Textual Similarity

Johan van Beusekom
Peter Gammelgaard Poulsen

Kongens Lyngby 2012
IMM-B.Sc-2012-19

Technical University of Denmark
Informatics and Mathematical Modelling
Building 321, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk IMM-B.Sc-2012-19

Summary (English)

The goal of the thesis is to try out different algorithms intended for measuring semantic similarity between documents. In order to do this, a tool *Similarity Tool* has been developed in Java. The tool has four implemented algorithms that all can be run on a set of documents to compute the similarity scores between pairs of documents. To test out how accurately an algorithm solves the problem, similarity scores have been assigned to the pairs of documents in a set by both humans and algorithms and the correlation coefficients between the results have been calculated. The structure of the tool is discussed and the algorithms are then analyzed in terms of time and space complexity as well as accuracy. It is concluded that each algorithm has its own advantages and that it is possible to achieve satisfying results with all algorithms by using certain preprocessing methods.

Summary (Danish)

Målet for denne afhandling er at afprøve forskellige algorithmer beregnet til at måle den semantiske lighed mellem dokumenter. For at kunne gøre dette, er der blevet udviklet et værktøj, *Similarity Tool*, i Java. I værktøjet er der implementeret fire algoritmer, der alle kan blive kørt på et sæt af dokumenter og udregne de indbyrdes lighedsværdier mellem disse. For at undersøge hvor nøjagtigt en algoritme løser problemet, er lighedsværdierne blevet fastsat af både mennesker såvel som algoritmer, og correlationskoefficienten mellem disse værdier er blevet udregnet. Opbygningen af værktøjet bliver diskuteret og algoritmerne bliver analyseret med hensyn til tidskompleksitet og pladsforbrug såvel som nøjagtighed. Det bliver konkluderet at hver algoritme har sine egne fordele, og at det er muligt at opnå tilfredsstillende resultater med alle algoritmer, ved at bruge visse metoder.

Preface

This thesis was prepared at the department of Informatics and Mathematical Modelling at the Technical University of Denmark in fulfillment of the requirements for acquiring an B.Sc. in Engineering.

The thesis deals with the different aspects for finding similarities among textual content. Among a lot of other topics it deals with linguistics, data structures, parallel programming and statistics.

The thesis consists of an introduction to the subject of textual similarities. The next chapter contains an analysis of the problems for finding textual similarities follow by a chapter about the theoretical knowledge used in the solutions. Next the chosen designs are discussed and argued before a presentation of the implemented tool. The next chapter covers the results that has been obtained using the tool. Finally a conclusion to summarize the important findings through the thesis.

Johan van Beusekom
Peter Gammelgaard Poulsen

Acknowledgements

We would like to thank our supervisor Robin Sharp for our weekly conversations and for all the advices he has provided during the project.

We would also like to thank all the people who answered our questionnaire.

Contents

Chapter 1

Introduction

The measurement of textual similarity between documents is both an interesting and very much needed task. Textual similarity measures are heavily used in text retrieval systems such as search engines or tools used for detecting plagiarism or copyright infringement of texts where a document or list of words is given as the query and one or more documents similar to the query are returned. The problem of how to measure the textual similarity has been around for a long time and with the ever-growing Internet it is of great interest to improve the search engine queries in both speed and accuracy, therefore many different approaches on how to solve it have been proposed throughout the years.
Some tools for measuring textual similarity are solely based on the shared amount of terms in the two texts and while this may be a fast way to go about the problem, the expressions of the texts are lost, as the input documents are merely viewed as lists of words. To state whether two documents are semantically similar is a problem that is easily solved by a human but it is a very complex task for a computer. To a human it may be obvious that two documents might concern the same topic while they do not have very many terms in common, but for a computer to overcome such vocabulary mismatches, techniques such as stemming or query expansions can be included, which we will go into further detail with during the report.

1.1 Goal

The goal for this project is to create a tool that can be used for measuring textual similarities between an input document and a list of other documents. The tool should have several algorithms implemented and allow for the user to use certain flags such as stop word removal, stemming or *part of speech* tagging if he/she desires to test how these features affects a similarity measure.

1.1.1 Purpose

The purpose of this project is to look at different strategies used to compare the similarities in text documents. This will be done by analyzing known algorithms for comparing documents and extending them with strategies that are suited for human-written documents. All this will be implemented in an application that provides a GUI to easily test performance and accuracy of several strategies for doing textual similarities on documents. Each strategy will be based on one or more algorithms for solving such a problem.

1.1.2 Focus

The focus of this project will be on creating a tool that can act as an environment for testing several different approaches to the problem concerning textual similarities. It is important that new algorithms and strategies easily can be added to the application and already implemented approached can be tweaked to provide useful results.

1.1.3 Constraints

The meaning of the application is to provide a tool for testing several different strategies for textual similarities in documents. It means that the tool would not be suited for actually finding documents that are very similar but rather showing which strategy is most suited for solving the problem.

1.2 Thesis outline

Chapter 2 of the thesis will be an analysis. The analysis brings forth thoughts made on how to make the textual similarity query, i.e. which aspects of a text are interesting to look at, while not presenting concrete solutions to the problem.
Chapter 3 covers the theory and known solutions to this problem. In this chapter a discussion on the pros and cons of the different algorithms will be made, which will include analysis of running time, space complexity etc.
In chapter 4 the design will be discussed. We will look at the problem at hand and the thoughts made before starting the actual implementation with regards to use-cases, functionality requirement, structure of the code, the decisions that has been made during the project and other aspects that the final product should fulfill.
Chapter 5 covers the actual implementation where the structure of the program is laid out including tests. The problems encountered are explained and discussed.
In chapter 6 the results of different algorithms are presented and discussed.
Finally in chapter 7 is the conclusion where we try to give an answer to how the textual similarity problem is solved in the best way and if it is possible at all for a computer to answer a textual similarity query satisfactorily.

1.2.1 Common terms

Throughout the thesis several technical terms are used. A short explanation of words written in *italic* can be found in appendix A.

Chapter 2

Analysis

2.1 Chapter outline

This chapter will go over the thoughts that we as well as others have made towards the task of solving a textual similarity problem. We will also try to propose solutions to any possible problems.

2.2 Overview

To determine whether two documents concern the same topic is quite a different task than that of seeing how many terms they share. The latter task works by taking a list of terms as a query and returning a list of documents that match the query i.e. contain the same terms. Assuming a large enough *corpus* and a very specific query, the chances of getting a satisfactory answer to the query is quite high, however the method is very naive as it does not take the expressions of the documents into account.
Natural language processing (*NLP*) is a discipline in computer science that deals with extracting useful information from a natural language input. *NLP* concerns a long list of tasks that might be relevant for the preprocessing of a query.

One of the most obvious preprocessing operations is the *stop word* removal, in which all stop words are removed from the document, theoretically leaving it with words of more significance to the topic. In rare cases a query might be based around a *multi-word term* where one or more of the terms are stop words, an example of this could be "Take That"[1]. Both words in "Take That" are (often) considered stop words and would therefore be removed prior to handling the query, in which case stop word removal could decrease the correctness of the output. However a stop word list is human made and can therefore be shaped to suit the user, so it is easy to bypass the former mentioned problem.

Another problem with the words in a query is that they often appear in various conjugations which are all part of the same *lexeme* and therefore have the same *lemma* and possible also the same *stem*. This will of course be problematic if the similarity measure only looks for an exact match between terms in the query and *corpus*. Therefore it might be of interest as part of the preprocessing of a query to reduce all words to their stem or lemma. Stemming and lemmatization each have their own advantages
Consider the following examples:

- The word "went" has "go" as its lemma. A stemmer will not give this answer, as it requires a dictionary look-up.
- The word "dance" is the base form of "danced" which both stemming and lemmatization will conclude.
- The word "meeting" can either have the base form of a noun or a verb, which the stemmer will not know, while lemmatization can give the right answer based on the context of the word.

Stemming of words, which is a task of *NLP*, can be done very quickly as knowledge about the word's context is not needed, this however also means that the result might not be as usable as with lemmatization, for example with the word "ran" a stemmer would simply return "ran" as the stem while lemmatization would return "run".

The stemming or lemmatization of words in a document might help the accuracy of the algorithms to some extent, but there is still an obvious problem with only looking at terms explicitly written in the documents. For example if two documents concern two different types of dogs, it is obvious to humans that they share a common topic *"dogs", "pets"* or *"animals"*. A query expansion can be used in order to take terms that are not explicitly in the query, but somehow related to it, into account. The theory chapter will cover more about

[1]The name of an english band.

the concept of query expansion and discuss which linguistic relations that could be interesting to include and how this should be done.

Another interesting aspect is the *word classes*. Human based judgements may be based heavily on one or more specific word classes, for example nouns or verbs. If this is the case, it might be interesting to look at only these words when processing a query.

The representation of the content may also play a part in how people compare documents[UDK04b]. While some documents may not concern the same topic at all, a person might still see them as similar if they are written in a similar style. This question of content similarity versus expression similarity might be an interesting viewpoint to bring into the picture, as expression similarity is often neglected in tasks that seek to solve the textual similarity problem [UDK04b].
Consider the following sentences:

- Early this morning John and his wife were robbed on their way to the baker. The robber was caught later during the day.
- Today the police caught Mike, who is believed to have committed a robbery on an elderly couple just this morning.
- Early this morning a young man ran across a red light on his way to work. The man did it again later during the day.

Sentence 1 and 2 are similar in content, but not very similar in expression while sentence 1 and 3 are similar in expression, but not in content. A person might rate the two first sentences as being very similar because they are talking about the same incident, but may also rate the first and last sentence as somewhat similar because they are structured in the same way.
In [UDK04b] a questionnaire is made in order to test how people evaluate content and expression similarity, and they come to the conclusion that both the expression and content of a document are of importance to how a person makes a similarity judgement. In the theory chapter it will be discussed how the expression of a document can be captured.

2.3 Requirements analysis

A tool for measuring textual similarities has been developed as part of this thesis. The requirement analysis that was made to this tool before the implementation started, is listed below.

2.3.1 Measure

A way to compare the results of different approaches that have been implemented should be defined. This can be done by selecting a set of text documents and ask a group of people to read and compare each pair of documents and give them ratings based on their similarity and computing the correlation between the human scores and algorithm scores.

2.3.2 Functional requirements

The tool should fulfill the following functional requirements.

- Basic operations such as loading documents, stemming, removing stop words.
- Integration with lexical database.
- Integration with a POS tagger.
- Contain at least 3 algorithms that can be used to compare documents.
- Provide results for each algorithm that can be used to compare their performance and accuracy.
- For each strategy it should be possible to tweak several parameters such as whether including synonyms, stemming words or removal of stop word removal should be turned on.

The first algorithm to be implemented should be based on Levenshtein distance. The implementation of the algorithm is quite simple and it will provide a way of setting up the environment and get things up and running.

Later more algorithms will be chosen among following candidates:

- Longest Common Subsequence
- Vector Space Model
- Term Frequency-Inverse Document Frequency (*TF*IDF*)

- Jaro-Winkler
- Ontology Based Query
- Textual Fuzzy Logic

2.3.3 Nonfunctional requirements

The tool should fulfill the following nonfunctional requirements.

- An easy to use GUI (usability).
- Run on several platforms (Linux, Windows, Mac OS X).
- Provide acceptable performance (algorithms and data structures).
- Contain low bug rate (reliability).
- Parallel processing of documents (concurrent programming).

2.3.4 Use cases

A typical use case for the application is listed below. The corresponding use case diagram can be found on appendix F.

2.3.4.1 Compute similarity score.

Main Success Scenario:

1. The user starts tool.
2. The user loads the main document.
3. The system loads main document into data structures.
4. The user loads additional documents for comparison.
5. The user chooses an algorithm.
6. The user selects custom option such as stemming, pos-tagging etc.

7. The user presses compute button.
8. The system loads documents into data structures.
9. The system performs necessary preprocessing operations based on user chosen options.
10. The system computes similarity scores.
11. The system presents scores visually to the user.
12. The user can perform any of step [2,7] again.

Extensions:

8a: Either no main document or additional documents chosen. Computation halted.

8b: Invalid combination of custom options. Computation halted.

8c: The user has chosen an option that requires external library, which is not installed. Computation halted.

9a: System caches document information if certain computationally hard options have been chosen.

Chapter 3

Theory

3.1 Chapter outline

This chapter starts by explaining some processes of *NLP* that are relevant to the textual similarity problem. A classification of the different known algorithms and approaches to solve the textual similarity problem will also be done, as well as the theoretical background for the algorithms.

3.2 On linguistics

Linguistics is the study of human language and contains the three categories *language form, language meaning* and *language in context.* More specifically language form is the study of a languages structure or grammar and focuses on the rules that are used when constructing phrases in a natural language.
Language meaning is concerned with the logical structuring of natural languages to assign meaning to something and resolve ambiguity. Among the subfields of this area are both semantics and pragmatics.
Language in context is a study about the evolution of natural languages as well as languages in relation to each other. Many languages originate from the

same language family, such as danish, swedish and norwegian that all come from the family of Germanic languages, who again is a subfamily of Indo-European languages.

Often more abstract levels of analysis can benefit from reliable low-level information [Mit03], which is why the *NLP* is an important part of preprocessing a query in the textual similarity problem. Natural language processing uses the knowledge of linguistics to perform several tasks that allow for the extraction of useful information from natural language input. The next sections will cover some of the relevant tasks of NLP in relation to the textual similarity query.

3.3 Part-of-speech tagging

In most natural languages some words can appear in different word classes. For example in english the word "fly" can either be noun, referring to the insect, or a verb, referring to flying. To resolve which word class the word belongs to in the given context part-of-speech tagging (*POS-tagging*) can be used, which is the process of word-class disambiguation that assigns contextually appropriate grammatical descriptors to words in a text [Mit03]. There exists two types of algorithms for POS-tagging; rule-based and stochastic.
The rule-based POS-taggers are, as the name implies, based on rules in the natural language such as the grammar and syntax. A simple rule could for example be: "article then noun can occur, but article verb cannot".
The stochastic POS-taggers are based on statistics made on a corpus. They are trained on the corpus, and use statistical methods to determine the optimal sequence of part-of-speech tags. The stochastic POS-taggers are the most common, as they are easily made and have high accuracy, however a disadvantage of the stochastic POS taggers is the huge amount of stored information that is required. In figure 3.1 is an example of how a POS-tagger can work its way through a sentence, by first recursively identifying noun and verb phrases, before determining word classes of the atomic elements, i.e. the words. In appendix C is a list of the different word classes used in this thesis.

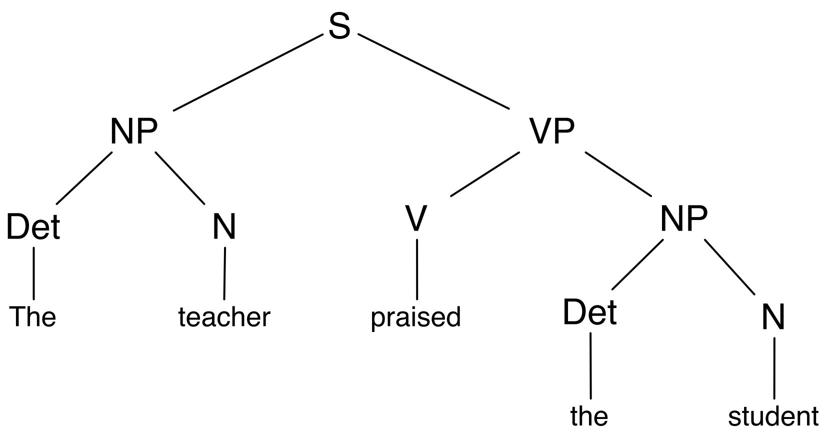


Figure 3.1: POS-tag tree for the sentence "The teacher praised the student".

3.4 Word-sense disambiguation

While part-of-speech tagging might help a computer understand the semantics of a document a lot better it cannot solve the *word sense disambiguation problem*. Some words have multiple meanings[1] and the detection of which sense of a word is used in a text is called *word-sense disambiguation*. Identifying the sense of a word wrongly might drastically change ones perception of a what a text is about, but this is very rarely an issue for humans. For a computer however, solving the problem is far from trivial.
An example of an ambiguous word is the word "club":

- A blunt weapon.
- A group of persons organized for a social or other purpose.

Now consider the sentences:

- The caveman hit his wife with the club.
- The young girl joined a study club.

[1]polysemy

To a human it is obvious that in the first sentence the sense of "club" is a weapon, and in the second sentence it is referred to as a group of people. There exists many algorithms that try to solve the word sense ambiguity problem. The algorithms are, like those concerning POS-tagging, based around different approaches where some are trained on a corpus of manually sense disambiguated examples. Another approach use dictionary definitions of words close to each other in a text to see which sense definitions have the greatest overlap, meaning that these senses are the most likely for the words in the context.
Some approaches are completely unsupervised meaning they work without the use of external information and work by clustering occurrences of words, and determining the sense. Of the mentioned methods, the most common and accurate approach has shown to be the one trained on a corpus.
This problem is far more complex than the POS-tagging problem, which could affect its usability in real time applications.

3.5 Ontology

The term ontology originates from philosophy where it is the study of the nature of being. In computer science an ontology is a set of concepts within a domain and the relationship between these concepts. In a textual similarity project context one might choose ontologies to be knowledge structures that specify terms and their properties as well as relations among these terms as it thought in [OP03]. Ontologies consist of many different components that all contain information about the nature of the ontology. In this thesis, the most interesting concept of an ontology is linguistic relations(shown below) to other terms.

- *Hypernymy* - Describes the relation between a term and a generalization of said term, a hypernym. For example the word "vehicle" is a hypernym of the term "car".

- *Hyponomy* - Is the opposite of hypernymy and describes the relation between a term and a specification of said term. The specification is called a hyponym, and an example of this could be the word "shirt" which is a hyponym of the term "clothes". The relation is also called an *is-a* relation.

- *Synonymy* - The relationship between words that are synonymous i.e. have the same meaning, such as the words "sick" and "ill".

- *Antonomy* - Like synonyms, but instead of having the same meaning, antonyms are words that have opposite meanings. Example; "hot" and "cold".

- *Meronymy* - A meronym describes a *has-a* relation. For example a "leg" is a meronym of a "body", because a leg is a part of a body, i.e. body has-a leg.

- *Holonymy* - The opposite of meronymy. It describes what a term is part of. For example "car" is a holonym of "wheel", i.e. wheel is a part of a car.

Extracting knowledge about these relations when processing a query, could help expand the query to not just including the terms in the query, but also some or all of the semantically related terms listed above, and use this information to perform the similarity measure more accurately. Later in this chapter it will be discussed how inclusion of the previously mentioned terms could be included in an actual similarity algorithm.

3.6 Textual Similarity Algorithms

As it has been mentioned earlier, a lot of algorithms exist that try to solve the textual similarity problem. This section covers some of the known algorithms.

3.6.1 Levenshtein distance

The Levenshtein distance, also known as the edit distance, is a number defining the minimum numbers of edits[2] that must be performed on one string in order for it to be identical to another string. The calculation of the Levenshtein distance can be visualized represented as a matrix where the first string makes up the columns and the second string makes up the rows. For each entry in the matrix the characters at the row and column are evaluated, and if they are not equal, a deletion, insertion og substitution must be performed with the cost of 1 edit.
An example of the Levenshtein distance between the two strings "dance" and "dive" is shown here.:

[2] deletions, insertions and substitutions

		d	a	n	c	e
	0	1	2	3	4	5
d	1	0	1	2	3	4
i	2	1	1	2	3	4
v	3	2	2	2	3	4
e	3	3	3	3	3	3

The edit distance is shown as the green number in the last index of the matrix. The edit distance between "dance" and "dive" is 3.
For two strings of length n and m the running time of the algorithm is $O(n \cdot m)$. The space complexity of the algorithm can be reduced to $O(n)$, because the entire matrix doesn't have to be kept in memory at once, but only 2 rows of length n.

