Bachelorarbeit

im Studiengang Computerlinguistik

an der Ludwig- Maximilians- Universität München

Fakultät für Sprach- und Literaturwissenschaften

Ludwig Wittgenstein's Nachlass: tokenizing and frequency list for WiTTFind

vorgelegt von
Lina Garcia Jordan

Betreuer:	Dr. Maximilian Hadersbeck
Prüfer:	Dr. Maximilian Hadersbeck
Bearbeitungszeitraum:	24. September - 03. December 2018

Selbstständigkeitserklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig angefertigt, alle Zitate als solche kenntlich gemacht sowie alle benutzten Quellen und Hilfsmittel angegeben habe.

München, den 03. Dezember 2018

.....
Lina Garcia Jordan

Abstract

Keywords: *Frequency list; POS-tagging; Ludwig Wittgenstein; Nachlass*

The Center for Information and Language Processing at the Ludwig-Maximilians-Universität (CIS) has been working together with the University of Bergen on the project Wittgenstein Advanced Search Tools. The CIS developed a search engine, WiTTFind that provides different possibilities to explore Wittgenstein's *Nachlass*. One of these is the semantic search. Until recently, only 5.000 pages out of the 20.000 pages of Ludwig Wittgenstein's *Nachlass* were open to the public, and some of WiTTFind's programs and tools have not yet been updated to process all of the files. This thesis improves the semantic search for WiTTFind by creating semantic frequency lists for the categories of color and music for all the pages of the *Nachlass* and by creating a program that can easily be adapted to search for any other semantic category. To do so, the deploy chain for the search engine was analyzed and the programs were implemented following its structure. The objective of this thesis is to obtain semantic frequency lists with high accuracy and to identify sources of errors for the semantic search in WiTTFind. The findings made through this work enable the improvement of the lexicon used by the search engine and the frequency lists obtained will be used by WiTTFind.

Übersicht

Schlagwörter: *Frequenzlisten; POS-tagging; Ludwig Witggenstein; Nachlass*

Das Zentrum für Informations- und Sprachverarbeitung an der Ludwig-Maximilians-Universität (CIS) arbeitet seit mehreren Jahren gemeinsam mit der Universität Bergen an dem Projekt Wittgenstein Advanced Search Tools. Das CIS entwickelte eine Suchmaschine, WiTTFind, die verschiedene Möglichkeiten bietet, um Wittgensteins Nachlass zu erforschen. Eines davon ist die semantische Suche. Bis vor kurzem waren nur 5.000 der 20.000 Seiten von Ludwig Wittgensteins Nachlass für die Öffentlichkeit zugänglich. Einige Programme und Werkzeuge von WiTTFind sind noch nicht auf dem neuesten Stand, um alle Dateien zu verarbeiten. Diese Bachelorarbeit verbessert die semantische Suche für WiTTFind, indem semantische Frequenzlisten für die Kategorien Farbe und Musik für alle Seiten des Nachlass erstellt werden. Diese werden mit Hilfe eines Programmes, das leicht für die Suche nach anderen semantischen Kategorien angepasst werden kann, erstellt. Dazu wurde die *deploy chain* für die Suchmaschine analysiert und die Programme entsprechend ihrer Struktur implementiert. Ziel dieser Bachelorarbeit ist es, semantische Häufigkeitslisten mit ho-

her Genauigkeit zu erhalten und Fehlerquellen für die semantische Suche in WiTTFind zu identifizieren. Die Ergebnisse dieser Arbeit ermöglichen die Verbesserung des von der Suchmaschine verwendeten Lexikons, und die erhaltenen Frequenzlisten werden von WiTTFind verwendet.

Resumen

Palabras clave: *Lista de frecuencias; POS-tagging; Ludwig Witggenstein; obras póstumas;*

El Centro de Información y Procesamiento de Idiomas en la Ludwig-Maximilians-Universität (CIS) ha estado trabajando junto con la Universidad de Bergen en el proyecto Wittgenstein Advanced Search Tools. El CIS desarrolló un buscador, WiTTFind, que ofrece diferentes posibilidades para explorar la obra de Wittgenstein. Una de ellas es la búsqueda semántica. Hasta hace poco, solo 5.000 de las 20.000 páginas de las obras póstumas de Ludwig Wittgenstein estaban abiertas al público. Algunos de los programas y herramientas utilizados por WiTTFind aún no se han actualizado para procesar todos los archivos. Esta tesis mejora la búsqueda semántica de WiTTFind creando listas de frecuencias semánticas para las categorías de color y música. Para esto, un programa que se puede adaptar fácilmente para buscar cualquier otra categoría semántica fue creado. La *deploy chain* del buscador fue analizada y los programas creados fueron implementados siguiendo su estructura. El objetivo de esta tesis es obtener listas de frecuencias semánticas con alta precisión e identificar fuentes de errores para la búsqueda semántica en WiTTFind. Los hallazgos realizados a través de este trabajo permiten la mejora del léxico utilizado por el buscador y las listas de frecuencia obtenidas serán utilizadas por WiTTFind.

Contents

1. Introduction

The changes produced in the last years by the technological advances are not limited to the everyday life. A revolution in the traditional field of humanities has been taking place as well. The new computational tools and methods can help to get a new approach to old and generate new questions and identify research possibilities. The intersection between the humanities and the use of computational tools is called Digital Humanities. Computational linguistics, which is an interdisciplinary field of research that involves the areas of mathematics, computer science and linguistic, has a major role to play in it[3].

The field of humanities includes a vast number of studies such as ancient and modern languages, history, literature and philosophy. The latter is characterized by its connection to many other scientific fields and can too, profit from the development of Digital Humanities among other things by suitably digitizing philosophical works and then applying new methods to analyze and interpret them.

Under this context the Wittgenstein Archives at the University of Bergen (WAB) was borne. Ludwig Wittgenstein (1889-1951) was one of the most important philosophers of the 20th century. His thinking influenced both psychology, music and architecture. His *Nachlass*[1] is composed of 20.000 pages of manuscripts and type scripts and it documents his philosophical work from 1913 until 1951.

The WAB is known among other things for the digitization of Wittgenstein's *Nachlass*[14]. The 20.000 pages composing the *Nachlass* can be found either as text editions in XML format or as facsimiles in JPG format. The digitization was a big step in opening the philosophers' work to the world.

The WAB works in cooperation with the Center for Information and Speech Processing (CIS) at the Ludwig-Maximilians-Universität in Munich on the project Wittgenstein Advanced Search Tools (WAST). The aim of the project is to work with concepts and tools of the computational linguistic's field on Wittgenstein's *Nachlass*. In particular, to make several linguistic analyses on it and to make it accessible and searchable. This thesis is being developed as a way of examining the *Nachlass*.

The search engine for Wittgenstein's *Nachlass* is called WiTTFind and it is developed by the CIS under the direction of Dr. Maximilian Hadersbeck. WiTTFind provides different possibilities for exploring the philosopher's *Nachlass*, such as rule-based search and semantic search. This thesis aims to improve the semantic search by creating word fre-

[1] The German word Nachlass is used to describe the collection of notes, manuscripts and so on left behind by a scholar after his death

quency lists for the *Nachlass*.

Many tools of the FinderApp use frequency lists previously calculated to process or display data. Some of these frequency lists were generated at the time when only 5.000 out of the 20.000 pages of the *Nachlass* were open source. Consequently, the frequencies found on them are not up to date. The frequencies shown by the search bar in WiTTFind are calculated differently and encompass the whole Nachlass. This results in a big discrepancy between both and in the need to fix that.

This thesis aims to create semantic frequency lists that are dynamic. This means that they can be recreated when the input files change or as new input comes in. To do so, the programs that generate them have to follow the structure used to process the data in WiTTFind. This results in a necessary work with makefiles and scripts. There are many semantic categories that could be explored throughout Ludwig Wittgenstein's works. Because of the time constraint of the thesis, it was decided that only frequency lists for the categories of color and music should be created.

The text editions of Wittgenstein's *Nachlass* are saved in XML format and follow the Text Encoding Initiative (TEI) guidelines. This is why the second chapter offers a short introduction to XML and its elements as well as to the TEI project.

In the third chapter the deploy chain of the FinderApp WiTTFind is described. To understand how the process works, a simple example of a makefile is shown. The chapter then describes what POS-Tagging is and how it is done for the FinderApp.

Some of the pages of the *Nachlass* are written in English. Because of this, certain difficulties arise for analyzing the data and creating the semantic frequency lists. In the fourth chapter these problems are analyzed and then some ideas on how to solve them are presented.

In the fifth chapter the creation of the frequency lists is explained. In order to generate the lemmatized frequencies, a semantic lexicon is used. As mentioned above, the semantic frequency lists should be created for the categories of colors and music, after showing how the lexicon works, the programs to create the frequencies are explained. What follows is a comparison between the just generated frequencies and the ones belonging to the first 5.000 pages of the *Nachlass*.

The evaluation of the frequency lists obtained in chapter 5, is done in the sixth chapter. The errors found during this process are also described in this chapter.

In the last chapter this thesis is concluded with a summary and some final observations.

2. XML and TEI

This chapter offers a short introduction to the markup language XML and to its elements and attributes. It also introduces the reader to the Text Encoding Initiative and finishes by giving an overview of the structure of the text edition of Ludwig Wittgenstein's *Nachlass*.

2.1. Short introduction to XML

The need to work with texts and documents in digital form led to the creation of the standard generalized markup language (SGML) in the 1960s. It was unofficially used until 1986 when it was officially defined. The specifications of SGML were rather complex and therefore the range of users was scarce. Nevertheless HTML and other data formats were initially created for SGML.

The inventors of the extensible mark-up language (XML) wanted to simplify SGML by decreasing many of the features that had made the use of SGML difficult. XML was standardized by the World Wide Web Consortium (W3C) in 1998. It is widely used to store and transport data and as a mark-up language it describes tree structures and it aims to distinguish between annotations in a text and the text itself. The data set that is obtained is generally called a document [6].

Tags are introduced into a document to organize and label it for machine processing [2]. One of the things that makes XML special is that the set of tag names and how they are used is not predefined. This means that users can create their own schemas and that applications can define their own data format. At the same time, general software tools can manipulate the different XML documents.

2.1.1. XML syntax and rules

Elements, attributes, entities and name spaces are the most important components of an XML document.

A tree structure is created by inserting between fragments of text opening and closing tags that are balanced the same way parentheses are [6, p. 1]. XML is not tolerant to errors, and if a tag is unbalanced, the parser won't be able to process the document correctly. There is a list of rules that has to be followed in order for an XML document to be well formed. These rules also narrow down the way in which a document is syntactically constructed:

1. All XML documents must have a root which is the parent of all other elements

2. All elements must have a closing tag (it doesn't matter if they have other elements, text or if they are empty). The closing tag can be recognized because it has a slash / in it:
   `<p>Text for paragraph element</p>`
   `</br>`

3. The tags are case sensitive

4. Proper nesting is essential:
   `<text><body>...</body></text>`

5. All attribute values must be inside quotes:
   `<lb rend="hl"/>` [15]

XML documents can optionally have a prolog, and if present, it comes in the first line of the document, see listing 2.1. The prolog also indicates which is the standard version to which the XML document conforms to. In the example below, it is version 1.0. The character encoding for the document can be either specified in this element or the file can be saved with the encoding. UTF-8 is the default character encoding for XML documents.

```
1 <?xml version="1.0" encoding="UTF-8"?>
```

Listing 2.1: XML prolog

XML documents are based on elements which can contain text or other documents. They divide the document into parts which can be later used by a search engine. An element must have an opening and a closing tag.

In listing 2.2 there is one element `<editor>` which has text content (Edited by). The starting tag is `<editor>` and the closing tag is `</editor>`.

```
<editor>Edited by</editor>
```

Listing 2.2: Element <editor>

In listing 2.3 the element `<editor>` and the element `<orgName>` contain text . Attributes are there to expand the information of an element. They are inserted inside an element and they have a value. In the example below `rend` is an attribute of `<orgName>` and has the value http://wab.uib.no/.

```
<editor>
  Edited by
  <orgName ref="http://wab.uib.no/">
     Wittgenstein Archives at the University of Bergen (WAB)
  </orgName>
</editor>
```

Listing 2.3: Element <editor> has element <orgName>

Elements can also have no content. Such elements are often called empty elements and there are two ways of writing them, look at listing 2.4. The element `</lb>` produces a line break. The XML documents containing Wittgenstein's *Nachlass* use the form on the second line for empty elements.

```
<lb></lb>
<lb/>
```

Listing 2.4: Empty element <lb/> can be written either way

In XML there are some characters that have a special meaning and cannot be used as text. An example for this is the symbol "&". The line in listing 2.5 throws an error since it contains this special character.

```
<s>Think of the several ways of distinguishing different kind of pieces in
    the game of chess (e.g., pawns & \officers).</s>
```

Listing 2.5: Special characters in XML

Entities are placeholders for content and they are there to avoid ambiguity while using some symbols. In XML there are 5 pre-defined entity references, see table 2.1. Other entities can be declared once and can be used in almost any part of the document[9, p. 51].

entity reference	symbol
<	&
"	"
'	'
<	<
>	>

Table 2.1.: Entity references

2.1.2. lxml.etree

For almost all programming languages there is a XML-Parser. The `lxml.etree` library for Python was used during this work to parse the XML files. The library allows not only to parse an XML document, but also to change it or to create a new one. In chapter 4 and in chapter 5 `lxml.etree` is used to iterate through the XML documents.

In what follows, some of the different functions of `lxml.etree` will be described. These functions together with others enable the creation of a document with a hierarchical tree structure.

The library can be imported into a python file with the following command:

```
from lxml import etree
```

- `Element()`: is the principal container object for the API of ElementTree. Most of the XML Tree functionality can be accessed through it.
  `root = etree.Element("root")`

- `append()`: creates a child element and adds it to a parent element.
  `root.append(etree.Element("child"))`

- `SubElement`: is like the `Element()` factory. It requires the parent as a first argument and the child to be created as the second one. It can be use instead of `append()`.
  `child = etree.SubElement(root, "child")`

Elements are list. Their sub elements can be retrieved in the same way elements in a Python list are:
`child = root[0]`

The class `iterparse` from `lxml.etree` parses XML into a tree and creates tuples of the form (event, element). The `element` part of the tuple is the element that the parser found opening or closing [8, p. 423]. The tag of the element can therefore be accessed through `element.tag`. Element attributes can be found with the method get, which takes as parameter the attribute name, e.g. `element.get("name_of_attribute")`. If the element contains text, it can be obtained with `element.text`.

2.2. Short introduction to TEI

The Text Encoding Initiative (TEI) is one of the most influential projects in the field of Digital humanities. The scholarly community created it with the purpose of managing in a digital way different types of data like manuscripts, source text, archival documents and so on. This community is also in charge of maintaining the project. What differentiates TEI from other word processors is that it focuses on the text itself rather than how to display it. TEI documents are expressed using XML and as such, they are also software independent, which means the TEI XML document's information that is captured looks the same to the different softwares using it. See Burnard's book [2] for more information about the TEI project.

The TEI has hundreds of tags and provides rules about how they can be combined. But most of the documents conforming to the TEI use only a small subset of all the possible components.

The University of Bergen is a member of the Text Encoding Initiative. This is why the text edition of the Wittgenstein Source Bergen *Nachlass* Edition (BNE) in XML follow the TEI guidelines. The XML documents that were specially prepared for the CIS, called CISWAB, have some TEI elements that were specially adapted for them.

