Georg-August-Universität
Göttingen
Zentrum für Informatik

ISSN 1612-6793
Nummer ZFI-BSC-2009-02

Bachelorarbeit

im Studiengang "Angewandte Informatik"

Semantic Text Mining - linguistische Tools im Preprocessing von Text Mining Methoden

Roman Hausner

am Institut für

Informatik

Bachelor- und Masterarbeiten
des Zentrums für Informatik
an der Georg-August-Universität Göttingen

29. April 2009

Georg-August-Universität Göttingen
Zentrum für Informatik

Lotzestraße 16-18
37083 Göttingen
Germany

Tel. +49 (5 51) 39-1 44 14
Fax +49 (5 51) 39-1 44 15
Email office@informatik.uni-goettingen.de
WWW www.informatik.uni-goettingen.de

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Göttingen, den 29. April 2009

Bachelorarbeit

Semantic Text Mining - linguistische Tools im Preprocessing von Text Mining Methoden

Roman Hausner

29. April 2009

Betreut durch
Prof. Dr. Bernhard Neumair
Dr. Heike Neuroth
Andreas Aschenbrenner

Diese Arbeit dokumentiert die Entwicklung eines Service zur Aufbereitung von Textkorpora für Text Mining Methoden, welcher unter anderem als Grundlage für jegliche Form von Text Mining innerhalb der virtuellen Forschungsumgebung *TextGrid* dienen soll. Über ein ebenfalls in diesem Rahmen entwickeltes Service-Template können dabei beliebige linguistische Tools eingebunden werden, mit dem Ziel die Informationsdichte der gewonnenen Daten zu erhöhen. Für die Verarbeitung der Textkorpora wird unter anderem auf Konzepte aus dem Information Retrieval zurück gegriffen.

In this thesis it is my objective to illustrate the development of a service that can be used for the preprocessing of text corpora prior to the application of Text Mining methods. It is particularly suitable as a basis for various kinds of Text Mining within the virtual research environment *TextGrid*. A service template has also been developed which enables the user to employ different kinds of linguistic tools in order to increase the density of the informational data obtained. Among other concepts, those that originate in Information Retrieval have proved to be particularly helpful and are therefore utilized here.

Inhaltsverzeichnis

1. Einleitung

1.1. Ziel der Arbeit

Im Rahmen dieser Arbeit entsteht ein Dienst zur Aufbereitung von Textkorpora für Text Mining Methoden. Dieser soll unter anderem als Grundlage für jegliche Form von Text Mining innerhalb der virtuellen Forschungsumgebung *TextGrid* dienen.

Es wird zunächst über allen Texten des zu verarbeitenden Korpus ein Volltextindex erstellt. Anschließend werden für jeden Text die Auftrittshäufigkeiten (Frequenzen) aller im Index enthaltenen Terme ermittelt. Um die Informationsdichte der erhaltenen Daten zu erhöhen kann der Index nun über beliebige linguistische Tools reduziert werden. Für diesen Zweck wird ein Service-Template bereit gestellt, über welches linguistische Tools um die nötigen Schnittstellen für solche Reduktionen erweitert werden können. Nach beliebig vielen Reduktionsschritten werden die Daten bei Bedarf normalisiert und anschließend in einem geeigneten Format zurückgegeben.

1.2. Aufbau der Arbeit

Der Hauptteil der Arbeit beginnt mit einer ausführlichen Darstellung der Grundlagen. Hierbei wird zunächst auf das Projekt *TextGrid* eingegangen, welches den konzeptuellen Rahmen dieser Arbeit bildet. Anschließend werden alle im Weiteren verwendeten Techniken und Konzepte vorgestellt. Grundlegende Kenntnisse im Bereich XML und verwandter Techniken werden dabei vorausgesetzt. Das Kapitel Architektur beleuchtet den grundlegenden Aufbau der entstehenden Software. Hier werden die zentralen Konzepte vorgestellt und diskutiert sowie einige verwendete Algorithmen erläutert. Das Kapitel Implementierung wirft einen Blick auf die konkrete Umsetzung der Software. Einzelne Aspekte werden anhand des Programmcodes dargestellt, außerdem finden sich hier Hinweise zur Installation und Nutzung. Für das Verständnis dieses Kapitels sind Kenntnisse in der Programmiersprache Java empfehlenswert. Im Kapitel Evaluation wird unter anderem der Ablauf eines Performance Tests dokumentiert sowie der Aufbau des zugrunde liegenden Test-Systems vorgestellt. Abschließend findet sich ein Fazit mit einem Ausblick auf mögliche sinnvolle Weiterführungen dieser Arbeit.

1.3. Parallele Arbeit „Identification of Relevant Words“

In einer parallel entstehenden Bachelorarbeit beschäftigt sich Ubbo Veentjer mit der Identifizierung und Darstellung der relevanten Wörter von Texten. Unter anderem wird dabei das *TextGridLab* um die Möglichkeit erweitert, die relevanten Wörter eines Textes sowie weitere Informationen über diese Wörter aus dem WordNet, einer lexikalischen Datenbank der Englischen Sprache, anzuzeigen. Die hierzu entwickelte Software greift auch auf im Rahmen dieser Arbeit entstandene Services zurück. So kommt für die Berechnung der relevanten Wörter der *Preprocessing Service* zum Einsatz. Ein auf dem *Reduktions Service* basierender WordNet Service ermöglicht die Abbildung dieser Wörter auf URIs aus dem WordNet, welche anschließend dazu genutzt werden die entsprechenden Informationen über die Wörter abzurufen. Die Abbildung 1.1 stellt den Zusammenhang zwischen den beiden Arbeiten dar.

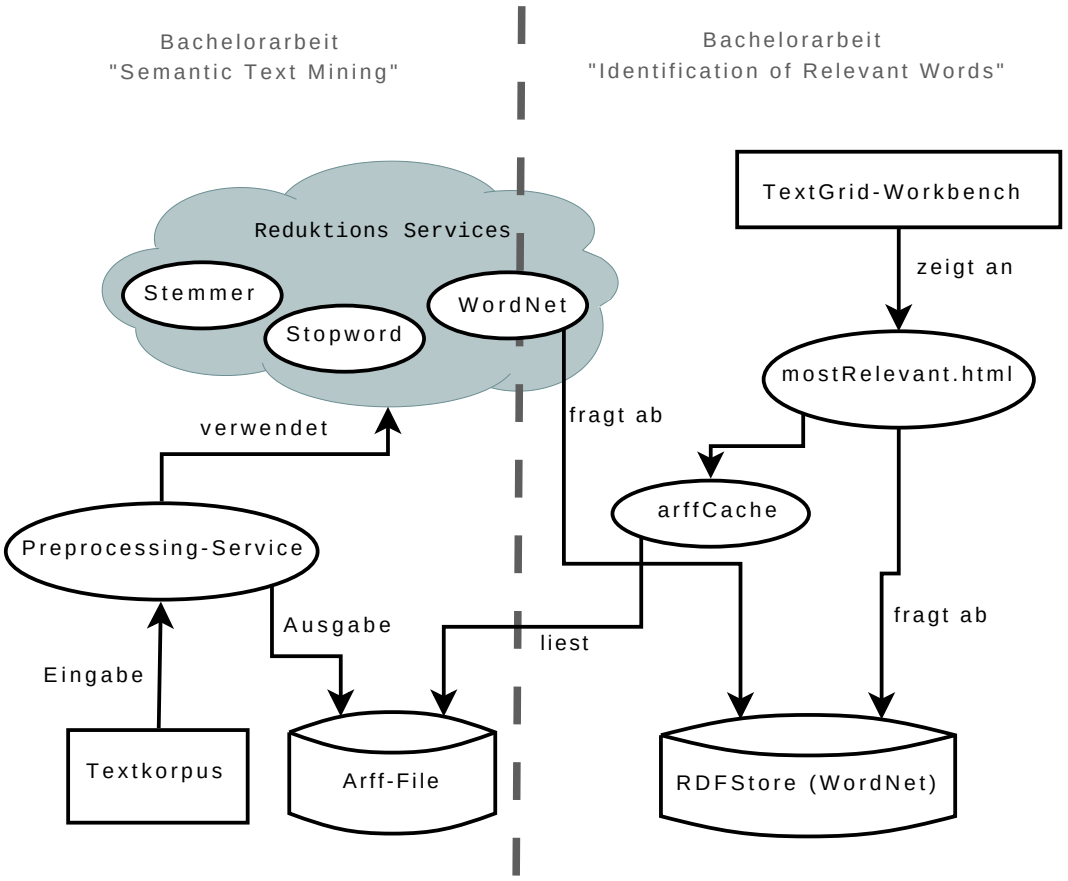

Abbildung 1.1.: Zusammenhang der Arbeiten

2. Grundlagen

2.1. TextGrid

2.1.1. Das Projekt

Das Projekt *TextGrid*[1] ist Teil der D-Grid Initiative[2] zum Aufbau einer nachhaltigen Grid-Infrastruktur in Deutschland. *TextGrid* ist dabei das einzige Projekt mit klarem geisteswissenschaftlichen Bezug und nimmt somit eine Vorreiterrolle in der Entwicklung von E-Humanities ein. Als solche bezeichnet man Geisteswissenschaften, welche in vernetzten Forschungsverbünden zusammenarbeiten. *TextGrid* stellt hierzu eine Workbench zur Verfügung, welche sich vorerst hauptsächlich an Textwissenschaftler richtet. Diese *TextGridLab* genannte Software beinhaltet verschiedene Tools zur Bearbeitung, Analyse, Annotation, Edition und Publikation von Texten und bietet Zugriff auf Ressourcen im Grid. Ein wichtiges Ziel des Projektes ist es, eine engere Zusammenarbeit der Fachwissenschaftler zu fördern. Deren Arbeit soll nicht mehr einzeln oder in abgeschotteten Kleingruppen stattfinden, sondern in offenen, von gegenseitigem Austausch profitierenden Communities. Das *TextGrid* bietet hierzu die Möglichkeit sich untereinander zu vernetzen und andere unkompliziert in die eigene Arbeit einzubinden. Gleichzeitig wird es so möglich über räumliche Distanzen hinweg zusammenzuarbeiten. Das *TextGridLab* erlaubt eine einfache Integration bestehender Textkorpora, Wörterbücher, Lexika, Sekundärliteratur und bereits verfügbarer Textwerkzeuge. Lokal vorhandene Tools und Ressourcen können so global verfügbar gemacht werden. Durch die zugrunde liegende Grid-Infrastruktur entsteht mit *TextGrid* außerdem ein mächtiges Werkzeug zum Umgang mit stetig wachsenden Datenmengen.[3]

2.1.2. Die Architektur

TextGrid basiert auf dem Paradigma der *Service-orientierten Architektur* (*SOA*). Dieses wird in [2] unter der Einschränkung, dass es nicht möglich sei, eine für alle Fälle gültige Definition zu liefern, folgendermaßen beschrieben:

[1] Siehe http://www.textgrid.de (Stand 02/09)
[2] Siehe http://www.d-grid.de (Stand 02/09)
[3] Siehe [1] Seite 64 - 66

> *„Unter einer SOA versteht man eine Systemarchitektur, die vielfältige, verschiedene und eventuell inkompatible Methoden oder Applikationen als wiederverwendbare und offen zugreifbare Dienste repräsentiert und dadurch eine plattform- und sprachunabhängige Nutzung und Wiederverwendung ermöglicht.“*[4]

Die *TextGrid*-Architektur gliedert sich in vier Schichten:

1. Benutzerumgebung
2. Service Layer
3. Middleware
4. Archive

Benutzerumgebung

Die Benutzerumgebung bildet die Schnittstelle zwischen *TextGrid* und seinen Nutzern, sie bietet Zugang zu Ressourcen und Tools. Momentan existiert eine Umsetzung einer solchen Benutzerumgebung auf Basis der *Eclipse Rich Client Platform*[5], das *TextGridLab*. Dieses ist sowohl plattformunabhängig als auch, durch die zugrunde liegende Plug-in-Struktur von *Eclipse*, sehr leicht erweiterbar. Grundsätzlich ist es jedoch denkbar, dass verschiedene Benutzerumgebungen existieren, welche sich in Art und Umfang der Funktionalitäten sowie verwendeter Technik unterscheiden können, verschiedene Nutzergruppen ansprechen und doch auf den selben *TextGrid Service Layer* und die selbe *TextGrid Middleware* aufsetzen.

Service Layer

Der *Service Layer* besteht aus Services, welche in der Sprache *WSDL* (siehe dazu auch 2.2.2 auf Seite 6) beschrieben sind und mindestens über das Protokoll *SOAP* (siehe dazu auch 2.2.1 auf Seite 6) kommunizieren können. Hierdurch entsteht sowohl client- als auch server-seitig eine größtmögliche Freiheit in der Verwendung von Plattformen und Programmiersprachen, da *SOAP*-basierte Services und Clients miteinander kompatibel sind, unabhängig davon, in welcher Technik sie implementiert sind und in welcher Umgebung sie laufen. Durch die Konvention *SOAP*-Nachrichten per *HTTP* zu übertragen werden zudem mögliche Probleme mit Firewall-Konfigurationen oder Netzzugangsbeschränkungen vermieden, da die Kommunikation mit den Services so von überall dort aus stattfinden kann, von wo einfaches Web-Surfen möglich ist.

