Universität Bielefeld

Fakultät für Linguistik und Literaturwissenschaft

Masterarbeit

im Studiengang Interdisziplinäre Medienwissenschaft

zum Thema:

Automatische Auszeichnung und linguistische Analyse der Bundestagsplenarprotokolle von 1949 bis 2017 für interaktive Darstellung von N-Grammen und weiteren Statistiken in einer Webanwendung

vorgelegt von

Stephan Porada

Erstgutachter/in: Herr Prof. Dr. David Schlangen

Zweitgutachter/in: Frau Dr. Sina Zarrieß

Bielefeld, im März 2019

Kurzzusammenfassung

Das Ziel dieser Masterarbeit ist es, die von der Bundesregierung bereitgestellten Plenarprotokolle automatisch mit Informationen und Metadaten auszuzeichnen. Für die automatische Auszeichnung der Protokolle wurde eine eigene Software entwickelt und mit dieser alle Protokolle von 1949 bis 2017 ausgezeichnet. Der Fokus der automatischen Auszeichnung liegt insbesondere auf der eindeutigen Identifikation von Redner und Rednerinnen innerhalb der Protokolle, damit diesen ihre jeweiligen Redebeiträge zugeordnet werden können.

Auf Grundlage der automatisch ausgezeichneten Protokolle werden verschiedene N-Gramme berechnet, die in einer eigenen entwickelten Webanwendung dargestellt werden. Die Darstellung der N-Gramme in der Webanwendung kann mit der Funktionsweise des Google Ngram Viewers verglichen werden.

Eine weitere Funktion der Webanwendung ist es die einzelnen Reden aller Redner sowie die gesamten Protokolle übersichtlich und strukturiert bereitzustellen, damit Nutzer diese für Recherchen durchsuchen und lesen können.

Inhaltsverzeichnis

Abbildungsverzeichnis

Quellcodeverzeichnis

Abkürzungsverzeichnis

PDok	Parlamentsdokumentation
MdB	Mitglied des Deutschen Bundestags
Regex	regulärer Ausdruck
XML	Extensible Markup Language
WSGI	Web Server Gateway Interface
App	Application
API	Programmierschnittstelle
HTTPS	Hypertext Transfer Protocol Secure

1 Anlass für das Projekt

Der Deutsche Bundestag trat zum ersten mal am 7. September 1949 zusammen. Beginnend mit dieser ersten Sitzung wurde bis heute kontinuierlich ohne Unterbrechung Protokoll über diese und deren Inhalte geführt. [8] Diese Protokolle werden vom Deutschen Bundestag online für alle Bürger und Bürgerinnen bereitgestellt.
Offizielle Anlaufstelle ist das elektronische Archiv Parlamentsdokumentation (PDok)[1]. Hier können alle Plenarprotokolle im PDF-Format eingesehen und nach Stichworten durchsucht werden. Der Fokus der Plattform liegt klar darauf, dass die Daten in einem für den Menschen lesbaren Format vorliegen und veröffentlicht werden.

Für einfache Recherchen ist diese Art der Datenaufbereitung und Darstellung ausreichend. Sollen jedoch computergestützte Analyseverfahren angewandt werden, ist es umständlich mit PDF-Dateien zu arbeiten. Zwar wurden die Dokumente einer Texterkennung unterzogen, aber ein Computer kann nicht erkennen an welcher Stelle ein Redner eingeführt wird oder wo das Inhaltsverzeichnis beginnt und endet. Menschen können dies allein durch die unterschiedliche visuelle Formatierung des Textes erkennen. Die PDF-Dateien sind keine strukturierten Daten und somit nicht maschinenlesbar. [36]

Um dieser Problematik entgegenzukommen hat die Bundesregierung im Rahmen einer Open-Data Initiative Protokolle der 1. bis 18. Wahlperiode im Extensible Markup Language (XML)-Format veröffentlicht. [35] Somit liegen die Protokolle in maschinenlesbarer Form vor. Allerdings sind die Protokolle nur äußerst rudimentär mit Informationen ausgezeichnet. Bis auf wenige Metadaten über den Zeitpunkt der Sitzung und die Sitzungsnummer, können keine weiteren Informationen maschinell aus den XML-Protokollen extrahiert werden.

[1] http://pdok.bundestag.de/

Erstes Ziel dieser Masterarbeit und dieses Projekts ist es, die vorliegenden XML-Protokolle automatisch mit weiteren Informationen auszuzeichnen, so dass komplexere maschinengestützte Analysen durchgeführt werden können. Zweites Ziel ist es, die Protokolle und die darin enthaltenen automatisch hinzugefügten Daten in einer Webanwendung darzustellen und für die Darstellung von N-Grammen zu nutzen.

2 Situationsanalyse: Datengrundlage und Ausgangsposition

2.1 Ausgangsdaten: Plenarprotokolle als XML

Die im Rahmen dieses Projekts verwendeten Bundestagsplenarprotokolle wurden am 14.10.2018 heruntergeladen. Die Protokolle der 1. bis 18. Wahlperiode wurden von Seiten der Bundesregierung zuletzt am 09.07.2015 geändert. Die Protokolle der 18. Wahlperiode wurden zuletzt am 09.01.2018 geändert. Bei den heruntergeladenen XML-Protokollen handelt es sich um die rudimentär ausgezeichneten Protokolle, welche als Input für die Software zur automatischen Auszeichnung dienen. Quelle der Daten ist die offizielle Seite des Deutschen Bundestags. [35] Die Protokolle können ebenfalls im zugehörigen Repository (Kapitel 8) heruntergeladen werden. Insgesamt umfassen die Protokolle den Zeitraum vom 7. September 1949 bis zum 5. September 2017. Der gesamte Ausgangskorpus umfasst 4106 einzelne XML-Protokolle. Eine genaue Aufschlüsselung pro Wahlperiode kann folgender Liste entnommen werden:

1. Wahlperiode: 282
2. Wahlperiode: 227
3. Wahlperiode: 168
4. Wahlperiode: 198
5. Wahlperiode: 247
6. Wahlperiode: 199
7. Wahlperiode: 259
8. Wahlperiode: 230
9. Wahlperiode: 142

10. Wahlperiode: 256
11. Wahlperiode: 236
12. Wahlperiode: 243
13. Wahlperiode: 248
14. Wahlperiode: 253
15. Wahlperiode: 187
16. Wahlperiode: 233
17. Wahlperiode: 253
18. Wahlperiode: 245

Die Problematik der Dateien ist, dass diese wie bereits angemerkt, nur sehr minimal ausgezeichnet sind.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <DOKUMENT>
3    <WAHLPERIODE>18</WAHLPERIODE>
4    <DOKUMENTART>PLENARPROTOKOLL</DOKUMENTART>
5    <NR>18/245</NR>
6    <DATUM>05.09.2017</DATUM>
7    <TITEL>Plenarprotokoll vom 05.09.2017</TITEL>
8    <TEXT>Plenarprotokoll 18/245
9
10 Deutscher Bundestag
11 Stenografischer Bericht
12
13 245. Sitzung
14
15 Berlin, Dienstag, den 5. September 2017
```

Quellcode 2.1: Gekürztes Beispiel eines offiziellen Protokolls (protocols_raw_xml/18_Wahlperiode_2013-2017/18245.xml)

In Quellcode 2.1 sind alle verwendeten XML-Tags ersichtlich. Lediglich die schließenden Tags </DOKUMENT> und </TEXT> sind nicht sichtbar. Innerhalb des Tags <TEXT> befindet sich das gesamte Protokoll der Sitzung, sowie Inhaltsverzeichnis und Anhang ohne weitere Auszeichnung und Strukturierung. Es lassen sich lediglich die Wahlperiode, die Dokumentenart, die Sitzungsnummer, das Datum und der Titel

automatisiert auslesen.
Die offizielle Auszeichnung kann somit nur als minimal bezeichnet werden. Folgende Informationen und Strukturen lassen sich nicht automatisch erkennen:

- Redner/Abgeordneter/MdB
  - Redner-ID
  - Vorname
  - Nachname
  - Akademischer Titel
  - Adelstitel
  - Fraktionszugehörigkeit
  - Parteizugehörigkeit
- Reden und Redebeiträge
  - Rede-ID
  - Kommentare und Zwischenrufe innerhalb dieser
- Inhaltsverzeichnis
- Anlage
- Enduhrzeit der Sitzung
- Startuhrzeit der Sitzung

2.1.1 Fehler in den XML-Protokollen

Neben der eigentlichen Herausforderung, die Protokolle automatisch und sinnvoll auszuzeichnen, gibt es zwei weitere Faktoren, welche dieses Vorhaben erschweren.
Zum einem sind in den Protokollen Tipp- und Flüchtigkeitsfehler zu finden, die es besonders in Hinsicht auf die Arbeit mit regulären Ausdrücke erschweren, fehlerfrei automatisch eine Struktur innerhalb der Protokolle zu erfassen.
Kritischer zu betrachten sind die Protokolle der 15., 16. und 17. Wahlperiode, welche von der Bundesregierung fehlerhaft abgespeichert wurden.

I

–

C

```
1920
1921 D
1922 d
1923
1924 W
1925
1926 D
1927
1928 D
1929
1930 e
1931 E
1932 m
1933 9
1934 t
1935 b
1936 a
1937 s
1938 (C
1939
1940 (D
1941
1942 Ich kann heute hier nur feststellen: Recht hat er ge-
1943 habt, der Kollege Rüttgers
1944
1945 sagt der Kollege Schmidt –,
1946
1947 auch wenn ich zugeben muß: Das haben wir erst ein
1948 bißchen später richtig mitgekriegt.
1949
1950 (Heiterkeit und Beifall bei Abgeordneten des
1951 BÜNDNISSES 90/DIE GRÜNEN, der FDP
1952 und der LINKEN)
1953
1954 mmerhin Selbstkritik!
1955
```

Quellcode 2.2: Fehlerhaftes offizielles Protokoll (faulty_raw_xml/16_Wahlperiode_2005-2009/16001.xml)

In den Zeilen 1916 bis 1937 von Quellcode 2.2 ist zu sehen, dass pro Zeile einzelne

Buchstaben in die Datei geschrieben wurden. In Zeile 1954 ist zu sehen, dass der Buchstabe „I“ aus Zeile 1916 fehlt, da die Zeichenkette „mmerhin“ das Wort „Immerhin“ ergeben müsste. Dieser Fehler zieht sich pro Buchstabe durch den nachfolgenden Absatz und weiter durch das gesamte Protokoll.

Die Abteilung, welche für die Plenarprotokolle im Bundestag zuständig ist, wurde im Laufe dieser Arbeit auf die fehlerhafte Speicherung der XML-Protokolle hingewiesen. Allerdings wurden erst zum Ende der Arbeit neue korrekt abgespeicherte XML-Protokolle veröffentlicht. Es war somit zeitlich nicht möglich, diese zu verwenden.
Wie sich diese Problematiken auf die Fehlerquote der automatischen Auszeichnung und die Berechnung der N-Gramme auswirkt, wird in Kapitel 6 genauer beschrieben.

2.2 XML-Protokolle der aktuellen 19. Wahlperiode

Ab der aktuellen 19. Wahlperiode werden die Plenarprotokolle im Gegensatz zu den vorherigen Perioden von der Bundesregierung umfassend ausgezeichnet und veröffentlicht. [37, 35]

```
233       <ivz-block-titel>Anlage 6</ivz-block-titel>
234       <ivz-eintrag>
235         <ivz-eintrag-inhalt>Amtliche Mitteilungen </ivz-eintrag-inhalt>
236         <a href="S35" typ="druckseitennummer"><seite>35</seite>
            ↪   <seitenbereich>A</seitenbereich></a>
237       </ivz-eintrag>
238     </ivz-block>
239   </inhaltsverzeichnis>
240 </vorspann>
241 <sitzungsverlauf>
242   <sitzungsbeginn sitzung-start-uhrzeit="11:00">
243     <a id="S1" name="S1" typ="druckseitennummer"/>
244
245   </sitzungsbeginn>
246   <tagesordnungspunkt top-id="Tagesordnungspunkt 1">
247     <rede id="ID19100100">
248       <p klasse="redner">
249
```

```
250     <redner id="11002190"><name><vorname>Alterspräsident Dr.
        ↪ Hermann</vorname><nachname>Otto
        ↪ Solms</nachname><rolle><rolle_lang>Alterspräsident</rolle_lang>
251     <rolle_kurz>Alterspräsident</rolle_kurz></rolle></name></redner>
252
253     Alterspräsident Dr. Hermann Otto Solms:</p>
254     <p klasse="J_1">Guten Morgen, liebe Kolleginnen und Kollegen! Nehmen Sie bitte Platz.</p>
```

Quellcode 2.3: Gekürzte offizielle neue Auszeichnung der 19. Wahlperiode (current_official_protocols_xml/19001.xml)

In Quellcode 2.3 ist zu sehen, dass mittels der neuen Auszeichnung Reden und Redner strukturiert voneinander getrennt sind. Zeile 247 leitet die Rede mit dem Element <rede> ein, welches in Zeile 250 das Kind <redner> untergeordnet wird. Innerhalb des Elements <redner> befinden sich weitere Kinderelemente, die es unter anderem ermöglichen automatisch den Vor- und Nachnamen eines Redners zu erfassen. Die Elemente <rede> und <redner> sind beide mit eindeutigen IDs durch das Attribut @id ausgezeichnet. Durch diese können gegebenenfalls noch weitere Informationen automatisiert ermittelt werden. Es ist ebenfalls ersichtlich, dass das Inhaltsverzeichnis vom Inhalt der eigentlichen Sitzung getrennt ist, da in Zeile 239 das Element <inhaltsverzeichnis> geschlossen wird.

Diese neue Auszeichnungssyntax soll als Vorlage dienen, an der sich die eigene automatische Auszeichnung der Protokolle orientiert. Für die neue Auszeichnungssyntax existiert eine von der Bundesregierung offiziell veröffentlichte Dokumenttypdefinition. [18] Diese soll ebenfalls dabei helfen, eine sinnvolle automatisch erstellte Auszeichnungssyntax für die Protokolle zu entwickeln.
Mehr zu den Zielen dieses Projekts und der Arbeit in Kapitel 3.

2.3 Stammdaten der Abgeordneten

Neben den XML-Protokollen veröffentlicht die Bundesregierung auch die Stammdaten aller Mitglieder des Deutschen Bundestags (MdBs), die seit 1949 im Bundestag

vertreten waren. Diese Daten werden ebenfalls im XML-Format veröffentlicht. [48]

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE DOCUMENT SYSTEM "MDB_STAMMDATEN.DTD">
<!--Erstellt am: 03.10.2018 22:01:26-->
<DOCUMENT>
 <VERSION>1538624713</VERSION>
 <MDB>
  <ID>11000001</ID>
  <NAMEN>
   <NAME>
    <NACHNAME>Abelein</NACHNAME>
    <VORNAME>Manfred</VORNAME>
    <ORTSZUSATZ/>
    <ADEL/>
    <PRAEFIX/>
    <ANREDE_TITEL>Dr.</ANREDE_TITEL>
    <AKAD_TITEL>Prof. Dr.</AKAD_TITEL>
    <HISTORIE_VON>19.10.1965</HISTORIE_VON>
    <HISTORIE_BIS/>
   </NAME>
  </NAMEN>
  <BIOGRAFISCHE_ANGABEN>
   <GEBURTSDATUM>1930</GEBURTSDATUM>
   <GEBURTSORT>Stuttgart</GEBURTSORT>
   <GEBURTSLAND/>
   <STERBEDATUM>2008</STERBEDATUM>
   <GESCHLECHT>männlich</GESCHLECHT>
   <BERUF>Rechtsanwalt, Wirtschaftsprüfer, Universitätsprofessor</BERUF>
   <PARTEI_KURZ>CDU</PARTEI_KURZ>
   <VITA_KURZ/>
   <VEROEFFENTLICHUNGSPFLICHTIGES/>
  </BIOGRAFISCHE_ANGABEN>
  <WAHLPERIODEN>
   <WAHLPERIODE>
    <WP>5</WP>
    <MDBWP_VON>19.10.1965</MDBWP_VON>
    <MDBWP_BIS>19.10.1969</MDBWP_BIS>
    <WKR_NUMMER>174</WKR_NUMMER>
    <WKR_NAME/>
    <WKR_LAND>BWG</WKR_LAND>
    <LISTE/>
    <MANDATSART>Direktwahl</MANDATSART>
```

```
42        <INSTITUTIONEN>
43          <INSTITUTION>
44            <INSART_LANG>Fraktion/Gruppe</INSART_LANG>
45            <INS_LANG>Fraktion der Christlich Demokratischen Union/Christlich - Sozialen
              ↪ Union</INS_LANG>
46            <MDBINS_VON/>
47            <MDBINS_BIS/>
48            <FKT_LANG/>
49            <FKTINS_VON/>
50            <FKTINS_BIS/>
51          </INSTITUTION>
52        </INSTITUTIONEN>
53      </WAHLPERIODE>
```

Quellcode 2.4: Auszug aus den Stammdaten aller MdBs seit 1949 (MdB_data/MdB_Stammdaten.xml)

In Quellcode 2.4 ist der Eintrag eines MdB zu sehen. Jedes aktuelle und ehemalige MdB wird durch ein eigenes <MDB>-Element repräsentiert. Kinder des Elements <MDB> sind <ID>, <NAMEN>, <BIOGRAFISCHE_ANGABEN> und <WAHLPERIODEN>. Diesen sind jeweils weitere Kinder untergeordnet, welche die dazugehörigen Daten (Vorname, Geburtsdatum, Wahlperiode etc.) einer Person enthalten. Die ID ist eindeutig einer einzelnen Person zugeordnet.

Mit Hilfe der Stammdaten aller MdBs kann die automatische Auszeichnung von Namen, Parteizugehörigkeiten etc. innerhalb der Protokolle realisiert werden. Mehr dazu in Kapitel 4.4.3.

3 Zielsetzung und Aufbau des Projekts

Der Ablauf des Projekts ist in zwei Phasen unterteilt. Die erste Phase ist die der Datenaufbereitung während in der zweiten Phase die aufbereiteten und erstellten Datensätze für Darstellungen und Analysetools in der Webanwendung genutzt werden.
Aus den zwei Phasen lassen sich die jeweiligen Teilziele ableiten, welche erfüllt werden müssen, um sowohl eine erfolgreiche automatische Auszeichnung und die Darstellung der Daten in der Webanwendung für den Endnutzer zu gewährleisten.

I. Datenaufbereitung
    1. Automatische Auszeichnung der offiziellen Protokolle
    2. Lemmatisierung und Tokenisierung der Reden und Redebeiträge aller MdBs
    3. Berechnen von 1- bis 5-Grammen gruppiert nach Jahren, Sprechern etc.
II. Darstellung der Daten, Protokolle und Redebeiträge in einer Webanwendung
    1. Erstellen einer Webanwendung
    2. Darstellung der Protokolle und Reden verknüpft mit Metadaten
    3. Entwicklung des N-Gramm Viewers
    4. Bereitstellen der Daten aus Phase I für die Nachnutzung

Das erste Hauptziel und somit die erste Phase des Projekts befasst sich mit der Aufbereitung der Ausgangsdaten. In Phase II werden die in Phase I erzeugten Daten als Datengrundlage für eine Webanwendung verwendet. Mit dieser solle es Endnutzern möglich sein, die Protokolle und Redebeiträge im Kontext von Metadaten zu analysieren.

Hinsichtlich der automatischen Auszeichnung wird versucht, sich der offiziellen Auszeichnungssyntax der Bundesregierung für die aktuellen Protokolle der 19. Wahlperiode anzunähern (siehe Kapitel 2.2).

Es ist nicht angedacht, dass die automatische Auszeichnung der Protokolle, die gesamte Syntax beziehungsweise das gesamte Tagset der aktuellen offiziellen Auszeichnungssyntax verwenden wird. Dies ist zum eine nicht möglich, da zum Beispiel Referenzen aus dem Inhaltsverzeichnis zu den passenden Tagesordnungspunkten im Sitzungsverlauf nur händisch gesetzt werden können. Auch ist es nicht das Ziel dieser Arbeit, das Inhaltsverzeichnis und die Anlage detailliert auszuzeichnen. Der Fokus liegt darauf, dass Reden und die dazugehörigen Redner erkennbar sind. Der nachfolgenden Liste kann entnommen werden, welche Informationen genau automatisch erkannt und ausgezeichnet werden sollen.

- Redner/Abgeordneter/MdB
  - Redner-ID
  - Vorname
  - Nachname
  - Akademischer Titel
  - Adelstitel
  - Fraktionszugehörigkeit
  - Parteizugehörigkeit
- Reden und Redebeiträge
  - Rede-ID
  - Kommentare und Zwischenrufe innerhalb dieser
- Inhaltsverzeichnis
- Anlage
- Enduhrzeit der Sitzung
- Startuhrzeit der Sitzung

Sind diese Merkmale erkannt und ausgezeichnet, können so Reden und Redebeiträge einzeln lemmatisiert und einer Tokenisierung unterzogen werden, so dass zwei verschiedene Grundkorpora entstehen. Zusätzlich wird es zu jedem Korpus ein Version mit und ohne Sotpwörter geben. Diese vier Korpora können dann genutzt werden, um N-Gramme gruppiert nach Jahren, Rednern oder auch Parteien zu berechnen. Wie diese Korpora genau erstellt werden und welche Vorteile die jeweiligen mit sich bringen wird in Kapitel 4.6 erläutert.

Ziel hinsichtlich der Webanwendung ist es für den Endnutzer verschiedene Tools zur

Verfügung zu stellen, mit deren Hilfe die Protokolle, Reden, Redner sowie dazugehörige Metadaten in Bezug zu einander gesetzt und analysiert werden können.
Hierfür soll es möglich sein, alle Protokolle von 1949 bis 2017 lesen zu können. Ebenfalls wird es möglich sein alle einzelnen Reden und Redebeiträge seit 1949 durchsuchen und aufrufen zu können. Zusätzlich dazu wird es möglich sein die Profile aller MdBs seit 1949 durchsuchen zu können. Eine Profilseite enthält für jedes MdB unter anderem Informationen darüber bereit, welche Rede dieses wann in welchem Kontext gehalten hat.

Umfassendere Analysen sollen mittels eines Ngram Viewers durchgeführt werden können, welcher sich in seiner Funktionsweise am Google Ngram Viewer[1] orientiert.

Momentan werden nur die Protokolle der 1. bis 18. Wahlperiode (1949 bis 2017) verwendet. Ein weiteres langfristiges Ziel über die Arbeit hinaus wird es sein, auch die Protokolle der 19. und kommenden Wahlperioden mit zu nutzen. Da sich die automatische Auszeichnung an der offiziellen neuen Auszeichnungssyntax orientiert, wird dies voraussichtlich mit nur wenigen Änderungen möglich sein.

Im Sinne der Nachnutzbarkeit sollen alle Daten, die im Verlauf des Projekts erzeugt und verwendet wurden, den Nutzern zugänglich gemacht werden. So können eventuell neue Projekte entstehen, die andere Analysen und Ergebnisse produzieren. Unter diesem Gesichtspunkt wurde auch die Entscheidung getroffen, dass der Code und die dazugehörige Dokumentation sowie Readmes in Englisch verfasst sind. Zwar befasst sich das Projekt mit deutschen Daten, aber eventuell können einige Verfahren und Ideen auf ähnliche Aufgabenstellungen, Probleme und oder Projekte übertragen werden.

[1] Link zum Google Ngram Viewer: https://books.google.com/ngrams

4 Datenaufbereitung: Funktionsweise der Software

Dieses Kapitel beschreibt die grundlegende Funktionsweise der Software für die automatische Auszeichnung und N-Gramm-Berechnung sowie deren Entwicklung. Es wird nicht jede Funktion ins kleinste Detail besprochen, sondern grundlegend erläutert, wie die automatische Auszeichnung der Protokolle mittels der Software durchgeführt wird. Besonders wichtige Designentscheidungen hinsichtlich der Performance und der finalen Ausgabedateien werden detailliert besprochen. Wie der Quellcode der Software heruntergeladen werden kann, ist in Kapitel 8 beschrieben.
Grundlegende linguistische Begriffe wie Lemma oder Token werden erläutert, wenn das Verständnis dieser wichtig ist, um bestimmte Entscheidung im Design der Software nachvollziehen zu können.

In Abbildung 4.1 ist die Struktur von *bundesdata_markup_nlp* zu sehen. Dieses Projekt umfasst alle Python-Skripte, Klassen, und Daten, welche für die automatische Auszeichnung erstellt und genutzt werden.
Mittels der zwei Skripte *bundesdata_markup.py* und *bundesdata_nlp.py* werden die automatische Auszeichnung der Protokolle beziehungsweise die Lemmatisierung, Tokenisierung und Berechnung der N-Gramme durchgeführt. Beide Skripts benötigen die Datei *config.ini*, welche unter anderem reguläre Ausdrücke, Optionseinstellungen und Dateipfade enthält beziehungsweise speichert, die für die Ausführung notwendig sind. In *config_readme.md* ist genauer beschrieben welche Sektinonen welche Art von Daten und Informationen enthalten. Die Module *markup* und *nlp* werden jeweils von dem dazugehörigen Skript genutzt. Das Modul *utility* wird von beiden gleichermaßen verwendet. Mit dem Modul *samples* werden Test- und Entwicklungsdatensets randomisiert erstellt.

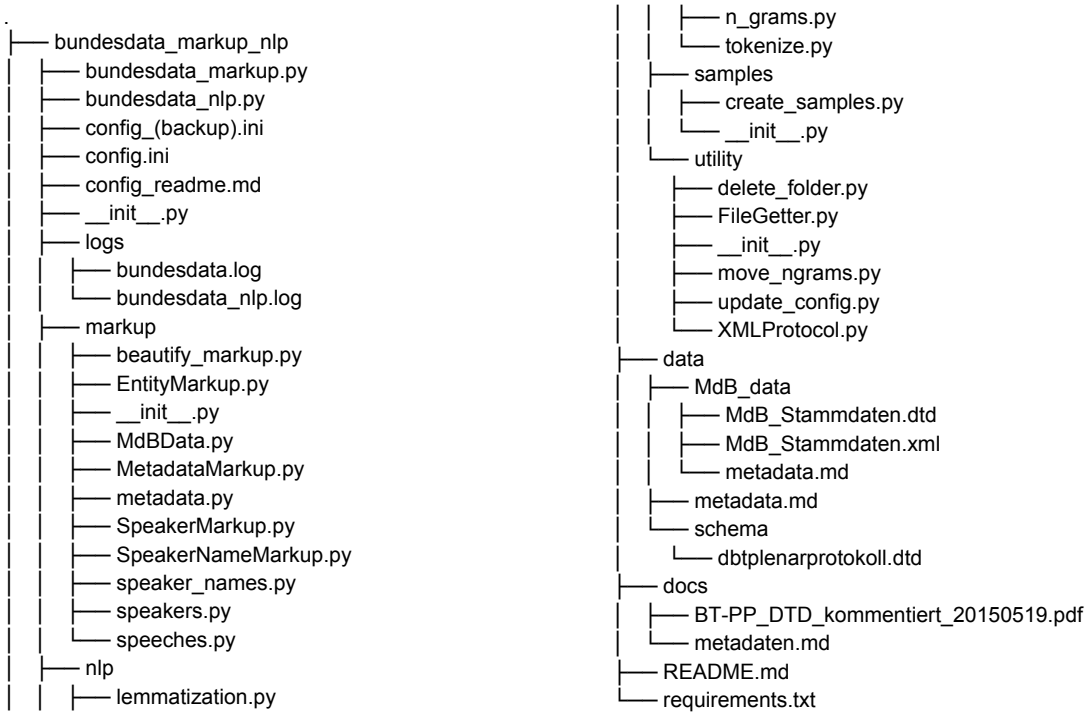

Abbildung 4.1: Strukturübersicht: bundesdata_markup_nlp

Im Verzichnis *data* sind die Datensets, Stammdaten und Schemata zu finden, welche für die Ausführung der Skripts benötigt beziehungsweise als Eingabe dienen. Im Verzeichnis *docs* sind Dokumentationen wie die offizielle Dokumentation der aktuellen Auszeichnungssyntax der 19. Wahlperiode enthalten.
Die Datei *README.md* enthält eine detaillierte Anleitung zur Installation der Software. Mittels *requirements.txt* können die für die Ausführung notwendigen Pakete installiert werden.

