Textual Similarity

Aqeel Hussain

Kongens Lyngby 2012
IMM-BSc-2012-16

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

Abstract

The purpose of this thesis is to identify methods for textual similarity measurement. Many proposed solutions for this problem are suggested in literature. Three of these proposals are discussed in depth and implemented. Two focuses on syntax similarity and one focus on semantic similarity. The two syntax algorithms represents edit distance and vector space model algorithms. The semantic algorithm is an ontology based algorithm, which lookup words in WordNet. Using this tool the relatedness between two given texts is estimated. The other algorithms use Levenshtein and n-gram, respectively. The performance of these implementations are tested and discussed.

The thesis concludes that performance is very different and all algorithms perform well in their respective fields. The algorithms cannot be distinguished as to determining one, which outshines the others. Thus an algorithm implementation has to be picked based on the task at hand.

Resumé

Formålet med denne afhandling er at identificere metoder til tekst sammenligning. Mange forslåede løsninger til dette problem er givet i litteraturen. Der er valgt tre metoder til dybdegående diskussion og implementering. To af disse fokuserer på syntaktisk sammenlign og en fokuserer på semantisk sammenligning. De to syntaktiske algoritmer repræsenterer henholdsvis afstands algoritmer og vektorrums algoritmer. Den semantiske algoritme er baseret på en ontologi, hvor ord er slået op i WordNet. Ved at anvende dette værktøj er der er målt hvor meget to givne tekster er beslægtet med hinanden. De opnåede resultaters præstation er målt igennem test og bliver diskuteret.

Denne afhandling konkluderer at resultaterne er meget forskellige og at alle algoritmer klare sig godt i de hovedsageligt henvender sig til. Der kan ikke vælges én bestem algoritme, som overgår de andre i alle henseender. Dette betyder at en implantation skal vælges ud fra den givne opgave der er stillet.

Preface

This thesis was prepared at Informatics Mathematical Modelling, the Technical University of Denmark in partial fulfilment of the requirements for acquiring the B.Sc. degree in engineering.

This project is concerned with comparing two texts in order to discover how closely they discuss the same topic. This is related to how humans perceive similarity and what measures need to be taken, in order to duplicate human behaviour.

The thesis consists of a summary in English and Danish, a discussion of relevant algorithms to sort textual similarity and a conclusion on how these perform in relation to conducted tests.

The source code, executable file, installation guide and other relevant files can be found on the attached CD.

Kongens Lyngby, June 2012

Aqeel Hussain

Acknowledgements

I would like to thank Robin Sharp for his guidance and moral support, which he has provided, for the duration of this thesis.

Contents

CHAPTER 1

Introduction

The ability to distinguish when something is similar is an important task for humans, since this is the main way concepts are understood. Relating things to each other and categorizing elements helps the individual perceive material and relate how relevant this material is. Similarity is a wide term, which must be defined in the context it is used. When referring to similarity, it may refer to different levels of resemblance. The actual similarity therefore dependents on what is measured. In some sense something may be similar, but in another it may not.

One subfield of measuring similarity is measuring textual similarity, which is the focus of this thesis. Text is a representation of written language, which again is a representation of language. Language is the main form in which humans communicate and thus have a huge impact on everyday life. These languages are governed by syntactical rules and semantic relations. By parsing syntax and semantics of a given text it is possible to extract the meaning, which this text carries.

The main use of text is to carry information, which is to be used at some later point. These are often made for the sake of information sharing. It is therefore of great relevance to be able to extract the meaning of a given text and relate it in some sense. Information sharing through the internet has become quite popular in recent years and is a daily routine for many, since knowledge is freely available. Textual similarity can be used to categorize this information and allows retrieval of more relevant data.

Several proposals for finding textual similarity have been made. These differ in the way they evaluate text, some are simple and others are more complex. This paper discusses these approaches by looking at how similarity is measured. A selection of approaches is made, which are carried out in order to further examine them.

Chapter 2 presents an overview of the field of information retrieval. Here the different methods, which can be used to compare text in order to determine textual similarity, are formulated. This starts with an introduction to similarity and how words can be looked up and compared. Based on this, an in depth explanation of how this can be used

in practice, is given. Three proposed methods are chosen and discussed for further evaluation. This analysis serves as a description for the design of the selected approaches, which can be found in chapter 3. Chapter 4 presents the structure of the actual implementation. An evaluation of this implementation is given in the form of expectations to the constructed program. This can be found in chapter 5. The program is tested to see if the expected performance can be validated. The measures for this validation are explained in chapter 6. Chapter 7 presents the results of the constructed tests. A discussion of these results and how well they correlate with the stated expectancy for the program can be found in chapter 8. Further this chapter presents improvements for the program, based on the test results. Finally the whole process is summed up in chapter 9 and conclusions are drawn, based on the achieved result.

CHAPTER 2

Analysis

This section describes an analysis of information retrieval with specific interest in the field of textual similarity and what is needed to approximate an estimate of similarity between two texts.

When concerned with finding similarity between two texts there are two main thoughts of how to compare these. A semantic comparison is concerned with the meaning of the text i.e. in which context the text is written. The meaning of the objects in language is interconnected and shares a large number of relations between each other. Examples of such relations are: "*synonymy, hypernymy, hyponymy, meronymy, holonymy* and *antonymy"* (described later). In other words semantics are what is understood from a given text, connected with previous knowledge about the content. In contrast to semantics, syntax focuses on the structural build-up of a language. These grammatical rules govern correct form, for composition of language. Breaking down sentences and analysing them from a syntactical view, can give meaning as to what the sentence is trying to say. These are made by following certain rules, which obey the overall allowed structures specified for the given language. A combination of the two can be used to retrieve a more in-depth relation between two texts. This is the way humans interpret text. Many proposals using the above approaches have been made and some examples are given below. [1]

2.1 WordNet

WordNet is a large lexical database containing the words of the English language. It resembles the traits of a thesaurus in that it structures words that have similar meaning together. WordNet is something more, since it also specifies different connections for each of the senses of a given word. These connections place words that are semantically related close to one another in a network. WordNet also displays some quality of a dictionary, since it describes the definition of words and their corresponding part-of-speech.

Synonym relation is the main connection between words, which means that words which are conceptually equivalent, and thus interchangeable in most contexts, are grouped together. These groupings are called synsets and consist of a definition and relations to other synsets. A word can be part of more than one synset, since it can bear more than one meaning. WordNet has a total of 117 000 synsets, which are linked together. Not all synsets have a distinct path to another synset. This is the case, since the data structure in WordNet is split into four different groups; nouns, verbs, adjectives and adverbs (since they follow different rules of grammar). Thus it is not possible to compare words in different groups, unless all groups are linked together with a common entity. There are some exceptions which links synsets cross part-of-speech in WordNet, but these are rare. It is not always possible to find a relation between two words within a group, since each group are made of different base types. The relations that connect the synsets within the different groups vary based on the type of the synsets. The synsets are connected within the different groups by the following relationships:

- Nouns
  - *hypernyms*: *Y* is a hypernym of *X* if every *X* is a (kind of) *Y* (*canine* is a hypernym of *dog*)
  - *hyponyms*: *Y* is a hyponym of *X* if every *Y* is a (kind of) *X* (*dog* is a hyponym of *canine*)
  - *coordinate terms*: *Y* is a coordinate term of *X* if *X* and *Y* share a hypernym (*wolf* is a coordinate term of *dog*, and *dog* is a coordinate term of *wolf*)
  - *holonyms*: *Y* is a holonym of *X* if *X* is a part of *Y* (*building* is a holonym of *window*)
  - *meronyms*: *Y* is a meronym of *X* if *Y* is a part of *X* (*window* is a meronym of *building*)
- Verbs
  - *hypernym*: the verb *Y* is a hypernym of the verb *X* if the activity *X* is a (kind of) *Y* (*to perceive* is an hypernym of *to listen*)
  - *troponym*: the verb *Y* is a troponym of the verb *X* if the activity *Y* is doing *X* in some manner (*to lisp* is a troponym of *to talk*)
  - *entailment*: the verb *Y* is entailed by *X* if by doing *X* you must be doing *Y* (*to sleep* is entailed by *to snore*)
  - *coordinate terms*: those verbs sharing a common hypernym (*to lisp* and *to yell*)
- Adjectives
  - *related nouns*
  - *similar to*
  - *participle of verb*
- Adverbs
  - *root adjectives*

***Table 2.1**: explaining the relations in WordNet* [4]

The most used relation connecting synsets is the hypernym/hyponym relation, which specifies "*IS-A*" relations. The easiest way to capture the nature of these relations is to think of them as taxonomies. It then becomes evident that hyponym relations are transitive i.e. all dogs are canines and all golden retrievers are canines. In terms hypernyms are more generic than their hyponyms, which are more specific.

The coordinate term is self-explanatory from the above example, since both wolf and dog shares the hypernym canine.

The holonym/meronym relation connecting noun synsets specifies the part-whole relation. These relations are also called "*HAS-A*" relations and inherits from their superordinate. Properties are inherited downward and show that the meronym is part of the holonym. The reverse is not necessarily true i.e. a building is not part of a window.

The troponym relation is the manner in which something is being done. These relate to one another in the way they are performed i.e. to yell is to communicate in some manner. Specificity is inherited downward, thus the more general terms are superordinate.

Entailment describes dependencies. By doing something you must also be doing something else i.e. by driving you must also be moving.

Adjectives are stored together in antonyms, i.e. opposites. These are then linked to semantically equivalent words. In some sense these semantically related words are antonyms of their counterparts, which they are stored together with.

Most adverbs are easily derived from adjectives. WordNet relates these adverbs to adjectives. [2] [3]

2.1.1 Application Programming Interface

Several Application Programming Interfaces (API) exists for WordNet. These allow easy access to the platform and often additional functionality. As an example of this the Java WordNet Library (JWNL) can be mentioned. This allows for access to the WordNet Library files. [5]

2. 2 Algorithms

Similarity is the measure, which quantifies the relation between two objects and portrays how much they reflect each other. Similarity is of diffuse character, since comparing two objects is abstract. They might be similar to each other in one regard and different in another. Thus determining similarity is not always a precise science. An approximation of similarity is therefore often the only measure of representing coherence between two objects. One way to represent this is to look at the relation between similarity and dissimilarity. Similarity is what is alike and dissimilarity is that which is different. Dissimilarity can be further divided into what is different and what is opposite. Opposite still carries some dependence on similarity, since they are related in

opposite meanings. Different then becomes a measure of independence. The field of comparing two textual representations of language can thus be represented by indicating if they express the same thing; opposite things or independent things. Some examples of calculating similarity between texts are given below, where the main focus for similarity versus dissimilarity is thought to be in the sense of either similar or not similar.

2.2.1 Edit Distance

Edit distance refers to the class of string similarity metrics, which focuses on finding a measure for either similarity or dissimilarity of two given strings. More specific these algorithms finds the distance between two strings, which amounts to the number of operations required for one string to be transformed into the other string. The algorithms differ in which operations are allowed to transform the strings. Examples of these operations are: Substitution, insertion, deletion, transposition etc. The idea is that different operations can carry different weight, such that certain operations are more costly to perform. The measured distance can then be calculated into a similarity score for the strings. Since the similarity measure is a distance, the algorithms will perform poorly for strings of different size. The magnitude of this poor performance is relative to the difference of size. [6]

These algorithms typically use a bottom-up dynamic programming approach, which computes a distance value for the words in the strings and finds an optimal solution. This is done by dividing the problem into sub-problems and comparing the results of these. Whenever the same sub-problem appears, it is only necessary to lookup the solution instead of re-computing the result. In this respect the result is the comparison between words for all combinations of words. [7]

Examples of different Edit Distance algorithms are; Hamming Distance, Jaro-Winkler Distance[1] and Levenshtein Distance. These are discussed below; more Edit Distance algorithms exists, but will not be discussed.

2.2.1.1 Hamming Distance

The Hamming Distance calculates the number of substitutions required to make two strings of equal length similar. This metric is not very flexible, since it requires strings of equal length and only allows substitutions. This method is not suited for string similarity checks, but it has it domains of use. One example is analysis of bits in telecommunication. [8]

2.2.1.2 Jaro-Winkler Distance

The Jaro-Winkler Distance calculates how many matching's[2] are found in the compared strings and relates them to how far they are from each other, in terms of positioning in the strings. The matching's carry weight if they are not too far from each other. This is

[1] Jaro-Winkler distance is technically not a distance metric, since it does not satisfy the triangular inequality.
[2] Meant as the intersection

also true for transpositions. The larger the distance the more similar the two strings are. Further, the length of the common prefix in the strings is given weight in correlation with a constant scaling factor. This scaling factor should not go beyond a certain threshold. In formal terms the Jaro-Winkler Distance is described as follows:

$$d_j = \frac{1}{3}\left(\frac{m}{|s_1|} + \frac{m}{|s_2|} + \frac{m-t}{m}\right) \quad (2.1)$$

Where:

s_1 and s_2 are the two strings, which are being compared.

m is the number of matching's between the two strings.

t is half the number of transpositions.

Matching's, *m* are only considered if they are within a certain distance of each other. This relationship is given by:

$$\left\lfloor \frac{max(|s_1| + |s_1|)}{2} \right\rfloor - 1$$

This means that in order for a transposition to be counted as a matching it needs to be within the given range above.

The weight each comparison is given is related to the common prefix factor. This can be described as in the following formulation:

$$d_w = d_j + \left(l \cdot p(1 - d_j)\right) \quad (2.2)$$

Where:

l is the length of the common prefix for both strings.[3]

p is the scaling factor giving weight to the common prefix length.[4]

Common use for this algorithm is record linkage for duplicate detection. Given its nature it is mostly suitable for comparison of smaller strings. [9]

2.2.1.3 Levenshtein Distance

Levenshtein distance is the most used algorithm in this category and is often used as a synonym for Edit Distance. The method allows substitution, insertion and deletion. Each operation is typically given a weight of one, and the calculated distance is how many steps that are needed to transform the strings into similar strings. The algorithm is simple, yet it offers more flexibility than Hamming distance. Levenshtein has a wide array of uses. Examples are; spell-checking and fuzzy string searching.

[3] The allowed maximum common prefix should not be above 4.
[4] This value should not exceed 0.25, otherwise it will allow for a distance larger than 1 i.e. more than 100 %

As mentioned above, Levenshtein Distance is a simple algorithm and the most used for comparing similarity in the class of Edit Distance algorithms. The algorithm works by computing a cost table for the minimum required steps to transform one string into another. This is done by running through each string and checking if each letter matches. If it does not match then one of the allowed operations is executed. Before executing the transformation, the algorithm checks if the entry already exists. The distance is then found in the lower right corner of the cost table.

An example of how the algorithm works on the words “written” and “sitting” is as follows:

written → **s**ritten (“w” is substituted with “s”)

s**r**itten → sitten (“r” is deleted)

sitt**e**n → sitt**i**n (“e” is substituted with “i”)

sittin → sitting (“g” is inserted)

This gives a distance of four. There are often different ways to calculate the minimum cost, in which case, one of the solutions is chosen. Only the distance is important and not the path of the transformation.

Since this algorithm compares the syntax of the given strings, it is highly dependent on the word formulation and organization of the strings. If the strings essentially say the same thing, but uses different words to express these meanings and/or the word structure is very different, similarity is not detected.

The algorithm has its use in comparing short strings to larger strings, which is relevant for searches such as those in a search engine or spell-checkers. Here the string is compared by the distance to “popular” search terms and proposes a suggested search term based on relative distance. Another use is in the area where little difference among the compared objects is expected. The threshold for similarity should be set such that it takes the strings length into consideration. The correlation factor should be tolerant for smaller strings and less tolerant for larger strings. [10] Dameru-Levenshtein distance is a version of the Levenshtein algorithm, which allows for transposition of adjacent characters. A higher similarity score can be measured by this method, since a transposition boils an edit of two operations down to just one operation. This can be used in conjunction with a lower associated weight to further heighten the similarity score. [11]

2.2.1.4 Longest Common Subsequence

The Longest Common Subsequence (LCS) finds similarity between two strings, by relating the longest subsequence each have in common. This means that shared words are counted and words in between these are disregarded. A suitably threshold for

similarity between two strings should be chosen in relation to the length of the LCS. The LCS is not always unique. An example of this is the two sequences:

The cow ran away, fast into the distance.

The cow was fast away.

Here the LCS can be both “The cow away” and “The cow fast”.