[4] Siehe [2] S.11
[5] Siehe http://www.eclipse.org/ (Stand 02/09)

Externe Werkzeuge können mit wenig Aufwand in *TextGrid* integriert werden. Falls eine solche noch nicht vorhanden ist wird ihnen dazu eine *SOAP*-Schnittstelle hinzugefügt. Anschließend muss die *TextGrid*-Benutzerumgebung um eine entsprechende Client-Anwendung erweitert werden. Für diesen Zweck stellt *TextGrid* Dokumentations-Material zur Verfügung bzw. bietet Entwickler-Workshops an.

Im Sinne einer *Service-orientierten Architektur* können einzelne Services miteinander kombiniert und somit zu komplexen Funktionen zusammengefasst werden.[6]

Middleware/Archive

Middleware und *Archive* lassen sich nicht so deutlich voneinander abgrenzen wie es bei den anderen Schichten der Fall ist, da die *Archive* ein Teil der *Middleware* sind.[7] *„Die Middleware verwaltet verteilte Ressourcen in TextGrid und virtualisiert diese für homogenen Zugriff.“*[8] Da bei *TextGrid*, zumindest in der momentanen Phase des Projekts, das Hauptaugenmerk auf der Realisierung eines sogenannten *Data Grid* liegt, ist hierunter im Wesentlichen die Integration und Virtualisierung räumlich getrennter Speicher-Ressourcen zu verstehen. Die Schaffung eines *Computational Grid*, bei dem es um die effektive Nutzung von verteilten Rechenkapazitäten geht, gehört hingegen nicht zu den Zielen von *TextGrid*.

2.2. Web Services

Das *W3C (World Wide Web Consortium[9])* definiert Web Services wie folgt:

> *„A Web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards.“*[10]

Im Folgenden werden zunächst die beiden elementaren Web Service Standards *SOAP* und *WSDL* vorgestellt. Im Anschluss daran findet sich ein Abschnitt über das Framework *Axis2*, mit dessen Hilfe die im Rahmen dieser Arbeit entstehenden Web Services auf Basis der genannten Standards umgesetzt werden.

[6] Siehe [3] S.9-10
[7] Siehe [3] S.23
[8] Siehe [3] S.11
[9] Siehe http://www.w3.org/ (Stand 02/09)
[10] Siehe [4] 1.4 What is a Web Service?

2.2.1. SOAP

Der vom *W3C* herausgegebene *SOAP*-Standard beschreibt ein betriebssystem- und programmiersprachenunabhängiges Nachrichtenformat zum Datenaustausch auf Basis von XML. Der Begriff *SOAP* stand dabei ursprünglich als Akronym für *Simple Object Access Protocol*, wird jedoch für die aktuelle Version 1.2 als Eigenname angesehen, da die Umschreibung nicht mehr treffend ist. Für den Transport von *SOAP*-Messages ist kein bestimmtes Protokoll vorgeschrieben, es können also beliebige Standards wie beispielsweise *HTTP*, *SMTP* oder *FTP* genutzt werden. In der Praxis wird jedoch häufig auf *HTTP* zurück gegriffen, da dieses beinah überall ungehindert zur Verfügung steht. Zwischen Sender und Empfänger kann eine Nachricht verschiedene Zwischenstationen, sogenannte *intermediaries* passieren. Dabei ist es auch denkbar, dass eine solche Zwischenstation die Nachricht von einem Transportprotokoll in ein anderes transferiert. Jede *SOAP*-Nachricht besteht aus einem XML-Dokument, dessen Wurzelelement der *SOAP* Envelope bildet. Dieses Element hat zwei mögliche Kinder, den *SOAP* Header und den *SOAP* Body. Alle von *SOAP*-1.2 spezifizierten Elemente und Attribute liegen im Namensraum „http://www.w3.org/2003/05/soap-envelope“[11].

2.2.2. WSDL

WSDL steht für *Web Services Description Language*. Es handelt sich dabei um eine XML-Sprache zur Beschreibung von Web Services, welche vom *W3C* standardisiert wird. Die Version 1.1 existiert seit März 2001 und hat bis heute lediglich den Status einer Anmerkung („Note“). Seit dem 26.06.2007 gibt es die Version 2.0 unter dem beim *W3C* für Standards üblichen Status Empfehlung („Recommendation“). Da für diese Arbeit *WSDL* 1.1 verwendet wird, wird hier auch nur diese Version näher beschrieben.

Die per *WSDL* vorgenommene Beschreibung eines Web Service setzt sich aus einem abstrakten und einem konkreten Teil zusammen, wobei im abstrakten Teil die Funktionalität des Service und im konkreten Teil technische Aspekte wie das verwendete Protokoll beschrieben sind. Ziel einer solchen Beschreibung ist es, alle für den Aufruf des Web Service nötigen Informationen bereit zu stellen, wodurch unter anderem das automatisierte Erstellen von Client-Code ermöglicht wird. Für eine semantische Definition von Services ist *WSDL* nicht vorgesehen.

Das Wurzelelement eines *WSDL*-Dokuments bildet das *definitions* Element. Es enthält die Definitionen aller im Dokument vorkommenden Namensräume und weist per Attribut *targetNamespace* allen definierten Elementen einen Namensraum zu. Zu *definitions* gibt es fünf mögliche Kindelemente:

[11] Siehe [2] S. 45 - 49

types

Das Element *types* ist optional und bildet zusammen mit den Elementen *message* und *portType* den abstrakten Abschnitt des *WSDL*-Files. In *types* werden, falls benötigt, per *XML Schema*[12] (ein ebenfalls vom *W3C* herausgegebener Standard zur Definition von Strukturen in XML-Dokumenten) komplexe Datentypen, welche in den zu versendenden Nachrichten verwendet werden sollen, definiert. Diese Deklarationen müssen nicht an Ort und Stelle zu finden sein, sondern können auch aus anderen Dateien importiert werden. Einfache Datentypen wie beispielsweise *integer* oder *string* müssen nicht extra definiert werden, da diese bereits in der *XML-Schema*-Spezifikation enthalten sind. Sollen ausschließlich einfache Datentypen zum Einsatz kommen, kann *types* somit wegfallen.

message

Mit *message* Elementen werden die auszutauschenden Nachrichten beschrieben. Jedes *message* Element setzt sich aus beliebig vielen *part* genannten Kindelementen zusammen, innerhalb derer auf vorher unter *types* definierte Datentypen referenziert werden kann.

portType

Das Element *portType* definiert die Schnittstelle eines Service über eine Menge von Operationen in Form von *operation* Elementen. Innerhalb dieser Operationen sind verschiedene Kommunikations-Muster, sogenannte *Message Exchange Patterns* (*MEP*) denkbar. Folgende vier Möglichkeiten werden von *WSDL* 1.1 unterstützt:

One-Way Der Client sendet eine Nachricht an den Service.

Notification Der Service sendet eine Nachricht an den Client.

Request-Response Der Client sendet eine Nachricht an den Service, dieser sendet eine Nachricht zurück.

Solicit-Response Der Service sendet eine Nachricht an den Client, dieser sendet eine Nachricht zurück.

Als Kindelemente von *operation* stehen *input*, *output* und *fault* zur Verfügung. Je nachdem welches *Message Exchange Pattern* in der Operation umgesetzt werden soll, werden in entsprechender Reihenfolge ein *input* und/oder ein *output* Element definiert. Darauf können beliebig viele *fault* Elemente folgen. Alle Kinder von *operation* referenzieren auf unter *message* definierte Nachrichten-Typen.

[12] Siehe http://www.w3.org/XML/Schema (Stand 02/09)

binding

binding Elemente beschreiben welche Kommunikations- bzw. Transportprotokolle für den Nachrichtenaustausch verwendet werden und bilden zusammen mit dem Element *service* den konkreten Teil des *WSDL*-Dokuments. Es können mehrere *binding* Elemente definiert werden, da ein Service mehrere Schnittstellen für verschiedene Protokolle zur Verfügung stellen kann. Über das Attribut *type* wird jeweils auf eine unter *portType* spezifizierte Schnittstelle verwiesen. Als Kindelemente von *binding* sind auch Elemente aus anderen Namensräumen erlaubt, um protokollspezifische Angaben zu ermöglichen. Wird als Kommunikationsprotokoll beispielsweise *SOAP* gewählt, so kann etwa über das Element *body* aus dem *SOAP*-Namensraum bestimmt werden, dass eine Nachricht im *SOAP Body* verschickt werden soll.

service

Das Element *service* setzt sich zusammen aus beliebig vielen *port* Elementen. Ein solches *port* Element definiert einen sogenannten Service Endpunkt, indem es einem *binding* eine konkrete Adresse zuordnet unter welcher die entsprechende Service Schnittstelle zu finden ist.[13]

2.2.3. Axis2

Axis2[14] ist ein Framework zur Erstellung und Verfügbarmachung von Web Services auf Basis des *SOAP*-Standards. Die Software steht unter einer Open-Source Lizenz der *Apache Software Foundation*[15] und es existieren sowohl eine Java- als auch eine C-Implementierung. Hier wird lediglich die Java-Variante betrachtet.

Axis2 ist der Nachfolger von *Axis*[16], wurde allerdings komplett neu entwickelt. *Axis*, dessen erste stabile Version bereits seit Oktober 2002 verfügbar ist, hatte im Laufe der Zeit - gemessen an den aktuellen Ansprüchen an Web Services - einige Schwächen entwickelt. Es war langsam, ressourcenhungrig und kompliziert in der Bedienung. *Axis2* stand erstmals im Mai 2006 in der stabilen Version 1.0 bereit, aktuell ist die Version 1.4.1 erhältlich. Eine wesentliche Stärke von *Axis2* ist der Einsatz des zu diesem Zweck entwickelten XML-Objektmodells *AXIOM* (Axis Object Model)[17]. Dieses basiert auf StAX (Streaming API for XML), einem sogenannten Pull-Parser. Dabei wird das XML-Dokument als Datenstrom einmal durchlaufen, ähnlich wie bei SAX (Simple API for XML), jedoch mit dem Unterschied, dass die lesende Anwendung die Kontrolle darüber behält was zu welchem Zeitpunkt eingelesen wird. Das Dokument muss

[13] Siehe [5] S. 44-52
[14] Siehe http://ws.apache.org/axis2/ (Stand 02/09)
[15] Siehe http://www.apache.org (Stand 02/09)
[16] Siehe http://ws.apache.org/axis/ (Stand 02/09)
[17] Siehe [5] Seite 44

folglich nur so weit gelesen werden, bis die Anwendung alle benötigten Daten extrahiert hat. Hieraus ergibt sich ein Geschwindigkeitsvorteil, da unnötiges Parsing vermieden wird. Gleichzeitig entsteht keine übermäßige Speicherbelastung, wie es bei DOM (Document Object Model) der Fall ist. Bei dieser noch von *Axis* verwendeten Technik wird ein Baummodell des gesamten Dokuments im Speicher aufgebaut, was besonders bei großen Dokumenten schnell zu Problemen führt.

Die *Axis2*-Laufzeitumgebung kann entweder als Standalone-Server betrieben werden, oder man deployed die als WAR (Web Archive) erhältliche Variante der Distribution in einem Servlet-Container wie *Tomcat*[18] und betreibt *Axis2* somit als „Container im Container“. In beiden Fällen erfolgt die Administration über ein komfortables Web-Interface. Dank der Features *Hot Deployment* und *Hot Update* können Services deployed oder durch andere Versionen ersetzt werden, ohne dass dazu ein Neustart des Servers erforderlich ist. Für das Deployment werden alle vom Service benötigten Klassen, zusätzliche Bibliotheken und andere Dateien in einem Service-Archiv abgelegt. Zusätzlich muss sich im Archiv ein Ordner *META-INF* und darin eine Konfigurationsdatei namens *services.xml* befinden. Falls ein eigenes WSDL-File genutzt werden soll, muss dieses ebenfalls im Ordner *META-INF* liegen. Das Archiv ist eine JAR-Datei, welche jedoch zur besseren Unterscheidbarkeit von Bibliotheken mit der Endung *.aar* benannt wird.[19]

Des Weiteren sind in der Distribution zwei Kommandozeilentools namens *wsdl2java* und *java2wsdl* enthalten. Diese sind in der Lage automatisiert aus Java-Code ein *WSDL*-File bzw. aus einem *WSDL*-File Java-Code zu erzeugen. Mit *wsdl2java* können die *Service-Skeletons* genannten Code-Gerüste für einen Service generiert werden oder aber ein Client für einen (nicht notwendigerweise *Axis2*-basierten) Web Service in Form der sogenannten *Client-Stubs*. *Axis2* unterstützt neben WSDL 1.1 auch den neuen Standard WSDL 2.0.