In what follows, the most frequently used TEI XML elements for the text edition of the *Nachlass* will be presented together with the concrete attributes used by them. Most of the examples are in English and come from the Ts-310_OA_Norm.xml document, but there are other examples that are in German and come from different documents of the *Nachlass*.

2.2.1. <text> element

A `<text>` is typically something like an article or a book. In the case of the BNE, it is either the digital transformation of the manuscripts or of the type scripts of Ludwig

Wittgenstein's *Nachlass*.

`<text xml:lang="en">`: the atributte `xml:lang` is used to denote the language of the file. In the CISWAB documents only the English files have the optional attribute.

2.2.2. <ab> element and concrete attributes

The element `<ab>` represents an abstract block and it acts like an anonymous container for phrases or other arbitrary units of text. In BNE this element denotes a paragraph.

`<ab n="..." ana="...">`: n refers to the coordinates in the facsimile and ana (analysis) to the paragraph number.

Below, in listing 2.6, the typical structure of an `<ab>` element is shown. The elements contained by `<ab>` are explained in what follows. A paragraph can have one or more sentences. The example below has only one for simplification reasons but in the actual XML document it is composed of several sentences.

```
<ab n="Ts-310,1[2]et2[1]" ana="abnr:2">
  <seg type="publ">
    <s n="Ts-310,1[2]et2[1]_1" ana="facs:Ts-310,1 abnr:2 satznr:4">
      Suppose a man described a game of chess, without mentioning<lb/> the
      existence and operations of the pawns.
    </s>
  </seg>
</ab>
```

Listing 2.6: Typical structure of <ab> element in BNE (from Ts-310,1[2G]et2[1]_1)

2.2.3. <choice> and concrete attributes

The element `<choice>` groups alternatives for a passage in the text. In the example listing 2.7 the word "We" can be either followed by "treat" or by "regard".

```
We <lb/>
<choice type="s">
  <seg n="s_alt1">treat</seg>
  <seg n="s_alt2">regard</seg>
</choice> the case as though the child already possessed a language...
```

Listing 2.7: <choice> element (from Ts-310,43[2]et44[1]et45[1]_12)

2.2.4. <seg> element and concrete attributes

The element `<seg>` represents a segment and it describes a part of a text below a chunk-level.

`<seg n="s_alt...">`: attribute n for `<seg>` look at `<choice>` above.

`<seg type="stripped">`: the value stripped indicates that the editors of WAB omitted some information for the CISWAB document, see listing 2.8 for an example.

```
<seg type="stripped">)</seg>
```

Listing 2.8: <seg> with attribute value stripped (from Ts-310,5[2]_8)

`<seg type="notation">`: the attribute value notation links images, an example can be found in listing 2.9.

```
In others according to such schemes as:
<seg type="notation"
 corresp="http://wab.uib.no/cost-a32_fax/bmp/notatio310-22b.BMP">notatio310
    -22b.BMP
</seg>
```

Listing 2.9: <seg> with attribute value notation (from Ts-310,21[2]et22[1]_6)

`<seg type="date" n="..." part="N">`: the attribute value date comes up when a date is mentioned. The example in listing 2.10 is in German and its translation is "that she wrote on 20.08".

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

Listing 2.10: <seg> with attribute value date (from Ms-101,27v[2]_7)

2.2.5. <s> element and concrete attributes

The element `<s>` stands for sentence and it entails a sentence in a text.

`<s n="..." ana="..." abnr="..." satznr="...">`: n describes the coordinates in the facsimile while the attribute ana (analysis) provides additional information that mentions the facsimile name. The paragraph number is saved in the attribute abnr and the sentence number in satznr.

2.2.6. <lb> element and concrete attributes

The element `<lb>` represents a line break and it marks the beginning of a new typographic line in a particular edition or version of a text.

`<lb rend="shyphen"/>`: The element `<lb>` can have an optional parameter `rend`. The attribute rendition marks how the existing element appears in the source code. In this case, the attribute value shyphen (hyphenation) indicates that the line break separated a word. The example below, see listing 2.11, shows the use of the element `<lb>` with and without attribute.

```
<s n="Ts-310,1[2]et2[1]_14" ana="facs:Ts-310,1 abnr:2 satznr:17">Part of
    this train<lb rend="shyphen"/>ing is that we point to a building stone,
    direct the attention<lb/> of the child towards it, &amp;amp; pronounce a word
    .</s>
```

Listing 2.11: <lb> element with and without attribute (from Ts-310,1[2]et2[1]_14)

`<lb rend="shyphen0"/>`: indicates that the syphen separetes a word but that the dash of the separation is not present. An example of how a facsimile that has this kind of separation looks like can be found in figure 2.1 and its text representation can be seen in listing 2.12.

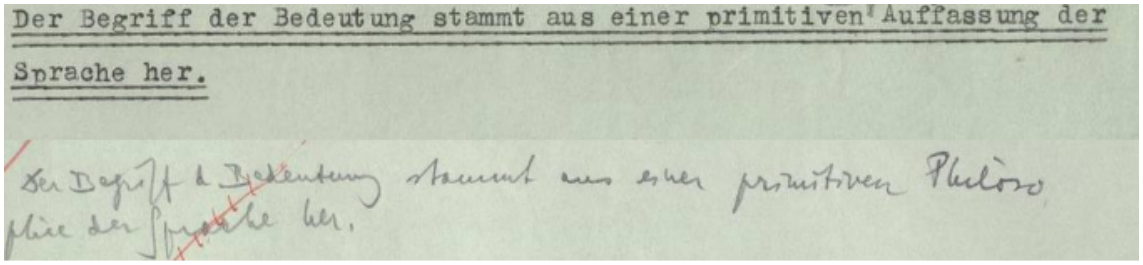


Figure 2.1.: Sentence in Ts-310

```
Der Begriff der Bedeutung stammt aus einer primitiven Philoso<lb rend="
    shyphen0"/>phie der Sprache her.</seg>
```

Listing 2.12: <lb rend

2.2.7. <pb> element and concrete attributes

The element `<pb>` stands for page break and it marks the beginning of a new page in the document.

`<pb facs="..."/>`: the attribute facs shows the facsimile to which the page belongs to. An example of its use can be found below in listing 2.13.

```
When we make use of a pattern we compare something with it, e.g., <pb facs="
    Ts-310,12"/> a chair with the picture of a chair.
```

Listing 2.13: <pb> element with attribute (from Ts-310,11[3]et12[1]et13[1]et14[1]_6)

2.2.8. <abbr> element and concrete attributes

The element `<abbr>` contains an abbreviation of any form.

`<abbr type="abb">`: the attribute value abb signals a standard abbreviation, look at the example beneath (listing 2.14).

```
<abbr type="abb">etc.</abbr>
```

Listing 2.14: <abbr> with attribute and value abb (from Ts-310,24[3]et25[1]et26[1]et27[1]_9)

`<abbr corresp="...">`: the attribute corresp shows that the abbreviation is not standardized. In Listing 2.15 there is an example of a non standard abbreviation for the German word Stimmung (mood).

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

Listing 2.15: <abbr> with attribute corresp (from Ms-101,1r[1]_49)

2.2.9. <emph> and concrete attributes

The element `<emph>` emphasizes words or phrases that have to be marked due to a linguistic or rhetorical purpose.

`<emph rend="us1">`: this attribute value makes an emphasis with a straight underline, look at the following German example (one has to understand a command) in listing 2.16.

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

Listing 2.16: <emph> with attribute value us1 (from Ms-115,230[3]_1)

Some of the other types of attribute values used for `<emph>` in the CISWAB documents are:

- `<emph rend="us1_c">`: canceled accent with straight underline
- `<emph rend="us1_h">`: emphasize with straight underline and other styling
- `<emph rend="us1_h_ch">`: canceled accent emphasized with straight underline and other styling
- `<emph rend="usb">`: emphasis through underline and line break
- `<emph rend="us2">`: emphasis with two underlines

2.3. Text edition of Ludwig Wittgenstein's *Nachlass*

The *Nachlass* of Ludwig Wittgenstein is composed of manuscripts (Ms) and type scripts (Ts) and it is 20.000 pages long. Most of the original documents are privately owned and are spread throughout Europe and Canada. The Trinity College Library in Cambridge owns the biggest part of the documents [4]. The Wittgenstein Archive from the University of Bergen in Norway copied and transcribed all the *Nachlass* and made it open access. It can be found in the Wittgenstein Source Bergen *Nachlass* Edition Website.

The work was digitized in XML format and three different data types were created for each text edition of the manuscripts and of the type scripts. The OA.xml files contain the text together with the exact suggestions for corrections that Wittgenstein wrote on them. The OA_DIPLO.xml files attempt to describe with help of XML elements the details of the original document as precise as possible . The OA_NORM.xml are a good readable version of the *Nachlass* from the perspective of the editors and experts. This data type is comparable with the author's last will and could possibly be printed as a book.

Fig. 2.2 shows the correction made in one of the sentences in the type script 310. The XML code in listing 2.17 represents the sentence in the file Ts-310_OA.xml. It varies from the code for that sentence in the Ts-310_OA_DIPLO.xml file, see listing 2.18 and the representation of it in the nomalized text Ts-310_OA_NORM.xml in listing 2.19.

```
<s type="es">Now it is
 <choice type="dsl_h">
```

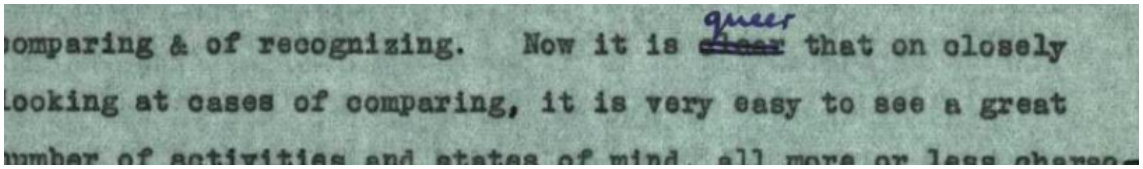


Figure 2.2.: Sentence in Ts-310

```
    <orig type="alt1">
      <del type="d_h">clear</del>
    </orig>
    <orig type="alt2">
      <add rend="i_h">queer</add>
    </orig>
  </choice>
  that on closely<lb/> looking at cases...
</s>
```

Listing 2.17: Ts-310_OA.xml fragment (from Ts-310,14[2]et15[1]et16[1])

```
<s n="Ts-310,14[2]et15[1]et16[1]_4" ana="facs:Ts-310,14 abnr:16 satznr:196">
    Now it is
  <choice type="dsl_h">
    <seg n="dsl_h_alt1">
      <del type="d_h">clear</del>
    </seg>
    <seg n="dsl_h_alt2">queer</seg>
  </choice>
  that on closely<lb/> looking at cases...
```

Listing 2.18: Ts-310_OA_DIPL.xml fragment (from Ts-310,14[2]et15[1]et16[1]_4)

```
<s n="Ts-310,14[2]et15[1]et16[1]_4" ana="facs:Ts-310,14 abnr:16 satznr:196">
  Now it is
    <seg type="stripped">
      queer
    </seg>
   that on closely<lb/> looking at cases...
</s>
```

Listing 2.19: Ts-310_OA_NORM.xml fragment (from Ts-310,14[2]et15[1]et16[1]_4)

The code presented in later chapters for the creation of the frequency lists works with the normalized (OA_NORM.xml) files.

3. Deploy chain and POS-Tagging

This chapter shows first how the deploy chain for WiTTFind works and how it can be initialized. Then it introduces the reader to what part of speech tagging is and explains how this is done with help of the TreeTagger for the files of Wittgenstein's *Nachlass*.

3.1. Deploy chain for the FinderApp WiTTFind

The FinderApp WiTTFind uses the data provided by the edition partners. The data needs to be processed and edited so that it can be used as the basis for the FinderApp. To do so, numerous [CW]AST tools were created and sorted into directories in the repository witt-data, which is the one in charge of processing the data for WiTTFind.

The preparation of the data is carried out by a deployment process that works with Continuous Integration (CI) and a central Makefile. This file includes all other makefiles in the sub directory `witt-data/deployment/make` with help of the command `include make/*.make`.

The order in which the different programs are called is important since some dependencies exist between them. The particular sequence in which the files must be called bares the name [CW]AST Toolchain and can automatically be called with the makefile target `make deploy`. The makefile target has to be called from the `witt-data/deployment` directory. The output of the different programs is then stored in directories specified in the principal Makefile.

The principal Makefile is a macro file in which different variables are defined that are used both in this file and by all other makefiles in the sub directory `witt-data/deployment/make`. If a new tool needs to be implemented, the developer has to think whether the variables should be declared globally or locally. This decision depends on whether this variables need to be accessed from other tools and files or not.

The first step of the Toolchain that leads to the output in the FinderApp WiTTFind is deciding if the XML files representing the type scripts and manuscripts should be expanded or not. A text is expanded when all the different reading possibilities annotated by the editors are displayed. This is done in the XML files with the element `choice`. The OA_NORM.xml and the OA_DIPLO.xml can be expanded with the makefile targets `make expand-norm-choices` and `make expand-diplo-choices`.

The next step of the chain is to create the tagged files. To do so, the TreeTagger has to be downloaded with the makefile target `make download-tree-tagger`. After the tagger is installed, the files can be tagged with `make tagged`. If the files are expanded, the tagging

is done with `make expanded-norm-tagged` or respectively `make expanded-diplo-tagged`.

The third step of the [CW]AST Toolchain is to create the frequency lists. The output of `make lemma_freqlist` are token and lemma lists which are saved inside the directory that contains the data to be exported to the FinderApp.

Because different tools from WiTTFind use sentence separated text for processing, the fourth step is to generate the sentence list. The makefile target `make sentence-list` takes care of this task.

The fifth step is to create with help of the makefile target `make document-ids` a file that contains all the document ids.

The sixth step is to convert the frequency list into json format with the makefile target `make convert-freqlist-json`.

The last step is to export the data created by copying all tagged input files and the sentence list created in the fourth step of the [CW]AST Toolchain into the export data directory. This is done with `make make export-data`. The last target to be called is `make export-converted-freqlist-json` which copies the json files generated in the sixth step into the same directory mentioned above.

3.2. Short introduction to makefiles

Make builds automatically executable programs and other non source files by reading a Makefile. A Makefile specifies how to obtain the target program. Make can also be used to manage projects where files have to be updated automatically when other dependent files change.

A make file has rules and targets. A rule, see figure 3.1, is there to tell Make how to carry out a sequence of commands to build a target file from source files. A target file can also have a list of dependencies, which contains all files that need to be use as input in the rule's command, see GNU Make documentation [5]. Makefiles are use in the deploy

```
target:   dependencies ...
          commands
          ...
```

Figure 3.1.: Simple rule from GNU Make documentation [5].

chain of WiTTFind and it is important to understand how they work. What follows is an example of a simple Makefile.

3.2.1. Makefile example

To start, in a directory there are the following files (see listing 3.1): `Makefile`, `1.xml` and `2.xml`.

```
$ ls
1.xml 2.xml Makefile
```

Listing 3.1: Terminal - files in directory before calling make

The complete Makefile for the example can be seen in listing 3.2[1]. The .xml files are found and saved into the variable `UNTAGGED_NORM_FILES` with the help of the shell command `find`, which searches for all files that end in .xml. Thanks to the grep's option -v, the files that include the string "-tagged" are ignored. `UNTAGGED_NORM_FILES` contains the files information and the file types and looks like this: "1.xml 2.xml".