This algorithm has its uses in bioinformatics, such as detecting mutations in DNA-strings. To name a use in information retrieval, it can be used to detect plagiarism. [12]

2.2.2 Vector Space Model

Statistical similarity refers to the string similarity metrics, which focuses on vectorization of strings or queries, check how often a term appears in the strings and computing their similarity. These algorithms find how much two strings have in common by weighting their communality. The term, which is the measure for the similarity comparison, depends on the given context. It can be phrases, words or keywords. [13]

Examples of different vector space models are; Cosine Similarity, Term Frequency-Inverse Document Frequency (TF-IDF), Jaccard Index and Dice’s Coefficient. These are described below; more Vector Space Models exist, but are not described.

2.2.2.1 Cosine Similarity

Cosine similarity measures the similarity in terms of comparing the cosine angle between two vectors (strings). The result is a number between minus one and one. One corresponds to similar and minus one corresponds to dissimilar. Zero corresponds to the vectors being unfamiliar and has nothing to do with each other. The result can be derived from the Euclidian dot product (2.3), where the term frequency is compared to the magnitude of the comparison. This is formalized in the following manner:

$$\cos(\theta) = \frac{A \cdot B}{|A|\,|B|} \tag{2.3}$$

Where:

A and *B* are vectorizations of text.

A similarity score can be computed by comparing the difference of the angel between a set of strings. This is done by comparing the intersection and the union of a set of strings. Specifically this is done by the above mentioned vectorization of the strings i.e. the union between a binary occurrence vector and the frequency occurrence vector over an allowed alphabet for the given string. The resulting frequency occurrence vector is then a representation of the string as a set. If these sets are orthogonal in relation to each other there are no matches, thus they have nothing in common. If the vectors are parallel (and facing in the same direction) they are similar.

Uses of this algorithm are typically in the domain of text and data mining. [14]

2.2.2.2 Term Frequency-Inverse Document Frequency

The TF-IDF measures how often certain terms appear in strings and bases the similarity score on how many matching's it encounters, hence *term frequency*. The Inverse Document Frequency part of the algorithm ensures that terms which are common are not counted as heavily. This weights the importance of terms in context of the comparison. Mathematically this can be expressed by:

$$TF(t,d) \times IDF(t,D) \tag{2.4}$$

Where:

TF(t, d) is the term frequency of a given term, *t* in a given string, *d*.

IDF(t, D) is the inverse document frequency of a given term, *t* in the collection of strings, *D*.

The TF(t,d) is found by term counting in a specific string. This value is typically normalized, since larger strings statistically have a higher term count regardless of the importance of the given term. The *IDF(t, D)* is found by taking the total number of strings that are compared and dividing it by the number of strings containing the specified term. The logarithm of this value is then the *IDF(t, D)*.

This algorithm can be used to rank strings in relation to relevance or it can be used to filter stop words, which will be discussed later. The relevance factor can be taken as a measure for similarity, given the right terms. The algorithm works best when comparing larger sets of strings with each other [16]

2.2.2.3 Jaccard Index

The Jaccard Index measures similarity in terms of the relation between the intersection and union of two sample sets:

$$S_j = \frac{|A \cap B|}{|A \cup B|} \tag{2.5}$$

Dissimilarity, which is called The Jaccard Distance, is found by subtracting The Jaccard Index from 1. This dissimilarity is a measure of differences i.e. how independent the two strings are. Uses for this algorithm are measurements of similarity in binary data.

2.2.2.4 Dice's Coefficient

Dice's Coefficient also known as Sørensens Similarity Index has a lot in common with the Jaccard Index, but weights matching's twice, compared to The Jaccard Index. The Dice Coefficient can be used in the domain of information retrieval and other areas.

Dice's Coefficient measures similarity over sets in the given way:

$$s = \frac{2\,|A \cap B|}{|A| + |B|} \tag{2.6}$$

For a similarity measure between strings the total number of bigrams which the strings have in common, can be used. Such a method is called *n*-gram. [16]

2.2.2.5 N-gram

The n-grams for each string is computed and then compared to each other. The *n*-gram can be unigrams, bigrams, trigrams, etc. These are neighbouring elements in a sequence, where *n* specifies the number of elements that are combined. A similarity score is calculated for each *n*-gram the strings have in common, using Dice's Coefficient.

This is done in the manner of the example below:

The bigrams of meditate and meditation are {me,ed,di,it,ta,at,te} and {me,ed,di,it,ta,at,ti,io,on} respectively. This gives the following similarity score:

$$\frac{2 * 6}{8 + 9} = 0.706$$

There are 6 matching bigrams and a total of 8+9 bigrams. Thus the two words/strings are *70.6 %* similar.

By comparing bigrams this method becomes unaffected by word order and displays a high similarity with strings that have little difference. The method is language independent, since it only compares character order (syntax). [17]

2.3 Natural Language Processing

Natural Language Processing (NLP) is a large field which encompasses a lot of categories that are related to this thesis. Specifically NLP is the process of computationally extracting meaningful information of natural languages. In other words: the ability for a computer to interpret the expressive power of natural language. Subcategories of NLP which are relevant for this thesis are presented below.

2.3.1 Stop Words

Stop words are words which occur often in text and speech. They do not tell much about the content they are wrapped in, but helps humans understand and interpret the remainder of the content. These terms are so generic that they do not mean anything by themselves. In the context of text processing they are basically just empty words, which only takes up space, increases computational time and affects the similarity measure in a way which is not relevant. This can result in false positives.

Stop words is a broad term and there is no precise specification of which words are stop words. To specify if a given word is a stop word, it has to be put in context. In some situations a word might carry relevance for the content and in others it may not. This is defined by the area in which the content resides. A stop word list should thus be chosen such that it reflects the field which is being analysed. The words in such a list should be filtered away from the content in question.

Examples of common stop words are; *a, the, in, is, at, that* etc. Search engines typically filter these very common stop words out. This can cause problems when searching for phrases where these words are present.

2.3.2 Stemming

Stemming is the process of reducing an inflected or derived word to its base form. In other words all morphological deviations of a word are reduced to the same form, which makes comparison easier. The stemmed word is not necessarily returned to its morphological root, but a mutual stem. The morphological deviations of a word have different suffixes, but in essence describe the same. These different variants can therefore be merged into a distinct representative form. Thus a comparison of stemmed words turns up a higher relation for equivalent words. In addition storing becomes more effective. Words like observes, observed, observation, observationally should all be reduced to a mutual stem such as observe.

There are a lot of proposed methods for finding stems. Some examples are; lookup tables, suffix-stripping, affix stemming and lemmatisation. Stemmers can both be language dependant and independent. This is based on how the relevant stemmer is implemented. A lot of work has been put into the area of stemming; some of the more popular stemmers are Porter and Snowball. [18]

2.3.3 Part-of-speech tagging

Part-of-speech (POS) tagging is the field which is concerned with analysing a text and assigning different grammatical roles to each entity. These roles are based on the definition of the particular word and the context in which it is written. Words that are in close proximity of each other often affect and assign meaning to each other. The POS taggers job is to assign grammatical roles such as nouns, verbs, adjectives, adverbs, etc. based upon these relations. The tagging of POS is important in information retrieval and generally text processing. This is the case since natural languages contain a lot of ambiguity, which can make distinguishing words/terms difficult. There are two main schools when tagging POS. These are rule-based and stochastic. Examples of the two are Brill's tagger and Stanford POS tagger, respectively. Rule-based taggers work by applying the most used POS for a given word. Predefined/lexical rules are then applied to the structure for error analysis. Errors are corrected until a satisfying threshold is reached. Stochastic taggers use a trained corpus to determine the POS of a given word. These trained corpuses contain pre-tagged text, which define the correct POS of a given word in a given context. The corpuses are vast enough to cover a large area, which defines different uses of terms. The stochastic tagger retrieves the context of the word in question and relates it to the trained data. A correlation is found by statistical analysis upon the use of the word in the trained data. This means that the content of the trained corpus greatly influences the outcome. Trained corpuses should thus be picked, such that reflection of the field they are trying to tag is maximal. Current taggers have a success rate above the ninety-seven percent mark. This is to the extent where even

some linguists argue, which is the correct result. It can thus be concluded that these taggers exhibit near human results. [19]

2.3.3.1 Treebank

Treebank's uses a way of marking syntactic structure to a sentence. An example of how this is done is given below:

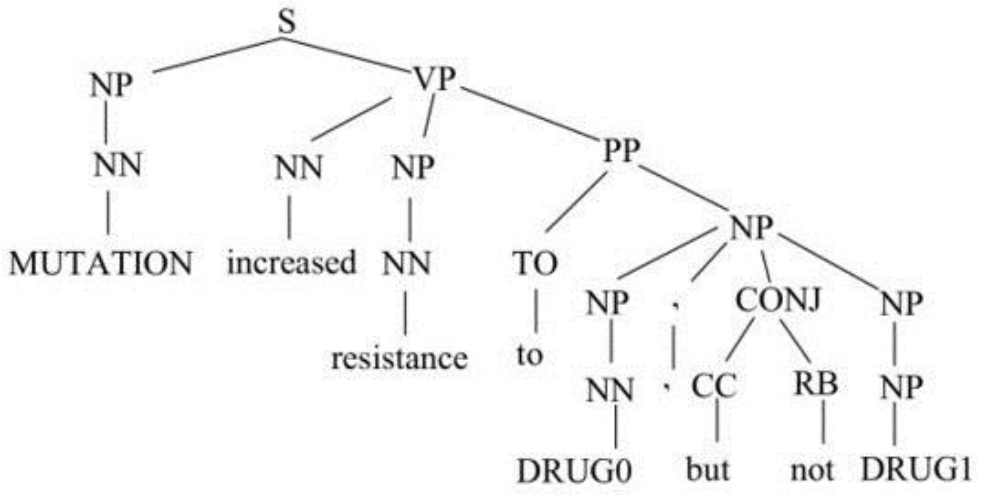


***Figure 2.1**: Illustrating Treebank structure.* [20]

These tree structures can be used to identify common grammatical structures. This can be correlated with given word structures to identify expected sentence structures and thus the meaning of the sentence.

Most POS taggers use the Penn Treebank tag set to apply which tag has been found most relevant for the appropriate word. A list of these tags can be found in Appendix A. The Penn Treebank is a collection of parsed sentences. It is an example of the above described trained corpuses. [21]

2.3.4 Named Entity Recognition

Named Entity Recognition (NER) is the task of categorizing atomic elements in a given text. These could be specific persons, groups, companies, locations or expression of time. Approaches for NER-systems are similar to that of the POS taggers. These systems are great for determining if a word is a proper noun. This is particularly important in comparison tasks where wording contains ambiguity. An example of this is "The Who", which is a music group. Here "The" would normally refer to the determiner, but is part of the proper noun "The Who". [22]

2.3.5 Word Ambiguity

As mentioned earlier words can have multiple meanings, also called polysemy. The ability to distinguish word senses in NLP is very relevant, in the sense that a reference to a word is handled in the right manner. It is intuitive for humans to disambiguate words,

since humans correlate all known senses of a word with a given word and derive the correct meaning by comparing context. This is done instinctively and the result is computed very fast. Humans are aided by grammatical rules, which prime the individual for an implicit sense. These grammatical rules are not needed, but reduce computational time, since a priming event has occurred. Some structure is still needed, enough for putting the input into context and then referencing it through the network of known senses.

The process of disambiguation is carried out by comparing a list of senses with a text formulated from a language. All of the content in the text is cross referenced with all the possible senses in accordance to the structure in the text and the most likely sense is computed, comparing relevance. Approaches to disambiguating words are similar to those of POS tagging. The methods with highest success rate use a statistical approach with supervised learning. A corpus of trained data set, with pre-tagged senses for words in specific situations is used to attribute word senses in a given text. The appearances of words in the text to be disambiguated is looked up and compared to the trained corpus. Statistical analysis finds the highest coherence between the context of the word in question and context in the trained corpus. The word is then assigned the sense for the given context. In other words all words are looked up and checked, by looking for instances where the word is used in a similar manner. The highest coherence in the way the word is being used is picked as a candidate for the sense of the word. Success rates for word sense disambiguation algorithms are not as high as those of POS taggers. Depending on the specificity of the senses, typical success rates are lower than seventy-five percent, due to the very nature of senses. Senses are part of natural languages and are thus given different weight in connection to how people perceive these senses. This is based on the individual's relationship with the given sense and which associations are made, depending on which situations the sense has been encountered. Another approach is to directly compare the word in the context it is written. This method is described by the Lesk algorithm. [23]

2.3.5.1 Lesk Algorithm

The Lesk Algorithm assumes that words in close proximity of each other share a common topic. Words can then be disambiguated by comparing all senses of a word with senses of neighbouring words. An often used example to illustrate this behaviour is for the context of "pine cone":

Pine has two meanings, namely:

1. Kinds of evergreen tree with needle-shaped leaves
2. Waste away through sorrow or illness

Cone has three meanings, namely:

1. Solid body which narrows to a point
2. Something of this shape whether solid or hollow
3. Fruit of certain evergreen trees

The senses are then compared and it is seen that both words share a sense, where "evergreen tree" is part of the description. Thus the senses with the highest overlap are chosen.

This is a very simple procedure. Points of critique for this algorithm are the dependency in formulation of the different senses and low number of comparisons.

The algorithm can however be improved by means of comparisons. This is done by looking at the relation between words and maximizing the possibility for a match: One way is to also consider synonyms and their senses. [24]

To score the overlaps in the definition of senses, Adapted Lesk Algorithm can be used. This relates the number of overlaps to the number of consecutive overlaps, thus giving higher weight to consecutiveness. [25] This is in correspondence with Zipf's law, which states that smaller words are used more often. Consequently smaller overlaps are more common and therefore transmit less importance. [26] This can be carried out by using Longest Common Subsequence with consecutiveness. The score of consecutiveness can be denoted by x^2 where x is the number of overlaps.

2.3.6 Ontology

Ontology is here used as the "*specification of conceptualization*" and not in the philosophical way. Ontology can be described in the following way:

"the objects, concepts, and other entities that are assumed to exist in some area of interest and the relationships that hold among them" (Genesereth & Nilsson, 1987)

The relations described in WordNet can thus be seen as ontology. Examples of these are the synonym and hypernym relations described earlier. Since concepts are connected, similarity can be measured through the study of the relationships between these concepts. An elaborate comparison can be done by expanding search through the relationships between concepts. At some point a unification of concepts is reached, since concepts start relating back to previous concepts. All of these concepts are related in some manner and a comparison of similarity can be done on this union. The similarity between two strings is measured by the comparison of the generated union for each entity of both strings. The degree of intersection is then a measure for similarity. This of course expects a high degree of interconnectedness between concepts and assumes that all related concepts are connected, which is not always the case as in the WordNet data structure. [27]

CHAPTER 3

Design

This section covers the selected approaches to determine similarity between two given strings of text, chosen for implementation.

Syntax is quite easy for a computer to understand. This is due to the fact that syntax is based on rules, which can be written down and followed. These rules can be passed to the computer, which in turn can relate these rules to a given context and check for consistency in the given structure. Most of the algorithms in the analysis are of syntactic nature, since they conduct similarity based on word representation. This means that they expect the words to follow certain syntax and are thus able to do a direct comparison based on this assumption. On the other hand semantics are quite hard to represent in a complete way by means of a data structure. Therefore a computer has a hard time understanding semantics. Semantics carry the meaning and relationships between words and attribute different weight of these relations. As described in the analysis, the weight of these relationships is not always agreed upon, and consequently makes it hard for a computer to interpret these fuzzy definitions. The difference is illustrated by comparing an apple with an apple, which is straightforward since they are both apples. This resembles syntactic similarity. However comparing an apple with an orange, resembling semantic similarity, is much harder. A lot of factors have to be taken into consideration. The question then becomes whether or not it is even possible to compare an apple to an orange in the complete sense. This is one of the reasons why similarity is talked about in the approximate similarity sense, as discussed earlier. The two can easily work together and generate a more reliable result. This is the way humans use them.

The algorithms described in the analysis are categorized into different approaches; from Vector Space Models and Edit Distances to Ontology based approaches. Within each category they all work in a similar fashion. It would thus be interesting to see how each category compares to one another. For this reason one algorithm from each category is chosen for implementation.