2.3. Text Mining / Information Retrieval

Die Begriffe Text Mining und Information Retrieval lassen sich nur schwer exakt voneinander abgrenzen. In [6] findet sich ein Zitat, welches eine einfache Definition für Text Mining liefert und dieses gleichzeitig klar gegenüber dem Data Mining abgrenzt: *„Data mining is about looking for patterns in data. Likewise, text mining is about looking for patterns in text: the process of analyzing text to extract information that is useful for particular pruposes.“*[20] In [7] wird zwar keine eindeutige Definition von Information Retrieval geliefert, wohl aber werden die damit verbundenen Ziele beschrieben: Ein Nutzer hat einen bestimmten Informationsbedarf

[18] Siehe http://tomcat.apache.org/ (Stand 02/09)
[19] Siehe [5] Seite 69
[20] Siehe [6] Seite 331

und formuliert diesen als Anfrage, welche er an ein Information Retrieval-System stellt. Dieses hat nun die Aufgabe für den Nutzer unter den zur Verfügung stehenden Dokumenten jene zu finden, die am besten zu seinem Informationsbedarf passen.[21] Unter einem solchen Dokument ist hierbei ein in digitaler Form vorliegendes Schriftstück zu verstehen, *„welches in einem Datenformat (z.B. ASCII-Text, HTML, RTF) vorliegt, das einen direkten Zugriff auf alle Zeichen des Dokuments ermöglicht.“*[22]

Die beiden Disziplinen Text Mining und Information Retrieval sind demnach eng miteinander verwandt, da sie sich mit dem selben Gegenstand, Text in digitaler Form, auseinandersetzen. Die Zielsetzungen unterscheiden sich jedoch erheblich. Für diese Arbeit finden ursprünglich aus dem Information Retrieval stammende Techniken Verwendung als Grundlage von Text Mining. Sie werden in den folgenden beiden Unterabschnitten vorgestellt. Des Weiteren wird auf die Data Mining-Software *WEKA*, welche auch für Text Mining nutzbar ist, sowie auf das darin enthaltene Dateiformat *ARFF* eingegangen.

2.3.1. Das Vector Space Model

Das *Vector Space Model* ist ein Modell zur Repräsentation von natürlichsprachigen Dokumenten und findet im Information Retrieval Verwendung. Grundsätzlich geht man davon aus, dass ein solches Dokument für die Repräsentation in einem geeigneten Modell entsprechend aufbereitet wird. Dazu wird es von einem Parser durchlaufen, welcher alle Sonderzeichen sowie andere eventuell vorhandene unbrauchbare Zeichenketten entfernt, und lediglich einen Strom von durch Leerzeichen getrennten Worten, den sogenannten Termen ausgibt. Ein Dokument wird als eine Anzahl von Termen betrachtet, wobei:

- D die Menge aller Dokumente d
- T die Menge aller in den Dokumenten $d \in D$ enthaltenen Terme t
- $a_{d,t} \in \mathbb{Z}$ die Anzahl des Vorkommens von Term $t \in T$ in Dokument $d \in D$

bezeichnet.[23]

Das *Vector Space Model* gehört zu den Modellen ohne Terminterdependenzen. Hierbei geht man von einer Orthogonalität bzw. Unabhängigkeit von Termen aus, alle Terme in einem Dokument werden also als komplett zusammenhangslos angesehen. Da dies in natürlicher Sprache nicht der Fall ist, stellt eine solche Betrachtungsweise durchaus eine Einschränkung hinsichtlich der Aussagekraft solcher Modelle dar. Andererseits erhält man so ein unkompliziertes Modell und bis zu einem gewissen Grad lassen sich die Nachteile gegenüber weitaus

[21] Siehe [7] Seite 7 - 10
[22] Siehe [7] S.8
[23] Siehe [7] S.45-46

aufwendigeren Modellen durch den Einsatz von linguistischen Tools wie beispielsweise einem Stemmer wieder ausgleichen.[24]

Im ursprünglich von G. Salton im Jahre 1968 vorgestellten *Vector Space Model* werden die Dokumente $d \in D$ jeweils von einem Vektor $\vec{d}$ in einem Vektorraum $\mathbb{R}^{\#T}$ repräsentiert, wobei jede Dimension dieses Raumes einem der in der Gesamtmenge der Dokumente vorkommenden Terme $t_i \in T$ entspricht. Des Weiteren werden die Komponenten der Dokumentvektoren $\vec{d}$ über Gewichte w_{d,t_i} bestimmt:

$$\vec{d} = \left(w_{d,t_1}, w_{d,t_2}, ..., w_{d,t_{\#T}}\right) \text{ mit } t_i \in T \tag{2.1}$$

Es existieren vielfältige Möglichkeiten solche Gewichte zu berechnen (siehe dazu auch 2.3.2 über *tf-idf*). Als triviale Variante sei an dieser Stelle $w_{d,t} = a_{d,t}$ genannt.

Als Maß für die Ähnlichkeit zweier Dokumente findet beim *Vector Space Model* für gewöhnlich der Kosinus des Winkels der Dokument-Vektoren Verwendung:

$$sim\left(d_i, d_j\right) = \frac{\vec{d_i}\vec{d_j}}{|\vec{d_i}||\vec{d_j}|} = \frac{\sum_{t \in T} w_{d_i,t} w_{d_j,t}}{\sqrt{\sum_{t \in T} w_{d_i,t}^2}\sqrt{\sum_{t \in T} w_{d_j,t}^2}} \tag{2.2}$$

Jedoch sind auch hier viele andere Möglichkeiten denkbar.[25]

2.3.2. tf-idf

tf-idf steht für *term frequency - inverse document frequency* und bezeichnet eine Methode zur Gewichtung von Termen. Sie ist im Wesentlichen durch zwei Überlegungen motiviert:

1. Die Dokumentenlänge sollte berücksichtigt werden. Dies ist bei der trivialen Methode der Gewichtung von Termen durch ihre absolute Häufigkeit nicht der Fall.
2. Die Häufigkeitsverteilung von Termen im gesamten Korpus sollte berücksichtigt werden. Die Idee dabei ist, dass Terme welche in wenigen Dokumenten vorkommen eine höhere Aussagekraft besitzen als jene die in vielen Dokumenten vorkommen.

Insgesamt sollen jene Terme hoch gewichtet werden, welche innerhalb ihres Dokuments häufig auftreten, allerdings nur in wenigen Dokumenten zu finden sind. Es existieren verschiedene Varianten von *tf-idf*, häufig unterscheiden sich die Formeln jedoch lediglich durch die Basis des verwendeten Logarithmus. Die hier vorgestellte Version findet sich in [8].

[24] Siehe [7] S.49
[25] Siehe [7] S.50-52

tf

Um Punkt 1 Rechnung zu tragen verwendet man den sogenannten Termfrequenz-Faktor *tf*. Dieser dient als Maß für die *„Intra-Cluster-Ähnlichkeit“*[26]. Betrachtet wird die Anzahl eines Terms in einem Dokument im Verhältnis zur Anzahl des Terms mit der größten Häufigkeit in diesem Dokument:

$$tf_{i,m} = \frac{freq_{i,m}}{\max_l freq_{l,m}} \tag{2.3}$$

wobei:

- $freq_{i,m}$ die Anzahl des Terms t_i in Dokument d_m
- $\max_l freq_{l,m}$ die Anzahl des häufigsten Terms in Dokument d_m

bezeichnet.[27]

idf

Der Berücksichtigung von Punkt 2 dient die sogenannte inverse Dokument-Frequenz *idf*. Sie fungiert als Maß für die *„Inter-Cluster-Unähnlichkeit“* und ist wie folgt definiert:

$$idf_i = \log \frac{N}{n_i} \tag{2.4}$$

wobei:

- N die Gesamtzahl der Dokumente im Korpus
- n_i die Anzahl der Dokumente in denen t_i auftritt

bezeichnet.[28]

tf-idf

Zusammen ergibt sich die Formel für die Gewichtung von Term t_i in Dokument d_m:

$$w_{i,m} = tf_{i,m} \cdot idf_i = \frac{freq_{i,m}}{\max_l freq_{l,m}} \cdot \log \frac{N}{n_i} \tag{2.5}$$

[26] Siehe [8] S.21
[27] Siehe [8] S.23
[28] Siehe [8] S.23

2.3.3. WEKA

WEKA[29] steht für *Waikato Environment for Knowledge Analysis* und ist der Name einer Java-basierten Software für *Machine Learning* und *Data Mining*, entwickelt von der Universität Waikato. Sie ist open source und steht unter einer *GNU General Public License*. Aktuell (Stand 03/09) ist die Version 3.6 erhältlich. Enthalten sind verschiedene Algorithmen und Visualisierungs-Tools zur Datenanalyse, welche entweder Standalone, unter einer grafischen Oberfläche wie auch von der Kommandozeile, oder aber aus eigenen Java-Klassen heraus genutzt werden können. Die *WEKA*-Software ist weit verbreitet an Universitäten, wo sie bei Forschung und Lehre Verwendung findet. Für die Repräsentation von Ein- und Ausgabedaten kommt bei *WEKA* unter anderem das *ARFF*-Dateiformat zum Einsatz, welches im nächsten Abschnitt vorgestellt wird. Zusätzlich dazu bringt die *WEKA*-Suite das *XRFF*-Format mit. Es handelt sich dabei um eine XML-basierte Variante von *ARFF*, auf die hier jedoch nicht näher eingegangen werden soll.

2.3.4. ARFF

ARFF ist die Abkürzung von *Attribute-Relation File Format* und bezeichnet ein Dateiformat, welches Instanzen über eine gemeinsame Menge von Attributen beschreibt. Diese Instanzen werden als voneinander unabhängig und ungeordnet betrachtet, Beziehungen zwischen ihnen können in *ARFF* nicht dargestellt werden[30]. Ein *ARFF*-File enthält ASCII Text und besteht aus zwei Abschnitten, der *Header Section* und der *Data Section*. Jede Zeile die mit einem % beginnt ist Kommentar. Listing 2.1 auf Seite 14 zeigt ein einfaches Beispiel für ein *ARFF*-File welches Wetterdaten enthält.

Header Section

Die *Header Section* beginnt mit der Angabe eines Namens für das enthaltene Datenset. Dieser wird angezeigt durch das Schlüsselwort *@relation* (welches wie die anderen *ARFF*-Schlüsselwörter *@attribute*, *@data*, *numeric*, *real*, *integer*, *string* und *date* case-insensitive ist), gefolgt von mindestens einem Leerzeichen und dem Namen als String in der selben Zeile. Der Name muss in Anführungszeichen stehen, falls er Leerzeichen enthält. Anschließend folgt die Deklaration der verwendeten Attribute, wobei für jedes Attribut eine Zeile verwendet wird. Eine solche Deklaration beginnt mit dem Schlüsselwort *@attribute*, gefolgt von einem Namen und einem Typ für das Attribut, jeweils durch Leerzeichen getrennt. Dabei stehen vier Datentypen zur Verfügung:

[29] Siehe http://www.cs.waikato.ac.nz/ml/weka/ (Stand 03/09)
[30] Siehe [6] S.49

Listing 2.1: Beispiel *ARFF*-File *Quelle:* [6] S.50

```
% ARFF file for weather data with some numeric features
%
@relation weather

@attribute outlook { sunny, overcast, rainy }
@attribute temperature numeric
@attribute humidity numeric
@attribute windy { true, false }
@attribute play? { yes, no }

@data
%
% 14 instances
%
sunny, 85, 85, false, no
sunny, 80, 90, true, no
overcast, 83, 86, false, yes
rainy, 70, 96, false, yes
rainy, 68, 80, false, yes
rainy, 65, 70, true, no
overcast, 64, 65, true, yes
sunny, 72, 95, false, no
sunny, 69, 70, false, yes
rainy, 75, 80, false, yes
sunny, 75, 70, true, yes
overcast, 72, 90, true, yes
overcast, 81, 75, false, yes
rainy, 71, 91, true, no
```

numeric Ganze oder Reelle Zahlen. Diese können auch genauer als *integer* oder *float* deklariert werden.

string Eine Zeichenkette. Anführungszeichen müssen gesetzt werden, falls Leerzeichen enthalten sind.

date Eine Datumsangabe. Ihr Format kann über einen optionalen String, welcher auf das Schlüsselwort *date* folgt, näher spezifiziert werden.

nominal-specification Hierbei wird als Deklaration eine Liste möglicher Werte in geschweiften Klammern angegeben, die Einträge werden jeweils durch Kommata getrennt und müssen wiederum in Anführungszeichen stehen, falls Leerzeichen enthalten sind.

Data Section

Die *Data Section* wird eingeleitet durch das Schlüsselwort *@data*. Die Daten einer Instanz stehen jeweils in einer Zeile, durch den Beginn einer neuen Zeile wird somit eine neue Instanz

angezeigt. Die Attribut-Werte müssen jeweils in der selben Reihenfolge stehen wie sie in der *Header Section* deklariert wurden und werden durch Kommata getrennt. Ein fehlender Wert kann durch ein Fragezeichen ersetzt werden.[31]

[31] Siehe [9] S.155-159

3. Architektur

3.1. Anforderungen

Wie in der Einleitung bereits erwähnt, soll der entstehende Service in *TextGrid* genutzt werden. Deshalb soll er sich möglichst nahtlos in die bestehende Architektur von *TextGrid*, genauer gesagt in den Service Layer einfügen (siehe hierzu auch 2.1.2 auf Seite 3). Ein weiterer Fokus liegt auf einem generischen Aufbau der Software. Es soll ein möglichst vielseitig einsetzbarer Dienst entstehen, welcher einfach zu erweitern und flexibel an verschiedene Anforderungen anzupassen ist. Da der Dienst in Form von Web Services umgesetzt wird, sind die zu übertragenden Datenmengen so gering wie möglich zu halten.

3.2. Verwendete Technologien

Um eine reibungslose Integration in den *TextGrid* Service Layer zu erreichen, werden für die Services *SOAP* (siehe Abschnitt 2.2.1 auf Seite 6) als Kommunikationsprotokoll sowie *WSDL* (siehe Abschnitt 2.2.2 auf Seite 6) zur Beschreibung der Schnittstellen verwendet. Konkret soll - wie auch in *TextGrid* - *WSDL 1.1* zum Einsatz kommen, da die Version *2.0* sich bis jetzt nicht flächendeckend durchgesetzt hat. Für die Umsetzung des *SOAP*-Standards wird das *Axis2*-Framework (siehe Abschnitt 2.2.3 auf Seite 8) verwendet, welches bei der Erstellung von Web Services erfahrungsgemäß gute Dienste leistet. Die Funktionalität der Web Services wird in der Sprache Java implementiert.

