LMU
LUDWIG-
MAXIMILIANS-
UNIVERSITÄT
MÜNCHEN

CENTRUM FÜR INFORMATIONS- UND SPRACHVERARBEITUNG
STUDIENGANG COMPUTERLINGUISTIK

Bachelorarbeit

im Studiengang Computerlinguistik

an der Ludwig-Maximilians-Universität München

Fakultät für Sprach- und Literaturwissenschaften

Department 2

Optimierung der linguistischen Suche beim XML-annotierten Nachlass von Ludwig Wittgenstein

vorgestellt von

Faridis Alberteris Azar

Betreuer: Dr. Maximilian Hadersbeck
Aufgabensteller: Dr. Maximilian Hadersbeck
Bearbeitungszeitraum: 03. April 2017 – 12. Juni 2017

Erklärung

Hiermit versichere ich, dass ich diese Bachelorarbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 12. Juni 2017

.....

Faridis Alberteris Azar

Zusammenfassung

Die vorliegende Bachelorarbeit entsteht im Rahmen des Digital-Humanities-Projekts „Wittgenstein in Co-Text“, das seit nunmehr sieben Jahren vom Centrum für Informations- und Sprachverarbeitung (CIS) in Zusammenarbeit mit dem Wittgenstein Archiv der Universität Bergen (WAB) in Norwegen, sehr erfolgreich die FinderApp „WiTTFind“ für die semantische Suche durch den Nachlass von Ludwig Wittgenstein entwickelt und computerlinguistische Verfahren, zusammen mit einer optimalen Web-Präsentation implementiert. Die Dokumente des Philosophs werden vom WAB als XML-annotierte Editionstexte geliefert.

Das Ziel dieser Arbeit ist die Optimierung der linguistischen Suche im XML-annotierten Nachlass von Ludwig Wittgenstein durch die optimale Ausnutzung der XML-Annotation und die Verbesserung der XML-Edition. Diese Arbeit setzt die Schwerpunkt in der Optimierung des Findens von Personennamen in der FinderApp WiTTFind, was zum aktuellen Zeitpunkt nicht optimal funktioniert. Zu diesem Zweck wird hier der Python3 XML-Parser „ElementTree“ verwendet und einige bereits existierende Perl-Programme angepasst.

Abstract

This Bachelor Thesis is included within the Digital Humanities Project “Wittgenstein in Co-Text,” in cooperation for more than seven years between the Center for Information and Language Processing (CIS) and the Wittgenstein Archives at the University of Bergen in Norway (WAB). The CIS developed the finderApp “WiTTFind” for the semantic search through Ludwig Wittgenstein's unpublished works. WiTTFind combines Computational Linguistics methods with an optimal web presentation. The works of the philosopher are supplied by the WAB as XML-Annotated texts.

The aim of this Bachelor Thesis is the optimization of linguistic search through Wittgenstein's XML-Annotated texts by means of expansion of the XML-Annotation and the improvement of the XML-Edition. The main focus of this work relies on the optimization of proper noun search in WiTTFind, which does not currently operate correctly. The Python3 XML-Parser “ElemnetTree” has been used and certain existing Perl-Programs have been adapted for this purpose.

Resumen

El trabajo que se presenta a continuación se enmarca dentro del proyecto de colaboración de Humanidades Digitales “Wittgenstein in Co-Text“, que se viene realizando desde hace más de siete años entre el Archivo Wittgenstein de la Universidad de Bergen, Noruega (WAB) y el Centro de Procesamiento de Información y Lenguaje (CIS). Este último se encarga de desarrollar con éxito el buscador “WiTTFind“ para la búsqueda semántica en las obras que conforman el legado de Ludwig Wittgenstein y de implementar métodos de Linguística Computacional junto con una presentación web óptima. El WAB proporciona al CIS la obra del filósofo en textos editados en formato XML.

El objetivo del presente trabajo es la optimización de las búsquedas lingüísticas en los archivos anotados en formato XML del legado de Ludwig Wittgenstein mediante el aprovechamiento óptimo de la anotación XML y la mejora de la edición XML. Lo esencial de este trabajo reside fundamentalmente en la optimización de la búsqueda

de nombres propios en el buscador WiTTFind, que en la actualidad no funciona correctamente. Para ello se ha utilizado el analizador sintáctico (parser) XML de Python3 “ElementTree“ y se han adaptado determinados programas en Perl ya existentes.

Danksagung

An dieser Stelle möchte ich mich bei all denjenigen bedanken, die mich während der Anfertigung dieser Bachelorarbeit unterstützt und motiviert haben.

Zuerst gebührt mein Dank Herrn Maximilian Hadersbeck, der meine Arbeit betreut hat. Für seine ansteckende Motivation, sein Vertrauen in mich, seine Geduld und seine konstruktive Kritik möchte ich mich herzlich bedanken.
Ein besonderer Dank gilt Ina Arends, die mir selbstlos und geduldig bei dem Korrekturlesen geholfen und zahlreiche Tipps vorgeschlagen hat.

Ebenfalls bedanke ich mich bei Cristina Fontán Fariña für ihre uneingeschränkte und liebevolle Unterstützung. Ihre Hilfe und Beiträge sind auch Teil dieser Bachelorarbeit.

Abschließend möchte ich mich bei meiner Mutter und Oma bedanken, die mein Streben nach Überwindung geweckt und immer unterstützt haben. Dank ihnen bin ich heute, wer ich bin.

„Die Grenzen meiner Sprache bedeuten die Grenzen meiner Welt“[1]

Ludwig Wittgenstein

1 Im Jahr 2001, als ich an die erste Vorlesung für Germanistik an der Universität Havanna teilgenommen habe, sollte ich diesen Zitat von Ludwig Wittgenstein von der Tafel vorlesen. Obwohl ich damals gar kein Deutsch konnte, wusste ich schon die Tragweite dieser Worte. Nun in München, 16 Jahre danach, treffe ich wieder auf dieses Zitat. Jetzt kann ich ein bisschen mehr Deutsch und kenne ein bisschen mehr den Werk des Philosophs. Die Zeit vergeht, das Leben ändert sich ständig. Aber die Bedeutsamkeit dieser Worte ist unvergänglich.

En el año 2001, cuando asistí a mi primera clase de Filología Alemana en la Universidad de La Habana, me tocó leer en voz alta esta frase de Ludwig Wittgenstein. En aquel momento apenas hablaba alemán, pero ya supe reconocer la trascendencia de esta frase. Actualmente estoy en Múnich y 16 años después me topo de nuevo con esta frase. Ahora sé un poco más de alemán y conozco algo más de la vida del filósofo. El tiempo pasa, la vida cambia, pero la vigencia de esta frase es imperecedera.

Inhaltsverzeichnis

Einleitung…….. 1
1 XML, TEI und XML-annotierte Editionstexte aus dem Nachlass Ludwig Wittgensteins…… 3
1.1 Kurze Einführung in XML…… 3
1.2 Kurze Einführung in TEI-P5…… 7
1.2.1 Element <persName> und konkrete Attribute in BNE …… 7
1.2.2 Element <abbr> und konkrete Attribute in BNE…….. 7
1.2.3 Element <emph> und konkrete Attribute in BNE…….. 8
1.2.4 Element <del> und konkrete Attribute in BNE……. 10
1.2.5 Element <choice> und konkrete Attribute in BNE…….. 10
1.2.6 Element <seg> und konkrete Attribute in BNE……. 10
1.2.7 Element <lb> und konkrete Attribute in BNE……... 11
1.2.8 Element <pb> und konkrete Attribute in BNE……... 12
1.2.9 Element <s> und konkrete Attribute in BNE……. 12
1.2.10 Element <ab> und konkrete Attribute in BNE……... 12
1.3 Hinweise über die Editionstexte des Ludwig Wittgenstein Nachlasses…….. 13
2 POS-Tagging……... 17
2.1 POS-Tagging mit dem TreeTagger…… 17
2.2 Arbeitsweise des TreeTaggers bei NORM.xml-Dateien…… 19
3 Erkennung von Personennamen: Probleme und Lösungen……. 23
3.1 Erkennung von Personennamen……... 23
3.1.1 Tagging der XML-Editionstexte……... 23
3.2 Generalisierende, problematische Fälle beim Tagging…….. 25
3.2.1 Korrekter Tag, unbekanntes Lemma……. 25
3.2.2 Adjektiv…… 25
3.2.3 Abkürzung…….. 25
3.2.4 Tag Nomen, Lemma unbekannt…….. 26
3.2.5 Verb…….. 26
3.2.6 Name, der mit einem Nomen übereinstimmt……. 27
3.2.7 Komposita……... 27
3.3 Lösungsorientierte Methoden…… 27
3.3.1 Erstellung einer Personennamensliste mit XML-Parsing……... 27
3.3.2 Semantische Suche in WiTTFind verbessern…….. 28

3.4 Fälle nach der Anwendung der Methoden…….. 34
3.4.1 Korrekter Tag, unbekanntes Lemma……. 34
3.4.2 Adjektiv…… 35
3.4.3 Abkürzung……... 35
3.4.4 Tag Nomen, Lemma unbekannt…….. 36
3.4.5 Verb…….. 36
3.4.6 Name, der mit einem Nomen übereinstimmt……. 36
3.4.7 Komposita……... 37
3.5 Evaluierung……... 37
4 Transkriptionsfehler vs. Editionsprobleme …….....…….. 39
4.1 Transkriptionsfehler…….…… 39
4.1.1 Platzierung der XML-Elemente `<emph>` und `<lb>`……. 39
4.1.2 Transkriptionsfehler bei den XML-Elementen `<emph>` und `<lb>`……. 40
4.2 Editionsprobleme…….. 41
4.2.1 Platzierung des XML-Elements `<seg type="stripped">`…….. 42
4.2.2 Editionsprobleme beim XML-Element `<seg type="stripped">`……. 42
4.3 Lösung von Transkriptionsfehlern…….. 43
4.4 Lösung bei Editionsproblemen……... 43
4.4.1 Anpassung von TreeTagger mit Regex II……... 45
4.4.2 Verbesserung nach der Implementierung von Regex II…….. 46
4.4.3 Weitere problematische `<seg type="stripped">` Strukturen……. 47
4.5 Evaluierung…… 47
Schlussfolgerung…….. 49
Anhang…….. 51
Abbildungsverzeichnis…… 61
Verzeichnis der Tabellen……. 62
Inhaltsverzeichnis der beigelegten CD……. 63
Literaturverzeichnis…….. 64

Einleitung

Ludwig Wittgenstein (* 26. April 1889 in Wien, † 29. April 1951 in Cambridge) war einer der bedeutendsten Philosophen des 20. Jahrhunderts. Sein Denken umfasste sowohl philosophische als auch logische, mathematische, musikalische, linguistische und politische Themen. Seine wichtigsten Entwürfe, Überlegungen und sein Briefaustausch vor seinem großen Werk *Tractatus Logico-Philosophicus* (1921) stehen uns aus seinem Nachlass zur Verfügung. Der österreichische Philosoph hat in seinem Nachlass viele Referenzen auf andere Philosophen, Musiker, Schriftsteller und Wissenschaftler gemacht. Sie bedeuten eine wertvolle Quelle, sein Werk zu interpretieren.

Für die Forscher, die sich mit dem Nachlass von Wittgenstein beschäftigen, ist es von immenser Bedeutung, einen schnellen Zugriff auf diese Referenzen zu haben. Die Digitalisierung des Nachlasses bedeutet einen großen Schritt für den freien Zugriff auf die Wittgenstein Werke. Der Nachlass enthält 20.000 Seiten zum Teil unveröffentlichter Manuskripte und Typoskripte, die von dem Philosophen zwischen 1906 und 1916 verfasst wurden. Sie wurden vom Wittgenstein Archiv der Universität Bergen (WAB) in Norwegen digitalisiert und in Faksimile im JPG-Format und in Editionstexten im XML-Format in „Wittgenstein Source Bergen Nachlass Edition“ (BNE) zur freien Verfügung gestellt.

Im Rahmen des Digital-Humanities-Projekts „Wittgenstein in Co-Text“ wird seit sieben Jahren in Zusammenarbeit mit dem WAB am Centrum für Informations- und Sprachverarbeitung (CIS) der LMU München die FinderApp „WiTTFind“ entwickelt. Hierbei handelt es sich um eine Suchmaschine, die ausschließlich den Nachlass Wittgensteins durchsucht. Aktuell umfasst dieser innerhalb des Projekts etwa 5.000 Seiten. Die Suchmaschine ist auf die semantische Suche ausgelegt und wird durch computerlinguistische Verfahren optimiert, die speziell für dieses Projekt entwickelt wurden. Eine eigens implementierte Web-Oberfläche bietet zudem eine einfache und passende Bedienbarkeit.

Die Suche nach Personennamen erfolgt nicht optimal bei der FinderApp. Für das semantische Finden arbeitet WiTTFind im Hintergrund mit einem am CIS erstellten Lexikon, mit den getaggten XML-Editionstexten vom WAB und einem probabilistischen Tagger. Das Problem besteht darin, dass die Informationen über die Personennamen, die die XML-Annotationen anbieten, für das Tagging nicht verwendet werden, was allerdings auch nicht von einem Tagger verlangt wird. Dieser arbeitet u.a. mit Wortarten, die aufgrund der fehlerhaften Erkennung von Namen nicht korrekt zugewiesen werden.

Die Motivation dieser Bachelorarbeit wird von drei grundsätzlichen Fragen ausgelöst: Können die Personennamen mit Hilfe der XML-Annotationen besser erkannt werden? Wie können diese Informationen vom Tagger benutzt werden? Kann eine höhere Anzahl von Personennamen trotz XML-Editionsproblemen und -Transkriptionsfehlern extrahiert werden?

Diese Arbeit hat an erster Stelle das Ziel, durch eine optimale Ausnutzung der XML-Annotationen über die Personennamen und durch eine Verbesserung der XML-Editionen die Erkennung von Personennamen im WiTTFind zu optimieren. Für die Lösung dieser Aufgaben werden einige XML-Elemente bzw. -Attribute mit der Hilfe der XML-Programmierschnittstelle von Python „ElementTree“ geparst und modifiziert.

Alle Beispiele, die im Laufe dieser Arbeit gezeigt werden, stammen aus dem Nachlass von Ludwig Wittgenstein.

Zu Beginn dieser Arbeit werden die wichtigsten Aspekte der XML und der Text Encoding Initiative (TEI) vorgestellt. Diese kurzen Einführungen helfen dabei, die Struktur von den XML-Editionstexten des Nachlasses zu verstehen, die im letzten Punkt des Kapitel 1 erklärt wird.

In Kapitel 2 wird der probabilistische Tagger präsentiert. Mit diesem Tagger werden die vorhandenen XML-Editionstexte getaggt. Deswegen wird an dieser Stelle die Arbeitsweise des Taggers genau beschrieben.

Das Kapitel 3 befasst sich direkt mit der Optimierung der Personennamenserkennung. Zu diesem Zweck werden alle Beispiele gesammelt und generalisiert, die das Scheitern des Taggers bei der Erkennung von Personennamen zeigen und es werden lösungsorientierte Methoden vorgeschlagen, um diese Probleme zu beseitigen. Am Ende des Kapitels wird evaluiert, ob diese Methoden das Finden von Personennamen in WiTTFind tatsächlich verbessern können.

Das letzte Kapitel enthält eine Beschreibung der möglichen Transkriptionsfehler sowie der Editionsprobleme, die auch das Finden der Personennamen in den XML-Editionen verhindern. Auch für diese Hindernisse werden Lösungen vorgeschlagen, getestet und evaluiert.

1 XML, TEI und XML-annotierte Editionstexte aus dem Nachlass Ludwig Wittgensteins

1.1 Kurze Einführung in XML

Gegen Ende des 20. Jahrhunderts wurde die genormte verallgemeinerte Auszeichnungssprache SGML (Standard Generalized Markup Language) im Rahmen eines IBM-Projektes von Charles Goldfarb, Edward Moser und Raymond Lorie erfunden [15]. Das Ziel des Projektes war ein einheitliches Zeichensystem aus deskriptiven Auszeichnungen zu kreieren, das eine elektronische Zusammenarbeit mit Texten und Dokumenten ermöglicht. SGML war später aber so ausführlich, dass es sich nicht an Webseiten anpassen konnte. Aus diesem Grund wurden viele unnötige Zeichen durch Verallgemeinernde ersetzt, die diese Zeichensysteme kleiner, anpassungsfähiger und effizienter machten. Aus dieser Idee hat sich XML herauskristallisiert.

Die Buchstaben XML stehen für eXtensible Markup Language (Erweiterbare Auszeichnungssprache) [13]. Um zu verstehen, was XML bedeutet, sollte man zuerst die Begriffe verstehen, die den Namen bilden. Markups sind Auskünfte, bzw. Auszeichnungen, die zu einem bestimmten Dokument hinzugefügt werden. Sie kennzeichnen den Inhalt und die Teilstrukturen des Dokumentes und stellen dar, wie sie miteinander verbunden sind [8, S. 2]. Diese Auskünfte können zusammengefasst und in einem Dokument an gewünschten Stellen anlegt werden. Somit können die einzelnen Teile des Dokumentes von den anderen differenziert oder klassifiziert werden. Die Menge dieser Symbole wird Auszeichnungssprache (Markup Language) genannt. Sie enthält keine bestimmte Anzahl an Auszeichnungen und ist beliebig erweiterbar (extensible). Im digitalen Bereich sind die Markups sehr beliebt, denn sie können von dem Computer erkannt werden. Deswegen kann ein Entwickler oder eine Entwicklerin mit Hilfe von Auszeichnungen einem Programm beibringen, welche die einzelnen Teile eines Dokumentes sind, die bearbeitet oder geändert werden sollen.

Laut Erik T. Ray ist XML jedoch selbst keine Auszeichnungssprache. Es beinhaltet aber eine Menge von Regeln [8, S. 2], die die Erzeugung von Auszeichnungssprachen ermöglichen. XML ist also ein sehr wichtiges Werkzeug für die Verwaltung und Aufnahme von Daten. Die wichtigsten XML-Bestandteile sind Elemente, Attribute, Namensräume und Entitäten.