If this algorithm were to be used on documents rather than short strings it is obvious that looking at the amount of symbols that needs to be changed does not give much information about the similarity of the two documents, so instead of this, the algorithm calculates how many words in one of the documents for it to be identical to the other. The edit distance should be evaluated with respect to the length of the documents, as longer documents cause higher edit distances. The lower the score, the higher similarity between the documents.

The advantages of this algorithm is it's simplicity and ability to give documents credit for beign similar in structure. A downside of the algorithm is the fact that it neglects large parts of the documents and it is very susceptible to noise.

3.6.2 Longest Common Subsequence

An algorithm concerned with finding the longest common subsequence between two sequences. In this context, the atomic elements in the sequences are the words of the two documents.
Given the two sentences:

- "The man jumped over the fence."
- "A man bought the fence."

A common subsequence of the two sentences is "fence" and the longest common subsequence is "man the fence".

The running time of the algorithm, when two sequences of length n and m respectively, is $O(n \cdot m)$ when using a dynamic programming approach.

This algorithm has the same advantages and disadvantages as the Levenshtein distance.

3.6.3 Jaro-Winkler

With this algorithm a number, the *Jaro-Winkler distance*, is computed as a measure of the similarity between two strings. The numbers is in the interval from 0.0 to 1.0, and the higher the score the is the higher the similarity is. It is based on the *Jaro* distance d_j, which for two strings s_1 and s_2 is defined as:

$$d_j = \frac{1}{3}(\frac{m}{|s_1|} + \frac{m}{|s_2|} + \frac{m-t}{m}) \tag{3.1}$$

Where m is the number of matching characters and t is the number of transpositions. The number of transpositions is the number of mismatched characters[3] divided by two. Only characters within distance $\lfloor \frac{max(|s_1|,|s_2|)}{2} \rfloor - 1$ are considered when determining the number of maximum matching characters.
The Jaro-Winkler distance uses this number in the formula:

$$d_w = d_j + (l \cdot p(1 - d_j)) \tag{3.2}$$

Where p is a scale (usually 0.1) that ensures that more favorable ratings are given to strings that match from the beginning of a prefix of length l, i.e. l is the length of the longest common prefix between the two strings s_1 and s_2.
An example of the *Jaro-Winkler* distance on the two strings "home" and "hope" is shown here:

$m = 3$
$|s_1| = 4$
$|s_2| = 4$

$t = 0$ as there are no mismatched characters.

$$d_j = \frac{1}{3}(\frac{3}{4} + \frac{3}{4} + \frac{4-0}{4}) = 0.83 \tag{3.3}$$

p is set to 0.1 and we find that $l = 2$. This gives us the *Jaro-Winkler distance*:

$$d_w = 0.83 + (2 \cdot 0.1(1 - 0.83)) = 0.86 \tag{3.4}$$

[3]matching but in the wrong sequence

Unlike the *Levenshtein distance* and the *LCS* algorithms, this algorithm does not completely neglect shared elements that are not in the same sequence, however sequence mismatch penalizes the result, which might be a good thing, because while the strings have a match, it is not represented in the same way. Another interesting point is that it only looks for matches within a certain distance of the item that is to be matched, which is both positive and negative. Positive because it reduces the running time and negative because on long strings, the algorithm will not find shared words that far from each other. When using this algorithm on documents rather than short strings, the atomic actions are words instead of characters.
The space complexity of this algorithm is $O(n + m)$ and the running time is $O(n \cdot \lfloor \frac{max(|s_1|,|s_2|)}{2} \rfloor)$.

3.6.4 Vector Space Model

The *Vector Space Model* (VSM) is a way of representing documents as points in space (vectors in a vectorspace). The idea is that points that are close to each other are semantically similar, while points far from each other are semantically distant [TP10]. The VSM is particularly good for indexing and relevancy ranking of documents in regard to a query. In the VSM, the vectors have a dimension equal to number of distinct words in the corpus, where each dimension corresponds to a separate word. In each dimension of each vector, the number of occurrences for the corresponding word is stored. The similarity between two documents is then calculated as the angle between the two vectors. For the query q and a document d, the similarity is calculated as follows:

$$\Theta = \cos^{-1}(\frac{d \cdot q}{||d|| \cdot ||q||}) \tag{3.5}$$

The smaller the angle is between the two vectors, the more semantically similar the documents they represent are.
The fact that the VSM does not take the sequence of the words within the documents into account can both an advantage because it makes it unable to capture expressions used in the documents, but on the other hand, it credits documents for containing the same words, even if they are ordered differently, unlike the Levenshtein distance. The comparison of vectors can be done in linear time of the number of dimensions of the vector. The construction of the vectors requires that all distinct terms in the corpus are determined. If the size of the corpus is N, then this can be done in $O(N)$. Insertion into the dimensions of the vector i can be done in $O(n)$ time, where n is the amount of words in document i. The space complexity of the vectors is $O(U)$ where U is the number of unique words in the corpus.

3.6.5 Term Frequency-Inverse Document Frequency

Or *TF*IDF* for short, is a variation of the VSM. What sets it apart from the VSM, is that it takes into account how important a word is for a document. The idea is, that words that occur in many documents in a corpus are less important for the meaning of the documents and vice versa. The inverse document frequency of a term t, which is the measure of a words commonality in the corpus, is calculated as:

$$idf(t,d) = \log(\frac{|D|}{1 + |d \in D : t \in d|}) \tag{3.6}$$

Where D is the number of documents in the corpus and $1 = |d \in D : t \in d|$ is the number of documents in the corpus that contain t. The *TF*IDF* is then calculated as:

$$tf(t,d) \cdot idf(t,D) \tag{3.7}$$

Where $tf(t,d)$ is the term frequency t in document d.
Normally a query in the *TF*IDF* is a set of words, and in this case, the input query would be a list containing the words of the document we want to compare to the other documents.
The interesting thing about *TF*IDF* is the fact that it has the ability to find words that are less interesting to the meaning of a text and down prioritize these when computing the textual similarity, hence why the algorithm is also usable for finding stop words on its own. This algorithm needs to keep information about the unique words in the corpus in memory, giving it a space complexity $O(U)$. The time and space complexity are the same as in the VSM, but an additional vector has to be kept in memory, holding information about how many documents the unique words of the corpus appear in. Creation of this vector takes $O(N)$ time, and it uses $O(U)$ space.

3.6.6 Textual Fuzzy Similarity

Moving away from the data vector models, lets have a look at Textual Fuzzy Similarity (TFS). Fuzzy similarity tries to solve an issue that all of the previous covered methods have had, namely that there is need for an exact match between words in query and document, in order for the words to actually be considered matching. In fuzzy logic, as it is presented in [SG03], the idea is that word pairs can be similar, while not being identical. Computing the similarity between words with this method can be done without the use of any knowledge base or dictionary. The fuzzy aspect of the algorithm is that a word pair similarities can be any number in the interval [0,1].

A fuzzy set is a set of pairs, defined as

$$A = < x, \mu_A(x) : x \in X > \tag{3.8}$$

Where $\mu_A(x) : X \rightarrow [0,1]$ is the similarity function for x in the set.[SG03]
A fuzzy relation is defined as

$$R = < (x,y), \mu_R(x,y) >: x \in X, y \in Y \tag{3.9}$$

with $\mu_R : X \times Y \rightarrow [0,1]$ as the similarity function between x and y. The similarity function should have the following properties:

Reflexivity: $\mu_R(x,x) = 1$

Symmetry: $\mu_R(x,y) = \mu_R(y,x)$

For two documents, a fuzzy relation would have to be set up like this:

$$RW = (< w_1, w_2 >, \mu_{RW}(w_1, w_2)) : w_1, w_2 \in W \tag{3.10}$$

where W is the set containing all words from the two documents. Now lets take a look at a possible membership function $\mu_{RW}(w_1, w_2) : W \times W \rightarrow [0,1]$ for two words as stated in[SG03]:

$$\mu_{RW}(w_1, w_2) = \frac{2}{N^2 + N} \sum_{i=1}^{N(w_1)} \sum_{j=1}^{N(w_1 - i + 1)} h(i,j) \tag{3.11}$$

Where:

- $N(w_1)$ is the number of letters in w_1
- $N(w_2)$ is the number of letters in w_2
- N is $max(N(w_1), N(w_2))$
- $h(i,j) = 1$ if a subsequence of i letters from index j in w_1 is located at least once in w_2. If it doesn't, $h(i,j) = 0$.

Here is an example that should help understanding how the algorithm works:

The two words w_1 and w_2 are "choose" and "chose" respectively.
We have that $N = 6$. The fuzzy relation between the words is:

$$\mu_{RW}(w_1, w_2) = \frac{2}{36+6} \sum_{i=1}^{6} \sum_{j=1}^{5} h(i,j) = \frac{2 \cdot (5+4+2)}{42} = \frac{11}{21} = 0.52 \tag{3.12}$$

We get this because in w_2 the following substrings of length shown below match:

1. (c,h,o,s,e)
2. (ch,ho,os,se)
3. (cho,ose)

As mentioned earlier, the higher the score is, the more semantically similar the two words are.

This function for word similarity can be used in document comparisons, when it is not possible to find an exact match of words between the two documents. For document similarity, the documents are represented as sets of words rather than sequences, and a possible similarity function is as follows [SG03]:

$$\mu_{RD}(d_1, d_2) = \frac{1}{N} \sum_{i=1}^{N(d_1)} \max_{j \in \{1,..,N(d_2)\}} \mu_{RW}(w_i, w_j) \tag{3.13}$$

Where:

- $N(d_1)$ is the number of words in d_1
- $N(d_2)$ is the number of words in d_2
- N is $max(N(d_1), N(d_2))$

If a word from d_1 is not found in d_2 the word with the highest similarity score is chosen instead. The similarity scores for all unique word pairs in the involved documents can be pre-computed in $O(u \cdot v)$ time, where u and v are the numbers of distinct words in the respective documents. Finding the unique word pairs takes $O(n + m)$ time where n and m are the number of words in the respective documents. Only similarity scores above an arbitrary threshold are kept. Assuming the similarity scores for all unique word pairs are stored, such that the highest similarity score of a word can be accessed in constant time, we get that the running time of the algorithm is $O(u)$ with $O(n+m+u \cdot v)$ preprocessing time.

This method is interesting because it assumes that the presence of identical sequences of letters could mean semantic similarity. However this is not always the case, and there is a possibility that this assumption might reduce the quality of the final result.
From assuming similarity between words that are not identical, lets move on to a knowledge based algorithm, that can look beyond what is explicitly stated in the documents.

3.6.7 Ontology Based Query

What sets this approach apart from previously covered approaches, is that it acquires new knowledge about the documents using a knowledge base. Now the visual appearance of words in the documents is not enough. It is vital for this algorithm to work that it knows exactly which sense of a word is in question. For this, *POS-tagging* and word sense disambiguation can be used. The reason why it is important to have this knowledge about the words in a query, is that it allows for *query expansion*. Figure 3.2 shows two documents A and B represented as sets of words. Document A has been expanded to include more words, which is shown as the set A+. The intersection $A \cap B$ contains the words that are both in A and B. The intersection $A + \cap B$ contains words that are in B and are not in A, but related to the words in A.
In section 3.5, some possible concepts that could be included in a query expansion were listed.
The inclusion of these concepts could be modeled as a directed weighted graph where the the distance between concepts mark their relationship. The further the distance, the smaller similarity. The query expansion could range several layers out, meaning that not only concepts directly related to the query are included. Of course, with the inclusion of more layers, the size of the graph expands rapidly while the concepts furthest from the core concept are probably not of much use, as their similarity is very small.

Here only the specifications, i.e. the hyponyms, are included, but the addition of generalization i.e. the hypernyms, allows for calculating a similarity score between concepts on the same level of abstraction, such a "sandal" and "sneaker" in figure 3.3.

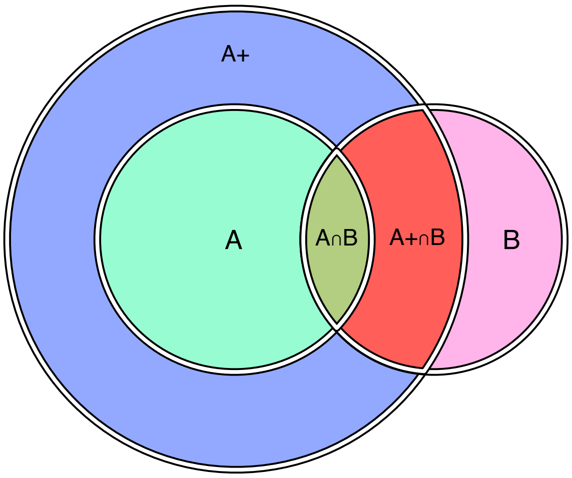


Figure 3.2: Expansion of the query A.

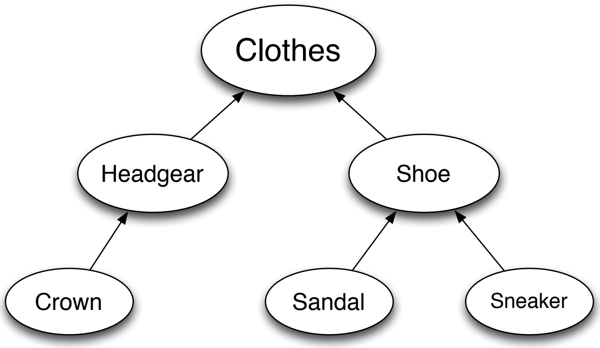


Figure 3.3: Graph representing an *is-a* relationship. For example sandal is a (type of) shoe.

A similarity function *sim* as stated in [BKA02], should have the following properties:

- $sim : U \times U \rightarrow [0, 1]$ - The similarity score between all concepts in U should be in the interval [0,1].
- $sim(X, Y) = 1$ if and only if $X = Y$.
- $sim(X, Y) < sim(X, Z)$ if and only if $dist(X, Y) < dist(X, Z)$.

For each type of edges, a value in the interval [0,1] should be assigned, telling something about the similarity between two concepts connected via such an edge. The higher the value, the greater similarity. The following example only uses the concepts of generalization and specification, but it would be possible to build the graph with inclusion of more concepts. The values for generalization and specification edges are respectively represented as γ and δ. A notion to make about the values of γ and δ is that a generalization is normally less relevant to a concept than a specification, as a specification has the same attributes as the original concept, while a generalization does not necessarily.
When modeled as a graph, $G = (V, E)$, the similarity of concepts is defined as the edge-product of the maximum cost path between the two concepts to be compared [BKA02].

$$sim(X,Y) = \prod_{p(e_{i=0})}^{p(e_n)} v(e_i) \tag{3.14}$$

Where

- p is the maximum cost path[4] between concept X and Y in G.
- $p(e_i)$ is edge number i on path p.
- $v(e_i)$ is the edge value of edge number i on p.

In figure 3.4 the graph from figure 3.3 has been expanded to include edges for concept specification, and appropriate edge values [BKA02] have been added to the edges.

Using equation 3.14, the semantic distance between concepts can be calculated. For example, the similarity between "sandal" and "crown" is:

$sim("sandal","crown") = 0.4 \cdot 0.4 \cdot 0.9 \cdot 0.9 = 0.13$

While the similarity between "sandal" and "sneaker" is:

$sim("sandal","sneaker") = 0.4 \cdot 0.9 = 0.36$

From this, it can be concluded that sandal and sneaker are more semantically related than sandal and crown. This algorithm uses a principal of fuzzy similarity, and the similarity between concepts is not either false or true meaning identical or not identical.

[4]multiplicative

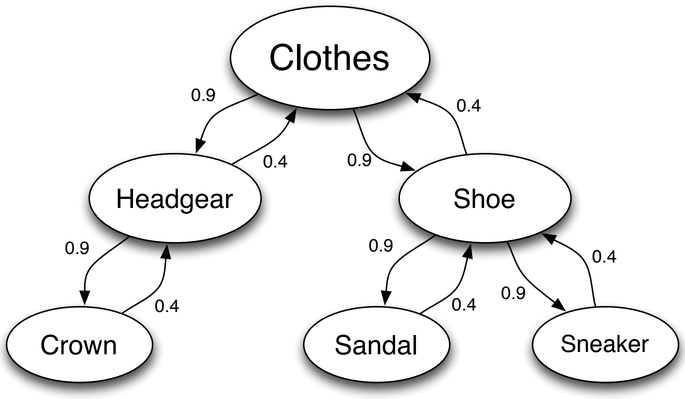


Figure 3.4: Concept relation graph, with edge values defining the similarity between concepts.

In contrast to the textual fuzzy similarity, this algorithm does not make any assumption about word's relatedness. It uses a knowledge base containing all of these relations, while TFS bases its similarity of concepts on the visual similarity between them. This gives the ontology based query an advantage over the TFS.
To use this concept on documents, as a part of the preprocessing, the query expansion shoul be done. One solution could be have a graph for each unique word that is explicitly in the query, where related concepts up to an arbitrary *distance class* are stored along with their similarity to the core concept, an example of this is shown below, for the concept "vegetable", with maximum *distance class* of 2.

A maximum distance class of two can result in a massive expansion in the amount of concepts related to vegetable, meaning that a certain threshold could be required in order for concepts to be considered "related", to decrease the space usage but also eliminate irrelevant relations.
When processing a query, first a match on the core concept is sought in the document for comparison. If there is no match the graph is traversed, seeking matches for the nodes in descending order, such that the best matches are sought first. Using figure 3.5 as an example, if there is no match on "vegetable", "potato" and "carrot" are tried and so on. Alternatively instead of a tree, the concepts could be stored in a sorted list, which would make it easy to go through them.

The final output of the algorithm using this concept should have the follow-

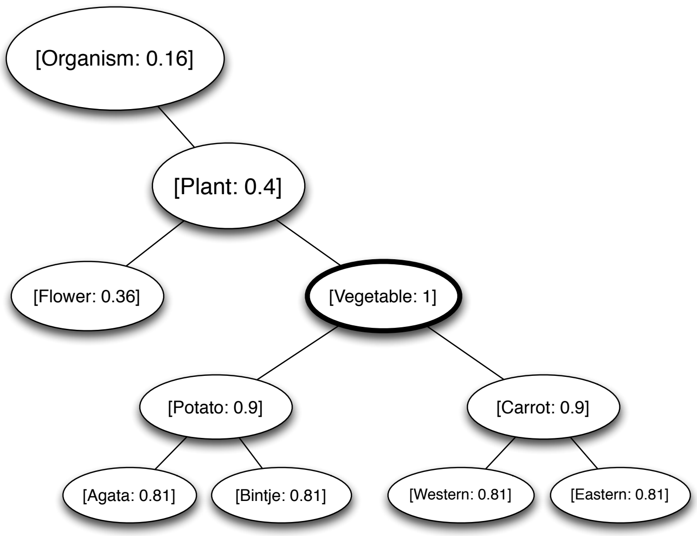


Figure 3.5: Concept relation graph for the concept "vegetable". The graph shows only a small subset of the concepts within edge-distance 2 of "vegetable".

ing properties:

- $sim : U \times U \rightarrow [0,1]$ - The similarity measure between document and query should be in the interval [0,1]
- The higher the similarity score, the more similarity between documents.
- $sim(d,q) = 1$ *iff* $w(d) = w(q)$ - Maximal similarity means that the two documents contain the same words.