3.3. Die Komponenten

Die entstehende Software gliedert sich in die zwei Komponenten *Preprocessing Service* und *Reduktions Service*, welche in den folgenden beiden Unterabschnitten vorgestellt werden.

3.3.1. Der Preprocessing Service

Der *Preprocessing Service* bildet das Kernstück des Systems und nimmt die Anfragen zur Verarbeitung von Textkorpora entgegen. Die Abbildung 3.1 auf Seite 18 zeigt schematisch die zur Verarbeitung nötigen Schritte zwischen Aufruf und Rückgabe.

Zunächst muss der als Eingabe erhaltene Textkorpus aus dem für die Übertragung genutzten binären Format extrahiert werden. Anschließend wird der Korpus zur Verarbeitung auf ein entsprechendes Modell abgebildet, welches in Abschnitt 3.6 auf Seite 22 näher beschrieben ist. Als nächster Schritt folgt nun die Durchführung einer Volltextindexierung auf dem gesamten Korpus. Bei dieser werden alle im Korpus enthaltenen Wörter (im Folgenden auch Terme genannt) ermittelt. Diese bilden den Index des Korpus. Des Weiteren sollen für jeden Text die Auftrittshäufigkeiten aller im Korpus enthaltenen Terme (Termfrequenzen) berechnet werden. Da für diese beiden Schritte jeweils das Durchlaufen aller Texte im Korpus nötig ist, werden sie im Sinne eines sparsamen Umgangs mit Rechenressourcen zu einem Schritt zusammengefasst. Dadurch muss der Korpus insgesamt lediglich einmal durchlaufen werden, die genaue Vorgehensweise hierbei ist in Abschnitt 3.7 auf Seite 22 dargestellt. Im nun folgenden Schritt kann der Index über ein dazu geeignetes linguistisches Tool reduziert werden, dazu wird auf einen im folgenden Abschnitt vorgestellten *Reduktions Service* zurück gegriffen. Anschließend müssen auch alle Termfrequenzen neu berechnet werden. Da der *Preprocessing Service* den Korpusindex über beliebig viele verschiedene Tools reduzieren kann, können sich diese beiden Schritte dementsprechend häufig wiederholen. Das Zusammenspiel der Services und die Algorithmen für die nötigen Berechnungen sind in Abschnitt 3.8 auf Seite 24 ausführlich dargestellt. Sind alle gewünschten Reduktionen ausgeführt, können die ermittelten Termfrequenz-Werte nun nach der *tf-idf*-Methode normalisiert werden. Anschließend werden die gewonnenen Daten in das Ausgabeformat *ARFF* überführt, in ein ZIP-Archiv verpackt und schließlich zurück gegeben.

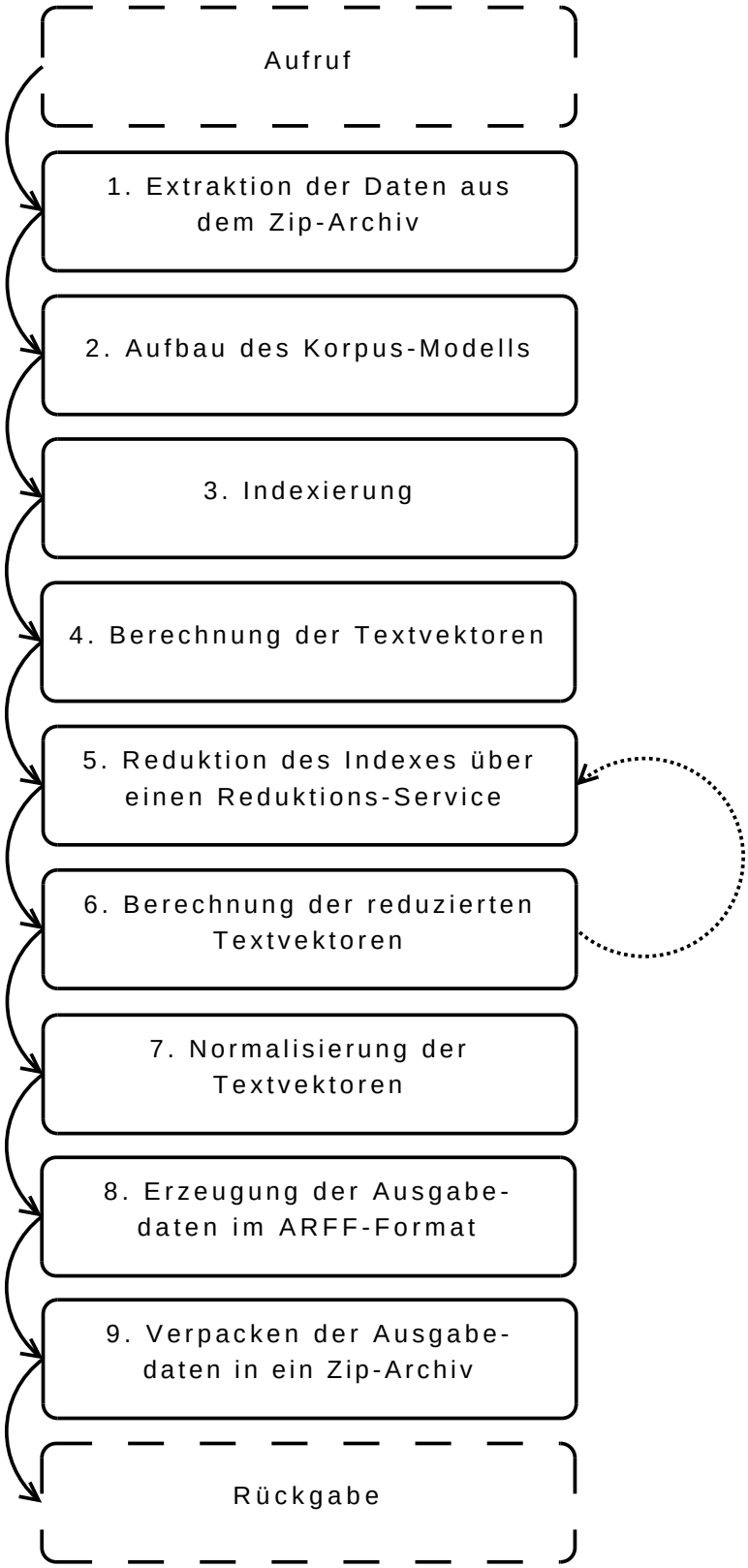

Abbildung 3.1.: Schematischer Workflow des *Preprocessing Service*

3.3.2. Der Reduktions Service

Der *Reduktions Service* stellt eine Art Template dar, mit dem beliebige linguistische Tools, welche zur Reduktion eines Volltextindexes geeignet sind, für die Nutzung durch den *Preprocessing Service* gekapselt werden können. Geeignet heißt hierbei in erster Linie, dass die Tools eine in diesem Zusammenhang sinnvolle Funktionalität bereit stellen sollten. Die Reduktion des Volltextindexes findet entweder über das Eliminieren einzelner Einträge oder aber über das Zusammenfassen mehrerer Einträge zu einem neuen Eintrag statt. Es ergeben sich für die Verwendung eines linguistischen Tools im *Reduktions Service* im Wesentlichen zwei Bedingungen:

1. Das Tool muss in der Lage sein, eine Zeichenkette auf eine andere Zeichenkette (welche auch leer sein darf) abzubilden.
2. Das Tool muss aus einer Java-Klasse heraus aufrufbar sein.

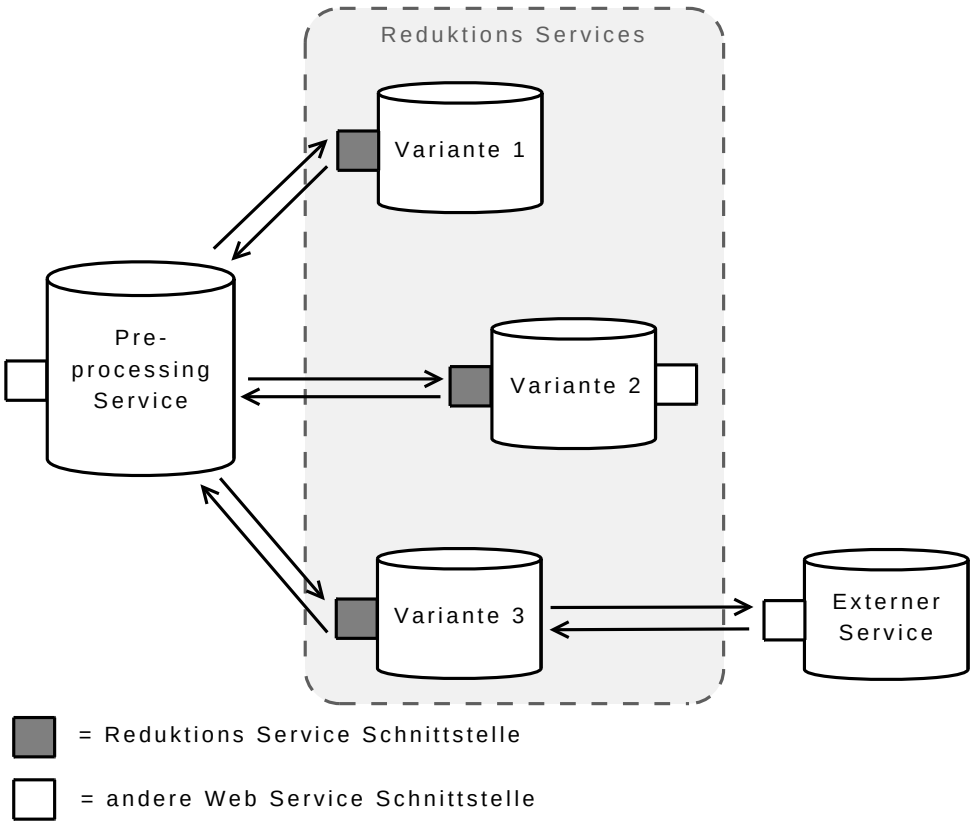

Abbildung 3.2.: Interaktion der Services

Beispiele für geeignete Tools im Sinne von Punkt 1 sind eine Stopwort-Liste und ein Stemmer. Eine Stopwort-Liste bildet ein eingegebenes Wort entweder auf sich selbst oder auf das

leere Wort ab. Ein Stemmer bildet ein eingegebenes Wort auf seine Grundform ab. Die Abbildung 3.2 auf Seite 19 zeigt schematisch die Interaktion zwischen *Preprocessing Service* und verschiedenen Varianten des *Reduktions Service*. Diese Varianten sind dabei allerdings nicht im Sinne von verschiedenen Funktionalitäten zu verstehen, sondern vielmehr als verschiedene Möglichkeiten Tools, die in unterschiedlichen Formen zur Verfügung stehen, zu kapseln.

Variante 1 steht dabei für die Möglichkeit ein Tool als *Reduktions Service* verfügbar zu machen, welches lokal nutzbar vorliegt, jedoch bislang keine Web Service Schnittstelle implementiert. Variante 2 hingegen zeigt den Fall, dass ein Tool bereits als Web Service verfügbar ist. Mittels des *Reduktions Service* wird ein weiterer Service hinzugefügt, welcher vom *Preprocessing Service* angesprochen werden kann. Variante 3 schließlich bringt erhebliche Nachteile bezüglich der Performance mit sich, soll jedoch der Vollständigkeit halber erwähnt werden. Hierbei greift der *Reduktions Service* auf ein linguistisches Tool, welches seine Funktion als Web Service bereit stellt, über eben diese Web Service Schnittstelle zu.

3.4. Die Eingabedaten

Der *Preprocessing Service* erhält seine Eingabedaten in Form eines ZIP-Archivs mit Plain Text Files. Obwohl in *TextGrid* für die Speicherung von Texten das XML-basierte *TEI*-Format[1] verwendet wird, fällt die Wahl des Eingabeformats hier aus verschiedenen Gründen auf Plain Text. Zum Einen werden für die Funktion des Services über den eigentlichen Text hinaus keinerlei weitere Informationen benötigt. Ganz im Gegenteil stellen diese sowohl überflüssigen Overhead in Bezug auf die zu übertragenden Datenmengen als auch eine zusätzliche Hürde beim Extrahieren des Textes dar. Auf der anderen Seite soll im Sinne einer Service-orientierten Architektur ein leichtgewichtiger Dienst entstehen, welcher sich auf den Kernbereich seiner Funktion beschränkt. Die Extraktion von Texten aus einem anderen Format soll also, falls nötig, im Vorfeld mittels eines anderen Tools geschehen.

Des Weiteren erscheint der Einsatz einer Technik zur Datenkompression hier sinnvoll, da gerade Textdateien einen großen Spielraum zur Kompression bieten und die zu übertragenden Datenmengen dadurch erheblich reduziert werden können. Die Wahl fällt auf ZIP, da hierfür eine optimale Unterstützung durch Java gegeben ist.

