

On Text Similarity and the Development of a Language Statistics Server

BJÖRN ANDRIST



**KTH Computer Science
and Communication**

Master of Science Thesis
Stockholm, Sweden 2006

On Text Similarity and the Development of a Language Statistics Server

B J Ö R N A N D R I S T

Master's Thesis in Computer Science (20 credits)
at the IT Program
Royal Institute of Technology year 2006
Supervisor at CSC was Martin Hassel
Examiner was Stefan Arnborg

TRITA-CSC-E 2006:092
ISRN-KTH/CSC/E--06/092--SE
ISSN-1653-5715

Royal Institute of Technology
School of Computer Science and Communication

KTH CSC
SE-100 44 Stockholm, Sweden

URL: www.csc.kth.se

On text similarity and the development of a language statistics server

Abstract

This thesis describes the design and implementation of STATSERV, a language statistics server, as well as TEXTSIM, a system for determining the similarity between texts. Further, this thesis shows the results of a comparison between two various configurations of TEXTSIM; one with and one without linguistic analysis. The aim of the language statistics server is to provide a common service for other NLP applications that could benefit from corpus statistics. The architecture of the server is modular and hence new functionality could be added to the server in the future. Statistics can be added to the server by passing the server new documents to be analyzed. To assure the reliability of the statistics stored on the server, a text should not be allowed to affect the statistics more than once. In order to detect similar documents sent to the server the text similarity system was implemented and integrated into the server. TEXTSIM was implemented using a Machine Learning approach and can be trained on example data. To evaluate and compare the two models of TEXTSIM we used two sets of examples: a set of examples generated automatically from programs and a set of examples acquired from two assessors. Depending on the type of documents, we found the model using linguistic analysis to perform equally well or better than the model not using linguistic analysis.

Likhet mellan texter samt utveckling av en språkstatistikserver

Sammanfattning

Denna rapport beskriver utformningen och implementationen av STATSERV, en språkstatistikserver, och TEXTSIM, ett system för att avgöra likheten mellan texter. Vidare redogör denna uppsats för resultaten av en jämförelse mellan två olika konfigurationer av TEXTSIM; en med och en utan lingvistisk analys. Syftet med språkstatistikservern är att tillhandahålla tjänster gemensamma för andra tillämpningar inom språkteknologi som kan nyttja korpusstatistik. Arkitekturen av servern är modulär varför ny funktionallitet kan läggas till i efterhand. Statistik kan adderas till servern genom att skicka nya dokument för analys. För att säkerställa validiteten av statistiken som lagras på servern får en text inte påverka statistiken mer än en gång. I syfte att detektera liknande dokument som skickas till servern implementerades textlikhetssystemet som därefter integrerades i servern. TEXTSIM implementerades med en maskininlärningsmetod och kan tränas på exempeldata. För att jämföra de två konfigurationerna av TEXTSIM användes två exempelmängder: en mängd automatiskt genererad från program och en mängd insamlad från två försökspersoner. Den konfiguration som använde lingvistisk analys visade lika bra eller bättre förmåga att uppskatta likheten mellan texter beroende på texternas typ.

Contents

1	Introduction	1
1.1	Background	1
1.2	Problem	3
1.3	Aim	4
1.4	Thesis outline	4
2	Method survey	5
2.1	Identification of similarities in texts	5
2.2	Developing a language statistics server	5
3	Theoretical background	7
3.1	Machine learning	7
3.1.1	The perceptron	7
3.1.2	Training and evaluation	9
3.2	Resemblance and containment	9
3.3	A method for unbiased estimation	10
3.3.1	Producing the sketches	11
3.3.2	Fingerprinting	11
3.3.3	Rabin fingerprinting scheme	11
3.3.4	Fingerprint collisions	11
4	Implementing a text similarity system	13
4.1	Introduction	13
4.2	Similarity between documents	13
4.3	Features for document comparison	15
4.4	A naive approach	15
4.5	An ML approach	16
4.5.1	A perceptron model	16
4.5.2	Training	17
4.6	Implementation	18
4.6.1	Sketch parameters	18
4.6.2	Counting the duplicated shingles	19
4.6.3	Complexity and performance	19
5	Evaluation of the text similarity system	21
5.1	Introduction	21
5.1.1	Hypothesis	22
5.1.2	Method overview	22
5.2	System configurations	22

5.3	Data sets	22
5.3.1	Reference documents	23
5.3.2	Document variants	23
5.3.3	Example types	23
5.3.4	The generated example set	24
5.3.5	The assessor example set	24
5.4	Performance	24
5.4.1	Performance on generated examples	25
5.4.2	Performance on the assessor example set	25
5.5	Discussion	25
6	Architecture and design	27
6.1	System overview	27
6.2	Behavioral view	27
6.3	Layered architecture	28
6.4	Interaction layer	28
6.4.1	The Java servlet technology	28
6.4.2	Choice of communication protocol	30
6.4.3	The client request	30
6.4.4	Generating XML	30
6.5	Application layer	31
6.6	Persistence layer	31
6.6.1	Connection pooling	31
6.6.2	Indexing	31
6.6.3	Stored procedures	31
6.6.4	Denormalization	32
6.7	Utility packages	32
6.7.1	Caching	32
6.7.2	XML	32
6.8	Design patterns	32
6.8.1	Controller	33
6.8.2	Concrete factory	33
6.8.3	Value Object	33
6.8.4	Facade	33
6.8.5	Data Access Object	33
6.8.6	Strategy	33
6.9	Modules	34
6.9.1	Extending the server with new modules	34
6.9.2	A module for lemmas and tags	34
7	Discussion	35
7.1	Summary	35
7.2	Future work	35
7.2.1	The language statistics server	35
7.2.2	Text similarity	36
8	Acknowledgements	37
	References	39

Chapter 1

Introduction

1.1 Background

Language statistics is of great importance in Natural Language Processing (NLP) as a substitute or complement to rule based methods. In our case, language statistics will refer to statistics calculated on written language. The access to digital documents of natural language increases daily. These documents are all potential subjects for linguistic applications using statistical methods. The field of applications grows rapidly. Statistical NLP methods can be used in application methods, acquisition methods, and evaluation methods [Nivre 02]. Here follows a few examples of frequently used types of statistics within NLP:

Term frequency (tf) – the number of occurrences of a term in a document or a collection of documents.

Inverse document frequency (idf) – a measure of how rare a term is in a collection of documents.

Term weighting – weights the term in a document according to its relevance to some certain criteria. A well known implementation is the $tf \cdot idf$ weighting scheme [Jurafsky & Martin 00].

Such types of statistics can be used in applications such as word class disambiguation [Church 88], automatic summarization [Lin & Hovy 97], machine translation, literary detective work such as authorship attribution [Oakes 98] etc.

Language statistics can be based on data in a single document or on a corpus (a large and structured collection of documents). In this thesis the former will be referred to as *local statistics* and the latter *reference statistics*. Some types of statistics require both local and reference statistics. E.g. the term weighting scheme, $tf \cdot idf$ weighting, mentioned above requires tf which can be based on local statistics and idf which is based on reference statistics.

Methods to calculate local statistics can be implemented as a part of statistical NLP systems. When using reference statistics in a certain application, each instance of that system needs to hold a copy of the reference statistics. Many NLP systems are based on similar types of statistics, therefore a common *language statistics server* could preferably provide the systems with statistics and thus facilitate the development of the systems. A set of requirements would need to be fulfilled for a language statistics server to be useful. Here follows the most important functional and non-functional requirements of such a server:

1. **Parallel request handling.** Incoming requests to the server must be handled concurrently by separate processes or threads.
2. **Modularity.** New statistical modules responsible for providing a certain type of statistics should be able to extend the server functionality.
3. **Handling of local and reference statistics.** Each module should be able to provide local and reference statistics relevant for a certain document passed to the server.
4. **Handling of different corpora.** Reference statistics are calculated on documents from one or many sets of documents, i.e. corpora. A request for reference statistics must include references to the document sets, on which the statistics should be calculated. Further, it should be possible to add and remove document sets.
5. **Standardized protocol for request and response handling.** The protocol for communicating with the server should be based on both standardized and platform independent techniques.
6. **Performance.** Response time is a critical factor especially when adding and fetching reference statistics. Therefore, response times should be kept at a minimum.
7. **Automatic detection of similar documents.** In order to assure the reliability of the references statistics, a text must not affect the reference statistics more than once. This requires an efficient and automated mechanism for identifying similar texts. All incoming texts are written in Swedish ¹.

The requirements listed above concern many interesting and profound aspects and are subjects for further research. The first six requirements have been fulfilled by implementing conventional techniques for building web server applications. It is however the last requirement concerning text similarity, that this project has aimed to investigate.

Methods for determining the similarities of documents can be found in the field of Information Retrieval, IR. Two well known applications are:

- SCAM: a copy detection mechanism for digital libraries [Shivakumar et al. 95]
- the identification of nearly identical documents used by search engines in the purpose of returning only one variant of similar texts. [Broder 97]

In this Master's project two mathematical notions called *resemblance* and *containment* [Broder 97] have been examined for determining the similarity of documents. *Sketches* of documents, created by hashing sequences of strings, are compared to determine the resemblance and the containment. This technique originates from the work by Andrei Broder [Broder 93] [Broder 97] and can be applied to different types of data such as MIDI documents [Mitzenmacher & Owen 01]. Our language statistics server is restricted to documents written in Swedish. Therefore we can make use of linguistic analysis in an attempt to improve the technique using resemblance and containment.

A grammar checker for Swedish called Granska [Domeij et al. 99] can be used for linguistic analysis of documents. Granska can, among other things, be utilized for *tokenizing*, *lemmatizing* and *part-of-speech tagging*.

¹Theoretically, our method could be used for any language, assuming that an underlying analyzation system is used for the language in question.

1.2. PROBLEM

The result from tokenizing a document is the list of *tokens* (the words and delimiters etc) contained in the document. While the result from lemmatizing a document is a list of *lemmas*. A lemma is the canonical form of a token. E.g. lemmatizing transforms all verb tokens to their infinitive form, noun tokens to their singular form etc. Part-of-speech tagging (also known as PoS tagging) is the process of determining the *part-of-speech tag* or *PoS tag* for a certain token in a sentence. A part-of-speech tag is a lexical category. Table 1.1 gives an example of a sentence transformed into tokens, lemmas, and part-of-speech tags.