3.1 Levenshtein Distance

From the class of Edit Distance algorithms, Levenshtein Distance is chosen. Levenshtein Distance is often used for similarity checks, given its simple nature. It also exhibits the general idea, represented by the Edit Distance class of algorithms, quite well. It is not too simple like Hamming Distance, but still not too far away, since transposition is not allowed. Hence Levenshtein Distance is a good representative for this class. A major influence for picking this algorithm is how widely it is used. It seems to be the go-to algorithm for measuring similarity via distance. The Jaro-Winkler Distance also proposes an interesting measure for similarity, but is not chosen, since Levenshtein Distance is simpler in nature. It will be interesting to compare a simple algorithm to a more complex one in regards to how well they perform.

3.2 N-gram/Dice's Coefficient

From the class of Vector Space Model algorithms, n-gram in combination with Dice's Coefficient is chosen. Vector Space Models are in general more robust towards changes in word order, compared to Edit Distance algorithms. This is the case, because they count the number of occurrences in the text. The words are taken out and compared in relation to overlap. The intersection of these overlaps is computed with disregard to their placement in the text, in contrast to Edit Distance which evaluates from the given position in the text. An easy to understand example of this behaviour is the relation of *n*-grams between two texts. The *n*-grams for each text is computed and then compared. Each match corresponds to resemblance and therefore contributes weighting towards a higher total similarity result. After each match the corresponding *n*-gram is removed from the text being compared. This is done to avoid re-runs counting multiple times. The relation between matching's and non-matching's can be calculated by using Dice's Coefficient, for a similarity score. The encountered matching's counted double, one for each text, over the total area.

The Term Frequency-Inverse Document Frequency algorithm has not been chosen, since the special functionality which it exhibits is almost redundant because the use of stop words. It is not totally redundant, since it gives weight to the terms, in contrast to stop words which are just removed. Apart from this behaviour terms are just counted and related to the text size. To be more precise the algorithm requires a larger set of texts to compare, which makes it inappropriate to compare only two given texts. The *n*-gram is able to compare parts of a word, which detects words that are alike. The TF-IDF could be used together with *n*-gram, to rate often occurring *n*-grams lower for common *n*-grams and higher for rare *n*-grams. However this approach has not been chosen. The Cosine Similarity is an interesting algorithm, but has not been chosen for implementation due to thesis delimitation. The Jaccard Index is similar to that of Dice's Coefficient, hence it is not interesting to implement.

3.3 Ontology

The Ontology based approach is a lot more complex than the above proposed rivals. There is an endless amount of methods which can be used to improve the related similarity score that the ontology approach generates. Some of these methods are described below in relation to WordNet, since this is the Ontology which will be used. This is the main focus of this thesis, thus most of the effort has been put into this area.

3.3.1 Stop Words

Described in the terms of the analysis, stop words are basically just filling which is not needed, in the field of textual comparison. WordNet does not contain the most common stop words, thus no comparison between these are made. To reduce the amount of computational executions required in order to compare two given texts, stop words should still be removed.

3.3.2 Stemming

WordNet stores words by their dictionary form (lemma). Hence it is necessary to retrieve the lemma of a given word in order to look it up, using WordNet. Since stemming is done in regards of looking up a word in WordNet for later use, the process of the stemming is important. As described in the analysis the various methods of stemming may not always reduce the word to its morphological root. A method should be chosen such that words are only stemmed to their morphological root. WordNet handles this problem by including a morphological processor, which returns a word to its base form using lemmatization. In this regard words are looked up in WordNet, which computes their base form and checks for existence in the WordNet database. If there are no matching's between the base form of a given word and the lemmas in WordNet, then no stemming should be performed. This is due to the fact that the word might carry a special meaning in the form it is presented, which can be used for a direct comparison.

3.3.3 Similarity

WordNet provides a great basis for associating words among themselves. To compensate for some of the pitfalls in syntax comparison, semantics can be used. Two texts can discuss the same topic without using the same words. Syntax similarity measures do not cope well with this category. An easy way to check if two given texts are discussing the same topic is to check synonym relations. If they are discussing the same topic, they are bound to have some overlap in their synonym relations.

3.3.3.1 Word Disambiguation

To compare words, a data structure which relates the associations between words is needed. This is provided by WordNet. The Ontology of WordNet describes the relationships between words in the manner shown in Table 2.1. The represented relations depend on the POS of the given word. This means that the words of the texts in question need to be defined in terms of their POS. This is done by using the Stanford POS Tagger, which iterates through a text and assigns POS two words. A stochastic

method is used to achieve this, which is described in the analysis section. The texts should now be ready for a comparison using the WordNet Ontology, since they are tagged with POS. By further examination it is shown that this is not the case. WordNet has a lot of different senses for words within a given POS. Therefore, each POS word has to be disambiguated. The analysis presents two solutions to disambiguate words. One way is to compare the word and its context with a corpus of trained data. This approach is not used, since this requires training a dataset, which could be a whole thesis in its own regard. Systems which disambiguate words exist, but are not used because they do not offer much insight into the whole area of disambiguation, other than statistical analysis. The Lesk algorithm is therefore an interesting approach, offering insight into how disambiguation can be handled.

3.3.3.1.1 Lesk Algorithm

Each sense of a word is compared to neighbouring words and all of their senses. Here the sense with the highest overlap in the definition of the senses is chosen, as described in the analysis. To maximize likelihood of a correct/optimal result, all of the relations between senses should be considered. This means when a sense is encountered it should be compared to all other senses in the compared text. This evaluation should consist of relating the hypernym/hyponym relationships, the holonym/meronym relationships and coordinate terms for noun senses, which WordNet states. The same is true for the other POS. Verb senses can correspondingly be checked for relatedness through the hypernym/hyponym relationship. To maximize success the other relationships should also be considered. This is done (for verbs) by checking troponyms, hypernyms, entailment and coordinate terms for both the compared senses and cross-referencing all these relations for relatedness. This can be continued in a manner such that troponyms of troponyms are checked for a match by means of comparing hypernym/hyponym, entailment, troponym etc. relationships. It is evident that this becomes very complex and should only be done to a certain extent before it all becomes too tangled and too far away from the root word. This idea is hard to grasp, since it is of such a vast dimension. This could also be done to adjectives and adverbs, with the relationships they are represented with. This thesis is delimited to only deal with comparison of sense definition for all senses of the word in question and all senses of neighbouring words. This means that relationships of senses and their further relationships are not taken into account in regard to the sense definition and therefore word disambiguation.

3.3.3.2 Word Relatedness

With the sense of each word in the texts defined, a relation between each word in the texts can be found. The exact position of the given sense in the WordNet data structure can be found and related to the compared sense. A similarity score for these relationships can be computed, by measure of distance between the two. For the sake of simplicity only one of these relationships is enough to evaluate relatedness. For this the hypernym/hyponym relationship can be chosen. To further improve relatedness, the methods described to disambiguate senses can be used to further investigate the

relation between the final senses. This relates to the approach described in the ontology section of the analysis. This approach will not be implemented, again due to delimitation of the thesis.

3.3.3.3 Matching

The problem of finding the final similarity score between the two texts is boiled down to solving the problem of finding the maximal matching for a bipartite graph. In other words the similarity score is found by combining word pairs of two sets, where the resulting weight is maximal.

3.4 Language

The chosen language which the program will accept will be English. This is due to the restrictions of WordNet, which operate with English words. It is possible to get WordNet with different languages, but it is still restricted to one language at a time. For the program to be able to handle multiple languages, multiple instances of WordNet would have to be loaded and the languages would have to be analysed in order to specify which instance to operate on. The English version of WordNet is by far the most elaborate version of WordNet and is thus a good candidate for this thesis.

The algorithms which concentrate on syntactical similarity are still language independent, since they compare the exact content. For language independence, stop words and stemmers may not be used in cooperation with these algorithms, unless a language independent stemmer that does not confine by the rules of a certain language is used.

CHAPTER 4

Implementation

This section documents the implementation of the program and explains the result of following the approaches specified in the Design section (chapter 3).

4.1 Model-view-controller

The model-view-controller framework [31] has been followed in the construction of the program. This separates the different parts of the program in how they interact with the user. This supplies a good overview over the classes in the program and how they are used. Others who may be interested in the program or continue work on the program can more easily get acquainted with the program structure.

The model part of the program is where the data is stored and all the computation is done. This encapsulates the program state and allows actions to be made.

The view part of the program shows the data representation of the program to the user. It allows the user to see the model and choose actions based upon the program state. A typical example of this is a graphical user interface.

The controller is the binding link between the view and model part of the program. It allows actions of the user to be performed on the data representation of the model or view. The behaviour of the program and which actions are allowed is specified by the controller. In a way it hands out commands to be followed by the view or model.

A class diagram can be found in Appendix B. The content of this diagram is explained below.

4.2 gui-package

This package contains the view part of the program. This is split into two classes; *Driver.java* and *Frame.java*.

4.2.1 Driver.java

This class contains the main method of the program, thus it is the class which initializes the program. It tries to use the default look and feel of the operating system and creates an instance of the *Frame.java* class.

4.2.2 Frame.java

This class defines the main graphical user interface for the program. It assigns all the attributes for the constructed frame. This is done by using a *GridBagLayout* in which all the graphical components are put. The implemented layout is simple with few elements and should be intuitive to use. Elements are placed such that they relate to the area they are in. As an example, the button to load the first text is placed under the *JTextArea* (textbox), which will display the content of the first text and is labeled “First text”. Listeners to all the components, which allows for interaction are added, in this class. These are *ButtonListeners* for the *Buttons*, *MenuListeners* for the Menu and *ActionListener* for the *JComboBox* (dropdown menu).

4.3 control-package

This package contains the controller part of the program. This is split into three classes; *MenuListener.java*, *ButtonListener.java* and *SimilarityScore.java*. These allow the user to interact with the core of the program. Users are through these allowed certain requests and are able to see the results of these requests.

4.3.1 MenuListener.java

This class implements the *ActionListener* class and specifies the actions for each menu entry. If the menu item “New” is chosen, a method, which wipes all relevant data for comparing two strings in the program, is called. The user window is reset to how it looked when the program was first started. If the menu item “Exit” is chosen, the program is simply closed.

4.3.2 ButtonListener.java

This class implements the *ActionListener* class and specifies the actions for all the buttons in the program. This is the main way the user interacts with the program. In addition, the requests to the dropdown menu are also handled through this class.

The class contains a method for resetting all values in the program and a method that handles events, triggered by the buttons. The *ResetValues()* method returns all the labels, the textboxes and the dropdown menu to their initial state and resets the text files in the program. The method *actionPerformed()* holds all the actions for each button, which is “First text”, “Second text and “Process”. Actions for “First text” and “Second text” are similar, but loads text into different places. A *JFileChooser* is created which allows the user to load a text file into the program. The File is processed and saved as a *String*. All the relevant labels, displaying attributes of this file, are updated. “Process” checks which algorithm the user has chosen, based on the state of the

dropdown menu and makes a call to the relevant algorithm, with the first text and the second text as parameters.

4.3.3 SimilarityScore.java

This class functions as an intermediary class, which holds methods to compute the similarity score for all the specified algorithms and passes the texts as parameters to the relevant algorithm. The supported algorithms are Levenshtein Distance, Dice's Coefficient (bigram) and Semantic Similarity (Ontology based).

The method *levenshteinDistance()* takes two strings and makes a call to the constructor of the *Levensthein.java* class, which computes the Levenshtein Distance (discussed later) of two given texts. After retrieving the Levenshtein Distance, the similarity score is calculated by the following code:

```
similarityProcentage = ((longestText-LD)/longestText)*100;
```

Where *longestText* is the length of the longest of the two given texts and *LD* is the Levenshtein Distance.

The bigram similarity is computed by the method *diceSimilarity()*, which makes a call to the constructor of the *DiceCoefficient.java* class (discussed later). The retrieved score is multiplied with one hundred to obtain the percentile difference in similarity.

The method *semanticSimilarity()* computes the Semantic Similarity score. This is done by calling the constructor *SemanticRelatedness()* of the *SemanticRelatedness.java class*, which gives a score based on how well the words of the two given strings relate to each other (discussed later).

All the above computed scores are displayed in the "result" label of the *Frame.java* class. The scores thus shown to the user.

4.4 filters-package

This package contains the portion of classes in the model part of the program, which concerns itself with filtering of the given text. This is split into four classes; *ReadFile.java*, *Tokenizer.java*, *StopWords.java* and *Stemmer.java*. These classes all prepare the given text in a manner that organizes the text for later use.

4.4.1 ReadFile.java

This class contains two methods; one for processing a text file and one for counting the number of words in a given string. The computed text string is constructed by a *StringBuilder* and is used later for text similarity.

4.4.2 Tokenizer.java

This class consists of tree methods; one for putting the elements of a string into a list, one for simplifying the POS tag of a word and a private method for simplifying POS tags. The first method *tokenizeText()* works simply by splitting elements around whitespaces

and putting them into an *ArrayList* of strings. The second method *tokenizePOS()* works by splitting each word and its corresponding POS tag into a row in a two dimensional array. Each of the POS tags are then simplified with the assistance of a private helper method called *CorrectPOSTags()*.

4.4.3 StopWords.java

This class includes only one method; which runs through a list of words and removes all occurrences of words specified in a file. A text file, which specifies the stop words, is loaded into the program. This file is called "stop-words.txt" and is located at the home directory of the program. The text file can be edited such that it only contains the desired stop words. A representation of the stop words used in the text file can be found in Appendix C. After the list of stop words has been loaded, it is compared to the words in the given list. If a match is found the given word in the list is removed. A list, stripped from stop words, is then returned.

4.4.4 Stemmer.java

This class contains five methods; one for converting a list of words into a string, two for stemming a list of words and two for handling the access to WordNet through the JWNL API. The first method *listToString()* takes an *ArrayList* of strings and concatenate these into a string representation. The second method *stringStemmer()* takes an *ArrayList* of strings and iterates through each word, stemming these by calling the private method *wordStemmer()*. This method checks if the JWNL API has been loaded and starts stemming by looking up the lemma of a word in WordNet. Before this is done, each word starting with an uppercase letter is checked to see if it can be used as a noun. If the word can be used as a noun, it does not qualify for stemming and is returned in its original form. The lemma lookup is done by using a morphological processor, which is provided by WordNet. This morphs the word into its lemma, after which the word is checked for a match in the database of WordNet. This is done by running through all the specified POS databases defined in WordNet. If a match is found, the lemma of the word is returned, otherwise the original word is simply returned. Lastly, the methods allowing access to WordNet initializes the JWNL API and shuts it down, respectively. The *initializer()* method gets an instance of the dictionary files and loads the morphological processor. If this method is not called, the program is not able to access the WordNet files. The method *close()* closes the dictionary files and shuts down the JWNL API. This method is not used in the program, since it would not make sense to uninstall the dictionary once it has been installed. It would only increase the total execution time. It has been implemented for good measure, should it be needed.

4.5 algorithms-package

This package contains the remaining model part of the program, where all the "heavy lifting" is being done. This is split into eight classes; *DiceCoefficient.java*, *Levenshtein.java, POSTagger.java, MostLikelySense.java, Disambiguity.java, SemanticDistance.java, HeuristicAlgorithm.java, and SemanticRelatedness.java.* This

collection of classes is concerned with computing the data which lays the foundation for comparing two texts.

4.5.1 DiceCoefficient.java

This class contains three methods. The constructor computes bigrams of two given texts and compares these to get a similarity score. The constructor is assisted by two private methods. The last method allows the retrieval of the computed similarity score. The first method *bigramDiceSimilarity()* takes two strings of text and computes their bigrams. Each bigram pair is then compared and counted as a weight if a match is encountered. The matched bigram is removed and the process is continued. The total similarity score is computed by the following code:

```
diceSimilarityScore = (2.0*communality)/totalTextLength;
```

Where *communality* is the number of matching bigrams and *totalTextLength* is the total number of bigrams of the two given texts.

This corresponds to Dice's Coefficients formula (2.6). The above used bigrams are computed by making a call to the private method *totalBigrams()*. Here, each word of the given text is passed to the private method *wordBigrams()*, which calculates the bigrams for a given word. All the bigrams are returned in an *ArrayList* of strings, which is handled in the above explained manner. The final similarity score can be retrieved by calling the method *getDiceSimilarityScore()*.

4.5.2 Levenshtein.java

This class consists of two methods. The constructor computes the number of operations required to transform one of the given strings into the other, i.e. the distance between the two given strings. The first method gets the minimum value of three given values. The last method allows retrieval of the distance value computed in the constructor.

The constructor takes two strings of texts and initializes two cost arrays with the length of the given text. A helper array is also created to swap these arrays later. The initial cost array is set to the highest possible cost. Each character of both texts are run through and compared to each other. The current comparison is compared to the cost array specifying the position to the left, up and diagonal left position to the given character. The best transformation option is chosen in this manner. The value is passed to the cost array and the algorithm continues to the next character comparison, after updating the cost arrays, in order to associate the cost to the next comparison.