4.1 Die Konfigurationsdatei: config.ini

Die Datei enthält unter anderem reguläre Ausdrücke, die im Verlauf der automatischen Auszeichnung genutzt werden, um bestimmte Elemente wie Uhrzeiten oder Redner innerhalb eines Protokolls zu identifizieren. Zusätzlich dazu werden in der Datei alle Datei- und Ordnerpfade gespeichert, welche die Software benötigt und selbst erstellt.

Wenn reguläre Ausdrücke aus der config-Datei mit der Methode *class XMLProtocol.**compile_regex(regex)*** kompiliert werden, werden diese mit der Option **re.MULTILINE** versehen, so dass diese über mehrere Zeilen hinweg funktionieren.
Alle regulären Ausdrücke werden als UTF-8 interpretiert, da dies der Standard in Python 3 ist. [44] Der Ausdruck „\w“ identifiziert somit zum Beispiel auch deutsche Umlaute und französische Buchstaben mit *accent aigu*.

Viele der Sektionen mit regulären Ausdrücken lassen sich mit eigenen Einträge erweitern, so dass die automatische Auszeichnung nachträglich verbessert und detaillierter wird. Auf diese Funktion wird in den nachfolgenden Kapiteln weiter eingegangen.

Neben den regulären Ausdrücken werden noch dazugehörige Optionen und alle notwendigen Ordnerpfade in der Config-Datei gespeichert.

4.2 Softwarevoraussetzungen: Benötigte Pakete und Betriebssystem

Die Software wurde in Python 3.7.1+ [41] entwickelt und verwendet folgende Pakete:

- virtualenv 16.0.0 [4]
- lxml 4.2.5 [2]
- Babel 2.6.0 [45]
- tqdm 4.28.1 [50]
- spaCy 2.0.18 [46]
- scikit-learn 0.20.2 [33]
- js-beautify 1.8.9 (optionales paket) [27]

Damit spaCy genutzt werden kann, wird das statistische Modell *de_core_news_sm-2.0.0* verwendet [7]. Auf der github Seite des Projekts könne neuere Versionen heruntergeladen werden. Diese sind jedoch noch als pre-release gekennzeichnet. [47]

Für die Entwicklung und Nutzung der Software wurde mittels virtualenv [4] eine isolier-

te Python-Umgebung basierend auf Python 3.7.1 [41] erstellt, in der alle benötigten Pakete installiert wurden. Mit dieser Maßnahme sollen Problematiken bezüglich der Versionsabhängigkeiten verhindert werden. [3]
Mit virtualenv werden die Pakete aus der Datei *requirements.txt* in der dort festgelegten Version in die isolierte Umgebung installiert, so dass diese zum Beispiel nicht automatisch aktualisiert werden. So wird sichergestellt, dass die Funktion der Software nicht verändert wird oder diese im schlimmsten Fall nicht mehr funktioniert. Ebenfalls kann die Software so einfach auf verschiedenen System installiert und genutzt werden.

Entwickelt und getestet wurde die Software auf Debian und Arch basierenden Linux-Distributionen. Die Software kann aber auch auf macOS genutzt werden. Eine Nutzung unter Windows ist nicht vorgesehen.

4.3 Entwicklungs- und Testdaten

Insgesamt umfasst der Korpus 4106 XML-Protokolle. Die Software wird in der Lage sein, jede einzelne Datei automatisch auszuzeichnen, so dass die Protokolle um zusätzliche Metadaten in validem XML erweitert wurden.

Da der Korpus relativ groß ist und es zu zeitaufwändig ist, einzelne Entwicklungsschritte und Teilfunktionen an diesem zu testen, wurden aus dem Korpus zwei kleinere Korpora erzeugt (*development_data* und *test_data*). Mit dem Skript *create_samples.py* kann eine beliebige Anzahl an Protokollen zufällig aus dem Hauptkorpus ausgewählt und kopiert werden. Die Größe beider Korpora wurde auf 300 Dateien gesetzt. Die Größe wurde mit Hilfe der klassischen Lehrbuchformel für das Ermitteln von Stichproben aus einer Grundgesamtheit ermittelt [28, S. 3-6].
Die Grundgesamtheit sind die 4106 Dateien. Der Wert der Sicherheit wurde auf 95 Prozent gesetzt. Die Fehlerspanne beträgt somit 5 Prozent. Die Proportion der Grundgesamtheit wurde auf 50 Prozent gesetzt. Mit diesen Werten beträgt die Größe einer Stichprobe 352 Dateien und der Stichprobenfehler liegt bei 5 Prozent. Da mit einer Stichprobengröße von 300 Dateien der Fehler auf nur 5,45 Prozent ansteigt, wurde die Größe für beide Korpora auf 300 Protokolle gesetzt. [28, S. 3-6]

Ziel dieser Vorgehensweise ist es, zwei Korpora zu haben, die signifikant kleiner sind, so dass mit diesen einfacher und schneller gearbeitet werden kann. Da es sich um statistisch relevante Stichproben handelt, sollten theoretisch innerhalb dieser auch alle Fehler, Probleme und Sonderfälle auftreten, die im Gesamtkorpus zu finden sind, so dass diese behoben und Problemlösungen für diese gefunden werden können. Die Software wird unter Verwendung des Korpus *development_data* entwickelt. Nachdem eine Funktion, Klasse oder Modul erfolgreich auf den Korpus angewendet werden kann, wird in einem zweiten Schritt erneut am Korpus *test_data* ermittelt, ob diese Funktion, Klasse oder Modul ebenfalls fehlerfrei auf diese Protokolle angewendet werden kann.

Da sich im Verlauf der Arbeit herausgestellt hat, dass die 15., 16. und 17. Wahlperiode fehlerhaft abgespeichert wurden (siehe Kapitel 2.1.1), sind alle Protokolle dieser Perioden nachträglich aus den Korpora entfernt worden. Die Grundgesamtheit beträgt dann 3433 Protokolle. Der Korpus *development_data* besteht aus 264 Dateien, während der *test_data* Korpus auf 249 Dateien reduziert wurde. Die aussortierten Wahlperioden wurden während der weiteren Entwicklung nicht mehr verwendet und getestet.

Nach Fertigstellung der Software konnte diese erfolgreich auf den gesamten 3433 Dateien umfassenden Korpus angewendet werden. Automatische Auszeichnung, Lemmatisierung, Tokenisierung und die Berechnung der N-Gramme wurden ohne Abbrüche und schwerwiegende Fehler durchgeführt.

Da es aus zeitlichen Gründen nicht mehr möglich war die von der Bundesregierung korrigierten Protokolle automatisch auszuzeichnen und N-Gramme für diese zu berechnen, wurde getestet, ob die Software auch mit den fehlerhaften Protokollen arbeiten kann. Ein Testlauf ergab,dass die Software tatsächlich die fehlerhaften Protokolle auszeichnen und N-Gramme für diese erstellen kann.
Die in Kapitel 6 berechneten Fehlerquoten sind nicht gültig für die fehlerhaften Protokolle, da diese nur für die korrekt abgespeicherten Protokolle ermittelt wurde. Die Auszeichnung und Berechnung der N-Gramme für die fehlerhaften Protokolle wurden im Rahmen der Arbeit nur durchgeführt, um den Nutzern einen besseren Gesamteindruck der Webanwendung vermitteln zu können. Für die Zukunft sollen die neuen korrigierten Protokolle nachträglich mit der Software aufbereitet und in die Webanwen-

dung integriert werden.

4.4 Automatische Auszeichnung mit bundesdata_markup.py

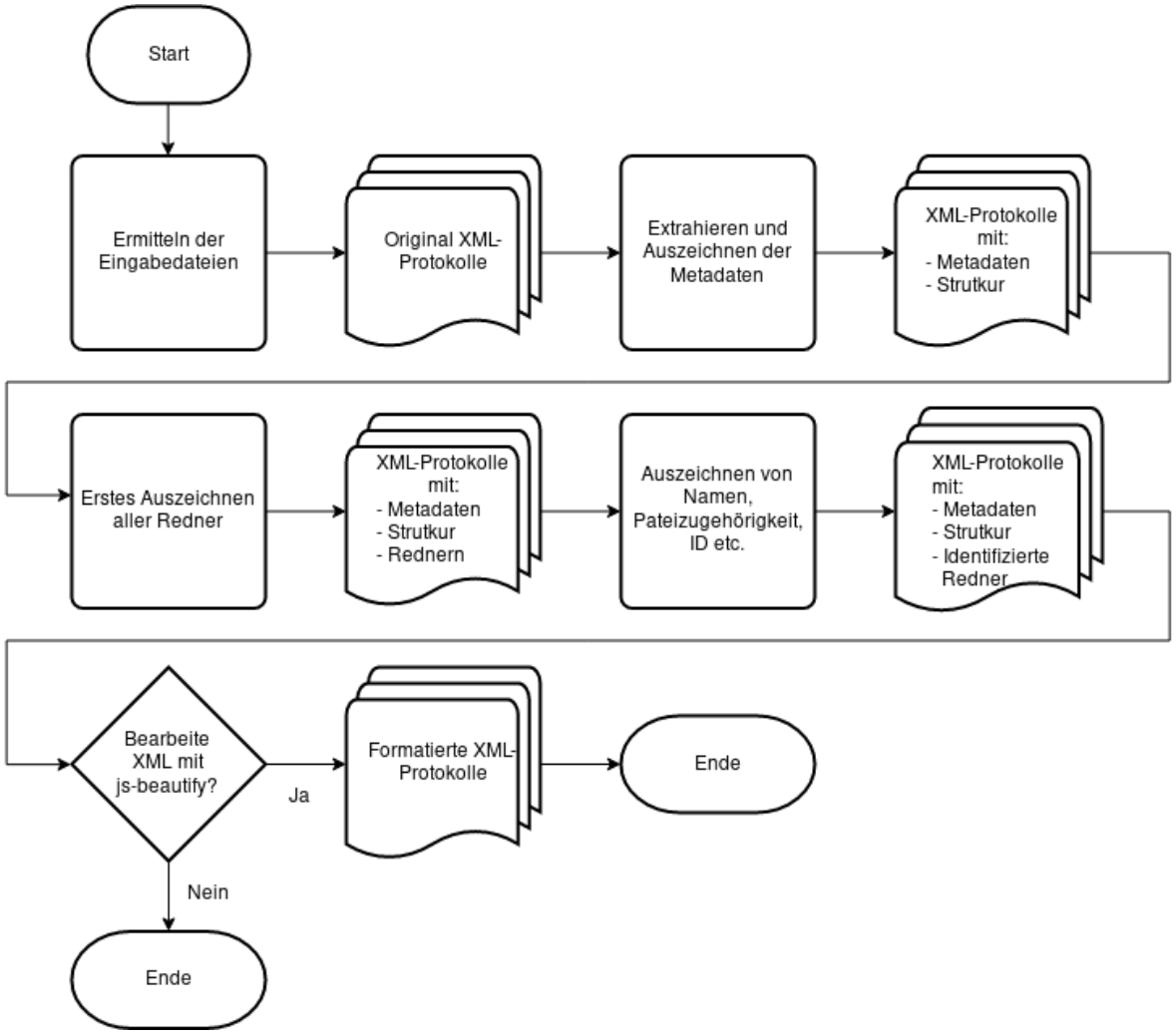

Abbildung 4.2: Programmablaufübersicht der automatischen Auszeichnung

Abbildung 4.2 zeigt den groben Ablauf der automatischen Auszeichnung mehrerer XML-Protokolle. Die automatische Auszeichnung erfolgt in Teilschritten. Jeder Schritt benötigt eine bestimmte ausgezeichnete Art von XML-Protokoll, dass dann bearbeitet und abgespeichert wird, damit dieses neue Format im nächsten Verarbeitungsschritt

als Eingabe genutzt wird. Ein XML-Protokoll durchläuft verschiedene Zustände, die in der Komplexität ihrer Auszeichnung zunehmen. Alle in den Teilschritten erzeugten Protokolle können gespeichert werden, so dass es möglich ist, die automatische Auszeichnung an einem bestimmten Schritt fortzusetzen und Schritte zu überspringen, die bereits durchgeführt wurden. Die Protokolle der verschiedenen Teilschritte werden jeweils in einzelnen Ordnern gespeichert. Ein Protokoll durchläuft somit quasi alle Ordner in folgender Reihenfolge: *Eingabeordner→new_metadata →simple_xml→complex_markup →beautiful_xml*

Von Vorteil ist diese Funktionsweise auch hinsichtlich des Identifizierens von Fehlern und der Beseitigung dieser. Da die Protokolle verschiedene Teilschritte durchlaufen und als Zwischenprodukt dieser abgespeichert werden, können diese Ausgabedateien leicht auf Fehler untersucht werden, die während des dazugehörigen Teilschritts aufgetreten sind. Besonders systematische Fehler, die sich durch den gesamten Prozess der Auszeichnung fortsetzen, können so einfach und schnell an der Stelle ihres Auftretens identifiziert werden.
Ebenfalls wichtig und von Vorteil für die Fehlersuche ist die Log-Funktion, welche in die Software integriert wurde. Bei jedem Durchlauf der Auszeichnung werden Informationen über den Verlauf und den Status dieser in die Datei *docs/bundesdata.log* geschrieben.

Die automatische Auszeichnung wird vom Nutzer mit dem Skript *bundesdata_markup.py* gestartet. Der Datei *README.md* und der internen Dokumentation des Skriptes kann entnommen werden, wie die Software zur Auszeichnung genau verwendet wird. Das Skript *bundesdata_markup.py* ruft weitere Unterfunktionen, Skripts und Klassen auf, die für die automatische Auszeichnung genutzt werden.

Eine Übersicht der verschiedenen Klassen sowie deren Methoden kann der Abbildung 4.3 entnommen werden.

Abbildung 4.3: Klassenhierarchie der Software für die automatische Auszeichnung

4.4.1 Auszeichnen der Metadaten und erste Struktur

Im ersten Teilschritt der automatischen Auszeichnung werden die rudimentär ausgezeichneten offiziellen Protokolle entgegengenommen, die Metadaten aus diesen extrahiert und diese anschließend für die neue Auszeichnung verwendet.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <?xml-stylesheet href="dbtplenarprotokoll.css" type="text/css" charset="UTF-8"?>
3  <!DOCTYPE dbtplenarprotokoll SYSTEM "dbtplenarprotokoll.dtd">
4  <dbtplenarprotokoll herstellung="Satz: Satzweiss.com Print, Web, Software GmbH, Mainzer
   ↪ Straße 116, 66121 Saarbrücken, www.satzweiss.com, Druck: Printsystem GmbH,
   ↪ Schafwäsche 1-3, 71296 Heimsheim, www.printsystem.de" sitzung-datum="24.10.2017"
   ↪ sitzung-ende-uhrzeit="17:03" sitzung-naechste-datum="22.11.2017" sitzung-nr="1"
   ↪ sitzung-start-uhrzeit="11:00" start-seitennr="1" vertrieb="Bundesanzeiger Verlagsgesellschaft
   ↪ mbH, Postfach 1 0 05 34, 50445 Köln, Telefon (02 21) 97 66 83 40, Fax (02 21) 97 66 83 44,
   ↪ www.betrifft-gesetze.de" wahlperiode="19">
5    <vorspann>
6      <kopfdaten>
7        <plenarprotokoll-nummer>Plenarprotokoll
         ↪ <wahlperiode>19</wahlperiode>/<sitzungsnr>1</sitzungsnr></plenarprotokoll-nummer>
8        <herausgeber>Deutscher Bundestag</herausgeber>
9        <berichtart>Stenografischer Bericht</berichtart>
10       <sitzungstitel><sitzungsnr>1</sitzungsnr>. Sitzung</sitzungstitel>
11       <veranstaltungsdaten><ort>Berlin</ort>, <datum date="24.10.2017">Dienstag, den 24.
         ↪ Oktober 2017</datum></veranstaltungsdaten>
12     </kopfdaten>
13     <inhaltsverzeichnis>
```

Quellcode 4.1: Metadaten in der offiziellen Auszeichnung (current_official_protocols_xml/19001.xml)

In Quellcode 4.1 ist die offizielle Auszeichnung der Metadaten zu sehen. Mit dem Skript *metadata.py* wird die vorliegende Auszeichnung (siehe Quellcode 2.1) in die neue Form überführt. Die Struktur der offiziell verwendeten Tags wird hierbei vollständig übernommen. Lediglich einige Attribute aus dem Wurzelelement wie zum Beispiel @herstellung werden nicht verwendet.

Das Skript nutzt die Klasse *MetadataMarkup*, welche eine Unterklasse von *XMLPro-*

tocol ist. Die Klasse *XMLProtocol* bringt grundlegende Methoden mit, die bei der Bearbeitung der Plenarprotokolle benötigt werden. So gibt es beispielsweise die Methode *class XMLProtocol.**read_xml(file_path)*** mit der per Dateipfad identifizierte XML-Protokolle per *lxml etree* eingelesen werden.
Mit der Methode *class XMLProtocol.**save_to_file(...)*** kann das eingelesene und veränderte ElementTree-Modell in eine neue XML-Datei gespeichert werden.

Mit der Methode *class MetadataMarkup.**extract_metadata(etree_element_object)*** werden die gegebenen Metadaten aus dem original Protokoll ausgelesen. Die meisten Metadaten können ohne weitere Bearbeitung verwendet werden. Das extrahierte Datum wird mit weiteren Methoden in ein ISO-Format überführt, um aus diesem zuverlässig einen voll formatierten Datums-String zu erstellen, wie er in der neuen offiziellen Auszeichnung gefordert ist.

Nachdem die gesamten Metadaten extrahiert wurden, werden alle alten Elemente mit der Methode *class MetadataMarkup.**delete_old_metadata(etree_element_object)*** gelöscht. Anschließend werden Inhaltsverzeichnis, Sitzungsverlauf und Anhang voneinander getrennt. Hierfür werden zwei verschiedene reguläre Ausdrücke aus der config-Datei verwendet. Die Ausdrücke werden der Sektion *Regular expressions splits* entnommen und von der Methode *class MetadataMarkup.**split_content(etree_element_object)*** genutzt.

```
5  [Regular expressions splits]
6  session_start_president_split = (\n\w*(?:P|p)räsident\w* [ÜÖÄA-züöäß \-\.,]+ ?:)
7  attachment_split = ((?:\(Schlu(?:(?:ss)|ß)(?: .?der .?Sitzung)?(?:[;:]*)[\w\n,\.
   ↪ ]*\s(?:(?:(?:\d{1,2})\sUhr\s(?:\d{1,2}).?\sMinuten?)|(?:(?:\d{1,2})[\.,
   ↪ ]+(?:\d{1,2})\sUhr)|(?:(?:\d{1,2})[\., ]+(?:\d{1,2})\sUhr\))|(?:(?:\d{1,2})\sUhr))[\.,
   ↪ ]*\))|(?:\(Schlu(?:(?:ss)|ß)(?: .?der .?Sitzung)?(?:[;: ]*)\s?(?:(?:(?:\d{1,2})\sUhr
   ↪ (?:\d{1,2})\sMinuten?)|(?:(?:\d{1,2})[\., ]+(?:\d{1,2})\sUhr)|(?:(?:\d{1,2})[\.,
   ↪ ]+(?:\d{1,2})\sUhr\))|(?:(?:\d{1,2})\sUhr))[\., ]*\))|(?:\(Schlu(?:(?:ss)|ß)(?: .?der
   ↪ .?Sitzung)?(?:[;:]*)[\w\n,\. ]*\s(?:(?:(?:\d{1,2})\sUhr
   ↪ und\s.?(?:\d{1,2}).?\sMinuten?\.\))))|(?:\(Schlu(?:(?:ss)|ß):? (?:\d{1,2}) Uhr (?:\d{1,2})\.\)))
```

Quellcode 4.2: Reguläre Ausdrücke zum Trennen von Anlage, Inhaltsverzeichnis und Sitzungsverlauf (bundesdata_markup_nlp/config.ini)

Mit Hilfe des Ausdrucks *session_start_president_split* wird das erste Auftreten des Präsidenten im Sitzungsverlauf gefunden. Da dieser immer die Sitzung einleitet, kann der gesamte Text vor diesem als Inhaltsverzeichnis ausgezeichnet werden. Der Ausdruck *attachement_split* identifiziert den Punkt im Protokoll, an dem die Sitzung geschlossen wird. Text, der nach diesem kommt, wird als Anlage ausgezeichnet. Der Ausdruck ist sehr komplex, da über die Jahre hinweg verschiedene Arten verwendet wurden, wie der Schluss einer Sitzung aufgeschrieben wurde. So musste zum Beispiel immer beachtet werden, dass sich die Rechtschreibung des Worts Schluss durch eine Rechtschreibreform von „Schluß“ zu „Schluss“ geändert hat.

Nach der Trennung von Inhaltsverzeichnis, Sitzungsverlauf und Anlagen, werden mit der Methode *class MetadataMarkup.**get_session_times()*** die Start- und Endzeit der Sitzung aus dem gesamten Protokoll ausgelesen.

```
1 [Regular expressions time extraction]
2 session_start_time = (?:Die Sitzung wird [umrn]+ (?:(?:(\d{1,2}) Uhr
  ↪ (?:(\d{1,2})?)|(?:(\d{1,2})\.(?:(\d{1,2})) Uhr)) ?(?:Minuten?)?.?)? ?(?:durch\n*[\w \.;'\(\)]*)?[\s \.
  ↪ A-z]*(?:(?:eröffnet\.)|(?:eingeleitet[\w „\",\.]+)))|(?:Begi[\w]+:? (\d{1,2})(?:[, \.]*)?(?:(\d{1,2}))?
  ↪ ?Uhr\.?)|(?:Die Sitzung wird [umrn]+ (\d{1,2}) Uhr eröffnet.)
3 session_end_time = (?:\(Schlu(?:(?:ss)|ß)(?: .?der .?Sitzung)?(?:[;:]*)[\w\n,\.
  ↪ ]*\s(?:(?:(\d{1,2})\sUhr\s(\d{1,2}).?\sMinuten?)|(?:(\d{1,2})[\., ]+(\d{1,2})\sUhr)|(?:(\d{1,2})[\.,
  ↪ ]+(\d{1,2})\sUhr\))|(?:(\d{1,2})\sUhr))[\., ]*\))|(?:\(Schlu(?:(?:ss)|ß)(?: .?der .?Sitzung)?(?:[;:
  ↪ ]*)\s?(?:(?:(\d{1,2})\sUhr (\d{1,2})\sMinuten?)|(?:(\d{1,2})[\., ]+(\d{1,2})\sUhr)|(?:(\d{1,2})[\.,
  ↪ ]+(\d{1,2})\sUhr\))|(?:(\d{1,2})\sUhr))[\., ]*\))|(?:\(Schlu(?:(?:ss)|ß)(?: .?der
  ↪ .?Sitzung)?(?:[;:]*)[\w\n,\. ]*\s(?:(?:(\d{1,2})\sUhr
  ↪ und\s.?(\d{1,2}).?\sMinuten?\.\)))))|(?:\(Schlu(?:(?:ss)|ß):? (\d{1,2}) Uhr (\d{1,2})\.\))
```

Quellcode 4.3: Reguläre Ausdrücke zum extrahieren der Start- und Endzeit (bundesdata_markup_nlp/config.ini)

In Quellcode 4.3 sind zwei weitere Ausdrücke zu sehen, mit denen Start- und Enduhrzeiten der Sitzung im gesamten Protokoll identifiziert werden könne. Der Ausdruck *session_end_time* sieht dem Ausdruck *attachment_split* aus Quellcode 4.2 sehr ähnlich. Allerdings enthalten die Ausdrücke *session_end_time* und *session_start_time* jeweils detaillierte *match groups* mittels derer die Uhrzeit extrahiert und in das Format %H:%M überführt wird.
Sollten im Protokoll zwei Endzeiten vermerkt sein, da diese eventuell vorübergehend

unterbrochen wurde, wird immer nur die letzte Endzeit beachtet. Fehlende Zeiten und Zeiten, die nicht standardisiert aufgeschrieben und somit nicht erfasst werden konnten, werden mit dem Wert „xx:xx“ in die XML-Datei geschrieben.