Original text:	<i>Han springer fortare!</i>
Tokens:	Han springer fortare !
Lemmas:	han springa fort !
Part-of-speech tags:	pn vb ab mad

Table 1.1. A sentence transformed into tokens, lemmas and part-of-speech tags. (pn, vb, ab and mad are PoS tags as used in Granska.)

We aim to develop a system to be capable of mirroring the human perception of similarity between documents. Systems which are able to learn a concept, in our case the concept of similarity between documents, can preferably be implemented using machine learning (ML) algorithms [Mitchell 97]. A well established type of ML algorithms is the Artificial Neural Network (ANN) algorithms [Callan 99]. An ANN can be trained to learn a concept given a set of training instances. The learning process (i.e. training) of an ANN can be either unsupervised or supervised. Unsupervised learning is based on algorithms which can improve by processing input with unknown output. Supervised learning, on the other hand, requires the training data to contain both the input and the required output for each input pattern. When an ANN produces an output which differs from the required output, the ANN is updated in the purpose of learning to classify the instance correctly. Training data for a supervised ANN can be acquired by having people determine the similarity between documents. In this project we have used supervised learning solely.

1.2 Problem

To build a successful language statistics server, the requirements listed above need to be fulfilled.

How can the architecture of a modular language statistics server which provides reliable local and reference statistics be defined?

We aim to build an efficient system for determining the similarity between documents for use in the statistics server. Examples of such systems have successfully been implemented with the use of containment and resemblance. We wish to see if an ANN system using containment and resemblance, treating texts written in Swedish, can be improved by using tokens, lemmas and part-of-speech tags as input.

Can an ANN system, with the use of containment and resemblance, based on tokens, lemmas, and PoS tags outperform a similar system solely based on tokens?

1.3 Aim

The aim of this Master's project is:

1. to state how an architecture and a design of a modular language statistics server can be defined.
2. to implement an effective text similarity system to be used in conjunction with the language statistics server in the purpose of detecting similar texts.
3. to examine whether identification of similar texts using resemblance and containment can be improved by using tokens, lemmas and PoS tags instead of solely using tokens.

1.4 Thesis outline

The rest of this thesis is outlined as follows: Chapter 2 describes the overall process of this project. Chapter 3 presents the theoretical foundation that the text similarity system is based on. Chapter 4 describes the text similarity system that was implemented in this project. Chapter 5 examines whether a text similarity system based on tokens, lemmas and PoS tags can outperform a similar system based on tokens solely. Chapter 6 is a technical report of the architecture and the design of the language statistics server that was implemented in this project. Finally, Chapter 7 includes a summary of the achieved results and conclusions of this project and suggests future work.

Chapter 2

Method survey

This project has involved three distinct tasks:

- developing a text similarity system able to identify similarities between documents.
- examining whether the text similarity system could be improved by using linguistic analysis of the texts.
- defining an architecture and a design of a language statistics server.

2.1 Identification of similarities in texts

Within this project, a system named `TEXTSIM` was developed in Java for finding similarities in text documents. `TEXTSIM` is based on resemblance and containment combined with a supervised machine learning algorithm. `TEXTSIM` can be trained on any combination of tokens, lemmas, and PoS tags. Two instances of the `TEXTSIM` system were trained: one based on tokens and one based on tokens, lemmas, and PoS tags. Evaluation data collected from two assessors were used for comparison between the systems. This will be discussed in detail in Chapter 6.

2.2 Developing a language statistics server

To define an architecture and a design of a language statistics server a prototype which fulfills the requirements outlined in Chapter 1 was developed in Java and SQL. The prototype is named `STATSERV` and realizes the suggested architecture. `TEXTSIM` was integrated into `STATSERV` to classify documents as similar or non-similar to the existing documents in the database.

To reach a sound architecture, object oriented principles were used throughout the development process. Design and architectural patterns were used and are documented in this thesis (see Chapter 6).

The development process was iterative where each iteration involved designing, implementation and testing. Chapter 6 is a technical report of the architecture and the design of `STATSERV`. The final prototype was tested with multiple concurrent clients requesting reference statistics and local statistics from document sets which contained more than 10.000 documents. A module called `LEMMA TAGMODULE` was developed in the purpose of testing the modularity of the server.

Chapter 3

Theoretical background

This chapter describes the theory behind the text identification system TEXTSIM which is used by STATSERV (the language statistics server) to identify similar texts.

3.1 Machine learning

Mitchell (1997) states: “Machine Learning is the study of computer algorithms that improve automatically through experience.”

Given a set of training *instances*, an ML *model* can be trained to grasp a particular *concept*. Learning can be divided into two categories: supervised learning and unsupervised learning. Supervised learning requires training *examples*. An example consists of both an instance and the desired output target. Instances will also be referred to as input patterns. During training, the model is updated whenever it generates an output that differs from the desired output. When a model has been generated the aim is that the model can produce correct output on unseen input patterns.

A well approved category of ML algorithms is the Artificial Neural Network (ANN) algorithms [Callan 99]. Successful applications using ANNs include, among others, face recognition and optical character recognition [Mitchell 97]. ANNs consist of one or many connected layers of computing units called neurons. Each neuron has one or several input nodes and one output node. When input is received from the input nodes, the neuron generates an output signal depending on the input values. The input nodes are normally attached with a corresponding *weight* which represents the strength of the connection between the input node and the neuron.

In order to compare ML models, trained to learn the same concept, it is essential to be able to *evaluate* the models. Many sound strategies for evaluating ML models have been obtained from the field of statistics [Witten 00].

3.1.1 The perceptron

The perceptron, invented by Frank Rosenblatt (1958), is a single layer ANN which can learn linearly separable functions. A perceptron consists of a single neuron connected with an arbitrary number of real value input nodes. Figure 3.1 illustrates a perceptron.

The input nodes are represented by a vector \vec{x} . A weight is attached to each node to represent the strength of the connection between the input node and the neuron. The weights are also represented by a vector \vec{w} of real number weights. A linear combination of

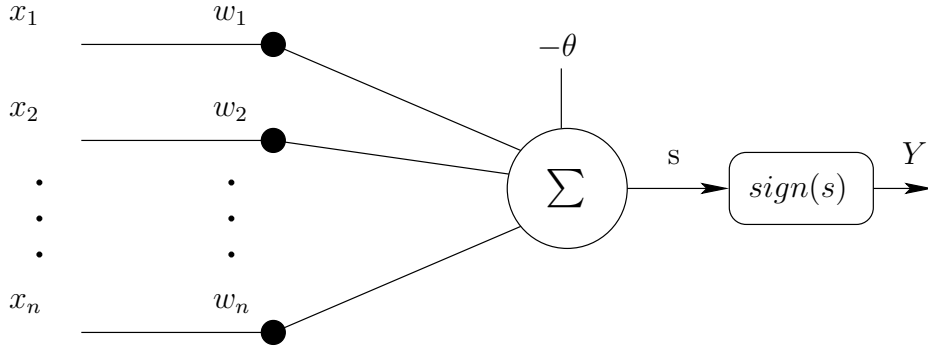


Figure 3.1. A perceptron using the $sign$ function to determine its output. $\vec{x} = (x_1, x_2, \dots, x_n)$ are inputs while $\vec{w} = (w_1, w_2, \dots, w_n)$ are the attached weights. The sum $s = \sum_{i=1}^n x_i \cdot w_i - \theta$, is passed as an argument to the activation function $sign : \mathbb{R} \mapsto \{1, -1\}$.

the n inputs and the corresponding weights produces an activation level given by

$$\sum_{i=1}^n x_i \cdot w_i = \vec{x} \cdot \vec{w}.$$

For each input pattern presented for the perceptron, an *activation function* determines the output of the perceptron based on the activation level and a threshold value θ . We use the $sign$ function as the activation function:

$$sign(x) = \begin{cases} 1 & \text{if } x > 0, \\ -1 & \text{otherwise.} \end{cases}$$

Given the threshold value θ and the input weights \vec{w} , the output Y of the perceptron for input \vec{x} is given by

$$Y(\vec{x}) = sign(\vec{x} \cdot \vec{w} - \theta).$$

Further, if the desired output target Y_t is known, the error e is given by

$$e(\vec{x}) = Y_t(\vec{x}) - Y(\vec{x}).$$

The "knowledge" of the perceptron is stored in its weights and its threshold value. For each training example, the perceptron uses the *perceptron learning rule* to update its weights:

$$w'_i \leftarrow w_i + \alpha \cdot e \cdot x_i$$

where α is a positive constant known as the learning rate. In a similar manner the threshold value is updated by

$$\theta' \leftarrow \theta - \alpha \cdot e.$$

If the output Y equals the desired target Y_t , the weights and the threshold are intact. Learning only takes place when the output differs from the target. A *cycle* is complete when all available training examples have been processed once.

The learning rate and the number of training cycles are two parameters that entail optimizing when generating a perceptron model. The perceptron described above can be utilized for classification where the task, given an input pattern, is to specify the class of

3.2. RESEMBLANCE AND CONTAINMENT

the input. E.g., the output values $+1$ and -1 can represent two different classes. When using a perceptron we first need to define the input encoding as well as the output encoding for the applied problem.

Even though the perceptron can only learn linearly separable functions it has achieved more than acceptable empirical results [Bylander 98].

3.1.2 Training and evaluation

Training ML models requires training data. The more reliable training data we have, the greater the chances are that the model will learn the desired concept. A portion of the data is usually reserved for predicting the error rate on unseen data, this method is often referred to as the *holdout procedure*. None of the data involved during training can be a part of the evaluation process, because evaluation on training data normally leads to overoptimistic predictions. If we want to evaluate the error rate on never before seen data, the evaluation data also is required to be never seen before. However, during the training process it might be interesting to know how well a model performs on the training data set. The error rate calculated on training data is often referred to as the *resubstitution error rate*.

When dividing the data set into training data and test data, both samples need to be representative for the entire data set. *Stratified sampling* (or stratification) ensure that each class in the data set is properly represented in each sample. To maximize the use of the data and to minimize the risk of biased test data (i.e. non-representative test data), stratification is usually combined with *cross-validation*. With cross-validation, the data set is partitioned into k number of cells (or folds). One fold is selected as test data while the other $k - 1$ folds are used for training the model. The error rate is then calculated on the fold selected for testing. The procedure is repeated until every fold has been used as test data once. Finally, the mean value of the error rates can be calculated.