Traversal of the concept-tree or list, can be done in linear time of the amount of concepts, assuming they are sorted as part of the preprocessing. This means that for a query, the traversal of all words and their related concepts can be done in $O(n_+)$ time where n_+ is the total size of all unique concept trees in the query. Assuming that the preprocessing also computes the occurrences of each unique word in the text, only one lookup for each unique term is needed and can then be multiplied by the amount of occurrences. Using a hash map with no collisions [5] to store words of the document for comparison, lookup operations

[5]collisions resolved in preprocessing

can be done in constant time, giving a total running time of:

$O(u_+)$

Where u_+ is the size of all unique concept trees in the query.
While the space complexity is the total size of the unique trees plus the total amount of unique words in the document for comparison, v:

$O(u_+ + v)$

3.7 Expression versus content

From the previous section it follows that a lot of different approaches exist on how to measure textual similarity between documents. Some algorithms are concerned with how similar documents are in structure while others are concerned with how different they are, i.e. how much needs to be changed for them to be identical. Other algorithms are not concerned with structure at all, and instead look at the shared vocabulary between documents.
Then there are the algorithms that try to be a bit more intelligent and look beyond what is explicitly stated in the documents and instead introduce a fuzzy similarity between concepts that are not necessarily identical, either using formulas that derive semantic relationships based on word structures or using a knowledge base that holds information of the semantic relatedness between concepts.
All of these methods have been tested out in the past, and most of them have proved useful to some degree when determining textual similarity, yet they do it in quite different ways. It could be interesting to look at how different algorithmic approaches could be put together to create an algorithm that takes the best of both worlds. In [UDK04b] a study has been made to see how people rate expression and content in terms of their importance to the meaning of a text. The paper claims that expression of text is often neglected, which means that for example a characteristic writing style of a writer is not included in the similarity evaluation. The conclusion made in [UDK04b] is that knowledge of stylistic components in a document help understand how a human perceives the text. Methods like the similarity vectors, fuzzy similarity or query expansion all use the *bag of words* model, meaning that the expression of the text is lost. Therefore it could be interesting to combine these methods with aspects of edit distance to include document expression.

CHAPTER 4

Design

4.1 Chapter outline

The chapter starts with a discussion of the overall design and the algorithms we chose to implement. Then a discussion on how to go from a file to a processed document. All of the important design choices are discussed with a focus on flexibility and performance.

4.2 Overall design

Before designing the overall structure several goals were established. The main goal was to develop a tool that could calculate the similarity among documents using different algorithms and strategies. The process of going from some textual content from a data source e.g. text files on the hard drive to calculated results requires several steps. The content must be loaded into the program, processed using methods mentioned in chapter 2 and then finally a score must be computed using an algorithm. This is illustrated in figure 4.1.

The user should be able to specify which documents the program should use

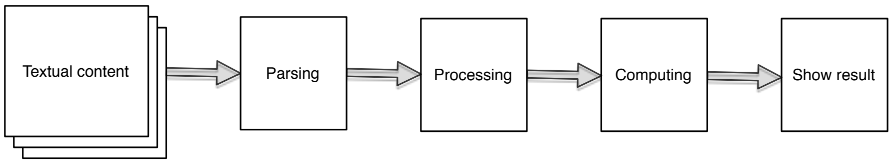


Figure 4.1: The process of going from some textual content to results.

as well as the algorithm that should provide the result. Furthermore the user should be able to adjust how the content should be processed. To make this easy for the user, the tool should have a visual layer that the user can interact with. In order to keep the program structured and different parts of it separated, design patterns must be used when structuring the overall design. *Model-View-Controller* has been selected as the overall way of separating the different parts of the program.

4.2.1 Platform

As mentioned in the requirements analysis in section 2.3 the tool should be able to run on all of the popular platforms. In order to fulfill this goal Java was chosen as the programming language. It provides the flexibility needed and with a huge amount of open source frameworks it was an obvious choice. The primary toolkit for designing GUI applications in Java is Swing, so this was chosen for the GUI layer.

4.3 Selected similarity algorithms

As mentioned in chapter 3, different algorithms can be used when comparing the similarities among textual content. In order to investigate how different methods perform, the tool should work with several different strategies. The selected algorithms are listed below. A theoretical description for each of the algorithms can be found in chapter 3

- Levenshtein distance
- Textual Fuzzy Similarity
- Term frequency–inverse document frequency

- Ontology based query

Levenshtein distance has been selected because of the simplicity in implementation and understanding. The algorithm has a good performance because the documents don't have to be *Part-of-speech tagged* or use a lexical database to look up related words. The disadvantages is that the algorithm works best with documents of similar lengths. The order of the words is also important and it does not take *Word-sense disambiguation* into account.

Textual fuzzy similarity allows a lot of adjustments. In the standard case the running time is very good, but by integrating the tool with WordNet, a POS-tagger and a sense relater, the algorithm can require a lot of computing time because of the required processing of words in each document. The algorithm is good for testing what impact the different user settings has on the result and running time. The algorithm will be able to give an indicator of whether it is worth spending time processing a document compared to the result obtained. The order of the words in a document does not matter with this algorithm.

Term frequency–inverse document frequency is based on similarity vectors. Instead of comparing the similarity between two documents par wise it uses a *corpus* of document to create vectors representing each document. When comparing the angle between these vectors it is able to determine the similarity. If a document is compared to itself from a set of other documents it will, contrary to the other algorithms, not necessary result in a perfect similarity because the result for each document depends on all the documents in the set. The algorithm is fast, even when working with a lot of documents, but has the disadvantage that is needs a corpus in order to work.

Ontology based query is based on the fact most words have synonyms, hypernyms and hyponyms. An advantage with this algorithm is that is does not require the exact same words in different document. Documents about the same topic can be written using different words that are related to each other. This algorithm takes this into account and will try to find a matching even though the exact same words are not included in both documents. By letting the user decide the weights when using neighbor words it makes the algorithm highly adjustable. A disadvantage is the running time when including a lot of neighbor words. Finding the synonyms, hypernyms and hyponyms can be a time consuming job because the words in the documents needs to be POS-tagged and sense related in order to provide most accuracy. Furthermore a lot of comparisons are needed when calculating the score.

4.4 Processing of documents

Before an algorithm is able to compute a score for some textual content it must be loaded into a data structure and processed. Below is a discussion of the choices for solving the problem of processing a document and afterwards a discussion of the third-party tools selected to be a part of the final program.

4.4.1 Flexibility

A high flexibility is very important when testing possible solutions to a problem that does not seem to have a fixed solution. It is important that the task of processing a document is flexible in the sense that it is easy to extend it or change it and in that way makes it better. This is accomplished by splitting the transformation of going from some textual content to a processed document into several steps. For each step the user can decide which method to use, and it is possible for a developer to implement another solution for a specific step and still take advantage of the other steps. By analyzing the process of going from some textual content to a document object that an algorithm can use, the steps listed below were suggested. The steps should be executed in the same order as they are listed.

1. Loader - Loads data from a data source[1] and turns it into a string.
2. Parser - Takes a string and turns it into a document object by parsing it into sentences with words.
3. POS-tagger - Takes a document object and determins Part-Of-Speech for each word.
4. Sense relater - Takes a document object and find senses for each of the words.
5. Stemmer - Takes a document object and stems all the words.
6. Trimmer - Takes a document object and removes words from it.
7. Includer - Takes a document object and adds words to it.

The process of document going though all the steps is illustrated on figure 4.2.

[1]The data source is usually a file on the hard drive but could actually be a URL, data from another application or whatever source the user prefers.

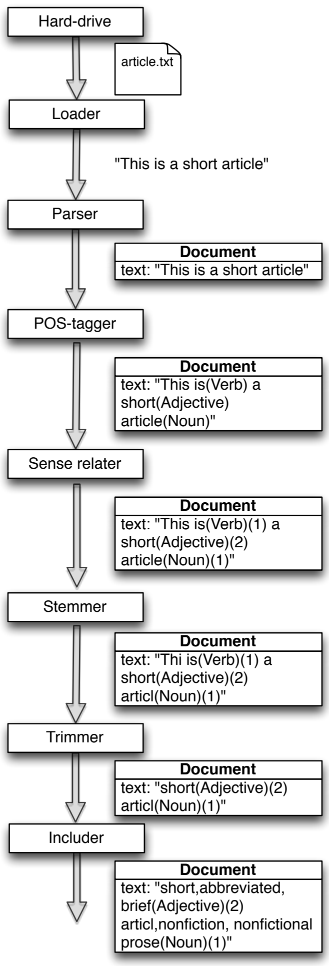

Hard-drive
article.txt
Loader
"This is a short article"
Parser
Document
text: "This is a short article"
POS-tagger
Document
text: "This is(Verb) a short(Adjective) article(Noun)"
Sense relater
Document
text: "This is(Verb)(1) a short(Adjective)(2) article(Noun)(1)"
Stemmer
Document
text: "Thi is(Verb)(1) a short(Adjective)(2) articl(Noun)(1)"
Trimmer
Document
text: "short(Adjective)(2) articl(Noun)(1)"
Includer
Document
text: "short,abbreviated, brief(Adjective)(2) articl,nonfiction, nonfictional prose(Noun)(1)"

Figure 4.2: Illustrates the process of going through all the steps when loading a document.

For each of the steps several different methods could be implemented. For instance stemming can be done using different strategies. By implementing several of these strategies, the user is able to try each of them out and see which effect it has on the end result. Some steps might require a time-consuming algorithm to run, for instance sense relating words. By implementing different methods for sense relating a document, the user can test how fast methods perform in comparison to slow. It should be possible select which steps from the processing of a document that should be performed. For instance it should be possible to skip POS-tagging while still doing sense relating. One should note that by doing so the process of sense relating becomes harder as the POS for the words is unknown. Different algorithms sometimes require different data structures. Using the step model new data structures can be added if a certain algorithm needs it.

In order to make it easy to provide new methods for each step, an interface for each step is provided, which makes it easy for developers to integrate with the system, by creating their own implementations. This is discussed further in chapter 5.

4.4.2 WordNet

The most central tool when doing processing of textual content is a lexical database. WordNet is a highly comprehensive lexical database for the english language. It groups the words into set of synonyms called synsets and holds information about semantic relations between synsets. The current version of WordNet consists of 155.000 words and 117.000 synsets[Wor] from the english language. WordNet categorizes the words into four lexical categories. Noun, verb, adjective and adverb. The part of speech for a word is its lexical category.

WordNet provides an *API* that allows developers to take advantage of the database. Several libraries exist in Java that interfaces with WordNet. MIT [2] provides a stable interface called JWI [3] which supports all newer versions of WordNet. This is important because the newest version of WordNet on the Windows platform is 2.1 while Unix uses 3.0. The framework provides access to all the functionality in an easy to use high-level way[Fin11] and because of that it has been chosen as framework used to connect the program to the WordNet database.

4.4.3 POS-tagger

In order to look up a word in WordNet a POS tag must be specified. Because the same word sometimes belongs to several lexical categories a POS-tagger is needed. It is a software component that determines the POS tag for each word in a document. Several implementations exists that offer a Java library for doing so. Two libraries that are flexible and suitable are listed below.

[2]Massachusetts Institute of Technology
[3]The MIT Java WordNet interface

- Stanford Log-linear Part-Of-Speech Tagger
- Illinois Part of Speech Tagger

Both of the solutions use the *Penn Treebank Tagset*. This tagset is actually more detailed than required because it uses 36 different tags and WordNet only uses 4 categories for the words. After a POS-tagging the tag should be converted into one of the 4 categories WordNet uses and a 5th category for all other words.

Both of the taggers are statistical and they use models that are trained using The Wall Street Journal (WSJ). For the Stanford tagger all of the models provide an accuracy about 97% for the articles it has been trained on and 90% for unknown words[4]. The Illinois tagger provides an accuracy about 97% as well[RZ] for the trained articles.

Because the features of the two POS-taggers were very equal, it was at first decided to pick one of them as a part of the design. The Stanford tagger included several different models to choose from that allowed us to adjust it for our needs to it was decided to go with that one. By testing the required resources for the different models it was discovered that the ones proving the most accuracy would use a large amount of memory. Because of the gain of only 0.35% for the most accurate models, it was decided to go with the 'wsj-0-18-left3words.tagger' model as it used the least amount of resources.

After working with the Stanford POS tagger it was decided to compare the running time of two frameworks in order to check if the Illinois POS Tagger could provide faster results. The two taggers were tested with a 4.5mb txt file[5]. The running time for the two taggers is listed in table 4.1.

Tagger	Running time
Stanford POS Tagger	70.35s
Illinois POS Tagger	12.67s

Table 4.1: Shows the running time for POS tagging a large file

As shown the running time for the Illinois tagger was significantly faster than the Stanford tagger. The memory usage of the tagger was also measured to be marginally lower for the Illinois tagger. Because of this it was decided to

[4]The stats are provided by the README file from the Stanford POS-tagger framework which can be found on appendix D

[5]The Bible was used as the large testing file, it was downloaded from Project Gutenberg at `http://www.gutenberg.org/cache/epub/10/pg10.txt` and consists of 950.457 words

include both of the taggers in the final design thereby allowing the user to switch between them in order to compare the result.

4.4.4 Sense relation

By POS-tagging a document the tool is able to narrow down the number of possible senses of each word dramatically. But each word can still have different senses as discussed in section 3.4. Different strategies can be used in order to narrow it down to one sense. Below is listed two different approaches.

- Assign a random sense to each word.
- Assign the most frequent sense for each word.

In order to test the accuracy of the word sense disambiguation, the result can be compared to a text tagged by humans. An example of a collection of such texts is the *SemCor* Corpus. It consists of about 360.000 word of which about 221.000 are sense tagged. The strategy of randomly assigning a sense from WordNet to each words gives a *F-measure* of 41%[PK] for SemCor. By assigning the most common sense results in a F-measure of 76%[PK] for SemCor. The reason why this strategy works so well is that the way WordNet determines which senses are most common is based on the frequency in SemCor[Mic05].

Of these two strategies it was decided to include the assignment of the most frequent sense in the final design.

It was later decided to include another strategy for solving word sense disambiguation in order to allow the user to try out different settings. The Lesk algorith is another algorithm that solves word sense disambiguation [Ban02]. It is based on the assumption that words that are close to one another in a sentence tend to share a common topic. It works by looking at a word in a sentence and finds the description of the possible senses for it using a lexical database. It then finds the descriptions of the senses for the neighbor words. The score for each combination of sense is then calculated by looking at how many words the descriptions have in common. The sense representing the highest score will be the sense assigned.

In order to include this strategy of sense disambiguation in the design it was decided to include a framework instead of implementing the algorithm. The framework `WordNet-SenseRelate-AllWords`[6] was selected. It is a Perl module

[6]The framework can be downloaded at http://senserelate.sourceforge.net/

that uses the algorithm described above together with WordNet. It takes a sentence as a string and tries to find the sense index in WordNet. It requires a lot computing time in order to determine the senses for a sentence because a lot of scores has to be calculated. On SemCor it gives an F-measure of 59%[PK].

4.4.5 Stemming

Several algorithms depend on finding the same words in different documents. In order to make the probability of a match higher *Word stemming* can be used. Several stemming methods exists in order to stem a word. Two are listed here:

- Lovins stemmer
- Porter stemmer

The Lovins stemmer was the first published stemming algorithm. The stemmer will look at the suffixes of the words and remove the longest one that is in the list of known suffixes while still keeping the stemmed word at a length of minimum 3 letters. The ending may then be transformed using one of the transformations rules. The algorithm uses a list consisting of 294 suffixes and 35 transformation rules[Lov68].

The Porter stemmer uses the fact that the suffixes of a word are often made up of smaller and simpler suffixes. Each word will go through 5 steps. Each step has several rules that it will try to match with the suffix of the word. If a rule is met it will go on to the next step[M.F80].

The Lovins stemmer is faster than the Porter stemmer but uses more memory because it needs to have a list of all the suffixes. It has been decided to include both stemmers in the final design.

Furthermore WordNet will be used as a stemming option by using lemmatization. This works simply by looking the word up in WordNet and assigning the lemma WordNet uses for the word.

4.5 Performance

Loading several documents can be quite time-consuming, especially if all the words has to be processed. In order to make the tool as usable as possible a goal was to make the loading process fast, but still keeping it flexible. Loading several documents at the same time on different threads allowed the tool to take full advantage of a multi-processor computer and gain performance. Furthermore some algorithms can run in parallel as well, as long as the needed data is ready. In order to suit most types of algorithms a *barrier* has been set after the loading of the documents. This means that all documents must have been loaded and processed before the computing for any document will begin. This is done in order to support algorithms which needs knowledge about the corpus in order to compute the score for a document. A disadvantage with this solution is that the required need of space in memory is higher. This is also known as the Space-time tradeoff, where the memory use can be lowered by the cost of computing time and vice versa. This could be improved by using a different loading strategy for each algorithm and using the one that suits the selected algorithm, but this would take away some of the flexibility and make things more complicated. In figure 4.3 it is illustrated how the loading and computing process works. The number of threads should be a fixed number and should not be too large compared to the number of processor available. When a thread is done with a document it should be reused for the next document instead of releasing and creating a new thread in order to avoid wasted time doing so.

In order to improve performance it is important to use suitable data structures that allow interacting with the data in a fast way. To achieve this documents keeps their words in two data structures. A hash table for quick look up of words and arrays for keeping track of the order of the words. The hash table allows to search a document for a specific word in constant time, and arrays makes it fast to go through all the words in a document, while maintaining the order of the words, which is a important feature for some algorithms. Keeping two data structures for the words in each document will of course increase the memory usage of the program. In order to minimize this it should be structured so the value for each key in the hash table contains the index of the word in the array instead of keeping a copy of the same object.

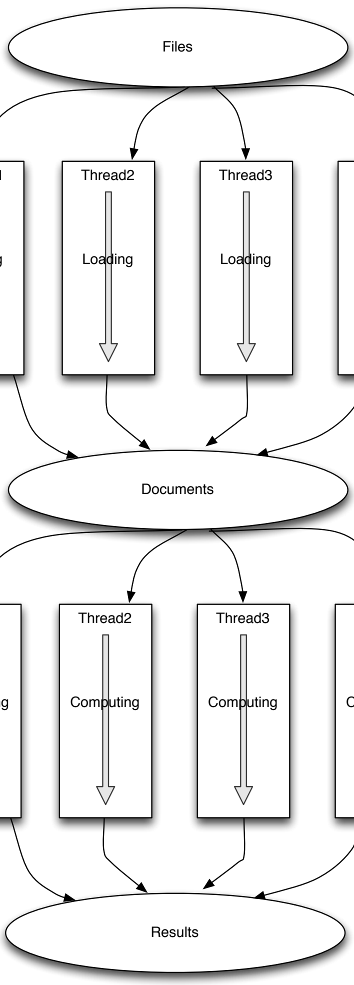


Figure 4.3: Illustrates the process of loading and afterwards computing the results for several documents on different threads.