3.4.1. Zusätzliche Aufruf-Parameter

Neben den in Abschnitt 3.4 beschriebenen Eingabedaten erhält der *Preprocessing Service* beim Aufruf zusätzlich folgende Parameter:

[1] Siehe http://www.tei-c.org/ (Stand 03/09)

ReduceEndpoints Über den Parameter *ReduceEndpoints* erfährt der *Preprocessing Service* welche *Reduktions Services* für die Reduktion des Indexes genutzt werden sollen, und in welcher Reihenfolge dies zu geschehen hat. *ReduceEndpoints* enthält ein Feld von sogenannten Service Endpoints. Es handelt sich dabei um eindeutige Adressen von *SOAP*-basierten Web Services.

Normalizing Der Parameter *Normalizing* enthält ein Feld von Strings mit Schlüsselwörtern für Normalisierungsmethoden. Die Rückgabe des *Preprocessing Service* enthält für jedes hier angegebene gültige Schlüsselwort ein *ARFF*-File mit nach der dem Schlüsselwort entsprechenden Methode normalisierten Werten. Dieser Parameter ist optional. Bleibt er beim Aufruf leer, werden nach *tf-idf* normalisierte Werte zurück gegeben.

TermPattern Über diesen Parameter kann beim Aufruf ein regulärer Ausdruck übergeben werden, welcher diejenigen Zeichenketten beschreibt, die beim Indexieren als Term gelten und in den Index aufgenommen werden sollen. Alle Zeichenketten auf die der Ausdruck nicht passt werden verworfen. Der Parameter ist optional. Bleibt er beim Aufruf leer, wird als Default-Wert der Ausdruck „[a-zA-Z][a-zA-Z][a-zA-Z]+“ genutzt.

3.5. Das Ausgabeformat

Um eine vielseitige Nutzbarkeit des Dienstes sicherzustellen ist es sinnvoll das Ausgabeformat geschickt zu wählen. Dieses sollte primär dazu in der Lage sein, die anfallenden Daten adäquat darzustellen. Darüber hinaus sollen die Daten möglichst einfach wieder zu extrahieren sein und im Hinblick auf eine effiziente Übertragung per Netzwerk sollte bei der Darstellung der Daten möglichst wenig Overhead anfallen. Das in Abschnitt 2.3.4 auf Seite 13 näher beschriebene *ARFF*-Format erfüllt diese Anforderungen. Es ist ausgelegt für die Beschreibung von Instanzen über eine gemeinsame Menge von Attributen, was genau den hier anfallenden Daten entspricht: Texte werden über die Frequenzen einer gemeinsamen Menge von Termen beschrieben. Die *WEKA* Software-Suite (siehe hierzu auch 2.3.3 WEKA auf Seite 13) stellt mit der Java-Klasse *weka.core.converters.ArffLoader* ein Tool zur Verfügung mittels dessen die Daten in einer eigenen Java-Applikation extrahiert werden können. Auch die Weiterverarbeitung der Daten direkt in *WEKA* ist ohne Umwege möglich. Aufgrund des unkomplizierten Aufbaus von *ARFF* können für die Extraktion aber auch leicht eigene Lösungen umgesetzt werden. Zudem ist *ARFF* sparsam im Speicherplatzbedarf weil die Darstellung der Daten keinerlei Redundanzen enthält. Da es sich nicht um ein binäres Format handelt, kann auch hier durch das Verpacken der Ausgabedaten in ein ZIP-Archiv die Datenmenge noch einmal deutlich reduziert werden.

Als Alternative zu *ARFF* käme noch das ebenfalls der *WEKA*-Suite entstammende Format *XRFF* in Frage. Dieses entspricht im Wesentlichen einer XML-basierten Umsetzung von *ARFF*, es bietet jedoch für diese Anwendung gegenüber *ARFF* keinerlei Vorteile. Demgegenüber steht die Tatsache, dass sich durch die XML-Darstellung die Datenmenge erhöht.

3.6. Das Korpus-Modell

Während der Verarbeitung durch den *Preprocessing Service* wird der zu verarbeitende Korpus durch ein Modell repräsentiert. Der Korpus besteht darin aus einem Vektor in dem die Terme aus der Volltextindexierung gespeichert werden, im Folgenden Indexvektor genannt, sowie den Texten. Die Darstellung der Texte basiert auf dem in Abschnitt 2.3.1 auf Seite 10 vorgestellten *Vector Space Model*. Jeder Text wird dabei von einem Vektor repräsentiert, dem Textvektor. Dieser enthält in seinen Komponenten jeweils die Termfrequenzen der korrespondierenden Terme aus dem Indexvektor bezogen auf den jeweiligen Text als absolute Werte. Dementsprechend haben sowohl die Textvektoren aller Texte im Korpus als auch der Indexvektor die selbe Dimension. Zusätzlich zum Textvektor wird für jeden Text der Dateiname unter dem er im ZIP-Archiv der Eingabe abgelegt war gespeichert.

3.7. Indexierung und Berechnung der Textvektoren

Die in 3.3.1 auf Seite 17 vorgestellten Schritte *Volltextindexierung* und *Berechnung der Textvektoren* (siehe hierzu auch Abbildung 3.1 auf Seite 18) werden zu einem Schritt zusammengefasst, damit die Texte des Korpus nur einmal durchlaufen werden müssen. Bei der Indexierung ist es nicht möglich vorauszusagen, welche Dimension der fertige Indexvektor haben wird. Deshalb ist es notwendig für die Repräsentation des Indexvektors eine Datenstruktur zu wählen, welche ihre Größe dynamisch anpassen kann. Das Berechnen von Indexvektor und Textvektoren im selben Schritt hat zur Folge, dass auch die endgültige Größe der Textvektoren nicht vorausgesagt werden kann und folglich auch hier eine entsprechende Datenstruktur zu wählen ist.

In Algorithmus 3.1 auf Seite 23 ist die Berechnung von Indexvektor und Textvektoren als Pseudo-Code aufgeführt. Als Eingaben sind alle Texte des Korpus sowie ein regulärer Ausdruck nötig, welcher jene Zeichenketten beschreibt, die als Term in den Index aufgenommen werden sollen. Zunächst werden Indexvektor und alle Textvektoren dahingehend initialisiert, dass sie keine Komponenten enthalten, ihre Dimension somit 0 ist. Nun werden nacheinander alle Texte durchlaufen. Dabei werden jeweils der Reihe nach alle dem gegebenen regulären Ausdruck entsprechenden Zeichenketten aus einem Text betrachtet. Anhand der Tatsache,

Algorithmus 3.1 Indexierung und gleichzeitige Berechnung der Textvektoren

Require: Texte A_i (wobei $1 \leq i \leq n$)
Require: Regulärer Ausdruck *pattern*
1: Textvektoren $\vec{a_i} = new$ Vektor; (wobei $1 \leq i \leq n$)
2: Indexvektor $\vec{b} = new$ Vektor;
3: **for** $i = 1$ to n **do**
4: String $temp = new$ String;
5: **while** $temp = getNextMatch(A_i, pattern)\ != null$ **do**
6: (wobei $getNextMatch\,(Text, reg.Ausdruck)$ die nächste $reg.Ausdruck$ entsprechende Zeichenkette in $Text$ liefert, falls es eine solche gibt, sonst $null$)
7: **if** $temp == b_j$ (wobei $1 \leq j \leq k$ mit k = momentane Dimension von $\vec{b}$) **then**
8: $a_{ij} = a_{ij} + 1$; (a_{ij} bezeichnet die j-te Komponente von $\vec{a_i}$)
9: **else**
10: $b_p = temp$; (wobei $p = k + 1$)
11: **for** $q = 1$ to n **do**
12: $a_{qp} = 0$;
13: **end for**
14: $a_{ip} = 1$;
15: **end if**
16: **end while**
17: **end for**

ob sich der betrachtete Term bereits im Index befindet oder nicht, wird nun eine Fallunterscheidung vorgenommen. Ist der Term bereits im Index vorhanden, wird die Komponente des zum betrachteten Text gehörenden Textvektors, welche dem Term im Indexvektor entspricht, um eins hochgezählt. Dass diese Komponente im Textvektor bereits existiert ergibt sich aus dem Prozedere im anderen Fall, dass der Term noch nicht im Index vorhanden ist. Dann geschieht Folgendes: Der Term wird dem Indexvektor in einer neuen Dimension hinzugefügt und auch alle Textvektoren werden um eine Dimension erweitert. Im Textvektor des aktuell betrachteten Textes wird diese neue Komponente auf 1 gesetzt, in allen anderen Textvektoren auf 0. Dies erklärt sich so: In allen vor dem aktuellen Text durchlaufenen Texten ist der Term nicht vorgekommen, sonst stünde er bereits im Indexvektor. Seine Häufigkeit in diesen Texten beträgt also 0. Im aktuellen Text ist der Term zum ersten Mal aufgetreten, sonst stünde er ebenfalls bereits im Index. Seine Häufigkeit für diesen Text beträgt zum jetzigen Zeitpunkt also 1. In allen noch zu untersuchenden Texten kann der Term noch auftreten, die Häufigkeit kann dann beim Durchlaufen der Texte entsprechend angepasst werden. Bei diesem Vorgehen haben alle Vektoren nach jedem Schritt weiterhin die selbe Dimension.

3.8. Interaktion von Preprocessing Service und Reduktions Service

Dieser Abschnitt behandelt die Schritte 5. *Reduktion des Indexes über einen Reduktions Service* und 6. *Berechnung der reduzierten Textvektoren* aus Abbildung 3.1. Um den Index in seiner Dimension zu reduzieren ist der Zugriff auf einen *Reduktions Service* nötig. Mit diesem wird dabei per Netzwerk über das *SOAP*-Protokoll kommuniziert. Bei der Planung einer effizienten Interaktion der Services sind also zwei wesentliche Dinge zu bedenken: Zum Einen sind die insgesamt auszutauschenden Datenmengen so gering wie möglich zu halten. Zum Anderen sollten nur so wenig Service-Aufrufe stattfinden wie unbedingt nötig. Hieraus ergibt sich, dass die Terme nicht in einzelnen Anfragen an den *Reduktions Service* geschickt werden können um mit Hilfe der Antworten die Reduktion des Indexvektors sukzessive im *Preprocessing Service* vorzunehmen. Dafür wären so viele Service-Aufrufe nötig, wie sich Terme im Index befinden. Eine andere Möglichkeit wäre es, den Indexvektor und alle Textvektoren in einem Aufruf an den *Reduktions Service* zu übermitteln und diesen die Indexreduktion sowie die Neuberechnung der Textvektoren vornehmen zu lassen. Anschließend würde der *Reduktions Service* in seiner Antwort alle fertig reduzierten Vektoren an den *Preprocessing Service* zurück senden. Bei dieser Variante würden jedoch viele redundante Daten übertragen, die auszutauschende Datenmenge wäre keineswegs optimal gering. Eine dritte Möglichkeit jedoch kommt ebenfalls mit nur einem Aufruf aus und überträgt dabei keinerlei redundante Daten: Es wird lediglich der gesamte Index in einem Aufruf an den *Reduktions Service* übermittelt. Dieser führt nun mit Hilfe des von ihm gekapselten linguistischen Tools die Reduktion des Indexvektors durch und berechnet gleichzeitig eine Abbildungsvorschrift von dem ursprünglichen auf den neuen reduzierten Indexvektor. Beides wird in der Antwort an den *Preprocessing Service* zurück gesandt. Um die Abbildungsvorschrift so kompakt wie möglich zu halten wird sie durch einen Vektor mit ganzzahligen Komponenten, den Mappingvektor, repräsentiert. Dieser hat die selbe Dimension wie der ursprüngliche Indexvektor und ist wie folgt aufgebaut: Jede seiner Komponenten entspricht gemäß ihres Indizes einem Term im ursprünglichen Indexvektor und enthält den Index derjenigen Komponente des reduzierten Indexvektors, auf die der ursprüngliche Term abgebildet wird. Ist ein Term bei der Reduktion aus dem Index entfernt worden, enthält seine korrespondierende Komponente im Mappingvektor den Wert -1. Diese Informationen reichen aus um im *Preprocessing Service* die neuen Textvektoren zu berechnen.

Die beiden folgenden Unterabschnitte gehen näher auf die in den beiden Services nötigen Berechnungen ein.

3.8.1. Berechnungen auf Seiten des Reduktions Service

Der *Reduktions Service* erhält wie bereits erwähnt als Eingabe den Indexvektor. Er hat nun die Aufgabe, in Abhängigkeit von dem ihm zugrunde liegenden linguistischen Tool daraus einen neuen reduzierten Indexvektor sowie den entsprechenden Mappingvektor zu berechnen. Algorithmus 3.2 stellt die dazu nötigen Berechnungen in Pseudo-Code dar.