Cross-validation is a well established statistical method [Witten 00] for evaluating models. When using stratification when folding the data, the complete process is referred to as *stratified k-fold cross-validation*. It is conventional to use 3, 5, or, 10 folds. Stratified 10-fold cross-validation greatly reduces the risk for biased test data while making use of a large portion of the data for training. After the evaluation process all data can be used for training to further improve the model.

3.2 Resemblance and containment

In order to use the perceptron described above for text identification, we need to specify which features to use as input. Here we examine two plausible features, the mathematical concepts: resemblance and containment [Broder 97].

Resemblance and containment quantify the similarity between two documents. The degree of resemblance and containment is represented by a real number $x \in [0, 1]$. A resemblance value close to 1 indicates a high level of similarity between two documents. A containment value close to 1 indicates that one of two documents nearly contains the other document. Resemblance and containment are defined as set intersection problems [Broder 97]. A short summary is given below.

Consider a document as a sequence of tokens. A token could be a word, a punctuation mark, etc. We call a subsequence of a document a *shingle*. Further, a shingle of a specific size w will be referred to as a *w-shingle*.

Let $S(A, w)$ represent the multiset of w -shingles for a document A . A multiset has the properties of an ordinary set except for that each element has a multiplicity. E.g. $\{a, c, b\}$ is

an ordinary set and a multiset, while $\{c, a, a, b\}$ is only a multiset because of the multiplicity of the element a .

We refer to $S(A, w)$ as a *w-shingling*. For the document

$A = (\text{en rysk docka i en rysk docka är en rysk gumma .}),$

the multiset of w-shingles would be

$$S(A, 3) = \{(\text{en rysk docka}), (\text{rysk docka i}), (\text{docka i en}), \\ (\text{i en rysk}), (\text{en rysk docka}), (\text{rysk docka är}), (\text{docka är en}), \\ (\text{är en rysk}), (\text{en rysk gumma}), (\text{rysk gumma .})\}.$$

Given two documents A and B and a fixed shingle size of w , the resemblance can be computed by dividing the number of w-shingles in common with the total number of distinct w-shingles:

$$r_w(A, B) = \frac{|S(A, w) \cap S(B, w)|}{|S(A, w) \cup S(B, w)|}.$$

To compute the containment of A in B , divide the number of w-shingles in common, with the number of w-shingles in A :

$$c_w(A, B) = \frac{|S(A, w) \cap S(B, w)|}{|S(A, w)|}.$$

Here follows a short example. Let

$A = (\text{en rysk docka i en rysk docka är en rysk gumma})$

and

$B = (\text{en rysk docka är en rysk gumma}).$

With a shingle size of 3, the number of 3-shingles in common is 5 and the total number of distinct shingles is 9, and hence the resemblance of A and B is

$$r_3(A, B) = \frac{5}{9}.$$

The containment of A in B is

$$c_3(A, B) = \frac{5}{9}.$$

And finally, the containment of B in A is

$$c_3(B, A) = \frac{5}{5}.$$

3.3 A method for unbiased estimation

When attempting to use the method described in the past section as a text similarity system for our statistics server, two problems arise:

- It is necessary to store the text unabridged. This is problematic because the texts we want to analyze might be copyrighted, and therefore it would be illegal to store it in a manner that it could be restored.

3.3. A METHOD FOR UNBIASED ESTIMATION

- The comparison between every shingle in all texts we want to compare is a time consuming task which involves many string comparisons.

Therefore we need to find an effective method of representing the documents which solves these two problems.

Instead of making an exact calculation defined in the previous section, it is adequate to make an unbiased estimate of the resemblance and the containment. To estimate the resemblance and the containment of two documents, it is sufficient to store only a *sketch* of the documents. The remainder of this section aims to describe how to produce and handle these sketches. The content in this section is derived from [Broder 97][Broder 93][Rabin 81].

3.3.1 Producing the sketches

A sketch of a document is produced by fingerprinting (i.e. hashing) each shingle contained in the document. That is, for each shingle, we compute a semi-unique value representing the shingle. Each shingle is now represented by an integer. From now on a shingle will refer to the fingerprinted version of the shingle. We designate a suitable integer m and select those shingles that are divisible by m , i.e. the shingles which are congruent to 0 modulo m . A low value of m will result in larger sketches and hence a higher accuracy when estimating the similarity. However, a low value of m requires a greater amount of storage space which in turn decreases the performance. The value of m can easily be configured at deploy time, nevertheless sketches created with different values of m cannot be compared with each other.

3.3.2 Fingerprinting

The hash function we have used for creating the sketches is the Rabin fingerprinting method [Rabin 81]. The fingerprinting method has two main features that makes it useful in this context: it can be computed in linear time, and the probabilities for collisions to occur can be calculated and understood and therefore it is possible to determine a proper size of the fingerprints.

3.3.3 Rabin fingerprinting scheme

The Rabin fingerprinting method is based on arithmetics with large polynomials over \mathbb{Z}_2 . The scheme is thoroughly described in [Rabin 81] and [Broder 93]. Here follows a short repetition of the scheme.

We represent a binary string S of size n , $S = (s_1, s_2, \dots, s_n)$, by a polynomial $P_S(x)$ over \mathbb{Z}_2 according to

$$P_S(x) = s_1x^{n-1} + s_2x^{n-2} + \dots + s_{n-1}x^1 + s_n.$$

When producing the fingerprint of the string S , we compute $P_S(x) \bmod Q(x)$. Where $Q(x)$ is an irreducible polynomial over \mathbb{Z}_2 . A polynomial is irreducible if it cannot be factored into any nontrivial polynomials (non-constant polynomials) [Biggs 02]. $Q(x)$ can be chosen arbitrarily at random.

3.3.4 Fingerprint collisions

A fingerprint collision occurs when two strings, S_1 and S_2 , are different, but their fingerprint values, $f(S)$, are equal:

$$S_1 \neq S_2 \text{ and } f(S_1) = f(S_2).$$

Note, if $f(S_1) \neq f(S_2) \Rightarrow S_1 \neq S_2$. Every collision that occurs will decrease the reliability of the estimation, it is therefore important to minimize the number of collisions. The degree k of $Q(x)$ ($k = \deg Q(t)$) determines how many bits with which we choose to represent our fingerprint. To minimize the memory allocation, we aim to use the smallest possible fingerprints (small value for k) but with sufficiently low collision probability. The probability for two distinct strings, S_1 and S_2 , to obtain the same fingerprint value can be estimated to

$$\Pr(f(S_1) = f(S_2) \mid S_1 \neq S_2) < \frac{\max\{\deg P_{S_1}(x), \deg P_{S_2}(x)\}}{2^{k-1}}.$$

To determine a proper size of k , suitable for our text identification system, it is necessary to know the size of a typical document and the size of the w -shingles. For a document $D = \{S_1, S_2, \dots, S_n\}$, the number of unordered pairs of strings in D is $n(n-1)/2$. We assume that every string has a length less than m bits. Then the probability that any pair of strings in D would obtain the same fingerprint value can be estimated to

$$\frac{n(n-1)}{2} \cdot \frac{m}{2^{k-1}} < \frac{n^2 m}{2^k}.$$

In the subsequent chapter, we shall see how this estimation was used when determining an appropriate number of bits to represent shingles.

Chapter 4

Implementing a text similarity system

TEXTSIM is a system developed in Java for finding the similarity between documents. It provides a class library/API as well as a set of stand-alone applications. The language statistics server, STATSERV, uses the TEXTSIM-API in order to identify similar texts in the reference statistics. The system was implemented with the intention to be used by STATSERV. The functionality, the design strategies, and the parameter choices are therefore discussed seen from this perspective. Nevertheless, TEXTSIM can be used separately from STATSERV in other contexts.

4.1 Introduction

To assure the reliability of the reference statistics in STATSERV, a text should not be allowed to affect the statistics more than once. For each *incoming document* sent to the server with the purpose of affecting the statistics, we need to assure that the document is neither similar nor identical to one of the previously processed files. Therefore a mechanism for automatic detection of similar documents is required. The process of determining whether an incoming document should be classified as a unique document within a document set is performed in two steps:

Step 1: to identify the most similar document existing in a document set.

Step 2: to determine whether this incoming document differs sufficiently from the most similar document found in Step 1 in order to be accepted as a new document.

By dividing the process into two steps, each step can be optimized independently. For example, the comparison in Step 1 may be less comprehensive than the comparison in Step 2 for reasons of performance. This is of greater importance when larger document sets are examined. The main focus when implementing TEXTSIM has been on the more general problem described in Step 1. Step 2 can be adjusted to suite the context in which it is intended to operate. In the process of searching for similar documents, we need to define the concept of similarity more precisely.

4.2 Similarity between documents

Let the symbol Ω represent the set of all documents. Every document set is a subset of Ω . Let $sim : \Omega \times \Omega \mapsto [0, 1]$ be a function that denotes the degree of similarity between two

documents. We define that the value of $sim(a, b)$ is 1 iff a and b are equal:

$$sim(a, b) = 1 \Leftrightarrow a = b.$$

Given a document $x \in \Omega$ and a document set $D \subseteq \Omega$, we aim to find the document in D which is most similar to x . We denote this as a function $\phi : \Omega \mapsto D$ given by

$$\phi(x) = \operatorname{argmax}_{d \in D} sim(x, d).$$

It is possible that $x \in D$, if so, we aim to find x itself. In order for our similarity function sim to be useful, we claim that a *transitive* relation \mathcal{R} on a document set D , given by

$$a\mathcal{R}_x b \Leftrightarrow sim(x, a) > sim(x, b)$$

where $a, b \in D$, must exist for every document $x \in D$. The transitive property of \mathcal{R} implies that

$$\text{if } a\mathcal{R}_x b \text{ and } b\mathcal{R}_x c \Rightarrow a\mathcal{R}_x c.$$

An alternative way of using the similarity function (sim) is by constructing a test with three documents: an incoming document $x \in \Omega$ and two other documents $a, b \in D$. The test determines which one of the documents a or b that is most similar to x . We define this test as a function $\psi : \Omega \times D \times D \mapsto D$ given by:

$$\psi(x, a, b) = \begin{cases} a & \text{if } sim(x, a) > sim(x, b), \\ b & \text{otherwise.} \end{cases}$$

Given the existence of both the function ψ and the relation \mathcal{R} , the function ϕ can be implemented using Algorithm 1.