Chapter 5

Implementation

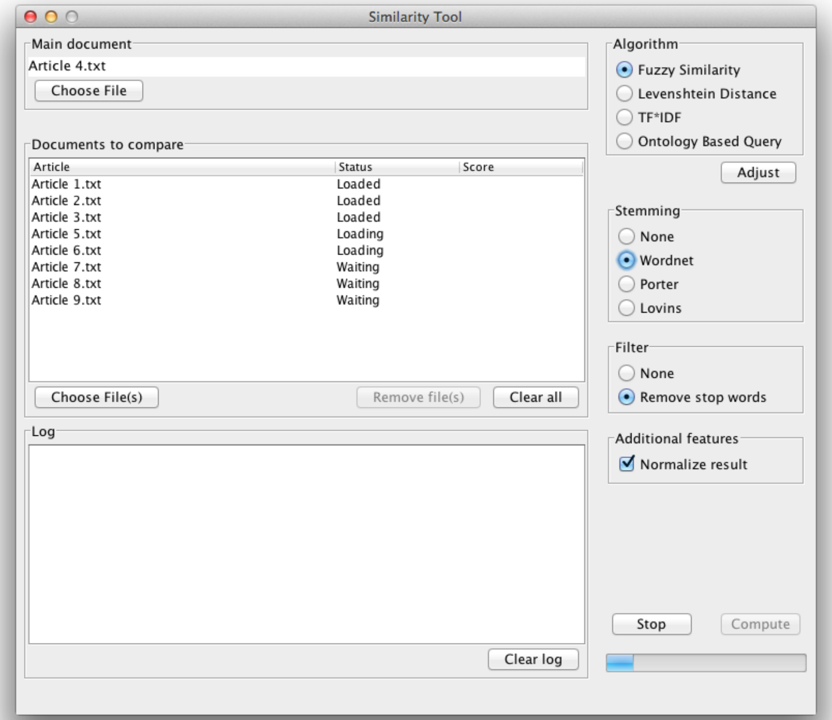
5.1 Chapter outline

Through this chapter the implemented tool will be explained and discussed. The structure of the implementation will be represented by diagrams and it will be discussed how the tool is implemented in order to make it easy to use and easy to extend for a developer. The implemented tests will be discussed and finally the requirements to run the tool will be stated. Through this chapter `typewriter` font will be used for objects, classes, methods, enums and instance variables.

5.2 Similarity Tool

The implemented tool, which is called Similarity Tool, provides a GUI interface for testing different strategies when trying to find textual similarities among a set of documents. The tool works on all major platforms[1] and lets the user adjust several settings before doing a computation. A screenshot of the tool can be seen on figure 5.1

[1] The tool has been tested on Linux (Ubuntu), Mac OS X Lion and Windows 7



Similarity Tool

Main document: Article 4.txt

Documents to compare

Article	Status	Score
Article 1.txt	Loaded	
Article 2.txt	Loaded	
Article 3.txt	Loaded	
Article 5.txt	Loading	
Article 6.txt	Loading	
Article 7.txt	Waiting	
Article 8.txt	Waiting	
Article 9.txt	Waiting	

Algorithm: [✓ Fuzzy Similarity] Levenshtein Distance TF*IDF Ontology Based Query

Stemming: None [✓ Wordnet] Porter Lovins

Filter: None [✓ Remove stop words]

Additional features: [✓ Normalize result]

Figure 5.1: Screenshot of Similarity Tool running on Mac OS X.

The tool lets the user select a document that is the main document of the comparison. Next the user selects the other documents that should be a part of the comparison. The desired algorithm should be selected. For every algorithm it is possible to use stemming and removal of stop words. As an additional feature the user can choose to normalize the result for each document. This will make sure that the result is from 0.0 to 1.0. When the desired setup has been selected the user can click the compute button and the calculations will begin. The user can follow the overall progress using the progress bar in the lower right corner and for each document see the current status. The calculations can be stopped by pressing the stop button. A log will display extra information about what is going on behind the scene.

Instructions on how to install the tool in provided for the supported platforms on appendix E.

5.2.1 Adjustments

Ontology Based Query and Fuzzy Similarity can be adjusted by opening a popup window with various settings. For Fuzzy Similarity the following can be adjusted.

- **POS tagger** - The POS tagger that should be used to tag all the words in each document.
- **Sense relate** - The method that should be used for sense relating.
- **Threshold** - The minimum value that a word pair similarity must exceed in order to be a part of the final result.
- **Hyper/hyponyms** - Whether to include hypernyms and hyponyms as neighbor words as well as the number of layers to include.
- **Synonyms** - Whether to include synonyms as neighbor words.
- **Include words** - Whether to use words that have a POS-tagging only, sense relation only or all words when finding neighbor words.

Ontology Based Query adds some additional settings.

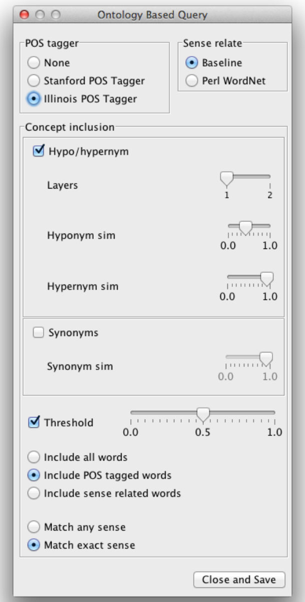
- **Hyponym/hypernym sim** - The weight of the hypernym and hyponym edges of the ontology graph.
- **Synonyms sim** - The weight associated with synonyms found by query expansion.
- **Match** - When finding matches for words among articles; whether to match any sense between two words or use the exact sense only.

A screenshot of the settings panel for the algorithm Ontology Based Query can be seen on figure 5.2 .

5.3 Structure

The tool has been structured into several packages each grouping the classes that concern the same area.[2] The source code can easily be imported into *Eclipse*[3]

[2]See the documentation on appendix H for more details.
[3]See appendix G for information of how to download the source code.



POS tagger
None
Stanford POS Tagger
[✓ Illinois POS Tagger]

Sense relate
[✓ Baseline]
Perl WordNet

Concept inclusion

[✓ Hypo/hypernym]
Layers 1 2
Hyponym sim 0.0 1.0
Hypernym sim 0.0 1.0

Synonyms
Synonym sim 0.0 1.0

[✓ Threshold] 0.0 0.5 1.0

Include all words
[✓ Include POS tagged words]
Include sense related words

Match any sense
[✓ Match exact sense]

Figure 5.2: Screenshot of the settings panel for Ontology Based Query.

or another IDE.

When the `main` method is called it will check the system to see if the required software is installed. For more details see section 5.6. If that is the case the tool will be loaded by a `MainController` object which will instantiate the GUI by allocating a `MainFrame` object and the model layer with a `JPWordTool` object. This is shown on figure F.1. The tool is structured so it has to go through all of the steps from figure 4.3. In order to support skipping a step dummy implementations have been implemented for the steps that can be skipped. A dummy implementation will not change anything and just returns the input.

A document is represented by a `JPDocument` object. This contains an array of `JPSentences` which contain arrays of `JPWords`. This is illustrated in figure F.2.

As a user decides to run an algorithm on a set of documents a `JPConfiguration` object will be created. This object provides all the information about the current setup, such as the chosen algorithm, files and settings. The associations are shown in the class diagram on figure F.2. Before the algorithm begins running, the selected files must be loaded and analyzed. This is done by the `run` method of `JPWordTool` and it is illustrated in the sequence diagram in figure F.9. It is worth noting that several documents will be loaded in parallel for better performance. Afterwards the `JPWordTool` will call the `compute` method of the selected algorithm. This will regularly notify the UI about status changes. A sequence diagram for `FuzzySimilarityAlgorithm` is shown on figure F.10. Other algorithms uses the same pattern.

To keep the interface smooth and responsive the main thread must not be blocked at any time. By always running heavy work on a background thread the tool seems more user-friendly as it will always responsive to interaction. One should keep in mind that updates to the UI always should take place on the main thread because Swing is not thread-safe.

5.4 Tests

Functional tests have been implemented in order lower the amount of bugs and to test the different parts of the program. This has been accomplished using a testing framework called TestNG. It is a Java framework that provides the possibility of writing test cases for methods in a structured way. Each test uses a data provider to populate it with data and a test will either pass or fail. Tests for each of the major parts of the program have been implemented and will be explained in section 5.5. TestNG is able to generate reports for the tests, but only in a very poor format. Instead ReportNG[4] is used to generate the report. They can be found for all the tests on the website `http://www.student.dtu.dk/~s093263/similaritytool/`.

To really test the application a huge database of articles about different topics were needed. Such a database was downloaded from `http://mydatamaster.com/free-downloads/`. It contains a `.sql` file with 50.000 articles about all sorts of topics. This file was imported on a local mysql server and a small Java application was developed to convert each entry in the database into a `.txt` file that could be used in the tool.

[4]A Java framework that works together with TestNG that can generate a html overview of the test results.

5.5 Interfaces

Interfaces and abstract classes are provided in order to make it easy to extend the tool with new algorithms, stemmers etc. When a developer wants to extend the application with a class he has developed that implements the provided interface all he has to do is add a `enum` representing his additions to `JPConfiguration` and add a `JRadioButton` to the view. Below descriptions for the different abstract classes are explained and the already implemented classes for each step are discussed.

5.5.1 JPAbstractAlgorithm

The abstract class `JPAbstractAlgorithm` provides methods for easily implementing an algorithm that takes full advantage of several threads and makes it possible to notify the user about status updates. The developer should simply subclass the abstract class in order to get all the provided functionality. It provides methods for updating a delegate about progress during the calculations using `JPAlgorithmProgressDelegate` and `JPProgress`. This would typically be used to inform the user of what is going on and how far the calculations are for each document the algorithm uses.

In order to take full advantage of the CPUs in a computer, the algorithm must run in parallel. It is the job of the developer to decide which part of the algorithm that is suited to run in parallel. If the algorithm can calculate the score for a pair of documents without being dependent on other scores it is an obvious choice to let this part run on different threads for a performance gain. The abstract class allows a subclass to submit pieces of code that should run in parallel. The order in which the pieces of code will be executed is managed by a FIFO[5] queue. During the calculations threads will automatically be reused as they finish their current work.

A subclass should implement the `compute` method that provides all the necessary objects to do a computation. This includes the `JPDocumens`'s that has been selected by the user and a `Runnable` that should run after all the calculations are done. Before running the `Runnable` all documents should have updated their `score`. This can either be a normalized score or the actual score from the algorithm. This depends on the value of the boolean `normalizeScore`.

The user should be able to stop a running algorithm. This requires a small implementation detail for the developer. If the algorithm does heavy work for a

[5] First in, first out.

long time the thread will not be interrupted. Instead it should once in a while check if this has happened by calling the method `threadUpdate`. This will make sure to interrupt the thread if needed.

The tool implements 4 different algorithms. Each of those are a subclass of `JPAbstractAlgorithm` as shown on the class diagram in figure F.3.

The results of each algorithm are revealed in chapter 6.

5.5.2 JPAbstractStemmer

By subclassing `JPAbstractStemmer` a new stemmer can be implemented. The method `stem(JPDocument document)` must be implemented and a `JPDocument` where all the words have been stemmed should be returned. The `stem` method will automatically be called on a background thread by `JPWordTool` for all the selected `JPDocuments`. Several documents can be stemmed at the same time on different threads so it is important that the stemmer is thread-safe and only calls thread-safe methods.

The tools implement 3 different stemmers[6] and a dummy stemmer as shown in figure F.4. The two actual stemmers have been tested for different edge cases and the result of a stemming has been compared to the expected result for different words. The implementation of tests can be found in `JPStemmerLovinsTest` and `JPStemmerPorterTest`.

5.5.3 JPAbstractPOSTagger

POS-taggers should subclass `JPAbstractPOSTagger` and implement the method `tag(JPDocument document)`. After a `JPSentence` has been tagged the boolean called `isPOSTagged` should be set to `true`. Furthermore each `JPWord` should have its `tag` string set to the POS-tag according to the Penn Treebank tagset. Because WordNet only uses 4 categories for the words a `JPWord` should also be mapped to a category that can be used with WordNet. This is done by the `JPWordPOS`, which is an `enum` that represents the 4 categories plus a category extra for words that do not fit in any of the first categories. Below is listed the values of the `enum`.

- JPWordPOSUnknown

[6]The WordNet stemmer actually finds the lemma of a word rather than the stem.

Tagger	Success percent	Computing time
`StanfordPOSTaggerManagerTest`	92.67%	41.3s
`IllinoisPOSTaggerManagerTest`	90.01%	7.2s

Table 5.1: Shows the performance of the two POS-taggers.

- JPWordPOSNoun
- JPWordPOSVerb
- JPWordPOSAdjective
- JPWordPOSAdverb

Besides the dummy POS-tagger, that does not change anything, the tool includes two taggers. The classes `JPPOSTaggerStanford` and `JPPOSTaggerIllinois` hold the respective POS taggers. The Stanford tagger uses a singleton[7] object called `StanfordPOSTaggerManager`. It works as a wrapper class for the Stanford POS-Tagger Java framework and provides only the necessary features. The Illinois tagger uses the same idea with the class `IllinoisPOSTaggerManager` that works as a wrapper class for the Illinois POS-Tagger Java framework.

In order to test the implemented POS-taggers, the classes `StanfordPOSTaggerManagerTest` and `IllinoisPOSTaggerManagerTest` have been implemented. The focus of the tests has mostly been to measure their performance by running them on a collection of articles that has been POS-tagged correctly by humans. That way information about both the execution time and accuracy is obtained. To do so the *SemCor corpus* has been used. The Java framework *JSemCor*[Fin] makes it easy to load the articles into Java objects that can be used to generate `JPDocuments`. Table 5.1 shows the result obtained by comparing the tagging for each word to the tagging in SemCor. It is worth noting that we only care if tagger is able to place each word in the correct category among the 4 categories WordNet uses. The test does not compare the found tag with the extensive Penn Treebank Tagset but rather ensures that it falls into one of the 5 categories defined above.

The accuracy of the Stanford POS-tagger is slightly better but quite a lot slower.

[7]A design pattern that restricts the number of instantiation of a class to one and makes it possible to access this object in a static way, so it is easily accessible from all classes.

5.5.4 JPAbstractSenseRelate

Classes that do sense relation on documents should subclass `JPAbstractSenseRelate`. A `JPDocument` has a boolean indicating if it has been sense related. If this boolean is true, every `JPWord` should have its `senseIndex` set. This index corresponds to the sense index in WordNet. Subclasses of `JPAbstractSenseRelate` should implement the method `senseRelate(JPDocument document)` and return a `JPDocument` where the `senseIndex` has been set for each word.

The object `JPSenseRelateBaseline` assigns the most frequent sense to all the words. This is done by using a sense of the value 0 for all the words that are in WordNet, because the senses of words in WordNet are sorted in descending order according to their frequency.

The tool comes with an option for doing more advanced sense relation. This is implemented in the class `JPSenseRelateWordNet`. The actual calculations are done using a Perl module that must be installed on the computer. A wrapper for the module has been implemented in Perl for easy access. The source code for this can be found on appendix I. It works by taking a sentence as a string and returns a string with a senseIndex for each word. By using the optional parameter `"-j"` JSON[8] will be returned instead. An example of a call could be:

```
> perl SenseRelate.pl -j "A short article"
```

The result of this call is shown in figure 5.3. The result is then parsed to a object called `JPSenseRelation` by `JPSenseRelateWordNet` and the index is assigned to each `JPWord` for later use when looking up the word in WordNet.

The Perl module also supports sentences that have been POS-tagged beforehand. The Penn Treebank Tagset[9] will be used as the tags. The following is an example of a call using a POS-tagged sentence.

```
> perl SenseRelate.pl -j -t "A/DT short/JJ article/NN"
```

Because the calculations for the implemented `JPSenseRelateWordNet` are very heavy it will use several threads in order to gain performance. It does this by finding senses for several sentences from a document at the same time. Also a cache is used to cache the results for each sentence. For this the object `JPCache` is used. It provides methods for *key-value pair* caching of objects, that can be stored both in memory and on disk. It is possible to clear the cache using the menubar in top of the program.

[8] JavaScript Object Notation. Used for easy human-readable data exchange.

[9] The tags can be found on appendix C

```
{
    "success": 1,
    "relations": [
        {
            "word": "A",
            "newWord": "A",
            "tag": "n",
            "senseIndex": 1
        },
        {
            "word": "short",
            "newWord": "short",
            "tag": "a",
            "senseIndex": 1
        },
        {
            "word": "article",
            "newWord": "article",
            "tag": "n",
            "senseIndex": 1
        }
    ]
}
```

Figure 5.3: The JSON result of a call to SenseRelate.pl.

Sense relater	Success percent	Computing time
`JPSenseRelateWordNetTest`	51.92%	27064.07s
`JPSenseRelateBaselineTest`	66.72%	0.21s

Table 5.2: Shows the performance of the two word sense disambiguation methods.

A class diagram of the implemented sense relaters can be found on figure F.6.

To test the implementations the SemCor corpus is again used to measure the accuracy and performance. The tests are implemented in `JPSenseRelateWordNetTest` and `JPSenseRelateBaselineTest`. This is done by comparing the result of each of the sense relaters with the result of the SemCor corpus. The result is shown in table 5.2.

It shows that the simple approach by picking the most frequent sense gives the best result. A thing to keep in mind is that WordNet is structured so it list the senses according how frequent they are in the SemCor corpus, which of course will improve the result for the baseline approach.

5.5.5 JPAbstractTrimmer

The `JPAbstractTrimmer` provides the option of trimming a document. This can have several applications. The implemented trimmer called `JPTrimmerStopWords` removes words from a document if they are in the list of stop words[10].

Trimmers should implement the `trim(JPDocument document)` method and return a `JPDocument`. A class diagram for `JPAbstractTrimmer` is available at figure F.7.

5.5.6 JPAbstractInclude

A subclass of `JPAbstractInclude` has many applications as well. Other than the dummy implementation, the tool includes one class for expanding a document to include related words, namely `JPIncludeNeighbourWords` which makes it possible to find synonyms, hypernyms and hyponyms for every word in a article. This is done using the object `NeighbourWordsFactory` that uses WordNet

[10]The list consists of 173 common words from the english language. The list can be found at `http://www.ranks.nl/resources/stopwords.html`.

to find the neighbor words for a word in an efficient way using a `JPCache`. For looking up neighbor words the `WordNetManager` singleton object is used which wraps around the `JWI` Java framework that interfaces with WordNet. It provides the necessary functionality of WordNet needed for the tool. The neighbor words are assigned to each `JPWord` in a array. For each neighbor word a score will be calculated based on the sim settings for the algorithm. The array is sorted in ascending order according to these scores.

The `NeighbourWordsFactory` was tested using the class `NeighbourWordsFactoryTest` that can be used for testing if the expected synonyms, hypernyms and hyponyms were found.

5.6 Requirements

The tool has certain requirements for the system that must be met. They are listed below.

- Java 1.5 or newer.
- WordNet 2.1 for Windows and WordNet 3.0 for Linux and Mac OS X.
- Perl[11].
- Make[12].

On all platforms the `WNHOME` environment variable must be set and the `bin` folder must be added to the PATH in order for the tool to use WordNet. Furthermore the `bin` folder for Perl must be added to the PATH as well. This is explained in the installation guide on appendix E.

In order to use the Lesk algorithm for sense disambiguation the following Perl modules must be installed.

- Digest-SHA1.
- Text-Similarity.
- WordNet-QueryData.

[11]Programming language that runs on all major platforms
[12]Utility that can be used to build libraries and install software.

- WordNet-Similarity.
- WordNet-SenseRelate-AllWords.

All of the requirements will be installed when following the installation guide on appendix E. There is a guide for each of the 3 supported platforms.

Chapter 6

Results

6.1 Chapter outline

In this chapter the findings that have been made with the tool developed as a part of the thesis will be presented. The results of the different algorithms as well as how they have been held up against human judgement will be covered here.

6.2 Human judgements

The main goal when doing textual similarities based on the semantic between documents is to get a result close to what humans perceive as textual similarity. There is no definite answer of what textual similarity really is, hence why human judgement must be considered truth[1]. Upon evaluation, algorithms that resemble human judgement the best are those that solve the textual similarity problem most correctly.

[1]The evaluation of similarity has to be made by a test sample of large enough size, to ensure that the human evaluation is actually a representation of the general consensus between people and not just the opinion of a few.

