:  $collision\_docs \leftarrow []$
5:  for doc in segment do
6:      if doc not in filtered_docs then
7:          if length of $doc \geq subset\_size$ then
8:              $subsets \leftarrow$ all subsets of size subset_size for doc in segment
9:              for subset in subsets do
10:                 if subset not in $D$ then
11:                     $D[subset] \leftarrow$ [doc id for doc]
12:                 else
13:                     $D[subset]$.add(doc id for doc)
14:                     $C$.add(subset)
15:                 end if
16:             end for
17:         end if
18:     end if
19: end for
20: $pairs \leftarrow$ empty set
21: for subset in $C$ do
22:     $docs \leftarrow D[subset]$
23:     for pair in docs do
```

```
24:         if $pair$ not in $H$ then
25:             $pairs$.add($pair$)
26:         end if
27:     end for
28:     $collisions\_docs$.append($docs$)
29: end for
30: $n \leftarrow$ length of $pairs$
31: return $n$, $collision\_docs$
```

We also have a function for initialising the segment heap, which is given by the code in Algorithm 4. The subset size of every segment is initially given by r, so for each segment we find all collisions of this size. Then we push the id of the segment, its collisions, the current subset size, and number of pairs in the collisions on the heap.

Algorithm 4 initialize_segment_heap

Input: S - segment heap
Input: r - largest size of subsets initially
Input: *tdm_segmented* - segmented tdm
Output: S initialized

```
1: filtered_docs $\leftarrow$ empty list
2: H $\leftarrow$ empty hash table
3: for $segment$ in $tdm\_segmented$ do
4:     $n$, $collisions$ = $Segment\_Collisions$($segment$, $r$, [],H)
5:     $S$.push($n$, id of $segment$, $r$, $collisions$)
6: end for
7: return $S$
```

Chapter 5

Results

We have found three algorithms during this project that are applicable to our problem: TOP-MATA, Top-k Join, and SBA. We will now evaluate these, and compare them to each other. Before we look at the running times of the algorithms we will make sure that we use the version of each which has the best running time. For both Top-k Join and SBA this means that we have to test optimisations to see whether they really improve the time, and we also need to test which segment sizes that works best for SBA.

5.1 Data sets and experimental setup

Two data sets will be used to evaluate our algorithms. The first one is the Opinosis data set [12], which was also the set used during the evaluation of the approach to summarization that we have been trying to speedup. It consists of extracts from user reviews of different topics, of which there are 51 in total, and there are between 50 and 575 documents (reviews) for each. The extracts are rather short, since they all consist only of one sentence.

We will also use the TREC9 - Filtering Track collections [22] (TREC), which was used during the evaluation of the Top-k Join algorithm [23]. This set consists of abstracts from medical journals, and is extracted from MEDLINE: a medical information database. Each reference originally contains more information than just the abstract, but this will be filtered out. There are no topics in this data set and the documents are significantly longer in comparison with the ones in the Opinosis data set.

The data sets will be used in somewhat different ways. In the case of

Opinosis, we will create one term-document matrix per topic - so when we run an algorithm on this data set it will be searching for similar pairs of documents which all discuss the same product or service. In most tests we will apply our algorithms to all of the topics, one at a time. For the TREC data set we will consider all of the documents to make up a single topic. We do not have the time though to perform all our tests on the whole set (348 566 references), so instead we create nine different smaller sets of varying sizes containing randomly chosen documents. For each test involving the TREC set we will use the same sets, and their sizes are 50, 100, 200, 400, 800, 1600, 3200, 6400, or 12800 documents.

To be able to evaluate how well the algorithms perform when the number of documents increases we can run them for each of the sets we created for TREC. If we want to do the same test for the Opinosis data, we can run the algorithms on each of the 51 topics, which vary in number of documents. The number of pairs to be returned, defined by k will be selected to 1, 5, and 100 in our experiments since lower values will probably be more relevant for our application area, but it can also be interesting to see how the results scale for a larger k-value. We will not always include the results from all three values of k, but then the rest will be available in Appendix A.

When we measure the running times of the algorithms, we will include the time for preprocessing; this involves scanning all the documents and creating a term-document matrix, and also different kinds of sorting of the matrix. These tasks were not illustrated in the pseudocodes, where the term-document matrix was simply handed over as input to the algorithms ready to be used. When we use the Top-k Join algorithm, the words in the matrix will be sorted by inverse frequency, which means that the indexes in the index vectors will occur in this order. For SBA, we will find out in Section 5.2.2 whether or not to sort the matrix according to frequency. For TOP-MATA the words will not be sorted at all, but the documents will be sorted by length - the longest ones will appear first in the matrix.

We did not have access to the source code of the TOP-MATA or Top-k Join, so we implemented them ourselves using the pseudocodes and adding our own modification as discussed in Chapter 3. All the implementations were done in Python 2.6.6, and the tests were all performed on a PC with an Intel Core i5-3470S 2.90GHz CPU and 16GB RAM. The operating system used was 64-bit Red Hat Enterprise Linux.

5.2 Optimizing the algorithms

In this section we will search for the best setups of the algorithms before we evaluate and compare them. We will investigate the effects of different optimisation ideas to see if they really can improve the running time, and we will also decide the optimal values of some parameters for the algorithms.

5.2.1 Top-k join

In Section 3.2 we argued that for the Top-k Join algorithm, some of the optimisations described in [23] were unnecessary for the case of Overlap similarity. There were also two other methods that we did not know how well they would work - suffix filtering and the idea of checking whether a pair really needs to be put into the hash map of calculated pairs before doing so. In this section we will experimentally test these ideas to decide if we should included them or not.

Suffix filtering

We perform the suffix filtering using different depths for the recursion, and Table 5.1 contains the running times for the Opinosis data set. Each value is the total running time of all the topics added together, in order to make the results compact and readable. The first value is the time for the case when we do not add any filtering at all. We skip depth 5-8, since the running time for depth 4 and up is approximately the same. Figure 5.1 displays the results graphically for each individual topic. We also test suffix filtering on the TREC data set, using 1600 documents. The running times can be seen in Table 5.2.

Depth	Time (s)
Top-k	0.797
1	1.020
2	1.142
3	1.297
4	1.370
9	1.373

Table 5.1: Top-k Join with suffix filtering for Opinosis, k=5

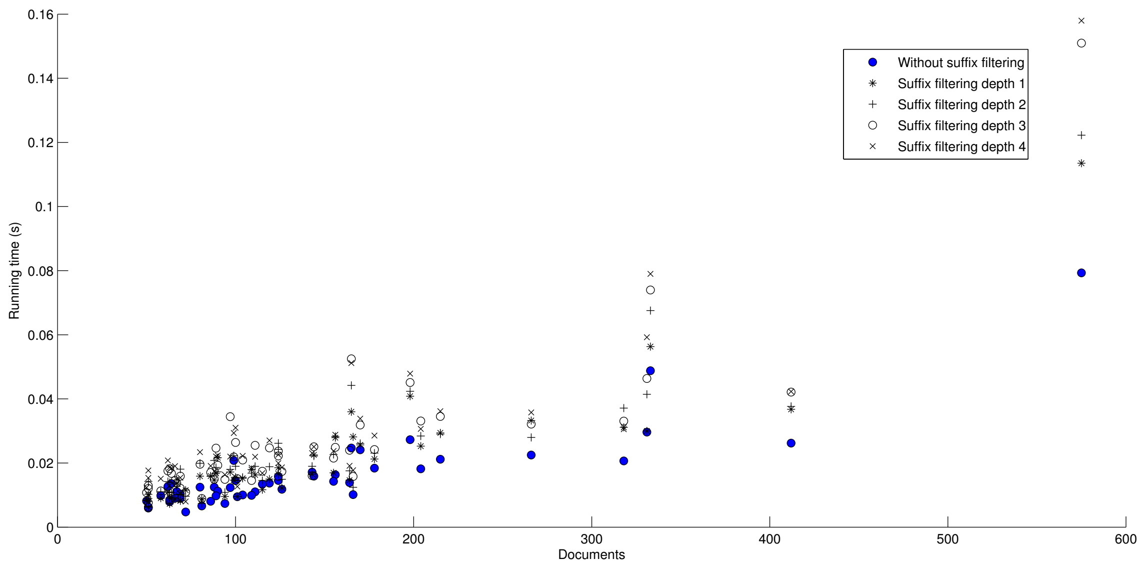


Figure 5.1: Top-k Join with suffix filtering for Opinosis, k=5

Depth	Time (s)
Top-k	17.452
1	20.309
2	24.330
3	29.247
4	36.053
9	34.673

Table 5.2: Suffix filtering for TREC, $k = 5$

As we can see, the running time always gets worse when we add suffix filtering, no matter the depth. When the depth is increased, the time seems to become worse and worse until it finally remains approximately the same. The number of documents that are filtered should increase along with the depth, but after a certain limit it becomes impossible to continue splitting up the suffixes. When we reach this point, the filtering will have no more effect and the running time will stay the same.