Algorithm 1 FindSimMax

Require: $x, D[1 \dots n]$ where $n > 1$
procedure FINDSIMMAX($x, D[1 \dots n]$)
 $d \leftarrow D[1]$
 for $i \leftarrow 2, n$ **do**
 $d \leftarrow \psi(x, d, D[i])$
 end for
 return d
end procedure

Given an incoming document $x \in \Omega$ and a document set D , our text identification task described in the previous section can now be summarized by our new notation. The two steps are:

Step 1: to find the document $d = \phi(x)$ either by using sim , or by using Algorithm 1 with ψ .

Step 2: to determine whether d differs sufficiently from x in order to be accepted as a new element in D .

The different approaches using sim , ϕ and ψ were useful when implementing TEXTSIM and determining the methods for collecting training and evaluation data.

4.3 Features for document comparison

There are many potential features that could be of use when comparing documents, such as document size, token overlap, etc. However, the focus in this project, as stated in Chapter 1, has been on resemblance and containment for their demonstrated ability to denote similarities between documents [Broder 97]. When comparing two documents, A and B , the following features are of interest:

- the resemblance of A and B , $r(A, B)$, which determines in what degree the documents resemble one another.
- the containment of A in B , $c(A, B)$, which is required to detect those documents which are contained in previously added documents.
- the containment of B in A , $c(B, A)$, which is required to detect documents which contain previously added documents.

By using GTA, Granska Text Analyzler [Domeij et al. 99], for the analysis of documents, a document A can be described in three different forms:

- by its token form, A_{token} ,
- by its lemma form, A_{lemma} ,
- and by its tag form, A_{tag} .

Thus, we let A_t represent a form of the document for a certain type $t \in \{token, lemma, tag\}$. For each pair of documents, we can choose to compare the tokens, the lemmas, and/or the tags. The resemblance and the containment can in turn be based on the different forms. We let $r(A_t, B_t)$ represent the resemblance of A and B for a certain type $t \in \{token, lemma, tag\}$. The containment $c(A_t, B_t)$ is defined analogously.

4.4 A naive approach

Given the resemblance and the containment features, an approach for combining them effectively to approximate sim is needed. A possible approximation of sim is the arithmetic mean value of the relevant containment and resemblance values. Let X_T be the mean value for a set of types T :

$$X_T(A, B) = \frac{1}{|T|} \sum_{t \in T} \overline{(r(A_t, B_t), c(A_t, B_t), c(B_t, A_t))}$$

Using our new notation, we can now define different approximations of sim and compare them with each other. For example, let sim' represent the approximation of sim when using tokens:

$$sim'(A, B) = X_T(A, B) \text{ where } T = \{token\}.$$

When combining different types of data such as tokens, lemmas and tags, we simply use the mean value to combine the result from the multiple types of data. For example, an approximation of sim using tokens, lemmas and tags is given by:

$$sim''(A, B) = X_T(A, B) \text{ where } T = \{token, lemma, tag\}$$

This method for document comparison was implemented in TEXTSIM and can be used as an alternative to the more extensive ML approach, described below. Unfortunately, within

the time limits of this project, it has not been possible to compare this naive approach with the ML approach.

The naive approach is capable of combining different types of data. However, there are some obvious problems with this method:

1. It assumes that every type of value ($c(A_t, B_t), r(A_t, B_t)$, etc) is equally important. An improvement would be to attach a real number weight to each value.
2. If it were improved with attached weights as suggested above, the weights would have to be configured to attain the best possible estimation of the function sim .

4.5 An ML approach

We applied a machine learning (ML) approach in order to extend the model outlined above by using weights and a learning algorithm to adjust the weights. For the algorithm to be able to adjust the weights correctly, training examples were needed.

As stated in Chapter 1, we aimed to develop a system capable of mirroring the human perception of similarity between documents. Therefore the training and evaluation data should preferably be comprised of examples from human subjects. Furthermore, the training data ought to be reliable and preferably easy to generate. In this project, reliable data is data from a subject who produces approximately the same results when repeating a test. Two methods for generating training examples were considered:

Method 1: Let a subject estimate the similarity between two documents A and B by specifying a real value between 0 and 1. This method determines the actual value of $sim(A, B)$.

Method 2: Give a subject three documents: a reference document R and two other documents A and B . Thereafter the subject specifies which of the documents A or B that he/she considers to be most similar to R . This method determines the value of $\psi(R, A, B)$.

We assumed that Method 2 would generate more reliable data than Method 1. Hence the second method was chosen. Furthermore, it was also possible to automatically generate large amounts of training examples by using hypotheses about the similarity between documents. For example, one hypothesis was that texts that were modified by replacing a certain amount of words with synonyms were more similar to the reference document than a document modified with the same amount of spelling errors. Generation of training data and test data will be discussed in detail in Section 5.3.

Method 2 would not give us the function sim , but as outlined in Section 4.2 we can determine $\phi : \Omega \mapsto D$ with use of Algorithm 1 and $\psi : \Omega \times D \times D \mapsto D$. We used the training data produced with Method 2 to adjust the weights by applying the perceptron training rule.

4.5.1 A perceptron model

Method 2 was chosen for its ability to generate consistent reliable test and training data. A system was needed that could be trained on this type of data (see Section 3.1.1). Such a system was designed and implemented using the perceptron.

The system compares two documents A and B with a reference document R and returns the document that is most similar to R . This is the definition of ψ . We let half of the input

4.5. AN ML APPROACH

nodes represent the comparison of R with A and the other half of the input nodes represent the comparison between R and B . The comparison is based on the containment and the resemblance as outlined in Section 4.3. The perceptron generates $+1$ or -1 depending on its input. An output of $+1$ indicates that $sim(R, A) > sim(R, B)$ while an output of -1 indicates that $sim(R, A) < sim(R, B)$.

A perceptron for handling document comparisons using tokens solely was built using 6 input nodes. Each input node receives one of the resemblance or containment features derived from the comparison of either R and A , or R and B . The design of the perceptron using comparisons based on tokens is illustrated in Figure 4.1. When using resemblance and containment based on tokens, lemmas and tags, a perceptron with a total of 18 input nodes was used: six for tokens, six for lemmas, and six for tags. Each input node is attached with a weight that can be adjusted by using the perceptron training rule with the training data.

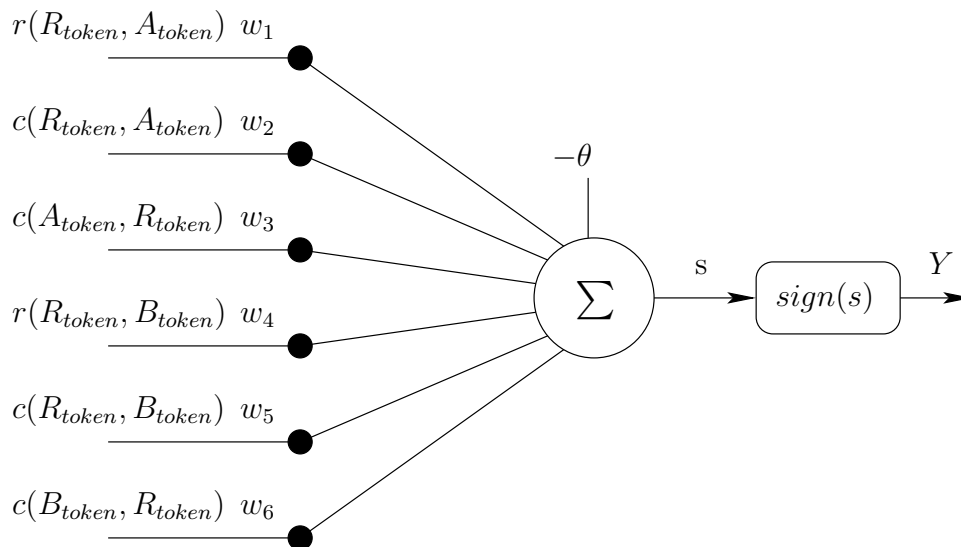


Figure 4.1. A perceptron using resemblance and containment based on tokens as input. The perceptron outputs $+1$ to indicate that $sim(R, A) > sim(R, B)$ and -1 to indicate that $sim(R, A) < sim(R, B)$.

4.5.2 Training

When training the perceptron in order to adjust the weights and the threshold value, learning rate and the number of epochs need to be determined. An epoch (also known as cycle) is a run through all the training examples. The perceptron may learn to classify the examples in the training set but what is more important is that it can *generalize*, that is, output the correct target for instances it has never seen before.

When generating a perceptron model in TEXTSIM, three different methods are possible to use. We can choose the model:

1. generated after a fixed number of epochs. This method enables us to use all training data for training.
2. that generates the lowest resubstitution error rate after a fixed number of epochs. This method enables us to use all training data for training.

3. that generates the lowest error rate on instances never before seen after a fixed number of epochs. This method requires us to reserve a portion of the data to a validation set, which lowers the number of training examples.

Empirically, we achieved the best results using the third method. However, depending on the training data the other methods might lead to better results, wherefore the choice of method is left as an option for application programmers using TEXTSIM.

4.6 Implementation

This section describes implementation details, such as parameter choices when estimating the resemblance and the containment, and the complexity of the text similarity algorithms.

4.6.1 Sketch parameters

By using resemblance and containment as features for the document similarity method, we took advantage of the efficient estimation approach by using sketches created with the Rabin fingerprinting method (see Section 3.3). A third party library [Owen 04] developed in Java was used to compute the Rabin fingerprints. In order to achieve an optimal system for text similarity, several parameters needed to be configured.

When using the Rabin fingerprint library, we may choose between 32- or 64-bits fingerprints. That is, the degree k of the polynomial $Q(x)$ from Section 3.3 can be either 32 or 64. The polynomials of degree 32 and 64 are represented by a Java `int` and a Java `long` respectively. To determine which option to choose we calculated the probability for collisions to occur using the inequality obtained from Section 3.3.4:

$$\Pr(f(S_1) = f(S_2) \mid S_1 \neq S_2) < \frac{\max\{\deg P_{S_1}(x), \deg P_{S_2}(x)\}}{2^{k-1}}.$$

After inspecting SUC and the training documents we found that the average word length was 8.6. Therefore we approximated the word length to 10. Then, by using 16-bit Unicode characters, an average word is 160 bits long. When using w -shingles with $w = 3$ the length of m is 480 bits. We estimate the size of a typically large document to contain 1000 words. If we choose to merely use every 10^{th} shingle from the document, it will result in 100 shingles of 480 bits each. When comparing two sketches we obtain a total of 200 shingles. The probability that a collision would occur when comparing the two sketches using 32-bit fingerprints can then be estimated to

$$\frac{200^2 \cdot 480}{2^{32}} < 2^{-7} \approx 0.7\%$$

while using a 64-bit fingerprint results in an estimated probability of

$$\frac{200^2 \cdot 480}{2^{64}} < 2^{-41}.$$

When using the 64-bit version, it is very unlikely to have collisions at all. The 32-bit version gives us a higher probability that collisions could occur, but it is still sufficiently low considering the fact that one collision in a sketch of 100 shingles still has a very small impact of the total comparison between sketches. Hence, we opted for the most efficient method, and the 32-bit version was chosen.