The variable `TAGGED_NORM_FILES`, in line 6, is formed with help of the command:
`$(patsubst pattern,replacement,text)`
For this function the % represents a wild-card which means the command searches for all files that end with .xml. The text match by % in pattern is then replaced in % in the replacement, e.g. `$(patsubst %.xml,%-tagged.xml,1.xml)` produces the value `1-tagged.xml`, see GNU Make documentation [5] for more details. `TAGGED_NORM_FILES` value is "1-tagged.xml 2-tagged.xml".

The files have not been created yet, which means the output of the commando `ls` in the terminal is still the same as in listing 3.1.

The variable `SILENT` makes that the commando called after it doesn't get echoed, hence the name silent.

```
1  ## List of the files that should be created:
2  ##TAGGED_NORM_FILES=1-tagged.xml 2-tagged.xml 3-tagged.xml
3
4  ## Example using shell command find
5  UNTAGGED_NORM_FILES             = $(shell find -L . -type f -name \*.xml |
       grep -v '\-tagged')
6  TAGGED_NORM_FILES               = $(patsubst %.xml,%-tagged.xml, $(
       UNTAGGED_NORM_FILES))
7  SILENT                          ?= @
8
9  ## tagged is the rule that creates the files
10 ## dependencies: TAGGED_NORM_FILES
11 ## the rule is fulfilled when the dependencies are fulfilled ...
12 tagged: $(TAGGED_NORM_FILES)
13
14 ## a rule is needed to tell make how to fullfill the dependencies:
15 ## the rule is fulfilled when the dependencies are fulfilled ...
16 ## this happens when the .xml files are newer than the files of the rule (-
       tagged.xml)
17 ## the tagged file 1-tagged.xml will be executed, when 1.xml is newer than
       1-tagged.xml
```

[1] Makefile can be found attached in the SD Card

```
18 %-tagged.xml: %.xml
19 	$(SILENT) printf "execute: $< \n"
20 	$(SILENT) echo "calling touch for $@"
21 	touch $@
22
23 clean:
24 	rm $(TAGGED_NORM_FILES)
25
26 touch:
27 	touch $(TAGGED_NORM_FILES)
28
29
30 info:
31 	$(SILENT) echo "\n\n################################  make tagged  INFO"
32 	$(SILENT) echo "UNTAGGED_NORM_FILES = $(UNTAGGED_NORM_FILES)"
33 	$(SILENT) echo "TAGGED_NORM_FILES = $(TAGGED_NORM_FILES)"
```

Listing 3.2: Makefile

The rule `tagged` in line 12 has the dependencies `TAGGED_NORM_FILES` and it gets fulfilled when it dependencies do as well. This means this rules activates the pattern rule in line 18, which tells make how to make a *something*-tagged.xml from another *something*.xml file. Listing 3.3 shows the chain of events that happen when the makefile target `make tagged` is called. The function `touch` in line 21 creates the files (and not the one in line 27).

```
$ make tagged
execute: 2.xml
calling touch for 2-tagged.xml
touch 2-tagged.xml
execute: 1.xml
calling touch for 1-tagged.xml
touch 1-tagged.xml
```

Listing 3.3: Terminal - call "make tagged"

The output of the command `ls` after calling `make tagged` can be seen in listing 3.4. The files 1-tagged.xml and 2-tagged.xml were created.

```
$ ls
1.xml 2.xml Makefile
1-tagged.xml 2-tagged.xml
```

Listing 3.4: Terminal - files in directory after calling make

Calling the makefile target `make tagged` again will cause the output seen in listing 3.5. The -tagged.xml files already exist and the .xml files on which they are dependent on haven't changed, which means the rule doesn't need to update or create something.

```
$ make tagged
make: Nothing to be done for 'tagged'.
```

Listing 3.5: Terminal - call "make tagged"

On the other hand, if we remove for example the file 1.xml and touch it from the terminal again, `make tagged` will recreate 1-tagged.xml file, see listing 3.6. This happens because the 1.xml file has a newer time stamp than the stamp in the 1-tagged.xml file.

```
$ rm 1.xml

$ ls
2.xml Makefile 1-tagged.xml 2-tagged.xml

$ touch 1.xml

$ make tagged
execute: 1.xml
calling touch for 1-tagged.xml
touch 1-tagged.xml
```

Listing 3.6: Terminal - "make tagged" after deleting a file

The rule `clean` in listing 3.2 line 23 has no dependencies and removes all -tagged.xml files. After calling it, see listing 3.7, the output of the `ls` command in the terminal is the same as in listing 3.1.

```
$ make clean
```

Listing 3.7: Terminal - call "make clean"

The rule `touch` in line 26 has no dependencies, which means it can always create the -tagged.xml with the function touch. That is why calling `make touch` twice produces the same output, see listing 3.8.

```
$ make touch
touch ./2-tagged.xml ./1-tagged.xml

$ make touch
touch ./2-tagged.xml ./1-tagged.xml
```

Listing 3.8: Terminal - call "make touch"

The example above was used to create the new files 1-tagged.xml and 2-tagged.xml. For this example it would have been easier to type twice the commando touch in the terminal to create the files. But what happens when there are not 2 files but 50? This is one of the reasons why using Make is so practical. The code of the Makefile has to be written once, but then it calls its rules in a chain and checks if the dependencies have been updated or not. If they weren't, then the rule doesn't need to be processed, else the rule is called.

3.3. Short introduction to POS-Tagging

Marking a word in a text as a corresponding part of the speech is called Part-Of-Speech Tagging (POS-Tagging). The process used to be done by hand and is now one of the research areas in the field of computational linguistics. POS-Tagging can be done by a piece of software which takes into consideration the language of the text (corpus) as well as the definition of the word (token) and the context in which it appears. One of the reasons for assigning a word to a part of speech is to disambiguate its meaning.

Many areas of computational linguistics such as language identification, Named Entity Recognition (NER) and machine translation require removing uncertainty of meaning from a word. Different methods have been proposed for tagging the words with parts of speech.

Some of the systems are ruled based while others use probabilistic methods. The WiTTFind Tagger falls into the second category. To do semantic search in the FinderApp, the words in the *Nachlass* need to be tagged with parts of speech.

The next sections aim to explain the most important characteristics of the POS-tagger used in WiTTFind and how it works with the text transcripts in XML format of Wittgenstein's *Nachlass*.

3.4. The TreeTagger

The POS-tagger used in WiTTFind is called TreeTagger and it uses a decision tree to estimate the transition probabilities to mark a word with a part of speech.

The tagger was developed by PD Dr. Helmut Schmid, who currently teaches at the CIS, in the faculty for machine language processing at the University of Stuttgart.

Sparse data is a problem that affects many probabilistic classifications systems. It occurs when not enough data was recorded for training which leads to a worse classification since the system didn't have enough data to learn from. The TreeTagger is based on Markov Model, which accuracy of classification suffers when encountered with sparse data. The POS-tagger develop by Dr. Schmid avoids this problem by recursively generating decision trees from n-grams, see Schmid's paper [12] for a more detailed explanation.

The first version of the TreeTagger was most efficient when marking words in the English language. Its accuracy was tested in 1994 with the Penn-Treebank data and it tagged correctly 96.36% of the words [12]. The TreeTagger accuracy was not that good for German and other languages due to a lack of tagged corpora to train the tagger for these other languages. In 1995 Dr. Helmut Schmid published a new paper in which he proposed a new method to achieve better accuracy with little training data. For the German language the improvements made on the original TreeTagge meant a reduction of the errors by more than a third. For more details on the improvements of the tagger see Schmid's paper of 1995 [13].

The TreeTagger determines for each token the most probable tag and lemma. The lemma can only be determined if it is part of the lexicon that the tagger is using. A special lexicon was created for WiTTFind and has been further developed throughout the years. Some words are not in the lexicon and this causes the tagger to mark the lemma as `UNKNOWN`.

For each word, the TreeTagger saves a line with the word in first place, followed by a tab and then by the part of speech tag, a space and then the lemma. See 3.2 for an example of the output for the sentence "The TreetTagger is easy to use.".

The STTS tagset

The University of Stuttgart and the University of Tübingen annotated manually German Text corpora in order to create the Stuttgat-Tübingen-Tag (STTS). This tag set is hier-

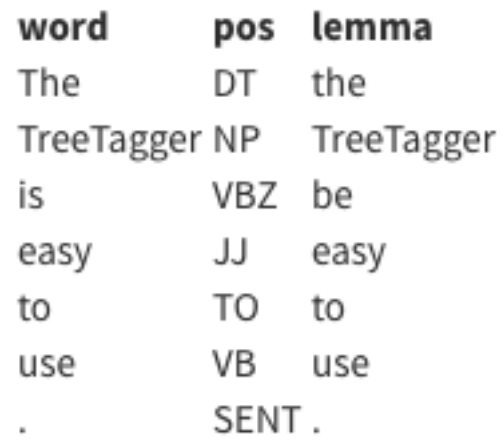


Figure 3.2.: Output example of the TreeTagger from Dr. Schmid [11].

archical. Each tag is composed of a self explanatory sequence of letters that is read from left to right, the main word class coming first followed by the subclass, see Schiller, Teufel and Stöckert [10] for more information about the STTS.

Due to the fact that most of Ludwig Wittgenstein *Nachlass* is written in German, the tagger uses the STTS tag set. The main type of words and their abbreviations are shown in table 3.1.

POS	**Description**
N	Nouns
V	Verbs
ART	Articles
ADJ	Adjectives
P	Pronouns
CARD	Cardinal numbers
ADV	Adverbs
KO	Conjunctions
AP	Adpositionen
ITJ	Interjections
PTK	Particles
FM	Foreign language material

Table 3.1.: Main POS in STTS [10]

3.4.1. Download of the TreeTagger

The TreeTagger can be directly downloaded from the links in the TreeTagger tools page (http://www.cis.uni-muenchen.de/ schmid/tools/TreeTagger/) from Dr. Schmid. The executable code for the installation of the TreeTagger depends on the operating system and the tagger can be installed on PC-Linux, ARM-Linux, Mac-OS and Windows systems. In this website there are also various language files to be taken as the parameters for the

TreeTagger.

For the WiTTFind App the download of the tagger is done with the help of makefiles and shell scripts. As mentioned in section 3.1, WiTTFind works with CI and the download of the TreeTagger is the second step of the [CW]AST Toolchain.

The makefile target `make download-tree-tagger` calls the shell program download-tree-tagger.sh, which can be found in the directory `witt-data/ciswab/tools/tree-tagger`. The program takes as a parameter the directory in which the TreeTagger should be installed.

The url from which the files should be downloaded is defined in line 1 of listing 3.9. All the files required for the installation are defined in the lines 5 to 11. The download of the TreeTagger is configured for Linux. The actual download of the files occurs from line 31 to 34 with help of `curl`.

```
1  BASE_URL=http://www.cis.uni-muenchen.de/~schmid/tools/TreeTagger
2  DOWNLOAD_URL=${BASE_URL}/data
3  INSTALL_DIR=$1
4
5  FILES="tagger-scripts.tar.gz \
6         install-tagger.sh \
7         english-par-linux-3.2-utf8.bin.gz \
8         german-par-linux-3.2-utf8.bin.gz \
9         english-chunker-par-linux-3.2-utf8.bin.gz \
10        german-chunker-par-linux-3.2-utf8.bin.gz \
11        tree-tagger-linux-3.2.1.tar.gz"
12
13 echo "Before installing, please read TreeTagger's LICENSE first [OK]"
14
15 read
16
17 curl --silent ${BASE_URL}/Tagger-Licence | ${LESS}
18 echo "Do you agree to TreeTagger's license, i.e. you don't use TreeTagger
       commercially? (Yes/No/Whatever)"
19
20 read answer
21
22 case $answer in
23 y|Y|Yes) : ;;
24 n|N|No) exit 1 ;;
25 * ) exit 1 ;;
26 esac
27
28 echo "Creating install dir ${INSTALL_DIR}"
29 mkdir -p ${INSTALL_DIR} && cd $_
30
31 for file in $FILES
32 do
33   curl -LO ${DOWNLOAD_URL}/$file
34 done
```

Listing 3.9: Section of code in download-tree-tagger.sh

The commando `make download-tree-tagger`can be called from other operating systems and it doesn't throw an error, which can be miss leading. First the calling of the next step of the [CW]AST Toolchain, `make tagged`, throws an error in other operating systems since the TreeTagger was not properly installed.

The modifications done to the program `download-tree-tagger.sh`, see listing 3.10, lead to a correct installation of the TreeTagger for Linux and Mac. The variable unameOut saves the output of the command `uname -s`, which prints the operating system's name. Depending on the operating system, the machine variable is set in lines 2 to 8 to either "linux" or "MacOSX". Since the version for Linux requires an extra number, the variable appendNr for this system is set to "1." while for Mac the value is set to an empty string. For other operating systems, the program breaks and ask the user to download the TreeTagger manually. The `FILES` variable is now set by interpolating machine and appendNr to the strings[2]. The rest of the process is analogous to listing 3.9.

```
1  unameOut="$(uname -s)"
2  case "${unameOut}" in
3      Linux*)     machine=linux;
4  ____appendNr="1.";;
5      Darwin*)    machine=MacOSX;
6  ____appendNr="";;
7      *)          { echo 'Operating System uknown, download tagger from http
       ://www.cis.uni-muenchen.de/~schmid/tools/TreeTagger' ; exit 1; };;
8  esac
9
10 FILES="tagger-scripts.tar.gz \
11       install-tagger.sh \
12       english-par-${machine}-3.2-utf8.bin.gz \
13       german-par-${machine}-3.2-utf8.bin.gz \
14       english-chunker-par-${machine}-3.2-utf8.bin.gz \
15       german-chunker-par-${machine}-3.2-utf8.bin.gz \
16       tree-tagger-${machine}-3.2.${appendNr}tar.gz"
17
18 echo "Before installing, please read TreeTagger's LICENSE first [OK]"
```

Listing 3.10: Section of code modified download-tree-tagger.sh

3.4.2. Workflow of the TreeTagger for the OA-NORM.xml files

The FinderApp WiTTFind uses the TreeTagger to tag the tokens in Wittgenstein's *Nachlass*. The makefile target `make tagged` has a dependencies the `TAGGED_NORM_FILES` and it calls the pattern rule in line 6, see listing 3.11, which tells make how to create a *something*OA_NORM-tagged.xml file from a *something*OA_NORM.xml document. The process is analogous to the one shown in the simple Makefile example in 3.2.1. Instead of creating empty files with `touch`, the target calls the shell script `process.sh` in line 7 and passes it two directories as well as the OA_NORM.xml files (passed with $<).

[2]The modified program can be found in the attached SD Card.

```
1  SHELL         = /bin/bash
2  SILENT        ?= @
3
4  UNTAGGED_NORM_FILES = $(shell find -L $(\textit{Nachlass}_DIR)/*/norm -type
       f -name \*.xml | grep -v '\-tagged')
5  TAGGED_NORM_FILES   = $(patsubst %.xml,%-tagged.xml, $(UNTAGGED_NORM_FILES))
6
7  tagged_data: $(TAGGED_NORM_FILES)
8
9  %-tagged.xml: %.xml
10 	$(SILENT) $(SHELL) $(TREETAGGER_DIR)/process.sh $(TREETAGGER_DIR) $(
       TAGGED_DIR) $<
11
12 tagged: tagged_data
```

Listing 3.11: Fragment of tagged.make file[3]

The shell script, `process.sh`, iterates over the files OA_NORM.xml and generates a new tagged file with the ending OA_NORM-tagged.xml for each one of them, see loop in figure 3.3 line 1.