4.5.3 POSTagger.java

This class consists of only one method, which takes a string of text and assigns POS tags to the elements of the text, using the Penn Treebank Tag Set shown in Appendix A. The method *tagString()* loads a tagger by using the Stanford POS tagger API. This is done by loading the *left3words* tagger which is located in the tagger folder of the home directory of the program. This tagger consists of a trained tagger, which is trained on the Wall Street Journal (section 0-18) part of the Penn Treebank and some additional training

data[5], thus making it good for general purpose tagging. The tagger works by comparing the 3 words to the left of the given word. This tagger has roughly a success rate of ninety percent for unknown words. [28] The tagger is applied to the given text and a tagged text is returned, where each word is connected to its POS tag by a slash ("/").

4.5.4 MostLikelySense.java

This class is a helper class and specifies a way to store a synset and its related weight together. This structure is used by the *Disambiguity.java* class to determine the sense with the highest overlap of synset definitions. Two methods allow retrieving either a given synset or the associated weight.

4.5.5 Disambiguity.java

This class includes three methods. The constructor computes all senses of a given word and its surrounding words from an array of words. The first method compares these senses pairwise by their definition. The second method processes these definitions, enabling them to be compared and finally a third method, which allows retrieval of the disambiguated words.

The class constructor takes a two dimensional array, which consists of words and their corresponding POS. A check is made to see if the JWNL API is initialized, if not, a call to *initializer()* in the *Stemmer.java* class is made. Each word is looked up in WordNet and all senses for the given word are retrieved. The requests for looking up adjectives and adverbs are commented out at the time of writing, since only the hypernym relation between words are looked up for comparison. Loading the adjectives and adverbs will only lessen the similarity score, since the relationships of these are not considered. If the word is not found in WordNet, the algorithm skips to the next word. For each of these words, the senses of the *k*-nearest words are looked up. This is done by the following code:

```
int l = Math.min(i+k, wordAndPOS.length)

            for(int j = Math.max(i-k, 0); j < l; j++)
```

Where *wordAndPOS.length* is the number of words in the array passed to the constructor and *i* is the current position of the given word in the array.

As for the word above, the nouns and verbs of these *k*-nearest words are looked up and all their senses are retrieved. The *k*-nearest word is skipped, if its position is identical to that of the given word. *A* call to the private method *findMeaningOfWord()* is made passing the senses of each word pair. This method gets the definition for each of the senses and finds the senses with the highest overlap in the definition of the senses. Before the sense definitions are compared, a call to the private method *processDescription()* is made, passing the sense definitions. This method removes stop words and stems the definition of a given sense. The overlap is computed by a weighting

[5] Information found in the README file of the tagger.

function that increases the weight by one for each match in the definitions. *findMeaningOfWord()* returns the most likely sense of the given word and its associated weight in the structure defined in the *MostLikelySense.java* class. As specified above, this is done for each word pair of the given word and the *k*-nearest words. These are put into an *ArrayList* of *MostLikelySense*'s, where the sense with the highest weight is picked as the final sense of the given word. This sense is put into an *ArrayList* of synsets, which can be retrieved by making a call to the method *getDisambiguatedWords()*.

4.5.6 SemanticDistance.java

This class contains two methods. The constructor of the class calculates the distance between all the elements of two sets, which consist of a list of synsets. The first method finds the distance of two given synsets in WordNet. The second method allows retrieval of the distance matrix computed by the constructor.

The constructor takes two *ArrayList*'s of synsets and runs through each of these synsets. For each pair of synsets between the two lists, a relation is found by finding the distance between these in WordNet. This is done by making a call to the private method *synsetDistance()*, passing the two given synsets. If a relation is found between the two synsets, a score for their relatedness is computed by the following code:

```
tempDistanceMatrix[i][j] = 1.0/distance
```

Where *distance* is the "depth" between the two synsets in WordNet.

The similarity score is thus the inverse of the distance between two given synsets. A matrix is filled with these similarity scores, representing the score for each pair of synsets. *synsetDistance()* checks if two given synsets are the same. This is done because there is a problem when the two synsets are identical. For some of these identical synsets the returned distance between them is zero[6]. It is not known if the problem lies in the JWNL API or in WordNet, but the workaround consisted of returning the distance 1 if the synsets are identical. If the synsets are not identical, a relation is attempted to be found by looking up their hypernym/hyponym relationship in WordNet. Several relationships between two given synsets can exist, but the one with minimum distance is always chosen. The depth of this relationship is returned, i.e. the distance between the two synsets. The method *getDistanceMatrix()* allows retrieval of the distance matrix, computed by the constructor.

4.5.7 HeuristicAlgorithm.java

This class contains a heuristic method, which finds a best matching between two sets, given their score matrix. The constructor takes a two dimensional array of doubles and runs through it. The highest score in each row is picked as the best candidate and the corresponding element is removed. All of the scores from the matching are added together and the final score is stored. This score can be retrieved by the method *getSimilarityScore()*. The Hungarian algorithm[32], which finds a maximal matching, was

[6] The default distance between two identical synsets in WordNet is one.

tried but some problems occurred with the implementation. Because of time constraints, the cause of the problem was not found, and a heuristic function was constructed as an alternative. The heuristic function just finds a best matching and not a maximal matching.

4.5.8 SemanticRelatedness.java

This class contains three methods. The constructor of the class calculates the similarity score between two texts. The first method prepares a given text for a similarity comparison, running it through a series of steps, transforming the text. The second method calculates a maximal matching between two sets and merges it into a combined similarity score.

The constructor takes two strings and processes each by making a call to the private method *processText()*, passing the given string. This method returns a list of disambiguated words, which is done by running the text through a series of steps. These steps are:

Tokenize text → remove stop words → stem remaining words → convert back to string → POS tag string → simplify POS tags → disambiguate words.

These disambiguated words are represented as synsets, which makes it easy to look up in WordNet for later use. All of these steps are describe in the classes above. The lists of found synsets for both texts are passed to the *SemanticDistance.java* class. As mentioned, the constructor of this class computes a distance matrix of similarity scores, between each synset pair for the two sets. This distance matrix is passed to the private method *semanticSimilarityScore()*. Here, the distance matrix is passed to the *HeuristicAlgorithm.java* class, which calculates a summarized score for the best matching's as described above. The score is used to calculate the final similarity score, by using Dice's Coefficient (2.6). The magnitude is the total number of synsets, used for the comparison.

CHAPTER 5

Expectations

This section describes the expected performance of the program. The important parts of the program are investigated and evaluated by their performance. These parts are the different algorithms. Performance is mentioned in the regards to the complexity of the constructed code and how reliable the program output is, in terms of the similarity score results.

5.1 Complexity

It is relevant to investigate the three algorithms to calculate the similarity between two given texts, which are presented by the program. This is the case since they carry out the main functionality of the program. The three algorithms as presented by the program are: “Levenshtein”, “Dice’s Coefficient” and “Semantic Similarity”.

5.1.1 Levenshtein

The Levenshtein Distance is in the collection of algorithms which have a complexity of $O(n^2)$. This is the case since the algorithm compares each character of a string to each character of another string. Specifically this means that the implemented Levenshtein Distance has a complexity of:

$$O(n \times m) \tag{5.1}$$

Where *n* and *m* are the length of the given texts, respectively.

Other operations, which assist in computing the Levenshtein Distance, are also performed. These do not affect the overall complexity of the *Levenshtein.java* class. Examples of these are the initializing of the cost array, done in $O(n)$ and assignment operations done in constant time, $O(1)$.

The space complexity of the normal Levenshtein algorithm is $O(n \times m)$, since it stores the cost array of all the characters in two texts. The implemented algorithm has a lower complexity, since cost is maintained by two separate arrays with the size of $n+1$ and $m+1$. This evaluates to a space complexity of:

$$O(n) \tag{5.2}$$

Where *n* is the length of the text.

5.1.2 Dice's Coefficient

The Dice Coefficient has to compute bigrams for each text. This is done by iterating through the text and finding each word, requiring *O(n).* Each of these words are separately processed and the bigrams are computed. This is done by iterating through each word resulting in a time of *O(m)*, which maximum can be half the length of the entire text, giving a complexity of $O\left(\frac{(n+1)}{2}\right)$. Each of these bigrams are put into a list, which totals a runtime of *O(n).* Thus giving a complexity of *O(m × (n + n)) → O(m × n)*. The computed bigrams for these two texts (*j* and *i,* respectively) are each run through and compared among the two sets. This gives a complexity of *O(i × j)*. The comparisons and assignment operations have a complexity of *O(1).* The resulting complexity for Dice's Coefficient is therefore:

$$O(i \times j) \tag{5.3}$$

Where *i* is the number of bigrams for the first text and *j* is the number of bigrams for the second text.

This is the resulting complexity because the number of bigrams will always be the length of the text minus one. The computation of bigrams is only done by iterating through each word, giving it a lower runtime than the bigram comparison.

The Dice's Coefficient algorithm stores all the bigrams for both texts, thus giving a space complexity of:

$$O(n) \tag{5.4}$$

Where, *n* is the number of bigrams of the longest text.

5.1.3 Semantic Similarity

This algorithm has without a doubt, the largest complexity, due to the large amount of comparisons it uses. To compute the similarity score, this method goes through several steps, which will be evaluated separately.

5.1.3.1 Tokenize Text

Since each character of a given text is run through in order to split the text into words, the time complexity of this step is:

$$O(n) \tag{5.5}$$

Where *n* is the length of the given text.

The space complexity of this method is:

$$O(m) \tag{5.6}$$

Where *m* is the number of words put into an *ArrayList*.

5.1.3.2 Stop Words

Each word is checked to be in a list and removed if present. Since each word is run through, the given complexity is:

$$O(n) \tag{5.7}$$

Where *n* is the number of words in the given *ArrayList*.

Since no stop words might be removed, the space complexity of this method is:

$$O(n) \tag{5.8}$$

Where *n* is the number of elements in the given *ArrayList*.

5.1.3.3 Stemming

For each word in an *ArrayList* of words, all POS are attempted to be looked up in WordNet until a match has been found. This gives a time complexity of:

$$O(n \times m) \tag{5.9}$$

Where *n* is the number of words in the given *ArrayList* and *m* is the number of available POS lookups.

The space complexity, which can be at most all the words given in the *ArrayList*, is:

$$O(n) \tag{5.10}$$

Where *n* is the number of words in the given *ArrayList*.

5.1.3.4 Stanford POS Tagger

Each word is looked up in the trained corpus provided by the *left3words* tagger. The time complexity of tagging is therefore:

$$O(n \times m) \tag{5.11}$$

Where *n* is the number of words in a given *ArrayList* and *m* is the number words in the trained corpus.[7]

The space complexity of the tagger is:

$$O(n) \tag{5.12}$$

Where *n* is the number of words in the given *ArrayList* with their attached POS.

5.1.3.5Simplified POS

Every word and its corresponding POS are looked up and, thus the time complexity of this method is:

$$O(n) \tag{5.13}$$

Where *n* is the number of elements in the two dimensional array given to the method.

The space complexity of this method is *2n*, since the words are stored in an array and their POS in an equivalent array. Giving an effective space complexity of:

[7] The inner workings of this tagger are not known and this complexity is therefore just a look at how the tagger works at the surface.

$$O(n) \tag{5.14}$$

Where *n* is the size of the given two-dimensional array.

5.1.3.6 Disambiguation

This part is the most complex, since this step makes a lot of comparisons. Each word is looked up in WordNet and if a noun or a verb is found, all of its possible senses are retrieved and stored. This takes *O(n × m)* time, since all words have to be looked up and all their corresponding senses. For each of these looked up words, the *k*-nearest words are looked up, and all their senses are stored, resulting in a complexity of *O(k × p)*. All the dictionary descriptions for each sense of a given word and the *k*-nearest word are looked up and stemmed. Each of the words in the stemmed description, are run through and it is checked if they are present in the other stemmed description. This gives a complexity of *O(m × p × q)*. The *k*-nearest best candidates are then compared, retrieving the candidate with the highest weight assigned to it. This has a time complexity of *O(k)*, since *k* best candidates are found. This gives a final complexity of:

$$O(n \times m^2 \times k \times p^2 \times q) \tag{5.15}$$

Where *n* is the number of words looked up, *m* is the number of senses for a given word, *k* is the number of compared words, *p* is the number of senses for the k^{th} word and *q* is the number of words in the stemmed definition of the m^{th} sense.

For each word comparison, the senses of the two words are stored, resulting in *O(n)*. The dictionary definition for each of these senses are looked up and processed, resulting in *O(m)*. The space complexity of the algorithm is therefore:

$$O(n + m) \tag{5.16}$$

Where *n* is the number of senses for a given word and *m* is the number of elements in the description of a sense.

5.1.3.7 Relatedness

The similarity score is computed by comparing all synsets in a given list with all synsets in another list. For each comparison all the relationships found between two given synsets are run through in order to find the relationship with minimum distance. The time complexity of this is therefore:

$$O(n \times m \times p) \tag{5.17}$$

Where *n* is the number of synsets in the first list, *m* is the number of synsets in the second list and *p* is the number of relationships found between an element in *n* and an element in *m*.

Each of these similarity scores is stored in a matrix. This gives a space complexity of:

$$O(n \times m) \tag{5.18}$$

Where *n* is the number of elements in the first list and *m* is the number of elements in the second list.

5.2 Similarity

The three algorithms carry out similarity comparisons in different manners. Because of this fact it is important to investigate what the resulting similarity scores mean. Even though all algorithms output a similarity between zero and one hundred, it may not mean that an output of thirty percent for two particular algorithms is equivalent. For one algorithm it might mean the similarity is high and for another it might indicate low similarity. The expected outputs and what they mean, for each of the three algorithms, are presented below.

5.2.1 Levenshtein

The Levenshtein Distance computes how far two strings are from each other. This means that small strings will be close to each other and larger strings will be farther from each other. The nature of larger strings allows them to have more noise, even though they are discussing the same topic. This is true because the larger strings have more room for expressive power. This allows for expressing the topics in a large string in more forms, in the way the string is structured. Since Levenshtein compares the direct distance, it is very sensitive to word ordering. It may indicate very little similarity if the topics of two strings are the same, but different ordering of these topics is used. It is also required that the same words are used to express something, for the direct comparison. This means that synonyms which mean the same will not count toward similarity.

The Levenshtein algorithm gives a high similarity for strings that are alike, in the sense of word usage and structural build-up i.e. direct copies. It is restricted in the way that the compared strings have to be fairly equal in length, otherwise counting this difference in length towards dissimilarity. Comparing a small string to a large string will therefore give a small similarity score.

5.2.2 Dice's Coefficient

This algorithm computes how much two strings have in common. Since the strings are split up and compared by their intersection, this algorithm is not dependant of word ordering i.e. the structure of the string. No interpretation of the content of the string is made, thus making it vulnerable to synonym usage and the like. Derivations of a word are handled, since they mostly are the same in bigram structure. In the case of a word having irregular derivations, such as mouse and mice, the similarity is not spotted. This is also the case for Levenshtein.

The Dice's Coefficient produces a high similarity score for strings that use the same wording. The placement of words in relation to each other is not important, since each bigram pair is compared across the whole space of the strings. It might give a higher similarity score than expected, since popular combinations of bigrams in a certain language may result false positives. The similarity score for this algorithm is dependent on the size of the compared strings. If one string is larger, the extra bigrams it contains may not be matched, resulting in a lower similarity score. In general a higher similarity score, than that of Levenshtein, is expected.

5.2.3 Semantic Similarity

This algorithm computes a similarity score between two strings, based on how related the contents of these are. All words are put into lists and compared to each other. The ordering of the words is therefore not important. Stop words are taken out of the similarity comparison and do not influence the final result. Furthermore, all the words which are not found in WordNet are likewise taken out of the equation. Words with a capitalized first letter are also taken out if they can be used as a noun. This means that the set of words, which the comparison is being made on, is reduced. In some cases this may increase the similarity score. In others it may reduce the similarity score. The algorithm is dependent on a correct tagging of POS, since only the POS of a word is looked up and compared. With a POS tagging success rate of roughly ninety percent, tagging results are reliable. Since disambiguation is done by comparing fairly short descriptions of senses, the correct designation of the meaning of a word is not expected to be high. In most cases the most common meaning of the word will be given, which is tolerable, since it is most likely the meaning of the word. In situations where a match has been found, a more appealing meaning is given, suiting the interest of comparing words. The best matching's from each set are grouped together and a similarity score is computed by these relations. This means that the text have to be fairly similar in length in order to get a correct similarity score. There can only be a certain amount of matching's between two sets i.e. the number of elements in the smallest set. In reality the length of the texts is not important, but the number of words which are found in WordNet is.