We think that the reason that the time initially gets worse as the depth increases is that the filtering becomes more time consuming while we do not filter enough pairs. We will probably lose more time on checking whether pairs should be filtered or not compared, to the time we win by not having to check the exact similarity for some of them. It is mentioned in [23] that suffix filtering might not be very effective on the TREC data set since there are not enough pairs that stands out with their high similarity values, and therefore it can be difficult to filter out documents fast. We think that the same reasoning can be applied to the Opinosis data, since the documents in a topic all discuss the same product or review and are likely to use the same words. There are probably few pairs with much higher similarities than the others. We will therefore not use suffix filtering for the Overlap measure.

Saving fewer pairs in the hash table

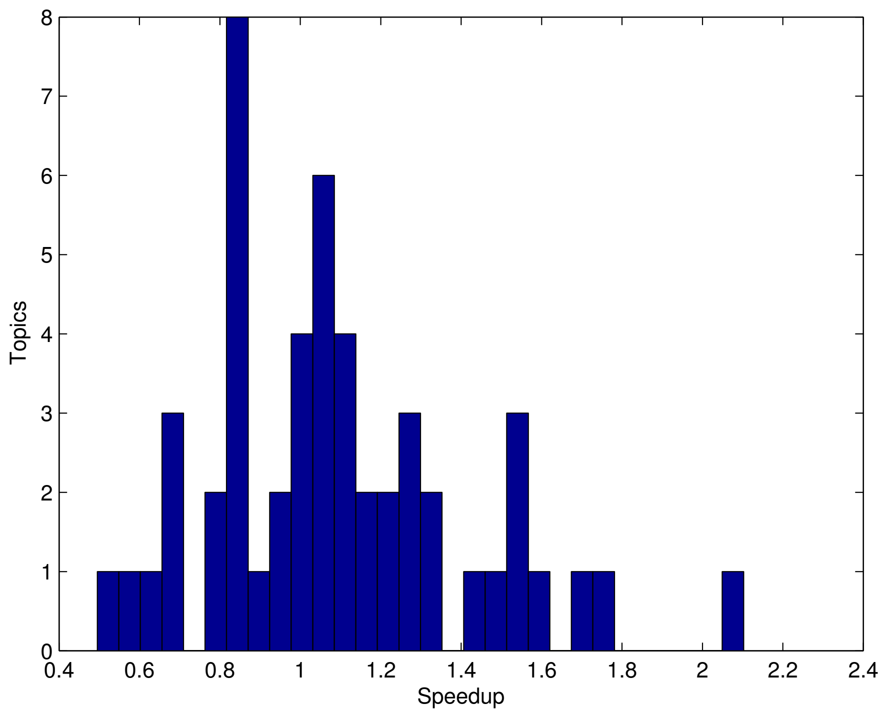


Figure 5.2: Top-k Join speedup when saving all pairs in hash table for Opinosis, $k = 5$

Top-k Join keeps all the seen pairs in a hash table to prevent repeated calculations, but to save space one could avoid adding the pairs that would never be looked up. We want to find out how this optimisation affects the running time, since we are interested in the most time efficient version of the algorithms. It turns out that it is better to store all the calculated pairs in the hash table, and Figure 5.2 displays the speedup. We will lose space, but win time on not having to find out whether we should store the pairs or not. Therefore we will continue to save all pairs when we later compare running times.

5.2.2 Segment Bounding Algorithm

For the Segment Bounding Algorithm we will consider the method of size filtering, since we are not sure whether or not this will improve the running time. We will also find optimal values for the segment size on both data sets, and test how sorting the term-document matrix before executing the algorithm affects the running time.

Size filtering

When we add size filtering to the SBA we barely notice any difference. Table 5.3 contains the running times for the topics of Opinosis added together, with and without the filtering. Table 5.4 displays the results for the same test on the TREC data set. The reason for why the times are so similar is probably that when the algorithm searches for collisions it already, without the size filtering, skips the documents that have too few words in the segment. Many of the short documents that are removed due to the filtering would probably not be considered anyway. Since the times seem to be a little bit better when using size filtering, we will add this to the algorithm during upcoming tests though.

k	With SF	Without SF
1	1.1912	1.2009
5	1.3295	1.3406
100	1.9892	1.9637

Table 5.3: SBA with and without size filtering for Opinosis

Nr of docs	k	With SF	Without SF
1600	1	3.612	3.740
1600	5	10.467	10.545
1600	100	15.087	15.328
800	1	1.030	1.136
800	5	3.228	3.198
800	100	5.417	5.607
100	1	0.131	0.136
100	5	0.144	0.146
100	100	0.200	0.207

Table 5.4: SBA with and without size filtering for TREC

Segment size

The SBA divides the term-document matrix into segments, and an important question is how large these should be for a good performance. If they are too large, the collision detection within the segments will be too time consuming, but if they are too small the highest scoring pairs in each might less likely be the same as the overall highest scoring pairs, and there would be more collisions since the average subset size is smaller, i.e. more pairs to fully evaluate.

We start by testing on the Opinosis data, and run the algorithm on each topic using different segment sizes and different k-values. We add together all the times for the different topics, and obtain the running times in Figure 5.3. It is obvious that the best size lies somewhere around 8 to 10; it depends on the value of k, but the neighbouring values give decent times as well.

Then we do the test for the TREC data to see if there is any difference. Table 5.5 shows the optimal sizes for each of the TREC sets, and we can observe that as the number of documents increases, a larger size will give better running times. This is probably since we will get more pairs with high scores in the segments, which requires more pairs to be calculated; by increasing the size we make sure that the real winners stand out more. We could have tested if the running times for the larger topics in the Opinosis data also would have improved if we increased the subset size, but since the variety in the number of documents is not that large, it would probably not have made any significant difference. It would also be too much work having to change the size before testing every topic since there are so many.

When the segment size is to be decided for a new type of data, one has to experiment a bit, but we can at least notice that the optimal size depends on how many documents there are. In our future tests on the TREC data set we will make sure to use the segment sizes shown to be optimal in our tests when using different numbers of documents. For the Opinosis data we will always use a size of 9 words.