6.2.1 Experiment

In order to compare the algorithms to human judgements a small experiment was made. We selected 9 articles. All of them were about 200 words, which we considered a reasonable length to ask test subjects to read and also because longer articles causes a lot a of noise while very short articles may hold too few words of importance. The articles covered several topics, some of them more related than others. We asked a group of people to read all the articles and for each pair of articles give a score from 0-100 depending how much they agreed on the following questions:

- Do the articles concern the same topic?
- Do the articles come to the same conclusion on a topic? (If they indeed do concern the same topic)

Where 0 means, that they have nothing in common and 100 means that they are identical. Question 1 was the interesting one and the second question was asked to make sure that participant understood the difference between the questions.

The questionnaire can be found on appendix J.

The results were normalized to lie a a scale from 0.0-1.0, and then compared to the results of the algorithms. To calculate how well an algorithm performed, the correlation coefficient was calculated for the set of points (x_i, y_i) where x_i is the human similarity score for document i and y_i is the algorithm's similarity score for document i. The correlation coefficient, r, is a number in the interval [-1,1]. A correlation on ± 1 means that the data set points lie on a straight line meaning strong linear relation. Normally a correlation over $|0.8|$ is considered strong and a correlation under $|0.5|$ is considered weak. To calculate the correlation coefficient equation 6.1 from [JFM00] is used.

$$r_{xy} = \frac{\sum_{i=1}^{n}(x_i - x)(y_i - y)}{\sqrt{\sum_{i=1}^{n}(x_i - x)^2 \sum_{i=1}^{n}(y_i - y)^2}} \tag{6.1}$$

The average similarity score assigned by the 27 people in the questionnaire, is listed in table 6.1.

Article no.	1	2	3	4	5	6	7	8	9
1	100	-	-	-	-	-	-	-	-
2	1	100	-	-	-	-	-	-	-
3	1	40	100	-	-	-	-	-	-
4	4	4	3	100	-	-	-	-	-
5	0	8	63	5	100	-	-	-	-
6	2	39	39	3	6	100	-	-	-
7	0	0	0	1	1	1	100	-	-
8	2	3	3	83	5	5	0	100	-
9	64	1	0	0	0	1	0	0	100

Figure 6.1: The result of human evaluations.

6.3 Correlations

Based on the questionnaire and the results we have found using the various algorithms with different settings, we are going to calculate the correlation between human judgements and algorithm result. As the amount of combinations in settings for some of the algorithms is very high, we have only included the most interesting ones.
To each algorithm, the best settings, based on the correlation, will be used to make queries on a larger database with 1200 articles on various topics.

6.3.1 Levenshtein distance

Figure 6.2 shows the correlations between human judgements and the levenshtein distance measure with different settings. The result of the levenshtein distance has been normalized such that identical articles have a score of 1 while articles with nothing in common have score 0. The first thing to note is that all scores are in the interval [0.0,0.1], meaning that the articles generally have very little in common, i.e. the edit distance is large. The levenshtein distance generally has a very bad correlation with the human judgements, but to our big surprise, when stop words are removed, the correlation is increased significantly. When using the WordNet stemmer along with stop word removal, we got $r = 0.78$. Of course stop word removal means that there are quite a lot less words that need to be matched, meaning that matches have a higher influence

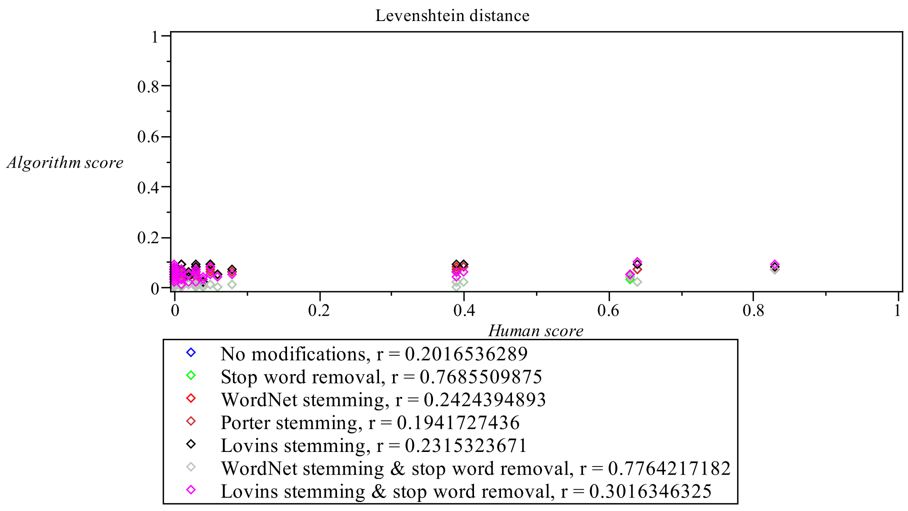


Figure 6.2: The correlation between human scores and the Levenshtein distance algorithm.

on the final score of the algorithm.
Levenshtein distance is an algorithm that rewards text for being similar in structure so one could say that the removal of stop words defeats the purpose of the algorithm, as it changes the structure of the texts.
We got the best result using the following settings:

- Stop word removal
- WordNet stemming

6.3.1.1 Database query

Using the WordNet stemmer and stop word removal:

Query *A hacker inside your computer!* it.[2]

[2] http://www.articlesfactory.com/articles/site-security/a-hacker-inside-your-computer.html

Best match *A basic introduction to Spyware* computer.[3]

Second best match *A beginner's guide to affiliate marketing* [4]

Worst match - *A trickle of electricity* [5]

Running time 66615 ms.

6.3.2 TF*IDF

Figure 6.3 shows the correlations between human judgements and the term frequency–inverse document frequency distance measure with different settings. Values of the *TF*IDF* lie in the interval [0,1]. The similarity score between two documents rely on the rest of the corpus, which is why, identical articles don't necessarily have a similarity score of 1. As seen on figure 6.3, the *TF*IDF*

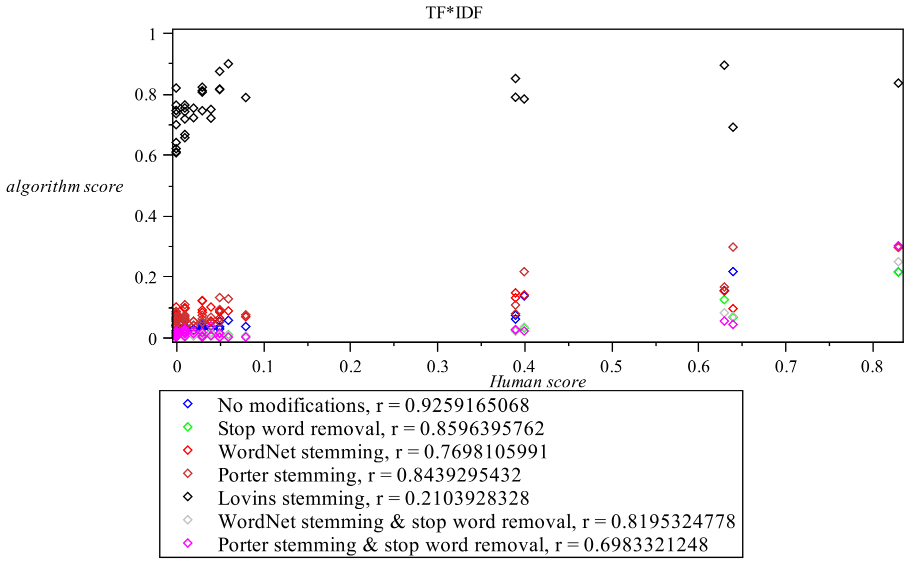


Figure 6.3: The correlation between human scores and the TFIDF algorithm.

generally produces an answer of very high quality. The best correlation was found using the algorithm with no modifications at all, which possibly has to

[3]http://www.articles3k.com/article/366/3775/A_Basic_Introduction_To_Spyware/
[4]http://www.dollarman.com/earnmoney/a-beginner-s-guide-to-affiliate-marketing.html
[5]http://www.simplysearch4it.com/article/1810.html

do with the fact, that the algorithm reflects how important each word is to documents, meaning that stop words are already given very little influence. Stemming is also a factor that makes words "less unique", in the sense, that when stemming the words in the documents, the probability of finding identical words in documents gets higher, decreasing the importance of these words.

6.3.2.1 Database query

Using no modifications on the algorithm:

Query *A hacker inside your computer*

Best match *Access your PC from the road* [6]

Second best match *A beginner's guide to avoiding viruses* [7]

Worst match - *A chicken recipe for every occasion* [8]

Running time 12419 ms.

6.3.3 Textual Fuzzy Similarity

Figure 6.4 shows the correlations between human judgements and the term textual fuzzy similarity algorithm with different settings. Values of the fuzzy similarity lie in the interval [0,1]. The amount of combinations in settings is very high for this algorithm, so we have tried to include results that cover all the different settings.

[6] http://www.albaspectrum.com/Articles2/Misc1/02820.html
[7] http://www.profsr.com/articles/avoidvirus.html
[8] http://www.articles3k.com/article/411/26356/A_Chicken_Recipe_for_Every_Occasion/

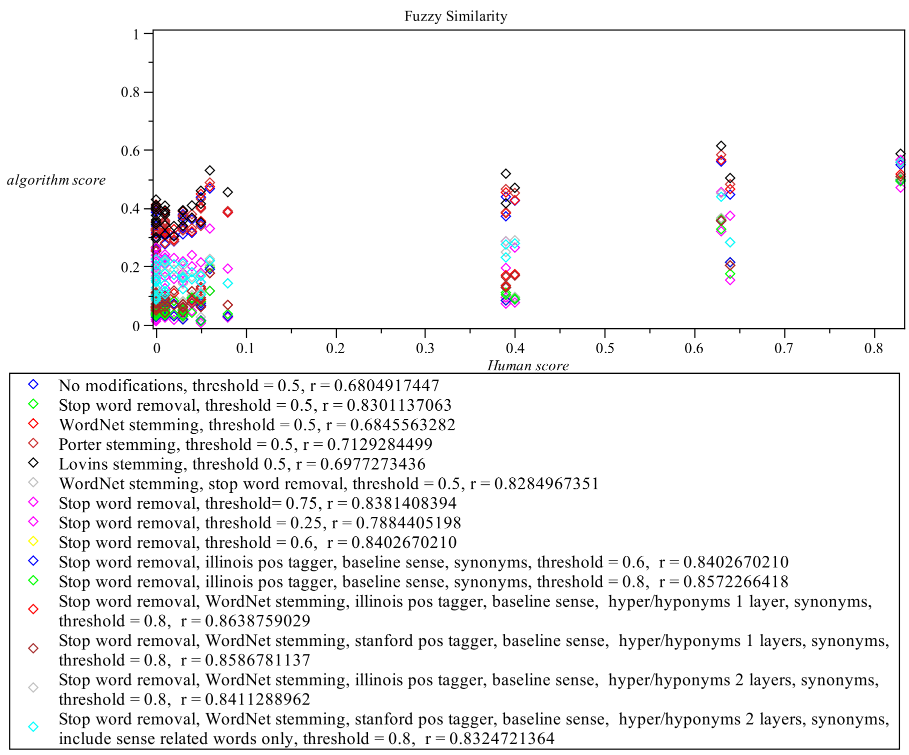


Figure 6.4: The correlation between human scores and the Fuzzy Similarity algorithm.

From figure 6.4 it follows that stop word removal improves the correlation. In this algorithm, each word in a document is seen as equally important, which means that stop words infer more noise in the query, which could explain why stop word removal improves the overall result. Stemming increases the correlation slightly.

We found out that without query expansion, a threshold between 0.6-0.8 gave the best result, which means that pairs of words generally need a rather high similarity score to actually be considered semantically similar. Of course a low threshold would infer a lot of noise. We were only able to improve the correlation slightly by expanding the query to include synonyms, hyponyms and hypernyms, to words in the query. When expanding the query, the threshold had to be increased, because the increase in pairs of words, means we should be more demanding with the similarity measure between words. A reason why the query expansion does not help increase the correlation significantly, might be the fact, that words found by expansion are rated the same as words in the actual documents. We got the best result using the following settings:

- Stop word removal
- WordNet stemming
- The Stanford POS tagger
- Baseline sense relation
- 1 layer of hyper/hyponyms
- Synonym inclusion
- A threshold of 0.8

6.3.3.1 Database query

Using stop word removal, WordNet stemming, illinois pos tagger, baseline senserelate, 1 layer hyper/hyponyms, synonyms and threshold of 0.8:

Query *A hacker inside your computer!*

Best match *A word on Comic Book Pricing* [9]

Second best match *Accelerating Your Downlines* [10]

Worst match - "*About Modern Snowshoes*" [11]

Running time 803522 ms.

6.3.4 Ontology Based Query

Figure 6.5 shows the correlations between human judgements and the ontology based query algorithm with different settings. The result of the algorithm is normalized to lie in the interval [0,1]. The amount of combinations in settings is very high for this algorithm, so we have tried to include results that cover all the different settings.

[9] http://ezinearticles.com/?A-Word-on-Comic-Book-Pricing&id=29033
[10] http://www.articles3k.com/article/342/139659/Accelerating_Your_Downlines/
[11] http://www.directorylistings.info/article/About+Modern+Snowshoes.html

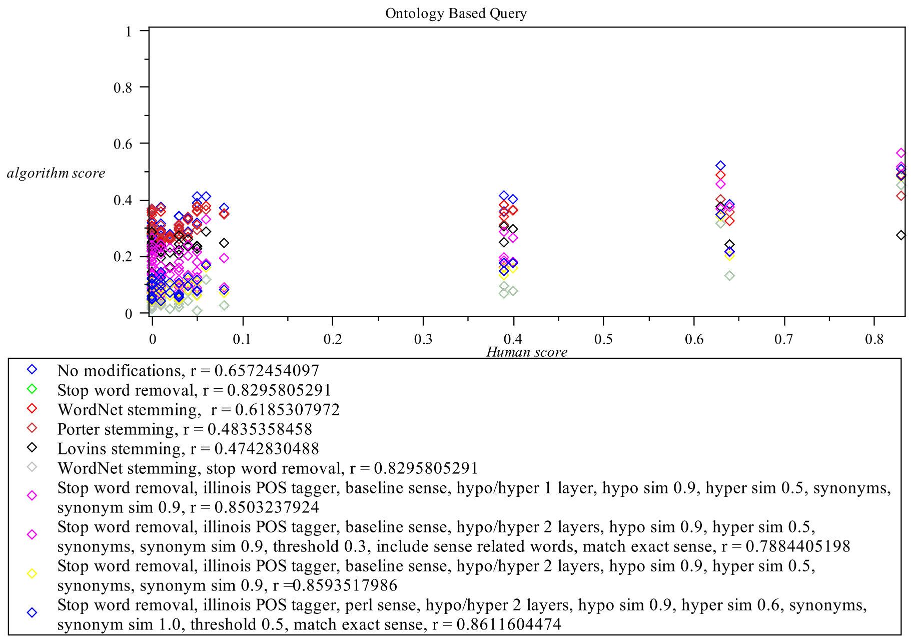


Figure 6.5: The correlation between human scores and the Ontology Based Query algorithm.

Like textual fuzzy similarity, stop word removal significantly improves the correlation in this algorithm, because stop words are common to almost any text and therefore generate noise. Stemming did not improve the correlation which we find peculiar, and the only explanation we can think of, is that the stemming makes the words "less unique", as mentioned earlier.
By expanding the query, the result was slightly increased using different values for the similarity edges for synonyms, hypernyms and hyponyms. A threshold also helped discard irrelevant relations between words. We found out that expanding the query by two layers instead of one only slightly improved the overall results.
The best result was achieved using:

- Stop word removal
- The Illinois POS tagger
- Perl WordNet sense relation
- 2 layers of hyper/hyponyms with a hyponym edge score of 0.9 and hypernym edge score of 0.6

- Inclusion of synonyms with edge score 1 (meaning synonyms are considered as important as words in the query).
- A threshold of 0.5
- Word matching on exact words only.

What this tells us, is that we can slightly improve the result by having knowledge about the words in a document. Knowing the sense of a word, and collecting related words means that we can do a match on words that are not identical, but are related enough for a person to see them as semantically similar.

6.3.4.1 Database query

We were not able to use the Perl WordNet sense relater on the database query, because sense disambiguation on 1200 articles would take a lot of time. We used the settings mentioned in the previous section, but with baseline senserelation instead of Wordnet sense relation, because it gives almost the same result and runs much faster:

Query "*A hacker inside your computer!*"

Best match "*A beginners guide to avoiding viruses.*"

Second best match "*Access your PC from the road.*"

Worst match - "*ABC's of Becoming an effective teen*" recipes.[12]

Running time 50636 ms.

6.4 Asymptotic running times

To see how the algorithms scale, we tried running the algorithms with their optimal settings, on documents of varying length. The documents that were used were `Article 1` and `Article 2` from the questionnaire, concatenated with

[12] http://www.magnet4web.com/content/motivational/abc-s-of-becoming-an-effective-teen.php

themselves, 32, 64 and 128 times, so we could measure the increase in running time when doubling the length of the two documents for comparison.

Levenshtein Distance

Article 1	Article 2	Approximate running time (ms)
x32	x32	800
x64	x64	2200
x128	x128	8200

Showing that the running time is proportional to the square of the length of the documents.

Term Frequency-Inverse Document Frequency

Article 1	Article 2	Approximate running time (ms)
x32	x32	450
x64	x64	850
x128	x128	1600

Showing that the running time is linearly proportional to the size of the documents.

Textual Fuzzy Similarity

Article 1	Article 2	Approximate running time (ms)
x32	x32	238500
x64	x64	961700
x128	x128	3622600

Showing that the running time is proportional to the square of the length of the documents.

Ontology Based Query

Article 1	Article 2	Approximate running time (ms)
x32	x32	1000
x64	x64	2000
x128	x128	4000

Showing that the running time is linearly proportional to the size of the documents.

Chapter 7

Conclusion

7.1 Chapter Outline

Here we will be making a discussion of the different algorithms, their advantages and disadvantages and use this to give a conclusion based on the obtained results. We will talk about the goals we set ourselves from the beginning and which of these we have been able to achieve. Lastly we will give suggestions to future extensions of the tool developed as part of the thesis.

7.2 Discussion of algorithms

Evaluating the four algorithms has shown that each algorithm has its own advantages and disadvantages, and that it is hard to say which algorithm worked best.
The time and space complexity analysis are done on the scenario where two documents are being compared. In the following runtime analysis we will use different variables: N the number of words in the corpus, U the number of unique words in the corpus, n the number of words in the query, u the number of unique words in the query, m the number of words in the document for comparison, v the number of unique words in the document for comparison.

7.2.1 Levenshtein Distance

The Levenshtein Distance was the simplest algorithm that was implemented, which was also reflected in the result. Without modifications, the result was not satisfiable, but this was to be expected, as this algorithm does not the use the *bag of words* principal, and therefore relies on the same words appearing in two documents in the same order, which is unlikely to happen. It was the only of the algorithms that credited documents for being similar in expression, a feature that is interesting, but did not seem to affect the result of the algorithm very much. When removing stop words as a part of the preprocessing, the resulting correlation between algorithm score and human judgement improved significantly, which follows intuition because stop words are regarded as noise.
The time complexity of the algorithm is $O(n \cdot m)$. The space complexity could be reduced to $O(n)$ by only keeping the needed rows of the calculation matrix in memory.