The Semantic Similarity algorithm should produce higher similarity scores than those given by Levenshtein and Dice's Coefficient. That is because, since this algorithm takes the relation between words into account. Words can have different forms, but still express the same thing, thus giving higher likelihood of striking a match. Exceptions are gibberish words, slang, names etc. i.e. words which cannot be found in a dictionary.

CHAPTER 6

Test

This section covers the tests which have been made in order to validate that the program runs as intended and to see how well the program performs in the terms described in chapter 5.

6.1 Validation

Often programs contain errors which may be obvious and others which may be hidden. Generally errors can be put into two categories; structural and functional, white-box and black-box respectively. Structural tests are concerned with testing of the inner workings of the program, i.e. the parts that are hidden. Structural errors may not always be evident, since they do not cause runtime errors in the program. They cause a different behaviour of the program, compared to that, which was intended. Functional errors are more evident in that they directly influence the functionality of the program. If a certain function in the program is not working it is obvious, since the desired functionality is not present and can therefore not be used.

Since there is a graphical user interface for the program, most tests have been conducted via functional tests. However it is important to see if the program behaves as intended, thus structural tests have also been executed.

6.1.1 Structural Tests

The structural tests have been conducted by looking at the state of different steps in the program. This has been done by print statements, showing the interior of selected components. Some very important bugs have been found by these tests. The Most critical ones have been found in the *Disambiguity.java* class. This class exhibits some of the main functionality in the program and thereby augments the impact of these errors.

Examples of these errors are wrong weighting of senses and mistakes in passing the retrieved senses. The error with wrong weighting of senses resulted in incorrect extraction of the most appropriate sense for a given word. As explained earlier, the method should return the most likely sense of a word for each *k*-nearest word. The

sense with the highest weight (most definition overlap) should have been picked. Instead the sense which occurred most times was picked. The sense occurring most times will usually be the most used sense of a given word, since in most cases no relation in the definition is found. The second error was a simple assignment error, which had a huge impact on which senses would be compared. All the elements containing the senses for a word would be assigned the same sense. This means in essence only one sense of a word would be compared to one sense of the compared word. The result of disambiguation is therefore invalidated, since no real comparison is made, i.e. the comparison only has one possible outcome. Both mistakes have been corrected. The structural tests also highlighted an important assignment issue. If only one word is looked up WordNet, no sense is given to that word. Because a sense needs to be compared to another sense in order to obtain a sense. This does not influence the overall performance of the program, since more than one word is found in most comparisons. Comparing one word is not that interesting either. For this purpose a dictionary or thesaurus can be used, thus this bug is left as is. Another example of an error which has not been corrected, since it was discovered rather late is: When removing stop words; words which are part of stop words, but not stop words in their own regard, are removed. An example of this behaviour is the removal of the word “al”. “al” is part of the stop word “all” and is consequently removed. This only impacts the resulting similarity score in a discrete way, since not many natural words exist in the form of subsequence of the stop words used by the program. This faulty behaviour is caused by the use of the *contains()* method specified by the *String* class.

6.1.2 Functional Test

The functional tests have been done by inputting some data to the program and relating the resulting output. In this manner tests can show if a given function displays the anticipated functionality. A classical way of conducting such tests is in the form of use cases. These specify the tested functionality, which is required by the program. As part of the design process they specify the behaviour of the program i.e. how it should act in certain situations. The program is fairly simple and does not provide much functionality. A simplified version of the use cases can be seen in the table below [29]:

Use Case 1:	**Load text files into the program**
Goal	Store a representation of a text file in the program
Requirements	A text file
Restrictions	Program must not be computing
Success	The text file is stored in the program and relevant labels are updated (text and word count label)
Failure	The text is not stored in the program and no labels are updated
Summary	The user clicks the text button and is presented with a FileChooser, from where the user can navigate to a text file and load it into the program.
Priority	Top
Utilized	Very often

Table 6.1*: showing an example of a use case*

The rest of the use cases can be seen in Appendix D.

The test of the use cases show that the program mostly statisfies the required functionality. It is worthy to mention that the program hangs, while it is computing. This means that the program cannot be closed at any point. The program satisfies the loading of text, but does not check if the selected file is in fact a text file. Invalid files have successfully been loaded into the program. Therefore it is important that the user ensures that the given file is in fact a text file. This can easily be done by utilizing the feature in the program only displaying text files, when a file is to be loaded. An important bug found by functionality test was a difference in the final similarity score for some texts, if the loading of text files was reversed. This is due to the heuristic function used to compute the final similarity score. By reversing the input, the final similarity score is affected because the compared sets are swapped. A new matching is made and a different result is given. This difference should not be too large, but can be large enough to have an influence. Larger texts suffer more, since difference for each incorrect score can scale up.

6.2 Experimentation

In this part the different algorithms are tested to see how well they perform in terms of runtime and "correctness" of the computed similarity score.

The runtime is calculated using the *System.currentTimeMillis()* method provided by the java platform. This function takes the time of the operation system and stores it. The runtime for a given algorithm can then be calculated by subtracting the time when the algorithm start from the time when the algorithm has completed. A lot of processes are carried out on a general purpose computer and may influence computational time of the algorithm. This is the case since other operations may demand the available resources. To minimize variance this algorithm is run multiple times and an average computational time is calculated. This test is conducted on the string: "running man hunted vigorously yesterday.". The algorithms are also tested to see how they perform on larger strings. To eliminate variance in the lookup operations for longer texts the string is copied and run through several times, giving the same operations for each test.[8]

Each test is done one hundred times for Levenshtein and Dice's Coefficient, since they have relatively short runtimes. This is done because a sudden drop in available resources may have a large impact on the overall result. The Semantic Similarity algorithm is run ten times for each test, since the variance is less likely to influence the overall result. The cache of the processor may also influence the runtime, since it optimizes lookup operations based on the operations being executed. The more likely an operation is to be executed, the more likely the data is to be stored in the fast cache memory for quick retrieval. More computational time is thus spent on an operation the first time it is executed, since the data has to be retrieved from slower memory. For this reason the

[8] The number of words in the tested strings varies from 5 to 11520 words.

first run of the algorithm has not been included in the test. The program is closed and re-opened between each test run for better duplication of the testing environment. All tests are run on the same computer, with the following central processing unit (CPU): *Intel Core i7 920 @ 3.50 GHz.*

It is hard to say what correct similarity scores are, but the algorithms can be compared to each other in order to distinguish what they output. One way to measure correctness of the given results is to correlate with human judgment, since natural languages are the product of human interaction in the form of communication. Human judgment must therefore be the baseline for assigning meaning and comparing text. Some texts have been chosen to measure the similarity. These texts are primarily news articles or part of such. They provide a good foundation to measure similarity, since different articles often highlight the same events. Further news agencies like Reuters[9] deliver news to the news companies, which construct their own articles based on a given story reported by Reuters. These articles are bound to incorporate some similarity, since one is constructed from the other. The texts used for the similarity comparison can be found in Appendix F[10]. The tested texts have varying length, to see how the algorithms cope with this factor. Most of the test texts are kept short, in order to get people to take the test and get the human perspective. Test persons are asked to evaluate two texts each test to decide how similar the texts are (and how much they relate to each other). The test persons are asked to evaluate the texts by giving them a number from zero to ten for each test. The number zero corresponds to zero percent similarity and ten corresponds to one hundred percent similarity. The result of these tests can be seen in Appendix E. Complaints about Test 4 has been received from several test participants. Two of the test participants did not complete this test, as they felt it was too long.

Further tests have also been made with texts that are not interpreted by humans. The tests taken by humans are test one to test four. All other stated tests are only done using the three algorithms.

[9] www.Reuters.com

[10] These comes from news sites such as CNN.com and BBC.com

CHAPTER 7

Results

This section covers the results from the mentioned tests in chapter 6.

7.1 Similarity Tests

The results from the various tests are listed below; the given percent is the similarity percentage.

Test 1

Levenshtein	28,1 %	execution time: 0 *ms*
Dice's Coefficient	26,7 %	execution time: 0 *ms*
Semantic Similarity	8,3 %	execution time: 4558 *ms*
Human Judgement	40,5 %	

Test 2

Levenshtein	27,9 %	execution time: 0 ms
Dice's Coefficient	48,4 %	execution time: 0 *ms*
Semantic Similarity	15,8 %	execution time: 8083 *ms*
Human Judgement	54,0 %	

Test 3

Levenshtein	25,3 %	execution time: 15 *ms*
Dice's Coefficient	55,2 %	execution time: 15 *ms*
Semantic Similarity	13,3 %	execution time: 23988 *ms*
Human Judgement	27,0 %	

Test 4

Levenshtein	26,5 %	execution time: 297 *ms*
Dice's Coefficient	81,0 %	execution time: 78 *ms*
Semantic Similarity	33,6 %	execution time: 227389 *ms*
Human Judgement	61,9 %	

Test 5

Levenshtein	29,5 %	execution time: 0 *ms*
Dice's Coefficient	46,2 %	execution time: 0 *ms*
Semantic Similarity	47,2 %	execution time: 4199 *ms*

Test 6

Levenshtein	100 %	execution time: 0 *ms*
Dice's Coefficient	100 %	execution time: 0 *ms*
Semantic Similarity	100 %	execution time: 4185 *ms*

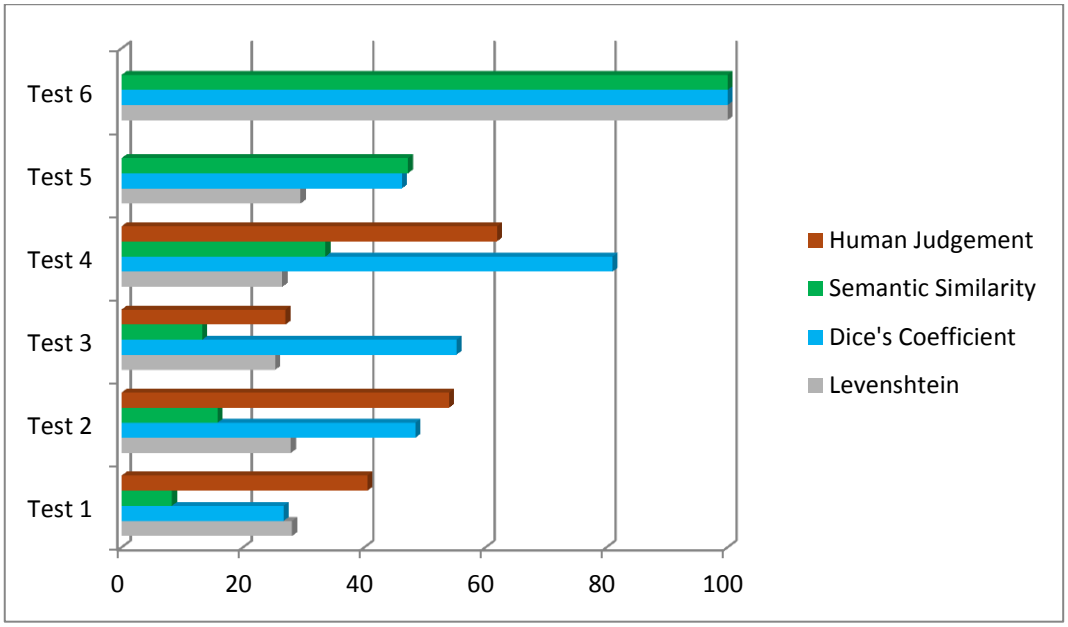


***Figure 7.1**: Showing a graphical overview of the results from the similarity tests, where the x-axis displays the percentage similarity*

7.2 Runtime Test

The runtime for the different algorithms are shown, by plotting the collected data into graphic content, as to give a better overview.

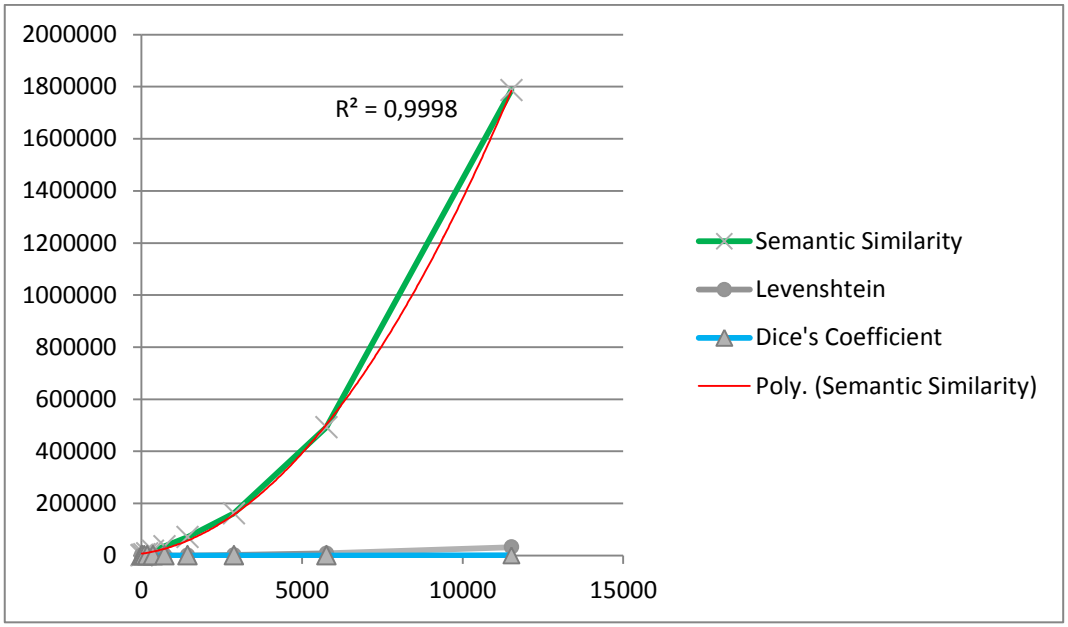


***Figure 7.2**: Showing the runtime in correlation to number of words*

Figure 7.2 shows the runtime for each algorithm, where the x-axis is the number of compared words and the y-axis is the elapsed time in milliseconds.

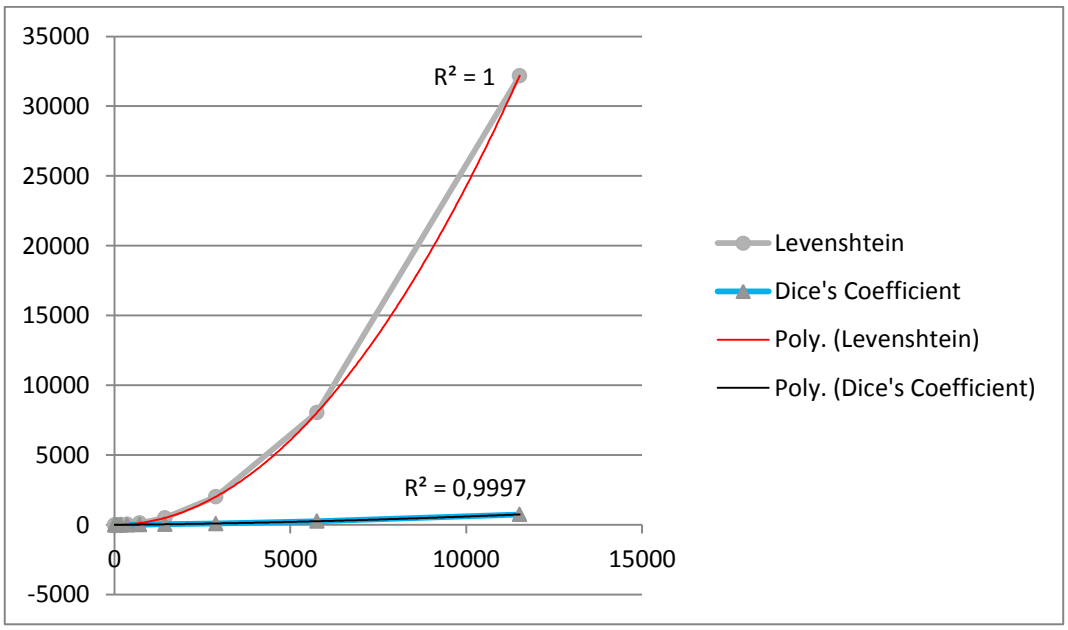


***Figure 7.3**: Showing the runtime for number of compared words*

Figure 7.3 is shown because the data of Levenshtein and Dice's Coefficient is not clearly displayed in Figure 7.2. The x-axis is the number of compared words and the y-axis is the elapsed time in milliseconds.