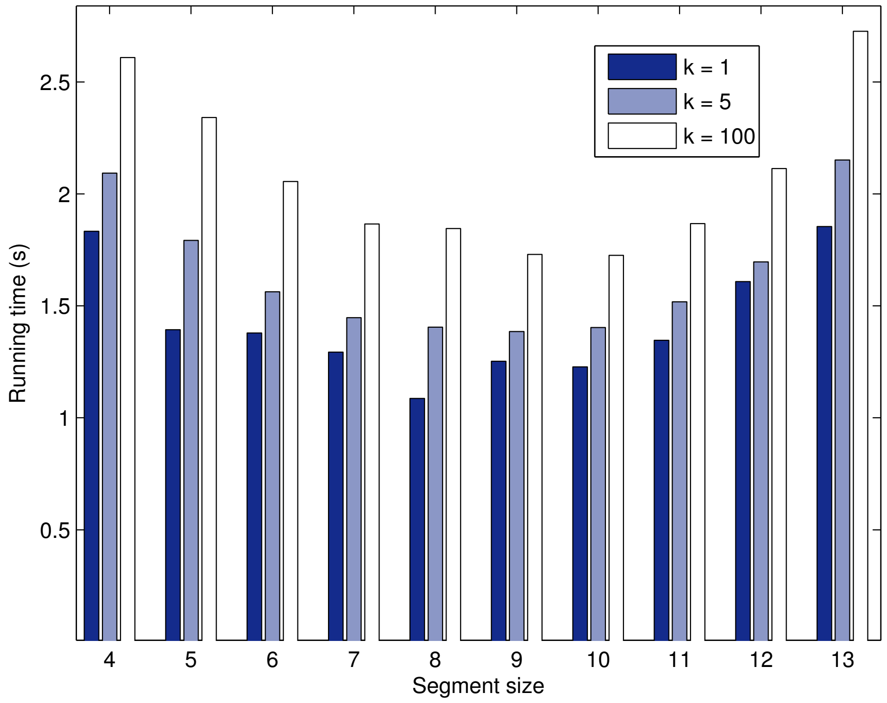


Figure 5.3: SBA with varying segment size for Opinosis

TREC documents	Optimal size $k = 1$	$k = 5$	$k = 100$
50	9	8	6
100	8	9	8
200	11	11	11
400	12	12	12
800	12	13	14
1600	13	13	15
3200	13	15	15
6400	15	16	15
12800	14	15	16

Table 5.5: Optimal segment sizes for SBA

Sorting the term-document matrix

Since Zipf's law applies to our data, it is probably meaningful to sort the term-document matrix by frequency of the words. In Table 5.6 we can see the comparison between sorting and not sorting the matrix by descending frequency for some topics in the Opinosis set, and Table 5.7 contains the results for the same test on some TREC sets.

Documents in topic	Time sorted (s)	Time unsorted (s)
50	0.018	0.018
72	0.017	0.018
126	0.027	0.028
170	0.035	0.041
215	0.039	0.040
266	0.036	0.062
333	0.064	0.083
575	0.138	0.261

Table 5.6: SBA sorted vs unsorted tdm for Opinosis, k=5

TREC documents	Time sorted (s)	Time unsorted (s)
100	0.144	0.171
800	3.226	4.683
1600	10.467	16.581

Table 5.7: SBA sorted vs unsorted term-document matrix for TREC, k=5

When we have few documents the two versions give similar running times, but as the number increases we can see that it becomes more and more beneficial to sort the term-document matrix. Figure 5.4 contains a histogram displaying the speedup we get by sorting the matrix.

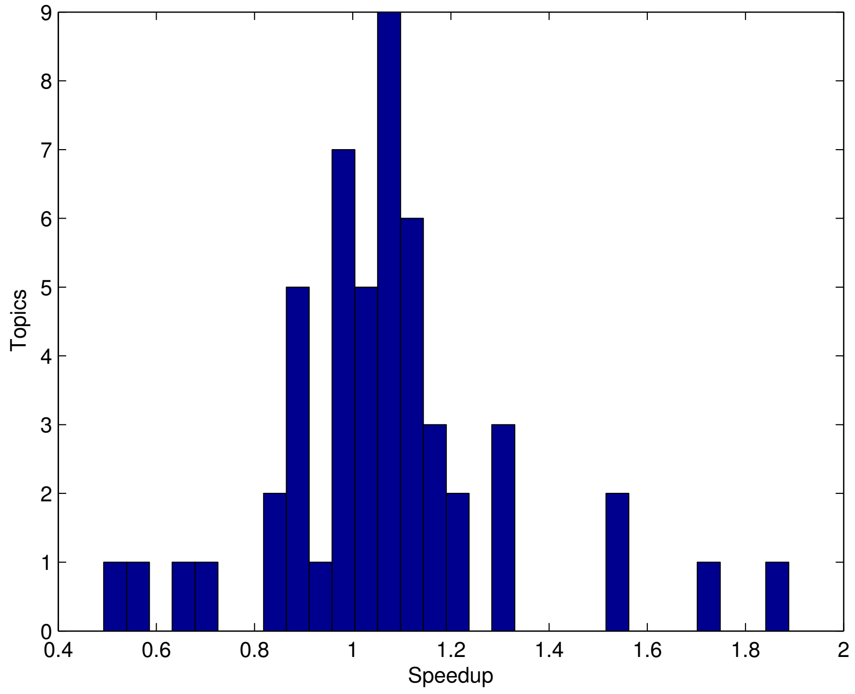


Figure 5.4: Speedup using sorted term-document matrix instead of unsorted for Opinosis, $k = 5$

The reason for this behaviour could be that the algorithm is forced to choose segments with more pairs in average for the largest subsets when the matrix is unsorted. In a sorted matrix, the *early* segments will have much more pairs in total in their collisions compared to the later ones as the number of documents increases, according to Zipf's law. In an unsorted matrix on the other hand, all the segments include both rare and frequent words and they will probably contain about the same number of pairs in their collisions. This could imply that the algorithm has to calculate more pairs in total for the unsorted version, since its choices are more limited. From now on we will always sort the term-document matrix before giving it as input to SBA.

5.3 Comparing the algorithms

We will now use the algorithms with the best settings from the previous section (no particular settings for TOP-MATA) and compare their running times to each other and also to the naive algorithm. We will in addition analyze the behaviour of the algorithms by for example looking at how many pairs they calculate similarity for during execution.

5.3.1 Running times

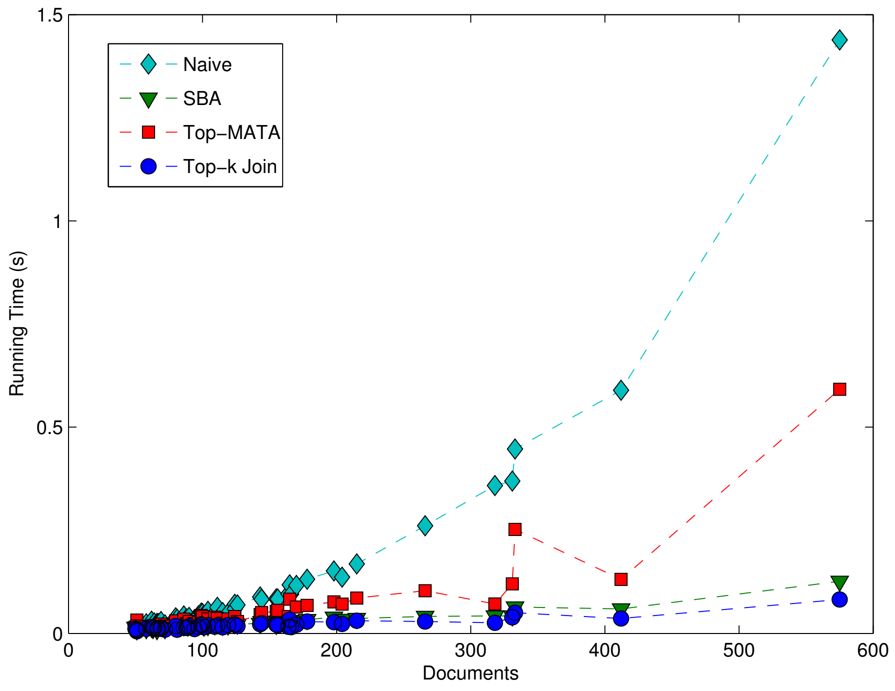


Figure 5.5: Running times for topics in Opinosis, $k = 5$

We execute each of the algorithms on the topics of the Opinosis data, and as we can see in Figure 5.5, the Top-k Join algorithm and the SBA behaves very similarly. Top-k Join is slightly faster than SBA, but both achieve very short running times. TOP-MATA is better than the naive approach, but it performs significantly worse compared to Top-k Join and SBA as the number of documents increases.