```
1  <?xml version='1.0' encoding='UTF8'?>
2  <!DOCTYPE dbtplenarprotokoll SYSTEM 'dbtplenarprotokoll_minimal.dtd'>
3  <dbtplenarprotokoll sitzung-datum="05.09.2017" sitzung-start-uhrzeit="09:01"
   ↪ sitzung-ende-uhrzeit="14:21" sitzungs-nr="245" wahlperiode="18">
4    <vorspann>
5      <kopfdaten>
6        <plenarprotokoll-nummer>Plenarprotokoll
         ↪ <wahlperiode>18</wahlperiode>/<sitzungsnr>245</sitzungsnr>
7          (neu)</plenarprotokoll-nummer>
8        <herausgeber>Deutscher Bundestag</herausgeber>
9        <berichtart>Steongrafischer Bericht</berichtart>
10       <sitzungstitel>
11         <sitzungsnr>245</sitzungsnr>. Sitzung
12       </sitzungstitel>
13       <veranstaltungsdaten>
14         <ort>Berlin</ort>, <datum date="05.09.2017">Dienstag, den 5. September
           ↪ 2017</datum>
15       </veranstaltungsdaten>
16     </kopfdaten>
17     <inhaltsverzeichnis>Plenarprotokoll 18/245
```

Quellcode 4.4: Automatisch erzeugte Auszeichnung der Metadaten (outputs/markup/full_periods/18_Wahlperiode_2013-2017/18245.xml)

In Quellcode 4.4 ist die finale automatische Auszeichnung der Metadaten zu sehen. Ebenfalls kann dem Code entnommen werden, dass das Wurzelelement umbenannt und mit weiteren Attributen versehen wurde. Auch das Element Inhaltsverzeichnis ist sichtbar.

4.4.2 Auszeichnen der Redner und Reden

Auf die Auszeichnung der Metadaten folgt die erste Identifizierung und simple Auszeichnung aller Redner und Reden im Sitzungsverlauf mittels des Skriptes *speaker.py*. Hierbei werden die Klassen *EntityMarkup* und deren Unterklasse *SpeakerMarkup* sowie deren Methoden verwendet. Beide Klassen erben grundlegende Funktionen der Klasse *XMLProtocol*.

In Abbildung 4.4 ist der Hauptteil des Skriptes *speakers.py* als Programmablaufplan dargestellt. Nachdem das Programm den Endzustand erreicht hat, werden noch einige XML-Elemente verändert und Attribute eingefügt. Die eigentliche wichtige Auszeichnung der Reden und Redner findet vorher statt und nur diese wird auf Grund dessen in dem Ablaufplan dargestellt.
Die Methoden *class SpeakerMarkup.**identify_speaker()*** und *class SpeakerMarkup.**markup_speaker(case)*** werden detaillierter erläutert.

Zuerst werden die mit den neuen Metadaten ausgezeichneten XML-Potokolle entgegengenommenen und eine Liste all dieser Dateien erstellt. Diese Liste mit Dateipfaden wird pro Eintrag durchgearbeitet. Jedes einzelne Protokoll wird als Instanzvariable an *EntityMarkup* übergeben, wenn diese Klasse instanziiert wird. Ist das Protokolle wohlgeformt, wird in einem nächsten Schritt zuerst mit der Methode *class EntityMarkup.**read_xml(file_path)*** das gesamte XML-Protokoll geparst. Anschließend wird mit der Methode *class EntityMarkup.**get_element_text(regex)*** der gesamte String des Elements <sitzungsverlauf> in eine Instanzvariable eingelesen. Dabei wird dieser auch „escaped“. Das heißt Zeichen, die eine Funktion innerhalb des XML-Syntax haben werden mit einem sogenannten Markierungszeichen versehen, welches verhindert, dass diese Zeichen vom XML-Parser hinsichtlich ihrer Funktion interpretiert werden. Diese ist notwendig, da in den eingelesenen String XML-Elemente als String eingefügt werden.

Im nächsten Schritt wird die Klasse *SpeakerMarkup* instanziiert. Hierbei wird der eingelesene String des Elements <sitzungsverlauf> sowie ein regulärer Ausdruck (Regex) als Instanzvariablen übergeben. Der Regex wird aus der Datei *config.ini* entnommen. Hierfür wird die Sektion „Regular expressions speakers“ als eine Liste von Schlüssel-Wert-Paaren eingelesen. Bei den Schlüsseln handelt es sich um deskriptive Namen

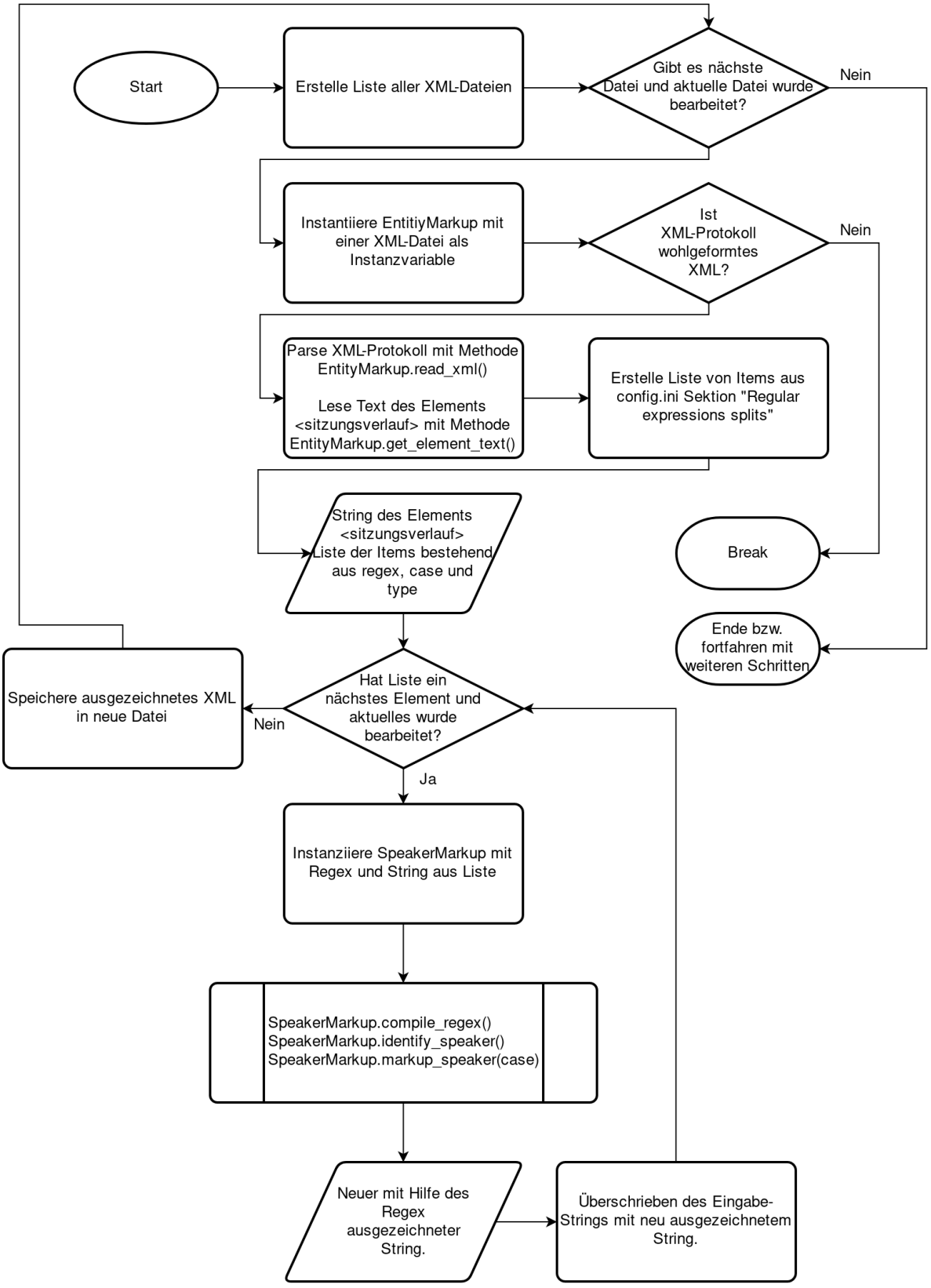

Abbildung 4.4: Teilprogrammablauf des Skriptes *speakers.py*

für den dazugehörigen Wertinhalt.

```
9  [Regular expressions speakers]
10 speaker_president_first = ^\w*(?:P|p)räsident\w* [ÜÖÄA-züööäß \-\.,]+ ?: ; first ; Präsident
11 speaker_state_secretary = ^[ \-\.,\w]+ Staatssekretär[\-\w\n, \n]+: ; middle ; Staatssekretär
```

Quellcode 4.5: Config-Einträge zur Identifizierung von Rednern in den Protokollen (bundesdata_markup_nlp/config.ini)

In Zeile 11 von Quellcode 4.5 ist beispielsweise der Schlüssel *speaker_state_secretary* zu sehen. Der Wertinhalt wird an der Zeichenfolge „ ; “ geteilt, so dass dem Schlüssel quasi eine Liste von Werten zugeordnet wird. Das erste Element der Liste ist der eigentliche Regex, welcher genutzt wird, um alle Redner zu identifizieren, die in der Rolle des Staatssekretärs innerhalb der Sitzung auftreten. An zweiter Stelle wird der „case“ aufgeführt. Insgesamt gibt es drei verschiedene Fälle, die als Wert angegebene werden können:

1. **first**: Kennzeichnet, dass der Redner, welcher mit dem dazugehörigen Regex identifiziert wird, den ersten Redebeitrag der Sitzung hält.
2. **middle**: Kennzeichnet, dass der Redner, welcher mit dem dazugehörigen Regex identifiziert wird, einen Redebeitrag hält, der nicht die erste Rede der Sitzung ist.
3. **last**: Kennzeichnet, dass die Zeichenkette, welche mit dem dazugehörigen Regex identifiziert wird, das Ende der Sitzung markiert.

Diese Kennzeichnung ist notwendig, da die XML-Tags „manuell“ gesetzt werden. Dies wird im Laufe des Kapitels genauer erläutert.
An dritter Stelle steht eine Typbezeichnung beziehungsweise Rollenbezeichnung, die als Attribut für das Element <redner> verwendet wird.

Vorteil dieser Config-Sektion ist, dass diese einfach erweitert werden kann, falls eine bestimmte Art von Redner übersehene wurde. So ist im Verlauf der Softwareentwicklung der Eintrag *speaker_state_secretary* erst sehr spät hinzugefügt worden. So können beispielsweise Dritte Personen und Nutzer und Nutzerinnen weitere Einträge für die automatische Erkennung von Redner hinzufügen und gegebenenfalls Rednertypen nachträglich erkennen lassen, die sehr selten auftreten und während der Entwick-

lung der Software übersehen wurden. Bei der Erweiterung muss lediglich beachtet werden, dass es nur einen Eintrag mit dem case „first“ und einen mit dem case „last“, welche jeweils am Anfang beziehungsweise am Ende der Sektion stehen, geben darf.

Nachdem die Klasse *SpeakerMarkup* instanziiert wurde, wird der dazugehörige Regex mit der Methode *class SpeakerMarkup.**compile_regex()*** kompiliert. Mit den Methoden *class SpeakerMarkup.**identify_speaker()*** und *class SpeakerMarkup.**markup_speaker (case)*** wird die eigentliche Auszeichnung durchgeführt.

Die Methode *class SpeakerMarkup.**identify_speaker()*** identifiziert mittels des regulären Ausdrucks alle Teilzeichenketten, die diesem entsprechen und fügt diese als „Matchobjekte“(<re.Match object>) in eine Liste ein. Matchobjekte enthalten neben dem Match an sich auch die Start- und Endposition, welche für die spätere Auszeichnung benötigt werden. [43] Die Liste wird in einer Instanzvariable gespeichert, damit dieses später von der Methode *class SpeakerMarkup.**markup_speaker(case)*** genutzt werden kann.

Grundlegende Idee der ersten Auszeichnung der Redner und Reden ist es, dass Auftreten von Rednern als eine Art Grenze zu sehen. Text, der nach dem ausgezeichneten Redner auftritt, ist diesem als seine Rede zu zuordnen, bis der nächste Redner auftritt und der Redebeitrag beendet wird.

```
54        if(case == "first"):
55            # Uses re.sub because it is only for one match.
56            start_tags = "<rede><redner>"
57            end_tags = "</redner>"
58            self.matches_count = 1  # sets count to 1 because it only marks the first match
59            markup_logging()
60            first_match = self.matches[0]
61            start_xml = start_tags + first_match.group() + end_tags
62            if(len(first_match.group().split()) <= 10):
63                self.string_to_search = self.regex_compiled.sub(start_xml,
64                                                self.string_to_search,
65                                                count=1)
66            self.markuped_string = self.string_to_search
```

Quellcode 4.6: Auszug aus der Methode *class SpeakerMarkup.**markup_speaker()***: case = „first“ (bundesdata_markup_nlp/markup/SpeakerMarkup.py)

In Quellcode 4.6 ist zu sehen, wie die erste Grenze gesetzt wird. Mittels der if-Abfrage in Zeile 54 wird sichergestellt, dass das in der Funktion verwendete Matchobjekt den ersten Redner und dessen Rede der Sitzung kennzeichnet. Da es sich um die erste Grenze beziehungsweise das erste neue XML-Element handelt, werden in Zeile 56 die Tags <rede> und <redner> geöffnet, aber in Zeile 57 nur das Tag </redner> geschlossen.
Anschließend wird in Zeile 61 der neue XML-String aus den Tags und der Matchgruppe des Matchobjekts, welches an der ersten Position der verwenden Liste steht, erstellt. Das Element <rede> wird erst geschlossen, wenn der nächste Redner ausgezeichnet wird.
Der neue String wird in Zeile 62 per *re.sub()* an die Position geschrieben, an der zum ersten mal der Regex auftritt. Da in der Config-Datei nur ein Eintrag existiert, der den Wert „first“ als „case“ enthält, wird dieser Teil der Funktion nur einmal durchgeführt.

Auf die Auszeichnung des ersten Redners und dessen Rede folgt die Auszeichnung aller nachfolgenden Redner und deren Reden. Hierbei werden alle Config-Einträge verwendet, die den Wert „middle“ beinhalten.

```
68      elif(case == "middle"):
69          """
70          Does not use re.sub because it is faster to work on the string.
71          Also it avoids looping two times to get the specific match.group()
72          which caused some errors.
73          """
74          index_shift = 0
75          start_tags = "\n</rede><rede><redner>"
76          end_tags = "</redner>"
77          markup_logging()
78          for match in self.matches:
79              index_start = match.start() + index_shift
80              index_end = match.end() + index_shift
81              whole_match_len = len(match.group())
82              # Handels cases where lots of text before the actual speaker is # matched
83              linebrks_in_match = len(match.group().split("\n"))
84              if(linebrks_in_match >= 2):
85                  last_part_match = "".join(match.group().split("\n")[1:])
```

```
86              first_line_of_match = match.group().split("\n")[0]
87              if(len(first_line_of_match.split()) <= 10):
88                  match = first_line_of_match + last_part_match
89              else:
90                  match = last_part_match
91
92              delta_start_index = whole_match_len - len(match)
93              index_start = index_start + delta_start_index
94
95              self.string_to_search = (self.string_to_search[:index_start]
96                                       + start_tags
97                                       + match
98                                       + end_tags
99                                       + self.string_to_search[index_end:]
100                                      )
101             index_shift += len(start_tags) + len(end_tags)
102
103         else:
104             self.string_to_search = (self.string_to_search[:index_start]
105                                      + start_tags
106                                      + match.group()
107                                      + end_tags
108                                      + self.string_to_search[index_end:]
109                                      )
110             index_shift += len(start_tags) + len(end_tags)
111
112     self.markuped_string = self.string_to_search
113
```

Quellcode 4.7: Auszug aus der Methode *class SpeakerMarkup.**markup_speaker()***: case = „middle“ (bundesdata_markup_nlp/markup/SpeakerMarkup.py)

In Zeile 68 von Quellcode 4.7 wird sichergestellt, dass nur Redner gekennzeichnet werden, die die Sitzung nicht einleiten. In den Zeilen 75 und 76 werden die Start- und End-Tags definiert, mit denen der per Regex identifizierte Redner umgeben wird. Der Variable *start_tags* wird der Wert „\n</rede><rede><redner>“ zugeordnet. An erster Stelle befindet sich ein schließendes <rede>-Tag, da noch das vorangegangene Tag geschlossen werden muss, welches entweder von einem vorangegangenem Durchlauf mit dem case-Wert „middle“ oder „first“ gesetzt wurde. Der Variable *end_tag* wird

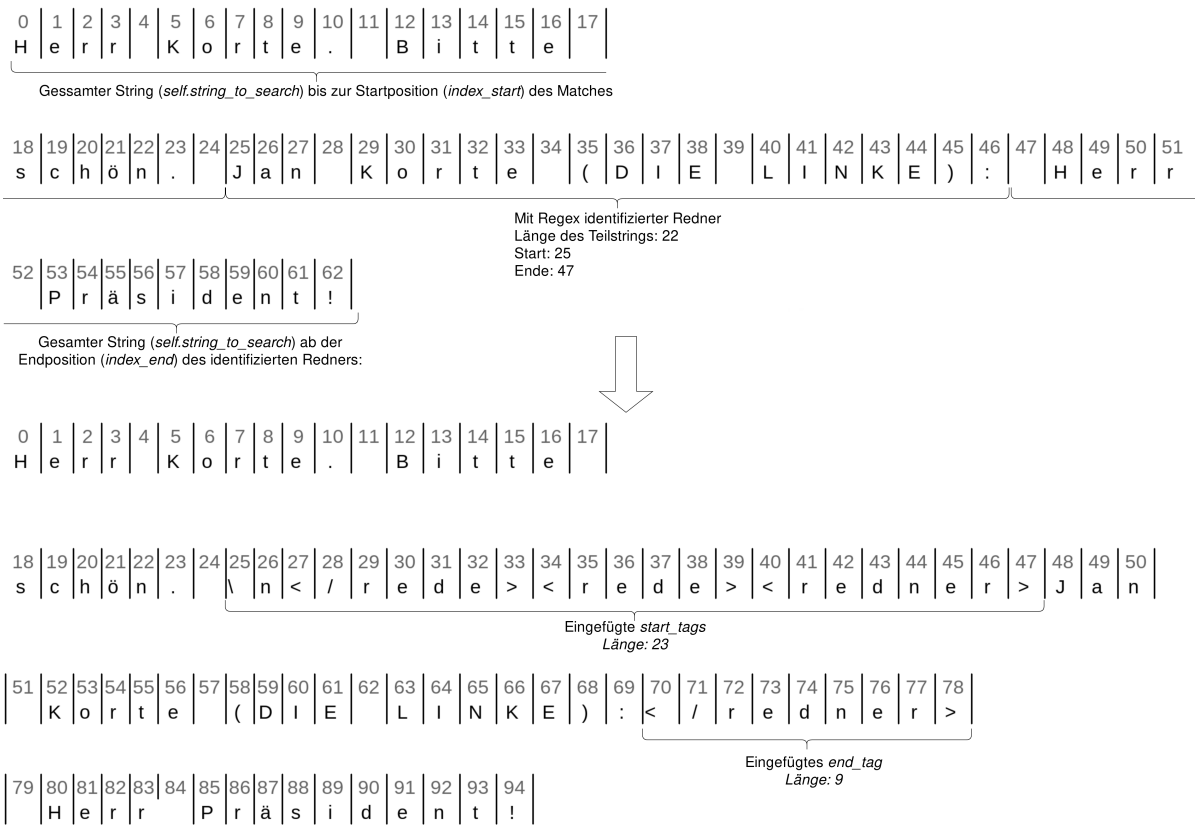

Abbildung 4.5: Schaubild der Auszeichnung eines Redners mittels *Slice-Notation*

der Wert „</redner>“ zugewiesen.
In der ab Zeile 78 startenden for-Schleife wird über die Liste aller Matchobjekte iteriert, die mittels des aktuellen regulären Ausdrucks identifiziert wurden. Zeile 79 und 80 legen fest an welcher Position jeder Match im String beginnt und endet. Hierbei wird noch die Variable *index_shift* addiert, da sich die Position eines jeden Matches mit dem Ersetzen des vorangegangene Matches verändert.
Die Auszeichnung der identifizierten Redner wird grundlegend durch eine Manipulation und direkte Veränderung des Strings *self.string_to_search*, welcher bei der Instanziierung der Klasse gesetzt wurde, realisiert. In den Zeilen 104 bis 109 wird mittels *Slice-Notation* jeder Match durch sich sowie die Start- und End-Tags ersetzt. Dafür wird der gesamte String von Position 0 bis zur Anfangsposition (*index_start*) des aktuellen Matches ausgewählt und an diesen die Variablen *start_tags*, *match* und der gesamte String ab der Endposition des aktuellen Matches bis zum Ende des gesamten Strings angehängt. Abbildung 4.5 zeigt diese Funktionswiese schematisch.

Nachdem der indentifizierte Redner durch das ausgezeichnete XML-ersetzt wurde, wird die Variable *index_shift* um die Gesamtlänge von *start_tags* und *end_tags* erhöht, damit im nächsten Schleifendurchlauf die Start- und Endpositionen des Matchobjekts korrekt verschoben sind.

In den Zeilen 83 bis 101 von Quellcode 4.7 wird das selbe Verfahren für Matches angewendet, die über mehr als eine Zeile hinweg identifiziert wurden. Hierfür wird der Match an den Zeilenumbrüchen geteilt und überprüft, ob der erste Zeile weniger als zehn Worte enthält. Ist dies der Fall werden die Zeilen wieder zusammengefügt und als korrekter Match eines Redners betrachtet. Ist die erste Zeile länger als zehn Worte, wird nur die letzte Zeile als Match eines Redners verwendet. Die Variable *index_start* wird dementsprechend in Zeile 93 angepasst.

Mit dieser Überprüfung wird verhindert, dass ein Match ausgezeichnet wird, welcher zwar einen Redner beinhaltet, aber vor diesem eigentlich gesuchten Teilstring weitere Wörter erfasst wurden, die nicht gesucht sind. Diese Problematik kann auftreten, da es Tippfehler und Abweichung innerhalb der Protokolle bezüglich der einheitlichen Schreibweise von Rednern gibt. Bei der Länge von zehn Wörtern handelt es sich um einen selbst festgesetzten Wert, der durch iterative Kontrolle der Funktion ermittelt wurde. Eine ähnliche Überprüfung findet auch in Quellcode 4.6 Zeile 62 statt. Dort ist jedoch nie der Fall eingetreten, dass eine Matchgruppe aus mehr als zehn Wörtern besteht.

Zuletzt wird das Ende der Sitzung ausgezeichnet. Hierfür wird der Config-Eintrag verwendet, der den *case* mit dem Wert „last“ aufweist. Hierbei können drei Fälle eintreten.
Im ersten Fall gibt es genau einen Match, so dass dieser mit den Tags </rede> gefolgt von <sitzungsende/> umgeben wird.
Im zweiten Fall gibt es keinen Match und das Element </rede> wird an das Ende des Strings *self.string_to_search* angehängt. So wird sichergestellt, dass das XML auch in diesem Fall wohlgeformt ist. Nachteil ist, dass die letzte Rede in diesem Fall Text enthält, der nicht zu dieser gehört.
Im dritten Fall gibt es mehr als einen Match, der das Ende einer Sitzung markiert. Hier wird immer nur der letzte Match verwendet, um das Ende der Sitzung auszuzeichnen.

Nachdem der String, mit dem die Klasse instanziiert wurde, mit der neuen Auszeich-

nung versehen wurde, wird der Eingabe-String mit dem veränderten String überschrieben. Im nächsten Schleifendurchlauf wird dann der bereits veränderte String verwendet, wenn die Klasse *SpeakerMarkup* instanziiert wird. Diese Schleife wird solange durchlaufen, wie es Einträge beziehungsweise reguläre Ausdrücke in der aus der Sektion „Regular expressions speakers“ erzeugten Liste gibt. Der String wird also pro Schleifendurchlauf um die Auszeichnung für eine weitere Rednerrolle erweitert. Ist die Schleife am Ende angelangt, wird das neu ausgezeichnete XML in eine neue Datei abgespeichert und mit der nächsten Protokolldatei fortgefahren, bis diese alle abgearbeitet wurden.

Im letzten Teil des Skriptes *speakers.py* werden die ausgezeichneten Redner um das Attribut @typ erweitert, welches aus der Config-Datei entnommen wird. Mit dem dazugehörigen Regex werden die jeweiligen redner identifiziert, und mit dem passenden Attribut versehen. Zusätzlich werden noch die Attribute @sitzung-start-uhrzeit und @sitzung-ende-uhrzeit gesetzt.

4.4.2.1 Unterschiede zur offiziellen Auszeichnungssyntax

Im Vergleich zur offiziellen neuen Auszeichnung der Bundesregierung wird in diesem Schritt des Auszeichnungsverfahrens bewusst von der offiziellen Struktur abgewichen.

```
790      <p klasse="J">Jetzt frage ich die Frau Kollegin Roth, ob sie die Wahl annimmt.<a id="S18"
         ↪ name="S18" typ="druckseitennummer"/></p>
791    </rede>
792    <rede id="ID19101800">
793      <p klasse="redner"><redner
         ↪ id="11003212"><name><vorname>Claudia</vorname><nachname>Roth</nachname>
794      <fraktion>BÜNDNIS 90/DIE
         ↪ GRÜNEN</fraktion><ortszusatz>(Augsburg)</ortszusatz></name></redner>Claudia
         ↪ Roth (Augsburg) (BÜNDNIS 90/DIE GRÜNEN):</p>
795      <p klasse="J_1">Ja, ich nehme die Wahl mit großer Freude an und freue mich auch auf die
         ↪ Zusammenarbeit.</p>
796      <kommentar>(Beifall beim BÜNDNIS 90/DIE GRÜNEN sowie bei Abgeordneten der
         ↪ CDU/CSU)</kommentar>
797      <name>Präsident Dr. Wolfgang Schäuble:</name>
```

```
798     <p klasse="J_1">Die Freude ist unübersehbar. Ich beglückwünsche Sie im Namen des ganzen
        ↪ Hauses, Frau Kollegin Roth.</p>
```

Quellcode 4.8: Redebeitrag des Präsidenten als Teil einer fremden Rede (current_official_protocols_xml/19001.xml)

In Quellcode 4.8 ist zu sehen, wie in Zeile 792 eine neues Redeelement geöffnet wird. Die Rednerin ist hier Claudia Roth. In Zeile 796 wird jedoch mit dem Element <name> der Präsident der aktuellen Sitzung als Redner gekennzeichnet und der gesamte nachfolgende Text ist diesem zuzuordnen. Hier werden somit Redebeiträge verschiedener Personen vermischt. Ziel der vorliegenden Arbeit und der Software ist es jedoch Redner und die dazugehörigen Reden und Redebeiträge eindeutig zuzuordnen. Aus diesem Grund wird jeder Redebeitrag des Präsidenten und auch Zwischenfragen anderer Abgeordneter (nicht zu verwechseln mit Kommentaren und Zurufen), die nach dem gleichen Schema wie in Quellcode 4.8 gekennzeichnet sind, als einzelne Elemente mit dem Tag <rede> versehen.

Das pro Redner gesetzte Attribut @typ ist ebenfalls eine Abweichung von der offiziellen Auszeichnung, welche zusätzliche Informationen in der Protokolldatei hinterlegt.

4.4.3 Detailtiere Auszeichnung der Redner mit Hilfe der offiziellen Stammdaten

In Kapitel 4.4.2 wurde erläutert wie die verschiedenen Redner und ihre Redebeiträge ausgezeichnet wurden, so dass die eindeutige Zuordnung von Redner und Rede möglich ist.
Dieses Kapitel erläutert, wie die <redner>-Elemente mittels weiterer Kinderelemente feiner strukturiert und ausgezeichnet werden. Hierfür wird der Text des Elements <redner> mit Hilfe der Daten aus der Datei *MdB_Stammdaten.xml* (siehe Quellcode 2.4) analysiert und ausgezeichnet. Zu diesem Zweck wird die Klasse *SpeakerNameMarkup* eingeführt, deren Methoden in dem Skript *speaker_names.py* verwendet werden.

```
13891 </rede><rede typ="MdB"><redner>Dr. André Hahn (DIE LINKE):</redner>
13892 Frau Präsidentin! Meine Damen und Herren! Der Titel
```

Quellcode 4.9: Beispiel eines Redner Strings (/dev_data/simple_xml/18243.xml)

In Quellcode 4.9 sind ein ausgezeichneter Redner sowie der Anfang seiner zugeordneten Rede zu sehen. Dies ist das Ergebnis der Auszeichnung mittels des Skripts *speakers.py*, dessen Funktionsweise in Kapitel 4.4.2 beschrieben wurde. Ziel ist es die vorliegende Auszeichnung so zu erweitern, dass diese die folgenden Elemente, Kinderelemente und Attribute mit den entsprechenden Daten des Redners enthält:

- <name>
  - <titel>
  - <vorname>
  - <nachname>
  - <fraktion>
- <rolle>
  - <rolle_lang>
  - <rolle_kurz>
- @id für <rede> und <redner>

Diese Elemente und deren Struktur sind in der offiziellen Strukturdefinition beschrieben. [5, S. 8] Auch in diesem Schritt der Auszeichnung soll sich so nah wie möglich an die offizielle Struktur angenähert werden.