Die Elemente sind die Bausteine eines XML-Dokumentes. Sie enthalten Texte oder andere Dokumente und gliedern das Dokument in Teile, die von einer Suchmaschine verwendet werden können. Die Elemente sind in einer Baumstruktur angelegt und müssen eine eindeutige Hierarchie aufweisen. Ein Element besteht aus zwei Teilen: einem öffnenden Tag (Auszeichnung), bzw. Starttag und einem schließenden Tag, bzw. Endtag. Das Element `<persName>` in Abbildung 1.1 beinhaltet nur Text:

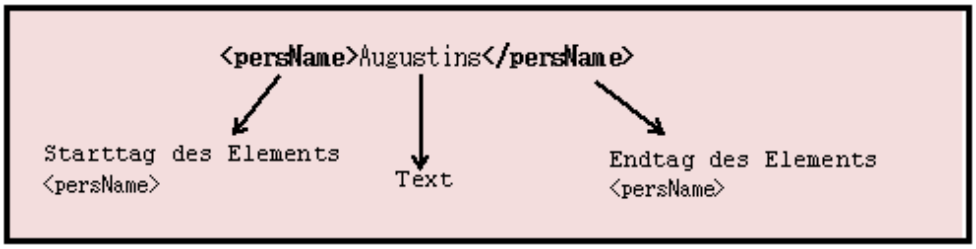

Abbildung 1.1: Element `<persName>`

In Abbildung 1.2 hingegen enthält das Element `<persName>` Text und auch ein anderes Element, das wiederum Text enthält:

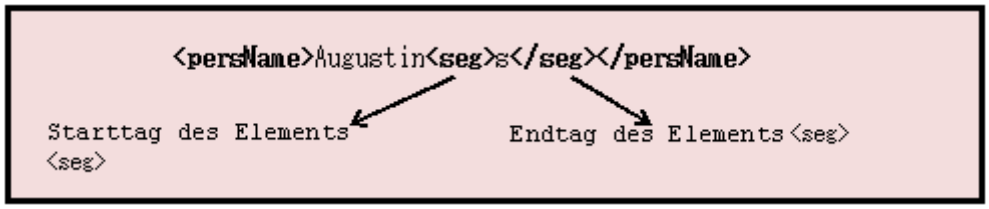

Abbildung 1.2: Element `<persName>` enthält Element `<seg>`

Elemente können auch leer sein. In dem Fall gibt es nur ein Tag mit Schrägstrich am Ende. Ihre Funktion hängt von ihrer Position und ihren Attributen ab. In Abbildung 1.3 beispielsweise ist das Element `<lb>` ein leeres Element, das einen Zeilenumbruch bedeute:

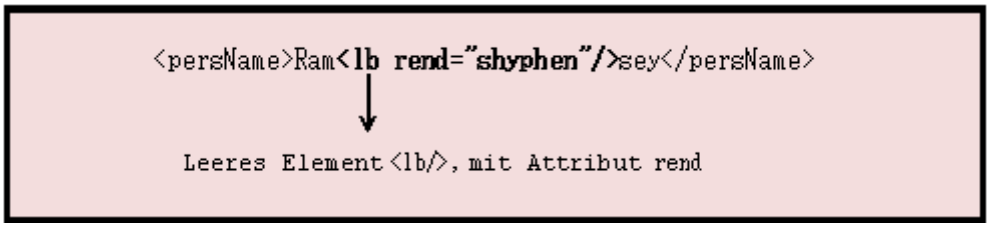

Abbildung 1.3: Leeres Element `<lb>`

Die Attribute erweitern die Informationen der Elemente. Sie werden innerhalb des Elementes eingefügt und ihnen wird ein bestimmter Wert zugewiesen. In Abbildung 1.4 hat das Element `<persName>` das Attribut `key` mit dem Wert `Ramsey, Frank Plumpton` und das Element `<lb>` das Attribut `rend` mit dem Wert `shyphen`:

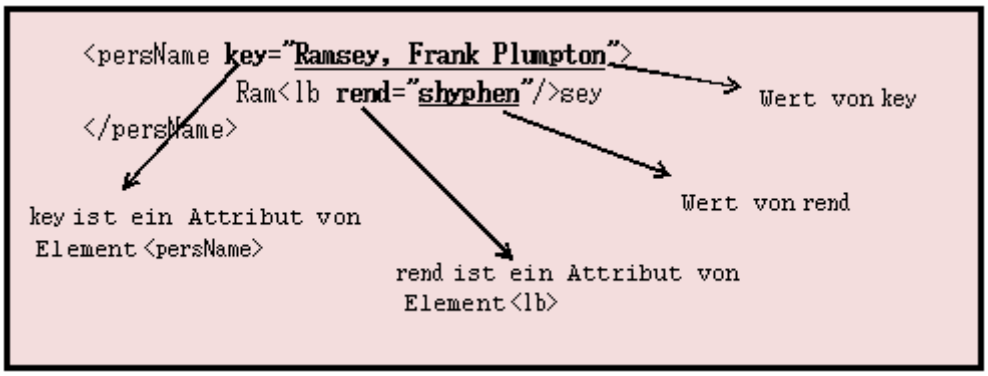

Abbildung 1.4: Elemente `<persName>` und `<lb>` und ihre Attribute

Ein XML-Dokument kann andere XML-Dokumente enthalten, die sich auch im Typ unterscheiden dürfen [8, S. 47]. Die Abgrenzung dieser Dokumenttypen erfolgt, indem ein sog. Namensraum (Namespace) pro Dokumententyp mit den jeweiligen dazugehörigen Elementen und Attributen deklariert wird. Im Kapitel 3 wird gezeigt, welchen Namensraum die XML-Editionstexte von Wittgensteins Nachlass besitzen, warum er in diesem konkreten Fall wichtig ist und wie er mit dem Python3-Parser ElementTree deklariert werden kann.

Die Entitäten (Entities) sind Platzhalter für Inhalte [8, S. 51]. Sie werden nur einmal deklariert und können an fast jeder Position des Dokumentes verwendet werden. Einige Entitäten enthalten nur ein Zeichen und sind als Zeichen-Entitäten bekannt. Es gibt schon einige von XML vordefinierte Zeichen-Entitäten, die in Tabelle 1.1 präsentiert werden. Hierbei werden die Entitäten als Name bezeichnet und die Inhalte, für die sie stehen, als Wert:

Name	Wert
amp	&
apos	‘
gt	>
lt	<
quot	“

Tabelle 1.1: Zeichen-Entitäten [8, S. 54]

Fast jede Programmiersprache stellt einen XML-Parser zur Verfügung. In dieser Bachelorarbeit wird den Python3-Parser `xml.etree.ElementTree` (auch ElementTree oder einfach etree genannt) verwendet. Mit diesem Werkzeug kann eine XML-Datei erzeugt, geparst oder geändert werden. Im Kapitel 3 wird durch konkrete Beispiele gezeigt, wie dieser Parser bei dem Parsen und der Veränderung von XML-Dateien verwendet werden kann. Im Anhang befindet sich ein einfaches Programm mit dem Namen *xmlET_schnell_Ueberblick.py,* das einen schnellen Überblick über die Generierung von XML-Dateien mit der Python-Programmierschnittstelle etree beinhaltet.

ElementTree bietet eine Vielzahl von praktischen Funktionen, die die Erzeugung einer XML-Datei mit einer hierarchischen Baumstruktur ermöglicht, zum Beispiel:

- **`Element()`:** Die Funktion `Element()` hilft dabei, den Wurzel- bzw. Elternknoten herzustellen:

  ```
  wurzel = Element('inhalt')
  ```

- **`SubElement()`:** Die Tochterknoten werden mit der Funktion `SubElement()` generiert und dem gewünschten Element angehängt. Mithilfe der Methode `text()` kann ein Inhalt einem Element zugewiesen oder auch außerhalb des Elementes mit `tail()` hinzugefügt werden:

  ```
  subElem2 = SubElement(wurzel, 'werke')
  subElem2.text = 'hier den Text schreiben'
  subElem2.tail = 'hier den Text schreiben'
  ```

- **`Comment()`:** Kommentare können zu jeder Zeit eingefügt werden. Dafür wird eine Variable erzeugt, die den Inhalt des Kommentars im String-Format als Wert bekommt, der in der Funktion `Comment()` als Argument dargestellt wird. Mit der Verwendung der Methode `append()` wird diese Variable den gewünschten Elementen angehängt:

  ```
  comment = Comment('hier den Text schreiben')
  ```

```
wurzel.append(comment)
```

ElementTree stellt die Methode `tostring` für das Ausgeben des resultierenden XML-Dokuments zur Verfügung. Wenn sich die Programmierenden aber eine stilisierte Ausgabe wünschen, werden sie von `tostring` enttäuscht. Durch einen anderen XML-Python-Parser DOM (Document Object Model) und das dazugehörige Modul `xml.dom.minidom` wird die Standardfunktion `printString()` aufgerufen. Sie wird in der zusammengebauten Methode `prettify()` [5] in *xmlET_schnell_Ueberblick.py* implementiert, die eine strukturierte und deutliche Ausgabe der neuen XML-Datei ausgibt.

Beispielausgabe:

```
1.      <inhalt>
2.          <!-- Kommentar -->
3.          <werke>… </werke>…
4.      </inhalt>
```

XML tut sich durch die Flexibilität hervor, was sehr bequem für die Programmierenden und alle XML-Nutzer ist. Da der XML-Parser Fehler kaum toleriert, repräsentiert diese Flexibilität allerdings auch eine Gefahr für ihn. Dementsprechend wurden feste Regeln gesetzt, die die Wohlgeformtheit eines XML-Dokumentes überprüft, in dem einige syntaktische Strukturen der XML-Dateien beschränkt werden. Diese Regeln werden im Folgenden aufgelistet:

1) Ein Element, das Text oder Elemente enthält, muss ein Start- und ein End-Tag haben.
2) Der Tag eines leeres Elements muss vor der Abschlussklammer einen Schrägstrich (/) haben.
3) Alle Attribut-Werte müssen in Anführungszeichen stehen.
4) Elemente dürfen sich nicht überlappen.
5) In vom Parser analysierten Inhalten dürfen keine isolierten Markup-Zeichen erscheinen (vgl. Tabelle 1.1).
6) Namen von Elementen dürfen nur mit Buchstaben oder Unterstrichen beginnen und nur Buchstaben, Zahlen, Bindestriche, Punkte und Unterstriche enthalten. Für Namensräume sind Doppelpunkte erlaubt.

Auflistung 1.1: Wohlgeformtheits-Regeln. [8, S. 64-66]

Die Präsentation eines XML-Dokumentes ist auch ein wichtiger Aspekt. Bei der Präsentation wird beschrieben, wie XML-Dateien aussehen sollen. Diese Anweisungen zum Format werden in den sogenannten Stylesheet-Dokumenten definiert. Man spricht hier auch von einer Transformation der XML-Dateien. Diese Transformation wird heutzutage meistens mittels XSL (Extensible Stylesheet Language) durchgeführt. Wie Frank Bongers erklärt, teilt sich XSL in zwei Untergruppen [1, S. 28]:

1. *„XSL-FO: XSL Formatting Objekts“*
2. *„XSLT: Extensible Stylesheet Language Transformation“*

Im Punkt 1.3 werden drei wichtige XML-Dokumente beschrieben und eine davon im Laufe dieser Bachelorarbeit genauer analysiert. Die Editoren aus dem Wittgenstein Archiv der Universität Bergen haben sie aus den originalen XML-Dateien des

Wittgenstein Nachlasses mittels XSLT-Dokumenten nach Bedarf der WiTTFind Entwickler transformiert.

1.2 Kurze Einführung in TEI-P5

Die Text Encoding Initiative (TEI) ist eines der wichtigsten Projekte im Bereich Digital Humanities. Dieses Projekt hat sich in der gleichnamigen Organisation herauskristallisiert, die im Jahr 1987 gegründet wurde [16]. Das Ziel von TEI ist es, Richtlinien für die Erzeugung und Verarbeitung von Texten, Manuskripten, Nachlässen, etc. festzulegen, die in einem digitalen Format von Wissenschaftlern bzw. Forschern im Bereich der Geisteswissenschaft erzeugt werden. Die aktuellste Version ist die sogenannte TEI P5. Sie basiert auf XML [16] und hat eine enorme Bedeutung für die Kodierung und den Austausch von XML-Dateien:

> *„[...]. What advantage is there in learning XML or TEI to do the same job? There are many answers, [...]. The first is that TEI XML focuses on the meaning of text, rather than its appearance. The second is that TEI XML is independent of any particular software environment. The third is that TEI XML was designed by and for the scholarly research community, which is also responsible for its ongoing development.“* [2]

Die Universität Bergen, zu der das WAB gehört, ist ein Kernmitglied der Text Encoding Initiative. Dementsprechend erfolgt die Transformation der „Wittgenstein Source Bergen Nachlass Edition“ (BNE) im XML Format entsprechend der TEI Richtlinien. In den XML-Dateien, die speziell für das CIS erzeugt wurden (genannt CISWAB), findet man auch noch andere TEI-Elemente, die für diese CIS-Dateien angepasst werden.

Im Folgenden werden die TEI-XML-Elemente [14] vorgestellt, die relevant für diese Bachelorarbeit sind und bei den hier verwendeten normalisierten Dateien [7] am häufigsten vorkommen.

1.2.1 Element `<persName>` und konkrete Attribute in BNE

Das Element `<persName>` (person name) ist spezifisch für CISWAB-Dateien. Es markiert einen Personennamen.

a) `<persName key="...">`: Das Attribut `key` besitzt als Wert den korrekten und vollständigen Personennamen, der im Text erwähnt wurde.

   Beispiel 1.1 (aus *Ms-115_OA.NORM.xml,* Ms-115,124[4]et125[1]_4):

```
<persName key="Russell, Bertrand">Russells</persName>
```

1.2.2 Element `<abbr>` und konkrete Attribute in BNE

Das Element `<abbr>` (abbreviation) enthält eine Abkürzung beliebiger Form.

a) `<abbr type="abb">`: Attribut-Wert `abb` bedeutet, dass es sich um eine Standard Abkürzung handelt.

Beispiel 1.2 (aus *Ms-115_OA_NORM.xml*, Ms-115,10[4]et11[1]_3):

```
<abbr type="abb">z.B.</abbr>
```

b) `<abbr corresp="...">`: Das Attribut `corresp` weist darauf hin, dass es sich um eine nicht konventionelle Abkürzung handelt.

Beispiel 1.3 (aus *Ms-101_OA_NORM.xml,* Ms-101,1r[1]_49):

```
Guter <abbr corresp="Stimmung">St.</abbr>
```

1.2.3 Element `<emph>` und konkrete Attribute in BNE

Das Element `<emph>` (emphasized) markiert Wörter oder Phrasen, die aufgrund eines linguistischen oder rhetorischen Zwecks hervorgehoben werden.

a) `<emph rend="us1">`: Betonung mit einem geraden Unterstrich.

Beispiel 1.4 (aus *Ms-115_OA_NORM.xml*, Ms-115,230[3]_1):

```
Man <emph rend="us1">muß</emph> einen Befehl verstehen
```

b) `<emph rend="us1_c">`: Betonung mit einem geraden Unterstrich, stornierte Betonung.

Beispiel 1.5 (aus *Ms-115_OA_NORM.xml*, Ms-115,232[4]et233[1]_1):

```
wenn er nicht <emph rend="us1_c">vorher</emph> erkannt hat
```

c) `<emph rend="us1_h">`: Betonung mit einem geraden Unterstrich und mit einer unterschiedlichen Schreibweise.

Beispiel 1.6 (aus *Ts-310_OA_NORM.xml*, Ts-310,158[3]et159[1]et160[1]_6):

```
<emph rend="us1_h">expression</emph>
```

d) `<emph rend="us1_h_ch">`: Betonung mit einem geraden Unterstrich mit einer unterschiedlichen Schreibweise, stornierte Betonung.

Beispiel 1.7 (aus *Ts-122_OA_NORM.xml,* Ts-222,102[1]et103[1]_6):

```
<emph rend="us1_h_ch">Nähen</emph>
```

e) `<emph rend="usb">`: Betonung mit Unterstrich und Zeilenumbruch.

Beispiel 1.8 (aus *Ts-245_OA_NORM.xml,* Ts-245,134[2]_11):

```
<emph rend="usb">Beziehungen</emph>
```

f) `<emph rend="us2">`: Betonung mit zwei geraden Unterstrichen.

Beispiel 1.9 (aus *Ms-115_OA_NORM.xml,* Ms-115,259[2]_3):

```
<emph rend="us2">Worte</emph>
```

g) `<emph rend="uw1">`: Betonung mit unterkringeltem Unterstrich.

Beispiel 1.10 (aus *Ms-115_OA_NORM.xml*, Ms-115,259[3]et260[1]_1):

```
<emph rend="uw1">(überall)</emph>
```

h) `<emph rend="uw1_c">`: Betonung mit unterkringeltem Unterstrich, stornierte Betonung.

Beispiel 1.11 (aus *Ms-149_OA_NORM.xml*, Ms-149,26v[1]et27r[1]_16):

```
<emph rend="uw1_c">are</emph>
```

i) `<emph rend="uw1_h">`: Betonung mit unterkringeltem Unterstrich mit einer unterschiedlichen Schreibweise.

Beispiel 1.12 (aus *Ts-240_OA_NORM.xml*, Ts-240,216b[3]et217[1]_1):

```
<emph rend="uw1_h">wesentlicher</emph>
```

j) `<emph rend="space">`: Wort oder Phrase betont mit Leerzeichen-Trennungen.

Beispiel 1.13 (aus *Ts-240_OA_NORM.xml*, Ts-240,216b[3]et217[1]_4):

```
<emph rend="space">Bedeutung</emph>
```

k) `<emph rend="sub">`: tiefgestellte Betonung (in subscript).

Beispiel 1.14 (aus *Ms-115_OA_NORM.xml*, Ms-115,183[4]et184[1]et185[1]_2):

```
<emph rend="sub">n</emph> =  n²  +  n  +  1
```

l) `<emph rend="sup">`: hochgestellte Betonung (in superscript).

Beispiel 1.15 (aus *Ms-115_OA_NORM.xml,* Ms-115,271[2]_2):

```
um 3<emph rend="sup"> <abbr type="abb">h</abbr>30</emph>
```

1.2.4 Element `<del>` und konkrete Attribute in BNE

Das Element `<del>` (deletion) enthält Buchstaben, Wörter oder Phrasen, die von einem Autor, Schreiber oder im Rahmen einer zuvor erfolgten Annotation oder Korrektur in dem Ausgangstext weggelassen wurden.

a) `<del type="d" status="unremarkable">`: Das weggelassene Objekt übt keinen auffälligen Einfluss auf den gebliebenen Text aus.

Beispiel 1.16 (aus *Ms-115_OA_NORM.xml,* Ms-115,271[2]_4):

```
Gefühle den<lb rend="shyphen"/>ken,
     <del type="d" status="unremarkable">
          die Duhattest
     </del>
```

b) `<del type="d_h">`: Weglassung mittels unterschiedlicher Schreibweise.

Beispiel 1.17 (aus *Ts-245_OA_NORM.xml,* Ts-245,163[4]et164[1]_5):

```
<del type="d_h" status="unremarkable"/>
     werfen
<del type="d_h" status="unremarkable"/>
```

1.2.5 Element `<choice>` und konkrete Attribute in BNE

Das Element `<choice>` gruppiert alternative Kodierungen für eine Stelle im Text.

a) `<choice type="s"> <orig type="alt1">a</orig> <orig type="alt2">b</orig> </choice>`: Ersetzung zw. Alternativen *a* und *b*.

Beispiel 1.18 (aus *Ms-115_OA_NORM.xml,* Ms-115,271[2]_4). Ersetzung von *es* durch die Alternativen *diese/die Worte*:

```
<choice type="s">
     <seg n="s_alt1">es</seg>
     <seg n="s_alt2">
          <choice type="s">
               <seg n="s_alt1">diese</seg>
               <seg n="s_alt2">die</seg>
          </choice> Worte</seg>
</choice>
```

1.2.6 Element `<seg>` und konkrete Attribute in BNE

Das Element `<seg>` (arbitrary segment) repräsentiert irgendein Textsegment unter einer willkürlichen Segment-Ebene.

a) `<seg n="s_alt..">`: (siehe `<choice type="s">`).
b) `<seg type="stripped">` bedeutet, dass die WAB-Editoren an dieser Stelle irgendeine Information absichtlich bei der Transformation in CISWAB-Dateien weggelassen haben.

Beispiel 1.19 (aus *Ts-245_OA_NORM.xml,* Ts-245,164[5]_2):

```
<seg type="stripped">V</seg>orschweben
```

c) `<seg type="notation">`: Bilder werden gelinkt.

Beispiel 1.20 (aus *Ms-115_OA_NORM.xml,* Ms-115,140[5]et141[1]_3):

```
nach dem Schema
<seg type="notation">
     corresp="http://wab.uib.no/cost-
          a32_fax/graphicsHB/115,141-1.jpg">115008
</seg>
```

d) `<seg type="date" n="..." part="N">`: Falls ein Teil einer Datums-Angabe erwähnt wird, wird dieses Datum mit dem Wert von `n` vollständig angezeigt.

Beispiel 1.21 (aus *Ms-101_OA_NORM.xml*, Ms-101,27v[2]_7):

```
die sie am <seg type="date" n="19140820" part="N">20.8.</seg>
schrieb
```

1.2.7 Element `<lb>` und konkrete Attribute in BNE

Das Element `<lb>` (line break) markiert den Beginn einer neuen typographischen Zeile in einer bestimmten Auflage oder Version eines Textes.

a) `<lb rend="shyphen"/>`: Attribut `rend` (rendition) markiert wie das vorhandene Element im Quelltext vorkommt. In diesem Fall zeigt der Attribut-Wert `shyphen` (hyphenation), dass der Zeilenumbruch ein gewiesenes Wort trennt.