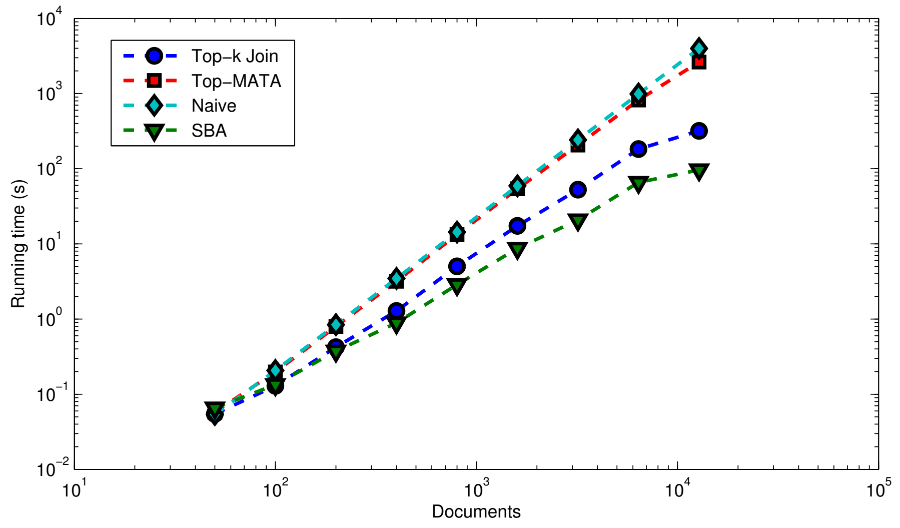


Figure 5.6: Running times for TREC sets, $k = 5$

In Figure 5.6 we can see the running times of the algorithms when executed on the TREC sets of different sizes. TOP-MATA is still the slowest of the three, but now SBA outperforms Top-k Join. The difference gets more and more significant as the number of documents increases.

The reason that our implementation of TOP-MATA is so much slower compared to SBA and Top-k Join is that it is too naive. It uses the heuristic that longer documents give higher similarities, but does not take anything else into consideration. The problem is that the pairs containing the longest documents are not always the ones with largest similarity values. Also, within our data, most documents have similar lengths, which means that extremely much work might be needed to be performed until the algorithm can terminate with a result that is guaranteed to be optimal. It cannot terminate until the worst of the best pairs has a similarity value larger than the upper bounds of all remaining pairs, which is the length of the shorter document.

5.3.2 Pairs calculated

To compare the behaviour of the algorithms we will also count the number of pairs that each of them will calculate exact similarity for during execution (also referred to as *candidate pairs*). The results are shown in Table 5.8 - 5.13 - the first three are for the Opinosis data and the last three are for the

TREC data. The values in the tables are the percentage of the total number of possible pairs that the algorithms calculate similarity for. For the Opinosis data we show the mean value of the percentage for all its topic, along with the variance, and the lowest and highest percentage encountered.

k	Max	Min	Mean	Variance
1	2.69 %	0.0043 %	0.72%	0.0036%
5	5.01%	0.312%	1.755%	0.011%
100	22.27%	1.96%	9.1%	0.225 %

Table 5.8: Pairs considered by SBA in Opinosis

k	Max	Min	Mean	Variance
1	7.93%	0.0087%	2.43%	0.0341%
5	10.68%	0.59%	4.43%	0.07%
100	30.82%	2.75%	14.93%	0.569 %

Table 5.9: Pairs considered by Top-k Join in Opinosis

k	Max	Min	Mean	Variance
1	52.24%	0.0696 %	20.63%	1.367%
5	67.53%	6.92%	31.71%	1.51%
100	88.47%	24.94%	64.02%	2.18%

Table 5.10: Pairs considered by TOP-MATA in Opinosis

k	Pairs
1	3.68%
5	15.87%
100	24.53%

Table 5.11: Pairs considered by SBA in TREC 1600

k	Pairs
1	8.17%
5	21.63%
100	30.73%

Table 5.12: Pairs considered by Top-k Join in TREC 1600

k	Pairs
1	58.42%
5	79.87%
100	86.95%

Table 5.13: Pairs considered by TOP-MATA in TREC 1600

As k increases, the percentage of the total pairs that we need to consider increases for all algorithms, since we need to search for more top-similarity pairs. The values of the percentage are very different between the algorithms though. TOP-MATA calculates the most pairs, but notable is that for $k = 1$, the percentage is very small for some topics. This can happen when the winning pair consists of two documents that are longer than most other ones and has a high similarity.

If we compare the SBA and Top-k Join, the percentage is always lower for SBA. Yet, Top-k Join is faster for the Opinosis data. This could be because SBA spends more time trying to generate good candidate pairs compared to Top-k Join, which results in less similarity calculations. But since the documents are very short, these calculations can be performed very quickly. The overhead times that arise when SBA generates and hashes subsets become larger than the time it takes for Top-k Join to calculate the larger amount of pairs. This was also an assumption made by [23], when the Top-k Join was compared to another algorithm on data containing short records. The TREC data on the other hand consists of documents which are much longer, and this makes the similarity calculations very time consuming. It is then more efficient to spend more time finding good candidate pairs compared to calculating a larger percentage of the total pairs.

5.3.3 Pairs returned

The number of top-scoring pairs that we want to be returned to be able to produce a summary will probably be quite small, and therefore we have only considered the values 1, 5, and 100 for k so far. It could also be interesting to look at some larger values though, since the algorithms could be used for other purposes than summarization. In Figure 5.7 we can see how the running times of the algorithms changes for different k-values for the Opinosis set. The time for a certain value is here the total time for the topics added together.

Top-k Join is always slightly faster than SBA, but the differences between them seem to stay the same for all k's. Both of the algorithms will do an equal amount of extra work for finding more high-scoring pairs. TOP-MATA is worse than the other two for all k-values, and it will already for very low k-values need to search a lot of pairs.

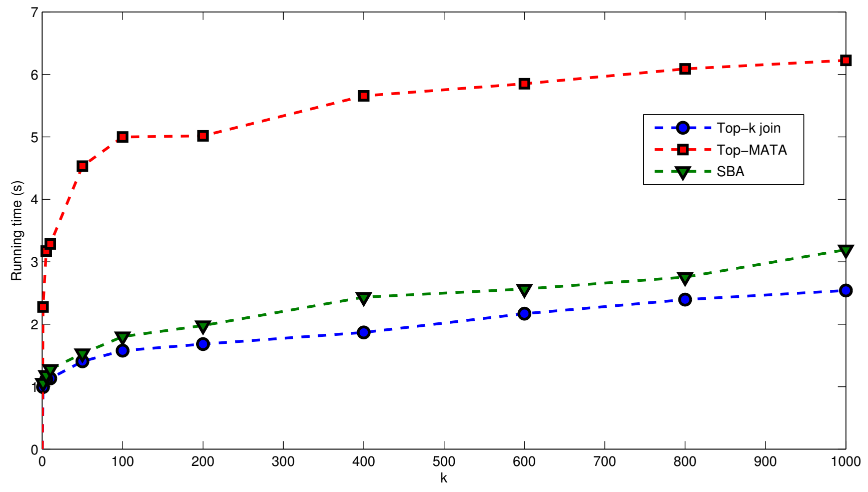


Figure 5.7: Running times for different values of k for Opinosis

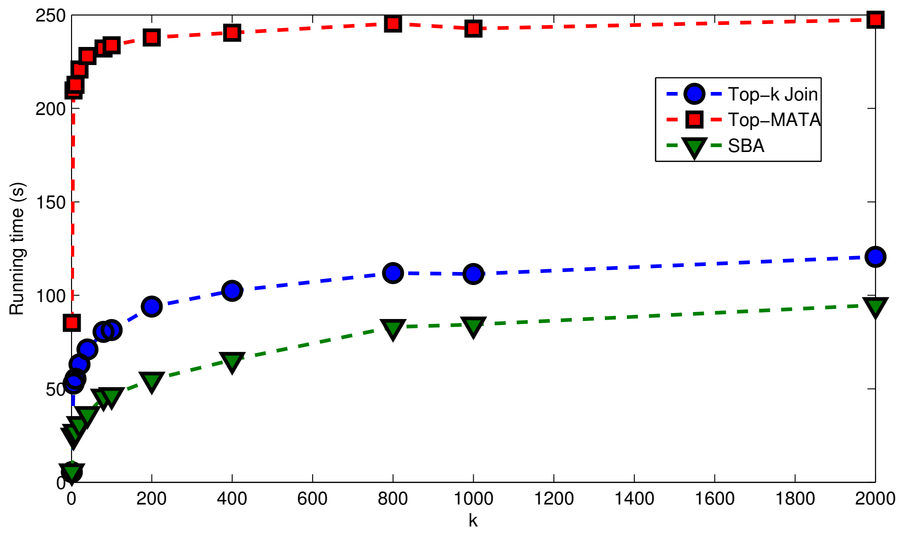


Figure 5.8: Running times for different values of k for TREC 3200

Figure 5.8 shows the running time on the TREC data set using different values on k. SBA outperforms Top-k Join for all values but also here their running times in general seem to increase equally fast.