```
13891 </rede><rede typ="MdB" id="ID1824311600"><redner
      ↪ id="11004288"><name><titel>Dr.</titel><vorname>André</vorname>
13892 <nachname>Hahn</nachname>
13893 <damalige_fraktion/><fraktion>DIE LINKE.</fraktion><partei>DIE
      ↪ LINKE.</partei><original_string>Dr. André Hahn (DIE LINKE):</original_string></name>Dr.
      ↪ André Hahn (DIE LINKE):</redner>
13894 Frau Präsidentin! Meine Damen und Herren! Der Titel
13895
```

Quellcode 4.10: Beispiel eines detailliert ausgezeichneten Redners (/dev_data/complex_markup/18243.xml)

In Quellcode 4.10 ist der Output des aktuellen Auszeichnungsschrittes *speaker_names.py* zu sehen. Der Redner wurde mit den gewünschten Elementen ausgezeichnet und gegebenenfalls um zusätzlich vorliegende Informationen erweitert. Ebenfalls wurden dem Redner sowie dessen Rede jeweils eine ID zugeordnet. Die ID für den Redner wird aus den offiziellen Stammdaten entnommen, während die ID für die Rede automatisch erstellt wird.

Grundlegende Idee, die hinter der Funktionsweise des aktuellen Auszeichnungsschrittes steckt, ist, dass jeder Text eines jeden Elements <redner> gegen die Stammdaten verglichen wird, um die benötigten Informationen zu erhalten.
In einem ersten Schritt wird hierfür die Datei *MdB_Stammdaten.xml* (siehe Quellcode 2.4) eingelesen, um Zugriff auf alle darin enthaltenen Daten zu haben. So können in einem nächsten Schritt die Vornamen (<VORNAME>) aller MdBs ausgelesen und in ein Set überführt werden, so dass eine Menge von Vornamen entsteht. Diese Menge beschreibt somit alle seit 1949 validen Vornamen, die ein MdB haben darf beziehungsweise kann. Dieser Schritt wird ebenfalls für das Element <NACHNAME> durchgeführt, so dass für diese Elemente eine Menge an möglicher valider Werte besteht: So gibt es jeweils eine Menge aller möglicher valider Vor- und Nachnamen, die ein MdB haben darf.

Diese Sets werden mittels des Skripts *speaker_names.py* in dem *dictionary feature_set_dict* gespeichert. Neben den beiden genannten Sets werden unter anderem noch die Sets für alle validen Parteien (<PARTEI_KURZ>) und Ortsnamen (<ORTSZUSAT>) erstellt und in das *dictionary* gespeichert.

Für die automatische Auszeichnung des Redner-Strings wird dieser nun in seine einzelnen Token aufgeteilt. [23] Der String „Dr. André Hahn (DIE LINKE):“ wird mittels Tokenisierung in eine Liste seiner Tokens zerlegt, bei der Satzzeichen ignoriert werden. Da es sich um relativ kurze und wenig komplexe Strings handelt, werden diese lediglich an der Stelle des Leerzeichens aufgeteilt. Vor der Zerteilung werden noch Kommata und der Doppelpunkt am Ende entfernt. Punkte werden nicht entfernt, da diese Teil des Akademischen Titels sind. Die Liste der Tokens sieht für das konkrete

Beispiel so aus: [’Dr.’, ’André’, ’Hahn’, ’(DIE’, ’LINKE)’]

Die nachfolgende Überprüfung und Auszeichnung wird mittels der Methode *class SpeakerNameMarkup.**cross_reference_markup(...)*** durchgeführt, welche weiter Hilfsfunktionen definiert.

Mit Hilfe der genannten Methode werden nun alle Tokens einzeln ausgewählt und überprüft, ob diese Teil der Menge der möglichen validen Vor- oder Nachnamen sind. Ist ein Token Teil der Menge der mögliche Vornamen, wird diese Information in einem für den aktuellen Redner erstellten Python *dictionary* abgespeichert. Die Struktur des *dictionaries* sieht wie folgt aus:

```
1   speaker: {
2   'vorname': 'None',
3   'nachname': 'None',
4   'titel': 'None',
5   'fraktion': None,
6   'namenszusatz': None,
7   'ortszusatz': None,
8   'partei': None,
9   'wahlperiode': 'None',
10  'feature_complete': False,
11  'id': None,
12  'rolle_lang': None,
13  'rolle_kurz': None,
14  'original_string': 'None',
15  'identified': False,
16  'damalige_fraktion': None,
17  'präsident': None
18  }
```

Quellcode 4.11: Struktur des *dictionaries* zur Identifikation eines Redners

Fast alle Schlüssel des *dictionaries* sind die Element- und Attributnamen der offiziellen neuen Auszeichnungssyntax. Die Schlüssel „feature_complete“, „original_string“, „identified“, „damalige_fraktion“, „partei“ und „präsident“ sind selbst eingeführte Schlüsselnamen, denen zusätzliche Informationen zugeordnet werden. Die Werte der Schlüssel werden initial auf „None“ oder „False“ gesetzt und nach und nach mit der passen-

den Information aufgefüllt.

```
284     # Start of main function cross_reference_markup
285     ###
286
287         # Initiates empty dict and gets keys for it
288         redner_dict = dict()
289         features = list(feature_set_dict.keys())
290
291         # Counters to calculate how successful the identification of speakers is
292         identified_speakers = 0
293         unidentified_speakers = 0
294         multiple_identified_speakers = 0
295
296         # Cross references every <redner> string
297         for string in tqdm(strings, desc="Cross reference name markup for speakers in strings"):
298             self.logger.info("\nStarting name markup process for new speaker:")
299             # Sets values in redner_dict to None or specific value
300             initiate_dict(features, [feature for feature in features])
301             tokens = string.replace(":", "").replace(",", "").split()  # replaces ":" and "," with nothing
                ↪  because some names would be "name:" and some names would contain a ","
302             for token in tokens:
303                 get_names(features, feature_set_dict, token)
304             self.logger.info("nachname is: " + str(redner_dict["nachname"]))
305             feature_keys = [key for key in features if key not in ["vorname",
306                                                                    "nachname"]]
307             for f_key in feature_keys:
308                 get_feature(f_key, string, feature_set_dict[f_key][0])
309             get_party(redner_dict)
310             check_name(redner_dict)
311             regex_p = r"^\w*(?:P|p)räsident\w*"
312             get_match_in_str("präsident", string, regex_p)
313             get_role(string)
314
315         ###
316         # Checks if script is still running for the same current periode.
317         # If this is not the case the known_redner_dicts will be emptied.
318         ###
319             current_wahlperiode = get_periode(MdB_etree)
320             if(current_wahlperiode != SpeakerNameMarkup.last_wahlperiode):
321                 SpeakerNameMarkup.known_redner_dicts = dict()
322             SpeakerNameMarkup.last_wahlperiode = current_wahlperiode
```

Quellcode 4.12: Teilauszug aus der Methode *class SpeakerNameMarkup.**cross_reference_markup(...)*** (markup/SpeakerNameMarkup.py)

In Quellcode 4.12 Zeile 288 ist zu sehen, wie das *dictionary* initialisiert wird. Die nächste Zeile zeigt wie die Schlüssel für das *dictionary* als eine Liste erstellt werden. Hierfür wird das *dictionary feature_set_dict* verwendet, welches im Skript *spekaer_names.py* erstellt und als Parameter an die Methode *class SpeakerNameMarkup.**cross_reference_markup(...)*** übergeben wurde. In Zeile 300 wird mittels *initiate_dict(...)* das *dictionary redner_dict* mit den Schlüsseln aufgefüllt, so das die Struktur aus Quellcode 4.11 entsteht.

Der Abgleich, ob der aktuelle Token aus der Liste ein valider Vor- oder Nachname ist, wird mittels der Funktion *get_names* durchgeführt, die in Zeile 303 innerhalb einer for-Schleife aufgerufen wird. Die Funktion führt ebenfalls eine Überprüfungen durch, die verhindert, dass ein Nachname fälschlicherweise als Vorname erkannt wird. Diese Fall tritt relativ häufig ein, da die meisten Redner-Strings nur einen Nachnamen enthalten. Ein Nachname kann jedoch auch als ein valider Vorname erkannt werden, da jeder Token gegen beide Mengen getestet wird. Sind Vor- und Nachname identisch, wird der Wert des Vornamens im dictionary auf „None“ gesetzt und nur der Nachname gespeichert. Der Vorname wird im weiteren Verlauf automatisch ermittelt. Der Fall, dass für einen Redner nur der Vorname identifiziert wird, kann nicht eintreten, da mindestens immer der Nachname im Redner-String aufgeführt wird. Der einzige Fall, bei dem kein Nachname erkannt wird, ist, wenn der Redner nicht in der Stammdatenbank aufgeführt oder auf Grund von Tippfehlern nicht gefunden werden kann.
Nach diesem Schritt sind folgenden Informationen im *dictionary* gespiechert:

```
1 speaker: {  'vorname': 'André',  'nachname': 'Hahn', ...
```

Grundlegend wird in diesem Schritt der Auszeichnung somit nicht direkt überprüft, ob der String ein MdB beschreibt, welches in der Stammdatenbank aufgeführt wird, sondern ob die Tokens des Strings überhaupt valide Teilmengen der möglichen Namen

und Nachnamen sind. Diese Vorgehensweise ist schneller, als zu überprüfen, ob ein Redner-String einer Person in der Stammdaten-XML zugeordnet werden kann.

Nachdem Vor- und Nachname ermittelt wurden, werden im nächsten Schritt mittels der Funktion *get_feature* die Fraktion, der Ortszusatz, Titel und Namenszusatz des Redners ermittelt. Da die Sets dieser Merkmale eher klein sind, werden diese auf die umgekehrte Weise verwendet, um die Auszeichnung durchzuführen. Hierbei wird zum Beispiel jedes Item des Sets, welches alle validen möglichen Fraktionen enthält, als Teil eines Regex verwendet, mittels dem überprüft wird, ob diese Fraktion im Redner-String genannt wird. Ist dies der Fall wird diese Fraktion als Wert im *dictionary* eingetragen. Selbiges geschieht für die anderen Merkmale, so dass das *dictionary* nach Vollendung der Funktion wie folgt aussieht:

```
1 speaker: {  'vorname': 'André',  'nachname': 'Hahn',  'titel': 'Dr. ',  'fraktion': 'DIE LINKE'.,
  ↪ 'namenszusatz': None,  'ortszusatz': None,  'partei': 'None',...
```

Diese umgekehrte Vorgehensweise wird hier angewendet, da zum Beispiel Titel wie „Dr. Dr. h. c.“ oder Fraktionen wie „DIE LINKE“ bei der Tokenisierung zerteilt werden. Bei dem vorliegenden Beispiel wird kein Namens- oder Ortszusatz genannt.

Mittels der Funktion *get_party()* in Zeile 309 wird zusätzlich die Unterscheidung zwischen Fraktion und Partei vorgenommen, welche in der offiziellen Auszeichnung nicht existiert. Diese ist jedoch eine wichtige Information, da MdBs nicht nur einer Fraktion sondern auch einer Partei angehören. Besonders wichtig ist dies bei allen MdBs, die der Fraktion CDU/CSU angehören. Hier wird in der offiziellen Auszeichnung nicht aufgeschlüsselt, bei welcher der beiden Parteien der jeweilige Redner Mitglied ist. In der automatisch erstellten Auszeichnung wird diese Unterscheidung zusätzlich vorgenommen und neben der Fraktion auch die Partei ermittelt.

Ebenfalls ermittelt werden die Merkmale „rolle_lang“, „rolle_kurz“ und Wahlperiode. Während die ersten beiden Merkmale mittels regulärer Ausdrücke aus dem String extrahiert werden, wird die Wahlperiode aus den bereits vorhandenen Metadaten ausgelesen. All diese Informationen werden ebenfalls in dem *dictionary* des Redners ge-

speichert.

Nachdem versucht wurde aus dem Redner-String so viele Merkmale und Informationen wie möglich zu extrahieren, werden diese im zweiten Teil der Methode *class SpeakerNameMarkup.**cross_reference_markup(...)*** genutzt, um weitere fehlende Merkmale eines Redners zu identifizieren, die nicht auf Grundlage der Informationen im Redner-String, ermittelt werden konnten. Hierfür wird die Hilfsmethode *add_missing_MdB_feature()* verwendet.
Diese Hilfsfunktion benötigt als Input eine Liste von Merkmalen des aktuellen Redners, die genutzt werden soll, um für diesen die weiteren fehlenden Merkmale zu identifizieren. Zum Beispiel kann der Funktion die Liste *['vorname', 'nachname']* übergeben werden, mit der das Merkmal „ortszusatz“ nachträglich für diesen identifiziert werden soll. Aus den beiden Merkmalen der Liste wird automatisch ein XPath erstellt. Hierfür werden die beiden Merkmalschlüssel in die entsprechenden XML-Elementnamen übersetzt, die in der Stammdatenbank verwendet werden. So wird aus<nachname> das Element <NACHNAME> und aus <vorname> das Element <VORNAME>. Der XPath für beide Elemente sieht nun wie folgt aus:

```
1 .//MDB[.//VORNAME/text()='André' and .//NACHNAME/text()='Hahn']
```

Dieser Pfad nutzt somit die beiden bereits bekannten Merkmale, um den der aktuellen Person entsprechenden <MDB>-Eintrag in der Stammdatenbank eindeutig zu identifizieren. Wenn mit dem XPath genau ein Eintrag identifiziert wurde, wird dem gesuchten Merkmal der Wert zugewiesen, der in der Stammdatenbank im ausgewählten <MDB>-Eintrag hinterlegt ist.

Wird mittels des erstellten XPaths mehr als ein Eintrag identifiziert, kann das gesuchte Merkmal nicht gesetzt werden. Damit jedoch immer so viele Merkmale wie möglich nachträglich identifiziert werden, wird die Hilfsfunktion mit jeder möglichen Kombination aus den Merkmalen aufgerufen. Hierfür wird aus der Liste *['vorname', 'nachname', 'fraktion', 'ortszusatz', 'partei', 'wahlperiode']* jede erdenkliche Merkmalkombination in der Länge eins bis fünf erzeugt. Eine Merkmalkombination mit der Länge eins wä-

re zum Beispiel *['vorname']*, eine Merkmalkombination mit der Länge drei wäre zum Beispiel *['vorname', 'nachname', 'fraktion']* und eine Merkmalkombination mit der Länge vier wäre zum Beispiel *['nachname', 'ortszusatz', 'partei', 'wahlperiode']* etc. Diese Kombinationen werden der Funktion nach und nach übergeben, bis mit einer ein XPath erstellt werden kann, der eindeutig einen <MDB>-Eintrag identifiziert. Dieser Prozess wird während der automatischen Auszeichnung iterativ durchgeführt, bis mindestens die Merkmale „vorname“, „nachname“, „fraktion“ und „partei“ ermittelt sind. Ist dies der Fall, wird im *dictionary* des aktuellen Redners der Wert des Schlüssels „feature_complete“ auf „True“ gesetzt und die Merkmalsuche beendet.

In einem letzten Schritt wird nun die ID des aktuellen Redners ermittelt. Hierfür wird die Liste *['vorname', 'nachname', 'partei', 'wahlperiode']* an die Funktion *add_missing_MdB_feature* übergeben, mit der das Merkmal „id“ ermittelt werden soll. Mittels der Kombination wird ein XPath mit den bestmöglichen Informationen erstellt, der die höchste Chance hat einen <MDB>-Eintrag eindeutig und richtig zu identifizieren, so dass die ID für den Redner gesetzt werden kann.

Nachdem in diesem letzten Schritt das *dictionary* des Redners um die dazugehörige ID ergänzt wurde, ist dieser Redner erfolgreich erkannt worden. Es liegen alle notwendigen Information vor, so dass die offizielle XML-Auszeichnung erstellt werden kann. Das aktuelle *dictionary* wird anschließend abgespeichert, damit das Programm im späteren Verlauf überprüfen kann, ob die noch zu identifizierenden Redner-Strings nicht bereits bekannt sind. Taucht zum Beispiel wieder ein Rednerelement (<redner>) auf, welches als Text den selben Redner-String enthält, wie er in Quellcode 4.9 zu sehen ist, kann der gesamte obige beschriebene Prozess übersprungen werden, da für diesen bereits alle nötigen Informationen ermittelt und abgespeichert wurden. Die Informationen von bereits einmal erfolgreich identifizierten und ausgezeichneten Rednern wird nur für den Zeitraum einer Wahlperiode gespeichert. Erkennt die Software, dass das aktuelle Protokoll nicht mehr der gleichen Wahlperiode wie das vorherige angehört, wird der Speicher der bekannten Redner gelöscht.
Diese Vorgehensweise ist notwendig, weil Redner ihre Parteizugehörigkeit im Verlauf verschiedener Wahlperioden ändern können. So kann es zum Beispiel sein, dass ein Redner erstmals in der 14. Wahlperiode erkannt und ausgezeichnet wurde. Diese Informationen wurden dann abgespeichert. Gehört der Redner nun jedoch in der 15. Wahlperiode einer anderen Partei an, würde dieser mit falschen Informationen aus-

gezeichnet werden. Um dies zu verhindern, wird der Speicher der bereits erkannten Redner beim wechsel der Wahlperioden gelöscht. Die falsche Informationsauszeichnung betrifft vor allem das Element <damalige_fraktion>. In dieses Element wird, wenn ein Redner zum ersten mal erkannt wurde, die im Redner-String genannte Partei eingetragen. Dieses Element wird ebenfalls benötigt, da in der Stammdatenbank nur die aktuelle Parteizugehörigkeit aufgeführt ist.

Wechselt ein Redner innerhalb einer Wahlperiode die Parteizugehörigkeit, kann dies nicht erfasst werden.

4.4.3.1 Probleme und etwaige Fehler

Der oben beschriebene Prozess ist nicht gänzlich fehlerfrei, so dass der Fall eintreten kann, dass ein Redner nicht eindeutig identifiziert werden kann. Während der Entwicklung der Software sind folgende Fälle beobachtet worden, die eine erfolgreiche Auszeichnung eines Redners verhindern.

Ein einfacher Fall, der dazu führt, dass die Auszeichnung fehlschlägt, tritt ein wenn der gekennzeichnete Redner nicht in der Stammdatenbank aufgeführt ist. Dies ist unter anderem der Fall bei einigen Staatssekretären oder Gastrednern, die keine MdBs sind und es auch niemals gewesen sind.

Auch kann es sein, dass es eine Diskrepanz zwischen den in den Protokollen genannten Namen und den in der Stammdatenbank aufgeführten Namen gibt. Ein Beispiel ist hier Petra Bläss. Diese wird in den Protokollen mit folgendem String genannt: „Petra Bläss (PDS/Linke Liste):“. Allerdings ist sie in der Stammdatenbank unter dem Namen Petra Bläss-Rafajlovski gespeichert. Es ist somit nicht möglich, den Nachnamen aus dem Protokoll dem Doppelnamen aus der Stammdatenbank zuzuordnen.

Interessanterweise ist der Nachname Müller eine Problematik. Es gibt bis heute insgesamt 53 MdBs, die diesen Nachnamen führen. Hier ist es schwierig nur auf Grundlage des Nachnamens den Redner erfolgreich zu identifizieren. Oft wird diese Problematik dadurch aufgelöst, dass der Ortszusatz, die Partei und/oder die Wahlperiode als weitere Merkmale für die Suche mittels XPath verwendet werden. Es kann jedoch sein,

dass es in einer Wahlperiode innerhalb einer Partei zwei oder mehrere Personen mit dem Nachnamen Müller gibt und im Protokoll der Ortszusatz sowie Vorname nicht genannt werden. Der Redner kann somit nicht eindeutig identifiziert werden.

Eine weitere eher seltene Problematik ist es, wenn es zwei Personen gibt, die den gleichen Vor- und Nachnamen haben. Dies ist interessanterweise der Fall für Gerhard Schröder. So gibt es einen Gerhard Schröder (11002077), der im Jahr 1910 geboren wurde und den ehemaligen Bundeskanzler Gerhard Schröder (11002078), welcher 1944 geboren wurde.

Kann ein Redner nicht identifiziert werden, wird dieser zwar um die offiziellen Tags für Vor- und Nachname etc. erweitert, diese haben jedoch keinen Inhalt.

Abschließend kann festgehalten werden, dass es immer Fälle geben wird, bei der die automatische Auszeichnung nicht funktioniert, da innerhalb der Protokolle nicht genug oder inkorrekte Informationen über eine Person vorliegen.

4.4.3.2 Abweichungen von der offiziellen Auszeichnung

In Quellcode 4.10 ist die finale Auszeichnung eines Redners zu sehen. Diese unterscheidet sich in zwei Punkten von der offiziellen Auszeichnung. Zum einen ist die Auszeichnung um zusätzliche Elemente wie <damalige_fraktion> und <original_string> erweitert worden. Zum anderen ist der String, mittels dem der Redner im Protokoll eingeführt wird, Teil des Elements <redner> und nicht wie in der offiziellen Auszeichnung Teil des Elements <rede>. Diese Entscheidung wurde getroffen, da die Nennung des Redners kein Redeinhalt ist. Außerdem würde der String bei Verwendung der offiziellen Struktur bei der Berechnung der N-Gramme verwendet werden, obwohl dieser gar keine gesprochene Äußerung ist.

4.4.3.3 Erstellen und Einfügen der Rede-IDs

Nachdem die Redner ausgezeichnet wurden, wird in einem letzten Schritt die ID für die Redeelemente (<rede>) erzeugt und gesetzt. Eine Rede-ID besteht aus einer zehnstelligen Zahl, welcher der String „ID“ vorangestellt ist. Die ersten beiden Ziffern geben die Wahlperiode des Protokolls wieder. Die darauffolgenden drei Ziffern entsprechen der aktuellen Sitzungsnummer. Die nächsten drei Ziffern zeigen die fortlaufende Nummer der aktuellen Rede. Die letzten zwei Ziffern sind für etwaige Korrekturen reserviert. [5, S. 10] Ein Beispiel wäre die ID „ID1809901100“, welche wie folgt zerlegt werden kann:

ID (Vorangestellter String) | 18 (Wahlperiode 18.) | 099 (Sitzungnsummer 99) |
011 (Rede mit der Nummer 11) | 00 (Keine Korrekturen)

Jede Rede wird mit einer ID versehen. Auch das Element <sitzungsbeginn> wird mit einer ID ausgezeichnet. In diesem Punkt unterscheidet sich die automatische Auszeichnung von der offiziellen Auszeichnung.

4.4.4 Auszeichnen von Kommentaren und Absätzen

Auf die detaillierte Auszeichnung der Redner folgt die letzte Auszeichnung, bei der Kommentare, Metadaten und Absätze ausgezeichnet werden. Hierfür wird das Skript *speeches.py* verwendet, welches die Klasse *EntityMarkup* benötigt.

Ziel dieses Schritts ist es, die Auszeichnung wie sie in Quellcode 4.10 zu sehen ist, so zu erweitern, dass diese der offiziellen Auszeichnung entspricht wie sie in Quellcode 4.13 gezeigt wird.

```
406     <rede id="ID19100400">
407       <p klasse="redner"><redner
          ↪  id="11004662"><name><titel>Dr.</titel><vorname>Bernd</vorname>
408       <nachname>Baumann</nachname><fraktion>AfD</fraktion></name></redner>
409       Dr. Bernd Baumann (AfD):</p>
```

```
410     <p klasse="J_1">Herr Präsident! Meine Damen und Herren! Immer deutlicher zeigte sich im
        ↪ Verlauf dieses Jahres, dass die AfD in den Bundestag einziehen würde und dass sie auch
        ↪ den Alterspräsidenten stellen würde, weil sie neben vielen jungen Abgeordneten eben
        ↪ auch den ältesten Abgeordneten mit an Bord hatte.</p>
411     <kommentar>(Zurufe von der SPD)</kommentar>
```

Quellcode 4.13: Gekürzte offizielle neue Auszeichnung der 19. Wahlperiode (current_official_protocols_xml/19001.xml)

Mittels der Methode *class EntityMarkup.**markup_speech_lines(...)*** wird jede Textzeile einer Rede als ein Absatzelement (<p>) ausgezeichnet. Das Element <redner> wird nicht als Absatz gekennzeichnet.

In jedem Absatzelement wird das Attribut @klasse eingefügt, dessen Wert immer „J“ ist. Der Wert steht laut der offiziellen Dokumentation für normalen Text. [5, S. 11] Eine automatische Analyse des Textes, ob es sich um einen Tagesordnungspunkt oder ein Zitat handelt wird nicht durchgeführt.

Die Vorgehensweise jede Zeile als Absatz zu kennzeichnen funktioniert für die Wahlperioden 1 bis einschließlich 14 sehr gut. Bei den Protokollen dieser Perioden wurde pro Zeile immer ein Satz oder ein Absatz erfasst, so dass dieser korrekt ausgezeichnet werden kann. Ab der 15. Periode sind die Protokolle jedoch so gestaltet, dass jede Zeile auf etwa 80 Zeichen begrenzt ist. Zwar werden so mittels der automatischen Auszeichnung keine Absätze mehr ausgezeichnet, aber zumindest die Struktur und Länge der einzelnen Zeilen. Da jedoch der Großteil der Protokolle bezüglich der Absätze sinnvoll automatisch ausgezeichnet werden kann, wird diese Vorgehensweise auf alle Protokolle angewendet. Auch ist die Auszeichnung aller Textteile als Absätze notwendig, um im nächsten Schritt Kommentare besser auszeichnen zu können, die über mehrere Zeilen reichen.

Für die Auszeichnung von Kommentaren und Metadaten werden reguläre Ausdrücke aus der Datei *config.ini* geladen. Hierbei werden nur die Inhalte der beiden Sektionen *Regular expressions speeches* und *Multiline entities* verwendet.

```
26 [Regular expressions speeches]
27 comments = \B\([^\(\)]*\)\B ; kommentar
28 date_string = [\d\t ]*Deutscher Bundestag (?:–|—|-|--) \d{1,2} ?\. Wahlperiode (?:–|—|-|--) \d{1,3} ?\.
   ↪ Sitzung ?\. (?:Bonn|Berlin),
   ↪ (?:Montag|Dienstag|Mittwoch|Donnerstag|Freitag|Samstag|Sonntag), den \d{1,2} ?\.
   ↪ (?:Januar|Februar|März|April|Mai|Juni|Juli|September|Oktober|November|Dezmber) \d{4}[\d\t
   ↪ ]*|[\d\t ]*Deutscher Bundestag (?:–|—|-|--) \d{1,3}\. Sitzung ?\. (?:Bonn|Berlin),
   ↪ (?:Montag|Dienstag|Mittwoch|Donnerstag|Freitag|Samstag|Sonntag), den \d{1,2} ?\.
   ↪ (?:Januar|Februar|März|April|Mai|Juni|Juli|September|Oktober|November|Dezmber) \d{4}[\d\t ]*
   ↪ ; metadata
29
30 [Multiline entities]
31 multiline_comment = \B\([^\(\)]* ; [^\(\)]*\)\B ; kommentar
```

Quellcode 4.14: Reguläre Ausdrücke zur Identifikation von Kommentaren etc. (bundesdata_markup_nlp/config.ini)

Die in Quellcode 4.14 gezeigten Einträge enthalten als Wert eine String, der an der Zeichenfolge „ ; “ geteilt wird, so dass in der Sektion *Regular expressions speeches* der Regex vom Tag-Namen getrennt werden kann. In der Sektion *Multiline entities* handelt es sich bei den ersten beiden Teilen um jeweils einen Regex, der das Anfangsmuster und das Endmuster eines gesuchten Strings beschreibt. Zwischen Anfang und Endmuster dürfen alle Zeichen außer „(“ und „)“ vorkommen. Der dritte Teil enthält den Tag-Namen.
Die Methode *class EntityMarkup.**inject_element(...)*** nutzt die regulären Ausdrücke aus den Zeilen 27 und 28, um Vorkommen dieser in den Absätzen einer Rede zu identifizieren und dann als Kommentar auszuzeichnen.
Mit der *class EntityMarkup.**get_multiline_entities(...)*** werden die Ausdrücke aus Zeile 31 genutzt, um Kommentare zu identifizieren, die sich über mehrere <p>-Elemente erstrecken.

Neben den Kommentaren werden noch Metadaten ausgezeichnet, die als eine Art Kopfzeile innerhalb der Protokolle zu finden sind. Beispiel einer solchen Kopfzeile ist: „9318 Deutscher Bundestag – 18. Wahlperiode – 98. Sitzung. Berlin, Freitag, den 27. März 2015“.

Dritte können die entsprechenden Sektionen in der Datei *config.ini* um eigene reguläre Ausdrücke erweitern, so dass noch weitere Arten von Informationen innerhalb der Texte ausgezeichnet werden können.

4.4.5 Menschenfreundliche XML-Formatierung

Nachdem die automatische Auszeichnung durchgeführt wurde, werden in einem optionalen Schritt mittels *js-beautify* [27], die XML-Protokolle korrekt eingerückt und generell formatiert, so dass diese einfacher von Menschen gelesen werden können. Dies geschieht unter dem Gesichtspunkt der Nachnutzung, damit es für Dritte einfacher ist, sich ein Bild über die ausgezeichneten Daten zu machen.

Für diese Formatierung wird das Skript *beautify_speeches* genutzt. Dieses formatiert die Protokolle so, dass Inhaltsverzeichnis und Anlage in ihrer Struktur nicht verändert werden. Lediglich alle Kinderelemente von <sitzungsverlauf> werden verändert. Dies wird so gehandhabt, weil insbesondere der Inhaltsverzeichnis-String strukturelle Informationen enthält, die sich direkt in der Art und Weise wie der Text formatiert wurde (Zeilenumbrüche etc.) widerspiegelt. Eventuell kann in einem zukünftigen Projekt so das Inhaltsverzeichnis einfacher automatisch ausgezeichnet werden. Gleiches gilt für die Anlage.

4.5 Dauer der automatischen Auszeichnung

Die automatische Auszeichnung ist mit dem Schritt der menschenfreundlichen Formatierung (wenn dieser nicht übersprungen wird) abgeschlossen. Insgesamt dauert die Auszeichnung der Protokolle des Korpus *dev_data* 6 Stunden 40 Minuten und 20 Sekunden[1].
Der Korpus umfasst 264 Protokolle. Die durchschnittliche Auszeichnungszeit für ein Protokoll liegt somit bei circa einer Minute und 52 Sekunden. Da der Korpus pro Wahl-

[1] Die Laufzeit kann der Logdatei *dev_data_markup_bundesdata.log* entnommen werden, welche in *bundesdata_markup_nlp_data/outputs/markup/dev_data* zu finden ist.

periode nur relativ wenige Protokolle enthält, ist diese Zeit nicht repräsentativ für die Auszeichnung der gesamten Protokolle einer Wahlperiode. Je mehr Protokolle für eine Wahlperiode vorliegen, desto schneller wird die Auszeichnung, da insgesamt mehr Redner identifiziert und als bereits bekannt abgespeichert werden können.
Für die Wahlperiode 18 beträgt die Auszeichnungszeit nur 58 Minuten und 54 Sekunden. Bei 245 Protokollen ergibt dies circa eine durchschnittliche Bearbeitungszeit von 14 Sekunden.[2]

4.6 Erstellen der N-Gramme mit bundesdata_nlp_.py

Die erfolgreich ausgezeichneten Protokolle können als Grundlage für die Berechnung von N-Grammen verwendet werden. Hierfür wird das Skript *bundesdata_nlp.py* genutzt, welches ebenfalls für die Lemmatisierung und Tokenisierung der Reden zuständig ist.

4.6.1 Lemmatisierung und Tokenisierung der Reden

Bevor die ausgezeichneten Protokolle für die Berechnung der N-Gramme genutzt werden können, müssen die einzelnen Reden innerhalb dieser entweder lemmatisiert oder tokenisert werden. Dies geschieht mit den Skripten *lemmatization.py* und *tokenize.py*. Beide Skripte verwenden für diese Prozesse *spaCy*. [46]

Ziel der Lemmatisierung ist es jedes Wort einer Rede auf seine Grundform zu reduzieren. So sollen zum Beispiel die Wörter „Gesetze“ und „Gesetzes“ auf die Grundform „Gesetz“ reduziert werden, damit diese eigentlich verschiedenen Wörter zusammengefasst und auf Basis ihrer Grundform gezählt werden können. [23, S. 11, 20–26]

Bei der Lemmatisierung der Protokolle werden die XML-Dateien einzeln abgearbeitet. Das Skript ermittelt pro Protokoll per XPath alle Redeelemente so wie das Element

[2] Die Laufzeit kann der Logdatei *18_Wahlperiode_markup_bundesdata.log* entnommen werden, welche in *bundesdata_markup_nlp_data/outputs/markup/full_periods* zu finden ist.

<sitzungsbeginn>. Für jedes dieser Elemente werden dann alle <p>-Elemente ermittelt und der Text dieser Knoten zu einem String zusammengefasst. Kommentare und Metadaten werden somit nicht erfasst. Da es sich bei den einzelnen Strings um Absätze handelt, wird an der entsprechenden Stelle „\n“ für einen Zeilenumbruch eingefügt. Diese Struktur wird später für die wieder Zusammenführung von getrennten Wörtern genutzt. Zusätzlich werden noch aus den Strings alle Unterstriche („_“) entfernt, da diese in den Protokollen manchmal fälschlicherweise anstatt eines Leerzeichens gesetzt wurden. SpaCy würde so zwei Wörter als eines erkennen.