Algorithmus 3.2 Berechnung des reduzierten Indexvektors und des Mappingvektors

Require: Indexvektor $\vec{a} = a_i$ (wobei $1 \leq i \leq n$)
1: Vektor $\vec{a'} = new$ Vektor;
2: Vektor $\vec{b} = new$ Vektor mit Dimension n;
3: **for** $i = 1$ to n **do**
4: String $s = f(a_i)$; (wobei $f(x)$ Ausgabe der Reduktionsmethode bei Eingabe x)
5: **if** $s == null$ **then**
6: $b_i = -1$;
7: **else**
8: $p = j$; (wobei j = Index von $f(a_i)$ in $\vec{a'}$, falls $f(a_i) \notin \vec{a'}$: $j = -1$)
9: **if** $p == -1$ **then**
10: $a'_{k+1} = f(a_i)$; (wobei k = Dimension von $\vec{a'}$ vor dieser Operation)
11: $p = k + 1$;
12: **end if**
13: $b_i = p$;
14: **end if**
15: **end for**
16: **return** $\vec{a'}, \vec{b}$;

Zunächst werden die Variablen $\vec{a'}$ für den reduzierten Indexvektor und $\vec{b}$ für den Mappingvektor initialisiert. Zu diesem Zeitpunkt ist lediglich vom Mappingvektor $\vec{b}$ die endgültige Dimension bekannt, sie entspricht derjenigen des ursprünglichen Indexvektors, hier durch die Variable $\vec{a}$ repräsentiert. Zumindest für den reduzierten Indexvektor $\vec{a'}$ muss also wiederum eine Datenstruktur mit dynamisch veränderbarer Größe verwendet werden. Eine Schleife durchläuft nun alle Terme in $\vec{a}$ und lässt diese jeweils vom gekapselten linguistischen Tool verarbeiten. Ist die Rückgabe des Tools leer, wird der Term im neuen Index nicht berücksichtigt. Es wird also lediglich an der dem betrachteten Term entsprechenden Stelle im Mappingvektor der Wert -1 eingetragen. Ist die Rückgabe jedoch nicht leer, wird zunächst überprüft ob sie sich bereits im entstehenden neuen Indexvektor $\vec{a'}$ befindet. In diesem Fall wird der Index derjenigen Komponente des Vektors $\vec{a'}$, welche die Rückgabe enthält, als Wert an der dem betrachteten Term entsprechenden Stelle in den Mappingvektor $\vec{b}$ geschrieben. Befindet sich die Rückgabe jedoch noch nicht im Vektor $\vec{a'}$, wird sie ihm als neue Komponente hinzugefügt. Der Index dieser neuen Komponente wird nun in der entsprechenden Stelle im Mappingvektor gespeichert.

3.8.2. Berechnungen auf Seiten des Preprocessing Service

Der *Preprocessing Service* erhält als Rückgabe vom *Reduktions Service* einen reduzierten Indexvektor sowie einen Mappingvektor, mit dessen Hilfe die neuen Textvektoren berechnet werden können. In Algorithmus 3.3 ist diese Berechnung für einen Textvektor dargestellt.

Algorithmus 3.3 Berechnung des reduzierten Textvektors

Require: Mappingvektor $\vec{b} = b_i$ (wobei $1 \leq i \leq n$)
Require: Textvektor $\vec{c} = c_i$ (wobei $1 \leq i \leq n$)
Require: k (wobei k = Dimension des zu berechnenden neuen Textvektors)
1: Vektor $\vec{c'} = \{0\}_i$; (wobei $1 \leq i \leq k$)
2: **for** $i = 1$ to n **do**
3: **if** $b_i \; ! = -1$ **then**
4: $c'_{b_i} = c'_{b_i} + c_i$;
5: **end if**
6: **end for**
7: $\vec{c} = \vec{c'}$;

Neben dem Mappingvektor $\vec{b}$ und dem zu reduzierenden Textvektor $\vec{c}$ benötigt der Algorithmus als Eingabe zusätzlich die Dimension k des reduzierten Indexvektors. Diese entspricht derjenigen des zu berechnenden Textvektors und wird dazu genutzt, eben diesen in Form des Vektors $\vec{c'}$ mit k Einträgen des Wertes 0 zu initialisieren. Der Mappingvektor $\vec{b}$ wird in einer Schleife durchlaufen. Dabei wird für jeden Eintrag zunächst geprüft, ob er verschieden von -1 ist. Ist dies der Fall, wird der Eintrag aus dem ursprünglichen Textvektor $\vec{c}$ mit demselben Index wie der betrachtete Eintrag aus dem Mappingvektor $\vec{b}$ zu demjenigen Eintrag von $\vec{c'}$, dessen Index dem Wert des betrachteten Eintrags aus $\vec{b}$ entspricht, hinzu addiert. Abschließend wird der alte Textvektor $\vec{c}$ durch den neuen $\vec{c'}$ ersetzt.

4. Implementierung

4.1. Vorgehensweise

Beide Services wurden mit *Axis2* (siehe auch Abschnitt 2.2.3 auf Seite 8) nach dem Contract-First Prinzip umgesetzt. Bei dieser Vorgehensweise wird zunächst ein den zu implementierenden Web Service beschreibendes *WSDL*-File erstellt (der genaue Aufbau eines *WSDL*-Files wird in Abschnitt 2.2.2 auf Seite 6 erklärt). Im nächsten Schritt kommt dann das von *Axis2* mitgelieferte Tool *wsdl2java* zum Einsatz, welches anhand des *WSDL*-Files Java Codegerüste erzeugt. In diese wird nun der Code, welcher die Funktionalität des Service implementiert, eingebettet. Diese Herangehensweise bietet gegenüber dem sogenannten Code-First Ansatz, bei welchem zuerst Java-Klassen mit der entsprechenden Funktionalität erstellt und daraus im Anschluss automatisch ein *WSDL*-File generiert wird, einen wesentlichen Vorteil: die Definition der für Ein- und Ausgaben verwendeten Datentypen findet bereits während der Erstellung des *WSDL*-Files statt. Hierzu wird die Sprache *XML Schema* verwendet. Zum Einen werden dadurch Interoperabilitätsprobleme mit Clients, welche mit Hilfe anderer Techniken oder in anderen Programmiersprachen erstellt wurden, vermieden. Für die Abbildung von *XML Schema*-Datentypen auf programmiersprachenspezifische Datentypen gibt es nämlich eindeutige Regeln, was im umgekehrten Fall nicht unbedingt gewährleistet ist. Zum Anderen werden von *Axis2* bei der Generierung der Codegerüste automatisch Klassen erzeugt, welche die Ein- und Ausgaben des Service kapseln. Dadurch wird die Programmierarbeit erheblich erleichtert. Beispielsweise sind auf diesem Weg Konstellationen, welche die gleichzeitige Rückgabe von verschiedenen Datentypen durch einen Service vorsehen, unkompliziert zu realisieren.

4.2. Die WSDL-Files

Wie im vorigen Abschnitt erwähnt, bildet die Erstellung der *WSDL*-Files einen wichtigen Schritt bei der Umsetzung der Services. Das Hauptaugenmerk liegt hierbei auf der Definition der verwendeten Datentypen für Ein- und Ausgabe, welche im Element *types* stattfindet. Listing 4.1 zeigt das *types* Element aus dem *WSDL*-File des *Preprocessing Service*. In ihm werden die Typen *preprocess* und *preprocessResponse* definiert. Der Typ *preprocess*, welcher als Eingabe genutzt wird, besteht aus den Elementen *corpus*, *reduceEndpoints*, *normalizing* und

Listing 4.1: Auszug aus dem *WSDL*-File des *Preprocessing Service*

```
<wsdl:types>
  <xs:schema attributeFormDefault="qualified" elementFormDefault="qualified" targetNamespace="
       http://preprocessing.bac.hausner">
    <xs:element name="preprocess">
      <xs:complexType>
        <xs:sequence>
          <xs:element maxOccurs="1" minOccurs="1" name="corpus" nillable="true" type="
               xs:base64Binary"/>
          <xs:element maxOccurs="unbounded" minOccurs="0" name="reduceEndpoints" nillable="true"
                type="xs:string"/>
          <xs:element maxOccurs="unbounded" minOccurs="1" name="normalizing" nillable="true"
               type="xs:string"/>
          <xs:element maxOccurs="1" minOccurs="1" name="termPattern" nillable="true" type="
               xs:string"/>
        </xs:sequence>
      </xs:complexType>
    </xs:element>
    <xs:element name="preprocessResponse">
      <xs:complexType>
        <xs:sequence>
          <xs:element maxOccurs="1" minOccurs="1" name="arff" nillable="true" type="
               xs:base64Binary"/>
        </xs:sequence>
      </xs:complexType>
    </xs:element>
  </xs:schema>
</wsdl:types>
```

termPattern. Diese entsprechen den in den Abschnitten 3.4 und 3.4.1 vorgestellten Aufruf-Parametern. Für die Rückgabe kommt der Typ *preprocessResponse* zum Einsatz, welcher lediglich ein Element vom Typ *base64Binary* enthält. Dieses entspricht dem in Abschnitt 3.5 beschriebenen vom *Preprocessing Service* zurückgegebenen ZIP-Archiv.

Listing 4.2 zeigt das *types* Element des den *Reduktions Service* beschreibenden *WSDL*-Files. Für die Eingabe wird der Typ *vectorReduce* verwendet, welcher lediglich aus dem Element *vector* besteht. Der zugrunde liegende Typ ist dabei *string* und der Wert *unbounded* des Attributs *maxOccurs* zeigt an, dass es sich um ein Feld handelt. *vectorReduceResponse* wird als Rückgabetyp des Service verwendet. Er enthält die beiden Felder *indexVector* vom Typ *string* und *mappingVector* vom Typ *int*.

4.3. Die Klassen des Preprocessing Service

Die Abbildung 4.1 auf Seite 30 zeigt ein Klassendiagramm des *Preprocessing Service*. Hierbei sind jedoch nur diejenigen Klassen berücksichtigt, in denen die Funktionalität des Service implementiert wurde. Nicht aufgeführt sind jene Klassen, die von *Axis2* automatisch generiert

Listing 4.2: Auszug aus dem *WSDL*-File des *Reduktions Service*

```
<wsdl:types>
  <xs:schema attributeFormDefault="qualified" elementFormDefault="qualified" targetNamespace="
       http://reduce.bac.hausner">
    <xs:element name="vectorReduce">
      <xs:complexType>
        <xs:sequence>
          <xs:element maxOccurs="unbounded" minOccurs="0" name="vector" nillable="true" type="
               xs:string"/>
        </xs:sequence>
      </xs:complexType>
    </xs:element>
    <xs:element name="vectorReduceResponse">
      <xs:complexType>
        <xs:sequence>
          <xs:element maxOccurs="unbounded" minOccurs="0" name="indexVector" nillable="true"
               type="xs:string"/>
          <xs:element maxOccurs="unbounded" minOccurs="0" name="mappingVector" nillable="true"
               type="xs:int"/>
        </xs:sequence>
      </xs:complexType>
    </xs:element>
  </xs:schema>
</wsdl:types>
```

und danach nicht mehr verändert wurden. Dazu zählen unter anderem die Klassen *Preprocess* und *PreprocessResponse*, welche als Eingabe- bzw. Rückgabetyp dienen.

Beim Serviceaufruf wird die Methode *preprocess()* der Klasse *PreprocessingServiceSkeleton* ausgeführt. Eine Instanz der Klasse *ProcessObject* repräsentiert während der Verarbeitung den Textkorpus. Jeder einzelne Text im Korpus wird von einer Instanz der Klasse *TextObject* repräsentiert.

4.4. Die Programmierschnittstelle des Reduktions Service

Die Funktionalität des *Reduktions Service* ist in der Klasse *VectorReduceServiceSkeleton* implementiert. Es handelt sich dabei um eine als *abstract* deklarierte Klasse, welche nicht direkt instantiiert werden kann. Neben der konkreten Methode *vectorReduce()*, welche beim Aufruf des Service ausgeführt wird, verfügt die Klasse über die abstrakte Methode *reduce()*. Diese erhält als Parameter einen String und liefert auch einen String zurück. Sie wird für jeden zu reduzierenden Term aufgerufen. Um ein linguistisches Tool als *Reduktions Service* umzusetzen muss nun eine neue Klasse geschrieben werden, welche die Klasse *VectorReduceServiceSkeleton* erweitert. In dieser Klasse muss die Methode *reduce()* implementiert werden. Hier wird der entsprechende Code eingefügt, welcher den Zugriff auf das linguistische Tool umsetzt.