Also for TREC, TOP-MATA performs worst of the algorithms, and we can see that its running time increases extremely fast early. It has a stop condition that is too weak, and often requires a lot of pairs to be calculated. When k is very small we can be lucky and find some pairs of long documents with very high similarity values and be able to finish fast, but it does not seem as though there are many such pairs in the TREC data set used for testing - considering the behaviour of the running time for TOP-MATA.

Comparison of SBA and Top-k Join

The algorithms that have shown the best results in our tests are SBA and Top-k Join, and we now show their running times more in detail. Figure 5.9 displays the running times of the algorithms for all the different topics of Opinosis. As we have already discussed, Top-k Join is faster for these short documents.

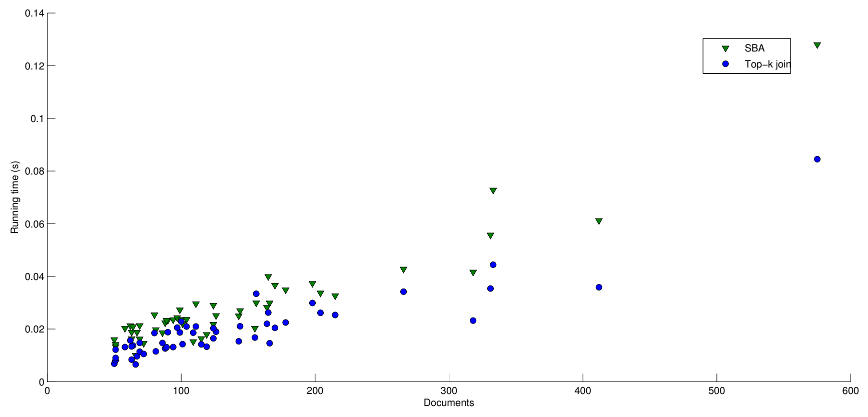


Figure 5.9: Running times for Opinosis, $k = 5$

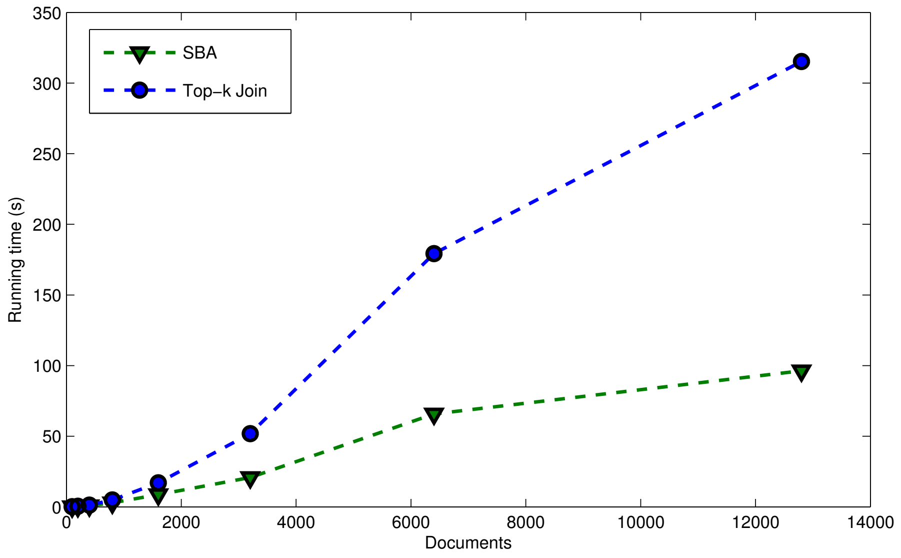


Figure 5.10: Running time for TREC, $k = 5$

TREC Documents	SBA	Top-k Join	Naive
100	0.1339 s	0.1368 s	0.2074 s
200	0.3747 s	0.4184 s	0.8404 s
400	0.8887 s	1.264 s	3.474 s
800	2.872 s	4.928 s	14.38 s
1600	8.724 s	17.08 s	59.20 s
3200	20.81 s	51.85 s	241.6 s
6400	65.80 s	179.3 s	986.4 s
12800	96.36 s	315.2 s	3989 s

Table 5.14: Running times for TREC sets, k=5

In Figure 5.10 we can see the running times for the TREC data of different sizes, and in Table 5.14 we can compare values to the naive algorithm. When the documents are very few, the performances of Top-k Join and SBA are similar. As the number of documents increases, the strength of SBA becomes apparent. For TREC with 12800 documents, SBA is more than 3.5 minutes faster than Top-k Join.

Chapter 6

Conclusion

In this project we implemented and tested existing algorithms for finding the top-k most overlapping pairs of documents. We also proposed a new algorithm, the Segment Bounding Algorithm, which in some cases performs better than our implementation of the Top-k Join - the fastest previously known approach that we found.

We have seen that SBA is faster than Top-k Join for larger median document sizes. As the number of documents increases, SBA performs better and better compared to Top-k Join on these documents. It also performs well for shorter document size, even though our version of the Top-k Join algorithm is slightly more efficient.

For the problem domain that this project was directed towards, the documents can be very many and both short and long. If one knows that all the documents will be very short, we would recommend the Top-k Join algorithm, but otherwise it is probably a better idea to go with SBA. Even if the documents should be very short, this algorithm would still perform well, and it would execute in significantly shorter time compared to the Top-k Join algorithm if we get longer documents and as we get more of them. The running time also scales well as the number of pairs to be returned increases.

SBA performs significantly better than the original naive algorithm, making the task of document-summarization much more feasible than earlier. It can also be used for other purposes - it is worth considering when one wants to find top-k overlapping records of data efficiently, if the records are not too small. The data should approximately follow Zipf's law in order for the algorithm to be fast.

The problem with SBA could be to find the optimal segment size. One could experiment until it is found and then continue to use the same value on the same type of data. If the number of documents increases or decreases, it could be modified a bit, but it would probably not be absolutely necessary, since there is a pretty large interval of sizes that work well.

6.1 Future work

A question remaining is if it is possible to select the segment size used in SBA ahead of time instead of experimenting to find the best one. While good results still can be achieved with suboptimal segment sizes, poorly chosen segment sizes can adversely affect the performance. Finding a good segment size automatically would be desirable.

Another task for the future is to try to find a faster way of generating collisions of a certain size in the segments of SBA. This is where most time is consumed, and it is because of this that the algorithm is a bit slower than Top-k Join for short documents.