```
1   def lemmatization(files, no_stop_words=False):
2   nlp = de_core_news_sm.load()
3   config = configparser.ConfigParser()
4   config.read("config.ini")
5   output_path = config["File paths"]["nlp_output"]
6   for file_path in tqdm(sorted(files), desc="Lemmatization file status"):
7       xml = XMLProtocol()
8       xml.read_xml(file_path)
9       speeches = xml.xml_tree.xpath(".//rede | .//sitzungsbeginn")
10      for speech in speeches:
11          parts = speech.xpath(".//p")
12          tmp_list = []
13          for part in parts:
14              if(part.text is not None):
15                  tmp_list.append(re.sub(r"_", " ", str(part.text + "\n")))
16              part.getparent().remove(part)
17          new_text = "".join(tmp_list)
18          new_text = re.sub(r"(?P<wordend>[a-zßüöä])(?P<replace>\-\n)(?P<wordstart>[a-ßzäüö])",
            ↪   "\g<wordend>\g<wordstart>", new_text)
19          new_text = re.sub(r"(?P<wordend>[a-zßüöä])(?P<replace>\-\n)(?P<wordstart>[A-ZÄÜÖ])",
            ↪   "\g<wordend>-\g<wordstart>", new_text)
20          lemmatized_speech = etree.Element("rede_lemmatisiert")
21          doc = nlp(new_text)
22          if(no_stop_words is False):
23              lemmatized = " ".join([token.lemma_ for token in doc
24                              if token.pos_ != "PUNCT" and token.text != "_"])
25              filename_sufix = "_lemmatized_with_stopwords.xml"
26          elif(no_stop_words is True):
27              lemmatized = " ".join([token.lemma_ for token in doc
28                              if token.is_stop is False
29                              and token.pos_ != "PUNCT" and token.text != "_"])
30              filename_sufix = "_lemmatized_without_stopwords.xml"
31          lemmatized_speech.text = lemmatized
```

```
32            speech.append(lemmatized_speech)
33        xml.save_to_file(output_path, file_path, "lemmatized", "File paths",
34                    "nlp_lemmatized_tokenized", filename_sufix)
```

Quellcode 4.15: Code für die Lemmatisierung der Reden (nlp/lemmatization.py)

Diese Aufbereitung der Strings findet in Zeile 15 des Quellcode 4.15 statt. In Zeile 18 werden mittels eines regulären Ausdrucks über Zeilen hinweg getrennte Wörter zusammengeführt. Aus dem String „Länderfinanz-\nausgleich“ wird zum Beispiel der String „Länderfinanzausgleich“. Der reguläre Ausdruck ist für diese Funktionsweise in drei Matchgruppen aufgeteilt. Die erste Gruppe „(?P<wortende>)“ identifiziert den letzten *lower case* Buchstaben eines jeden Wortes. In Kombination mit der Gruppe „?P<replace>“ wird dann die eigentliche Stelle identifiziert, an der ein Wort getrennt wird. Die folgende letzte Gruppe „(?P<wordstart>)“ identifiziert *lower case* Buchstaben und somit den zweiten Teil eines getrennten Wortes. Mittels der Funktion *re.sub* wird dann der Match des gesamten regulären Ausdrucks durch die erste und letzte Gruppe ersetzt, so dass getrennte Wörter wieder zusammengeführt werden.
In Zeile 19 werden mit einer ähnlichen Vorgehensweise die Zeilenumbrüche bei Doppelnamen, die über mehrere Zeilen reichen, wieder entfernt, so dass diese als ein Token identifiziert werden können. Aus dem String „Sütterlin-\nWaack“ wird so wieder der String „Sütterlin-Waack“.

Nachdem die Strings aufbereitet wurden, werden die Tokens dieser mittels Lemmatisierung auf ihre Grundform reduziert. Dieser Prozess startet entweder in Zeile 22 oder 26. Bei dieser if-Abfrage wird unterschieden, ob sogenannte Stopwörter aus dem Eingabe-String gelöscht oder beibehalten werden sollen. Stopwörter sind Wörter, die in einer Sprache am häufigsten vorkommen. In der Deutschen Sprache wären dies zum Beispiel Wörter wie „ist“ oder „ein“. [23, S. 68] Neben den Stopwörtern werden auch alle Satzzeichen entfernt, da diese nicht relevant für die Berechnung der N-Gramme sind.

Die Lemmatisierung mittels *spaCy* findet auf Basis einer *lookup table* statt. Dies ist eine simple Methode, um das Lemma eines Tokens zu bestimmen. *SpaCy* nimmt lediglich den String und vergleicht diesen mit den Schlüsseln eines Python *dictionaries* und gibt bei einer Übereinstimmung das dazugehörige Lemma zurück. [25] Die

deutsche *lookup table* umfasst 355.354 Einträge. [26] Da diese Methode der Lemmatisierung weniger komplex ist, werden nicht alle Wörter erfolgreich auf ihre Grundform reduziert. Für die Berechnung der N-Gramme ist dies jedoch ausreichend, da genug Wörter zusammengefasst werden können. Zusätzlich dazu können die N-Gramme in der Webanwendung auch mittels regulärer Ausdrücke durchsucht werden. So können von Nutzerseite aus Wörter noch weiter zusammengefasst werden. Mehr zur N-Gramm-Suche und dem Ngram Viewer kann in Kapitel 5.6 gefunden werden.

Die lemmatisierten oder tokenisierten Reden werden in einer neuen XML-Datei gespeichert. Hierbei wird die vorherige Auszeichnung der Rede sowie deren Inhalt gelöscht und der neue Text mittels des neuen Elements <rede_lemmatisiert> an der entsprechenden Stelle eingefügt. Siehe Zeile 179 in Quellcode 4.16.

Die Tokenisierung der Reden verläuft analog zum oben beschriebenen Prozess. Hierbei werden die einzelnen Wörter jedoch nicht in ihre Grundform überführt. Die Textbereinigung etc. findet statt wie oben beschrieben. Die Rede wird somit nur in ihre einzelnen Wörter zerlegt, damit diese für die Berechnung der N-Gramme verwendet werden können.

```
167  <sitzungsbeginn id="ID1824500000" sitzung-start-uhrzeit="09:01" typ="Präsident">
168    <redner id="11001274">
169      <name>
170        <titel>Dr.</titel>
171        <vorname>Norbert</vorname>
172        <nachname>Lammert</nachname>
173        <damalige_fraktion />
174        <fraktion>CDU/CSU</fraktion>
175        <partei>CDU</partei>
176        <original_string>Präsident Dr. Norbert Lammert:</original_string>
177      </name>Präsident Dr. Norbert Lammert:
178    </redner>
179    <rede_lemmatisiert>Nehmen ich bitte platzen der Sitzung eröffnen lieben
180      Kollegin Kollege verehrt Gast ich begrüßen ich herzlich letzt
```

Quellcode 4.16: Gekürztes Beispiel einer lemmatisierten Rede (protocols_lemmatized_without_stopwords/18245_lemmatized_without_stopwords.xml)

4.6.2 Berechnen der N-Gramme

Aus den mittels *lemmatization.py* oder *tokenize.py* erstellten Protokollen werden die für die Webanwendung benötigten N-Gramme erzeugt. Die Berechnung dieser wird mit dem Skript *n_grams.py* durchgeführt. Das Skript berechnet für die angegebenen Protokolle 1- bis 5-Gramme.

Die N-Gramme werden konkret mittels des Paketes *scikit-learn* und der darin enthaltenen Klasse *CountVectorizer* berechnet. [17, 33]
Der initialisierten Klasse werden pro Protokoll nach und nach jeweils die Reden übergeben, so dass für diese die N-Gramme berechnet werden. Die N-Gramme werden in einer Liste gespeichert. In einem nächsten Schritt wird jedem N-Gramm das gewünschte Merkmal zugeordnet unter dem diese summiert werden sollen. Das Merkmal kann zum Beispiel das Jahr oder der Redner der Rede sein. Somit wird jedes N-Gramm-Listenelement in ein Tupel umgewandelt, dass den N-Gramm String und das zugehörige Merkmal enthält.
Da dieser Vorgang pro Rede und Protokoll durchgeführt wird, müssen die N-Gramm-Tupel in eine „globale“ Liste gespeichert werden. Die Elemente dieser Liste werden dann mittels der Klasse Counter gezählt.

Nachdem die N-Gramme gezählt wurden, werden diese sortiert, damit sie später alphabetisch pro Anfangsbuchstabe in einzelne CSV-Dateien gespeichert werden können. Für die Sortierungsreihenfolge wird in Python die *system locale* mit dem Befehl