After the program is done, the files are tagged. The examples in listing 3.12 and 3.13 aim to show the difference between a sentence of the *Nachlass* before and after it was tagged.

```
<s n="Ts-213,IIIr[1]_1" ana="facs:Ts-213,IIIr abnr:47 satznr:111">
  42) Kann man etwas Rotes nach dem Wort "rot" suchen? braucht man ein Bild
  <lb/> dazu?
</s>
```

Listing 3.12: Sentence in Ts-213_OA_NORM.xml (s-213,IIIr[1]_1)

```
<s n="Ts-213,IIIr[1]_1" ana="facs:Ts-213,IIIr abnr:47 satznr:111">
    <w t="CARD" l="@card@">42</w>
    <w t="$(" l=")">)</w>
    <sp/>
    <w t="VMFIN" l="k{\"o}nnen">Kann</w>
    <sp/>
    <w t="PIS" l="man">man</w>
    <sp/>
    <w t="PIAT" l="etwas">etwas</w>
    <sp/>
    <w t="NN" l="Rot|Rote">Rotes</w>
    <sp/>
    <w t="APPR" l="nach">nach</w>
    <sp/>
    <w t="ART" l="die">dem</w>
    <sp/>
    <w t="NN" l="Wort">Wort</w>
    <sp/>
    <w t="$(" l="''">"</w>
```

[3]For readability purposes, lines 1 to 3 were added to tagged.make. This lines are in the macro Makefile. For the original code of tagged.make and Make, see attached SD Card (code was extracted from witt-data repository and was not modified).

```
    <w t="ADJD" l="rot">rot</w>
    <w t="$(" l="''">"</w>
    <sp/>
    <w t="VVINF" l="suchen">suchen</w>
    <w t="$." l="?">?</w>
    <sp/>
    <w t="VVFIN" l="brauchen">braucht</w>
    <sp/>
    <w t="PIS" l="man">man</w>
    <sp/>
    <w t="ART" l="eine">ein</w>
    <sp/>
    <w t="NN" l="Bild">Bild</w>
    <lb/>
    <sp/>
    <w t="PAV" l="dazu">dazu</w>
    <w t="$." l="?">?</w>
  </s>
```

Listing 3.13: Sentence in Ts-213_OA_NORM-tagged.xml (from s-213,IIIr[1]_1)

To process the OA_NORM.xml files `process.sh` calls in its main part of the function other programs, see 3.3[4] The files need to be preprocessed before the words in them can be tagged. This occurs between the lines 2 and 8. In line 5 the script `extract-text.perl`

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

Figure 3.3.: Section of code in process.sh by Dr. Schmid

is called, which removes the header of the XML document and leaves only the actual text of the *Nachlass* to be processed. To distinguish between the spaces of the words in the the actual text and the spacing inside the XML tags, this program inserts between each word in the text the element `</sp>` separeted by spaces.

In line 7 the program calls the script `tokenize-german-utf8` which in turns activates other programs. The spaces within the XML tags are replaced by the symbol "ð" in the

[4]The complete original program extracted from witt-data repository can be found in the attached SD Card.

script `filtergerman-tokin-utf8.perl` found witt-data repository. After this step the `<s>` element in listing 3.12 will look like listing 3.14.

```
<sðn="Ts-213,IIIr[1]_1"ðana="facs:Ts-213,IIIrðabnr:47ðsatznr:111">
  42) <sp/> Kann <sp/> man <sp/> etwas <sp/> Rotes <sp/> nach <sp/> dem <sp
   /> Wort <sp/> "rot" <sp/> suchen? <sp/> braucht <sp/> man <sp/> ein <sp/>
    Bild<lb/> <sp/> dazu?
</s>
```

Listing 3.14: Sentence in Ts-213_OA_NORM.xml after preprocessing spaces (from s-213,IIIr[1]_1)

In the next step, the text is split on punctuation symbols and on the new inserted element `<sp>`. After this, the "ð" is deleted from inside the tags and replaced by a normal space. Each token and `<sp>` occupies now a line, see figure 3.15.

```
<s n="Ts-213,IIIr[1]_1" ana="facs:Ts-213,IIIr abnr:47 satznr:111">
  ...
  <sp/>
  nach
  <sp/>
  dem
  <sp/>
  Wort
  "
  rot
  "
  ...
</s>
```

Listing 3.15: Sentence in Ts-213_OA_NORM.xml after deleting "ð" (s-213,IIIr[1]_1)

The tagging occurs in line 9 of `process.sh` and it is followed by the post-processing steps (line 10-12). To create the new OA_NORM-tagged.xml file, the header that was originally taken off in the pre-processing step, is added to the file again with help of regular expressions. This happens in the program `generate-xml-output.perl` in line 12. The tokens are now inside the XML element `<w>` which has two attributes: l (for lemma) and t (for token). The output for an `<s>` element was already shown in listing 3.13.

4. German-English files and unknown words

The FinderApp WiTTFind uses the tagged files to create frequency lists. There are two main reasons why the semantic frequency lists are incomplete. The first one is that the lemma of some of the words, as mentioned in chapter 3.4, are tagged as `UNKNOWN`. The second reason is because in order to create semantic frequency list the POS-tag of the word is necessary. The TreeTagger is configured for the German language. This is because most of Wittgenstein's *Nachlass* is written in German. But the Austrian philosopher studied and then taught at the University of Cambridge, where he also interchanged ideas and thoughts with other professors and students. It is therefore not surprising that some of the files that make up his *Nachlass*, such as the Ts-310, are written in English. Other files have passages in both languages. The TreeTagger marks the tag of the English words as "FM" (foreign language material as defined by the STTS) and for this reason it is not possible to know to which part of speech they belong to. See listing 4.1 for an example of an English tagged sentence.

Another consequence of the TreeTagger language setting is that some English words are treated as German words. In the same example, listing 4.1, the English word "such" is read as a German word by the TreeTagger. Its lemma is set to "suchen" (German word for search) and its tag is set to "VVIMP", which in STTS stands for imperative, main verb.

```
s n="Ts-310,1[1]_3" ana="facs:Ts-310,1 abnr:1 satznr:3">
<w t="ITJ" l="he">He</w>
<sp/>
<w t="FM" l="do">does</w>
<sp/>
<w t="PTKVZ" l="not">not</w>
<sp/>
<w t="FM" l="primarily">primarily</w>
<sp/>
<w t="FM" l="think">think</w>
<sp/>
<w t="FM" l="of">of</w>
<sp/>
<w t="VVIMP" l="suchen">such</w>
<sp/>
<w t="FM" l="word">words</w>
...
</s>
```

Listing 4.1: English tagged sentence (from Ts-310,1[1]_3)

In order to achieve better and more accurate results when creating the semantic frequency lists, the Tagger should be calibrated in the future depending on the language of the file. In order to do so, the language of the files has to be known.

To recognize the language of a file, the program `language_finder.py` was created. This program serves two purposes. On one hand, it recognizes whether a file is mostly written in German or in English. On the other hand it searches for all words which lemmas are `UNKNOWN`. This way the unknown words can be added if needed to the lexicon used by the FinderApp WiTTFind. Expanding the lexicon will ensure that more words will be tagged correctly in the future.

What follows is an explanation of how the program `language_finder.py` searches for `UNKNOWN` words and how it decides what is the language of a file.

4.1. Make deploy chain for the language finder

The `language_finder.py` can be called by the makefile target `make language finder`, which has no dependencies. The program `language_finder.py` receives as argument the OA_NORM-tagged.xml files. The `language_finder.make` and `language_finder.py` files can be found in the attached SD Card.

4.2. Unknown words

At the beginning of this work it was believed that the lemma of the English words was tagged as `UNKNOWN`. By creating two counters, one for the words with `UNKNOWN` lemma and one for all the words, one could find out whether a file was written in German or in English by comparing the two counters. If a file had more unknown words than known, it was believed to be written in English. Otherwise in German.

It was then established that all files had more known words than unknown ones although the existence of the English file TS-310 was not unfamiliar. The lemmas of the English words were also being tagged correctly. This findings created a more in depth investigation of the way in which the TreeTagger works. In this search it was found, that one of the parameters given to the tagger is the aux-lex.txt, see figure 3.3 line 10.

The aux_lex.txt file contains in each row an entry for a word. The entry is composed of a word's full form, followed by a tab. Then the tag (which for almost every word is set to "FM") comes followed by a space and at the end comes the lemma of that word. See 4.2 for some entry samples. The TreeTagger uses the additional lexicon to tag the English words.

```
...
Always_FM always
Among_FM among
Austerlitz_NE Austerlitz
Amongst_FM amongst
```

```
Analogously FM analogously
...
```

Listing 4.2: Entires in aux-lex.txt

The words tagged as `UNKNOWN` are still causing the frequency list to be imprecise. For this reason, it was decided to expand the lexicon by creating a list of unknown words and later adding them to it.

Some of the tokens considered unknown have entries in the lexicon but the TreeTagger marks them as `UNKNOWN` because they are being normalized incorrectly in the preproccesing step.

One of the known problems in the normalization of the words is caused by the sometimes arbitrary positioning of the XML element `<seg>`. This element can be used to suppress some information which isn't relevant for the developers of the FinderApp and can be found inside or outside a word, as well as inside a sentence. A good description of this edition problem is done in Faridis thesis [1]. An example where a word is split incorrectly in the prepossessing step can be found in listing 4.3. As a consequence, the word (or a part of it) is lemmatized as `UNKNOWN` from the TreeTagger, see listing 4.4.

```
<seg type="stripped">O</seg>rganisches
```

Listing 4.3: Problematic <seg> (from Ms-114,125r[1]_2)

```
<ab n="Ts-310,1[2]et2[1]" ana="abnr:2">
<seg type="stripped">
  <w t="NN" l="O">O</w>
</seg>
<w t="ADJA" l="UNKNOWN">rganisches</w>
```

Listing 4.4: Wrongly normalized word because of <seg type"stripped"> (from Ms-114,125r[1]_2)

Other cases in which a word's lemma has been tagged as `UNKNOWN` have been discussed in previous bachelor theses and will therefore not be repeated in this one. The example above was to show that the problem of wrongly done lemmatization comes not only from lacking entries in the lexicon, but also when words are split wrongly.

The program `language_finder.py` creates for all the OA_NORM.xml files a dictionary of words which lemmas are `UNKNOWN`. The key of the dictionary `unknown_words_dict` is the full form of the word and its value is an array which elements are the name of the files in which the word appears.

To create the dictionary, the program parses each file into a tree with the help of `iterparse` from `lxml.etree`. It searches for the elements which tags are "w" (for word). Then it looks for the attribute "l" (for lemma), and if its value is "UNKNOWN", a series of steps begin to extract the word and added to the dictionary, see listing 4.6 for a section of the code. The whole program can be found in the attached SD Card. Some of the words that are marked as `UNKNOWN` are not really tokens, but rather mathematical formulas written by

Wittgenstein. They should not be added to dictionary of unknown words and are ignored in line 2 in listing 4.6.

Other tokens are marked as `UNKNOWN` because they are separated by a "-" inside, see listing 4.5. It was decided that if a word contains a dash which is followed by a lowercase letter, the dash should be removed in order to obtain a "clean" word. This is done in line 6 of listing 4.6.

```
<w t="NN" l="UNKNOWN">Ab–nehmen</w>
```

Listing 4.5: Dash inside text of a <w> (from Ts-227a,105][4]et106[1]_7)

```
1  if element.get("l") == 'UNKNOWN':
2      if "type=\"notation\"" not in element.text:
3          word = element.text
4
5          # if inside the word there is an underscore followed by a lowercase
       letter, it should be deleted
6          word = re.sub(r"(.*)-([a-zaäoöuü].*)", r"\1\2", word)
7          # delete <lb> tag
8          word = re.sub(r"<lb .+?>", r"", word)
9          # delete <sp/>
10         word = re.sub(r"<sp/>", "", word)
11         # delete
12         word = re.sub(r"<pb.*?/>", "", word)
13         # delete <seg> tags
14         word = re.sub(r"<seg .+?>", r"", word)
15         word = re.sub(r"</seg>", "", word)
```

Listing 4.6: Extracting full form of UNKNOWN word in language_finde.py

The preprocessing step of the TreeTagger sometimes leaves XML elements in the text content of the element `<w>`, see listing for an example. In order to see if a word is marked as `UNKNOWN` because it doesn't exist in the lexicon, we need to delete all XML elements that are found inside the text of `<w>`. This is done from line 7 until 15 with help of regular expressions and the function `re.sub(pattern, replacement, string)`.

```
<persName key="Cantor, Georg">
  <w t="ADJA" l="UNKNOWN">Cantor<lb rend="shyphen"/>sche</w>
<persName>
```

Listing 4.7: Text of <w> has XML element (from Ms-121,71r[2]et71v[1]_1)

After this steps, the word is inserted as a key into the dictionary `unknown_words_dict` and its file name is added to the list of its values. When the program finishes iterating through the OA_NORM-tagged.xml files, the `unknown_words_dict` is sorted alphabetical and saved into unknown_words.txt file, which can be found in the attached SD Card. The unknown words in this file should be compared to the entries in the lexicon used for the FinderApp WiTTFind and all words that don't already form part of it should be added [1].

[1] This follow up step is outside the scope of this thesis.

4.3. Recognizing the language of a file

Until recently, only 5.000 pages out of the 20.0000 pages of Ludwig Wittgenstein's *Nachlass* were open to the public. The other 15.000 pages were considered secure. The division of the OA_NORM.xml documents into two different lists comes from that time.