Beispiel 1.22 (aus *Ms-115_OA_NORM.xml,* Ms-115,68[3]et69[1]_1):

```
Aber wie kann ich entscheiden, welches<lb/> ein wesentlicher
&amp; welches
ein unwesent<lb rend="shyphen"/>licher,zufälliger Zug der
Notation ist?
```

1.2.8 Element <pb> und konkrete Attribute in BNE

Das Element <pb> (page break) markiert den Beginn einer neuen Seite in einem Dokument mit Seitenzahlen.

a) <pb facs="Ms-115_95"/>: Attribut facs bezeichnet die Faksimile, zu der diese Seite gehört.

Beispiel 1.23 (aus *Ms-115_OA_NORM.xml,* Ms-115,95[7]et96[1]_3):

```
Aber das ist doch nur eine unkorrekte Ausdrucksweise; es
müßte heißen:<pb facs="Ms-115_96"/> man schließt, daß man
Zahnschmerzen
```

1.2.9 Element <s> und konkrete Attribute in BNE

Das Element <s> (sentence) enthält die Satztrennungen vom Text.

a) <s n="..." ana="... abnr:... satznr:...">: Das Attribut n bezeichnet die Koordinate von Faksimile. Das Attribut ana (analysis) bietet zusätzliche Information an, die den Faksimile-Namen erwähnt. Das Attribut abnr (number of anonymous block) zeigt die Absatznummer und das Attribut satznr die Satznummer.

Beispiel 1.24 (aus *Ms-115_OA_NORM.xml,* Ms-115,70[2]_1):

```
<s n="Ms-115,70[2]_1" ana="facs:Ms-115_70 abnr:174 satznr:702">
     Das Spiel soll doch durch die Regeln<lb/> bestimmt sein!
</s>
```

1.2.10 Element <ab> und konkrete Attribute in BNE

Das Element <ab> (anonymous block) enthält eine willkürliche Komponenten-Ebene, die sich als ein anonymer Container für Phrasen und Sätze verhält. In diesem Fall bezieht sich dieses Element auf einen Absatz.

a) <ab n="..." ana="...">: Das Attribut n bezeichnet die Koordinate vom Faksimile. Das Attribut ana (analysis) zeigt die Absatznummer.

Beispiel 1.25 (aus *Ms-115_OA_NORM.xml,* Ms-115,94[4]et95[1]):

```
<ab n="Ms-115,94[4]et95[1]" ana="abnr:240">
     <s n="Ms-115,94[4]et95[1]_1" ana="facs:Ms-115_94 abnr:240
          satznr:952">Ist der Satz
          <seg type="stripped">“es wird regnen” die</seg>
          Beschreibung meiner Geistestätigkeit,<lb/> da er doch
          die Wiedergabe meines Gedan<lbrend="shyphen"/>kens
          ist, daß es regnen wird?
```

```
     </s>
     <s n="Ms-115,94[4]et95[1]_2" ana="facs:Ms-115_94 abnr:240
          satznr:953">— Wir<lb/> werden nicht so leicht geneigt
          sein,<lb/> den Satz die Beschreibung der
          Geistestätig<lb rend="shyphen"/>keit zu nennen, wenn
          wir bedenken, daß<lb/> das Denken im Reden bestehen
          kann,
          <pb facs="Ms-115_95"/> keine Begleitung des
          Gedankenausdruckist.
     </s>
</ab>
```

1.3 Hinweise über die Editionstexten des Ludwig Wittgenstein Nachlasses

Der Wittgenstein Nachlass ist eine Sammlung von teils unveröffentlichten Materialien, die vom Philosophen Ludwig Wittgenstein im Zeitraum 1906-1916 verfasst wurden. Er besteht aus 20.000 Seiten und teilt sich in zwei Gruppen: die Manuskripte (Ms) und die Typoskripte (Ts).

Wie Prof. Dr. Hans Walter Gabler erklärt, befinden sich die originalen Dokumente des Wittgenstein-Nachlasses in „*Archiven und in Privatbesitz verstreut über Europa und Kanada. Ihren Hauptbestand verwahrt die Trinity College Library in Cambridge, England*“ [3, S. 159].

Das WAB hat diese Manuskripte und Typoskripte ins XML Format transkribiert und in Teilen als Open-Access-Daten auf seiner Webseite des BNE zur Verfügung gestellt.

Das CIS arbeitet seit mehr als sieben Jahren zusammen mit dem WAB an dem Digital-Humanities-Projekt „Wittgenstein in Co-Text“. Aus diesem Projekt wurde sehr erfolgreich die FinderApp „WiTTFind“ für die semantische Suche durch den Nachlass erzeugt, die den europäischen Preis „European Union - OPEN Humanities AWARD“ im Jahr 2014 gewann. Zu diesem Projekt hat der Direktor des WAB Prof. Dr. Alois Pichler mit 5.000 Seiten im XML-Format beigetragen. In enger Zusammenarbeit mit dem Leiter des Projekts am CIS Dr. Maximilian Hadersbeck haben die Editoren aus Bergen mehrere XSLT-Dateien erzeugt, um die XML-Manuskripte und -Typoskripte immer mehr dem Bedarf der Entwickler der FinderApp anzunähern. Diese Tatsache beweist, wie nützlich die Anwendung von XML in der Digitalisierung des Wittgenstein Nachlasses ist.

Diese XSLT-Dateien erzeugen die folgenden drei unterschiedlichen Dateitypen, mit denen WiTTFind unmittelbar arbeitet:

OA.xml-Dateien: Sie enthalten die Dateien, exakt wie im Dokument von Ludwig Wittgenstein, mit Vorschlägen zur Korrektur.

Beispiel 1.26 (aus *Ms-114_OA.xml,* Ms-114,23r[2]et23v[1]):

```
<s type="es" part="N">
```

```
     <seg type="q" part="N">„Eine herrenlose Wohnung”</seg>,
     <seg type="q" part="N">„herrenlose<lb/>
          <choice type="em">
               <orig type="em1">
                    <add rend="i" status="unremarkable">
                         Zahn
                    </add>
                    <add rend="el" status="unremarkable">-
                    </add>
                    <corr type="trsn">
                         <orig type="trsn1">S</orig>
                         <reg type="trsn2">s</reg>
                    </corr>
                         chmerzen
               </orig>
               ...
     </seg>
</s>
```

NORM.xml-Dateien: Die Dateien wurden in eine gut lesbare Version aus der Sicht der Experten und Editoren gebracht. Vergleichbar mit des „author's last will“, könnten sie wahrscheinlich in einem Buch gedruckt werden.

Beispiel 1.27 (aus *Ms-114_OA_NORM.xml,* Ms-114,23r[2]et23v[1]_6)

```
<s n="Ms-114,23r[2]et23v[1]_6" ana="facs:Ms-114_23r abnr:48
satznr:397">
     „Eine herrenlose Wohnung”, „herrenlose<lb/>
     <seg type="stripped">
          <seg type="stripped">
               Zahn<seg type="notation">-</seg>
          schmerzen
          </seg>
     </seg>”.
</s>
```

DIPLO.xml-Dateien: Die Texte behalten die originale Version von Wittgenstein. Es wird hier zudem versucht, die Details des Nachlasses mit XML möglichst genau zu beschreiben.

Beispiel 1.28 (aus *Ms-114_OA_DIPLO.xml,* Ms-114,23r[2]et23v[1]_6)

```
<s n="Ms-114,23r[2]et23v[1]_6" ana="facs:Ms-114_23r abnr:48
     satznr:397">
     „Eine herrenlose Wohnung”, „herrenlose<lb/>
     <seg type="stripped">
          <choice type="dsl">
               <seg n="dsl_alt1">Schmerzen</seg>
               <seg n="dsl_alt2">Zahn<seg type="notation">-
                    Schmerzen
```

```
                </seg>
            </choice>
        </seg>”.
</s>
```

Im Laufe dieser Bachelorarbeit werden hauptsächlich die normalisierten Dateien (NORM.xml Dateien) analysiert.

2 POS-Tagging

Ein wichtiger Forschungsschwerpunkt in der Computerlinguistik ist die Wortarterkennung („Part of Speech-Tagging“ oder einfach „POS-Tagging“). Die richtige Zuordnung von Wörtern eines Textes zu Wortarten dient u.a. zur Desambiguierung der Wortbedeutungen, was eine riesige Bedeutsamkeit in vielen Teilgebieten der Computerlinguistik hat, z.B.: Named Entity Recognition (NER), maschinelle Übersetzung, Sprachidentifizierung, etc.

Zu den Methoden für das POS-Tagging zählen probabilistische bzw. nicht probabilistische, sowie lokale Grammatiken. Die App WiTTFind benutzt probabilistische Methoden, um das Tagging vom Wittgenstein Nachlass durchzuführen. Für die semantische Suche werden die Wortarten benötigt.

In diesem Kapitel wird gezeigt, welchen konkreten POS-Tagger WiTTFind verwendet, welche seine wichtigsten Merkmale sind, wie er im Einsatz bei XML-Dateien funktioniert, an welcher Stelle und warum manche Wörter irrtümlicherweise tokenisiert und getaggt werden.

2.1 POS-Tagging mit dem TreeTagger

Die Suchmaschine WiTTFind benutzt den „TreeTagger“ für die Wortarterkennung. Der TreeTagger wurde von einem aktuellen Dozenten des CIS Dr. Helmut Schmid am Institut für maschinelle Sprachverarbeitung (IMS-CL) der Universität Stuttgart entwickelt.

Der TreeTagger ist ein statistischer bzw. probabilistischer POS-Tagger, der auf Markov Modellen basiert. Die Absicht des TreeTaggers ist es, das wahrscheinlichste Wortart für ein Wort zu finden und diese als Tag an das Wort zu hängen. Dafür berechnet er die Tagsequenz- und die Übergangswahrscheinlichkeiten [10].

Im Gegensatz zu anderen probabilistischen Taggern, wie zum Beispiel n-gram-Taggern, berechnet der TreeTagger die Übergangswahrscheinlichkeiten mit einem binären Entscheidungsbaum (binary decision tree) [10, S. 2]. Das Ziel dieser Innovation, wie Dr. Schmid berichtet, ist das Sparse-Data-Problem zu reduzieren, von dem alle anderen statistischen Tagger betroffen sind. Das Sparse-Date-Problem erscheint, wenn nicht genug Trainingsdaten vorhanden sind, was zu einem schlechten Tagging führt. Die Entscheidungsbäume werden deswegen vom TreeTagger rekursiv aus N-Grammen generiert.

Früher hat sich die Arbeit des TreeTaggers auf die englische Sprache konzentriert, bei der er eine hohe Genauigkeit erreichen konnte. Es wurde aber bemerkt, dass der TreeTagger nicht effizient genug bei der Deutschen Sprache funktioniert aufgrund des Mangels an annotierten Trainingsdaten. Im Jahr 1995 hat Dr. Schmid ein neues Paper [11, S. 1-9] veröffentlicht, in dem er eine Erweiterung des TreeTaggers angekündigt hat, durch die er eine höhere Genauigkeit beim Training mit kleineren Korpora erreicht und somit auch in anderen Sprachen gute Resultate erzielt.

Der TreeTagger berechnet den wahrscheinlichsten Tag und das wahrscheinlichste Lemma eines Wortes. Leider kann das Lemma nur zurückgegeben werden, wenn es explizit in dem Lexikon des TreeTaggers vorhanden ist. Diese Tatsache kann auch als eine Art von Sparse-Data-Problem bezeichnet werden. In Bezug auf den Nachlass besteht dieses Sparse-Data-Problem im Falle von Personennamen immer noch. In den nächsten Kapiteln, vor allem in Kapitel 3, wird gezeigt, dass wenn ein Personenname oder seine Varianten in den Trainingsdaten des TreeTaggers nicht vorhanden sind, wird dieser Name falsch getaggt und mit dem Lemma `UNKNOWN` zurückgeliefert.

Jede Zeile aus dem Lexikon des TreeTaggers entspricht dem Eintrag eines Wortes (in Vollform) gefolgt von einem Tabulator-Zeichen und einer Sequenz von Tag-Lemma-Paaren, getrennt durch ein Leerzeichen. Dieses Lexikon sieht wie folgt aus [12]:

muessen VMFIN müssen

Die Universität Stuttgart und die Universität Tübingen haben das spezielle Tagset STTS (Stuttgart-Tübingen-TagSet) für das Tagging deutscher Korpora entwickelt [9, S. 3], das der TreeTagger für die Annotation deutscher Texte verwendet. Da die meisten Dokumente aus dem Wittgenstein Nachlass in der deutschen Sprache verfasst wurden, wird hier auch das STTS verwendet. Aus diesem Grund werden im Nachstehenden die Tag-Abkürzungen und Ihre Beschreibung aufgelistet, auf die in den nächsten Kapiteln Bezug genommen wird:

Tag	**Beschreibung**
N	Nomen
NN	Appellativ
NE	Eigenname
TRUNC	Kompositions-Erstglied
ART	Artikel
ADJ	Adjektiv
ADJA	Attributives Adjektiv oder adjektivischer Gebrauch von Herkunftsbezeichnungen
ADJD	Adverbiales oder prädikatives Adjektiv
V	Verb
VVFIN	Finites Verb, voll
VMFIN	Finites Modalverb
VVIMP	Imperativ, voll
XY	Nichtwort, Sonderzeichen
$.	Satzbeendende Interpunktion
ABK	Abkürzung
CARD	Kardinalzahl

Tabelle 2.1: Tags und ihre Beschreibung [9, S. 2-6]

2.2 Arbeitsweise des TreeTaggers bei NORM.xml-Dateien

Wie schon erwähnt wurde, verwendet die WiTTFind App den TreeTagger für das Tagging der Tokens im Wittgenstein Nachlass. Der Tagger läuft über alle NORM.xml-Dateien und generiert jeweils eine neue getaggte Datei namens NORM-tagged.xml. In den Beispielen 2.1 und 2.2 kann man den Unterschied zwischen beiden Dateien deutlich merken:

Beispiel 2.1 (aus *Ms-101_OA_NORM.xml,* Ms-101,11r[4]et12r[1]_10):

```
<s n="Ms-101,11r[4]et12r[1]_10" ana=" abnr:18 satznr:125">
     Gestern fing ich an in
     <persName key="Tolstoj, Lew">
          Tol<seg type="stripped">s</seg>tois
     </persName>
     <lb/> Erläuterungen zu den Evangelien zu lesen.
</s>
```

Beispiel 2.2 (aus *Ms-101_OA_NORM-tagged.xml,* Ms-101,11r[4]et12r[1]_10):

```
<s n="Ms-101,11r[4]et12r[1]_10" ana=" abnr:18 satznr:125">
<w t="ADV" l="gestern">Gestern</w>
<sp/>
<w t="VVFIN" l="fangen">fing</w>
<sp/>
<w t="PPER" l="ich">ich</w>
...
<persName key="Tolstoj, Lew">
<w t="ADJA" l="UNKNOWN">Tol<seg type="stripped">s</seg>tois</w>
</persName>
<lb/>
...
<w t="NN" l="Erläuterung">Erläuterungen</w>
<sp/>
...
<sp/>
<w t="VVINF" l="lesen">lesen</w>
<w t="$." l=".">.</w>
</s>
```

Das Shell Hauptprogramm *prozess.sh* (vgl. Abbildung 2.1) steuert andere Programme, die die NORM.xml-Dateien in mehreren Schritten bearbeiten. Bevor der TreeTagger mit dem Tagging anfängt (Zeile 9), wird die Datei vorbereitet bzw. normalisiert. Die ersten Schritte vor dem Tagging werden als Preprocessing bezeichnet (Zeile 2-8). Der Header der NORM.xml-Datei wird mit dem Programm *extract-text.perl*[2] in der Zeile 5 entfernt und der Teil der XML-Datei extrahiert, der nur den Text enthält. Außerdem führt dieses Skript das XML-Element `<sp/>` zwischen jedem Wort des Textes, getrennt mit Leerzeichen ein, aber nicht innerhalb der bereits vorhandenen XML-Elemente. Die

2 Skript *extract-text.perl* von Dr. Helmut Schmid wurde im Laufe dieser Arbeit nicht geändert. Deswegen ist es auf der beigelegten CD nicht vorhanden.

bereits vorhandenen XML-Elemente werden auch von dem Inhaltstext durch ein Leerzeichen getrennt und innerhalb derer wird das Sonderzeichen „ð“ mittels des Programms *filtergerman-tokin-utf8.perl* (siehe CD im Anhang) hinzugefügt, damit es kein Leerzeichen gibt. Dieses Programm wird vom Shell Skript *tokenize-german-utf8* in Zeile 7 aufgerufen. Nach Zeile 7 werden die XML-Dateien wie im Beispiel 2.3 aussehen.

```
1.  do
2.    echo "$f"
3.    destination="$(dirname $f)/$(basename $f .xml)-tagged.xml"
4.    cmd/put-xml-tags-on-one-line.perl $f | \
5.    cmd/extract-text.perl | \
6.    perl -pe 's/ ([0-9]+)\)/ $1\n) /g' | \
7.    $SHELL tokenize-german-utf8 | \
8.    perl -pe 's/^(<seg .*"notation">.*<\/seg>)$/§§§$1§§§\tNN/' | \
9.    $BIN/tree-tagger lib/german.par -token -lemma -sgml -proto
             -cap-     heuristics -lex lib/aux-lex.txt | \
10.   perl -pe 's/^§§§(.*)§§§\tNN.*/$1\tXY\t<unknown>/' | \
11.   cmd/filter-german-tags.perl | \
12.   cmd/generate-xml-output.perl $f > $destination
13. done
```

Abbildung 2.1: Hauptteil des Programms process.sh von Dr. Helmut Schmid[3]

Beispiel 2.3 Dokumentausgabe nach Zeile 8 des Programms *process.sh*:

```
<sðn="Ts-210,10[3]_1"ðana="ðabnr:46ðsatznr:116"> Ich <sp/>
sagte <sp/> früher <sp/> einmal, <sp/> man <sp/> könnte <sp/>
sich <sp/> eine <sp/> <persNameðkey="Euklid"> Euklidische
</persName> <sp/> Demonstration <lb/> <sp/> auch <sp/> an <sp/>
einer <sp/> bewegten <sp/> Figur <sp/> ausgeführt <sp/> denken.
</s>
```

Weiterhin wird der Text an Punktuationszeichen und dem Element `<sp/>` gesplittet. Das Symbol „ð“ wird gleich danach getilgt. Diese zwei letzten beschriebenen Schritte des Preprocessings werden von dem Programm *filtergerman-tokin2-utf8.perl* (siehe CD im Anhang) durchgeführt, das außerdem vom Shell Skript *tokenize-german-utf8* in Zeile 7 aufgerufen wird. Auf diese Art bleiben die Tokens allein in jeder Zeile (vgl. Beispiel 2.4), damit sie vom TreeTagger gelesen werden können.

Beispiel 2.4 Gesplittetes Dokument. Symbol „ð“ wurde getilgt:

```
<s n="Ts-210,10[3]_1" ana=" abnr:46 satznr:116">
...
<sp/>
eine
<sp/>
<persName key="Euklid">
Euklidische
</persName>
<sp/>
Demonstration
```

3 Siehe CD im Anhang.

```
<lb/>
<sp/>
auch
<sp/>
an
<sp/>
einer
<sp/>
bewegten
...
.
</s>
```

Das Tagging findet in Zeile 9 statt. Die Schritte von Zeile 10 bis 12 werden Postprocessing genannt. Im Postprocessing wird mit dem Programm *generate-xml-output.perl* (siehe CD im Anhang) in Zeile 12 der Header aus der ursprünglichen NORM-Datei ins neue Dokument kopiert und das getaggte Fragment mit Hilfe von regulären Ausdrücken angehängt, um die Erzeugung des neuen Dokuments NORM-tagged.xml-Datei zu ermöglichen. Die getaggten Tokens sind jetzt in dem neuen XML Element `<w>` eingeschlossen, das die neuen Attribute `t` (für Tag) und `l` (für Lemma) enthält. Wie dieses resultierende Dokument aussieht, wurde bereits zuvor im Beispiel 2.2 gezeigt.

3 Erkennung von Personennamen: Probleme und Lösungen

3.1 Erkennung von Personennamen

Viele Wissenschaftler, die die Suchmaschine WiTTFind benutzt haben, haben mehrmals darauf hingedeutet, dass hier nur eine Minderheit der Personennamen gefunden werden, die Ludwig Wittgenstein in seinen Texten erwähnt hatte. In seinen Unterlagen erwähnt, zitiert und analysiert Wittgenstein andere Philosophen, Mathematiker, Denker, Musiker und Schriftsteller. Viele von diesen Referenzen gehen verloren, wenn unsere Suchmaschine unfähig ist, diese Personennamen zu finden.