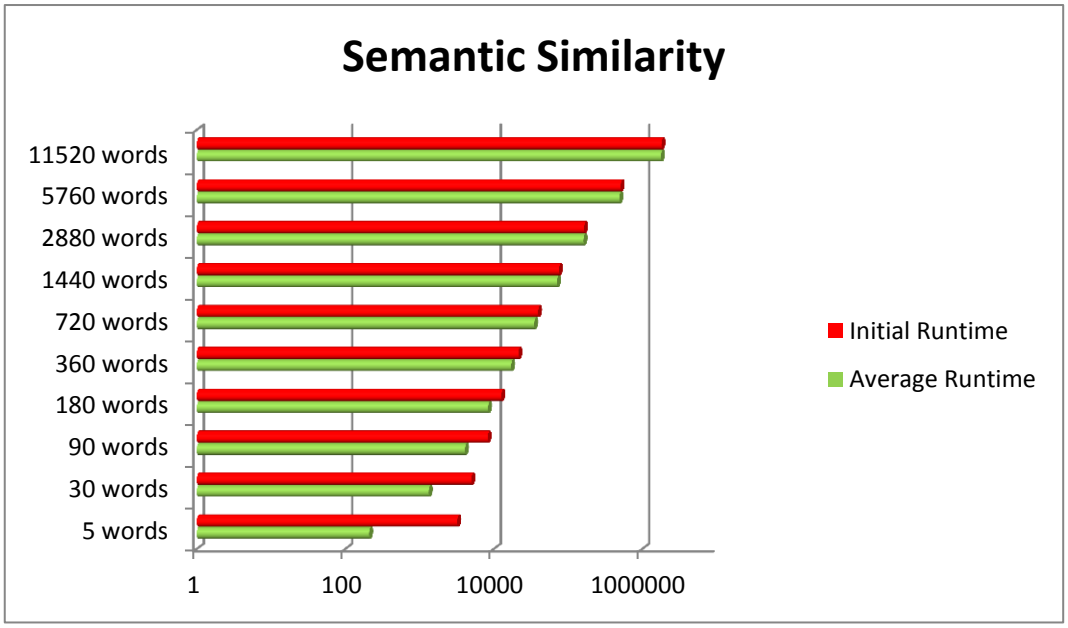


***Figure 7.4**: Showing the average runtime and initial runtime of Semantic Similarity on a logarithmic scale, where the x-axis is time in milliseconds and y-axis the number compared words*

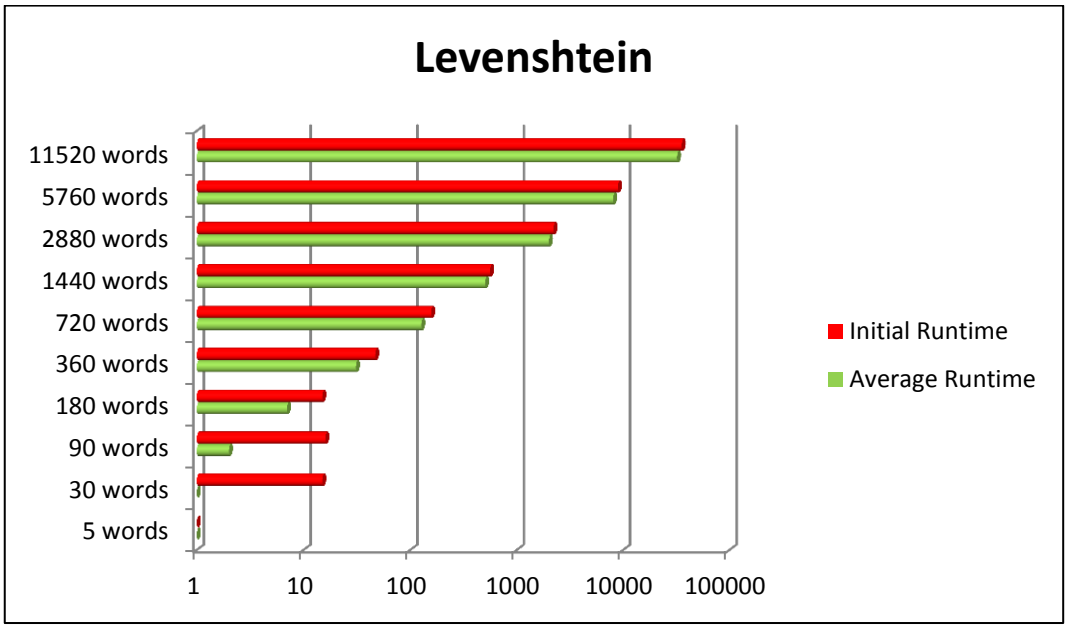


***Figure 7.5**: Showing the average and initial runtime for Levenshtein on a logarithmic scale, where the x-axis is time in milliseconds and y-axis the number compared words.*

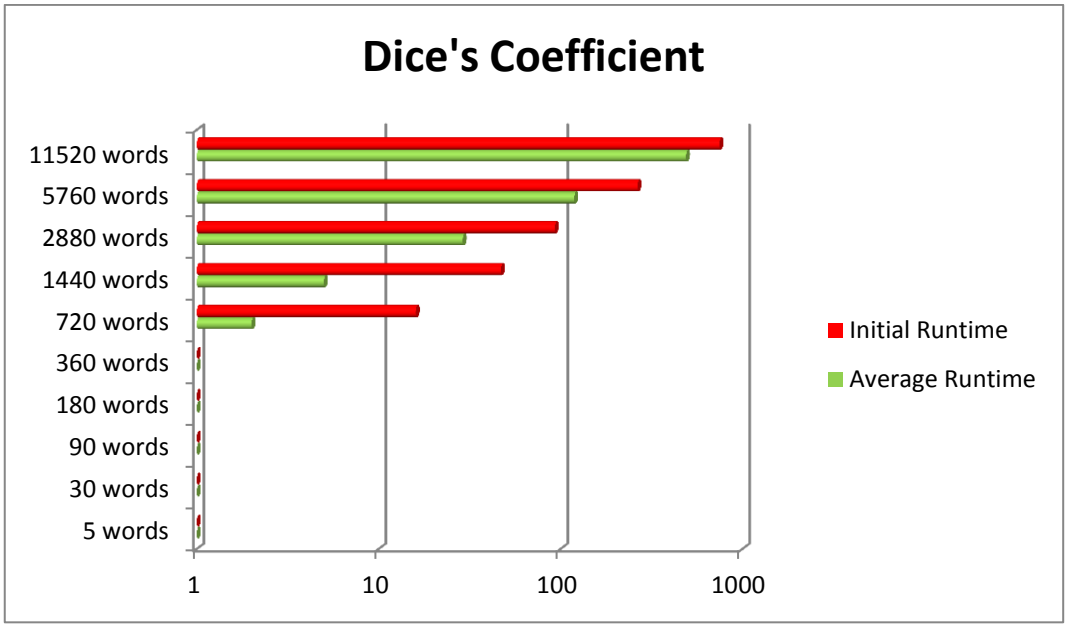


***Figure 7.6**: Showing the average and initial runtime for Dice's Coefficient on a logarithmic scale, where the x-axis is time in milliseconds and y-axis the number compared words*

CHAPTER 8

Discussion

This section covers a discussion of the results obtained in this thesis. The discussion is based on the results given in chapter 7 and a total evaluation of the final product. This merges into a discussion of possible improvements to the program.

8.1 Comparisons

It is very hard to measure similarity, since the definition of similarity is ambiguous. Humans relate differently to what similarity is. This is clearly seen in the variance of answers for the similarity tests. Some humans judge thematically, which means if two texts are from the same category they automatically assign some similarity even though the texts discuss completely different things. This has been seen in the responses to the human similarity tests, where some people discriminate more than others. A trend can be seen in how the test persons rate the tested texts. It seems that the test persons in general give either a high or low similarity score across all the texts. The divergence between the tests is not as high as the divergences between test persons. In hindsight it would have been a good idea to also ask how similar two texts are in a verbal expression. Examples of this can be little similarity, similar, very similar, essentially the same etc. In this way the test person's threshold for similarity can be found. The different thresholds can be correlated and thus a more precise measure can be used for the comparison.

Relating the results of the tests (Figure 7.1), it can be seen that Dice's Coefficient generally gives a higher similarity score than the other algorithms. This behaviour is expected, as the algorithm produces false positives for stop words and high frequency bigrams. Generally the similarity score follows the trend of the other algorithms, meaning coherence, is with approximation, easily spotted. This means the threshold for this algorithm should be set high, when comparing the similarity score for this algorithm in relation to the others. Levenshtein is the algorithm with least variance in the produced similarity score over the span of the tests. This is probably due to the high dependency of the text structure and the wording. As mentioned earlier Levenshtein is

best used where similarity is quite high. This correlates well with the test results, since only Test 6 has a high similarity. There are only twenty-six letters in the alphabet and a small number of these are used very frequently. This allows construction of a word by transformation in fewer steps, since most letters are already present. Therefore a lower bound for similarity should be considered in relation to the threshold. Semantic Similarity performs quite poor for the small test texts. The expected similarity is higher, since the small texts discuss the same topics with semantically similar words. The explanation for this is the high dependency for texts of equal length. Each missing word, in small texts, influences the overall similarity in a significant way. Analysing Test 5, the given similarity score should be higher. Only four words of the second text are looked up in WordNet, in contrast to six words from the first text. The word “Bergen” in the second text is picked out, because of the simplified proper noun recognition, provided by the algorithm. The consequence of this is an effective comparison of three words. The similarity score can therefore be fifty percent at maximum. The final output of forty-seven percent is therefore a quite good measure of similarity. The syntactical algorithms will not find the link between “left” and “kill”, as is done in this case. It is obvious that in cases where words are not looked up, they could be compared to the unmatched words in another way e.g. Levenshtein.

The human similarity results vary quite a lot and cannot be used as a good baseline for similarity. To use these results as a comparison tool, a more clinical test has to be constructed. This can be conducted as mentioned above, where similarity scores are correlated according to the perceived understanding of similarity. Another way is to instruct the participants by giving specific examples and providing judging principles. The average of human similarity results has been plotted into Figure 7.1. Even though results are not consistent, they can still be interpreted in some way. In all the tests human similarity scores are clearly above that of the Semantic Similarity scores. This can be seen as the capacity of human computation. Humans are able to associate words much better and faster than the constructed program. This results in a higher similarity score for the given tests. It can be mentioned that the results were expected to have a higher similarity score in the human test, especially for the smaller texts. This, again, might display that humans have very different ways of judging similarity. Another explanation could be that the constructor of the tests knows the context of these smaller texts and thus relates them in a certain fashion. This shows how prior knowledge might affect comparison results. Participants might also carry this trait, since they may have prior knowledge of certain texts or subjects and relate them in a different matter.

The runtime varies from the first execution of an algorithm to subsequent executions. There are two main reasons for this, as described earlier. When loading the program for the first time, data is collected from slow memory. After subsequent runs of the algorithm, this data is mainly stored in the cache making it faster to retrieve. The CPU interprets on the processes it executes and stores relevant data in fast memory, since it might need this data in the near future. This is a design feature of modern CPU’s aimed at making overall performance better. [30] In this manner subsequent runs benefit from

earlier runs. In Figure 7.4 the average and initial runtime for Semantic Similarity are compared. It is shown that initial runtime is always higher than the average runtime. A further reason for this is the dictionary files and the POS tagging libraries, which the program uses. These are loaded into the program, the first time Semantic Similarity is executed. The smaller the texts are, the larger the gap between initial and average runtime. This can be attributed to the loading of the program libraries. The tendency mentioned above can also be seen for Levenshtein (Figure 7.5) and Dice's Coefficient (Figure 7.6). This is though only attributed to the use of cache, since these do not use the libraries.

The tests indicate that the expected runtimes for Levenshtein, Dice's Coefficient and Semantic Similarity ((5.1), (5.3) and (5.15), respectively) discussed in chapter 5 are very coherent with actual results. This can be seen by how well all algorithms follow their trend line in Figure 7.2 and Figure 7.3. All algorithms execute in polynomial time. The Semantic Similarity algorithm has the largest runtime in accordance to its more complex nature, by the larger number of comparisons it computes.

8.2 Extensions

The list of possible improvements to the program is long, especially with regard to the semantic part. A lot of work has been put into this field and the whole area of NLP is too big to fit into the scope of this thesis. Many possible solutions have been simplified, but still display the essential features for the comparison process. Starting from one end, possible improvements for future work are listed below.

The ability to choose whether the given text should be removed of stop words and stemmed or not, for the syntactic algorithms, is a possible improvement. This may help the algorithm archive a better similarity score, since stop words are not counted toward similarity and words carrying similar meaning are presented in a mutual form. This method already exists, though it is currently a private method and therefore not reachable. Another improvement can be tokenizing the texts given to the Levenshtein algorithm and performing the comparison across words. This will make the algorithm independent of word ordering, however altering how the final result should be interpreted.

The most interesting improvements lie in the semantic part of the program. A better similarity score can be obtained by using more comparison operations. Thus bringing the different relationships specified by WordNet into the picture. The algorithm already has a complex runtime and these improvements will only heighten the complexity. Looking each immediate hypernym or hyponym up for a given sense and comparing the senses of these to give a better disambiguation process. All the relations specified in *Table 2.1*, can be used to further improve the similarity score in this manner. However this is very costly and should be used with caution. It has been shown how the comparison works and adding a new relationship to be checked for every sense is a fairly simple job. In

regard to the definition comparison the method for scoring each sense can be improved. Instead of counting occurrences of words the longest common subsequence with consecutiveness can be used. This satisfies Zipf's law and gives a better sense determination, since more weight is put on the correlation of sense definition. Another approach with higher word disambiguation success rate is the statistical approach described in the analysis (chapter 2). This, however, does not work with WordNet, unless a corpus of trained data, tagged with WordNet synsets, is used. The simplification of proper noun detection can be greatly improved, since most proper nouns are not in WordNet. To handle this, a naming entity recognition system should be used. The candidates for proper nouns in the program are currently not compared to other words. Since WordNet does not specify relationships for these, these should be compared directly. This could be done by Levenshtein. When in the domain of direct comparisons, the words which are not able to be looked up in WordNet could also be compared in this manner. This puts these words back into the equation of the similarity comparison, giving a better similarity score since more words are compared.

CHAPTER 9

Conclusion

The methods for computing textual similarity were chosen such that two approaches focused on syntax similarity and one mainly focused on semantics. Specifically these were Levenshtein, Dice's Coefficient (n-gram) and an ontology based approach. These respectively represent Edit Distance algorithms, Vector Space Model algorithms and semantic based algorithms.

From the analysis it is evident that NLP is a large field, which proposes many solutions for extracting information from natural languages. Only a small portion of this field has been portrayed in this thesis. To get an insight into the different areas of textual similarity requires quite a lot of research. As described in the thesis, the different approaches have both advantages and disadvantages. A solution must be picked in order to solve a specific task, which has been hard since this thesis looks generally at textual similarity.

The attained result is of satisfactory degree, since it has been shown how measuring of textual similarity can be accomplished. Tough simplified, the semantic algorithm approach, highlights the important steps of textual processing. Many improvements for the acquired result have been suggested and offer insight into how complex, semantic similarity measuring can be. When relating this project to the computation the brain does every day, it is clear how complex a mechanism the brain really is. The chosen approach only does a fraction of comparisons, compared to that of the brain, in order to obtain textual similarity.

The test results indicate how hard similarity is to quantify. Especially, the test of human similarity measure has shown how hard it is to construct a similarity measure, based on human perception of similarity. The test results show some form of coherence between the algorithms and human results. The outputted result should not be taken as a direct measure, but should be correlated for each algorithm. The number of conducted tests is not vast enough to get a good correlation factor. More tests have not been made due to delimitation, which has to be made at some point.

The results show that Levenshtein is good for the purpose of textual similarity, where texts have a high degree of overlap. For texts with little overlap in wording sequence it tends to measure similarity in the range of twenty to thirty percent, which must be set as a lower bound. This makes it good for detecting edits in a document. Dice's Coefficient in association with bigrams gives quite a high similarity measure. This is because of frequently occurring bigram combinations. When using this algorithm, one should be sceptical. The ontology based algorithm works well, but is limited by simplifications. When using this algorithm, one should correlate the results, such that they correlate with human perception of similarity or some other perception of similarity. This has to be done since the given scores otherwise do not make much sense.