It may also be possible to further improve SBA by exploiting the segmentation of the term-document matrix. Instead of immediately calculating the exact overlap of the pairs colliding for the largest subsets, it could perhaps be beneficial to create pairs of segments and by looking at intersections of their top-documents, create a stricter upper bound on unseen pairs. This upper bound could be limited by the sum of upper bounds of the pairs of segments instead of the upper bound of each segment. This could potentially reduce the number of full overlap calculations needed.

References

[1] R. J. Bayardo, Y. Ma, and R. Srikant. “Scaling Up All Pairs Similarity Search”. In: *Proceedings of the 16th International Conference on World Wide Web*. WWW ’07. Banff, Alberta, Canada: ACM, 2007, pp. 131–140.

[2] S. Chaudhuri, V. Ganti, and R. Kaushik. “A Primitive Operator for Similarity Joins in Data Cleaning”. In: *Data Engineering, 2006. ICDE ’06. Proceedings of the 22nd International Conference on*. Apr. 2006, pp. 5–5.

[3] S. Chen, B. Ma, and K. Zhang. “On the similarity metric and the distance metric”. In: *Theoretical Computer Science* 410.24-25 (2009). Formal Languages and Applications: A Collection of Papers in Honor of Sheng Yu, pp. 2365–2376.

[4] M. Cristelli, M. Batty, and L. Pietronero. “There is more than a power law in Zipf”. In: *Scientific reports* 2 (2012).

[5] P. Damaschke. Personal meeting. Jan. 12, 2014.

[6] D. Das and A. F. Martins. “A survey on automatic text summarization”. In: *Literature Survey for the Language and Statistics II course at CMU* 4 (2007), pp. 192–195.

[7] A. Dasgupta, R. Kumar, and T. Sarlos. “Fast locality-sensitive hashing”. In: *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM. 2011, pp. 1073–1081.

[8] M. Datar et al. “Locality-sensitive hashing scheme based on p-stable distributions”. In: *Proceedings of the twentieth annual symposium on Computational geometry*. ACM. 2004, pp. 253–262.

[9] J. Dean and S. Ghemawat. “MapReduce: simplified data processing on large clusters”. In: *Communications of the ACM* 51.1 (2008), pp. 107–113.

[10] T. Elsayed, J. Lin, and D. Oard. “Pairwise Document Similarity in Large Collections with MapReduce”. In: *Proceedings of ACL-08: HLT, Short Papers*. Columbus, Ohio: Association for Computational Linguistics, June 2008, pp. 265–268.

[11] I. Foster. *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*. ADDISON WESLEY, 1995.

[12] K. Ganesan, C. Zhai, and J. Han. “Opinosis: a graph-based approach to abstractive summarization of highly redundant opinions”. In: *Proceedings of the 23rd International Conference on Computational Linguistics*. Association for Computational Linguistics. 2010, pp. 340–348.

[13] V. Gupta and G. S. Lehal. “A Survey of Text Summarization Extractive Techniques”. English. In: *Journal of Emerging Technologies in Web Intelligence* 2.3 (2010), p. 258.

[14] U. Hahn and I. Mani. “The challenges of automatic summarization”. In: *Computer* 33.11 (Nov. 2000), pp. 29–36.

[15] Z. He, X. Xu, and S. Deng. “Mining top-k strongly correlated item pairs without minimum correlation threshold”. In: *International Journal of Knowledge-based and Intelligent Engineering Systems* 10.2 (2006), pp. 105–112.

[16] H. Koga, T. Ishibashi, and T. Watanabe. “Fast agglomerative hierarchical clustering algorithm using Locality-Sensitive Hashing”. In: *Knowledge and Information Systems* 12.1 (2007), pp. 25–53.

[17] W. Li. “Random texts exhibit Zipf’s-law-like word frequency distribution”. In: *Information Theory, IEEE Transactions on* 38.6 (1992), pp. 1842–1845.

[18] C.-Y. Lin. “Rouge: A package for automatic evaluation of summaries”. In: *Text Summarization Branches Out: Proceedings of the ACL-04 Workshop*. 2004, pp. 74–81.

[19] J. Lin. “Brute Force and Indexed Approaches to Pairwise Document Similarity Comparisons with MapReduce”. In: *Proceedings of the 32Nd International ACM SIGIR Conference on Research and Development in Information Retrieval*. SIGIR ’09. Boston, MA, USA: ACM, 2009, pp. 155–162.

[20] A. Rajaraman and J. D. Ullman. *Mining of Massive Datasets*. New York, NY, USA: Cambridge University Press, 2011.

[21] M. Slaney and M. Casey. “Locality-sensitive hashing for finding nearest neighbors [lecture notes]”. In: *Signal Processing Magazine, IEEE* 25.2 (2008), pp. 128–131.

[22] *TREC-9 Filtering Track Collections*. Accessed: 2014-03-18. URL: `http://trec.nist.gov/data/t9_filtering.html`.

[23] C. Xiao et al. “Top-k Set Similarity Joins”. In: *Data Engineering, 2009. ICDE ’09. IEEE 25th International Conference on*. Mar. 2009, pp. 916–927.

[24] C. Xiao et al. “Efficient Similarity Joins for Near-duplicate Detection”. In: *ACM Trans. Database Syst.* 36.3 (Aug. 2011), 15:1–15:41.

[25] H. Xiong, M. Brodie, and S. Ma. “Top-cop: Mining top-k strongly correlated pairs in large databases”. In: *Data Mining, 2006. ICDM’06. Sixth International Conference on*. IEEE. 2006, pp. 1162–1166.

[26] R. B. Zadeh and A. Goel. “Dimension independent similarity computation”. In: *The Journal of Machine Learning Research* 14.1 (2013), pp. 1605–1626.

[27] S. Zhu et al. “Scaling up top-K cosine similarity search”. In: *Data & Knowledge Engineering* 70.1 (2011), pp. 60–83.

[28] G. K. Zipf. “Human behavior and the principle of least effort.” In: (1949).

Appendix A

Extensive Test Results

We here give the results for the tests performed in Chapter 5, but now for k-values 1 and 100.

A.1 Suffix filtering

Suffix filtering with varying depth for Top-k Join using the Opinosis data.

Depth	Time (s)
Top-k	0.680
1	0.945
2	0.901
3	0.989
4	0.915
7	1.190
8	0.870
9	0.924

Table A.1: Top-k Join with suffix filtering for Opinosis, k=1

Depth	Time (s)
Top-k	1.360
1	1.962
2	2.390
3	2.964
4	3.188
7	3.270
8	3.328
9	3.377
10	3.248

Table A.2: Top-k Join with suffix filtering for Opinosis, k=100

A.2 Saving fewer pairs in the hash table

The speedup gained for Top-k Join when saving all calculated pairs in a hash table compared to saving only those which will be calculated a second time, using the Opinosis data set.