4.6.2 Counting the duplicated shingles

When computing the resemblance and the containment of two sketches, S_1 and S_2 , it is necessary to count the duplicated shingles in S_1 and S_2 . We define a shingle to be a *duplicate* if it is contained in both S_1 and S_2 . Algorithm 2, which is similar to a merging algorithm, returns the number of duplicated shingles. The sketches, S_1 and S_2 , are represented by sorted arrays of shingles.

Algorithm 2 Counts the duplicated shingles from two sketches

Require: S_1, S_2 are sorted

```

procedure COUNTDUPLICATES( $S_1[1 \dots n_1], S_2[1 \dots n_2]$ )
   $i, j \leftarrow 1$ 
   $d \leftarrow 0$ 
  while  $i \leq n_1 \wedge j \leq n_2$  do
    if  $S_1[i] < S_2[j]$  then
       $i \leftarrow i + 1$ 
    else if  $S_1[i] > S_2[j]$  then
       $j \leftarrow j + 1$ 
    else ▷ A duplicate was found
       $i \leftarrow i + 1$ 
       $j \leftarrow j + 1$ 
       $d \leftarrow d + 1$ 
    end if
  end while
  return  $d$ 
end procedure

```

Algorithm 2 runs in $O(n)$ where n is the total number of elements in the sketches, i.e. the arrays of shingles. For this algorithm to be applicable, it is required to have the two arrays sorted before the algorithm is executed. We guarantee this by only generating sorted unmodifiable sketches.

4.6.3 Complexity and performance

It was required that the text identification was efficient in order to be used by STATSERV for handling large document sets (>10.000). When classifying an incoming document x in relation to a document set D , two main issues are of interest:

1. creation of the sketch(es) of x . One sketch per type $t \in \{token, lemma, tag\}$ is required.
2. comparison of the new sketch(es) of x with all sketches contained in D .

Recall from Section 3.3 that a sketch is comprised of a set of w -shingles. We need to generate a sketch for each type $t \in \{token, lemma, tag\}$. Thus to generate one sketch for a certain type t of a document containing n tokens, we need to compute n w -shingles using the Rabin fingerprinting method. The complexity of computing n w -shingles is $O(nw)$ [Broder 97]. Hence, generating sketches for the set of types $T \subseteq \{token, lemma, tag\}$ for a document can be done in $O(|T|nw)$.

Further we are interested in the complexity of comparing x with every document in D . In order to count the number of duplicates when computing the resemblance and the

containment for a sketch comprised of n shingles, we used algorithm 2 that runs in $O(n)$. The complexity of comparing two documents using sketches of types T is therefore $O(|T|n)$. Finally, the complexity of comparing document x with all documents in D is $O(|T||D|n)$.

A test was performed on a Sparc UltraAX-i2 (CPU: 500 Mhz) when using 3-shingles and sketches for tokens, lemmas and tags, i.e. $|T| = 3$. We used a document set containing 10.000 documents with an average size of 200 tokens and an incoming document of equal size. Before the test started, x had previously been analyzed and the sketches of the documents in D were already computed and accessible in main memory. Creating the sketches of x and comparing them with the sketches in the entire document set took 0.8 seconds.

Chapter 5

Evaluation of the text similarity system

Here we evaluate and compare the two variants of TEXTSIM: one based on tokens and one based on tokens, lemmas, and PoS tags.

5.1 Introduction

We used TEXTSIM in order to answer the second question at issue outlined in Section 1.2:

Can an ANN system, with the use of containment and resemblance, based on tokens, lemmas, and PoS tags outperform a similar system solely based on tokens?

Assume the existence of a reference document R and two other documents A and B with the following resemblance and containment values:

$$\begin{aligned}r(R_{token}, A_{token}) &= r(R_{token}, B_{token}), \\c(R_{token}, A_{token}) &= c(R_{token}, B_{token}), \text{ and} \\c(A_{token}, R_{token}) &= c(B_{token}, R_{token}).\end{aligned}$$

Further, assume that document A differs from R in such a manner that certain words in R have been replaced by synonyms. While B is a completely different document however possessing a high degree of token overlap of R . An assessor would most likely choose document A to be the more similar document (see Section 5.4). Though, a system using containment and resemblance based on tokens will not be capable of determining which document that is most similar to R . We can assume that the sequences of PoS tags in A are more similar to R than the PoS tags in B because the synonyms in most cases will not change the PoS tags. Hence a system using containment and resemblance based on tags and tokens would have a greater possibility to determine the correct output target, since:

$$\begin{aligned}r(R_{tag}, A_{tag}) &\geq r(R_{tag}, B_{tag}), \\c(R_{tag}, A_{tag}) &\geq c(R_{tag}, B_{tag}), \text{ and} \\c(A_{tag}, R_{tag}) &\geq c(B_{tag}, R_{tag}).\end{aligned}$$

In a similar manner we can argue that lemmas would provide useful information when document A originates from document R but is expressed in another tense. The sequences

of lemmas in A and R would be highly similar and could therefore be an important complement when the values of resemblance and containment based on tokens are insufficient to determine the difference between A and B .

5.1.1 Hypothesis

Based on the theories outlined above, our hypothesis is that a system based on tokens, lemmas, and tags would outperform a system based solely on tokens. This applies principally when comparing documents where lemmas and tags would provide additional information, such as in the above examples, where documents were modified by synonyms or by tense.

5.1.2 Method overview

Two TEXTSIM-models were configured: one based solely on tokens and one based on tokens, lemmas, and PoS tags. In order to evaluate and compare the models we used two sets of examples:

- The generated example set – a set of examples generated automatically from programs.
- The assessor example set – a set of examples acquired from two assessors.

Further, two evaluation experiments were conducted:

- The first experiment used the generated example set for training as well as testing.
- The second experiment used the generated example set for training and the assessor example set for testing.

5.2 System configurations

Using TEXTSIM, two models for determining the similarity between documents were configured:

Model 1 was based on the perceptron with 6 input nodes. The input was collected from the resemblance and the containment of documents in token form.

Model 2 was based on the perceptron with 18 input nodes. The input was collected from the resemblance and the containment of documents using token, lemmas, and tags.

Further, both models were configured using 3-shingles where every fourth shingle formed the sketches. The learning rate and the number of maximum epochs were determined by evaluating various values of these parameters. The evaluation was conducted by means of one representative example set containing 400 examples. This resulted in a learning rate of 0.07 and a maximum number of epochs of 1000. During training, the perceptron model that generated the lowest error rate on a validation set of instances never before seen was chosen (see item three in Section 4.5.2).

5.3 Data sets

Recall Section 4.3, a training example is comprised of the comparison between two document pairs, (R, A) and (R, B) , where R is the reference document while A and B are often variants of R . In order to generate training examples we commenced by collecting reference documents.

5.3.1 Reference documents

The reference documents that were used during training and testing were collected from two sources:

- KTH News Corpus [Hassel 01], a digital corpus comprising articles from the web sites of widely spread Swedish newspapers, e.g. Dagens Nyheter, Svenska Dagbladet, Aftonbladet.
- the results from an experiment using Trace-it [Severinson-Eklundh & Kollberg 96] where a group of writers were asked to write short essays. Trace-it is a revision analysis tool that automatically stores a revision of a text after every modification of the text.

5.3.2 Document variants

To automatically generate variants of the reference documents we substituted words with synonyms using Synlex and replaced words with misspellings using the Missplel program.

Missplel is a program for generating human like spelling errors [Bigert et al. 03]. It can be configured to produce a great variety of spelling errors. For example, a very common type of misspelling in Swedish is a compound word that has been incorrectly split into two or more words. One can also configure Missplel to only substitute words so that the PoS tag of a word is changed and vice versa. These features were useful when generating data.

Synlex is an on-line Swedish dictionary of synonyms constructed by having people vote for the synonymity of possible synonym pairs [Kann & Rosell 05]. The dictionary is available as a web application or can be downloaded as an XML file. A minor program was implemented to replace a certain amount of the words in a document by synonyms from the XML file. This technique will most likely result in a great amount of inappropriate replacements. For example, the probability for an inappropriate synonym is relatively high when replacement is carried out without consideration to word-class or context. This is due to the ambiguity of many words. However, this method was sufficient for generating training data.

5.3.3 Example types

In order to verify our hypothesis, we needed to test different types of variations. Hence, the data used for training and evaluation was divided into the following five main types:

Type 1.a: *A* and *B* were generated from *R* with Missplel using the same type of errors, but with different amounts of misspelled words. *A* contained the larger amount of misspelled words.

Type 1.b: *A* and *B* were generated from *R* with Synlex using different amounts of synonyms. *A* contained the larger amount of synonyms.

Type 2: *A* and *B* were generated from *R* by Missplel using different types of errors though the total sum of errors in *A* and *B* was equal.

Type 3: *A* was generated from *R* by substituting a certain amount of words with synonyms fetched from Synlex. *B* was generated from *R* with rearranged paragraphs and sentences.

Type 4: A was generated from R by introducing spelling errors with Misspell. B was generated from R by replacing a certain amount of words with synonyms fetched from Synlex.

Type 5: A was generated from R by introducing spelling errors and/or by replacing words with synonyms fetched from Synlex. B originated from R but was expressed in another tense.

Examples of Type 1 (a and b) and Type 4 were not difficult to generate automatically while examples of the types 2, 3, and 5 required manual editing.

5.3.4 The generated example set

The generated example set was generated in order to acquire large amounts of training and evaluation data. It contained examples of the types 1.a, 1.b, and 4. Documents were sampled from the KTH News Corpus. The samples contained documents containing 200–300 tokens each. 1000 reference documents were used to generate 5850 examples, approximately 1950 examples of each type (1.a, 1.b, and 4).