Wie schon erwähnt wurde, setzt diese Bachelorarbeit ihren Schwerpunkt auf die Erkennung von Personennamen. Die Dateien, in denen die Suche nach Personennamen stattfindet, sind die NORM.xml-Dateien.

3.1.1 Tagging der XML-Editionstexte

Das Tagging der älteren Version der XML-Editionstexte

In der älteren Version dieser Dateien gab es keinen Hinweis auf die Erwähnungen von Personennamen, wie das Beispiel 3.1 zeigt.

Beispiel 3.1 (aus *Ms-101_OA_NORM-tagged.xml,* Ms-101,20v[2]_4):

```
Jeden Tag viel <lb/><abbr
corresp="gearbeitet">gearb.</abbr>
&amp; viel in Tolstois Erläuterung <lb/>zu den Evangelien
gelesen.
```

Der TreeTagger agiert über die NORM.xml-Dateien, um die Tokens zu taggen. Das Verhalten des POS-Taggers in Bezug auf die Personennamen in den älteren Dateien wird anhand der Beispiele 3.2 und 3.3 gezeigt.

Beispiel 3.2 (aus *Ms-104_OA_NORM-tagged.xml,* Ms-104,V[1]_1):

```
<w t="NE" l="Ludwig">Ludwig</w>
<sp/>
<w t="NE" l="Wittgenstein">Wittgenstein</w>
```

Im Beispiel 3.2 ist ein erfolgreiches Ergebnis zu sehen, im Beispiel 3.3 aber ist das Tagging gescheitert. Das passiert, wenn die Personennamen nicht in den Trainingsdaten des TreeTaggers vorkommen, was als einen Nachteil der statistischen POS-Tagger gesehen wird (vgl. Kapitel 2). Das ist zum Beispiel der Fall, wenn der Name *Tolstoi* mit dem Genitiv-Merkmal „s“ im Text auftaucht:

Beispiel 3.3 (aus *Ms-101_OA_NORM-tagged.xml,* Ms-101,20v[2]_4):

```
<w l="UNKNOWN" t="ADJA">Tolstois</w>
```

Im Beispiel 3.3 wird die Personennamen-Erwähnung nicht als solches erkannt, sondern mit dem Tag ADJA (siehe Tabelle 2.1) und dem Lemma UNKNOWN getaggt.

Das Tagging der aktualisierten Version der XML-Editionstexte

Die Editoren aus Bergen haben bei einer aktualisierten Version des Nachlasses seit März 2017 alle Personennamen mit dem neuen XML-Element <persName> ausgezeichnet. Dieses neue Element besitzt außerdem ein Attribut mit dem Namen key, dessen Wert den kompletten Namen der erwähnten Person enthält. Die aktualisierte Version des Beispiels 3.1 mit dem Element <persName> und dem Attribut key ist im Beispiel 3.4 zu sehen:

Beispiel 3.4 (aus *Ms-101_OA_NORM-tagged.xml,* Ms-101,20v[2]_4):

```
Jeden Tag viel <lb/> <abbr
corresp="gearbeitet">gearb.</abbr>
&amp; viel in
<persName key="Tolstoj,Lew">Tolstois</persName>
Erläuterung <lb/>zu den Evangelien gelesen.</s>
```

In den folgenden Beispielen 3.5 und 3.6 wird gezeigt, wie der TreeTagger nach der Aktualisierung der XML-Dateien mit den markierten Personennamen funktioniert. Dafür werden dieselben Fragmente aus den Editionstexten wie in den Beispielen 3.2 und 3.3 verwendet:

Beispiel 3.5 (aus *Ms-104_OA_NORM-tagged.xml,* Ms-104,V[1]_1):

```
<persName key="Wittgenstein, Ludwig">
<w t="NE" l="Ludwig">Ludwig</w>
<sp/>
<w t="NE" l="Wittgenstein">Wittgenstein</w>
</persName>
```

Bespiel 3.6 (aus *Ms-101_OA_NORM-tagged.xml,* Ms-101,20v[2]_4):

```
<persName key="Tolstoj, Lew">
<w l="UNKNOWN" t="ADJA">Tolstois</w>
</persName>
```

Die Performanz des TreeTaggers bei den aktualisierten Dateien demonstriert, dass der TreeTagger von denjenigen Informationen nicht unmittelbar profitiert, die die XML-Strukturen zur Verfügung stellen. Das Ergebnis des Taggings hat keine Verbesserung im Vergleich zu den älteren NORM.xml-Dateien gezeigt.

Nach dem Tagging liefert der TreeTagger das Lemma und die Wortart des getaggten Tokens, wie im Unterkapitel 2.2 bereits erwähnt wurde. Wenn der TreeTagger einen Personennamen als solchen erkennt, wird innerhalb des XML-Elements <w> (word) der richtige Personenname als Lemma und ein richtiger Tag nach der entsprechenden syntaktischen Funktion des Namens zurückgeliefert. Aufgrund des Sparse-Data-Problems, ignoriert der TreeTagger aber die Personennamen, die er nicht kennt und taggt sie nach anderen Kriterien. Bei manchen für den Tagger unbekannten Namen wird das Lemma als UNKNOWN bezeichnet. Falls der vorkommende Personenname

ähnliche morphologische Merkmale wie ein Adjektiv enthält, werden sie als ADJA bzw. ADJD getaggt. Viele Personennamen stimmen mit Substantiven überein, deswegen reagiert hier der TreeTagger anders, indem er den Tag-Wert NN in diesem Fall zurückgibt. Eine ähnliche Performanz des TreeTaggers kann beobachtet werden, wenn die Personennamen typische Merkmale von Verben und ihren unterschiedlichen morphologischen Varianten zeigen. In diesem Fall werden die Personennamen als VVFIN oder VVIMP getaggt. Wittgenstein hat viele Namen in Form von Abkürzungen notiert. An dieser Stelle hat der Tagger die einzelne Buchstaben entweder als XY oder als NN getaggt.

3.2 Generalisierende, problematische Fälle beim Tagging

Sieben unterschiedliche, generalisierende Fälle werden in der Gesamtheit des Nachlasses identifiziert, in denen der TreeTagger die Personennamen falsch erkennt, was einen Informationsverlust für die Suchmaschine WiTTFind bedeutet. Im Nachstehenden werden diese Fälle aufgelistet.

3.2.1 Korrekter Tag, unbekanntes Lemma

In diesem Fall weist der Tagger zwar ein korrektes Tag aber ein unbekanntes Lemma zu, wie im Beispiel 3.7 zu sehen ist.

Beispiel 3.7 (aus *Ms-115_OA_NORM-tagged.xml,* Ms-115,81[2]_1):
NORM.xml: `<persName key="Nicod, Jean"> … </persName>`
Tagger: `<w t="NE" l="UNKNOWN">Nicod</w>`

3.2.2 Adjektiv

In den Beispielen 3.7 und 3.8 wird das Lemma ebenfalls nicht erkannt, aber auch das Tag wird falsch zugewiesen.

a) t="ADJA" l="UNKNOWN"

Beispiel 3.7 (aus *Ms-101_OA_NORM-tagged.xml,* Ms-101,20v[2]_4):
NORM.xml: `<persName key="Tolstoj, Lew"> … </persName>`
Tagger: `<w t="ADJA" l="UNKNOWN">Tolstois</w>`

b) t="ADJD" l="UNKNOWN"

Beispiel 3.8 (aus *Ms-114_OA_NORM-tagged.xml,* Ms-114,29v[2]_1):
NORM.xml: `<persName key="Ursell, Harold Douglas"> … </persName>`
Tagger: `<w t="ADJD“ l="UNKNOWN">Ursell</w>`

3.2.3 Abkürzung

Abkürzungen werden entweder als Nomen mit nur einem Buchstaben (siehe Beispiel 3.9) oder als Abkürzung mit unbekannten Lemma (siehe Beispiel 3.10) erkannt.

a) `t="NN" l=` *Einzelbuchstabe*

Beispiel 3.9 (aus *Ts-241a_OA_NORM-tagged.xml,* Ts-241a,32[1]_1):
NORM.xml: `<persName key="James, William"> … </persName>`
Tagger:
```
<seg type="p" part="N">
    <w t="NN" l="W">W</w>
    <w t="$." l=".">.</w>
    </seg>
    <sp/>
    <w t="NE" l="James">James</w>
```

b) `t="XY l="UNKNOWN"`

Beispiel 3.10 (aus *Ms-115_OA_NORM-tagged.xml,* Ms-115,217[2]_8):
NORM.xml: `<persName key="James, William"> ... </persName>`
Tagger:
```
<w t="XY" l="UNKNOWN">W</w>
    <w t="$." l=".">.</w>
    <w t="NE" l="James">James</w>
```

3.2.4 Tag Nomen, Lemma unbekannt

In Beispiel 3.11 ist zu sehen, wie der Tagger den Tag Nomen und ein unbekanntes Lemma zuweist.

Beispiel 3.11 (aus *Ms-154_OA_NORM-tagged.xml,* Ms-154,15v[3]et16r[1]_3):
NORM.xml: `<persName key="Sraffa, Piero"> ... </persName>`
Tagger: `<w t="NN" l="UNKNOWN">Sraffa</w>`

3.2.5 Verb

Der Tagger weist den Namen die Wortart Verb zu, wie die Beispiele 3.12 und 3.13 zeigen.

a) `t="VVFIN" l="UNKNOWN"`

Beispiel 3.12 (aus *Ms-101_OA_NORM-tagged.xml,* Ms-101,20v[4]_4):
NORM.xml: `<persName key="Tolstoj, Lew"> … </persName>`
Tagger: `<w t="VVFIN" l="UNKNOWN">Tolstois</w>`

b) `t="VVIMP l=` *Name oder Nachname, der mit einem Verb übereinstimmt*

Beispiel 3.13 (aus *Ms-114_OA_NORM-tagged.xml,* Ms-114,6r[2]et6v[1]_3):
NORM.xml: `<persName key="Mach, Ernst"> … </persName>`
Tagger: `<w t="VVIMP" l="machen">Mach</w>`

3.2.6 Name, der mit einem Nomen übereinstimmt

Wenn ein Name einem Nomen sehr ähnlich oder gleich ist, wird dieser als Nomen getaggt, wie im Beispiel 3.14 zu finden ist.

Beispiel 3.14 (aus *Ms-154_OA_NORM-tagged.xml,* Ms-154,15v[3]et16r[1]_3):
NORM.xml: `<persName key="Hertz, Heinrich"> … </persName>`
Tagger: `<w t="NN" l="Hertz">Hertz</w>`

3.2.7 Komposita

Ein Name wird als Teil eines Kompositums erkannt, wenn auf den Namen ein Bindestrich folgt. Das Lemma bleibt dabei unbekannt (siehe Beispiel 3.15).

Beispiel 3.15 (aus *Ts-213_OA_NORM-tagged.xml,* Ts-213,IVr[17]_1):
NORM.xml: `<persName key="Frege, Gottlob"> … </persName>`
Tagger: `<w t="TRUNC" l="UNKNOWN">Frege-</w>`

3.3 Lösungsorientierte Methoden

Das Ziel von diesem Kapitel, und das Hauptziel dieser Bachelorarbeit ist es, Ideen zu finden, damit die Ausgabe des POS-Taggers nützliche Auskünfte über die Personennamen liefern kann, damit die Suche nach Personennamen in der WiTTFind App optimiert werden kann. Im Folgenden werden zwei Methoden vorgeschlagen, die beschreiben, wie dieses Ziel erreicht werden kann.

3.3.1 Erstellung einer Personennamensliste mit XML-Parsing

Die Python Bibliothek bietet einige nützliche Schnittstellen, die das Parsen von XML-Dateien unterstützen. Sie sind „Simple API for XML“ (SAX), „Document Object Model API" (DOM API) und „ElementTree XML API“ (etree bzw. ET API).

Für die Zwecke dieser Arbeit wird der ElementTree verwendet, denn das `xml.etree.ElementTree` Modul von Python3 implementiert einfach und effizient diese Programmierschnittstelle (API).

Die geparsten XML-Dokumente sind im Speicher mittels ElementTree- und Element-Objekten abgebildet, die gleichzeitig in einer Baumstruktur angebunden sind und zudem dieselbe Struktur aufweisen, wie die Knoten in dem XML-Dokument.

Die ET-Methode zum Parsen heißt `parse()`. Wenn ein komplettes Dokument mit dieser Methode geparst wird, wird eine ElementTree-Instanz zurückgeliefert. Dadurch bekommt das Input Dokument (NORM.xml-Datei) eine Baumstruktur, deren Knoten in situ mittels des Parsers durchsucht und manipuliert werden können. Die Python Standard-Funktion `open()` wird verwendet, um die XML-Datei zu öffnen und ein neues File-Objekt zu erhalten. Diese Instanz der Klasse File wird dann als Parameter für die Methode `parse()` verwendet.

Der Quelltext des Programms *child_in_Norm.py* aus Abbildung 3.1 zeigt die eben beschriebene Prozedur:

```
1. # Identiefizierung von Elementen und ihre Attribute in
2. # einem XML-Datei
3. # Autorin: Faridis Alberteris Azar
4. # Aufruf: python3 <NORM.xml>
5. from xml.etree import ElementTree
6. from xml.etree.ElementTree import iterparse
7. import sys
8. output = open('child_in_Norm.txt','wt')
9. with open(sys.argv[1],'rt') as f:
10.        tree = ElementTree.parse(f)
11.        for tochterknote in tree.iter():
12.            print(tochterknote.tag,'=>',\
13.            tochterknote.attrib,file=output)
```

Abbildung 3.1: Quelltext des Programms child_in_Norm.py.

Das Ziel ist es nun, das Element der XML-Datei zu identifizieren, in dem die Personennamen zu finden sind. Die ET-Methode `iter()` in Zeile 11 erlaubt es, über die ganze ElementTree-Instanz zu iterieren und erlaubt einen Zugang zu den Tochterknoten. Die Anweisungen `tochterknote.tag` in Zeile 12 und `tochterknote.attrib` in Zeile 13 ermöglichen den Zugriff auf die Tochterknoten und ihre Attribute, die die Datenstruktur eines Dictionarys aufweisen.

Das Attribut key vom Element `<persName>`, das den ganzen Namen als Wert aufweist, wird mit Hilfe von der `get()`-Funktion herausgezogen: `tochterknote.get('key')`. Die eigentliche Erwähnung des entsprechenden Personennamens, die Ludwig Wittgenstein in seinem Text gemacht hat, findet man versteckt als Text bzw. als Inhalt der Tochterknoten. Dieser Inhalt ist mit Hilfe des folgenden Ausdrucks zu extrahieren: `tochterknote.itertext()`. Nach der Implementierung der Methoden `get()` und `itertext()` im Programm *namenListe.py* (siehe Anhang) erreicht man das Ziel, die Personennamen zu identifizieren. Damit ist man bereit, eine Liste zu erzeugen, die die von dem österreichischen Philosophen erwähnten Personennamen als Token-Namen und die entsprechenden vollständigen Namen als Lemma-Namen enthält. Nach jeder Auflistung pro Dokument wird auch die Gesamtsumme der Personennamen angegeben.

Diese große Liste (vollständige Namensliste ist auf der beigelegten CD vorhanden) ist ein Beitrag, den das WiTTFind zur Verfügung stellen kann, als eine Art Archiv. Forscher, Studierende, Homepage-Entwickler und andere Interessierte haben einen raschen Zugriff auf diese tabellarischen Daten, falls sie schnell nachschlagen möchten, wen und in welchem Dokument Ludwig Wittgenstein zitiert bzw. erwähnt hat.

3.3.2 Semantische Suche in WiTTFind verbessern

Die WiTTFind Homepage bietet zwei Möglichkeiten, eine Suche nach einem Begriff zu unternehmen. Eine heißt „Regelbasiertes Finden“ und die andere „Semantisches Finden“. Das Regelbasierte Finden verlangt einen Begriff und sucht im Hintergrund die

entsprechenden Wortarten und die semantische Kategorie des Begriffes nach einem regelbasierten Muster dessen Tag zu dem STTS (siehe Tabelle 2.1) gehört.

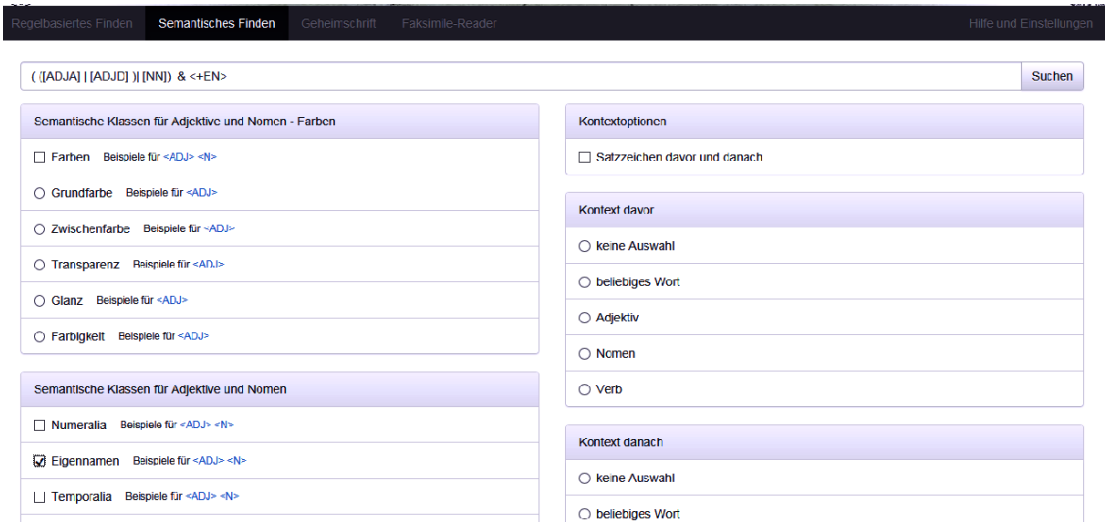

Abbildung 3.2: Semantische Suche nach Personennamen in WiTTFind

Das Semantische Finden (siehe Abbildung 3.2) stellt bereits dieses Muster zur Verfügung, abhängig von dem Begriff, den man findet möchte. In beiden Fällen lautet das Suchmuster für einen Personennamen:

(([ADJA] | [ADJD])| [NN]) & <+EN>

Nach diesem Muster werden Personennamen erkannt, die in getaggten XML-Dokumenten mit den Tags ADJA, ADJD, NN markiert sind und die zusätzliche Kategorie EN besitzen. Dieses Pattern hat viele Vorteile, denn es werden auch solche Personennamen gefunden, die als Adjektiv oder als Nomen getaggt werden.

Wenn man aber die sieben aufgelisteten Fälle betrachtet, merkt man, dass auch Personennamen, die vom Tagger als Verben oder Abkürzungen notiert wurden, nicht im WiTTFind gefunden werden, denn es existiert keine Regel dafür. Das heißt, die bestehende Regel (([ADJA] | [ADJD])| [NN]) & <+EN> ist unzureichend.

Die Abbildungen 3.3 bis 3.6 zeigen einige Personennamen, die mit dem regelbasierten Finden und dem semantischen Finden im Dokument *Ts-213* gefunden werden.