7.2.2 TF*IDF

To our big surprise the unmodified version of this algorithm gave the best result of all our tests. The automatic accounting for stop words helped to improve the result more than removing the stop words. The fact that this algorithm does not rely on an external knowledge base is a big plus, but on the other hand, it relies on having a corpus of several documents to determine the importance of each word in proportion to the corpus, and is therefore not good for comparing two documents only. Keeping knowledge about word occurrences across the whole corpus is a very memory consuming task, which is also an issue. The *bag of words* principal is used, meaning that the expression of the text is lost.
The query time is linear in the size of unique words in the corpus, i.e. $O(U)$ and preprocessing time is $O(N)$. The space complexity is $O(U)$.

7.2.3 Textual Fuzzy Similarity

The fuzzy similarity is interesting, because it does not categorize pairs of words as either "similar" or "not similar". With this algorithm, the similarity score was based on the visual similarity between the words, and the idea was that words that look alike, with some probability are semantically related. This means that a similarity score between words can be calculated without the use of a knowledge base, which means that the algorithm is usable on any natural language. Removing stop words improved the result a lot, again because they

infer noise on the result, rather than improve it. The algorithm works with query expansion, however this only improves the result marginally, and while it was possible to achieve a very good result without query expansion, the resulting slowing in runtime means that it is a matter of preferences whether this should be used or not. Like in *TF*IDF*, the *bag of words* model means that word order and grammar are lost.
The time complexity is proportional to the square of the words in the documents for comparison, $O(n+m+u \cdot v)$ because all unique pairs words need to computed in the worst case scenario. We found that this algorithm scales badly and gives running times that are not fit for real time applications.
The space complexity is linearly proportional to the number of unique words across the documents, $O(u+v)$. Alternatively, with a preprocessing time of $O(n+m+u \cdot v)$, the query time can be reduced to $O(u)$.

7.2.4 Ontology Based Query

The last algorithm we implemented was the ontology based algorithm, which stood out from the previous algorithms because it relies on an external knowledge base. This dependency means that the algorithm only works on natural languages where such a knowledge base is available. On the other hand, the advantage of this algorithm is that it can expand the query to look beyond what is explicitly stated in a document, making it more "intelligent" than the other algorithms. We were able to achieve an overall good result with this algorithm, but were limited to only allow for 2 layer concept inclusion, because of the massive increase in terms for each level. Like the fuzzy similarity algorithm, the inclusion of more concepts slightly increased the correlation, but of course increased running time and space usage. Again stop word removal helped reduce the amount of noise in the texts.
The time complexity of the algorithm can be reduced to $O(u_+)$ where u_+ is the size of the set of unique words in the query and their related words found by concept inclusion. This requires a preprocessing time of $O(n+m)$. The space complexity is $O(u+v)$.

7.2.5 Roundup

After comparing the four implemented algorithms it is still really hard to say which algorithm is best for determining textual similarities. With different modifications we got all algorithms to deliver a satisfiable result. We found out that the complex methods give marginally better results, keeping in mind that *TF*IDF* relies on a corpus and is not ideal for comparison of two documents.

It turned out that the more simple preprocessing operations on documents were those that improved the result the most, such as stop word removal, while the more time-demanding and complex operations only slightly improved the result. For that reason, the complex operations might be scrapped in real time applications. On the contrary, in scenarios where the accuracy is more important than running time, the inclusion of complex operations can help achieve higher accuracy.

7.3 Achieved goals

In the requirements analysis in section 2.3 several goals were stated. Below is a discussing of the functional goals.

- **Loading documents, stemming, removing stop words**. The tools supports loading plain text document and stemming each word using 3 different stemmers. Stop words can also be removed by looking in a list of 173 words.
- **Integration with lexical database**. Integration with WordNet has been made. It provides an extensive database of words from the english language.
- **Integration with a POS tagger**. Two POS taggers has been added to the tools and the user can select which tagger to use.
- **Contain 3 or more algorithms**. The tools allows the user to select between 4 algorithms.
- **Provide results for each strategy**. For every strategy used when comparing documents, a score is provided for every document. In chapter 6 it was shown how these scores can be used to compare the algorithms.
- **It should be possible to tweak several parameters**. For each algorithm it is possible to adjust several settings allowing the user to find the best settings for a specific algorithm. This was also discussed in 6

All of the functional requirements have been fulfilled. Below is a discussion of the nonfunctional requirements.

- **GUI**. A Swing GUI is provided for easily selecting the desired settings.

- **Platforms**. The tool has been implemented in Java which allows it to run on all major platforms.
- **Performance**. Parallel and concurrent programming has been used when loading and calculating in order to make the tool feel fast.
- **Reliability**. The different parts of the tool has been tested in order to get rid of bugs and make the tool reliable.

We were able to fulfill all of the nonfunctional requirements as well.

7.4 Further development

In order to improve the tool new features can be implemented. Below is listed some suggestions for extensions of the current version.

- **More algorithms**. To make the tool more comprehensive more algorithms should be added, which could take advantage of the different features of the tool.
- **Support for more languages**. At the moment only english is supported because the tool uses WordNet which only works with the english language. WordNet exists for other languages and it would be possible to integrate with those without too much work. It could also be interesting if the tool were able to find similar content across different languages. In such a case some kind of translation mechanism had to be build in to the tool.
- **Support for more file formats**. The tool only supports plain text documents at the moment. Support for more text file formats should be easy as the ability to support several formats are build right into the architecture of tool.
- **Interface with other applications**. By allowing another application to be the data source instead of the hard drive, it would allow integration with existing system and make it easy to compare documents directly from another program.
- **Visual results**. To easier understand the results of the different algorithms a way to show it in a visual way could be implemented. It would allow the user to get a better overview of how each algorithm performs.

- **Comparison of algorithms**. A way of comparing the different algorithms could be implemented making it easier to see which algorithms works best on which kind of input data.
- **Lazy loading of documents**. The tool loads all the documents needed into memory. In order to keep memory usage lower, a system, that would only load the documents that are currently being used in the calculations, could be implemented.

Appendix A

Common Terms

A definition of the terms that are used throughout the thesis:

Corpus - A large set of texts that is usually used for statistical analysis or hypothesis testing.

Natural Language Processing (NLP) - A field in computer science concerned with the interactions between computers and natural languages. Concerns how computers can extract meaningful information from natural language input. (Wiki)

Stop words - Stop words are a humanly defined list of words that should be removed prior to or after the processing of natural language data. They are words which are very common to any kind of text and therefore create noise in queries rather than being helpful as they occur a lot. Examples: *the, at, in, that* and *is.*

Lemma and lexeme - A *lemma* is the dictionary form or citation form of a set of words. A *lexeme* is a collection of all words that have the same meaning as the lemma of the lexeme. An example of this could be the words *walk, walked, walks and walking* that are all forms of the same lexeme with *walk* as the lemma.

Word stem - A stem is a part of a word that is common to all of it's inflected variants. An example are the words *tall, taller, tallest* that all have the stem *tall.* A list of all the inflected variants of a stem is called the inflected paradigm. Some paradigms such as *good, better,best* do not have the same stem. This is called a suppletive paradigm.

Multi-word term - A term that consists of several words, such as "bus driver". In danish multi-word terms do not exist, as the terms are just concatenated, for example "skoletaske" (school bag).

Word classes - The different classes of words in a natural language. Nous, verbs, adverbs etc.

Part-of-speech tagging (POS-tagging) - The process of assigning an appropriate word class to a word in a context.

Penn Treebank Tagset - Tagset that provides a tag for each word. A table of the tagset can be found on appendix C.

Word-sense disambiguation - The process of determining the sense of a word in a given context.

Query expansion - Expansion of a query to include more concepts than those explicitly stated in the base query

Bag of words - A concept where a text is stored as a set of words. The order of the words in a document is not stored.

Barrier - In computer science a barrier is a synchronization method. If a set of threads share a barrier they all must stop and wait until all threads have reached the barrier.

Model-View-Controller - A design pattern that separates model, view and controller into 3 separated parts. The view is the visual representation for the user. The model is the engine of the program and the controller is able to coordinate communication between model and view according to the actions the user performs.

API - Application programming interface. A way for software components to communicate with each other.

SemCor corpus - Collection of documents that has been POS-tagged and senses for WordNet has been found by humans.[1]

F-measure - A way to determine a tests accuracy by looking at the precision and recall.

[1] Can be downloaded at `http://www.cse.unt.edu/~rada/downloads.html`

Appendix B

Use case diagrams

Figure B.1 shows the use case diagram of the tool developed in this thesis. The different actors, the use cases and the relation between use cases is shown in the figure.

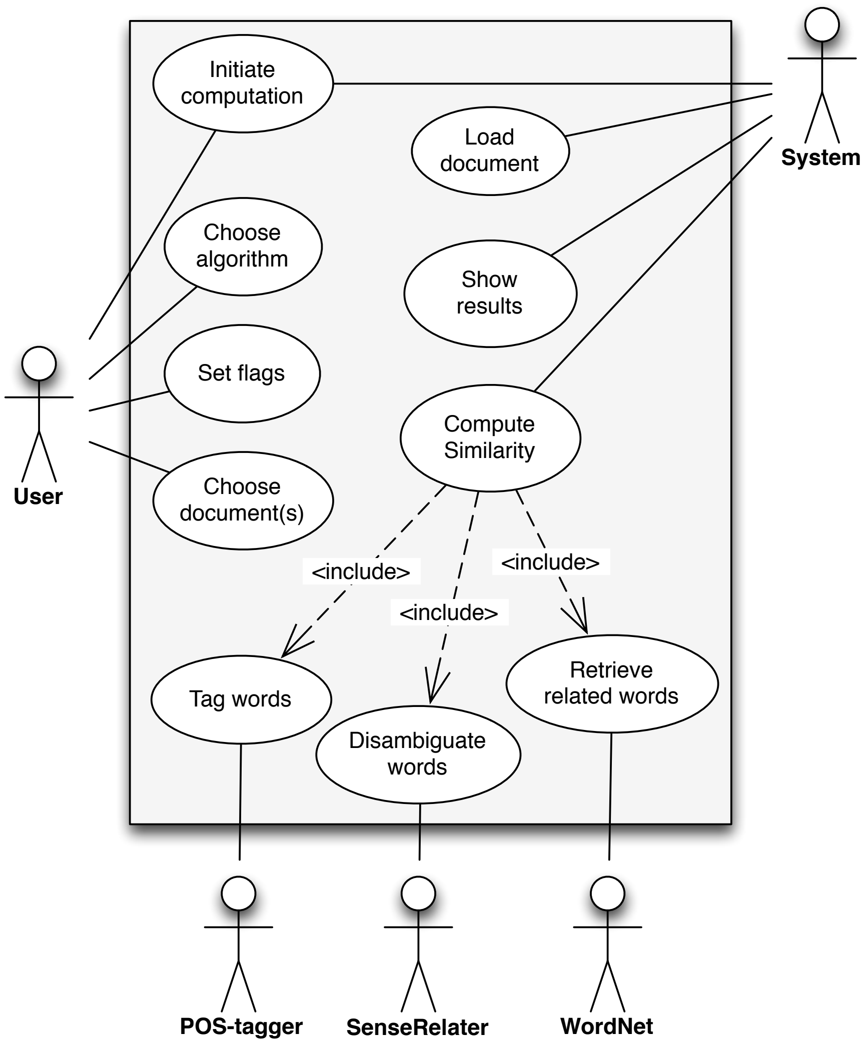


Figure B.1: Use case diagram showing actors and their use cases.

Appendix C

Penn Treebank tagset

Contains the total Penn Treebank tagset[1], which provides a tag for all words.

[1] The list was downloaded from http://www.computing.dcu.ie/~acahill/tagset.html

Tag	Tag description
CC	Coordinating conjunction
CD	Cardinal Number
DT	Determiner
EX	Existential there
FW	Foreign Word
IN	Preposision or subordinating conjunction
JJ	Adjective
JJR	Adjective, comparative
JJS	Adjective, superlative
LS	List Item Marker
MD	Modal
NN	Noun, singular or mass
NNP	Proper Noun, singular
NNPS	Proper Noun, plural
NNS	Noun, plural
PDT	Predeterminer
POS	Possessive Ending
PRP	Personal Pronoun
PRP$	Possessive Pronoun
RB	Adverb
RBR	Adverb, comparative
RBS	Adverb, superlative
RP	Particle
SYM	Symbol
TO	to
UH	Interjection
VB	Verb, base form
VBD	Verb, past tense
VBG	Verb, gerund or persent participle
VBN	Verb, past participle
VBP	Verb, non-3rd person singular present
VBZ	Verb, 3rd person singular present
WDT	Wh-determiner
WP	Wh-pronoun
WP$	Possessive wh-pronoun
WRB	Wh-adverb

Appendix D

Stanford POS-Tagger Readme

Stanford POS Tagger, v. 3.1.1 - 2012-03-09
Copyright (c) 2002-2012 The Board of Trustees of The Leland Stanford Junior University. All Rights Reserved.

This document contains (some) information about the models included in this release and that may be downloaded for the POS tagger website at http://nlp.stanford.edu/software/tagger.shtml . If you have downloaded the full tagger, all of the models mentioned in this document are in the downloaded package in the same directory as this readme. Otherwise, included in the download are two
English taggers, and the other taggers may be downloaded from the website. All taggers are accompanied by the props files used to create them; please examine these files for more detailed information about the creation of the taggers.

For English, the bidirectional taggers are slightly more accurate, but tag much more slowly; choose the appropriate tagger based on your speed/performance needs.

English taggers

---

wsj-0-18-bidirectional-distsim.tagger
Trained on WSJ sections 0-18 using a bidirectional architecture and including word shape and distributional similarity features. Penn Treebank tagset.
Performance:
97.28% correct on WSJ 19-21
(90.46% correct on unknown words)

wsj-0-18-left3words.tagger
Trained on WSJ sections 0-18 using the left3words architecture and includes word shape features. Penn tagset.
Performance:
96.97% correct on WSJ 19-21
(88.85% correct on unknown words)

wsj-0-18-left3words-distsim.tagger
Trained on WSJ sections 0-18 using the left3words architecture and includes word shape and distributional similarity features. Penn tagset.
Performance:
97.01% correct on WSJ 19-21
(89.81% correct on unknown words)

english-left3words-distsim.tagger
Trained on WSJ sections 0-18 and extra parser training data using the left3words architecture and includes word shape and distributional similarity features. Penn tagset.

english-bidirectional-distsim.tagger
Trained on WSJ sections 0-18 using a bidirectional architecture and including word shape and distributional similarity features. Penn Treebank tagset.

wsj-0-18-caseless-left3words-distsim.tagger
Trained on WSJ sections 0-18 left3words architecture and includes word shape and distributional similarity features. Penn tagset. Ignores case.

english-caseless-left3words-distsim.tagger
Trained on WSJ sections 0-18 and extra parser training data using the left3words architecture and includes word shape and distributional similarity features. Penn tagset. Ignores case.

Appendix E

Installation guide

The tool has some requirements in order to run. See section 5.6. This guide will show how to install the requirements including the tool. The guide is split into 3 sections depending on which platform the tool should be installed on. For Windows WordNet and Perl including modules should be installed manually, The guide covers how to do so. On Mac and Linux almost everything will be installed automatically using a Shell script. The needed resources for installing the tool on each platform is located in the "Binaries" folder and on the website. The guide below can also be downloaded as a separate PDF on `http://www.student.dtu.dk/~s093263/similaritytool/`.

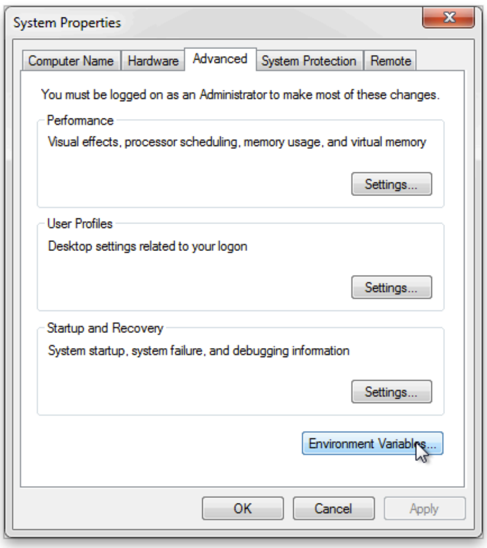
Similarity Tool install guide

Download the software from `http://www.student.dtu.dk/~s093263/similaritytool/`

Windows

This has been tested working with Windows 7 but should work on older versions as well.

- Start by downloading the latest version of Similarity Tool for Windows on the website.
- Unzip it.
- Install WordNet using the included installer `WordNet-2-1.1.exe`.
- Add WordNet to the PATH as a environment variable by going to "Control Panel" → "System and Security" → "System". Click on "Advanced system settings". In the popup window click on "Environment Variables..." as shown in the screenshot below.



System Properties

Computer Name | Hardware | Advanced | System Protection | Remote

You must be logged on as an Administrator to make most of these changes.

Performance

Visual effects, processor scheduling, memory usage, and virtual memory

Settings...

User Profiles

Desktop settings related to your logon

Settings...

Startup and Recovery

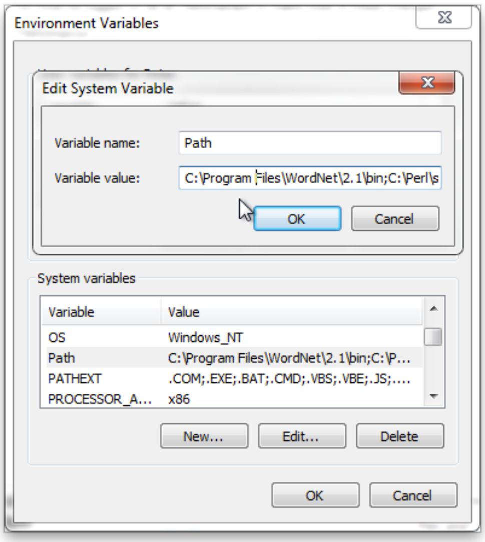
System startup, system failure, and debugging information

Settings...

Environment Variables...

OK | Cancel | Apply

Find the variable called "Path" under "System variables". Select it and press the "Edit.." button. If the variable does not exists, create it by clicking the "New.." button. Add the location of the WordNet "Bin" folder. Usually "C:\Program Files\WordNet\2.1\bin". See the screenshot below.


Environment Variables

Edit System Variable

Variable name: Path

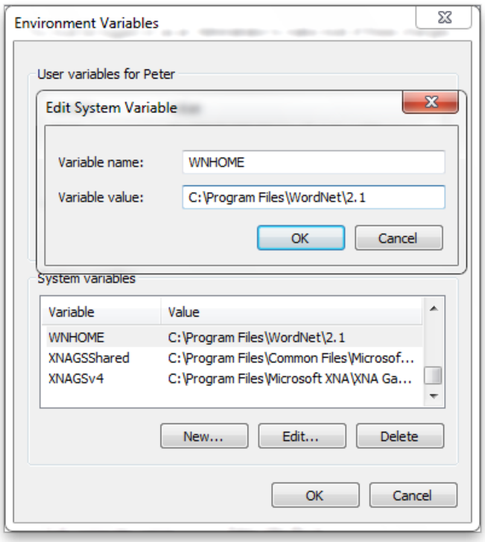
Variable value: C:\Program Files\WordNet\2.1\bin;C:\Perl\s

System variables

Variable	Value
OS	Windows_NT
Path	C:\Program Files\WordNet\2.1\bin;C:\P...
PATHEXT	.COM;.EXE;.BAT;.CMD;.VBS;.VBE;.JS;....
PROCESSOR_A...	x86

Click OK.

- Furthermore add the WNHOME as a variable by clicking the "New..." button under "System variables" and typing in "WNHOME" as the Variable name and the location of WordNet for the Variable value. Typically this is "C:\Program Files\WordNet\2.1". See the screenshot below.