5.3.5 The assessor example set

The assessor example set was acquired from two assessors. It contained examples of all types, i.e. 1.a, 1.b, 2, 3, 4, and 5. We aimed to find test data that were both difficult to classify and reliable (see Section 4.5 for our meaning of reliable data). This is an inconsistency since the less difficult the documents are to classify the more reliable the test data ought to be. A pilot study was conducted in order to find a balance between the degree of difficulty and the degree of reliability. The pilot study was done by one assessor. The document examples given to the assessor were shown to be too difficult to classify with high reliability. Therefore, the document examples used in the main test that followed were considerably easier to classify.

The main test was carried out with two assessors. The tests were the same with the exception that the examples were randomly ordered prior to giving them to the assessors. The assessors were briefly informed about the context in which the examples would be used but nothing about the different types of modifications in the documents. The assessors were given 25 examples, five of each main type (1, 2, 3, 4, and 5). Each example was comprised of a document triple (R, A, B) . For each example the assessors were asked to read the documents R , A , and B in order to determine which of the documents A or B that they thought was most similar to R . A choice was demanded; however, if the assessors found it difficult to determine which choice to make, they were permitted to add a question mark on the example that indicated that the choice was difficult. The test was limited to 75 minutes, an average of one minute per document. The results from the assessors and the TEXTSIM models are reported in the section that follows.

5.4 Performance

The performance of the two systems was evaluated and compared with each other using the generated example set as well as the assessor example set.

5.4.1 Performance on generated examples

The reported error rates are the mean values of 10-fold cross-validations repeated ten times using the generated example set. Table 5.1 presents the error rates for each type of variation and the overall error rate when evaluating the entire example set.

Example type	Model 1	Model 2	Examples
Type 1.a	14.5%	13.3%	1970
Type 1.b	0.4%	0.4%	1940
Type 4	29.4%	5.6%	1940
All types	14.2%	8.0%	5850

Table 5.1. Error rates for each type of text for Model 1 and Model 2 respectively.

The most significant difference between Model 1 and Model 2 appears in the examples of Type 4. This corresponds well with our hypothesis. Model 1 has shown not to be efficient in separating document pairs when the resemblance and the containment values between (R_{token}, A_{token}) and (R_{token}, B_{token}) are equal. Further, the differences in performance between Model 1 and Model 2 are negligible on the Type 1 examples. Changes that have been initiated in the Type 1 examples are mirrored equally in the token representation, the lemma representation and the tag representation of the documents. Thus the tags and lemmas are superfluous and do not improve the performance of Model 2.

The reason for the substantial differences between 1.a and 1.b is a result of the more accurate data obtained from the method using Synlex compared to the method using Mispel. However, our attention is not drawn to the differences in values between the various types but to the differences between Models 1 and 2.

Finally, a significant difference between Models 1 and 2 was apparent when using the entire set of generated examples.

5.4.2 Performance on the assessor example set

Model 1 and Model 2 were trained on the generated example set and thereafter compared with the assessors perception of similarity.

Table 5.2 shows the choices of A and B for each example made by the assessors as well as the TEXTSIM models. The assessors choices are reported in the target column wherein hyphens indicate disagreements between the assessors. The assessors agreed on 19 of the 25 examples. The greater portion of the examples in which they did not agree was found in the Type 3 examples. This may be a result of the substantial difference between the type of variation in A and the type of variation in B .

As in the evaluation using the generated example set our attention is drawn to the differences between Models 1 and 2, and not the absolute error rates. By using the examples that the assessors agreed upon, the error rate for Model 1 is 6/19 while the error rate on Model 2 is 4/19.

5.5 Discussion

The system using tokens, lemmas, and PoS tags outperformed the system using tokens, both when evaluation was performed on the generated example set and the examples acquired from assessors. The most obvious differences between the systems appeared on documents with the same amount of token overlap but with different types of changes.

When analyzing the results, consideration should be taken to the fact that not every type of the examples in the assessor example set was included in the training examples. Preferably, all example types should be properly represented in both the training set as well as the test set. Furthermore, the evaluation was performed on a small amount of assessor examples. A more reliable method, in that respect, would have been to acquire a substantially larger amount of assessor examples in order to facilitate the performance of a stratified k -fold cross-validation on the assessor examples without the generated example set.

Example type	Target	Model 1	Model 2
Type 1	A	A	A
	-	-	-
	A	A	A
	A	B	A
	A	A	A
Type 2	B	B	B
	B	B	B
	B	B	B
	B	B	B
	B	A	A
Type 3	-	-	-
	-	-	-
	B	B	B
	-	-	-
	-	-	-
Type 4	B	B	B
	B	A	B
	B	B	B
	B	B	B
	B	A	B
Type 5	A	A	B
	A	B	B
	-	-	-
	A	B	B
	A	A	A

Table 5.2. The generated output of Model 1 and Model 2. The target column indicates agreement and disagreement between the assessors. The examples which the assessors disagreed on are marked with a hyphen.

Chapter 6

Architecture and design

This chapter describes the overall structure of the STATSERV components and its interface to client applications. The architecture and the design of the server demonstrates the modularity and the competence of the server and hence this chapter is a considerable part of this thesis. For implementation details, see the source code of the server prototype and its adherent API documentation. Technical concepts are described as they are introduced, likewise the basis for the decisions regarding the architecture and the design.

6.1 System overview

STATSERV is a servlet-based web application written in Java. It uses an external RDBMS (Relational Database Management System) for persistent storage of document data and statistics. The server is built as a *framework* and can be extended with arbitrary *modules*. Each module supplies the client with statistics of a certain type. For linguistic analysis of the documents sent to server, GTA, Granska Text Analyzer [Domeij et al. 99], is used. The detection of similar documents is handled by TEXTSIM which is integrated into the server.

6.2 Behavioral view

Figure 6.1 is a sequence diagram describing the basic flow of handling a request sent to STATSERV. A client application sends a request to the servlet container. The request is an XML document [XML 1.0] sent via an HTTP POST message [RFC2616]. STATSERV parses the request which contains the document to be processed and specifications for which methods to invoke on which modules.

If the statistics of the incoming document is intended to be added to the reference statistics, the document needs to be classified as new. To be able to classify the incoming document, the document text is sent to and analyzed by the GTA server. Document data from the document set which is specified by the client request, are read from the database (persistent storage). The incoming document is compared with the documents from the database. This comparison determines whether the document should be classified as new or not. If the document is not similar to, nor contained in, another document in the document set, the document is classified as new.

For each module specified in the request the following three methods may be invoked:

Get local statistics analyzes the text and returns statistics for the text. Generally this involves a text analyzer such as Granska.

Add statistics adds local statistics to persistent storage. This method can only be invoked if the document to add has been classified as new. Otherwise an error message will be returned to the client.

Get reference statistics fetches reference statistics from persistent storage. Only statistics relevant to the received text will be returned.

After the modules, requested by the client, have been invoked, `STATSERV` generates an XML document based on the objects returned from the modules. The XML document is finally sent back to the client application.

6.3 Layered architecture

The server is divided into three logical layers. We aim to reach high cohesion within the layers, i.e. each layer should be dedicated to perform a certain type of logic such as persistence logic or application logic, etc. To minimize complexity, only adjacent layers may invoke each other. Further, the layers are hierarchically structured to allow only downward invocations, see Figure 6.2. The three layers are here described in hierarchical order, starting with the top layer:

Interaction layer handles communication and interaction with client applications. This layer controls the flow of the application and, based on the client requests, determines which objects and methods in the lower layer to invoke. It is also responsible for returning a response message to the client.

Application layer provides domain objects and functionality for handling documents e.g. producing sketches and analyzing the structure of the documents. The text identification mechanism is integrated into this layer likewise the modules, i.e. the server extensions.

Persistence layer handles the storage of document sketches and statistics. It is responsible for the *CRUD* operations (Create, Read, Update, and Delete) of the application entities.

6.4 Interaction layer

The interaction layer handles all interaction with client applications. The main classes of the interaction layer are two servlets: `FrontController` and `XMLGenerator`.

The `FrontController` serves as a centralized access point for HTTP client requests. It facilitates concurrent request handling by being a Java servlet.

The application layer is invoked from the `FrontController`, and subsequently, the request is delegated to the `XMLGenerator` servlet which generates the XML document and sends it back to the client.

6.4.1 The Java servlet technology

The servlet technology provides a general and standardized method for building HTTP applications in Java. It requires a *web server* and a *servlet container*. The servlet container determines the target servlet to invoke, depending on the URL sent from the HTTP client. A servlet can therefore be seen as an extension to the servlet container. Servlets are ordinary Java classes which run in a multithreaded environment. For each request sent to the server,