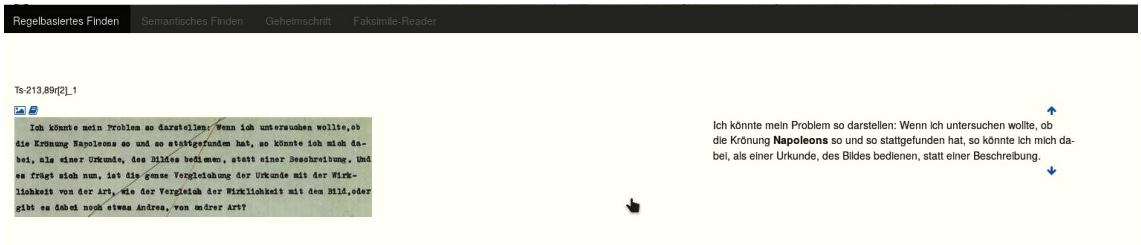

Abbildung 3.3: Gefundene Namen beim Semantischen Finden in WiTTFind: Napoleons

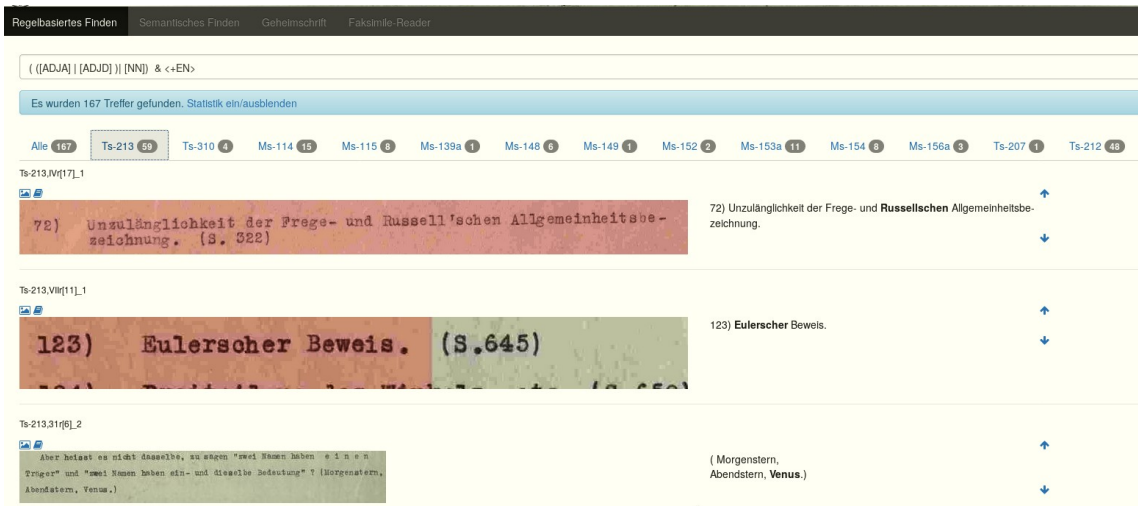

Abbildung 3.4: Gefundene Namen beim Semantischen Finden in WiTTFind: Russellschen, Eulerscher und Venus

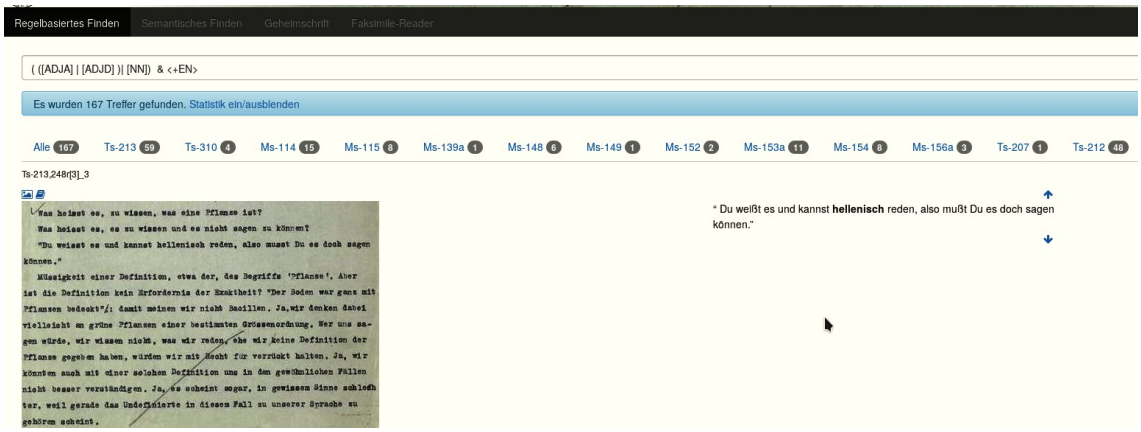

Abbildung 3.5: Gefundene Namen beim Semantischen Finden in WiTTFind: hellenisch

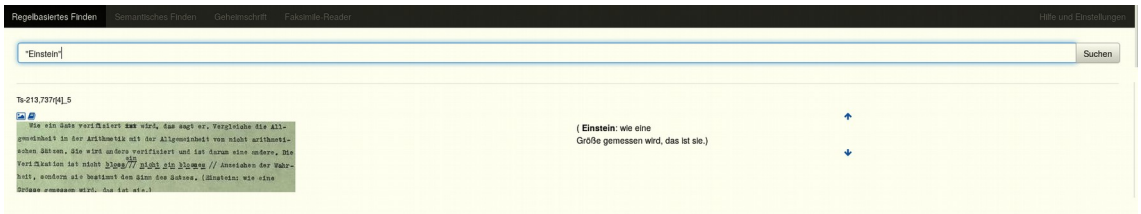

Abbildung 3.6: Gefundene Namen beim Semantischen Finden in WiTTFind: Einstein

Im Nachstehenden wird gezeigt, wie die Treffer von den Abbildungen in den getaggten Dateien repräsentiert sind.

Zur Abbildung 3.3 (aus *Ts-213_OA_NORM.xml,* Ts-213,89r[2]_1):

```
<persName key="Napoleon Bonaparte">
<w t="NN" l="Napoleon">Napoleons</w>
```

```
</persName>
```

Zur Abbildung 3.4 (aus *Ts-213_OA_NORM.xml*, Ts-213,IVr[17]_1):

```
<persName key="Russell, Bertrand">
<w t="ADJA" l="Russellsche">Russellschen</w>
</persName>
```

Zur Abbildung 3.4 (aus *Ts-213_OA_NORM.xml,* Ts-213,31r[6]_2):

```
<w t="NN" l="Morgenstern">Morgenstern</w>
<w t="NN" l="Abendstern">Abendstern</w>
<w t="NN" l="perVenus">Venus</w>
```

Zur Abbildung 3.5 (aus *Ts-213_OA_NORM.xml,* Ts-213,248r[3]_3):

```
<w t="PPER" l="du">Du</w>
<w t="VVFIN" l="weißen">weißt</w>
<w t="PPER" l="es">es</w>
<w t="KON" l="und">und</w>
<w t="VMFIN" l="können">kannst</w>
<w t="ADJD" l="hellenisch">hellenisch</w>
<w t="VVINF" l="reden">reden</w>
```

Zur Abbildung 3.6 (aus *Ts-213_OA_NORM.xml,* Ts-213,737r[4]_5):

```
<persName key="Einstein, Albert">
<w t="NE" l="Einstein">Einstein</w>
</persName>
```

Nach der Betrachtung dieser Beispiele ist zu erkennen, dass der Name *Einstein*, und sogar die adjektivische Verwendung *Russellsche* (von Namen Russell) richtig getaggt werden und daher auch von WiTTFind richtig getroffen werden können. Die Variante *Napoleos* wurde von TreeTagger falsch erkannt, aber trotzdem von der WiTTFind App gefunden, denn NN ist eine der morphologischen Kategorien, die laut Suchmuster gesucht wird.

Aber wenn die Treffer genauer betrachtet werden, ist ein Problem zu erkennen. Die Webseite findet auch Wörter wie *Venus* und *hellenisch*. Sie könnten auch tatsächlich in manchen Kontexten als Personennamen oder ihrer Varianten fungieren. Eine Frau oder eine Göttin kann Venus heißen, hellenisch könnte eine adjektivische Verwendung des weiblichen Namens Hellen sein. Beides ist hier jedoch nicht der Fall. Venus bezieht sich in diesem konkreten Fall auf einen Sternnamen und hellenisch auf die griechische Sprache. Daraus folgt, dass auch für eine Optimierung der semantischen Suche im Wittgenstein Nachlass, eine Disambiguierung der Wortbedeutungen nötig ist.

Ein weiteres Problem besteht darin, dass andere Personennamen, die vom TreeTagger falsch als Verben getaggt wurden, nach der Suche nicht zu treffen sind, denn das Muster enthält keine Angabe über VVIMP oder VVFIN. Ein Exempel dafür ist der Name *Mach* aus dem Beispiel 3.13, denn WiTTFind findet im Moment im Dokument *Ms-114_OA_NORM-taggerd.xml* diesen Namen nicht, weil er als VVIMP getaggt ist.

Erste Maßnahme der Methode 2: Erweiterung des XML-Attributs `t`

Eine mögliche erste Lösung, die hier vorgeschlagen werden könnte, wäre einfach die Suchmuster in WiTTFind zu erweitern, damit die Verben VVFIN bzw. VVIMP oder die unterschiedlichen Abkürzungen getroffen werden können. Aber diese angebliche, evidente Lösung führt zu einer neuen Problematik. Die Statistik bezüglich der Personennamen wäre auch falsch, denn weitere Verben bzw. Abkürzungen würden auch getroffen, die keine Personennamen sind.

In dieser Hinsicht wird hier als Lösung vorgeschlagen, eine neue morphologische Kategorie mit dem Namen „persName“ zu erstellen, die zu den vom TreeTagger benutzten Tags hinzugefügt wird. Diese neue Wortart soll zu einer Disambiguierung der Bedeutung beitragen.

Für jedes Element `<w>` in getaggten Dateien, die unter dem Element `<persName>` eingeschlossen sind, wird das Attribut `t` erweitert, indem zu der bereits vorhandenen morphologischen Kategorie auch die alternative Kategorie persName hinzugefügt wird. Zu diesem Zweck wird das Python Skript *addKeyundLemmaName.py* (siehe Anhang) erzeugt. Dadurch wird auf das gewünschte Element `<w>` zugegriffen. Im Nachhinein wird das Attribut geändert und eine neue getaggte XML-Datei zurückgeliefert.

Damit das Programm *addKeyundLemmaName.py* die richtige XML-Datei liefert, muss der Namensraum unbedingt vor der `print`-Anweisung festgelegt werden. Der Namensraum, so Erik T. Ray [8, S. 47-48], sei eine Gruppe von Element- und Attribut-Namen (siehe Unterkapitel 1.1). Im Falle von den getaggten NORM.xml-Dateien, in denen die vorgeschlagene Änderung stattfindet, befinden sich die Elemente und Attribute innerhalb des Namensraums `{http://www.tei-c.org/ns/1.0}` und immer wenn man sich auf diese Element- bzw. Attribut-Namen bezieht, muss man diesen Namensraum als Präfix anhängen.

Wenn der Namensraum nicht spezifiziert wird, könnte ein falsches Dokument mit solchen Elementen und Subelementen zurückgeliefert werden:

```
...
<ns0:persName key="Wittgenstein, Leopoldine">
<ns0:w l="UNKNOWN" t="NE|persName">Poldy</ns0:w>
<ns0:sp/>
<ns0:w l="Wittgenstein" t="NE|persName">Wittgenstein</ns0:w>
</ns0:persName>
<ns0:lb rend="hl"/>
...
```

Der Python Parser etree besitzt eine Methode namens `register_namespace()`, mit der der Namespace registriert werden kann. Damit wird eine entsprechende richtig getaggte Datei nach der vorgeschlagenen Änderung erzeugt. Diese Funktion ist in Zeile 74 des Programms *addKeyundLemmaName.py* zu finden. Mit Hilfe von Xpath-Ausdrücken[4], die selbstverständlich den Namensraum enthalten müssen, greift das Skript auf die jeweiligen Elemente zu und ändert wie gewünscht die Werte des Attributes `t` gemäß der entsprechenden aufgelisteten Fälle.

4 Ausdrücke, die den Pfad zum XML-Element wiedergeben

Somit erfolgt die erzielte Erweiterung bei den Tags, die sich unter dem Element `<w>` befinden, wenn das Element `<w>` in Element `<persName>` enthalten ist.

Zweite Maßnahme der Methode 2: Erweiterung des TreeTagger-Lexikons

Bei den TreeTagger-Erklärungen[5] von Dr. Schmid ist zu lesen, dass das TreeTagger-Korpus erweitert werden kann, indem neu gefundene Wörter zu der in der TreeTagger-Lib vorhandenen Datei *german-lexikon.txt* hinzugefügt werden. Da Dr. Schmid den Tagger für das WiTTFind angepasst hat, dient hier zur Erweiterung des Lexikons die andere Datei *aux-lex.txt*. Mit Hilfe von dem Shell-Skript *nameLemmaMaker.sh* (siehe CD) und dem Python-Skript *namenLemmaListe.py* (siehe CD) wurde eine neue Namensliste *namensLemmaListe.txt* (siehe CD) erzeugt, mit folgendem einfachen Format: erwähnter Personenname, ein Tab, Vollname (Lemma-Name). Dieses Format ermöglicht einen einfachen Vergleich mit der Datei *aux-lex.txt* (siehe CD), in der mittels des Python Skripts *erweiterelexikon.py* (siehe Anhang) die neu gefundenen Namen hinzugefügt wurden. Die neuen Personennamen wurden zu dem von Dr. Schmid benötigten Format addiert: Name-Erwähnung, ein Tab, Tag (hier NE), Leerzeichen und Lemma (hier den ganzen Namen). Ein Beispiel von einem neuen Eintrag ist folgendes:

Picassos NE Picasso, Pablo

Diese Maßnahme hat die Anzahl der „`UNKNOWN`"-Lemma-Attribute reduziert, aber nicht komplett beseitigt. Das Problem hierbei besteht darin, dass der TreeTagger die XML-Dateien an Punktuationszeichen für die Tokenisierung splittet (siehe Unterkapitel 2.2). Wenn Ludwig Wittgenstein Namen wie *L.W.* oder *W. James* erwähnt hat, werden diese Namen in die einzelnen Buchstaben durch den TreeTagger gesplittet und nicht als Teil eines Personennamens erkannt. Dabei bleibt das „`UNKNOWN`"-Lemma-Attribut auch vorhanden. Im Folgenden wird einer der sieben problematischen Fälle von Unterkapitel 3.2 gezeigt, auf den die Erweiterung des TreeTagger-Lexikons keinen Einfluss ausgeübt hat. Es handelt sich um den Fall 3.2.3, bei dem Wittgenstein Abkürzungen für die Personennamen verwendet hat und das Lemma im Falle vom Beispiel 3.10 als `UNKNOWN` vom Tagger wiedergegeben wird.

b) `t="XY l="UNKNOWN"`

Beispiel 3.16 (aus *Ms-115_OA_NORM-tagged.xml,* Ms-115,217[2]_8):
NORM.xml: `<persName key="James, William"> ... </persName>`
Tagger vor der Erweiterung des Lexikons und Änderung des Tags:

```
<w t="XY" l="UNKNOWN">W</w>
<w t="$." l=".">.</w>
<w t="NE" l="James">James</w>
```

Tagger nach der Erweiterung des Lexikons und des Tags:

```
<w l="UNKNOWN" t="XY|persName">W</w>
<w l="." t="$.">.</w>
<w l="James,_William" t="NE|persName">James</w>
```

Dritte Maßnahme der Methode 2: Korrektur der übrigen „`UNKNOWN`"-Lemma-Attribute

5 README.script von Dr. Schmid ist auf der beigelegten CD zu finden.

Um die übrigen „UNKNOWN“-Lemma-Attribute zu korrigieren, wird das bereits existierende Programm *addKeyundLemmaName.py* erweitert. Es wird bei jedem Fall überprüft, ob das „UNKNOWN“-Lemma-Attribut noch erscheint. Wenn es der Fall ist, wird dem Attribut `l` der Wert vom Attribut `key` zugewiesen, das die Information mit dem komplett erwähnten Personennamen beinhaltet.

Das Programm *addKeyundLemmaName.py* wird zum Tagger-Vorgang inkludiert. Es wird ins selbe Verzeichnis wie die anderen TreeTagger-Skripte kopiert und auch im Programm *process.sh* hinzugefügt[6]. Damit wird das Programm *addKeyundLemmaName.py* vom Skript *angepasstes_prozess.sh* nach dem Tagging aufgerufen, die getaggte Datei verändert und wie gewünscht ausgegeben.

Vierte Maßnahme der Methode 2: Erweiterung des XML-Elements `<w>`

Als zusätzliche Maßnahme wird das Element `<w>` erweitert, in dem das Attribut `key` mit dem vollständigen Personennamen aus dem Element `<persName>` neben den Attributen `t` und `l` hinzugefügt wird. Damit enthält das Element `<w>` alle wichtigen Informationen über die Personennamen, obwohl für den POS-Tagger das Lemma (vollständiger Personenname) immer noch unbekannt ist. Dieser Schritt wird auch innerhalb des Programms *addKeyundLemmaName.py* durchgeführt.

Fünfte Maßnahme der Methode 2: Erweiterung des Suchmusters im WiTTFind

Mit dem Ziel, die neue morphologische Kategorie persName in den ganzen Ablauf von WiTTFind zu integrieren und das vorhandene Suchmuster zu verfeinern, wird zusätzlich vorgeschlagen, die neue morphologische Kategorie persName in das Suchmuster hinzuzufügen. Das neue Suchmuster sollte nach diesem Vorschlag wie folgt aussehen:

(([ADJA] | [ADJD])| [NN]) & <+EN> & **<+persName>**

Die Erweiterung des Suchmusters mittels dieser neuen Kategorie sollte den Suchvorgang verfeinern und vermeiden, dass andere Wortformen, die keine Personennamen oder seine Ableitung sind, als solche gefunden werden.