```
1 locale.setlocale(locale.LC_COLLATE, "C")
```

auf die *C locale* gesetzt. Diese wird genutzt, da die N-Gramm-Strings so nach Byte-Wert und damit quasi nach ASCII sortiert werden. [49][29, Kpt. 7]

Durch diese Sortierung kann sichergestellt werden, das es immer die folgende Sortierungsreihenfolge der N-Gramme gibt: *[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y', 'Z', '_Non_ASCII']*

Jedem Listenelement werden die entsprechenden N-Gramme beginnend mit dem passenden Zeichen (bei Buchstaben jeweils in Groß- und Kleinschreibung) zugeordnet. Alle nicht ASCII-Zeichen werden der letzten Gruppe zugeordnet. Darunter fallen auch die deutschen Umlaute.

Nach dieser Sortierung können die N-Gramme pro Zeichen oder Gruppe von Nicht-ASCII-Symbolen in jeweils eine CSV-Datei gespeichert werden.

Der gesamte Prozess wird schrittweise für 1- bis 5-Gramme durchgeführt.

Mit dem Skript *move_ngrams.py* werden die N-Gramme in Ordner gruppiert, so dass alle 37 1-Gramm-CSV-Dateien, alle 37 2-Gramm-CSV-Dateien etc. jeweils in einem Ordner gespeichert werden.

4.6.3 Verschiedene Korpora für verschiedene N-Gramme

Mit den Skripten *lemmatization.py* und *tokenize.py* können, wie in Kapitel 4.6.2 beschrieben, verschiedene Korpora erzeugt werden. Aus diesen können somit unterschiedliche N-Gramm-Datensätze erstellt werden, welche jeweils besser oder schlechter geeignet sind, um bestimmte Anfragen zu beantworten.

So eignet sich der lemmatisierte Korpus besser dafür, wenn nicht nach einzelnen Wörtern sondern nach einer Wortgrundform gesucht wird. Sollen Wörter der selben Wortgruppe (zum Beispiel Migrant und Migrantin) im Vergleich gesehen werden, kann der tokenisierte Korpus verwendet werden.

Beide Korpora können mit oder ohne Stopwörter erzeugt werden. Korpora mit Stopwörtern können zum Beispiel hilfreich dabei sein, wenn „natürlichsprachliche“ Satzfragmente gesucht werden. Diese können leichter überlegt und formuliert werden, als eine Abfolge von Wörtern ohne Stopwörter. Ein Beispiel hierfür ist „Kampf gegen den Terror“.

Im Rahmen dieser Arbeit werden insgesamt vier Korpora erzeugt.

- lm_ns_year (lemmatized no stop words, gruppiert nach Jahr)
- tk_ws_year (tokenized with stop words, gruppiert nach Jahr)
- lm_ns_speaker (lemmatized no stop words, gruppiert nach Redner)
- tk_ws_speaker (tokenized with stop words, gruppiert nach Redner)

Pro Gruppierung gibt es somit zwei Korpora, die quasi jeweils ein Extrem (Stark zusammengefasst und bereinigt vs. unbearbeitete „natürliche“ Sprache) darstellen, welches für bestimmte Abfragen besser geeignet ist.

4.6.4 Vor- und Nachteile des N-Gramm Skriptes

Das Skript, mit dem die Berechnung der N-Gramme durchgeführt wird, weist bedingt durch die Konzipierung und Funktionsweise einige Vor- und Nachteile auf.

So kann Skript N-Gramme berechnen, die nach Jahr, Monat, Redner und Rede gruppiert sind. Auch wäre es zukünftig einfach möglich weitere Merkmale einzubauen, nach denen die N-Gramme gruppiert werden. Hierfür eigenen sich theoretisch alle Metadaten, die innerhalb eines Protokolls zu finden sind.

Diese Funktionsweise ist jedoch auch gleichzeitig ein Nachteil, da bei der Berechnung der N-Gramme immer der gesamte Korpus, also alle 4106 Protokolle, eingelesen werden muss. Dies ist ein sehr ressourcenintensiver Prozess. So wurden für die Berechnung der N-Gramme Computer verwendet, die über 32 Gigabyte Arbeitsspeicher und 32 GB Auslagerungsdateien verfügen.
Diese Problematik kann umgangen werden, wenn N-Gramme pro Jahr oder pro Monat berechnet werden. Hier kann die Berechnung pro Zeiteinheit stattfinden, da nicht nach einem Merkmal gruppiert wird, welches wie zum Beispiel ein Redner über verschiedene Zeitpunkte hinweg existiert.
Die starke Arbeitsspeicher- und Auslagerungsdateinutzung macht sich besonders bei der Berechnung von 4 und 5-Grammen bemerkbar.

Aus diesen Zeit- und Ressourcengründen liegen für diese Arbeit die Korpora somit wie folgt vor:

- lm_ns_year (1 bis 5-Gramme)
- tk_ws_year (1 bis 4-Gramme)
- lm_ns_speaker (1 bis 5-Gramme)
- tk_ws_speaker (1 bis 3-Gramme)

Allgemein kann gesagt werden, dass die Daten der 4- und 5-Gramme weniger aussagekräftig sind als die der 1- bis 3-Gramme, da es natürlich sehr viel mehr unterschiedliche 4- bis 5-Gramme gibt. Ein Wiederauftreten einzelner 4- bis 5-Gramme wird somit weniger wahrscheinlich. Eventuell ist der Korpus der Reden auch zu klein, um wirkliche Effekte bei diesen sehen zu können.

5 Darstellung der N-Gramme, Protokolle und Redebeiträge in einer Webanwendung

Die ausgezeichneten Reden und daraus erstellten N-Gramme sind die Datengrundlage für die Webanwendung, welche als zweiter Teil dieser Arbeit entwickelt wurde. Die Webanwendung bietet verschiedene Tools an, mit deren Hilfe die Protokolle und deren Inhalte durchsucht werden können. Ebenfalls können die N-Gramme abgefragt und in einer interaktiven Grafik dargestellt werden.

Die nachfolgenden Kapitel erläutern die Struktur und Funktionsweise der Webanwendung, welche sich aus verschiedenen sogenannten „Apps“ zusammensetzt. Der Begriff und das Konzept der Application (App) ist auf das Webframework *django* zurückzuführen, welches für die Entwicklung der Webanwendung verwendet wurde. In Abbildung 5.1 ist die Struktur der Anwendung und der einzelnen Apps dargestellt. Wie der Quellcode der Webanwendung heruntergeladen werden kann, ist in Kapitel 8 beschrieben.

Die Ordner *blog, speakers, speeches ngram_viewer* sind jeweils eigene Apps, die eine bestimmte Funktion der Webanwendung bereitstellen. In *bundesata_app* sind die globalen Einstellungen der gesamten Anwendung zu finden.
Die Dateien *docker-compose.yml, Dockerfile und requirements.txt* werden für die Installation und das Starten der Webanwendung benötigt. In *utils* sind Skripts für kleinere Aufgaben zu finden.

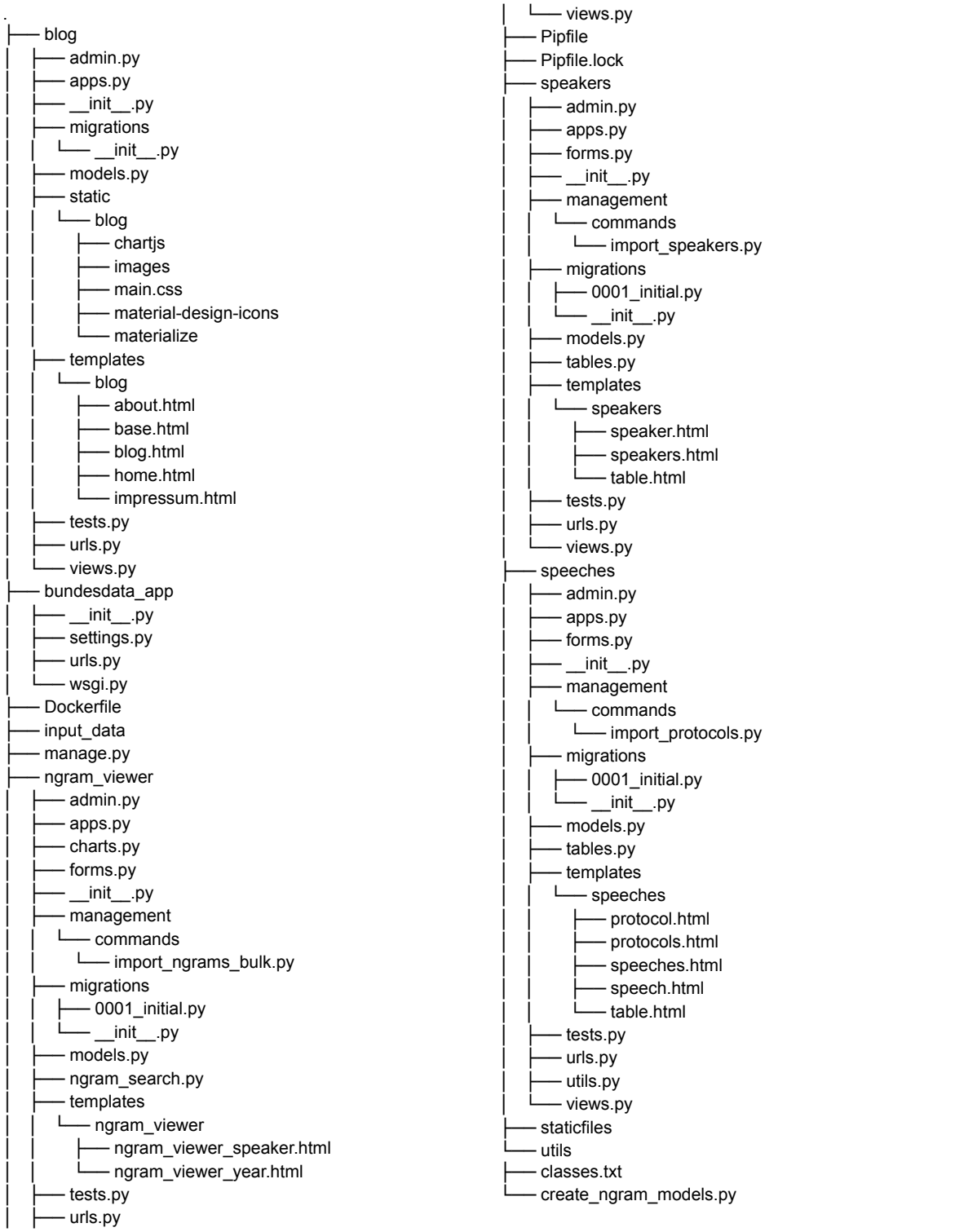

Abbildung 5.1: Strukturübersicht: bundesdata_markup_nlp

5.1 Verwendete Software und Pakete

Die Webanwendung wurde mit dem Webframework *django* in der Version 2.1.4 [10] sowie dem CSS-Framework *Materialize* in der Version 1.0.0 [30] entwickelt und verwendet folgende Pakete:

- gunicorn 19.9.0 [6]
- lxml 4.2.5 [2]
- tqdm 4.28.1 [50]
- psycopg2 2.7.6.1 [9]
- django-watson 1.5.2 [19]
- django-tables2 2.0.3 [1]
- django-jchart 0.4.2 [20]

Die App wird mittels Containervirtualisierung entwickelt und betrieben. Hierfür wird *Docker* in der Version 18.09.1-ce [15] sowie *docker-compose* in der Version 1.23.2 [16] genutzt.
Django benötigt eine Datenbank, welche die darzustellenden Inhalte speichert und dynamisch bereitstellt. Als Datenbanklösung wird *PostgreSQL* 11.2 mittels eines offiziellen *Docker Images* verwendet. [38, 40]
Neben dem *PostgreSQL Docker Image* wird noch das offizielle *Python Docker Image* in der Version 3.7.2 verwendet. [42]
Als Webserver wird *nginx* in der Version 1.15.8 mittels des offiziellen *Docker Images* verwendet. [34]

Entwickelt und getestet wurde die Webanwendung auf Debian und Arch basierenden Linux-Distributionen. Die Webanwendung kann aber auch auf macOS genutzt werden. Eine Nutzung unter Windows ist nicht getestet worden.

5.2 Entwicklung und Nutzung der Webanwendung mittels Containervirtualisierung

Die Webanwendung besteht insgesamt aus drei verschiedenen Containern, die jeweils eine Funktion erfüllen. Als Webserver wird *nginx* in genutzt, welcher dem Nutzer die statischen Inhalte der Webanwendung bereitstellt. Die *django*-Webanwendung sowie die *PostgreSQL*-Datenbank werden ebenfalls in eigenen Containern betrieben. Alle drei Container werden mittels *docker-compose* und der dazugehörigen Datei *docker-compose.yml* konfiguriert, erstellt und anschließend gestartet. Innerhalb des *django-containers* wird mit *gunicorn* noch ein Web Server Gateway Interface (WSGI) betrieben, welches die HTTP-Anfragen verwaltet. Nachdem alle drei Container erfolgreich gestartet wurden, können diese untereinander kommunizieren, so dass der Container der Webanwendung Daten aus der Datenbank abfragen kann und diese mittels des Webservers für den Nutzer dargestellt werden. Für diese Methode der Einrichtung wurden verschiedene Anleitungen und Beispiele verwendet. [21, 22, 31]

Die Entwicklung der Webanwendung wurde in der beschriebenen Containervisualisierungsumgebung durchgeführt. In der Anfangsphase wurde der *django* interne Webserver verwendet, welcher im späteren Verlauf durch *nginx* ersetzt wurde.

5.3 Aufbau der Webanwendung

Insgesamt besteht die Webanwendung aus vier Apps, welche die jeweils verschiedenen Funktionen der Webanwendung realisieren.

Die App *blog* stellt hauptsächlich HTML-Templates und statische Dateien (CSS, JavaScript und Bilder) bereit, die von den anderen Apps genutzt werden, um die Inhalte einheitlich darzustellen. Ebenfalls ist die App dafür verantwortlich statische Seiten, wie die Homepage oder das Impressum, zu erzeugen.
Mit der App *speakers* werden die Profilseiten sowie die Übersichtsliste aller MdBs erzeugt.

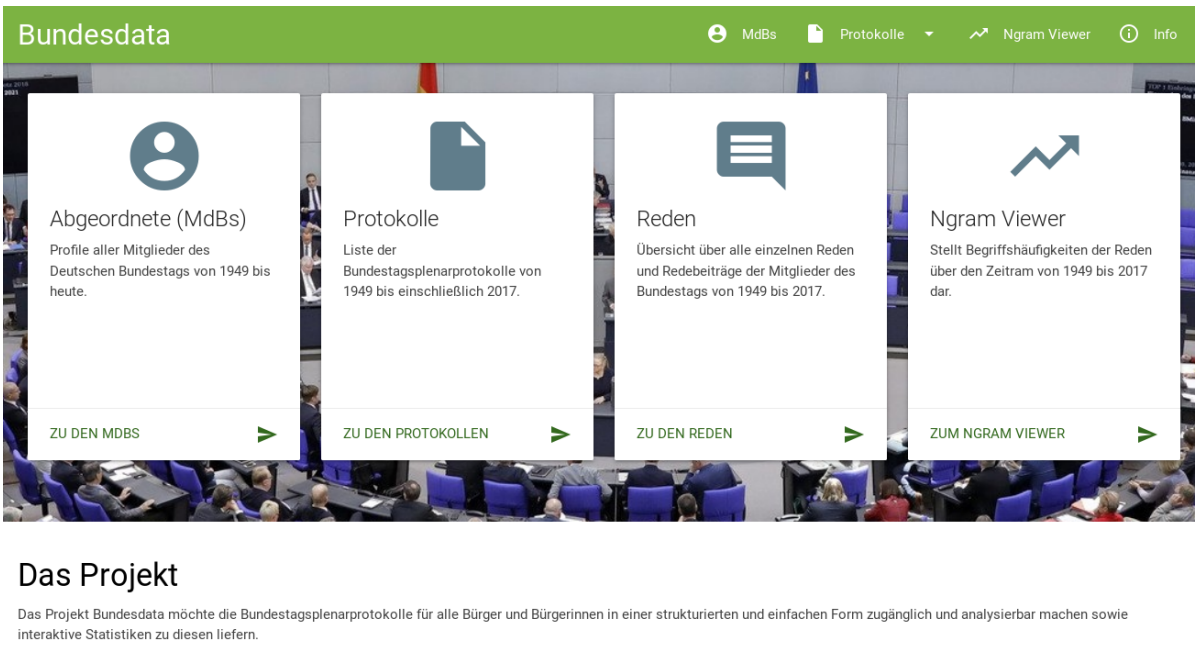

Abbildung 5.2: Hompage der Webanwendung

Die Profilseiten enthalten pro MdB jeweils alle Reden und Redebeiträge, die dieses gehalten hat. Die Darstellung der einzelne Reden und eine Übersichtsliste all dieser wird von der App *speeches* gewährleistet. Die App ist ebenfalls für die Darstellung der gesamten Protokolle so wie der dazugehörigen Übersichtsliste verantwortlich.
Die vierte App *ngram_viewer* implementiert den Ngram Viewer, welcher die erzeugten N-Gramm-Daten nutzt und für Nutzer in einer Grafik darstellt.

Die drei verschiedenen Funktionen der Apps *speakers, speeches und ngram_viewer* spiegeln sich im Aufbau der Webanwendung wieder. Auf der Homepage (Abbildung 5.2) sind alle drei Funktionen sichtbar und schnell zugänglich.

5.4 Liste der MdBs und Profilseiten

Die App *speakers* realisiert neben der Übersichtsliste aller MdBs (Abbildung 5.3) auch deren Profilseiten. Ein Beispiel einer Profilseite ist in Abbildung 5.4 zu sehen. Die Übersichtsliste wird als Tabelle dargestellt, welche mit dem Paket *django-tables2* erzeugt wird. Die Liste kann sortiert und durchsucht werden. Die Suchfunktion wird mit dem

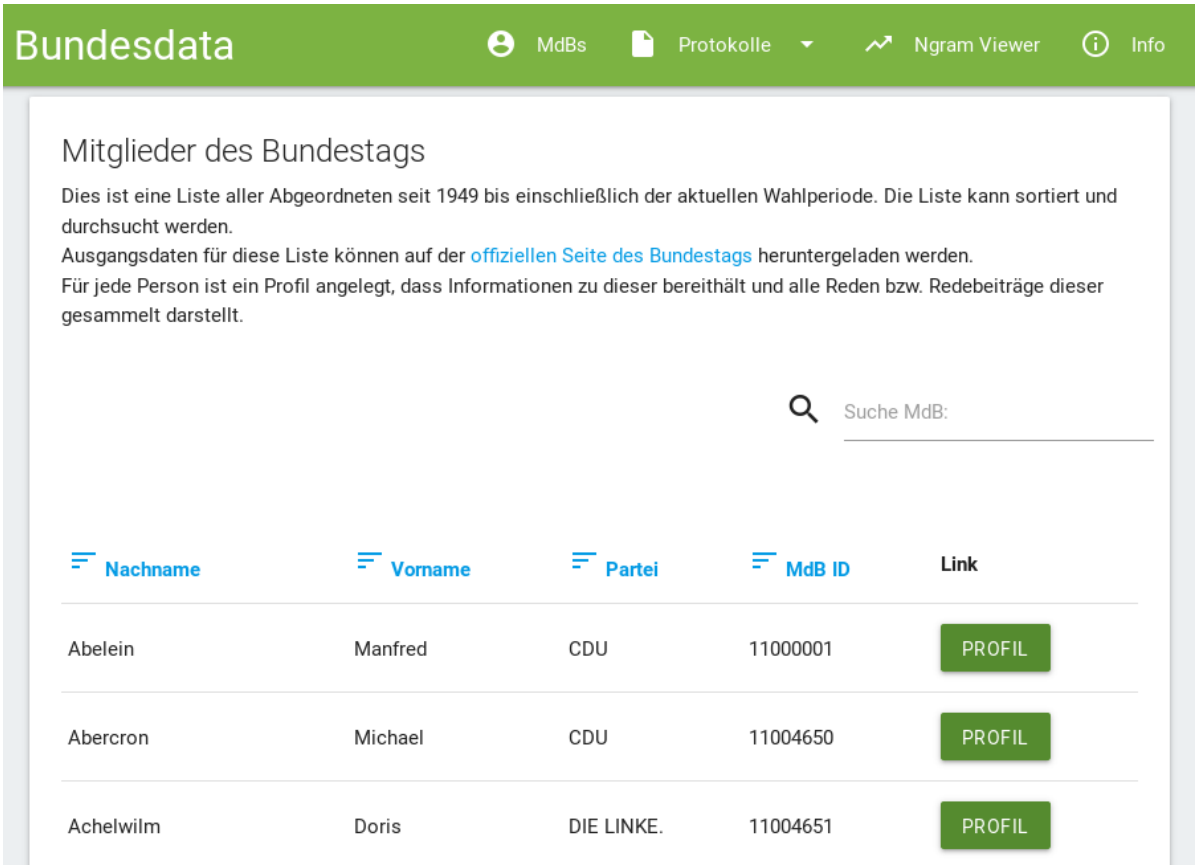

Abbildung 5.3: Übersichtsliste aller MdBs

Paket *django-watson* verwirklicht.

Auf der Profilseite sind neben Informationen über die Person alle Reden und Redebeiträge aufgeführt, die diese im Zeitraum ihrer politischen Laufbahn gehalten hat. Grundlage für diese Verknüpfung von Rednern und Reden sind die *models*. Ein *model* in django beschreibt Informationen, die über die Daten vorliegen. Jedes *model* ist eine eigene Python-Klasse, welche mittels Attributen Felder eines Datenbankeintrags beschreibt. Ein *model* entspricht generell einer *table* in einer relationalen Datenbank. Ein *model* definiert somit wie genau ein Eintrag einer Entität in einer *table* einer Datenbank aussehen muss. Es kann genau beschrieben werden, welche Felder welche Werte enthalten dürfen. Die durch *models* erstellten *Tabels* können dann mittels der django internen Datenbank-Programmierschnittstelle (API) abgefragt werden. Die API wiederum verwendet gängige SQL-Befehle. [11]
Die Struktur der eigenen definierten *models* kann in Abbildung 5.5 gesehen werden.

Einem MdB (*Speaker*) können 0 bis beliebig viele Reden (*Speech*) zugeordnet werden. Jede dieser Reden kann nur einem Protokoll (*Protocol*) zugeordnet werden. Natürlich können mehrere Reden dem gleichen Protokoll zugeordnet werden. Zusätzlich können einem Redner noch 0 bis beliebig viele legislative Institutionen (*LegislativeInstitution*) und die dazugehörigen Informationen (*LegislativeInfo*) zugeordnet werden.

Die Daten für das *model Speaker* werden mittels der eigenen geschriebenen django-admin Funktion *import_speakers.py* eingelesen und in die PostgreSQL-Datenbank gespeichert. [12] Mittels dieser Funktion werden die benötigten Daten aus der offiziellen Stammdatenbank eingelesen. Da die Stammdatenbank alle Redner der aktuellen 19. Wahlperiode enthält, gibt es MdBs, denen keine Reden zugeordnet werden.

5.5 Durchsuchen und Darstellen der Reden sowie Protokolle

Wie in Abbildung 5.4 zu sehen ist, kann von der Profilseite aus jede Rede eines Redners aufgerufen werden. Die Darstellung dieser Reden wird von der App *speeches*

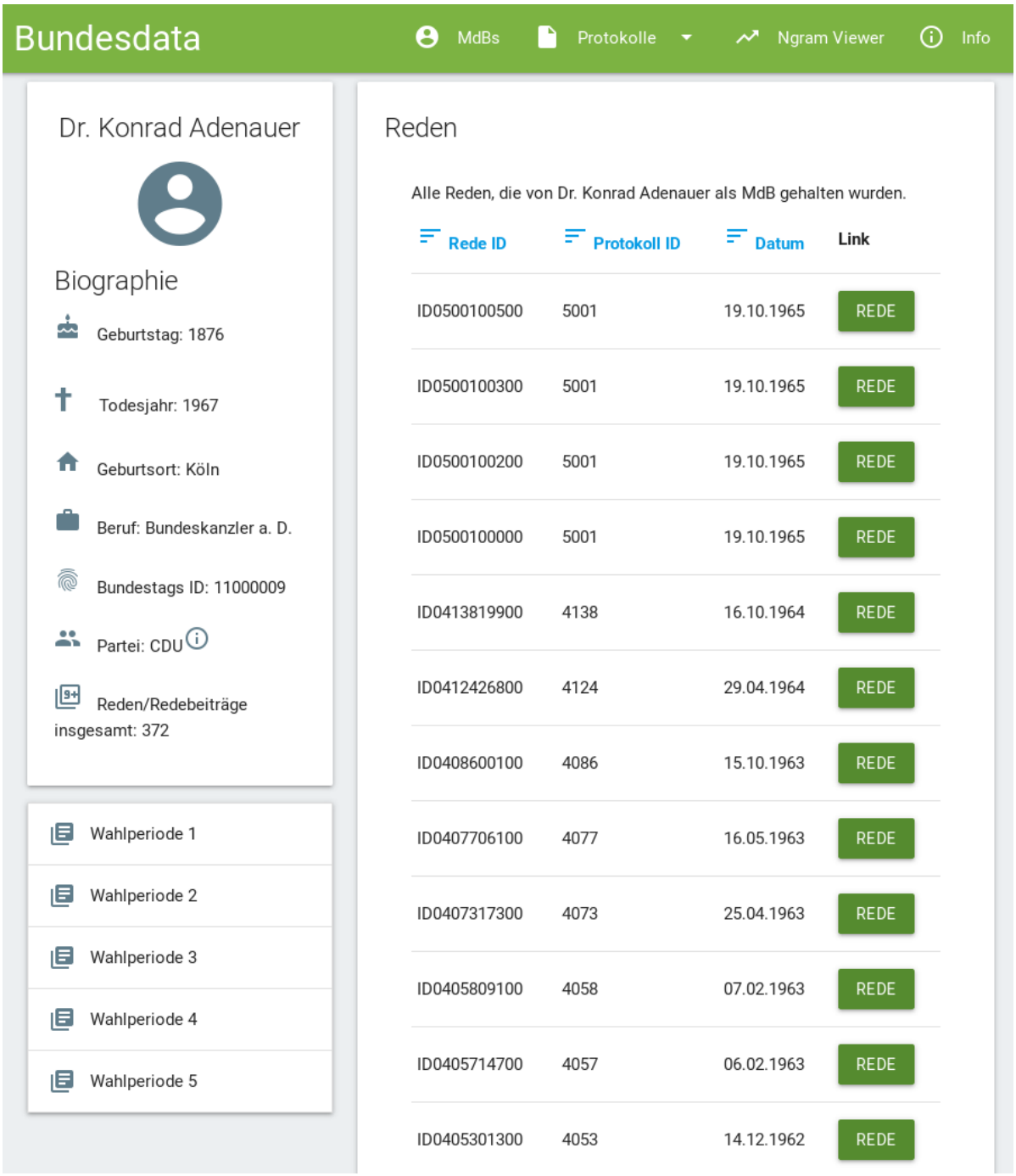

Abbildung 5.4: Profilseite eines MdB

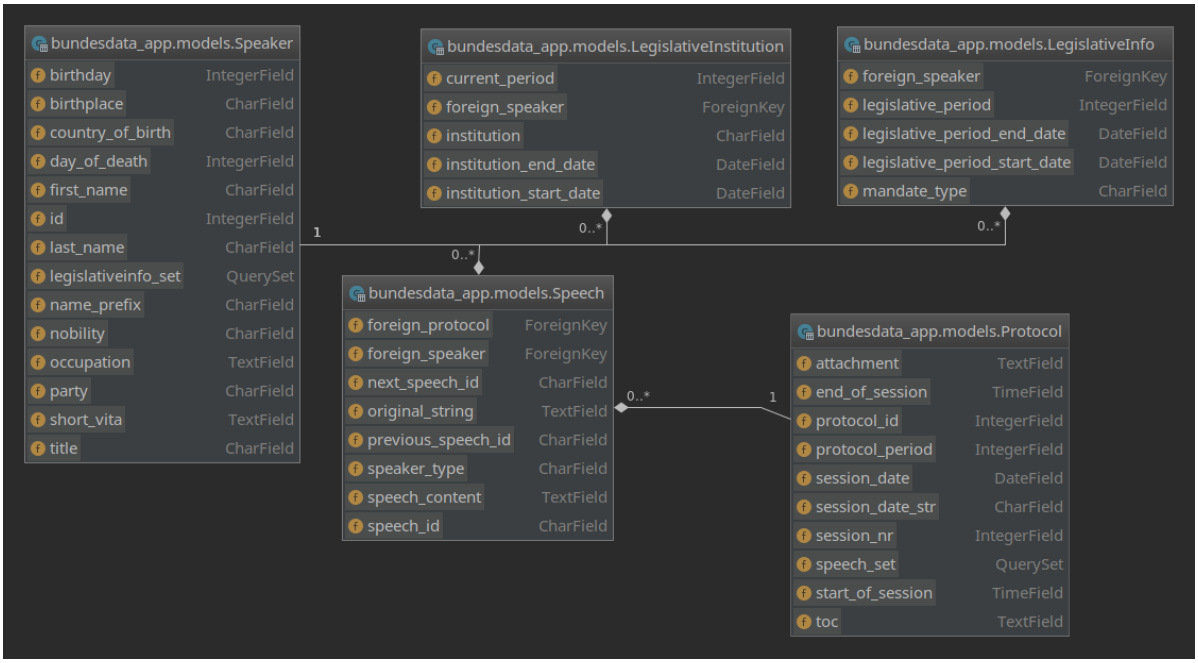

Abbildung 5.5: Übersicht der django *models* für Redner und Reden

implementiert.

Abbildung 5.6 zeigt die Darstellung einer Rede, die Konrad Adenauer am 19. Oktober 1965 gehalten hat. Auf der Seite sind neben dem Inhalt der Rede noch weitere Metadaten wie die ID, das Datum oder die Länge gezeigt. Der Eintrag „Original String“ enthält die Zeichenkette, mit der der Redner der aktuellen Rede am Anfang dieser innerhalb des Protokolls eingeführt wurde. Mit diesem String können Nutzer gegebenenfalls sehen, falls die Zuordnung der Rede zu einem Redner aufgrund einer fehlerhaften Auszeichnung nicht korrekt ist.

Die für die Darstellung der Reden benötigten Daten werden mittels der *django-admin* Funktion *import_protocols.py* eingelesen und in die *PostgreSQL-Datenbank* gespeichert. Die Funktion nimmt die automatisch ausgezeichneten Protokolle entgegen und speichert die Metadaten eines jeden Protokolls als einen Eintrag in der *table Protocol*. Diesem Protokoll werden dann alle entsprechenden Reden zugeordnet. Hierfür wird jede Rede als ein Eintrag in der *table Speech* gespeichert. Mit den Fremdschlüsseln *foreign_speech* und *foreign_speaker* wird jeder Rede genau ein Redner und genau ein Protokoll zugeordnet. Mit den Feldern *next_speech_id previous_speech_id* können einer Rede jeweils der vorherige und nachfolgende Redebeitrag zugeordnet wer-

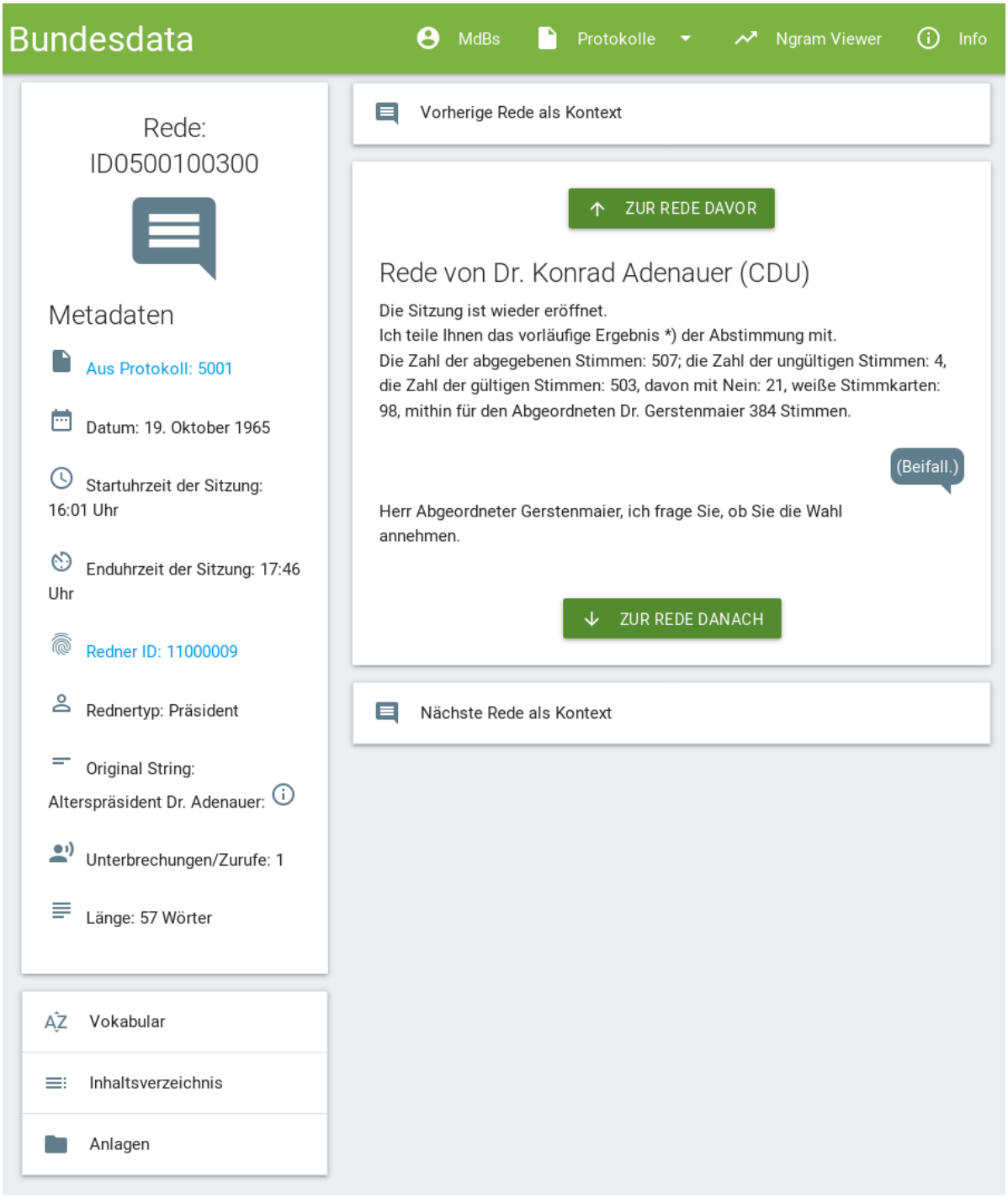

Abbildung 5.6: Darstellung einer Rede eines MdB

den.
Das Feld *speech_content* enthält den eigentlichen Redetext nicht als reinen Text, sondern als XML-String. Dieser enthält somit die Informationen darüber an welcher Stelle Kommentare oder Zwischenrufe in der Rede auftreten. Würden Redeinhalt und Kommentare getrennt gespeichert werden, könnten diese nicht mehr an der richtigen Stelle in den Redetext eingefügt werden.
Wird eine Rede für die Darstellung abgerufen, wird der ausgelesene XML-String mittels der Funktion *create_html_speech.py* in HTML umgewandelt und auf der angeforderten Seite dargestellt. Hierbei werden Textinhalt und Kommentare mit verschiedenen Klassen versehen, um diese mit unterschiedlichen CSS-Regeln visuell voneinander getrennt darzustellen.

Neben einzelnen Reden können auch ganze Protokolle dargestellt werden. Hierfür werden die einzelnen Reden in ihrer korrekten Reihenfolge wieder zusammengefügt und auf einer Seite dargestellt.

Reden und Protokolle können in ihrer Gesamtheit auch mit zwei Übersichtslisten durchsucht werden.

5.6 Der Ngram-Viewer

Herzstück der Seite ist der Ngram Viewer, welcher mit der App *ngram_viewer* realisiert wird. Die Funktion des Ngram Viewers orientiert sich an der des Google Ngram Viewers[1], welcher die Frequenz von N-Grammen über einen Korpus verschiedenster Bücher und Zeitschriften von 1800 bis 2015 darstellt. Mit der Darstellung dieser Frequenzen können kulturelle Phänomene beobachtet oder sichtbar gemacht werden, die sich in der englischen Sprache widerspiegeln. [32]
Der eigene Ngram Viewer versucht diese Funktionsweise für den Korpus aller Bundestagsplenarprotokolle von 1949 bis 2017 bereitzustellen.

[1] Link zum Google Ngram Viewer: https://books.google.com/ngrams

5.6.1 Datenbankdesign

In einem ersten Schritt müssen die bereits berechneten N-Gramme in die PostgreSQL-Datenbank eingelesen werden. In diesem Zusammenhang wurde sich die Frage gestellt, ob eine relationale Datenbank performant genug ist. Schon alleine für die 1-Gramme des lemmatisierten Korpus ohne Stopwörter pro Jahr gibt es insgesamt 5.148.382 Millionen Zeilen an Daten. Bei den 5-Grammen liegen bereits 83.295.065 Millionen Zeilen vor.
Für die 4-Gramme des nicht lemmatisierten Korpus mit Stopwörtern pro Jahr liegen sogar 157.309.038 Millionen Zeilen an Daten vor.

Mittels des Ngram Viewers sollen Nutzer interaktiv N-Gramm abfragen können, so dass zu den zur Abfrage passenden Einträgen die Anzahl der N-Gramme pro Jahr in einem Kurvendiagramm dargestellt werden. Die Suchanfrage soll innerhalb weniger Sekunden beantwortet werden.

In der Veröffentlichung *Managing the Google Web 1T 5-gram with Relational Database* wird beschrieben, wie von Google bereitgestellte N-Gramm-Daten[2] in einer relationalen Datenbank gespeichert wurden, um diese dann abzufragen. [24] Die in der Veröffentlichung verwendete Datenmenge übersteigt in ihrem Umfang die für diese Arbeit erstellten und verwendeten Daten um ein Vielfaches. Die Anzahl der 4- und 5-Gramme übersteigt hier die Milliarden-Marke. Nichtsdestotrotz konnten Anfragen an die mit den N-Grammen befüllte Datenbank je nach Länge und Anfrageart mitunter in wenigen Sekunden beantwortet werden. Auch war die damals verwendete Hardware weniger leistungsfähig als heute. Auf Grundlage dieser Ergebnisse wurde entschieden, dass die N-Gramme in einer relationalen Datenbank abgespeichert und auch schnell genug abgefragt werden können.

Um sicherzustellen, dass die Suchanfrage so schnell wie möglich ist, wurden die N-Gramme, wie bereits in Kapitel 4.6.2 beschrieben, alphabetisch in verschiedene CSV-Dateien gespeichert. Diese können nun jeweils in ein passendes *model* und eine passende *table* eingelesen werden. In Abbildung 5.7 ist schematisch dargestellt wie viele

[2] Die in der Veröffentlichung genannten Daten werden als *Web 1T 5-gram Version 1* bezeichnet. Die Daten können unter https://catalog.ldc.upenn.edu/LDC2006T13 angefordert werden. Mehr zu den Daten unter: https://ai.googleblog.com/2006/08/all-our-n-gram-are-belong-to-you.html

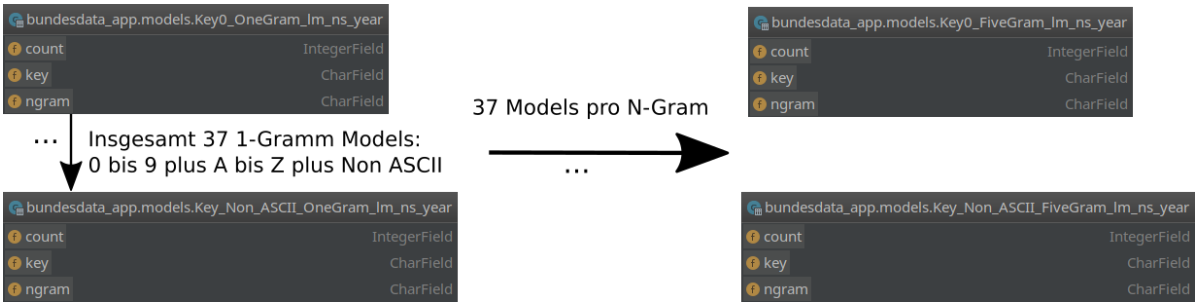

Abbildung 5.7: Schematische Darstellung der verschieden *models* für die sortierten N-Gramme

models es für den N-Gramm Korpus *lm_ns_year* gibt. Es gibt pro N-Gramm 37 *models*, welche fünf mal pro N-Gramm erstellt werden. Insgesamt werden so für einen Korpus von 1- bis 5-Grammen 185 *models* benötigt. Diese werden automatisch mit dem Skript *create_ngram_models.py* erzeugt und in die entsprechende *models.py*-Datei übertragen. Da es insgesamt vier Korpora gibt, (*lm_ns_year, lm_ns_speaker, tk_ws_year, tk_ws_speaker*) beläuft sich die Summe aller *models* auf 740.

Diese Partitionierung der Daten auf verschiedene *table* wird auch in der offiziellen Postgres-Dokumentation als eine Maßnahme genannt, um Suchanfragen zu beschleunigen. [39]

Ziel dieser Aufteilung ist es, dass bei einer Suchanfrage nach einem N-Gramm der erste Buchstabe dieser Anfrage verwendet werden kann, um festzustellen, welche *table* durchsucht werden muss.

Eingelesen werden die N-Gramme mit der *django-admin* Funktion *import_ngrams_bulk.py*. Diese erfasst stapelweise 1 Millionen Zeilen (oder einen selbst gesetzten Wert) und fügt diese mittels eines Befehls in die Datenbank ein. [13] Diese Methode ist weit aus effizienter, als jede Zeile mit einem Befehl in die Datenbank einzulesen. Nachteil dieser Methode ist jedoch, dass die mit *django-watson* implementierte Suche als Stapel eingefügte Einträge nicht automatisch indexiert und diese somit nicht durchsucht werden können. Zwar könnten diese nachträglich indexiert werden, aber dies würde den Geschwindigkeitsvorteil wieder negieren.
Alternativ können die Einträge jedoch mit der django internen QuerySet API abgefragt werden. Dieser wiederum nutzt die in Postgres verfügbaren Funktionen und Abfragemöglichkeiten. In Kapitel 5.6.2 wird genauer beschrieben, wie eine eigenen Suche

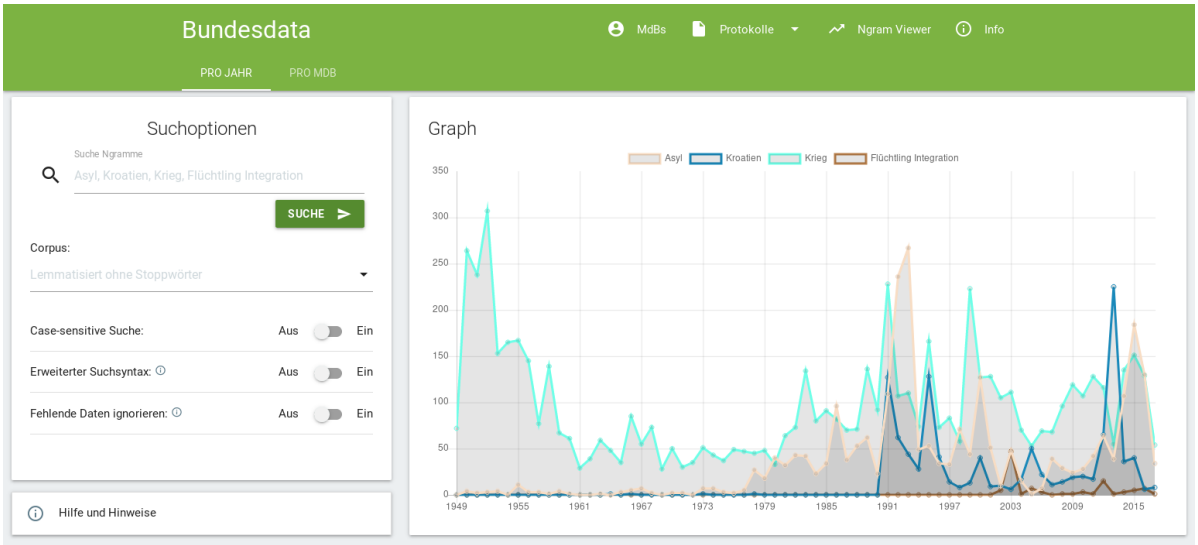

Abbildung 5.8: Ngram Viewer mit einer Suchanfrage für mehrere N-Gramme pro Jahr

entwickelt wurde, welche N-Gramm-Suchanfragen entgegennimmt und aus diesen eine interaktive Grafik erzeugt.

5.6.2 Exemplarischer Ablauf einer Suchanfrage

In Abbildung 5.8 ist der Ngram Viewer und dessen Funktionsweise zu sehen. Am linken Bildschirmrand befindet sich die Eingabe für die Suchanfrage sowie Einstellungsmöglichkeiten wie diese interpretiert beziehungsweise mit welchen Parametern diese an die Datenbank gestellt werden soll. Rechts neben diesen Bedienelemente wird die eigentliche Grafik dargestellt. In diesem konkreten Fall wird das Ergebnis der Suchanfrage „Asyl, Kroatien, Krieg, Flüchtling Integration“ dargestellt.

Ein Nutzer kann in das Suchfeld einen Zeichenkette eingeben, die durch Kommata getrennt ist. Durch das Komma werden einzelne N-Gramme voneinander abgetrennt und einzeln abgefragt. Der Beispielstring „Asyl, Kroatien, Krieg, Flüchtling Integration“ besteht somit aus vier N-Grammen: „Asyl“, „Kroatien“ und „Krieg“ als jeweils ein Unigramm und Flüchtling Integration als ein Bigramm. Nutzer können somit mehrere verschiedene N-Gramme gleichzeitig abfragen.

Konkret realisiert wird die Suche mit der Klasse *NgramSearch*, welche in der Datei

ngram_search.py zu finden ist. Bei einer Suchanfrage wird die Klasse mit den Eingabedaten des Nutzer initialisiert. In diesem Beispiel wird die Zeichenkette „Asyl, Kroatien, Krieg, Flüchtling Integration“, sowie die dazugehörigen Suchparameter (Korpusauswahl, Case-sensitivity etc.) für die Initialisierung der Klasse verwendet.

In einem ersten Schritt wird die Zeichenkette der Suchanfrage mit der Methode *class NgramSearch.**get_sub_querys()*** in die einzelnen N-Gramme zerlegt. Anschließend wird der erste Buchstabe eines jeden N-Gramms identifiziert und ermittelt, ob es sich bei dem N-Gramm um ein Unigramm, Bigramm, Trigramm etc. handelt. Mit der zusätzlichen Information, welcher Korpus durchsucht werden soll, wird dann ermittelt, welche *tables* nach welchen N-Grammen durchsucht werden sollen.