The array named `dirs`, see listing 4.8, contains the name of the files that were open source from the beginning, while the `sec_dirs` array, see listing 4.9, is filled with the names of the files that used to be secure.

```
dirs = ["Ms-114_OA","Ms-139a_OA", "Ms-141_OA","Ms-149_OA","Ms-152_OA","Ms
    -153b_OA","Ms-155_OA",
    "Ts-201a1_OA","Ts-207_OA","Ts-213_OA","Ms-115_OA","Ms-140,39v_OA", "Ms
    -148_OA","Ms-150_OA",
    "Ms-153a_OA","Ms-154_OA", "Ms-156a_OA","Ts-201a2_OA","Ts-212_OA","Ts-310
    _OA"]
```

Listing 4.8: Array dirs in language_finder.py

```
sec_dirs = ["Ms-101_OA","Ms-102_OA","Ms-103_OA","Ms-104_OA","Ms-105_OA",
    "Ms-106_OA","Ms-107_OA","Ms-108_OA","Ms-109_OA","Ms-110_OA",
    "Ms-111_OA","Ms-112_OA","Ms-113_OA","Ms-116_OA","Ms-117_OA","Ms-118_OA",
    "Ms-119_OA","Ms-120_OA","Ms-121_OA","Ms-122_OA","Ms-123_OA","Ms-124_OA",
    "Ms-125_OA",
    "Ms-126_OA","Ms-127_OA","Ms-128_OA","Ms-129_OA","Ms-130_OA","Ms-131_OA",
    "Ms-132_OA",
    "Ms-133_OA","Ms-134_OA","Ms-135_OA","Ms-136_OA","Ms-137_OA","Ms-138_OA",
    "Ms-139b_OA",
    "Ms-140_OA","Ms-142_OA","Ms-143_OA","Ms-144_OA","Ms-145_OA","Ms-146_OA",
    "Ms-147_OA",
    "Ms-151_OA","Ms-156b_OA","Ms-157a_OA","Ms-157b_OA","Ms-158_OA","Ms-159
    _OA","Ms-160_OA",
    "Ms-161_OA","Ms-162a_OA","Ms-162b_OA","Ms-163_OA","Ms-164_OA","Ms-165_OA
    ","Ms-166_OA",
    "Ms-167_OA","Ms-168_OA","Ms-169_OA","Ms-170_OA","Ms-171_OA","Ms-172_OA",
    "Ms-173_OA",
    "Ms-174_OA","Ms-175_OA","Ms-176_OA","Ms-177_OA","Ms-178a_OA","Ms-178b_OA
    ","Ms-178c_OA",
    "Ms-178d_OA","Ms-178e_OA","Ms-178f_OA","Ms-178g_OA","Ms-178h_OA","Ms-179
    _OA","Ms-180a_OA",
    "Ms-180b_OA","Ms-181_OA","Ms-182_OA","Ms-183_OA","Ms-301_OA","Ts-202_OA"
    ,"Ts-203_OA",
    "Ts-204_OA","Ts-205_OA","Ts-206_OA","Ts-208_OA","Ts-209_OA","Ts-210_OA",
    "Ts-211_OA",
    "Ts-214a1_OA","Ts-214a2_OA","Ts-214b1_OA","Ts-214b2_OA","Ts-214c1_OA","
    Ts-214c2_OA",
    "Ts-215a_OA","Ts-215b_OA","Ts-215c_OA","Ts-216_OA","Ts-217_OA","Ts-218
    _OA","Ts-219_OA",
    "Ts-220_OA","Ts-221a_OA","Ts-221b_OA","Ts-222_OA","Ts-223_OA","Ts-224_OA
    ","Ts-225_OA",
    "Ts-226_OA","Ts-227a_OA","Ts-227b_OA","Ts-228_OA","Ts-229_OA","Ts-230
    a_OA","Ts-230b_OA",
    "Ts-230c_OA","Ts-231_OA","Ts-232_OA","Ts-233a_OA","Ts-233b_OA","Ts-235
    _OA","Ts-236_OA",
```

```
    "Ts-237_OA","Ts-238_OA","Ts-239_OA","Ts-240_OA","Ts-241a_OA","Ts-241b_OA
    ","Ts-242a_OA",
    "Ts-242b_OA","Ts-243_OA","Ts-244_OA","Ts-245_OA","Ts-246_OA","Ts-247_OA"
    ,"Ts-248_OA",
    "Ts-302_OA","Ts-303_OA","Ts-304_OA","Ts-305_OA","Ts-306_OA","Ts-309_OA"]
```

Listing 4.9: Array sec_dirs in language_finder.py

The distinction of the files into two groups is still important since some of the data shown in the FinderApp WiTTFind was obtained from the files in the `dirs` array. All the tools and programs used by WiTTFind are slowly being converted to take all the files as input, but at least until then, the separation into the `dirs` and `sec_dirs` array is useful.

The `language_finder.py` program contains the aforementioned arrays and it receives the path and names of the OA_NORM-tagged.xml files which it then saves into a list. The program iterates through the files and decides in which language they are written. Four new arrays are derived from the `dirs` and `sec_dirs` to categorize the files and their language: `dirs_de`, `dirs_en`, `sec_dirs_de` and `sec_dirs_en`. The appended "de" stands for the German language while "en" stands for the English one.

To obtain the language of the files and categorized them into the new arrays, the program parses the XML document with help of `iterparse` from `lxml.etree` into a tree and generates (event, element) tuples. To recognize the language of a file, we are only interested in the elements which tag is "w" (stands for word) and which have "FM" as value for the attribute "t" (tag). To get the tag of an element iterparse provides the function `.tag` and to get the value of an attribute the function `.get(attribute_name)` can be used, see listing 4.10.

```
1 if element.tag == 'w':
2   if element.get('t') == 'FM':
3     not_tagged_w += 1
4   else:
5     tagged_w +=1
```

Listing 4.10: if statement inside of iterparse for a file

To decide the language of a file, the variables `tagged_w` and `not_tagged_w` are initialized with 0. Since the program is iterating through the elements of the document, every time it finds a word which attribute "t" is "FM" (representing a non German word), the counter for `not_tagged_w` increases. If a word attribute has any other "t" value, the counter for `tagged_w` increases. At the end of the document the program checks which of these two variables is bigger.

If the variable `tagged_w` is bigger, the program classifies the file as German and inserts the file name either in the `dirs_de` array or the `sec_dirs_de` depending on which array the file was originally from. On the other hand, if there are more untagged words, i.e. the counter `not_tagged_w` has a bigger number than `tagged_w`, the file is considered to be written in English and it is sorted into the `dirs_en` or `sec_dirs_en` array depending on its origin.

The program found out that seven files of the *Nachlass* are written in English. Their pages belong to the first 5.000 open source pages, see 4.11.

```
1 dirs_en= ['Ms-139a_OA', 'Ms-148_OA', 'Ms-149_OA', 'Ts-201a1_OA', 'Ts-201
    a2_OA', 'Ts-207_OA', 'Ts-310_OA']
```

Listing 4.11: dirs_en

As mentioned at the beginning of this chapter, the TreeTagger should be calibrated to do the POS-tagging of the files depending on their language. Before the actual tagging, there are preprocessing steps that should also be adapted for the English language since they are written for German. This implementations would have been too much for the time scope of this work but should be done in the future in order to obtain correct parts of speech tags for the English words and also to avoid the tagging of English tokens as German ones.

5. Frequency Lists

Users of WiTTFind can find frequency lists grouped by semantic categories in the website. Most of these lists were done with the first 5.000 pages of Wittgenstein's *Nachlass* that were open to the public. Now that all the 20.000 pages of the manuscripts and type scripts are accessible, the semantic frequency lists need to be updated.

Ludwig Wittgenstein wrote about a varying number of topics. One of them is music. Colors also played a big role in the work of this philosopher and in his type script Ts-213 he wrote under the chapter "Phänomenologue" (phenomenology) a subchapter on color and color mixing. It is not surprising that the second most common adjective in this document is "rot" (red). Many semantic categories could be analyzed in Wittgenstein's *Nachlass*, because of the time constraint of the thesis, it was decided that only the categories of color and music would be explored.

In a previous thesis, frequency lists for these and other semantic categories were created. These frequency lists were static and covered only the first 5.000 pages of the *Nachlass* open at that time. In order for the users to further analyze Wittgenstein's work, the frequency lists need to cover all of the *Nachlass*. This chapter explains the process of creating new semantic frequency lists for the FinderApp WiTTFind. The process is implemented in such way, that it can be added to the [CW]AST Toolchain. The semantic frequency lists are therefore dynamic and can be called with a makefile target as all the other tools in the chain.

5.1. Lexicon

The lexicon used by the FinderApp WiTTFind is called witt_WAB_DELA. It intends to include all the words that appear in Ludwig Wittgenstein's *Nachlass* and it's sorted alphabetically. This lexicon comprises grammatical and semantic characteristics for the tokens that can be use to extract the semantic frequency lists and other important informations for the FinderApp. It is an electronic lexicon held in the German DELAF format. This format is specially suitable for working with local grammars and for processing text corpus with the help of Unitex. For a better explanation of the Lexicon, see the thesis done by Angela Krey [7].

Since the lexicon has been improved over time, there are different versions of it. As in fall 2018, the lexicon being used is the witt_WAB_dele_XIX.txt.

Each line in the lexicon represents a word and it's composed following the schema:

`fullform,lemma.grammatic_categories+semantic_categories...`

As mentioned above, the frequency lists created in the scope of this thesis comprise the semantic categories for music and color. The semantic category `MUSIK` (music) marks in the lexicon all words that fall into this category. Some examples for these kind of entries can be found in 5.1.

```
Akkord,.N+MUSIK
Antoni,.EN+persName+MUSIK
Bachanten,Bacchant.N+MUSIK
Bach Johann Sebastian,Bach.EN+persName+MUSIK+KOMPONIST
Bach,.N+persName+MUSIK+KOMPONIST:aeM:deM:neM
Blasinstrumente,Blasinstrument.N+MUSIK:amN:deN:gmN:nmN
...
```

Listing 5.1: words in witt_WAB_dele_XIX.txt with sematic category MUSIK

As mentioned a the beginning of this chapter, Wittgenstein worked intensively with color theory. There are many semantic categories for color such as Zwischenfarbe (intermidiate color), Transparenz (transparency), Glanz (shine) and so on, but they all are a subset of the category `COL`, standing for color. In listing 5.2 there are some example for words that fall into the color semantic category.

```
dunkel,.ADJ+COL+Zwischenfarbe:up
dunkelblau,.ADJ+COL+Zwischenfarbe
Dunkelrot,.N+COL
durchsichtige,durchsichtig.ADJ+COL+Transparenz
einfarbige,einfarbig.ADJ+NUM+COL+Farbigkeit
farbloses,farblos.ADJ+REL+COL+Farbigkeit
...
```

Listing 5.2: words in witt_WAB_dele_XIX.txt with sematic category COL

Understanding the structure of the entries in the lexicon is important because it will make the extraction of words for the semantic frequency lists easier.

5.2. Make deploy chain for the semantic frequency lists

The tools needed to create the semantic frequency lists are controlled by the makefile `semantic_freqlist.make` and the logic follows the structure of the [CW]AST deploy chain. The makefile can be found under the `witt-data/deployment/makefile` folder, where all other makefiles needed to deploy the FinderApp are stored. All the makefile targets in `semantic_freqlist.make` have to be called from inside the `witt-data/deployment` folder.

In `semantic_freqlist.make` the path to the different programs needed to produce the frequency lists as well as the path destinations for the output are saved into variables. These variables are later used in the command part of the make rules.

To generate the semantic frequency lists for music and color, a frequency list over all words of the *Nachlass* is needed first. This frequency list is created with the script `all_frequencies.py` that can be found in the `witt-data/tools/frequency` folder and

can be called with the makefile target `make all-freqlist`, see listing 5.3 line 5. This rule has as dependency the OA_NORM-tagged.xml files, which means that if they change, the rule can be called again to redo the frequency list.

```
1 TAGGED_UNEXPANDED_NORM_FILES   = $(shell find -L $(NACHLASS_DIR)/*/norm -
      type f -name \*.xml | grep '\-tagged' | grep -v '\expanded')
2
3 semantic_freqlist: all-freqlist music-freqlist color-freqlist
4
5 all-freqlist: $(TAGGED_UNEXPANDED_NORM_FILES)
6 	$(SILENT) $(PYTHON3_RUNNER) $(FREQ_ALL_DIR)/$(FREQ_ALL_CMD) $(ALL_FREQLIST
      ) $(ALL_FREQ_PICKLE) $^
7 	$(SILENT) echo "written $(ALL_FREQLIST)"
8 	$(SILENT) echo "written $(ALL_FREQ_PICKLE)"
9
10 music-freqlist:
11 	$(SILENT) $(PYTHON3_RUNNER) $(FREQ_MUSIC_DIR)/$(FREQ_MUSIC_CMD) $(
      DICT_WITT) $(ALL_FREQ_PICKLE) $(MUSIC_FREQLIST)
12 	$(SILENT) echo "written $(MUSIC_FREQLIST)"
13
14 color-freqlist:
15 	$(SILENT) $(PYTHON3_RUNNER) $(FREQ_COLOR_DIR)/$(FREQ_COLOR_CMD) $(
      DICT_WITT) $(ALL_FREQ_PICKLE) $(COLOR_FREQLIST)
16 	$(SILENT) echo "written $(COLOR_FREQLIST)"
```

Listing 5.3: section of semantic_freqlist.make[1]

The program to create the lemmatized frequency list for music, `music_freqlist.py`, can be found in `witt-data/tools/frequency/music` folder and this program can be called with the makefile target `make music_freqlist`.

For the colors, the program is called `color_freqlist.py` and it's located in the directory `witt-data/tools/frequency/color`. The makefile target `make color-freqlist` calls this script.

The three makefile targest can be called at once with the target `make semantic_freqlist`, see line 5 in listing 5.3.

5.3. Frequency of words over all files

The program `all_frequencies.py` can be called with the following command:

```
$ python all_freqlist.py arg1 arg2 arg3...
```

The first argument expected is the output file for the frequency list in txt format. The secon one is the output file for the frequency list in pickle format and arg3 until argn represent all the OA_NORM-tagged.xml files. The program saves the tagged files into an array to later iterate through them.

[1] For readability purposes, line 1 was added to semantic_freqlist.make. This line is declared in the macro Makefile. For original code of semantic_freqlist.make, see attached SD Card.

This script works in a similar way to the `language_finder.py` explained in chapter 4. It initializes a dictionary `all_word_freqs` which keys will be the tokens and their values the amount of times that world appears throughout the *Nachlass*. It then reads one by one the tagged files and parses each document with the help of `iterparse` from `lxml.etree` into a tree, creating a tuple of the form (event,element). Only the elements with the tag "w" (words) are important to create the frequency lists. Again, the program ignores the the mathematical formulas found in the *Nachlass*, since they are not considered tokens.

The word is then cleaned from possible XML elements due to different types of input and preprocessing errors. This is done exactly as in 4.2 with the help of the regular expressions (regex) in listing 4.6. After these steps, there are still strings representing a token that start with a punctuation symbol. These should not be inserted into the frequency list and therefore an additional step is required.

To do so, a string `all_punctuation`, see listing 5.4, is declared with the help of the Pyhton method `string.punctuation` (gives the ASCII characters which are considered punctuation back). Other punctuation characters found in the *Nachlass* need to be added to this string. The processed word is only added to the dictionary if it doesn't start with a punctuation symbol.//

```
all_punctuation = string.punctuation + "”»‹–’‘„..."

if word[0] not in all_punctuation:
  all_words_freqs[word] += 1
```

Listing 5.4: Check that word has no punctuation in first character

After finishing iterating through all the files, the program sorts the frequency list of the words by descending order of their value and saves the sorted dictionary in a pickle file. It also saves a txt version in which each line represents a word followed by a space followed by its frequency, this file can be found in the attached SD Card.

5.4. Semantic frequency lists

The same program is used to create the semantic frequency list for music and for color. They search for a different semantic category in the witt_WAB_DELA lexicon. While `color_freqlist.py` searches for the entries in the dictionary that have the `COL` semantic category, `music_freqlist.py` searches for the entries with the semantic category `MUSIK`.

What follows is a short explanation of how the program creates a lemmatized frequency list for the semantic category color. It is a lemmatized frequency list because at the end the frequencies should be sorted by their lemma and an entry in the list should look like this:

```
lemma,sum_of_all_freqs; first_fullform, freq; second_fullform, freq; ...
```

A concrete example of the output of `color_freqlist.py` can be seen in listing 5.5. For the complete list see attached SD Card.

```
dunkelblau,6; dunkelblau,3; dunkelblauen,3;
Dunkelrot,4; Dunkelrot,4;
einfarbig,6; einfarbig,2; einfarbige,2; einfarbigen,2;
farblos,33; farblos,14; farblose,7; farbloser,5; farbloses,5; farblosen,2;
...
```

Listing 5.5: Some entries in lemmatized_color_frequencies.txt

The script `color_frqlist.py` can be called as:

```
$ python color_freqlist.py arg1 arg2 arg3
```

The first argument expected is the lexicon `witt_WAB_dela_XIX.txt`, the second argument should be the frequency over all words in the lexicon in pickle format and the third argument is the file where the output should be saved.