6.4. INTERACTION LAYER

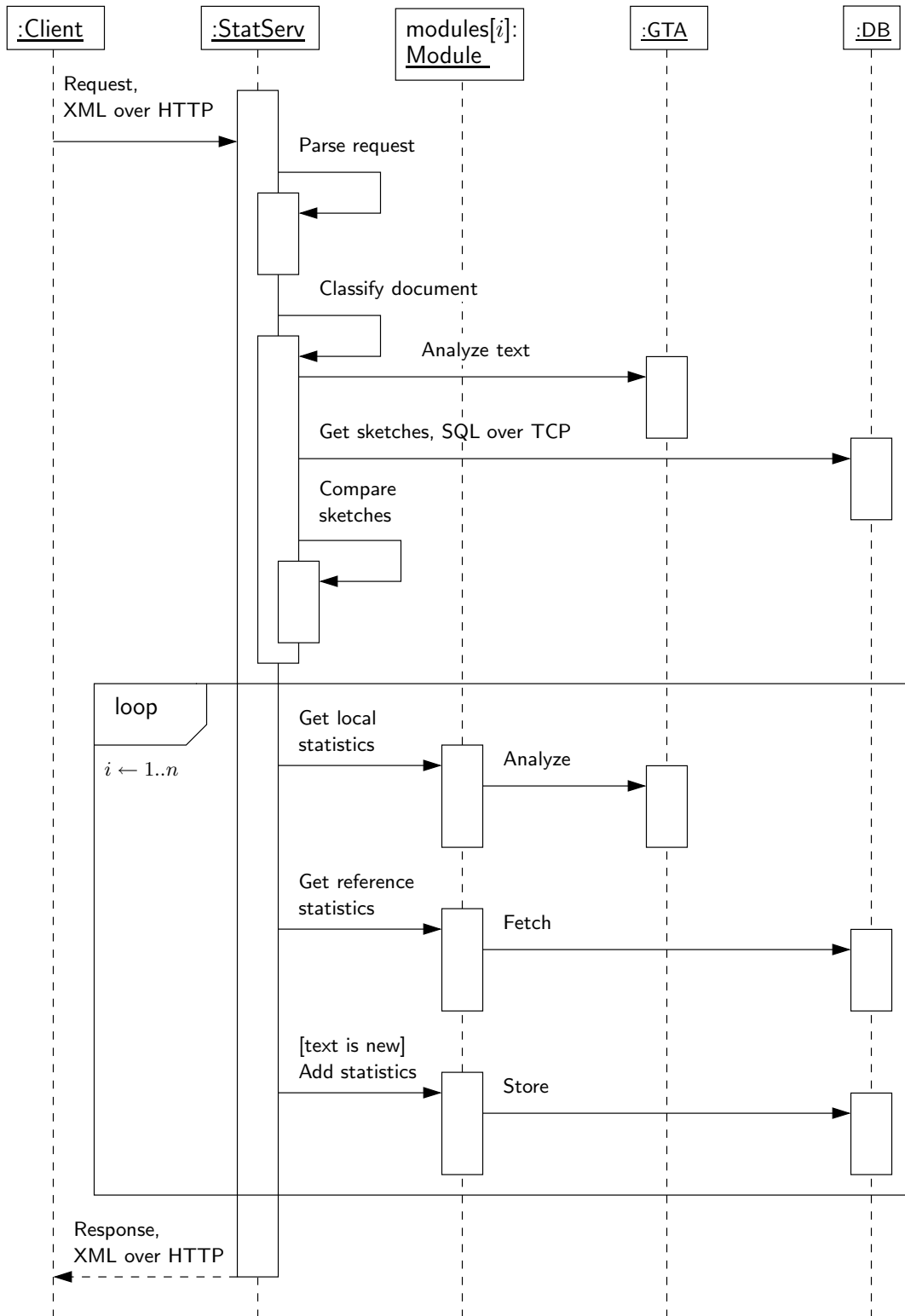


Figure 6.1. Sequence diagram, basic flow

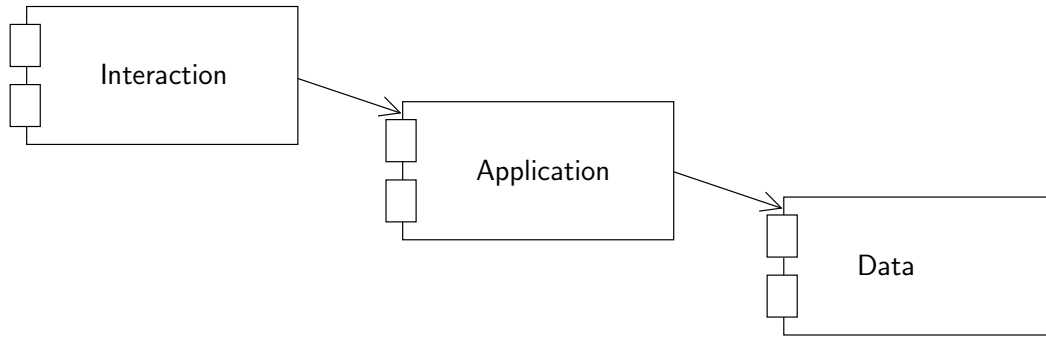


Figure 6.2. Layered architecture

the servlet container invokes the service method of the appropriate servlet in a separate thread. Hence, servlet classes need to be thread safe, an important issue when sharing resources between requests. An example of a typically shared resource in `STATSERV` is the document cache mentioned below.

6.4.2 Choice of communication protocol

`STATSERV` is one of many servers at KTH CSC related to NLP. It was therefore required that the communication protocols were similar to the other servers. The server communicates with clients with XML over HTTP connections. Clients transmit data with an HTTP POST request. Utility classes are included in the `STATSERV` distribution to facilitate Java clients to connect and send requests to the server.

6.4.3 The client request

The request from a client is an XML document which specifies the following:

- the content of the document to be processed
- the selection of modules to be activated
- which methods to invoke on the selected modules.

The request is parsed and interpreted with use of the Java DOM API.

6.4.4 Generating XML

Modules are responsible for returning response objects that represent the statistics requested by the client. The objects are responsible of creating a string of XML representing the state of the objects. All response objects implement the interface `ResponseObject` which assures the ability to produce the XML string.

The `FrontController` servlet retrieves a `StatServResponse` object from the application layer. The `StatServResponse` object is the root node in the response object hierarchy and contains the response objects from the modules and potential error messages. The request is dispatched to the `XMLGenerator` servlet which returns an XML document representing the state of the response object hierarchy back to the client.

6.5 Application layer

The application layer contains the core logic of the server. `TEXTSIM` is integrated into this layer to facilitate the identification of similar texts. Furthermore, the language statistics modules are invoked from this layer and can be seen as extensions to this layer. The main class of the application layer is the `StatServFacade` which provides a high level interface for client classes.

Objects in this layer invoke the data layer for persistent storage of sketches and statistics and invoke other resources for analyzing documents.

6.6 Persistence layer

The persistence layer handles all persistent storage of the application data. The layer consists of a relational database management system, and Java classes dedicated to handle all communication with the database. The classes use the `JDBC` (Java DataBase Connectivity) API.

PostgreSQL was chosen as our relational database management system. Portability between different database systems was not a requirement and therefore it was possible to take full advantage of PostgreSQL-specific functionality such as stored procedures (with `PL/pgSQL`) and non-standard data types (e.g. the array type).

Great importance has been attached to attain an effective persistence layer, since this layer was ought to be a potential bottleneck. The rest of this section describes the general strategies for improving the performance in the persistence layer.

6.6.1 Connection pooling

`STATSERV` makes wide use of the database and it is therefore important to use the database resources effectively. To minimize connection overheads `STATSERV` takes advantage of the application server's ability to pool database connections. A pooled database connection is received from a `datasource`.

`Datasources` are provided by the application server through `JNDI`, Java Naming and Directory Interface, an API for accessing arbitrary resources from Java programs. `JNDI` is included in the Java 2 SDK since version 1.3.

6.6.2 Indexing

Indexes are used in the database on columns which are used as search keys. Without indexes, database queries which involve `WHERE` clauses require full sequential scans on the table. Therefore, indexing has great impact on the performance and entails very few disadvantages such as slower inserts and updates. Searching the statistics is the most critical performance operation and therefore we create an index on every search column.

6.6.3 Stored procedures

PostgreSQL provides a procedural language called `PL/pgSQL` which makes it possible to write persistence logic that is executed on the database server (instead of the Java client code). The use of `PL/pgSQL` simplifies the Java database client code and minimizes the client/server communication, i.e. it reduces the number of queries sent to the database. Both `STATSERV` and the `LEMMA TAG`-module use stored procedures.

6.6.4 Denormalization

As a last resort to improve data layer performance, denormalization is used to avoid multi-table joins on frequent and time consuming queries. Denormalization affects the database model in such a way that it goes from a higher to a lower normal form. E.g. a database in domain-key normal form has no modification anomalies and contains no redundant data, but practical queries on the database could result in all too complex joins for the database to be useful in practical situations. Denormalization can have great impact on performance but comes with serious drawbacks. The most obvious drawback is the redundancy of data and therefore the risk for insert, delete and update anomalies. As an example of denormalization in STATSERV, the database model contains array columns for storing the document data.

6.7 Utility packages

Utility packages collect useful help classes which can be used by the framework and the modules. The utility packages aim to collect common code for code reuse and to simplify the development of future modules.

6.7.1 Caching

A general cache mechanism was built into STATSERV to minimize the number of database requests.

The implemented cache is an LRU cache (Least Recently Used cache). It is implemented with a hashtable and a doubly linked list. The hashtable provides fast methods, $O(1)$, for searching and adding items, while the linked list is used for keeping the cached items ordered by the last time they were used. The most recently used item is the first element in the list. An element is considered in use when it is fetched from the cache. When the cache has reached its maximum size the last item in the list (the least recently used) gets removed from the cache.

The behavior of the cache mechanism is only verified under a non-distributed (i.e. collocated) configuration of the servlet container. When using clustered servlet containers, the state of the cache needs to be replicated among the nodes in the cluster. This is not implemented in the current version of STATSERV.

6.7.2 XML

The widespread use of XML in STATSERV necessitated the development of help classes which simplified the use of the Java DOM and SAX APIs. The interaction layer as well as the application layer including the modules make use of XML, and therefore this functionality was placed in a utility package.

6.8 Design patterns

Design patterns, proven solutions to common design problems, are consistently used throughout the code. We aim to reach maintainable and robust source code by using widespread solution strategies. This section introduces the most frequently used patterns in STATSERV. Most of the patterns are described in [Gamma et al. 95]. This is by no means a full description of the design patterns but an account of why and where the design patterns are used in STATSERV.

6.8.1 Controller

A controller class (the `FrontController` servlet) is used in the interaction layer to serve as a centralized access point for incoming system events. By using a centralized point for access, HTTP-issues and error handling are processed in a uniform way without duplicated code.

6.8.2 Concrete factory

A concrete factory [Larman 04] is a class dedicated to provide initialized instances of a certain class or interface. Dynamic loading of the extensible `STATSERV` modules is an example of where the factory pattern plays an important role. The factory dynamically loads instances of the requested `STATSERV` modules without any hard-coded dependencies with respect to the actual module classes. This makes it possible to add modules to `STATSERV` without recompiling the source code of the server.

6.8.3 Value Object

Value objects, also known as data transfer objects, are objects which transfer data between layers. A value object groups coherent attributes and objects into an atomic object. This simplifies method parameter lists and complex return values.

The response objects described in Section 6.4.4 are examples of value objects. The primary use of the response objects is to exchange data between the application layer and the interaction layer.

6.8.4 Facade

The Facade design pattern originates from [Gamma et al. 95] where the Facade is described as: “a unified interface to a set of interfaces in a subsystem. It defines a higher-level interface that makes the subsystem easier to use.”

The functionality of the application layer is defined by a facade (`StatServFacade`) to hide the complexity of the underlying classes. This resulted in lucid and well-structured code in the interaction layer, which invokes the application layer.

6.8.5 Data Access Object

The DAO (Data Access Object) pattern provides the persistence storage with ordinary Java interfaces. The implementation classes of these interfaces handle database access through JDBC. This results in low coupling between the application layer and the database. Further, high cohesion within the DAO classes is reached by letting the classes handle persistence logic only.