Environment Variables

User variables for Peter

Edit System Variable

Variable name: WNHOME

Variable value: C:\Program Files\WordNet\2.1

System variables

Variable	Value
WNHOME	C:\Program Files\WordNet\2.1
XNAGSShared	C:\Program Files\Common Files\Microsof...
XNAGSv4	C:\Program Files\Microsoft XNA\XNA Ga...

Click OK all the way out.

- Install Perl using the included installer `ActivePerl-5.14.2.1402-MSWin32-x86-295342.msi`
- Because of a bug in one of the Perl modules a few files have to be modified for the modules to work[1].
  Open the dict folder for WordNet. Typically in "C:\Program Files\WordNet\2.1\dict".
  Take a copy of "data.noun" and parse it in the same location. Rename it to "noun.dat"
  Take a copy of "data.verb" and parse it in the same location. Rename it to "verb.dat"
- Open up Command Prompt and type in the following. Press enter for each line:
  ```
  cpan App::cpanminus
  cpanm WordNet::SenseRelate::AllWords --force
  ```
- Finally install the actual tool by running the included installer `Similarity Tool`.
- If the location of Perl has been modified the settings.xml file must be edited.
- To start the tool open "C:\Program Files\Similarity Tool\Similarity Tool.exe".

Mac OS X

This has been tested working with Mac OS X Lion (10.7.3). `Xcode` must be installed on the computer. It can be downloaded for free at the Mac App Store or at `developer.apple.com`.

- Start by downloading the latest version of Similarity Tool for Mac OSX on the website.
- Open the .dmg file
- Launch the .mpkg file and follow the instructions. Note that the installation can take several minutes.
- If the location of Perl has been modified the settings.xml file must be edited. It is located in "/Applications/SimilarityTool.app/Contents/Resources/Java"
- To start the tool open `Similarity Tool.app` located in the `Applications` folder.

Linux

This has been tested working with Ubuntu 12.04. It requires `apt-get` to be installed on the system.

- Start by downloading the latest version of Similarity Tool for Linux on the website.
- Unzip it.
- Open up a terminal and cd into the unzipped folder.

[1]This is explained further here `http://www.mail-archive.com/wn-similarity@yahoogroups.com/msg00595.html`

- Make sure that `Java` is installed by running
  `java -version`.

  If it is not installed, do so by running
  ```
  sudo apt-get -y install default-jre
  ```

- Make sure that `Perl` is installed by running
  ```
  perl -v
  ```

  If it is not installed, do so by running
  ```
  sudo apt-get -y install perl
  ```

- Next run the following:
  ```
  sudo sh INSTALL
  ```

  This will install WordNet, set the `PATH` and install the required Perl modules. Please note that this can take several minutes.

- If the location of Perl has been modified the settings.xml file must be edited.

- Start the jar file by running
  ```
  java -jar SimilarityTool.jar
  ```

In order to uninstall run
```
sudo sh UINSTALL
```

Appendix F

Structure diagrams

Diagrams showing how selected parts of the program is structured. Note that the diagrams are simplified a bit compared to the actual implementation. Check out appendix H for documentaion about the implementation and appendix G for the source code.

F.1 Class diagrams

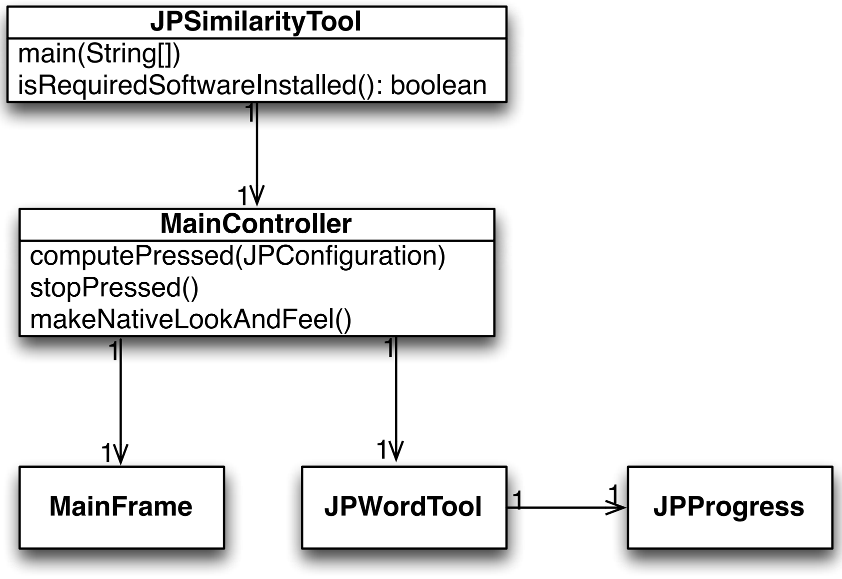


Figure F.1: Overall class diagram with associations.

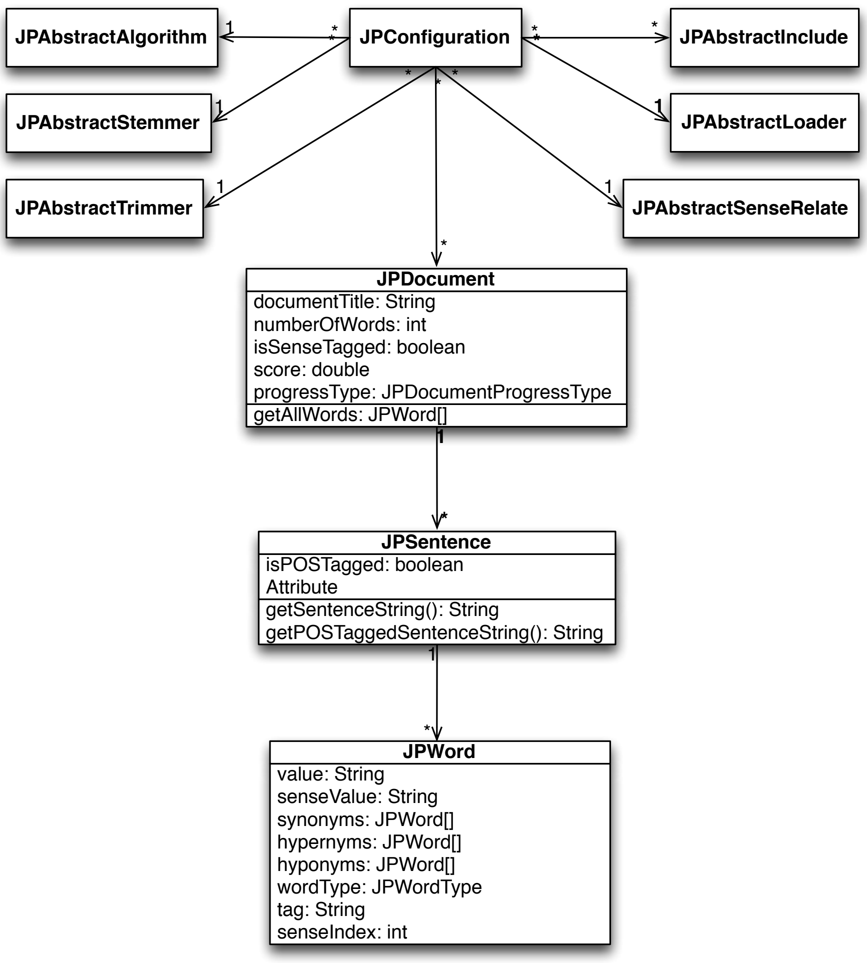


Figure F.2: Class diagram for configuration with associations.

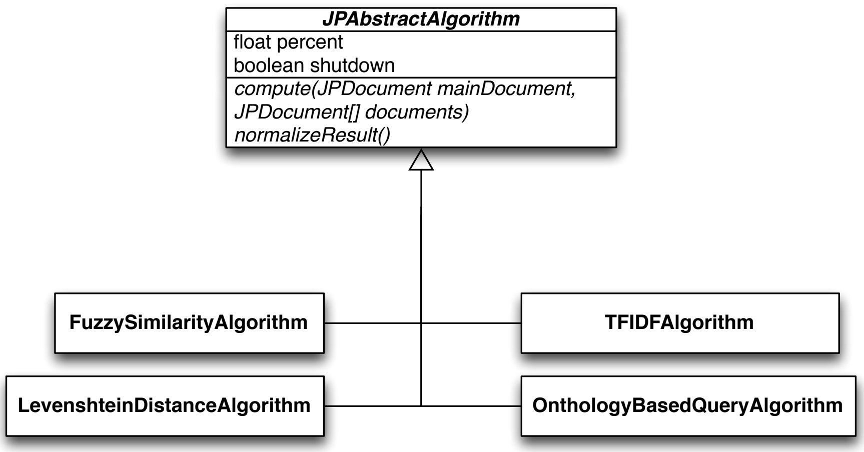

JPAbstractAlgorithm
float percent
boolean shutdown
compute(JPDocument mainDocument, JPDocument[] documents)
normalizeResult()
FuzzySimilarityAlgorithm
TFIDFAlgorithm
LevenshteinDistanceAlgorithm
OnthologyBasedQueryAlgorithm

Figure F.3: Class diagram for algorithms.

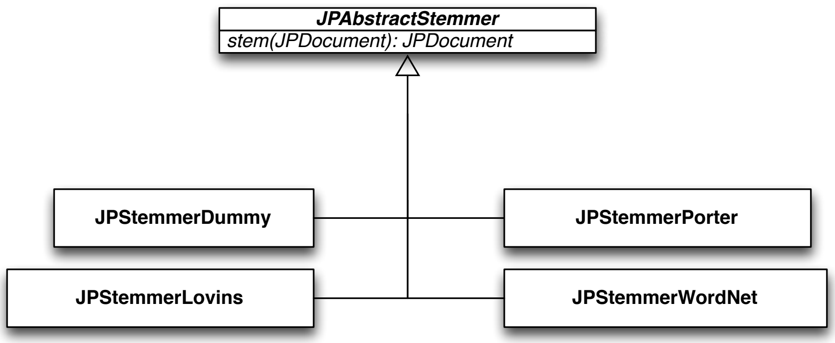


Figure F.4: Class diagram for stemmers.

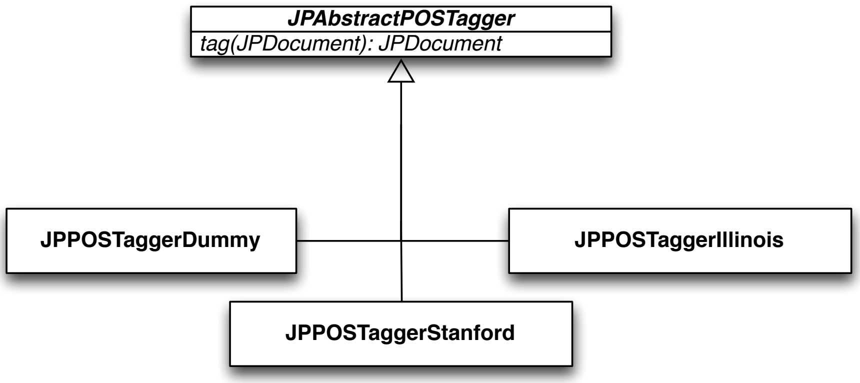


Figure F.5: Class diagram for POS-taggers.

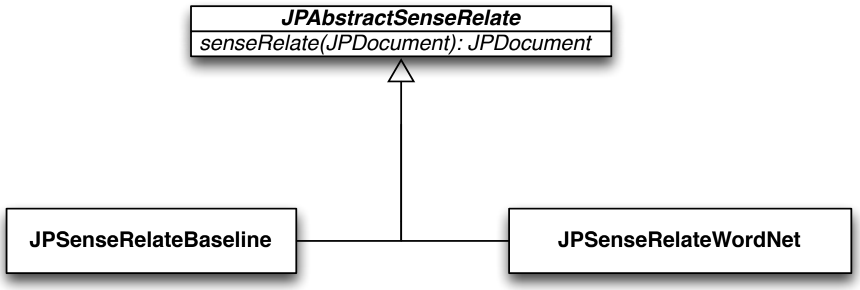


Figure F.6: Class diagram for sense relaters.

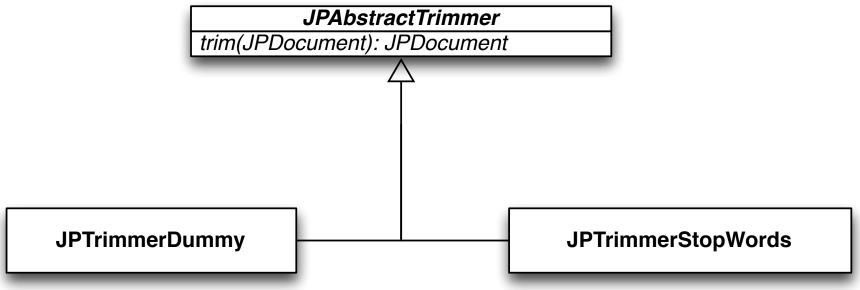


Figure F.7: Class diagram for trimmers.

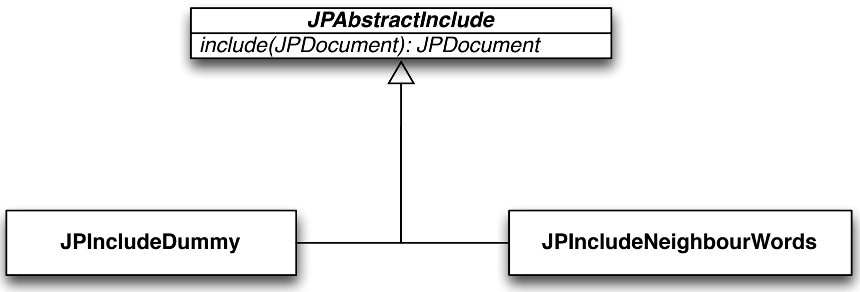


Figure F.8: Class diagram for includers.

F.2 Sequence diagrams

On figure F.9 and F.10 are sequence diagrams from the implementation.

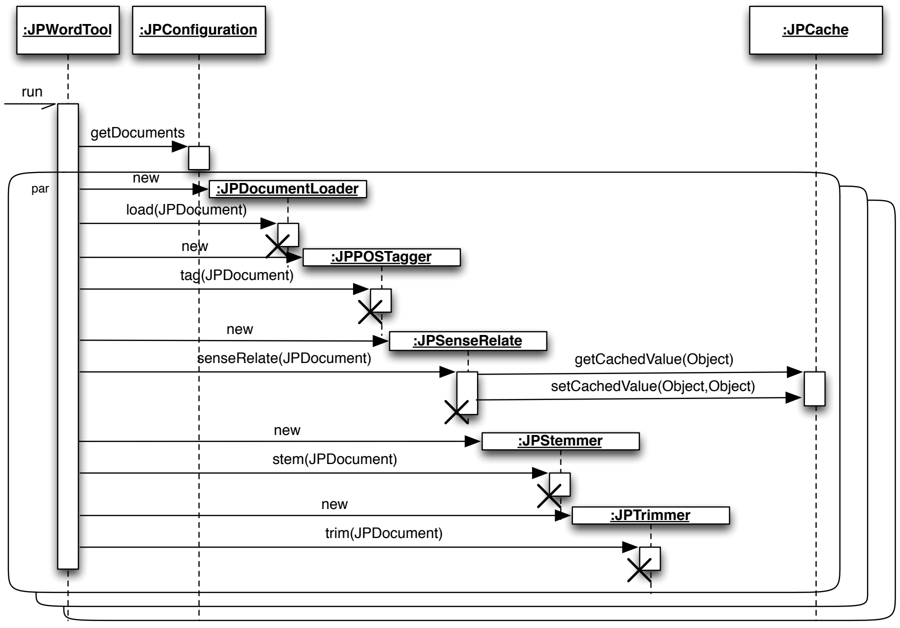


Figure F.9: Sequence diagram for loading documents.

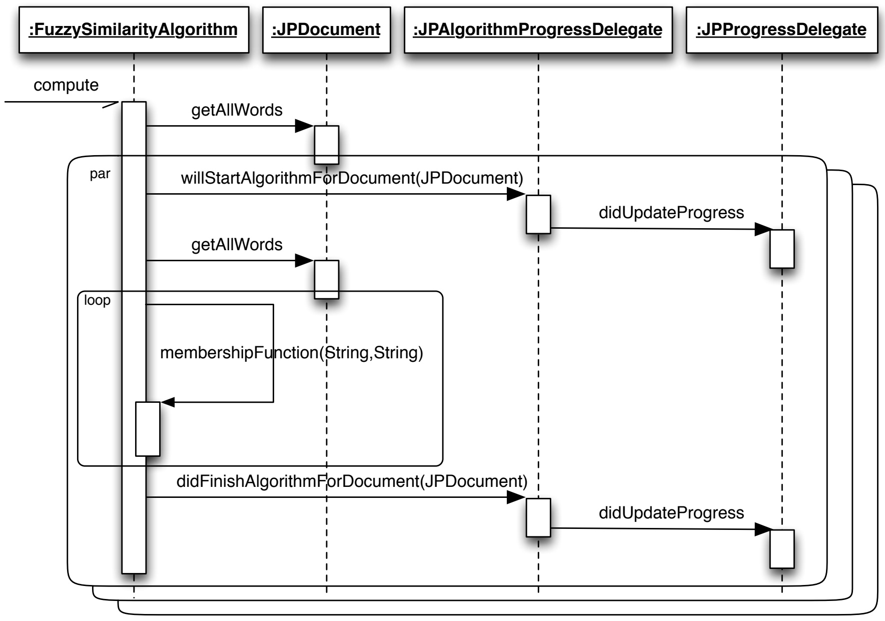

:FuzzySimilarityAlgorithm
:JPDocument
:JPAlgorithmProgressDelegate
:JPProgressDelegate
compute
getAllWords
par
willStartAlgorithmForDocument(JPDocument)
didUpdateProgress
getAllWords
loop
membershipFunction(String,String)
didFinishAlgorithmForDocument(JPDocument)
didUpdateProgress

Figure F.10: Sequence diagram for fuzzy similarity.

Appendix G

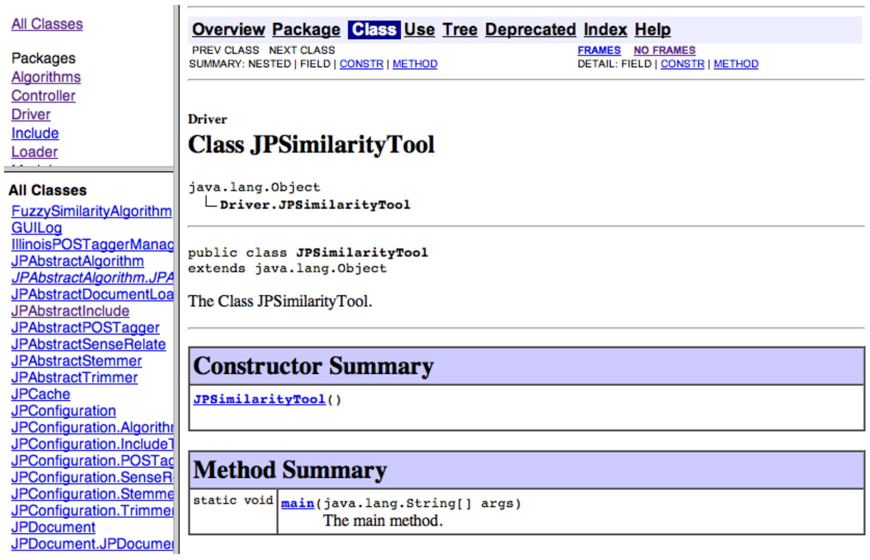
Source code

The source code is located in the "Source" folder and on
http://www.student.dtu.dk/~s093263/similaritytool/.