The dictionary of dictionaries `color_freqs` is initialized. Its keys are the lemma of different full forms and its values are dictionaries with all full forms mapped to their frequencies.

The program reads one by one the lines in the lexicon, see listing 5.6, and checks with help of `re.match` at the beginning of the string for anything (fullform) followed by a comma, then anything (lemma) follwed by a period follwed by anything and then +COL. As we mentioned above, COL symbolizes the semantic category for colors. The pattern match has to follow the DELAF format explained at the beginning of this chapter.

```
color_freqs = defaultdict(lambda: defaultdict(int))

with open(lexicon, 'r') as witt_lex:
    for entry in witt_lex:
        col = re.match("(.*),(.*)\..*\+COL", entry)
        if col:
            # lstrip is used because some words in the dictionary have
    leading spaces
            full_form = col.group(1).lstrip()
            lemma = col.group(2).lstrip()
            if not lemma:
                lemma = full_form
            if full_form in all_frequencies:
                color_freqs[lemma][full_form] = all_frequencies[full_form]
```

Listing 5.6: Section of code from color_freqlist.py

If the entry in the lexicon matches the pattern, the full form for the word is set to the first group of the match. Sometimes a full form of a word is also its lemma. In this case, the lemma is left empty in the lexicon entry. See entry for "Dunkelrot" (dark red) in listing 5.2. The program deals with these kinds of entries, see lines 10 to 11, by checking if the second group captured something. If it didn't, it sets the lemma of the word to be the same as its full form.

With the variables `full_form` and `lemma` filled, the program checks if `full_form` is a key in the dictionary created by `all_freqlist.py`, see subchapter 5.3. If the full form of the entry in the lexicon is in the frequency list over all words in the *Nachlass*, then it is added to the lemmatized dictionary together with its frequency, see line 13 of 5.6.

When `color_freqlist.py` finishes iterating through the lexicon, it still needs to write the obtained frequency list into a txt file. To do so, it iterates through the items in the dictionary `color_freqs` and sums all the values for the different full forms of a lemma entry with the help of the function `sum`, see 5.7. The lemma followed by this sum, and then the full forms with their frequencies are written in the output file that looks, as mentioned before, like listing 5.5.

```
1     for lemma, full_forms in color_freqs.items():
2         sum_of_freqs = sum(full_forms.values())
```

Listing 5.7: Section of code from color_freqlist.py

The program to extract the semantic frequency list for music uses a different regex for the matching, see bellow. This is the only line that is different between the two scripts `color_freqlist.py` and `music_freqlist.py`.

```
music = re.match("(.*),(.*)\..*\+MUSIK", entry)
```

Listing 5.8: Regex to match semantic category music

By replacing the regex on line 5 of listing 5.6 one can find different semantic categories and do a frequency list for them if needed.

5.4.1. Old frequencies

As mentioned at the beginning of the chapter, frequency lists for different semantic categories were created as part of a previous bachelor thesis. The resulting frequency lists for the first 5.000 open source pages can be found in the FinderApp WiTTFind.

To compare the old frequencies with the ones created in this work, the 10 most common words of the old frequencies for color adjectives, see table 5.1, were extracted.

The composers frequency list shown in table 5.2 and later on in 5.6, shows the 10 most frequent mentioned composers by lemma (not by full form as in the color adjectives).

The pages of the type script Ts-213 form part of the first 5.000 pages of Wittgenstein's Nachass that were open to the public. As previously mentioned, the second most common adjective in this type script is "rot" (red). This type script is one of the largest documents in the *Nachlass* and it is not surprising to find this color adjective in the first place of the color list with a frequency of 904. Different representation forms for this adjective, depending on whether the noun it is modifying is singular or plural and its gender, make it also to the list of the most common color adjectives used by Wittgenstein.

Music was another topic the philosopher wrote about and therefore it is important to research this semantic category in his work. To make the comparison of the old frequencies with the new ones, it was decided that the semantic subcategory KOMPONIST (composer) should be explored. The composer that appears the most throughout the first 5.000 open pages of the *Nachlass* is Beethoven. He is followed by Schubert.

Wort	Frequency
rot	904
klar	267
blau	209
gelb	152
roten	141
rote	104
gelbe	92
schwarz	75
blue	73
rotes	68

Table 5.1.: Old frequencies for color adj retrieved from http://wittfind.cis.uni-muenchen.de/?semantics#

Wort	Frequency
Beethoven	41
Schubert	31
Brahms	26
Mozart	23
Mendelssohn	16
Bruckner	15
Labor	10
Wagner	9
Schumann	8
Haydn	7

Table 5.2.: Old frequencies for composers retrieved from http://wittfind.cis.uni-muenchen.de/?semantics#

5.4.2. Frequencies of additional 15.000 pages

The frequencies for the then secure pages can be found in table 5.3 for color adjectives and in table 5.4 for the composers. A few interesting things can be observed.

The color adjective "rot" was found 1256 times in the 15.000 left pages. The adjective "klar" was found even more often than "rot", occurring 1415 times. The word "klar" can mean different things depending on the context, for example clear or transparent.

The use of color adjectives continues to be strong for the rest 3/4 of Wittgenstein's *Nachlass*.

The words retrieved regarding music composers in the additional 15.000 pages are very scarce to say the least. In these pages, Wittgenstein mentions Bruckner 3 times. No other composer is mentioned. This shows, that the philosopher talks mostly about mu-

Wort	Frequency
rot	1256
klar	1415
blau	499
gelb	290
roten	282
rote	190
gelbe	85
schwarz	166
blue	27
rotes	53

Table 5.3.: Additional frequencies for color adj

sic in one or more of the documents belonging to the first 5.000 open pages of the *Nachlass*.

Wort	Frequency
Bruckner	3

Table 5.4.: Additional frequencies for composers

5.5. Difference between old frequencies and new frequencies

The frequencies shown below are the new total frequencies for the 10 most frequent words appearing in the old frequencies table 5.1 and 5.2.

Wort	Frequency
rot	2160
klar	1682
blau	499
gelb	247
roten	423
rote	294
gelbe	177
schwarz	241
blue	100
rotes	121

Table 5.5.: New frequencies for color adj

The frequencies for composers decreased from the old frequencies, see table 5.2, to the

frequencies for composers over all 20.000 pages, see table 5.6. This would be really odd if it weren't for the fact that the XML documents received from Bergen have changes in their format from time to time. The changes implemented are to fix some errors but also new transcription or edition problems can be found in them. To understand why the frequency for Beethoven, Schubert and Brahms decreased by 1, for Haydn by 2 and for Mendelssohn by 3, a deep research would need to be make but this exceeds the scope of this work.

Wort	**Frequency**
Beethoven	40
Schubert	30
Brahms	25
Mozart	23
Mendelssohn	13
Bruckner	18
Labor	Not marked as MUSIC
Wagner	9
Schumann	8
Haydn	5

Table 5.6.: New frequencies for composers

No entry for Labor was found either in the search bar of WiTTFind or in the newly created frequency list. Joseph Labor was a composer and an entry for his name can be found in the witt_WAB_delaXIX.txt lexicon:

```
Labor Josef,Labor.EN+MUSIK+KOMPONIST
```

The reason why this composer doesn't appear either in the new semantic frequencies or in the search bar in WiTTFind is because both programs check that the full form of a an entry in the lexicon is a key in the frequency over all words dictionary. The full form given by the Lexicon is "Josef Labor".

The entries found for Labor in the dictionary `all_words_freqs` created by the script `all_freqlist.py` are:

```
all_words_freqs ={
  ...
  Labor: 8,
  Labors: 2,
  ...
}
```

"Josef Labor" is not found as key of any item in the dictionary, since Wittgenstein never writes his complete name. The error lies on the incomplete Lexicon witt_WAB_delaXIX.txt but can easily be fixed by adding the two following entries:

```
Labor,.EN+MUSIK+KOMPONIST

Labors,Labor.EN+MUSIK+KOMPONIST
```

This type of error is better explained in chapter 6.

5.6. New frequencies

Until now we compared the old frequencies with the new ones. In this last part of the chapter, the new frequencies over all documents are shown. The 20 most common words for the semantic category color and for the semantic category music are shown in table 5.7 and table 5.8 respectively.

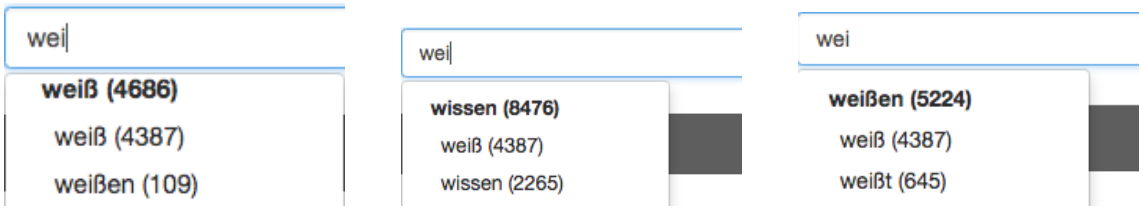
Wort	**Frequency**
weiß	4387
rot	2160
klar	1682
Rot	865
blau	499
roten	423
grün	411
Weiß	375
Grün	298
rote	294
gelb	247
Blau	243
schwarz	241
Gelb	236
rein	221
roter	187
gelbe	177
heller	177
schwarzen	170
Schwarz	169

Table 5.7.: New frequencies over all color semantic category

The word that ranks first in the new semantical frequency list for color is "weiß", which means white. This word is also is the present form of the verb wissen (to know) for the first and third person singular.

The figures 5.1a, 5.1b and 5.1c show that the full form word "weiß" has the same frequency for its different possible lemmas. This problem is created when the semantic categories of

a word are created with only the help of a lexicon. To disambiguate the meaning of a word, its POS-tag should be taken into account when creating the list of frequency over all words.



(a) weiß (b) wissen (c) weißen

Figure 5.1.: weiß with different lemmas

The following examples aim to show two different uses of the the word "weiß". In the first example, "D.h. also, er weiß immer mehr, als er zeigen kann." (That means, he always knows more than he can show.) found in Ts-213,12r[3]_3 the word is used as a verb. The tagging for it can be found in listing 5.9.

```
<w ana="pagenr:29 linenr:11 tokennr:6" l="wissen" t="VVFIN">weiß</w>
```

Listing 5.9: weiß as verb (from Ts-213,12r[3]_3)

In the sentence "im Schachspiel wird die weiße Farbe von Fi- guren zur Unterscheidung von der schwarzen Farbe andrer Figuren gebraucht." (In chess, the white color of figures is used to distinguish them from the black color of other figures.) from Ts-213,441r[6]_2 the word "weiße" is an adjective. Listing 5.10 shows how this word is tagged.

```
<w ana="pagenr:613 linenr:1 tokennr:28" l="weiß" t="ADJA">weiße</w>
```

Listing 5.10: weiß as adjective (from Ts-213,441r[6]_2)

The disambiguation of the meaning of words has to be done with help of their tag. A lexical frequency list does not suffice to disambiguate the meaning of some full forms. This approach for creating frequency list could be a research possibility for a future thesis since this kind of problem is not specific to the word shown in the example.

Red is one of the most common color adjectives by Wittgenstein. It ranks first in the old frequency list and second in the new one. Other declination's of this adjective make it again to the list.

The 20 most common words in the semantic category of music are topped by the word "Form". In music a form refers to the structure of performance or composition. It is clear though, that this word can also be used in many other non musical context and therefore it is not strange that the frequency of this word is by far higher, than all other words frequencies that fall into this semantic category. All other words found in the list are less ambiguous.

The complete lemmatized frequency list for music and color can be found in the attached SD Card.

Wort	**Frequency**
Form	3085
spielen	839
Ton	446
hören	326
Musik	200
Melodie	184
Thema	155
Klang	142
Töne	132
play	92
singen	75
Noten	73
Rhythmus	67
Klavier	52
playing	46
klingen	45
tone	44
Phrase	43
hear	39
Note	36
Musikstück	35

Table 5.8.: New frequencies over all music semantic category

6. Evaluation

This chapter will describe the way in which the frequency list over all words in the *Nachlass* and the semantic frequency lists for color and music were evaluated.

6.1. Evaluation of the new frequencies

To measure if the newly created frequency list over all words in the *Nachlass* is accurate, the new frequencies have to be compared to the frequencies found in the search bar in the WiTTFind website. The new frequency list is composed of 55039 lines, each one representing a token and its frequency. Some of the entries are composed of only digits or symbols. Others have a combination of letters and other characters. Since we are only interested in finding the frequency of words, the lines containing this sort of entries were erased for the evaluation, leaving 48813 entries.

Taking into consideration that it is not possible to prove one by one all the 48813 entries by hand, it was decided to evaluate each 200th word of the frequency list. To do so, the words to be evaluated were copied into a new file `eval_words.txt` with the help of unix commandos. The resulting file has 244 lines, each one containing a word followed by its frequency. This lines are the ones to be evaluated.

Although most of the mixed strings that didn't represent a word were removed in the first step, some of the lines in `eval_words.txt` don't represent a word either. Some of these entries have punctuation symbols, digit or other kind of symbols in them. Others were wrongly preprocessed probably because of an XML `<lb>` element. The listing 6.1 is a subset of some of this type of entries.

```
H–O–O–O–H
a112
anzes
er↑
ezug
g(y)
notatio310−118
nwesentlich
```

Listing 6.1: Some non words elements in eval_words.txt

The comparison between the frequencies of the words in `eval_words.txt` and their frequencies in the search bar of WiTTFind shows that most of the words, 183 out of 244, have the same frequency. This equates to an accuracy of 75%.

For other 12 entries in `eval_words.txt`, WiTTFind suggested a similar word, see table 6.1. In some of the cases, e.g. "Gemeinsamen" (common or mutual), the word exists

but it's not recognized. For other entries, WiTTFind suggest adding a missing letter at the beginning of what could be a word, e.g. "uadrat" could possibly be missing a "Q" to compose the word "Quadrat" (square).

The words "Gemeinsamen" and "Dahin" (then or there) that can be seen in the table 6.1, are an adjective and an adverb respectively. They are normally written in lowercase and their entries in the lexicon are in lowercase as well. Therefore it is not surprising that WiTTFind doesn't find them when their first letter is uppercase.

Entry in eval_words.txt	**WiTTFind suggestion**
Brechen	Brechens
Dahin	dahin
Französischen	Französische
Gemeinsamen	Gemeinsame
chrittweise	schrittweise
raduellen	graduellen
rojektion	Projektion
sychologische	psychologische
uadrat	Quadrat
urchlaufen	durchlaufen
xterne	externe
ührt	führt

Table 6.1.: Suggested words in WiTTFind

For the sample word "Klangbilder", the new frequency list found one more occurrence than WiTTFind. The reason for this is that the program `all_freqlist.py` described in 5.3 takes away a dash in the middle of a word if the letter that follows it is in lowercase. This "cleans" some tokens and because of this, they are now recognized as entries of the lexicon.