No best approach is chosen since it has become clear that the algorithms all have their domain of use. Each performs well in their own regard. When using humans as a standard the ontology approach displays quite good coherence if the results are correlated with that of the tested results.

An open problem in the program is the heuristic function, which calculates a matching for scoring the ontology based algorithm. This matching is not maximal and therefore the real calculated similarity score, which is calculated earlier in the process, is not displayed.

Bibliography

[1] R. DANIEL. 2010. *Cognition: Exploring the science of the mind*. W.W. Norton & Company. 324-326.

[2] A. M. GEORGE, B. RICHARD, F. CHRISTIANE, G. DEREK AND M. KATHERINE. Introduction to wordnet: An on-line lexical database. 1993. Five papers on WordNet.

[3] PRINCETON UNIVERSITY, 2012. What is WordNet? http://wordnet.princeton.edu/ [Visited 21 June 2012]

[4] Wikipedia contributors, 2012. Wikipedia: WordNet, In Wikipedia, The Free Encyclopaedia, http://en.wikipedia.org/wiki/WordNet [Visited 21 June 2012].

[5] GEEKNET.INC, 2009. Sourceforge: Main Page, http://sourceforge.net/apps/mediawiki/jwordnet/index.php?title=Main_Page [Visited 21 June 2012].

[6] J. NICO, 2004. Relational Sequence Learning and User Modelling. Ph.D. thesis U.D.C. 681.3*I26. Faculteit Wetenschappen, Katholieke Universiteit Leuven. 190

[7] S. DASGUPTA, C.H. PAPADIMITRIOU, and U.V. VAZIRANI, 2006. *Algorithms*. 173

[8] R.W. HAMMING, 1950, *Error detecting and error correcting codes*, *Bell System Technical Journal* 29 (2): 147–160, MR 003593

[9] M.A. JARO, 1989. *Advances in record linkage methodology as applied to the 1985 census of Tampa Florida*. *Journal of the American Statistical Society*. 414–20

[10] LEVENSHTEIN VI, 1966. *Binary codes capable of correcting deletions, insertions, and reversals*. *Soviet Physics Doklady*.

[11] Wikipedia contributors, 2012. Wikipedia: Dameru-Levenshtein distance, In Wikipedia, The Free Encyclopaedia, http://en.wikipedia.org/wiki/Damerau-Levenshtein_distance [Visited 21 June 2012].

[12] R.I. GREENBERG, 2003. *Bounds on the Number of Longest Common Subsequences*. Dept. of Mathematical and Computer Sciences, Loyola University

[13] D. DUBIN, 2004. *The Most Influential Paper Gerard Salton Never Wrote*.

[14] Wikipedia contributors, 2012. Wikipedia: Cosine similarity, In Wikipedia, The Free Encyclopaedia, http://en.wikipedia.org/wiki/Cosine_similarity [Visited 21 June 2012].

[15] S.J. KAREN. 1972. *A statistical interpretation of term specificity and its application in retrieval*. *Journal of Documentation.* 11–21.

[16] L.R. DICE 1945. *Measures of the Amount of Ecologic Association Between Species*. *Ecology*. 297–302

[17] W.B. CAVNAR, J.M. TRENKLE, 1994. *N-grambased text categorizatio*n. 3rd Annual Symposium on Document Analysis and Information Retrieval.

[18] Wikipedia contributors, 2012. Wikipedia: Stemming, In Wikipedia, The Free Encyclopaedia, http://en.wikipedia.org/wiki/Stemming [Visited 21 June 2012].

[19] E. BRILL. 1992. *A simple rule-based part of speech tagger*. Association for Computational Linguistics, Stroudsburg, PA, USA, 152-155.

[20] SPRINGER, 2012. http://www.springerimages.com/img/Images/BMC/MEDIUM_1471-2105-11-101-2.jpg [Visited 21 June 2012]

[21] S.WALLIS, 2008. *Searching treebanks and other structured corpora*. Lüdeling, A.

[22] J. R. FINKEL, T. GRENAGER, AND C. MANNING, *Incorporating Non-local Information into Information Extraction Systems by Gibbs Sampling*. Computer Science Department, Stanford University

[23] Wikipedia contributors, 2012. Wikipedia: Word-sense disambiguation, In Wikipedia, The Free Encyclopaedia, http://en.wikipedia.org/wiki/Word_sense_disambiguation [Visited 21 June 2012].

[24] M. LESK, 1986. *Automatic sense disambiguation using machine readable dictionaries: how to tell a pine cone from an ice cream cone.* New York, NY, USA. ACM. 24-26

[25] T. PEDERSEN AND J. MICHELIZZI, 2008. http://marimba.d.umn.edu/similarity/measures.html [Visited 21 June 2012]

[26] G.K ZIPF, 1949. *Human Behavior and the Principle of Least Effort*. Cambridge, Massachussetts: Addison-Wesley

[27] T. GRUBER, 2007. What is an Ontology? *http://www.ksl.stanford.edu/kst/what-is-an-ontology.html* [Visited 21 June 2012]

[28] E. E. DAHLGREN, 2012. The Rank Distributional Space of Negative Polarity Items. M.Sc. thesis. Department of Linguistics, The University of Chicago. 25

[29] M. FOWLER, 2004. *UML Distilled: A brief guide to the standard object modelling language, Addison-Wesley.* 99

[30] D. A. PATTERSON AND J. L. HENNESY, 2009. *Computer Organization and Design: The hardware/software interface.* Elsevier. 457

[31] S. BURBECK, 1997. How to Use Model-View-Controller http://st-www.cs.illinois.edu/users/smarch/st-docs/mvc.html [Visited 21 June 2012]

[32] B. CASTELLO, 2011. The Hungarian algorithm. http://www.ams.jhu.edu/~castello/362/Handouts/hungarian.pdf [Visited 21 June 2012]

Appendix A

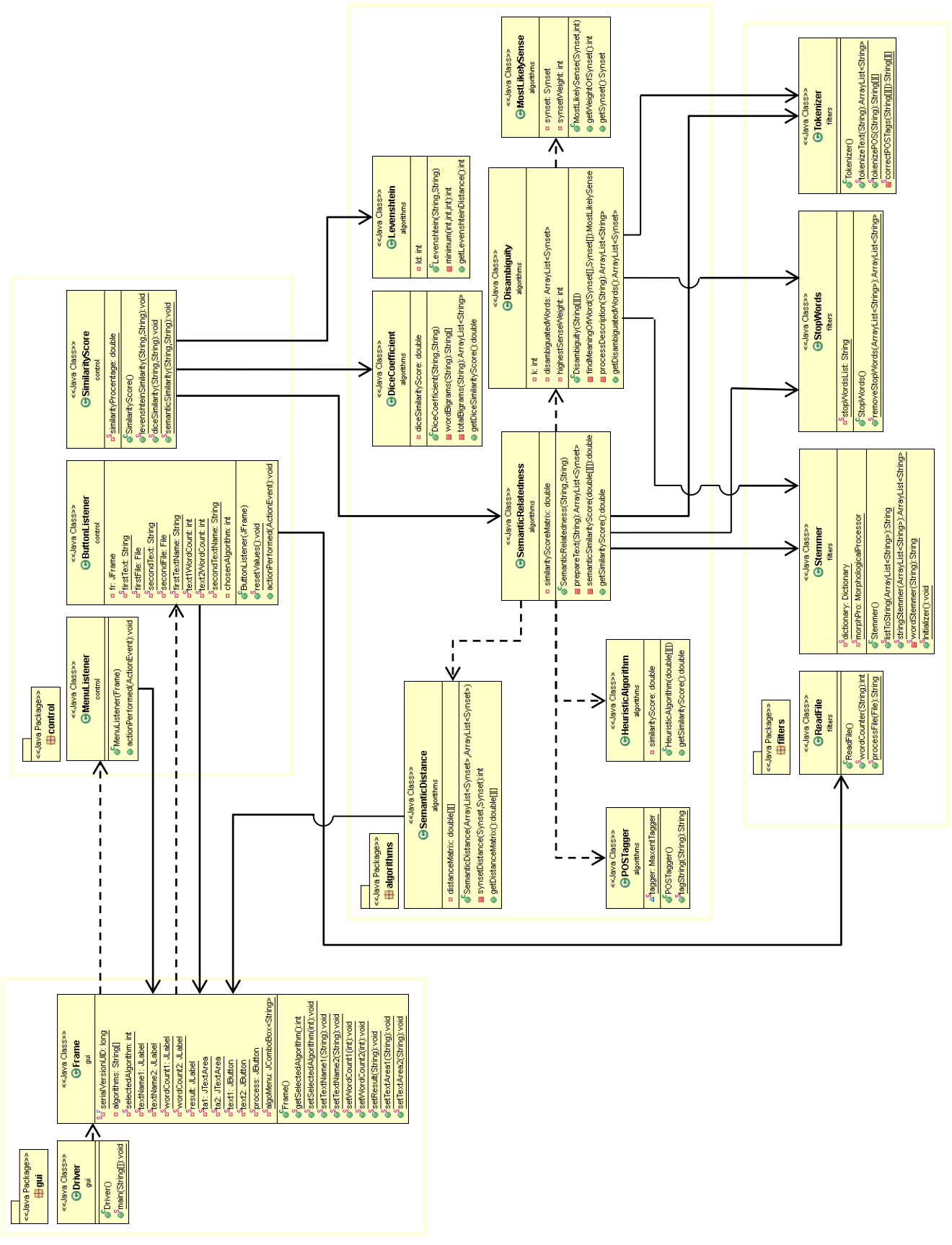
Penn Treebank Tag Set

CC	Coordinating conjunction
	e.g. and, but, or...
CD	Cardinal Number
DT	Determiner
EX	Existential *there*
FW	Foreign Word
IN	Preposition or subordinating conjunction
JJ	Adjective
JJR	Adjective, comparative
JJS	Adjective, superlative
LS	List Item Marker
MD	Modal
	e.g. can, could, might, may... NN Noun, singular or mass
NNP	Proper Noun, singular
NNPS	Proper Noun, plural
NNS	Noun, plural
PDT	Pre-determiner
	e.g. all, both ... when they precede an article POS Possessive Ending
	e.g. Nouns ending in 's
PRP	Personal Pronoun
	e.g. I, me, you, he...
PRP$	Possessive Pronoun
	e.g. my, your, mine, yours...
RB	Adverb
	Most words that end in -ly as well as degree words like quite, too and very

RBR	Adverb, comparative Adverbs with the comparative ending -er, with a strictly comparative meaning.
RBS	Adverb, superlative
RP	Particle
SYM	Symbol Should be used for mathematical, scientific or technical symbols
TO	*to*
UH	Interjection **e.g.** uh, well, yes, my... VB Verb, base form subsumes imperatives, infinitives and subjunctives
VBD	Verb, past tense includes the conditional form of the verb to be
VBG	Verb, gerund or present participle
VBN	Verb, past participle
VBP	Verb, non-3rd person singular present
VBZ	Verb, 3rd person singular present
WDT	Wh-determiner **e.g.** which, and *that* when it is used as a relative pronoun
WP	Wh-pronoun **e.g.** what, who, whom...
WP$	Possessive wh-pronoun **e.g.** WRB Wh-adverb **e.g.** how, where why

Appendix B

Class Diagram



Appendix C

Stop-words.txt

a	don't	in	she'd	wasn't
about	down	into	she'll	we
above	during	is	she's	we'd
after	each	isn't	should	we'll
again	few	it	shouldn't	we're
against	for	it's	so	we've
all	from	its	some	were
am	further	itself	such	weren't
an	had	let's	than	what
and	hadn't	me	that	what's
any	has	more	that's	when
are	hasn't	most	the	when's
aren't	have	mustn't	their	where
as	haven't	my	theirs	where's
at	having	myself	them	which
be	he	no	themselves	while
because	he'd	nor	then	who
been	he'll	not	there	who's
before	he's	of	there's	whom
being	her	off	these	why
below	here	on	they	why's
between	here's	once	they'd	with
both	hers	only	they'll	won't
but	herself	or	they're	would
by	him	other	they've	wouldn't
can't	himself	ought	this	you
cannot	his	our	those	you'd
could	how	ours	through	you'll
couldn't	how's	ourselves	to	you're
did	i	out	too	you've
didn't	i'd	over	under	your
do	i'll	own	until	yours
does	i'm	same	up	yourself
doesn't	i've	shan't	very	yourselves
doing	if	she	was	

Appendix D

Use Cases

Use Case 1: Load text files into the program

Goal	Store a representation of a text file in the program
Requirements	A text file
Restrictions	Program must not be computing
Success	The text file is stored in the program and relevant labels are updated (text and word count label)
Failure	The text is not stored in the program and no labels are updated
Summary	The user clicks the text button and is presented with a FileChooser, from where the user can navigate to a text file and load it into the program.
Priority	Top
Utilized	Very often

Use Case 2: Get similarity of two texts with Levenshtein

Goal	Get a similarity score of two given texts
Requirements	Two text files are loaded into the program
Restrictions	Program must not be computing
Success	The similarity score is computed and show in the result label
Failure	The similarity score is not computed and no labels are updated
Summary	The user chooses the Levenshtein algorithm in the dropdown menu and clicks the process button.
Priority	Top
Utilized	Very often

Use Case 3: Get similarity of two texts with Dice's Coefficient

Goal	Get a similarity score of two given texts
Requirements	Two text files are loaded into the program
Restrictions	Program must not be computing
Success	The similarity score is computed and show in the result label
Failure	The similarity score is not computed and no labels are updated
Summary	The user chooses the Dice's Coefficient algorithm in the dropdown menu and clicks the process button.
Priority	Top
Utilized	Very often

Use Case 4: Get similarity of two texts with Semantic Similarity

Goal	Get a similarity score of two given texts
Requirements	Two text files are loaded into the program
Restrictions	Program must not be computing
Success	The similarity score is computed and show in the result label
Failure	The similarity score is not computed and no labels are updated
Summary	The user chooses the Semantic Similarity algorithm in the dropdown menu and clicks the process button.
Priority	Top
Utilized	Very often

Use Case 5: Close the program

Goal	The program window is closed
Requirements	Either the default close button is pressed or the menu item "Close" is pressed
Restrictions	None
Sucess	The program is terminated
Failure	The program is not closed and the frame is still visible
Summary	The user presses the "File" menu and selects the "Close" item
Priority	Top
Utilized	Often

Use Case 6: Reset program stats

Goal	All the values in the program are reset
Requirements	The menu item "Reset" is pressed
Restrictions	Program must not be computing
Success	All values in the program are reset and all relevant labels are updated
Failure	Program values remain the same and labels are not updated
Summary	The user presses the "File" menu and selects the "Reset" item
Priority	Low
Utilized	Rarely

Appendix E

Human Similarity Test Results

Test Person 1
Test 1: 20 %
Test 2: 35 %
Test 3: 15 %
Test 4: N/A

Test Person 2
Test 1: 10 %
Test 2: 20 %
Test 3: 10 %
Test 4: 25 %

Test Person 3
Test 1: 50 %
Test 2: 70 %
Test 3: 40 %
Test 4: 80 %

Test Person 4
Test 1: 30 %
Test 2: 60 %
Test 3: 30 %
Test 4: N/A

Test Person 5
Test 1: 90 %
Test 2: 90 %
Test 3: 35 %
Test 4: 65 %

Test Person 6
Test 1: 30 %
Test 2: 50 %
Test 3: 20 %
Test 4: 75 %

Test Person 7
Test 1: 50 %
Test 2: 70 %
Test 3: 40 %
Test 4: 80 %

Test Person 8
Test 1: 20 %
Test 2: 40 %
Test 3: 10 %
Test 4: 50 %

Test Person 9
Test 1: 60 %
Test 2: 70 %
Test 3: 50 %
Test 4: 80 %

Test Person 10
Test 1: 45 %
Test 2: 60 %
Test 3: 20 %
Test 4: 40 %

Appendix F

Test Texts

Here the texts, which are used for similarity test, are presented.

Test 1

This test is of headlines from two different news articles discussing the same topic.

First Text

Egyptian military keeps power despite presidential vote

Second Text

Egypt rivals claim victory as army tightens grip

Test 2

This test is of article summaries, discussing the same topic.