APPENDIX H

Documentation

The tool is documented with JavaDoc. It is included in the "Documentation" folder and on http://www.student.dtu.dk/~s093263/similaritytool/


All Classes

Packages
Algorithms
Controller
Driver
Include
Loader

All Classes
FuzzySimilarityAlgorithm
GUILog
IllinoisPOSTaggerManag
JPAbstractAlgorithm
JPAbstractAlgorithm.JPA
JPAbstractDocumentLoa
JPAbstractInclude
JPAbstractPOSTagger
JPAbstractSenseRelate
JPAbstractStemmer
JPAbstractTrimmer
JPCache
JPConfiguration
JPConfiguration.Algorithr
JPConfiguration.IncludeT
JPConfiguration.POSTac
JPConfiguration.SenseR
JPConfiguration.Stemme
JPConfiguration.Trimmer
JPDocument
JPDocument.JPDocume

Overview Package Class Use Tree Deprecated Index Help
PREV CLASS NEXT CLASS FRAMES NO FRAMES
SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

Driver
Class JPSimilarityTool

```
java.lang.Object
  └ Driver.JPSimilarityTool
```

```
public class JPSimilarityTool
extends java.lang.Object
```

The Class JPSimilarityTool.

Constructor Summary

`JPSimilarityTool()`

Method Summary

static void	`main(java.lang.String[] args)` The main method.

Figure H.1: Screenshot of the JavaDoc.

Appendix I

SenseRelate

Perl program used to performs sense relation. It uses several Perl modules. The program is also included in the source code

```perl
#!/usr/bin/perl -w

my @context = $ARGV[0];
my $jsonResponse = 0;
my $posTag = 0;

# Check parameters
if (@ARGV > 3 || @ARGV < 1 || $ARGV[0] eq "-h") {
        print "usage: disambiguate.pl string\neg: disambiguate.pl \"this is a string\"\n";
        print "use -j for JSON responses\"\n";
        print "use -t to force POS tagging\"\n";
        exit;
}
if (@ARGV == 1) {

} elsif (@ARGV == 2 && $ARGV[0] eq "-j") {
        $jsonResponse = 1;
        @context = $ARGV[1];
} elsif (@ARGV == 2 && $ARGV[0] eq "-t") {
        $posTag = 1;
        @context = $ARGV[1];
} elsif (@ARGV == 3 && $ARGV[0] eq "-j" && $ARGV[1] eq "-t") {
        $posTag = 1;
        $jsonResponse = 1;
        @context = $ARGV[2];
} else {
        print "Something went wrong";
        exit;
}

# Needed modules
use WordNet::SenseRelate::AllWords;
use WordNet::QueryData;
use WordNet::Tools;

# Create data query
my $qd = WordNet::QueryData->new;
my $queryDataError;
if ($jsonResponse) {
        $queryDataError = "{\"success\":0, \"message\":\"Construction of WordNet::
             QueryData failed\"}";
} else {
        $queryDataError = "Construction of WordNet::QueryData failed";
}

defined $qd or die $queryDataError;

# Create wordnet tool
my $wntools = WordNet::Tools->new($qd);
my $wnToolsError;
if ($jsonResponse) {
        $wnToolsError = "{\"success\":0, \"message\":\"Couldn't construct WordNet
             ::Tools object\"}";
} else {
        $wnToolsError = "Couldn't construct WordNet::Tools object";
}

defined $wntools or die "\nCouldn't construct WordNet::Tools object";

# Create sense relate
my $wsd = WordNet::SenseRelate::AllWords->new (wordnet => $qd,wntools =>
     $wntools,measure => 'WordNet::Similarity::lesk');

# Disambiguate text
my @results = $wsd->disambiguate (window => 3, tagged => $posTag,context => [
     @context], scheme => "sense1");

```

```
# If JSON response is expected
if($jsonResponse) {
        my $words = "";
        my $index = 0;
        @oldText = split(/ /, $context[0]);
        foreach (@results) {
                @values = split(/#/, $_);
                $count = scalar grep { defined $_ } @values;
                my $senseIndex;
                if ($count == 3) {
                        $senseIndex = $values[2];
                } else {
                        $senseIndex = 0;
                }

                $object = $oldText[$index] . "\", \"newWord\":\"" . $values[0] . "\",
                      \"tag\":\"" . $values[1] . "\", \"senseIndex\":" . $senseIndex;

                $words = $words . "\{\"word\":\"" . $object . "\},";
                $index = $index + 1;
        }
        chop($words);
        my @response = "{\"success\": 1, \"relations\": [" . $words . "] }";
        print @response;
} else {
        print "@results";
}
```

Appendix J

Questionnaire

The questionnaire used for creating a human judgement of similarity for 9 different documents. `.txt` files for each of these articles can be found in the "Articles" folder and on http://www.student.dtu.dk/~s093263/similaritytool/.

Johan van Beusekom & Peter Gammelgaard Poulsen

Article comparisons

This document contains a range of short articles. We would like to you to read these articles, and as you do so, think about these two questions:

- Do the articles concern the same topic?
- Do the articles come to the same conclusion on a topic? (If they indeed do concern the same topic)

This comparison has to be made for all articles, and we will provide a scoreboard, where you can easily insert your scores. We will ask you to rate the article similarities on a scale from 0-100, where 100 would mean identical articles and 0 means the articles have nothing in common. While some of the articles might not concern the same topic, they might be of the same genre, which is something to keep in mind when rating their similarities.
This is not a test, but rather a measurement for statistical purposes and it is not possible to answer the questions wrongly.

The scoreboard for question 1.

Article no.	1	2	3	4	5	6	7	8	9
1	100	-	-	-	-	-	-	-	-
2		100	-	-	-	-	-	-	-
3			100	-	-	-	-	-	-
4				100	-	-	-	-	-
5					100	-	-	-	-
6						100	-	-	-
7							100	-	-
8								100	-
9									100

The scoreboard for question 2.

Article no.	1	2	3	4	5	6	7	8	9
1	100	-	-	-	-	-	-	-	-
2		100	-	-	-	-	-	-	-
3			100	-	-	-	-	-	-
4				100	-	-	-	-	-
5					100	-	-	-	-
6						100	-	-	-
7							100	-	-
8								100	-
9									100

Article 1

So yes, I already saw The Avengers last night. Despite the fact that the movie is already getting a lot of praise both from critics, fans and regular moviegoers, I still can't help but be worried about what will the movie's final product will turn out.

I have those experiences that when people says good things about a movie, I get too excited and end up being disappointed in the end. But I'm completely relieved because everything that I read and heard from people is true. The Avengers is totally awesome! I know it may sound like a big understatement but The Avengers is definitely the best movie of the season (So far). The fact that the movie works smoothly in all aspects of film making (story, character dynamics, visuals etc.) is enough proof that Marvel did what no other company can do - create a massive cinematic universe that has consistency and cohesion.

At this point, it already doesn't matter who's comic company is better. In the end no one can ever deny the universal fact that Marvel took the risk to make a movie that only exist on all geeks' dreams. Now, that dream is already a reality.

Article 2

Steven Jobs was born February 24, 1955, in San Francisco, California, and was adopted by Paul and Clara Jobs. He grew up with one sister, Patty. Paul Jobs was a machinist and fixed cars as a hobby. Jobs remembers his father as being very skilled at working with his hands.

In 1961 the family moved to Mountain View, California. This area, just south of Palo Alto, California, was becoming a center for electronics. Electronics form the basic elements of devices such as radios, televisions, stereos, and computers. At that time people started to refer to the area as "Silicon Valley." This is because a substance called silicon is used in the manufacturing of electronic parts.

As a child, Jobs preferred doing things by himself. He swam competitively, but was not interested in team sports or other group activities. He showed an early interest in electronics and gadgetry. He spent a lot of time working in the garage workshop of a neighbor who worked at Hewlett-Packard, an electronics manufacturer.

Jobs also enrolled in the Hewlett-Packard Explorer Club. There he saw engineers demonstrate new products, and he saw his first computer at the age of twelve. He was very impressed, and knew right away that he wanted to work with computers.

Article 3

Barack H. Obama is the 44th President of the United States.

His story is the American story values from the heartland, a middle-class upbringing in a strong family, hard work and education as the means of getting ahead, and the conviction that a life so blessed should be lived in service to others.

With a father from Kenya and a mother from Kansas, President Obama was born in Hawaii on August 4, 1961. He was raised with help from his grandfather, who served in Patton's army, and his grandmother, who worked her way up from the secretarial pool to middle management at a bank.

After working his way through college with the help of scholarships and student loans, President Obama moved to Chicago, where he worked with a group of churches to help rebuild communities devastated by the closure of local steel plants.

He went on to attend law school, where he became the first African-American president of the Harvard Law Review. Upon graduation, he returned to Chicago to help lead a voter registration drive, teach constitutional law at the University of Chicago, and remain active in his community.

President Obama's years of public service are based around his unwavering belief in the ability to unite people around a politics of purpose. In the Illinois State Senate, he passed the first major ethics reform in 25 years, cut taxes for working families, and expanded health care for children and their parents. As a United States Senator, he reached across the aisle to pass groundbreaking lobbying reform, lock up the world's most dangerous weapons, and bring transparency to government by putting federal spending online.

He was elected the 44th President of the United States on November 4, 2008, and sworn in on January 20, 2009. He and his wife, Michelle, are the proud parents of two daughters, Malia, 13, and Sasha, 10.

Article 4

The scale of Arsenals achievement is perhaps best appreciated by acknowledging its rarity. No modern-day team, not Liverpool in the 1970s and 1980s or Manchester United in the 1990s and 2000s, have matched the virtual perfection of Arsenals unbeaten league season in 2003-4.

Indeed, in the entire history of English football, to find even one other club that has emulated Arsne Wengers team you have to go way back to the original Invincibles, Preston North End in 1888-89. Yet Prestons unbeaten season comprised only 22 matches whereas Arsenals stretched to almost double that in a rather more competitive era. Of their 38 games in 2003-4, they won 26 and drew 12 to also achieve one of the highest ever points tallies of 90.

Of course, the full unbeaten run actually lasted some 49 games and went from the end of the 2002-3 season until well into the 2004-5 campaign.

Wenger, who recently became Arsenals longest-serving manager, still rates it as the greatest accomplishment in his entire coaching career. The teams defensive strength rested on the centre-back partnership between Sol Campbell and Kolo Toure. In front of them, Patrick Vieira and Gilberto Silva formed a base from which the wonderful attacking talents of Robert Pires, Thierry Henry and Dennis Bergkamp could really flourish.

Article 5

The President of the United States of America is the head of state and head of government of the United States. The president leads the executive branch of the federal government and is the commander-in-chief of the United States Armed Forces.

Article II of the U.S. Constitution vests the executive power of the United States in the president and charges him with the execution of federal law, alongside the responsibility of appointing federal executive, diplomatic, regulatory, and judicial officers, and concluding treaties with foreign powers, with the advice and consent of the Senate. The president is further empowered to grant federal pardons and reprieves, and to convene and adjourn either or both houses of Congress under extraordinary circumstances. Since the founding of the United States, the power of the president and the federal government have grown substantially and each modern president, despite possessing no formal legislative powers beyond signing or vetoing congressionally passed bills, is largely responsible for dictating the legislative agenda of his party and the foreign and domestic policy of the United States. The president is frequently described as the most powerful person in the world.

The president is indirectly elected by the people through the Electoral College to a four-year term, and is one of only two nationally elected federal officers, the other being the Vice President of the United States. The Twenty-second Amendment, adopted in 1951, prohibits anyone from ever being elected to the presidency for a third full term. It also prohibits a person from being elected to the presidency more than once if that person previously had served as president, or acting president, for more than two years of another person's term as president. In all, 43 individuals have served 55 four-year terms. On January 20, 2009, Barack Obama became the 44th and current president.

Article 6

Albert Einstein was born in Ulm, Germany on March 14, 1879. As a child, Einstein revealed an extraordinary curiosity for understanding the mysteries of science (started only at age 10/11). A typical child (only to his socio-economic class educated middle class), Einstein took music lessons, playing both the violin and piano a passion that followed him into adulthood. Moving first to Italy and then to Switzerland, the young prodigy graduated from high-school in 1896. In 1905, while working as a patent clerk in Bern, Switzerland, Einstein had what came to be known as his "Annus Mirabilis" or "miracle year". It was during this time that the young physicist obtained his Doctorate degree and published four of his most influential research papers, including the Special Theory of Relativity. In that, the now world famous equation "e = mc2" unlocked mysteries of the Universe theretofore unknown.

Ten years later, in 1915, Einstein completed his General Theory of Relativity and in 1921 he was awarded the Nobel Prize in Physics (iconic status cemented in 1919 when Arthur Eddington's expedition confirmed Albert Einstein's prediction). It also launched him to international superstardom and his name became a household word synonymous with genius all over the world. Einstein emigrated to the United States in the autumn of 1933 and took up residence in Princeton, New Jersey and a professorship at the prestigious Institute for Advanced Study. Today, the practical applications of Einstein's theories include the development of the television, remote control devices, automatic door openers, lasers, and DVD-players. Recognized as TIME magazine's "Person of the Century" in 1999, Einstein's intellect, coupled his strong passion for social justice and dedication to pacifism, left the world with infinite knowledge and pioneering moral leadership.

Article 7

Cooking is the process of preparing food, often with the use of heat. Cooking techniques and ingredients vary widely across the world, reflecting unique environmental, economic, and cultural traditions. Cooks themselves also vary widely in skill and training. Cooking can also occur through chemical reactions without the presence of heat, most notably as in Ceviche, a traditional Spanish dish where fish is cooked with the acids in lemon or lime juice. Sushi also utilizes a similar chemical reaction between fish and the acidic content of rice glazed with vinegar.

Chicken, pork and bacon-wrapped corn cooked in a barbecue smoker Preparing food with heat or fire is an activity unique to humans, and some scientists believe the advent of cooking played an important role in human evolution. Most anthropologists believe that cooking fires first developed around 250,000 years ago. The development of agriculture, commerce and transportation between civilizations in different regions offered cooks many new ingredients. New inventions and technologies, such as pottery for holding and boiling water, expanded cooking techniques. Some modern cooks apply advanced scientific techniques to food preparation.

Article 8

In English football, "The Invincibles" has been used to refer to either the Preston North End team of the 1880s, or the Arsenal team of the 2003-04 season. In both cases, the teams won the top division of English football unbeaten, the only two times this has occurred in English football history.

Arsenal emulated Preston's unbeaten run in the 2003-04 season going unbeaten for all 38 games, almost twice as many league games as Preston had played. Their final record for the 2003-2004 league campaign stood at 26 wins, 12 draws and 0 losses, out of 38 games total, an unbeaten run not matched in any single season by any team in an English league division.

Arsenal went 49 Premier League games unbeaten which was a new record for the most League games without defeat, the sequence coming to an end with a controversial 20 defeat to Manchester United. The Premier League commissioned a special gold version of the Premier League trophy to commemorate Arsenal's unbeaten season. A very important part of this season was Thierry Henry who scored 30 goals that season, with Robert Pirs chipping in with 14 goals.

Article 9

I was excited to see this film based on all the hype from my friends. I haven't read the books, but thought the story was interesting based on the one-line explanations I got from others.

I went into this film expecting to see something similar to the Twilight films (teen obsession over book turned film), but better.

What I got was much, much worse.

First off, I want to say that I thought the acting was great. The actors did a great job of portraying their characters.

What WASN'T so great was the lack of character development. The death scenes, even those meant to be emotional, didn't affect me at all. I am one who isn't afraid to cry like a baby at a movie, and I often do. But while the others around me were sobbing, I was just ready for the scene to be over with.

Another reason I strongly disliked this film, has to do with my title. It seems like 95% of the shots in this movie were closeup on the characters faces. These are fine, to an extent, but you begin to wonder after a while if the actors aren't all just bodiless floating heads. The fight scenes were quickly cut together, which could work, if you weren't zoomed in so much that you have no idea what was going on.

For a film with a supposed 100,000,000 dollar budget, this came across as a low-budget independent adaption of the book. They could have done so much more to improve the visual aspect of the film as well as make the characters more likable.

Overall, I would give this film 4 stars, mainly due to the great acting and costume design. Everything else, was horrible.

Bibliography

[ADFH01] Adrian Akmajian, Richard A. Demers, Ann K. Farmer, and Robert M. Harnish. *Linguistics - An Introduction to Language and Communication.* The MIT Press, 2001.

[Ban02] Satanjeev Banerjee. Adapting the lesk algorithm for word sense disambiguation to wordnet. *Department of Computer Science University of Minnesota*, 2002.

[BKA02] Henrik Bulskov, Rasmus Knappe, and Troels Andreasen. On measuring similarity for conceptual querying. *Department of Computer Science, Roskilde University*, 2002.

[BS06] Cédric Beust and Hani Suleiman. *Next Generation Java Testing - TestNG and Advanded Concepts.* Addison-Wesley, 2006.

[Fin] Mark A. Finlayson. *JSemcor 1.0.x User's Guide.*

[Fin11] Mark A. Finlayson. *MIT Java Wordnet Interface (JWI) User's Guide*, 2011.

[Fow04] Martin Fowler. *UML Distilled - A Brief Guidde to the Standard Object Modeling Language.* Pearson Education, third edition, 2004.

[JFM00] Richard Johnson, John Freund, and Irwin Miller. *Probability and Statistics for Engineers.* Pearson, eighth edition, 2000.

[Lee] Michael D. Lee. A comparison of machine measures of text document similarity with human judgments.

[Lov68] Julie Beth Lovins. Development of a stemming algorithm. *Massachusetts Institute of Technology*, 1968.

[M.F80] M.F.Porter. An algorithm for suffix stripping. 1980.

[Mic05] Jason Michelizzi. Semantic relatedness applied to all words sense disambiguation. Master's thesis, 2005.

[Mit03] Ruslan Mitkov. *The Oxford Handbook Of Computational Linguistics*. University Press, 2003.

[OP03] Vladimir Oleshchuk and Asle Pedersen. Ontology based semantic similarity comparison of documents. *Agder University College*, 2003.

[Pin04] Brandon Pincombe. Comparison of human and latent semantic analysis (lsa) judgements of pairwise document similarities for a news corpus. *Intelligence, Surveillance and Reconnaissance Division*, 2004.

[PK] Ted Pedersen and Varada Kolhatkar. Wordnet::senserelate::allwords - a broad coverage word sense tagger that maximizes semantic relatedness. *Department of Computer Science University of Minnesota*.

[POS] Pos tagging (state of the art). http://aclweb.org/aclwiki/index.php?title=POS_Tagging_%28State_of_the_art%29.

[RZ] Dan Roth and Dmitry Zelenko. Part of speech tagging using a network of linear separators. *University of Illinois - Department of Computer Science*.

[Sch] Helmut Schmid. Probabilistic part-of-speech tagging using decision trees. *Universistät Stuttgart - Insitut für maschinelle Sprachverarbeitung*.

[SG03] Piotr S. Szczepaniak and Marcin Gil. Practical evaluation of textual fuzzy similarity as a tool for information retrieval. *Institute of Computer Science, Technical University of Lodz*, 2003.

[TP10] Peter D. Turney and Patrick Pantel. From frequency to meaning: Vector space models of semantics. *Journal of Articial Intelligence Research 37*, 2010.

[UDK04a] Ozlem Uzuner, Randall Davis, and Boris Katz. Recognizing text similarity. *MIT Computer Science and Artificial Intelligence Laboratory*, 2004.

[UDK04b] Ozlem Uzuner, Randall Davis, and Boris Katz. Using empicical methods for evaluationg expression and content similarity. *Massachusetts Institute of Technology*, 2004.

[Wor] Wordnet statistics. http://wordnet.princeton.edu/wordnet/man/wnstats.7WN.html.