6.8.6 Strategy

The algorithm which classifies the incoming documents needs to be replaceable when evaluating different algorithms. To facilitate replaceability, we let an interface define the family of relevant algorithms. Each algorithm to be evaluated implements the interface and can therefore be replaced. By combining the strategy pattern with the factory pattern, the choice of algorithm can be determined at deploy time.

6.9 Modules

Arbitrary statistical modules can be added to the server to extend the server with new functionality.

New instances of the modules, requested by the client are created for each request. A module is a Java class implementing the `StatServModule` interface which defines four methods:

- `init(String documentContent)`
- `getLocalStats(): LocalStats`
- `getReferenceStats(List<String> documentSets): ReferenceStats`
- `addStatistics(long documentId, String documentSet)`

See Section 6.2 for a description of the methods.

An instance of a module class will only exist during a client request, i.e. for each request, the server will create new instances of the requested modules. The instances are finally destroyed after the request is completed.

Module classes may use the cache utility package (described in Section 6.7.1) for caching data between requests. The sharing of data between module instances needs to be thread safe, since requests are handled concurrently in a multithreaded environment.

6.9.1 Extending the server with new modules

The process of adding a new module to the server involves four steps:

1. creating a class which implements the `StatServModule` interface.
2. adding the database tables required for persistent storage of the reference statistics specific for the new module.
3. creating the module specific response objects returned by the module. The response objects must implement the `ResponseObject` interface and are therefore responsible for producing the XML which will be returned to the client.
4. redeploying the `STATSERV` web application with the newly created classes included.

Thereafter, clients are able to invoke the new module by specifying the class name of the module in the request.

6.9.2 A module for lemmas and tags

The `LEMMA TAGMODULE` is a `STATSERV` module for counting the frequencies of lemma-tag pairs. It exemplifies how a typical `STATSERV` module can be implemented. It uses the `GTA-server` for text analysis and stores statistics specific for the module in a database. To minimize the number of requests sent to the database it shares reference statistics between multiple instances of the module by using the cache utility package.

Chapter 7

Discussion

7.1 Summary

We have seen how an architecture and a design of a language statistics server can be defined. `STATSERV`, the prototype of the language statistics server, fulfills the requirements listed in Chapter 1. Incoming requests are handled concurrently with the use of Java Servlet technology. The architecture is modular and can be extended with new functionality by implementing the `STATSERV` module interface. Each module is required to provide local statistics as well as reference statistics from different corpora. Various types of statistics can also be added to the reference statistics. Further, `STATSERV` uses a standardized protocol including XML and HTTP for communicating with clients. Caching and database query optimizations were used to increase the performance of the server. To detect similar documents sent to the server, `TEXTSIM`, an efficient system for detecting such documents, was implemented and integrated into `STATSERV`.

`TEXTSIM` was implemented using machine learning algorithms with the ability to learn the concept of similarity between documents. The comparison between documents is based on resemblance and containment. Two configurations of `TEXTSIM` were evaluated: one based on tokens, and one based on tokens, lemmas, and PoS tags. The latter system was more sensitive to certain types of differences at the cost of performance with respect to memory and speed. The most obvious differences between the systems appeared on documents with the same amount of token overlap but with different types of variations. Which of the two systems to use can only be determined by the context.

7.2 Future work

To narrow the scope of this project, certain problems had to be left as subjects for future work. This section outlines a selection of these problems.

7.2.1 The language statistics server

In order to make `STATSERV` scalable in the sense that it could be clustered among several nodes, the caching mechanism needs to be improved. The state of the cache must be replicated among the nodes in the cluster, a functionality that has not been implemented in the current version. A third party library for caching could be taken into consideration.

The architecture was defined to be modular. For the server to be of wider use, additional modules need to be implemented, such as LIX [Björnsson 68], word-tag, etc.

7.2.2 Text similarity

A number of performance improvements can be made in the text identification package. In the current version of TEXTSIM, a new document is compared with all documents in a document set. This might not be necessary if we clustered the documents. Each cluster could hold one typical document that could be compared with a new document. If the similarity is below a given threshold, the remaining documents in the cluster could be ignored. Likewise, we could discontinue the comparison between two documents when a certain quantity of shingles has been compared and the number of shingles in common has not passed a certain level.

Instead of using the perceptron, which can only learn linearly separable functions, a more sophisticated ANN could be used such as a multilayer ANN with backpropagation learning. Another possible alternative to the perceptron is the Support vector machine that is known to produce more general classifiers than the perceptron. These alternative algorithms could be implemented into TEXTSIM by using the same supervised learning model including the examples comprised of document triples.

We have only compared two configurations of TEXTSIM, though other configurations are possible by using and combining tokens, lemmas, and PoS tags in various ways.

It would be interesting to acquire a more extensive and larger set of examples from assessors. In that way, the learning algorithms could be both trained and evaluated on data acquired from assessors, instead of using generated data for training as in our evaluation.

Chapter 8

Acknowledgements

First and foremost I would like to thank my supervisor Ph.Lic. Martin Hassel for his encouragement, advices, and excellent reviewing of this thesis. Many thanks as well to Prof. Viggo Kann for his inspiring inventiveness during this project and also for reviewing the thesis. Thank you Forest Andrist for the rigorous language check. Finally, thanks to my whole family for their sincere support.

References

- [Bigert et al. 03] Bigert, J., L. Ericson and A. Solis, 2003. Missplel and AutoEval: Two generic tools for automatic evaluation. In *Proceedings of the Nordic Conference in Computational Linguistics 2003*. Reykjavik, Iceland.
- [Biggs 02] Biggs, N.L., 2002. *Discrete Mathematics 2nd edition* Clarendon Press, Oxford.
- [Björnsson 68] Björnsson, C.H., 1968. *Läsbarhet*. Almqvist och Wiksell, Stockholm.
- [Broder 93] Broder, A. Z., 1993. Some applications of Rabin’s fingerprinting method In Renato Capocelli, Alfredo De Santis, and Ugo Vaccaro, editors, *Sequences II: Methods in Communications, Security, and Computer Science*, 143–152. Springer-Verlag, 1993.
- [Broder 97] Broder, A. Z., 1997. On the resemblance and containment of documents. In *Compression and Complexity of Sequences (SEQUENCES 97)*, 21–29. IEEE Computer Society, 1998.
- [Bylander 98] Bylander, T., 1998. Worst-Case Analysis of the Perceptron and Exponentiated Update Algorithms. In *Artif. Intell.*, vol 106, 335–352. 1998.
- [Callan 99] Callan, R., 1999. *The Essence of Neural Networks* Prentice Hall, Europe.
- [Church 88] Church, K., 1988. A Stochastic Parts Program and Noun Phrase Parser for Unrestricted Text. In *Proceedings of the Second Conference on Applied Natural Language Processing, ACL*, 136–143. 1988.
- [Domeij et al. 99] Domeij, R., Knutsson, O., Carlberg, J., and Kann, V. 1999. Granska – an efficient hybrid system for Swedish grammar checking. In *Proc. 12th Nordic Conf. on Computational Linguistics*.
- [Gamma et al. 95] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. 1995. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley.
- [Hassel 01] Hassel, M., 2001. Automatic construction of a Swedish news corpus. In *Proceedings of the 13th Nordic Conference on Computational Linguistics – NODALIDA ’01*.
- [Lin & Hovy 97] Lin, C-Y., Hovy, E., 1997. Identify Topics by Position, In *Proceedings of the 5th Conference on Applied Natural Language Processing*.
- [Jurafsky & Martin 00] Jurafsky, D. and Martin, J. H., 2000. *Speech and Language Processing*. Prentice Hall, Upper Saddle River, NJ.
- [Kann & Rosell 05] Kann, V., Rosell, M., 2005. Free Construction of a Free Swedish Dictionary of Synonyms.

- [Larman 04] Larman C., 2004. *Applying UML and Patterns : An Introduction to Object-Oriented Analysis and Design and Iterative Development 3rd Edition* Prentice Hall.
- [Mitchell 97] Mitchell, T., 1997. *Machine Learning*. McGraw Hill.
- [Mitzenmacher & Owen 01] Mitzenmacher M., Owen, S., 2001. Estimating Resemblance of MIDI Documents *Lecture Notes in Computer Science*, Vol. 2153.
- [Nivre 02] Nivre, J., 2002. On Statistical Methods in Natural Language Processing. *Second Conference for the Promotion of Research in IT at New Universities and University Colleges in Sweden*, 684–694. University of Skövde.
- [Oakes 98] Oakes M. P., 1998. *Statistics for Corpus Linguistics*. Edinburgh University Press, Edinburgh.
- [Rabin 81] Rabin M, 0, 1981. Fingerprinting by random polynomials. Center for Research in Computing Technology, Harvard University, Report TR-15-81, 1981.
- [Severinson-Eklundh & Kollberg 96] Severinson-Eklundh, K., Kollberg, P., 1996. A computer tool and framework for analysing on-line revisions. In *Levy, C. M. and Ransdell, S. (eds), The science of writing: Theories, Methods, Individual Differences, and Applications, 163–188, Lawrence Erlbaum Ass.*
- [Shivakumar et al. 95] Shivakumar, N., and Garcia-Molina, H., 1995. SCAM: A Copy Detection Mechanism for Digital Documents, In *Proceedings of the Second Annual Conference on the Theory and Practice of Digital Libraries*, 1995.
- [Witten 00] Witten, I., H., 2000. *Data Mining*. Morgan Kaufmann Publishers.
- [Owen 04] Owen, J., 2004. Rabin Hash Function. *SourceForge*. 13 June 2004. 20 Jan. 2006. <http://sourceforge.net/projects/rabinhash>
- [RFC2616] Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P. and T. Berners-Lee, Hypertext Transfer Protocol, HTTP/1.1, RFC 2616 *W3C*, June 1999. 20 Jan. 2006. <http://www.w3.org/Protocols/rfc2616/rfc2616.html>
- [XML 1.0] Bray, T., Paoli, J., Sperberg-McQueen, C. M., Maler, E., Yergeau, F., Extensible Markup Language (XML) 1.0, Third Edition *W3C* Feb 2004. 20 Jan. 2006. <http://www.w3.org/TR/2004/REC-xml-20040204>

TRITA-CSC-E 2006:092
ISRN-KTH/CSC/E--06/092--SE
ISSN-1653-5715