3.4 Fälle nach der Anwendung der Methoden

Die sieben Fälle, bei denen der TreeTagger während des Taggings gescheitert ist und die im Unterkapitel 3.2 aufgelistet wurden, sehen nach den durchgeführten Erweiterungen und Änderungen beim Postprocessing in den jeweiligen getaggten NORM.xml-Dateien wie folgt aus:

3.4.1 Korrekter Tag, unbekanntes Lemma

Beispiel 3.17 (aus *Ms-115_OA_NORM-tagged.xml,* Ms-115,81[2]_1):
NORM.xml: `<persName key="Nicod, Jean"> … </persName>`
Tagger vor den Erweiterungen: `<w t="NE" l="UNKNOWN">Nicod</w>`
Tagger nach den Erweiterungen:

6 Verändertes Programm *prozess.sh* ist als *angepasstes_prozess.sh* im Anhang zu finden.

```
<w key="Nicod, Jean" l="Nicod,_Jean" t="NE|persName">
     Nicod
</w>
```

3.4.2 Adjektiv

a) t="ADJA" l="UNKNOWN"

Beispiel 3.18 (aus *Ms-101_OA_NORM-tagged.xml,* Ms-101,20v[2]_4):
NORM.xml: `<persName key="Tolstoj, Lew"> … </persName>`
Tagger vor den Erweiterungen: `<w t="ADJA"l="UNKNOWN">Tolstois</w>`
Tagger nach den Erweiterungen:

```
<w key="Tolstoj, Lew" l="Tolstoj,_Lew"
     t="NE|persName">Tolstois
</w>
```

b) t="ADJD" l="UNKNOWN"

Beispiel 3.19 (aus *Ms-114_OA_NORM-tagged.xml,* Ms-114,29v[2]_1):
NORM.xml: `<persName key="Ursell, Harold Douglas"> … </persName>`
Tagger vor den Erweiterungen: `<w t="ADJD“ l="UNKNOWN">Ursell</w>`
Tagger nach den Erweiterungen:

```
<w key="Ursell, Harold Douglas" l="Ursell, Harold
     Douglas" t="ADJD|persName">Ursell
</w>
```

3.4.3 Abkürzung

a) t="NN" l= *Einzelbuchstabe*

Beispiel 3.20 (aus *Ts-241a_OA_NORM-tagged.xml,* Ts-241a,32[1]_1):
NORM.xml: `<persName key="James, William"> … </persName>`
Tagger vor den Erweiterungen:

```
<w t="NN" l="W">W</w>
<w t="$." l=".">.</w>
<w t="NE" l="James">James</w>
```

Tagger nach den Erweiterungen:

```
<w key="James, William" l="W" t="NN|persName">W</w>
<w key="James, William" l="." t="$.">.</w>
<w key="James, William" l="James,_William"
     t="NE|persName">James
</w>
```

b) t="XY l="UNKNOWN"

Beispiel 3.21 (aus *Ms-115_OA_NORM-tagged.xml,* Ms-115,217[2]_8):
NORM.xml: `<persName key="James, William"> ... </persName>`
Tagger vor den Erweiterungen:

```
<w t="XY" l="UNKNOWN">W</w>
<w t="$." l=".">.</w>
```

```
                        <w t="NE" l="James">James</w>
```

Tagger nach den Erweiterungen:

```
<w key="James, William" l="James, William"
     t="XY|persName">W
</w>
<w key="James, William" l="." t="$.">.</w>
<w key="James, William" l="James,_William"
     t="NE|persName">James
</w>
```

3.4.4 Tag Nomen, Lemma unbekannt

Beispiel 3.22 (aus *Ms-154_OA_NORM-tagged.xml,* Ms-154,15v[3]et16r[1]_3):
NORM.xml: `<persName key="Sraffa, Piero"> ... </persName>`
Tagger vor den Erweiterungen: `<w t="NN" l="UNKNOWN">Sraffa</w>`
Tagger nach den Erweiterungen:

```
<w key="Sraffa, Piero" l="Sraffa,_Piero"
     t="NE|persName">Sraffa
</w>
```

3.4.5 Verb

a) `t="VVFIN" l="UNKNOWN"`

Beispiel 3.23 (aus *Ms-101_OA_NORM-tagged.xml,* Ms-101,20v[4]_4):
NORM.xml: `<persName key="Tolstoj, Lew"> … </persName>`
Tagger vor den Erweiterungen: `<w t="VVFI l="UNKNOWN">Tolstois</w>`
Tagger nach den Erweiterungen:

```
<w key="Tolstoj, Lew" l="Tolstoj,_Lew"
     t="NE|persName">Tolstois
</w>
```

b) `t="VVIMP l=` *Name oder Nachname, der mit einem Verb übereinstimmt*

Beispiel 3.24 (aus *Ms-114_OA_NORM-tagged.xml,* Ms-114,6r[2]et6v[1]_3):
NORM.xml: `<persName key="Mach, Ernst"> … </persName>`
Tagger vor den Erweiterungen: `<w t="VVIMP" l="machen">Mach</w>`
Tagger nach den Erweiterungen:

```
<w key="Mach, Ernst" l="Mach,_Ernst"
     t="NE|persName">Mach
</w>
```

3.4.6 Name, der mit einem Nomen übereinstimmt

Beispiel 3.25 (aus *Ms-154_OA_NORM-tagged.xml,* Ms-154,15v[3]et16r[1]_3):
NORM.xml: `<persName key="Hertz, Heinrich"> … </persName>`
Tagger vor den Erweiterungen: `<w t="NN" l="Hertz">Hertz</w>`

Tagger nach den Erweiterungen:

```
<w key="Hertz, Heinrich" l="Hertz"
     t="NN|persName">Hertz
</w>
```

3.4.7 Komposita

Beispiel 3.26 (aus *Ts-213_OA_NORM-tagged.xml,* Ts-213,IVr[17]_1):
NORM.xml: `<persName key="Frege, Gottlob"> … </persName>`
Tagger vor den Erweiterungen: `<w t="TRUNC" l="UNKNOWN">Frege-</w>`
Tagger nach den Erweiterungen:

```
<w key="Frege, Gottlob" l="Frege, Gottlob"
     t="TRUNC|persName">Frege-
</w>
```

3.5 Evaluierung

Für die Bewertung der in diesem Kapitel empfohlenen Vorschläge wurde eine dreispaltige Tabelle 3.1 (S. 62) mit dem Namen *Statistik-Vergleich von Personennamen* erzeugt. In der ersten Spalte befinden sich die 20 Dateinamen, die die 5.000 Seiten des Wittgenstein Nachlasses enthalten, mit denen WiTTFind momentan arbeitet. In der zweiten Spalte ist die Gesamtzahl der Personennamen zu beobachten, die die WiTTFind App in den jeweiligen Dokumenten bei der aktuellen Suche findet. Die dritte Spalte repräsentiert die Gesamtzahl der Personennamen, die von dem in diesem Kapitel empfohlenen System gefunden werden sollen.

Die Anzahl der Eigennamen der dritten Spalte wurden mit dem Programm *namenListe.py* berechnet und die Anzahl der zweiten Spalte wurde aus der Statistik (vgl. Abbildung 3.7 auf Seite 61) herausgezogen.

Diese Tabelle stellt deutlich dar, wie viele Information bzw. Personennamen mit dem Suchvorgang ohne den Optimierungen des WiTTFinds nach wie vor verloren gehen. Während durch die WiTTFind Suche insgesamt 168 Personennamen in 13 Dokumenten getroffen werden, unter denen einige gar keine Personennamen sind, werden nach den vorgeschlagenen Optimierungsmaßnahmen insgesamt 833 Personennamen in allen 20 Dateien gefunden. Die totalen Summen zeigen, dass mit dem empfohlenen System die Anzahl der getroffen Personennamen erheblich steigt.

Die 5.000 XML-annotierten Dateien des Wittgenstein Nachlasses, das Suchmuster sowie die Anpassung des TreeTaggers sollten in der Weise aktualisiert werden, dass die hier empfohlenen Maßnahmen angewendet werden können.

4 Transkriptionsfehler vs. Editionsprobleme

Wie im Kapitel 1 erklärt wurde, werden die XML- bzw. XSLT-Dateien direkt vom WAB geliefert. Die Konvertierung der XML-Dateien erfolgt auf der Basis der WiTTFind Bedürfnisse. Daraus ergeben sich hier zwei unterschiedliche Perspektiven. Eine ist durch die Editoren aus Bergen vertreten, die die XSLT Programme verfassen und versuchen so gut wie möglich ihre komplexen ursprünglichen XML-Dateien anzupassen und Informationen in XML-Elemente bzw. -Attribute umzuwandeln, die von den originalen Dateien nicht gelöscht werden können, die aber für die WiTTFind Entwickler irrelevant sind. Die zweite Perspektive ist von den WiTTFind Entwicklern repräsentiert. Ausgehend von den XSLT Programmen erzeugen sie nach ihren Bedürfnissen die XML-Dateien, die im WiTTFind verwendet werden: OA.xml, NORM.xml und DIPLO.xml (CISWAB).

Zwischen den beiden Perspektiven befinden sich die Studenten oder andere Mitarbeiter, die die CISWAB-Dateien für andere Zwecke benutzen, z.B. Lexikon aufbauen, Tagging, Frequenzlisten erzeugen, semantische Suche, usw. In dieser Zwischenposition trifft man auf Hindernisse, die für beide obengenannten Perspektiven irrelevant sind, aber die an dieser Zwischenstelle das Umgehen mit den resultierenden CISWAB-Dateien erheblich bremsen. Diese Bachelorarbeit befasst sich nur mit den NORM.xml-Dateien. Diese Hindernisse sind auch in NORM.xml zu finden. Sie werden in zwei Gruppen geteilt: Transkriptionsfehler und Editionsprobleme.

4.1 Transkriptionsfehler

Die Transkriptionsfehler sind diejenigen Fehler, die bei der Konvertierung der Originaldateien aus Bergen ins NORM.xml unterlaufen sind. An manchen Abschnitten der resultierenden Dateien sind einige XML-Elemente falsch platziert oder mit fehlenden Attributen zu finden. Das hindert den TreeTagger daran, eine richtige Tokenisierung durchzuführen. Das bringt mit sich, dass mehrere Wörter in den getaggten Dateien inkorrekt zerlegt werden. Nach einer schlechten Tokenisierung hat der Python Parser keinen richtigen Zugriff auf den Inhalt der XML-Elemente. Zwei betroffene Elemente sind `<emph>`, das die Hervorhebung bzw. Betonung repräsentiert und `<lb>`, das Element, das für den Zeilenumbruch reserviert ist.

4.1.1 Platzierung der XML-Elemente `<emph>` und `<lb>`

Richtige Platzierung des XML-Elements `<emph>`

Das Element `<emph>` wird im Beispiel 4.1 richtig platziert und bedeutet kein Hindernis für die Tokeniesierung der Editionstexte.

Beispiel 4.1 (aus *Ms-121_OA_NORM.xml,* Ms-121,37r[4]et37v[1]_2):

```
<emph rend="us1">Entwickelungen</emph>
```

Richtige Platzierung des XML-Elements `<lb>`

Das Element `<lb>` wird im Beispiel 4.2 richtig platziert und bedeutet kein Hindernis für die Tokeniesierung der Editionstexte.

Beispiel 4.2 (aus *Ms-101_OA_NORM.xml,* Ms-101,23r[5]_2):

```
Immer die <lb/> uralten Fragen!
```

Der Zeilenumbruch kann auch ein Wort zerlegen. In dem Fall des Beispiels 4.3 besitzt das Element `<lb>` das Attribut `rend="shyphen"`:

Beispiel 4.3 (aus *Ms-101_OA_NORM.xml,* Ms-101,24r[2]et25r[1]_1):

```
Fun<lb rend="shyphen"/>ktion
```

4.1.2 Transkriptionsfehler bei den XML-Elementen `<emph>` und `<lb>`

Transkriptionsfehler beim XML-Element `<emph>`

Eine verkehrte Platzierung des Elements `<emph>` verursacht eine falsche Tokenisierung der Editionstexte und damit ein irrtümliches Tagging. In dem Beispiel 4.4 wird dargestellt, wie das Element `<emph>` in den NORM.xml-Dateien falsch positioniert ist und wie die resultierende, getaggte Datei aussieht. Nach dem Beispiel wird eine richtige Position für das Element `<emph>` vorgeschlagen, um diesen Fehler zu vermeiden.

Beispiel 4.4 (aus *Ms-122_OA_NORM.xml,* Ms-122,53r[3]et53v[1]_1):

```
<emph rend="uwl">im Bewe</emph>is
```

Das Wort *Beweis* wird falsch von Tagger getaggt, deswegen findet man es in den entsprechenden getaggten NORM.xml wie folgt:

Tagger:

```
<w l="in" t="APPRART">im</w>
<sp/>
<w l="UNKNOWN" t="NN">Bewe</w>
</emph>
<w l="is" t="FM">is</w>
```

Eine richtige Platzierung in *Ms-122_OA_NORM.xml* wäre:

```
<emph rend="uwl">im Beweis</emph>
```

Transkriptionsfehler beim XML-Element `<lb>`

Manchmal fehlt das Attribut `rend="shyphen"` beim Element `<lb>`, oder dieses ist doppelt geschrieben, wenn es innerhalb eines Wortes vorkommt. Das führt zu einer falschen Tokenisierung des Wortes, was ein irrtümliches Tagging verursacht.

In den Beispielen 4.5 und 4.6 werden diese Fehler aus den NORM.xml-Dateien gezeigt und außerdem wie diese Wörter vom TreeTagger falsch getaggt werden. Nach jedem Beispiel wird auf eine richtige Verwendung des Elements explizit hingedeutet.

Beispiel 4.5 (aus *Ts-203_OA_NORM.xml,* Ts-203,54r[2]_1):

```
Kausa<lb/>litätsgesetz
```

Das Wort *Kausalitätsgesetz* wird falsch von Tagger getaggt, deswegen findet man es in der entsprechend getaggten NORM.xml wie folgt:

Tagger:

```
<w l="UNKNOWN" t="NN">Kausa</w>
<lb/>
<w l="UNKNOWN" t="ADJD">litätsgesetz</w>
```

Eine richtige Platzierung in *Ts-203_OA_NORM.xml* wäre:

```
Kausa<lb rend="shyphen"/>litätsgesetz
```

Beispiel 4.6 (aus *Ts-233b_OA_NORM.xml,* Ts-233b,4[3]_1):

```
unge<lb rend="shyphen"/> <lb/>wohnter
```

Das Wort *ungewohnter* wird falsch von Tagger getaggt, deswegen findet man es in der entsprechend getaggten NORM.xml wie folgt:

Tagger:

```
<w l="UNKNOWN" t="ADJA">unge<lb
                        rend="shyphen">
</w>
<lb/>
<w l="UNKNOWN" t="ADJA">wohnter</w>
```

Eine richtige Platzierung in *Ts-233b_OA_NORM.xml* wäre:

```
unge<lb rend="shyphen"/>wohnter
```

4.2 Editionsprobleme

Wie am Anfang dieses Kapitels erwähnt wurde, werden von den Editoren aus Bergen einige für CIS-Entwickler nicht nützliche Informationen supprimiert. Die getilgten Informationen wurden durch ein neues XML-Element namens `<seg type="stripped">` ersetzt. Dieses Element kann sich sowohl außerhalb als auch innerhalb eines Wortes, einer Phrase oder eines ganzen Satzes befinden. Manchmal gibt es Leerzeichen zwischen den bestehenden Teilen des Elementes (Starttag und Endtag) und dessen Inhalt. Aber manchmal ist es nicht der Fall. Diese willkürliche Positionierung des `<seg type="stripped">` Elements erschwert die Suche nach einer Lösung, mit der der TreeTagger eine richtige Zerlegung eines Wortes schaffen könnte, wenn das Element innerhalb dieses Wortes vorkommt. Diese Fälle, in denen das Element `<seg type="stripped">` innerhalb eines Wortes erscheint, werden ab jetzt als Editionsprobleme bezeichnet.

4.2.1 Platzierung des XML-Elements `<seg type="stripped">`

Das Element `<seg type="stripped">` bedeutet nicht immer ein Problem für die Zerlegung der Wörter in Editionstexten, was mittels des Beispiels 4.7 gezeigt wird.

Beispiel 4.7 (aus *Ms-102_OA_NORM.xml,* Ms-102,6v[3]et7v[1]_1):

```
Schi<seg type="stripped">f</seg>f
```

Das Wort *Schiff* wird richtig von Tagger getaggt, deswegen findet man es in der entsprechend getaggten NORM.xml wie folgt:

Tagger:

```
<w l="Schiff" t="NN">Schi<seg
      type="stripped">f</seg>f</w>
```

4.2.2 Editionsprobleme beim XML-Element `<seg type="stripped">`

In den Beispielen 4.8 und 4.9 dagegen wird klargestellt, wie dieses Element eine richtige Aufsplittung der NORM.xml-Dateien verhindert. Daher zeigen die resultierenden, getaggten Dateien ein falsches Ergebnis beim Tagging.

Beispiel 4.8 (aus *Ts-210_OA_NORM.xml,* Ts-210,76[2]et77[1]_1):

```
<persName key="Euklid"><seg
      type="stripped">E</seg>uklidische</persName>
```

Das Wort *Euklidische* wird falsch von Tagger getaggt, deswegen findet man es in der entsprechend getaggten NORM.xml wie folgt:

Tagger:

```
<persName key="Euklid">
<seg type="stripped">
<w t="XY" l="E">E</w>
</seg>
<w t="ADJA" l="UNKNOWN">uklidische</w>
</persName>
```

Beispiel 4.9 (aus *Ms-114_OA_NORM.xml,* Ms-114,125r[1]_2):

```
<seg type="stripped">M</seg>enschliches
```

Das Wort *Menschliches* wird falsch von Tagger getaggt, deswegen findet man es in der entsprechend getaggten NORM.xml wie folgt:

Tagger:

```
<seg type="stripped">
<w t="NE" l="M">M</w>
</seg>
```

```
<w t="ADJA" l="UNKNOWN">enschliches</w>
```

4.3 Lösung von Transkriptionsfehlern

Leider lassen sich die Transkriptionsfehler nicht sofort lösen. Eine Suche bei unterschiedlichen Fällen wurde unternommen. Dafür wurde eine Wortliste und eine entsprechende Frequenzliste erzeugt. Die Wörter mit den Häufigkeiten im Bereich von 1 bis 10 wurden kontrolliert. Dabei wurde festgestellt, dass sehr viele Wörter aufgrund dieser Transkriptionsfehler falsch tokenisiert erscheinen. Die ursprüngliche Idee war, die entsprechenden, auslösenden Transkriptionsfehler unmittelbar in den NORM.xml-Dateien zu korrigieren. Im Laufe der Korrekturen stellte sich aber heraus, dass es einen riesigen Aufwand erfordert.

Die Schlussfolgerung ist, dass die einzige Lösung dafür ist, bei den Bergen-Editoren alle Transkriptionsfehler zu melden und auf die aktualisierten XSLT-Dateien zu warten.

4.4 Lösung bei Editionsproblemen

Wie bereits am Anfang dieses Kapitels erwähnt wurde, verursachen Editionsprobleme eine irrtümliche Trennung der Tokens, wenn der TreeTagger über die NORM.xml-Dateien läuft. Daher scheitert auch das Tagging bei diesen Wörtern.

Wenn die einzelnen Fälle betrachtet werden, in denen diese Fehler stattfinden, ist festzustellen, dass das Element `<seg type="stripped">` immer miteinbezogen ist. Deswegen werden alle diese `<seg type="stripped">`-Strukturen gesammelt.

Im Unterkapitel 2.2 wurde das Programm *process.sh* beschrieben und dazu erklärt, dass während des Preprocessings der TreeTagger zwischen den Elementen ein Leerzeichen und das leere Element <sp/> hinzufügt. Es wird im Folgenden analysiert, wie das `<seg type="stripped">` Element nach diesem Schritt des TreeTaggers erscheint.

Richtige `<seg type="stripped">` Struktur

Bei den `<seg type="stripped">` Elementen innerhalb eines Wortes ist festzustellen, dass die Wörter, bei denen sich der TreeTagger richtig verhält, immer folgende Struktur zeigen:

```
[A-Za-z]<seg type="stripped">[A-Za-z]|[PUNCT]</seg>
```

Diese Struktur erlaubt einen Teil des Wortes gefolgt von Starttag, anderen Wortteilen, und dem Endtag. Zwischen den Wortabschnitten und den Elementteilen sind keine Leerzeichen vorhanden. Beispiel 4.7 zeigt diese Struktur.

Problematische `<seg type="stripped">` Struktur

Bei dem falschen Verhalten des Taggers kann man feststellen, dass sich bei den meisten Fällen eine Tatsache wiederholt: das Wort fängt nach dem Starttag an und

zwischen den Elementteilen und den Wortteilen sind Leerzeichen vorhanden. Eine generalisierende Vorlage für diese problematische Struktur ist folgende:

```
^|\s<seg type="stripped">\s[A-Za-z]|[PUCT]\s</seg>\s([A-
Za-z]|[PUNCT])*
```

Das Wort fängt hierbei nach dem Starttag an und zwischen den Elementteilen und den Wortteilen sind Leerzeichen vorhanden.

An dieser Stelle des Preprocessings wird das Wort *Euklidische* vom Beispiel 4.8 wie folgt strukturiert:

Tagger beim Preprocessing:

```
<segðtype="stripped"> E </seg> uklidische
```

Diese Struktur führt zur folgenden, falschen späteren Aufsplittung und daher zu einem falschen Tagging der Personennamen.

Tagger beim Preprocessing:

```
<seg type="stripped">
E
</seg>
uklidische
```

Suche nach problematischen Leerzeichen

Die problematischen Leerzeichen müssen unbedingt korrigiert werden, damit das ganze Wort *Euklidische* in einer einzigen Zeile nach der Tokenisierung vorkommt. Sie müssen aber an einer Stelle beibehalten werden, und zwar genau an der Stelle hinter dem `<seg type="stripped">` Element. Auf diese Weise kann der TreeTagger das Starttag von dem Rest des Wortes teilen.

Nach der Tokenisierung fängt das Wort in einer neuen Zeile an, und kann richtig vom TreeTagger erkannt und getaggt werden. Zum Beispiel, wenn eine Zeile nach der Tokenisierung wie folgt anfängt: `<seg type="stripped">E</seg>uklidische`, kann der TreeTagger die Personennamensvariante *`Euklidische`* nicht erkennen, obwohl alles in der selben Zeile steht. Wenn das Leerzeichen zwischen `<seg type="stripped">` und dem Buchstaben *`E`* bleibt, aber auch kein Leerzeichen zwischen *`E`*, `</seg>` und *`uklidische`* vorkommt, wird das Token von dem TreeTagger richtig gesplittet:

```
<seg type="stripped">
E</seg>uklidische
```

In diesem Fall erkennt der TreeTagger das Wort *Euklidische* richtig und das Tagging ist erfolgreich.