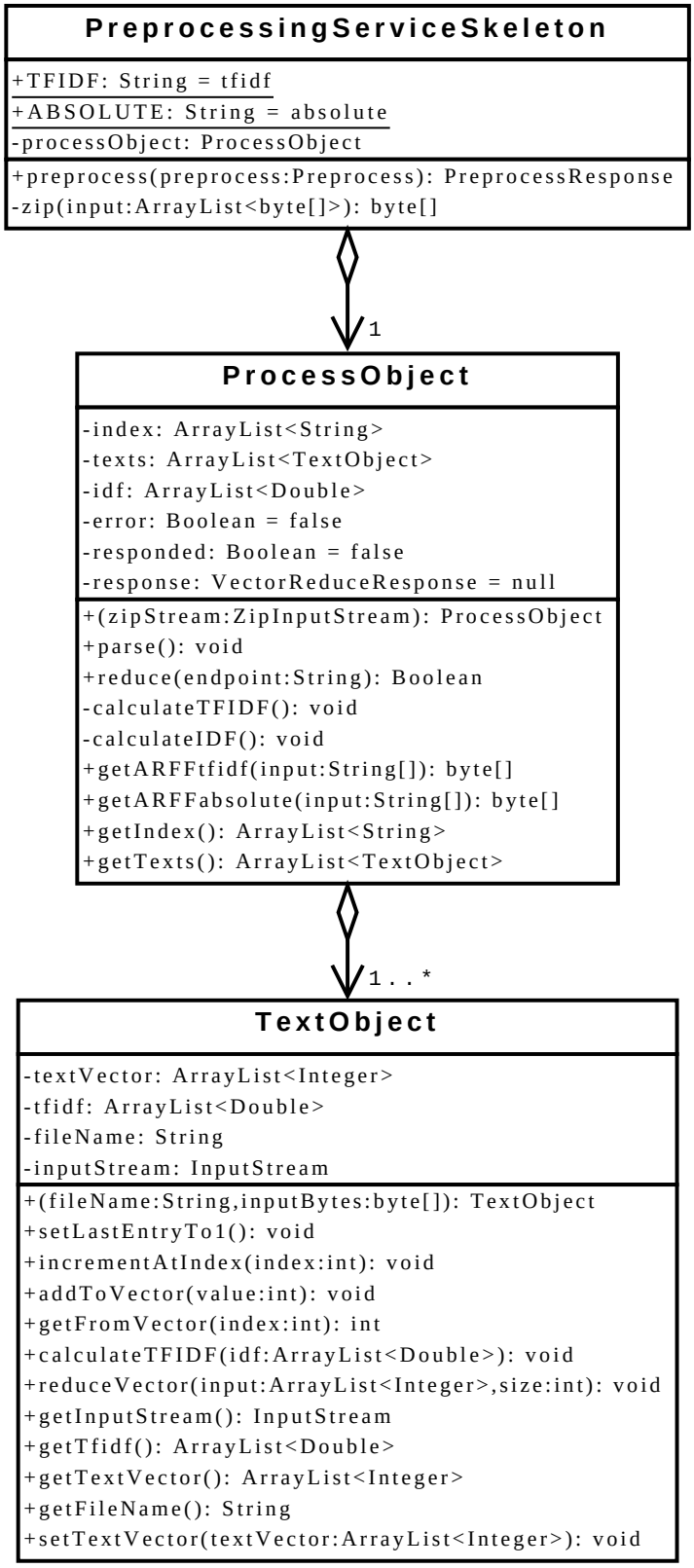

Abbildung 4.1.: Klassendiagramm *Preprocessing Service*

4.5. Übertragen der Daten ins ARFF-Format

Listing 4.3: Die Methode *getARFFabsolute()* der Klasse *ProcessObject*

```
1  public byte[] getARFFabsolute(String[] input){
2      ByteArrayOutputStream stream = new ByteArrayOutputStream();
3      write(stream, "% absolute values\n");
4      write(stream, "% \n");
5      for(int i = 0; i < input.length; i++){
6        write(stream, "% " + input[i] + "\n");
7      }
8      write(stream, "% \n");
9      write(stream, "@RELATION corpus\n");
10     write(stream, "\n");
11     write(stream, "@ATTRIBUTE filename STRING\n");
12     for(int i = 0; i < this.getIndex().size(); i++){
13       write(stream, ("@ATTRIBUTE " + this.getIndex().get(i) + " NUMERIC\n"));
14     }
15     write(stream, "\n");
16     write(stream,"@DATA\n");
17     for(int i = 0; i < this.getTexts().size(); i++){
18       write(stream, this.getTexts().get(i).getFileName());
19       for(int j = 0; j < this.getTexts().get(i).getTextVector().size(); j++){
20         write(stream, ("," + this.getTexts().get(i).getTextVector().get(j)));
21       }
22       write(stream, "\n");
23     }
24     return stream.toByteArray();
25   }
```

Wie in Abschnitt 3.5 beschrieben, werden die Daten im *ARFF*-Format ausgegeben. Die *WEKA* Software-Suite (siehe hierzu auch Abschnitt 2.3.3 auf Seite 13) bringt mit der Klasse *weka.core.converters.ArffSaver* ein Tool zum Speichern von *ARFF*-Dateien mit, welches Dank einer *GNU General Public License* für die Überführung der Daten ins *ARFF*-Format zum Einsatz kommen könnte. Es gibt jedoch mehrere Gründe, die dagegen sprechen. Es handelt sich um eine als Kommandozeilentool ausgelegte Klasse, welche dazu gedacht ist Daten im *ARFF*-Format als Datei auf ein Speichermedium zu schreiben. Im *Preprocessing Service* sollen die Daten jedoch lediglich in das *ARFF*-Format gebracht und anschließend als Byte-Array (*byte[]*) zurück gegeben werden. Des Weiteren müssten die Daten zunächst in ein *WEKA* eigenes Objektmodell überführt werden um von dem Tool als Input akzeptiert zu werden. Dies wäre recht aufwendig, außerdem müssten im Endeffekt mehrere Pakete der *WEKA*-Software in den Service-Code eingebunden werden, was diesen unnötig aufblähen würde. Die wesentlich leichtgewichtigere und auch unkomplizierter zu realisierende Lösung ist es somit, die Konvertierung der Daten in das *ARFF*-Format selbst zu implementieren. Sehr hilfreich ist hierbei die Tatsache, dass die Spezifikation von *ARFF* im Hinblick auf das Schreiben von Daten recht einfach umsetzbar ist. Darüber hinaus bietet diese Lösung den Vorteil, dass zusätzliche Informationen in Form von Kommentaren in die *ARFF*-Ausgabe geschrieben werden

können.

Die Klasse *PreprocessObject* bietet zwei verschiedene Methoden zur Erzeugung von Daten im *ARFF*-Format an. Zum Einen gibt es die Methode *getARFFabsolute()*, welche nicht normalisierte, absolute Werte liefert. Dazu kann sie direkt auf die im Korpusmodell gespeicherten absoluten Termfrequenzen zugreifen. Die andere Methode *getARFFtfidf()* schreibt *tf-idf*-normalisierte Werte in die Ausgabe. Hierzu wird zunächst die Berechnung der *tf-idf* Werte ausgelöst, welche im folgenden Abschnitt 4.6 beschrieben wird. Ansonsten unterscheiden sich die beiden Methoden nur in Details. Listing 4.3 zeigt die Methode *getARFFabsolute()*. Sie erhält als Parameter ein String-Array (*String[]*), dessen Einträge jeweils als eine Kommentarzeile in das entstehende *ARFF*-File geschrieben werden sollen. Enthalten sind hierin Informationen darüber, welcher reguläre Ausdruck für die Volltextindexierung genutzt wurde, sowie welche *Reduktions Services* in welcher Reihenfolge für die Reduktion des Indexes genutzt wurden, und ob diese Reduktionen erfolgreich waren oder ob dabei Fehler auftraten. Die Methode arbeitet auf einem *ByteArrayOutputStream*, welcher direkt vor der Rückgabe über die Methode *toByteArray()* in eine Byte-Array (*byte[]*) umgewandelt wird. Zunächst werden jedoch alle relevanten Informationen zeilenweise als Strings in den *ByteArrayOutputStream* geschrieben. Bezüglich des genauen Aufbaus eines *ARFF*-Files sei hier auf den Abschnitt 2.3.4 auf Seite 13 verwiesen. Die Texte werden im *ARFF*-Format wie folgt über Attribute repräsentiert: Das erste Attribut enthält den Dateinamen unter welchem der Text gespeichert ist, es ist vom Typ *STRING*. Alle weiteren Attribute sind jeweils nach einem Eintrag im Index benannt. Sie sind vom Typ *NUMERIC* und enthalten als Wert für jeden Text jeweils die entsprechende Termfrequenz.

4.6. Berechnung der tf-idf Werte

Listing 4.4: Die Methode *calculateIDF()* der Klasse *ProcessObject*

```
1  private void calculateIDF(){
2      this.idf = new ArrayList<Double>(this.index.size());
3      for(int i = 0; i < this.index.size(); i++){
4        int count = 0;
5        for(int j = 0; j < this.texts.size(); j++){
6          if(this.texts.get(j).getFromVector(i) > 0){
7            count++;
8          }
9        }
10       this.idf.add(Math.log10(((double)this.texts.size())/((double)count)));
11     }
12   }
```

Für die Berechnung der *tf-idf* Werte wird auf die im Abschnitt 2.3.2 auf Seite 11 vorgestellte

Formel zurück gegriffen. Die Berechnung findet in zwei Schritten statt. Listing 4.4 zeigt die Methode *calculateIDF()* der Klasse *ProcessObject*. Sie berechnet zunächst einen Vektor mit den für alle Texte im Korpus identischen *idf* Werten und speichert diesen als auf den Typ *Double* parametrisierte *ArrayList* in der Objektvariablen *idf*. In Zeile 3 von Listing 4.4 beginnt eine äußere *for*-Schleife, welche über alle Terme in der den Indexvektor repräsentierenden Objektvariable *index* iteriert. Die in Zeile 5 beginnende innere *for*-Schleife iteriert jeweils über alle Texte im Korpus und zählt dabei diejenigen, in denen der aktuell betrachtete Term vorkommt. Der Aufruf in Zeile 10 berechnet schließlich den *idf* Wert des Terms und fügt ihn der Liste *idf* hinzu.

Listing 4.5: Die Methode *calculateTFIDF()* der Klasse *TextObject*

```
1  public void calculateTFIDF(ArrayList<Double> idf){
2      this.tfidf = new ArrayList<Double>(this.textVector.size());
3      int tfMax = 0;
4      for(int i = 0; i < this.textVector.size(); i++){
5        if(this.textVector.get(i) > tfMax){
6          tfMax = this.textVector.get(i);
7        }
8      }
9      for(int i = 0; i < this.textVector.size(); i++){
10       this.tfidf.add(((double)this.textVector.get(i) / (double)tfMax) * idf.get(i));
11     }
12   }
```

Der zweite Schritt findet für jeden Text in der Methode *calculateTFIDF()* der Klasse *TextObject* statt, sie ist in Listing 4.5 aufgeführt. Hierbei wird jeweils ein Vektor mit *tf-idf* Werten berechnet und in der Objektvariablen *tfidf* abgelegt. Die Zeilen 4 bis 7 des Listings 4.5 zeigen eine *for*-Schleife, welche zunächst über alle Einträge des Textvektors iteriert und dabei den größten auftretenden Wert speichert. Anschließend läuft eine weitere *for*-Schleife über alle Einträge des Textvektors und berechnet für jeden Eintrag einen *tfidf* Wert. Dazu wird jeweils auf den korrespondierenden Eintrag aus dem in Schritt 1 berechneten und dieser Methode als Argument übergebenen Vektor mit *idf* Werten, sowie den eben berechneten größten Eintrag des Textvektors zurück gegriffen.

4.7. Installation

Für das Bauen der Services wird folgende Software benötigt:

- Java Development Kit Version 6
- Apache Axis2 mindestens Version 1.4
- Apache Ant

Die Quellcodes beider Services liegen jeweils mit entsprechenden Build-Files vor. Hier müssen vor dem Build-Vorgang einige Änderungen vorgenommen werden, welche in den folgenden beiden Unterabschnitten erläutert werden. Des Weiteren müssen die Umgebungsvariablen *JAVA_HOME* und *AXIS2_HOME* korrekt gesetzt sein. Um den Build-Vorgang auszulösen genügt dann ein Aufruf von Apache Ant.

4.7.1. Build-Vorgang Preprocessing Service

Am Anfang des Build-Files befinden sich mehrere *property* Elemente. In das Element mit dem Namen *deploy.serverandport* muss als Wert die Domain bzw. IP-Adresse inklusive einer eventuell nötigen Port-Angabe des Servers eingetragen werden, auf dem der Service deployed werden soll. Ein Ant-Task fügt diesen Wert dann an den entsprechenden Stellen in das *WSDL*-File ein.

4.7.2. Build-Vorgang Reduktions Service

Genau wie im vorigen Abschnitt beschrieben muss auch hier im Build-File das *property* Element *deploy.serverandport* angepasst werden. Darüber hinaus sind jedoch noch weitere Änderungen nötig. Zunächst sollte im Wert des *property* Elements *service.name* ein passender Name für den jeweiligen *Reduktions Service* angegeben werden. Dieser wird von einem Ant-Task an den entsprechenden Stellen des *WSDL*-Files und der Datei *services.xml* eingetragen. Auch das entstehende Service-Archiv wird dementsprechend benannt. Der Hintergrund ist hierbei, dass mehrere Instanzen des *Reduktions Service* mit verschiedenen zugrunde liegenden linguistischen Tools im selben Servlet-Container lauffähig sein sollen. Dazu ist es sowohl für die Verwaltung im Container als auch für die Adressierung der verschiedenen Services notwendig, dass diese unterschiedlich benannt sind. Eventuell vom Service benötigte und als Quellcode vorliegende zusätzliche Klassen müssen im *includes*-Attribut des *javac*-Tasks eingetragen werden, damit sie beim Kompilieren berücksichtigt werden. Falls zusätzliche Bibliotheken benötigt werden, können diese einfach in dem Ordner *lib* abgelegt werden. Weitere Änderungen im Build-File sind in diesem Fall nicht nötig.

4.7.3. Deployment

Nach einem erfolgreichen Build-Vorgang befindet sich im Ordner *build* ein Service-Archiv, welches mittels des von *Axis2* bereitgestellten Web-Interfaces deployed werden kann.

Da der *Preprocessing Service* tendenziell große Mengen Binärdaten sowohl empfangen als auch senden senden muss, ist es sinnvoll hier die Nutzung des von *Axis2* unterstützten

Standards *MTOM* zuzulassen. Bei *MTOM* handelt es sich um eine Methode zur optimierten Übertragung von Binärdaten mit *SOAP*. Die Unterstützung von *MTOM* ist im *Axis2*-Container jedoch standardmäßig ausgeschaltet. Um sie zu aktivieren muss in dem File *WEB-INF/conf/axis2.xml* der Parameter *enableMTOM* auf den Wert *optional* gesetzt werden. Der Wert sollte nicht auf *true* gesetzt werden, da es sonst zu Problemen mit Clients kommen kann, die *MTOM* nicht unterstützen.