The word "Vorstellende" (performer) has an occurrence of 4 in the new frequency list but of 7 in WiTTFind. When searching for the word in the manuscripts and typescript in WiTTFind, one can see that some of the entries are not for "Vorstellende" but for its plural "Vorstellenden". The plural form of this word doesn't show up in the WiTTFind search bar and this is another type of error that will be discussed later in this chapter.

If the entries that don't form a proper word in either English or German are not considered in the evaluation, then 183 out of 219 words are correct. This represents an accuracy of 86%. As mentioned before, some of the words were either not found in WiTTFind or their frequency was higher than expected. This is due to irregularities in the lexicon that will be discussed later on.

6.2. Errors in WiTTFind

All the entries in the lemmatized frequency lists for the semantic categories color and music were manually evaluated by comparing each word's frequency with it's frequency in WiTTFind's search bar.

6.2.1. Lexicon

One of the main sources of errors for the semantic frequency lists is the lexicon `witt_WAB_dela`. The lexicon has 62918 lines and it should be comprised by all the words and their lemmas found throughout Ludwig Witgenstein's *Nachlass*. Mantaining the lexicon is an incredibly time consuming and hard task.

Semantic category missing in the lexicon

The evaluation of the words for the semantical category of color and music showed a discrepancy in some words that were grouped under a lemma. While in WiTTFind the frequency for the lemma braun (brown) is 105, see 6.1, in the lemmatized frequency list obtained from `color_freqlist.py` is 103:

```
braun,103; braun,60; braunen,21; braune,20; braunes,2;
```

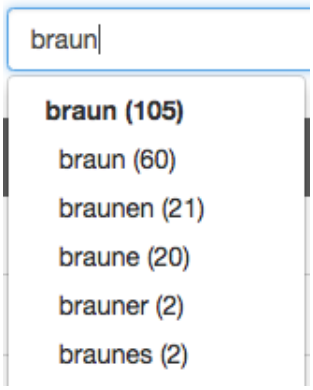


Figure 6.1.: Lemmatized frequency for braun in WiTTFind

In the frequency obtained by `color_freqlist.py` the full form "brauner" is missing. As shown in the subchapter 5.4, the lemmatized frequency list is created with help of a regex that searches in the `witt_WAB_dela` lexicon for entries that contain the sematic category `COL` (color). The entry for "brauner" in the lexicon looks like this:

```
brauner,braun.ADJ
```

It is missing the semantic category for color and because of this, it is not being grouped as part of the lemma "braun" in the output of `color_freqlist.py`. All the other forms of this color have the semantic category `COL` in the lexicon. This category should be added

to its entry to extend the morphological category. The corrected entry will look as follows:

```
brauner,braun.ADJ+COL
```

This kind of error in the lexicon occurs a few times for both semantic categories color and music. In the attached SD Card there are two list with the worlds that don't have either one of this categories in their lexicon entry but should. This list should help extend their categories in the lexicon, improving the semantic frequency lists.

Typos in the lexicon

To extract information from the lexicon most of the tools in WiTTFind use regular expressions. For this reason the lexicon's entries need to follow the structure shown in the subchapter 5.1. If they don't, errors start happening. An example for this is the following entry in the lexicon:

```
blasseren, blaß.ADJ+COL
```

A space between the comma and the lemma got inserted accidentally and this prevented a correct generation of the semantic frequencies. To deal with this type of error, the programs `color_freqlist.py` and `music_freqlist.py` use the python function `strip`, which deletes leading spaces in the words.

6.2.2. Preprocessing

As seen in chapters 4 and 5, the preproccesing step of the TreeTagger sometimes delivers words with either XML elements on them or that are wrongly separated. This causes the TreeTagger to not work properly and some words get tagged as `UNKNOWN`. For the semantic frequency lists the words are processed again as shown in the previously mentioned chapters to leave a "clean" token.

One of the steps that differs between `makeFrequencyList.perl` called by the [CW]WAST Toolchain and the script `all_freqlist.py` is that the latter replaces dashes that are found in the middle of a string if the letter after it is lowercase. This was done to recreate words that were wrongly separated and that still have the separation symbol inside of them.

This creates a difference in the frequency for some of the words. In WiTTFind for example the word "blaurot" (blue-red) is found 22 times but in the new frequency list it is found 23 times. In table 6.2 other frequency differences can be found for the words belonging to the semantic category of color.
The above mentioned step should be added for other tools in the [CW]AST Toolchain to ensure that more words are found in the future.

word	WiTTFind frequency	New frequency
rötlichgelb	3	6
rötlichgrün	6	7
schwarzweiß	1	4

Table 6.2.: Improved frequencies for words

6.3. Morphological extension

Some of the words and their frequency found by the programs `color_freqlist.py` and `music_freqlist.py` are not found under their full form but rather under their lemma in WiTTFind. They appear in their full form in the *Nachlass* though .

An example of this is when the word "andersfärbigem" (differently colored) is searched, WiTTFind doesn't find it but suggest its lemma, see figure 6.2. The lemmatized frequency obtained from `color_freqlist.py` is:

```
andersfärbig,2; andersfärbigem,2;
```

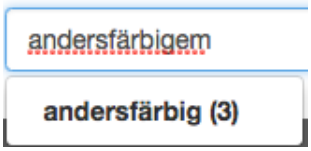


Figure 6.2.: Wronlgy suggested word

When clicking on "andersfärbig" in the search bar, WiTTFind shows different sentences in the *Nachlass* in which this word appears. Not the lemma, but one of its fullform "andersfärbigem" was used by Wittgenstein in the sentence found in Ms-106:

> Wenn ich sage das kleine Quadrat im 106008 großen ist rot was immer das übrige für eine Farbe haben mag so kann ich mir doch das kleine Quadrat gar nicht vorstellen wenn es nicht von etwas **andersfärbigem** begrenzt ist.

This shows that some of the words are not being lemmatized correctly for the output of the search bar.

The reason why the frequency for the lemma "andersfärbig" is 3 in WiTTFind but 2 in the semantic list obtained here, is that the entry for "andersfärbigen" in the lexicon doesn't have the semantic category `COL`. This is the same problem as mentioned in the section above and can be easily be fixed by adding the category to the word.

Another examples for when a word is only found by its lemma and not by its full form, creating a wrongly lemmatized frequency, was mentioned in the subchapter 6.1. The word "Vorstellenden" is not found in WiTTFind, but its lemma "Vorstellende" is. The plural form of this word can be seen in some of the manuscripts and type scripts.

In order for the words to be found by their full form in the FinderApp WiTTFind, the lexicon has to have an entry for the lemma and an entry for the full form. This error appears when there is no entry for the lemma. In `witt_WAB_dela_` the following two entries can be found:

```
andersfärbigem,andersfärbig.ADJ+KOMP+COL
```

```
Vorstellenden,Vorstellende.N
```

By adding the following 2 lines, the words will be correctly displayed under their lemma and in their full form in WiTTFind:

```
andersfärbig,.ADJ+KOMP+COL
```

```
Vorstellende,.N
```

A file with all the words for the semantic categories of music and color which full forms don't appear in WiTTFind but their lemmas do, can be found in the attached SD Card.

This chapter showed the methods that were used to evaluate the different frequency lists and pointed out some errors that are leading to a wrong lemmatization for the semantic frequency lists. The lexicon is continuously being improved, but, as mentioned above, this is a laborious task. On one hand, the lexicon should have only the entries that are really necessary, on the other, as shown above, some entries need to be added although their words don't appear as such in the *Nachlass*.

7. Final observations

The *Nachlass* of Ludwig Wittgenstein is an important heritage for the world. The CIS continues to develop the search engine WiTTFind that offers tools to analyze his works. One of the things that distinguishes this engine from others is the ability to search by semantic category. This thesis was develop in order to further improve the semantic search by generating lemmatized frequency lists for the semantic categories of color and music.

Some semantic frequency lists already exist, but these were created in the time were only the first 5.000 pages of the *Nachlass* were open to the public. Now that the 20.000 pages are open source, the frequencies had to be recalculated. The aim of this thesis was to create frequency lists for the semantic categories music and color taking as inputs all the pages of the *Nachlass*.

The programs to calculate the semantic frequencies were created in such way, that if the input files change, they can be called again to update the frequencies. To do so, the work flow of the deploy chain for WiTTFind was analyzed and the new programs follow its structure.

Different difficulties were encountered while creating the frequency lists. Some words are not being recognized by the tagger because they are being wrongly preprocessed. To fix this, the programs implemented in the course of this thesis use regular expressions to clean the tokens from possible XLM elements left on them and to fix words that have a hyphenation. It was also recognized that although most of the English words are correctly lemmatized, their tags are not being set correctly. The TreeTagger uses an auxiliary lexicon to tag the English words an in this lexicon most of the tags are set to FM (Foreign Language Material). Seven documents of the *Nachlass* are written in English and their parts of speech should be tagged as such. The English lexicon used by WiTTFind (auxlex.txt) should be expanded to encompass proper tags for all the words and the settings of the TreeTagger should be change depending on the language of the file. This exceeded the scope of this thesis but offers new research possibilities for the future.

Some discrepancies were recognized when evaluating the newly created frequency lists against the frequencies displayed in the search bar in WiTTFind. An analysis of the lexicon yield some answers. The programs to create the frequency lists use a regex to evaluate the semantic category of a word. If the word doesn't include a searched category, it isn't added to the lemmatized frequency list. It was discovered that some words are missing their semantic categories. A list of this word is included in the attached files and should help improve the lexicon.

It was also discovered that some words are not being found in WiTTFind by their full form but by their lemma. In order for the words to be lemmatized correctly, an entry for

their lemma in the lexicon is necessary. This morphological expansion can be done for the semantic categories of music and color with help of lists that can be found in the attached SD Card.

The frequency lists that were created in the course of this thesis encompass only the semantic categories of music and color. The programs were constructed in such way, that by changing one line containing a regular expression, lemmatized frequencies for other semantic categories can be created.

The semantic frequency lists generated only with help of the lexicon are not able to disambiguate some words. This limitation can be overcome by creating frequency lists that also take into account the POS-tag of a word. Further research in this area could significantly improve the semantic frequency lists.

Bibliography

[1] Azar, F. A.: *Semantische Annotation von Adjektiven im Big Typescript von Ludwig Wittgenstein*, 2017. dissertation.

[2] Burnard, L.: *What is the Text Encoding Initiative? How to add intelligent markup to digital resources.* OpenEdition Press, 2014, ISBN 9782821834606. `http://books.openedition.org/oep/426`, Accessed: 2018-11-18.

[3] *The center for information and language processing (cis).* `http://www.cis.uni-muenchen.de/ueber_uns/index.html`, Accessed: 2018-11-28.

[4] Gabler, H. W.: *Wittgensteins Nachlass. The Bergen Electronic Edition.* In: Henrikson, Paula; Janss, C. (Hrsg.): *Geschichte der Edition in Skandinavien. Bausteine zur Geschichte der Edition*, S. 156–165. De Gruyter, 2013. `https://epub.ub.uni-muenchen.de/17082/7/pre17082.pdf`, Accessed: 2018-20-18.

[5] *GNU Make.* `https://www.gnu.org/software/make/`, Accessed: 2018-11-21.

[6] Hosoya, H.: *Foundations of XML processing: the tree-automata approach.* Cambridge University Press, 2011, ISBN 9780511762093. `https://opacplus.bsb-muenchen.de/metaopac/singleHit.do?methodToCall=showHit&curPos=1&identifier=100_SOLR_SERVER_1926989969`.

[7] Krey, A. C.: *Semantische Annotation von Adjektiven im Big Typescript von Ludwig Wittgenstein*, 2013. dissertation.

[8] *lxml documentation*, 2018. `https://lxml.de/4.2/lxmldoc-4.2.5.pdf`, Accessed: 2018-11-16.

[9] Ray, E. T.: *Einführung in XML.* O'Reilly Verlag GmbH, 2001, ISBN 9783897212862.

[10] Schiller, A., S. Teufel und C. Stöckert: *Guidelines für das Tagging deutscher Textcorpora mit STTS (Kleines und großes Tagset)*, 1999. `http://www.sfs.uni-tuebingen.de/resources/stts-1999.pdf`, Accessed: 2018-11-16.

[11] Schmid, H.: *Treetagger - a part-of-speech tagger for many languages.* `http://www.cis.uni-muenchen.de/~schmid/tools/TreeTagger/`, Accessed: 2018-11-20.

[12] Schmid, H.: *Probabilistic part-of-speech tagging using desicion trees.* Proceedings of International Conference on New Methods in Language Processing, 1994. `http://www.cis.uni-muenchen.de/~schmid/tools/TreeTagger/data/tree-tagger1.pdf`, Accessed: 2018-11-16.

[13] Schmid, H.: *Improvements in part-of-speech tagging with an application to german.* Proceedings of the ACL SIGDAT-Workshop, 1995. `http://www.cis.uni-muenchen.de/~schmid/tools/TreeTagger/data/tree-tagger1.pdf`, Accessed: 2018-11-16.

[14] *The wittgenstein archives at the university of bergen (wab).* `http://wab.uib.no/`, Accessed: 2018-11-28.

[15] *Xml tutorial.* `https://www.w3schools.com/xml/default.asp`, Accessed: 2018-11-16.