4.4.1 Anpassung von TreeTagger mit Regex II

Weiterhin muss herausgefunden werden, in welchem Skript des Preprocessings die Änderungen gemacht werden sollen. Dafür wird die Arbeitsweise des TreeTaggers Schritt für Schritt analysiert und bestimmte Programme vom Skript *process.sh* angepasst.

Dabei wird festgestellt, dass das entscheidende Programm *filter-german-tokin-utf8.perl* ist, in dem das Element ð innerhalb der XML-Tags hinzugefügt wird. Um die Leerzeichen wie gewünscht zu korrigieren, wird das Symbol „€" an die gewünschte Stelle addiert. In einem späteren Schritt des Preprocessings, wird das Hilfssymbol „€" mit dem Befehl `s/€//g`; gelöscht. Das findet im Skript *filter-german-tokin2-utf8.perl*[7] statt. Um das Symbol „€“ hinzuzufügen, wird folgender regulärer Ausdruck im Programm *filter-german-tokin-utf8.perl*[8] implementiert:

Regex I:

```
s/(\s)(<segðtype="stripped">)(\s)(.*)(\s)(<\/seg>)(\s)/$1$2$3$4€
$6€/g;
```

Bei diesem regulären Ausdruck stößt man jedoch auf ein neues Problem. Wie im Kapitel 2 erklärt wurde, generiert der TreeTagger das neues XML-Element `<w>`. Die Aufsplittung des Wortes nach der Einführung unseres regulären Ausdrucks sieht in der neuen getaggten NORM.xml-Datei bei manchen Fällen wie folgt aus:

```
<seg type="stripped">
<w key="Euklid" l="euklidisch"
     t="ADJA">E</seg>uklidische
</w>
```

Dieses Resultat führt zu einer unwohlgeformten XML-Datei. Innerhalb des Elementes `<w>` erscheint nur der Endtag `</seg>`. Damit eine XML-Datei richtig bzw. wohlgeformt ist, muss auch unbedingt der Starttag `<seg type="stripped">` innerhalb von `<w>` vorhanden sein (vgl. Unterkapitel 1.1). Um dieses Problem zu lösen, wird der reguläre Ausdruck geändert, indem nur bei diesen Fällen, sowohl das Starttag als auch das Endtag vom `<seg>` Element in jeweils ein leeres Element umgewandelt werden. Das heißt, der Schrägstrich wird am Ende der jeweiligen Elementteile hinzugefügt: `<seg type="stripped"/>` und `<seg/>`.

Der verbesserte reguläre Ausdruck sieht wie folgt aus:

Regex II:

```
s/(<segðtype="stripped">)(\s)(.*?)(\s)(<\/seg>)
(\s)/<segðtype="stripped"\/>$2$3€<seg\/>€/g;
```

7 Verändertes Programm *filter-german-tokin2-utf8.perl* ist im Anhang als *angepasste_filter-german-tokin2-utf8.perl* zu finden.

8 Verändertes Programm *filter-german-tokin-utf8.perl* ist im Anhang als *angepasste_filter-german-tokin-utf8.perl* zu finden.

4.4.2 Verbesserung nach der Implementierung von Regex II

Nach der Implementierung von Regex II im TreeTagger wurden die Beispiele 4.8 und 4.9 wieder von den getaggten Dateien extrahiert und betrachtet. Es wurde festgestellt, dass der TreeTagger die Wörter richtig gesplittet und getaggt hat. Außerdem werden keine Fehlermeldungen vom Parser angezeigt, was bedeutet, dass die resultierenden XML-Dateien wohlgeformt sind.

In den Beispielen 4.10 und 4.11 sind die korrigierten Beispiele von 4.8 und 4.9 zu sehen.

Beispiel 4.10 (aus *Ts-210_OA_NORM.xml* , Ts-210,76[2]et77[1]_1):

```
<persName key="Euklid"><seg
     type="stripped">E</seg>uklidische</persName>
```

Tagger vor Implementierung von Regex II:

```
<persName key="Euklid">
<seg type="stripped">
<w t="XY" l="E">E</w>
</seg>
<w t="ADJA" l="UNKNOWN">uklidische</w>
</persName>
```

Das Wort *Euklidische* wird nun richtig von Tagger getaggt, deswegen findet man es in der entsprechend getaggten NORM.xml wie folgt:

Tagger nach Implementierung von Regex II:

```
<seg type="stripped"/>
<w key="Euklid" l="euklidisch" t="ADJA|
     persName">E<seg/>uklidische</w>
```

Beispiel 4.11 (aus *Ms-114_OA_NORM.xml,* Ms-114,125r[1]_2):

```
<seg type="stripped">M</seg>enschliches
```

Tagger vor Implementierung von Regex II:

```
<seg type="stripped">
<w t="NE" l="M">M</w>
</seg>
<w t="ADJA" l="UNKNOWN">enschliches</w>
```

Das Wort *Menschliches* wird nun richtig von Tagger getaggt, deswegen findet man es in der entsprechend getaggten NORM.xml wie folgt:

Tagger nach Implementierung von Regex II:

```
<seg type="stripped"/>
<w t="NN" l="Menschliche"><seg/>enschliches</w>
```

4.4.3 Weitere problematische `<seg type="stripped">` Strukturen

Die Lösung mit Regex II bietet im Falle von Editionsproblemen nur eine schnelle aber nicht endgültige Lösung. Das `<seg type="stripped">` Element erscheint mit anderen Strukturen, die sich nicht einfach mit regulären Ausdrücken normalisieren lassen. Weitere reguläre Ausdrücke für andere bestimmte Fälle würden die Wortextraktion nur wieder verschlimmern, denn problematische Leerzeichen kommen in denselben NORM.xml-Dateien vor und werden nicht erst beim Preprocessing des TreeTaggers eingefügt. Ein Beispiel dafür ist das nächste Wort:

Beispiel 4.12 (aus *Ts-245_OA_NORM.xml,* Ts-245,151[5]et152[1]_6):

```
ent <seg type="stripped">s</seg>innen
```

Im Beispiel 4.12 befindet sich ein Leerzeichen zwischen dem Wortteil *ent* und dem Starttag des `<seg>` Elements bereits in der NORM.xml-Datei. Laut des TreeTagger-Verfahrens, das bereits beschrieben wurde, wird dieses Wort falsch gesplittet und getaggt:

Tagger:

```
<w l="UNKNOWN" t="ADJD">ent</w>
<sp/>
<seg type="stripped"/>
<w l="sinnen" t="VVINF">s<seg/>innen</w>
<w l="," t="$,">,</w>
<sp/>
```

Aber ein Muster zu erzeugen, das dieses Leerzeichen in der NORM.xml-Datei entfernt, würde andere Fällen treffen, bei denen die Struktur gleich aussehend ist, aber in denen dieses Leerzeichen nötig ist, wie Beispiel 4.13 zeigt:

Beispiel 4.13 (aus *Ms-112_OA_NORM.xml*,
Ms-112,93r[2]et93r[3]et93r[4]et93r[5]et93r[6]et93v[1]_5):

```
liegt <seg type="stripped">A</seg>lles
```

Auch hier ist die Schlussfolgerung, dass weitere Editionsprobleme nach einer sehr aufwendigen Arbeit gesammelt und bei den Bergen Editoren gemeldet werden sollten, wie es bereits bei den Transkriptionsfehlern vorgeschlagen wurde.

4.5 Evaluierung

Mit dem Programm *wordlist.py* wurden zwei Wortlisten erzeugt (siehe CD). Eine enthält die Liste aller Wörter aus dem ganzen Wittgenstein Nachlass vor der Implementierung des Regex II. Die andere dagegen enthält alle Wörter nach der Implementierung des

Regex II. Aus beiden Wortlisten wurden mit dem Skript *frequenz.py* jeweils zwei Frequenzlisten erstellt (siehe ebenfalls CD).

Bei der Analyse der Frequenzlisten wurde festgestellt, dass die Mehrheit der Wörter, die falsch tokenisiert wurden, eine Häufigkeit zwischen 1 und 10 haben. Das Programm *vollform_inventar.py* (siehe CD) ermittelt die Anzahl der Wörter, die n-mal in der Frequenzliste vorkommen. Im Folgenden sind zwei Fragmente aus diesen Inventaren zu finden:

```
Vollformen mit Frequenz 1 kommen 30658 Mal vor.
Vollformen mit Frequenz 2 kommen 9685 Mal vor.
Vollformen mit Frequenz 3 kommen 5877 Mal vor.
Vollformen mit Frequenz 4 kommen 4033 Mal vor.
Vollformen mit Frequenz 5 kommen 2881 Mal vor.
Vollformen mit Frequenz 6 kommen 2142 Mal vor.
Vollformen mit Frequenz 7 kommen 1687 Mal vor.
Vollformen mit Frequenz 8 kommen 1379 Mal vor.
Vollformen mit Frequenz 9 kommen 1096 Mal vor.
Vollformen mit Frequenz 10 kommen 984 Mal vor.
...
```

Fragment aus der Datei „vollform_inventar_vor_Regex II.txt“ (siehe CD)

```
Vollformen mit Frequenz 1 kommen 21673 Mal vor.
Vollformen mit Frequenz 2 kommen 6685 Mal vor.
Vollformen mit Frequenz 3 kommen 4226 Mal vor.
Vollformen mit Frequenz 4 kommen 3338 Mal vor.
Vollformen mit Frequenz 5 kommen 2303 Mal vor.
Vollformen mit Frequenz 6 kommen 1807 Mal vor.
Vollformen mit Frequenz 7 kommen 1330 Mal vor.
Vollformen mit Frequenz 8 kommen 1165 Mal vor.
Vollformen mit Frequenz 9 kommen 942 Mal vor.
Vollformen mit Frequenz 10 kommen 809 Mal vor.
...
```

Fragment aus der Datei „vollform_inventar_nach_Regex II.txt“ (siehe CD)

Nach einem Vergleich beider Inventare ist zu sehen, dass nach der Implementierung des Regex II die Anzahl der Happax legomena[9] sowie der anderen Wörter, die von 2 bis 10 Mal vorkommen, geringer ist. Das bedeutet das Regex II tatsächlich die Editionsprobleme teilweise lösen kann. Wie in 4.4.2 erwähnt wurde, muss an den weiteren Editionsproblemen gearbeitet werden.

9 Wörter, die im Text nur ein Mal vorkommen

Schlussfolgerung

Das Ziel der vorgestellten Arbeit bestand darin, die Optimierung der linguistischen Suche beim XML-annotierten Nachlass von Ludwig Wittgenstein durch die optimale Ausnutzung der XML-Annotation und die Verbesserung der XML-Edition zu erreichen. Da die Personennamen im Nachlass von Ludwig Wittgenstein eine sehr wichtige Rolle spielen, wurde der Schwerpunkt auf die Personennamenserkennung gesetzt.

Es wurde festgestellt, dass die FinderApp WiTTFind am CIS nicht alle Personennamen im Nachlass findet. In den aktuellen XML-Editionstexten von Wittgensteins Nachlass sind die Personennamen oder seine Ableitungen im XML-Element `<persName>` eingeschlossen.

Die Schwierigkeit bestand darin, dass der TreeTagger manche Personennamen nicht richtig erkennt und daher falsch getaggt hat, da er von den XML-Annotationen nicht profitiert. In dieser Hinsicht wurden zwei Methoden verwendet. In der ersten Methode wurden die NORM.xml-Dateien mit dem XML-Parser ElementTree von Python3 geparst und alle Personennamen lokalisiert. Damit wurde zum ersten Mal eine Liste erstellt, die alle Personennamen aus dem Wittgenstein Nachlass umfasst.

In einer zweiten Methode wurden fünf Maßnahmen vorgeschlagen. Es wurden zuerst alle problematischen Beispiele gesammelt und generalisiert, durch die der TreeTaggers bei der Erkennung von Personennamen scheitert. Danach wurde eine neue morphologische Kategorie mit dem Namen persName erzeugt, um das XML-Attribut `t` mithilfe vom Python-Parser in den getaggten Dateien zu erweitern. Außerdem wurde das XML-Attribut `l` verbessert, in dem das Lexikon des TreeTaggers erweitert wurde, um die „`UNKNOWN`“-Lemma-Attribute zu reduzieren. Es wurde ein zusätzliches Programm erzeugt, um die noch vorhandenen „`UNKNOWN`“-Lemma-Attribute endgültig zu korrigieren. Als vierte Maßnahme wurde das XML-Element `<w>` mit dem zusätzlichen Attribut `key` erweitert, das den vollständigen Namen der erwähnten Person beinhaltet.

Die letzte Maßnahme bestand darin, eine neue morphologische Kategorie mit dem Namen persName zu dem Suchmuster in WiTTFind einzuführen, um die Suche in der Suchmaschine zu verfeinern.

Der TreeTagger steuert seine Programme durch das Hauptprogramm *process.sh*. Das Programm *addKeyundLemmaName.py*, dass für die Erweiterung der XML-Attribute `t` und `l` erzeugt wurde, wird in *process.sh* implementiert. Damit können die Änderungen automatisch für alle XML-Dateien durchgeführt werden.

Ein Vergleich mit der aktuellen Suche von WiTTFind hat bestätigt, dass unsere Vorschläge die Suche der Personennamen optimiert. Im WiTTFind werden im Moment von insgesamt 20 Dokumenten 168 Personennamen in nur 13 Dokumenten getroffen. Nach unseren Maßnahmen sollte die Treffer-Anzahl auf 833 steigen. Außerdem hat unser System festgestellt, dass in allen Dokumenten, die im WiTTFind zur Verfügung stehen, Personennamen zu finden sind.

Ein weiteres Problem bei der Erkennung von Personennamen stellen die Transkriptionsfehler und die Editionsprobleme dar. Unsere erste Idee war, beide Schwierigkeiten direkt am CIS zu lösen. Im ersten Fall wurde nach harter Arbeit

festgestellt, dass die einzige Lösung ist, die entdeckten Transkriptionsfehler bei den Editoren im WAB zu melden, was zwar einen großen Aufwand bedeutet, aber die Qualität der XML-annotierten Editionen erheblich erhöht und gleichzeitig die Zusammenarbeit mit dem WAB bereichert.

Im Falle von Editionsproblemen konnte in dieser Arbeit eine kleine Lösung erreicht werden. Mittels eines regulären Ausdrucks (Regex II), der in einem Programm des Preprocessings vom TreeTagger implementiert wurde, ist die Anzahl der falsch tokenisierten Wörter, unter denen sich auch Personennamen befinden, gesunken. Das wurde mit der Anzahl der Happax legomena gemessen. Mit dem neuen regulären Ausdruck ist diese Anzahl um 9021 Wörter zurückgegangen. Auch bei den anderen Wörtern, deren Häufigkeit zwischen 2 und 10 liegt, hat sich die Anzahl reduziert. Es gibt aber noch viele Fälle, bei denen Editionsprobleme bestehen. Diese Fälle sollen, genau wie bei Transkriptionsfehlern, gesammelt und bei den Editoren in Bergen gemeldet werden.

Es ist zu befürchten, dass bei der Edition bei einer großen Menge an Personennamen übersehen wurde, diese mit dem Element `<persName>` zu etikettieren. In diesem Fall könnte man von einem achten Fall (zu den Fällen aus Unterkapitel 3.2) sprechen. Es wird vorgeschlagen, die bereits erstellte Namensliste mit allen Tokens aus den NORM.xml-Dateien mit Natural Language Processing Methoden zu vergleichen, um eine automatische Suche zu unternehmen. Im Vergleich dazu können Forscher durch traditionelle Methoden alle Textdateien durchlesen und solche Fälle nach großen Bemühungen herausfinden. Die Computerlinguistik bietet die Möglichkeit der Automatisierung an, was sie im Bereich der Digital Humanities so beliebt macht.

Anhang

Programm *xmlET_schnell_Ueberblick.py*

```
1.  ##########################################################################
2.  # Einfaches Programm für die Erklärung von xml.etree.ElementTree        #
3.  # im Kapitel 1                                                          #
4.  # Aufgabe: Erzeugung einer wohlgeformten XML-Datei mit                  #
5.  # xml.etree.ElementTree                                                 #
6.  # Autorin: Faridis Alberteris Azar                                      #
7.  # Aufruf: python3 xmlET_schnell_Ueberblick.py                           #
8.  ##########################################################################
9.
10. from xml.etree import ElementTree
11. from xml.etree.ElementTree import Element, SubElement, Comment, tostring
12. from xml.dom import minidom
13.
14. # Die XML- Datei wird in der Form von einem Baum hierarchisch erzeugt:
15.
16. # Die Funktion Element() generiert eine Knote
17. # (engl.: node). In diesem Fall wird das Wurzelelement erzeugt
18. wurzel = Element('inhalt')
19.
20. # Die Funktion Comment() generiert einen Kommentar
21. comment = Comment('Generating Xml Text with etree')
22. # Kommentar wird dem gewünschten Element zugewiesen
23. wurzel.append(comment)
24.
25. # Die Funktion SubElement() hängt ein neues Element zum Elternelement an
26. subElem1 = SubElement(wurzel, 'persName')
27.
28. # Inhalt eines Elementes bzw. Kindelementes generieren
29. subElem1.text = 'Ludwig Wittgenstein'
30.
31. subElem2 = SubElement(wurzel, 'werke')
32. subElem2.text = 'Der Philosph hat viele wichtege Werke hinterlassen:'
33.
34. # Text nach einem Element bzw. Kindelement einfügen
35. subElem2.tail = 'Schön, dass wir ein Stück Wittgensteins am CIS
    haben :)'
36.
37. subElem3 = SubElement (subElem2, 'tractatus')
38. subElem3.text = 'Der Philosoph schrieb das Buch Tarctatus'
39. subElem3.tail = 'Das ist sein wichtigster und bekanntester Werk'
40.
41. subElem4 = SubElement(subElem2, 'nachlass')
42. subElem4.text = 'Die Manuskripte und Typoskripte von L.W. zwischen 1906'
    \
43.                 'und 1916 sind als sein Nachlass bekannt'
44.
45. subElem5_mit_Entity = SubElement(wurzel, 'hier_ein_Entity')
46.
47. # Das Zeichen & wird automatisch in &amp; umgewandelt.
48. subElem5_mit_Entity.text = 'Wittgenstein & Die Philosophie'
49.
50. ###### Zwei Output möglichkeiten:
51.
52. ### 1) Direkter Aufruf mit der Funktion tostring()
53. ###    Leider ist dabei der Output nicht so schön formatiert:
54. #print (tostring(word))
55.
56. ### 2) Gebastelte Funktion prettify()
57. ##     (Quelle: https://pymotw.com/2/xml/etree/ElementTree/create.html)
58. ###    Empfohlene Möglichkeit:
```

```
59.
60. def prettify(docu):
61.     '''Diese Funktion gibt einen pretty-printed XML string aus
62.     '''
63.     rough_string = tostring(docu, 'utf-8', method="xml")
64.     reparsed = minidom.parseString(rough_string)
65.     return reparsed.toprettyxml(indent="  ")
66.
67. print (prettify(wurzel))
```