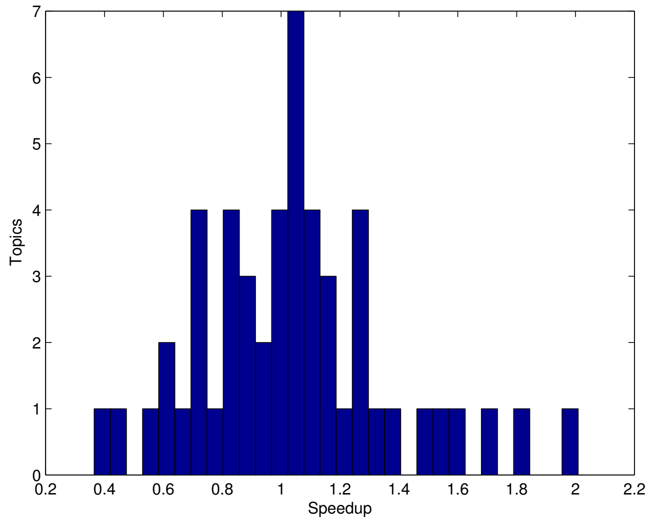


Figure A.1: Top-k Join speedup when saving all pairs in hash table for Opinosis, $k = 1$

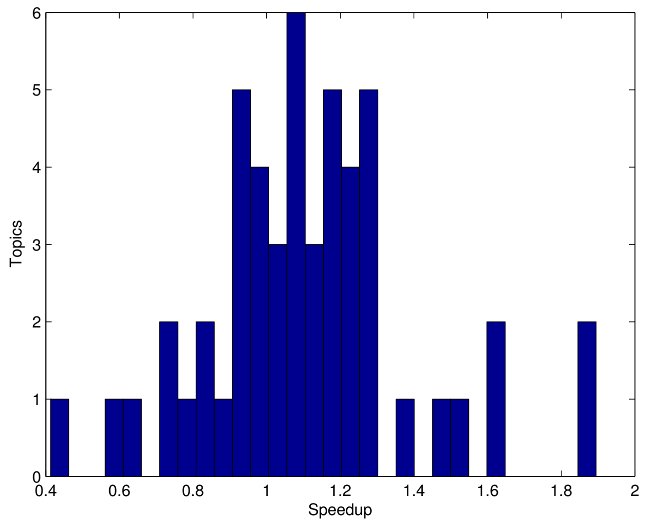


Figure A.2: Top-k Join speedup when saving all pairs in hash table for Opinosis, $k = 100$

A.3 Sorting the term-document matrix

Speedup and running times when sorting the term-document matrix for SBA.

Documents in topic	Time sorted (s)	Time unsorted (s)
50	0.013	0.014
72	0.012	0.020
126	0.020	0.019
170	0.029	0.025
215	0.024	0.017
266	0.034	0.044
333	0.043	0.065
575	0.105	0.164

Table A.3: SBA sorted vs unsorted term-document matrix for Opinosis, k=1

Documents in topic	Time sorted (s)	Time unsorted (s)
50	0.017	0.019
72	0.018	0.016
126	0.034	0.037
170	0.047	0.058
215	0.057	0.071
266	0.068	0.129
333	0.083	0.133
575	0.227	0.542

Table A.4: SBA sorted vs unsorted term-document matrix for Opinosis, k=100

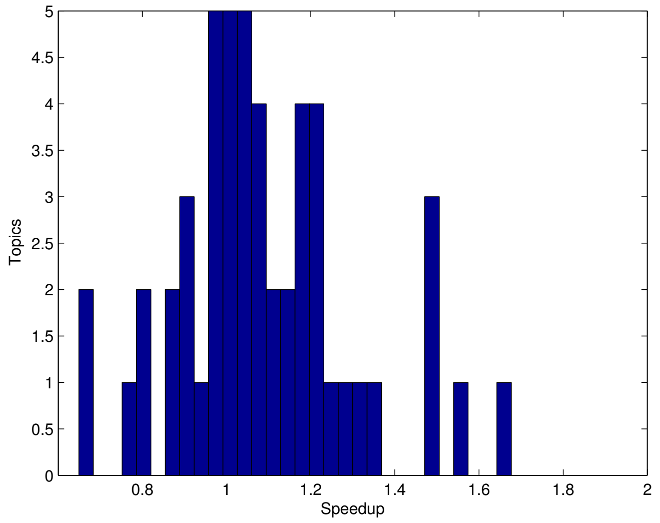