```
97          model = "Key{}_{}Gram_{}".format(sort_key,
98                                           main_class,
99                                           self.corpus_choice)
100         model = globals()[model]
101         sub_querys_dict[model].append(sub_query)
102     self.sub_querys_dict = sub_querys_dict
```

Quellcode 5.1: Ermitteln des django models für den Teilstring einer Suchanfrage (app/ngram_viewer/ngram_search.py)

In Zeile 97 von Quellcode 5.1 ist zu sehen wie aus den vorliegenden und extrahierten Informationen ein String erstellt wird, welcher ein bestimmtes *model* beschreibt. Für das Unigramm „Asyl“ liegen konkret die folgenden Informationen vor:

- *sort_key* ist „A“.
- *main_class* ist „One“, da die Zeichenkette eine Länge von einem Wort hat.
- *self.corpus_choice* ist „lm_ns_year“. Diese Information wurde durch das Optionsparameter im linken Bedienfeld gesetzt.

Mittels dieser drei Informationen kann dann der genaue Name des *models* erstellt werden: *KeyA_OneGram_lm_ns_year*. Diesem *model* werden dann als *dictionary*-Schlüssel in Zeile 101 jeweils die zu suchende N-Gramm-Zeichenkette zugeordnet. Dieser Schritt wird für jedes in der Suchanfrage übergebene N-Gramm wiederholt.

Sind alle N-Gramm-Zeichenketten abgearbeitet, wird in Zeile 102 das gesamte *dictionary* in *self.sub_querys_dict* gespeichert.

```
1 {defaultdict(<class 'list'>, {
2 <class 'ngram_viewer.models.KeyA_OneGram_lm_ns_year'>: ['Asyl'],
3 <class 'ngram_viewer.models.KeyK_OneGram_lm_ns_year'>: ['Kroatien', 'Krieg'],
4 <class 'ngram_viewer.models.KeyF_TwoGram_lm_ns_year'>: ['Flüchtling
5 Integration']})}
```

Das erstellte *dictionary* wird dann von der Methode *class NgramSearch.**enhanced_search()*** genutzt, um die angegebenen *models* beziehungsweise *tabels* nach den zugeordneten N-Gramm-Strings zu durchsuchen.
Die Suche kann in zwei Modi durchgeführt werden. Der Modus wird vom Nutzer im Bedienfeld des Ngram Viewers gesetzt, in dem der Schieberegler „Erweiterter Suchsyntax“ auf „Aus“ oder „Ein“ gestellt wird. Zusätzlich dazu kann mit dem Schieberegel „Case-sensitive Suche“ zwischen der Beachtung von Groß- und Kleinschreibung gewechselt werden.
Ist die erweiterte Suchsyntax ausgestellt, wird jeder gesuchte N-Gramm-String unter Verwendung der django internen *database-abstraction API* gesucht. Für das N-Gramm „Asyl“ sieht der konkrete Abfragebefehl zum Beispiel wie folgt aus:

```
1 KeyA_OneGram_lm_ns_year.filter(ngram__exact="Asyl")
```

Mit diesem wird nur die *table KeyA_OneGram_lm_ns_year* nach exakten case-sensitiven Vorkommen von „Asyl“ durchsucht und die Einträge zurückgegeben, die exakt dieser Anfrage entsprechen. Wird Groß- und Kleinschreibung nicht beachtet, sieht der Befehl wie folgt aus:

```
1 KeyA_OneGram_lm_ns_year.filter(ngram__iexact="Asyl")
```

Ist vom Nutzer die erweiterte Suchsyntax aktiviert worden, werden die Suchanfragen als reguläre Ausdrücke interpretiert. Auch hier werden jeweils wieder Groß- und Kleinschreibung beachtet oder missachtet. Als Syntax für die regulären Ausdrücke wird die PostgresSQL interne Syntax verwendet. [14]
Bei der erweiterten Suchsyntax ist zu beachten, dass diese nur beschränkt genutzt werden kann. Da für die Suche immer der erste Buchstabe eines Wortes identifiziert werden muss, kann die Syntax für reguläre Ausdrücke nur an das Ende einer Zeichenkette gehängt werden. Es muss immer mindestens ein Buchstabe am Anfang der N-Gramm-Suchanfrage stehen. (Beispiel: „A[\w]+“)

In Quellcode 5.2 ist zu sehen wie die interne Datenstruktur der Suchanfrage nach der Bearbeitung durch die Methode *class NgramSearch.**enhanced_search()*** aussieht.

```
defaultdict(
<class 'list'>,
  {
    <class 'ngram_viewer.models.KeyA_OneGram_lm_ns_year'>:
    [
      (<QuerySet
          [
            <KeyA_OneGram_lm_ns_year: ASYL 2011>,
            <KeyA_OneGram_lm_ns_year: Asyl 1950>,
            '...(remaining elements truncated)...'
          ]>, 'Asyl'
      )
    ],
    <class 'ngram_viewer.models.KeyK_OneGram_lm_ns_year'>:
    [
      (<QuerySet
          [
            <KeyK_OneGram_lm_ns_year: Kroatien 1964>,
            <KeyK_OneGram_lm_ns_year: Kroatien 1966>,
            <KeyK_OneGram_lm_ns_year: Kroatien 1973>,
            '...(remaining elements truncated)...'
          ]>, 'Kroatien'),
      (<QuerySet
          [
            <KeyK_OneGram_lm_ns_year: Krieg 1961>,
            <KeyK_OneGram_lm_ns_year: Krieg 1962>,
```

```
                    <KeyK_OneGram_lm_ns_year: Krieg 1963>,
                    '...(remaining elements truncated)...'
                ]>, 'Krieg')
        ],
        <class 'ngram_viewer.models.KeyF_TwoGram_lm_ns_year'>:
        [
            (<QuerySet
                [
                    <KeyF_TwoGram_lm_ns_year: Flüchtling Integration 2002>,
                    <KeyF_TwoGram_lm_ns_year: Flüchtling Integration 2003>,
                    <KeyF_TwoGram_lm_ns_year: Flüchtling Integration 2004>,
                    '...(remaining elements truncated)...'
                ]>, 'Flüchtling Integration')
        ]
    }
)
```

Quellcode 5.2: Interne Datenstruktur einer bearbeiteten Suchanfrage I

Pro *model* sind die jeweiligen *QuerySets* der gesuchten N-Gramm-Strings aufgeführt. Die Informationen werden in der Variable *self.filtered_sets_dict* gespeichert. Dieses *dictionary* wird dann von der Methode *class NgramSearch.**query_sets_to_data()*** genutzt, um aus den Objekten der *Querysets* die benötigten Daten (Anzahl der N-Gramme pro Jahr) zu extrahieren. In Quellcode 5.3 sind die gekürzten Daten zu sehen, welche aus den *QuerySets* pro N-Gramm extrahiert wurden.

```
[
    {
        'Asyl': {'1949': 0, '1950': 4, '1951': 2, '1952': 3,'...', '2015': 184, '2016': 130, '2017': 34}
    },
    {
        'Kroatien': {'1949': 0, '1950': 0, '1951': 0, '...', '2015': 40, '2016': 6, '2017': 8}
    },
    {
        'Krieg': {'1949': 72, '1950': 264, '1951': 238, '...', '2015': 151, '2016': 129, '2017': 54}
    },
    {
        'Flüchtling Integration': {'1949': 0, '1950': 0, '1951': 0, '...', 2015': 5, '2016': 7, '2017': 1}
    }
```

```
14 ]
```

Quellcode 5.3: Interne Datenstruktur einer bearbeiteten Suchanfrage II

Mit der Methode *class NgramSearch.**convert_to_data_set()*** werden diese Datensätze dann in einem letzten Schritt in JSON-Objekte umgewandelt (siehe Quellcode 5.4).