First Text

World leaders meeting at a G20 summit in Mexico have urged Europe to take all necessary measures to overcome the eurozone debt crisis.

Second Text

European officials sought to deflect mounting pressure from world leaders, warning of a long road ahead to end the region's debt crisis.

Test 3

This test is of some cosmology in Greek and Nordic mythology. A lot of names are used, testing the algorithms with “unfamiliar” words. The texts do not discuss the same topic, other than the cosmology of the religions.

First Text

According to Norse myth, the beginning of life was fire and ice, with the existence of only two worlds: Muspelheim and Niflheim. When the warm air of Muspelheim hit the cold ice of Niflheim, the jötunn Ymir and the icy cow Audhumla were created. Ymir's foot bred a son and a man and a woman emerged from his armpits, making Ymir the progenitor of the Jötnar. Whilst Ymir slept, the intense heat from Muspelheim made him sweat, and he sweated out Surtr, a jötunn of fire. Later Ýmir woke and drank Auðhumla's milk. Whilst he drank, the cow Audhumbla licked on a salt stone. On the first day after this a man's hair appeared on the stone, on the second day a head and on the third day an entire man emerged from the stone. His name was Búri and with an unknown jötunn female he fathered Borr, the father of the three gods Odin, Vili and Ve.

Second Text

A motif of father against son conflict was repeated when Cronus was confronted by his son, Zeus. Because Cronus had betrayed his father, he feared that his offspring would do the same, and so each time Rhea gave birth, he snatched up the child and ate it. Rhea hated this and tricked him by hiding Zeus and wrapping a stone in a baby's blanket, which Cronus ate. When Zeus was full grown, he fed Cronus a drugged drink which caused him to vomit, throwing up Rhea's other children and the stone, which had been sitting in Cronus's stomach all along. Zeus then challenged Cronus to war for the kingship of the gods. At last, with the help of the Cyclopes (whom Zeus freed from Tartarus), Zeus and his siblings were victorious, while Cronus and the Titans were hurled down to imprisonment in Tartarus.

Test 4

This test is of full news articles discussing the same topic.

First Text

U.N. monitors came under fire on Thursday and were prevented from reaching a village where forces loyal to Syrian President Bashar al-Assad were reported to have massacred at least 78 civilians, U.N. officials said.

U.N. Secretary-General Ban Ki-moon described the reported events in Mazraat al-Qubeir as "unspeakable barbarity," while the White House condemned the "outrageous targeted killings of civilians" and again called on other countries to halt support for Assad.

Opposition activists said up to 40 women and children were among those killed in the Sunni Muslim village near Hama on Wednesday, posting film on the Internet of bloodied or charred bodies.

"There was smoke rising from the buildings and a horrible smell of human flesh burning," said a Mazraat al-Qubeir resident who told how he had watched Syrian troops and "shabbiha" gunmen attack his village as he hid in his family's olive grove.

"It was like a ghost town," he told Reuters by telephone, asking not to be identified because he feared for his safety.

"After the shabbiha and tanks left, the first thing I did was run to my house. It was burned. All seven people from my house were killed. I saw bodies on the stairs, the bathroom and bedroom. They were all burned," the witness said.

The latest killings, less than two weeks after 108 men, women and children were slain in the town of Houla, piled pressure on world powers to halt the carnage in Syria, but they have been paralyzed by rifts pitting Western and most Arab states against Assad's defenders in Russia, China and Iran.

Ban, addressing a special U.N. General Assembly session on the crisis, again urged Assad to implement international envoy Kofi Annan's six-point peace plan immediately.

He said U.N. monitors, in Syria to check compliance with a truce declared by Annan on April 12 but never implemented, had come under small-arms fire on their way to Mazraat al-Qubeir.

There was no mention of any of the monitors being injured.

The chief of the monitoring mission, General Robert Mood, said Syrian troops and civilians had barred the team, stopping them at checkpoints and turning them back.

"They are going back to their base in Hama and they will try again tomorrow morning," spokeswoman Sausan Ghosheh said.

A Syrian official denied reports from the village, telling the state news agency that residents had asked security forces for help after "terrorists" killed nine women and children.

Assad, who has yet to comment on Wednesday's violence, decried the Houla killings as "monstrous" and denied his forces were responsible.

U.S. Secretary of State Hillary Clinton described the latest reported massacre as unconscionable and again demanded that Assad step down and leave Syria. "We are disgusted by what we are seeing," she said during a visit to Istanbul.

British Foreign Secretary William Hague told reporters there was evidence of "escalating criminality" by pro-government forces in Syria.

"Syria is clearly on the edge ... of deeper violence, of deep sectarian violence; village against village, pro-government militias against opposition areas and of looking more like Bosnia in the 1990s than of Libya last year," he said.

Clinton said the United States was willing to work with all U.N. Security Council members, including Russia, on a conference on Syria's political future, but made clear that Assad must go and his government be replaced with a democratic one.

Annan, the U.N.-Arab League envoy for Syria, was due to brief the Security Council at closed-door talks in New York on Thursday.

A senior Russian diplomat said Moscow would accept a Yemen-style power transition in Syria if it were decided by the people, referring to a deal under which Yemeni leader Ali Abdullah Saleh stepped down in February after a year of unrest.

"The Yemen scenario was discussed by the Yemenis themselves. If this scenario is discussed by Syrians themselves and is adopted by them, we are not against it," Deputy Foreign Minister Mikhail Bogdanov said, according to the Interfax news agency.

Video purportedly from Mazraat al-Qubeir showed the bodies of at least a dozen women and children wrapped in blankets or white shrouds, as well as the remains of burned corpses.

"These are the children of the Mazraat al-Qubeir massacre ... Look, you Arabs and Muslims, is this a terrorist?" asks the cameraman, focusing on a dead infant's face. "This woman was a shepherd, and this was a schoolgirl."

A Hama-based activist using the name Abu Ghazi listed more than 50 names of victims, many from the al-Yateem family, but said some burned bodies could not be identified. The bodies of between 25 and 30 men were taken away by the killers, he said.

Shabbiha, drawn mostly from Assad's minority Alawite sect that is an offshoot of Shi'ite Islam, have been blamed for the killings of civilians from the Sunni Muslim majority. That has raised fears of an Iraq-style sectarian bloodbath and worsened tensions between Shi'ite Iran and mainly Sunni-led Arab states.

Events in Syria's 15-month-old uprising are difficult to verify due to tight state curbs on international media access.

U.N. diplomats said they expected Annan to present the Security Council with a new proposal to rescue his failing peace plan - a "contact group" of world powers and regional ones like Saudi Arabia, Turkey, Qatar and Iran, which is an ally of Syria.

Rebel groups in Syria say they are no longer bound by Annan's truce plan and want foreign weapons and other support.

Western leaders, wary of new military engagements in the Muslim world, have offered sympathy but shown no appetite for taking on Assad's military, supplied by Russia and Iran.

Annan sees his proposed forum as a way to break a deadlock among the five permanent members of the Security Council, where Russia and China have twice vetoed resolutions critical of Syria that were backed by the United States, Britain and France.

It would seek to map out a political transition under which Assad would leave office ahead of free elections, envoys said.

Russian Foreign Minister Sergei Lavrov on Wednesday proposed an international meeting on Syria that would include the prime candidates for Annan's proposed contact group, including Iran.

Clinton reacted coolly to that idea, accusing Iran of "stage-managing" Syria's repression of its opponents in which the United Nations says more than 10,000 people have been killed.

Britain's Hague told reporters the Annan plan had failed so far but was not dead yet.

"Russia has important leverage over the Syrian regime and if all the members of the Security Council and the whole Arab World increased the pressure on the Assad regime to implement that plan, then it is still possible to do so," Hague said.

German Foreign Minister Guido Westerwelle said it was important to involve Russia in peace efforts on Syria, saying the conflict there could ignite a regional conflagration.

"Assad's regime must know there is no protective hand over these atrocities," he said in Istanbul on Thursday.

Leaders of a bloc grouping China, Russia and Central Asian states called for dialogue to resolve the Syria conflict, rather than any firmer action by the Security Council.

France said it would host a conference of the "Friends of Syria" - countries hostile to Assad - on July 6.

Second Text

Gunfire from small arms targeted U.N. monitors in Syria trying to get to the scene of another massacre, U.N. Secretary-General Ban Ki-moon said on Thursday.

The incident came as Ban, international envoy Kofi Annan and others implored the U.N. General Assembly to stop the violence in Syria, which started 15 months ago when a tough Syrian crackdown against peaceful protesters developed into an uprising.

The massacre occurred on Wednesday in the village of Qubeir, west of Hama, amid reports that dozens of civilians, including women and children, were slain.

Ban said the U.N. monitors were initially denied access to the area and were working to get to the scene.

"I just learned a few minutes ago that while trying to do so, the U.N. monitors were shot at by small arms," he said.

Maj. Gen. Robert Mood, head of the U.N. Supervision Mission in Syria, said observers heading to the village to verify reports of the killings have been blocked by soldiers and civilians. Residents also observers that they are at risk if they enter the village.

"Despite these challenges, the observers are still working to get into the village to try to establish the facts on the ground. UNSMIS is concerned about the restriction imposed on its movement as it will impede our ability to monitor, observe and report," Mood said.

Annan, the envoy tasked with forging peace in Syria, deplored the Qubeir massacre and told assembly members Thursday that Syrians could face the prospects of full-blown civil war if peace isn't made.

Speaking before the U.N. General Assembly, Annan said the six-point plan he forged isn't working and the "crisis is escalating."

"If things do not change, the future is likely to be one of brutal repression, massacres, sectarian violence and even all-out civil war," Annan said. "All Syrians will lose.

"The violence is getting worse. The abuses are continuing. The country is becoming more polarized and more radicalized. And Syria's immediate neighbors are increasingly worried about the threat of spillover."

Annan, who is the joint special envoy of the United Nations and the Arab League, said three months after he accepted the "tough job" to forge peace and launch a "political process for transition," success has not occurred.

"Despite the acceptance of the six-point plan and the deployment of a courageous mission of United Nations observers to Syria, I must be frank and confirm that the plan is not being implemented," he said.

Annan said Arab League ministers he addressed Saturday "offered concrete ideas on how to increase pressure for compliance."

"Clearly, the time has come to determine what more can be done to secure implementation of the plan and/or what other options exist to address the crisis," he said.

He spoke amid worldwide horror over the Qubeir massacre and another in Houla two weeks ago. More than 100 people, including women and children, were killed in Houla. Opposition activists blamed it on government forces and allied militia, a claim Syrian President Bashar al-Assad denied.

"This took place just two weeks after the massacre in Houla that shocked the world. Those responsible for perpetrating these crimes must be held to account. We cannot allow mass killing to become part of everyday reality in Syria," Annan said.

He said he told al-Assad nine days ago that his peace plan was not being implemented.

"I strongly urged him to take bold and visible steps to now radically change his military posture and honor his commitments to the six-point plan. I urged him to make a strategic decision to change his path. I also made clear that his government must work with my mediation effort on behalf of both organizations that I represent. President Assad believed the main obstacle was the actions of militants. Clearly, all parties must cease violence. But equally clearly, the first responsibility lies with the government," he said.

Even though Syria has released some detainees and there has been agreement "on modalities for humanitarian assistance," more is required, Annan said.

After the conversation, he said, "shelling of cities has intensified" and "government-backed militia has free rein with appalling consequences." Also, he said, al-Assad's address at the National Assembly this week has not indicated charge.

Annan said armed opposition forces haven't seen a "reason to respect cessation of hostilities" and "have intensified their attacks." Referring to bombings in Damascus and Aleppo, he said the situation is "made more complex" by attacks that are "indicative of the presence of a third actor." Some analysts say jihadist groups are responsible for those acts.

"We must find the will and the common ground to act -- and act as one. Individual actions or interventions will not resolve the crisis. As we demand compliance with international law and the six-point plan, it must be made clear that there will be consequences if compliance is not forthcoming. We must also chart a clearer course for a peaceful transition if we are to help the government and opposition, as well as Syrian society, to help resolve the crisis," he said.

Annan will also address the Security Council in New York. The council comprises 15 countries, while the assembly is made up of 193 members of the United Nations.

Ban also sounded the alarm about deteriorating conditions in Syria. He said the country "is at the pivotal moment. So are we. Syria and the region can move from tipping point to breaking point. The dangers of full-scale civil war are imminent and real."

Syrian Ambassador to the U.N. Bashar Jaafari repeated the government's position that anti-government terrorists, and not the regime, are responsible for the massacres and the bloodshed during the crisis. The government blamed a terrorist group for the latest massacre, saying it was timed to coincide with the U.N. meetings to make the regime look bad.

He said Syria is open to true reform, the government is amenable to dialogue, and it has no problem with the opposition. However, he said, some forces are taking up arms and have no desire for reconciliation. He also said outside elements are instigating violence in the country.

Opposition activists accuse forces loyal to al-Assad of the killings at Qubeir, and they placed the number of dead at 78.

Regime forces shelled Qubeir before militias on foot used knives, guns and AK-47 rifles to kill residents, the opposition Local Coordination Committees of Syria said.

About 40 victims of the attack were buried in a mass grave Thursday, according to a youth activist whom CNN is not naming for safety reasons. Shabiha -- or pro-government gangs -- took other bodies to neighboring villages, the activist said. More than half of those killed were women and children, according to a local activist who claimed to have carried bodies.

CNN cannot independently confirm reports from within Syria because the government strictly limits access by foreign journalists.

International outrage has grown over the recent violence, reviving calls to isolate the regime and toughen sanctions.

U.S. Secretary of State Hillary Clinton, meeting in Istanbul with Turkish Foreign Minister Ahmet Davutoglu on Thursday, called the latest violence "simply unconscionable." She said al-Assad must go and the international community must unite around a plan for Syria after al-Assad.

Clinton said it is important to give Annan's peace initiative "the last amount of support we can muster."

Davutoglu, whose country has been harshly critical of the al-Assad regime's actions, said all members of the U.N. Security Council must work together to stop the regime's actions.

China and Russia, steadfast regime allies, have stressed their opposition to outside interference in Syria and continue to back Annan's plan. As permanent Security Council

members, the two nations have veto powers and have blocked tough draft resolutions against the regime.

Meeting in Shanghai on Thursday, leaders of Russia, China and four Central Asian nations signed a statement opposing outside intervention in Syria. The statement called for "dialogues that respect Syria's sovereignty, independence and territorial integrity."

The United States plans to send a delegation to Russia this week to press for tough action against the Syrian regime, a senior State Department official said.

At least 15 people have been killed so far Thursday in Syria, the Local Coordination Committees of Syria said.

The United Nations for months has said more than 9,000 people have died in Syria. But death counts from opposition groups range from more than 12,000 to more than 14,000. Tens of thousands of people have been displaced.

Test 5

This test is of headlines from two different news articles discussing the same topic.

First Text

Key al Qaeda leader killed in drone strike

Second Text

Bergen One senior al Qaeda leader left

Test 6

This test compares two identical texts.

First Text

To repeat what others have said requires education. To challenge it requires brains.

Second Text

To repeat what others have said requires education. To challenge it requires brains.

Appendix G

Demo Instructions

This is a manual for the constructed program.

First Text: This button opens a dialog which allows navigation to a text file. Select a text file to load the text into the program. This is the first text, which will be compared to the second text.

Second Text: This button opens a dialog which allows navigation to a text file. Select a text file to load it into the program. This is the second text, which will be compared to the first text.

*Caution is advised: It is possible to loading other file formats into the program. These files are not supported by the programs inner functions and should be avoided.

Choose Algorithm (dropdown box): Select the desired algorithm for the text processing

Process: This button starts the computation of similarity, given two texts are loaded and an algorithm has been chosen.

*The program hangs when computation starts. One must wait until the program finished execution in order to perform further tasks. If it is wished to close the program while in this state, a force close has to be made.

New: The menu option resets all the values in the program and updates the corresponding labels.

Close: This menu option simply closes the program.

About: This menu option should open a window, but have not been implemented.

Appendix H

Installation Instructions

The program needs WordNet to be installed in order to properly work. WordNet must be stored on the C drive.

Specifically: C:\WordNet\2.1\dict\ (This path is located in the file_properties.xml - the path must be changed to use other versions of WordNet)

The program loads the files in the dict folder, which is where the WordNet database files are located.

There are some problems with the JWNL API and the current version of WordNet. This problem consists of some naming issues with files. A functioning dict folder has therefore been provided on the CD. This can be found in the WordNet folder.

Before running the program, replace the provided dict folder with the installed dict folder in WordNet.