Figure A.3: Speedup using sorted term-document matrix instead of unsorted for Opinosis, $k = 1$

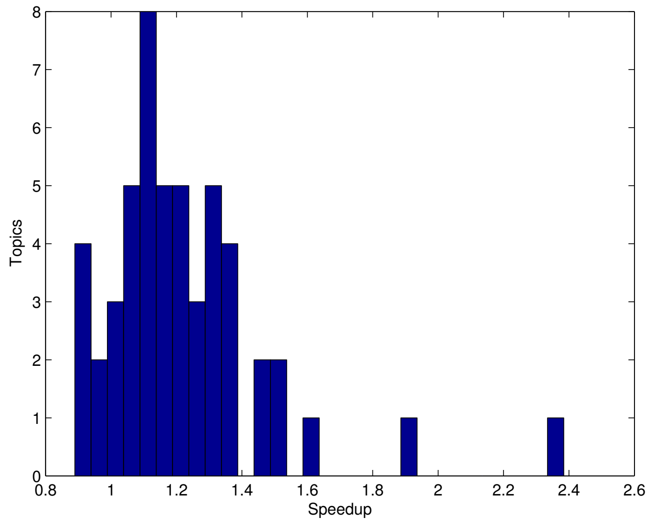


Figure A.4: Speedup using sorted term-document matrix instead of unsorted for Opinosis, $k = 100$

A.4 Running times

Running times for TOP-MATA, SBA, and Top-k Join for Opinosis and TREC.

A.4.1 Opinosis

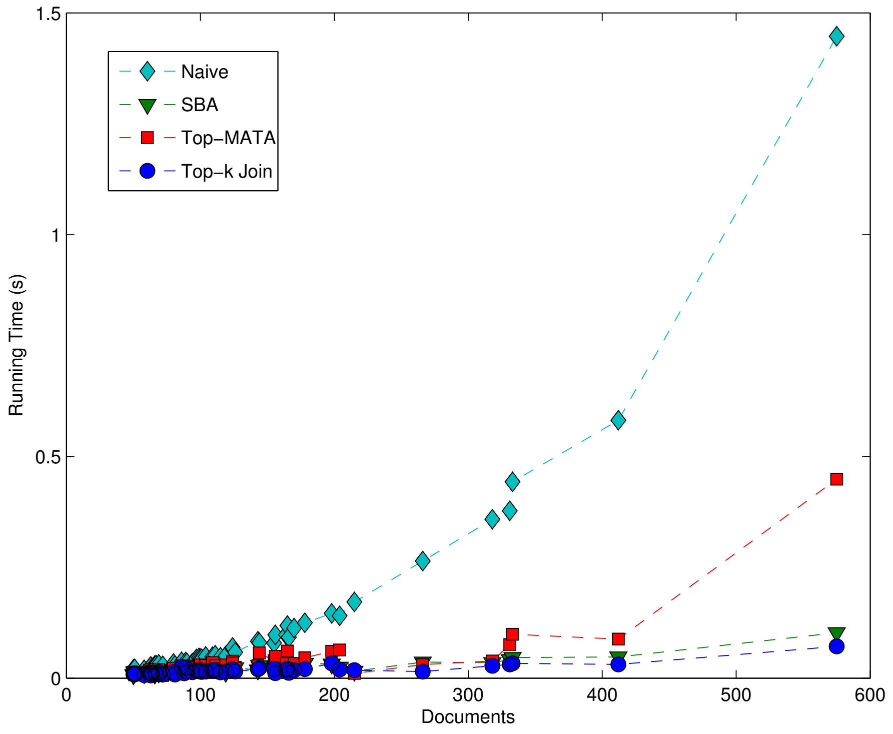


Figure A.5: Running times for topics in Opinosis, $k = 1$

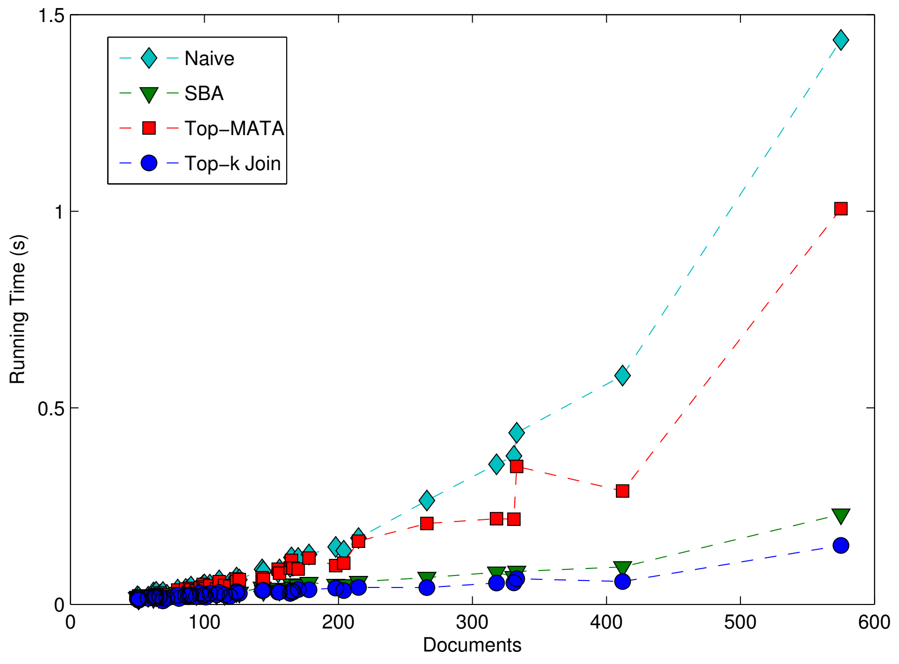


Figure A.6: Running times for topics in Opinosis, $k = 100$

A.4.2 TREC

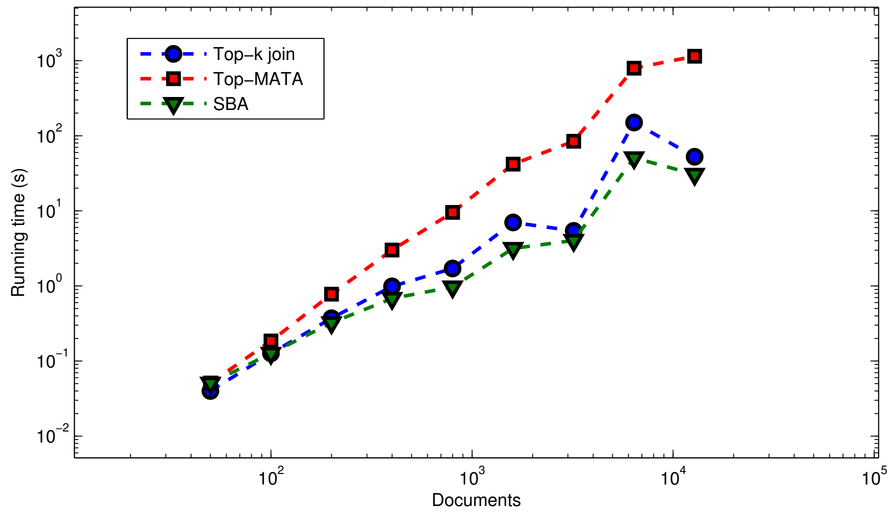


Figure A.7: Running times for TREC sets, $k = 1$

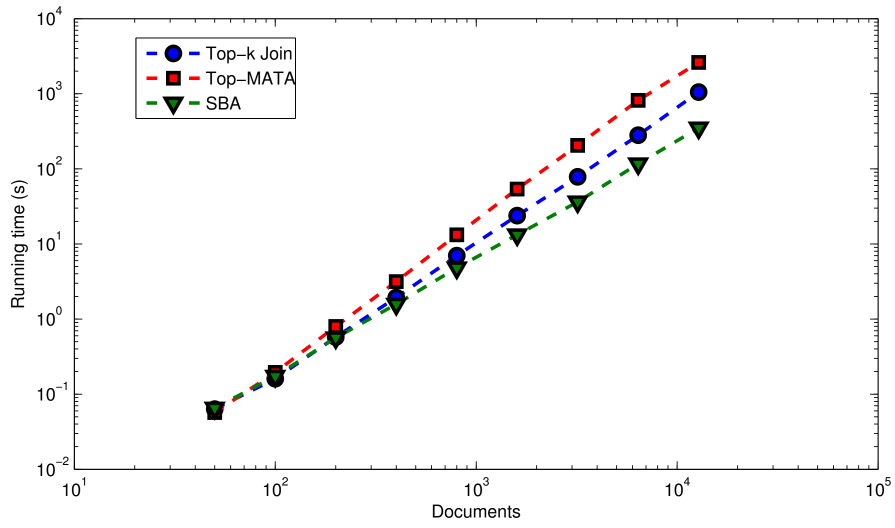


Figure A.8: Running times for TREC sets, $k = 100$

A.5 Top-k vs SBA

Detailed graphs of running times of Top-k Join and SBA using the Opinosis.

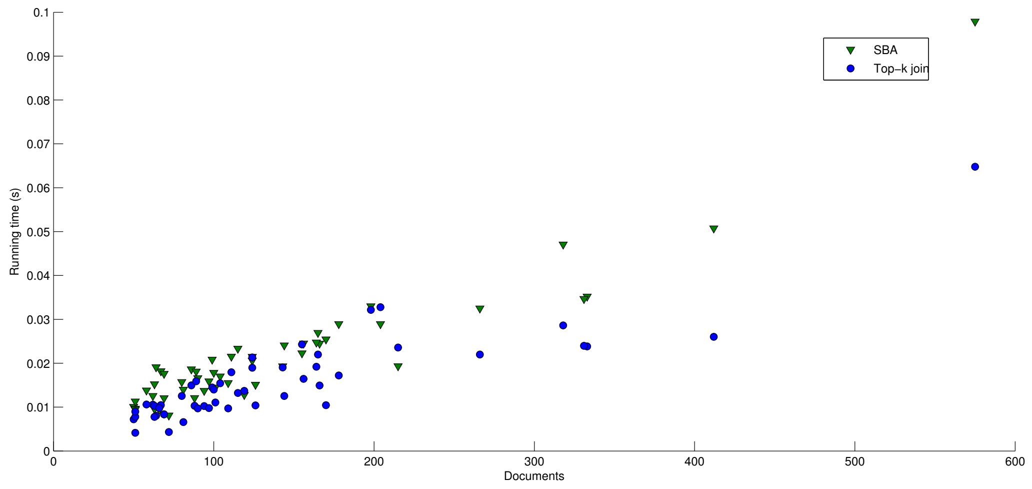


Figure A.9: Running times for Opinosis, $k = 1$

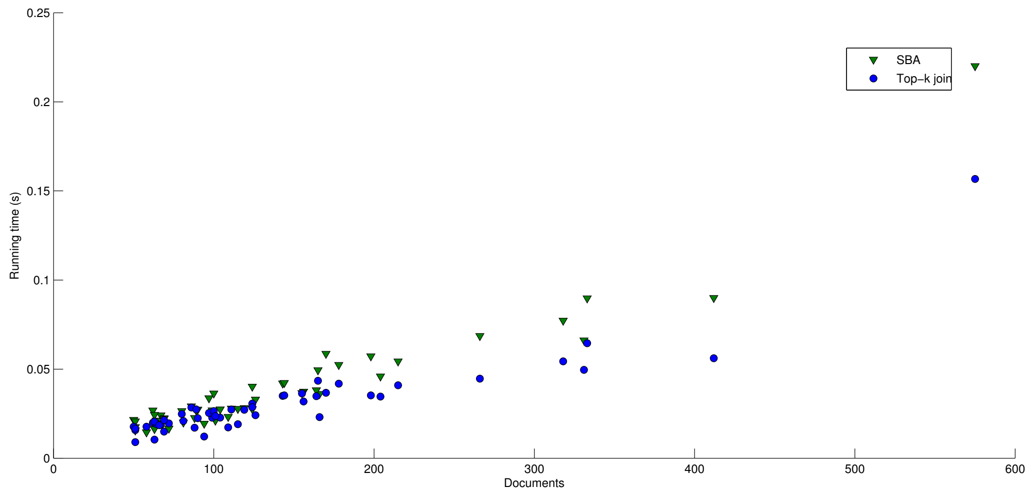


Figure A.10: Running times for Opinosis, $k = 100$