```
1  [
2     defaultdict(<class 'list'>,
3        {'Asyl': [{'y': 0, 'x': '1949'}, {'y': 4, 'x': '1950'}, {'y': 2, 'x': '1951'}, {...} {'y': 184, 'x': '2015'}, {'y': 130,
         ↪  'x': '2016'}, {'y': 34, 'x': '2017'}]}),
4     defaultdict(<class 'list'>,
5        {'Kroatien': [{'y': 0, 'x': '1949'}, {'y': 0, 'x': '1950'}, {'y': 0, 'x': '1951'}, {...}, {'y': 40, 'x': '2015'}, {'y': 6,
         ↪  'x': '2016'}, {'y': 8, 'x': '2017'}]}),
6     defaultdict(<class 'list'>,
7        {'Krieg': [{'y': 72, 'x': '1949'}, {'y': 264, 'x': '1950'}, {'y': 238, 'x': '1951'}, {...}, {'y': 151, 'x': '2015'},
         ↪  {'y': 129, 'x': '2016'}, {'y': 54, 'x': '2017'}]}),
8     defaultdict(<class 'list'>,
9        {'Flüchtling Integration': [{'y': 0, 'x': '1949'}, {'y': 0, 'x': '1950'}, {'y': 0, 'x': '1951'}, {...}, {'y': 5, 'x':
         ↪  '2015'}, {'y': 7, 'x': '2016'}, {'y': 1, 'x': '2017'}]})
10 ]
```

Quellcode 5.4: Interne Datenstruktur einer bearbeiteten Suchanfrage III

Diese Objekte werden dann als Datengrundlage für die Erstellung des Graphen genommen. Der Graph wird mit dem Paket *django-jchart* erzeugt, welches mittels Python-Code valides Chart.js-JavaScript für die Darstellung von Graphen erzeugt. [20]

In der Datei *charts.py* sind die Klassen *TimeChart* und *BarChart* definiert, welche für die Nutzung des Pakets benötigt werden. Innerhalb der Klassen werden einige Darstellungsparameter des Graphen gesetzt und definiert wie die JSON-Daten für die X- und Y-Achsen erfasst werden.

6 Evaluation der automatischen Auszeichnung und des Ngram-Viewers

Die Bundestagsplenarprotokolle können nicht gänzlich fehlerfrei automatisch ausgezeichnet werden. Hauptgrund hierfür ist die Natur der Ausgangsdaten. Die offiziellen Protokolle sind von Menschen erstellte Daten, welche Tippfehler und Inkonsistenzen enthalten. Inkonsistenzen sind vor allem in Bezug auf die Schreibweise von Rednern innerhalb der Protokolle zu finden.
Diese Probleme führen dazu, dass die für die Auszeichnung verwendeten regulären Ausdrücke nicht jeden Redner erfassen können, da zu viele Einzelfälle vorliegen. Es müssten noch weitere Ausdrücke der config-Datei hinzugefügt werden, die alle Sonderfälle, alle verschiedenen Rednerbezeichnungen und alle Tippfehler erfassen.

Neben der Berechnung der Fehlerquote für die automatische Auszeichnung wird in diesem Kapitel auch auf die Ausgabedaten des Ngram Viewers eingegangen. Anhand verschiedener exemplarischer Suchanfragen wird erläutert, wie die Ergebnisse interpretiert und für die Beantwortung verschiedener Fragen genutzt werden können. Ebenfalls soll anhand dieser Abfragen aufgezeigt werden, ob der Ngram Viewer valide und signifikante Ergebnisse liefert und die zugrundeliegenden N-Gramme nicht zu fehlerbehaftet sind.

6.1 Fehlerquote der Automatischen Auszeichnung

In der Abbildung 6.1 sind für 15 Protokolle die Fehler- und Erkennungsquoten der Software *bundesdata_markup_nlp_software* aufgeführt, welche in Kapitel 4 beschrieben wurde.
Die 15 Protokolle sind dem Datenset *development_data_xml* entnommen. Es handelt sich jeweils um das erste beziehungsweise jüngste Protokoll einer Wahlperiode. Ausnahmen bilden die Dateien *05024.xml* und *06026.xml* welche die Dateien *05004.xml* und *06003.xml* ersetzen. Die Protokolle wurden nicht verwendet, weil diese zu kurz sind.

Für die Berechnung der Erkennungsrate und der Fehlerquoten wurden in den offiziellen Protokollen händisch alle vorkommenden Redner sowie der Endzeitpunkt der jeweiligen Sitzung gezählt. Die Werte sind in den Spalten „Manuell gezählte Redner“ und „Manuell gezählte Endzeit“ zu finden. Diese Ist-Werte werden dann mit den tatsächlich automatisch erfassten Rednern und Endzeiten verglichen. Wie viele Redner automatisch mittels der regulären Ausdrücke erfasst wurden, kann der Logdatei *dev_data_markup_bundesdata.log* im Verzeichnis *bundesdata_markup_nlp_data/outputs/markup/dev_data* entnommen werden. Alternative könne auch die einzelnen XML-Elemente innerhalb der Protokolle aus dem Verzeichnis *bundesdata_markup_nlp_data-/outputs/markup/dev_data/beautiful_xml* gezählt werden. Diese Werte sind in den Spalten „Automatisch erfasste Redner“ und „Automatisch erfasste Endzeit“ zu finden. Zusätzlich wurde ermittelt, welche Redner nicht identifiziert werden konnten. Hierfür wurden alle Redner in den Protokollen gezählt, deren ID den Wert „None“ enthält. Die Werte sind in der Spalte „Automatisch nicht identifizierte Redner“ zu finden. Ebenfalls erfasst wurden *false positive* Vorkommen von Rednern. Hierbei handelt es sich um Strings, die fälschlicherweise als Redner gekennzeichnet wurden, aber keinen Redner bezeichnen.

Mit diesen Werten können pro Protokoll die Fehler- und Erkennungsraten berechnet werden. So wurden im Protokoll *04014.xml* 96,28 Prozent der Redner inklusive der Endzeit erkannt. Nicht erkannt wurden 3,72 Prozent. Von den korrekt erkannten Rednern wurden wiederum 89,22 Prozent erfolgreich und 10,78 Prozent nicht erfolgreich identifiziert. Falsch erkannte Redner gibt es in diesem Fall nicht.

Protokolle	Manuell gezählte Redner	Manuell gezählte Endzeit	Automatisch erfasste Redner	Automatisch erfasste Endzeit	Automatisch nicht identifizierte Redner	Zufällige Strings als Redner erfasst (false positive)	Redner und Endzeiten korrekt erfasst	Redner und Endzeit falsch erfasst	Redner korrekt identifiziert	Redner nicht identifiziert	False positive Quote
01025.xml	81	1	81	1	6	0	100,00 %	0,00 %	92,59 %	7,41 %	0,00 %
02006.xml	53	1	50	1	3	0	94,44 %	5,56 %	94,00 %	6,00 %	0,00 %
03007.xml	21	1	21	1	0	0	100,00 %	0,00 %	100,00 %	0,00 %	0,00 %
04014.xml	241	1	232	1	25	0	96,28 %	3,72 %	89,22 %	10,78 %	0,00 %
05024.xml	300	1	298	1	33	2	99,34 %	0,66 %	88,93 %	11,07 %	0,67 %
06026.xml	303	1	303	1	21	0	100,00 %	0,00 %	93,07 %	6,93 %	0,00 %
07031.xml	247	1	247	1	3	0	100,00 %	0,00 %	98,79 %	1,21 %	0,00 %
08025.xml	385	1	288	1	27	0	74,87 %	25,13 %	90,63 %	9,38 %	0,00 %
09012.xml	220	1	165	1	3	0	75,11 %	24,89 %	98,18 %	1,82 %	0,00 %
10014.xml	303	1	226	1	4	2	74,67 %	25,33 %	98,23 %	1,77 %	0,66 %
11005.xml	206	1	205	1	9	2	99,52 %	0,48 %	95,61 %	4,39 %	0,97 %
12025.xml	476	1	418	1	10	1	87,84 %	12,16 %	97,61 %	2,39 %	0,21 %
13008.xml	145	1	145	1	0	0	100,00 %	0,00 %	100,00 %	0,00 %	0,00 %
14063.xml	269	1	250	1	41	0	92,96 %	7,04 %	83,60 %	16,40 %	0,00 %
18004.xml	42	1	27	1	0	0	65,12 %	34,88 %	100,00 %	0,00 %	0,00 %
Durchschnitt							**90,68 %**	**9,32 %**	**94,70 %**	**5,30 %**	**0,17 %**

Abbildung 6.1: Übersicht der Fehler- und Erkennungsraten

Aus den Werten aller Protokolle können dann die Durchschnittswerte berechnet werden, mit der eine allgemeine Aussage über die Erkennungs- und Fehlerrate der Software getroffen werden kann. Hier ist anzumerken, dass theoretisch alle Protokolle des *development-Datasets* analysiert hätten werden müssten. Die Analyse all dieser Protokolle ist sehr zeitintensiv. Pro Protokoll kann die manuelle Überprüfung bis zu 30 Minuten dauern. Für alle Protokolle müssten dann 132 Stunden aufgewendet werden.

Durchschnittlich beläuft sich die Erkennungsrate der Redner auf 90,68 Prozent. Nicht erkannt werden somit 9,32 Prozent. Von den erfassten Rednern werden wiederum 94,7 Prozent erfolgreich identifiziert. Die Rate der false positive erkannten Redner ist mit 0,17 sehr niedrig.

Bei genauerer Betrachtung der Abbildung 6.1 fallen einige Ausreißer auf. So ist für Protokoll *10014.xml* die Erkennungsrate der Redner nur 74,67 Prozent. Grund hierfür ist, dass innerhalb des Protokolls eine Fragestunde stattfindet. Bei dieser kommen viele Staatssekretäre zu Wort, die mit relativ komplizierten Zeichenketten beschrieben werden. Ebenfalls interessant ist die Identifikationsrate von Rednern für Protokoll *14063.xml*. Hier wurden 16,40 Prozent der erfassten Redner nicht identifiziert. Grund für diese hohe Rate ist, dass Petra Bläss Vizepräsidentin der Sitzung gewesen ist. In der Stammdatenbank wird Frau Bläss jedoch als Bläss-Rafajlovski aufgeführt.

Abschließend kann gesagt werden, dass die allgemeine Erkennungsrate mit 90,68 Prozent sowie die allgemeine Identifikationsrate mit 94,70 Prozent annehmbar hoch ist. Allerdings gibt es wie oben beschrieben einige Sonderfälle, bei denen die Werte signifikant unter dem Durchschnitt liegen. Die Erkennungsrate der Redner kann für diese Fälle gegebenenfalls durch die Entwicklung weiterer regulärer Ausdrücke verbessert werden.
Schwieriger ist es die Identifikationsrate zu erhöhen. Da die Identifikation der Redner mittels der Stammdaten erfolgt, können keine Redner identifiziert werden, die nicht oder falsch in diesen aufgeführt sind. Ein Beispiel hierfür ist wie oben beschrieben Petra Bläss.

6.2 Fehler und Probleme bei den Ngrammen

Die im vorherigen Kapitel erläuterten Fehlerquoten haben auf die N-Gramme pro Jahr keinen signifikanten Einfluss, da nicht erkannte Redner einfach nur als Teil einer anderen Rede angesehen werden. Es findet lediglich eine geringere Verfälschung der Ergebnisse dadurch statt, dass die Zeichenkette, mit der ein Redner beschrieben wird als Teil des Textes für die Berechnung der N-Gramme angesehen wird. In diesem Zusammenhang sei auch gesagt, dass die Zuordnung der N-Gramme zum jeweiligen Jahr fehlerfrei ist. Die Datumsangabe eines jeden Protokolls war eine der wenigen Angaben, welche von der Bundesregierung mittels XML ausgezeichnet waren. Diese mussten nur in ein neues Format überführt werden.

Eine größere Auswirkung haben falsch und nicht erfasste Redner und deren Reden auf die N-Gramme, welche pro Redner oder in Zukunft z.B pro Partei berechnet werden. Hier würde eine Rede fremden Text enthalten, welcher dann für die Berechnung der N-Gramme des aktuellen Redners oder der Partei des Redners verwendet wird.

Eine allgemeine Problematik, die bei allen N-Gramm-Berechnungen auftritt, ist das Zahlen im Bereich ab Zehntausend nicht als ein Token oder Lemma erfasst werden. Grund hierfür ist, dass Zahlen wie „100.000“ in den Protokollen als „100 000“ aufgeschrieben werden.

Alle N-Gramme wurden auch für die Perioden 15., 16. und 17. berechnet, welche in Kapitel 2.1.1 als grundsätzlich fehlerhaft beschrieben wurden. Dies bedeutet, dass die N-Gramme dieser Protokolle grundsätzlich fehlerhaft sind. Allerdings werden diese in Zukunft ausgetauscht, da die Bundesregierung eine korrigierte Version dieser zur Verfügung stellt.

6.3 Evaluation der Ergebnisse des Ngram Viewers

In diesem Kapitel wird die Funktionsweise des Ngram Viewers evaluiert. Anhand verschiedener Suchanfragen wird dargestellt, welche Ergebnisse dieser liefert und wie

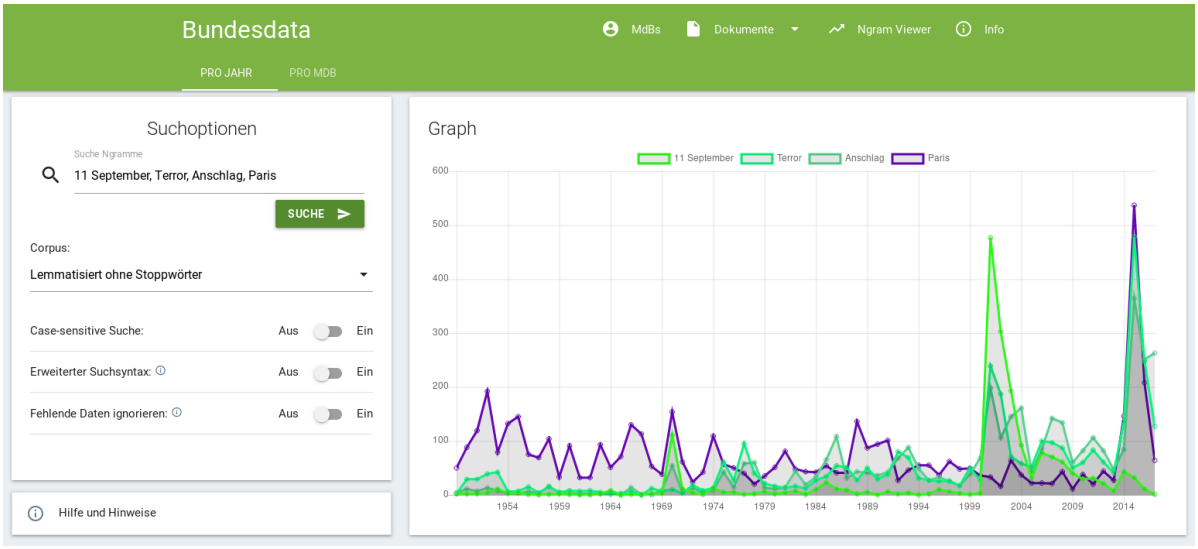

Abbildung 6.2: Ergebnis der Suchanfrage „11 September, Terror, Anschlag, Paris“

diese interpretiert und für die Beantwortung von Forschungsfragen genutzt werden können.
Im nachfolgenden Unterkapitel wird zuerst evaluiert, ob der NGram Viewer überhaupt valide und signifikante Daten ausgibt. Die weiteren Kapitel widmen sich dann komplexerer Suchanfragen und der Interpretationen dieser.

6.3.1 Identifikation geschichtlicher Ereignisse

Wird die Suchanfrage „11 September, Terror, Anschlag, Paris“ an den Ngram Viewer mit der Korpusauswahl „Lemmatisiert ohne Stopwörter“ gestellt, ist schnell ersichtlich, dass dieser valide und signifikante Daten zurückgibt.

In Abbildung 6.2 ist direkt zu erkennen, dass es in den Jahren 2001 und 2015 signifikante Ausschläge für die gesuchten Begriffe gibt. Im Jahr 2001 wurden die Terroranschläge auf das World Trade Center in New York verübt und im Jahr 2015 ereignete sich eine Serie von terroristischen Attentaten in Paris.

Ähnlich signifikante Ausschläge lassen sich bei der Suchanfrage „Atomausstieg, Fukushima, Energiewende“ (Abbildung 6.3) beobachten. Im Jahr 2011 ereignete sich in Japan die Reaktorkatastrophe von Fukushima. Hervorgerufen durch dieses Ereignis

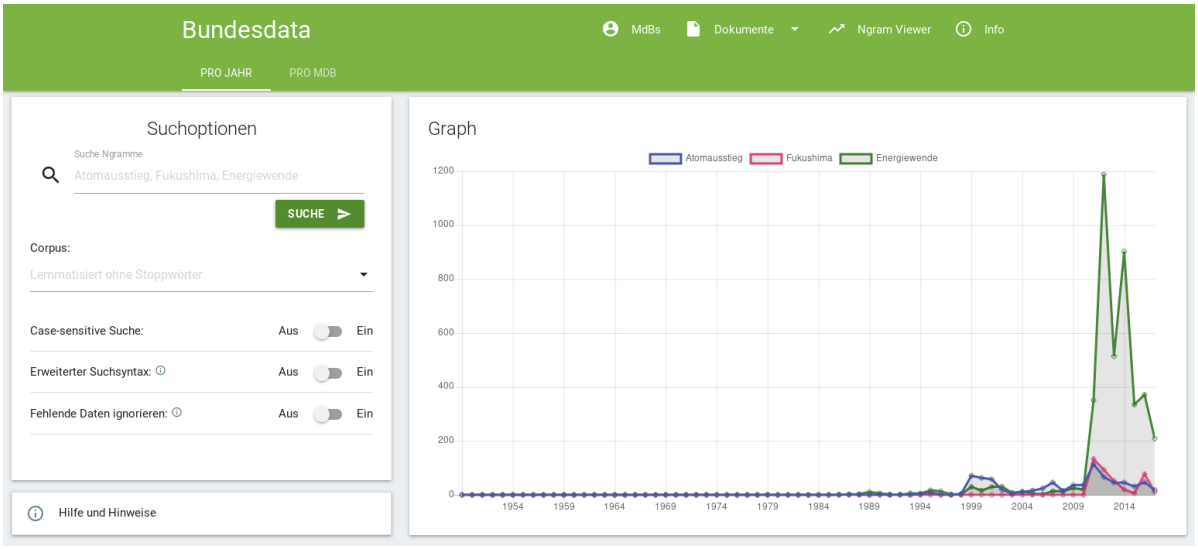

Abbildung 6.3: Ergebniss der Suchanfrage „Atomausstieg, Fukushima, Energiewende“

wurde im Bundestag wieder verstärkt über den Atomausstieg debattiert. Im Zusammenhang mit dem Begriff „Energiewende“ ist auch schnell ersichtlich, dass erst ab diesem Ereignis signifikant über selbige und damit möglicherweise einhergehende politische Maßnahmen debattiert wurde.

Abschließend kann gesagt werden, dass der Ngram Viewer valide und signifikante Ergebnisse liefert, da sich große geschichtliche Ereignisse direkt in der Sprache der Protokolle widerspiegeln und dies sichtbar gemacht werden kann.

6.3.2 Erkennen von ähnlichen Ereignissen und Diskussionen zu verschiedenen Zeitpunkten

In Abbildung 6.4 ist das Ergebnis der Suchanfrage „Krieg, Flucht, Asyl“ zu sehen.

Hier ist zu sehen, dass mit der Diskussion über Krieg oft die Diskussion über Flucht und Asyl verwoben ist. So begannen im Jahr 1991 eine Reihe von Kriegen und Konflikten in Osteuropa, die im gleichen Jahr und im Jahr darauf eine Debatte über Asyl und Flucht nach sich gezogen hat. Dieser Zusammenhang kann bis 1999 beobachtet werden, da immer wieder verschiedene Konflikte und Kriege in Osteuropa ausgetra-

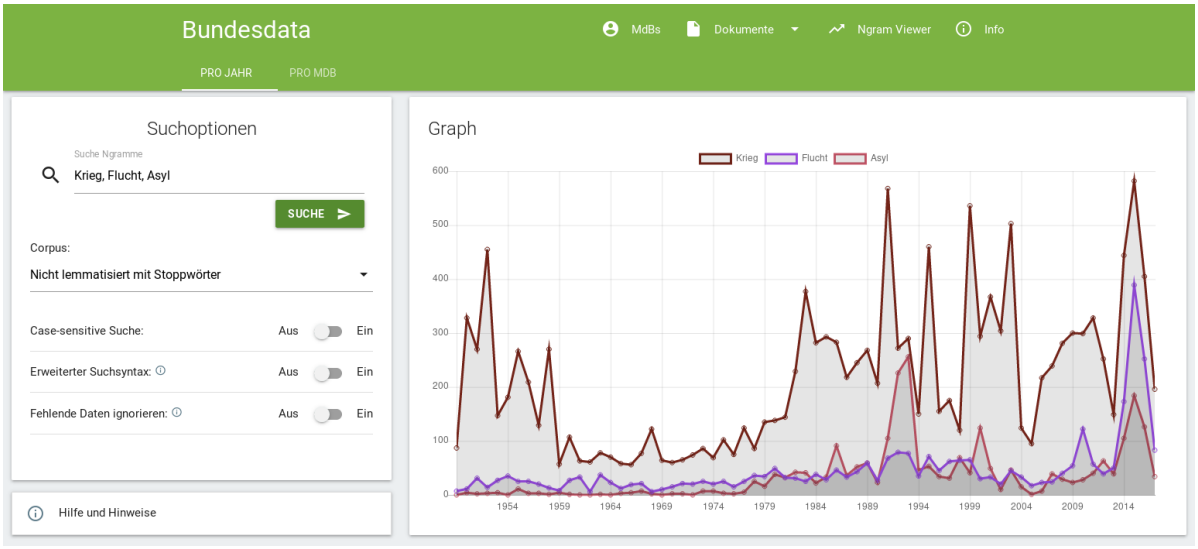

Abbildung 6.4: Ergebniss der Suchanfrage „Krieg, Flucht, Asyl“

gen wurden.
Eine ähnlicher Zusammenhang dieser Begriffe in der politischen Diskussion kann ab 2011 beobachtet werden. In diesem Jahr begann der Syrienkrieg, welcher eine Faktor für die europäische Flüchtlingskrise war.

6.3.3 Entwicklung verschiedener Begriffe und Sprachnutzung über die Zeit

In Abbildung 6.5 ist die Suchanfrage „Studenten, Studierende“ mit der Korpuswahl „Nicht lemmatisiert mit Stoppwörtern“ zu sehen. Für den Vergleich dieser beiden Begriffe wird der nicht lemmatisierte Korpus verwendet, da das Wort „Studenten“ ansonsten immer auf seine Grundform „Student“ zurückgeführt wird. Bei diesem Vergleich soll jedoch das Wort „Studenten“ in seiner Form als generisches Maskulinum betrachtet werden. Dem gegenüber steht die Partizipkonstruktion „Studierende“. Beide Wörter beschrieben demnach sowohl Studenten als auch Studentinnen.

Der Abbildung 6.5 kann entnommen werden, dass das Wort „Studierende“ bis zum Jahr 1988 wenig bis fast gar nicht im Sprachgebrauch der Redner vorgekommen ist. Im Jahr 1989 wird es insgesamt 29 mal verwendet und erreicht somit einen ersten

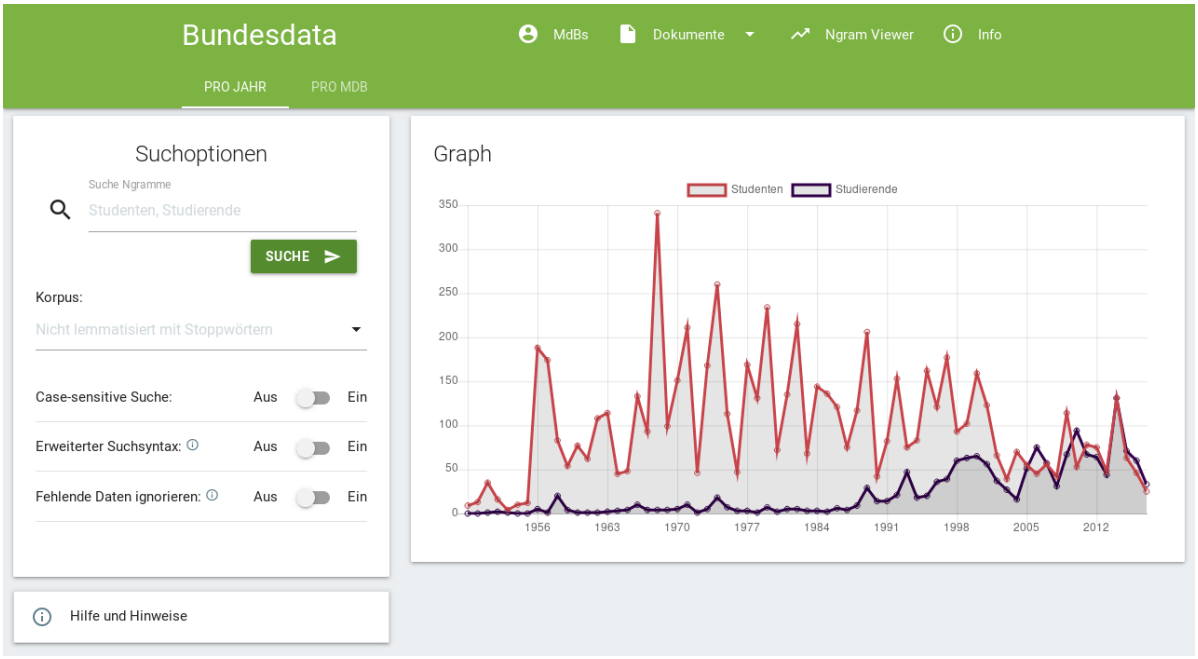

Abbildung 6.5: Ergebnis der Suchanfrage „Studenten, Studierende“

Hochpunkt. Verglichen mit den 206 Verwendungen des Wortes „Studenten“, ist es jedoch immer noch ein kleiner Ausschlag. Allerdings ist ab diesem Zeitpunkt schon zu erkennen, das die Nutzung des Wortes „Studierende“ zunimmt. Im Jahr 1993 wird das Wort „Studierende“ bereits 47 mal verwendet. Im gleichen Jahr wurde das Wort „Studenten“ 75 mal verwendet. Ab dem Jahr 2000 kann beobachtet werden, wie sich der Verlauf der beiden Kurven sich immer weiter angleicht. Im Jahr 2014 werden die beiden Worte mit 131 Nutzungen jeweils gleich viel verwendet.

Die Suchanfrage „Studenten, Studierende“ ist ein gutes Beispiel um aufzuzeigen, dass mittels des Ngram Viewers die Entwicklung verschiedener Begriffe über die Zeit dargestellt und verglichen werden kann. Dieser Vergleich lässt wiederum Rückschlüsse auf den Sprachgebrauch und die Entwicklung des selbigen zu. Mittels der Daten dieser Suchanfrage könnte die Theorie aufgestellt oder gestützt werden, dass sich der politische Sprachgebrauch in eine geschlechtsneutrale Richtung entwickelt.

7 Ausblick

Mit der Abgabe dieser Arbeit ist die Entwicklung der Webanwendung und die Erstellung der zugrundeliegenden Daten nur vorübergehend abgeschlossen. Die Daten sowie die Webanwendung können beide noch verbessert und um weitere Funktionen erweitert werden.

7.1 Ausbessern der Ergebnisse der automatischen Auszeichnung

Grundlage für die automatische Auszeichnung und die daraus resultierenden N-Gramme sind die offiziellen Protokolle der Bundesregierung. Wie in Kapitel 2.1.1 beschrieben, sind einige Wahlperioden jedoch grundsätzlich fehlerhaft abgespeichert worden.
Leider konnten die neuen korrigierten Protokolle aus zeitliche Gründen nicht mehr verwendet werden. Der erste Schritt zur Verbesserung der Daten ist somit die neuen Protokolle als Grundlage für die automatische Auszeichnung zu verwenden.

Bevor jedoch die neuen oder auch die alten fehlerfreien Protokolle erneut ausgezeichnet werden, müssten in einem zweiten Schritt weitere reguläre Ausdrücke entwickelt werden, welche die verschiedenen Redner erfassen, die zurzeit übersehen werden. Diese Ausdrücke können dank der Funktionsweise der Software einfach in die entsprechende Sektion der config-Datei *config.ini* eingefügt werden.

Die verbesserten ausgezeichneten Protokolle können dann erneut für die Berechnung der diversen N-Gramme verwendet werden.

7.2 Weitere Funktionen für die Webanwendung

Auf Grundlage der bereits vorliegenden Daten und der Struktur der Datenbank der Webanwendung kann diese einfach um weitere Funktionen erweitert werden.

Folgende Funktionen könnten in die Webanwendung implementiert werden:

- Key Word In Context (KWIC)
  - Eine Suchfunktion bei der alle Reden nach der eingegebenen Zeichenkette durchsucht und die Vorkommen dieser innerhalb des Satzes oder Textabschnittes gezeigt werden in dem diese auftreten.
- Relativer Modus für den Ngram Viewer
  - Momentan stellt der Ngram Viewer nur absolute Zahlen als Ergebnis dar. Es könnte eine zusätzliche Funktion eingebaut werden, welche die Anzahl eines gesuchten N-Gramms pro Jahr in das Verhältnis zur Anzahl der insgesamt verwendeten N-Gramme dieses Jahres setzt. Bei der Suchanfrage „Kampf gegen den Terror“ würde die jeweilige Anzahl pro Jahr dann in Relation zur Menge der insgesamt verwendeten 4-Gramme dieses Jahres gesetzt werden.
- Ngram Viewer pro Partei, pro Monat etc.
  - Das Skript zur Berechnung der N-Gramme kann so erweitert und genutzt werden, dass sich auch N-Gramme pro Partei und pro Monat berechnen lassen. Diese Daten können dann als Grundlage für einen weiteren Ngram Viewer genutzt werden, wie es zum Beispiel schon für die N-Gramme pro Redner der Fall ist.

Abseits der Implementierung größerer Funktionen können auch noch einige kleinere Fehler ausgebessert werden. So liegt zum Beispiel über jeden Redner die Information vor, welcher Partei dieser während einer Wahlperiode angehört hat. Diese wird jedoch nicht in der Webanwendung bei den entsprechenden Reden aufgeführt.

7.3 Öffentliche Version bereitstellen

Da zurzeit nur eine universitätsinterne Version der Webanwendung bereitsteht, soll in Zukunft noch eine öffentliche Version der Website online gestellt werden. Die Webanwendung ist als ein Service für Journalisten, Forscher und Bürger gedacht.

Bevor die App jedoch online gestellt wird, muss noch die Datenübertragung zwischen Server und Client mittels Hypertext Transfer Protocol Secure (HTTPS) verschlüsselt werden.

7.4 Kommende Wahlperioden

Langfristig ist es angedacht auch die Protokolle der aktuellen 19 und der kommenden Wahlperioden in die Webanwendung zu integrieren. Hierfür müsste einige kleinere Änderungen vorgenommen werden. Zum einem müsste das Skript für die Berechnung der N-Gramme leicht erweitert werden. Ebenfalls müsste das Skript zum Einlesen der Protokolle in die Datenbank erweitert werden. Beide Aufgaben sollten jedoch nicht allzu aufwendig sein, da sich die Strukturen der offiziellen und der eigenen automatischen Auszeichnung ähneln beziehungsweise die eigene Auszeichnung der offiziellen Auszeichnung nachempfunden ist.

8 Repositories mit Installations- und Bedienungsanleitungen

Insgesamt liegen für die vorliegende Arbeit drei Repositorien vor. Diese enthalten sowohl den Quellcode der Webanwendung und Auszeichnungssoftware sowie die dazugehörigen Ein- und Ausgabedaten.

Quellcode der Software für die automatische Auszeichnung der Bundestagsplenarprotokolle:

Der Quellcode der in Kapitel 4 beschrieben Software für die automatische Auszeichnung der Bundestagsplenarprotokolle ist in einem im git-Repository veröffentlicht. Das Repository kann mit dem Link https://gitlab.ub.uni-bielefeld.de/sporada/bundesdata_markup_nlp_software aufgerufen werden. In dem Repository ist eine Anleitung enthalten, welche die Installation und Verwendung der Software beschreibt.

Ein- und Ausgabedaten für die Software und die Webanwendung:

In einem zweiten Repository sind sowohl die Eingabe- als auch Ausgabedaten enthalten, die von der Software so wie der Webanwendung genutzt werden. Die Daten können unter dem Link https://gitlab.ub.uni-bielefeld.de/sporada/bundesdata_markup_nlp_data herunter geladen werden.

Quellcode der Webanwendung:

Der Quellcode für die Webanwendung, welche die erzeugten Daten verwendet, ist ebenfalls in einem Repository veröffentlicht. Das Repository kann mit dem Link https:

//gitlab.ub.uni-bielefeld.de/sporada/bundesdata_web_app aufgerufen werden. In dem Repository ist eine Anleitung enthalten, welche die Installation der Webanwendung beschreibt. Ebenfalls ist eine Live-Version der Webanwendung unter der Adresse http://129.70.12.88:8000/ zu erreichen. Diese kann jedoch nur im internen Netzwerk der Universität Bielefeld aufgerufen werden. Um die Seite von außerhalb zu erreichen, muss sich demzufolge erst per VPN in das Universitätsnetzwerk eingeloggt werden.

Literaturverzeichnis

[1] Ayers, B. *django-tables2*. Version 2.0.3. 11. Nov. 2018. URL: https://github.com/jieter/django-tables2 (besucht am 04. 02. 2019).

[2] Behnel, S. u. a. *lxml*. Version 4.2.5. 29. Juni 2018. URL: https://github.com/lxml/lxml (besucht am 29. 06. 2018).

[3] Bicking, I. *Virtualenv*. The Open Planning Project. 2018. URL: https://virtualenv.pypa.io/en/latest/ (besucht am 11. 02. 2019).

[4] Bicking, I. *virtualenv*. Version 16.0.0. 17. Mai 2018. URL: https://github.com/pypa/virtualenv (besucht am 04. 02. 2019).

[5] *Bundestags-Plenar-Protokolle im XML-Format: Aufbau der Strukturdefinition – DTD*. Deutscher Bundestag. 19. Mai 2015. URL: https://www.bundestag.de/blob/577234/f9159cee3e045cbc37dcd6de6322fcdd/dbtplenarprotokoll_kommentiert-data.pdf (besucht am 04. 02. 2019).

[6] Chesneau, B. *Gunicorn*. Version 19.9.0. 3. Juli 2018. URL: https://github.com/benoitc/gunicorn (besucht am 23. 02. 2019).

[7] *de_core_news_sm-2.0.0*. Version 2.0.0. Explosion AI. 7. Nov. 2017. URL: https://github.com/explosion/spacy-models/releases/download/de_core_news_sm-2.0.0/de_core_news_sm-2.0.0.tar.gz (besucht am 04. 02. 2019).

[8] *Deutscher Bundestag — 1. Sitzung. Bonn, Mittwoch, den 7. September 1949*. Deutscher Bundestag. 7. Sep. 1949. URL: https://dip21.bundestag.de/dip21/btp/01/01001.pdf (besucht am 05. 02. 2019).

[9] Di Gregorio, F. *psycopg2*. Version 2.7.6.1. 11. Nov. 2018. URL: https://github.com/psycopg/psycopg2 (besucht am 04. 02. 2019).

[10] *django*. Version 2.1.4. Django Software Foundation. 3. Dez. 2018. URL: https://github.com/django/django (besucht am 04. 02. 2019).

[11] *Django documentation. Models*. Django Software Foundation. 2019. URL: https://docs.djangoproject.com/en/2.1/topics/db/models/ (besucht am 23. 02. 2019).

[12] *Django documentation. Writing custom django-admin commands*. Django Software Foundation. 2019. URL: https://docs.djangoproject.com/en/2.1/howto/custom-management-commands/ (besucht am 23. 02. 2019).

[13] *Django documentation. QuerySet API reference*. Version 2.1. Django Software Foundation. 2019. URL: https://docs.djangoproject.com/en/2.1/ref/models/querysets/#bulk-create (besucht am 05. 02. 2019).

[14] *Django documentation. Making queries*. Version 2.1. Django Software Foundation. 2019. URL: https://docs.djangoproject.com/en/2.1/topics/db/queries/ (besucht am 25. 02. 2019).

[15] *docker*. Version 18.09.1-ce. Docker Inc. 9. Jan. 2019. URL: https://github.com/docker/docker-ce (besucht am 04. 02. 2019).

[16] *docker-compose*. Version 1.23.2. Docker Inc. 1. Nov. 2018. URL: https://github.com/docker/compose (besucht am 04. 02. 2019).

[17] *Documentation of scikit-learn 0.20.2*. Version 0.20.2. scikit-learn developers. 2018. Kap. sklearn.feature_extraction.text.CountVectorizer. URL: https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.CountVectorizer.html (besucht am 05. 02. 2019).

[18] *DTD für Plenarprotokolle des Deutschen Bundestags, gültig ab 19. Wahlperiode*. SRZ - Satz-Rechen-Zentrum Hartmann+Heenemann GmbH&Co. KG. 19. Mai 2015. URL: https://www.bundestag.de/blob/575720/0cae0e2f2c1854b12d571a59594ab4c2/dbtplenarprotokoll-data.dtd (besucht am 04. 02. 2019).

[19] Hall, D. *django-watson*. Version 1.5.2. 23. Feb. 2018. URL: https://github.com/etianen/django-watson (besucht am 04. 02. 2019).

[20] Heimensen, M. *django-jchart*. Version 0.4.2. 15. Okt. 2017. URL: https://github.com/matthisk/django-jchart (besucht am 04. 02. 2019).

[21] Herman, M. *Dockerizing Django with Postgres, Gunicorn, and Nginx*. 12. Nov. 2018. URL: https://testdriven.io/blog/dockerizing-django-with-postgres-gunicorn-and-nginx (besucht am 23. 02. 2019).

[22] Herman, M. *Dockerizing Django with Postgres, Gunicorn, and Nginx*. 12. Nov. 2018. URL: https://github.com/testdrivenio/django-on-docker (besucht am 23. 02. 2019).

[23] Jurafsky, D. und Martin, J. H. *Speech and Language Processing. An Introduction to Natural Language Processing,Computational Linguistics, and Speech Recognition*. 23. Sep. 2018. URL: https://web.stanford.edu/~jurafsky/slp3/ed3book.pdf (besucht am 08. 02. 2019).

[24] Lam, Y. C. *Managing the Google Web 1T 5-gram with Relational Database*. 2010. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.456.9390&rep=rep1&type=pdf (besucht am 04. 02. 2019).

[25] *Lemmatizer*. Version 2.0. Explosion AI. 2018. URL: https://spacy.io/usage/adding-languages#lemmatizer (besucht am 08. 02. 2019).

[26] *lemmatizer.py – lookup-table*. Version 2.0. Explosion AI. 10. Juli 2018. URL: https://github.com/explosion/spaCy/blob/master/spacy/lang/de/lemmatizer.py (besucht am 08. 02. 2019).

[27] Lielmanis, E. u. a. *js-beautify*. Version 1.8.9. 2018. URL: https://github.com/beautify-web/js-beautify (besucht am 08. 02. 2019).

[28] Lippe, P. von der. *Wie groß muss meine Stichprobe sein, damit sie repräsentativ ist? Wie viele Einheiten müssen befragt werden? Was heißt Repräsentativität"?* 2011. URL: http://von-der-lippe.org/dokumente/Wieviele.pdf (besucht am 05. 02. 2019).

[29] *locale — Internationalization service*. Python Software Foundation. 2019. URL: https://docs.python.org/3/library/locale.html (besucht am 19. 02. 2019).

[30] *MaterializeCSS. Materialize, a CSS Framework based on material design*. Materialize. 2018. URL: https://github.com/Dogfalo/materialize (besucht am 21. 02. 2019).

[31] T. Mazzucotelli, Hrsg. *Docker Compose with NginX, Django, Gunicorn and multiple Postgres databases*. 1. Feb. 2018. URL: https://pawamoy.github.io/2018/02/01/docker-compose-django-postgres-nginx.html (besucht am 23. 02. 2019).

[32] Michel, J.-B. u. a. „Quantitative Analysis of Culture Using Millions of Digitized Books". In: *Science* 331.6014 (Dez. 2010), S. 176–182. DOI: 10.1126/science.1199644.

[33] Mueller, A. *scikit-learn*. Version 0.20.2. 19. Dez. 2018. URL: https://github.com/scikit-learn/scikit-learn.

[34] *nginx Docker Image*. Version 1.15.8. NGINX, Inc. 2018. URL: https://hub.docker.com/_/nginx (besucht am 23. 02. 2019).

[35] *Open Data*. Deutscher Bundestag. URL: https://www.bundestag.de/service/opendata (besucht am 04. 02. 2019).

[36] *Open Data handbook – Machine readable*. Open Knowledge International. URL: http://opendatahandbook.org/glossary/en/terms/machine-readable/ (besucht am 05. 02. 2019).

[37] *Plenarprotokoll der 1. Sitzung von Dienstag, dem 24. Oktober 2017*. Deutscher Bundestag. 24. Okt. 2017. URL: https://www.bundestag.de/blob/543388/e95b7194470ed3c4e8bca546dd0da950/19001-data.xml (besucht am 06. 02. 2019).

[38] *postgres Docker Image*. Version 11.2. The PostgreSQL Global Development Group. 2019. URL: https://hub.docker.com/_/postgres/ (besucht am 04. 02. 2019).

[39] *PostgreSQL 11 Documentation*. Version 11. The PostgreSQL Global Development Group. 2018. Kap. 5.10. Table Partitioning. URL: https://www.postgresql.org/docs/11/ddl-partitioning.html (besucht am 05. 02. 2019).

[40] *PostgreSQL Database Management System*. The PostgreSQL Global Development Group. 2019. URL: https://github.com/postgres/postgres (besucht am 20. 02. 2019).

[41] *python*. Version 3.7.1. Python Software Foundation. 20. Okt. 2018. URL: https://www.python.org/downloads/release/python-371/ (besucht am 04. 02. 2019).

[42] *python – Official Docker Image*. Version 3.7.2. Python Software Foundation. 2019. URL: https://hub.docker.com/_/python (besucht am 20. 02. 2019).

[43] *Regular expression operations – Match objects*. Version 3.7+. Python Software Foundation. 2019. URL: https://docs.python.org/3.7/library/re.html#match-objects (besucht am 13. 02. 2019).

[44] *Regular expression operations – Module Contents*. Version 3.7+. Python Software Foundation. 2019. URL: https://docs.python.org/3/library/re.html#re.ASCII (besucht am 12. 02. 2019).

[45] Ronacher, A. *Babel*. Version 2.6.0. 28. Mai 2018. URL: https://github.com/python-babel/babel (besucht am 04. 02. 2019).

[46] *spacy*. Version 2.0.18. Explosion AI. 1. Dez. 2018. URL: https://github.com/explosion/spaCy (besucht am 04. 02. 2019).

[47] *spacy-models*. Explosion AI. 2019. URL: https://github.com/explosion/spacy-models/releases (besucht am 09. 02. 2019).

[48] *Stammdaten aller Abgeordneten seit 1949 im XML-Format (Stand 04.10.2018)*. Deutscher Bundestag. 4. Okt. 2018. URL: https://www.bundestag.de/blob/472878/e207ab4b38c93187c6580fc186a95f38/mdb-stammdaten-data.zip (besucht am 04. 02. 2019).

[49] *The Open Group Base Specifications Issue 7, 2018 edition. IEEE Std 1003.1-2017 (Revision of IEEE Std 1003.1-2008)*. IEEE. 2018. URL: http://pubs.opengroup.org/onlinepubs/9699919799/ (besucht am 19. 02. 2019).

[50] Yorav-Raphael, N. *tqdm*. Version 4.28.1. 21. Okt. 2018. URL: https://github.com/tqdm/tqdm (besucht am 04. 02. 2019).

E r k l ä r u n g

Hiermit erkläre ich, dass ich die vorliegende Masterarbeit selbständig verfasst und gelieferte Datensätze, Zeichnungen, Skizzen und graphische Darstellungen selbständig erstellt habe. Ich habe keine anderen Quellen als die angegebenen benutzt und habe die Stellen der Arbeit, die anderen Werken entnommen sind - einschl. verwendeter Tabellen und Abbildungen - in jedem einzelnen Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht.

Bielefeld, den

_____

(Unterschrift)