List of Figures

List of Tables

Listings

A. Complete code

A.1. Download TreeTagger

A.1.1. Modified download-tree-tagger.sh

```
#!/bin/bash

LESS=${LESS:-less}

BASE_URL=http://www.cis.uni-muenchen.de/~schmid/tools/TreeTagger
DOWNLOAD_URL=${BASE_URL}/data
INSTALL_DIR=$1

unameOut="$(uname -s)"
case "${unameOut}" in
    Linux*)     machine=linux;
                appendNr="1.";;
    Darwin*)    machine=MacOSX;
                appendNr="";;
    *)          { echo 'Operating System uknown, download tagger from http://www.cis.uni-muenchen.de/~schmid/tools/TreeTagger' ; exit 1; };;
esac

FILES="tagger-scripts.tar.gz \
      install-tagger.sh \
      english-par-${machine}-3.2-utf8.bin.gz \
      german-par-${machine}-3.2-utf8.bin.gz \
      english-chunker-par-${machine}-3.2-utf8.bin.gz \
      german-chunker-par-${machine}-3.2-utf8.bin.gz \
      tree-tagger-${machine}-3.2.${appendNr}tar.gz"

echo "Before installing, please read TreeTagger's LICENSE first [OK]"

read

curl --silent ${BASE_URL}/Tagger-Licence | ${LESS}
echo "Do you agree to TreeTagger's license, i.e. you don't use TreeTagger commercially? (Yes/No/Whatever)"

read answer

case $answer in
y|Y|Yes) : ;;
n|N|No) exit 1 ;;
* ) exit 1 ;;
esac

echo "Creating install dir ${INSTALL_DIR}"
```

```
mkdir -p ${INSTALL_DIR} && cd $_

for file in $FILES
do
  curl -LO ${DOWNLOAD_URL}/$file
done

# Unpack things
find . -type f -iname \*tar.gz -exec tar xzf {} \;
find . -type f -iname \*bin.gz -exec gunzip -k -f {} \;

# Install tree-tagger
bash install-tagger.sh
```

A.2. Language finder

A.2.1. language_finder.make

```
LANGUAGE_FINDER_DIR      ?= $(CISWAB_TOOLS_DIR)/language_finder
LANGUAGE_FINDER_CMD      ?= language_finder.py
LANGUAGE_FINDER_STARTER  ?= $(PYTHON3_RUNNER)
LANGUAGE_FINDER_RUNNER   ?= $(LANGUAGE_FINDER_STARTER) \
                               $(LANGUAGE_FINDER_DIR)/$(LANGUAGE_FINDER_CMD)

info_language_finder:
	$(SILENT) echo -e "\n\n ################################  make language_finder INFO"
	$(SILENT) echo "PROCESSED LANGUAGE_FILES = $(TAGGED_UNEXPANDED_NORM_FILES) "

language-finder:
	$(SILENT) $(LANGUAGE_FINDER_RUNNER) $(LANGUAGE_FINDER_DIR)/${LANGUAGE_FINDER_CMD} $(TAGGED_UNEXPANDED_NORM_FILES)



language_finder-clean:
	$(SILENT) echo -e "\n Clean language_finder"
	$(SILENT) echo -e "\n language_finder = $(LANGUAGE_FINDER_DIR)/unknown_words"
	$(SILENT) echo -e "\n language_finder = $(LANGUAGE_FINDER_DIR)/unknown_words.txt"
	$(SILENT) $(RM) $(LANGUAGE_FINDER_DIR)/unknown_words
	$(SILENT) $(RM) $(LANGUAGE_FINDER_DIR)/unknown_words.txt
```

A.2.2. language_finder.py

1

```
import sys
from pathlib import Path
import os
import pickle
```

[1] Definition of dirs and sec_dirs array was suppressed for readability purposes. See attached SD Card for complete language_finder.py program

```
from collections import defaultdict
import ntpath

from lxml.etree import iterparse

from collections import OrderedDict
import re

args = sys.argv

# Check if a file path was given, else throw error and exit program
if len(args) < 2:
    print("Please give the file paths as arguments")
    sys.exit(1)

working_directory = os.path.dirname(args[1])

# Get all files passed by make (every file n gets passed as a args[n])
    therefore we need to put the files into a list
files = args[2:]
# Initialize dictionaries and lists
unknown_words_dict = defaultdict(set)
sec_dirs_de = []
sec_dirs_en = []
dirs_de = []
dirs_en = []

for file_path in files:

    # Check if the path leads to a file, else throw error and try with next
    file
    if not Path(file_path).is_file():
        print(file_path + " is not a File")
        continue

    # get the name of the file to be parsed. ntpath works also with windows.
    parsed_file = ntpath.basename(file_path)

    nr_known_words = 0
    nr_unknown_words = 0

    tagged_w = 0
    not_tagged_w = 0

    iterator = iterparse(file_path)
    for _, element in iterator:
        if '}' in element.tag:  # if there is a namespace in the tag then
    remove it
            element.tag = element.tag.split('}', 1)[1]  # strip all
    namespaces
        if element.tag == 'w':
            if element.get("l") == 'UNKNOWN':
                if "type=\"notation\"" not in element.text:
                    word = element.text

```

```
                    # if inside the word there is an underscore followed by a lowercase letter, it should be deleted
                    word = re.sub(r"(.*)-([a-zaäoöuü].*)", r"\1\2", word)
                    # delete <lb> tag
                    word = re.sub(r"<lb .+?>", r"", word)
                    # delete <sp/>
                    word = re.sub(r"<sp/>", "", word)
                    # delete
                    word = re.sub(r"<pb.*?/>", "", word)
                    # delete <seg> tags
                    word = re.sub(r"<seg .+?>", r"", word)
                    word = re.sub(r"</seg>", "", word)

                    nr_unknown_words += 1
                    # Prove that value of the key is a set before adding an element to it
                    if type(unknown_words_dict[word]) != set:
                        unknown_words_dict[word] = {parsed_file}
                    else:
                        unknown_words_dict[word].add(parsed_file)
            else:
                nr_known_words += 1
            if element.get('t') == 'FM':
                not_tagged_w += 1
            else:
                tagged_w +=1



    original_file = re.sub("_NORM-tagged\.xml$", "", parsed_file)
    if tagged_w > not_tagged_w:
        if original_file in dirs:
            dirs_de.append(original_file)
        elif original_file in sec_dirs:
            sec_dirs_de.append(original_file)
    elif not_tagged_w > tagged_w:
        if original_file in dirs:
            dirs_en.append(original_file)
        elif parsed_file in sec_dirs:
            sec_dirs_en.append(original_file)

print("sec_dirs_de= {}".format(sec_dirs_de))
print("dirs_de= {}".format(dirs_de))
print("sec_dirs_en= {}".format(sec_dirs_en))
print("dirs_en= {}".format(dirs_en))

unknown_file_path = working_directory + "/unknown_words"
unknown_words_file = open(unknown_file_path, 'wb')
sorted_unknown = OrderedDict(sorted(unknown_words_dict.items()))
pickle.dump(dict(sorted_unknown), unknown_words_file)
unknown_words_file.close()

with open(working_directory + "/unknown_words.txt", 'w') as file:
    for key, values in sorted_unknown.items():
        file.write("{} in: {}\n".format(key, ' '.join(values)))
```

A.3. Frequency lists

A.3.1. sematic_freqlist.make

```
FREQ_ALL_DIR     ?= $(CISWAB_TOOLS_DIR)/frequency
FREQ_ALL_CMD     ?= all_freqlist.py
ALL_FREQLIST     ?= $(CISWAB_TOOLS_DIR)/frequency/all_frequencies.txt
ALL_FREQ_PICKLE ?= $(CISWAB_TOOLS_DIR)/frequency/all_frequencies_pickle
FREQ_MUSIC_DIR  ?= $(CISWAB_TOOLS_DIR)/frequency/music
FREQ_MUSIC_CMD  ?= music_freqlist.py
MUSIC_FREQLIST  ?= $(EXPORT_DIR)/lexikon/music/lemmatized_music_frequencies.
    txt
FREQ_COLOR_DIR  ?= $(CISWAB_TOOLS_DIR)/frequency/color
FREQ_COLOR_CMD  ?= color_freqlist.py
COLOR_FREQLIST  ?= $(EXPORT_DIR)/lexikon/color/lemmatized_color_frequencies.
    txt

info_semantic_freqlist:
	$(SILENT) echo -e "\n\n############################## make
    semantic_freqlist INFO"
	$(SILENT) echo "FREQ_ALL_DIR = $(FREQ_ALL_DIR)"
	$(SILENT) echo "FREQ_MUSIC_DIR = $(FREQ_MUSIC_DIR)"
	$(SILENT) echo "FREQ_COLOR_DIR = $(FREQ_COLOR_DIR)"
	$(SILENT) echo "DICT_WITT = $(DICT_WITT)"
	$(SILENT) echo "ALL_FREQLIST = $(ALL_FREQLIST)"
	$(SILENT) echo "MUSIC_FREQLIST = $(MUSIC_FREQLIST)"
	$(SILENT) echo "COLOR_FREQLIST = $(COLOR_FREQLIST)"
	$(SILENT) echo "TAGGED_UNEXPANDED_NORM_FILES = $(
    TAGGED_UNEXPANDED_NORM_FILES)"

TAGGED_UNEXPANDED_NORM_FILES   = $(shell find -L $(NACHLASS_DIR)/*/norm -
    type f -name \*.xml | grep '\-tagged' | grep -v '\expanded')

semantic_freqlist: all-freqlist music-freqlist color-freqlist

all-freqlist: $(TAGGED_UNEXPANDED_NORM_FILES)
	$(SILENT) $(PYTHON3_RUNNER) $(FREQ_ALL_DIR)/$(FREQ_ALL_CMD) $(ALL_FREQLIST
    ) $(ALL_FREQ_PICKLE) $^
	$(SILENT) echo "written $(ALL_FREQLIST)"
	$(SILENT) echo "written $(ALL_FREQ_PICKLE)"

music-freqlist:
	$(SILENT) $(PYTHON3_RUNNER) $(FREQ_MUSIC_DIR)/$(FREQ_MUSIC_CMD) $(
    DICT_WITT) $(ALL_FREQ_PICKLE) $(MUSIC_FREQLIST)
	$(SILENT) echo "written $(MUSIC_FREQLIST)"

color-freqlist:
	$(SILENT) $(PYTHON3_RUNNER) $(FREQ_COLOR_DIR)/$(FREQ_COLOR_CMD) $(
    DICT_WITT) $(ALL_FREQ_PICKLE) $(COLOR_FREQLIST)
	$(SILENT) echo "written $(COLOR_FREQLIST)"

semantic_freqlist-clean:
	$(SILENT) echo -e "\n Clean all_freqlist = $(ALL_FREQLIST)"
	$(SILENT) echo -e "\n Clean all_freqlist = $(ALL_FREQ_PICKLE)"
	$(SILENT) echo -e "\n Clean Music_freqlist = $(MUSIC_FREQLIST)"
```

```
	$(SILENT) echo -e "\n Clean Color_freqlist = $(COLOR_FREQLIST)"
	$(SILENT) $(RM) $(ALL_FREQLIST)
	$(SILENT) $(RM) $(ALL_FREQ_PICKLE)
	$(SILENT) $(RM) $(MUSIC_FREQLIST)
	$(SILENT) $(RM) $(COLOR_FREQLIST)
```

A.3.2. all_freqlist.py

```
import sys
from pathlib import Path
from collections import defaultdict
import ntpath
from lxml.etree import iterparse
import pickle

import string
import re

args = sys.argv

# Check if a file path was given, else throw error and exit program
if len(args) < 3:
    print("Please give the file paths as arguments")
    sys.exit(1)

out_file = args[1]
out_pickle = args[2]
# Get all files passed by make (every file n gets passed as a args[n])
    therefore we need to put the files into a list
files = args[3:]

all_words_freqs = defaultdict(int)
all_punctuation = string.punctuation + "”×‘–’‘„‟…"

for file_path in files:

    # Check if the path leads to a file, else throw error and try with next
    file
    if not Path(file_path).is_file():
        print(file_path + " is not a File")
        continue

    # get the name of the file to be parsed. ntpath works also with windows.
    parsed_file = ntpath.basename(file_path)
    print("processed file: {}".format(parsed_file))

    iterator = iterparse(file_path)
    for _, element in iterator:
        if '}' in element.tag:  # if there is a namespace in the tag then
    remove it
            element.tag = element.tag.split('}', 1)[1]  # strip all
    namespaces
        if element.tag == 'w':
            if "type=\"notation\"" not in element.text:
                word = element.text

```

```
                # if inside the word there is an underscore followed by a lowercase letter, it should be deleted
                word = re.sub(r"(.*)-([a-zaäoöuü].*)", r"\1\2", word)
                # delete <lb> tag
                word = re.sub(r"<lb .+?>", r"", word)
                # delete <sp/>
                word = re.sub(r"<sp/>", "", word)
                # delete
                word = re.sub(r"<pb.*?/>", "", word)
                # delete <seg> tags
                word = re.sub(r"<seg .+?>", r"", word)
                word = re.sub(r"</seg>", "", word)

                if word[0] not in all_punctuation:
                    all_words_freqs[word] += 1

all_sorted = dict(sorted(all_words_freqs.items(), key=lambda x: x[1], reverse=True))

all_words_pickle = open(out_pickle, 'wb')
pickle.dump(dict(all_sorted), all_words_pickle)
all_words_pickle.close()

with open(out_file, 'w') as file:
    for key, value in all_sorted.items():
```

A.3.3. color_freqlist.py

```
import sys
from pathlib import Path
from collections import defaultdict
import pickle
import re
import os

args = sys.argv

# Check if a file path was given, else throw error and exit program
if len(args) < 3:
    print("Pass the dictionary, the frequency list and the output file ")
    sys.exit(1)

lexicon = args[1]
all_frequencies = args[2]
out_file = args[3]

# check that the dictionary and and the frequencies are files
if not Path(lexicon).is_file():
    print(lexicon + " is not a valid file")
    sys.exit(1)
if not Path(all_frequencies).is_file():
    print(all_frequencies + " is not a valid file")
    sys.exit(1)

with open(all_frequencies, 'rb') as handle:
    all_frequencies = pickle.load(handle)
```

```

color_freqs = defaultdict(lambda: defaultdict(int))

with open(lexicon, 'r') as witt_lex:
    for entry in witt_lex:
        col = re.match("(.*),(.*)\..*\+COL", entry)
        if col:
            # lstrip is used because some words in the dictionary have leading spaces
            full_form = col.group(1).lstrip()
            lemma = col.group(2).lstrip()
            if not lemma:
                lemma = full_form
            if full_form in all_frequencies:
                color_freqs[lemma][full_form] = all_frequencies[full_form]

color_dir = os.path.dirname(out_file)
if not os.path.exists(color_dir):
    os.makedirs(color_dir)

with open(out_file, 'w') as file:
    for lemma, full_forms in color_freqs.items():
        sum_of_freqs = sum(full_forms.values())
        line = "{},{};".format(lemma, sum_of_freqs)
        sorted_values = dict(sorted(full_forms.items(), key=lambda x: x[1], reverse=True))
        for full_form, freq in sorted_values.items():
            line += " {},{};".format(full_form, freq)
        line += "\n"
        file.write(line)
```

A.3.4. music_freqlist.py

```
import sys
from pathlib import Path
from collections import defaultdict
import pickle
import re
import os

args = sys.argv

# Check if a file path was given, else throw error and exit program
if len(args) < 3:
    print("Pass the dictionary, the frequency list and the output file ")
    sys.exit(1)

lexicon = args[1]
all_frequencies = args[2]
out_file = args[3]

# check that the dictionary and and the frequencies are files
if not Path(lexicon).is_file():
    print(lexicon + " is not a valid file")
    sys.exit(1)
if not Path(all_frequencies).is_file():
```

```
24      print(all_frequencies + " is not a valid file")
25      sys.exit(1)
26
27  with open(all_frequencies, 'rb') as handle:
28      all_frequencies = pickle.load(handle)
29
30  music_freqs = defaultdict(lambda: defaultdict(int))
31
32  with open(lexicon, 'r') as witt_lex:
33      for entry in witt_lex:
34          music = re.match("(.*),(.*)\..*\+MUSIK", entry)
35          if music:
36              # lstrip is used because some words in the dictionary have
        leading spaces
37              full_form = music.group(1).lstrip()
38              lemma = music.group(2).lstrip()
39              if not lemma:
40                  lemma = full_form
41              if full_form in all_frequencies:
42                  music_freqs[lemma][full_form] = all_frequencies[full_form]
43
44  music_dir = os.path.dirname(out_file)
45  if not os.path.exists(music_dir):
46      os.makedirs(music_dir)
47
48  with open(out_file, 'w') as file:
49      for lemma, full_forms in music_freqs.items():
50          sum_of_freqs = sum(full_forms.values())
51          line = "{},{};".format(lemma, sum_of_freqs)
52          sorted_values = dict(sorted(full_forms.items(), key=lambda x: x[1],
        reverse=True))
53          for full_form, freq in sorted_values.items():
54              line += " {},{};".format(full_form, freq)
55          line += "\n"
56          file.write(line)
```

Contents of the SD Card

- PDF copy of this thesis
- Latex code for this thesis
- Code for the Python scripts
- Code for makefile scripts
- simple Makefile example
- Frequency list over all words in the Nachlass
- Lemmatized frequency list color
- Lemmatized frequency list music
- List of unknown words
- List of words evaluated
- List of words that are missing semantic category color
- List of words that are missing semantic category music
- List of words which morphology needs to be extended for color
- List of words which morphology needs to be extended for music