5. Evaluation

5.1. Performance Test

Der Korpus

Für den Test wurde ein Korpus aus 78 Büchern in Englischer Sprache zusammengestellt. Bei den Büchern handelt es sich um eine zufällige Auswahl aus dem freien Angebot des *Project Gutenberg*[1]. Die Bücher liegen als TXT-Dateien vor und sind ungepackt 28,1 MB, als komprimiertes ZIP-Archiv noch 10,2 MB groß. Hier zeigt sich, dass durch den Einsatz einer Datenkompressions Technik die zu übertragenden Datenmengen sehr effektiv reduziert werden können.

Das Test-System

Für den Performance Test kamen neben dem *Preprocessing Service* folgende *Reduktions Services* zum Einsatz:

Ein Stoppwort-Service Hierbei wurde eine einfache Funktion in den *Reduktions Service* integriert, welche ein eingegebenes Wort mit einer Liste von 319 Stoppwörtern vergleicht.

Ein Stemmer-Service Die *University of East Anglia* bietet einen regelbasierten Stemmer unter anderem in einer Java-Implementierung zum freien Download an.[2] Dieser wurde als *Reduktions Service* umgesetzt.

Ein WordNet-Service Bei dieser Variante handelt es sich um einen von Ubbo Veentjer im Rahmen seiner Bachelorarbeit (siehe Abschnitt 1.3 auf Seite 2) umgesetzten *Reduktions Service*. Das *WordNet*[3] ist eine lexikalische Datenbank der Englischen Sprache. Alle darin enthaltenen Worte haben einen eindeutigen URI. Dieser *Reduktions Service* bildet ein Wort auf seinen *WordNet*-URI ab, falls es in *WordNet* erfasst ist und es einen solchen URI gibt. Ansonsten wird es auf das leere Wort abgebildet.

[1] Siehe http://www.gutenberg.org/ (Stand 03/09)
[2] Siehe http://www.uea.ac.uk/cmp/research/graphicsvisionspeech/speech/WordStemming (Stand 03/09)
[3] Siehe http://wordnet.princeton.edu/ (Stand 03/09)

Alle vier Services liefen während des Tests auf dem selben Server. Bei diesem handelte es sich um ein virtuelles System, welchem eine Intel Xeon CPU mit einer Taktfrequenz von 3.00 GHz sowie 2 GB RAM zur Verfügung standen. Als Betriebssystem fungierte ein 64 Bit Linux. Der genutzte Servlet Container war *Apache Tomcat* in der Version 6.0, dessen Prozess über einen Java Heapspace von 500 MB verfügte. Im Container lief *Apache Axis2* in der Version 1.4.

Der Ablauf

Beim Aufruf des *Preprocessing Service* wurden die Endpoints der genannten *Reduktions Services* in der selben Reihenfolge übergeben wie oben aufgeführt. Ein regulärer Ausdruck für die Volltextindexierung wurde nicht angegeben, somit kam hier der Default-Wert zum Tragen. Als gewünschte Normalisierungs Varianten wurden *absolute* und *tf-idf* angegeben.

Zunächst musste der Service die eingegebenen Texte aus dem ZIP-Archiv extrahieren. Dies dauerte im Test ca. 1 Sekunde. Hier zeigt sich, dass durch die Verwendung von ZIP für die Eingabedaten in Hinsicht auf die Performance kein Nachteil entsteht. Als nächstes mussten alle Texte durchlaufen werden, wobei gleichzeitig der Volltextindex erstellt und die Textvektoren berechnet wurden. Die Volltextindexierung mit dem regulären Ausdruck „[a-zA-Z][a-zA-Z][a-zA-Z]+“ ergab 62483 Terme. Im Test nahm dieser Schritt ca. 3 Minuten und 21 Sekunden in Anspruch, mehr Zeit als jeder andere Schritt. An dieser Stelle wird deutlich, dass es sinnvoll war die Berechnung von Volltextindex und Textvektoren in einem Schritt vorzunehmen, da das langwierige Durchlaufen aller Texte so nur einmal stattfinden muss. Nichtsdestotrotz bietet dieser Schritt sicherlich einen Ansatzpunkt für zukünftige Optimierungen. Möglicherweise kann das Parsing der Texte noch effizienter gestaltet werden. Die nächsten Schritte bildeten die Reduktionen des Volltextindexes über die *Reduktions Services* sowie jeweils die Neuberechnungen der Textvektoren. Der Stoppwort-Service benötigte für die Reduktion ca. 28 Sekunden und lieferte einen Index mit 62203 Termen. Der Stemmer-Service brauchte lediglich ca. 17 Sekunden um den Index auf 46993 Terme zu reduzieren. Die meiste Zeit für die Reduktion benötigte der WordNet-Service mit ca. 1 Minute und 57 Sekunden. Dies ist darauf zurückzuführen, dass der Service für jeden Term eine Suchoperation in einer Datenbank durchführen musste. Der Index wurde dabei auf 21502 Terme reduziert. Die Neuberechnung der Textvektoren dauerte nach jedem Aufruf eines *Reduktions Service* 2 bis 3 Sekunden. Nun mussten noch jeweils ein *ARFF*-File mit absoluten und eines mit *tf-idf*-normalisierten Werten erstellt und beide anschließend in ein ZIP-Archiv verpackt werden. Die Erstellung der Files dauerte ca. 1 bzw. ca. 2 Sekunden bei der zweiten Variante, da hierbei zunächst die *tf-idf*-Werte berechnet werden mussten. Das abschließende Anlegen eines ZIP-Archivs dauerte weniger als 1 Sekunde.

Die Rückgabe

Das vom *Preprocessing Service* gelieferte ZIP-Archiv ist 1,8 MB groß. Die beiden enthaltenen *ARFF*-Dateien sind nach der Extraktion zusammen 16,8 MB groß. Hier ergibt sich also eine noch deutlich effektivere Reduktion der zu übertragenden Datenmengen als es bei den Eingabedaten der Fall ist.

5.2. Verbesserung der Service Interaktion

Der ursprüngliche Entwurf der Software sah folgendes Szenario für den Datenaustausch zwischen *Preprocessing Service* und *Reduktions Service* vor:

Der *Preprocessing Service* übermittelt den aktuellen Indexvektor $\vec{a}$ an den jeweiligen *Reduktions Service*. Dieser reduziert nun $\vec{a}$ zu $\vec{a'}$ und berechnet gleichzeitig eine Abbildungsmatrix M so dass $M \cdot \vec{a} = \vec{a'}$ mit folgendem Algorithmus:

Algorithmus 5.1 Berechnung des reduzierten Indexvektors und einer Abbildungsmatrix

Require: Indexvektor $\vec{a} = a_i$ (wobei $1 \leq i \leq n$)
1: Vektor $\vec{a'} = new$ Vektor;
2: Matrix $M = \{0\}_{nn}$;
3: **for** $i = 1$ to n **do**
4: String $s = f(a_i)$; (wobei $f(x)$ Ausgabe der Reduktionsmethode bei Eingabe x)
5: **if** $s == null$ **then**
6: do nothing;
7: **else if** $s == a'_j$ (wobei $1 \leq j \leq k$ mit k = momentane Dimension von $\vec{a'}$) **then**
8: $M_{ji} = 1$;
9: **else if** $s != a'_j$ (wobei $1 \leq j \leq k$ mit k = momentane Dimension von $\vec{a'}$) **then**
10: $a'_p = s$; (wobei $p = k + 1$)
11: $M_{pi} = 1$;
12: **end if**
13: **end for**
14: Entferne die unteren $n - k$ Zeilen von M;
15: **return** $\vec{a'}, M$;

Nachdem $\vec{a'}$ und M an den *Preprocessing Service* zurück gesendet wurden, kann jetzt aus der ursprünglichen Textvektoren-Matrix B und M die neue reduzierte Textvektoren-Matrix B' über diese Formel berechnet werden:

$$M \cdot B = B' \tag{5.1}$$

Abgesehen von der Tatsache, dass hierzu mit einer Matrizenmultiplikation eine recht kostenintensive Operation nötig ist, ergab sich bereits einen Schritt vorher ein schwerwiegendes Problem: Geht man davon aus, dass ein Korpus n Texte umfasst, welche wiederum insgesamt m verschiedene Wörter enthalten, so ergibt sich, dass B eine $m \times n$-Matrix ist. Die reduzierte Textvektoren-Matrix B' hat weniger Zeilen bzw. im ungünstigsten Fall genauso viele Zeilen wie B. B' ist also eine $s \times n$-Matrix wobei $s \leq m$. Aus der Formel 5.1 ergibt sich nun, dass es sich bei M um eine $s \times m$-Matrix handelt.

Bei einem aus 78 Büchern bestehenden Testkorpus ergab die Indexierung 62483 Terme. Das bedeutet, m wäre in diesem Fall 62483 und M hätte im ungünstigsten Fall $62483 \cdot 62483 = 3904125289$ Einträge. Bei der Verpackung in eine *SOAP*-Message entsteht außerdem noch zusätzlicher Overhead und es müssen letztendlich sehr große Datenmengen über das Netzwerk verschickt werden. Dieses Konzept zum Datenaustausch ist also völlig ungeeignet und wurde durch die in Abschnitt 3.8 auf Seite 24 vorgestellte, wesentlich effizientere Methode ersetzt.

5.3. Fazit und Ausblick

Mit dem *Preprocessing Service* ist ein generischer Dienst für die Aufbereitung von Textkorpora für Text Mining Methoden entstanden. Für die Index-Reduktion können mittels des *Reduktions Service* beliebige linguistische Tools genutzt werden. Beim Aufruf des *Preprocessing Service* kann unter anderem ein frei wählbarer regulärer Ausdruck für die Volltextindexierung übergeben werden. Darüber hinaus kann jeweils individuell festgelegt werden welche *Reduktions Services* in welcher Reihenfolge für die Index-Reduktion genutzt werden sollen. Dadurch ist der Dienst flexibel und vielseitig einsetzbar. Durch den Einsatz der Standards *SOAP* und *WSDL* gliedern sich die Services homogen in den *TextGrid* Service Layer ein.

Der Performance Test hat gezeigt, dass die Services stabil und relativ schnell laufen. Als großer Vorteil hat sich der Einsatz einer Datenkompressionstechnik für Ein- und Ausgabedaten herausgestellt. Hierdurch konnten die zu übertragenden Datenmengen effektiv reduziert werden ohne nennenswerte Einbußen hinsichtlich der Performance hinnehmen zu müssen. Auch in Bezug auf die Kommunikation zwischen *Preprocessing Service* und *Reduktions Service* gilt: Es wird bei minimaler zu übertragender Datenmengen eine gute Performance erreicht. Hier ergibt sich ein Ansatz zur Optimierung: Momentan werden nach jedem Reduktions Schritt alle Textvektoren neu berechnet. Dies könnte dahingehend verändert werden, dass lediglich die Ergebnisse aller Reduktionen mitgeführt und nach Abarbeitung dieser Schritte die Textvektoren einmal neu berechnet werden. Die Ergebnisse des Performance Tests zeigen jedoch, dass hierdurch nur eine geringe Verbesserung in Punkto Geschwindigkeit zu erwarten wäre. Ein größeres Optimierungs-Potential birgt sicherlich das Parsing der Texte. Der Schritt in welchem die Texte durchlaufen werden um den Index zu erstellen und die Textvektoren zu

berechnen kostete im Test mit Abstand am meisten Zeit. Möglicherweise kann hier durch ein effizienteres Vorgehen ein deutlicher Geschwindigkeits-Gewinn erreicht werden.

A. Literaturverzeichnis

[1] Neuroth, Heike, Martina Kerzel und Wolfgang Gentzsch (Hg.): *Die D-Grid Initiative.* Universitätsverlag Göttingen, 2007. http://resolver.sub.uni-goettingen.de/purl/?webdoc-1533.

[2] Dostal, Wolfgang, Mario Jeckle, Ingo Melzer und Barbara Zengler: *Service-orientierte Architekturen mit Web Services : Konzepte - Standards - Praxis.* Elsevier, Spektrum Akademischer Verlag, 2005.

[3] Aschenbrenner, Andreas, Peter Gietz, Martin Haase, Frank Knoll, Christoph Ludwig, Wolfgang Pempe, Markus Sosto und Thorsten Vitt: *TextGrid Architektur.* http://www.textgrid.de/fileadmin/TextGrid/reports/TextGrid_Report_3_2.pdf.

[4] World Wide Web Consortium: *W3C Web Services Architecture Website (Stand 02/09).* http://www.w3.org/TR/ws-arch/.

[5] Frotscher, Thilo, Marc Teufel und Dapeng Wang: *Java Web Services mit Apache Axis2.* Software & Support Verlag GmbH, 2007.

[6] Witten, Ian H. und Eibe Frank: *Data mining: practical machine learning tools and techniques with Java implementations.* Academic Press, 2000.

[7] Kuropka, Dominik: *Modelle zur Repräsentation natürlichsprachiger Dokumente.* Logos Verlag Berlin, 2004.

[8] Karin Haenelt: *Information Retrieval Modelle. Vektormodell. Kursfolien.*, Oktober 2006. http://kontext.fraunhofer.de/haenelt/kurs/folien/Haenelt_IR_Modelle_Vektor.pdf.

[9] Remco R. Bouckaert and Eibe Frank and Mark Hall and Richard Kirkby and Peter Reutemann and Alex Seewald and David Scuse: *WEKA Manual for Version 3-6-0*, 2008. http://prdownloads.sourceforge.net/weka/WekaManual-3.6.0.pdf?download.