Programm *namenListe.py*

```
1.  #!/usr/bin/python3
2.
3.  ##########################################################################
4.  # Dieses Programm erzeigt die Liste "namensListe.txt" mit allen         #
5.  # Personennamen im XML-annotierten Editionstexten des                   #
6.  # Wittgenstein Nachlasses                                                #
7.  # Autorin: Faridis Alberteris Azar                                       #
8.  # Aufruf: sh namenslisteMaker.sh (um mehrere Dateien gliechzeitig        #
9.  # zu parsen!)                                                            #
10. # sonst: python3 namenListe.py <NORM.xml> <outputFile.txt>              #
11. ##########################################################################
12.
13. from xml.etree import ElementTree
14. from xml.etree.ElementTree import iterparse
15. import sys
16.
17. class PersonenNamen(object):
18.     '''Klasse zur Extraktion der Personennamen aus NORM.xml Dateien.
19.     Als XML- Parser wird das Modul xml.etree.ElementTree aus Python3
20.     verwendet
21.     '''
22.
23.     def __init__(self):
24.         self.indatei = open(sys.argv[2],'a')
25.
26.
27.     def getText(self):
28.         with open(sys.argv[1], 'rt') as f:
29.             self.__texte = ElementTree.parse(f)
30.         return self.__texte
31.
32.
33.     def getDocuName(self):
34.         self.path = './/{http://www.tei-c.org/ns/1.0}title'
35.         for node in self.getText().findall(self.path):
36.             self.docID= node.attrib.get \
37.                 ('{http://www.w3.org/XML/1998/namespace}id')
38.         print('\nDokument-ID: ' ,self.docID+'_OA_NORM.xml',\
39.               '\n',file=self.indatei)
40.
41.
42.     def getNamen(self):
43.         self.path = './/{http://www.tei-c.org/ns/1.0}persName'
44.
45.         kopf = 'Eingennamen in Wittgenstein-Lexikon'.center(50)+'\n\n'
46.         kopf += 'Namen als Vollform'.ljust(20)
47.         kopf += 'Namen als Lemma'.rjust(30)+'\n'
48.         print(kopf, file=self.indatei)
49.
50.         zaehler = 0
51.
52.         for tochterknote in self.getText().findall(self.path):
53.             self.lemmaName = tochterknote.attrib.get('key')
54.             if (self.lemmaName):
55.                 print("".join(tochterknote.itertext()).strip().ljust(15),
                        '\t\t',self.lemmaName.strip().center(45), \
56.                     file=self.indatei)
57.                  zaehler+=1
58.         print("Total von Personennamen: ",zaehler,file=self.indatei)
59.
60.
61. if __name__=='__main__':
62.     personen=PersonenNamen()
```

```
63.     personen.getText()
64.     personen.getDocuName()
65.     personen.getNamen()
```

Programm *addKeyundLemmaName.py*.

```
1. #!/usr/bin/python3
2.
3. ##########################################################################
4. # Dieses Programm ändert den Wert vom Attribut 't'.                      #
5. # Außerdem wird Attribut Lemma geändert, falls l = "UNKNOWN".            #
6. # Dieses Programm wird als Teil des TreeTaggers Ablauf eingesetzt.       #
7. # deswegen nimmt es als Input eine Output Datei aus einem vorherigen     #
8. # Schitt vom TreeTragger (siehe shell Skript prozess.sh)                 #
9. # Autorin: Faridis Alberteris Azar                                       #
   ##########################################################################
10.
11. from xml.etree import ElementTree
12. from xml.etree.ElementTree import tostring
13. import sys
14. from xml.dom import minidom
15.
16.
17. # XML-Dateie einlesen
18. texte = ElementTree.parse(sys.stdin)
19.
20.
21. path = './/{http://www.tei-c.org/ns/1.0}persName'
22. path1 = './/{http://www.tei-c.org/ns/1.0}w'
23. for elem in texte.findall(path):
24.     for node in elem.findall(path1):
25.         # Fügt key-Tattribut mit Lemma-Name zum Wort hinzu
26.         node.set('key',elem.attrib.get('key'))
27.         #vom Fall 1) wenn t=NE
28.         if node.attrib.get('t') == 'NE':
29.             node.set('t','NE|persName')
30.         elif node.attrib.get('l') == 'UNKNOWN':
31.             node.set('l',elem.attrib.get('key'))
32.         #vom Fall 3, 4) und 6) wenn t=NN
33.         if node.attrib.get('t') == 'NN':
34.             node.set('t','NN|persName')
35.         elif node.attrib.get('l') == 'UNKNOWN':
36.             node.set('l',elem.attrib.get('key'))
37.          #vom Fall 3 wenn t=XY
38.         if node.attrib.get('t') == 'XY':
39.             node.set('t','XY|persName')
40.         elif node.attrib.get('l') == 'UNKNOWN':
41.             node.set('l',elem.attrib.get('key'))
42.         #vom Fall 2.a) wenn t=ADJA
43.         if node.attrib.get('t') == 'ADJA':
44.             node.set('t','ADJA|persName')
45.         elif node.attrib.get('l') == 'UNKNOWN':
46.             node.set('l',elem.attrib.get('key'))
47.         #vom Fall 2.b) wenn t=ADJD
48.         if node.attrib.get('t') == 'ADJD':
49.             node.set('t','ADJD|persName')
50.         elif node.attrib.get('l') == 'UNKNOWN':
51.             node.set('l',elem.attrib.get('key'))
52.         #vom Fall 5.a) wenn t=VVFIN
53.         if node.attrib.get('t') == 'VVFIN':
54.             node.set('t','VVFIN|persName')
55.         elif node.attrib.get('l') == 'UNKNOWN':
56.             node.set('l',elem.attrib.get('key'))
57.         #vom Fall 5.b) wenn t=VVIMP
58.         if node.attrib.get('t') == 'VVIMP':
59.             node.set('t','VVIMP|persName')
60.         elif node.attrib.get('l') == 'UNKNOWN':
61.             node.set('l',elem.attrib.get('key'))
62.         #vom Fall 7) wenn t=TRUNC
```

```
63.         if node.attrib.get('t') == 'TRUNC':
64.             node.set('t','TRUNC|persName')
65.         elif node.attrib.get('l') == 'UNKNOWN':
66.             node.set('l',elem.attrib.get('key'))
67.
68.
69.
70. root = texte.getroot()
71.
72. # Die Registrierungs des Namespaces ist sehr wichtich!!!
73. # Sonst bekommt man keine Elemente vom urspünglichen XML-Dokument:
74. ElementTree.register_namespace("", "http://www.tei-c.org/ns/1.0")
75.
76.
77. def prettify(docu):
78.     '''Diese Funktion gibt einen pretty-printed XML string für
79.        das Element Element.
80.        (Quelle: https://pymotw.com/2/xml/etree/ElementTree/create.html)
81.     '''
82.     rough_string = tostring(docu,'utf-8', method="xml")
83.     reparsed = minidom.parseString(rough_string)
84.     return reparsed.toxml()
85.
86.
87.
88. def remove_whitespace(docu):
89.     return '\n'.join([x for x in docu.split('\n') if x.strip()!=''])
90.
91. print (remove_whitespace(prettify(root)))
```

Programm *angepasstes_prozess.sh*

```
1.  BASE_DIR=$1; shift
2.  TAGGED_DIR=$1; shift
3.
4.  DIR=$BASE_DIR/tree-tagger
5.  BIN=$DIR/bin
6.  CMD=$DIR/cmd
7.  LIB=$DIR/lib/TreeTagger
8.
9.  pushd $BASE_DIR
10.
11. for f in "$@"
12. do
13.     echo "$f"
14.     destination="$(dirname $f)/  \
15.                 $(basename $f .xml)-tagged.xml"
16.     cmd/put-xml-tags-on-one-line.perl $f | \
17.     cmd/extract-text.perl | \
18.     perl -pe 's/ ([0-9]+)\)/ $1\n) /g' | \
19.     $SHELL tokenize-german-utf8 | \
20.     perl -pe \
21.       's/^(<seg .*"notation">.*<\/seg>)$/§§§$1§§§\tNN/' | \
22.     $BIN/tree-tagger lib/german.par -token -lemma -sgml \
23.        -proto -cap-heuristics -lex lib/aux-lex.tx | \
24.     perl -pe 's/^§§§(.*)§§§\tNN.*/$1\tXY\t<unknown>/' | \
25.     cmd/filter-german-tags.perl | \
26.     cmd/generate-xml-output.perl $f | \
27.     cmd/addKeyundLemmaName.py | \
28.     cmd/generate-xml-zeichenentities.perl $f > $destination
29. done
30.
31. popd
```

Programm *erweitereLexikon.py*

```
1.        ##########################################################
2.        # Das Programm vergleicht eine Liste von Personenamen    #
3.        # und Lemma-Namen mit das lexikon von TreeTagger.        #
4.        # Lexikon von TreeTagger wird erwitert.                  #
5.        # Autorin: Faridis Alberteris Azar                       #
6.        # Aufruf:                                                #
7.        # python3 erweitereLexikon.py namensLemmaListe.txt aux-  #
8.        # lex.txt                                                #
9.        ##########################################################
10.
11. import sys
12. import collections
13. import pickle
14.
15. #'namensLemmaListe.txt' einlesen:
16. nameslist=open(sys.argv[1],'r',encoding='utf8')
17.
18. #'aux-lex.txt' einlesen:
19. german_lexikon=open('aux-lex.txt','r',encoding='utf8')
20.
21. #'aux-lex.txt' für die Erweiterung öffnen:
22. aux_lexikon=open(sys.argv[2],'a',encoding='utf8')
23.
24. gerlex=dict()
25. for line in german_lexikon:
26.     lex = line.split('\t',1)
27.     if len(lex) >=2:
28.             gerlex[lex[0].strip()]=lex[1].rstrip()
29.
30.       # Namensliste einlesen
31. namesliste=dict()
32. for line in nameslist:
33.    entry = line.split('\t',1)
34.    if len(entry) >= 2:
35.      namesliste[entry[0].strip()] = entry[1].rstrip()
36. nameslist.close()
37.
38.
39. # Namensliste mit TreeTagger Format:
40. # Vollform gefolgt von einem Tab und eine
41. # Sequenz von tag/lemma-pairs getrennt von Leerzeichen
42. for key,value in namesliste.items():
43.     if key not in gerlex:
44.         print(str(key).strip()+'\t'+'NE' \
45.           +' '+value.strip(),file=aux_lexikon)
46.
47.
48. german_lexikon.close()
49. nameslist.close()
50. aux_lexikon.close()
```

Programm *angepasste_filter-german-tokin-utf8.perl*

```
1.  #!/usr/bin/env perl
2.  # Author: Helmut Scmid:
3.  # Anpassung: Faridis Albeteris Azar
4.  use utf8;
5.  use Encode;
6.
7.  while (<>) {
8.    $_ = Encode::decode('utf8',$_);
9.
10.   tr/\177/ /;
11.
12.   # replace blanks within XML tags with ð
13.   while (s/(<[A-Za-z].*?) ([^<>]*>)/$1ð$2/g) {}
14.
15.   # Mark parentheses and quotation symbols
16.   # which are immediately preceded by punctuation
17.   s/(\w[.\?\!])(\"|\)|\'\'|”|"|«)/$1 $2 ÿ /g unless /<.*[.\?\!]
    (\"|\)|\'\'|”|"|«).*>/;
18.
19.
20.   #Erase the Space between tokens around </seg>(Faridis)
21.   s/(<segðtype="stripped">)(\s)(.*?)(\s)
22.      (<\/seg>)  (\s)/<segðtype="stripped"\/>$2$3€<seg\/>€/g;
23.
24.
25.   print Encode::encode('utf8',$_);
26. }
```

Programm *angepasste_filter-german-tokin2-utf8.perl*

```
1.  #!/usr/bin/perl
2.  #Author: Helmut Schmid
3.  #Anpassung: Faridis Albeteris Azar
4.
5.  use utf8;
6.  use Encode;
7.
8.  while (<>) {
9.      $_ = Encode::decode('utf8',$_);
10.
11.     # restore whitespace inside of XML tags
12.     tr/ð/ /;
13.     # Erase € (Faridis)
14.     tr/€//;
15.     # insert missing whitespace after punctuation
16.     if (length($_) > 5 &&
17.        (/^([A-Za-zÀ-ÿ]+)([.:\?\!])([\"\)\']*)([\(,]*)([A-Za-
                zÀ-ÿ]+)$/ ||
18.         /^([A-Za-zÀ-ÿ]+)([,;\/])([\"\)\']*)([\(,]*)([A-Za-
           zÀ-ÿ]+)$/))
19.     {
20.        print Encode::encode('utf8',"$1\n$2\n");
21.        print Encode::encode('utf8',"$3\n") if ($3 ne '');
22.        print Encode::encode('utf8',"$4\n") if ($4 ne '');
23.        print Encode::encode('utf8',"$5\n");
24.     }
25.     else {
26.        print Encode::encode('utf8',$_);
27.     }
28. }
```

Abbildungsverzeichnis

Abbildung 1.1: Element <persName> 3

Abbildung 1.2: Element <persName> enthält Element <seg> 4

Abbildung 1.3: Leeres Element <lb> 4

Abbildung 1.4: Elemente <persName> und <lb> und ihre Attribute 4

Abbildung 2.1: Hauptteil des Programms *process.sh* von Dr. Helmut Schmid..... 20

Abbildung 3.1: Quelltext des Programms *child_in_Norm.py*..... 28

Abbildung 3.2: Semantische Suche nach Personennamen in WiTTFind 29

Abbildung 3.3: Gefundene Namen beim Semantischen Finden in WiTTFind: *Napoleons* 29

Abbildung 3.4: Gefundene Namen beim Semantischen Finden in WiTTFind: *Russellschen, Eulerscher* und *Venus* 30

Abbildung 3.5: Gefundene Namen beim Semantischen Finden in WiTTFind: *hellenisch* 30

Abbildung 3.6: Gefundenen Namen beim Semantischen Finden in WiTTFind: *Eintein* 30

Abbildung 3.7: Statistik von gefundenen Personennamen aus WiTTFind..... 61

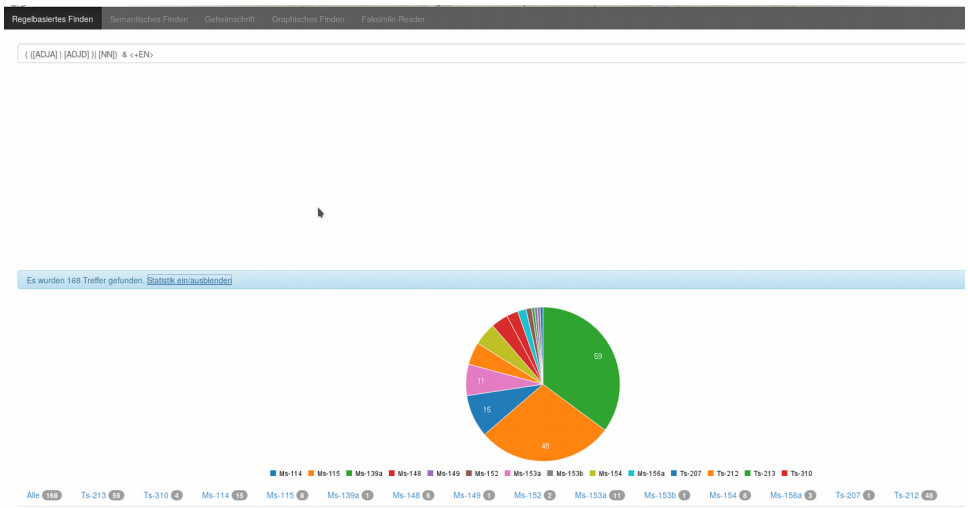

Abbildung 3.7: Statistik von gefundenen Personennamen aus WiTTFind

Verzeichnis der Tabellen

Tabelle 1.1: Zeichen-Entitäten …… 5
Tabelle 2.1: Tags und ihre Beschreibung …….. 18
Tabelle 3.1: Statistik-Vergleich von Personennamen …….. 62

Tabelle 3.1: ***Statistik-Vergleich von Personennamen***

Faksimile	WiTTFind-Statistik	Erstellte Namensliste
Ms-114_OA	15	67
Ms-115_OA	8	39
Ms-139a_OA	1	3
Ms-140_39v_OA	0	2
Ms-141_OA	0	4
Ms-148_OA	6	8
Ms-149_OA	1	11
Ms-150_OA	0	2
Ms-152_OA	2	29
Ms-153a_OA	11	39
Ms-153b_OA	1	17
Ms-154_OA	8	58
Ms-155_OA	0	8
Ms-156a_OA	3	11
Ts-201a1_OA	0	47
Ts-201a2_OA	0	28
Ts-207_OA	1	2
Ts-212_OA	48	207
Ts-213_OA	59	237
Ts-310_OA	4	14
Total in 5.000 Seiten:	**168**	**833**

Inhaltsverzeichnis der beigelegten CD

01-Bachelorarbeit

02-Generierung der Namensliste

03-TreeTagger Optimierung

04-erstellte Programme

05-angepasste Programme

06-erstellte Listen

07-erstellte Abbildungen

08-zitierte Quellen

Literaturverzeichnis

[1] Bongers, Frank. XSLT 2.0 & Xpath 2.0. Galileo Computing, Bonn, 2008.

[2] Burnard, Lou. Introduction In: What is the Text Encoding Initiative? How to add intelligent markup to digital resources. Marseille: OpenEdition Press, 2014. 22. Mai 17. Online: <http://books.openedition.org/oep/679>.

[3] Gabler, Hans Walter. Wittgensteins Nachlass. The Bergen Electronic Edition. In: Henrikson, Paula; Janss, Christian (eds.) : Geschichte der Edition in Skandinavien. *Bausteine zur Geschichte der Edition*, Vol. 4. Berlin: De Gruyter, 2013. S. 156-165. 22.Mai.2017. Online: <https://epub.ub.uni-muenchen.de/17082/7/pre17082.pdf>.

[4] Hadersbeck, Maximilian. Informationen zu „XML CISWAB“, 2013. 20. Mai 2017. Online: <http://wastwiki.cis.uni-muenchen.de/w/index.php5?title=XML_CISWAB&action=info>.

[5] Hellmann, Doug. Creating XML Documents-Python Module of the Week. Pymotw.com. 23. Mai 2017. Online: <https://pymotw.com/2/xml/etree/ElementTree/create.html>.

[6] Ordi i Fernández, Joan, Ludwig Wittgenstein, en Fernández Labastida, Francisco – Mercado, Juan Andrés (editores), Philosophica: Enciclopedia filosófica. 19. Mai 2017. Online: <http://www.philosophica.info/archivo/2009/voces/wittgenstein/Wittgenstein.html>.

[7] Pichler, Alois. Wittgenstein Archives at the University of Bergen (WAB): Open Access to Wittgenstein's Nachlass. XML based Interactive Dynamic Presentation (IDP) of WAB's Nachlass transcriptions. 16. Mai 2017. Online: <http://wab.uib.no/transform/wab.php?modus=opsjoner>.

[8] Ray, Erik T. Einführung in XML. O‘Reilly Verlag, Köln, 2001.

[9] Schiller, Anne, Thielen, Christine u.a. Guidelines für das Tagging deutscher Textcorpora mit STTS (Kleines und großes Tagset). Universität Stuttgart/Tübingen, 1999. 18. Mai 2017. Online: <http://www.sfs.uni-tuebingen.de/resources/stts-1999.pdf>.

[10] Schmid, Helmut. Probabilistic Part-of-Speech Tagging Using Decision Trees. *Proceedings of International Conference on New Methods in Language Processing*, Manchester, UK, 1994. 12. Mai. 2017 Online: <http://www.cis.uni-

muenchen.de/~schmid/tools/TreeTagger/data/tree-tagger1.pdf>.

[11] Schmid, Helmut. Improvements in Part-of-Speech Tagging with an Application to German. Proceedings of the ACL SIGDAT-Workshop. Dublin, Ireland, 1995. 12. Mai. 2017. Online: <http://www.cis.uni-muenchen.de/~schmid/tools/TreeTagger/data/tree-tagger2.pdf>.

[12] Schmid, Helmut. README File How to use the TreeTagger, University of Stuttgart, Germany, 1996. 12. Mai 2017. Online: <http://www.cis.uni-muenchen.de/~schmid/tools/TreeTagger/>.

[13] Extensible Markup Language. In: Wikipedia, Die freie Enzyklopädie.01. Juni 2017. Online: <https://de.wikipedia.org/w/index.php?title=Extensible_Markup_Language&oldid=164814823>.

[14] P5: Guidelines for Electronic Text Encoding and Interchange. 23. Mai 2017. Online: <http://www.tei-c.org/release/doc/tei-p5-doc/en/html/>.

[15] Standard Generalized Markup Language. In: Wikipedia, Die freie Enzyklopädie.01. Juni 2017. Online: <https://de.wikipedia.org/w/index.phptitle=Standard_Generalized_Markup_Language&oldid=164899412>.

[16] Text Encoding Initiative. In: Wikipedia, Die freie Enzyklopädie. Bearbeitungsstand: 01. Juni 2017. Online: <https://de.wikipedia.org/w/index.phptitle=Text_Encoding_Initiative&oldid=164712784>.